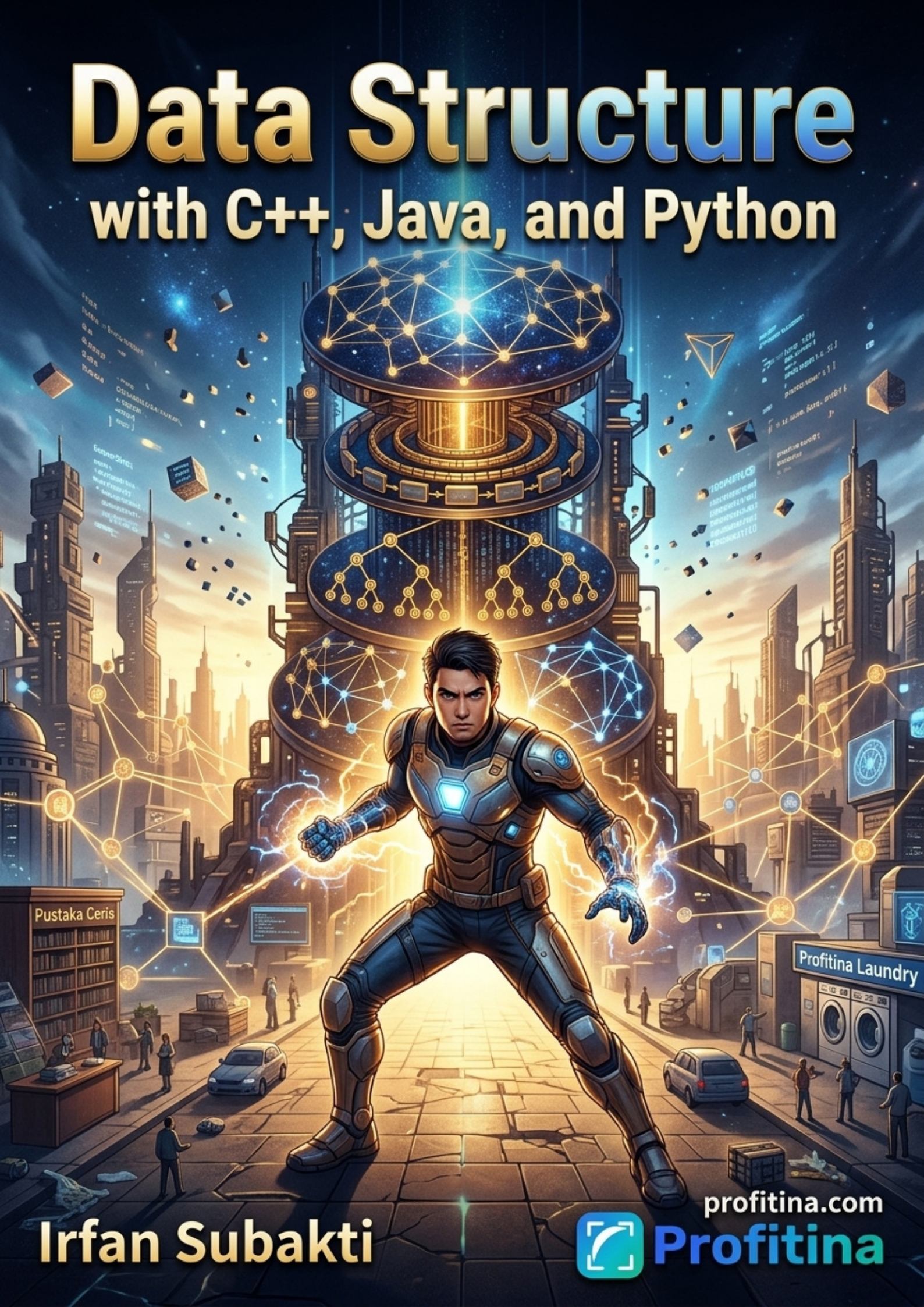
Data Structure

with C++, Java, and Python



Irfan Subakti

profitina.com
Profitina

Data Structure

with C++, Java and Python

Irfan Subakti



Copyright © 2026 Irfan Subakti. All rights reserved.

Published by Profitina · profitina.com

First edition, September 2026.

Every program in this book was compiled and run before its output was printed. The complete source code is in the accompanying Code folder. The programs are offered for teaching and learning; Irfan Subakti and the publisher accept no liability for their use elsewhere.

Contents

Preface ..... **lvii**
Who this book is for lvii
One idea, three languages lvii
What is in it lvii
How to read it lviii
About the author lix
Acknowledgments lix

CHAPTER 1 Foundations: C++ Refresher, Memory, and Arrays **1**
1.1 Where We Begin: A Library Without Shelves 1
1.1.1 Pustaka Ceria 1
1.1.2 What a data structure is 2
1.1.3 Why the choice matters 2
1.1.4 Data structures behind every web page 3
1.1.5 Six families, and the road map of this book 4
1.2 Three Languages, One Idea 5
1.2.1 Why C++ first, and why also Java and Python 5
1.2.2 How a program becomes something that runs 6
1.2.3 How the code is presented 7
1.3 How Memory Holds Data 7
1.3.1 Bits and bytes 7
1.3.2 Addresses 7
1.3.3 A type gives the bytes a meaning 8
1.3.4 Seeing it for yourself 8
1.3.5 The stack and the heap 11
1.4 A C++ Refresher for Beginners 13
1.4.1 The shape of a C++ program 13
1.4.2 Types and variables 13
1.4.3 Input and output 14
1.4.4 Making decisions and repeating 16
1.4.5 Functions 17
1.4.6 Structs: many values under one name 17
1.4.7 The address-of operator, and a first look at pointers 22
1.5 Arrays in Depth 25
1.5.1 One row of numbered slots 25
1.5.2 Why indexes start at 0 26
1.5.3 An array of records: Pustaka Ceria's catalog 26
1.5.4 Staying inside the bounds 32
1.5.5 Walking an array: the everyday traversals 34
1.5.6 Two-dimensional arrays and row-major layout 37
1.5.7 Tables of numbers 42
1.5.8 `std::array` and `std::vector` 45
1.5.9 Java arrays and `ArrayList` 47
1.5.10 Python lists and the `array` module 48
1.5.11 The three languages side by side 52
1.6 A First Taste of Complexity: $O(1)$ Access versus $O(n)$ Insertion 52
1.6.1 Counting steps instead of seconds 52
1.6.2 Access: the same cost for every index 52
1.6.3 Insertion and deletion: the cost of keeping the row tight 53
1.6.4 Big-O, informally 58

1.6.5 A glimpse of amortized cost 59
1.7 Setting Up VS Code for C++, Java and Python on Windows, macOS and Linux 59
1.7.1 Three words first 59
1.7.2 Visual Studio Code 59
1.7.3 A C++ compiler and debugger 60
1.7.4 Java: JDK 25 61
1.7.5 Python 3.14 61
1.7.6 The VS Code extensions 62
1.7.7 Checking the whole workbench 62
1.7.8 Your first program, in three languages 63
1.7.9 Running from VS Code, feeding input, and debugging 65
1.8 Profitina in Practice 65
1.9 Common Mistakes, and How to Avoid Them 66
1.10 Chapter Summary 67
1.11 Exercises 68
1.12 Solutions to the Exercises 69
References 77

CHAPTER 2 Searching 79
2.1 Where We Left Off 79
2.2 The Searching Problem 80
2.2.1 Records, keys and the found/not-found contract 80
2.2.2 How we measure the cost of a search 80
2.2.3 Pustaka Ceria's computing shelf 80
2.3 Sequential Search 81
2.3.1 The idea 81
2.3.2 From a flag to an index 82
2.3.3 The program 82
2.3.4 How much does sequential search cost? 92
2.4 Sequential Search with a Sentinel 92
2.4.1 Two tests where one would do 92
2.4.2 Counting the tests 93
2.4.3 What the sentinel does and does not buy 97
2.5 Binary Search 97
2.5.1 Using the order 97
2.5.2 A trace by hand 97
2.5.3 The procedure and its invariant 98
2.5.4 How many probes? 99
2.5.5 Binary search, recursively 100
2.5.6 The programs 100
2.6 Off-by-One Traps, Lower Bound and Upper Bound 107
2.6.1 Closed and half-open intervals 107
2.6.2 Four traps 107
2.6.3 Lower bound and upper bound 107
2.6.4 The overflowing middle 108
2.6.5 The program 109
2.7 Interpolation Search 116
2.7.1 Guessing where the key should be 116
2.7.2 The shelf, by hand 116
2.7.3 The program 117
2.7.4 Brilliant or terrible: it depends on the data 121
2.8 Jump Search 122
2.8.1 Leaping ahead in blocks 122
2.8.2 Choosing the block size 123
2.8.3 The program 123

2.9 Searching Two-Dimensional Arrays ... 127
2.9.1 How a two-dimensional array sits in memory ... 127
2.9.2 Three ways to search a grid ... 128
2.9.3 The program ... 129
2.10 Measuring on Real Data ... 137
2.10.1 A real catalog ... 137
2.10.2 The experiment ... 137
2.10.3 What the numbers say ... 151
2.10.4 How probe counts grow ... 152
2.11 Searching with the Standard Libraries ... 162
2.11.1 C++ ... 162
2.11.2 Java ... 164
2.11.3 Python ... 166
2.12 Choosing a Search Method, and the Limit of Comparison Searching ... 168
2.12.1 The whole picture ... 168
2.12.2 Why nothing that compares can beat binary search ... 168
2.13 Profitina in Practice ... 169
2.13.1 Where this chapter appears in a real product ... 169
2.13.2 The worked case study: Profitina Laundry ... 170
2.14 Common Mistakes, and How to Avoid Them ... 177
2.15 Chapter Summary ... 177
2.15.1 Looking ahead ... 178
2.16 Exercises ... 178
2.17 Solutions to the Exercises ... 180
References ... 188

CHAPTER 3 Sorting ... 190
3.1 Where We Left Off ... 190
3.2 Why Sort, and the Words We Need ... 191
3.2.1 What sorting is ... 191
3.2.2 Why it is worth the trouble ... 191
3.2.3 Six families of sorting methods ... 192
3.2.4 The one move every sort needs: exchanging two values ... 192
3.2.5 Counting the work: comparisons and swaps ... 195
3.2.6 Three properties: in-place, stable, adaptive ... 196
3.3 Bubble Sort ... 196
3.3.1 The idea ... 196
3.3.2 The early exit ... 197
3.3.3 A complete trace ... 197
3.3.4 The program ... 198
3.3.5 How much work, exactly ... 201
3.3.6 Other forms you will meet ... 202
3.4 Exchange Sort ... 203
3.5 Selection Sort ... 206
3.5.1 Search first, swap once ... 206
3.5.2 The program ... 207
3.5.3 Fewest moves, but never fewer comparisons ... 210
3.6 Insertion Sort ... 210
3.6.1 Sorting a hand of cards ... 210
3.6.2 A trap in the loop condition ... 211
3.6.3 The program ... 212
3.6.4 Why insertion sort is the best of the simple sorts ... 214
3.7 Shell Sort ... 215
3.7.1 Long jumps first ... 215
3.7.2 A trace with gaps 3 and 1 ... 215

3.7.3 The gap sequence is everything 216
3.8 Merge Sort 222
3.8.1 Divide and conquer 222
3.8.2 The trace 223
3.8.3 The program 224
3.8.4 Why merge sort is $n\lg n$, always 228
3.9 Quicksort 228
3.9.1 Partition, then recurse 228
3.9.2 Lomuto's partition 228
3.9.3 Hoare's partition 230
3.9.4 The program, and the choice of pivot 231
3.9.5 Best, worst and average 240
3.10 Heap Sort: a First Look 241
3.10.1 A tree hidden inside an array 241
3.10.2 The trace 242
3.11 Counting Sort and Radix Sort 248
3.11.1 The barrier that comparison sorts cannot cross 248
3.11.2 Counting sort 248
3.11.3 Radix sort 249
3.12 Stability, In-Place, and the Complete Table 254
3.12.1 Stability you can see 254
3.12.2 The complete table 258
3.13 Measuring the Sorts 259
3.13.1 The benchmark program 259
3.13.2 Reading the numbers 265
3.13.3 Input order matters for some sorts 267
3.13.4 The same benchmark in Java and Python 268
3.14 Library Sorts Today 279
3.14.1 C++: introsort and merge sort 280
3.14.2 Java: dual-pivot quicksort and TimSort 280
3.14.3 Python: Timsort and Powersort 281
3.14.4 Using them 281
3.15 Profitina in Practice 285
3.16 Common Mistakes, and How to Avoid Them 288
3.17 Chapter Summary 289
3.17.1 Looking ahead 290
3.18 Exercises 290
3.19 Solutions to the Exercises 292
References 300

CHAPTER 4 Evaluation and Discussion I: Arrays, Searching and Sorting, Solved Step by Step 302
4.1 Where We Left Off 302
4.2 How to Approach a Data-Structure Problem 303
4.2.1 The eight steps 304
4.2.2 Estimating the cost before writing the code 305
4.2.3 How each problem is laid out, and where the code lives 306
4.3 Problem 4.1: Sorting Words Alphabetically 307
4.3.1 The problem 307
4.3.2 Understanding it by hand 307
4.3.3 Designing it on paper 307
4.3.4 The C++ solution 308
4.3.5 The Java and Python solutions 310
4.3.6 A tempting wrong solution 312
4.4 Problem 4.2: The First Repeated Character 313

4.4.1 The problem....313
4.4.2 Understanding it by hand....314
4.4.3 Designing it on paper....314
4.4.4 The C++ solution....315
4.4.5 The Java and Python solutions....317
4.4.6 A tempting wrong solution....318
4.5 Problem 4.3: Looking Up Money by Name....320
4.5.1 The problem....320
4.5.2 Understanding it by hand....320
4.5.3 Designing it on paper....320
4.5.4 The C++ solution....321
4.5.5 The Java and Python solutions....323
4.5.6 A tempting wrong solution....324
4.6 Problem 4.4: Sorting by Importance....326
4.6.1 The problem....326
4.6.2 Understanding it by hand....326
4.6.3 Designing it on paper....326
4.6.4 The C++ solution....327
4.6.5 The Java and Python solutions....329
4.6.6 A tempting wrong solution....331
4.7 Problem 4.5: Array Addresses by Hand....332
4.7.1 The problem....332
4.7.2 Understanding it by hand....332
4.7.3 Designing it on paper....333
4.7.4 The C++ solution....333
4.7.5 The Java and Python solutions....335
4.7.6 A tempting wrong solution....336
4.8 Problem 4.6: Counting a Key in a Sorted Array....336
4.8.1 The problem....336
4.8.2 Understanding it by hand....337
4.8.3 Designing it on paper....337
4.8.4 The C++ solution....338
4.8.5 The Java and Python solutions....340
4.8.6 A tempting wrong solution....341
4.9 Problem 4.7: Interpolation Search on Book Codes....343
4.9.1 The problem....343
4.9.2 Understanding it by hand....343
4.9.3 Designing it on paper....343
4.9.4 The C++ solution....344
4.9.5 The Java and Python solutions....346
4.9.6 A tempting wrong solution....350
4.10 Problem 4.8: Tracing the Elementary Sorts....351
4.10.1 The problem....351
4.10.2 Understanding it by hand....351
4.10.3 Designing it on paper....351
4.10.4 The C++ solution....352
4.10.5 The Java and Python solutions....355
4.10.6 A tempting wrong solution....359
4.11 Problem 4.9: Merging Two Sorted Lists....360
4.11.1 The problem....360
4.11.2 Understanding it by hand....360
4.11.3 Designing it on paper....360
4.11.4 The C++ solution....361
4.11.5 The Java and Python solutions....362
4.11.6 A tempting wrong solution....364

4.12 Problem 4.10: Deleting a Value in Place ... 365
4.12.1 The problem ... 365
4.12.2 Understanding it by hand ... 365
4.12.3 Designing it on paper ... 366
4.12.4 The C++ solution ... 367
4.12.5 The Java and Python solutions ... 369
4.12.6 A tempting wrong solution ... 370
4.13 Problem 4.11: Two Prices That Match a Voucher ... 371
4.13.1 The problem ... 371
4.13.2 Understanding it by hand ... 371
4.13.3 Designing it on paper ... 371
4.13.4 The C++ solution ... 372
4.13.5 The Java and Python solutions ... 374
4.13.6 A tempting wrong solution ... 376
4.14 The Problems Side by Side ... 377
4.15 Profitina in Practice ... 377
4.16 Common Mistakes, and How to Avoid Them ... 378
4.17 Chapter Summary ... 379
4.18 Exercises ... 379
4.19 Solutions to the Exercises ... 380
References ... 381

CHAPTER 5 Stacks and Queues ... 383

5.1 Where We Left Off ... 383
5.1.1 From free access to disciplined access ... 383
5.1.2 Abstract data types ... 384
5.2 A Story to Begin: the Towers of Hanoi ... 384
5.2.1 Three disks, by hand ... 385
5.2.2 How many moves? ... 385
5.2.3 Solving Hanoi with three stacks and no recursion ... 386
5.3 The Stack as an Abstract Data Type ... 389
5.3.1 The operations ... 389
5.3.2 The operations in pseudocode ... 390
5.4 An Array-Based Stack ... 391
5.4.1 Watching the array ... 391
5.4.2 The stack in C++ ... 392
5.4.3 The same stack in Java and Python ... 395
5.4.4 Every operation is $O(1)$ ... 399
5.5 Growing Stacks and Linked Stacks ... 400
5.5.1 Doubling the array ... 400
5.5.2 A preview: the linked stack ... 401
5.6 Application: Matching Brackets ... 404
5.6.1 The algorithm ... 404
5.6.2 The program ... 405
5.7 Application: Undo and Redo ... 407
5.8 Application: Infix to Postfix with the Shunting-Yard Algorithm ... 409
5.8.1 Three ways to write the same sum ... 410
5.8.2 Precedence and associativity ... 410
5.8.3 The algorithm ... 410
5.8.4 Tracing `3+2*5` ... 411
5.8.5 Tracing `a+b*c-d` ... 412
5.9 Application: Evaluating Postfix Expressions ... 412
5.9.1 The complete calculator ... 413
5.10 Application: The Function Call Stack ... 419
5.10.1 What happens when a function is called ... 419

5.10.2 Watching it happen 419
5.11 The Queue as an Abstract Data Type 421
5.11.1 The operations 422
5.12 Queues in a Linear Array, and the Drifting Problem 423
5.12.1 First attempt: shift everything on every dequeue 423
5.12.2 Second attempt: let the indices move 424
5.13 The Circular Queue 427
5.13.1 Modular arithmetic 427
5.13.2 First design: front plus a count 427
5.13.3 The circular queue in C++, Java and Python 429
5.13.4 Second design: leave one slot empty 434
5.14 The Deque 437
5.15 Priority Queues: a Preview 441
5.16 Stacks and Queues in the Standard Libraries 441
5.16.1 C++: container adaptors 441
5.16.2 Java: `ArrayDeque` for both 443
5.16.3 Python: `list`, `collections.deque`, `queue` and `heapq` 444
5.17 Choosing Among Them: Costs at a Glance 446
5.18 Profitina in Practice 447
5.19 Common Mistakes, and How to Avoid Them 448
5.20 Chapter Summary 449
5.21 Exercises 449
5.22 Solutions to the Exercises 451
References 455

CHAPTER 6 Pointers, References, and Functions 457
6.1 Where We Left Off 457
6.2 Memory, Addresses, and What a Pointer Is 458
6.2.1 Memory as a street of numbered houses 458
6.2.2 The address-of operator `&` and a first pointer 459
6.2.3 The dereference operator `*` 460
6.2.4 A pointer's type matters 461
6.2.5 Drawing memory: the box-and-arrow convention 461
6.3 Operations on Pointers 462
6.3.1 Assignment: copying an address versus copying a value 462
6.3.2 Comparison 463
6.3.3 Arithmetic: moving within an array 463
6.3.4 What is allowed, and what is not 466
6.4 Pointers and Arrays 467
6.4.1 An array name becomes a pointer 467
6.4.2 Walking an array with a pointer 467
6.4.3 Strings are arrays of `char`, and the terminator matters 468
6.4.4 Arrays as function parameters 470
6.4.5 Two-dimensional arrays 472
6.5 Pointers to Structures and the Arrow Operator 472
6.6 Dynamic Memory: `new` and `delete` 475
6.6.1 Two places to live: the stack and the heap 475
6.6.2 `new`, `delete`, `new[]` and `delete[]` 475
6.6.3 The growable array and the doubling rule 477
6.6.4 When `new` fails 478
6.7 When Pointers Go Wrong 479
6.7.1 Five bugs, one root cause 479
6.7.2 The bugs, on purpose 480
6.7.3 AddressSanitizer 483
6.7.4 Valgrind 484

6.7.5 Which tool, on which computer ... 485
6.8 `nullptr`: A Pointer to Nothing ... 485
6.9 References ... 487
6.9.1 A second name for an existing object ... 487
6.9.2 References versus pointers ... 489
6.10 Functions and the Three Ways to Pass an Argument ... 489
6.10.1 Functions: a short review ... 489
6.10.2 What happens during a call ... 491
6.10.3 The swap story ... 491
6.10.4 The cost of passing by value ... 493
6.10.5 Choosing a mode ... 494
6.11 `const` Correctness ... 494
6.12 Returning Pointers and References Safely ... 497
6.13 Smart Pointers: Ownership Written in the Type ... 500
6.13.1 The problem with raw `new` and `delete` ... 500
6.13.2 `std::unique_ptr`: exactly one owner ... 500
6.13.3 `std::shared_ptr` and `std::weak_ptr`: counted owners ... 502
6.14 Java References and Python Names ... 506
6.14.1 One rule each ... 506
6.14.2 Mutation versus rebinding, in Java ... 506
6.14.3 Mutation versus rebinding, in Python ... 508
6.14.4 The swap story in Java and Python ... 510
6.14.5 Who frees the memory? ... 511
6.14.6 Three languages side by side ... 512
6.15 The Cost of Each Operation ... 512
6.16 Profitina in Practice ... 513
6.17 Common Mistakes, and How to Avoid Them ... 517
6.18 Chapter Summary ... 517
6.19 Exercises ... 518
6.20 Solutions to the Exercises ... 520
References ... 525

CHAPTER 7 Singly Linked Lists ... 526
7.1 Where We Left Off ... 526
7.2 A Chain Instead of a Row: History and Idea ... 527
7.2.1 A treasure hunt ... 527
7.2.2 Where the idea came from ... 527
7.2.3 The vocabulary ... 528
7.3 Array versus Linked List ... 528
7.3.1 Two ways to keep five numbers ... 528
7.3.2 The comparison, feature by feature ... 530
7.3.3 The memory each element costs ... 530
7.4 Nodes and Pointers in C++ ... 531
7.4.1 Declaring a node ... 531
7.4.2 Making a node with `new` ... 531
7.4.3 Three nodes by hand ... 532
7.4.4 The other way: `malloc` and `free` ... 533
7.4.5 How to read the pointer diagrams in this chapter ... 534
7.5 The Singly Linked List with a Head Pointer ... 535
7.5.1 `init` and `isEmpty` ... 535
7.5.2 `frontInsertion`: a new first node ... 535
7.5.3 `backInsertion`: a new last node ... 536
7.5.4 `show`: walking the list ... 538
7.5.5 `frontDeletion`: removing the first node ... 539
7.5.6 `backDeletion`: removing the last node ... 539

7.5.7 `clear`: releasing every node 540
7.5.8 The complete program 541
7.5.9 The same list in Java and Python 544
7.6 More Operations: Search, Length, Insertion in the Middle, Deletion by Key, Reversal 550
7.6.1 `search` 550
7.6.2 `length`, iteratively and recursively 550
7.6.3 `insertAfter`: insertion in the middle 551
7.6.4 `insertSorted`: keeping the list in order 552
7.6.5 `deleteKey`: deletion by value 553
7.6.6 Reversing a list iteratively 554
7.6.7 Reversing a list recursively 555
7.6.8 `destroy` 557
7.6.9 The complete program 557
7.7 A Dummy Head Node: Fewer Special Cases 560
7.7.1 Where the special cases come from 560
7.7.2 The operations without special cases 561
7.7.3 A third way: a pointer to the link 564
7.7.4 Which variant to use 566
7.8 Head and Tail Pointers 566
7.8.1 The idea 566
7.8.2 Why `tail` cannot speed up back deletion 567
7.8.3 The program 568
7.9 The Circular Singly Linked List 571
7.9.1 Closing the chain 571
7.9.2 Traversal needs a new stop signal 572
7.9.3 Round-robin service at the reading desk 572
7.10 Stacks and Queues Built on a Linked List 575
7.10.1 The fit 575
7.10.2 The Pustaka Ceria return cart and hold queue 576
7.10.3 Linked or array-based? 580
7.11 Memory Safety: Proving There Are No Leaks 580
7.11.1 Why “it printed the right answer” is not enough 580
7.11.2 Valgrind 583
7.11.3 AddressSanitizer 585
7.11.4 A checklist for every linked-list function 587
7.12 Linked Lists in the Standard Libraries 588
7.12.1 C++: `std::forward_list` and `std::list` 588
7.12.2 Java: `java.util.LinkedList` 589
7.12.3 Python: why there is no built-in linked list, and `collections.deque` 590
7.12.4 Measuring: array against list on a real machine 592
7.13 Profitina in Practice 595
7.14 Common Mistakes, and How to Avoid Them 596
7.15 Chapter Summary 597
7.16 Exercises 598
7.17 Solutions to the Exercises 599
References 605

CHAPTER 8 Evaluation and Discussion II: Stacks, Pointers, and Linked Lists at Work 607

8.1 Where We Stand 607
8.2 How to Approach a Pointer Problem 608
8.3 Problem 8.1 — Completing an Array Stack 611
8.3.1 The problem 611
8.3.2 In our own words 611
8.3.3 By hand, step by step 612
8.3.4 The C++ solution 612

8.3.5 The Java and Python solutions....615
8.3.6 A tempting wrong solution....619
8.3.7 Check yourself....619
8.4 Problem 8.2 — Walking a Two-Way List by Directions....619
8.4.1 The problem....619
8.4.2 Just enough about nodes with two links....620
8.4.3 The walk, by hand....621
8.4.4 What can go wrong in the input....622
8.4.5 The C++ solution....622
8.4.6 The Java and Python solutions....625
8.4.7 A tempting wrong solution....629
8.4.8 Check yourself....629
8.5 Problem 8.3 — Evens Up, Odds Down, With Linked Lists Only....630
8.5.1 The problem....630
8.5.2 In our own words....630
8.5.3 By hand, step by step....630
8.5.4 The C++ solution....632
8.5.5 The Java and Python solutions....633
8.5.6 A tempting wrong solution....636
8.5.7 Check yourself....637
8.6 Problem 8.4 — Reversing the Words of a Sentence....637
8.6.1 The problem....637
8.6.2 In our own words....637
8.6.3 By hand, step by step....637
8.6.4 The C++ solution....638
8.6.5 The Java and Python solutions....639
8.6.6 A tempting wrong solution....641
8.6.7 Check yourself....642
8.7 Problem 8.5 — Removing Duplicate Values From a List....642
8.7.1 The problem....642
8.7.2 In our own words....642
8.7.3 By hand, step by step....642
8.7.4 The C++ solution....644
8.7.5 The Java and Python solutions....646
8.7.6 A tempting wrong solution....650
8.7.7 Check yourself....651
8.8 New Worked Problems on Stacks and Queues....651
8.8.1 Problem 8.6 — Matching brackets....651
8.8.2 Problem 8.7 — A circular queue of borrow requests....657
8.8.3 Problem 8.8 — From infix to postfix, and the value of an expression....662
8.9 A New Worked Problem on Pointers and Functions....669
8.9.1 Problem 8.9 — By value, by reference, by pointer....669
8.10 New Worked Problems on Singly Linked Lists....676
8.10.1 Problem 8.10 — Reversing a list in place....676
8.10.2 Problem 8.11 — Merging two sorted lists....682
8.10.3 Problem 8.12 — A queue made of linked nodes....689
8.11 The Problems of This Chapter Compared....696
8.12 Profitina in Practice....697
8.13 Common Mistakes, and How to Avoid Them....697
8.14 Chapter Summary....698
8.15 Exercises....699
8.16 Solutions to the Exercises....700
References....711

CHAPTER 9 Doubly and Circular Linked Lists 713
9.1 Where We Left Off 713
9.2 Two Arrows per Node 714
9.2.1 A train you can walk in both directions 714
9.2.2 The definition and the vocabulary 714
9.2.3 Declaring the node, and making a new one 715
9.2.4 What the second pointer costs 717
9.2.5 How to read the pointer diagrams in this chapter 718
9.3 The Doubly Linked List with a Head Pointer 718
9.3.1 `init` and `isEmpty` 718
9.3.2 `frontInsertion`: a new first node 718
9.3.3 `backInsertion`: a new last node 720
9.3.4 Traversing in both directions 721
9.3.5 `frontDeletion`: removing the first node 721
9.3.6 `backDeletion`: removing the last node 722
9.3.7 `clear` 723
9.3.8 The complete program 723
9.3.9 The same list in Java and Python 728
9.4 Head and Tail: Every End in O(1) 734
9.4.1 Adding `tail`, and back deletion at last in O(1) 734
9.4.2 `insertAfter` and `insertBefore`: inserting next to a node you hold 735
9.4.3 `insertSorted`: keeping the list in order 737
9.4.4 `search`, `deleteNode` and `deleteKey` 738
9.4.5 Checking the links 739
9.4.6 The complete program 739
9.5 A Doubly Linked List with a Head Node 744
9.5.1 Where the remaining special cases come from 744
9.5.2 The program 745
9.6 Circular Doubly Linked Lists 748
9.6.1 Closing both chains 748
9.6.2 Traversal needs a new stop signal 749
9.6.3 The lobby carousel 749
9.7 The Sentinel: Removing Every Special Case 752
9.7.1 The idea 752
9.7.2 Why the special cases disappear 753
9.7.3 Searching with the key in the sentinel 755
9.7.4 The program 755
9.7.5 The special cases, counted 758
9.8 A Deque for the Returns Desk 758
9.8.1 What a deque is 758
9.8.2 The Pustaka Ceria returns desk 758
9.9 A “Recently Borrowed” List: The LRU Pattern 762
9.9.1 The problem 762
9.9.2 Two structures working together 762
9.9.3 The program 764
9.10 Costs Compared: Singly Linked, Doubly Linked, Array 766
9.10.1 The table 766
9.10.2 Measuring what the second pointer buys 767
9.11 Memory Safety for Two-Way Links 771
9.11.1 What changes with two links 771
9.11.2 What the tools say 775
9.11.3 A checklist for two-way links 777
9.12 Doubly Linked Lists in the Standard Libraries 777
9.12.1 C++: `std::list` 777
9.12.2 Java: `LinkedList` and `LinkedHashMap` 779

9.12.3 Python: `collections.deque`, `OrderedDict` and `lru_cache` 781
9.13 Profitina in Practice 783
9.14 Common Mistakes, and How to Avoid Them 785
9.15 Chapter Summary 786
9.16 Exercises 787
9.17 Solutions to the Exercises 788
References 795

CHAPTER 10 Recursion 797
10.1 Where We Left Off 797
10.2 Thinking Recursively 798
10.2.1 A question passed along a queue 798
10.2.2 Recursive definitions 799
10.2.3 Proving a recursive definition right: induction 799
10.2.4 From definition to program 800
10.2.5 Three rules, and an honest word about cost 802
10.3 The Call Stack, Frame by Frame 802
10.3.1 Frames, revisited 802
10.3.2 Factorial, frame by frame 803
10.3.3 The trace as a program 804
10.3.4 When the base case is missing 808
10.4 Factorial: Iterative and Recursive 809
10.4.1 Two algorithms for one function 809
10.4.2 What each costs 809
10.5 Fibonacci and the Rabbits 810
10.5.1 The rabbit problem 810
10.5.2 Naive recursion, and why it explodes 811
10.5.3 Memoization: remember what you have computed 813
10.5.4 Bottom-up: no recursion at all 813
10.5.5 Tail recursion: the loop, written as a recursion 813
10.5.6 Fast doubling: $O(\log n)$ 814
10.5.7 Binet's closed form 814
10.5.8 The six ways in one program 815
10.5.9 Measured: calls and time 819
10.6 Euclid's Greatest Common Divisor 821
10.6.1 The problem and the idea 821
10.6.2 By hand, and as a picture 821
10.6.3 The program 822
10.6.4 How many divisions? Lamé's theorem 825
10.7 Fast Exponentiation 825
10.7.1 Subtracting one: n multiplications 825
10.7.2 Halving: about $2\lg n$ multiplications 825
10.7.3 Modular exponentiation 826
10.8 The Towers of Hanoi, Solved Recursively 829
10.8.1 The recursive idea 829
10.8.2 The program 830
10.8.3 The order of the calls: a recursion tree 836
10.8.4 Why exactly $2n - 1$ moves, and why no fewer 836
10.9 Recursion on Arrays and Linked Lists 837
10.9.1 Two ways to make an array smaller 837
10.9.2 A linked list is a recursive structure 837
10.9.3 Costs, and a caution 841
10.10 Recursion Trees 841
10.10.1 Drawing the calls 841
10.10.2 Merge sort and Hanoi, level by level 842

10.10.3 A table of recurrences 842
10.11 Tail Recursion and What Compilers Actually Do 843
10.11.1 Tail calls 843
10.11.2 What GCC does: an experiment 844
10.11.3 Why you must not rely on it in C++ 847
10.11.4 The JVM and CPython do not eliminate tail calls 847
10.12 How Deep Can Recursion Go? Limits in C++, Java and Python 849
10.12.1 C++: the size of the stack 849
10.12.2 Java: `-Xss`, and threads with a stack of their own 851
10.12.3 Python: the recursion limit 852
10.12.4 The limits at a glance 854
10.13 From Recursion to Iteration with an Explicit Stack 854
10.13.1 When to convert 854
10.13.2 The general method 855
10.13.3 Hanoi with a stack of frames 855
10.13.4 A long list walked backward 856
10.14 Profitina in Practice 858
10.15 Common Mistakes, and How to Avoid Them 862
10.16 Chapter Summary 863
10.17 Exercises 864
10.18 Solutions to the Exercises 865
References 871

CHAPTER 11 Trees and Binary Search Trees 873
11.1 Where We Left Off 873
11.2 What a Tree Is 874
11.2.1 Four trees from everyday life 874
11.2.2 The definition 875
11.2.3 The vocabulary 875
11.2.4 A worked example on a small tree 877
11.2.5 Static and dynamic trees 877
11.3 Drawing and Writing a Tree 878
11.4 Representing a Tree in Memory 878
11.4.1 The static representation: an array 878
11.4.2 The dynamic representation: linked nodes 881
11.4.3 Trees with any number of children 882
11.5 Binary Trees 883
11.5.1 Definition 883
11.5.2 Full, perfect, complete and skewed 883
11.5.3 How many nodes fit, and how low a tree can be 884
11.6 Traversing a Binary Tree 885
11.6.1 Three depth-first orders 885
11.6.2 A walk around the tree 887
11.6.3 Level order with a queue 887
11.6.4 Traversals without recursion 888
11.6.5 What traversals cost 893
11.6.6 Expression trees 893
11.7 The Binary Search Tree 897
11.7.1 The BST property 897
11.7.2 Insertion: `insert(65, 5, 70, 3, 10)` 898
11.7.3 Search 899
11.7.4 Minimum and maximum 900
11.7.5 Counting nodes, leaves and height 901
11.7.6 Printing a tree sideways 902
11.7.7 The complete program 902

11.7.8 The same tree in Java and Python 908
11.7.9 Successor and predecessor 916
11.7.10 Deletion: three cases 918
11.7.11 Two variants: successor or predecessor 920
11.7.12 The deletion program 921
11.7.13 Costs 926
11.8 Balanced and Degenerate Trees 927
11.8.1 The order of insertion decides the shape 927
11.8.2 Measured heights 927
11.8.3 What "balanced" means 931
11.9 A First Look at Self-Balancing Trees 931
11.9.1 Rotations 931
11.9.2 AVL trees 932
11.9.3 Red-black trees 935
11.9.4 Other balanced trees 936
11.10 General Trees: Conversion and Reconstruction 936
11.10.1 First child, next sibling 936
11.10.2 Rebuilding a tree from preorder and degrees 937
11.10.3 The program 938
11.11 Search Trees in the Standard Libraries 942
11.11.1 C++: `std::set` and `std::map` 943
11.11.2 Java: `TreeMap` and `TreeSet` 945
11.11.3 Python: no built-in search tree 946
11.12 Profitina in Practice 949
11.13 Common Mistakes, and How to Avoid Them 955
11.14 Chapter Summary 956
11.15 Exercises 956
11.16 Solutions to the Exercises 958
References 963

CHAPTER 12 Evaluation and Discussion III: Recursion at Work on Lists and Trees 965
12.1 Where We Stand 965
12.2 How to Approach a Recursive Problem on a Linked Structure 966
12.3 The Tree We Work On 969
12.3.1 The starting point 969
12.3.2 The toolkit, in three languages 971
12.3.3 Looking at the tree 977
12.4 Problem 12.1 — Depth-First Traversal, Left Branch First 979
12.4.1 The problem 979
12.4.2 In our own words 979
12.4.3 By hand, step by step 980
12.4.4 The C++ solution 980
12.4.5 The Java and Python solutions 981
12.4.6 A tempting wrong solution 983
12.4.7 Check yourself 984
12.5 Problem 12.2 — Printing One Level 984
12.5.1 The problem 984
12.5.2 In our own words 984
12.5.3 By hand, step by step 984
12.5.4 The C++ solution 985
12.5.5 The Java and Python solutions 986
12.5.6 A tempting wrong solution 987
12.5.7 Check yourself 988
12.6 Problem 12.3 — The Sum of All Nodes 988
12.6.1 The problem 988

12.6.2 In our own words 988
12.6.3 By hand, step by step 988
12.6.4 The C++ solution 989
12.6.5 The Java and Python solutions 990
12.6.6 A tempting wrong solution 991
12.6.7 Check yourself 991
12.7 Problem 12.4 — Tripling Every Node 991
12.7.1 The problem 991
12.7.2 In our own words 991
12.7.3 The C++ solution 992
12.7.4 The Java and Python solutions 993
12.7.5 A tempting wrong solution 994
12.7.6 Check yourself 995
12.8 Problem 12.5 — The Mirror Image 995
12.8.1 The problem 995
12.8.2 In our own words 995
12.8.3 By hand, step by step 995
12.8.4 The C++ solution 996
12.8.5 The Java and Python solutions 997
12.8.6 A tempting wrong solution 999
12.8.7 Check yourself 999
12.9 Problem 12.6 — Is the Tree Height-Balanced? 1000
12.9.1 The problem 1000
12.9.2 In our own words 1000
12.9.3 By hand, step by step 1000
12.9.4 The C++ solution 1001
12.9.5 The Java and Python solutions 1003
12.9.6 A tempting wrong solution 1005
12.9.7 Check yourself 1006
12.10 Problem 12.7 — Deleting a Value 1006
12.10.1 The problem 1006
12.10.2 In our own words 1006
12.10.3 By hand, step by step 1007
12.10.4 The C++ solution 1008
12.10.5 The Java and Python solutions 1009
12.10.6 A tempting wrong solution 1012
12.10.7 Check yourself 1012
12.11 Problem 12.8 — Finding the Parent 1012
12.11.1 The problem 1012
12.11.2 In our own words 1012
12.11.3 By hand, step by step 1013
12.11.4 The C++ solution 1013
12.11.5 The Java and Python solutions 1014
12.11.6 A tempting wrong solution 1016
12.11.7 Check yourself 1017
12.12 Problem 12.9 — Reporting the Children 1017
12.12.1 The problem 1017
12.12.2 In our own words 1017
12.12.3 The C++ solution 1017
12.12.4 The Java and Python solutions 1018
12.12.5 A tempting wrong solution 1020
12.12.6 Check yourself 1020
12.13 Problem 12.10 — From a Tree to a Doubly Linked List, in Place 1021
12.13.1 The problem 1021
12.13.2 In our own words 1021

12.13.3 The idea, by hand 1022
12.13.4 The C++ solution 1022
12.13.5 The Java and Python solutions 1024
12.13.6 A tempting wrong solution 1027
12.13.7 Check yourself 1028
12.14 New Worked Problems on Doubly and Circular Lists 1028
12.14.1 Problem 12.11 — Reversing a doubly linked list in place 1028
12.14.2 Problem 12.12 — The Josephus circle 1034
12.15 A New Worked Problem on Recursion 1040
12.15.1 Problem 12.13 — Choosing books 1040
12.16 New Worked Problems on Trees 1045
12.16.1 Problem 12.14 — Is it really a binary search tree? 1045
12.16.2 Problem 12.15 — The lowest common ancestor 1049
12.16.3 Problem 12.16 — Printing a tree level by level 1052
12.17 The Problems of This Chapter Compared 1056
12.18 Profitina in Practice 1057
12.19 Common Mistakes, and How to Avoid Them 1057
12.20 Chapter Summary 1058
12.21 Exercises 1058
12.22 Solutions to the Exercises 1059
References 1066

CHAPTER 13 Graphs 1068
13.1 Where We Left Off 1068
13.2 A Story to Begin: Seven Bridges and a Campus Map 1069
13.2.1 The bridges of Königsberg 1069
13.2.2 From bridges to library branches 1070
13.3 The Vocabulary of Graphs 1070
13.3.1 Vertices and edges 1070
13.3.2 Directed and undirected 1071
13.3.3 Weighted graphs 1071
13.3.4 Degree and the handshake lemma 1071
13.3.5 Paths and cycles 1072
13.3.6 Connectivity and connected components 1073
13.3.7 Directed acyclic graphs 1073
13.3.8 How many edges can a graph have? Sparse and dense 1073
13.4 The Running Example: the Pustaka Ceria Campus Map 1074
13.5 Representing a Graph in Memory 1075
13.5.1 The adjacency matrix 1075
13.5.2 The adjacency list 1076
13.5.3 The edge list 1076
13.5.4 The costs side by side 1076
13.5.5 The program in C++ 1077
13.5.6 The same program in Java and Python 1080
13.6 Breadth-First Search 1084
13.6.1 The idea: ripples in a pond 1084
13.6.2 The procedure 1084
13.6.3 Tracing BFS from the main library 1085
13.6.4 Why BFS finds fewest-hop routes 1087
13.6.5 The running time 1087
13.6.6 The program in C++ 1088
13.6.7 The same program in Java and Python 1090
13.7 Depth-First Search 1094
13.7.1 The idea: exploring a maze with a ball of thread 1094
13.7.2 Recursive DFS with discovery and finish times 1095

13.7.3 Tracing DFS on the campus map 1095
13.7.4 The parenthesis structure 1097
13.7.5 DFS without recursion 1097
13.7.6 Finding a cycle in an undirected graph 1098
13.7.7 The programs 1098
13.8 Connectivity and Connected Components 1107
13.8.1 One search per component 1107
13.8.2 The shuttle road 1107
13.8.3 The programs 1108
13.8.4 Directed graphs: a note on strong connectivity 1114
13.9 Directed Acyclic Graphs and Topological Sort 1114
13.9.1 The new-book workflow 1114
13.9.2 Kahn's algorithm 1115
13.9.3 Topological sort by DFS 1116
13.9.4 Detecting a cycle in a directed graph 1116
13.9.5 The programs 1117
13.10 Weighted Shortest Paths: a Gentle Dijkstra 1125
13.10.1 Why BFS is not enough 1125
13.10.2 Relaxation 1125
13.10.3 Dijkstra's algorithm 1125
13.10.4 Tracing Dijkstra from the main library 1126
13.10.5 Why it works, and when it does not 1127
13.10.6 Running time, and a look ahead to Chapter 14 1127
13.10.7 The programs 1128
13.11 Graphs at Real-World Scale 1134
13.11.1 Road networks 1134
13.11.2 Social graphs 1135
13.11.3 What changes at scale, and what does not 1135
13.12 Choosing an Algorithm: Costs at a Glance 1136
13.13 Profitina in Practice 1136
13.14 Common Mistakes, and How to Avoid Them 1141
13.15 Chapter Summary 1142
13.16 Exercises 1143
13.17 Solutions to the Exercises 1144
References 1154

CHAPTER 14 Heaps and Priority Queues 1156

14.1 Where We Left Off 1156
14.2 Serving the Most Urgent First 1157
14.2.1 The priority queue as an abstract data type 1157
14.2.2 Why the obvious structures are not enough 1157
14.2.3 Where the heap came from 1158
14.3 A Complete Binary Tree in an Array 1158
14.3.1 Complete binary trees 1158
14.3.2 Numbering the nodes level by level 1159
14.3.3 Why the formulas are right 1160
14.3.4 Height, and where the leaves are 1161
14.4 The Heap Property 1163
14.4.1 Max-heaps and min-heaps 1163
14.4.2 What the property guarantees 1163
14.4.3 What the property does not say 1164
14.5 Sift-Up: Repairing a Key That Is Too Large 1164
14.5.1 The procedure 1164
14.5.2 Why it is correct 1165
14.5.3 What it costs 1166

14.6 Sift-Down: Repairing a Key That Is Too Small ... 1166
14.6.1 The procedure ... 1166
14.6.2 Why it is correct, and what it costs ... 1167
14.7 Insert, GetMax and ExtractMax ... 1168
14.7.1 `getMax` ... 1168
14.7.2 `insert` ... 1168
14.7.3 `extractMax` ... 1169
14.8 GetMin in a Max-Heap ... 1169
14.9 IncreaseKey and DecreaseKey ... 1170
14.10 Building a Heap Bottom-Up in Linear Time ... 1172
14.10.1 Two ways to build ... 1172
14.10.2 Why it takes only linear time ... 1173
14.10.3 Building by insertion is really slower ... 1174
14.11 Checking the Heap Property: `isMaxHeap` ... 1176
14.12 The Complete `MaxHeap` Class ... 1177
14.12.1 The interface ... 1177
14.12.2 The C++ class ... 1178
14.12.3 The same class in Java and Python ... 1184
14.13 Heap Sort in Full ... 1192
14.13.1 The algorithm ... 1192
14.13.2 Correctness ... 1196
14.13.3 Cost ... 1196
14.13.4 Why heap sort is not stable ... 1196
14.14 Priority Queues in the Standard Libraries ... 1197
14.14.1 C++: `std::priority_queue` ... 1197
14.14.2 Java: `java.util.PriorityQueue` ... 1199
14.14.3 Python: `heapq`, including the new max-heap functions ... 1200
14.15 Applications: Top-k, Merging k Sorted Lists, and Dijkstra ... 1202
14.15.1 The k largest items of a stream ... 1202
14.15.2 Merging k sorted lists ... 1203
14.15.3 The hook to Chapter 13: Dijkstra's shortest paths ... 1206
14.16 Profitina in Practice ... 1208
14.17 Common Mistakes, and How to Avoid Them ... 1209
14.18 Chapter Summary ... 1210
14.19 Exercises ... 1211
14.20 Solutions to the Exercises ... 1212
References ... 1218

CHAPTER 15 Hashing and Hash Tables ... 1219
15.1 Where We Left Off ... 1219
15.2 From $O(n)$ to $O(\lg n)$ to $O(1)$: the Idea of Hashing ... 1220
15.2.1 The librarian who never searches ... 1220
15.2.2 Direct addressing ... 1220
15.2.3 Hashing: the vocabulary ... 1221
15.2.4 The progression in one table ... 1222
15.3 Hash Functions ... 1222
15.3.1 What a good hash function must do ... 1222
15.3.2 The division method ... 1223
15.3.3 The multiplication method ... 1223
15.3.4 Hashing strings ... 1224
15.3.5 The rolling update ... 1225
15.3.6 Avalanche: how well does a function mix? ... 1225
15.3.7 The program ... 1226
15.4 Collisions and the Birthday Bound ... 1232
15.4.1 Collisions cannot be avoided ... 1232

15.4.2 The birthday problem ... 1232
15.4.3 The numbers, computed ... 1233
15.4.4 What this means for a hash table ... 1236
15.5 Separate Chaining ... 1236
15.5.1 A linked list in every bucket ... 1236
15.5.2 The program ... 1237
15.5.3 The same table in Java and Python ... 1243
15.5.4 How long are the chains? ... 1250
15.6 Open Addressing ... 1251
15.6.1 Keys that move instead of chains that grow ... 1251
15.6.2 Linear probing ... 1251
15.6.3 Quadratic probing ... 1252
15.6.4 Double hashing ... 1252
15.6.5 Deletion and tombstones ... 1253
15.6.6 The program ... 1254
15.6.7 Chaining or open addressing? ... 1265
15.7 Load Factor and Rehashing, Measured ... 1266
15.7.1 The one number that matters ... 1266
15.7.2 What the theory predicts ... 1266
15.7.3 The measurements ... 1267
15.7.4 How chain lengths are distributed ... 1272
15.7.5 Rehashing, and why doubling is cheap ... 1273
15.8 Complexity, Worst Cases and Hash Flooding ... 1274
15.8.1 The complete bill ... 1274
15.8.2 Hash flooding ... 1274
15.8.3 Randomized hashing: the defense ... 1277
15.9 `std::unordered_map` versus `std::map` ... 1278
15.9.1 Same interface, different machines ... 1278
15.9.2 The program ... 1278
15.10 Inside Java's `HashMap` and Python's `dict` ... 1283
15.10.1 Java: `HashMap` and its treeified bins ... 1283
15.10.2 Python: the compact, insertion-ordered `dict` ... 1288
15.11 Applications ... 1290
15.11.1 Counting words in the catalog ... 1294
15.11.2 Detecting duplicates ... 1294
15.11.3 Two-sum ... 1294
15.11.4 Caches ... 1295
15.11.5 Reverse maps ... 1295
15.12 Profitina in Practice ... 1295
15.13 Common Mistakes, and How to Avoid Them ... 1296
15.14 Chapter Summary ... 1297
15.15 Exercises ... 1298
15.16 Solutions to the Exercises ... 1299
References ... 1305

CHAPTER 16 Evaluation and Discussion IV: Graphs, Heaps, and Hash Tables at Work ... 1307
16.1 Where We Stand ... 1307
16.2 How to Approach a Problem About a Structure ... 1309
16.3 Problem 16.1 — Completing a Max-Heap Class ... 1311
16.3.1 The problem ... 1311
16.3.2 In our own words ... 1312
16.3.3 By hand, step by step ... 1312
16.3.4 The C++ solution ... 1316
16.3.5 The Java and Python solutions ... 1323
16.3.6 A tempting wrong solution ... 1331

16.3.7 Check yourself 1332
16.4 Problem 16.2 — Putting Tickets in Travel Order 1332
16.4.1 The problem 1332
16.4.2 In our own words 1332
16.4.3 By hand, step by step 1333
16.4.4 Which map is a hash map? 1333
16.4.5 The C++ solution 1335
16.4.6 The Java and Python solutions 1338
16.4.7 A tempting wrong solution 1341
16.4.8 Check yourself 1342
16.5 New Worked Problems on Graphs 1342
16.5.1 Problem 16.3 — The fewest hops between branches 1342
16.5.2 Problem 16.4 — An order in which to read the topics 1352
16.5.3 Problem 16.5 — The quickest route for the book van 1359
16.6 New Worked Problems on Heaps 1368
16.6.1 Problem 16.6 — The k largest values with a heap of size k 1368
16.6.2 Problem 16.7 — The median so far, with two heaps 1377
16.7 New Worked Problems on Hashing 1384
16.7.1 Problem 16.8 — A hash table with separate chaining, by hand 1384
16.7.2 Problem 16.9 — Open addressing, and deleting safely 1394
16.8 Choosing the Right Data Structure 1402
16.8.1 Start from the operations, not from the structure 1402
16.8.2 The experiment 1404
16.8.3 Reading the numbers 1431
16.8.4 The choice, in one paragraph 1433
16.9 The Problems of This Chapter Compared 1433
16.10 Profitina in Practice 1434
16.11 Common Mistakes, and How to Avoid Them 1435
16.12 Chapter Summary 1436
16.13 Exercises 1437
16.14 Solutions to the Exercises 1438
References 1453

List of Figures

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. 4
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. 6
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. 8
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. 12
Figure 1.5 The array `loans` of eight `int`s in memory: each element takes 4 bytes, so element i begins at $1000 + 4i$. 26
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. 29
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. 38
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. 46
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. 49
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. 53
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. 81
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. 93
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. 98
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. 99
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. 108
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. 116
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. 122
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. 127
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. 128
Figure 2.10 Nanoseconds per search on the 10,000-book catalog, by search method and language, on a logarithmic scale. Measured by **Listing 2.22**, **Listing 2.23** and **Listing 2.24** in September 2026. 151
Figure 2.11 Average probes per successful search as n grows from 103 to 106, on logarithmic axes. Sequential search rises with slope 1, jump search with slope 12; 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. 161
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. 169
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. 192
Figure 3.2 Exchanging two values with a spare variable. Each highlighted box is the one that changes in that step. 193
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. 198
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. 203
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. 207
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. 211
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. 216
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. 224
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. 229
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]. 230
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. 242
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. 249
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. 250
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. 258
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. 266
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. 267
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. 304
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. 308
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. 315
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. 321
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. 327
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. 333
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). 338
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. 344

Figure 4.9 The six comparisons of the first pass of bubble sort; amber pairs were swapped, teal pairs were already in order. 352
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. 361
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. 367
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. 372
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). 384
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. 385
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. 392
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. 402
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. 405
Figure 5.6 The undo and redo stacks at four moments of **Listing 5.8**. A new edit (panel 4) pushes onto the undo stack and empties the redo stack. 409
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 * +`. 411
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. 413
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. 421
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. 423
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. 423
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*. 424
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. 427
Figure 5.14 The circular queue of **Listing 5.12** 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. 428
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`. 435
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. 438
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`. 459
Figure 6.2 Two ways to assign with pointers. `*q = *p` moves a value between the boxes; `q = p` moves an arrow. 463
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. 464
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. 474
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. 475

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. 478
Figure 6.7 Three lifetime bugs drawn as boxes and arrows. Hatched blocks have already been freed. 479
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. 493
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. 503
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. 506
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. 528
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. 529
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. 536
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. 537
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. 538
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. 539
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. 540
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. 551
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. 552
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. 553
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. 555
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. 556
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. 561
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`. 567
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`. 571
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_`. 576
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. 581
Figure 7.18 The measurements of **Listing 7.16**: 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). 595
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. 612

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. 621
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. 621
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. 631
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. 638
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. 643
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. 658
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. 670
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. 677
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` 683
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. 689
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 715
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. 717
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. 719
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. 720
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. 721
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. 722
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. 723
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. 735
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. 736
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 738
Figure 9.11 `deleteKey(35)`: (a) `search` returns `p`, (b) the predecessor and the successor are linked to each other, (c) `p` is deleted 739
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. 745

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`. 748
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. 753
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. 754
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 $O(1)$. 759
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. 763
Figure 9.18 The measurements of **Listing 9.11** on logarithmic axes: the singly linked list's time grows fourfold per doubling of n (quadratic), the doubly linked list's twofold (linear). 771
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. 772
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). 798
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. 804
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. 811
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. 812
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. 820
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 $\gcd(228,90) = 6$. 822
Figure 10.7 Computing 313. (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. 826
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. 830
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. 836
Figure 10.10 Recursion trees with the cost of each level on the right. (a) Merge sort: $\lg n + 1$ levels of cost cn each. (b) Hanoi: level i costs $2i$, and the levels add up to $2n - 1$. 842
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). 847
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. 856
Figure 10.13 The service menu of Profitina Laundry: categories (teal) contain services (amber) and smaller categories, to any depth. 859

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. 874
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. 876
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. 878
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). 879
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. 882
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. 883
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. 884
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. 886
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. 887
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. 888
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. 889
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. 894
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. 895
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. 897
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. 899
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. 900
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. 901
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. 917
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). 918
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). 920
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. 927
Figure 11.22 The measurements of **Listing 11.8**: 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 four-fold with every doubling of n (right, logarithmic scale). 930

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. 932
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. 932
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. 935
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. 937
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). 938
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. 954
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. 970
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. 980
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. 985
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. 989
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. 992
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. 996
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. 1000
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. 1007
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. 1013
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. 1021
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. 1028
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. 1035
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. 1041
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. 1045
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. 1049
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. 1052

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. 1069

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. 1071

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. 1074

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. 1075

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. 1086

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. 1086

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. 1096

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. 1097

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. 1108

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. 1114

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. 1117

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. 1127

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. 1134

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. 1137

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. 1159

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. 1159

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. 1163

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. 1165

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. 1167

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. 1170

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. 1171

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. 1173

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. 1173
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 the intermediate steps shown in **Listing 14.6**'s output. 1193
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. 1197
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. 1204
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. 1206
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. 1221
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). 1236
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. 1237
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. 1253
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. 1253
Figure 15.6 Average probes per search in a table of 16,384 slots, measured by **Listing 15.9** (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. 1272
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. 1273
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. 1278
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. 1284
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.) 1288
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. 1313
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. 1313
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. 1314
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 $n - 1$. 1315
Figure 16.5 `buildMaxHeap` on {10, 7, 6, 5, 1, 8, 2}: only index 2 violates the invariant, and one swap repairs it. 1315

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). 1333
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. 1335
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. 1344
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. 1353
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). 1360
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. 1368
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. 1378
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. 1386
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. 1395
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. 1404
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. 1432

List of Tables

Table 1.1 Looks needed in the worst case to find one book among n, for three ways of keeping the books. 3
Table 1.2 The sixteen chapters of this book and what each one adds. 5
Table 1.3 The ten most popular programming languages in the TIOBE Index of September 2026. 6
Table 1.4 The basic C++ types used in this book, with their sizes on a 64-bit machine 14
Table 1.5 Arrays and growable arrays in C++, Java and Python. 52
Table 1.6 The cost of the basic operations on an array of n elements. 58
Table 1.7 Installing VS Code on each operating system. 60
Table 1.8 Installing and using a C++ compiler on each operating system. Fedora uses `sudo dnf install gcc-c++ gdb` and Arch `sudo pacman -S gcc gdb` in step 1. 60
Table 1.9 Installing JDK 25 on each operating system. 61
Table 1.10 Installing Python on each operating system. 62
Table 1.11 The VS Code extensions used in this book. 62
Table 1.12 The verification commands, what each should print, and the usual fix when it does not. 63
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. 81
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. 107
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). 151
Table 2.4 Searching with the standard libraries of C++, Java and Python. 162
Table 2.5 The search methods of this chapter, what each needs, and what each costs. 168

Table 3.1 How $n\lg n$ and $n2/2$ grow. At a billion items, a computer doing 109 steps per second needs 30 seconds for $n\lg n$ steps and 16 years for $n2/2$ steps. 195
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. 258
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. 259
Table 3.4 A historical measurement: sorting 10,000 numbers on a Pentium II at 450 MHz (late 1990s). Selection sort's descending time repeats bubble sort's and is almost certainly a copying slip; its time does not depend on the order of the input. 267
Table 3.5 The same experiment in 2026 (from the output of **Listing 3.34**). Absolute times fell by a factor of 10 to 60; the relationships between the algorithms did not change. 267
Table 3.6 The standard sorts of C++, Java and Python in September 2026, the algorithms inside them, and their guarantees. 280
Table 4.1 The eleven problems of this chapter and the tools from Chapters 1 to 3 that each one exercises most. 303
Table 4.2 Rough step counts for the growth rates of Chapters 2 and 3; at about 108 to 109 simple steps per second, 1012 steps take many minutes and 1016 steps take years. 305
Table 4.3 The two keys of every string of Problem 4.4, counted by hand. 326
Table 4.4 The counts of Problem 4.8 for the array 5, 6, 8, 1, 3, 25, 10, which has 7 inversions. 355
Table 4.5 The eleven problems of this chapter, the idea each one teaches, its cost, and the trap its tempting wrong solution demonstrates. 377
Table 5.1 The operations of the stack ADT, their preconditions, and their cost in the array implementation of Section 5.4. 390
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. 410
Table 5.3 Operator precedence and associativity used by the calculator. 410
Table 5.4 Converting `a+b*c-d` to postfix, one token per row. 412
Table 5.5 The operations of the queue ADT and their cost in the circular-array implementation of Section 5.13. 422
Table 5.6 The two ways to distinguish a full circular queue from an empty one. 435
Table 5.7 Three ways to implement a priority queue. The heap of Chapter 14 is the only one that makes both operations fast. 441
Table 5.8 The same operations in the three standard libraries. 446
Table 5.9 The cost of every structure in this chapter. 446
Table 6.1 The three meanings of `*` and `&` in C++. Context always decides which one is meant. 460
Table 6.2 An ordinary variable and a pointer side by side. 461
Table 6.3 The operations C++ defines on a pointer `p` of type `T*`. 466
Table 6.4 Memory-safety weaknesses in the 2025 CWE Top 25 Most Dangerous Software Weaknesses, published by MITRE in December 2025 [4]. 480
Table 6.5 AddressSanitizer and Valgrind compared, as of September 2026 [5], [6], [7], [8]. 485
Table 6.6 Pointers and references compared. 489
Table 6.7 Choosing how a C++ function receives its argument. 494
Table 6.8 The four combinations of `const` with a pointer. 495
Table 6.9 Raw pointers, references and smart pointers, and when to use each. 505
Table 6.10 Pointers, references and parameter passing in C++, Java and Python. 512
Table 6.11 The running time of the operations in this chapter. 513
Table 7.1 Arrays and singly linked lists compared. 530
Table 7.2 Running time of each operation of the head-pointer list with n nodes. 544
Table 7.3 Cost of the operations of Section 7.6 on a list of n nodes. 560
Table 7.4 The singly linked list variants compared, for a list of n nodes. 571
Table 7.5 The linked lists of the C++, Java and Python standard libraries. 592
Table 8.1 The twelve problems of this chapter, the section that solves each one, and the chapter that supplies its main idea. 608

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. 609
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. 610
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. 621
Table 8.5 The ten insertions of the sample input of Problem 8.3. Printing `evens` and then `odds` gives the expected output. 631
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. 663
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`. 676
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. 696
Table 9.1 Running times with only a `head` pointer, for a list of n nodes: the second pointer changes the backward traversal but not yet the back end. 728
Table 9.2 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. 744
Table 9.3 Special-case tests in the two basic operations, by representation. Only the sentinel removes all of them. 758
Table 9.4 The dynamic array, the singly linked list and the doubly linked list compared, for n elements. 766
Table 9.5 Doubly linked structures in the C++, Java and Python standard libraries. 783
Table 10.1 Seven ways to compute $F(n)$, counting each operation on machine words as one step. 820
Table 10.2 Common recurrences, their solutions, and the algorithms of this book that produce them. 843
Table 10.3 Recursion depth limits in C++, Java and Python, as measured on the machine of this book in September 2026. 854
Table 10.4 Recursion or iteration: a decision for each algorithm of this chapter. 855
Table 11.1 The vocabulary of trees, evaluated on the tree of **Figure 11.2**. 877
Table 11.2 The four shapes of binary tree, and the other names they go by. 884
Table 11.3 The cost of the operations of a binary search tree with n nodes and height h. 926
Table 11.4 Plain, AVL and red-black binary search trees compared. 936
Table 11.5 Ordered sets and maps in the standard libraries of C++, Java and Python, and their hash-based alternatives. 949
Table 12.1 The sixteen problems of this chapter, the section that solves each one, and the chapter that supplies its main idea. 966
Table 12.2 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. 968
Table 12.3 The tree ideas of this chapter in the three languages of the book. Only C++ lets a function receive the address of the caller's pointer; Java and Python return the new root instead, or keep the variables to be changed inside an object. 969
Table 12.4 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. 1007
Table 12.5 The sixteen problems of the chapter: the structure each uses, its cost, and the input on which its tempting wrong solution fails. n is the number of nodes and h the height of the tree. 1056
Table 13.1 The costs of the three graph representations, for a graph with V vertices and E edges. 1077
Table 13.2 Breadth-first search from LIB on the campus map, step by step. 1085
Table 13.3 Kahn's algorithm on the new-book workflow. 1116
Table 13.4 Dijkstra's algorithm from LIB on the campus map. The priority queue is listed in increasing order of distance. 1126
Table 13.5 Real graphs and the memory their two main representations would need. The OpenStreetMap segment count is estimated as nodes minus ways. 1135
Table 13.6 The graph algorithms of this chapter, with the structure each relies on and its cost on adjacency lists. 1136

Table 14.1 The operations of the `MaxHeap` class of this chapter and their costs. 1157
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. 1158
Table 14.3 The index arithmetic of a complete binary tree of n nodes stored in an array. 1160
Table 14.4 The public interface of the `MaxHeap` class in C++, Java and Python. 1178
Table 14.5 Heap sort compared with the other $O(n \lg n)$ comparison sorts of Chapter 3. 1196
Table 14.6 The priority queues of the three standard libraries. All three are binary heaps in arrays. 1202
Table 15.1 Searching for a key: from sequential search to hashing. 1222
Table 15.2 How many random keys make a collision likely, computed by **Listing 15.2** from Equation (15.7); the middle column is the approximation Equation (15.9). 1235
Table 15.3 Separate chaining and open addressing compared. 1265
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. 1274
Table 15.5 `std::map` and `std::unordered_map` compared. 1283
Table 15.6 The hash tables of the C++, Java and Python standard libraries (GCC 13, JDK 25, CPython 3.14). 1290
Table 16.1 The nine problems of this chapter, the section that solves each one, and the chapters that supply its main idea. 1308
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. 1310
Table 16.3 The structures of Chapters 13 to 15 in the three languages of this book. The C++ priority queue is a max-heap by default, the Java and Python ones are min-heaps; and in C++ the word `map` means a search tree, not a hash table. 1311
Table 16.4 Choosing a data structure by the operation the task performs most often: the whole book on one page. n is the number of items, r the number of results of a range query. 1403
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. 1432
Table 16.6 The nine problems of the chapter: the structure each uses, its cost, and the input on which its tempting wrong solution fails. 1434

List of Listings

Listing 1.1 `memory_basics.cpp`: the sizes of the basic types, the bytes of an `int`, and an address on the stack and on the heap. 8
Listing 1.2 `MemoryBasics.java`: Java's fixed type sizes and the bytes of an `int` in the machine's own order. 10
Listing 1.3 `memory_basics.py`: C type sizes seen through `struct`, the bytes of 2026, and the true size of Python objects. 11
Listing 1.4 `io_sum.cpp`: reading a count and that many numbers with `cin`, and summarizing them. 14
Listing 1.5 `IoSum.java`: the same summary with `Scanner`. 15
Listing 1.6 `io_sum.py`: the same summary, reading all tokens from standard input at once. 16
Listing 1.7 `late_fee.cpp`: constants, a struct, a function with `if` and `switch`, an array of structs, and a range-based `for` loop with `continue`. 18
Listing 1.8 `LateFee.java`: the late fees with a Java `record` and a `switch`. 20
Listing 1.9 `late_fee.py`: the late fees with a `dataclass` and a `match` statement. 21
Listing 1.10 `address_of.cpp`: `&` gives an address, `*` follows it, and a function that receives an address can change the caller's variable. 23
Listing 1.11 `AddressOf.java`: Java copies basic values into methods, but shares objects by reference. 24

Listing 1.12 `address_of.py`: Python names refer to objects; `id()` shows that two names refer to the same list. 24
Listing 1.13 `book.h`: the `Book` record of Pustaka Ceria, with fixed-size character arrays, and helpers to fill and print it. 26
Listing 1.14 `catalog_1d.cpp`: Pustaka Ceria's catalog as an array of eight `Book`s, with the byte offset of every element. 27
Listing 1.15 `Book.java`: the `Book` record in Java, with `String` fields. 29
Listing 1.16 `Catalog1D.java`: the catalog as a Java array of `Book` references. 30
Listing 1.17 `book.py`: the `Book` record in Python, as a dataclass. 30
Listing 1.18 `catalog_1d.py`: the catalog as a Python list. 31
Listing 1.19 `bounds.cpp`: indexing inside the bounds, and the checked accessor `at()` refusing an index that is out of range. 32
Listing 1.20 `Bounds.java`: Java checks every index, for arrays and for `ArrayList`. 33
Listing 1.21 `bounds.py`: Python checks every index, and also accepts negative ones. 33
Listing 1.22 `array_ops.cpp`: sum, position of the maximum, linear search, counting and in-place reversal of an array. 34
Listing 1.23 `ArrayOps.java`: the five traversals in Java. 35
Listing 1.24 `array_ops.py`: the five traversals in Python. 36
Listing 1.25 `shelves_2d.cpp`: a two-dimensional array of `Book`s, and the row-major formula checked for every element. 38
Listing 1.26 `Shelves2D.java`: a Java array of row arrays. 40
Listing 1.27 `shelves_2d.py`: a Python list of row lists. 41
Listing 1.28 `loan_table.cpp`: a 3 × 4 table of loans with row totals, column totals and a grand total. 42
Listing 1.29 `LoanTable.java`: the loan table in Java. 43
Listing 1.30 `loan_table.py`: the loan table in Python. 44
Listing 1.31 `aliasing_rows.py`: `[[0] * 4] * 3` makes three references to one row; a list comprehension makes three rows. 45
Listing 1.32 `vector_growth.cpp`: a fixed-size `std::array`, and a `std::vector` doubling its capacity as it grows. 46
Listing 1.33 `ListGrowth.java`: a fixed-length Java array, and the growth steps of an `ArrayList` from empty to 100 elements. 47
Listing 1.34 `list_growth.py`: a tuple cannot grow; a list grows in steps, leaving room for future appends. 49
Listing 1.35 `compact_array.py`: 1,000 numbers in a list of references and in a compact `array('i')`. 50
Listing 1.36 `compact_array.cpp`: 1,000 `int32_t`s in a raw array and in a vector. 51
Listing 1.37 `CompactArray.java`: `int[]` stores values, `Integer[]` stores references. 51
Listing 1.38 `insert_shift.cpp`: insertion and deletion by shifting, with the number of moves counted. 54
Listing 1.39 `InsertShift.java`: insertion and deletion by shifting, in Java. 55
Listing 1.40 `insert_shift.py`: insertion and deletion by shifting, in Python. 57
Listing 1.41 `hello.cpp`: the first program; it proves that the compiler, the editor and the terminal work together. 63
Listing 1.42 `Hello.java`: the first program in Java. 64
Listing 1.43 `hello.py`: the first program in Python. 64
Listing 1.44 `sol_address3d.cpp`: Equation (1.9) checked against the real address of all 24 elements. 70
Listing 1.45 `SolAddress3D.java`: the three-dimensional position formula checked in Java. 71
Listing 1.46 `sol_address3d.py`: the three-dimensional position formula checked in Python. 71
Listing 1.47 `sol_rotate.cpp`: rotation by three reversals. 72
Listing 1.48 `SolRotate.java`: rotation by three reversals, in Java. 73
Listing 1.49 `sol_rotate.py`: rotation by three reversals, in Python. 74

Listing 1.50 `sol_transpose.cpp`: transposing the loan table. 75
Listing 1.51 `SolTranspose.java`: transposing the loan table, in Java. 75
Listing 1.52 `sol_transpose.py`: transposing the loan table, in Python. 75
Listing 1.53 `sol_padding.cpp`: offsets and size of a `Book` with an `int` field. 77
Listing 2.1 `sequential_search.cpp`: sequential search with a flag, with early exit, for counting, and on records. 82
Listing 2.2 `SequentialSearch.java`: the same searches in Java. 85
Listing 2.3 `sequential_search.py`: the same searches in Python. 88
Listing 2.4 `sentinel_search.cpp`: the plain loop and the sentinel loop, with every loop test counted. 93
Listing 2.5 `SentinelSearch.java`: the same comparison in Java. 94
Listing 2.6 `sentinel_search.py`: the same comparison in Python. 95
Listing 2.7 `binary_search.cpp`: iterative binary search with a trace, and the recursive form. 100
Listing 2.8 `BinarySearch.java`: the same program in Java. 102
Listing 2.9 `binary_search.py`: the same program in Python. 104
Listing 2.10 `bounds.cpp`: lower bound, upper bound, a non-terminating binary search, and the overflowing middle. 109
Listing 2.11 `Bounds.java`: the same program in Java, where the overflow happens in a real `int`. 111
Listing 2.12 `bounds.py`: the same program in Python. 113
Listing 2.13 `interpolation_search.cpp`: interpolation search on the computing shelf, printing each estimate. 117
Listing 2.14 `InterpolationSearch.java`: the same program in Java. `Locale.ROOT` makes the decimal point a point on every computer. 118
Listing 2.15 `interpolation_search.py`: the same program in Python. 119
Listing 2.16 `jump_search.cpp`: jump search on sixteen call numbers, and the worst case for several n. 123
Listing 2.17 `JumpSearch.java`: the same program in Java. 125
Listing 2.18 `jump_search.py`: the same program in Python. 126
Listing 2.19 `search_2d.cpp`: addresses in a two-dimensional array, a scan, the staircase, and binary search in row-major order. 129
Listing 2.20 `Search2D.java`: the same program in Java. 131
Listing 2.21 `search_2d.py`: the same program in Python, with a list of lists as the grid. 134
Listing 2.22 `measure_catalog.cpp`: five searches on a catalog of ten thousand real books, with probes counted and time measured. 137
Listing 2.23 `MeasureCatalog.java`: the same experiment in Java. 142
Listing 2.24 `measure_catalog.py`: the same experiment in Python. 146
Listing 2.25 `scaling.cpp`: average probes as n grows, on generated uniform keys and on cubic keys. 152
Listing 2.26 `Scaling.java`: the same experiment in Java. The helper `fixed` rounds the exact binary value, as C and Python do, so that the three tables agree digit for digit. 155
Listing 2.27 `scaling.py`: the same experiment in Python. 158
Listing 2.28 `library_search.cpp`: `std::find`, `std::binary_search`, `std::lower_bound`, `std::upper_bound` and `std::equal_range`, including a search of records by code. 163
Listing 2.29 `LibrarySearch.java`: `indexOf`, `Arrays.binarySearch` and `Collections.binarySearch`, including the negative return value and a search of records with a comparator. 164
Listing 2.30 `library_search.py`: `in`, `list.index` and the `bisect` module, including a search of records with `key=`. 166
Listing 2.31 `laundry_orders.cpp`: Profitina Laundry's orders searched by time range, by number, by nearest time and by name. 170
Listing 2.32 `LaundryOrders.java`: the same program in Java. 172
Listing 2.33 `laundry_orders.py`: the same program in Python. 174
Listing 2.34 `fibonacci_exponential.cpp`: Fibonacci search and exponential search. 182

Listing 2.35 `FibonacciExponential.java`: the same two searches in Java. 183
Listing 2.36 `fibonacci_exponential.py`: the same two searches in Python. 185
Listing 3.1 `swap_demo.cpp`: exchanging two `int`s through pointers, and exchanging two elements of an array. 193
Listing 3.2 `SwapDemo.java`: a method cannot exchange its caller's `int` variables, but it can exchange two slots of an array. 194
Listing 3.3 `swap_demo.py`: exchanging by tuple assignment, for two names and for two list elements. 194
Listing 3.4 `bubble_sort.cpp`: bubble sort with early exit, ascending or descending, with counts of comparisons and swaps. 198
Listing 3.5 `BubbleSort.java`: the same bubble sort in Java. 199
Listing 3.6 `bubble_sort.py`: the same bubble sort in Python. 200
Listing 3.7 `exchange_sort.cpp`: exchange sort, printing each swap and the array after each round. 203
Listing 3.8 `ExchangeSort.java`: exchange sort in Java. 204
Listing 3.9 `exchange_sort.py`: exchange sort in Python. 205
Listing 3.10 `selection_sort.cpp`: selection sort, reporting where each round found its minimum. 207
Listing 3.11 `SelectionSort.java`: selection sort in Java. 208
Listing 3.12 `selection_sort.py`: selection sort in Python. 209
Listing 3.13 `insertion_sort.cpp`: insertion sort with counts of comparisons and shifts, on random, sorted and reversed input. 212
Listing 3.14 `InsertionSort.java`: insertion sort in Java. 212
Listing 3.15 `insertion_sort.py`: insertion sort in Python. 213
Listing 3.16 `shell_sort.cpp`: Shell sort with three gap sequences, compared on 100,000 random numbers 216
Listing 3.17 `ShellSort.java`: Shell sort in Java. `>>>` is Java's unsigned shift, which the generator needs. 218
Listing 3.18 `shell_sort.py`: Shell sort in Python. Python integers never overflow, so the generator reduces the result modulo 2 to the power 64 by hand. 220
Listing 3.19 `merge_sort.cpp`: top-down merge sort with one shared buffer, printing every merge. 224
Listing 3.20 `MergeSort.java`: merge sort in Java. 225
Listing 3.21 `merge_sort.py`: merge sort in Python. 226
Listing 3.22 `quick_sort.cpp`: quicksort with Lomuto's and Hoare's partitions, and an experiment with three pivot rules. 231
Listing 3.23 `QuickSort.java`: quicksort in Java; the pivot experiment runs on a thread with a larger stack. 233
Listing 3.24 `quick_sort.py`: quicksort in Python. 237
Listing 3.25 `heap_sort.cpp`: heap sort on the running array, and on an array that has to be turned into a heap first. 243
Listing 3.26 `HeapSort.java`: heap sort in Java. 244
Listing 3.27 `heap_sort.py`: heap sort in Python. 246
Listing 3.28 `counting_radix.cpp`: a stable counting sort parameterized by a key function, and LSD radix sort built on it. 250
Listing 3.29 `CountingRadix.java`: counting and radix sort in Java, with `IntUnaryOperator` as the key function. 251
Listing 3.30 `counting_radix.py`: counting and radix sort in Python, with a `lambda` as the key function. 252
Listing 3.31 `stability_demo.cpp`: the catalog sorted by category with a stable and an unstable sort. The comparison is passed as a function pointer. 254
Listing 3.32 `StabilityDemo.java`: the same demonstration in Java, with a `record` for the book and a `Comparator` for the key. 255
Listing 3.33 `stability_demo.py`: the same demonstration in Python, with a `dataclass` and a key function. 256

Listing 3.34 `sort_benchmark.cpp`: every sort of the chapter on a thousand to a million random integers, with counts and times. 259
Listing 3.35 `SortBenchmark.java`: the benchmark in Java, including `Arrays.sort` on primitives and on objects. 268
Listing 3.36 `sort_benchmark.py`: the benchmark in Python; the quadratic sorts stop at ten thousand elements. 274
Listing 3.37 `library_sorts.cpp`: `std::sort` with and without a comparator, `std::partial_sort`, and `std::stable_sort` with a lambda. 281
Listing 3.38 `LibrarySorts.java`: `Arrays.sort` on `int[]` and on `Integer[]` with `Collections.reverseOrder()`, and `List.sort` with `Comparator.comparing`. 282
Listing 3.39 `library_sorts.py`: `list.sort`, `sorted` with a key, and `heapq.nlargest` 283
Listing 3.40 `laundry_orders.cpp`: Profitina Laundry's washing queue with `std::stable_sort`, and its pickup list with `std::sort`. 286
Listing 3.41 `LaundryOrders.java`: the same queue with `List.sort`, which is stable, and `Comparator.comparingDouble(...).reversed()`. 286
Listing 3.42 `laundry_orders.py`: the same queue with `sorted(..., reverse=True)`. 287
Listing 3.43 `cocktail_sort.cpp`: cocktail shaker sort, the solution of Exercise 3.4. 292
Listing 3.44 `CocktailSort.java`: cocktail shaker sort in Java. 293
Listing 3.45 `cocktail_sort.py`: cocktail shaker sort in Python. 294
Listing 3.46 `count_inversions.cpp`: inversions counted by merge sort, checked against the quadratic count from the definition. Solution of Exercise 3.14. 296
Listing 3.47 `CountInversions.java`: the inversion count in Java. 297
Listing 3.48 `count_inversions.py`: the inversion count in Python. 298
Listing 4.1 `sort_words.cpp`: insertion sort with a dictionary-order comparison. 308
Listing 4.2 `SortWords.java`: the same insertion sort in Java. 310
Listing 4.3 `sort_words.py`: the same insertion sort in Python. 311
Listing 4.4 `wrong_sort_words.cpp`: the same insertion sort with the plain `<` on strings. 312
Listing 4.5 `first_repeat.cpp`: two passes over the string with 256 counters. 315
Listing 4.6 `FirstRepeat.java`: the same two passes in Java, with one counter per Java `char`. 317
Listing 4.7 `first_repeat.py`: the same two passes in Python, with a dictionary of counters. 317
Listing 4.8 `wrong_first_repeat.cpp`: stopping at the first character that was seen before. 318
Listing 4.9 `search_money.cpp`: sequential search over an array of `Person` records. 321
Listing 4.10 `SearchMoney.java`: the same search with a small `Person` class. 323
Listing 4.11 `search_money.py`: the same search over a list of (name, money) pairs. 323
Listing 4.12 `wrong_search_money.cpp`: binary search on data that is not sorted by name. 324
Listing 4.13 `importance_sort.cpp`: a two-key comparison rule inside a stable insertion sort. 327
Listing 4.14 `ImportanceSort.java`: the same rule and sort in Java. 329
Listing 4.15 `importance_sort.py`: the same rule and sort in Python. 330
Listing 4.16 `wrong_importance_sort.py`: two stable sorts, applied in the wrong order. 331
Listing 4.17 `array_address.cpp`: the address formulas, checked against the distances in real memory. 334
Listing 4.18 `ArrayAddress.java`: the address formulas in Java. 335
Listing 4.19 `array_address.py`: the address formulas in Python. 335
Listing 4.20 `wrong_array_address.py`: forgetting the element size, and multiplying by the number of rows. 336
Listing 4.21 `count_range.cpp`: lower bound, upper bound, and their difference. 338
Listing 4.22 `CountRange.java`: the two boundary searches in Java. 340
Listing 4.23 `count_range.py`: the two boundary searches in Python. 340
Listing 4.24 `WrongCountRange.java`: the search range starts as $[0, n-1]$, so the answer n can never be returned. 341
Listing 4.25 `interpolation_search.cpp`: interpolation search with a printed trace, next to a binary search that counts its probes. 344

Listing 4.26 `InterpolationSearch.java`: the same search in Java. 346
Listing 4.27 `interpolation_search.py`: the same search in Python, with the trace collected in a string. 348
Listing 4.28 `wrong_interpolation.py`: interpolation search without the range guard. 350
Listing 4.29 `trace_sorts.cpp`: the three elementary sorts, each printing the array after every pass and its counts. 352
Listing 4.30 `TraceSorts.java`: the same three traced sorts in Java. 355
Listing 4.31 `trace_sorts.py`: the same three traced sorts in Python. 357
Listing 4.32 `WrongBubble.java`: the inner loop of bubble sort runs one step too far. 359
Listing 4.33 `merge_sorted.cpp`: the merge of two sorted arrays, counting comparisons. 361
Listing 4.34 `MergeSorted.java`: the same merge in Java, returning a new array. 362
Listing 4.35 `merge_sorted.py`: the same merge in Python, with the tails added by `extend`. 363
Listing 4.36 `wrong_merge.cpp`: a merge that forgets the leftover elements. 364
Listing 4.37 `delete_value.cpp`: deleting every occurrence of a value with a read index and a write index. 367
Listing 4.38 `DeleteValue.java`: the same two-index deletion in Java. 369
Listing 4.39 `delete_value.py`: the same two-index deletion in Python. 369
Listing 4.40 `wrong_delete_value.py`: shift-left deletion that always advances the index. 370
Listing 4.41 `pair_sum.cpp`: sort the prices, then find every matching pair with a two-index walk. 372
Listing 4.42 `PairSum.java`: the same sort and two-index walk in Java. 374
Listing 4.43 `pair_sum.py`: the same sort and two-index walk in Python. 375
Listing 4.44 `wrong_pair_sum.cpp`: the two-index walk on prices that were never sorted. 376
Listing 5.1 `01_hanoi_stacks.cpp`: the Towers of Hanoi solved with three stacks and a loop, then the move count for 1 to 10 disks. 386
Listing 5.2 `02_array_stack.cpp`: an array-based stack template in C++, used as Pustaka Ceria's book return cart. 392
Listing 5.3 `ArrayStackDemo.java`: the same array stack as a Java generic class. 395
Listing 5.4 `02_array_stack.py`: the same array stack in Python, with a fixed-size list and an explicit top index. 397
Listing 5.5 `03_growing_stack.cpp`: a stack that doubles its array when it fills, counting every element copy. 400
Listing 5.6 `04_linked_stack.cpp`: a preview of a stack built from linked nodes. 402
Listing 5.7 `05_brackets.cpp`: bracket matching with `std::stack`, reporting the position and kind of the first error. 405
Listing 5.8 `06_undo_redo.cpp`: undo and redo with two stacks of saved states. 407
Listing 5.9 `07_calculator.cpp`: a calculator that converts infix to postfix with the shunting-yard algorithm and evaluates the postfix with an operand stack. 413
Listing 5.10 `08_call_stack.cpp`: making the call stack visible by pushing a name on every entry and popping it on every return. 419
Listing 5.11 `09_linear_queue.cpp`: the shifting queue (correct but $O(n)$ per dequeue) and the drifting queue ($O(1)$ but falsely full). 424
Listing 5.12 `10_circular_queue.cpp`: a circular queue with a count, as Pustaka Ceria's new-arrivals line. 429
Listing 5.13 `CircularQueueDemo.java`: the circular queue as a Java generic class. 431
Listing 5.14 `10_circular_queue.py`: the circular queue in Python. 432
Listing 5.15 `11_circular_queue_slot.cpp`: a circular queue that leaves one slot empty and computes its size from the two indices. 435
Listing 5.16 `12_array_deque.cpp`: a circular-array deque, and a palindrome test that pops from both ends. 438
Listing 5.17 `13_library.cpp`: `std::stack`, `std::queue`, `std::deque` and `std::priority_queue`. 441
Listing 5.18 `Library.java`: `ArrayDeque` as a stack, as a queue and as a deque, and `PriorityQueue` in both orders. 443

Listing 5.19 `13_library.py`: a list as a stack, `collections.deque` as a queue and deque, and `heapq` as a priority queue. 444
Listing 6.1 `01_first_pointer.cpp`: taking an address with `&`, storing it in a pointer, and following the pointer with `*`. 459
Listing 6.2 `02_pointer_assign.cpp`: `q = p` copies an address, `*q = *p` copies a value. 462
Listing 6.3 `03_pointer_arith.cpp`: each step of `p + i` moves `sizeof(T)` bytes, and pointer subtraction counts elements. 464
Listing 6.4 `04_array_walk.cpp`: jumping, stepping and writing through a pointer, then the one-past-the-end loop. 467
Listing 6.5 `05_danaya.cpp`: walking a C string forward and backward with a pointer; `overrun` repeats the classic mistake. 469
Listing 6.6 `06_array_param.cpp`: an array argument arrives as a pointer; a count, a pointer range, or a container supplies the missing length. 470
Listing 6.7 `07_book_pointer.cpp`: the arrow operator on an array of `Book`, and a first chain of nodes. 472
Listing 6.8 `08_dynamic.cpp`: single objects and arrays on the heap, and a growable array that doubles its capacity. 476
Listing 6.9 `09_mem_bugs.cpp`: seven classic pointer bugs, each selected by a command-line word, and their corrected forms. 480
Listing 6.10 `10_nullptr.cpp`: `nullptr` picks the pointer overload, and a search returns `nullptr` for "not found". 486
Listing 6.11 `11_references.cpp`: a reference is an alias; assigning to it assigns to the object, and a `for` loop over references changes the container. 487
Listing 6.12 `12_functions_scope.cpp`: prototypes, formal and actual parameters, global, local and static variables, and a first comparison of passing by value and by pointer. 490
Listing 6.13 `13_swap.cpp`: swapping by value (fails), by pointer and by reference (both work), and `std::swap`. 491
Listing 6.14 `14_const_correct.cpp`: the four `const` forms, a const member function, and the cost of copying measured by a copy counter. 495
Listing 6.15 `15_find_book.cpp`: returning a pointer into the caller's array or `nullptr`, a reference into the caller's vector, and a new value. 497
Listing 6.16 `16_unique_ptr.cpp`: creating, moving, handing over and collecting `std::unique_ptr` owners; no `delete` appears anywhere. 500
Listing 6.17 `17_shared_ptr.cpp`: a book shared by two lists and destroyed when the last owner lets go, and a pair of members linked with `weak_ptr` so that they can be destroyed. 503
Listing 6.18 `RefsDemo.java`: aliasing, mutation through a parameter, rebinding a parameter, a primitive argument, a real copy, `==` versus `equals`, and a null reference. 506
Listing 6.19 `refs_demo.py`: the same experiment with Python names, plus the mutable-default-argument trap. 508
Listing 6.20 `SwapDemo.java`: Java can swap the contents of two objects, or two slots of one array, but never two of the caller's variables. 510
Listing 6.21 `swap_demo.py`: the same attempts in Python, ending with the tuple assignment that swaps in the caller. 511
Listing 6.22 `18_laundry.cpp`: Profitina Laundry's queues own their orders through `unique_ptr`; promotion moves ownership, not the order. 514
Listing 6.23 `laundry.py`: the same queues in Python, where moving an order between queues moves a reference. 516
Listing 6.24 `19_sol_functions.cpp`: solution to Exercise 6.12. 521
Listing 6.25 `20_sol_pointers.cpp`: solutions to Exercises 6.10 and 6.15. 523
Listing 7.1 `node_basics.cpp`: three nodes created with `new`, linked by hand, walked, and released with `delete`. 532
Listing 7.2 `malloc_node.cpp`: a node allocated with `malloc` and released with `free`, next to one made with `new` and released with `delete`. 533

Listing 7.3 `sll_head.cpp`: the singly linked list with a head pointer, with every operation of Section 7.5. 541
Listing 7.4 `SllHead.java`: the head-reference singly linked list in Java. 544
Listing 7.5 `sll_head.py`: the head-reference singly linked list in Python. 547
Listing 7.6 `sll_ops.cpp`: search, length, insertion after a node and in sorted order, deletion by key, iterative and recursive reversal, and destruction. 557
Listing 7.7 `sll_dummy.cpp`: a singly linked list with a dummy head node; insertion and deletion need no special case for the first element. 561
Listing 7.8 `sll_link.cpp`: walking a pointer to the link (`Node **`) removes the special case for the first node without a dummy node. 564
Listing 7.9 `sll_head_tail.cpp`: a singly linked list with `head` and `tail`; back insertion is $O(1)$, back deletion still walks. 568
Listing 7.10 `csll.cpp`: a circular singly linked list reached through `last`, used for round-robin service with a 2-minute time slice. 572
Listing 7.11 `linked_stack_queue.cpp`: `LinkedStack` and `LinkedQueue` built on singly linked nodes, with destructors that release every node. 576
Listing 7.12 `memory_bugs.cpp`: a correct run and four deliberate memory bugs (`leak`, `uaf`, `dangle`, `double`). 581
Listing 7.13 `std_lists.cpp`: the singly linked `std::forward_list` and the doubly linked `std::list`. 588
Listing 7.14 `LibraryLists.java`: `java.util.LinkedList` used as a list and as a deque. 589
Listing 7.15 `library_lists.py`: `collections.deque`, Python's closest built-in to a linked list. 591
Listing 7.16 `bench.cpp`: front insertion and traversal timed for an array and a linked list. 592
Listing 7.17 `exercises.cpp`: solutions to Exercises 7.3 to 7.9 and 7.11. 600
Listing 8.1 `p801_stack.cpp`: the array stack of Problem 8.1, completed, with the given driver and our own edge-case checks. 613
Listing 8.2 `P801Stack.java`: Problem 8.1 in Java. 615
Listing 8.3 `p801_stack.py`: Problem 8.1 in Python, with a fixed block of `MAX` cells so that the logic matches the C++ array. 617
Listing 8.4 `p802_deque_walk.cpp`: Problem 8.2, the two-way list built by `slist_pushBack` and walked by `traverse`. 622
Listing 8.5 `P802DequeWalk.java`: Problem 8.2 in Java. 625
Listing 8.6 `p802_deque_walk.py`: Problem 8.2 in Python. `__slots__` fixes the fields of a node, which makes each node smaller and catches misspelled field names. 627
Listing 8.7 `p803_evens_odds.cpp`: Problem 8.3 with two sorted linked lists and one insertion function that receives the order as a function pointer. 632
Listing 8.8 `P803EvensOdds.java`: Problem 8.3 in Java, with the order passed as a lambda expression. 634
Listing 8.9 `p803_evens_odds.py`: Problem 8.3 in Python, with the order passed as a function. 635
Listing 8.10 `p804_reverse_words.cpp`: Problem 8.4 with a linked list used as a stack. 638
Listing 8.11 `P804ReverseWords.java`: Problem 8.4 in Java. 640
Listing 8.12 `p804_reverse_words.py`: Problem 8.4 in Python. 640
Listing 8.13 `p805_remove_duplicates.cpp`: Problem 8.5 with the two-pointer removal, plus a `slist_popBack` that drops the end marker. 644
Listing 8.14 `P805RemoveDuplicates.java`: Problem 8.5 in Java. 646
Listing 8.15 `p805_remove_duplicates.py`: Problem 8.5 in Python. 648
Listing 8.16 `p806_brackets.cpp`: Problem 8.6, bracket matching with an array stack whose capacity is chosen at run time. 651
Listing 8.17 `P806Brackets.java`: Problem 8.6 in Java. 653
Listing 8.18 `p806_brackets.py`: Problem 8.6 in Python. 655
Listing 8.19 `p807_circular_queue.cpp`: Problem 8.7, a circular array queue with `front` and `count`. 658
Listing 8.20 `P807CircularQueue.java`: Problem 8.7 in Java. 659

Listing 8.21 `p807_circular_queue.py`: Problem 8.7 in Python. 661
Listing 8.22 `p808_postfix.cpp`: Problem 8.8, the shunting-yard conversion and the postfix evaluation with `std::stack` 663
Listing 8.23 `P808Postfix.java`: Problem 8.8 in Java, with `ArrayDeque` as the stack. 665
Listing 8.24 `p808_postfix.py`: Problem 8.8 in Python, with a list as the stack. 667
Listing 8.25 `p809_three_ways.cpp`: Problem 8.9, the three ways to pass an argument in C++, with a check of which memory each parameter uses. 670
Listing 8.26 `P809ThreeWays.java`: Problem 8.9 in Java, where every argument is passed by value and object variables hold references. 672
Listing 8.27 `p809_three_ways.py`: Problem 8.9 in Python, where parameters are new names for the same objects. 674
Listing 8.28 `p810_reverse_list.cpp`: Problem 8.10, in-place reversal with three pointers, traced step by step. 677
Listing 8.29 `P810ReverseList.java`: Problem 8.10 in Java. 679
Listing 8.30 `p810_reverse_list.py`: Problem 8.10 in Python. 680
Listing 8.31 `p811_merge_sorted.cpp`: Problem 8.11, merging two sorted linked lists with a dummy node. 683
Listing 8.32 `P811MergeSorted.java`: Problem 8.11 in Java. 685
Listing 8.33 `p811_merge_sorted.py`: Problem 8.11 in Python. 687
Listing 8.34 `p812_linked_queue.cpp`: Problem 8.12, a queue built from a linked list with `head` and `tail` 690
Listing 8.35 `P812LinkedQueue.java`: Problem 8.12 in Java. 691
Listing 8.36 `p812_linked_queue.py`: Problem 8.12 in Python. 693
Listing 8.37 `ex08_solutions.cpp`: the programming exercises of Chapter 8 in C++. 700
Listing 8.38 `Ex08Solutions.java`: the programming exercises of Chapter 8 in Java. 703
Listing 8.39 `ex08_solutions.py`: the programming exercises of Chapter 8 in Python. 707
Listing 9.1 `dll_node.cpp`: three doubly linked nodes created with `new`, linked in both directions by hand, walked both ways and released. 716
Listing 9.2 `dll_head.cpp`: the doubly linked list with a head pointer, with every operation of Section 9.3. 724
Listing 9.3 `DllHead.java`: the head-reference doubly linked list in Java. 728
Listing 9.4 `dll_head.py`: the head-reference doubly linked list in Python. `__slots__` fixes the three fields of a node. 731
Listing 9.5 `dll_ops.cpp`: the doubly linked list with head and tail: front and back insertion and deletion, insertion after and before a node, sorted insertion, search, deletion by pointer and by key, both traversals, and a link check. 739
Listing 9.6 `dll_headnode.cpp`: a doubly linked list with a head node; each operation has one special case left, at the back end. 745
Listing 9.7 `cdll.cpp`: a circular doubly linked list reached through `head`, used as the lobby carousel; `next` and `previous` wrap around at both ends. 749
Listing 9.8 `cdll_sentinel.cpp`: a circular doubly linked list with a sentinel; `insertAfter` and `deleteNode` contain no special cases, and every other operation is built on them. 755
Listing 9.9 `deque.cpp`: a deque of books on a circular doubly linked list with a sentinel, used at the Pustaka Ceria returns desk. 759
Listing 9.10 `recently_borrowed.cpp`: the "recently borrowed" panel, an LRU list built from a circular doubly linked list with a sentinel and an `std::unordered_map` from title to node. 764
Listing 9.11 `bench.cpp`: emptying a list from the back, and deleting nodes through pointers the program already holds, for singly and doubly linked lists. 767
Listing 9.12 `memory_bugs.cpp`: a correct run and three deliberate bugs of doubly linked lists (`halflink`, `stale`, `leak`) 772
Listing 9.13 `std_dll.cpp`: `std::list` walked in both directions, with $O(1)$ insertion and erasure at an iterator, stable iterators and `splice` 778
Listing 9.14 `LibraryDll.java`: `LinkedList` edited through a `ListIterator` and walked backward, and a `LinkedHashMap` in access order used as an LRU list of capacity 3. 779

Listing 9.15 `library_dll.py`: `collections.deque` (with `rotate` and `maxlen`), an `OrderedDict` used as an LRU list, and `functools.lru_cache`. 781
Listing 9.16 `laundry_weeks.cpp`: Profitina Laundry's weekly order totals in a doubly linked list, browsed one week back or forward from any week. 784
Listing 9.17 `exercises.cpp`: solutions to Exercises 9.3 to 9.9. 789
Listing 10.1 `01_recursive_f.cpp`: the recursive definition of Equation (10.1) as a C++ function, traced, and checked against its closed form. 800
Listing 10.2 `02_factorial.cpp`: factorial with a loop and with recursion, a trace of `fact(4)`, and the limit of 64-bit arithmetic. 804
Listing 10.3 `Factorial.java`: the same program in Java; `BigInteger` gives the exact 21! 806
Listing 10.4 `02_factorial.py`: the same program in Python, where integers grow as needed. 807
Listing 10.5 `03_fibonacci.cpp`: the rabbits, and $F(n)$ by naive recursion, memoization, a bottom-up table, two variables, tail recursion, fast doubling and Binet's formula. 815
Listing 10.6 `05_gcd.cpp`: Euclid's algorithm recursively and iteratively, traced, and its worst case. 822
Listing 10.7 `06_power.cpp`: exponentiation by n multiplications and by repeated squaring, with call counts, and modular exponentiation. 827
Listing 10.8 `07_hanoi.cpp`: the recursive Towers of Hanoi, with every move checked on three real stacks. 830
Listing 10.9 `Hanoi.java`: the recursive Towers of Hanoi in Java. 832
Listing 10.10 `07_hanoi.py`: the recursive Towers of Hanoi in Python. 834
Listing 10.11 `08_arrays_lists.cpp`: recursion over an array (peeling one element or halving) and over a singly linked list (the first node and the rest). 838
Listing 10.12 `09_tail.cpp`: a non-tail recursion, a tail recursion, and a recursion whose frames must survive, selectable from the command line. 844
Listing 10.13 `TailCalls.java`: in Java a tail-recursive method uses one frame per call, exactly like any other. 847
Listing 10.14 `09_tail.py`: in Python a tail-recursive function hits the recursion limit like any other. 848
Listing 10.15 `10_depth.cpp`: measuring the size of a stack frame and recursing to a chosen depth. 850
Listing 10.16 `Depth.java`: the recursion depth reachable in Java's main thread, and a deep recursion in a thread with a larger stack. 851
Listing 10.17 `10_depth.py`: Python's recursion limit, raising it, and a deep recursion in a thread with a larger stack. 853
Listing 10.18 `11_explicit_stack.cpp`: the Towers of Hanoi with an explicit stack of frames, and a million-node list visited backward. 856
Listing 10.19 `12_laundry_menu.cpp`: recursive functions over Profitina Laundry's nested service menu, and the same count with an explicit stack. 860
Listing 10.20 `13_exercises.cpp`: solutions to the programming exercises of Chapter 10. 865
Listing 11.1 `array_tree.cpp`: a binary tree stored in an array; children and parents are found by index arithmetic. 879
Listing 11.2 `traversals_iter.cpp`: preorder, inorder and postorder with an explicit stack, and level order with a queue built from linked nodes, with traces of the stack and the queue. 889
Listing 11.3 `expr_tree.cpp`: an expression tree built from postfix notation, printed in prefix, infix and postfix, and evaluated by a postorder traversal. 895
Listing 11.4 `bst.cpp`: a binary search tree with insertion, the four traversals, search, count, leaf count, height, minimum, maximum, leaf listing, sideways printing, and release of all nodes. 902
Listing 11.5 `Bst.java`: the binary search tree of **Listing 11.4** in Java; `insert` returns the new root of each subtree. 908
Listing 11.6 `bst.py`: the same binary search tree in Python. 912
Listing 11.7 `bst_delete.cpp`: successor and predecessor without parent pointers, and deletion in all three cases with the successor and the predecessor variant. 921
Listing 11.8 `height_experiment.cpp`: the height and average depth of binary search trees built from random and from sorted insertion orders, with the time to build them. 928
Listing 11.9 `avl_first_look.cpp`: AVL insertion with single and double rotations, fed the sorted keys that turn a plain binary search tree into a chain. 933

Listing 11.10 `general_tree.cpp`: a general tree in first-child/next-sibling form, its conversion to a binary tree, and trees rebuilt from preorder listings and degrees. 938
Listing 11.11 `std_map.cpp`: the Pustaka Ceria catalog in `std::map`, with in-order listing, search, a range query, successor and predecessor, minimum and maximum, and deletion; and a `std::set` that rejects a duplicate. 943
Listing 11.12 `CatalogTreeMap.java`: the same catalog in `java.util.TreeMap`, with `subMap`, `higherKey`, `lowerKey`, `firstKey` and `lastKey`; and a `TreeSet`. 945
Listing 11.13 `library_trees.py`: the catalog kept in order with a sorted list and the `bisect` module, with a `dict` for the codes. 947
Listing 11.14 `sorted_catalog.py`: the catalog in a `SortedDict` from the `sortedcontainers` package. 948
Listing 11.15 `laundry_tree.cpp`: Profitina Laundry's pickup board, a binary search tree of orders keyed by ready time, with a pruned range query, ceiling, floor and deletion. 950
Listing 11.16 `exercises.cpp`: solutions to Exercises 11.6 and 11.8 to 11.11. 958
Listing 12.1 `tree_kit.hpp`: the C++ toolkit shared by the tree programs of this chapter. 971
Listing 12.2 `TreeKit.java`: the Java toolkit. 974
Listing 12.3 `tree_kit.py`: the Python toolkit. 975
Listing 12.4 `p1200_sample_tree.cpp`: the tree of this chapter, built and inspected. 977
Listing 12.5 `P1200SampleTree.java`: the tree inspected in Java. 978
Listing 12.6 `p1200_sample_tree.py`: the tree inspected in Python. 979
Listing 12.7 `p1201_depth_first.cpp`: Problem 12.1, recursively and with an explicit stack. 981
Listing 12.8 `P1201DepthFirst.java`: Problem 12.1 in Java. 982
Listing 12.9 `p1201_depth_first.py`: Problem 12.1 in Python. 982
Listing 12.10 `p1202_kth_level.cpp`: Problem 12.2, printing every level from 0 to one past the height. 985
Listing 12.11 `P1202KthLevel.java`: Problem 12.2 in Java. 986
Listing 12.12 `p1202_kth_level.py`: Problem 12.2 in Python. 987
Listing 12.13 `p1203_sum.cpp`: Problem 12.3, one line of recursion. 989
Listing 12.14 `P1203Sum.java`: Problem 12.3 in Java. 990
Listing 12.15 `p1203_sum.py`: Problem 12.3 in Python. 990
Listing 12.16 `p1204_triple.cpp`: Problem 12.4, changing the nodes through the pointer. 992
Listing 12.17 `P1204Triple.java`: Problem 12.4 in Java. 993
Listing 12.18 `p1204_triple.py`: Problem 12.4 in Python. 994
Listing 12.19 `p1205_mirror.cpp`: Problem 12.5, a swap with a saved pointer at every node. 996
Listing 12.20 `P1205Mirror.java`: Problem 12.5 in Java. 997
Listing 12.21 `p1205_mirror.py`: Problem 12.5 in Python, with the swap written as one tuple assignment. 998
Listing 12.22 `p1206_balanced.cpp`: Problem 12.6, the helper that uses `height`, and a one-pass version. 1001
Listing 12.23 `P1206Balanced.java`: Problem 12.6 in Java. 1003
Listing 12.24 `p1206_balanced.py`: Problem 12.6 in Python. 1004
Listing 12.25 `p1207_delete.cpp`: Problem 12.7, deletion in all three cases, with every result re-attached by the parent. 1008
Listing 12.26 `P1207Delete.java`: Problem 12.7 in Java. 1009
Listing 12.27 `p1207_delete.py`: Problem 12.7 in Python. 1010
Listing 12.28 `p1208_find_parent.cpp`: Problem 12.8, the search-tree version and the any-tree version. 1013
Listing 12.29 `P1208FindParent.java`: Problem 12.8 in Java. 1014
Listing 12.30 `p1208_find_parent.py`: Problem 12.8 in Python. 1015
Listing 12.31 `p1209_find_child.cpp`: Problem 12.9, one call of `search` and two careful looks. 1017
Listing 12.32 `P1209FindChild.java`: Problem 12.9 in Java. 1018
Listing 12.33 `p1209_find_child.py`: Problem 12.9 in Python. 1019

Listing 12.34 `p1210_to_dll.cpp`: Problem 12.10, an inorder walk that appends every node to the list. 1022
Listing 12.35 `P1210ToDll.java`: Problem 12.10 in Java, with `head` and `prev` kept in one shared object. 1024
Listing 12.36 `p1210_to_dll.py`: Problem 12.10 in Python. 1025
Listing 12.37 `p1211_reverse_dll.cpp`: Problem 12.11, swapping the two arrows of every node. 1029
Listing 12.38 `P1211ReverseDll.java`: Problem 12.11 in Java. 1030
Listing 12.39 `p1211_reverse_dll.py`: Problem 12.11 in Python. 1032
Listing 12.40 `p1212_josephus.cpp`: Problem 12.12, a circular list and the recursive formula, compared. 1036
Listing 12.41 `P1212Josephus.java`: Problem 12.12 in Java. 1037
Listing 12.42 `p1212_josephus.py`: Problem 12.12 in Python. 1038
Listing 12.43 `p1213_choose_books.cpp`: Problem 12.13, every choice of three books by take-or-skip recursion with backtracking. 1041
Listing 12.44 `P1213ChooseBooks.java`: Problem 12.13 in Java. 1042
Listing 12.45 `p1213_choose_books.py`: Problem 12.13 in Python. 1043
Listing 12.46 `p1214_is_bst.cpp`: Problem 12.14, every node checked against the interval inherited from its ancestors. 1045
Listing 12.47 `P1214IsBst.java`: Problem 12.14 in Java. 1046
Listing 12.48 `p1214_is_bst.py`: Problem 12.14 in Python. 1047
Listing 12.49 `p1215_lca.cpp`: Problem 12.15, one walk down until the two values part. 1049
Listing 12.50 `P1215Lca.java`: Problem 12.15 in Java. 1050
Listing 12.51 `p1215_lca.py`: Problem 12.15 in Python. 1051
Listing 12.52 `p1216_levels.cpp`: Problem 12.16, a level-order walk that records the width of each level. 1052
Listing 12.53 `P1216Levels.java`: Problem 12.16 in Java. 1053
Listing 12.54 `p1216_levels.py`: Problem 12.16 in Python. 1054
Listing 12.55 `ex12_solutions.cpp`: the programming exercises of Chapter 12 in C++. 1059
Listing 12.56 `Ex12Solutions.java`: the programming exercises of Chapter 12 in Java. 1061
Listing 12.57 `ex12_solutions.py`: the programming exercises of Chapter 12 in Python. 1063
Listing 13.1 `graph_repr.cpp`: the campus map as an adjacency matrix, an adjacency list and an edge list. 1077
Listing 13.2 `GraphRepr.java`: the three representations in Java. 1080
Listing 13.3 `graph_repr.py`: the three representations in Python. 1082
Listing 13.4 `bfs.cpp`: breadth-first search on the campus map, with a trace, hop distances, parents and fewest-hop routes. 1088
Listing 13.5 `Bfs.java`: breadth-first search in Java, using `ArrayDeque` as the queue. 1090
Listing 13.6 `bfs.py`: breadth-first search in Python, using `collections.deque` as the queue. 1092
Listing 13.7 `dfs.cpp`: recursive and iterative depth-first search with discovery and finish times, and cycle detection in an undirected graph. 1099
Listing 13.8 `Dfs.java`: depth-first search, recursive and iterative, in Java. 1102
Listing 13.9 `dfs.py`: depth-first search, recursive and iterative, in Python. 1104
Listing 13.10 `components.cpp`: connected components of the campus map, before and after the shuttle road. 1108
Listing 13.11 `Components.java`: connected components in Java. 1110
Listing 13.12 `components.py`: connected components in Python. 1112
Listing 13.13 `topo.cpp`: topological sort by Kahn's algorithm and by depth-first search, with cycle detection. 1117
Listing 13.14 `Topo.java`: topological sort in Java. 1120
Listing 13.15 `topo.py`: topological sort in Python. 1122
Listing 13.16 `dijkstra.cpp`: Dijkstra's algorithm on the campus map with `std::priority_queue` as a min-heap. 1128

Listing 13.17 `Dijkstra.java`: Dijkstra's algorithm with `java.util.PriorityQueue` 1131
Listing 13.18 `dijkstra.py`: Dijkstra's algorithm with the `heapq` module 1132
Listing 13.19 `laundry_site.cpp`: click depth, inbound links and orphan pages of the Profitina Laundry website, by breadth-first search. 1137
Listing 13.20 `LaundrySite.java`: the website audit in Java 1139
Listing 13.21 `laundry_site.py`: the website audit in Python. 1140
Listing 13.22 `exercises.cpp`: solutions to Exercises 13.10 to 13.12. 1145
Listing 13.23 `Exercises.java`: solutions to Exercises 13.10 to 13.12 in Java. 1149
Listing 13.24 `exercises.py`: solutions to Exercises 13.10 to 13.12 in Python. 1151
Listing 14.1 `heap_index.cpp`: parent and children of every node of a ten-node complete tree, in the 0-based and 1-based numberings. 1161
Listing 14.2 `build_count.cpp`: key comparisons made by n insertions and by one bottom-up build, on ascending, pseudo-random and descending keys. 1175
Listing 14.3 `max_heap.cpp`: the complete `MaxHeap` class and a demonstration of every operation. 1178
Listing 14.4 `MaxHeap.java`: the `MaxHeap` class in Java, with an array that doubles when it is full. 1184
Listing 14.5 `max_heap.py`: the `MaxHeap` class in Python. 1188
Listing 14.6 `heap_sort_full.cpp`: heap sort with a trace of every state, and the same sort applied to book records to show that it is not stable. 1193
Listing 14.7 `library_pq.cpp`: `std::priority_queue` with a custom comparison serving hold requests, and the default max-queue and a `std::greater` min-queue of integers. 1198
Listing 14.8 `LibraryPQ.java`: `java.util.PriorityQueue` with a comparator built from `comparingInt`, `reversed` and `thenComparingInt`. 1200
Listing 14.9 `library_pq.py`: `heapq` with tuples as records, and the max-heap functions of Python 3.14 with a fallback for older versions. 1201
Listing 14.10 `topk_merge.cpp`: the three most borrowed books of a stream with a min-heap of size 3, and the merge of three sorted branch catalogs. 1204
Listing 14.11 `dijkstra_pq.cpp`: Dijkstra's algorithm on the delivery roads with `std::priority_queue` as a min-heap and lazy deletion. 1207
Listing 14.12 `exercises.cpp`: solutions to Exercises 14.9 to 14.12. 1213
Listing 15.1 `hash_functions.cpp`: the division and multiplication methods, the sum and polynomial string hashes, the rolling update and the avalanche measurement. 1226
Listing 15.2 `birthday.cpp`: exact and approximate collision probabilities, the number of keys for even and near-certain odds, and a simulation. 1233
Listing 15.3 `chaining.cpp`: a hash table with separate chaining built from singly linked nodes, with search, update, deletion and rehashing. 1238
Listing 15.4 `Chaining.java`: the chained hash table in Java. 1243
Listing 15.5 `chaining.py`: the chained hash table in Python. 1246
Listing 15.6 `open_addressing.cpp`: linear probing, quadratic probing and double hashing in a table of 16 slots, with deletion by tombstones. 1254
Listing 15.7 `OpenAddressing.java`: the three probe sequences and tombstones in Java 1259
Listing 15.8 `open_addressing.py`: the three probe sequences and tombstones in Python; `None` is not needed because the slot states are kept in their own list. 1262
Listing 15.9 `load_factor.cpp`: probes per search against the load factor, measured and predicted; the distribution of chain lengths; and the cost of rehashing by doubling. 1267
Listing 15.10 `flooding.cpp`: 2048 keys that all collide under the public 31-based polynomial hash, and a keyed polynomial hash with a secret base that spreads them again. 1274
Listing 15.11 `maps.cpp`: `std::map` against `std::unordered_map` on the same data, the bucket interface, range queries, a hash function for a two-field key, and a timing of one million keys. 1279
Listing 15.12 `LibraryMaps.java`: `HashMap`, `TreeMap` and `LinkedHashMap` on the same data, how a bin is chosen, a record as a key, 65,536 colliding keys, and a timing. 1284
Listing 15.13 `library_maps.py`: insertion order, integer and string hash values, hash randomization across interpreter runs, the growth of a `dict`, and a timing against binary search 1288

Listing 15.14 `applications.cpp`: counting the words of a catalog, detecting duplicate returns, two-sum on book prices, and an LRU cache of catalog records. 1291
Listing 15.15 `exercises.cpp`: checks for Exercises 15.1 to 15.4 and solutions to Exercises 15.6 to 15.9. 1299
Listing 16.1 `p1601_max_heap.cpp`: the max-heap class of Problem 16.1 with `extractMax`, `increaseKey`, `getMin`, `buildMaxHeap`, `isMaxHeap` and `clear`. 1316
Listing 16.2 `P1601MaxHeap.java`: Problem 16.1 in Java. 1323
Listing 16.3 `p1601_max_heap.py`: Problem 16.1 in Python. 1327
Listing 16.4 `p1602_map_vs_hash.cpp`: the same four keys in `std::map`, a search tree, and in `std::unordered_map`, a hash table. 1334
Listing 16.5 `p1602_itinerary.cpp`: Problem 16.2 with two hash maps, and a report for every way a pile of tickets can fail to form one chain. 1335
Listing 16.6 `P1602Itinerary.java`: Problem 16.2 in Java with two `HashMaps`. 1338
Listing 16.7 `p1602_itinerary.py`: Problem 16.2 in Python with two `dicts`. 1340
Listing 16.8 `p1603_bfs_hops.cpp`: breadth-first search from Pusat, with the fewest hops, the parent of every branch, and three routes. 1344
Listing 16.9 `P1603BfsHops.java`: Problem 16.3 in Java. 1347
Listing 16.10 `p1603_bfs_hops.py`: Problem 16.3 in Python. 1349
Listing 16.11 `p1604_reading_order.cpp`: Kahn's algorithm on the eleven topics of the book, with and without the arc that creates a cycle. 1353
Listing 16.12 `P1604ReadingOrder.java`: Problem 16.4 in Java. 1356
Listing 16.13 `p1604_reading_order.py`: Problem 16.4 in Python. 1357
Listing 16.14 `p1605_dijkstra.cpp`: Dijkstra's algorithm with a binary min-heap and lazy deletion, tracing every step. 1361
Listing 16.15 `P1605Dijkstra.java`: Problem 16.5 in Java. 1363
Listing 16.16 `p1605_dijkstra.py`: Problem 16.5 in Python. 1365
Listing 16.17 `p1606_top_k.cpp`: the k largest values with a min-heap of size k, traced on eight loan counts and run on the whole catalog in three orders. 1369
Listing 16.18 `P1606TopK.java`: Problem 16.6 in Java. 1372
Listing 16.19 `p1606_top_k.py`: Problem 16.6 in Python. 1374
Listing 16.20 `p1607_running_median.cpp`: the running median with a max-heap for the lower half and a min-heap for the upper half, checked against sorting after every value. 1378
Listing 16.21 `P1607RunningMedian.java`: Problem 16.7 in Java. 1380
Listing 16.22 `p1607_running_median.py`: Problem 16.7 in Python, with the lower half stored negated. 1382
Listing 16.23 `p1608_chaining.cpp`: a hash table with separate chaining, front insertion, removal through a pointer to the link, and rehashing when the load factor exceeds 1. 1386
Listing 16.24 `P1608Chaining.java`: Problem 16.8 in Java. 1389
Listing 16.25 `p1608_chaining.py`: Problem 16.8 in Python. 1391
Listing 16.26 `p1609_linear_probing.cpp`: open addressing with linear probing and tombstones, tracing every probe sequence. 1396
Listing 16.27 `P1609LinearProbing.java`: Problem 16.9 in Java. 1397
Listing 16.28 `p1609_linear_probing.py`: Problem 16.9 in Python. 1399
Listing 16.29 `p1610_choose.cpp`: six structures measured on the same catalog (build, lookup, range query, insert), the ten most-rated books by heap and by sorting, and the graph of people credited on the same book. 1404
Listing 16.30 `P1610Choose.java`: the experiment of Section 16.8 in Java, with `TreeMap` as the balanced search tree and `HashMap` as the hash table. 1413
Listing 16.31 `p1610_choose.py`: the experiment of Section 16.8 in Python, with `bisect` for the sorted array, our own nodes for the list and the tree, and `dict` as the hash table. 1423
Listing 16.32 `ex16_solutions.cpp`: solutions to the programming exercises of Chapter 16. 1438
Listing 16.33 `Ex16Solutions.java`: the same solutions in Java. 1442
Listing 16.34 `ex16_solutions.py`: the same solutions in Python. 1447

List of Pseudocode

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. 53
Pseudocode 1.2 `ARRAY-DELETE`: close the gap left by the element at index `pos` by moving the elements after it one place left. 53
Pseudocode 2.1 `SEQUENTIAL-SEARCH`: examine the elements from the front and stop at the first match. 81
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. 93
Pseudocode 2.3 `BINARY-SEARCH`, iterative, on the closed interval $[low, high]$. 98
Pseudocode 2.4 `BINARY-SEARCH-REC`: the recursive form. The first call is `BINARY-SEARCH-REC(A, x, 0, n − 1)`. 100
Pseudocode 2.5 `LOWER-BOUND`: the first position whose element is not less than x. Changing `<` to `≤` in line 5 gives `UPPER-BOUND`. 108
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. 117
Pseudocode 2.7 `JUMP-SEARCH`: jump over whole blocks, then walk through the single block that can contain the key. 122
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. 128
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. 197
Pseudocode 3.2 Exchange sort: position i is compared with every later position and takes any smaller value it meets. 203
Pseudocode 3.3 Selection sort: find the smallest remaining element, then move it into place with one exchange. 206
Pseudocode 3.4 Insertion sort: grow a sorted prefix by inserting one key at a time, shifting larger elements right. 210
Pseudocode 3.5 Shell sort: insertion sort on elements h apart, for a decreasing sequence of gaps ending with 1. 215
Pseudocode 3.6 `MERGE` and `MERGE-SORT`. The buffer T has the same length as A and is allocated once, before the first call. 223
Pseudocode 3.7 `QUICKSORT` with a partition that returns the pivot's final index. 228
Pseudocode 3.8 Lomuto's partition: one left-to-right scan that grows a region of elements no larger than the pivot. 229
Pseudocode 3.9 Hoare's partition: two indices move toward each other and exchange pairs that are on the wrong sides. 230
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. 241
Pseudocode 3.11 Counting sort for integer keys in $0..k-1$. No two keys are ever compared. 249
Pseudocode 4.1 Insertion sort of words; the comparison COMES-BEFORE implements dictionary order. 308
Pseudocode 4.2 The first repeated character in two passes, with one counter per possible character. 314
Pseudocode 4.3 Sequential search over an array of records; the result is an index, or −1 when the name is absent. 320
Pseudocode 4.4 A two-key comparison rule and the stable insertion sort that uses it. 327
Pseudocode 4.5 The address of an array element in one and two dimensions (row-major order). 333
Pseudocode 4.6 Binary search for the two boundaries of a block of equal keys in a sorted array. 337

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. 343
Pseudocode 4.8 Bubble sort, selection sort and insertion sort, as traced in Problem 4.8. 351
Pseudocode 4.9 Merging two sorted arrays into a third; at most one of the two copy loops does any work. 360
Pseudocode 4.10 Deleting every occurrence of x in one pass with a read index r and a write index w. 366
Pseudocode 4.11 The two-index walk that finds every pair with a given sum in a sorted array. 372
Pseudocode 5.1 The stack operations on an array $S[0..N-1]$ with index top; $top = -1$ means empty. 390
Pseudocode 5.2 Checking that brackets nest properly with one stack and one left-to-right scan. 404
Pseudocode 5.3 Dijkstra's shunting-yard algorithm: infix to postfix with one operator stack. 411
Pseudocode 5.4 Evaluating a postfix expression with one operand stack. 412
Pseudocode 5.5 `ENQUEUE` and `DEQUEUE` on a circular array of N slots with a `front` index and a `count`. 428
Pseudocode 6.1 `DYNAMIC-ARRAY-PUSH`: append `x`, first doubling the capacity when the array is full. 477
Pseudocode 7.1 `FRONT-INSERTION`: insert a new first node; two pointer assignments, in this order. 535
Pseudocode 7.2 `BACK-INSERTION` on a list with only a head pointer: walk to the last node, then attach. 537
Pseudocode 7.3 `TRAVERSE`: visit every node from the first to the last. 538
Pseudocode 7.4 `FRONT-DELETION`: remember the first node, move `head` past it, then release it. 539
Pseudocode 7.5 `BACK-DELETION` with only a head pointer: stop on the second-to-last node, cut the link, release the last node. 540
Pseudocode 7.6 `SEARCH`: return the first node whose data equals `key`, or NIL. 550
Pseudocode 7.7 `INSERT-AFTER`: insert a new node right after node `p` in $O(1)$ time. 551
Pseudocode 7.8 `INSERT-SORTED`: insert `x` so that the list stays in ascending order. 552
Pseudocode 7.9 `DELETE-KEY`: remove the first node whose data equals `key`, using a trailing pointer `prev`. 553
Pseudocode 7.10 `REVERSE-ITERATIVE`: reverse the list in one pass by turning each `next` arrow around. 554
Pseudocode 7.11 `REVERSE-RECURSIVE`: reverse the rest of the list, then hang the first node on its end. 556
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. 718
Pseudocode 9.2 `BACK-INSERTION` with only a head pointer: walk to the last node, then link in both directions. 720
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. 721
Pseudocode 9.4 `FRONT-DELETION`: remember the first node, move `head`, clear the new first node's `prev`, release the old node. 721
Pseudocode 9.5 `BACK-DELETION` with only a head pointer: walk to the last node; its predecessor is `del.prev`. 722
Pseudocode 9.6 `BACK-DELETION` with head and tail: the node before the last is `tail.prev`, so no walk is needed. 734
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. 736
Pseudocode 9.8 `INSERT-SORTED`: find the first node that is not smaller than `x`, and insert before it. 737
Pseudocode 9.9 `DELETE-NODE`: unlink node `p` in $O(1)$ time; no search and no trailing pointer are needed. 738
Pseudocode 9.10 `LIST-INSERT-AFTER` in a circular list with a sentinel: the four steps of **Pseudocode 9.7** without its test. 753
Pseudocode 9.11 `LIST-DELETE` in a circular list with a sentinel: two assignments, no tests. 754
Pseudocode 9.12 `LIST-SEARCH-SENTINEL`: store the key in the sentinel, so that the loop needs only one test per step. 755

Pseudocode 9.13 `BORROW` on the "recently borrowed" panel: a hash-map lookup plus a constant number of pointer assignments. 763
Pseudocode 10.1 Factorial with a loop and with recursion. 809
Pseudocode 10.2 Fibonacci numbers by memoized recursion. 813
Pseudocode 10.3 Euclid's algorithm, recursive and iterative. 822
Pseudocode 10.4 Exponentiation by repeated squaring. 826
Pseudocode 10.5 The recursive Towers of Hanoi. 830
Pseudocode 11.1 The three depth-first traversals; they differ only in the line on which the node itself is visited. 886
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. 887
Pseudocode 11.3 Inorder traversal with an explicit stack: go left pushing every node, pop and visit, then continue with the right subtree. 889
Pseudocode 11.4 Building an expression tree from postfix notation with a stack of trees. 894
Pseudocode 11.5 `TREE-INSERT`: walk down as in a search and hang the new node on the empty subtree where the walk ends. 898
Pseudocode 11.6 Searching a binary search tree, recursively and iteratively; both return the node holding k, or NIL. 899
Pseudocode 11.7 The minimum is the leftmost node and the maximum the rightmost node of a nonempty tree. 900
Pseudocode 11.8 Printing a tree sideways: reverse inorder, indented by depth. 902
Pseudocode 11.9 `TREE-SUCCESSOR` without parent pointers: search for k while remembering the last left turn. 917
Pseudocode 11.10 `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. 919
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. 967
Pseudocode 13.1 `BFS`: breadth-first search from the source s, computing hop distances d and parents π. 1084
Pseudocode 13.2 `DFS` and `DFS-VISIT`: depth-first search with discovery and finish times. 1095
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. 1097
Pseudocode 13.4 `CONNECTED-COMPONENTS`: label every vertex with the number of its component, one breadth-first search per component. 1107
Pseudocode 13.5 `KAHN`: topological sort by repeatedly removing a vertex of in-degree 0 1115
Pseudocode 13.6 `TOPOLOGICAL-SORT`: topological order by decreasing DFS finish time. 1116
Pseudocode 13.7 `DIJKSTRA`: single-source shortest paths with nonnegative weights, using a min-priority queue with lazy deletion of stale entries. 1125
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. 1165
Pseudocode 14.2 `SIFT-DOWN`: let the key at index i sink, exchanging it with its larger child, until no child is larger. 1166
Pseudocode 14.3 `INSERT`: append the key as the last leaf, then let it rise. 1168
Pseudocode 14.4 `EXTRACT-MAX`: save the root, move the last leaf into the root, shrink the heap, and let the new root sink. 1169
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. 1170
Pseudocode 14.6 `INCREASE-KEY` and `DECREASE-KEY` in a max-heap: change the key, then let it rise or sink. 1170
Pseudocode 14.7 `BUILD-MAX-HEAP`: sift down every internal node, from the last one back to the root. 1172
Pseudocode 14.8 `IS-MAX-HEAP`: check every child against its parent; an empty or one-element array is a heap. 1177

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. 1193
Pseudocode 14.10 `TOP-K`: keep the k best items in a min-heap whose root is the weakest of them. 1202
Pseudocode 14.11 `MERGE-K`: merge k sorted lists with a min-heap that holds one front element from each list. 1203
Pseudocode 14.12 `SIFT-UP-HOLE`: the key rises through a moving hole and is written once at the end. 1217
Pseudocode 15.1 The polynomial hash of a string by Horner's rule; M is 264 when the machine's unsigned overflow does the reduction for free. 1225
Pseudocode 15.2 Search, insertion and deletion in a hash table with separate chaining. 1237
Pseudocode 15.3 Search, insertion and deletion with open addressing and tombstones. 1253

List of Equations

(1.1) 7
(1.2) 26
(1.3) 38
(1.4) 40
(1.5) 47
(1.6) 53
(1.7) 54
(1.8) 58
(1.9) 70
(2.1) 79
(2.2) 92
(2.3) 92
(2.4) 99
(2.5) 100
(2.6) 109
(2.7) 116
(2.8) 123
(2.9) 123
(2.10) 127
(2.11) 169
(3.1) 191
(3.2) 196
(3.3) 202
(3.4) 202
(3.5) 202
(3.6) 214
(3.7) 216
(3.8) 228
(3.9) 228
(3.10) 240
(3.11) 240
(3.12) 241
(3.13) 248
(3.14) 249
(3.15) 250

(4.1) 305
(4.2) 306
(4.3) 333
(4.4) 333
(4.5) 337
(4.6) 343
(5.1) 386
(5.2) 386
(5.3) 386
(5.4) 390
(5.5) 401
(5.6) 427
(5.7) 427
(5.8) 434
(5.9) 435
(5.10)....437
(6.1) 463
(6.2) 464
(6.3) 467
(6.4) 472
(6.5) 478
(6.6) 494
(7.1) 529
(7.2) 529
(7.3) 529
(7.4) 529
(7.5) 530
(7.6) 551
(7.7) 556
(8.1) 611
(8.2) 622
(8.3) 632
(8.4) 643
(8.5) 657
(8.6) 683
(9.1) 715
(9.2) 717
(9.3) 754
(9.4) 755
(9.5) 763
(9.6) 766
(9.7) 767
(10.1)....799
(10.2)....799
(10.3)....803
(10.4)....809
(10.5)....811
(10.6)....812
(10.7)....812
(10.8)....812
(10.9)....814

(10.10) 814
(10.11) 821
(10.12) 825
(10.13) 825
(10.14) 826
(10.15) 826
(10.16) 836
(10.17) 842
(10.18) 850
(11.1) 875
(11.2) 879
(11.3) 882
(11.4) 885
(11.5) 885
(11.6) 885
(11.7) 885
(11.8) 893
(11.9) 901
(11.10) 926
(11.11) 927
(11.12) 927
(11.13) 933
(11.14) 936
(12.1) 967
(12.2) 970
(12.3) 970
(12.4) 971
(12.5) 988
(12.6) 992
(12.7) 1003
(12.8) 1035
(12.9) 1040
(13.1) 1070
(13.2) 1072
(13.3) 1072
(13.4) 1073
(13.5) 1073
(13.6) 1076
(13.7) 1087
(13.8) 1087
(13.9) 1097
(13.10) 1125
(13.11) 1127
(14.1) 1160
(14.2) 1160
(14.3) 1161
(14.4) 1161
(14.5) 1161
(14.6) 1163
(14.7) 1163
(14.8) 1167

(14.9) 1174
(14.10) 1174
(14.11) 1174
(14.12) 1174
(14.13) 1196
(14.14) 1203
(14.15) 1203
(14.16) 1208
(15.1) 1222
(15.2) 1223
(15.3) 1223
(15.4) 1224
(15.5) 1224
(15.6) 1225
(15.7) 1232
(15.8) 1232
(15.9) 1232
(15.10) 1232
(15.11) 1250
(15.12) 1251
(15.13) 1251
(15.14) 1251
(15.15) 1252
(15.16) 1252
(15.17) 1266
(15.18) 1266
(15.19) 1266
(15.20) 1266
(15.21) 1267
(15.22) 1272
(15.23) 1273
(15.24) 1277
(15.25) 1277
(15.26) 1288
(16.1) 1312
(16.2) 1314
(16.3) 1314
(16.4) 1315
(16.5) 1344
(16.6) 1360
(16.7) 1368
(16.8) 1369
(16.9) 1384
(16.10) 1384
(16.11) 1396
(16.12) 1433

Preface

Every program you have ever used keeps something: a list of songs, a queue of messages, a map of streets, a catalog of books. How it keeps those things — side by side in a row, linked one to the next, arranged as a tree, scattered into numbered buckets — decides whether the same program answers in a blink or makes you wait. That choice is what a data structure is, and it is the whole subject of this book. Keep that sentence in mind and nothing in these pages will feel arbitrary.

The book grew out of the Data Structure course (EF234201) taught in the Department of Informatics at Institut Teknologi Sepuluh Nopember (ITS), Surabaya, and it has been rebuilt so that it stands entirely on its own. Its sixteen chapters follow the rhythm of a sixteen-week semester, but you need no classroom, no lecturer and no other document to learn from it.

Who this book is for

You are assumed to know nothing. Not "nothing about data structures" — nothing. If you have never written a line of code, never compiled a program, never wondered where a variable actually lives, you are exactly the reader this book was written for. Every term is introduced in plain words before it is used, every structure is drawn state by state before it is programmed, and every program in the book was run before its output was printed here.

That principle makes the book long. A shorter book would assume you already knew the hard parts, and such a book is no use to the person who does not.

One idea, three languages

A data structure is an idea, not a feature of one language, and this book treats it that way. The language it teaches is **C++**, because C++ lets you see the memory, the addresses and the pointers that every structure is made of. Then every program is given again, complete and runnable, in **Java** and in **Python**, doing the same thing and printing the same output, line for line. Seeing one idea written three ways is the quickest route to understanding the idea itself rather than the habits of one tool, and it shows you what the libraries of Java and Python are quietly doing for you — so that whichever language your next job uses, you will already know what lies underneath it.

What is in it

Chapters 1 to 3 build the ground: how memory holds data, a C++ refresher from zero, arrays in depth and a first taste of complexity, and a full set-up of VS Code for C++, Java and Python on Windows, macOS and Linux; then searching, from sequential search and its sentinel to binary, interpolation and jump search; then sorting, from bubble, selection and insertion sort through Shell sort, merge sort and quicksort to counting and radix sort, all measured on real data. Chapters 5 to 7 add structures that grow and shrink: stacks and queues with their classic applications — matching brackets, undo and redo, infix to postfix, the function call stack, the circular queue and the deque; pointers, references and functions, with dynamic memory, `nullptr`, `const` and smart pointers, and what Java references and Python names do instead; and singly linked lists, with head and tail pointers, dummy head nodes, circular lists and a proof that no memory leaks. Chapters 9 to 11 go further along the links: doubly and circular linked lists with

sentinels, a deque and a "recently used" list; recursion, followed frame by frame on the call stack through factorial, Fibonacci, Euclid's algorithm, fast exponentiation and the Towers of Hanoi, down to the depth limits of each language; and trees and binary search trees, with every traversal, balanced and degenerate shapes and a first look at self-balancing trees. Chapters 13 to 15 finish the journey: graphs, with their representations, breadth-first and depth-first search, connected components, topological sort and a gentle Dijkstra; heaps and priority queues, from sift-up and sift-down to building a heap in linear time, heap sort and top-*k*; and hashing, with hash functions, collisions and the birthday bound, separate chaining, open addressing, load factor and rehashing measured, and a look inside `std::unordered_map`, Java's `HashMap` and Python's `dict`.

Chapters 4, 8, 12 and 16 are **Evaluation and Discussion** chapters. Each closes one quarter of the book by solving complete problems in full — every problem restated, designed on paper, programmed, tested, run, and explained — with the tempting wrong solution shown and taken apart, and a "check yourself" variant at the end of every problem. The last of them ends by choosing the right data structure for one catalog and measuring every structure of the book against it.

Throughout, one small library grows chapter by chapter: **Pustaka Ceria**, Indonesian for "Cheerful Library", a campus library that opens with eight books piled on the floor and a year later owns eighty thousand. Its catalog is stored in an array, searched and sorted; its book-return cart is a stack and its cataloging line a queue; its records are linked, arranged into search trees and hashed by title; its campus map is a graph and its hold requests a heap. Beside it runs **Profitina Laundry**, the small laundry business that is the hands-on case study of the Profitina books: its day's orders sorted heaviest first, its regular and express queues, its weekly totals, its service menu as a tree, its website as a graph and its service desk built from a hash map and a priority queue. Each chapter also has a *Profitina in Practice* section that shows where the same ideas appear in a real product.

Every program in this book was run before its output was printed. The programs are collected, chapter by chapter, in the accompanying `Code` folder, each chapter with its `cpp`, `java` and `python` versions side by side and a README that explains how to build and run each one in a terminal and in VS Code on Windows, macOS and Linux. Every listing in the book is the file in that folder, character for character.

How to read it

With an editor open and a compiler ready. Type the examples in rather than copying them; the typing is where the errors happen, and the errors are where the learning happens. Before you run a program, draw the structure on paper — the boxes, the arrows, the indices — and predict what it will print; then run it and compare. Break each example deliberately once it works, and read what the failure looks like. The exercises at the end of each chapter all have complete solutions — use them after your own attempt, not instead of it.

Numbered figures, tables, listings, pseudocode and equations are all live links: in the electronic edition, click "Listing 5.3" or "[4]" anywhere in the text and you are taken straight to it. The contents and the lists at the front of the book work the same way.

About the author

Irfan Subakti describes himself in one line — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — and every part of that line has found its way into this book. He has taught in the Department of Informatics at ITS since 1998. He supervises and examines undergraduate theses, teaches MLOps for the IBM SkillsBuild program, and builds products: he founded Profitina, an AI-visibility platform whose engineering appears in the "Profitina in Practice" section of every chapter. Away from the keyboard he tends the family fruit and durian orchard, where he learned that good software, like a good orchard, is grown patiently, one well-tended piece at a time. The storyteller in him is the reason this book prefers a story to a list whenever a story will do.

Acknowledgments

To the students who worked through the earlier forms of this material and asked the questions that showed where an explanation was still thin, and to the colleagues at ITS with whom the course has been taught and refined over the years. The book owes its standard, too, to the classic textbooks of algorithms and data structures and to the official documentation of C++, Java and Python, which are cited wherever they are used.

Surabaya, September 2026



Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Chapter 1

Foundations: C++ Refresher, Memory, and Arrays

By the end of this chapter you will be able to say what a data structure is and why the choice of one can make the same program a thousand times faster or slower; describe how a computer's memory holds data as numbered bytes, how a type gives those bytes a meaning, and why some variables live on the *stack* while others live on the *heap*; read and write the C++ that the rest of this book uses, including types, variables, input and output, decisions, loops, functions, structs and the address-of operator `&`; declare, fill, walk and bound-check one- and two-dimensional arrays, compute the exact address of any element with the row-major formula, and choose between a raw array, `std::array` and `std::vector`, between a Java array and an `ArrayList`, and between a Python `list` and the `array` module; explain, with a count of steps, why reading an array element costs $O(1)$ while inserting one costs $O(n)$; and install, on Windows, macOS or Linux, a complete workbench of Visual Studio Code with C++, Java and Python, and prove that it works. Every program in the chapter is given in all three languages, and every output printed here is the output the program really produced.

1.1 Where We Begin: A Library Without Shelves

1.1.1 Pustaka Ceria

Imagine a small, cheerful library on a university campus. Its name is **Pustaka Ceria**, Indonesian for "Cheerful Library", and on its first morning it owns eight books, a desk, a librarian and nothing else. There are no shelves yet. The books arrived in a cardboard box, and the librarian, who is new, simply stacks them on the floor in the order in which she happened to take them out.

At 8 a.m. a student asks for *Filosofi Kopi*. The librarian looks at the pile, lifts the first book, reads the cover, puts it aside, lifts the second, and finds the book on the third try. Easy. With eight books, any method works.

A year later Pustaka Ceria owns eighty thousand books, and the pile method has become a disaster. A student asking for one title may wait while the librarian lifts and reads forty thousand covers on average. Nobody has become slower at reading covers; the library has simply grown, and the *method of keeping the books* has not grown with it. So the library buys shelves, numbers them, sorts the books by a code, and keeps a card catalog. Now the librarian can walk straight to the right shelf, glance along it, and return in a minute.

Nothing about the books changed. The same eighty thousand titles, the same writers, the same pages. What changed is the **organization**: where each book is kept, what is written down about where it is, and what the librarian may assume when she goes to look. That organization is exactly what this book calls a *data structure*, and Pustaka Ceria will stay with us through all sixteen chapters. In this chapter it gets its first real shelf, an array. In later chapters it gets a stack of returned books waiting to be reshelved, a queue of reservations, a tree of categories, a graph of books that are borrowed together, and a hash table that finds any book by its code in a single step.

1.1.2 What a data structure is

A program is a set of instructions that works on data: numbers, names, prices, dates, pictures. As long as a program handles one or two values, it can give each one its own variable and be done. The moment it handles *many* values of the same kind, such as the books of a library, the students of a class or the orders of a shop, it needs a way to keep them together, to find one of them, to add one, to remove one, and to visit all of them in some order. Deciding *how* to keep them is designing a data structure.

> **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 can be carried out correctly and efficiently. The five operations that nearly every data structure is judged by are **insert** (add an item), **delete** (remove an item), **search** (find an item, or learn that it is absent), **traverse** (visit every item once) and **sort** (put the items in order) [1].

Two ideas hide inside this definition, and it pays to separate them at once.

The first is the *interface*: the list of operations, what each one promises, and nothing about how it keeps that promise. "I can give you the book with code B1005, or tell you it is not here" is an interface. Computer scientists call a data type described only by its interface an **abstract data type**, or ADT.

The second is the *implementation*: the concrete arrangement of bytes in memory and the concrete steps that carry out each operation. "The books sit in a row in code order, and I find one by opening the row in the middle and deciding which half to keep" is an implementation of the interface above. So is "the books sit in eight bins chosen by the last digit of the code, and I look only in the right bin". Both keep the same promise; they keep it at very different costs.

A useful plain-language test: the interface is what the library's *visitor* needs to know; the implementation is what the *librarian* needs to know. A good data structure lets the visitor forget about the implementation entirely, and lets the librarian change it without the visitor noticing.

1.1.3 Why the choice matters

"But computers are fast," a reader may object. "Why worry about the method, when a machine can read a million covers in the blink of an eye?" The answer is that the number of steps a method takes does not grow at the same rate for every method, and when the collection is large the difference in *rate* swamps any difference in the speed of the machine.

Suppose Pustaka Ceria holds n books. Three methods of finding one book are compared in **Table 1.1**. The first reads every cover, in the worst case all n of them. The second keeps the books sorted and repeatedly halves the part still worth searching, so it needs about $\log_2 n$ looks: the number of times n can be halved before one book is left. The third is a hash table, which computes the position of the book from its code and, on average, looks once. Chapter 2 builds the first two, and Chapter 15 builds the third.

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

Read the last row slowly. A billion books, and the second method needs thirty looks where the first needs a billion. If every look took one microsecond, the first method would take about seventeen minutes and the second thirty microseconds. No faster computer closes a gap like that: making the machine a hundred times faster turns seventeen minutes into ten seconds, which is still three hundred thousand times slower than the better method on the old machine. *Choosing the right data structure is usually worth more than buying the fastest computer.* That sentence is the reason this book exists.

1.1.4 Data structures behind every web page

It is easy to think of data structures as something that lives only in textbooks. In fact you use hundreds of them every time you open a web page, and it is worth pausing to see where, because the example makes the abstract word concrete.

First, two terms that are often confused. The **Internet** is a very large network of networks: millions of smaller computer networks, owned by universities, companies, telecommunication providers and households, connected to each other and speaking a common family of rules for moving data called **TCP/IP** [2]. The **World Wide Web** is only one of the services that run on top of the Internet. It is the service in which a program called a *web browser* asks a program called a *web server* for documents, using a rule book called **HTTP**, the HyperText Transfer Protocol, and shows the documents to a person. E-mail, file transfer, chat, video calls, online games, streaming music and the apps on a phone are other services that use the same Internet without being the web. So the answer to the old question “are the Internet and the web the same thing?” is no: the web is one passenger on the Internet’s roads.

Now follow one request. You type the address of a page into the browser.

1. The browser looks the site’s name up in its **cache**, a table of recently seen names and their numeric addresses. That table is a *hash table*, the structure of Chapter 15, because it must answer “have I seen this name?” in one step.
2. If the name is new, the question travels to a chain of name servers, whose zones of responsibility form a *tree*, the structure of Chapter 11.
3. The request crosses the Internet through routers, each of which keeps the requests waiting for the same outgoing line in a *queue*, the structure of Chapter 5, so that the first to arrive is the first to leave. The routers choose their paths by computing over a *graph* of links, the structure of Chapter 13.
4. The server receives the request and reads the page’s data from files whose blocks are listed in *arrays* and indexed by *trees*.
5. The browser receives the page and builds from its text a *tree* of elements, walking it with a *stack* to lay out and draw each element on the screen.

Five steps and at least six data structures, each chosen because it makes the operation that step needs cheap. Every one of them is in this book.

1.1.5 Six families, and the road map of this book

For all their variety, the data structures in daily use belong to a handful of families. **Figure 1.1** shows the six that this book studies, each drawn in the form in which later chapters will draw it.

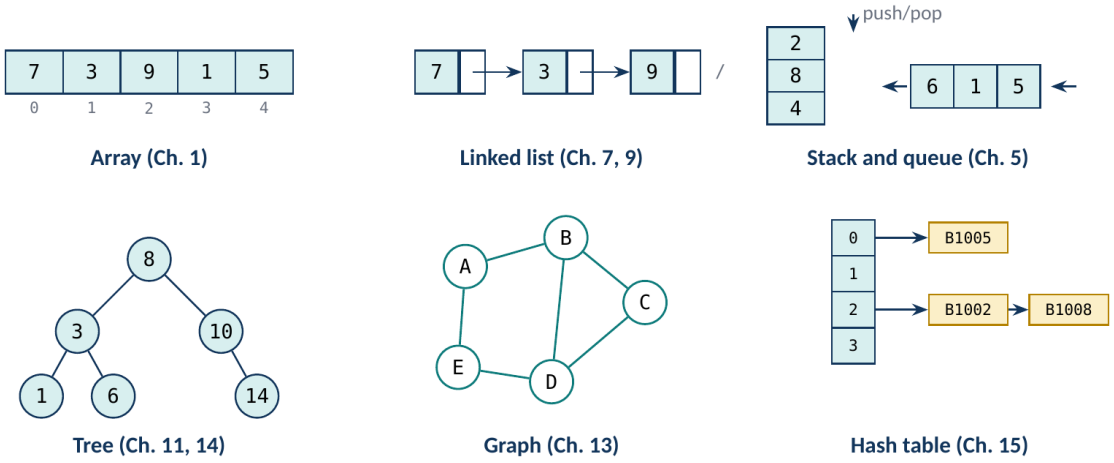


Figure 1.1 The six families of data structures in this book: arrays, linked lists, stacks and queues, trees (with heaps), graphs and hash tables, each labeled with the chapters that build it.

- An **array** keeps its items side by side in one block of memory, each reachable by its position number. It is the subject of this chapter and the foundation of almost everything else.
- A **linked list** keeps each item in its own little box that also records where the next box is, so that items can be added and removed without moving the others.
- A **stack** lets you add and remove only at one end, like a pile of plates; a **queue** adds at one end and removes at the other, like a line at a counter.
- A **tree** arranges items in levels below a single root, like a family tree or the folders on a disk; a **heap** is a tree kept in a special order so that the largest or smallest item is always on top.
- A **graph** records items and the connections between them, such as towns and roads, or web pages and links.
- A **hash table** computes where an item should be from the item's own key, and so can often find it in a single step.

Table 1.2 lays out the sixteen chapters. Every fourth chapter, marked *Evaluation and Discussion*, stops to work through a set of problems on the three chapters before it, in full detail.

Table 1.2 The sixteen chapters of this book and what each one adds.

Ch.	Title	What it adds
1	Foundations: C++ Refresher, Memory, and Arrays	Memory, C++, arrays, a first taste of complexity, the workbench
2	Searching	Sequential, sentinel, binary and interpolation search; *O*-notation in full
3	Sorting	The classic sorts, from bubble sort to quicksort, merge sort and heapsort
4	Evaluation and Discussion I	Worked problems on Chapters 1 to 3
5	Stacks and Queues	Last-in first-out and first-in first-out, on arrays
6	Pointers, References, and Functions	The pointer story begun in this chapter, completed
7	Singly Linked Lists	Nodes and links; insertion without shifting
8	Evaluation and Discussion II	Worked problems on Chapters 5 to 7
9	Doubly and Circular Linked Lists	Two-way links and rings
10	Recursion	Functions that call themselves
11	Trees and Binary Search Trees	Hierarchies, traversals, ordered search
12	Evaluation and Discussion III	Worked problems on Chapters 9 to 11
13	Graphs	Adjacency matrices and lists, breadth- and depth-first search
14	Heaps and Priority Queues	The heap property and its operations
15	Hashing and Hash Tables	Hash functions, collisions, chaining and open addressing
16	Evaluation and Discussion IV	Worked problems on Chapters 13 to 15 and the whole book

1.2 Three Languages, One Idea

1.2.1 Why C++ first, and why also Java and Python

This book teaches data structures in **C++**, and gives every program again in **Java** and in **Python**, complete and runnable, printing the same results wherever the languages allow it. There are good reasons for each of the three.

C++ is the language of the main text because it hides nothing. When a C++ program declares an array of eight books, the eight books really do sit side by side in memory, and the program can ask where each one begins. When a C++ program asks for memory while it runs, it must also give that memory back. A data structures course is, at bottom, a course about how data sits in memory, and C++ lets us look at the memory directly. The same property makes C++ the language of operating systems, browsers, databases, game engines and the fast inner parts of many other systems.

Java and Python are the languages in which much of the world's other software is written, and each makes a different choice about memory. Java puts every object on the heap and frees it automatically, but still distinguishes plain numbers from objects. Python makes *everything* an object and every variable a name that refers to one. Seeing the same data structure in all three languages shows which parts of it are the idea itself and which are the habits of one language.

All three are among the most used languages in the world. **Table 1.3** shows the top of the TIOBE Programming Community Index for September 2026, which estimates a language's popularity from the number of engineers, courses and vendors that search engines find for it [3], [4].

Table 1.3 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%

An index of popularity is not a ranking of quality, and TIOBE itself stresses that it measures neither the "best" language nor the one in which most code is written [3]. It does tell us that the three languages of this book, together with C, the language from which C++ grew, are at the top of the list, and have been there for many years.

1.2.2 How a program becomes something that runs

A computer's processor understands only its own *machine code*: long sequences of numbers, each an instruction such as "add these two registers" or "load four bytes from this address". Nobody writes machine code by hand any more. We write *source code* in a language people can read, and a program translates it. The three languages of this book translate in three different ways, drawn side by side in **Figure 1.2**.

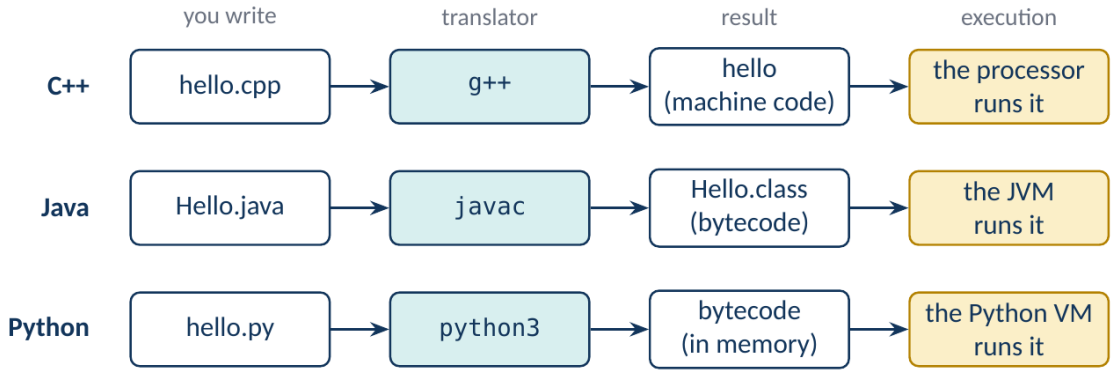


Figure 1.2 From source file to running program: C++ is compiled to machine code, Java to bytecode for the Java Virtual Machine, and Python to bytecode that its own virtual machine runs at once.

- **C++** is *compiled*. The compiler, `g++` or `clang++`, reads the whole of `hello.cpp`, checks it, and writes a new file, the *executable*, full of machine code. You run the executable; the source is no longer needed. Compiling takes a moment, but the result runs at the full speed of the processor.
- **Java** is compiled too, but not to machine code. The compiler `javac` turns `Hello.java` into `Hello.class`, which holds *bytecode*, the machine code of an imaginary processor called the **Java Virtual Machine** (JVM). The `java` command starts a JVM, which runs the bytecode and, for the parts that run often, compiles it further into real machine code while the program is running.
- **Python** is usually described as *interpreted*. The `python3` command reads `hello.py`, translates it to bytecode in memory, and immediately runs that bytecode on the Python virtual machine.

There is no separate step and no separate file to run, which makes Python very convenient for trying things out, and usually slower than C++ or Java for heavy computation.

1.2.3 How the code is presented

Every program in this chapter lives in the folder `Chapter01` of the book's accompanying code, in three subfolders, `cpp`, `java` and `python`, with a `README.md` that says how to build and run each program on Windows, macOS and Linux, in a terminal and in VS Code. Section 1.7 installs everything you need. If you have never installed a compiler, you may want to read Section 1.7 first and come back; everything between here and there is written so that it can be read, understood and checked against the printed output even before you run it.

The C++ programs were compiled with `g++ -std=c++17 -O2 -Wall -Wextra` and produce no warnings; the Java programs with `javac` from JDK 21, and they also run on the current JDK 25; the Python programs with Python 3.11, and they also run on the current Python 3.14. Where the three languages print exactly the same thing, the output is printed once. Where they differ, because the languages themselves differ, each output is printed, and the difference is explained.

1.3 How Memory Holds Data

Before we can talk sensibly about arrays, or about any data structure, we need a clear picture of the thing they are built in: the computer's memory. The picture is simple, and once you have it, much of this book becomes common sense.

1.3.1 Bits and bytes

At the very bottom, a computer stores everything as **bits**. A bit is a tiny switch that is either off or on, written 0 or 1. One bit alone can tell only two things apart: yes or no, lamp off or lamp on. Two bits together can be in four different states, 00, 01, 10 and 11; three bits in eight. Each extra bit doubles the number of states, because every old state can now be followed by a 0 or by a 1. So a group of k bits can be in

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

and Equation (1.1) is the reason the number 2 and its powers turn up everywhere in computing.

Bits are grouped in eights. A group of eight bits is a **byte**, and by Equation (1.1) a byte has $2^8 = 256$ states. We can use those 256 states to stand for the whole numbers 0 to 255: the bits are read as a number in base 2, just as the digits of 407 are read in base 10 as $4 \cdot 100 + 0 \cdot 10 + 7 \cdot 1$. For example the byte 00000111 is $4 + 2 + 1 = 7$, and 11101010 is $128 + 64 + 32 + 8 + 2 = 234$.

Writing eight binary digits is tiring, so programmers write bytes in **hexadecimal**, base 16. Its sixteen digits are 0 to 9 and then `a` to `f` for ten to fifteen. Four bits make exactly one hexadecimal digit, so a byte is always exactly two: 00000111 is `07`, and 11101010, which splits into $1110 = 14$ and $1010 = 10$, is `ea`. In C++, Java and Python a hexadecimal number is written with the prefix `0x`, as in `0xEA`.

1.3.2 Addresses

The memory of a running computer is one very long row of bytes, many billions of them. Each byte has a number, its **address**, counting up from zero, exactly like the houses along a street numbered from one end. When a program says "read the byte at address 1004", the memory

hardware goes straight to house number 1004; it does not need to walk past the houses before it. This single fact, that any address can be reached in the same short time, is called **random access**, and it is what makes the array, the structure of this chapter, so fast.

A value that needs more than one byte, such as a whole number that can be larger than 255, is kept in several *consecutive* bytes, and the value's address is the address of its first byte. In the pictures of this book we use small, friendly addresses such as 1000; real addresses are much larger numbers, usually printed in hexadecimal, as the programs below will show.

1.3.3 A type gives the bytes a meaning

Bytes by themselves mean nothing. The four bytes `ea 07 00 00` could be a number, four letters, part of a picture or part of a machine instruction. What decides is the **type** that the program says they have. The type fixes two things: *how many* bytes the value occupies, and *how* those bytes are to be read.

Figure 1.3 shows twelve consecutive bytes that hold four variables of different types.

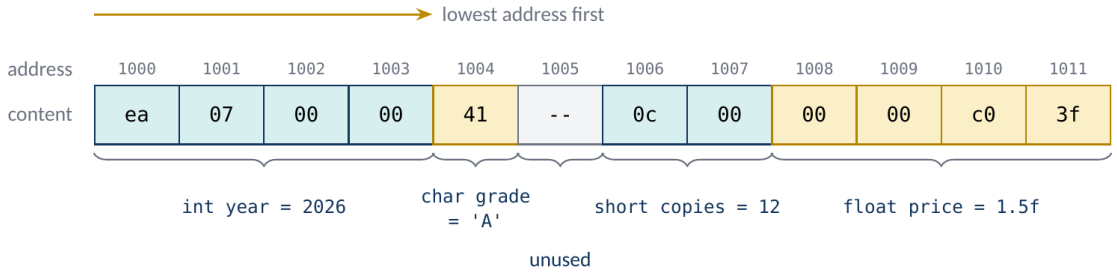


Figure 1.3 Twelve bytes of memory holding an `int`, a `char`, a `short` and a `float`; each type decides how many bytes a value uses and how they are read, and the lowest-addressed byte comes first.

- `int year = 2026` takes four bytes, at addresses 1000 to 1003. Written in hexadecimal, 2026 is `0x000007EA`, that is, the four bytes `00 00 07 ea` from the most significant to the least. Most computers today, including every Intel, AMD and Apple processor, store the **least significant byte first**, at the lowest address; this is called **little-endian** order, and it is why the figure shows `ea 07 00 00`. The opposite order, most significant byte first, is **big-endian**; it is used in network protocols and in Java's own file formats [5].
- `char grade = 'A'` takes one byte. The letter A is stored as its character code, 65, which is `0x41`.
- The byte at 1005 is unused. A `short` must start at an even address on this machine, so the compiler leaves one byte out. This gap is called **padding**, and we meet it again in Section 1.5.3.
- `short copies = 12` takes two bytes, `0c 00`.
- `float price = 1.5f` takes four bytes, but a `float` is read in a completely different way from an `int`: part of the bits give the digits and part give the position of the decimal point. The same four bytes read as an `int` would mean 1,069,547,520.

That last point is worth saying again, in plain words: *the memory does not know what it holds; only the program does, through the types it uses.*

1.3.4 Seeing it for yourself

Listing 1.1 asks the machine how many bytes each basic C++ type takes, looks at the four bytes of the `int` 2026 one by one, and prints the addresses of two variables.

Listing 1.1 `memory_basics.cpp`: the sizes of the basic types, the bytes of an `int`, and an address on the stack and on the heap.

```cpp
// memory_basics.cpp - sizes, bytes, and where variables live
#include <iomanip>
#include <iostream>
using namespace std;

int main() {
    cout << "Sizes on this machine, in bytes:\n";
    cout << "  char      " << sizeof(char) << '\n';
    cout << "  bool      " << sizeof(bool) << '\n';
    cout << "  short     " << sizeof(short) << '\n';
    cout << "  int       " << sizeof(int) << '\n';
    cout << "  long long " << sizeof(long long) << '\n';
    cout << "  float     " << sizeof(float) << '\n';
    cout << "  double    " << sizeof(double) << '\n';

    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";

    int onStack = 7;                        // a local variable: on the stack
    int* onHeap = new int(8);               // asked for at run time: on the heap
    cout << "\nThe local variable lives at " << &onStack << " (stack)\n";
    cout << "The new int lives at        " << onHeap << " (heap)\n";
    delete onHeap;                          // give the heap memory back
    return 0;
}
```

```
Sizes on this machine, in bytes:
  char      1
  bool      1
  short     2
  int       4
  long long 8
  float     4
  double    8

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

The local variable lives at 0x7ffd287f05e4 (stack)
The new int lives at        0x55a6fbee52c0 (heap)
```

The program uses four ideas that the refresher of Section 1.4 explains in full, so a short tour is enough here. The operator `sizeof` gives the number of bytes a type or a variable takes. The expression `&year` is the *address* of `year`, and `reinterpret_cast<const unsigned char*>` tells the compiler to let us look at the bytes found there one at a time. The manipulators `hex`, `setw(2)` and `setfill('0')` print each byte as two hexadecimal digits. The last paragraph puts one `int` on the stack and one on the heap, which Section 1.3.5 explains, and prints both addresses. Your addresses will be different, and they change from run to run, because modern operating systems deliberately place a program's memory at a random position each time it starts, which makes some kinds of attack much harder.

The sizes printed are those of a 64-bit Linux, Windows or macOS machine. The C++ standard fixes only that `char` is one byte and that each type in the list `short`, `int`, `long long` is at least as large as the one before; the exact sizes are chosen by the compiler for each machine. That is why the book always asks `sizeof` rather than assuming.

Java fixes the sizes of its basic types in the language itself, the same on every machine, and does not let a program see addresses at all. **Listing 1.2** prints what Java allows us to see.

Listing 1.2 `MemoryBasics.java`: Java's fixed type sizes and the bytes of an `int` in the machine's own order.

```
// MemoryBasics.java - sizes, bytes, and where values live
import java.nio.ByteBuffer;
import java.nio.ByteOrder;

public class MemoryBasics {
    public static void main(String[] args) {
        System.out.println("Sizes fixed by the Java language, in bytes:");
        System.out.println("  byte      " + Byte.BYTES);
        System.out.println("  char      " + Character.BYTES);
        System.out.println("  short     " + Short.BYTES);
        System.out.println("  int       " + Integer.BYTES);
        System.out.println("  long      " + Long.BYTES);
        System.out.println("  float     " + Float.BYTES);
        System.out.println("  double    " + Double.BYTES);

        int year = 2026;                      // 2026 = 0x000007EA
        byte[] b = ByteBuffer.allocate(Integer.BYTES)
                .order(ByteOrder.nativeOrder()).putInt(year).array();
        StringBuilder sb = new StringBuilder();
        for (byte x : b) {
            sb.append(String.format(" %02x", x & 0xFF));
        }
        System.out.println("\nThe int " + year + " in this machine's byte order:"
                + sb);
        boolean little = ByteOrder.nativeOrder() == ByteOrder.LITTLE_ENDIAN;
        System.out.println("So this machine is " + (little ? "little" : "big")
                + "-endian.");

        int onStack = 7;                      // a local: in the stack frame
        int[] onHeap = new int[] {8};         // an array is a heap object
        System.out.println("\nThe local variable holds " + onStack + " (stack)");
        System.out.println("The array object holds   " + onHeap[0] + " (heap)");
        System.out.println("Java never shows addresses;"
                + " the garbage collector frees the heap.");
    }
}
```

```
Sizes fixed by the Java language, in bytes:
  byte      1
  char      2
  short     2
  int       4
  long      8
  float     4
  double    8

The int 2026 in this machine's byte order: ea 07 00 00
So this machine is little-endian.

The local variable holds 7 (stack)
The array object holds   8 (heap)
Java never shows addresses; the garbage collector frees the heap.
```

Notice the one difference in the table: a Java `char` takes *two* bytes, because Java stores characters in the UTF-16 encoding, which has room for the letters of most of the world's scripts. Notice also what is missing: no address is printed, because Java deliberately hides them.

Python goes further still. Its `struct` module can tell us the sizes of the C types that the Python interpreter is built from, which are the same as those of C++ on this machine, and `sys.getsizeof` tells us how many bytes a Python object really takes. **Listing 1.3** does both.

Listing 1.3 `memory_basics.py`: C type sizes seen through `struct`, the bytes of 2026, and the true size of Python objects.

```
# memory_basics.py - sizes, bytes, and where values live
import struct
import sys

print("Sizes of the C types Python's struct module uses, in bytes:")
for label, code in [("char", "c"), ("bool", "?"), ("short", "h"), ("int", "i"),
                    ("long long", "q"), ("float", "f"), ("double", "d")]:
    print(f"  {label:<9} {struct.calcsize(code)}")

year = 2026                                # 2026 = 0x000007EA
raw = year.to_bytes(4, sys.byteorder)
print(f"\nThe int {year} as 4 bytes in this machine's order: {raw.hex(' ')}")
print(f"So this machine is {sys.byteorder}-endian.")

on_stack = 7
on_heap = [8]
print(f"\nsys.getsizeof(7)   = {sys.getsizeof(on_stack)} bytes:"
      " even 7 is a whole object")
print(f"sys.getsizeof([8]) = {sys.getsizeof(on_heap)} bytes:"
      " a list object on the heap")
print("Every Python value is an object on the heap; names only refer to it.")
```

```
Sizes of the C types Python's struct module uses, in bytes:
  char      1
  bool      1
  short     2
  int       4
  long long 8
  float     4
  double    8

The int 2026 as 4 bytes in this machine's order: ea 07 00 00
So this machine is little-endian.

sys.getsizeof(7)   = 28 bytes: even 7 is a whole object
sys.getsizeof([8]) = 64 bytes: a list object on the heap
Every Python value is an object on the heap; names only refer to it.
```

The number 7 takes *28 bytes* in Python, seven times the four bytes of a C++ `int`. The reason is that a Python integer is a complete object on the heap: besides the value it records its type, a count of how many names refer to it, and its length, because Python integers may grow to any size. That price buys convenience, and we will see in Section 1.5.10 what it costs when there are millions of numbers.

1.3.5 The stack and the heap

When a program runs, the operating system gives it a large range of addresses and divides it into regions with different jobs. **Figure 1.4** shows the regions of a typical C++ program.

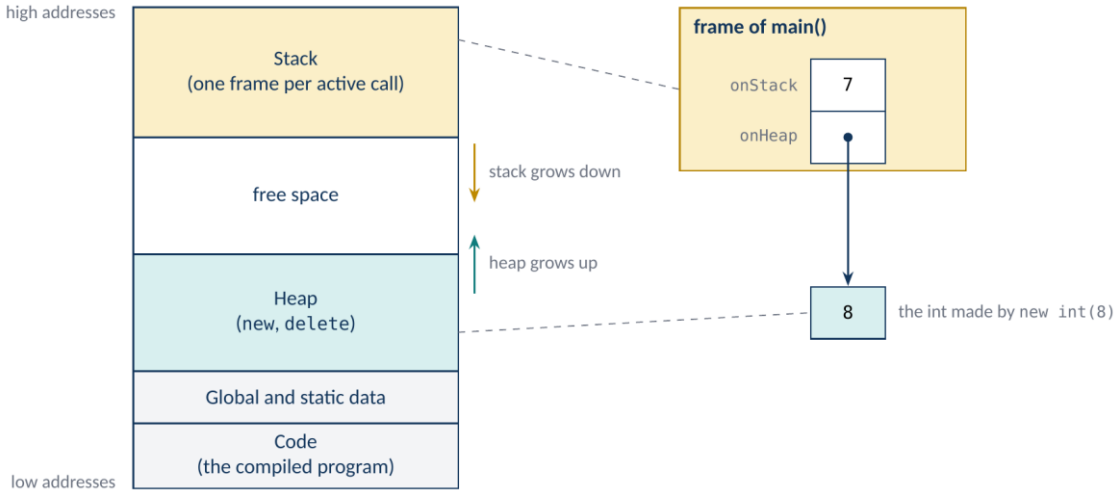


Figure 1.4 The memory of a running C++ program: code and global data at the bottom, the heap growing upward, the stack growing downward, and, on the right, the frame of `main` whose pointer `onHeap` refers to an `int` on the heap.

- The **code** region holds the program's own machine instructions.
- The **global and static data** region holds variables declared outside every function. They exist for the whole run.
- The **stack** holds the variables of the functions that are running right now. Each time a function is called, a block of memory called a **stack frame** is set aside at the top of the stack for its parameters and local variables; when the function returns, its frame is released at once, simply by moving the top of the stack back. Because frames are added and removed only at the top, in last-in first-out order, the region behaves exactly like the stack data structure of Chapter 5, which is where its name comes from. The stack is very fast and needs no bookkeeping by the programmer, but it is small, typically one to eight megabytes, and a variable on it lives only as long as the function that made it.
- The **heap** holds memory that the program asks for explicitly while it runs, with `new` in C++, and must give back explicitly with `delete`. A heap variable is not tied to any function: it lives until the program deletes it. The heap is large, often gigabytes, and slower to use than the stack because the system must find a free piece of the right size each time. (This region shares its name with the heap data structure of Chapter 14, but the two have nothing else in common.)

In **Listing 1.1**, `onStack` is a local variable of `main`, so it is in `main`'s frame on the stack. The expression `new int(8)` asks the heap for room for one `int`, stores 8 in it, and gives back its address, which the program keeps in `onHeap`. The printed addresses show the two regions clearly: the stack address begins `0x7ff...`, near the top of the program's address range, and the heap address begins `0x55...`, much lower. Finally `delete onHeap;` gives the heap memory back. Forgetting that line would not break this little program, but in a program that runs for days, every forgotten `delete` is a small, permanent loss of memory called a **memory leak**.

Java and Python simplify this picture for the programmer. In Java, local variables of the basic types (`int`, `double` and the like) and *references* live in the stack frame, while every object and every array lives on the heap. In Python, every value is an object on the heap, and a variable is only a name that refers to one. In both languages a part of the runtime called the **garbage collector** finds the heap objects that nothing refers to any more and frees them automatically, so there is no `delete`. This is a great convenience; the price is that the programmer no longer decides exactly when memory is released.

This chapter keeps almost everything on the stack, to concentrate on arrays. Chapter 6 returns to the heap, to `new` and `delete`, and to pointers, in full.

1.4 A C++ Refresher for Beginners

This section gathers, in one place, all the C++ that the rest of the book assumes. If you already program in C++, skim it for the vocabulary and the conventions. If C++ is new to you, or you last used it long ago, read it slowly, type the programs in, and change them to see what happens: nothing in a data structures course is harder than it needs to be once the language itself stops being in the way. Stroustrup's *A Tour of C++* [6] is a good companion for everything in this section.

1.4.1 The shape of a C++ program

Every C++ program in this book has the same skeleton:

```
#include <iostream>          // bring in the library we need
using namespace std;         // let us write cout instead of std::cout

int main() {                 // the program starts running here
    cout << "Hello\n";       // statements, each ending with a semicolon
    return 0;                // 0 tells the operating system "all went well"
}
```

- Lines beginning with `#include` bring in parts of the **standard library**: `<iostream>` for input and output, `<string>` for text, `<vector>` for growable arrays, and so on.
- Every name in the standard library lives in a *namespace* called `std`, so its full names are `std::cout`, `std::string`, `std::vector`. The line `using namespace std;` lets us drop the prefix. It is convenient in short programs such as ours; in large programs, where two libraries might use the same name, professional code usually writes `std::` in full.
- `int main()` is the function where every C++ program begins. The `int` before its name says that it returns a whole number to the operating system; by convention `0` means success.
- Braces `{ }` group statements into a **block**. Each simple statement ends with a semicolon.
- `//` starts a comment that runs to the end of the line; `/* ... */` encloses a comment of any length. The compiler ignores comments; people do not.

1.4.2 Types and variables

A **variable** is a named piece of memory that holds a value of one type. Declaring it reserves the bytes and gives them the name:

```
int age = 20;                // four bytes, named age, holding 20
double price = 24500.0;      // eight bytes holding a real number
char grade = 'A';            // one byte holding a character code
bool available = true;       // one byte holding true or false
string title = "Laskar Pelangi";   // a text object from <string>
const int LOAN_DAYS = 7;     // a constant: the compiler refuses to change it
auto count = 8;              // auto: the compiler deduces the type, here int
```

Table 1.4 lists the basic types this book uses, with the sizes **Listing 1.1** printed and the range of values each can hold.

Table 1.4 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 $2^{64}-1$

Three habits prevent most beginners' mistakes with variables.

Always give a variable a value before reading it. A local variable declared as `int x;` holds whatever bytes happened to be in that piece of the stack. Reading it is a bug that may appear to work for months. Write `int x = 0;`, or give it its first value immediately.

Choose a type that is large enough. An `int` cannot count past about 2.1 billion. Pustaka Ceria's total fines in rupiah over several years could exceed that, so money in this book is kept in `long` or `long long`. When a value outgrows its type, it silently *overflows* and wraps around to a wrong number.

Mind integer division. When both sides of `/` are integers, C++ divides and throws the remainder away: `7 / 2` is `3`, not `3.5`. To get the real quotient, make one side a real number first, as `static_cast<double>(7) / 2`, which is `3.5`. The operator `%` gives the remainder: `7 % 2` is `1`.

1.4.3 Input and output

C++ writes to the screen through the object `cout` ("character output") with the operator `<<`, and reads from the keyboard through `cin` with `>>`. The arrows point the way the data flows: from the value *into* `cout`, and from `cin` *into* the variable. Several values can be chained: `cout << "Total: " << sum << '\n';`. The character `'\n'` starts a new line.

Listing 1.4 reads how many days of loan counts follow, then that many whole numbers, and prints their total, average, smallest and largest.

Listing 1.4 `io_sum.cpp`: reading a count and that many numbers with `cin`, and summarizing them.

```cpp
// io_sum.cpp - read a count and that many numbers, then summarize them
#include <iomanip>
#include <iostream>
using namespace std;

int main() {
    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;
    cout << "Days read : " << n << '\n';
    cout << "Total     : " << sum << '\n';
    cout << "Average   : " << fixed << setprecision(2) << average << '\n';
    cout << "Smallest  : " << smallest << '\n';
    cout << "Largest   : " << largest << '\n';
    return 0;
}
```

The file `week_loans.txt` in the chapter's folder holds the loan counts of one week at Pustaka Ceria:

```
7
12 9 15 4 20 11 6
```

Instead of typing the numbers, we can let the program read the file as if it were the keyboard, by **redirecting** standard input with `<` when we run it: `./io_sum < week_loans.txt` on macOS and Linux, `io_sum.exe < week_loans.txt` in the Windows Command Prompt, or `Get-Content week_loans.txt | .\io_sum.exe` in PowerShell. The program prints:

```
Days read : 7
Total     : 77
Average   : 11.00
Smallest  : 4
Largest   : 20
```

A few details deserve a word. The test `if (!(cin >> n) || n <= 0)` reads the count *and* checks that the reading succeeded: `cin >> n` fails, and the test becomes true, if the input does not start with a number. The variables `smallest` and `largest` are set from the first number, `i == 0`, because before that there is nothing to compare with. The average is computed as `static_cast<double>(sum) / n` to avoid integer division, and `fixed << setprecision(2)` prints it with exactly two decimals.

Java reads input through a `Scanner` wrapped around `System.in`, and Python through `input()` or, for a whole file of tokens, `sys.stdin.read().split()`. **Listing 1.5** and **Listing 1.6** are the two versions; both read the same file and print exactly the same five lines.

Listing 1.5 `IoSum.java`: the same summary with `Scanner`.

```java
// IoSum.java - read a count and that many numbers, then summarize them
import java.util.Scanner;

public class IoSum {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        if (!in.hasNextInt()) {
            System.out.println("Expected a positive count first.");
            return;
        }
        int n = in.nextInt();
        if (n <= 0) {
            System.out.println("Expected a positive count first.");
            return;
        }
        long sum = 0;
        int smallest = 0, largest = 0;
        for (int i = 0; i < n; i++) {
            int x = in.nextInt();
            sum += x;
            if (i == 0 || x < smallest) smallest = x;
            if (i == 0 || x > largest) largest = x;
        }
        double average = (double) sum / n;
        System.out.println("Days read : " + n);
        System.out.println("Total     : " + sum);
        System.out.printf("Average   : %.2f%n", average);
        System.out.println("Smallest  : " + smallest);
        System.out.println("Largest   : " + largest);
    }
}
```

Listing 1.6 `io_sum.py`: the same summary, reading all tokens from standard input at once.

```python
# io_sum.py - read a count and that many numbers, then summarize them
import sys

tokens = sys.stdin.read().split()
if not tokens or not tokens[0].lstrip("-").isdigit() or int(tokens[0]) <= 0:
    print("Expected a positive count first.")
    sys.exit(1)
n = int(tokens[0])
values = [int(t) for t in tokens[1:1 + n]]

total = sum(values)
average = total / n
print(f"Days read : {n}")
print(f"Total     : {total}")
print(f"Average   : {average:.2f}")
print(f"Smallest  : {min(values)}")
print(f"Largest   : {max(values)}")
```

The Python version shows a different style that Python encourages: rather than updating a running total inside a loop, it builds the list `values` in one line and hands it to the built-in functions `sum`, `min` and `max`. The work done is the same.

1.4.4 Making decisions and repeating

Programs choose and repeat. C++ has three statements for choosing and three for repeating, and Java has exactly the same ones with the same spelling.

`if` and `else` run one block or another depending on a condition:

```
if (daysLate <= 0) {
    fee = 0;
} else if (daysLate <= 7) {
    fee = daysLate * 1000;
} else {
    fee = 7000 + (daysLate - 7) * 2000;
}
```

Conditions are built from comparisons, `==` (equal; note the *two* equals signs), `!=`, `<`, `<=`, `>` and `>=`, combined with `&&` (and), `||` (or) and `!` (not).

`switch` chooses among many cases by the value of one integer or character. Each `case` should end with `break` or `return`; otherwise execution "falls through" into the next case, which is occasionally useful and usually a bug. The `default` case catches every value not listed.

`while` repeats a block as long as a condition holds, testing before each round; `do ... while` tests after each round, so its block runs at least once; and `for` packs the three parts of a counting loop into one line:

```
for (int i = 0; i < n; i++) {    // start; keep going while; step
    // runs for i = 0, 1, ..., n-1
}
```

Read `for (int i = 0; i < n; i++)` as "start `i` at zero; while `i` is less than `n`, run the block and then add one to `i`". It runs exactly n times, with `i` taking the values 0 to $n - 1$: precisely the valid positions of an array of n elements, which is why this line appears in almost every program in this book. The **range-based for**, `for (const Loan& loan : loans)`, visits every element of an array or container without an index at all.

Inside any loop, `break` leaves the loop at once, and `continue` skips the rest of the current round and goes on with the next.

1.4.5 Functions

A **function** is a named piece of program that takes some values, its *parameters*, does some work, and may give back one value, its *return value*. Functions let us write an idea once, give it a name, and use it many times; they also let us read a long program as a few meaningful steps rather than a thousand lines.

```
long lateFee(int daysKept, char memberType) {   // return type, name, parameters
    ...
    return base / 2;                             // give a value back
}
```

The first line is the function's **signature**: it returns a `long`, is called `lateFee`, and takes an `int` and a `char`. A function that returns nothing has the return type `void`. A function must be *declared* before it is called; in our short programs we simply define each function above `main`.

By default C++ passes parameters **by value**: the function receives a *copy* of each argument, and changing the copy does not change the caller's variable. Section 1.4.7 shows how to let a function change the caller's variable, and Chapter 6 treats the question fully.

1.4.6 Structs: many values under one name

A book at Pustaka Ceria has a code, a title, a writer and a category; a loan has a title, a number of days and a kind of member. Keeping each of these in a separate variable soon becomes impossible to manage. A **struct** groups related values into one new type:

```
struct Loan {
    string title;
    int daysKept;
    char memberType;
};

Loan first = {"Bumi Manusia", 10, 'S'};   // make one, giving each field a value
cout << first.title;                       // read a field with a dot
first.daysKept = 12;                       // change a field
```

The values inside a struct are its **fields** or *members*, and a dot selects one. A struct is the simplest way to describe a record, and an array of structs is the simplest table of records, one of the most common data structures of all.

Listing 1.7 brings every idea of this section together. Pustaka Ceria lends a book for seven days. A book returned late costs Rp1,000 a day for the first week of lateness and Rp2,000 a day after that; students pay half, lecturers the full amount and guests double. The program keeps six loans in an array of `Loan` structs and computes each fee with one function.

Listing 1.7 `late_fee.cpp`: constants, a struct, a function with `if` and `switch`, an array of structs, and a range-based `for` loop with `continue`.

```cpp
// late_fee.cpp - control flow, functions and a struct: Pustaka Ceria's late fees
#include <iostream>
#include <string>
using namespace std;

const int LOAN_DAYS = 7;                  // a book may be kept for 7 days

struct Loan {
    string title;
    int daysKept;
    char memberType;                      // 'S' student, 'L' lecturer, 'G' guest
};

// 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;
    }
}

int main() {
    Loan loans[] = {
        {"Laskar Pelangi", 5, 'S'},
        {"Bumi Manusia", 10, 'S'},
        {"Filosofi Kopi", 10, 'L'},
        {"Fisika Dasar", 16, 'G'},
        {"Negeri 5 Menara", 20, 'S'},
        {"Sejarah Nusantara", 9, 'X'},
    };
    long total = 0;
    for (const Loan& loan : loans) {
        long fee = lateFee(loan.daysKept, loan.memberType);
        if (fee < 0) {
```

Listing 1.7 (Continued) *late_fee.cpp: constants, a struct, a function with if and switch, an array of structs, and a range-based for loop with continue.*

```cpp
            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';
    return 0;
}
```

```
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
```

Check two of the lines by hand, because working a program by hand is the most useful habit this book can teach. *Fisika Dasar* was kept 16 days by a guest: it is $16 - 7 = 9$ days late, so the base fee is $7 \times 1000 + (9 - 7) \times 2000 = 7000 + 4000 = 11{,}000$, and a guest pays double, 22,000. *Negeri 5 Menara* was kept 20 days by a student: 13 days late, base $7000 + 6 \times 2000 = 19{,}000$, half of which is 9,500. The unknown member type `'X'` makes `lateFee` return -1; the loop reports it and `continue`s to the next loan, so that one bad record does not spoil the total.

In Java the natural home for a small group of values is a **record**, a class whose fields are fixed when it is made and which Java equips automatically with a constructor and accessor methods such as `loan.title()`. In Python it is a **dataclass**, and Python 3.10 and later can replace the chain of `if` statements on one value with a `match` statement. **Listing 1.8** and **Listing 1.9** print exactly the same seven lines as the C++ version.

Listing 1.8 `LateFee.java`: the late fees with a Java `record` and a `switch`.

```java
// LateFee.java - control flow, methods and a record: Pustaka Ceria's late fees
public class LateFee {
    static final int LOAN_DAYS = 7;           // a book may be kept for 7 days

    record Loan(String title, int daysKept, char memberType) { }

    // Returns the fee in rupiah, or -1 for an unknown member type.
    static 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;
        }
    }

    public static void main(String[] args) {
        Loan[] loans = {
            new Loan("Laskar Pelangi", 5, 'S'),
            new Loan("Bumi Manusia", 10, 'S'),
            new Loan("Filosofi Kopi", 10, 'L'),
            new Loan("Fisika Dasar", 16, 'G'),
            new Loan("Negeri 5 Menara", 20, 'S'),
            new Loan("Sejarah Nusantara", 9, 'X'),
        };
        long total = 0;
        for (Loan loan : loans) {
            long fee = lateFee(loan.daysKept(), loan.memberType());
            if (fee < 0) {
                System.out.println(loan.title() + ": unknown member type '"
                        + loan.memberType() + "'");
                continue;                     // skip it, go on with the next loan
            }
            System.out.println(loan.title() + " (" + loan.memberType() + ", "
                    + loan.daysKept() + " days): Rp" + fee);
            total += fee;
```

Listing 1.8 (Continued) *LateFee.java: the late fees with a Java record and a switch.*

```java
        }
        System.out.println("Total fees: Rp" + total);
    }
}
```

Listing 1.9 `late_fee.py`: the late fees with a `dataclass` and a `match` statement.

```python
# late_fee.py - control flow, functions and a dataclass: Pustaka Ceria's late fees
from dataclasses import dataclass

LOAN_DAYS = 7                           # a book may be kept for 7 days


@dataclass
class Loan:
    title: str
    days_kept: int
    member_type: str                    # 'S' student, 'L' lecturer, 'G' guest


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


loans = [
    Loan("Laskar Pelangi", 5, "S"),
    Loan("Bumi Manusia", 10, "S"),
    Loan("Filosofi Kopi", 10, "L"),
    Loan("Fisika Dasar", 16, "G"),
    Loan("Negeri 5 Menara", 20, "S"),
    Loan("Sejarah Nusantara", 9, "X"),
]
total = 0
for loan in loans:
    fee = late_fee(loan.days_kept, loan.member_type)
    if fee < 0:
        print(f"{loan.title}: unknown member type '{loan.member_type}'")
```

Listing 1.9 (Continued) *late_fee.py: the late fees with a dataclass and a match statement.*

```python
        continue                        # skip it, go on with the next loan
    print(f"{loan.title} ({loan.member_type}, {loan.days_kept} days): Rp{fee}")
    total += fee
print(f"Total fees: Rp{total}")
```

One small difference is easy to miss: C++ and Java compute half the fee with `/`, which for integers already throws the remainder away, but Python's `/` always gives a real number, so the Python version uses `//`, Python's integer division.

1.4.7 The address-of operator, and a first look at pointers

We have used the `&` operator once already, in **Listing 1.1**. It is time to look at it properly, because it is the door to half of this book.

Every variable has two things we may ask about: its **value**, the data stored in it, and its **address**, where in memory it is stored. Writing the variable's name gives its value. Writing `&` in front of the name gives its address. A variable whose value is an address is called a **pointer**. It is declared with a `*` after the type of the thing it points to:

```
int age = 20;
int* p = &age;      // p is a "pointer to int", holding the address of age
cout << *p;         // * in front of a pointer "follows" it: prints 20
*p = 21;            // writes through the pointer: age is now 21
```

The same `*` symbol plays two roles, which confuses everybody at first. In a *declaration*, `int* p`, it says "p is a pointer". In an *expression*, `*p`, it means "the thing p points to", and is called **dereferencing**. **Listing 1.10** shows all of this, and also the practical reason we care: a function that receives an *address* can change the caller's variable, while a function that receives a *copy* cannot.

Listing 1.10 `address_of.cpp`: `&` gives an address, `*` follows it, and a function that receives an address can change the caller's variable.

```
// address_of.cpp - the address-of operator, a first pointer, and two ways to pass
#include <iostream>
using namespace std;

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

void birthdayByAddress(int* p) {          // p holds the caller's address
    *p = *p + 1;
}

int main() {
    int age = 20;
    int* p = &age;                        // p now holds the address of age

    cout << "age          = " << age << '\n';
    cout << "&age         = " << &age << "  (an address)\n";
    cout << "p            = " << p << "  (the same address)\n";
    cout << "*p           = " << *p << "  (the value found there)\n";
    cout << "sizeof(age)  = " << sizeof(age) << " bytes\n";
    cout << "sizeof(p)    = " << sizeof(p) << " bytes\n";

    *p = 21;                              // write through the pointer
    cout << "after *p = 21, age = " << age << '\n';

    birthdayByValue(age);
    cout << "after birthdayByValue(age), age = " << age << '\n';
    birthdayByAddress(&age);
    cout << "after birthdayByAddress(&age), age = " << age << '\n';
    return 0;
}
```

```
age          = 20
&age         = 0x7ffd1c876cd4  (an address)
p            = 0x7ffd1c876cd4  (the same address)
*p           = 20  (the value found there)
sizeof(age)  = 4 bytes
sizeof(p)    = 8 bytes
after *p = 21, age = 21
after birthdayByValue(age), age = 21
after birthdayByAddress(&age), age = 22
```

`birthdayByValue(age)` receives a copy of 21 in its parameter `a`, adds one to the copy, and the copy disappears when the function returns; `age` is still 21. `birthdayByAddress(&age)` receives the

address of `age`, and `*p = *p + 1` changes the `int` stored at that address, which is `age` itself; afterwards `age` is 22. Notice also `sizeof(p) = 8`: on a 64-bit machine every address, and therefore every pointer, takes eight bytes, whatever it points to.

Java and Python have no `&` operator and never show addresses. They still have the underlying idea, however: in both, a variable of an object type holds a **reference**, which behaves like a pointer that the language manages for us. Assigning one reference to another makes two names for the *same* object, and a method that receives a reference to an array or a list can change its contents. **Listing 1.11** and **Listing 1.12** show the same experiment with a one-slot array or list playing the part of the address.

Listing 1.11 `AddressOf.java`: Java copies basic values into methods, but shares objects by reference.

```java
// AddressOf.java - no & in Java: values are copied, objects are shared
public class AddressOf {
    static void birthdayByValue(int a) {      // a is a copy
        a = a + 1;
    }

    static void birthdayInBox(int[] box) {    // box refers to the caller's array
        box[0] = box[0] + 1;
    }

    public static void main(String[] args) {
        int age = 20;
        int[] box = {20};                     // a one-slot array on the heap
        int[] sameBox = box;                  // a second reference, same array

        System.out.println("age          = " + age);
        System.out.println("box[0]       = " + box[0]);
        sameBox[0] = 21;                      // write through the second reference
        System.out.println("after sameBox[0] = 21, box[0] = " + box[0]);

        birthdayByValue(age);
        System.out.println("after birthdayByValue(age), age = " + age);
        birthdayInBox(box);
        System.out.println("after birthdayInBox(box), box[0] = " + box[0]);
    }
}
```

```
age          = 20
box[0]       = 20
after sameBox[0] = 21, box[0] = 21
after birthdayByValue(age), age = 20
after birthdayInBox(box), box[0] = 22
```

Listing 1.12 `address_of.py`: Python names refer to objects; `id()` shows that two names refer to the same list.

```python
# address_of.py - Python has no &: names refer to objects, and id() identifies them
def birthday_by_value(a):
    a = a + 1                           # rebinds the local name only


def birthday_in_box(box):
    box[0] = box[0] + 1                 # changes the caller's list


age = 20
box = [20]
same_box = box                          # a second name for the same list

print(f"age          = {age}")
print(f"box[0]       = {box[0]}")
same_box[0] = 21
print(f"after same_box[0] = 21, box[0] = {box[0]}")
print(f"id(box) == id(same_box) ? {id(box) == id(same_box)}")

birthday_by_value(age)
print(f"after birthday_by_value(age), age = {age}")
birthday_in_box(box)
print(f"after birthday_in_box(box), box[0] = {box[0]}")
```

```
age          = 20
box[0]       = 20
after same_box[0] = 21, box[0] = 21
id(box) == id(same_box) ? True
after birthday_by_value(age), age = 20
after birthday_in_box(box), box[0] = 22
```

This is as far as pointers go in this chapter: far enough to read an address and to understand why `catalog[i]` in the next section is the same thing as `*(catalog + i)`. Chapter 6 completes the story, and from Chapter 7 onward pointers carry the weight of every linked structure in the book.

1.5 Arrays in Depth

1.5.1 One row of numbered slots

Pustaka Ceria's first shelf is a single plank with numbered places, and its first data structure is exactly that.

> **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*. In C++, Java and Python the first index is 0, so an array of n elements has the indexes $0, 1, \ldots, n-1$ [1], [7].

In C++ an array is declared by giving the element type, the name and, in square brackets, the number of elements:

```
int loans[8];                               // 8 ints, values not yet set
int loans[8] = {12, 9, 15, 4, 20, 11, 6, 15};  // 8 ints, all set
int loans[] = {12, 9, 15, 4, 20, 11, 6, 15};   // the compiler counts: 8
int zeros[5] = {0};                         // the first is 0, the rest too
```

The number of elements must be known when the program is compiled, and it never changes. The element at index `i` is written `loans[i]`, and it can be read or changed like any variable: `loans[3] = 5;` puts 5 in the fourth place.

Figure 1.5 shows the array `loans` in memory. The eight `int`s take $8 \times 4 = 32$ bytes, from address 1000 to 1031, with nothing between them.

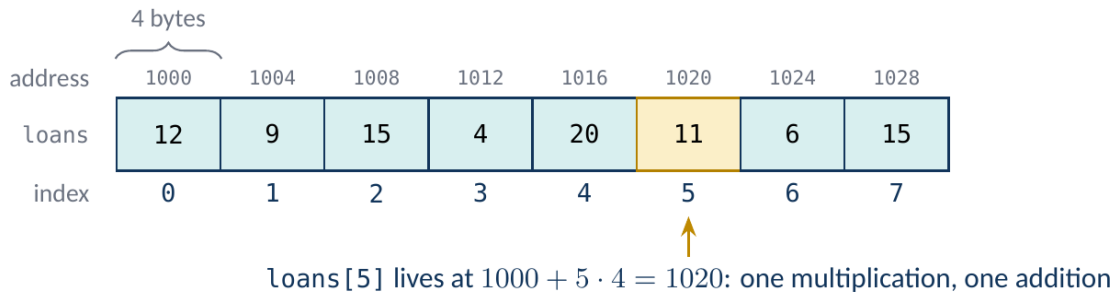


Figure 1.5 The array `loans` of eight `int`s in memory: each element takes 4 bytes, so element i begins at $1000 + 4i$.

Because the elements are all the same size and sit side by side, the address of any element can be *computed* instead of searched for. If the array begins at the **base address** b and each element takes s bytes, the element with index i begins at

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

In **Figure 1.5**, $b = 1000$ and $s = 4$, so `loans[5]` begins at $1000 + 5 \cdot 4 = 1020$, and `loans[0]` at 1000 itself. The C++ compiler inserts exactly this computation every time it meets `loans[i]` in a program: one multiplication, one addition, and a jump to the address. It does not matter whether the array has eight elements or eight million; reaching element i costs the same. Equation (1.2) is the whole secret of the array, and Section 1.6 turns it into our first statement about complexity.

1.5.2 Why indexes start at 0

Beginners often find it odd that the first element is number 0 and the last is number $n - 1$. Equation (1.2) explains it: the index is not "which element in the count", but "how many elements past the start". The first element is zero elements past the start, so its address is $b + 0 \cdot s = b$. If indexes started at 1, every access would have to compute $b + (i - 1) \cdot s$, one subtraction more, forever.

C++ makes the connection between arrays and addresses explicit. In most expressions the name of an array stands for the address of its first element, and adding an integer i to an address of type "pointer to `T`" moves it forward by i *elements*, that is, by $i \cdot$ sizeof(T) bytes. So

```
loans[i]    is exactly the same as    *(loans + i)
&loans[i]   is exactly the same as    loans + i
```

The square brackets are only a friendlier spelling of "start at the base, move i elements, follow the address". The programs of the next section check this equality, element by element.

Remember one sentence from this section: **an array of n elements has no element with index n.** The last one is $n - 1$.

1.5.3 An array of records: Pustaka Ceria's catalog

An array may hold elements of any type, including structs. Pustaka Ceria keeps four facts about every book: a five-character code such as `B1001`, a title, the name of its writer and a category. The header file `book.h` in **Listing 1.13** describes that record and three small helpers; both of the next C++ programs include it.

Listing 1.13 `book.h`: the `Book` record of Pustaka Ceria, with fixed-size character arrays, and helpers to fill and print it.

```cpp
// book.h - Pustaka Ceria's record type, shared by catalog_1d.cpp and shelves_2d.cpp
#ifndef BOOK_H
#define BOOK_H

#include <cstdio>
#include <cstring>

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';
}

inline void populateBook(Book& b, const char* id, const char* title,
                         const char* author, const char* category) {
    safeCopy(b.id, sizeof(b.id), id);
    safeCopy(b.title, sizeof(b.title), title);
    safeCopy(b.author, sizeof(b.author), author);
    safeCopy(b.category, sizeof(b.category), category);
}

inline void printCatalogHeader() {
    std::printf("%-6s %-26s %-20s %s\n", "ID", "TITLE", "AUTHOR", "CATEGORY");
}

inline void printBook(const Book& b) {
    std::printf("%-6s %-26s %-20s %s\n", b.id, b.title, b.author, b.category);
}

#endif
```

Each field is a fixed-size array of `char` rather than a `std::string`, and that is deliberate. A `std::string` keeps its characters elsewhere, on the heap, and holds only a small record of where they are; a struct of `char` arrays holds all its characters *inside* itself, so that the whole `Book` is one solid block of $6 + 100 + 60 + 30 = 196$ bytes. That makes it the cleanest possible example of how an array of records is laid out. The price is a limit: a title can have at most 99 characters, because one `char` is needed for the `'\0'` that marks the end of a C-style string. The helper `safeCopy` respects that limit, cutting a longer text short instead of writing past the end of the field.

Listing 1.14 declares an array of eight `Book`s, fills and prints it, and then measures where each element begins, relative to the first.

Listing 1.14 `catalog_1d.cpp`: Pustaka Ceria's catalog as an array of eight `Book`s, with the byte offset of every element.

```cpp
// catalog_1d.cpp - a one-dimensional array of Book, and where each element lives
#include <cstddef>
#include <cstdio>
#include "book.h"

const int N = 8;

int main() {
    Book catalog[N];
    populateBook(catalog[0], "B1001", "Laskar Pelangi",
                 "Andrea Hirata", "Fiksi");
    populateBook(catalog[1], "B1002", "Bumi Manusia",
                 "Pramoedya A. Toer", "Fiksi");
    populateBook(catalog[2], "B1003", "Filosofi Kopi",
                 "Dee Lestari", "Fiksi");
    populateBook(catalog[3], "B1004", "Sejarah Nusantara",
                 "Slamet Muljana", "Sejarah");
    populateBook(catalog[4], "B1005", "Algoritma & Struktur Data",
                 "Rinaldi Munir", "Teknologi");
    populateBook(catalog[5], "B1006", "Cerita Rakyat Nusantara",
                 "Tim Pustaka Ceria", "Anak-anak");
    populateBook(catalog[6], "B1007", "Fisika Dasar",
                 "Halliday & Resnick", "Sains");
    populateBook(catalog[7], "B1008", "Negeri 5 Menara",
                 "Ahmad Fuadi", "Fiksi");

    std::printf("Pustaka Ceria catalog, %d books\n", N);
    printCatalogHeader();
    for (int i = 0; i < N; i++) {
        printBook(catalog[i]);
    }

    std::printf("\nsizeof(Book) = %zu bytes\n", sizeof(Book));
    std::printf("field offsets: id %zu, title %zu, author %zu, category %zu\n",
                offsetof(Book, id), offsetof(Book, title),
                offsetof(Book, author), offsetof(Book, category));
```

Listing 1.14 (Continued) *catalog_1d.cpp: Pustaka Ceria's catalog as an array of eight Books, with the byte offset of every element.*

```cpp
    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");
    }
    return 0;
}
```

```
Pustaka Ceria catalog, 8 books
ID     TITLE                      AUTHOR               CATEGORY
B1001  Laskar Pelangi             Andrea Hirata        Fiksi
B1002  Bumi Manusia               Pramoedya A. Toer    Fiksi
B1003  Filosofi Kopi              Dee Lestari          Fiksi
B1004  Sejarah Nusantara          Slamet Muljana       Sejarah
B1005  Algoritma & Struktur Data  Rinaldi Munir        Teknologi
B1006  Cerita Rakyat Nusantara    Tim Pustaka Ceria    Anak-anak
B1007  Fisika Dasar               Halliday & Resnick   Sains
B1008  Negeri 5 Menara            Ahmad Fuadi          Fiksi

sizeof(Book) = 196 bytes
field offsets: id 0, title 6, author 106, category 166

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
```

The output confirms, byte for byte, the picture of **Figure 1.6**. The struct takes exactly 196 bytes, its fields begin at offsets 0, 6, 106 and 166 (the macro `offsetof` asks the compiler), and consecutive elements of the array begin exactly 196 bytes apart, as Equation (1.2) predicts with $s = 196$. The last column confirms that `&catalog[i]` and `catalog + i` are the same address for every `i`.

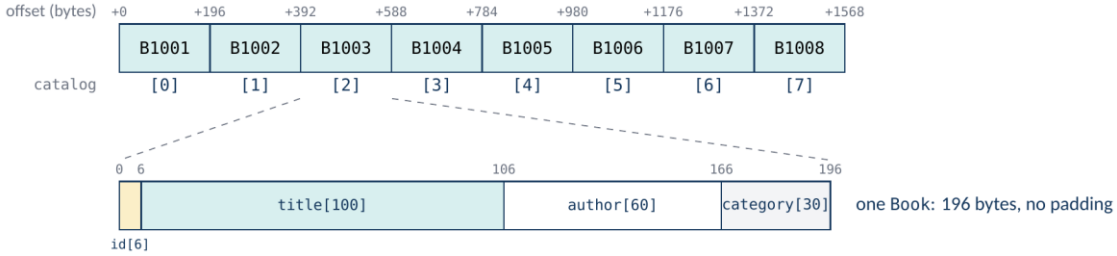


Figure 1.6 The catalog as eight 196-byte `Book` records laid end to end, and the inside of one record: its four fields begin at offsets 0, 6, 106 and 166.

Why exactly 196, with no padding like the byte we saw in **Figure 1.3**? Every field of `Book` is a `char` array, and a `char` may begin at any address, so the compiler has nothing to align. Exercise 1.14 adds an `int` field and watches the padding appear.

Java and Python keep a catalog differently, and it is important to see how. In Java, `new Book[8]` makes an array of eight *references*, all `null` at first; each `new Book(...)` makes a separate object somewhere on the heap, and the array slot receives a reference to it. A Python list is the same: a row of references to separate objects. **Listing 1.16**, with the class of **Listing 1.15**, and **Listing 1.18**, with the module of **Listing 1.17**, print the same catalog; instead of offsets, they show that copying an element copies only the reference.

Listing 1.15 `Book.java`: the `Book` record in Java, with `String` fields.

```java
// Book.java - Pustaka Ceria's record type, shared by Catalog1D and Shelves2D
public class Book {
    final String id;
    final String title;
    final String author;
    final String category;

    Book(String id, String title, String author, String category) {
        this.id = id;
        this.title = title;
        this.author = author;
        this.category = category;
    }

    static void printCatalogHeader() {
        System.out.printf("%-6s %-26s %-20s %s%n",
                "ID", "TITLE", "AUTHOR", "CATEGORY");
    }

    void print() {
        System.out.printf("%-6s %-26s %-20s %s%n", id, title, author, category);
    }
}
```

Listing 1.16 `Catalog1D.java`: the catalog as a Java array of `Book` references.

```java
// Catalog1D.java - a one-dimensional array of Book references
public class Catalog1D {
    static final int N = 8;

    public static void main(String[] args) {
        Book[] catalog = new Book[N];         // eight slots, each null at first
        catalog[0] = new Book("B1001", "Laskar Pelangi",
                "Andrea Hirata", "Fiksi");
        catalog[1] = new Book("B1002", "Bumi Manusia",
                "Pramoedya A. Toer", "Fiksi");
        catalog[2] = new Book("B1003", "Filosofi Kopi",
                "Dee Lestari", "Fiksi");
        catalog[3] = new Book("B1004", "Sejarah Nusantara",
                "Slamet Muljana", "Sejarah");
        catalog[4] = new Book("B1005", "Algoritma & Struktur Data",
                "Rinaldi Munir", "Teknologi");
        catalog[5] = new Book("B1006", "Cerita Rakyat Nusantara",
                "Tim Pustaka Ceria", "Anak-anak");
        catalog[6] = new Book("B1007", "Fisika Dasar",
                "Halliday & Resnick", "Sains");
        catalog[7] = new Book("B1008", "Negeri 5 Menara",
                "Ahmad Fuadi", "Fiksi");

        System.out.println("Pustaka Ceria catalog, " + N + " books");
        Book.printCatalogHeader();
        for (int i = 0; i < N; i++) {
            catalog[i].print();
        }

        System.out.println("\ncatalog.length = " + catalog.length);
        System.out.println(
                "Each slot holds a reference; each Book is its own object.");
        Book alias = catalog[4];              // copies the reference, not the book
        System.out.println("alias == catalog[4] ? " + (alias == catalog[4]));
    }
}
```

Listing 1.17 `book.py`: the `Book` record in Python, as a dataclass.

```python
# book.py - Pustaka Ceria's record type, shared by catalog_1d.py and shelves_2d.py
from dataclasses import dataclass


@dataclass
class Book:
    id: str
    title: str
    author: str
    category: str

    def print(self) -> None:
        print(f"{self.id:<6} {self.title:<26} {self.author:<20} {self.category}")


def print_catalog_header() -> None:
    print(f"{'ID':<6} {'TITLE':<26} {'AUTHOR':<20} CATEGORY")
```

Listing 1.18 `catalog_1d.py`: the catalog as a Python list.

```python
# catalog_1d.py - a Python list of Book references
from book import Book, print_catalog_header

catalog = [
    Book("B1001", "Laskar Pelangi", "Andrea Hirata", "Fiksi"),
    Book("B1002", "Bumi Manusia", "Pramoedya A. Toer", "Fiksi"),
    Book("B1003", "Filosofi Kopi", "Dee Lestari", "Fiksi"),
    Book("B1004", "Sejarah Nusantara", "Slamet Muljana", "Sejarah"),
    Book("B1005", "Algoritma & Struktur Data", "Rinaldi Munir", "Teknologi"),
    Book("B1006", "Cerita Rakyat Nusantara", "Tim Pustaka Ceria", "Anak-anak"),
    Book("B1007", "Fisika Dasar", "Halliday & Resnick", "Sains"),
    Book("B1008", "Negeri 5 Menara", "Ahmad Fuadi", "Fiksi"),
]

print(f"Pustaka Ceria catalog, {len(catalog)} books")
print_catalog_header()
for book in catalog:
    book.print()

print(f"\nlen(catalog) = {len(catalog)}")
print("Each slot holds a reference; each Book is its own object.")
alias = catalog[4]                      # copies the reference, not the book
print(f"alias is catalog[4] ? {alias is catalog[4]}")
```

The Java program prints the same catalog as the C++ one, followed by:

```
catalog.length = 8
Each slot holds a reference; each Book is its own object.
alias == catalog[4] ? true
```

and the Python program the same catalog followed by:

```
len(catalog) = 8
Each slot holds a reference; each Book is its own object.
alias is catalog[4] ? True
```

So the Java and Python arrays of books are contiguous rows of *references*, eight bytes each, while the books themselves are scattered over the heap. The formula of Equation (1.2) still finds any *reference* in one step; one more step follows it to the book.

1.5.4 Staying inside the bounds

What happens if a program asks for `loans[8]`, or `loans[-1]`, in an array of eight elements? The answer is the most important difference between C++ on one side and Java and Python on the other.

C++ does not check. The expression `loans[8]` simply computes $b + 8 \cdot 4$ and reads whatever bytes lie there: part of another variable, part of a stack frame, or memory the program does not own at all. The C++ standard calls this **undefined behavior**: anything may happen, including, most dangerously, nothing visible at all. The program may print a strange number, crash much later in an unrelated place, or run correctly on your machine and fail on another. Writing past the end of an array is how some of the most famous security holes in history were made, because an attacker who can choose what gets written can overwrite the data, or the return address, of a stack frame.

C++ gives us two defences. The first is discipline: every loop over an array runs `for (int i = 0; i < n; i++)`, with `<` and never `<=`. The second is the checked accessor `at(i)` of the standard containers `std::array` and `std::vector` (Section 1.5.8), which verifies the index and *throws an exception* of type `out_of_range` if it is invalid. **Listing 1.19** uses both.

Listing 1.19 `bounds.cpp`: indexing inside the bounds, and the checked accessor `at()` refusing an index that is out of range.

```
// bounds.cpp - staying inside an array, and what the checked accessor does
#include <array>
#include <iostream>
#include <stdexcept>
#include <vector>
using namespace std;

int main() {
    array<int, 5> a = {10, 20, 30, 40, 50};
    vector<int> v = {10, 20, 30, 40, 50};

    cout << "a.size() = " << a.size() << ", valid indexes 0.." << a.size() - 1
         << '\n';
    cout << "a[0] = " << a[0] << ", a[4] = " << a[4] << '\n';

    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[5] would compile, but reading it is undefined behavior: never do it.
    return 0;
}
```

```
a.size() = 5, valid indexes 0..4
a[0] = 10, a[4] = 50
v.at(3) = 40
v.at(5) refused: vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)
```

A `try` block runs code that might throw; if it does, control jumps to the matching `catch` block, and `e.what()` describes the problem. The message is written by the library, and the one shown is that of GCC's library; other compilers word it differently.

Java and Python always check. Every index is compared with the array's length before use, and an invalid one raises an exception, `ArrayIndexOutOfBoundsException` in Java and `IndexError` in Python, at the exact line where the mistake was made. **Listing 1.20** and **Listing 1.21** show it.

Listing 1.20 `Bounds.java`: Java checks every index, for arrays and for `ArrayList`.

```java
// Bounds.java - Java checks every index, and throws when it is out of range
import java.util.ArrayList;
import java.util.List;

public class Bounds {
    public static void main(String[] args) {
        int[] a = {10, 20, 30, 40, 50};
        List<Integer> v = new ArrayList<>(List.of(10, 20, 30, 40, 50));

        System.out.println("a.length = " + a.length + ", valid indexes 0.."
                + (a.length - 1));
        System.out.println("a[0] = " + a[0] + ", a[4] = " + a[4]);

        for (int i : new int[] {3, 5}) {
            try {
                int value = a[i];              // throws if i is not a valid index
                System.out.println("a[" + i + "] = " + value);
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("a[" + i + "] refused: " + e.getMessage());
            }
        }
        try {
            System.out.println("v.get(5) = " + v.get(5));
        } catch (IndexOutOfBoundsException e) {
            System.out.println("v.get(5) refused: " + e.getMessage());
        }
    }
}
```

```
a.length = 5, valid indexes 0..4
a[0] = 10, a[4] = 50
a[3] = 40
a[5] refused: Index 5 out of bounds for length 5
v.get(5) refused: Index 5 out of bounds for length 5
```

Listing 1.21 `bounds.py`: Python checks every index, and also accepts negative ones.

```python
# bounds.py - Python checks every index, and negative indexes count from the end
a = [10, 20, 30, 40, 50]

print(f"len(a) = {len(a)}, valid indexes 0..{len(a) - 1} (and -1..-{len(a)})")
print(f"a[0] = {a[0]}, a[4] = {a[4]}")

for i in (3, 5):
    try:
        value = a[i]                       # raises IndexError if i is not valid
        print(f"a[{i}] = {value}")
    except IndexError as e:
        print(f"a[{i}] refused: {e}")
print(f"a[-1] = {a[-1]}, a[-5] = {a[-5]}")
```

```
len(a) = 5, valid indexes 0..4 (and -1..-5)
a[0] = 10, a[4] = 50
a[3] = 40
a[5] refused: list index out of range
a[-1] = 50, a[-5] = 10
```

Python adds one twist: a *negative* index counts from the end, so `a[-1]` is the last element and `a[-5]`, in a list of five, is the first. The valid indexes of a list of length n are therefore $-n$ to $n-1$. This

is a convenience in Python, and a trap for anyone who carries the habit back to C++, where `a[-1]` is simply the memory just *before* the array.

1.5.5 Walking an array: the everyday traversals

A great many array algorithms are the same loop with a different body: visit every element once, from index 0 to $n-1$, and do something with it. This is called a **traversal**. **Listing 1.22** collects the five traversals every programmer should be able to write without thinking: the sum, the position of the largest element, the first position of a given value, the number of times a value occurs, and reversing the array in place.

Listing 1.22 `array_ops.cpp`: sum, position of the maximum, linear search, counting and in-place reversal of an array.

```
// array_ops.cpp - the everyday traversals of a one-dimensional array
#include <iostream>
using namespace std;

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;
}

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;
}

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;
    }
}
```

Listing 1.22 (Continued) *array_ops.cpp: sum, position of the maximum, linear search, counting and in-place reversal of an array.*

```cpp
void print(const int a[], int n) {
    cout << '[';
    for (int i = 0; i < n; i++) cout << (i ? ", " : "") << a[i];
    cout << "]\n";
}

int main() {
    int loans[] = {12, 9, 15, 4, 20, 11, 6, 15};
    int n = sizeof loans / sizeof loans[0];    // works here, not in a function

    cout << "loans      = "; print(loans, n);
    cout << "n          = " << n << '\n';
    cout << "sum        = " << sumOf(loans, n) << '\n';
    int m = indexOfMax(loans, n);
    cout << "max        = " << loans[m] << " at index " << m << '\n';
    cout << "find 15    = " << findFirst(loans, n, 15) << '\n';
    cout << "find 99    = " << findFirst(loans, n, 99) << '\n';
    cout << "count 15   = " << countOf(loans, n, 15) << '\n';
    reverseInPlace(loans, n);
    cout << "reversed   = "; print(loans, n);
    return 0;
}
```

```
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]
```

Three details make these functions correct, and each is a lesson that later chapters rely on.

An array passed to a function does not carry its length. A parameter written `const int a[]` is, to the compiler, just a pointer `const int* a` to the first element; the function cannot tell how many elements follow. That is why every function takes `n` as a second parameter. In `main` the idiom `sizeof loans / sizeof loans[0]` computes the length, $32/4 = 8$, because there `loans` is still a real array whose size the compiler knows; inside a function the same expression would divide the size of a *pointer* by the size of an `int`, and give the wrong answer. The word `const` promises that the function will not change the array, and the compiler holds it to that promise.

`findFirst` returns -1 for "not found". Since every valid index is 0 or more, -1 cannot be mistaken for a position. This convention, "a position, or -1", is the contract of every search algorithm in Chapter 2.

`reverseInPlace` needs no second array. Two indexes start at the two ends, `i = 0` and `j = n - 1`, swap their elements, and step toward each other until they meet. Each swap uses a temporary variable `t`, because writing `a[i] = a[j]; a[j] = a[i];` would lose the old `a[i]`. An array of n elements needs $\lfloor n/2 \rfloor$ swaps, where $\lfloor x \rfloor$, the *floor* of x, means x rounded down to a whole number.

Java arrays know their own length, `a.length`, so the Java functions in **Listing 1.23** need no `n` parameter; Python lists know theirs through `len(a)`, and Python swaps two elements in one line, `a[i], a[j] = a[j], a[i]`, as **Listing 1.24** shows. Both print exactly the output above.

Listing 1.23 `ArrayOps.java`: the five traversals in Java.

```java
// ArrayOps.java - the everyday traversals of a one-dimensional array
import java.util.Arrays;

public class ArrayOps {
    static long sumOf(int[] a) {
        long s = 0;
        for (int x : a) s += x;
        return s;
    }

    static int indexOfMax(int[] a) {
        int best = 0;
        for (int i = 1; i < a.length; i++) {
            if (a[i] > a[best]) best = i;
        }
        return best;
    }

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

    static int countOf(int[] a, int key) {
        int c = 0;
        for (int x : a) {
            if (x == key) c++;
        }
        return c;
    }

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

    public static void main(String[] args) {
        int[] loans = {12, 9, 15, 4, 20, 11, 6, 15};
        int n = loans.length;                 // an array knows its own length
```

Listing 1.23 (Continued) *ArrayOps.java: the five traversals in Java.*

```java
        System.out.println("loans      = " + Arrays.toString(loans));
        System.out.println("n          = " + n);
        System.out.println("sum        = " + sumOf(loans));
        int m = indexOfMax(loans);
        System.out.println("max        = " + loans[m] + " at index " + m);
        System.out.println("find 15    = " + findFirst(loans, 15));
        System.out.println("find 99    = " + findFirst(loans, 99));
        System.out.println("count 15   = " + countOf(loans, 15));
        reverseInPlace(loans);
        System.out.println("reversed   = " + Arrays.toString(loans));
    }
}
```

Listing 1.24 `array_ops.py`: the five traversals in Python.

```python
# array_ops.py - the everyday traversals of a one-dimensional list
def sum_of(a):
    s = 0
    for x in a:
        s += x
    return s


def index_of_max(a):
    best = 0
    for i in range(1, len(a)):
        if a[i] > a[best]:
            best = i
    return best


def find_first(a, key):                  # -1 means "not found"
    for i in range(len(a)):
        if a[i] == key:
            return i
    return -1


def count_of(a, key):
    c = 0
    for x in a:
        if x == key:
            c += 1
    return c


def reverse_in_place(a):
    i, j = 0, len(a) - 1
    while i < j:
        a[i], a[j] = a[j], a[i]
        i += 1
        j -= 1


loans = [12, 9, 15, 4, 20, 11, 6, 15]
n = len(loans)                           # a list knows its own length
```

Listing 1.24 (Continued) *array_ops.py: the five traversals in Python.*

```python
print(f"loans      = {loans}")
print(f"n          = {n}")
print(f"sum        = {sum_of(loans)}")
m = index_of_max(loans)
print(f"max        = {loans[m]} at index {m}")
print(f"find 15    = {find_first(loans, 15)}")
print(f"find 99    = {find_first(loans, 99)}")
print(f"count 15   = {count_of(loans, 15)}")
reverse_in_place(loans)
print(f"reversed   = {loans}")
```

1.5.6 Two-dimensional arrays and row-major layout

Pustaka Ceria's building has real shelves, and each shelf has a fixed number of slots. A one-dimensional catalog does not capture that shape; a **two-dimensional array** does. In C++,

```
Book shelf[2][4];      // 2 shelves, each with 4 slots
shelf[1][2]            // shelf 1, slot 2
```

declares a grid of $2 \times 4 = 8$ books. The first index chooses the **row** (the shelf), the second the **column** (the slot). Rows and columns are both numbered from 0.

But memory is a single row of bytes, not a grid. How does a grid fit into a row? C++ lays out a two-dimensional array in **row-major order**: the whole of row 0 first, then the whole of row 1 immediately after it, and so on, exactly as if the grid had been unrolled into one long one-dimensional array, row by row [7], [8]. **Figure 1.7** shows it.

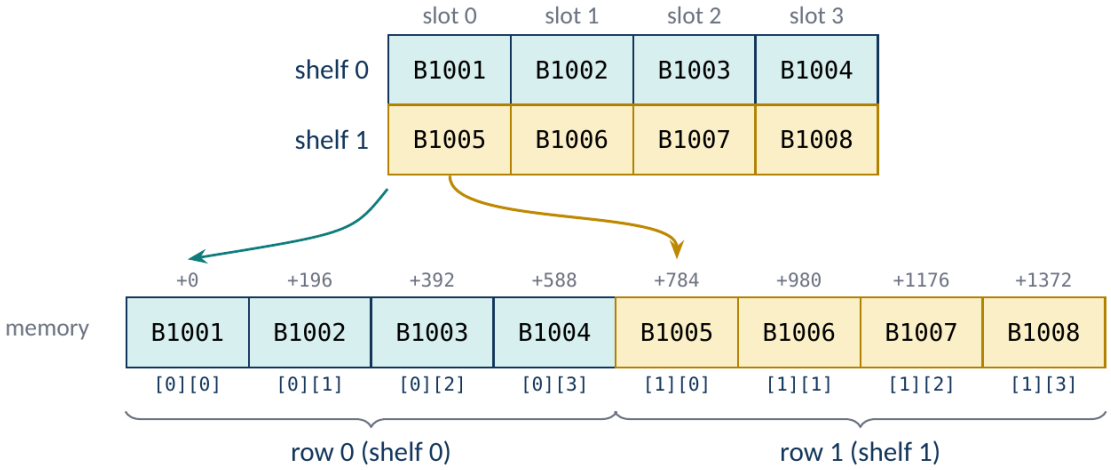


Figure 1.7 The 2×4 array `shelf` as a grid, and as it really lies in memory: row 0 first, then row 1, each element 196 bytes from the last.

In the unrolled row, the element `shelf[s][p]` is preceded by s complete rows of `SLOTS` elements each, plus p elements of its own row. Its position in the unrolled row is therefore $s \cdot \text{SLOTS} + p$, and by Equation (1.2) its address is

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

where C is the number of columns, r and c the row and column indexes, b the base address and s the size of one element. Notice what does *not* appear in Equation (1.3): the number of rows. To find an element, the compiler needs to know how long each row is, but not how many rows there are. That is why a C++ function receiving a two-dimensional array must be told the number of columns in its parameter, `void f(Book a[][4], int rows)`, while the number of rows may be passed separately.

Listing 1.25 fills the shelves with the same eight books and checks Equation (1.3) for every element.

Listing 1.25 `shelves_2d.cpp`: a two-dimensional array of `Book`s, and the row-major formula checked for every element.

```
// shelves_2d.cpp - a two-dimensional array of Book and its row-major layout
#include <cstdio>
#include "book.h"

const int SHELVES = 2;
const int SLOTS = 4;

int main() {
    Book shelf[SHELVES][SLOTS];
    populateBook(shelf[0][0], "B1001", "Laskar Pelangi",
                 "Andrea Hirata", "Fiksi");
    populateBook(shelf[0][1], "B1002", "Bumi Manusia",
                 "Pramoedya A. Toer", "Fiksi");
    populateBook(shelf[0][2], "B1003", "Filosofi Kopi",
                 "Dee Lestari", "Fiksi");
    populateBook(shelf[0][3], "B1004", "Sejarah Nusantara",
                 "Slamet Muljana", "Sejarah");
    populateBook(shelf[1][0], "B1005", "Algoritma & Struktur Data",
                 "Rinaldi Munir", "Teknologi");
    populateBook(shelf[1][1], "B1006", "Cerita Rakyat Nusantara",
                 "Tim Pustaka Ceria", "Anak-anak");
    populateBook(shelf[1][2], "B1007", "Fisika Dasar",
                 "Halliday & Resnick", "Sains");
    populateBook(shelf[1][3], "B1008", "Negeri 5 Menara",
                 "Ahmad Fuadi", "Fiksi");

    for (int s = 0; s < SHELVES; s++) {
        std::printf("shelf %d:", s);
        for (int p = 0; p < SLOTS; p++) std::printf(" %s", shelf[s][p].id);
        std::printf("\n");
    }

    const char* base = reinterpret_cast<const char*>(&shelf[0][0]);
    std::printf("\nsizeof(Book) = %zu, SLOTS = %d\n", sizeof(Book), SLOTS);
    std::printf("%-2s %-2s %-6s %-10s %-24s %s\n",
                "s", "p", "id", "offset", "(s*SLOTS+p)*sizeof(Book)", "same?");
    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");
        }
    }
    return 0;
```

Listing 1.25 (Continued) *shelves_2d.cpp: a two-dimensional array of Books, and the row-major formula checked for every element.*

```
}
```

```
shelf 0: B1001 B1002 B1003 B1004
shelf 1: B1005 B1006 B1007 B1008

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
1  3  B1008  1372       1372                     yes
```

Take one line and do the arithmetic yourself: `shelf[1][2]` holds B1007, and $(1 \cdot 4 + 2) \cdot 196 = 6 \cdot 196 = 1176$, exactly the measured offset.

Row-major order is a choice, not a law of nature. The language Fortran, and numerical libraries descended from it, use **column-major** order, storing column 0 first, then column 1. For an array with R rows, the column-major address is

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

Code that computes addresses by hand, or that passes arrays between languages, must know which order it is dealing with; mixing them up is a real and easy-to-miss source of bugs. Exercise 1.4 practices both formulas.

Java and Python have no true two-dimensional arrays. A Java `Book[][]` is an *array of arrays*: an array of two references, each referring to a separate row array of four references. A Python list of lists is the same idea. The rows need not even have the same length, and in Java such an array is called **jagged**. **Listing 1.26** and **Listing 1.27** therefore cannot measure offsets; they print the shelves and the row-major *position* $s \cdot \text{SLOTS} + p$ that Equation (1.3) would use if the grid were packed into one row.

Listing 1.26 `Shelves2D.java`: a Java array of row arrays.

```java
// Shelves2D.java - a two-dimensional array: an array of row arrays
public class Shelves2D {
    static final int SHELVES = 2;
    static final int SLOTS = 4;

    public static void main(String[] args) {
        Book[][] shelf = new Book[SHELVES][SLOTS];
        shelf[0][0] = new Book("B1001", "Laskar Pelangi",
                "Andrea Hirata", "Fiksi");
        shelf[0][1] = new Book("B1002", "Bumi Manusia",
                "Pramoedya A. Toer", "Fiksi");
        shelf[0][2] = new Book("B1003", "Filosofi Kopi",
                "Dee Lestari", "Fiksi");
        shelf[0][3] = new Book("B1004", "Sejarah Nusantara",
                "Slamet Muljana", "Sejarah");
        shelf[1][0] = new Book("B1005", "Algoritma & Struktur Data",
                "Rinaldi Munir", "Teknologi");
        shelf[1][1] = new Book("B1006", "Cerita Rakyat Nusantara",
                "Tim Pustaka Ceria", "Anak-anak");
        shelf[1][2] = new Book("B1007", "Fisika Dasar",
                "Halliday & Resnick", "Sains");
        shelf[1][3] = new Book("B1008", "Negeri 5 Menara",
                "Ahmad Fuadi", "Fiksi");

        for (int s = 0; s < SHELVES; s++) {
            System.out.print("shelf " + s + ":");
            for (int p = 0; p < SLOTS; p++) System.out.print(" " + shelf[s][p].id);
            System.out.println();
        }

        System.out.println("\nshelf.length = " + shelf.length
                + " rows, shelf[0].length = " + shelf[0].length + " slots");
        System.out.printf("%-2s %-2s %-6s %s%n",
                "s", "p", "id", "row-major position s*SLOTS+p");
        for (int s = 0; s < SHELVES; s++) {
            for (int p = 0; p < SLOTS; p++) {
                System.out.printf("%-2d %-2d %-6s %d%n",
                        s, p, shelf[s][p].id, s * SLOTS + p);
            }
        }
    }
}
```

```
shelf 0: B1001 B1002 B1003 B1004
shelf 1: B1005 B1006 B1007 B1008

shelf.length = 2 rows, shelf[0].length = 4 slots
s  p  id     row-major position s*SLOTS+p
0  0  B1001  0
0  1  B1002  1
0  2  B1003  2
0  3  B1004  3
1  0  B1005  4
1  1  B1006  5
1  2  B1007  6
1  3  B1008  7
```

Listing 1.27 `shelves_2d.py`: a Python list of row lists.

```python
# shelves_2d.py - a two-dimensional list: a list of row lists
from book import Book

SHELVES = 2
SLOTS = 4

shelf = [
    [Book("B1001", "Laskar Pelangi", "Andrea Hirata", "Fiksi"),
     Book("B1002", "Bumi Manusia", "Pramoedya A. Toer", "Fiksi"),
     Book("B1003", "Filosofi Kopi", "Dee Lestari", "Fiksi"),
     Book("B1004", "Sejarah Nusantara", "Slamet Muljana", "Sejarah")],
    [Book("B1005", "Algoritma & Struktur Data", "Rinaldi Munir", "Teknologi"),
     Book("B1006", "Cerita Rakyat Nusantara", "Tim Pustaka Ceria", "Anak-anak"),
     Book("B1007", "Fisika Dasar", "Halliday & Resnick", "Sains"),
     Book("B1008", "Negeri 5 Menara", "Ahmad Fuadi", "Fiksi")],
]

for s in range(SHELVES):
    print(f"shelf {s}:", " ".join(book.id for book in shelf[s]))

print(f"\nlen(shelf) = {len(shelf)} rows, len(shelf[0]) = {len(shelf[0])} slots")
print(f"{'s':<2} {'p':<2} {'id':<6} row-major position s*SLOTS+p")
for s in range(SHELVES):
    for p in range(SLOTS):
        print(f"{s:<2} {p:<2} {shelf[s][p].id:<6} {s * SLOTS + p}")
```

The Python program prints the same lines, with `len(shelf)` and `len(shelf[0])` in place of `shelf.length` and `shelf[0].length`.

1.5.7 Tables of numbers

The most common two-dimensional array holds numbers: a table of measurements, a spreadsheet, a game board, the pixels of an image. Pustaka Ceria counts how many books were borrowed from each of three shelves in each of four weeks, and wants the total of every row, of every column, and of the whole table. **Listing 1.28** does it with two nested loops: the outer loop walks the rows, the inner loop walks the columns of one row.

Listing 1.28 `loan_table.cpp`: a 3 × 4 table of loans with row totals, column totals and a grand total.

```cpp
// loan_table.cpp - a 2-D array of numbers: row totals and column totals
#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

const int ROWS = 3;                       // three shelves
const int COLS = 4;                       // four weeks

int main() {
    string shelfName[ROWS] = {"Fiksi", "Sejarah", "Sains"};
    int loans[ROWS][COLS] = {
        {31, 28, 35, 40},
        {12, 15, 9, 14},
        {20, 22, 18, 25},
    };

    cout << left << setw(8) << "shelf";
    for (int c = 0; c < COLS; c++) {
        cout << right << setw(5) << ("W" + to_string(c + 1));
    }
    cout << setw(7) << "total" << '\n';

    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;
    }
    cout << left << setw(8) << "total";
    for (int c = 0; c < COLS; c++) cout << right << setw(5) << columnTotal[c];
    cout << setw(7) << grandTotal << '\n';
    return 0;
}
```

```
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
```

The row totals are accumulated in the variable `rowTotal`, reset for each row; the column totals need one running total per column, so they live in the array `columnTotal`, declared with `= {0}`, which sets every element to zero. The manipulators `left`, `right` and `setw(k)` print each value in a column *k* characters wide.

The loops visit the table row by row, in the same order in which the elements lie in memory. That is not only natural but fast: modern processors fetch memory in blocks of 64 bytes, called *cache lines*, so reading consecutive addresses is much quicker than jumping about. For a small table the difference is invisible; for a table of millions of numbers, swapping the two loops so that the inner one runs down a column can make the same program several times slower [5].

The Java and Python versions in **Listing 1.29** and **Listing 1.30** print the same table.

Listing 1.29 `LoanTable.java`: the loan table in Java.

```java
// LoanTable.java - a 2-D array of numbers: row totals and column totals
public class LoanTable {
    static final int ROWS = 3;                // three shelves
    static final int COLS = 4;                // four weeks

    public static void main(String[] args) {
        String[] shelfName = {"Fiksi", "Sejarah", "Sains"};
        int[][] loans = {
            {31, 28, 35, 40},
            {12, 15, 9, 14},
            {20, 22, 18, 25},
        };

        System.out.printf("%-8s", "shelf");
        for (int c = 0; c < COLS; c++) System.out.printf("%5s", "W" + (c + 1));
        System.out.printf("%7s%n", "total");

        int[] columnTotal = new int[COLS];    // every element starts at 0
        int grandTotal = 0;
        for (int r = 0; r < ROWS; r++) {
            int rowTotal = 0;
            System.out.printf("%-8s", shelfName[r]);
            for (int c = 0; c < COLS; c++) {
                System.out.printf("%5d", loans[r][c]);
                rowTotal += loans[r][c];
                columnTotal[c] += loans[r][c];
            }
            System.out.printf("%7d%n", rowTotal);
            grandTotal += rowTotal;
        }
        System.out.printf("%-8s", "total");
        for (int c = 0; c < COLS; c++) System.out.printf("%5d", columnTotal[c]);
        System.out.printf("%7d%n", grandTotal);
    }
}
```

Listing 1.30 `loan_table.py`: the loan table in Python.

```python
# loan_table.py - a 2-D list of numbers: row totals and column totals
ROWS = 3                                  # three shelves
COLS = 4                                  # four weeks

shelf_name = ["Fiksi", "Sejarah", "Sains"]
loans = [
    [31, 28, 35, 40],
    [12, 15, 9, 14],
    [20, 22, 18, 25],
]

header = "".join(f"{'W' + str(c + 1):>5}" for c in range(COLS))
print(f"{'shelf':<8}" + header + f"{'total':>7}")

column_total = [0] * COLS                 # every element starts at 0
grand_total = 0
for r in range(ROWS):
    row_total = 0
    line = f"{shelf_name[r]:<8}"
    for c in range(COLS):
        line += f"{loans[r][c]:>5}"
        row_total += loans[r][c]
        column_total[c] += loans[r][c]
    print(line + f"{row_total:>7}")
    grand_total += row_total
totals = "".join(f"{t:>5}" for t in column_total)
print(f"{'total':<8}" + totals + f"{grand_total:>7}")
```

Python has one trap with tables that has caught nearly every Python programmer once. The expression `[0] * 4` makes a list of four zeros, and it is tempting to make a 3 × 4 table as `[[0] * 4] * 3`. But `*` on a list repeats *references*: the result is a list of three references to the *same* row. **Listing 1.31** shows what happens when one element is changed.

Listing 1.31 `aliasing_rows.py`: `[[0] * 4] * 3` makes three references to one row; a list comprehension makes three rows.

```
# 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
```

Changing `wrong[0][1]` appears to change all three rows, because there is only one row. The correct idiom, `[[0] * 4 for _ in range(3)]`, runs `[0] * 4` three times and makes three separate rows.

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

The arrays we have used so far are the ones C++ inherited from C, often called **raw** or **built-in arrays**. They are as fast as memory allows, and they have three weaknesses: they forget their length when passed to a function, they cannot be copied with `=`, and their size is fixed when the program is compiled. The C++ standard library offers two containers that keep the speed and remove the weaknesses [9].

`std::array<T, N>`, from `<array>`, is a fixed-size array of `N` elements of type `T` that knows its own size (`a.size()`), can be copied and compared with `=` and `==`, offers the checked accessor `a.at(i)`, and costs nothing extra: its elements lie in memory exactly as a raw array's do. Wherever this book needs an array whose size is known in advance, `std::array` is the better choice.

`std::vector<T>`, from `<vector>`, is an array whose size can change while the program runs. `v.push_back(x)` adds `x` at the end, `v.pop_back()` removes the last element, `v.size()` gives the number of elements, and `v[i]` and `v.at(i)` read and write them just as for an array. It is the most used container in C++, and it is worth understanding how it can grow, because at first sight an array *cannot*.

A vector keeps three things: the address of a block of memory on the heap, its **size** (how many elements are in use) and its **capacity** (how many elements the block has room for). As long as `size < capacity`, `push_back` simply writes the new element into the next free slot. When the block is full, `push_back` must

1. ask the heap for a new, larger block,
2. copy (or move) every existing element into it,
3. free the old block, and
4. only then add the new element.

Figure 1.8 shows one such regrowth, and **Listing 1.32** watches a vector grow from empty to 100 elements, printing a line every time its capacity changes.

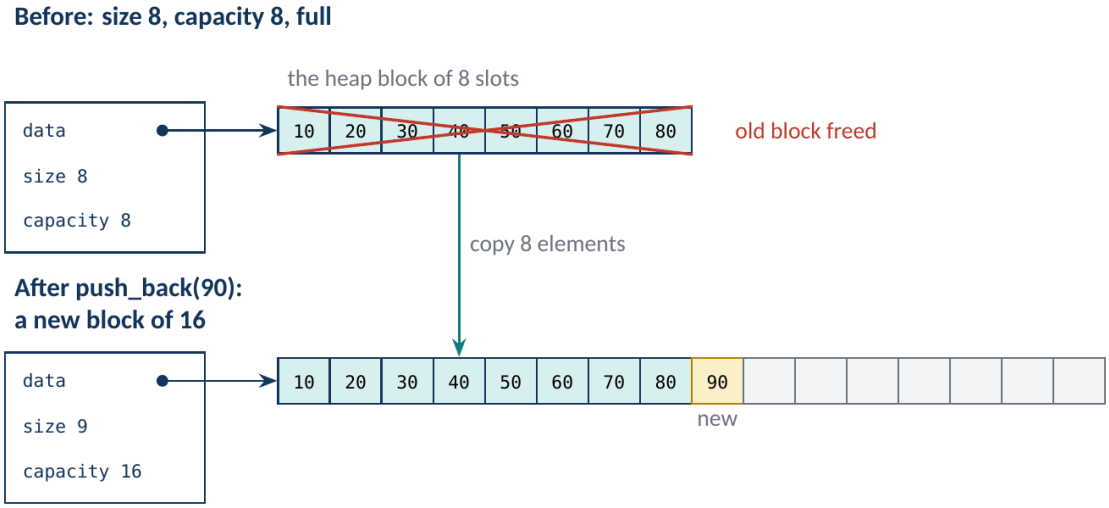


Figure 1.8 A full `std::vector` of capacity 8 receives a ninth element: a block of 16 is allocated, the 8 elements are copied, the old block is freed, and 90 is added.

Listing 1.32 `vector_growth.cpp`: a fixed-size `std::array`, and a `std::vector` doubling its capacity as it grows.

```
// vector_growth.cpp - std::array has a fixed size; std::vector grows by itself
#include <array>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    array<int, 5> fixedFive = {1, 2, 3, 4, 5};
    cout << "std::array<int, 5> holds " << fixedFive.size() << " ints, always\n\n";

    vector<int> v;
    size_t lastCapacity = v.capacity();
    cout << "empty vector: size 0, capacity " << lastCapacity << '\n';
    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();
        }
    }
    cout << "after 100 push_backs: size " << v.size() << ", capacity "
         << v.capacity() << ", elements moved " << moves << '\n';
    return 0;
}
```

```
std::array<int, 5> holds 5 ints, always

empty vector: size 0, capacity 0
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)
after 100 push_backs: size 100, capacity 128, elements moved 127
```

With GCC's library, the capacity *doubles* each time: 1, 2, 4, 8, and so on up to 128. (The C++ standard does not fix the growth factor; Microsoft's library multiplies by 1.5, but the reasoning below is the same for any constant factor.) Copying is expensive, so why is doubling a good idea? Count the copies. To reach n elements, the vector grew at sizes 1,2,4, … up to the largest power of two below n, and each growth copied that many elements. The total number of copies is at most

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

because the largest power 2^k is less than n. For $n = 100$ the program counted $1 + 2 + 4 + 8 + 16 + 32 + 64 = 127$ copies, indeed less than 200. So although an individual `push_back` occasionally copies everything, the *average* cost per `push_back`, over a long run, is fewer than two copies: a constant. Section 1.6.5 gives this idea its proper name. A vector that grew by one slot at a time would instead copy $1 + 2 + \cdots + (n-1) = n(n-1)/2$ elements, nearly 5,000 for $n = 100$, and five billion for $n = 100{,}000$.

If you know in advance roughly how many elements a vector will hold, `v.reserve(n)` sets the capacity once and avoids every regrowth.

1.5.9 Java arrays and `ArrayList`

A Java array is an object on the heap. `int[] a = new int[8];` makes one with eight elements, all set to `0`, and the variable `a` holds a reference to it. The array knows its length, `a.length`, checks every index, and cannot change its length after it is made. Its elements are stored contiguously, so Equation (1.2) applies to them exactly: for an `int[]` the elements are the numbers themselves, and for an array of objects they are references [10].

Java's growable array is `ArrayList<T>`, from `java.util`. It works exactly like `std::vector`: an internal array, a size, a capacity, and regrowth when full. Its main operations are `add(x)`, `get(i)`, `set(i, x)`, `remove(i)` and `size()`. An `ArrayList` does not reveal its capacity, so **Listing 1.33** replays the rule that the OpenJDK implementation uses: the first `add` allocates room for 10 elements, and a full list grows to 1.5 times its old capacity, rounded down [11].

Listing 1.33 `ListGrowth.java`: a fixed-length Java array, and the growth steps of an `ArrayList` from empty to 100 elements.

```java
// ListGrowth.java - a Java array has a fixed length; an ArrayList grows by itself
import java.util.ArrayList;

public class ListGrowth {
    public static void main(String[] args) {
        int[] fixedFive = {1, 2, 3, 4, 5};
        System.out.println("int[] fixedFive holds " + fixedFive.length
                + " ints, always\n");

        // ArrayList hides its capacity, so we replay the growth rule OpenJDK uses:
        // the first add allocates 10 slots, and a full list grows to old + old / 2.
        ArrayList<Integer> list = new ArrayList<>();
        int capacity = 0;
        int moves = 0;
        System.out.println("empty list: size 0, capacity 0");
        for (int i = 1; i <= 100; i++) {
            int sizeBefore = list.size();
            if (sizeBefore == capacity) {
                int newCapacity = (capacity == 0) ? 10 : capacity + capacity / 2;
                moves += sizeBefore;
                System.out.println("add #" + i + ": capacity " + capacity + " -> "
                        + newCapacity + " (moved " + sizeBefore + ")");
                capacity = newCapacity;
            }
            list.add(i);
        }
        System.out.println("after 100 adds: size " + list.size() + ", capacity "
                + capacity + ", elements moved " + moves);
    }
}
```

```
int[] fixedFive holds 5 ints, always

empty list: size 0, capacity 0
add #1: capacity 0 -> 10 (moved 0)
add #11: capacity 10 -> 15 (moved 10)
add #16: capacity 15 -> 22 (moved 15)
add #23: capacity 22 -> 33 (moved 22)
add #34: capacity 33 -> 49 (moved 33)
add #50: capacity 49 -> 73 (moved 49)
add #74: capacity 73 -> 109 (moved 73)
after 100 adds: size 100, capacity 109, elements moved 202
```

Growing by a factor of 1.5 instead of 2 regrows more often (7 times instead of 8 for 100 elements here, but starting from 10) and wastes less space in the unused part of the block. The same argument as Equation (1.5), with ratio 1.5 instead of 2, shows that the total number of copies stays below $3n$: again a constant per `add`, on average. The program counted 202 copies for 100 elements.

An `ArrayList` can hold only objects, not the basic types, so a list of numbers must be declared `ArrayList<Integer>`, and every `int` put into it is automatically wrapped in an `Integer` object. That wrapping, called **boxing**, costs memory and time, which is why numerical work in Java prefers a plain `int[]`.

1.5.10 Python lists and the `array` module

Python's built-in `list` is a growable array of *references*, like `ArrayList`. A list of the numbers 10 to 50 is really a contiguous row of eight-byte references, each pointing to a separate integer object on the heap. **Figure 1.9** draws it, together with the alternative that the standard module `array` provides.

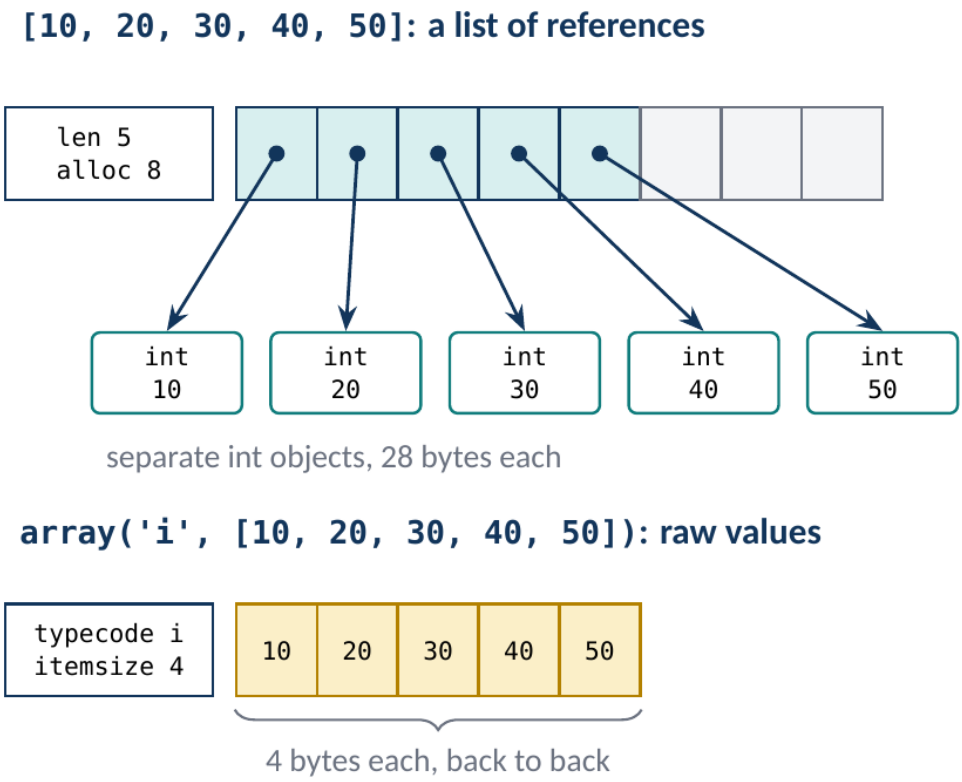


Figure 1.9 A Python `list` stores references to separate objects; an `array('i')` from the `array` module stores the raw 4-byte values themselves, back to back.

A list also over-allocates, keeping spare slots at the end so that most `append`s need no copying. **Listing 1.34** watches `sys.getsizeof` of a list as 100 elements are appended.

Listing 1.34 `list_growth.py`: a tuple cannot grow; a list grows in steps, leaving room for future appends.

```
# list_growth.py - a list over-allocates, so most appends need no copying
import sys

fixed_five = (1, 2, 3, 4, 5)             # a tuple cannot grow at all
print(f"the tuple fixed_five holds {len(fixed_five)} ints, always\n")

lst = []
last = sys.getsizeof(lst)
print(f"empty list: size 0, {last} bytes")
for i in range(1, 101):
    lst.append(i)
    now = sys.getsizeof(lst)
    if now != last:
        slots = (now - sys.getsizeof([])) // 8   # 8 bytes per reference
        print(f"append #{i}: {last} -> {now} bytes (room for {slots} references)")
        last = now
print(f"after 100 appends: size {len(lst)}, {sys.getsizeof(lst)} bytes")
```

```
the tuple fixed_five holds 5 ints, always

empty list: size 0, 56 bytes
append #1: 56 -> 88 bytes (room for 4 references)
append #5: 88 -> 120 bytes (room for 8 references)
append #9: 120 -> 184 bytes (room for 16 references)
append #17: 184 -> 248 bytes (room for 24 references)
append #25: 248 -> 312 bytes (room for 32 references)
append #33: 312 -> 376 bytes (room for 40 references)
append #41: 376 -> 472 bytes (room for 52 references)
append #53: 472 -> 568 bytes (room for 64 references)
append #65: 568 -> 664 bytes (room for 76 references)
append #77: 664 -> 792 bytes (room for 92 references)
append #93: 792 -> 920 bytes (room for 108 references)
after 100 appends: size 100, 920 bytes
```

CPython, the standard Python interpreter, grows a full list by about one eighth of its size plus a few slots, rounded to a multiple of four, so that the steps above run 4, 8, 16, 24, 32, 40, 52, 64, 76, 92, 108 [12]. The exact numbers are an implementation detail and may differ between Python versions; what matters is that each step is a constant fraction larger than the last, which again keeps the average cost of an `append` constant. A **tuple**, written with round brackets, is Python's fixed-size, unchangeable sequence.

When a program handles many numbers of one kind, a list of references is wasteful: each element costs an eight-byte reference plus a 28-byte integer object. The `array` **module** stores numbers the way C++ does, as raw values of a fixed type packed side by side [13]. `array('i', ...)` holds C `int`s of four bytes; `'d'` holds eight-byte `double`s. **Listing 1.35** compares the two for 1,000 numbers.

Listing 1.35 `compact_array.py`: 1,000 numbers in a list of references and in a compact `array('i')`.

```
# compact_array.py - a list holds references; array('i') holds raw 4-byte ints
import sys
from array import array

N = 1000
as_list = list(range(N))
as_array = array("i", range(N))

print(f"bytes per element      : {as_array.itemsize}")
print(f"array('i') element data: {as_array.itemsize * len(as_array)} bytes")
print(f"array object in total  : {sys.getsizeof(as_array)} bytes")
print(f"list object in total   : {sys.getsizeof(as_list)} bytes (references only)")
print(f"one int object         : {sys.getsizeof(999)} bytes,"
      f" and the list refers to {N} of them")
print(f"as_list[999] + as_array[999]: {as_list[999] + as_array[999]}")
```

```
bytes per element      : 4
array('i') element data: 4000 bytes
array object in total  : 4200 bytes
list object in total   : 8056 bytes (references only)
one int object         : 28 bytes, and the list refers to 1000 of them
as_list[999] + as_array[999]: 1998
```

The list alone takes 8,056 bytes, eight per reference plus a small header, before counting the integer objects it refers to, which bring the total to about 36,000 bytes; the `array` takes 4,200 bytes in all, of which 4,000 are the numbers themselves. For a thousand numbers the difference does not matter. For a hundred million, it is the difference between 400 megabytes and 3.6

gigabytes, which is why scientific Python uses compact arrays, most famously those of the NumPy library, built on the same idea.

Listing 1.36 and **Listing 1.37** are the C++ and Java counterparts. In C++ a raw array and a vector's block both hold the numbers directly, 4,000 bytes for 1,000 `int32_t`s, and the vector object itself is only 24 bytes: the three fields of **Figure 1.8**. In Java, an `int[]` holds the numbers directly, and an `Integer[]` holds references to separate objects, exactly like a Python list.

Listing 1.36 `compact_array.cpp`: 1,000 `int32_t`s in a raw array and in a vector.

```cpp
// compact_array.cpp - 1,000 ints stored back to back, 4 bytes each
#include <cstdint>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    const int N = 1000;
    int32_t plain[N];
    vector<int32_t> v(N);
    for (int i = 0; i < N; i++) {
        plain[i] = i;
        v[i] = i;
    }
    cout << "bytes per element      : " << sizeof(int32_t) << '\n';
    cout << "plain array, N elements: " << sizeof plain << " bytes\n";
    cout << "vector's element block : " << v.size() * sizeof(int32_t) << " bytes\n";
    cout << "vector object itself   : " << sizeof v
         << " bytes (pointers to the block)\n";
    cout << "plain[999] + v[999]    : " << plain[999] + v[999] << '\n';
    return 0;
}
```

```
bytes per element      : 4
plain array, N elements: 4000 bytes
vector's element block : 4000 bytes
vector object itself   : 24 bytes (pointers to the block)
plain[999] + v[999]    : 1998
```

Listing 1.37 `CompactArray.java`: `int[]` stores values, `Integer[]` stores references.

```java
// CompactArray.java - int[] stores values; Integer[] stores references to objects
public class CompactArray {
    public static void main(String[] args) {
        final int N = 1000;
        int[] plain = new int[N];
        Integer[] boxed = new Integer[N];
        for (int i = 0; i < N; i++) {
            plain[i] = i;
            boxed[i] = i;                    // each value becomes an Integer
        }
        System.out.println("bytes per element      : " + Integer.BYTES);
        System.out.println("int[] element data     : " + N * Integer.BYTES
                + " bytes");
        System.out.println("Integer[] holds        : " + N + " references + up to "
                + N + " separate objects");
        System.out.println("plain[999] + boxed[999]: " + (plain[999] + boxed[999]));
    }
}
```

```
bytes per element      : 4
int[] element data     : 4000 bytes
Integer[] holds        : 1000 references + up to 1000 separate objects
plain[999] + boxed[999]: 1998
```

1.5.11 The three languages side by side

Table 1.5 gathers the differences this section has met.

Table 1.5 Arrays and growable arrays in C++, Java and Python.

Question	C++	Java	Python
Fixed-size array	raw array `int a[8]`; `std::array<int, 8>`	`int[] a = new int[8]`	`tuple`; `array('i', ...)` has fixed item type
Growable array	`std::vector<int>`	`ArrayList<Integer>`	`list`
Where the elements are	in the array's own block	basic types: in the block; objects: references in the block	references in the block; `array`: raw values
Knows its length	`std::array`, `vector`: `size()`; raw array: no	`a.length`, `list.size()`	`len(a)`
Out-of-range index	undefined behavior; `at()` throws `out_of_range`	`ArrayIndexOutOfBoundsException`	`IndexError`
Negative index	undefined behavior	exception	counts from the end
Two-dimensional	contiguous, row-major	array of row arrays, may be jagged	list of row lists
Frees memory	automatically for arrays and containers; `delete` for `new`	garbage collector	garbage collector
Growth factor when full	2 (GCC), 1.5 (Microsoft)	1.5 (OpenJDK)	about 1.125 (CPython)

1.6 A First Taste of Complexity: $O(1)$ Access versus $O(n)$ Insertion

1.6.1 Counting steps instead of seconds

How fast is an operation on an array? We could time it with a stopwatch, but the answer would depend on the computer, the compiler, what else the machine was doing, and the phase of the moon. Computer scientists therefore ask a different question, one that has the same answer on every machine: **how does the number of basic steps grow as the amount of data grows?** A basic step is anything that takes a fixed, small amount of time regardless of the data: reading or writing one element, comparing two numbers, adding two numbers. We write n for the number of elements and count steps as a function of n [1].

1.6.2 Access: the same cost for every index

Reading `A[i]` is, by Equation (1.2), one multiplication, one addition and one memory read. That is the same handful of steps whether the array holds eight elements or eight million, and whether i is 0 or $n - 1$. The cost does not grow with n at all. We say that access takes **constant time**, and write it $O(1)$, read "order one" or "big-O of one". Writing an element, `A[i] = x`, is $O(1)$ for the same reason.

This is the array's great strength, and nearly every other structure in this book gives it up in exchange for something else. A linked list, for instance, must walk from its first element past i others to reach element i.

1.6.3 Insertion and deletion: the cost of keeping the row tight

The array's strength is also its weakness. Because the elements must stay side by side, with no gaps, putting a new element *between* two existing ones means making room first: every element from the insertion point to the end must move one place to the right. **Figure 1.10** shows 25 being inserted at index 2 of the array `10 20 30 40 50`.

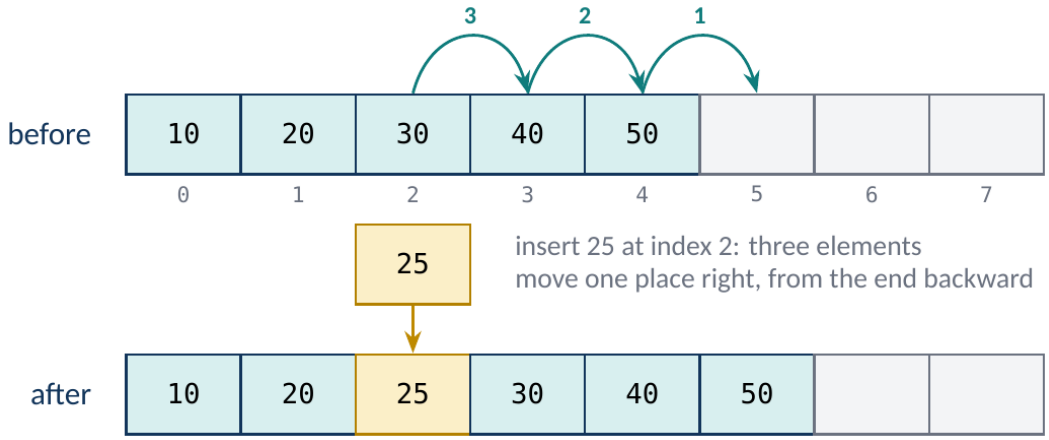


Figure 1.10 Inserting 25 at index 2: the elements at indexes 4, 3 and 2 move one place right, in that order, and 25 is written into the freed slot.

The moves must go from the end backward: 50 first into the empty slot 5, then 40 into slot 4, then 30 into slot 3. Moving from the front would overwrite 40 with 30 before 40 had been saved. **Pseudocode 1.1** states the procedure precisely, in the pseudocode style used throughout this book: numbered lines, ← for assignment, and the array written $A[0..n-1]$ for the elements with indexes 0 to $n-1$.

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
```

Deletion is the mirror image, given in **Pseudocode 1.2**: every element after the removed one moves one place *left*, from the front forward, to close the gap.

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
```

How many moves does `ARRAY-INSERT` make? Line 2 runs once for each i from n down to $pos + 1$, that is, for

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

values of i. Inserting at the end, $pos = n$, makes no moves at all. Inserting at the front, $pos = 0$, moves every one of the n elements. And if the insertion point is equally likely to be any of the $n + 1$ positions $0,1, \ldots, n$, the *average* number of moves is

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

The middle step uses the sum $0 + 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$. To see why that sum is right, write it twice, once forward and once backward, and add the two rows column by column: each column adds to n, and there are $n + 1$ columns, so twice the sum is $n(n + 1)$. Equation (1.7) says that an insertion into an array of a million elements moves, on average, half a million of them. Deletion has exactly the same count, $n - 1 - pos$ moves, averaging $(n - 1)/2$.

Listing 1.38 implements both procedures, applies them to the array of **Figure 1.10**, and then counts the moves needed to insert one element at the front, in the middle and at the end of arrays of 10 to 10,000 elements.

Listing 1.38 `insert_shift.cpp`: insertion and deletion by shifting, with the number of moves counted.

```
// insert_shift.cpp - O(1) access versus O(n) insertion, counted
#include <iomanip>
#include <iostream>
#include <vector>
using namespace std;

// 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;
}

// Removes the element at index pos; returns the number of elements moved left.
int eraseAt(vector<int>& a, int& n, int pos) {
    int moves = 0;
    for (int i = pos; i < n - 1; i++) {
        a[i] = a[i + 1];
        moves++;
    }
    n--;
    return moves;
}

void show(const vector<int>& a, int n) {
    cout << '[';
    for (int i = 0; i < n; i++) cout << (i ? " " : "") << a[i];
    cout << "]";
}
```

Listing 1.38 (Continued) *insert_shift.cpp: insertion and deletion by shifting, with the number of moves counted.*

```
int main() {
    vector<int> a(8);
    int n = 0;
    for (int x : {10, 20, 30, 40, 50}) a[n++] = x;
    cout << "start            "; show(a, n); cout << '\n';
    cout << "a[3] is read in one step: " << a[3] << '\n';
    int m = insertAt(a, n, 2, 25);
    cout << "insert 25 at 2   "; show(a, n); cout << "  moved " << m << '\n';
    m = insertAt(a, n, 0, 5);
    cout << "insert 5 at 0    "; show(a, n); cout << "  moved " << m << '\n';
    m = insertAt(a, n, n, 60);
    cout << "insert 60 at end "; show(a, n); cout << "  moved " << m << '\n';
    m = eraseAt(a, n, 1);
    cout << "erase index 1    "; show(a, n); cout << "  moved " << m << '\n';

    cout << "\nmoves needed to insert one element into an array of n:\n";
    cout << setw(6) << "n" << setw(8) << "front" << setw(8) << "middle"
         << setw(6) << "end" << '\n';
    for (int size : {10, 100, 1000, 10000}) {
        int results[3];
        int positions[3] = {0, size / 2, size};
        for (int k = 0; k < 3; k++) {
            vector<int> b(size + 1);
            int len = size;
            results[k] = insertAt(b, len, positions[k], -1);
        }
        cout << setw(6) << size << setw(8) << results[0] << setw(8) << results[1]
             << setw(6) << results[2] << '\n';
    }
    return 0;
}
```

```
start            [10 20 30 40 50]
a[3] is read in one step: 40
insert 25 at 2   [10 20 25 30 40 50]  moved 3
insert 5 at 0    [5 10 20 25 30 40 50]  moved 6
insert 60 at end [5 10 20 25 30 40 50 60]  moved 0
erase index 1    [5 20 25 30 40 50 60]  moved 6

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
```

The table is Equation (1.6) in action. Reading `a[3]` took one step, whatever the size. Inserting at the front took exactly n moves and in the middle exactly $n/2$: multiply the size by ten and the work is multiplied by ten. Only insertion at the very end is free, which is precisely why `push_back`, `add` and `append` are the operations that growable arrays make cheap.

Listing 1.39 and **Listing 1.40** print exactly the same output. Note in the Python version that the loop `for i in range(n, pos, -1)` counts *down* from n to $pos + 1$: the third argument of `range` is the step, and the second is the first value *not* reached, exactly like the bound in line 1 of **Pseudocode 1.1**.

Listing 1.39 `InsertShift.java`: insertion and deletion by shifting, in Java.

```java
// InsertShift.java - O(1) access versus O(n) insertion, counted
public class InsertShift {
    static int n = 0;                          // how many slots are in use

    // Inserts x at index pos; returns the number of elements moved right.
    static int insertAt(int[] a, 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;
    }

    // Removes the element at index pos; returns the number of elements moved left.
    static int eraseAt(int[] a, int pos) {
        int moves = 0;
        for (int i = pos; i < n - 1; i++) {
            a[i] = a[i + 1];
            moves++;
        }
        n--;
        return moves;
    }

    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < n; i++) sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    public static void main(String[] args) {
        int[] a = new int[8];
        for (int x : new int[] {10, 20, 30, 40, 50}) a[n++] = x;
        System.out.println("start            " + show(a));
        System.out.println("a[3] is read in one step: " + a[3]);
        int m = insertAt(a, 2, 25);
        System.out.println("insert 25 at 2   " + show(a) + "  moved " + m);
        m = insertAt(a, 0, 5);
        System.out.println("insert 5 at 0    " + show(a) + "  moved " + m);
        m = insertAt(a, n, 60);
        System.out.println("insert 60 at end " + show(a) + "  moved " + m);
        m = eraseAt(a, 1);
        System.out.println("erase index 1    " + show(a) + "  moved " + m);
```

Listing 1.39 (Continued) *InsertShift.java: insertion and deletion by shifting, in Java.*

```java

        System.out.println(
                "\nmoves needed to insert one element into an array of n:");
        System.out.printf("%6s%8s%8s%6s%n", "n", "front", "middle", "end");
        for (int size : new int[] {10, 100, 1000, 10000}) {
            int[] positions = {0, size / 2, size};
            int[] results = new int[3];
            for (int k = 0; k < 3; k++) {
                int[] b = new int[size + 1];
                n = size;
                results[k] = insertAt(b, positions[k], -1);
            }
            System.out.printf("%6d%8d%8d%6d%n",
                    size, results[0], results[1], results[2]);
        }
    }
}
```

Listing 1.40 `insert_shift.py`: insertion and deletion by shifting, in Python.

```
# insert_shift.py - O(1) access versus O(n) insertion, counted
def insert_at(a, n, pos, x):
    """Insert x at index pos of a[0..n-1]; return the number of elements moved."""
    moves = 0
    for i in range(n, pos, -1):          # walk from the end back to pos
        a[i] = a[i - 1]
        moves += 1
    a[pos] = x
    return moves


def erase_at(a, n, pos):
    """Remove the element at index pos; return the number of elements moved left."""
    moves = 0
    for i in range(pos, n - 1):
        a[i] = a[i + 1]
        moves += 1
    return moves


def show(a, n):
    return "[" + " ".join(str(a[i]) for i in range(n)) + "]"


a = [0] * 8
n = 0
for x in (10, 20, 30, 40, 50):
    a[n] = x
    n += 1
print(f"start            {show(a, n)}")
print(f"a[3] is read in one step: {a[3]}")
m = insert_at(a, n, 2, 25)
n += 1
print(f"insert 25 at 2   {show(a, n)}  moved {m}")
m = insert_at(a, n, 0, 5)
n += 1
print(f"insert 5 at 0    {show(a, n)}  moved {m}")
m = insert_at(a, n, n, 60)
n += 1
print(f"insert 60 at end {show(a, n)}  moved {m}")
m = erase_at(a, n, 1)
n -= 1
print(f"erase index 1    {show(a, n)}  moved {m}")
```

Listing 1.40 (Continued) *insert_shift.py: insertion and deletion by shifting, in Python.*

```
print("\nmoves needed to insert one element into an array of n:")
print(f"{'n':>6}{'front':>8}{'middle':>8}{'end':>6}")
for size in (10, 100, 1000, 10000):
    results = []
    for pos in (0, size // 2, size):
        b = [0] * (size + 1)
        results.append(insert_at(b, size, pos, -1))
    print(f"{size:>6}{results[0]:>8}{results[1]:>8}{results[2]:>6}")
```

The standard containers do exactly the same work behind a friendlier interface: `v.insert(v.begin() + pos, x)` in C++, `list.add(pos, x)` in Java and `lst.insert(pos, x)` in Python all shift the elements after `pos`. A single line of code can therefore hide a million moves, and a programmer who inserts at the front of a large list inside a loop can write a program that is correct and hopelessly slow. Knowing what the line costs is the whole point of this course.

1.6.4 Big-O, informally

We need a short way to say "the number of steps grows in proportion to n" without fussing over whether it is exactly n, or $n/2$, or $n/2 + 3$. That short way is **big-O notation**. Chapter 2 defines it with full rigor; here is enough to use it.

> **DEFINITION.** Let $f(n)$ and $g(n)$ be the numbers of steps of two computations on inputs of size n. We write $f(n) = O(g(n))$ if there are positive constants c and n_0 such that Equation (1.8) holds. In words: from some size onward, f grows no faster than a constant multiple of g [1].

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0 \tag{1.8}$$

Equation (1.8) deliberately ignores two things. It ignores **constant factors**: $n/2$ moves and n moves are both $O(n)$, because $n/2 \leq 1 \cdot n$. And it ignores **small inputs**: only the behavior from n_0 onward counts. Both omissions are exactly what we want. Constant factors depend on the machine and the compiler, which we agreed to leave out, and for small inputs every method is fast. What remains is the *shape* of the growth, and the shape decides everything when n is large, as **Table 1.1** showed.

Two shapes appear in this chapter:

- $O(1)$, **constant**: the steps do not grow with n at all. Array access, and insertion or deletion at the end.
- $O(n)$, **linear**: the steps grow in proportion to n. Double n and the work doubles. A traversal, a linear search, and insertion or deletion at the front or in the middle.

As an example of the definition at work, take the average insertion cost $f(n) = n/2$ from Equation (1.7) and $g(n) = n$. Choosing $c = 1$ and $n_0 = 1$, we have $n/2 \leq 1 \cdot n$ for every $n \geq 1$, so $n/2 = O(n)$. For the traversal in `sumOf`, the loop body runs n times and each run does a comparison, an addition and an increment, say $3n + 2$ steps counting the start and the return; with $c = 5$ and $n_0 = 1$, $3n + 2 \leq 5n$ for all $n \geq 1$, so the sum is $O(n)$ too.

Table 1.6 summarizes the cost of the operations on an array of n elements. It is the first of many such tables in this book; every data structure is, in the end, a choice of which rows of this table to make cheap.

Table 1.6 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)

1.6.5 A glimpse of amortized cost

The last row of **Table 1.6** deserves one more paragraph, because it contains an idea we will meet again. A single `push_back` on a full vector costs $O(n)$: it copies all n elements. Yet Equation (1.5) showed that n `push_back`s starting from empty cost fewer than $2n$ copies *in total*. Spreading that total evenly, each `push_back` costs fewer than two copies plus its own write: $O(1)$ per operation *on average over any sequence*. This is called **amortized** constant time, from the accountant's word for spreading a large cost over many payments. It is not the same as "usually fast": it is a guarantee about every long sequence of operations, however unlucky. Chapter 5 uses the same reasoning for stacks and queues built on growable arrays.

1.7 Setting Up VS Code for C++, Java and Python on Windows, macOS and Linux

Everything in this book runs on an ordinary laptop, and nothing has to be bought. This section installs, once and carefully, the complete workbench: an editor, a C++ compiler and debugger, a Java Development Kit and Python, and then proves that each of them works by running the same first program in all three languages. Later chapters assume that it has been done.

1.7.1 Three words first

A **terminal** is a window in which you type commands and read their answers. On Windows it is **PowerShell** or the older **Command Prompt**; on macOS and Linux it is called **Terminal**; and VS Code has one built in, opened with **Terminal → New Terminal**. In this book a line to type into a terminal is printed in a monospaced font, and you press Enter at its end.

A **package manager** installs software by command instead of by clicking through an installer: `winget` on Windows 10 and 11, **Homebrew** (the `brew` command, from `https://brew.sh`) on macOS, and `apt` on Ubuntu and Debian. Each step below gives the package-manager command and, where it exists, the graphical alternative; either is fine.

The **PATH** is the list of folders in which a terminal looks for programs. When you type `g++`, the terminal searches each folder on the PATH for a program called `g++`. "Command not found" or "is not recognized" almost always means that the program is installed but its folder is not on the PATH, or that the terminal was opened before the folder was added. Opening a **new** terminal is the first cure to try.

1.7.2 Visual Studio Code

Visual Studio Code, VS Code for short, is a free editor from Microsoft that looks and works the same on all three systems, and becomes a complete environment for any language through **extensions** [14]. Its release in September 2026 is version 1.138; it updates itself every month, so any recent version will do [15]. **Table 1.7** gives the installation.

Table 1.7 Installing VS Code on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	Installer from `https://code.visualstudio.com`; tick **Add to PATH**. Or `winget install Microsoft.VisualStudioCode`	Download, unzip and drag *Visual Studio Code* into **Applications**. Or `brew install --cask visual-studio-code`	The `.deb` from the website, then `sudo apt install ./code_*.deb`. Or `sudo snap install code --classic`
Make `code` work in a terminal	Done by the installer	In VS Code: **Cmd+Shift+P**, then *Shell Command: Install 'code' command in PATH*	Done by the package
Check	`code --version`	`code --version`	`code --version`

Make one folder for this book, for example `DS` in your Documents folder, with one subfolder per chapter, `Chapter01`, `Chapter02` and so on, as the accompanying code is organized. Always open the **folder** in VS Code (**File → Open Folder…**), never a single file: VS Code's terminal then opens inside that folder, so that commands such as `./io_sum < week_loans.txt` find their files.

1.7.3 A C++ compiler and debugger

C++ needs a **compiler**, which turns `.cpp` files into programs, and a **debugger**, which lets you stop a running program and look inside it. We use **GCC** (the commands `g++` and `gdb`) on Windows and Linux, and **Clang** (`clang++` and `lldb`) on macOS. Both understand standard C++17, which is all this book uses. Any recent release will do; in September 2026 the newest are GCC 16.2, released on 7 August 2026 [16], and Clang 23.1, released on 26 August 2026 [17]. Recent compilers use a newer C++ standard unless told otherwise, which is why every command in this book passes `-std=c++17` explicitly. **Table 1.8** gathers the procedure; the three short step-by-step lists after it explain each row.

Table 1.8 Installing and using a C++ compiler on each operating system. Fedora uses `sudo dnf install gcc-c++ gdb` and Arch `sudo pacman -S gcc gdb` in step 1.

Step	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
1. Install	MSYS2 from `https://www.msys2.org`; in the **MSYS2 UCRT64** terminal: `pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain`	`xcode-select --install`	`sudo apt update && sudo apt install build-essential gdb`
2. Put it on the PATH	Add `C:\msys64\ucrt64\bin` to the user **Path**; open a new terminal	Nothing to do	Nothing to do
3. Check	`g++ --version` and `gdb --version`	`clang++ --version`	`g++ --version` and `gdb --version`
4. Compile	`g++ -std=c++17 -O2 -Wall -Wextra -o hello hello.cpp`	`clang++ -std=c++17 -O2 -Wall -Wextra -o hello hello.cpp`	as Windows
5. Run	`.\hello.exe`	`./hello`	`./hello`

Windows, step by step. Windows has no C++ compiler of its own that works from a terminal, so we take GCC from **MSYS2**, a small, well-maintained collection of the tools Linux programmers use, which is also the route that the VS Code documentation recommends [14], [18].

1. Download the installer from `https://www.msys2.org` and run it, accepting the folder `C:\msys64`.
2. From the Start menu open **MSYS2 UCRT64**. The name matters: UCRT64 is the modern variant, and packages installed from another MSYS2 terminal will not match. Type `pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain`, press Enter to accept all members of the group, and answer `Y`.
3. Press the Windows key, type *environment variables*, choose **Edit environment variables for your account**, select **Path**, click **Edit → New**, type `C:\msys64\ucrt64\bin`, and click **OK** in every window.
4. Open a *new* PowerShell window and type `g++ --version` and `gdb --version`. Each should print a version line.

macOS, step by step. Apple supplies Clang with its *Command Line Tools*; it only needs to be asked. Open **Terminal** (Cmd+Space, type *Terminal*, Enter), type `xcode-select --install`, click **Install** in the window that appears, and wait. Then `clang++ --version` prints a version line. The command `g++` also works on a Mac, but it quietly runs Clang; every program in this book compiles with both. The debugger on macOS is `lldb`, which VS Code uses automatically.

Linux, step by step. Open a terminal (Ctrl+Alt+T on Ubuntu) and type `sudo apt update && sudo apt install build-essential gdb`, enter your password, and confirm. `build-essential` brings `gcc`, `g++` and `make`.

1.7.4 Java: JDK 25

Java programs are compiled with a **JDK**, a Java Development Kit, which contains the compiler `javac` and the launcher `java`. Install **JDK 25**, the long-term-support release of September 2025, which receives updates for years; JDK 27, released on 15 September 2026, is the newest feature release, supported only until the next one appears in six months [19], [20]. The **Eclipse Temurin** builds from `https://adoptium.net` are free and identical on all three systems [21]. The book's Java programs need Java 17 or newer and were compiled here with JDK 21.

Table 1.9 Installing JDK 25 on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	`winget install EclipseAdoptium.Temurin.25.JDK`, or the `.msi` from adoptium.net with **Set JAVA_HOME** ticked	`brew install --cask temurin@25`, or the `.pkg` from adoptium.net	`sudo apt install openjdk-25-jdk` (Ubuntu 26.04), or Temurin from the Adoptium apt repository
Check	`java -version` and `javac -version`	same	same

Both checks must answer, with the same version number. If `java` answers and `javac` does not, only a *runtime* was installed, which can run Java programs but not compile them; install the JDK.

1.7.5 Python 3.14

Install **Python 3.14**, the current series; its latest release in September 2026 is 3.14.7, of 5 August 2026 [22]. The book's Python programs need Python 3.10 or newer (for the `match` statement of **Listing 1.9**) and were run here with Python 3.11.

Table 1.10 Installing Python on each operating system.

	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
Install	Installer from `https://www.python.org`; on its first screen tick **Add python.exe to PATH**. Or `winget install Python.Python.3.14`	Installer from python.org, or `brew install python@3.14`	The system `python3`: `sudo apt install python3`; Ubuntu 26.04 ships Python 3.14
Run a program	`python hello.py`	`python3 hello.py`	`python3 hello.py`
Check	`python --version`	`python3 --version`	`python3 --version`

On Windows the command is `python`; on macOS and Linux it is `python3`, because on older systems `python` meant the long-retired Python 2. Where this book writes `python3`, Windows users type `python`.

1.7.6 The VS Code extensions

Open the Extensions view (the square icon in the left bar, or **Ctrl+Shift+X**, **Cmd+Shift+X** on a Mac), search for each name in **Table 1.11**, and press **Install**, or type the three commands below the table into any terminal.

Table 1.11 The VS Code extensions used in this book.

Extension	Publisher	Identifier	Gives you
C/C++ Extension Pack	Microsoft	`ms-vscode.cpptools-extension-pack`	C++ highlighting, error squiggles, completion, the GDB/LLDB debugger
Extension Pack for Java	Microsoft	`vscjava.vscode-java-pack`	Java editing, running, debugging and testing
Python	Microsoft	`ms-python.python`	Python editing, running, debugging and virtual environments

```
code --install-extension ms-vscode.cpptools-extension-pack
code --install-extension vscjava.vscode-java-pack
code --install-extension ms-python.python
```

1.7.7 Checking the whole workbench

Open a *new* terminal, so that it sees the updated PATH, and type each command of **Table 1.12**. On Windows, type `python` where the table says `python3`.

Table 1.12 The verification commands, what each should print, and the usual fix when it does not.

Command	A good answer begins with	If it fails
`code --version`	`1.`	Reinstall VS Code with **Add to PATH**, or run *Shell Command: Install 'code' command in PATH*
`g++ --version`	`g++` (on a Mac: `Apple clang`)	Windows: add `C:\msys64\ucrt64\bin` to Path and open a new terminal
`gdb --version`	`GNU gdb` (on a Mac use `lldb --version`)	Windows: repeat the `pacman` step in **MSYS2 UCRT64**
`javac -version`	`javac 25`	Only a runtime is installed, or an older JDK is earlier on the PATH
`java -version`	`openjdk version "25`	Install the JDK; check `JAVA_HOME`
`python3 --version`	`Python 3.14`	Windows: rerun the installer and tick **Add python.exe to PATH**

On the Linux machine on which this chapter's programs were run, the checks printed the following. The machine is a little behind the recommended versions, with GCC 13, JDK 21 and Python 3.11, which shows that the programs of this book run on these too.

```
$ g++ --version
g++ (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0
$ gdb --version
GNU gdb (Ubuntu 15.1-1ubuntu1~24.04.1) 15.1
$ javac -version
javac 21.0.10
$ java -version
openjdk version "21.0.10" 2026-01-20
$ python3 --version
Python 3.11.15
```

1.7.8 Your first program, in three languages

Create the file `hello.cpp` in your `Chapter01` folder, type in **Listing 1.41**, and save it.

Listing 1.41 `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;
}
```

Open the terminal in VS Code with **Terminal → New Terminal** and compile it with the line from step 4 of **Table 1.8**:

```
g++ -std=c++17 -O2 -Wall -Wextra -o hello hello.cpp
```

Each part of the line has a job. `-std=c++17` selects the C++17 standard; `-O2` asks the compiler to optimize, so that any timing measures the algorithm rather than a lazy translation; `-Wall -Wextra` switch on warnings, which catch many mistakes before they become bugs; `-o hello` names the resulting program. If the compiler prints nothing, it succeeded: silence is good news. Run the program (`./hello`, or `.\hello.exe` on Windows) and type a name:

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

The last line is a small proof: `201703` is the value that the C++ standard gives the predefined name `__cplusplus` under C++17, so you now know for certain which standard your programs are compiled against.

Now the Java version, **Listing 1.42**. Save it as `Hello.java`: in Java, a public class must live in a file with exactly its name, capital letter included.

Listing 1.42 `Hello.java`: the first program in Java.

```java
// Hello.java - the first program of the book
import java.util.Scanner;

public class Hello {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("What is your name? ");
        String name = in.hasNextLine() ? in.nextLine() : "";
        if (name.isEmpty()) {
            name = "friend";                  // no input: greet a friend
        }
        System.out.println("Hello, " + name + "! Welcome to Data Structure.");
        System.out.println("Language version: Java " + Runtime.version().feature());
    }
}
```

Compile and run it in two steps; since Java 11, `java Hello.java` also works in one step, because the launcher can compile a single-file program itself:

```
javac Hello.java
java Hello
```

```
What is your name? Sari
Hello, Sari! Welcome to Data Structure.
Language version: Java 21
```

With JDK 25 the last line says `Java 25`. Finally the Python version, **Listing 1.43**, saved as `hello.py` and run with `python3 hello.py` (`python hello.py` on Windows):

Listing 1.43 `hello.py`: the first program in Python.

```python
# hello.py - the first program of the book
import sys

try:
    name = input("What is your name? ")
except EOFError:
    name = ""
if not name:
    name = "friend"                       # no input: greet a friend
print(f"Hello, {name}! Welcome to Data Structure.")
print(f"Language version: Python {sys.version_info.major}.{sys.version_info.minor}")
```

```
What is your name? Sari
Hello, Sari! Welcome to Data Structure.
Language version: Python 3.11
```

With Python 3.14 the last line says `Python 3.14`. All three programs handle the case where there is no input at all, for example when they are run with an empty file redirected into them: they greet "friend".

1.7.9 Running from VS Code, feeding input, and debugging

Running from the editor. With the extensions of **Table 1.11** installed, a Java or Python file shows a **Run** triangle at the top right of the editor, and a C++ file shows **Run C/C++ File**, which compiles the file with the compiler it finds and runs it in the terminal. The first time, VS Code asks which compiler to use and writes its choice into a file `.vscode/tasks.json` in your folder. For the exact flags of this book, add `"-std=c++17", "-Wall", "-Wextra"` to the `args` list in that file.

Feeding input from a file. Programs such as `io_sum` read standard input. Rather than typing the numbers every time, redirect a file, as Section 1.4.3 did: `./io_sum < week_loans.txt`, `java IoSum < week_loans.txt` or `python3 io_sum.py < week_loans.txt` on macOS, Linux and the Windows Command Prompt, and `Get-Content week_loans.txt | .\io_sum.exe` in PowerShell, which does not understand `<`.

Debugging: watching the program think. When a program gives a wrong answer, the fastest way to find out why is to *watch* it. Click in the margin to the left of a line number: a red dot appears, a **breakpoint**. Press **F5** and choose the debugger for your language (for C++, *C++ (GDB/LLDB)* and your compiler). The program runs until it reaches the red dot and stops; the **Variables** panel then shows the value of every variable at that moment, and you can hover over any name in the code to see its value. **F10** runs one line, **F11** steps into a function call, **Shift+F11** steps out again, and **F5** runs on to the next breakpoint. Put a breakpoint inside the loop of `reverseInPlace` in **Listing 1.22** and watch `i` and `j` walk toward each other: it is the best way to understand a loop that you did not write.

> **WHEN SOMETHING GOES WRONG.** *"g++ is not recognized"* (Windows): the Path entry is missing, or the terminal was opened before it was added; open a new terminal. *"fatal error: iostream: No such file or directory"*: the compiler was installed from the wrong MSYS2 terminal; repeat the `pacman` step in **MSYS2 UCRT64**. *The program window flashes and closes*: run it from a terminal, not by double-clicking it. *Error squiggles everywhere in VS Code, but the program compiles*: open the Command Palette (**Ctrl+Shift+P**), run **C/C++: Select IntelliSense Configuration**, and choose the compiler you installed. *`javac` is not found but `java` is*: you installed a runtime; install the JDK. *`error: class Hello is public, should be declared in a file named Hello.java`*: rename the file to match the class exactly. *Python prints `SyntaxError` at a `match` line*: the Python is older than 3.10; install 3.14.

With the workbench standing, every listing in the rest of this book is one copy, one compile and one run away from being yours.

1.8 Profitina in Practice

Profitina, the AI visibility platform at app.profitina.com, measures how ready a brand's website is to be cited by AI answer engines, and then helps fix what it finds. Under its dashboards, a good deal of the data it handles has exactly the shapes of this chapter, and the costs of **Table 1.6** are the costs it lives with.

A fixed row of scores for every page. A scan examines each page of a site against a fixed list of signals, such as how easily the content can be extracted, whether it carries structured data, and how technically sound the page is, and scores each one. When the list of signals is fixed, the natural home for one page's scores is a fixed-size array with one slot per signal: slot k always means signal k, reading any score is $O(1)$ by Equation (1.2), and combining the scores into one number is a single $O(n)$ traversal like `sumOf` in **Listing 1.22**. A site's pages then form a table, one row per page and one column per signal, exactly the shape of **Listing 1.28**; a column total says how the whole site does on one signal, and a row total how one page does on all of them.

A history that only grows at the end. The historical analytics dashboard shows each brand's score week by week, and the scheduler adds a new point after every automatic re-scan. A series that only ever grows at its end is the ideal case for a growable array: appending is amortized $O(1)$ (Equation (1.5)), the score of week i is read in $O(1)$, and the CSV export is one traversal. Nothing is ever inserted in the middle, so the $O(n)$ row of **Table 1.6** is never paid. Profitina's back end is written in Python with FastAPI, so the everyday container is Python's `list` of Section 1.5.10.

Where the middle is expensive. The community feed and the notifications gathered on the dashboard are shown newest first. If new items were inserted at index 0 of an array, every insertion would shift every older item, n moves each time by Equation (1.6). The standard remedy is to append at the end, which is cheap, and to *read* the array backward when displaying it; Chapter 5's queues and Chapter 14's priority queues are the tools for the cases where that is not enough. The PROFIT leaderboard raises the same question in another form: keeping a ranked array exactly sorted while scores change means moving entries up and down it, and Chapters 3 and 14 study better ways.

Honest about limits. A fixed-size array fails loudly or silently when it overflows, depending on the language, as Section 1.5.4 showed. The Profitina house rule is to fail loudly: when the local AI model that powers the Content Engine is not available, the application says so rather than silently serving a template as if it came from the model. The C++ habit of `at()` over unchecked `[]`, and Java's and Python's refusal to read past an array's end, are the same principle at the smallest scale.

1.9 Common Mistakes, and How to Avoid Them

Looping one step too far. `for (int i = 0; i <= n; i++)` visits the non-existent `a[n]`. In C++ this is undefined behavior that may go unnoticed; in Java and Python it throws. *Avoid it:* the condition is always `i < n`. Better still, use the range-based `for` in C++, the enhanced `for` in Java, or `for x in a` in Python when the index is not needed.

Starting at 1. Writing `for (int i = 1; i <= n; i++)` because "the first book is book one" skips `a[0]` and reads `a[n]`. *Avoid it:* indexes are distances from the start, Equation (1.2); the first is 0.

Using an uninitialized array. `int a[8];` inside a function contains leftover bytes, not zeros. *Avoid it:* write `int a[8] = {0};`, or `std::array<int, 8> a{};`, or fill it before reading.

Computing the length inside a function with `sizeof`. In a function, an array parameter is a pointer, and `sizeof a / sizeof a[0]` gives $8/4 = 2$ on a 64-bit machine, whatever the array's real length. *Avoid it:* pass the length as a parameter, or use `std::array` or `std::vector`, which know their `size()`.

Losing a value while swapping. `a[i] = a[j]; a[j] = a[i];` leaves both slots holding the old `a[j]`. *Avoid it:* use a temporary, `int t = a[i]; a[i] = a[j]; a[j] = t;`, or `std::swap(a[i], a[j])`, or Python's `a[i], a[j] = a[j], a[i]`.

Shifting in the wrong direction. Making room for an insertion by moving elements from the *front* toward the back overwrites each element before it has been moved. *Avoid it:* insertion moves from the end backward, deletion from the gap forward, as in **Pseudocode 1.1** and **Pseudocode 1.2**.

Mixing up rows and columns. `m[c][r]` instead of `m[r][c]` reads the wrong element, or runs out of bounds when the table is not square. *Avoid it:* name the indexes `r` and `c` (or `s` and `p`), never `i` and `j`, in two-dimensional code, and keep the row first.

Assuming a Java or Python "2-D array" is one block. A `Book[][]` or a list of lists is an array of references to rows; the rows may have different lengths and may even be shared. *Avoid it:* check `a[r].length` or `len(a[r])` for each row when rows may differ, and never build a Python table with `[[0] * c] * r`; use `[[0] * c for _ in range(r)]` (**Listing 1.31**).

Copying a reference and calling it a copy. In Java, `int[] b = a;` and in Python `b = a` make a second *name* for the same array; changing `b[0]` changes `a[0]`. In C++, `int b[8] = a;` does not even compile. *Avoid it:* copy with `Arrays.copyOf(a, a.length)` in Java, `a.copy()` or `list(a)` in Python, and `std::array` or `std::vector` assignment in C++.

Inserting at the front of a large array in a loop. Each insertion shifts everything, so n insertions cost about $n^2/2$ moves. *Avoid it:* append at the end and reverse once at the finish, or use a structure built for insertion at both ends, such as the queues of Chapter 5.

Integer overflow and integer division. Summing a million `int`s of about 5,000 each overflows an `int`, and `sum / n` with two integers throws the fraction away. *Avoid it:* accumulate in `long` or `long long`, and divide as `static_cast<double>(sum) / n`.

Forgetting `delete`, or deleting twice. Every `new` needs exactly one `delete`. *Avoid it:* in this book, prefer `std::vector` and other containers, which free their memory themselves; Chapter 6 treats `new` and `delete` properly.

1.10 Chapter Summary

- A **data structure** is a way of organizing data in memory, together with its operations, usually **insert**, **delete**, **search**, **traverse** and **sort**. Its *interface* is what it promises; its *implementation* is how. The right choice can change the number of steps from n to $\log_2 n$ or to 1 (**Table 1.1**), which no faster computer can match.
- **Memory** is a row of bytes, each with an **address**; k bits have 2^k states (Equation (1.1)), so a byte holds 0 to 255. A **type** decides how many bytes a value takes and how they are read. Most machines are **little-endian**. The compiler may insert **padding** to align values.
- Local variables live in the function's frame on the **stack**, released automatically on return; memory from `new` lives on the **heap** until `delete`. Java and Python put objects on the heap and free them with a **garbage collector**.
- The **C++ refresher** covered the program skeleton, types (**Table 1.4**), `cin` and `cout`, `if`, `switch`, `while`, `for` and the range-based `for`, functions with parameters passed by value, structs, and the operators `&` (address of) and `*` (follow a pointer). Java and Python use references instead of visible addresses.

- An **array** is a contiguous row of n elements of one type, indexed 0 to $n-1$. Element i is at $b + i \cdot s$ (Equation (1.2)), which makes **access** $O(1)$. `a[i]` is `*(a + i)`. C++ does not check bounds; `at()` does. Java and Python always check, and Python accepts negative indexes.
- A **two-dimensional array** in C++ is stored **row-major**: element $[r][c]$ is at $b + (r \cdot C + c) \cdot s$ (Equation (1.3)); column-major order uses Equation (1.4). Java and Python build tables as arrays of row arrays.
- `std::array` is a fixed-size array that knows its size; `std::vector`, `ArrayList` and Python's `list` grow by allocating a larger block and copying. Growing by a constant factor keeps the total copies below a constant times n (Equation (1.5)), so appending is **amortized** $O(1)$. Python's `array` module stores raw values compactly.
- **Insertion and deletion** in the middle shift $n - pos$ elements (Equation (1.6)), $n/2$ on average (Equation (1.7)): $O(n)$. **Big-O** (Equation (1.8)) keeps the shape of growth and drops constant factors; **Table 1.6** summarizes the array's costs.
- The **workbench** is VS Code with the C/C++, Java and Python extensions, GCC through MSYS2 on Windows, Clang through the Command Line Tools on macOS, `build-essential` on Linux, JDK 25 and Python 3.14, checked with the commands of **Table 1.12**.

1.11 Exercises

Exercise 1.1. (a) Name the five operations by which data structures are usually judged. (b) For the contact list of a mobile phone, give one concrete example of each operation. (c) "Find a contact by name" can be offered by two different implementations. Describe two, in one sentence each, and say which part of the definition of a data structure they share and which part they do not.

Exercise 1.2. (a) How many different values can 16 bits hold? And 32? (b) The `int` value `0x12345678` is stored at address 2000. List the byte stored at each of the addresses 2000 to 2003 on a little-endian machine and on a big-endian machine. (c) Write the byte 11111111 in decimal and in hexadecimal.

Exercise 1.3. An array `double prices[50]` begins at address 5000, and a `double` takes 8 bytes. (a) At which address does `prices[17]` begin? (b) Which element begins at address 5320? (c) Does any element begin at address 5324? Explain. (d) What is the address of the byte just after the array?

Exercise 1.4. The array `int m[6][9]` begins at address 2000, with 4-byte `int`s. (a) Compute the address of `m[4][7]` in row-major order, as C++ stores it. (b) Compute the address the same element would have in column-major order. (c) In row-major order, which element begins at address $2000 + 4 \cdot 50$?

Exercise 1.5. Derive the row-major address formula for a three-dimensional array `A[D1][D2][D3]` with base address b and element size s, by the same reasoning as Equation (1.3). Check it with a program on `int a[2][3][4]`, and give the offset of `a[1][2][3]` from `a[0][0][0]`.

Exercise 1.6. Using the rules of **Listing 1.7**, compute by hand the fee for (a) a lecturer who kept a book 15 days, (b) a guest who kept a book 8 days, (c) a student who kept a book 30 days, and (d) anyone who returned a book after exactly 7 days. Then run the program with these four loans to check.

Exercise 1.7. Write a function that rotates an array of n elements k places to the left (so that `1 2 3 4 5 6 7` rotated by 3 becomes `4 5 6 7 1 2 3`) in $O(n)$ time, using only a constant amount of extra memory, whatever k is. *Hint:* reverse three parts of the array.

Exercise 1.8. Write a program that transposes the 3×4 loan table of **Listing 1.28** into a 4×3 table, one row per week, and prints it.

Exercise 1.9. (a) An array holds $n = 1000$ elements. How many moves does `ARRAY-INSERT` make to insert at index 0, at index 250 and at index 1000? (b) Starting from an empty array with room for 1,000 elements, a program inserts 1,000 values, each at index 0. How many moves does it make in total? (c) How many if it inserts each at the end instead? (d) Express both totals in big-O notation.

Exercise 1.10. A `std::vector` that doubles its capacity, as in **Listing 1.32**, receives 1,000 `push_back`s starting from empty. (a) How many times does it grow? (b) How many elements are copied in total? (c) What is its final capacity? (d) Check (b) against Equation (1.5).

Exercise 1.11. Predict exactly what this Python program prints, then fix it so that only one element changes:

```
grid = [[0] * 3] * 2
grid[1][2] = 7
print(grid)
```

Exercise 1.12. Find every mistake in this C++ fragment, and say which of them the compiler would report:

```
int squares[5];
for (int i = 1; i <= 5; i++) {
    squares[i] = i * i;
}
int total;
for (int i = 0; i < 5; i++) total += squares[i];
cout << "average = " << total / 5 << '\n';
```

Exercise 1.13. For each variable, say whether it lives in global data, on the stack or on the heap: (a) `int counter = 0;` written outside every function in a C++ program; (b) `int n = 8;` inside `main`; (c) the `int` created by `new int(5)`; (d) the pointer variable `p` in `int* p = new int(5);` inside `main`; (e) the elements of `std::vector<int> v(100);` declared inside `main`; (f) the array object made by `new int[10]` in Java; (g) the value `7` in the Python statement `x = 7`.

Exercise 1.14. Suppose the `Book` record of **Listing 1.13** gains a field `int yearPublished;` placed between `id` and `title`. Predict the offset of every field and `sizeof(Book)`, given that an `int` must begin at an address divisible by 4 and that the size of a struct is always a multiple of the alignment of its most demanding field. Check with a program.

Exercise 1.15. A friend setting up the workbench reports three messages. Explain each and give the fix: (a) on Windows, `'g++' is not recognized as an internal or external command`, although MSYS2 is installed; (b) `java -version` prints a version but `javac -version` is not found; (c) `python3 late_fee.py` stops with `SyntaxError` at the line `match member_type:`.

1.12 Solutions to the Exercises

Solution 1.1. (a) Insert, delete, search, traverse and sort. (b) Insert: save a new contact. Delete: remove a contact. Search: find “Sari” to call her. Traverse: scroll through all contacts, or back them up. Sort: show the contacts in alphabetical order. (c) One implementation keeps the contacts in an array sorted by name and halves the search range at each step; another keeps them in a hash table keyed by name. They share the *interface*, “give me the contact with this name, or say there is none”, and differ in the *implementation*: the arrangement in memory and the steps that find a contact, and therefore in cost.

Solution 1.2. (a) By Equation (1.1), $2^{16} = 65{,}536$ and $2^{32} = 4{,}294{,}967{,}296$. (b) The bytes of `0x12345678` from most to least significant are `12 34 56 78`. Little-endian stores the least significant first: 2000 holds `78`, 2001 `56`, 2002 `34`, 2003 `12`. Big-endian stores them in reading order: `12`, `34`, `56`, `78`. (c) $11111111 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$, which is `ff` in hexadecimal: each group of four ones is 15, the digit `f`.

Solution 1.3. By Equation (1.2) with $b = 5000$ and $s = 8$: (a) $5000 + 17 \cdot 8 = 5136$. (b) $5320 = 5000 + i \cdot 8$ gives $i = 320/8 = 40$, so `prices[40]`. (c) No: $(5324 - 5000)/8 = 40.5$ is not a whole number, so 5324 is the fifth byte inside `prices[40]`. (d) The array occupies $50 \cdot 8 = 400$ bytes, from 5000 to 5399, so the next byte is 5400, which is also the address that `prices + 50` would compute; it may be computed but must never be read.

Solution 1.4. Here $R = 6$, $C = 9$, $s = 4$ and $b = 2000$. (a) By Equation (1.3): $2000 + (4 \cdot 9 + 7) \cdot 4 = 2000 + 43 \cdot 4 = 2172$. (b) By Equation (1.4): $2000 + (7 \cdot 6 + 4) \cdot 4 = 2000 + 46 \cdot 4 = 2184$. (c) Position 50 in the unrolled row is row $\lfloor 50/9 \rfloor = 5$, column $50 - 5 \cdot 9 = 5$: the element `m[5][5]`.

Solution 1.5. Unroll `A[D1][D2][D3]` into one row. The element `A[i][j][k]` is preceded by i complete "planes" of $D_2 \cdot D_3$ elements, then j complete rows of D_3 elements within its own plane, then k elements within its row. Its position is $i \cdot D_2 D_3 + j \cdot D_3 + k$, which factors as $(i \cdot D_2 + j) \cdot D_3 + k$, and so

$$\text{address}(A[i][j][k]) = b + ((i \cdot D_2 + j) \cdot D_3 + k) \cdot s \tag{1.9}$$

Like Equation (1.3), Equation (1.9) does not use the first dimension D_1. For `int a[2][3][4]`, `a[1][2][3]` is at position $(1 \cdot 3 + 2) \cdot 4 + 3 = 23$, which is $23 \cdot 4 = 92$ bytes after `a[0][0][0]`; it is the last of the 24 elements. **Listing 1.44** measures every address; **Listing 1.45** and **Listing 1.46**, which cannot see addresses, store the elements in one flat array at the position Equation (1.9) computes and check that the positions come out in the order a row-major walk visits them. All three print the two lines shown after the listings.

Listing 1.44 `sol_address3d.cpp`: Equation (1.9) checked against the real address of all 24 elements.

```cpp
// sol_address3d.cpp - Solution 1.5: the row-major address formula in 3-D
#include <iostream>
using namespace std;

const int D1 = 2, D2 = 3, D3 = 4;

int main() {
    int a[D1][D2][D3];
    const char* base = reinterpret_cast<const char*>(&a[0][0][0]);
    int checked = 0, agreed = 0;
    for (int i = 0; i < D1; i++) {
        for (int j = 0; j < D2; j++) {
            for (int k = 0; k < D3; k++) {
                const char* here = reinterpret_cast<const char*>(&a[i][j][k]);
                long actual = static_cast<long>(here - base);
                long position = (i * D2 + j) * D3 + k;
                long formula = position * static_cast<long>(sizeof(int));
                checked++;
                if (actual == formula) agreed++;
            }
        }
    }
    cout << "int a[" << D1 << "][" << D2 << "][" << D3 << "]: " << agreed << " of "
         << checked << " addresses agree with the formula\n";
    long offset = ((1 * D2 + 2) * D3 + 3) * static_cast<long>(sizeof(int));
    cout << "a[1][2][3] is " << offset << " bytes after a[0][0][0]\n";
    return 0;
}
```

Listing 1.45 `SolAddress3D.java`: the three-dimensional position formula checked in Java.

```java
// SolAddress3D.java - Solution 1.5: the row-major position formula in 3-D
public class SolAddress3D {
    static final int D1 = 2, D2 = 3, D3 = 4;

    public static void main(String[] args) {
        // Store a[i][j][k] flat, in one int[] of D1*D2*D3 slots, as C++ does.
        int[] flat = new int[D1 * D2 * D3];
        int next = 0, checked = 0, agreed = 0;
        for (int i = 0; i < D1; i++) {
            for (int j = 0; j < D2; j++) {
                for (int k = 0; k < D3; k++) {
                    int position = (i * D2 + j) * D3 + k;
                    flat[position] = next;    // next: the row-major visiting order
                    checked++;
                    if (flat[position] == position) agreed++;
                    next++;
                }
            }
        }
        System.out.println("int a[" + D1 + "][" + D2 + "][" + D3 + "]: " + agreed
                + " of " + checked + " addresses agree with the formula");
        int offset = ((1 * D2 + 2) * D3 + 3) * Integer.BYTES;
        System.out.println("a[1][2][3] is " + offset + " bytes after a[0][0][0]");
    }
}
```

Listing 1.46 `sol_address3d.py`: the three-dimensional position formula checked in Python.

```python
# sol_address3d.py - Solution 1.5: the row-major position formula in 3-D
D1, D2, D3 = 2, 3, 4
INT_BYTES = 4

# Store a[i][j][k] flat, in one list of D1*D2*D3 slots, as C++ does.
flat = [None] * (D1 * D2 * D3)
visit = 0
checked = agreed = 0
for i in range(D1):
    for j in range(D2):
        for k in range(D3):
            position = (i * D2 + j) * D3 + k
            flat[position] = visit       # visit is the row-major visiting order
            checked += 1
            if flat[position] == position:
                agreed += 1
            visit += 1
print(f"int a[{D1}][{D2}][{D3}]: {agreed} of {checked}"
      " addresses agree with the formula")
offset = ((1 * D2 + 2) * D3 + 3) * INT_BYTES
print(f"a[1][2][3] is {offset} bytes after a[0][0][0]")
```

```
int a[2][3][4]: 24 of 24 addresses agree with the formula
a[1][2][3] is 92 bytes after a[0][0][0]
```

Solution 1.6. (a) A lecturer, 15 days: 8 days late, base $7 \cdot 1000 + 1 \cdot 2000 = 9{,}000$, full price: Rp9,000. (b) A guest, 8 days: 1 day late, base 1,000, doubled: Rp2,000. (c) A student, 30 days: 23 days late, base $7000 + 16 \cdot 2000 = 39{,}000$, halved: Rp19,500. (d) Exactly 7 days: `daysLate` is 0, and the first `if` returns 0 before the member type is even examined, so the fee is Rp0 for everyone, including an unknown member type. Replacing the six loans in `main` with these four and running the program prints exactly these fees and a total of Rp30,500.

Solution 1.7. Rotating left by k is the same as rotating by $k \bmod n$, where mod gives the remainder of a division, so first reduce k. Then reverse the first k elements, reverse the remaining $n - k$, and reverse the whole array. With `1 2 3 4 5 6 7` and $k = 3$: reversing `1 2 3` gives `3 2 1 4 5 6 7`; reversing `4 5 6 7` gives `3 2 1 7 6 5 4`; reversing everything gives `4 5 6 7 1 2 3`. Each reversal of m elements makes $\lfloor m/2 \rfloor$ swaps, so the total is at most n swaps: $O(n)$ time, and only the variable `t` of extra memory. **Listing 1.47**, **Listing 1.48** and **Listing 1.49** implement it; rotating once more by 11, which is 4 modulo 7, brings the array back to its start, because $3 + 4 = 7$ places is a full turn.

Listing 1.47 `sol_rotate.cpp`: rotation by three reversals.

```cpp
// sol_rotate.cpp - Solution 1.7: rotate an array left by k with three reversals
#include <iostream>
using namespace std;

void reverseRange(int a[], int lo, int hi) {         // reverses a[lo..hi]
    while (lo < hi) {
        int t = a[lo];
        a[lo] = a[hi];
        a[hi] = t;
        lo++;
        hi--;
    }
}

void rotateLeft(int a[], int n, int k) {
    k %= n;
    reverseRange(a, 0, k - 1);
    reverseRange(a, k, n - 1);
    reverseRange(a, 0, n - 1);
}

void print(const int a[], int n) {
    for (int i = 0; i < n; i++) cout << (i ? " " : "") << a[i];
    cout << '\n';
}

int main() {
    int a[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof a / sizeof a[0];
    cout << "before        : "; print(a, n);
    rotateLeft(a, n, 3);
    cout << "rotated by 3  : "; print(a, n);
    rotateLeft(a, n, 11);                 // 11 mod 7 = 4 more places
    cout << "rotated by 11 : "; print(a, n);
    return 0;
}
```

Listing 1.48 `SolRotate.java`: rotation by three reversals, in Java.

```java
// SolRotate.java - Solution 1.7: rotate an array left by k with three reversals
public class SolRotate {
    static void reverseRange(int[] a, int lo, int hi) {   // reverses a[lo..hi]
        while (lo < hi) {
            int t = a[lo];
            a[lo] = a[hi];
            a[hi] = t;
            lo++;
            hi--;
        }
    }

    static void rotateLeft(int[] a, int k) {
        int n = a.length;
        k %= n;
        reverseRange(a, 0, k - 1);
        reverseRange(a, k, n - 1);
        reverseRange(a, 0, n - 1);
    }

    static String show(int[] a) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < a.length; i++) sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.toString();
    }

    public static void main(String[] args) {
        int[] a = {1, 2, 3, 4, 5, 6, 7};
        System.out.println("before        : " + show(a));
        rotateLeft(a, 3);
        System.out.println("rotated by 3  : " + show(a));
        rotateLeft(a, 11);                    // 11 mod 7 = 4 more places
        System.out.println("rotated by 11 : " + show(a));
    }
}
```

Listing 1.49 `sol_rotate.py`: rotation by three reversals, in Python.

```python
# sol_rotate.py - Solution 1.7: rotate a list left by k with three reversals
def reverse_range(a, lo, hi):            # reverses a[lo..hi]
    while lo < hi:
        a[lo], a[hi] = a[hi], a[lo]
        lo += 1
        hi -= 1


def rotate_left(a, k):
    n = len(a)
    k %= n
    reverse_range(a, 0, k - 1)
    reverse_range(a, k, n - 1)
    reverse_range(a, 0, n - 1)


a = [1, 2, 3, 4, 5, 6, 7]
print("before        :", *a)
rotate_left(a, 3)
print("rotated by 3  :", *a)
rotate_left(a, 11)                       # 11 mod 7 = 4 more places
print("rotated by 11 :", *a)
```

```
before        : 1 2 3 4 5 6 7
rotated by 3  : 4 5 6 7 1 2 3
rotated by 11 : 1 2 3 4 5 6 7
```

Solution 1.8. The transpose T of an $R \times C$ table M is the $C \times R$ table with $T[c][r] = M[r][c]$: row r of M becomes column r of T. Two nested loops over M fill T. **Listing 1.50**, **Listing 1.51** and **Listing 1.52** print the same table: each row is now one week, and each column one shelf.

Listing 1.50 `sol_transpose.cpp`: transposing the loan table.

```cpp
// sol_transpose.cpp - Solution 1.8: transpose a 3 x 4 matrix into a 4 x 3 matrix
#include <iomanip>
#include <iostream>
using namespace std;

const int R = 3, C = 4;

int main() {
    int m[R][C] = {
        {31, 28, 35, 40},
        {12, 15, 9, 14},
        {20, 22, 18, 25},
    };
    int t[C][R];
    for (int r = 0; r < R; r++) {
        for (int c = 0; c < C; c++) {
            t[c][r] = m[r][c];
        }
    }
    cout << "transpose (" << C << " x " << R << "):\n";
    for (int r = 0; r < C; r++) {
        for (int c = 0; c < R; c++) cout << setw(4) << t[r][c];
        cout << '\n';
    }
    return 0;
}
```

Listing 1.51 `SolTranspose.java`: transposing the loan table, in Java.

```java
// SolTranspose.java - Solution 1.8: transpose a 3 x 4 matrix into a 4 x 3 matrix
public class SolTranspose {
    public static void main(String[] args) {
        int[][] m = {
            {31, 28, 35, 40},
            {12, 15, 9, 14},
            {20, 22, 18, 25},
        };
        int rows = m.length, cols = m[0].length;
        int[][] t = new int[cols][rows];
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                t[c][r] = m[r][c];
            }
        }
        System.out.println("transpose (" + cols + " x " + rows + "):");
        for (int[] row : t) {
            for (int x : row) System.out.printf("%4d", x);
            System.out.println();
        }
    }
}
```

Listing 1.52 `sol_transpose.py`: transposing the loan table, in Python.

```python
# sol_transpose.py - Solution 1.8: transpose a 3 x 4 matrix into a 4 x 3 matrix
m = [
    [31, 28, 35, 40],
    [12, 15, 9, 14],
    [20, 22, 18, 25],
]
rows, cols = len(m), len(m[0])
t = [[0] * rows for _ in range(cols)]
for r in range(rows):
    for c in range(cols):
        t[c][r] = m[r][c]

print(f"transpose ({cols} x {rows}):")
for row in t:
    print("".join(f"{x:>4}" for x in row))
```

```
transpose (4 x 3):
  31  12  20
  28  15  22
  35   9  18
  40  14  25
```

Solution 1.9. (a) By Equation (1.6): $1000 - 0 = 1000$, $1000 - 250 = 750$, and $1000 - 1000 = 0$. (b) The j-th insertion, $j = 0,1, \ldots, 999$, is made into an array of j elements at index 0 and moves all j of them, so the total is $0 + 1 + \cdots + 999 = \frac{999 \cdot 1000}{2} = 499{,}500$ moves. (c) Zero moves: every insertion at the end moves nothing. (d) For n insertions, the first total is $\frac{n(n-1)}{2}$, which is $O(n^2)$, and the second is 0 moves plus n writes, which is $O(n)$. Choosing the end instead of the front turns half a million moves into none.

Solution 1.10. The capacity goes $0 \to 1 \to 2 \to 4 \to \cdots \to 512 \to 1024$. (a) That is 11 growths. (b) Each growth copies the elements present, $0,1,2,4, \ldots, 512$, for a total of $1 + 2 + 4 + \cdots + 512 = 1023$ copies. (c) 1024. (d) Equation (1.5) with $2^k = 512 < 1000$ gives $2^{k+1} - 1 = 1023 < 2 \cdot 1000$, as found: on average about one copy per `push_back`.

Solution 1.11. `[[0] * 3] * 2` is a list of two references to *one* row, so changing `grid[1][2]` changes "both" rows, and the program prints `[[0, 0, 7], [0, 0, 7]]`. The fix makes two separate rows: `grid = [[0] * 3 for _ in range(2)]`, after which it prints `[[0, 0, 0], [0, 0, 7]]` (**Listing 1.31**).

Solution 1.12. Three mistakes. None of them is a compile *error*: the fragment compiles. Compiled with GCC 13 and `-O2 -Wall -Wextra`, however, it draws four *warnings*, `iteration 4 invokes undefined behavior`, `array subscript 5 is above array bounds of 'int [5]'`, `'squares' is used uninitialized` and `'total' is used uninitialized`, which is one more reason always to compile with warnings on and to read them. (1) The first loop runs $i = 1$ to 5: it never sets `squares[0]` and writes `squares[5]`, past the end, which is undefined behavior. It should be `for (int i = 0; i < 5; i++) squares[i] = (i + 1) * (i + 1);`, if the squares $1,4, \ldots, 25$ are wanted. (2) `total` is not initialized, so the sum starts from whatever the stack held; write `int total = 0;`. (3) `total / 5` is integer division, and no compiler warns about it; for the squares 1 to 25 the average is $55/5 = 11$ exactly, but in general write `static_cast<double>(total) / 5`.

Solution 1.13. (a) Global data: it is declared outside every function and lives for the whole run. (b) The stack, in `main`'s frame. (c) The heap. (d) The stack: the *pointer* is a local variable of `main`; only the `int` it points to is on the heap. (e) The heap: the `vector` object itself (its three fields) is in `main`'s frame, but its block of 100 elements is on the heap, which is how it can grow. (f) The heap:

every Java array is an object. (g) The heap: every Python value is an object, and `x` is a name referring to it.

Solution 1.14. `id` takes offsets 0 to 5. The `int` must begin at a multiple of 4, so offsets 6 and 7 become padding and `yearPublished` takes 8 to 11. `title` begins at 12 and ends at 111; the name of the writer takes 112 to 171; `category` 172 to 201. The fields end at 202, but the struct's size must be a multiple of 4, the alignment of its `int`, so that in an array every element's `int` stays aligned; two more bytes of padding give `sizeof = 204`, four more than the 200 bytes of the fields. **Listing 1.53** confirms every number. Java and Python do not let a program see the layout of its objects, so this exercise has only a C++ program.

Listing 1.53 `sol_padding.cpp`: offsets and size of a `Book` with an `int` field.

```cpp
// sol_padding.cpp - Solution 1.14: an int field brings alignment padding
#include <cstddef>
#include <iostream>
using namespace std;

struct BookWithYear {
    char id[6];
    int yearPublished;
    char title[100];
    char author[60];
    char category[30];
};

int main() {
    cout << "alignof(int)          = " << alignof(int) << '\n';
    cout << "offset of id          = " << offsetof(BookWithYear, id) << '\n';
    cout << "offset of year        = " << offsetof(BookWithYear, yearPublished)
         << '\n';
    cout << "offset of title       = " << offsetof(BookWithYear, title) << '\n';
    cout << "offset of author      = " << offsetof(BookWithYear, author) << '\n';
    cout << "offset of category    = " << offsetof(BookWithYear, category) << '\n';
    cout << "sum of field sizes    = " << 6 + sizeof(int) + 100 + 60 + 30 << '\n';
    cout << "sizeof(BookWithYear)  = " << sizeof(BookWithYear) << '\n';
    return 0;
}
```

```
alignof(int)          = 4
offset of id          = 0
offset of year        = 8
offset of title       = 12
offset of author      = 112
offset of category    = 172
sum of field sizes    = 200
sizeof(BookWithYear)  = 204
```

Solution 1.15. (a) The folder `C:\msys64\ucrt64\bin` is not on the user Path, or the terminal was opened before it was added: add it as in Section 1.7.3 and open a new terminal. (b) Only a Java *runtime* is installed, which runs programs but has no compiler: install the JDK of **Table 1.9**. If a JDK is installed, an older runtime is earlier on the PATH. (c) The `match` statement arrived in Python 3.10; the installed Python is older. Install Python 3.14 (**Table 1.10**) and check `python3 --version`.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed. Hoboken, NJ, USA: Pearson, 2021.

[3] TIOBE Software, "TIOBE Index for September 2026," *TIOBE Programming Community Index*. [Online]. Available: https://www.tiobe.com/tiobe-index/ (accessed Sep. 2026).

[4] TechRepublic, "TIOBE Index September 2026: Julia nears top 20," Sep. 2026. [Online]. Available: https://www.techrepublic.com/article/news-tiobe-september-2026-julia-nears-top-20/

[5] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed. Boston, MA, USA: Pearson, 2016.

[6] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[7] cppreference.com, "Array declaration." [Online]. Available: https://en.cppreference.com/w/cpp/language/array (accessed Sep. 2026).

[8] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 2.2.6, "Arrays and orthogonal lists."

[9] cppreference.com, "std::vector." [Online]. Available: https://en.cppreference.com/w/cpp/container/vector (accessed Sep. 2026).

[10] Oracle, "Class ArrayList," *Java SE 25 & JDK 25 API Specification*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/ArrayList.html

[11] OpenJDK, "java/util/ArrayList.java," *JDK source code*. [Online]. Available: https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/ArrayList.java

[12] Python Software Foundation, "Objects/listobject.c: list_resize," *CPython source code*. [Online]. Available: https://github.com/python/cpython/blob/main/Objects/listobject.c

[13] Python Software Foundation, "array — Efficient arrays of numeric values," *The Python Standard Library*, Python 3.14. [Online]. Available: https://docs.python.org/3/library/array.html

[14] Microsoft, "Using GCC with MinGW," *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/cpp/config-mingw (accessed Sep. 2026).

[15] Microsoft, "Visual Studio Code updates," *Visual Studio Code Release Notes*. [Online]. Available: https://code.visualstudio.com/updates (accessed Sep. 2026).

[16] GNU Project, "GCC 16.2 Released," gcc mailing list, Aug. 7, 2026. [Online]. Available: https://gcc.gnu.org/pipermail/gcc/2026-August/248711.html

[17] LLVM Project, "LLVM 23.1.0 Released!," LLVM Discourse, Aug. 26, 2026. [Online]. Available: https://discourse.llvm.org/t/llvm-23-1-0-released/91654

[18] MSYS2 Developers, "MSYS2: Software distribution and building platform for Windows." [Online]. Available: https://www.msys2.org

[19] Inside.java, "The Arrival of Java 27!," Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/

[20] Oracle, "Oracle Java SE Support Roadmap." [Online]. Available: https://www.oracle.com/java/technologies/java-se-support-roadmap.html (accessed Sep. 2026).

[21] Eclipse Foundation, "Eclipse Temurin," *Adoptium*. [Online]. Available: https://adoptium.net

[22] Python Software Foundation, "Python 3.14.7," Aug. 5, 2026. [Online]. Available: https://www.python.org/downloads/latest/python3.14/

Chapter 2
Searching

By the end of this chapter you will be able to answer the oldest question a collection of data is ever asked — *is it here, and where?* — in six different ways, and to choose between them with numbers rather than guesses. You will write sequential search with and without a sentinel and count exactly what each one costs; write binary search iteratively and recursively, state the loop invariant that makes it correct, and prove that it never needs more than $\lfloor \lg n \rfloor + 1$ probes; recognize and avoid the off-by-one traps that have broken binary searches in textbooks and in production libraries for decades; turn binary search into `lower_bound` and `upper_bound`, which answer the far more useful questions "where does the first copy start?", "how many copies are there?" and "where would a new item go?"; estimate a position with interpolation search and explain why it is brilliant on evenly spread keys and terrible on skewed ones; jump through an array in blocks of $\sqrt{n}$; search two-dimensional arrays in three different ways; measure all of these on a real catalog of ten thousand books in C++, Java and Python; and reach for `std::lower_bound`, `Arrays.binarySearch` and Python's `bisect` knowing precisely what each returns. Throughout, the running example is the catalog of **Pustaka Ceria**, the small library whose collection we began storing in Chapter 1.

2.1 Where We Left Off

Chapter 1 gave us two things. The first was a working toolchain for C++, Java and Python on whatever computer you own. The second was a way of *thinking* about data in memory: an array is a block of equally sized cells laid side by side, cell i lives at the address

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

where w is the width of one element in bytes, and therefore any cell can be reached in one step, no matter how large i is. Equation (2.1) is the reason arrays are fast, and it is also the reason this chapter can exist at all: every search algorithm we meet is a strategy for deciding *which* cells to look at, trusting that each look costs the same small amount.

Pustaka Ceria, our library, now has its books stored in an array. The librarian's very next wish is the one every owner of a collection has: a borrower walks up and asks for a book by its code, and the librarian needs to know, quickly, whether the library has it and, if so, where it is. That is **searching**, and it is the most frequently performed operation in computing. Every time you log in, a system searches for your account. Every time a shop scans a barcode, a system searches for the product. Every time an answer engine quotes a web page, something searched an index for it first.

This chapter is built around one idea, stated plainly now so that you can watch it unfold: **the more you know about how the data is arranged, the less of it you need to look at.** If you know nothing, you must look at everything (sequential search, Section 2.3). If you know the data is sorted, you can throw half of it away with every look (binary search, Section 2.5). If you know the data is sorted *and* evenly spread, you can often land almost on top of the answer with your first look (interpolation search, Section 2.7). Every gain is paid for by an assumption, and the whole art is knowing which assumptions your data actually satisfies.

2.2 The Searching Problem

2.2.1 Records, keys and the found/not-found contract

A library catalog is a collection of **records**, one per book. Each record has several **fields**: a code, a title, a writer's name, perhaps a year. When we search, we search on one field, called the **key**. The key we are looking for is the **search key** or **target**.

> **DEFINITION.** The **searching problem**: given a collection of n records $A[0], A[1], \ldots, A[n-1]$ and a search key x, return an index i such that the key of $A[i]$ equals x, or report that no such index exists. A search that finds the key is **successful**; one that does not is **unsuccessful**.

Notice what the definition asks for. It does not ask for a yes or a no; it asks for a *position*. This matters more than it seems. The first programs most of us write answer "FOUND" or "NOT FOUND" and stop there, and the very next thing the caller wants to do — print the title, mark the book as borrowed, change its shelf — requires the position, which the program has thrown away. So every function in this chapter returns an index, and uses -1 to mean "not found", because -1 is never a valid array index. The caller can then write `if (i >= 0) ...` and has everything it needs.

2.2.2 How we measure the cost of a search

To compare search algorithms fairly we need a unit of work that does not depend on the speed of the computer. The natural one is the **probe**: one inspection of one array element, in which the element's key is compared with the search key. A probe may involve one comparison (`==`) or two (`==` then `<`), but it is one visit to one cell, and it is the visits we want to count. When we measure time as well, in Section 2.10, we shall see how well probe counts predict the clock — and where they do not.

For each algorithm we shall ask three questions:

- **Best case**: the fewest probes over all inputs of size n.
- **Worst case**: the most probes over all inputs of size n. This is the guarantee: no search will ever cost more.
- **Average case**: the expected number of probes when the search key is chosen according to some stated assumption, usually "every position equally likely".

We shall summarize these costs with the notation $O(\cdot)$, read "big-O" or "order of".

> **DEFINITION.** For functions f and g of n, we write $f(n) = O(g(n))$ if there are positive constants c and n_0 such that $f(n) \le c \cdot g(n)$ for every $n \ge n_0$. In words: from some size onwards, f never exceeds a fixed multiple of g.

For example, $(n+1)/2 = O(n)$, because $(n+1)/2 \le n$ for every $n \ge 1$ (take $c = 1$, $n_0 = 1$). The notation deliberately ignores constant factors and small inputs, and keeps only the *rate of growth*: $O(n)$ grows in proportion to n, $O(\lg n)$ grows by one each time n doubles, and $O(1)$ does not grow at all. We write $\lg n$ for $\log_2 n$, the logarithm to base two, which counts how many times n can be halved before it reaches one; $\lg 8 = 3$ because $8 \to 4 \to 2 \to 1$ takes three halvings. Chapter 10 returns to this notation and adds its companions for lower bounds and tight bounds.

2.2.3 Pustaka Ceria's computing shelf

The running example of this chapter is the computing shelf of Pustaka Ceria, eight books, each with a three-digit code. **Table 2.1** lists them in the order of their codes.

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

Books do not arrive in code order, of course. They arrive in the order they were bought, and the shelf in the reading room holds them in that **arrival order**: 072, 025, 096, 041, 088, 034, 063, 056. Keep both orders in mind. The arrival order is what a library has before anyone has done any work; the sorted order is what it has after someone has sorted it — the subject of Chapter 3. Which searches are possible depends on which of the two we have.

2.3 Sequential Search

2.3.1 The idea

Suppose you are handed a box of eight index cards in no particular order and asked whether code 088 is among them. You have no choice: you take the first card and look, then the second, then the third, and you stop when you find 088 or when the box is empty. That is **sequential search**, also called **linear search**. It assumes nothing about the order of the data, which is exactly its strength — it works on any array, sorted or not — and exactly its weakness, because an algorithm that assumes nothing cannot skip anything.

Figure 2.1 shows sequential search for code 088 on the shelf in arrival order. The first four cards are not 088; the fifth is.

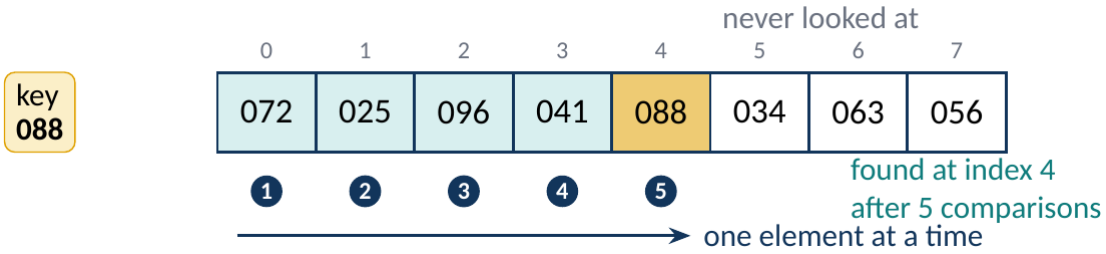


Figure 2.1 Sequential search for code 088 on the shelf in arrival order. The numbered circles give the order of the probes; the search stops at the fifth one, and the last three cells are never examined.

Pseudocode 2.1 writes the procedure down. We use the pseudocode style of Cormen et al. [1]: numbered lines, ← for assignment, and indentation for the body of a loop.

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
```

2.3.2 From a flag to an index

The first version of sequential search that most people write uses a **flag**: a variable that starts at 0 and is set to 1 when a match is seen. The loop always runs to the end of the array, and after the loop the program prints FOUND if the flag is 1 and NOT FOUND otherwise. It is correct, but it has two faults. It keeps looking after it has found the key, so it always makes n comparisons; and it forgets *where* it found the key, so the caller cannot use the answer.

Both faults are cured by the same change: return from inside the loop the moment the key is found, and return the index. **Pseudocode 2.1** already does this. There is one situation, however, in which the loop *must* run to the end, and it is worth recognizing: when the question is not "is it here?" but "how many times is it here?" or "where are all the copies?". A loan log that records the code of every book borrowed today contains repeats, and counting how often book 041 went out requires looking at every entry. The rule is: stop early for existence, never stop early for counting.

2.3.3 The program

Listing 2.1 is the complete C++ program. It runs the flag version and the early-exit version on the array [8,10,6, −2,11,7,1,100], counts the occurrences of 041 in a loan log, and then searches the shelf of **Table 2.1** — in arrival order — by code and by title. Every function counts its comparisons so that the output shows the cost, not just the answer.

Listing 2.1 `sequential_search.cpp`: sequential search with a flag, with early exit, for counting, and on records.

```cpp
// Sequential search: on plain numbers, in a loan log, and on Pustaka Ceria's shelf.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

struct Book {
    int code;
    string title;
    string writer;
};

string plural(int count, const string& word) {
    return to_string(count) + " " + word + (count == 1 ? "" : "s");
}

// First version: look at every element and remember only yes or no.
int searchFlag(const vector<int>& data, int key, int& comparisons) {
    int flag = 0;
    comparisons = 0;
    for (size_t i = 0; i < data.size(); i++) {
        comparisons++;
        if (data[i] == key) flag = 1;
    }
    return flag;
}

// 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 (Continued) *sequential_search.cpp: sequential search with a flag, with early exit, for counting, and on records.*

```
int findByCode(const vector<Book>& shelf, int code, int& comparisons) {
    comparisons = 0;
    for (size_t i = 0; i < shelf.size(); i++) {
        comparisons++;
        if (shelf[i].code == code) return (int)i;
    }
    return -1;
}

int findByTitle(const vector<Book>& shelf, const string& title, int& comparisons) {
    comparisons = 0;
    for (size_t i = 0; i < shelf.size(); i++) {
        comparisons++;
        if (shelf[i].title == title) return (int)i;
    }
    return -1;
}

int main() {
    vector<int> data = {8, 10, 6, -2, 11, 7, 1, 100};
    printf("data:");
    for (int x : data) printf(" %d", x);
    printf("\n");
    int c;
    for (int key : {6, 5}) {
        int flag = searchFlag(data, key, c);
        printf("flag version, key %d: %s after %s\n", key,
               flag == 1 ? "FOUND" : "NOT FOUND", plural(c, "comparison").c_str());
    }
    for (int key : {8, 6, 100, 5}) {
        int i = sequentialSearch(data, key, c);
        if (i >= 0)
            printf("early exit,   key %d: found at index %d after %s\n", key, i,
                   plural(c, "comparison").c_str());
        else
            printf("early exit,   key %d: not found after %s\n", key,
                   plural(c, "comparison").c_str());
    }
```

Listing 2.1 (Continued) *sequential_search.cpp: sequential search with a flag, with early exit, for counting, and on records.*

```
    vector<int> loans = {41, 88, 25, 41, 63, 41, 96, 88};
    vector<int> where = countOccurrences(loans, 41);
    printf("loan log:");
    for (int x : loans) printf(" %03d", x);
    printf("\nbook 041 was borrowed %zu times, at positions", where.size());
    for (int p : where) printf(" %d", p);
    printf("\n");

    vector<Book> shelf = {{72, "Calculus Make it Easy", "Gunner Christian"},
                          {25, "The C++ Programming", "James Wood"},
                          {96, "Artificial Life : Volume 1", "Gloria Virginia"},
                          {41, "Professional C#", "Simon Webe"},
                          {88, "Visual Basic 2005 Express", "Antonie"},
                          {34, "Mastering Delphi 6", "Marcopolo"},
                          {63, "Advanced JSP & Servlet", "David Dunn"},
                          {56, "Pure JavaScript v2", "Michael Bolton"}};
    for (int code : {88, 60}) {
        int i = findByCode(shelf, code, c);
        if (i >= 0)
            printf("code %03d: index %d, %s by %s (%s)\n", code, i,
                   shelf[i].title.c_str(), shelf[i].writer.c_str(),
                   plural(c, "comparison").c_str());
        else
            printf("code %03d: not on the shelf (%s)\n", code,
                   plural(c, "comparison").c_str());
    }
    int i = findByTitle(shelf, "Professional C#", c);
    printf("title \"Professional C#\": index %d, code %03d (%s)\n", i,
           shelf[i].code, plural(c, "comparison").c_str());
    return 0;
}
```

Listing 2.2 is the same program in Java, and **Listing 2.3** in Python. In Java a `record` gives us the `Book` type in one line; in Python a small class does the same job. Each search function returns a pair — the index and the number of comparisons — as a two-element `int[]` in Java and a tuple in Python.

Listing 2.2 `SequentialSearch.java`: the same searches in Java.

```java
// Sequential search: on plain numbers, in a loan log, and on Pustaka Ceria's shelf.
import java.util.ArrayList;
import java.util.List;

public class SequentialSearch {
    record Book(int code, String title, String writer) {}

    static String plural(int count, String word) {
        return count + " " + word + (count == 1 ? "" : "s");
    }

    // First version: look at every element and remember only yes or no.
    // Returns {flag, comparisons}.
    static int[] searchFlag(int[] data, int key) {
        int flag = 0, comparisons = 0;
        for (int x : data) {
            comparisons++;
            if (x == key) flag = 1;
        }
        return new int[] {flag, comparisons};
    }

    // Stop at the first match. Returns {index or -1, comparisons}.
    static int[] sequentialSearch(int[] data, int key) {
        int comparisons = 0;
        for (int i = 0; i < data.length; i++) {
            comparisons++;
            if (data[i] == key) return new int[] {i, comparisons};
        }
        return new int[] {-1, comparisons};
    }

    // Every position holding key: the loop must not stop early.
    static List<Integer> countOccurrences(int[] data, int key) {
        List<Integer> where = new ArrayList<>();
        for (int i = 0; i < data.length; i++)
            if (data[i] == key) where.add(i);
        return where;
    }
```

Listing 2.2 (Continued) *SequentialSearch.java: the same searches in Java.*

```
    static int[] findByCode(Book[] shelf, int code) {
        int comparisons = 0;
        for (int i = 0; i < shelf.length; i++) {
            comparisons++;
            if (shelf[i].code() == code) return new int[] {i, comparisons};
        }
        return new int[] {-1, comparisons};
    }

    static int[] findByTitle(Book[] shelf, String title) {
        int comparisons = 0;
        for (int i = 0; i < shelf.length; i++) {
            comparisons++;
            if (shelf[i].title().equals(title)) return new int[] {i, comparisons};
        }
        return new int[] {-1, comparisons};
    }

    public static void main(String[] args) {
        int[] data = {8, 10, 6, -2, 11, 7, 1, 100};
        StringBuilder line = new StringBuilder("data:");
        for (int x : data) line.append(" ").append(x);
        System.out.println(line);
        for (int key : new int[] {6, 5}) {
            int[] r = searchFlag(data, key);
            System.out.printf("flag version, key %d: %s after %s%n", key,
                    r[0] == 1 ? "FOUND" : "NOT FOUND", plural(r[1], "comparison"));
        }
        for (int key : new int[] {8, 6, 100, 5}) {
            int[] r = sequentialSearch(data, key);
            if (r[0] >= 0)
                System.out.printf(
                        "early exit,   key %d: found at index %d after %s%n", key,
                        r[0], plural(r[1], "comparison"));
            else
                System.out.printf("early exit,   key %d: not found after %s%n", key,
                        plural(r[1], "comparison"));
        }
```

Listing 2.2 (Continued) *SequentialSearch.java: the same searches in Java.*

```
        int[] loans = {41, 88, 25, 41, 63, 41, 96, 88};
        List<Integer> where = countOccurrences(loans, 41);
        line = new StringBuilder("loan log:");
        for (int x : loans) line.append(String.format(" %03d", x));
        System.out.println(line);
        line = new StringBuilder(
                "book 041 was borrowed " + where.size() + " times, at positions");
        for (int p : where) line.append(" ").append(p);
        System.out.println(line);

        Book[] shelf = {new Book(72, "Calculus Make it Easy", "Gunner Christian"),
                new Book(25, "The C++ Programming", "James Wood"),
                new Book(96, "Artificial Life : Volume 1", "Gloria Virginia"),
                new Book(41, "Professional C#", "Simon Webe"),
                new Book(88, "Visual Basic 2005 Express", "Antonie"),
                new Book(34, "Mastering Delphi 6", "Marcopolo"),
                new Book(63, "Advanced JSP & Servlet", "David Dunn"),
                new Book(56, "Pure JavaScript v2", "Michael Bolton")};
        for (int code : new int[] {88, 60}) {
            int[] r = findByCode(shelf, code);
            if (r[0] >= 0)
                System.out.printf("code %03d: index %d, %s by %s (%s)%n", code,
                        r[0], shelf[r[0]].title(), shelf[r[0]].writer(),
                        plural(r[1], "comparison"));
            else
                System.out.printf("code %03d: not on the shelf (%s)%n", code,
                        plural(r[1], "comparison"));
        }
        int[] r = findByTitle(shelf, "Professional C#");
        System.out.printf("title \"Professional C#\": index %d, code %03d (%s)%n",
                r[0], shelf[r[0]].code(), plural(r[1], "comparison"));
    }
}
```

Listing 2.3 `sequential_search.py`: the same searches in Python.

```
"""Sequential search: plain numbers, a loan log, and Pustaka Ceria's shelf."""


class Book:
    def __init__(self, code, title, writer):
        self.code = code
        self.title = title
        self.writer = writer


def plural(count, word):
    return "%d %s%s" % (count, word, "" if count == 1 else "s")


def search_flag(data, key):
    """First version: look at every element and remember only yes or no."""
    flag = 0
    comparisons = 0
    for x in data:
        comparisons += 1
        if x == key:
            flag = 1
    return flag, comparisons


def sequential_search(data, key):
    """Stop at the first match; return (index, comparisons), index -1 if absent."""
    comparisons = 0
    for i in range(len(data)):
        comparisons += 1
        if data[i] == key:
            return i, comparisons
    return -1, comparisons


def count_occurrences(data, key):
    """Every position holding key: the loop must not stop early."""
    return [i for i in range(len(data)) if data[i] == key]
```

Listing 2.3 (Continued) *sequential_search.py: the same searches in Python.*

```
def find_by_code(shelf, code):
    comparisons = 0
    for i in range(len(shelf)):
        comparisons += 1
        if shelf[i].code == code:
            return i, comparisons
    return -1, comparisons


def find_by_title(shelf, title):
    comparisons = 0
    for i in range(len(shelf)):
        comparisons += 1
        if shelf[i].title == title:
            return i, comparisons
    return -1, comparisons


def main():
    data = [8, 10, 6, -2, 11, 7, 1, 100]
    print("data:", " ".join(str(x) for x in data))
    for key in (6, 5):
        flag, c = search_flag(data, key)
        print(
            "flag version, key %d: %s after %s"
            % (key, "FOUND" if flag == 1 else "NOT FOUND", plural(c, "comparison"))
        )
    for key in (8, 6, 100, 5):
        i, c = sequential_search(data, key)
        if i >= 0:
            print(
                "early exit,   key %d: found at index %d after %s"
                % (key, i, plural(c, "comparison"))
            )
        else:
            print(
                "early exit,   key %d: not found after %s"
                % (key, plural(c, "comparison"))
            )
```

Listing 2.3 (Continued) *sequential_search.py: the same searches in Python.*

```
    loans = [41, 88, 25, 41, 63, 41, 96, 88]
    where = count_occurrences(loans, 41)
    print("loan log:", " ".join("%03d" % x for x in loans))
    print(
        "book 041 was borrowed %d times, at positions %s"
        % (len(where), " ".join(str(p) for p in where))
    )

    shelf = [
        Book(72, "Calculus Make it Easy", "Gunner Christian"),
        Book(25, "The C++ Programming", "James Wood"),
        Book(96, "Artificial Life : Volume 1", "Gloria Virginia"),
        Book(41, "Professional C#", "Simon Webe"),
        Book(88, "Visual Basic 2005 Express", "Antonie"),
        Book(34, "Mastering Delphi 6", "Marcopolo"),
        Book(63, "Advanced JSP & Servlet", "David Dunn"),
        Book(56, "Pure JavaScript v2", "Michael Bolton"),
    ]
    for code in (88, 60):
        i, c = find_by_code(shelf, code)
        if i >= 0:
            print(
                "code %03d: index %d, %s by %s (%s)"
                % (
                    code,
                    i,
                    shelf[i].title,
                    shelf[i].writer,
                    plural(c, "comparison"),
                )
            )
        else:
            print(
                "code %03d: not on the shelf (%s)" % (code, plural(c, "comparison"))
            )
    i, c = find_by_title(shelf, "Professional C#")
    print(
        'title "Professional C#": index %d, code %03d (%s)'
        % (i, shelf[i].code, plural(c, "comparison"))
    )


if __name__ == "__main__":
    main()
```

All three programs print exactly the same lines:

```
data: 8 10 6 -2 11 7 1 100
flag version, key 6: FOUND after 8 comparisons
flag version, key 5: NOT FOUND after 8 comparisons
early exit,   key 8: found at index 0 after 1 comparison
early exit,   key 6: found at index 2 after 3 comparisons
early exit,   key 100: found at index 7 after 8 comparisons
early exit,   key 5: not found after 8 comparisons
loan log: 041 088 025 041 063 041 096 088
book 041 was borrowed 3 times, at positions 0 3 5
code 088: index 4, Visual Basic 2005 Express by Antonie (5 comparisons)
code 060: not on the shelf (8 comparisons)
title "Professional C#": index 3, code 041 (4 comparisons)
```

Read the output against the definitions. The flag version made 8 comparisons whether or not it found the key. The early-exit version found 8 in 1 comparison (the best case), 6 in 3, and 100 in 8 (the worst successful case), and needed all 8 to be sure that 5 is absent. Searching the shelf for 088 cost 5 comparisons, exactly as **Figure 2.1** predicted.

2.3.4 How much does sequential search cost?

Best case. The key is in $A[0]$: one probe. The best case is $O(1)$.

Worst case. The key is in $A[n-1]$, or it is absent. Either way every element is examined: n probes. The worst case is $O(n)$, and an unsuccessful search *always* costs exactly n.

Average case, successful. Assume the key is present and is equally likely to be at any of the n positions. If it is at position i, the search makes $i+1$ probes. Averaging over the n positions,

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

The step from the sum to $n(n+1)/2$ is the formula for $1+2+\cdots+n$, which you can check by writing the sum forwards and backwards and adding: each of the n columns adds to $n+1$, so twice the sum is $n(n+1)$. For our eight-book shelf Equation (2.2) gives 4.5 probes on average; for a catalog of ten thousand books it gives 5,000.5.

Average case, key may be absent. In real use a key is not always present. Let p be the probability that it is. With probability p the search costs $(n+1)/2$ on average, and with probability $1-p$ it costs n. Weighting the two,

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

At $p=1$ this is Equation (2.2); at $p=0$ it is n; and at $p=\frac{1}{2}$ it is about $\frac{3}{4}n$. In every case it grows in direct proportion to n: double the catalog and you double the average search. That is what $O(n)$ means in practice, and it is why sequential search, for all its simplicity, is the algorithm we try to avoid on large collections.

> **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 (Chapter 7). When the data is small, or will be searched only once, or cannot be sorted, it is the right tool. When the data is large and searched often, it is the tool to replace.

2.4 Sequential Search with a Sentinel

2.4.1 Two tests where one would do

Look closely at the loop of **Pseudocode 2.1**. On every step it asks *two* questions: "is i still less than n?" (hidden in the `for`) and "is $A[i]$ the key?". The first question exists only to stop the loop from running off the end of the array when the key is absent. On a successful search it is answered "yes" every time, and is pure overhead.

The **sentinel** trick removes it. Give the array one spare cell at index n, and before searching, put a copy of the key there. Now the loop is *guaranteed* to find the key — at worst in the spare cell — so it no longer needs to watch the boundary. When the loop stops at position i, a single test afterwards decides the answer: if $i < n$ the key was really there; if $i = n$ the loop found only its own copy, and the key is absent. **Figure 2.2** shows the idea on the array $[3,12,9,-4,21,6]$ with the key 5.

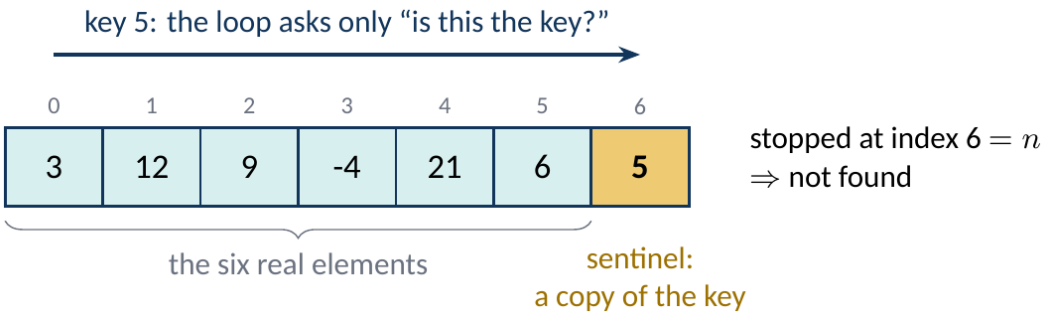


Figure 2.2 Sequential search with a sentinel. The key 5 is copied into the spare cell at index 6, so the loop must stop there at the latest; stopping at index $6 = n$ means the key is not among the real elements.

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
```

2.4.2 Counting the tests

Listing 2.4 makes the saving visible by counting every test the two loops perform. The plain search is written with both of its tests spelled out, so that each can be counted; **Listing 2.5** and **Listing 2.6** are the Java and Python versions.

Listing 2.4 `sentinel_search.cpp`: the plain loop and the sentinel loop, with every loop test counted.

```cpp
// Sequential search with and without a sentinel, counting every loop test.
#include <cstdio>
#include <string>
using namespace std;

// Two tests per step: "i < n" and "data[i] != key".
int plainSearch(const int data[], int n, int key, int& boundTests, int& keyTests) {
    boundTests = keyTests = 0;
    int i = 0;
    while (true) {
        boundTests++;
        if (!(i < n)) break;
        keyTests++;
        if (!(data[i] != key)) break;
        i++;
    }
    return i < n ? i : -1;
}

// 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;
}

int main() {
    const int n = 6;
    int data[n + 1] = {3, 12, 9, -4, 21, 6, 0};  // 6 real elements + 1 spare slot
    printf("data:");
    for (int i = 0; i < n; i++) printf(" %d", data[i]);
    printf(" | spare slot at index %d\n", n);
    for (int key : {21, 5}) {
        int b, k, s;
        int i = plainSearch(data, n, key, b, k);
        int j = sentinelSearch(data, n, key, s);
        string where = i >= 0 ? "index " + to_string(i) : "not found";
        string where2 = j >= 0 ? "index " + to_string(j) : "not found";
```

Listing 2.4 (Continued) *sentinel_search.cpp: the plain loop and the sentinel loop, with every loop test counted.*

```cpp
        printf("key %2d  plain:    %-9s %2d loop tests (%d bound + %d key)\n", key,
               where.c_str(), b + k, b, k);
        printf("        sentinel: %-9s %2d loop tests (key only)\n", where2.c_str(),
               s);
    }
    printf("spare slot after the searches: %d\n", data[n]);
    return 0;
}
```

Listing 2.5 `SentinelSearch.java`: the same comparison in Java.

```
// Sequential search with and without a sentinel, counting every loop test.
public class SentinelSearch {
    // Two tests per step: "i < n" and "data[i] != key".
    // Returns {index or -1, bound tests, key tests}.
    static int[] plainSearch(int[] data, int n, int key) {
        int boundTests = 0, keyTests = 0;
        int i = 0;
        while (true) {
            boundTests++;
            if (!(i < n)) break;
            keyTests++;
            if (!(data[i] != key)) break;
            i++;
        }
        return new int[] {i < n ? i : -1, boundTests, keyTests};
    }

    // data has one spare slot at index n; the key is planted there first.
    // Returns {index or -1, key tests}.
    static int[] sentinelSearch(int[] data, int n, int key) {
        int saved = data[n];
        // the sentinel: the loop is now sure to stop
        data[n] = key;
        int 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 new int[] {i < n ? i : -1, keyTests};
    }

    public static void main(String[] args) {
        int n = 6;
        int[] data = {3, 12, 9, -4, 21, 6, 0};  // 6 real elements + 1 spare slot
        StringBuilder line = new StringBuilder("data:");
        for (int i = 0; i < n; i++) line.append(" ").append(data[i]);
        System.out.println(line + " | spare slot at index " + n);
        for (int key : new int[] {21, 5}) {
            int[] p = plainSearch(data, n, key);
            int[] s = sentinelSearch(data, n, key);
            String where = p[0] >= 0 ? "index " + p[0] : "not found";
            String where2 = s[0] >= 0 ? "index " + s[0] : "not found";
            System.out.printf(
```

Listing 2.5 (Continued) *SentinelSearch.java: the same comparison in Java.*

```
                    "key %2d  plain:    %-9s %2d loop tests (%d bound + %d key)%n",
                    key, where, p[1] + p[2], p[1], p[2]);
            System.out.printf("        sentinel: %-9s %2d loop tests (key only)%n",
                    where2, s[1]);
        }
        System.out.println("spare slot after the searches: " + data[n]);
    }
}
```

Listing 2.6 `sentinel_search.py`: the same comparison in Python.

```
"""Sequential search with and without a sentinel, counting every loop test."""


def plain_search(data, n, key):
    """Two tests per step: 'i < n' and 'data[i] != key'."""
    bound_tests = key_tests = 0
    i = 0
    while True:
        bound_tests += 1
        if not i < n:
            break
        key_tests += 1
        if not data[i] != key:
            break
        i += 1
    return (i if i < n else -1), bound_tests, key_tests


def sentinel_search(data, n, key):
    """data has one spare slot at index n; the key is planted there first."""
    saved = data[n]
    data[n] = key  # the sentinel: the loop is now sure to stop
    key_tests = 0
    i = 0
    while True:
        key_tests += 1
        if not data[i] != key:
            break
        i += 1
    data[n] = saved  # leave the array as we found it
    return (i if i < n else -1), key_tests


def main():
    n = 6
    data = [3, 12, 9, -4, 21, 6, 0]  # 6 real elements + 1 spare slot
    print("data:", " ".join(str(x) for x in data[:n]), "| spare slot at index", n)
    for key in (21, 5):
        i, b, k = plain_search(data, n, key)
        j, s = sentinel_search(data, n, key)
        where = "index %d" % i if i >= 0 else "not found"
        where2 = "index %d" % j if j >= 0 else "not found"
        print(
            "key %2d  plain:    %-9s %2d loop tests (%d bound + %d key)"
            % (key, where, b + k, b, k)
        )
```

Listing 2.6 (Continued) *sentinel_search.py: the same comparison in Python.*

```
        print("        sentinel: %-9s %2d loop tests (key only)" % (where2, s))
    print("spare slot after the searches:", data[n])


if __name__ == "__main__":
    main()
```

All three print:

```
data: 3 12 9 -4 21 6 | spare slot at index 6
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)
spare slot after the searches: 0
```

For the key 21 at index 4 the plain loop performs 5 bound tests and 5 key tests; the sentinel loop performs only the 5 key tests. For the absent key 5 the plain loop performs $n + 1 = 7$ bound tests (the seventh is the one that fails) and $n = 6$ key tests, $2n + 1 = 13$ in all; the sentinel loop performs $n + 1 = 7$ key tests, the last of them on the sentinel. In general the plain loop needs about $2n$ tests in the worst case and the sentinel loop about n. The last line of the output confirms that the spare cell was restored.

2.4.3 What the sentinel does and does not buy

The sentinel halves the number of *tests*. It does not reduce the number of *probes*: both loops look at the same elements in the same order. It does not change the order of growth: both are $O(n)$. And it has costs of its own. The array needs a spare cell; the array must be writable, which rules out searching a constant table; and between planting the sentinel and removing it the array temporarily holds a value that is not really there, so two threads must never search the same array this way at the same time.

Is the halving worth it? On the machines of the 1970s, when Knuth analyzed the trick [2], a comparison and a branch were expensive and the saving was real. On a modern processor the answer is a measurement, not an argument, and Section 2.10 makes it: on our real catalog of ten thousand books the two loops ran at the same speed within measurement noise in C++ and in Java, because the processor predicts the always-true bound test almost perfectly and executes it alongside the key test. In Python, where every test is an interpreted operation, the sentinel version was about 9% faster. Learn the trick, because it appears in library code and in other algorithms (the insertion step of insertion sort in Chapter 3 can use it), but do not expect it to rescue a slow search. Only a better algorithm can do that.

2.5 Binary Search

2.5.1 Using the order

Now suppose the shelf is sorted by code, as in **Table 2.1**. Search for 088 again, but this time open the shelf in the middle. The middle book is 056. Because the shelf is sorted, every book to the left of 056 has a smaller code, so 088 cannot be there: in one look we have discarded half the shelf. Look at the middle of the remaining half, and repeat. This is **binary search**, and it is the most important algorithm in this chapter.

The principle is exactly how you look up a word in a paper dictionary: you do not read from *aardvark* onwards; you open somewhere, see that you are too early or too late, and continue in the half that can still contain your word.

> **DEFINITION. Binary search** searches a *sorted* array $A[0..n-1]$ for a key x by maintaining an interval $A[low..high]$ that must contain x if x is present anywhere. At each step it probes the middle element $A[mid]$ with $mid = \lfloor (low + high)/2 \rfloor$. If $A[mid] = x$ the search succeeds; if $A[mid] < x$ the interval becomes $A[mid+1..high]$; if $A[mid] > x$ it becomes $A[low..mid-1]$. When the interval is empty ($low > high$) the search fails.

2.5.2 A trace by hand

Take the sorted array $A = [3,9,11,12,15,17,23,31,35]$, with $n = 9$, and search for $x = 17$. **Figure 2.3** shows the three steps.



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.

- **Step 1.** $low = 0$, $high = 8$, so $mid = \lfloor 8/2 \rfloor = 4$ and $A[4] = 15$. Since $17 > 15$, the key can only be to the right: $low \leftarrow 5$.
- **Step 2.** $low = 5$, $high = 8$, so $mid = \lfloor 13/2 \rfloor = 6$ and $A[6] = 23$. Since $17 < 23$, the key can only be to the left: $high \leftarrow 5$.
- **Step 3.** $low = high = 5$, so $mid = 5$ and $A[5] = 17$. Found, at index 5.

Sequential search would have needed six probes. Now search for $x = 10$, which is absent:

- $low = 0, high = 8, mid = 4, A[4] = 15 > 10$, so $high \leftarrow 3$.
- $low = 0, high = 3, mid = 1, A[1] = 9 < 10$, so $low \leftarrow 2$.
- $low = 2, high = 3, mid = 2, A[2] = 11 > 10$, so $high \leftarrow 1$.
- Now $low = 2 > high = 1$: the interval is empty and 10 is absent.

Notice where low ended: at 2, which is exactly the index at which 10 would have to be inserted to keep the array sorted (between 9 and 11). This is not a coincidence, and Section 2.6 builds on it.

2.5.3 The procedure and its invariant

Pseudocode 2.3 is the iterative procedure.

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
```

Line 4 computes the middle as $low + \lfloor (high - low)/2 \rfloor$ rather than $\lfloor (low + high)/2 \rfloor$. Mathematically the two are equal; on a computer they are not, and Section 2.6.4 explains why.

Binary search is famous for being easy to describe and hard to write correctly. Knuth reports that although the first binary search was published in 1946, the first version that worked correctly for

every n did not appear until 1962 [2], and Bentley found that only about one professional programmer in ten could write a correct one in a couple of hours [3]. The defence against such bugs is not cleverness but a **loop invariant**: a statement that is true every time the loop condition is tested, and that together with the loop's exit gives the answer.

> **INVARIANT.** At the start of every iteration of the `while` loop in **Pseudocode 2.3**: every element of $A[0..low-1]$ is less than x, and every element of $A[high+1..n-1]$ is greater than x. Consequently, if x is anywhere in A, it is in $A[low..high]$.

Figure 2.4 pictures it: the array is always divided into three regions, and the search only ever works on the middle one.

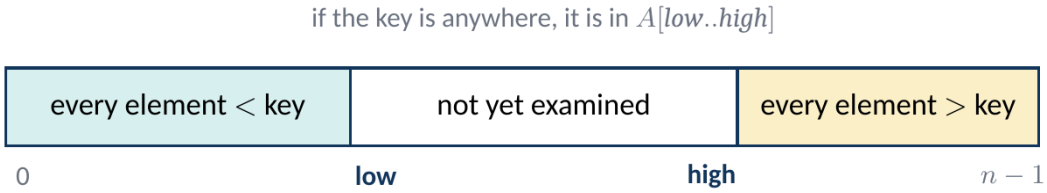


Figure 2.4 The loop invariant of binary search. Everything left of *low* is known to be smaller than the key, everything right of *high* is known to be larger, and the key, if present, is in the middle region.

We prove the invariant the way every loop invariant is proved, in three parts [1].

Initialization. Before the first iteration $low = 0$ and $high = n-1$. The regions $A[0..-1]$ and $A[n..n-1]$ are empty, and a statement about every element of an empty region is true.

Maintenance. Suppose the invariant holds at the start of an iteration, and the loop does not return. If $A[mid] < x$, then because the array is sorted every element of $A[low..mid]$ is at most $A[mid]$, hence less than x. Together with the invariant, every element of $A[0..mid]$ is less than x, so setting $low \leftarrow mid + 1$ keeps the first half of the invariant true; the second half is untouched. The case $A[mid] > x$ is the mirror image: every element of $A[mid..high]$ is at least $A[mid]$, hence greater than x, and $high \leftarrow mid - 1$ keeps the second half true.

Termination. The loop either returns at line 6, in which case $A[mid] = x$ and the answer is correct, or it ends because $low > high$. Then the middle region is empty, and by the invariant every element is either less than x or greater than x: x is absent, and returning -1 is correct. Moreover *low* now points at the first element greater than x (or at n), which is the insertion point we noticed in the trace.

The loop does end. The length of the interval, $high - low + 1$, is a non-negative integer, and every iteration that does not return makes it strictly smaller, because *mid* lies inside the interval and is excluded from the next one. A non-negative integer cannot decrease forever. ■

2.5.4 How many probes?

Let the current interval hold m elements. The probe at *mid* splits the other $m-1$ elements into a left part of $\lfloor (m-1)/2 \rfloor$ elements and a right part of $\lceil (m-1)/2 \rceil$ elements, and the search continues in one of them. The larger of the two has $\lceil (m-1)/2 \rceil = \lfloor m/2 \rfloor$ elements. So, in the worst case, each probe replaces m by $\lfloor m/2 \rfloor$. Writing $T(m)$ for the largest number of probes on an interval of m elements,

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

Unfold Equation (2.4) from $m = n$: the sizes are $n, \lfloor n/2 \rfloor, \lfloor n/4 \rfloor$, ..., and the k-th term is $\lfloor n/2^k \rfloor$. The probes continue while this is at least 1, that is while $2^k \le n$, that is for $k = 0,1, \ldots, \lfloor \lg n \rfloor$. That is $\lfloor \lg n \rfloor + 1$ values of k, so

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

> **EXAMPLE.** For $n = 9$, $\lg 9 \approx 3.17$, so $T(9) = 3 + 1 = 4$: no search in the array of **Figure 2.3** needs more than four probes. For the ten-thousand-book catalog of Section 2.10, $\lg 10^4 \approx 13.29$ and $T = 14$. For a million books, $T = 20$; for a billion, $T = 30$. Sequential search on a billion books needs, in the worst case, a billion probes.

That last comparison is the whole case for binary search. Doubling n adds *one* probe to binary search and doubles the cost of sequential search. The average is only a little below the worst case: a successful search on n equally likely keys needs about $\lg n - 1$ probes, because half of all elements sit on the last level of halving [2]. Section 2.10 measures 12.4 probes on 10^4 keys, against $\lg 10^4 - 1 \approx 12.3$.

2.5.5 Binary search, recursively

Binary search has a natural recursive description: to search $A[low..high]$, probe the middle, and if that does not settle it, *search one half* — which is the same problem on a smaller interval. **Pseudocode 2.4** writes this down.

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)
```

The recursive form makes the correctness argument almost automatic: if the procedure is right for all shorter intervals, it is right for this one, because it either answers directly or hands the question to a shorter interval that must contain the key. That is proof by induction, and Chapter 10 is devoted to thinking this way. The cost is one function call per probe, and one stack frame per level of recursion — at most $\lfloor \lg n \rfloor + 2$ frames, which is 32 for a billion elements, so stack space is never a concern here. Because the recursive call is the last thing each call does (a *tail call*), the recursion converts mechanically into the loop of **Pseudocode 2.3**, and many compilers do exactly that.

2.5.6 The programs

Listing 2.7 runs the iterative version with a printed trace for 17 and for 10, the recursive version for five keys, and finally searches the sorted shelf for 088 and 060. **Listing 2.8** and **Listing 2.9** are the same program in Java and Python.

Listing 2.7 `binary_search.cpp`: iterative binary search with a trace, and the recursive form.

```cpp
// Binary search, iterative (traced) and recursive, on numbers and on the shelf.
#include <cstdio>
#include <vector>
using namespace std;

// Closed interval [low, high]. Returns the index or -1; probes and the final low
// (where key would be inserted) come back through the reference parameters.
int binarySearch(const vector<int>& data, int key, int& probes, int& lowOut,
                 bool trace) {
    int low = 0, high = (int)data.size() - 1;
    probes = 0;
    while (low <= high) {  // the interval is not empty
        int mid = low + (high - low) / 2;
        probes++;
        if (data[mid] == key) {
            if (trace)
                printf("  low=%d high=%d mid=%d data[mid]=%d -> found\n", low, high,
                       mid, data[mid]);
            lowOut = low;
            return mid;
        }
        if (data[mid] < key) {
            if (trace)
                printf("  low=%d high=%d mid=%d data[mid]=%d -> %d > %d, "
                       "low = %d\n",
                       low, high, mid, data[mid], key, data[mid], mid + 1);
            // key can only be to the right of mid
            low = mid + 1;
        } else {
            if (trace)
                printf("  low=%d high=%d mid=%d data[mid]=%d -> %d < %d, "
                       "high = %d\n",
                       low, high, mid, data[mid], key, data[mid], mid - 1);
            // key can only be to the left of mid
            high = mid - 1;
        }
    }
    // low is where key would be inserted
    lowOut = low;
    return -1;
}
```

Listing 2.7 (Continued) *binary_search.cpp: iterative binary search with a trace, and the recursive form.*

```
// 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);
}

int main() {
    vector<int> data = {3, 9, 11, 12, 15, 17, 23, 31, 35};
    printf("data:");
    for (int x : data) printf(" %d", x);
    printf("\n");
    int probes, low;
    for (int key : {17, 10}) {
        printf("iterative search for %d\n", key);
        int i = binarySearch(data, key, probes, low, true);
        if (i >= 0)
            printf("  result: index %d after %d probes\n", i, probes);
        else
            printf("  result: not found after %d probes; it would go at index %d\n",
                   probes, low);
    }
    for (int key : {17, 10, 3, 35, 40}) {
        int calls = 0;
        int i = binarySearchRec(data, key, 0, (int)data.size() - 1, calls);
        printf("recursive search for %2d: index %2d, %d calls\n", key, i, calls);
    }

    // the shelf, sorted by code
    vector<int> codes = {25, 34, 41, 56, 63, 72, 88, 96};
    for (int key : {88, 60}) {
        int i = binarySearch(codes, key, probes, low, false);
        printf("shelf code %03d: index %d, %d probes\n", key, i, probes);
    }
    return 0;
}
```

Listing 2.8 `BinarySearch.java`: the same program in Java.

```java
// Binary search, iterative (traced) and recursive, on numbers and on the shelf.
public class BinarySearch {
    // Closed interval [low, high]. Returns {index or -1, probes, low at exit}.
    static int[] binarySearch(int[] data, int key, boolean trace) {
        int low = 0, high = data.length - 1;
        int probes = 0;
        while (low <= high) {  // the interval is not empty
            int mid = low + (high - low) / 2;
            probes++;
            if (data[mid] == key) {
                if (trace)
                    System.out.printf(
                            "  low=%d high=%d mid=%d data[mid]=%d -> found%n", low,
                            high, mid, data[mid]);
                return new int[] {mid, probes, low};
            }
            if (data[mid] < key) {
                if (trace)
                    System.out.printf("  low=%d high=%d mid=%d data[mid]=%d -> " +
                                      "%d > %d, low = %d%n",
                            low, high, mid, data[mid], key, data[mid], mid + 1);
                // key can only be to the right of mid
                low = mid + 1;
            } else {
                if (trace)
                    System.out.printf("  low=%d high=%d mid=%d data[mid]=%d -> " +
                                      "%d < %d, high = %d%n",
                            low, high, mid, data[mid], key, data[mid], mid - 1);
                // key can only be to the left of mid
                high = mid - 1;
            }
        }
        // low is where key would be inserted
        return new int[] {-1, probes, low};
    }
```

Listing 2.8 (Continued) *BinarySearch.java: the same program in Java.*

```
    // Same search, written as a method that calls itself on one half.
    // Returns {index or -1, calls}.
    static int[] binarySearchRec(
            int[] data, int key, int low, int high, int calls) {
        if (low > high) return new int[] {-1, calls};
        int mid = low + (high - low) / 2;
        if (data[mid] == key) return new int[] {mid, calls};
        if (data[mid] < key)
            return binarySearchRec(data, key, mid + 1, high, calls + 1);
        return binarySearchRec(data, key, low, mid - 1, calls + 1);
    }

    public static void main(String[] args) {
        int[] data = {3, 9, 11, 12, 15, 17, 23, 31, 35};
        StringBuilder line = new StringBuilder("data:");
        for (int x : data) line.append(" ").append(x);
        System.out.println(line);
        for (int key : new int[] {17, 10}) {
            System.out.println("iterative search for " + key);
            int[] r = binarySearch(data, key, true);
            if (r[0] >= 0)
                System.out.printf(
                        "  result: index %d after %d probes%n", r[0], r[1]);
            else
                System.out.printf("  result: not found after %d probes; it would " +
                                  "go at index %d%n",
                        r[1], r[2]);
        }
        for (int key : new int[] {17, 10, 3, 35, 40}) {
            int[] r = binarySearchRec(data, key, 0, data.length - 1, 1);
            System.out.printf("recursive search for %2d: index %2d, %d calls%n",
                    key, r[0], r[1]);
        }

        // the shelf, sorted by code
        int[] codes = {25, 34, 41, 56, 63, 72, 88, 96};
        for (int key : new int[] {88, 60}) {
            int[] r = binarySearch(codes, key, false);
            System.out.printf(
                    "shelf code %03d: index %d, %d probes%n", key, r[0], r[1]);
        }
    }
}
```

Listing 2.9 `binary_search.py`: the same program in Python.

```
"""Binary search, iterative (traced) and recursive, on numbers and on the shelf."""


def binary_search(data, key, trace=False):
    """Closed interval [low, high]. Returns (index or -1, probes, low at exit)."""
    low, high = 0, len(data) - 1
    probes = 0
    while low <= high:  # the interval is not empty
        mid = low + (high - low) // 2
        probes += 1
        if data[mid] == key:
            if trace:
                print(
                    "  low=%d high=%d mid=%d data[mid]=%d -> found"
                    % (low, high, mid, data[mid])
                )
            return mid, probes, low
        if data[mid] < key:
            if trace:
                print(
                    "  low=%d high=%d mid=%d data[mid]=%d -> %d > %d, low = %d"
                    % (low, high, mid, data[mid], key, data[mid], mid + 1)
                )
            low = mid + 1  # key can only be to the right of mid
        else:
            if trace:
                print(
                    "  low=%d high=%d mid=%d data[mid]=%d -> %d < %d, high = %d"
                    % (low, high, mid, data[mid], key, data[mid], mid - 1)
                )
            high = mid - 1  # key can only be to the left of mid
    return -1, probes, low  # low is where key would be inserted


def binary_search_rec(data, key, low, high, calls=1):
    """Same search, written as a function that calls itself on one half."""
    if low > high:
        return -1, calls
    mid = low + (high - low) // 2
    if data[mid] == key:
        return mid, calls
    if data[mid] < key:
        return binary_search_rec(data, key, mid + 1, high, calls + 1)
    return binary_search_rec(data, key, low, mid - 1, calls + 1)
```

Listing 2.9 (Continued) *binary_search.py: the same program in Python.*

```
def main():
    data = [3, 9, 11, 12, 15, 17, 23, 31, 35]
    print("data:", " ".join(str(x) for x in data))
    for key in (17, 10):
        print("iterative search for %d" % key)
        i, probes, low = binary_search(data, key, trace=True)
        if i >= 0:
            print("  result: index %d after %d probes" % (i, probes))
        else:
            print(
                "  result: not found after %d probes; it would go at index %d"
                % (probes, low)
            )
    for key in (17, 10, 3, 35, 40):
        i, calls = binary_search_rec(data, key, 0, len(data) - 1)
        print("recursive search for %2d: index %2d, %d calls" % (key, i, calls))

    codes = [25, 34, 41, 56, 63, 72, 88, 96]  # the shelf, sorted by code
    for key in (88, 60):
        i, probes, low = binary_search(codes, key)
        print("shelf code %03d: index %d, %d probes" % (key, i, probes))


if __name__ == "__main__":
    main()
```

The output, identical for all three:

```
data: 3 9 11 12 15 17 23 31 35
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
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
shelf code 088: index 6, 3 probes
shelf code 060: index -1, 3 probes
```

The two traces are the hand traces of Section 2.5.2, line for line. The recursive counts need one remark: the recursive function counts *calls*, and an unsuccessful search makes one final call on an empty interval that probes nothing. So the search for 10 made 3 probes in 4 calls, and the search for 40 made 4 probes — the maximum for $n = 9$, as Equation (2.5) promised — in 5 calls. The search for 35, the last element, also needed the maximum of 4 probes.

> **COMMON TRAP.** Binary search on an *unsorted* array does not crash and does not warn. It simply gives wrong answers. On the shelf in arrival order, [72,25,96,41,88,34,63,56], a search for 25 probes $A[3] = 41$, concludes that 25 must be to the left, probes $A[1] = 25$ and is lucky; a search for 34 probes 41, goes left, probes 25, goes right, probes 96, goes left, and reports that 34 is absent although it is at index 5. Sort first (Chapter 3), or search sequentially.

2.6 Off-by-One Traps, Lower Bound and Upper Bound

2.6.1 Closed and half-open intervals

Pseudocode 2.3 works on a **closed** interval $[low, high]$: both ends belong to it. Many libraries, including the C++ standard library, prefer a **half-open** interval $[low, high)$: low belongs, $high$ does not. Both conventions are correct. What is not correct is mixing them, and almost every binary-search bug is a mixture. **Table 2.2** sets the two side by side.

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	$high - low + 1$	$high - low$

The half-open convention has two small advantages that explain its popularity: the number of elements is simply $high - low$, and the empty array needs no special case ($low = high = 0$).

2.6.2 Four traps

Every one of the following has appeared in published code.

1. **The wrong loop condition.** Closed interval with `while (low < high)`: the loop stops when one element is left without probing it. Searching [5] for 5 fails immediately. Half-open interval with `while (low <= high)`: when $low = high = n$ the loop probes $A[n]$, which is past the end of the array.
2. **An update that does not shrink the interval.** Writing $low \leftarrow mid$ instead of $mid + 1$. When the interval has two elements, $mid = low$, the assignment changes nothing, and the loop runs forever. The maintenance step of our proof fails precisely here: mid was already known to be too small, and keeping it in the interval makes no progress.
3. **An overflowing middle.** Computing $(low + high)/2$ in 32-bit arithmetic. Section 2.6.4 explains.
4. **Ignoring duplicates.** When the key occurs several times, **Pseudocode 2.3** returns *one* of the copies, and which one depends on n and on where the copies sit. Code that assumes it gets the first copy is wrong. The cure is the next subsection.

2.6.3 Lower bound and upper bound

A more useful question than "is x here?" is "where does x belong?". Two functions answer it.

> **DEFINITION.** For a sorted array $A[0..n-1]$ and a key x, **lower bound** is the smallest index i with $A[i] \geq x$, and **upper bound** is the smallest index i with $A[i] > x$; either is n if no such index exists. The copies of x occupy exactly $A[lower..upper-1]$, so the number of copies is $upper - lower$, and x is present if and only if that number is positive. If x is absent, lower bound is the index at which x should be inserted to keep A sorted.

Figure 2.5 shows both on a list of publication years with repeats.

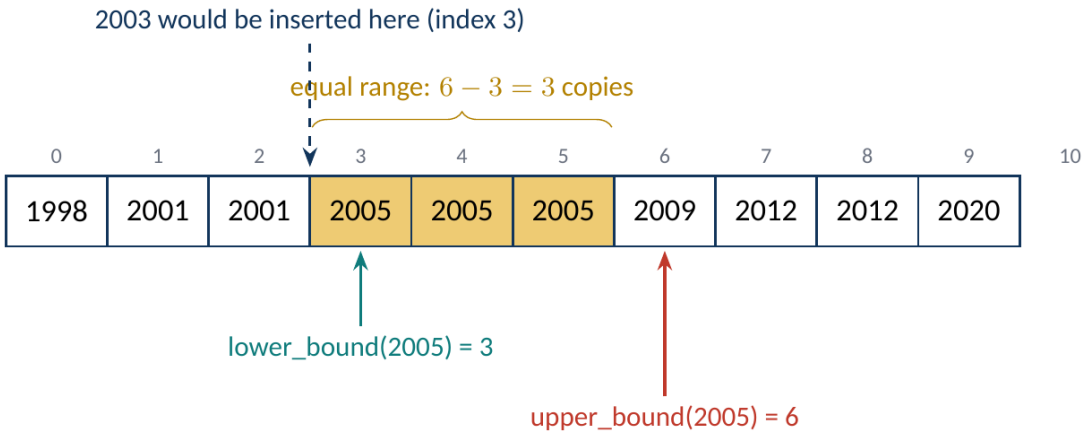


Figure 2.5 Lower bound and upper bound of 2005 in a sorted list of publication years. The three copies of 2005 occupy indices 3 to 5, the half-open range [3,6). For the absent 2003, lower bound and upper bound are both 3, the place where 2003 would be inserted.

Pseudocode 2.5 computes lower bound on a half-open interval. It never tests for equality: it only asks whether $A[mid] < x$, and moves one boundary or the other until the interval is empty.

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)
```

Its invariant is a close relative of the one in **Figure 2.4**: every element of $A[0..low-1]$ is less than x, and every element of $A[high..n-1]$ is at least x. When the loop ends, $low = high$, the unknown region has vanished, and low is the boundary between "less than x" and "at least x" — the lower bound. The interval shrinks by at least half each time, so the loop makes at most $\lceil \lg(n+1) \rceil$ probes, and — unlike **Pseudocode 2.3** — it makes almost exactly that many on every search, because it never stops early. For upper bound, change the test to $A[mid] \le x$, and the boundary moves to the one between "at most x" and "greater than x".

2.6.4 The overflowing middle

In 2006 Joshua Bloch reported that the binary search in Java's own library, and the one in Bentley's *Programming Pearls* from which it was derived, contained a bug that had gone unnoticed for about two decades [4]: the line `int mid = (low + high) / 2;`. If low and $high$ are both large — larger than about 2^{30}, or roughly a billion — their sum exceeds the largest 32-bit `int`, $2^{31} - 1 = 2{,}147{,}483{,}647$, and the sum wraps around to a negative number. Dividing by two leaves it negative, and the program indexes the array with a negative number. Arrays with over a billion elements were rare when the code was written; by 2006 they were not.

The repair is to compute the middle as

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

which is mathematically identical but never forms a number larger than *high*. Every binary search in this book uses Equation (2.6). Python's integers never overflow, so the bug cannot occur there, but the habit costs nothing.

In C++ the overflow is worse than a wrong answer: overflow of a signed integer is *undefined behavior*, and the compiler is allowed to assume it never happens. So **Listing 2.10** does not overflow a real `int`; it computes in 64 bits what a 32-bit `int` would have held. The Java version, **Listing 2.11**, overflows a real `int`, because in Java that is defined behavior: the value simply wraps around. The Python version, **Listing 2.12**, imitates the 32-bit wrap-around with a little arithmetic, since Python's own integers never overflow.

2.6.5 The program

Listing 2.10 implements lower and upper bound, answers five questions about the years of **Figure 2.5**, runs the buggy `low = mid` version with a safety limit to show that it loops forever, and reproduces the overflow.

Listing 2.10 `bounds.cpp`: lower bound, upper bound, a non-terminating binary search, and the overflowing middle.

```cpp
// lower_bound and upper_bound written by hand, and two classic binary-search bugs.
#include <cstdint>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

// 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;
}

// First index i with a[i] > key (a.size() if none).
int upperBound(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;
        else
            high = mid;
    }
    return low;
}

// "low = mid" instead of "low = mid + 1": may never finish. Returns false if
// it is still looping after `limit` steps.
bool buggyLowerBound(const vector<int>& a, int key, int limit, int& low,
                     int& high) {
    low = 0;
    high = (int)a.size();
    int steps = 0;
    while (low < high) {
        if (++steps > limit) return false;
        int mid = (low + high) / 2;
        // BUG: mid itself is already known to be too small
        if (a[mid] < key)
            low = mid;
```

Listing 2.10 (Continued) *bounds.cpp: lower bound, upper bound, a non-terminating binary search, and the overflowing middle.*

```
        else
            high = mid;
    }
    return true;
}

// What a 32-bit signed int would hold after the addition overflows. (Letting a
// real int overflow is undefined behavior in C++, so we compute it in 64 bits.)
int64_t wrap32(int64_t x) {
    const int64_t two31 = INT64_C(1) << 31, two32 = INT64_C(1) << 32;
    return ((x + two31) % two32 + two32) % two32 - two31;
}

int main() {
    vector<int> years = {1998, 2001, 2001, 2005, 2005,
                         2005, 2009, 2012, 2012, 2020};
    printf("years:");
    for (int y : years) printf(" %d", y);
    printf("\nkey   lower upper count  meaning\n");
    for (int key : {2005, 2001, 2003, 1990, 2024}) {
        int lo = lowerBound(years, key), up = upperBound(years, key);
        string meaning = up > lo ? "first at " + to_string(lo) + ", last at " +
                                   to_string(up - 1)
                                 : "absent, insert at " + to_string(lo);
        printf("%d  %5d %5d %5d  %s\n", key, lo, up, up - lo, meaning.c_str());
    }

    int low, high;
    if (!buggyLowerBound(years, 2020, 20, low, high))
        printf("buggy version, key 2020: still looping after 20 steps "
               "(low=%d, high=%d)\n", low, high);

    int64_t l = 1500000000, h = 2000000000;
    int64_t s = wrap32(l + h);
    printf("low + high = %lld, but a 32-bit int holds %lld\n", (long long)(l + h),
           (long long)s);
    printf("(low + high) / 2 in 32 bits = %lld  <- a negative index\n",
           (long long)(s / 2));
    printf("low + (high - low) / 2     = %lld\n", (long long)(l + (h - l) / 2));
    return 0;
}
```

Listing 2.11 `Bounds.java`: the same program in Java, where the overflow happens in a real `int`.

```java
// lower_bound and upper_bound written by hand, and two classic binary-search bugs.
public class Bounds {
    // First index i with a[i] >= key (a.length if none). Half-open [low, high).
    static int lowerBound(int[] a, int key) {
        int low = 0, high = a.length;
        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;
    }

    // First index i with a[i] > key (a.length if none).
    static int upperBound(int[] a, int key) {
        int low = 0, high = a.length;
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (a[mid] <= key)
                low = mid + 1;
            else
                high = mid;
        }
        return low;
    }

    // "low = mid" instead of "low = mid + 1": may never finish.
    // Returns {finished ? 1 : 0, low, high}.
    static int[] buggyLowerBound(int[] a, int key, int limit) {
        int low = 0, high = a.length, steps = 0;
        while (low < high) {
            if (++steps > limit) return new int[] {0, low, high};
            int mid = (low + high) / 2;
            // BUG: mid itself is already known to be too small
            if (a[mid] < key)
                low = mid;
            else
                high = mid;
        }
        return new int[] {1, low, high};
    }
```

Listing 2.11 (Continued) *Bounds.java: the same program in Java, where the overflow happens in a real int.*

```
    public static void main(String[] args) {
        int[] years = {1998, 2001, 2001, 2005, 2005, 2005, 2009, 2012, 2012, 2020};
        StringBuilder line = new StringBuilder("years:");
        for (int y : years) line.append(" ").append(y);
        System.out.println(line);
        System.out.println("key   lower upper count  meaning");
        for (int key : new int[] {2005, 2001, 2003, 1990, 2024}) {
            int lo = lowerBound(years, key), up = upperBound(years, key);
            String meaning = up > lo ? "first at " + lo + ", last at " + (up - 1)
                                     : "absent, insert at " + lo;
            System.out.printf(
                    "%d  %5d %5d %5d  %s%n", key, lo, up, up - lo, meaning);
        }

        int[] b = buggyLowerBound(years, 2020, 20);
        if (b[0] == 0)
            System.out.printf("buggy version, key 2020: still looping after 20 "
                    + "steps (low=%d, high=%d)%n", b[1], b[2]);

        int low = 1500000000, high = 2000000000;
        int sum = low + high;  // Java ints wrap around silently
        System.out.printf("low + high = %d, but a 32-bit int holds %d%n",
                (long) low + high, sum);
        System.out.printf("(low + high) / 2 in 32 bits = %d  <- a negative index%n",
                (low + high) / 2);
        System.out.printf(
                "low + (high - low) / 2     = %d%n", low + (high - low) / 2);
    }
}
```

Listing 2.12 `bounds.py`: the same program in Python.

```python
"""lower_bound and upper_bound by hand, and two classic binary-search bugs."""


def lower_bound(a, key):
    """First index i with a[i] >= key (len(a) if none). Half-open [low, high)."""
    low, high = 0, len(a)
    while low < high:
        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


def upper_bound(a, key):
    """First index i with a[i] > key (len(a) if none)."""
    low, high = 0, len(a)
    while low < high:
        mid = low + (high - low) // 2
        if a[mid] <= key:
            low = mid + 1
        else:
            high = mid
    return low


def buggy_lower_bound(a, key, limit):
    """'low = mid' instead of 'low = mid + 1': may never finish."""
    low, high = 0, len(a)
    steps = 0
    while low < high:
        steps += 1
        if steps > limit:
            return None, low, high
        mid = (low + high) // 2
        if a[mid] < key:
            low = mid  # BUG: mid itself is already known to be too small
        else:
            high = mid
    return low, low, high
```

Listing 2.12 (Continued) *bounds.py: the same program in Python.*

```
def wrap32(x):
    """What a 32-bit signed int would hold after the addition overflows."""
    return (x + 2**31) % 2**32 - 2**31


def main():
    years = [1998, 2001, 2001, 2005, 2005, 2005, 2009, 2012, 2012, 2020]
    print("years:", " ".join(str(y) for y in years))
    print("key   lower upper count  meaning")
    for key in (2005, 2001, 2003, 1990, 2024):
        lo, up = lower_bound(years, key), upper_bound(years, key)
        if up > lo:
            meaning = "first at %d, last at %d" % (lo, up - 1)
        else:
            meaning = "absent, insert at %d" % lo
        print("%d  %5d %5d %5d  %s" % (key, lo, up, up - lo, meaning))

    result, low, high = buggy_lower_bound(years, 2020, 20)
    if result is None:
        print(
            "buggy version, key 2020: still looping after 20 steps "
            "(low=%d, high=%d)" % (low, high)
        )

    low, high = 1500000000, 2000000000
    s = wrap32(low + high)
    print("low + high = %d, but a 32-bit int holds %d" % (low + high, s))
    print("(low + high) / 2 in 32 bits = %d  <- a negative index" % int(s / 2))
    print("low + (high - low) / 2     = %d" % (low + (high - low) // 2))


if __name__ == "__main__":
    main()
```

All three print:

```
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
buggy version, key 2020: still looping after 20 steps (low=8, high=9)
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
```

Check the table against **Figure 2.5**. For 2005 the copies are at 3, 4 and 5. For 2003 both bounds are 3, so the count is 0 and the insertion point is 3. For 1990, smaller than everything, both bounds are 0; for 2024, larger than everything, both are $n = 10$. The buggy version reached $low = 8$, $high = 9$ and could go no further: $mid = 8$, $A[8] = 2012 < 2020$, and $low \leftarrow 8$ changes nothing. The last three lines are Bloch's bug in miniature: two perfectly valid indices whose average, computed the naive way, is $-397{,}483{,}648$.

> **KEY IDEA.** Lower bound and upper bound are the binary searches you should reach for first. They answer "is it here?" (is $upper > lower$?), "where is the first copy?" ($lower$), "where is the last?" ($upper - 1$), "how many are there?" ($upper - lower$), "where would it go?" ($lower$), and "how many are in the range $[a, b)$?" ($lower(b) - lower(a)$) — all with at most $\lceil \lg(n + 1) \rceil$ probes each.

2.7 Interpolation Search

2.7.1 Guessing where the key should be

Binary search always probes the middle. But if you are looking for the name *Tanaka* in a thick paper telephone directory, you do not open it in the middle; you open it about three quarters of the way through, because you know that T is late in the alphabet. You are using not only the *order* of the data but its *values*. **Interpolation search** does the same: it estimates where the key ought to be, assuming the keys rise steadily from $A[low]$ to $A[high]$, and probes there. The idea goes back to Peterson in 1957 [5].

If the keys rise in a straight line from $A[low]$ at position low to $A[high]$ at position $high$, the key x lies at a fraction $(x - A[low])/(A[high] - A[low])$ of the way along, and the estimated position is

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

When $x = A[low]$ the fraction is 0 and $pos = low$; when $x = A[high]$ it is 1 and $pos = high$; in between, the probe moves in proportion. Binary search is the special case that ignores the values and always uses the fraction $\frac{1}{2}$.

2.7.2 The shelf, by hand

Figure 2.6 plots the eight codes of **Table 2.1** against their indices. They lie close to the dashed straight line from (0,025) to (7,096), which is exactly the assumption Equation (2.7) makes.

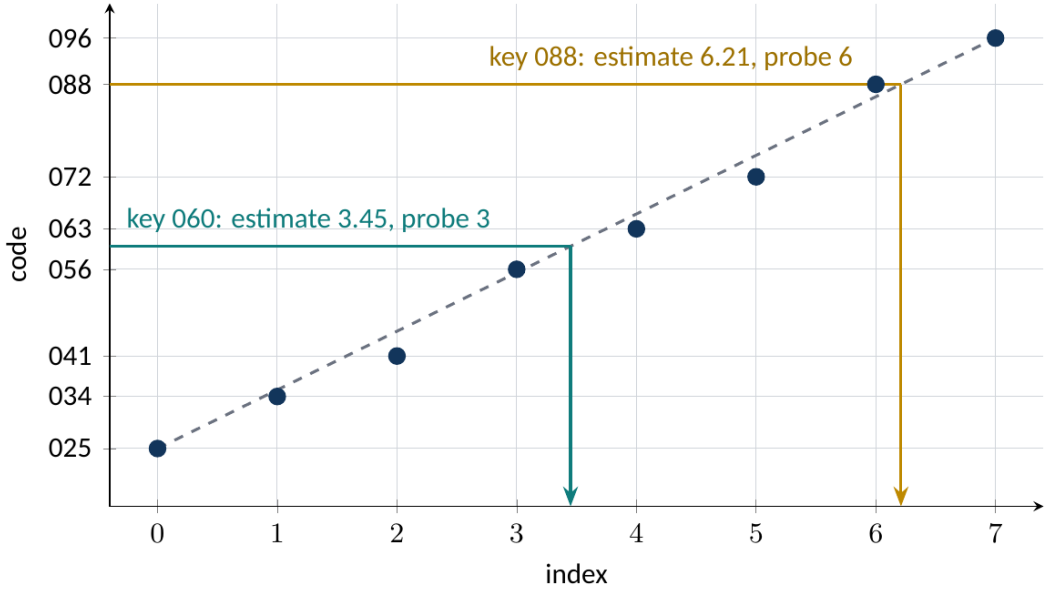


Figure 2.6 Interpolation search on the computing shelf. The dashed line runs straight from code[0] = 025 to code[7] = 096. For the key 088 it estimates position 6.21, and the probe at index 6 finds 088 at once; for the key 060 it estimates 3.45 and probes index 3.

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

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

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

Key 060. With $low = 0$, $high = 7$: $pos = \lfloor (35/71) \cdot 7 \rfloor = \lfloor 3.45 \rfloor = 3$, and $A[3] = 056 < 060$, so $low \leftarrow 4$. Now the key 060 is *smaller* than $A[low] = A[4] = 063$. Since the interval $A[4..7]$ holds only codes of at least 063, the key cannot be there, and the search stops: 060 is absent, after one probe.

That last step shows a guard that interpolation search needs and binary search does not. Equation (2.7) is only meaningful when $A[low] \le x \le A[high]$; outside that range the fraction is negative or greater than one, and the "estimate" points outside the interval — possibly outside the array. So the loop condition checks the range, and a key outside it is reported absent at once.

A second guard is needed when $A[low] = A[high]$: then Equation (2.7) divides by zero. If the range check has passed, every element of the interval equals x, and probing $A[low]$ settles the matter.

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
```

2.7.3 The program

Listing 2.13 searches the shelf for five codes and prints every estimate. It computes the fraction in floating point, as the formula does, and converts to an index by truncation, which for a non-negative number is the floor. **Listing 2.14** and **Listing 2.15** do the same in Java and Python.

Listing 2.13 `interpolation_search.cpp`: interpolation search on the computing shelf, printing each estimate.

```
// Interpolation search on Pustaka Ceria's computing shelf, sorted by code.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

const vector<int> CODES = {25, 34, 41, 56, 63, 72, 88, 96};
const vector<string> TITLES = {
        "The C++ Programming",       "Mastering Delphi 6",
        "Professional C#",           "Pure JavaScript v2",
        "Advanced JSP & Servlet",    "Calculus Make it Easy",
        "Visual Basic 2005 Express", "Artificial Life : Volume 1"};

// Returns the index or -1, printing each estimate it makes.
int interpolationSearch(const vector<int>& data, int key, int& probes) {
    int low = 0, high = (int)data.size() - 1;
    probes = 0;
    while (low <= high && data[low] <= key && key <= data[high]) {
        double estimate;
        if (data[high] == data[low])  // all equal: no slope to follow
            estimate = 0.0;
        else
            estimate = (double)(key - data[low]) /
                       (double)(data[high] - data[low]) * (high - low);
        // the cast truncates, like floor here
        int pos = low + (int)estimate;
        probes++;
        printf("  low=%d high=%d: position = %d + %.2f -> %d, code[%d] = %03d\n",
               low, high, low, estimate, pos, pos, data[pos]);
        if (data[pos] == key) return pos;
        if (data[pos] < key)
            low = pos + 1;
        else
            high = pos - 1;
    }
    if (low > high)
        printf("  low=%d > high=%d: the range is empty\n", low, high);
    else
        printf("  key %03d lies outside code[%d..%d] = %03d..%03d\n", key, low,
               high, data[low], data[high]);
    return -1;
}
```

Listing 2.13 (Continued) *interpolation_search.cpp: interpolation search on the computing shelf, printing each estimate.*

```
int main() {
    for (size_t i = 0; i < CODES.size(); i++)
        printf("[%zu] %03d %s\n", i, CODES[i], TITLES[i].c_str());
    for (int key : {88, 60, 72, 96, 30}) {
        printf("search %03d\n", key);
        int probes;
        int i = interpolationSearch(CODES, key, probes);
        if (i >= 0)
            printf("  found: %s (%d probe%s)\n", TITLES[i].c_str(), probes,
                   probes == 1 ? "" : "s");
        else
            printf("  not found (%d probe%s)\n", probes, probes == 1 ? "" : "s");
    }
    return 0;
}
```

Listing 2.14 `InterpolationSearch.java`: the same program in Java. `Locale.ROOT` makes the decimal point a point on every computer.

```
// Interpolation search on Pustaka Ceria's computing shelf, sorted by code.
import java.util.Locale;

public class InterpolationSearch {
    static final int[] CODES = {25, 34, 41, 56, 63, 72, 88, 96};
    static final String[] TITLES = {"The C++ Programming", "Mastering Delphi 6",
            "Professional C#", "Pure JavaScript v2", "Advanced JSP & Servlet",
            "Calculus Make it Easy", "Visual Basic 2005 Express",
            "Artificial Life : Volume 1"};

    // Returns {index or -1, probes}, printing each estimate it makes.
    static int[] interpolationSearch(int[] data, int key) {
        int low = 0, high = data.length - 1;
        int probes = 0;
        while (low <= high && data[low] <= key && key <= data[high]) {
            double estimate;
            // all equal: no slope to follow
            if (data[high] == data[low])
                estimate = 0.0;
            else
                estimate = (double) (key - data[low]) /
                        (double) (data[high] - data[low]) * (high - low);
            // the cast truncates, like floor here
            int pos = low + (int) estimate;
            probes++;
            System.out.printf(Locale.ROOT,
                    "  low=%d high=%d: position = %d + %.2f -> %d, " +
                    "code[%d] = %03d%n",
                    low, high, low, estimate, pos, pos, data[pos]);
            if (data[pos] == key) return new int[] {pos, probes};
            if (data[pos] < key)
                low = pos + 1;
            else
                high = pos - 1;
        }
        if (low > high)
            System.out.printf(
                    "  low=%d > high=%d: the range is empty%n", low, high);
        else
            System.out.printf("  key %03d lies outside code[%d..%d] = %03d..%03d%n",
                    key, low, high, data[low], data[high]);
        return new int[] {-1, probes};
    }
```

Listing 2.14 (Continued) *InterpolationSearch.java: the same program in Java. Locale.ROOT makes the decimal point a point on every computer.*

```
    public static void main(String[] args) {
        for (int i = 0; i < CODES.length; i++)
            System.out.printf("[%d] %03d %s%n", i, CODES[i], TITLES[i]);
        for (int key : new int[] {88, 60, 72, 96, 30}) {
            System.out.printf("search %03d%n", key);
            int[] r = interpolationSearch(CODES, key);
            String s = r[1] == 1 ? "" : "s";
            if (r[0] >= 0)
                System.out.printf(
                        "  found: %s (%d probe%s)%n", TITLES[r[0]], r[1], s);
            else
                System.out.printf("  not found (%d probe%s)%n", r[1], s);
        }
    }
}
```

Listing 2.15 `interpolation_search.py`: the same program in Python.

```
"""Interpolation search on Pustaka Ceria's computing shelf, sorted by code."""

CODES = [25, 34, 41, 56, 63, 72, 88, 96]
TITLES = [
    "The C++ Programming",
    "Mastering Delphi 6",
    "Professional C#",
    "Pure JavaScript v2",
    "Advanced JSP & Servlet",
    "Calculus Make it Easy",
    "Visual Basic 2005 Express",
    "Artificial Life : Volume 1",
]


def interpolation_search(data, key):
    """Returns (index or -1, probes), printing each estimate it makes."""
    low, high = 0, len(data) - 1
    probes = 0
    while low <= high and data[low] <= key <= data[high]:
        if data[high] == data[low]:  # all equal: no slope to follow
            estimate = 0.0
        else:
            estimate = (
                float(key - data[low])
                / float(data[high] - data[low])
                * (high - low)
            )
        pos = low + int(estimate)  # int() truncates, like floor here
        probes += 1
        print(
            "  low=%d high=%d: position = %d + %.2f -> %d, code[%d] = %03d"
            % (low, high, low, estimate, pos, pos, data[pos])
        )
        if data[pos] == key:
            return pos, probes
        if data[pos] < key:
            low = pos + 1
        else:
            high = pos - 1
    if low > high:
        print("  low=%d > high=%d: the range is empty" % (low, high))
    else:
        print(
            "  key %03d lies outside code[%d..%d] = %03d..%03d"
            % (key, low, high, data[low], data[high])
```

Listing 2.15 (Continued) *interpolation_search.py: the same program in Python.*

```
        )
    return -1, probes


def main():
    for i in range(len(CODES)):
        print("[%d] %03d %s" % (i, CODES[i], TITLES[i]))
    for key in (88, 60, 72, 96, 30):
        print("search %03d" % key)
        i, probes = interpolation_search(CODES, key)
        if i >= 0:
            print(
                "  found: %s (%d probe%s)"
                % (TITLES[i], probes, "" if probes == 1 else "s")
            )
        else:
            print("  not found (%d probe%s)" % (probes, "" if probes == 1 else "s"))


if __name__ == "__main__":
    main()
```

All three print:

```
[0] 025 The C++ Programming
[1] 034 Mastering Delphi 6
[2] 041 Professional C#
[3] 056 Pure JavaScript v2
[4] 063 Advanced JSP & Servlet
[5] 072 Calculus Make it Easy
[6] 088 Visual Basic 2005 Express
[7] 096 Artificial Life : Volume 1
search 088
  low=0 high=7: position = 0 + 6.21 -> 6, code[6] = 088
  found: Visual Basic 2005 Express (1 probe)
search 060
  low=0 high=7: position = 0 + 3.45 -> 3, code[3] = 056
  key 060 lies outside code[4..7] = 063..096
  not found (1 probe)
search 072
  low=0 high=7: position = 0 + 4.63 -> 4, code[4] = 063
  low=5 high=7: position = 5 + 0.00 -> 5, code[5] = 072
  found: Calculus Make it Easy (2 probes)
search 096
  low=0 high=7: position = 0 + 7.00 -> 7, code[7] = 096
  found: Artificial Life : Volume 1 (1 probe)
search 030
  low=0 high=7: position = 0 + 0.49 -> 0, code[0] = 025
  key 030 lies outside code[1..7] = 034..096
  not found (1 probe)
```

The traces for 088 and 060 are the hand computations above. The key 072 needed two probes: the first estimate, 4.63, landed on 063, and the second, on the interval $A[5..7]$, landed exactly on 072 because the key equals $A[low]$. The key 030 was estimated at 0.49, probed 025, and was then smaller than $A[1] = 034$: absent after one probe.

2.7.4 Brilliant or terrible: it depends on the data

When the keys are spread uniformly at random, the analysis of Perl, Itai and Avni shows that interpolation search makes on average about $\lg\lg n$ probes [6]. The function $\lg\lg n$ grows so slowly that it is almost a constant: for $n = 10^6$ it is about 4.3, and for $n = 10^{18}$ only about 6. Section 2.10 measures 4.26 probes on a million uniformly spread keys, against 18.8 for binary search.

But the estimate is only as good as the straight-line assumption. When keys are bunched — many small values and a few enormous ones — the line through $A[low]$ and $A[high]$ is a poor guide, the estimate lands far from the key, and each probe moves the interval boundary by only a few places. In the worst case interpolation search degrades to $O(n)$ probes, worse than binary search and no better than sequential search. Section 2.10 shows this on real data: on the identifiers of a real book catalog, which are heavily bunched, interpolation search needed 211 probes on average where binary search needed 12.

> **WARNING.** Interpolation search needs keys on which subtraction and division make sense — numbers, or strings you have deliberately encoded as numbers. For codes like `B1001` or titles, use binary search. And every step costs a division, which on current processors is several times slower than a comparison, so even when interpolation search makes fewer probes it is not always faster.

2.8 Jump Search

2.8.1 Leaping ahead in blocks

Some storage can only be read efficiently in one direction — a magnetic tape, a file read in large blocks, a list that can be skipped forwards but not backwards. On such storage binary search, which jumps back and forth, is awkward, but sequential search is slow. **Jump search**, analyzed by Shneiderman in 1978 [7], is the compromise. On a sorted array it jumps forward m places at a time, looking only at the last element of each block of m, until it reaches a block whose last element is at least the key. The key, if present, must be in that block, and a short sequential walk through the block finds it. **Pseudocode 2.7** gives the procedure.

Figure 2.7 shows jump search for 52 in sixteen sorted call numbers with blocks of $m = 4$.

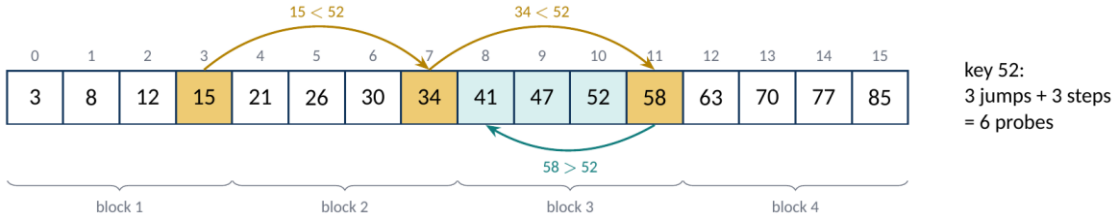


Figure 2.7 Jump search for 52 with blocks of four. The jumps probe the last element of each block (15, 34, 58); since $58 > 52$ the key must be in block 3, and a walk through 41, 47, 52 finds it.

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
```

The programs below refine lines 3 and 8 slightly: when the last element of a block *equals* the key the search stops at once, and the walk then skips that last element, which has already been checked. This makes the probe count match the analysis exactly.

2.8.2 Choosing the block size

In the worst case jump search makes about n/m jumps (one per block) and then walks through $m-1$ elements of the last block. Its worst-case cost is therefore

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

Large blocks mean few jumps and a long walk; small blocks mean many jumps and a short walk. To find the best balance, treat m as a real number and set the derivative of Equation (2.8) to zero: $f'(m) = -n/m^2 + 1 = 0$ gives $m^2 = n$, that is

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

The second derivative, $2n/m^3$, is positive, so this is a minimum. You can also see it without calculus: the product of the two terms n/m and m is always n, and two positive numbers with a fixed product have the smallest sum when they are equal. With $m = \lfloor\sqrt{n}\rfloor$, jump search needs at most about $2\sqrt{n}$ probes — 199 for $n = 10^4$ and 1,999 for $n = 10^6$. That is far better than sequential search's n but far worse than binary search's $\lfloor \lg n \rfloor + 1$; jump search earns its place only where backward jumps are expensive.

2.8.3 The program

Listing 2.16 `jump_search.cpp`: jump search on sixteen call numbers, and the worst case for several n.

```cpp
// Jump search: leap ahead m places at a time, then walk back through one block.
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

// Returns the index or -1; jump probes and walk probes come back separately.
int jumpSearch(const vector<int>& data, int key, int m, int& jumps, int& walks) {
    int n = (int)data.size();
    jumps = walks = 0;
    // the current block is data[start..end-1]
    int start = 0, end = min(m, n);
    while (true) {
        jumps++;
        // the block's last element is the key
        if (data[end - 1] == key) return end - 1;
        // the key, if present, is in this block
        if (data[end - 1] > key) break;
        start = end;
        end = min(end + m, n);
        // jumped off the end: key > every element
        if (start >= n) return -1;
    }
    for (int i = start; i < end - 1; i++) {  // data[end-1] was already checked
        walks++;
        if (data[i] == key) return i;
        if (data[i] > key) break;  // passed the place where key would be
    }
    return -1;
}

int isqrt(int n) {  // largest m with m * m <= n
    int m = (int)sqrt((double)n);
    while (m * m > n) m--;
    while ((m + 1) * (m + 1) <= n) m++;
    return m;
}
```

Listing 2.16 (Continued) *jump_search.cpp: jump search on sixteen call numbers, and the worst case for several n.*

```cpp
int main() {
    vector<int> calls = {3,  8,  12, 15, 21, 26, 30, 34,
                         41, 47, 52, 58, 63, 70, 77, 85};
    int m = isqrt((int)calls.size());
    printf("call numbers:");
    for (int x : calls) printf(" %d", x);
    printf("\nn = %zu, block size m = %d\n", calls.size(), m);
    for (int key : {52, 58, 60, 3, 90}) {
        int j, w;
        int i = jumpSearch(calls, key, m, j, w);
        string where = i >= 0 ? "index " + to_string(i) : "not found";
        printf("key %2d: %-9s  jump %d + walk %d = %d probes\n", key, where.c_str(),
               j, w, j + w);
    }
    printf("worst case n/m + m - 1 for the best m = floor(sqrt(n)):\n");
    for (int n : {16, 100, 10000, 1000000}) {
        int best = isqrt(n);
        printf("  n = %7d  m = %4d  worst = %4d probes\n", n, best,
               n / best + best - 1);
    }
    return 0;
}
```

Listing 2.17 `JumpSearch.java`: the same program in Java.

```
// Jump search: leap ahead m places at a time, then walk back through one block.
public class JumpSearch {
    // Returns {index or -1, jump probes, walk probes}.
    static int[] jumpSearch(int[] data, int key, int m) {
        int n = data.length;
        int jumps = 0, walks = 0;
        // the current block is data[start..end-1]
        int start = 0, end = Math.min(m, n);
        while (true) {
            jumps++;
            if (data[end - 1] == key)
                return new int[] {
                        end - 1, jumps, walks};  // last element is the key
            // the key, if present, is in this block
            if (data[end - 1] > key) break;
            start = end;
            end = Math.min(end + m, n);
            // key > every element
            if (start >= n) return new int[] {-1, jumps, walks};
        }
        // data[end-1] was already checked
        for (int i = start; i < end - 1; i++) {
            walks++;
            if (data[i] == key) return new int[] {i, jumps, walks};
            // passed the place where key would be
            if (data[i] > key) break;
        }
        return new int[] {-1, jumps, walks};
    }

    static int isqrt(int n) {  // largest m with m * m <= n
        int m = (int) Math.sqrt(n);
        while (m * m > n) m--;
        while ((m + 1) * (m + 1) <= n) m++;
        return m;
    }
```

Listing 2.17 (Continued) *JumpSearch.java: the same program in Java.*

```
    public static void main(String[] args) {
        int[] calls = {
                3, 8, 12, 15, 21, 26, 30, 34, 41, 47, 52, 58, 63, 70, 77, 85};
        int m = isqrt(calls.length);
        StringBuilder line = new StringBuilder("call numbers:");
        for (int x : calls) line.append(" ").append(x);
        System.out.println(line);
        System.out.printf("n = %d, block size m = %d%n", calls.length, m);
        for (int key : new int[] {52, 58, 60, 3, 90}) {
            int[] r = jumpSearch(calls, key, m);
            String where = r[0] >= 0 ? "index " + r[0] : "not found";
            System.out.printf("key %2d: %-9s  jump %d + walk %d = %d probes%n", key,
                    where, r[1], r[2], r[1] + r[2]);
        }
        System.out.println(
                "worst case n/m + m - 1 for the best m = floor(sqrt(n)):");
        for (int n : new int[] {16, 100, 10000, 1000000}) {
            int best = isqrt(n);
            System.out.printf("  n = %7d  m = %4d  worst = %4d probes%n", n, best,
                    n / best + best - 1);
        }
    }
}
```

Listing 2.18 `jump_search.py`: the same program in Python.

```
"""Jump search: leap ahead m places at a time, then walk back through one block."""

import math


def jump_search(data, key, m):
    """Returns (index or -1, jump probes, walk probes)."""
    n = len(data)
    jumps = walks = 0
    start, end = 0, min(m, n)  # the current block is data[start..end-1]
    while True:
        jumps += 1
        if data[end - 1] == key:  # the block's last element is the key
            return end - 1, jumps, walks
        if data[end - 1] > key:  # the key, if present, is in this block
            break
        start, end = end, min(end + m, n)
        if start >= n:  # jumped off the end: key > every element
            return -1, jumps, walks
    for i in range(start, end - 1):  # data[end-1] was already checked
        walks += 1
        if data[i] == key:
            return i, jumps, walks
        if data[i] > key:  # passed the place where key would be
            break
    return -1, jumps, walks


def main():
    calls = [3, 8, 12, 15, 21, 26, 30, 34, 41, 47, 52, 58, 63, 70, 77, 85]
    m = math.isqrt(len(calls))
    print("call numbers:", " ".join(str(x) for x in calls))
    print("n = %d, block size m = %d" % (len(calls), m))
    for key in (52, 58, 60, 3, 90):
        i, j, w = jump_search(calls, key, m)
        where = "index %d" % i if i >= 0 else "not found"
        print(
            "key %2d: %-9s  jump %d + walk %d = %d probes"
            % (key, where, j, w, j + w)
        )
    print("worst case n/m + m - 1 for the best m = floor(sqrt(n)):")
    for n in (16, 100, 10000, 1000000):
        m = math.isqrt(n)
        print("  n = %7d  m = %4d  worst = %4d probes" % (n, m, n // m + m - 1))


```

Listing 2.18 (Continued) *jump_search.py: the same program in Python.*

```
if __name__ == "__main__":
    main()
```

Listing 2.16, **Listing 2.17** and **Listing 2.18** all print:

```
call numbers: 3 8 12 15 21 26 30 34 41 47 52 58 63 70 77 85
n = 16, block size m = 4
key 52: index 10   jump 3 + walk 3 = 6 probes
key 58: index 11   jump 3 + walk 0 = 3 probes
key 60: not found  jump 4 + walk 1 = 5 probes
key  3: index 0    jump 1 + walk 1 = 2 probes
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
  n = 1000000  m = 1000  worst = 1999 probes
```

The search for 52 is **Figure 2.7**: three jumps, three steps. The search for 58 stopped on its third jump, because 58 is the last element of block 3. The search for 60 jumped four times, stepped once onto 63, and stopped because 63 is already too large. The search for 90 jumped past the end of the array. The table at the bottom evaluates Equation (2.9).

2.9 Searching Two-Dimensional Arrays

2.9.1 How a two-dimensional array sits in memory

The shelves of Pustaka Ceria form a grid: three shelves, five slots each. In C++ that is `int B[3][5]`, a two-dimensional array with 3 **rows** and 5 **columns**. Memory, however, is one-dimensional: a single long line of bytes. C and C++ store a two-dimensional array in **row-major order**: all of row 0, then all of row 1, then all of row 2, as **Figure 2.8** shows.

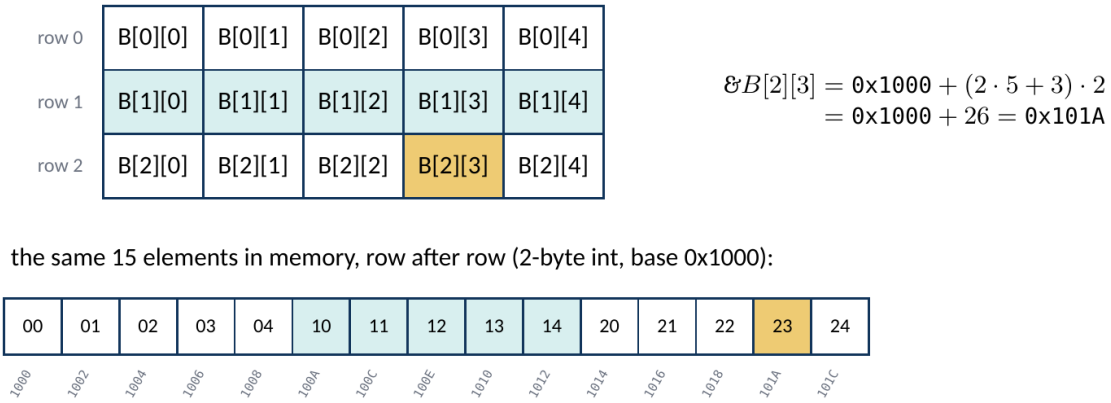


Figure 2.8 A 3 × 5 array and its row-major layout in memory. Each memory cell is labeled with the row and column of the element it holds; the small numbers are the addresses when an `int` occupies 2 bytes and the array starts at 0x1000.

To reach $B[i][j]$ the machine must skip i complete rows of C elements each, then j more elements. So the element is number $i \cdot C + j$ in the line, and

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

where C is the number of columns and w the width of one element. Equation (2.10) generalizes Equation (2.1): a two-dimensional index is turned into a one-dimensional one and then treated exactly as before. Fortran, MATLAB and R use the other convention, **column-major** order, in which the roles of i and j swap: $base + (j \cdot R + i) \cdot w$, with R the number of rows.

> **EXAMPLE.** Let `int A[10]` start at address 0x1000 on a compiler whose `int` occupies 2 bytes (as on the 16-bit compilers of the 1990s). Then $\&A[7] = \texttt{0x1000} + 7 \cdot 2 = \texttt{0x1000} + 14$. Fourteen is E in hexadecimal, so $\&A[7] = \texttt{0x100E}$. Now let `int B[3][5]` start at 0x1000. For

> $B[2][3]$: skip 2 rows of 5, then 3: element number $2 \cdot 5 + 3 = 13$, at byte offset $13 \cdot 2 = 26 = 0\text{x}1\text{A}$, so $\&B[2][3] = 0\text{x}101\text{A}$. On today's compilers an `int` has 4 bytes, and the same two addresses are $0\text{x}1000 + 28 = 0\text{x}101\text{C}$ and $0\text{x}1000 + 52 = 0\text{x}1034$. A common slip is to add the element *count* instead of the byte offset — writing $0\text{x}1000 + 7$ — which forgets the factor w.

2.9.2 Three ways to search a grid

Unsorted grid: scan. If the grid has no order, there is nothing to exploit: visit the rows one by one and the slots of each row one by one. This is sequential search on the $R \cdot C$ elements, $O(RC)$ probes. Scanning row by row — the inner loop over the columns — visits memory in the order it is laid out in **Figure 2.8**, which is also the fastest order for the processor's cache.

Rows and columns both sorted: the staircase. Suppose every row increases from left to right *and* every column increases from top to bottom — the layout of a table in which both the shelf number and the slot number follow the code. Such a grid is not sorted as one line, so binary search does not apply directly, but a beautiful $O(R + C)$ method does. Start at the **top-right** corner. If the element there equals the key, stop. If it is *larger* than the key, the key cannot be anywhere below it in that column (everything below is larger still), so move one step left. If it is *smaller*, the key cannot be anywhere to its left in that row (everything to the left is smaller still), so move one step down. Each step discards a whole row or a whole column, so after at most $R + C - 1$ steps the search either finds the key or walks off the grid.

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)
```

Figure 2.9 traces it for a key that is present and a key that is absent.

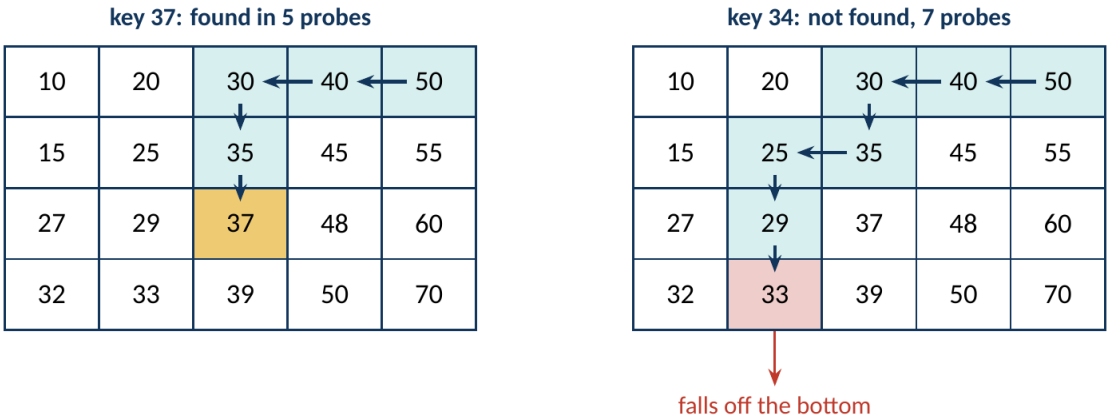


Figure 2.9 The staircase search on a 4×5 grid whose rows and columns ascend. For 37 the path 50, 40, 30, 35, 37 finds the key in five probes; for 34 the path steps left and down until it leaves the grid below 33.

Whole grid sorted in row-major order: binary search. If the last element of each row is smaller than the first element of the next, the grid, read in row-major order, is a single sorted line of $R \cdot C$ elements. Binary search it as one array of indices $0..RC-1$, converting each index k back to a row and a column with $r = \lfloor k/C \rfloor$ and $c = k \bmod C$ — Equation (2.10) run backwards. That costs $\lfloor \lg(RC) \rfloor + 1$ probes.

2.9.3 The program

Listing 2.19 computes the addresses of the example above, scans an unsorted grid of shelf codes, runs the staircase for 37 and 34, and binary-searches a row-major sorted grid. **Listing 2.20** and **Listing 2.21** are its Java and Python versions.

Listing 2.19 `search_2d.cpp`: addresses in a two-dimensional array, a scan, the staircase, and binary search in row-major order.

```
// Searching two-dimensional arrays: addresses, a plain scan, the staircase,
// and binary search over a matrix that is sorted in row-major order.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

using Matrix = vector<vector<int>>;

long address(long base, int width, int index) {
    return base + (long)index * width;
}

// Row by row, slot by slot: works on any grid.
bool scan(const Matrix& grid, int key, int& row, int& col, int& probes) {
    probes = 0;
    for (size_t r = 0; r < grid.size(); r++)
        for (size_t c = 0; c < grid[r].size(); c++) {
            probes++;
            if (grid[r][c] == key) {
                row = (int)r;
                col = (int)c;
                return true;
            }
        }
    return false;
}

// Rows and columns both ascending: start top-right, move left or down.
bool staircase(const Matrix& m, int key, int& row, int& col, vector<int>& path) {
    int r = 0, c = (int)m[0].size() - 1;
    while (r < (int)m.size() && c >= 0) {
        path.push_back(m[r][c]);
        if (m[r][c] == key) {
            row = r;
            col = c;
            return true;
        }
        if (m[r][c] > key)
            c--;  // everything below in this column is even larger
        else
            r++;  // everything to the left in this row is even smaller
    }
    return false;
}
```

Listing 2.19 (Continued) *search_2d.cpp: addresses in a two-dimensional array, a scan, the staircase, and binary search in row-major order.*

```
// The whole matrix is one sorted run in row-major order.
bool binary2d(const Matrix& m, int key, int& row, int& col, int& probes) {
    int rows = (int)m.size(), cols = (int)m[0].size();
    int low = 0, high = rows * cols - 1;
    probes = 0;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        int r = mid / cols, c = mid % cols;  // index k -> (row, column)
        probes++;
        if (m[r][c] == key) {
            row = r;
            col = c;
            return true;
        }
        if (m[r][c] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return false;
}

int main() {
    long base = 0x1000;
    for (int width : {2, 4}) {
        long a7 = address(base, width, 7);
        long b23 = address(base, width, 2 * 5 + 3);
        printf("int of %d bytes: &A[7] = 0x%lX, &B[2][3] = 0x%lX\n", width, a7,
               b23);
    }

    Matrix shelves = {
            {72, 25, 96, 41, 88}, {34, 17, 63, 56, 12}, {80, 45, 29, 91, 7}};
    int r, c, p;
    for (int key : {63, 50}) {
        if (scan(shelves, key, r, c, p))
            printf("scan for %d: shelf %d, slot %d after %d probes\n", key, r, c,
                   p);
        else
            printf("scan for %d: not found after %d probes\n", key, p);
    }
```

Listing 2.19 (Continued) *search_2d.cpp: addresses in a two-dimensional array, a scan, the staircase, and binary search in row-major order.*

```
    Matrix m = {{10, 20, 30, 40, 50},
                {15, 25, 35, 45, 55},
                {27, 29, 37, 48, 60},
                {32, 33, 39, 50, 70}};
    for (int key : {37, 34}) {
        vector<int> path;
        bool found = staircase(m, key, r, c, path);
        string where = found ? "at (" + to_string(r) + ", " + to_string(c) + ")"
                             : "not found";
        string steps;
        for (size_t k = 0; k < path.size(); k++)
            steps += (k ? " " : "") + to_string(path[k]);
        printf("staircase for %d: %s, path %s (%zu probes)\n", key, where.c_str(),
               steps.c_str(), path.size());
    }

    Matrix s = {{2, 5, 9, 14}, {17, 23, 31, 38}, {42, 47, 55, 61}};
    for (int key : {31, 40}) {
        bool found = binary2d(s, key, r, c, p);
        string where = found ? "at (" + to_string(r) + ", " + to_string(c) + ")"
                             : "not found";
        printf("row-major binary search for %d: %s after %d probes\n", key,
               where.c_str(), p);
    }
    return 0;
}
```

Listing 2.20 `Search2D.java`: the same program in Java.

```java
// Searching two-dimensional arrays: addresses, a plain scan, the staircase,
// and binary search over a matrix that is sorted in row-major order.
import java.util.ArrayList;
import java.util.List;
import java.util.StringJoiner;

public class Search2D {
    static long address(long base, int width, int index) {
        return base + (long) index * width;
    }

    // Row by row, slot by slot: works on any grid. Returns {row, col, probes}.
    static int[] scan(int[][] grid, int key) {
        int probes = 0;
        for (int r = 0; r < grid.length; r++)
            for (int c = 0; c < grid[r].length; c++) {
                probes++;
                if (grid[r][c] == key) return new int[] {r, c, probes};
            }
        return new int[] {-1, -1, probes};
    }

    // Rows and columns both ascending: start top-right, move left or down.
    // Returns {row, col}; the values looked at are added to path.
    static int[] staircase(int[][] m, int key, List<Integer> path) {
        int r = 0, c = m[0].length - 1;
        while (r < m.length && c >= 0) {
            path.add(m[r][c]);
            if (m[r][c] == key) return new int[] {r, c};
            if (m[r][c] > key)
                c--;  // everything below in this column is even larger
            else
                r++;  // everything to the left in this row is even smaller
        }
        return new int[] {-1, -1};
    }
```

Listing 2.20 (Continued) *Search2D.java: the same program in Java.*

```
    // The whole matrix is one sorted row-major run. Returns {row, col, probes}.
    static int[] binary2d(int[][] m, int key) {
        int rows = m.length, cols = m[0].length;
        int low = 0, high = rows * cols - 1, probes = 0;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            int r = mid / cols, c = mid % cols;  // index k -> (row, column)
            probes++;
            if (m[r][c] == key) return new int[] {r, c, probes};
            if (m[r][c] < key)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return new int[] {-1, -1, probes};
    }

    public static void main(String[] args) {
        long base = 0x1000;
        for (int width : new int[] {2, 4}) {
            long a7 = address(base, width, 7);
            long b23 = address(base, width, 2 * 5 + 3);
            System.out.printf("int of %d bytes: &A[7] = 0x%X, &B[2][3] = 0x%X%n",
                    width, a7, b23);
        }

        int[][] shelves = {
                {72, 25, 96, 41, 88}, {34, 17, 63, 56, 12}, {80, 45, 29, 91, 7}};
        for (int key : new int[] {63, 50}) {
            int[] r = scan(shelves, key);
            if (r[0] >= 0)
                System.out.printf(
                        "scan for %d: shelf %d, slot %d after %d probes%n", key,
                        r[0], r[1], r[2]);
            else
                System.out.printf(
                        "scan for %d: not found after %d probes%n", key, r[2]);
        }
```

Listing 2.20 (Continued) *Search2D.java: the same program in Java.*

```
        int[][] m = {{10, 20, 30, 40, 50}, {15, 25, 35, 45, 55},
                {27, 29, 37, 48, 60}, {32, 33, 39, 50, 70}};
        for (int key : new int[] {37, 34}) {
            List<Integer> path = new ArrayList<>();
            int[] r = staircase(m, key, path);
            String where =
                    r[0] >= 0 ? "at (" + r[0] + ", " + r[1] + ")" : "not found";
            StringJoiner steps = new StringJoiner(" ");
            for (int x : path) steps.add(String.valueOf(x));
            System.out.printf("staircase for %d: %s, path %s (%d probes)%n", key,
                    where, steps, path.size());
        }

        int[][] s = {{2, 5, 9, 14}, {17, 23, 31, 38}, {42, 47, 55, 61}};
        for (int key : new int[] {31, 40}) {
            int[] r = binary2d(s, key);
            String where =
                    r[0] >= 0 ? "at (" + r[0] + ", " + r[1] + ")" : "not found";
            System.out.printf(
                    "row-major binary search for %d: %s after %d probes%n", key,
                    where, r[2]);
        }
    }
}
```

Listing 2.21 `search_2d.py`: the same program in Python, with a list of lists as the grid.

```
"""Searching two-dimensional arrays: addresses, a plain scan, the staircase,
and binary search over a matrix that is sorted in row-major order."""


def address(base, width, index):
    return base + index * width


def scan(grid, key):
    """Row by row, slot by slot: works on any grid. Returns (row, col, probes)."""
    probes = 0
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            probes += 1
            if grid[r][c] == key:
                return r, c, probes
    return -1, -1, probes


def staircase(m, key):
    """Rows and columns both ascending: start top-right, move left or down."""
    r, c = 0, len(m[0]) - 1
    path = []
    while r < len(m) and c >= 0:
        path.append(m[r][c])
        if m[r][c] == key:
            return r, c, path
        if m[r][c] > key:
            c -= 1  # everything below in this column is even larger
        else:
            r += 1  # everything to the left in this row is even smaller
    return -1, -1, path


def binary_2d(m, key):
    """The whole matrix is one sorted run in row-major order."""
    rows, cols = len(m), len(m[0])
    low, high = 0, rows * cols - 1
    probes = 0
    while low <= high:
        mid = low + (high - low) // 2
        r, c = mid // cols, mid % cols  # index k -> (row, column)
        probes += 1
        if m[r][c] == key:
            return r, c, probes
        if m[r][c] < key:
```

Listing 2.21 (Continued) *search_2d.py: the same program in Python, with a list of lists as the grid.*

```
            low = mid + 1
        else:
            high = mid - 1
    return -1, -1, probes


def main():
    base = 0x1000
    for width in (2, 4):
        a7 = address(base, width, 7)
        b23 = address(base, width, 2 * 5 + 3)
        print("int of %d bytes: &A[7] = 0x%X, &B[2][3] = 0x%X" % (width, a7, b23))

    shelves = [[72, 25, 96, 41, 88], [34, 17, 63, 56, 12], [80, 45, 29, 91, 7]]
    for key in (63, 50):
        r, c, p = scan(shelves, key)
        if r >= 0:
            print("scan for %d: shelf %d, slot %d after %d probes" % (key, r, c, p))
        else:
            print("scan for %d: not found after %d probes" % (key, p))

    m = [
        [10, 20, 30, 40, 50],
        [15, 25, 35, 45, 55],
        [27, 29, 37, 48, 60],
        [32, 33, 39, 50, 70],
    ]
    for key in (37, 34):
        r, c, path = staircase(m, key)
        where = "at (%d, %d)" % (r, c) if r >= 0 else "not found"
        print(
            "staircase for %d: %s, path %s (%d probes)"
            % (key, where, " ".join(str(x) for x in path), len(path))
        )

    s = [[2, 5, 9, 14], [17, 23, 31, 38], [42, 47, 55, 61]]
    for key in (31, 40):
        r, c, p = binary_2d(s, key)
        where = "at (%d, %d)" % (r, c) if r >= 0 else "not found"
        print(
            "row-major binary search for %d: %s after %d probes" % (key, where, p)
        )


if __name__ == "__main__":
    main()
```

All three print:

```
int of 2 bytes: &A[7] = 0x100E, &B[2][3] = 0x101A
int of 4 bytes: &A[7] = 0x101C, &B[2][3] = 0x1034
scan for 63: shelf 1, slot 2 after 8 probes
scan for 50: not found after 15 probes
staircase for 37: at (2, 2), path 50 40 30 35 37 (5 probes)
staircase for 34: not found, path 50 40 30 35 25 29 33 (7 probes)
row-major binary search for 31: at (1, 2) after 3 probes
row-major binary search for 40: not found after 4 probes
```

The scan found 63 at shelf 1, slot 2, after 8 probes — the five of row 0 and three of row 1 — and needed all 15 to rule out 50. The staircase paths are those of **Figure 2.9**. The binary search for 31

probed index 5, that is (1,1) holding 23, then index 8, that is (2,0) holding 42, then index 6, that is (1,2) holding 31.

2.10 Measuring on Real Data

2.10.1 A real catalog

Counting probes by hand on eight books teaches the mechanism; it does not tell us what happens at scale. So we now leave the computing shelf and give Pustaka Ceria a real catalog: the metadata of the ten thousand most-rated books on the Goodreads site, published as the goodbooks-10k data set [8]. From it we kept, for each book, its Goodreads identifier, its year of first publication, its number of ratings, its title and its writers, in a tab-separated file `catalog.tsv` that ships with this chapter's code. The file lists the books in order of popularity, which, from the point of view of the identifiers, is an arbitrary order: it plays the part of the arrival order.

The identifiers are real, and real data has shape. They run from 1 to 33,288,638, but half of them are below 395,090 — the older, more popular books were registered early, when the numbers were small. That is a heavily bunched distribution, and it will matter.

2.10.2 The experiment

Listing 2.22 loads the catalog, keeps the identifiers in file order for the two sequential searches, sorts a copy for the other three, and prepares two sets of queries: 1,000 identifiers that are present (every tenth book in file order) and 905 identifiers that are absent (one more than every tenth identifier in sorted order, skipping the few that happen to be taken). For each of the five searches it reports the average number of probes on each set, and the time per search, taking the best of nine timed runs so that interruptions by other programs on the computer affect the result as little as possible. At the end it uses binary search and lower bound for two everyday questions.

Listing 2.22 `measure_catalog.cpp`: five searches on a catalog of ten thousand real books, with probes counted and time measured.

```cpp
// Five searches on a real catalog of 10,000 books: probes counted, time measured.
#include <algorithm>
#include <chrono>
#include <cmath>
#include <cstdio>
#include <fstream>
#include <functional>
#include <numeric>
#include <sstream>
#include <string>
#include <unordered_set>
#include <vector>
using namespace std;

struct Book {
    long long id;
    int year;
    long long ratings;
    string title, writers;
};

struct Result {
    int index;
    int probes;
};

vector<Book> load(const string& path) {
    vector<Book> books;
    ifstream in(path);
    string line;
    getline(in, line);  // skip the header line
    while (getline(in, line)) {
        if (!line.empty() && line.back() == '\r') line.pop_back();
        stringstream ss(line);
        string id, year, ratings;
        Book b;
        getline(ss, id, '\t');
        getline(ss, year, '\t');
        getline(ss, ratings, '\t');
        getline(ss, b.title, '\t');
        getline(ss, b.writers, '\t');
        b.id = stoll(id);
        b.year = stoi(year);
        b.ratings = stoll(ratings);
        books.push_back(b);
    }
```

Listing 2.22 (Continued) *measure_catalog.cpp: five searches on a catalog of ten thousand real books, with probes counted and time measured.*

```
    return books;
}

Result sequential(const vector<long long>& a, long long key) {
    for (size_t i = 0; i < a.size(); i++)
        if (a[i] == key) return {(int)i, (int)i + 1};
    return {-1, (int)a.size()};
}

// a has a spare slot a[n]
Result sentinel(vector<long long>& a, int n, long long key) {
    a[n] = key;
    int i = 0;
    while (a[i] != key) i++;
    return {i < n ? i : -1, i + 1};
}

Result binary(const vector<long long>& a, long long key) {
    int low = 0, high = (int)a.size() - 1, probes = 0;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        probes++;
        if (a[mid] == key) return {mid, probes};
        if (a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return {-1, probes};
}

Result interpolation(const vector<long long>& a, long long key) {
    int low = 0, high = (int)a.size() - 1, probes = 0;
    while (low <= high && a[low] <= key && key <= a[high]) {
        int pos = low;
        if (a[high] != a[low])
            pos = low + (int)((double)(key - a[low]) / (double)(a[high] - a[low]) *
                              (high - low));
        probes++;
        if (a[pos] == key) return {pos, probes};
        if (a[pos] < key)
            low = pos + 1;
        else
            high = pos - 1;
    }
    return {-1, probes};
```

Listing 2.22 (Continued) *measure_catalog.cpp: five searches on a catalog of ten thousand real books, with probes counted and time measured.*

```
}

Result jump(const vector<long long>& a, long long key, int m) {
    int n = (int)a.size(), probes = 0;
    int start = 0, end = min(m, n);
    while (true) {
        probes++;
        if (a[end - 1] == key) return {end - 1, probes};
        if (a[end - 1] > key) break;
        start = end;
        end = min(end + m, n);
        if (start >= n) return {-1, probes};
    }
    for (int i = start; i < end - 1; i++) {
        probes++;
        if (a[i] == key) return {i, probes};
        if (a[i] > key) break;
    }
    return {-1, probes};
}

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;
        else
            high = mid;
    }
    return low;
}

int main() {
    vector<Book> books = load("catalog.tsv");
    int n = (int)books.size();
    // ids in the order the file lists them
    vector<long long> arrival;
    for (const Book& b : books) arrival.push_back(b.id);
    vector<long long> padded = arrival;
    // one spare slot for the sentinel
    padded.push_back(0);
    vector<int> byId(n);
    iota(byId.begin(), byId.end(), 0);
    sort(byId.begin(), byId.end(),
         [&](int x, int y) { return books[x].id < books[y].id; });
```

Listing 2.22 (Continued) *measure_catalog.cpp: five searches on a catalog of ten thousand real books, with probes counted and time measured.*

```
    vector<long long> ids;  // ids in ascending order
    for (int i : byId) ids.push_back(books[i].id);
    vector<long long> present, absent;
    for (int i = 0; i < n; i += 10) present.push_back(arrival[i]);
    unordered_set<long long> known(arrival.begin(), arrival.end());
    for (int j = 5; j < n; j += 10)
        if (!known.count(ids[j] + 1)) absent.push_back(ids[j] + 1);
    int m = (int)sqrt((double)n);
    printf("loaded %d books; ids from %lld to %lld, median %lld\n", n, ids[0],
           ids[n - 1], ids[n / 2]);
    printf("queries: %zu present ids, %zu absent ids\n", present.size(),
           absent.size());

    vector<pair<string, function<Result(long long)>>> methods = {
            {"sequential (file order)",
             [&](long long k) { return sequential(arrival, k); }},
            {"sentinel (file order)",
             [&](long long k) { return sentinel(padded, n, k); }},
            {"binary (sorted ids)", [&](long long k) { return binary(ids, k); }},
            {"interpolation (sorted ids)",
             [&](long long k) { return interpolation(ids, k); }},
            {"jump, m = " + to_string(m) + " (sorted ids)",
             [&](long long k) { return jump(ids, k, m); }}};
    printf("%-28s %10s %10s %14s\n", "method", "present", "absent",
           "ns per search");
    // keeps the optimizer honest
    volatile long long sink = 0;
    for (auto& [name, search] : methods) {
        double averages[2];
        const vector<long long>* sets[2] = {&present, &absent};
        for (int s = 0; s < 2; s++) {
            long long total = 0;
            for (long long k : *sets[s]) total += search(k).probes;
            averages[s] = (double)total / sets[s]->size();
        }
        long long best = -1;
        for (int run = 0; run < 9; run++) {  // best of nine timed runs
            auto t0 = chrono::steady_clock::now();
            for (long long k : present) sink = sink + search(k).index;
            for (long long k : absent) sink = sink + search(k).index;
            long long elapsed = chrono::duration_cast<chrono::nanoseconds>(
                                    chrono::steady_clock::now() - t0)
                                    .count();
            best = (best < 0) ? elapsed : min(best, elapsed);
        }
        double per = (double)best / (present.size() + absent.size());
```

Listing 2.22 (Continued) *measure_catalog.cpp: five searches on a catalog of ten thousand real books, with probes counted and time measured.*

```
        printf("%-28s %10.1f %10.1f %14.0f\n", name.c_str(), averages[0],
               averages[1], per);
    }

    Result r = binary(ids, 3);
    printf("id 3 -> %s (%d probes)\n", books[byId[r.index]].title.c_str(),
           r.probes);
    vector<int> years;
    for (const Book& b : books) years.push_back(b.year);
    sort(years.begin(), years.end());
    int first = lowerBound(years, 2005), after = lowerBound(years, 2006);
    printf("first published in 2005: %d books (sorted positions %d..%d)\n",
           after - first, first, after - 1);
    printf("first published 1990-1999: %d books\n",
           lowerBound(years, 2000) - lowerBound(years, 1990));
    return 0;
}
```

Listing 2.23 `MeasureCatalog.java`: the same experiment in Java.

```
// Five searches on a real catalog of 10,000 books: probes counted, time measured.
import java.io.BufferedReader;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.function.LongFunction;

public class MeasureCatalog {
    record Book(long id, int year, long ratings, String title, String writers) {}

    static List<Book> load(String path) throws IOException {
        List<Book> books = new ArrayList<>();
        try (BufferedReader in = Files.newBufferedReader(
                    Path.of(path), StandardCharsets.UTF_8)) {
            in.readLine();  // skip the header line
            String line;
            while ((line = in.readLine()) != null) {
                String[] f = line.split("\t", -1);
                books.add(new Book(Long.parseLong(f[0]), Integer.parseInt(f[1]),
                        Long.parseLong(f[2]), f[3], f[4]));
            }
        }
        return books;
    }

    // Round the exact binary value of x, as C's printf and Python's % do.
    static String fixed(double x, int digits) {
        return new BigDecimal(x)
                .setScale(digits, RoundingMode.HALF_EVEN)
                .toPlainString();
    }
```

Listing 2.23 (Continued) *MeasureCatalog.java: the same experiment in Java.*

```
    // Every search returns {index or -1, probes}.
    static int[] sequential(long[] a, long key) {
        for (int i = 0; i < a.length; i++)
            if (a[i] == key) return new int[] {i, i + 1};
        return new int[] {-1, a.length};
    }

    // a has a spare slot a[n]
    static int[] sentinel(long[] a, int n, long key) {
        a[n] = key;
        int i = 0;
        while (a[i] != key) i++;
        return new int[] {i < n ? i : -1, i + 1};
    }

    static int[] binary(long[] a, long key) {
        int low = 0, high = a.length - 1, probes = 0;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            probes++;
            if (a[mid] == key) return new int[] {mid, probes};
            if (a[mid] < key)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return new int[] {-1, probes};
    }

    static int[] interpolation(long[] a, long key) {
        int low = 0, high = a.length - 1, probes = 0;
        while (low <= high && a[low] <= key && key <= a[high]) {
            int pos = low;
            if (a[high] != a[low])
                pos = low +
                        (int) ((double) (key - a[low]) /
                                (double) (a[high] - a[low]) * (high - low));
            probes++;
            if (a[pos] == key) return new int[] {pos, probes};
            if (a[pos] < key)
                low = pos + 1;
            else
                high = pos - 1;
        }
        return new int[] {-1, probes};
    }
```

Listing 2.23 (Continued) *MeasureCatalog.java: the same experiment in Java.*

```
    static int[] jump(long[] a, long key, int m) {
        int n = a.length, probes = 0;
        int start = 0, end = Math.min(m, n);
        while (true) {
            probes++;
            if (a[end - 1] == key) return new int[] {end - 1, probes};
            if (a[end - 1] > key) break;
            start = end;
            end = Math.min(end + m, n);
            if (start >= n) return new int[] {-1, probes};
        }
        for (int i = start; i < end - 1; i++) {
            probes++;
            if (a[i] == key) return new int[] {i, probes};
            if (a[i] > key) break;
        }
        return new int[] {-1, probes};
    }

    static int lowerBound(int[] a, int key) {
        int low = 0, high = a.length;
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (a[mid] < key)
                low = mid + 1;
            else
                high = mid;
        }
        return low;
    }

    public static void main(String[] args) throws IOException {
        List<Book> books = load("catalog.tsv");
        int n = books.size();
        // ids in the order the file lists them
        long[] arrival = new long[n];
        for (int i = 0; i < n; i++) arrival[i] = books.get(i).id();
        // one spare slot for the sentinel
        long[] padded = Arrays.copyOf(arrival, n + 1);
        Integer[] byId = new Integer[n];
        for (int i = 0; i < n; i++) byId[i] = i;
        Arrays.sort(byId, Comparator.comparingLong(i -> books.get(i).id()));
        // ids in ascending order
        long[] ids = new long[n];
        for (int i = 0; i < n; i++) ids[i] = books.get(byId[i]).id();
```

Listing 2.23 (Continued) *MeasureCatalog.java: the same experiment in Java.*

```
        List<Long> presentList = new ArrayList<>(), absentList = new ArrayList<>();
        for (int i = 0; i < n; i += 10) presentList.add(arrival[i]);
        Set<Long> known = new HashSet<>();
        for (long id : arrival) known.add(id);
        for (int j = 5; j < n; j += 10)
            if (!known.contains(ids[j] + 1)) absentList.add(ids[j] + 1);
        long[] present = presentList.stream().mapToLong(Long::longValue).toArray();
        long[] absent = absentList.stream().mapToLong(Long::longValue).toArray();
        int m = (int) Math.sqrt(n);
        System.out.printf("loaded %d books; ids from %d to %d, median %d%n", n,
                ids[0], ids[n - 1], ids[n / 2]);
        System.out.printf("queries: %d present ids, %d absent ids%n",
                present.length, absent.length);

        String[] names = {"sequential (file order)", "sentinel (file order)",
                "binary (sorted ids)", "interpolation (sorted ids)",
                "jump, m = " + m + " (sorted ids)"};
        List<LongFunction<int[]>> methods = List.of(k
                -> sequential(arrival, k),
                k
                -> sentinel(padded, n, k),
                k
                -> binary(ids, k),
                k -> interpolation(ids, k), k -> jump(ids, k, m));
        System.out.printf("%-28s %10s %10s %14s%n", "method", "present", "absent",
                "ns per search");
        // keeps the JIT compiler honest
        long sink = 0;
        for (int t = 0; t < names.length; t++) {
            LongFunction<int[]> search = methods.get(t);
            double[] averages = new double[2];
            long[][] sets = {present, absent};
            for (int s = 0; s < 2; s++) {
                long total = 0;
                for (long k : sets[s]) total += search.apply(k)[1];
                averages[s] = (double) total / sets[s].length;
            }
            long best = -1;
            // best of nine timed runs
            for (int run = 0; run < 9; run++) {
                long t0 = System.nanoTime();
                for (long k : present) sink += search.apply(k)[0];
                for (long k : absent) sink += search.apply(k)[0];
                long elapsed = System.nanoTime() - t0;
                best = best < 0 ? elapsed : Math.min(best, elapsed);
            }
```

Listing 2.23 (Continued) *MeasureCatalog.java: the same experiment in Java.*

```
            double per = (double) best / (present.length + absent.length);
            System.out.printf("%-28s %10s %10s %14s%n", names[t],
                    fixed(averages[0], 1), fixed(averages[1], 1), fixed(per, 0));
        }
        if (sink == Long.MIN_VALUE) System.out.println(sink);

        int[] r = binary(ids, 3);
        System.out.printf(
                "id 3 -> %s (%d probes)%n", books.get(byId[r[0]]).title(), r[1]);
        int[] years = new int[n];
        for (int i = 0; i < n; i++) years[i] = books.get(i).year();
        Arrays.sort(years);
        int first = lowerBound(years, 2005), after = lowerBound(years, 2006);
        System.out.printf(
                "first published in 2005: %d books (sorted positions %d..%d)%n",
                after - first, first, after - 1);
        System.out.printf("first published 1990-1999: %d books%n",
                lowerBound(years, 2000) - lowerBound(years, 1990));
    }
}
```

Listing 2.24 `measure_catalog.py`: the same experiment in Python.

```
"""Five searches on a real 10,000-book catalog: probes counted, time measured."""

import math
import time


def load(path):
    books = []
    with open(path, encoding="utf-8") as f:
        f.readline()  # skip the header line
        for line in f:
            book_id, year, ratings, title, writers = line.rstrip("\n").split("\t")
            books.append((int(book_id), int(year), int(ratings), title, writers))
    return books


def sequential(a, key):
    for i in range(len(a)):
        if a[i] == key:
            return i, i + 1
    return -1, len(a)


def sentinel(a, n, key):  # a has a spare slot a[n]
    a[n] = key
    i = 0
    while a[i] != key:
        i += 1
    return (i if i < n else -1), i + 1


def binary(a, key):
    low, high, probes = 0, len(a) - 1, 0
    while low <= high:
        mid = low + (high - low) // 2
        probes += 1
        if a[mid] == key:
            return mid, probes
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1, probes
```

Listing 2.24 (Continued) *measure_catalog.py: the same experiment in Python.*

```
def interpolation(a, key):
    low, high, probes = 0, len(a) - 1, 0
    while low <= high and a[low] <= key <= a[high]:
        if a[high] == a[low]:
            pos = low
        else:
            pos = low + int(
                float(key - a[low]) / float(a[high] - a[low]) * (high - low)
            )
        probes += 1
        if a[pos] == key:
            return pos, probes
        if a[pos] < key:
            low = pos + 1
        else:
            high = pos - 1
    return -1, probes


def jump(a, key, m):
    n = len(a)
    probes = 0
    start, end = 0, min(m, n)
    while True:
        probes += 1
        if a[end - 1] == key:
            return end - 1, probes
        if a[end - 1] > key:
            break
        start, end = end, min(end + m, n)
        if start >= n:
            return -1, probes
    for i in range(start, end - 1):
        probes += 1
        if a[i] == key:
            return i, probes
        if a[i] > key:
            break
    return -1, probes
```

Listing 2.24 (Continued) *measure_catalog.py: the same experiment in Python.*

```
def lower_bound(a, key):
    low, high = 0, len(a)
    while low < high:
        mid = low + (high - low) // 2
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid
    return low


def main():
    books = load("catalog.tsv")
    n = len(books)
    # ids in the order the file lists them
    arrival = [b[0] for b in books]
    # one spare slot for the sentinel
    padded = arrival + [0]
    by_id = sorted(range(n), key=lambda i: books[i][0])
    ids = [books[i][0] for i in by_id]  # ids in ascending order
    present = [arrival[i] for i in range(0, n, 10)]
    known = set(arrival)
    absent = [ids[j] + 1 for j in range(5, n, 10) if ids[j] + 1 not in known]
    m = math.isqrt(n)
    print(
        "loaded %d books; ids from %d to %d, median %d"
        % (n, ids[0], ids[-1], ids[n // 2])
    )
    print("queries: %d present ids, %d absent ids" % (len(present), len(absent)))

    methods = [
        ("sequential (file order)", lambda k: sequential(arrival, k)),
        ("sentinel (file order)", lambda k: sentinel(padded, n, k)),
        ("binary (sorted ids)", lambda k: binary(ids, k)),
        ("interpolation (sorted ids)", lambda k: interpolation(ids, k)),
        ("jump, m = %d (sorted ids)" % m, lambda k: jump(ids, k, m)),
    ]
    print("%-28s %10s %10s %14s" % ("method", "present", "absent", "ns per search"))
    for name, search in methods:
        averages = []
        for queries in (present, absent):
            total = 0
            for k in queries:
                total += search(k)[1]
            averages.append(total / len(queries))
        best = None
```

Listing 2.24 (Continued) *measure_catalog.py: the same experiment in Python.*

```
        for run in range(9):  # best of nine timed runs
            t0 = time.perf_counter_ns()
            for k in present:
                search(k)
            for k in absent:
                search(k)
            elapsed = time.perf_counter_ns() - t0
            best = elapsed if best is None else min(best, elapsed)
        per = best / (len(present) + len(absent))
        print("%-28s %10.1f %10.1f %14.0f" % (name, averages[0], averages[1], per))

    i, probes = binary(ids, 3)
    print("id 3 -> %s (%d probes)" % (books[by_id[i]][3], probes))
    years = sorted(b[1] for b in books)
    first, after = lower_bound(years, 2005), lower_bound(years, 2006)
    print(
        "first published in 2005: %d books (sorted positions %d..%d)"
        % (after - first, first, after - 1)
    )
    print(
        "first published 1990-1999: %d books"
        % (lower_bound(years, 2000) - lower_bound(years, 1990))
    )


if __name__ == "__main__":
    main()
```

The C++ program, compiled with `g++ -std=c++17 -O2`, printed:

```
loaded 10000 books; ids from 1 to 33288638, median 395090
queries: 1000 present ids, 905 absent ids
method                          present     absent  ns per search
sequential (file order)          4996.0    10000.0           2243
sentinel (file order)            4996.0    10001.0           2248
binary (sorted ids)                12.4       13.3             89
interpolation (sorted ids)        211.2      191.8           2612
jump, m = 100 (sorted ids)         99.2      105.6            130
id 3 -> Harry Potter and the Sorcerer's Stone (Harry Potter, #1) (13 probes)
first published in 2005: 326 books (sorted positions 5067..5392)
first published 1990-1999: 1360 books
```

The Java program printed the same probe counts and different times:

```
loaded 10000 books; ids from 1 to 33288638, median 395090
queries: 1000 present ids, 905 absent ids
method                          present     absent  ns per search
sequential (file order)          4996.0    10000.0           1481
sentinel (file order)            4996.0    10001.0           1487
binary (sorted ids)                12.4       13.3             74
interpolation (sorted ids)        211.2      191.8           3414
jump, m = 100 (sorted ids)         99.2      105.6            214
id 3 -> Harry Potter and the Sorcerer's Stone (Harry Potter, #1) (13 probes)
first published in 2005: 326 books (sorted positions 5067..5392)
first published 1990-1999: 1360 books
```

And the Python program:

```
loaded 10000 books; ids from 1 to 33288638, median 395090
queries: 1000 present ids, 905 absent ids
method                          present     absent  ns per search
sequential (file order)          4996.0    10000.0         135376
sentinel (file order)            4996.0    10001.0         123270
binary (sorted ids)                12.4       13.3           1054
interpolation (sorted ids)        211.2      191.8          39667
jump, m = 100 (sorted ids)         99.2      105.6           7782
id 3 -> Harry Potter and the Sorcerer's Stone (Harry Potter, #1) (13 probes)
first published in 2005: 326 books (sorted positions 5067..5392)
first published 1990-1999: 1360 books
```

The three runs were made one after another on the same computer, a virtual machine with two cores of a 2.1 GHz Intel Xeon processor, in September 2026, with g++ 13.3, OpenJDK 21 and CPython 3.11. On your computer every time will be different; the probe counts will be exactly the same. **Table 2.3** gathers the results.

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

Figure 2.10 draws the three time columns on a logarithmic scale, on which equal ratios look like equal distances.

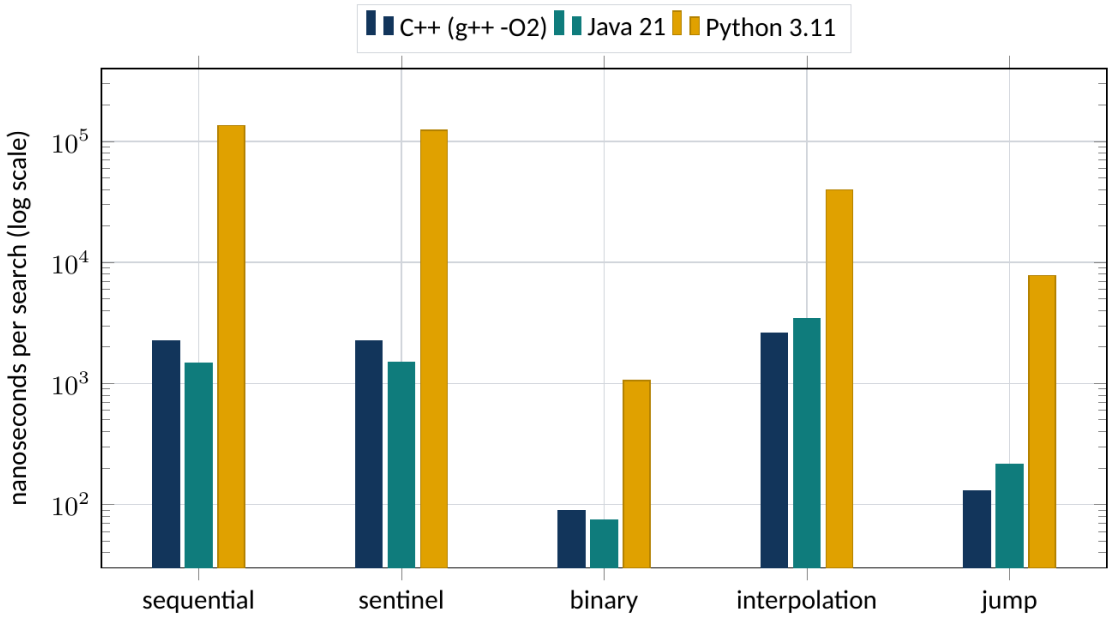


Figure 2.10 Nanoseconds per search on the 10,000-book catalog, by search method and language, on a logarithmic scale. Measured by **Listing 2.22**, **Listing 2.23** and **Listing 2.24** in September 2026.

2.10.3 What the numbers say

The probe counts match the theory. Sequential search averaged 4,996 probes on present keys — Equation (2.2) predicts $(n + 1)/2 = 5{,}000.5$ for a key at a random position; our queries were every tenth position, whose average is 4,996 exactly — and precisely $n = 10{,}000$ on absent keys.

The sentinel version needed 10,001 on absent keys: the extra probe is the sentinel itself. Binary search averaged 12.4 on present keys and 13.3 on absent ones, never more than the $\lfloor \lg 10^4 \rfloor + 1 = 14$ of Equation (2.5). Jump search with $m = \lfloor \sqrt{10^4} \rfloor = 100$ averaged about 100, half of the worst case of Equation (2.9), as expected for keys spread over all the blocks.

Interpolation search failed on real identifiers. 211 probes where binary search needed 12. Look at the numbers of Section 2.10.1: when the search begins, the straight line runs from identifier 1 to identifier 33,288,638, but half of all books have identifiers below 395,090, so a book whose identifier is, say, 400,000 is at position about 5,000, while the line predicts position $\lfloor 400{,}000/33{,}288{,}638 \cdot 9{,}999 \rfloor = 120$. The probe lands thousands of places too early, the interval shrinks only a little from the left, and the next estimate is again too early. This is the degradation that Section 2.7.4 warned about, happening on real data.

Probes are not time. In C++ binary search was about 25 times faster than sequential search, not the almost 600 times that the probe counts suggest, and interpolation search, with about 36 times fewer probes than sequential search, was *slower*. Three effects explain it. First, a sequential scan reads memory in order, which the processor prefetches ahead of time, so each probe costs a fraction of a nanosecond; binary and interpolation probes jump around memory. Second, a sequential scan's branch — "not this one, not this one…" — is predictable, whereas each binary-search branch is a coin toss that the processor guesses wrongly half the time. Third, each interpolation probe performs a floating-point division, one of the slowest arithmetic instructions. Probe counts tell you how an algorithm *scales*; only a clock tells you how fast it runs on a given machine.

The sentinel did not help in C++ or Java. 2,243 against 2,248 nanoseconds in C++, 1,481 against 1,487 in Java: the difference is smaller than the variation between runs. The optimizing compilers, and the processor's branch predictor, already make the bound test nearly free. In Python, where each test is an interpreted instruction, the sentinel saved about 9%.

The language matters by a constant factor; the algorithm matters by an order of magnitude. Python was about 60 times slower than C++ for sequential search and about 12 times slower for binary search, because Python pays a fixed interpretation cost on each probe. But binary search in Python (1,054 ns) beat sequential search in C++ (2,243 ns). A better algorithm in a slower language outran a worse algorithm in a faster one — and the gap only widens as n grows.

2.10.4 How probe counts grow

To see growth, we need to vary n, and a real catalog has only one size. **Listing 2.25** therefore generates sorted keys itself with a small, fully specified pseudo-random generator, SplitMix64, seeded with 2026, so that all three languages produce the same keys and print the same table. For each n from 10^3 to 10^6 it makes 200 successful searches with every method. It also builds a second array of **cubic** keys, 0,1,8,27, … , i^3, which crowd together at the low end and spread out at the high end, to show interpolation search on bunched data. **Listing 2.26** and **Listing 2.27** repeat the experiment in Java and Python.

Listing 2.25 `scaling.cpp`: average probes as n grows, on generated uniform keys and on cubic keys.

```cpp
// How the number of probes grows with n: every search, on sorted data we generate
// ourselves with a fixed seed, so that C++, Java and Python print the same table.
#include <algorithm>
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <vector>
using namespace std;

// A tiny, well-known pseudo-random generator with 64-bit arithmetic.
struct SplitMix64 {
    uint64_t state;
    explicit SplitMix64(uint64_t seed) : state(seed) {}
    uint64_t next() {
        state += 0x9E3779B97F4A7C15ULL;
        uint64_t z = state;
        z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9ULL;
        z = (z ^ (z >> 27)) * 0x94D049BB133111EBULL;
        return z ^ (z >> 31);
    }
    uint64_t below(uint64_t bound) { return next() % bound; }
};

using Keys = vector<int64_t>;

int sequential(const Keys& a, int64_t key) {
    for (size_t i = 0; i < a.size(); i++)
        if (a[i] == key) return (int)i + 1;
    return (int)a.size();
}

int binary(const Keys& a, int64_t key) {
    int low = 0, high = (int)a.size() - 1, probes = 0;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        probes++;
        if (a[mid] == key) return probes;
        if (a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return probes;
}
```

Listing 2.25 (Continued) *scaling.cpp: average probes as n grows, on generated uniform keys and on cubic keys.*

```
int interpolation(const Keys& a, int64_t key) {
    int low = 0, high = (int)a.size() - 1, probes = 0;
    while (low <= high && a[low] <= key && key <= a[high]) {
        int pos = low;
        if (a[high] != a[low])
            pos = low + (int)((double)(key - a[low]) / (double)(a[high] - a[low]) *
                              (high - low));
        probes++;
        if (a[pos] == key) return probes;
        if (a[pos] < key)
            low = pos + 1;
        else
            high = pos - 1;
    }
    return probes;
}

int isqrt(int n) {
    int m = (int)sqrt((double)n);
    while (m * m > n) m--;
    while ((m + 1) * (m + 1) <= n) m++;
    return m;
}

int jump(const Keys& a, int64_t key) {
    int n = (int)a.size(), m = isqrt(n), probes = 0;
    int start = 0, end = min(m, n);
    while (true) {
        probes++;
        if (a[end - 1] == key) return probes;
        if (a[end - 1] > key) break;
        start = end;
        end = min(end + m, n);
        if (start >= n) return probes;
    }
    for (int i = start; i < end - 1; i++) {
        probes++;
        if (a[i] == key || a[i] > key) break;
    }
    return probes;
}
```

Listing 2.25 (Continued) *scaling.cpp: average probes as n grows, on generated uniform keys and on cubic keys.*

```
int main() {
    const int queries = 200;
    printf("average probes over %d successful searches\n", queries);
    printf("%8s %10s %7s %6s %9s %9s %7s\n", "n", "sequential", "binary", "lg n",
           "interp-U", "interp-C", "jump");
    for (int n : {1000, 10000, 100000, 1000000}) {
        SplitMix64 rng(2026);
        Keys uniform(n), cubic(n);
        for (int i = 0; i < n; i++) uniform[i] = (int64_t)rng.below(10ULL * n);
        sort(uniform.begin(), uniform.end());
        // keys crowd together at the low end
        for (int i = 0; i < n; i++) cubic[i] = (int64_t)i * i * i;
        vector<int> picks(queries);
        for (int& p : picks) p = (int)rng.below(n);
        long long seq = 0, bi = 0, iu = 0, ic = 0, ju = 0;
        for (int p : picks) {
            seq += sequential(uniform, uniform[p]);
            bi += binary(uniform, uniform[p]);
            iu += interpolation(uniform, uniform[p]);
            ic += interpolation(cubic, cubic[p]);
            ju += jump(uniform, uniform[p]);
        }
        printf("%8d %10.1f %7.2f %6.2f %9.2f %9.2f %7.1f\n", n,
               (double)seq / queries, (double)bi / queries, log2((double)n),
               (double)iu / queries, (double)ic / queries, (double)ju / queries);
    }
    return 0;
}
```

Listing 2.26 `Scaling.java`: the same experiment in Java. The helper `fixed` rounds the exact binary value, as C and Python do, so that the three tables agree digit for digit.

```java
// How the number of probes grows with n: every search, on sorted data we generate
// ourselves with a fixed seed, so that C++, Java and Python print the same table.
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.Arrays;

public class Scaling {
    // A tiny, well-known pseudo-random generator with 64-bit arithmetic.
    static final class SplitMix64 {
        private long state;
        SplitMix64(long seed) { state = seed; }
        long next() {
            state += 0x9E3779B97F4A7C15L;
            long z = state;
            z = (z ^ (z >>> 30)) * 0xBF58476D1CE4E5B9L;
            z = (z ^ (z >>> 27)) * 0x94D049BB133111EBL;
            return z ^ (z >>> 31);
        }
        long below(long bound) { return Long.remainderUnsigned(next(), bound); }
    }

    // Round the exact binary value of x, as C's printf and Python's % do.
    static String fixed(double x, int digits) {
        return new BigDecimal(x)
                .setScale(digits, RoundingMode.HALF_EVEN)
                .toPlainString();
    }

    static int sequential(long[] a, long key) {
        for (int i = 0; i < a.length; i++)
            if (a[i] == key) return i + 1;
        return a.length;
    }

    static int binary(long[] a, long key) {
        int low = 0, high = a.length - 1, probes = 0;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            probes++;
            if (a[mid] == key) return probes;
            if (a[mid] < key)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return probes;
```

Listing 2.26 (Continued) *Scaling.java: the same experiment in Java. The helper fixed rounds the exact binary value, as C and Python do, so that the three tables agree digit for digit.*

```
    }

    static int interpolation(long[] a, long key) {
        int low = 0, high = a.length - 1, probes = 0;
        while (low <= high && a[low] <= key && key <= a[high]) {
            int pos = low;
            if (a[high] != a[low])
                pos = low +
                        (int) ((double) (key - a[low]) /
                                (double) (a[high] - a[low]) * (high - low));
            probes++;
            if (a[pos] == key) return probes;
            if (a[pos] < key)
                low = pos + 1;
            else
                high = pos - 1;
        }
        return probes;
    }

    static int isqrt(int n) {
        int m = (int) Math.sqrt(n);
        while (m * m > n) m--;
        while ((m + 1) * (m + 1) <= n) m++;
        return m;
    }

    static int jump(long[] a, long key) {
        int n = a.length, m = isqrt(n), probes = 0;
        int start = 0, end = Math.min(m, n);
        while (true) {
            probes++;
            if (a[end - 1] == key) return probes;
            if (a[end - 1] > key) break;
            start = end;
            end = Math.min(end + m, n);
            if (start >= n) return probes;
        }
        for (int i = start; i < end - 1; i++) {
            probes++;
            if (a[i] == key || a[i] > key) break;
        }
        return probes;
    }
```

Listing 2.26 (Continued) *Scaling.java: the same experiment in Java. The helper fixed rounds the exact binary value, as C and Python do, so that the three tables agree digit for digit.*

```
    public static void main(String[] args) {
        int queries = 200;
        System.out.printf("average probes over %d successful searches%n", queries);
        System.out.printf("%8s %10s %7s %6s %9s %9s %7s%n", "n", "sequential",
                "binary", "lg n", "interp-U", "interp-C", "jump");
        for (int n : new int[] {1000, 10000, 100000, 1000000}) {
            SplitMix64 rng = new SplitMix64(2026);
            long[] uniform = new long[n], cubic = new long[n];
            for (int i = 0; i < n; i++) uniform[i] = rng.below(10L * n);
            Arrays.sort(uniform);
            // keys crowd together at the low end
            for (int i = 0; i < n; i++) cubic[i] = (long) i * i * i;
            int[] picks = new int[queries];
            for (int q = 0; q < queries; q++) picks[q] = (int) rng.below(n);
            long seq = 0, bi = 0, iu = 0, ic = 0, ju = 0;
            for (int p : picks) {
                seq += sequential(uniform, uniform[p]);
                bi += binary(uniform, uniform[p]);
                iu += interpolation(uniform, uniform[p]);
                ic += interpolation(cubic, cubic[p]);
                ju += jump(uniform, uniform[p]);
            }
            System.out.printf("%8d %10s %7s %6s %9s %9s %7s%n", n,
                    fixed((double) seq / queries, 1),
                    fixed((double) bi / queries, 2),
                    fixed(Math.log(n) / Math.log(2), 2),
                    fixed((double) iu / queries, 2),
                    fixed((double) ic / queries, 2),
                    fixed((double) ju / queries, 1));
        }
    }
}
```

Listing 2.27 `scaling.py`: the same experiment in Python.

```python
"""How the number of probes grows with n: every search, on sorted data we generate
ourselves with a fixed seed, so that C++, Java and Python print the same table."""

import math

MASK = (1 << 64) - 1


class SplitMix64:
    """A tiny, well-known pseudo-random generator with 64-bit arithmetic."""

    def __init__(self, seed):
        self.state = seed

    def next(self):
        self.state = (self.state + 0x9E3779B97F4A7C15) & MASK
        z = self.state
        z = ((z ^ (z >> 30)) * 0xBF58476D1CE4E5B9) & MASK
        z = ((z ^ (z >> 27)) * 0x94D049BB133111EB) & MASK
        return z ^ (z >> 31)

    def below(self, bound):
        return self.next() % bound


def sequential(a, key):
    for i in range(len(a)):
        if a[i] == key:
            return i + 1
    return len(a)


def binary(a, key):
    low, high, probes = 0, len(a) - 1, 0
    while low <= high:
        mid = low + (high - low) // 2
        probes += 1
        if a[mid] == key:
            return probes
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return probes
```

Listing 2.27 (Continued) *scaling.py: the same experiment in Python.*

```
def interpolation(a, key):
    low, high, probes = 0, len(a) - 1, 0
    while low <= high and a[low] <= key <= a[high]:
        if a[high] == a[low]:
            pos = low
        else:
            pos = low + int(
                float(key - a[low]) / float(a[high] - a[low]) * (high - low)
            )
        probes += 1
        if a[pos] == key:
            return probes
        if a[pos] < key:
            low = pos + 1
        else:
            high = pos - 1
    return probes


def jump(a, key):
    n = len(a)
    m = math.isqrt(n)
    probes = 0
    start, end = 0, min(m, n)
    while True:
        probes += 1
        if a[end - 1] == key:
            return probes
        if a[end - 1] > key:
            break
        start, end = end, min(end + m, n)
        if start >= n:
            return probes
    for i in range(start, end - 1):
        probes += 1
        if a[i] == key or a[i] > key:
            break
    return probes
```

Listing 2.27 (Continued) *scaling.py: the same experiment in Python.*

```
def main():
    queries = 200
    print("average probes over %d successful searches" % queries)
    print(
        "%8s %10s %7s %6s %9s %9s %7s"
        % ("n", "sequential", "binary", "lg n", "interp-U", "interp-C", "jump")
    )
    for n in (1000, 10000, 100000, 1000000):
        rng = SplitMix64(2026)
        uniform = sorted(rng.below(10 * n) for _ in range(n))
        # keys crowd together at the low end
        cubic = [i * i * i for i in range(n)]
        picks = [rng.below(n) for _ in range(queries)]
        seq = sum(sequential(uniform, uniform[p]) for p in picks) / queries
        bi = sum(binary(uniform, uniform[p]) for p in picks) / queries
        iu = sum(interpolation(uniform, uniform[p]) for p in picks) / queries
        ic = sum(interpolation(cubic, cubic[p]) for p in picks) / queries
        ju = sum(jump(uniform, uniform[p]) for p in picks) / queries
        print(
            "%8d %10.1f %7.2f %6.2f %9.2f %9.2f %7.1f"
            % (n, seq, bi, math.log2(n), iu, ic, ju)
        )


if __name__ == "__main__":
    main()
```

All three print:

```
average probes over 200 successful searches
       n sequential  binary   lg n  interp-U  interp-C    jump
    1000      506.2    8.86    9.97      2.91     22.41    31.3
   10000     5032.7   12.38   13.29      3.64     54.13    99.5
  100000    51919.3   15.72   16.61      4.05    108.86   312.3
 1000000   500932.9   18.79   19.93      4.26    301.42   999.3
```

Figure 2.11 plots the table on logarithmic axes, where a quantity proportional to n^k appears as a straight line of slope k.

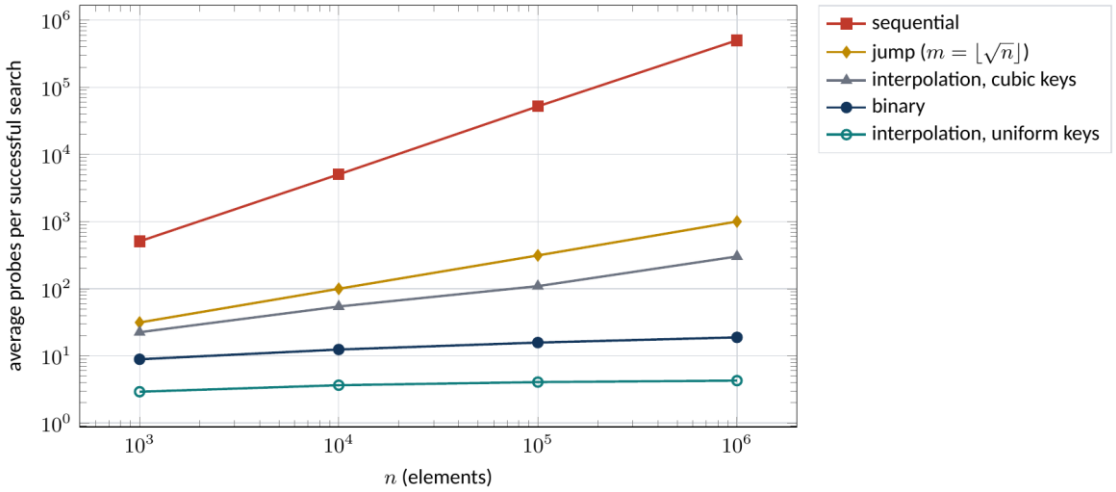


Figure 2.11 Average probes per successful search as n grows from 10^3 to 10^6, on logarithmic axes. Sequential search rises with slope 1, jump search with slope $1/2$; binary search rises by about 3.3 probes for every tenfold increase; interpolation search on uniform keys barely rises at all, but on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

Every curve behaves as its analysis predicts. Sequential search multiplies by ten when n does. Jump search multiplies by about $\sqrt{10} \approx 3.16$. Binary search adds about $\lg 10 \approx 3.32$ probes per tenfold step and stays just below $\lg n$, the column printed beside it. Interpolation search on uniform keys moves from 2.91 to 4.26 while n grows a thousandfold — the $\lg\lg n$ behavior of Section 2.7.4 — and on cubic keys it grows from 22 to 301, overtaking binary search already at $n = 1{,}000$.

2.11 Searching with the Standard Libraries

You now know how the searches work; in production code you should almost always call the version in your language's standard library, which has been tested by millions of programs. But you must know exactly what each one promises, because they differ in small ways that cause large bugs. **Table 2.4** lines them up.

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 $\geq x$	`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+)

2.11.1 C++

The C++ `<algorithm>` header provides `std::find` for linear search, and four binary searches that all take a sorted range `[first, last)` — the half-open convention of Section 2.6. `std::binary_search` answers only yes or no. `std::lower_bound` and `std::upper_bound` return iterators to the positions defined in Section 2.6.3; subtract `begin()` to get an index. `std::equal_range` returns both at once. The C++ standard guarantees at most $\log_2(last - first) + O(1)$ comparisons for each [9]. All four accept an optional comparator, which is how you search an array of records by one field: the comparator receives an element and the key and says whether the element comes before the key.

Since C++20 each of these also exists in the `std::ranges` namespace, taking a whole container, `std::ranges::lower_bound(v, x)`, and a *projection* that picks the field to compare. GCC 16,

released in April 2026, compiles C++20 by default [10]; this book's programs use C++17 and the classic forms so that they compile on every compiler in use. **Listing 2.28** exercises all of them, including a search of records by code with a comparator.

Listing 2.28 `library_search.cpp`: `std::find`, `std::binary_search`, `std::lower_bound`, `std::upper_bound` and `std::equal_range`, including a search of records by code.

```
// Searching with the C++ standard library: find, binary_search, lower_bound,
// upper_bound and equal_range from <algorithm>.
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

struct Book {
    int code;
    string title;
};

void print(const char* label, const vector<int>& v) {
    printf("%s", label);
    for (int x : v) printf(" %d", x);
    printf("\n");
}

int main() {
    vector<int> arrival = {72, 25, 96, 41, 88, 34, 63, 56};
    auto it = find(arrival.begin(), arrival.end(), 63);  // linear scan
    printf("find(63): index %td\n", it - arrival.begin());
    it = find(arrival.begin(), arrival.end(), 60);
    printf("find(60): %s\n", it == arrival.end() ? "end() -- not found" : "found");

    vector<int> codes = arrival;
    sort(codes.begin(), codes.end());
    print("sorted codes:", codes);
    printf("binary_search(88): %s\n",
           binary_search(codes.begin(), codes.end(), 88) ? "true" : "false");
    printf("binary_search(60): %s\n",
           binary_search(codes.begin(), codes.end(), 60) ? "true" : "false");
    auto lb = lower_bound(codes.begin(), codes.end(), 60);
    printf("lower_bound(60): index %td, where 60 would be inserted\n",
           lb - codes.begin());
```

Listing 2.28 (Continued) *library_search.cpp: std::find, std::binary_search, std::lower_bound, std::upper_bound and std::equal_range, including a search of records by code.*

```
    vector<int> years = {1998, 2001, 2001, 2005, 2005,
                         2005, 2009, 2012, 2012, 2020};
    auto lo = lower_bound(years.begin(), years.end(), 2005);
    auto hi = upper_bound(years.begin(), years.end(), 2005);
    printf("2005: lower_bound = %td, upper_bound = %td, count = %td\n",
           lo - years.begin(), hi - years.begin(), hi - lo);
    auto range = equal_range(years.begin(), years.end(), 2012);
    printf("equal_range(2012) = [%td, %td)\n", range.first - years.begin(),
           range.second - years.begin());
    // keeps it sorted
    years.insert(upper_bound(years.begin(), years.end(), 2003), 2003);
    print("after inserting 2003:", years);

    vector<Book> shelf = {
            {25, "The C++ Programming"},       {34, "Mastering Delphi 6"},
            {41, "Professional C#"},           {56, "Pure JavaScript v2"},
            {63, "Advanced JSP & Servlet"},    {72, "Calculus Make it Easy"},
            {88, "Visual Basic 2005 Express"}, {96, "Artificial Life : Volume 1"}};
    auto rec = lower_bound(shelf.begin(), shelf.end(), 63,
                           [](const Book& b, int code) { return b.code < code; });
    if (rec != shelf.end() && rec->code == 63)
        printf("record with code 063: %s\n", rec->title.c_str());
    return 0;
}
```

```
find(63): index 6
find(60): end() -- not found
sorted codes: 25 34 41 56 63 72 88 96
binary_search(88): true
binary_search(60): false
lower_bound(60): index 4, where 60 would be inserted
2005: lower_bound = 3, upper_bound = 6, count = 3
equal_range(2012) = [7, 9)
after inserting 2003: 1998 2001 2001 2003 2005 2005 2005 2009 2012 2012 2020
record with code 063: Advanced JSP & Servlet
```

2.11.2 Java

Java offers `List.indexOf` and `List.contains` for linear search, and two binary searches: `Arrays.binarySearch` for arrays and `Collections.binarySearch` for lists. Their return value packs two answers into one `int` [11], [12]. If the key is found, the result is its index, which is non-negative. If it is not found, the result is $-(ip) - 1$, where *ip*, the *insertion point*, is the index of the first element greater than the key — our lower bound. The "-1" guarantees that an absent key never returns 0, which would look like "found at index 0". So a negative result r means "absent, and would go at $-r - 1$".

Two warnings from the documentation deserve repeating. If the array is not sorted, the result is undefined. And if the key occurs several times, "there is no guarantee which one will be found" [11]: `Arrays.binarySearch` is **Pseudocode 2.3**, not lower bound. Java has no `upper_bound`; when you need one, the ten lines of **Pseudocode 2.5** with `<=` are the answer. **Listing 2.29** shows the methods at work, including the negative return value.

Listing 2.29 `LibrarySearch.java`: `indexOf`, `Arrays.binarySearch` and `Collections.binarySearch`, including the negative return value and a search of records with a comparator.

```
// Searching with the Java library: List.indexOf, Arrays.binarySearch and
// Collections.binarySearch.
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class LibrarySearch {
    record Book(int code, String title) {}

    public static void main(String[] args) {
        List<Integer> arrival = List.of(72, 25, 96, 41, 88, 34, 63, 56);
        // linear scan
        System.out.println("arrival.contains(63): " + arrival.contains(63));
        // linear scan
        System.out.println("arrival.indexOf(63): " + arrival.indexOf(63));
        System.out.println("arrival.indexOf(60): " + arrival.indexOf(60));

        int[] codes = {72, 25, 96, 41, 88, 34, 63, 56};
        Arrays.sort(codes);
        System.out.println("sorted codes: " + Arrays.toString(codes));
        System.out.println("Arrays.binarySearch(codes, 88) = " +
                Arrays.binarySearch(codes, 88));
        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);
```

Listing 2.29 (Continued) *LibrarySearch.java: indexOf, Arrays.binarySearch and Collections.binarySearch, including the negative return value and a search of records with a comparator.*

```
        List<Book> shelf = List.of(new Book(25, "The C++ Programming"),
                new Book(34, "Mastering Delphi 6"), new Book(41, "Professional C#"),
                new Book(56, "Pure JavaScript v2"),
                new Book(63, "Advanced JSP & Servlet"),
                new Book(72, "Calculus Make it Easy"),
                new Book(88, "Visual Basic 2005 Express"),
                new Book(96, "Artificial Life : Volume 1"));
        int i = Collections.binarySearch(
                shelf, new Book(63, ""), Comparator.comparingInt(Book::code));
        System.out.println("record with code 063: " + shelf.get(i).title());
    }
}
```

```
arrival.contains(63): true
arrival.indexOf(63): 6
arrival.indexOf(60): -1
sorted codes: [25, 34, 41, 56, 63, 72, 88, 96]
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]
record with code 063: Advanced JSP & Servlet
```

The call for 60 returned -5, so 60 would be inserted at $-(-5)-1=4$, between 56 and 63. The call for 2005 happened to return index 4, the middle copy — not the first.

The Java programs in this book are written for Java 25, the current long-term-support release, and tested with javac 21; Java 27, the newest feature release, appeared on 15 September 2026 [13]. None of the methods used here changed between those versions.

2.11.3 Python

Python's `in` operator and `list.index` perform linear search; `list.index` raises `ValueError` rather than returning -1. For sorted lists the `bisect` module provides `bisect_left` (lower bound) and `bisect_right` (upper bound, also available as `bisect`), each with optional `lo` and `hi` arguments to search a slice [14]. Since Python 3.10 each also takes a `key` function, which is how you search a list of records by one field: the function is applied to the elements, not to the search key, so you pass the bare key value. `insort` inserts while keeping the list sorted; its documentation points out that the $O(\lg n)$ search is "dominated by the slow $O(n)$ insertion step", since every later element must move one place to the right [14]. **Listing 2.30** tries each of them.

Listing 2.30 `library_search.py`: `in`, `list.index` and the `bisect` module, including a search of records with `key=`.

```
"""Searching with Python's own tools: in, list.index, and the bisect module."""

import bisect


def main():
    arrival = [72, 25, 96, 41, 88, 34, 63, 56]
    print("63 in arrival:", 63 in arrival)  # linear scan, True/False
    print("arrival.index(63):", arrival.index(63))  # linear scan, position
    try:
        arrival.index(60)
    except ValueError as e:
        print("arrival.index(60): ValueError:", e)

    codes = sorted(arrival)
    print("sorted codes:", codes)
    i = bisect.bisect_left(codes, 88)
    print(
        "bisect_left(codes, 88) = %d, found: %s"
        % (i, i < len(codes) and codes[i] == 88)
    )
    i = bisect.bisect_left(codes, 60)
    print(
        "bisect_left(codes, 60) = %d, found: %s"
        % (i, i < len(codes) and codes[i] == 60)
    )

    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)

    shelf = [
        (25, "The C++ Programming"),
        (34, "Mastering Delphi 6"),
        (41, "Professional C#"),
        (56, "Pure JavaScript v2"),
        (63, "Advanced JSP & Servlet"),
        (72, "Calculus Make it Easy"),
        (88, "Visual Basic 2005 Express"),
        (96, "Artificial Life : Volume 1"),
    ]
    i = bisect.bisect_left(shelf, 63, key=lambda book: book[0])  # Python 3.10+
```

Listing 2.30 (Continued) *library_search.py: in, list.index and the bisect module, including a search of records with key=.*

```
    print("record with code 063:", shelf[i][1])


if __name__ == "__main__":
    main()
```

```
63 in arrival: True
arrival.index(63): 6
arrival.index(60): ValueError: 60 is not in list
sorted codes: [25, 34, 41, 56, 63, 72, 88, 96]
bisect_left(codes, 88) = 6, found: True
bisect_left(codes, 60) = 4, found: False
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
```

The three programs of this section print slightly different lines, because the three libraries report their answers differently; each answer is the same fact about the same data.

2.12 Choosing a Search Method, and the Limit of Comparison Searching

2.12.1 The whole picture

Table 2.5 summarizes the chapter.

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	$O(1)$

Choosing is then a matter of asking questions in order. Is the data sorted, or can it be kept sorted? If not, and it will be searched only a few times, use sequential search; if it will be searched many times, sort it once (Chapter 3) or put it in a hash table (Chapter 15). If it is sorted, use binary search — in its lower-bound form if duplicates or ranges matter. Consider interpolation search only if the keys are numbers, you have *measured* that they are evenly spread, and probes are expensive (for instance, each probe is a disk read). Consider jump search only when moving backwards is expensive.

2.12.2 Why nothing that compares can beat binary search

Could a cleverer comparison-based algorithm search a sorted array in fewer than $\lfloor \lg n \rfloor + 1$ probes in the worst case? No — and the reason is a counting argument that we shall meet again for sorting in Chapter 3.

Any algorithm that learns about the array only by comparing the key with elements can be drawn as a **decision tree**. Each internal node is a comparison of the key with one element, with three outcomes: equal (stop, found), smaller (go left), larger (go right). Each leaf is a final answer "absent, and the key falls in this gap". **Figure 2.12** shows the tree of binary search for $n = 7$.

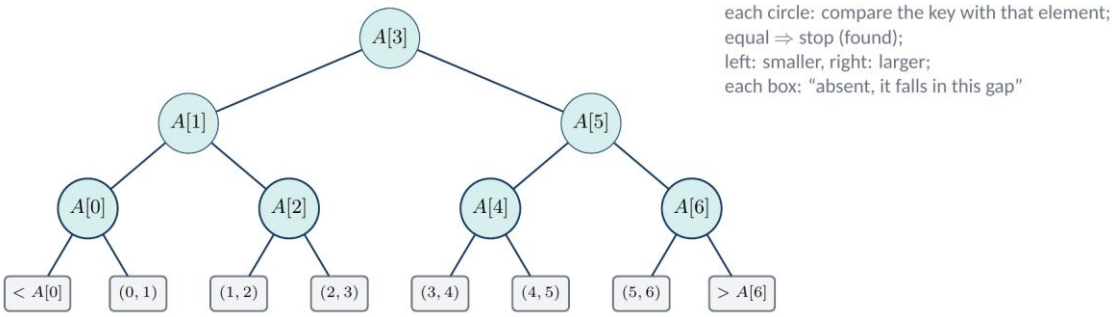


Figure 2.12 The decision tree of binary search on seven sorted elements. Each circle compares the key with one element; each box is one of the eight gaps into which an absent key can fall. Every root-to-box path has three comparisons.

An absent key can fall into any of $n + 1$ gaps: before $A[0]$, between each neighboring pair, and after $A[n - 1]$. A correct algorithm must be able to distinguish them all, so its tree has at least $n + 1$ leaves. A binary tree of height h has at most 2^h leaves, so $2^h \geq n + 1$, and taking logarithms,

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

The height is the number of comparisons on the longest path, that is, the worst case. And $\lceil \lg(n + 1) \rceil$ equals $\lfloor \lg n \rfloor + 1$ for every $n \geq 1$ (both are the number of binary digits of n), so Equation (2.11) says that binary search's worst case, Equation (2.5), is the best possible. Binary search is not merely good; among algorithms that only compare, it is optimal.

The argument has an escape clause, and it is instructive. It applies to algorithms that learn *only by comparing*. Interpolation search does arithmetic on the key values, which is why it can beat $\lg n$ on friendly data. Hash tables (Chapter 15) compute a position directly from the key and, on average, find it in $O(1)$ probes. To beat binary search you must know something about your keys beyond their order.

2.13 Profitina in Practice

2.13.1 Where this chapter appears in a real product

Profitina is an AI-visibility platform: it scans a brand's website, scores how ready the site is to be cited by AI answer engines, explains each signal, and helps the owner fix what it finds. It runs in production at `app.profitina.com`, as a Python web application on PostgreSQL with a locally hosted Qwen2.5 model for its content and technical assistant [15]. Searching, in the forms of this chapter, sits under several of its features.

Time series are sorted, so ranges are two bound searches. The historical analytics dashboard shows each brand's AI-visibility score week by week, and exports it as CSV for a chosen period. A brand's history is naturally kept in time order, because scans are appended as they happen. "The scans between 1 July and 30 September" is then exactly $A[lower(\text{1 July})..lower(\text{1 October}) - 1]$ — two lower-bound searches, $O(\lg n)$, however long the history grows. In the database the same idea appears as an index on the timestamp column, which is a sorted structure searched the same way.

"N new" is an upper bound. The dashboard shows, for each followed community channel, how many top-level posts are new since the user last looked. Posts are stored in time order, so the count is the number of posts later than the last-read time t: $n - upper(t)$. No scan of the channel is needed.

A rank is an insertion point. A leaderboard is a list of scores in order. A user's rank is the number of scores above theirs, which is one bound search in the sorted list rather than a pass over all users.

Screening needs fast membership tests. Every link posted in the community is screened by a rule floor that combines several signals before an optional local-AI opinion is consulted. Any screen of this kind has to answer questions of the form "have we seen this before?" many times per second, and the difference between a scan of a list and a search of a sorted one — n probes against $\lfloor \lg n \rfloor + 1$ — is the difference between a screen that keeps up with the community and one that does not. (Chapter 15 shows the hash table, which answers membership questions in $O(1)$ probes on average.)

The pattern to take away is general: **keep data in an order that matches the questions you will ask of it, and every question becomes a bound search.** Profitina's actual implementation details are its own; the *shape* of these problems — sorted history, count-since, rank, membership — is universal.

2.13.2 The worked case study: Profitina Laundry

Our runnable case study is **Profitina Laundry**, a small laundry service that records each day's orders: an order number, the time it was received, the customer's first name and the weight in kilograms. Orders arrive in time order and receive consecutive numbers, so the day's list is sorted by time *and* by number at no cost at all — the data is sorted because of how it is produced. **Listing 2.31** answers four everyday questions with it: the orders taken between 10:00 and 12:00 and their total weight (two lower bounds), the lookup of an order by number (one lower bound and an equality test), the order closest to 13:00 (a lower bound and a comparison with its left neighbor), and the first order of a given customer — by name, which is *not* sorted, so it needs a sequential scan. **Listing 2.32** and **Listing 2.33** are the Java and Python versions.

Listing 2.31 `laundry_orders.cpp`: Profitina Laundry's orders searched by time range, by number, by nearest time and by name.

```cpp
// Profitina Laundry: one day's orders, kept in the order they arrived.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

struct Order {
    int number;
    string time;  // "HH:MM"
    string customer;
    double kg;
};

const vector<Order> ORDERS = {
        {1001, "07:45", "Sari", 3.5},  {1002, "08:10", "Budi", 5.0},
        {1003, "08:55", "Dewi", 2.0},  {1004, "09:30", "Andi", 7.5},
        {1005, "10:05", "Rina", 4.0},  {1006, "10:40", "Joko", 6.0},
        {1007, "11:15", "Maya", 3.0},  {1008, "11:50", "Tono", 8.0},
        {1009, "12:20", "Lina", 2.5},  {1010, "13:05", "Agus", 5.5},
        {1011, "14:30", "Fitri", 4.5}, {1012, "15:10", "Hadi", 3.0}};

int minutes(const string& hhmm) {
    return stoi(hhmm.substr(0, 2)) * 60 + stoi(hhmm.substr(3));
}

// First index whose value is >= key; a is sorted ascending.
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;
        else
            high = mid;
    }
    return low;
}
```

Listing 2.31 (Continued) *laundry_orders.cpp: Profitina Laundry's orders searched by time range, by number, by nearest time and by name.*

```
int main() {
    vector<int> times, numbers;
    for (const Order& o : ORDERS) {
        // ascending: orders arrive in time order
        times.push_back(minutes(o.time));
        // ascending too: numbers are issued in turn
        numbers.push_back(o.number);
    }

    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);

    for (int number : {1009, 1020}) {
        int i = lowerBound(numbers, number);
        if (i < (int)numbers.size() && numbers[i] == number)
            printf("order #%d: %s at %s, %.1f kg\n", number,
                   ORDERS[i].customer.c_str(), ORDERS[i].time.c_str(),
                   ORDERS[i].kg);
        else
            printf("order #%d: no such order today\n", number);
    }

    int t = minutes("13:00");
    int i = lowerBound(times, t);  // first order at or after 13:00
    if (i > 0 && (i == (int)times.size() || t - times[i - 1] <= times[i] - t))
        i--;  // the one just before is closer
    printf("order closest to 13:00: #%d at %s (%s)\n", ORDERS[i].number,
           ORDERS[i].time.c_str(), ORDERS[i].customer.c_str());

    string name = "Maya";  // names are not sorted: scan them
    for (const Order& o : ORDERS)
        if (o.customer == name) {
            printf("first order by %s: #%d at %s\n", name.c_str(), o.number,
                   o.time.c_str());
            break;
        }
    return 0;
}
```

Listing 2.32 `LaundryOrders.java`: the same program in Java.

```java
// Profitina Laundry: one day's orders, kept in the order they arrived.
import java.util.List;
import java.util.Locale;

public class LaundryOrders {
    record Order(int number, String time, String customer, double kg) {}

    static final List<Order> ORDERS = List.of(new Order(1001, "07:45", "Sari", 3.5),
            new Order(1002, "08:10", "Budi", 5.0),
            new Order(1003, "08:55", "Dewi", 2.0),
            new Order(1004, "09:30", "Andi", 7.5),
            new Order(1005, "10:05", "Rina", 4.0),
            new Order(1006, "10:40", "Joko", 6.0),
            new Order(1007, "11:15", "Maya", 3.0),
            new Order(1008, "11:50", "Tono", 8.0),
            new Order(1009, "12:20", "Lina", 2.5),
            new Order(1010, "13:05", "Agus", 5.5),
            new Order(1011, "14:30", "Fitri", 4.5),
            new Order(1012, "15:10", "Hadi", 3.0));

    static int minutes(String hhmm) {
        return Integer.parseInt(hhmm.substring(0, 2)) * 60 +
                Integer.parseInt(hhmm.substring(3));
    }

    // First index whose value is >= key; a is sorted ascending.
    static int lowerBound(int[] a, int key) {
        int low = 0, high = a.length;
        while (low < high) {
            int mid = low + (high - low) / 2;
            if (a[mid] < key)
                low = mid + 1;
            else
                high = mid;
        }
        return low;
    }
```

Listing 2.32 (Continued) *LaundryOrders.java: the same program in Java.*

```
    public static void main(String[] args) {
        Locale.setDefault(Locale.ROOT);  // print 21.0, never 21,0
        int n = ORDERS.size();
        int[] times = new int[n], numbers = new int[n];
        for (int i = 0; i < n; i++) {
            // ascending: orders arrive in time order
            times[i] = minutes(ORDERS.get(i).time());
            // ascending too: numbers are issued in turn
            numbers[i] = ORDERS.get(i).number();
        }

        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.get(i).kg();
        System.out.printf("orders from 10:00 to before 12:00: %d orders, %.1f kg%n",
                stop - first, total);
        for (int i = first; i < stop; i++) {
            Order o = ORDERS.get(i);
            System.out.printf("  #%d %s %-5s %.1f kg%n", o.number(), o.time(),
                    o.customer(), o.kg());
        }

        for (int number : new int[] {1009, 1020}) {
            int i = lowerBound(numbers, number);
            if (i < n && numbers[i] == number) {
                Order o = ORDERS.get(i);
                System.out.printf("order #%d: %s at %s, %.1f kg%n", number,
                        o.customer(), o.time(), o.kg());
            } else {
                System.out.printf("order #%d: no such order today%n", number);
            }
        }

        int t = minutes("13:00");
        int i = lowerBound(times, t);  // first order at or after 13:00
        if (i > 0 && (i == n || t - times[i - 1] <= times[i] - t))
            i--;  // the one just before is closer
        Order c = ORDERS.get(i);
        System.out.printf("order closest to 13:00: #%d at %s (%s)%n", c.number(),
                c.time(), c.customer());
```

Listing 2.32 (Continued) *LaundryOrders.java: the same program in Java.*

```
        // names are not sorted: scan them
        String name = "Maya";
        for (Order o : ORDERS)
            if (o.customer().equals(name)) {
                System.out.printf("first order by %s: #%d at %s%n", name,
                        o.number(), o.time());
                break;
            }
    }
}
```

Listing 2.33 `laundry_orders.py`: the same program in Python.

```
"""Profitina Laundry: one day's orders, kept in the order they arrived."""

ORDERS = [
    (1001, "07:45", "Sari", 3.5),
    (1002, "08:10", "Budi", 5.0),
    (1003, "08:55", "Dewi", 2.0),
    (1004, "09:30", "Andi", 7.5),
    (1005, "10:05", "Rina", 4.0),
    (1006, "10:40", "Joko", 6.0),
    (1007, "11:15", "Maya", 3.0),
    (1008, "11:50", "Tono", 8.0),
    (1009, "12:20", "Lina", 2.5),
    (1010, "13:05", "Agus", 5.5),
    (1011, "14:30", "Fitri", 4.5),
    (1012, "15:10", "Hadi", 3.0),
]


def minutes(hhmm):
    return int(hhmm[:2]) * 60 + int(hhmm[3:])


def lower_bound(a, key):
    """First index whose value is >= key; a is sorted ascending."""
    low, high = 0, len(a)
    while low < high:
        mid = low + (high - low) // 2
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid
    return low


def main():
    # ascending: orders arrive in time order
    times = [minutes(o[1]) for o in ORDERS]
    # ascending too: numbers are issued in turn
    numbers = [o[0] for o in ORDERS]
```

Listing 2.33 (Continued) *laundry_orders.py: the same program in Python.*

```
    first, stop = lower_bound(times, minutes("10:00")), lower_bound(
        times, minutes("12:00")
    )
    total = sum(ORDERS[i][3] for i in range(first, stop))
    print(
        "orders from 10:00 to before 12:00: %d orders, %.1f kg"
        % (stop - first, total)
    )
    for i in range(first, stop):
        print("  #%d %s %-5s %.1f kg" % ORDERS[i])

    for number in (1009, 1020):
        i = lower_bound(numbers, number)
        if i < len(numbers) and numbers[i] == number:
            print(
                "order #%d: %s at %s, %.1f kg"
                % (number, ORDERS[i][2], ORDERS[i][1], ORDERS[i][3])
            )
        else:
            print("order #%d: no such order today" % number)

    t = minutes("13:00")
    i = lower_bound(times, t)  # first order at or after 13:00
    if i > 0 and (i == len(times) or t - times[i - 1] <= times[i] - t):
        i -= 1  # the one just before is closer
    print(
        "order closest to 13:00: #%d at %s (%s)"
        % (ORDERS[i][0], ORDERS[i][1], ORDERS[i][2])
    )

    name = "Maya"  # names are not sorted: scan them
    for o in ORDERS:
        if o[2] == name:
            print("first order by %s: #%d at %s" % (name, o[0], o[1]))
            break


if __name__ == "__main__":
    main()
```

All three print:

```
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
order #1009: Lina at 12:20, 2.5 kg
order #1020: no such order today
order closest to 13:00: #1010 at 13:05 (Agus)
first order by Maya: #1007 at 11:15
```

Two design choices in this program are worth copying. The first: times are converted to minutes since midnight, a number, before searching, because comparing `"9:30"` with `"10:05"` as text gives the wrong order (the character `9` comes after `1`); with a leading zero, `"09:30"`, text comparison happens to work, and the program keeps the leading zeros, but converting to numbers removes the doubt. The second: customers are identified here by first name only, and the search by name is a plain scan over one day's orders. A real system would search by a customer identifier, not by a personal field, and under Indonesia's personal-data law, UU PDP 27/2022, would keep personal fields out of sort keys, logs and caches, so that a customer's data can be found and deleted on request.

2.14 Common Mistakes, and How to Avoid Them

1. **Binary search on unsorted data.** It returns wrong answers silently (Section 2.5.6). If you did not sort the data yourself, check it, or search sequentially.
2. **Mixing interval conventions.** A closed-interval initialization with a half-open loop condition, or the reverse. Choose one column of **Table 2.2** and take every line from it.
3. **Updating a boundary to `mid` on a closed interval.** `low = mid` or `high = mid` can leave the interval unchanged and loop forever. On a closed interval both updates move past `mid`.
4. **Computing `(low + high) / 2`.** Overflows for large indices in C, C++ and Java. Write `low + (high - low) / 2` everywhere.
5. **Expecting the first copy from a plain binary search.** `Arrays.binarySearch` and **Pseudocode 2.3** return *some* copy. Use lower bound for the first copy, upper bound minus one for the last.
6. **Misreading Java's negative return value.** A negative result is not "−1 means absent"; it encodes the insertion point as $-(ip) - 1$. Test `r >= 0` for presence and compute `-r - 1` for the position.
7. **Returning a flag instead of a position.** A search that only prints FOUND forces the caller to search again. Return the index, and -1 for absent.
8. **Stopping early when counting.** To count copies or list all of them, the sequential loop must run to the end — or, on sorted data, use upper bound minus lower bound.
9. **Forgetting to restore the sentinel cell, or using a sentinel on shared or read-only data.** Save the spare cell, restore it, and never plant a sentinel in an array another thread may be reading.
10. **Interpolation search without its guards.** Without the range check $A[low] \le x \le A[high]$ the estimate can point outside the array; without the $A[low] = A[high]$ check it divides by zero; and on large keys the product $(x - A[low])(high - low)$ can overflow an `int`. The original C program that inspired Section 2.7 used a `do … while` loop that probed before checking the range, and needed exactly these repairs.
11. **Trusting interpolation search on unmeasured data.** On the real catalog it was 17 times worse than binary search. Measure your key distribution before you rely on it.
12. **Adding an element count to an address.** The address of $A[7]$ is $base + 7w$, not $base + 7$. Multiply by the element width.
13. **Scanning a two-dimensional array column by column in C++, Java or Python.** The answer is the same, but the memory order of **Figure 2.8** makes the row-by-row loop faster. Put the column loop inside.
14. **Comparing times or numbers as text.** `"9:30" > "10:05"` as strings. Convert to numbers first, or zero-pad consistently.
15. **Believing probe counts are times.** They predict growth, not speed. Measure, as Section 2.10 did.

Chapter 4 works problems of exactly this kind, several of them built around searching the Pustaka Ceria catalog.

2.15 Chapter Summary

- The **searching problem** asks for the *position* of a key in a collection, or a clear "absent" (-1). We measure cost in **probes**, inspections of one element.

- **Sequential search** works on any data and costs $(n+1)/2$ probes on average for a present key and n for an absent one: $O(n)$. It is the only choice for unsorted data and for data read front to back. To count copies, it must not stop early.
- A **sentinel** — a copy of the key in a spare cell after the data — removes the bound test from the loop, halving the tests but not the probes. On modern compilers and processors the measured gain was nil in C++ and Java and about 9% in Python.
- **Binary search** on sorted data halves the interval with each probe. Its invariant — everything left of low is smaller, everything right of $high$ is larger — proves it correct, and $T(n) = \lfloor \lg n \rfloor + 1$ bounds its probes: 14 for ten thousand elements, 30 for a billion. It exists in iterative and recursive forms.
- Binary search is easy to get wrong: use one interval convention consistently, make every update shrink the interval, compute the middle as $low + \lfloor (high - low)/2 \rfloor$, and do not assume which copy of a duplicated key you get.
- **Lower bound** and **upper bound** find the first element $\geq x$ and the first $> x$. Between them they answer presence, first and last copy, count, insertion point and range counts.
- **Interpolation search** estimates the position from the key's value. On evenly spread keys it averages about $\lg\lg n$ probes; on bunched keys it degrades toward $O(n)$ — on a real catalog it needed 211 probes where binary search needed 12.
- **Jump search** jumps $\sqrt{n}$ places at a time and walks one block: at most about $2\sqrt{n}$ probes, for storage where backward moves are expensive.
- Two-dimensional arrays are stored in **row-major** order in C and C++: $address(B[i][j]) = base + (iC + j)w$. An unsorted grid is scanned; a grid with sorted rows and columns is searched by the $O(R + C)$ **staircase**; a grid sorted in row-major order is binary-searched as one line.
- **Measure**: probe counts predict growth, but caches, branch prediction and the cost of division decide speed. Binary search in Python still beat sequential search in C++.
- The standard libraries provide all of this: `std::find`, `std::binary_search`, `std::lower_bound`, `std::upper_bound`, `std::equal_range` in C++; `indexOf`, `Arrays.binarySearch` and `Collections.binarySearch` (negative result $= -(ip) - 1$) in Java; `in`, `list.index`, `bisect_left`, `bisect_right` and `insort` in Python.
- No algorithm that only compares can beat $\lceil \lg(n+1) \rceil$ probes in the worst case, so binary search is optimal among comparison searches; beating it requires arithmetic on keys (interpolation) or hashing (Chapter 15).

2.15.1 Looking ahead

Binary search, interpolation search and jump search all assumed that someone had already sorted the data. Chapter 3 is about that someone: it builds the classic sorting algorithms, from bubble sort to quicksort and merge sort, measures them on the same catalog, and shows that sorting once so that you can binary-search many times is one of the best bargains in computing.

2.16 Exercises

2.1 Trace the flag version and the early-exit version of sequential search on [8,10,6,−2,11,7,1,100] for the key 11. How many comparisons does each make? Then trace the search for the *last* occurrence of 041 in the loan log [041,088,025,041,063,041,096,088] by scanning from the end. How many comparisons does it make, and why is scanning from the end the right idea here?

2.2 A catalog holds $n = 2{,}000$ records in no particular order, and 70% of the searches are for keys that are present. Use Equation (2.3) to compute the average number of probes of sequential search. How large must p be for the average to fall below 1,200?

2.3 In **Pseudocode 2.2**, what goes wrong if line 6 (restoring the spare cell) is omitted and the same array is later searched with the plain **Pseudocode 2.1** for $n + 1$ elements? Give a concrete example with the array of **Figure 2.2**. Name two situations in which the sentinel technique must not be used at all.

2.4 Trace **Pseudocode 2.3** on $A = [3,9,11,12,15,17,23,31,35]$ for the keys 31 and 4. For each probe give low, $high$, mid and $A[mid]$. Where does low end for the key 4, and what does that index mean?

2.5 Using Equation (2.5), find the worst-case number of probes of binary search for $n = 1$, 2, 7, 8, 1,000, 1,000,000 and 10^9. For which values of n does the worst case increase by one?

2.6 The following C++ function is meant to be a binary search on a closed interval. Find every bug, give an input on which each one shows itself, and correct the function.

```
int search(const std::vector<int>& a, int x) {
    int low = 0, high = a.size();
    while (low < high) {
        int mid = (low + high) / 2;
        if (a[mid] == x) return mid;
        if (a[mid] < x) low = mid;
        else high = mid - 1;
    }
    return -1;
}
```

2.7 Using only a lower-bound function, write expressions for (a) the index of the last copy of x, (b) the number of elements in $[a, b]$ (both ends included) for integer keys, and (c) the number of elements strictly greater than x. Check each on the years of **Figure 2.5** with $x = 2012$, $a = 2001$, $b = 2009$.

2.8 Trace **Pseudocode 2.6** on the shelf of **Table 2.1** for the keys 041 and 090. Give every estimate to two decimal places.

2.9 Show that Equation (2.7) always gives $low \le pos \le high$ when $A[low] \le x \le A[high]$ and $A[low] < A[high]$. Why does this guarantee disappear if the range check in line 3 of **Pseudocode 2.6** is removed?

2.10 For $n = 10{,}000$, evaluate Equation (2.8) for $m = 10$, 50, 100, 200 and 1,000, and confirm that $m = 100$ is best. A variant of jump search replaces the sequential walk inside the block by a binary search. What is its worst case, and what is the best block size for it?

2.11 An array is declared `int A[4][6]` and starts at address 0x2000; an `int` occupies 4 bytes. Compute the address of $A[3][2]$ in row-major order, and the address it would have if the array were stored in column-major order.

2.12 Trace **Pseudocode 2.8** on the grid of **Figure 2.9** for the keys 48 and 26. List the elements probed. Why must the search start at the top-right (or bottom-left) corner, and not at the top-left?

2.13 *Fibonacci search and exponential search.* (a) Fibonacci search [16] splits the interval at Fibonacci numbers instead of halves, so that it needs only addition and subtraction to compute its probe positions. (b) Exponential search [17] probes positions 1,2,4,8, … until it passes the key, and then binary-searches the last stretch; it suits arrays whose length is unknown or unbounded.

Implement both in C++, Java and Python and run them on [3,9,11,12,15,17,23,31,35] for the keys 17, 35, 3 and 10, counting probes. Show that exponential search finds a key at position i with $O(\lg i)$ probes.

2.14 Profitina Laundry wants to know how many orders arrived *before* 09:00 and how many arrived *at or after* 14:00. Write each as an expression in lower bound or upper bound on the `times` array of **Listing 2.31**, and evaluate both by hand. What does `Arrays.binarySearch(times, 600)` return in Java, and what does that value mean?

2.15 In **Table 2.3**, why is the sentinel's average on absent keys 10,001 rather than 10,000? Why did the Python sentinel save time while the C++ one did not? Using Section 2.10.1, explain in your own words why interpolation search needed 211 probes on the real identifiers.

2.16 A web search engine answers a query over billions of pages in a fraction of a second. Explain why it cannot run sequential search over the pages for each query, and describe how an **inverted index** — for each word, the sorted list of the pages that contain it — turns a two-word query into operations of this chapter.

2.17 Solutions to the Exercises

2.1 The flag version always makes 8 comparisons and ends with the flag set to 1. The early-exit version compares 8, 10, 6, −2, 11 and stops: 5 comparisons, index 4. For the last occurrence of 041, scan from index 7 downwards: 088 (index 7), 096 (6), 041 (5) — found after 3 comparisons, index 5. Scanning from the end is right because the *first* match met going backwards is the *last* occurrence, so the loop may stop early; scanning forwards would have to run to the end, remembering the latest match.

2.2 By Equation (2.3) with $n = 2{,}000$ and $p = 0.7$: $C = 0.7 \cdot 2{,}001/2 + 0.3 \cdot 2{,}000 = 700.35 + 600 = 1{,}300.35$ probes. For $C < 1{,}200$ we need $p \cdot 1{,}000.5 + (1-p) \cdot 2{,}000 < 1{,}200$, that is $2{,}000 - 999.5p < 1{,}200$, that is $p > 800/999.5 \approx 0.8004$. More than about 80% of the searches must be successful.

2.3 If the spare cell is not restored, it still holds the last key searched. After a sentinel search for 5 on [3,12,9,−4,21,6], the cell $A[6]$ holds 5; a later plain search over $n + 1 = 7$ cells for 5 then "finds" 5 at index 6, which is not real data. The damage is invisible until someone searches for exactly that key. The technique must not be used on a read-only (constant) array, which cannot receive the sentinel, nor on an array that another thread may be reading or searching at the same time, because for a moment the array contains a value that is not part of the data. It also needs the spare cell to exist, so it cannot be applied to an array that is exactly full.

2.4 Key 31: $(low, high, mid, A[mid]) = (0,8,4,15)$, $31 > 15$, so $low = 5$; then (5,8,6,23), $31 > 23$, so $low = 7$; then (7,8,7,31): found at index 7 after 3 probes. Key 4: (0,8,4,15), $4 < 15$, $high = 3$; (0,3,1,9), $4 < 9$, $high = 0$; (0,0,0,3), $4 > 3$, $low = 1$; now $low = 1 > high = 0$, absent after 3 probes. $low = 1$ is the insertion point: 4 belongs between 3 and 9.

2.5 $\lfloor \lg n \rfloor + 1$: $n = 1$ gives 1; $n = 2$ gives 2; $n = 7$ gives 3; $n = 8$ gives 4; $n = 1{,}000$ gives $9 + 1 = 10$; $n = 10^6$ gives $19 + 1 = 20$; $n = 10^9$ gives $29 + 1 = 30$. The worst case increases by one exactly when n reaches a power of two: at $n = 2,4,8,16, \ldots$

2.6 There are four bugs. (1) The initialization `high = a.size()` and the loop condition `low < high` are half-open, but the update `high = mid - 1` is closed, so an element can be skipped. On [1,2] searching for 1: $mid = 1$, $A[1] = 2 > 1$, $high = 0$, and the loop stops with $low = high = 0$

without ever probing $A[0]$; it returns -1 although 1 is at index 0. (2) The update `low = mid` can leave the interval unchanged. On [1,2,3] searching for 4: $(low, high) = (0,3)$, $mid = 1$, $low = 1$; then $mid = 2$, $low = 2$; then $mid = 2$ again, $low = 2$ again, forever. (3) `(low + high) / 2` overflows for arrays of more than about 2^{30} elements. (4) `a.size()` is unsigned; converting it silently to `int` hides the conversion and draws a warning under `-Wextra`. A correct closed-interval version:

```
int search(const std::vector<int>& a, int x) {
    int low = 0, high = (int)a.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (a[mid] == x) return mid;
        if (a[mid] < x) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

2.7 Write $L(v)$ for the lower bound of v. (a) The last copy of x is at $L(x+1) - 1$ for integer keys (in general, upper bound minus one), provided $L(x+1) > L(x)$. (b) For integer keys, the elements of $[a, b]$ are those $\geq a$ and $< b + 1$: $L(b+1) - L(a)$. (c) Elements greater than x: $n - L(x+1)$. On the years: (a) $L(2013) - 1 = 9 - 1 = 8$, and indeed $A[8] = 2012$ is the last 2012; (b) $L(2010) - L(2001) = 7 - 1 = 6$ (2001, 2001, 2005, 2005, 2005, 2009); (c) $10 - L(2013) = 10 - 9 = 1$ (only 2020).

2.8 Key 041: $pos = 0 + \lfloor (16/71) \cdot 7 \rfloor = \lfloor 1.58 \rfloor = 1$, $A[1] = 034 < 041$, $low = 2$; then $pos = 2 + \lfloor (0/55) \cdot 5 \rfloor = 2 + \lfloor 0.00 \rfloor = 2$, $A[2] = 041$: found after 2 probes. Key 090: $pos = \lfloor (65/71) \cdot 7 \rfloor = \lfloor 6.41 \rfloor = 6$, $A[6] = 088 < 090$, $low = 7$; now $090 < A[7] = 096$, so the range check fails: absent after 1 probe.

2.9 Write $f = (x - A[low])/(A[high] - A[low])$. The denominator is positive. From $A[low] \leq x$ the numerator is ≥ 0, so $f \geq 0$; from $x \leq A[high]$ the numerator is at most the denominator, so $f \leq 1$. Hence $0 \leq f(high - low) \leq high - low$, and flooring keeps it in $[0, high - low]$, so $low \leq pos \leq high$. Without the range check x may be below $A[low]$ (then $f < 0$ and $pos < low$, possibly negative) or above $A[high]$ (then $f > 1$ and $pos > high$, possibly past the end): the program reads outside the array.

2.10 $f(m) = 10{,}000/m + m - 1$: $f(10) = 1{,}000 + 9 = 1{,}009$; $f(50) = 200 + 49 = 249$; $f(100) = 100 + 99 = 199$; $f(200) = 50 + 199 = 249$; $f(1{,}000) = 10 + 999 = 1{,}009$. The minimum is at $m = 100 = \sqrt{10{,}000}$, and the function is symmetric about it in the sense that m and n/m give the same cost. With a binary search inside the block the cost is about $n/m + \lg m$. This decreases as m grows, all the way to $m = n$, where it becomes one jump plus a binary search of the whole array — that is, plain binary search. Jumping is only worth it when backward moves are expensive, and then the walk inside the block is sequential anyway.

2.11 Row-major: element number $3 \cdot 6 + 2 = 20$, offset $20 \cdot 4 = 80 = 0\text{x}50$, address $0\text{x}2050$. Column-major (4 rows): element number $2 \cdot 4 + 3 = 11$, offset $11 \cdot 4 = 44 = 0\text{x}2\text{C}$, address $0\text{x}202\text{C}$.

2.12 Key 48: probe 50 (larger, go left), 40 (smaller, go down), 45 (smaller, go down), 48 — found at (2,3) after 4 probes. Key 26: 50 (left), 40 (left), 30 (left), 20 (smaller, down), 25 (smaller, down), 29 (larger, left), 27 (larger, left); the column index becomes -1: absent after 7 probes. At the top-left corner both moves, right and down, lead to *larger* elements, so a comparison with a larger key does not tell you which way to go. At the top-right corner the two moves lead one to

smaller and one to larger elements, so every comparison decides the direction and eliminates a row or a column; the bottom-left corner has the same property.

2.13 **Listing 2.34**, **Listing 2.35** and **Listing 2.36** implement both searches.

Listing 2.34 `fibonacci_exponential.cpp`: Fibonacci search and exponential search.

```
// Two more searches on sorted data: Fibonacci search and exponential search.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

// Split at Fibonacci numbers instead of halves. Returns the index or -1.
int fibonacciSearch(const vector<int>& a, int key, int& probes) {
    int n = (int)a.size();
    int fib2 = 0, fib1 = 1;  // F(k-2), F(k-1)
    int fib = fib1 + fib2;   // F(k): grow until F(k) >= n
    while (fib < n) {
        fib2 = fib1;
        fib1 = fib;
        fib = fib1 + fib2;
    }
    int offset = -1;  // everything up to offset is < key
    probes = 0;
    while (fib > 1) {
        int i = min(offset + fib2, n - 1);
        probes++;
        if (a[i] < key) {  // drop the front F(k-2) part
            fib = fib1;
            fib1 = fib2;
            fib2 = fib - fib1;
            offset = i;
        } else if (a[i] > key) {  // keep only the front F(k-2) part
            fib = fib2;
            fib1 = fib1 - fib2;
            fib2 = fib - fib1;
        } else {
            return i;
        }
    }
    if (fib1 == 1 && offset + 1 < n) {
        probes++;
        if (a[offset + 1] == key) return offset + 1;
    }
    return -1;
}
```

Listing 2.34 (Continued) *fibonacci_exponential.cpp: Fibonacci search and exponential search.*

```
// Double a bound until it passes key, then binary-search the last stretch.
int exponentialSearch(const vector<int>& a, int key, int& probes) {
    int n = (int)a.size();
    probes = 1;
    if (a[0] == key) return 0;
    int bound = 1;
    while (bound < n) {
        probes++;
        if (a[bound] >= key) break;
        bound *= 2;
    }
    // a[bound / 2] < key already
    int low = bound / 2 + 1, high = min(bound, n - 1);
    while (low <= high) {
        int mid = low + (high - low) / 2;
        probes++;
        if (a[mid] == key) return mid;
        if (a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    vector<int> data = {3, 9, 11, 12, 15, 17, 23, 31, 35};
    printf("data:");
    for (int x : data) printf(" %d", x);
    printf("\n");
    for (int key : {17, 35, 3, 10}) {
        int p, q;
        int i = fibonacciSearch(data, key, p);
        int j = exponentialSearch(data, key, q);
        printf("key %2d: Fibonacci -> %2d (probes: %d), "
               "exponential -> %2d (probes: %d)\n", key, i, p, j, q);
    }
    return 0;
}
```

Listing 2.35 `FibonacciExponential.java`: the same two searches in Java.

```java
// Two more searches on sorted data: Fibonacci search and exponential search.
public class FibonacciExponential {
    // Split at Fibonacci numbers instead of halves. Returns {index or -1, probes}.
    static int[] fibonacciSearch(int[] a, int key) {
        int n = a.length;
        int fib2 = 0, fib1 = 1;  // F(k-2), F(k-1)
        int fib = fib1 + fib2;  // F(k): grow until F(k) >= n
        while (fib < n) {
            fib2 = fib1;
            fib1 = fib;
            fib = fib1 + fib2;
        }
        int offset = -1;  // everything up to offset is < key
        int probes = 0;
        while (fib > 1) {
            int i = Math.min(offset + fib2, n - 1);
            probes++;
            if (a[i] < key) {  // drop the front F(k-2) part
                fib = fib1;
                fib1 = fib2;
                fib2 = fib - fib1;
                offset = i;
            } else if (a[i] > key) {  // keep only the front F(k-2) part
                fib = fib2;
                fib1 = fib1 - fib2;
                fib2 = fib - fib1;
            } else {
                return new int[] {i, probes};
            }
        }
        if (fib1 == 1 && offset + 1 < n) {
            probes++;
            if (a[offset + 1] == key) return new int[] {offset + 1, probes};
        }
        return new int[] {-1, probes};
    }
```

Listing 2.35 (Continued) *FibonacciExponential.java: the same two searches in Java.*

```
    // Double a bound until it passes key, then binary-search the last stretch.
    static int[] exponentialSearch(int[] a, int key) {
        int n = a.length;
        int probes = 1;
        if (a[0] == key) return new int[] {0, probes};
        int bound = 1;
        while (bound < n) {
            probes++;
            if (a[bound] >= key) break;
            bound *= 2;
        }
        // a[bound / 2] < key already
        int low = bound / 2 + 1, high = Math.min(bound, n - 1);
        while (low <= high) {
            int mid = low + (high - low) / 2;
            probes++;
            if (a[mid] == key) return new int[] {mid, probes};
            if (a[mid] < key)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return new int[] {-1, probes};
    }

    public static void main(String[] args) {
        int[] data = {3, 9, 11, 12, 15, 17, 23, 31, 35};
        StringBuilder line = new StringBuilder("data:");
        for (int x : data) line.append(" ").append(x);
        System.out.println(line);
        for (int key : new int[] {17, 35, 3, 10}) {
            int[] f = fibonacciSearch(data, key);
            int[] e = exponentialSearch(data, key);
            System.out.printf("key %2d: Fibonacci -> %2d (probes: %d), " +
                              "exponential -> %2d (probes: %d)%n",
                    key, f[0], f[1], e[0], e[1]);
        }
    }
}
```

Listing 2.36 `fibonacci_exponential.py`: the same two searches in Python.

```
"""Two more searches on sorted data: Fibonacci search and exponential search."""


def fibonacci_search(a, key):
    """Split at Fibonacci numbers, not halves. Returns (index or -1, probes)."""
    n = len(a)
    fib2, fib1 = 0, 1  # F(k-2), F(k-1)
    fib = fib1 + fib2  # F(k): grow until F(k) >= n
    while fib < n:
        fib2, fib1 = fib1, fib
        fib = fib1 + fib2
    offset = -1  # everything up to offset is < key
    probes = 0
    while fib > 1:
        i = min(offset + fib2, n - 1)
        probes += 1
        if a[i] < key:  # drop the front F(k-2) part
            fib, fib1 = fib1, fib2
            fib2 = fib - fib1
            offset = i
        elif a[i] > key:  # keep only the front F(k-2) part
            fib, fib1 = fib2, fib1 - fib2
            fib2 = fib - fib1
        else:
            return i, probes
    if fib1 == 1 and offset + 1 < n:
        probes += 1
        if a[offset + 1] == key:
            return offset + 1, probes
    return -1, probes


def exponential_search(a, key):
    """Double a bound until it passes key, then binary-search the last stretch."""
    n = len(a)
    probes = 1
    if a[0] == key:
        return 0, probes
    bound = 1
    while bound < n:
        probes += 1
        if a[bound] >= key:
            break
        bound *= 2
    low, high = bound // 2 + 1, min(bound, n - 1)  # a[bound // 2] < key already
    while low <= high:
```

Listing 2.36 (Continued) *fibonacci_exponential.py: the same two searches in Python.*

```
        mid = low + (high - low) // 2
        probes += 1
        if a[mid] == key:
            return mid, probes
        if a[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1, probes


def main():
    data = [3, 9, 11, 12, 15, 17, 23, 31, 35]
    print("data:", " ".join(str(x) for x in data))
    for key in (17, 35, 3, 10):
        i, p = fibonacci_search(data, key)
        j, q = exponential_search(data, key)
        print(
            "key %2d: Fibonacci -> %2d (probes: %d), "
            "exponential -> %2d (probes: %d)" % (key, i, p, j, q)
        )


if __name__ == "__main__":
    main()
```

All three print:

```
data: 3 9 11 12 15 17 23 31 35
key 17: Fibonacci ->  5 (probes: 3), exponential ->  5 (probes: 7)
key 35: Fibonacci ->  8 (probes: 3), exponential ->  8 (probes: 8)
key  3: Fibonacci ->  0 (probes: 3), exponential ->  0 (probes: 1)
key 10: Fibonacci -> -1 (probes: 4), exponential -> -1 (probes: 4)
```

Fibonacci search finds the smallest Fibonacci number $F_k \geq n$ and probes at offsets F_{k-2}; after each comparison the interval shrinks to F_{k-1} or F_{k-2} elements, a ratio of about $1/\varphi \approx 0.618$ per probe rather than $1/2$, so it needs about $1.44 \lg n$ probes in the worst case — slightly more than binary search, but computed with additions only. Exponential search probes $1, 2, 4, \ldots, 2^j$ and stops at the first 2^j with $A[2^j] \geq x$. If the key is at position i, then $2^{j-1} < i$, so $j \leq \lfloor \lg i \rfloor + 1$ doubling probes, followed by a binary search over fewer than $2^{j-1} \leq i$ elements, which costs at most $\lfloor \lg i \rfloor + 1$ more. The total is $O(\lg i)$, independent of n — which is why the method works even when n is unknown, as in the unbounded search of Bentley and Yao [17]. The output shows the effect: the key 3 at position 0 took one probe, while 35 at the far end took eight.

2.14 In minutes, 09:00 is 540 and 14:00 is 840. Orders before 09:00: $L(540) = 3$ (07:45, 08:10, 08:55). Orders at or after 14:00: $n - L(840) = 12 - 10 = 2$ (14:30 and 15:10). `Arrays.binarySearch(times, 600)` looks for 10:00 = 600, which is not an order time; the insertion point is 4 (between 09:30 and 10:05), so the call returns $-(4) - 1 = -5$. It means "no order at exactly 10:00; the first order after 10:00 is at index 4".

2.15 For an absent key the sentinel loop examines all 10,000 real elements and then the sentinel itself, which is the 10,001st probe. In C++ and Java the optimizing compiler and the processor's branch prediction make the bound test `i < n` almost free, so removing it changes nothing measurable; in Python each test is a separate interpreted operation with a real cost, so removing half of them helps. Interpolation search assumes that identifiers rise evenly from the smallest to the largest. In the real catalog half the identifiers lie in the first 1.2% of the range, so the straight

line through the ends of the interval badly underestimates the position of a typical key; each probe lands far to the left, moves *low* only a little, and the next estimate errs in the same way.

2.16 A sequential search over billions of pages would read every page for every query — petabytes of text per second across all users — which is impossible. An inverted index is built once, in advance: for every word, the sorted list of identifiers of the pages containing it. A query for one word is then a search for the word in the sorted vocabulary (binary search, or hashing) followed by reading its list. A query for two words is the intersection of two sorted lists: walk both lists together like the merge step of merge sort (Chapter 3), or, when one list is much shorter, take each identifier of the short list and binary-search or exponential-search it in the long one, costing about $s \lg \ell$ for lists of lengths $s \ll \ell$. Ranking which of the matching pages to show first is a further problem, but finding them is the searching of this chapter at enormous scale.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, secs. 6.1–6.2.

[3] J. Bentley, *Programming Pearls*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2000, column 4.

[4] J. Bloch, "Extra, extra — read all about it: Nearly all binary searches and mergesorts are broken," *Google Research Blog*, Jun. 2, 2006. [Online]. Available: https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/. [Accessed: 24-Sep-2026].

[5] W. W. Peterson, "Addressing for random-access storage," *IBM Journal of Research and Development*, vol. 1, no. 2, pp. 130–146, Apr. 1957, doi: 10.1147/rd.12.0130.

[6] Y. Perl, A. Itai, and H. Avni, "Interpolation search — a log log N search," *Communications of the ACM*, vol. 21, no. 7, pp. 550–553, Jul. 1978, doi: 10.1145/359545.359557.

[7] B. Shneiderman, "Jump searching: A fast sequential search technique," *Communications of the ACM*, vol. 21, no. 10, pp. 831–834, Oct. 1978, doi: 10.1145/359619.359623.

[8] Z. Zajac, "goodbooks-10k: A new dataset for book recommendations," 2017. [Online]. Available: https://github.com/zygmuntz/goodbooks-10k. License CC BY-SA 4.0. [Accessed: 24-Sep-2026].

[9] cppreference.com, "std::lower_bound," *C++ reference*. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/lower_bound. [Accessed: 24-Sep-2026].

[10] Standard C++ Foundation, "GCC 16.1 released: C++26 reflection / contracts / safety hardening, C++20 by default, and more!," isocpp.org, Apr. 2026. [Online]. Available: https://isocpp.org/blog/2026/04/gcc-16.1. [Accessed: 24-Sep-2026].

[11] Oracle, "Class Arrays," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html. [Accessed: 24-Sep-2026].

[12] Oracle, "Class Collections," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Collections.html. [Accessed: 24-Sep-2026].

[13] Oracle, "The arrival of Java 27," *Inside.java*, Sep. 15, 2026. [Online]. Available: https://inside.java/2026/09/15/jdk-27-available/. [Accessed: 24-Sep-2026].

[14] Python Software Foundation, "bisect — Array bisection algorithm," *The Python Standard Library*, Python 3.14.7 documentation. [Online]. Available: https://docs.python.org/3/library/bisect.html. [Accessed: 24-Sep-2026].

[15] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

[16] D. E. Ferguson, “Fibonaccian searching,” *Communications of the ACM*, vol. 3, no. 12, p. 648, Dec. 1960, doi: 10.1145/367487.367496.

[17] J. L. Bentley and A. C.-C. Yao, “An almost optimal algorithm for unbounded searching,” *Information Processing Letters*, vol. 5, no. 3, pp. 82–87, Aug. 1976, doi: 10.1016/0020-0190(76)90071-5.

Chapter 3
Sorting

By the end of this chapter you will be able to say precisely what it means for data to be sorted and why so much of computing depends on it; exchange two values safely, in C++ through pointers and in Java and Python in the way those languages allow; trace, by hand and on paper, bubble sort (with its early exit), exchange sort, selection sort, insertion sort, Shell sort, merge sort, quicksort with both Lomuto's and Hoare's partitions, counting sort, radix sort and a first version of heap sort, all on the same six numbers; write each of them from memory in C++, Java and Python; count the comparisons and the swaps each one makes and derive those counts as formulas in n; explain what *stable*, *in-place* and *adaptive* mean and which algorithm has which property; read a best, average and worst case table and defend every entry in it; measure real sorts on a thousand to a million elements and read the resulting chart; and choose, for a real job, between writing your own sort and calling the one your language already provides, knowing what `std::sort`, `std::stable_sort`, Java's `Arrays.sort` and Python's `list.sort` actually do inside in 2026.

3.1 Where We Left Off

Chapter 2 asked Pustaka Ceria's catalog a question, "is this book here, and where?", and answered it in several ways. Two of them, sequential search and sentinel search, walk the shelf from one end to the other and need no preparation at all. Two others, binary search and interpolation search, are dramatically faster: binary search finds one book among a million with about twenty looks instead of up to a million. But both carry a condition that is easy to read past. They work only on data that is **already sorted**. Binary search decides to throw away the whole right half of the shelf because the middle book's code is larger than the one it wants; on an unsorted shelf that conclusion is simply false, and the search quietly returns the wrong answer.

So Chapter 2 left a debt. Its binary-search programs sorted their arrays with the standard library before searching, and promised that a later chapter would explain how sorting is actually done. This chapter pays that debt, and it pays it in full: eleven algorithms, from the simplest one anybody invents to the ones inside the libraries you use every day.

Why eleven, when one good sort would do? Because each one teaches an idea that returns later in the book, and because no single sort is best for every job. Bubble sort teaches the *adjacent swap* and the idea of an *early exit*. Selection sort teaches that the number of comparisons and the number of moves are different costs. Insertion sort teaches *adaptivity*: doing less work when the data is nearly sorted already. Shell sort shows that a small change to a slow algorithm can make it fast. Merge sort and quicksort introduce *divide and conquer*, the design idea behind recursion in Chapter 10. Counting and radix sort show that the famous $n\log n$ barrier binds only algorithms that compare, and heap sort gives a first look at the heap, the structure that Chapter 14 turns into a priority queue.

Throughout the chapter every algorithm sorts the same six numbers,

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

so that you can put the traces side by side and see how differently eleven procedures reach the same answer. We number positions from 0, as C++, Java and Python do, so $A[0] = 22$ and $A[5] = 2$.

3.2 Why Sort, and the Words We Need

3.2.1 What sorting is

To **sort** a collection is to rearrange its items so that they follow a rule of order. For numbers the usual rules are **ascending** order, smallest first, and **descending** order, largest first:

```
random data:  5  6  8  1  3 25 10
ascending:    1  3  5  6  8 10 25
descending:  25 10  8  6  5  3  1
```

Nothing is added and nothing is removed; the same seven numbers appear in each line. Sorting only changes *where* each item sits.

The rule of order does not have to be "smaller number first". Book titles are sorted alphabetically, dates chronologically, laundry orders by weight, search results by relevance. All of these reduce to one question that the sort asks over and over about two items x and y: *should x come before y?* The part of each item that the question looks at is called the **key**. A book might be sorted by the key "title" today and by the key "category" tomorrow; the record is the same, the key is different.

> **DEFINITION.** Given a sequence $a_0, a_1, \ldots, a_{n-1}$ of n items and an order on their keys, **sorting** produces a rearrangement $a'_0, a'_1, \ldots, a'_{n-1}$ of the same items such that

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

> where $\le$ compares keys. Descending order is the same definition with $\ge$ in place of $\le$.

Equation (3.1) says two things at once, and both matter. The output must be *in order*, and the output must be a *rearrangement*: a "sort" that returned six copies of the smallest number would satisfy the first condition and fail the second.

3.2.2 Why it is worth the trouble

Sorting is worth learning because sorted data makes so many other jobs cheap.

- **Searching.** Chapter 2's binary search needs about $\log_2 n$ steps on sorted data and cannot run at all on unsorted data.
- **Finding duplicates.** In a sorted list, equal items stand next to each other, so one pass that compares neighbors finds every duplicate. Without sorting you must compare every item with every other one.
- **Grouping and reporting.** A library report "books per category" is easy once the catalog is sorted by category: each category is one unbroken run.
- **Merging.** Two sorted lists can be combined into one sorted list in a single pass, which is the heart of merge sort (Section 3.8) and of every database that joins two tables.
- **Ranking and the top k.** A leaderboard, the ten heaviest laundry orders, the median price: all are questions about order.
- **Humans read sorted lists.** A telephone book, a dictionary, a class list and a bank statement are all sorted because people cannot scan an unordered list of a thousand entries.

Because sorting is everywhere, it is also one of the most studied problems in computing. Knuth devotes most of a 780-page volume to it [1], and the algorithms in this chapter have been refined for more than seventy years. That long history is good news for us: the ideas are clean, well understood and well documented.

3.2.3 Six families of sorting methods

The algorithms fall into families according to the idea that drives them. **Figure 3.1** groups them; the section numbers say where each is taught.

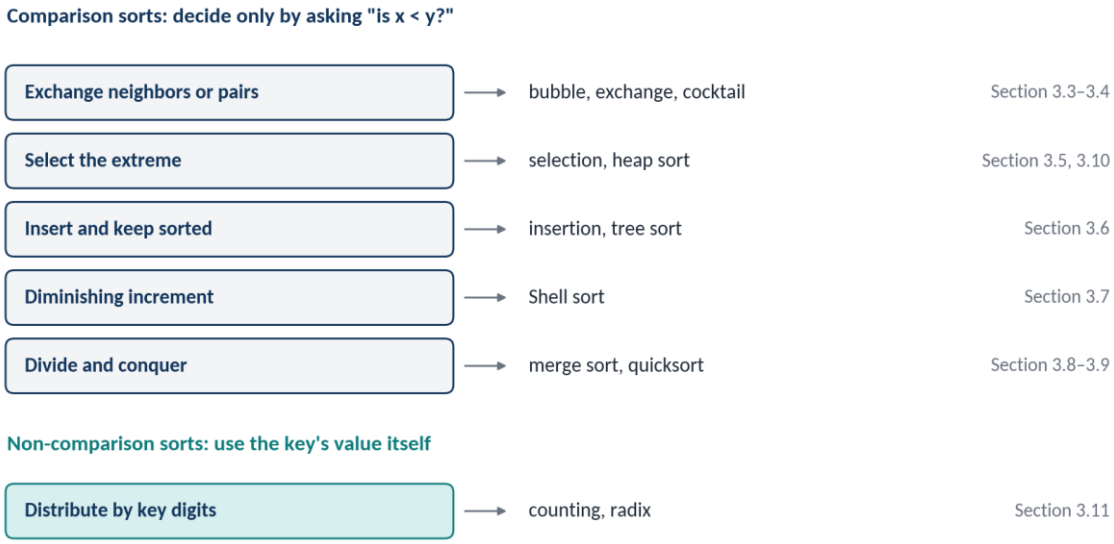


Figure 3.1 The families of sorting methods in this chapter. The first five compare keys; the last uses the digits of the keys directly.

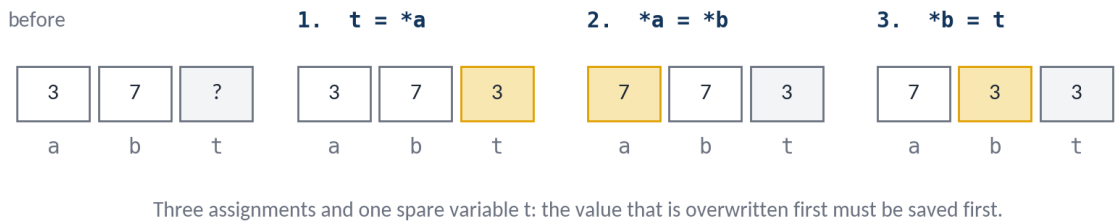
1. **Exchange** methods compare two items and swap them if they are out of order: bubble sort and exchange sort.
2. **Selection** methods repeatedly select the smallest (or largest) remaining item and put it in place: selection sort, and heap sort, which uses a tree to select quickly.
3. **Insertion** methods take items one at a time and insert each into its place among those already sorted: insertion sort, and tree sort, which inserts into a binary search tree (Chapter 11).
4. **Diminishing increment**: Shell sort, which runs insertion sort on items far apart first and then on items closer together.
5. **Divide and conquer**: merge sort and quicksort split the problem into smaller problems of the same kind.
6. **Distribution** methods do not compare keys at all; they place items by the digits or the value of the key: counting sort and radix sort.

3.2.4 The one move every sort needs: exchanging two values

Almost every algorithm in this chapter is built from one small operation: exchange the values in two places. It sounds too simple to discuss, and it is the first thing beginners get wrong. Suppose `a` holds 3 and `b` holds 7, and we write

```
a = b;      // a is now 7
b = a;      // b is now 7 as well: the 3 is gone
```

The first assignment destroyed the only copy of the 3. The fix is a spare variable, traditionally called `t` for “temporary”, that keeps the first value safe while it is overwritten. **Figure 3.2** shows the three assignments.



Three assignments and one spare variable t: the value that is overwritten first must be saved first.

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

In C++ we want a *function* that exchanges two variables belonging to its caller. A function that receives two `int`s receives copies, and exchanging copies changes nothing outside. So the function receives the **addresses** of the two variables instead, as pointers, and works through them: `*a` means "the variable whose address is in `a`". Chapter 1 introduced the address-of operator `&` and the idea of a pointer; Chapter 6 treats pointers in full. **Listing 3.1** is the exchange function that every C++ sort could use, together with a small test.

Listing 3.1 `swap_demo.cpp`: exchanging two `int`s through pointers, and exchanging two elements of an array.

```
// swap_demo.cpp - exchanging two values, the move every sort is built on
#include <cstdio>

void swapInts(int* a, int* b) {        // receives two addresses, not two values
    int t = *a;                         // keep the first value safe
    *a = *b;
    *b = t;
}

void printArray(const char* label, const int data[], int n) {
    std::printf("%s", label);
    for (int i = 0; i < n; ++i) std::printf(" %d", data[i]);
    std::printf("\n");
}

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);

    int data[] = {22, 10, 15, 3, 8, 2};
    int n = sizeof(data) / sizeof(data[0]);
    printArray("data:", data, n);
    // exchange the first and the last element
    swapInts(&data[0], &data[n - 1]);
    printArray("after exchanging data[0] and data[5]:", data, n);
    return 0;
}
```

The call `swapInts(&a, &b)` passes the addresses of `a` and `b`; inside the function, `*a` and `*b` are `a` and `b` themselves. The same function exchanges two array elements, because `&data[0]` is simply the address of the first element. In real C++ you would call the standard `std::swap(x, y)`, which takes its arguments by reference and does exactly the same three assignments; from Section 3.3 onward our C++ programs use it.

Java has no pointers to local variables: every argument is passed by value, so a method that receives two `int`s can never exchange its caller's variables. What a method *can* do is change the contents of an array it receives, because the array is an object and the method receives a

reference to it. **Listing 3.2** shows both facts: `swapInts` has no effect, and `swap(data, i, j)` works.

Listing 3.2 `SwapDemo.java`: a method cannot exchange its caller's `int` variables, but it can exchange two slots of an array.

```
// SwapDemo.java - Java passes values, so swap exchanges two array slots
public class SwapDemo {
    static void swapInts(int a, int b) {  // receives copies: changes are lost
        int t = a;
        a = b;
        b = t;
    }

    // receives the array: changes stay
    static void swap(int[] data, int i, int j) {
        int t = data[i];
        data[i] = data[j];
        data[j] = t;
    }

    static void printArray(String label, int[] data) {
        StringBuilder sb = new StringBuilder(label);
        for (int x : data) sb.append(' ').append(x);
        System.out.println(sb);
    }

    public static void main(String[] args) {
        int a = 3, b = 7;
        System.out.println("before: a = " + a + ", b = " + b);
        swapInts(a, b);                              // no effect on a and b
        int t = a;  // so we exchange them right here
        a = b;
        b = t;
        System.out.println("after:  a = " + a + ", b = " + b);

        int[] data = {22, 10, 15, 3, 8, 2};
        printArray("data:", data);
        // exchange the first and the last element
        swap(data, 0, data.length - 1);
        printArray("after exchanging data[0] and data[5]:", data);
    }
}
```

Python has neither pointers nor primitive variables; a name refers to an object. Its tuple assignment `a, b = b, a` evaluates the right side completely before assigning, so no spare variable is needed, and `data[i], data[j] = data[j], data[i]` exchanges two list elements. **Listing 3.3** is the Python version.

Listing 3.3 `swap_demo.py`: exchanging by tuple assignment, for two names and for two list elements.

```python
# swap_demo.py - Python exchanges by tuple assignment
def swap_ints(a, b):
    """Receives two references; rebinding them does not touch the caller."""
    a, b = b, a


def swap(data, i, j):
    """Receives the list itself: changing its slots is visible to the caller."""
    data[i], data[j] = data[j], data[i]


def print_array(label, data):
    print(label, *data)


a, b = 3, 7
print(f"before: a = {a}, b = {b}")
swap_ints(a, b)                          # no effect on a and b
a, b = b, a                              # the Python idiom: exchange right here
print(f"after:  a = {a}, b = {b}")

data = [22, 10, 15, 3, 8, 2]
print_array("data:", data)
swap(data, 0, len(data) - 1)  # exchange the first and the last element
print_array("after exchanging data[0] and data[5]:", data)
```

All three programs print the same lines:

```
before: a = 3, b = 7
after:  a = 7, b = 3
data: 22 10 15 3 8 2
after exchanging data[0] and data[5]: 2 10 15 3 8 22
```

3.2.5 Counting the work: comparisons and swaps

To compare sorting algorithms we need to measure them, and "time in milliseconds" is a poor first measure: it depends on the computer, the compiler, the language and whatever else the machine is doing. Instead we count the two operations a sort is made of.

- A **comparison** asks "is x before y?" about two keys.
- A **swap** exchanges two items; a **move** (or **shift**) copies one item to another place. A swap is roughly three moves.

These counts do not depend on the machine, and they grow with n in a way we can predict with algebra. Chapter 2 introduced **big-O notation** for exactly this purpose: we say that an algorithm is $O(f(n))$ when its count of steps is at most a constant times $f(n)$ for all large n. In this chapter we meet two growth rates you have not needed before. An algorithm is **quadratic**, $O(n^2)$, when doubling n roughly quadruples the work; it is **linearithmic**, $O(n\log n)$, when doubling n slightly more than doubles it. **Table 3.1** shows why that difference decides everything once n is large. Here $\lg n$ means $\log_2 n$, the number of times n can be halved before it reaches 1.

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

There is one more count that runs through the whole chapter. An **inversion** in a sequence is a pair of positions that are in the wrong order relative to each other.

> **DEFINITION.** An **inversion** of $a_0, \dots, a_{n-1}$ is a pair of indices (i, j) with $i < j$ and $a_i > a_j$. The number of inversions is

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

A sorted sequence has 0 inversions. A sequence in exactly reverse order has every pair inverted, $n(n-1)/2$ of them. Our running array has

$$\underbrace{5}_{22} + \underbrace{3}_{10:\ 3,8,2} + \underbrace{3}_{15:\ 3,8,2} + \underbrace{1}_{3:\ 2} + \underbrace{1}_{8:\ 2} + \underbrace{0}_{2} = 13$$

inversions: 22 is larger than all five numbers after it, 10 is larger than three of the four after it, and so on. Keep the number 13 in mind; it will reappear as the exact number of swaps made by bubble sort and exchange sort and the exact number of shifts made by insertion sort, and Exercise 3.14 counts inversions in $O(n \lg n)$ time with a small change to merge sort.

3.2.6 Three properties: in-place, stable, adaptive

Three more words describe *how* a sort behaves, beyond how fast it is.

> **DEFINITION.** A sort is **in-place** when it uses only a constant amount of memory besides the array itself: a few index variables and a temporary, however large n is. It is **stable** when items with equal keys come out in the same relative order as they went in. It is **adaptive** when it does less work on input that is already nearly sorted.

In-place matters when memory is short or the data is huge. Stability matters whenever a record is sorted by one key after being arranged by another: if Pustaka Ceria's catalog is in order of book id and we then sort it by category, a stable sort keeps the books of each category in id order, and an unstable one may scramble them. Section 3.12 shows exactly that happening. Adaptivity matters because real data is very often *almost* sorted already: yesterday's sorted list plus a few new entries.

3.3 Bubble Sort

3.3.1 The idea

Bubble sort is the method most people invent on their own when asked to sort a row of numbers with the rule "you may only look at two neighbors at a time". Walk along the row from left to right. At each step compare the current element with the next one; if they are in the wrong order, swap them. When you reach the end, start again from the left.

Why does this work? Follow the largest number. The first time the walk reaches it, it is larger than its right neighbor, so it is swapped one place to the right; at its new position it is again larger than its right neighbor, so it is swapped again; and so on until it reaches the last position. One complete walk, called a **pass**, therefore carries the largest element all the way to the end, where it belongs. The name comes from this picture: the large values rise through the array one step at a time like bubbles rising through a glass of soda. Astrachan traces the name and the algorithm's surprisingly long history in [2].

After the first pass the largest element is in place, so the second pass can stop one position earlier; it carries the second-largest element to the second-to-last position. In general, after pass p the last p positions hold the p largest elements in order. That sentence is the **loop invariant** of bubble sort, the fact that is true after every pass and that makes the algorithm correct: after $n-1$ passes the last $n-1$ positions are right, and then the one remaining element, at position 0, must be the smallest.

3.3.2 The early exit

There is one more observation, and it is what separates a careful bubble sort from a careless one. If a whole pass makes **no swap at all**, then every element is already no larger than its right neighbor, and by Equation (3.1) the array is sorted. There is no reason to do the remaining passes. A flag, `swapped`, records whether the current pass swapped anything, and the algorithm stops as soon as a pass ends with the flag still false. **Pseudocode 3.1** writes the whole procedure; indices run from 0 to $n-1$.

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
```

3.3.3 A complete trace

Figure 3.3 runs **Pseudocode 3.1** on $A = [22,10,15,3,8,2]$. Each row shows the array *after* one comparison; the two amber cells are the pair just compared, and the note says whether they were swapped. Teal cells have reached their final positions.

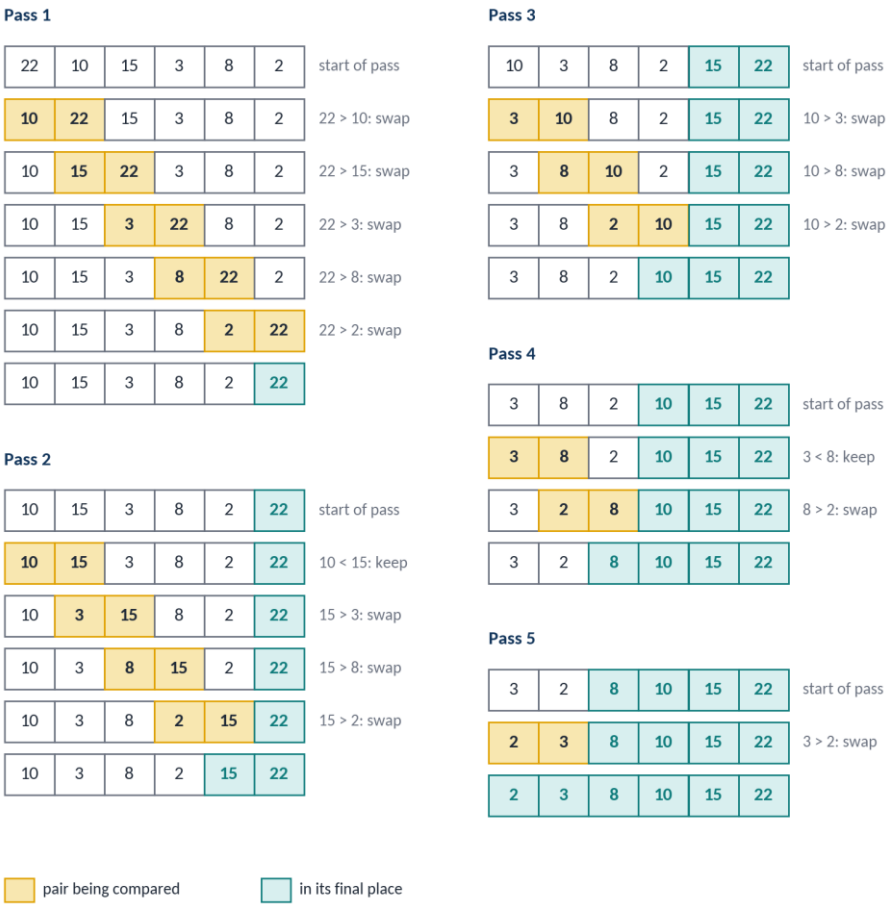


Figure 3.3 Bubble sort on $A = [22,10,15,3,8,2]$, one row per comparison. Pass 1 carries 22 to the end, pass 2 carries 15, and so on; five passes and fifteen comparisons in all.

Read pass 1 slowly. $22 > 10$, swap: [10,22,15,3,8,2]. Now 22 is at index 1 and meets 15: swap. It meets 3, 8 and 2 in turn and is swapped every time, and after five comparisons it sits at index 5. Pass 2 starts again from the left: $10 < 15$, so nothing happens; then 15 travels right past 3, 8 and 2 and stops next to 22. Pass 3 carries 10 to index 3; pass 4 compares 3 with 8 (no swap) and then moves 8 past 2; pass 5 finally swaps 3 and 2. Notice that the small number 2, which started at the far right, moves only *one* step to the left per pass. Large numbers travel quickly to the right and small numbers crawl to the left; Exercise 3.4 builds a variant that fixes this imbalance.

3.3.4 The program

Listing 3.4 implements **Pseudocode 3.1** in C++. The parameter `ascending` selects the direction: for descending order only the comparison changes, from `a[j] > a[j+1]` to `a[j] < a[j+1]`. The function counts its comparisons and swaps and can print the array after every pass. `main` runs it on the running array, then on the sorted result (to see the early exit), on the reversed array (the worst case), and finally in descending order.

Listing 3.4 `bubble_sort.cpp`: bubble sort with early exit, ascending or descending, with counts of comparisons and swaps.

```cpp
// bubble_sort.cpp - bubble sort with early exit, traced pass by pass
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

// Sorts a in ascending order (or descending if ascending == false).
void bubbleSort(vector<int>& a, bool ascending, bool trace) {
    long long comparisons = 0, swaps = 0;
    int n = static_cast<int>(a.size());
    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 (trace)
            cout << "  pass " << pass << ": " << show(a)
                 << (swapped ? "" : "  no swap, stop") << "\n";
        if (!swapped) break;  // early exit: nothing moved
    }
    cout << "  result " << show(a) << ", comparisons = " << comparisons
         << ", swaps = " << swaps << "\n";
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Bubble sort, ascending, on " << show(data) << "\n";
    bubbleSort(data, true, true);

    cout << "Already sorted input " << show(data) << "\n";
    bubbleSort(data, true, true);                    // best case: one pass
```

Listing 3.4 (Continued) *bubble_sort.cpp: bubble sort with early exit, ascending or descending, with counts of comparisons and swaps.*

```cpp
    vector<int> reversed = {22, 15, 10, 8, 3, 2};
    cout << "Reversed input " << show(reversed) << "\n";
    bubbleSort(reversed, true, false);  // worst case: every pair swaps

    vector<int> again = {22, 10, 15, 3, 8, 2};
    cout << "Bubble sort, descending, on " << show(again) << "\n";
    bubbleSort(again, false, false);
    return 0;
}
```

Listing 3.5 is the same program in Java. Java has no `std::swap` for array elements, so the exchange is written out with a temporary `t`, exactly as in **Figure 3.2**.

Listing 3.5 `BubbleSort.java`: the same bubble sort in Java.

```java
// BubbleSort.java - bubble sort with early exit, traced pass by pass
public class BubbleSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    // Sorts a in ascending order (or descending if ascending == false).
    static void bubbleSort(int[] a, boolean ascending, boolean trace) {
        long comparisons = 0, swaps = 0;
        int n = a.length;
        for (int pass = 1; pass < n; pass++) {
            boolean swapped = false;
            // the last pass-1 slots are final
            for (int j = 0; j < n - pass; j++) {
                comparisons++;
                boolean outOfOrder =
                        ascending ? a[j] > a[j + 1] : a[j] < a[j + 1];
                if (outOfOrder) {
                    int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
                    swaps++;
                    swapped = true;
                }
            }
            if (trace)
                System.out.println("  pass " + pass + ": " + show(a)
                        + (swapped ? "" : "  no swap, stop"));
            if (!swapped) break;  // early exit: nothing moved
        }
        System.out.println("  result " + show(a) + ", comparisons = "
                + comparisons
                + ", swaps = " + swaps);
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Bubble sort, ascending, on " + show(data));
        bubbleSort(data, true, true);

        System.out.println("Already sorted input " + show(data));
        bubbleSort(data, true, true);                // best case: one pass
```

Listing 3.5 (Continued) *BubbleSort.java: the same bubble sort in Java.*

```java
        int[] reversed = {22, 15, 10, 8, 3, 2};
        System.out.println("Reversed input " + show(reversed));
        bubbleSort(reversed, true, false);  // worst case: every pair swaps

        int[] again = {22, 10, 15, 3, 8, 2};
        System.out.println("Bubble sort, descending, on " + show(again));
        bubbleSort(again, false, false);
    }
}
```

Listing 3.6 is the Python version. `pass` is a keyword in Python, so the pass counter is called `pass_no`.

Listing 3.6 `bubble_sort.py`: the same bubble sort in Python.

```
# bubble_sort.py - bubble sort with early exit, traced pass by pass
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def bubble_sort(a, ascending=True, trace=False):
    """Sorts the list a in place, ascending (or descending)."""
    comparisons = swaps = 0
    n = len(a)
    for pass_no in range(1, n):
        swapped = False
        for j in range(n - pass_no):     # the last pass_no-1 slots are final
            comparisons += 1
            out_of_order = a[j] > a[j + 1] if ascending else a[j] < a[j + 1]
            if out_of_order:
                a[j], a[j + 1] = a[j + 1], a[j]
                swaps += 1
                swapped = True
        if trace:
            note = "" if swapped else "  no swap, stop"
            print(f"  pass {pass_no}: {show(a)}{note}")
        if not swapped:                  # early exit: nothing moved
            break
    print(f"  result {show(a)}, comparisons = {comparisons}, swaps = {swaps}")


data = [22, 10, 15, 3, 8, 2]
print(f"Bubble sort, ascending, on {show(data)}")
bubble_sort(data, True, True)

print(f"Already sorted input {show(data)}")
bubble_sort(data, True, True)            # best case: one pass

reversed_data = [22, 15, 10, 8, 3, 2]
print(f"Reversed input {show(reversed_data)}")
bubble_sort(reversed_data, True, False)  # worst case: every pair swaps

again = [22, 10, 15, 3, 8, 2]
print(f"Bubble sort, descending, on {show(again)}")
bubble_sort(again, False, False)
```

The three programs print identical output:

```
Bubble sort, ascending, on [22 10 15 3 8 2]
  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
Already sorted input [2 3 8 10 15 22]
  pass 1: [2 3 8 10 15 22]  no swap, stop
  result [2 3 8 10 15 22], comparisons = 5, swaps = 0
Reversed input [22 15 10 8 3 2]
  result [2 3 8 10 15 22], comparisons = 15, swaps = 15
Bubble sort, descending, on [22 10 15 3 8 2]
  result [22 15 10 8 3 2], comparisons = 9, swaps = 2
```

The trace lines match **Figure 3.3** row for row at the end of each pass. The second run shows the early exit at work: an already sorted array costs one pass of $n - 1 = 5$ comparisons and no swaps. The reversed array costs every comparison and every one of them is a swap.

3.3.5 How much work, exactly

Let us count, because the counts are the whole point of this chapter.

Comparisons in the worst case. Pass p compares positions 0 and 1, 1 and 2, …, up to $n-p-1$ and $n-p$: that is $n-p$ comparisons. If no pass ends early, the total is

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

The last step is the classic sum of the first $n-1$ whole numbers: write the sum forwards and backwards, add the two lines column by column, and every column gives n; there are $n-1$ columns, and we counted the sum twice. For $n = 6$ this is $6 \cdot 5/2 = 15$, the number printed by the program. For $n = 100{,}000$ it is almost exactly five billion.

Comparisons in the best case. On sorted input the first pass makes $n-1$ comparisons, swaps nothing, and the algorithm stops. So bubble sort *with the early exit* is adaptive, with a best case of $n-1$ comparisons, $O(n)$. Without the flag it would make $n(n-1)/2$ comparisons even on sorted input.

Swaps, in every case. Here is a beautiful fact. Swapping two *adjacent* elements $a_j > a_{j+1}$ changes the relative order of exactly one pair, that pair itself; every other element is still on the same side of both. So every swap bubble sort makes removes exactly one inversion, and it stops when none are left. Therefore

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

For our array that is 13, as printed; for the reversed array it is $15 = 6 \cdot 5/2$, every pair.

The average case. If all $n!$ orders of n distinct keys are equally likely, each of the $n(n-1)/2$ pairs is inverted in exactly half of them (reversing an order inverts precisely the pairs that were not inverted), so the average number of inversions, and by Equation (3.4) the average number of swaps, is

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

Half the worst case is still quadratic: bubble sort is $\Theta(n^2)$ on average. (The symbol Θ means "grows exactly like", bounded above *and* below by constant multiples.)

> **KEY IDEA.** Any algorithm that only ever swaps *adjacent* elements removes at most one inversion per swap, so by Equation (3.5) it needs $n(n-1)/4$ swaps on average. No amount of cleverness inside that rule can make it faster than quadratic. To beat n^2, an algorithm must move elements over long distances in one step. Shell sort, merge sort and quicksort all do exactly that.

3.3.6 Other forms you will meet

Bubble sort is sometimes written with the inner loop running from right to left, `for j = n - 1 down to pass`, comparing `A[j]` with `A[j-1]`. Then each pass carries the *smallest* remaining element to the *front* instead of the largest to the back. The two forms do exactly the same amount of work. Either form sorts in descending order when the comparison is reversed, and the program above does precisely that with its `ascending` parameter. Bubble sort is in-place (it needs only the temporary of the swap) and stable (it swaps only on a strict `>`, so two equal neighbors are never exchanged).

3.4 Exchange Sort

Exchange sort is bubble sort's close cousin, and many books use the two names interchangeably. The difference lies in *which* pairs are compared, as **Pseudocode 3.2** shows. Bubble sort always compares neighbors, and the element being carried changes as it moves. Exchange sort fixes one position i, which we may call the **pivot** position of the round, and compares it with *every* later position $j = i + 1, \ldots, n - 1$, swapping whenever `A[j]` is smaller than `A[i]`. After the round, position i holds the smallest value of $A[i..n-1]$.

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]
```

Figure 3.4 is the trace. The navy cell is position i; the amber cell is position j, shown after the step.

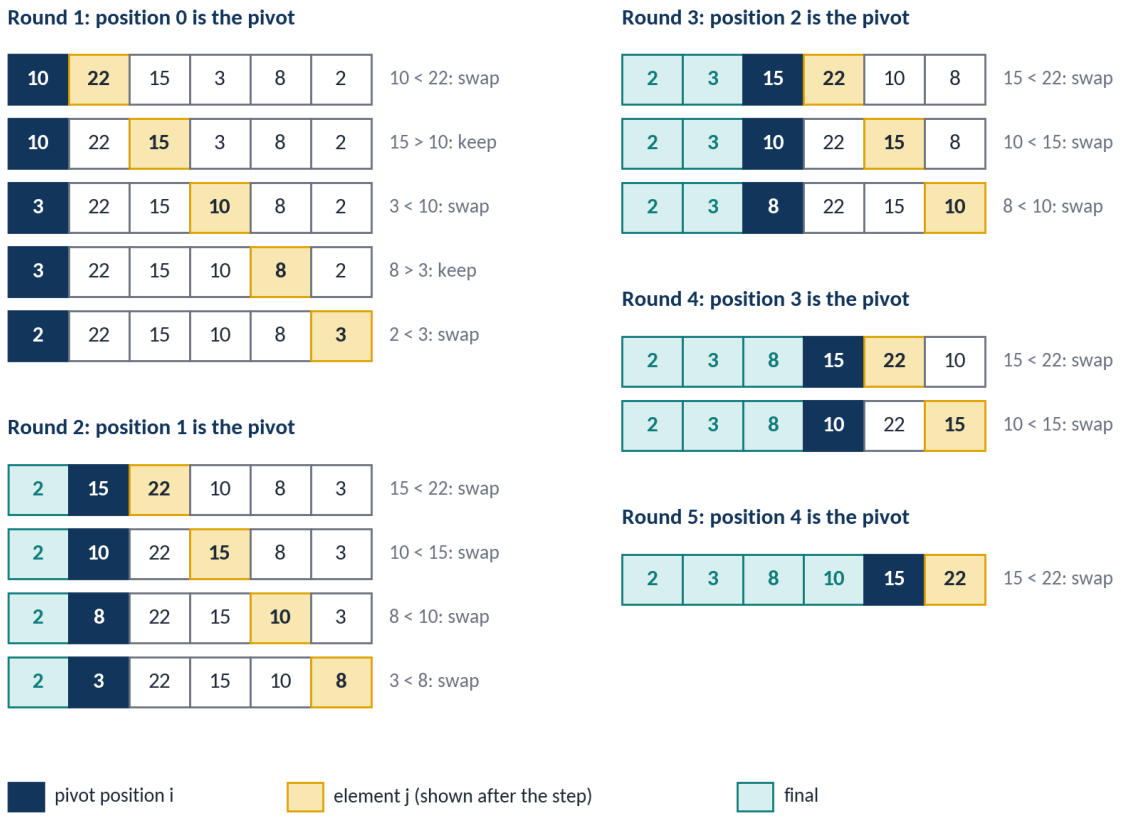


Figure 3.4 Exchange sort on A. In round 1 the value at position 0 changes three times, $22 \to 10 \to 3 \to 2$, before the smallest value settles there.

In round 1, position 0 holds 22; 10 is smaller, so they swap and position 0 holds 10; 15 is not smaller; 3 is, so position 0 now holds 3; 8 is not; 2 is, and position 0 finally holds 2. Each round is a little search for the minimum in which the current candidate is kept at position i itself, paid for with a swap every time a better candidate appears. **Listing 3.7**, **Listing 3.8** and **Listing 3.9** print every swap.

Listing 3.7 `exchange_sort.cpp`: exchange sort, printing each swap and the array after each round.

```cpp
// exchange_sort.cpp - exchange sort: position i meets every later element
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

void exchangeSort(vector<int>& a) {
    long long comparisons = 0, swaps = 0;
    int n = static_cast<int>(a.size());
    for (int i = 0; i < n - 1; ++i) {
        for (int j = i + 1; j < n; ++j) {
            ++comparisons;
            if (a[j] < a[i]) {  // a[i] is the "pivot" of this round
                swap(a[i], a[j]);
                ++swaps;
                cout << "    swap a[" << i << "] and a[" << j << "]: "
                     << show(a) << "\n";
            }
        }
        cout << "  round " << i + 1 << " done: " << show(a) << "\n";
    }
    cout << "  comparisons = " << comparisons << ", swaps = " << swaps << "\n";
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Exchange sort on " << show(data) << "\n";
    exchangeSort(data);
    return 0;
}
```

Listing 3.8 `ExchangeSort.java`: exchange sort in Java.

```java
// ExchangeSort.java - exchange sort: position i meets every later element
public class ExchangeSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static void exchangeSort(int[] a) {
        long comparisons = 0, swaps = 0;
        int n = a.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 1; j < n; j++) {
                comparisons++;
                if (a[j] < a[i]) {  // a[i] is the "pivot" of this round
                    int t = a[i]; a[i] = a[j]; a[j] = t;
                    swaps++;
                    System.out.println("    swap a[" + i + "] and a[" + j
                            + "]: " + show(a));
                }
            }
            System.out.println("  round " + (i + 1) + " done: " + show(a));
        }
        System.out.println("  comparisons = " + comparisons + ", swaps = "
                + swaps);
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Exchange sort on " + show(data));
        exchangeSort(data);
    }
}
```

Listing 3.9 `exchange_sort.py`: exchange sort in Python.

```python
# exchange_sort.py - exchange sort: position i meets every later element
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def exchange_sort(a):
    comparisons = swaps = 0
    n = len(a)
    for i in range(n - 1):
        for j in range(i + 1, n):
            comparisons += 1
            if a[j] < a[i]:              # a[i] is the "pivot" of this round
                a[i], a[j] = a[j], a[i]
                swaps += 1
                print(f"    swap a[{i}] and a[{j}]: {show(a)}")
        print(f"  round {i + 1} done: {show(a)}")
    print(f"  comparisons = {comparisons}, swaps = {swaps}")


data = [22, 10, 15, 3, 8, 2]
print(f"Exchange sort on {show(data)}")
exchange_sort(data)
```

```
Exchange sort on [22 10 15 3 8 2]
    swap a[0] and a[1]: [10 22 15 3 8 2]
    swap a[0] and a[3]: [3 22 15 10 8 2]
    swap a[0] and a[5]: [2 22 15 10 8 3]
  round 1 done: [2 22 15 10 8 3]
    swap a[1] and a[2]: [2 15 22 10 8 3]
    swap a[1] and a[3]: [2 10 22 15 8 3]
    swap a[1] and a[4]: [2 8 22 15 10 3]
    swap a[1] and a[5]: [2 3 22 15 10 8]
  round 2 done: [2 3 22 15 10 8]
    swap a[2] and a[3]: [2 3 15 22 10 8]
    swap a[2] and a[4]: [2 3 10 22 15 8]
    swap a[2] and a[5]: [2 3 8 22 15 10]
  round 3 done: [2 3 8 22 15 10]
    swap a[3] and a[4]: [2 3 8 15 22 10]
    swap a[3] and a[5]: [2 3 8 10 22 15]
  round 4 done: [2 3 8 10 22 15]
    swap a[4] and a[5]: [2 3 8 10 15 22]
  round 5 done: [2 3 8 10 15 22]
  comparisons = 15, swaps = 13
```

The two nested loops always run in full, whatever the data: round i makes $n-1-i$ comparisons, and the total is again $n(n-1)/2$ by the same sum as Equation (3.3). Exchange sort has no early exit and is $\Theta(n^2)$ comparisons in *every* case, even on sorted input. Its swaps happen to be 13 here, like bubble sort's, but that is a coincidence of this array rather than a law: exchange sort swaps elements that are *not* neighbors, so one swap can change many inversions at once, in either direction. In round 2 above, the swap of 15 and 22 fixes one pair but the later swap of 10 with 15 moves 15 back behind 22 on its way. Exchange sort is in-place but **not stable**: a long-distance swap can carry an element past another with an equal key.

3.5 Selection Sort

3.5.1 Search first, swap once

Exchange sort wastes effort: it swaps into position i every temporary winner, only to swap it out again when a better one appears. **Selection sort** removes the waste by separating the two jobs. First it *searches* the unsorted part $A[i..n-1]$ for the position `pos` of its smallest element, remembering only the index; then it makes a *single* exchange, `A[i]` with `A[pos]`. Selection sort is sorting built from searching: each round of **Pseudocode 3.3** is a sequential search for the minimum, exactly the linear scan of Chapter 2.

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]
```

Figure 3.5 traces the six numbers. The amber cell is the smallest found so far in the round; it jumps to the right each time the scan finds something smaller.

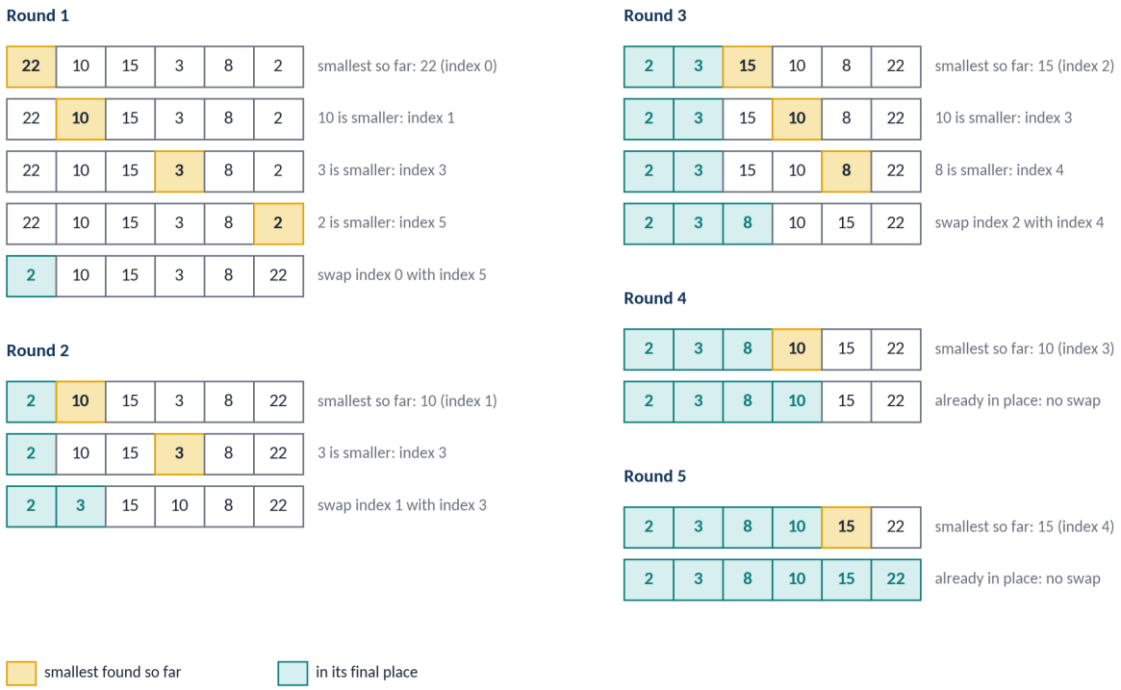


Figure 3.5 Selection sort on *A*. Round 1 finds the minimum 2 at index 5 and makes one exchange; rounds 4 and 5 find the minimum already in place and make none.

In round 1 the candidate starts as 22 at index 0, becomes 10 at index 1, 3 at index 3 and 2 at index 5; one exchange of indices 0 and 5 puts 2 in front. Round 2 searches [10,15,3,8,22], finds 3 at index 3, and exchanges it with index 1. Round 3 finds 8 at index 4 and exchanges it with index 2. In rounds 4 and 5 the smallest element is already at position *i*, and no exchange is made at all.

3.5.2 The program

Listing 3.10 `selection_sort.cpp`: selection sort, reporting where each round found its minimum.

```cpp
// selection_sort.cpp - selection sort: find the smallest, then one exchange
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

void selectionSort(vector<int>& a) {
    long long comparisons = 0, swaps = 0;
    int n = static_cast<int>(a.size());
    for (int i = 0; i < n - 1; ++i) {
        int pos = i;  // index of the smallest so far
        for (int j = i + 1; j < n; ++j) {
            ++comparisons;
            if (a[j] < a[pos]) pos = j;  // only remember; do not move yet
        }
        if (pos != i) {
            swap(a[i], a[pos]);
            ++swaps;
        }
        cout << "  round " << i + 1 << ": smallest is " << a[i]
             << " (was at index " << pos << ") -> " << show(a) << "\n";
    }
    cout << "  comparisons = " << comparisons << ", swaps = " << swaps << "\n";
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Selection sort on " << show(data) << "\n";
    selectionSort(data);
    return 0;
}
```

Listing 3.11 `SelectionSort.java`: selection sort in Java.

```java
// SelectionSort.java - selection sort: find the smallest, then one exchange
public class SelectionSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static void selectionSort(int[] a) {
        long comparisons = 0, swaps = 0;
        int n = a.length;
        for (int i = 0; i < n - 1; i++) {
            int pos = i;  // index of the smallest so far
            for (int j = i + 1; j < n; j++) {
                comparisons++;
                if (a[j] < a[pos]) pos = j;  // only remember; do not move yet
            }
            if (pos != i) {
                int t = a[i]; a[i] = a[pos]; a[pos] = t;
                swaps++;
            }
            System.out.println("  round " + (i + 1) + ": smallest is " + a[i]
                    + " (was at index " + pos + ") -> " + show(a));
        }
        System.out.println("  comparisons = " + comparisons + ", swaps = "
                + swaps);
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Selection sort on " + show(data));
        selectionSort(data);
    }
}
```

Listing 3.12 `selection_sort.py`: selection sort in Python.

```python
# selection_sort.py - selection sort: find the smallest, then one exchange
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def selection_sort(a):
    comparisons = swaps = 0
    n = len(a)
    for i in range(n - 1):
        pos = i                          # index of the smallest so far
        for j in range(i + 1, n):
            comparisons += 1
            if a[j] < a[pos]:
                pos = j                  # only remember; do not move yet
        if pos != i:
            a[i], a[pos] = a[pos], a[i]
            swaps += 1
        print(f"  round {i + 1}: smallest is {a[i]} (was at index {pos}) -> "
              f"{show(a)}")
    print(f"  comparisons = {comparisons}, swaps = {swaps}")


data = [22, 10, 15, 3, 8, 2]
print(f"Selection sort on {show(data)}")
selection_sort(data)
```

Listing 3.10, **Listing 3.11** and **Listing 3.12** all print:

```
Selection sort on [22 10 15 3 8 2]
  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
```

3.5.3 Fewest moves, but never fewer comparisons

The comparisons are exactly those of exchange sort: $n(n-1)/2$ in every case, 15 here, because the inner loop always scans the whole unsorted part. The swaps are what changed: at most one per round, so **at most $n-1$ swaps** in total, and only 3 for our array against 13 for bubble and exchange sort. When moving an item is expensive, such as a large record, or a write to flash memory that wears out with every write, selection sort's $O(n)$ moves are a genuine advantage. When comparisons are what cost, it is as slow as the others, and it is *not adaptive*: a sorted array still costs $n(n-1)/2$ comparisons, because the algorithm has no way to know that nothing needs to move until it has looked at everything.

Selection sort is in-place and, in the form above, **not stable**. The single long-distance exchange of round i can jump `A[i]` over elements equal to it. Section 3.12 demonstrates this on Pustaka Ceria's catalog.

3.6 Insertion Sort

3.6.1 Sorting a hand of cards

Watch someone pick up a hand of playing cards one at a time. The cards already in the left hand are kept in order. Each new card is compared with the cards in the hand from the right-hand end, the larger ones are slid a little to the right to make room, and the new card is dropped into the gap. After the last card is picked up, the hand is sorted. That is **insertion sort**, and it is the sort that people actually use.

In an array the "hand" is the prefix $A[0..i-1]$, which is kept sorted. Step i takes the next element, the **key**, out of position i, leaving a hole there. It then walks left through the prefix: every element larger than the key is shifted one place right, into the hole, which moves the hole one place left. When the walk meets an element that is not larger than the key, or runs off the left end, the key is dropped into the hole.

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
```

The invariant is simple and strong: at the start of step i, the prefix $A[0..i-1]$ holds the elements that were originally there, in sorted order. The step preserves it, and when i reaches n the prefix is the whole array.

Figure 3.6 shows every shift. The navy cell is the key; the pale cell marked with a dot is the hole; the amber cell is the element just shifted into it.

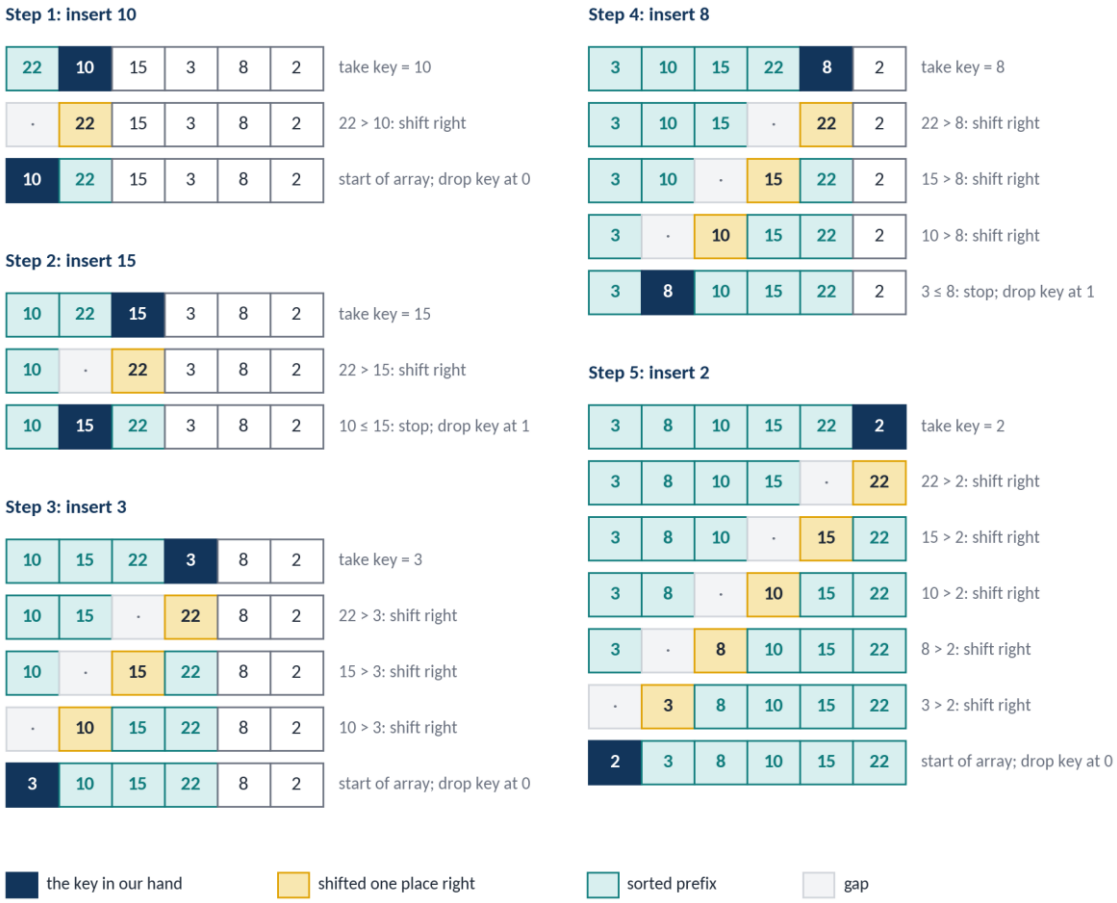


Figure 3.6 Insertion sort on A. Each step takes a key, shifts the larger elements of the sorted prefix one place right, and drops the key into the gap.

Step 1 takes 10; $22 > 10$, so 22 shifts right and 10 drops into index 0. Step 2 takes 15; 22 shifts, but $10 \le 15$ stops the walk, and 15 lands at index 1. Step 3 takes 3, which is smaller than everything in the prefix, so 22, 15 and 10 all shift and 3 lands at the front. Step 4 takes 8: 22, 15 and 10 shift, $3 \le 8$ stops the walk, and 8 lands at index 1. Step 5 takes 2, and all five shift.

3.6.2 A trap in the loop condition

Look again at line 4 of **Pseudocode 3.4**: `j ≥ 0 and A[j] > key`. The order of the two tests matters. When the key is smaller than everything in the prefix, j eventually becomes -1, and at that moment `A[j]` does not exist. In C++, Java and Python the operator `&&` / `and` evaluates its left operand first and does not evaluate the right one at all if the left one is false (this is called *short-circuit evaluation*). Writing `j >= 0` first therefore guarantees that `A[-1]` is never read.

Written the other way round, `while (data[j] > temp && j >= 0)`, the program reads `data[-1]` before it checks. In C++ that is *undefined behavior*: it reads whatever memory lies before the array, and usually seems to work, which is the worst kind of bug. Java throws `ArrayIndexOutOfBoundsException`. Python is sneakier still: `a[-1]` is legal and means *the last element*, so the loop compares the key with the wrong element and may shift it into the wrong place. Our programs write the loop as `while j >= 0` and test `a[j] <= key` inside it, which makes the order impossible to get wrong and also lets us count each comparison.

3.6.3 The program

Listing 3.13 `insertion_sort.cpp`: insertion sort with counts of comparisons and shifts, on random, sorted and reversed input.

```
// insertion_sort.cpp - insertion sort: take a card, shift, drop it in the gap
#include <iostream>
#include <string>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

void insertionSort(vector<int>& a, bool trace) {
    long long comparisons = 0, shifts = 0;
    int n = static_cast<int>(a.size());
    for (int i = 1; i < n; ++i) {
        int key = a[i];                              // the card in our hand
        int j = i - 1;
        while (j >= 0) {  // test j >= 0 BEFORE reading a[j]
            ++comparisons;
            if (a[j] <= key) break;  // equal keys stay put: stable
            a[j + 1] = a[j];  // shift the larger one right
            ++shifts;
            --j;
        }
        a[j + 1] = key;
        if (trace)
            cout << "  insert " << key << " at index " << j + 1 << ": "
                 << show(a) << "\n";
    }
    cout << "  result " << show(a) << ", comparisons = " << comparisons
         << ", shifts = " << shifts << "\n";
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Insertion sort on " << show(data) << "\n";
    insertionSort(data, true);
    cout << "Already sorted input\n";
    insertionSort(data, false);  // best case: n-1 comparisons
    vector<int> reversed = {22, 15, 10, 8, 3, 2};
    cout << "Reversed input\n";
    insertionSort(reversed, false);                  // worst case: n(n-1)/2
    return 0;
}
```

Listing 3.14 `InsertionSort.java`: insertion sort in Java.

```java
// InsertionSort.java - insertion sort: take a card, shift, drop it in the gap
public class InsertionSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static void insertionSort(int[] a, boolean trace) {
        long comparisons = 0, shifts = 0;
        for (int i = 1; i < a.length; i++) {
            int key = a[i];                          // the card in our hand
            int j = i - 1;
            while (j >= 0) {  // test j >= 0 BEFORE reading a[j]
                comparisons++;
                if (a[j] <= key) break;  // equal keys stay put: stable
                a[j + 1] = a[j];  // shift the larger one right
                shifts++;
                j--;
            }
            a[j + 1] = key;
            if (trace)
                System.out.println("  insert " + key + " at index " + (j + 1)
                        + ": " + show(a));
        }
        System.out.println("  result " + show(a) + ", comparisons = "
                + comparisons
                + ", shifts = " + shifts);
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Insertion sort on " + show(data));
        insertionSort(data, true);
        System.out.println("Already sorted input");
        insertionSort(data, false);  // best case: n-1 comparisons
        int[] reversed = {22, 15, 10, 8, 3, 2};
        System.out.println("Reversed input");
        insertionSort(reversed, false);              // worst case: n(n-1)/2
    }
}
```

Listing 3.15 `insertion_sort.py`: insertion sort in Python.

```python
# insertion_sort.py - insertion sort: take a card, shift, drop it in the gap
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


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
        if trace:
            print(f"  insert {key} at index {j + 1}: {show(a)}")
    print(f"  result {show(a)}, comparisons = {comparisons}, shifts = {shifts}")


data = [22, 10, 15, 3, 8, 2]
print(f"Insertion sort on {show(data)}")
insertion_sort(data, True)
print("Already sorted input")
insertion_sort(data)                    # best case: n-1 comparisons
print("Reversed input")
insertion_sort([22, 15, 10, 8, 3, 2])   # worst case: n(n-1)/2
```

Listing 3.13, **Listing 3.14** and **Listing 3.15** print:

```
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]
  result [2 3 8 10 15 22], comparisons = 15, shifts = 13
Already sorted input
  result [2 3 8 10 15 22], comparisons = 5, shifts = 0
Reversed input
  result [2 3 8 10 15 22], comparisons = 15, shifts = 15
```

3.6.4 Why insertion sort is the best of the simple sorts

Each shift moves one element past the key, and the key and that element were an inverted pair; after the step they are in order, and no other pair changed. So, exactly as for bubble sort,

$$\text{shifts made by insertion sort} = \text{inv}(A),$$

13 for our array. Each step also makes one comparison that *stops* the walk, unless the key went all the way to the front. So the comparisons are the inversions plus at most $n-1$: here $13+5-3=15$, since three keys (10, 3 and 2) went to the front and had no stopping comparison. Putting the extremes together:

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

On sorted input there are no inversions and insertion sort makes $n-1$ comparisons and no shifts, as the second run printed. On reversed input it makes $n(n-1)/2$ of both. On random input it makes about $n^2/4$ of each, half the worst case, by Equation (3.5).

The work is proportional to $n + \text{inv}(A)$. That is the precise meaning of "adaptive": insertion sort's cost measures how unsorted the input is. A list of a hundred thousand names with a dozen new names appended costs a few hundred thousand steps, not five billion. Add that it is in-place, stable (the walk stops at an element *equal* to the key, so equal keys never pass each other), tiny, and fast for small n because its inner loop does almost nothing, and you understand why every serious library sort in Section 3.14 finishes its work by running insertion sort on small pieces.

3.7 Shell Sort

3.7.1 Long jumps first

The Key Idea of Section 3.3.5 says that a sort that moves elements only one place at a time is doomed to quadratic time. In 1959 Donald Shell published a way out that keeps insertion sort's simplicity [3]. Instead of comparing each element with its neighbor, first compare it with the element h places away, for some large **gap** h; that is, run insertion sort on each of the h interleaved sub-lists $A[0], A[h], A[2h], \ldots$ and $A[1], A[1+h], \ldots$ and so on. An element that is far from home now travels h places per shift. Then repeat with a smaller gap, and finish with gap 1, which is ordinary insertion sort. By then the array is nearly sorted, and insertion sort, being adaptive, finishes quickly.

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
```

With gap 1, lines 2 to 8 are exactly **Pseudocode 3.4**, so Shell sort is always correct, whatever the other gaps are; the earlier gaps only make the last pass cheaper.

3.7.2 A trace with gaps 3 and 1

Shell's original choice was to start at $h = \lfloor n/2 \rfloor$ and halve until $h = 1$. For $n = 6$ that gives gaps 3 and 1. **Figure 3.7** shows both passes; each color is one sub-list for gap 3.

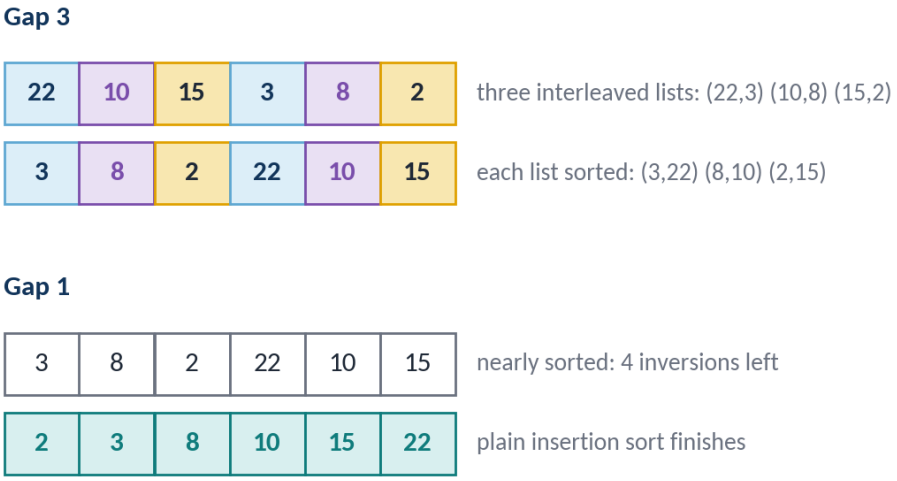


Figure 3.7 Shell sort on A with gaps 3 and 1. Sorting the three interleaved pairs removes 9 of the 13 inversions at once; insertion sort then removes the remaining 4.

With gap 3 the sub-lists are the pairs at indices (0,3), (1,4) and (2,5), holding (22,3), (10,8) and (15,2). Each pair is out of order, and each one shift moves a number three places: 22 jumps from index 0 to index 3 in a single move. The array becomes [3,8,2,22,10,15], which has only 4 inversions, so the closing insertion sort with gap 1 has little to do. The whole sort makes 11 comparisons against 15 for insertion sort alone, a small saving on six numbers but, as the program shows, an enormous one on a hundred thousand.

3.7.3 The gap sequence is everything

The speed of Shell sort depends on the gaps, and finding good ones turned out to be a hard mathematical problem that is still not fully solved [1], [4]. Three sequences are worth knowing.

- **Shell's**: $\lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots, 1$. Simple, but when n is a power of two the even and odd positions are never compared with each other until the last pass, and the worst case is still $\Theta(n^2)$.
- **Knuth's** [1]: 1,4,13,40,121, ..., defined by

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

used from the largest h_k below $n/3$ downward. Its worst case is $O(n^{3/2})$, which for a million elements is a thousand times better than n^2.

- **Ciura's** [4]: 1,4,10,23,57,132,301,701, found in 2001 by computer experiment rather than by proof, and usually extended beyond 701 by multiplying by about 2.25. It is the best sequence known in practice for arrays of up to a few million elements.

Listing 3.16 implements **Pseudocode 3.5** with any list of gaps, traces the six-number example, and then counts the comparisons that each of the three sequences needs on the same 100,000 random numbers. The random numbers come from a tiny generator (a *linear congruential generator*) written out in the program, identical in the three languages, so that all three versions sort exactly the same numbers and print exactly the same counts.

Listing 3.16 `shell_sort.cpp`: Shell sort with three gap sequences, compared on 100,000 random numbers.

```cpp
// shell_sort.cpp - Shell sort: insertion sort over shrinking gaps
#include <cstdint>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

uint64_t state = 2026;  // tiny generator, same in all three languages
int nextRandom(int bound) {
    state = state * 6364136223846793005ULL + 1442695040888963407ULL;
    return static_cast<int>((state >> 33) % static_cast<uint64_t>(bound));
}

// Gapped insertion sort for each gap in turn; the last gap must be 1.
long long shellSort(vector<int>& a, const vector<int>& gaps, bool trace) {
    long long comparisons = 0;
    int n = static_cast<int>(a.size());
    for (int gap : gaps) {
        for (int i = gap; i < n; ++i) {
            int key = a[i];
            int j = i - gap;
            while (j >= 0) {
                ++comparisons;
                if (a[j] <= key) break;
                a[j + gap] = a[j];                  // shift by gap, not by 1
                j -= gap;
            }
            a[j + gap] = key;
        }
        if (trace) cout << "  after gap " << gap << ": " << show(a) << "\n";
    }
    return comparisons;
}

vector<int> shellGaps(int n) {  // n/2, n/4, ..., 1 (Shell, 1959)
    vector<int> g;
    for (int h = n / 2; h >= 1; h /= 2) g.push_back(h);
    return g;
}
```

Listing 3.16 (Continued) *shell_sort.cpp: Shell sort with three gap sequences, compared on 100,000 random numbers.*

```
vector<int> knuthGaps(int n) {  // ..., 40, 13, 4, 1 (Knuth)
    vector<int> g;
    int h = 1;
    while (h < n / 3) h = 3 * h + 1;
    for (; h >= 1; h /= 3) g.push_back(h);
    return g;
}

vector<int> ciuraGaps(int n) {  // 1, 4, 10, 23, ... (Ciura, 2001)
    vector<int> up = {1, 4, 10, 23, 57, 132, 301, 701, 1750};
    while (up.back() * 9 / 4 < n) up.push_back(up.back() * 9 / 4);
    vector<int> g;
    for (int i = static_cast<int>(up.size()) - 1; i >= 0; --i)
        if (up[i] < n) g.push_back(up[i]);
    return g;
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Shell sort on " << show(data) << " with gaps 3, 1\n";
    long long c = shellSort(data, shellGaps(6), true);
    cout << "  comparisons = " << c << "\n";

    int n = 100000;
    vector<int> base(n);
    for (int& x : base) x = nextRandom(1000000);
    cout << "Comparisons on " << n << " random numbers:\n";
    const char* names[] = {"Shell n/2", "Knuth 3h+1", "Ciura"};
    vector<int> gapLists[] = {shellGaps(n), knuthGaps(n), ciuraGaps(n)};
    for (int k = 0; k < 3; ++k) {
        vector<int> a = base;
        long long comps = shellSort(a, gapLists[k], false);
        cout << "  " << names[k] << ": " << gapLists[k].size() << " gaps, "
             << comps << " comparisons\n";
    }
    return 0;
}
```

Listing 3.17 `ShellSort.java`: Shell sort in Java. `>>>` is Java's unsigned shift, which the generator needs.

```java
// ShellSort.java - Shell sort: insertion sort over shrinking gaps
import java.util.ArrayList;
import java.util.List;

public class ShellSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static long state = 2026;  // tiny generator, same in all three languages
    static int nextRandom(int bound) {
        state = state * 6364136223846793005L + 1442695040888963407L;
        return (int) ((state >>> 33) % bound);
    }

    // Gapped insertion sort for each gap in turn; the last gap must be 1.
    static long shellSort(int[] a, List<Integer> gaps, boolean trace) {
        long comparisons = 0;
        int n = a.length;
        for (int gap : gaps) {
            for (int i = gap; i < n; i++) {
                int key = a[i];
                int j = i - gap;
                while (j >= 0) {
                    comparisons++;
                    if (a[j] <= key) break;
                    a[j + gap] = a[j];              // shift by gap, not by 1
                    j -= gap;
                }
                a[j + gap] = key;
            }
            if (trace) System.out.println("  after gap " + gap + ": "
                    + show(a));
        }
        return comparisons;
    }

    static List<Integer> shellGaps(int n) {  // n/2, n/4, ..., 1 (Shell, 1959)
        List<Integer> g = new ArrayList<>();
        for (int h = n / 2; h >= 1; h /= 2) g.add(h);
        return g;
    }
```

Listing 3.17 (Continued) *ShellSort.java: Shell sort in Java. >>> is Java's unsigned shift, which the generator needs.*

```
    static List<Integer> knuthGaps(int n) {  // ..., 40, 13, 4, 1 (Knuth)
        List<Integer> g = new ArrayList<>();
        int h = 1;
        while (h < n / 3) h = 3 * h + 1;
        for (; h >= 1; h /= 3) g.add(h);
        return g;
    }

    static List<Integer> ciuraGaps(int n) {  // 1, 4, 10, 23, ... (Ciura, 2001)
        List<Integer> up =
                new ArrayList<>(List.of(1, 4, 10, 23, 57, 132, 301, 701, 1750));
        while (up.get(up.size() - 1) * 9 / 4 < n)
            up.add(up.get(up.size() - 1) * 9 / 4);
        List<Integer> g = new ArrayList<>();
        for (int i = up.size() - 1; i >= 0; i--)
            if (up.get(i) < n) g.add(up.get(i));
        return g;
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Shell sort on " + show(data) + " with gaps 3, 1");
        long c = shellSort(data, shellGaps(6), true);
        System.out.println("  comparisons = " + c);

        int n = 100000;
        int[] base = new int[n];
        for (int i = 0; i < n; i++) base[i] = nextRandom(1000000);
        System.out.println("Comparisons on " + n + " random numbers:");
        String[] names = {"Shell n/2", "Knuth 3h+1", "Ciura"};
        List<List<Integer>> gapLists =
                List.of(shellGaps(n), knuthGaps(n), ciuraGaps(n));
        for (int k = 0; k < 3; k++) {
            int[] a = base.clone();
            long comps = shellSort(a, gapLists.get(k), false);
            System.out.println("  " + names[k] + ": " + gapLists.get(k).size()
                    + " gaps, "
                    + comps + " comparisons");
        }
    }
}
```

Listing 3.18 `shell_sort.py`: Shell sort in Python. Python integers never overflow, so the generator reduces the result modulo 2 to the power 64 by hand.

```python
# shell_sort.py - Shell sort: insertion sort over shrinking gaps
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


state = 2026  # tiny generator, same in all three languages


def next_random(bound):
    global state
    state = (state * 6364136223846793005 + 1442695040888963407) % 2**64
    return (state >> 33) % bound


def shell_sort(a, gaps, trace=False):
    """Gapped insertion sort for each gap in turn; the last gap must be 1."""
    comparisons = 0
    n = len(a)
    for gap in gaps:
        for i in range(gap, n):
            key = a[i]
            j = i - gap
            while j >= 0:
                comparisons += 1
                if a[j] <= key:
                    break
                a[j + gap] = a[j]        # shift by gap, not by 1
                j -= gap
            a[j + gap] = key
        if trace:
            print(f"  after gap {gap}: {show(a)}")
    return comparisons


def shell_gaps(n):                       # n/2, n/4, ..., 1 (Shell, 1959)
    g, h = [], n // 2
    while h >= 1:
        g.append(h)
        h //= 2
    return g
```

Listing 3.18 (Continued) *shell_sort.py: Shell sort in Python. Python integers never overflow, so the generator reduces the result modulo 2 to the power 64 by hand.*

```
def knuth_gaps(n):                      # ..., 40, 13, 4, 1 (Knuth)
    h = 1
    while h < n // 3:
        h = 3 * h + 1
    g = []
    while h >= 1:
        g.append(h)
        h //= 3
    return g


def ciura_gaps(n):                      # 1, 4, 10, 23, ... (Ciura, 2001)
    up = [1, 4, 10, 23, 57, 132, 301, 701, 1750]
    while up[-1] * 9 // 4 < n:
        up.append(up[-1] * 9 // 4)
    return [h for h in reversed(up) if h < n]


data = [22, 10, 15, 3, 8, 2]
print(f"Shell sort on {show(data)} with gaps 3, 1")
c = shell_sort(data, shell_gaps(6), True)
print(f"  comparisons = {c}")

n = 100000
base = [next_random(1000000) for _ in range(n)]
print(f"Comparisons on {n} random numbers:")
for name, gaps in (("Shell n/2", shell_gaps(n)), ("Knuth 3h+1", knuth_gaps(n)),
                   ("Ciura", ciura_gaps(n))):
    comps = shell_sort(base[:], gaps)
    print(f"  {name}: {len(gaps)} gaps, {comps} comparisons")
```

Listing 3.17 and **Listing 3.18** print the same as **Listing 3.16**:

```
Shell sort on [22 10 15 3 8 2] with gaps 3, 1
  after gap 3: [3 8 2 22 10 15]
  after gap 1: [2 3 8 10 15 22]
  comparisons = 11
Comparisons on 100000 random numbers:
  Shell n/2: 16 gaps, 4488726 comparisons
  Knuth 3h+1: 11 gaps, 3842607 comparisons
  Ciura: 13 gaps, 2554426 comparisons
```

Plain insertion sort on those 100,000 numbers would make about $n^2/4 \approx 2.5$ billion comparisons (Section 3.13 measures 2,500,332,969). Shell's sequence needs 4.5 million, Knuth's 3.8 million and Ciura's 2.6 million: roughly a thousand times less work, from a change of eight lines. Shell sort is in-place and adaptive, but **not stable**, because a jump of h places can pass over an equal key.

3.8 Merge Sort

3.8.1 Divide and conquer

Every sort so far works on the whole array at once. **Merge sort** takes the opposite approach, called **divide and conquer**:

1. **Divide** the array into two halves.
2. **Conquer** each half by sorting it, with the same method, recursively.
3. **Combine** the two sorted halves into one sorted array by **merging** them.

A piece of one element is already sorted, so the recursion stops there. All the real work happens in step 3, and merging is easy. Picture two sorted piles of index cards, smallest on top. Compare the two top cards, take the smaller, put it face down on a new pile, and repeat. Each comparison places one card, and when one pile runs out, the rest of the other pile is simply moved over. Two sorted runs of total length m are merged with at most $m - 1$ comparisons.

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)
```

Ranges are written as half-open intervals: `lo..hi-1` means indices $lo, lo + 1, \ldots, hi - 1$, so the range has $hi - lo$ elements and splits cleanly at `mid`. The midpoint is computed as `lo + (hi - lo)/2` rather than `(lo + hi)/2` because the sum `lo + hi` can overflow an `int` for arrays of more than a billion elements, a bug that lived unnoticed in widely used binary-search and merge-sort code for years.

3.8.2 The trace

Figure 3.8 shows the whole computation on A. The upper half is the division, top down; the lower half is the merging, bottom up.

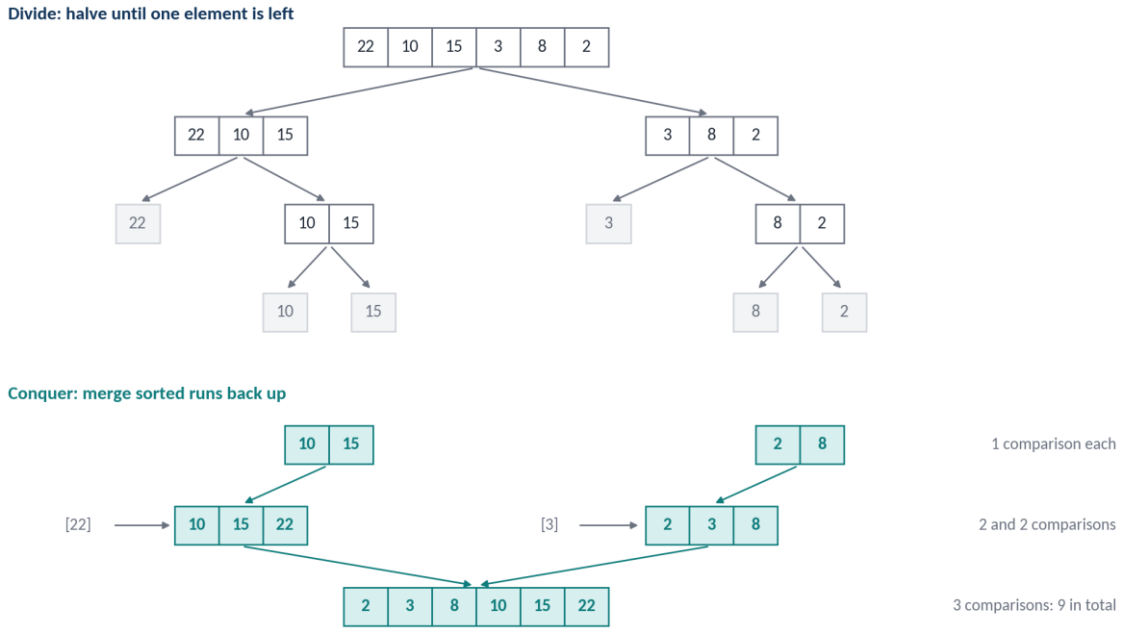


Figure 3.8 Merge sort on *A*. Dividing (top) splits the array until every piece has one element; merging (bottom) combines sorted runs, and the comparisons at each level are noted on the right.

[22,10,15,3,8,2] splits into [22,10,15] and [3,8,2]; the first splits into [22] and [10,15], and [10,15] into [10] and [15]. Merging [10] with [15] costs one comparison and gives [10,15]. Merging [22] with [10,15]: $22 > 10$ takes 10, $22 > 15$ takes 15, the right run is empty, and 22 is copied over: two comparisons, [10,15,22]. The right half does the same, giving [2,3,8]. The last merge compares 10 with 2, 10 with 3 and 10 with 8, taking 2, 3 and 8; the right run is then empty and 10,15,22 are copied without further comparisons. In total $1 + 2 + 1 + 2 + 3 = 9$ comparisons.

3.8.3 The program

Listing 3.19 prints each merge, indented by its depth in the recursion, so the printout has the shape of the lower half of **Figure 3.8**. The strings `left` and `right` are captured *before* `merge` overwrites the range, so that the line can show what was merged.

Listing 3.19 `merge_sort.cpp`: top-down merge sort with one shared buffer, printing every merge.

```cpp
// merge_sort.cpp - top-down merge sort, printing every merge
#include <iostream>
#include <string>
#include <vector>
using namespace std;

string show(const vector<int>& a, int lo, int hi) {  // the slice a[lo..hi-1]
    string s = "[";
    for (int i = lo; i < hi; ++i) s += (i > lo ? " " : "") + to_string(a[i]);
    return s + "]";
}

long long comparisons = 0, moves = 0;

// 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);
    string left = show(a, lo, mid), right = show(a, mid, hi);
    merge(a, tmp, lo, mid, hi);
    cout << string(2 * depth + 2, ' ') << "merge " << left << " + " << right
         << " -> " << show(a, lo, hi) << "\n";
}
```

Listing 3.19 (Continued) *merge_sort.cpp: top-down merge sort with one shared buffer, printing every merge.*

```cpp
int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    vector<int> tmp(data.size());  // one buffer, allocated once
    cout << "Merge sort on " << show(data, 0, 6) << "\n";
    mergeSort(data, tmp, 0, static_cast<int>(data.size()), 0);
    cout << "  result " << show(data, 0, 6) << ", comparisons = " << comparisons
         << ", moves = " << moves << "\n";
    return 0;
}
```

Listing 3.20 `MergeSort.java`: merge sort in Java.

```java
// MergeSort.java - top-down merge sort, printing every merge
public class MergeSort {
    static String show(int[] a, int lo, int hi) {    // the slice a[lo..hi-1]
        StringBuilder sb = new StringBuilder("[");
        for (int i = lo; i < hi; i++)
            sb.append(i > lo ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static long comparisons = 0, moves = 0;

    // Merges sorted runs a[lo..mid-1] and a[mid..hi-1] via the buffer tmp.
    static void merge(int[] a, 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++; }
    }

    static void mergeSort(int[] a, 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);
        String left = show(a, lo, mid), right = show(a, mid, hi);
        merge(a, tmp, lo, mid, hi);
        System.out.println(" ".repeat(2 * depth + 2) + "merge " + left + " + "
                + right
                + " -> " + show(a, lo, hi));
    }
```

Listing 3.20 (Continued) *MergeSort.java: merge sort in Java.*

```java
    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        int[] tmp = new int[data.length];  // one buffer, allocated once
        System.out.println("Merge sort on " + show(data, 0, 6));
        mergeSort(data, tmp, 0, data.length, 0);
        System.out.println("  result " + show(data, 0, 6) + ", comparisons = "
                + comparisons
                + ", moves = " + moves);
    }
}
```

Listing 3.21 `merge_sort.py`: merge sort in Python.

```
# merge_sort.py - top-down merge sort, printing every merge
def show(a, lo=0, hi=None):              # the slice a[lo..hi-1]
    return "[" + " ".join(map(str, a[lo:hi])) + "]"


comparisons = moves = 0


def merge(a, tmp, lo, mid, hi):
    """Merges sorted runs a[lo..mid-1] and a[mid..hi-1] through the buffer."""
    global comparisons, moves
    i, j, k = lo, mid, lo
    while i < mid and j < hi:
        comparisons += 1
        if a[i] <= a[j]:                 # <= takes the left one on ties: stable
            tmp[k] = a[i]
            i += 1
        else:
            tmp[k] = a[j]
            j += 1
        k += 1
    while i < mid:
        tmp[k] = a[i]
        i += 1
        k += 1
    while j < hi:
        tmp[k] = a[j]
        j += 1
        k += 1
    for k in range(lo, hi):
        a[k] = tmp[k]
        moves += 1


def merge_sort(a, tmp, lo, hi, depth=0):
    if hi - lo < 2:                      # 0 or 1 element: already sorted
        return
    mid = lo + (hi - lo) // 2
    merge_sort(a, tmp, lo, mid, depth + 1)
    merge_sort(a, tmp, mid, hi, depth + 1)
    left, right = show(a, lo, mid), show(a, mid, hi)
    merge(a, tmp, lo, mid, hi)
    indent = " " * (2 * depth + 2)
    print(f"{indent}merge {left} + {right} -> {show(a, lo, hi)}")
```

Listing 3.21 (Continued) *merge_sort.py: merge sort in Python.*

```
data = [22, 10, 15, 3, 8, 2]
tmp = [0] * len(data)                    # one buffer, allocated once
print(f"Merge sort on {show(data)}")
merge_sort(data, tmp, 0, len(data))
print(f"  result {show(data)}, comparisons = {comparisons}, moves = {moves}")
```

Listing 3.19, **Listing 3.20** and **Listing 3.21** all print:

```
Merge sort on [22 10 15 3 8 2]
      merge [10] + [15] -> [10 15]
    merge [22] + [10 15] -> [10 15 22]
      merge [8] + [2] -> [2 8]
    merge [3] + [2 8] -> [2 3 8]
  merge [10 15 22] + [2 3 8] -> [2 3 8 10 15 22]
  result [2 3 8 10 15 22], comparisons = 9, moves = 16
```

3.8.4 Why merge sort is $n\lg n$, always

Let $T(n)$ be the number of steps merge sort takes on n elements. It makes two recursive calls on halves and then a merge that touches each of the n elements a constant number of times, so, for some constant c,

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

The easiest way to solve Equation (3.8) is to draw it as a tree. The top call costs cn for its merge. It has two children, each merging $n/2$ elements at a cost of $cn/2$, together cn. They have four children costing $cn/4$ each, together cn again. Every level of the tree costs cn, because the number of pieces doubles exactly as their size halves. The sizes go $n, n/2, n/4, \ldots, 1$, so there are $\lg n$ halvings and $\lg n + 1$ levels. Hence

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

Nothing in that argument depends on the data. Merge sort splits in the middle whether the input is sorted, reversed or random, so it is $\Theta(n \lg n)$ in the best, average *and* worst case. Counting comparisons exactly, the worst case is $n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$ [1]; for $n = 6$ that is $18 - 8 + 1 = 11$, and our array needed only 9.

Merge sort pays for its guarantee with memory: the buffer T holds n elements, so it is **not in-place**, needing $\Theta(n)$ extra space. It is **stable**, because `MERGE` takes from the left run when the keys are equal (line 3 uses $\leq$, not $<$), so equal keys keep their original order. It is not adaptive in the simple form above, but the library sorts of Section 3.14, Timsort and Powersort, are merge sorts made adaptive by finding the sorted runs that already exist in the data.

3.9 Quicksort

3.9.1 Partition, then recurse

Merge sort does trivial work to divide and real work to combine. **Quicksort**, invented by C. A. R. Hoare in 1959 and published in 1962 [5], does the opposite: real work to divide, and nothing at all to combine. Choose one element, the **pivot**. **Partition** the array so that everything not larger than the pivot comes before it and everything larger comes after it. The pivot is now in its final position, the two sides can be sorted independently by recursion, and when they are sorted the whole array is sorted, with no merge.

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)
```

Everything depends on `PARTITION`, and there are two classic ways to write it.

3.9.2 Lomuto's partition

Lomuto's scheme, popularized by Bentley and used in CLRS [6], takes the last element as the pivot x and scans the range once from left to right with an index j. It keeps a boundary i: positions $lo..i$ hold elements $\leq x$, positions $i+1..j-1$ hold elements $> x$, and $j..hi-1$ have not been looked

at yet. When `A[j]` is $\leq x$ it belongs to the left region, so i moves one step right and `A[j]` is exchanged into position i. At the end the pivot is exchanged into position $i + 1$, between the two regions.

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
```

Figure 3.9 traces it. The upper panel is the partition of [10,15,3,8], the most interesting of the four partitions that sort A; the lower panel is the whole recursion.

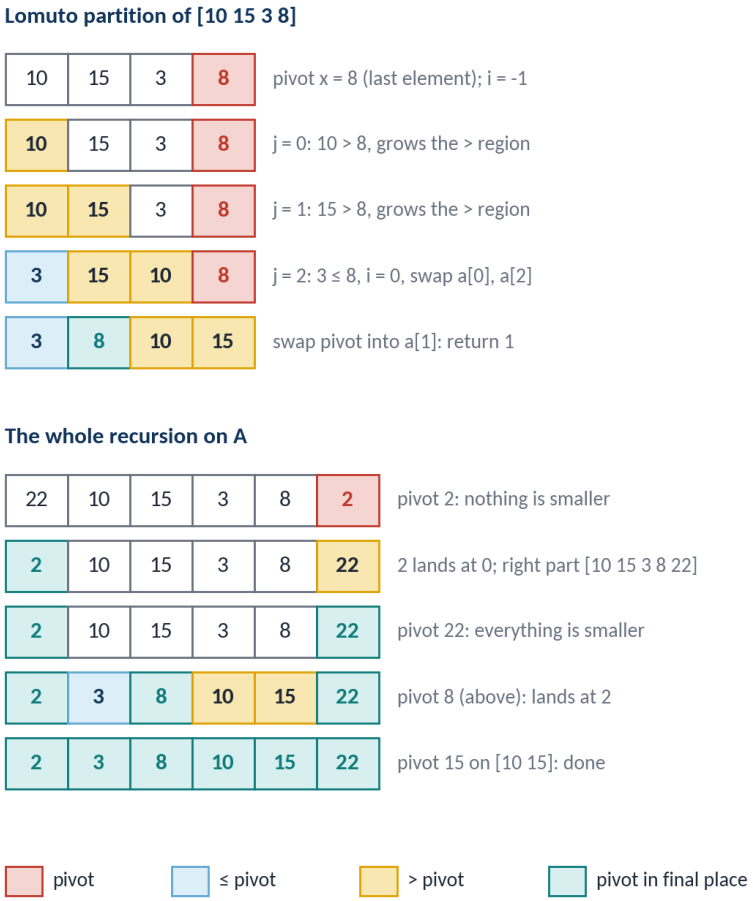


Figure 3.9 Lomuto's partition. Top: partitioning [10,15,3,8] around the pivot 8. Bottom: the four partitions that sort A with the last element as pivot.

In the upper panel, the pivot is 8 and $i = -1$. $10 > 8$ and $15 > 8$ just extend the "greater" region. $3 \leq 8$: i becomes 0 and 3 is exchanged with 10, the first element of the "greater" region, which moves to the end of that region. Finally the pivot is exchanged with `A[i+1] = 15`, giving [3,8,10,15] with 8 at index 1 of the sub-array. In the lower panel you can see the weakness of taking the last element as pivot: the first pivot, 2, is the smallest element, so the partition

produces nothing on the left and five elements on the right; the second pivot, 22, is the largest of those five, and again one side is empty. Only the third pivot, 8, splits evenly.

3.9.3 Hoare's partition

Hoare's original scheme runs two indices toward each other. i moves right from the left end and stops at an element $\geq x$; j moves left from the right end and stops at an element $\leq x$. Both stopped elements are on the wrong side, so they are exchanged, and the walk continues. When the indices cross, everything at or left of j is $\leq x$ and everything right of j is $\geq x$. Our version takes the middle element as the pivot.

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]
```

With Hoare's partition the pivot does *not* necessarily end at index j, so the recursive calls are `QUICKSORT(A, lo, j)` and `QUICKSORT(A, j+1, hi)`, not the calls in **Pseudocode 3.7**. **Figure 3.10** partitions the whole of A around the middle value 15.

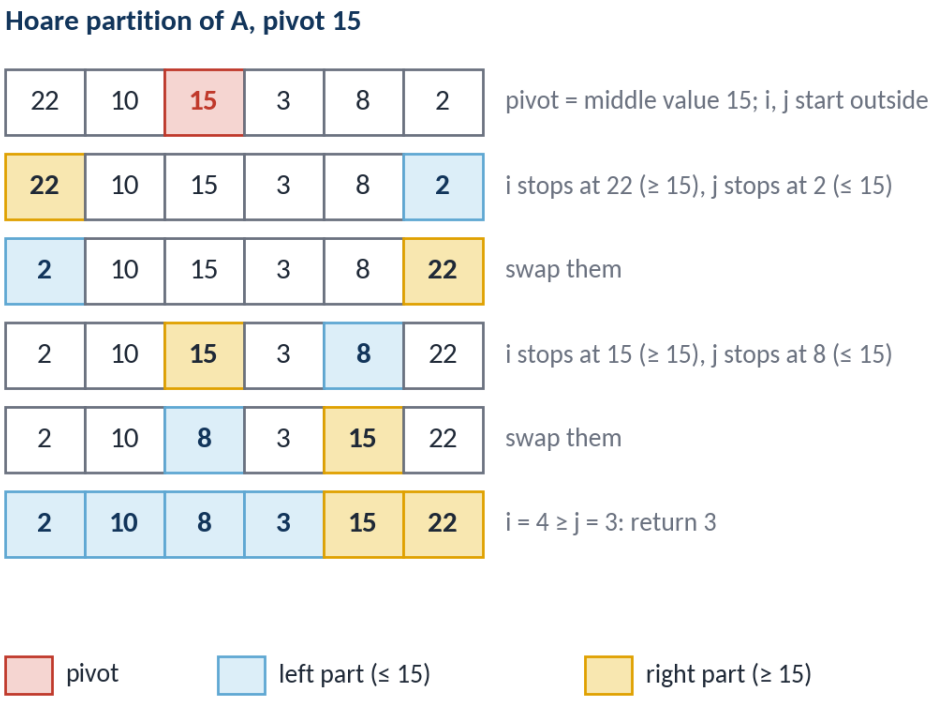


Figure 3.10 Hoare's partition of A with pivot 15. Two exchanges suffice; the indices cross with $j = 3$, splitting A into [2,10,8,3] and [15,22].

i stops at once at $22 \geq 15$ and j stops at once at $2 \leq 15$; they are exchanged, giving [2,10,15,3,8,22]. Then i passes 10 and stops at index 2, on 15 itself (the test is $\geq$, so an element equal to the pivot stops it), and j steps to index 4 and stops at once on 8; exchanged, giving [2,10,8,3,15,22]. On the third round i passes 3 and stops at index 4 on 15, while j stops at index 3

on 3. Now $i = 4 \geq j = 3$: the indices have crossed and the partition returns 3. Hoare's scheme does fewer exchanges than Lomuto's on average, about a third as many [1], and it splits an array of equal keys evenly, where Lomuto's degenerates; that is why library implementations descend from it.

3.9.4 The program, and the choice of pivot

Listing 3.22 contains both partitions. The Lomuto version accepts a **pivot rule**: 0 uses the last element, 1 uses the **median of three** (the middle value of the first, middle and last elements, moved to the end before partitioning), and 2 uses a **random** element. After tracing both schemes on A, the program sorts 3000 elements that are already sorted and 3000 random elements with each rule, counting comparisons and the maximum depth of the recursion.

Listing 3.22 `quick_sort.cpp`: quicksort with Lomuto's and Hoare's partitions, and an experiment with three pivot rules.

```
// quick_sort.cpp - quicksort: Lomuto and Hoare partitions, three pivot rules
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

// the slice a[lo..hi], inclusive
string show(const vector<int>& a, int lo, int hi) {
    string s = "[";
    for (int i = lo; i <= hi; ++i) s += (i > lo ? " " : "") + to_string(a[i]);
    return s + "]";
}

uint64_t state = 2026;
int nextRandom(int bound) {
    state = state * 6364136223846793005ULL + 1442695040888963407ULL;
    return static_cast<int>((state >> 33) % static_cast<uint64_t>(bound));
}

long long comparisons = 0, swaps = 0;
int maxDepth = 0;
bool trace = false;

// 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
}
```

Listing 3.22 (Continued) *quick_sort.cpp: quicksort with Lomuto's and Hoare's partitions, and an experiment with three pivot rules.*

```
// Hoare: pivot = middle value; two indices walk inward and swap.
int hoarePartition(vector<int>& a, int lo, int hi) {
    int pivot = a[lo + (hi - lo) / 2];
    int i = lo - 1, j = hi + 1;
    while (true) {
        do { ++i; ++comparisons; } while (a[i] < pivot);
        do { --j; ++comparisons; } while (a[j] > pivot);
        if (i >= j) return j;  // a[lo..j] <= pivot <= a[j+1..hi]
        swap(a[i], a[j]);
        ++swaps;
    }
}

void quickLomuto(vector<int>& a, int lo, int hi, int rule, int depth) {
    if (lo >= hi) return;
    maxDepth = max(maxDepth, depth);
    if (rule == 1) {  // median of three: a[lo], a[mid], a[hi]
        int mid = lo + (hi - lo) / 2;
        if (a[mid] < a[lo]) swap(a[mid], a[lo]);
        if (a[hi] < a[lo]) swap(a[hi], a[lo]);
        if (a[mid] < a[hi]) swap(a[mid], a[hi]);  // now a[hi] holds the median
    } else if (rule == 2) {                          // random pivot
        swap(a[lo + nextRandom(hi - lo + 1)], a[hi]);
    }
    string before = trace ? show(a, lo, hi) : "";
    int p = lomutoPartition(a, lo, hi);
    if (trace)
        cout << string(2 * depth + 2, ' ') << "partition " << before
             << " pivot " << a[p] << " -> " << show(a, lo, hi)
             << ", pivot at " << p << "\n";
    quickLomuto(a, lo, p - 1, rule, depth + 1);
    quickLomuto(a, p + 1, hi, rule, depth + 1);
}

void quickHoare(vector<int>& a, int lo, int hi, int depth) {
    if (lo >= hi) return;
    string before = show(a, lo, hi);
    int pivot = a[lo + (hi - lo) / 2];
    int p = hoarePartition(a, lo, hi);
    cout << string(2 * depth + 2, ' ') << "partition " << before
         << " pivot " << pivot << " -> " << show(a, lo, p) << " "
         << show(a, p + 1, hi) << "\n";
    quickHoare(a, lo, p, depth + 1);
    quickHoare(a, p + 1, hi, depth + 1);
}
```

Listing 3.22 (Continued) *quick_sort.cpp: quicksort with Lomuto's and Hoare's partitions, and an experiment with three pivot rules.*

```
int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Quicksort, Lomuto partition, last element as pivot:\n";
    trace = true;
    quickLomuto(data, 0, 5, 0, 0);
    cout << "  result " << show(data, 0, 5) << ", comparisons = " << comparisons
         << ", swaps = " << swaps << "\n";

    vector<int> data2 = {22, 10, 15, 3, 8, 2};
    comparisons = swaps = 0;
    cout << "Quicksort, Hoare partition, middle element as pivot:\n";
    quickHoare(data2, 0, 5, 0);
    cout << "  result " << show(data2, 0, 5) << ", comparisons = "
         << comparisons << ", swaps = " << swaps << "\n";

    trace = false;
    const int n = 3000;
    const char* rules[] = {"last element", "median of three", "random"};
    const char* inputs[] = {"sorted", "random"};
    cout << "Pivot rules on " << n << " elements (Lomuto):\n";
    for (int in = 0; in < 2; ++in) {
        for (int rule = 0; rule < 3; ++rule) {
            vector<int> a(n);
            state = 2026;
            for (int k = 0; k < n; ++k)
                a[k] = (in == 0) ? k : nextRandom(1000000);
            comparisons = swaps = 0;
            maxDepth = 0;
            quickLomuto(a, 0, n - 1, rule, 0);
            cout << "  " << inputs[in] << " input, " << rules[rule] << ": "
                 << comparisons << " comparisons, depth " << maxDepth << "\n";
        }
    }
    return 0;
}
```

The Java version needs one extra precaution, and it is instructive. Sorting 3000 sorted elements with the last element as pivot recurses almost 3000 levels deep, and each level is a Java method call with its own frame on the call stack; on the Java runtime used here, that exceeded the default stack of the main thread and the program stopped with a `StackOverflowError`. **Listing 3.23** therefore runs the experiment on a new `Thread` created with a 64 MB stack.

Listing 3.23 `QuickSort.java`: quicksort in Java; the pivot experiment runs on a thread with a larger stack.

```java
// QuickSort.java - quicksort: Lomuto and Hoare partitions, three pivot rules
public class QuickSort {
    // the slice a[lo..hi], inclusive
    static String show(int[] a, int lo, int hi) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = lo; i <= hi; i++)
            sb.append(i > lo ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static long state = 2026;
    static int nextRandom(int bound) {
        state = state * 6364136223846793005L + 1442695040888963407L;
        return (int) ((state >>> 33) % bound);
    }

    static long comparisons = 0, swaps = 0;
    static int maxDepth = 0;
    static boolean trace = false;

    static void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

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

Listing 3.23 (Continued) *QuickSort.java: quicksort in Java; the pivot experiment runs on a thread with a larger stack.*

```
    // Hoare: pivot = middle value; two indices walk inward and swap.
    static int hoarePartition(int[] a, int lo, int hi) {
        int pivot = a[lo + (hi - lo) / 2];
        int i = lo - 1, j = hi + 1;
        while (true) {
            do { i++; comparisons++; } while (a[i] < pivot);
            do { j--; comparisons++; } while (a[j] > pivot);
            if (i >= j) return j;  // a[lo..j] <= pivot <= a[j+1..hi]
            swap(a, i, j);
            swaps++;
        }
    }

    static void quickLomuto(int[] a, int lo, int hi, int rule, int depth) {
        if (lo >= hi) return;
        maxDepth = Math.max(maxDepth, depth);
        if (rule == 1) {  // median of three: a[lo], a[mid], a[hi]
            int mid = lo + (hi - lo) / 2;
            if (a[mid] < a[lo]) swap(a, mid, lo);
            if (a[hi] < a[lo]) swap(a, hi, lo);
            if (a[mid] < a[hi]) swap(a, mid, hi);  // now a[hi] holds the median
        } else if (rule == 2) {                      // random pivot
            swap(a, lo + nextRandom(hi - lo + 1), hi);
        }
        String before = trace ? show(a, lo, hi) : "";
        int p = lomutoPartition(a, lo, hi);
        if (trace)
            System.out.println(" ".repeat(2 * depth + 2) + "partition "
                    + before + " pivot "
                    + a[p] + " -> " + show(a, lo, hi) + ", pivot at " + p);
        quickLomuto(a, lo, p - 1, rule, depth + 1);
        quickLomuto(a, p + 1, hi, rule, depth + 1);
    }

    static void quickHoare(int[] a, int lo, int hi, int depth) {
        if (lo >= hi) return;
        String before = show(a, lo, hi);
        int pivot = a[lo + (hi - lo) / 2];
        int p = hoarePartition(a, lo, hi);
        System.out.println(" ".repeat(2 * depth + 2) + "partition " + before
                + " pivot " + pivot
                + " -> " + show(a, lo, p) + " " + show(a, p + 1, hi));
        quickHoare(a, lo, p, depth + 1);
        quickHoare(a, p + 1, hi, depth + 1);
    }
```

Listing 3.23 (Continued) *QuickSort.java: quicksort in Java; the pivot experiment runs on a thread with a larger stack.*

```
    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println(
                "Quicksort, Lomuto partition, last element as pivot:");
        trace = true;
        quickLomuto(data, 0, 5, 0, 0);
        System.out.println("  result " + show(data, 0, 5) + ", comparisons = "
                + comparisons
                + ", swaps = " + swaps);

        int[] data2 = {22, 10, 15, 3, 8, 2};
        comparisons = swaps = 0;
        System.out.println(
                "Quicksort, Hoare partition, middle element as pivot:");
        quickHoare(data2, 0, 5, 0);
        System.out.println("  result " + show(data2, 0, 5)
                + ", comparisons = " + comparisons
                + ", swaps = " + swaps);

        trace = false;
        // A sorted input with the last-element rule recurses about 3000 levels
        // deep, more than the default thread stack holds, so this part runs on
        // a thread with a 64 MB stack.
        long stackBytes = 64L << 20;
        Thread big = new Thread(null, QuickSort::pivotExperiment, "pivots",
                stackBytes);
        big.start();
        try {
            big.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    static void pivotExperiment() {
        final int n = 3000;
        String[] rules = {"last element", "median of three", "random"};
        String[] inputs = {"sorted", "random"};
        System.out.println("Pivot rules on " + n + " elements (Lomuto):");
        for (int in = 0; in < 2; in++) {
            for (int rule = 0; rule < 3; rule++) {
                int[] a = new int[n];
                state = 2026;
                for (int k = 0; k < n; k++)
                    a[k] = (in == 0) ? k : nextRandom(1000000);
                comparisons = swaps = 0;
```

Listing 3.23 (Continued) *QuickSort.java: quicksort in Java; the pivot experiment runs on a thread with a larger stack.*

```
                maxDepth = 0;
                quickLomuto(a, 0, n - 1, rule, 0);
                System.out.println("  " + inputs[in] + " input, "
                        + rules[rule] + ": "
                        + comparisons + " comparisons, depth " + maxDepth);
            }
        }
    }
}
```

Python limits recursion depth to 1000 calls by default, so **Listing 3.24** raises the limit with `sys.setrecursionlimit` before the experiment. Python's `do … while` loops do not exist, so each `repeat … until` of **Pseudocode 3.9** becomes a first step followed by a `while`.

Listing 3.24 `quick_sort.py`: quicksort in Python.

```
# quick_sort.py - quicksort: Lomuto and Hoare partitions, three pivot rules
import sys


def show(a, lo, hi):                    # the slice a[lo..hi], inclusive
    return "[" + " ".join(map(str, a[lo:hi + 1])) + "]"


state = 2026


def next_random(bound):
    global state
    state = (state * 6364136223846793005 + 1442695040888963407) % 2**64
    return (state >> 33) % bound


comparisons = swaps = max_depth = 0
trace = False


def lomuto_partition(a, lo, hi):
    """Pivot = a[hi]; a[lo..i] holds the elements <= pivot seen so far."""
    global comparisons, swaps
    pivot = a[hi]
    i = lo - 1
    for j in range(lo, hi):
        comparisons += 1
        if a[j] <= pivot:
            i += 1
            if i != j:
                a[i], a[j] = a[j], a[i]
                swaps += 1
    if i + 1 != hi:
        a[i + 1], a[hi] = a[hi], a[i + 1]
        swaps += 1
    return i + 1                        # the pivot's final index
```

Listing 3.24 (Continued) *quick_sort.py: quicksort in Python.*

```
def hoare_partition(a, lo, hi):
    """Pivot = middle value; two indices walk inward, swapping wrong pairs."""
    global comparisons, swaps
    pivot = a[lo + (hi - lo) // 2]
    i, j = lo - 1, hi + 1
    while True:
        i += 1
        comparisons += 1
        while a[i] < pivot:
            i += 1
            comparisons += 1
        j -= 1
        comparisons += 1
        while a[j] > pivot:
            j -= 1
            comparisons += 1
        if i >= j:
            return j                    # a[lo..j] <= pivot <= a[j+1..hi]
        a[i], a[j] = a[j], a[i]
        swaps += 1


def quick_lomuto(a, lo, hi, rule, depth=0):
    global max_depth
    if lo >= hi:
        return
    max_depth = max(max_depth, depth)
    if rule == 1:                       # median of three: a[lo], a[mid], a[hi]
        mid = lo + (hi - lo) // 2
        if a[mid] < a[lo]:
            a[mid], a[lo] = a[lo], a[mid]
        if a[hi] < a[lo]:
            a[hi], a[lo] = a[lo], a[hi]
        if a[mid] < a[hi]:
            a[mid], a[hi] = a[hi], a[mid]    # now a[hi] holds the median
    elif rule == 2:                     # random pivot
        r = lo + next_random(hi - lo + 1)
        a[r], a[hi] = a[hi], a[r]
    before = show(a, lo, hi) if trace else ""
    p = lomuto_partition(a, lo, hi)
    if trace:
        print(" " * (2 * depth + 2) + f"partition {before} pivot {a[p]} -> "
              f"{show(a, lo, hi)}, pivot at {p}")
    quick_lomuto(a, lo, p - 1, rule, depth + 1)
    quick_lomuto(a, p + 1, hi, rule, depth + 1)
```

Listing 3.24 (Continued) *quick_sort.py: quicksort in Python.*

```
def quick_hoare(a, lo, hi, depth=0):
    if lo >= hi:
        return
    before = show(a, lo, hi)
    pivot = a[lo + (hi - lo) // 2]
    p = hoare_partition(a, lo, hi)
    print(" " * (2 * depth + 2) + f"partition {before} pivot {pivot} -> "
          f"{show(a, lo, p)} {show(a, p + 1, hi)}")
    quick_hoare(a, lo, p, depth + 1)
    quick_hoare(a, p + 1, hi, depth + 1)


data = [22, 10, 15, 3, 8, 2]
print("Quicksort, Lomuto partition, last element as pivot:")
trace = True
quick_lomuto(data, 0, 5, 0)
print(f"  result {show(data, 0, 5)}, comparisons = {comparisons}, "
      f"swaps = {swaps}")

data2 = [22, 10, 15, 3, 8, 2]
comparisons = swaps = 0
print("Quicksort, Hoare partition, middle element as pivot:")
quick_hoare(data2, 0, 5)
print(f"  result {show(data2, 0, 5)}, comparisons = {comparisons}, "
      f"swaps = {swaps}")

trace = False
sys.setrecursionlimit(10000)  # sorted input + last-element pivot: ~3000 levels
n = 3000
print(f"Pivot rules on {n} elements (Lomuto):")
for kind in ("sorted", "random"):
    for rule, name in enumerate(("last element", "median of three", "random")):
        state = 2026
        if kind == "sorted":
            a = list(range(n))
        else:
            a = [next_random(1000000) for _ in range(n)]
        comparisons = swaps = max_depth = 0
        quick_lomuto(a, 0, n - 1, rule)
        print(f"  {kind} input, {name}: {comparisons} comparisons, "
              f"depth {max_depth}")
```

The three programs print the same output:

```
Quicksort, Lomuto partition, last element as pivot:
  partition [22 10 15 3 8 2] pivot 2 -> [2 10 15 3 8 22], pivot at 0
    partition [10 15 3 8 22] pivot 22 -> [10 15 3 8 22], pivot at 5
      partition [10 15 3 8] pivot 8 -> [3 8 10 15], pivot at 2
        partition [10 15] pivot 15 -> [10 15], pivot at 4
  result [2 3 8 10 15 22], comparisons = 13, swaps = 3
Quicksort, Hoare partition, middle element as pivot:
  partition [22 10 15 3 8 2] pivot 15 -> [2 10 8 3] [15 22]
    partition [2 10 8 3] pivot 10 -> [2 3 8] [10]
      partition [2 3 8] pivot 3 -> [2 3] [8]
        partition [2 3] pivot 2 -> [2] [3]
    partition [15 22] pivot 15 -> [15] [22]
  result [2 3 8 10 15 22], comparisons = 24, swaps = 3
Pivot rules on 3000 elements (Lomuto):
  sorted input, last element: 4498500 comparisons, depth 2998
  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
```

3.9.5 Best, worst and average

Partitioning m elements costs $m-1$ comparisons. What happens next depends entirely on where the pivot lands.

The worst case. If the pivot is always the smallest or largest element, one side is empty and the other has $m-1$ elements. The recursion then has n levels, costing $(n-1)+(n-2)+\cdots+1$ comparisons:

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

With the last element as pivot this happens on *sorted* input, the most common input of all. The experiment confirms Equation (3.10) to the digit: $3000 \cdot 2999/2 = 4{,}498{,}500$ comparisons and a recursion 2998 levels deep, compared with about 38,000 comparisons on random input of the same size. The depth is as dangerous as the time. Every level of recursion occupies memory on the call stack, and a sorted array of a million elements would need a million nested calls, which crashes any of our three languages.

The best case. If every pivot lands in the middle, each call splits its range in half, and the recurrence is Equation (3.8) again, with solution $\Theta(n\lg n)$ by Equation (3.9).

The average case. With random input, or with a random pivot, the pivot lands at a random position. Some splits are bad and some are good, but a split of 1:9 is just as useful as 1:1 up to a constant factor, and bad splits are rare. A careful analysis [6], [1] gives an expected number of comparisons of

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

where $H_n = 1 + 1/2 + \cdots + 1/n \approx \ln n$ is the n-th harmonic number. That is only 39% more comparisons than the best case, and quicksort's inner loop is so simple (one comparison, one increment, occasionally a swap, all on neighboring memory) that it is usually the fastest comparison sort in practice.

Choosing the pivot. The experiment shows what the pivot rule buys.

- *Last (or first) element*: fine on random data, quadratic on sorted or reversed data. Never use it on data you do not control.
- *Median of three*: on sorted data the median of first, middle and last is the true middle, so sorted input becomes the *best* case: 28,917 comparisons and depth 10, close to $\lg 3000 \approx 11.6$. On random data it also helps a little, because the median of three samples is a better guess at the middle than one sample. It can still be defeated by inputs built on purpose to fool it.
- *Random element*: the running time no longer depends on the order of the input at all; every input has the same expected cost Equation (3.11). A quadratic run is still possible but astronomically unlikely.

Quicksort is **in-place** apart from its recursion stack, and it is **not stable**: partitioning exchanges elements across long distances. Two refinements make it safe for production. First, recurse only into the *smaller* side and loop on the larger one; then the stack never grows beyond $\lg n$ levels, and Section 3.13's benchmark program does exactly that. Second, if the recursion gets too deep, stop and switch to heap sort, which is never quadratic. That second idea is **introsort**, the algorithm inside `std::sort` (Section 3.14).

3.10 Heap Sort: a First Look

Selection sort spends almost all its time *finding* the smallest (or largest) remaining element: a full scan, round after round, that forgets everything it learned in the previous round. Heap sort keeps selection sort's plan, "repeatedly take the largest and put it at the end", but stores the unsorted elements in a shape that makes the largest one free to find and cheap to replace. That shape is the **binary heap**. This section shows just enough of it to sort; Chapter 14 builds the heap properly, proves its costs, and uses it as a priority queue.

3.10.1 A tree hidden inside an array

Read the array $A[0..n-1]$ as a binary tree: $A[0]$ is the root, and the children of the node at index i are at indices

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

No pointers are stored; the tree is pure arithmetic on indices. The tree fills level by level from the left, so its height is $\lfloor \lg n \rfloor$: six elements make a tree of height 2, a million elements a tree of height 19.

> **DEFINITION.** An array is a **max-heap** when every node is at least as large as its children: $A[\text{parent}(i)] \geq A[i]$ for every $i \geq 1$. Then the largest element is at the root, $A[0]$.

Two operations are all we need.

- **Sift down** repairs a node that may be smaller than a child, when both subtrees below it are already heaps: exchange the node with its larger child, and repeat from the child's position, until the value is at least as large as both children or reaches a leaf. It walks down one path of the tree, so it costs at most $2\lfloor \lg n \rfloor$ comparisons.
- **Build a heap** from an arbitrary array by sifting down every node that has children, from the last one, index $\lfloor n/2 \rfloor - 1$, back to the root. Chapter 14 proves that this costs only $O(n)$ in total, less than the $O(n \lg n)$ it seems to cost.

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
```

3.10.2 The trace

Our running array happens to be a max-heap already: $22 \geq 10$ and $22 \geq 15$ at the root, $10 \geq 3$ and $10 \geq 8$ below the left child, $15 \geq 2$ below the right one. So the build phase checks three nodes and moves nothing. **Figure 3.11** shows the heap and the five extraction steps that follow.

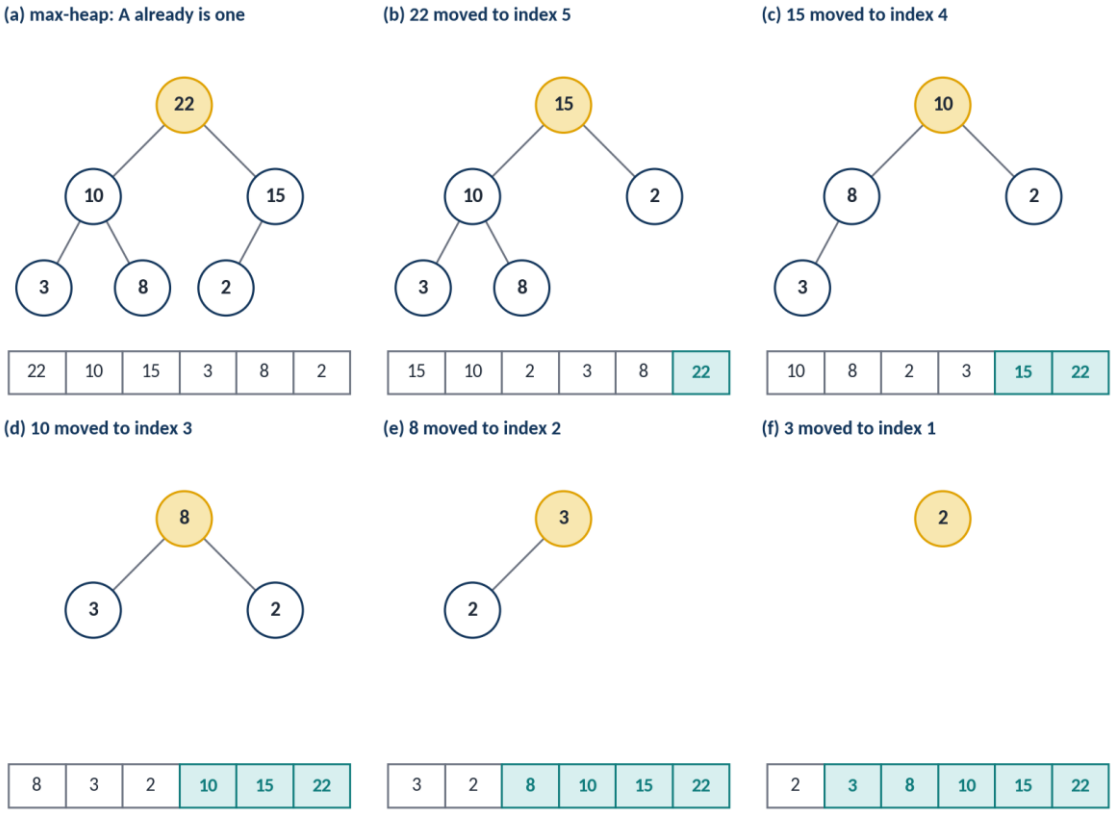


Figure 3.11 Heap sort on *A*. Each panel shows the heap as a tree (root in amber) and the whole array beneath it; the teal tail is the sorted part, which grows by one element per step.

Step (b) exchanges the root 22 with the last heap element 2, so 22 is in its final place at index 5 and the heap shrinks to five elements. The new root 2 is smaller than both children 10 and 15, so it sifts down, exchanging with the larger child 15; at index 2 it has no children left inside the heap, and the heap is [15,10,2,3,8]. Step (c) moves 15 to index 4; the new root 8 sifts below 10, giving [10,8,2,3]. Steps (d) to (f) continue until the heap has one element and the array is sorted.

Listing 3.25 `heap_sort.cpp`: heap sort on the running array, and on an array that has to be turned into a heap first.

```
// heap_sort.cpp - a first look at heap sort: build a max-heap, take the max
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

string show(const vector<int>& a, int lo, int hi) {  // a[lo..hi-1]
    string s = "[";
    for (int i = lo; i < hi; ++i) s += (i > lo ? " " : "") + to_string(a[i]);
    return s + "]";
}

long long comparisons = 0, swaps = 0;

// 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;
    }
}

void heapSort(vector<int>& a) {
    int n = static_cast<int>(a.size());
    // build the heap bottom-up
    for (int i = n / 2 - 1; i >= 0; --i) siftDown(a, i, n);
    cout << "  max-heap built: " << show(a, 0, n) << "\n";
    for (int end = n - 1; end > 0; --end) {
        swap(a[0], a[end]);  // the maximum goes to its final slot
        ++swaps;
        siftDown(a, 0, end);                        // repair the smaller heap
        cout << "  " << a[end] << " placed: heap " << show(a, 0, end)
             << " | sorted " << show(a, end, n) << "\n";
    }
```

Listing 3.25 (Continued) *heap_sort.cpp: heap sort on the running array, and on an array that has to be turned into a heap first.*

```
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Heap sort on " << show(data, 0, 6) << "\n";
    heapSort(data);
    cout << "  result " << show(data, 0, 6) << ", comparisons = " << comparisons
         << ", swaps = " << swaps << "\n";

    vector<int> other = {3, 8, 2, 22, 10, 15};
    comparisons = swaps = 0;
    cout << "Heap sort on " << show(other, 0, 6) << "\n";
    heapSort(other);
    cout << "  result " << show(other, 0, 6) << ", comparisons = "
         << comparisons << ", swaps = " << swaps << "\n";
    return 0;
}
```

Listing 3.26 `HeapSort.java`: heap sort in Java.

```java
// HeapSort.java - a first look at heap sort: build a max-heap, take the max
public class HeapSort {
    static String show(int[] a, int lo, int hi) {    // a[lo..hi-1]
        StringBuilder sb = new StringBuilder("[");
        for (int i = lo; i < hi; i++)
            sb.append(i > lo ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static long comparisons = 0, swaps = 0;

    static void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    // Lets a[i] sink until neither child is larger; the heap is a[0..size-1].
    static void siftDown(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, largest);
            swaps++;
            i = largest;
        }
    }

    static void heapSort(int[] a) {
        int n = a.length;
        // build the heap bottom-up
        for (int i = n / 2 - 1; i >= 0; i--) siftDown(a, i, n);
        System.out.println("  max-heap built: " + show(a, 0, n));
        for (int end = n - 1; end > 0; end--) {
            swap(a, 0, end);  // the maximum goes to its final slot
            swaps++;
            siftDown(a, 0, end);                    // repair the smaller heap
            System.out.println("  " + a[end] + " placed: heap "
                    + show(a, 0, end)
                    + " | sorted " + show(a, end, n));
```

Listing 3.26 (Continued) *HeapSort.java: heap sort in Java.*

```
        }
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Heap sort on " + show(data, 0, 6));
        heapSort(data);
        System.out.println("  result " + show(data, 0, 6) + ", comparisons = "
                + comparisons
                + ", swaps = " + swaps);

        int[] other = {3, 8, 2, 22, 10, 15};
        comparisons = swaps = 0;
        System.out.println("Heap sort on " + show(other, 0, 6));
        heapSort(other);
        System.out.println("  result " + show(other, 0, 6)
                + ", comparisons = " + comparisons
                + ", swaps = " + swaps);
    }
}
```

Listing 3.27 `heap_sort.py`: heap sort in Python.

```
# heap_sort.py - a first look at heap sort: build a max-heap, take the max
def show(a, lo, hi):                    # a[lo..hi-1]
    return "[" + " ".join(map(str, a[lo:hi])) + "]"


comparisons = swaps = 0


def sift_down(a, i, size):
    """Lets a[i] sink until no child is larger; the heap is a[0..size-1]."""
    global comparisons, swaps
    while True:
        largest, left, right = i, 2 * i + 1, 2 * i + 2
        if left < size:
            comparisons += 1
            if a[left] > a[largest]:
                largest = left
        if right < size:
            comparisons += 1
            if a[right] > a[largest]:
                largest = right
        if largest == i:
            return
        a[i], a[largest] = a[largest], a[i]
        swaps += 1
        i = largest


def heap_sort(a):
    global swaps
    n = len(a)
    for i in range(n // 2 - 1, -1, -1):  # build the heap bottom-up
        sift_down(a, i, n)
    print(f"  max-heap built: {show(a, 0, n)}")
    for end in range(n - 1, 0, -1):
        a[0], a[end] = a[end], a[0]      # the maximum goes to its final slot
        swaps += 1
        sift_down(a, 0, end)             # repair the smaller heap
        print(f"  {a[end]} placed: heap {show(a, 0, end)} | "
              f"sorted {show(a, end, n)}")
```

Listing 3.27 (Continued) *heap_sort.py: heap sort in Python.*

```
for data in ([22, 10, 15, 3, 8, 2], [3, 8, 2, 22, 10, 15]):
    comparisons = swaps = 0
    print(f"Heap sort on {show(data, 0, 6)}")
    heap_sort(data)
    print(f"  result {show(data, 0, 6)}, comparisons = {comparisons}, "
          f"swaps = {swaps}")
```

Listing 3.25, **Listing 3.26** and **Listing 3.27** print:

```
Heap sort on [22 10 15 3 8 2]
  max-heap built: [22 10 15 3 8 2]
  22 placed: heap [15 10 2 3 8] | sorted [22]
  15 placed: heap [10 8 2 3] | sorted [15 22]
  10 placed: heap [8 3 2] | sorted [10 15 22]
  8 placed: heap [3 2] | sorted [8 10 15 22]
  3 placed: heap [2] | sorted [3 8 10 15 22]
  result [2 3 8 10 15 22], comparisons = 13, swaps = 9
Heap sort on [3 8 2 22 10 15]
  max-heap built: [22 10 15 8 3 2]
  22 placed: heap [15 10 2 8 3] | sorted [22]
  15 placed: heap [10 8 2 3] | sorted [15 22]
  10 placed: heap [8 3 2] | sorted [10 15 22]
  8 placed: heap [3 2] | sorted [8 10 15 22]
  3 placed: heap [2] | sorted [3 8 10 15 22]
  result [2 3 8 10 15 22], comparisons = 15, swaps = 14
```

The second run starts from [3,8,2,22,10,15], which is not a heap; building the heap moves 22 to the root and produces [22,10,15,8,3,2], and from there the extraction proceeds as before.

Each of the $n-1$ extractions costs one exchange and one sift-down of at most $2\lg n$ comparisons, and the build costs $O(n)$, so heap sort makes $O(n\lg n)$ comparisons **in the worst case**, with no bad inputs at all, and it is **in-place**: the heap and the sorted tail share the one array. This is the combination that no other algorithm in the chapter has: merge sort is $O(n\lg n)$ but needs a buffer, and quicksort is in-place but can be quadratic. Heap sort's price is that its sift-downs jump around the array (i, $2i+1$, $4i+3$, ...), which is unkind to the processor's cache, so it is usually slower than quicksort in practice, as Section 3.13 measures. It is **not stable**. Heap sort was published by Williams in 1964 [7], with the linear-time build by Floyd in the same year [8].

3.11 Counting Sort and Radix Sort

3.11.1 The barrier that comparison sorts cannot cross

Every algorithm so far learns about the data only by asking "is x before y?". There is a limit to how fast any such algorithm can be. Think of the sort as a game of twenty questions. There are $n!$ possible orders of n distinct keys, and the sort must tell them all apart, because each needs a different rearrangement. Each comparison has two possible answers, so k comparisons can distinguish at most 2^k cases. To distinguish $n!$ cases we need $2^k \geq n!$, that is,

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

(The middle step keeps only the largest $n/2$ factors of $n!$, each at least $n/2$.) So **every comparison sort needs $\Omega(n\lg n)$ comparisons in the worst case**; merge sort and heap sort are optimal up to a constant factor [6]. For six elements, $\lg(6!) = \lg 720 \approx 9.49$, so some input needs at least 10 comparisons whatever the algorithm; our merge sort needed 9 on A, which is allowed, because the bound speaks of the *worst* input.

The way past the barrier is to stop comparing. If the keys are small integers, or strings of digits, we can use their *values* directly as array indices.

3.11.2 Counting sort

Suppose every key is an integer between 0 and $k-1$. **Counting sort**, **Pseudocode 3.11**, works in three simple loops.

1. **Count**: `count[v]` = how many elements have key v.
2. **Running total**: replace each count by the sum of itself and all counts before it. Now `count[v]` is the number of elements with key $\leq v$, which is exactly one past the last position where a key v belongs in the output.
3. **Place**: walk the input *from right to left*; for each element with key v, decrease `count[v]` and put the element at that position of the output array.

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
```

Figure 3.12 applies it to A with $k = 23$ (keys 0 to 22).

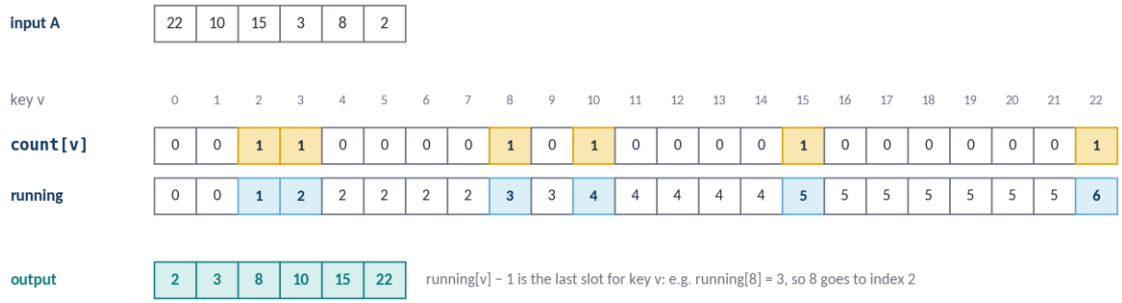


Figure 3.12 Counting sort on A with keys 0 to 22. The counts (amber) become running totals (blue), and each running total says where its key ends in the output.

The running total at 8 is 3: three keys (2, 3 and 8) are ≤ 8, so 8 belongs at index $3 - 1 = 2$. Line 6 walks from right to left so that, when several elements share a key, the last one takes the last slot, the one before it the slot before, and so on; equal keys therefore keep their original order, and counting sort is **stable**. That stability is not a luxury: radix sort depends on it.

Counting sort runs in time proportional to $n + k$: one loop over the n elements twice and one over the k counters.

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

When k is no larger than a constant times n, that is linear, which beats $n\lg n$. When k is huge (keys up to a billion) the counter array is too big to be practical, and that is where radix sort comes in.

3.11.3 Radix sort

Radix sort sorts numbers digit by digit, starting with the **least significant** digit (LSD), using a stable counting sort with $k = 10$ for each digit. It sounds backwards, since we compare numbers by their leading digit, and that is exactly why the stability matters: when the tens digits are sorted in the second pass, numbers with the same tens digit keep the order the first pass gave them, which is the order of their ones digits. **Figure 3.13** sorts A in two passes.

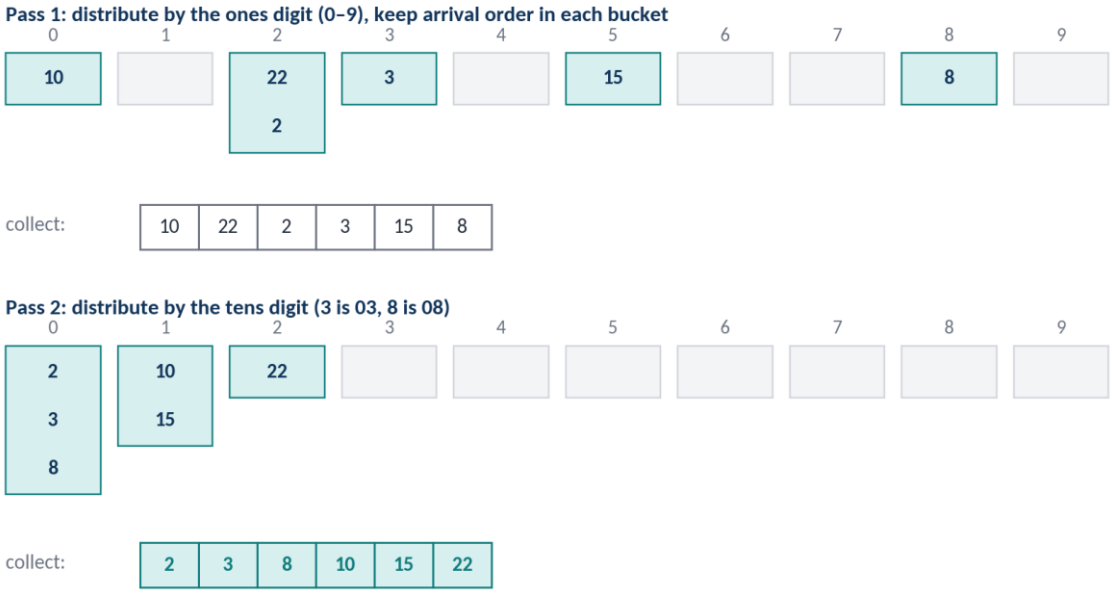


Figure 3.13 LSD radix sort on A. The first pass distributes by the ones digit; the second, by the tens digit, keeps each bucket in first-pass order, and the array comes out sorted.

After the ones-digit pass the order is [10,22,2,3,15,8]: 22 comes before 2 because it came first in the input and both have ones digit 2. In the tens-digit pass, 2, 3 and 8 all have tens digit 0 and, because the pass is stable, stay in the order 2,3,8 that the first pass produced; 10 and 15 share tens digit 1 and stay in order 10,15. The result is sorted. For d-digit keys in base b radix sort makes d passes of counting sort:

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

For 32-bit integers the benchmark in Section 3.13 uses base $b = 256$, one byte per pass, so $d = 4$ and each pass uses 256 counters. **Listing 3.28** implements both algorithms; the digit to sort by is passed to counting sort as a small function (a *lambda*), so one counting sort serves both purposes. The last example sorts seven three-digit numbers in three passes.

Listing 3.28 `counting_radix.cpp`: a stable counting sort parameterized by a key function, and LSD radix sort built on it.

```cpp
// counting_radix.cpp - counting sort and LSD radix sort, with no comparisons
#include <iostream>
#include <string>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

// 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;
}

// base 10, least significant digit first
vector<int> radixSort(vector<int> a, bool trace) {
    int maxValue = 0;
    for (int x : a) if (x > maxValue) maxValue = x;
    for (int place = 1; maxValue / place > 0; place *= 10) {
        auto digit = [place](int x) { return x / place % 10; };
        a = countingSort(a, 10, digit, false);
        if (trace)
            cout << "  by digit of " << place << "s: " << show(a) << "\n";
    }
    return a;
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    cout << "Counting sort on " << show(data) << " (keys 0..22)\n";
    vector<int> sorted = countingSort(data, 23, [](int x) { return x; }, true);
    cout << "  result " << show(sorted) << "\n";
```

Listing 3.28 (Continued) *counting_radix.cpp: a stable counting sort parameterized by a key function, and LSD radix sort built on it.*

```cpp
    cout << "Radix sort (base 10) on " << show(data) << "\n";
    sorted = radixSort(data, true);
    cout << "  result " << show(sorted) << "\n";

    vector<int> big = {329, 457, 657, 839, 436, 720, 355};
    cout << "Radix sort (base 10) on " << show(big) << "\n";
    sorted = radixSort(big, true);
    cout << "  result " << show(sorted) << "\n";
    return 0;
}
```

Listing 3.29 `CountingRadix.java`: counting and radix sort in Java, with `IntUnaryOperator` as the key function.

```java
// CountingRadix.java - counting sort and LSD radix sort, with no comparisons
import java.util.function.IntUnaryOperator;

public class CountingRadix {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    // Stable counting sort of a by key(x) in 0..k-1; returns the sorted copy.
    static int[] countingSort(int[] a, int k, IntUnaryOperator key,
                              boolean trace) {
        int[] count = new int[k];
        for (int x : a) count[key.applyAsInt(x)]++;  // 1. how many of each key
        if (trace) System.out.println("  counts:        " + show(count));
        // 2. running totals
        for (int v = 1; v < k; v++) count[v] += count[v - 1];
        if (trace) System.out.println("  running total: " + show(count));
        int[] out = new int[a.length];
        for (int i = a.length - 1; i >= 0; i--)  // 3. right to left: stable
            out[--count[key.applyAsInt(a[i])]] = a[i];
        return out;
    }

    // base 10, lowest digit first
    static int[] radixSort(int[] a, boolean trace) {
        int maxValue = 0;
        for (int x : a) if (x > maxValue) maxValue = x;
        for (int place = 1; maxValue / place > 0; place *= 10) {
            final int p = place;
            a = countingSort(a, 10, x -> x / p % 10, false);
            if (trace) System.out.println("  by digit of " + place + "s: "
                    + show(a));
        }
        return a;
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        System.out.println("Counting sort on " + show(data) + " (keys 0..22)");
        int[] sorted = countingSort(data, 23, x -> x, true);
        System.out.println("  result " + show(sorted));
```

Listing 3.29 (Continued) *CountingRadix.java: counting and radix sort in Java, with IntUnaryOperator as the key function.*

```java
        System.out.println("Radix sort (base 10) on " + show(data));
        sorted = radixSort(data, true);
        System.out.println("  result " + show(sorted));

        int[] big = {329, 457, 657, 839, 436, 720, 355};
        System.out.println("Radix sort (base 10) on " + show(big));
        sorted = radixSort(big, true);
        System.out.println("  result " + show(sorted));
    }
}
```

Listing 3.30 `counting_radix.py`: counting and radix sort in Python, with a `lambda` as the key function.

```
# counting_radix.py - counting sort and LSD radix sort, with no comparisons
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def counting_sort(a, k, key, trace=False):
    """Stable counting sort of a by key(x) in 0..k-1; returns a sorted copy."""
    count = [0] * k
    for x in a:                          # 1. how many of each key
        count[key(x)] += 1
    if trace:
        print(f"  counts:         {show(count)}")
    for v in range(1, k):                # 2. running totals
        count[v] += count[v - 1]
    if trace:
        print(f"  running total: {show(count)}")
    out = [0] * len(a)
    for x in reversed(a):                # 3. right to left: stable
        count[key(x)] -= 1
        out[count[key(x)]] = x
    return out


def radix_sort(a, trace=False):  # base 10, least significant digit first
    max_value = max(a)
    place = 1
    while max_value // place > 0:
        a = counting_sort(a, 10, lambda x: x // place % 10)
        if trace:
            print(f"  by digit of {place}s: {show(a)}")
        place *= 10
    return a


data = [22, 10, 15, 3, 8, 2]
print(f"Counting sort on {show(data)} (keys 0..22)")
print(f"  result {show(counting_sort(data, 23, lambda x: x, True))}")

print(f"Radix sort (base 10) on {show(data)}")
result = radix_sort(data, True)
print(f"  result {show(result)}")

big = [329, 457, 657, 839, 436, 720, 355]
print(f"Radix sort (base 10) on {show(big)}")
result = radix_sort(big, True)
print(f"  result {show(result)}")
```

Listing 3.29 and **Listing 3.30** give the same output as **Listing 3.28**:

```
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]
Radix sort (base 10) on [22 10 15 3 8 2]
  by digit of 1s: [10 22 2 3 15 8]
  by digit of 10s: [2 3 8 10 15 22]
  result [2 3 8 10 15 22]
Radix sort (base 10) on [329 457 657 839 436 720 355]
  by digit of 1s: [720 355 436 457 657 329 839]
  by digit of 10s: [720 329 436 839 355 457 657]
  by digit of 100s: [329 355 436 457 657 720 839]
  result [329 355 436 457 657 720 839]
```

Both sorts need an output array of n elements and so are **not in-place**; both are **stable**. They are not general: they work on keys that are integers, or that can be cut into digits (fixed-length

strings, dates), not on arbitrary objects with a comparison function. Within that range they are the fastest sorts in this chapter, as the benchmark will show.

3.12 Stability, In-Place, and the Complete Table

3.12.1 Stability you can see

Stability is easiest to understand when it fails. **Listing 3.31** takes Pustaka Ceria's eight books in order of their ids, `B1001` to `B1008`, and sorts them by category twice: once with insertion sort, which is stable, and once with selection sort, which is not. The comparison looks only at the category, so the four books of category *Fiksi* have equal keys.

Listing 3.31 `stability_demo.cpp`: the catalog sorted by category with a stable and an unstable sort. The comparison is passed as a function pointer.

```
// stability_demo.cpp - stable and unstable sorts of the catalog by category
#include <cstdio>
#include <string>
#include <utility>
#include <vector>
using namespace std;

struct Book {
    string id, title, category;
};

bool categoryLess(const Book& a, const Book& b) {
    return a.category < b.category;
}

void insertionSort(vector<Book>& a, bool (*less)(const Book&, const Book&)) {
    for (size_t i = 1; i < a.size(); ++i) {
        Book key = a[i];
        size_t j = i;
        // strictly less: equal keys never pass
        while (j > 0 && less(key, a[j - 1])) {
            a[j] = a[j - 1];
            --j;
        }
        a[j] = key;
    }
}

void selectionSort(vector<Book>& a, bool (*less)(const Book&, const Book&)) {
    for (size_t i = 0; i + 1 < a.size(); ++i) {
        size_t pos = i;
        for (size_t j = i + 1; j < a.size(); ++j)
            if (less(a[j], a[pos])) pos = j;
        if (pos != i) swap(a[i], a[pos]);  // a long jump can pass equal keys
    }
}

void print(const char* heading, const vector<Book>& a) {
    printf("%s\n", heading);
    for (const Book& b : a)
        printf("  %-10s %s  %s\n", b.category.c_str(), b.id.c_str(),
               b.title.c_str());
}
```

Listing 3.31 (Continued) *stability_demo.cpp: the catalog sorted by category with a stable and an unstable sort. The comparison is passed as a function pointer.*

```
int main() {
    vector<Book> catalog = {
        {"B1001", "Laskar Pelangi", "Fiksi"},
        {"B1002", "Bumi Manusia", "Fiksi"},
        {"B1003", "Filosofi Kopi", "Fiksi"},
        {"B1004", "Sejarah Nusantara", "Sejarah"},
        {"B1005", "Algoritma & Struktur Data", "Teknologi"},
        {"B1006", "Cerita Rakyat Nusantara", "Anak-anak"},
        {"B1007", "Fisika Dasar", "Sains"},
        {"B1008", "Negeri 5 Menara", "Fiksi"},
    };
    // both start in id order
    vector<Book> stable = catalog, unstable = catalog;
    insertionSort(stable, categoryLess);
    selectionSort(unstable, categoryLess);
    print("By category with insertion sort (stable):", stable);
    print("By category with selection sort (not stable):", unstable);
    return 0;
}
```

Listing 3.32 `StabilityDemo.java`: the same demonstration in Java, with a `record` for the book and a `Comparator` for the key.

```
// StabilityDemo.java - stable and unstable sorts of the catalog by category
import java.util.Comparator;

public class StabilityDemo {
    record Book(String id, String title, String category) { }

    static void insertionSort(Book[] a, Comparator<Book> cmp) {
        for (int i = 1; i < a.length; i++) {
            Book key = a[i];
            int j = i;
            // strictly less: an equal key never passes the one before it
            while (j > 0 && cmp.compare(key, a[j - 1]) < 0) {
                a[j] = a[j - 1];
                j--;
            }
            a[j] = key;
        }
    }

    static void selectionSort(Book[] a, Comparator<Book> cmp) {
        for (int i = 0; i + 1 < a.length; i++) {
            int pos = i;
            for (int j = i + 1; j < a.length; j++)
                if (cmp.compare(a[j], a[pos]) < 0) pos = j;
            if (pos != i) {                  // a long jump can pass equal keys
                Book t = a[i]; a[i] = a[pos]; a[pos] = t;
            }
        }
    }

    static void print(String heading, Book[] a) {
        System.out.println(heading);
        for (Book b : a)
            System.out.printf("  %-10s %s  %s%n",
                    b.category(), b.id(), b.title());
    }
```

Listing 3.32 (Continued) *StabilityDemo.java: the same demonstration in Java, with a record for the book and a Comparator for the key.*

```
    public static void main(String[] args) {
        Book[] catalog = {
            new Book("B1001", "Laskar Pelangi", "Fiksi"),
            new Book("B1002", "Bumi Manusia", "Fiksi"),
            new Book("B1003", "Filosofi Kopi", "Fiksi"),
            new Book("B1004", "Sejarah Nusantara", "Sejarah"),
            new Book("B1005", "Algoritma & Struktur Data", "Teknologi"),
            new Book("B1006", "Cerita Rakyat Nusantara", "Anak-anak"),
            new Book("B1007", "Fisika Dasar", "Sains"),
            new Book("B1008", "Negeri 5 Menara", "Fiksi"),
        };
        Comparator<Book> byCategory = Comparator.comparing(Book::category);
        // both start in id order
        Book[] stable = catalog.clone(), unstable = catalog.clone();
        insertionSort(stable, byCategory);
        selectionSort(unstable, byCategory);
        print("By category with insertion sort (stable):", stable);
        print("By category with selection sort (not stable):", unstable);
    }
}
```

Listing 3.33 `stability_demo.py`: the same demonstration in Python, with a `dataclass` and a key function.

```
# stability_demo.py - stable and unstable sorts of the catalog by category
from dataclasses import dataclass


@dataclass
class Book:
    id: str
    title: str
    category: str


def insertion_sort(a, key):
    for i in range(1, len(a)):
        item = a[i]
        j = i
        # strictly less: equal keys never pass
        while j > 0 and key(item) < key(a[j - 1]):
            a[j] = a[j - 1]
            j -= 1
        a[j] = item


def selection_sort(a, key):
    for i in range(len(a) - 1):
        pos = i
        for j in range(i + 1, len(a)):
            if key(a[j]) < key(a[pos]):
                pos = j
        if pos != i:
            a[i], a[pos] = a[pos], a[i]  # a long jump can pass equal keys


def print_books(heading, a):
    print(heading)
    for b in a:
        print(f"  {b.category:<10} {b.id}  {b.title}")
```

Listing 3.33 (Continued) *stability_demo.py: the same demonstration in Python, with a dataclass and a key function.*

```
catalog = [
    Book("B1001", "Laskar Pelangi", "Fiksi"),
    Book("B1002", "Bumi Manusia", "Fiksi"),
    Book("B1003", "Filosofi Kopi", "Fiksi"),
    Book("B1004", "Sejarah Nusantara", "Sejarah"),
    Book("B1005", "Algoritma & Struktur Data", "Teknologi"),
    Book("B1006", "Cerita Rakyat Nusantara", "Anak-anak"),
    Book("B1007", "Fisika Dasar", "Sains"),
    Book("B1008", "Negeri 5 Menara", "Fiksi"),
]
stable, unstable = catalog[:], catalog[:]    # both start in id order
insertion_sort(stable, key=lambda b: b.category)
selection_sort(unstable, key=lambda b: b.category)
print_books("By category with insertion sort (stable):", stable)
print_books("By category with selection sort (not stable):", unstable)
```

Listing 3.32 and **Listing 3.33** print the same as **Listing 3.31**:

```
By category with insertion sort (stable):
  Anak-anak  B1006  Cerita Rakyat Nusantara
  Fiksi      B1001  Laskar Pelangi
  Fiksi      B1002  Bumi Manusia
  Fiksi      B1003  Filosofi Kopi
  Fiksi      B1008  Negeri 5 Menara
  Sains      B1007  Fisika Dasar
  Sejarah    B1004  Sejarah Nusantara
  Teknologi  B1005  Algoritma & Struktur Data
By category with selection sort (not stable):
  Anak-anak  B1006  Cerita Rakyat Nusantara
  Fiksi      B1002  Bumi Manusia
  Fiksi      B1003  Filosofi Kopi
  Fiksi      B1001  Laskar Pelangi
  Fiksi      B1008  Negeri 5 Menara
  Sains      B1007  Fisika Dasar
  Sejarah    B1004  Sejarah Nusantara
  Teknologi  B1005  Algoritma & Struktur Data
```

Figure 3.14 puts the three orders side by side. Insertion sort keeps the four *Fiksi* books in id order. Selection sort does not: in its first round it exchanges `B1001` (position 0) with `B1006` (the only *Anak-anak* book, the smallest category), which throws `B1001` to position 5, behind `B1002` and `B1003`. Every book is still in the right category group; only the order *inside* the group has changed. If the librarian had first arranged the shelf by id and expected that order to survive, the unstable sort broke that expectation.

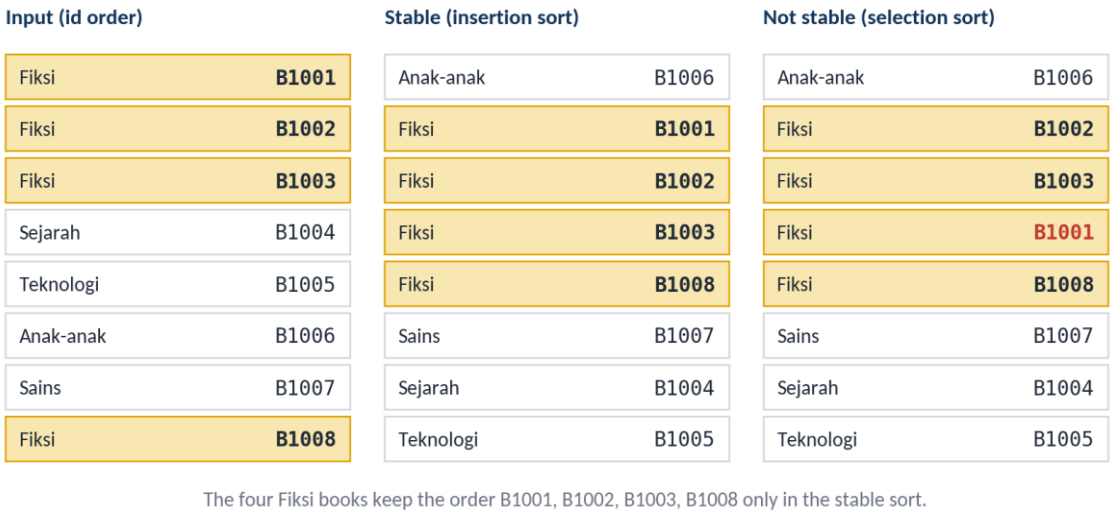


Figure 3.14 Sorting the catalog by category. Only the stable sort keeps the four *Fiksi* books in their original id order; selection sort moves B1001 behind B1002 and B1003.

The practical rule follows: **to sort by several keys, sort by the least important key first and by the most important key last, with a stable sort each time.** To list books by category and, inside each category, by title, sort by title and then stably by category. Radix sort is this rule applied to digits. Any unstable sort can be made stable by adding the original position as a last tie-breaking key, at the cost of storing that position.

3.12.2 The complete table

Table 3.2 collects everything the chapter has derived. "Adaptive" means that the cost falls on nearly sorted input.

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

Three entries deserve a word. Quicksort's extra space is its recursion stack, $\lg n$ levels when it always recurses into the smaller side. Shell sort's average case has no known formula for any practical gap sequence; the figure given is the usual empirical estimate [1], [9]. And heap sort's best case is $n\lg n$ only when the keys are distinct; with all keys equal every sift-down stops at once and it takes linear time.

Table 3.3 lists what each algorithm actually did to our six numbers. The same input, the same answer, and 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

Hoare's count of 24 looks high only because our program counts every test of the two scanning loops, including the ones that stop them; on large inputs Hoare's scheme makes about as many comparisons as Lomuto's and far fewer exchanges. On six elements such differences are noise. To see which algorithms really are fast, we must go large.

3.13 Measuring the Sorts

3.13.1 The benchmark program

Formulas predict how the work *grows*; only a measurement says how long it actually *takes*. **Listing 3.34** runs every algorithm of the chapter on the same random numbers for $n = 10^3, 10^4, 10^5$ and 10^6, and reports comparisons, swaps or moves, and time. The three quadratic sorts stop at 10^5 elements: at 10^6, bubble sort would need about fifty minutes. The last two rows of each size are the C++ library's own sorts, whose comparisons are counted by a comparator that increments a counter; the library does not tell us how many elements it moves, so that column shows a dash. The program checks that every result really is sorted, and it finishes by timing the three quadratic sorts at $n = 10^4$ on random, ascending and descending input.

The quicksort in this program is the production form of Section 3.9: median-of-three pivot, and a loop in place of the second recursive call so that it recurses only into the smaller side and its stack stays below lg$n$ levels. The Shell sort uses Ciura's gaps.

Listing 3.34 `sort_benchmark.cpp`: every sort of the chapter on a thousand to a million random integers, with counts and times.

```cpp
// sort_benchmark.cpp - counts and times for every sort in this chapter
#include <algorithm>
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <string>
#include <utility>
#include <vector>
using namespace std;

struct Counts { long long cmp = 0, mov = 0; };  // comparisons, swaps or moves

uint64_t state = 2026;
int nextRandom(int bound) {
    state = state * 6364136223846793005ULL + 1442695040888963407ULL;
    return static_cast<int>((state >> 33) % static_cast<uint64_t>(bound));
}

void bubble(vector<int>& a, Counts& c) {
    int n = static_cast<int>(a.size());
    for (int pass = 1; pass < n; ++pass) {
        bool swapped = false;
        for (int j = 0; j < n - pass; ++j) {
            ++c.cmp;
            if (a[j] > a[j + 1]) {
                swap(a[j], a[j + 1]); ++c.mov;
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}

void selection(vector<int>& a, Counts& c) {
    int n = static_cast<int>(a.size());
    for (int i = 0; i < n - 1; ++i) {
        int pos = i;
        for (int j = i + 1; j < n; ++j) { ++c.cmp; if (a[j] < a[pos]) pos = j; }
        if (pos != i) { swap(a[i], a[pos]); ++c.mov; }
    }
}
```

Listing 3.34 (Continued) *sort_benchmark.cpp: every sort of the chapter on a thousand to a million random integers, with counts and times.*

```
void gappedInsertion(vector<int>& a, int gap, Counts& c) {
    int n = static_cast<int>(a.size());
    for (int i = gap; i < n; ++i) {
        int key = a[i], j = i - gap;
        while (j >= 0) {
            ++c.cmp;
            if (a[j] <= key) break;
            a[j + gap] = a[j]; ++c.mov;
            j -= gap;
        }
        a[j + gap] = key;
    }
}

void insertion(vector<int>& a, Counts& c) { gappedInsertion(a, 1, c); }

void shell(vector<int>& a, Counts& c) {  // Ciura's gaps, extended by x2.25
    vector<int> gaps = {1, 4, 10, 23, 57, 132, 301, 701, 1750};
    while (gaps.back() * 9 / 4 < static_cast<int>(a.size()))
        gaps.push_back(gaps.back() * 9 / 4);
    for (int k = static_cast<int>(gaps.size()) - 1; k >= 0; --k)
        if (gaps[k] < static_cast<int>(a.size()))
            gappedInsertion(a, gaps[k], c);
}

void mergeRec(vector<int>& a, vector<int>& t, int lo, int hi, Counts& c) {
    if (hi - lo < 2) return;
    int mid = lo + (hi - lo) / 2;
    mergeRec(a, t, lo, mid, c);
    mergeRec(a, t, mid, hi, c);
    int i = lo, j = mid, k = lo;
    while (i < mid && j < hi) {
        ++c.cmp;
        t[k++] = (a[i] <= a[j]) ? a[i++] : a[j++];
    }
    while (i < mid) t[k++] = a[i++];
    while (j < hi) t[k++] = a[j++];
    for (k = lo; k < hi; ++k) { a[k] = t[k]; ++c.mov; }
}

void mergeSort(vector<int>& a, Counts& c) {
    vector<int> t(a.size());
    mergeRec(a, t, 0, static_cast<int>(a.size()), c);
}
```

Listing 3.34 (Continued) *sort_benchmark.cpp: every sort of the chapter on a thousand to a million random integers, with counts and times.*

```
void quickRec(vector<int>& a, int lo, int hi, Counts& c) {
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;                // median of three to a[hi]
        if (a[mid] < a[lo]) swap(a[mid], a[lo]);
        if (a[hi] < a[lo]) swap(a[hi], a[lo]);
        if (a[mid] < a[hi]) swap(a[mid], a[hi]);
        c.cmp += 3;
        int pivot = a[hi], i = lo - 1;
        for (int j = lo; j < hi; ++j) {
            ++c.cmp;
            if (a[j] <= pivot) {
                ++i;
                if (i != j) { swap(a[i], a[j]); ++c.mov; }
            }
        }
        swap(a[i + 1], a[hi]); ++c.mov;
        int p = i + 1;
        // recurse into the smaller side, loop on the larger one
        if (p - lo < hi - p) { quickRec(a, lo, p - 1, c); lo = p + 1; }
        else                 { quickRec(a, p + 1, hi, c); hi = p - 1; }
    }
}

void quick(vector<int>& a, Counts& c) {
    quickRec(a, 0, static_cast<int>(a.size()) - 1, c);
}

void siftDown(vector<int>& a, int i, int size, Counts& c) {
    while (true) {
        int largest = i, l = 2 * i + 1, r = l + 1;
        if (l < size) { ++c.cmp; if (a[l] > a[largest]) largest = l; }
        if (r < size) { ++c.cmp; if (a[r] > a[largest]) largest = r; }
        if (largest == i) return;
        swap(a[i], a[largest]); ++c.mov;
        i = largest;
    }
}
```

Listing 3.34 (Continued) *sort_benchmark.cpp: every sort of the chapter on a thousand to a million random integers, with counts and times.*

```
void heap(vector<int>& a, Counts& c) {
    int n = static_cast<int>(a.size());
    for (int i = n / 2 - 1; i >= 0; --i) siftDown(a, i, n, c);
    for (int end = n - 1; end > 0; --end) {
        swap(a[0], a[end]); ++c.mov;
        siftDown(a, 0, end, c);
    }
}

// base 256, four passes, no comparisons
void radix(vector<int>& a, Counts& c) {
    vector<int> out(a.size());
    for (int shift = 0; shift < 32; shift += 8) {
        vector<size_t> count(257, 0);
        for (int x : a) ++count[((x >> shift) & 255) + 1];
        for (int v = 0; v < 256; ++v) count[v + 1] += count[v];
        for (int x : a) { out[count[(x >> shift) & 255]++] = x; ++c.mov; }
        a.swap(out);
    }
}

void stdSort(vector<int>& a, Counts& c) {  // moves are not visible: -1
    c.mov = -1;
    sort(a.begin(), a.end(), [&c](int x, int y) { ++c.cmp; return x < y; });
}

void stdStable(vector<int>& a, Counts& c) {
    c.mov = -1;
    stable_sort(a.begin(), a.end(),
                [&c](int x, int y) { ++c.cmp; return x < y; });
}

string cell(long long v) { return v < 0 ? "-" : to_string(v); }

double run(void (*sortFn)(vector<int>&, Counts&), vector<int> a, Counts& c) {
    auto t0 = chrono::steady_clock::now();
    sortFn(a, c);
    auto t1 = chrono::steady_clock::now();
    if (!is_sorted(a.begin(), a.end())) printf("  NOT SORTED!\n");
    return chrono::duration<double, milli>(t1 - t0).count();
}
```

Listing 3.34 (Continued) *sort_benchmark.cpp: every sort of the chapter on a thousand to a million random integers, with counts and times.*

```
int main() {
    struct Algo {
        const char* name;
        void (*fn)(vector<int>&, Counts&);
        int maxN;
    };
    Algo algos[] = {
        {"bubble", bubble, 100000},     {"selection", selection, 100000},
        {"insertion", insertion, 100000}, {"shell", shell, 1000000},
        {"merge", mergeSort, 1000000},  {"quick", quick, 1000000},
        {"heap", heap, 1000000},        {"radix", radix, 1000000},
        {"std::sort", stdSort, 1000000},
        {"std::stable_sort", stdStable, 1000000},
    };
    printf("%-17s %8s %15s %15s %11s\n", "algorithm", "n", "comparisons",
           "swaps/moves", "time ms");
    for (int n = 1000; n <= 1000000; n *= 10) {
        state = 2026;
        vector<int> data(n);
        for (int& x : data) x = nextRandom(1000000000);
        for (const Algo& al : algos) {
            if (n > al.maxN) continue;
            Counts c;
            double ms = run(al.fn, data, c);
            printf("%-17s %8d %15s %15s %11.2f\n", al.name, n,
                   cell(c.cmp).c_str(), cell(c.mov).c_str(), ms);
        }
    }
    printf("\nn = 10000, time ms by input order\n");
    printf("%-17s %11s %11s %11s\n", "algorithm", "random", "ascending",
           "descending");
    int n = 10000;
    state = 2026;
    vector<int> rnd(n), asc(n), desc(n);
    for (int i = 0; i < n; ++i) {
        rnd[i] = nextRandom(1000000000); asc[i] = i; desc[i] = n - i;
    }
    for (int k = 0; k < 3; ++k) {
        Counts c1, c2, c3;
        double t1 = run(algos[k].fn, rnd, c1), t2 = run(algos[k].fn, asc, c2),
               t3 = run(algos[k].fn, desc, c3);
        printf("%-17s %11.2f %11.2f %11.2f\n", algos[k].name, t1, t2, t3);
    }
    return 0;
}
```

The measurements below were taken on a shared cloud server with an Intel Xeon processor at 2.1 GHz, using one core, with `g++` 13.3 at `-O2`, OpenJDK 21 and Python 3.11. Times on your computer will differ, and even on this one a second run differs by ten or twenty percent; the counts, by contrast, are identical on every machine and in every language, because every version sorts the same numbers from the same generator.

```
algorithm                n      comparisons     swaps/moves     time ms
bubble                1000           499290          252221        1.74
selection             1000           499500             987        1.18
insertion             1000           253216          252221        0.17
shell                 1000            13187            6893        0.06
merge                 1000             8723            9976        0.06
quick                 1000            10666            3682        0.06
heap                  1000            16846            9095        0.06
radix                 1000                0            4000        0.01
std::sort             1000            11870               -        0.04
std::stable_sort      1000             9864               -        0.05
bubble               10000         49994994        24649368      190.22
selection            10000         49995000            9982      132.19
insertion            10000         24659362        24649368       15.98
shell                10000           190702          101054        0.91
merge                10000           120445          133616        0.78
quick                10000           151336           62176        0.57
heap                 10000           235396          124322        0.82
radix                10000                0           40000        0.07
std::sort            10000           156934               -        0.49
std::stable_sort     10000           128101               -        0.57
bubble              100000       4999861169      2500232977    28013.78
selection           100000       4999950000           99988    12961.01
insertion           100000       2500332969      2500232977     2841.53
shell               100000          2554967         1384924       13.34
merge               100000          1536335         1668928        9.91
quick               100000          1881706          828252        7.76
heap                100000          3019684         1574735       10.56
radix               100000                0          400000        0.84
std::sort           100000          1956930               -        5.87
std::stable_sort    100000          1595435               -        7.58
shell              1000000         32014545        17456447      158.35
merge              1000000         18675220        19951424      110.50
quick              1000000         22507428        10026671       85.28
heap               1000000         36794517        19047984      180.60
radix              1000000                0         4000000       14.37
std::sort          1000000         23732145               -       85.77
std::stable_sort   1000000         19824445               -      125.10

n = 10000, time ms by input order
algorithm              random   ascending  descending
bubble                265.30        0.01      337.10
selection             124.65      138.71      128.45
insertion              17.85        0.01       35.96
```

3.13.2 Reading the numbers

Figure 3.15 plots the times on logarithmic axes, where a power law $t = c\,n^{p}$ becomes a straight line whose slope is p: slope 2 for a quadratic algorithm, slope just above 1 for $n\lg n$.

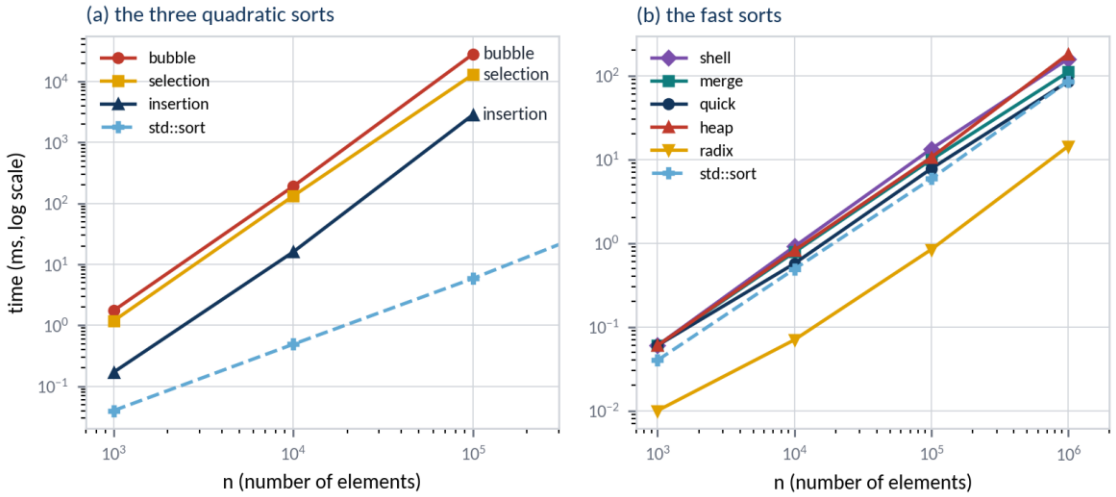


Figure 3.15 Measured times of the C++ programs, on log–log axes. (a) The three quadratic sorts climb two decades of time for each decade of n; `std::sort` is shown for scale. (b) The fast sorts climb only slightly more than one decade per decade; radix sort is fastest.

The quadratic sorts behave exactly as Equation (3.3) **predicts.** Bubble sort's comparisons are 499,290, 49,994,994 and 4,999,861,169: each is almost exactly $n(n-1)/2$, slightly less because the early exit saved the last few passes. The swaps, 252,221 for $n = 1000$, are the number of inversions of those random numbers, very close to the $n(n-1)/4 = 249{,}750$ of Equation (3.5). Insertion sort's shifts are *identical* to bubble sort's swaps at every size, as Equation (3.4) and Section 3.6.4 said they must be: both equal the number of inversions. Selection sort makes 25,000 times fewer swaps than bubble sort at $n = 10^5$ and is still slow, because its comparisons are the same five billion.

Time grows faster than the counts. From 10^4 to 10^5 the counts grow by exactly 100, but bubble sort's time grows from 190 ms to 28 s, about 147 times, and insertion sort's by about 178 times. The extra factor comes from the machine, not the algorithm: 10^4 integers (40 KB) fit in the processor's fastest cache, 10^5 integers (400 KB) do not, and each memory access costs more. Insertion sort is still ten times faster than bubble sort with the same number of moves, because a shift is one write while a swap is three, and because its inner loop stops as soon as the key is placed.

The $n\lg n$ sorts grow gently. Merge sort takes 0.78 ms, 9.9 ms and 110 ms for 10^4, 10^5 and 10^6 elements, a factor of 12 to 13 per tenfold increase, where $n\lg n$ predicts $10 \times \lg(10^5)/\lg(10^4) = 12.5$ and $10 \times \lg(10^6)/\lg(10^5) = 12$. At a million elements, where bubble sort would need almost an hour, every $n\lg n$ sort finishes in less than a fifth of a second.

Comparisons are not the whole story. **Figure 3.16** divides the comparisons by n. For the $n\lg n$ sorts the result grows like $\lg n$ (the dotted line); for the quadratic ones it grows like n.

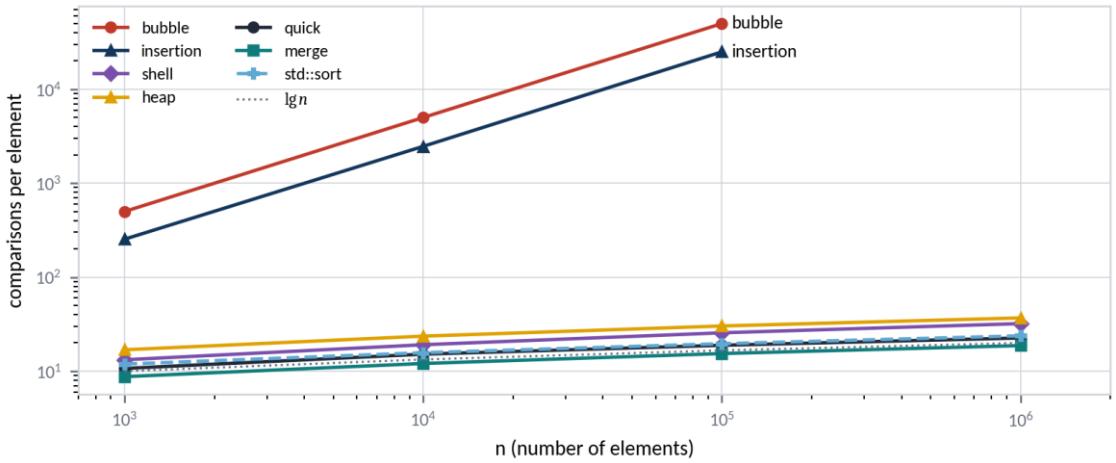


Figure 3.16 Comparisons per element. The quadratic sorts climb steeply; the others stay close to lg$n$, with merge sort the most economical and heap sort the least.

Merge sort makes the fewest comparisons of all (18.7 million at $n = 10^6$, just below $n\lg n = 19.9$ million), and heap sort the most (36.8 million, almost twice as many). Yet the fastest comparison sorts at 10^6 are quicksort and `std::sort`, at about 85 ms, ahead of merge sort at 110 ms and heap sort at 181 ms. Quicksort does more comparisons than merge sort but moves half as much data, needs no buffer, and scans memory from left to right, which the cache rewards; heap sort jumps from i to $2i + 1$ and loses to both. Radix sort, which makes no comparisons at all, sorts a million integers in 14 ms, six times faster than `std::sort`. Its limitation is not speed but generality: it only sorts things that can be cut into digits.

3.13.3 Input order matters for some sorts

The last table of the output repeats, on today's machine, a measurement that has been used to teach this chapter for many years. **Table 3.4** is the old version: 10,000 numbers on a Pentium II processor at 450 MHz, a typical desktop computer of 1998.

Table 3.4 A historical measurement: sorting 10,000 numbers on a Pentium II at 450 MHz (late 1990s). Selection sort's descending time repeats bubble sort's and is almost certainly a copying slip; its time does not depend on the order of the input.

Algorithm	Random	Ascending	Descending
Bubble sort	11.20 s	1.32 s	19.77 s
Insertion sort	1.09 s	0.00 s	2.25 s
Selection sort	1.32 s	1.32 s	19.77 s

The pattern is the same today, in C++ on one core, as **Table 3.5** shows:

Table 3.5 The same experiment in 2026 (from the output of **Listing 3.34**). Absolute times fell by a factor of 10 to 60; the relationships between the algorithms did not change.

Algorithm	Random	Ascending	Descending
Bubble sort	265 ms	0.01 ms	337 ms
Insertion sort	17.9 ms	0.01 ms	36.0 ms
Selection sort	125 ms	139 ms	128 ms

Bubble sort and insertion sort are adaptive: on ascending input bubble sort's early exit stops after one pass, and insertion sort's inner loop never runs, so both take about ten microseconds. On descending input both do the maximum work, and insertion sort's twice-as-fast random case is explained by Equation (3.5): random input has half the inversions of reversed input. Selection

sort takes the same time whatever the order. (The old bubble sort's 1.32 s on sorted input shows that it had no early exit.) Hardware made these loops ten to sixty times faster in a quarter of a century; the difference between a quadratic and an $n \lg n$ algorithm at a million elements is a factor of *twenty-five thousand* (**Table 3.1**). Better algorithms beat better hardware.

3.13.4 The same benchmark in Java and Python

Listing 3.35 is the Java version. In place of `std::sort` it measures the two library sorts Java offers: `Arrays.sort` on an `int[]`, and `Arrays.sort` on an array of `Integer` objects with a counting comparator.

Listing 3.35 `SortBenchmark.java`: the benchmark in Java, including `Arrays.sort` on primitives and on objects.

```
// SortBenchmark.java - counts and times for every sort in this chapter
import java.util.Arrays;

public class SortBenchmark {
    static long cmp, mov;  // comparisons, swaps or moves

    static long state = 2026;
    static int nextRandom(int bound) {
        state = state * 6364136223846793005L + 1442695040888963407L;
        return (int) ((state >>> 33) % bound);
    }

    static void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    static void bubble(int[] a) {
        int n = a.length;
        for (int pass = 1; pass < n; pass++) {
            boolean swapped = false;
            for (int j = 0; j < n - pass; j++) {
                cmp++;
                if (a[j] > a[j + 1]) {
                    swap(a, j, j + 1); mov++;
                    swapped = true;
                }
            }
            if (!swapped) break;
        }
    }

    static void selection(int[] a) {
        int n = a.length;
        for (int i = 0; i < n - 1; i++) {
            int pos = i;
            for (int j = i + 1; j < n; j++) {
                cmp++;
                if (a[j] < a[pos]) pos = j;
            }
            if (pos != i) { swap(a, i, pos); mov++; }
        }
    }
```

Listing 3.35 (Continued) *SortBenchmark.java: the benchmark in Java, including Arrays.sort on primitives and on objects.*

```
    static void gappedInsertion(int[] a, int gap) {
        for (int i = gap; i < a.length; i++) {
            int key = a[i], j = i - gap;
            while (j >= 0) {
                cmp++;
                if (a[j] <= key) break;
                a[j + gap] = a[j]; mov++;
                j -= gap;
            }
            a[j + gap] = key;
        }
    }

    static void insertion(int[] a) { gappedInsertion(a, 1); }

    static void shell(int[] a) {  // Ciura's gaps, extended by x2.25
        int[] gaps = new int[20];                    // room for extra gaps
        int[] ciura = {1, 4, 10, 23, 57, 132, 301, 701, 1750};
        System.arraycopy(ciura, 0, gaps, 0, ciura.length);
        int count = 9;
        while (gaps[count - 1] * 9 / 4 < a.length) {
            gaps[count] = gaps[count - 1] * 9 / 4;
            count++;
        }
        for (int k = count - 1; k >= 0; k--)
            if (gaps[k] < a.length) gappedInsertion(a, gaps[k]);
    }

    static void mergeRec(int[] a, int[] t, int lo, int hi) {
        if (hi - lo < 2) return;
        int mid = lo + (hi - lo) / 2;
        mergeRec(a, t, lo, mid);
        mergeRec(a, t, mid, hi);
        int i = lo, j = mid, k = lo;
        while (i < mid && j < hi) {
            cmp++;
            t[k++] = (a[i] <= a[j]) ? a[i++] : a[j++];
        }
        while (i < mid) t[k++] = a[i++];
        while (j < hi) t[k++] = a[j++];
        for (k = lo; k < hi; k++) { a[k] = t[k]; mov++; }
    }
```

Listing 3.35 (Continued) *SortBenchmark.java: the benchmark in Java, including Arrays.sort on primitives and on objects.*

```
    static void mergeSort(int[] a) {
        mergeRec(a, new int[a.length], 0, a.length);
    }

    static void quickRec(int[] a, int lo, int hi) {
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;            // median of three to a[hi]
            if (a[mid] < a[lo]) swap(a, mid, lo);
            if (a[hi] < a[lo]) swap(a, hi, lo);
            if (a[mid] < a[hi]) swap(a, mid, hi);
            cmp += 3;
            int pivot = a[hi], i = lo - 1;
            for (int j = lo; j < hi; j++) {
                cmp++;
                if (a[j] <= pivot) {
                    i++;
                    if (i != j) { swap(a, i, j); mov++; }
                }
            }
            swap(a, i + 1, hi); mov++;
            int p = i + 1;
            // recurse into the smaller side, loop on the larger one
            if (p - lo < hi - p) { quickRec(a, lo, p - 1); lo = p + 1; }
            else                 { quickRec(a, p + 1, hi); hi = p - 1; }
        }
    }

    static void quick(int[] a) { quickRec(a, 0, a.length - 1); }

    static void siftDown(int[] a, int i, int size) {
        while (true) {
            int largest = i, l = 2 * i + 1, r = l + 1;
            if (l < size) { cmp++; if (a[l] > a[largest]) largest = l; }
            if (r < size) { cmp++; if (a[r] > a[largest]) largest = r; }
            if (largest == i) return;
            swap(a, i, largest); mov++;
            i = largest;
        }
    }
```

Listing 3.35 (Continued) *SortBenchmark.java: the benchmark in Java, including Arrays.sort on primitives and on objects.*

```
    static void heap(int[] a) {
        int n = a.length;
        for (int i = n / 2 - 1; i >= 0; i--) siftDown(a, i, n);
        for (int end = n - 1; end > 0; end--) {
            swap(a, 0, end); mov++;
            siftDown(a, 0, end);
        }
    }

    static void radix(int[] a) {  // base 256, four passes, no comparisons
        int[] out = new int[a.length];
        for (int shift = 0; shift < 32; shift += 8) {
            int[] count = new int[257];
            for (int x : a) count[((x >> shift) & 255) + 1]++;
            for (int v = 0; v < 256; v++) count[v + 1] += count[v];
            for (int x : a) { out[count[(x >> shift) & 255]++] = x; mov++; }
            System.arraycopy(out, 0, a, 0, a.length);
        }
    }

    static void arraysSortInt(int[] a) { cmp = -1; mov = -1; Arrays.sort(a); }

    static void arraysSortObj(int[] a) {  // TimSort on Integer objects
        Integer[] boxed = new Integer[a.length];
        for (int i = 0; i < a.length; i++) boxed[i] = a[i];
        mov = -1;
        Arrays.sort(boxed, (x, y) -> { cmp++; return Integer.compare(x, y); });
        for (int i = 0; i < a.length; i++) a[i] = boxed[i];
    }

    interface Sorter { void sort(int[] a); }
    record Algo(String name, Sorter fn, int maxN) { }

    static String cell(long v) { return v < 0 ? "-" : Long.toString(v); }

    static double run(Sorter fn, int[] data) {
        int[] a = data.clone();
        cmp = 0; mov = 0;
        long t0 = System.nanoTime();
        fn.sort(a);
        long t1 = System.nanoTime();
        for (int i = 1; i < a.length; i++)
            if (a[i - 1] > a[i]) { System.out.println("  NOT SORTED!"); break; }
        return (t1 - t0) / 1e6;
    }
```

Listing 3.35 (Continued) *SortBenchmark.java: the benchmark in Java, including Arrays.sort on primitives and on objects.*

```
    public static void main(String[] args) {
        Algo[] algos = {
            new Algo("bubble", SortBenchmark::bubble, 100000),
            new Algo("selection", SortBenchmark::selection, 100000),
            new Algo("insertion", SortBenchmark::insertion, 100000),
            new Algo("shell", SortBenchmark::shell, 1000000),
            new Algo("merge", SortBenchmark::mergeSort, 1000000),
            new Algo("quick", SortBenchmark::quick, 1000000),
            new Algo("heap", SortBenchmark::heap, 1000000),
            new Algo("radix", SortBenchmark::radix, 1000000),
            new Algo("Arrays.sort int", SortBenchmark::arraysSortInt, 1000000),
            new Algo("Arrays.sort Obj", SortBenchmark::arraysSortObj, 1000000),
        };
        System.out.printf("%-17s %8s %15s %15s %11s%n", "algorithm", "n",
                "comparisons", "swaps/moves", "time ms");
        for (int n = 1000; n <= 1000000; n *= 10) {
            state = 2026;
            int[] data = new int[n];
            for (int i = 0; i < n; i++) data[i] = nextRandom(1000000000);
            for (Algo al : algos) {
                if (n > al.maxN()) continue;
                double ms = run(al.fn(), data);
                System.out.printf("%-17s %8d %15s %15s %11.2f%n",
                        al.name(), n, cell(cmp), cell(mov), ms);
            }
        }
        System.out.printf("%nn = 10000, time ms by input order%n");
        System.out.printf("%-17s %11s %11s %11s%n", "algorithm", "random",
                "ascending", "descending");
        int n = 10000;
        state = 2026;
        int[] rnd = new int[n], asc = new int[n], desc = new int[n];
        for (int i = 0; i < n; i++) {
            rnd[i] = nextRandom(1000000000); asc[i] = i; desc[i] = n - i;
        }
        for (int k = 0; k < 3; k++) {
            double t1 = run(algos[k].fn(), rnd), t2 = run(algos[k].fn(), asc),
                   t3 = run(algos[k].fn(), desc);
            System.out.printf("%-17s %11.2f %11.2f %11.2f%n",
                    algos[k].name(), t1, t2, t3);
        }
    }
}
```

```
algorithm                n      comparisons     swaps/moves     time ms
bubble                1000           499290          252221        9.23
selection             1000           499500             987        6.98
insertion             1000           253216          252221        8.10
shell                 1000            13187            6893        0.10
merge                 1000             8723            9976        4.23
quick                 1000            10666            3682        0.50
heap                  1000            16846            9095        0.85
radix                 1000                0            4000        0.23
Arrays.sort int       1000                -               -        4.75
Arrays.sort Obj       1000             8690               -        2.02
bubble               10000         49994994        24649368      186.51
selection            10000         49995000            9982       77.25
insertion            10000         24659362        24649368       32.82
shell                10000           190702          101054        1.97
merge                10000           120445          133616        1.23
quick                10000           151336           62176        5.62
heap                 10000           235396          124322        2.12
radix                10000                0           40000        9.73
Arrays.sort int      10000                -               -       13.57
Arrays.sort Obj      10000           120420               -       32.26
bubble              100000       4999861169      2500232977    19339.70
selection           100000       4999950000           99988     3744.82
insertion           100000       2500332969      2500232977     3764.32
shell               100000          2554967         1384924       11.34
merge               100000          1536335         1668928       13.89
quick               100000          1881706          828252        9.06
heap                100000          3019684         1574735       16.62
radix               100000                0          400000        9.32
Arrays.sort int     100000                -               -       15.90
Arrays.sort Obj     100000          1535047               -       67.50
shell              1000000         32014545        17456447      207.64
merge              1000000         18675220        19951424      141.01
quick              1000000         22507428        10026671       95.51
heap               1000000         36794517        19047984      199.82
radix              1000000                0         4000000       20.43
Arrays.sort int    1000000                -               -      226.85
Arrays.sort Obj    1000000         18640977               -      734.53

n = 10000, time ms by input order
algorithm              random   ascending  descending
bubble                 120.72        0.01       82.30
selection               23.93       25.11       28.27
insertion               48.96        0.03       74.02
```

The counts are identical to the C++ counts, digit for digit. The times tell two lessons about Java. First, our own sorts run at nearly C++ speed once the just-in-time compiler has translated them to machine code, but the early, small measurements are distorted by that compilation, which is why the times for $n = 1000$ are erratic. Second, the library call `Arrays.sort` was measured only once per size, cold, and a single cold call is a poor measurement; reliable Java timing needs a harness such as JMH that runs the code many times first. Sorting `Integer` objects is several times slower than sorting `int`s, because each element is a separate object reached through a reference, and comparing two of them means following two references. One curiosity: Java's bubble sort was *faster* on descending input than on random input. On descending input every comparison says "swap", the processor predicts that perfectly, and it never has to discard work it started speculatively; on random input its guesses are wrong half the time.

Listing 3.36 is the Python version. Pure Python executes each basic step through the interpreter, roughly twenty to eighty times more slowly than compiled code, so the quadratic sorts stop at $n = 10^4$.

Listing 3.36 `sort_benchmark.py`: the benchmark in Python; the quadratic sorts stop at ten thousand elements.

```
# sort_benchmark.py - counts and times for every sort in this chapter
import time

cmp = mov = 0                           # comparisons, swaps or moves
state = 2026


def next_random(bound):
    global state
    state = (state * 6364136223846793005 + 1442695040888963407) % 2**64
    return (state >> 33) % bound


def bubble(a):
    global cmp, mov
    n = len(a)
    for pass_no in range(1, n):
        swapped = False
        for j in range(n - pass_no):
            cmp += 1
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                mov += 1
                swapped = True
        if not swapped:
            break


def selection(a):
    global cmp, mov
    n = len(a)
    for i in range(n - 1):
        pos = i
        for j in range(i + 1, n):
            cmp += 1
            if a[j] < a[pos]:
                pos = j
        if pos != i:
            a[i], a[pos] = a[pos], a[i]
            mov += 1
```

Listing 3.36 (Continued) *sort_benchmark.py: the benchmark in Python; the quadratic sorts stop at ten thousand elements.*

```
def gapped_insertion(a, gap):
    global cmp, mov
    for i in range(gap, len(a)):
        key, j = a[i], i - gap
        while j >= 0:
            cmp += 1
            if a[j] <= key:
                break
            a[j + gap] = a[j]
            mov += 1
            j -= gap
        a[j + gap] = key


def insertion(a):
    gapped_insertion(a, 1)


def shell(a):                              # Ciura's gaps, extended by x2.25
    gaps = [1, 4, 10, 23, 57, 132, 301, 701, 1750]
    while gaps[-1] * 9 // 4 < len(a):
        gaps.append(gaps[-1] * 9 // 4)
    for gap in reversed(gaps):
        if gap < len(a):
            gapped_insertion(a, gap)


def merge_rec(a, t, lo, hi):
    global cmp, mov
    if hi - lo < 2:
        return
    mid = lo + (hi - lo) // 2
    merge_rec(a, t, lo, mid)
    merge_rec(a, t, mid, hi)
    i, j, k = lo, mid, lo
    while i < mid and j < hi:
        cmp += 1
        if a[i] <= a[j]:
            t[k] = a[i]
            i += 1
        else:
            t[k] = a[j]
            j += 1
        k += 1
    t[k:hi] = a[i:mid] + a[j:hi]
    a[lo:hi] = t[lo:hi]
```

Listing 3.36 (Continued) *sort_benchmark.py: the benchmark in Python; the quadratic sorts stop at ten thousand elements.*

```
    mov += hi - lo


def merge_sort(a):
    merge_rec(a, [0] * len(a), 0, len(a))


def quick_rec(a, lo, hi):
    global cmp, mov
    while lo < hi:
        mid = lo + (hi - lo) // 2        # median of three to a[hi]
        if a[mid] < a[lo]:
            a[mid], a[lo] = a[lo], a[mid]
        if a[hi] < a[lo]:
            a[hi], a[lo] = a[lo], a[hi]
        if a[mid] < a[hi]:
            a[mid], a[hi] = a[hi], a[mid]
        cmp += 3
        pivot, i = a[hi], lo - 1
        for j in range(lo, hi):
            cmp += 1
            if a[j] <= pivot:
                i += 1
                if i != j:
                    a[i], a[j] = a[j], a[i]
                    mov += 1
        a[i + 1], a[hi] = a[hi], a[i + 1]
        mov += 1
        p = i + 1
        if p - lo < hi - p:              # recurse on the smaller side
            quick_rec(a, lo, p - 1)
            lo = p + 1
        else:
            quick_rec(a, p + 1, hi)
            hi = p - 1


def quick(a):
    quick_rec(a, 0, len(a) - 1)
```

Listing 3.36 (Continued) *sort_benchmark.py: the benchmark in Python; the quadratic sorts stop at ten thousand elements.*

```
def sift_down(a, i, size):
    global cmp, mov
    while True:
        largest, l, r = i, 2 * i + 1, 2 * i + 2
        if l < size:
            cmp += 1
            if a[l] > a[largest]:
                largest = l
        if r < size:
            cmp += 1
            if a[r] > a[largest]:
                largest = r
        if largest == i:
            return
        a[i], a[largest] = a[largest], a[i]
        mov += 1
        i = largest


def heap(a):
    global mov
    n = len(a)
    for i in range(n // 2 - 1, -1, -1):
        sift_down(a, i, n)
    for end in range(n - 1, 0, -1):
        a[0], a[end] = a[end], a[0]
        mov += 1
        sift_down(a, 0, end)


def radix(a):                           # base 256, four passes, no comparisons
    global mov
    for shift in (0, 8, 16, 24):
        count = [0] * 257
        for x in a:
            count[((x >> shift) & 255) + 1] += 1
        for v in range(256):
            count[v + 1] += count[v]
        out = [0] * len(a)
        for x in a:
            d = (x >> shift) & 255
            out[count[d]] = x
            count[d] += 1
            mov += 1
        a[:] = out
```

Listing 3.36 (Continued) *sort_benchmark.py: the benchmark in Python; the quadratic sorts stop at ten thousand elements.*

```
def list_sort(a):  # Timsort in C: nothing to count from here
    global cmp, mov
    cmp = mov = -1
    a.sort()


def cell(v):
    return "-" if v < 0 else str(v)


def run(fn, data):
    global cmp, mov
    a = data[:]
    cmp = mov = 0
    t0 = time.perf_counter()
    fn(a)
    t1 = time.perf_counter()
    if any(a[i - 1] > a[i] for i in range(1, len(a))):
        print("  NOT SORTED!")
    return (t1 - t0) * 1000


ALGOS = [("bubble", bubble, 10000), ("selection", selection, 10000),
         ("insertion", insertion, 10000), ("shell", shell, 1000000),
         ("merge", merge_sort, 1000000), ("quick", quick, 1000000),
         ("heap", heap, 1000000), ("radix", radix, 1000000),
         ("list.sort", list_sort, 1000000)]

print(f"{'algorithm':<17} {'n':>8} {'comparisons':>15} {'swaps/moves':>15} "
      f"{'time ms':>11}")
n = 1000
while n <= 1000000:
    state = 2026
    data = [next_random(1000000000) for _ in range(n)]
    for name, fn, max_n in ALGOS:
        if n <= max_n:
            ms = run(fn, data)
            print(f"{name:<17} {n:>8} {cell(cmp):>15} {cell(mov):>15} "
                  f"{ms:>11.2f}")
    n *= 10
```

Listing 3.36 (Continued) *sort_benchmark.py: the benchmark in Python; the quadratic sorts stop at ten thousand elements.*

```
print("\nn = 10000, time ms by input order")
print(f"{'algorithm':<17} {'random':>11} {'ascending':>11} {'descending':>11}")
n = 10000
state = 2026
rnd = [next_random(1000000000) for _ in range(n)]
asc = list(range(n))
desc = [n - i for i in range(n)]
for name, fn, _ in ALGOS[:3]:
    t1, t2, t3 = run(fn, rnd), run(fn, asc), run(fn, desc)
    print(f"{name:<17} {t1:>11.2f} {t2:>11.2f} {t3:>11.2f}")
```

```
algorithm                n     comparisons     swaps/moves     time ms
bubble                1000          499290          252221       70.93
selection             1000          499500             987       18.72
insertion             1000          253216          252221       20.38
shell                 1000           13187            6893        1.22
merge                 1000            8723            9976        1.09
quick                 1000           10666            3682        0.82
heap                  1000           16846            9095        1.63
radix                 1000               0            4000        0.60
list.sort             1000               -               -        0.08
bubble               10000        49994994        24649368     4596.55
selection            10000        49995000            9982     2151.18
insertion            10000        24659362        24649368     2077.68
shell                10000          190702          101054       16.41
merge                10000          120445          133616       15.29
quick                10000          151336           62176       19.83
heap                 10000          235396          124322       41.10
radix                10000               0           40000        5.17
list.sort            10000               -               -        1.07
shell               100000         2554967         1384924      388.77
merge               100000         1536335         1668928      170.69
quick               100000         1881706          828252      146.43
heap                100000         3019684         1574735      297.72
radix               100000               0          400000       61.88
list.sort           100000               -               -       14.49
shell              1000000        32014545        17456447     7896.72
merge              1000000        18675220        19951424     3778.53
quick              1000000        22507428        10026671     4650.59
heap               1000000        36794517        19047984    14914.96
radix              1000000               0         4000000     1513.46
list.sort          1000000               -               -      307.26

n = 10000, time ms by input order
algorithm              random   ascending  descending
bubble                7287.39        0.46     7513.43
selection             2645.77     3273.60     3412.94
insertion             3371.85        0.74     6455.52
```

The counts again agree with C++ exactly. The instructive row is the last: Python's built-in `list.sort`, written in C, sorts a million numbers in about 0.3 s, while our pure-Python merge sort, with the same $n \lg n$ growth, needs almost 4 s. In Python the rule is plain: always use the built-in sort, and spend your own code on the key function.

3.14 Library Sorts Today

You should understand every algorithm in this chapter, and you should almost never write one in a program that other people depend on. The sorts in the standard libraries are faster, better tested and more carefully engineered than anything written in an afternoon, and they are hybrids that combine several of the algorithms you now know. **Table 3.6** summarizes them as of September 2026.

Table 3.6 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	$O(n \lg n)$

3.14.1 C++: introsort and merge sort

The C++ standard does not name an algorithm; it states requirements. `std::sort` must make $O(n \lg n)$ comparisons, and since C++11 that bound applies to the *worst* case; before the correction known as LWG 713 only the average was required, which allowed a plain quicksort [10]. Every major library meets the requirement with **introsort**, invented by Musser in 1997 [11]. Introsort is quicksort that watches its own recursion depth: if the depth exceeds a limit proportional to $\lg n$, the input is behaving badly, and that piece is finished with heap sort, whose worst case is $O(n \lg n)$. In GCC's library (libstdc++) the limit is $2\lfloor \lg n \rfloor$, the pivot is a median of three, partitioning stops on pieces of 16 or fewer elements, and one final insertion sort over the whole array finishes them all at once: by then every element is within 16 places of its home, so the adaptive insertion sort costs only $O(n)$ [12]. Three algorithms of this chapter, each doing what it does best.

`std::stable_sort` is a merge sort, which is why it is stable [13]. The standard requires $O(n \lg n)$ comparisons if it can obtain a buffer, and permits $O(n \lg^2 n)$ if it must merge in place. GCC's version first insertion-sorts chunks of 7 elements, then merges. `std::partial_sort(first, middle, last)` puts the smallest `middle - first` elements, sorted, at the front using a heap, which is the right tool for "the top ten" without sorting everything.

3.14.2 Java: dual-pivot quicksort and TimSort

For arrays of primitive types (`int`, `double` and so on), `Arrays.sort` uses a **dual-pivot quicksort** by Yaroslavskiy, Bentley and Bloch, which partitions around *two* pivots into three parts; its documentation promises $O(n \lg n)$ on all data sets [14]. The current implementation, rewritten for JDK 14, also switches to insertion sort for small parts, recognizes and merges already sorted runs, falls back to heap sort when the recursion becomes too deep, and uses counting sort for `byte`, `short` and `char` arrays, whose keys have at most 65,536 values [15]. Stability does not matter for primitives: two equal `int`s cannot be told apart.

For arrays of objects and for `List.sort` and `Collections.sort`, stability does matter, and the specification requires it. Since Java 7 the algorithm is **TimSort**, a port of Python's list sort [16]: a stable, adaptive, iterative merge sort that "requires far fewer than $n \lg n$ comparisons when

running on partially sorted arrays". It finds the runs that are already ascending (or strictly descending, which it reverses), extends short runs to a minimum length with binary insertion sort, and merges the runs; its minimum merge length is 32 elements.

3.14.3 Python: Timsort and Powersort

Python's `list.sort` and `sorted` are always stable, which the documentation guarantees, so multi-key sorting by successive sorts (Section 3.12.1) is officially supported [17]. The algorithm was designed by Tim Peters in 2002 and is described in his notes `listsort.txt` in the CPython source [18]. Like TimSort it scans for natural runs, lengthens short ones with binary insertion sort, and merges them, using *galloping* (exponential search followed by binary search) when one run keeps winning. What changed in **Python 3.11** is the *merge policy*, the rule deciding which neighboring runs to merge and when. The old rule had rare cases in which it merged badly; since 3.11 CPython uses the rule of Munro and Wild's **Powersort** [19], [20], which is provably within a small additive term of the best possible merge order for the given run lengths. The effect on ordinary data is small; the effect on the bad cases is large. The same notes record a further refinement merged into CPython's development branch in 2025, after the 3.14 feature freeze: the minimum run length is now varied slightly from run to run so that the merge tree is perfectly balanced [18].

Python's `sort` takes a **key function** instead of a comparison: `key=lambda b: b.category` computes each element's key once, which is both faster and easier to get right than comparing, and `reverse=True` sorts in descending order while *preserving* stability.

3.14.4 Using them

Listing 3.37, **Listing 3.38** and **Listing 3.39** do the same five jobs with each language's library: ascending, descending, the top three of a list of scores, a stable sort of the catalog by category, and a sort by title.

Listing 3.37 `library_sorts.cpp`: `std::sort` with and without a comparator, `std::partial_sort`, and `std::stable_sort` with a lambda.

```cpp
// library_sorts.cpp - the standard library sorts, with comparators
#include <algorithm>
#include <functional>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct Book {
    string id, title, category;
};

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

int main() {
    vector<int> data = {22, 10, 15, 3, 8, 2};
    sort(data.begin(), data.end());                    // introsort, ascending
    cout << "ascending:    " << show(data) << "\n";
    // the comparator decides the order
    sort(data.begin(), data.end(), greater<int>());
    cout << "descending:   " << show(data) << "\n";

    vector<int> scores = {71, 94, 58, 88, 94, 63, 79};
    partial_sort(scores.begin(), scores.begin() + 3, scores.end(),
                 greater<int>());
    vector<int> top3(scores.begin(), scores.begin() + 3);
    cout << "top 3 scores: " << show(top3) << "\n";

    vector<Book> catalog = {
        {"B1001", "Laskar Pelangi", "Fiksi"},
        {"B1002", "Bumi Manusia", "Fiksi"},
        {"B1003", "Filosofi Kopi", "Fiksi"},
        {"B1004", "Sejarah Nusantara", "Sejarah"},
        {"B1005", "Algoritma & Struktur Data", "Teknologi"},
        {"B1006", "Cerita Rakyat Nusantara", "Anak-anak"},
        {"B1007", "Fisika Dasar", "Sains"},
        {"B1008", "Negeri 5 Menara", "Fiksi"},
    };
    auto byCategory = [](const Book& a, const Book& b) {
        return a.category < b.category;
    };
    stable_sort(catalog.begin(), catalog.end(), byCategory);
```

Listing 3.37 (Continued) *library_sorts.cpp: std::sort with and without a comparator, std::partial_sort, and std::stable_sort with a lambda.*

```cpp
    cout << "by category, id order kept inside each category:\n";
    for (const Book& b : catalog)
        cout << "  " << b.category << " " << b.id << "\n";

    sort(catalog.begin(), catalog.end(),
         [](const Book& a, const Book& b) { return a.title < b.title; });
    cout << "by title:\n";
    for (const Book& b : catalog) cout << "  " << b.title << "\n";
    return 0;
}
```

Listing 3.38 `LibrarySorts.java`: `Arrays.sort` on `int[]` and on `Integer[]` with `Collections.reverseOrder()`, and `List.sort` with `Comparator.comparing`.

```java
// LibrarySorts.java - the standard library sorts, with comparators
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class LibrarySorts {
    record Book(String id, String title, String category) { }

    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    public static void main(String[] args) {
        int[] data = {22, 10, 15, 3, 8, 2};
        Arrays.sort(data);  // dual-pivot quicksort for int[]
        System.out.println("ascending:    " + show(data));
        Integer[] boxed = {22, 10, 15, 3, 8, 2};
        // objects: TimSort, with a comparator
        Arrays.sort(boxed, Collections.reverseOrder());
        System.out.println("descending:   "
                + Arrays.toString(boxed).replace(",", ""));

        Integer[] scores = {71, 94, 58, 88, 94, 63, 79};
        Arrays.sort(scores, Collections.reverseOrder());
        System.out.println("top 3 scores: "
                + Arrays.toString(Arrays.copyOf(scores, 3)).replace(",", ""));

        List<Book> catalog = new ArrayList<>(List.of(
            new Book("B1001", "Laskar Pelangi", "Fiksi"),
            new Book("B1002", "Bumi Manusia", "Fiksi"),
            new Book("B1003", "Filosofi Kopi", "Fiksi"),
            new Book("B1004", "Sejarah Nusantara", "Sejarah"),
            new Book("B1005", "Algoritma & Struktur Data", "Teknologi"),
            new Book("B1006", "Cerita Rakyat Nusantara", "Anak-anak"),
            new Book("B1007", "Fisika Dasar", "Sains"),
            new Book("B1008", "Negeri 5 Menara", "Fiksi")));
        // List.sort is always stable
        catalog.sort(Comparator.comparing(Book::category));
        System.out.println("by category, id order kept inside each category:");
        for (Book b : catalog) System.out.println("  " + b.category() + " "
                + b.id());
```

Listing 3.38 (Continued) *LibrarySorts.java: Arrays.sort on int[] and on Integer[] with Collections.reverseOrder(), and List.sort with Comparator.comparing.*

```java

        catalog.sort(Comparator.comparing(Book::title));
        System.out.println("by title:");
        for (Book b : catalog) System.out.println("  " + b.title());
    }
}
```

Listing 3.39 `library_sorts.py`: `list.sort`, `sorted` with a key, and `heapq.nlargest`.

```python
# library_sorts.py - the standard library sorts, with comparators
import heapq
from dataclasses import dataclass


@dataclass
class Book:
    id: str
    title: str
    category: str


def show(a):
    return "[" + " ".join(map(str, a)) + "]"


data = [22, 10, 15, 3, 8, 2]
data.sort()                                 # Timsort, in place, ascending
print(f"ascending:    {show(data)}")
data.sort(reverse=True)                     # still stable, just reversed order
print(f"descending:   {show(data)}")

scores = [71, 94, 58, 88, 94, 63, 79]
print(f"top 3 scores: {show(heapq.nlargest(3, scores))}")

catalog = [
    Book("B1001", "Laskar Pelangi", "Fiksi"),
    Book("B1002", "Bumi Manusia", "Fiksi"),
    Book("B1003", "Filosofi Kopi", "Fiksi"),
    Book("B1004", "Sejarah Nusantara", "Sejarah"),
    Book("B1005", "Algoritma & Struktur Data", "Teknologi"),
    Book("B1006", "Cerita Rakyat Nusantara", "Anak-anak"),
    Book("B1007", "Fisika Dasar", "Sains"),
    Book("B1008", "Negeri 5 Menara", "Fiksi"),
]
catalog.sort(key=lambda b: b.category)   # every Python sort is stable
print("by category, id order kept inside each category:")
for b in catalog:
    print(f"  {b.category} {b.id}")

print("by title:")
for b in sorted(catalog, key=lambda b: b.title):   # sorted() returns a new list
    print(f"  {b.title}")
```

```
ascending:    [2 3 8 10 15 22]
descending:   [22 15 10 8 3 2]
top 3 scores: [94 94 88]
by category, id order kept inside each category:
  Anak-anak B1006
  Fiksi B1001
  Fiksi B1002
  Fiksi B1003
  Fiksi B1008
  Sains B1007
  Sejarah B1004
  Teknologi B1005
by title:
  Algoritma & Struktur Data
  Bumi Manusia
  Cerita Rakyat Nusantara
  Filosofi Kopi
  Fisika Dasar
  Laskar Pelangi
  Negeri 5 Menara
  Sejarah Nusantara
```

In C++ the comparator must be a **strict weak ordering**: it answers "is a strictly before b?", so it uses `<`, never `<=`. A comparator that returns true for equal elements breaks the library's assumptions, and `std::sort` may then read outside the array. Java has no partial sort, so **Listing 3.38** sorts all seven scores and takes the first three; Python's `heapq.nlargest` uses a heap exactly as `std::partial_sort` does.

3.15 Profitina in Practice

Profitina is an AI-visibility platform: it scans a brand's website, scores how ready its pages are to be quoted by AI answer engines, explains each signal it measured, and helps the owner fix what it found. Its web application is written in Python with FastAPI and stores its data in PostgreSQL. Sorting appears in it at every level, usually invisibly.

The signals list. A scan produces a list of findings, each with a severity and an estimated impact on the score. A non-technical brand owner should see the most important fix first, so the list is sorted by impact, highest first. Two findings often have the same impact; the order among them should not change each time the page is reloaded, because a list that reshuffles itself on every visit looks unreliable. A stable sort, applied to findings that arrive in a fixed order (by signal type), gives exactly that: equal impacts keep a predictable order. In Python this is one line, `sorted(findings, key=lambda f: f.impact, reverse=True)`, because `reverse=True` keeps equal keys in their original order (Section 3.14.3).

The weekly trend. The historical dashboard shows each brand's score week by week, and the scheduler adds a new score after each automatic re-scan. The data therefore arrives *almost sorted by time*: a long ascending run with, occasionally, a late result from a delayed scan appended at the end. That is the best case of an adaptive sort. Timsort detects the long run in one pass and merges the few stragglers into it, so sorting the trend costs close to n comparisons rather than $n \lg n$, which is why re-sorting the series every time a point is added costs almost nothing.

Leaderboards and rankings. Users earn PROFIT by improving AI responses, and a leaderboard ranks them. Ranking by a primary key (points) with a tie-breaker (who reached the total first) is the multi-key sort of Section 3.12.1, and the page only needs the top few entries, which is the job of a heap: `heapq.nlargest(10, …)` in Python, `std::partial_sort` in C++, $O(n \lg k)$ instead of $O(n \lg n)$.

In the database. When a query says `ORDER BY`, PostgreSQL sorts. Its `EXPLAIN ANALYZE` output names the method it chose: `quicksort` when the rows fit in the memory allowed for sorting, `top-N heapsort` when the query has a `LIMIT` and only the first rows are needed, and `external merge` when the rows do not fit in memory and are sorted in pieces on disk and then merged [21]. Those are quicksort, heap sort and merge sort, the three $n \lg n$ algorithms of this chapter, chosen automatically according to exactly the trade-offs of **Table 3.2**. Knowing them lets a developer read a slow query plan and see, for instance, that adding an index would let the database skip the sort altogether.

The runnable part of this section uses **Profitina Laundry**, the small laundry business that serves as this book's worked case study. Its twelve orders of one day arrive in time order, as in Chapter 2. At the start of the washing shift the staff want the queue with the heaviest loads first, and among loads of equal weight the customer who came first should be served first; at the counter they want the pickup list alphabetical. The first request is a stable sort by weight, descending.

Listing 3.40 `laundry_orders.cpp`: Profitina Laundry's washing queue with `std::stable_sort`, and its pickup list with `std::sort`.

```cpp
// laundry_orders.cpp - Profitina Laundry: heaviest first, ties by arrival
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

struct Order {
    int number;
    string time, customer;
    double kg;
};

void print(const char* heading, const vector<Order>& orders) {
    printf("%s\n", heading);
    for (const Order& o : orders)
        printf("  #%d %s %-5s %4.1f kg\n", o.number, o.time.c_str(),
               o.customer.c_str(), o.kg);
}

int main() {
    vector<Order> orders = {  // in the order they arrived
        {1001, "07:45", "Sari", 3.5}, {1002, "08:10", "Budi", 5.0},
        {1003, "08:55", "Dewi", 2.0}, {1004, "09:30", "Andi", 7.5},
        {1005, "10:05", "Rina", 4.0}, {1006, "10:40", "Joko", 6.0},
        {1007, "11:15", "Maya", 3.0}, {1008, "11:50", "Tono", 8.0},
        {1009, "12:20", "Lina", 2.5}, {1010, "13:05", "Agus", 5.5},
        {1011, "14:30", "Fitri", 4.5}, {1012, "15:10", "Hadi", 3.0},
    };
    vector<Order> byWeight = orders;
    stable_sort(byWeight.begin(), byWeight.end(),
                [](const Order& a, const Order& b) { return a.kg > b.kg; });
    print("Washing queue, heaviest first (equal weights keep arrival order):",
          byWeight);

    vector<Order> byName = orders;
    sort(byName.begin(), byName.end(), [](const Order& a, const Order& b) {
        return a.customer < b.customer;
    });
    printf("Pickup list:");
    for (const Order& o : byName) printf(" %s", o.customer.c_str());
    printf("\n");
    return 0;
}
```

Listing 3.41 `LaundryOrders.java`: the same queue with `List.sort`, which is stable, and `Comparator.comparingDouble(...).reversed()`.

```java
// LaundryOrders.java - Profitina Laundry: heaviest first, ties by arrival
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class LaundryOrders {
    record Order(int number, String time, String customer, double kg) { }

    static void print(String heading, List<Order> orders) {
        System.out.println(heading);
        for (Order o : orders)
            System.out.printf("  #%d %s %-5s %4.1f kg%n",
                    o.number(), o.time(), o.customer(), o.kg());
    }

    public static void main(String[] args) {
        List<Order> orders = List.of(  // in the order they arrived
            new Order(1001, "07:45", "Sari", 3.5),
            new Order(1002, "08:10", "Budi", 5.0),
            new Order(1003, "08:55", "Dewi", 2.0),
            new Order(1004, "09:30", "Andi", 7.5),
            new Order(1005, "10:05", "Rina", 4.0),
            new Order(1006, "10:40", "Joko", 6.0),
            new Order(1007, "11:15", "Maya", 3.0),
            new Order(1008, "11:50", "Tono", 8.0),
            new Order(1009, "12:20", "Lina", 2.5),
            new Order(1010, "13:05", "Agus", 5.5),
            new Order(1011, "14:30", "Fitri", 4.5),
            new Order(1012, "15:10", "Hadi", 3.0));

        List<Order> byWeight = new ArrayList<>(orders);
        // stable
        byWeight.sort(Comparator.comparingDouble(Order::kg).reversed());
        print("Washing queue, heaviest first (equal weights keep arrival "
                + "order):", byWeight);

        List<Order> byName = new ArrayList<>(orders);
        byName.sort(Comparator.comparing(Order::customer));
        StringBuilder sb = new StringBuilder("Pickup list:");
        for (Order o : byName) sb.append(' ').append(o.customer());
        System.out.println(sb);
    }
}
```

Listing 3.42 `laundry_orders.py`: the same queue with `sorted(..., reverse=True)`.

```python
# laundry_orders.py - Profitina Laundry: heaviest first, ties by arrival
# in the order they arrived
ORDERS = [(1001, "07:45", "Sari", 3.5), (1002, "08:10", "Budi", 5.0),
          (1003, "08:55", "Dewi", 2.0), (1004, "09:30", "Andi", 7.5),
          (1005, "10:05", "Rina", 4.0), (1006, "10:40", "Joko", 6.0),
          (1007, "11:15", "Maya", 3.0), (1008, "11:50", "Tono", 8.0),
          (1009, "12:20", "Lina", 2.5), (1010, "13:05", "Agus", 5.5),
          (1011, "14:30", "Fitri", 4.5), (1012, "15:10", "Hadi", 3.0)]

# 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)

by_name = sorted(ORDERS, key=lambda o: o[2])
print("Pickup list:", " ".join(o[2] for o in by_name))
```

Listing 3.41 and **Listing 3.42** print exactly what **Listing 3.40** prints:

```
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
  #1011 14:30 Fitri  4.5 kg
  #1005 10:05 Rina   4.0 kg
  #1001 07:45 Sari   3.5 kg
  #1007 11:15 Maya   3.0 kg
  #1012 15:10 Hadi   3.0 kg
  #1009 12:20 Lina   2.5 kg
  #1003 08:55 Dewi   2.0 kg
Pickup list: Agus Andi Budi Dewi Fitri Hadi Joko Lina Maya Rina Sari Tono
```

Two orders weigh 3.0 kg: Maya's, which arrived at 11:15, and Hadi's, at 15:10. In the washing queue Maya's comes first, because every one of the three programs used a stable sort and the input was in arrival order. With `std::sort` in **Listing 3.40** that order would not be guaranteed. The C++ comparator `a.kg > b.kg` is still a strict weak ordering: "heavier than" is strict, so equal weights compare as neither before nor after.

3.16 Common Mistakes, and How to Avoid Them

1. Exchanging without a temporary. `a = b; b = a;` leaves both variables holding the old `b`. *Fix:* save the first value before overwriting it (Section 3.2.4), or use `std::swap` in C++ and tuple assignment in Python.

2. Writing a C++ or Java swap function that receives values. `void swap(int a, int b)` exchanges its own copies and changes nothing for the caller. *Fix:* in C++ pass pointers or references; in Java exchange two slots of an array that the method receives.

3. Running the inner loop one step too far. Bubble sort compares `a[j]` with `a[j+1]`, so `j` must stop at `n - pass - 1`; a loop to `n - 1` reads `a[n]`, past the end. *Fix:* for every loop that reads `a[j+1]`, check the largest `j` it reaches.

4. Testing the element before the index. `while (a[j] > key && j >= 0)` reads `a[-1]` when the key belongs at the front. *Fix:* test the index first, `j >= 0 && a[j] > key`, and rely on short-circuit evaluation (Section 3.6.2). In Python the wrong order does not crash; it silently compares with the last element.

5. Forgetting the early exit, or putting it in the wrong place. A bubble sort without the `swapped` flag is never adaptive. A flag that is reset only once, before the outer loop, never becomes false again. *Fix:* set `swapped = false` at the start of *every* pass and test it at the end of that pass.

6. Losing stability with one character. Bubble sort with `>=`, insertion sort with `>=` in the shifting test, or merge with `<` instead of `<=` all swap equal keys and make a stable algorithm unstable. *Fix:* shift or swap only on *strict* inequality, and take from the left run on ties.

7. Allocating the merge buffer inside every merge. A `new` or `vector` inside `merge` allocates and frees memory $n - 1$ times, which can dominate the running time. *Fix:* allocate one buffer of size n once and pass it down, as **Pseudocode 3.6** does.

8. Computing the midpoint as `(lo + hi) / 2`. For very large arrays the sum overflows `int`. *Fix:* `lo + (hi - lo) / 2`.

9. Mixing the two partition schemes. Lomuto's partition returns the pivot's position, and the recursive calls exclude it: `(lo, p - 1)` and `(p + 1, hi)`. Hoare's returns a split point, and the

calls are `(lo, j)` and `(j + 1, hi)`. Using Lomuto's calls with Hoare's partition loses elements or recurses forever. *Fix:* write the partition and its two recursive calls together, as one unit.

10. Using the first or last element as the pivot on data you do not control. Sorted input, the most common input of all, becomes the worst case, $\Theta(n^2)$ time and n levels of recursion. *Fix:* median of three or a random pivot; recurse into the smaller side; or call the library, which does both and more.

11. Believing quadratic sorts are fine "because computers are fast". At 10^5 elements our bubble sort took 28 seconds and `std::sort` 6 milliseconds. *Fix:* use **Table 3.2** before choosing, and measure.

12. Believing the fastest algorithm is the one with the fewest comparisons. Merge sort compares least, yet quicksort was faster, and heap sort, with the same asymptotic cost, was twice as slow. *Fix:* remember that memory traffic and cache behavior count as much as comparisons.

13. A comparator that is not a strict weak ordering. In C++, `[](int a, int b) { return a <= b; }` tells `std::sort` that an element comes before itself; the result is undefined behavior, sometimes a crash. *Fix:* comparators use `<` or `>`, never `<=` or `>=`. In Java a `compare` method must return a negative number, zero or a positive number consistently.

14. Counting in `int`. Bubble sort on 10^5 elements makes about 5×10^9 comparisons, more than a 32-bit `int` can hold. *Fix:* count with `long long` in C++ and `long` in Java, as every program in this chapter does.

15. Writing your own sort in production code. A hand-written sort is slower, less tested and usually unstable when stability was assumed. *Fix:* call `std::sort`/`std::stable_sort`, `Arrays.sort`/`List.sort`, or `sorted`/`list.sort`, and put your effort into the comparator or key.

3.17 Chapter Summary

1. **Sorting** rearranges items so their keys are in order (Equation (3.1)). It turns searching, duplicate detection, grouping, merging and ranking into cheap operations, which is why it is among the most studied problems in computing.
2. **Every sort rests on exchanging values safely**: three assignments and a temporary. C++ can write a swap function with pointers or references; Java can only exchange array slots; Python uses tuple assignment.
3. **Comparisons, swaps and moves** measure a sort independently of the machine. The number of **inversions** (Equation (3.2)) measures how unsorted an array is; any algorithm that swaps only neighbors makes exactly one swap per inversion (Equation (3.4)), $n(n-1)/4$ on average (Equation (3.5)), and is therefore quadratic.
4. **Bubble sort** carries the largest element to the end in each pass; with an **early exit** it is adaptive, $n-1$ comparisons on sorted input and $n(n-1)/2$ in the worst case (Equation (3.3)). **Exchange sort** compares one position with all later ones and is quadratic always. Both are in-place; bubble sort is stable, exchange sort is not.
5. **Selection sort** searches for the minimum and makes one exchange per round: always $n(n-1)/2$ comparisons, at most $n-1$ swaps, not stable, not adaptive.

6. **Insertion sort** grows a sorted prefix by shifting; its cost is proportional to n plus the number of inversions (Equation (3.6)). It is stable, in-place, adaptive and fast on small or nearly sorted arrays, which is why library sorts use it to finish.
7. **Shell sort** runs insertion sort over decreasing gaps. Long early jumps remove many inversions at once; with Knuth's gaps (Equation (3.7)) the worst case is $O(n^{3/2})$, and Ciura's gaps are best in practice. It is not stable.
8. **Merge sort** divides in the middle and merges; Equation (3.8) solves to $\Theta(n\lg n)$ in every case (Equation (3.9)). It is stable but needs n extra elements of memory.
9. **Quicksort** partitions around a pivot (Lomuto's single scan or Hoare's two converging indices) and recurses without merging. It is $\Theta(n\lg n)$ on average with about $1.39\,n\lg n$ comparisons (Equation (3.11)) but $\Theta(n^2)$ in the worst case (Equation (3.10)), which a fixed pivot hits on sorted input. Median-of-three or random pivots, and recursing into the smaller side, make it robust. It is the fastest comparison sort in practice.
10. **Heap sort** stores a max-heap inside the array (Equation (3.12)) and repeatedly moves the root to the end: $O(n\lg n)$ in the worst case and in-place, but not stable and unfriendly to caches. Chapter 14 builds the heap in full.
11. **No comparison sort beats** $\Omega(n\lg n)$ in the worst case (Equation (3.13)). **Counting sort** (Equation (3.14)) and **radix sort** (Equation (3.15)) avoid comparisons by using key values as indices, and are linear for small integer keys; both are stable, which radix sort depends on.
12. **Stable** sorts keep equal keys in input order; that makes multi-key sorting possible by sorting the least important key first. **In-place** sorts need only constant extra memory. **Adaptive** sorts do less work on nearly sorted input. **Table 3.2** lists all three for every algorithm.
13. **Measurements** confirm the formulas: quadratic sorts grow a hundredfold (or more, because of the cache) for each tenfold growth of n, $n\lg n$ sorts about twelvefold; at a million elements the gap is a factor of about 25,000.
14. **Library sorts are hybrids**: `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, and Python's `list.sort` is Timsort with the Powersort merge policy since 3.11. Use them.

3.17.1 Looking ahead

Chapter 4 works a set of evaluation problems drawn from Chapters 1 to 3, several of which ask you to sort strings and records with your own comparator. Chapter 5 begins the linked structures with stacks and queues. Recursion, which merge sort and quicksort used here without ceremony, gets a chapter of its own in Chapter 10, where the recursion trees of this chapter are drawn and solved in general. The heap returns in Chapter 14 as a priority queue, and hashing in Chapter 15 answers the question that sorting leaves open: how to find an item in $O(1)$ expected time without keeping anything in order at all.

3.18 Exercises

3.1 Trace bubble sort with early exit on [5,1,4,2,8]. Show the array after each pass, and count the comparisons and swaps. How many inversions does the input have?

3.2 Run selection sort by hand on the reversed array [6,5,4,3,2,1]. How many comparisons and how many swaps does it make? Compare with bubble sort on the same input.

3.3 Run insertion sort on [3,1,2,5,4] and count comparisons and shifts. Check your counts against Equation (3.6).

3.4 *(Programming.)* Bubble sort moves a small element near the end of the array only one place left per pass. **Cocktail shaker sort** alternates a left-to-right pass, which carries the largest element right, with a right-to-left pass, which carries the smallest element left. Implement it in C++, Java and Python, run it on [2,3,8,10,15,22,1], and compare its comparisons with those of bubble sort with early exit on the same input.

3.5 Prove that bubble sort as written in **Pseudocode 3.1** is stable. Then show, with a two-element example, that changing `>` to `>=` in line 4 makes it unstable.

3.6 Run exchange sort on [4,3,2,1]. How many swaps does it make? How many would selection sort make? Explain the difference.

3.7 Which gaps does the Knuth sequence Equation (3.7) produce, as used in **Listing 3.16**, for $n = 100$ and for $n = 1000$?

3.8 Merge the sorted runs [1,4,7,9] and [2,3,8,10,12] as **Pseudocode 3.6** does. How many comparisons are made? What is the largest number of comparisons that merging two runs of lengths 4 and 5 can ever need, and what input needs it?

3.9 Apply Lomuto's partition (**Pseudocode 3.8**) to [7,2,9,4,3,8,5]. Show the array and the value of i after each value of j, and give the index returned.

3.10 What happens to quicksort with Lomuto's partition when all n keys are equal? What happens with Hoare's partition? Give the number of comparisons in each case, as a function of n.

3.11 Build a max-heap from [4,1,3,2,16,9,10] with the build phase of **Pseudocode 3.10**. Show the array after each call of `SIFT-DOWN`.

3.12 Sort [3,0,2,3,1,0,2] with counting sort, $k = 4$. Show `count` after the counting loop and after the running totals, and the output.

3.13 Sort [170,45,75,90,802,24,2,66] with LSD radix sort in base 10. Show the array after each pass.

3.14 *(Programming.)* Modify merge sort so that it also returns the number of inversions of the array, in $O(n \lg n)$ time. Test it against an $O(n^2)$ count on A, on a sorted and a reversed array, and on [5,1,4,2,3], in all three languages.

3.15 Using Equation (3.13), find a lower bound on the number of comparisons any comparison sort needs, in the worst case, to sort 5 elements, and for $n = 1000$.

3.16 Show with a three-element example that quicksort with Lomuto's partition is not stable.

3.17 Profitina Laundry wants the day's orders listed by customer name and, for the same customer, heaviest first. Using only stable sorts, which key must be sorted first? Explain why the other order fails.

3.18 For each situation, choose a sorting method from this chapter or a library call, and justify the choice in one or two sentences: (a) 20 numbers on a small microcontroller with very little memory; (b) a log file of a million lines that is sorted by time except for a few hundred late entries; (c) 10^8 32-bit sensor readings; (d) customer records that must keep their previous order

among equal keys; (e) a real-time system in which the worst case must be $O(n\lg n)$ and no extra memory is available.

3.19 Find three sorting methods not covered in this chapter. For each, describe the idea, write it in pseudocode, and give its best, average and worst running times and whether it is stable and in-place.

3.19 Solutions to the Exercises

3.1 Pass 1: $5 > 1$ swap, $5 > 4$ swap, $5 > 2$ swap, $5 < 8$: [1,4,2,5,8], 4 comparisons, 3 swaps. Pass 2: $1 < 4$, $4 > 2$ swap, $4 < 5$: [1,2,4,5,8], 3 comparisons, 1 swap. Pass 3: $1 < 2$, $2 < 4$: no swap, so the early exit stops the sort after 2 comparisons. Total: 9 comparisons and 4 swaps. The inversions are (5,1), (5,4), (5,2) and (4,2): four, equal to the swaps, as Equation (3.4) requires.

3.2 Round 1 finds 1 at index 5 and exchanges it with 6: [1,5,4,3,2,6]. Round 2 finds 2 at index 4: [1,2,4,3,5,6]. Round 3 finds 3 at index 3: [1,2,3,4,5,6]. Rounds 4 and 5 find the minimum in place. Comparisons: $5 + 4 + 3 + 2 + 1 = 15$; swaps: 3. Bubble sort on the same input makes 15 comparisons and 15 swaps, one per inversion. On reversed input selection sort's advantage in swaps is at its largest.

3.3 Key 1: $3 > 1$ shift, front reached: 1 comparison, 1 shift, [1,3,2,5,4]. Key 2: $3 > 2$ shift, $1 \leq 2$ stop: 2 comparisons, 1 shift. Key 5: $3 \leq 5$ stop: 1 comparison. Key 4: $5 > 4$ shift, $3 \leq 4$ stop: 2 comparisons, 1 shift. Total 6 comparisons and 3 shifts. The inversions are (3,1), (3,2), (5,4), so $\text{inv} = 3$, and Equation (3.6) gives $4 \leq 6 \leq 3 + 4 = 7$; the exact count is $\text{inv} + (n - 1) - 1 = 6$, because one key (1) went to the front.

3.4 **Listing 3.43** is the C++ solution; each round is a left-to-right pass followed by a right-to-left pass, and the range shrinks from both ends.

Listing 3.43 `cocktail_sort.cpp`: cocktail shaker sort, the solution of Exercise 3.4.

```cpp
// cocktail_sort.cpp - bubble sort in both directions (cocktail shaker sort)
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

string show(const vector<int>& a) {
    string s = "[";
    for (size_t i = 0; i < a.size(); ++i) s += (i ? " " : "") + to_string(a[i]);
    return s + "]";
}

void cocktailSort(vector<int>& a) {
    int lo = 0, hi = static_cast<int>(a.size()) - 1;
    long long comparisons = 0;
    bool swapped = true;
    while (swapped && lo < hi) {
        swapped = false;
        for (int j = lo; j < hi; ++j) {  // left to right: largest to a[hi]
            ++comparisons;
            if (a[j] > a[j + 1]) { swap(a[j], a[j + 1]); swapped = true; }
        }
        --hi;
        for (int j = hi; j > lo; --j) {  // right to left: smallest to a[lo]
            ++comparisons;
            if (a[j - 1] > a[j]) { swap(a[j - 1], a[j]); swapped = true; }
        }
        ++lo;
        cout << "  round: " << show(a) << "\n";
    }
    cout << "  comparisons = " << comparisons << "\n";
}

int main() {
    // one small value stuck at the end
    vector<int> turtle = {2, 3, 8, 10, 15, 22, 1};
    cout << "Cocktail sort on " << show(turtle) << "\n";
    cocktailSort(turtle);
    return 0;
}
```

Listing 3.44 `CocktailSort.java`: cocktail shaker sort in Java.

```java
// CocktailSort.java - bubble sort in both directions (cocktail shaker sort)
public class CocktailSort {
    static String show(int[] a) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < a.length; i++)
            sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.append("]").toString();
    }

    static void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    static void cocktailSort(int[] a) {
        int lo = 0, hi = a.length - 1;
        long comparisons = 0;
        boolean swapped = true;
        while (swapped && lo < hi) {
            swapped = false;
            for (int j = lo; j < hi; j++) {  // left to right: largest to a[hi]
                comparisons++;
                if (a[j] > a[j + 1]) { swap(a, j, j + 1); swapped = true; }
            }
            hi--;
            for (int j = hi; j > lo; j--) {  // right to left: smallest to a[lo]
                comparisons++;
                if (a[j - 1] > a[j]) { swap(a, j - 1, j); swapped = true; }
            }
            lo++;
            System.out.println("  round: " + show(a));
        }
        System.out.println("  comparisons = " + comparisons);
    }

    public static void main(String[] args) {
        // one small value stuck at the end
        int[] turtle = {2, 3, 8, 10, 15, 22, 1};
        System.out.println("Cocktail sort on " + show(turtle));
        cocktailSort(turtle);
    }
}
```

Listing 3.45 `cocktail_sort.py`: cocktail shaker sort in Python.

```python
# cocktail_sort.py - bubble sort in both directions (cocktail shaker sort)
def show(a):
    return "[" + " ".join(map(str, a)) + "]"


def cocktail_sort(a):
    lo, hi = 0, len(a) - 1
    comparisons = 0
    swapped = True
    while swapped and lo < hi:
        swapped = False
        for j in range(lo, hi):          # left to right: largest to a[hi]
            comparisons += 1
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                swapped = True
        hi -= 1
        for j in range(hi, lo, -1):      # right to left: smallest to a[lo]
            comparisons += 1
            if a[j - 1] > a[j]:
                a[j - 1], a[j] = a[j], a[j - 1]
                swapped = True
        lo += 1
        print(f"  round: {show(a)}")
    print(f"  comparisons = {comparisons}")


turtle = [2, 3, 8, 10, 15, 22, 1]        # one small value stuck at the end
print(f"Cocktail sort on {show(turtle)}")
cocktail_sort(turtle)
```

Listing 3.44 and **Listing 3.45** print the same as the C++ version:

```
Cocktail sort on [2 3 8 10 15 22 1]
  round: [1 2 3 8 10 15 22]
  round: [1 2 3 8 10 15 22]
  comparisons = 18
```

The first round already sorts the array: the right-to-left pass carries 1 from the end to the front in one sweep. The second round finds nothing to swap and stops, for 18 comparisons in all. Bubble sort needs six passes, because 1 moves one place per pass, and makes $6 + 5 + 4 + 3 + 2 + 1 = 21$ comparisons. Both make the same 6 swaps, one per inversion: cocktail sort is still an adjacent-swap sort and still quadratic in the worst and average case.

3.5 Bubble sort changes the relative order of two elements only when it swaps them, and it swaps only neighbors with `A[j] > A[j+1]`. Two equal keys are never in that relation, so they are never swapped with each other. Could they pass each other some other way? Two elements can only exchange relative order by being swapped with each other, since every swap involves exactly two neighbors. So equal keys keep their relative order: the sort is stable. With `>=`, the input $[2_a, 2_b]$ (two equal keys, with subscripts to tell them apart) is swapped to $[2_b, 2_a]$, and the next pass swaps them back only if another pass runs; with the early exit and `>=` the flag never becomes false, and the resulting order of equal keys depends on how many passes run. The sort is no longer stable.

3.6 Round 1: $3 < 4$ swap, [3,4,2,1]; $2 < 3$ swap, [2,4,3,1]; $1 < 2$ swap, [1,4,3,2]. Round 2: $3 < 4$ swap, [1,3,4,2]; $2 < 3$ swap, [1,2,4,3]. Round 3: $3 < 4$ swap. Six swaps, one for every comparison. Selection sort makes two: $4 \leftrightarrow 1$ and $3 \leftrightarrow 2$. Exchange sort pays a swap each time a better candidate for position i appears; selection sort only remembers its index and swaps once.

3.7 The loop multiplies by three and adds one while $h < n/3$. For $n = 100$, $n/3 = 33$: h goes 1,4,13,40, and the gaps are 40,13,4,1. For $n = 1000$, $n/3 = 333$: h goes 1,4,13,40,121,364, and the gaps are 364,121,40,13,4,1.

3.8 $1 < 2$ take 1; $4 > 2$ take 2; $4 > 3$ take 3; $4 < 8$ take 4; $7 < 8$ take 7; $9 > 8$ take 8; $9 < 10$ take 9. The left run is empty after 7 comparisons, and 10,12 are copied. At most $4 + 5 - 1 = 8$ comparisons are ever needed: each comparison outputs one element, and the last element is always output without one. The worst case interleaves the runs so that the two largest elements are in different runs, for example [1,3,5,9] and [2,4,6,8,10]: then 9 is compared with 6, 8 and 10, and 8 comparisons are made before one run is empty.

3.9 The pivot is $x = 5$ and $i = -1$. $j = 0$: $7 > 5$, $i = -1$. $j = 1$: $2 \leq 5$, $i = 0$, exchange $A[0], A[1]$: [2,7,9,4,3,8,5]. $j = 2$: $9 > 5$. $j = 3$: $4 \leq 5$, $i = 1$: [2,4,9,7,3,8,5]. $j = 4$: $3 \leq 5$, $i = 2$: [2,4,3,7,9,8,5]. $j = 5$: $8 > 5$. Finally exchange $A[3]$ with the pivot: [2,4,3,5,9,8,7], and the partition returns 3 after 6 comparisons.

3.10 With Lomuto's partition every `A[j] ≤ x` is true, so i advances with j, the pivot lands at the right end, and the left part has $m - 1$ elements: the worst case of Equation (3.10), $n(n-1)/2$ comparisons and n levels of recursion. With Hoare's partition both indices stop at *every* element (each is $\geq x$ and $\leq x$), every stop produces a swap, and the indices meet in the middle, so each partition splits evenly and the total is $\Theta(n \lg n)$, about $n \lg n$ comparisons plus the stops. This is one reason libraries descend from Hoare's scheme; another fix is three-way partitioning into $<$, $=$ and $>$ regions.

3.11 $n = 7$, so the build calls `SIFT-DOWN` for $i = 2,1,0$. $i = 2$: 3 has children 9 and 10; exchange with 10: [4,1,10,2,16,9,3]. $i = 1$: 1 has children 2 and 16; exchange with 16: [4,16,10,2,1,9,3]; index 4 has no children. $i = 0$: 4 has children 16 and 10; exchange with 16: [16,4,10,2,1,9,3]; at index 1, 4 is larger than its children 2 and 1, and the heap is built.

3.12 Counts: `count = [2, 1, 2, 2]` (two 0s, one 1, two 2s, two 3s). Running totals: `[2, 3, 5, 7]`. Placing from right to left: the last 2 goes to index 4, the last 0 to index 1, the 1 to index 2, the last 3 to index 6, the first 2 to index 3, the first 0 to index 0, the first 3 to index 5. Output: [0,0,1,2,2,3,3].

3.13 By the ones digit: [170,90,802,2,24,45,75,66]. By the tens digit: [802,2,24,45,66,170,75,90] (note that 802 and 2, both with tens digit 0, keep the order of the first pass). By the hundreds digit: [2,24,45,66,75,90,170,802].

3.14 When `MERGE` takes an element from the right run while $mid - i$ elements remain in the left run, that element was smaller than every one of them and came after all of them: exactly $mid - i$ inversions, all removed at once. Adding those amounts during the merges, plus the inversions inside each half (counted recursively), gives the total. The extra work is one addition per step of the merge, so the time stays $\Theta(n \lg n)$.

Listing 3.46 `count_inversions.cpp`: inversions counted by merge sort, checked against the quadratic count from the definition. Solution of Exercise 3.14.

```cpp
// count_inversions.cpp - counting inversions with merge sort in O(n log n)
#include <iostream>
#include <vector>
using namespace std;

// Sorts a[lo..hi-1] and returns how many pairs i < j in it had a[i] > a[j].
long long sortCount(vector<int>& a, vector<int>& t, int lo, int hi) {
    if (hi - lo < 2) return 0;
    int mid = lo + (hi - lo) / 2;
    long long inv = sortCount(a, t, lo, mid) + sortCount(a, t, mid, hi);
    int i = lo, j = mid, k = lo;
    while (i < mid && j < hi) {
        if (a[i] <= a[j]) t[k++] = a[i++];
        else {
            inv += mid - i;  // a[j] jumps over every a[i..mid-1]
            t[k++] = a[j++];
        }
    }
    while (i < mid) t[k++] = a[i++];
    while (j < hi) t[k++] = a[j++];
    for (k = lo; k < hi; ++k) a[k] = t[k];
    return inv;
}

long long bruteForce(const vector<int>& a) {          // O(n^2) check
    long long inv = 0;
    for (size_t i = 0; i < a.size(); ++i)
        for (size_t j = i + 1; j < a.size(); ++j)
            if (a[i] > a[j]) ++inv;
    return inv;
}

int main() {
    vector<vector<int>> tests = {{22, 10, 15, 3, 8, 2}, {2, 3, 8, 10, 15, 22},
                                 {22, 15, 10, 8, 3, 2}, {5, 1, 4, 2, 3}};
    for (vector<int> a : tests) {
        long long slow = bruteForce(a);
        vector<int> t(a.size());
        long long fast = sortCount(a, t, 0, static_cast<int>(a.size()));
        cout << "inversions: merge sort " << fast << ", brute force " << slow
             << "\n";
    }
    return 0;
}
```

Listing 3.47 `CountInversions.java`: the inversion count in Java.

```java
// CountInversions.java - counting inversions with merge sort in O(n log n)
public class CountInversions {
    // Sorts a[lo..hi-1] and returns how many pairs i < j in it had a[i] > a[j].
    static long sortCount(int[] a, int[] t, int lo, int hi) {
        if (hi - lo < 2) return 0;
        int mid = lo + (hi - lo) / 2;
        long inv = sortCount(a, t, lo, mid) + sortCount(a, t, mid, hi);
        int i = lo, j = mid, k = lo;
        while (i < mid && j < hi) {
            if (a[i] <= a[j]) t[k++] = a[i++];
            else {
                inv += mid - i;  // a[j] jumps over every a[i..mid-1]
                t[k++] = a[j++];
            }
        }
        while (i < mid) t[k++] = a[i++];
        while (j < hi) t[k++] = a[j++];
        for (k = lo; k < hi; k++) a[k] = t[k];
        return inv;
    }

    static long bruteForce(int[] a) {                // O(n^2) check
        long inv = 0;
        for (int i = 0; i < a.length; i++)
            for (int j = i + 1; j < a.length; j++)
                if (a[i] > a[j]) inv++;
        return inv;
    }

    public static void main(String[] args) {
        int[][] tests = {{22, 10, 15, 3, 8, 2}, {2, 3, 8, 10, 15, 22},
                         {22, 15, 10, 8, 3, 2}, {5, 1, 4, 2, 3}};
        for (int[] a : tests) {
            long slow = bruteForce(a);
            long fast = sortCount(a, new int[a.length], 0, a.length);
            System.out.println("inversions: merge sort " + fast
                    + ", brute force " + slow);
        }
    }
}
```

Listing 3.48 `count_inversions.py`: the inversion count in Python.

```
# count_inversions.py - counting inversions with merge sort in O(n log n)
def sort_count(a, t, lo, hi):
    """Sorts a[lo..hi-1]; returns how many pairs i < j in it had a[i] > a[j]."""
    if hi - lo < 2:
        return 0
    mid = lo + (hi - lo) // 2
    inv = sort_count(a, t, lo, mid) + sort_count(a, t, mid, hi)
    i, j, k = lo, mid, lo
    while i < mid and j < hi:
        if a[i] <= a[j]:
            t[k] = a[i]
            i += 1
        else:
            inv += mid - i              # a[j] jumps over every a[i..mid-1]
            t[k] = a[j]
            j += 1
        k += 1
    t[k:hi] = a[i:mid] + a[j:hi]        # whichever run is left over
    a[lo:hi] = t[lo:hi]
    return inv


def brute_force(a):                     # O(n^2) check
    n = len(a)
    return sum(1 for i in range(n) for j in range(i + 1, n) if a[i] > a[j])


TESTS = ([22, 10, 15, 3, 8, 2], [2, 3, 8, 10, 15, 22],
         [22, 15, 10, 8, 3, 2], [5, 1, 4, 2, 3])
for a in TESTS:
    slow = brute_force(a)
    fast = sort_count(a, [0] * len(a), 0, len(a))
    print(f"inversions: merge sort {fast}, brute force {slow}")
```

Listing 3.46, **Listing 3.47** and **Listing 3.48** print:

```
inversions: merge sort 13, brute force 13
inversions: merge sort 0, brute force 0
inversions: merge sort 15, brute force 15
inversions: merge sort 6, brute force 6
```

The 13 inversions of A are the ones counted by hand in Section 3.2.5; the reversed array has all $6 \cdot 5/2 = 15$.

3.15 For $n = 5$: $5! = 120$ and $\lg 120 \approx 6.91$, so at least 7 comparisons in the worst case (and 7 is achievable, by a clever method in [1]). For $n = 1000$: $\lg(1000!) \approx 8530$, so at least 8530 comparisons; the cruder bound in Equation (3.13) gives $500 \lg 500 \approx 4483$. Merge sort's worst case, $n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1 = 10\,000 - 1024 + 1 = 8977$, is within 6% of the true minimum.

3.16 Take $[2_a, 2_b, 1]$. The pivot is 1; $2_a > 1$ and $2_b > 1$, so i stays at -1, and the pivot is exchanged with $A[0] = 2_a$: $[1, 2_b, 2_a]$. The two equal keys have changed order, and the sub-array $[2_b, 2_a]$ is already "sorted", so the sorted array keeps them reversed.

3.17 Sort by weight (heaviest first) first, then stably by customer name. The last sort decides the main order (by name), and stability keeps, among orders of the same customer, the order produced by the first sort (by weight). Sorting by name first and weight last would produce a list ordered by weight, with names only as a tie-breaker.

3.18 (a) Insertion sort: tiny code, no extra memory, and fastest of all for 20 elements. (b) Insertion sort or a library Timsort: the input has few inversions, so an adaptive sort is almost linear; a plain merge sort or heap sort would ignore the existing order. (c) Radix sort with base 256, four passes: linear time, as **Listing 3.34** showed, at the cost of a second array of 10^8

elements. (d) A stable sort: merge sort, `std::stable_sort`, `List.sort` or `sorted`. (e) Heap sort: guaranteed $O(n \lg n)$ and in-place; introsort would also do, because it falls back to heap sort.

3.19 Many answers are possible; three are sketched here. **Comb sort** is bubble sort with a gap: compare elements `gap` apart, shrink the gap by a factor of about 1.3 each pass, and finish with ordinary bubble sort at gap 1; like Shell sort it removes far-away inversions early; worst case $O(n^2)$, typically close to $n \lg n$ in practice; in-place, not stable. **Gnome sort** keeps one position `p`: if `A[p-1] ≤ A[p]`, step right, otherwise swap them and step left; it is insertion sort done by adjacent swaps, $O(n)$ best, $O(n^2)$ average and worst, in-place and stable. **Bucket sort** for keys spread evenly over [0,1) puts each key into one of n buckets by `⌊n·key⌋`, sorts each bucket with insertion sort, and concatenates the buckets; expected $O(n)$ for uniform keys, $O(n^2)$ worst case if all keys fall into one bucket, $O(n)$ extra space, stable if the bucket sort is stable. Tree sort, mentioned in Section 3.2.3, inserts every key into a binary search tree and reads it back in order; it appears in Chapter 11.

References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[2] O. Astrachan, “Bubble sort: An archaeological algorithmic analysis,” in *Proc. 34th SIGCSE Technical Symposium on Computer Science Education*, 2003, pp. 1–5.

[3] D. L. Shell, “A high-speed sorting procedure,” *Communications of the ACM*, vol. 2, no. 7, pp. 30–32, 1959.

[4] M. Ciura, “Best increments for the average case of Shellsort,” in *Fundamentals of Computation Theory (FCT 2001)*, LNCS 2138. Berlin, Germany: Springer, 2001, pp. 106–117.

[5] C. A. R. Hoare, “Quicksort,” *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 2, 6, 7, 8.

[7] J. W. J. Williams, “Algorithm 232: Heapsort,” *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[8] R. W. Floyd, “Algorithm 245: Treesort 3,” *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.

[9] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011, ch. 2.

[10] cppreference.com, “std::sort.” [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/sort (accessed Sep. 2026).

[11] D. R. Musser, “Introspective sorting and selection algorithms,” *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, 1997.

[12] Free Software Foundation, GNU C++ Library (libstdc++), GCC 13, source file `bits/stl_algo.h` (`__introsort_loop`, `__final_insertion_sort`, `_S_threshold = 16`, `_S_chunk_size = 7`).

[13] cppreference.com, “std::stable_sort.” [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/stable_sort (accessed Sep. 2026).

[14] Oracle, “Class Arrays,” *Java SE 25 & JDK 25 API Specification*, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html (accessed Sep. 2026).

[15] V. Yaroslavskiy, J. Bentley, J. Bloch, and D. Lea, `DualPivotQuicksort.java`, OpenJDK. [Online]. Available: https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/DualPivotQuicksort.java (accessed Sep. 2026).

[16] OpenJDK, `TimSort.java`, and issue JDK-6804124, "Replace 'modified mergesort' in java.util.Arrays.sort with timsort." [Online]. Available: https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/TimSort.java and https://bugs.openjdk.org/browse/JDK-6804124 (accessed Sep. 2026).

[17] Python Software Foundation, "Sorting Techniques," *Python 3.14 Documentation, HOWTOs*. [Online]. Available: https://docs.python.org/3/howto/sorting.html (accessed Sep. 2026).

[18] T. Peters *et al.*, `Objects/listsort.txt`, CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Objects/listsort.txt (accessed Sep. 2026).

[19] J. I. Munro and S. Wild, "Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs," in *Proc. 26th European Symposium on Algorithms (ESA 2018)*, LIPIcs vol. 112, 2018, pp. 63:1–63:16.

[20] T. Peters, "bpo-34561: Switch to Munro & Wild 'powersort' merge strategy," CPython pull request #28108, Sep. 2021 (released in Python 3.11); and S. Wild, "Powersort in official Python 3.11 release." [Online]. Available: https://github.com/python/cpython/pull/28108 and https://www.wild-inter.net/posts/powersort-in-python-3.11 (accessed Sep. 2026).

[21] The PostgreSQL Global Development Group, "Using EXPLAIN," *PostgreSQL 18 Documentation*, sec. 14.1, and source file `src/backend/utils/sort/tuplesort.c`. [Online]. Available: https://www.postgresql.org/docs/current/using-explain.html (accessed Sep. 2026).

Chapter 4

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

This chapter adds no new data structure. Instead, it takes everything Chapters 1 to 3 have given you (arrays in memory, the four ways of searching, and the family of sorting algorithms) and puts it to work on eleven complete problems. Four of them are the classic first problems of a data structures course: sorting words into dictionary order, finding the first character of a string that repeats, looking up how much money a person owns, and sorting messages by how important they are. Seven more are new: computing the address of an array element by hand, counting how often a value occurs in a sorted array, interpolation search on a library's book codes, tracing the three elementary sorts pass by pass, merging two sorted lists, deleting values from an array in place, and finding two laundry services whose prices add up to a voucher. Every problem is solved from nothing, slowly, and always in the same order: the problem restated in plain words; examples worked by hand; a design drawn and traced on paper; a complete C++ program; the same program in Java and in Python, with the real output of all three; a *tempting wrong solution* together with the exact way it fails; and a *check yourself* variant with its answer. By the end of the chapter you will own a method for attacking any small data-structure problem, and eleven fully worked examples of that method to lean on.

4.1 Where We Left Off

Before we solve anything, let us lay out the toolkit. Every problem in this chapter reaches for a handful of tools from the first three chapters, and naming the tool you need is already half of the work.

From **Chapter 1** you have the workbench and the memory model. You can compile a C++ program with `g++`, run a Java class with `java` and a Python script with `python3`, and read what they print. More importantly, you know what an *array* is: a row of equally sized boxes that sit next to each other in memory, numbered from 0, so that the computer can jump to box number i in one step by computing its address. You also met two-dimensional arrays, which C++ stores row after row.

From **Chapter 2** you have *searching*. *Sequential search* (also called linear search) looks at the items one by one from the front and needs no order at all; its *sentinel* variant places the key at the end so that the loop needs only one test. *Binary search* needs sorted data and halves the part still to be searched at every step. *Interpolation search* also needs sorted data, and guesses the position of the key from its value, the way you open a telephone directory near the back when you look for a name that starts with W. Chapter 2 also introduced the language for comparing algorithms: the number of steps as a function of the input size n, written in Big-O notation such as $O(n)$ and $O(\log n)$.

From **Chapter 3** you have *sorting*. Bubble sort and exchange sort compare neighbors or pairs and swap them; selection sort repeatedly picks the smallest remaining item; insertion sort grows a sorted part on the left one item at a time; merge sort and quicksort split the work in halves; and you learned the word *stable*, which describes a sort that keeps items with equal keys in their original order. Chapter 3 ended with the observation that every comparison sort needs a rule that says which of two items comes first, and that changing only this rule changes what "sorted" means [1], [2]. Problems 4.1 and 4.4 grow directly out of that observation.

Table 4.1 shows which tools each problem of this chapter exercises most. Use it as a map: if you feel unsure about binary search, start with Problems 4.3 and 4.6; if sorting still feels mechanical rather than understood, start with Problem 4.8.

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

4.2 How to Approach a Data-Structure Problem

A beginner who is handed a problem usually opens the editor and starts typing `#include <iostream>`. An experienced programmer usually does not touch the keyboard for the first few minutes. The difference is not talent; it is a routine. The mathematician George Pólya described such a routine for mathematical problems (understand the problem, devise a plan, carry it out, look back) [3], and the same four movements, stretched into eight small steps, serve programmers just as well. **Figure 4.1** shows the eight steps we follow for every problem in this chapter.

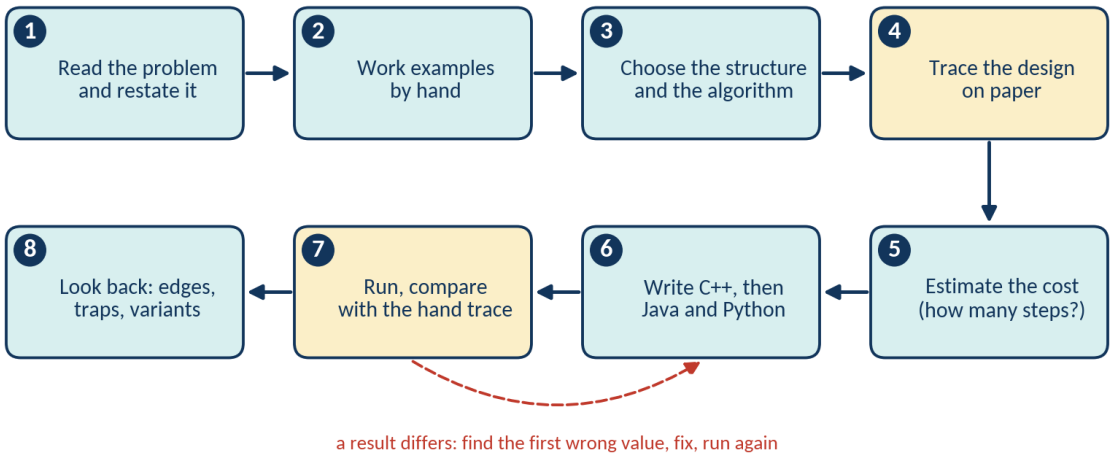


Figure 4.1 The eight steps used for every problem in this chapter; the dashed arrow is the loop between running the program and fixing it.

4.2.1 The eight steps

Step 1: Read the problem and restate it. Read the statement twice. Then write, in one or two sentences of your own, what goes *in* and what must come *out*. If you cannot do this, you do not yet understand the problem, and no amount of code will help. Underline every example in the statement; examples are the first tests of your program.

Step 2: Work examples by hand. Take the examples of the statement and compute the answers with pencil and paper *before* writing any code, writing down every intermediate value. This is where you discover whether you really understand the problem. Add one example of your own at an edge: an empty input, a single item, a key that is missing, a key at the very first or very last position, two items that tie.

Step 3: Choose the structure and the algorithm. Ask two questions. *Where will the data live?* In this chapter the answer is almost always an array (or its Java and Python cousins), but *how* it is arranged matters: sorted or unsorted, one row or a grid, one array of records or two parallel arrays. *Which known algorithm does the job, or nearly does it?* Most problems are a known algorithm in disguise. Finding the disguise is the heart of the step: “first repeated character” is counting; “how many times does 2001 occur” is binary search for two boundaries; “two prices that add up to a voucher” is sorting followed by a walk from both ends.

Step 4: Trace the design on paper. Draw the array as a row of boxes with their indices underneath, and move the algorithm through it by hand, one line per step, writing down the value of every index variable. Every problem in this chapter has such a picture. A trace on paper finds most off-by-one errors before they exist, and it gives you, line by line, the values that the program must reproduce in step 7.

Step 5: Estimate the cost. Count, roughly, how many basic steps the design takes as a function of the input size n. **Table 4.2** turns such a count into a feeling for time. A plan that needs n^2 steps is perfectly fine for a hundred words and hopeless for ten million prices; a plan that needs $n\log_2 n$ steps is fine for both. When a statement says “the optimal solution is $O(n)$”, it is telling you which row of the table to aim for.

Step 6: Write the code, C++ first. Write the program in small pieces: the data first, compiled and printed; then one function, compiled and tested; then the next. A program that grows in small, tested steps never presents you with fifty errors at once. When the C++ version works, translate it

to Java and Python. The translation is a useful check in itself: a line you cannot translate is often a line you do not fully understand.

Step 7: Run, and compare with the hand trace. Run the program on the examples of step 2. When a result differs, do not guess. Print the variables at every step (or step through the program in a debugger) and compare them with your hand trace until you find the *first* value that differs. That difference is the bug. Fix it, and run all the examples again; this is the dashed arrow in **Figure 4.1**.

Step 8: Look back. When everything agrees, ask three questions. Which input did I not try? What is the most *tempting* wrong way to solve this problem, and why does it fail? Can I solve a slightly different version? These three questions are exactly the last three parts of every problem in this chapter.

> **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 of binary search through nine boxes on paper, you cannot check a program that claims to do it. Every expected output in this chapter was first computed by hand in the "Understanding it by hand" and "Designing it on paper" parts of its problem.

4.2.2 Estimating the cost before writing the code

Step 5 deserves a closer look, because it is the step beginners skip most often and the one that matters most once the data grows. A modern computer carries out somewhere around 10^8 to 10^9 simple steps (an addition, a comparison, one pass through a short loop body) per second. That is only an order of magnitude, but an order of magnitude is all we need. **Table 4.2** applies it to the growth rates of Chapters 2 and 3.

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}

Two counts come back again and again in this chapter, so we give them names. The first is the number of comparisons made by two nested loops in which the inner loop runs over the items to the right of the outer one, as in bubble sort, selection sort and the brute-force search for a pair. For n items the inner loop runs $n-1$ times, then $n-2$ times, and so on down to once, so the total is the sum in Equation (4.1).

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

Why is the sum equal to $\frac{n(n-1)}{2}$? Write the sum twice, once forward and once backward, and add the two rows column by column: $(n-1)+1$, $(n-2)+2$, and so on, each column adds up to n, and there are $n-1$ columns. The doubled sum is therefore $n(n-1)$, and the sum itself is half of that. For $n = 7$ it is $\frac{7 \cdot 6}{2} = 21$, a number we shall meet in Problem 4.8, and for $n = 8$ it is 28, which reappears in Problem 4.11.

The second count is the number of probes of binary search. Every probe at least halves the part of the array that can still contain the key, so after k probes at most $\frac{n}{2^k}$ items remain. The search must end when fewer than one item remains, so the number of probes is at most the number in Equation (4.2).

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

The symbols $\lfloor x \rfloor$ mean "x rounded down to a whole number", and $\log_2 n$ is the power to which 2 must be raised to give n; so $\log_2 8 = 3$ because $2^3 = 8$. For $n = 8$ book codes, binary search makes at most $\lfloor \log_2 8 \rfloor + 1 = 4$ probes, and for a million items at most 20. That is the whole reason sorted data is worth having.

4.2.3 How each problem is laid out, and where the code lives

Every problem section has the same seven parts, so that you always know where you are:

1. **The problem**: the statement, in a box.
2. **Understanding it by hand**: the statement restated in plain words and the examples computed on paper (steps 1 and 2).
3. **Designing it on paper**: the choice of structure and algorithm, a picture traced by hand, the pseudocode, and the cost (steps 3 to 5).
4. **The C++ solution**: a complete program, what it prints, and a walk through its important lines (steps 6 and 7).
5. **The Java and Python solutions**: the same design in the other two languages; all three programs print exactly the same output, byte for byte.
6. **A tempting wrong solution**: a program that looks right, its real output, and why it fails (step 8).
7. **Check yourself**: a variant of the problem, with its answer.

All programs are in the chapter's code folder: `Chapter04/cpp`, `Chapter04/java` and `Chapter04/python`, each with a `tempting` subfolder that holds the tempting wrong solutions, and a shared folder `Chapter04/input` with the input files. The C++ programs were compiled with `g++ -std=c++17 -O2 -Wall -Wextra` without a single warning and were also run under the AddressSanitizer and UndefinedBehaviorSanitizer (`-fsanitize=address,undefined`), which report any read or write outside an array. The Java programs were compiled with `javac` 21 and the Python programs run with Python 3.11; both also run unchanged on JDK 25 and Python 3.14. Programs that read input are run with the input file redirected, for example `./sort_words < ../input/words.txt` on Linux and macOS, or `Get-Content ..\input\words.txt | .\sort_words.exe` in Windows PowerShell; the README of the folder lists the command for every program.

> **ONE RULE FOR ALL ELEVEN PROBLEMS.** Every program in this chapter uses plain arrays and hand-written searches and sorts, even where the standard library offers a ready-made function. This is on purpose: the point is to see the data structure work. Each problem ends its C++ section by naming the library function that does the same job (`std::sort`, `std::lower_bound`, `std::merge`, and so on), so that you know what to reach for once you understand what it does.

4.3 Problem 4.1: Sorting Words Alphabetically

4.3.1 The problem

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

That is the whole statement: one sentence, no example, no input format. Short statements are the most dangerous kind, because every decision the statement does not make, you must make yourself, and write down. We make three decisions here. *Input:* the first number n says how many words follow, and then come n words separated by spaces or line breaks; a word contains no spaces. *Output:* the words once as they were read, then the sorted words, one per line. *Order:* "alphabetical" means the order of a dictionary, in which *apple* and *Apple* are the same word and both come before *banana*; when two words differ only in capital letters, we put the capitalized one first, so that the result is always the same.

4.3.2 Understanding it by hand

In plain words: we get a list of words and must print them in dictionary order. Our example is the input file `words.txt`:

```
8
mango Banana apple Cherry durian banana Apple grape
```

By hand, ignoring capitals, the dictionary order of the eight words is: apple, Apple, banana, Banana, Cherry, durian, grape, mango. Two pairs tie when capitals are ignored (*apple*/*Apple* and *banana*/*Banana*), and our third decision puts the capitalized word of each pair first. The expected answer is therefore

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

Why do we need a decision about capitals at all? Because a computer does not compare letters; it compares numbers. Every character is stored as a small whole number called its *character code*, and in the ASCII code that C++, Java and Python all agree on for these letters, the capital letters `A` to `Z` have the codes 65 to 90 and the small letters `a` to `z` the codes 97 to 122 [4]. When a program compares two strings with the ordinary `<`, it compares their first characters by code; if those are equal, the second characters; and so on. This is called *lexicographic* order. Since every capital letter has a smaller code than every small letter, plain `<` says that `"Cherry" < "apple"`, because 67 is less than 97. That is not dictionary order, and it is the trap of Section 4.3.6.

Our own edge examples: an empty list ($n = 0$) must print nothing after `Sorted:`; a list of one word prints that word; a list that is already sorted must stay as it is.

4.3.3 Designing it on paper

The structure is an array of strings, `words[0..n-1]`. **The algorithm** is any sort from Chapter 3, with one change: the test "is `a` before `b`?" is no longer the plain `<` but a small function of our own, `comesBefore(a, b)`. Its rule, written out:

1. Make copies of `a` and `b` in which every capital letter is turned into the small letter (in ASCII, add $97 - 65 = 32$ to its code).
2. If the two copies differ, `a` comes first exactly when its copy is smaller.
3. If the copies are equal, the words differ at most in capitals; then `a` comes first exactly when `a < b` in plain code order, which puts `Apple` before `apple`.

For the sort we choose *insertion sort*, for three reasons: it is short, it is stable (Problem 4.4 will need that), and it is fast on lists that are already nearly sorted. **Pseudocode 4.1** is the

pseudocode; the only thing that distinguishes it from the insertion sort of Chapter 3 is the name of the comparison in line 4.

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
```

Figure 4.2 traces the algorithm on our eight words. Row `i = 3`, for example, starts from `apple Banana mango Cherry …`: the key is `Cherry`; it comes before `mango` (because *cherry* < *mango*), so `mango` shifts right; it does not come before `Banana` (because *banana* < *cherry*), so the loop stops and `Cherry` lands at index 2. Follow at least rows `i = 5` and `i = 6` with a pencil: in row 6 the key `Apple` travels all the way to index 0, past `apple`, because the two words tie in small letters and `Apple` < `apple` in code order.

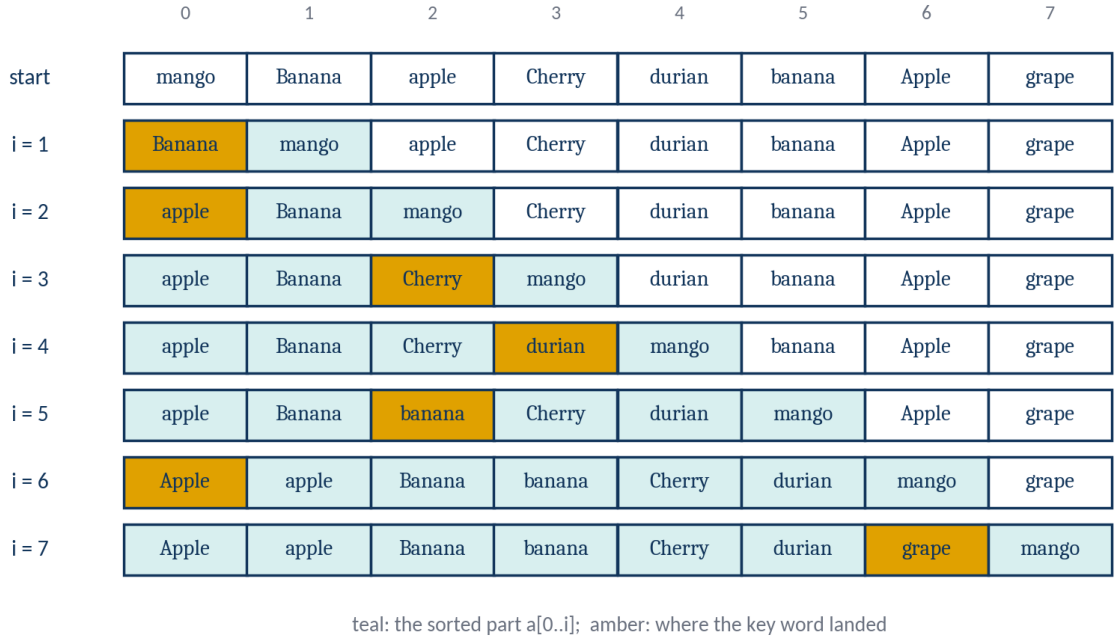


Figure 4.2 Insertion sort of the eight words: after step i the first $i + 1$ words (teal) are in dictionary order, and the amber cell is where the key word landed.

The cost. Insertion sort makes at most $\frac{n(n-1)}{2}$ comparisons (Equation (4.1)), each of which compares two words of at most L characters, so the work is $O(n^2L)$ in the worst case and $O(nL)$ on a list that is already sorted. For a few hundred words that is nothing. For a million words we would switch to an $O(n\log n)$ sort, and the comparison function would stay exactly the same.

4.3.4 The C++ solution

Listing 4.1 is the complete program.

Listing 4.1 `sort_words.cpp`: insertion sort with a dictionary-order comparison.

```cpp
// sort_words.cpp - Problem 4.1: sort words into alphabetical (dictionary) order
#include <iostream>
#include <string>
using namespace std;

const int MAX_WORDS = 100;

// The word in small letters, so that "Banana" and "banana" look alike.
string lowerCopy(const string& w) {
    string s = w;
    for (size_t i = 0; i < s.size(); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] = s[i] - 'A' + 'a';
        }
    }
    return s;
}

// 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
}

// Insertion sort (Chapter 3): grow a sorted part on the left, word by word.
void insertionSort(string words[], int n) {
    for (int i = 1; i < n; i++) {
        string key = words[i];
        int j = i - 1;
        while (j >= 0 && comesBefore(key, words[j])) {
            words[j + 1] = words[j];   // shift the larger word one place right
            j--;
        }
        words[j + 1] = key;
    }
}
```

Listing 4.1 (Continued) *sort_words.cpp: insertion sort with a dictionary-order comparison.*

```cpp
int main() {
    int n;
    string words[MAX_WORDS];
    cin >> n;
    if (n < 0 || n > MAX_WORDS) {
        cout << "n must be between 0 and " << MAX_WORDS << '\n';
        return 1;
    }
    for (int i = 0; i < n; i++) {
        cin >> words[i];
    }
    cout << "Read " << n << " words:";
    for (int i = 0; i < n; i++) {
        cout << ' ' << words[i];
    }
    cout << "\nSorted:\n";
    insertionSort(words, n);
    for (int i = 0; i < n; i++) {
        cout << words[i] << '\n';
    }
    return 0;
}
```

The important lines, one at a time:

- `lowerCopy` receives the word by `const string&`, a reference that promises not to change it (Chapter 6 treats references in full), and returns a changed copy. The test `s[i] >= 'A' && s[i] <= 'Z'` asks "is this a capital letter?", and `s[i] - 'A' + 'a'` turns, for example, `'C'` (67) into $67 - 65 + 97 = 99$, which is `'c'`.
- `comesBefore` is the rule of Section 4.3.3 written in C++. Notice that it returns `la < lb` only when the copies *differ*; the second `return` handles ties.
- `insertionSort` is **Pseudocode 4.1** line for line. The condition `j >= 0 && comesBefore(key, words[j])` relies on the *short-circuit* rule of `&&`: when `j` has become `-1`, the left side is false and C++ never evaluates `words[-1]`. Swap the two sides and the program reads outside the array.
- `main` checks that n fits into the array of `MAX_WORDS` boxes before reading. A plain array has a fixed size, and writing past its end is not caught by C++ (Chapter 1); a check at the door is cheap insurance.

Run with the input of Section 4.3.2, the program prints:

```
Read 8 words: mango Banana apple Cherry durian banana Apple grape
Sorted:
Apple
apple
Banana
banana
Cherry
durian
grape
mango
```

This is exactly the order we computed by hand. In everyday C++ you would write `std::sort(words, words + n, comesBefore)` or, if the order of ties matters, `std::stable_sort` [5]; both accept the same `comesBefore` as their third argument.

4.3.5 The Java and Python solutions

Listing 4.2 and **Listing 4.3** follow the same design.

Listing 4.2 `SortWords.java`: the same insertion sort in Java.

```java
// SortWords.java - Problem 4.1: sort words into alphabetical (dictionary) order
import java.util.Scanner;

public class SortWords {
    // True when word a must stand before word b in the dictionary.
    static boolean comesBefore(String a, String b) {
        String la = a.toLowerCase(), lb = b.toLowerCase();
        if (!la.equals(lb)) {
            return la.compareTo(lb) < 0;   // different words: dictionary
        }
        return a.compareTo(b) < 0;           // same letters: capital first
    }

    // Insertion sort (Chapter 3): grow a sorted part on the left.
    static void insertionSort(String[] words, int n) {
        for (int i = 1; i < n; i++) {
            String key = words[i];
            int j = i - 1;
            while (j >= 0 && comesBefore(key, words[j])) {
                words[j + 1] = words[j];     // shift the larger word right
                j--;
            }
            words[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        String[] words = new String[n];
        for (int i = 0; i < n; i++) {
            words[i] = in.next();
        }
        System.out.print("Read " + n + " words:");
        for (int i = 0; i < n; i++) {
            System.out.print(" " + words[i]);
        }
        System.out.println();
        System.out.println("Sorted:");
        insertionSort(words, n);
        for (int i = 0; i < n; i++) {
            System.out.println(words[i]);
        }
    }
}
```

Listing 4.3 `sort_words.py`: the same insertion sort in Python.

```python
# sort_words.py - Problem 4.1: sort words into alphabetical (dictionary) order
import sys


def comes_before(a, b):
    """True when word a must stand before word b in the dictionary."""
    la, lb = a.lower(), b.lower()
    if la != lb:
        return la < lb          # different words: dictionary order
    return a < b                # same letters: capital first


def insertion_sort(words):
    """Insertion sort (Chapter 3): grow a sorted part on the left."""
    for i in range(1, len(words)):
        key = words[i]
        j = i - 1
        while j >= 0 and comes_before(key, words[j]):
            words[j + 1] = words[j]     # shift the larger word one place right
            j -= 1
        words[j + 1] = key


def main():
    tokens = sys.stdin.read().split()
    n = int(tokens[0])
    words = tokens[1:1 + n]
    print(f"Read {n} words:", " ".join(words))
    print("Sorted:")
    insertion_sort(words)
    for w in words:
        print(w)


main()
```

Both print, character for character, the output of the C++ program. Three differences between the languages are worth noticing. Java compares strings with `compareTo`, which returns a negative number, zero or a positive number; `a.compareTo(b) < 0` means "`a` before `b`", and `!la.equals(lb)` is how Java asks "are the contents different?" (with `!=` it would ask whether they are different *objects*). Java's `toLowerCase` and Python's `lower` replace our hand-written `lowerCopy`; they also know accented and non-Latin letters, which our ASCII-only version does not. And Python reads the whole input at once with `sys.stdin.read().split()`, which splits on any mixture of spaces and line breaks. The library routes are `Arrays.sort(words, comparator)` in Java [6] and `sorted(words, key=lambda w: (w.lower(), w))` in Python [7]; the Python *key* builds, for every word, the pair (small-letter copy, original word), and pairs are compared first by their first element and then by their second, which is our rule exactly.

4.3.6 A tempting wrong solution

The most tempting shortcut is to keep the sort and compare the words with the plain `<` that strings already offer. **Listing 4.4** does exactly that.

Listing 4.4 `wrong_sort_words.cpp`: the same insertion sort with the plain `<` on strings.

```cpp
// wrong_sort_words.cpp - a tempting wrong solution to Problem 4.1:
// it compares the words with plain <, which compares character codes.
#include <iostream>
#include <string>
using namespace std;

int main() {
    int n;
    string words[100];
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> words[i];
    }
    for (int i = 1; i < n; i++) {          // the same insertion sort ...
        string key = words[i];
        int j = i - 1;
        while (j >= 0 && key < words[j]) { // ... but with plain <
            words[j + 1] = words[j];
            j--;
        }
        words[j + 1] = key;
    }
    for (int i = 0; i < n; i++) {
        cout << words[i] << '\n';
    }
    return 0;
}
```

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

The program runs, and the output is sorted, but by character code, not in dictionary order: all capitalized words come first, then all the others, because every capital letter has a smaller code than every small letter. *Cherry* stands before *apple*. Notice that the error is invisible on an input written entirely in small letters; this is why step 2 asks for an example of your own, and why our example mixes capitals on purpose.

> **CHECK YOURSELF.** Change the program so that it sorts in *reverse* dictionary order (mango first). Which single function do you change, and what is the new first line of the output after `Sorted:`?
>
> *Answer.* Only `comesBefore` changes: return `la > lb` when the copies differ, and `a > b` when they tie. Nothing in `insertionSort` changes, because the sort only ever asks "does this come before that?". The sorted list starts with `mango` and ends with `apple`, then `Apple`, since reversing the whole order also reverses the tie-break.

4.4 Problem 4.2: The First Repeated Character

4.4.1 The problem

> **PROBLEM 4.2.** Find the first duplicate character in a string. *Input:* a string. *Output:* a single character which satisfies the condition. *Example 1.* Input `abbcbbca`, output `a`. *Example 2.* Input `abbcc`, output `b`. *Tip:* the optimal solution is $O(n)$.

4.4.2 Understanding it by hand

This statement looks simple and hides a genuine puzzle: what does “first duplicate character” mean? There are two natural readings.

- **Reading A:** walk from left to right and stop at the first character that you have *already seen*. In `abbcbbca` you see `a`, then `b`, then `b` again, so the answer would be `b`.
- **Reading B:** among the characters that occur *more than once*, take the one whose *first appearance* is leftmost. In `abbcbbca` the letter `a` occurs twice (at indices 0 and 7), `b` four times and `c` twice; the leftmost of them is `a` at index 0, so the answer is `a`.

The examples settle the question: Example 1 says `a`, so the statement means Reading B. Check Example 2 against it: in `abbcc` the letter `a` occurs once, `b` twice and `c` twice; the leftmost character that occurs more than once is `b`, which is the expected output. Reading A would also give `b` on Example 2, which shows why you should always check *every* example, not only the first: Example 2 alone could not tell the two readings apart.

Restated in plain words: *for each character, count how often it occurs in the whole string; the answer is the leftmost character whose count is at least 2.* Our own examples, worked with this rule:

- `abcdef`: every count is 1, so there is no answer. The statement does not say what to print; we decide to print `no repeated character`.
- `Profitina`: counts P 1, r 1, o 1, f 1, i 2, t 1, n 1, a 1; the answer is `i`. Note that `P` and `p` would be different characters, because their codes differ.
- `hello world`: h 1, e 1, l 3; the answer is `l`. The space is a character like any other and occurs once.
- `xyzzyx`: x 2, y 2, z 2; the answer is `x` under Reading B, and `z` under Reading A. This is the example that tells the readings apart best.

4.4.3 Designing it on paper

The obvious design compares every character with every character after it: for i from 0 to $n-1$, for j from $i+1$ to $n-1$, if `s[i] == s[j]` then `s[i]` is the answer. It is correct for Reading B, because it tries the positions i from left to right and stops at the first i that has a partner. But it makes up to $\frac{n(n-1)}{2}$ comparisons (Equation (4.1)); for a string of a million characters that is about $5 \cdot 10^{11}$, and **Table 4.2** tells us that means many minutes. The tip asks for $O(n)$.

The $O(n)$ design trades a little memory for a lot of time. A character in C++ is a byte, a number from 0 to 255 once we read it as `unsigned char`. So we can keep an *array of 256 counters*, one for every possible character, and use the character itself as the index: `count['a']` is the counter for `a`, which is `count[97]`. This is the most important idea of the problem, and one you will meet again as the *hash table* of Chapter 15: *the value you are looking for tells you where to look*, so no search is needed at all. The design needs two passes over the string, as **Pseudocode 4.2** shows.

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
```

Figure 4.3(a) traces it on `abbcbbca`. Pass 1 visits the eight characters and leaves `count['a'] = 2`, `count['b'] = 4`, `count['c'] = 2`. Pass 2 starts again at index 0, sees `a`, looks up `count['a'] = 2`, and stops at once: the answer is `a`. Part (b) of the figure shows what Reading A would do, which is the tempting wrong solution of Section 4.4.6.

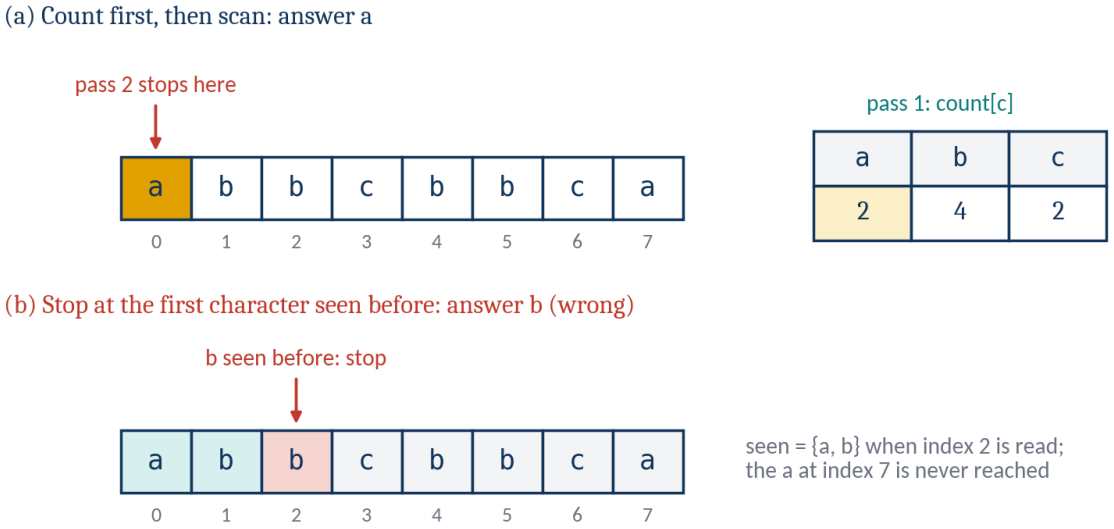


Figure 4.3 (a) Pass 1 fills the counters, pass 2 stops at the first character whose counter is at least 2; (b) stopping at the first character seen before gives the wrong answer b.

The cost. Each pass looks at each of the n characters once and does a constant amount of work there, and creating the 256 counters is a constant amount of work too. The total is $2n + 256$ steps, which is $O(n)$. The memory is 256 counters, whatever the length of the string: $O(1)$ extra space.

4.4.4 The C++ solution

Listing 4.5 reads one string per line and answers for each.

Listing 4.5 `first_repeat.cpp`: two passes over the string with 256 counters.

```cpp
// first_repeat.cpp - Problem 4.2: the first character of a string that occurs
// more than once, found in O(n) time with a table of counts.
#include <iostream>
#include <string>
using namespace std;

// 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;
}

int main() {
    string line;
    while (getline(cin, line)) {
        if (!line.empty() && line.back() == '\r') {
            line.pop_back();               // Windows line ending
        }
        int pos = firstRepeated(line);
        if (pos < 0) {
            cout << line << " -> no repeated character\n";
        } else {
            cout << line << " -> " << line[pos] << '\n';
        }
    }
    return 0;
}
```

- `int count[256] = {0};` creates 256 counters and sets them all to zero. Without `= {0}` a local array in C++ contains whatever happened to be in that memory before (Chapter 1), and the counts would be garbage.
- `count[(unsigned char)s[i]]++` is the heart of the program. On most computers a plain `char` is *signed*, so a byte such as the first byte of the letter é in UTF-8 has a negative value, and a negative index would write outside the array. The cast to `unsigned char` turns every byte into a number from 0 to 255.
- `getline(cin, line)` reads a whole line, spaces included, which is why `hello world` arrives as one string. The two lines after it remove a carriage return `'\r'` that files written on Windows carry at the end of every line.
- The function returns a *position*, not a character, so that "no answer" can be said with `-1`. Returning a character would force us to pick some character to mean "none", and that character might be a real answer.

With the input file `strings.txt`,

```
abbcbbca
abbcc
abcdef
Profitina
hello world
xyzzyx
```

the program prints:

```
abbcbbca -> a
abbcc -> b
abcdef -> no repeated character
Profitina -> i
hello world -> l
xyzzyx -> x
```

Every line agrees with our hand computation, including both examples of the statement.

4.4.5 The Java and Python solutions

Listing 4.6 `FirstRepeat.java`: the same two passes in Java, with one counter per Java `char`.

```
// FirstRepeat.java - Problem 4.2: the first character of a string that occurs
// more than once, found in O(n) time with a table of counts.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class FirstRepeat {
    // Returns the position of the first character that occurs at least twice,
    // or -1 when every character occurs only once.
    static int firstRepeated(String s) {
        int[] count = new int[65536];          // one counter per Java char
        for (int i = 0; i < s.length(); i++) { // pass 1: count them all
            count[s.charAt(i)]++;
        }
        for (int i = 0; i < s.length(); i++) { // pass 2: first count >= 2
            if (count[s.charAt(i)] >= 2) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in =
                new BufferedReader(new InputStreamReader(System.in));
        String line;
        while ((line = in.readLine()) != null) {
            int pos = firstRepeated(line);
            if (pos < 0) {
                System.out.println(line + " -> no repeated character");
            } else {
                System.out.println(line + " -> " + line.charAt(pos));
            }
        }
    }
}
```

Listing 4.7 `first_repeat.py`: the same two passes in Python, with a dictionary of counters.

```python
# first_repeat.py - Problem 4.2: the first character of a string that occurs
# more than once, found in O(n) time with a table of counts.
import sys


def first_repeated(s):
    """Position of the first character that occurs at least twice, or -1."""
    count = {}                          # character -> how many times it occurs
    for ch in s:                        # pass 1: count every character
        count[ch] = count.get(ch, 0) + 1
    for i, ch in enumerate(s):          # pass 2: first one with count >= 2
        if count[ch] >= 2:
            return i
    return -1


for line in sys.stdin:
    line = line.rstrip("\r\n")
    pos = first_repeated(line)
    if pos < 0:
        print(f"{line} -> no repeated character")
    else:
        print(f"{line} -> {line[pos]}")
```

Listing 4.6 and **Listing 4.7** print exactly the same six lines. The Java version needs 65,536 counters instead of 256, because a Java `char` is a 16-bit number (a UTF-16 code unit); the array still has a fixed size, so the extra space is still $O(1)$, just a larger constant. The Python version does not use an array at all but a *dictionary*, `count`, which maps each character to its count; `count.get(ch, 0)` means "the count of `ch`, or 0 if `ch` has not been seen yet". A dictionary is a hash table (Chapter 15), and it has the advantage of working for every character that Python knows, not only for 256 of them. Python also offers `collections.Counter(s)`, which does pass 1 in one call [7].

4.4.6 A tempting wrong solution

Reading A leads to a program that is shorter, makes only one pass, and is wrong. **Listing 4.8** keeps an array `seen` of 256 Boolean values and stops at the first character that has been seen before.

Listing 4.8 `wrong_first_repeat.cpp`: stopping at the first character that was seen before.

```cpp
// wrong_first_repeat.cpp - a tempting wrong solution to Problem 4.2:
// it stops at the first character that has been SEEN BEFORE.
#include <iostream>
#include <string>
using namespace std;

int main() {
    string line;
    while (getline(cin, line)) {
        if (!line.empty() && line.back() == '\r') {
            line.pop_back();
        }
        bool seen[256] = {false};
        int pos = -1;
        for (size_t i = 0; i < line.size(); i++) {
            unsigned char c = line[i];
            if (seen[c]) {                  // c appeared earlier: stop here
                pos = (int)i;
                break;
            }
            seen[c] = true;
        }
        if (pos < 0) {
            cout << line << " -> no repeated character\n";
        } else {
            cout << line << " -> " << line[pos] << '\n';
        }
    }
    return 0;
}
```

```
abbcbbca -> b
abbcc -> b
abcdef -> no repeated character
Profitina -> i
hello world -> l
xyzzyx -> z
```

Four of the six answers agree with the correct program, including Example 2, and that is what makes this solution so tempting: if you test only with `abbcc`, it passes. But on Example 1 it answers `b` instead of `a`, and on `xyzzyx` it answers `z` instead of `x`, exactly as **Figure 4.3**(b) predicts. It finds the character whose *second* appearance comes first, while the problem asks for the character whose *first* appearance comes first. A one-pass program cannot know, when it reads the first `a`, that another `a` is waiting at the very end; that is why the correct design needs to count everything before deciding.

A second trap is worth a sentence because it is so common in C++. It is tempting to write the loops as `for (int i = 0; i <= s.length(); i++)` and to index with `s[i] - 97`, assuming that the string contains only small letters. The `<=` makes the loop visit one position too many, where a C++ string keeps an invisible terminating character with code 0; then `0 - 97 = -97` becomes an index, and the program writes to memory 97 counters *before* the array. On many computers it still prints the right answer, which makes the bug worse, not better. Running with `-fsanitize=address` reports the out-of-bounds write immediately, and it is the reason every C++ program of this chapter was run under the sanitizers.

> **CHECK YOURSELF.** Change the problem to “find the first character that occurs *exactly once*” (the first non-repeated character). What do the six input lines give, and which line of **Pseudocode 4.2** changes?

Answer. Only line 5 changes, to `if count[s[i]] == 1`. The answers are: `abbcbbca` has no such character (every letter occurs at least twice); `abbcc` gives `a`; `abcdef` gives `a`; `Profitina` gives `P`; `hello world` gives `h`; `xyzzyx` has none. The cost is still $O(n)$.

4.5 Problem 4.3: Looking Up Money by Name

4.5.1 The problem

PROBLEM 4.3. Make a program in C++ that can search the amounts of money owned by persons. Print the amount of money if the person exists in the data, and print "Not found" otherwise. *Data:* Puspita 50, Bejo 35, Sarita 20, Pujo 45, Lestari 60. *Example 1.* Input `Pujo`, output `45`. *Example 2.* Input `Komeng`, output `Not found`.

4.5.2 Understanding it by hand

In plain words: we hold a small table of five people, each with an amount of money, and for every name we are asked about we print the amount, or `Not found`. The table is the data; the names typed by the user are the *queries*. By hand, answering `Pujo` means running a finger down the name column: Puspita? no. Bejo? no. Sarita? no. Pujo? yes, the amount is 45. Answering `Komeng` means running the finger down all five rows and falling off the bottom: `Not found`.

We add three queries of our own, each chosen to test one edge:

- `Lestari` is in the *last* row, the worst case for a search from the top: the answer is 60 after five comparisons.
- `Puspita` is in the *first* row, the best case: 50 after one comparison.
- `pujo` with a small `p` is not in the table, because the comparison of strings is exact and `p` (112) is not `P` (80). The answer is `Not found`. Whether it *should* be is a question for whoever wrote the statement; we note the decision (exact match) and keep it.

So for the five queries `Pujo`, `Komeng`, `Lestari`, `pujo`, `Puspita`, the expected answers are 45, Not found, 60, Not found, 50.

4.5.3 Designing it on paper

The structure. Each person is one *record* with two fields, a name and an amount, so we declare a `struct Person { string name; int money; }` and keep the five records in one array, `people[0..4]`. The alternative, two *parallel arrays* `names[]` and `money[]` in which index i of one belongs to index i of the other, works too but has a weakness: if you ever sort one array and forget the other, every amount belongs to the wrong person. A record keeps the two values together by construction.

The algorithm. The data is not sorted by name (Puspita, Bejo, Sarita, … is not alphabetical), so the only search of Chapter 2 that applies without preparation is *sequential search*, shown in **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
```

Figure 4.4(a) traces the search for `Pujo`: four comparisons, the fourth one succeeds, and the answer is `people[3].money`, which is 45. Part (b) of the figure belongs to Section 4.5.6.

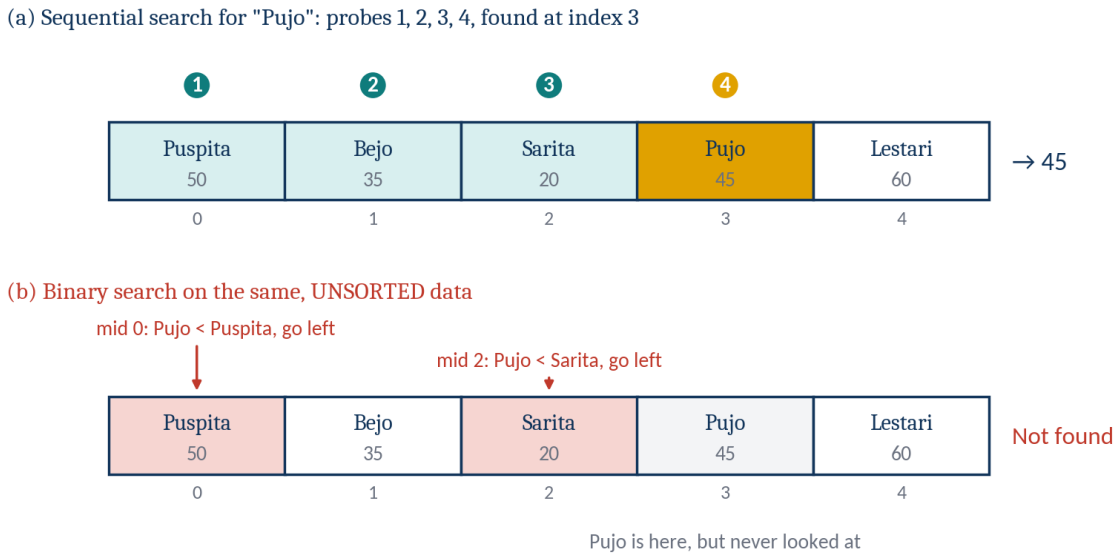


Figure 4.4 (a) Sequential search finds Pujo with its fourth probe; (b) binary search on the same unsorted data discards the part that holds Pujo and reports Not found.

The cost. A search for a name at index i makes $i + 1$ comparisons; a search for a missing name makes n. So one query costs $O(n)$ in the worst case, and q queries cost $O(qn)$. For five people and five queries that is at most 25 comparisons. For a bank with a million customers and a million queries a day it would be 10^{12}, and then the data should be sorted once, in $O(n \log n)$, so that every query can use binary search in $O(\log n)$; the check-yourself of this section and Problem 4.6 take that road.

4.5.4 The C++ solution

Listing 4.9 is the complete program.

Listing 4.9 `search_money.cpp`: sequential search over an array of `Person` records.

```cpp
// search_money.cpp - Problem 4.3: look up how much money a person owns
// with a sequential search (Chapter 2) over an array of records.
#include <iostream>
#include <string>
using namespace std;

struct Person {
    string name;
    int money;
};

// 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
}

int main() {
    const int N = 5;
    Person people[N] = {
        {"Puspita", 50}, {"Bejo", 35}, {"Sarita", 20},
        {"Pujo", 45}, {"Lestari", 60}
    };
    string name;
    while (cin >> name) {                      // one name per query
        int i = sequentialSearch(people, N, name);
        if (i >= 0) {
            cout << people[i].money << '\n';
        } else {
            cout << "Not found\n";
        }
    }
    return 0;
}
```

- `Person people[N] = { {"Puspita", 50}, … };` fills the array of records with *brace initialization*: each inner pair of braces fills one record, field by field, in the order in which the fields are declared.
- `sequentialSearch` receives the array as `const Person people[]`. An array passed to a function in C++ is really passed as the address of its first box (Chapter 6 explains why), so the function cannot know how many boxes there are; that is why `n` travels with it.
- `people[i].name == name` compares two `std::string` objects by content. (In Java the same `==` would compare *references*; see Section 4.5.5.)
- `return i;` inside the loop is the early exit that makes the best case one comparison. A version that sets a flag and keeps looping to the end is correct but always makes n comparisons.
- `while (cin >> name)` keeps reading names until the input ends, so one run answers any number of queries.

With the queries of Section 4.5.2 in the file `names.txt`,

```
Pujo
Komeng
Lestari
pujo
Puspita
```

the program prints one answer per query:

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

The answers agree with our hand computation line by line. When the data grows and the queries are many, C++ programmers keep such a table in a `std::map<string, int>` (a balanced search tree, sorted by name) or a `std::unordered_map<string, int>` (a hash table), which answer a query in $O(\log n)$ and, on average, $O(1)$ steps; Chapters 11 and 15 build both structures from scratch.

4.5.5 The Java and Python solutions

Listing 4.10 and **Listing 4.11** give the same design in Java and in Python.

Listing 4.10 `SearchMoney.java`: the same search with a small `Person` class.

```java
// SearchMoney.java - Problem 4.3: look up how much money a person owns
// with a sequential search (Chapter 2) over an array of records.
import java.util.Scanner;

public class SearchMoney {
    static class Person {
        String name;
        int money;

        Person(String name, int money) {
            this.name = name;
            this.money = money;
        }
    }

    // Returns the index of the person called name, or -1 if there is none.
    static int sequentialSearch(Person[] people, String name) {
        for (int i = 0; i < people.length; i++) {
            if (people[i].name.equals(name)) {   // equals, never == on strings
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        Person[] people = {
            new Person("Puspita", 50), new Person("Bejo", 35),
            new Person("Sarita", 20), new Person("Pujo", 45),
            new Person("Lestari", 60)
        };
        Scanner in = new Scanner(System.in);
        while (in.hasNext()) {                   // one name per query
            String name = in.next();
            int i = sequentialSearch(people, name);
            if (i >= 0) {
                System.out.println(people[i].money);
            } else {
                System.out.println("Not found");
            }
        }
    }
}
```

Listing 4.11 `search_money.py`: the same search over a list of (name, money) pairs.

```
# search_money.py - Problem 4.3: look up how much money a person owns
# with a sequential search (Chapter 2) over a list of records.
import sys

PEOPLE = [("Puspita", 50), ("Bejo", 35), ("Sarita", 20),
          ("Pujo", 45), ("Lestari", 60)]


def sequential_search(people, name):
    """Index of the person called name, or -1 if there is none."""
    for i in range(len(people)):
        if people[i][0] == name:
            return i                    # found: stop at once
    return -1                           # looked at all n: not there


for name in sys.stdin.read().split():   # one name per query
    i = sequential_search(PEOPLE, name)
    if i >= 0:
        print(PEOPLE[i][1])
    else:
        print("Not found")
```

Both print the same five lines as the C++ program. In Java, **Listing 4.10** declares `Person` as a small *nested class* instead of a `struct`, and compares names with `people[i].name.equals(name)`. This is the single most common Java slip with strings: `==` between two `String` objects asks whether they are *the same object in memory*, and a name read from the keyboard is a new object, so `==` would say "different" even when the letters agree. Python's `==` compares contents, as C++ does. **Listing 4.11** stores each record as a *tuple* `("Pujo", 45)`, so `people[i][0]` is the name and `people[i][1]` the amount. Python's own built-in answer to this whole problem is the dictionary: `money = {"Puspita": 50, …}` followed by `money.get(name, "Not found")` [7].

4.5.6 A tempting wrong solution

Chapter 2 made the point that binary search is much faster than sequential search, so it is tempting to use it here. **Listing 4.12** does, on the data exactly as given.

Listing 4.12 `wrong_search_money.cpp`: binary search on data that is not sorted by name.

```cpp
// wrong_search_money.cpp - a tempting wrong solution to Problem 4.3:
// binary search on data that is NOT sorted by name.
#include <iostream>
#include <string>
using namespace std;

int main() {
    const int N = 5;
    string names[N] = {"Puspita", "Bejo", "Sarita", "Pujo", "Lestari"};
    int money[N] = {50, 35, 20, 45, 60};
    string name;
    while (cin >> name) {
        int lo = 0, hi = N - 1, found = -1;
        while (lo <= hi) {
            int mid = (lo + hi) / 2;
            if (names[mid] == name) {
                found = mid;
                break;
            } else if (name < names[mid]) {
                hi = mid - 1;                // "it must be on the left"
            } else {
                lo = mid + 1;                // "it must be on the right"
            }
        }
        if (found >= 0) {
            cout << money[found] << '\n';
        } else {
            cout << "Not found\n";
        }
    }
    return 0;
}
```

```
Not found
Not found
Not found
Not found
50
```

Only `Puspita` is found. **Figure 4.4**(b) shows why `Pujo` is not: the first probe looks at the middle, index 2, `Sarita`; since `Pujo` < `Sarita` alphabetically, binary search concludes "Pujo must be to the left" and throws away indices 2 to 4, including index 3 where Pujo actually sits. The conclusion would be valid in a *sorted* array; in this one it is simply false. The second probe, at index 0, makes the same kind of false conclusion, and the search ends empty-handed. The program does not crash, it does not warn, it just answers `Not found`, which is the most dangerous kind of wrong. Binary search has a precondition, and a precondition that is not met gives no error message, only wrong answers.

CHECK YOURSELF. Sort the five records by name first (with insertion sort, comparing `people[j].name`), then use binary search. What does the sorted array look like, and how many probes does binary search make for `Pujo` and for `Komeng`?

Answer. Sorted by name: Bejo 35, Lestari 60, Pujo 45, Puspita 50, Sarita 20 (records move as a whole, so each amount stays with its owner). For `Pujo`: `lo = 0`, `hi = 4`, `mid = 2` is `Pujo`, found with one probe, answer 45. For `Komeng`: `mid = 2` (Pujo), `Komeng` < `Pujo`, so `hi = 1`; `mid = 0` (Bejo), `Komeng` > `Bejo`, so `lo = 1`; `mid = 1` (Lestari), `Komeng` < `Lestari`, so `hi = 0`; now `lo > hi`, three probes, `Not found`. Both agree with Equation (4.2): at most $\lfloor \log_2 5 \rfloor + 1 = 3$ probes.

4.6 Problem 4.4: Sorting by Importance

4.6.1 The problem

> **PROBLEM 4.4.** You are given an array of strings. Your task is to sort these strings based on their importance. The importance of a string is determined by two factors: (1) the number of `'!'` characters it contains, more `'!'` characters implying higher importance; (2) if two strings have the same number of `'!'` characters, the longer string is considered less important. Write a C++ program to sort the array of strings from most to least important. *Input:* `!important`, `not important at all`, `meh`, `Most Important!!!!`, `how about this?`, `important!`. *Output:* `Most Important!!!!`, `!important`, `important!`, `meh`, `how about this?`, `not important at all`.

4.6.2 Understanding it by hand

In plain words: every string gets a *score* made of two numbers, and we sort by the score. The first number is the count of exclamation marks, and more is better. The second number, which only matters when the first ties, is the length, and shorter is better. We count both by hand for the six strings, carefully, because the length includes spaces and punctuation:

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

Sorting by the first key alone gives three groups: the string with 4 marks, then the two with 1 mark, then the three with 0 marks. Inside the group with 1 mark, `!important` (9 characters) beats `important!` (10). Inside the group with 0 marks, `meh` (3) beats `how about this?` (15), which beats `not important at all` (20). Written out, that is exactly the expected output of the statement. Our own edge case: two strings that tie on *both* keys, such as `a!` and `b!`. The statement says nothing about them; we decide that they keep the order in which they were read. A sort that guarantees this is *stable* (Chapter 3), and that is the reason for our choice of algorithm.

4.6.3 Designing it on paper

This is Problem 4.1 again with a different comparison, which is the whole point of Chapter 3's remark that a sort only needs a rule for "which comes first". **Table 4.3** and **Figure 4.5** show the two keys and the result. The rule `moreImportant(a, b)`, which says whether `a` must stand before `b`, is:

1. Count the marks: c_a and c_b. If they differ, `a` comes first exactly when $c_a > c_b$.
2. Otherwise `a` comes first exactly when it is *strictly* shorter than `b`.

The word *strictly* matters. When the lengths are also equal, the rule must answer "no" in both directions, `moreImportant(a, b)` and `moreImportant(b, a)`, so that insertion sort leaves the two strings where they are; that is what keeps the sort stable. **Pseudocode 4.4** is the same insertion sort as **Pseudocode 4.1** with the new 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
```

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.

The cost. Counting the marks of a string of length L takes L steps, and insertion sort calls the rule up to $\frac{n(n-1)}{2}$ times, so the program does $O(n^2L)$ work. A careful programmer would count the marks of every string once, before sorting, and store the counts in a second array; that makes the counting cost $O(nL)$ in total. For six strings we keep the simpler version.

4.6.4 The C++ solution

Listing 4.13 is the complete program.

Listing 4.13 `importance_sort.cpp`: a two-key comparison rule inside a stable insertion sort.

```cpp
// importance_sort.cpp - Problem 4.4: sort strings from most to least important.
// More '!' means more important; with equal '!', the shorter string wins.
#include <iostream>
#include <string>
using namespace std;

const int MAX_LINES = 100;

int countBangs(const string& s) {
    int c = 0;
    for (char ch : s) {
        if (ch == '!') {
            c++;
        }
    }
    return c;
}

// 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
}

// Insertion sort is stable: strings that tie on both keys keep their order.
void sortByImportance(string lines[], int n) {
    for (int i = 1; i < n; i++) {
        string key = lines[i];
        int j = i - 1;
        while (j >= 0 && moreImportant(key, lines[j])) {
            lines[j + 1] = lines[j];
            j--;
        }
        lines[j + 1] = key;
    }
}
```

Listing 4.13 (Continued) *importance_sort.cpp: a two-key comparison rule inside a stable insertion sort.*

```cpp
int main() {
    string lines[MAX_LINES];
    int n = 0;
    string line;
    while (n < MAX_LINES && getline(cin, line)) {
        if (!line.empty() && line.back() == '\r') {
            line.pop_back();
        }
        lines[n++] = line;
    }
    sortByImportance(lines, n);
    for (int i = 0; i < n; i++) {
        cout << lines[i] << '\n';
    }
    return 0;
}
```

- `for (char ch : s)` is the *range-based for loop* of C++11 [8]: it visits every character of `s` in turn, without an index. It is the right tool when the index itself is not needed.
- `moreImportant` returns `a.size() < b.size()`: strict, as the design demands.

- `main` reads whole lines with `getline`, because these strings contain spaces, and stops at `MAX_LINES` so that the array cannot overflow.

With the six strings of the statement in the file `messages.txt`,

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

the output is:

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

This is exactly the output of the statement. In C++ the library way is `std::stable_sort(lines, lines + n, moreImportant)` [5]. Plain `std::sort` would also give the right answer for *this* input, because no two strings tie on both keys, but it does not promise to keep ties in their input order.

4.6.5 The Java and Python solutions

Listing 4.14 and **Listing 4.15** give the same design in Java and in Python.

Listing 4.14 `ImportanceSort.java`: the same rule and sort in Java.

```java
// ImportanceSort.java - Problem 4.4: sort strings from most to least important.
// More '!' means more important; with equal '!', the shorter string wins.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class ImportanceSort {
    static int countBangs(String s) {
        int c = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '!') {
                c++;
            }
        }
        return c;
    }

    // True when a is MORE important than b, so a must come first.
    static boolean moreImportant(String a, String b) {
        int ca = countBangs(a), cb = countBangs(b);
        if (ca != cb) {
            return ca > cb;                     // first key: more '!' first
        }
        return a.length() < b.length();         // second key: shorter first
    }

    // Insertion sort is stable: ties on both keys keep their input order.
    static void sortByImportance(String[] lines) {
        for (int i = 1; i < lines.length; i++) {
            String key = lines[i];
            int j = i - 1;
            while (j >= 0 && moreImportant(key, lines[j])) {
                lines[j + 1] = lines[j];
                j--;
            }
            lines[j + 1] = key;
        }
    }
```

Listing 4.14 (Continued) *ImportanceSort.java: the same rule and sort in Java.*

```java
    public static void main(String[] args) throws IOException {
        BufferedReader in =
                new BufferedReader(new InputStreamReader(System.in));
        ArrayList<String> read = new ArrayList<>();
        String line;
        while ((line = in.readLine()) != null) {
            read.add(line);
        }
        String[] lines = read.toArray(new String[0]);
        sortByImportance(lines);
        for (String s : lines) {
            System.out.println(s);
        }
    }
}
```

Listing 4.15 `importance_sort.py`: the same rule and sort in Python.

```
# importance_sort.py - Problem 4.4: sort strings from most to least important.
# More '!' means more important; with equal '!', the shorter string wins.
import sys


def more_important(a, b):
    """True when a is MORE important than b, so a must come first."""
    ca, cb = a.count("!"), b.count("!")
    if ca != cb:
        return ca > cb                  # first key: more '!' first
    return len(a) < len(b)              # second key: shorter first


def sort_by_importance(lines):
    """Insertion sort is stable: ties on both keys keep their input order."""
    for i in range(1, len(lines)):
        key = lines[i]
        j = i - 1
        while j >= 0 and more_important(key, lines[j]):
            lines[j + 1] = lines[j]
            j -= 1
        lines[j + 1] = key


lines = [line.rstrip("\r\n") for line in sys.stdin]
sort_by_importance(lines)
for s in lines:
    print(s)
```

Both print the same six lines. The Java program collects the lines in an `ArrayList` first, because it cannot know in advance how many lines will come, and then turns the list into an array with `toArray`. Python counts the marks with the string method `count("!")`. The library routes are instructive. Java's `Arrays.sort` on an array of objects is guaranteed to be stable [6], so `Arrays.sort(lines, Comparator.comparingInt((String s) -> -countBangs(s)).thenComparingInt(String::length))` does the whole job. Python's sort is stable too, and a *key* that returns the pair `(-s.count("!"), len(s))` makes Python compare the negated mark count first and the length second: `sorted(lines, key=lambda s: (-s.count("!"), len(s)))` [7]. The minus sign turns "more marks first" into "smaller number first", which is what an ascending sort does.

4.6.6 A tempting wrong solution

A popular idea is to sort twice with a stable library sort: once by one key and once by the other. It is a good idea, but only in the right order, and the tempting order is the wrong one. **Listing 4.16** sorts by the marks first and by the length second.

Listing 4.16 `wrong_importance_sort.py`: two stable sorts, applied in the wrong order.

```
# wrong_importance_sort.py - a tempting wrong solution to Problem 4.4:
# two stable sorts, but in the wrong order, so the LAST key decides.
import sys

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

The strings are now sorted by length, from `meh` (3) to `not important at all` (20), and the marks count only among strings of equal length. The reason is that a stable sort keeps the previous order only *among ties of the current key*; the last sort therefore decides the main order, and the earlier sort decides only among ties. So the rule for multi-pass sorting is: *sort by the least important key first and by the most important key last* [7]. Swap the two lines of **Listing 4.16**, sorting by `len` first and by the marks second, and the output becomes correct. (The C++ program of Section 4.6.4 avoids the question altogether by combining both keys in one rule.)

> **CHECK YOURSELF.** Add the two lines `Hi!` and `Yo!` to the end of the input, in that order. Where do they appear in the output of the correct program, and in which order?
>
> *Answer.* Both have one mark and length 3, so they belong to the group with one mark and are shorter than `!important` (9). They are placed right after `Most Important!!!!`: the output starts `Most Important!!!!`, `Hi!`, `Yo!`, `!important`, `important!`. `Hi!` stays before `Yo!` because they tie on both keys and insertion sort is stable.

4.7 Problem 4.5: Array Addresses by Hand

The four problems so far came from a classic problem set. The seven that follow are new, and each one isolates one idea of Chapters 1 to 3 that the first four used without looking at it closely. We begin at the bottom, with memory itself.

4.7.1 The problem

> **PROBLEM 4.5.** An array `int A[10]` starts at address `0x1000`, and so does a two-dimensional array `int B[3][5]`. (a) If an `int` occupies 4 bytes, what are the addresses of `A[7]` and `B[2][3]`? (b) The same question for an old compiler on which an `int` occupies 2 bytes. (c) Check part (a) against the real memory of your computer.

4.7.2 Understanding it by hand

In plain words: an array is a row of boxes, all of the same width, lying next to each other in memory, and each byte of memory has a number, its *address*. Addresses are usually written in *hexadecimal* (base 16), with the prefix `0x`; the digits are 0 to 9 and then A to F for ten to fifteen, so `0x1000` is $1 \cdot 16^3 = 4096$ and `0x1C` is $1 \cdot 16 + 12 = 28$. We are asked where box number 7 of `A`, and box row 2, column 3 of `B`, begin.

For `A`: box 0 starts at `0x1000`, box 1 one `int` later, at `0x1004`, box 2 at `0x1008`, and in general box i starts i boxes of 4 bytes after the start. So `A[7]` starts $7 \cdot 4 = 28$ bytes later: $4096 + 28 = 4124$, which is `0x101C`. With 2-byte `int`s it is $7 \cdot 2 = 14$ bytes later, `0x100E`.

For `B`: memory is one long row, not a grid, so the grid must be laid out in a row somehow. C++ stores it *row by row*: first all five boxes of row 0, then all five of row 1, then row 2. This is called *row-major order*. To reach `B[2][3]` we must pass two complete rows of 5 boxes (10 boxes) and then 3 more boxes in row 2, 13 boxes in all. With 4 bytes each that is $13 \cdot 4 = 52$ bytes, so the address is $4096 + 52 = 4148$, which is `0x1034`. With 2 bytes each it is 26 bytes, `0x101A`.

4.7.3 Designing it on paper

The two hand computations become two formulas. For a one-dimensional array with first address b (the *base*) and element size s bytes, element i starts at the address of Equation (4.3).

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

For a two-dimensional array with C columns stored in row-major order, element (r, c) is preceded by r complete rows of C elements and by c elements of its own row, which gives Equation (4.4).

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

Notice what does *not* appear in Equation (4.4): the number of rows. It limits how large r may be, but it never enters the address. **Figure 4.6** draws both arrays: part (a) shows `A` with the address under every box, and part (b) numbers the fifteen cells of `B` in the order in which they lie in memory, so that the cell number $k = r \cdot 5 + c$ can be read off directly. **Pseudocode 4.5** is the computation, which is so short that its value lies entirely in the formulas.

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
```

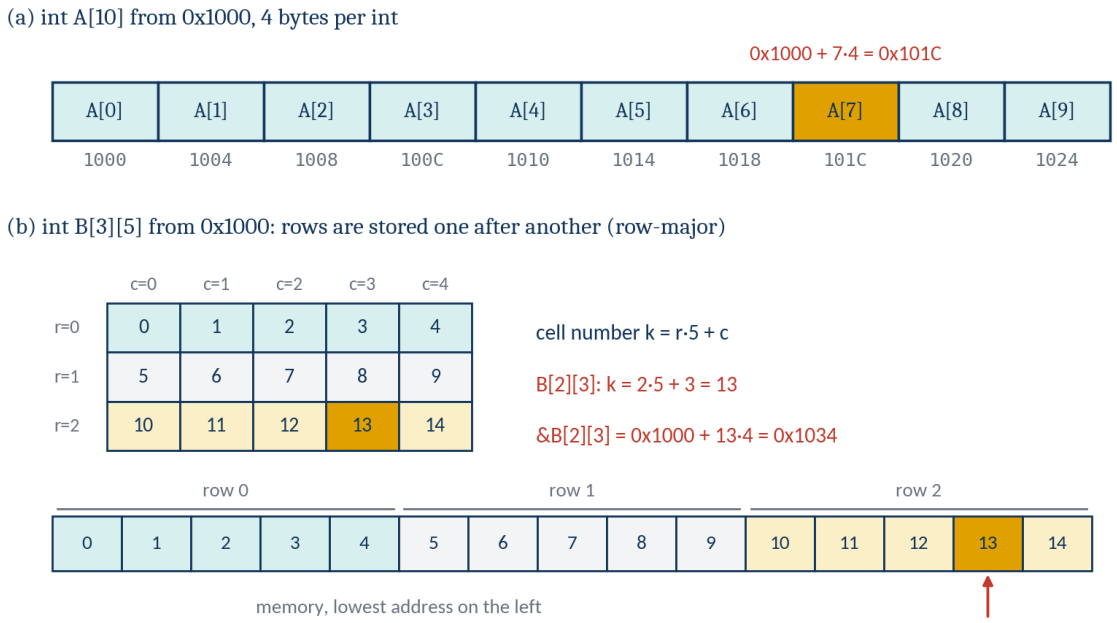


Figure 4.6 (a) The ten boxes of A and their addresses; (b) the grid B, its cells numbered in memory order, and the same cells as they really lie in memory.

The cost is a constant: one multiplication and one addition in one dimension, two of each in two. This constant-time jump to any element is *the* property of arrays, and the reason they are the first data structure of every course. It is also the property that the linked lists of Chapter 7 give up.

4.7.4 The C++ solution

Listing 4.17 computes both answers with the formulas for both sizes of `int` and then asks the real memory of the computer.

Listing 4.17 `array_address.cpp`: the address formulas, checked against the distances in real memory.

```
// array_address.cpp - Problem 4.5: the address of an array element, computed
// by hand with the formulas of Chapter 1 and checked against real memory.
#include <iostream>
#include <cstdio>
using namespace std;

// Address of A[i] in a one-dimensional array that starts at base.
long address1D(long base, int size, int i) {
    return base + (long)i * size;
}

// Address of B[r][c] in a two-dimensional array with cols columns,
// stored row by row (row-major order, as C++ does).
long address2D(long base, int size, int cols, int r, int c) {
    return base + ((long)r * cols + c) * size;
}

int main() {
    long base = 0x1000;
    for (int size = 4; size >= 2; size -= 2) {
        printf("int of %d bytes, both arrays starting at 0x%lX\n", size, base);
        printf("  &A[7]    = 0x%lX + 7 * %d = 0x%lX\n",
               base, size, address1D(base, size, 7));
        printf("  &B[2][3] = 0x%lX + (2 * 5 + 3) * %d = 0x%lX\n",
               base, size, address2D(base, size, 5, 2, 3));
    }

    // The same questions asked of the real memory of this computer.
    int A[10];
    int B[3][5];
    long dA = (long)((char*)&A[7] - (char*)&A[0]);         // distance in bytes
    long dB = (long)((char*)&B[2][3] - (char*)&B[0][0]);
    printf("On this computer sizeof(int) = %zu\n", sizeof(int));
    printf("  &A[7] - &A[0]       = %ld bytes\n", dA);
    printf("  &B[2][3] - &B[0][0] = %ld bytes\n", dB);
    return 0;
}
```

```
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
```

- `printf` with `%lX` prints a `long` in hexadecimal with capital letters, and `%zu` prints the result of `sizeof`, whose type is `size_t`.
- The casts `(long)i * size` make the multiplication happen in `long`. With huge arrays, `int` arithmetic could overflow; with ours it cannot, but the habit costs nothing.
- The last part cannot print the *addresses* of `A[7]` and `B[2][3]` usefully, because the operating system places the arrays at a different address on every run. It prints the *distances* from the start instead: `(char*)&A[7] - (char*)&A[0]`. The casts to `char*` make the subtraction count in bytes; without them C++ would count in elements and print 7 (Chapter 6 explains pointer arithmetic in full). The measured distances, 28 and 52 bytes, are exactly $7 \cdot 4$ and $13 \cdot 4$, which confirms both Equation (4.3) and Equation (4.4) and the row-major layout.

4.7.5 The Java and Python solutions

Java and Python deliberately hide addresses, so **Listing 4.18** and **Listing 4.19** compute the formulas but cannot perform the check of part (c).

Listing 4.18 `ArrayAddress.java`: the address formulas in Java.

```java
// ArrayAddress.java - Problem 4.5: the address of an array element, computed
// by hand with the formulas of Chapter 1. (Java hides real addresses.)
public class ArrayAddress {
    // Address of A[i] in a one-dimensional array that starts at base.
    static long address1D(long base, int size, int i) {
        return base + (long) i * size;
    }

    // Address of B[r][c] in an array with cols columns, stored row by row.
    static long address2D(long base, int size, int cols, int r, int c) {
        return base + ((long) r * cols + c) * size;
    }

    public static void main(String[] args) {
        long base = 0x1000;
        for (int size = 4; size >= 2; size -= 2) {
            System.out.printf("int of %d bytes, both arrays starting at 0x%X%n",
                    size, base);
            System.out.printf("  &A[7]    = 0x%X + 7 * %d = 0x%X%n",
                    base, size, address1D(base, size, 7));
            System.out.printf("  &B[2][3] = 0x%X + (2 * 5 + 3) * %d = 0x%X%n",
                    base, size, address2D(base, size, 5, 2, 3));
        }
    }
}
```

Listing 4.19 `array_address.py`: the address formulas in Python.

```python
# array_address.py - Problem 4.5: the address of an array element, computed
# by hand with the formulas of Chapter 1. (Python hides real addresses.)


def address_1d(base, size, i):
    """Address of A[i] in a one-dimensional array that starts at base."""
    return base + i * size


def address_2d(base, size, cols, r, c):
    """Address of B[r][c] in an array with cols columns, stored row by row."""
    return base + (r * cols + c) * size


base = 0x1000
for size in (4, 2):
    print(f"int of {size} bytes, both arrays starting at 0x{base:X}")
    print(f"  &A[7]    = 0x{base:X} + 7 * {size} = "
          f"0x{address_1d(base, size, 7):X}")
    print(f"  &B[2][3] = 0x{base:X} + (2 * 5 + 3) * {size} = "
          f"0x{address_2d(base, size, 5, 2, 3):X}")
```

Both print the first six lines of the C++ output:

```
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
```

The formulas still matter in these languages. A Java `int[]` is stored exactly as Equation (4.3) says, inside an object that also records its length; that recorded length is what lets Java throw `ArrayIndexOutOfBoundsException` instead of silently reading the neighbor's memory, as we shall see in Problem 4.8. A Java `int[3][5]`, on the other hand, is *not* one block: it is an array of three references to three separate rows of five, so Equation (4.4) applies only within a row. A Python list stores references to objects rather than the numbers themselves; the numeric library NumPy stores its arrays as real blocks, row-major by default, and uses exactly Equation (4.4).

4.7.6 A tempting wrong solution

Two slips are common enough to deserve a program of their own, **Listing 4.20**.

Listing 4.20 `wrong_array_address.py`: forgetting the element size, and multiplying by the number of rows.

```
# wrong_array_address.py - two tempting wrong ways to compute an address.
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
```

The first line adds the *index* 7 to the base instead of 7 *elements* of 4 bytes; the address `0x1007` is the last byte of `A[1]`, not the start of any element. The second multiplies the row number by the number of rows (3) instead of the number of columns (5), which counts $2 \cdot 3 + 3 = 9$ cells instead of 13 and lands on `B[1][4]`. A quick sanity check catches both: the address of any element of an `int` array must differ from the base by a multiple of 4, which `0x1007` does not; and `B[2][3]`, lying in the last row, must be at least two full rows (40 bytes) beyond the base, which `0x1024` (36 bytes) is not.

> **CHECK YOURSELF.** A `double` occupies 8 bytes. An array `double M[4][6]` starts at `0x2000`. What is the address of `M[3][5]`, the very last element, and what is the address of the first byte *after* the array?
>
> *Answer.* By Equation (4.4), $k = 3 \cdot 6 + 5 = 23$ and the address is $0x2000 + 23 \cdot 8 = 0x2000 + 184$; since $184 = 11 \cdot 16 + 8$, that is `0x20B8`. The array holds $4 \cdot 6 = 24$ elements of 8 bytes, 192 bytes in all, so the first byte after it is $ + 192 = $ `0x20C0`, which is also `0x20B8` plus one element of 8 bytes, as it should be.

4.8 Problem 4.6: Counting a Key in a Sorted Array

4.8.1 The problem

> **PROBLEM 4.6.** The Pustaka Ceria library keeps the publication years of one shelf in a sorted array: 1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023. For a given year, how many books on the shelf were published in that year? Answer in $O(\log n)$ time for the years 2001, 2010, 2002, 2023, 1990 and 2030.

4.8.2 Understanding it by hand

In plain words: we must count how many times a value occurs in a sorted array, and we must not look at every element to do it. By hand the counts are easy to read off: 2001 occurs 3 times, 2010 twice, 2023 once, and 2002, 1990 and 2030 not at all.

The key observation is that in a sorted array *equal values stand together in one block*. The three 2001s occupy indices 1, 2 and 3. So instead of counting them one by one, we can find where the block *starts* (index 1) and where the first element *after* the block is (index 4), and subtract: $4 - 1 = 3$. The same works for a value that does not occur: the block of 2002s is empty, it "starts" at index 4 (the first element not smaller than 2002 is 2005, at index 4) and ends at index 4, and $4 - 4 = 0$. For 2030, larger than everything, both boundaries lie *past the end*, at index 9, which is not a box of the array but the position just after the last one; the count is $9 - 9 = 0$.

4.8.3 Designing it on paper

We name the two boundaries. The *lower bound* of a key is the first index whose element is greater than or equal to the key; the *upper bound* is the first index whose element is strictly greater than the key. If no such index exists, the bound is n, the position just past the end. Then the count is simply the difference in Equation (4.5).

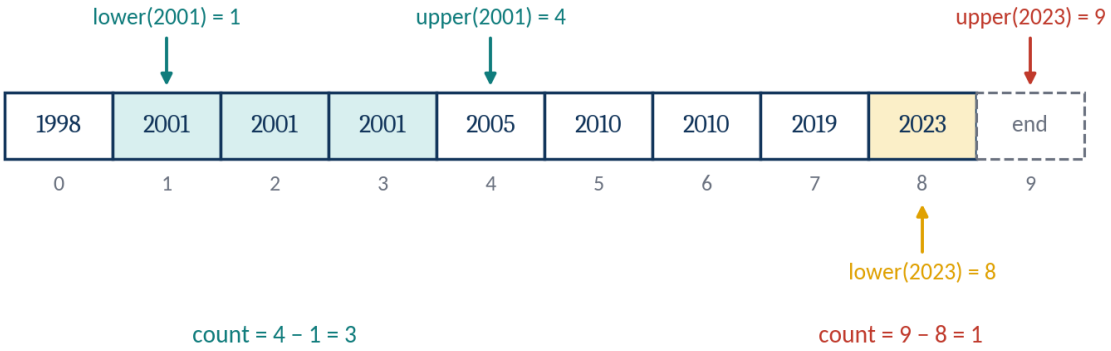
$$\text{count}(key) = \text{upper}(key) - \text{lower}(key) \tag{4.5}$$

How do we find a lower bound in $O(\log n)$ steps? With binary search, but binary search for a *boundary* rather than for an element. Keep a range `[lo, hi]` that is sure to contain the answer; at the start that is `[0, n]`, including the past-the-end position n, since the answer may be n. Look at the middle element `a[mid]`. If it is smaller than the key, the boundary lies to its right, so `lo = mid + 1`. Otherwise `a[mid]` is at least the key, so `mid` itself might be the boundary, and we keep it: `hi = mid`. When `lo` and `hi` meet, the range holds exactly one candidate, and that is the answer. The upper bound is the same search with `<` replaced by `<=`. **Pseudocode 4.6** gives both.

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
```

Figure 4.7 marks the boundaries for 2001 and 2023 and traces `LOWER-BOUND(2001)` step by step. Follow the table: the range `[0, 9]` shrinks to `[0, 4]`, `[0, 2]`, `[0, 1]` and finally `[1, 1]`, after four probes, in agreement with Equation (4.2), which allows at most $\lfloor \log_2 9 \rfloor + 1 = 4$.



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).

Why `mid = lo + (hi - lo) / 2` and not `(lo + hi) / 2`? The two are equal in mathematics, but in a program `lo + hi` can exceed the largest `int` when the array has more than about a billion elements, and the sum then *overflows* into a negative number. This exact bug lived for years in widely used library code before it was noticed [9]; the safe form never computes a number larger than `hi`.

The cost. Each search makes at most $\lfloor \log_2 n \rfloor + 1$ probes, so a count takes $O(\log n)$ steps, whether the key occurs three times or three million times. Counting by walking outward from one found element would cost $O(\log n + k)$ for k copies, which is $O(n)$ when most of the array holds the same key.

4.8.4 The C++ solution

Listing 4.21 is the complete program.

Listing 4.21 `count_range.cpp`: lower bound, upper bound, and their difference.

```cpp
// count_range.cpp - Problem 4.6: how many times does a key occur in a sorted
// array? Two binary searches find where its block starts and where it ends.
#include <iostream>
using namespace std;

// 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;
}

// First index i with a[i] > key (n if there is none).
int upperBound(const int a[], int n, int key) {
    int lo = 0, hi = n;
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] <= key) {
            lo = mid + 1;
        } else {
            hi = mid;
        }
    }
    return lo;
}

int main() {
    const int N = 9;
    int years[N] = {1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023};
    int keys[] = {2001, 2010, 2002, 2023, 1990, 2030};
    for (int key : keys) {
        int first = lowerBound(years, N, key);
        int last = upperBound(years, N, key);
        cout << key << ": lower " << first << ", upper " << last
             << ", count " << last - first << '\n';
    }
    return 0;
}
```

```
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
1990: lower 0, upper 0, count 0
2030: lower 9, upper 9, count 0
```

Every line matches our hand counts, and the lines for 2002, 1990 and 2030 show the boundaries of an empty block: before the first element (0), between two elements (4), and past the end (9). The loop condition `lo < hi` and the update `hi = mid` (not `mid - 1`) belong together: the range always contains at least one candidate, and it shrinks at every step, because `mid` is always strictly less than `hi`. The C++ standard library offers exactly these two functions, `std::lower_bound` and `std::upper_bound`, with the same meaning, and `std::equal_range`, which returns both at once [5].

4.8.5 The Java and Python solutions

Listing 4.22 and **Listing 4.23** give the same design in Java and in Python.

Listing 4.22 `CountRange.java`: the two boundary searches in Java.

```
// CountRange.java - Problem 4.6: how many times does a key occur in a sorted
// array? Two binary searches find where its block starts and where it ends.
public class CountRange {
    // First index i with a[i] >= key (a.length if there is none).
    static int lowerBound(int[] a, int key) {
        int lo = 0, hi = a.length;           // 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;
    }

    // First index i with a[i] > key (a.length if there is none).
    static int upperBound(int[] a, int key) {
        int lo = 0, hi = a.length;
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (a[mid] <= key) {
                lo = mid + 1;
            } else {
                hi = mid;
            }
        }
        return lo;
    }

    public static void main(String[] args) {
        int[] years = {1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023};
        int[] keys = {2001, 2010, 2002, 2023, 1990, 2030};
        for (int key : keys) {
            int first = lowerBound(years, key);
            int last = upperBound(years, key);
            System.out.println(key + ": lower " + first + ", upper " + last
                    + ", count " + (last - first));
        }
    }
}
```

Listing 4.23 `count_range.py`: the two boundary searches in Python.

```python
# count_range.py - Problem 4.6: how many times does a key occur in a sorted
# array? Two binary searches find where its block starts and where it ends.


def lower_bound(a, key):
    """First index i with a[i] >= key (len(a) if there is none)."""
    lo, hi = 0, len(a)                  # the answer lies in [lo, hi]
    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] < key:
            lo = mid + 1            # a[mid] too small: answer right of mid
        else:
            hi = mid                # a[mid] >= key: mid may be the answer
    return lo


def upper_bound(a, key):
    """First index i with a[i] > key (len(a) if there is none)."""
    lo, hi = 0, len(a)
    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] <= key:
            lo = mid + 1
        else:
            hi = mid
    return lo


years = [1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023]
for key in (2001, 2010, 2002, 2023, 1990, 2030):
    first = lower_bound(years, key)
    last = upper_bound(years, key)
    print(f"{key}: lower {first}, upper {last}, count {last - first}")
```

Both print the same six lines as the C++ program. In Python, integers never overflow, so `(lo + hi) // 2` is safe there; Java, like C++, needs the careful form. Python's standard module `bisect` provides the two functions as `bisect_left` and `bisect_right` [7]. Java's `Arrays.binarySearch` returns the index of *some* matching element, not necessarily the first, so counting with it needs extra work [6].

4.8.6 A tempting wrong solution

"An index must lie between 0 and $n - 1$", so it is tempting to start the search range as `[0, n-1]`. **Listing 4.24** does that in both functions.

Listing 4.24 `WrongCountRange.java`: the search range starts as $[0, n-1]$, so the answer n can never be returned.

```java
// WrongCountRange.java - a tempting wrong solution to Problem 4.6:
// the search range starts as [0, n-1], so "past the end" can never be answered.
public class WrongCountRange {
    static int lowerBound(int[] a, int key) {
        int lo = 0, hi = a.length - 1;           // should be a.length
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (a[mid] < key) {
                lo = mid + 1;
            } else {
                hi = mid;
            }
        }
        return lo;
    }

    static int upperBound(int[] a, int key) {
        int lo = 0, hi = a.length - 1;           // should be a.length
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (a[mid] <= key) {
                lo = mid + 1;
            } else {
                hi = mid;
            }
        }
        return lo;
    }

    public static void main(String[] args) {
        int[] years = {1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023};
        int[] keys = {2001, 2010, 2002, 2023, 1990, 2030};
        for (int key : keys) {
            int first = lowerBound(years, key);
            int last = upperBound(years, key);
            System.out.println(key + ": lower " + first + ", upper " + last
                    + ", count " + (last - first));
        }
    }
}
```

```
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 8, count 0
1990: lower 0, upper 0, count 0
2030: lower 8, upper 8, count 0
```

Five of the six lines are right. The line for 2023 says that the shelf holds no book from 2023, although the last book is exactly that. The upper bound of 2023 should be 9, past the end, but a range that starts at `[0, 8]` can never produce 9, so it returns 8, and $8 - 8 = 0$. For 2030 the same mistake happens twice (both bounds become 8 instead of 9) and cancels, which is why that line looks fine. This is the classic off-by-one error of boundary searches, and it hides in exactly one place, the last element; a test with the *last* element as the key finds it, which is why the edge examples of step 2 include one.

> **CHECK YOURSELF.** Using only `LOWER-BOUND`, how many books were published in the years 2000 to 2010 inclusive? Trace the two calls you need.
>
> *Answer.* The books with $2000 \le year \le 2010$ occupy the indices from `LOWER-BOUND(2000)` up to, but not including, `LOWER-BOUND(2011)`. `LOWER-BOUND(2000)` = 1 (the first year ≥ 2000 is the 2001 at index 1) and `LOWER-BOUND(2011)` = 7 (the first year ≥ 2011 is 2019 at index 7), so the count is $7 - 1 = 6$: three books from 2001, one from 2005 and two from 2010.

4.9 Problem 4.7: Interpolation Search on Book Codes

4.9.1 The problem

> **PROBLEM 4.7.** Pustaka Ceria's computer shelf is catalogued by numeric code, sorted: 025 *The C++ Programming*, 034 *Mastering Delphi 6*, 041 *Professional C#*, 056 *Pure JavaScript v2*, 063 *Advanced JSP & Servlet*, 072 *Calculus Make it Easy*, 088 *Visual Basic 2005 Express*, 096 *Artificial Life: Volume 1*. Use interpolation search to look up the codes 88, 60, 41, 25, 96 and 100, print every probe, and compare the number of probes with binary search.

4.9.2 Understanding it by hand

In plain words: we search a sorted array of eight numbers, but instead of always probing the middle, as binary search does, we *estimate* where the key should be from its value. This is how people search a dictionary: nobody looking up "zebra" opens it in the middle.

The estimate assumes that the codes are spread fairly evenly between the smallest and the largest. Between `low` and `high` the codes run from $a[low]$ to $a[high]$; the key lies a fraction $\frac{key - a[low]}{a[high] - a[low]}$ of the way along that range of *values*, so we probe the same fraction of the way along the range of *indices*. That gives the probe position of Equation (4.6), rounded down to a whole index.

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

By hand, for the key 88 with `low = 0` and `high = 7`: the fraction is $\frac{88-25}{96-25} = \frac{63}{71} \approx 0.887$, and $0.887 \cdot 7 \approx 6.21$, which rounds down to 6. And $a[6] = 88$: found with one probe. For the key 60: $\frac{(60-25)\cdot 7}{71} = \frac{245}{71} \approx 3.45$, so the probe is index 3, where $a[3] = 56 < 60$; the key, if present, lies to the right, so `low = 4`. Now $a[4] = 63$ is already larger than 60, so the key cannot lie between $a[4]$ and $a[7]$: not found, after one probe. For the key 41: $\frac{16\cdot 7}{71} \approx 1.58$, probe index 1, $a[1] = 34 < 41$, so `low = 2`; the next probe is $2 + \frac{(41-41)\cdot 5}{96-41} = 2$, and $a[2] = 41$: found after two probes.

4.9.3 Designing it on paper

Pseudocode 4.7 is interpolation search. It has two guards that binary search does not need, and both are essential.

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
```

The test `a[low] <= key <= a[high]` in line 2 is what keeps the probe inside the range: when the key lies between the two end values, the fraction in Equation (4.6) is between 0 and 1, so `pos` lies between `low` and `high`. The test in line 3 handles a range in which all values are equal, where the denominator of Equation (4.6) would be zero. **Figure 4.8** draws the idea: the upper line is the axis of *values*, the lower line the axis of *indices*, and a key is projected from one to the other.

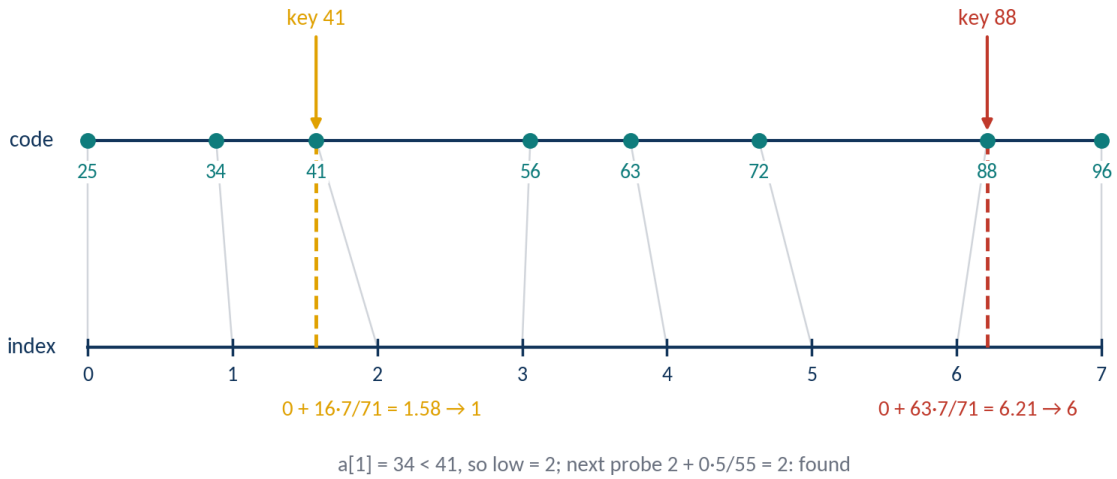


Figure 4.8 Interpolation search projects a key from the axis of code values onto the axis of indices; 88 lands at 6.21 and is found at index 6, while 41 lands at 1.58 and needs a second probe.

The cost. When the keys are spread evenly, interpolation search needs on average only about $\log_2 \log_2 n$ probes [10], which is 5 for a billion items, against 30 for binary search. When they are not, it can be very slow: on the codes 1, 2, 3, …, 99 and 10^9, a search for 99 moves only one index per probe, $O(n)$ probes in all. Binary search never needs more than Equation (4.2), whatever the values [2]. That is the trade: a better average in exchange for a worse worst case.

4.9.4 The C++ solution

Listing 4.25 is the complete program.

Listing 4.25 `interpolation_search.cpp`: interpolation search with a printed trace, next to a binary search that counts its probes.

```cpp
// interpolation_search.cpp - Problem 4.7: interpolation search on the
// Pustaka Ceria book codes, printing every probe and comparing with
// binary search.
#include <iostream>
#include <string>
using namespace std;

const int N = 8;
int codes[N] = {25, 34, 41, 56, 63, 72, 88, 96};
string titles[N] = {"The C++ Programming", "Mastering Delphi 6",
                    "Professional C#", "Pure JavaScript v2",
                    "Advanced JSP & Servlet", "Calculus Make it Easy",
                    "Visual Basic 2005 Express", "Artificial Life: Volume 1"};

int interpolationSearch(int key, int& probes) {
    int low = 0, high = N - 1;
    probes = 0;
    // 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]);
        }
        probes++;
        cout << " probe " << pos << " (code " << codes[pos] << ")";
        if (codes[pos] == key) {
            return pos;
        } else if (codes[pos] < key) {
            low = pos + 1;
        } else {
            high = pos - 1;
        }
    }
    return -1;
}
```

Listing 4.25 (Continued) *interpolation_search.cpp: interpolation search with a printed trace, next to a binary search that counts its probes.*

```
int binarySearchProbes(int key) {
    int low = 0, high = N - 1, probes = 0;
    while (low <= high) {
        int mid = (low + high) / 2;
        probes++;
        if (codes[mid] == key) {
            break;
        } else if (codes[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return probes;
}

int main() {
    int keys[] = {88, 60, 41, 25, 96, 100};
    for (int key : keys) {
        int probes;
        cout << "Search " << key << ":";
        int i = interpolationSearch(key, probes);
        if (i >= 0) {
            cout << " -> found at index " << i << ", \"" << titles[i] << "\"";
        } else {
            cout << " -> not found";
        }
        cout << "; probes: " << probes << " (binary search: "
             << binarySearchProbes(key) << ")\n";
    }
    return 0;
}
```

```
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)
Search 41: probe 1 (code 34) probe 2 (code 41) -> found at index 2, "Professional C#";
probes: 2 (binary search: 3)
Search 25: probe 0 (code 25) -> found at index 0, "The C++ Programming"; probes: 1 (binary
search: 3)
Search 96: probe 7 (code 96) -> found at index 7, "Artificial Life: Volume 1"; probes: 1
(binary search: 4)
Search 100: -> not found; probes: 0 (binary search: 4)
```

The first three lines agree with our hand traces. The others show both ends of the range: 25 and 96 are found at once, because the fraction in Equation (4.6) is exactly 0 or 1. The key 100 is larger than every code, so the guard of line 2 stops the search before the first probe: zero probes. Over these six keys, interpolation search needs 1 or 2 probes where binary search needs 3 or 4, as the theory predicts for evenly spread keys. The expression `(key - codes[low]) * (high - low)` is computed *before* the division, on purpose: dividing first, `(key - codes[low]) / (codes[high] - codes[low])`, would be an integer division of a smaller number by a larger one and give 0 almost every time.

4.9.5 The Java and Python solutions

Listing 4.26 and **Listing 4.27** give the same design in Java and in Python.

Listing 4.26 `InterpolationSearch.java`: the same search in Java.

```java
// InterpolationSearch.java - Problem 4.7: interpolation search on the
// Pustaka Ceria book codes, printing every probe and comparing with
// binary search.
public class InterpolationSearch {
    static int[] codes = {25, 34, 41, 56, 63, 72, 88, 96};
    static String[] titles = {"The C++ Programming", "Mastering Delphi 6",
        "Professional C#", "Pure JavaScript v2",
        "Advanced JSP & Servlet", "Calculus Make it Easy",
        "Visual Basic 2005 Express", "Artificial Life: Volume 1"};
    static int probes;

    static int interpolationSearch(int key) {
        int low = 0, high = codes.length - 1;
        probes = 0;
        // 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]);
            }
            probes++;
            System.out.print(" probe " + pos + " (code " + codes[pos] + ")");
            if (codes[pos] == key) {
                return pos;
            } else if (codes[pos] < key) {
                low = pos + 1;
            } else {
                high = pos - 1;
            }
        }
        return -1;
    }

    static int binarySearchProbes(int key) {
        int low = 0, high = codes.length - 1, count = 0;
        while (low <= high) {
            int mid = (low + high) / 2;
            count++;
            if (codes[mid] == key) {
                break;
            } else if (codes[mid] < key) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
```

Listing 4.26 (Continued) *InterpolationSearch.java: the same search in Java.*

```
        }
        return count;
    }

    public static void main(String[] args) {
        int[] keys = {88, 60, 41, 25, 96, 100};
        for (int key : keys) {
            System.out.print("Search " + key + ":");
            int i = interpolationSearch(key);
            if (i >= 0) {
                System.out.print(" -> found at index " + i
                        + ", \"" + titles[i] + "\"");
            } else {
                System.out.print(" -> not found");
            }
            System.out.println("; probes: " + probes + " (binary search: "
                    + binarySearchProbes(key) + ")");
        }
    }
}
```

Listing 4.27 `interpolation_search.py`: the same search in Python, with the trace collected in a string.

```
# interpolation_search.py - Problem 4.7: interpolation search on the
# Pustaka Ceria book codes, printing every probe and comparing with
# binary search.
CODES = [25, 34, 41, 56, 63, 72, 88, 96]
TITLES = ["The C++ Programming", "Mastering Delphi 6",
          "Professional C#", "Pure JavaScript v2",
          "Advanced JSP & Servlet", "Calculus Make it Easy",
          "Visual Basic 2005 Express", "Artificial Life: Volume 1"]


def interpolation_search(key):
    """Returns (index or -1, number of probes, text describing the probes)."""
    low, high = 0, len(CODES) - 1
    probes, trace = 0, ""
    # The key can only be here while CODES[low] <= key <= CODES[high].
    while low <= high and CODES[low] <= key <= CODES[high]:
        pos = low
        if CODES[high] != CODES[low]:   # avoid dividing by zero
            pos = low + ((key - CODES[low]) * (high - low)
                         // (CODES[high] - CODES[low]))
        probes += 1
        trace += f" probe {pos} (code {CODES[pos]})"
        if CODES[pos] == key:
            return pos, probes, trace
        elif CODES[pos] < key:
            low = pos + 1
        else:
            high = pos - 1
    return -1, probes, trace


def binary_search_probes(key):
    low, high, probes = 0, len(CODES) - 1, 0
    while low <= high:
        mid = (low + high) // 2
        probes += 1
        if CODES[mid] == key:
            break
        elif CODES[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return probes
```

Listing 4.27 (Continued) *interpolation_search.py: the same search in Python, with the trace collected in a string.*

```
for key in (88, 60, 41, 25, 96, 100):
    i, probes, trace = interpolation_search(key)
    if i >= 0:
        result = f' -> found at index {i}, "{TITLES[i]}"'
    else:
        result = " -> not found"
    print(f"Search {key}:{trace}{result}; probes: {probes} "
          f"(binary search: {binary_search_probes(key)})")
```

Both print the same six lines as the C++ program. The Python version returns three values at once, the index, the number of probes and the trace, as a tuple, where the C++ version passes `probes` by reference and prints as it goes. Python writes the guard as the *chained comparison* `CODES[low] <= key <= CODES[high]`, which means exactly the same as the two comparisons joined by `and`, and uses `//` for the integer division of Equation (4.6). Python's `/` would produce a floating-point number, which cannot be used as an index.

4.9.6 A tempting wrong solution

The guard in line 2 of **Pseudocode 4.7** looks like a detail, so it is tempting to write only `while low <= high`, exactly as in binary search. **Listing 4.28** does.

Listing 4.28 `wrong_interpolation.py`: interpolation search without the range guard.

```
# wrong_interpolation.py - a tempting wrong solution to Problem 4.7:
# the loop forgets to check that CODES[low] <= key <= CODES[high].
CODES = [25, 34, 41, 56, 63, 72, 88, 96]


def interpolation_search(key):
    low, high = 0, len(CODES) - 1
    while low <= high:                  # no range check!
        pos = low + ((key - CODES[low]) * (high - low)
                     // (CODES[high] - CODES[low]))
        print(f"  key {key}: probe {pos}")
        if CODES[pos] == key:
            return pos
        elif CODES[pos] < key:
            low = pos + 1
        else:
            high = pos - 1
    return -1


print("88 ->", interpolation_search(88))
print("200 ->", interpolation_search(200))
```

```
  key 88: probe 6
88 -> 6
  key 200: probe 17
Traceback (most recent call last):
  File "/tmp/Chapter04/python/tempting/wrong_interpolation.py", line 22, in <module>
    print("200 ->", interpolation_search(200))
                    ^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/tmp/Chapter04/python/tempting/wrong_interpolation.py", line 12, in
interpolation_search
    if CODES[pos] == key:
       ~~~~~^^^^^
IndexError: list index out of range
```

For the key 88 nothing is wrong. For 200 the fraction $\frac{200-25}{71}$ is much larger than 1, and Equation (4.6) gives $0 + \lfloor 175 \cdot 7/71 \rfloor = 17$, an index far outside an array of eight elements. Python stops with an `IndexError`; Java would stop with an `ArrayIndexOutOfBoundsException`; C++, which does not check indices, would read whatever lies 17 `int`s after the start of the array and carry on with garbage, unless it runs under the sanitizers. A key *smaller* than the first code is worse in Python: it produces a negative index, and Python's negative indices count from the end of the list, so `CODES[-2]` quietly reads 88 instead of failing. The guard is not a detail; it is what makes the formula safe.

> **CHECK YOURSELF.** The code 063 is replaced by 090, so the sorted codes become 25, 34, 41, 56, 72, 88, 90, 96. Trace interpolation search for the key 72.
>
> *Answer.* Probe 1: $0 + \lfloor 47 \cdot 7/71 \rfloor = \lfloor 4.63 \rfloor = 4$, and $a[4] = 72$: found with one probe. (Binary search: `mid` = 3, $56 < 72$, `low` = 4; `mid` = 5, $88 > 72$, `high` = 4; `mid` = 4, found: three probes.)

4.10 Problem 4.8: Tracing the Elementary Sorts

4.10.1 The problem

> **PROBLEM 4.8.** Sort the array 5, 6, 8, 1, 3, 25, 10 into ascending order with bubble sort, selection sort and insertion sort. For each algorithm, write the array after every pass, and count the comparisons between elements and the swaps (or, for insertion sort, the shifts).

4.10.2 Understanding it by hand

In plain words: this is not a problem with one answer but a *trace*, the exercise that turns the three algorithms of Chapter 3 from recipes into something you can see. The array has $n = 7$ elements, and all three algorithms will end with 1, 3, 5, 6, 8, 10, 25. What differs is the road.

Before tracing, it pays to count one thing: the *inversions* of the array, the pairs of elements that stand in the wrong order relative to each other. Go through the array and, for each element, count the smaller elements to its right: 5 has two (1 and 3), 6 has two (1 and 3), 8 has two (1 and 3), 1 has none, 3 has none, 25 has one (10), and 10 has none. That makes $2 + 2 + 2 + 1 = 7$ inversions. Keep that number in mind; it will reappear in the counts.

4.10.3 Designing it on paper

Pseudocode 4.8 collects the three algorithms in one place, exactly as Chapter 3 defined them.

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
```

Figure 4.9 traces the first pass of bubble sort comparison by comparison. The pass compares the neighbors at $j = 0,1, \ldots ,5$ and swaps three times (8 with 1, 8 with 3, 25 with 10). The largest value, 25, is carried like a bubble to the right end, where it stays; that is why the inner loop of the next pass can stop one place earlier.

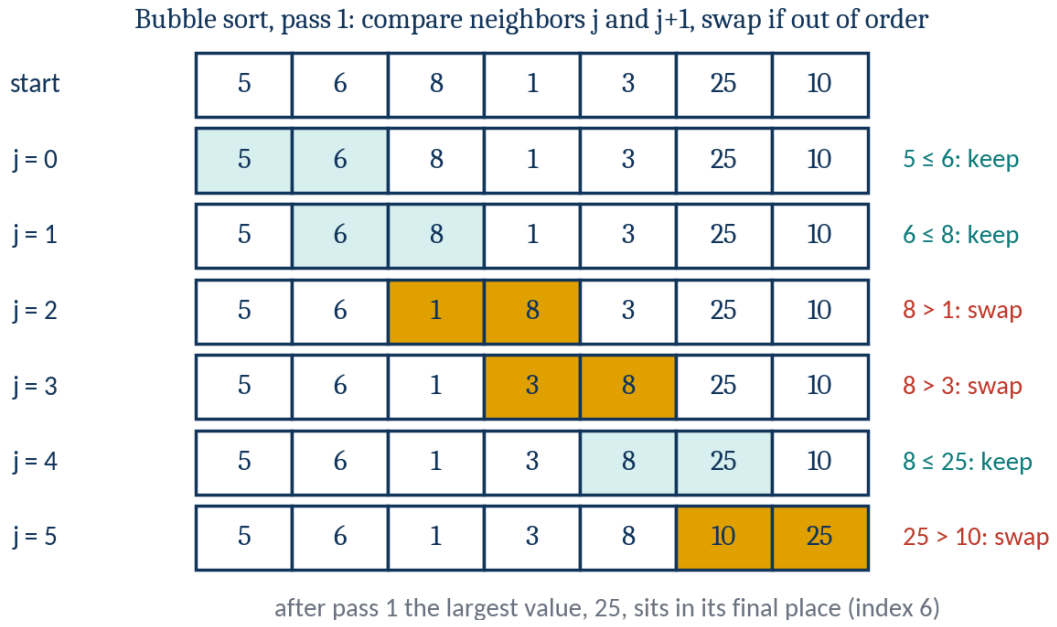


Figure 4.9 The six comparisons of the first pass of bubble sort; amber pairs were swapped, teal pairs were already in order.

Now the counts, derived rather than guessed. *Bubble sort* compares $6 + 5 + 4 + 3 + 2 + 1 = 21$ times, which is Equation (4.1) for $n = 7$, whatever the data. Every swap exchanges two neighbors that are out of order, which removes exactly one inversion and creates none; the array is sorted when no inversions remain; so bubble sort makes exactly as many swaps as there are inversions: 7. *Selection sort* also compares 21 times, since finding the smallest of $n - i$ elements takes $n - i - 1$ comparisons, but it swaps at most once per pass: at most 6 swaps. *Insertion sort* shifts each element left past exactly the larger elements before it, so it also makes exactly 7 shifts, one per inversion; but it compares only until the key finds its place, which is usually much less than 21 times.

The cost. All three are $O(n^2)$ in the worst case, a reversed array, which has the maximum of $\frac{n(n-1)}{2}$ inversions. On an array that is already sorted, insertion sort needs only $n - 1$ comparisons, while bubble sort (in this form) and selection sort still make $\frac{n(n-1)}{2}$. That is why insertion sort is the one used in practice for small or nearly sorted arrays, including inside the library sorts of C++, Java and Python [11].

4.10.4 The C++ solution

Listing 4.29 is the complete program.

Listing 4.29 `trace_sorts.cpp`: the three elementary sorts, each printing the array after every pass and its counts.

```cpp
// trace_sorts.cpp - Problem 4.8: bubble, selection and insertion sort traced
// pass by pass on the same seven numbers, counting comparisons and moves.
#include <iostream>
using namespace std;

const int N = 7;

void show(const int a[]) {
    for (int i = 0; i < N; i++) {
        cout << ' ' << a[i];
    }
    cout << '\n';
}

void copyData(int a[]) {
    int data[N] = {5, 6, 8, 1, 3, 25, 10};
    for (int i = 0; i < N; i++) {
        a[i] = data[i];
    }
}

void bubbleSort(int a[]) {
    int comparisons = 0, swaps = 0;
    for (int i = 0; i < N - 1; i++) {
        for (int j = 0; j < N - 1 - i; j++) {   // the last i places are final
            comparisons++;
            if (a[j] > a[j + 1]) {
                int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
                swaps++;
            }
        }
        cout << "  pass " << i + 1 << ":";
        show(a);
    }
    cout << "  comparisons " << comparisons << ", swaps " << swaps << '\n';
}
```

Listing 4.29 (Continued) *trace_sorts.cpp: the three elementary sorts, each printing the array after every pass and its counts.*

```
void selectionSort(int a[]) {
    int comparisons = 0, swaps = 0;
    for (int i = 0; i < N - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < N; j++) {       // smallest of a[i..N-1]
            comparisons++;
            if (a[j] < a[minIndex]) {
                minIndex = j;
            }
        }
        if (minIndex != i) {
            int t = a[i]; a[i] = a[minIndex]; a[minIndex] = t;
            swaps++;
        }
        cout << "  pass " << i + 1 << ":";
        show(a);
    }
    cout << "  comparisons " << comparisons << ", swaps " << swaps << '\n';
}

void insertionSort(int a[]) {
    int comparisons = 0, shifts = 0;
    for (int i = 1; i < N; i++) {
        int key = a[i];
        int j = i - 1;
        while (j >= 0) {
            comparisons++;
            if (a[j] <= key) {
                break;                          // key has found its place
            }
            a[j + 1] = a[j];                    // shift the larger number right
            shifts++;
            j--;
        }
        a[j + 1] = key;
        cout << "  pass " << i << ":";
        show(a);
    }
    cout << "  comparisons " << comparisons << ", shifts " << shifts << '\n';
}
```

Listing 4.29 (Continued) *trace_sorts.cpp: the three elementary sorts, each printing the array after every pass and its counts.*

```
int main() {
    int a[N];
    copyData(a);
    cout << "Start:";
    show(a);
    cout << "Bubble sort\n";
    bubbleSort(a);
    copyData(a);
    cout << "Selection sort\n";
    selectionSort(a);
    copyData(a);
    cout << "Insertion sort\n";
    insertionSort(a);
    return 0;
}
```

```
Start: 5 6 8 1 3 25 10
Bubble sort
  pass 1: 5 6 1 3 8 10 25
  pass 2: 5 1 3 6 8 10 25
  pass 3: 1 3 5 6 8 10 25
  pass 4: 1 3 5 6 8 10 25
  pass 5: 1 3 5 6 8 10 25
  pass 6: 1 3 5 6 8 10 25
  comparisons 21, swaps 7
Selection sort
  pass 1: 1 6 8 5 3 25 10
  pass 2: 1 3 8 5 6 25 10
  pass 3: 1 3 5 8 6 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 21, swaps 5
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
```

The printed counts confirm every prediction of Section 4.10.3: 21 comparisons and 7 swaps for bubble sort, 21 comparisons and 5 swaps for selection sort (in pass 5 the smallest remaining value was already in place), 12 comparisons and 7 shifts for insertion sort. **Table 4.4** collects them.

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

Notice two things in the trace. Bubble sort had finished its real work after pass 3, yet made three more passes that changed nothing; a Boolean `swapped` flag that ends the sort after a pass without swaps would have stopped after pass 4, the first pass without a swap, saving the $2 + 1 = 3$ comparisons of passes 5 and 6. And the three counting variables (`comparisons`, `swaps`, `shifts`) are exactly the kind of instrumentation that turns a vague feeling ("insertion sort seems faster") into a measurement.

4.10.5 The Java and Python solutions

Listing 4.30 and **Listing 4.31** give the same design in Java and in Python.

Listing 4.30 `TraceSorts.java`: the same three traced sorts in Java.

```java
// TraceSorts.java - Problem 4.8: bubble, selection and insertion sort traced
// pass by pass on the same seven numbers, counting comparisons and moves.
public class TraceSorts {
    static final int[] DATA = {5, 6, 8, 1, 3, 25, 10};

    static void show(int[] a) {
        StringBuilder sb = new StringBuilder();
        for (int x : a) {
            sb.append(' ').append(x);
        }
        System.out.println(sb);
    }

    static void bubbleSort(int[] a) {
        int n = a.length, comparisons = 0, swaps = 0;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - 1 - i; j++) {    // last i places settled
                comparisons++;
                if (a[j] > a[j + 1]) {
                    int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
                    swaps++;
                }
            }
            System.out.print("  pass " + (i + 1) + ":");
            show(a);
        }
        System.out.println("  comparisons " + comparisons + ", swaps " + swaps);
    }

    static void selectionSort(int[] a) {
        int n = a.length, comparisons = 0, swaps = 0;
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {        // smallest of a[i..n-1]
                comparisons++;
                if (a[j] < a[minIndex]) {
                    minIndex = j;
                }
            }
            if (minIndex != i) {
                int t = a[i]; a[i] = a[minIndex]; a[minIndex] = t;
                swaps++;
            }
            System.out.print("  pass " + (i + 1) + ":");
            show(a);
        }
```

Listing 4.30 (Continued) *TraceSorts.java: the same three traced sorts in Java.*

```
        System.out.println("  comparisons " + comparisons + ", swaps " + swaps);
    }

    static void insertionSort(int[] a) {
        int n = a.length, comparisons = 0, shifts = 0;
        for (int i = 1; i < n; i++) {
            int key = a[i];
            int j = i - 1;
            while (j >= 0) {
                comparisons++;
                if (a[j] <= key) {
                    break;                          // key has found its place
                }
                a[j + 1] = a[j];                    // shift the larger right
                shifts++;
                j--;
            }
            a[j + 1] = key;
            System.out.print("  pass " + i + ":");
            show(a);
        }
        System.out.println("  comparisons " + comparisons
                + ", shifts " + shifts);
    }

    public static void main(String[] args) {
        System.out.print("Start:");
        show(DATA);
        System.out.println("Bubble sort");
        bubbleSort(DATA.clone());
        System.out.println("Selection sort");
        selectionSort(DATA.clone());
        System.out.println("Insertion sort");
        insertionSort(DATA.clone());
    }
}
```

Listing 4.31 `trace_sorts.py`: the same three traced sorts in Python.

```
# trace_sorts.py - Problem 4.8: bubble, selection and insertion sort traced
# pass by pass on the same seven numbers, counting comparisons and moves.
DATA = [5, 6, 8, 1, 3, 25, 10]


def row(a):
    return "".join(f" {x}" for x in a)


def bubble_sort(a):
    n, comparisons, swaps = len(a), 0, 0
    for i in range(n - 1):
        for j in range(n - 1 - i):          # the last i places are final
            comparisons += 1
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                swaps += 1
        print(f"  pass {i + 1}:{row(a)}")
    print(f"  comparisons {comparisons}, swaps {swaps}")


def selection_sort(a):
    n, comparisons, swaps = len(a), 0, 0
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):           # find the smallest of a[i..n-1]
            comparisons += 1
            if a[j] < a[min_index]:
                min_index = j
        if min_index != i:
            a[i], a[min_index] = a[min_index], a[i]
            swaps += 1
        print(f"  pass {i + 1}:{row(a)}")
    print(f"  comparisons {comparisons}, swaps {swaps}")
```

Listing 4.31 (Continued) *trace_sorts.py: the same three traced sorts in Python.*

```
def insertion_sort(a):
    n, comparisons, shifts = len(a), 0, 0
    for i in range(1, n):
        key = a[i]
        j = i - 1
        while j >= 0:
            comparisons += 1
            if a[j] <= key:
                break                       # key has found its place
            a[j + 1] = a[j]                 # shift the larger number right
            shifts += 1
            j -= 1
        a[j + 1] = key
        print(f"  pass {i}:{row(a)}")
    print(f"  comparisons {comparisons}, shifts {shifts}")


print(f"Start:{row(DATA)}")
print("Bubble sort")
bubble_sort(DATA[:])
print("Selection sort")
selection_sort(DATA[:])
print("Insertion sort")
insertion_sort(DATA[:])
```

Both print exactly the output of the C++ program. Each sort receives a fresh *copy* of the data, `DATA.clone()` in Java and `DATA[:]` in Python; without the copy, the second sort would receive an

array that the first had already sorted, and the counts would be meaningless. Python swaps two elements in one statement, `a[j], a[j + 1] = a[j + 1], a[j]`, which evaluates the right side completely before assigning; C++ and Java need the three-step swap through a temporary variable `t`.

4.10.6 A tempting wrong solution

The most common bug in a hand-written bubble sort is the bound of the inner loop. **Listing 4.32** lets `j` run up to `n - i - 1` instead of `n - i - 2`, one step too far.

Listing 4.32 `WrongBubble.java`: the inner loop of bubble sort runs one step too far.

```java
// WrongBubble.java - a tempting wrong solution to Problem 4.8:
// the inner loop runs one step too far and reads past the end of the array.
public class WrongBubble {
    public static void main(String[] args) {
        int[] a = {5, 6, 8, 1, 3, 25, 10};
        int n = a.length;
        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;
                }
            }
        }
        System.out.println(java.util.Arrays.toString(a));
    }
}
```

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

In the very first pass, $i = 0$, the last value of `j` is 6, and the comparison `a[j] > a[j + 1]` reads `a[7]`, which does not exist in an array of length 7. Java checks every index and stops the program with an exception that names the bad index, the array length and the line. That is a gift. The same bug in C++ produces no message at all: `a[7]` reads whatever `int` happens to lie after the array, and if that value is smaller than the last element, the swap *writes* into memory the program does not own. The C++ program might print a sorted array, a wrong array, or crash later somewhere else entirely. That is why every C++ program of this chapter was also run under `-fsanitize=address`, which turns such a read into an error report with the exact line.

> **CHECK YOURSELF.** Without running anything: how many comparisons and shifts does insertion sort make on the reversed array 25, 10, 8, 6, 5, 3, 1, and how many on the sorted array 1, 3, 5, 6, 8, 10, 25?
>
> *Answer.* The reversed array has the maximum number of inversions, $\frac{7 \cdot 6}{2} = 21$, so 21 shifts; each key travels to the front, comparing with every element before it and never meeting a smaller one, so 21 comparisons as well. The sorted array has no inversions, so 0 shifts, and each of the 6 keys needs a single comparison to find that it is already in place: 6 comparisons, which is $n - 1$.

4.11 Problem 4.9: Merging Two Sorted Lists

4.11.1 The problem

> **PROBLEM 4.9.** Two branches of Pustaka Ceria keep their new book codes in sorted arrays: branch A has 3, 9, 12, 15, 31 and branch B has 5, 9, 17, 23, 35, 40, 41. Produce the combined catalog as one sorted array, in one pass, without sorting from scratch, and count the comparisons.

4.11.2 Understanding it by hand

In plain words: we have two sorted piles and want one sorted pile. The quickest way by hand is the way you would merge two sorted stacks of cards lying face up: look at the top card of each stack, take the smaller one, and repeat. The top cards are 3 (A) and 5 (B): take 3. Now 9 (A) and 5 (B): take 5. Now 9 and 9: a tie; we take the one from A first, so that equal codes keep the order "A before B" (this keeps the merge stable). Now 12 and 9: take 9 from B. Then 12 (A), 15 (A), 17 (B), 23 (B), 31 (A). Branch A is now empty, and we have made nine comparisons, one for each card taken while both stacks had cards. The rest of B, 35, 40, 41, can be taken without any comparison, because there is nothing left to compare it with. The result is 3, 5, 9, 9, 12, 15, 17, 23, 31, 35, 40, 41: twelve codes.

4.11.3 Designing it on paper

We keep three indices: `i` into A, `j` into B, and `k` into the result C. **Pseudocode 4.9** is the merge, the step that gives merge sort its name (Chapter 3).

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
```

Figure 4.10 draws where every element of C came from: teal lines from A, amber lines from B, and dashed lines for the tail of B that was copied without comparisons. The small number next to each element of C is the comparison that placed it.

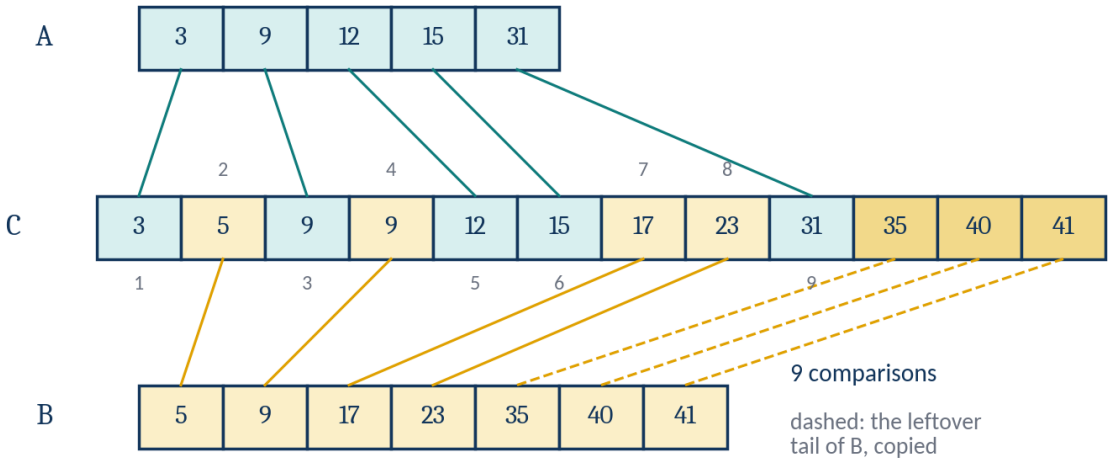


Figure 4.10 Merging A and B into C: each element of C is joined to its source, the small numbers count the comparisons, and the dashed tail of B is copied without comparisons.

Why is the result sorted? At every step, the element taken is the smallest of all the elements not yet taken: A and B are sorted, so the smallest remaining element of A is `a[i]` and the smallest remaining element of B is `b[j]`, and we take the smaller of these two. Since each element taken is at least as large as everything taken before it, C grows in ascending order.

The cost. Every comparison places one element into C, and the last element needs no comparison, so a merge of n and m elements makes at most $n + m - 1$ comparisons, and $n + m$ copies: $O(n + m)$ steps. Here $n + m - 1 = 11$, but only 9 were needed, because A ran out while three elements of B remained. Sorting the twelve codes from scratch with insertion sort could cost up to $\frac{12 \cdot 11}{2} = 66$ comparisons; the sortedness of the two inputs is worth having.

4.11.4 The C++ solution

Listing 4.33 is the complete program.

Listing 4.33 `merge_sorted.cpp`: the merge of two sorted arrays, counting comparisons.

```cpp
// merge_sorted.cpp - Problem 4.9: merge two sorted arrays into one sorted array
// in a single left-to-right walk (the heart of merge sort, Chapter 3).
#include <iostream>
using namespace std;

void show(const char* label, const int a[], int n) {
    cout << label;
    for (int i = 0; i < n; i++) {
        cout << ' ' << a[i];
    }
    cout << '\n';
}

// Merges a[0..n-1] and b[0..m-1] into c; returns the number of comparisons.
int mergeArrays(const int a[], int n, const int b[], int m, int c[]) {
    int i = 0, j = 0, k = 0, comparisons = 0;
    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++];
    }
    return comparisons;
}

int main() {
    int a[] = {3, 9, 12, 15, 31};
    int b[] = {5, 9, 17, 23, 35, 40, 41};
    int n = 5, m = 7;
    int c[12];
    int comparisons = mergeArrays(a, n, b, m, c);
    show("Branch A:", a, n);
    show("Branch B:", b, m);
    show("Merged:  ", c, n + m);
    cout << "Comparisons: " << comparisons << '\n';
    return 0;
}
```

```
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
```

The output is the list we built by hand, with the nine comparisons we counted. `c[k++] = a[i++];` is a compact C++ idiom: it copies `a[i]` into `c[k]` and *then* increases both `i` and `k` by one, because the `++` written after a variable (the *postfix* form) uses the old value first. The result array `c` has exactly $5 + 7 = 12$ boxes; the merge never writes more than $n + m$ elements, so it can never overflow. The library version is `std::merge(a, a + n, b, b + m, c)` [5].

4.11.5 The Java and Python solutions

Listing 4.34 and **Listing 4.35** give the same design in Java and in Python.

Listing 4.34 `MergeSorted.java`: the same merge in Java, returning a new array.

```java
// MergeSorted.java - Problem 4.9: merge two sorted arrays into one sorted array
// in a single left-to-right walk (the heart of merge sort, Chapter 3).
public class MergeSorted {
    static int comparisons;

    static void show(String label, int[] a) {
        StringBuilder sb = new StringBuilder(label);
        for (int x : a) {
            sb.append(' ').append(x);
        }
        System.out.println(sb);
    }

    static int[] mergeArrays(int[] a, int[] b) {
        int[] c = new int[a.length + b.length];
        int i = 0, j = 0, k = 0;
        comparisons = 0;
        while (i < a.length && j < b.length) {   // 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 < a.length) {                   // copy what is left of a ...
            c[k++] = a[i++];
        }
        while (j < b.length) {                   // ... or what is left of b
            c[k++] = b[j++];
        }
        return c;
    }

    public static void main(String[] args) {
        int[] a = {3, 9, 12, 15, 31};
        int[] b = {5, 9, 17, 23, 35, 40, 41};
        int[] c = mergeArrays(a, b);
        show("Branch A:", a);
        show("Branch B:", b);
        show("Merged:  ", c);
        System.out.println("Comparisons: " + comparisons);
    }
}
```

Listing 4.35 `merge_sorted.py`: the same merge in Python, with the tails added by `extend`.

```
# merge_sorted.py - Problem 4.9: merge two sorted arrays into one sorted array
# in a single left-to-right walk (the heart of merge sort, Chapter 3).


def merge_arrays(a, b):
    """Returns (merged list, number of comparisons)."""
    c = []
    i = j = comparisons = 0
    while i < len(a) and j < len(b):    # both lists still have items
        comparisons += 1
        if a[i] <= b[j]:                # <= takes a tie from a first
            c.append(a[i])
            i += 1
        else:
            c.append(b[j])
            j += 1
    c.extend(a[i:])                     # copy what is left of a ...
    c.extend(b[j:])                     # ... or what is left of b
    return c, comparisons


a = [3, 9, 12, 15, 31]
b = [5, 9, 17, 23, 35, 40, 41]
c, comparisons = merge_arrays(a, b)
print("Branch A:", *a)
print("Branch B:", *b)
print("Merged:  ", *c)
print("Comparisons:", comparisons)
```

Both print exactly the same four lines. The Java method creates the result array itself, with the right size, and returns it, which is safer than asking the caller for an array of the right size. The Python version builds a list with `append` and adds both leftover tails with `extend`; one of the two slices `a[i:]` and `b[j:]` is always empty, so adding both is correct and short. Python's standard library offers `heapq.merge(a, b)`, which merges any number of sorted sequences [7].

4.11.6 A tempting wrong solution

The main loop feels like the whole algorithm, and the two little copy loops after it feel like an afterthought. **Listing 4.36** leaves them out.

Listing 4.36 `wrong_merge.cpp`: a merge that forgets the leftover elements.

```cpp
// wrong_merge.cpp - a tempting wrong solution to Problem 4.9:
// the loop stops when ONE array is used up, and the rest is forgotten.
#include <iostream>
using namespace std;

int main() {
    int a[] = {3, 9, 12, 15, 31};
    int b[] = {5, 9, 17, 23, 35, 40, 41};
    int n = 5, m = 7;
    int c[12];
    int i = 0, j = 0, k = 0;
    while (i < n && j < m) {
        if (a[i] <= b[j]) {
            c[k++] = a[i++];
        } else {
            c[k++] = b[j++];
        }
    }
    cout << "Merged:";
    for (int t = 0; t < k; t++) {
        cout << ' ' << c[t];
    }
    cout << "\n(" << k << " of " << n + m << " items)\n";
    return 0;
}
```

```
Merged: 3 5 9 9 12 15 17 23 31
(9 of 12 items)
```

Nine of twelve codes survive; 35, 40 and 41 are gone. The main loop stops as soon as *one* of the arrays is exhausted, and whatever remains in the other one is never copied. The program looks right on inputs whose largest elements happen to alternate between the arrays until the very end, which is why it survives casual testing. The printed count, `(9 of 12 items)`, is a cheap check that every merge should make: the output must contain exactly $n + m$ elements.

> **CHECK YOURSELF.** The combined catalog must list each code once, so the duplicate 9 must disappear. Which lines of **Pseudocode 4.9** change, and what are the result and the number of comparisons?
>
> *Answer.* Add a third case to the main loop: when `a[i] == b[j]`, copy the element once and advance *both* `i` and `j`. The result is 3, 5, 9, 12, 15, 17, 23, 31, 35, 40, 41 (eleven codes). The comparisons: 3 against 5, 9 against 5, 9 against 9 (equal, both advance), 12 against 17, 15 against 17, 31 against 17, 31 against 23, 31 against 35: eight passes through the main loop. (If each pass tests `<`, `==` and `>` separately, count the *passes*, not the individual tests.)

4.12 Problem 4.10: Deleting a Value in Place

4.12.1 The problem

> **PROBLEM 4.10.** An array holds the numbers 7, 3, 3, 5, 3, 9. Delete every occurrence of a given value x from it *in place* (without a second array), keeping the other numbers in their order, and report the new size. Test with x = 3, 7, 9 and 4, and on the array 3, 3, 3 with $x = 3$.

4.12.2 Understanding it by hand

In plain words: an array in C++ has a fixed number of boxes, so “deleting” cannot make the array physically shorter. What it means is: move the numbers we keep to the front, in their order, and remember how many of the front boxes now count. That number is the new *size*; whatever lies in the boxes after it no longer belongs to the array, even if it is still in memory. This separation of

capacity (boxes that exist) and *size* (boxes in use) runs through the whole course: the stack and the queue of Chapter 5 are built on it.

By hand, deleting 3 from 7, 3, 3, 5, 3, 9 leaves 7, 5, 9, a new size of 3, and 3 elements removed. Deleting 7 (the first element) leaves 3, 3, 5, 3, 9; deleting 9 (the last) leaves 7, 3, 3, 5, 3; deleting 4 (absent) changes nothing; and deleting 3 from 3, 3, 3 leaves an empty array of size 0. The edges are all here: first, last, absent, everything, and two equal neighbors in a row.

4.12.3 Designing it on paper

The obvious design follows the "search, and if found, delete" recipe of Chapter 2: find a 3, shift everything after it one place to the left, reduce the size, and continue. It works if you are careful (see Section 4.12.6), but each deletion shifts up to n elements, so deleting many values costs $O(n^2)$.

The one-pass design uses two indices that move through the same array: a *read* index `r` that visits every element once, and a *write* index `w` that marks the next free box of the kept part. Whenever `a[r]` is a number we keep, we copy it to `a[w]` and advance `w`; when it is x, we only advance `r`. **Pseudocode 4.10** is the whole algorithm.

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
```

Figure 4.11(a) traces it for $x = 3$. The invariant, the property that holds before every pass of the loop, is: *`a[0..w-1]` holds, in their original order, exactly the kept elements among `a[0..r-1]`.* It is true at the start, when both ranges are empty. Each pass keeps it true: a kept element is appended at `w`, a deleted one is skipped. When the loop ends, `r = n`, so `a[0..w-1]` holds exactly the kept elements of the whole array. Why can writing to `a[w]` never destroy an element we still need? Because `w <= r` always holds (`w` grows at most as fast as `r`), so we only ever overwrite a box that the read index has already passed.

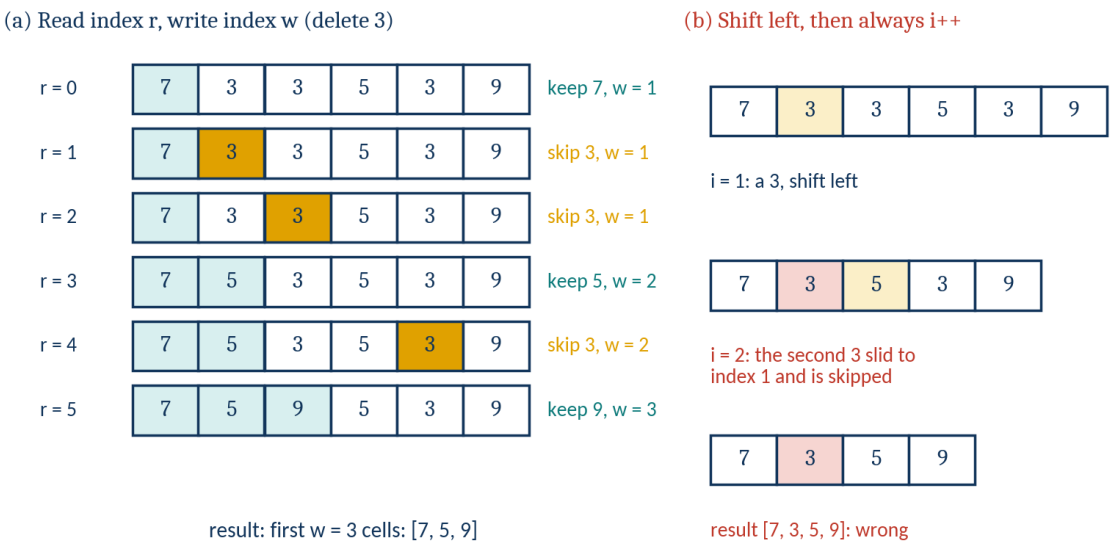


Figure 4.11 (a) The read index r visits every box once and the write index w packs the kept numbers to the front; (b) shifting left and then always advancing i skips the second of two adjacent 3s.

The cost. One pass, one comparison and at most one copy per element: $O(n)$ time and $O(1)$ extra space, against $O(n^2)$ for repeated shifting.

4.12.4 The C++ solution

Listing 4.37 is the complete program.

Listing 4.37 `delete_value.cpp`: deleting every occurrence of a value with a read index and a write index.

```cpp
// delete_value.cpp - Problem 4.10: delete every occurrence of a value from an
// array in place, in one pass, keeping the order of the other items.
#include <iostream>
using namespace std;

void show(const int a[], int n) {
    cout << '[';
    for (int i = 0; i < n; i++) {
        cout << (i > 0 ? ", " : "") << a[i];
    }
    cout << ']';
}

// 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;
}

void test(const int start[], int n, int x) {
    int a[10] = {0};
    for (int i = 0; i < n; i++) {
        a[i] = start[i];
    }
    cout << "delete " << x << " from ";
    show(a, n);
    int newSize = deleteAll(a, n, x);
    cout << " -> ";
    show(a, newSize);
    cout << " (" << n - newSize << " removed)\n";
}
```

Listing 4.37 (Continued) *delete_value.cpp: deleting every occurrence of a value with a read index and a write index.*

```cpp
int main() {
    int data[] = {7, 3, 3, 5, 3, 9};
    int threes[] = {3, 3, 3};
    test(data, 6, 3);
    test(data, 6, 7);
    test(data, 6, 9);
    test(data, 6, 4);
    test(threes, 3, 3);
    return 0;
}
```

```
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)
delete 9 from [7, 3, 3, 5, 3, 9] -> [7, 3, 3, 5, 3] (1 removed)
delete 4 from [7, 3, 3, 5, 3, 9] -> [7, 3, 3, 5, 3, 9] (0 removed)
delete 3 from [3, 3, 3] -> [] (3 removed)
```

All five lines agree with the hand results. The function `test` copies the original data into a local array first, so that every test starts from the same numbers; `int a[10] = {0};` gives that local array well-defined contents in every box, including the ones the test does not use. The expression `(i > 0 ? ", " : "")` in `show` is the *conditional operator*: it prints the separator before every element except the first, which is the standard trick for printing a list without a trailing comma.

The C++ library performs this exact algorithm in `std::remove(a, a + n, x)`, which returns a pointer to the new end; the name misleads many programmers, because it does not change the size of anything, it only packs the kept elements to the front [5].

4.12.5 The Java and Python solutions

Listing 4.38 and **Listing 4.39** give the same design in Java and in Python.

Listing 4.38 `DeleteValue.java`: the same two-index deletion in Java.

```
// DeleteValue.java - Problem 4.10: delete every occurrence of a value from an
// array in place, in one pass, keeping the order of the other items.
public class DeleteValue {
    static String show(int[] a, int n) {
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < n; i++) {
            sb.append(i > 0 ? ", " : "").append(a[i]);
        }
        return sb.append(']').toString();
    }

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

    static void test(int[] start, int x) {
        int[] a = start.clone();
        int n = a.length;
        String before = show(a, n);
        int newSize = deleteAll(a, n, x);
        System.out.println("delete " + x + " from " + before + " -> "
                + show(a, newSize) + " (" + (n - newSize) + " removed)");
    }

    public static void main(String[] args) {
        int[] data = {7, 3, 3, 5, 3, 9};
        test(data, 3);
        test(data, 7);
        test(data, 9);
        test(data, 4);
        test(new int[] {3, 3, 3}, 3);
    }
}
```

Listing 4.39 `delete_value.py`: the same two-index deletion in Python.

```
# delete_value.py - Problem 4.10: delete every occurrence of a value from an
# array in place, in one pass, keeping the order of the other items.


def show(a, n):
    return "[" + ", ".join(str(a[i]) for i in range(n)) + "]"


def delete_all(a, n, x):
    """Keeps every item that is not x, packed to the front; returns new size."""
    w = 0                                # next free place for an item we keep
    for r in range(n):                   # r reads every item exactly once
        if a[r] != x:
            a[w] = a[r]
            w += 1
    return w


def test(start, x):
    a = start[:]
    n = len(a)
    before = show(a, n)
    new_size = delete_all(a, n, x)
    print(f"delete {x} from {before} -> {show(a, new_size)} "
          f"({n - new_size} removed)")


data = [7, 3, 3, 5, 3, 9]
test(data, 3)
test(data, 7)
test(data, 9)
test(data, 4)
test([3, 3, 3], 3)
```

Both print the same five lines. They keep the size in a separate variable too, to show the algorithm exactly as in C++. In everyday Python the idiom is a *list comprehension*, `a[:] = [v for v in a if v != x]`, which builds the kept list and then replaces the contents of `a` with it; the assignment to the slice `a[:]` keeps the same list object, so other variables that refer to it see the change. In Java one would use an `ArrayList<Integer>` and `list.removeIf(v -> v == x)`.

4.12.6 A tempting wrong solution

The obvious design of Section 4.12.3, written the obvious way, is **Listing 4.40**.

Listing 4.40 `wrong_delete_value.py`: shift-left deletion that always advances the index.

```
# wrong_delete_value.py - a tempting wrong solution to Problem 4.10:
# after shifting the rest left, i moves on and skips the item that slid in.
a = [7, 3, 3, 5, 3, 9]
n = len(a)
x = 3
i = 0
while i < n:
    if a[i] == x:
        for k in range(i, n - 1):        # shift a[i+1..n-1] one place left
            a[k] = a[k + 1]
        n -= 1
    i += 1                               # always moves on: the bug
print(a[:n])
```

```
[7, 3, 5, 9]
```

One 3 survived. **Figure 4.11**(b) shows why: at `i = 1` the program finds a 3 and shifts everything after it left, which moves the *second* 3 into box 1. Then `i += 1` moves on to box 2, and the 3 now

sitting in box 1 is never looked at. The bug appears only when two copies of x stand next to each other; on 7, 3, 5, 3, 9 the program works, which is why tests with adjacent duplicates matter. The fix is to advance `i` only when nothing was deleted (in an `else` branch), which makes the design correct but still $O(n^2)$. The two-index design avoids the question entirely, because it never moves an element that has not been read yet.

> **CHECK YOURSELF.** Adapt **Pseudocode 4.10** to remove *duplicates* from a sorted array, keeping one copy of each value: 1, 1, 2, 3, 3, 3, 7 must become 1, 2, 3, 7. What changes, and what is the new size?
>
> *Answer.* In a sorted array the duplicates of a value stand together, so an element is kept exactly when it differs from the last element kept. Start with `w = 1` (the first element is always kept, if the array is not empty) and let `r` run from 1; keep `a[r]` when `a[r] != a[w - 1]`. The kept elements are 1, 2, 3, 7, so the new size is 4, still in one pass. (This is `std::unique` in C++.)

4.13 Problem 4.11: Two Prices That Match a Voucher

4.13.1 The problem

> **PROBLEM 4.11.** Profitina Laundry offers eight services at the prices (in rupiah) 18000, 7000, 25000, 12000, 30000, 5000, 20000 and 15000. A customer holds a voucher that pays for *exactly two different services* whose prices add up to the value of the voucher. For vouchers of 37000, 12000 and 11000, list every pair of prices that matches, using fewer steps than trying all pairs.

4.13.2 Understanding it by hand

In plain words: given eight numbers and a target, find the pairs of numbers whose sum is the target. The honest first method is to try every pair: the first price with each of the seven others, the second with the six after it, and so on, which is $\frac{8 \cdot 7}{2} = 28$ pairs by Equation (4.1). Doing that by hand for 37000 finds two pairs: $7000 + 30000$ and $12000 + 25000$. For 12000 it finds one, $5000 + 7000$. For 11000 it finds none: the two cheapest services already cost 12000, more than 11000.

Twenty-eight pairs is little. But the same question for a catalog of a million products would mean about $5 \cdot 10^{11}$ pairs, and **Table 4.2** says no. We want a method that grows like $n \log n$.

4.13.3 Designing it on paper

Step 1: sort. Sorted, the prices are 5000, 7000, 12000, 15000, 18000, 20000, 25000, 30000.

Step 2: walk inward from both ends. Put an index `i` on the cheapest price and `j` on the dearest, and look at their sum:

- If the sum *equals* the target, we have a pair. Record it and move both indices inward.
- If the sum is *too small*, the price at `i` cannot be part of any pair: even its dearest possible partner, `a[j]`, is not enough, and every other partner still available is cheaper. So we drop `a[i]` by moving `i` one step right.
- If the sum is *too large*, the price at `j` cannot be part of any pair: even its cheapest possible partner, `a[i]`, gives too much. So we drop `a[j]` by moving `j` one step left.

The walk ends when the indices meet. **Pseudocode 4.11** is the method, and **Figure 4.12** traces the voucher of 37000: five steps instead of 28 pairs.

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
```

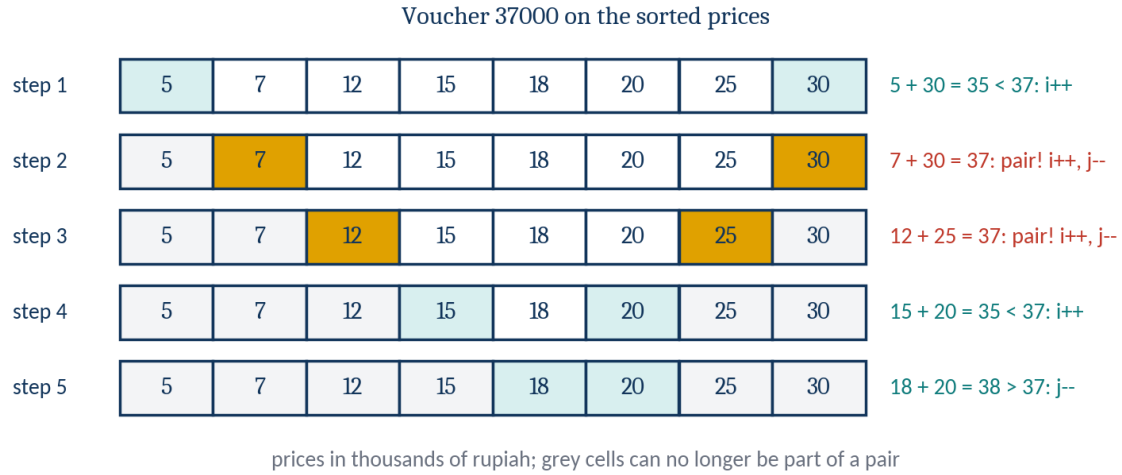


Figure 4.12 The two-index walk for the voucher of 37000 on the sorted prices, in thousands of rupiah: each step either finds a pair or discards the cheapest or the dearest remaining price.

Why can the walk never miss a pair? Every step discards one or two prices, and we showed above that a discarded price cannot belong to any pair *with the prices still between the indices*; it cannot belong to a pair with an already discarded price either, because that price was discarded for the same reason. So every pair that exists is found before either of its members is discarded. (With distinct prices, as here, moving both indices after a match is safe; the check-yourself of this section asks what changes when prices repeat.)

The cost. Sorting costs $O(n \log n)$ with a good sort (our program uses insertion sort, $O(n^2)$, because $n = 8$). The walk moves at least one index per step and stops when they meet, so it makes at most $n - 1$ steps: $O(n)$. The total is dominated by the sort: $O(n \log n)$, against $O(n^2)$ for all pairs. An alternative, also $O(n \log n)$, is to sort and then binary-search for `target - a[i]` for every `i`: n searches of $O(\log n)$ each.

4.13.4 The C++ solution

Listing 4.41 is the complete program.

Listing 4.41 `pair_sum.cpp`: sort the prices, then find every matching pair with a two-index walk.

```cpp
// pair_sum.cpp - Problem 4.11: which two Profitina Laundry services cost
// exactly the value of a voucher? Sort the prices, then walk inward with
// two indices.
#include <iostream>
using namespace std;

const int N = 8;

void show(const char* label, const int a[]) {
    cout << label;
    for (int i = 0; i < N; i++) {
        cout << ' ' << a[i];
    }
    cout << '\n';
}

void insertionSort(int a[]) {
    for (int i = 1; i < N; i++) {
        int key = a[i], j = i - 1;
        while (j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
}

// Prints every pair a[i] + a[j] == target (i < j) of the SORTED array a.
void pairsWithSum(const int a[], int target) {
    int i = 0, j = N - 1, steps = 0, found = 0;
    cout << "Voucher " << target << ":";
    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
        }
    }
    if (found == 0) {
```

Listing 4.41 (Continued) *pair_sum.cpp: sort the prices, then find every matching pair with a two-index walk.*

```cpp
        cout << " no pair;";
    }
    cout << " steps " << steps << " (every pair: " << N * (N - 1) / 2 << ")\n";
}

int main() {
    int prices[N] = {18000, 7000, 25000, 12000, 30000, 5000, 20000, 15000};
    show("Prices:", prices);
    insertionSort(prices);
    show("Sorted:", prices);
    pairsWithSum(prices, 37000);
    pairsWithSum(prices, 12000);
    pairsWithSum(prices, 11000);
    return 0;
}
```

```
Prices: 18000 7000 25000 12000 30000 5000 20000 15000
Sorted: 5000 7000 12000 15000 18000 20000 25000 30000
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)
```

The pairs agree with the hand search, and the step counts confirm the analysis: 5 steps for 37000 and 7 (that is, $n - 1$) for the other two, against 28 pairs for the brute-force method. For 12000 the walk had to move `j` all the way down from 30000 to 7000, one step at a time, before the sum became small enough; that is the worst case of the walk, and still linear.

4.13.5 The Java and Python solutions

Listing 4.42 and **Listing 4.43** give the same design in Java and in Python.

Listing 4.42 `PairSum.java`: the same sort and two-index walk in Java.

```java
// PairSum.java - Problem 4.11: which two Profitina Laundry services cost
// exactly the value of a voucher? Sort the prices, then walk inward with
// two indices.
public class PairSum {
    static void show(String label, int[] a) {
        StringBuilder sb = new StringBuilder(label);
        for (int x : a) {
            sb.append(' ').append(x);
        }
        System.out.println(sb);
    }

    static void insertionSort(int[] a) {
        for (int i = 1; i < a.length; i++) {
            int key = a[i], j = i - 1;
            while (j >= 0 && a[j] > key) {
                a[j + 1] = a[j];
                j--;
            }
            a[j + 1] = key;
        }
    }

    // Prints every pair a[i] + a[j] == target (i < j) of the SORTED array a.
    static void pairsWithSum(int[] a, int target) {
        int n = a.length, i = 0, j = n - 1, steps = 0, found = 0;
        StringBuilder sb = new StringBuilder("Voucher " + target + ":");
        while (i < j) {
            steps++;
            int sum = a[i] + a[j];
            if (sum == target) {
                sb.append(' ').append(a[i]).append(" + ").append(a[j])
                  .append(';');
                found++;
                i++;
                j--;
            } else if (sum < target) {
                i++;                    // too little: a dearer small item
            } else {
                j--;                    // too much: a cheaper large item
            }
        }
        if (found == 0) {
            sb.append(" no pair;");
        }
        sb.append(" steps ").append(steps).append(" (every pair: ")
```

Listing 4.42 (Continued) *PairSum.java: the same sort and two-index walk in Java.*

```
          .append(n * (n - 1) / 2).append(')');
        System.out.println(sb);
    }

    public static void main(String[] args) {
        int[] prices = {18000, 7000, 25000, 12000, 30000, 5000, 20000, 15000};
        show("Prices:", prices);
        insertionSort(prices);
        show("Sorted:", prices);
        pairsWithSum(prices, 37000);
        pairsWithSum(prices, 12000);
        pairsWithSum(prices, 11000);
    }
}
```

Listing 4.43 `pair_sum.py`: the same sort and two-index walk in Python.

```
# pair_sum.py - Problem 4.11: which two Profitina Laundry services cost exactly
# the value of a voucher? Sort the prices, then walk inward with two indices.


def insertion_sort(a):
    for i in range(1, len(a)):
        key, j = a[i], i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key


def pairs_with_sum(a, target):
    """Prints every pair a[i] + a[j] == target (i < j) of the SORTED list a."""
    n = len(a)
    i, j, steps, text = 0, n - 1, 0, f"Voucher {target}:"
    found = 0
    while i < j:
        steps += 1
        total = a[i] + a[j]
        if total == target:
            text += f" {a[i]} + {a[j]};"
            found += 1
            i += 1
            j -= 1
        elif total < target:
            i += 1                      # too little: take a dearer small item
        else:
            j -= 1                      # too much: take a cheaper large item
    if found == 0:
        text += " no pair;"
    print(f"{text} steps {steps} (every pair: {n * (n - 1) // 2})")


prices = [18000, 7000, 25000, 12000, 30000, 5000, 20000, 15000]
print("Prices:", *prices)
insertion_sort(prices)
print("Sorted:", *prices)
for voucher in (37000, 12000, 11000):
    pairs_with_sum(prices, voucher)
```

Both print exactly the output of the C++ program. The Java version builds each output line in a `StringBuilder`, which is the efficient way to join many small pieces of text in Java; the Python version collects the line in a string `text` and prints it once. Python offers a third design worth knowing: walk through the prices once, keep a *set* of the prices seen so far, and for each price *p*

ask whether `target - p` is in the set. Set membership takes $O(1)$ steps on average (Chapter 15), so the whole search is $O(n)$ with no sorting at all, at the price of the extra memory for the set.

4.13.6 A tempting wrong solution

The two-index walk is so short that it is tempting to forget the step that makes it work. **Listing 4.44** walks inward on the prices as they came.

Listing 4.44 `wrong_pair_sum.cpp`: the two-index walk on prices that were never sorted.

```
// wrong_pair_sum.cpp - a tempting wrong solution to Problem 4.11:
// the two-index walk on prices that were never sorted.
#include <iostream>
using namespace std;

int main() {
    const int N = 8;
    int prices[N] = {18000, 7000, 25000, 12000, 30000, 5000, 20000, 15000};
    int targets[] = {37000, 12000, 11000};
    for (int target : targets) {
        int i = 0, j = N - 1, found = 0;
        cout << "Voucher " << target << ":";
        while (i < j) {
            int sum = prices[i] + prices[j];
            if (sum == target) {
                cout << ' ' << prices[i] << " + " << prices[j] << ';';
                found++;
                i++;
                j--;
            } else if (sum < target) {
                i++;
            } else {
                j--;
            }
        }
        if (found == 0) {
            cout << " no pair;";
        }
        cout << '\n';
    }
    return 0;
}
```

```
Voucher 37000: no pair;
Voucher 12000: no pair;
Voucher 11000: no pair;
```

Three vouchers, three times `no pair`, although two of them have matches. On unsorted data, "the sum is too small" no longer implies that `a[i]` is too cheap for *every* partner, because a dearer partner may be hiding anywhere in the array. The argument of Section 4.13.3 rested entirely on the sorted order, and without it the walk discards prices that belong to pairs. Like binary search in Problem 4.3, the two-index walk has a precondition that gives no error message when it is broken.

> **CHECK YOURSELF.** A ninth service costing 18500 is added, and a customer brings a voucher of 36000. Trace the walk on the sorted nine prices, and say which pairs it reports.
>
> *Answer.* Sorted: 5000, 7000, 12000, 15000, 18000, 18500, 20000, 25000, 30000. In thousands: $5 + 30 = 35 < 36$, `i++`; $7 + 30 = 37 > 36$, `j--`; $7 + 25 = 32 < 36$, `i++`; $12 + 25 = 37 > 36$, `j--`; $12 + 20 = 32 < 36$, `i++`; $15 + 20 = 35 < 36$, `i++`; $18 + 20 = 38 > 36$, `j--`; $18 + 18.5 = 36.5 > 36$, `j--`; now `i = j`, stop. No pair, after 8 steps. (Indeed no two different prices add up to 36000; 18000 + 18000 would, but there is only one service at 18000.)

4.14 The Problems Side by Side

Eleven problems can blur into one another. **Table 4.5** puts them in a row with the idea each one teaches best, its cost, and the trap its tempting wrong solution fell into. If you remember only the right-hand column, you will avoid most of the bugs that beginners write in their first data structures course.

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

Three patterns stand out. First, *preconditions*: binary search (Problems 4.3 and 4.6), interpolation search (4.7), the merge (4.9) and the two-index walk (4.11) are all correct only on sorted data, and none of them reports an error when the data is not sorted; they simply answer wrongly. Second, *boundaries*: an index that runs one step too far (4.2 and 4.8), a range that cannot reach past the end (4.6), and a tail that is never copied (4.9) are all the same kind of mistake at the edge of an array. Third, *the edge examples found every bug*: a mixed-case word, the string `xyzzyx`, the last book on the shelf, a key outside the range, two adjacent equal values. Testing the edges is not an extra; it *is* testing.

4.15 Profitina in Practice

Profitina (app.profitina.com) is an AI-visibility platform: it measures how ready a brand's website is to be cited by AI answer engines, explains every signal it measures, and helps the owner fix what it finds. The problems of this chapter are small, but their shapes appear in such a product at full size.

Sorting with a tie-break. Profitina's community ranks its members on a PROFIT leaderboard, where PROFIT is the platform's utility coin, earned by improving AI responses. A ranking is Problem 4.4 in another costume: a primary key (the amount earned) and a secondary key that decides ties, applied with a stable sort or combined into one comparison rule, so that two

members with the same amount always appear in the same order and the list does not flicker between two page loads.

Searching in time. The historical analytics dashboard shows a weekly score trend for each brand and exports it as CSV. A history of scans is an array that is sorted by time *by construction*, because scans are appended in the order they happen. "All scans between 1 July and 30 September" is then exactly the check-yourself of Problem 4.6: two boundary searches and a subtraction, $O(\log n)$ however long the history grows, instead of a pass over every scan ever made.

Merging sorted streams. The dashboard is designed as a cockpit: a single request gathers the user's notifications of many kinds (replies, likes, reposts, direct messages, pins, feedback status, PROFIT purchases) into one list. When each kind already arrives sorted by time, the natural way to combine them into one time-ordered list is the merge of Problem 4.9, generalized from two lists to several, rather than a fresh sort of everything.

Counting instead of comparing. The channels of the community show "N new" counters, and those counters count only top-level posts. Counting per category, the idea of Problem 4.2, is how such numbers stay cheap: one pass over the new items, one counter per channel, no comparison of every item with every other.

The runnable practice for this book stays with Profitina Laundry, the teaching case study. Problem 4.11 is its first appearance here: eight services, their prices in rupiah, and a voucher, solved with a sort and a two-index walk. Later chapters will give the laundry a queue of orders (Chapter 5), a list of customers (Chapter 7) and a price lookup table (Chapter 15).

4.16 Common Mistakes, and How to Avoid Them

Using binary search on data that is not sorted. It gives no error, only wrong answers (Problem 4.3). The same holds for interpolation search, merging and the two-index walk. Before you use one of them, point at the line of your program that guarantees the order.

Comparing strings by character code when you meant dictionary order. In ASCII every capital letter comes before every small letter, so plain `<` puts `Cherry` before `apple` (Problem 4.1). Decide what "alphabetical" means and write the comparison rule yourself.

Misreading a short statement. "First duplicate character" has two readings, and only the examples decide between them (Problem 4.2). Check your reading against *every* example, not only the first.

Letting a loop run one step too far. `i <= n` where `i < n` was meant, or an inner bubble-sort loop that reads `a[j + 1]` at `j = n - 1` (Problems 4.2 and 4.8). Trace the *last* pass of every loop on paper. In C++, run with `-fsanitize=address,undefined` while developing.

Starting a boundary search on the range $[0, n-1]$**.** A lower or upper bound can be n, one past the end (Problem 4.6). Start with `hi = n`.

Computing the middle as `(lo + hi) / 2` on huge arrays. In C++ and Java the sum can overflow; write `lo + (hi - lo) / 2` [9].

Applying the keys of a multi-pass sort in the wrong order. With stable sorts, sort by the least important key first and the most important last; or put all keys into one comparison rule (Problem 4.4).

Forgetting the element size in an address. An index counts elements, an address counts bytes; multiply by the size, and in two dimensions by the number of *columns* (Problem 4.5).

Dividing before multiplying in integer arithmetic. In Equation (4.6), `(key - lo_value) / (hi_value - lo_value)` is almost always 0 in integer division; multiply first (Problem 4.7).

Forgetting the tail of a merge. When one array runs out, the rest of the other must still be copied; check that the output has $n + m$ elements (Problem 4.9).

Advancing the index after deleting by shifting. The element that moves into the current box is skipped (Problem 4.10). Use a read index and a write index.

Comparing Java strings with `==`. It compares references, not letters; use `equals` (Problem 4.3).

Testing only the examples in the statement. Every tempting wrong solution of this chapter passed at least one of the given examples. Add the empty input, one element, the first and the last position, a missing key, ties, and adjacent duplicates [12].

4.17 Chapter Summary

This chapter consolidated Chapters 1 to 3 by solving problems rather than introducing new structures.

A method. Section 4.2 set out eight steps: restate the problem; work examples by hand; choose the structure and the algorithm; trace the design on paper; estimate the cost; write the code, C++ first; run and compare with the hand trace; and look back for edges, traps and variants. **Table 4.2**, Equation (4.1) and Equation (4.2) turned step 5 into arithmetic: $\frac{n(n-1)}{2}$ comparisons for all pairs, and at most $\lfloor \log_2 n \rfloor + 1$ probes for binary search.

Four classic problems. Words were sorted into dictionary order by an insertion sort whose only new part was the comparison rule (Problem 4.1). The first repeated character was found in $O(n)$ with one counter per possible character, after the examples had decided which of two readings the statement meant (Problem 4.2). Money was looked up by name with sequential search over an array of records, and binary search on the unsorted data was shown to fail silently (Problem 4.3). Strings were sorted by two keys in one stable sort, and two stable sorts in the wrong order were shown to put the minor key in charge (Problem 4.4).

Seven new problems. Array addresses by Equation (4.3) and Equation (4.4), confirmed against real memory (Problem 4.5); counting a key with two boundary searches and Equation (4.5) (Problem 4.6); interpolation search by Equation (4.6), with the guard that keeps its probes inside the array (Problem 4.7); the three elementary sorts traced pass by pass, with swaps and shifts equal to the number of inversions (Problem 4.8); the merge of two sorted arrays in $O(n + m)$ (Problem 4.9); deletion in place with a read index and a write index (Problem 4.10); and pairs with a given sum by sorting and walking inward (Problem 4.11).

Looking ahead. Chapter 5 builds the stack and the queue on exactly the array-with-a-size idea of Problem 4.10, and Chapter 6 opens up the pointers and references that Problems 4.3 and 4.5 used without comment.

4.18 Exercises

4.1 Trace the insertion sort of Problem 4.1 on the five words `delta Alpha charlie alpha Bravo`, writing the array after every step. What is the sorted result?

4.2 For the strings `abcba` and `programming`, give the answer of the correct program of Problem 4.2 and the answer of the tempting wrong solution. For which of the two strings do they differ, and why?

4.3 How many name comparisons does the sequential search of Problem 4.3 make for each of the five queries in `names.txt`, and in total?

4.4 Sort the strings `wow!!`, `ok`, `yes!`, `no!`, `hm` (in this input order) by the importance rule of Problem 4.4. Which two strings tie on both keys, and in which order do they appear?

4.5 An array `int C[4][6]` of 4-byte integers starts at `0x3000`. (a) What is the address of `C[3][2]`? (b) Which element starts at address `0x3034`?

4.6 Using the lower and upper bounds of Problem 4.6 on the shelf of years, how many books were published from 2005 to 2019 inclusive? Name the two calls you make and their results.

4.7 Trace interpolation search and binary search for the key 63 on the eight book codes of Problem 4.7. How many probes does each make?

4.8 Trace selection sort on 4, 2, 5, 1, 3: write the array after every pass, and count comparisons and swaps. How many swaps would bubble sort make on the same array, and why can you answer without tracing it?

4.9 Merge 1, 4, 7 with 2, 3, 8, 9 by **Pseudocode 4.9**. How many comparisons are made, and which elements are copied by the tail loop?

4.10 Use **Pseudocode 4.10**, with the test “keep `a[r]` when it is odd”, to delete all even numbers from 2, 7, 4, 4, 9, 6, 1. Give the values of `w` after each pass and the result.

4.11 On the sorted laundry prices of Problem 4.11, trace the two-index walk for a voucher of 45000. Which pairs are reported, and after how many steps?

4.12 Apply the eight steps of Section 4.2 to this problem, writing out each step: “Given an array of integers, find the second largest *distinct* value, or report that there is none.”

4.19 Solutions to the Exercises

4.1 In small letters the words are delta, alpha, charlie, alpha, bravo. Step $i = 1$: `Alpha` comes before `delta`, giving `Alpha delta charlie alpha Bravo`. Step 2: `charlie` goes between them: `Alpha charlie delta alpha Bravo`. Step 3: `alpha` passes `delta` and `charlie`; against `Alpha` the small-letter copies tie, and `alpha` < `Alpha` is false in code order (97 > 65), so it stops: `Alpha alpha charlie delta Bravo`. Step 4: `Bravo` passes `delta` and `charlie` and stops after `alpha`: `Alpha alpha Bravo charlie delta`. That is the result.

4.2 `abcba`: counts a 2, b 2, c 1; the correct program answers `a` (the leftmost character with count at least 2), while the wrong one sees a, b, c and then b again and answers `b`. `programming`: counts p 1, r 2, o 1, g 2, a 1, m 2, i 1, n 1; the correct answer is `r` (index 1); the wrong program sees p, r, o, g and then r again, and also answers `r`. They differ for `abcba` because there the character whose first appearance is leftmost (`a`) repeats *later* than another character (`b`) does.

4.3 `Pujo` is found at index 3 after 4 comparisons; `Komeng` is absent, 5 comparisons; `Lestari` is at index 4, 5 comparisons; `pujo` is absent, 5; `Puspita` is at index 0, 1. The total is $4 + 5 + 5 + 5 + 1 = 20$ comparisons.

4.4 Keys (marks, length): `wow!!` (2, 5), `ok` (0, 2), `yes!` (1, 4), `no!` (1, 3), `hm` (0, 2). The order is `wow!!`, then the one-mark strings shorter first, `no!`, `yes!`, then the zero-mark strings. `ok` and `hm` tie on both keys, and the stable sort keeps their input order: `ok`, `hm`.

4.5 (a) By Equation (4.4), $k = 3 \cdot 6 + 2 = 20$ and the offset is $20 \cdot 4 = 80$ bytes, `0x50`, so the address is `0x3050`. (b) The offset is 0x34 = 52 bytes, that is $52/4 = 13$ elements; $13 = 2 \cdot 6 + 1$, so the element is `C[2][1]`.

4.6 The books with $2005 \leq year \leq 2019$ lie from `lower(2005)` up to, not including, `upper(2019)`. `lower(2005)` = 4 (the 2005 at index 4) and `upper(2019)` = 8 (the first year greater than 2019 is the 2023 at index 8), so the count is $8 - 4 = 4$: 2005, 2010, 2010 and 2019.

4.7 Interpolation: $0 + \lfloor 38 \cdot 7/71 \rfloor = \lfloor 3.75 \rfloor = 3$, and $a[3] = 56 < 63$, so `low` = 4; then $4 + \lfloor 0 \cdot 3/33 \rfloor = 4$, and $a[4] = 63$: found after 2 probes. Binary search: `mid` = 3 (56 < 63, `low` = 4), `mid` = 5 (72 > 63, `high` = 4), `mid` = 4 (found): 3 probes.

4.8 Pass 1: the smallest of all five is 1 at index 3; swap with index 0: 1, 2, 5, 4, 3. Pass 2: the smallest of 2, 5, 4, 3 is 2, already in place, no swap. Pass 3: the smallest of 5, 4, 3 is 3 at index 4; swap: 1, 2, 3, 4, 5. Pass 4: the smallest of 4, 5 is 4, in place. Comparisons $4 + 3 + 2 + 1 = 10$, swaps 2. Bubble sort makes one swap per inversion; the inversions are (4,2), (4,1), (4,3), (2,1), (5,1), (5,3), so it makes 6 swaps.

4.9 1 vs 2, take 1; 4 vs 2, take 2; 4 vs 3, take 3; 4 vs 8, take 4; 7 vs 8, take 7. The first array is exhausted after 5 comparisons, and the tail loop copies 8 and 9. The result is 1, 2, 3, 4, 7, 8, 9.

4.10 `r` = 0: 2 is even, skip, `w` = 0. `r` = 1: 7 kept, `w` = 1. `r` = 2 and 3: 4 and 4 skipped, `w` = 1. `r` = 4: 9 kept, `w` = 2. `r` = 5: 6 skipped. `r` = 6: 1 kept, `w` = 3. The result is 7, 9, 1, new size 3; the adjacent 4s cause no trouble, because nothing is shifted.

4.11 In thousands: $5 + 30 = 35 < 45$, `i++`; $7 + 30 = 37$, `i++`; $12 + 30 = 42$, `i++`; $15 + 30 = 45$, pair, both move; $18 + 25 = 43$, `i++`; $20 + 25 = 45$, pair, both move; now `i` > `j`, stop. Two pairs, 15000 + 30000 and 20000 + 25000, after 6 steps.

4.12 Step 1: input an array of n integers; output the largest value that is strictly smaller than the maximum, or "none". Step 2: 4, 9, 2, 9, 7 gives 7 (the second 9 is not distinct); 5, 5, 5 gives none; a single element gives none; $-3, -8$ gives -8. Step 3: an array and one pass, keeping two variables, `first` (largest so far) and `second` (largest so far that is smaller than `first`), with a flag for "not yet seen". Step 4: trace 4, 9, 2, 9, 7: first 4; first 9, second 4; 2 changes nothing; 9 equals first, nothing; 7 is between them, second 7. Step 5: one pass, $O(n)$ time, $O(1)$ space; sorting first would cost $O(n \log n)$. Step 6: write it in C++, then Java and Python. Step 7: compare with the traces of step 2. Step 8: the tempting wrong solution sorts and returns `a[n-2]`, which answers 9 for 4, 9, 2, 9, 7, because it forgets *distinct*; a variant asks for the k-th largest distinct value.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[3] G. Pólya, *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed. Princeton, NJ, USA: Princeton University Press, 1957.

[4] American National Standards Institute, *Coded Character Set — 7-Bit American National Standard Code for Information Interchange*, ANSI X3.4-1986, 1986.

[5] cppreference.com, "Algorithms library" (`std::sort`, `std::stable_sort`, `std::lower_bound`, `std::upper_bound`, `std::equal_range`, `std::merge`, `std::remove`, `std::unique`). Accessed: Sep. 24, 2026. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm

[6] Oracle, *Java Platform, Standard Edition 25 API Specification*, class `java.util.Arrays`, 2025. Accessed: Sep. 24, 2026. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html

[7] Python Software Foundation, "Sorting Techniques," "bisect — Array bisection algorithm," "heapq — Heap queue algorithm," and "collections — Container datatypes," Python 3.14 documentation. Accessed: Sep. 24, 2026. [Online]. Available: https://docs.python.org/3/

[8] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[9] J. Bloch, "Extra, Extra — Read All About It: Nearly All Binary Searches and Mergesorts are Broken," Google Research Blog, Jun. 2006. [Online]. Available: https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/

[10] Y. Perl, A. Itai, and H. Avni, "Interpolation search—a log log N search," *Communications of the ACM*, vol. 21, no. 7, pp. 550–553, 1978.

[11] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

[12] J. Bentley, *Programming Pearls*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2000.

Chapter 5
Stacks and Queues

By the end of this chapter you will be able to explain, with a picture for every step, what a stack and a queue are and why each one deliberately forbids you to touch most of its own data; implement both on a plain array in C++, Java and Python, with every operation running in constant time; recognize the four classic jobs a stack does in real software — matching brackets, undoing edits, translating arithmetic from the way people write it into the way machines evaluate it, and keeping track of which function is running — and write each of them yourself; diagnose the "drifting" bug that makes a naive array queue refuse new data while half its slots stand empty, and cure it with modular arithmetic; choose between the two classic ways of telling a full circular queue from an empty one; extend the queue into a double-ended queue and preview the priority queue; and use the ready-made versions in `std::stack`, `std::queue`, `std::deque`, Java's `ArrayDeque` and Python's `list`, `collections.deque` and `queue`, knowing which one to pick and why. The chapter opens with a very old puzzle about monks and golden disks, because that puzzle is a stack in disguise.

5.1 Where We Left Off

Chapters 1 to 3 treated an array as a room with an open door. Chapter 1 stored the Pustaka Ceria library catalog in contiguous memory and showed that any element $A[i]$ can be reached in one step, because its address is the base address plus i times the element size. Chapter 2 searched that array — sequentially when it was unordered, by halving when it was sorted — and Chapter 3 sorted it, moving elements to whatever position the algorithm liked. Chapter 4 then worked full problems of exactly that kind. In all four chapters the algorithm had total freedom: read any element, write any element, in any order.

This chapter takes that freedom away on purpose.

5.1.1 From free access to disciplined access

Picture the book-return cart at the front desk of Pustaka Ceria. Returned books are dropped on top of the pile; a librarian reshelving them always lifts the top book first. Nobody pulls a book out of the middle — the pile would topple. Now picture the new-arrivals desk in the back office, where freshly unpacked books wait in a line to be cataloged. The book that arrived first is cataloged first; a new box joins at the end of the line. Nobody jumps the line — it would not be fair to the book that has waited longest.

Both of these are collections of books, and both could be stored in an array. But in both, only **one** position is ever touched: the top of the pile in the first case; the two ends of the line in the second. Everything else is off limits. That restriction sounds like a loss. It is actually the source of three gains:

- **Simplicity.** An operation that may touch only one known position needs no search and no shifting. Every operation in this chapter's basic structures runs in $O(1)$ time — constant time, independent of how many elements are stored.

- **Correctness by construction.** If the only way to remove a book is from the top, no piece of code can accidentally reshelve the wrong book. The rule is enforced by the structure, not by the programmer's memory.
- **A match with the problem.** Many real problems *are* last-in-first-out or first-in-first-out by nature: undoing edits, matching brackets, serving customers, printing documents. When the data structure has the same shape as the problem, the program almost writes itself.

5.1.2 Abstract data types

Before we build anything, we need one piece of vocabulary.

> **DEFINITION.** An *abstract data type* (ADT) is a description of 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 the values so that the operations can be carried out.

The distinction matters. "Stack" is an ADT: it promises `push`, `pop`, `peek` and `isEmpty`, and it promises that `pop` returns the most recently pushed value that has not yet been removed. It says nothing about arrays or pointers. An array with a `top` index is one data structure that implements the stack ADT; a chain of linked nodes is another; a Python `list` is a third. Code that uses a stack only through its operations works unchanged with any of them. Cormen et al. present stacks and queues exactly this way, as *dynamic sets* whose deletion operation is prespecified [1], and Knuth's classic treatment of "linear lists" names the same three restricted forms we study here: the stack, the queue and the deque [2].

Figure 5.1 shows the two disciplines side by side. In the stack on the left, items enter and leave through the same opening, the top, so the last one in is the first one out: **LIFO**. In the queue on the right, items enter at the rear and leave from the front, so the first one in is the first one out: **FIFO**.

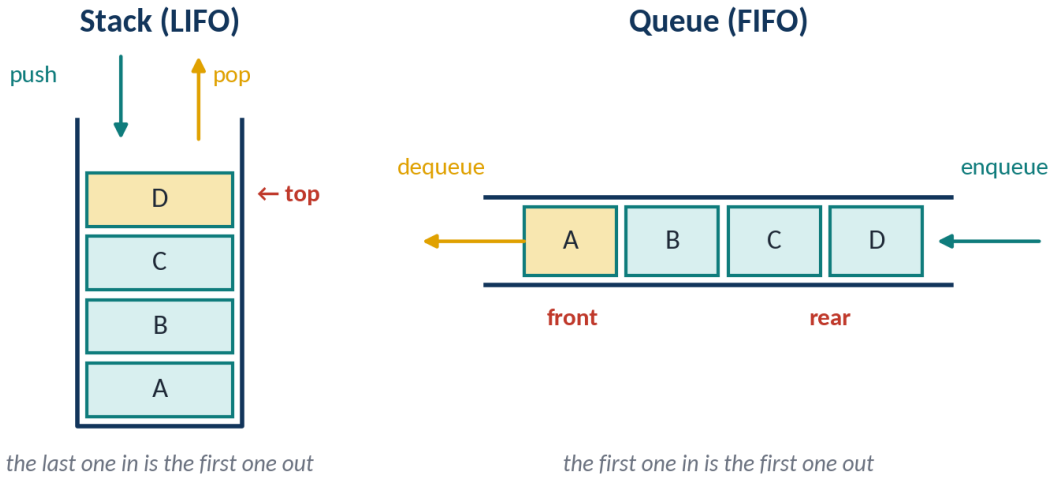


Figure 5.1 The two disciplines of this chapter. A stack adds and removes at the same end (LIFO); a queue adds at the rear and removes at the front (FIFO).

5.2 A Story to Begin: the Towers of Hanoi

In 1883 the French mathematician Édouard Lucas sold a puzzle in Paris under the pen name "Professeur N. Claus de Siam" — an anagram of "Lucas d'Amiens". The box came with a legend [3]. In a temple in the city of Benares, the story went, there are three diamond needles, and on one of them God placed sixty-four disks of pure gold, the largest at the bottom and each disk smaller than the one beneath it. Day and night the temple priests move the disks from the first needle to

the third, following two unbreakable rules. When they finish, the temple will crumble into dust and the world will end.

The rules are the whole puzzle:

1. Move **one disk at a time**, and only the disk that lies **on top** of a needle.
2. Never place a disk on top of a **smaller** disk.

Look at rule 1 again. On each needle — we shall say *peg* — the only disk you may take is the one on top, and the only place you may put a disk is on top. The disk you placed last on a peg is the first one you can take off it. Each peg is a stack. The Towers of Hanoi is a puzzle about three stacks and one extra rule about sizes.

5.2.1 Three disks, by hand

Let us name the pegs A (where the disks start), B (the helper) and C (the goal), and number the disks 1 (smallest) to 3 (largest). **Figure 5.2** shows the shortest solution, one panel per state.

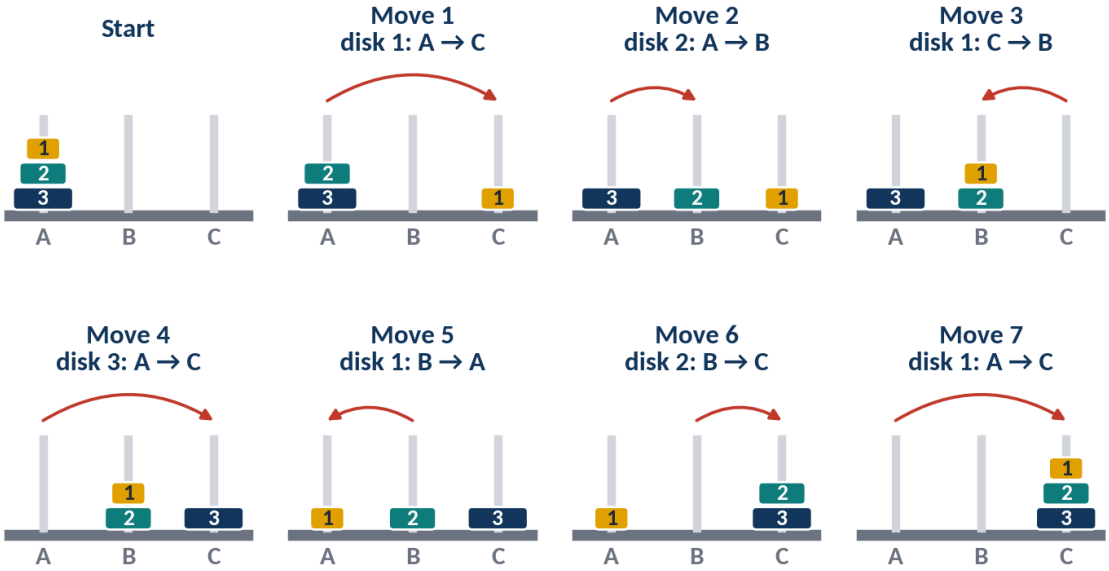


Figure 5.2 The seven moves that transfer three disks from peg A to peg C. The red arrow marks the move that produced each state; at every step only a top disk moves, and it never lands on a smaller disk.

Walk through it slowly, because the pattern is the point.

- **Moves 1–3** move the two small disks out of the way. Disk 1 goes to C, disk 2 goes to B, and disk 1 climbs on top of disk 2. After move 3, peg B holds a correct little tower of two disks, and disk 3 on A is finally uncovered.
- **Move 4** is the only move in the whole solution that touches disk 3: A to C. It can happen only because C is empty and disk 3 is on top of A.
- **Moves 5–7** repeat the first three moves in a different frame: the two-disk tower on B is carried onto disk 3 on C, using A as the helper.

So solving three disks consists of solving two disks twice, with one move of the big disk in between. The same argument works for any number of disks, and it gives us a count.

5.2.2 How many moves?

Write $M(n)$ for the smallest number of moves that transfers n disks from one peg to another. The largest disk must move at least once. Before it can move, every other disk — all $n-1$ of them — must be off its peg *and* off the target peg, which means they are all stacked on the third peg: that

took at least $M(n-1)$ moves. After the largest disk lands, the $n-1$ disks must be carried onto it, which takes at least $M(n-1)$ moves again. And the strategy of **Figure 5.2** achieves exactly that. So

$$M(n) = 2M(n-1) + 1, \qquad M(1) = 1. \tag{5.1}$$

Equation (5.1) is a *recurrence*: it defines each value in terms of the previous one. Unroll it a few times to see the pattern:

$$M(1) = 1, \quad M(2) = 2 \cdot 1 + 1 = 3, \quad M(3) = 2 \cdot 3 + 1 = 7, \quad M(4) = 2 \cdot 7 + 1 = 15, \quad M(5) = 31.$$

Every value is one less than a power of two, and indeed

$$M(n) = 2^n - 1. \tag{5.2}$$

To check Equation (5.2), substitute it into the right-hand side of Equation (5.1): $2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$, which is the left-hand side, and $2^1 - 1 = 1$ matches the starting value. (Chapter 10 proves results like this formally by induction and solves the puzzle with a recursive function; here we only need the number.)

Now the legend. With $n = 64$ disks the priests need $2^{64} - 1 = 18{,}446{,}744{,}073{,}709{,}551{,}615$ moves. At one move per second, day and night, that is

$$\frac{2^{64} - 1}{60 \cdot 60 \cdot 24 \cdot 365.25} \approx 5.85 \times 10^{11} \text{ years}, \tag{5.3}$$

about 585 billion years by Equation (5.3), some forty times the present age of the universe. The world is safe for a while. The lesson for a programmer is sharper: an algorithm whose cost doubles with every extra unit of input becomes hopeless long before the input looks large.

5.2.3 Solving Hanoi with three stacks and no recursion

Because each peg is a stack, we can write a program that holds the three pegs as three stacks and moves disks only with `push` and `pop`. Chapter 10 will solve the puzzle with recursion, which is the natural way to express "solve $n-1$, move one, solve $n-1$". But there is also a beautiful non-recursive method, and it uses nothing but the stack operations:

- On every **odd-numbered** move, move disk 1 one peg along a fixed cycle: A → C → B → A → … when n is odd, and A → B → C → A → … when n is even.
- On every **even-numbered** move, make the only legal move that does not involve disk 1. Between the two pegs that do not hold disk 1 there is exactly one legal move: the smaller of their two top disks goes onto the other peg (or onto it, if one peg is empty).

Why is there only ever one choice on the even moves? Disk 1 is the smallest, so it sits on top of its peg; the other two pegs have tops that are both larger than 1, and between two different top disks only the smaller may move onto the larger. **Listing 5.1** implements the method. Each peg is a `vector<int>` used strictly as a stack — `push_back` to push, `back` to peek, `pop_back` to pop — and `moveDisk` refuses any move that would break rule 2.

Listing 5.1 `01_hanoi_stacks.cpp`: the Towers of Hanoi solved with three stacks and a loop, then the move count for 1 to 10 disks.

```cpp
// Towers of Hanoi solved WITHOUT recursion: three pegs, each one a stack.
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct Peg {
    char name;
    vector<int> disks;                          // back() is the top of the peg
    bool isEmpty() const { return disks.empty(); }
    int  peek()    const { return disks.back(); }
};

string show(const Peg& p) {                     // bottom -> top, e.g. "[3 2 1]"
    string s = "[";
    for (size_t i = 0; i < p.disks.size(); ++i) {
        if (i > 0) s += " ";
        s += to_string(p.disks[i]);
    }
    return s + "]";
}

// 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;
}

// The only legal move between two pegs (the smaller top disk goes on the other).
void legalMoveBetween(Peg& x, Peg& y) {
    if (x.isEmpty())                moveDisk(y, x);
    else if (y.isEmpty())           moveDisk(x, y);
    else if (x.peek() < y.peek())   moveDisk(x, y);
    else                            moveDisk(y, x);
}
```

Listing 5.1 (Continued) *01_hanoi_stacks.cpp: the Towers of Hanoi solved with three stacks and a loop, then the move count for 1 to 10 disks.*

```
long long solve(int n, bool verbose) {
    Peg peg[3] = {{'A', {}}, {'B', {}}, {'C', {}}};
    for (int d = n; d >= 1; --d) peg[0].disks.push_back(d);
    // Disk 1 cycles A->C->B->A when n is odd, A->B->C->A when n is even.
    int step = (n % 2 == 1) ? 2 : 1;            // index distance of disk 1's moves
    int small = 0;                              // the peg that holds disk 1
    long long moves = 0;
    auto report = [&](const string& what) {
        if (!verbose) return;
        cout << left << setw(23) << what << "A: " << setw(9) << show(peg[0])
             << "B: " << setw(9) << show(peg[1]) << "C: " << show(peg[2]) << "\n";
    };
    report("start");
    while (peg[2].disks.size() != static_cast<size_t>(n)) {
        int next = (small + step) % 3;          // move disk 1 one peg along its cycle
        moveDisk(peg[small], peg[next]);
        ++moves;
        report("move " + to_string(moves) + ": disk 1 " + peg[small].name + " -> " +
peg[next].name);
        small = next;
        if (peg[2].disks.size() == static_cast<size_t>(n)) break;
        int o1 = (small + 1) % 3, o2 = (small + 2) % 3;   // the two pegs without disk 1
        Peg& x = peg[o1];
        Peg& y = peg[o2];
        char fromName = (x.isEmpty() || (!y.isEmpty() && y.peek() < x.peek())) ? y.name :
x.name;
        char toName   = (fromName == x.name) ? y.name : x.name;
        int  disk     = (fromName == x.name) ? x.peek() : y.peek();
        legalMoveBetween(x, y);
        ++moves;
        report("move " + to_string(moves) + ": disk " + to_string(disk) + " " + fromName +
" -> " + toName);
    }
    return moves;
}

int main() {
    cout << "Towers of Hanoi, 3 disks; each peg printed bottom -> top\n";
    long long m = solve(3, true);
    cout << "moves made: " << m << "\n\n";
    cout << "disks   moves   2^n - 1\n" << right;
    for (int n = 1; n <= 10; ++n)
        cout << setw(5) << n << setw(8) << solve(n, false) << setw(10) << (1LL << n) - 1 <<
"\n";
    return 0;
}
```

Compiled with `g++ -std=c++17 -O2 -Wall -Wextra` and run, it prints:

```
Towers of Hanoi, 3 disks; each peg printed bottom -> top
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]
moves made: 7

disks   moves   2^n - 1
    1       1         1
    2       3         3
    3       7         7
    4      15        15
    5      31        31
    6      63        63
    7     127       127
    8     255       255
    9     511       511
   10    1023      1023
```

The first block reproduces **Figure 5.2** move for move, printing each peg from bottom to top. The table confirms Equation (5.2) for every n from 1 to 10: the loop, which never looks further than the top of any peg, makes exactly $2^n - 1$ moves. The Java and Python versions in the Code folder print the same lines.

Two things are worth noticing in the code. First, `moveDisk` checks `to.peek() < from.peek()` before pushing: the stack interface gives us exactly the information rule 2 needs — the top disk — and nothing else. Second, nobody ever indexes into the middle of a peg. That is what makes the three vectors stacks rather than arrays.

5.3 The Stack as an Abstract Data Type

We now make the idea precise.

> **DEFINITION.** A *stack* is a collection of elements in which insertion and removal both happen at one end, called the *top*. The element removed is always the one inserted most recently among those still present. This discipline is called *last in, first out* (LIFO).

The everyday examples are everywhere: a pile of plates in a canteen, a stack of books on a desk, a tube of badminton shuttlecocks, the tray of trays at a food court. In each one you add on top and take from the top.

5.3.1 The operations

Table 5.1 lists the operations of the stack ADT. The names `push`, `pop` and `peek` are universal; some libraries say `top` instead of `peek`.

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

Two of the preconditions have names, because breaking them is the most common stack bug of all:

- **Stack overflow**: calling `push` on a full bounded stack.
- **Stack underflow**: calling `pop` or `peek` on an empty stack.

An implementation must decide what happens in each case. The choices are to refuse politely (return `false`), to signal an error (throw an exception), or to grow (allocate more room, Section 5.5). Leaving the behavior undefined — reading `data[-1]` — is not a choice; it is a bug.

5.3.2 The operations in pseudocode

Pseudocode 5.1 writes the operations for a stack stored in an array $S[0..N-1]$ with an integer top that holds the index of the top element. The convention is that $top = -1$ means empty, so the number of elements is

$$\text{size} = top + 1. \tag{5.4}$$

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

PEEK(S)
12.  if STACK-EMPTY(S)
13.      error "underflow"
14.  return S[S.top]
```

In plain words: `PUSH` moves the top marker up one slot and writes there; `POP` reads the slot under the marker and moves the marker down. The *order* inside each procedure matters. If `PUSH` wrote first and then incremented, the first push would write to $S[-1]$. If `POP` decremented first and then read, it would return the element *under* the top. In C++ and Java both steps fit in one expression — `data[++top] = x` and `return data[top--]` — and the position of `++` and `--` encodes the order exactly: pre-increment happens before the write, post-decrement happens after the read.

Some books use the other convention: top is the number of elements, so $top = 0$ means empty and the top element is $S[top - 1]$. Both work. What does not work is mixing them — initializing top to 0 and then testing emptiness with $top = -1$. Equation (5.4) is the one-line reminder of which convention this book uses.

5.4 An Array-Based Stack

Let us build the stack the way the ADT describes it: an array of fixed capacity plus one integer. Pustaka Ceria's return cart holds four books, so our stack has capacity 4.

5.4.1 Watching the array

Before reading any code, watch the array. **Figure 5.3** follows eight operations on a stack of capacity 4, with the book titles abbreviated: LP for *Laskar Pelangi*, FK for *Filosofi Kopi*, BM for *Bumi Manusia*, N5M for *Negeri 5 Menara* and RDP for *Ronggeng Dukuh Paruk*. Index 0 is at the bottom of each column.

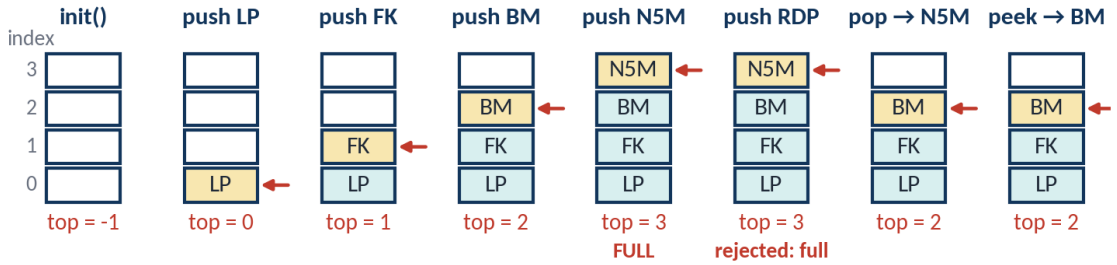


Figure 5.3 An array stack of capacity 4 through eight operations. The amber cell is the top element and the red arrow is the `top` index; after the fourth push the stack is full and the fifth push is rejected.

Read the columns from left to right:

1. `init()` sets $top = -1$. The array has four slots, but none is in use. (The slots may contain leftover garbage; it does not matter, because nothing below or at *top* is garbage and nothing above it is ever read.)
2. Each of the next four pushes moves *top* up one slot and writes there: *top* becomes 0,1,2,3. After the fourth push, $top = 3 = N - 1$, so `isFull()` is true.
3. The fifth push, of RDP, is refused. Nothing changes — not the array, not *top*.
4. `pop` returns the element under the marker, N5M, and moves the marker to 2. Note what happens in the array: *nothing*. N5M is still physically in slot 3 until something overwrites it. It is gone from the stack only because *top* no longer covers it.
5. `peek` returns BM and changes nothing at all.

Point 4 deserves a second look, because it is the most surprising thing about array stacks for beginners: **removing an element does not erase it**. The stack's contents are defined as "slots 0 to *top*", so moving *top* is enough. This is also why `clear()` is a single assignment, $top \leftarrow -1$, in the C++ version below.

5.4.2 The stack in C++

Listing 5.2 implements the stack as a class template, so that the same code stores titles today and integers, characters or `Book` records tomorrow. The private members are exactly the ones in **Pseudocode 5.1**: an array `data_`, its capacity, and `top_`.

Listing 5.2 `02_array_stack.cpp`: an array-based stack template in C++, used as Pustaka Ceria's book return cart.

```
// An array-based stack: Pustaka Ceria's book return cart.
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

template <typename T>
class ArrayStack {
public:
    explicit ArrayStack(int capacity) : data_(new T[capacity]), capacity_(capacity), top_(-
1) {}
    ~ArrayStack() { delete[] data_; }
    ArrayStack(const ArrayStack&) = delete;             // no accidental copies of the
array
    ArrayStack& operator=(const ArrayStack&) = delete;

    bool isEmpty() const { return top_ == -1; }
    bool isFull()  const { return top_ == capacity_ - 1; }
    int  size()    const { return top_ + 1; }
    int  topIndex() const { return top_; }

    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
    const T& at(int i) const { return data_[i]; }       // for printing only

private:
    T*  data_;
    int capacity_;
    int top_;                                           // index of the top element, -1 if
empty
};
```

Listing 5.2 (Continued) *02_array_stack.cpp: an array-based stack template in C++, used as Pustaka Ceria's book return cart.*

```
void printCart(const ArrayStack<string>& s) {
    cout << "cart, top to bottom:";
    for (int i = s.topIndex(); i >= 0; --i) cout << (i == s.topIndex() ? " " : " | ") <<
s.at(i);
    cout << "\n";
}

int main() {
    cout << "=== Pustaka Ceria: book return cart (array stack, capacity 4) ===\n";
    ArrayStack<string> cart(4);
    const string returned[] = {"Laskar Pelangi", "Filosofi Kopi", "Bumi Manusia",
                               "Negeri 5 Menara", "Ronggeng Dukuh Paruk"};
    for (const string& title : returned) {
        if (cart.push(title))
            cout << "push " << title << " -> ok, size " << cart.size()
                 << ", top index " << cart.topIndex() << "\n";
        else
            cout << "push " << title << " -> REJECTED, the cart is full\n";
    }
    printCart(cart);
    cout << "peek -> " << cart.peek() << " (size is still " << cart.size() << ")\n";
    while (!cart.isEmpty()) {
        string t = cart.pop();
        cout << "pop  -> " << t << ", size " << cart.size()
             << ", top index " << cart.topIndex() << "\n";
    }
    try {
        cart.pop();
    } catch (const underflow_error& e) {
        cout << "pop on the empty cart -> error: " << e.what() << "\n";
    }
    cart.push("Cantik Itu Luka");
    cart.push("Gadis Kretek");
    cout << "refilled, size " << cart.size() << "; clear() -> ";
    cart.clear();
    cout << "size " << cart.size() << ", isEmpty " << (cart.isEmpty() ? "true" : "false")
<< "\n";
    return 0;
}
```

```
=== Pustaka Ceria: book return cart (array stack, capacity 4) ===
push Laskar Pelangi -> ok, size 1, top index 0
push Filosofi Kopi -> ok, size 2, top index 1
push Bumi Manusia -> ok, size 3, top index 2
push Negeri 5 Menara -> ok, size 4, top index 3
push Ronggeng Dukuh Paruk -> REJECTED, the cart is full
cart, top to bottom: Negeri 5 Menara | Bumi Manusia | Filosofi Kopi | Laskar Pelangi
peek -> Negeri 5 Menara (size is still 4)
pop  -> Negeri 5 Menara, size 3, top index 2
pop  -> Bumi Manusia, size 2, top index 1
pop  -> Filosofi Kopi, size 1, top index 0
pop  -> Laskar Pelangi, size 0, top index -1
pop on the empty cart -> error: pop from an empty stack
refilled, size 2; clear() -> size 0, isEmpty true
```

Match the output against **Figure 5.3**. The four accepted pushes move the top index to 0, 1, 2, 3; the fifth push is rejected with “the cart is full”; the pops return the books in exactly the reverse of the order they were pushed; and the pop on an empty cart is reported as an error rather than silently reading garbage.

Several design decisions in the class are worth explaining, because each one prevents a real bug.

- `push` **returns** `bool`**;** `pop` **and** `peek` **throw.** A full cart is a normal event at a busy desk — the caller should check and react — so `push` reports it with a return value. Popping an empty stack, on the other hand, means the calling code has a logic error, so it throws `std::underflow_error`. This mirrors the standard library's own choices (Section 5.16).
- **The copy constructor and copy assignment are deleted.** The class owns a raw array allocated with `new[]`. If C++ were allowed to copy the object member by member, two objects would hold the same pointer, and both destructors would `delete[]` it — a double free. Deleting the copy operations turns that runtime disaster into a compile-time error. Chapter 6 explains ownership and pointers properly.
- `peek` **returns** `const T&`**.** The caller can look at the top element without copying it and without being able to modify it through the reference.
- `at(i)` **exists only for printing.** A true stack would not offer it. We keep it so that `printCart` can show the whole cart, and we name the function so that nobody mistakes it for a stack operation.

5.4.3 The same stack in Java and Python

The book is written for C++, but every program in the Code folder also exists in Java and Python, with identical output. For the stack itself it is worth seeing all three side by side, because the differences teach something. **Listing 5.3** is the Java version.

Listing 5.3 `ArrayStackDemo.java`: the same array stack as a Java generic class.

```
// An array-based stack: Pustaka Ceria's book return cart.
public class ArrayStackDemo {
    static class ArrayStack<T> {
        private final T[] data;
        private int top = -1;                       // index of the top element, -1 if
empty

        @SuppressWarnings("unchecked")
        ArrayStack(int capacity) { data = (T[]) new Object[capacity]; }

        boolean isEmpty() { return top == -1; }
        boolean isFull()  { return top == data.length - 1; }
        int size()        { return top + 1; }
        int topIndex()    { return top; }

        boolean push(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 new IllegalStateException("pop from an empty stack");
            T value = data[top];
            data[top--] = null;                     // let the garbage collector reclaim it
            return value;
        }
        T peek() {
            if (isEmpty()) throw new IllegalStateException("peek at an empty stack");
            return data[top];
        }
        void clear() { while (top >= 0) data[top--] = null; }
        T at(int i)  { return data[i]; }            // for printing only
    }

    static void printCart(ArrayStack<String> s) {
        StringBuilder line = new StringBuilder("cart, top to bottom:");
        for (int i = s.topIndex(); i >= 0; --i) line.append(i == s.topIndex() ? " " : " |
").append(s.at(i));
        System.out.println(line);
    }
```

Listing 5.3 (Continued) *ArrayStackDemo.java: the same array stack as a Java generic class.*

```
    public static void main(String[] args) {
        System.out.println("=== Pustaka Ceria: book return cart (array stack, capacity 4)
===");
        ArrayStack<String> cart = new ArrayStack<>(4);
        String[] returned = {"Laskar Pelangi", "Filosofi Kopi", "Bumi Manusia",
                             "Negeri 5 Menara", "Ronggeng Dukuh Paruk"};
        for (String title : returned) {
            if (cart.push(title))
                System.out.println("push " + title + " -> ok, size " + cart.size()
                                   + ", top index " + cart.topIndex());
            else
                System.out.println("push " + title + " -> REJECTED, the cart is full");
        }
        printCart(cart);
        System.out.println("peek -> " + cart.peek() + " (size is still " + cart.size() +
")");
        while (!cart.isEmpty()) {
            String t = cart.pop();
            System.out.println("pop  -> " + t + ", size " + cart.size() + ", top index " +
cart.topIndex());
        }
        try {
            cart.pop();
        } catch (IllegalStateException e) {
            System.out.println("pop on the empty cart -> error: " + e.getMessage());
        }
        cart.push("Cantik Itu Luka");
        cart.push("Gadis Kretek");
        System.out.print("refilled, size " + cart.size() + "; clear() -> ");
        cart.clear();
        System.out.println("size " + cart.size() + ", isEmpty " + cart.isEmpty());
    }
}
```

Java cannot create an array of a generic type directly — `new T[capacity]` is illegal, because generic types are erased at run time — so the constructor creates an `Object[]` and casts it, and the `@SuppressWarnings` annotation acknowledges that the cast is safe because the array never leaves the class. Java's `pop` also writes `null` into the vacated slot. In C++ the stale value in the slot was harmless; in Java it would be a *reference* that keeps the object alive, so the garbage collector could never reclaim the book even after it has left the stack. Nulling the slot is the Java way of saying "this slot no longer holds anything".

Listing 5.4 is the Python version. A Python `list` could serve as a stack with no class at all (Section 5.16 does exactly that), but here we deliberately allocate a fixed block of `capacity` slots and keep our own `_top`, so the program has the same shape as the C++ one.

Listing 5.4 `02_array_stack.py`: the same array stack in Python, with a fixed-size list and an explicit top index.

```
# An array-based stack: Pustaka Ceria's book return cart.


class ArrayStack:
    def __init__(self, capacity):
        self._data = [None] * capacity   # a fixed block of slots, like a C++ array
        self._top = -1                   # index of the top element, -1 if empty

    def is_empty(self):
        return self._top == -1

    def is_full(self):
        return self._top == len(self._data) - 1

    def size(self):
        return self._top + 1

    def top_index(self):
        return self._top

    def push(self, value):               # False = overflow, nothing changed
        if self.is_full():
            return False
        self._top += 1                   # first move top up, then write
        self._data[self._top] = value
        return True

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from an empty stack")
        value = self._data[self._top]
        self._data[self._top] = None     # drop the reference
        self._top -= 1
        return value

    def peek(self):
        if self.is_empty():
            raise IndexError("peek at an empty stack")
        return self._data[self._top]

    def clear(self):
        while self._top >= 0:
            self._data[self._top] = None
            self._top -= 1
```

Listing 5.4 (Continued) *02_array_stack.py: the same array stack in Python, with a fixed-size list and an explicit top index.*

```
    def at(self, i):                    # for printing only
        return self._data[i]


def print_cart(s):
    parts = [s.at(i) for i in range(s.top_index(), -1, -1)]
    print("cart, top to bottom: " + " | ".join(parts))


def main():
    print("=== Pustaka Ceria: book return cart (array stack, capacity 4) ===")
    cart = ArrayStack(4)
    returned = ["Laskar Pelangi", "Filosofi Kopi", "Bumi Manusia",
                "Negeri 5 Menara", "Ronggeng Dukuh Paruk"]
    for title in returned:
        if cart.push(title):
            print(f"push {title} -> ok, size {cart.size()}, top index {cart.top_index()}")
        else:
            print(f"push {title} -> REJECTED, the cart is full")
    print_cart(cart)
    print(f"peek -> {cart.peek()} (size is still {cart.size()})")
    while not cart.is_empty():
        t = cart.pop()
        print(f"pop  -> {t}, size {cart.size()}, top index {cart.top_index()}")
    try:
        cart.pop()
    except IndexError as e:
        print(f"pop on the empty cart -> error: {e}")
    cart.push("Cantik Itu Luka")
    cart.push("Gadis Kretek")
    print(f"refilled, size {cart.size()}; clear() -> ", end="")
    cart.clear()
    print(f"size {cart.size()}, isEmpty {'true' if cart.is_empty() else 'false'}")


if __name__ == "__main__":
    main()
```

Both programs print exactly the output shown after **Listing 5.2**. Python raises `IndexError` for underflow, which is what Python's own `list.pop()` raises on an empty list, so a Python programmer meets a familiar exception.

5.4.4 Every operation is $O(1)$

Look at each method again and count the work. `push` does one comparison, one increment and one write. `pop` does one comparison, one read and one decrement. `peek`, `isEmpty`, `isFull` and `size` do one or two primitive steps. None of them contains a loop, and none of them depends on how many elements are stored. Each is therefore $O(1)$ — constant time — whether the stack holds four books or four million. `clear` is $O(1)$ in C++ (one assignment) and $O(n)$ in the Java and Python versions, which walk down the array writing `null`/`None` so that the garbage collector can reclaim the objects; that extra cost buys correct memory behavior.

The price of this speed is the capacity limit. A bounded stack can overflow. The next section removes the limit in two different ways.

5.5 Growing Stacks and Linked Stacks

A fixed capacity is sometimes exactly right — the return cart really does hold four books, and an embedded controller may really have room for only 256 entries. More often we do not know in advance how many elements will arrive. There are two standard cures. The first keeps the array but replaces it with a larger one when it fills up. The second abandons the array and links elements together with pointers.

5.5.1 Doubling the array

When `push` finds the array full, allocate a new array **twice** as large, copy every element across, free the old array, and carry on. **Listing 5.5** does exactly this, starting from a capacity of 1 so that we can watch several doublings.

Listing 5.5 `03_growing_stack.cpp`: a stack that doubles its array when it fills, counting every element copy.

```
// A stack that never becomes full: when the array is full, allocate one twice as big.
#include <iostream>
using namespace std;

class GrowingStack {
public:
    GrowingStack() : data_(new int[1]), capacity_(1), top_(-1), copies_(0) {}
    ~GrowingStack() { delete[] data_; }
    GrowingStack(const GrowingStack&) = delete;
    GrowingStack& operator=(const GrowingStack&) = delete;

    void push(int value) {
        if (top_ == capacity_ - 1) grow();              // full: double first
        data_[++top_] = value;
    }
    int  pop()      { return data_[top_--]; }            // caller checks isEmpty() first
    bool isEmpty()  const { return top_ == -1; }
    int  size()     const { return top_ + 1; }
    int  capacity() const { return capacity_; }
    long copies()   const { return copies_; }

private:
    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;
    }
    int* data_;
    int  capacity_;
    int  top_;
    long copies_;                                       // total element copies so far
};
```

Listing 5.5 (Continued) *03_growing_stack.cpp: a stack that doubles its array when it fills, counting every element copy.*

```
int main() {
    GrowingStack s;
    const int n = 17;
    cout << "pushing 1.." << n << " onto a stack that starts with capacity 1\n";
    for (int v = 1; v <= n; ++v) s.push(v);
    cout << "size " << s.size() << ", capacity " << s.capacity()
         << ", total copies " << s.copies() << " (less than 2n = " << 2 * n << ")\n";
    cout << "popping everything:";
    while (!s.isEmpty()) cout << " " << s.pop();
    cout << "\n";
    return 0;
}
```

```
pushing 1..17 onto a stack that starts with capacity 1
  grow: capacity 1 -> 2, copied 1 element(s)
  grow: capacity 2 -> 4, copied 2 element(s)
  grow: capacity 4 -> 8, copied 4 element(s)
  grow: capacity 8 -> 16, copied 8 element(s)
  grow: capacity 16 -> 32, copied 16 element(s)
size 17, capacity 32, total copies 31 (less than 2n = 34)
popping everything: 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

A single push can now be expensive: the push that found 16 elements in a 16-slot array had to copy all 16 before it could write the seventeenth. So is `push` still $O(1)$? Not for every individual push — but on average over any sequence of pushes, yes, and the output shows why.

Seventeen pushes caused copies of $1 + 2 + 4 + 8 + 16 = 31$ elements. In general, if n elements are pushed starting from capacity 1, the array doubles when its size reaches 1,2,4, …, 2^k, where 2^k is the largest power of two below n. The total number of copies is a geometric series:

$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n. \tag{5.5}$$

The last inequality holds because $2^k < n$ implies $2^{k+1} < 2n$. So n pushes cost at most n writes plus fewer than $2n$ copies — fewer than $3n$ element operations in total, which is an average of fewer than 3 per push. Equation (5.5) is the heart of what is called **amortized analysis**: the occasional expensive push is paid for by the many cheap ones before it, so the *amortized* cost per push is $O(1)$ even though the *worst-case* cost of a single push is $O(n)$. The factor matters: growing by a fixed amount (say, 10 slots at a time) instead of doubling would make the copies sum to roughly $n^2/20$, which is $O(n)$ per push on average. Sedgewick and Wayne call the doubling array a *resizing array* and prove the same bound [4].

This is precisely how `std::vector`, Java's `ArrayList` and `ArrayDeque`, and Python's `list` grow — with growth factors that differ in detail, but always by a constant *factor* rather than a constant amount.

5.5.2 A preview: the linked stack

The second cure stores each element in its own small block of memory, a *node*, which also records where the next node is. The top of the stack is a pointer to the first node; each node points to the node beneath it; the bottom node points to nothing (`nullptr`). Pushing creates a new node that points at the old top and then becomes the top. Popping moves the top pointer to the next node and frees the old one. **Figure 5.4** shows both.

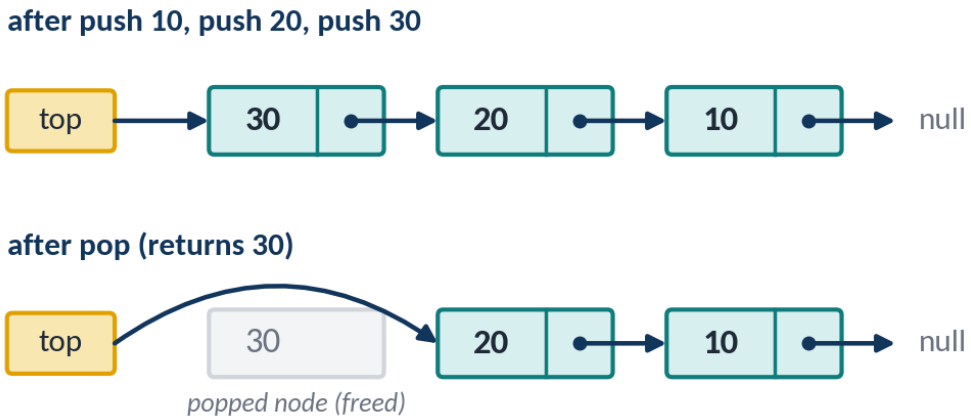


Figure 5.4 A linked stack after pushing 10, 20 and 30 (top), and after one pop (bottom): the top pointer skips the old first node, which is then freed.

Pointers are the subject of Chapter 6 and linked lists of Chapter 7, so **Listing 5.6** is only a preview: read it for the idea, and come back to it after Chapter 7.

Listing 5.6 `04_linked_stack.cpp`: a preview of a stack built from linked nodes.

```
// A preview of Chapters 6 and 7: a stack made of linked nodes instead of an array.
#include <iostream>
#include <stdexcept>
using namespace std;

struct Node {
    int   value;
    Node* next;                                     // the node underneath this one
};

class LinkedStack {
public:
    LinkedStack() : top_(nullptr), size_(0) {}
    ~LinkedStack() { while (!isEmpty()) pop(); }        // free every node
    LinkedStack(const LinkedStack&) = delete;
    LinkedStack& operator=(const LinkedStack&) = delete;

    void push(int v) {
        top_ = new Node{v, top_};                       // new node points at the old top
        ++size_;
    }
    int pop() {
        if (isEmpty()) throw underflow_error("pop from an empty stack");
        Node* old = top_;
        int v = old->value;
        top_ = old->next;                               // unlink first ...
        delete old;                                     // ... then free
        --size_;
        return v;
    }
    int  peek()    const { if (isEmpty()) throw underflow_error("peek at an empty stack");
return top_->value; }
    bool isEmpty() const { return top_ == nullptr; }
    int  size()    const { return size_; }
    void print()   const {
        cout << "top";
        for (Node* p = top_; p != nullptr; p = p->next) cout << " -> " << p->value;
        cout << " -> null   (size " << size_ << ")\n";
    }

private:
    Node* top_;
    int   size_;
};
```

Listing 5.6 (Continued) *04_linked_stack.cpp: a preview of a stack built from linked nodes.*

```
int main() {
    LinkedStack s;
    s.print();
    for (int v : {10, 20, 30}) { s.push(v); cout << "push " << v << ": "; s.print(); }
    cout << "peek -> " << s.peek() << "\n";
    int v = s.pop();
    cout << "pop  -> " << v << ": "; s.print();
    s.push(40);
    cout << "push 40: "; s.print();
    cout << "(" << s.size() << " nodes remain; they are released when s goes out of
scope)\n";
    return 0;
}
```

```
top -> null   (size 0)
push 10: top -> 10 -> null   (size 1)
push 20: top -> 20 -> 10 -> null   (size 2)
push 30: top -> 30 -> 20 -> 10 -> null   (size 3)
peek -> 30
pop  -> 30: top -> 20 -> 10 -> null   (size 2)
push 40: top -> 40 -> 20 -> 10 -> null   (size 3)
(3 nodes remain; they are released when s goes out of scope)
```

Three lines carry the whole idea. `top_ = new Node{v, top_};` builds a node whose `next` is the old top and makes it the new top, in one statement. In `pop`, `top_ = old->next;` unlinks the node *before* `delete old;` frees it — reversing those two lines would read `old->next` from freed memory. And the destructor pops until the stack is empty, so no node is ever leaked. We compiled the program with `-fsanitize=address,undefined` and ran it under Valgrind: zero errors, and "All heap blocks were freed — no leaks are possible".

A linked stack never overflows (until the machine runs out of memory), and every operation is $O(1)$ in the worst case, not merely on average. Its costs are one allocation per push, one extra pointer per element, and nodes scattered around memory, which modern processors read more slowly than a contiguous array. In practice the doubling array usually wins; the standard libraries of all three languages use arrays for their stacks.

5.6 Application: Matching Brackets

Every compiler, every code editor that colors a matching parenthesis, every JSON and HTML parser must answer the same question: do the brackets in this text *nest* properly? The text `{[()()]}` is fine. The text `([)]` is not: the `)` arrives while the innermost open bracket is `[`. The text `((a+b)*c` is not either: one `(` is never closed.

Counting is not enough. `([)]` has two openers and two closers of each kind, and it is still wrong. What matters is that **every closing bracket must match the most recent opening bracket that has not been matched yet** — and "most recent not yet matched" is the definition of the top of a stack.

5.6.1 The algorithm

Pseudocode 5.2 scans the text once from left to right.

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"
```

In plain words: push every opener; at every closer, the top of the stack must be its partner, and if so pop it; at the end, the stack must be empty. There are exactly three ways to fail, and each has its own line: a closer with nothing open (line 8), a closer that does not match the innermost opener (line 10), and an opener that is never closed (line 14).

Figure 5.5 traces the balanced case `{[()()]}` character by character.

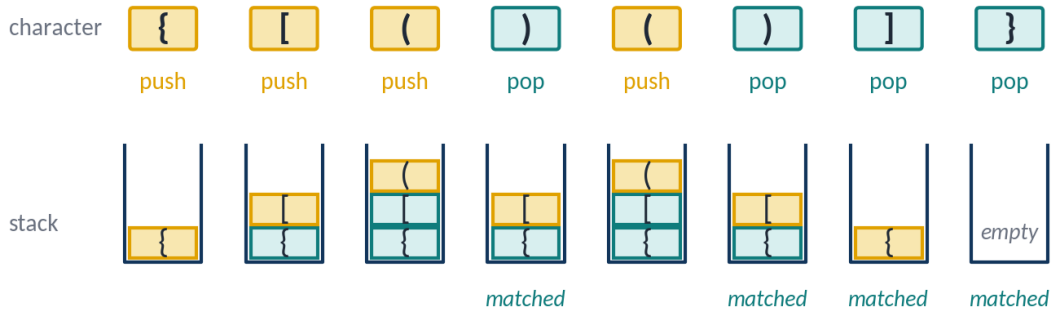


Figure 5.5 Checking `{[()()]}`: each opener is pushed, each closer pops its partner from the top, and the stack is empty at the end, so the text is balanced.

5.6.2 The program

Listing 5.7 implements **Pseudocode 5.2** with the standard library's `std::stack<char>` — we built our own stack in Section 5.4, and from now on we are allowed to use the library's. It prints a trace for the first test and a verdict for six texts.

Listing 5.7 `05_brackets.cpp`: bracket matching with `std::stack`, reporting the position and kind of the first error.

```
// Bracket matching with a stack: ( ) [ ] { } must nest properly.
#include <iomanip>
#include <iostream>
#include <stack>
#include <string>
using namespace std;

bool isOpen(char c)  { return c == '(' || c == '[' || c == '{'; }
bool isClose(char c) { return c == ')' || c == ']' || c == '}'; }
char partner(char close) { return close == ')' ? '(' : close == ']' ? '[' : '{'; }

string contents(stack<char> s) {                        // a copy, printed bottom -> top
    string r;
    while (!s.empty()) { r = string(1, s.top()) + (r.empty() ? "" : " ") + r; s.pop(); }
    return r.empty() ? "(empty)" : r;
}

// Returns "balanced" or a message naming the first problem and its position (0-based).
string check(const string& text, bool trace) {
    stack<char> open;
    for (size_t i = 0; i < text.size(); ++i) {
        char c = text[i];
        if (isOpen(c)) {
            open.push(c);
            if (trace) cout << "  " << c << "  push           stack: " << contents(open) <<
"\n";
        } 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";
}
```

Listing 5.7 (Continued) *05_brackets.cpp: bracket matching with std::stack, reporting the position and kind of the first error.*

```
int main() {
    cout << "trace of {[()()]}:\n";
    string verdict = check("{[()()]}", true);
    cout << "  result: " << verdict << "\n\n";
    const string tests[] = {"{[()()]}", "([)]", "((a+b)*c", "a+b)*(c",
                            "if (a[i] == b[j]) { x(); }", ""};
    for (const string& t : tests)
        cout << left << setw(30) << ("\"" + t + "\"") << check(t, false) << "\n";
    return 0;
}
```

```
trace of {[()()]}:
  {  push           stack: {
  [  push           stack: { [
  (  push           stack: { [ (
  )  pop, matches   stack: { [
  (  push           stack: { [ (
  )  pop, matches   stack: { [
  ]  pop, matches   stack: {
  }  pop, matches   stack: (empty)
  result: 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
```

Each of the three failure modes appears once. In `([)]` the `)` at position 2 finds `[` on top: a mismatch. In `((a+b)*c` the scan ends with one `(` still on the stack. In `a+b)*(c` the `)` at position 3 finds the stack empty. The line `if (a[i] == b[j]) { x(); }` shows that ordinary characters are simply skipped, and the empty string is balanced, because an empty stack at the end is the only requirement.

The algorithm reads each character once and does $O(1)$ stack work per character, so it runs in $O(n)$ time for a text of length n. In the worst case — a text of n openers — the stack holds n characters, so it also needs $O(n)$ extra space.

5.7 Application: Undo and Redo

Press Ctrl+Z in any editor and the most recent change disappears. Press it again and the change before that disappears. The *last* change made is the *first* one undone: undo is LIFO. Press Ctrl+Y (or Ctrl+Shift+Z) and the most recently undone change comes back — redo is LIFO too. An editor therefore keeps two stacks.

- Before every edit, **push the current state onto the undo stack**, make the edit, and **clear the redo stack** — once you type something new, the changes you undid earlier no longer have a place to go back to.
- To **undo**, push the current state onto the redo stack and pop the undo stack into the current state.
- To **redo**, push the current state onto the undo stack and pop the redo stack into the current state.

Listing 5.8 applies this to the one-line note a librarian types into the Pustaka Ceria catalog. For clarity it stores whole copies of the text (a technique called a *memento*); real editors store the edit operations instead, which is cheaper for long documents but follows the same two-stack discipline.

Listing 5.8 `06_undo_redo.cpp`: undo and redo with two stacks of saved states.

```
// Undo and redo with two stacks, for a one-line note in the Pustaka Ceria catalog.
#include <iomanip>
#include <iostream>
#include <stack>
#include <string>
using namespace std;

class Editor {
public:
    void type(const string& s) {
        undo_.push(text_);                              // remember the state before the
change
        text_ += s;
        while (!redo_.empty()) redo_.pop();             // a new edit makes old redos
meaningless
        report("type \"" + s + "\"");
    }
    void erase(int k) {                                 // delete the last k characters
        undo_.push(text_);
        text_.erase(text_.size() - static_cast<size_t>(k));
        while (!redo_.empty()) redo_.pop();
        report("erase " + to_string(k));
    }
    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");
    }

private:
    void report(const string& action) const {
        cout << left << setw(24) << action << setw(26) << ("\"" + text_ + "\"")
             << "undo " << undo_.size() << "  redo " << redo_.size() << "\n";
    }
    string text_;
    stack<string> undo_, redo_;
};
```

Listing 5.8 (Continued) *06_undo_redo.cpp: undo and redo with two stacks of saved states.*

```
int main() {
    Editor e;
    cout << left << setw(24) << "action" << setw(26) << "text" << "stack sizes\n";
    e.type("Laskar");
    e.type(" Pelangi");
    e.type(", 2005");
    e.undo();
    e.undo();
    e.redo();
    e.erase(4);
    e.redo();
    e.undo();
    e.undo();
    e.undo();
    e.undo();
    return 0;
}
```

```
action                 text                      stack sizes
type "Laskar"          "Laskar"                  undo 1  redo 0
type " Pelangi"        "Laskar Pelangi"          undo 2  redo 0
type ", 2005"          "Laskar Pelangi, 2005"    undo 3  redo 0
undo                   "Laskar Pelangi"          undo 2  redo 1
undo                   "Laskar"                  undo 1  redo 2
redo                   "Laskar Pelangi"          undo 2  redo 1
erase 4                "Laskar Pel"              undo 3  redo 0
redo (nothing to redo) "Laskar Pel"              undo 3  redo 0
undo                   "Laskar Pelangi"          undo 2  redo 1
undo                   "Laskar"                  undo 1  redo 2
undo                   ""                        undo 0  redo 3
undo (nothing to undo) ""                        undo 0  redo 3
```

Figure 5.6 draws the two stacks at four moments of the run. After three edits (panel 1) the undo stack holds the three earlier states. Two undos (panel 2) move two states across to the redo stack. One redo (panel 3) moves one back. Then comes the important case: `erase 4` is a new edit, so it pushes onto the undo stack **and empties the redo stack** (panel 4). That is why the following `redo` in the output reports "nothing to redo". If the redo stack were not cleared, a redo would resurrect "Laskar Pelangi, 2005" from a history that the new edit has already overwritten — a genuinely confusing bug in some early editors.

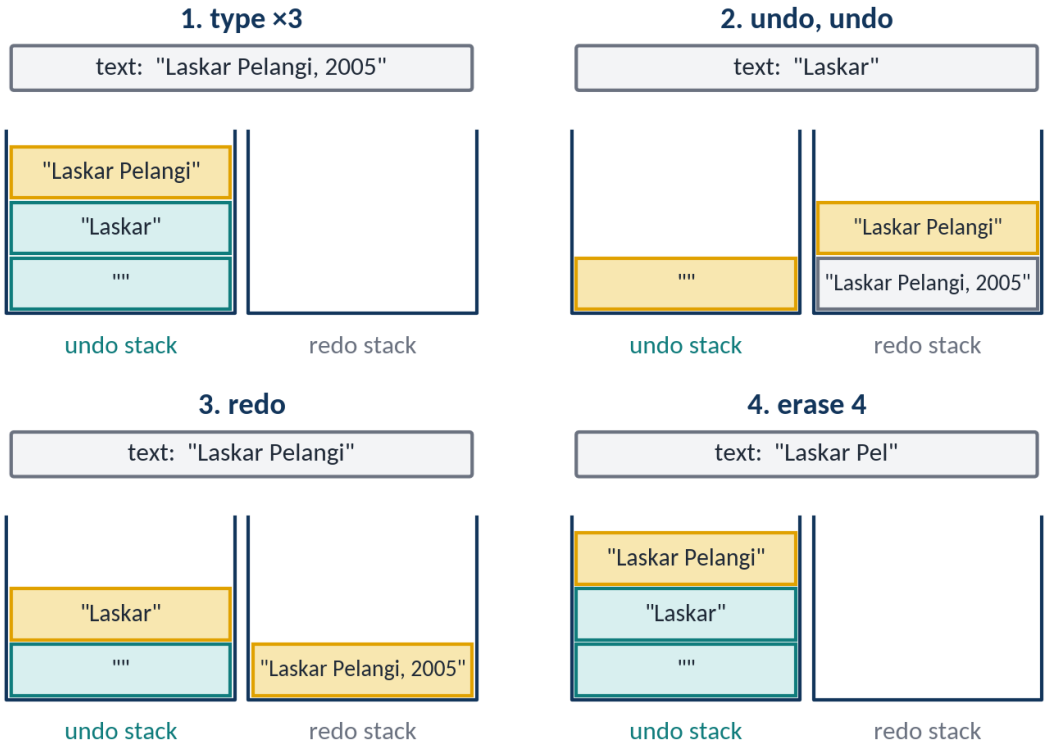


Figure 5.6 The undo and redo stacks at four moments of **Listing 5.8**. A new edit (panel 4) pushes onto the undo stack and empties the redo stack.

Every undo and redo is a pop and a push: $O(1)$ stack operations, plus the cost of copying the text. The same two stacks drive the Back and Forward buttons of a web browser, where "a new edit" corresponds to clicking a fresh link — which, as you may have noticed, wipes out your Forward history.

5.8 Application: Infix to Postfix with the Shunting-Yard Algorithm

This section and the next build a small scientific calculator, and together they are the best argument ever made for the stack.

5.8.1 Three ways to write the same sum

We write arithmetic in **infix** notation: each binary operator stands *between* its two operands, as in `3 + 2 * 5`. Infix is easy for people and awkward for machines, because the order in which the operators are written is not the order in which they are applied. In `3 + 2 * 5` the `*` must be applied first, even though the `+` comes first; to override that, we need parentheses, as in `(3 + 2) * 5`. A program that evaluates infix must therefore know precedence, associativity and parentheses.

In the 1920s the Polish logician Jan Łukasiewicz observed that if every operator is written *before* its operands — **prefix** notation, `+ 3 * 2 5` — no parentheses are ever needed. The mirror image, with every operator written *after* its operands, is **postfix** notation, `3 2 5 * +`, also called *reverse Polish notation* (RPN). Charles Hamblin in Australia and Friedrich Bauer and Edsger Dijkstra in Europe independently saw in the late 1950s that postfix is exactly what a machine with a stack can evaluate in a single left-to-right pass [5], [6]. **Table 5.2** compares the three notations.

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

Read the postfix column carefully. The operands appear in the same order as in the infix column — only the operators move. And the operators appear in **exactly the order in which they are applied**: in `3 2 5 * +` the multiplication comes first because it is performed first. That is why no parentheses are needed: the order of the text *is* the order of evaluation.

5.8.2 Precedence and associativity

To convert infix to postfix we must know which operator binds more tightly. **Table 5.3** gives the rules used in this chapter; they are the rules of ordinary school arithmetic, with `^` for exponentiation.

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: subtraction groups from the left, so `8-3-2` is $(8 - 3) - 2 = 3$ and not $8 - (3 - 2) = 7$; exponentiation groups from the right, so `2^3^2` is $2^{9} = 512$ and not $8^{2} = 64$.

5.8.3 The algorithm

The conversion was published by Dijkstra in 1961 [6], who named it after a railway *shunting yard*: operands travel straight through from input to output, while operators are diverted onto a side

track — a stack — and wait there until it is their turn to rejoin the train. The rule for when an operator may leave the side track is the whole 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
```

In plain words: an operand goes straight to the output. An opening parenthesis goes on the stack as a barrier. A closing parenthesis pops operators to the output until it meets its barrier, which it throws away. An incoming operator first pops every operator on the stack that must be applied *before* it — those with higher precedence, or with equal precedence when the incoming one is left-associative — and then waits on the stack itself. At the end, whatever is still waiting is popped to the output.

Line 12 is the only subtle line, so here it is once more in terms of the example. When `-` arrives in `a+b*c-d`, the stack holds `+ *`. The `*` has higher precedence than `-`, so it must be applied first: pop it. Then `+` has the *same* precedence as `-`, and `-` is left-associative, so `a+b*c` must be finished before the `-` is applied: pop `+` as well. Only then is `-` pushed. For the right-associative `^`, the equal-precedence case does **not** pop, which is exactly what makes `2^3^2` group from the right.

5.8.4 Tracing `3+2*5`

Figure 5.7 traces the smallest interesting example, token by token.

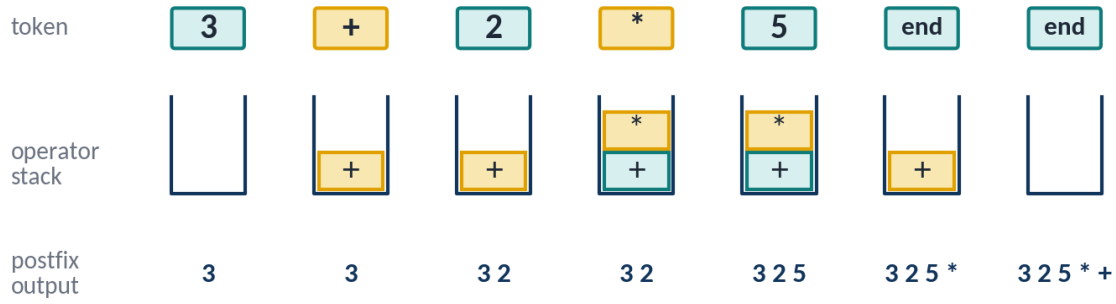


Figure 5.7 Converting `3+2*5` to postfix. The `*` is pushed on top of `+` because it binds more tightly; at the end the stack is emptied from the top, giving `3 2 5 * +`.

1. `3` is an operand: output `3`.
2. `+` meets an empty stack: push it.

3. `2`: output `3 2`.
4. `*` meets `+` on top. Is `prec(+) > prec(*)`? No, $1 < 2$. So nothing is popped, and `*` is pushed on top of `+`. This single comparison is what makes multiplication happen before addition.
5. `5`: output `3 2 5`.
6. End of input: pop `*`, output `3 2 5 *`; pop `+`, output `3 2 5 * +`.

5.8.5 Tracing `a+b*c-d`

Table 5.4 traces a longer example with variables, including the double pop described after **Pseudocode 5.3**.

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 -`

The result `a b c * + d -` reads: multiply `b` and `c`; add `a`; subtract `d`. That is exactly $(a + (b \times c)) - d$, the meaning of the infix text.

5.9 Application: Evaluating Postfix Expressions

Once an expression is in postfix, evaluating it needs one stack of *numbers* and one rule.

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)
```

In plain words: push numbers; when an operator arrives, the two numbers it needs are the two most recent ones, so pop them, apply the operator, and push the result, which may itself become an operand of a later operator. At the end exactly one number must remain.

Lines 8 and 9 hide the classic trap. For `8 3 -` the stack holds `8 3` with 3 on top, so the first pop returns `3` and the second returns `8`. The operator must compute $x - y = 8 - 3 = 5$. Computing

$y - x$ would give -5. Addition and multiplication do not care about the order, which is exactly why the bug survives testing on `3 2 5 * +` and then appears on the first subtraction or division.

Figure 5.8 evaluates `3 2 5 * +`.

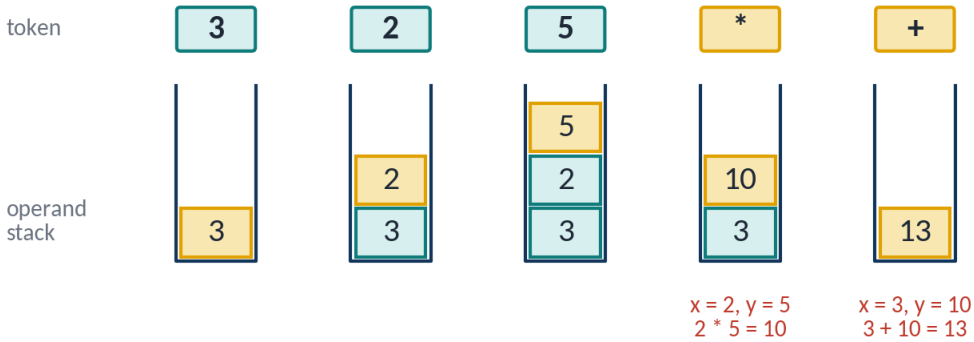


Figure 5.8 Evaluating `3 2 5 * +`. Operands are pushed; each operator pops y (the top) and then x, and pushes x op y. The single value left, 13, is the result.

5.9.1 The complete calculator

Listing 5.9 puts the two algorithms together. It splits the input into tokens (so that multi-digit numbers like `100` work), converts to postfix with **Pseudocode 5.3**, evaluates with **Pseudocode 5.4**, prints full traces for the two worked examples, and then runs a batch of harder cases — including four inputs that are wrong on purpose.

Listing 5.9 `07_calculator.cpp`: a calculator that converts infix to postfix with the shunting-yard algorithm and evaluates the postfix with an operand stack.

```cpp
// The scientific calculator: infix -> postfix (shunting-yard), then postfix evaluation.
#include <cctype>
#include <iomanip>
#include <iostream>
#include <map>
#include <sstream>
#include <stdexcept>
#include <string>
#include <vector>
using namespace std;

map<char, long long> vars = {{'a', 2}, {'b', 3}, {'c', 4}, {'d', 5}};

int  prec(const string& op)       { return op == "^" ? 3 : (op == "*" || op == "/") ? 2 :
1; }
bool rightAssoc(const string& op) { return op == "^"; }
bool isOperator(const string& t)  { return t == "+" || t == "-" || t == "*" || t == "/" ||
t == "^"; }

vector<string> tokenize(const string& s) {            // "12+x" -> {"12", "+", "x"}
    vector<string> out;
    for (size_t i = 0; i < s.size();) {
        if (isspace(static_cast<unsigned char>(s[i]))) { ++i; continue; }
        if (isdigit(static_cast<unsigned char>(s[i]))) {
            size_t j = i;
            while (j < s.size() && isdigit(static_cast<unsigned char>(s[j]))) ++j;
            out.push_back(s.substr(i, j - i));
            i = j;
        } else {
            out.push_back(string(1, s[i++]));
        }
    }
    return out;
}

string join(const vector<string>& v) {
    string r;
    for (size_t i = 0; i < v.size(); ++i) r += (i ? " " : "") + v[i];
    return r;
}
```

Listing 5.9 (Continued) *07_calculator.cpp: a calculator that converts infix to postfix with the shunting-yard algorithm and evaluates the postfix with an operand stack.*

```
void row(const string& tok, const string& action, const string& stack, const string& out) {
    ostringstream line;
    line << "  " << left << setw(6) << tok << setw(24) << action << setw(10) << stack <<
out;
    string text = line.str();
    while (!text.empty() && text.back() == ' ') text.pop_back();   // no trailing blanks
    cout << text << "\n";
}

vector<string> toPostfix(const string& infix, bool trace) {
    vector<string> out, ops;                          // ops is the operator stack (back
= top)
    if (trace) row("token", "action", "stack", "postfix");
    for (const string& t : tokenize(infix)) {
        string action;
        if (isalnum(static_cast<unsigned char>(t[0]))) {
            out.push_back(t);
            action = "output operand";
        } else if (t == "(") {
            ops.push_back(t);
            action = "push (";
        } else if (t == ")") {
            while (!ops.empty() && ops.back() != "(") {
                action += "pop " + ops.back() + ", ";
                out.push_back(ops.back());
                ops.pop_back();
            }
            if (ops.empty()) throw runtime_error("unmatched ')'");
            ops.pop_back();                           // discard the "("
            action += "drop (";
        } else if (isOperator(t)) {
            while (!ops.empty() && ops.back() != "(" &&
                   (prec(ops.back()) > prec(t) || (prec(ops.back()) == prec(t) &&
!rightAssoc(t)))) {
                action += "pop " + ops.back() + ", ";
                out.push_back(ops.back());
                ops.pop_back();
            }
            ops.push_back(t);
            action += "push " + t;
        } else {
            throw runtime_error("unknown symbol '" + t + "'");
        }
        if (trace) row(t, action, join(ops), join(out));
    }
    while (!ops.empty()) {                            // drain what is left
        if (ops.back() == "(") throw runtime_error("unmatched '('");
        out.push_back(ops.back());
        ops.pop_back();
```

Listing 5.9 (Continued) *07_calculator.cpp: a calculator that converts infix to postfix with the shunting-yard algorithm and evaluates the postfix with an operand stack.*

```
        if (trace) row("end", "pop " + out.back(), join(ops), join(out));
    }
    return out;
}

long long apply(long long x, const string& op, long long y) {
    if (op == "+") return x + y;
    if (op == "-") return x - y;
    if (op == "*") return x * y;
    if (op == "/") { if (y == 0) throw runtime_error("division by zero"); return x / y; }
    long long r = 1;                                    // "^": y >= 0 in this book
    for (long long i = 0; i < y; ++i) r *= x;
    return r;
}

long long evaluate(const vector<string>& postfix, bool trace) {
    vector<long long> st;                               // the operand stack (back = top)
    auto show = [&]() { string r; for (size_t i = 0; i < st.size(); ++i) r += (i ? " " :
"") + to_string(st[i]); return r; };
    if (trace) row("token", "action", "stack", "");
    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 5.9 (Continued) *07_calculator.cpp: a calculator that converts infix to postfix with the shunting-yard algorithm and evaluates the postfix with an operand stack.*

```
int main() {
    for (const string e : {"3+2*5", "a+b*c-d"}) {
        cout << "infix " << e << "  (a=2, b=3, c=4, d=5)\n";
        vector<string> p = toPostfix(e, true);
        cout << "postfix: " << join(p) << "\n";
        long long v = evaluate(p, true);
        cout << "value: " << v << "\n\n";
    }
    for (const string e : {"(7+3)*(10-4)/5", "2^3^2", "(2^3)^2", "100/10/5", "8-3-2",
                           "(1+2", "1+2)", "1+", "7/(3-3)"}) {
        cout << left << setw(16) << e;
        try {
            vector<string> p = toPostfix(e, false);
            cout << "postfix " << setw(22) << join(p);
            long long v = evaluate(p, false);
            cout << "= " << v << "\n";
        } catch (const exception& ex) {
            cout << "error: " << ex.what() << "\n";
        }
    }
    return 0;
}
```

```
infix 3+2*5  (a=2, b=3, c=4, d=5)
  token action                  stack     postfix
  3     output operand                    3
  +     push +                  +         3
  2     output operand          +         3 2
  *     push *                  + *       3 2
  5     output operand          + *       3 2 5
  end   pop *                   +         3 2 5 *
  end   pop +                               3 2 5 * +
postfix: 3 2 5 * +
  token action                  stack
  3     push 3                  3
  2     push 2                  3 2
  5     push 5                  3 2 5
  *     2 * 5 = 10              3 10
  +     3 + 10 = 13             13
value: 13

infix a+b*c-d  (a=2, b=3, c=4, d=5)
  token action                  stack     postfix
  a     output operand                    a
  +     push +                  +         a
  b     output operand          +         a b
  *     push *                  + *       a b
  c     output operand          + *       a b c
  -     pop *, pop +, push -    -         a b c * +
  d     output operand          -         a b c * + d
  end   pop -                             a b c * + d -
postfix: a b c * + d -
  token action                  stack
  a     push 2                  2
  b     push 3                  2 3
  c     push 4                  2 3 4
  *     3 * 4 = 12              2 12
  +     2 + 12 = 14             14
  d     push 5                  14 5
  -     14 - 5 = 9              9
value: 9

(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
```

Check the traces against **Figure 5.7**, **Table 5.4** and **Figure 5.8**: they agree token for token, and the two values, 13 and 9, are what $3 + 2 \cdot 5$ and $2 + 3 \cdot 4 - 5$ give by hand. The batch confirms the rest of the theory:

- `(7+3)*(10-4)/5` shows parentheses at work — each `)` pops back to its `(` — and multi-digit numbers: $10 \cdot 6/5 = 12$.
- `2^3^2` gives 512 and `(2^3)^2` gives 64: right associativity, and parentheses overriding it.
- `100/10/5` gives 2 and `8-3-2` gives 3: left associativity, with x and y in the right order.
- `(1+2` and `1+2)` are caught during conversion (an unmatched parenthesis is a bracket-matching failure, Section 5.6); `1+` is caught during evaluation because `+` finds only one operand; and `7/(3-3)` is caught as division by zero.

Both phases read each token once and do $O(1)$ stack work per token, apart from the pops, and every token is pushed and popped at most once. Conversion and evaluation are therefore each

$O(n)$ for n tokens. The Java and Python versions of the calculator in the Code folder print identical output; the Python one needs one extra line, because Python's `//` rounds toward minus infinity while C++ and Java truncate toward zero.

5.10 Application: The Function Call Stack

The most important stack in your computer is one you have used in every program you have ever written, without seeing it.

5.10.1 What happens when a function is called

When `main` calls `lateFee(7, "reference")`, the processor must remember where to continue in `main` when `lateFee` returns, and `lateFee` needs somewhere to keep its parameters and local variables. When `lateFee` in turn calls `dailyRate`, the same thing happens again, one level deeper. When `dailyRate` returns, the processor must go back to `lateFee` — the most recent caller — not to `main`. The last function called is the first to return: function calls are LIFO.

So the runtime keeps a **call stack**. Each call pushes a *stack frame* (also called an *activation record*) holding:

- the **return address** — where to continue in the caller;
- the function's **parameters**;
- its **local variables**;
- saved processor registers and bookkeeping.

Each return pops the frame, and with it every local variable of that call. This is why a local variable "disappears" when its function returns, and why a function must never return a pointer or reference to one of its own local variables: the frame it lived in has been popped and will be overwritten by the next call.

5.10.2 Watching it happen

The call stack itself is managed by the compiler and the processor, so we cannot print it directly. But we can keep our own copy. In **Listing 5.10**, each function creates a `Frame` object as its first statement; the constructor pushes the function's name onto a vector, and the destructor — which C++ runs automatically when the function returns, by any path — pops it.

Listing 5.10 `08_call_stack.cpp`: making the call stack visible by pushing a name on every entry and popping it on every return.

```
// Making the call stack visible: every function pushes a frame on entry and pops it on
return.
#include <iostream>
#include <string>
#include <vector>
using namespace std;

vector<string> callStack;                                   // our own copy of what the machine
keeps

string picture() {                                          // bottom -> top
    string s;
    for (size_t i = 0; i < callStack.size(); ++i) s += (i ? " | " : "") + callStack[i];
    return s;
}

struct Frame {                                              // constructor = push, destructor =
pop
    explicit Frame(const string& name) {
        callStack.push_back(name);
        cout << "push  " << picture() << "\n";
    }
    ~Frame() {
        callStack.pop_back();
        cout << "pop   " << (callStack.empty() ? "(empty)" : picture()) << "\n";
    }
    Frame(const Frame&) = delete;
    Frame& operator=(const Frame&) = delete;
};

int dailyRate(const string& category) {
    Frame f("dailyRate(" + category + ")");
    return category == "reference" ? 5000 : 1000;           // rupiah per day
}

int capFee(int fee, int cap) {
    Frame f("capFee(" + to_string(fee) + ")");
    return fee < cap ? fee : cap;
}

int lateFee(int days, const string& category) {
    Frame f("lateFee(" + to_string(days) + ")");
    int fee = days * dailyRate(category);                   // a call inside an expression
    return capFee(fee, 25000);
}
```

Listing 5.10 (Continued) *08_call_stack.cpp: making the call stack visible by pushing a name on every entry and popping it on every return.*

```
int countdown(int n) {                                      // a preview of Chapter 10
    Frame f("countdown(" + to_string(n) + ")");
    if (n == 0) return 0;
    return 1 + countdown(n - 1);
}

int main() {
    Frame f("main");
    int fee = lateFee(7, "reference");
    cout << "        late fee = Rp" << fee << "\n";
    int steps = countdown(2);
    cout << "        countdown made " << steps << " recursive calls\n";
    return 0;
}
```

```
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
push  main | countdown(2)
push  main | countdown(2) | countdown(1)
push  main | countdown(2) | countdown(1) | countdown(0)
pop   main | countdown(2) | countdown(1)
pop   main | countdown(2)
pop   main
        countdown made 2 recursive calls
pop   (empty)
```

Figure 5.9 draws the first half of the run. Notice the order of events in `lateFee`: the expression `days * dailyRate(category)` calls `dailyRate` *in the middle* of computing `fee`, so `dailyRate`'s frame is pushed on top of `lateFee`'s, returns 5000, and is popped before `capFee` is even called. Then `capFee` gets its own frame in the same place, returns 25000 (the fine of 7 × 5000 = 35000 capped at Rp25,000), and is popped. Finally `lateFee` returns and only `main` remains.

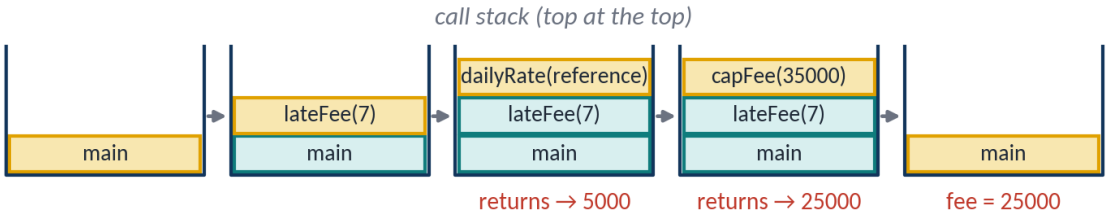


Figure 5.9 The call stack during `lateFee(7, "reference")`: each call pushes a frame on top, and each return pops it, handing its value to the frame below.

The second half of the output previews Chapter 10. `countdown(2)` calls `countdown(1)`, which calls `countdown(0)`: three frames of the *same* function are on the stack at once, each with its own `n`. Recursion is nothing more than a function whose frames are pushed on top of its own frames. That also explains its danger. Each frame costs memory, and the call stack is small — by default 1 MB for a Windows program built with Microsoft's linker [7], typically 8 MB on Linux (`ulimit -s` prints 8192 on the machine used for this book), and Python refuses to go deeper than 1,000 nested calls unless told otherwise. A recursion that never reaches its base case pushes frames until that memory runs out: a **stack overflow**, the error that gave a famous programming website its name.

The Java and Python versions of the program use `try`/`finally` and a context manager (`with`) to get the same push-on-entry, pop-on-exit behavior, and print the same lines.

5.11 The Queue as an Abstract Data Type

Turn now to the other discipline of **Figure 5.1**.

> **DEFINITION.** A *queue* is a collection of elements in which insertion happens at one end, the *rear* (or *tail*), and removal happens at the other end, the *front* (or *head*). The element removed is always the one that has been in the queue longest. This discipline is called *first in, first out* (FIFO).

Queues are the fairest structures in computing, and daily life is full of them: the line at a bank teller or an ATM, the cars at a toll gate on the Surabaya–Gempol toll road, passengers at an airport

security check, customers at a supermarket checkout. Inside computers they are, if anything, more common:

- **Print jobs.** Several people send documents A.doc, B.doc, C.doc and D.doc to one shared printer. The printer cannot print four at once, so it queues them in arrival order and prints them one by one.
- **Processes and threads.** An operating system keeps processes that are ready to run in a *ready queue*. In *round-robin* scheduling, the process at the front runs for a short time slice and then, if unfinished, goes back to the rear.
- **Network packets.** A router that receives packets faster than its outgoing link can send them keeps them in a queue — a *buffer* — and forwards them in order.
- **Input and output.** Keystrokes typed faster than a program reads them wait in a keyboard buffer; the Windows message queue holds mouse clicks and key presses for an application's event loop.
- **Graph search.** Breadth-first search (Chapter 13) visits a graph level by level by keeping the vertices it has discovered but not yet explored in a queue.

5.11.1 The operations

Table 5.5 lists the queue operations. As with stacks, the names vary between books and libraries — `enqueue` is also *add*, *offer*, *push* or *put*; `dequeue` is also *remove*, *poll*, *pop* or *get* — but the behavior does not.

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	—	$O(1)$

Figure 5.10 runs a short sequence on a queue: enqueue 70, 80 and 50; dequeue twice; enqueue 90 and 60. Watch which element leaves. Although 50 is the most recent arrival at the moment of the first dequeue, it is 70 — the element that has waited longest — that leaves; then 80. When 90 and 60 arrive they join behind 50.

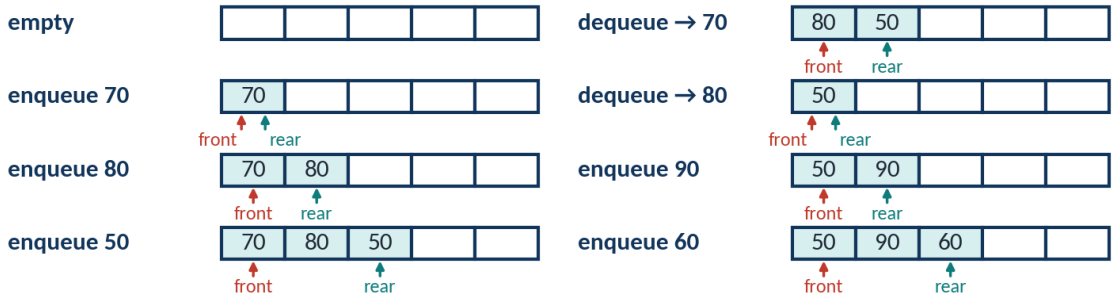


Figure 5.10 A queue through seven operations, drawn with the front on the left. Elements leave in exactly the order they arrived: 70, then 80.

The figure deliberately hides *how* the elements are stored: it redraws the queue from the left edge each time. The next two sections show that storing a queue in an array is harder than storing a stack, because a queue changes at **both** ends.

5.12 Queues in a Linear Array, and the Drifting Problem

A stack needed one index, because it changes at one end. A queue changes at two ends, so the obvious array implementation keeps two indices: `head` for the front and `tail` for the rear. There are two obvious ways to use them, and both are flawed in instructive ways.

5.12.1 First attempt: shift everything on every dequeue

Keep the front permanently at slot 0. Enqueue writes at `++tail`, exactly like a push. Dequeue takes `data[0]`, then shifts every remaining element one slot to the left to close the gap, and decrements `tail`. **Figure 5.11** shows one dequeue from the queue `4 45 1 8`.

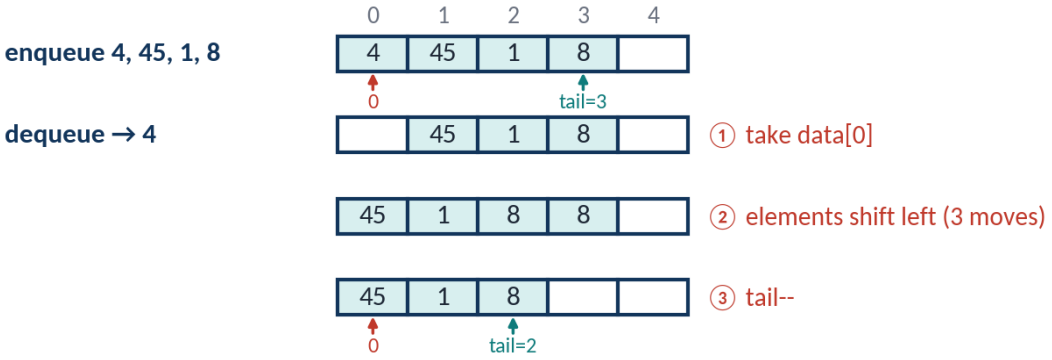


Figure 5.11 The shifting queue: dequeue takes slot 0 (step 1), shifts every other element one place left (step 2), and decrements `tail` (step 3). A queue of n elements costs $n - 1$ moves per dequeue.

This version is correct and easy to understand, and it never wastes a slot. But step 2 is a loop over the whole queue: dequeue costs $n - 1$ element moves for a queue of n elements, so it is $O(n)$, not $O(1)$. A printer queue of ten documents does not care; a router queue of a hundred thousand packets cares very much.

Step 3 is also the step that is easiest to forget. Without `tail--`, the queue believes it still holds four elements after the dequeue — the last one now duplicated — as the middle row of **Figure 5.11** makes visible (`45 1 8 8`). A dequeue that shifts but does not decrement `tail`, and a print loop that stops at `i < tail` instead of `i <= tail`, are two slips that turn up again and again in first attempts at this code.

5.12.2 Second attempt: let the indices move

To make dequeue $O(1)$, stop shifting. Let dequeue simply take `data[head]` and increment `head`. Now both indices move rightward: enqueue moves `tail`, dequeue moves `head`, and the elements stay where they were written. The queue is the stretch of slots from `head` to `tail`, and **Figure 5.12** shows it drifting across the array.

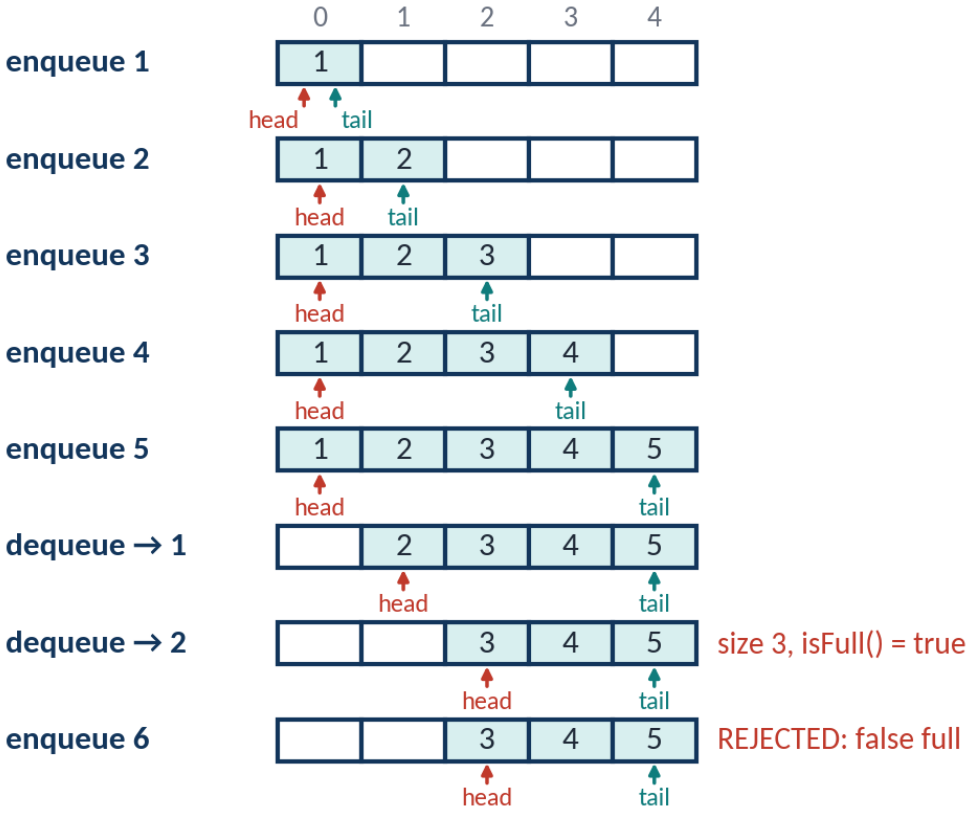


Figure 5.12 The drifting queue. Both indices only move right. After two dequeues, slots 0 and 1 are free, but `tail` has reached the last slot, so `isFull()` reports true and the next enqueue is refused: a *false full*.

Every operation is now $O(1)$. But look at the last two rows. After five enqueues and two dequeues the queue holds three elements in a five-slot array, yet `tail = 4 = MAX - 1`, and the test `isFull()` — "has `tail` reached the last slot?" — says yes. The enqueue of 6 is refused while two slots sit empty. This is the **drifting problem**, also called *false full*: the indices only ever move right, so every slot to the left of `head` is lost forever. Given enough traffic, a drifting queue of capacity 1,000 refuses its next element after the thousandth enqueue, even if it is empty.

Listing 5.11 runs both attempts and prints their states.

Listing 5.11 `09_linear_queue.cpp`: the shifting queue (correct but $O(n)$ per dequeue) and the drifting queue ($O(1)$ but falsely full).

```cpp
// Two linear-array queues: one that shifts on every dequeue, one whose indices drift.
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

const int CAP = 5;

string pad(const string& label) {                       // left-justify a label in 15
columns
    ostringstream o;
    o << left << setw(15) << label;
    return o.str();
}

string cells(const int a[], int lo, int hi) {           // print slots, "." for unused
    string s = "[";
    for (int i = 0; i < CAP; ++i) s += " " + (i >= lo && i <= hi ? to_string(a[i]) :
string("."));
    return s + " ]";
}

// Version 1: the front is always slot 0; dequeue shifts everything one place left.
struct ShiftQueue {
    int data[CAP];
    int tail = -1;                                      // index of the last element
    long moves = 0;                                     // element moves done by shifting
    bool isEmpty() const { return tail == -1; }
    bool isFull()  const { return tail == CAP - 1; }
    bool enqueue(int v) { if (isFull()) return false; data[++tail] = v; return true; }
    int dequeue() {                                     // caller checks isEmpty() first
        int front = data[0];
        for (int i = 0; i < tail; ++i) { data[i] = data[i + 1]; ++moves; }
        --tail;                                         // the step that is easy to forget
        return front;
    }
};
```

Listing 5.11 (Continued) *09_linear_queue.cpp: the shifting queue (correct but O(n) per dequeue) and the drifting queue (O(1) but falsely full).*

```
// Version 2: head and tail both move right and never come back.
struct DriftQueue {
    int data[CAP];
    int head = 0, tail = -1;
    bool isEmpty() const { return head > tail; }
    bool isFull()  const { return tail == CAP - 1; }   // asks the wrong question
    int  size()    const { return tail - head + 1; }
    bool enqueue(int v) { if (isFull()) return false; data[++tail] = v; return true; }
    int  dequeue()      { return data[head++]; }
};

int main() {
    cout << "--- shifting queue, capacity " << CAP << " ---\n";
    ShiftQueue s;
    for (int v : {4, 45, 1, 8}) {
        s.enqueue(v);
        cout << pad("enqueue " + to_string(v)) << cells(s.data, 0, s.tail) << "  tail=" <<
s.tail << "\n";
    }
    for (int k = 0; k < 2; ++k) {
        long before = s.moves;
        int v = s.dequeue();
        cout << pad("dequeue -> " + to_string(v)) << cells(s.data, 0, s.tail) << "  tail="
<< s.tail
             << "  (" << s.moves - before << " elements shifted)\n";
    }

    cout << "\n--- drifting queue, capacity " << CAP << " ---\n";
    DriftQueue d;
    for (int v = 1; v <= 5; ++v) {
        d.enqueue(v);
        cout << pad("enqueue " + to_string(v)) << cells(d.data, d.head, d.tail)
             << "  head=" << d.head << " tail=" << d.tail << "\n";
    }
    for (int k = 0; k < 2; ++k) {
        int v = d.dequeue();
        cout << pad("dequeue -> " + to_string(v)) << cells(d.data, d.head, d.tail)
             << "  head=" << d.head << " tail=" << d.tail << "\n";
    }
    cout << "size " << d.size() << " of " << CAP << ", but isFull() says "
         << (d.isFull() ? "true" : "false") << "\n";
    cout << "enqueue 6 -> " << (d.enqueue(6) ? "ok" : "REJECTED: false full") << "\n";
    return 0;
}
```

```
--- shifting queue, capacity 5 ---
enqueue 4      [ 4 . . . . ]  tail=0
enqueue 45     [ 4 45 . . . ]  tail=1
enqueue 1      [ 4 45 1 . . ]  tail=2
enqueue 8      [ 4 45 1 8 . ]  tail=3
dequeue -> 4   [ 45 1 8 . . ]  tail=2  (3 elements shifted)
dequeue -> 45  [ 1 8 . . . ]  tail=1  (2 elements shifted)

--- drifting queue, capacity 5 ---
enqueue 1      [ 1 . . . . ]  head=0 tail=0
enqueue 2      [ 1 2 . . . ]  head=0 tail=1
enqueue 3      [ 1 2 3 . . ]  head=0 tail=2
enqueue 4      [ 1 2 3 4 . ]  head=0 tail=3
enqueue 5      [ 1 2 3 4 5 ]  head=0 tail=4
dequeue -> 1   [ . 2 3 4 5 ]  head=1 tail=4
dequeue -> 2   [ . . 3 4 5 ]  head=2 tail=4
size 3 of 5, but isFull() says true
enqueue 6 -> REJECTED: false full
```

The shifting queue's output counts its work: three moves for the first dequeue, two for the second. The drifting queue's output ends with the contradiction in black and white: "size 3 of 5, but isFull() says true", followed by the refused enqueue. We want the $O(1)$ operations of the second attempt without its lost slots. The fix is one of the most elegant ideas in this book.

5.13 The Circular Queue

When `tail` reaches the last slot, let it **wrap around** to slot 0 — which a dequeue has probably freed by now — and carry on. Bend the array into a ring, as in **Figure 5.13**, so that slot $N-1$ is followed by slot 0. The indices still only move forward, but "forward" now goes round and round, reusing every slot.

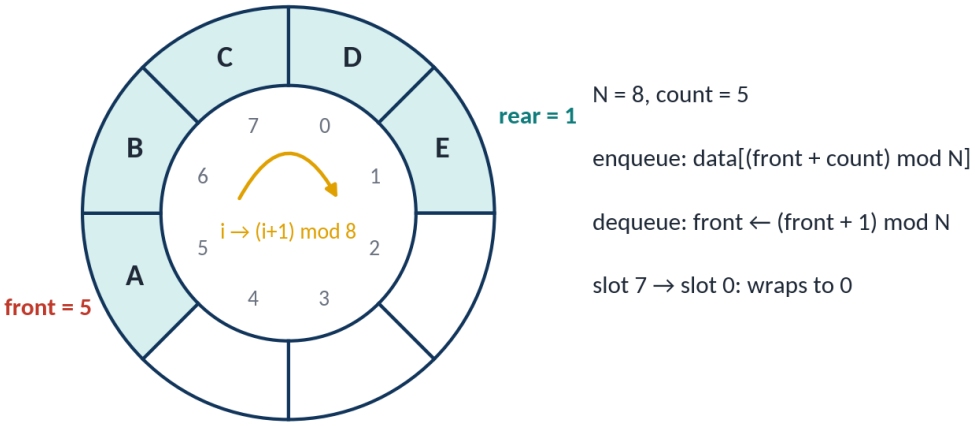


Figure 5.13 A circular queue of capacity $N = 8$ holding A to E. The queue starts at `front` = 5 and runs clockwise through slots 6, 7, 0 and 1: after slot 7 it wraps to slot 0.

5.13.1 Modular arithmetic

The tool that makes the wrap happen is the remainder operator: $a \bmod N$ is the remainder when a is divided by N, always a number from 0 to $N-1$. So instead of `i = i + 1`, a circular queue advances an index with

$$i \leftarrow (i+1) \bmod N. \tag{5.6}$$

For $N = 8$, Equation (5.6) takes $0 \to 1 \to \cdots \to 6 \to 7$, and then $(7+1) \bmod 8 = 8 \bmod 8 = 0$. The index never leaves the array, and no `if` statement is needed to catch the end. In C++, Java and Python the remainder operator is written `%`.

5.13.2 First design: front plus a count

The cleanest design stores `front` (the index of the oldest element) and `count` (how many elements there are). The rear is then not stored but computed. The elements occupy `count` consecutive slots, round the ring, starting at `front`, so the first **free** slot — where the next enqueue writes — is

$$\text{free} = (\text{front} + \text{count}) \bmod N, \tag{5.7}$$

and the last element sits one slot before it, at $(\text{front} + \text{count} - 1) \bmod N$. Emptiness and fullness are trivial: empty when `count = 0`, full when `count = N`. **Pseudocode 5.5** writes the two main operations.

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
```

Figure 5.14 replays the new-arrivals desk with capacity 4. Books B2001 to B2004 fill the array; B2005 is refused and waits at the desk; two books are cataloged, which advances `front` to 2; then B2005 and B2006 arrive and **wrap** into slots 0 and 1, exactly the slots the two dequeues freed. By Equation (5.7) the first free slot for B2005 is $(2 + 2) \bmod 4 = 0$, and for B2006 it is $(2 + 3) \bmod 4 = 1$.

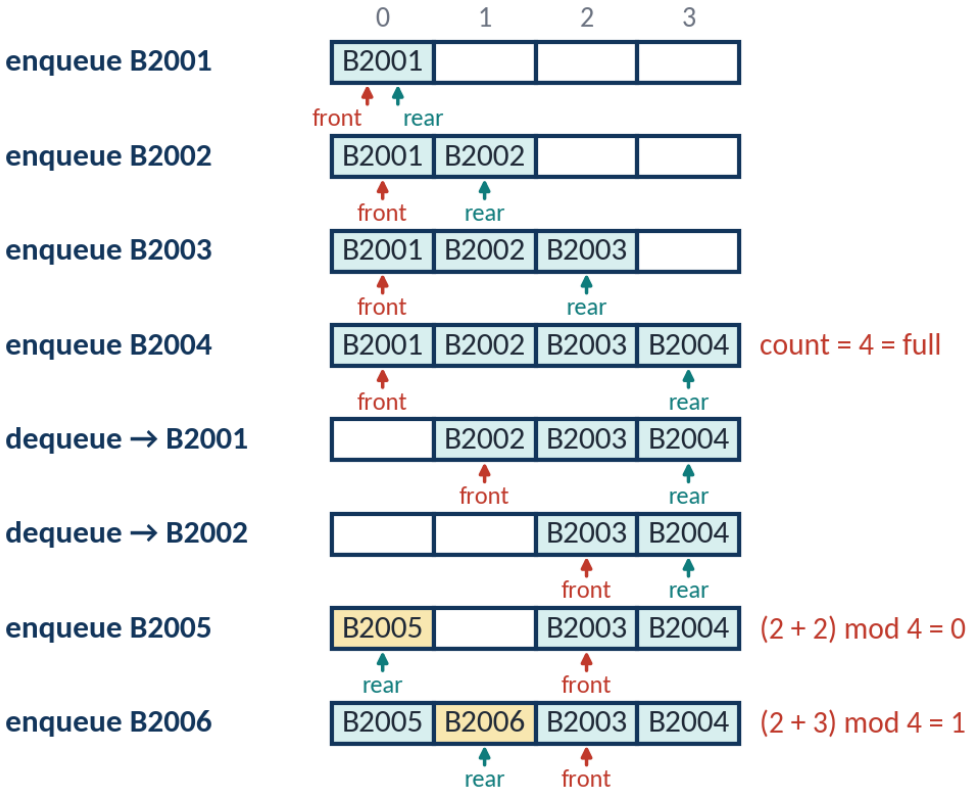


Figure 5.14 The circular queue of **Listing 5.12** with capacity 4. The last two enqueues wrap into slots 0 and 1, so the rear is now to the left of the front — and the queue is still in perfect FIFO order.

Notice the last row: the physical order in the array is B2005 B2006 B2003 B2004, but the *queue* order, reading from `front` round the ring, is B2003 B2004 B2005 B2006. The array has been cut and rejoined, and the queue does not care.

5.13.3 The circular queue in C++, Java and Python

Listing 5.12 implements **Pseudocode 5.5** as a class template. The `slots()` method prints the raw array, with `.....` for free slots, so we can compare the run with **Figure 5.14**.

Listing 5.12 `10_circular_queue.cpp`: a circular queue with a count, as Pustaka Ceria's new-arrivals line.

```
// A circular queue with a count: Pustaka Ceria's new-arrivals cataloging line.
#include <iomanip>
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

template <typename T>
class CircularQueue {
public:
    explicit CircularQueue(int capacity)
        : data_(new T[capacity]), capacity_(capacity), front_(0), count_(0) {}
    ~CircularQueue() { delete[] data_; }
    CircularQueue(const CircularQueue&) = delete;
    CircularQueue& operator=(const CircularQueue&) = delete;

    bool isEmpty() const { return count_ == 0; }
    bool isFull()  const { return count_ == capacity_; }
    int  size()    const { return count_; }
    int  front()   const { return front_; }
    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;
    }
    const T& peek() const {
        if (isEmpty()) throw underflow_error("peek at an empty queue");
        return data_[front_];
    }
    string slots() const {                              // the raw array, "." for a free
slot
        string s = "[";
        for (int i = 0; i < capacity_; ++i) {
            int offset = (i - front_ + capacity_) % capacity_;   // how far i is behind the
front
            s += " " + (offset < count_ ? data_[i] : string("."));
        }
        return s + " ]";
```

Listing 5.12 (Continued) *10_circular_queue.cpp: a circular queue with a count, as Pustaka Ceria's new-arrivals line.*

```
    }

private:
    T*  data_;
    int capacity_;
    int front_;                                          // index of the oldest element
    int count_;                                          // how many elements are stored
};

void state(const string& what, const CircularQueue<string>& q) {
    cout << left << setw(26) << what << q.slots() << "  front=" << q.front()
         << " rear=" << q.rear() << " count=" << q.size() << "\n";
}

int main() {
    cout << "=== Pustaka Ceria: new arrivals (circular queue, capacity 4) ===\n";
    CircularQueue<string> q(4);
    for (const string id : {"B2001", "B2002", "B2003", "B2004", "B2005"}) {
        if (q.enqueue(id)) state("enqueue " + id, q);
        else cout << "enqueue " << id << " -> REJECTED, full (it waits at the desk)\n";
    }
    for (int k = 0; k < 2; ++k) { string id = q.dequeue(); state("dequeue -> " + id, q); }
    for (const string id : {"B2005", "B2006"}) { q.enqueue(id); state("enqueue " + id, q);
}
    cout << "peek -> " << q.peek() << "\n";
    cout << "cataloging order:";
    while (!q.isEmpty()) cout << " " << q.dequeue();
    cout << "\n";
    try {
        q.dequeue();
    } catch (const underflow_error& e) {
        cout << "dequeue on the empty queue -> error: " << e.what() << "\n";
    }
    return 0;
}
```

```
=== Pustaka Ceria: new arrivals (circular queue, capacity 4) ===
enqueue B2001             [ B2001 ..... ..... ..... ]  front=0 rear=0 count=1
enqueue B2002             [ B2001 B2002 ..... ..... ]  front=0 rear=1 count=2
enqueue B2003             [ B2001 B2002 B2003 ..... ]  front=0 rear=2 count=3
enqueue B2004             [ B2001 B2002 B2003 B2004 ]  front=0 rear=3 count=4
enqueue B2005 -> REJECTED, full (it waits at the desk)
dequeue -> B2001          [ ..... B2002 B2003 B2004 ]  front=1 rear=3 count=3
dequeue -> B2002          [ ..... ..... B2003 B2004 ]  front=2 rear=3 count=2
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
dequeue on the empty queue -> error: dequeue from an empty queue
```

The run matches the figure row for row, and the closing line "cataloging order: B2003 B2004 B2005 B2006" confirms that FIFO order survived the wrap. One line deserves a closer look. `rear()` computes `(front_ + count_ - 1 + capacity_) % capacity_`. Why add `capacity_`? Because when the queue is empty and `front_` is 0, `front_ + count_ - 1` is −1, and **in C++ and Java, `-1 % 4` is `-1`**, not 3. Both languages define `%` so that the result has the sign of the left operand. Adding N first keeps the left operand non-negative and the result in range. Python is different — its `%` always returns a result with the sign of the divisor, so `-1 % 4 == 3` — which is why the Python version can omit the `+ N`. Forgetting this difference is one of the classic circular-buffer bugs; Section 5.19 returns to it.

Listing 5.13 and **Listing 5.14** are the Java and Python versions.

Listing 5.13 `CircularQueueDemo.java`: the circular queue as a Java generic class.

```
// A circular queue with a count: Pustaka Ceria's new-arrivals cataloging line.
public class CircularQueueDemo {
    static class CircularQueue<T> {
        private final T[] data;
        private int front = 0;                      // index of the oldest element
        private int count = 0;                      // how many elements are stored

        @SuppressWarnings("unchecked")
        CircularQueue(int capacity) { data = (T[]) new Object[capacity]; }

        boolean isEmpty() { return count == 0; }
        boolean isFull()  { return count == data.length; }
        int size()        { return count; }
        int front()       { return front; }
        int rear()        { return (front + count - 1 + data.length) % data.length; }

        boolean enqueue(T value) {
            if (isFull()) return false;
            data[(front + count) % data.length] = value;   // the slot just after the rear, wrapped
            ++count;
            return true;
        }
        T dequeue() {
            if (isEmpty()) throw new IllegalStateException("dequeue from an empty queue");
            T value = data[front];
            data[front] = null;
            front = (front + 1) % data.length;      // advance and wrap
            --count;
            return value;
        }
        T peek() {
            if (isEmpty()) throw new IllegalStateException("peek at an empty queue");
            return data[front];
        }
        String slots() {                            // the raw array, "." for a free slot
            StringBuilder s = new StringBuilder("[");
            for (int i = 0; i < data.length; ++i) {
                int offset = (i - front + data.length) % data.length;   // how far i is behind the front
                s.append(' ').append(offset < count ? data[i] : ".....");
            }
            return s.append(" ]").toString();
        }
    }
```

Listing 5.13 (Continued) *CircularQueueDemo.java: the circular queue as a Java generic class.*

```
    static void state(String what, CircularQueue<String> q) {
        System.out.printf("%-26s%s  front=%d rear=%d count=%d%n", what, q.slots(),
q.front(), q.rear(), q.size());
    }

    public static void main(String[] args) {
        System.out.println("=== Pustaka Ceria: new arrivals (circular queue, capacity 4)
===");
        CircularQueue<String> q = new CircularQueue<>(4);
        for (String id : new String[]{"B2001", "B2002", "B2003", "B2004", "B2005"}) {
            if (q.enqueue(id)) state("enqueue " + id, q);
            else System.out.println("enqueue " + id + " -> REJECTED, full (it waits at the
desk)");
        }
        for (int k = 0; k < 2; ++k) { String id = q.dequeue(); state("dequeue -> " + id,
q); }
        for (String id : new String[]{"B2005", "B2006"}) { q.enqueue(id); state("enqueue "
+ id, q); }
        System.out.println("peek -> " + q.peek());
        StringBuilder order = new StringBuilder("cataloging order:");
        while (!q.isEmpty()) order.append(' ').append(q.dequeue());
        System.out.println(order);
        try {
            q.dequeue();
        } catch (IllegalStateException e) {
            System.out.println("dequeue on the empty queue -> error: " + e.getMessage());
        }
    }
}
```

Listing 5.14 `10_circular_queue.py`: the circular queue in Python.

```python
# A circular queue with a count: Pustaka Ceria's new-arrivals cataloging line.


class CircularQueue:
    def __init__(self, capacity):
        self._data = [None] * capacity
        self._front = 0                  # index of the oldest element
        self._count = 0                  # how many elements are stored

    def is_empty(self):
        return self._count == 0

    def is_full(self):
        return self._count == len(self._data)

    def size(self):
        return self._count

    def front(self):
        return self._front

    def rear(self):
        return (self._front + self._count - 1) % len(self._data)

    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

    def peek(self):
        if self.is_empty():
            raise IndexError("peek at an empty queue")
        return self._data[self._front]
```

Listing 5.14 (Continued) *10_circular_queue.py: the circular queue in Python.*

```
    def slots(self):                    # the raw array, "." for a free slot
        n = len(self._data)
        cells = []
        for i in range(n):
            offset = (i - self._front) % n   # how far i is behind the front
            cells.append(self._data[i] if offset < self._count else ".....")
        return "[ " + " ".join(cells) + " ]"


def state(what, q):
    print(f"{what:<26}{q.slots()}  front={q.front()} rear={q.rear()} count={q.size()}")


def main():
    print("=== Pustaka Ceria: new arrivals (circular queue, capacity 4) ===")
    q = CircularQueue(4)
    for book_id in ("B2001", "B2002", "B2003", "B2004", "B2005"):
        if q.enqueue(book_id):
            state(f"enqueue {book_id}", q)
        else:
            print(f"enqueue {book_id} -> REJECTED, full (it waits at the desk)")
    for _ in range(2):
        book_id = q.dequeue()
        state(f"dequeue -> {book_id}", q)
    for book_id in ("B2005", "B2006"):
        q.enqueue(book_id)
        state(f"enqueue {book_id}", q)
    print(f"peek -> {q.peek()}")
    order = []
    while not q.is_empty():
        order.append(q.dequeue())
    print("cataloging order: " + " ".join(order))
    try:
        q.dequeue()
    except IndexError as e:
        print(f"dequeue on the empty queue -> error: {e}")


if __name__ == "__main__":
    main()
```

All three print the same fourteen lines.

5.13.4 Second design: leave one slot empty

Many textbooks and much real code — including the classic presentation in Cormen et al. [1] — store two indices and no count: `front`, the index of the oldest element, and `rear`, the index of the **next free slot**. Then enqueue writes at `rear` and advances it with Equation (5.6), and dequeue reads at `front` and advances it. The number of elements is the distance from `front` to `rear` round the ring:

$$\text{size} = (\text{rear} - \text{front} + N) \bmod N. \tag{5.8}$$

The `+ N` in Equation (5.8) again protects against a negative left operand when `rear` has wrapped to the left of `front`. But now there is a real problem, and **Figure 5.15** shows it. An empty queue has `front = rear` (both point at the same free slot). Fill all N slots and `rear` goes all the way round the ring and lands on `front` again — so a **full** queue also has `front = rear`. The two indices cannot tell a full queue from an empty one.

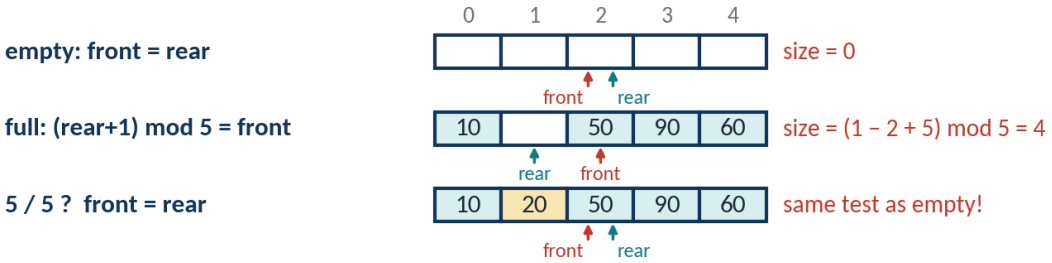


Figure 5.15 Without a count, `front = rear` would mean both "empty" (top) and "all five slots used" (bottom). Keeping one slot always free (middle) makes the full state distinct: the queue is full when advancing `rear` would reach `front`.

The standard cure is to **never fill the last slot**. Declare the queue full when one more enqueue would make `rear` catch `front`:

$$\text{full} \Leftrightarrow (\text{rear} + 1) \bmod N = \text{front}, \qquad \text{empty} \Leftrightarrow \text{rear} = \text{front}. \tag{5.9}$$

With Equation (5.9), an array of N slots holds at most $N - 1$ elements. That is the price: one slot. **Table 5.6** compares the two designs.

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`

The one-empty-slot design has a real advantage that the table's last row hints at: when one thread only enqueues and another only dequeues, each thread writes only its own index, so the two never write the same variable. With a shared `count`, both would have to update it. That is why ring buffers in audio drivers, network cards and inter-thread message passing often use this design.

Listing 5.15 implements it and replays the enqueue 70/80/50 sequence of **Figure 5.10**, this time with the real indices, then keeps going until the queue is full.

Listing 5.15 `11_circular_queue_slot.cpp`: a circular queue that leaves one slot empty and computes its size from the two indices.

```
// A circular queue WITHOUT a count: one slot is always left empty.
#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

const int N = 5;                                        // array length; holds at most N -
1

struct SlotQueue {
    int data[N];
    int front = 0;                                      // index of the oldest element
    int rear  = 0;                                      // index of the NEXT free slot
    bool isEmpty() const { return front == rear; }
    bool isFull()  const { return (rear + 1) % N == front; }
    int  size()    const { return (rear - front + N) % N; }
    bool enqueue(int v) {
        if (isFull()) return false;
        data[rear] = v;
        rear = (rear + 1) % N;
        return true;
    }
    int dequeue() {                                     // caller checks isEmpty() first
        int v = data[front];
        front = (front + 1) % N;
        return v;
    }
};

void state(const string& what, const SlotQueue& q) {
    string s = "[";
    for (int i = 0; i < N; ++i) {
        bool used = (i - q.front + N) % N < q.size();
        s += " " + (used ? to_string(q.data[i]) : string(".."));
    }
    cout << left << setw(14) << what << s << " ]  front=" << q.front << " rear=" << q.rear
         << "  size=(" << q.rear << "-" << q.front << "+" << N << ") mod " << N << " = " <<
q.size()
         << (q.isFull() ? "  FULL" : q.isEmpty() ? "  EMPTY" : "") << "\n";
}
```

Listing 5.15 (Continued) *11_circular_queue_slot.cpp: a circular queue that leaves one slot empty and computes its size from the two indices.*

```
int main() {
    SlotQueue q;
    state("start", q);
    for (int v : {70, 80, 50}) { q.enqueue(v); state("enqueue " + to_string(v), q); }
    for (int k = 0; k < 2; ++k) { int v = q.dequeue(); state("dequeue -> " + to_string(v),
q); }
    for (int v : {90, 60, 10}) { q.enqueue(v); state("enqueue " + to_string(v), q); }
    cout << "enqueue 20    -> " << (q.enqueue(20) ? "ok" : "REJECTED: full with 4 of 5
slots used") << "\n";
    while (!q.isEmpty()) { int v = q.dequeue(); state("dequeue -> " + to_string(v), q); }
    return 0;
}
```

```
start         [ .. .. .. .. .. ]  front=0 rear=0  size=(0-0+5) mod 5 = 0  EMPTY
enqueue 70    [ 70 .. .. .. .. ]  front=0 rear=1  size=(1-0+5) mod 5 = 1
enqueue 80    [ 70 80 .. .. .. ]  front=0 rear=2  size=(2-0+5) mod 5 = 2
enqueue 50    [ 70 80 50 .. .. ]  front=0 rear=3  size=(3-0+5) mod 5 = 3
dequeue -> 70 [ .. 80 50 .. .. ]  front=1 rear=3  size=(3-1+5) mod 5 = 2
dequeue -> 80 [ .. .. 50 .. .. ]  front=2 rear=3  size=(3-2+5) mod 5 = 1
enqueue 90    [ .. .. 50 90 .. ]  front=2 rear=4  size=(4-2+5) mod 5 = 2
enqueue 60    [ .. .. 50 90 60 ]  front=2 rear=0  size=(0-2+5) mod 5 = 3
enqueue 10    [ 10 .. 50 90 60 ]  front=2 rear=1  size=(1-2+5) mod 5 = 4  FULL
enqueue 20    -> REJECTED: full with 4 of 5 slots used
dequeue -> 50 [ 10 .. .. 90 60 ]  front=3 rear=1  size=(1-3+5) mod 5 = 3
dequeue -> 90 [ 10 .. .. .. 60 ]  front=4 rear=1  size=(1-4+5) mod 5 = 2
dequeue -> 60 [ 10 .. .. .. .. ]  front=0 rear=1  size=(1-0+5) mod 5 = 1
dequeue -> 10 [ .. .. .. .. .. ]  front=1 rear=1  size=(1-1+5) mod 5 = 0  EMPTY
```

Each line prints Equation (5.8) with the actual numbers substituted, so you can check the arithmetic: after `enqueue 60`, `rear` has wrapped to 0 and the size is $(0 - 2 + 5) \bmod 5 = 3$. After `enqueue 10` the size is 4 and $(1 + 1) \bmod 5 = 2 =$ front, so Equation (5.9) reports FULL, and `enqueue 20` is refused with slot 1 still empty — on purpose this time. The four dequeues then drain the queue in FIFO order, 50, 90, 60, 10, and the last line shows `front = rear = 1`: empty.

5.14 The Deque

A stack works at one end; a queue adds at one end and removes at the other. The natural generalization allows adding **and** removing at **both** ends. It is called a *double-ended queue*, abbreviated *deque* and pronounced “deck”; Knuth credits the name to E. J. Schweppe [2].

> **DEFINITION.** A *deque* supports `pushFront`, `pushBack`, `popFront`, `popBack`, `peekFront` and `peekBack`, together with `isEmpty` and `size`. Used only with `pushBack` and `popBack` it is a stack; used only with `pushBack` and `popFront` it is a queue.

A circular array implements a deque with one new idea: `pushFront` must move `front` one slot **backward**, wrapping from slot 0 to slot $N - 1$. Moving backward is subtracting 1, and to keep the left operand of `%` non-negative in C++ and Java we add N first:

$$\text{front} \leftarrow (\text{front} - 1 + N) \bmod N. \tag{5.10}$$

`popBack` needs no index change at all beyond decrementing `count`, because the back element’s position, $(\text{front} + \text{count} - 1) \bmod N$, is computed from `count`. **Figure 5.16** traces a deque of capacity 6.

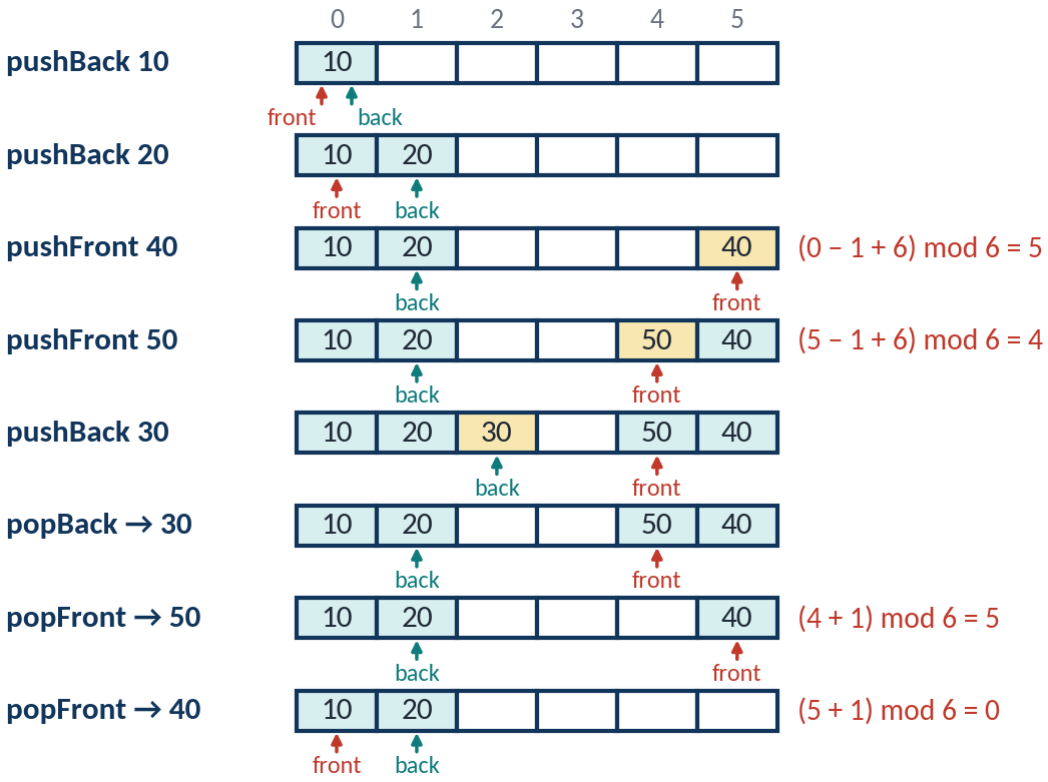


Figure 5.16 A circular-array deque of capacity 6. `pushFront` moves `front` backward with wrap-around, from 0 to 5 and then to 4; `popFront` moves it forward again.

The first `pushFront` is the interesting one: with `front = 0`, Equation (5.10) gives $(0 - 1 + 6) \bmod 6 = 5$, so 40 lands in the *last* slot of the array, which is logically in front of 10. **Listing 5.16** implements the deque and uses it for a classic job: testing whether a phrase is a palindrome by repeatedly comparing and discarding its two ends.

Listing 5.16 `12_array_deque.cpp`: a circular-array deque, and a palindrome test that pops from both ends.

```
// A double-ended queue on a circular array, and a palindrome test that uses it.
#include <cctype>
#include <iomanip>
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

template <typename T>
class ArrayDeque {
public:
    explicit ArrayDeque(int capacity)
        : data_(new T[capacity]), cap_(capacity), front_(0), count_(0) {}
    ~ArrayDeque() { delete[] data_; }
    ArrayDeque(const ArrayDeque&) = delete;
    ArrayDeque& operator=(const ArrayDeque&) = delete;

    bool isEmpty() const { return count_ == 0; }
    bool isFull()  const { return count_ == cap_; }
    int  size()    const { return count_; }
    int  front()   const { return front_; }

    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_;
    }
    void pushBack(const T& v) {
        if (isFull()) throw overflow_error("deque is full");
        data_[(front_ + count_) % cap_] = v;
        ++count_;
    }
    T popFront() {
        if (isEmpty()) throw underflow_error("deque is empty");
        T v = data_[front_];
        front_ = (front_ + 1) % cap_;
        --count_;
        return v;
    }
    T popBack() {
        if (isEmpty()) throw underflow_error("deque is empty");
        --count_;
        return data_[(front_ + count_) % cap_];         // the old back slot
    }
    string slots() const {
```

Listing 5.16 (Continued) *12_array_deque.cpp: a circular-array deque, and a palindrome test that pops from both ends.*

```
        string s = "[";
        for (int i = 0; i < cap_; ++i) {
            bool used = (i - front_ + cap_) % cap_ < count_;
            s += " " + (used ? to_string(data_[i]) : string(".."));
        }
        return s + " ]";
    }

private:
    T*  data_;
    int cap_, front_, count_;
};

bool isPalindrome(const string& text) {
    ArrayDeque<char> d(64);
    for (char c : text)
        if (isalpha(static_cast<unsigned char>(c)))
            d.pushBack(static_cast<char>(tolower(static_cast<unsigned char>(c))));
    while (d.size() > 1)
        if (d.popFront() != d.popBack()) return false;  // compare the two ends, then
discard them
    return true;
}

int main() {
    ArrayDeque<int> d(6);
    auto show = [&](const string& what) {
        cout << left << setw(16) << what << d.slots() << "  front=" << d.front()
             << " size=" << d.size() << "\n";
    };
    show("start");
    d.pushBack(10);  show("pushBack 10");
    d.pushBack(20);  show("pushBack 20");
    d.pushFront(40); show("pushFront 40");
    d.pushFront(50); show("pushFront 50");
    d.pushBack(30);  show("pushBack 30");
    int b = d.popBack();  show("popBack -> " + to_string(b));
    int f = d.popFront(); show("popFront -> " + to_string(f));
    f = d.popFront();     show("popFront -> " + to_string(f));

    cout << "\n";
    for (const string t : {"kasur rusak", "Taat", "racecar", "stack", "A man, a plan, a
canal: Panama"})
        cout << left << setw(34) << ("\"" + t + "\"") << (isPalindrome(t) ? "palindrome" :
"not a palindrome") << "\n";
    return 0;
}
```

```
start           [ .. .. .. .. .. .. ]  front=0 size=0
pushBack 10     [ 10 .. .. .. .. .. ]  front=0 size=1
pushBack 20     [ 10 20 .. .. .. .. ]  front=0 size=2
pushFront 40    [ 10 20 .. .. .. 40 ]  front=5 size=3
pushFront 50    [ 10 20 .. .. 50 40 ]  front=4 size=4
pushBack 30     [ 10 20 30 .. 50 40 ]  front=4 size=5
popBack -> 30   [ 10 20 .. .. 50 40 ]  front=4 size=4
popFront -> 50  [ 10 20 .. .. .. 40 ]  front=5 size=3
popFront -> 40  [ 10 20 .. .. .. .. ]  front=0 size=2

"kasur rusak"                     palindrome
"Taat"                            palindrome
"racecar"                         palindrome
"stack"                           not a palindrome
"A man, a plan, a canal: Panama"  palindrome
```

The palindrome test ignores spaces, punctuation and case, so “kasur rusak” (Indonesian for “broken mattress”, and a famous palindrome) and “A man, a plan, a canal: Panama” both pass, while “stack” fails at the first comparison of `s` with `k`. Each character is pushed once and popped once, so the test runs in $O(n)$.

Deques shine whenever work arrives at one end and is taken from both. A *work-stealing* scheduler gives each processor core a deque of tasks: the core pushes and pops its own tasks at the back (stack-like, for cache locality), while idle cores steal from the front. The *sliding-window maximum* problem — the largest value in every window of k consecutive readings, used for instance to report the peak traffic in each five-minute window — is solved in $O(n)$ with a deque of candidate indices (Exercise 5.14).

5.15 Priority Queues: a Preview

A hospital emergency room does not treat patients in arrival order. A patient with a heart attack who arrived a minute ago is seen before a patient with a sprained ankle who has waited an hour. Each patient has a **priority**, and the next one served is the one with the highest priority, whatever their arrival time. This is a *priority queue*: an ADT with `insert(x, priority)`, `extractMax()` (or `extractMin()`), `peek` and `isEmpty`.

Despite the name, a priority queue is not a queue in the FIFO sense: arrival order matters only to break ties, if at all. The interesting question is how to make both operations fast. **Table 5.7** compares the obvious implementations with the one Chapter 14 builds.

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

Neither naive version is satisfactory: one of the two operations is linear. The binary heap balances them at $O(\lg n)$ each, which for a million elements means about 20 steps instead of a million. All three standard libraries provide a heap-based priority queue, and Section 5.16 shows how to use them now, before Chapter 14 shows how they work.

5.16 Stacks and Queues in the Standard Libraries

Having built every structure by hand, you have earned the right to use the library versions — and you will use them far more often than your own. Each language offers them in a slightly different way, and each has a trap or two.

5.16.1 C++: container adaptors

In C++, `std::stack`, `std::queue` and `std::priority_queue` are *container adaptors*: thin wrappers that take an underlying container and expose only the stack, queue or priority-queue operations. By default `std::stack` and `std::queue` wrap a `std::deque`, and `std::priority_queue` wraps a `std::vector` organized as a max-heap [8], [9]. `std::deque` itself is a full double-ended queue with $O(1)$ insertion and removal at both ends. **Listing 5.17** exercises all four.

Listing 5.17 `13_library.cpp`: `std::stack`, `std::queue`, `std::deque` and `std::priority_queue`.

```cpp
// The standard library's ready-made stack, queue, deque and priority queue.
#include <deque>
#include <iostream>
#include <queue>
#include <stack>
#include <vector>
using namespace std;

int main() {
    stack<int> s;                                       // adapter over deque<int> by default
    for (int v : {1, 2, 3}) s.push(v);
    cout << "stack  : push 1 2 3 -> top " << s.top() << ", pop order:";
    while (!s.empty()) { cout << " " << s.top(); s.pop(); }   // pop() returns nothing
    cout << "\n";

    queue<int> q;                                       // adapter over deque<int> by default
    for (int v : {1, 2, 3}) q.push(v);                  // push = enqueue at the back
    cout << "queue  : push 1 2 3 -> front " << q.front() << ", back " << q.back() << ", pop order:";
    while (!q.empty()) { cout << " " << q.front(); q.pop(); }
    cout << "\n";

    deque<int> d;
    d.push_back(1); d.push_back(2); d.push_front(0); d.push_front(-1);
    cout << "deque  : front to back:";
    for (int v : d) cout << " " << v;
    d.pop_front(); d.pop_back();
    cout << " | after pop_front, pop_back:";
    for (int v : d) cout << " " << v;
    cout << "\n";

    priority_queue<int> pq;                             // a max-heap underneath (Chapter 14)
    for (int v : {5, 1, 4, 2, 3}) pq.push(v);
    cout << "pqueue : push 5 1 4 2 3 -> pop order:";
    while (!pq.empty()) { cout << " " << pq.top(); pq.pop(); }
    cout << "\n";

    priority_queue<int, vector<int>, greater<int>> minpq;   // smallest first
    for (int v : {5, 1, 4, 2, 3}) minpq.push(v);
    cout << "min-pq : push 5 1 4 2 3 -> pop order:";
    while (!minpq.empty()) { cout << " " << minpq.top(); minpq.pop(); }
    cout << "\n";
    return 0;
}
```

```
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
deque  : front to back: -1 0 1 2 | after pop_front, pop_back: 0 1
pqueue : push 5 1 4 2 3 -> pop order: 5 4 3 2 1
min-pq : push 5 1 4 2 3 -> pop order: 1 2 3 4 5
```

Three facts about the C++ adaptors surprise almost everyone the first time:

- **`pop()` returns `void`.** To remove and use the top element you write `x = s.top(); s.pop();`. The standard library was designed this way so that `pop` can never lose an element if copying it throws an exception halfway through.
- **`top()`, `front()` and `pop()` on an empty container are undefined behavior**, not exceptions. The library assumes you checked `empty()` first. Our own classes in Sections 5.4 and 5.13 were stricter.
- **`std::priority_queue` is a max-heap by default.** For smallest-first order, pass `std::greater<int>` as the comparison, as the `min-pq` line does.

5.16.2 Java: `ArrayDeque` for both

In Java, use `ArrayDeque` as both your stack and your queue. The official documentation says that it "is likely to be faster than `Stack` when used as a stack, and faster than `LinkedList` when used as a queue" [10]. The older class `java.util.Stack` extends `Vector`, whose methods are all synchronized and which lets you insert into the middle of the "stack"; its own documentation recommends the `Deque` interface "in preference to this class" [11]. **Listing 5.18** produces the same output as **Listing 5.17**.

Listing 5.18 `Library.java`: `ArrayDeque` as a stack, as a queue and as a deque, and `PriorityQueue` in both orders.

```
// The standard library's ready-made stack, queue, deque and priority queue.
import java.util.ArrayDeque;
import java.util.Collections;
import java.util.PriorityQueue;

public class Library {
    public static void main(String[] args) {
        ArrayDeque<Integer> s = new ArrayDeque<>();     // used as a stack: push/pop/peek
at the head
        for (int v : new int[]{1, 2, 3}) s.push(v);
        StringBuilder line = new StringBuilder("stack  : push 1 2 3 -> top " + s.peek() +
", pop order:");
        while (!s.isEmpty()) line.append(' ').append(s.pop());
        System.out.println(line);

        ArrayDeque<Integer> q = new ArrayDeque<>();     // used as a queue: offer at the
tail, poll at the head
        for (int v : new int[]{1, 2, 3}) q.offer(v);
        line = new StringBuilder("queue  : push 1 2 3 -> front " + q.peek() + ", back " +
q.peekLast() + ", pop order:");
        while (!q.isEmpty()) line.append(' ').append(q.poll());
        System.out.println(line);

        ArrayDeque<Integer> d = new ArrayDeque<>();
        d.addLast(1); d.addLast(2); d.addFirst(0); d.addFirst(-1);
        line = new StringBuilder("deque  : front to back:");
        for (int v : d) line.append(' ').append(v);
        d.pollFirst(); d.pollLast();
        line.append(" | after pop_front, pop_back:");
        for (int v : d) line.append(' ').append(v);
        System.out.println(line);

        PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());   //
largest first
        for (int v : new int[]{5, 1, 4, 2, 3}) pq.add(v);
        line = new StringBuilder("pqueue : push 5 1 4 2 3 -> pop order:");
        while (!pq.isEmpty()) line.append(' ').append(pq.poll());
        System.out.println(line);

        PriorityQueue<Integer> minpq = new PriorityQueue<>();                           //
smallest first
        for (int v : new int[]{5, 1, 4, 2, 3}) minpq.add(v);
        line = new StringBuilder("min-pq : push 5 1 4 2 3 -> pop order:");
        while (!minpq.isEmpty()) line.append(' ').append(minpq.poll());
        System.out.println(line);
    }
}
```

Two details matter. As a stack, `ArrayDeque` pushes and pops at its **head** — so iterating over it goes from top to bottom — while as a queue it offers at the tail and polls at the head. And `ArrayDeque`

rejects `null` elements, so `poll()` returning `null` can unambiguously mean "empty". Java's `PriorityQueue` is a min-heap by default; `Collections.reverseOrder()` flips it.

5.16.3 Python: `list`, `collections.deque`, `queue` and `heapq`

Python has no class called `Stack`. A `list` is the stack: `append` pushes and `pop()` pops, both at the end of the list and both $O(1)$ amortized. For a queue, **do not** use `list.pop(0)`: removing the first element shifts every other element left, exactly like the shifting queue of Section 5.12.1, and the Python documentation warns that `pop(0)` and `insert(0, v)` incur "O(n) memory movement costs". Use `collections.deque`, which supports appends and pops "from either side of the deque with approximately the same O(1) performance in either direction" [12]. For queues shared between threads, the `queue` module provides `Queue`, `LifoQueue` and `PriorityQueue` with "all the required locking semantics" [13]; and the `heapq` module turns a list into a binary min-heap. **Listing 5.19** is the Python version of the library tour.

Listing 5.19 `13_library.py`: a list as a stack, `collections.deque` as a queue and deque, and `heapq` as a priority queue.

```
# The standard library's ready-made stack, queue, deque and priority queue.
import heapq
from collections import deque


def main():
    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)))

    d = deque()
    d.append(1)
    d.append(2)
    d.appendleft(0)
    d.appendleft(-1)
    line = "deque  : front to back: " + " ".join(map(str, d))
    d.popleft()
    d.pop()
    print(line + " | after pop_front, pop_back: " + " ".join(map(str, d)))

    pq = []                                 # heapq is a min-heap: store -v to get largest
first
    for v in (5, 1, 4, 2, 3):
        heapq.heappush(pq, -v)
    out = []
    while pq:
        out.append(-heapq.heappop(pq))
    print("pqueue : push 5 1 4 2 3 -> pop order: " + " ".join(map(str, out)))
```

Listing 5.19 (Continued) *13_library.py: a list as a stack, collections.deque as a queue and deque, and heapq as a priority queue.*

```
    minpq = []                              # smallest first, heapq's natural order
    for v in (5, 1, 4, 2, 3):
        heapq.heappush(minpq, v)
    out = []
    while minpq:
        out.append(heapq.heappop(minpq))
    print("min-pq : push 5 1 4 2 3 -> pop order: " + " ".join(map(str, out)))


if __name__ == "__main__":
    main()
```

Because `heapq` only provides a min-heap, the largest-first line pushes $-v$ and negates again when popping — the standard idiom. All three programs print the five lines shown after **Listing 5.17**.

Table 5.8 collects the vocabulary.

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`

5.17 Choosing Among Them: Costs at a Glance

Table 5.9 gathers the costs of every structure in this chapter. "Amortized" means averaged over a long sequence of operations, as derived in Equation (5.5).

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

The table leads to a short decision rule. If the problem needs "the most recent first" — undo, backtracking, nested structure, function calls — use a stack. If it needs "the oldest first" — fairness, buffering, level-by-level search — use a queue, and implement it as a circular buffer, never as a drifting one. If it needs both ends, use a deque. If it needs "the most important first", use a priority queue. And in production code, use the library's version unless you have measured a reason not to.

5.18 Profitina in Practice

Profitina is an AI Visibility (GEO) platform, in production at app.profitina.com: it scans a brand's website, measures how ready the site is to be quoted by AI answer engines, and helps fix what it finds [14]. The product is written in Python on FastAPI, runs its AI work on a local Qwen2.5 model, and serves a real-time community feed over Server-Sent Events. Nothing in it is a textbook exercise, and yet the structures of this chapter appear throughout, because the problems it solves have the same shapes as the ones we have studied. What follows describes those shapes and the class of solution each one calls for.

Structured data must be balanced. Answer engines read schema.org structured data, which a site embeds as a block of JSON-LD. One missing `}` in that block makes the whole block unreadable, and the site's structured data silently disappears from the engines' view. The very first thing any JSON parser does is the check of **Pseudocode 5.2**: every `{` and `[` must be closed by its partner, in the right order. A technical assistant that helps a site owner write structured data has to report such an error as precisely as **Listing 5.7** does — which bracket, at which position, failed to match what.

A crawl is a queue. To scan a site, a crawler starts at the home page, discovers the links on it, and must decide which page to fetch next. Keeping the discovered-but-unvisited pages in a FIFO queue fetches them level by level — the home page, then everything one click away, then everything two clicks away — which is exactly the order in which an answer engine's own crawler is likely to value them. This is breadth-first search, and Chapter 13 builds it on a graph; the queue at its heart is the circular queue of Section 5.13 or a library deque.

Slow work goes into a job queue. Running a 7-billion-parameter model locally is slow compared with answering a web request. Work such as drafting GEO content or giving an optional AI opinion on a posted link cannot be done while the user's request waits; it is placed in a queue and taken out by a worker in arrival order. The FIFO discipline is what makes this fair — nobody's request is overtaken by a later one — and the bound on the queue is what protects the server: when the queue is full, the application must say so rather than accept work it cannot finish, the same honest choice our `push` and `enqueue` make by returning `false`.

Scheduled re-scans are a priority queue. Each brand is re-scanned automatically at a cadence set by its subscription tier. The scheduler's question is always "which scan is due soonest?", and the answer is the minimum of a changing set of due times — a priority queue keyed by time, the structure previewed in Section 5.15 and built in Chapter 14. A plain FIFO queue would be wrong here: a weekly scan queued early must not delay a daily scan that is due first.

A live feed needs bounded buffers. A real-time community feed pushes each new post to every connected browser. If one reader's connection is slow, the posts waiting to be sent to that reader must be held somewhere — and held in a buffer of *bounded* size, or one slow phone could make the server run out of memory. A ring buffer with the full/empty rules of Equation (5.9) is the classic structure for exactly this situation, and it also leaves the server a clear policy decision to make when the buffer fills: drop the oldest undelivered item, or ask the reader to reconnect and catch up.

An optimistic post is a one-level undo. When a user posts to the community, the composer clears at once and the post appears immediately, before the server has confirmed it; if sending fails, the text is restored. That is the undo pattern of Section 5.7 with a stack of depth one: the text is saved before the optimistic change, and popped back if the change must be undone.

The feedback pipeline is a queue of queues. A user's suggestion travels through submit, curate, dataset, train-and-measure, and release. Within each stage, items are handled in the order they arrived, and no item skips a stage — a sequence of FIFO queues in which the output of one stage is the input of the next.

The runnable practice for these ideas uses the Profitina Laundry teaching case study: Exercise 5.15 asks you to design its order counter.

5.19 Common Mistakes, and How to Avoid Them

1. **Incrementing and writing in the wrong order.** `data[top++] = x` with *top* starting at -1 writes to `data[-1]`. With the "-1 means empty" convention, push is `data[++top] = x` and pop is `return data[top--]`. Write the two lines of **Pseudocode 5.1** once, test them on a stack of capacity 1, and copy them from then on.
2. **Mixing conventions for `top`.** Initializing `top = 0` and testing `top == -1` for emptiness (or the reverse) makes a stack that is never empty or never full. Pick one convention, write Equation (5.4) in a comment next to the field, and keep to it.
3. **Popping without checking.** `std::stack::top()` and `pop()` on an empty stack are undefined behavior in C++; the program may crash, or worse, appear to work. Check `empty()` first, or wrap the library stack in a function that checks for you (Exercise 5.18).
4. **Swapping the operands in postfix evaluation.** The first pop is the *right* operand. `8 3 -` is $8 - 3$, not $3 - 8$. Always test an evaluator on subtraction and division, never only on `+` and `*`.
5. **Treating `^` as left-associative in the shunting-yard algorithm.** Popping on equal precedence for `^` turns `2^3^2` into $(2^3)^2 = 64$ instead of 512. The equal-precedence pop in line 12 of **Pseudocode 5.3** applies only to left-associative operators.
6. **Forgetting the parenthesis barrier.** An incoming operator must never pop a `(`; a `)` must pop *up to* its `(` and then discard it. A missing check turns `(3+2)*5` into `3 2 5 * +`.
7. **Forgetting `tail--` in the shifting queue**, and printing with `i < tail` instead of `i <= tail`. The first duplicates the last element; the second hides it. Trace one dequeue by hand, as in **Figure 5.11**.
8. **Shipping a drifting queue.** A queue whose `head` and `tail` only increase will declare itself full after `capacity` enqueues, however many dequeues happened. Every array queue must wrap: use Equation (5.6).
9. **Negative remainders.** In C++ and Java, `(i - 1) % N` is -1 when `i` is 0, and `data[-1]` is out of bounds. Write `(i - 1 + N) % N`, as in Equation (5.10). Python's `%` is always non-negative, so code ported *from* Python is especially at risk.
10. **Testing `front == rear` for both full and empty.** With two indices and no count, a completely full array looks exactly like an empty one (**Figure 5.15**). Either keep a count or leave one slot empty and use Equation (5.9).
11. **Using `list.pop(0)` as a Python queue or `java.util.Stack` as a Java stack.** The first is $O(n)$ per dequeue; the second is a legacy synchronized class that also allows access to the middle. Use `collections.deque` and `ArrayDeque`.
12. **Keeping the redo stack after a new edit.** A new edit must empty the redo stack; otherwise redo resurrects a state from an abandoned history.
13. **Returning the address of a local variable.** A local variable lives in its function's stack frame, which is popped on return. A pointer or reference to it dangles immediately, and the next function call overwrites the memory.

14. **Copying a class that owns an array.** A stack or queue class that allocates with `new[]` and is copied member by member ends with two objects deleting the same array. Delete the copy operations (as our classes do) or implement them properly, which Chapter 6 explains.

5.20 Chapter Summary

- A **stack** adds and removes at one end, the top: last in, first out. Its operations `push`, `pop`, `peek`, `isEmpty` and `isFull` each run in $O(1)$ time on an array with a `top` index, where $top = -1$ means empty and size $= top + 1$. Overflow (push on full) and underflow (pop on empty) must be handled deliberately.
- The **Towers of Hanoi** is three stacks with a size rule. The minimum number of moves satisfies $M(n) = 2M(n-1) + 1$, so $M(n) = 2^n - 1$; it can be solved by a loop that uses only stack operations, and Chapter 10 solves it by recursion.
- A **doubling array** removes the capacity limit at an amortized cost of $O(1)$ per push, because $1 + 2 + \cdots + 2^k < 2n$. A **linked stack** removes it with one node per element.
- Stacks **match brackets** (push openers, pop and compare at closers, finish empty), implement **undo and redo** (two stacks; a new edit clears redo), **convert infix to postfix** (Dijkstra's shunting-yard algorithm, driven by precedence and associativity) and **evaluate postfix** (push operands; an operator pops y then x and pushes x op y), and they are the machinery of **function calls** (one frame per call; recursion is frames of the same function; a runaway recursion overflows the stack).
- A **queue** adds at the rear and removes at the front: first in, first out. A linear array implementation either **shifts** on every dequeue ($O(n)$) or **drifts** until it reports a false full.
- The **circular queue** wraps its indices with $i \leftarrow (i + 1) \bmod N$, giving $O(1)$ operations with no lost slots. It tells full from empty either with a **count** or by leaving **one slot empty** (full when $(\text{rear} + 1) \bmod N = \text{front}$). In C++ and Java, keep the left operand of `%` non-negative.
- A **deque** adds and removes at both ends; on a circular array, `pushFront` uses $(\text{front} - 1 + N) \bmod N$. A **priority queue** serves the highest priority first; the heap of Chapter 14 makes it $O(\lg n)$ per operation.
- In practice: `std::stack`, `std::queue`, `std::deque` and `std::priority_queue` in C++ (`pop` returns `void`; empty access is undefined); `ArrayDeque` and `PriorityQueue` in Java; `list`, `collections.deque`, `queue.Queue` and `heapq` in Python.

5.21 Exercises

5.1 Starting from an empty array stack of capacity 5, perform: push 5, push 3, pop, push 7, push 2, pop, pop, push 9, push 4, peek. List the value returned by every pop and peek, the contents from bottom to top at the end, and the value of `top` at the end.

5.2 (a) How many moves does the Towers of Hanoi need for 20 disks, and how long does that take at one move per second? (b) In the solution for $n = 4$ disks, on which move numbers does disk 1 move, and on which single move does disk 4 move? (c) Generalize: on which moves does disk k move?

5.3 The numbers 1, 2, 3, 4 are pushed onto an empty stack in that order, but pops may be performed at any time, and each popped number is printed. Which of the following output sequences are possible? For each possible one, give the sequence of pushes and pops; for each impossible one, explain why. (a) 1 2 3 4 (b) 4 3 2 1 (c) 2 4 3 1 (d) 3 1 4 2 (e) 1 4 2 3.

5.4 Add three member functions to the `ArrayStack` of **Listing 5.2**: `search(x)`, which returns the distance of the topmost copy of `x` from the top (0 for the top itself) or -1; `edit(k, x)`, which replaces the element k positions below the top; and `stats()`, which prints the total, average, largest and smallest element of an integer stack. Should these functions exist in a stack at all? Discuss.

5.5 Trace **Pseudocode 5.2** on `{()}`, on `(((` and on `)(`. Give the message each input produces.

5.6 Convert to postfix with **Pseudocode 5.3**, showing the stack after every token, then evaluate the first one: (a) `9-2*3+1` (b) `(a+b)*(c-d)/e` (c) `a^b^c*d`.

5.7 Evaluate the postfix expression `6 2 3 + - 3 8 2 / + * 2 ^ 3 +` with **Pseudocode 5.4**, showing the stack after every token.

5.8 In prefix notation the operator comes first. Explain how to evaluate a prefix expression with one stack by scanning it from **right to left**, state which pop is the left operand, and evaluate `- * 2 3 4` and `* - 7 3 + 1 1`.

5.9 Two stacks must share one array of N slots so that neither overflows until all N slots are used. Design the representation and the four operations `push1`, `push2`, `pop1`, `pop2`, and state the overflow test.

5.10 Implement a queue using two stacks and no other storage. Show that although a single `dequeue` may cost $O(n)$, any sequence of m operations costs $O(m)$ in total, so each operation is $O(1)$ amortized.

5.11 A circular queue in the count design of Section 5.13.2 has $N = 6$, `front` = 4 and `count` = 0. Perform: enqueue A, enqueue B, enqueue C, dequeue, enqueue D, enqueue E, enqueue F, enqueue G, enqueue H. After each operation give `front`, `count`, the slot written or read, and whether the operation succeeded.

5.12 A circular queue in the one-empty-slot design has $N = 8$, `front` = 6 and `rear` = 2. (a) How many elements does it hold? (b) Which slots do they occupy, in queue order? (c) How many more enqueues succeed before it is full? (d) What are `front` and `rear` after two dequeues and then as many enqueues as succeed?

5.13 A student proposes to cure the drifting queue of Section 5.12.2 as follows: whenever a dequeue makes the queue empty, reset `head = 0` and `tail = -1`. Does this cure the false full? Give a sequence of operations that still fails, or argue that none exists.

5.14 The *sliding-window maximum* of an array $A[0..n-1]$ for window size k is the list of $\max(A[i..i+k-1])$ for $i = 0, \dots, n-k$. Describe an $O(n)$ algorithm that keeps a deque of indices, and trace it on $A = [1,3,-1,-3,5,3,6,7]$ with $k = 3$.

5.15 At the Profitina Laundry counter, orders are served in arrival order, but *express* orders must be served before all regular orders, and among express orders arrival order must still be respected. (a) Explain why a single FIFO queue is not enough. (b) Design a solution with two queues. (c) Design a solution with one priority queue, and explain what the priority must contain so that arrival order is kept within each class.

5.16 Design a stack that supports `push`, `pop` and `getMin` (the smallest element currently in the stack), all in $O(1)$ time.

5.17 A recursive function's stack frame occupies 64 bytes. Roughly how deep can the recursion go before it overflows a 1 MB call stack? And an 8 MB one? Why is the real answer smaller?

5.18 `std::stack::pop` returns `void`, and `top()` on an empty stack is undefined behavior. Write a small C++ function `template <typename T> T popValue(std::stack<T>& s)` that returns the removed element and throws `std::underflow_error` on an empty stack. Why did the designers of the standard library not make `pop` do this?

5.22 Solutions to the Exercises

5.1 The operations and their results, with the stack written bottom to top:

- push 5 → [5], $top = 0$; push 3 → [5 3], $top = 1$.
- pop → returns **3**; [5], $top = 0$.
- push 7 → [5 7]; push 2 → [5 7 2], $top = 2$.
- pop → returns **2**; pop → returns **7**; [5], $top = 0$.
- push 9 → [5 9]; push 4 → [5 9 4], $top = 2$.
- peek → returns **4**; nothing changes.

Final contents, bottom to top: 5 9 4, with $top = 2$ and size $top + 1 = 3$ by Equation (5.4). Capacity 5 was never reached.

5.2 (a) By Equation (5.2), $M(20) = 2^{20} - 1 = 1{,}048{,}575$ moves. At one move per second that is $1{,}048{,}575/86{,}400 \approx 12.1$ days.

(b) For $n = 4$ (even), disk 1 moves on every odd-numbered move: 1, 3, 5, 7, 9, 11, 13, 15 — eight times. Disk 4 moves exactly once, on move $2^3 = 8$, the middle of the 15 moves: the first $7 = M(3)$ moves clear three disks away, move 8 moves disk 4, and the last 7 moves rebuild the three-disk tower on top of it.

(c) Disk k first moves on move 2^{k-1} and then every 2^k moves: on moves $2^{k-1}, 3 \cdot 2^{k-1}, 5 \cdot 2^{k-1}, \ldots$ So disk 1 moves on odd moves, disk 2 on moves 2,6,10, ..., disk 3 on moves 4,12,20, ... In total disk k moves 2^{n-k} times, and $\sum_{k=1}^{n} 2^{n-k} = 2^n - 1$, which is Equation (5.2) again.

5.3 A number can be printed only when it is on top, and the numbers enter in increasing order.

(a) 1 2 3 4: possible — push 1, pop; push 2, pop; push 3, pop; push 4, pop.

(b) 4 3 2 1: possible — push 1, 2, 3, 4, then pop four times.

(c) 2 4 3 1: possible — push 1, push 2, pop (2); push 3, push 4, pop (4); pop (3); pop (1).

(d) 3 1 4 2: **impossible.** To print 3 first, 1 and 2 must already have been pushed, and they lie under 3 with 2 above 1. After 3 is popped, the top is 2, so 1 cannot come out before 2.

(e) 1 4 2 3: **impossible.** After 1 is printed and 4 is printed, 2 and 3 must both be on the stack with 3 above 2, so 3 must be printed before 2.

The general rule: a sequence is impossible exactly when it contains three values $a < b < c$ printed in the order c, a, b.

5.4 One possible implementation, written as members of `ArrayStack<T>`:

```
int search(const T& x) const {              // distance from the top, or -1
    for (int i = top_; i >= 0; --i)
        if (data_[i] == x) return top_ - i;
    return -1;
}
bool edit(int k, const T& x) {              // k = 0 is the top
    if (k < 0 || k > top_) return false;
    data_[top_ - k] = x;
    return true;
}
// stats() for ArrayStack<int>: walk i = 0..top_, accumulating
// total, max and min; average = total / (double)(top_ + 1),
// guarding the empty case first.
```

`search` and `stats` are $O(n)$ and `edit` is $O(1)$. They are useful for debugging and for exercises, but strictly they break the stack ADT: a stack promises access to the top only. A class that offers them is really an array with a stack-like interface. Real libraries follow the same line — `std::stack` offers no search at all, while Java's legacy `Stack.search` exists precisely because `Stack` was built on `Vector`. If you need to search, ask whether a stack is the right structure.

5.5

- `{(})`: push `{`, push `(`; at position 2 the closer `}` finds `(` on top: **"error at 2: '}' cannot close '('"**.
- `(((`: three pushes, the scan ends with three brackets on the stack: **"error at end: 3 bracket(s) never closed, top is '('"**.
- `)(`: at position 0 the closer `)` finds the stack empty: **"error at 0: ')' has no opening partner"**. The algorithm stops at the first error, so the unclosed `(` is never reported.

5.6 (a) `9-2*3+1`: `9` → out `9`; `-` → push; `2` → out `9 2`; `*` (above `-`, higher) → push, stack `- *`; `3` → out `9 2 3`; `+` → pop `*` (higher), pop `-` (equal, left-assoc.), push `+`, out `9 2 3 * -`; `1` → out `9 2 3 * - 1`; end → pop `+`. Postfix **`9 2 3 * - 1 +`**. Evaluation: push 9, 2, 3; `*` → $2 \cdot 3 = 6$, stack `9 6`; `-` → $9 - 6 = 3$; push 1; `+` → \$3 + 1 = \$ **4**, which matches $9 - 6 + 1$.

(b) `(a+b)*(c-d)/e`: `(` push; `a` out; `+` push (above `(`); `b` out; `)` pops `+`, drops `(` → out `a b +`; `*` push; `(` push; `c` out; `-` push; `d` out; `)` pops `-` → out `a b + c d -`; `/` → pop `*` (equal, left) and push `/`, out `a b + c d - *`; `e` out; end pops `/`. Postfix **`a b + c d - * e /`**.

(c) `a^b^c*d`: `a` out; `^` push; `b` out; `^` meets `^` (equal, **right**-assoc.) → no pop, push, stack `^ ^`; `c` out; `*` → pop `^`, pop `^` (both higher), push `*`, out `a b c ^ ^`; `d` out; end pops `*`. Postfix **`a b c ^ ^ d *`**, meaning $(a^{(b^c)}) \cdot d$.

5.7 Stack after each token (bottom → top):

Token	Stack	Token	Stack
6	6	2	1 3 8 2
2	6 2	/	1 3 4
3	6 2 3	+	1 7
+	6 5	*	7
-	1	2	7 2
3	1 3	^	49
8	1 3 8	3	49 3
		+	52

The result is **52**. At `-` the pops give $y = 5$, $x = 6$, and $x - y = 1$; at `/`, $y = 2$, $x = 8$, $8/2 = 4$.

5.8 Scanning prefix from right to left meets every operand before its operator, just as a left-to-right scan does for postfix. So: push operands; at an operator, pop twice and push the result. Because the scan is reversed, the **first** pop is now the **left** operand x and the second is the right operand y.

`- * 2 3 4` from the right: push 4, push 3, push 2; `*` pops $x = 2, y = 3$, pushes 6; `-` pops $x = 6, y = 4$, pushes **2**. (Infix: $2 \cdot 3 - 4$.)

`* - 7 3 + 1 1` from the right: push 1, push 1; `+` → 2; push 3, push 7; `-` pops $x = 7, y = 3 \to 4$; `*` pops $x = 4, y = 2 \to$ **8**. (Infix: $(7 - 3)(1 + 1)$.)

5.9 Let stack 1 grow upward from slot 0 with index `top1` (initially -1) and stack 2 grow downward from slot $N - 1$ with index `top2` (initially N). Then `push1(x)`: `data[++top1] = x`; `pop1()`: `return data[top1--]`; `push2(x)`: `data[--top2] = x`; `pop2()`: `return data[top2++]`. Both stacks are empty when `top1 = -1` and `top2 = N` respectively, and the array is full — for **both** pushes — exactly when `top1 + 1 == top2`. No slot is wasted, and neither stack overflows while the other has room to spare. Every operation is $O(1)$.

5.10 Keep an *in* stack and an *out* stack. `enqueue(x)` pushes x onto *in*. `dequeue()`: if *out* is empty, pop every element of *in* and push it onto *out* — which reverses their order, putting the oldest element on top — and then pop *out*. A single dequeue that triggers the transfer costs $O(n)$. But follow one element through its life: it is pushed onto *in* once, popped from *in* once, pushed onto *out* once and popped from *out* once. Four $O(1)$ steps per element, whatever the interleaving. So m operations cost at most $4m$ steps in total — $O(1)$ amortized per operation, by the same accounting as Equation (5.5).

5.11 With $N = 6$ the first free slot is (front + count) mod 6 by Equation (5.7):

Operation	Slot	`front`	`count`	Result
start	—	4	0	empty
enqueue A	$(4 + 0) \bmod 6 = 4$	4	1	ok
enqueue B	$(4 + 1) \bmod 6 = 5$	4	2	ok
enqueue C	$(4 + 2) \bmod 6 = 0$	4	3	ok, wrapped
dequeue	read slot 4 → A	5	2	ok
enqueue D	$(5 + 2) \bmod 6 = 1$	5	3	ok
enqueue E	$(5 + 3) \bmod 6 = 2$	5	4	ok
enqueue F	$(5 + 4) \bmod 6 = 3$	5	5	ok
enqueue G	$(5 + 5) \bmod 6 = 4$	5	6	ok, now full
enqueue H	—	5	6	refused: `count = N`

Queue order at the end, from `front`: B (5), C (0), D (1), E (2), F (3), G (4).

5.12 (a) By Equation (5.8), size $= (2 - 6 + 8) \bmod 8 = 4$. (b) Slots 6, 7, 0, 1, in that queue order; `rear` = 2 is the next free slot. (c) The queue holds at most $N - 1 = 7$, so **3** more enqueues succeed (into slots 2, 3, 4); then `rear` = 5 and $(5 + 1) \bmod 8 = 6 =$ front: full. (d) Two dequeues read slots 6 and 7, so `front` = 0. Starting from `rear` = 2 with 2 elements, five enqueues succeed (slots 2, 3, 4, 5, 6), leaving `rear` = 7 and $(7 + 1) \bmod 8 = 0 =$ front: full with 7 elements. So `front` = 0, `rear` = 7.

5.13 It does not cure it. The reset helps only when the queue becomes completely empty. Counterexample with capacity 5: enqueue 1, 2, 3, 4, 5 (`tail` = 4); dequeue once (`head` = 1, one element gone, queue not empty, so no reset); enqueue 6 → refused, although slot 0 is free. Any

traffic pattern in which the queue never quite empties — the normal state of a busy printer or router — drifts to a false full. Only wrap-around fixes the problem.

5.14 Keep a deque of **indices** whose values are in decreasing order from front to back. For each i from 0 to $n-1$: (1) pop from the front any index $\le i-k$ (it has left the window); (2) pop from the back every index whose value is $\le A[i]$ (it can never be a maximum again while $A[i]$ is in the window); (3) push i at the back; (4) once $i \ge k-1$, output A[front]. Each index is pushed once and popped at most once, so the total work is $O(n)$.

Trace on $[1,3,-1,-3,5,3,6,7]$, $k=3$ (deque shows indices, values in brackets):

i	$A[i]$	Deque after the step	Output
0	1	0 [1]	—
1	3	1 [3]	—
2	−1	1 [3], 2 [−1]	3
3	−3	1 [3], 2 [−1], 3 [−3]	3
4	5	4 [5]	5
5	3	4 [5], 5 [3]	5
6	6	6 [6]	6
7	7	7 [7]	7

The window maxima are **3 3 5 5 6 7**. At $i=4$, index 1 is popped from the front because $1 \le 4-3$, and indices 3 and 2 are popped from the back because their values are below 5.

5.15 (a) A single FIFO queue serves strictly by arrival, so an express order that arrives after five regular orders waits behind all five.

(b) Keep two FIFO queues, `express` and `regular`. Enqueue each order into the queue of its class. To serve: if `express` is not empty, dequeue from it; otherwise dequeue from `regular`. Both operations stay $O(1)$, and arrival order is kept within each class because each class is its own FIFO queue.

(c) Use a priority queue whose key is the pair (class, arrival number), where express sorts before regular and a counter increments on every arrival. Comparing pairs lexicographically serves all express orders first and, within a class, the smallest arrival number first. The arrival number is essential: a heap is not *stable*, so without it two orders of the same class could come out in either order. Solution (b) is simpler and faster when there are only two classes; (c) extends to any number of priority levels.

5.16 Keep a second stack, `mins`, alongside the main one. On `push(x)`: push x onto the main stack, and push min(x,mins.top()) onto `mins` (or just x if `mins` is empty). On `pop()`: pop both stacks. Then `getMin()` is `mins.top()`. The invariant is that the top of `mins` is always the minimum of the elements currently in the main stack, because `mins` records the minimum *as of each push* and popping restores exactly the earlier state. All three operations are $O(1)$; the extra space is $O(n)$. (A space-saving variant pushes onto `mins` only when $x \le$ the current minimum.)

5.17 A 1 MB stack is 1,048,576 bytes, so $1{,}048{,}576/64 = 16{,}384$ frames. An 8 MB stack allows $8 \times 16{,}384 = 131{,}072$ frames. The real limit is lower because `main`, the C++ runtime and the operating system's start-up code already occupy part of the stack, the compiler may add padding and saved registers to every frame, and in a debug build frames are usually larger. In Python the interpreter stops at its recursion limit (1,000 by default) long before the operating system's stack is full.

5.18 One version:

```
template <typename T>
T popValue(std::stack<T>& s) {
    if (s.empty()) throw std::underflow_error("pop from an empty stack");
    T value = std::move(s.top());   // take the element out first ...
    s.pop();                        // ... then remove it
    return value;
}
```

The standard library separates `top` and `pop` for *exception safety*. If `pop` removed the element and then returned it by value, the copy into the caller's variable could throw (for example, if copying a `std::string` runs out of memory) after the element had already left the stack — and the element would be lost forever. With separate `top` and `pop`, the caller copies first; if the copy throws, the stack is untouched. In our helper the same risk exists at `return value;`, but modern C++ moves or elides that copy, which for standard types does not throw. The check for emptiness is left to the caller in the library for speed: a program that already knows the stack is non-empty should not pay for a second test.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, sec. 10.1.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 2.2.1.

[3] É. Lucas, *Récréations mathématiques*, vol. 3. Paris, France: Gauthier-Villars, 1893, pp. 55–59 ("La Tour d'Hanoï").

[4] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011, sec. 1.3.

[5] C. L. Hamblin, "Translation to and from Polish notation," *The Computer Journal*, vol. 5, no. 3, pp. 210–213, 1962, doi: 10.1093/comjnl/5.3.210.

[6] E. W. Dijkstra, "Algol 60 translation: An Algol 60 translator for the X1 and Making a translator for Algol 60," Mathematisch Centrum, Amsterdam, The Netherlands, Rep. MR 35/61, 1961.

[7] Microsoft, "/STACK (Stack allocations)," *Microsoft Learn — C/C++ Build Reference*. [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/reference/stack-stack-allocations. [Accessed: 24-Sep-2026].

[8] ISO/IEC 14882:2024, *Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2024, subclause "Container adaptors".

[9] cppreference.com, "std::stack." [Online]. Available: https://en.cppreference.com/w/cpp/container/stack. [Accessed: 24-Sep-2026].

[10] Oracle, "Class ArrayDeque," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/ArrayDeque.html. [Accessed: 24-Sep-2026].

[11] Oracle, "Class Stack," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Stack.html. [Accessed: 24-Sep-2026].

[12] Python Software Foundation, "collections — Container datatypes," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/collections.html. [Accessed: 24-Sep-2026].

[13] Python Software Foundation, "queue — A synchronized queue class," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/queue.html. [Accessed: 24-Sep-2026].

[14] Profitina, “Profitina — AI Visibility (GEO) platform.” [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Chapter 6
Pointers, References, and Functions

By the end of this chapter you will be able to draw any small C++ program as boxes and arrows in memory and read the drawing back as code; declare a pointer, take an address with `&`, follow it with `*`, and say in one sentence what each of those three symbols means in every position it can appear; move a pointer through an array and explain why `p + 1` jumps four bytes for an `int` and one hundred and four for a `Book`; reach a structure's fields through a pointer with `->` and follow a chain of nodes from one to the next, which is exactly what Chapter 7 does to build a linked list; allocate memory at run time with `new` and `new[]`, give it back with `delete` and `delete[]`, and build a growable array that doubles itself; recognize the five classic pointer bugs (dangling pointers, leaks, double frees, mismatched deletes and null dereferences), make AddressSanitizer and valgrind catch each one, and read their reports; use `nullptr`, references and `const` so that the compiler catches mistakes before they run; tell the swap story three ways, by value, by pointer and by reference, and say why only two of them work; return pointers and references without leaving them dangling; replace most hand-written `delete` with `std::unique_ptr` and `std::shared_ptr`; and explain precisely what Java and Python do when you pass an object to a method, including the phrase "pass by object reference" and the difference between changing an object and rebinding a name.

6.1 Where We Left Off

Chapter 1 drew memory as a long row of numbered bytes and showed that an array is a run of equal-sized slots laid side by side, so that the computer can find slot *i* with one multiplication and one addition. Chapters 2 and 3 used arrays to search and sort. Chapter 5 built stacks and queues on top of arrays and found their weak spot: an array is born with a fixed size, and a queue that must grow beyond it has nowhere to go.

Throughout those chapters we used a few symbols without explaining them fully. We wrote `&catalog[i]` to talk about where an element lives. We wrote function headers such as `void printBook(const Book& b)` and trusted that the `&` did something useful. We passed arrays to functions and noticed, perhaps with some surprise, that the function could change the caller's array even though nothing seemed to say it could. This chapter explains all of that from the beginning, and then goes further.

The reason a data structures book spends a whole chapter here is simple. Arrays keep their elements next to each other, and that closeness is what makes them fast. But many of the structures still ahead of us, the linked lists of Chapters 7 and 9, the trees of Chapter 11, the graphs of Chapter 13, keep their pieces scattered wherever memory happens to be free, and join them with *addresses*. A node in a linked list does not know its neighbor by position; it knows its neighbor's address. Everything from Chapter 7 onward rests on being completely comfortable with three ideas: an address, a variable that stores an address, and a piece of memory that exists because the program asked for it while running. Those are this chapter's pointers, and its dynamic memory.

The chapter also answers a question that every programmer eventually asks about functions: *when I hand something to a function, does the function get the thing itself or a copy of it?* C++ lets you choose, and the choice is written into every function header. Java and Python do not let you choose, and each has its own fixed rule. Knowing all three rules precisely will save you from a class of bugs that are otherwise very hard to see.

We keep the running example of the earlier chapters: *Pustaka Ceria*, a small community library whose catalog holds records like this one.

```
struct Book {
    std::string id, title, author;
    int year;
};
```

Five books will appear again and again: *Laskar Pelangi* (B4001, Andrea Hirata, 2005), *Filosofi Kopi* (B4002, Dee Lestari, 2006), *Bumi Manusia* (B4003, Pramoedya Ananta Toer, 1980), *Negeri 5 Menara* (B4004, Ahmad Fuadi, 2009) and *Cantik Itu Luka* (B4005, Eka Kurniawan, 2002).

All C++ in this chapter is C++17, compiled with `g++ -std=c++17 -O2 -Wall -Wextra` and producing no warnings; the language is defined by the ISO standard [1], and a readable tour of the modern language is Stroustrup's [2]. Every C++ program also exists in Java and in Python in the chapter's code folder, and the Java and Python versions that teach something new appear in the text, in Section 6.14.

6.2 Memory, Addresses, and What a Pointer Is

6.2.1 Memory as a street of numbered houses

Picture a very long street in which every house is the same size, holds exactly one byte, and has a house number painted on the door. The numbers start at zero and go up by one from house to house. That street is your computer's memory, and the house number is an *address*. A byte is eight bits, enough to hold a number from 0 to 255 or a single Latin letter.

A variable is a name that the compiler gives to one or more consecutive houses. A `char` needs one house. An `int` on today's machines needs four consecutive houses, a `float` four, a `double` eight. When we say "the address of a variable" we mean the number on the first of its houses.

Figure 6.1 shows sixteen consecutive bytes of memory. The addresses are written in hexadecimal, base sixteen, because that is how debuggers and C++ print them: the digits run 0 to 9 and then A to F for ten to fifteen, so the byte after `1009` is `100A`, and `0x` in front of a number simply announces that it is hexadecimal. Three variables live here. The `char x` holds the letter `'a'` in the single byte at address `0x1003`. The `float value` holds 13.4 in the four bytes `0x1004` to `0x1007`. The third variable, `addressX`, occupies eight bytes, and what it holds is not a letter and not a measurement. It holds the number `0x1003`, which is the address of `x`.

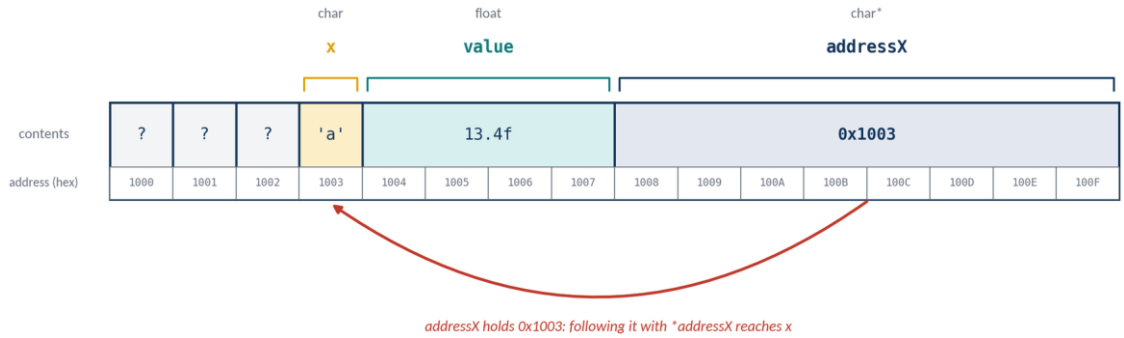


Figure 6.1 Sixteen bytes of memory. Each variable occupies consecutive bytes; the pointer `addressX` holds the address `0x1003`, so following it leads back to `x`.

That third variable is a pointer.

> **DEFINITION.** A *pointer* is a variable whose value is the address of a location in memory. We say the pointer *points to* whatever lives at that address. A pointer that points to nothing holds the special value *null*, written `nullptr` in C++.

Read the definition twice, because every later idea depends on it. A pointer does not contain the book, the number or the letter. It contains *where* the book, the number or the letter can be found. The house number of a friend written on a scrap of paper is a pointer: the paper is not the house, it is a way to reach the house, and you can photocopy the paper as often as you like without building a single new house.

6.2.2 The address-of operator `&` and a first pointer

C++ gives two operators for moving between a thing and its address. The first is `&`, the *address-of* operator. Written in front of a variable, `&x` means "the address of `x`". The second is `*`, which we meet in a moment. **Listing 6.1** puts both to work on the variables of **Figure 6.1**.

Listing 6.1 `01_first_pointer.cpp`: taking an address with `&`, storing it in a pointer, and following the pointer with `*`.

```
// A first pointer: take an address with &, keep it in a pointer
// variable, and follow it with *.
#include <iostream>

int main() {
    char x = 'a';
    char* addressX = &x;                  // addressX holds the address of x
    std::cout << "x has value " << x << ", stored at address "
              << static_cast<void*>(addressX) << '\n';

    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';

    std::cout << "sizeof(char*) = " << sizeof(char*)
              << ", sizeof(float*) = " << sizeof(float*)
              << ", sizeof(double*) = " << sizeof(double*) << '\n';
    return 0;
}
```

```
x has value a, stored at address 0x7ffc2d719d93
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
```

Let us read it slowly, one line at a time.

- `char* addressX = &x;` declares a variable named `addressX` whose type is `char*`, read aloud as "pointer to `char`". The `*` in a *declaration* means "this variable is a pointer". The value we store in it is `&x`, the address of `x`.
- Printing the address needs one small trick. `std::cout` treats a `char*` as the start of a text string and would try to print characters until it found a zero byte, so we convert the pointer to `void*`, "a pointer to something unspecified", which `cout` prints as a hexadecimal number. For every other pointer type, such as `float*`, no conversion is needed.
- `float* address = &value;` is a pointer to `float`. Printing it shows an address; printing `*address` shows 13.4.
- `*address = 27.5f;` writes *through* the pointer. The variable `value` itself changes, because there is only one float in this picture and the pointer is a second route to it.
- The last line shows that every pointer, whatever it points to, is the same size on a given machine: eight bytes on a 64-bit computer, because an address there is a 64-bit number.

The printed addresses are real addresses from one run on a 64-bit Linux machine. On your computer, and even on the next run of the same program, they will be different, because the operating system deliberately places programs at unpredictable addresses to make attacks harder (this is called *address space layout randomization*). The *differences* between addresses are stable, and that is what the rest of the chapter measures. Notice that `x` sits at an address ending in `3` and `value` at the very next address ending in `4`: the compiler happened to place them side by side, just as in **Figure 6.1**.

6.2.3 The dereference operator `*`

The second operator, `*` written in front of a pointer inside an *expression*, is called *dereference* or *indirection*. `*address` means "go to the address stored in `address`, and use the thing that lives there". Used on the right of an assignment it reads; used on the left it writes.

> **DEFINITION.** For a pointer `p` of type `T*` that points to an object, the expression `*p` *is* that object: an lvalue of type `T` that can be read or assigned. The operators are inverses: `*&x` is `x`, and `&*p` is `p`.

This double life of `*` confuses almost every beginner, so it deserves a table of its own, **Table 6.1**. The same happens to `&`, which we will see in Section 6.9 also declares references.

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 compares a pointer with an ordinary variable, point by point.

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

The last row of **Table 6.2** is the seed of every pointer bug in Section 6.7: an ordinary variable and its value live and die together, while a pointer and the thing it points to have separate lives, and nothing forces them to agree.

6.2.4 A pointer's type matters

Every pointer has a pointed-to type, and the compiler takes it seriously. A `float*` is used to reach a `float`, and when you write `*address` the compiler reads exactly four bytes and interprets them as a floating-point number. If a pointer could point to something of another type, the program would read the right bytes with the wrong meaning.

Suppose a program stores the address of a `long` in an `int*`. A C compiler only warns about that. A C++ compiler refuses:

```
int main() {
    long value = 202320242;
    int* warningAddress = &value;
    return *warningAddress == 0;
}
```

```
wrong_type.cpp:3:27: error: cannot convert 'long int*' to 'int*' in initialization
    3 |     int* warningAddress = &value;
      |                           ^~~~~~
      |                           |
      |                           long int*
```

That refusal is a feature. On a 64-bit Linux machine a `long` is eight bytes and an `int` four, so an `int*` aimed at a `long` would silently see only half of it. C++ lets you override the check with an explicit cast, and the program in **Listing 6.3** does exactly that once, deliberately, to count bytes; in ordinary code you should never need to.

6.2.5 Drawing memory: the box-and-arrow convention

Real addresses are long, change from run to run, and hide the structure we care about. From now on we draw memory the way programmers draw it on whiteboards: a variable is a box with its name beside it; a value is written inside the box; and a pointer is a box with a dot inside and an arrow from the dot to the box it points to. A null pointer is a box with the symbol ⌀ or the word `nullptr`. That is all. With this convention a program's state becomes a picture you can check by eye, and every pointer bug in this chapter becomes an arrow that points somewhere it should not.

Drawing takes discipline in three respects. Draw *every* variable, including pointers, as its own box: a pointer is not "the arrow", it is a variable that *holds* the arrow. When a statement runs, redraw only what it changes, and change nothing else. And when an assignment copies a pointer, draw a *new* arrow from the new box; never move the old one. Most mistakes in the next two sections come from skipping one of these rules.

6.3 Operations on Pointers

A pointer is a variable, so it can be assigned, compared and, within an array, moved. Each operation is simple on its own; the trick is never to confuse an operation on the *pointer* with an operation on the thing it points to.

6.3.1 Assignment: copying an address versus copying a value

Listing 6.2 works through four small cases: two pointers to one variable; copying a pointed-to value into an ordinary variable; copying a value between two variables through pointers; and changing a variable through a pointer.

Listing 6.2 `02_pointer_assign.cpp`: `q = p` copies an address, `*q = *p` copies a value.

```
// Assigning pointers versus assigning through pointers.
#include <iostream>

int main() {
    std::cout << std::boolalpha;

    // 1. One address, two pointers.
    float y = 3.14f;
    float* x1 = &y;
    float* x2 = x1;                         // copies the address, not the value
    std::cout << "1. *x1 = " << *x1 << ", *x2 = " << *x2
              << ", x1 == x2: " << (x1 == x2) << '\n';

    // 2. Copy the pointed-to value into an ordinary variable.
    int a = 13;
    int* p = &a;
    int b = *p;                             // b gets 13; b is a separate box
    std::cout << "2. a = " << a << ", b = " << b
              << ", p points at b: " << (p == &b) << '\n';

    // 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';

    // 5. Writing through either pointer changes the one variable a.
    *q = 99;
    std::cout << "5. after *q = 99:  a = " << a << ", *p = " << *p
              << ", c = " << c << '\n';
    return 0;
}
```

```
1. *x1 = 3.14, *x2 = 3.14, x1 == x2: true
2. a = 13, b = 13, p points at b: false
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
5. after *q = 99:  a = 99, *p = 99, c = 3
```

Step 1 makes two pointers, `x1` and `x2`, hold the same address. Nothing is copied except eight bytes of address; there is still one `float`, and both pointers reach it. That is why `x1 == x2` prints `true`: comparing two pointers compares the addresses they hold.

Step 2 declares `int b = *p;`. The right side follows `p` to `a` and reads 13; the left side is an ordinary new variable that receives a *copy* of 13. From now on `a` and `b` are unrelated, and `p` does not point to `b`.

Steps 3 and 4 are the heart of the matter, and **Figure 6.2** draws them. In (a), `p` points to `a` and `q` points to `c`. Statement 3, `*q = *p`, reads through `p` and writes through `q`: the *value* 13 travels from `a` into `c`. The arrows do not move; `p == q` is still false. Statement 4, `q = p`, copies the *address* stored in `p` into `q`. Now both arrows reach `a`, and `c` is no longer reachable through `q` at all. Statement 5 writes 99 through `q`, and because `q` now points to `a`, it is `a` that changes.

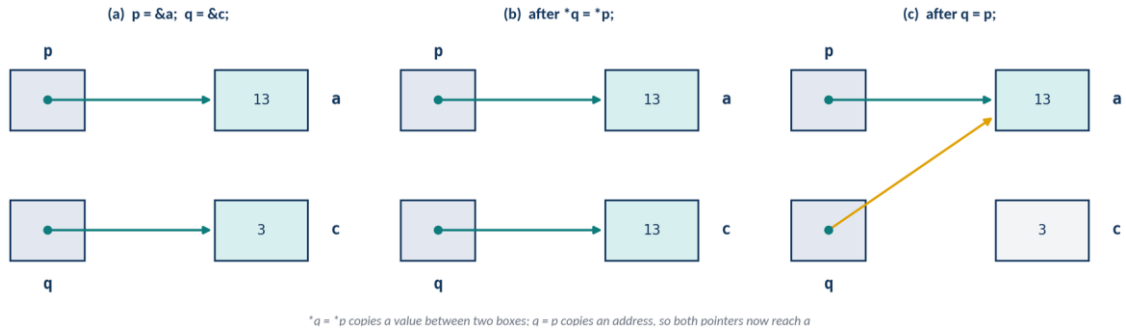


Figure 6.2 Two ways to assign with pointers. `*q = *p` moves a value between the boxes; `q = p` moves an arrow.

> **KEY INSIGHT.** Ask of every assignment involving pointers: *is there a `*` on the left?* With a star, the statement changes the object at the end of the arrow and leaves every arrow where it was. Without a star, it changes where an arrow points and leaves every object untouched.

6.3.2 Comparison

Two pointers of the same type can be compared with `==` and `!=`, which ask whether they hold the same address. This is the test "do these two pointers point to the same object?", and it is completely different from asking whether two objects have equal *values*: in Step 2 above, `a` and `b` are equal, $13 = 13$, but `&a == &b` is false because they are two different boxes.

A pointer can also be compared with `nullptr`, which asks whether it points anywhere at all, and inside a single array two pointers can be ordered with `<`, `<=`, `>` and `>=`: the pointer to the element with the smaller index is the smaller pointer. Ordering pointers into *different* arrays or unrelated variables has no defined meaning in C++, and a program must not rely on it.

6.3.3 Arithmetic: moving within an array

Because an array's elements sit next to each other, a pointer to one element can be moved to its neighbors by adding or subtracting integers. The rule that makes this work is the most important formula in the chapter. If `p` has type `T*`, then

$$\text{address}(p + i) = \text{address}(p) + i \cdot \text{sizeof}(T) \tag{6.1}$$

where `sizeof(T)` is the number of bytes one `T` occupies. In words: adding 1 to a pointer does not move it one byte; it moves it one *element*. Equation (6.1) is the same address formula that Chapter 1 used to find slot i of an array, now written with a pointer in place of the array's base address.

Figure 6.3 shows it for an `int` array starting, for the sake of the picture, at address 1000. Each step of the pointer moves four bytes, because `sizeof(int)` is 4.

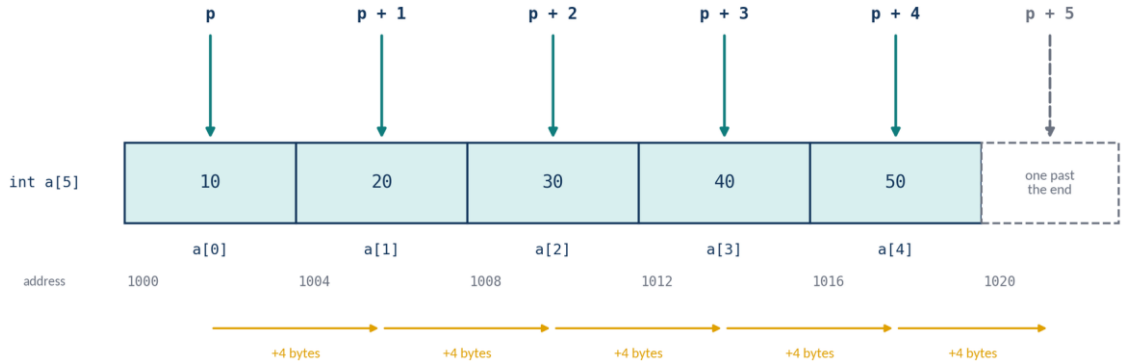


Figure 6.3 Pointer arithmetic on `int a[5]`. Each `+1` moves one element, four bytes; `p + 5` points one past the end, a position that may be computed and compared but never dereferenced.

The same idea runs in reverse. Subtracting two pointers into the same array gives the number of *elements* between them, not the number of bytes:

$$q - p = \frac{\text{address}(q) - \text{address}(p)}{\text{sizeof}(T)} \tag{6.2}$$

The result has type `std::ptrdiff_t`, a signed integer, because $q - p$ is negative when `q` comes before `p`. **Listing 6.3** checks both formulas on four element types. To see the raw byte distance it converts the pointers to `const char*`, since `sizeof(char)` is 1 by definition and subtraction of `char` pointers therefore counts bytes.

Listing 6.3 `03_pointer_arith.cpp`: each step of `p + i` moves `sizeof(T)` bytes, and pointer subtraction counts elements.

```cpp
// Pointer arithmetic moves in whole elements, not in bytes.
#include <cstddef>
#include <iostream>
#include <string>

struct Book {
    std::string id, title, author;
    int year;
};

template <typename T>
void showSteps(const char* name, T* p) {
    std::cout << name << " (sizeof " << sizeof(T) << ")";
    for (int i = 0; i < 4; ++i) {
        std::ptrdiff_t bytes = reinterpret_cast<const char*>(p + i)
                             - reinterpret_cast<const char*>(p);
        std::cout << "  p+" << i << ": +" << bytes;
    }
    std::cout << '\n';
}

int main() {
    char letters[4] = {'D', 'a', 't', 'a'};
    int counts[4] = {0, 0, 0, 0};
    double prices[4] = {0.0, 0.0, 0.0, 0.0};
    Book shelf[4];
    showSteps("char  ", letters);
    showSteps("int   ", counts);
    showSteps("double", prices);
    showSteps("Book  ", shelf);

    int a[5] = {10, 20, 30, 40, 50};
    int* first = a;                       // same as &a[0]
    int* last = a + 4;                    // same as &a[4]
    std::cout << std::boolalpha;
    std::cout << "*first = " << *first << ", *last = " << *last << '\n';
    std::cout << "*(first + 2) = " << *(first + 2) << ", first[2] = " << first[2] << '\n';
    std::cout << "last - first = " << (last - first) << " elements\n";
    std::cout << "first < last: " << (first < last) << '\n';
```

Listing 6.3 (Continued) *03_pointer_arith.cpp: each step of p + i moves sizeof(T) bytes, and pointer subtraction counts elements.*

```cpp
    int* p = first;
    ++p;                                  // now at a[1]
    p += 2;                               // now at a[3]
    --p;                                  // now at a[2]
    std::cout << "after ++p, p += 2, --p: *p = " << *p
              << ", index = " << (p - a) << '\n';

    bool allSame = true;
    for (int i = 0; i < 5; ++i) {
        allSame = allSame && (&a[i] == a + i) && (a[i] == *(a + i));
    }
    std::cout << "&a[i] == a + i and a[i] == *(a + i) for every i: " << allSame << '\n';
    return 0;
}
```

```
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
first < last: true
after ++p, p += 2, --p: *p = 30, index = 2
&a[i] == a + i and a[i] == *(a + i) for every i: true
```

The first four lines confirm Equation (6.1): a step is 1 byte for `char`, 4 for `int`, 8 for `double`, and 104 for our `Book`. That last number is not a law of nature. A `std::string` object in the GNU C++ library is 32 bytes (it keeps a pointer to its characters, a length, and a small internal buffer), three of them make 96, the `int` adds 4, and the compiler pads the total to 104 so that the next `Book` in an array starts at a multiple of 8. Another compiler could give another size; the point is that *you never have to know it*, because the compiler multiplies by `sizeof(Book)` for you. The remaining lines check Equation (6.2) (`last - first` is 4 elements, not 16 bytes) and walk `p` forward and back.

6.3.4 What is allowed, and what is not

Table 6.3 collects the operations. Everything not in the table, such as adding two pointers, multiplying a pointer by a number, or subtracting pointers into different arrays, is either rejected by the compiler or has no defined meaning.

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

WARNING. “Undefined behavior” is a technical term in C++, and it is harsher than it sounds. It does not mean “the program crashes” or “the result is garbage”. It means the language standard places *no requirement at all* on what happens: the program may crash, may print a plausible wrong answer, may work perfectly today and fail after the next compiler upgrade. The compiler is even allowed to assume that undefined behavior never happens and optimize accordingly. Every “condition” column entry in **Table 6.3** that is violated produces undefined behavior.

6.4 Pointers and Arrays

6.4.1 An array name becomes a pointer

In almost every expression, the name of an array is automatically converted into a pointer to its first element. This conversion is called *array-to-pointer decay*. So after `int a[5];`, writing `int* p = a;` is exactly the same as writing `int* p = &a[0];`. The two exceptions worth knowing are `sizeof(a)`, which gives the size of the whole array, and `&a`, which gives a pointer to the whole array rather than to its first element.

Decay, together with Equation (6.1), explains what the square brackets really mean. The C++ standard *defines* subscripting in terms of pointer arithmetic [1]:

$$a[i] \equiv *(a + i) \tag{6.3}$$

The array name decays to a pointer to `a[0]`, the pointer is moved i elements, and the result is dereferenced. One amusing consequence follows from the fact that addition is commutative: $*(a + i)$ is the same as $*(i + a)$, so `i[a]` is legal C++ and means `a[i]`. Nobody writes that in real code, but seeing it once, at the end of **Listing 6.4**, removes any lingering feeling that array subscripts are magic.

6.4.2 Walking an array with a pointer

A pointer can walk a five-element array in two ways, jumping directly with `p = a + k` and stepping with `p++`, and it can then be used to overwrite every element. **Listing 6.4** does all of that and ends with the idiomatic C++ loop, which stops when the pointer reaches one past the end.

Listing 6.4 `04_array_walk.cpp`: jumping, stepping and writing through a pointer, then the one-past-the-end loop.

```cpp
// Walking an array with a pointer: read, update, read again.
#include <iostream>

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    int* p = a;                             // p = &a[0]
    std::cout << "The first p = " << *p << '\n';
    for (int k = 1; k < 5; ++k) {
        p = a + k;                          // jump straight to element k
        std::cout << "p = a + " << k << ", *p = " << *p << '\n';
    }

    p = a;                                  // reset
    std::cout << "The value of a =";
    for (int i = 0; i < 5; ++i) {
        std::cout << ' ' << *p;
        ++p;                                // step to the next element
    }

    p = &a[0];                              // reset, the other spelling
    std::cout << "\nUpdate the items...\n";
    for (int i = 0; i < 5; ++i) {
        *p = i * 10;                        // write through the pointer
        ++p;
    }

    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";
    return 0;
}
```

```
The first p = 1
p = a + 1, *p = 2
p = a + 2, *p = 3
p = a + 3, *p = 4
p = a + 4, *p = 5
The value of a = 1 2 3 4 5
Update the items...
The value of a = 0 10 20 30 40
2[a] = 20 (the same as a[2] = 20)
```

Look at the last loop, `for (int* q = a; q != a + 5; ++q)`. The pointer starts at the first element and stops when it *equals* `a + 5`, the position just beyond the last element. C++ promises that this one-past-the-end pointer can be computed and compared for every array, even though it must never be dereferenced. The pair (first element, one past the last) describes a *half-open range*, written [first,last): the square bracket includes the left end and the round bracket excludes the right end. Half-open ranges are the convention of the whole C++ standard library (`v.begin()` and `v.end()` are exactly such a pair), and they have two pleasant properties: the number of elements is simply last − first, and an empty range is simply first = last.

6.4.3 Strings are arrays of `char`, and the terminator matters

A string literal such as `"Danaya"` is stored as an array of `char` ending with an extra byte of value zero, written `'\0'` and called the *null terminator*. So `char s[] = "Danaya";` creates an array of *seven* characters: six letters and the terminator. Functions such as `std::strlen` find the length by walking until they meet the zero.

A classic example is instructive. It walks `"Danaya"` with `p++` six times, prints it, and then, in “Test 1”, runs the same loop *again without resetting the pointer*. The second loop starts at the terminator and carries on into whatever memory happens to follow the array. On one machine where this example was run it printed an `H` and some invisible bytes; “Test 2” then stepped backwards with `p--` through the same foreign memory. The output looked harmless, which is the most dangerous thing undefined behavior can do. **Listing 6.5** keeps the correct parts, walking forward to the terminator and backward to the start, and reproduces the mistake only when you ask for it with the word `overrun`.

Listing 6.5 `05_danaya.cpp`: walking a C string forward and backward with a pointer; `overrun` repeats the classic mistake.

```
// A string is an array of char ending in '\0'. Walking it with a pointer
// must stay inside the array. Run with the word "overrun" to repeat the
// classic mistake of not resetting the pointer.
#include <cstring>
#include <iostream>
#include <string>

int main(int argc, char* argv[]) {
    char s[] = "Danaya";                  // 7 chars: D a n a y a '\0'
    std::cout << "sizeof(s) = " << sizeof(s) << ", strlen(s) = " << std::strlen(s) << '\n';

    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;
    }
    std::cout << '\n';

    if (argc > 1 && std::string(argv[1]) == "overrun") {
        p = s + 6;                        // the terminator, as after the loop
        std::cout << "Overrun:  ";
        for (int i = 0; i < 6; ++i) {     // forgot to reset p = s
            ++p;                          // s + 7, s + 8, ... are not ours
            std::cout << *p;
        }
        std::cout << '\n';
    }
    return 0;
}
```

```
sizeof(s) = 7, strlen(s) = 6
Forward:  Danaya
p now points at s[6], the terminator
Backward: ayanaD
```

Run normally, the program is correct. Now build it with AddressSanitizer, a checking tool built into GCC and Clang that we study properly in Section 6.7, and ask for the overrun. The report below is real but shortened: we keep its first lines and cut the long ones at the right.

```
g++ -std=c++17 -g -fsanitize=address -o danaya 05_danaya.cpp
./danaya overrun
```

```
sizeof(s) = 7, strlen(s) = 6
Forward:  Danaya
p now points at s[6], the terminator
Backward: ayanaD
=================================================================
==23802==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7f67b7d00087 ...
READ of size 1 at 0x7f67b7d00087 thread T0
    #0 0x55e0eaca1a26 in main 05_danaya.cpp:32
```

The report names the kind of error (*stack-buffer-overflow*: reading past the end of an array that lives on the stack), whether it was a read or a write, how many bytes, and the exact source line, line 32, which is `std::cout << *p;` inside the overrun loop. The first read at `s + 6` would have been harmless (it is the terminator); the loop's first `++p` moves to `s + 7`, one past the end, and the very first dereference there is caught. Without the sanitizer the same program usually prints a few odd characters and exits normally, which is exactly how such a bug survives unnoticed.

6.4.4 Arrays as function parameters

When an array is passed to a function, decay happens at the call: the function receives only the address of the first element, never a copy of the elements. That is why a function such as `fill` can fill the caller's array, and why every function that takes an array needs a second piece of information, its length, from somewhere. It also means that a parameter written `int data[]` is not an array at all. The compiler rewrites it to `int* data`, and GCC even warns you if you ask for its `sizeof`:

```
sizeof_param.cpp: In function 'void fill(int*)':
sizeof_param.cpp:3:25: warning: 'sizeof' on array function parameter 'data' will return
size of 'int*' [-Wsizeof-array-argument]
    3 |     std::cout << sizeof(data) << '\n';
      |                         ~^~~~
```

Listing 6.6 makes the point without a warning by writing the parameter honestly as `int*`. It also shows the three modern ways to pass a sequence: a pointer and a count, a half-open range of two pointers, and a standard container passed by `const` reference, which carries its size with it.

Listing 6.6 `06_array_param.cpp`: an array argument arrives as a pointer; a count, a pointer range, or a container supplies the missing length.

```cpp
// An array passed to a function arrives as a pointer to its first element.
#include <array>
#include <iostream>
#include <vector>

void fill(int* data, int n) {            // int data[] means exactly int* data
    for (int i = 0; i < n; ++i) {
        data[i] = i + 1;                  // writes into the caller's array
    }
    std::cout << "  inside fill: sizeof(data) = " << sizeof(data)
              << " (the size of a pointer)\n";
}

void show(const int* data, int n) {      // const: show promises not to write
    for (int i = 0; i < n; ++i) {
        std::cout << data[i] << ' ';
    }
    std::cout << '\n';
}

int sum(const int* first, const int* last) {   // the half-open range [first, last)
    int total = 0;
    for (const int* p = first; p != last; ++p) {
        total += *p;
    }
    return total;
}

int sum(const std::vector<int>& v) {     // a vector knows its own size
    int total = 0;
    for (int x : v) total += x;
    return total;
}

int main() {
    int data[10];
    std::cout << "Fill the data...\n";
    std::cout << "  in main: sizeof(data) = " << sizeof(data) << " (10 ints)\n";
    fill(data, 10);
    std::cout << "The data has been filled.\nShow the data...\n";
    show(data, 10);
    std::cout << "sum of all ten = " << sum(data, data + 10) << '\n';
    std::cout << "sum of data[2..4] = " << sum(data + 2, data + 5) << '\n';
```

Listing 6.6 (Continued) *06_array_param.cpp: an array argument arrives as a pointer; a count, a pointer range, or a container supplies the missing length.*

```cpp
    std::array<int, 4> fixed = {5, 6, 7, 8};      // size is part of the type
    std::vector<int> grown = {1, 2, 3};
    grown.push_back(4);
    std::cout << "std::array sum = " << sum(fixed.data(), fixed.data() + fixed.size())
              << ", std::vector sum = " << sum(grown) << '\n';
    return 0;
}
```

```
Fill the data...
  in main: sizeof(data) = 40 (10 ints)
  inside fill: sizeof(data) = 8 (the size of a pointer)
The data has been filled.
Show the data...
1 2 3 4 5 6 7 8 9 10
sum of all ten = 55
sum of data[2..4] = 12
std::array sum = 26, std::vector sum = 10
```

In `main`, `sizeof(data)` is 40: ten four-byte integers. Inside `fill` it is 8, the size of a pointer, because inside `fill` there is no array, only an address. The `const` in `show(const int* data, int n)` is a promise, checked by the compiler, that `show` only reads through the pointer; Section 6.11 is about such promises. The two-pointer `sum(data + 2, data + 5)` adds exactly `data[2]`, `data[3]` and `data[4]`, which is $3 + 4 + 5 = 12$, showing how naturally half-open ranges describe part of an array. C++20 adds `std::span`, a small object that packages the pointer and the length together; this book stays with C++17, where the pointer-and-count or the pair of pointers is the standard idiom.

6.4.5 Two-dimensional arrays

A two-dimensional array such as `int grid[R][C]` is stored as R rows placed one after another, each row being C consecutive elements. This layout is called *row-major order*. Element `grid[i][j]` is therefore i whole rows plus j elements from the start:

$$\text{address}(\text{grid}[i][j]) = \text{address}(\text{grid}[0][0]) + (i \cdot C + j) \cdot \text{sizeof}(T) \tag{6.4}$$

Equation (6.4) is Equation (6.1) applied twice, once for the row and once for the column, and it explains why a function receiving a two-dimensional array must be told C in its parameter type (`int g[][4]`): without the row length it could not compute where row i starts. For a 3×4 grid of `int` starting at address 1000, `grid[2][1]` lives at $1000 + (2 \cdot 4 + 1) \cdot 4 = 1036$.

6.5 Pointers to Structures and the Arrow Operator

A pointer can point to a whole structure as easily as to an `int`. Given `Book* bp = &catalog[2];`, the expression `*bp` is the `Book` itself, so its title is `(*bp).title`. The parentheses are necessary: the dot binds more tightly than the star, so `*bp.title` would mean `*(bp.title)`, which tries to treat `bp` as a structure and fails to compile. Because reaching a member through a pointer is so common, C++ has a shorthand, the *arrow operator*:

> **DEFINITION.** For a pointer `p` to a structure or class, `p->m` means exactly `(*p).m`: follow the pointer, then select the member `m`.

Listing 6.7 uses the arrow three ways: to read a field, to change a field of the caller's array element, and to walk the whole catalog with a `const Book*`. It then builds something new: three `Node` objects, each holding a `Book` and a pointer to the next node, and follows the chain from `head` until it meets `nullptr`.

Listing 6.7 `07_book_pointer.cpp`: the arrow operator on an array of `Book`, and a first chain of nodes.

```cpp
// Pointers to structures: (*bp).field and bp->field, a pointer walking an
// array of Book, and a first chain of nodes linked by pointers.
#include <iostream>
#include <string>

struct Book {
    std::string id, title, author;
    int year;
};

struct Node {
    Book data;
    Node* next;                          // the address of the next node, or nullptr
};

int main() {
    Book catalog[5] = {
        {"B4001", "Laskar Pelangi", "Andrea Hirata", 2005},
        {"B4002", "Filosofi Kopi", "Dee Lestari", 2006},
        {"B4003", "Bumi Manusia", "Pramoedya Ananta Toer", 1980},
        {"B4004", "Negeri 5 Menara", "Ahmad Fuadi", 2009},
        {"B4005", "Cantik Itu Luka", "Eka Kurniawan", 2002},
    };

    Book* bp = &catalog[2];
    std::cout << "catalog[2].title = " << catalog[2].title << '\n';
    std::cout << "(*bp).title      = " << (*bp).title << '\n';
    std::cout << "bp->title        = " << bp->title << '\n';

    bp->year = 1981;                     // writes into catalog[2] itself
    std::cout << "after bp->year = 1981, catalog[2].year = " << catalog[2].year << '\n';
    bp->year = 1980;

    std::cout << "\nWalking the catalog with a Book*:\n";
    for (const Book* p = catalog; p != catalog + 5; ++p) {
        std::cout << "  [" << (p - catalog) << "] " << p->id << "  "
                  << p->title << " (" << p->year << ")\n";
    }

    // 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;
```

Listing 6.7 (Continued) *07_book_pointer.cpp: the arrow operator on an array of Book, and a first chain of nodes.*

```cpp
    std::cout << "\nFollowing the chain from head:\n";
    for (Node* cur = head; cur != nullptr; cur = cur->next) {
        std::cout << "  " << cur->data.title << '\n';
    }
    std::cout << "head->next->next->data.title = "
              << head->next->next->data.title << '\n';
    return 0;
}
```

```
catalog[2].title = Bumi Manusia
(*bp).title      = Bumi Manusia
bp->title        = Bumi Manusia
after bp->year = 1981, catalog[2].year = 1981

Walking the catalog with a Book*:
  [0] B4001  Laskar Pelangi (2005)
  [1] B4002  Filosofi Kopi (2006)
  [2] B4003  Bumi Manusia (1980)
  [3] B4004  Negeri 5 Menara (2009)
  [4] B4005  Cantik Itu Luka (2002)

Following the chain from head:
  Bumi Manusia
  Laskar Pelangi
  Cantik Itu Luka
head->next->next->data.title = Cantik Itu Luka
```

Figure 6.4 draws both halves. At the top, `bp` holds the address of `catalog[2]`; `bp->title` and `(*bp).title` are two spellings of one field. At the bottom, `head` points to a node, whose `next` points to a node, whose `next` points to a node, whose `next` is `nullptr`. Read `head->next->next->data.title` left to right, one arrow at a time: start at `head`, follow it to the first node (*Bumi Manusia*), take that node's `next` and follow it to the second (*Laskar Pelangi*), take *its* `next` and follow it to the third (*Cantik Itu Luka*), and read the title of that node's `data`.

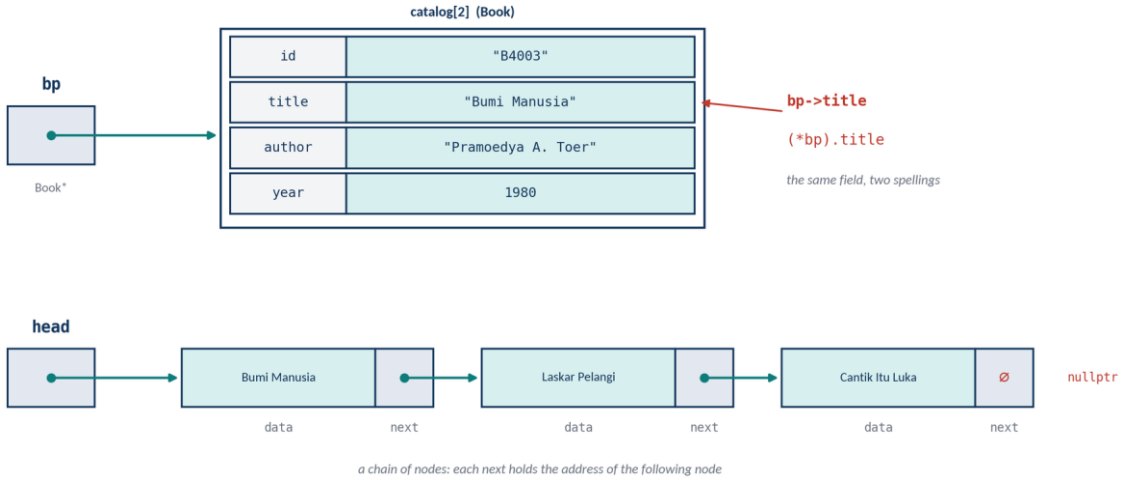


Figure 6.4 Top: a `Book*` and the two spellings of a field access. Bottom: three nodes linked by their `next` pointers, the shape of the linked lists of Chapter 7.

Notice what the chain does *not* need. The three nodes are not in an array and not in any particular order in memory; in the program they are three separate local variables, declared in reverse. The order of the list lives entirely in the `next` pointers. Change one pointer and the order changes, without moving a single `Book`. That observation is the whole idea of the linked list, and Chapter 7 builds on it. What Chapter 7 adds is the ability to create nodes while the program runs, as many as needed, which is the subject of the next section.

The loop `for (Node* cur = head; cur != nullptr; cur = cur->next)` is worth memorizing, because you will write it hundreds of times: start at the head, stop at null, and advance by following `next`. Compare it with the array loop `for (int* q = a; q != a + 5; ++q)`: the structure is identical, and only the two ideas of "end" and "next" differ.

6.6 Dynamic Memory: `new` and `delete`

6.6.1 Two places to live: the stack and the heap

Every variable we have declared so far has been *automatic*: it comes into existence when execution reaches its declaration and disappears when execution leaves the enclosing block. The machine keeps automatic variables in a region called the *stack*, and the name fits. Each function call pushes a *frame* holding that call's parameters and local variables on top; each return pops it off. Chapter 5's stack of plates is literally how function calls are managed. Automatic storage is fast and needs no bookkeeping from the programmer, but it has two hard limits. The size of every automatic variable must be known when the program is compiled, and nothing on the stack can outlive the function that created it.

Data structures break both limits all the time. A library does not know on Monday how many books it will own on Friday. A function that builds a new linked-list node must hand that node back to its caller, and the node must go on existing after the function returns. For such objects C++ provides *dynamic storage*, taken from a second region called the *free store* or, more commonly, the *heap*. (This "heap" has nothing to do with the heap data structure of Chapter 14; the two words simply collided decades ago.)

> **DEFINITION.** A *dynamically allocated* object is created by the expression `new T` (or `new T[n]` for an array) while the program runs. It lives until the program destroys it explicitly with `delete` (or `delete[]`), regardless of which function created it. The `new` expression returns a pointer to the object; that pointer is the only way to reach it.

Figure 6.5 shows the state of `main` in the program we are about to run, halfway through. The stack frame holds three variables: two pointers and an integer. The heap holds the objects they point to: a single `int` with value 42 and an array of five `Book`s. The pointers are ordinary automatic variables and will vanish when `main` returns; the heap objects will not vanish unless someone calls `delete`.

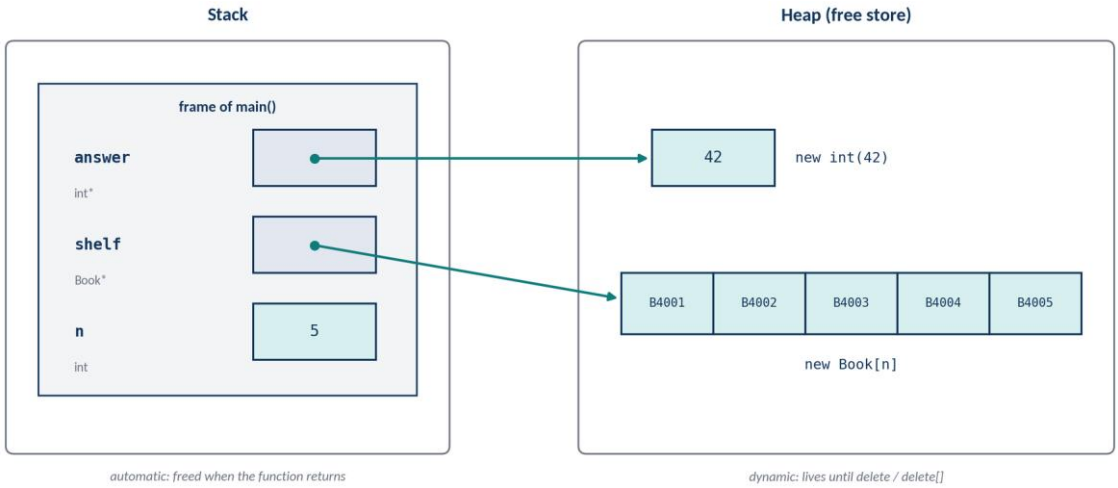


Figure 6.5 Automatic variables live in the stack frame of their function; objects created with `new` live on the heap and are reached only through pointers.

6.6.2 `new`, `delete`, `new[]` and `delete[]`

The rule for using dynamic memory fits in one line and is broken in countless programs: *every `new` is paired with exactly one `delete`, and every `new[]` with exactly one `delete[]`*. **Listing 6.8** follows it

for a single `int`, a single `Book`, and an array of `Book` whose size is a variable. Then it builds something more interesting.

Listing 6.8 `08_dynamic.cpp`: single objects and arrays on the heap, and a growable array that doubles its capacity.

```
// Dynamic memory: new/delete for one object, new[]/delete[] for an array,
// and a growable array that doubles its capacity.
#include <iostream>
#include <string>

struct Book {
    std::string id, title;
    int year;
};

// A minimal growable array of int, managed by hand.
struct IntArray {
    int* data = nullptr;
    int size = 0;
    int capacity = 0;
    long copies = 0;                    // elements copied during growth

    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;
    }

    ~IntArray() { delete[] data; }      // the last block goes back too
};

int main() {
    int* answer = new int(42);          // one int on the heap
    std::cout << "*answer = " << *answer << '\n';
    delete answer;                      // one new, one delete
    answer = nullptr;

    Book* book = new Book{"B4003", "Bumi Manusia", 1980};
    std::cout << "book->title = " << book->title << '\n';
    delete book;
```

Listing 6.8 (Continued) *08_dynamic.cpp: single objects and arrays on the heap, and a growable array that doubles its capacity.*

```
    int n = 5;                              // known only when the program runs
    Book* shelf = new Book[n];              // n default-constructed Books
    for (int i = 0; i < n; ++i) {
        shelf[i].id = "B400" + std::to_string(i + 1);
        shelf[i].year = 2000 + i;
    }
    std::cout << "shelf[4].id = " << shelf[4].id << ", year " << shelf[4].year << '\n';
    delete[] shelf;                         // new[] pairs with delete[]

    IntArray arr;
    const int count = 17;
    for (int v = 1; v <= count; ++v) {
        arr.push(v * v);
    }
    std::cout << "size " << arr.size << ", capacity " << arr.capacity
              << ", last " << arr.data[arr.size - 1]
              << ", total copies " << arr.copies << " (< 2 * " << count << " = "
              << 2 * count << ")\n";
    return 0;
}
```

```
*answer = 42
book->title = Bumi Manusia
shelf[4].id = B4005, year 2004
  grow: capacity 1, copied 0
  grow: capacity 2, copied 1
  grow: capacity 4, copied 2
  grow: capacity 8, copied 4
  grow: capacity 16, copied 8
  grow: capacity 32, copied 16
size 17, capacity 32, last 289, total copies 31 (< 2 * 17 = 34)
```

Four details repay attention.

- `new int(42)` creates an `int` initialized to 42; `new Book{"B4003", "Bumi Manusia", 1980}` creates a `Book` with its fields initialized in order. Both return pointers.
- `new Book[n]` accepts an `n` that is known only at run time, which an ordinary array declaration does not allow in standard C++. Each of the n `Book`s is default-constructed: empty strings and an indeterminate `year`, which is why the loop fills them before they are read.
- After `delete answer;` the program sets `answer = nullptr;`. The `delete` destroys the `int` on the heap but does nothing to the pointer variable, which still holds the old address. Setting it to null afterward is a cheap habit that turns an accidental second use into an obvious crash instead of a silent corruption; Section 6.7 shows why.
- `IntArray` owns a heap array. Its *destructor*, the member function `~IntArray()` that C++ calls automatically when an `IntArray` goes out of scope, performs the matching `delete[]`. That pattern, a heap resource released by the destructor of the object that owns it, is the seed of the smart pointers in Section 6.13.

6.6.3 The growable array and the doubling rule

The `IntArray` in **Listing 6.8** solves the fixed-size problem of Chapter 5. It keeps a heap block of some *capacity*, fills it, and when it is full asks for a new block of twice the capacity, copies the old elements across, and gives the old block back. **Pseudocode 6.1** states the procedure in pseudocode.

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
```

Doubling looks wasteful, because a full array of a million elements suddenly asks for room for two million. It is in fact the reason the structure is fast. Count the element copies performed while pushing n values. Growth happens when the size reaches 1,2,4,8, ..., up to the largest power of two below n, say 2^k with $2^k < n$. Each growth copies all current elements, so the total number of copies is

$$C(n) = 1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n \tag{6.5}$$

The middle step is the sum of a geometric series: each term doubles, so the sum is one less than the next power of two. The final step uses $2^k < n$, so $2^{k+1} < 2n$. Equation (6.5) says that pushing n values costs fewer than $2n$ copies in total, fewer than two copies *per push on average*, even though a single push can copy everything. The output of **Listing 6.8** checks it for $n = 17$: growth at sizes 1,2,4,8,16, total 31 copies, below $2 \cdot 17 = 34$. An average cost per operation that holds over any sequence of operations is called an *amortized* cost, and the doubling array's push is $O(1)$ amortized; the full theory is in Cormen et al. [3]. Growing by a *fixed* amount instead, say $+10$ each time, would make the total $\Theta(n^2)$, which is why every serious library, including `std::vector`, grows by a constant *factor*.

Figure 6.6 shows one growth step and adds a warning that will return in Section 6.7: any pointer that pointed into the *old* block still holds the old address after growth, and the old block has been freed.

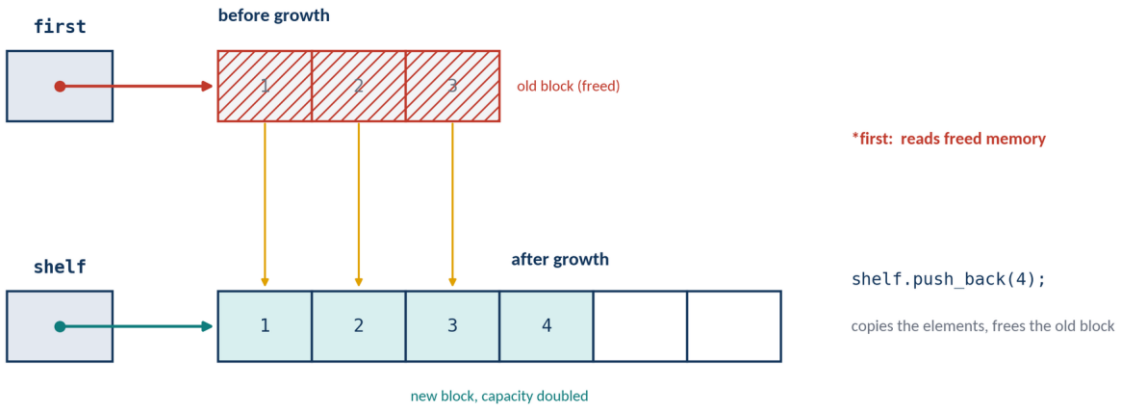


Figure 6.6 One growth step of a doubling array. The elements are copied into a new block and the old block is freed, so a pointer taken before the growth now points into freed memory.

6.4.4 When `new` fails

Memory is finite. If `new` cannot find enough free memory it does not return a null pointer; it *throws* an exception of type `std::bad_alloc`, which ends the program unless something catches

it. If you prefer a null result, the form `new (std::nothrow) T` returns `nullptr` on failure instead, and you must then check it. Most programs let `bad_alloc` propagate, because a machine that has truly run out of memory rarely has a sensible way to continue.

6.7 When Pointers Go Wrong

6.7.1 Five bugs, one root cause

Recall the last row of **Table 6.2**: a pointer and the object it points to have separate lifetimes. Every classic pointer bug is a disagreement between the two.

- A **dangling pointer** still holds the address of an object that no longer exists: it was deleted, or it was a local variable of a function that has returned, or it was an element of a container that has since moved its storage. Using it is a *use after free*.
- A **memory leak** is the opposite disagreement: the object still exists but no pointer to it remains, so the program can never delete it. Each leaked block is lost until the program ends.
- A **double free** deletes the same block twice, usually through two pointers that were copies of each other.
- A **mismatched delete** frees an array created with `new[]` using plain `delete`, or a single object with `delete[]`. The two forms use different bookkeeping, and mixing them is undefined behavior.
- A **null dereference** follows a pointer that holds `nullptr`, usually because a function that "might not find anything" was trusted to always find something.

To these five add the *out-of-bounds access* we met with `"Danaya"`: a pointer that has walked beyond its array. **Figure 6.7** draws the first three in the box-and-arrow convention. In each case the drawing makes the bug visible at once: an arrow into a hatched (freed) block, a block with no arrow into it, and two arrows into a block that is freed twice.

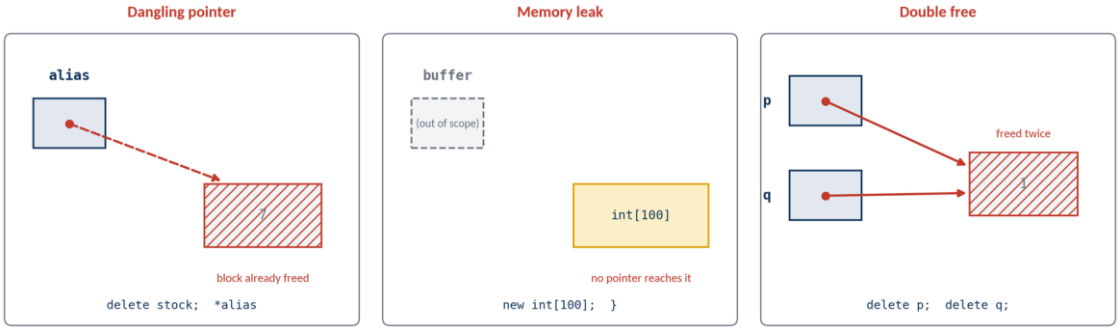


Figure 6.7 Three lifetime bugs drawn as boxes and arrows. Hatched blocks have already been freed.

These are not academic curiosities. MITRE's 2025 list of the 25 most dangerous software weaknesses, compiled from recently published vulnerability records, contains six entries that are exactly the bugs of this section [4]. **Table 6.4** lists them with their 2025 ranks.

Table 6.4 Memory-safety weaknesses in the 2025 CWE Top 25 Most Dangerous Software Weaknesses, published by MITRE in December 2025 [4].

2025 rank	Weakness (CWE identifier)	In this chapter
5	Out-of-bounds Write (CWE-787)	`overflow` in **Listing 6.9**
7	Use After Free (CWE-416)	`dangling`, `stale` in **Listing 6.9**
8	Out-of-bounds Read (CWE-125)	`overrun` in **Listing 6.5**
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 6.9**
16	Heap-based Buffer Overflow (CWE-122)	`overflow` in **Listing 6.9**

6.7.2 The bugs, on purpose

Listing 6.9 commits each bug in its own small function, chosen by a word on the command line. Run with no word, it prints a menu and runs the corrected versions, which is the output shown. Two of the functions keep their pointers inside a `std::vector<int*>`. That is not the natural way to write them, and the reason is revealing: when the `new`, the `delete` and the bad use sit plainly in three consecutive lines, GCC 13 notices and warns (`-Wuse-after-free`, `-Wmismatched-new-delete`), and the book's code must compile without warnings. Real bugs are rarely so kind. The two pointers usually live in different functions or different data structures, and the compiler, which checks one function at a time, cannot see them. The vector plays that role here.

Listing 6.9 `09_mem_bugs.cpp`: seven classic pointer bugs, each selected by a command-line word, and their corrected forms.

```cpp
// Seven classic pointer bugs, one per command-line word. Run with no word for
// the menu and a correct run; run under AddressSanitizer or valgrind to see
// each bug caught.
#include <cstring>
#include <iostream>
#include <string>
#include <vector>

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';
}

void leak(int times) {
    for (int i = 0; i < times; ++i) {
        int* buffer = new int[100];     // 400 bytes each time
        buffer[0] = i;
    }                                   // buffer goes out of scope: block lost
    std::cout << "leaked " << times << " blocks\n";
}

void doubleFree() {
    int* p = new int(1);
    int* q = p;
    delete p;
    delete q;                           // the same block, a second time
}

void mismatch() {
    std::vector<int*> rows;
    rows.push_back(new int[4]);         // allocated with new[] ...
    rows[0][0] = 1;
    delete rows[0];                     // ... but freed with delete, not delete[]
}
```

Listing 6.9 (Continued) *09_mem_bugs.cpp: seven classic pointer bugs, each selected by a command-line word, and their corrected forms.*

```
void overflow(int n) {
    int* marks = new int[n];
    for (int i = 0; i <= n; ++i) {      // <= : one step too far
        marks[i] = i;
    }
    std::cout << "last mark " << marks[n - 1] << '\n';
    delete[] marks;
}

void nullDeref(const char* id) {
    int* found = nullptr;               // "not found"
    if (std::strcmp(id, "B4001") == 0) {
        found = new int(2005);
    }
    std::cout << "year " << *found << '\n';  // no check before *
    delete found;
}

void correct() {
    int* stock = new int(7);
    std::cout << "stock is " << *stock << '\n';
    delete stock;
    stock = nullptr;                    // no dangling pointer left behind

    std::vector<int> shelf = {1, 2, 3};
    shelf.push_back(4);
    int* first = &shelf[0];             // taken AFTER the last growth
    std::cout << "first is " << *first << '\n';

    int n = 5;
    int* marks = new int[n];
    for (int i = 0; i < n; ++i) marks[i] = i;
    std::cout << "last mark " << marks[n - 1] << '\n';
    delete[] marks;                     // new[] pairs with delete[]
}
```

Listing 6.9 (Continued) *09_mem_bugs.cpp: seven classic pointer bugs, each selected by a command-line word, and their corrected forms.*

```
int main(int argc, char* argv[]) {
    std::string bug = argc > 1 ? argv[1] : "";
    if (bug == "dangling")      useAfterDelete();
    else if (bug == "stale")    staleAfterGrowth();
    else if (bug == "leak")     leak(3);
    else if (bug == "double")   doubleFree();
    else if (bug == "mismatch") mismatch();
    else if (bug == "overflow") overflow(5);
    else if (bug == "null")     nullDeref("B9999");
    else {
        std::cout << "usage: mem_bugs dangling|stale|leak|double|mismatch|overflow|null\n";
        std::cout << "no bug chosen; the correct versions:\n";
        correct();
    }
    return 0;
}
```

```
usage: mem_bugs dangling|stale|leak|double|mismatch|overflow|null
no bug chosen; the correct versions:
stock is 7
first is 1
last mark 4
```

Compiled normally and run with a bug word, most of these functions *appear to work*. On our test machine `./mem_bugs dangling` printed `stock is 7`, because the freed block had not yet been reused; `./mem_bugs leak` printed its line and exited normally; `./mem_bugs overflow` wrote past the array and printed `last mark 4`. That is the insidious part: the program is wrong, the output is right, and the damage is invisible until some unrelated change moves memory around. We need tools that watch every access.

6.7.3 AddressSanitizer

AddressSanitizer (ASan), introduced by Serebryany and colleagues at Google [5], is built into GCC and Clang. When a program is compiled with `-fsanitize=address`, the compiler surrounds every heap block, every stack array and every global array with small *red zones* of forbidden bytes, records in a compact *shadow memory* which bytes are currently valid, and inserts a check before every load and store. Freed blocks are not reused immediately but kept in *quarantine*, so a use after free touches memory that is known to be dead. The program runs about twice as slowly, which is a bargain during development. Add `-g` so that reports name source lines:

```
g++ -std=c++17 -g -fsanitize=address -o mem_bugs 09_mem_bugs.cpp
./mem_bugs dangling
```

Here are the first lines of the reports for four of the bugs, from real runs; each report continues with more stack frames, which we omit, and the long hexadecimal addresses differ on every run.

```
==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
```

Read it as a detective's report. *What*: a heap use after free. *Which access*: a read of four bytes, an `int`. *Where*: line 14, the `std::cout` that dereferences `owners[1]`. *What was there*: a 4-byte block, and the address is 0 bytes inside it. *Who freed it*: the `delete` on line 13. Every fact needed to fix the bug is present.

```
==23668==ERROR: AddressSanitizer: heap-use-after-free on address 0x502000000010 ...
READ of size 4 at 0x502000000010 thread T0
    #0 0x56342fa1bcb3 in staleAfterGrowth() 09_mem_bugs.cpp:21
0x502000000010 is located 0 bytes inside of 12-byte region [0x502000000010,0x50200000001c)
freed by thread T0 here:
    #0 0x7f128daff5e8 in operator delete(void*, unsigned long)
    #1 0x56342fa21460 in std::__new_allocator<int>::deallocate(int*, unsigned long)
    ...
    #4 0x56342fa2016e in void std::vector<int, std::allocator<int>
>::_M_realloc_insert<int>(...)
```

The `stale` report is the one to remember from this chapter. Nobody called `delete`. The 12-byte region is the vector's original block of three `int`s, and it was freed *inside* `push_back`, by the reallocation drawn in **Figure 6.6**. A pointer, a reference or an iterator into a `std::vector` becomes invalid whenever the vector grows. This rule will matter every time a later chapter stores addresses of elements that live inside a growing container.

```
==23675==ERROR: AddressSanitizer: attempting double-free on 0x502000000010 in thread T0:
    #0 0x7f920a2ff5e8 in operator delete(void*, unsigned long)
    #1 0x55c2a4b5df2a in doubleFree() 09_mem_bugs.cpp:36
==23678==ERROR: AddressSanitizer: alloc-dealloc-mismatch (operator new [] vs operator
delete) on 0x502000000010
    #1 0x55b14b3220ee in mismatch() 09_mem_bugs.cpp:43
==23681==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x503000000054 ...
WRITE of size 4 at 0x503000000054 thread T0
    #0 0x5617ff2ce230 in overflow(int) 09_mem_bugs.cpp:49
0x503000000054 is located 0 bytes after 20-byte region [0x503000000040,0x503000000054)
```

The double free, the mismatch and the overflow are each caught at their exact line. The overflow report is especially precise: the write is *0 bytes after* a 20-byte region, which is five `int`s, so the loop wrote `marks[5]`, one element past the end, exactly the off-by-one of `i <= n`. For `null`, ASan reports `SEGV on unknown address 0x000000000000` with the hint `address points to the zero page`, and for `leak`, the *LeakSanitizer* built into ASan reports at program exit:

```
leaked 3 blocks
==23797==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 1200 byte(s) in 3 object(s) allocated from:
    #0 0x7f2e41afe6c8 in operator new[](unsigned long)
    #1 0x5600bf86ee0e in leak(int) 09_mem_bugs.cpp:26
SUMMARY: AddressSanitizer: 1200 byte(s) leaked in 3 allocation(s).
```

Three blocks of 100 four-byte integers are 1200 bytes, allocated on line 26, and never freed. Adding `-fsanitize=undefined` alongside (`-fsanitize=address,undefined`) switches on the *UndefinedBehaviorSanitizer* as well, which reports further mistakes such as signed integer overflow and misaligned or null member accesses.

6.7.4 Valgrind

Valgrind takes the opposite approach: instead of recompiling, it runs the unchanged program on a simulated processor and checks every memory access as it happens [6]. Its default tool, *Memcheck*, needs no special compiler flags (though `-g` still gives line numbers), runs the program twenty to fifty times more slowly, and is especially good at leak reports. The current release is Valgrind 3.27.1 of May 2026 [7]; the runs below used the 3.22 package of Ubuntu 24.04, whose reports have the same form.

```
g++ -std=c++17 -g -O0 -o mem_bugs_vg 09_mem_bugs.cpp
valgrind --leak-check=full ./mem_bugs_vg leak
```

```
leaked 3 blocks
==23744== HEAP SUMMARY:
==23744==     in use at exit: 1,200 bytes in 3 blocks
==23744==   total heap usage: 5 allocs, 2 frees, 79,024 bytes allocated
==23744==
==23744== 1,200 bytes in 3 blocks are definitely lost in loss record 1 of 1
==23744==    at 0x48485C3: operator new[](unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==23744==    by 0x10A7D4: leak(int) (09_mem_bugs.cpp:26)
==23744==    by 0x10AE48: main (09_mem_bugs.cpp:86)
==23744==
==23744== LEAK SUMMARY:
==23744==    definitely lost: 1,200 bytes in 3 blocks
```

“Definitely lost” is Valgrind’s name for a block that no pointer anywhere in the program still reaches: a true leak. For the dangling pointer Valgrind reports `Invalid read of size 4`, then `Address 0x4e26080 is 0 bytes inside a block of size 4 free'd`, with the line of the `delete` and the line of the original `new`. Most importantly, the tools also confirm *good* code. Every

program in this chapter other than the deliberate bugs was run under both tools; for example, Valgrind's verdict on **Listing 6.8** is

```
==23823==   total heap usage: 10 allocs, 10 frees, 78,516 bytes allocated
==23823== All heap blocks were freed -- no leaks are possible
==23823== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

and AddressSanitizer is silent on all of them. (The allocations beyond our own come from the C++ runtime library, which frees them too.)

6.7.5 Which tool, on which computer

Table 6.5 summarizes availability. The simplest rule is to build with `-g -fsanitize=address,undefined` whenever you are developing pointer code, and to reach for Valgrind when you cannot recompile or want the most detailed leak report.

Table 6.5 AddressSanitizer and Valgrind compared, as of September 2026 [5], [6], [7], [8].

	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

The language itself is also moving in this direction. The C++26 standard, whose technical content was completed in March 2026, removes undefined behavior from reading an uninitialized local variable (the read is now *erroneous behavior*, with a defined if unhelpful value) and adds a *hardened* standard library whose containers check their bounds [9], [10]; GCC 16.1, released in April 2026, implements much of the new standard, including library hardening, and is the first GCC release to compile C++20 by default [11]. None of this makes the discipline of this chapter unnecessary: hardened containers do not check raw pointers, and no standard can tell when a pointer outlives its object.

> **KEY INSIGHT.** Tools find bugs that have *happened* in a run. They cannot prove a program correct. The real defense is structural: give every heap object exactly one clear owner, prefer containers and smart pointers to raw `new` and `delete`, and let a destructor do the freeing. Sections 6.12 and 6.13 are about building that structure.

6.8 `nullptr`: A Pointer to Nothing

A pointer that does not point to any object must still hold *some* value, and the safe choice is the null pointer. Old C and C++ code wrote it as `0` or as the macro `NULL`, which is usually defined as `0`. That caused a real problem: `0` is first of all an *integer*, so a function overloaded for `int` and for `int*` picks the `int` version when called with `0` or, on some compilers, with `NULL`. C++11 introduced the keyword `nullptr`, which has its own type, `std::nullptr_t`, and converts to every pointer type but to no integer type.

Listing 6.10 shows the overload problem solved, three equivalent null tests, and the most important use of null in data structures: a search function that returns a pointer to what it found, or `nullptr` if it found nothing.

Listing 6.10 `10_nullptr.cpp`: `nullptr` picks the pointer overload, and a search returns `nullptr` for "not found".

```
// nullptr: a null pointer with a type of its own.
#include <iostream>
#include <string>

void describe(int value)  { std::cout << "  describe(int " << value << ")\n"; }
void describe(int* where) { std::cout << "  describe(int* " << (where ? "non-null" :
"null") << ")\n"; }

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
}

int main() {
    std::cout << "Calling describe with 0 and with nullptr:\n";
    describe(0);                          // 0 is an int
    describe(nullptr);                    // nullptr converts only to pointers

    int* p{};                             // value-initialized: null
    int* q = nullptr;
    std::cout << std::boolalpha << "p == nullptr: " << (p == nullptr)
              << ", p == q: " << (p == q) << ", !p: " << !p << '\n';

    std::string titles[3] = {"Laskar Pelangi", "Filosofi Kopi", "Bumi Manusia"};
    for (char letter : {'B', 'Z'}) {
        const std::string* hit = findTitle(titles, 3, letter);
        if (hit != nullptr) {             // check BEFORE dereferencing
            std::cout << letter << ": found \"" << *hit << "\" at index "
                      << (hit - titles) << '\n';
        } else {
            std::cout << letter << ": no title starts with this letter\n";
        }
    }
    return 0;
}
```

```
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
```

The rule that makes null useful is equally short: *check before you dereference*. `if (hit != nullptr)`, `if (hit)` and `if (!hit)` are the three common spellings. Dereferencing a null pointer is undefined behavior (on ordinary operating systems it crashes with a segmentation fault, as ASan showed), so any pointer that *can* be null must be checked on every path that uses it. A convenient convention follows: a function parameter or return value of pointer type announces "this may be null", while a reference, the subject of the next section, announces "this is never null".

Uninitialized pointers are sometimes loosely called "dangling", but the standard vocabulary distinguishes two cases. An *uninitialized* (or *wild*) pointer has never been given a value; it holds

whatever bits were left in its memory. A *dangling* pointer once pointed to a real object that has since died. Both are dangerous, and the cure for the first is trivial: never declare a pointer without initializing it, if only to `nullptr`, as in `int* p{};` or `int* q = nullptr;` in **Listing 6.10**.

6.9 References

6.9.1 A second name for an existing object

A pointer is a variable that *holds* an address. A *reference* is something simpler and more restricted: another name for an object that already exists.

> **DEFINITION.** A declaration `T& r = x;` makes `r` a *reference to* `x`. From then on `r` is an alias of `x`: every use of `r`, reading, writing or taking its address, is a use of `x`. A reference must be initialized when it is declared, can never be made to refer to a different object, and can never be null.

Listing 6.11 explores the three restrictions and the two places references are used most, in loops and as names for elements.

Listing 6.11 `11_references.cpp`: a reference is an alias; assigning to it assigns to the object, and a `for` loop over references changes the container.

```cpp
// References: a second name for an existing object.
#include <iostream>
#include <string>
#include <vector>

struct Book {
    std::string id, title;
    int year;
};

int main() {
    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';

    const int& sum = a + b;              // a const reference may bind a temporary
    std::cout << "const int& sum = a + b gives " << sum << '\n';

    std::vector<int> stock = {3, 5, 8};
    for (int x : stock) x *= 10;         // x is a copy: stock unchanged
    std::cout << "after copy loop:      ";
    for (int x : stock) std::cout << x << ' ';
    for (int& x : stock) x *= 10;        // x is a reference: stock changed
    std::cout << "\nafter reference loop: ";
    for (int x : stock) std::cout << x << ' ';
    std::cout << '\n';

    Book catalog[2] = {{"B4001", "Laskar Pelangi", 2005},
                       {"B4002", "Filosofi Kopi", 2006}};
    Book& favorite = catalog[1];         // no copy, no arrow
    favorite.title += " (2nd ed.)";
    std::cout << "catalog[1].title = " << catalog[1].title << '\n';
    return 0;
}
```

```
after r = 20: a = 20, &r == &a: true
after r = b; b = 7: a = 99, b = 7, r = 99
const int& sum = a + b gives 106
after copy loop:      3 5 8
after reference loop: 30 50 80
catalog[1].title = Filosofi Kopi (2nd ed.)
```

The second line of output is the trap that catches every newcomer. `r = b;` does *not* make `r` refer to `b`. A reference cannot be re-seated, so the statement means exactly what `a = b;` means: copy 99 into `a`. When `b` later becomes 7, `a` and `r` stay 99. With a pointer, `p = &b` would have moved the arrow; with a reference there is no arrow to move, only a second label glued permanently to the same box.

The next line shows a special rule: a *const* reference may bind to a temporary value such as `a + b`, and the temporary is kept alive as long as the reference. The rule exists so that functions taking `const T&` can be called with expressions, as in `printBook(makeBook())`.

The two loops show the most common everyday use. `for (int x : stock)` gives each element to a fresh copy `x`, so multiplying `x` changes nothing in `stock`. `for (int& x : stock)` makes `x` a reference to each element in turn, so the multiplication happens in place. Finally, `Book& favorite`

`= catalog[1];` gives a convenient short name to an element without copying it and without the arrow syntax of a pointer.

6.9.2 References versus pointers

Under the hood a compiler usually implements a reference as a pointer that is dereferenced automatically. The difference is in what the *language* lets you do, and **Table 6.6** puts the two side by side.

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

The practical rule that follows from **Table 6.6** is also the one recommended by the C++ Core Guidelines, the style guide maintained by Stroustrup and Sutter [12]: *use a reference when there must always be an object and it will not change; use a pointer when "no object" is a legitimate possibility or when the pointer must later point somewhere else.* Linked structures need pointers for exactly those two reasons: the last node's `next` has no object to refer to, and insertion and deletion must re-point links.

6.10 Functions and the Three Ways to Pass an Argument

6.10.1 Functions: a short review

A *function* is a named piece of program that does one task and is written separately from the code that uses it. Splitting a program into functions is the oldest form of divide and conquer: a large program becomes small ones that can be written, tested, fixed and reused one at a time, by different people if necessary. A function that returns nothing is declared `void` (older texts call it a *procedure*); a function that returns a value states the type of that value in front of its name and ends with `return`.

Several further ideas belong to this review, and **Listing 6.12** gathers them in one runnable program.

- **Declaration and definition.** The line `int add(int x, int y);` near the top is a *declaration*, also called a *prototype*: a promise that a function with this name, these parameter types and this return type exists. The *definition*, with the body, comes after `main`. The compiler reads a file from top to bottom, so without the declaration the call `add(13, 3)` inside `main` would be an error. In multi-file programs, declarations go in header files and definitions in `.cpp` files.
- **Formal and actual parameters.** In the definition, `x` and `y` are the *formal* parameters, local variables of `add`. In the call `add(13, 3)` the values 13 and 3 are the *actual* parameters, also called *arguments*. The call initializes each formal parameter from the matching argument.

- **Scope and lifetime.** A *global* variable, declared outside every function, is visible in all the code that follows it and lives for the whole run. A *local* variable is visible only inside its block and lives only while the block runs. A local variable with the same name as a global *hides* it, and `::a` reaches the global anyway. A *static local* variable is visible only inside its function but, like a global, is created once and keeps its value between calls: `countCalls` counts by keeping `calls` alive.
- **The return value of `main`.** `int main()` returns a status to the operating system: 0 means success, anything else failure. The form `void main()`, which appears in some older C material, is not valid C++.

Listing 6.12 `12_functions_scope.cpp`: prototypes, formal and actual parameters, global, local and static variables, and a first comparison of passing by value and by pointer.

```
// Functions: declaration before use, parameters, and the three kinds of
// variable lifetime (global, local, static local).
#include <iostream>

int add(int x, int y);                    // declaration (prototype)

int a = 3;                                // global: visible everywhere below

void aGlobal() {
    std::cout << "a in aGlobal() is " << a << " (the global)\n";
}

void passByValue(int a) {                 // this a is a new local copy
    a = a * 2;
    std::cout << "a in passByValue() is " << a << '\n';
}

void passByPointer(int* a) {              // this a holds main's address
    *a = *a * 2;
    std::cout << "*a in passByPointer() is " << *a << '\n';
}

int countCalls() {
    static int calls = 0;                 // created once, survives between calls
    return ++calls;
}

int main() {
    int a = 13;                           // local: hides the global a
    aGlobal();
    std::cout << "a in main() is " << a << ", global ::a is " << ::a << '\n';
    passByValue(a);
    std::cout << "a in main() after passByValue() is " << a << '\n';
    passByPointer(&a);
    std::cout << "a in main() after passByPointer() is " << a << '\n';

    int t = add(13, 3);                   // actual parameters 13 and 3
    std::cout << "add(13, 3) = " << t << '\n';

    countCalls();
    countCalls();
    std::cout << "countCalls() has now been called " << countCalls() << " times\n";
    return 0;                             // 0 tells the operating system "success"
}
```

Listing 6.12 (Continued) *12_functions_scope.cpp: prototypes, formal and actual parameters, global, local and static variables, and a first comparison of passing by value and by pointer.*

```
int add(int x, int y) {                  // definition: formal parameters x, y
    int r = x + y;
    return r;
}
```

```
a in aGlobal() is 3 (the global)
a in main() is 13, global ::a is 3
a in passByValue() is 26
a in main() after passByValue() is 13
*a in passByPointer() is 26
a in main() after passByPointer() is 26
add(13, 3) = 16
countCalls() has now been called 3 times
```

The middle lines of the output already contain the rest of this section in miniature. `passByValue` doubles *its own* `a`, a copy, and `main`'s `a` is still 13 afterward. `passByPointer` receives `main`'s *address* and doubles what lives there, so `main`'s `a` becomes 26.

6.10.2 What happens during a call

When `main` calls a function, the machine pushes a new frame on the stack for the callee. The frame has a slot for every parameter and every local variable of the callee. Before the body runs, each parameter slot is *initialized from* the corresponding argument. Everything in this section follows from one question: *what* is used to initialize the parameter?

- If the parameter is an ordinary type `T`, it is initialized with a **copy of the argument's value**. This is *pass by value*.
- If the parameter is a pointer `T*`, it is initialized with a copy of an **address**, which the caller supplies with `&`. This is *pass by pointer* (strictly speaking, passing a pointer by value).
- If the parameter is a reference `T&`, no new object is created at all: the parameter becomes **another name for the argument itself**. This is *pass by reference*.

6.10.3 The swap story

The classic test of these three modes is a function that exchanges two integers. It sounds trivial, and it is the best experiment in the whole topic. **Listing 6.13** writes it three ways and calls them one after the other on the same two variables.

Listing 6.13 `13_swap.cpp`: swapping by value (fails), by pointer and by reference (both work), and `std::swap`.

```cpp
// The swap story: by value, by pointer, by reference.
#include <iostream>
#include <utility>

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;
}

int main() {
    int a = 13, b = 3;
    std::cout << "start:                  a = " << a << ", b = " << b << '\n';

    swapByValue(a, b);
    std::cout << "after swapByValue:      a = " << a << ", b = " << b << '\n';

    swapByPointer(&a, &b);
    std::cout << "after swapByPointer:    a = " << a << ", b = " << b << '\n';

    swapByReference(a, b);
    std::cout << "after swapByReference:  a = " << a << ", b = " << b << '\n';

    std::swap(a, b);                      // the library's reference-based swap
    std::cout << "after std::swap:        a = " << a << ", b = " << b << '\n';
    return 0;
}
```

```
start:                  a = 13, b = 3
  inside swapByValue:     x = 3, y = 13
after swapByValue:      a = 13, b = 3
after swapByPointer:    a = 3, b = 13
after swapByReference:  a = 13, b = 3
after std::swap:        a = 3, b = 13
```

Follow the variables through **Figure 6.8**.

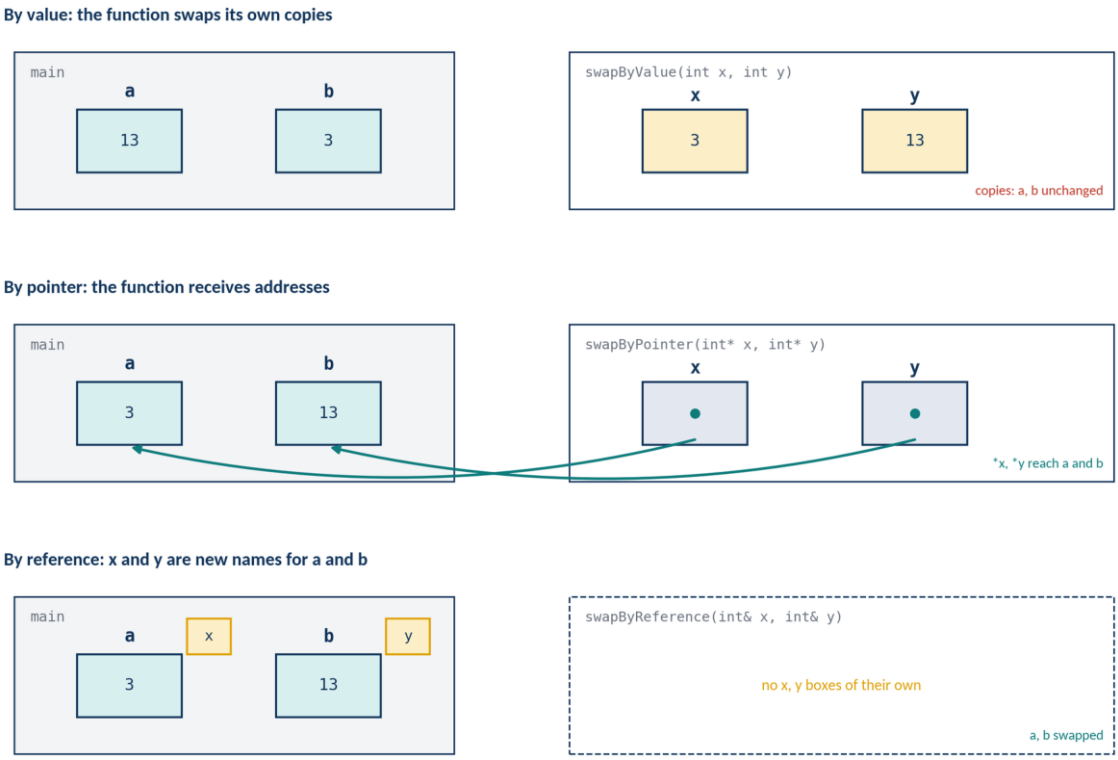


Figure 6.8 The same swap written three ways. By value the function exchanges its own copies; by pointer it exchanges the caller's variables through their addresses; by reference its parameters are the caller's variables under new names.

By value. `swapByValue(a, b)` creates `x` $=13$ and `y` $=3$ in its own frame. The body exchanges them, and the line printed from inside proves it: `x = 3, y = 13`. Then the function returns, its frame is popped, and the swapped copies vanish. `main`'s `a` and `b` never took part. This is not a bug in C++; it is exactly what pass by value means, and it is the reason the mode is safe: *a function called by value cannot damage its caller's variables.*

By pointer. `swapByPointer(&a, &b)` passes the addresses of `a` and `b`. The parameters `x` and `y` are pointers, and the body works through them: `t = *x` reads `main`'s `a`, `*x = *y` writes `main`'s `a` from `main`'s `b`, and `*y = t` writes `main`'s `b`. The caller's variables are exchanged: `a = 3, b = 13`. Two things are visible in the source that a reader can check: the call site says `&a`, warning the reader that `a` may change, and the function checks both pointers against `nullptr` before using them, because a pointer parameter can be null.

By reference. `swapByReference(a, b)` looks at the call site exactly like the by-value version, yet the body's `x` *is* `a` and `y` *is* `b`. No copies, no addresses to follow, no null to check. The variables, already swapped back to their original values by the pointer call's reversal, are exchanged once more. The standard library's `std::swap` is written the same way.

> **EXAMPLE.** Trace the three calls starting from `a = 13, b = 3`. After `swapByValue`: $a = 13, b = 3$, unchanged. After `swapByPointer`: $a = 3, b = 13$. After `swapByReference`: the values are exchanged again, so $a = 13, b = 3$. After `std::swap`: $a = 3, b = 13$. Every line of the program's output matches this trace.

6.10.4 The cost of passing by value

Swapping is about *whether the caller changes*. There is a second reason to avoid pass by value, and it matters even when the function only reads: a copy costs time and memory proportional to

the size of the object copied. If an object occupies s bytes and a function taking it by value is called k times, the program copies

$$\text{bytes copied} = k \cdot s \quad \text{by value,} \qquad 0 \quad \text{by reference or pointer} \tag{6.6}$$

and for objects that own heap memory, such as `std::string` and `std::vector`, each copy also allocates and later frees memory of its own. For an `int` ($s = 4$) the copy is cheaper than following a reference, so small built-in types are normally passed by value. For a `Book` with three strings, and above all for a container of thousands of elements, Equation (6.6) decides the matter: pass by `const` reference. Section 6.11 measures it.

6.10.5 Choosing a mode

Table 6.7 summarizes the choice. It restates the Core Guidelines' advice on parameter passing [12] in terms of the questions a programmer actually asks.

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)

Returning *several* results is a related question. The classic approach, and C's only approach, is the *output parameter*: pass pointers or references and let the function write into them, as `minMaxByPointer` and `minMaxByReference` do in the solution to Exercise 6.12. Modern C++ usually prefers to return a small structure or a `std::pair` and unpack it with a *structured binding*, `auto [lo, hi] = minMax(v);`, because a return value is easier to read at the call site than an argument that is secretly written.

6.11 `const` Correctness

A *const* object is one the program promises not to change, and the compiler holds it to the promise. With pointers there are *two* things that could be constant, the pointer and the object it points to, and so there are four combinations. **Table 6.8** lists them. The reliable way to read them is *from right to left*, starting at the variable name:

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

(`int const* p` is a legal alternative spelling of `const int* p`; the `const` applies to what is on its left, or to the right if nothing is on its left.) **Listing 6.14** tries each combination and also demonstrates a *const member function*: `int age(int now) const` promises not to change the `Book` it is called on, which is exactly what allows it to be called on the `const Book novel`.

Listing 6.14 `14_const_correct.cpp`: the four `const` forms, a const member function, and the cost of copying measured by a copy counter.

```
// const correctness, and what a by-value parameter really costs.
#include <iostream>
#include <string>
#include <utility>

struct Book {
    std::string id, title;
    int year;
    static long copies;                   // how many Books have been copied
    Book(std::string i, std::string t, int y) : id(std::move(i)), title(std::move(t)),
year(y) {}
    Book(const Book& other) : id(other.id), title(other.title), year(other.year) {
++copies; }
    int age(int now) const { return now - year; }   // const: does not change *this
};
long Book::copies = 0;

int ageByValue(Book b, int now) { return b.age(now); }          // copies the Book
int ageByConstRef(const Book& b, int now) { return b.age(now); } // no copy, read-only

int main() {
    int stock = 5, spare = 9;

    const int* readOnly = &stock;         // pointer to const int
    readOnly = &spare;                    // fine: the pointer itself may move
    std::cout << "*readOnly = " << *readOnly << '\n';

    int* const fixedPlace = &stock;       // const pointer to int
    *fixedPlace = 6;                      // fine: the int may change
    std::cout << "stock = " << stock << '\n';

    const int* const frozen = &stock;     // neither may change
    std::cout << "*frozen = " << *frozen << '\n';

    const Book novel("B4003", "Bumi Manusia", 1980);
    std::cout << "age in 2026 = " << novel.age(2026) << '\n';   // allowed: age is const

    long total = 0;
    for (int i = 0; i < 1000; ++i) total += ageByValue(novel, 2026);
    std::cout << "1000 calls by value:     " << Book::copies << " copies\n";
    Book::copies = 0;
    for (int i = 0; i < 1000; ++i) total += ageByConstRef(novel, 2026);
    std::cout << "1000 calls by const ref: " << Book::copies << " copies\n";
    std::cout << "(sum of ages " << total << ")\n";
    return 0;
}
```

```
*readOnly = 9
stock = 6
*frozen = 6
age in 2026 = 46
1000 calls by value:     1000 copies
1000 calls by const ref: 0 copies
(sum of ages 92000)
```

Everything in the listing compiles because it keeps its promises. Break them and the compiler stops you, which is the entire point of writing `const`:

```
*readOnly = 1;          // error: assignment of read-only location '* readOnly'
fixedPlace = &spare;    // error: assignment of read-only variable 'fixedPlace'
novel.reprint();        // error: passing 'const Book' as 'this' argument discards
qualifiers
```

(Those are GCC 13's exact messages for the three statements, compiled in a small test file.) The copy counter measures Equation (6.6): a thousand calls to `ageByValue` made a thousand copies of the `Book`, each copying two strings, while a thousand calls to `ageByConstRef` made none, and both

computed the same total. That is why a large read-only parameter is `const T&` in essentially all C++ code, and why Chapters 2 to 5 wrote `const Book&` wherever a book was only read.

Const correctness is the habit of marking everything `const` that does not need to change: parameters, member functions, local variables that are computed once. It costs nothing at run time, documents intent, and turns a whole family of accidental-modification bugs into compile errors.

6.12 Returning Pointers and References Safely

A function can return a pointer or a reference, and this is where dangling pointers are most often born. The rule is short: *never return the address of, or a reference to, an object that dies when the function returns.* The textbook example is a local variable:

```
int* makeCounter() {
    int counter = 0;
    return &counter;
}
```

```
local_addr.cpp:3:12: warning: address of local variable 'counter' returned [-Wreturn-
local-addr]
```

GCC catches this simple case with a warning, and it goes further: knowing that the address is useless, GCC 13 makes the function return a null pointer (we checked at `-O0` and at `-O2`), so the caller crashes at once. Less obvious versions, such as returning a pointer to an element of a local `std::vector`, or a reference to a member of a temporary, are not always caught.

What *is* safe? **Listing 6.15** shows the four safe patterns.

Listing 6.15 `15_find_book.cpp`: returning a pointer into the caller's array or `nullptr`, a reference into the caller's vector, and a new value.

```cpp
// Returning pointers and references safely.
#include <iostream>
#include <string>
#include <vector>

struct Book {
    std::string id, title;
    int year;
};

// 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;
}

// A non-const twin that lets the caller modify what was found.
Book* findById(Book* first, Book* last, const std::string& id) {
    for (Book* p = first; p != last; ++p) {
        if (p->id == id) return p;
    }
    return nullptr;
}

// Returning a reference to the oldest book: the vector outlives the call.
const Book& oldest(const std::vector<Book>& books) {
    std::size_t best = 0;
    for (std::size_t i = 1; i < books.size(); ++i) {
        if (books[i].year < books[best].year) best = i;
    }
    return books[best];
}

// Returning a new value: the result is a fresh object the caller owns.
Book makeReprint(const Book& original, int year) {
    Book copy = original;
    copy.title += " (reprint)";
    copy.year = year;
    return copy;
}
```

Listing 6.15 (Continued) *15_find_book.cpp: returning a pointer into the caller's array or nullptr, a reference into the caller's vector, and a new value.*

```
int main() {
    Book shelf[3] = {{"B4001", "Laskar Pelangi", 2005},
                     {"B4003", "Bumi Manusia", 1980},
                     {"B4005", "Cantik Itu Luka", 2002}};

    const Book* view = shelf;            // a read-only view of the same array
    for (const char* id : {"B4003", "B4999"}) {
        const Book* hit = findById(view, view + 3, id);
        if (hit) std::cout << id << " -> " << hit->title << '\n';
        else     std::cout << id << " -> not in the catalog\n";
    }

    if (Book* b = findById(shelf, shelf + 3, "B4005")) {
        b->year = 2004;                  // modify through the returned pointer
    }
    std::cout << "shelf[2].year is now " << shelf[2].year << '\n';

    std::vector<Book> books(shelf, shelf + 3);
    const Book& first = oldest(books);
    std::cout << "oldest: " << first.title << " (" << first.year << ")\n";

    Book fresh = makeReprint(first, 2025);
    std::cout << "reprint: " << fresh.title << " (" << fresh.year << ")\n";
    return 0;
}
```

```
B4003 -> Bumi Manusia
B4999 -> not in the catalog
shelf[2].year is now 2004
oldest: Bumi Manusia (1980)
reprint: Bumi Manusia (reprint) (2025)
```

1. **A pointer into something the caller owns.** `findById` searches the caller's array and returns the address of the matching element, or `nullptr`. The array outlives the call, so the pointer stays valid as long as the caller keeps the array unchanged. The `const` version serves readers; the non-`const` twin lets the caller modify the element it found, as `b->year = 2004` does. This is the pattern of every search in a linked structure, where the result is a pointer to a node.
2. **A reference into something the caller owns.** `oldest` returns `const Book&` to an element of the caller's vector: no copy, and never null, which suits a function that always has an answer (for a non-empty vector).
3. **A new value.** `makeReprint` returns a `Book` by value. Returning by value used to be feared as expensive; modern C++ constructs the result directly in the caller's variable in most cases (*return value optimization*), or moves it, so returning even large objects by value is the clean default for "make something new".
4. **A new heap object with an owner.** When the new object must live on the heap, for example because it will be linked into a structure, the function returns a `std::unique_ptr`, the subject of the next section, so that ownership is written into the type.

The phrase "as long as the caller keeps the array unchanged" in pattern 1 is not a formality. If the array were a `std::vector` and the caller appended to it, the pointer would dangle exactly as in the `stale` bug of Section 6.7.

6.13 Smart Pointers: Ownership Written in the Type

6.13.1 The problem with raw `new` and `delete`

Every bug in Section 6.7 comes from one question that raw pointers cannot answer: *who is responsible for deleting this object?* A raw `Book*` might point into an array, at a local variable, at a heap object someone else will delete, or at a heap object *you* must delete. Nothing in the type says which. Worse, even a correct `delete` can be skipped: if an exception is thrown or a function returns early between the `new` and the `delete`, the object leaks.

C++ answers with a technique called RAII, *Resource Acquisition Is Initialization*: tie every resource to an object whose destructor releases it. Because C++ guarantees that destructors run when objects go out of scope, by normal exit, early `return` or exception alike, the release cannot be forgotten. We already used RAII in `IntArray` in **Listing 6.8**. The standard library packages it for heap objects as *smart pointers*, defined in the header `<memory>` [1], [2].

6.13.2 `std::unique_ptr`: exactly one owner

A `std::unique_ptr<T>` holds a pointer to a heap object and *owns* it: when the `unique_ptr` is destroyed, it deletes the object. It cannot be copied, because two owners would delete twice; it can only be *moved*, which transfers ownership and leaves the source empty. It costs nothing over a raw pointer in size or speed. **Listing 6.16** gives `Book` a destructor that announces itself, so we can watch every object die at exactly the right moment.

Listing 6.16 `16_unique_ptr.cpp`: creating, moving, handing over and collecting `std::unique_ptr` owners; no `delete` appears anywhere.

```
// std::unique_ptr: exactly one owner, and delete happens by itself.
#include <iostream>
#include <memory>
#include <string>
#include <utility>
#include <vector>

struct Book {
    std::string id, title;
    Book(std::string i, std::string t) : id(std::move(i)), title(std::move(t)) {
        std::cout << "  [create " << id << "]\n";
    }
    ~Book() { std::cout << "  [destroy " << id << "]\n"; }
};

std::unique_ptr<Book> acquire(const std::string& id, const std::string& title) {
    return std::make_unique<Book>(id, title);    // ownership goes to the caller
}

void withdraw(std::unique_ptr<Book> b) {         // takes ownership (a "sink")
    std::cout << "  withdrawing " << b->title << '\n';
}                                                // b destroyed here

int main() {
    std::cout << "1. make_unique\n";
    std::unique_ptr<Book> a = acquire("B4001", "Laskar Pelangi");
    std::cout << "  a->title = " << a->title << '\n';

    std::cout << "2. move ownership from a to b\n";
    std::unique_ptr<Book> b = std::move(a);      // a copy would not compile
    std::cout << std::boolalpha << "  a is empty: " << (a == nullptr)
              << ", b->id = " << b->id << '\n';

    std::cout << "3. hand b to a function that keeps it\n";
    withdraw(std::move(b));
    std::cout << "  back in main, b is empty: " << (b == nullptr) << '\n';
```

Listing 6.16 (Continued) *16_unique_ptr.cpp: creating, moving, handing over and collecting std::unique_ptr owners; no delete appears anywhere.*

```
    std::cout << "4. a vector of owners\n";
    {
        std::vector<std::unique_ptr<Book>> shelf;
        shelf.push_back(acquire("B4002", "Filosofi Kopi"));
        shelf.push_back(acquire("B4003", "Bumi Manusia"));
        Book* observer = shelf[1].get();         // non-owning, just looks
        std::cout << "  observer sees " << observer->title << '\n';
        std::cout << "  leaving the block...\n";
    }                                            // the vector destroys both

    std::cout << "5. an owned array\n";
    std::unique_ptr<int[]> counts = std::make_unique<int[]>(4);   // zero-filled
    counts[2] = 7;
    std::cout << "  counts = " << counts[0] << ' ' << counts[1] << ' '
              << counts[2] << ' ' << counts[3] << '\n';
    std::cout << "end of main\n";
    return 0;
}
```

```
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
4. a vector of owners
   [create B4002]
   [create B4003]
   observer sees Bumi Manusia
   leaving the block...
   [destroy B4002]
   [destroy B4003]
5. an owned array
   counts = 0 0 7 0
end of main
```

Trace the output against the code.

1. `std::make_unique<Book>(id, title)` allocates a `Book` on the heap and returns the owning pointer. The factory function `acquire` returns it, and ownership passes to `a` in `main`.
2. `std::unique_ptr<Book> b = std::move(a);` transfers ownership. `std::move` does not move anything by itself; it marks `a` as something that may be taken from. Afterward `a` is empty (it compares equal to `nullptr`) and `b` owns *B4001*. Writing `b = a;` without `std::move` does not compile, and GCC's message says why: `use of deleted function 'std::unique_ptr<...>::unique_ptr(const std::unique_ptr<...>&)'`. The copy operation has been deliberately deleted.
3. `withdraw(std::move(b))` hands ownership to a function that takes a `unique_ptr` by value. When `withdraw` returns, its parameter is destroyed, and with it the `Book`: `[destroy B4001]` is printed *inside* step 3, before `main` continues.
4. A vector of `unique_ptr` owns a whole shelf. `shelf[1].get()` returns a raw, *non-owning* pointer for someone who only needs to look. When the block ends, the vector is destroyed, it destroys its `unique_ptr`s in order, and they destroy the books.
5. `std::make_unique<int[]>(4)` owns an array; its destructor uses `delete[]`, so the mismatch bug of Section 6.7 cannot happen.

Run under Valgrind or AddressSanitizer, **Listing 6.16** is clean: every `new` hidden inside `make_unique` was matched by a `delete` hidden inside a destructor, and the programmer wrote neither.

6.13.3 `std::shared_ptr` and `std::weak_ptr`: counted owners

Sometimes an object genuinely has several owners, none of which knows which will finish last. In Pustaka Ceria, one book can sit on both the *New Arrivals* list and the *Recommended* list; it should disappear when it has left both. A `std::shared_ptr<T>` handles this with a *reference count*. Next to the object, in a small *control block*, it keeps the number of `shared_ptr`s that currently own it. Copying a `shared_ptr` increments the count; destroying or resetting one decrements it; and the object is destroyed exactly when

$$\text{use\_count} = 0.$$

A `std::weak_ptr` observes a shared object *without* owning it: it does not keep the count up, and before using the object you call `lock()`, which returns a `shared_ptr` if the object still exists or an empty one if it has gone. **Figure 6.9** contrasts the two kinds of ownership.

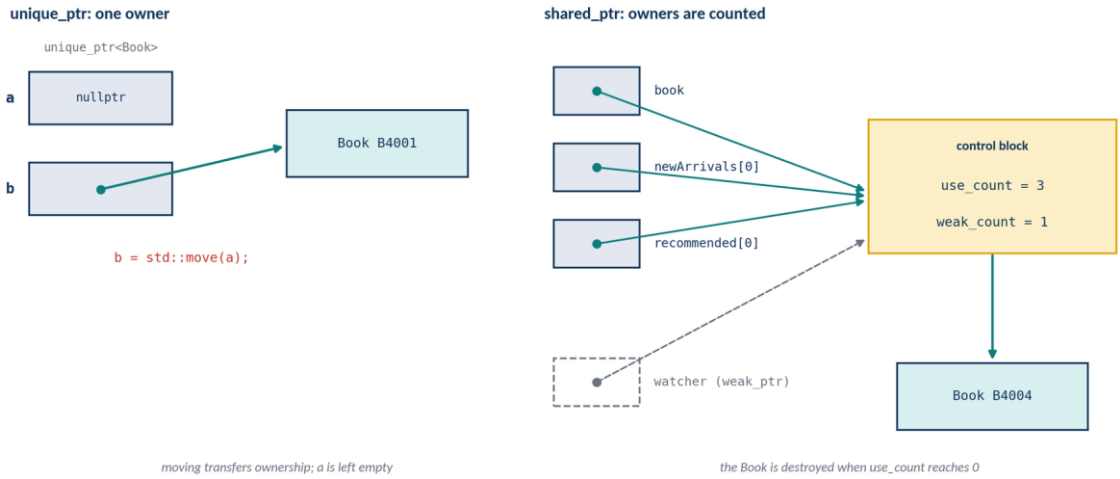


Figure 6.9 Left: a `unique_ptr` has one owner, and moving transfers it. Right: three `shared_ptr`s share one `Book` through a control block that counts them; a `weak_ptr` observes without being counted.

Listing 6.17 `17_shared_ptr.cpp`: a book shared by two lists and destroyed when the last owner lets go, and a pair of members linked with `weak_ptr` so that they can be destroyed.

```
// std::shared_ptr and std::weak_ptr: shared ownership, counted.
#include <iostream>
#include <memory>
#include <string>
#include <utility>
#include <vector>

struct Book {
    std::string id;
    explicit Book(std::string i) : id(std::move(i)) { std::cout << "  [create " << id <<
"]\n"; }
    ~Book() { std::cout << "  [destroy " << id << "]\n"; }
};

struct Member {                          // two members who recommend each other
    std::string name;
    std::shared_ptr<Member> friendStrong; // owning link: can form a cycle
    std::weak_ptr<Member> friendWeak;     // observing link: cannot
    explicit Member(std::string n) : name(std::move(n)) {}
    ~Member() { std::cout << "  [destroy member " << name << "]\n"; }
};

int main(int argc, char* argv[]) {
    std::cout << "1. one book on two lists\n";
    std::vector<std::shared_ptr<Book>> newArrivals, recommended;
    {
        auto book = std::make_shared<Book>("B4004");
        std::cout << "  use_count = " << book.use_count() << '\n';
        newArrivals.push_back(book);
        recommended.push_back(book);
        std::cout << "  use_count = " << book.use_count() << " (book + 2 lists)\n";
    }
    std::cout << "  local gone, use_count = " << newArrivals[0].use_count() << '\n';
    std::weak_ptr<Book> watcher = newArrivals[0];
    newArrivals.clear();
    std::cout << "  newArrivals cleared, use_count = " << recommended[0].use_count() <<
'\n';
    recommended.clear();                 // the last owner lets go
    std::cout << "  watcher expired: " << std::boolalpha << watcher.expired() << '\n';
```

Listing 6.17 (Continued) *17_shared_ptr.cpp: a book shared by two lists and destroyed when the last owner lets go, and a pair of members linked with weak_ptr so that they can be destroyed.*

```
    bool cycle = argc > 1 && std::string(argv[1]) == "cycle";
    std::cout << "2. two members who point at each other ("
              << (cycle ? "shared_ptr both ways" : "weak_ptr links") << ")\n";
    {
        auto ani = std::make_shared<Member>("Ani");
        auto budi = std::make_shared<Member>("Budi");
        if (cycle) {
            ani->friendStrong = budi;
            budi->friendStrong = ani;     // a cycle of owners
        } else {
            ani->friendWeak = budi;
            budi->friendWeak = ani;
        }
        if (auto f = ani->friendWeak.lock()) {
            std::cout << "  Ani's friend is " << f->name << '\n';
        }
        std::cout << "  leaving the block...\n";
    }
    std::cout << "end of main\n";
    return 0;
}
```

```
1. one book on two lists
  [create B4004]
  use_count = 1
  use_count = 3 (book + 2 lists)
  local gone, use_count = 2
  newArrivals cleared, use_count = 1
  [destroy B4004]
  watcher expired: true
2. two members who point at each other (weak_ptr links)
  Ani's friend is Budi
  leaving the block...
  [destroy member Budi]
  [destroy member Ani]
end of main
```

Listing 6.17 runs both situations. The first part follows the count: 1 after `make_shared`, 3 once both lists hold copies, 2 when the local `book` goes out of scope, 1 after `newArrivals` is cleared, and 0, with `[destroy B4004]`, when `recommended` lets go. The `watcher` then reports that its object has expired.

The second part shows the one thing reference counting cannot do by itself. Ani and Budi recommend each other. If each held the other with a `shared_ptr`, each count would stay at 1 forever after the local names disappear, because each object is kept alive by the other: a *reference cycle*, and a leak. Run the program with the word `cycle` and the two `[destroy member ...]` lines are missing; AddressSanitizer's leak checker then reports two leaked control blocks of 80 bytes each. Linking them with `weak_ptr`, the default run above, breaks the cycle and both are destroyed. The general rule: owners point *down* a structure with `unique_ptr` or `shared_ptr`, and back-links point *up* with `weak_ptr` or a raw non-owning pointer. The doubly linked lists of Chapter 9 and the parent pointers of trees in Chapter 11 are exactly such back-links.

Table 6.9 summarizes the choice. In current C++ a raw pointer is still perfectly good, even preferred, as a *non-owning* observer, such as `observer` in **Listing 6.16** or the result of `findById` in **Listing 6.15**; what modern code avoids is a raw pointer that *owns*.

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

The C++ Core Guidelines condense this into rules R.20 to R.24 [12]: use `unique_ptr` or `shared_ptr` to represent ownership, prefer `unique_ptr` to `shared_ptr`, create them with `make_unique` and `make_shared`, and use `weak_ptr` to break cycles.

> **KEY INSIGHT.** Why, then, do Chapters 7 to 13 still write raw `new` and `delete` in their linked lists and trees? Because a first study of data structures is where you learn what those smart pointers do on your behalf. Writing the `delete`s yourself, and checking them with the tools of

> Section 6.7, is how the idea of ownership becomes concrete. In production code, the node that owns the rest of a list would hold its successor in a `unique_ptr`.

6.14 Java References and Python Names

6.14.1 One rule each

Java and Python have no pointer arithmetic, no `&`, no `*`, and no `delete`. Yet every non-trivial Java or Python program is full of the ideas of this chapter, hidden behind a single rule per language.

Java. A variable of a *primitive* type (`int`, `double`, `char`, `boolean` and four more) holds its value directly, like a C++ `int`. A variable of any *class or array* type holds a *reference*: in C++ terms, a pointer that cannot be moved with arithmetic and is followed automatically by the dot. Objects are created only with `new` and always on the heap; nobody deletes them; a *garbage collector* reclaims an object once no reference to it remains. And every argument, without exception, is **passed by value** [13]: a method receives a copy of the argument. For a primitive, that copy is the number; for an object, the copy is *the reference*, so the method and the caller refer to the same object.

Python. Every value is an object, and every variable is a *name bound to an object*. Assignment `x = y` never copies an object; it binds the name `x` to the object that `y` is bound to. A function call binds each parameter name to the object passed, so the function and the caller share the object [14]. This rule has several names in the literature. The Python documentation says arguments are passed by *assignment*; programmers often say *pass by object reference*; and the term *call by sharing*, coined for the language CLU by Liskov and her colleagues in the 1970s [15], describes the same behavior in both Java and Python.

Put in C++ language: a Java object variable and a Python name behave like a C++ *pointer passed by value*. The callee can follow its copy of the pointer and change the object at the other end, and the caller will see the change. The callee can also make its own copy of the pointer point somewhere else, and the caller will *not* notice. The first is **mutation**; the second is **rebinding**. **Figure 6.10** draws both, and the difference between them is the single most important idea in this section.

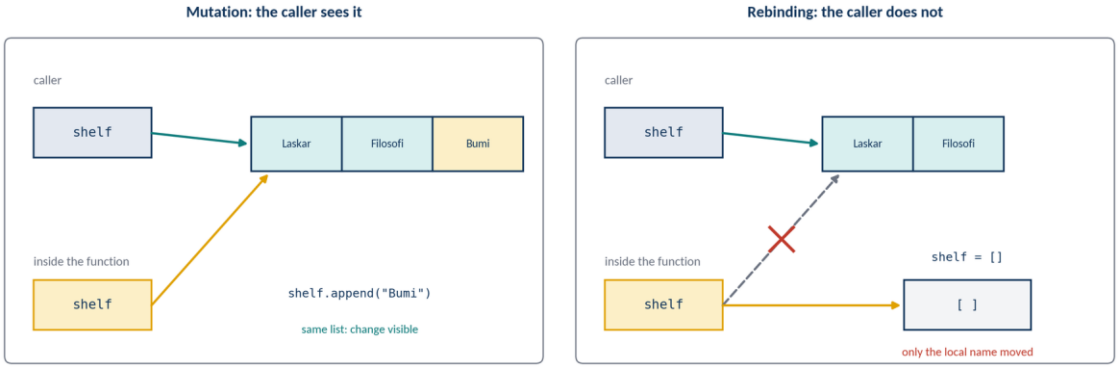


Figure 6.10 Left: the function mutates the list its parameter refers to, and the caller sees the change. Right: the function rebinds its parameter to a new list, and the caller's name still refers to the old one.

6.14.2 Mutation versus rebinding, in Java

Listing 6.18 runs the experiment in Java on a shelf of titles.

Listing 6.18 `RefsDemo.java`: aliasing, mutation through a parameter, rebinding a parameter, a primitive argument, a real copy, `==` versus `equals`, and a null reference.

```java
import java.util.ArrayList;
import java.util.List;

// Mutation versus rebinding: what a method can and cannot do to its caller.
public class RefsDemo {
    static List<String> nothing = null;

    static void addBook(List<String> shelf) {
        shelf.add("Bumi Manusia");                  // mutates the caller's list
    }

    static void replaceShelf(List<String> shelf) {
        shelf = new ArrayList<>();                  // rebinds the local copy only
        shelf.add("Negeri 5 Menara");
    }

    static void bumpYear(int year) {
        year++;                                     // a copy of the int
    }

    public static void main(String[] args) {
        List<String> shelf = new ArrayList<>(List.of("Laskar Pelangi"));
        List<String> alias = shelf;                 // two variables, one list
        alias.add("Filosofi Kopi");
        System.out.println("shelf after alias.add:   " + shelf);
        System.out.println("shelf == alias: " + (shelf == alias));

        addBook(shelf);
        System.out.println("shelf after addBook:     " + shelf);
        replaceShelf(shelf);
        System.out.println("shelf after replaceShelf: " + shelf);

        int year = 2005;
        bumpYear(year);
        System.out.println("year after bumpYear:     " + year);

        List<String> copy = new ArrayList<>(shelf);  // a real copy
        copy.clear();
        System.out.println("copy cleared, shelf has " + shelf.size() + " books");

        String a = new String("Pustaka"), b = new String("Pustaka");
        System.out.println("a == b: " + (a == b) + ", a.equals(b): " + a.equals(b));
```

Listing 6.18 (Continued) *RefsDemo.java: aliasing, mutation through a parameter, rebinding a parameter, a primitive argument, a real copy, == versus equals, and a null reference.*

```java
        try {
            nothing.add("x");                       // nothing is a null field
        } catch (NullPointerException e) {
            System.out.println("NullPointerException: " + e.getMessage());
        }
    }
}
```

```
shelf after alias.add:   [Laskar Pelangi, Filosofi Kopi]
shelf == alias: true
shelf after addBook:     [Laskar Pelangi, Filosofi Kopi, Bumi Manusia]
shelf after replaceShelf: [Laskar Pelangi, Filosofi Kopi, Bumi Manusia]
year after bumpYear:     2005
copy cleared, shelf has 3 books
a == b: false, a.equals(b): true
NullPointerException: Cannot invoke "java.util.List.add(Object)" because "RefsDemo.nothing"
is null
```

Line by line:

- `alias = shelf` copies a reference. There is one list with two names, so `alias.add` changes what `shelf` shows, and `shelf == alias` is `true`. For objects, Java's `==` compares *references*, exactly like comparing two C++ pointers.
- `addBook(shelf)` receives a copy of the reference and follows it: mutation, visible to the caller.
- `replaceShelf(shelf)` receives a copy of the reference and then *reassigns the copy* to a brand-new list. The caller's `shelf` still refers to the old list, which is unchanged. This is precisely `swapByValue` from **Listing 6.13**, with a reference in place of an `int`.
- `bumpYear(year)` receives a copy of a primitive `int`; nothing the method does can reach `main`'s `year`.
- `new ArrayList<>(shelf)` makes a genuine copy, a new list with the same elements; clearing the copy leaves `shelf` alone.
- Two `String` objects with the same characters are different objects: `a == b` is `false`, while `a.equals(b)`, which compares contents, is `true`. Using `==` on strings is one of the most common Java mistakes, and it is the pointer-versus-value distinction of Section 6.3.2 in disguise.
- Following a `null` reference throws a `NullPointerException`. Since JDK 15 the message names the exact expression that was null [16], as the last line shows; the C++ equivalent is the segmentation fault of Section 6.7.

6.14.3 Mutation versus rebinding, in Python

Listing 6.19 is the same experiment in Python, plus one trap that exists only there.

Listing 6.19 `refs_demo.py`: the same experiment with Python names, plus the mutable-default-argument trap.

```
# Mutation versus rebinding: what a function can and cannot do to its caller.
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 = ["Laskar Pelangi"]
alias = shelf                          # two names, one list
alias.append("Filosofi Kopi")
print(f"shelf after alias.append:  {shelf}")
print(f"shelf is alias: {shelf is alias}")

add_book(shelf)
print(f"shelf after add_book:      {shelf}")
replace_shelf(shelf)
print(f"shelf after replace_shelf: {shelf}")

year = 2005
bump_year(year)
print(f"year after bump_year:      {year}")

copy = list(shelf)                     # a real (shallow) copy
copy.clear()
print(f"copy cleared, shelf has {len(shelf)} books")

a, b = "".join(["Pus", "taka"]), "".join(["Pust", "aka"])
print(f"a is b: {a is b}, a == b: {a == b}")

print(f"add_tag('novel')  -> {add_tag('novel')}")
print(f"add_tag('sejarah') -> {add_tag('sejarah')}   (the same default list!)")
```

Listing 6.19 (Continued) *refs_demo.py: the same experiment with Python names, plus the mutable-default-argument trap.*

```
nothing = None
try:
    nothing.append("x")
except AttributeError as e:
    print(f"AttributeError: {e}")
```

```
shelf after alias.append:  ['Laskar Pelangi', 'Filosofi Kopi']
shelf is alias: True
shelf after add_book:      ['Laskar Pelangi', 'Filosofi Kopi', 'Bumi Manusia']
shelf after replace_shelf: ['Laskar Pelangi', 'Filosofi Kopi', 'Bumi Manusia']
year after bump_year:      2005
copy cleared, shelf has 3 books
a is b: False, a == b: True
add_tag('novel')  -> ['novel']
add_tag('sejarah') -> ['novel', 'sejarah']   (the same default list!)
AttributeError: 'NoneType' object has no attribute 'append'
```

The results match the Java program line for line, with Python's own vocabulary: `is` asks whether two names are bound to the same object (Java's `==` on references), while `==` compares values (Java's `equals`). Rebinding the parameter in `replace_shelf` has no effect on the caller, and neither does `year += 1` in `bump_year`. The reason for the latter is worth spelling out: Python's `int` is *immutable*, so `year += 1` cannot change the object 2005; it creates the object 2006 and rebinds the *local* name to it. Strings and tuples are immutable too. Lists, dictionaries, sets and ordinary class instances are mutable, and those are the objects a function can change on its caller's behalf.

The two lines about `add_tag` show the trap. A default value such as `tags=[]` is evaluated *once*, when the `def` statement runs, and the resulting list is stored with the function. Every call that relies on the default gets the *same* list, which therefore remembers previous calls: the second call returns `['novel', 'sejarah']`. The standard fix is `def add_tag(tag, tags=None):` followed by `if tags is None: tags = []`, which creates a fresh list on each call.

6.14.4 The swap story in Java and Python

Neither language can write a function that swaps two of the caller's variables, and now we can say exactly why: the function receives copies of values or of references, and rebinding copies never reaches the caller. **Listing 6.20** and **Listing 6.21** show what *can* be done.

Listing 6.20 `SwapDemo.java`: Java can swap the contents of two objects, or two slots of one array, but never two of the caller's variables.

```
// The swap story in Java: arguments are always passed by value, and for an
// object that value is a reference.
public class SwapDemo {
    static void swapInts(int x, int y) {             // copies of the ints
        int t = x; x = y; y = t;
    }

    static void swapBoxes(int[] x, int[] y) {        // copies of the references
        int[] t = x; x = y; y = t;                   // rebinding the copies only
    }

    static void swapContents(int[] x, int[] y) {     // mutate the shared objects
        int t = x[0]; x[0] = y[0]; y[0] = t;
    }

    static void swapInArray(int[] a, int i, int j) { // the usual Java idiom
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    public static void main(String[] args) {
        int a = 13, b = 3;
        swapInts(a, b);
        System.out.println("after swapInts:     a = " + a + ", b = " + b);

        int[] boxA = {13}, boxB = {3};
        swapBoxes(boxA, boxB);
        System.out.println("after swapBoxes:    boxA[0] = " + boxA[0] + ", boxB[0] = " +
boxB[0]);
        swapContents(boxA, boxB);
        System.out.println("after swapContents: boxA[0] = " + boxA[0] + ", boxB[0] = " +
boxB[0]);

        int[] pair = {13, 3};
        swapInArray(pair, 0, 1);
        System.out.println("after swapInArray:  pair = " + pair[0] + ", " + pair[1]);
    }
}
```

```
after swapInts:     a = 13, b = 3
after swapBoxes:    boxA[0] = 13, boxB[0] = 3
after swapContents: boxA[0] = 3, boxB[0] = 13
after swapInArray:  pair = 3, 13
```

Listing 6.21 `swap_demo.py`: the same attempts in Python, ending with the tuple assignment that swaps in the caller.

```
# The swap story in Python: a function receives references to objects and
# can rebind only its own local names.
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


a, b = 13, 3
swap_ints(a, b)
print(f"after swap_ints:     a = {a}, b = {b}")

box_a, box_b = [13], [3]
swap_boxes(box_a, box_b)
print(f"after swap_boxes:    box_a[0] = {box_a[0]}, box_b[0] = {box_b[0]}")
swap_contents(box_a, box_b)
print(f"after swap_contents: box_a[0] = {box_a[0]}, box_b[0] = {box_b[0]}")

a, b = b, a                             # the Python idiom: rebind in the caller
print(f"after a, b = b, a:   a = {a}, b = {b}")
```

```
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
```

`swapInts` and `swap_ints` fail for the same reason as `swapByValue`. `swapBoxes` and `swap_boxes` fail too, and more instructively: they exchange *references*, but only the function's own copies of them. `swapContents` and `swap_contents` succeed, because they mutate the two shared objects instead of rebinding names; this is the Java and Python analogue of `swapByPointer`. In real Java code the usual form is `swapInArray(a, i, j)`, which is how every sorting routine in Chapter 3 exchanged elements. In Python the idiomatic swap is not a function at all: `a, b = b, a` builds a tuple of the two objects and unpacks it into the two names, in the caller's own scope.

6.14.5 Who frees the memory?

C++ frees heap objects when the program says so, directly with `delete` or through a smart pointer's destructor. Java and Python free them automatically, but not in the same way, and the difference connects neatly to Section 6.13.

CPython, the standard Python implementation, keeps a reference count in every object, exactly like the control block of a `shared_ptr`, and frees an object the moment its count drops to zero. `sys.getrefcount` shows the count. Because counting alone cannot free cycles, CPython adds a *cycle collector*, the `gc` module [17], that periodically looks for groups of objects reachable only from each other. The program `shared_demo.py` in the chapter's code folder shows both: a book destroyed as soon as the last list lets go, and two members in a cycle that survive until

`gc.collect()` finds them. Java uses no counts at all. Its *tracing* garbage collector periodically starts from the variables that are certainly alive (local variables of running methods, static fields) and marks everything reachable from them; whatever is unmarked is garbage, cycles included. The price is that the moment of reclamation is unpredictable, which is why Java and Python both provide an explicit construct, `try`-with-resources and `with`, for resources such as files that must be released at a known moment: the RAII idea again, spelled differently.

6.14.6 Three languages side by side

Table 6.10 gathers the comparison.

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`
Freeing heap objects	`delete`, or a smart pointer's destructor	tracing garbage collector	reference counting plus cycle collector
Dangling pointer or double free possible?	yes	no	no
Leaks possible?	yes	yes, by keeping unneeded references	yes, by keeping unneeded references

The last two rows deserve a comment. Garbage collection removes dangling pointers and double frees completely, which is a large part of why Java and Python are called memory-safe languages. It does not remove leaks. An object that is still reachable, because a long-lived list or cache still refers to it, is never collected even if the program will never use it again. The Java and Python versions of `mem_bugs` in the code folder show this: the `leak` case adds arrays to a static list, and the garbage collector rightly refuses to free them.

6.15 The Cost of Each Operation

A data structures book owes its reader the cost of every operation. **Table 6.11** gives them for this chapter's tools, with n the number of elements involved and s the size in bytes of an object.

Table 6.11 The running time of the operations in this chapter.

Operation	Cost	Why
`&x`, `*p`, `p->m`, `p + i`, `q - p`, `p == q`	$\Theta(1)$	a few machine instructions (Equation (6.1), Equation (6.2))
`a[i]` for an array or through a pointer	$\Theta(1)$	one address computation and one access (Equation (6.3))
following a chain of k `next` pointers	$\Theta(k)$	one dereference per link; no shortcut exists
pass or return by reference or pointer	$\Theta(1)$	one address
pass by value	$\Theta(s)$	a copy of every byte, plus any heap memory the object owns (Equation (6.6))
`new T`, `delete p`	$O(1)$ typical, not guaranteed	depends on the allocator; far slower than a stack variable
`new T[n]`, `delete[] p`	$\Theta(n)$ for types with constructors or destructors	each element is constructed or destroyed
push onto a doubling array	$O(1)$ amortized, $\Theta(n)$ worst single push	Equation (6.5)
`unique_ptr` operations	same as raw pointer	no extra data
copying or destroying a `shared_ptr`	$O(1)$, slower than raw	updates an atomic counter

Two lines of **Table 6.11** will shape everything that follows. Reaching element i of an array is $\Theta(1)$; reaching the i-th node of a chain is $\Theta(i)$. And inserting into the middle of an array means shifting everything after it, $\Theta(n)$, while inserting into a chain means changing two pointers, $\Theta(1)$ once you are there. Chapter 7 is the full development of that trade.

6.16 Profitina in Practice

Profitina, the AI-visibility platform introduced in earlier chapters, measures how ready a website is to be cited by AI answer engines and helps its owner fix what it finds [18]. Its web application is written in Python with the FastAPI framework, and its language models (Qwen2.5, served locally through Ollama) run on the same server. Two ideas from this chapter appear in its daily work in a very concrete form.

Passing results through a pipeline without copying them. A scan of a website is a sequence of measurements: fetch the page, extract its text, check its structured data, check its technical signals, compute a score, and explain each signal. In Python each stage is a function that receives the scan's result object, and by the rule of Section 6.14 every stage receives *the same object*, not a copy. That is exactly what a pipeline wants. A page's extracted text can run to many thousands of words; if each stage copied it, as pass by value would in C++, the copies would cost time and memory proportional to the page size at every step (Equation (6.6)). Sharing is free. The same rule brings the obligation that sharing always brings: a stage that is only supposed to read must not mutate, because every later stage would see the change. Profitina's release gate, an automated test suite of well over a thousand tests that must pass before any release, includes tests of exactly this kind, which feed a stage a result and check that what should be untouched is untouched. The mutable-default-argument trap of **Listing 6.19** is a classic source of such bugs in Python web services, because a default list created once is shared by every request the server ever handles.

One expensive object, many users. A local language model occupies gigabytes of memory, and it would be absurd for every request to load its own copy. The model is loaded once and every request that needs it works with a reference to that one shared instance, the pattern of `shared_ptr` and of Python's reference counting in Sections 6.13 and 6.14. When no model is available, the application says so rather than silently serving a template: a "null" result is handled explicitly, never dereferenced blindly, which is the `nullptr` discipline of Section 6.8 at the scale of a whole service.

The runnable case study for this chapter is **Profitina Laundry**, the small laundry business used as the hands-on example in all the Profitina books. Its orders wait in two queues, *regular* and *express*, and a customer can pay to move an order from one to the other. **Listing 6.22** implements this in C++ with every order owned by exactly one queue through a `unique_ptr`. Promoting an order moves the *pointer*, not the order, so a non-owning observer keeps working. It also contrasts a quote computed on a copy with a discount applied through a reference.

Listing 6.22 `18_laundry.cpp`: Profitina Laundry's queues own their orders through `unique_ptr`; promotion moves ownership, not the order.

```
// Profitina Laundry: orders owned by queues, moved between them without
// copying, observed by a raw pointer, and updated through a reference.
#include <deque>
#include <iostream>
#include <memory>
#include <string>
#include <utility>

struct Order {
    int number;
    std::string customer;
    double kilograms;
    long price;                             // in rupiah
    Order(int n, std::string c, double kg, long p)
        : number(n), customer(std::move(c)), kilograms(kg), price(p) {}
    ~Order() { std::cout << "  [order " << number << " released]\n"; }
};

using Queue = std::deque<std::unique_ptr<Order>>;

void applyDiscount(Order& o, int percent) {        // by reference: changes the order
    o.price -= o.price * percent / 100;
}

long quoteWithDiscount(Order o, int percent) {     // by value: works on a copy
    o.price -= o.price * percent / 100;
    return o.price;
}

// Move the order with this number from one queue to another. Only the
// pointer moves; the Order object itself stays where it is on the heap.
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;
}
```

Listing 6.22 (Continued) *18_laundry.cpp: Profitina Laundry's queues own their orders through unique_ptr; promotion moves ownership, not the order.*

```
void print(const char* name, const Queue& q) {
    std::cout << name << ":";
    for (const auto& o : q) {
        std::cout << "  #" << o->number << ' ' << o->customer << " Rp" << o->price;
    }
    std::cout << '\n';
}

int main() {
    Queue regular, express;
    regular.push_back(std::make_unique<Order>(101, "Sari", 3.0, 21000));
    regular.push_back(std::make_unique<Order>(102, "Joko", 5.5, 38500));
    regular.push_back(std::make_unique<Order>(103, "Rina", 2.0, 14000));
    print("regular", regular);

    const Order* watched = regular[1].get();       // observe order 102
    promote(regular, express, 102);
    print("regular", regular);
    print("express", express);
    std::cout << "watched still sees #" << watched->number
              << " (same object, new owner)\n";

    std::cout << "quote with 10% off: Rp" << quoteWithDiscount(*express[0], 10)
              << ", stored price still Rp" << express[0]->price << '\n';
    applyDiscount(*express[0], 10);
    std::cout << "after applyDiscount: Rp" << express[0]->price << '\n';
    std::cout << "(quoteWithDiscount copied the order, and the copy was released)\n";

    express.pop_front();                           // order 102 collected
    std::cout << "end of main\n";
    return 0;
}
```

```
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]
after applyDiscount: Rp34650
(quoteWithDiscount copied the order, and the copy was released)
  [order 102 released]
end of main
  [order 101 released]
  [order 103 released]
```

Read the ownership in the output. Order 102 is created in `regular`, moved to `express` with `std::move`, and never copied; the raw pointer `watched`, taken before the move, still sees `#102` afterward because the `Order` object itself never moved in memory, only the `unique_ptr` that owns it changed queues. `quoteWithDiscount` takes the order *by value*, computes $38500 - 3850 = 34650$ on its copy, and leaves the stored price at 38500; the first `[order 102 released]` line is that copy being destroyed at the end of the statement. `applyDiscount` takes the order *by reference* and changes the real price. Finally `pop_front` collects order 102, and the remaining orders are released automatically when the queues are destroyed at the end of `main`. There is no `delete` anywhere, and Valgrind reports all heap blocks freed.

Listing 6.23 is the same program in Python. Promotion is a matter of removing a reference from one `deque` and appending it to another; `watched is express[0]` prints `True` because it is the very

same object; and the quote must ask for a copy explicitly with `copy.copy`, since Python never copies on its own.

Listing 6.23 `laundry.py`: the same queues in Python, where moving an order between queues moves a reference.

```
# Profitina Laundry in Python: moving an order between queues moves a
# reference; a copy must be asked for explicitly.
from collections import deque
from copy import copy
from dataclasses import dataclass


@dataclass
class Order:
    number: int
    customer: str
    kilograms: float
    price: int                          # in rupiah


def apply_discount(order, percent):     # mutates the caller's order
    order.price -= order.price * percent // 100


def quote_with_discount(order, percent):
    c = copy(order)                     # work on an explicit copy
    c.price -= c.price * percent // 100
    return c.price


def promote(source, target, number):
    for order in source:
        if order.number == number:
            source.remove(order)        # the object is not destroyed ...
            target.append(order)        # ... it just changes queues
            return True
    return False


def show(name, queue):
    print(f"{name}:" + "".join(f"  #{o.number} {o.customer} Rp{o.price}" for o in queue))
```

Listing 6.23 (Continued) *laundry.py: the same queues in Python, where moving an order between queues moves a reference.*

```
regular, express = deque(), deque()
regular.append(Order(101, "Sari", 3.0, 21000))
regular.append(Order(102, "Joko", 5.5, 38500))
regular.append(Order(103, "Rina", 2.0, 14000))
show("regular", regular)
watched = regular[1]
promote(regular, express, 102)
show("regular", regular)
show("express", express)
print(f"watched still sees #{watched.number} (same object, new owner): {watched is
express[0]}")
print(f"quote with 10% off: Rp{quote_with_discount(express[0], 10)}, "
      f"stored price still Rp{express[0].price}")
apply_discount(express[0], 10)
print(f"after apply_discount: Rp{express[0].price}")
express.popleft()
print("end of main")
```

```
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): True
quote with 10% off: Rp34650, stored price still Rp38500
after apply_discount: Rp34650
end of main
```

6.17 Common Mistakes, and How to Avoid Them

1. **Confusing `q = p` with `*q = *p`.** The first moves an arrow, the second copies a value. Ask whether there is a star on the left (Section 6.3.1).
2. **Declaring several pointers on one line.** `int* p, q;` declares one pointer `p` and one plain `int` `q`; the star belongs to each name, not to the type. Declare one pointer per line.
3. **Using an uninitialized pointer.** Always initialize, if only to `nullptr`.
4. **Dereferencing without checking for null.** Any pointer that can be `nullptr`, especially the result of a search, must be checked on every path that dereferences it.
5. **Walking past the end of an array or string.** Use half-open loops `for (p = first; p != last; ++p)`, never `<=` with the length, and reset a pointer before reusing it, as the `"Danaya"` example teaches.
6. **Asking `sizeof` for the length of an array parameter.** Inside the function it is the size of a pointer. Pass the length, a pair of pointers, or a container.
7. **Returning the address of a local variable,** or a reference to one. Return a value, a pointer into caller-owned storage, or a smart pointer.
8. **Forgetting `delete`, deleting twice, or mixing `delete` with `new[]`.** Pair every `new` with one `delete` and every `new[]` with one `delete[]`; better, let `std::vector`, `std::unique_ptr` and `std::make_unique<T[]>` do it.
9. **Using a pointer, reference or iterator into a `std::vector` after it grows.** Growth moves the elements. Take the address after the last insertion, store indices instead, or call `reserve` first.
10. **Expecting `r = b` to re-seat a reference.** It assigns `b`'s value to the object `r` refers to.
11. **Passing large objects by value "because it is simpler".** A `const T&` costs nothing and reads the same at the call site.
12. **Writing `shared_ptr` cycles.** Two objects that own each other are never destroyed; make one of the links a `weak_ptr` or a raw non-owning pointer.
13. **(Java) Comparing strings or other objects with `==`.** That compares references; use `equals`.
14. **(Java, Python) Expecting a method to rebind the caller's variable.** Reassigning a parameter changes only the method's copy of the reference; return the new object instead.
15. **(Python) Using a mutable default argument.** Use `None` as the default and create the list or dictionary inside the function.

6.18 Chapter Summary

- Memory is a sequence of numbered bytes; an **address** is a byte's number, and a **pointer** is a variable whose value is an address. `&x` takes an address, `*p` follows one, and in a declaration `T*` means "pointer to `T`".
- A pointer's type decides how many bytes a dereference reads and how far arithmetic moves: `p + i` is `i · sizeof(T)` bytes further on (Equation (6.1)), and `q - p` counts elements

(Equation (6.2)). Arithmetic and ordering are defined only within one array and its one-past-the-end position.

- An array name decays to a pointer to its first element, and `a[i]` *is* `*(a + i)` (Equation (6.3)). An array passed to a function arrives as a pointer and loses its length. Two-dimensional arrays are stored in row-major order (Equation (6.4)).
- `p->m` is `(*p).m`. A chain of structures linked by `next` pointers keeps its order in the pointers, not in memory: the idea behind every linked structure in this book.
- `new` and `new[]` create objects on the heap that live until `delete` and `delete[]`. A doubling array grows in $O(1)$ amortized time because the total copying is below $2n$ (Equation (6.5)).
- Dangling pointers, leaks, double frees, mismatched deletes, null dereferences and out-of-bounds accesses are all disagreements between a pointer's life and its object's life. They are among the most dangerous weaknesses in real software. AddressSanitizer (`-fsanitize=address`) and Valgrind find them in running programs and report the exact line.
- `nullptr` is the typed null pointer. A reference is an alias that must be initialized, cannot be null, and cannot be re-seated.
- Pass by value copies (and cannot change the caller); pass by pointer copies an address (and can, via `*`); pass by reference aliases (and can, directly). The swap story proves it, and Equation (6.6) explains why large read-only arguments are `const T&`.
- `const` placed right-to-left says what may not change; const correctness turns accidental modifications into compile errors.
- Return values, pointers into caller-owned storage, or smart pointers; never the address of a local.
- `std::unique_ptr` is the default owner of a heap object; `std::shared_ptr` counts shared owners; `std::weak_ptr` observes and breaks cycles. Raw pointers remain fine as non-owning observers.
- Java passes everything by value, and an object's value is a reference; Python binds parameter names to the caller's objects. In both, a function can **mutate** a shared mutable object but cannot **rebind** the caller's variable. Garbage collection prevents dangling pointers and double frees but not leaks.

6.19 Exercises

6.1 Draw the box-and-arrow picture after each statement, and give the values of `a`, `b`, `*p` and `*q` at the end:

```
int a = 5;  int* p = &a;  int* q = p;  *q = 10;
int b = *p;  p = &b;  *p = 20;  *q = *p + 1;
```

6.2 A `double d[6]` starts at address 2000 and `sizeof(double)` is 8. Give the address of `d + 4`, the value of `&d[5] - &d[1]`, and the address that `d + 6` denotes. Which of these three may be dereferenced?

6.3 An `int g[4][6]` starts at address 5000. Using Equation (6.4), find the address of `g[3][2]` and of `g[1][5]`. Which element lives at address 5048?

6.4 For each line, say whether it compiles, and if not, why:

```
int x = 1, y = 2;
const int* a = &x;   *a = 5;   a = &y;
int* const b = &x;   *b = 5;   b = &y;
const int* const c = &x;   int& r = *c;
```

6.5 Predict the output, then check it by running the program:

```
void f(int a, int& b, int* c) { a = a + 1; b = b + 1; *c = *c + 1; c = &a; *c = 100; }
int main() { int x = 1, y = 1, z = 1; f(x, y, &z); std::cout << x << y << z; }
```

6.6 The function below contains three distinct memory bugs. Name each one, say which line causes it, and give the heading of the report AddressSanitizer would print for each (for example `heap-use-after-free`); ASan stops at the first error, so in practice you meet them one at a time as you fix them. Then rewrite the function with `std::vector` so that no bug is possible.

```
int* scores(int n) {
    int* s = new int[n];
    for (int i = 1; i <= n; ++i) s[i] = i * 10;
    int* copy = s;
    delete s;
    return copy;
}
```

6.7 Explain why `for (auto x : v) x *= 2;` leaves a `std::vector<int> v` unchanged, and give two different ways to double every element.

6.8 What does the compiler say about `int& r;`? What does `int& r = a; r = b;` do? Why does C++ forbid a reference from being null or re-seated, and which data structure tasks therefore require pointers instead?

6.9 A growable array starts empty. Count the total number of element copies performed while pushing $n = 1000$ values (a) with the doubling rule of **Pseudocode 6.1**, and (b) with a rule that grows the capacity by 10 each time it is full. Generalize (b) to show that the total is $\Theta(n^2)$.

6.10 Write `void pushFront(Node*& head, const std::string& title)` that inserts a new node at the front of a chain, and `void freeAll(Node*& head)` that deletes every node. Explain why `head` must be passed as `Node*&` and what would go wrong with plain `Node*`. How do Java and Python solve the same problem?

6.11 Predict what this Java program prints, and explain each line using **Figure 6.10**:

```
static void work(List<String> list) { list.add("B"); list = new ArrayList<>();
list.add("C"); }
public static void main(String[] args) { List<String> x = new ArrayList<>(List.of("A"));
work(x); System.out.println(x); }
```

6.12 Write functions that compute a factorial, compute x^y for an integer exponent y that may be negative, and decide whether a number is prime; then a function that prints every prime in a range by calling the primality function. Finally, write a function that returns both the minimum and the maximum of an array, once through pointer parameters and once through reference parameters.

6.13 Predict the output of three successive calls `log_scan("a")`, `log_scan("b")`, `log_scan("c", [])` for `def log_scan(url, seen=[]): seen.append(url); return seen`, and fix the function.

6.14 Give the value of `use_count()` after each line, and say on which line the `Book` is destroyed:

```
auto a = std::make_shared<Book>("B4001");
auto b = a;
std::shared_ptr<Book> c;
c = b;
b.reset();
a = nullptr;
std::weak_ptr<Book> w = c;
c.reset();
```

6.15 Using pointers only (no square brackets), write `reverse(int* first, int* last)` for a half-open range, `length(const char* s)` computed by pointer subtraction, and `int* largest(int* first, int* last)` that returns `nullptr` for an empty range.

6.16 A program keeps `std::vector<Book> catalog;` and, for fast lookup by author, `std::vector<Book*> byAuthor;` holding pointers into `catalog`. New books are appended to `catalog` over time. Explain precisely what goes wrong, which ASan report it would produce, and propose three different fixes, stating the cost of each.

6.20 Solutions to the Exercises

6.1 After `int a = 5; int* p = &a; int* q = p;` both pointers reach `a`. `*q = 10` makes $a = 10$. `int b = *p` creates `b` $= 10$, a separate box. `p = &b` moves `p`'s arrow to `b`; `q` still points to `a`. `*p = 20` makes $b = 20$. `*q = *p + 1` reads 20 through `p` and writes 21 through `q` into `a`. Final values: $a = 21, b = 20, *p = 20, *q = 21$.

6.2 By Equation (6.1), `d + 4` is at $2000 + 4 \cdot 8 = 2032$. By Equation (6.2), `&d[5] - &d[1]` is $(2040 - 2008)/8 = 4$ elements. `d + 6` denotes 2048, one past the end. `d + 4` may be dereferenced (it is `d[4]`); the difference is a number, not a pointer; `d + 6` may be computed and compared but never dereferenced.

6.3 With $C = 6$ and `sizeof(int)` $= 4$: `g[3][2]` is at $5000 + (3 \cdot 6 + 2) \cdot 4 = 5080$ and `g[1][5]` at $5000 + (1 \cdot 6 + 5) \cdot 4 = 5044$. For address 5048, $(5048 - 5000)/4 = 12 = 2 \cdot 6 + 0$, so it is `g[2][0]`, the first element of row 2, right after `g[1][5]`.

6.4 `const int* a = &x;` compiles; `*a = 5` does not (the pointed-to `int` is read-only through `a`); `a = &y` compiles. `int* const b = &x;` compiles; `*b = 5` compiles; `b = &y` does not (`b` itself is const). `const int* const c = &x;` compiles; `int& r = *c;` does not, because it would create a modifiable name for an object promised read-only; `const int& r = *c;` would be fine.

6.5 `a` is a copy of `x`, so `x` stays 1. `b` is `y`, so `y` becomes 2. `c` holds `&z`, so `*c = *c + 1` makes `z` $= 2$. Then `c = &a` re-points the *local* pointer to the local copy `a`, and `*c = 100` changes only that copy. The program prints `122`.

6.6 Bug 1: the loop runs `i` from 1 to n, so `s[n]` is written, one past the end: `heap-buffer-overflow` (a WRITE of size 4, 0 bytes after the region). Bug 2: memory from `new[]` is freed with `delete`: `alloc-dealloc-mismatch (operator new [] vs operator delete)`. Bug 3: the function returns `copy`, which points into the block just freed, so the caller's first use is a `heap-use-after-free`. A safe version returns a vector by value, which owns and frees its memory and cannot be used after it is gone:

```
std::vector<int> scores(int n) {
    std::vector<int> s(n);
    for (int i = 0; i < n; ++i) s[i] = (i + 1) * 10;
    return s;
}
```

6.7 `auto x` makes `x` a copy of each element, and doubling a copy changes nothing in `v`. Either loop by reference, `for (auto& x : v) x *= 2;`, or by index, `for (std::size_t i = 0; i < v.size(); ++i) v[i] *= 2;`.

6.8 `int& r;` is an error: GCC reports that `'r' declared as reference but not initialized`. `int& r = a; r = b;` assigns the value of `b` to `a`. A reference is designed to be an unconditional alias, which is what lets the compiler and the reader trust it without checks. Structures whose links must sometimes point to nothing (the last node of a list, the missing child of a tree node) or must be re-pointed during insertion and deletion therefore use pointers.

6.9 (a) Growth at sizes 1,2,4, ... ,512 copies $1 + 2 + \cdots + 512 = 1023$ elements, less than $2n = 2000$, as Equation (6.5) promises. (b) With capacity 10,20,30, ..., growth happens at sizes

10,20, … ,990 and copies $10 + 20 + \cdots + 990 = 10 \cdot (1 + 2 + \cdots + 99) = 10 \cdot 4950 = 49500$ elements. In general, growing by a constant c performs about n/c growths copying $c, 2c, 3c, \ldots$ elements, a total of $c \cdot \frac{(n/c)(n/c+1)}{2} \approx \frac{n^2}{2c} = \Theta(n^2)$.

6.10 The functions are in **Listing 6.25** below. `pushFront` must change the *caller's* `head` variable so that it points to the new node; with `Node* head` the function would change only its own copy of the pointer, and the caller's list would never see the new node (and the node would leak). `freeAll` takes `Node*&` so that it can leave the caller's `head` as `nullptr` instead of dangling. Java and Python cannot rebind the caller's variable, so their versions *return* the new head and the caller writes `head = pushFront(head, t)`; the twins in the chapter's code folder do exactly that.

6.11 It prints `[A, B]`. `list.add("B")` follows the copied reference and mutates the caller's list. `list = new ArrayList<>()` rebinds the method's copy only; `"C"` goes into a new list that is discarded when the method returns.

6.12 **Listing 6.24** gives all of them. The factorial uses `long long`, which holds $20! \approx 2.4 \times 10^{18}$ but not 21!. The power function multiplies $|y|$ times and inverts for negative y. The primality test stops at $d^2 \le n$, because a composite n always has a divisor no larger than $\sqrt{n}$. The two `minMax` versions show the pointer style (caller writes `&lo, &hi`, the function writes `*lo`) and the reference style (no symbols at the call site).

Listing 6.24 `19_sol_functions.cpp`: solution to Exercise 6.12.

```cpp
// Solutions: factorial, power, primes, and two ways to return two results.
#include <iostream>

long long factorial(int n) {
    long long result = 1;
    for (int k = 2; k <= n; ++k) result *= k;
    return result;
}

double power(double x, int y) {           // x to the power y, y may be negative
    double result = 1.0;
    int steps = (y < 0) ? -y : y;
    for (int k = 0; k < steps; ++k) result *= x;
    return (y < 0) ? 1.0 / result : result;
}

bool isPrime(int n) {
    if (n < 2) return false;
    for (int d = 2; d * d <= n; ++d) {
        if (n % d == 0) return false;
    }
    return true;
}

void printPrimes(int low, int high) {     // reuses isPrime
    for (int n = low; n <= high; ++n) {
        if (isPrime(n)) std::cout << n << ' ';
    }
    std::cout << '\n';
}

void minMaxByPointer(const int* a, int n, int* lo, int* hi) {
    *lo = *hi = a[0];
    for (const int* p = a + 1; p != a + n; ++p) {
        if (*p < *lo) *lo = *p;
        if (*p > *hi) *hi = *p;
    }
}
```

Listing 6.24 (Continued) *19_sol_functions.cpp: solution to Exercise 6.12.*

```cpp
void minMaxByReference(const int* a, int n, int& lo, int& hi) {
    lo = hi = a[0];
    for (int i = 1; i < n; ++i) {
        if (a[i] < lo) lo = a[i];
        if (a[i] > hi) hi = a[i];
    }
}

int main() {
    std::cout << "10! = " << factorial(10) << ", 20! = " << factorial(20) << '\n';
    std::cout << "2^10 = " << power(2, 10) << ", 2^-3 = " << power(2, -3) << '\n';
    std::cout << "primes in [1, 50]: ";
    printPrimes(1, 50);

    int years[5] = {2005, 2006, 1980, 2009, 2002};
    int lo = 0, hi = 0;
    minMaxByPointer(years, 5, &lo, &hi);
    std::cout << "by pointer:   oldest " << lo << ", newest " << hi << '\n';
    minMaxByReference(years, 5, lo, hi);
    std::cout << "by reference: oldest " << lo << ", newest " << hi << '\n';
    return 0;
}
```

```
10! = 3628800, 20! = 2432902008176640000
2^10 = 1024, 2^-3 = 0.125
primes in [1, 50]: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
by pointer:   oldest 1980, newest 2009
by reference: oldest 1980, newest 2009
```

6.13 The calls return `['a']`, then `['a', 'b']` (the default list remembered `'a'`), then `['c']` (an explicit new list was passed). Fix: `def log_scan(url, seen=None):` with `if seen is None: seen = []` as the first line.

6.14 After line 1: 1. Line 2: 2. Line 3: `c` is empty, still 2. Line 4: 3. Line 5: 2. Line 6: 1 (only `c` owns). Line 7: still 1; a `weak_ptr` is not counted. Line 8: 0, and the `Book` is destroyed on this line; `w.expired()` is now `true`.

6.15 See **Listing 6.25**. `reverse` moves `last` back first, because `last` is one past the end, then swaps and closes in until the two pointers meet or cross. `length` walks to the terminator and returns the distance $p - s$ by Equation (6.2). `largest` returns a pointer, so the caller can modify the element found, which the program does by setting it to 0.

Listing 6.25 `20_sol_pointers.cpp`: solutions to Exercises 6.10 and 6.15.

```cpp
// Solutions: reverse with two pointers, strlen by subtraction, a pointer
// to the largest element, and inserting at the front of a chain of nodes.
#include <cstddef>
#include <iostream>
#include <string>

void reverse(int* first, int* last) {     // reverses [first, last)
    while (first < last) {
        --last;                           // last now names a real element
        if (first == last) break;
        int t = *first;
        *first = *last;
        *last = t;
        ++first;
    }
}

std::size_t length(const char* s) {
    const char* p = s;
    while (*p != '\0') ++p;               // stop on the terminator
    return static_cast<std::size_t>(p - s);
}

int* largest(int* first, int* last) {     // nullptr for an empty range
    if (first == last) return nullptr;
    int* best = first;
    for (int* p = first + 1; p != last; ++p) {
        if (*p > *best) best = p;
    }
    return best;
}

struct Node {
    std::string title;
    Node* next;
};

// head is passed by reference: the function must change the caller's pointer.
void pushFront(Node*& head, const std::string& title) {
    head = new Node{title, head};         // the new node points at the old head
}
```

Listing 6.25 (Continued) *20_sol_pointers.cpp: solutions to Exercises 6.10 and 6.15.*

```
void freeAll(Node*& head) {                 // one delete for every new
    while (head != nullptr) {
        Node* doomed = head;
        head = head->next;                  // step first, then delete
        delete doomed;
    }
}

int main() {
    int a[6] = {1, 2, 3, 4, 5, 6};
    reverse(a, a + 6);
    std::cout << "reversed:";
    for (int x : a) std::cout << ' ' << x;
    reverse(a + 1, a + 4);                  // only a[1..3]
    std::cout << "\nmiddle three reversed back:";
    for (int x : a) std::cout << ' ' << x;

    std::cout << "\nlength(\"Pustaka Ceria\") = " << length("Pustaka Ceria") << '\n';

    int* big = largest(a, a + 6);
    *big = 0;                               // modify the caller's element
    std::cout << "largest set to 0:";
    for (int x : a) std::cout << ' ' << x;
    std::cout << "\nlargest of an empty range is null: " << std::boolalpha
              << (largest(a, a) == nullptr) << '\n';

    Node* head = nullptr;                   // an empty chain
    pushFront(head, "Cantik Itu Luka");
    pushFront(head, "Laskar Pelangi");
    pushFront(head, "Bumi Manusia");
    std::cout << "chain:";
    for (Node* cur = head; cur != nullptr; cur = cur->next) std::cout << " [" << cur->title
<< ']';
    freeAll(head);
    std::cout << "\nafter freeAll, head is null: " << (head == nullptr) << '\n';
    return 0;
}
```

```
reversed: 6 5 4 3 2 1
middle three reversed back: 6 3 4 5 2 1
length("Pustaka Ceria") = 13
largest set to 0: 0 3 4 5 2 1
largest of an empty range is null: true
chain: [Bumi Manusia] [Laskar Pelangi] [Cantik Itu Luka]
after freeAll, head is null: true
```

6.16 When `catalog` grows beyond its capacity, `push_back` allocates a new block, moves every `Book` into it and frees the old block (**Figure 6.6**). Every pointer in `byAuthor` then points into freed memory; the next lookup is a use after free, reported by ASan as `heap-use-after-free` with the vector's reallocation in the "freed by" stack, exactly the `stale` case of **Listing 6.9**. Fixes: (1) store *indices* into `catalog` instead of pointers; an index stays valid when the vector moves, at the cost of one extra addition per lookup, $\Theta(1)$. (2) Call `catalog.reserve(N)` for a known maximum N so that no reallocation happens; this is cheap but fails silently if N is exceeded. (3) Store the books themselves on the heap, `std::vector<std::unique_ptr<Book>> catalog;`, so that growth moves only the owning pointers and every `Book` stays at its address; `byAuthor` may then hold raw non-owning `Book*` safely, at the cost of one heap allocation per book and one extra indirection per access.

References

[1] ISO/IEC 14882:2024, *Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2024.

[2] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 16 (amortized analysis and dynamic tables).

[4] MITRE, "2025 CWE Top 25 Most Dangerous Software Weaknesses," Dec. 15, 2025. [Online]. Available: https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html. [Accessed: 24-Sep-2026].

[5] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. 2012 USENIX Annual Technical Conference*, Boston, MA, USA, 2012, pp. 309–318.

[6] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. 28th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, San Diego, CA, USA, 2007, pp. 89–100, doi: 10.1145/1250734.1250746.

[7] Valgrind Developers, "Valgrind: Current releases (3.27.1, 20 May 2026)." [Online]. Available: https://valgrind.org/downloads/current.html. [Accessed: 24-Sep-2026].

[8] Microsoft, "AddressSanitizer," *Microsoft C++, C, and Assembler documentation*. [Online]. Available: https://learn.microsoft.com/en-us/cpp/sanitizers/asan. [Accessed: 24-Sep-2026].

[9] H. Sutter, "C++26 is done! — Trip report: March 2026 ISO C++ standards meeting (London Croydon, UK)," Mar. 29, 2026. [Online]. Available: https://herbsutter.com/2026/03/29/c26-is-done-trip-report-march-2026-iso-c-standards-meeting-london-croydon-uk/. [Accessed: 24-Sep-2026].

[10] T. Köppe, "Erroneous behaviour for uninitialized reads," ISO/IEC JTC1/SC22/WG21 paper P2795R5, 2024. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2795r5.html. [Accessed: 24-Sep-2026].

[11] GNU Project, "GCC 16 Release Series" (GCC 16.1 released April 30, 2026). [Online]. Available: https://gcc.gnu.org/gcc-16/. [Accessed: 24-Sep-2026].

[12] B. Stroustrup and H. Sutter, Eds., *C++ Core Guidelines*, sections F.15–F.21 (parameter passing) and R.20–R.24 (smart pointers). [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines. [Accessed: 24-Sep-2026].

[13] J. Gosling *et al.*, *The Java Language Specification, Java SE 25 Edition*, Oracle, 2025, §4.3 (reference types and values) and §15.12.4 (run-time evaluation of method invocation). [Online]. Available: https://docs.oracle.com/javase/specs/jls/se25/html/. [Accessed: 24-Sep-2026].

[14] Python Software Foundation, "Execution model: naming and binding," *The Python Language Reference*, and "How do I write a function with output parameters (call by reference)?," *Programming FAQ*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/reference/executionmodel.html. [Accessed: 24-Sep-2026].

[15] B. Liskov, "A history of CLU," in *Proc. 2nd ACM SIGPLAN Conf. History of Programming Languages (HOPL-II)*, Cambridge, MA, USA, 1993, pp. 133–147, doi: 10.1145/154766.155367.

[16] G. Lindenmaier and R. Schmelter, "JEP 358: Helpful NullPointerExceptions," OpenJDK, 2019 (delivered in JDK 14, enabled by default since JDK 15). [Online]. Available: https://openjdk.org/jeps/358. [Accessed: 24-Sep-2026].

[17] Python Software Foundation, "gc — Garbage Collector interface," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/gc.html. [Accessed: 24-Sep-2026].

[18] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Chapter 7
Singly Linked Lists

By the end of this chapter you will be able to explain what a linked list is and where the idea came from; say precisely when a linked list beats an array and when it loses, with the costs written as formulas and checked by a stopwatch; declare a node in C++, Java and Python and read and draw the pointer diagrams that describe it; implement a singly linked list reached through a single `head` pointer with every everyday operation (initialize, test for emptiness, insert at the front, at the back, after a given node and in sorted position, delete at the front, at the back and by key, search, traverse, measure the length, reverse it both iteratively and recursively, and destroy it); build the same list again with a dummy head node and see the special cases disappear; add a `tail` pointer and understand exactly which operation it speeds up and which it cannot; close the list into a circle and use it to serve readers in turn; build a stack and a queue out of nodes; prove that your programs neither leak memory nor touch memory they have already given back, using Valgrind and AddressSanitizer; and choose between the linked lists that C++, Java and Python already provide. Every operation is drawn before and after, every cost is derived, and every program is complete, compiled, run and memory-checked.

7.1 Where We Left Off

Chapter 5 built a stack and a queue on top of an array. They worked, and they were fast, but they carried a quiet assumption inherited from the array underneath: a maximum size, decided before the first element arrived. When the Pustaka Ceria library used the array queue for hold requests, somebody had to guess how many requests could ever be waiting at once. Guess too low and the queue reports “full” while the computer has gigabytes of memory to spare; guess too high and the program reserves space it will never use.

Chapter 6 then gave us the tool that removes the guess: the pointer. We learned that a variable can hold an *address*, that `new` asks the system for a fresh piece of memory at run time and hands back its address, that `delete` gives it back, that `->` reaches a field through a pointer, and that a function can change the caller’s pointer when the pointer is passed by reference. Chapter 6 used these tools mostly one at a time. This chapter puts them together into the first data structure that is built entirely out of pointers: the *linked list*.

The idea is simple enough to state in one sentence. Instead of keeping the elements side by side in one block of memory, we keep each element in its own small piece of memory, called a *node*, and every node remembers where the next node is. The whole list is then reached from a single pointer to its first node. Nothing has to be reserved in advance; each new element asks for exactly one node’s worth of memory at the moment it arrives, and each departing element gives exactly that much back.

That simplicity has a price, and much of this chapter is about the price. A list cannot jump to its fifth element the way an array can; it has to walk there. Each node spends extra bytes on its pointer. And because we now manage memory by hand, a single misplaced line can lose a whole chain of nodes or read memory that no longer belongs to us. We will meet every one of these problems deliberately, draw it, measure it and fix it.

Chapter 8 works problems of exactly this kind, mixing the stacks and queues of Chapter 5, the pointers of Chapter 6 and the linked lists of this chapter. Chapter 9 continues the story with doubly linked lists, in which every node also remembers the node before it.

7.2 A Chain Instead of a Row: History and Idea

7.2.1 A treasure hunt

Think of a treasure hunt organized for children at a school festival. The organizers do not line up all the prizes on one long table. Instead, the first clue is handed to the children at the gate: “Look under the big mango tree.” Under the mango tree lies a small prize and a second clue: “Now look behind the library door.” Behind the door lies another prize and a third clue, and so on, until the last clue says “The hunt is over.”

Three features of the hunt matter for us.

- **Each stop holds two things:** something valuable (the prize) and directions to the next stop (the clue).
- **The stops can be anywhere.** They need not be next to each other, in order, or evenly spaced. The mango tree may be at the far end of the field and the library door right beside the gate.
- **You can only follow the chain from the start.** A child who wants the fifth prize must visit the first four stops, because only the fourth clue says where the fifth stop is.

Adding a new stop is easy: write a clue for it, and change one existing clue to point to it. Removing a stop is just as easy: change the clue that pointed at it so that it points at the stop after it. Nobody has to move any prize.

A linked list is exactly this treasure hunt, played inside the computer’s memory. The prize is the data, the clue is a pointer, the stops are nodes scattered wherever the memory allocator placed them, and the card handed out at the gate is a pointer variable, conventionally called `head`.

7.2.2 Where the idea came from

The linked list is one of the oldest data structures in computing. It was developed in 1955 and 1956 by Allen Newell, Cliff Shaw and Herbert A. Simon at the RAND Corporation, as the central data structure of their Information Processing Language (IPL). They needed it because their programs, the Logic Theory Machine, which proved theorems of symbolic logic, the General Problem Solver, and a chess-playing program, manipulated symbolic structures whose size and shape could not be known in advance. The box-and-arrow drawing that every textbook still uses, including this one, appeared in Newell and Shaw’s 1957 paper “Programming the Logic Theory Machine” [1], and IPL itself was later documented for a wide audience [2]. Newell and Simon received the ACM Turing Award in 1975, in part for this work on list processing.

At about the same time, Victor Yngve at the Massachusetts Institute of Technology used linked lists in COMIT, a programming language he designed for linguistic research and machine translation [3]. In 1958 John McCarthy, also at MIT, began LISP, whose very name means “list processor”; its 1960 description [4] made the linked list, built from two-field cells, the basic material of an entire programming language. By the early 1960s linked lists were standard equipment, and Knuth’s *The Art of Computer Programming* later gave them the careful treatment that still defines the field [5].

The problem those pioneers faced is the one Section 7.1 described: collections that grow and shrink unpredictably while the program runs. Seventy years later the answer has not changed: a node that holds some data and the address of the next node.

7.2.3 The vocabulary

We now make the treasure hunt precise.

> **DEFINITION.** A *linked list* is a sequence of *nodes* in which every node stores one element (its *data*) and a pointer, called its *link* or `next` field, to the following node. The list is reached through an external pointer to its first node, called `head`. The last node's `next` field holds the null pointer (`nullptr` in C++, `null` in Java, `None` in Python), which marks the end of the list. A list with no nodes at all is *empty*, and its `head` is null.

Figure 7.1 shows a list of four nodes holding 13, 3, 86 and 74. Each node is drawn as a box with two compartments. The left compartment is the data field. The right compartment is the `next` field; the dot inside it is the start of an arrow, and the arrow ends at the node whose address the field holds. The last node's arrow ends at the word NULL. The pointer variable `head` sits outside the chain and points at the first node.

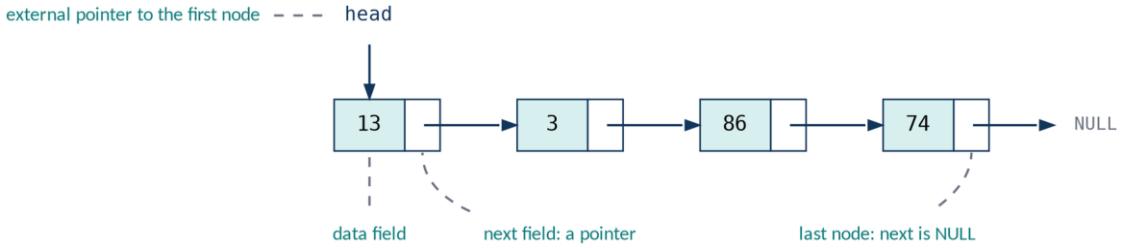


Figure 7.1 A singly linked list holding 13, 3, 86 and 74. Every node has a data field and a `next` field; `head` points at the first node, and the last node's `next` is NULL.

Two adjectives describe the list in the figure, and they give this chapter its full name.

- **Singly linked.** Each node has *one* link, pointing forward. From a node you can reach the node after it in one step, but there is no way back to the node before it. Chapter 9 adds a second link per node, pointing backward, which gives a *doubly linked* list.
- **Non-circular.** The chain ends: the last node points at NULL. That NULL is not a nuisance; it is the stop signal every loop in this chapter relies on. In Section 7.9 we will connect the last node back to the first, which gives a *circular* list, and every loop will need a different stop signal.

The terms *singly linked list non-circular* (SLLNC) and *chained list* are also common for the structure in **Figure 7.1**. In this book we say simply *singly linked list* and mention circularity only when it applies.

> **IN PLAIN LANGUAGE.** A linked list is a chain of boxes. Each box holds a value and an arrow to the next box. You hold an arrow to the first box. To find anything you start at the first box and follow the arrows.

7.3 Array versus Linked List

7.3.1 Two ways to keep five numbers

Before writing any list code, let us compare the two structures side by side in memory, because every advantage and disadvantage of the linked list follows from the picture. **Figure 7.2** shows the numbers 13, 3, 86, 74 and 52 stored twice: in an array of `int` and in a linked list.

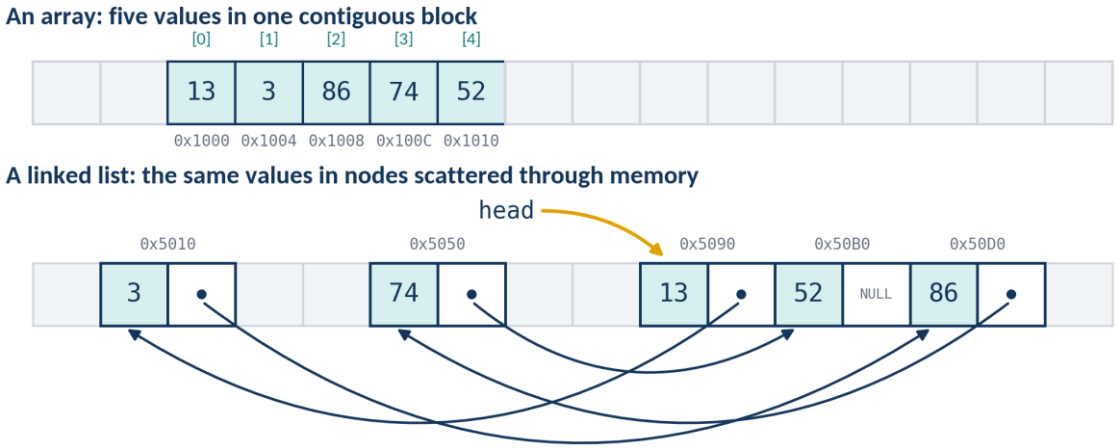


Figure 7.2 The same five values in an array (one contiguous block, consecutive addresses) and in a linked list (nodes wherever the allocator put them, joined by `next` pointers). The addresses are illustrative.

The array occupies one contiguous block. Its elements are 4 bytes apart because an `int` is 4 bytes wide, so if the block starts at address b and each element takes s bytes, element i lives at

$$\text{address}(i) = b + i \cdot s \tag{7.1}$$

Chapter 1 met this formula already; what matters here is its consequence. To read `a[3]` the computer does one multiplication and one addition and goes straight there, no matter whether the array has five elements or five million. That is *random access*, and it costs $O(1)$.

The linked list has no such formula. Its nodes sit at addresses chosen by `new` at the moment each node was created, and those addresses follow no pattern: in **Figure 7.2** the first node (13) happens to live at `0x5090`, the second (3) at `0x5010`, far to its left, the third (86) at `0x50D0`. The only record of where node $k+1$ lives is the `next` field inside node k. To reach the node at index k (counting from 0), we must start at `head` and follow k links:

$$T_{\text{reach}}(k) = k \text{ pointer steps}, \qquad 0 \le k \le n-1 \tag{7.2}$$

If every index is equally likely to be requested, the average number of steps is

$$\frac{1}{n}\sum_{k=0}^{n-1} k = \frac{1}{n}\cdot\frac{(n-1)n}{2} = \frac{n-1}{2} \tag{7.3}$$

which grows in proportion to n. Access by position in a linked list is therefore $\Theta(n)$ on average and in the worst case: *sequential access*. A list of a million nodes needs, on average, about half a million steps to reach an element that an array reaches in one.

Now turn the question around and ask what it costs to *insert* a new value at the front. In the array, every existing element must move one place to the right to make room, because the elements must stay contiguous: inserting at index 0 of an array holding n elements moves n elements. Building an array of n elements by always inserting at the front therefore moves

$$0 + 1 + 2 + \cdots + (n-1) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \tag{7.4}$$

elements in total, which is $\Theta(n^2)$. In the linked list, inserting at the front creates one node and changes two pointers, whatever n is: $O(1)$, and $\Theta(n)$ for all n insertions together. The same

contrast holds for deleting the first element. More generally, once we hold a pointer to a node, inserting after it or deleting the node after it is $O(1)$ in a list, while the corresponding operation in the middle of an array shifts every element that follows.

7.3.2 The comparison, feature by feature

Table 7.1 collects the differences.

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

Two lines of the table deserve comment, because they are often misremembered. First, a linked list is sometimes described as having "unlimited" size. It is more accurate to say that its size is limited only by the memory the program can obtain, and that it never needs to guess its maximum in advance. Second, the claim that an array "cannot be deleted" while a list "is easy to delete" gets things backward in C++: a dynamically allocated array is released with a single `delete[]`, whereas a list must visit and release every node, which takes $\Theta(n)$ time and is a classic source of memory leaks. What *is* easy with a list is removing *individual* elements without disturbing the others.

The last line of the table, cache friendliness, is invisible in the O-notation but very visible on a real machine. Modern processors read memory in blocks called *cache lines*, typically 64 bytes, and keep recently used lines in small, very fast *caches* [6]. Walking through an array uses every byte of every line it loads, and the processor can predict the next address and fetch it early. Walking through a list whose nodes are scattered loads a new line for almost every node, and the processor cannot fetch node $k+1$ until it has read node k's `next` field. Section 7.12 measures the effect; the difference can be a factor of a hundred.

7.3.3 The memory each element costs

A node stores its data *and* a pointer. On the 64-bit machines this book assumes, a pointer takes $p = 8$ bytes. The memory allocator also rounds every request up and keeps a little bookkeeping of its own; call that extra a bytes per node. If one element's data takes d bytes, then n elements take

$$M_{\text{array}} = n\,d, \qquad M_{\text{list}} = n\,(d + p + a) \tag{7.5}$$

bytes. For a list of `int` values, $d = 4$, and alignment rules pad the node so that the pointer starts at a multiple of 8, so a node occupies 16 bytes before the allocator adds anything; on the GNU C library the allocator then rounds each 16-byte request up to a 32-byte chunk. An `int` in a list

therefore costs about 8 times its size, while an array of `int` costs exactly its size. When the data is large, say a 200-byte book record, the 8-byte pointer becomes a small tax. The rule of thumb that follows from Equation (7.5) is: linked lists of tiny elements are expensive in memory; linked lists of large records are not.

> **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.

7.4 Nodes and Pointers in C++

7.4.1 Declaring a node

A node is a record with two fields, so in C++ it is a `struct`. The traditional declaration, inherited from C, gives the structure a tag name and then a shorter alias with `typedef`:

```
typedef struct MyNode {
    int data;
    struct MyNode *next;
} TNode;
```

Read it from the inside out. The structure has an integer field `data` and a field `next` whose type is "pointer to `struct MyNode`", that is, a pointer to a node of this very kind. A structure that contains a pointer to its own type is called *self-referential*, and it is the whole trick of the linked list: a node cannot *contain* another node (that node would contain a third, and so on forever, and the structure would have infinite size), but it can contain the *address* of another node, which is always 8 bytes. The `typedef` then lets us write `TNode` instead of `struct MyNode` everywhere else.

In C++ the `typedef` is not needed, because a `struct` name is already a type name, and the pointer can use the name while the structure is still being declared:

```
struct Node {
    int data;
    Node *next;
};
```

Both forms describe exactly the same memory layout. This chapter uses `TNode` in the programs that follow the classic presentation of Section 7.5 and Section 7.8, and the shorter `Node` elsewhere. Once a node type exists we need a variable that will point at the first node:

```
TNode *head;
```

`head` is not a node. It is a pointer, 8 bytes that will hold the address of the first node, or `nullptr` while the list is empty. This distinction, between the node and a pointer to it, is the single most important thing to keep straight in this chapter.

7.4.2 Making a node with `new`

A node is created at run time with `new`, which asks the system for enough memory for one `TNode` and returns its address. The fields are then filled through the pointer with `->`:

```
int newData;
// ...
TNode *newNode;
newNode = new TNode;
newNode->data = newData;
newNode->next = nullptr;
```

After these lines, `newNode` points at a node on the *heap* (the region of memory that `new` manages) whose data is `newData` and whose `next` is `nullptr`: a list of length one, not yet attached to

anything. C++ uses the keyword `nullptr` for the null pointer; older C and C++ code writes `NULL` or `0`, which work but can be confused with the integer zero, so modern code uses `nullptr`.

Since C++11 the four lines can be written as one, with *aggregate initialization*, which fills the fields in declaration order:

```
TNode *newNode = new TNode{newData, nullptr};
```

Every `new` must eventually be matched by exactly one `delete` of the same address, or the memory is lost until the program ends. We will be careful about this from the very first program.

7.4.3 Three nodes by hand

Listing 7.1 builds a three-node list without any list functions at all, so that nothing hides what is happening. It creates three separate nodes, links them by writing into their `next` fields, walks the chain with a helper pointer, reads data through chains of arrows, and finally releases every node.

Listing 7.1 `node_basics.cpp`: three nodes created with `new`, linked by hand, walked, and released with `delete`.

```
// node_basics.cpp -- three nodes linked by hand, walked, and released.
#include <iostream>

struct Node {
    int data;     // the value this node carries
    Node *next;   // address of the following node, or nullptr at the end
};

int main() {
    // Make three separate nodes on the heap. They are not linked yet.
    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";

    std::cout << "head->data             = " << head->data << '\n';
    std::cout << "head->next->data       = " << head->next->data << '\n';
    std::cout << "head->next->next->data = " << head->next->next->data << '\n';

    // Release every node: remember the next one BEFORE deleting the current one.
    int released = 0;
    while (head != nullptr) {
        Node *del = head;
        head = head->next;
        delete del;
        ++released;
    }
    std::cout << "Nodes released: " << released << '\n';
    return 0;
}
```

```
Three nodes linked by hand: 13 -> 3 -> 86 -> NULL
head->data             = 13
head->next->data       = 3
head->next->next->data = 86
Nodes released: 3
```

Three details deserve a slow reading.

- The loop `for (Node *aux = head; aux != nullptr; aux = aux->next)` is the heartbeat of every linked-list program. The helper pointer `aux` (short for *auxiliary*) starts at the first node, does its work, and then moves to the next node by copying that node's `next` field into itself. When it copies the last node's `next`, it becomes `nullptr` and the loop stops.
- `head->next->next->data` is read from left to right, one arrow at a time. `head` holds the address of the node holding 13. `head->next` is that node's `next` field, which holds the address of the node holding 3. `head->next->next` is the `next` field of *that* node, which holds the address of the node holding 86, and `->data` reads its data. Each `->` is one hop along the chain.
- The release loop saves the node about to be deleted in `del`, moves `head` forward, and only then deletes `del`. Deleting first and moving afterward would read `head->next` from memory that no longer belongs to the program. Section 7.11 shows what the memory checkers say when that happens.

7.4.4 The other way: `malloc` and `free`

C programs, and C++ programs that talk to C libraries, allocate nodes with the function `malloc` from `<cstdlib>` (`<stdlib.h>` in C), whose declaration is essentially `void *malloc(size_t size)`. It returns the address of `size` uninitialized bytes, or a null pointer if memory has run out; the caller converts the address to the right pointer type and fills in the fields. Memory from `malloc` is returned with `free`. **Listing 7.2** shows both styles in one program, using a small student record in the same shape as our nodes.

Listing 7.2 `malloc_node.cpp`: a node allocated with `malloc` and released with `free`, next to one made with `new` and released with `delete`.

```
// malloc_node.cpp -- the same node made two ways: new/delete and malloc/free.
#include <cstdio>
#include <cstdlib>   // std::malloc, std::free

typedef struct MyStruct {
    int id;
    struct MyStruct *next;
} Student;

void init(Student **p) {        // p is the ADDRESS of the caller's pointer
    *p = nullptr;
}

Student *allocate(int id) {
    Student *p = (Student *) std::malloc(sizeof(Student));
    if (p != nullptr) {         // malloc returns nullptr when memory runs out
        p->next = nullptr;
        p->id = id;
    }
    return p;
}

void add(Student **p, int id) {
    *p = allocate(id);          // changes the caller's pointer through its address
    if (*p != nullptr) std::printf("Student ID: %d\n", (*p)->id);
}

int main() {
    Student *head;
    init(&head);
    add(&head, 13);
    std::printf("sizeof(Student) = %zu bytes\n", sizeof(Student));
    std::free(head);            // memory from malloc goes back with free, never delete

    Student *other = new Student{42, nullptr};   // the C++ way: no cast, no sizeof
    std::printf("Student ID: %d (made with new)\n", other->id);
    delete other;               // memory from new goes back with delete, never free
    return 0;
}
```

```
Student ID: 13
sizeof(Student) = 16 bytes
Student ID: 42 (made with new)
```

The function `init` receives `Student **p`, a *pointer to a pointer*: the address of the caller's variable `head`. Writing `*p = nullptr` therefore changes `head` itself, which is how C, which has no references, lets a function modify the caller's pointer (Chapter 6 compared this with passing a pointer by reference). Notice also the output line `sizeof(Student) = 16 bytes`: an `int` (4 bytes) plus a pointer (8 bytes) would be 12, but the compiler inserts 4 bytes of padding so that the pointer starts at an address divisible by 8, exactly as Section 7.3.3 said.

There are three rules to remember. Memory from `malloc` goes back with `free`, memory from `new` goes back with `delete`, and the two must never be mixed. `malloc` does not run constructors, so it is safe only for plain records such as `Student`; a node that contains a `std::string`, as in Section 7.9, must be created with `new`. And `malloc`'s result must be checked for null before use, whereas `new` reports failure by throwing `std::bad_alloc`. In the rest of this book we use `new` and `delete`.

7.4.5 How to read the pointer diagrams in this chapter

Every operation in the rest of the chapter is drawn before and after, in the style of **Figure 7.1**, with a few extra conventions.

- A **teal** node is an ordinary node. A **darker teal** node with a teal border is the node that a moving helper pointer (`aux`, `cur`) is looking at right now.
- An **amber** node is a node that has just been created and is being inserted. A **red** node is being deleted.
- An **amber arrow** is a pointer that the step being shown has just changed. When a step changes two pointers, small amber numbers give the order in which the code changes them. The order is not decoration: with the wrong order the operation breaks, as Section 7.11 shows.
- A name in `monospace` above a node is a pointer variable (`head`, `aux`, `prev`, `newNode`) and its arrow shows which node it points at.

7.5 The Singly Linked List with a Head Pointer

The simplest complete linked list keeps exactly one pointer variable outside the nodes: `head`, which always points at the first node, or is `nullptr` when the list is empty. Every node is reached from `head` and from nowhere else. That gives the first rule of list programming:

> **THE RULE OF THE HEAD.** Never move `head` to walk through the list. `head` is the only record of where the list starts; if a loop advances `head`, every node it passes becomes unreachable and is lost. Walk with a helper pointer (`aux`, `cur`) and move `head` only when the first node itself changes.

In this section we write every operation of the classic head-pointer list: `init`, `isEmpty`, `frontInsertion`, `backInsertion`, `show`, `frontDeletion`, `backDeletion` and `clear`. For each one we first describe the idea, then give the procedure in pseudocode, draw it before and after, and derive its cost. The complete C++ program, with its output, follows in Section 7.5.8, and the same list in Java and Python in Section 7.5.9.

The functions take `head` by reference, `TNode *&head`, so that a function that must change which node is first (inserting or deleting at the front) changes the caller's variable, not a copy of it. Chapter 6 introduced reference parameters for exactly this purpose. A function that only reads the list takes `const TNode *head` by value, which promises that it will not change any node.

7.5.1 `init` and `isEmpty`

An empty list is one whose `head` is null, so initialization is a single assignment and the emptiness test a single comparison:

```
void init(TNode *&head) { head = nullptr; }
bool isEmpty(const TNode *head) { return head == nullptr; }
```

Both run in $O(1)$ time. `init` matters more than it looks: a local pointer variable in C++ starts with whatever bits happened to be in its memory, so a `head` that was never initialized does not mean "empty", it means "somewhere random", and the first loop over it will wander into memory at random.

7.5.2 `frontInsertion`: a new first node

To insert at the front we create the new node, make it point at the current first node, and then make `head` point at the new node, so that the new node becomes the first one. **Pseudocode 7.1** states the procedure; L is the list and $L.head$ its head pointer.

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
```

Figure 7.3 shows the two steps on a list holding 13 and 86 when 3 is inserted. In panel (a) the new node exists but nothing points at it and its `next` is still NULL. Step 1 (panel b) makes it point at 13; at this moment two pointers lead to 13, `head` and the new node's `next`. Step 2 (panel c) moves `head` to the new node, and the list reads 3, 13, 86.

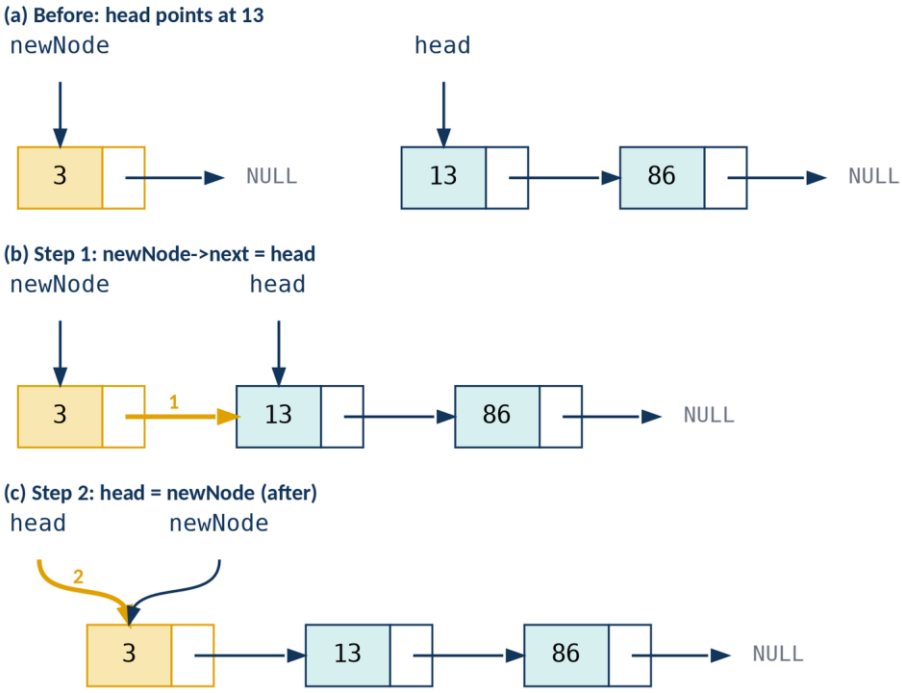


Figure 7.3 `frontInsertion(3)` on the list 13, 86: (a) before, (b) the new node is linked to the old first node, (c) `head` moves to the new node.

The order of the two steps is essential. If line 4 ran before line 3, `head` would already point at the new node when line 3 copies it, the new node would point at *itself*, and the old nodes 13 and 86 would be lost forever. Section 7.11 draws exactly this mistake.

What happens when the list is empty? Then `L.head` is NULL, line 3 sets `z.next` to NULL, and line 4 makes `z` the first and only node. The same two lines are correct for the empty list, so no special case is needed. (Many presentations test `isEmpty()` first and write `head = newNode; head->next = nullptr;` in that branch; the result is identical, and the test is harmless but unnecessary. Exercise 7.10 asks you to check this carefully.)

Cost. Lines 1 to 4 each take constant time and nothing depends on n, so `frontInsertion` is $O(1)$. Inserting n values at the front therefore costs $\Theta(n)$ in total, against the $\Theta(n^2)$ of Equation (7.4) for an array. Note one visible side effect: because each new value goes in front of the previous ones, inserting 13 and then 3 at the front gives the order 3, 13, the reverse of the order of insertion. Front insertion is how a *stack* works, a fact Section 7.10 will use.

7.5.3 `backInsertion`: a new last node

To insert at the back we must find the last node, because it is the node whose `next` must change. The head-pointer list does not know where its last node is, so it walks there: a helper pointer `aux`

starts at `head` and moves forward *while* `aux->next` *is not NULL*. When the loop stops, `aux` is at the last node, and one assignment attaches the new node after it.

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
```

Figure 7.4 traces the insertion of 74 into the list 3, 13, 86. Panel (a) shows `aux` starting at the first node. The loop test looks one node *ahead*: it asks whether the node after `aux` exists. After two moves (panel b) `aux` is at 86, whose `next` is NULL, so the loop stops there, not one step later. Line 10 (panel c) attaches the new node.

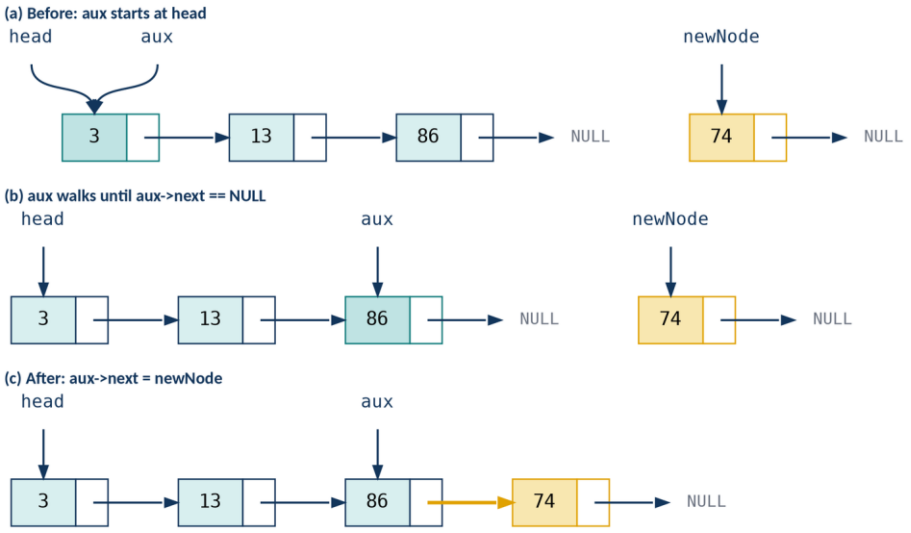


Figure 7.4 `backInsertion(74)` on the list 3, 13, 86: (a) `aux` starts at `head`, (b) `aux` stops at the node whose `next` is NULL, (c) that node's `next` becomes the new node.

The empty list does need its own branch here (lines 4 and 5), because with `head` equal to NULL there is no node for `aux` to start at: evaluating `aux->next` on a null `aux` is undefined behavior, which in practice usually crashes the program.

Why must the loop condition be `aux.next ≠ NIL` rather than `aux ≠ NIL`? With `aux ≠ NIL` the loop would run until `aux` itself became NULL, one step *past* the last node, and line 10 would then try to write into a node that does not exist. Stopping one node early is the price of a singly linked list: to change the link *into* a position, you must stand on the node *before* it.

Cost. The loop runs once for every node except the last, so on a list of n nodes it makes $n-1$ moves: $\Theta(n)$. Building a list of n values by repeated back insertion starting from empty therefore makes $0+1+\cdots+(n-2)$ moves plus a constant amount of work per insertion, which is $\Theta(n^2)$ in total by the same sum as Equation (7.4). Section 7.8 removes this cost with a second pointer.

7.5.4 `show`: walking the list

Traversal visits every node in order, doing something at each one; `show` prints the data. **Pseudocode 7.3** is the pattern that every loop in this chapter follows.

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
```

Figure 7.5 follows `aux` through a four-node list. Here the loop test is `aux ≠ NIL`, not `aux.next ≠ NIL`, because we want to visit the last node too; the loop ends only after `aux` has moved past the last node onto NULL (panel e).

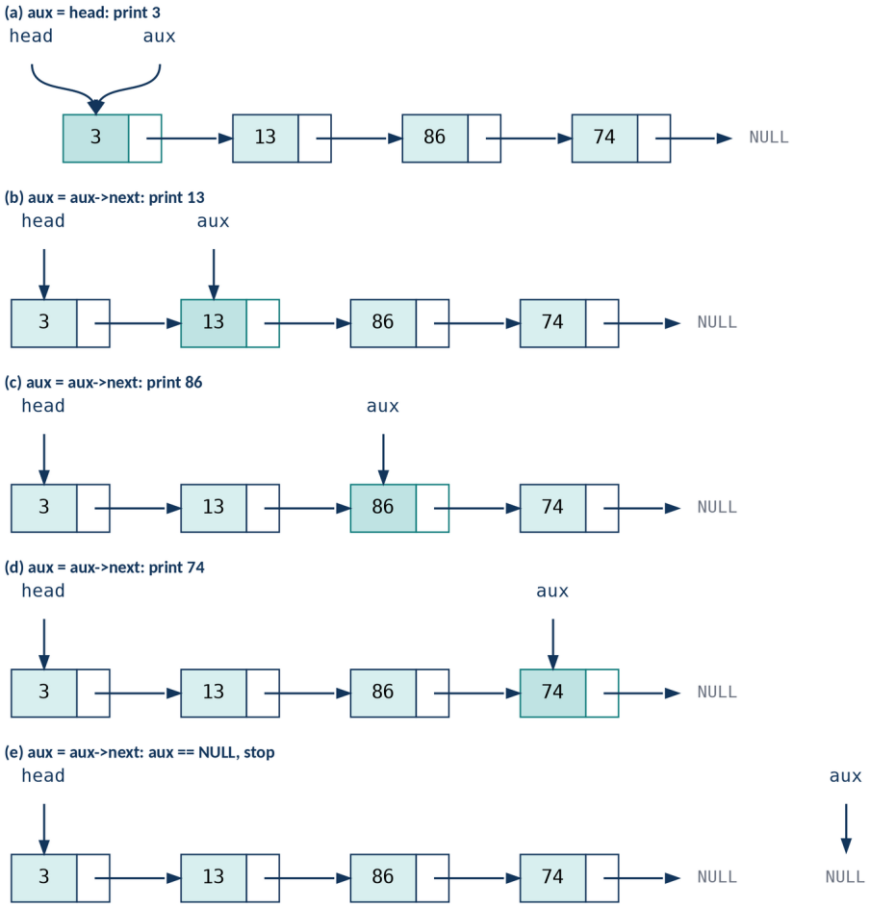


Figure 7.5 Traversing the list 3, 13, 86, 74: `aux` visits each node in turn and the loop ends when `aux` becomes NULL; `head` never moves.

The two loop conditions, `aux ≠ NIL` in **Pseudocode 7.3** and `aux.next ≠ NIL` in **Pseudocode 7.2**, are the two most common loops in list programming, and choosing the wrong one is the most common off-by-one error. A simple way to choose: ask on which node the loop must *end*. If it must end having visited every node, test `aux`; if it must end *standing on* the last node, test `aux.next`.

Cost. Line 3 runs once per node, so `show` is $\Theta(n)$, which is optimal: printing n values cannot take less than n steps.

7.5.5 `frontDeletion`: removing the first node

To delete the first node we must do two things: move `head` to the second node, so that the list still starts somewhere valid, and give the old first node back with `delete`. Once `head` has moved, though, nothing points at the old first node any more, so we must first remember its address in a second helper pointer, `del`.

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
```

Figure 7.6 shows the three steps on the list 3, 13, 86. After step 2 the red node is no longer part of the list, but it still exists and `del` still points at it; step 3 releases it.

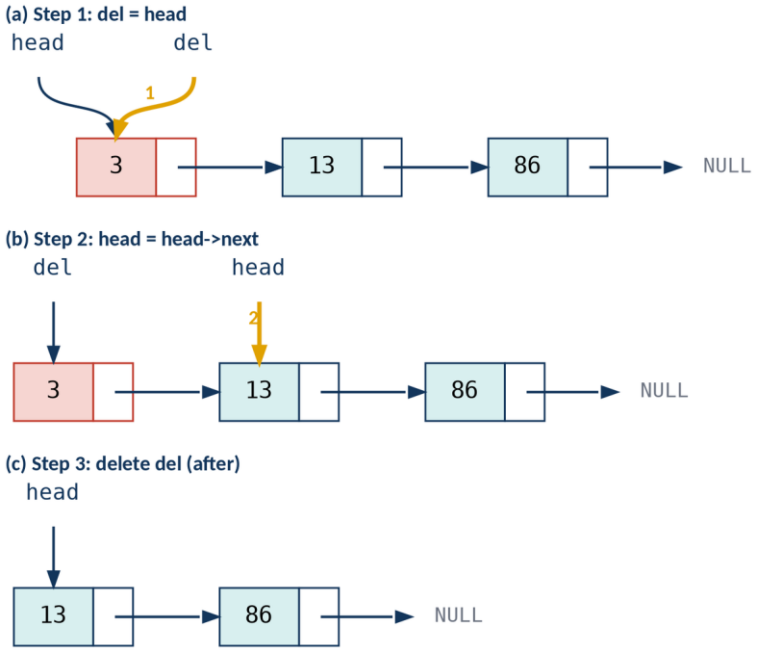


Figure 7.6 `frontDeletion()` on the list 3, 13, 86: (a) `del` remembers the first node, (b) `head` moves to the second node, (c) the old first node is deleted.

Both orders of the remaining work are tempting and both are wrong. Deleting `head` first and then writing `head = head->next` reads the `next` field of a node that has already been given back. Moving `head` first *without* saving the old address makes the old first node unreachable: it can never be deleted, so its memory leaks. The correct order, save, move, delete, is worth memorizing, because every deletion in this chapter follows it.

A list of one node is handled correctly too: `head.next` is NULL, so line 4 makes the list empty. **Cost:** $O(1)$.

7.5.6 `backDeletion`: removing the last node

To delete the last node we must change the `next` field of the node *before* it, which becomes the new last node and must point at NULL. A singly linked node cannot look backward, so once again we walk from the front. This time the helper `aux` must stop on the second-to-last node, which is the node for which `aux.next.next` is NULL.

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
```

Figure 7.7 traces the deletion of 74 from 3, 13, 86, 74. In panel (a) the loop has stopped with `aux` on 86, because the node after 86 (74) has a NULL `next`. Panel (b) cuts the link from 86 and remembers 74 in `del`; panel (c) shows the list after `del` is released.

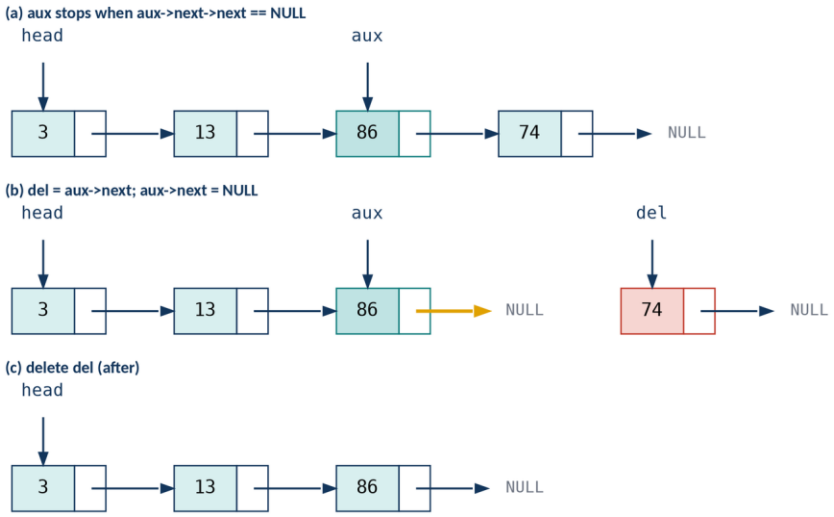


Figure 7.7 `backDeletion()` on the list 3, 13, 86, 74: (a) `aux` stops on the second-to-last node, (b) the last node is detached, (c) it is deleted.

The one-node case (lines 3 to 5) is separate because the loop condition `aux.next.next` would dereference a NULL `aux.next` when there is only one node. In that case the last node is also the first node, so it is `head` that must change.

Line 11 is the line most often forgotten. Without it, 86 keeps pointing at the node that line 12 releases, and the next traversal walks straight into freed memory. Section 7.11 runs exactly this bug under the memory checkers.

Cost. The loop makes $n - 2$ moves on a list of $n \geq 2$ nodes, so `backDeletion` is $\Theta(n)$. Unlike back insertion, this cost will *not* go away when we add a tail pointer in Section 7.8, and understanding why is one of the central lessons of this chapter.

7.5.7 `clear`: releasing every node

`clear` empties the list and returns every node to the system. It is repeated front deletion: save the first node, move `head`, delete the saved node, until `head` is NULL.

```
void clear(TNode *&head) {
    while (head != nullptr) {
        TNode *del = head;
        head = head->next;
        delete del;
    }
}
```

The loop runs once per node, so `clear` is $\Theta(n)$. When it finishes, `head` is `nullptr`, so the list is a valid empty list and can be used again. A program must call `clear` (or the equivalent `destroy` of Section 7.6.8) before its last pointer to the list disappears, or every node leaks. Section 7.10 shows how a C++ class can make this automatic.

7.5.8 The complete program

Listing 7.3 puts the eight operations together. The `main` function inserts 13 and 3 at the front, 86 and 74 at the back, prints the list, deletes one node at each end, prints the list again, and finally clears it.

Listing 7.3 `sll_head.cpp`: the singly linked list with a head pointer, with every operation of Section 7.5.

```
// sll_head.cpp -- a singly linked list reached through one pointer, head.
#include <cstdio>

typedef struct MyNode {
    int data;
    struct MyNode *next;
} TNode;

void init(TNode *&head) {                // head is passed by reference
    head = nullptr;
}

bool isEmpty(const TNode *head) {
    return head == nullptr;
}

void frontInsertion(TNode *&head, int newData) {
    TNode *newNode = new TNode;
    newNode->data = newData;
    newNode->next = head;                // 1. new node points at the old first node
    head = newNode;                      // 2. head points at the new node
    std::printf("%d has been inserted by frontInsertion()\n", newData);
}

void backInsertion(TNode *&head, int newData) {
    TNode *newNode = new TNode;
    newNode->data = newData;
    newNode->next = nullptr;             // it will be the last node
    if (isEmpty(head)) {
        head = newNode;
    } else {
        TNode *aux = head;
        while (aux->next != nullptr) {   // walk to the last node
            aux = aux->next;
        }
        aux->next = newNode;
    }
    std::printf("%d has been inserted by backInsertion()\n", newData);
}
```

Listing 7.3 (Continued) *sll_head.cpp: the singly linked list with a head pointer, with every operation of Section 7.5.*

```
void show(const TNode *head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    const TNode *aux = head;              // head itself never moves
    while (aux != nullptr) {
        std::printf("%d ", aux->data);
        aux = aux->next;
    }
    std::printf("\n");
}

void frontDeletion(TNode *&head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    TNode *del = head;
    int d = del->data;
    head = head->next;                    // move head BEFORE the node disappears
    delete del;
    std::printf("%d has been deleted by frontDeletion()\n", d);
}

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 7.3 (Continued) *sll_head.cpp: the singly linked list with a head pointer, with every operation of Section 7.5.*

```
}

void clear(TNode *&head) {
    while (head != nullptr) {
        TNode *del = head;
        head = head->next;
        delete del;
    }
    std::printf("The linked list has been cleared\n");
}

int main() {
    TNode *head;
    init(head);

    std::printf("Front insertion...\n");
    frontInsertion(head, 13);
    frontInsertion(head, 3);

    std::printf("\nBack insertion...\n");
    backInsertion(head, 86);
    backInsertion(head, 74);

    std::printf("\nThe content of the linked list is in the following.\n");
    show(head);

    std::printf("\nFront deletion...\n");
    frontDeletion(head);
    std::printf("\nBack deletion...\n");
    backDeletion(head);

    std::printf("\nThe content of the linked list is in the following.\n");
    show(head);

    std::printf("\n");
    clear(head);
    show(head);
    return 0;
}
```

```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

The content of the linked list is in the following.
3 13 86 74

Front deletion...
3 has been deleted by frontDeletion()

Back deletion...
74 has been deleted by backDeletion()

The content of the linked list is in the following.
13 86

The linked list has been cleared
The linked list is empty
```

Follow the list through the run. After `frontInsertion(13)` it is 13; after `frontInsertion(3)` it is 3, 13, because front insertion places each value before the previous ones. Back insertion then appends 86 and 74, giving 3, 13, 86, 74. `frontDeletion` removes 3 and `backDeletion` removes 74, leaving 13, 86. Finally `clear` releases the last two nodes and `show` reports an empty list. Section 7.11 runs this program under Valgrind to confirm that all four nodes were released.

Table 7.2 summarizes the costs we derived.

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

7.5.9 The same list in Java and Python

Java and Python have no pointers in the C++ sense, but every variable of a class type holds a *reference*, which behaves like a pointer that can never be used for arithmetic and can never point at freed memory. A node is therefore an ordinary class with a data field and a field of the node's own class. **Listing 7.4** is the Java version of **Listing 7.3**; the node class is nested inside the program, and the list becomes a class `SinglyLinkedList` whose private field `head` plays the role of the C++ reference parameter.

Listing 7.4 `SllHead.java`: the head-reference singly linked list in Java.

```java
// SllHead.java -- a singly linked list reached through one reference, head.
public class SllHead {
    static class Node {
        int data;
        Node next;

        Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    static class SinglyLinkedList {
        private Node head;                    // null means the list is empty

        SinglyLinkedList() {                  // init
            head = null;
        }

        boolean isEmpty() {
            return head == null;
        }

        void frontInsertion(int newData) {
            head = new Node(newData, head);   // new node points at the old first node
            System.out.println(newData + " has been inserted by frontInsertion()");
        }

        void backInsertion(int newData) {
            Node newNode = new Node(newData, null);
            if (isEmpty()) {
                head = newNode;
            } else {
                Node aux = head;
                while (aux.next != null) {    // walk to the last node
                    aux = aux.next;
                }
                aux.next = newNode;
            }
            System.out.println(newData + " has been inserted by backInsertion()");
        }
```

Listing 7.4 (Continued) *SllHead.java: the head-reference singly linked list in Java.*

```
    void show() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        StringBuilder sb = new StringBuilder();
        for (Node aux = head; aux != null; aux = aux.next) {
            sb.append(aux.data).append(' ');
        }
        System.out.println(sb);
    }

    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()");
    }

    void backDeletion() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        int d;
        if (head.next == null) {           // exactly one node
            d = head.data;
            head = null;
        } else {
            Node aux = head;
            while (aux.next.next != null) {   // stop at the second-to-last node
                aux = aux.next;
            }
            d = aux.next.data;
            aux.next = null;               // aux becomes the new last node
        }
        System.out.println(d + " has been deleted by backDeletion()");
    }
```

Listing 7.4 (Continued) *SllHead.java: the head-reference singly linked list in Java.*

```
        void clear() {
            head = null;                        // every node is now unreachable
            System.out.println("The linked list has been cleared");
        }
    }

    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();

        System.out.println("Front insertion...");
        list.frontInsertion(13);
        list.frontInsertion(3);

        System.out.println("\nBack insertion...");
        list.backInsertion(86);
        list.backInsertion(74);

        System.out.println("\nThe content of the linked list is in the following.");
        list.show();

        System.out.println("\nFront deletion...");
        list.frontDeletion();
        System.out.println("\nBack deletion...");
        list.backDeletion();

        System.out.println("\nThe content of the linked list is in the following.");
        list.show();

        System.out.println();
        list.clear();
        list.show();
    }
}
```

Listing 7.5 is the Python version. `__slots__` tells Python that a `Node` has exactly the two attributes `data` and `next`; this saves memory (no per-object dictionary) and turns a misspelled attribute such as `node.nxt = ...` into an error instead of a silently created new attribute.

Listing 7.5 `sll_head.py`: the head-reference singly linked list in Python.

```
# sll_head.py -- a singly linked list reached through one reference, head.
class Node:
    __slots__ = ("data", "next")

    def __init__(self, data, next=None):
        self.data = data
        self.next = next


class SinglyLinkedList:
    def __init__(self):                         # init
        self.head = None                        # None means the list is empty

    def is_empty(self):
        return self.head is None

    def front_insertion(self, new_data):
        self.head = Node(new_data, self.head)  # new node points at the old first node
        print(f"{new_data} has been inserted by frontInsertion()")

    def back_insertion(self, new_data):
        new_node = Node(new_data)
        if self.is_empty():
            self.head = new_node
        else:
            aux = self.head
            while aux.next is not None:         # walk to the last node
                aux = aux.next
            aux.next = new_node
        print(f"{new_data} has been inserted by backInsertion()")

    def show(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        parts = []
        aux = self.head
        while aux is not None:
            parts.append(f"{aux.data} ")
            aux = aux.next
        print("".join(parts))
```

Listing 7.5 (Continued) *sll_head.py: the head-reference singly linked list in 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()")

    def back_deletion(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        if self.head.next is None:              # exactly one node
            d = self.head.data
            self.head = None
        else:
            aux = self.head
            while aux.next.next is not None:    # stop at the second-to-last node
                aux = aux.next
            d = aux.next.data
            aux.next = None                     # aux becomes the new last node
        print(f"{d} has been deleted by backDeletion()")

    def clear(self):
        self.head = None                        # every node is now unreachable
        print("The linked list has been cleared")


def main():
    lst = SinglyLinkedList()

    print("Front insertion...")
    lst.front_insertion(13)
    lst.front_insertion(3)

    print("\nBack insertion...")
    lst.back_insertion(86)
    lst.back_insertion(74)

    print("\nThe content of the linked list is in the following.")
    lst.show()
```

Listing 7.5 (Continued) *sll_head.py: the head-reference singly linked list in Python.*

```
    print("\nFront deletion...")
    lst.front_deletion()
    print("\nBack deletion...")
    lst.back_deletion()

    print("\nThe content of the linked list is in the following.")
    lst.show()

    print()
    lst.clear()
    lst.show()


if __name__ == "__main__":
    main()
```

Both programs print exactly the output of **Listing 7.3**. Two differences from C++ stand out. First, there is no `delete`: when `frontDeletion` moves `head` past the first node, that node becomes unreachable, and the *garbage collector* reclaims its memory some time later. For the same reason

`clear` is a single assignment, `head = null`; all the nodes become unreachable at once. Second, the typical C++ bugs of Section 7.11, reading a freed node or freeing a node twice, cannot happen, because a reference is never left pointing at freed memory. What *can* still happen in Java and Python is losing nodes you meant to keep (for example by moving `head` in a traversal loop) and holding on to nodes you meant to discard (a forgotten reference keeps a whole chain alive), so the rule of the head applies in all three languages.

7.6 More Operations: Search, Length, Insertion in the Middle, Deletion by Key, Reversal

The eight operations of Section 7.5 work only at the two ends of the list. Real programs also need to find a value, count the nodes, insert or delete in the middle, keep the list sorted and turn it around. This section adds all of these to the head-pointer list. The complete program, **Listing 7.6**, and its output come in Section 7.6.9; the sections before it explain each function in turn. The node type there is the shorter `struct Node` of Section 7.4.1.

7.6.1 `search`

Searching a linked list is the sequential search of Chapter 2, written with pointers instead of indexes: walk from the first node and stop at the first node whose data equals the key. **Pseudocode 7.6** gives the procedure.

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
```

The order of the two tests in line 2 matters. When `aux` is NIL, the test `aux.data ≠ key` would read through a null pointer; because `and` in C++ (`&&`) evaluates its right operand only when the left one is true (*short-circuit evaluation*), writing `aux ≠ NIL` first guarantees that `aux.data` is read only for a real node.

In **Listing 7.6** the function returns the node, which is more useful than a yes or no: the caller can read its data, change it, or pass it to `insertAfter`. It also reports the node's index through a reference parameter, returning -1 when the key is absent.

Cost. In the worst case (the key is in the last node or absent) the loop visits all n nodes, so `search` is $\Theta(n)$ in the worst case and $O(1)$ in the best case. Unlike an array, a sorted linked list does *not* allow binary search to run in $O(\lg n)$ time: binary search jumps to the middle element, and by Equation (7.2) merely reaching the middle of a list costs $n/2$ steps.

7.6.2 `length`, iteratively and recursively

The iterative length is a traversal that counts. The recursive length rests on a definition of a list that is worth stating on its own:

> **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`).

The definition immediately gives a recursive formula for the length $L(p)$ of the list that starts at pointer p:

$$L(p) = \begin{cases} 0 & \text{if } p = \text{NULL} \\ 1 + L(p.\text{next}) & \text{otherwise} \end{cases} \tag{7.6}$$

and Equation (7.6) translates into C++ line for line:

```
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
}
```

For the list 20, 30, 40 the calls unfold as $L(20) = 1 + L(30) = 1 + (1 + L(40)) = 1 + (1 + (1 + L(\text{NULL}))) = 1 + 1 + 1 + 0 = 3$. Each call does a constant amount of work and there are $n + 1$ calls, so both versions take $\Theta(n)$ time. The difference is memory. The iterative version uses one counter. The recursive version keeps $n + 1$ calls waiting on the *call stack* at the deepest point, each holding its own copy of `head` and its return address, so it uses $\Theta(n)$ extra memory. A call stack is typically limited to a few megabytes (8 MB is the usual default on Linux), so a recursive walk over a list of a few million nodes can overflow it and crash. Chapter 10 studies recursion in depth; for lists, the lesson is that recursion expresses the definition beautifully and iteration is the safe choice for long lists.

7.6.3 `insertAfter`: insertion in the middle

Given a pointer `p` to a node in the list, inserting a new value after it needs no walking at all: the new node must point at `p`'s old successor, and `p` must point at the new node, as **Pseudocode 7.7** states.

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
```

Figure 7.8 inserts 35 after the node holding 30. In panel (b) the new node already points at 40 while 30 still points at 40 too; nothing is broken, because the list read from `head` is unchanged. Only step 2 (panel c) makes the new node part of the list. As in front insertion, reversing the two steps would make the new node point at itself and cut off 40 and everything after it.

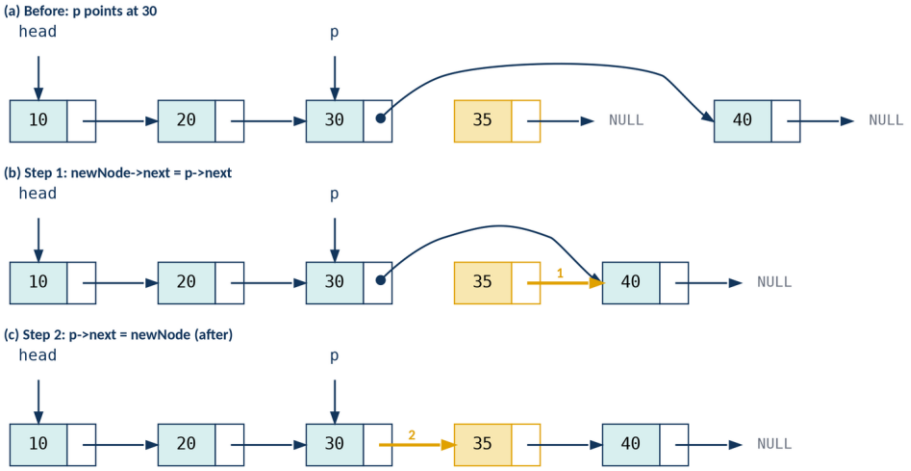


Figure 7.8 `insertAfter(p, 35)` with `p` at 30: (a) before, (b) the new node points at `p`'s successor, (c) `p` points at the new node.

`insertAfter` is $O(1)$. This is the operation that gives linked lists their reputation for cheap insertion; notice, though, that it is cheap only because the caller already holds `p`. Finding `p` in the first place usually needs a `search`, which is $\Theta(n)$.

Inserting *before* a given node is harder in a singly linked list, because the node before it is the one whose `next` must change and we cannot step backward to it. Exercise 7.9 solves the problem by searching with a lagging pointer.

7.6.4 `insertSorted`: keeping the list in order

A list that must stay in ascending order (a catalog kept sorted by book number, say) needs insertion at the right place. We walk a pointer `aux` forward while the *next* node's value is smaller than the new value; `aux` then stands on the last node that is smaller, and we insert after it. One case needs separate treatment: if the list is empty, or the new value is smaller than the first value, the new node becomes the new first node and `head` must change. **Pseudocode 7.8** handles that case first.

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)
```

Figure 7.9 shows both cases. Inserting 20 into 10, 30, 40, 50 stops `aux` at 10, because the node after 10 holds 30, which is not smaller than 20 (panel a), and `insertAfter` does the rest (panel b). Inserting 5 falls into the special case of line 1, so 5 becomes the first node (panel c).

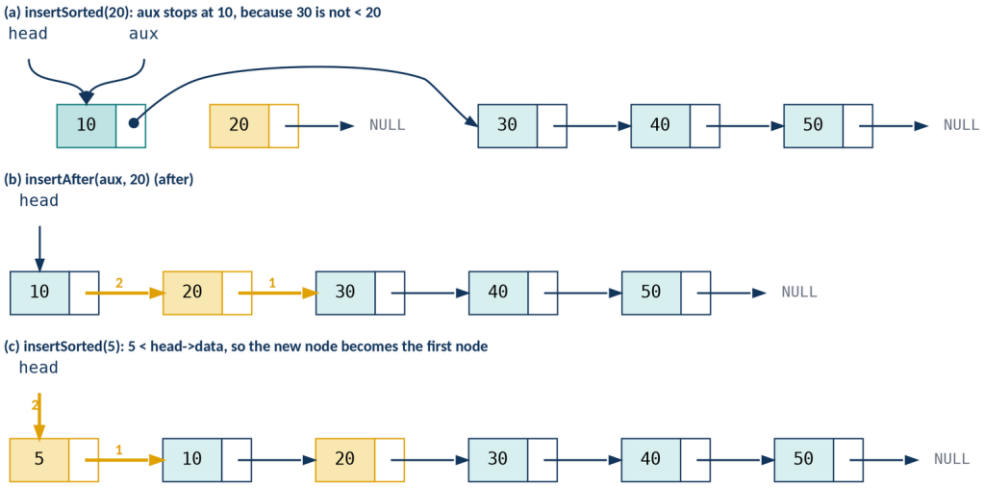


Figure 7.9 `insertSorted`: (a) for 20, `aux` stops on the last node smaller than 20, (b) the new node is inserted after it, (c) 5 is smaller than every value and becomes the first node.

Because line 5 uses `<`, a value equal to existing values is inserted after all of them, so equal keys keep their order of arrival: the insertion is *stable*, in the sense of Chapter 3. Inserting n values one by one with `insertSorted` is in fact *insertion sort* performed on a list, with the same $\Theta(n^2)$ worst case, but without a single element being shifted.

Cost. In the worst case (the new value is the largest) the loop visits every node: $\Theta(n)$.

7.6.5 `deleteKey`: deletion by value

To delete the first node holding a given key, we must find that node *and* the node before it, because the node before it is the one whose `next` must change. The standard technique, **Pseudocode 7.9**, walks two pointers together, `cur` on the candidate and `prev` one node behind it:

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
```

Figure 7.10 deletes 35 from 20, 30, 35, 40. When the loop stops (panel a), `cur` is on 35 and `prev` on 30. Line 11 makes 30 point at 40, so 35 is *bypassed*: a walk from `head` no longer meets it (panel b). Line 12 then releases it (panel c).

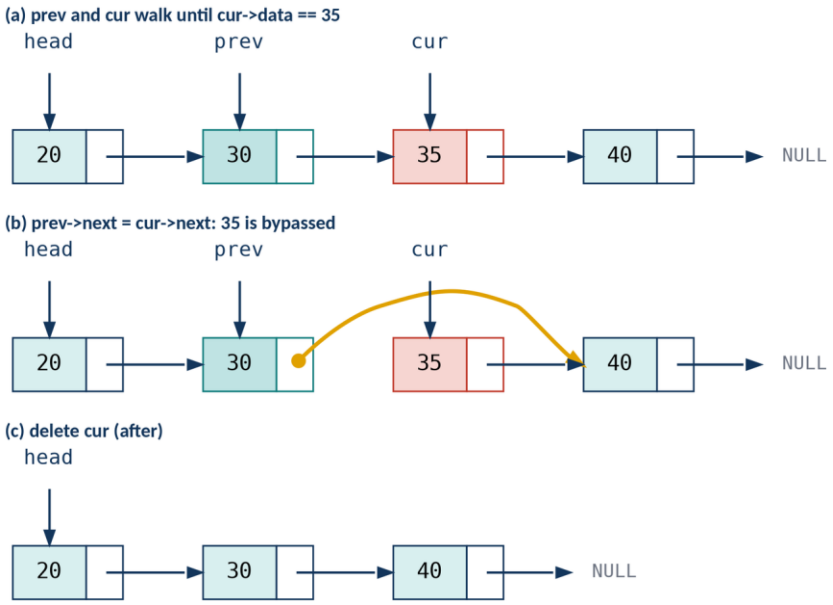


Figure 7.10 `deleteKey(35)`: (a) `prev` and `cur` stop with `cur` on 35, (b) `prev->next` skips over it, (c) the node is deleted.

The function covers four situations, and a correct implementation must handle every one of them; the program in Section 7.6.9 tests all four.

1. **The key is in the first node.** The loop does not run, `prev` stays NIL, and line 9 moves `head`. This is front deletion.
2. **The key is in a middle node.** Line 11 bypasses it.
3. **The key is in the last node.** Line 11 sets `prev.next` to `cur.next`, which is NIL, so `prev` becomes the new last node. This is back deletion, without a separate case.

4. **The key is absent.** `cur` runs off the end and the function reports failure without changing anything.

A fifth situation, the empty list, is case 4 with a loop that never starts.

Cost. $\Theta(n)$ in the worst case, for the walk; the unlinking itself is $O(1)$.

7.6.6 Reversing a list iteratively

To reverse a list we do not move any data; we turn every arrow around. The first node must end up pointing at NULL, every other node at the node that used to precede it, and `head` at what used to be the last node. The iterative algorithm, **Pseudocode 7.10**, walks once through the list with three pointers:

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
```

Figure 7.11 traces the four iterations on 20, 30, 40, 50. Line 4 is the one that makes the algorithm work: line 5 overwrites `cur.next`, which is the only record of where the rest of the list is, so it must be saved in `nxt` first.

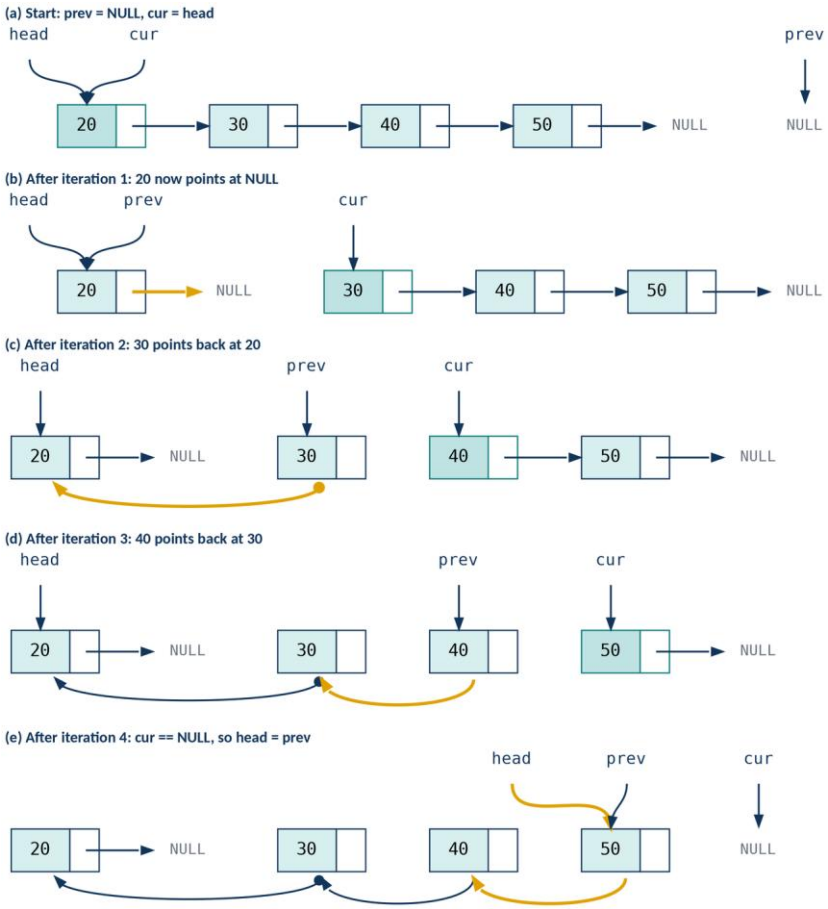


Figure 7.11 Iterative reversal of 20, 30, 40, 50: after each iteration one more arrow points backward; when `cur` reaches NULL, `prev` is the new first node.

Why is the algorithm correct? A *loop invariant*, a statement that is true every time the loop test is about to be evaluated, makes the argument precise.

> **LOOP INVARIANT.** At the start of each iteration of the loop in lines 3 to 7, `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 the two lists contain every node exactly once.

- **Initialization.** Before the first iteration no node has been visited: `prev` is NIL, the empty reversed list, and `cur` is `L.head`, the whole untouched list. The invariant holds.
- **Maintenance.** One iteration detaches the first node of the untouched list (after saving the rest in `nxt`), makes it point at the reversed list, and makes it the new head of the reversed list (`prev ← cur`); the untouched list now starts at `nxt`. Each list is still correct and every node is still in exactly one of them.
- **Termination.** The loop stops when `cur` is NIL, that is, when the untouched list is empty. By the invariant, `prev` is then the head of the whole list, reversed, and line 8 makes it the new head.

Cost. Each iteration is $O(1)$ and there are n iterations, so the reversal is $\Theta(n)$ time and $O(1)$ extra memory: three pointers, whatever the length.

7.6.7 Reversing a list recursively

The recursive definition of Section 7.6.2 suggests a second algorithm. To reverse the list that starts at `head`: if it has zero or one node it is already reversed; otherwise reverse the rest of the list (everything after `head`), and then attach `head` at the end of that reversed rest. **Pseudocode 7.11** writes this down.

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
```

Line 4 is the clever one. After line 3 has reversed the rest, `head.next` still points at the node that *used to be* second, which is now the *last* node of the reversed rest. So `head.next.next ← head` makes that node point back at `head`, and line 5 terminates the list at `head`. No walk to the end of the reversed rest is needed. **Figure 7.12** shows the three calls on 20, 30, 40 as they return.

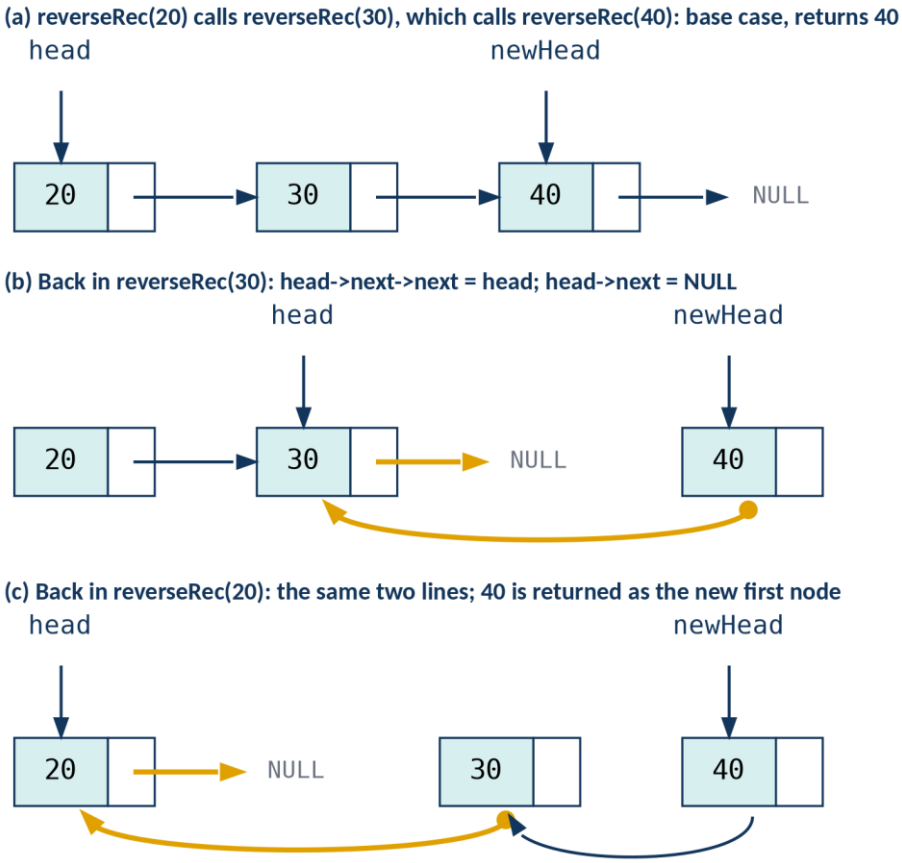


Figure 7.12 Recursive reversal of 20, 30, 40: the deepest call returns 40; each returning call turns one arrow around and cuts its own node's forward link.

Each call does constant work besides its recursive call, so its running time $R(n)$ on a list of n nodes satisfies

$$R(n) = R(n-1) + c, \quad R(1) = c \quad \Rightarrow \quad R(n) = c\,n = \Theta(n) \tag{7.7}$$

where c is the constant cost of lines 1, 2, 4, 5 and 6 (unfold the recurrence: $R(n) = R(n-2) + 2c = \cdots = R(1) + (n-1)c = nc$). The time is the same as the iterative version, but, as with `lengthRec`, the recursion depth is n, so the extra memory is $\Theta(n)$ and very long lists can overflow the call stack. The iterative version is the one to use in production; the recursive one is the one to understand, because the same pattern, "solve the rest, then fix up the first node", reappears in Chapter 10 and in the tree algorithms of Chapter 11.

7.6.8 `destroy`

`destroy` releases every node, exactly like `clear` in Section 7.5.7, and in **Listing 7.6** it also returns how many nodes it released, so that the output can confirm the count. Whatever the name, a list must be destroyed exactly once, and after destruction its `head` must be NULL so that no one uses the freed nodes by accident.

7.6.9 The complete program

Listing 7.6 exercises every function of this section. It builds a sorted list from 40, 10, 30, 50, 20 with `insertSorted`, searches for a present and an absent key, inserts 35 after 30, measures the length both ways, deletes a first, a middle, a last and an absent key, reverses the list twice (once each way), and destroys it.

Listing 7.6 `sll_ops.cpp`: search, length, insertion after a node and in sorted order, deletion by key, iterative and recursive reversal, and destruction.

```
// sll_ops.cpp -- search, length, insert after, sorted insert, delete by key,
// reverse (iterative and recursive) and destroy on a head-pointer list.
#include <cstdio>

struct Node {
    int data;
    Node *next;
};

void show(const char *label, const Node *head) {
    std::printf("%-22s", label);
    for (const Node *aux = head; aux != nullptr; aux = aux->next) {
        std::printf("%d -> ", aux->data);
    }
    std::printf("NULL\n");
}

// Returns the first node holding key, or nullptr. Also reports its index.
Node *search(Node *head, int key, int &index) {
    index = 0;
    for (Node *aux = head; aux != nullptr; aux = aux->next, ++index) {
        if (aux->data == key) return aux;
    }
    index = -1;
    return nullptr;
}

int lengthIter(const Node *head) {
    int n = 0;
    for (const Node *aux = head; aux != nullptr; aux = aux->next) ++n;
    return n;
}

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
}

// Inserts a new node right after node p (p must not be nullptr).
void insertAfter(Node *p, int newData) {
    Node *newNode = new Node{newData, p->next};   // 1. new node points at p's successor
    p->next = newNode;                            // 2. p points at the new node
}
```

Listing 7.6 (Continued) *sll_ops.cpp: search, length, insertion after a node and in sorted order, deletion by key, iterative and recursive reversal, and destruction.*

```
// Keeps the list in ascending order.
void insertSorted(Node *&head, int newData) {
    if (head == nullptr || newData < head->data) {    // new first node
        head = new Node{newData, head};
        return;
    }
    Node *aux = head;
    while (aux->next != nullptr && aux->next->data < newData) {
        aux = aux->next;                    // aux: last node smaller than newData
    }
    insertAfter(aux, newData);
}

// Deletes the first node holding key. Returns true if one was found.
bool deleteKey(Node *&head, int key) {
    Node *prev = nullptr;
    Node *cur = head;
    while (cur != nullptr && cur->data != key) {
        prev = cur;
        cur = cur->next;
    }
    if (cur == nullptr) return false;       // key is not in the list
    if (prev == nullptr) head = cur->next; // deleting the first node
    else prev->next = cur->next;            // bypass cur
    delete cur;
    return true;
}

void reverseIter(Node *&head) {
    Node *prev = nullptr;
    Node *cur = head;
    while (cur != nullptr) {
        Node *nxt = cur->next;              // 1. remember the rest of the list
        cur->next = prev;                   // 2. turn this arrow around
        prev = cur;                         // 3. step forward
        cur = nxt;
    }
    head = prev;
}
```

Listing 7.6 (Continued) *sll_ops.cpp: search, length, insertion after a node and in sorted order, deletion by key, iterative and recursive reversal, and destruction.*

```
// Reverses the list that starts at head and returns its new first node.
Node *reverseRec(Node *head) {
    if (head == nullptr || head->next == nullptr) return head;
    Node *newHead = reverseRec(head->next);    // reverse everything after head
    head->next->next = head;                   // the old second node now points back
    head->next = nullptr;                      // head is the new last node
    return newHead;
}

int destroy(Node *&head) {
    int released = 0;
    while (head != nullptr) {
        Node *del = head;
        head = head->next;
        delete del;
        ++released;
    }
    return released;
}

int main() {
    Node *head = nullptr;
    int keys[] = {40, 10, 30, 50, 20};
    for (int k : keys) {
        insertSorted(head, k);
        char label[32];
        std::snprintf(label, sizeof label, "insertSorted(%d):", k);
        show(label, head);
    }

    int index;
    Node *p = search(head, 30, index);
    std::printf("search(30): found at index %d\n", index);
    search(head, 99, index);
    std::printf("search(99): index %d (not found)\n", index);

    insertAfter(p, 35);
    show("insertAfter(30, 35):", head);
    std::printf("lengthIter = %d, lengthRec = %d\n", lengthIter(head), lengthRec(head));
```

Listing 7.6 (Continued) *sll_ops.cpp: search, length, insertion after a node and in sorted order, deletion by key, iterative and recursive reversal, and destruction.*

```
    int toDelete[] = {10, 35, 50, 99};
    for (int k : toDelete) {
        bool ok = deleteKey(head, k);
        char label[32];
        std::snprintf(label, sizeof label, "deleteKey(%d) %s:", k, ok ? "ok" : "no");
        show(label, head);
    }

    reverseIter(head);
    show("reverseIter:", head);
    head = reverseRec(head);
    show("reverseRec:", head);

    std::printf("destroy released %d nodes\n", destroy(head));
    show("after destroy:", head);
    return 0;
}
```

```
insertSorted(40):     40 -> NULL
insertSorted(10):     10 -> 40 -> NULL
insertSorted(30):     10 -> 30 -> 40 -> NULL
insertSorted(50):     10 -> 30 -> 40 -> 50 -> NULL
insertSorted(20):     10 -> 20 -> 30 -> 40 -> 50 -> NULL
search(30): found at index 2
search(99): index -1 (not found)
insertAfter(30, 35):  10 -> 20 -> 30 -> 35 -> 40 -> 50 -> NULL
lengthIter = 6, lengthRec = 6
deleteKey(10) ok:     20 -> 30 -> 35 -> 40 -> 50 -> NULL
deleteKey(35) ok:     20 -> 30 -> 40 -> 50 -> NULL
deleteKey(50) ok:     20 -> 30 -> 40 -> NULL
deleteKey(99) no:     20 -> 30 -> 40 -> NULL
reverseIter:          40 -> 30 -> 20 -> NULL
reverseRec:           20 -> 30 -> 40 -> NULL
destroy released 3 nodes
after destroy:        NULL
```

Every line of the output can be checked against the figures. The five sorted insertions produce 10, 20, 30, 40, 50, the case of **Figure 7.9**. 30 is at index 2. After `insertAfter`, 35 sits between 30 and 40 as in **Figure 7.8**. The four deletions exercise the four cases of Section 7.6.5 in order: first node (10), middle node (35), last node (50) and absent key (99, “no”). Reversing 20, 30, 40 gives 40, 30, 20, and reversing again restores 20, 30, 40. Finally three nodes are released and the list is empty. The Java and Python versions, `SllOps.java` and `sll_ops.py` in the chapter’s code folder, print the same lines.

Table 7.3 summarizes the costs of this section.

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`	$\Theta(n)$	$O(1)$

7.7 A Dummy Head Node: Fewer Special Cases

7.7.1 Where the special cases come from

Look back at the procedures of Sections 7.5 and 7.6 and count the special cases: `backInsertion` has a branch for the empty list, `insertSorted` a branch for “becomes the new first node”, `deleteKey` a branch for “deleting the first node”, `backDeletion` a branch for “exactly one node”. They all have the same cause. Every node except the first is pointed at by the `next` field of the node before it, but the first node is pointed at by `head`, which is a different kind of thing: a variable, not a field inside a node. So whenever an operation might change *which* node is first, it needs two versions of the same assignment, one for `head` and one for `prev->next`.

The classic cure is to make the first real node like every other node by putting a node before it. This extra node is called a *dummy node*, a *header node* or a *sentinel*. It is created when the list is created, it is never deleted while the list exists, and its data field is ignored.

DEFINITION. A singly linked list *with a dummy head node* always begins with one extra node whose data is meaningless. The list's pointer points at the dummy, the first real element is `dummy->next`, and the list is empty exactly when `dummy->next` is NULL. A list reached through a plain pointer to its first real node, as in Sections 7.5 and 7.6, is a list *without* a dummy node.

Figure 7.13 shows the empty list, which is not a null pointer any more but a single dummy node (panel a), the same list holding three values (panel b), and the insertion of 5 at the front of the real elements (panel c). In panel (c) the node before the insertion point is the dummy, so the insertion is an ordinary "insert after `prev`": no assignment to the list's pointer is needed.

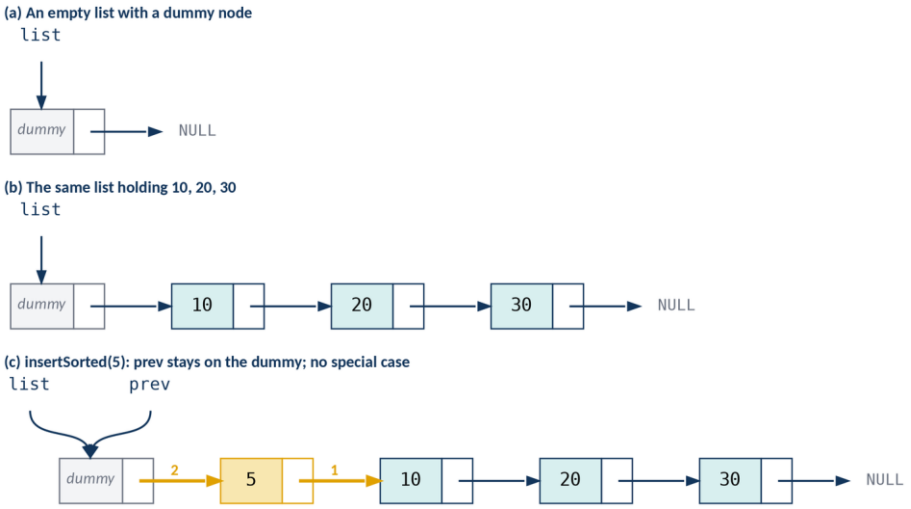


Figure 7.13 A list with a dummy head node: (a) the empty list is one dummy node, (b) a list holding 10, 20, 30, (c) inserting 5 before 10 is an insertion after the dummy, with no special case.

7.7.2 The operations without special cases

With a dummy in place, every node that holds data has a predecessor, so a pointer `prev` can always start at the dummy. **Listing 7.7** rewrites the operations. Compare `insertSorted` and `deleteKey` with their counterparts in **Listing 7.6**: each is now a single loop and a single link change, and the list pointer is never assigned after `init`, so the functions take it by value.

Listing 7.7 `sll_dummy.cpp`: a singly linked list with a dummy head node; insertion and deletion need no special case for the first element.

```cpp
// sll_dummy.cpp -- a singly linked list with a dummy (sentinel) head node.
// The dummy node holds no real data; the first real node is dummy->next.
#include <cstdio>

struct Node {
    int data;
    Node *next;
};

Node *init() {                          // an empty list is ONE node: the dummy
    return new Node{0, nullptr};
}

bool isEmpty(const Node *dummy) {
    return dummy->next == nullptr;
}

void show(const char *label, const Node *dummy) {
    std::printf("%-20s[dummy] -> ", label);
    for (const Node *aux = dummy->next; aux != nullptr; aux = aux->next) {
        std::printf("%d -> ", aux->data);
    }
    std::printf("NULL\n");
}

void insertFront(Node *dummy, int x) {       // no special case for an empty list
    dummy->next = new Node{x, dummy->next};
}

void insertBack(Node *dummy, int x) {
    Node *aux = dummy;                       // start at the dummy, never nullptr
    while (aux->next != nullptr) aux = aux->next;
    aux->next = new Node{x, nullptr};
}

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};
}
```

Listing 7.7 (Continued) *sll_dummy.cpp: a singly linked list with a dummy head node; insertion and deletion need no special case for the first element.*

```
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;
}

bool deleteFront(Node *dummy) {
    if (isEmpty(dummy)) return false;
    Node *del = dummy->next;
    dummy->next = del->next;
    delete del;
    return true;
}

void destroy(Node *&dummy) {                 // frees the real nodes AND the dummy
    while (dummy != nullptr) {
        Node *del = dummy;
        dummy = dummy->next;
        delete del;
    }
}

int main() {
    Node *list = init();
    show("init:", list);
    std::printf("isEmpty = %s\n", isEmpty(list) ? "true" : "false");

    insertSorted(list, 30);
    insertSorted(list, 10);
    insertSorted(list, 20);
    show("insertSorted x3:", list);
    insertFront(list, 5);
    insertBack(list, 99);
    show("front 5, back 99:", list);
```

Listing 7.7 (Continued) *sll_dummy.cpp: a singly linked list with a dummy head node; insertion and deletion need no special case for the first element.*

```
    deleteKey(list, 5);                      // the first real node: same code path
    deleteKey(list, 20);                     // a middle node
    deleteKey(list, 99);                     // the last node
    show("delete 5, 20, 99:", list);
    deleteFront(list);
    deleteFront(list);
    show("deleteFront x2:", list);
    std::printf("isEmpty = %s\n", isEmpty(list) ? "true" : "false");

    destroy(list);
    std::printf("destroyed, list == nullptr is %s\n", list == nullptr ? "true" : "false");
    return 0;
}
```

```
init:               [dummy] -> NULL
isEmpty = 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
isEmpty = true
destroyed, list == nullptr is true
```

The output shows the dummy explicitly as `[dummy]`. Deleting 5, the first real element, runs exactly the same code as deleting 20 in the middle or 99 at the end; so does inserting 5 at the front with `insertFront`, which is nothing but `insertAfter(dummy, x)`. Two things change elsewhere. `isEmpty` now tests `dummy->next`, and `destroy` releases the dummy too, which is why, once the list has been destroyed, the program reports that `list` is `nullptr`. The Java and Python versions, `SllDummy.java` and `sll_dummy.py`, print the same output.

The price of the dummy is one node of memory per list, and one rule that every function must respect: the dummy is not data, so traversals begin at `dummy->next`, and `size` must not count it. For a program with many short lists (a hash table of Chapter 15 may hold thousands of them) the extra node per list can matter; for one long list it does not.

7.7.3 A third way: a pointer to the link

C and C++ offer a third technique that removes the special cases *without* a dummy node. The observation is that the thing an insertion or deletion changes is always a *link*, a variable of type `Node *`, which is either `head` itself or the `next` field of some node. So instead of walking a pointer to the *node* before the position, we walk a pointer to the *link* that points at the position: a `Node **link`, which starts as `&head` and then becomes `&(node->next)` for successive nodes. Writing `*link = ...` then changes `head` or a `next` field, whichever `link` currently addresses, with the same line of code. **Listing 7.8** applies the idea to sorted insertion, deletion by key and deletion of every occurrence of a key.

Listing 7.8 `sll_link.cpp`: walking a pointer to the link (`Node **`) removes the special case for the first node without a dummy node.

```cpp
// sll_link.cpp -- no dummy node and no special case: walk a pointer to the LINK.
// link is the address of "the pointer that points at the current node":
// first &head, later &(some node)->next.
#include <cstdio>

struct Node {
    int data;
    Node *next;
};

void show(const char *label, const Node *head) {
    std::printf("%-18s", label);
    for (; head != nullptr; head = head->next) std::printf("%d -> ", head->data);
    std::printf("NULL\n");
}

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;
}

int removeAll(Node **link, int key) {    // delete every node holding key
    int removed = 0;
    while (*link != nullptr) {
        if ((*link)->data == key) {
            Node *del = *link;
            *link = del->next;           // do NOT advance: *link is a new node now
            delete del;
            ++removed;
        } else {
            link = &(*link)->next;
        }
    }
    return removed;
}
```

Listing 7.8 (Continued) *sll_link.cpp: walking a pointer to the link (Node **) removes the special case for the first node without a dummy node.*

```cpp
int main() {
    Node *head = nullptr;
    int keys[] = {30, 10, 20, 20, 5, 20};
    for (int k : keys) insertSorted(&head, k);
    show("sorted:", head);

    deleteKey(&head, 5);                 // the first node: head itself changes
    show("deleteKey(5):", head);
    int n = removeAll(&head, 20);
    std::printf("removeAll(20) removed %d nodes\n", n);
    show("now:", head);

    while (head != nullptr) deleteKey(&head, head->data);   // empty the list
    show("emptied:", head);
    return 0;
}
```

```
sorted:           5 -> 10 -> 20 -> 20 -> 20 -> 30 -> NULL
deleteKey(5):     10 -> 20 -> 20 -> 20 -> 30 -> NULL
removeAll(20) removed 3 nodes
now:              10 -> 30 -> NULL
emptied:          NULL
```

In `insertSorted(&head, x)`, if the new value is the smallest the loop does not move and `*link = new Node{x, *link}` assigns to `head`; otherwise `link` holds the address of the `next` field of the last smaller node and the same line assigns to that field. `removeAll` shows the technique at its best: when it deletes a node it does *not* advance `link`, because `*link` now holds the node after the deleted one, which must be examined too; the three consecutive 20s are removed correctly. This technique is often praised as an example of "good taste" in low-level code, because it replaces an `if` about the first node with a better choice of what to point at [7]. It has no equivalent in Java or Python, which cannot take the address of a field; there the dummy node is the tool.

7.7.4 Which variant to use

All three variants have the same asymptotic costs. The choice is about clarity and memory: the plain head pointer is the simplest to explain and costs nothing extra; the dummy node gives the shortest, least error-prone code in every language, for one extra node; the pointer to the link gives the same short code without the extra node, but only in C and C++, and it asks the reader to be comfortable with `Node **`. The library lists of Section 7.12 are built with sentinel ideas of this kind internally: `std::forward_list`'s `before_begin()` returns an iterator to a position *before* the first element precisely so that `insert_after` and `erase_after` can work at the front without a special case.

7.8 Head and Tail Pointers

7.8.1 The idea

Back insertion on the head-pointer list costs $\Theta(n)$ because the list does not know where its last node is. The obvious remedy is to remember it: keep a second pointer, `tail`, that always points at the last node. The list now has two external pointers, and every operation must keep both correct. The payoff is shown in panels (a) to (c) of **Figure 7.14**: back insertion attaches the new node to `tail` and moves `tail`, with no walk at all.

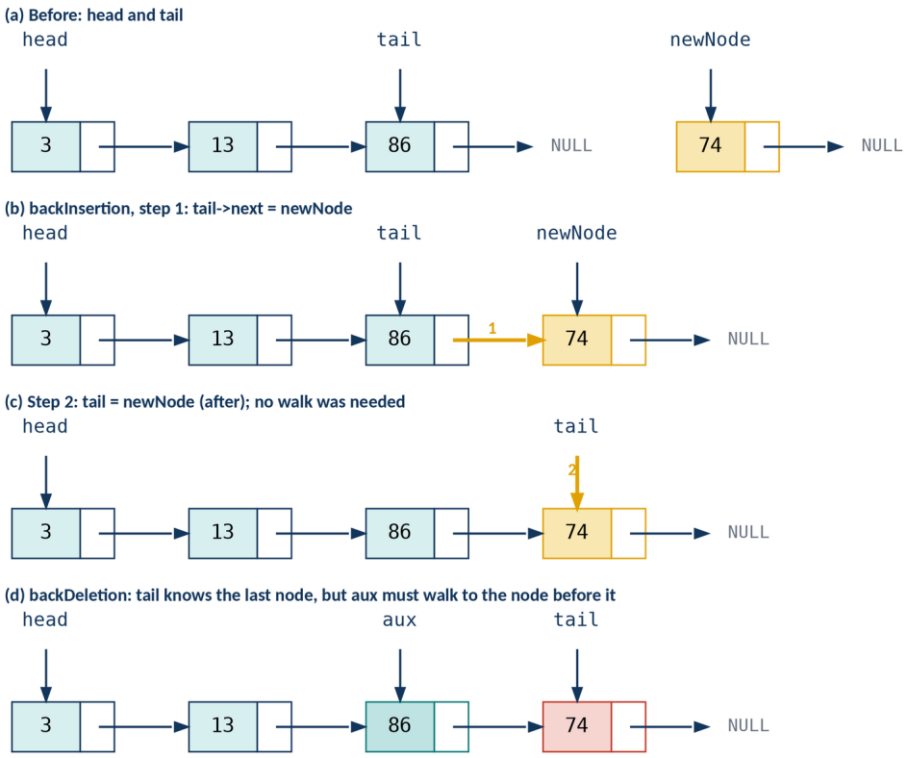


Figure 7.14 A list with `head` and `tail`: (a) before `backInsertion(74)`, (b) the old last node points at the new node, (c) `tail` moves to it; (d) `backDeletion` still has to walk `aux` to the node before `tail`.

The emptiness test can now look at either pointer; the classic formulation tests `tail == nullptr`. The invariant that every function must preserve is:

> **THE HEAD-TAIL INVARIANT.** Either the list is empty and `head` and `tail` are both NULL, or `head` points at the first node, `tail` points at the last node, and `tail->next` is NULL. In a list of exactly one node, `head` and `tail` point at the same node.

The invariant tells us where the new special cases are: every operation that can move the list between "empty" and "one node" must change *both* pointers. Front insertion into an empty list must set `tail` as well as `head`; front deletion of the only node must set `tail` back to NULL as well as `head`. Forgetting the second of these leaves `tail` pointing at a freed node, and the next back insertion writes into freed memory.

7.8.2 Why `tail` cannot speed up back deletion

It is tempting to conclude that `tail` makes back deletion $O(1)$ as well. It does not, and the reason is worth understanding exactly. `tail` tells us immediately *which* node to delete. But deleting it means that the node *before* it must become the new last node: its `next` must become NULL and `tail` must move to it. A singly linked node records who comes after it, never who comes before it, and nothing else in the list records that either. The only way to find the node whose `next` equals `tail` is to walk from `head` until we meet it, as panel (d) of **Figure 7.14** shows. So back deletion stays $\Theta(n)$ with a tail pointer: $n - 2$ steps on a list of $n \geq 2$ nodes.

Could a third pointer, to the second-to-last node, fix this? Only for one deletion: after that deletion we would need the *third*-to-last node, and so on. The real fix is to give every node a pointer to its predecessor, which is the doubly linked list of Chapter 9.

7.8.3 The program

Listing 7.9 follows the classic presentation with global `head` and `tail` pointers. `backDeletion` counts its steps so that the output shows the $n - 2$ cost directly.

Listing 7.9 `sll_head_tail.cpp`: a singly linked list with `head` and `tail`; back insertion is $O(1)$, back deletion still walks.

```
// sll_head_tail.cpp -- a singly linked list with two pointers: head and tail.
#include <cstdio>

typedef struct MyNode {
    int data;
    struct MyNode *next;
} TNode;

TNode *head, *tail;

void init() {
    head = nullptr;
    tail = nullptr;
}

bool isEmpty() {
    return tail == nullptr;
}

void frontInsertion(int newData) {
    TNode *newNode = new TNode{newData, head};
    if (isEmpty()) tail = newNode;        // the only node is also the last one
    head = newNode;
    std::printf("%d has been inserted by frontInsertion()\n", newData);
}

void backInsertion(int newData) {         // O(1): no walk, tail knows the end
    TNode *newNode = new TNode{newData, nullptr};
    if (isEmpty()) {
        head = tail = newNode;
    } else {
        tail->next = newNode;             // 1. old last node points at the new one
        tail = newNode;                   // 2. tail moves to the new node
    }
    std::printf("%d has been inserted by backInsertion()\n", newData);
}

void show() {
    if (isEmpty()) {
        std::printf("The linked list is empty\n");
        return;
    }
    for (TNode *aux = head; aux != nullptr; aux = aux->next) std::printf("%d ", aux->data);
    std::printf("\n");
}
```

Listing 7.9 (Continued) *sll_head_tail.cpp: a singly linked list with head and tail; back insertion is O(1), back deletion still walks.*

```
void frontDeletion() {
    if (isEmpty()) {
        std::printf("The linked list is empty\n");
        return;
    }
    TNode *del = head;
    int d = del->data;
    if (head == tail) {                  // one node: the list becomes empty
        head = tail = nullptr;
    } else {
        head = head->next;
    }
    delete del;
    std::printf("%d has been deleted by frontDeletion()\n", d);
}

void backDeletion() {                    // still O(n): who points at tail?
    if (isEmpty()) {
        std::printf("The linked list is empty\n");
        return;
    }
    TNode *del = tail;
    int d = del->data;
    int steps = 0;
    if (head == tail) {
        head = tail = nullptr;
    } else {
        TNode *aux = head;
        while (aux->next != tail) {      // find the node just before tail
            aux = aux->next;
            ++steps;
        }
        aux->next = nullptr;
        tail = aux;
    }
    delete del;
    std::printf("%d has been deleted by backDeletion() after %d step(s)\n", d, steps);
}
```

Listing 7.9 (Continued) *sll_head_tail.cpp: a singly linked list with head and tail; back insertion is O(1), back deletion still walks.*

```
void clear() {
    TNode *aux = head;
    while (aux != nullptr) {
        TNode *del = aux;
        aux = aux->next;
        delete del;
    }
    head = tail = nullptr;
    std::printf("The linked list has been cleared\n");
}

int main() {
    init();
    frontInsertion(13);
    frontInsertion(3);
    backInsertion(86);
    backInsertion(74);
    backInsertion(52);
    backInsertion(61);
    show();

    frontDeletion();
    backDeletion();
    backDeletion();
    backDeletion();
    show();
    std::printf("head->data = %d, tail->data = %d\n", head->data, tail->data);

    clear();
    show();
    return 0;
}
```

```
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()
52 has been inserted by backInsertion()
61 has been inserted by backInsertion()
3 13 86 74 52 61
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)
13 86
head->data = 13, tail->data = 86
The linked list has been cleared
The linked list is empty
```

After the six insertions the list is 3, 13, 86, 74, 52, 61. Front deletion removes 3, leaving five nodes, and the three back deletions take 3, 2 and 1 steps on lists of 5, 4 and 3 nodes: exactly $n - 2$ each time. The final line of real data confirms the invariant: `head` holds 13 and `tail` holds 86, the first and last of the remaining list 13, 86.

Table 7.4 compares the variants built so far.

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

7.9 The Circular Singly Linked List

7.9.1 Closing the chain

In the lists so far the last node's `next` is NULL. If we instead make it point back at the first node, the chain becomes a ring: a *circular singly linked list*. Nothing about the node changes; only the value stored in one `next` field does. Panel (a) of **Figure 7.15** shows a circular list of four nodes.

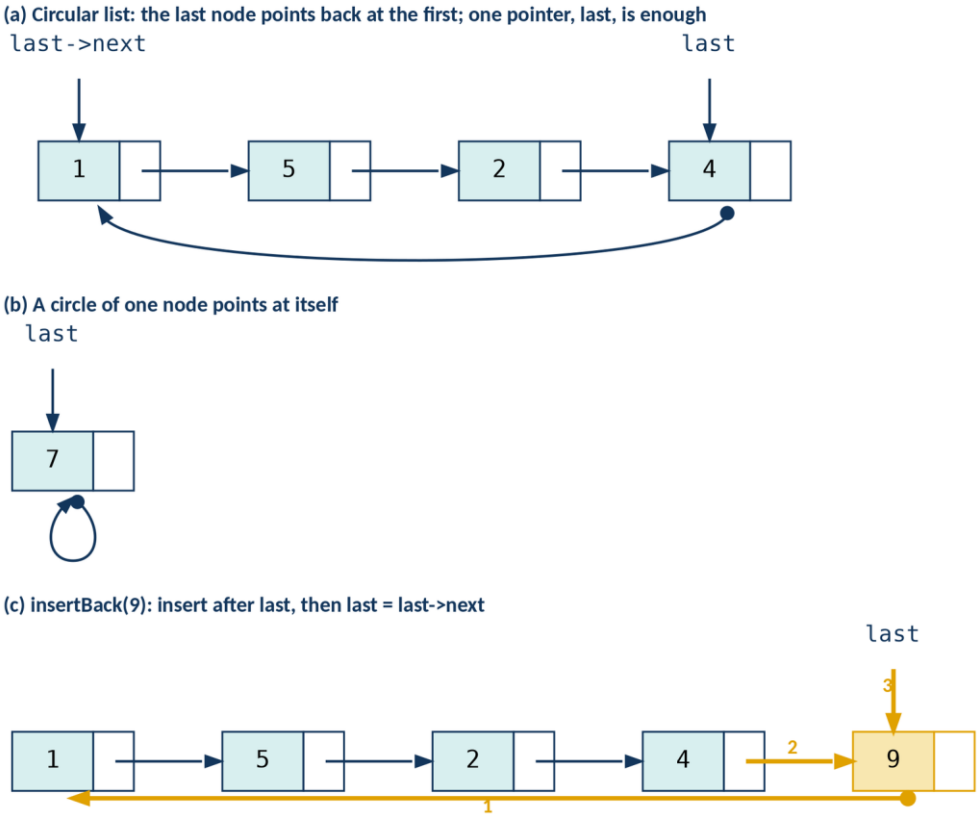


Figure 7.15 A circular singly linked list: (a) four nodes, reached through one pointer `last`, with the first node at `last->next`; (b) a one-node circle points at itself; (c) inserting at the back is an insertion after `last` followed by moving `last`.

A ring has no natural first node, so where should the external pointer point? The surprising best answer is: at the *last* node. From `last`, the first node is one step away, `last->next`, so a single pointer gives $O(1)$ access to *both* ends, the job that needed two pointers in Section 7.8. With `last` we can:

- **insert at the front** in $O(1)$: link the new node between `last` and `last->next`;

- **insert at the back** in $O(1)$: insert at the front, then move `last` one step forward onto the new node, which makes it the last node instead (panel c);
- **delete at the front** in $O(1)$: unlink `last->next`.

Deleting at the back is still $\Theta(n)$, for the reason of Section 7.8.2: the node before `last` can only be found by walking around the ring.

Two cases need care. An empty circular list is `last == nullptr`. A circle of *one* node must point at itself (panel b): its `next` is its own address. Deleting the only node must therefore set `last` back to `nullptr`, which is detected by the test "the node to delete is its own successor".

7.9.2 Traversal needs a new stop signal

The traversal loop of **Pseudocode 7.3**, `while (aux != nullptr)`, would never stop on a circular list, because no `next` field is ever NULL: the loop would print the ring forever. A circular traversal must instead stop when it returns to where it started. Since an empty list has no starting node, the loop runs only when the list is non-empty, and it is written as a `do-while`, which tests *after* the body so that the first node is printed before the test "are we back at the start?" is asked for the first time:

```
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
```

An ordinary `while (aux != last->next)` would fail immediately, before visiting anything, because `aux` *starts* at `last->next`. Counting the nodes and stopping after `size` steps is an equally good alternative when the list keeps its size.

7.9.3 Round-robin service at the reading desk

A ring is the natural structure for anything served *in turn, repeatedly*: every participant gets a short turn, then the next one does, and after the last participant the first one comes around again. Operating systems share a processor among running programs in exactly this way; the technique is called *round-robin scheduling*, and each turn is a *time slice*.

Pustaka Ceria has one reading desk with a special magnifying reader, and four visitors want to use it: Ani for 5 minutes, Budi for 2, Citra for 4 and Dewi, who arrived with a priority pass and is put at the front, for 1. To be fair, each visitor gets at most 2 minutes at a time and then goes to the back of the circle; a visitor whose time is used up leaves the circle. **Listing 7.10** simulates the session with a circular list. The pointer `prev` always stands on the node *before* the visitor being served, which is exactly what is needed to delete that visitor in $O(1)$ when they are done, using `deleteAfter(prev)`.

Listing 7.10 `csll.cpp`: a circular singly linked list reached through `last`, used for round-robin service with a 2-minute time slice.

```cpp
// csll.cpp -- a circular singly linked list kept through ONE pointer, last.
// The first node is always last->next. Used here for a round-robin reading room.
#include <cstdio>
#include <string>

struct Reader {
    std::string name;
    int minutesLeft;          // reading time the reader still needs
};

struct Node {
    Reader data;
    Node *next;
};

bool isEmpty(const Node *last) { return last == nullptr; }

void insertFront(Node *&last, const Reader &r) {
    Node *newNode = new Node{r, nullptr};
    if (isEmpty(last)) {
        newNode->next = newNode;          // a circle of one node points at itself
        last = newNode;
    } else {
        newNode->next = last->next;       // 1. new node points at the old first node
        last->next = newNode;             // 2. last node points at the new first node
    }
}

void insertBack(Node *&last, const Reader &r) {
    insertFront(last, r);                 // put it between last and first ...
    last = last->next;                    // ... and call it the last node: O(1)
}

// Removes the node that follows prev (prev->next) and 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;
}
```

Listing 7.10 (Continued) *csll.cpp: a circular singly linked list reached through last, used for round-robin service with a 2-minute time slice.*

```
void show(const char *label, const Node *last) {
    std::printf("%s", label);
    if (isEmpty(last)) {
        std::printf(" (empty)\n");
        return;
    }
    const Node *aux = last->next;         // start at the first node
    do {                                  // stop when we come back to the start
        std::printf(" %s(%d)", aux->data.name.c_str(), aux->data.minutesLeft);
        aux = aux->next;
    } while (aux != last->next);
    std::printf("\n");
}

void destroy(Node *&last) {
    while (!isEmpty(last)) deleteAfter(last, last);   // delete the first node
}

int main() {
    Node *last = nullptr;
    insertBack(last, {"Ani", 5});
    insertBack(last, {"Budi", 2});
    insertBack(last, {"Citra", 4});
    insertFront(last, {"Dewi", 1});
    show("Queue for the one reading desk:", last);

    const int slice = 2;                  // each turn lasts at most 2 minutes
    int clock = 0;
    Node *prev = last;                    // the node BEFORE the one being served
    while (!isEmpty(last)) {
        Node *cur = prev->next;
        int used = cur->data.minutesLeft < slice ? cur->data.minutesLeft : slice;
        cur->data.minutesLeft -= used;
        clock += used;
        std::printf("t=%2d  %-5s read %d min", clock, cur->data.name.c_str(), used);
        if (cur->data.minutesLeft == 0) {
            std::printf(" and is done\n");
            deleteAfter(last, prev);      // prev stays; its next is the following reader
        } else {
            std::printf(", %d min left\n", cur->data.minutesLeft);
            prev = cur;                   // advance around the circle
        }
    }
    show("After the session:", last);
```

Listing 7.10 (Continued) *csll.cpp: a circular singly linked list reached through last, used for round-robin service with a 2-minute time slice.*

```
    insertBack(last, {"Eko", 3});
    insertBack(last, {"Fajar", 3});
    show("Evening readers:", last);
    destroy(last);
    show("After destroy:", last);
    return 0;
}
```

```
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)
Evening readers: Eko(3) Fajar(3)
After destroy: (empty)
```

Trace the first round. Dewi needs 1 minute, reads it and leaves ($t = 1$). Ani reads 2 of her 5 minutes ($t = 3$). Budi reads his full 2 minutes and leaves ($t = 5$). Citra reads 2 of her 4 ($t = 7$). The ring now holds only Ani (3 minutes left) and Citra (2 left), and the service continues around it without any special code for "we reached the end", because in a ring there is no end: Ani reads 2 ($t = 9$), Citra finishes ($t = 11$), and Ani finishes last ($t = 12$). The total, 12 minutes, equals $1 + 5 + 2 + 4$: no time is wasted, however the ring shrinks. The program then reuses the empty ring for two evening readers and destroys it; the Java and Python versions print the same output.

Two other classic applications of circular lists appear in the exercises: the Josephus problem (Exercise 7.11), where people standing in a circle leave one by one, and cycle detection (Exercise 7.6), which answers the question "is this list accidentally circular?" without modifying it.

7.10 Stacks and Queues Built on a Linked List

7.10.1 The fit

Chapter 5 defined a *stack* (last in, first out: `push`, `pop`, `peek`) and a *queue* (first in, first out: `enqueue`, `dequeue`) and built both on arrays, with a fixed capacity and, for the queue, the circular-index arithmetic that avoids shifting. Linked lists give both structures without a capacity and without index arithmetic, because the operations they need are exactly the $O(1)$ operations of **Table 7.4**.

- A **stack** needs one end where elements are both added and removed. The front of a singly linked list is perfect: `push` is front insertion and `pop` is front deletion, both $O(1)$. The top of the stack is the first node, and one pointer, `top_`, suffices.
- A **queue** needs elements added at one end and removed at the other. With a head-and-tail list, insertion at the back and deletion at the front are both $O(1)$, so `enqueue` inserts at the tail (`rear_`) and `dequeue` deletes at the head (`front_`). The other assignment, adding at the front and removing at the back, would make `dequeue` a $\Theta(n)$ back deletion; the direction matters.

Figure 7.16 shows both, holding book identifiers of the Pustaka Ceria catalog.

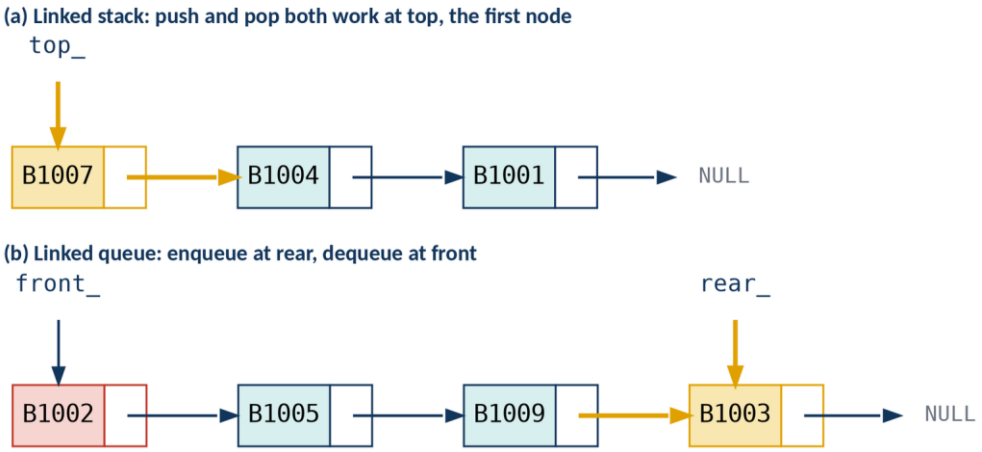


Figure 7.16 (a) A linked stack: `push` and `pop` work at the first node, `top_`. (b) A linked queue: `enqueue` links a new node after `rear_`, `dequeue` removes the node at `front_`.

7.10.2 The Pustaka Ceria return cart and hold queue

At Pustaka Ceria, returned books are stacked on a cart and shelved from the top down, so the last book returned is the first one shelved: a stack. Readers who reserve a book that is on loan wait in a hold queue and are served in order of request: a queue. **Listing 7.11** implements both as C++ classes built on singly linked nodes.

Listing 7.11 `linked_stack_queue.cpp`: `LinkedStack` and `LinkedQueue` built on singly linked nodes, with destructors that release every node.

```cpp
// linked_stack_queue.cpp -- a stack and a queue built on singly linked nodes,
// used by the Pustaka Ceria library: a cart of returned books and a hold queue.
#include <iostream>
#include <stdexcept>
#include <string>

struct Book {
    std::string id;
    std::string title;
};

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

    bool isEmpty() const { return top_ == nullptr; }
    int size() const { return size_; }
    void push(const Book &b) {            // front insertion: O(1)
        top_ = new Node{b, top_};
        ++size_;
    }
    Book pop() {                          // front deletion: O(1)
        if (isEmpty()) throw std::underflow_error("pop from an empty stack");
        Node *del = top_;
        Book b = del->data;
        top_ = top_->next;
        delete del;
        --size_;
        return b;
    }
    const Book &peek() const {
        if (isEmpty()) throw std::underflow_error("peek at an empty stack");
        return top_->data;
    }
};
```

Listing 7.11 (Continued) *linked_stack_queue.cpp: LinkedStack and LinkedQueue built on singly linked nodes, with destructors that release every node.*

```
class LinkedQueue {                       // front = first node, rear = last node
    struct Node { Book data; Node *next; };
    Node *front_ = nullptr;
    Node *rear_ = nullptr;
    int size_ = 0;
public:
    LinkedQueue() = default;
    LinkedQueue(const LinkedQueue &) = delete;
    LinkedQueue &operator=(const LinkedQueue &) = delete;
    ~LinkedQueue() { while (!isEmpty()) dequeue(); }

    bool isEmpty() const { return front_ == nullptr; }
    int size() const { return size_; }
    void enqueue(const Book &b) {         // back insertion with a tail: O(1)
        Node *n = new Node{b, nullptr};
        if (isEmpty()) front_ = n;
        else rear_->next = n;
        rear_ = n;
        ++size_;
    }
    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;
    }
};

int main() {
    LinkedStack cart;                     // returned books, the last one lands on top
    cart.push({"B1001", "Laskar Pelangi"});
    cart.push({"B1004", "Bumi Manusia"});
    cart.push({"B1007", "Filosofi Kopi"});
    std::cout << "Books on the return cart: " << cart.size() << '\n';
    std::cout << "Top of the cart: " << cart.peek().title << '\n';
    while (!cart.isEmpty()) {
        Book b = cart.pop();
        std::cout << "  shelve " << b.id << " " << b.title << '\n';
    }
```

Listing 7.11 (Continued) *linked_stack_queue.cpp: LinkedStack and LinkedQueue built on singly linked nodes, with destructors that release every node.*

```
    LinkedQueue holds;                    // hold requests, first come first served
    holds.enqueue({"B1002", "Negeri 5 Menara"});
    holds.enqueue({"B1005", "Ronggeng Dukuh Paruk"});
    holds.enqueue({"B1009", "Cantik Itu Luka"});
    std::cout << "Hold requests waiting: " << holds.size() << '\n';
    std::cout << "  serve " << holds.dequeue().title << '\n';
    holds.enqueue({"B1003", "Pulang"});
    while (!holds.isEmpty()) std::cout << "  serve " << holds.dequeue().title << '\n';

    try {
        holds.dequeue();
    } catch (const std::underflow_error &e) {
        std::cout << "Error caught: " << e.what() << '\n';
    }

    holds.enqueue({"B1010", "Gadis Kretek"});   // left in the queue on purpose:
    cart.push({"B1011", "Sang Pemimpi"});       // the destructors free these nodes
    std::cout << "Leaving main with " << holds.size() + cart.size() << " nodes still
linked\n";
    return 0;
}
```

```
Books on the return cart: 3
Top of the cart: Filosofi Kopi
  shelve B1007 Filosofi Kopi
  shelve B1004 Bumi Manusia
  shelve B1001 Laskar Pelangi
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
Leaving main with 2 nodes still linked
```

The output shows the two disciplines. The cart receives Laskar Pelangi, Bumi Manusia and Filosofi Kopi and shelves them in the reverse order. The hold queue serves Negeri 5 Menara first, and Pulang, requested last, is served last. Asking an empty queue for another request raises the exception `std::underflow_error`, which `main` catches.

This program introduces three features that the earlier programs of the chapter did not have, and each fixes a real danger.

- **The node type is private.** `struct Node` is declared inside each class, so no code outside the class can reach a node and break the links. The array-based classes of Chapter 5 hid their array for the same reason.
- **The destructor releases the nodes.** `~LinkedStack()` pops until the stack is empty, and `~LinkedQueue()` dequeues until the queue is empty. C++ calls a destructor automatically when an object's lifetime ends, here at the end of `main`, so the two nodes deliberately left in the structures ("Leaving main with 2 nodes still linked") are released without any explicit call. Tying the release of a resource to the lifetime of an object in this way is called *RAII* (resource acquisition is initialization), and it is the idiomatic C++ answer to the `clear`-before-you-forget problem of Section 7.5.7.
- **Copying is forbidden.** The lines `LinkedStack(const LinkedStack &) = delete;` and the matching assignment operator tell the compiler not to generate its default copy, which would copy only the pointer `top_`. Two stacks would then share the same nodes, and when both destructors ran, each node would be deleted twice. This is the *rule of three*: a class that needs a

destructor almost always needs a correct copy constructor and copy assignment, or must forbid them.

`dequeue` contains the queue's one subtle line, `if (front_ == nullptr) rear_ = nullptr;`. When the last node leaves, `front_` becomes NULL naturally, but `rear_` still points at the node just deleted. This class happens not to read `rear_` while `front_` is NULL, so it would survive without the line; but a version that tests emptiness with `rear_ == nullptr`, as the head-and-tail list of Section 7.8 tests `tail`, would then write `rear_->next` into freed memory on the next `enqueue`. Resetting both pointers whenever the queue becomes empty makes every such test safe. Exercise 7.12 asks you to follow the failure through and say what the memory checkers would report.

The Java version, `LinkedStackQueue.java`, uses a `record Book` and throws `NoSuchElementException`; the Python version, `linked_stack_queue.py`, uses a frozen `dataclass` and raises `IndexError`. Both print the same output and need no destructors, because the garbage collector reclaims the nodes.

7.10.3 Linked or array-based?

Both implementations give $O(1)$ `push`, `pop`, `enqueue` and `dequeue`. The linked versions never become full and use memory proportional to the current number of elements. The array versions use less memory per element (no pointer) and are faster in practice, because their elements share cache lines and they do not call the allocator for every operation. A growable array, such as `std::vector` used as a stack, gets $O(1)$ *amortized* push, meaning that an occasional expensive regrowth is paid for by many cheap pushes. For this reason the C++ standard library's `std::stack` and `std::queue` use a `std::deque`, a sequence of fixed-size arrays, by default rather than a list; Python's `collections.deque` takes a similar hybrid approach, as Section 7.12 explains. Linked stacks and queues remain the right choice when elements are large, when a fixed capacity is unacceptable and regrowth pauses must be avoided, or when nodes are already part of another linked structure.

7.11 Memory Safety: Proving There Are No Leaks

7.11.1 Why "it printed the right answer" is not enough

Every C++ program in this chapter calls `new` and `delete` directly, and a mistake with either one usually produces *no visible symptom at all*. A leaked node does not change the output. A node read after it was deleted often still contains its old values, because nothing has overwritten them yet, so the program prints the right answer, today, on this machine. The same program may crash tomorrow, under a different compiler option or with a different input, or, worse, silently corrupt other data. Four kinds of mistakes account for almost all linked-list memory bugs.

1. **Memory leak.** A node is never deleted. The typical cause is forgetting `clear`, or moving the only pointer to a node before deleting it.
2. **Use after free.** A node is read or written after `delete`. The typical cause is a second pointer (a saved `second`, a stale `tail`, a helper `aux`) that still holds the address of a node deleted through the first pointer.
3. **Dangling link.** A special case of use after free inside the list itself: a node is deleted but a `next` field (or `tail`) still points at it, as when line 11 of **Pseudocode 7.5** is forgotten.
4. **Double free.** The same node is deleted twice, typically because two pointers to it both went through a `clear`.

A fifth mistake is not a memory error by itself but produces the first one on a large scale: performing two pointer assignments in the wrong order. **Figure 7.17** shows `frontInsertion` with its two lines swapped. After `head = newNode`, the only pointer to the old first node is gone; `newNode->next = head` then makes the new node point at itself. The list now reads 3, 3, 3, forever, and the nodes 13 and 86 are leaked.

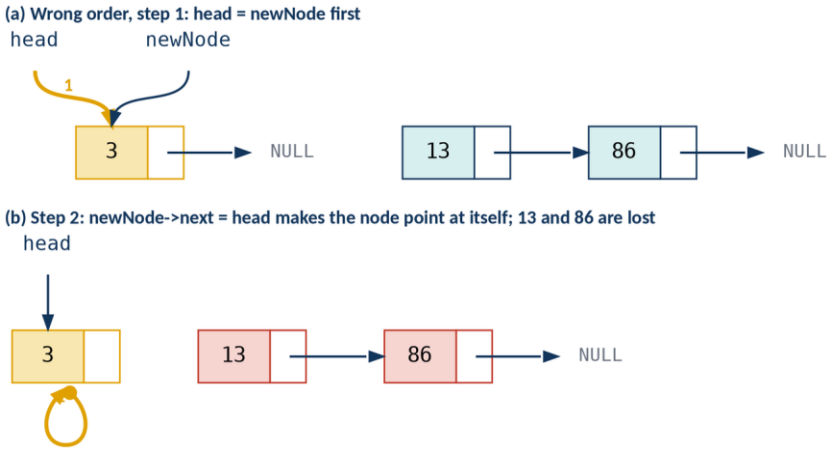


Figure 7.17 Front insertion with its two assignments in the wrong order: (a) `head` moves first, (b) the new node then points at itself and the old nodes 13 and 86 can never be reached again.

Because these bugs hide, we check for them with tools that watch every allocation and every memory access while the program runs. **Listing 7.12** is a small program containing each of the four bugs, selected by a command-line argument; with no argument it runs correctly.

Listing 7.12 `memory_bugs.cpp`: a correct run and four deliberate memory bugs (`leak`, `uaf`, `dangle`, `double`).

```cpp
// memory_bugs.cpp -- four classic linked-list memory bugs, one per argument.
//   ./memory_bugs        correct program
//   ./memory_bugs leak   forgets to free the list
//   ./memory_bugs uaf    keeps a pointer into the list after clearing it
//   ./memory_bugs dangle deletes the last node but leaves a pointer to it
//   ./memory_bugs double frees the same nodes twice
#include <cstdio>
#include <cstring>

struct Node {
    int data;
    Node *next;
};

Node *build() {                          // 1 -> 2 -> 3 -> NULL
    Node *head = nullptr;
    for (int x = 3; x >= 1; --x) head = new Node{x, head};
    return head;
}

void show(const Node *head) {
    for (; head != nullptr; head = head->next) std::printf("%d ", head->data);
    std::printf("\n");
}

void clearRight(Node *&head) {
    while (head != nullptr) {
        Node *del = head;
        head = head->next;               // step first ...
        delete del;                      // ... then free
    }
}

void backDeletionWrong(Node *head) {     // assumes at least two nodes
    Node *aux = head;
    while (aux->next->next != nullptr) aux = aux->next;
    delete aux->next;                    // BUG: aux->next is not set to nullptr
}
```

Listing 7.12 (Continued) *memory_bugs.cpp: a correct run and four deliberate memory bugs (leak, uaf, dangle, double).*

```
int main(int argc, char *argv[]) {
    const char *mode = argc > 1 ? argv[1] : "ok";
    Node *head = build();
    show(head);
    if (std::strcmp(mode, "leak") == 0) {
        return 0;                          // BUG: the three nodes are never freed
    } else if (std::strcmp(mode, "uaf") == 0) {
        Node *second = head->next;         // a second way into the list
        clearRight(head);                  // head is nullptr now, but second is not
        show(second);                      // BUG: reads freed nodes
    } else if (std::strcmp(mode, "dangle") == 0) {
        backDeletionWrong(head);
        show(head);                        // walks into the freed node
        clearRight(head);
    } else if (std::strcmp(mode, "double") == 0) {
        Node *copy = head;                 // copies the pointer, not the nodes
        clearRight(copy);                  // frees all three nodes
        clearRight(head);                  // BUG: frees them again
    } else {
        clearRight(head);
    }
    std::printf("done (%s)\n", mode);
    return 0;
}
```

7.11.2 Valgrind

Valgrind runs an unmodified program on a simulated processor and checks every memory access and every allocation; its default tool, Memcheck, reports invalid reads and writes, uses of freed memory, double frees and leaks [8]. It runs on Linux (and in WSL on Windows); the program runs 20 to 50 times more slowly under it, which does not matter for tests. The current release as of September 2026 is Valgrind 3.27.1, published on 20 May 2026 [8]; the runs below used the 3.22 release packaged with Ubuntu 24.04, whose output format is the same. Compile with `-g` so that reports name source lines, and with `-O0` so that the lines match the source exactly:

```
g++ -std=c++17 -g -O0 -Wall -Wextra -o memory_bugs memory_bugs.cpp
valgrind --leak-check=full ./memory_bugs leak
```

```
==1776== Memcheck, a memory error detector
==1776== Command: ./memory_bugs leak
==1776==
1 2 3
==1776==
==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==
==1776== 48 (16 direct, 32 indirect) bytes in 1 blocks are definitely lost in loss record 2
of 2
==1776==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1776==    by 0x10920F: build() (memory_bugs.cpp:17)
==1776==    by 0x109369: main (memory_bugs.cpp:42)
==1776==
==1776== LEAK SUMMARY:
==1776==    definitely lost: 16 bytes in 1 blocks
==1776==    indirectly lost: 32 bytes in 2 blocks
==1776==      possibly lost: 0 bytes in 0 blocks
==1776==    still reachable: 0 bytes in 0 blocks
==1776==         suppressed: 0 bytes in 0 blocks
==1776==
==1776== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Read the report from the bottom up. `LEAK SUMMARY` says 16 bytes are *definitely lost* and 32 *indirectly lost*. The distinction is instructive. Only the first node is "definitely" lost: it was pointed at by `head`, and `head` disappeared when `main` returned. The other two nodes are "indirectly" lost: pointers to them still exist, but only inside the first node, which is itself lost. So a leaked list shows up as one definite leak the size of a node plus indirect leaks for the rest of the chain, and fixing the definite leak (calling `clear`) fixes all of it. The record above `LEAK SUMMARY` says where the lost memory was allocated: by `operator new`, called from `build()` at line 17 of `memory_bugs.cpp`, called from `main` at line 42. Notice also `total heap usage: 5 allocs, 2 frees`: three of the five allocations were our nodes, and the other two belong to the C++ run-time library, which frees them itself.

The use-after-free run is more alarming, because the program appears to work:

```
valgrind --leak-check=full ./memory_bugs uaf
```

```
==1786== Memcheck, a memory error detector
==1786== Command: ./memory_bugs uaf
==1786==
==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==    at 0x484A61D: operator delete(void*, unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1786==    by 0x1092C3: clearRight(Node*&) (memory_bugs.cpp:30)
==1786==    by 0x1093CF: main (memory_bugs.cpp:48)
==1786==  Block was alloc'd at
==1786==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1786==    by 0x10920F: build() (memory_bugs.cpp:17)
==1786==    by 0x109369: main (memory_bugs.cpp:42)
==1786==
1 2 3
2 3
done (uaf)
==1786==
==1786== HEAP SUMMARY:
==1786==     in use at exit: 0 bytes in 0 blocks
==1786==   total heap usage: 5 allocs, 5 frees, 77,872 bytes allocated
==1786==
==1786== All heap blocks were freed -- no leaks are possible
==1786==
==1786== ERROR SUMMARY: 4 errors from 2 contexts (suppressed: 0 from 0)
```

The program printed `2 3`, the right values, from nodes it had already deleted. Valgrind reports each bad access with three stack traces: where the invalid read happened (`show`, line 22, called from `main` at line 49), where the block had been freed (`clearRight`, line 30, called from line 48), and where it had been allocated (`build`, line 17). Together they tell the whole story: the nodes were created by `build`, released by `clearRight(head)`, and read afterwards through the stale pointer `second`. (The run found two error contexts: the 4-byte read of `data` shown above and an 8-byte read of `next` at the same line, whose report has the same shape and is omitted here.) When the same program is compiled with `-O2` and run *without* Valgrind on the machine used for this book, it crashes with a segmentation fault; with other settings it can print garbage or appear to work. That unpredictability is exactly why the tool is needed.

Finally, the proof for a correct program. **Listing 7.3** under Valgrind ends with:

```
==3636==
==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 -- no leaks are possible` and `ERROR SUMMARY: 0 errors` together are the certificate we want; the six allocations are the list's four nodes and the two belonging to the run-time library. Every C++ program of this chapter was run under Valgrind with `--leak-check=full` and produced both lines.

7.11.3 AddressSanitizer

AddressSanitizer (ASan) takes the opposite approach: instead of simulating the processor, the compiler inserts checks into the program itself, marking the bytes around every allocation and

every freed block as poisoned in a *shadow memory* [9]. The resulting program runs only about twice as slowly as normal, works on Linux, macOS and Windows (with GCC, Clang and Microsoft's compiler), and reports the first bad access immediately. Its companion LeakSanitizer reports leaks when the program exits, and `-fsanitize=undefined` adds checks for undefined behavior such as signed overflow. The option `strip_path_prefix` shortens the file names in the report.

```
g++ -std=c++17 -g -O1 -Wall -Wextra -fsanitize=address,undefined \
    -fno-omit-frame-pointer -o memory_bugs memory_bugs.cpp
ASAN_OPTIONS=strip_path_prefix=$PWD/ ./memory_bugs dangle
```

```
=================================================================
==1805==ERROR: AddressSanitizer: heap-use-after-free on address 0x502000000010 at pc
0x55d758b734e0 bp 0x7fffe07441a0 sp 0x7fffe0744190
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:
    #0 0x7f1fd44ff5e8 in operator delete(void*, unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:164
    #1 0x55d758b73831 in backDeletionWrong(Node*) memory_bugs.cpp:37
    #2 0x55d758b73c8d in main memory_bugs.cpp:51

previously allocated by thread T0 here:
    #0 0x7f1fd44fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #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*)
```

(The frames inside the C library's start-up code, which follow `main` in each trace, and the shadow-memory map that ASan prints after the summary are omitted.) This is the dangling-link bug of `backDeletionWrong`: the last node was deleted at line 37, but the node before it still pointed at it, so `show` at line 22 read its `data` field. ASan reports the kind of error (`heap-use-after-free`), the size of the access (`READ of size 4`, an `int`), the position of the address inside the freed block (`0 bytes inside of 16-byte region`, the `data` field at offset 0 of a 16-byte node), and the same three stack traces as Valgrind, then stops the program.

The double-free run shows how a double free is usually caught even earlier:

```
ASAN_OPTIONS=strip_path_prefix=$PWD/ ./memory_bugs double
```

```
=================================================================
==1806==ERROR: AddressSanitizer: heap-use-after-free on address 0x502000000058 at pc
0x55c58ece3651 bp 0x7ffce90d0d90 sp 0x7ffce90d0d80
READ of size 8 at 0x502000000058 thread T0
    #0 0x55c58ece3650 in clearRight(Node*&) memory_bugs.cpp:29
    #1 0x55c58ece3a98 in main memory_bugs.cpp:57

0x502000000058 is located 8 bytes inside of 16-byte region [0x502000000050,0x502000000060)
freed by thread T0 here:
    #0 0x7f1f660ff5e8 in operator delete(void*, unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:164
    #1 0x55c58ece35bf in clearRight(Node*&) memory_bugs.cpp:30
    #2 0x55c58ece3a8e in main memory_bugs.cpp:56

previously allocated by thread T0 here:
    #0 0x7f1f660fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x55c58ece33d7 in build() memory_bugs.cpp:17
    #2 0x55c58ece39ba in main memory_bugs.cpp:42

SUMMARY: AddressSanitizer: heap-use-after-free memory_bugs.cpp:29 in clearRight(Node*&)
```

The second `clearRight(head)` (line 57) begins by reading `head->next` at line 29 from the first node, which the first `clearRight(copy)` (line 56) had already released, so ASan stops at that read, `8 bytes inside` the node, which is its `next` field, before the second `delete` is even attempted. Without ASan the same run is typically aborted by the C library with the message `free(): double free detected in tcache 2`, which says what happened but not where. The leak, finally, is reported by LeakSanitizer when the program exits:

```
ASAN_OPTIONS=strip_path_prefix=$PWD/ ./memory_bugs leak
```

```
=================================================================
==1810==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 16 byte(s) in 1 object(s) allocated from:
    #0 0x7f4b9c8fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x55ed654b03d7 in build() memory_bugs.cpp:17
    #2 0x55ed654b09ba in main memory_bugs.cpp:42

Indirect leak of 32 byte(s) in 2 object(s) allocated from:
    #0 0x7f4b9c8fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x55ed654b03d7 in build() memory_bugs.cpp:17
    #2 0x55ed654b09ba in main memory_bugs.cpp:42

SUMMARY: AddressSanitizer: 48 byte(s) leaked in 3 allocation(s).
```

The “direct” and “indirect” leaks correspond exactly to Valgrind’s “definitely” and “indirectly” lost. Every C++ program of this chapter was also built with `-fsanitize=address,undefined` and run: all of them printed their normal output and nothing else.

7.11.4 A checklist for every linked-list function

The four bugs have four matching habits, and a fifth habit catches what the others miss.

1. **Save, then move, then delete.** Before deleting a node, save its address in `del`, move every pointer that leads to it (the `head`, the predecessor’s `next`, `tail`), and delete last.
2. **Leave no pointer at a freed node.** After a deletion, ask which pointers held that node’s address: `head`, `tail`, a predecessor’s `next`, a saved helper. Each must be moved or set to `nullptr`.

3. **Every `new` has exactly one owner that deletes it.** In a list, the owner of a node is the list; release all nodes in exactly one place (`clear`, `destroy`, or a destructor) and make sure copying a list either copies the nodes or is forbidden (Section 7.10.2).
4. **Check the four shapes.** Test every function on the empty list, a one-node list, and at the first, a middle, and the last node. Most list bugs live in exactly one of these cases.
5. **Run the tools.** Compile the tests with `-fsanitize=address,undefined`, or run them under `valgrind --leak-check=full`, every time. A test suite that passes but has never run under a memory checker proves much less than it seems to.

7.12 Linked Lists in the Standard Libraries

7.12.1 C++: `std::forward_list` and `std::list`

The C++ standard library provides both kinds of linked list [10]. `std::forward_list<T>`, added in C++11, is a singly linked list designed to have no overhead compared with a hand-written one: its object is essentially one pointer (to the first node) and each node holds the element and one `next` pointer. `std::list<T>` is a doubly linked list, the structure of Chapter 9. **Listing 7.13** uses both.

Listing 7.13 `std_lists.cpp`: the singly linked `std::forward_list` and the doubly linked `std::list`.

```
// std_lists.cpp -- the standard library's linked lists: std::forward_list
// (singly linked) and std::list (doubly linked).
#include <forward_list>
#include <iostream>
#include <iterator>
#include <list>

template <typename Seq>
void print(const char *label, const Seq &s) {
    std::cout << label;
    for (int x : s) std::cout << ' ' << x;
    std::cout << '\n';
}

int main() {
    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);
    f.sort();                                 // O(n log n): merge sort on nodes
    print("sort:                ", f);
    // forward_list has no size(): counting the nodes is a walk, O(n)
    std::cout << "length:               " << std::distance(f.begin(), f.end()) << '\n';

    std::list<int> d(f.begin(), f.end());     // doubly linked: both ends are O(1)
    d.push_back(7);
    d.push_front(0);
    d.pop_back();
    print("doubly linked:       ", d);
    std::cout << "front " << d.front() << ", back " << d.back()
              << ", size " << d.size() << '\n';
    return 0;
}
```

```
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
sort:                 2 3 4 5 9
length:               5
doubly linked:        0 2 3 4 5 9
front 0, back 9, size 6
```

`std::forward_list` offers exactly the operations that are cheap on a singly linked list, and it names them honestly. Insertion and deletion are `insert_after` and `erase_after`, because, as Section 7.6.3 showed, a singly linked list can change only the link *after* a known node; `before_begin()` provides the position before the first element so that the first element needs no special case, the dummy-node idea of Section 7.7. There is `push_front` but no `push_back`, and, surprisingly at first, no `size()`: keeping a count would cost an extra field and an update in every operation, which the design refuses, so the length must be counted with a walk (`std::distance`), in $\Theta(n)$ time. `remove`, `reverse` and `sort` are member functions because they can relink nodes instead of moving elements; `sort` is a merge sort, $O(n \lg n)$, which needs no random access.

`std::list` adds `push_back`, `pop_back`, `back()` and a constant-time `size()` (required since C++11), all possible because every node knows its predecessor. Both containers share an important guarantee that arrays lack: inserting or erasing elements never moves the other elements, so pointers and iterators to them remain valid. That guarantee, not speed, is often the real reason to choose a list in C++.

As of September 2026, the most recent GCC release series is GCC 16, whose first release, 16.1, came out in April 2026 and made C++20 the default language mode [11]. Nothing in this chapter depends on it: `std::forward_list` and `std::list` behave identically in C++17 and in every later standard.

7.12.2 Java: `java.util.LinkedList`

Java's `LinkedList<E>` is a doubly linked list that implements the `List` and `Deque` interfaces, and, since Java 21, `SequencedCollection`, which adds `addFirst`, `addLast`, `getFirst`, `getLast` and `reversed()` to every ordered collection [12]. It is not synchronized, so a `LinkedList` shared between threads must be protected by the caller. **Listing 7.14** performs the same operations as **Listing 7.13** and prints the same output.

Listing 7.14 `LibraryLists.java`: `java.util.LinkedList` used as a list and as a deque.

```
// LibraryLists.java -- Java's built-in linked list, java.util.LinkedList
// (doubly linked), used as a list and as a deque.
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class LibraryLists {
    static void print(String label, List<Integer> s) {
        StringBuilder sb = new StringBuilder(label);
        for (int x : s) sb.append(' ').append(x);
        System.out.println(sb);
    }

    public static void main(String[] args) {
        LinkedList<Integer> f = new LinkedList<>(List.of(3, 1, 4, 1, 5));
        print("start:                ", f);
        f.addFirst(9);                          // O(1)
        print("add 9 at the front:  ", f);
        f.add(1, 2);                            // walks to index 1, then links
        print("insert 2 after first:", f);
        f.removeIf(x -> x == 1);                // O(n)
        print("remove every 1:      ", f);
        Collections.reverse(f);                 // O(n)
        print("reverse:              ", f);
        Collections.sort(f);                    // copies to an array, sorts, copies back
        print("sort:                 ", f);
        System.out.println("length:               " + f.size());   // O(1): kept in a field

        LinkedList<Integer> d = new LinkedList<>(f);
        d.addLast(7);
        d.addFirst(0);
        d.removeLast();
        print("doubly linked:       ", d);
        System.out.println("front " + d.getFirst() + ", back " + d.getLast() + ", size " +
d.size());
    }
}
```

The Java library has no singly linked list class; `LinkedList` serves both roles, at the cost of a second pointer per node, and it stores references to boxed `Integer` objects rather than the numbers themselves. Its `size()` is $O(1)$ because the count is kept in a field. Note the comments on `add(1, 2)`, which must walk to index 1, and on `Collections.sort`, which copies the elements into an array, sorts the array and writes the elements back, because sorting through a list's sequential access would be slow. In practice `ArrayList` (for lists) and `ArrayDeque` (for stacks and queues) are faster than `LinkedList` for almost every workload, for the cache reasons of Section 7.3.2; `LinkedList` pays off mainly when elements are inserted or removed in the middle through an iterator that is already positioned there.

7.12.3 Python: why there is no built-in linked list, and `collections.deque`

Python's built-in `list` is not a linked list at all: it is a dynamic array of references, with $O(1)$ indexing and $O(1)$ amortized `append`, but $\Theta(n)$ `insert(0, x)` and `pop(0)`, which shift every element. Python has no built-in linked list class, and for a good reason. In CPython every object already carries a header of its own (a reference count and a type pointer), so a node object of the kind in **Listing 7.5** costs about 48 bytes even with `__slots__`, and every step `aux = aux.next` runs as interpreted bytecode. A pure-Python linked list is therefore slow and memory-hungry compared with the built-in list, which is implemented in C.

When a program needs what linked lists are good at, cheap insertion and removal at both ends, Python's answer is `collections.deque`. The documentation promises "approximately the same

$O(1)$ performance in either direction" for appends and pops at both ends, while indexed access "slows to $O(n)$ in the middle" [13]. Internally, CPython implements a deque as a *doubly linked list of fixed-length blocks*, each holding 64 references [14]. This hybrid keeps the linked list's promise that appends and pops never move other elements, while cutting the pointer overhead 64-fold and keeping neighboring elements in the same memory block for the cache. **Listing 7.15** performs the same operations once more.

Listing 7.15 `library_lists.py`: `collections.deque`, Python's closest built-in to a linked list.

```
# library_lists.py -- Python has no built-in linked list class;
# collections.deque (a doubly linked list of 64-slot blocks) is the closest.
from collections import deque


def show(label, s):
    print(label + "".join(f" {x}" for x in s))


def main():
    f = deque([3, 1, 4, 1, 5])
    show("start:                ", f)
    f.appendleft(9)                          # O(1)
    show("add 9 at the front:  ", f)
    f.insert(1, 2)                           # O(n) in general
    show("insert 2 after first:", f)
    f = deque(x for x in f if x != 1)        # deque has no remove-all: rebuild
    show("remove every 1:      ", f)
    f.reverse()                              # in place, O(n)
    show("reverse:              ", f)
    f = deque(sorted(f))                     # deque has no sort(): sorted() makes a list
    show("sort:                 ", f)
    print(f"length:                {len(f)}")   # O(1)

    d = deque(f)
    d.append(7)
    d.appendleft(0)
    d.pop()
    show("doubly linked:       ", d)
    print(f"front {d[0]}, back {d[-1]}, size {len(d)}")


if __name__ == "__main__":
    main()
```

A deque cannot do everything a list can: it has no `sort` method (the program builds a sorted deque from `sorted(f)`), and `insert(i, x)` in the middle is $O(n)$. Writing your own node class, as in this chapter, remains the right thing to do in Python when you are learning the structure, or when the nodes are part of a larger linked structure that no library provides, such as the trees and graphs of later chapters.

Table 7.5 compares the three libraries.

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

7.12.4 Measuring: array against list on a real machine

The costs of Section 7.3 predict two things: inserting at the front is $\Theta(n^2)$ in total for an array and $\Theta(n)$ for a list, while walking through all elements is $\Theta(n)$ for both. **Listing 7.16** tests both predictions. Part 1 inserts n values at the front of a `std::vector` and of a `std::forward_list`. Part 2 adds up two million values stored in an array, in a list whose nodes happen to lie side by side in memory, and in the *same* nodes linked in a random order, which is what a long-lived list looks like after many insertions and deletions. Each time is the best of five runs.

Listing 7.16 `bench.cpp`: front insertion and traversal timed for an array and a linked list.

```cpp
// bench.cpp -- array versus linked list on a real machine.
// Part 1: insert n values at the FRONT. Part 2: walk n values and add them up.
#include <algorithm>
#include <chrono>
#include <cstdio>
#include <forward_list>
#include <numeric>
#include <random>
#include <vector>

struct Node {
    long long data;
    Node *next;
};

template <typename F>
double ms(F f) {                          // best of 5 runs, in milliseconds
    double best = 1e300;
    for (int r = 0; r < 5; ++r) {
        auto t0 = std::chrono::steady_clock::now();
        f();
        auto t1 = std::chrono::steady_clock::now();
        best = std::min(best, std::chrono::duration<double, std::milli>(t1 - t0).count());
    }
    return best;
}

int main() {
    std::printf("Part 1: n insertions at the front\n");
    std::printf("%8s %14s %14s\n", "n", "vector (ms)", "list (ms)");
    for (int n : {20000, 40000, 80000, 160000}) {
        double tv = ms([&] {
            std::vector<int> v;
            for (int i = 0; i < n; ++i) v.insert(v.begin(), i);   // shifts everything
        });
        double tf = ms([&] {
            std::forward_list<int> f;
            for (int i = 0; i < n; ++i) f.push_front(i);          // relinks one pointer
        });
        std::printf("%8d %14.2f %14.2f\n", n, tv, tf);
    }

    const int n = 2000000;
    std::vector<long long> v(n);
    std::iota(v.begin(), v.end(), 0);
```

Listing 7.16 (Continued) *bench.cpp: front insertion and traversal timed for an array and a linked list.*

```
    std::vector<Node> pool(n);            // all nodes in one block of memory
    std::vector<int> order(n);
    std::iota(order.begin(), order.end(), 0);
    auto link = [&](const std::vector<int> &ord) {   // chain the nodes in this order
        for (int i = 0; i < n; ++i) {
            pool[ord[i]].data = i;
            pool[ord[i]].next = (i + 1 < n) ? &pool[ord[i + 1]] : nullptr;
        }
        return &pool[ord[0]];
    };
    auto walk = [](const Node *h) {
        long long s = 0;
        for (; h != nullptr; h = h->next) s += h->data;
        return s;
    };

    long long s1 = 0, s2 = 0, s3 = 0;
    double ta = ms([&] { s1 = std::accumulate(v.begin(), v.end(), 0LL); });
    Node *inOrder = link(order);
    double tb = ms([&] { s2 = walk(inOrder); });
    std::shuffle(order.begin(), order.end(), std::mt19937(2026));
    Node *scattered = link(order);
    double tc = ms([&] { s3 = walk(scattered); });

    std::printf("\nPart 2: sum of %d values (all sums equal: %s)\n", n,
                (s1 == s2 && s2 == s3) ? "yes" : "no");
    std::printf("  array                          %8.2f ms\n", ta);
    std::printf("  list, nodes side by side       %8.2f ms\n", tb);
    std::printf("  list, nodes scattered          %8.2f ms\n", tc);
    return 0;
}
```

```
Part 1: n insertions at the front
       n    vector (ms)      list (ms)
   20000          17.71           0.27
   40000          57.58           0.62
   80000         241.76           1.12
  160000         962.67           1.95

Part 2: sum of 2000000 values (all sums equal: yes)
  array                          0.68 ms
  list, nodes side by side       2.91 ms
  list, nodes scattered         87.82 ms
```

Figure 7.18 plots the run. The exact numbers depend on the machine (this run used a shared 2.1 GHz Intel Xeon server) and vary by tens of percent from run to run, but their shape does not.

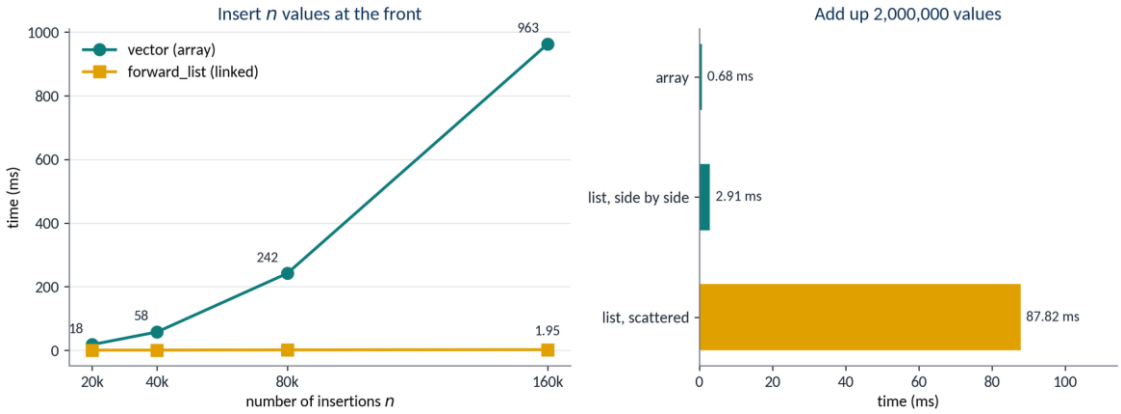


Figure 7.18 The measurements of **Listing 7.16**: front insertion is quadratic for the array and linear for the list (left); summing two million values is fastest in the array and about a hundred times slower in a list whose nodes are scattered (right).

- **Front insertion.** Each doubling of n multiplies the vector's time by about 4 (18, 58, 242, 963 ms), the signature of $\Theta(n^2)$ from Equation (7.4), and the list's time by about 2 (0.27, 0.62, 1.12, 1.95 ms), the signature of $\Theta(n)$. At $n = 160{,}000$ the list is roughly 500 times faster.
- **Traversal.** All three walks are $\Theta(n)$, yet the array takes 0.68 ms, the side-by-side list 2.91 ms, and the scattered list 87.82 ms, about 130 times the array. The only difference between the last two is where the nodes are in memory: exactly the cache effect of Section 7.3.2. The O-notation, which ignores constant factors, cannot see it.

The Java and Python versions in the code folder (`Bench.java`, `bench.py`) show the same shapes. In one run on the same machine, Java's `ArrayList.add(0, x)` took 11.5, 57.9, 232 and 989 ms for the four sizes against at most 1 ms for `LinkedList.addFirst`, and traversing two million shuffled hand-made nodes took about 100 ms against 0.7 ms for a `long[]`. In Python, `list.insert(0, x)` took 58, 234, 961 and 4,095 ms, while `deque.appendleft` took 0.4 to 4.8 ms and a hand-made node list 3.3 to 32 ms.

These measurements summarize the practical wisdom of the field, which the designer of C++ has demonstrated in the same way [15]: for most workloads an array (`std::vector`, `ArrayList`, Python's `list`) is faster than a linked list, even for insertions in the middle, because finding the position costs a walk in the list and the walk is slow. Linked lists win when insertions and deletions happen at positions the program already holds (the front, the back, a node found earlier), when elements must never move, when elements are large, or when the structure is itself a chain of references, as in the stacks, queues and circular schedules of this chapter.

7.13 Profitina in Practice

Profitina is an AI visibility (GEO) platform: it measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds [16]. Its server is a Python application built with FastAPI, and several of its features have the shape this chapter studied. We describe the shape of each problem and the class of structure that fits it; the actual implementation choices inside the product are its own.

- **A feed that grows at the front.** Profitina's real-time community delivers new posts to connected browsers with Server-Sent Events, and a new post appears at once. In a feed shown newest first, every arriving post goes to the *top*. The operation such a feed performs most is exactly `frontInsertion`, and the operation a reader performs is a traversal from the newest item. A structure with $O(1)$ insertion at the front and cheap removal of the oldest items (to keep

only the most recent k in memory) is the natural fit; in Python that is a `collections.deque` with a maximum length, which discards from the other end automatically when full.

- **A queue of re-scans.** The historical analytics feature re-scans each brand automatically at a cadence that depends on its subscription tier. Jobs that are due wait in a first-in, first-out queue until a worker takes them, the linked queue of Section 7.10: `enqueue` at the rear when a job becomes due, `dequeue` at the front when a worker is free, both $O(1)$ and never "full". (When jobs have different urgencies, a priority queue, built on the heap of Chapter 14, replaces the plain queue.)
- **Taking turns fairly.** When a small number of workers must share a large set of brands, round-robin service, the circular list of Section 7.9, gives every brand a turn before any brand gets a second one, so a brand with a large site cannot starve the others.
- **A history that only grows.** A brand's weekly score trend is an append-only sequence read from the oldest week to the newest when the dashboard draws its chart or exports it as CSV: `backInsertion` with a tail pointer and a forward traversal, the head-and-tail list of Section 7.8. Browsing backward from any week, one week at a time, needs a link to the previous node, the doubly linked list of Chapter 9.

The same measurements that closed Section 7.12 apply to the product. In Python a hand-written node list is rarely the fastest option; the built-in `list` and `collections.deque`, implemented in C, usually are. The lesson of this chapter transfers intact, though: identify which end of the sequence the operations touch, and choose the structure whose cheap operations are exactly those.

7.14 Common Mistakes, and How to Avoid Them

Moving `head` to walk the list. A loop such as `while (head != nullptr) { ...; head = head->next; }` in `show` loses the whole list. Walk with `aux`; move `head` only when the first node changes (Section 7.5).

Using the wrong loop condition. `while (aux != nullptr)` ends *after* the last node (use it to visit every node); `while (aux->next != nullptr)` ends *on* the last node (use it to append); `while (aux->next->next != nullptr)` ends on the second-to-last node (use it to delete the last node). Using the first where the second is needed makes `aux` NULL, and the next `aux->next` crashes.

Dereferencing NULL. Every `p->field` requires `p != nullptr`. The empty list and the one-node list are where this goes wrong: `backInsertion`'s walk and `backDeletion`'s `aux->next->next` both need a guard for them (Sections 7.5.3 and 7.5.6). In compound conditions, put the null test first: `aux != nullptr && aux->data != key`.

Assignments in the wrong order. In `frontInsertion` and `insertAfter`, link the new node to its successor *before* redirecting the predecessor to it; otherwise the new node points at itself and the rest of the list is lost (**Figure 7.17**).

Deleting before moving. `delete head; head = head->next;` reads a freed node. Save, move, then delete (Section 7.5.5).

Forgetting to cut the last link. After deleting the last node, the new last node's `next` must be set to `nullptr`, and `tail` must move. Otherwise the list ends in freed memory (**Listing 7.12**, mode `dangle`).

Forgetting the second pointer. In a head-and-tail list, inserting into an empty list must set `tail`, and deleting the only node must reset `tail`. In a linked queue, `dequeue` of the last node must reset `rear_` (Section 7.10.2).

Using a NULL test to stop a circular traversal. No `next` field in a ring is NULL, so `while (aux != nullptr)` never ends. Use a `do-while` that stops when it returns to the start, or count the nodes (Section 7.9.2).

Leaking the list. Every node created with `new` must be deleted exactly once. Call `clear` or `destroy` before the last pointer to the list disappears, or better, let a destructor do it, and forbid or implement copying (Section 7.10.2).

Believing a correct output. A program that reads freed memory can print the right answer. Run every linked-list program under Valgrind or AddressSanitizer (Section 7.11).

Recursing over a very long list. `lengthRec` and `reverseRec` use one stack frame per node; for lists of millions of nodes use the iterative versions (Section 7.6.2).

Choosing a list for random access. Indexing a list is $\Theta(n)$ (Equation (7.3)), and even a loop over indexes, `for (i = 0; i < n; i++) get(i)`, becomes $\Theta(n^2)$. If a program mostly reads by position, it wants an array.

7.15 Chapter Summary

- A **linked list** stores each element in a **node** that also holds a pointer to the next node; the list is reached through an external pointer, `head`, and the last node's `next` is NULL. The idea dates from 1955 and 1956 (Newell, Shaw and Simon's IPL), was used by Yngve's COMIT, and became the basis of LISP.
- **Array versus list.** An array gives $O(1)$ access by position (Equation (7.1)) but $\Theta(n)$ insertion and deletion at the front or in the middle (Equation (7.4)); a list gives $O(1)$ insertion and deletion at any position it already holds, but $\Theta(n)$ access by position (Equation (7.2), Equation (7.3)), one extra pointer per element (Equation (7.5)), and poor cache behavior when its nodes are scattered.
- **Head-pointer list.** `init`, `isEmpty`, `frontInsertion`, `frontDeletion` and `insertAfter` are $O(1)$; `backInsertion`, `backDeletion`, `show`, `search`, `insertSorted`, `deleteKey`, `length`, `reverse` and `clear` are $\Theta(n)$ in the worst case (**Table 7.2**, **Table 7.3**).
- **Two rules of order.** When inserting, link the new node to its successor before linking its predecessor to it. When deleting, save the node, move every pointer past it, and delete it last.
- **Reversal** turns every arrow around in one pass with three pointers ($\Theta(n)$ time, $O(1)$ memory), or recursively by reversing the rest and hanging the first node on its end (Equation (7.7), $\Theta(n)$ time and stack).
- A **dummy head node**, or in C++ a **pointer to the link**, removes the special cases for the first node.
- A **tail pointer** makes back insertion $O(1)$ but not back deletion, which needs the predecessor of the last node; that requires the doubly linked list of Chapter 9.
- A **circular list** reached through `last` gives $O(1)$ access to both ends with one pointer; its traversals stop when they return to the start, and it models round-robin service naturally.
- A **linked stack** pushes and pops at the front; a **linked queue** enqueues at the tail and dequeues at the head; all four operations are $O(1)$ and never report "full". In C++, a destructor releases the nodes and copying must be forbidden or implemented (the rule of three).
- **Memory safety** must be checked, not assumed: Valgrind and AddressSanitizer find leaks, uses after free, dangling links and double frees, with the exact source lines.

- **Libraries.** C++ has `std::forward_list` (singly linked, `insert_after`, no `size()`) and `std::list`; Java has the doubly linked `LinkedList`; Python has no built-in linked list, and `collections.deque`, a doubly linked list of 64-slot blocks, covers the use cases of one. Measured on a real machine, arrays win at traversal by up to two orders of magnitude, and lists win at front insertion by the same margin.

7.16 Exercises

7.1 Starting from an empty head-pointer list, perform `frontInsertion(5)`, `backInsertion(8)`, `frontInsertion(2)`, `backInsertion(6)`, `frontDeletion()`, `backDeletion()`, `frontInsertion(9)`. Draw the list after every operation and give the resulting list.

7.2 For a list of n nodes, give the running time of: (a) inserting at the back with only `head`; (b) inserting at the back with `head` and `tail`; (c) deleting at the back with `head` and `tail`; (d) deleting at the front of a circular list reached through `last`; (e) building a list of n values by `n` calls of `backInsertion` with only `head`. Justify each answer in one sentence, and for (e) give the exact number of `aux` moves.

7.3 Write `int countKey(const Node *head, int key)`, which returns how many nodes hold `key`. What is its running time?

7.4 Write `Node *nth(Node *head, int k)`, which returns the node at index k (counting from 0), or `nullptr` if the list has fewer than $k + 1$ nodes. How many steps does it take, by Equation (7.2)?

7.5 Write `Node *middle(Node *head)`, which returns the middle node of a non-empty list in *one* pass and without knowing the length. For an even length, return the second of the two middle nodes. (Hint: move two pointers, one twice as fast as the other.)

7.6 A bug can make a list circular by accident, and then every traversal of Section 7.5 loops forever. Write `bool hasCycle(const Node *head)`, which decides whether following `next` from `head` ever returns to a node already visited, in $O(n)$ time and $O(1)$ extra memory, without modifying the list.

7.7 Write `Node *mergeSorted(Node *a, Node *b)`, which merges two ascending lists into one ascending list by relinking their nodes, without creating or deleting any node. What is its running time for lists of lengths m and n?

7.8 Write `int dedupSorted(Node *head)`, which deletes repeated values from an ascending list so that each value appears once, and returns the number of nodes deleted.

7.9 Write `bool insertBefore(Node *&head, int key, int x)`, which inserts a new node holding `x` immediately *before* the first node holding `key`, and returns `false` (changing nothing) if `key` is absent. Make sure it works when `key` is in the first node.

7.10 A common textbook version of front insertion is

```
void frontInsertion(int newData) {
    TNode *newNode = new TNode;
    newNode->data = newData;
    newNode->next = nullptr;
    if (isEmpty()) {
        head = newNode;
        head->next = nullptr;
    } else {
        newNode->next = head;
        head = newNode;
    }
}
```

Show that the `if` branch is unnecessary, that is, that the two lines of the `else` branch alone are correct for the empty list too. Then explain why the same simplification is *not* possible in `backInsertion`.

7.11 In the *Josephus problem*, n people numbered 1 to n stand in a circle; starting from person 1, every k-th person leaves, and the circle closes up, until one person remains. Solve it with a circular singly linked list: print the order in which people leave and the survivor for $n = 7$, $k = 3$. What is the running time for general n and k?

7.12 In `LinkedQueue` of **Listing 7.11**, make two changes: delete the line `if (front_ == nullptr) rear_ = nullptr;` from `dequeue`, and let `isEmpty` test `rear_ == nullptr` instead of `front_ == nullptr` (as the head-and-tail list of Section 7.8 tests `tail`). Describe a sequence of three operations after which the program uses freed memory, say which of the four bugs of Section 7.11.1 it is, and say what AddressSanitizer would report. Why does the original class survive the first change alone, and why is it still wrong to omit the line?

7.13 `reverseRec` and `reverseIter` both run in $\Theta(n)$ time. Explain why `reverseIter` should be preferred for a list of ten million nodes. Estimate the stack memory `reverseRec` would need if each call frame takes 48 bytes.

7.14 For each of the following Pustaka Ceria tasks, say whether an array (or `std::vector`) or a linked list is the better choice, and why, in terms of the costs of **Table 7.1**: (a) the catalog of 50,000 books, looked up by position after sorting; (b) the queue of readers waiting for the single reading desk, which changes every few minutes; (c) the undo history of a librarian's edits, where only the most recent edit is ever undone; (d) the daily list of returned books, printed once in order of return at closing time.

7.17 Solutions to the Exercises

7.1 After each operation the list is: `frontInsertion(5)`: 5. `backInsertion(8)`: 5, 8. `frontInsertion(2)`: 2, 5, 8. `backInsertion(6)`: 2, 5, 8, 6. `frontDeletion()`: 5, 8, 6. `backDeletion()`: 5, 8. `frontInsertion(9)`: 9, 5, 8. The final list is 9, 5, 8.

7.2 (a) $\Theta(n)$: `aux` must walk from `head` to the last node, $n - 1$ moves. (b) $O(1)$: `tail` gives the last node directly and two pointer assignments attach the new one. (c) $\Theta(n)$: the node before `tail` must become the new last node, and it can only be found by walking from `head` (Section 7.8.2). (d) $O(1)$: the first node is `last->next`, and unlinking it changes one pointer. (e) The i-th call (counting from $i = 0$) finds a list of i nodes; it makes no move when $i = 0$ (the empty-list branch) and $i - 1$ moves otherwise, so the total is $0 + 0 + 1 + 2 + \cdots + (n - 2) = \frac{(n-2)(n-1)}{2}$ moves, which is $\Theta(n^2)$.

7.3 The solution program, **Listing 7.17**, contains `countKey`: a traversal that increments a counter whenever `h->data == key`. It must visit every node, because any node may hold the key, so it runs in $\Theta(n)$ time in every case.

Listing 7.17 `exercises.cpp`: solutions to Exercises 7.3 to 7.9 and 7.11.

```
// exercises.cpp -- solutions to the programming exercises of Chapter 7.
#include <cstdio>

struct Node {
    int data;
    Node *next;
};

Node *fromArray(const int *a, int n) {        // builds a[0] -> a[1] -> ... -> NULL
    Node *head = nullptr;
    for (int i = n - 1; i >= 0; --i) head = new Node{a[i], head};
    return head;
}

void show(const char *label, const Node *h) {
    std::printf("%-24s", label);
    for (; h != nullptr; h = h->next) std::printf("%d -> ", h->data);
    std::printf("NULL\n");
}

void destroy(Node *&head) {
    while (head != nullptr) {
        Node *del = head;
        head = head->next;
        delete del;
    }
}

// Exercise 7.3: how many nodes hold key?
int countKey(const Node *h, int key) {
    int c = 0;
    for (; h != nullptr; h = h->next) if (h->data == key) ++c;
    return c;
}

// Exercise 7.4: the node at index k (0-based), or nullptr if the list is shorter.
Node *nth(Node *h, int k) {
    while (h != nullptr && k > 0) {
        h = h->next;
        --k;
    }
    return (k == 0) ? h : nullptr;
}
```

Listing 7.17 (Continued) *exercises.cpp: solutions to Exercises 7.3 to 7.9 and 7.11.*

```
// Exercise 7.5: the middle node in one pass (the second middle if the length is even).
Node *middle(Node *h) {
    Node *slow = h, *fast = h;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;              // one step
        fast = fast->next->next;        // two steps
    }
    return slow;
}

// Exercise 7.6: does following next ever come back to a node already visited?
bool hasCycle(const Node *h) {
    const Node *slow = h, *fast = h;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return true;  // the fast runner lapped the slow one
    }
    return false;                       // the fast runner fell off the end
}

// Exercise 7.7: merge two ascending lists by relinking; no node is created or freed.
Node *mergeSorted(Node *a, Node *b) {
    Node dummy{0, nullptr};             // a dummy node on the stack
    Node *tail = &dummy;
    while (a != nullptr && b != nullptr) {
        Node *&smaller = (a->data <= b->data) ? a : b;
        tail->next = smaller;           // append the smaller front node
        tail = smaller;
        smaller = smaller->next;        // advance a or b
    }
    tail->next = (a != nullptr) ? a : b;   // whatever is left is already sorted
    return dummy.next;
}

// Exercise 7.8: remove repeated values from an ascending list.
int dedupSorted(Node *h) {
    int removed = 0;
    while (h != nullptr && h->next != nullptr) {
        if (h->next->data == h->data) {
            Node *del = h->next;
            h->next = del->next;
            delete del;
            ++removed;
        } else {
            h = h->next;
```

Listing 7.17 (Continued) *exercises.cpp: solutions to Exercises 7.3 to 7.9 and 7.11.*

```
        }
    }
    return removed;
}

// Exercise 7.9: insert x before the first node holding key; false if key is absent.
bool insertBefore(Node *&head, int key, int x) {
    Node **link = &head;
    while (*link != nullptr && (*link)->data != key) link = &(*link)->next;
    if (*link == nullptr) return false;
    *link = new Node{x, *link};
    return true;
}

// Exercise 7.11: Josephus with a circular list. n people, every k-th leaves.
int josephus(int n, int k) {
    Node *last = nullptr;
    for (int i = 1; i <= n; ++i) {      // build the circle 1, 2, ..., n
        Node *p = new Node{i, nullptr};
        if (last == nullptr) p->next = p;
        else { p->next = last->next; last->next = p; }
        last = p;
    }
    std::printf("Josephus n=%d k=%d, leaving order:", n, k);
    Node *prev = last;
    while (prev->next != prev) {        // more than one person left
        for (int s = 1; s < k; ++s) prev = prev->next;
        Node *del = prev->next;         // the k-th person
        std::printf(" %d", del->data);
        prev->next = del->next;
        delete del;
    }
    int survivor = prev->data;
    delete prev;
    std::printf("\n");
    return survivor;
}
```

Listing 7.17 (Continued) *exercises.cpp: solutions to Exercises 7.3 to 7.9 and 7.11.*

```
int main() {
    int a[] = {4, 7, 4, 1, 4, 9};
    Node *h = fromArray(a, 6);
    show("list:", h);
    std::printf("countKey(4) = %d, countKey(5) = %d\n", countKey(h, 4), countKey(h, 5));
    Node *p = nth(h, 3);
    std::printf("nth(3) = %d, nth(6) is %s\n", p->data, nth(h, 6) ? "a node" : "nullptr");
    std::printf("middle = %d\n", middle(h)->data);

    Node *last = nth(h, 5);
    last->next = nth(h, 2);             // make a cycle on purpose
    std::printf("hasCycle after linking 9 back to 4: %s\n", hasCycle(h) ? "true" :
"false");
    last->next = nullptr;               // undo it before anything walks the list
    std::printf("hasCycle after undoing it:          %s\n", hasCycle(h) ? "true" :
"false");
    destroy(h);

    int x[] = {1, 4, 4, 9}, y[] = {2, 4, 10};
    Node *m = mergeSorted(fromArray(x, 4), fromArray(y, 3));
    show("mergeSorted:", m);
    int r = dedupSorted(m);
    std::printf("dedupSorted removed %d\n", r);
    show("after dedup:", m);
    insertBefore(m, 1, 0);              // before the first node
    insertBefore(m, 9, 7);              // in the middle
    bool ok = insertBefore(m, 99, 5);   // key absent
    show("insertBefore x3:", m);
    std::printf("insertBefore(99, 5) returned %s\n", ok ? "true" : "false");
    destroy(m);

    std::printf("survivor = %d\n", josephus(7, 3));
    return 0;
}
```

```
list:                  4 -> 7 -> 4 -> 1 -> 4 -> 9 -> NULL
countKey(4) = 3, countKey(5) = 0
nth(3) = 1, nth(6) is nullptr
middle = 1
hasCycle after linking 9 back to 4: true
hasCycle after undoing it:          false
mergeSorted:           1 -> 2 -> 4 -> 4 -> 4 -> 9 -> 10 -> NULL
dedupSorted removed 2
after dedup:           1 -> 2 -> 4 -> 9 -> 10 -> NULL
insertBefore x3:       0 -> 1 -> 2 -> 4 -> 7 -> 9 -> 10 -> NULL
insertBefore(99, 5) returned false
Josephus n=7 k=3, leaving order: 3 6 2 7 5 1
survivor = 4
```

On the list 4, 7, 4, 1, 4, 9 the output confirms that 4 occurs three times and 5 never.

7.4 `nth` in **Listing 7.17** advances `h` and decrements `k` together until either `k` reaches 0 or the list ends. If `k` reached 0, `h` is the node at the requested index; when k equals the length, that "node" is the NULL after the last node, so the function correctly returns `nullptr`. If the list ended first, `k` is still positive and the function returns `nullptr`. By Equation (7.2) it makes $\min(k, n)$ steps, so it is $O(k)$ and $\Theta(n)$ in the worst case. In the output, index 3 holds 1 and index 6 does not exist in a list of 6 nodes.

7.5 `middle` moves `slow` one node and `fast` two nodes per iteration, and stops when `fast` cannot move two more. When `fast` has made $2t$ moves, `slow` has made t, so when `fast` reaches the end `slow` is halfway. For the six-node list 4, 7, 4, 1, 4, 9 the loop runs three times (`fast` visits indexes 2, 4 and then NULL), so `slow` stops at index 3, which holds 1: the second of the two middle nodes, as

required. The algorithm makes about $n/2$ iterations and uses two pointers, one pass and $O(1)$ memory.

7.6 `hasCycle` is Floyd's "tortoise and hare" algorithm, the idea of Exercise 7.5 applied to a possibly circular list. If the list has no cycle, `fast` reaches NULL and the function returns `false`. If it has a cycle, both pointers eventually enter it, and inside the cycle the distance from `slow` to `fast`, measured along the direction of the links, shrinks by exactly one node per iteration (`fast` gains one step on `slow`), so it reaches 0 and the pointers meet after at most one lap of the cycle. The total work is $O(n)$, and the extra memory is two pointers. The program creates a cycle on purpose by linking the last node (9) back to the node at index 2, confirms that `hasCycle` returns `true`, restores the NULL link before any traversal could loop forever, and confirms `false`.

7.7 `mergeSorted` uses a dummy node on the *stack* (a local variable, so it needs no `delete`) as the start of the result, and a pointer `tail` to the result's last node. At each step it appends the smaller of the two front nodes and advances in that list. The reference `Node *&smaller` refers to `a` or to `b` itself, so `smaller = smaller->next` advances the right list. When one list is exhausted, the rest of the other is already sorted and is attached in one assignment. Each iteration handles one node, so the time is $\Theta(m+n)$ in the worst case, and no memory is allocated. Merging 1, 4, 4, 9 with 2, 4, 10 gives 1, 2, 4, 4, 4, 9, 10. Because the comparison is `<=`, equal values from `a` come before those from `b`, so the merge is stable. This is the merge step of merge sort, which is how `std::forward_list::sort` sorts in $O(n \lg n)$ time (Section 7.12.1).

7.8 In an ascending list, equal values are adjacent, so `dedupSorted` compares each node with its successor: if they are equal it deletes the successor (save, relink, delete) and stays put, because the new successor may also be equal; otherwise it moves on. The list 1, 2, 4, 4, 4, 9, 10 loses two nodes and becomes 1, 2, 4, 9, 10, in $\Theta(n)$ time.

7.9 The node before the insertion point is the one whose link must change, so `insertBefore` walks a *pointer to the link*, the technique of Section 7.7.3: `link` starts at `&head` and moves to `&(*link)->next` until `*link` is the node holding `key`. Then `*link = new Node{x, *link}` inserts the new node in front of it, and this single line assigns to `head` when the key is in the first node. In the output, 0 is inserted before 1 (the first node, so `head` changes), 7 before 9, and the call for the absent key 99 returns `false` and changes nothing. With a trailing `prev` pointer instead, the first node needs a separate branch; the Java and Python versions (`Exercises.java`, `exercises.py`) use a temporary dummy node instead.

7.10 When the list is empty, `head` is `nullptr`. The `else` branch executes `newNode->next = head`, which sets `newNode->next` to `nullptr`, and `head = newNode`, which makes the new node the first node. The result is a one-node list whose `next` is `nullptr`, exactly what the `if` branch produces, so the branch can be deleted (together with the now redundant `newNode->next = nullptr`). This is why **Listing 7.3** has no `if` in `frontInsertion`. The simplification works because front insertion never *reads* a field of an existing node; it only copies the value of `head`, which is well defined even when it is null. `backInsertion` must execute `aux->next` on the last node, and in the empty list there is no node for `aux` to point at, so the empty case must be handled separately (or eliminated with a dummy node, Section 7.7).

7.11 `josephus` in **Listing 7.17** builds the circle 1 to 7 with back insertion through `last`, and keeps `prev` on the node before the next person to count. Each round advances `prev` by $k-1$ nodes, deletes `prev->next` (the k-th person), and closes the circle. For $n=7$, $k=3$ the leaving order is 3, 6, 2, 7, 5, 1 and the survivor is 4, as the output shows. Check the first rounds by hand: counting 1,

2, **3** removes 3; counting 4, 5, **6** removes 6; counting 7, 1, **2** removes 2. There are $n-1$ rounds of $k-1$ moves each, so the time is $\Theta(nk)$ plus $\Theta(n)$ to build the circle.

7.12 Consider `enqueue(A)`, `dequeue()`, `enqueue(B)`. Without the deleted line, `dequeue` leaves `front_` equal to `nullptr` but `rear_` still holding the address of A's node, which it has just freed: a stale pointer. With `isEmpty` testing `rear_`, the queue now claims *not* to be empty, so `enqueue(B)` takes the `else` branch and executes `rear_->next = n`, writing into A's freed node, and `front_` is never set, so B is lost as well. This is bug 2 of Section 7.11.1 (a use after free through a stale pointer; the pointer is outside the chain, so it is not a dangling link inside the list). AddressSanitizer would stop at the line `else rear_->next = n;` with `heap-use-after-free` and `WRITE of size 8` (the 8-byte `next` field, which follows the `Book` inside the node), followed by the stack traces of the `delete` in `dequeue` and of the `new` in `enqueue(A)`. The original class survives the first change alone only by luck: its `isEmpty` and `enqueue` happen never to read `rear_` while `front_` is NULL. Any later addition that does (a `back()` accessor, a `size_` computed differently, an `isEmpty` rewritten as above) would turn the stale pointer into a bug. The safe rule is the head-tail invariant of Section 7.8.1: whenever the structure becomes empty, *both* pointers become NULL.

7.13 `reverseRec` makes one nested call per node, so for $n = 10^7$ nodes the call stack reaches a depth of ten million frames. At 48 bytes per frame that is 4.8×10^8 bytes, about 480 MB, far beyond the usual 8 MB stack limit on Linux (1 MB by default for the main thread on Windows), so the program would crash with a stack overflow long before finishing. `reverseIter` uses three pointers whatever n is. Both take $\Theta(n)$ time; only the iterative one runs in $O(1)$ memory.

7.14 (a) An array: after sorting, looking up by position (and binary search by book number) needs random access, $O(1)$ per access; a list would need $\Theta(n)$ steps per lookup. (b) A linked queue (or a circular array): readers join at the back and leave at the front, both $O(1)$, and the number waiting is unpredictable. (c) A stack; a linked stack or a `std::vector` used as a stack both work, since only the top is touched. With edits being small records, the vector is more economical, while a linked stack never needs to regrow. (d) Either works, since the list is only appended to and traversed once; a `std::vector` is the simplest and most cache-friendly, and a head-and-tail list would be equally correct with $O(1)$ appends.

References

[1] A. Newell and J. C. Shaw, "Programming the Logic Theory Machine," in *Proc. Western Joint Computer Conference*, Los Angeles, CA, USA, Feb. 1957, pp. 230–240.

[2] A. Newell and F. M. Tonge, "An introduction to Information Processing Language V," *Communications of the ACM*, vol. 3, no. 4, pp. 205–211, Apr. 1960.

[3] V. H. Yngve, "A programming language for mechanical translation," *Mechanical Translation*, vol. 5, no. 1, pp. 25–41, Jul. 1958.

[4] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960.

[5] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 2.2.

[6] U. Drepper, "What every programmer should know about memory," Red Hat, Inc., Tech. Rep., Nov. 2007. [Online]. Available: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

[7] L. Torvalds, "The mind behind Linux," interview by C. Anderson, TED2016, Vancouver, BC, Canada, Feb. 2016. [Online]. Available: https://www.ted.com/talks/linus_torvalds_the_mind_behind_linux

[8] Valgrind Developers, "Valgrind: Current releases." [Online]. Available: https://valgrind.org/downloads/current.html. [Accessed: 24-Sep-2026].

[9] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *Proc. 2012 USENIX Annual Technical Conference*, Boston, MA, USA, Jun. 2012, pp. 309–318.

[10] ISO/IEC 14882:2017, *Programming Languages — C++*, 5th ed. Geneva, Switzerland: ISO, 2017, sec. 26.3 (sequence containers `forward_list` and `list`).

[11] GNU Project, “GCC 16 release series.” [Online]. Available: https://gcc.gnu.org/gcc-16/. [Accessed: 24-Sep-2026].

[12] Oracle, “Class LinkedList,” *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/LinkedList.html. [Accessed: 24-Sep-2026].

[13] Python Software Foundation, “collections — Container datatypes: deque objects,” *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/collections.html. [Accessed: 24-Sep-2026].

[14] Python Software Foundation, “Modules/_collectionsmodule.c,” CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Modules/_collectionsmodule.c. [Accessed: 24-Sep-2026].

[15] B. Stroustrup, “C++11 style,” keynote, GoingNative 2012, Redmond, WA, USA, Feb. 2012.

[16] Profitina, “Profitina — AI Visibility (GEO) platform.” [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Chapter 8

Evaluation and Discussion II: Stacks, Pointers, and Linked Lists at Work

This chapter adds no new structure. It takes the three tools of Chapters 5, 6 and 7, the stack and the queue, the pointer and the function that receives one, and the singly linked list, and it puts them to work on twelve problems, every one solved from a blank page, one small step at a time, in C++, Java and Python. Five of the problems form one connected set: completing a stack class, walking a two-way list by directions, sorting numbers into evens and odds with nothing but linked lists, reversing the words of a sentence, and removing duplicate values from a list. Seven more are new, and each aims at one idea that beginners find slippery: matching brackets, a queue that wraps around the end of its array, evaluating an arithmetic expression, the three ways a function can receive its argument, reversing a list in place, merging two sorted lists, and a queue built from linked nodes. For every problem you will draw the pointers before you write the code, trace the program by hand, read three complete programs and the output they really print, study a tempting wrong solution and see exactly how it fails (for the memory bugs, the failure is reported by a memory checker, and you will read its report), and finish with a *check yourself* variant and its answer. By the end of the chapter you will have a method for any problem in which boxes point at other boxes, and you will have used it twelve times.

8.1 Where We Stand

Three chapters lie behind us, and each of them gave us one tool. It is worth naming the tools before we use them, because half of solving a problem is recognizing which tool it is asking for.

Chapter 5 gave us two containers with rules about their ends. A *stack* lets us add and remove items at one end only, the *top*; the last item in is the first item out, which is why a stack is called LIFO (last in, first out). Its operations are `push`, `pop`, `peek`, `isEmpty`, `isFull` and `clear`. We built it on an array with one integer, `top`, that remembers where the top item is, and we used it for a scientific calculator: turning an expression such as `3+2*5` into *postfix* form, `325*+`, and then evaluating the postfix form. A *queue* adds at one end, the *rear*, and removes at the other, the *front*; the first item in is the first out, FIFO. We built it on an array too, met the “false full” problem of a queue whose rear only ever moves to the right, and repaired it with a *circular* array whose indices wrap around.

Chapter 6 explained what a *pointer* is: a variable whose value is the address of another variable. We met the address-of operator `&`, the dereference operator `*` that follows an address to the variable living there, the arrow `->` that does the same for a member of a `struct`, the null pointer `nullptr` that points nowhere, and pointer arithmetic on arrays. Above all, Chapter 6 compared the three ways a C++ function can receive an argument, *by value*, *by reference* and *by pointer*, and showed which of them let the function change the caller’s variable.

Chapter 7 used pointers to build the *singly linked list*: a chain of *nodes*, each holding some data and a pointer `next` to the following node, with the last node’s `next` equal to `nullptr`. We wrote the list with a `head` pointer only and then with `head` and `tail`, and we wrote its operations one by

one: `init`, `isEmpty`, `frontInsertion`, `backInsertion`, `show`, `frontDeletion`, `backDeletion` and `clear`. Every one of them came with a picture of boxes and arrows, and every one of them had an empty-list case that needed its own line of code.

This chapter uses all three tools, often in the same program. **Table 8.1** lists the twelve problems, where each is solved, and which chapter supplies its main idea.

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

Throughout the chapter we keep the running example of this book, the catalog of the small library *Pustaka Ceria*, whose books have identifiers such as `B1001` and whose members wait in line for popular titles. The problems about stacks of plates and sentences use their own small data, because the data of the original problem statements is part of what makes them good exercises.

> **THE ONE IDEA BEHIND ALMOST EVERY PROBLEM HERE.** Nearly every problem in this chapter is the same rule in different clothes: *before you change a link, make sure you still have a way to reach everything you will need after the change.* In a stack, read the top item before you move `top` away from it. In the walk of Problem 8.2, save the step count before the pointer leaves the node that holds it. In a list reversal, save `cur->next` before you overwrite it. In a deletion, link around a node before you free it. In a queue, reset `tail` when the last node disappears. When a program in this chapter goes wrong, it is almost always because this rule was broken, and the wrong solutions below break it in all of these ways.

8.2 How to Approach a Pointer Problem

A problem about stacks, queues or linked lists is really a problem about *boxes that point at other boxes*. The code is short. The difficulty is keeping track of which box points where at each moment, and that is something paper does better than memory. This section gives a method in seven steps. On a small problem they take five minutes; on a hard one they are the difference between a program that works and a program that works on the example.

Step 1 — Restate the problem in your own words, with one example. Write down what goes in, what comes out, and one small input with its output worked by hand. If you cannot do the

example by hand, you are not ready to write code, because you do not yet know what the code must do.

Step 2 — Choose the structure, and say why. "Last in, first out" means a stack. "First come, first served" means a queue. "Insert and delete in the middle without shifting" means a linked list. "Jump back and forth" means two links per node. Write the reason in one sentence; the reason will tell you which operations you need.

Step 3 — Draw the boxes. Draw each node as a rectangle split into a data part and a pointer part, draw each pointer as an arrow that starts with a dot inside the box it lives in, and write every pointer variable of your program (`head`, `tail`, `cur`, `prev`, `top`) as a label with an arrow to the box it points at. Draw `nullptr` explicitly. This is the single most useful habit in the whole subject [1], [2].

Step 4 — Write the invariant. An *invariant* is a sentence that is true every time the program passes a certain point. For a stack: "the items are `a[0]` to `a[top]`, and `top` is −1 when the stack is empty". For a list with a tail: "`tail` points to the last node, or is `nullptr` when the list is empty". Every operation must leave the invariant true, and checking that it does is how you know the operation is right.

Step 5 — List the edge cases before writing code. For linked structures the list is almost always the same: the empty structure, a single node, the first node, the last node, and two equal values next to each other. For arrays: an empty array, a full array, and the index that wraps around.

Step 6 — Change the pointers in a safe order. For every operation, number the pointer assignments on your drawing and check, before each one, that nothing you still need becomes unreachable. The insight box of Section 8.1 is this step.

Step 7 — Test, including under a memory checker. Run the example of Step 1, then every edge case of Step 5. In C++, compile once more with the sanitizers switched on and run again; a memory bug that prints the right answer by luck is still a bug, and the sanitizer will say so.

Table 8.2 condenses the method onto one page. Every problem below applies it, and the labels in bold inside each problem (**In our own words**, **By hand**, **The C++ solution**, and so on) follow its order.

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?

How the programs of this chapter were checked. Every program below is complete: you can copy it into a file, compile it and run it. The C++ programs were compiled with GCC in C++17 mode with all the usual warnings switched on, and not one of them produces a warning. Each was then compiled a second time with AddressSanitizer and UndefinedBehaviorSanitizer [3], [4], two checkers built into GCC and Clang that watch every memory access while the program runs, and

run under Valgrind [5], which does a similar job without recompiling; every run was clean, with no invalid access and no leaked byte. The commands are these (on Windows, the executable is `p801.exe` and is run as `p801`):

```
g++ -std=c++17 -O2 -Wall -Wextra -o p801 p801_stack.cpp        normal build
./p801
g++ -std=c++17 -g -Wall -Wextra -fsanitize=address,undefined -o p801_asan p801_stack.cpp
./p801_asan                                                    same output, plus a
report if memory is misused
valgrind --leak-check=full ./p801                              Linux: the same check
without recompiling
./p802 < ../input/p802.txt                                     a program that reads
input, fed from a file
```

The Java programs were compiled with `javac` and run with `java`, and the Python programs with `python3`; they are written for JDK 25 and Python 3.14 but use nothing newer than JDK 21 and Python 3.10, so older installations run them too. For every problem the three programs print exactly the same lines, which was checked by comparing their outputs byte for byte. The one exception is Problem 8.9, whose subject is precisely the thing in which the three languages differ. The code folder of this chapter, described in its `README.md`, holds every program, every input file and the wrong solutions used for the demonstrations.

One node, three languages. C++ lets us see the pointers; Java and Python hide them, but the boxes and arrows are still there. **Table 8.3** is a small dictionary that lets you read the three versions of each program side by side.

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`

Two rows of **Table 8.3** deserve a sentence each. First, only C++ makes *you* destroy nodes; that is why only the C++ programs can leak memory or use a node after freeing it, and why only they are run under the memory checkers. Second, Java and Python have no way for a function to change the caller’s variable itself, so where the C++ version takes `Node *&head` the Java and Python versions *return* the new first node and the caller writes `head = insertSorted(head, ...)`. Problem 8.9 explains this difference properly [6], [7], [8].

8.3 Problem 8.1 — Completing an Array Stack

8.3.1 The problem

> **PROBLEM.** Complete the program below in C++. Implement the following functions: (a) `pop()`, which pops the uppermost item of the stack; (b) `push()`, which pushes an item onto the stack; (c) `peek()`, which shows the topmost item of the stack without removing it; (d) `isEmpty()`, which checks whether the stack is empty or not.

The program to complete is this. Only the four member functions are missing; the class, its constructor and the short driver in `main` are given.

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000

class Stack {
    int top;
public:
    int a[MAX]; // Maximum size of Stack
    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};

int main()
{
    class Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " popped from stack\n";
    return 0;
}
```

8.3.2 In our own words

We have a box of 1000 integer cells, `a[0]` to `a[999]`, and one integer `top`. The constructor sets `top` to -1. We must write four functions so that the object behaves like the stack of Chapter 5: `push(x)` puts `x` on top and reports with `true` or `false` whether it succeeded; `pop()` removes the top item and returns it; `peek()` returns the top item and leaves it where it is; `isEmpty()` answers whether there is any item at all. For the given driver, the expected output is one line, `30 popped from stack`, because 30 was the last value pushed.

The skeleton has already made the most important decision for us, and the whole problem is to read it correctly: **`top` starts at -1.** That tells us the meaning of `top`. It is not "the number of items" and not "the next free cell"; it is *the index of the cell that holds the top item*, and -1 is the index of "no cell at all". Once that sentence is written down, every function follows from it.

> **DEFINITION.** The *stack invariant* of this class: the items of the stack are exactly `a[0]`, `a[1]`, …, `a[top]`, from the bottom to the top, and $-1 \le$ `top` $\le$ `MAX` -1. The stack is empty when `top` is -1 and full when `top` is `MAX` -1.

From the invariant we can read off how many items there are. Cells `a[0]` to `a[top]` are top + 1 cells, so

$$\text{size} = \text{top} + 1, \qquad \text{empty} \Leftrightarrow \text{top} = -1, \qquad \text{full} \Leftrightarrow \text{top} = \text{MAX} - 1. \tag{8.1}$$

Equation (8.1) is worth checking on the smallest cases. With no items, `top` is -1 and the size is $-1 + 1 = 0$. With one item, it lives in `a[0]`, `top` is 0 and the size is 1. With the stack full, `top` is 999 and the size is 1000, which is `MAX`. The formula is right at both ends.

8.3.3 By hand, step by step

Figure 8.1 draws the first four cells of the array after each operation of the driver. Follow it with the invariant in mind.

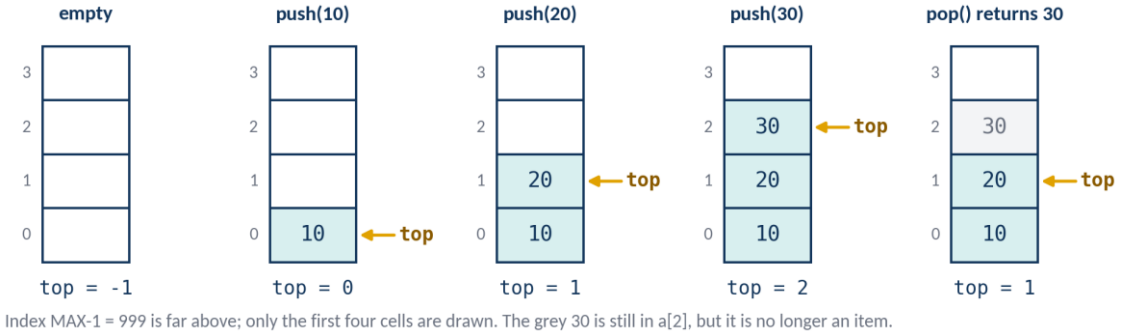


Figure 8.1 The array stack of Problem 8.1 after each operation of the driver: `top` is the index of the top item, and a pop only moves `top`; the value 30 stays in the array but is no longer part of the stack.

1. **Start.** `top` is -1. No cell is part of the stack.
2. **`push(10)`.** The stack is not full ($-1 \neq 999$). The new item must go into the cell *above* the current top, which is cell $-1 + 1 = 0$. So we first increase `top` to 0 and then store: `a[0] = 10`. The invariant holds again: the items are `a[0]..a[0]`.
3. **`push(20)`.** `top` becomes 1, then `a[1] = 20`.
4. **`push(30)`.** `top` becomes 2, then `a[2] = 30`. The items are `a[0]..a[2]` = 10, 20, 30.
5. **`pop()`.** The stack is not empty ($2 \neq -1$). The top item is `a[top]` = `a[2]` = 30. We must *first read it*, and *then* decrease `top` to 1. The function returns 30, and the driver prints `30 popped from stack`. Notice that nothing was erased: `a[2]` still contains 30. It is simply no longer between `a[0]` and `a[top]`, so by the invariant it is no longer an item. The next push will overwrite it.

The order of the two actions in `push` and `pop` is the whole difficulty of this problem, and C++ lets us write each pair in one expression. In `a[++top] = x`, the *pre-increment* `++top` increases `top` first and then uses the new value as the index. In `return a[top--];`, the *post-decrement* `top--` uses the old value as the index and decreases `top` afterwards. Read them aloud as "increase, then store" and "read, then decrease".

What happens at the edges. A push on a full stack must not write to `a[1000]`, which does not exist; our `push` prints a message and returns `false`, which is what its `bool` return type is for. A pop on an empty stack has nothing to return. The skeleton's return type is `int`, and returning some special number such as 0 or -1 would be a lie, because 0 and -1 are perfectly good items that a real stack might hold. So our `pop` and `peek` *throw an exception*, `std::underflow_error`, which a caller can catch; a well-behaved caller asks `isEmpty()` first and never triggers it.

8.3.4 The C++ solution

Listing 8.1 is the completed program. It keeps the skeleton exactly, adds one small helper `size()` that we use in the tests, and extends `main` with checks for the edge cases after the given driver. We replaced `<bits/stdc++.h>`, which exists only in GCC, by the two standard headers the program really needs, so that it compiles with every C++ compiler.

Listing 8.1 `p801_stack.cpp`: the array stack of Problem 8.1, completed, with the given driver and our own edge-case checks.

```
// p801_stack.cpp - Problem 8.1: complete an array-based Stack class.
#include <iostream>
#include <stdexcept>
using namespace std;

#define MAX 1000   // capacity of every Stack, kept from the given skeleton

class Stack {
    int top;       // index of the top item; -1 means "the stack is empty"

public:
    int a[MAX];    // the storage: the items live in a[0..top]

    Stack() {
        top = -1;
    }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
    int size();    // one extra helper, not in the skeleton
};

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
}

int Stack::peek() {
    if (isEmpty()) {
        throw underflow_error("peek at an empty stack");
    }
    return a[top];                 // read only; top does not move
}
```

Listing 8.1 (Continued) *p801_stack.cpp: the array stack of Problem 8.1, completed, with the given driver and our own edge-case checks.*

```
bool Stack::isEmpty() {
    return top < 0;
}

int Stack::size() {
    return top + 1;
}

int main() {
    Stack s;                                    // the given driver
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " popped from stack\n";

    // Our own extra checks.
    cout << "peek -> " << s.peek() << ", size " << s.size() << "\n";
    cout << "pop -> " << s.pop() << "\n";
    cout << "pop -> " << s.pop() << "\n";
    cout << boolalpha << "isEmpty -> " << s.isEmpty() << "\n";
    try {
        s.pop();
    } catch (const underflow_error& e) {
        cout << "caught: " << e.what() << "\n";
    }

    int pushed = 0;
    for (int v = 1; v <= MAX; v++) {
        if (s.push(v)) pushed++;
    }
    cout << "pushed " << pushed << " items, size " << s.size() << "\n";
    bool ok = s.push(1001);                     // prints the overflow message
    cout << "push(1001) returned " << ok << "\n";
    cout << "peek -> " << s.peek() << "\n";
    return 0;
}
```

```
30 popped from stack
peek -> 20, size 2
pop -> 20
pop -> 10
isEmpty -> true
caught: pop from an empty stack
pushed 1000 items, size 1000
Stack overflow: 1001 not pushed
push(1001) returned false
peek -> 1000
```

Line by line. The first line of output is the given driver's, and it is the expected `30 popped from stack`. The next lines check `peek` (it returns 20 and leaves the size at 2), empty the stack with two pops (20, then 10, in reverse order of pushing, as LIFO requires), confirm that `isEmpty` is now true, and show that a third pop is refused with an exception instead of returning a made-up value. The loop then pushes 1000 values and counts how many were accepted: all 1000. The 1001st push is refused, the message comes from inside `push`, and `peek` confirms that the top is still the value 1000, so nothing was overwritten. `boolalpha` makes `cout` print `true` and `false` instead of 1 and 0.

Why is `isEmpty` written as `top < 0` rather than `top == -1`? By the invariant the two are the same, so either is correct; `< 0` is merely a little more forgiving if some future bug ever pushed `top` below

−1. Why does `pop` call `isEmpty()` instead of repeating the test? Because then the meaning of "empty" is written in one place only, and a change to it cannot leave one function behind.

Cost. Every one of the five functions does a fixed number of steps, whatever the size of the stack: one comparison and one or two assignments. In the notation of Chapter 2, each costs $O(1)$ time. The whole array is allocated once, when the object is created, so the stack uses $\Theta(\text{MAX})$ memory even when it is empty; that is the price of an array stack, and the linked stack of Problem 8.4 is the answer to it.

8.3.5 The Java and Python solutions

Listing 8.2 and **Listing 8.3** are the same class in Java and Python. Both print exactly the lines above. Java has no `underflow_error`, so `pop` throws `IllegalStateException`, the standard Java exception for "you called this method at a moment when it cannot work". Python has no fixed-size arrays, so the program makes one with `[0] * MAX`, a list of 1000 zeros that it never lets grow; it raises `IndexError` on an empty stack, the same exception Python's own `list.pop()` raises. In real Python programs a plain list *is* the stack: `append` pushes, `pop()` pops and `[-1]` peeks, all in $O(1)$ time [8]. We write the array version here so that the three programs have the same shape.

Listing 8.2 `P801Stack.java`: Problem 8.1 in Java.

```java
// P801Stack.java - Problem 8.1: complete an array-based Stack class.
public class P801Stack {

    static final int MAX = 1000;       // capacity of every Stack

    static class Stack {
        private int top = -1;          // index of the top item; -1 means "the stack is
empty"
        private final int[] a = new int[MAX];   // the items live in a[0..top]

        boolean push(int x) {
            if (top >= MAX - 1) {      // a[MAX-1] is already in use: full
                System.out.println("Stack overflow: " + x + " not pushed");
                return false;
            }
            a[++top] = x;              // FIRST move top up, THEN store
            return true;
        }

        int pop() {
            if (isEmpty()) {
                throw new IllegalStateException("pop from an empty stack");
            }
            return a[top--];           // FIRST read a[top], THEN move top down
        }

        int peek() {
            if (isEmpty()) {
                throw new IllegalStateException("peek at an empty stack");
            }
            return a[top];
        }

        boolean isEmpty() {
            return top < 0;
        }

        int size() {
            return top + 1;
        }
    }
```

Listing 8.2 (Continued) *P801Stack.java: Problem 8.1 in Java.*

```
    public static void main(String[] args) {
        Stack s = new Stack();                // the given driver
        s.push(10);
        s.push(20);
        s.push(30);
        System.out.println(s.pop() + " popped from stack");

        // Our own extra checks.
        System.out.println("peek -> " + s.peek() + ", size " + s.size());
        System.out.println("pop -> " + s.pop());
        System.out.println("pop -> " + s.pop());
        System.out.println("isEmpty -> " + s.isEmpty());
        try {
            s.pop();
        } catch (IllegalStateException e) {
            System.out.println("caught: " + e.getMessage());
        }

        int pushed = 0;
        for (int v = 1; v <= MAX; v++) {
            if (s.push(v)) pushed++;
        }
        System.out.println("pushed " + pushed + " items, size " + s.size());
        boolean ok = s.push(1001);            // prints the overflow message
        System.out.println("push(1001) returned " + ok);
        System.out.println("peek -> " + s.peek());
    }
}
```

Listing 8.3 `p801_stack.py`: Problem 8.1 in Python, with a fixed block of `MAX` cells so that the logic matches the C++ array.

```
# p801_stack.py - Problem 8.1: complete an array-based Stack class.
MAX = 1000   # capacity of every Stack


class Stack:
    def __init__(self):
        self._top = -1              # index of the top item; -1 means "the stack is empty"
        self.a = [0] * MAX          # a fixed block of MAX cells, like the C++ array

    def push(self, x):
        if self._top >= MAX - 1:    # a[MAX-1] is already in use: full
            print(f"Stack overflow: {x} not pushed")
            return False
        self._top += 1              # FIRST move top up,
        self.a[self._top] = x       # THEN store
        return True

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from an empty stack")
        x = self.a[self._top]       # FIRST read a[top],
        self._top -= 1              # THEN move top down
        return x

    def peek(self):
        if self.is_empty():
            raise IndexError("peek at an empty stack")
        return self.a[self._top]

    def is_empty(self):
        return self._top < 0

    def size(self):
        return self._top + 1


def main():
    s = Stack()                                 # the given driver
    s.push(10)
    s.push(20)
    s.push(30)
    print(s.pop(), "popped from stack")
```

Listing 8.3 (Continued) *p801_stack.py: Problem 8.1 in Python, with a fixed block of MAX cells so that the logic matches the C++ array.*

```
    # Our own extra checks.
    print(f"peek -> {s.peek()}, size {s.size()}")
    print("pop ->", s.pop())
    print("pop ->", s.pop())
    print("isEmpty ->", str(s.is_empty()).lower())
    try:
        s.pop()
    except IndexError as e:
        print("caught:", e)

    pushed = sum(1 for v in range(1, MAX + 1) if s.push(v))
    print(f"pushed {pushed} items, size {s.size()}")
    ok = s.push(1001)                           # prints the overflow message
    print("push(1001) returned", str(ok).lower())
    print("peek ->", s.peek())


if __name__ == "__main__":
    main()
```

8.3.6 A tempting wrong solution

The most common mistake in this problem is to get the order of "move" and "read" backwards in `pop`:

```
int pop() { return a[--top]; }   // WRONG: moves top first, then reads
```

It compiles without a warning and looks almost identical to the right line. Run it on the given driver (the program `wrong/p801_wrong.cpp` in the code folder does exactly that, and pops twice):

```
20 popped from stack
10 popped from stack
isEmpty -> false
```

The first pop decreases `top` from 2 to 1 *before* reading, and so returns `a[1]`, which is 20, the item *under* the top. The 30 is never returned at all. After two pops `top` is 0, so the stack claims not to be empty although we pushed three items and popped only two; a third pop would compute `a[-1]`, a cell that does not exist. The invariant tells you immediately which version is right: the top item *is* `a[top]`, so `pop` must read `a[top]` while `top` still has its old value. The mirror-image mistake in `push`, `a[top++] = x`, is worse: with `top` equal to −1 it writes to `a[-1]`, outside the array, which is undefined behavior in C++. That is exactly the kind of error AddressSanitizer exists to catch.

8.3.7 Check yourself

> **CHECK YOURSELF.** Suppose the constructor were changed to `top = 0`, so that `top` now means "the index of the next free cell". Rewrite the one important line of `push`, of `pop` and of `peek`, and the test in `isEmpty`, and give the new version of Equation (8.1).
>
> *Answer.* `push`: `a[top++] = x;` (store, then increase), and the full test becomes `top >= MAX`. `pop`: `return a[--top];` (decrease, then read). `peek`: `return a[top - 1];`. `isEmpty`: `return top == 0;`. Now the items are `a[0]..a[top-1]`, so size = `top`, empty ⇔ `top` = 0, full ⇔ `top` = `MAX`. Both conventions are correct; what is never correct is to mix them, which is exactly what the wrong solution of Section 8.3.6 did.

8.4 Problem 8.2 — Walking a Two-Way List by Directions

8.4.1 The problem

> **PROBLEM.** Complete the program below in C++. Each input line has the form *direction step data*: *direction* is the direction in which to traverse next, `b` for backward or `f` for forward; *step* is the number of nodes to move in that direction (for example, `b 5` means traverse 5 nodes backward); *data* is the character stored in the node. The input stops when the direction is `0`. Print the data taken from the traversal. For the sample input `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` the output is `strukdat`.

The given code, with the two function bodies left empty, is:

```
typedef struct snode_t {            // Node
    char data, direction;
    int step;
    struct snode_t *next;
    struct snode_t *prev;
} node;

typedef struct dlist_t {            // Deque
    unsigned _size;
    node *_head;
} linkedList;

bool slist_isEmpty(linkedList *list) { return (list->_head == NULL); }
void slist_pushBack(linkedList *list, char direction, char data, int step) { /* push back
*/ }
void transverse(linkedList *myList) { /* traverse the list */ }

int main() {
    linkedList *myList = new linkedList;
    myList->_head = NULL;
    char direction, data;
    int step;
    do {
        cin >> direction >> step >> data;
        slist_pushBack(myList, direction, data, step);
    } while (direction != '0');
    transverse(myList);
    return 0;
}
```

(The given code calls the second function `transverse`; the usual word is *traverse*, and our program uses it.)

8.4.2 Just enough about nodes with two links

This problem reaches one step beyond Chapter 7. Look at the node type: besides `next` it has a second pointer, `prev`. A list whose nodes point both to their successor and to their predecessor is a *doubly linked list*, and Chapter 9 is devoted to it: insertion and deletion at both ends, the list with a tail, the circular version, and the *deque* (double-ended queue) that the comment in the given code mentions. For this problem we need much less than Chapter 9, and it fits in three sentences.

1. **Each node has two arrows.** `next` points to the node after it and `prev` to the node before it. The first node's `prev` is `nullptr`, and the last node's `next` is `nullptr`.
2. **Moving is one assignment.** Moving one node forward is `cur = cur->next`, exactly as in Chapter 7; moving one node backward is `cur = cur->prev`. Nothing else is new.
3. **Adding at the back sets two arrows, not one.** When a new node is appended after the current last node `last`, we must set `last->next = newNode` (the forward arrow, as in Chapter 7) *and* `newNode->prev = last` (the backward arrow). Forgetting the second assignment builds a list that looks perfect when walked forward and breaks the first time anybody walks backward.

> **DEFINITION.** The *two-way list invariant*: for every node `p` that has a successor, `p->next->prev == p`; the first node's `prev` is `nullptr`; the last node's `next` is `nullptr`. In words: every forward arrow has a matching backward arrow.

Figure 8.2 shows the list that the sample input builds. There is one more thing to notice in it, and it is the key to the whole problem.

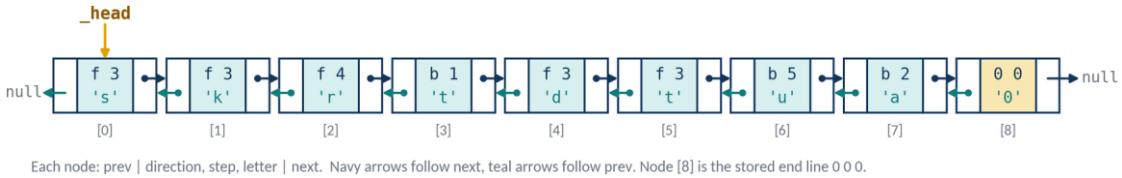


Figure 8.2 The two-way list built from the sample input of Problem 8.2: nine nodes, each with a `prev` and a `next` pointer. Node 8 holds the end line `0 0 0`, which the given reading loop also stores.

The end line is a node too. Read the given `main` slowly. It is a `do`–`while` loop: it reads a line, pushes it, and only *then* tests whether the direction was `'0'`. So the line `0 0 0` is pushed like every other line, and the list has nine nodes, not eight. This is not a bug to fix; it is how the walk knows where to stop. The walk ends when it *arrives* at a node whose direction is `'0'`. The sample input is built so that the eighth jump lands exactly on that node.

8.4.3 The walk, by hand

Here is the rule of the walk, in our own words. Start at the first node. Repeat: if the current node's direction is `'0'`, stop; otherwise print its letter, then move *step* nodes in its direction, one node at a time. **Table 8.4** follows the rule on the sample input, and **Figure 8.3** draws the same eight moves as arcs above and below the list.

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	—

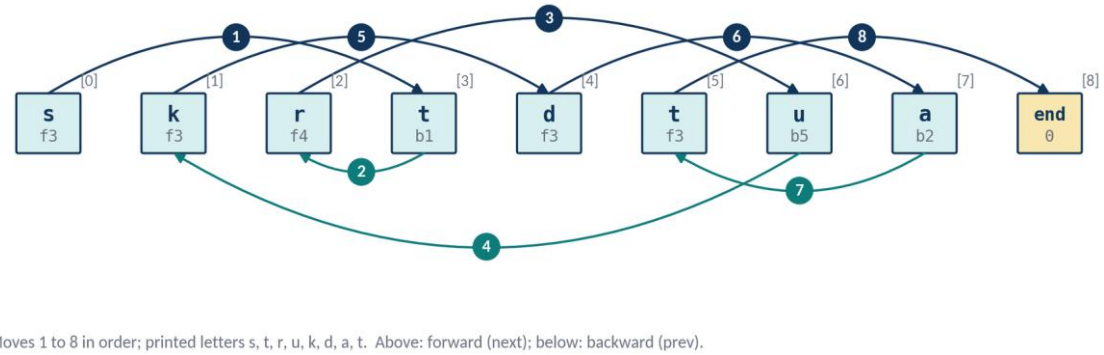


Figure 8.3 The eight moves of the walk: forward moves above the list, backward moves below, numbered in the order they happen. Every letter node is visited exactly once.

Reading the "Prints" column from top to bottom gives `s t r u k d a t`, the expected `strukdat`. (The word is the familiar Indonesian abbreviation of *struktur data*, data structures, spelled out by

a list that hides its letters out of order.) Notice from the figure that the walk visits every letter node exactly once. That is a property of this particular input, not of every input, and it leads to the next question.

8.4.4 What can go wrong in the input

A program that reads input must decide what to do with input that is not as promised. This one has three dangers, and the given code guards against none of them.

The input ends without the line `0 0 0`. Then `cin >> direction` fails, the variables keep their old values, and the `do-while` loop pushes the same node again and again, forever. Our `main` tests whether the reading succeeded and leaves the loop if it did not.

A move runs off the end. An instruction such as `f 5` on the second-to-last node would follow `next` past the last node into `nullptr`, and the next `cur->next` would crash the program. Our walk checks `cur != nullptr` inside the moving loop and reports `[the walk left the list]`.

The directions go around in a circle. Two nodes `f 1` and `b 1` side by side send the walk back and forth between them forever. Can the program notice? Yes, by a counting argument. Where the walk goes next depends *only* on the node it is standing on. So if the walk ever stands on the same node twice, everything that happened after the first visit will happen again after the second, and again, forever. In a list of n nodes, a walk that never repeats a node can make at most n visits; so if we count visits and the count exceeds n, some node has certainly been visited twice (with more visits than nodes, two visits must share a node), and the walk will never end:

$$\begin{aligned} \text{visits} > n \;&\Rightarrow\; \text{some node was visited twice} \\ &\Rightarrow\; \text{the walk never reaches the end.} \end{aligned} \tag{8.2}$$

Equation (8.2) is the *pigeonhole principle*: if more than n pigeons sit in n holes, some hole holds two pigeons. Our walk counts its visits and stops with `[the directions go around in a circle]` as soon as the count passes `_size`, the number of nodes. The check costs one integer and one comparison per visit, and it turns a program that could hang into one that always stops.

8.4.5 The C++ solution

Listing 8.4 keeps the given types, names and reading loop, fills in the two functions, adds the checks of Section 8.4.4, and frees every node at the end.

Listing 8.4 `p802_deque_walk.cpp`: Problem 8.2, the two-way list built by `slist_pushBack` and walked by `traverse`.

```cpp
// p802_deque_walk.cpp - Problem 8.2: build a two-way linked list, then walk it by
directions.
#include <iostream>
using namespace std;

typedef struct snode_t {       // one node: a letter, a direction, a step, and two links
    char data, direction;
    int step;
    struct snode_t *next;      // the node after this one (nullptr at the back)
    struct snode_t *prev;      // the node before this one (nullptr at the front)
} node;

typedef struct dlist_t {       // the list itself: a size and the first node
    unsigned _size;
    node *_head;
} linkedList;

bool slist_isEmpty(linkedList *list) {
    return (list->_head == nullptr);
}

void slist_pushBack(linkedList *list, char direction, char data, int step) {
    node *newNode = new node;
    newNode->data = data;
    newNode->direction = direction;
    newNode->step = step;
    newNode->next = nullptr;              // it will be the last node
    if (slist_isEmpty(list)) {
        newNode->prev = nullptr;          // nothing before the first node
        list->_head = newNode;
    } else {
        node *last = list->_head;
        while (last->next != nullptr) {   // walk to the current last node
            last = last->next;
        }
        last->next = newNode;             // link forward:  last -> newNode
        newNode->prev = last;             // link backward: last <- newNode
    }
    list->_size++;
}
```

Listing 8.4 (Continued) *p802_deque_walk.cpp: Problem 8.2, the two-way list built by slist_pushBack and walked by traverse.*

```
// Prints the letters met on the walk. Stops at the node whose direction is '0'.
void traverse(linkedList *myList) {
    node *cur = myList->_head;
    unsigned visits = 0;
    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]";
        }
    }
    cout << "\n";
}

void slist_destroy(linkedList *list) {
    node *cur = list->_head;
    while (cur != nullptr) {
        node *after = cur->next;          // save the way on before deleting
        delete cur;
        cur = after;
    }
    list->_head = nullptr;
    list->_size = 0;
}

int main() {
    linkedList *myList = new linkedList;
    myList->_size = 0;
    myList->_head = nullptr;
```

Listing 8.4 (Continued) *p802_deque_walk.cpp: Problem 8.2, the two-way list built by slist_pushBack and walked by traverse.*

```
    char direction, data;
    int step;
    do {
        if (!(cin >> direction >> step >> data)) {
            break;                        // input ended without the line 0 0 0
        }
        slist_pushBack(myList, direction, data, step);
    } while (direction != '0');

    traverse(myList);

    slist_destroy(myList);
    delete myList;
    return 0;
}
```

Here are four runs: the sample input, a second input of our own, an input whose directions go around in a circle, and one whose move leaves the list. The input files are in the `input` folder of the chapter's code.

```
$ ./p802 < p802.txt
strukdat
$ ./p802 < p802_check.txt
list
$ ./p802 < p802_loop.txt
aba [the directions go around in a circle]
$ ./p802 < p802_off.txt
xy [the walk left the list]
```

Line by line. `slist_pushBack` creates a node with `new`, fills its three data fields and sets `next` to `nullptr`, because a node appended at the back has no successor. If the list is empty, the new node becomes the head and its `prev` is `nullptr`. Otherwise the loop `while (last->next != nullptr) last = last->next;` walks to the current last node, the same auxiliary-pointer walk as Chapter 7's `backInsertion`, and the two assignments `last->next = newNode;` and `newNode->prev = last;` set the forward and the backward arrow. The size counter goes up by one in both cases.

`traverse` follows the rule of Section 8.4.3. The two lines to read twice are these:

```
int step = cur->step;             // SAVE the instructions of this node
char direction = cur->direction;  // before cur moves away from it
```

The instructions belong to the node where the move *starts*. The moment the inner loop executes its first `cur = cur->next`, `cur` is standing on a different node with different instructions, so the step count and the direction must be copied into local variables before the first move. Then the inner loop makes exactly `step` moves, following `prev` for `'b'` and `next` otherwise, and stops early if it falls off the list. `slist_destroy` frees the nodes one by one, saving `cur->next` before each `delete`, and `main` deletes the list header itself, so that every `new` has its `delete`. Valgrind and AddressSanitizer report no error and no leak on all four inputs.

Cost. Each `slist_pushBack` walks the whole list to find the last node, so building a list of n nodes costs $0 + 1 + \cdots + (n-1) = \frac{n(n-1)}{2}$ steps, which is $\Theta(n^2)$. The given list type has a `_head` but no tail pointer, so that cost comes with the problem; Chapter 9's list keeps a tail and appends in $O(1)$. The walk itself costs one step per node moved over: $\sum$ of the step values of the visited nodes, which is 24 for the sample input.

8.4.6 The Java and Python solutions

Listing 8.5 and **Listing 8.6** print the same four results as the C++ program. Both read all the input at once and split it into words, which makes a missing end line easy to detect. Neither has a destroy function: when `main` finishes, no variable refers to the nodes any more, and the garbage collector reclaims them.

Listing 8.5 `P802DequeWalk.java`: Problem 8.2 in Java.

```
// P802DequeWalk.java - Problem 8.2: build a two-way linked list, then walk it by
directions.
public class P802DequeWalk {

    static class Node {                // one node: a letter, a direction, a step, and two
links
        char data, direction;
        int step;
        Node next;                     // the node after this one (null at the back)
        Node prev;                     // the node before this one (null at the front)
    }

    static class LinkedList {          // the list itself: a size and the first node
        int size;
        Node head;
    }

    static boolean isEmpty(LinkedList list) {
        return list.head == null;
    }

    static void pushBack(LinkedList list, char direction, char data, int step) {
        Node newNode = new Node();
        newNode.data = data;
        newNode.direction = direction;
        newNode.step = step;
        newNode.next = null;                    // it will be the last node
        if (isEmpty(list)) {
            newNode.prev = null;                // nothing before the first node
            list.head = newNode;
        } else {
            Node last = list.head;
            while (last.next != null) {         // walk to the current last node
                last = last.next;
            }
            last.next = newNode;                // link forward:  last -> newNode
            newNode.prev = last;                // link backward: last <- newNode
        }
        list.size++;
    }
```

Listing 8.5 (Continued) *P802DequeWalk.java: Problem 8.2 in Java.*

```
    // Prints the letters met on the walk. Stops at the node whose direction is '0'.
    static void traverse(LinkedList list) {
        StringBuilder out = new StringBuilder();
        Node cur = list.head;
        int visits = 0;
        while (cur != null && cur.direction != '0') {
            if (++visits > list.size) {        // a node came around twice: endless loop
                out.append(" [the directions go around in a circle]");
                break;
            }
            out.append(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 != null; k++) {
                cur = (direction == 'b') ? cur.prev : cur.next;
            }
            if (cur == null) {
                out.append(" [the walk left the list]");
            }
        }
        System.out.println(out);
    }

    public static void main(String[] args) throws java.io.IOException {
        String text = new String(System.in.readAllBytes()).trim();
        String[] tok = text.isEmpty() ? new String[0] : text.split("\\s+");
        LinkedList myList = new LinkedList();
        int i = 0;
        char direction;
        do {
            if (i + 2 >= tok.length) {
                break;                         // input ended without the line 0 0 0
            }
            direction = tok[i].charAt(0);
            int step = Integer.parseInt(tok[i + 1]);
            char data = tok[i + 2].charAt(0);
            i += 3;
            pushBack(myList, direction, data, step);
        } while (direction != '0');
        traverse(myList);
        // No destroy step: the garbage collector frees the nodes once nothing refers to
them.
    }
}
```

Listing 8.6 `p802_deque_walk.py`: Problem 8.2 in Python. `__slots__` fixes the fields of a node, which makes each node smaller and catches misspelled field names.

```
# p802_deque_walk.py - Problem 8.2: build a two-way linked list, then walk it by
directions.
import sys


class Node:
    """One node: a letter, a direction, a step, and two links."""
    __slots__ = ("data", "direction", "step", "next", "prev")

    def __init__(self, direction, step, data):
        self.data = data
        self.direction = direction
        self.step = step
        self.next = None            # the node after this one (None at the back)
        self.prev = None            # the node before this one (None at the front)


class LinkedList:
    def __init__(self):
        self.size = 0
        self.head = None


def push_back(lst, direction, data, step):
    new_node = Node(direction, step, data)
    if lst.head is None:
        lst.head = new_node         # its prev stays None: nothing before the first node
    else:
        last = lst.head
        while last.next is not None:    # walk to the current last node
            last = last.next
        last.next = new_node            # link forward:  last -> new_node
        new_node.prev = last            # link backward: last <- new_node
    lst.size += 1


def traverse(lst):
    """Prints the letters met on the walk. Stops at the node whose direction is '0'."""
    out = []
    cur = lst.head
    visits = 0
    while cur is not None and cur.direction != "0":
        visits += 1
        if visits > lst.size:           # a node came around twice: endless loop
            out.append(" [the directions go around in a circle]")
            break
        out.append(cur.data)
```

Listing 8.6 (Continued) *p802_deque_walk.py: Problem 8.2 in Python. __slots__ fixes the fields of a node, which makes each node smaller and catches misspelled field names.*

```
        step = cur.step                 # SAVE the instructions of this node
        direction = cur.direction       # before cur moves away from it
        k = 0
        while k < step and cur is not None:
            cur = cur.prev if direction == "b" else cur.next
            k += 1
        if cur is None:
            out.append(" [the walk left the list]")
    print("".join(out))


def main():
    tok = sys.stdin.read().split()
    my_list = LinkedList()
    i = 0
    while i + 2 < len(tok):             # stops early if the line 0 0 0 is missing
        direction, step, data = tok[i], int(tok[i + 1]), tok[i + 2]
        i += 3
        push_back(my_list, direction, data, step)
        if direction == "0":
            break
    traverse(my_list)


if __name__ == "__main__":
    main()
```

8.4.7 A tempting wrong solution

The shortest-looking walk uses the node's fields directly in the loop condition:

```
for (int k = 0; k < cur->step; k++)
    cur = (cur->direction == 'b') ? cur->prev : cur->next;   // WRONG
```

It reads naturally: "take `step` steps in the node's direction". But `cur` changes inside the loop, so `cur->step` and `cur->direction` are read from a *different node* on every pass. On the sample input the walk starts correctly at node 0 (`f 3`), takes one step to node 1, then compares `k` with the step of node 1, takes another step to node 2, compares with the step of node 2 (which is 4), takes a third step to node 3, and there reads the direction `b`. From node 3 it steps back to node 2, whose direction is `f`, and forward again to node 3. It is trapped. The program `wrong/p802_wrong.cpp` makes this walk and stops it after twenty letters:

```
stttttttttttttttttttt
```

The repair is the rule of Section 8.1 in its purest form: *copy what you need out of a node before you leave it.*

8.4.8 Check yourself

> **CHECK YOURSELF.** What does the program print for the input `f 2 l`, `f 2 s`, `b 1 i`, `f 1 t`, `0 0 0`? Draw the five nodes and the moves before you look at the answer. Then say which single change to the second line would make the walk leave the list.
>
> *Answer.* The walk goes node 0 `l` → forward 2 → node 2 `i` → backward 1 → node 1 `s` → forward 2 → node 3 `t` → forward 1 → node 4, the end node. It prints `list`, as the second run of **Listing 8.4** shows. Changing the second line to `f 4 s` makes the move from node 1 go four nodes forward, to node 5, which does not exist: the walk prints `lis` and then `[the walk left the list]`.

8.5 Problem 8.3 — Evens Up, Odds Down, With Linked Lists Only

8.5.1 The problem

> **PROBLEM.** Write a C++ program that sorts the numbers it reads: output the even numbers in ascending order, then the odd numbers in descending order. You must use a linked list for this problem; do not use arrays. Input: the first line is N, the number of values; each of the next N lines holds one value. Output: the sorted values, one per line. For the input 4, 32, 34, 543, 3456, 654, 567, 87, 6789, 98 the output is 4, 32, 34, 98, 654, 3456, 6789, 567, 543, 87.

8.5.2 In our own words

Split the numbers into two groups, the even ones and the odd ones. Print the even group from smallest to largest, then the odd group from largest to smallest. The only restriction is on *how*: no array may hold the numbers, so we cannot read them into an array and call a sort. The numbers must live in linked lists.

That restriction points straight at a technique from Chapter 7: if every new value is inserted *at its correct place* in a list that is already sorted, the list stays sorted after every insertion, and when the input ends there is nothing left to sort. This is *insertion sort* (Chapter 3) done on a linked list, and on a linked list it is especially natural, because inserting in the middle costs two pointer assignments and no shifting of elements. We keep two lists, `evens` sorted ascending and `odds` sorted descending, and send each value to the right one.

The two lists need two different orders, but the insertion itself is the same procedure. Rather than write it twice, we write it once and pass it a *function* that answers one question: "should `a` come before `b`?" For `evens` the answer is `a < b`; for `odds` it is `a > b`. In C++ a function can be passed to another function through a *pointer to a function*, a Chapter 6 idea we now use for real. The parameter is written

```
bool (*before)(int, int)
```

and read from the inside out: `before` is a pointer (`*before`) to a function that takes two `int`s and returns a `bool`. Inside `insertSorted` we call it like any function, `before(value, head->data)`, and at the call site we pass the name of a function, `ascending` or `descending`, without parentheses.

8.5.3 By hand, step by step

Is a number even? A number is even exactly when dividing it by 2 leaves remainder 0, so the test is `x % 2 == 0`. Keep that exact form; Section 8.5.6 shows why the more natural-looking `x % 2 == 1` for "odd" is a trap.

Inserting one value. Take the moment in the sample when 98 arrives and `evens` already holds 4, 32, 34, 654, 3456 (**Figure 8.4**). We want the new node right after the *last node that should stay in front of it*. There are two cases.

1. **The new value belongs before the first node** (or the list is empty). Then the new node becomes the first: `fresh->next = head; head = fresh;`. This changes `head` itself, which is why the function receives `head` by reference.
2. **Otherwise**, start `cur` at the first node and move it forward while the node *after* it should still come first: `while (cur->next != nullptr && !before(value, cur->next->data)) cur = cur->next;`. For 98: 98 < 32? No, move to 32. 98 < 34? No, move to 34. 98 < 654? Yes: stop. `cur` is on 34. Now link the new node in, in the safe order: first `fresh->next = cur->next;` (the new node points to 654), and only then `cur->next = fresh;` (34 points to the new node).

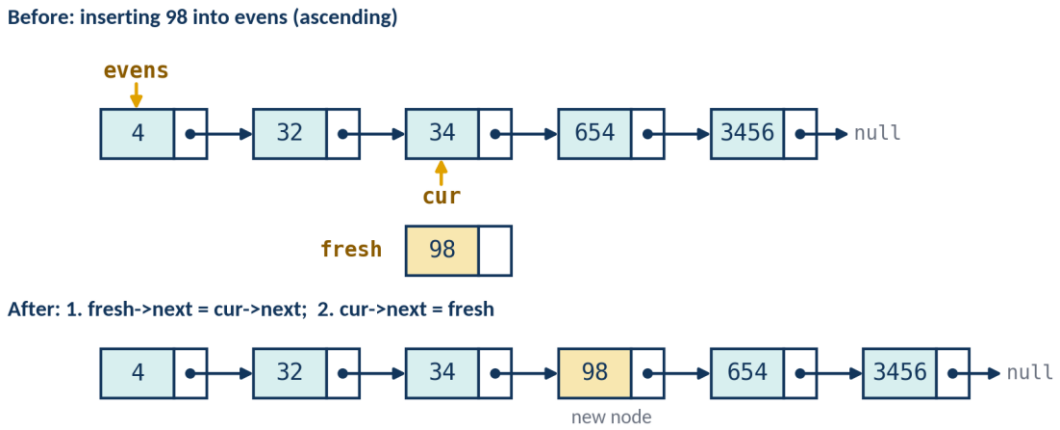


Figure 8.4 Inserting 98 into the ascending list `evens`: `cur` stops on the last node that stays in front (34); the new node is linked to 654 first, and only then is 34 linked to the new node.

Why must the two assignments come in that order? If we first wrote `cur->next = fresh`, the only pointer to 654 would be overwritten, and 654 and 3456 would be lost: unreachable, and in C++ leaked. The rule of Section 8.1 again.

Why does the loop compare with `cur->next` and not with `cur`? Because in a singly linked list we cannot go back. If `cur` walked *onto* 654 before we noticed that 98 belongs in front of it, we would have no way back to 34, whose `next` is the pointer we must change. Looking one node ahead keeps `cur` on the predecessor, where the change happens. This "stand on the predecessor" pattern appears in almost every linked-list insertion and deletion.

Why `!before(value, cur->next->data)` and not `before(cur->next->data, value)`? The two differ only for *equal* values. With our condition, a value equal to one already in the list keeps moving and is inserted *after* its equal. That keeps equal values in the order they arrived, which makes the sort *stable*; the second input below contains two 4s to test it.

Table 8.5 applies the procedure to all ten values of the sample.

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

Cost. Inserting into a list of k nodes looks at most at all k of them, so in the worst case (every value larger than all before it, in an ascending list) the n insertions look at $0 + 1 + 2 + \cdots + (n - 1)$ nodes:

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2). \tag{8.3}$$

For the ten values of the sample Equation (8.3) gives at most 45 steps, which is nothing; for a million values it would be half a trillion, and a program that must sort a million numbers uses merge sort instead (Chapter 3, and Problem 8.11 below shows its central step on linked lists). Printing costs one step per node. The memory is one node per value and nothing else, which is exactly what the problem asked for.

8.5.4 The C++ solution

Listing 8.7 `p803_evens_odds.cpp`: Problem 8.3 with two sorted linked lists and one insertion function that receives the order as a function pointer.

```
// p803_evens_odds.cpp - Problem 8.3: evens ascending, then odds descending, with linked
lists only.
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node *next;
};

bool ascending(int a, int b) { return a < b; }    // should a come before b?
bool descending(int a, int b) { return a > b; }

// Inserts value into the list whose first node is head, keeping the order
// given by the function before(). head is a REFERENCE to the caller's pointer,
// so the function can change which node is first.
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
}

void printAndFree(Node *&head) {
    while (head != nullptr) {
        cout << head->data << "\n";
        Node *gone = head;
        head = head->next;                // move on first ...
        delete gone;                      // ... then free the old first node
    }
}
```

Listing 8.7 (Continued) *p803_evens_odds.cpp: Problem 8.3 with two sorted linked lists and one insertion function that receives the order as a function pointer.*

```
int main() {
    Node *evens = nullptr;                  // two empty lists
    Node *odds = nullptr;
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int x;
        if (!(cin >> x)) break;
        if (x % 2 == 0) {                   // even, also for negative numbers
            insertSorted(evens, x, ascending);
        } else {
            insertSorted(odds, x, descending);
        }
    }
    printAndFree(evens);
    printAndFree(odds);
    return 0;
}
```

The sample input, and then a second input of seven values with negative numbers, a zero and a repeated 4:

```
$ ./p803 < p803.txt
4
32
34
98
654
3456
6789
567
543
87
$ ./p803 < p803_check.txt
-2
0
4
4
9
-3
-7
```

Line by line. `insertSorted` receives `Node *&head`, a *reference to a pointer*. Read the type from the right: `head` is a reference (`&`) to a `Node *`. Inside the function, `head` is therefore another name for the caller's variable `evens` or `odds`, so `head = fresh;` changes the caller's pointer. With a plain `Node *head` the function would change its own copy, the new first node would never reach `main`, and every value smaller than the current first would silently vanish (and leak). `printAndFree` prints the list and frees it in the same walk: it moves `head` forward *before* deleting the node it just left, so that the rest of the list is always reachable. At the end both `evens` and `odds` are `nullptr` and every node has been freed. In the second run, -7, -3 and 9 are odd and -2, 0, 4, 4 are even: zero is even, because $0 \bmod 2 = 0$.

8.5.5 The Java and Python solutions

Java has no function pointers and no references to variables, and it does not need either. The order is passed as an object that has one method: **Listing 8.8** declares a tiny *functional interface* `Before` with the method `test(a, b)`, and the two orders are *lambda expressions*, `(a, b) -> a < b` and `(a, b) -> a > b`. Since Java cannot change the caller's variable, `insertSorted` returns the

(possibly new) first node and `main` writes `evens = insertSorted(evens, x, ASCENDING);`. Python passes functions as ordinary values, so **Listing 8.9** simply hands over `ascending` or `descending`. All three programs print the same lines.

Listing 8.8 `P803EvensOdds.java`: Problem 8.3 in Java, with the order passed as a lambda expression.

```
// P803EvensOdds.java - Problem 8.3: evens ascending, then odds descending, with linked
lists only.
public class P803EvensOdds {

    static class Node {
        int data;
        Node next;

        Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    // "Should a come before b?" - a function that can be passed as an argument.
    interface Before {
        boolean test(int a, int b);
    }

    static final Before ASCENDING = (a, b) -> a < b;
    static final Before DESCENDING = (a, b) -> a > b;

    // Java cannot change the caller's variable, so the (possibly new) first node is
returned.
    static Node insertSorted(Node head, int value, Before before) {
        Node fresh = new Node(value, null);
        if (head == null || before.test(value, head.data)) {
            fresh.next = head;                  // the new node becomes the first
            return fresh;
        }
        Node cur = head;                        // find the last node that stays in front
        while (cur.next != null && !before.test(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
        return head;
    }

    static void print(Node head, StringBuilder out) {
        for (Node p = head; p != null; p = p.next) {
            out.append(p.data).append('\n');
        }
    }
```

Listing 8.8 (Continued) *P803EvensOdds.java: Problem 8.3 in Java, with the order passed as a lambda expression.*

```
    public static void main(String[] args) throws java.io.IOException {
        String text = new String(System.in.readAllBytes()).trim();
        String[] tok = text.isEmpty() ? new String[0] : text.split("\\s+");
        Node evens = null;                    // two empty lists
        Node odds = null;
        int n = tok.length > 0 ? Integer.parseInt(tok[0]) : 0;
        for (int i = 1; i <= n && i < tok.length; i++) {
            int x = Integer.parseInt(tok[i]);
            if (x % 2 == 0) {                 // even, also for negative numbers
                evens = insertSorted(evens, x, ASCENDING);
            } else {
                odds = insertSorted(odds, x, DESCENDING);
            }
        }
        StringBuilder out = new StringBuilder();
        print(evens, out);
        print(odds, out);
        System.out.print(out);
    }
}
```

Listing 8.9 `p803_evens_odds.py`: Problem 8.3 in Python, with the order passed as a function.

```
# p803_evens_odds.py - Problem 8.3: evens ascending, then odds descending, with linked
lists only.
import sys


class Node:
    __slots__ = ("data", "next")

    def __init__(self, data, next=None):
        self.data = data
        self.next = next


def ascending(a, b):        # should a come before b?
    return a < b


def descending(a, b):
    return a > b


def insert_sorted(head, value, before):
    """Returns the (possibly new) first node of the list, kept in the order of before()."""
    fresh = Node(value)
    if head is None or before(value, head.data):
        fresh.next = head               # the new node becomes the first
        return fresh
    cur = head                          # find the last node that stays in front
    while cur.next is not None and not 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
    return head


def main():
    tok = sys.stdin.read().split()
    evens = None                        # two empty lists
    odds = None
    n = int(tok[0]) if tok else 0
    for t in tok[1:n + 1]:
        x = int(t)
        if x % 2 == 0:                  # even; in Python -7 % 2 is 1, in C++ and Java -1
            evens = insert_sorted(evens, x, ascending)
        else:
            odds = insert_sorted(odds, x, descending)
    out = []
```

Listing 8.9 (Continued) *p803_evens_odds.py: Problem 8.3 in Python, with the order passed as a function.*

```
    for head in (evens, odds):
        p = head
        while p is not None:
            out.append(str(p.data))
            p = p.next
    if out:
        print("\n".join(out))


if __name__ == "__main__":
    main()
```

8.5.6 A tempting wrong solution

Many people think of “odd” first and write the split like this:

```
    if (x % 2 == 1) insertSorted(odds, x, descending);    // WRONG for negative x
    else            insertSorted(evens, x, ascending);
```

On the sample input it prints the right answer, because every number there is positive. On the second input the program `wrong/p803_wrong.cpp` prints

```
-7
-3
-2
0
4
4
9
```

The odd numbers -7 and -3 have joined the evens, and only 9 is left among the odds. The reason is the way C++ divides negative numbers. Since C++11 the quotient of `/` is *truncated toward zero*, so $-7/2$ is -3, and the remainder must satisfy $(-7/2)\cdot 2 + (-7 \,\%\, 2) = -7$, which makes `-7 % 2` equal to -1, not 1 [9]. Java defines `%` the same way. Python instead rounds its quotient *down*, so in Python `-7 % 2` is 1 and the wrong test happens to work, as the comment in **Listing 8.9** notes [8]. The safe habit, in every language, is to test parity with `x % 2 == 0` and let "odd" be everything else.

8.5.7 Check yourself

> **CHECK YOURSELF.** Give the output of **Listing 8.7** for $N = 6$ and the values 5, 0, -4, 15, -9, 8, and then the output of the wrong program of Section 8.5.6 for the same input. How many nodes does the correct program create, and how many does it free?
>
> *Answer.* Correct: -4, 0, 8, then 15, 5, -9. Wrong: -9, -4, 0, 8, then 15, 5, because `-9 % 2` is -1 and -9 is filed as even. Both create six nodes, one per value, and the correct program frees all six in `printAndFree`.

8.6 Problem 8.4 — Reversing the Words of a Sentence

8.6.1 The problem

> **PROBLEM.** Write a C++ program that reverses a string using a linked list. Sample input: `practice makes a man perfect`. Sample output: `perfect man a makes practice`.

8.6.2 In our own words

The sample shows what "reverse" means here: the *words* come out in reverse order, and each word keeps its own spelling. We read one line, cut it into words at the spaces, and print the words last-first, separated by single spaces.

Which structure? "The last word read is the first word printed" is last in, first out: a stack. And a stack is easy to build from a linked list, because Chapter 7's `frontInsertion` *is* a push: the new node goes in front of the current first node, and `head` moves to the new node. Reading the list from `head` to the end then gives the words in reverse order of insertion, which is exactly what we want. Nothing needs to be reversed afterwards; the reversal happens during the building.

8.6.3 By hand, step by step

Figure 8.5 shows the list after the first word, after the second, and after all five.

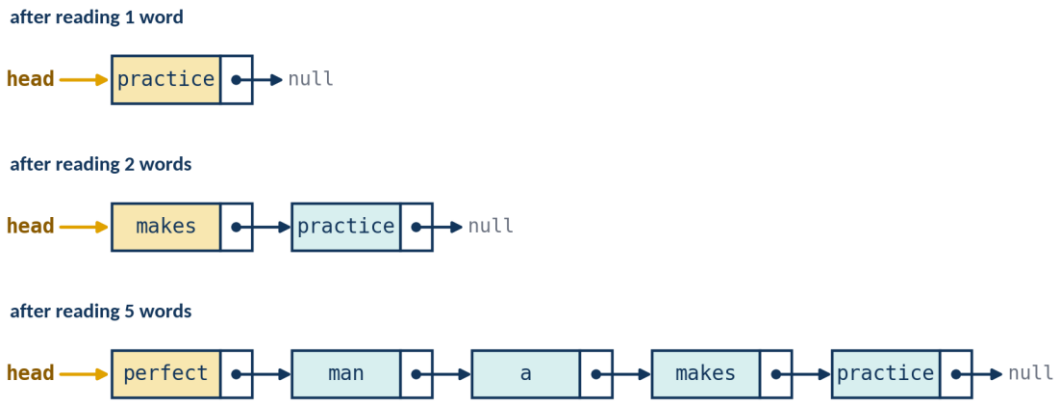


Figure 8.5 Building the list of Problem 8.4 by front insertion: each new word becomes the first node, so the list reads the sentence backward.

1. The list is empty: `head` is `nullptr`.
2. Read `practice`. Create a node whose `next` is the current `head` (`nullptr`), and make `head` point to it. The list is `practice`.
3. Read `makes`. Create a node whose `next` is the current `head` (the `practice` node), and move `head` to it. The list is `makes`, `practice`.
4. Read `a`, `man`, `perfect` the same way. The list is `perfect`, `man`, `a`, `makes`, `practice`.
5. Walk from `head` to `nullptr`, printing each word, with a space *between* words: `perfect man a makes practice`.

In C++ steps 2 and 3 are one line, `head = new WordNode{w, head};`. The right-hand side is evaluated first, so the new node receives the *old* value of `head` as its `next`, and only then is `head` changed to the new node. Once again the order is what makes it work.

Cutting the line into words. `getline(cin, line)` reads the whole line, spaces included (a plain `cin >> line` would stop at the first space). An `istringstream` then lets us read words out of that string with `>>`, exactly as we read numbers from the keyboard, and `>>` skips any number of spaces between words. So several spaces between words, or spaces at the start and the end of the line, cause no empty words. The second input below tests that.

Cost. Every word is pushed once and printed once, and every character is copied a constant number of times, so the program runs in time proportional to the length of the line, $\Theta(L)$ for a line of L characters. The memory is one node per word plus the characters themselves.

8.6.4 The C++ solution

Listing 8.10 `p804_reverse_words.cpp`: Problem 8.4 with a linked list used as a stack.

```cpp
// p804_reverse_words.cpp - Problem 8.4: reverse the words of a sentence with a linked
list.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

struct WordNode {
    string word;
    WordNode *next;
};

// Front insertion: the new node goes before the current first node.
void pushFront(WordNode *&head, const string &w) {
    head = new WordNode{w, head};          // new node points to the old first node
}

int main() {
    string line;
    getline(cin, line);                    // the whole sentence, spaces included

    WordNode *head = nullptr;
    istringstream words(line);
    string w;
    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;
    }
    cout << "\n";
    return 0;
}
```

```
$ ./p804 < p804.txt
perfect man a makes practice
$ ./p804 < p804_check.txt
friends are queues and stacks
```

Line by line. `pushFront` takes `WordNode *&head`, a reference to the caller's pointer, for the same reason as `insertSorted` in Problem 8.3: it must change which node is first. The printing loop uses a flag `first` to put a space before every word except the first, so the line has no space at its end; a trailing space is invisible on the screen and yet makes the output differ from the expected output. The loop frees each node right after printing it, moving `head` on before the `delete`. The second input, `stacks  and queues are  friends` with extra spaces everywhere, comes out as `friends are queues and stacks`.

8.6.5 The Java and Python solutions

Listing 8.11 reads the line with `BufferedReader.readLine()` and cuts it with `split("\\s+")`, a regular expression meaning "one or more whitespace characters", after `trim()` has removed the spaces at both ends. **Listing 8.12** uses `line.split()`, which with no argument splits at any run of whitespace and ignores it at the ends. In both, the node constructor takes the old `head` as its `next`,

which is the front insertion of the C++ version, and in Python `" ".join(words)` puts the spaces between the words only.

Listing 8.11 `P804ReverseWords.java`: Problem 8.4 in Java.

```
// P804ReverseWords.java - Problem 8.4: reverse the words of a sentence with a linked list.
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class P804ReverseWords {

    static class WordNode {
        String word;
        WordNode next;

        WordNode(String word, WordNode next) {
            this.word = word;
            this.next = next;
        }
    }

    public static void main(String[] args) throws java.io.IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line = in.readLine();              // the whole sentence, spaces included
        if (line == null) line = "";

        WordNode head = null;
        for (String w : line.trim().split("\\s+")) {
            if (!w.isEmpty()) {
                head = new WordNode(w, head);     // front insertion: last word read ends
up first
            }
        }

        StringBuilder out = new StringBuilder();
        for (WordNode p = head; p != null; p = p.next) {
            if (p != head) out.append(' ');       // a space BETWEEN words, never at the
end
            out.append(p.word);
        }
        System.out.println(out);
    }
}
```

Listing 8.12 `p804_reverse_words.py`: Problem 8.4 in Python.

```python
# p804_reverse_words.py - Problem 8.4: reverse the words of a sentence with a linked list.
import sys


class WordNode:
    __slots__ = ("word", "next")

    def __init__(self, word, next):
        self.word = word
        self.next = next


def main():
    line = sys.stdin.readline()             # the whole sentence, spaces included
    head = None
    for w in line.split():                  # one word at a time, spaces skipped
        head = WordNode(w, head)            # front insertion: the last word read ends up
first
    words = []
    p = head
    while p is not None:
        words.append(p.word)
        p = p.next
    print(" ".join(words))                  # a space BETWEEN words, never at the end


if __name__ == "__main__":
    main()
```

8.6.6 A tempting wrong solution

Programmers who learned C first often read the line into a fixed character array and cut it with `strtok`:

```
   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';
```

On the sample it works, apart from one detail: the version in the code folder prints each word followed by a space, so its output line ends in a space that the expected output does not have. The real problem appears with a sentence of 100 characters or more. Then `str[i++]` writes past the end of the array into whatever memory happens to follow it. Nothing obliges the program to crash; it may print nonsense, or print the right answer and corrupt a variable that is used later. This is exactly what the sanitizers of Section 8.2 are for. Fed a 104-character sentence, the program `wrong/p804_wrong.cpp`, built with `-fsanitize=address,undefined`, stops at the first bad write; the first lines of its report are (the addresses are shortened here, because they change from run to run):

```
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
```

The report names the kind of error (a write past the end of an array on the stack), its size (one byte), and the exact line. The repair is to let the string grow: `std::string` with `getline` has no fixed limit. Chapter 6's lesson about arrays, that C++ never checks an index for you, is the reason this whole class of bug exists.

8.6.7 Check yourself

> **CHECK YOURSELF.** What does **Listing 8.10** print for a line containing one word, `queue`, and for an empty line? How many nodes are created in each case? Then: if `pushFront` were replaced by Chapter 7's `backInsertion`, what would the sample print?
>
> *Answer.* `queue`, from a list of one node; an empty line produces no words, no nodes, and prints just the end of the line. With `backInsertion` the words keep their order, so the program prints the sentence unchanged, `practice makes a man perfect`: the reversal comes *only* from inserting at the front.

8.7 Problem 8.5 — Removing Duplicate Values From a List

8.7.1 The problem

> **PROBLEM.** Complete the following C++ code, where `remove_duplicate()` is a function that removes the nodes with the same data from the list. Sample input: 1, 2, 3, 4, 1, 5, 6, 3, 2, −1 (one value per line). Sample output: the list before the removal, `1 2 3 4 1 5 6 3 2 -1`, and after it, `1 2 3 4 5 6`.

The given `main` is:

```
int main() {
    linkedList *myList = new linkedList;
    myList->_head = NULL;
    int data;
    do {
        cin >> data;
        slist_pushBack(myList, data);
    } while (data != -1);
    print_linkedList(myList);
    remove_duplicate(myList);
    print_linkedList(myList);
    return 0;
}
```

8.7.2 In our own words

The list is built from the input in order. We must delete every node whose value has already appeared *earlier* in the list, and keep the first appearance of every value, so that the survivors are still in their original order: 1, 2, 3, 4, 5, 6.

One detail of the sample deserves a careful look, because it looks like a contradiction. Like the loop of Problem 8.2, the given reading loop is a `do`–`while`, so the end marker −1 is pushed into the list before the loop notices it; that is why the first printed line ends in `-1`. But the second line has no −1, although −1 appears only once and is therefore not a duplicate. The two lines are consistent only if the end marker is removed as well. The honest way to say this is: −1 *is not data, it is a signal that the data has ended*, and it should not survive into the cleaned list. Our program keeps the given reading loop, so that the first line matches, and after `remove_duplicate` it removes the last node, which is the marker, with a small `slist_popBack` (Chapter 7's `backDeletion`). `remove_duplicate` itself does only what its name says. A cleaner design would never store the marker in the first place; Exercise 8.12 asks for that version.

8.7.3 By hand, step by step

We use two pointers. `cur` walks along the list, one node at a time; it marks the value we are currently looking for. For each position of `cur`, a second pointer, `runner`, starts at `cur` and walks to

the end of the list, looking at `runner->next` each time. When `runner->next` holds the same value as `cur`, that node is a later copy, and we delete it.

Why does `runner` look at `runner->next` and not at itself? For the reason we met in Problem 8.3: to delete a node from a singly linked list we must change the `next` pointer of the node *before* it, so the pointer that does the work must stand on the predecessor. **Figure 8.6** shows the moment when `cur` is on the first 1 and `runner` is on 4, whose successor is the second 1.

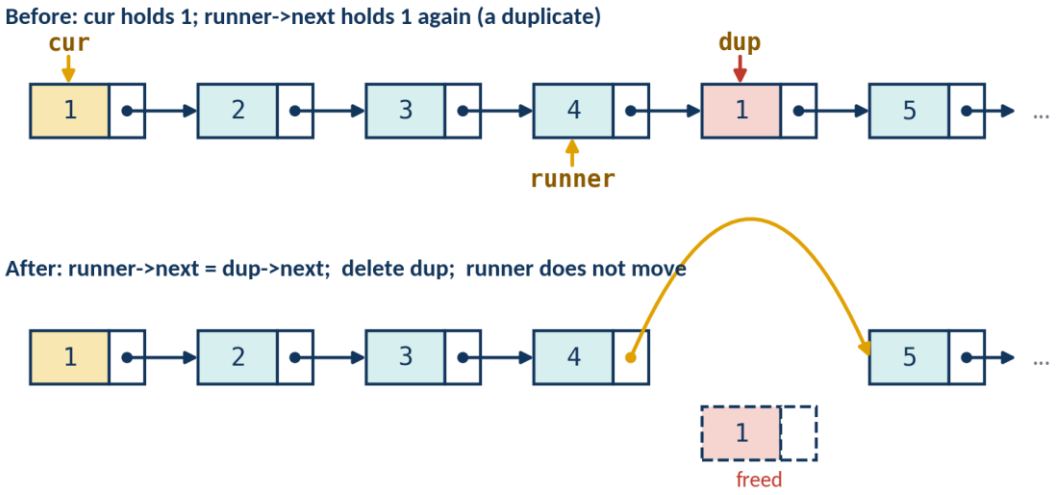


Figure 8.6 Deleting a later copy: `runner` stands on the predecessor of the duplicate, links around it, and only then frees it; `runner` does not move, because its new successor has not been examined yet.

The deletion is three statements, and their order matters:

1. `SListNode *dup = runner->next;` gives the doomed node a name, so we can still reach it.
2. `runner->next = dup->next;` links around it: 4 now points to 5. The list no longer contains `dup`.
3. `delete dup;` frees its memory. Doing this *before* step 2 would make step 2 read `dup->next` from freed memory, a *use after free*, the error of Problem 8.12's wrong solution.

After a deletion `runner` must **stay where it is**, because its `next` is now a node it has not examined. After a comparison that deletes nothing, `runner` moves one node on. On the sample the whole process is short:

- `cur` = 1: `runner` finds the 1 in the fifth position and deletes it. The list is 1 2 3 4 5 6 3 2 −1.
- `cur` = 2: deletes the 2 near the end. The list is 1 2 3 4 5 6 3 −1.
- `cur` = 3: deletes the last 3. The list is 1 2 3 4 5 6 −1.
- `cur` = 4, 5, 6, −1: nothing is deleted.
- `slist_popBack` deletes the end marker. The list is 1 2 3 4 5 6.

Cost. For each position of `cur`, `runner` looks at every node after it. With n nodes and no duplicates at all (the worst case, since nothing gets shorter), that is $(n-1)+(n-2)+\cdots+1+0$ comparisons:

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2) \text{ comparisons, and } O(1) \text{ extra memory.} \tag{8.4}$$

The same sum as Equation (8.3), for the same reason: every element is compared with every element on one side of it. There is a faster way, which trades memory for time. Walk the list once and keep a *set* of the values seen so far (`std::unordered_set` in C++, `HashSet` in Java, `set` in

Python); a node whose value is already in the set is deleted. Each set lookup takes $O(1)$ time on average, so the walk costs $O(n)$ time and $O(n)$ extra memory. How a hash set achieves $O(1)$ is the subject of Chapter 15; for a list of ten values the quadratic method is perfectly good, and it needs nothing beyond Chapter 7.

8.7.4 The C++ solution

Listing 8.13 keeps the given names and `main`, adding only the call to `slist_popBack`, a final line with the node count, and the cleanup.

Listing 8.13 `p805_remove_duplicates.cpp`: Problem 8.5 with the two-pointer removal, plus a `slist_popBack` that drops the end marker.

```
// p805_remove_duplicates.cpp - Problem 8.5: remove the nodes whose value appeared earlier.
#include <iostream>
using namespace std;

typedef struct snode_t {
    int data;
    struct snode_t *next;
} SListNode;

typedef struct slist_t {
    unsigned _size;
    SListNode *_head;
} linkedList;

bool slist_isEmpty(linkedList *myList) {
    return (myList->_head == nullptr);
}

void slist_pushBack(linkedList *myList, int value) {
    SListNode *newNode = new SListNode{value, nullptr};
    if (slist_isEmpty(myList)) {
        myList->_head = newNode;
    } else {
        SListNode *last = myList->_head;
        while (last->next != nullptr) {
            last = last->next;
        }
        last->next = newNode;
    }
    myList->_size++;
}

// Removes the last node (used below for the end marker -1).
void slist_popBack(linkedList *myList) {
    if (slist_isEmpty(myList)) return;
    if (myList->_head->next == nullptr) {  // only one node
        delete myList->_head;
        myList->_head = nullptr;
    } else {
        SListNode *aux = myList->_head;    // stop at the node BEFORE the last
        while (aux->next->next != nullptr) {
            aux = aux->next;
        }
        delete aux->next;                  // free the last node
        aux->next = nullptr;               // aux is the new last node
    }
```

Listing 8.13 (Continued) *p805_remove_duplicates.cpp: Problem 8.5 with the two-pointer removal, plus a slist_popBack that drops the end marker.*

```
    myList->_size--;
}

void print_linkedList(linkedList *myList) {
    for (SListNode *p = myList->_head; p != nullptr; p = p->next) {
        cout << p->data << (p->next != nullptr ? " " : "");
    }
    cout << "\n";
}

// Keeps the FIRST node of every value and deletes every later node with the same value.
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
            }
        }
    }
}

void slist_destroy(linkedList *myList) {
    while (!slist_isEmpty(myList)) {
        SListNode *gone = myList->_head;
        myList->_head = gone->next;
        delete gone;
    }
    myList->_size = 0;
}
```

Listing 8.13 (Continued) *p805_remove_duplicates.cpp: Problem 8.5 with the two-pointer removal, plus a slist_popBack that drops the end marker.*

```
int main() {
    linkedList *myList = new linkedList;
    myList->_size = 0;
    myList->_head = nullptr;
    int data;
    do {
        if (!(cin >> data)) break;         // input ended without -1
        slist_pushBack(myList, data);      // note: the end marker -1 is stored too
    } while (data != -1);
    print_linkedList(myList);
    remove_duplicate(myList);
    slist_popBack(myList);                 // drop the end marker, which is not data
    print_linkedList(myList);
    cout << "nodes left: " << myList->_size << "\n";
    slist_destroy(myList);
    delete myList;
    return 0;
}
```

```
$ ./p805 < p805.txt
1 2 3 4 1 5 6 3 2 -1
1 2 3 4 5 6
nodes left: 6
$ ./p805 < p805_check.txt
7 7 7 2 7 2 -1
7 2
nodes left: 2
```

Line by line. `slist_pushBack` is Chapter 7's `backInsertion` for a list that keeps only a head: an empty list gets the new node as its head; otherwise an auxiliary pointer walks to the last node and links the new node after it. `slist_popBack` has the two cases of Chapter 7's `backDeletion`: a one-node list becomes empty, and a longer list is walked until `aux` stands on the node *before* the last (`aux->next->next == nullptr`), whose successor is freed and whose `next` becomes `nullptr`. `print_linkedList` prints a space only between values. `remove_duplicate` is the procedure of Section 8.7.3; notice that the `else` branch is the only place where `runner` moves. Finally `main` prints the size as well, as a cross-check that the counter `_size` was kept up to date through every deletion, and frees everything with `slist_destroy`. The second input, `7 7 7 2 7 2 -1`, has three copies in a row, the case that catches the wrong solution below.

8.7.5 The Java and Python solutions

In Java and Python, deleting a node is only step 2, `runner.next = runner.next.next`: once nothing refers to the unlinked node, the garbage collector reclaims it, so there is no step 3 and no way to use a node after freeing it. The algorithm is otherwise identical, and **Listing 8.14** and **Listing 8.15** print the same lines.

Listing 8.14 `P805RemoveDuplicates.java`: Problem 8.5 in Java.

```java
// P805RemoveDuplicates.java - Problem 8.5: remove the nodes whose value appeared earlier.
public class P805RemoveDuplicates {

    static class Node {
        int data;
        Node next;

        Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    static class LinkedList {
        int size;
        Node head;
    }

    static boolean isEmpty(LinkedList list) {
        return list.head == null;
    }

    static void pushBack(LinkedList list, int value) {
        Node newNode = new Node(value, null);
        if (isEmpty(list)) {
            list.head = newNode;
        } else {
            Node last = list.head;
            while (last.next != null) {
                last = last.next;
            }
            last.next = newNode;
        }
        list.size++;
    }
```

Listing 8.14 (Continued) *P805RemoveDuplicates.java: Problem 8.5 in Java.*

```
    // Removes the last node (used below for the end marker -1).
    static void popBack(LinkedList list) {
        if (isEmpty(list)) return;
        if (list.head.next == null) {          // only one node
            list.head = null;
        } else {
            Node aux = list.head;              // stop at the node BEFORE the last
            while (aux.next.next != null) {
                aux = aux.next;
            }
            aux.next = null;                   // aux is the new last node
        }
        list.size--;
    }

    static void print(LinkedList list) {
        StringBuilder out = new StringBuilder();
        for (Node p = list.head; p != null; p = p.next) {
            out.append(p.data).append(p.next != null ? " " : "");
        }
        System.out.println(out);
    }

    // Keeps the FIRST node of every value and unlinks every later node with the same
value.
    static void removeDuplicate(LinkedList list) {
        for (Node cur = list.head; cur != null; cur = cur.next) {
            Node runner = cur;                 // runner.next is the node being examined
            while (runner.next != null) {
                if (runner.next.data == cur.data) {
                    runner.next = runner.next.next;   // bridge over the duplicate
                    list.size--;                       // runner stays: its NEW next is
unexamined
                } else {
                    runner = runner.next;              // nothing removed: move on
                }
            }
        }
    }
```

Listing 8.14 (Continued) *P805RemoveDuplicates.java: Problem 8.5 in Java.*

```
    public static void main(String[] args) throws java.io.IOException {
        String text = new String(System.in.readAllBytes()).trim();
        String[] tok = text.isEmpty() ? new String[0] : text.split("\\s+");
        LinkedList myList = new LinkedList();
        int i = 0;
        int data;
        do {
            if (i >= tok.length) break;        // input ended without -1
            data = Integer.parseInt(tok[i++]);
            pushBack(myList, data);            // note: the end marker -1 is stored too
        } while (data != -1);
        print(myList);
        removeDuplicate(myList);
        popBack(myList);                       // drop the end marker, which is not data
        print(myList);
        System.out.println("nodes left: " + myList.size);
    }
}
```

Listing 8.15 `p805_remove_duplicates.py`: Problem 8.5 in Python.

```python
# p805_remove_duplicates.py - Problem 8.5: remove the nodes whose value appeared earlier.
import sys


class Node:
    __slots__ = ("data", "next")

    def __init__(self, data, next=None):
        self.data = data
        self.next = next


class LinkedList:
    def __init__(self):
        self.size = 0
        self.head = None


def push_back(lst, value):
    new_node = Node(value)
    if lst.head is None:
        lst.head = new_node
    else:
        last = lst.head
        while last.next is not None:
            last = last.next
        last.next = new_node
    lst.size += 1


def pop_back(lst):
    """Removes the last node (used below for the end marker -1)."""
    if lst.head is None:
        return
    if lst.head.next is None:           # only one node
        lst.head = None
    else:
        aux = lst.head                  # stop at the node BEFORE the last
        while aux.next.next is not None:
            aux = aux.next
        aux.next = None                 # aux is the new last node
    lst.size -= 1
```

Listing 8.15 (Continued) *p805_remove_duplicates.py: Problem 8.5 in Python.*

```
def print_list(lst):
    values = []
    p = lst.head
    while p is not None:
        values.append(str(p.data))
        p = p.next
    print(" ".join(values))


def remove_duplicate(lst):
    """Keeps the FIRST node of every value and unlinks every later node with the same
value."""
    cur = lst.head
    while cur is not None:
        runner = cur                    # runner.next is the node being examined
        while runner.next is not None:
            if runner.next.data == cur.data:
                runner.next = runner.next.next  # bridge over the duplicate
                lst.size -= 1                   # runner stays: its NEW next is unexamined
            else:
                runner = runner.next            # nothing removed: move on
        cur = cur.next


def main():
    my_list = LinkedList()
    for t in sys.stdin.read().split():
        data = int(t)
        push_back(my_list, data)        # note: the end marker -1 is stored too
        if data == -1:
            break
    print_list(my_list)
    remove_duplicate(my_list)
    pop_back(my_list)                   # drop the end marker, which is not data
    print_list(my_list)
    print("nodes left:", my_list.size)


if __name__ == "__main__":
    main()
```

8.7.6 A tempting wrong solution

The natural way to write a walk is “look at the next node, then move on”, every time:

```
  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
```

It passes the sample, because in the sample no two copies of a value stand next to each other. On `7 7 7 2 7 2 -1` the program `wrong/p805_wrong.cpp` prints

```
7 7 7 2 7 2 -1
7 7 2
nodes left: 3
```

Follow it: `cur` is on the first 7 and `runner` too. The second 7 is deleted, and `runner->next` is now the third 7, but the wrong line moves `runner` onto it without comparing it with `cur`. The third 7 escapes. The rule is simple once said: *after a deletion, the node behind `runner->next` is new, so*

look at it before moving. The edge case that exposes the error is "two copies next to each other", which is on the list of Step 5 of the method for exactly this reason.

8.7.7 Check yourself

> **CHECK YOURSELF.** How many comparisons `runner->next->data == cur->data` does `remove_duplicate` make on the sample list of ten nodes (including −1)? Count them by hand.
>
> *Answer.* With `cur` on the first node, `runner` examines all 9 later nodes, and one of them is deleted, leaving 9 nodes. With `cur` on the second node, there are 7 later nodes (one deleted, leaving 8); on the third, 5 later nodes (one deleted, leaving 7); then the fourth, fifth, sixth and seventh nodes have 3, 2, 1 and 0 later nodes. The total is $9 + 7 + 5 + 3 + 2 + 1 + 0 = 27$, less than the $\frac{10 \cdot 9}{2} = 45$ of Equation (8.4), because each deletion shortens every later scan.

8.8 New Worked Problems on Stacks and Queues

The five problems so far came as one set. The seven that follow are new, and each aims at one idea of Chapters 5 to 7 that the set touched only lightly. They follow the same method, a little more briskly, with the labels of Section 8.2 in bold.

8.8.1 Problem 8.6 — Matching brackets

> **PROBLEM.** Write a function that decides whether the brackets `()`, `[]` and `{}` in a line of text are balanced: every opening bracket is closed by a closing bracket of the same kind, and the pairs are properly nested. If they are not, say what is wrong and at which position (counting the characters from 1). Test it on `(a+b)*[c-d]`, `{[()()]}`, `([)]`, `((x)`, `x)]` and the empty line.

In our own words. Reading from left to right, every closing bracket must match the *most recent* opening bracket that is still open. "Most recent still open" is last in, first out: a stack. Push each opening bracket (and its position, for the error message); at each closing bracket, the top of the stack must be its partner, and we pop it. There are exactly three ways to fail: a closing bracket arrives when the stack is empty (it has no partner), a closing bracket does not match the top (the pairs cross), or the text ends with the stack not empty (something was never closed).

By hand. For `([)]`: position 1 `(` is pushed; position 2 `[` is pushed, and the stack is `(`, `[` with `[` on top; position 3 is `)`, but the top is `[`, so `)` cannot close it, and we report `')' at 3 cannot close '[' from 2`. For `{[()()]}` the stack grows to three brackets and shrinks back to empty, every closer meeting its partner on top, so the line is balanced. For `((x)` the `)` at position 4 closes the `(` from position 2, and at the end the `(` from position 1 is still on the stack.

Design. The stack is an array stack like that of Problem 8.1, but its capacity is chosen when the line is known: a line of L characters can never contain more than L opening brackets. C++ allocates such an array with `new char[capacity]` and must release it with `delete[]` (with brackets, because it is an array). We let the stack's *destructor* do the release, so that it happens automatically when `check` returns, on every path, including the three early `return`s of **Listing 8.16**. Because the object owns its arrays, copying it would make two objects that both believe they own the same arrays and both delete them; the two `= delete` lines forbid copying, so that mistake cannot even compile.

Listing 8.16 `p806_brackets.cpp`: Problem 8.6, bracket matching with an array stack whose capacity is chosen at run time.

```cpp
// p806_brackets.cpp - Problem 8.6: check the brackets of an expression with a stack.
#include <iostream>
#include <string>
using namespace std;

// An array stack whose capacity is chosen at run time (new[] / delete[]).
struct BracketStack {
    char *symbol;     // the opening brackets
    int *where;       // their positions in the text (1-based)
    int top;          // -1 means empty

    explicit BracketStack(int capacity)
        : symbol(new char[capacity]), where(new int[capacity]), top(-1) {}
    ~BracketStack() {
        delete[] symbol;
        delete[] where;
    }
    BracketStack(const BracketStack &) = delete;             // no copies:
    BracketStack &operator=(const BracketStack &) = delete;  // one owner per array

    bool isEmpty() const { return top < 0; }
    void push(char c, int pos) {
        ++top;
        symbol[top] = c;
        where[top] = pos;
    }
    void pop() { --top; }
};

char partnerOf(char close) {
    if (close == ')') return '(';
    if (close == ']') return '[';
    return '{';
}

// Returns "balanced" or a sentence that says what is wrong, and where.
string check(const string &text) {
    BracketStack st(static_cast<int>(text.size()) + 1);  // never more openers than
characters
    for (size_t i = 0; i < text.size(); i++) {
        char c = text[i];
        int pos = static_cast<int>(i) + 1;
        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";
```

Listing 8.16 (Continued) *p806_brackets.cpp: Problem 8.6, bracket matching with an array stack whose capacity is chosen at run time.*

```
            }
            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();
        }
    }
    if (!st.isEmpty()) {
        return string("'") + st.symbol[st.top] + "' at " + to_string(st.where[st.top]) +
               " is never closed";
    }
    return "balanced";
}

int main() {
    const string tests[] = {"(a+b)*[c-d]", "{[()()]}", "([)]", "((x)", "x)]", ""};
    for (const string &t : tests) {
        cout << "\"" << t << "\" -> " << check(t) << "\n";
    }
    return 0;
}
```

```
"(a+b)*[c-d]" -> balanced
"{[()()]}" -> balanced
"([)]" -> ')' at 3 cannot close '[' from 2
"((x)" -> '(' at 1 is never closed
"x)]" -> ')' at 2 has no opening partner
"" -> balanced
```

The Java and Python programs, **Listing 8.17** and **Listing 8.18**, print the same six lines. Java arrays are created with `new char[capacity]` too, but they are freed by the garbage collector; Python uses lists of fixed length and a dictionary `PARTNER` that maps each closing bracket to its opening partner.

Listing 8.17 `P806Brackets.java`: Problem 8.6 in Java.

```
// P806Brackets.java - Problem 8.6: check the brackets of an expression with a stack.
public class P806Brackets {

    // An array stack whose capacity is chosen at run time.
    static class BracketStack {
        final char[] symbol;       // the opening brackets
        final int[] where;         // their positions in the text (1-based)
        int top = -1;              // -1 means empty

        BracketStack(int capacity) {
            symbol = new char[capacity];
            where = new int[capacity];
        }

        boolean isEmpty() { return top < 0; }

        void push(char c, int pos) {
            ++top;
            symbol[top] = c;
            where[top] = pos;
        }

        void pop() { --top; }
    }

    static char partnerOf(char close) {
        if (close == ')') return '(';
        if (close == ']') return '[';
        return '{';
    }

    // Returns "balanced" or a sentence that says what is wrong, and where.
    static String check(String text) {
        BracketStack st = new BracketStack(text.length() + 1);
        for (int i = 0; i < text.length(); i++) {
            char c = text.charAt(i);
            int pos = i + 1;
            if (c == '(' || c == '[' || c == '{') {
                st.push(c, pos);
            } else if (c == ')' || c == ']' || c == '}') {
                if (st.isEmpty()) {
                    return "'" + c + "' at " + pos + " has no opening partner";
                }
                if (st.symbol[st.top] != partnerOf(c)) {
                    return "'" + c + "' at " + pos + " cannot close '"
                           + st.symbol[st.top] + "' from " + st.where[st.top];
```

Listing 8.17 (Continued) *P806Brackets.java: Problem 8.6 in Java.*

```
                }
                st.pop();
            }
        }
        if (!st.isEmpty()) {
            return "'" + st.symbol[st.top] + "' at " + st.where[st.top] + " is never
closed";
        }
        return "balanced";
    }

    public static void main(String[] args) {
        String[] tests = {"(a+b)*[c-d]", "{[()()]}", "([)]", "((x)", "x)]", ""};
        for (String t : tests) {
            System.out.println("\"" + t + "\" -> " + check(t));
        }
    }
}
```

Listing 8.18 `p806_brackets.py`: Problem 8.6 in Python.

```
# p806_brackets.py - Problem 8.6: check the brackets of an expression with a stack.
PARTNER = {")": "(", "]": "[", "}": "{"}


class BracketStack:
    """An array stack whose capacity is chosen at run time."""

    def __init__(self, capacity):
        self.symbol = [""] * capacity   # the opening brackets
        self.where = [0] * capacity     # their positions in the text (1-based)
        self.top = -1                   # -1 means empty

    def is_empty(self):
        return self.top < 0

    def push(self, c, pos):
        self.top += 1
        self.symbol[self.top] = c
        self.where[self.top] = pos

    def pop(self):
        self.top -= 1


def check(text):
    """Returns "balanced" or a sentence that says what is wrong, and where."""
    st = BracketStack(len(text) + 1)
    for i, c in enumerate(text):
        pos = i + 1
        if c in "([{":
            st.push(c, pos)
        elif c in ")]}":
            if st.is_empty():
                return f"'{c}' at {pos} has no opening partner"
            if st.symbol[st.top] != PARTNER[c]:
                return f"'{c}' at {pos} cannot close '{st.symbol[st.top]}' from
{st.where[st.top]}"
            st.pop()
    if not st.is_empty():
        return f"'{st.symbol[st.top]}' at {st.where[st.top]} is never closed"
    return "balanced"


def main():
    for t in ["(a+b)*[c-d]", "{[()()]}", "([)]", "((x)", "x)]", ""]:
        print(f'"{t}" -> {check(t)}')
```

Listing 8.18 (Continued) *p806_brackets.py: Problem 8.6 in Python.*

```
if __name__ == "__main__":
    main()
```

Cost. Each character causes at most one push or one pop, so the check is $\Theta(L)$ time and $O(L)$ memory for a line of L characters.

A tempting wrong solution. Counting seems enough: add one for an opening bracket, subtract one for a closing bracket, fail if the count goes negative, and succeed if it ends at zero. The program `wrong/p806_wrong.cpp` does this, and prints

```
"(a+b)*[c-d]" -> balanced
"{[()()]}" -> balanced
"([)]" -> balanced
"((x)" -> not balanced
"x)]" -> not balanced
"(]" -> balanced
```

A counter knows *how many* brackets are open but not *which ones*, so it accepts `([)]` and even `(]`. With only one kind of bracket a counter is enough; with several kinds the order of the open brackets matters, and remembering an order is what a stack does.

> **CHECK YOURSELF.** What does `check` return for `{(a)[b]}` and for `[(])`? Trace the stack.
>
> *Answer.* `{(a)[b]}`: the stack becomes `{`, then `{(`, back to `{`, then `{[`, back to `{`, then empty at position 8; the `}` at position 9 finds an empty stack: `'}' at 9 has no opening partner`. `[(])`: `[` and `(` are pushed; the `]` at position 3 meets `(` on top: `']' at 3 cannot close '(' from 2`.

8.8.2 Problem 8.7 — A circular queue of borrow requests

> **PROBLEM.** Pustaka Ceria keeps the borrow requests for its reading room in a queue with room for four requests, stored in an array. Implement `enqueue`, `dequeue`, `isEmpty` and `isFull` so that a slot freed by `dequeue` can be used again, and show the state of the array after each group of operations for this sequence: enqueue `B1001` to `B1005`; dequeue twice; enqueue `B1005` and `B1006`; dequeue everything.

In our own words. The front of the queue is where requests leave, the rear is where they arrive. Chapter 5 showed the trouble with the simplest array queue, in which both ends only ever move to the right: after a few operations the rear reaches the last cell, and the queue reports "full" although the cells at the front are free. The repair is to let the indices *wrap around*: after the last cell comes cell 0 again, as if the array were bent into a ring.

Design. We store two numbers: `front`, the index of the oldest request, and `count`, how many requests are waiting. Everything else follows from them. The first free slot is `count` places after `front`, wrapping around a ring of `CAPACITY` cells, which is what the remainder operator does:

$$\text{rear} = (\text{front} + \text{count}) \bmod \text{CAPACITY}, \qquad \text{front} \leftarrow (\text{front} + 1) \bmod \text{CAPACITY after a dequeue.} \tag{8.5}$$

With `count` stored, "empty" is `count == 0` and "full" is `count == CAPACITY`, and the two can never be confused. (A queue that stores only `front` and `rear` cannot tell a full ring from an empty one, because in both cases the two indices are equal; such queues keep one cell always empty to tell them apart. Keeping `count` costs one integer and avoids the puzzle.)

By hand. Four enqueues fill cells 0 to 3 (rear = 0,1,2,3) and `count` becomes 4; the fifth is refused. Two dequeues return `B1001` and `B1002` and move `front` to 1 and then 2, leaving `count` = 2. Now Equation (8.5) gives the next free slot as $(2 + 2) \bmod 4 = 0$: `B1005` goes into cell 0, the cell that `B1001` left, and `B1006` into $(2 + 3) \bmod 4 = 1$. The array now reads `B1005 B1006 B1003 B1004`, which looks out of order, but the queue order starts at `front` = 2 and wraps around: `B1003`, `B1004`, `B1005`, `B1006`. **Figure 8.7** draws the two moments as a ring, and **Listing 8.19** runs the whole sequence.

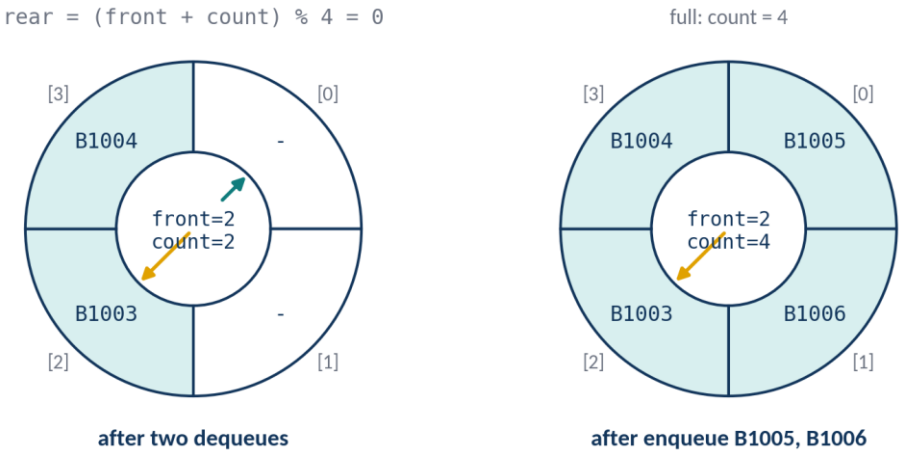


Figure 8.7 The circular queue of Problem 8.7 drawn as a ring. The amber arrow is `front`, the teal arrow is the next free slot `rear`. After two dequeues (left) the next request goes into cell 0; after two more enqueues (right) the ring is full, and the queue order starts at cell 2 and wraps around.

Listing 8.19 `p807_circular_queue.cpp`: Problem 8.7, a circular array queue with `front` and `count`.

```
// p807_circular_queue.cpp - Problem 8.7: a circular array queue of borrow requests.
#include <iostream>
#include <string>
using namespace std;

const int CAPACITY = 4;

struct CircularQueue {
    string slot[CAPACITY];
    int front = 0;      // index of the oldest request
    int count = 0;      // how many requests are waiting

    bool isEmpty() const { return count == 0; }
    bool isFull() const { return count == CAPACITY; }

    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;
    }

    void show() const {
        cout << "  slots [";
        for (int i = 0; i < CAPACITY; i++) {
            cout << (i ? " " : "") << (slot[i].empty() ? "-" : slot[i]);
        }
        cout << "]  front=" << front << " count=" << count << "\n";
    }
};
```

Listing 8.19 (Continued) *p807_circular_queue.cpp: Problem 8.7, a circular array queue with front and count.*

```
int main() {
    CircularQueue q;
    const string requests[] = {"B1001", "B1002", "B1003", "B1004", "B1005"};
    for (const string &r : requests) {
        bool ok = q.enqueue(r);
        cout << "enqueue " << r << (ok ? "" : " -> refused, the queue is full") << "\n";
    }
    q.show();
    cout << "dequeue -> " << q.dequeue() << "\n";
    cout << "dequeue -> " << q.dequeue() << "\n";
    q.show();
    cout << "enqueue B1005 -> " << (q.enqueue("B1005") ? "ok" : "refused") << "\n";
    cout << "enqueue B1006 -> " << (q.enqueue("B1006") ? "ok" : "refused") << "\n";
    q.show();
    cout << "served in order:";
    while (!q.isEmpty()) {
        cout << " " << q.dequeue();
    }
    cout << "\n";
    q.show();
    return 0;
}
```

```
enqueue B1001
enqueue B1002
enqueue B1003
enqueue B1004
enqueue B1005 -> refused, the queue is full
  slots [B1001 B1002 B1003 B1004]  front=0 count=4
dequeue -> B1001
dequeue -> B1002
  slots [- - B1003 B1004]  front=2 count=2
enqueue B1005 -> ok
enqueue B1006 -> ok
  slots [B1005 B1006 B1003 B1004]  front=2 count=4
served in order: B1003 B1004 B1005 B1006
  slots [- - - -]  front=2 count=0
```

The last `show` confirms the invariant after the queue empties: `count` is 0 and `front` has simply stopped at 2, which is fine, because with `count` equal to 0 its value does not matter. **Listing 8.20** and **Listing 8.21** print the same lines.

Listing 8.20 `P807CircularQueue.java`: Problem 8.7 in Java.

```
// P807CircularQueue.java - Problem 8.7: a circular array queue of borrow requests.
public class P807CircularQueue {

    static final int CAPACITY = 4;

    static class CircularQueue {
        final String[] slot = new String[CAPACITY];
        int front = 0;     // index of the oldest request
        int count = 0;     // how many requests are waiting

        boolean isEmpty() { return count == 0; }
        boolean isFull() { return count == CAPACITY; }

        boolean enqueue(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;
        }

        void show() {
            StringBuilder b = new StringBuilder("  slots [");
            for (int i = 0; i < CAPACITY; i++) {
                b.append(i > 0 ? " " : "").append(slot[i] == null ? "-" : slot[i]);
            }
            b.append("]  front=").append(front).append(" count=").append(count);
            System.out.println(b);
        }
    }
```

Listing 8.20 (Continued) *P807CircularQueue.java: Problem 8.7 in Java.*

```
    public static void main(String[] args) {
        CircularQueue q = new CircularQueue();
        String[] requests = {"B1001", "B1002", "B1003", "B1004", "B1005"};
        for (String r : requests) {
            boolean ok = q.enqueue(r);
            System.out.println("enqueue " + r + (ok ? "" : " -> refused, the queue is
full"));
        }
        q.show();
        System.out.println("dequeue -> " + q.dequeue());
        System.out.println("dequeue -> " + q.dequeue());
        q.show();
        System.out.println("enqueue B1005 -> " + (q.enqueue("B1005") ? "ok" : "refused"));
        System.out.println("enqueue B1006 -> " + (q.enqueue("B1006") ? "ok" : "refused"));
        q.show();
        StringBuilder served = new StringBuilder("served in order:");
        while (!q.isEmpty()) {
            served.append(" ").append(q.dequeue());
        }
        System.out.println(served);
        q.show();
    }
}
```

Listing 8.21 `p807_circular_queue.py`: Problem 8.7 in Python.

```
# p807_circular_queue.py - Problem 8.7: a circular array queue of borrow requests.
CAPACITY = 4


class CircularQueue:
    def __init__(self):
        self.slot = ["-"] * CAPACITY
        self.front = 0      # index of the oldest request
        self.count = 0      # how many requests are waiting

    def is_empty(self):
        return self.count == 0

    def is_full(self):
        return self.count == CAPACITY

    def enqueue(self, request_id):
        if self.is_full():
            return False
        rear = (self.front + self.count) % CAPACITY    # the first free slot, wrapping
around
        self.slot[rear] = request_id
        self.count += 1
        return True

    def dequeue(self):                                 # precondition: not is_empty()
        request_id = self.slot[self.front]
        self.slot[self.front] = "-"                    # only to make the picture clearer
        self.front = (self.front + 1) % CAPACITY       # the next oldest, wrapping around
        self.count -= 1
        return request_id

    def show(self):
        print(f"  slots [{' '.join(self.slot)}]  front={self.front} count={self.count}")


def main():
    q = CircularQueue()
    for r in ["B1001", "B1002", "B1003", "B1004", "B1005"]:
        ok = q.enqueue(r)
        print("enqueue", r + ("" if ok else " -> refused, the queue is full"))
    q.show()
    print("dequeue ->", q.dequeue())
    print("dequeue ->", q.dequeue())
    q.show()
    print("enqueue B1005 ->", "ok" if q.enqueue("B1005") else "refused")
    print("enqueue B1006 ->", "ok" if q.enqueue("B1006") else "refused")
```

Listing 8.21 (Continued) *p807_circular_queue.py: Problem 8.7 in Python.*

```
    q.show()
    served = []
    while not q.is_empty():
        served.append(q.dequeue())
    print("served in order:", " ".join(served))
    q.show()


if __name__ == "__main__":
    main()
```

Cost. Every operation is a few arithmetic steps: $O(1)$, whatever the capacity. That is the whole point of the ring. The alternative repair from Chapter 5, shifting all waiting items one cell to the left after each dequeue, also works but costs $O(n)$ per dequeue.

A tempting wrong solution. The linear queue of Chapter 5 keeps `front` and `rear` and declares itself full when `rear` reaches the last cell. The program `wrong/p807_wrong.cpp` fills it, dequeues twice and tries again:

```
dequeue -> B1001
dequeue -> B1002
waiting: 2 of 4
enqueue B1005 -> refused
```

Two of the four cells are free, and the queue refuses a request: the "false full" state. Its test `rear == CAPACITY - 1` asks where the rear *is*, when the question is how many items are *waiting*.

> **CHECK YOURSELF.** With `CAPACITY` = 4, start from an empty queue and do: enqueue A, B, C; dequeue; dequeue; enqueue D, E, F; dequeue. Give `front`, `count` and the four slots at the end, using "-" for a slot whose value was dequeued.
>
> *Answer.* After A, B, C: slots `A B C -`, `front` 0, `count` 3. Two dequeues: `- - C -`, `front` 2, `count` 1. D goes to $(2+1) \bmod 4 = 3$, E to $(2+2) \bmod 4 = 0$, F to $(2+3) \bmod 4 = 1$: `E F C D`, `count` 4. One dequeue returns C: slots `E F - D`, `front` 3, `count` 3. The queue order is D, E, F.

8.8.3 Problem 8.8 — From infix to postfix, and the value of an expression

> **PROBLEM.** Write two functions. `toPostfix` converts an arithmetic expression in ordinary (*infix*) notation, with single-letter or single-digit operands, the operators `+ - * /` and parentheses, into *postfix* notation, where each operator follows its two operands. `evaluate` computes the value of a postfix expression whose operands are digits. Test them on `3+2*5`, `a+b*c-d`, `8-3-2`, `(1+2)*(7-3)/2` and `9-4/2*3`.

In our own words. This is the scientific calculator of Chapter 5, completed with parentheses. In postfix notation `3+2*5` is `325*+`: there are no parentheses and no precedence rules, because the position of each operator says when it is applied. That makes postfix easy for a machine to evaluate, and the conversion is where the thinking goes. Both steps use a stack; this time we use the library stacks instead of writing our own: `std::stack` in C++, `ArrayDeque` used as a stack in Java, and a plain list in Python [9], [7], [8].

The conversion, in rules. Read the infix expression from left to right. The *degree* (precedence) of `*` and `/` is 2, of `+` and `-` is 1, and a `(` on the stack counts as 0.

1. An operand goes straight to the output.
2. `(` is pushed.
3. `)` pops operators to the output until the matching `(`, which is popped and thrown away.
4. An operator first pops to the output every operator on the stack whose degree is **greater than or equal to** its own, then is pushed itself.
5. At the end, everything left on the stack is popped to the output.

This procedure is Dijkstra's *shunting-yard algorithm*, named after the railway yard in which wagons wait on a side track until they can be sent on in the right order [10].

Why "greater than or equal to" in Rule 4? Because `-` and `/` are not associative: `8-3-2` means $(8-3)-2=3$, not $8-(3-2)=7$. When the second `-` arrives, the first `-` must leave the stack *before* it, so that it is applied first; an operator of *equal* degree must therefore be popped. With "greater than" only, the first `-` would stay on the stack, the output would be `832--`, and its value would be 7.

By hand. Table 8.6 converts `a+b*c-d`, the example of Chapter 5.

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 ≥ 1), pop `+` (1 ≥ 1), push `-`	`-`	`abc*+`
`d`	1	`-`	`abc*+d`
end	5: pop everything		`abc*+d-`

Evaluation. Read the postfix string from left to right with an empty stack of numbers. A digit is pushed. An operator pops two numbers, applies itself, and pushes the result. The first number popped is the **right** operand, because it was pushed last. For `83-2-`: push 8, push 3; `-` pops $y = 3$, then $x = 8$, and pushes $x - y = 5$; push 2; `-` pops $y = 2$, $x = 5$ and pushes 3. At the end exactly one number must be left, and it is the value. Fewer than two numbers at an operator, or more than one at the end, means the postfix string was malformed, and `evaluate` throws an exception with a message. **Listing 8.22** implements both functions and tests them.

Listing 8.22 `p808_postfix.cpp`: Problem 8.8, the shunting-yard conversion and the postfix evaluation with `std::stack`.

```cpp
// p808_postfix.cpp - Problem 8.8: infix to postfix with a stack, then evaluate the
postfix.
#include <cctype>
#include <iostream>
#include <stack>
#include <stdexcept>
#include <string>
using namespace std;

int degree(char op) {                          // the "degree" (precedence) of an operator
    if (op == '*' || op == '/') return 2;
    if (op == '+' || op == '-') return 1;
    return 0;                                  // '(' : lower than every operator
}

// Operands are single letters or digits; operators are + - * / and parentheses.
string toPostfix(const string &infix) {
    string postfix;
    stack<char> ops;
    for (char c : infix) {
        if (isalnum(static_cast<unsigned char>(c))) {
            postfix += c;                      // an operand goes straight to the output
        } else if (c == '(') {
            ops.push(c);
        } else if (c == ')') {
            while (ops.top() != '(') {         // unload down to the matching '('
                postfix += ops.top();
                ops.pop();
            }
            ops.pop();                         // drop the '(' itself
        } else if (degree(c) > 0) {
            while (!ops.empty() && degree(ops.top()) >= degree(c)) {
                postfix += ops.top();          // >= : equal degree leaves first,
                ops.pop();                     //      so 8-3-2 means (8-3)-2
            }
            ops.push(c);
        }                                      // anything else (a space) is skipped
    }
    while (!ops.empty()) {
        postfix += ops.top();
        ops.pop();
    }
    return postfix;
}
```

Listing 8.22 (Continued) *p808_postfix.cpp: Problem 8.8, the shunting-yard conversion and the postfix evaluation with std::stack.*

```
long long evaluate(const string &postfix) {
    stack<long long> values;
    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);
            }
        }
    }
    if (values.size() != 1) throw invalid_argument("too many operands");
    return values.top();
}

int main() {
    const string tests[] = {"3+2*5", "a+b*c-d", "8-3-2", "(1+2)*(7-3)/2", "9-4/2*3"};
    for (const string &t : tests) {
        string p = toPostfix(t);
        cout << t << "  ->  " << p;
        bool numeric = true;
        for (char c : p) {
            if (isalpha(static_cast<unsigned char>(c))) numeric = false;
        }
        if (numeric) cout << "  =  " << evaluate(p);
        cout << "\n";
    }
    try {
        evaluate("3+");
    } catch (const invalid_argument &e) {
        cout << "evaluate(\"3+\"): " << e.what() << "\n";
    }
    return 0;
}
```

```
3+2*5  ->  325*+  =  13
a+b*c-d  ->  abc*+d-
8-3-2  ->  83-2-  =  3
(1+2)*(7-3)/2  ->  12+73-*2/  =  6
9-4/2*3  ->  942/3*-  =  3
evaluate("3+"): missing operand
```

Listing 8.23 and **Listing 8.24** print the same lines. Two details are worth a look. In Java, `ArrayDeque` is the recommended stack class; its `push`, `pop` and `peek` work at the front of the deque [7]. In Python, integer division `//` rounds *down*, while C++ and Java truncate *toward zero*; the two agree for positive numbers but not for $-7/2$, so the Python program computes the quotient from absolute values and restores the sign, and the three programs give the same answer on every input.

Listing 8.23 `P808Postfix.java`: Problem 8.8 in Java, with `ArrayDeque` as the stack.

```java
// P808Postfix.java - Problem 8.8: infix to postfix with a stack, then evaluate the
postfix.
import java.util.ArrayDeque;

public class P808Postfix {

    static int degree(char op) {               // the "degree" (precedence) of an operator
        if (op == '*' || op == '/') return 2;
        if (op == '+' || op == '-') return 1;
        return 0;                              // '(' : lower than every operator
    }

    // Operands are single letters or digits; operators are + - * / and parentheses.
    static String toPostfix(String infix) {
        StringBuilder postfix = new StringBuilder();
        ArrayDeque<Character> ops = new ArrayDeque<>();   // used as a stack: push, peek,
pop
        for (char c : infix.toCharArray()) {
            if (Character.isLetterOrDigit(c)) {
                postfix.append(c);             // an operand goes straight to the output
            } else if (c == '(') {
                ops.push(c);
            } else if (c == ')') {
                while (ops.peek() != '(') {    // unload down to the matching '('
                    postfix.append(ops.pop());
                }
                ops.pop();                     // drop the '(' itself
            } else if (degree(c) > 0) {
                while (!ops.isEmpty() && degree(ops.peek()) >= degree(c)) {
                    postfix.append(ops.pop()); // >= : equal degree leaves first
                }
                ops.push(c);
            }
        }
        while (!ops.isEmpty()) {
            postfix.append(ops.pop());
        }
        return postfix.toString();
    }
```

Listing 8.23 (Continued) *P808Postfix.java: Problem 8.8 in Java, with ArrayDeque as the stack.*

```
    static long evaluate(String postfix) {
        ArrayDeque<Long> values = new ArrayDeque<>();
        for (char c : postfix.toCharArray()) {
            if (Character.isDigit(c)) {
                values.push((long) (c - '0'));
            } else {
                if (values.size() < 2) throw new IllegalArgumentException("missing
operand");
                long y = values.pop();         // the FIRST pop is the RIGHT operand
                long x = values.pop();         // the second pop is the left operand
                switch (c) {
                    case '+' -> values.push(x + y);
                    case '-' -> values.push(x - y);
                    case '*' -> values.push(x * y);
                    default -> {
                        if (y == 0) throw new IllegalArgumentException("division by zero");
                        values.push(x / y);
                    }
                }
            }
        }
        if (values.size() != 1) throw new IllegalArgumentException("too many operands");
        return values.pop();
    }

    public static void main(String[] args) {
        String[] tests = {"3+2*5", "a+b*c-d", "8-3-2", "(1+2)*(7-3)/2", "9-4/2*3"};
        for (String t : tests) {
            String p = toPostfix(t);
            boolean numeric = p.chars().noneMatch(Character::isLetter);
            System.out.println(t + "  ->  " + p + (numeric ? "  =  " + evaluate(p) : ""));
        }
        try {
            evaluate("3+");
        } catch (IllegalArgumentException e) {
            System.out.println("evaluate(\"3+\"): " + e.getMessage());
        }
    }
}
```

Listing 8.24 `p808_postfix.py`: Problem 8.8 in Python, with a list as the stack.

```
# p808_postfix.py - Problem 8.8: infix to postfix with a stack, then evaluate the postfix.


def degree(op):                         # the "degree" (precedence) of an operator
    if op in "*/":
        return 2
    if op in "+-":
        return 1
    return 0                            # '(' : lower than every operator


def to_postfix(infix):
    """Operands are single letters or digits; operators are + - * / and parentheses."""
    postfix = []
    ops = []                            # a Python list used as a stack: append, [-1], pop
    for c in infix:
        if c.isalnum():
            postfix.append(c)           # an operand goes straight to the output
        elif c == "(":
            ops.append(c)
        elif c == ")":
            while ops[-1] != "(":       # unload down to the matching '('
                postfix.append(ops.pop())
            ops.pop()                   # drop the '(' itself
        elif degree(c) > 0:
            while ops and degree(ops[-1]) >= degree(c):
                postfix.append(ops.pop())   # >= : equal degree leaves first
            ops.append(c)
    while ops:
        postfix.append(ops.pop())
    return "".join(postfix)


def evaluate(postfix):
    values = []
    for c in postfix:
        if c.isdigit():
            values.append(int(c))
        else:
            if len(values) < 2:
                raise ValueError("missing operand")
            y = values.pop()            # the FIRST pop is the RIGHT operand
            x = values.pop()            # the second pop is the left operand
            if c == "+":
                values.append(x + y)
            elif c == "-":
```

Listing 8.24 (Continued) *p808_postfix.py: Problem 8.8 in Python, with a list as the stack.*

```
                values.append(x - y)
            elif c == "*":
                values.append(x * y)
            else:
                if y == 0:
                    raise ValueError("division by zero")
                q = abs(x) // abs(y)    # truncate toward zero, as C++ and Java do
                values.append(q if (x >= 0) == (y >= 0) else -q)
    if len(values) != 1:
        raise ValueError("too many operands")
    return values[0]


def main():
    for t in ["3+2*5", "a+b*c-d", "8-3-2", "(1+2)*(7-3)/2", "9-4/2*3"]:
        p = to_postfix(t)
        numeric = not any(c.isalpha() for c in p)
        print(f"{t}  ->  {p}" + (f"  =  {evaluate(p)}" if numeric else ""))
    try:
        evaluate("3+")
    except ValueError as e:
        print(f'evaluate("3+"): {e}')


if __name__ == "__main__":
    main()
```

Cost. Each character is pushed and popped at most once in each function, so both run in $\Theta(L)$ time for an expression of L characters.

A tempting wrong solution. In `evaluate`, it is natural to call the first number you pop `x` and the second `y`, and compute `x - y`. The program `wrong/p808_wrong.cpp` does exactly that:

```
325*+  =  13
83-2-  =  7
12+73-*2/  =  0
942/3*-  =  -9
```

Addition and multiplication do not care about the order, so `325*+` still gives 13, and a test with that expression alone would pass. Subtraction and division do care. The stack returns items in reverse order of pushing, so the operand pushed *last*, the right one, comes out *first*.

> **CHECK YOURSELF.** Convert `(a+b)*(c-d)/e` to postfix by the rules above, then evaluate `72-3*` and `92/`.
>
> *Answer.* `ab+cd-*e/`. For `72-3*`: $7 - 2 = 5$, then $5 \cdot 3 = 15$. For `92/`: $9/2 = 4$ in integer division, truncated toward zero.

8.9 A New Worked Problem on Pointers and Functions

8.9.1 Problem 8.9 — By value, by reference, by pointer

> **PROBLEM.** A Pustaka Ceria member owes a fine of 1000 rupiah, stored in `int balance`. Write three functions that each add 500 to a fine, one receiving the fine *by value*, one *by reference* and one *by pointer*, call each on `balance`, and print `balance` after each call. Inside each function, print whether its parameter is the same box of memory as `balance`. Then write two `swap` functions, by value and by pointer, and a function `larger` that returns the address of the larger of two `int`s, and use it to add 10 to the larger one. Finally, show what Java and Python do in place of references and pointers.

In our own words. Chapter 6 compared the three ways in words; this problem makes a program *show* them. The question in each case is the same: when the function writes to its parameter, which box of memory changes?

By hand. **Figure 8.8** draws the memory during each call.

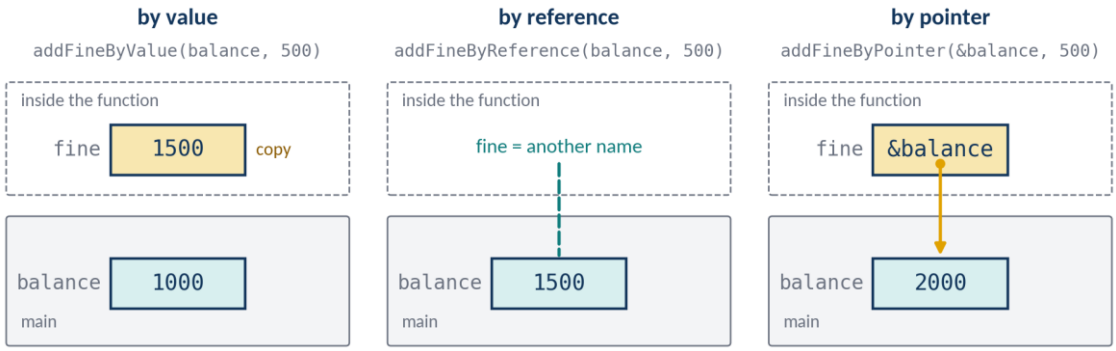


Figure 8.8 The three ways to pass `balance`, drawn at the end of each call. By value, the function works on a copy that disappears when it returns. By reference, `fine` is another name for `balance` itself. By pointer, `fine` holds the address of `balance`, and `*fine` follows it.

- **By value** (`int fine`). When the function is called, a *new* box named `fine` is created and the value 1000 is copied into it. `fine += amount` makes that box 1500, and when the function returns the box is destroyed. `balance` was never touched: it is still 1000.
- **By reference** (`int &fine`). No new box is created. `fine` is a second name for the box called `balance` in `main`; the ampersand in the parameter list means exactly that. `fine += amount` changes `balance` to 1500. The call looks innocent, `addFineByReference(balance, 500)`, which is the one danger of references: nothing at the call site warns that `balance` may change.
- **By pointer** (`int *fine`). A new box is created, but what is copied into it is the *address* of `balance`, which the caller must supply explicitly with `&balance`. The function follows the address with `*fine` and changes what it finds there: `balance` becomes 2000. The `&` at the call site is the warning that references lack. A pointer can also be `nullptr`, so a careful function checks before following it.

The same reasoning explains `swap`. A `swap` by value exchanges two copies and leaves `x` and `y` as they were; a `swap` by pointer exchanges the boxes the pointers lead to. And `larger(&x, &y)` returns an *address*, so `*larger(&x, &y) += 10` adds 10 to whichever variable is larger: the call is an address, and the `*` in front of it opens the box.

The C++ solution. To print "same box as `balance`?" the program compares addresses: `&fine == callerBox` asks whether the parameter lives at the same address as `balance`. (We print `true` or `false` rather than the addresses themselves, because addresses change from run to run and would make the output impossible to reproduce.) The whole program is **Listing 8.25**.

Listing 8.25 `p809_three_ways.cpp`: Problem 8.9, the three ways to pass an argument in C++, with a check of which memory each parameter uses.

```cpp
// p809_three_ways.cpp - Problem 8.9: pass by value, by reference and by pointer, traced.
#include <iostream>
using namespace std;

int *callerBox = nullptr;   // remembers where main's balance lives, for the checks below

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;
}

void addFineByPointer(int *fine, int amount) {    // fine holds the ADDRESS of balance
    cout << "  inside by pointer  : same box as balance? " << (fine == callerBox) << "\n";
    if (fine != nullptr) {
        *fine += amount;                           // follow the address, change what is
there
    }
}

void swapByValue(int a, int b) {
    int t = a;
    a = b;
    b = t;
}

void swapByPointer(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// Returns the address of the larger of two fines, so the caller can change that one.
int *larger(int *a, int *b) {
    return (*a >= *b) ? a : b;
}

int main() {
    cout << boolalpha;
    int balance = 1000;
    callerBox = &balance;
    cout << "start: balance = " << balance << "\n";
```

Listing 8.25 (Continued) *p809_three_ways.cpp: Problem 8.9, the three ways to pass an argument in C++, with a check of which memory each parameter uses.*

```
    addFineByValue(balance, 500);
    cout << "after by value    : balance = " << balance << "\n";
    addFineByReference(balance, 500);
    cout << "after by reference: balance = " << balance << "\n";
    addFineByPointer(&balance, 500);
    cout << "after by pointer  : balance = " << balance << "\n";

    int x = 1, y = 2;
    swapByValue(x, y);
    cout << "swapByValue(x, y)     : x = " << x << ", y = " << y << "\n";
    swapByPointer(&x, &y);
    cout << "swapByPointer(&x, &y) : x = " << x << ", y = " << y << "\n";

    *larger(&x, &y) += 10;                          // the call is an address; * opens it
    cout << "*larger(&x, &y) += 10 : x = " << x << ", y = " << y << "\n";
    return 0;
}
```

```
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
swapByValue(x, y)     : x = 1, y = 2
swapByPointer(&x, &y) : x = 2, y = 1
*larger(&x, &y) += 10 : x = 12, y = 1
```

Java and Python. Here the three languages part company, and this is the one problem whose three programs print different lines. Java has one way to pass arguments: **by value**, always [6], [7]. But the value of a variable of a class type *is a reference* to an object, so passing an object copies the reference, and the method can follow it and change the object. It cannot make the caller's variable refer to a different object. **Listing 8.26** shows all three facts: `addFine(int, ...)` cannot change `balance`; `addFine(Account, ...)` changes the object; `replace` re-points its own copy of the reference and the caller never sees it. A Java `swap(int, int)` is impossible; the usual way around it is to keep both values in one object, here an array, and swap inside it.

Listing 8.26 `P809ThreeWays.java`: Problem 8.9 in Java, where every argument is passed by value and object variables hold references.

```java
// P809ThreeWays.java - Problem 8.9: what Java does instead of pass by reference and by
pointer.
public class P809ThreeWays {

    static class Account {             // a member's fine balance, kept inside an object
        int balance;

        Account(int balance) {
            this.balance = balance;
        }
    }

    static void addFine(int fine, int amount) {        // fine is a copy of the int
        fine += amount;                                 // changes the copy only
    }

    static void addFine(Account acc, int amount) {     // acc is a copy of the REFERENCE
        acc.balance += amount;                          // follows it: the caller's object
changes
    }

    static void replace(Account acc, int amount) {
        acc = new Account(amount);                      // re-points the copy only;
    }                                                   // the new object is simply dropped

    static void swap(int a, int b) {                    // cannot work in Java
        int t = a;
        a = b;
        b = t;
    }

    static void swap(int[] box, int i, int j) {         // works: both ints live in one
object
        int t = box[i];
        box[i] = box[j];
        box[j] = t;
    }

    static Account larger(Account a, Account b) {       // returns a reference to one of
them
        return (a.balance >= b.balance) ? a : b;
    }

    public static void main(String[] args) {
        int balance = 1000;
        System.out.println("start: balance = " + balance);
        addFine(balance, 500);
        System.out.println("after addFine(int, 500)    : balance = " + balance);
```

Listing 8.26 (Continued) *P809ThreeWays.java: Problem 8.9 in Java, where every argument is passed by value and object variables hold references.*

```
        Account acc = new Account(1000);
        addFine(acc, 500);
        System.out.println("after addFine(Account, 500): acc.balance = " + acc.balance);
        replace(acc, 9999);
        System.out.println("after replace(acc, 9999)   : acc.balance = " + acc.balance);

        int x = 1, y = 2;
        swap(x, y);
        System.out.println("swap(x, y)          : x = " + x + ", y = " + y);
        int[] box = {1, 2};
        swap(box, 0, 1);
        System.out.println("swap(box, 0, 1)     : box = [" + box[0] + ", " + box[1] + "]");

        Account p = new Account(2), q = new Account(1);
        larger(p, q).balance += 10;
        System.out.println("larger(p, q) += 10  : p = " + p.balance + ", q = " +
q.balance);
    }
}
```

```
start: balance = 1000
after addFine(int, 500)    : balance = 1000
after addFine(Account, 500): acc.balance = 1500
after replace(acc, 9999)   : acc.balance = 1500
swap(x, y)          : x = 1, y = 2
swap(box, 0, 1)     : box = [2, 1]
larger(p, q) += 10  : p = 12, q = 1
```

Python behaves like Java, with one twist: *everything* is an object, including integers, and a parameter is simply a new name for the argument's object [8]. So `add_fine_int` does receive the very same `int` object as `balance`; but integers can never change, and `fine += amount` makes the local name `fine` refer to a *new* integer object, 1500, leaving `balance` alone. An object whose attributes can change, such as an `Account`, is changed for everybody who refers to it. The last line of **Listing 8.27** uses `is`, Python's "same object?" test.

Listing 8.27 `p809_three_ways.py`: Problem 8.9 in Python, where parameters are new names for the same objects.

```
# p809_three_ways.py - Problem 8.9: what Python does instead of pass by reference and by
pointer.


class Account:                          # a member's fine balance, kept inside an object
    def __init__(self, balance):
        self.balance = balance


def add_fine_int(fine, amount):     # fine is a new name for the same int object ...
    fine += amount                  # ... but ints cannot change: this re-binds the name


def add_fine_account(acc, amount):  # acc is a new name for the caller's object
    acc.balance += amount           # changes the object itself: the caller sees it


def replace(acc, amount):
    acc = Account(amount)           # re-binds the local name only; the new object is
dropped


def swap(a, b):                     # cannot work: it only re-binds local names
    a, b = b, a


def swap_in(box, i, j):             # works: both values live in one list object
    box[i], box[j] = box[j], box[i]


def larger(a, b):                   # returns one of the two objects themselves
    return a if a.balance >= b.balance else b


def main():
    balance = 1000
    print("start: balance =", balance)
    add_fine_int(balance, 500)
    print("after add_fine_int(balance, 500)    : balance =", balance)

    acc = Account(1000)
    add_fine_account(acc, 500)
    print("after add_fine_account(acc, 500)    : acc.balance =", acc.balance)
    replace(acc, 9999)
    print("after replace(acc, 9999)            : acc.balance =", acc.balance)
```

Listing 8.27 (Continued) *p809_three_ways.py: Problem 8.9 in Python, where parameters are new names for the same objects.*

```
    x, y = 1, 2
    swap(x, y)
    print(f"swap(x, y)          : x = {x}, y = {y}")
    box = [1, 2]
    swap_in(box, 0, 1)
    print(f"swap_in(box, 0, 1)  : box = {box}")

    p, q = Account(2), Account(1)
    larger(p, q).balance += 10
    print(f"larger(p, q) += 10  : p = {p.balance}, q = {q.balance}")
    print("same object?", larger(p, q) is p)


if __name__ == "__main__":
    main()
```

```
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
swap(x, y)          : x = 1, y = 2
swap_in(box, 0, 1)  : box = [2, 1]
larger(p, q) += 10  : p = 12, q = 1
same object? True
```

Table 8.7 puts the three languages side by side.

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

A tempting wrong solution. Suppose a function should create a new fine record and give the caller a pointer to it:

```
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
}
```

The local variable `fine` lives only while `makeFine` runs; its box is reused as soon as the function returns, so the caller holds a *dangling pointer*. GCC notices this particular case and warns when the program `wrong/p809_wrong.cpp` is compiled:

```
wrong/p809_wrong.cpp:7:12: warning: address of local variable 'fine' returned [-Wreturn-
local-addr]
```

and, with optimization switched on, GCC 13 even replaces the dead address by a null pointer, so the program prints `null`. Other compilers, or a slightly more complicated function that hides the local address from the compiler, give no warning, and the program reads garbage or crashes. The correct choices are to return the value itself (`int makeFine(int)`), or to create the box on the heap with `new` and make it clear who must `delete` it. Warnings like this one are why every program in this book is compiled with `-Wall -Wextra`, and why a warning is treated as an error to be fixed.

> **CHECK YOURSELF.** In C++, `int a = 5, b = 7; int *p = &a; int &r = b; *p = *p + r; r = 1; p = &b; *p += a;` What are `a` and `b` at the end?
>
> *Answer.* `*p = *p + r` sets `a` to 5 + 7 = 12. `r = 1` sets `b` to 1, because `r` is another name for `b`. `p = &b` re-points `p` at `b`. `*p += a` adds 12 to `b`. So `a` = 12 and `b` = 13.

8.10 New Worked Problems on Singly Linked Lists

8.10.1 Problem 8.10 — Reversing a list in place

> **PROBLEM.** Write a function `reverse` that reverses a singly linked list *in place*: it creates no node and deletes no node, it only turns the `next` pointers around, and it returns the new first

> node. Trace it on the list 1, 2, 3, 4, 5, printing the reversed part and the untouched part after every step, and test it on a one-node list and on the empty list.

In our own words. Problem 8.4 reversed a sentence by building a *new* list with front insertions. Here the nodes already exist, and we must reuse them: after the call, the node that held 5 is first, and its `next` leads to 4, and so on down to 1, whose `next` is `nullptr`. Every arrow must turn around.

Design. Imagine the list cut into two parts: a *done* part, already reversed, whose first node is `prev`, and a *rest* part, not yet touched, whose first node is `cur`. At the start `done` is empty (`prev = nullptr`) and `rest` is the whole list (`cur = head`). One step moves the first node of `rest` to the front of `done`, in four assignments:

1. `after = cur->next;` **save** the way to the rest of the list;
2. `cur->next = prev;` turn `cur`'s arrow around, so it joins the `done` part;
3. `prev = cur;` `cur` is now the first node of `done`;
4. `cur = after;` the rest starts at the saved node.

The invariant is: *`prev` is the first node of the reversed first part, and `cur` is the first node of the untouched remainder*. When `cur` becomes `nullptr`, the remainder is empty, so `prev` is the first node of the whole reversed list, and the function returns it. **Figure 8.9** shows one full step on the example, and **Listing 8.28** prints the state after every step.

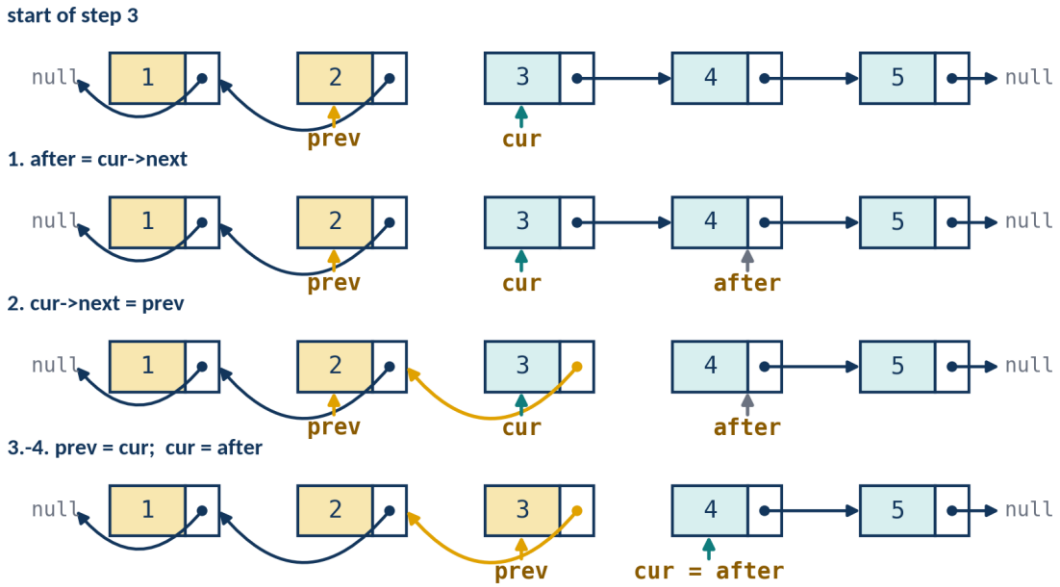


Figure 8.9 One step of the in-place reversal, the third one: save `cur->next` in `after`, turn `cur`'s arrow around to `prev`, then move `prev` and `cur` one node on. Amber nodes are the reversed part.

Listing 8.28 `p810_reverse_list.cpp`: Problem 8.10, in-place reversal with three pointers, traced step by step.

```cpp
// p810_reverse_list.cpp - Problem 8.10: reverse a singly linked list in place with three pointers.
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node *next;
};

void printChain(const char *label, const Node *p) {
    cout << label << "[";
    for (; p != nullptr; p = p->next) {
        cout << p->data << (p->next != nullptr ? " " : "");
    }
    cout << "]";
}

// Turns every arrow around. Returns the new first node (the old last one).
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
        if (trace) {
            cout << "after step " << ++step << ": ";
            printChain("done = ", prev);
            printChain("   rest = ", cur);
            cout << "\n";
        }
    }
    return prev;
}

Node *build(int from, int to) {            // the list from, from+1, ..., to
    Node *head = nullptr;
    for (int v = to; v >= from; v--) {
        head = new Node{v, head};           // front insertion, largest first
    }
    return head;
}
```

Listing 8.28 (Continued) *p810_reverse_list.cpp: Problem 8.10, in-place reversal with three pointers, traced step by step.*

```
void destroy(Node *head) {
    while (head != nullptr) {
        Node *after = head->next;
        delete head;
        head = after;
    }
}

int main() {
    Node *list = build(1, 5);
    printChain("start       : done = ", nullptr);
    printChain("   rest = ", list);
    cout << "\n";
    list = reverse(list, true);
    printChain("result: ", list);
    cout << "\n";
    destroy(list);

    Node *one = reverse(build(7, 7), false);
    printChain("one node  -> ", one);
    cout << "\n";
    destroy(one);
    Node *none = reverse(nullptr, false);
    printChain("empty list -> ", none);
    cout << "\n";
    return 0;
}
```

```
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]
one node  -> [7]
empty list -> []
```

The trace shows the invariant at work: after step k, `done` holds the first k values in reverse order and `rest` the others in their original order. The one-node and empty lists need no special code: for one node the loop runs once and returns that node; for the empty list it never runs and returns `nullptr`. **Listing 8.29** and **Listing 8.30** print the same lines.

Listing 8.29 `P810ReverseList.java`: Problem 8.10 in Java.

```java
// P810ReverseList.java - Problem 8.10: reverse a singly linked list in place with three
references.
public class P810ReverseList {

    static class Node {
        int data;
        Node next;

        Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    static String chain(Node p) {
        StringBuilder b = new StringBuilder("[");
        for (; p != null; p = p.next) {
            b.append(p.data).append(p.next != null ? " " : "");
        }
        return b.append("]").toString();
    }

    // Turns every arrow around. Returns the new first node (the old last one).
    static Node reverse(Node head, boolean trace) {
        Node prev = null;          // the part already reversed
        Node cur = head;           // the part not yet touched
        int step = 0;
        while (cur != null) {
            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
            if (trace) {
                System.out.println("after step " + (++step) + ": done = " + chain(prev)
                        + "   rest = " + chain(cur));
            }
        }
        return prev;
    }
```

Listing 8.29 (Continued) *P810ReverseList.java: Problem 8.10 in Java.*

```java
    static Node build(int from, int to) {      // the list from, from+1, ..., to
        Node head = null;
        for (int v = to; v >= from; v--) {
            head = new Node(v, head);
        }
        return head;
    }

    public static void main(String[] args) {
        Node list = build(1, 5);
        System.out.println("start       : done = []   rest = " + chain(list));
        list = reverse(list, true);
        System.out.println("result: " + chain(list));
        System.out.println("one node  -> " + chain(reverse(build(7, 7), false)));
        System.out.println("empty list -> " + chain(reverse(null, false)));
    }
}
```

Listing 8.30 `p810_reverse_list.py`: Problem 8.10 in Python.

```python
# p810_reverse_list.py - Problem 8.10: reverse a singly linked list in place with three
names.


class Node:
    __slots__ = ("data", "next")

    def __init__(self, data, next=None):
        self.data = data
        self.next = next


def chain(p):
    values = []
    while p is not None:
        values.append(str(p.data))
        p = p.next
    return "[" + " ".join(values) + "]"


def reverse(head, trace=False):
    """Turns every arrow around. Returns the new first node (the old last one)."""
    prev = None                 # the part already reversed
    cur = head                  # the part not yet touched
    step = 0
    while cur is not None:
        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
        if trace:
            step += 1
            print(f"after step {step}: done = {chain(prev)}   rest = {chain(cur)}")
    return prev


def build(first, last):
    head = None
    for v in range(last, first - 1, -1):
        head = Node(v, head)
    return head
```

Listing 8.30 (Continued) *p810_reverse_list.py: Problem 8.10 in Python.*

```python
def main():
    lst = build(1, 5)
    print(f"start       : done = []   rest = {chain(lst)}")
    lst = reverse(lst, trace=True)
    print("result:", chain(lst))
    print("one node  ->", chain(reverse(build(7, 7))))
    print("empty list ->", chain(reverse(None)))


if __name__ == "__main__":
    main()
```

Cost. One pass, four assignments per node: $\Theta(n)$ time and $O(1)$ extra memory, since only three pointer variables are used however long the list is.

A tempting wrong solution. Leave out step 1 and write the other three in the obvious order:

```
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)
```

After the first assignment, node 1's `next` is `nullptr`, and nothing else in the program points at node 2. The third line then follows the pointer just written and moves `cur` to `nullptr`. The loop ends after one step, the "reversed" list is the single node 1, and nodes 2 to 5 are unreachable. In Java or Python they would quietly be collected; in C++ they are leaked. The program `wrong/p810_wrong.cpp` prints the result and, compiled with AddressSanitizer, its leak checker reports what was lost when the program ends:

```
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).
```

(The lines showing *where* each block was allocated are left out.) Four nodes of 16 bytes each were lost: node 2 directly, because no pointer anywhere leads to it, and nodes 3 to 5 indirectly, because only lost memory points to them.

> **CHECK YOURSELF.** What does `reverse` return, and what are the `next` pointers, for the two-node list 8 → 9? How many times is each of the four assignments executed?
>
> *Answer.* It returns the node 9. Afterwards 9's `next` is 8 and 8's `next` is `nullptr`. The loop runs twice, so each of the four assignments is executed twice.

8.10.2 Problem 8.11 — Merging two sorted lists

> **PROBLEM.** Pustaka Ceria's two branches each keep a linked list of book identifiers in ascending order: branch A has 1003, 1007, 1010, 1015 and branch B has 1001, 1007, 1012. Write `merge(a, b)`, which combines the two lists into one ascending list *by relinking their nodes*, creating no new node. Count the comparisons, and test the case of one empty list, two empty lists, and two lists that do not overlap at all.

In our own words. At every moment the smallest remaining identifier is at the front of list `a` or at the front of list `b`, because both lists are sorted. So we repeatedly compare the two front nodes, detach the smaller one, and attach it to the end of the result. When one list runs out, the other one's remaining nodes are all larger than everything in the result, and they are already in order, so they can be attached in a single assignment. This is the *merge* step of merge sort (Chapter 3), and on linked lists it needs no extra array [1], [11].

Design. Attaching to "the end of the result" needs a pointer `tail` to the last node of the result. At the start the result is empty and there is no last node, which would force a special case for the first node. The standard trick is a *dummy node*: an extra node, not part of the answer, that the result starts from. `tail` starts on the dummy, every attachment is `tail->next = node; tail = node;`, and the real result is `dummy.next`. In C++ the dummy can be an ordinary local variable, `Node dummy{0, nullptr};`, living on the stack; `tail = &dummy` is a pointer to it, a use of `&` from Chapter 6. The dummy disappears when `merge` returns, and nothing points to it by then. **Figure 8.10** shows the state after two nodes have been attached.

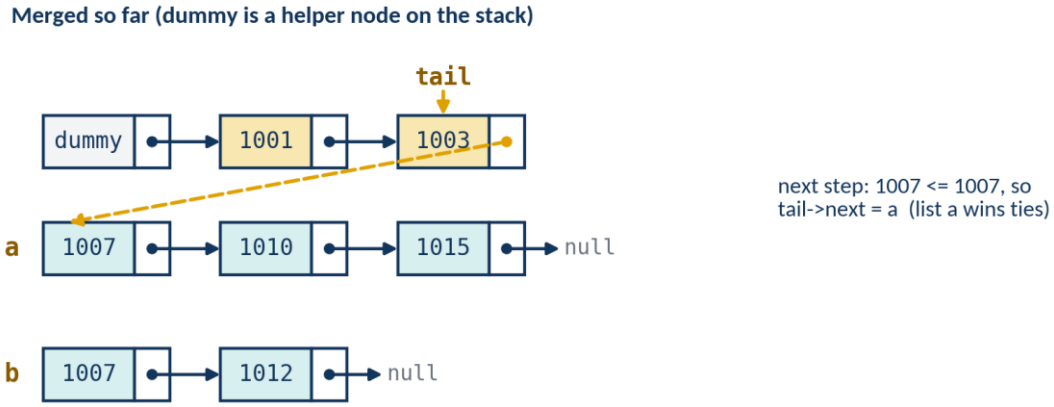


Figure 8.10 Merging the two branch lists: the result hangs from a dummy node, `tail` marks its last node, and the next comparison is between the front nodes of `a` and `b`.

By hand. Compare 1003 with 1001: take 1001 from `b`. Compare 1003 with 1007: take 1003 from `a`. Compare 1007 with 1007: equal; we take the one from `a` first, which makes the merge *stable* (equal keys keep the order of their lists). Then 1010 with 1007: take 1007 from `b`. 1010 with 1012: take 1010. 1015 with 1012: take 1012. Now `b` is empty after six comparisons; attach the rest of `a`, the single node 1015. The result is 1001, 1003, 1007, 1007, 1010, 1012, 1015.

Every comparison places exactly one node, and the last node is always placed without a comparison (it is part of the leftover), so for lists of lengths m and n

$$\text{comparisons} \le m + n - 1, \qquad \text{time} = \Theta(m+n), \qquad \text{extra memory} = O(1). \tag{8.6}$$

For the branches, $m + n - 1 = 6$, and the six comparisons above meet the bound exactly. When the lists do not overlap, as in `low + high` in the output of **Listing 8.31**, the comparisons stop as soon as the shorter list is used up: only 3.

Listing 8.31 `p811_merge_sorted.cpp`: Problem 8.11, merging two sorted linked lists with a dummy node.

```cpp
// p811_merge_sorted.cpp - Problem 8.11: merge two sorted linked lists by relinking their nodes.
#include <iostream>
using namespace std;

struct Node {
    int id;
    Node *next;
};

int comparisons = 0;

// Takes over the nodes of a and b (both sorted ascending) and returns one sorted list.
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;
}

Node *build(const int *ids, int n) {
    Node *head = nullptr;
    for (int i = n - 1; i >= 0; i--) {
        head = new Node{ids[i], head};
    }
    return head;
}
```

Listing 8.31 (Continued) *p811_merge_sorted.cpp: Problem 8.11, merging two sorted linked lists with a dummy node.*

```
void printAndFree(const char *label, Node *head) {
    cout << label;
    while (head != nullptr) {
        cout << " " << head->id;
        Node *gone = head;
        head = head->next;
        delete gone;
    }
    cout << "  (" << comparisons << " comparisons)\n";
}

int main() {
    const int branchA[] = {1003, 1007, 1010, 1015};
    const int branchB[] = {1001, 1007, 1012};

    comparisons = 0;
    printAndFree("A + B       :", merge(build(branchA, 4), build(branchB, 3)));
    comparisons = 0;
    printAndFree("A + empty   :", merge(build(branchA, 4), nullptr));
    comparisons = 0;
    printAndFree("empty+empty :", merge(nullptr, nullptr));
    const int low[] = {1, 2, 3};
    const int high[] = {7, 8, 9};
    comparisons = 0;
    printAndFree("low + high  :", merge(build(low, 3), build(high, 3)));
    return 0;
}
```

```
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)
```

The counts confirm Equation (8.6): 6 for the two branches, 0 when either list is empty, and 3 when every value of one list is smaller than every value of the other. **Listing 8.32** and **Listing 8.33** print the same lines; in them the dummy is an ordinary object.

Listing 8.32 `P811MergeSorted.java`: Problem 8.11 in Java.

```java
// P811MergeSorted.java - Problem 8.11: merge two sorted linked lists by relinking their
nodes.
public class P811MergeSorted {

    static class Node {
        int id;
        Node next;

        Node(int id, Node next) {
            this.id = id;
            this.next = next;
        }
    }

    static int comparisons = 0;

    // Takes over the nodes of a and b (both sorted ascending) and returns one sorted list.
    static Node merge(Node a, Node b) {
        Node dummy = new Node(0, null);    // a helper node, never printed
        Node tail = dummy;                 // tail = the last node of the merged list so
far
        while (a != null && b != null) {
            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 != null) ? a : b;   // attach whatever is left, all at once
        return dummy.next;
    }

    static Node build(int... ids) {
        Node head = null;
        for (int i = ids.length - 1; i >= 0; i--) {
            head = new Node(ids[i], head);
        }
        return head;
    }
```

Listing 8.32 (Continued) *P811MergeSorted.java: Problem 8.11 in Java.*

```
    static void print(String label, Node head) {
        StringBuilder b = new StringBuilder(label);
        for (Node p = head; p != null; p = p.next) {
            b.append(" ").append(p.id);
        }
        System.out.println(b + "  (" + comparisons + " comparisons)");
    }

    public static void main(String[] args) {
        comparisons = 0;
        print("A + B       :", merge(build(1003, 1007, 1010, 1015), build(1001, 1007,
1012)));
        comparisons = 0;
        print("A + empty   :", merge(build(1003, 1007, 1010, 1015), null));
        comparisons = 0;
        print("empty+empty :", merge(null, null));
        comparisons = 0;
        print("low + high  :", merge(build(1, 2, 3), build(7, 8, 9)));
    }
}
```

Listing 8.33 `p811_merge_sorted.py`: Problem 8.11 in Python.

```
# p811_merge_sorted.py - Problem 8.11: merge two sorted linked lists by relinking their
nodes.


class Node:
    __slots__ = ("id", "next")

    def __init__(self, id, next=None):
        self.id = id
        self.next = next


comparisons = 0


def merge(a, b):
    """Takes over the nodes of a and b (both sorted ascending) and returns one sorted
list."""
    global comparisons
    dummy = Node(0)             # a helper node, never printed
    tail = dummy                # tail = the last node of the merged list so far
    while a is not None and b is not None:
        comparisons += 1
        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 if a is not None else b   # attach whatever is left, all at once
    return dummy.next


def build(*ids):
    head = None
    for i in reversed(ids):
        head = Node(i, head)
    return head


def show(label, head):
    parts = [label]
    while head is not None:
        parts.append(str(head.id))
        head = head.next
    print(" ".join(parts) + f"  ({comparisons} comparisons)")
```

Listing 8.33 (Continued) *p811_merge_sorted.py: Problem 8.11 in Python.*

```
def main():
    global comparisons
    comparisons = 0
    show("A + B       :", merge(build(1003, 1007, 1010, 1015), build(1001, 1007, 1012)))
    comparisons = 0
    show("A + empty   :", merge(build(1003, 1007, 1010, 1015), None))
    comparisons = 0
    show("empty+empty :", merge(None, None))
    comparisons = 0
    show("low + high  :", merge(build(1, 2, 3), build(7, 8, 9)))


if __name__ == "__main__":
    main()
```

A tempting wrong solution. When the loop ends, it feels as if the work is done, and the last line is forgotten or replaced by `tail->next = nullptr;`. The program `wrong/p811_wrong.cpp` does that and, so that it can free them, keeps the leftover nodes aside and prints them as "lost":

```
A + B       : 1001 1003 1007 1007 1010 1012  (6 comparisons)
lost        : 1015  (6 comparisons)
```

The loop stops when *either* list is empty, so the other one almost always still has nodes, and they are all larger than the result. The edge case that reveals the bug is simply "the lists have different largest values", which is nearly every input; but a test that uses two lists ending in the same value, or whose largest values happen to be compared last, can miss it.

> **CHECK YOURSELF.** Merge `a` = 2, 4, 6, 8 with `b` = 5. List the comparisons, and say which list is empty when the loop stops.
>
> *Answer.* 2 vs 5 (take 2), 4 vs 5 (take 4), 6 vs 5 (take 5). Now `b` is empty after 3 comparisons, and the leftover 6, 8 of `a` is attached in one assignment: 2, 4, 5, 6, 8. The bound of Equation (8.6) is $4 + 1 - 1 = 4$.

8.10.3 Problem 8.12 — A queue made of linked nodes

> **PROBLEM.** Only one copy of a popular book is available, and members who want it wait in line. Implement a queue of names as a linked list with a `head` pointer (the front, where members are served) and a `tail` pointer (the back, where they join), so that `enqueue` and `dequeue` both take $O(1)$ time. Test it by enqueuing Ani, Budi and Citra, serving all three, then enqueuing Dewi and Eko and serving one more; a `dequeue` on an empty queue must be reported, not crash.

In our own words. Problem 8.7's circular queue has a fixed capacity; this queue grows as long as memory lasts. It is Chapter 7's list with head and tail, used under the rules of Chapter 5: join at the back (`backInsertion`, which with a tail pointer needs no walk), leave at the front (`frontDeletion`).

Design. The invariant has three parts: `head` points to the oldest member's node, or is `nullptr` if the queue is empty; `tail` points to the newest member's node, or is `nullptr` if the queue is empty; and `count` is the number of nodes. The two operations must keep all three true, and **Figure 8.11** shows the dangerous moment.

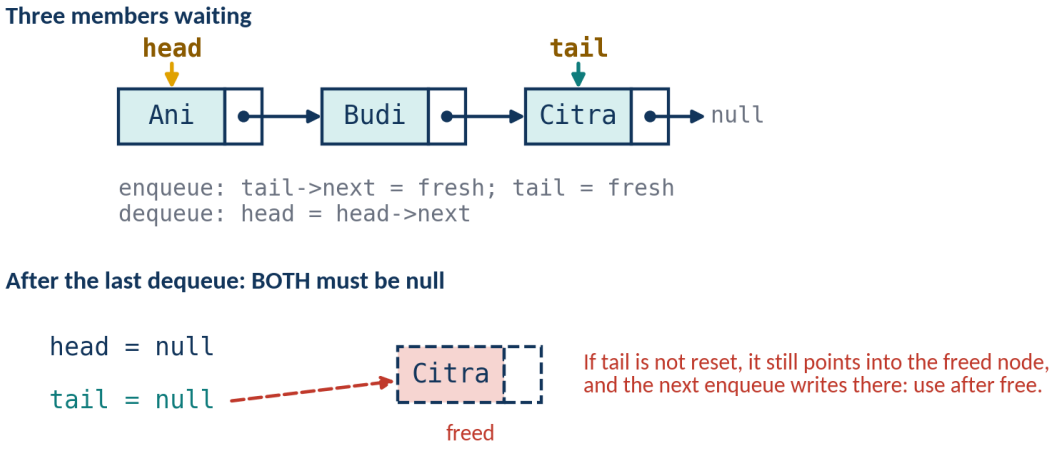


Figure 8.11 The linked queue of Problem 8.12. Enqueue works at `tail` and dequeue at `head`; when the last node leaves, `tail` must be reset to `null` too, or it is left pointing into freed memory.

- **Enqueue.** Create the node. If the queue is empty, the new node is both the first and the last, so both `head` and `tail` point to it. Otherwise link it after the current last node, `tail->next = fresh`, and make it the last, `tail = fresh`.
- **Dequeue.** If the queue is empty, throw an exception. Otherwise remember the first node, move `head` to its successor, and free the old first node. And then the line that is easy to forget: if `head` has become `nullptr`, the queue is now empty, and **`tail` must become `nullptr` too**, because the node it pointed to has just been freed.

Listing 8.34 `p812_linked_queue.cpp`: Problem 8.12, a queue built from a linked list with `head` and `tail`.

```
// p812_linked_queue.cpp - Problem 8.12: a queue built from a linked list with head and
tail.
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

struct Node {
    string name;
    Node *next;
};

class LinkedQueue {
    Node *head = nullptr;    // the front: the next member to be served
    Node *tail = nullptr;    // the back: the member who arrived last
    int count = 0;

public:
    LinkedQueue() = default;
    LinkedQueue(const LinkedQueue &) = delete;             // one owner for the nodes
    LinkedQueue &operator=(const LinkedQueue &) = delete;
    ~LinkedQueue() {
        while (!isEmpty()) dequeue();
    }

    bool isEmpty() const { return head == nullptr; }
    int size() const { return count; }

    void enqueue(const string &name) {                     // O(1): no walk to the back
        Node *fresh = new Node{name, nullptr};
        if (isEmpty()) {
            head = fresh;                                  // the only node is both
            tail = fresh;                                  // the front and the back
        } else {
            tail->next = fresh;                            // old back -> new node
            tail = fresh;                                  // the new node is the back
        }
        count++;
    }
```

Listing 8.34 (Continued) *p812_linked_queue.cpp: Problem 8.12, a queue built from a linked list with head and tail.*

```
    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;
    }

    void show() const {
        cout << "  queue:";
        for (Node *p = head; p != nullptr; p = p->next) cout << " " << p->name;
        cout << "  (size " << count << ", head " << (head ? head->name : "null")
             << ", tail " << (tail ? tail->name : "null") << ")\n";
    }
};

int main() {
    LinkedQueue waiting;              // members waiting for the only copy of a book
    waiting.enqueue("Ani");
    waiting.enqueue("Budi");
    waiting.enqueue("Citra");
    waiting.show();
    cout << "served " << waiting.dequeue() << "\n";
    cout << "served " << waiting.dequeue() << "\n";
    cout << "served " << waiting.dequeue() << "\n";
    waiting.show();
    waiting.enqueue("Dewi");          // the critical moment: enqueue after the queue
emptied
    waiting.enqueue("Eko");
    waiting.show();
    cout << "served " << waiting.dequeue() << "\n";
    try {
        LinkedQueue empty;
        empty.dequeue();
    } catch (const underflow_error &e) {
        cout << "caught: " << e.what() << "\n";
    }
    return 0;                         // the destructor frees Eko's node
}
```

```
  queue: Ani Budi Citra  (size 3, head Ani, tail Citra)
served Ani
served Budi
served Citra
  queue:  (size 0, head null, tail null)
  queue: Dewi Eko  (size 2, head Dewi, tail Eko)
served Dewi
caught: dequeue from an empty queue
```

The second state display, `(size 0, head null, tail null)`, is the invariant after the queue empties, and the enqueue of Dewi that follows works only because of it. The class also has a destructor that dequeues whatever is left, so the node of Eko is freed when `waiting` goes out of scope, and copying is forbidden for the same reason as in Problem 8.6. **Listing 8.35** and **Listing 8.36** print the same lines.

Listing 8.35 `P812LinkedQueue.java`: Problem 8.12 in Java.

```java
// P812LinkedQueue.java - Problem 8.12: a queue built from a linked list with head and
tail.
public class P812LinkedQueue {

    static class Node {
        String name;
        Node next;

        Node(String name) {
            this.name = name;
        }
    }

    static class LinkedQueue {
        private Node head = null;      // the front: the next member to be served
        private Node tail = null;      // the back: the member who arrived last
        private int count = 0;

        boolean isEmpty() { return head == null; }
        int size() { return count; }

        void enqueue(String name) {    // O(1): no walk to the back
            Node fresh = new Node(name);
            if (isEmpty()) {
                head = fresh;          // the only node is both
                tail = fresh;          // the front and the back
            } else {
                tail.next = fresh;     // old back -> new node
                tail = fresh;          // the new node is the back
            }
            count++;
        }

        String dequeue() {             // O(1)
            if (isEmpty()) throw new IllegalStateException("dequeue from an empty queue");
            String name = head.name;
            head = head.next;
            if (head == null) {
                tail = null;           // the queue is empty again
            }
            count--;
            return name;
        }
```

Listing 8.35 (Continued) *P812LinkedQueue.java: Problem 8.12 in Java.*

```
        void show() {
            StringBuilder b = new StringBuilder("  queue:");
            for (Node p = head; p != null; p = p.next) b.append(" ").append(p.name);
            b.append("  (size ").append(count)
             .append(", head ").append(head != null ? head.name : "null")
             .append(", tail ").append(tail != null ? tail.name : "null").append(")");
            System.out.println(b);
        }
    }

    public static void main(String[] args) {
        LinkedQueue waiting = new LinkedQueue();   // members waiting for the only copy of
a book
        waiting.enqueue("Ani");
        waiting.enqueue("Budi");
        waiting.enqueue("Citra");
        waiting.show();
        System.out.println("served " + waiting.dequeue());
        System.out.println("served " + waiting.dequeue());
        System.out.println("served " + waiting.dequeue());
        waiting.show();
        waiting.enqueue("Dewi");       // the critical moment: enqueue after the queue
emptied
        waiting.enqueue("Eko");
        waiting.show();
        System.out.println("served " + waiting.dequeue());
        try {
            new LinkedQueue().dequeue();
        } catch (IllegalStateException e) {
            System.out.println("caught: " + e.getMessage());
        }
    }
}
```

Listing 8.36 `p812_linked_queue.py`: Problem 8.12 in Python.

```
# p812_linked_queue.py - Problem 8.12: a queue built from a linked list with head and tail.


class Node:
    __slots__ = ("name", "next")

    def __init__(self, name):
        self.name = name
        self.next = None


class LinkedQueue:
    def __init__(self):
        self._head = None       # the front: the next member to be served
        self._tail = None       # the back: the member who arrived last
        self._count = 0

    def is_empty(self):
        return self._head is None

    def size(self):
        return self._count

    def enqueue(self, name):    # O(1): no walk to the back
        fresh = Node(name)
        if self.is_empty():
            self._head = fresh  # the only node is both
            self._tail = fresh  # the front and the back
        else:
            self._tail.next = fresh     # old back -> new node
            self._tail = fresh          # the new node is the back
        self._count += 1

    def dequeue(self):          # O(1)
        if self.is_empty():
            raise IndexError("dequeue from an empty queue")
        name = self._head.name
        self._head = self._head.next
        if self._head is None:
            self._tail = None   # the queue is empty again
        self._count -= 1
        return name
```

Listing 8.36 (Continued) *p812_linked_queue.py: Problem 8.12 in Python.*

```
    def show(self):
        names = []
        p = self._head
        while p is not None:
            names.append(p.name)
            p = p.next
        head = self._head.name if self._head else "null"
        tail = self._tail.name if self._tail else "null"
        print(f"  queue:{''.join(' ' + n for n in names)}  "
              f"(size {self._count}, head {head}, tail {tail})")


def main():
    waiting = LinkedQueue()     # members waiting for the only copy of a book
    waiting.enqueue("Ani")
    waiting.enqueue("Budi")
    waiting.enqueue("Citra")
    waiting.show()
    print("served", waiting.dequeue())
    print("served", waiting.dequeue())
    print("served", waiting.dequeue())
    waiting.show()
    waiting.enqueue("Dewi")     # the critical moment: enqueue after the queue emptied
    waiting.enqueue("Eko")
    waiting.show()
    print("served", waiting.dequeue())
    try:
        LinkedQueue().dequeue()
    except IndexError as e:
        print("caught:", e)


if __name__ == "__main__":
    main()
```

Cost. `enqueue` and `dequeue` each do a fixed number of pointer assignments: $O(1)$ time, with no walk along the list. The memory is one node per waiting member.

A tempting wrong solution. Leave out the three lines that reset `tail`, and test for an empty queue by looking at `tail` (as some textbooks do). The queue works perfectly until it becomes empty. Then `tail` still holds the address of Citra's freed node, the next `enqueue` sees a non-null `tail`, and executes `tail->next = fresh`: a write into freed memory. Without a sanitizer the program may even appear to work. The program `wrong/p812_wrong.cpp`, compiled with AddressSanitizer, is stopped at that very line; the first lines of its report (addresses shortened) are

```
served Ani
served Budi
served Citra
now enqueue Dewi
==...==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...)
freed by thread T0 here:
```

Read it like a sentence: an 8-byte write (a pointer) into memory that had already been freed, made by `enqueue` at line 34, landing 32 bytes into a 40-byte block. The block is a `Node` (a 32-byte `std::string` followed by an 8-byte `next` pointer), and offset 32 is exactly its `next` field: the stale `tail->next`. The report even names the line that freed the block. This is the most common bug in

linked queues, and it is found in seconds by the method's Step 5 ("the structure becomes empty") combined with Step 7 ("run it under a memory checker").

> **CHECK YOURSELF.** Starting from an empty queue: enqueue X; dequeue; enqueue Y; enqueue Z; dequeue. Give `head`, `tail` and `count` after each operation.
>
> *Answer.* After enqueue X: both point to X, `count` 1. After dequeue: both `nullptr`, `count` 0 (the reset line runs). After enqueue Y: both point to Y, `count` 1. After enqueue Z: `head` Y, `tail` Z, `count` 2. After dequeue (returns Y): `head` and `tail` both Z, `count` 1.

8.11 The Problems of This Chapter Compared

Table 8.8 lays the twelve problems side by side. Read down the last column. Every tempting wrong solution in this chapter fails on one particular kind of input, and in most problems that input is *not* the example given in the statement: the sample of Problem 8.3 has no negative numbers, the sample of Problem 8.5 has no neighboring copies, the test `325*+` of Problem 8.8 has no subtraction. That is the strongest argument this chapter can make for Step 5 of the method.

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	Θ(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	Θ(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

The costs repeat three patterns, and it pays to recognize them on sight. **A fixed number of pointer or index changes** is $O(1)$: every operation of a stack or a queue, whether built on an array or on nodes. **One walk along the structure** is $\Theta(n)$: reversal, merging, printing, bracket matching, the conversion to postfix. **A walk inside a walk**, where for each node we look at all the nodes after it, is $\Theta(n^2)$ by the sum $\frac{n(n-1)}{2}$ of Equation (8.3) and Equation (8.4): sorted insertion of n values, duplicate removal, and appending n times to a list without a tail. Whenever a program in the third group is too slow, the question to ask is which structure would turn the inner walk into a single step: a tail pointer (Chapter 9), a hash set (Chapter 15), or a heap (Chapter 14).

8.12 Profitina in Practice

Profitina is an AI-visibility (GEO) platform, in production at app.profitina.com, that measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds. The structures of this chapter are small, but their *shapes* are visible all over a product of that kind. We describe the shape of each problem and the class of solution, not how Profitina itself is built.

Work that waits its turn is a queue. Profitina re-scans each brand's site automatically at a cadence that depends on the subscription tier, and raises alerts when a score changes. Any system that does that has, somewhere, work waiting to be done in the order it became due: first in, first out, as in Problems 8.7 and 8.12. The real-time community feed, delivered to every open page by Server-Sent Events, has the same shape at a smaller scale: each viewer must receive the new posts in the order they were published. A streaming server must also decide what happens when one viewer's connection is slow and the posts for that viewer pile up. A queue of fixed capacity, the circular queue of Problem 8.7, is the classic way to bound the memory one slow reader can consume, and its "full" case forces the designer to decide, explicitly, what to drop or when to disconnect.

Following references is walking a list. In a threaded discussion every reply refers to the post it answers, and a quote-repost refers to the post it quotes. Getting from a deep reply back to the start of its thread means following those references one at a time, exactly like following `prev` in Problem 8.2, and it raises the same questions: what if a referenced post has been deleted (a `nullptr` in the middle of the walk), and what if a chain of references loops back on itself (the circle of Equation (8.2))? The dashboard's notification list, newest first, has the shape of the list of Problem 8.4: each new item goes in front.

Keep what you need before you change anything. Profitina's composer clears the text box the moment a post is sent, so posting feels instant, and restores the text if sending fails. That behavior is only possible if the program keeps a copy of the text *before* clearing the box: the rule of Section 8.1, in a user interface instead of a linked list.

The server is written in Python, so no Profitina engineer frees a node by hand, and the bugs of Problems 8.10 and 8.12 cannot occur in the form shown here. Their cousins can: a queue whose "last item" reference goes stale, a cache that keeps objects alive after they should be gone. The defense is the same as in this chapter, tests aimed at the edge cases, and Profitina releases nothing until its full automated suite, well over a thousand tests, passes. The runnable practice version of these ideas is Profitina Laundry, the case study of this book series; Exercise 8.13 turns the queue of Problem 8.12 into its washing-machine queue.

8.13 Common Mistakes, and How to Avoid Them

Overwriting the only pointer to something you still need. `cur->next = prev` before saving `cur->next` (Problem 8.10), `cur->next = fresh` before `fresh->next = cur->next` (Problem 8.3). Number the assignments on your drawing and check each one against the question "what becomes unreachable?"

Reading from a node after you have left it. A loop condition such as `k < cur->step` that re-reads the node as `cur` moves (Problem 8.2). Copy a node's instructions into local variables before moving.

Freeing before unlinking. `delete dup; runner->next = dup->next;` reads freed memory. Unlink first, then free (Problem 8.5).

Leaving a pointer to freed memory. `tail` still pointing at the node that the last `dequeue` freed (Problem 8.12), or a function returning the address of its own local variable (Problem 8.9). Whenever a node is freed, ask which other pointers still lead to it.

Mixing two conventions for an index. `top` as "index of the top item" in one function and "next free cell" in another (Problem 8.1); `rear` as a position when the question is a count (Problem 8.7). Write the invariant as a comment next to the field.

Moving the scanning pointer after a deletion. After a deletion the successor is new and unexamined (Problem 8.5).

Forgetting the leftover. A merge loop ends when *one* list is empty; the other one must still be attached (Problem 8.11).

Assuming `x % 2 == 1` means odd. In C++ and Java the remainder of a negative number is negative. Test `x % 2 == 0` (Problem 8.3).

Popping operands in the wrong order. The first value popped is the right operand (Problem 8.8).

Counting instead of remembering. A counter cannot tell `(` from `[`; a stack can (Problem 8.6).

Fixed-size buffers for input of unknown length. `char str[100]` overflows on the first long line; use `std::string` (Problem 8.4).

Forgetting that the end marker is not data. A `do-while` reading loop stores the sentinel too (Problems 8.2 and 8.5). Decide explicitly whether the marker belongs in the structure.

Testing only the example. In Problems 8.3, 8.5, 8.8 and 8.11 the sample input lets the wrong solution through. Test the empty structure, one node, the first and last positions, equal neighbors and negative values, and run the C++ tests once under AddressSanitizer or Valgrind.

8.14 Chapter Summary

- A problem about stacks, queues and linked lists is solved with seven steps (Section 8.2): restate it with an example, choose the structure, draw the boxes and pointers, write the invariant, list the edge cases, order the pointer changes safely, and test, including under a memory checker.
- One rule explains most of the bugs of the chapter: before changing a link, keep a way to reach everything you will still need.
- Stacks: an array stack is one array and one index with a written invariant, Equation (8.1) (Problem 8.1); brackets match with a stack because the last opened must be the first closed (Problem 8.6); the shunting-yard algorithm converts infix to postfix, and evaluation pops the right operand first (Problem 8.8).
- Queues: a circular array wraps its indices with the remainder operator, Equation (8.5), and stores a count so that full and empty differ (Problem 8.7); a linked queue with head and tail does both operations in $O(1)$ and must reset `tail` when it empties (Problem 8.12).
- Pointers and functions: by value copies, by reference renames, by pointer passes an address; Java and Python pass references to objects by value, so they return a new first node where C++ takes `Node *&` (Problem 8.9, and Problems 8.3 to 8.5).
- Singly linked lists: a two-way list needs both arrows set on every insertion, and a deterministic walk that visits more than n nodes never ends, Equation (8.2) (Problem 8.2); sorted insertion stands on the predecessor and costs Equation (8.3) (Problem 8.3); front insertion reverses (Problem 8.4); duplicate removal unlinks, then frees, then does not move, at the cost of

Equation (8.4) (Problem 8.5); in-place reversal needs three pointers (Problem 8.10); merging with a dummy node costs at most $m + n - 1$ comparisons, Equation (8.6) (Problem 8.11).

- Doubly linked lists, deques and circular lists, met here only as far as Problem 8.2 needed, are the subject of Chapter 9.

8.15 Exercises

Exercises 8.1 to 8.5 and 8.10 ask for code; their solutions are the programs of Section 8.16. The others ask for reasoning and are answered in words.

1. **(Stack.)** Add `isFull()` and `clear()` to the stack of Problem 8.1. What does `isFull()` return after 1000 pushes, and what does `clear()` do to the values stored in the array?
2. **(Length and sum.)** Write one function that walks a singly linked list once and computes both its length and the sum of its values. In C++ return the two results through reference parameters. What does it give for 4, 8, 15, 16, 23, 42?
3. **(Middle node.)** Find the middle node of a list in one walk, using two pointers: `slow` moves one node per step and `fast` two. For an even length, return the second of the two middle nodes. What is the middle of 4, 8, 15, 16, 23, 42, and of 4, 8, 15, 16, 23?
4. **(Delete the k-th node.)** Write a function that deletes node number k (the first node is number 1) and reports whether node k existed. Delete node 3, then node 1, then node 9 from 4, 8, 15, 16, 23, 42.
5. **(Palindrome.)** Decide whether a phrase reads the same backward, ignoring spaces and upper/lower case, by pushing its letters onto a stack *and* into a queue and comparing what comes out. Try “Kasur ini rusak” and “stack”.
6. **(A walk by hand.)** What does the program of Problem 8.2 print for the input `f 1 o`, `f 2 k`, `f 2 y`, `b 1 a`, `0 0 0`? Draw the list and the moves.
7. **(A circular queue by hand.)** With `CAPACITY` = 3, do: enqueue 1, 2, 3; dequeue; enqueue 4; dequeue; dequeue; enqueue 5, 6. Give the slots, `front` and `count` after every operation, and the order in which the remaining items will be served.
8. **(Postfix by hand.)** Convert `a*(b+c*d)-e` to postfix with the rules of Problem 8.8, showing the stack after each character, and evaluate `234*+5-`.
9. **(Pointer arithmetic.)** Predict the output of this fragment, then say what the array holds at the end: `int v[] = {10, 20, 30, 40}; int *p = v; *(p + 2) += 5; p++; *p = *p * 2; int *q = &v[3]; cout << *p + *q << " " << q - p;`
10. **(Duplicates in a sorted list.)** If the list is sorted, all copies of a value stand next to each other. Remove the duplicates in one walk, in $O(n)$ time and $O(1)$ extra memory. Test on 1, 1, 2, 3, 3, 3, 7, 7.
11. **(A queue from two stacks.)** Describe how to build a queue from two stacks, `in` and `out`, so that `enqueue` pushes onto `in` and `dequeue` pops from `out`. When must items move from `in` to `out`? Show that, over any sequence of operations, each item is pushed and popped at most twice, so the average cost per operation is $O(1)$.
12. **(No stored marker.)** Rewrite the reading loop of Problem 8.5 so that the end marker -1 is never stored. What do the two printed lines become for the sample input, and which function becomes unnecessary?
13. **(Profitina Laundry.)** Profitina Laundry keeps the orders waiting for its washing machine in the queue of Problem 8.12. Each order has an order number and a weight in kilograms. Say

which parts of **Listing 8.34** change, add a `peek` operation that returns the next order without removing it, and say what `peek` must do on an empty queue.

8.16 Solutions to the Exercises

Exercises 1 to 5 and 10. **Listing 8.37**, **Listing 8.38** and **Listing 8.39** contain all the requested functions and print the same lines.

Listing 8.37 `ex08_solutions.cpp`: the programming exercises of Chapter 8 in C++.

```
// ex08_solutions.cpp - solutions to the programming exercises of Chapter 8.
#include <cctype>
#include <iostream>
#include <queue>
#include <stack>
#include <string>
using namespace std;

// ---------- Exercise 8.1: isFull() and clear() for the array stack ----------
const int MAX = 1000;

class Stack {
    int top = -1;
    int a[MAX];

public:
    bool isEmpty() const { return top < 0; }
    bool isFull() const { return top == MAX - 1; }
    void clear() { top = -1; }            // the old values stay in a[], but are no longer
items
    bool push(int x) {
        if (isFull()) return false;
        a[++top] = x;
        return true;
    }
    int size() const { return top + 1; }
};

// ---------- the list used by Exercises 8.2 to 8.4 and 8.10 ----------
struct Node {
    int data;
    Node *next;
};

Node *build(const int *v, int n) {
    Node *head = nullptr;
    for (int i = n - 1; i >= 0; i--) head = new Node{v[i], head};
    return head;
}

void print(const char *label, const Node *p) {
    cout << label << "[";
    for (; p != nullptr; p = p->next) cout << p->data << (p->next ? " " : "");
    cout << "]\n";
}
```

Listing 8.37 (Continued) *ex08_solutions.cpp: the programming exercises of Chapter 8 in C++.*

```
void destroy(Node *p) {
    while (p != nullptr) {
        Node *after = p->next;
        delete p;
        p = after;
    }
}

// ---------- Exercise 8.2: length and sum, one walk ----------
void lengthAndSum(const Node *head, int &length, long long &sum) {
    length = 0;
    sum = 0;
    for (const Node *p = head; p != nullptr; p = p->next) {
        length++;
        sum += p->data;
    }
}

// ---------- Exercise 8.3: the middle node, slow and fast pointers ----------
const Node *middle(const Node *head) {
    const Node *slow = head, *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;          // one step
        fast = fast->next->next;    // two steps
    }
    return slow;                    // for an even length: the second of the two middles
}

// ---------- Exercise 8.4: delete the k-th node (k = 1 is the first) ----------
bool deleteKth(Node *&head, int k) {
    if (head == nullptr || k < 1) return false;
    if (k == 1) {
        Node *gone = head;
        head = head->next;
        delete gone;
        return true;
    }
    Node *before = head;            // walk to node k-1
    for (int i = 1; i < k - 1 && before != nullptr; i++) before = before->next;
    if (before == nullptr || before->next == nullptr) return false;   // no node k
    Node *gone = before->next;
    before->next = gone->next;      // bridge over node k ...
    delete gone;                    // ... then free it
    return true;
}
```

Listing 8.37 (Continued) *ex08_solutions.cpp: the programming exercises of Chapter 8 in C++.*

```
// ---------- Exercise 8.5: palindrome with a stack and a queue ----------
bool isPalindrome(const string &text) {
    stack<char> s;
    queue<char> q;
    for (char c : text) {
        if (isalpha(static_cast<unsigned char>(c))) {
            char lower = static_cast<char>(tolower(static_cast<unsigned char>(c)));
            s.push(lower);          // the stack gives the letters back reversed
            q.push(lower);          // the queue gives them back in order
        }
    }
    while (!s.empty()) {
        if (s.top() != q.front()) return false;
        s.pop();
        q.pop();
    }
    return true;
}

// ---------- Exercise 8.10: remove duplicates from a SORTED list in one walk ----------
void removeSortedDuplicates(Node *head) {
    Node *cur = head;
    while (cur != nullptr && cur->next != nullptr) {
        if (cur->next->data == cur->data) {
            Node *dup = cur->next;
            cur->next = dup->next;   // equal values are neighbors in a sorted list
            delete dup;
        } else {
            cur = cur->next;
        }
    }
}

int main() {
    cout << boolalpha;
    Stack s;
    for (int v = 1; v <= MAX; v++) s.push(v);
    cout << "8.1  after 1000 pushes: isFull = " << s.isFull() << ", push(0) = " <<
s.push(0) << "\n";
    s.clear();
    cout << "8.1  after clear(): isEmpty = " << s.isEmpty() << ", size = " << s.size() <<
"\n";
```

Listing 8.37 (Continued) *ex08_solutions.cpp: the programming exercises of Chapter 8 in C++.*

```
    const int v[] = {4, 8, 15, 16, 23, 42};
    Node *list = build(v, 6);
    int len;
    long long sum;
    lengthAndSum(list, len, sum);
    cout << "8.2  length = " << len << ", sum = " << sum << "\n";
    cout << "8.3  middle of 6 nodes = " << middle(list)->data;
    Node *odd = build(v, 5);
    cout << ", middle of 5 nodes = " << middle(odd)->data << "\n";
    destroy(odd);

    deleteKth(list, 3);
    print("8.4  after deleting node 3: ", list);
    deleteKth(list, 1);
    print("8.4  after deleting node 1: ", list);
    cout << "8.4  deleting node 9: " << deleteKth(list, 9) << "\n";
    destroy(list);

    cout << "8.5  \"Kasur ini rusak\" -> " << isPalindrome("Kasur ini rusak") << "\n";
    cout << "8.5  \"stack\" -> " << isPalindrome("stack") << "\n";

    const int w[] = {1, 1, 2, 3, 3, 3, 7, 7};
    Node *sorted = build(w, 8);
    removeSortedDuplicates(sorted);
    print("8.10 ", sorted);
    destroy(sorted);
    return 0;
}
```

Listing 8.38 `Ex08Solutions.java`: the programming exercises of Chapter 8 in Java.

```java
// Ex08Solutions.java - solutions to the programming exercises of Chapter 8.
import java.util.ArrayDeque;

public class Ex08Solutions {

    // ---------- Exercise 8.1: isFull() and clear() for the array stack ----------
    static final int MAX = 1000;

    static class Stack {
        private int top = -1;
        private final int[] a = new int[MAX];

        boolean isEmpty() { return top < 0; }
        boolean isFull() { return top == MAX - 1; }
        void clear() { top = -1; }         // the old values stay in a[], but are no longer
items
        boolean push(int x) {
            if (isFull()) return false;
            a[++top] = x;
            return true;
        }
        int size() { return top + 1; }
    }

    // ---------- the list used by Exercises 8.2 to 8.4 and 8.10 ----------
    static class Node {
        int data;
        Node next;

        Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    static Node build(int... v) {
        Node head = null;
        for (int i = v.length - 1; i >= 0; i--) head = new Node(v[i], head);
        return head;
    }

    static String chain(Node p) {
        StringBuilder b = new StringBuilder("[");
        for (; p != null; p = p.next) b.append(p.data).append(p.next != null ? " " : "");
        return b.append("]").toString();
    }
```

Listing 8.38 (Continued) *Ex08Solutions.java: the programming exercises of Chapter 8 in Java.*

```
    // ---------- Exercise 8.2: length and sum, one walk (returned together) ----------
    static long[] lengthAndSum(Node head) {
        long length = 0, sum = 0;
        for (Node p = head; p != null; p = p.next) {
            length++;
            sum += p.data;
        }
        return new long[] {length, sum};
    }

    // ---------- Exercise 8.3: the middle node, slow and fast references ----------
    static Node middle(Node head) {
        Node slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;              // one step
            fast = fast.next.next;         // two steps
        }
        return slow;                       // for an even length: the second of the two
middles
    }

    // ---------- Exercise 8.4: delete the k-th node (k = 1 is the first) ----------
    // Returns the (possibly new) first node; deleted[0] reports success.
    static Node deleteKth(Node head, int k, boolean[] deleted) {
        deleted[0] = false;
        if (head == null || k < 1) return head;
        if (k == 1) {
            deleted[0] = true;
            return head.next;
        }
        Node before = head;                // walk to node k-1
        for (int i = 1; i < k - 1 && before != null; i++) before = before.next;
        if (before == null || before.next == null) return head;   // no node k
        before.next = before.next.next;    // bridge over node k
        deleted[0] = true;
        return head;
    }
```

Listing 8.38 (Continued) *Ex08Solutions.java: the programming exercises of Chapter 8 in Java.*

```
    // ---------- Exercise 8.5: palindrome with a stack and a queue ----------
    static boolean isPalindrome(String text) {
        ArrayDeque<Character> stack = new ArrayDeque<>();
        ArrayDeque<Character> queue = new ArrayDeque<>();
        for (char c : text.toCharArray()) {
            if (Character.isLetter(c)) {
                char lower = Character.toLowerCase(c);
                stack.push(lower);         // push/pop at the front: a stack
                queue.addLast(lower);      // add at the back, take from the front: a queue
            }
        }
        while (!stack.isEmpty()) {
            if (!stack.pop().equals(queue.pollFirst())) return false;
        }
        return true;
    }

    // ---------- Exercise 8.10: remove duplicates from a SORTED list in one walk ---------
-
    static void removeSortedDuplicates(Node head) {
        Node cur = head;
        while (cur != null && cur.next != null) {
            if (cur.next.data == cur.data) {
                cur.next = cur.next.next;  // equal values are neighbors in a sorted list
            } else {
                cur = cur.next;
            }
        }
    }

    public static void main(String[] args) {
        Stack s = new Stack();
        for (int v = 1; v <= MAX; v++) s.push(v);
        System.out.println("8.1  after 1000 pushes: isFull = " + s.isFull() + ", push(0) =
" + s.push(0));
        s.clear();
        System.out.println("8.1  after clear(): isEmpty = " + s.isEmpty() + ", size = " +
s.size());

        Node list = build(4, 8, 15, 16, 23, 42);
        long[] ls = lengthAndSum(list);
        System.out.println("8.2  length = " + ls[0] + ", sum = " + ls[1]);
        System.out.println("8.3  middle of 6 nodes = " + middle(list).data
                + ", middle of 5 nodes = " + middle(build(4, 8, 15, 16, 23)).data);
```

Listing 8.38 (Continued) *Ex08Solutions.java: the programming exercises of Chapter 8 in Java.*

```
        boolean[] done = new boolean[1];
        list = deleteKth(list, 3, done);
        System.out.println("8.4  after deleting node 3: " + chain(list));
        list = deleteKth(list, 1, done);
        System.out.println("8.4  after deleting node 1: " + chain(list));
        list = deleteKth(list, 9, done);
        System.out.println("8.4  deleting node 9: " + done[0]);

        System.out.println("8.5  \"Kasur ini rusak\" -> " + isPalindrome("Kasur ini
rusak"));
        System.out.println("8.5  \"stack\" -> " + isPalindrome("stack"));

        Node sorted = build(1, 1, 2, 3, 3, 3, 7, 7);
        removeSortedDuplicates(sorted);
        System.out.println("8.10 " + chain(sorted));
    }
}
```

Listing 8.39 `ex08_solutions.py`: the programming exercises of Chapter 8 in Python.

```
# ex08_solutions.py - solutions to the programming exercises of Chapter 8.
from collections import deque

# ---------- Exercise 8.1: is_full() and clear() for the array stack ----------
MAX = 1000


class Stack:
    def __init__(self):
        self._top = -1
        self._a = [0] * MAX

    def is_empty(self):
        return self._top < 0

    def is_full(self):
        return self._top == MAX - 1

    def clear(self):
        self._top = -1          # the old values stay in _a, but are no longer items

    def push(self, x):
        if self.is_full():
            return False
        self._top += 1
        self._a[self._top] = x
        return True

    def size(self):
        return self._top + 1


# ---------- the list used by Exercises 8.2 to 8.4 and 8.10 ----------
class Node:
    __slots__ = ("data", "next")

    def __init__(self, data, next=None):
        self.data = data
        self.next = next
```

Listing 8.39 (Continued) *ex08_solutions.py: the programming exercises of Chapter 8 in Python.*

```
def build(*values):
    head = None
    for v in reversed(values):
        head = Node(v, head)
    return head


def chain(p):
    out = []
    while p is not None:
        out.append(str(p.data))
        p = p.next
    return "[" + " ".join(out) + "]"


# ---------- Exercise 8.2: length and sum, one walk ----------
def length_and_sum(head):
    length, total = 0, 0
    p = head
    while p is not None:
        length += 1
        total += p.data
        p = p.next
    return length, total


# ---------- Exercise 8.3: the middle node, slow and fast names ----------
def middle(head):
    slow = fast = head
    while fast is not None and fast.next is not None:
        slow = slow.next            # one step
        fast = fast.next.next       # two steps
    return slow                     # for an even length: the second of the two middles


# ---------- Exercise 8.4: delete the k-th node (k = 1 is the first) ----------
def delete_kth(head, k):
    """Returns (new first node, whether a node was deleted)."""
    if head is None or k < 1:
        return head, False
    if k == 1:
        return head.next, True
    before = head                   # walk to node k-1
    i = 1
    while i < k - 1 and before is not None:
        before = before.next
```

Listing 8.39 (Continued) *ex08_solutions.py: the programming exercises of Chapter 8 in Python.*

```
        i += 1
    if before is None or before.next is None:
        return head, False          # no node k
    before.next = before.next.next  # bridge over node k
    return head, True


# ---------- Exercise 8.5: palindrome with a stack and a queue ----------
def is_palindrome(text):
    stack, queue = [], deque()
    for c in text:
        if c.isalpha():
            stack.append(c.lower())     # the stack gives the letters back reversed
            queue.append(c.lower())     # the queue gives them back in order
    while stack:
        if stack.pop() != queue.popleft():
            return False
    return True


# ---------- Exercise 8.10: remove duplicates from a SORTED list in one walk ----------
def remove_sorted_duplicates(head):
    cur = head
    while cur is not None and cur.next is not None:
        if cur.next.data == cur.data:
            cur.next = cur.next.next    # equal values are neighbors in a sorted list
        else:
            cur = cur.next


def main():
    s = Stack()
    for v in range(1, MAX + 1):
        s.push(v)
    print(f"8.1  after 1000 pushes: isFull = {str(s.is_full()).lower()}, "
          f"push(0) = {str(s.push(0)).lower()}")
    s.clear()
    print(f"8.1  after clear(): isEmpty = {str(s.is_empty()).lower()}, size = {s.size()}")

    lst = build(4, 8, 15, 16, 23, 42)
    length, total = length_and_sum(lst)
    print(f"8.2  length = {length}, sum = {total}")
    print(f"8.3  middle of 6 nodes = {middle(lst).data}, "
          f"middle of 5 nodes = {middle(build(4, 8, 15, 16, 23)).data}")
```

Listing 8.39 (Continued) *ex08_solutions.py: the programming exercises of Chapter 8 in Python.*

```
    lst, _ = delete_kth(lst, 3)
    print("8.4  after deleting node 3:", chain(lst))
    lst, _ = delete_kth(lst, 1)
    print("8.4  after deleting node 1:", chain(lst))
    lst, done = delete_kth(lst, 9)
    print("8.4  deleting node 9:", str(done).lower())

    print('8.5  "Kasur ini rusak" ->', str(is_palindrome("Kasur ini rusak")).lower())
    print('8.5  "stack" ->', str(is_palindrome("stack")).lower())

    srt = build(1, 1, 2, 3, 3, 3, 7, 7)
    remove_sorted_duplicates(srt)
    print("8.10", chain(srt))


if __name__ == "__main__":
    main()
```

```
8.1  after 1000 pushes: isFull = true, push(0) = false
8.1  after clear(): isEmpty = true, size = 0
8.2  length = 6, sum = 108
8.3  middle of 6 nodes = 16, middle of 5 nodes = 15
8.4  after deleting node 3: [4 8 16 23 42]
8.4  after deleting node 1: [8 16 23 42]
8.4  deleting node 9: false
8.5  "Kasur ini rusak" -> true
8.5  "stack" -> false
8.10 [1 2 3 7]
```

A few words on each. **(1)** `isFull()` is `top == MAX - 1`, true after 1000 pushes, and the 1001st push fails. `clear()` only sets `top` back to -1; the old values stay in the array, but by the invariant they are no longer items, and later pushes overwrite them. It costs $O(1)$. **(2)** Length 6, sum 108; one walk, $\Theta(n)$. The sum is a `long long` so that long lists of large values cannot overflow it. **(3)** When `fast` reaches the end, `slow` has made half as many moves: the middle of the six-node list is 16 (the second of 15 and 16), of the five-node list 15. **(4)** Deleting node 3 removes 15, deleting node 1 removes 4 (and changes the first node, so C++ takes `Node *&`), and node 9 does not exist, so nothing changes and the function returns `false`. The walk stops on node $k-1$, the predecessor, as in Problem 8.5. **(5)** The stack returns the letters reversed and the queue in order; the phrase is a palindrome exactly when the two sequences agree. "Kasur ini rusak" is one; "stack" is not. **(10)** Compare each node with its successor; unlink the successor if it is equal, and move on only if it is not. The result is 1, 2, 3, 7.

Exercise 6. The nodes are node 0 `f 1 o`, node 1 `f 2 k`, node 2 `f 2 y`, node 3 `b 1 a`, node 4 the end. The walk prints `o` at node 0 and moves to node 1; prints `k` and moves forward 2 to node 3; prints `a` and moves back 1 to node 2; prints `y` and moves forward 2 to node 4, the end. The output is `okay`.

Exercise 7. Enqueue 1, 2, 3: slots `1 2 3`, `front` 0, `count` 3 (full). Dequeue returns 1: `- 2 3`, `front` 1, `count` 2. Enqueue 4 goes to $(1+2) \bmod 3 = 0$: `4 2 3`, `count` 3. Dequeue returns 2: `4 - 3`, `front` 2, `count` 2. Dequeue returns 3: `4 - -`, `front` $(2+1) \bmod 3 = 0$, `count` 1. Enqueue 5 goes to 1 and 6 to 2: `4 5 6`, `front` 0, `count` 3. They will be served as 4, 5, 6.

Exercise 8. `a` to the output; `*` pushed; `(` pushed; `b` out; `+` pushed on top of `(` (degree $1 > 0$); `c` out; `*` pushed ($2 > 1$); `d` out; `)` pops `*` and `+` and discards `(`; the output is now `abcd*+`, the stack holds `*`; `-` pops `*` ($2 \geq 1$) and is pushed; `e` out; the end pops `-`. Result: `abcd*+*e-`. For `234*+5-`: $3 \cdot 4 = 12$, $2 + 12 = 14$, $14 - 5 = 9$.

Exercise 9. `*(p + 2) += 5` makes `v[2]` 35. `p++` moves `p` to `v[1]` (one `int` further, by pointer arithmetic). `*p = *p * 2` makes `v[1]` 40. `q` points to `v[3]`, which is 40. The output is `80 2`: $40 + 40$, and `q - p` is the number of `int`s between them, $3 - 1 = 2$. The array ends as 10, 40, 35, 40.

Exercise 11. `enqueue(x)` pushes `x` onto `in`. `dequeue()` pops from `out`; only when `out` is empty does it first move *every* item from `in` to `out`, popping from one and pushing onto the other, which reverses their order so that the oldest item ends up on top of `out`. Each item is pushed onto `in` once, popped from `in` once, pushed onto `out` once and popped from `out` once: four $O(1)$ operations over its whole life. So k operations cost $O(k)$ in total, even though a single `dequeue` may cost $O(n)$ when it has to move everything. An average taken over a whole sequence like this is called an *amortized* cost [1].

Exercise 12. Test before storing: `while (cin >> data && data != -1) slist_pushBack(myList, data);`. The first line becomes `1 2 3 4 1 5 6 3 2` and the second `1 2 3 4 5 6`; `slist_popBack` is no longer needed. The sample output of the problem shows -1 on the first line, which is how we know the original loop stored it.

Exercise 13. Only the node changes: it holds `int orderNo` and `double kg` (or a small `struct Order`) instead of a `string`, and `enqueue` takes an `Order`. The pointer logic of `enqueue` and `dequeue`, and the reset of `tail`, do not change at all, because they never look at the data. `peek` returns `head->data` and moves nothing; on an empty queue it must not follow `head`, which is `nullptr`, so it throws the same `underflow_error` as `dequeue`, and a careful caller asks `isEmpty()` first.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[3] Free Software Foundation, “Program Instrumentation Options,” in *Using the GNU Compiler Collection (GCC)*, GCC 16.2, Aug. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://gcc.gnu.org/onlinedocs/

[4] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *Proc. USENIX Annual Technical Conf.*, Boston, MA, USA, 2012, pp. 309–318.

[5] Valgrind Developers, *Valgrind User Manual*, release 3.27.1, May 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://valgrind.org/docs/manual/

[6] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[7] Oracle, “Class ArrayDeque,” *Java SE 25 API Specification*, and “The Java Language Specification, Java SE 25 Edition,” §15.17.3 and §15.12.4. [Online]. Available: https://docs.oracle.com/en/java/javase/25/

[8] Python Software Foundation, “Data model,” “Binary arithmetic operations,” “collections.deque,” and “Programming FAQ: How do I write a function with output parameters?,” *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/

[9] cppreference.com, “Arithmetic operators,” “Reference declaration,” “new expression,” and “std::stack.” Accessed: Sep. 24, 2026. [Online]. Available: https://en.cppreference.com/w/cpp

[10] E. W. Dijkstra, “Algol 60 translation: An Algol 60 translator for the X1 and making a translator for Algol 60,” Mathematisch Centrum, Amsterdam, Report MR 35, 1961.

[11] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Boston, MA, USA: Pearson, 2014.

[12] A. Newell and J. C. Shaw, “Programming the Logic Theory Machine,” in *Proc. Western Joint Computer Conf.*, Los Angeles, CA, USA, 1957, pp. 230–240.

[13] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Python*. Hoboken, NJ, USA: Wiley, 2013.

[14] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

Stacks, queues, linked lists, the merge step and amortized analysis are treated in Cormen et al. [1], and linked allocation, with its history, in Knuth [2]; the linked list itself was introduced by Newell and Shaw for the IPL language they built with Simon at RAND in the mid-1950s [12]. The C++ rules for references, `new` and `delete`, and the remainder of negative numbers follow Stroustrup [6] and cppreference [9]; the Java rules for `%` and argument passing follow the language specification [7], and the Python rules the language reference [8]. The shunting-yard algorithm is Dijkstra's [10]. Weiss [11], Goodrich et al. [13] and Sedgewick and Wayne [14] give the same structures in C++, Python and Java respectively, and are good companions to this chapter. The memory checkers are described in the GCC manual [3], the Valgrind manual [5] and the original AddressSanitizer paper [4].

Chapter 9
Doubly and Circular Linked Lists

By the end of this chapter you will be able to explain why a second pointer in every node, pointing backward, removes the two limits that Chapter 7 could not engineer away; state the invariant that ties every forward link to its backward twin and check it in code; declare a doubly linked node in C++, Java and Python and say exactly how much memory it costs; implement a doubly linked list without a head node (reached through `head`, and then through `head` and `tail`) with every everyday operation (initialize, test for emptiness, insert at the front, at the back, after a node, before a node and in sorted position, delete at the front, at the back, by key and by pointer, search, and traverse in both directions); build the same list with a head node and watch half of its special cases disappear; close the list into a circle, first without and then with a sentinel node, and prove that the sentinel removes the remaining special cases; use the result to build a deque for the Pustaka Ceria returns desk and a "recently borrowed" list that always knows which title to drop next; compare singly linked lists, doubly linked lists and arrays operation by operation, with costs derived and then measured; prove with Valgrind and AddressSanitizer that the programs neither leak nor read freed memory; and choose between `std::list`, Java's `LinkedList` and Python's `collections.deque`. Every operation is drawn before and after, every cost is derived, and every program is complete, compiled, run and memory-checked.

9.1 Where We Left Off

Chapter 7 built the singly linked list and was honest about what it could not do. Two limits survived every trick in that chapter.

The first limit is *going backward*. A node of a singly linked list knows the node after it and nothing else. Once a helper pointer has walked past a node, the only way to return to it is to start again at `head` and walk forward. Printing a list in reverse order, stepping one place back from the current position, or comparing a node with its predecessor all cost a walk from the front.

The second limit is *deleting at the back*. Section 7.8 added a `tail` pointer and made back insertion $O(1)$, and Section 7.8.2 explained why back deletion stayed $\Theta(n)$ all the same: to delete the last node we must change the `next` field of the node *before* it, and a singly linked node cannot tell us which node that is. The same limit hides inside deletion by key: Section 7.6.5 needed a trailing pointer `prev` that walked one step behind `cur`, because a node cannot find its own predecessor. Even when a program already holds a pointer to the node it wants to delete, a singly linked list must search from the front for the node in front of it.

Chapter 8 then met a list whose nodes carried a second pointer. Problem 8.2 in Section 8.4 walked a two-way list forward and backward by directions, and Section 8.4.2 gave just enough of the idea to solve it: every node has a `next` and a `prev`, and every forward arrow must have a matching backward arrow. This chapter develops that idea fully.

The plan is simple to state. We add one field, `prev`, to every node. That single field lets us walk in both directions, delete the last node in $O(1)$, and delete *any* node we hold a pointer to in $O(1)$, without a search and without a trailing pointer. It also doubles the pointer work of every insertion and deletion, adds 8 bytes to every node, and creates a new way to break a list: updating

one direction and forgetting the other. We will meet all of this with the same care as in Chapter 7. Then we will close the list into a circle and add a *sentinel*, a node that holds no data and whose only job is to make every special case of the chapter disappear. Two applications for the Pustaka Ceria library follow: a deque for the returns desk and a list of recently borrowed titles.

9.2 Two Arrows per Node

9.2.1 A train you can walk in both directions

Think of a long-distance train, the kind that crosses Java overnight. Each carriage is coupled to the carriage in front of it and to the carriage behind it, and a door at each end lets the conductor walk from one carriage into either neighbor. The conductor checking tickets can walk from the locomotive to the last carriage, and then, without leaving the train, walk all the way back. If one carriage must be taken out at a station, the railway workers uncouple it at both ends and couple its two neighbors to each other. They do not need to walk to the front of the train to find out which carriage stands in front of the broken one; they can see it.

Compare this with the treasure hunt of Section 7.2.1. There, every stop knew only where the *next* stop was. A child standing at the fifth stop could not walk to the fourth stop except by starting again at the gate. The train is the treasure hunt with a second clue at every stop: “the stop before this one is behind the library door.”

Three features of the train matter for us.

- **Each carriage is coupled in two directions.** One coupling leads forward, one leads backward.
- **The two couplings of two neighbors agree.** If carriage A is coupled forward to carriage B, then B is coupled backward to A. A train in which A believes B is in front of it while B believes some other carriage is behind it is a broken train.
- **Removing or adding a carriage touches both neighbors.** Taking out a carriage in the middle changes two couplings; putting one in changes four.

A doubly linked list is this train, built in the computer’s memory.

9.2.2 The definition and the vocabulary

> **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 an external pointer `head` to the first node, often together with a second external pointer `tail` to the last node. An empty list has no nodes, and its `head` (and `tail`) are null.

The name has three parts, just as in Chapter 7. *Double* means two pointer fields per node, one in each direction. *Linked list* means the nodes are joined by pointers, not stored side by side. *Non-circular* means both chains end: following `next` from the first node eventually reaches null, and so does following `prev` from the last node. Section 9.6 removes that last word.

Figure 9.1 shows a doubly linked list holding 13, 3, 86 and 74. Each node is now drawn with three compartments: the `prev` field on the left, the data in the middle, and the `next` field on the right. A `next` arrow leaves the right compartment from its upper half and ends at the left edge of the following node; a `prev` arrow leaves the left compartment from its lower half and ends at the right edge of the preceding node. Between two neighbors there are therefore always two arrows, pointing in opposite directions.

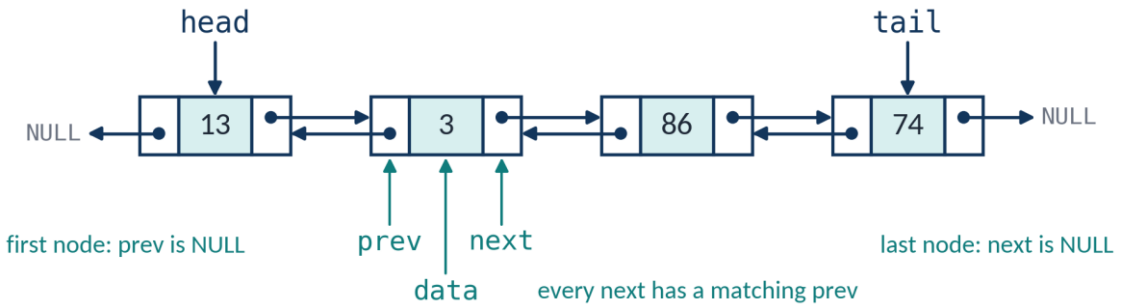


Figure 9.1 A doubly linked list holding 13, 3, 86 and 74. Each node has a `prev` field, a data field and a `next` field; `head` points at the first node and `tail` at the last, and both ends of the chain are NULL.

The second feature of the train, that two neighbors must agree, is the most important rule of this chapter. We state it as an invariant, a property that must hold before and after every operation.

DEFINITION. The *two-way invariant* of a non-circular doubly linked list: for every node p,

$$p.next \neq \text{NIL} \;\Rightarrow\; p.next.prev = p \qquad \text{and} \qquad p.prev \neq \text{NIL} \;\Rightarrow\; p.prev.next = p \tag{9.1}$$

together with $head.prev = \text{NIL}$ and $tail.next = \text{NIL}$ when the list is not empty. In words: if you step forward and then backward, you are back where you started, and the same holds if you step backward and then forward. Equation (9.1) is the rule Section 8.4.2 called the two-way list invariant, now written for both directions.

Knuth's classic treatment of linked structures presents the doubly linked list in exactly these terms, as a list in which each node carries two links, one in each direction, that must mirror each other [1]. Why insist on the invariant? Because every operation in this chapter relies on it without checking it. `backDeletion` will trust that `tail->prev` is the node before the last; `deleteNode(p)` will trust that `p->prev->next` is `p` itself. If the invariant is broken anywhere, these operations silently corrupt the list further. Section 9.4.5 writes a function that checks Equation (9.1) for a whole list, and Section 9.11 shows what happens in memory when an operation forgets half of it.

IN PLAIN LANGUAGE. A doubly linked list is a chain of boxes with two arrows between every pair of neighbors, one each way. The two arrows between neighbors must always agree. You can start at either end and walk to the other.

9.2.3 Declaring the node, and making a new one

The node of Chapter 7 gains exactly one field. In the traditional C style, with a tag name and a `typedef`:

```
typedef struct MyNode {
    int data;
    struct MyNode *prev;
    struct MyNode *next;
} TNode;
```

The structure is still *self-referential*: both `prev` and `next` have the type "pointer to `struct MyNode`". Nothing else changes. The order of the fields does not matter to the program, but by convention `prev` comes before `next`, matching the left-to-right order of the diagrams.

A new node is created with `new`, and, as the definition requires of a node that is not yet part of any list, both of its pointers start as null:

```
int newData;
// ...
TNode *newNode;
newNode = new TNode;
newNode->data = newData;
newNode->prev = nullptr;
newNode->next = nullptr;
```

After these lines the new node is a list of length one that nobody has linked to yet. The rest of the chapter is about linking such nodes into a list, and unlinking them again, without ever breaking Equation (9.1).

Listing 9.1 builds a three-node doubly linked list by hand, with no list functions at all, walks it forward from `head` and backward from `tail`, checks the invariant at one node, prints the size of a node, and releases every node.

Listing 9.1 `dll_node.cpp`: three doubly linked nodes created with `new`, linked in both directions by hand, walked both ways and released.

```
// 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;
}

int main() {
    TNode *a = makeNode(13);
    TNode *b = makeNode(3);
    TNode *c = makeNode(86);

    a->next = b;  b->prev = a;            // every forward link ...
    b->next = c;  c->prev = b;            // ... gets its backward twin
    TNode *head = a;
    TNode *tail = c;

    std::printf("forward : ");
    for (TNode *aux = head; aux != nullptr; aux = aux->next)
        std::printf("%d ", aux->data);
    std::printf("\nbackward: ");
    for (TNode *aux = tail; aux != nullptr; aux = aux->prev)
        std::printf("%d ", aux->data);
    std::printf("\n");

    std::printf("head->next->prev == head : %s\n",
                head->next->prev == head ? "yes" : "no");
    std::printf("tail->prev->data         = %d\n", tail->prev->data);
    std::printf("head->prev, tail->next   = %s, %s\n",
                head->prev == nullptr ? "NULL" : "?",
                tail->next == nullptr ? "NULL" : "?");
    std::printf("sizeof(TNode)            = %zu bytes\n", sizeof(TNode));
```

Listing 9.1 (Continued) *dll_node.cpp: three doubly linked nodes created with new, linked in both directions by hand, walked both ways and released.*

```
    int released = 0;
    while (head != nullptr) {  // save, move, delete -- as in Chapter 7
        TNode *del = head;
        head = head->next;
        delete del;
        ++released;
    }
    std::printf("nodes released           = %d\n", released);
    return 0;
}
```

```
forward : 13 3 86
backward: 86 3 13
head->next->prev == head : yes
tail->prev->data         = 3
head->prev, tail->next   = NULL, NULL
sizeof(TNode)            = 24 bytes
nodes released           = 3
```

Look first at the two linking lines. Each of them does two assignments, `a->next = b; b->prev = a;`, one forward and one backward, so that after both lines the three nodes satisfy Equation (9.1). The forward walk is the loop of Chapter 7. The backward walk is the same loop with `tail` in place of `head` and `prev` in place of `next`; nothing in it is new except the direction. The output line `head->next->prev == head : yes` is Equation (9.1) at the first node, read aloud: step forward from `head`, step back, and you are at `head` again.

9.2.4 What the second pointer costs

The output also says `sizeof(TNode) = 24 bytes`. **Figure 9.2** shows where the 24 bytes go on a 64-bit machine. The `int` takes bytes 0 to 3. A pointer must start at an address divisible by 8, so the compiler leaves bytes 4 to 7 empty (padding) and places `prev` at bytes 8 to 15 and `next` at 16 to 23. The singly linked node of Section 7.4.4 was 16 bytes; the second pointer adds exactly 8.

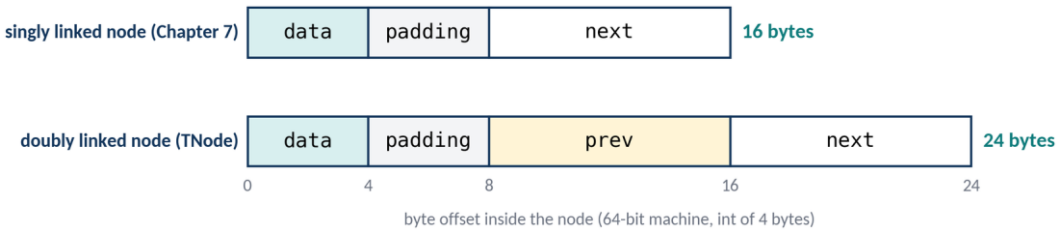


Figure 9.2 The memory layout of a node holding an `int` on a 64-bit machine: 16 bytes for a singly linked node, 24 bytes for a doubly linked one. The padding keeps each pointer on an address divisible by 8.

In general, with the notation of the memory formula of Section 7.3.3 (data of d bytes, pointers of p bytes, allocator overhead of a bytes per node), a doubly linked list of n elements occupies

$$M_{\mathrm{DLL}} = n\,(d + 2p + a) \quad \text{bytes, against} \quad M_{\mathrm{SLL}} = n\,(d + p + a). \tag{9.2}$$

The difference is exactly $n\,p$, one pointer per element, 8 MB for a million elements. For a list of `int` values this is a large tax: 24 bytes of node (and, on the GNU C library, a 32-byte allocator chunk) to carry 4 bytes of data. For a list of 200-byte book records it is 4 percent. The same rule of thumb as in Chapter 7 applies, with more force: doubly linked lists of tiny elements are expensive; doubly linked lists of large records are not.

9.2.5 How to read the pointer diagrams in this chapter

The diagrams of this chapter extend the conventions of Section 7.4.5.

- A node has three compartments: `prev`, data, `next`. A dot in a pointer compartment is the start of an arrow; `next` arrows start in the upper half and run to the right, `prev` arrows start in the lower half and run to the left. A pointer that holds null ends at the word NULL.
- Colors are those of Chapter 7: **teal** for an ordinary node, **darker teal** for the node a helper pointer is visiting, **amber** for a node being inserted, **red with a dashed border** for a node being deleted, and **grey** for a head node or a sentinel, which holds no data.
- An **amber arrow** is a pointer that the step being shown has just changed, and its small amber number is the order in which the code changes it. A **grey dashed arrow** is a link that has just been replaced. A **red arrow** is a link that is wrong.
- A node that is being inserted or deleted is often drawn *below* the row, so that the arrows that bypass it and the arrows that reach it can both be seen.
- In circular lists, the links that wrap around from the last node to the first run *under* the row: the upper of the two long lines is the `next` link, the lower one is the `prev` link.

9.3 The Doubly Linked List with a Head Pointer

We begin with the smallest complete version: a non-circular doubly linked list reached through one external pointer, `head`, which always points at the first node or is null when the list is empty. This is the list *without a head node*: every node in it holds data, and the first data node is the first node. The rule of the head from Section 7.5 still applies: never move `head` to walk the list; walk with a helper.

The functions take `head` by reference (`TNode *&head`) when they may change which node is first, and by value as `const TNode *` when they only read. For each operation we give the idea, the procedure, the before-and-after pictures and the cost. The complete program follows in Section 9.3.8.

9.3.1 `init` and `isEmpty`

Nothing changes from Chapter 7: an empty list is one whose `head` is null.

```
void init(TNode *&head) { head = nullptr; }
bool isEmpty(const TNode *head) { return head == nullptr; }
```

Both run in $O(1)$ time. A list that was never initialized is not empty; it is garbage, and the first loop over it wanders into random memory.

9.3.2 `frontInsertion`: a new first node

In a singly linked list, front insertion changed two pointers: the new node's `next` and `head`. In a doubly linked list it changes three, because the old first node must now point *back* at the new node. **Pseudocode 9.1** gives the procedure; $L.head$ is the head pointer of list L.

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
```

Figure 9.3 shows the three steps on the list 13, 86 when 3 is inserted. In panel (a) the new node exists but is linked to nothing: both of its pointers are NULL. Step 1 (panel b) makes it point forward at 13. Step 2 (panel c) makes 13 point back at it; at this moment the new node is correctly linked to 13 in both directions, but the list does not yet know about it, because `head` still points at 13. Step 3 (panel d) moves `head`.

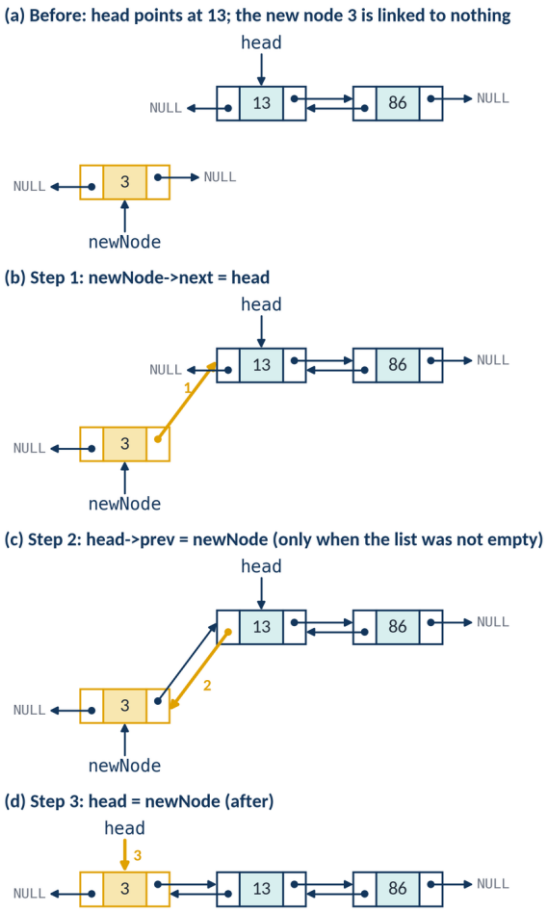


Figure 9.3 `frontInsertion(3)` on the list 13, 86: (a) before, (b) the new node points forward at the old first node, (c) the old first node points back at the new one, (d) `head` moves to the new node.

The order matters in one place. Step 3 must come last, because steps 1 and 2 both read `head` to find the old first node; if `head` moved first, both would find the new node itself. Steps 1 and 2 may be exchanged.

The empty list. When the list is empty, `L.head` is NIL, so step 1 sets `z.next` to NIL, the test in line 3 fails, and step 3 makes `z` the only node, with both pointers NIL. The test in line 3 is the first *special case* of this chapter: in a singly linked list front insertion needed none (Exercise 7.10), but here the old first node must be updated, and in an empty list there is no old first node. We will count such tests carefully, because Section 9.7 makes all of them disappear.

Cost. A constant number of assignments and one comparison: $O(1)$.

9.3.3 `backInsertion`: a new last node

With only `head`, the list does not know where its last node is, so back insertion still walks, exactly as in Section 7.5.3: `aux` starts at `head` and moves forward while `aux.next` is not NIL. When it stops, two assignments attach the new node, one in each direction.

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
```

Figure 9.4 traces the insertion of 74 into 3, 13, 86. The walk (panel a) is the walk of Chapter 7. The only new line is line 9: without it, a forward traversal would show 74 correctly, while a backward traversal starting at 74 would stop at once, because 74's `prev` would still be NIL.

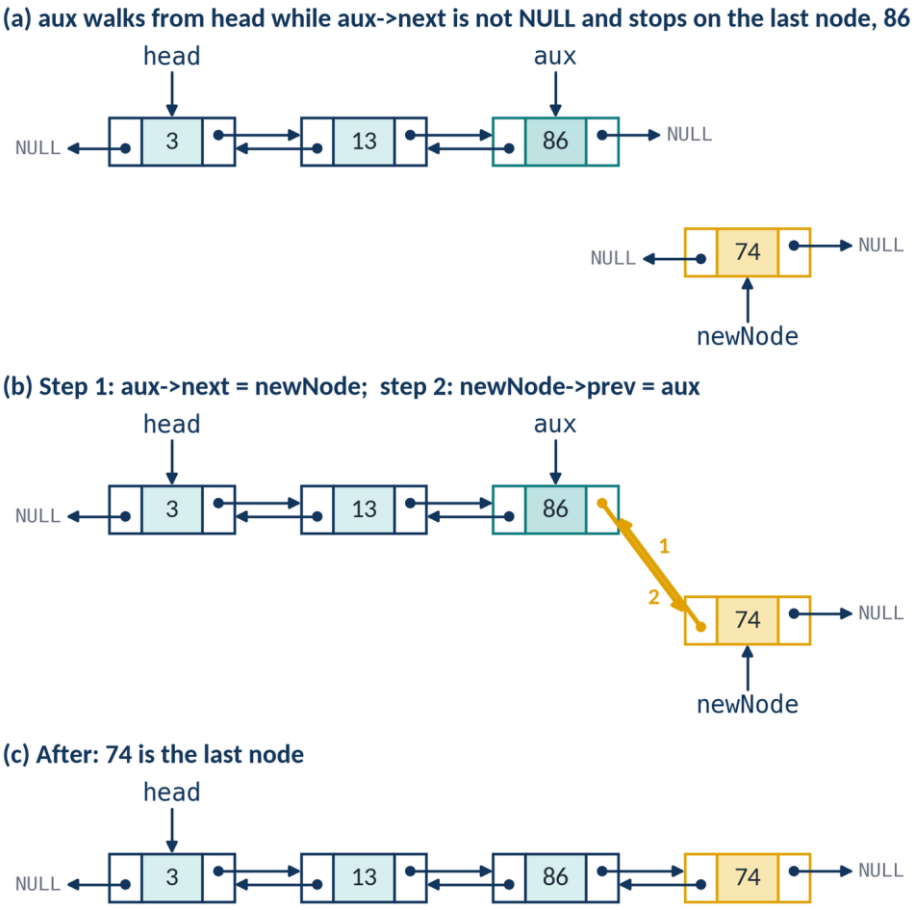


Figure 9.4 `backInsertion(74)` on the list 3, 13, 86: (a) `aux` stops on the last node, (b) two links are set, one in each direction, (c) after.

Cost. $\Theta(n)$ for the walk, the same as in a singly linked list. The second pointer does not help here, because the walk starts at the front and only a pointer to the *back* would shorten it. Section 9.4 adds that pointer.

9.3.4 Traversing in both directions

Forward traversal is the loop of Section 7.5.4, unchanged. Backward traversal is the new operation. With only a `head` pointer we must first find the last node, and then follow `prev`:

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
```

Figure 9.5 shows the two traversals of the list 3, 13, 86, 74 with the order in which the nodes are visited. Notice the two different loop conditions of Section 7.5.4 again: phase 1 tests `aux.next`, because it must *stop on* the last node, and phase 2 tests `aux`, because it must *visit* every node, including the first, and end when `aux` has stepped past the first node onto NIL.

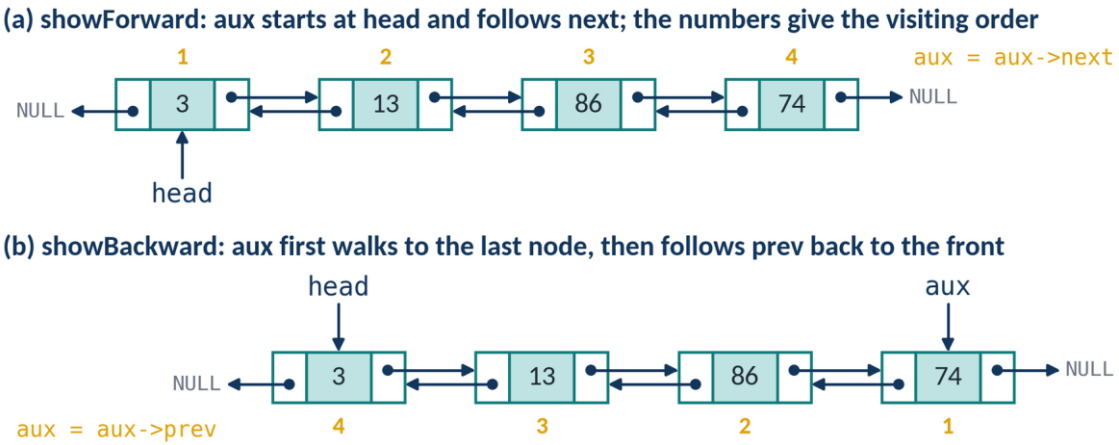


Figure 9.5 Traversing the list 3, 13, 86, 74: (a) forward from `head`, following `next`; (b) backward, following `prev` from the last node. The numbers give the order of the visits.

Cost. Forward traversal is $\Theta(n)$. Backward traversal with only `head` is $\Theta(n)$ as well, about $2n$ steps: $n-1$ to find the last node and n to come back. A singly linked list could do no better than $\Theta(n)$ even for printing *one* element in reverse position; printing the whole list backward took either $\Theta(n^2)$ time or $\Theta(n)$ extra memory for a stack. Here it takes two walks and no extra memory at all. With a `tail` pointer, phase 1 disappears.

9.3.5 `frontDeletion`: removing the first node

Front deletion follows the rule of Section 7.5.5 (save, move, delete) and adds one assignment: the new first node must forget its predecessor, because that predecessor is about to be released.

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
```

Figure 9.6 shows the steps on 3, 13, 86. Line 6 is the line that a programmer coming from singly linked lists forgets. Without it the list still *prints* correctly forward, but 13's `prev` keeps the address of the node that line 7 releases: a dangling pointer inside the list. Any later backward walk that reaches 13 continues into freed memory.

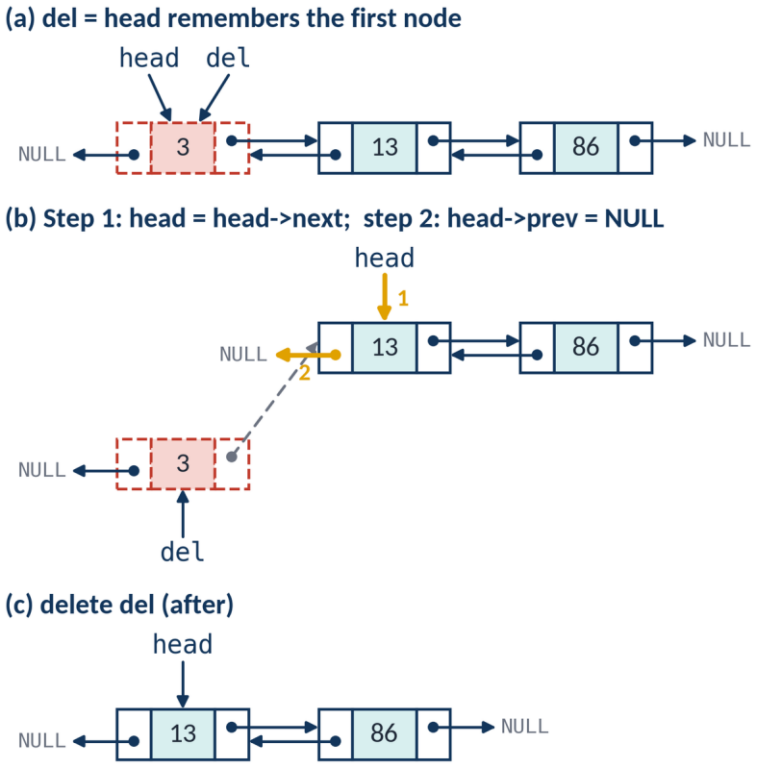


Figure 9.6 `frontDeletion()` on the list 3, 13, 86: (a) `del` remembers the first node, (b) `head` moves to 13 and 13's `prev` becomes NULL, (c) the old first node is deleted.

The test in line 5 is again a special case: when the list had one node, `L.head` becomes NIL in line 4 and there is no new first node to update. **Cost:** $O(1)$.

9.3.6 `backDeletion`: removing the last node

Here the second pointer pays off even without `tail`. In Section 7.5.6 the helper had to stop on the *second-to-last* node, with the awkward test `aux.next.next ≠ NIL`, because that node's `next` was the pointer to change. In a doubly linked list the helper can walk all the way to the last node itself, since the node before it is simply `del.prev`.

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
```

Figure 9.7 shows the deletion of 74 from 3, 13, 86, 74. Compare the loop with Chapter 7's. It is the ordinary "stop on the last node" loop of back insertion, and there is no longer any risk of evaluating `aux.next.next` on a one-node list: the one-node case is recognized by `del.prev = NIL` after the walk, not before it.

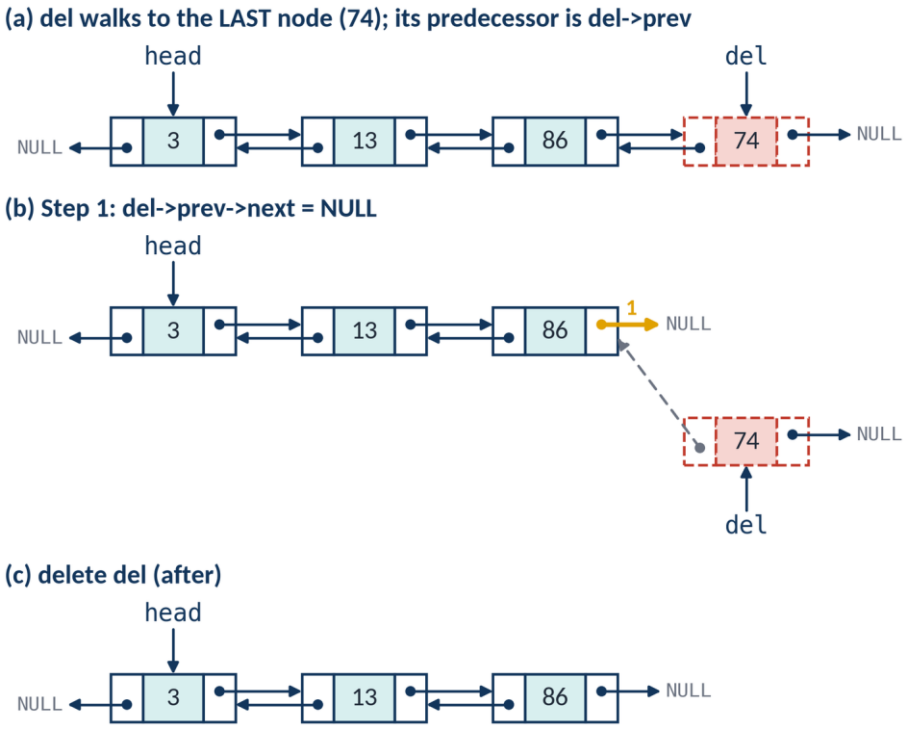


Figure 9.7 `backDeletion()` with only `head` on the list 3, 13, 86, 74: (a) `del` walks to the last node, (b) its predecessor, `del->prev`, stops pointing at it, (c) the node is deleted.

Cost. Still $\Theta(n)$, for the walk. The `prev` pointer made the code simpler, but it cannot make the walk shorter as long as the only way *in* is at the front. That is the job of `tail`, in Section 9.4.

9.3.7 `clear`

`clear` is repeated front deletion, exactly as in Section 7.5.7. It never needs to touch a `prev` field, because every node it visits is about to be released: there is nobody left to walk backward. It runs in $\Theta(n)$ time and leaves `head` null.

9.3.8 The complete program

Listing 9.2 puts the operations together. The `main` function performs the same sequence as the program `sll_head.cpp` of Section 7.5.8 (13 and 3 at the front, 86 and 74 at the back, one deletion at each end, then `clear`) and prints the list in both directions after each group of changes.

Listing 9.2 `dll_head.cpp`: the doubly linked list with a head pointer, with every operation of Section 9.3.

```
// dll_head.cpp -- a doubly linked list (non-circular) reached through head
// only.
#include <cstdio>

typedef struct MyNode {
    int data;
    struct MyNode *prev;
    struct MyNode *next;
} TNode;

void init(TNode *&head) {                // head is passed by reference
    head = nullptr;
}

bool isEmpty(const TNode *head) {
    return head == nullptr;
}

TNode *makeNode(int newData) {
    TNode *newNode = new TNode;
    newNode->data = newData;
    newNode->prev = nullptr;
    newNode->next = nullptr;
    return newNode;
}

void frontInsertion(TNode *&head, int newData) {
    TNode *newNode = makeNode(newData);
    newNode->next = head;  // 1. new node points forward at the old first
    if (!isEmpty(head)) {
        head->prev = newNode;  // 2. old first node points back at the new one
    }
    head = newNode;                      // 3. head moves to the new node
    std::printf("%d has been inserted by frontInsertion()\n", newData);
}
```

Listing 9.2 (Continued) *dll_head.cpp: the doubly linked list with a head pointer, with every operation of Section 9.3.*

```
void backInsertion(TNode *&head, int newData) {
    TNode *newNode = makeNode(newData);
    if (isEmpty(head)) {
        head = newNode;
    } else {
        TNode *aux = head;
        while (aux->next != nullptr) {    // walk to the last node
            aux = aux->next;
        }
        aux->next = newNode;  // 1. old last node points forward at it
        newNode->prev = aux;  // 2. new node points back at the old last
    }
    std::printf("%d has been inserted by backInsertion()\n", newData);
}

void showForward(const TNode *head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    for (const TNode *aux = head; aux != nullptr; aux = aux->next) {
        std::printf("%d ", aux->data);
    }
    std::printf("\n");
}

void showBackward(const TNode *head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    const TNode *aux = head;
    while (aux->next != nullptr) {  // with head only, first find the last node
        aux = aux->next;
    }
    // then follow prev back to the front
    for (; aux != nullptr; aux = aux->prev) {
        std::printf("%d ", aux->data);
    }
    std::printf("\n");
}
```

Listing 9.2 (Continued) *dll_head.cpp: the doubly linked list with a head pointer, with every operation of Section 9.3.*

```
void frontDeletion(TNode *&head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    TNode *del = head;                    // 1. remember the first node
    int d = del->data;
    head = head->next;                    // 2. head moves to the second node
    if (head != nullptr) {
        head->prev = nullptr;  // 3. the new first node has no predecessor
    }
    delete del;                           // 4. release the old first node
    std::printf("%d has been deleted by frontDeletion()\n", d);
}

void backDeletion(TNode *&head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    TNode *del = head;
    while (del->next != nullptr) {  // walk to the LAST node, not the one before
        del = del->next;
    }
    int d = del->data;
    if (del->prev == nullptr) {           // it was the only node
        head = nullptr;
    } else {
        del->prev->next = nullptr;  // its predecessor becomes the last node
    }
    delete del;
    std::printf("%d has been deleted by backDeletion()\n", d);
}

void clear(TNode *&head) {
    while (head != nullptr) {
        TNode *del = head;
        head = head->next;
        delete del;
    }
    std::printf("The linked list has been cleared\n");
}
```

Listing 9.2 (Continued) *dll_head.cpp: the doubly linked list with a head pointer, with every operation of Section 9.3.*

```
int main() {
    TNode *head;
    init(head);

    std::printf("Front insertion...\n");
    frontInsertion(head, 13);
    frontInsertion(head, 3);

    std::printf("\nBack insertion...\n");
    backInsertion(head, 86);
    backInsertion(head, 74);

    std::printf("\nForward : ");
    showForward(head);
    std::printf("Backward: ");
    showBackward(head);

    std::printf("\nFront deletion...\n");
    frontDeletion(head);
    std::printf("\nBack deletion...\n");
    backDeletion(head);

    std::printf("\nForward : ");
    showForward(head);
    std::printf("Backward: ");
    showBackward(head);

    std::printf("\n");
    clear(head);
    showForward(head);
    return 0;
}
```

```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

Forward : 3 13 86 74
Backward: 74 86 13 3

Front deletion...
3 has been deleted by frontDeletion()

Back deletion...
74 has been deleted by backDeletion()

Forward : 13 86
Backward: 86 13

The linked list has been cleared
The linked list is empty
```

The forward lines are exactly the lines that `sll_head.cpp` printed in Section 7.5.8, which is a good first check: the doubly linked list holds the same sequence as its singly linked cousin after the same operations. Each backward line is the forward line reversed, which checks the `prev` pointers: if any operation had forgotten its backward assignment, the backward line would be shorter or would contain a value that has already been deleted. That is a quick test worth copying into every program you write with two-way links: *print it both ways*.

Table 9.1 summarizes the costs we derived, next to the singly linked list of Section 7.5.

Table 9.1 Running times with only a `head` pointer, for a list of n nodes: the second pointer changes the backward traversal but not yet the back end.

Operation	Singly linked, head only	Doubly linked, head only	Why
`init`, `isEmpty`	$O(1)$	$O(1)$	one assignment or comparison
`frontInsertion`	$O(1)$	$O(1)$	three assignments (**Pseudocode 9.1**)
`backInsertion`	$\Theta(n)$	$\Theta(n)$	walk to the last node (**Pseudocode 9.2**)
forward traversal	$\Theta(n)$	$\Theta(n)$	visit every node
backward traversal	$\Theta(n^2)$, or $\Theta(n)$ with a stack	$\Theta(n)$, no extra memory	walk to the end, then follow `prev` (**Pseudocode 9.3**)
`frontDeletion`	$O(1)$	$O(1)$	save, move, clear `prev`, delete (**Pseudocode 9.4**)
`backDeletion`	$\Theta(n)$	$\Theta(n)$	walk to the last node (**Pseudocode 9.5**)
`clear`	$\Theta(n)$	$\Theta(n)$	delete every node

9.3.9 The same list in Java and Python

Listing 9.3 and **Listing 9.4** are the Java and Python versions. As in Section 7.5.9, a node is a small class with fields of its own type, and the list is a class whose field `head` plays the role of the C++ reference parameter.

Listing 9.3 `DllHead.java`: the head-reference doubly linked list in Java.

```java
// DllHead.java -- a doubly linked list (non-circular) reached through head
// only.
public class DllHead {
    static class Node {
        int data;
        Node prev, next;

        Node(int data) {
            this.data = data;
        }
    }

    static class DoublyLinkedList {
        private Node head;                    // null means the list is empty

        boolean isEmpty() {
            return head == null;
        }

        void frontInsertion(int newData) {
            Node newNode = new Node(newData);
            newNode.next = head;  // 1. new node points forward at the old first
            // 2. old first points back at it
            if (!isEmpty()) head.prev = newNode;
            head = newNode;                   // 3. head moves to the new node
            System.out.println(newData
                    + " has been inserted by frontInsertion()");
        }

        void backInsertion(int newData) {
            Node newNode = new Node(newData);
            if (isEmpty()) {
                head = newNode;
            } else {
                Node aux = head;
                // walk to the last node
                while (aux.next != null) aux = aux.next;
                aux.next = newNode;  // 1. old last node points forward at it
                newNode.prev = aux;  // 2. new node points back at the old last
            }
            System.out.println(newData
                    + " has been inserted by backInsertion()");
        }
```

Listing 9.3 (Continued) *DllHead.java: the head-reference doubly linked list in Java.*

```
    void showForward() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        StringBuilder sb = new StringBuilder();
        for (Node aux = head; aux != null; aux = aux.next)
            sb.append(aux.data).append(' ');
        System.out.println(sb);
    }

    void showBackward() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        Node aux = head;
        // first find the last node
        while (aux.next != null) aux = aux.next;
        StringBuilder sb = new StringBuilder();
        for (; aux != null; aux = aux.prev) sb.append(aux.data).append(' ');
        System.out.println(sb);
    }

    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
        // and nothing may point back at it
        if (head != null) head.prev = null;
        System.out.println(d + " has been deleted by frontDeletion()");
    }
```

Listing 9.3 (Continued) *DllHead.java: the head-reference doubly linked list in Java.*

```
        void backDeletion() {
            if (isEmpty()) {
                System.out.println("The linked list is empty");
                return;
            }
            Node del = head;
            while (del.next != null) del = del.next;  // walk to the LAST node
            int d = del.data;
            if (del.prev == null) head = null;  // it was the only node
            else del.prev.next = null;  // its predecessor becomes the last node
            System.out.println(d + " has been deleted by backDeletion()");
        }

        void clear() {
            head = null;
            System.out.println("The linked list has been cleared");
        }
    }

    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();

        System.out.println("Front insertion...");
        list.frontInsertion(13);
        list.frontInsertion(3);

        System.out.println("\nBack insertion...");
        list.backInsertion(86);
        list.backInsertion(74);

        System.out.print("\nForward : ");
        list.showForward();
        System.out.print("Backward: ");
        list.showBackward();

        System.out.println("\nFront deletion...");
        list.frontDeletion();
        System.out.println("\nBack deletion...");
        list.backDeletion();

        System.out.print("\nForward : ");
        list.showForward();
        System.out.print("Backward: ");
        list.showBackward();
```

Listing 9.3 (Continued) *DllHead.java: the head-reference doubly linked list in Java.*

```
        System.out.println();
        list.clear();
        list.showForward();
    }
}
```

Listing 9.4 `dll_head.py`: the head-reference doubly linked list in Python. `__slots__` fixes the three fields of a node.

```python
# dll_head.py -- a doubly linked list (non-circular) reached through head only.
class Node:
    __slots__ = ("data", "prev", "next")

    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None


class DoublyLinkedList:
    def __init__(self):
        self.head = None                        # None means the list is empty

    def is_empty(self):
        return self.head is None

    def front_insertion(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head  # 1. new node points forward at the old first
        if not self.is_empty():
            self.head.prev = new_node           # 2. old first points back at it
        self.head = new_node                    # 3. head moves to the new node
        print(f"{new_data} has been inserted by frontInsertion()")

    def back_insertion(self, new_data):
        new_node = Node(new_data)
        if self.is_empty():
            self.head = new_node
        else:
            aux = self.head
            while aux.next is not None:         # walk to the last node
                aux = aux.next
            aux.next = new_node  # 1. old last node points forward at it
            new_node.prev = aux  # 2. new node points back at the old last
        print(f"{new_data} has been inserted by backInsertion()")
```

Listing 9.4 (Continued) *dll_head.py: the head-reference doubly linked list in Python. __slots__ fixes the three fields of a node.*

```
    def show_forward(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        parts, aux = [], self.head
        while aux is not None:
            parts.append(f"{aux.data} ")
            aux = aux.next
        print("".join(parts))

    def show_backward(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        aux = self.head
        while aux.next is not None:            # first find the last node
            aux = aux.next
        parts = []
        while aux is not None:  # then follow prev back to the front
            parts.append(f"{aux.data} ")
            aux = aux.prev
        print("".join(parts))

    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()")

    def back_deletion(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        dele = self.head
        while dele.next is not None:           # walk to the LAST node
            dele = dele.next
        if dele.prev is None:                  # it was the only node
            self.head = None
        else:
            dele.prev.next = None  # its predecessor becomes the last node
        print(f"{dele.data} has been deleted by backDeletion()")
```

Listing 9.4 (Continued) *dll_head.py: the head-reference doubly linked list in Python. __slots__ fixes the three fields of a node.*

```
    def clear(self):
        self.head = None
        print("The linked list has been cleared")


def main():
    lst = DoublyLinkedList()

    print("Front insertion...")
    lst.front_insertion(13)
    lst.front_insertion(3)

    print("\nBack insertion...")
    lst.back_insertion(86)
    lst.back_insertion(74)

    print("\nForward : ", end="")
    lst.show_forward()
    print("Backward: ", end="")
    lst.show_backward()

    print("\nFront deletion...")
    lst.front_deletion()
    print("\nBack deletion...")
    lst.back_deletion()

    print("\nForward : ", end="")
    lst.show_forward()
    print("Backward: ", end="")
    lst.show_backward()

    print()
    lst.clear()
    lst.show_forward()


if __name__ == "__main__":
    main()
```

Both print exactly the output of **Listing 9.2**. Look at `frontDeletion` in the two garbage-collected versions. There is no `delete`, yet the line `head.prev = null` (Python: `self.head.prev = None`) is still there. It is no longer needed to avoid reading freed memory, since a reference can never point at freed memory, but it is still needed for the invariant: if the new first node kept a reference to the old one, a backward walk would visit a node that is no longer in the list, and, less visibly, the old node could never be collected while the list lives, because the list still refers to it. A stale backward link in Java or Python is a logic error and a memory leak at once.

9.4 Head and Tail: Every End in O(1)

9.4.1 Adding `tail`, and back deletion at last in O(1)

Section 7.8 added a `tail` pointer to the singly linked list and got half of what it wanted: back insertion became $O(1)$, back deletion did not. In a doubly linked list, `tail` gives both halves. **Pseudocode 9.6** is back deletion with a tail pointer.

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 that Chapter 7 could not write. **Figure 9.8** shows it on 3, 13, 86, 74: `tail` moves one step back along a `prev` pointer, 86 becomes the last node, and 74 is released, in a constant number of steps whatever the length of the list.

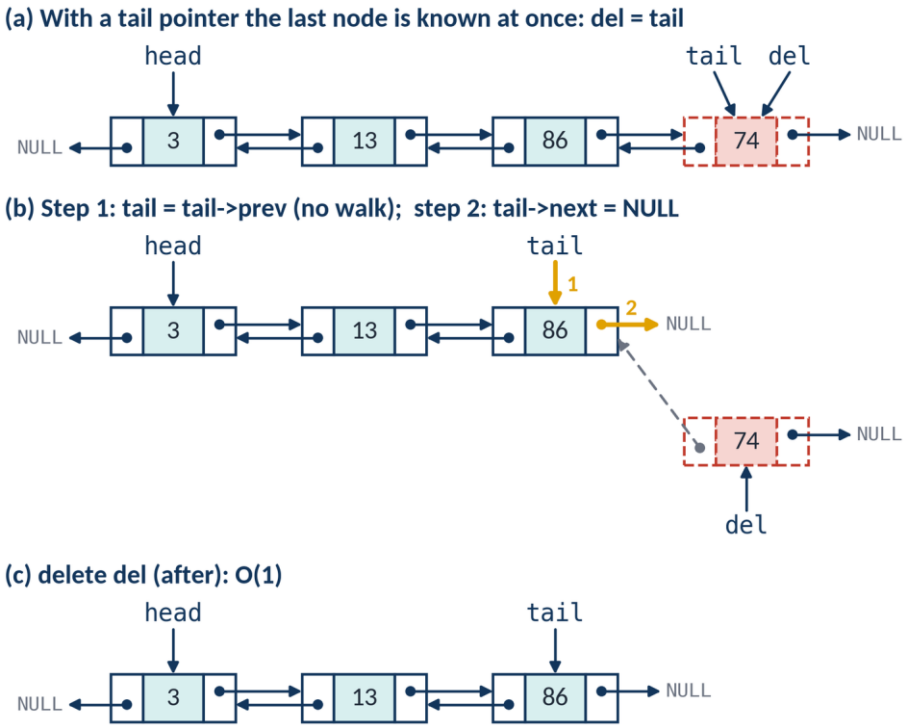


Figure 9.8 Back deletion with a tail pointer on 3, 13, 86, 74: (a) `del` = `tail`, (b) `tail` moves back along `prev` and the new last node's `next` becomes NULL, (c) after: $O(1)$, with no walk.

Back deletion is the mirror image of front deletion, line by line, with `head` and `tail` exchanged and `next` and `prev` exchanged. That symmetry is the whole design of the doubly linked list with head and tail: every operation at one end has a twin at the other end, obtained by swapping the two words. Back insertion likewise becomes the mirror image of front insertion: `z.prev ← L.tail`; if the list is not empty, `L.tail.next ← z`, otherwise `L.head ← z`; finally `L.tail ← z`.

Adding `tail` adds one clause to the invariant: `head` and `tail` are both null or both non-null, and when they are non-null, `head.prev` and `tail.next` are null. Every operation that can make the list empty (the last deletion) or non-empty (the first insertion) must now update *both* external pointers. That is where the special cases come from, and Section 9.4.2 counts them.

9.4.2 `insertAfter` and `insertBefore`: inserting next to a node you hold

Suppose the program holds a pointer `p` to a node of the list, perhaps returned by a search, and wants to insert a new value right after it. Section 7.6.3 did this for a singly linked list with two

assignments. A doubly linked list needs four, because there are four arrows between `p`, the new node `n` and `p`'s successor s: $p \to n$ and $n \to s$ forward, $s \to n$ and $n \to p$ backward.

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
```

Figure 9.9 traces `insertAfter(p, 35)` with `p` at 30 in the list 20, 30, 50. Steps 1 and 2 (panel b) touch only the new node: they make it point at its two future neighbors, and nothing in the list has changed yet. Steps 3 and 4 (panel c) then redirect the two neighbors to it. The grey dashed arrows are the direct links between 30 and 50 that the two steps replace.

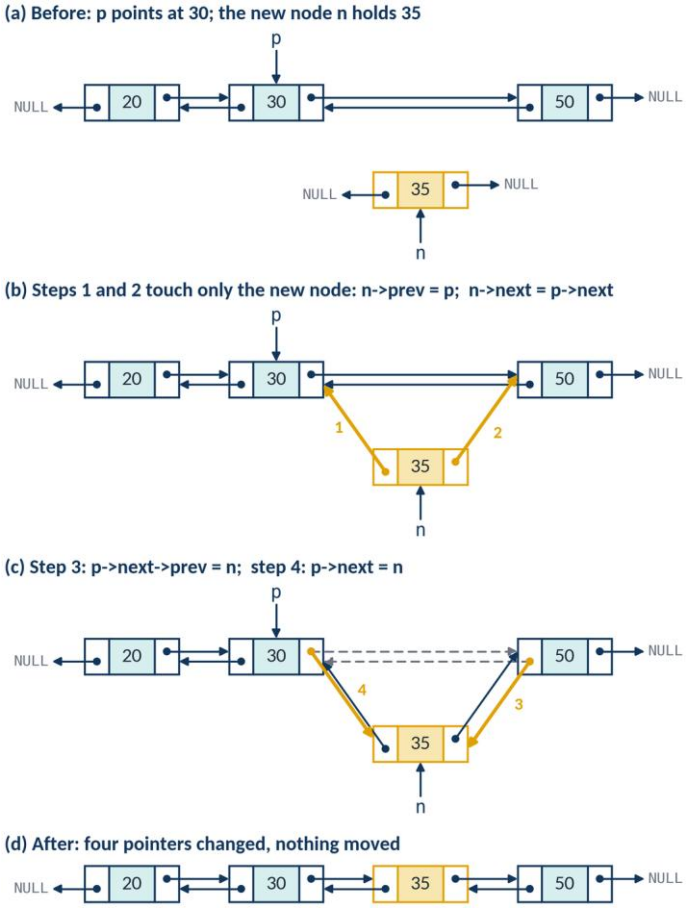


Figure 9.9 `insertAfter(p, 35)` with `p` at 30: (a) before, (b) the new node points at both neighbors, (c) both neighbors point at the new node, (d) after.

Why this order? Line 3 and line 7 both read `p.next` to find the successor. Line 8 overwrites `p.next`. If line 8 ran first, `p.next` would already be `n`, line 3 would make `n` point forward at itself, line 7 would make `n` point back at itself, and the node 50, together with everything after it, would be unreachable in the forward direction. Section 9.11 draws exactly that accident. A safe rule that

works for every insertion in this chapter is: *first set the new node's own two pointers, then update the neighbors, and update the link that you used to find a neighbor last.*

`insertBefore(p, x)` is the mirror image: `n.next ← p`, `n.prev ← p.prev`, then either `L.head ← n` (when p was the first node) or `p.prev.next ← n`, and finally `p.prev ← n`. Insertion *before* a node was impossible in $O(1)$ in a singly linked list, which had to walk from the front to find the node in front of `p` (Exercise 7.9 needed a pointer to the link for it). In a doubly linked list the predecessor is one step away.

Cost. Four assignments and one test: $O(1)$, whatever n is. Note what $O(1)$ means here: the insertion is cheap *once the program holds* `p`. Finding `p` by value is a search, which costs $\Theta(n)$ in the worst case. The same caution from Section 7.12.4 applies: a doubly linked list shines when the program already holds the positions it changes.

Counting special cases. `insertAfter` has one special case (line 4: the new node goes at the back), and `insertBefore` has one (the new node goes at the front). Front and back insertion each have one (the list was empty). Keep these numbers in mind for Section 9.7.

9.4.3 `insertSorted`: keeping the list in order

To insert x into an ascending list, walk until the first node whose data is not smaller than x, and insert before it; if there is no such node, x belongs at the back.

Pseudocode 9.8 `INSERT-SORTED`: find the first node that is not smaller than `x`, and insert before it.

```
INSERT-SORTED(L, x):
 1.  aux ← L.head
 2.  while aux ≠ NIL and aux.data < x
 3.      aux ← aux.next
 4.  if aux = NIL
 5.      BACK-INSERTION(L, x)    // x is larger than every value (or L is empty)
 6.  else
 7.      INSERT-BEFORE(L, aux, x)
```

Compare this with Section 7.6.4. There the helper had to stop on the last node *smaller* than x, one node early, because the singly linked list can only insert after a node, and the test had to look one node ahead. Here the helper stops on the node where it naturally stops, the first larger one, and the insertion happens before it. **Figure 9.10** shows `insertSorted(40)` on 10, 20, 30, 50, and the list after 5 (a new first node) and 60 (a new last node) have been inserted as well.

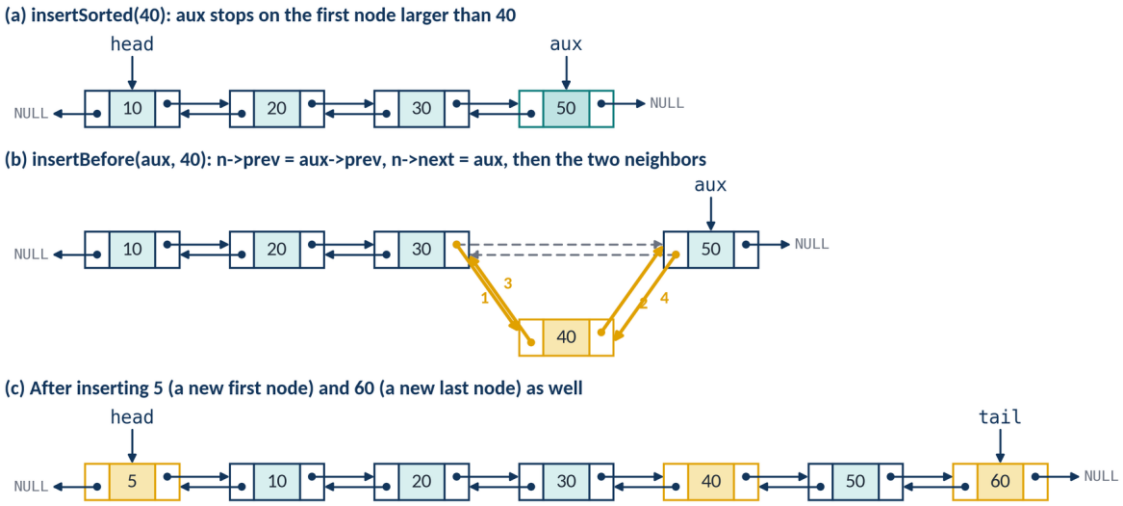


Figure 9.10 `insertSorted`: (a) for 40, `aux` stops on 50, the first larger value, (b) the new node is linked in before `aux` in four steps, (c) the list after 5 and 60 are also inserted, at the front and at the back.

Cost. The walk makes at most n steps and the insertion is $O(1)$, so `insertSorted` is $\Theta(n)$ in the worst case, the same as for a singly linked list. The doubly linked version is not faster, but it is simpler.

9.4.4 `search`, `deleteNode` and `deleteKey`

`search(L, key)` is the sequential search of Section 7.6.1, unchanged: walk from `head` and return the first node holding `key`, or NIL. With `tail` available it can equally well search from the back, which finds the *last* occurrence first.

Deletion is where the doubly linked list most clearly beats the singly linked one. Given a pointer `p` to any node, the node before it is `p.prev` and the node after it is `p.next`; two assignments make them skip over `p`.

Pseudocode 9.9 `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
```

Figure 9.11 shows `deleteKey(35)`: a search finds `p` (panel a), the two neighbors are linked to each other (panel b), and the node is released (panel c). Notice that `p`'s own pointers are *read* but never changed; the node is simply abandoned by its neighbors before it is freed. `deleteKey(L, key)` is nothing more than `p ← SEARCH(L, key)` followed by `DELETE-NODE(L, p)` when `p` is not NIL. `frontDeletion` is `DELETE-NODE(L, L.head)`, and `backDeletion` is `DELETE-NODE(L, L.tail)`: the one procedure, with its two special-case tests, covers every deletion.

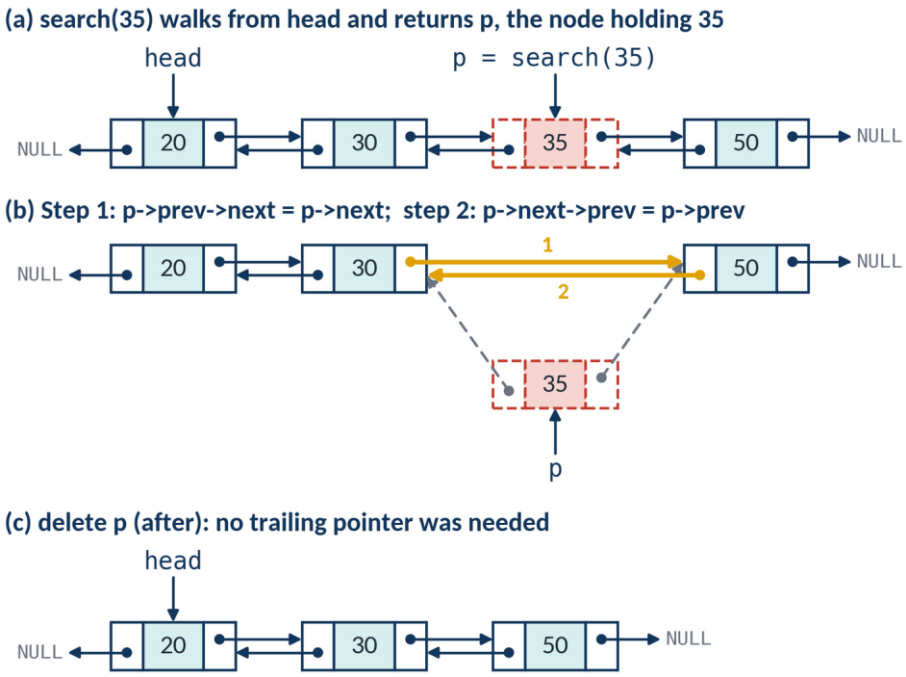


Figure 9.11 `deleteKey(35)`: (a) `search` returns `p`, (b) the predecessor and the successor are linked to each other, (c) `p` is deleted.

Compare this with the procedure `DELETE-KEY` of Section 7.6.5, where a trailing pointer `prev` walked one step behind `cur` during the search, and the node could only be removed because the search had kept its predecessor. That is fine when the deletion follows a search, but useless when the pointer to the node comes from somewhere else: from a hash map (Section 9.9), from an iterator that another part of the program holds, or from the node itself. A singly linked list must then search for the predecessor, $\Theta(n)$; a doubly linked list does not, $O(1)$. Section 9.10 measures the difference.

Cost. `DELETE-NODE` is $O(1)$. `deleteKey` is $\Theta(n)$ in the worst case, for the search.

9.4.5 Checking the links

The invariant Equation (9.1) is cheap to check in a test program. `checkLinks` in **Listing 9.5** walks the list forward once and verifies that every `next` has its matching `prev`, that `head.prev` and `tail.next` are null, and that `head` and `tail` are both null or both non-null. It runs in $\Theta(n)$ time, so it does not belong in the operations themselves, but calling it after every operation in a test is the fastest way to catch an operation that forgot one of its four assignments. A forgotten backward assignment produces a list that prints perfectly forward, the classic invisible bug of this chapter; `checkLinks` makes it visible at once.

9.4.6 The complete program

Listing 9.5 implements the list with `head` and `tail` and every operation of this section. The list type is a small `struct DList` holding the two pointers, passed by reference.

Listing 9.5 `dll_ops.cpp`: the doubly linked list with head and tail: front and back insertion and deletion, insertion after and before a node, sorted insertion, search, deletion by pointer and by key, both traversals, and a link check.

```cpp
// dll_ops.cpp -- a doubly linked list with head AND tail: every end operation
// is O(1), and so is inserting or deleting next to a node we already hold.
#include <cstdio>

struct Node {
    int data;
    Node *prev;
    Node *next;
};

struct DList {
    Node *head = nullptr;                // first node, or nullptr when empty
    Node *tail = nullptr;                // last node, or nullptr when empty
};

bool isEmpty(const DList &L) { return L.head == nullptr; }

void frontInsertion(DList &L, int x) {
    Node *n = new Node{x, nullptr, L.head};
    if (isEmpty(L)) L.tail = n;          // the only node is also the last
    else L.head->prev = n;
    L.head = n;
}

void backInsertion(DList &L, int x) {
    Node *n = new Node{x, L.tail, nullptr};
    if (isEmpty(L)) L.head = n;          // the only node is also the first
    else L.tail->next = n;
    L.tail = n;
}

void insertAfter(DList &L, Node *p, int x) {      // p must be a node of L
    // 1. n->prev = p   2. n->next = p's successor
    Node *n = new Node{x, p, p->next};
    if (p->next == nullptr) L.tail = n;   // p was the last node
    else p->next->prev = n;               // 3. the successor points back at n
    p->next = n;                          // 4. p points forward at n
}
```

Listing 9.5 (Continued) *dll_ops.cpp: the doubly linked list with head and tail: front and back insertion and deletion, insertion after and before a node, sorted insertion, search, deletion by pointer and by key, both traversals, and a link check.*

```
// the mirror image of insertAfter
void insertBefore(DList &L, Node *p, int x) {
    Node *n = new Node{x, p->prev, p};
    if (p->prev == nullptr) L.head = n;   // p was the first node
    else p->prev->next = n;
    p->prev = n;
}

void insertSorted(DList &L, int x) {      // keeps an ascending list ascending
    Node *aux = L.head;
    while (aux != nullptr && aux->data < x) aux = aux->next;
    if (aux == nullptr) backInsertion(L, x);          // larger than everything
    else insertBefore(L, aux, x);  // before the first larger node
}

Node *search(const DList &L, int key) {
    Node *aux = L.head;
    while (aux != nullptr && aux->data != key) aux = aux->next;
    return aux;                           // the node holding key, or nullptr
}

// 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 9.5 (Continued) *dll_ops.cpp: the doubly linked list with head and tail: front and back insertion and deletion, insertion after and before a node, sorted insertion, search, deletion by pointer and by key, both traversals, and a link check.*

```
void showForward(const char *label, const DList &L) {
    std::printf("%-26s", label);
    for (Node *aux = L.head; aux != nullptr; aux = aux->next)
        std::printf("%d ", aux->data);
    std::printf("\n");
}

void showBackward(const char *label, const DList &L) {
    std::printf("%-26s", label);
    for (Node *aux = L.tail; aux != nullptr; aux = aux->prev)
        std::printf("%d ", aux->data);
    std::printf("\n");
}

// Checks the two-way invariant: every next has a matching prev, both ends are
// NULL.
bool checkLinks(const DList &L) {
    if (L.head == nullptr || L.tail == nullptr) return L.head == L.tail;
    if (L.head->prev != nullptr || L.tail->next != nullptr) return false;
    for (Node *p = L.head; p->next != nullptr; p = p->next) {
        if (p->next->prev != p) return false;
    }
    return true;
}

void clear(DList &L) {
    while (L.head != nullptr) {
        Node *del = L.head;
        L.head = L.head->next;
        delete del;
    }
    L.tail = nullptr;
}

int main() {
    DList L;
    backInsertion(L, 20);
    backInsertion(L, 30);
    backInsertion(L, 50);
    frontInsertion(L, 10);
    showForward("start:", L);
```

Listing 9.5 (Continued) *dll_ops.cpp: the doubly linked list with head and tail: front and back insertion and deletion, insertion after and before a node, sorted insertion, search, deletion by pointer and by key, both traversals, and a link check.*

```
    insertAfter(L, search(L, 30), 35);
    showForward("insertAfter(30, 35):", L);
    insertBefore(L, search(L, 10), 5);
    showForward("insertBefore(10, 5):", L);
    insertSorted(L, 40);
    insertSorted(L, 1);
    insertSorted(L, 60);
    showForward("insertSorted 40, 1, 60:", L);
    showBackward("backward:", L);
    std::printf("%-26s%s\n", "links consistent:", checkLinks(L) ? "yes" : "no");

    Node *p = search(L, 35);
    std::printf("%-26s%d <- %d -> %d\n", "search(35):",
                p->prev->data, p->data, p->next->data);
    std::printf("%-26s%s\n", "search(99):",
                search(L, 99) == nullptr ? "not found" : "found");

    deleteKey(L, 35);
    showForward("deleteKey(35):", L);
    frontDeletion(L);
    backDeletion(L);
    showForward("front and back deletion:", L);
    showBackward("backward:", L);
    std::printf("%-26s%s\n", "deleteKey(99):",
                deleteKey(L, 99) ? "deleted" : "not found");
    std::printf("%-26s%s\n", "links consistent:", checkLinks(L) ? "yes" : "no");

    clear(L);
    std::printf("%-26s%s\n", "after clear, empty:", isEmpty(L) ? "yes" : "no");
    return 0;
}
```

```
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
search(99):               not found
deleteKey(35):            1 5 10 20 30 40 50 60
front and back deletion:  5 10 20 30 40 50
backward:                 50 40 30 20 10 5
deleteKey(99):            not found
links consistent:         yes
after clear, empty:       yes
```

Follow the run. The list starts as 10, 20, 30, 50. `insertAfter(30, 35)` puts 35 between 30 and 50; `insertBefore(10, 5)` puts 5 in front of the first node, so `head` changes. The three sorted insertions show the three cases of **Pseudocode 9.8**: 40 goes in the middle, 1 at the front and 60 at the back. The backward line is the forward line reversed, and `checkLinks` agrees. `search(35)` returns a node whose neighbors, read through its own `prev` and `next`, are 30 and 40. The deletions then remove 35 (in the middle), 1 (the front) and 60 (the back), each in $O(1)$ once the node is known.

Table 9.2 collects the costs of the list with head and tail.

Table 9.2 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 9.9**)
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

9.5 A Doubly Linked List with a Head Node

9.5.1 Where the remaining special cases come from

Count the special-case tests in the list of Section 9.4. `frontInsertion` and `backInsertion` test for the empty list. `insertAfter` tests whether `p` was the last node, `insertBefore` whether it was the first. `deleteNode` tests both whether `p` was the first node and whether it was the last. Every one of these tests has the same cause: a node at an *end* of the list has a null pointer on one side, and when an operation would write through that pointer, it must write to `head` or `tail` instead.

Section 7.7 met the same problem in a singly linked list and removed the tests at the front with a *dummy head node*, a node that is always first and holds no data. The same idea works here, and it removes exactly half of the tests.

> **DEFINITION.** A doubly linked list *with a head node* keeps one extra node, the *head node* (or dummy node), permanently at the front. Its data is never used. `head` always points at it, the first real node is `head->next`, and the empty list consists of the head node alone. The list *without a head node* of Sections 9.3 and 9.4 is the one in which `head` points directly at the first real node.

Figure 9.12 shows the empty list (panel a), the list 10, 20, 40 (panel b), and a front insertion (panel c). Because every real node now has a predecessor (the first real node's predecessor is the head node), no operation ever needs to write to `head` again. Front insertion becomes `insertAfter(head, x)`, and it performs the same four steps as an insertion anywhere else. In the same way `deleteNode(p)` can always write `p->prev->next = p->next` without testing, because `p->prev` is never null.

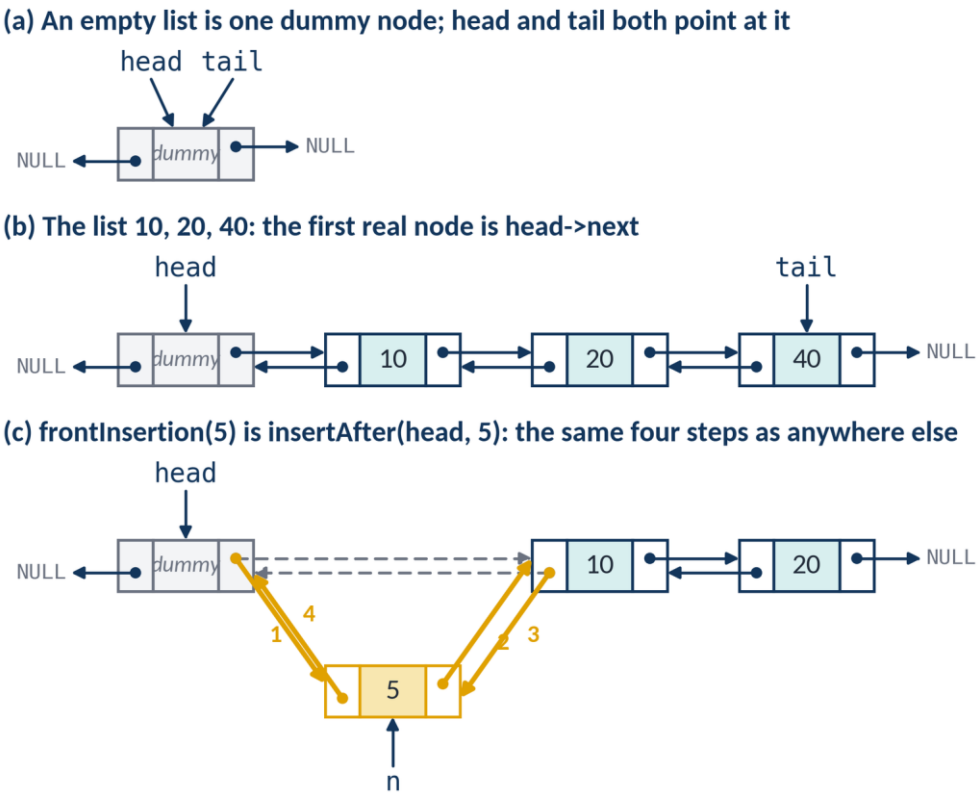


Figure 9.12 A doubly linked list with a head node: (a) the empty list is one dummy node, and `tail` points at it too, (b) the list 10, 20, 40, (c) front insertion is `insertAfter(head, 5)`, with the same four steps as any insertion.

The *back* end still has a null pointer, so one test remains in each operation: `insertAfter` must check whether `p->next` is null (then the new node becomes the last node and `tail` moves), and `deleteNode` must check whether `p->next` is null (then `tail` moves back). A convenient detail keeps even back insertion free of an empty-list test: in the empty list `tail` points at the head node, so `backInsertion(x)` is simply `insertAfter(tail, x)`.

9.5.2 The program

Listing 9.6 implements the list with a head node. The node type and the list type are those of Section 9.4, but `init` creates the head node and `destroy` releases it along with the real nodes.

Listing 9.6 `dll_headnode.cpp`: a doubly linked list with a head node; each operation has one special case left, at the back end.

```cpp
// dll_headnode.cpp -- a doubly linked list WITH a head node: a dummy node that
// is always first. Only the back end (a NULL next) still needs a special case.
#include <cstdio>

struct Node {
    int data;
    Node *prev;
    Node *next;
};

struct DList {
    Node *head;                          // always the dummy head node
    Node *tail;  // last node; the dummy itself when empty
};

void init(DList &L) {
    L.head = new Node{0, nullptr, nullptr};   // the dummy's data is never used
    L.tail = L.head;
}

bool isEmpty(const DList &L) { return L.head->next == nullptr; }

void insertAfter(DList &L, Node *p, int x) {  // p may be the dummy
    Node *n = new Node{x, p, p->next};
    if (p->next != nullptr) p->next->prev = n;
    else L.tail = n;  // the one special case: inserting at the back
    p->next = n;
}

void frontInsertion(DList &L, int x) { insertAfter(L, L.head, x); }
void backInsertion(DList &L, int x) { insertAfter(L, L.tail, x); }

void insertSorted(DList &L, int x) {
    Node *p = L.head;  // stop on the last node smaller than x
    while (p->next != nullptr && p->next->data < x) p = p->next;
    insertAfter(L, p, x);
}

void deleteNode(DList &L, Node *p) {      // p is a real node, never the dummy
    p->prev->next = p->next;  // a predecessor always exists: no test
    if (p->next != nullptr) p->next->prev = p->prev;
    else L.tail = p->prev;  // the one special case: deleting the last
    delete p;
}
```

Listing 9.6 (Continued) *dll_headnode.cpp: a doubly linked list with a head node; each operation has one special case left, at the back end.*

```
Node *search(const DList &L, int key) {
    Node *p = L.head->next;
    while (p != nullptr && p->data != key) p = p->next;
    return p;
}

void frontDeletion(DList &L) { if (!isEmpty(L)) deleteNode(L, L.head->next); }
void backDeletion(DList &L) { if (!isEmpty(L)) deleteNode(L, L.tail); }

void show(const char *label, const DList &L) {
    std::printf("%-22s", label);
    for (Node *p = L.head->next; p != nullptr; p = p->next)
        std::printf("%d ", p->data);
    std::printf("| back: ");
    for (Node *p = L.tail; p != L.head; p = p->prev)
        std::printf("%d ", p->data);
    std::printf("\n");
}

void destroy(DList &L) {  // releases the real nodes AND the dummy
    while (L.head != nullptr) {
        Node *del = L.head;
        L.head = L.head->next;
        delete del;
    }
    L.tail = nullptr;
}

int main() {
    DList L;
    init(L);
    show("empty:", L);
    frontInsertion(L, 20);
    frontInsertion(L, 10);
    backInsertion(L, 40);
    show("10, 20, 40:", L);
    insertSorted(L, 30);
    insertSorted(L, 5);
    insertSorted(L, 50);
    show("sorted 30, 5, 50:", L);
    deleteNode(L, search(L, 30));
    show("delete 30:", L);
    frontDeletion(L);
    backDeletion(L);
    show("front + back del:", L);
    while (!isEmpty(L)) frontDeletion(L);
```

Listing 9.6 (Continued) *dll_headnode.cpp: a doubly linked list with a head node; each operation has one special case left, at the back end.*

```
    show("delete the rest:", L);
    std::printf("%-22s%s\n", "empty again:", isEmpty(L) ? "yes" : "no");
    destroy(L);
    return 0;
}
```

```
empty:                | back:
10, 20, 40:           10 20 40 | back: 40 20 10
sorted 30, 5, 50:     5 10 20 30 40 50 | back: 50 40 30 20 10 5
delete 30:            5 10 20 40 50 | back: 50 40 20 10 5
front + back del:     10 20 40 | back: 40 20 10
delete the rest:      | back:
empty again:          yes
```

Each output line shows the list forward and then backward, and the backward walk stops when it reaches the head node (`p != L.head`), not at null: the head node serves as the stop signal at the front. After all real nodes are deleted, `isEmpty` reports an empty list, and `destroy` still has one node to release, the head node itself. Forgetting that last `delete` is a real and common leak; Section 9.11 catches the same leak for a sentinel.

9.6 Circular Doubly Linked Lists

9.6.1 Closing both chains

In Section 7.9 the last node of a singly linked list was made to point back at the first, and the list became a ring. A doubly linked list has two chains, and a *circular doubly linked list* (*DLLC*) closes both: the last node's `next` points at the first node, and the first node's `prev` points at the last node. No pointer in a non-empty circular list is null.

Figure 9.13 shows the result. Panel (a) is the circular list 1, 5, 2, 4 reached through `head`. A remarkable thing has happened: the list has no `tail` pointer, and yet the last node is only one step away, because it is `head->prev`. One external pointer now gives $O(1)$ access to both ends. Panel (b) is a circle of one node, which points at itself in both directions. Panel (c) inserts 9 at the back: the last node is found as `head->prev`, and four pointers change, as in every insertion of this chapter.

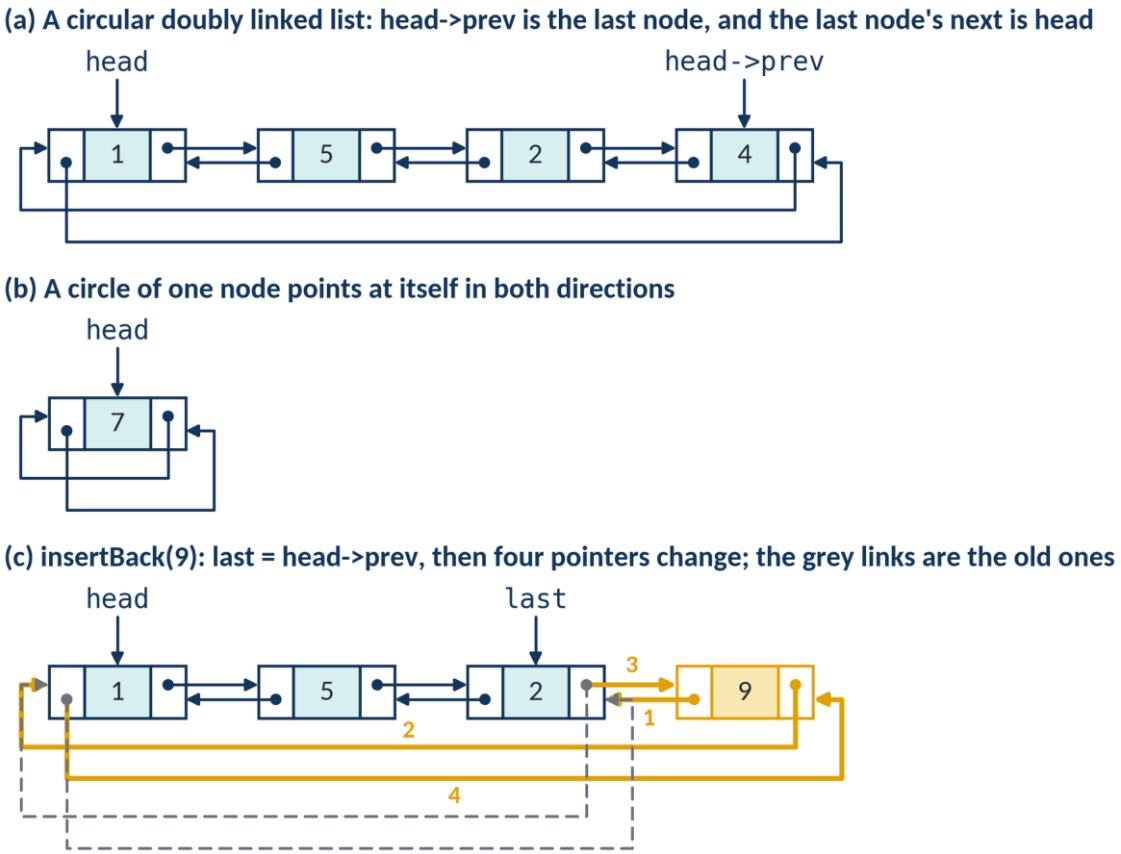


Figure 9.13 A circular doubly linked list reached through `head`: (a) four nodes; the last node is `head->prev`, (b) a circle of one node, (c) `insertBack(9)` links the new node between the last node and `head`.

Look closely at panel (c). Inserting at the back of a circle means inserting *between* the last node and the first node, and inserting at the front means inserting in exactly the same place; the only

difference is whether `head` moves to the new node afterwards. That is why the program of Section 9.6.3 implements `insertFront` as `head = insertBack(head, t)`.

9.6.2 Traversal needs a new stop signal

As in Section 7.9.2, a loop that waits for null never ends, because there is no null. A traversal must stop when it comes back to where it started, and because it must also visit the starting node, the natural loop is a `do`–`while`:

```
const Node *p = head;
do {
    visit(p);
    p = p->next;          // backward: start at head->prev and follow prev
} while (p != head);
```

The empty list must be tested before the loop, because a `do`–`while` runs its body at least once. Deleting from a circle without a sentinel has two special cases: the only node (then `head` becomes null) and the first node (then `head` must move to the next node).

9.6.3 The lobby carousel

The lobby of the Pustaka Ceria library has a screen that shows one featured book at a time. Visitors press **Next** and **Previous**, and the screen wraps around: after the last featured book comes the first again, and before the first comes the last. That is exactly a circular doubly linked list with a cursor. **Listing 9.7** builds a carousel of four titles, presses Next five times and Previous three times, and then removes two titles.

Listing 9.7 `cdll.cpp`: a circular doubly linked list reached through `head`, used as the lobby carousel; `next` and `previous` wrap around at both ends.

```cpp
// cdll.cpp -- a circular doubly linked list reached through head (no sentinel):
// the featured-books carousel in the Pustaka Ceria lobby.
#include <cstdio>
#include <string>

struct Node {
    std::string title;
    Node *prev;
    Node *next;
};

// Inserts a node just before head, i.e. at the back of the circle. O(1).
Node *insertBack(Node *&head, const std::string &t) {
    Node *n = new Node{t, nullptr, nullptr};
    if (head == nullptr) {  // a circle of one points at itself, both ways
        n->prev = n;
        n->next = n;
        head = n;
    } else {
        Node *last = head->prev;  // the last node is one step BACK from head
        n->prev = last;                    // 1.
        n->next = head;                    // 2.
        last->next = n;                    // 3.
        head->prev = n;                    // 4.
    }
    return n;
}

void insertFront(Node *&head, const std::string &t) {
    // same place in the circle; it just becomes head
    head = insertBack(head, t);
}

void deleteNode(Node *&head, Node *p) {   // p must be a node of the circle
    if (p->next == p) {                   // special case 1: the only node
        head = nullptr;
    } else {
        p->prev->next = p->next;
        p->next->prev = p->prev;
        // special case 2: deleting the first node
        if (p == head) head = p->next;
    }
    delete p;
}
```

Listing 9.7 (Continued) *cdll.cpp: a circular doubly linked list reached through head, used as the lobby carousel; next and previous wrap around at both ends.*

```
void showForward(const Node *head) {
    if (head == nullptr) { std::printf("(empty)\n"); return; }
    const Node *p = head;
    do {  // no NULL anywhere: stop on returning to head
        std::printf("[%s] ", p->title.c_str());
        p = p->next;
    } while (p != head);
    std::printf("\n");
}

void showBackward(const Node *head) {
    if (head == nullptr) { std::printf("(empty)\n"); return; }
    const Node *p = head->prev;           // start at the last node
    do {
        std::printf("[%s] ", p->title.c_str());
        p = p->prev;
    } while (p != head->prev);
    std::printf("\n");
}

void destroy(Node *&head) {
    if (head == nullptr) return;
    head->prev->next = nullptr;  // cut the circle open, then free as a line
    while (head != nullptr) {
        Node *del = head;
        head = head->next;
        delete del;
    }
}

int main() {
    Node *head = nullptr;
    insertBack(head, "Laskar Pelangi");
    insertBack(head, "Bumi Manusia");
    insertBack(head, "Filosofi Kopi");
    insertFront(head, "Ronggeng Dukuh Paruk");
    std::printf("forward : "); showForward(head);
    std::printf("backward: "); showBackward(head);
```

Listing 9.7 (Continued) *cdll.cpp: a circular doubly linked list reached through head, used as the lobby carousel; next and previous wrap around at both ends.*

```
    std::printf("\nThe lobby screen, pressing Next 5 times, "
                "then Previous 3 times:\n");
    Node *screen = head;
    for (int i = 0; i < 5; ++i) {
        screen = screen->next;
        std::printf("  next     -> %s\n", screen->title.c_str());
    }
    for (int i = 0; i < 3; ++i) {
        screen = screen->prev;
        std::printf("  previous -> %s\n", screen->title.c_str());
    }

    std::printf("\nremove the book on screen (%s):\n", screen->title.c_str());
    deleteNode(head, screen);
    std::printf("forward : "); showForward(head);
    std::printf("remove the first book (%s):\n", head->title.c_str());
    deleteNode(head, head);
    std::printf("forward : "); showForward(head);
    std::printf("backward: "); showBackward(head);
    destroy(head);
    return 0;
}
```

```
forward : [Ronggeng Dukuh Paruk] [Laskar Pelangi] [Bumi Manusia] [Filosofi Kopi]
backward: [Filosofi Kopi] [Bumi Manusia] [Laskar Pelangi] [Ronggeng Dukuh Paruk]

The lobby screen, pressing Next 5 times, then Previous 3 times:
  next     -> Laskar Pelangi
  next     -> Bumi Manusia
  next     -> Filosofi Kopi
  next     -> Ronggeng Dukuh Paruk
  next     -> Laskar Pelangi
  previous -> Ronggeng Dukuh Paruk
  previous -> Filosofi Kopi
  previous -> Bumi Manusia

remove the book on screen (Bumi Manusia):
forward : [Ronggeng Dukuh Paruk] [Laskar Pelangi] [Filosofi Kopi]
remove the first book (Ronggeng Dukuh Paruk):
forward : [Laskar Pelangi] [Filosofi Kopi]
backward: [Filosofi Kopi] [Laskar Pelangi]
```

After four Next presses from *Ronggeng Dukuh Paruk* the screen is back where it started, and the fifth shows *Laskar Pelangi* again; the Previous presses retrace the circle in the other direction, past the seam between the first and the last title without noticing it. Neither button ever meets a null pointer, so neither needs a test. The first deletion removes the book on the screen, *Bumi Manusia*, a node in the middle. The second removes the first node, which exercises special case 2: `head` moves to *Laskar Pelangi*. `destroy` uses a small trick worth remembering: it cuts the circle open (`head->prev->next = nullptr`) and then releases the nodes with the ordinary loop of Chapter 7.

9.7 The Sentinel: Removing Every Special Case

9.7.1 The idea

Combine the two ideas of the last two sections. From Section 9.5 take the head node, a node without data that is always present. From Section 9.6 take the circle. The result is a circular

doubly linked list with one dataless node, called the *sentinel*, sitting in the circle between the last node and the first.

> **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` is the last node. The empty list is `nil` alone, with `nil.next = nil.prev = nil`. Every real node therefore has a real `prev` and a real `next`: at worst, the sentinel.

This is the representation that Cormen, Leiserson, Rivest and Stein use for linked lists in *Introduction to Algorithms* [2], and it is the representation inside most production linked lists, as Section 9.12 will show. **Figure 9.14** draws the empty list and the list 10, 20, 30.

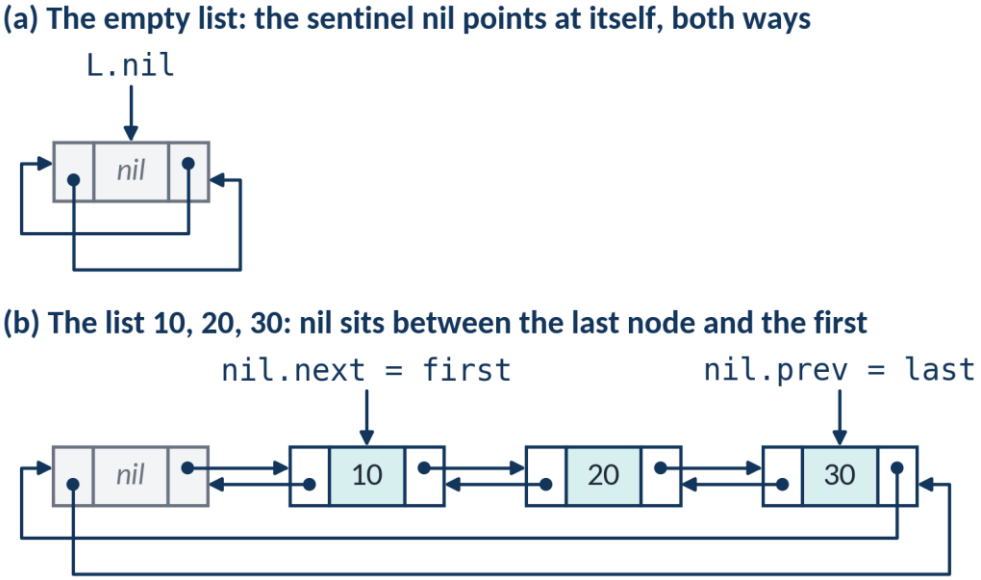


Figure 9.14 A circular doubly linked list with a sentinel: (a) the empty list is the sentinel alone, pointing at itself both ways, (b) the list 10, 20, 30; `nil.next` is the first node and `nil.prev` the last.

9.7.2 Why the special cases disappear

Every special case in Sections 9.3 to 9.6 came from one of two situations: a pointer that was null (so there was no neighbor to update), or an external pointer (`head`, `tail`) that had to change instead of a node's field. With a sentinel, neither situation can arise.

- **No pointer is ever null.** Every node, including `nil`, sits in a circle, so every `prev` and `next` holds the address of a node.
- **No external pointer ever changes.** The only external pointer is the address of `nil`, and `nil` never moves. The "first node" and the "last node" are not stored anywhere; they are read as `nil.next` and `nil.prev`, which are ordinary fields of an ordinary node.

So insertion and deletion can each be written once, with no `if` at all. **Pseudocode 9.10** inserts a node after any node `p`, where `p` may be the sentinel itself.

Pseudocode 9.10 `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 9.11 `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
```

From these two procedures every end operation follows as a one-liner:

$$\begin{aligned} \text{FRONT-INSERTION}(x) &= \text{LIST-INSERT-AFTER}(nil, x), \\ \text{BACK-INSERTION}(x) &= \text{LIST-INSERT-AFTER}(nil.prev, x), \\ \text{FRONT-DELETION}() &= \text{LIST-DELETE}(nil.next), \\ \text{BACK-DELETION}() &= \text{LIST-DELETE}(nil.prev). \end{aligned} \tag{9.3}$$

(The two deletions still require a non-empty list, a precondition of the operation rather than a special case inside it: deleting from an empty list is an error in any representation.)

It is worth checking the two most dangerous situations by hand, because that is where the special cases used to live. **Figure 9.15** does it.

- **Insertion into the empty list** (panel a). Before the call, $nil.next = nil.prev = nil$, and $p = nil$. Line 2 sets $n.prev = nil$. Line 3 sets $n.next = p.next = nil$. Line 4 sets $p.next.prev$, which is $nil.prev$, to n. Line 5 sets $nil.next = n$. Now $nil.next = nil.prev = n$ and $n.next = n.prev = nil$: a correct one-element list, whose first and last node are both n.
- **Deletion of the only node** (panel b). Before the call, $p.prev = p.next = nil$. Line 1 sets $nil.next = nil$; line 2 sets $nil.prev = nil$. The sentinel points at itself both ways: the empty list.

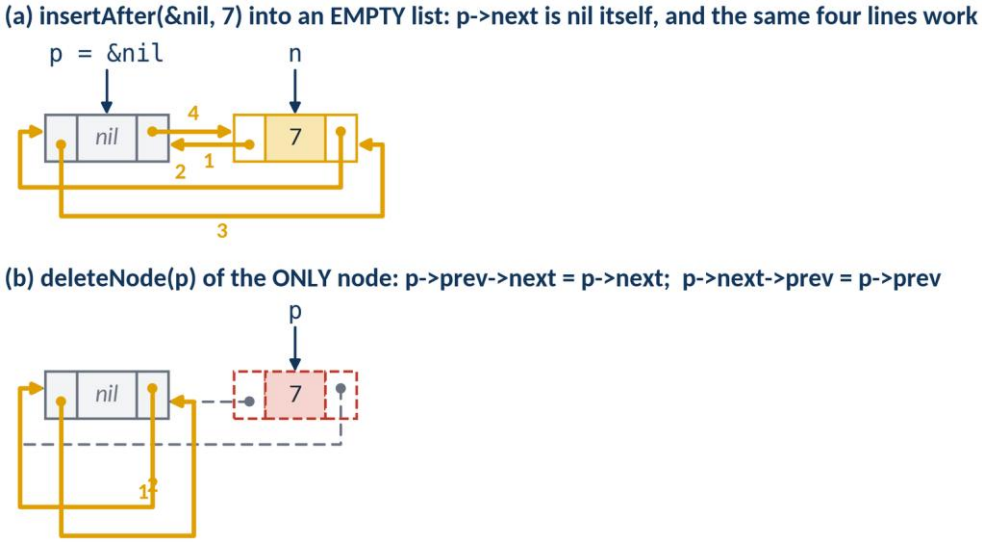


Figure 9.15 The two cases that needed special code before, handled by the ordinary code: (a) inserting after the sentinel into an empty list, (b) deleting the only node.

> **WHY IT WORKS.** A special case is a place where the structure is not *uniform*: the first node has no predecessor, the last node has no successor, the empty list has no nodes at all. A sentinel makes the structure uniform by guaranteeing that every real node has two real neighbors. The price is one node of memory and a small change in how we think: the ends of the list are no longer places, they are the two neighbors of `nil`.

The same argument proves Equation (9.1) is preserved. Suppose it holds before `LIST-INSERT-AFTER`. The only fields written are $n.prev$, $n.next$, $s.prev$ and $p.next$, where s is the old $p.next$. Afterwards $p.next = n$ and $n.prev = p$, so the pair (p, n) satisfies the invariant; $n.next = s$ and

$s.prev = n$, so the pair (n, s) does too; every other pair is untouched. For `LIST-DELETE` the argument is shorter: after the two assignments, p's old predecessor and successor point at each other, and p is gone. Because no case analysis is involved, this proof covers the empty list, the first node, the last node and the only node all at once. In the list without a sentinel, the same proof must be repeated for each branch.

9.7.3 Searching with the key in the sentinel

The sentinel has a second, older use. A sequential search normally tests two things in every step: whether the list has ended, and whether the current node holds the key. Before starting, we can *store the key in the sentinel*. The search then cannot fail to find the key, because the sentinel holds it, so the end-of-list test can be dropped from the loop:

Pseudocode 9.12 does exactly that.

Pseudocode 9.12 `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
```

For an unsuccessful search in a list of n nodes, the ordinary loop `while p ≠ nil and p.data ≠ key` makes n key comparisons and $n+1$ end tests; the sentinel version makes $n+1$ key comparisons and no end tests inside the loop:

$$C_{\text{ordinary}}(n) = 2n + 1, \qquad C_{\text{sentinel}}(n) = n + 1 \quad \text{tests in total.} \tag{9.4}$$

The number of steps is still $\Theta(n)$; what the sentinel halves is the constant factor. Knuth analyzes this trick for sequential search in arrays, where the key is placed just past the last element [3]; it works for any structure that has a spare slot at its end, and a circular list with a sentinel has exactly one.

9.7.4 The program

Listing 9.8 is the circular doubly linked list with a sentinel, as a C++ class. The sentinel is a member of the object, `Node nil_`, not a separately allocated node, so it lives and dies with the list and can never leak. Copying is forbidden (as in Section 7.10.2), because a copy of the object would contain a sentinel whose neighbors point back at the *original* sentinel.

Listing 9.8 `cdll_sentinel.cpp`: a circular doubly linked list with a sentinel; `insertAfter` and `deleteNode` contain no special cases, and every other operation is built on them.

```cpp
// cdll_sentinel.cpp -- a circular doubly linked list with a sentinel node, nil.
// nil.next is the first node and nil.prev the last; in an empty list both are
// &nil. Insertion and deletion have NO special cases at all.
#include <cstdio>

struct Node {
    int data;
    Node *prev;
    Node *next;
};

class SentinelList {
    Node nil_;  // the sentinel lives inside the object
public:
    // empty: nil is alone
    SentinelList() { nil_.prev = &nil_; nil_.next = &nil_; }
    SentinelList(const SentinelList &) = delete;
    SentinelList &operator=(const SentinelList &) = delete;
    ~SentinelList() { while (!isEmpty()) deleteNode(first()); }

    bool isEmpty() const { return nil_.next == &nil_; }
    Node *first() { return nil_.next; }   // &nil_ when empty
    Node *last() { return nil_.prev; }
    const Node *end() const { return &nil_; }

    // 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 9.8 (Continued) *cdll_sentinel.cpp: a circular doubly linked list with a sentinel; insertAfter and deleteNode contain no special cases, and every other operation is built on them.*

```
    void insertSorted(int x) {  // stop on the first node >= x (or on nil)
        Node *p = nil_.next;
        while (p != &nil_ && p->data < x) p = p->next;
        insertAfter(p->prev, x);
    }

    Node *search(int key) {  // one test per step: the key is in nil too
        nil_.data = key;
        Node *p = nil_.next;
        while (p->data != key) p = p->next;
        return (p == &nil_) ? nullptr : p;
    }

    void show(const char *label) const {
        std::printf("%-24s", label);
        for (const Node *p = nil_.next; p != &nil_; p = p->next)
            std::printf("%d ", p->data);
        std::printf("| back: ");
        for (const Node *p = nil_.prev; p != &nil_; p = p->prev)
            std::printf("%d ", p->data);
        std::printf("\n");
    }
};

int main() {
    SentinelList L;
    L.show("empty:");
    L.frontInsertion(20);  // into an EMPTY list: the same four lines
    L.frontInsertion(10);
    L.backInsertion(40);
    L.show("10, 20, 40:");
    L.insertSorted(30);
    L.insertSorted(5);
    L.insertSorted(50);
    L.show("sorted 30, 5, 50:");

    Node *p = L.search(30);
    L.insertAfter(p, 35);
    L.show("insertAfter(30, 35):");
    L.deleteNode(L.search(35));
    L.show("delete 35:");
    std::printf("%-24s%s\n", "search(99):",
                L.search(99) == nullptr ? "not found" : "found");
```

Listing 9.8 (Continued) *cdll_sentinel.cpp: a circular doubly linked list with a sentinel; insertAfter and deleteNode contain no special cases, and every other operation is built on them.*

```
    L.frontDeletion();
    L.backDeletion();
    L.show("front + back del:");
    // the last deletion needs no special case
    while (!L.isEmpty()) L.frontDeletion();
    L.show("delete the rest:");
    std::printf("%-24s%s\n", "nil points at itself:",
                (L.first() == L.end() && L.last() == L.end()) ? "yes" : "no");
    return 0;
}
```

```
empty:                  | back:
10, 20, 40:             10 20 40 | back: 40 20 10
sorted 30, 5, 50:       5 10 20 30 40 50 | back: 50 40 30 20 10 5
insertAfter(30, 35):    5 10 20 30 35 40 50 | back: 50 40 35 30 20 10 5
delete 35:              5 10 20 30 40 50 | back: 50 40 30 20 10 5
search(99):             not found
front + back del:       10 20 30 40 | back: 40 30 20 10
delete the rest:        | back:
nil points at itself:   yes
```

Read `insertAfter` and `deleteNode` in the listing: four assignments and two assignments, and not a single `if`. The first call, `L.frontInsertion(20)`, inserts into an empty list, and the last call of the `while` loop deletes the only remaining node; both use exactly the same lines as every other call. The last output line confirms that the empty list is back to its starting shape, with the sentinel pointing at itself.

9.7.5 The special cases, counted

Table 9.3 counts the special-case tests in insertion after a node and in deletion of a node, the two operations from which all others are built, in each representation of this chapter.

Table 9.3 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

9.8 A Deque for the Returns Desk

9.8.1 What a deque is

Chapter 5 built a stack, which adds and removes at one end, and a queue, which adds at one end and removes at the other. A *deque* (double-ended queue, pronounced “deck”) allows both at both ends.

> **DEFINITION.** A *deque* is a sequence that supports `pushFront`, `pushBack`, `popFront`, `popBack`, `front`, `back`, `isEmpty` and `size`. A deque used only through `pushFront` and `popFront` is a stack; used only through `pushBack` and `popFront` it is a queue.

A circular doubly linked list with a sentinel implements every deque operation in $O(1)$ by Equation (9.3), with no capacity limit. (A circular array, as in Problem 8.7 of Section 8.8.2, also gives $O(1)$ at both ends, but has a capacity, or must be copied when it grows.)

9.8.2 The Pustaka Ceria returns desk

Books returned at the Pustaka Ceria desk are placed on a cart and shelved in the order they arrived: that is a queue. Two things break the queue discipline, though. A returned book that a waiting reader has reserved must be handled *first*, so it goes to the front of the cart. And when the

librarian notices that the last scanned book was scanned by mistake, it is taken back off the *back* of the cart. Additions at both ends and removals at both ends: a deque. **Figure 9.16** shows the cart after the first few events.

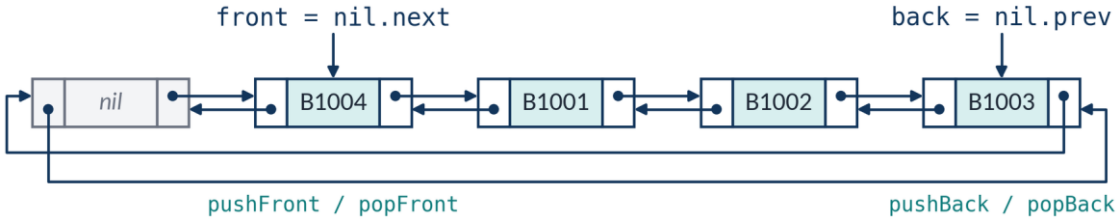


Figure 9.16 The returns desk as a deque on a circular doubly linked list with a sentinel: the front is `nil.next`, the back is `nil.prev`, and all four operations are $O(1)$.

Listing 9.9 implements the deque as a class holding a heap-allocated sentinel, and replays one afternoon at the desk.

Listing 9.9 `deque.cpp`: a deque of books on a circular doubly linked list with a sentinel, used at the Pustaka Ceria returns desk.

```cpp
// deque.cpp -- a deque (double-ended queue) built on a circular doubly linked
// list with a sentinel: the returns desk of the Pustaka Ceria library.
#include <iostream>
#include <stdexcept>
#include <string>

struct Book {
    std::string id;
    std::string title;
};

class Deque {
    struct Node {
        Book data;
        Node *prev;
        Node *next;
    };
    Node *nil_;  // sentinel: nil_->next = front, nil_->prev = back
    int size_ = 0;

    void linkAfter(Node *p, const Book &b) {
        Node *n = new Node{b, p, p->next};
        p->next->prev = n;
        p->next = n;
        ++size_;
    }
    Book unlink(Node *p) {
        if (p == nil_) throw std::underflow_error("pop from an empty deque");
        Book b = p->data;
        p->prev->next = p->next;
        p->next->prev = p->prev;
        delete p;
        --size_;
        return b;
    }
public:
    Deque() : nil_(new Node{Book{}, nullptr, nullptr}) {
        nil_->prev = nil_->next = nil_;       // empty: the sentinel is alone
    }
    Deque(const Deque &) = delete;
    Deque &operator=(const Deque &) = delete;
    ~Deque() {
        while (!isEmpty()) popFront();
        delete nil_;
    }
```

Listing 9.9 (Continued) *deque.cpp: a deque of books on a circular doubly linked list with a sentinel, used at the Pustaka Ceria returns desk.*

```
    bool isEmpty() const { return size_ == 0; }
    int size() const { return size_; }
    void pushFront(const Book &b) { linkAfter(nil_, b); }          // O(1)
    void pushBack(const Book &b) { linkAfter(nil_->prev, b); }     // O(1)
    Book popFront() { return unlink(nil_->next); }                 // O(1)
    Book popBack() { return unlink(nil_->prev); }                  // O(1)
    // precondition: !isEmpty()
    const Book &front() const { return nil_->next->data; }
    const Book &back() const { return nil_->prev->data; }

    void show() const {
        std::cout << "  front |";
        for (Node *p = nil_->next; p != nil_; p = p->next)
            std::cout << ' ' << p->data.id;
        std::cout << " | back   (size " << size_ << ")\n";
    }
};

int main() {
    Deque desk;
    std::cout << "Books returned at the desk go to the back:\n";
    desk.pushBack({"B1001", "Laskar Pelangi"});
    desk.pushBack({"B1002", "Bumi Manusia"});
    desk.pushBack({"B1003", "Filosofi Kopi"});
    desk.show();

    std::cout << "B1004 is reserved by a waiting reader, "
                 "so it jumps to the front:\n";
    desk.pushFront({"B1004", "Cantik Itu Luka"});
    desk.show();

    Book b = desk.popFront();
    std::cout << "shelved or handed out first: "
              << b.id << " " << b.title << '\n';
    b = desk.popBack();
    std::cout << "last scan was a mistake, taken back off: "
              << b.id << " " << b.title << '\n';
    desk.show();

    desk.pushBack({"B1005", "Ronggeng Dukuh Paruk"});
    std::cout << "front is " << desk.front().id
              << ", back is " << desk.back().id << '\n';
    desk.show();
```

Listing 9.9 (Continued) *deque.cpp: a deque of books on a circular doubly linked list with a sentinel, used at the Pustaka Ceria returns desk.*

```
    std::cout << "closing time, the desk is emptied from both ends:\n";
    bool fromFront = true;
    while (!desk.isEmpty()) {
        b = fromFront ? desk.popFront() : desk.popBack();
        std::cout << "  " << (fromFront ? "front: " : "back:  ")
                  << b.id << '\n';
        fromFront = !fromFront;
    }
    try {
        desk.popBack();
    } catch (const std::underflow_error &e) {
        std::cout << "one more popBack(): " << e.what() << '\n';
    }
    return 0;
}
```

```
Books returned at the desk go to the back:
  front | B1001 B1002 B1003 | back   (size 3)
B1004 is reserved by a waiting reader, so it jumps to the front:
  front | B1004 B1001 B1002 B1003 | back   (size 4)
shelved or handed out first: B1004 Cantik Itu Luka
last scan was a mistake, taken back off: B1003 Filosofi Kopi
  front | B1001 B1002 | back   (size 2)
front is B1001, back is B1005
  front | B1001 B1002 B1005 | back   (size 3)
closing time, the desk is emptied from both ends:
  front: B1001
  back:  B1005
  front: B1002
one more popBack(): pop from an empty deque
```

Every operation is two private helpers deep: `linkAfter` is **Pseudocode 9.10** and `unlink` is **Pseudocode 9.11**. `unlink` is also where an empty deque is detected: popping from an empty deque asks to unlink `nil_->next`, which is the sentinel itself, and the class throws `std::underflow_error` instead of destroying its own sentinel, as the last line of the output shows. Unlike the sentinel of **Listing 9.8**, this one is allocated with `new` (its `Book` field holds two strings), so the destructor must release it after all real nodes; Section 9.11 shows the leak that follows when it does not. The Java and Python versions, `DequeDesk.java` and `deque_desk.py` in the code folder, print the same output.

9.9 A “Recently Borrowed” List: The LRU Pattern

9.9.1 The problem

The Pustaka Ceria catalog kiosk has a panel titled *Recently borrowed*. It shows the k most recently borrowed *distinct* titles, newest first; the library uses $k = 4$. When a title is borrowed that is already on the panel, it moves to the top. When a new title is borrowed and the panel is full, the title that was borrowed *least recently* falls off the bottom.

This is the most famous application of the doubly linked list outside textbooks: the *least recently used* (LRU) policy, which decides what to discard from a cache (a web browser’s image cache, a database’s page cache, an operating system’s page table), and which works exactly like the panel. Each borrowing must do three things quickly: find out whether the title is already on the panel, move a title to the top, and remove the title at the bottom.

9.9.2 Two structures working together

No single structure of this book so far does all three in $O(1)$. A list keeps the order, but finding a title in it is a $\Theta(k)$ search. A *hash map*, which Chapter 15 studies (C++ `std::unordered_map`, Java `HashMap`, Python `dict`), finds a key in $O(1)$ expected time but keeps no order. The LRU pattern uses both:

- a circular doubly linked list with a sentinel holds the titles in order of recency: `nil.next` is the newest, `nil.prev` the oldest;
- a hash map maps each title to *the address of its node* in the list.

Figure 9.17 shows the panel after three borrowings, and then after the first title is borrowed again. The map finds the node of *Laskar* in $O(1)$; the node is unlinked from its place by **Pseudocode 9.11** (without being freed) and linked in again after `nil` by **Pseudocode 9.10**. Neither step needs to know where in the list the node was. This is the point of the whole design, and the reason the list must be *doubly* linked: the map hands us a pointer to a node in the middle

of the list, and only a doubly linked node can be removed from the middle without searching for its predecessor.

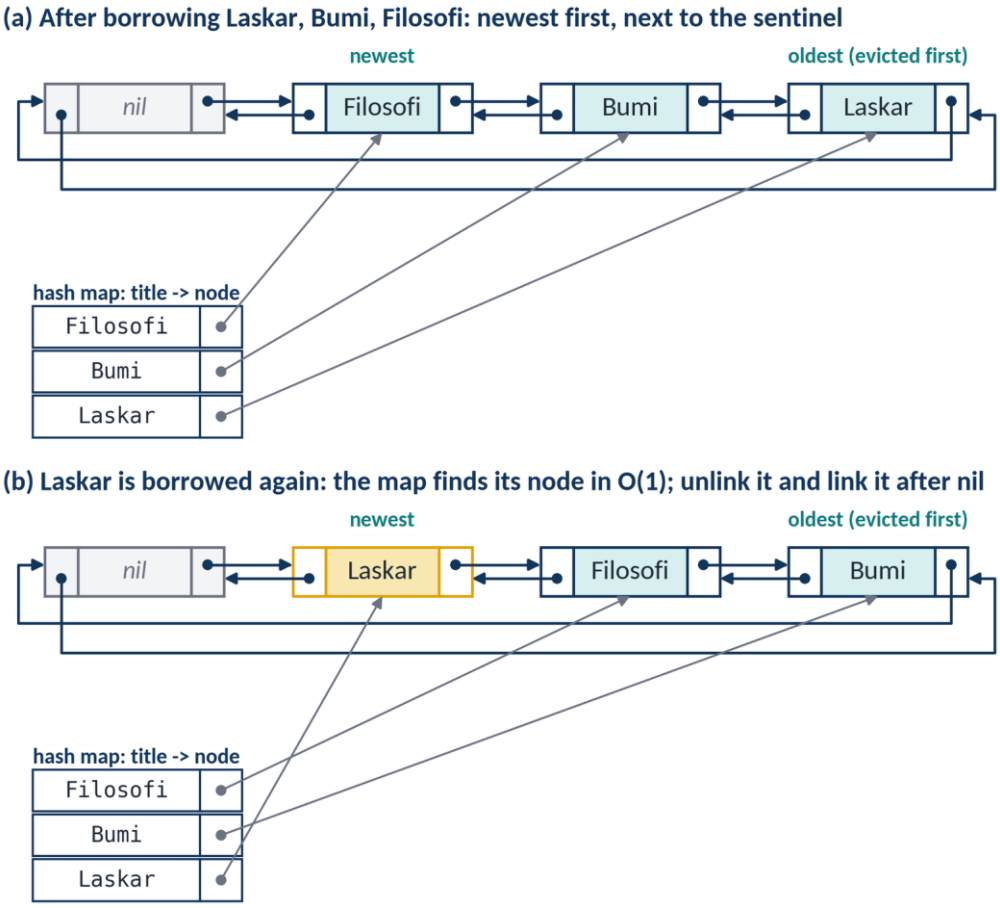


Figure 9.17 The "recently borrowed" panel: a circular doubly linked list with a sentinel keeps the titles from newest to oldest, and a hash map points from each title to its node. (a) After three borrowings; (b) after *Laskar* is borrowed again and moved to the front; the map itself does not change.

Pseudocode 9.13 is the whole algorithm.

Pseudocode 9.13 `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
```

Every line is $O(1)$, with the hash-map operations $O(1)$ on average, so

$$T_{\text{BORROW}} = O(1) \text{ expected, whatever } k \text{ is,} \tag{9.5}$$

whereas the same panel kept in an array, or in a singly linked list without the map, would need a $\Theta(k)$ search for every borrowing, and the singly linked list would need another $\Theta(k)$ walk to find the predecessor of the node to move.

9.9.3 The program

Listing 9.10 implements the panel with an `std::unordered_map` from title to node and replays eight borrowings with $k = 4$.

Listing 9.10 `recently_borrowed.cpp`: the "recently borrowed" panel, an LRU list built from a circular doubly linked list with a sentinel and an `std::unordered_map` from title to node.

```
// recently_borrowed.cpp -- the "recently borrowed" panel of Pustaka Ceria: the
// k most recently borrowed distinct titles, newest first (an LRU list). A
// circular doubly linked list with a sentinel keeps the order; a hash map finds
// a title's node in O(1).
#include <iostream>
#include <string>
#include <unordered_map>

class RecentlyBorrowed {
    struct Node {
        std::string title;
        Node *prev;
        Node *next;
    };
    Node nil_;  // sentinel: nil_.next = newest, nil_.prev = oldest
    std::unordered_map<std::string, Node *> where_;   // title -> its node
    std::size_t capacity_;

    void unlink(Node *p) {
        p->prev->next = p->next;
        p->next->prev = p->prev;
    }
    void linkFront(Node *p) {             // put p right after the sentinel
        p->prev = &nil_;
        p->next = nil_.next;
        nil_.next->prev = p;
        nil_.next = p;
    }
public:
    explicit RecentlyBorrowed(std::size_t k)
        // empty: the sentinel is alone
        : nil_{"", &nil_, &nil_}, capacity_(k) {}
    RecentlyBorrowed(const RecentlyBorrowed &) = delete;
    RecentlyBorrowed &operator=(const RecentlyBorrowed &) = delete;
    ~RecentlyBorrowed() {
        while (nil_.next != &nil_) {
            Node *del = nil_.next;
            unlink(del);
            delete del;
        }
    }
```

Listing 9.10 (Continued) *recently_borrowed.cpp: the "recently borrowed" panel, an LRU list built from a circular doubly linked list with a sentinel and an std::unordered_map from title to node.*

```
    // Records one borrowing. Returns the title that fell off the panel, or "".
    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;
    }

    void show() const {
        std::cout << "   panel:";
        for (const Node *p = nil_.next; p != &nil_; p = p->next)
            std::cout << " [" << p->title << "]";
        std::cout << '\n';
    }
};

int main() {
    RecentlyBorrowed panel(4);
    const char *loans[] = {"Laskar Pelangi", "Bumi Manusia", "Filosofi Kopi",
                           "Laskar Pelangi", "Cantik Itu Luka",
                           "Ronggeng Dukuh Paruk", "Filosofi Kopi",
                           "Negeri 5 Menara"};
    for (const char *t : loans) {
        std::string gone = panel.borrow(t);
        std::cout << "borrow " << t;
        if (!gone.empty()) std::cout << "  (drops off: " << gone << ")";
        std::cout << '\n';
        panel.show();
    }
    return 0;
}
```

```
borrow Laskar Pelangi
   panel: [Laskar Pelangi]
borrow Bumi Manusia
   panel: [Bumi Manusia] [Laskar Pelangi]
borrow Filosofi Kopi
   panel: [Filosofi Kopi] [Bumi Manusia] [Laskar Pelangi]
borrow Laskar Pelangi
   panel: [Laskar Pelangi] [Filosofi Kopi] [Bumi Manusia]
borrow Cantik Itu Luka
   panel: [Cantik Itu Luka] [Laskar Pelangi] [Filosofi Kopi] [Bumi Manusia]
borrow Ronggeng Dukuh Paruk  (drops off: Bumi Manusia)
   panel: [Ronggeng Dukuh Paruk] [Cantik Itu Luka] [Laskar Pelangi] [Filosofi Kopi]
borrow Filosofi Kopi
   panel: [Filosofi Kopi] [Ronggeng Dukuh Paruk] [Cantik Itu Luka] [Laskar Pelangi]
borrow Negeri 5 Menara  (drops off: Laskar Pelangi)
   panel: [Negeri 5 Menara] [Filosofi Kopi] [Ronggeng Dukuh Paruk] [Cantik Itu Luka]
```

Trace the eight borrowings. The first three titles fill the panel from the top. The fourth borrowing is *Laskar Pelangi* again: it is found in the map and moved from the bottom to the top, so no title is

dropped. *Cantik Itu Luka* fills the fourth place. *Ronggeng Dukuh Paruk* overflows the panel, and the title at the bottom, *Bumi Manusia*, drops off: it was borrowed second, and nothing has touched it since. *Filosofi Kopi* moves to the top, and finally *Negeri 5 Menara* pushes out *Laskar Pelangi*, which had become the least recent. The Java and Python versions, `RecentlyBorrowed.java` and `recently_borrowed.py`, print the same lines; Section 9.12 shows that both languages also offer the whole pattern ready-made.

9.10 Costs Compared: Singly Linked, Doubly Linked, Array

9.10.1 The table

Table 9.4 compares the three sequential structures of this book operation by operation: a dynamic array (`std::vector`, Java's `ArrayList`, Python's `list`), the singly linked list with head and tail of Section 7.8, and the doubly linked list with head and tail (or, equivalently, with a sentinel) of this chapter. "A held position" means that the program already has a pointer or iterator to the node, or the index into the array.

Table 9.4 The dynamic array, the singly linked list and the doubly linked list compared, for n elements.

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 front	$\Theta(n)$	$O(1)$	$O(1)$
insert at the back	$O(1)$ amortized	$O(1)$	$O(1)$
delete at the back	$O(1)$	$\Theta(n)$	$O(1)$
insert after a held position	$\Theta(n)$ (shift)	$O(1)$	$O(1)$
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, unsorted	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
search by value, sorted	$\Theta(\lg n)$ (binary search)	$\Theta(n)$	$\Theta(n)$
traverse forward	$\Theta(n)$, cache-friendly	$\Theta(n)$	$\Theta(n)$
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
other elements move on insert or delete	yes	never	never

Three rows deserve a comment.

Access by index. A doubly linked list with `tail` can walk to index k from whichever end is nearer. From the front it takes k steps; from the back, $n-1-k$ steps. So the cost is

$$T_{\text{reach}}(k) = \min(k,\ n-1-k), \qquad \frac{1}{n}\sum_{k=0}^{n-1} \min\left(k,\ n-1-k\right) \approx \frac{n}{4}. \tag{9.6}$$

The average follows because the minimum rises from 0 to about $n/2$ over the first half of the indexes and falls back over the second half, a triangle whose average height is half of its peak, $n/4$.

That halves the average of the singly linked list, $(n-1)/2$ by the formula of Section 7.3.1, but it is still $\Theta(n)$. Java's `LinkedList.get(i)` does exactly this: its documentation says that indexing operations traverse the list "from the beginning or the end, whichever is closer to the specified index" [4].

Delete at the back, and at a held position. These two rows are the reason this chapter exists: from $\Theta(n)$ in the singly linked list to $O(1)$ in the doubly linked list.

Extra memory and moving elements. The price is one more pointer per element (Equation (9.2)); the prize, shared with the singly linked list, is that inserting or deleting never moves any other element, so pointers to other nodes stay valid. For an array, both columns are reversed.

9.10.2 Measuring what the second pointer buys

The difference in the back-deletion row is easy to predict exactly. Emptying a singly linked list of n nodes from the back performs n back deletions. The one that finds $m \geq 2$ nodes must walk $m-2$ steps to the node before `tail`, and the last one (one node) walks none, so the total number of walking steps is

$$\sum_{m=2}^{n} (m-2) = 0 + 1 + \cdots + (n-2) = \frac{(n-1)(n-2)}{2}, \tag{9.7}$$

which is $\Theta(n^2)$, against exactly zero walking steps (and $\Theta(n)$ work in total) for the doubly linked list. The second experiment deletes half of the nodes, chosen at random, when the program already holds a pointer to each one: the singly linked list must search for each victim's predecessor from `head`, the doubly linked list unlinks it in $O(1)$.

Listing 9.11 runs both experiments for n from 2,500 to 20,000, each three times, and reports the fastest run.

Listing 9.11 `bench.cpp`: emptying a list from the back, and deleting nodes through pointers the program already holds, for singly and doubly linked lists.

```cpp
// bench.cpp -- what the prev pointer buys, measured.
// (1) Empty a list of n nodes from the back: singly linked with head and tail
//     (must walk to the node before tail) versus doubly linked (tail->prev).
// (2) Delete n/2 nodes in random order, given a pointer to each node: singly
//     linked (must search for the predecessor) versus doubly linked (O(1)
//     unlink).
#include <algorithm>
#include <chrono>
#include <cstdio>
#include <random>
#include <vector>

struct SNode { int data; SNode *next; };
struct DNode { int data; DNode *prev; DNode *next; };

using Clock = std::chrono::steady_clock;
double ms(Clock::time_point a, Clock::time_point b) {
    return std::chrono::duration<double, std::milli>(b - a).count();
}

double sllBackDeletions(int n) {
    SNode *head = nullptr, *tail = nullptr;
    for (int i = 0; i < n; ++i) {
        SNode *s = new SNode{i, nullptr};
        if (!head) head = s; else tail->next = s;
        tail = s;
    }
    auto t0 = Clock::now();
    while (head != nullptr) {
        if (head == tail) { delete head; head = tail = nullptr; break; }
        SNode *aux = head;
        while (aux->next != tail) aux = aux->next;   // the walk: Theta(n)
        delete tail;
        tail = aux;
        tail->next = nullptr;
    }
    return ms(t0, Clock::now());
}
```

Listing 9.11 (Continued) *bench.cpp: emptying a list from the back, and deleting nodes through pointers the program already holds, for singly and doubly linked lists.*

```
double dllBackDeletions(int n) {
    DNode *head = nullptr, *tail = nullptr;
    for (int i = 0; i < n; ++i) {
        DNode *d = new DNode{i, tail, nullptr};
        if (!head) head = d; else tail->next = d;
        tail = d;
    }
    auto t0 = Clock::now();
    while (tail != nullptr) {
        DNode *del = tail;
        tail = tail->prev;                              // one step: O(1)
        if (tail) tail->next = nullptr; else head = nullptr;
        delete del;
    }
    return ms(t0, Clock::now());
}

template <typename N>
std::vector<N *> randomHalf(std::vector<N *> v, unsigned seed) {
    std::mt19937 rng(seed);
    std::shuffle(v.begin(), v.end(), rng);
    v.resize(v.size() / 2);
    return v;
}

double sllDeleteByPointer(int n) {
    std::vector<SNode *> all;
    SNode *head = nullptr, *tail = nullptr;
    for (int i = 0; i < n; ++i) {
        SNode *s = new SNode{i, nullptr};
        if (!head) head = s; else tail->next = s;
        tail = s;
        all.push_back(s);
    }
    std::vector<SNode *> victims = randomHalf(all, 42);
    auto t0 = Clock::now();
    for (SNode *p : victims) {
        if (p == head) { head = p->next; delete p; continue; }
        SNode *q = head;
        while (q->next != p) q = q->next;  // find the predecessor: O(n)
        q->next = p->next;
        delete p;
    }
    double t = ms(t0, Clock::now());
    while (head) { SNode *d = head; head = head->next; delete d; }
    return t;
```

Listing 9.11 (Continued) *bench.cpp: emptying a list from the back, and deleting nodes through pointers the program already holds, for singly and doubly linked lists.*

```
}

double dllDeleteByPointer(int n) {
    std::vector<DNode *> all;
    DNode *head = nullptr, *tail = nullptr;
    for (int i = 0; i < n; ++i) {
        DNode *d = new DNode{i, tail, nullptr};
        if (!head) head = d; else tail->next = d;
        tail = d;
        all.push_back(d);
    }
    std::vector<DNode *> victims = randomHalf(all, 42);
    auto t0 = Clock::now();
    for (DNode *p : victims) {                        // unlink: O(1)
        if (p->prev) p->prev->next = p->next; else head = p->next;
        if (p->next) p->next->prev = p->prev; else tail = p->prev;
        delete p;
    }
    double t = ms(t0, Clock::now());
    while (head) { DNode *d = head; head = head->next; delete d; }
    return t;
}

double best(double (*f)(int), int n) {           // the fastest of three runs
    double t = f(n);
    for (int r = 0; r < 2; ++r) t = std::min(t, f(n));
    return t;
}

int main() {
    std::printf("%8s | %14s %14s | %14s %14s\n", "n",
                "SLL back del", "DLL back del", "SLL by ptr", "DLL by ptr");
    for (int n = 2500; n <= 20000; n *= 2) {
        std::printf("%8d | %11.2f ms %11.2f ms | %11.2f ms %11.2f ms\n", n,
                    best(sllBackDeletions, n), best(dllBackDeletions, n),
                    best(sllDeleteByPointer, n), best(dllDeleteByPointer, n));
    }
    return 0;
}
```

One run on the machine used for this book (compiled with `-O2`) printed:

```
       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
```

Figure 9.18 plots the same numbers on logarithmic axes. Each doubling of n multiplies the singly linked times by about 4 (8.68, 36.40, 156.41, 592.91 ms), the signature of the $\Theta(n^2)$ of Equation (9.7), and the doubly linked times by about 2 (0.02, 0.04, 0.08, 0.17 ms), the signature of $\Theta(n)$. At $n = 20{,}000$ the doubly linked list empties itself from the back roughly 3,500 times faster, and deleting 10,000 held nodes is roughly 2,900 times faster (232.24 ms against 0.08 ms). At 20,000 nodes, Equation (9.7) gives about 2×10^8 walking steps, done in 0.59 s, about 3 ns per step: the singly linked list is not slow per step, it simply takes far too many steps.

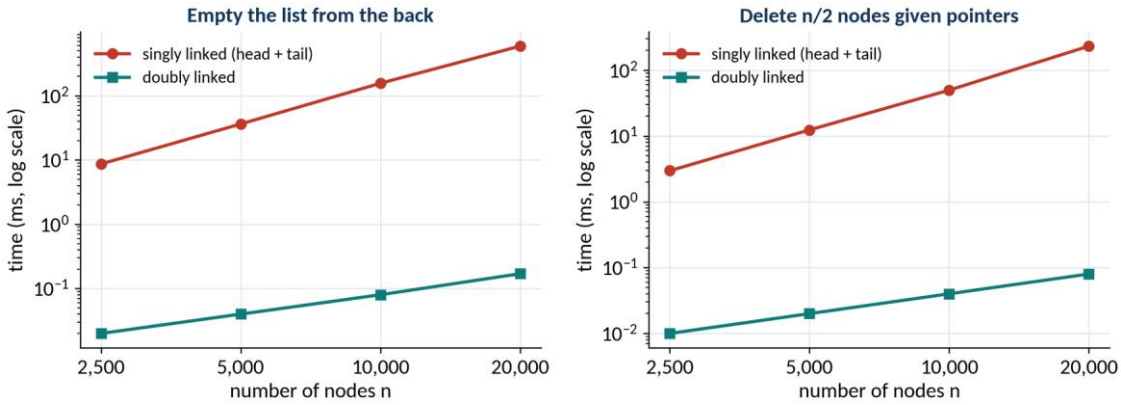


Figure 9.18 The measurements of **Listing 9.11** on logarithmic axes: the singly linked list's time grows fourfold per doubling of n (quadratic), the doubly linked list's twofold (linear).

The timings themselves depend on the machine; the shapes do not. The measurements are also a reminder of Section 7.12.4: when positions are *not* already held, a search dominates, and an array usually wins. A doubly linked list is the right tool when the program keeps pointers into the middle of a sequence and needs to change it there.

9.11 Memory Safety for Two-Way Links

9.11.1 What changes with two links

Every memory bug of Section 7.11 (leaks, uses after free, dangling links, double frees) can happen in a doubly linked list too, and the tools of that section, Valgrind [5] and AddressSanitizer [6], find them in the same way. Two links per node add a new family of bugs, all of them caused by updating one direction and not the other. They are especially treacherous because the forward direction, the one every quick test prints, usually still looks right. **Figure 9.19** shows three of them.

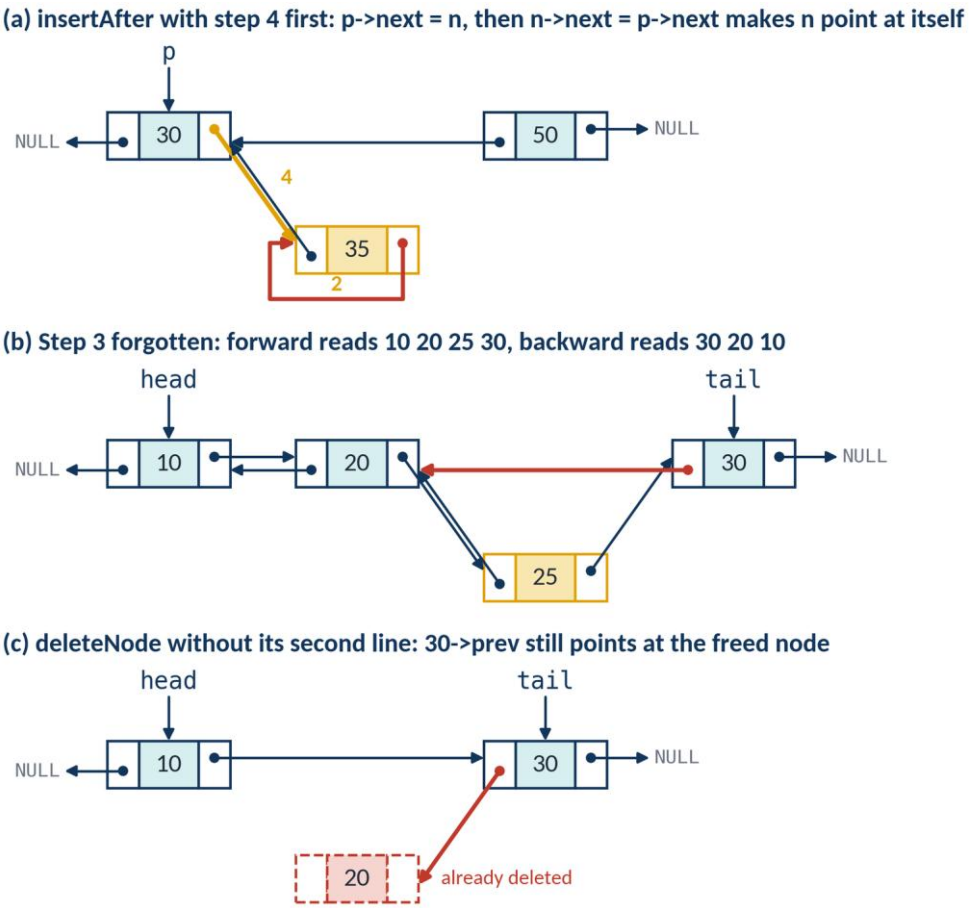


Figure 9.19 Three two-way-link bugs: (a) `insertAfter` with its steps in the wrong order makes the new node point at itself, (b) a forgotten backward link makes the list read differently in the two directions, (c) a forgotten backward link in a deletion leaves a pointer into freed memory.

- **Wrong order** (panel a). If `insertAfter` runs step 4 (`p->next = n`) before step 2 (`n->next = p->next`), then step 2 copies the new value of `p->next`, which is `n`, and the new node points at itself. The forward chain now ends in an infinite loop at 35, and 50 is unreachable going forward.
- **Half a link on insertion** (panel b). If `insertAfter` forgets step 3 (`p->next->prev = n`), the forward chain is 10, 20, 25, 30 and the backward chain is 30, 20, 10. No memory error occurs: every pointer points at a live node. Only Equation (9.1), checked by `checkLinks` of Section 9.4.5, reveals the damage, and it will cause a real memory error later, the first time an operation trusts 30's `prev`.
- **Half a link on deletion** (panel c). If `deleteNode` forgets its second line (`p->next->prev = p->prev`), 30's `prev` still holds the address of the deleted node 20. The next backward traversal reads freed memory.

Listing 9.12 contains the last two bugs and a sentinel leak, selected by a command-line argument.

Listing 9.12 `memory_bugs.cpp`: a correct run and three deliberate bugs of doubly linked lists (`halflink`, `stale`, `leak`).

```cpp
// memory_bugs.cpp -- three classic doubly-linked-list bugs, one per argument.
//   ./memory_bugs           correct program
//   ./memory_bugs halflink  insertAfter forgets the backward link (wrong list)
//   ./memory_bugs stale     deleteNode forgets the backward link (freed memory)
//   ./memory_bugs leak      destroy frees the nodes but forgets the sentinel
#include <cstdio>
#include <cstring>

struct Node {
    int data;
    Node *prev;
    Node *next;
};

struct DList {
    Node *head = nullptr;
    Node *tail = nullptr;
};

void backInsertion(DList &L, int x) {
    Node *n = new Node{x, L.tail, nullptr};
    if (L.head == nullptr) L.head = n;
    else L.tail->next = n;
    L.tail = n;
}

// assumes p is not the last node
void insertAfterWrong(DList &L, Node *p, int x) {
    Node *n = new Node{x, p, p->next};
    p->next = n;                          // BUG: p->next->prev = n is missing
    (void)L;
}

void deleteNodeWrong(DList &L, Node *p) {  // assumes p is a middle node
    p->prev->next = p->next;  // BUG: p->next->prev = p->prev is missing
    delete p;
    (void)L;
}
```

Listing 9.12 (Continued) *memory_bugs.cpp: a correct run and three deliberate bugs of doubly linked lists (halflink, stale, leak).*

```
void show(const DList &L) {
    std::printf("forward : ");
    for (Node *p = L.head; p != nullptr; p = p->next)
        std::printf("%d ", p->data);
    std::printf("\nbackward: ");
    for (Node *p = L.tail; p != nullptr; p = p->prev)
        std::printf("%d ", p->data);
    std::printf("\n");
}

void clear(DList &L) {
    while (L.head != nullptr) {
        Node *del = L.head;
        L.head = L.head->next;
        delete del;
    }
    L.tail = nullptr;
}

void sentinelListLeak() {
    Node *nil = new Node{0, nullptr, nullptr};    // a heap-allocated sentinel
    nil->prev = nil->next = nil;
    for (int x = 1; x <= 3; ++x) {  // back insertion, no special case
        Node *n = new Node{x, nil->prev, nil};
        nil->prev->next = n;
        nil->prev = n;
    }
    Node *p = nil->next;
    while (p != nil) {  // frees the three real nodes ...
        Node *del = p;
        p = p->next;
        delete del;
    }
}  // BUG: ... but never the sentinel

int main(int argc, char *argv[]) {
    const char *mode = argc > 1 ? argv[1] : "ok";
    if (std::strcmp(mode, "leak") == 0) {
        sentinelListLeak();
        std::printf("done (leak)\n");
        return 0;
    }
    DList L;
    for (int x = 10; x <= 30; x += 10) backInsertion(L, x);
    if (std::strcmp(mode, "halflink") == 0) {
        insertAfterWrong(L, L.head->next, 25);        // between 20 and 30
```

Listing 9.12 (Continued) *memory_bugs.cpp: a correct run and three deliberate bugs of doubly linked lists (halflink, stale, leak).*

```
        show(L);  // forward has 25, backward does not
    } else if (std::strcmp(mode, "stale") == 0) {
        deleteNodeWrong(L, L.head->next);             // delete 20
        show(L);  // 30->prev still points at 20
    } else {
        show(L);
    }
    clear(L);
    std::printf("done (%s)\n", mode);
    return 0;
}
```

9.11.2 What the tools say

The `halflink` run, under Valgrind or AddressSanitizer, prints only this:

```
forward : 10 20 25 30
backward: 30 20 10
done (halflink)
```

Neither tool reports anything, because nothing illegal happened: the program read and wrote only memory it owned. This is an important limit to understand. Memory checkers find *memory* errors; a broken invariant is a *logic* error, and only a check of the invariant, or a test that walks the list in both directions, finds it. That is why the programs of this chapter print every list both ways.

The `stale` run is a real memory error, and the program prints wrong output without crashing: the backward line still shows 20, a value that was deleted. Valgrind reports it (compile with `-g -O0`, as in Section 7.11.2):

```
==22996== Memcheck, a memory error detector
==22996== Command: ./memory_bugs stale
==22996==
==22996== Invalid read of size 4
==22996==    at 0x10938F: show(DList const&) (memory_bugs.cpp:46)
==22996==    by 0x109638: main (memory_bugs.cpp:89)
==22996==  Address 0x4e260e0 is 0 bytes inside a block of size 24 free'd
==22996==    at 0x484A61D: operator delete(void*, unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22996==    by 0x109305: deleteNodeWrong(DList&, Node*) (memory_bugs.cpp:36)
==22996==    by 0x10962C: main (memory_bugs.cpp:88)
==22996==  Block was alloc'd at
==22996==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22996==    by 0x109225: backInsertion(DList&, int) (memory_bugs.cpp:21)
==22996==    by 0x1095AD: main (memory_bugs.cpp:83)
==22996==
==22996== Invalid read of size 8
==22996==    at 0x1093AB: show(DList const&) (memory_bugs.cpp:45)
==22996==    by 0x109638: main (memory_bugs.cpp:89)
==22996==  Address 0x4e260e8 is 8 bytes inside a block of size 24 free'd
==22996==    at 0x484A61D: operator delete(void*, unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22996==    by 0x109305: deleteNodeWrong(DList&, Node*) (memory_bugs.cpp:36)
==22996==    by 0x10962C: main (memory_bugs.cpp:88)
==22996==  Block was alloc'd at
==22996==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22996==    by 0x109225: backInsertion(DList&, int) (memory_bugs.cpp:21)
==22996==    by 0x1095AD: main (memory_bugs.cpp:83)
==22996==
forward : 10 30
backward: 30 20 10
done (stale)
==22996==
==22996== HEAP SUMMARY:
==22996==     in use at exit: 0 bytes in 0 blocks
==22996==   total heap usage: 5 allocs, 5 frees, 77,896 bytes allocated
==22996==
==22996== All heap blocks were freed -- no leaks are possible
==22996==
==22996== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

The report has the three-part shape we met in Chapter 7. The first invalid read happened in `show` at line 46, the body of the backward loop, called from `main` at line 89. The block had been freed by

`deleteNodeWrong` at line 36, called from line 88. It had been allocated by `backInsertion` at line 21. The first error is a read of size 4 at offset 0 of a 24-byte block, the `data` field of the freed node 20; the second, at line 45, is a read of size 8 at offset 8, its `prev` field, which the loop's step `p = p->prev` follows to reach 10. Offset 8 is exactly where **Figure 9.2** placed `prev`. AddressSanitizer reports the same bug at its first bad access and stops the program:

```
=================================================================
==23023==ERROR: AddressSanitizer: heap-use-after-free on address 0x503000000070 at pc
0x55a6ee459b27 bp 0x7ffeb6f481d0 sp 0x7ffeb6f481c0
READ of size 4 at 0x503000000070 thread T0
    #0 0x55a6ee459b26 in show(DList const&) memory_bugs.cpp:46
    #1 0x55a6ee45a445 in main memory_bugs.cpp:89

0x503000000070 is located 0 bytes inside of 24-byte region [0x503000000070,0x503000000088)
freed by thread T0 here:
    #0 0x7f86376ff5e8 in operator delete(void*, unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:164
    #1 0x55a6ee45987b in deleteNodeWrong(DList&, Node*) memory_bugs.cpp:36
    #2 0x55a6ee45a43d in main memory_bugs.cpp:88

previously allocated by thread T0 here:
    #0 0x7f86376fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x55a6ee459389 in backInsertion(DList&, int) memory_bugs.cpp:21
    #2 0x55a6ee45a37f in main memory_bugs.cpp:83

SUMMARY: AddressSanitizer: heap-use-after-free memory_bugs.cpp:46 in show(DList const&)
```

Finally, the `leak` run frees the three real nodes of a sentinel list but forgets the heap-allocated sentinel:

```
==22999== Memcheck, a memory error detector
==22999== Command: ./memory_bugs leak
==22999==
done (leak)
==22999==
==22999== HEAP SUMMARY:
==22999==     in use at exit: 24 bytes in 1 blocks
==22999==   total heap usage: 6 allocs, 5 frees, 77,920 bytes allocated
==22999==
==22999== 24 bytes in 1 blocks are definitely lost in loss record 1 of 1
==22999==    at 0x4846FA3: operator new(unsigned long) (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22999==    by 0x10943C: sentinelListLeak() (memory_bugs.cpp:60)
==22999==    by 0x10956A: main (memory_bugs.cpp:78)
==22999==
==22999== LEAK SUMMARY:
==22999==    definitely lost: 24 bytes in 1 blocks
==22999==    indirectly lost: 0 bytes in 0 blocks
==22999==      possibly lost: 0 bytes in 0 blocks
==22999==    still reachable: 0 bytes in 0 blocks
==22999==         suppressed: 0 bytes in 0 blocks
==22999==
==22999== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Exactly one block of 24 bytes, one node, is *definitely lost*, allocated in `sentinelListLeak` at line 60. Compare this with the leaked list of Section 7.11.2, which showed one definite leak and two indirect ones: here the three real nodes were released correctly, and only the sentinel, whose address was held by nothing but the local variable `nil`, remained. LeakSanitizer says the same:

```
=================================================================
==23028==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 24 byte(s) in 1 object(s) allocated from:
    #0 0x7f33f48fe548 in operator new(unsigned long)
../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x559d07462d5d in sentinelListLeak() memory_bugs.cpp:60
    #2 0x559d074632af in main memory_bugs.cpp:78

SUMMARY: AddressSanitizer: 24 byte(s) leaked in 1 allocation(s).
```

And the certificate for a correct program: **Listing 9.5** under Valgrind ends with

```
==23008== HEAP SUMMARY:
==23008==     in use at exit: 0 bytes in 0 blocks
==23008==   total heap usage: 11 allocs, 11 frees, 78,040 bytes allocated
==23008==
==23008== All heap blocks were freed -- no leaks are possible
==23008==
==23008== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Eleven allocations: nine nodes created by the run and two blocks belonging to the run-time library, all freed. Every C++ program of this chapter except the timing program was run under Valgrind with `--leak-check=full` and under `-fsanitize=address,undefined`, and every one produced its normal output and no report.

9.11.3 A checklist for two-way links

1. **Four on insert, two on delete.** An insertion between two nodes changes four pointers; a deletion changes two. Count them in every function you write.
2. **The new node first, the neighbors next, the link you navigated by last.** In `insertAfter(p)`, set `n->prev` and `n->next`, then `p->next->prev`, and `p->next` last (Section 9.4.2).
3. **Every forward write has a backward twin.** After writing `a->next = b`, ask where `b->prev = a` is written. After unlinking `p` forward, ask where it is unlinked backward.
4. **Both external pointers.** Without a sentinel, the first insertion into an empty list must set both `head` and `tail`, and deleting the only node must reset both.
5. **Release the sentinel or head node.** If it is allocated with `new`, it must be deleted after the real nodes; if it is a member of the list object, copying the object must be forbidden.
6. **Print both ways, check the invariant, run the tools.** A two-way list that has only been tested forward has been half tested.

9.12 Doubly Linked Lists in the Standard Libraries

9.12.1 C++: `std::list`

`std::list<T>` is the doubly linked list of the C++ standard library [7]. It supports constant-time insertion and removal anywhere, given an iterator, and “adding, removing and moving the elements within the list or across several lists does not invalidate the iterators or references” [8]. The GNU implementation is precisely the structure of Section 9.7: a circular doubly linked list whose only data member is a header node acting as the sentinel. Its source code explains that the list is “actually circular”, that `end()` is the header node itself, and that the list is empty “when this member iterator’s next/previous pointers refer to itself”; the header also stores the element

count, "so that `std::list::size()` is fast" [9]. **Listing 9.13** exercises the operations that a doubly linked list makes cheap.

Listing 9.13 `std_dll.cpp`: `std::list` walked in both directions, with $O(1)$ insertion and erasure at an iterator, stable iterators and `splice`.

```
// std_dll.cpp -- std::list, the C++ standard library's doubly linked list:
// both directions, O(1) insert/erase at an iterator, splice, stable iterators.
#include <iostream>
#include <iterator>
#include <list>
#include <string>

template <typename Seq>
void print(const char *label, const Seq &s) {
    std::cout << label;
    for (const auto &x : s) std::cout << ' ' << x;
    std::cout << '\n';
}

int main() {
    std::list<int> a = {10, 20, 30, 40};
    print("start:              ", a);
    a.push_front(5);                              // O(1) at both ends
    a.push_back(50);
    print("push_front/back:    ", a);

    auto it = std::next(a.begin(), 3);  // walk to the node holding 30: O(k)
    auto kept = it;  // a second iterator to the same node
    a.insert(it, 25);                             // O(1): new node before 30
    a.erase(std::prev(it, 2));  // O(1): remove the node holding 20
    print("insert 25, erase 20:", a);
    std::cout << "iterator to 30 still valid: " << *kept << '\n';

    std::cout << "backward:           ";
    // follows prev
    for (auto r = a.rbegin(); r != a.rend(); ++r) std::cout << ' ' << *r;
    std::cout << '\n';

    std::list<int> b = {7, 8, 9};
    // move the node holding 8: O(1)
    a.splice(a.begin(), b, std::next(b.begin()));
    print("after splice, a:    ", a);
    print("after splice, b:    ", b);
    a.splice(a.end(), b);  // move all of b to the back of a
    print("splice all of b:    ", a);
    std::cout << "b is now empty: " << (b.empty() ? "yes" : "no")
              << ", a.size() = " << a.size() << '\n';
```

Listing 9.13 (Continued) *std_dll.cpp: std::list walked in both directions, with O(1) insertion and erasure at an iterator, stable iterators and splice.*

```
    std::list<std::string> recent = {"Filosofi Kopi", "Bumi Manusia",
                                     "Laskar Pelangi"};
    // "Laskar Pelangi" is borrowed again
    auto hit = std::next(recent.begin(), 2);
    // move it to the front, no copying
    recent.splice(recent.begin(), recent, hit);
    std::cout << "move-to-front:      ";
    for (const auto &t : recent) std::cout << " [" << t << ']';
    std::cout << '\n';
    return 0;
}
```

```
start:                10 20 30 40
push_front/back:      5 10 20 30 40 50
insert 25, erase 20: 5 10 25 30 40 50
iterator to 30 still valid: 30
backward:             50 40 30 25 10 5
after splice, a:      8 5 10 25 30 40 50
after splice, b:      7 9
splice all of b:      8 5 10 25 30 40 50 7 9
b is now empty: yes, a.size() = 9
move-to-front:        [Laskar Pelangi] [Filosofi Kopi] [Bumi Manusia]
```

Three features deserve attention. First, `insert(it, 25)` inserts *before* the iterator's node, which a singly linked `std::forward_list` could only do with `insert_after` on the previous node (Section 7.12.1); `std::prev(it, 2)` walks backward along `prev` links. Second, the iterator `kept` still refers to 30 after an insertion and an erasure next to it, because no node moved. Third, `splice` moves nodes from one list into another, or to another position in the same list, by relinking, without copying or allocating: moving one node or a whole list is $O(1)$ (moving a range between two different lists is linear in the length of the range, because the element counts must be updated). The last line of the output is the move-to-front step of the LRU panel, done by the library in one call.

9.12.2 Java: `LinkedList` and `LinkedHashMap`

Java's `java.util.LinkedList<E>` is a "doubly-linked list implementation of the `List` and `Deque` interfaces" [4]. Its `ListIterator` can move in both directions and insert or remove at its position in $O(1)$, and `descendingIterator()` walks the list from the last element to the first. Java also ships the LRU pattern of Section 9.9 ready-made: `LinkedHashMap` "maintains a doubly-linked list running through all of its entries", and a special constructor orders that list by access instead of by insertion, which "is well-suited to building LRU caches"; overriding `removeEldestEntry` sets the capacity [10]. **Listing 9.14** uses both.

Listing 9.14 `LibraryDll.java`: `LinkedList` edited through a `ListIterator` and walked backward, and a `LinkedHashMap` in access order used as an LRU list of capacity 3.

```
// LibraryDll.java -- Java's doubly linked structures: LinkedList walked in both
// directions with a ListIterator, and LinkedHashMap in access order as an LRU
// list.
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.Map;

public class LibraryDll {
    public static void main(String[] args) {
        LinkedList<Integer> a = new LinkedList<>(List.of(10, 20, 30, 40));
        a.addFirst(5);                                  // O(1) at both ends
        a.addLast(50);
        System.out.println("addFirst/addLast:    " + a);

        // a cursor between two nodes
        ListIterator<Integer> it = a.listIterator();
        while (it.next() != 30) { }                     // walk forward to 30
        it.previous();                                  // step back over 30
        it.add(25);  // O(1): link a node before 30
        it.previous();                                  // back over 25 ...
        it.previous();                                  // ... and over 20
        it.remove();                                    // O(1): unlink 20
        System.out.println("add 25, remove 20:   " + a);

        StringBuilder sb = new StringBuilder("backward:           ");
        Iterator<Integer> d = a.descendingIterator();     // follows prev
        while (d.hasNext()) sb.append(' ').append(d.next());
        System.out.println(sb);
        // Java 21: no copy
        System.out.println("reversed() view:     " + a.reversed());

        // access order = true: every get() moves the entry to the most-recent
        // end
        LinkedHashMap<String, String> recent =
                new LinkedHashMap<>(16, 0.75f, true) {
            @Override
            protected boolean removeEldestEntry(
                    Map.Entry<String, String> eldest) {
                return size() > 3;  // keep only the three most recent
            }
        };
        recent.put("B1001", "Laskar Pelangi");
        recent.put("B1002", "Bumi Manusia");
```

Listing 9.14 (Continued) *LibraryDll.java: LinkedList edited through a ListIterator and walked backward, and a LinkedHashMap in access order used as an LRU list of capacity 3.*

```
        recent.put("B1003", "Filosofi Kopi");
        recent.get("B1001");  // borrowed again: now the newest
        recent.put("B1004", "Cantik Itu Luka");  // evicts the eldest, B1002
        System.out.println("LRU, oldest first:   " + recent.keySet());
    }
}
```

```
addFirst/addLast:    [5, 10, 20, 30, 40, 50]
add 25, remove 20:   [5, 10, 25, 30, 40, 50]
backward:            50 40 30 25 10 5
reversed() view:     [50, 40, 30, 25, 10, 5]
LRU, oldest first:   [B1003, B1001, B1004]
```

The list operations produce the same list as **Listing 9.13**. The `reversed()` view, added by the `SequencedCollection` interface of Java 21, walks the same nodes backward without copying

them. In the map, `get("B1001")` makes B1001 the most recently used entry, so adding B1004 evicts B1002, the eldest: the map prints its keys from the oldest to the newest.

9.12.3 Python: `collections.deque`, `OrderedDict` and `lru_cache`

Python has no general-purpose linked list class, for the reasons given in Section 7.12.3, but three parts of its standard library are doubly linked lists inside. `collections.deque` stores its elements, in the words of the CPython source, in "a doubly-linked list of fixed length blocks" of 64 elements each, which "assures that appends or pops never move any other data elements besides the one being appended or popped" [11]; it gives $O(1)$ appends and pops at both ends [12]. The pure-Python `OrderedDict` keeps its keys in "a doubly linked list", a "circular doubly linked list [that] starts and ends with a sentinel element" [13], and the C implementation that CPython uses by default offers the same operations; `move_to_end` and `popitem(last=False)` are exactly the move-to-front and evict-oldest steps of the LRU panel. And the `functools.lru_cache` decorator, which caches the results of a function, is implemented with a circular doubly linked list whose `root` is a sentinel: a cache hit unlinks its entry and relinks it before `root`, and an eviction takes the entry after `root` [14]. **Listing 9.15** uses all three.

Listing 9.15 `library_dll.py`: `collections.deque` (with `rotate` and `maxlen`), an `OrderedDict` used as an LRU list, and `functools.lru_cache`.

```
# library_dll.py -- Python's built-in doubly linked structures:
# collections.deque (a doubly linked list of 64-slot blocks) and OrderedDict /
# lru_cache as LRU lists.
from collections import OrderedDict, deque
from functools import lru_cache


def main():
    d = deque([10, 20, 30, 40])
    d.appendleft(5)                           # O(1) at both ends
    d.append(50)
    print("appendleft/append:  ", list(d))
    d.rotate(2)                               # the last two move to the front
    print("rotate(2):          ", list(d))
    d.rotate(-2)
    print("backward:           ", list(reversed(d)))

    recent = deque(maxlen=3)  # a bounded deque drops from the other end
    for book in ["B1001", "B1002", "B1003", "B1004"]:
        recent.appendleft(book)
    print("maxlen=3, newest first:", list(recent))

    lru = OrderedDict()  # a dict with a doubly linked order
    for k, v in [("B1001", "Laskar Pelangi"), ("B1002", "Bumi Manusia"),
                 ("B1003", "Filosofi Kopi")]:
        lru[k] = v
    lru.move_to_end("B1001")                  # borrowed again: now the newest
    lru["B1004"] = "Cantik Itu Luka"
    if len(lru) > 3:
        lru.popitem(last=False)               # evict the oldest, B1002
    print("LRU, oldest first:  ", list(lru))

    @lru_cache(maxsize=2)
    def title_of(book_id):
        titles = {"B1001": "Laskar Pelangi", "B1002": "Bumi Manusia"}
        return titles.get(book_id, "?")

    for book_id in ["B1001", "B1002", "B1001", "B1003", "B1002"]:
        title_of(book_id)
    print("lru_cache:          ", title_of.cache_info())


if __name__ == "__main__":
    main()
```

```
appendleft/append:   [5, 10, 20, 30, 40, 50]
rotate(2):           [40, 50, 5, 10, 20, 30]
backward:            [50, 40, 30, 20, 10, 5]
maxlen=3, newest first: ['B1004', 'B1003', 'B1002']
LRU, oldest first:   ['B1003', 'B1001', 'B1004']
lru_cache:           CacheInfo(hits=1, misses=4, maxsize=2, currsize=2)
```

`rotate(2)` moves the last two elements to the front, a rotation of the circle, the kind of operation a circular list does naturally. A deque with `maxlen=3` silently drops an element from the far end when a fourth is added, which makes it a ready-made "last *k* items" buffer. The `OrderedDict` ends with the same keys as Java's `LinkedHashMap`. For `lru_cache` with room for two results, the calls B1001, B1002, B1001, B1003, B1002 give one hit (the second B1001) and four misses: B1003 evicts B1002, the least recently used, so the last call, for B1002, misses again.

Table 9.5 summarizes the three libraries.

Table 9.5 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`

The same design appears far outside the standard libraries. The Linux kernel's general-purpose list, `struct list_head`, is a circular doubly linked list in which the list head plays the role of the sentinel; the kernel documentation notes that "to get from the head node to the tail, we can just travel one edge backwards", and also warns that, because of their poor data locality, linked lists are a bad choice where performance matters [15]. Knuth's *dancing links* technique uses the fact that a node unlinked from a doubly linked list still remembers its two neighbors, so that `x.prev.next ← x; x.next.prev ← x` puts it back exactly where it was; backtracking searches use this to undo deletions in $O(1)$ [16]. Exercise 9.12 asks you to verify it.

9.13 Profitina in Practice

Profitina is an AI visibility (GEO) platform that measures how ready a brand's website is to be cited by AI answer engines, and helps fix what it finds [17]. Its server is a Python application built on FastAPI, and several of its features need exactly the operations that this chapter made cheap. We describe the shape of each problem and the class of structure that fits it.

- **Browsing a history in both directions.** The historical analytics dashboard shows a brand's score week by week. A user who opens the newest week and steps back to the week before an alert, and then forward again to compare, is walking a doubly linked sequence with a cursor: each step is `cur = cur->prev` or `cur = cur->next`, $O(1)$ whatever the length of the history. Section 7.13 noted that the forward-only history of Chapter 7 could not support "one week back" without a walk from the start; the `prev` pointer is what makes both buttons equally cheap.
- **Keeping the most recent items.** The real-time community feed and the dashboard's notification list keep the most recent items and discard the oldest. In Python this is a `collections.deque` with `maxlen`, whose blocks are doubly linked ([11], Section 9.12.3).
- **Caching expensive results.** A scan re-reads pages, and the local AI engine replies to questions; both are expensive, and repeated requests are common. A bounded cache with the LRU policy of Section 9.9, such as `functools.lru_cache` or an `OrderedDict` used as in **Listing 9.15**, keeps

recently used results and evicts the least recently used one first. Inside, it is the sentinel list plus hash map of **Figure 9.17**.

The runnable case study is Profitina Laundry, a small laundry business that records how many orders it takes each week. **Listing 9.16** keeps the weekly totals in a doubly linked list with a cursor, opens on the newest week, and follows a sequence of Back and Forward clicks, reporting for each week the change from the week before, which it reads through `prev` in $O(1)$.

Listing 9.16 `laundry_weeks.cpp`: Profitina Laundry's weekly order totals in a doubly linked list, browsed one week back or forward from any week.

```
// laundry_weeks.cpp -- Profitina Laundry: browsing weekly order totals one week
// back or forward from any week, with a doubly linked list and a cursor.
#include <cstdio>
#include <string>

struct Week {
    std::string label;                  // e.g. "2026-W36"
    int orders;                         // orders taken that week
    Week *prev;
    Week *next;
};

struct History {
    Week *first = nullptr;
    Week *latest = nullptr;             // the newest week (the tail)
};

void addWeek(History &h, const std::string &label, int orders) {
    Week *w = new Week{label, orders, h.latest, nullptr};
    if (h.latest == nullptr) h.first = w;
    else h.latest->next = w;
    h.latest = w;  // O(1): weeks only ever arrive at the back
}

void report(const Week *w) {
    std::printf("  %s: %3d orders", w->label.c_str(), w->orders);
    if (w->prev != nullptr) {
        int d = w->orders - w->prev->orders;  // compare with the week before
        std::printf("  (%+d vs %s)", d, w->prev->label.c_str());
    }
    std::printf("\n");
}

void destroy(History &h) {
    while (h.first != nullptr) {
        Week *del = h.first;
        h.first = h.first->next;
        delete del;
    }
    h.latest = nullptr;
}
```

Listing 9.16 (Continued) *laundry_weeks.cpp: Profitina Laundry's weekly order totals in a doubly linked list, browsed one week back or forward from any week.*

```
int main() {
    History h;
    const char *labels[] = {"2026-W32", "2026-W33", "2026-W34", "2026-W35",
                            "2026-W36", "2026-W37", "2026-W38"};
    const int orders[] = {118, 124, 97, 131, 140, 126, 152};
    for (int i = 0; i < 7; ++i) addWeek(h, labels[i], orders[i]);

    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);
    }
    destroy(h);
    return 0;
}
```

```
open:
  2026-W38: 152 orders  (+26 vs 2026-W37)
back:
  2026-W37: 126 orders  (-14 vs 2026-W36)
back:
  2026-W36: 140 orders  (+9 vs 2026-W35)
back:
  2026-W35: 131 orders  (+34 vs 2026-W34)
forward:
  2026-W36: 140 orders  (+9 vs 2026-W35)
back:
  2026-W35: 131 orders  (+34 vs 2026-W34)
back:
  2026-W34:  97 orders  (-27 vs 2026-W33)
back:
  2026-W33: 124 orders  (+6 vs 2026-W32)
back:
  2026-W32: 118 orders
back: no more weeks, stays on 2026-W32
```

New weeks are only ever added at the back, so the tail pointer makes `addWeek` $O(1)$; every click is one pointer step; and the comparison with the previous week is one more. When the cursor reaches the oldest week, `prev` is null and the Back button simply does nothing. `LaundryWeeks.java` and `laundry_weeks.py` print the same report.

9.14 Common Mistakes, and How to Avoid Them

Updating one direction only. Every `a->next = b` needs its `b->prev = a`. A list that prints correctly forward may be broken backward (**Figure 9.19**, panel b). Print every test list both ways and check Equation (9.1) with a function such as `checkLinks` (Section 9.4.5).

Overwriting the link you navigate by. In `insertAfter(p)`, `p->next` is the only way to the successor; change it last. Setting `p->next = n` first makes the new node point at itself (**Figure 9.19**, panel a).

Forgetting to clear `prev` at the new front. After `frontDeletion`, the new first node's `prev` must become null (Section 9.3.5); after `backDeletion`, the new last node's `next` must become null. Otherwise the list ends in freed memory.

Forgetting the second external pointer. In a list with `head` and `tail`, inserting into an empty list must set both, and deleting the only node must reset both (Section 9.4.1).

Testing for null in a circular list. No pointer in a non-empty circular list is null, so `while (p != nullptr)` never ends. Use a `do`–`while` that stops on returning to the start, or, with a sentinel, `for (p = nil.next; p != &nil; p = p->next)`.

Deleting the sentinel. `LIST-DELETE(nil)` destroys the list's anchor. Pop operations must check for emptiness first (as `Deque::unlink` does in **Listing 9.9**).

Leaking the head node or the sentinel. A heap-allocated head node or sentinel must be released after the real nodes (Sections 9.5.2 and 9.11.2).

Copying a list object that contains its sentinel. A copied sentinel's neighbors still point at the original sentinel. Forbid copying or implement it node by node.

Believing that $O(1)$ deletion means $O(1)$ deletion by value. `deleteNode(p)` is $O(1)$; finding `p` by value is still a $\Theta(n)$ search. The doubly linked list pays off when the program already holds the node, as in the LRU panel of Section 9.9.

Using a doubly linked list where an array would do. For small elements and mostly sequential reading, the array's contiguous memory wins by a wide margin (Section 7.12.4), and a doubly linked node spends 16 bytes on pointers (Equation (9.2)).

9.15 Chapter Summary

- A **doubly linked list** gives every node two pointers, `next` and `prev`. The **two-way invariant** (Equation (9.1)) says that every forward link has a matching backward link; every operation must preserve it.
- A doubly linked node costs one pointer more than a singly linked one: 24 bytes instead of 16 for an `int` on a 64-bit machine (**Figure 9.2**, Equation (9.2)).
- **With only `head`**, front insertion and deletion are $O(1)$, back insertion and deletion are $\Theta(n)$, and a backward traversal takes two walks and no extra memory (**Table 9.1**).
- **With `head` and `tail`**, every end operation is $O(1)$, including back deletion, because the node before the last is `tail->prev` (**Pseudocode 9.6**). Inserting before or after, and deleting, a node the program holds are all $O(1)$ (**Pseudocode 9.7**, **Pseudocode 9.9**, **Table 9.2**). An insertion changes four pointers, a deletion two.
- A **head node** removes the special cases at the front; a **circular** list reaches both ends through one pointer (`head->prev` is the last node) and needs a new stop signal for traversals.
- A **sentinel** in a circular doubly linked list removes every special case: no pointer is ever null and no external pointer ever changes, so insertion and deletion are written once, without tests (**Pseudocode 9.10**, **Pseudocode 9.11**, **Table 9.3**). Storing the key in the sentinel halves the tests of a sequential search (Equation (9.4)).

- A **deque** on a sentinel list does all its operations in $O(1)$; an **LRU list**, a sentinel list plus a hash map from key to node, records each access in $O(1)$ expected time (Equation (9.5)).
- Against the singly linked list, the doubly linked list turns back deletion and deletion at a held position from $\Theta(n)$ into $O(1)$; measured, emptying 20,000 nodes from the back became about 3,500 times faster (**Figure 9.18**). Against an array, it loses random access and cache locality (**Table 9.4**).
- Memory checkers find the uses of freed memory that half-updated links cause, but not a broken invariant that stays inside live memory; print both ways and check the invariant.
- **Libraries**: C++ `std::list` is a circular doubly linked list with a sentinel; Java has `LinkedList` and the LRU-ready `LinkedHashMap`; Python's `collections.deque`, `OrderedDict` and `functools.lru_cache` are doubly linked inside (**Table 9.5**).

9.16 Exercises

9.1 Starting from an empty doubly linked list with head and tail, perform `frontInsertion(5)`, `backInsertion(8)`, `frontInsertion(2)`, `insertAfter(p, 6)` where `p` is the node holding 5, `backDeletion()`, `deleteNode(p)` for the same `p`, and `frontInsertion(9)`. Give the list after every operation, read forward and backward, and say which operations changed `head` or `tail`.

9.2 For a doubly linked list of n nodes, give the running time of: (a) back deletion with only `head`; (b) back deletion with `head` and `tail`; (c) deleting a node that the program holds a pointer to, compared with a singly linked list with head and tail; (d) printing the list backward with only `head` (give the exact number of pointer steps); (e) reaching the node at index $k = n/2$ when both `head` and `tail` are available (use Equation (9.6)). Justify each answer in one sentence.

9.3 Write `void reverse(DList &L)`, which reverses a doubly linked list with head and tail in place, without creating or deleting any node, in one pass. How does it differ from the iterative reversal of a singly linked list in Section 7.6.6?

9.4 Write `bool isPalindrome(const DList &L)`, which decides whether the list reads the same in both directions, using $O(1)$ extra memory. Make sure it works for lists of odd and even length and for the empty list.

9.5 Write `int deleteAll(DList &L, int key)`, which deletes every node holding `key` and returns how many were deleted. What must the loop save before it deletes a node?

9.6 Write `void concat(DList &a, DList &b)`, which appends all nodes of `b` to the end of `a` in $O(1)$ time and leaves `b` empty. What is the cost of the same operation for two singly linked lists with head and tail, and for two arrays?

9.7 Write `Node *kthFromEnd(const DList &L, int k)`, which returns the k-th node from the back ($k = 1$ is the last node) or `nullptr` if the list has fewer than k nodes, in $O(k)$ time.

9.8 Write `void moveToFront(DList &L, Node *p)`, which moves the node `p` of a list with head and tail to the front in $O(1)$ time. Handle the cases in which `p` is already the first node and in which it is the last node. This is the step that the LRU panel of Section 9.9 repeats on every hit.

9.9 In a circular doubly linked list with a sentinel, write `void rotateLeft(Node *nil, int k)` for $0 \le k < n$, which makes the $(k+1)$-th node the first node (so that 1, 2, 3, 4, 5, 6 rotated left by 2 becomes 3, 4, 5, 6, 1, 2), by moving the sentinel itself. How many pointer assignments does it make, and what is its running time?

9.10 In **Pseudocode 9.7**, which of the four steps may be exchanged without breaking the list, and which may not? For each forbidden exchange, describe the broken list that results.

9.11 The procedure below is proposed as `frontInsertion` for the list of Section 9.3, without the test of **Pseudocode 9.1**:

```
void frontInsertion(TNode *&head, int x) {
    TNode *n = new TNode{x, nullptr, head};
    head->prev = n;
    head = n;
}
```

Find the input for which it fails, say what happens, and explain why the corresponding line in **Pseudocode 9.10** can never fail in the same way.

9.12 (*Dancing links.*) In a circular doubly linked list with a sentinel, node `x` is removed by the two assignments of **Pseudocode 9.11** but *not* freed, so that `x.prev` and `x.next` still hold its old neighbors. Show that the two assignments `x.prev.next ← x; x.next.prev ← x` put it back exactly where it was. Then show that if nodes `x` and then `y` are removed this way, they must be restored in the opposite order, `y` first, and give a list of three nodes on which restoring in the same order goes wrong.

9.13 An *XOR linked list* stores in each node a single field $link = prev \oplus next$, the bitwise exclusive-or of the two addresses, instead of two pointers. (a) Explain how a traversal that knows the address of the previous node can compute the address of the next one. (b) How much memory does it save for n nodes on a 64-bit machine, compared with Equation (9.2)? (c) Which operation of Section 9.4 becomes impossible in $O(1)$, and why? (d) Why can this structure not be written in Java or Python?

9.14 For each of the following Pustaka Ceria tasks, choose between an array, a singly linked list and a doubly linked list (possibly with a hash map), and justify the choice with the rows of **Table 9.4**: (a) the reading-room waiting line, where readers join at the back and are served at the front; (b) the undo and redo history of a librarian's catalog edits, where Undo steps back and Redo steps forward from the current edit; (c) the "recently borrowed" panel of Section 9.9 for $k = 1{,}000$; (d) the list of all 50,000 book records, sorted by call number and looked up by binary search.

9.17 Solutions to the Exercises

9.1 After each operation (forward | backward): `frontInsertion(5)`: 5 | 5, and `head` and `tail` both change, because the list was empty. `backInsertion(8)`: 5, 8 | 8, 5; `tail` changes. `frontInsertion(2)`: 2, 5, 8 | 8, 5, 2; `head` changes. `insertAfter(p, 6)`: 2, 5, 6, 8 | 8, 6, 5, 2; neither changes, because `p` was not the last node. `backDeletion()`: 2, 5, 6 | 6, 5, 2; `tail` moves back to 6. `deleteNode(p)`: 2, 6 | 6, 2; neither changes, because 5 was a middle node. `frontInsertion(9)`: 9, 2, 6 | 6, 2, 9; `head` changes.

9.2 (a) $\Theta(n)$: the last node can only be reached by walking from `head`, although the predecessor is then one step away (Section 9.3.6). (b) $O(1)$: the new last node is `tail->prev` (**Pseudocode 9.6**). (c) $O(1)$ for the doubly linked list (**Pseudocode 9.9**); $\Theta(n)$ for the singly linked list, which must search from `head` for the predecessor. (d) $n-1$ steps forward to the last node and $n-1$ steps back along `prev` (the loop visits n nodes and makes n assignments, the last of which produces NIL), about $2n$ in total: $\Theta(n)$. (e) By Equation (9.6), $\min(n/2,\ n/2-1) = n/2-1$ steps from the back, for even n: the middle is the worst position, and it still costs $\Theta(n)$.

9.3 The solution program **Listing 9.17** contains `reverse`. Reversing a doubly linked list does not change which nodes are neighbors, only which direction is called “forward”, so it is enough to swap `prev` and `next` in every node and then swap `head` and `tail`. The loop must advance to the old `next`, which after the swap is stored in `p->prev`; that is why it steps with `p = p->prev`. There is one pass and no allocation: $\Theta(n)$ time and $O(1)$ memory. The singly linked reversal of Section 7.6.6 had to keep three pointers (`prev`, `cur`, `nxt`) because each node’s only link had to be turned around while the rest of the list was still reachable only through it; here every node already holds both directions.

Listing 9.17 `exercises.cpp`: solutions to Exercises 9.3 to 9.9.

```
// exercises.cpp -- solutions to the programming exercises of Chapter 9.
#include <cstdio>

struct Node {
    int data;
    Node *prev;
    Node *next;
};

struct DList {
    Node *head = nullptr;
    Node *tail = nullptr;
};

void backInsertion(DList &L, int x) {
    Node *n = new Node{x, L.tail, nullptr};
    if (L.head == nullptr) L.head = n;
    else L.tail->next = n;
    L.tail = n;
}

DList fromArray(const int *a, int n) {
    DList L;
    for (int i = 0; i < n; ++i) backInsertion(L, a[i]);
    return L;
}

void show(const char *label, const DList &L) {
    std::printf("%-22s", label);
    for (Node *p = L.head; p != nullptr; p = p->next)
        std::printf("%d ", p->data);
    std::printf("| back: ");
    for (Node *p = L.tail; p != nullptr; p = p->prev)
        std::printf("%d ", p->data);
    std::printf("\n");
}

void clear(DList &L) {
    while (L.head != nullptr) {
        Node *del = L.head;
        L.head = L.head->next;
        delete del;
    }
    L.tail = nullptr;
}
```

Listing 9.17 (Continued) *exercises.cpp: solutions to Exercises 9.3 to 9.9.*

```
// Exercise 9.3: reverse in place by swapping every node's two links.
void reverse(DList &L) {
    // p->prev is the OLD next
    for (Node *p = L.head; p != nullptr; p = p->prev) {
        Node *t = p->next;
        p->next = p->prev;
        p->prev = t;
    }
    Node *t = L.head;
    L.head = L.tail;
    L.tail = t;
}

// Exercise 9.4: does the list read the same in both directions?
bool isPalindrome(const DList &L) {
    Node *a = L.head, *b = L.tail;
    // stop when they meet or cross
    while (a != b && a != nullptr && a->prev != b) {
        if (a->data != b->data) return false;
        a = a->next;
        b = b->prev;
    }
    return true;
}

// Exercise 9.5: delete every node holding key; return how many were deleted.
int deleteAll(DList &L, int key) {
    int count = 0;
    Node *p = L.head;
    while (p != nullptr) {
        Node *nxt = p->next;  // save the way on before p may disappear
        if (p->data == key) {
            if (p->prev) p->prev->next = p->next; else L.head = p->next;
            if (p->next) p->next->prev = p->prev; else L.tail = p->prev;
            delete p;
            ++count;
        }
        p = nxt;
    }
    return count;
}
```

Listing 9.17 (Continued) *exercises.cpp: solutions to Exercises 9.3 to 9.9.*

```
// Exercise 9.6: append all of b to a in O(1); b becomes empty.
void concat(DList &a, DList &b) {
    if (b.head == nullptr) return;
    if (a.head == nullptr) {
        a = b;
    } else {
        a.tail->next = b.head;
        b.head->prev = a.tail;
        a.tail = b.tail;
    }
    b.head = b.tail = nullptr;
}

// Exercise 9.7: the k-th node from the back (k = 1 is the last), or nullptr.
Node *kthFromEnd(const DList &L, int k) {
    Node *p = L.tail;
    while (p != nullptr && k > 1) {
        p = p->prev;
        --k;
    }
    return (k >= 1) ? p : nullptr;
}

// Exercise 9.8: move node p to the front in O(1) (the step an LRU list
// repeats).
void moveToFront(DList &L, Node *p) {
    if (p == L.head) return;
    p->prev->next = p->next;              // unlink p (it has a predecessor)
    if (p->next) p->next->prev = p->prev; else L.tail = p->prev;
    p->prev = nullptr;                    // relink p before the old first node
    p->next = L.head;
    L.head->prev = p;
    L.head = p;
}

// Exercise 9.9: rotate a circular list with a sentinel left by k (0 <= k < n)
// by moving the sentinel itself: O(k) steps, 6 pointer assignments.
void rotateLeft(Node *nil, int k) {
    if (k == 0 || nil->next == nil) return;
    Node *q = nil;
    for (int i = 0; i < k; ++i) q = q->next;    // q = the k-th real node
    nil->prev->next = nil->next;          // take the sentinel out of the ring
    nil->next->prev = nil->prev;
    nil->prev = q;                        // and put it back right after q
    nil->next = q->next;
    q->next->prev = nil;
```

Listing 9.17 (Continued) *exercises.cpp: solutions to Exercises 9.3 to 9.9.*

```
    q->next = nil;
}

int main() {
    int a[] = {1, 2, 3, 4, 5};
    DList L = fromArray(a, 5);
    reverse(L);
    show("reverse(1..5):", L);

    int p1[] = {3, 8, 1, 8, 3}, p2[] = {3, 8, 8, 3}, p3[] = {3, 8, 1};
    DList P1 = fromArray(p1, 5), P2 = fromArray(p2, 4);
    DList P3 = fromArray(p3, 3), E;
    std::printf("isPalindrome: 38183 %s, 3883 %s, 381 %s, empty %s\n",
                isPalindrome(P1) ? "yes" : "no",
                isPalindrome(P2) ? "yes" : "no",
                isPalindrome(P3) ? "yes" : "no",
                isPalindrome(E) ? "yes" : "no");
    clear(P1); clear(P2); clear(P3);

    int d[] = {7, 2, 7, 7, 5, 7};
    DList D = fromArray(d, 6);
    int removed = deleteAll(D, 7);
    std::printf("deleteAll(7) removed %d\n", removed);
    show("after deleteAll:", D);

    int c[] = {6, 7, 8};
    DList B = fromArray(c, 3);
    concat(L, B);
    show("concat:", L);
    std::printf("b is empty: %s\n", B.head == nullptr ? "yes" : "no");
    Node *k2 = kthFromEnd(L, 2);
    std::printf("kthFromEnd(2) = %d, kthFromEnd(9) is %s\n", k2->data,
                kthFromEnd(L, 9) ? "a node" : "nullptr");
    moveToFront(L, kthFromEnd(L, 4));     // the node holding 1
    show("moveToFront(1):", L);
    moveToFront(L, L.tail);               // the node holding 8
    show("moveToFront(8):", L);
    clear(L);
    clear(D);
```

Listing 9.17 (Continued) *exercises.cpp: solutions to Exercises 9.3 to 9.9.*

```
    // a circular list with a sentinel on the stack
    Node nil{0, nullptr, nullptr};
    nil.prev = nil.next = &nil;
    for (int x = 1; x <= 6; ++x) {
        Node *n = new Node{x, nil.prev, &nil};
        nil.prev->next = n;
        nil.prev = n;
    }
    rotateLeft(&nil, 2);
    std::printf("%-22s", "rotateLeft(2):");
    for (Node *p = nil.next; p != &nil; p = p->next)
        std::printf("%d ", p->data);
    std::printf("| back: ");
    for (Node *p = nil.prev; p != &nil; p = p->prev)
        std::printf("%d ", p->data);
    std::printf("\n");
    while (nil.next != &nil) {
        Node *del = nil.next;
        nil.next = del->next;
        delete del;
    }
    return 0;
}
```

```
reverse(1..5):         5 4 3 2 1 | back: 1 2 3 4 5
isPalindrome: 38183 yes, 3883 yes, 381 no, empty yes
deleteAll(7) removed 4
after deleteAll:       2 5 | back: 5 2
concat:                5 4 3 2 1 6 7 8 | back: 8 7 6 1 2 3 4 5
b is empty: yes
kthFromEnd(2) = 7, kthFromEnd(9) is nullptr
moveToFront(1):        1 5 4 3 2 6 7 8 | back: 8 7 6 2 3 4 5 1
moveToFront(8):        8 1 5 4 3 2 6 7 | back: 7 6 2 3 4 5 1 8
rotateLeft(2):         3 4 5 6 1 2 | back: 2 1 6 5 4 3
```

The first output line shows 1, 2, 3, 4, 5 reversed, read both ways.

9.4 `isPalindrome` moves `a` forward from `head` and `b` backward from `tail`, comparing their data, until they meet (odd length: `a == b` on the middle node) or cross (even length: `a->prev == b` after the last pair). The empty list gives `a == b == nullptr` at once and is a palindrome. It makes at most $\lfloor n/2 \rfloor$ comparisons and uses two pointers. In the output, 3, 8, 1, 8, 3 and 3, 8, 8, 3 are palindromes and 3, 8, 1 is not.

9.5 The loop must save `p->next` in `nxt` *before* deleting `p`, because after `delete p` the field can no longer be read. Each deletion is the unlink of **Pseudocode 9.9**, with its two tests for the ends. On 7, 2, 7, 7, 5, 7 the function deletes four nodes, including the first and the last, and leaves 2, 5; the backward reading 5, 2 confirms the links. $\Theta(n)$.

9.6 Link `a`'s last node and `b`'s first node in both directions, move `a.tail` to `b.tail`, and empty `b`; when `a` is empty, `a` simply takes over `b`'s two pointers. Four assignments: $O(1)$. For two singly linked lists with head and tail it is also $O(1)$ (`a.tail->next = b.head; a.tail = b.tail;`), because only forward links exist. For two arrays it is $\Theta(m + n)$ in general, or $\Theta(n)$ amortized when the second array is copied onto the end of a dynamic array with spare capacity. In the output, 6, 7, 8 are appended to the reversed list.

9.7 Start at `tail` and step back $k - 1$ times; stop early with `nullptr` if the list runs out. $O(k)$ steps, whatever n is. With `tail` in a singly linked list, the k-th node from the end costs $\Theta(n)$, because one can only walk from the front. In the output, the second node from the end of 5, 4, 3, 2, 1, 6, 7, 8 is 7, and a list of 8 nodes has no 9th node from the end.

9.8 If `p` is already first, nothing changes. Otherwise unlink `p` (it has a predecessor, so `p->prev->next = p->next` needs no test; if it was the last node, `tail` moves back to `p->prev`), then link it before the old first node: `p->prev = nullptr`, `p->next = head`, `head->prev = p`, `head = p`. Constant work. The output moves 1 (a middle node) and then 8 (the last node) to the front; both readings stay consistent.

9.9 Walk `q` from `nil` k steps, so that `q` is the k-th real node; take the sentinel out of the ring with two assignments; put it back after `q` with four. That is six pointer assignments, as the comment in the listing says, and $O(k)$ time for the walk. No data node moves and no data node's links change except those of the sentinel's old and new neighbors. Rotating 1, 2, 3, 4, 5, 6 left by 2 gives 3, 4, 5, 6, 1, 2, and the backward reading 2, 1, 6, 5, 4, 3 confirms the links.

9.10 Steps 1 and 2 (the new node's own fields) may be exchanged with each other, and may be done in either order relative to step 3, because none of them overwrites a field that another step reads: step 3 reads `p.next`, which only step 4 writes. Steps 3 and 4 may not be exchanged: if step 4 (`p.next ← n`) came before step 3, then step 3 would execute `n.prev ← n`, making the new node its own predecessor, and the old successor would keep pointing back at `p`, so the backward chain would skip the new node. Step 4 may not come before step 2 either: step 2 would then set `n.next ← n`, and the forward chain would loop forever at the new node, with everything after it unreachable (**Figure 9.19**, panel a). In short, step 4 must be last.

9.11 It fails on the empty list: `head` is null, so `head->prev = n` dereferences a null pointer, which is undefined behavior and usually a crash. The test in **Pseudocode 9.1** exists exactly for this case. In **Pseudocode 9.10** the corresponding line is `p.next.prev ← n`, and `p.next` is never NIL, because in a circular list with a sentinel every pointer holds the address of a node; in the empty list `p.next` is the sentinel itself, and the assignment correctly sets `nil.prev` to the new node.

9.12 After `x.prev.next ← x.next; x.next.prev ← x.prev`, the neighbors skip `x`, but `x.prev` and `x.next` still name them, so `x.prev.next ← x; x.next.prev ← x` makes the two neighbors point at `x` again, which is the state before removal. The restoration is correct only if the neighbors are still neighbors of each other. Take the list a, x, y, b. Removing x links $a \leftrightarrow y$; removing y then links $a \leftrightarrow b$, and y remembers a and b. Restoring y first links $a \leftrightarrow y \leftrightarrow b$, and then restoring x (which remembers a and y) gives a, x, y, b: correct. Restoring x first instead sets $a.next \leftarrow x$ and $y.prev \leftarrow x$ while y is not in the list, so $a.next$ is x but $x.next$ is y, a removed node, and $b.prev$ is still a: the forward chain is a, x, y, b and the backward chain is b, a. The invariant is broken. Knuth's backtracking algorithms rely on exactly this last-in, first-out discipline, which a recursive search provides naturally [16].

9.13 (a) Walking forward, a traversal holds the addresses of the previous node q and the current node p; since $p.link = q \oplus next$ and $q \oplus q = 0$, the next address is $p.link \oplus q$. The same formula walks backward if q is the following node. At the ends the missing neighbor is taken as address 0. (b) One pointer per node, $8n$ bytes: the memory of a singly linked list. (c) Deleting (or inserting next to) a node given *only* a pointer to it: without the address of one neighbor, the other cannot be computed, so the program must walk to the node from an end. (d) Java and Python do not let a program see or combine addresses: a reference cannot be converted to a number and back, and the garbage collector must be able to recognize every reference, which it could not do inside an exclusive-or.

9.14 (a) A queue: a singly linked list with head and tail (or a circular array) suffices, because only front deletion and back insertion are needed, and both are $O(1)$; the doubly linked list's second pointer would be unused. (b) A doubly linked list with a cursor: Undo is `cur = cur->prev` and

Redo is `cur = cur->next`, both $O(1)$; a new edit after some undos deletes everything after the cursor and appends. (Two stacks are an equally good alternative.) (c) A doubly linked list with a sentinel plus a hash map, as in Section 9.9: every borrowing costs $O(1)$ expected, while an array or a list without the map would search 1,000 entries per borrowing. (d) An array: binary search needs $O(1)$ access by index, which no linked list offers; the list is sorted once and read many times, so the array's $\Theta(n)$ insertions rarely matter.

References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 2.2.5 (doubly linked lists).

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, sec. 10.2 (linked lists and sentinels).

[3] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, sec. 6.1 (sequential searching).

[4] Oracle, "Class LinkedList," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/LinkedList.html. [Accessed: 24-Sep-2026].

[5] Valgrind Developers, "Valgrind: Current releases" (Valgrind 3.27.1, 20 May 2026). [Online]. Available: https://valgrind.org/downloads/current.html. [Accessed: 24-Sep-2026].

[6] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. 2012 USENIX Annual Technical Conference*, Boston, MA, USA, Jun. 2012, pp. 309–318.

[7] ISO/IEC 14882:2017, *Programming Languages — C++*, 5th ed. Geneva, Switzerland: ISO, 2017, sec. 26.3.10 (class template `list`).

[8] cppreference.com, "std::list." [Online]. Available: https://en.cppreference.com/w/cpp/container/list. [Accessed: 24-Sep-2026].

[9] GNU Project, "libstdc++-v3/include/bits/stl_list.h," GCC source repository. [Online]. Available: https://gcc.gnu.org/git/?p=gcc.git;a=blob;f=libstdc%2B%2B-v3/include/bits/stl_list.h. [Accessed: 24-Sep-2026].

[10] Oracle, "Class LinkedHashMap<K,V>," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/LinkedHashMap.html. [Accessed: 24-Sep-2026].

[11] Python Software Foundation, "Modules/_collectionsmodule.c," CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Modules/_collectionsmodule.c. [Accessed: 24-Sep-2026].

[12] Python Software Foundation, "collections — Container datatypes," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/collections.html. [Accessed: 24-Sep-2026].

[13] Python Software Foundation, "Lib/collections/**init**.py," CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Lib/collections/**init**.py. [Accessed: 24-Sep-2026].

[14] Python Software Foundation, "Lib/functools.py," CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Lib/functools.py. [Accessed: 24-Sep-2026].

[15] The Linux Kernel Documentation, "Linked Lists in Linux." [Online]. Available: https://docs.kernel.org/core-api/list.html. [Accessed: 24-Sep-2026].

[16] D. E. Knuth, "Dancing links," in *Millennial Perspectives in Computer Science*, J. Davies, B. Roscoe, and J. Woodcock, Eds. Basingstoke, U.K.: Palgrave, 2000, pp. 187–214. [Online]. Available: https://arxiv.org/abs/cs/0011047

[17] Profitina, “Profitina — AI Visibility (GEO) platform.” [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Chapter 10
Recursion

By the end of this chapter you will be able to say precisely what a recursive definition and a recursive function are, and recognize the two parts every correct one has: a base case that answers at once and a recursive case that hands a strictly smaller problem to the same function; prove that a recursive function is correct by mathematical induction, and find its cost by writing and solving a recurrence; draw the call stack of a running recursion frame by frame, and read a recursion tree; write factorial both with a loop and with recursion, and explain what each costs; compute the Fibonacci numbers of the famous rabbit story in six ways (naive recursion, memoized recursion, a bottom-up table, two variables, tail recursion and fast doubling) and with Binet's closed form, and explain from measured call counts and timings why the naive way is hopeless beyond a few dozen; compute a greatest common divisor with Euclid's algorithm and prove that it needs only logarithmically many steps; raise a number to a huge power with a handful of multiplications; write the recursive solver for the Towers of Hanoi promised in Chapter 5 and prove that its $2^n - 1$ moves are the fewest possible; recurse over arrays and linked lists; state what a tail call is and what GCC, the Java virtual machine and CPython actually do with one, shown in real assembly and real crashes; know how deep recursion may go in C++, Java and Python and how to raise each limit; and turn any recursion into a loop with a stack of your own. Every program is given in C++, Java and Python, compiled, run and, where it builds linked structures, memory-checked.

10.1 Where We Left Off

Recursion has been waiting in the wings for most of this book. Section 2.5.5 wrote binary search as a function that calls itself on half of the array and argued, informally, that it is correct because it is correct on every shorter interval. Chapter 3 used recursion without ceremony in merge sort and quicksort (Sections 3.8 and 3.9), and promised that the recursion trees drawn there would be drawn and solved in general later. Chapter 5 told the story of the Towers of Hanoi (Section 5.2), found that n disks need $2^n - 1$ moves, solved the puzzle with three stacks and a loop, and left the recursive solver for this chapter. Section 5.10 then showed the most important stack in any computer, the *call stack*, and ended with a small function `countdown` whose three frames sat on the stack at the same time. Chapter 7 computed the length of a linked list and reversed it recursively (Sections 7.6.2 and 7.6.7), and warned that such functions need one stack frame per node. Chapter 9 closed with `functools.lru_cache`, a cache of function results kept in a circular doubly linked list, whose most famous use we meet in Section 10.5.

This chapter gathers those threads and makes recursion a tool you can trust. We start with a story and a definition (Section 10.2), then watch the machine carry out a recursive call, frame by frame (Section 10.3). Five classic problems follow, each teaching one more idea: factorial (the simplest recursion, Section 10.4), Fibonacci (how recursion can explode, and four ways to tame it, Section 10.5), Euclid's greatest common divisor (a recursion that shrinks very fast, Section 10.6), fast exponentiation (halving instead of subtracting, Section 10.7) and the Towers of Hanoi (a problem whose recursive solution is optimal and still exponential, Section 10.8). Section 10.9 applies recursion to arrays and linked lists, Section 10.10 draws recursion trees and solves the

recurrences they picture, and Sections 10.11 to 10.13 face the practical questions: what compilers do with tail calls, how deep recursion may go in each language, and how to replace recursion by a loop and a stack when it goes too deep. Chapter 11, on trees, will use everything here on almost every page, because a tree is a recursive structure: a root with smaller trees below it.

10.2 Thinking Recursively

10.2.1 A question passed along a queue

Imagine the queue at the lending desk of the Pustaka Ceria library on a busy Saturday morning. You are somewhere in the line, the line bends around a bookshelf, and you cannot see the desk. You would like to know how many people are ahead of you. You could step out of the line and count, but then you might lose your place. There is a lazier way. You tap the shoulder of the person in front of you and ask: "How many people are ahead of *you*?" Whatever number they tell you, you add one (for them), and you have your answer.

The person in front does not know either, of course. But they can do exactly what you did: ask the person in front of them, and add one to the answer. The question travels forward along the line until it reaches the person standing at the desk. That person does not need to ask anyone: nobody is ahead of them, so they answer "zero" at once. Then the answers travel back: the second person hears 0 and says 1, the third hears 1 and says 2, and so on, until the answer arrives back at you. **Figure 10.1** shows the whole exchange for a line of five people.

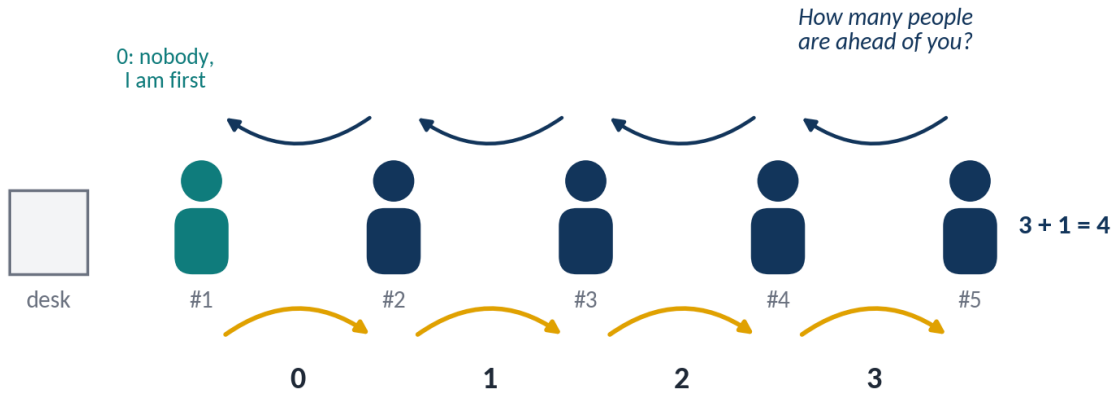


Figure 10.1 Counting the people ahead by asking: each question goes one place forward (navy), the first person answers 0 without asking anyone, and each answer comes back increased by one (amber).

Three features of this little procedure are the whole of recursion.

- **Everybody runs the same procedure.** No one needs a different set of instructions. "If nobody is ahead of me, answer 0; otherwise ask the person ahead and answer one more than they say."
- **Each question is smaller.** The person you ask has fewer people ahead of them than you do. The question shrinks by one person each time it is passed.
- **Someone answers without asking.** The person at the desk answers directly. Without that person, the question would travel forever (or, in a line that loops around a pillar, in circles).

Notice also what happens *while* the question travels: every person who has asked is standing there waiting, remembering that they must add one to whatever comes back. Five people are waiting at the deepest moment. In a program, those waiting people are *stack frames*, and the place where they wait is the call stack of Section 5.10. Section 10.3 watches them.

10.2.2 Recursive definitions

Mathematicians used recursion long before computers. A *recursive definition* defines a thing partly in terms of a smaller instance of the same thing. Here is the simplest kind, a sequence of numbers:

$$f(0) = 3, \qquad f(n) = 2f(n-1) + 3 \quad \text{for } n \geq 1. \tag{10.1}$$

Equation (10.1) does not tell us $f(4)$ directly. It tells us how to get $f(4)$ from $f(3)$, $f(3)$ from $f(2)$, and so on down to $f(0)$, which it states outright. Unfolding it:

$$f(0) = 3$$

$$f(1) = 2f(0) + 3 = 2 \cdot 3 + 3 = 9$$

$$f(2) = 2f(1) + 3 = 2 \cdot 9 + 3 = 21$$

$$f(3) = 2f(2) + 3 = 2 \cdot 21 + 3 = 45$$

$$f(4) = 2f(3) + 3 = 2 \cdot 45 + 3 = 93$$

Every recursive definition has the same two parts, and they have names that the rest of this chapter uses constantly.

> **DEFINITION.** A *recursive definition* defines an object in terms of smaller instances of itself. Its *base case* (also called the *anchor* or *stopping rule*) gives the answer for the smallest instances directly, without referring to the object. Its *recursive case* (also called the *inductive step*) 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, directly or through other functions; its base case returns without such a call.

In Equation (10.1) the base case is $f(0) = 3$ and the recursive case is $f(n) = 2f(n-1) + 3$. The word "strictly" matters. A definition such as $g(n) = g(n) + 1$ refers to an instance that is not smaller at all, and it defines nothing. A definition such as $h(n) = h(n+1) - 1$ refers to an instance that is *larger*, and the unfolding never reaches a base case.

10.2.3 Proving a recursive definition right: induction

The values 3, 9, 21, 45, 93 look as if they are always three less than a multiple of six: $3 = 6 - 3$, $9 = 12 - 3$, $21 = 24 - 3$, $45 = 48 - 3$, $93 = 96 - 3$. The multiples are $6 \cdot 1{,}6 \cdot 2{,}6 \cdot 4{,}6 \cdot 8{,}6 \cdot 16$, which suggests

$$f(n) = 6 \cdot 2^n - 3 \quad \text{for every } n \geq 0. \tag{10.2}$$

A formula such as Equation (10.2), which computes $f(n)$ without computing any earlier value, is called a *closed form*. Five matching values do not prove it; the sixth might fail. The tool that proves statements about recursive definitions is *mathematical induction*, and it has exactly the same two parts as the definition itself.

- **Base case.** Check the statement for the smallest n. For $n = 0$: $6 \cdot 2^0 - 3 = 6 - 3 = 3 = f(0)$. True.

- **Inductive step.** Assume the statement holds for $n-1$ (this assumption is the *induction hypothesis*), and show that it then holds for n. Suppose $f(n-1) = 6 \cdot 2^{n-1} - 3$. Then, by the recursive case of Equation (10.1),

$$f(n) = 2f(n-1) + 3 = 2(6 \cdot 2^{n-1} - 3) + 3 = 6 \cdot 2^n - 6 + 3 = 6 \cdot 2^n - 3.$$

Why does this prove the formula for *every* n? The base case gives $n = 0$. The step, applied with $n = 1$, turns the truth for 0 into the truth for 1; applied with $n = 2$, turns the truth for 1 into the truth for 2; and so on. Every n is reached after finitely many steps, exactly as the unfolding of the definition reaches $f(0)$ after finitely many steps. Induction and recursion are the same idea seen from two sides: recursion *computes* by reducing to smaller cases, and induction *proves* by building up from smaller cases.

> **KEY INSIGHT:** To convince yourself that a recursive function is correct, do not trace every call. Check the base case, then *assume* the recursive calls return correct answers for their smaller inputs, and check that the function combines those answers correctly. That is an induction proof, and it is the only way to reason about recursions a thousand calls deep. Programmers call this attitude "trusting the recursion", sometimes the *recursive leap of faith*; the induction proof is what makes the faith justified.

10.2.4 From definition to program

A recursive definition turns into a recursive function almost word for word. The general shape is

```
return_type function_name(parameters) {
    if (the problem is small enough)        // base case
        return the answer directly;
    ... function_name(smaller parameters) ...   // recursive case
    return an answer built from the smaller answers;
}
```

Listing 10.1 writes Equation (10.1) this way in C++, twice: once plainly, as `f`, and once as `fTraced`, which does the same work but prints a line when it is called and another when it returns, indented by four spaces per level of depth. The `main` function traces $f(3)$ and then checks the recursive function against the closed form Equation (10.2) for $n = 0, 1, \ldots, 10$.

Listing 10.1 `01_recursive_f.cpp`: the recursive definition of Equation (10.1) as a C++ function, traced, and checked against its closed form.

```cpp
// 01_recursive_f.cpp -- a first recursive function, f(0) = 3, f(n) = 2 f(n-1) + 3,
// traced call by call, then checked against the closed form 6 * 2^n - 3.
#include <cstdio>

long long f(int n) {
    if (n == 0) return 3;     // base case
    return 2 * f(n - 1) + 3;  // recursive case: a smaller n
}

long long fTraced(int n, int depth) {  // the same function, talking as it works
    std::printf("%*scall f(%d)\n", 4 * depth, "", n);
    long long result;
    if (n == 0) {
        result = 3;
    } else {
        long long smaller = fTraced(n - 1, depth + 1);
        result = 2 * smaller + 3;
    }
    std::printf("%*sf(%d) returns %lld\n", 4 * depth, "", n, result);
    return result;
}

int main() {
    std::printf("Tracing f(3):\n");
    fTraced(3, 0);

    std::printf("\n n      f(n)   6*2^n-3\n");
    for (int n = 0; n <= 10; ++n) {
        long long closed = 6LL * (1LL << n) - 3;
        std::printf("%2d %9lld %9lld%s\n", n, f(n), closed,
                    f(n) == closed ? "" : "  MISMATCH");
    }
    return 0;
}
```

```
Tracing f(3):
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

 n      f(n)   6*2^n-3
 0         3         3
 1         9         9
 2        21        21
 3        45        45
 4        93        93
 5       189       189
 6       381       381
 7       765       765
 8      1533      1533
 9      3069      3069
10      6141      6141
```

Read the trace slowly, because it is the queue of **Figure 10.1** again. `f(3)` is called and, before it can return anything, calls `f(2)`, which calls `f(1)`, which calls `f(0)`. Four calls are now unfinished at the same time, each indented one step further. Only `f(0)`, the base case, can answer immediately. Then the calls finish in the *opposite* order to the one in which they started: `f(0)` returns 3, `f(1)` uses it to return 9, `f(2)` returns 21 and `f(3)` returns 45. The table confirms the induction proof: the recursive function and the closed form agree on every row. The Java and

Python versions in the Code folder (`RecursiveF.java`, `01_recursive_f.py`) print exactly the same lines.

10.2.5 Three rules, and an honest word about cost

A recursive function works when three rules hold.

1. **A base case exists and is tested first.** Every chain of calls must end in a case that returns without recursing.
2. **Every recursive call makes progress toward a base case.** The argument must get strictly smaller in some measure that cannot decrease forever: a non-negative integer that goes down, an interval that gets shorter, a list that loses a node. In `f`, n decreases by one and stops at 0. Section 10.3.4 shows what happens when this rule is broken.
3. **Trust the recursive call, and combine correctly.** Assume the call on the smaller problem is right (the induction hypothesis) and make sure the answer for the whole problem is built correctly from it.

Recursion also has a price, and it is better to name it now than to discover it later. Every call that is still waiting occupies a stack frame, so a recursion n levels deep uses memory proportional to n in addition to whatever the problem needs, and the call stack is small (Section 10.12 measures it). Each call also costs time for its bookkeeping, typically a few nanoseconds in C++, so a recursive function is rarely faster than an equivalent loop. Worst of all, a recursive definition written down naively may solve the same subproblem again and again; Section 10.5 shows a function that makes 331 million calls to produce one number that a loop finds in 90 additions. An old rule of thumb says “if a problem can be solved easily with a loop, use the loop”. The rule is sound, and this chapter shows where it applies (factorial, Fibonacci, Euclid) and where it does not: the Towers of Hanoi, trees (Chapter 11), graphs (Chapter 13), divide-and-conquer sorting (Chapter 3), and any structure nested to an unknown depth, where the recursive program is short, clear and correct, and the loop version must build by hand the very stack that recursion provides for free.

The idea is old. Recursive definitions of sequences go back centuries, and the proof by induction was made rigorous in the nineteenth century. In programming, recursion arrived with LISP, whose 1960 description by John McCarthy was titled “Recursive functions of symbolic expressions and their computation by machine” [1], and with ALGOL 60, whose report allowed a procedure to call itself. That same year Edsger Dijkstra explained how to implement such calls with a stack of activation records, the mechanism every language still uses [2]. Cormen et al. [3] and Knuth [4] treat recursion, recurrences and induction with the rigor this chapter aims at.

10.3 The Call Stack, Frame by Frame

10.3.1 Frames, revisited

Section 5.10 explained what happens when one function calls another. The processor must remember where to continue in the caller, and the callee needs room for its parameters and local variables. So the running program keeps a *call stack*, and every call pushes a *stack frame* (an *activation record*) holding the return address, the parameters, the local variables and some saved registers; every return pops the frame and hands the returned value to the frame below.

Nothing about this mechanism is special to recursion, and that is exactly the point. When `f(3)` calls `f(2)`, the program does precisely what it does when `main` calls `printf`: it pushes a new

frame. The new frame belongs to the same *function* but is a different *activation* of it, with its own copy of `n`. The `n` of `f(3)` is 3 and stays 3 while `f(2)`, `f(1)` and `f(0)` run on top of it with their own `n` of 2, 1 and 0. That is why recursion works at all: the frames keep the waiting calls apart.

10.3.2 Factorial, frame by frame

The factorial of a non-negative integer n, written $n!$, is the product of the integers from 1 to n, with $0! = 1$ (the empty product):

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1.$$

For example $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ and $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. Look at the two products side by side: $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot 4!$. The factorial of a number is that number times the factorial of the number before it, which is a recursive definition:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases} \tag{10.3}$$

The base case of Equation (10.3) is $0! = 1$ and the recursive case is $n! = n \cdot (n-1)!$. Unfolding it for small values, one step per line:

$$0! = 1$$

$$1! = 1 \cdot 0! = 1 \cdot 1 = 1$$

$$2! = 2 \cdot 1! = 2 \cdot 1 = 2$$

$$3! = 3 \cdot 2! = 3 \cdot 2 = 6$$

$$4! = 4 \cdot 3! = 4 \cdot 6 = 24$$

Figure 10.2 shows what the call stack holds while the recursive function `factRec(4)` of the next section computes 4!. In the top row the stack *winds up*: `main` calls `fact(4)`, which cannot multiply yet because it needs `fact(3)`; `fact(3)` in turn waits for `fact(2)`, and so on until `fact(0)` sits on top, six frames high counting `main`. `fact(0)` is the base case and returns 1 without calling anything. In the bottom row the stack *unwinds*: each return pops one frame, and the frame below, which has been waiting with "$k \cdot$?", finally gets its missing number and performs its one multiplication: $1 \cdot 1 = 1$, $2 \cdot 1 = 2$, $3 \cdot 2 = 6$ and $4 \cdot 6 = 24$.

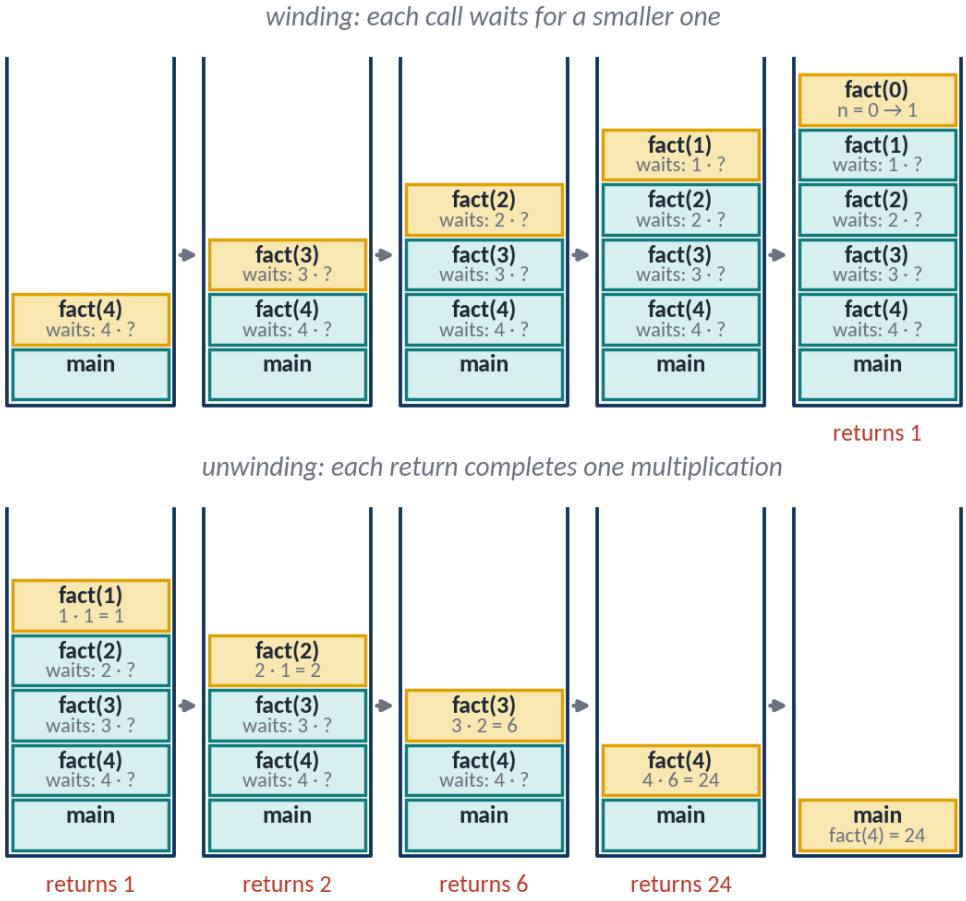


Figure 10.2 The call stack while `fact(4)` runs. Top row: every call waits for a smaller one until the base case `fact(0)` is on top. Bottom row: every return pops a frame and completes one multiplication in the frame below.

Two facts can be read straight off the figure. First, *no multiplication happens on the way down*; all four multiplications happen on the way back up, in the order $1 \cdot 1$, $2 \cdot 1$, $3 \cdot 2$, $4 \cdot 6$, which is the reverse of the order of the calls. Second, the stack reaches a height of $n + 2$ frames for $n!$ (the $n + 1$ calls `fact(n)` down to `fact(0)`, plus `main`). The time is proportional to the number of calls, $n + 1$, and so is the extra memory.

10.3.3 The trace as a program

Listing 10.2 computes $n!$ both ways, with a loop (`factIter`) and with recursion (`factRec`), and traces `fact(4)` with the same indentation trick as **Listing 10.1**. It then prints both versions side by side for $n = 0$ to 20 and finally asks for 21!.

Listing 10.2 `02_factorial.cpp`: factorial with a loop and with recursion, a trace of `fact(4)`, and the limit of 64-bit arithmetic.

```cpp
// 02_factorial.cpp -- n! computed iteratively and recursively, one recursive call
// traced frame by frame, and the limit of a 64-bit integer.
#include <cstdio>
#include <string>

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)!
}

unsigned long long factTraced(int n, int depth) {
    std::printf("%*sfact(%d): ", 4 * depth, "", n);
    if (n == 0) {
        std::printf("base case, returns 1\n");
        return 1;
    }
    std::printf("waits for fact(%d)\n", n - 1);
    unsigned long long below = factTraced(n - 1, depth + 1);
    unsigned long long result = n * below;
    std::printf("%*sfact(%d): %d * %llu = %llu\n", 4 * depth, "", n, n, below,
                result);
    return result;
}

std::string times(const std::string &decimal, int k) {  // exact decimal * small k
    std::string out;
    int carry = 0;
    for (int i = (int)decimal.size() - 1; i >= 0; --i) {
        int d = (decimal[i] - '0') * k + carry;
        out.insert(out.begin(), char('0' + d % 10));
        carry = d / 10;
    }
    while (carry > 0) {
        out.insert(out.begin(), char('0' + carry % 10));
        carry /= 10;
    }
    return out;
}
```

Listing 10.2 (Continued) *02_factorial.cpp: factorial with a loop and with recursion, a trace of fact(4), and the limit of 64-bit arithmetic.*

```cpp
int main() {
    std::printf("Tracing fact(4):\n");
    factTraced(4, 0);

    std::printf("\n n                    n! (loop)          n! (recursion)\n");
    for (int n = 0; n <= 20; ++n)
        std::printf("%2d %24llu %24llu\n", n, factIter(n), factRec(n));

    std::string exact = "1";
    for (int k = 2; k <= 21; ++k) exact = times(exact, k);
    std::printf("\n21! in 64 bits: %llu (wrong: it wrapped around)\n", factRec(21));
    std::printf("21! exactly   : %s\n", exact.c_str());
    return 0;
}
```

```
Tracing fact(4):
fact(4): waits for fact(3)
    fact(3): waits for fact(2)
        fact(2): waits for fact(1)
            fact(1): waits for fact(0)
                fact(0): base case, returns 1
            fact(1): 1 * 1 = 1
        fact(2): 2 * 1 = 2
    fact(3): 3 * 2 = 6
fact(4): 4 * 6 = 24

 n                   n! (loop)          n! (recursion)
 0                           1                       1
 1                           1                       1
 2                           2                       2
 3                           6                       6
 4                          24                      24
 5                         120                     120
 6                         720                     720
 7                        5040                    5040
 8                       40320                   40320
 9                      362880                  362880
10                     3628800                 3628800
11                    39916800                39916800
12                   479001600               479001600
13                  6227020800              6227020800
14                 87178291200             87178291200
15               1307674368000           1307674368000
16              20922789888000          20922789888000
17             355687428096000         355687428096000
18            6402373705728000        6402373705728000
19          121645100408832000      121645100408832000
20         2432902008176640000     2432902008176640000

21! in 64 bits: 14197454024290336768 (wrong: it wrapped around)
21! exactly   : 51090942171709440000
```

The trace is **Figure 10.2** in words: four lines of "waits for" going down, the base case, then four multiplications coming back up. The table shows the two versions agreeing up to $20! = 2{,}432{,}902{,}008{,}176{,}640{,}000$, which is the largest factorial that fits in 64 bits: the largest `unsigned long long` is $2^{64} - 1 \approx 1.8 \times 10^{19}$, and $21! \approx 5.1 \times 10^{19}$ is almost three times larger. C++ does not report the overflow; unsigned arithmetic silently keeps only the lowest 64 bits of the true product, which is $21! \bmod 2^{64} = 14{,}197{,}454{,}024{,}290{,}336{,}768$, a perfectly plausible-looking and completely wrong number. The program prints the exact value next to it, computed digit by digit with the small helper `times` that multiplies a decimal string by a small number, the way you multiply by hand.

The Java and Python versions print the same output. **Listing 10.3** uses `long`, which overflows in exactly the same way (Java's `long` is signed, so the program shows the 64 bits as an unsigned number with `Long.toUnsignedString` to match C++), and `BigInteger` for the exact value.

Listing 10.3 `Factorial.java`: the same program in Java; `BigInteger` gives the exact 21!.

```java
// Factorial.java -- n! computed iteratively and recursively, one recursive call
// traced frame by frame, and the limit of a 64-bit integer.
import java.math.BigInteger;

public class Factorial {
    static long factIter(int n) {
        long result = 1;
        for (int i = 2; i <= n; ++i) result *= i;  // 1 * 2 * 3 * ... * n
        return result;
    }

    static long factRec(int n) {
        if (n == 0) return 1;       // base case: 0! = 1
        return n * factRec(n - 1);  // recursive case: n! = n * (n-1)!
    }

    static long factTraced(int n, int depth) {
        String pad = " ".repeat(4 * depth);
        System.out.printf("%sfact(%d): ", pad, n);
        if (n == 0) {
            System.out.println("base case, returns 1");
            return 1;
        }
        System.out.printf("waits for fact(%d)%n", n - 1);
        long below = factTraced(n - 1, depth + 1);
        long result = n * below;
        System.out.printf("%sfact(%d): %d * %d = %d%n", pad, n, n, below, result);
        return result;
    }

    public static void main(String[] args) {
        System.out.println("Tracing fact(4):");
        factTraced(4, 0);

        System.out.println(
            "\n n                    n! (loop)          n! (recursion)");
        for (int n = 0; n <= 20; ++n)
            System.out.printf("%2d %24d %24d%n", n, factIter(n), factRec(n));
```

Listing 10.3 (Continued) *Factorial.java: the same program in Java; BigInteger gives the exact 21!.*

```java
        BigInteger exact = BigInteger.ONE;
        for (int k = 2; k <= 21; ++k) exact = exact.multiply(BigInteger.valueOf(k));
        // a long has 64 bits; read them as an unsigned number, as C++ does
        System.out.printf("%n21! in 64 bits: %s (wrong: it wrapped around)%n",
                          Long.toUnsignedString(factRec(21)));
        System.out.printf("21! exactly   : %s%n", exact);
    }
}
```

Python's integers never overflow: they grow to as many digits as needed. **Listing 10.4** therefore computes 21! exactly with the ordinary recursive function and imitates the 64-bit result with `% 2**64`, the remainder that 64 bits keep.

Listing 10.4 `02_factorial.py`: the same program in Python, where integers grow as needed.

```python
# 02_factorial.py -- n! computed iteratively and recursively, one recursive call
# traced frame by frame, and the limit of a 64-bit integer.


def fact_iter(n):
    result = 1
    for i in range(2, n + 1):  # 1 * 2 * 3 * ... * n
        result *= i
    return result


def fact_rec(n):
    if n == 0:
        return 1                # base case: 0! = 1
    return n * fact_rec(n - 1)  # recursive case: n! = n * (n-1)!


def fact_traced(n, depth):
    pad = " " * (4 * depth)
    print(f"{pad}fact({n}): ", end="")
    if n == 0:
        print("base case, returns 1")
        return 1
    print(f"waits for fact({n - 1})")
    below = fact_traced(n - 1, depth + 1)
    result = n * below
    print(f"{pad}fact({n}): {n} * {below} = {result}")
    return result


def main():
    print("Tracing fact(4):")
    fact_traced(4, 0)

    print("\n n                    n! (loop)          n! (recursion)")
    for n in range(21):
        print(f"{n:2d} {fact_iter(n):24d} {fact_rec(n):24d}")

    exact = fact_rec(21)     # Python integers never overflow
    wrapped = exact % 2**64  # what 64 bits keep of it
    print(f"\n21! in 64 bits: {wrapped} (wrong: it wrapped around)")
    print(f"21! exactly   : {exact}")


if __name__ == "__main__":
    main()
```

10.3.4 When the base case is missing

Break rule 2 of Section 10.2.5 and see what happens. Suppose we wrote the recursive case as `return n * factRec(n + 1);` by a slip of the finger. The argument now grows: `factRec(4)` calls `factRec(5)`, which calls `factRec(6)`, and the base case `n == 0` is never reached. The program does not loop quietly forever, as a `while (true)` would; each call pushes a frame, the frames pile up, and after some tens of thousands of calls the stack is full. What happens next depends on the language.

- **C++** reads or writes memory just beyond the end of the stack, where the operating system has placed a guard page that belongs to nobody, and the program is killed at once with a *segmentation fault* (on Linux and macOS; Windows reports a *stack overflow* exception). No message tells you which function was responsible; a debugger or AddressSanitizer does.
- **Java** checks the stack on every call and throws `StackOverflowError`, with a stack trace showing the same method repeated a thousand times.

- **Python** counts the depth of the interpreter's stack and raises `RecursionError: maximum recursion depth exceeded` when it passes a limit, 1,000 by default.

This failure has a name that has become the name of a famous programmers' website: *stack overflow*. Section 10.12 measures exactly where it happens in each language, and Section 10.13 shows how to avoid it for recursions that are legitimately deep.

> **WARNING:** A recursion that does not make progress toward its base case is not a slow program; it is a crash, usually within a fraction of a second. The two usual causes are a base case that the arguments can jump over (testing `n == 0` when `n` can go from 1 to -1, for example in a function that subtracts 2) and an argument that does not shrink (passing `n` instead of `n - 1`). Test the smallest inputs first: 0, 1 and 2 catch most such bugs.

10.4 Factorial: Iterative and Recursive

10.4.1 Two algorithms for one function

The *iterative* method computes $n!$ with a loop that multiplies each number into a running product, following the definition $n! = n(n-1)(n-2)\cdots 1$ from left to right or, as `factIter` does, from right to left. The *recursive* method follows Equation (10.3). **Pseudocode 10.1** writes both in pseudocode.

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)
```

Both are correct, and the proofs have the same shape. For the loop, the *loop invariant* "at the start of the iteration with index i, result $= (i-1)!$" holds before the first iteration ($1 = 1!$) and is preserved by line 3 ($(i-1)! \cdot i = i!$), so after the last iteration result $= n!$. For the recursive version, induction on n: the base case returns $1 = 0!$, and if the call on $n-1$ returns $(n-1)!$, line 3 returns $n \cdot (n-1)! = n!$. A loop invariant is induction on the number of iterations; induction on n is the same argument for the recursion.

10.4.2 What each costs

Let $T(n)$ be the running time of `FACTORIAL-RECURSIVE(n)`. The base case costs some constant c_0. For $n \geq 1$, the function does a constant amount of work c (a comparison, a multiplication, a call and a return) plus the work of the recursive call on $n-1$:

$$T(0) = c_0, \qquad T(n) = T(n-1) + c \quad (n \geq 1). \tag{10.4}$$

Equation (10.4) is a *recurrence*, an equation that defines a function in terms of its own smaller values, just as Equation (10.1) did. Unfolding it one step at a time:

$$T(n) = T(n-1) + c = T(n-2) + 2c = T(n-3) + 3c = \cdots = T(0) + nc$$
$$= c_0 + cn = \Theta(n).$$

The loop makes $n - 1$ iterations of constant cost, which is also $\Theta(n)$. The two differ in memory. The loop needs two variables whatever n is: $O(1)$ extra memory. The recursion keeps $n + 1$ frames on the stack at its deepest point: $\Theta(n)$ extra memory. For factorial this difference never matters in practice, because n cannot usefully exceed 20 in 64-bit arithmetic, and even with big integers a million-digit product is limited by the multiplications, not the stack. But the difference is real, and Section 10.12 shows how quickly it becomes decisive for recursions that go deeper.

> **KEY INSIGHT:** The *time* of a recursion is the total work of all its calls. The *extra memory* of a recursion is the largest number of frames that are on the stack *at the same moment*, times the size of a frame. For factorial both are proportional to n. Section 10.5 shows a recursion whose time is exponential while its memory is only linear, and Section 10.7 one whose time and memory are both logarithmic.

10.5 Fibonacci and the Rabbits

10.5.1 The rabbit problem

In 1202 Leonardo of Pisa, known today as Fibonacci, published *Liber Abaci*, the book that brought the Hindu–Arabic numerals and their arithmetic to European merchants. Among hundreds of practical problems it contains one about rabbits [5]. A man puts one pair of newborn rabbits, a male and a female, in an enclosed place. A newborn pair takes one month to grow up, and from then on every grown-up pair produces one new pair each month. No rabbit dies. How many pairs are there at the end of a year?

Count them month by month, as **Figure 10.3** does, at the start of each month. In month 1 there is one young pair (Y). In month 2 it has grown up into an adult pair (A), but has not yet given birth: still one pair. In month 3 the adult pair produces a young pair: two pairs. In month 4 the first pair produces again, and the pair born in month 3 grows up: three pairs. In month 5 both adult pairs produce: five pairs.

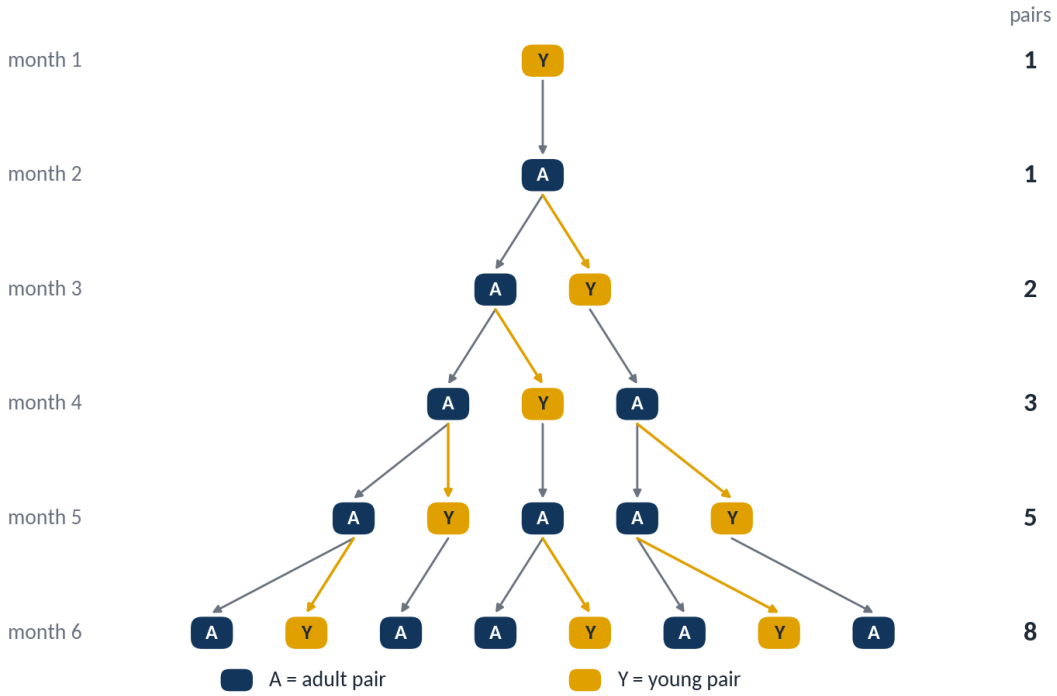


Figure 10.3 The first six months of Fibonacci's rabbits. Grey arrows show a pair living on into the next month; amber arrows show an adult pair producing a new young pair.

The rule behind the picture is this: the pairs alive in month n are all the pairs alive in month $n-1$ (nobody dies) plus one newborn pair for every pair that was already an adult in month $n-1$, and the adults of month $n-1$ are exactly the pairs that were alive in month $n-2$. So the number of pairs in a month is the sum of the numbers in the two months before it. The first part of **Listing 10.5** below follows adult and young pairs through the year and reaches 144 pairs in month 12, which is Fibonacci's answer. In modern notation, with $F(0)=0$ added in front for convenience,

$$F(0)=0, \qquad F(1)=1, \qquad F(n)=F(n-1)+F(n-2) \quad (n \geq 2), \tag{10.5}$$

which gives the *Fibonacci sequence* 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, … The number of pairs in month n is $F(n)$. The sequence was named after Fibonacci in the nineteenth century by Édouard Lucas, the same Lucas who invented the Towers of Hanoi. It turns up in the spirals of sunflower heads and pine cones, in the analysis of Euclid's algorithm (Section 10.6.4), in the balanced search trees of later courses, and, as we are about to see, in the cost of its own naive computation.

10.5.2 Naive recursion, and why it explodes

Equation (10.5) translates directly into code:

```
long long fibNaive(int n) {
    if (n <= 1) return n;                          // F(0) = 0, F(1) = 1
    return fibNaive(n - 1) + fibNaive(n - 2);
}
```

This function has two base cases and a recursive case that makes *two* recursive calls. It is correct; induction on n, with two base cases and the hypothesis that both smaller calls are right, proves it in two lines. It is also, as a way to compute Fibonacci numbers, one of the worst algorithms in this book. **Figure 10.4**(a) shows why. The calls of `fibNaive(5)` form a tree: `F(5)` calls `F(4)` and `F(3)`; `F(4)` calls `F(3)` and `F(2)`; and so on down to the leaves `F(1)` and `F(0)`. The tree has 15 nodes, one

per call, and many of them repeat work already done: `F(3)` is computed twice, `F(2)` three times, `F(1)` five times. The amber nodes are all wasted effort.

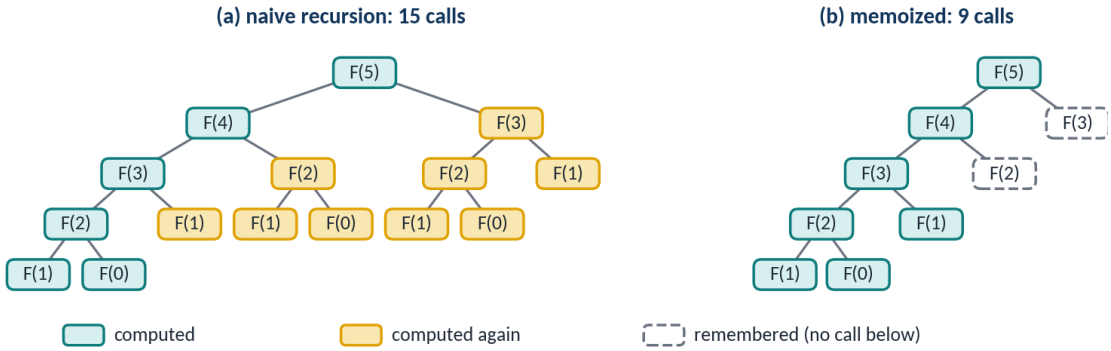


Figure 10.4 The calls made to compute $F(5)$. (a) Naive recursion: 15 calls, most of them recomputing values already known. (b) With memoization: 9 calls; a value already in the table is returned without any call below it.

How many calls does `fibNaive(n)` make in general? Let $C(n)$ be the number. A call with $n \le 1$ makes no further calls, so $C(0) = C(1) = 1$. A call with $n \ge 2$ is one call plus all the calls made by its two children:

$$C(0) = C(1) = 1, \qquad C(n) = C(n-1) + C(n-2) + 1 \quad (n \ge 2). \tag{10.6}$$

Equation (10.6) is almost the Fibonacci recurrence itself, and its solution is

$$C(n) = 2F(n+1) - 1. \tag{10.7}$$

The proof is by induction. For $n = 0$: $2F(1) - 1 = 1$; for $n = 1$: $2F(2) - 1 = 2 - 1 = 1$. For $n \ge 2$, assuming Equation (10.7) for $n-1$ and $n-2$:

$$\begin{aligned} C(n) &= (2F(n) - 1) + (2F(n-1) - 1) + 1 = 2(F(n) + F(n-1)) - 1 \\ &= 2F(n+1) - 1. \end{aligned}$$

So the naive function makes about twice as many calls as the answer it computes. That is a disaster, because the Fibonacci numbers grow exponentially. Let $\varphi = (1 + \sqrt{5})/2 \approx 1.618$, the *golden ratio*, the positive root of $x^2 = x + 1$. Then

$$F(n) \ge \varphi^{\,n-2} \quad \text{for every } n \ge 1. \tag{10.8}$$

Proof by induction: $F(1) = 1 \ge \varphi^{-1} \approx 0.618$ and $F(2) = 1 \ge \varphi^0 = 1$; and for $n \ge 3$, $F(n) = F(n-1) + F(n-2) \ge \varphi^{n-3} + \varphi^{n-4} = \varphi^{n-4}(\varphi + 1) = \varphi^{n-4}\varphi^2 = \varphi^{n-2}$, using $\varphi + 1 = \varphi^2$. (The same argument with $\le$ and exponent $n-1$ gives $F(n) \le \varphi^{n-1}$.) So $C(n) = 2F(n+1) - 1 = \Theta(\varphi^n)$: every increase of n by one multiplies the work by about 1.618, and every increase by five multiplies it by about $\varphi^5 \approx 11$.

The second part of **Listing 10.5** counts the calls for real and prints $2F(n+1) - 1$ beside the count; the third part counts how often each $F(k)$ is computed inside `fibNaive(6)`. Those counts, 1, 1, 2, 3, 5, 8 for $F(6)$ down to $F(1)$, are themselves Fibonacci numbers: the naive recursion computes $F(k)$ exactly $F(n-k+1)$ times for $k \ge 1$.

10.5.3 Memoization: remember what you have computed

The cure for recomputation is a memory. Keep a table `memo` with one entry per value of n, initially marked "unknown". Before computing $F(n)$, look in the table; if the value is there, return it; otherwise compute it recursively, *store it*, and return it. This technique is called *memoization* (from "memo", a note to oneself), and a recursion with memoization is called *top-down dynamic programming*. **Pseudocode 10.2** applies it to Equation (10.5).

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]
```

Figure 10.4(b) shows the effect on $F(5)$. The left spine $F(5), F(4), F(3), F(2), F(1)$ computes each value for the first time; every right child finds its value already in the table (the dashed nodes) and returns at once. Each $F(k)$ with $2 \le k \le n$ is computed exactly once, at a cost of two calls to its children, so the total number of calls is $1 + 2(n-1) = 2n - 1$ for $n \ge 1$: 9 for $n = 5$, 179 for $n = 90$. The time is $\Theta(n)$ and so is the memory (the table and the stack). The recursion is still exactly Equation (10.5); only the memory is new.

> **KEY INSIGHT:** An exponential recursion is often exponential only because it solves the same subproblems again and again. If the number of *distinct* subproblems is small (here, $n + 1$), storing each answer the first time it is found turns the exponential algorithm into a polynomial one. This is the whole idea of *dynamic programming*.

Python offers memoization ready-made: decorate the naive function with `@functools.lru_cache(maxsize=None)` (or `@functools.cache`) and every result is remembered automatically, in the circular doubly linked list with a sentinel that Section 9.12.3 described.

10.5.4 Bottom-up: no recursion at all

Memoization computes the values in the order the recursion asks for them, which for Fibonacci is $F(1), F(2), \ldots, F(n)$. We might as well fill the table in that order directly, with a loop, starting from the base cases:

```
F[0] = 0;  F[1] = 1;
for (int i = 2; i <= n; ++i) F[i] = F[i - 1] + F[i - 2];
```

This is *bottom-up dynamic programming*: no recursion, no stack, $\Theta(n)$ time. Look at the loop body once more: it only ever reads the last two entries of the table. So the table is unnecessary; two variables, $a = F(i)$ and $b = F(i+1)$, are enough, and each step replaces the pair (a, b) by $(b, a + b)$. That is `fibTwoVars` in **Listing 10.5**: $\Theta(n)$ time and $O(1)$ memory.

10.5.5 Tail recursion: the loop, written as a recursion

The pair-updating loop can be written as a recursion too, by passing the pair as parameters:

```
long long fibAux(long long x, long long y, int n) {   // x = F(i), y = F(i+1)
    if (n == 0) return x;
    return fibAux(y, x + y, n - 1);
}
long long fibTail(int n) { return fibAux(0, 1, n); }
```

Each call shifts the pair one step along the sequence and decrements the number of steps still to go; after n steps, `x` holds $F(n)$. The recursive call is the *very last thing* `fibAux` does: its result is returned unchanged, with no addition or multiplication waiting for it. Such a call is a *tail call*, and a function whose recursive calls are all tail calls is *tail recursive*. `fibNaive` is not tail recursive (it adds the results of its two calls), and neither is `factRec` (it multiplies). `fibAux` makes $n + 1$ calls, linear like the loop, and Section 10.11 shows that a compiler may turn it into exactly that loop.

10.5.6 Fast doubling: $O(\log n)$

We can do better than linear. The Fibonacci numbers satisfy a pair of *doubling identities*:

$$F(2k) = F(k)(2F(k+1) - F(k)), \qquad F(2k+1) = F(k)^2 + F(k+1)^2. \tag{10.9}$$

Here is one way to see where they come from. The step $(F(k), F(k+1)) \to (F(k+1), F(k+2))$ is a linear map, so it can be written with the 2×2 matrix $Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, and by induction on n

$$Q^n = \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix} \quad (n \geq 1).$$

Since $Q^{2k} = Q^k Q^k$, multiply the right-hand side by itself with $n = k$ and compare entries: the top-right entry gives $F(2k) = F(k+1)F(k) + F(k)F(k-1) = F(k)(F(k+1) + F(k-1))$, and substituting $F(k-1) = F(k+1) - F(k)$ gives the first identity; the top-left entry gives $F(2k+1) = F(k+1)^2 + F(k)^2$, the second. (Check with $k = 3$: $F(6) = 2 \cdot (2 \cdot 3 - 2) = 8$ and $F(7) = 2^2 + 3^2 = 13$. Correct.)

The identities give $F(n)$ and $F(n+1)$ from $F(k)$ and $F(k+1)$ with $k = \lfloor n/2 \rfloor$: if $n = 2k$, the pair $(F(2k), F(2k+1))$ comes straight from Equation (10.9); if $n = 2k+1$, it is $(F(2k+1), F(2k) + F(2k+1))$. So `fibPair(n)` in **Listing 10.5** makes one recursive call on $\lfloor n/2 \rfloor$ and a few multiplications, and the argument halves at each level. It makes $\lfloor \lg n \rfloor + 2$ calls for $n \geq 1$ (8 for $n = 90$), and runs in $\Theta(\log n)$ time and stack, if we count each multiplication of numbers that fit in a machine word as one step [6].

10.5.7 Binet's closed form

Finally, there is a closed form, published by Jacques Binet in 1843 and known earlier to de Moivre and Euler [7]. With $\varphi = (1 + \sqrt{5})/2$ and $\psi = (1 - \sqrt{5})/2 \approx -0.618$, the two roots of $x^2 = x + 1$,

$$F(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}}. \tag{10.10}$$

The proof is another two-line induction. For $n = 0$ the right side is $0/\sqrt{5} = 0$, and for $n = 1$ it is $\sqrt{5}/\sqrt{5} = 1$. If the formula holds for $n-1$ and $n-2$, then, because $\varphi^n = \varphi^{n-1} + \varphi^{n-2}$ (multiply $\varphi^2 = \varphi + 1$ by φ^{n-2}) and likewise for ψ, the right side for n equals the sum of the right sides for $n-1$ and $n-2$, which is $F(n-1) + F(n-2) = F(n)$. Since $\psi^n/\sqrt{5}$ lies strictly between -0.5 and 0.5 for every $n \geq 0$, $F(n)$ is simply $\varphi^n/\sqrt{5}$ *rounded to the nearest integer*, and that is what `fibBinet` computes with one call to `pow`.

It is the fastest method of all, and it is wrong. A `double` carries 53 significant bits, about 16 decimal digits, and φ itself is already rounded. The error of φ^n grows with n, and as soon as it

exceeds one half, the rounding lands on the wrong integer. The last part of **Listing 10.5** searches for the first n at which that happens.

10.5.8 The six ways in one program

Listing 10.5 puts everything together: the rabbit table, the naive recursion with its call counts, the counts of repeated subproblems, all six ways of computing $F(90)$, and the search for the first failure of Binet's formula.

Listing 10.5 `03_fibonacci.cpp`: the rabbits, and $F(n)$ by naive recursion, memoization, a bottom-up table, two variables, tail recursion, fast doubling and Binet's formula.

```
// 03_fibonacci.cpp -- Fibonacci's rabbits, and F(n) computed six ways:
// naive recursion, memoized recursion, bottom-up table, two variables,
// tail recursion, fast doubling, and Binet's closed form.
#include <cmath>
#include <cstdio>
#include <utility>
#include <vector>
using namespace std;
typedef long long ll;

ll naiveCalls = 0;
ll fibNaive(int n) {
    ++naiveCalls;
    if (n <= 1) return n;  // F(0) = 0, F(1) = 1
    return fibNaive(n - 1) + fibNaive(n - 2);
}

vector<int> computed;  // computed[k]: times F(k) was asked
ll fibCounted(int n) {
    ++computed[n];
    if (n <= 1) return n;
    return fibCounted(n - 1) + fibCounted(n - 2);
}

ll memoCalls = 0;
ll fibMemo(int n, vector<ll> &memo) {  // memo[k] == -1: not known yet
    ++memoCalls;
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];  // remembered: no recursion
    memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);
    return memo[n];
}

ll fibTable(int n) {  // bottom-up, the whole table
    if (n <= 1) return n;
    vector<ll> F(n + 1);
    F[0] = 0;
    F[1] = 1;
    for (int i = 2; i <= n; ++i) F[i] = F[i - 1] + F[i - 2];
    return F[n];
}
```

Listing 10.5 (Continued) *03_fibonacci.cpp: the rabbits, and F(n) by naive recursion, memoization, a bottom-up table, two variables, tail recursion, fast doubling and Binet's formula.*

```
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); }

ll doublingCalls = 0;
pair<ll, ll> fibPair(int n) {  // returns (F(n), F(n+1))
    ++doublingCalls;
    if (n == 0) return {0, 1};
    pair<ll, ll> h = fibPair(n / 2);  // (F(k), F(k+1)), k = n / 2
    ll a = h.first, b = h.second;
    ll c = a * (2 * b - a);  // F(2k)   = F(k) (2F(k+1) - F(k))
    ll d = a * a + b * b;    // F(2k+1) = F(k)^2 + F(k+1)^2
    if (n % 2 == 0) return {c, d};
    return {d, c + d};
}
ll fibDoubling(int n) { return fibPair(n).first; }

ll fibBinet(int n) {  // round(phi^n / sqrt 5)
    double phi = (1 + sqrt(5.0)) / 2;
    return llround(pow(phi, n) / sqrt(5.0));
}
```

Listing 10.5 (Continued) *03_fibonacci.cpp: the rabbits, and F(n) by naive recursion, memoization, a bottom-up table, two variables, tail recursion, fast doubling and Binet's formula.*

```
int main() {
    printf("Fibonacci's rabbits (pairs at the start of each month)\n");
    printf("month  adult  young  total\n");
    ll adult = 0, young = 1;
    for (int month = 1; month <= 12; ++month) {
        printf("%5d %6lld %6lld %6lld\n", month, adult, young, adult + young);
        ll born = adult;  // every adult pair has a pair
        adult += young;   // last month's young grow up
        young = born;
    }

    printf("\nNaive recursion: calls made to compute F(n)\n");
    printf(" n       F(n)      calls  2F(n+1)-1\n");
    int ns[] = {0, 1, 2, 3, 4, 5, 6, 10, 15, 20, 25, 30};
    for (int n : ns) {
        naiveCalls = 0;
        ll v = fibNaive(n);
        printf("%2d %10lld %10lld %10lld\n", n, v, naiveCalls,
               2 * fibTwoVars(n + 1) - 1);
    }

    computed.assign(7, 0);
    fibCounted(6);
    printf("\nInside fibNaive(6), F(k) is computed this many times:\n");
    for (int k = 6; k >= 0; --k) printf("  F(%d): %d\n", k, computed[k]);

    int n = 90;
    vector<ll> memo(n + 1, -1);
    ll m = fibMemo(n, memo);
    ll t = fibTail(n);
    ll d = fibDoubling(n);
    printf("\nF(%d) six ways\n", n);
    printf("  memoized recursion : %lld  (%lld calls)\n", m, memoCalls);
    printf("  bottom-up table    : %lld\n", fibTable(n));
    printf("  two variables      : %lld\n", fibTwoVars(n));
    printf("  tail recursion     : %lld  (%lld calls)\n", t, tailCalls);
    printf("  fast doubling      : %lld  (%lld calls)\n", d, doublingCalls);
    printf("  Binet, double      : %lld\n", fibBinet(n));

    int k = 0;
    while (fibBinet(k) == fibTwoVars(k)) ++k;
    printf("\nBinet's formula in double precision is exact up to F(%d);\n", k - 1);
    printf("F(%d) = %lld, but it gives %lld\n", k, fibTwoVars(k), fibBinet(k));
    return 0;
}
```

```
Fibonacci's rabbits (pairs at the start of each month)
month  adult  young  total
    1      0      1      1
    2      1      0      1
    3      1      1      2
    4      2      1      3
    5      3      2      5
    6      5      3      8
    7      8      5     13
    8     13      8     21
    9     21     13     34
   10     34     21     55
   11     55     34     89
   12     89     55    144

Naive recursion: calls made to compute F(n)
 n       F(n)      calls  2F(n+1)-1
 0          0          1          1
 1          1          1          1
 2          1          3          3
 3          2          5          5
 4          3          9          9
 5          5         15         15
 6          8         25         25
10         55        177        177
15        610       1973       1973
20       6765      21891      21891
25      75025     242785     242785
30     832040    2692537    2692537

Inside fibNaive(6), F(k) is computed this many times:
  F(6): 1
  F(5): 1
  F(4): 2
  F(3): 3
  F(2): 5
  F(1): 8
  F(0): 5

F(90) six ways
  memoized recursion : 2880067194370816120  (179 calls)
  bottom-up table    : 2880067194370816120
  two variables      : 2880067194370816120
  tail recursion     : 2880067194370816120  (91 calls)
  fast doubling      : 2880067194370816120  (8 calls)
  Binet, double      : 2880067194370824704

Binet's formula in double precision is exact up to F(70);
F(71) = 308061521170129, but it gives 308061521170130
```

Every claim of this section is in that output. The rabbit table reaches 144 in month 12, with $F(n-1)$ adult and $F(n-2)$ young pairs in month n. The measured call counts of the naive recursion agree with Equation (10.7) on every row, up to 2,692,537 calls for $F(30) = 832{,}040$. Inside `fibNaive(6)` the repeated computations are Fibonacci numbers themselves. The five exact methods agree on $F(90) = 2{,}880{,}067{,}194{,}370{,}816{,}120$, with 179 calls for memoization ($2 \cdot 90 - 1$), 91 for the tail recursion and 8 for fast doubling. Binet's formula in `double` precision is off by 8,584 at $n = 90$ and exact only up to $F(70)$: at $n = 71$ it answers 308,061,521,170,130 instead of 308,061,521,170,129. (The Java program `Fibonacci.java` and the Python program `03_fibonacci.py` print the same lines, including the same failure at 71: all three use the same IEEE 754 `double` arithmetic and a correctly behaving `pow`.) The lesson is general: a closed form evaluated in floating point is an approximation, and exact integer answers need exact integer arithmetic.

10.5.9 Measured: calls and time

How long does all this take? The programs `04_fib_bench.cpp`, `FibBench.java` and `04_fib_bench.py` in the Code folder time the naive recursion for $n = 20,25, \ldots,40$ and the other methods for $n = 90$ (averaged over many runs, because a single run is too short to measure). The numbers below are one real run on the machine used for this book, a shared two-core Linux server; yours will differ, but not their shape. First C++ (g++ 13.3 with `-O2`):

```
Naive recursion
  n           F(n)        calls     time (ms)
 20           6765        21891         0.016
 25          75025       242785         0.150
 30         832040      2692537         1.502
 35        9227465     29860703        16.958
 40      102334155    331160281       196.942

F(90), average of 100000 runs
  memoized recursion :    888.4 ns
  two variables      :     47.2 ns
  tail recursion     :     28.0 ns
  fast doubling      :     16.1 ns
  Binet, double      :     15.5 ns
```

then Java (javac 21, HotSpot JIT, with a warm-up pass before the fast methods are timed):

```
Naive recursion
  n           F(n)        calls     time (ms)
 20           6765        21891         0.316
 25          75025       242785         0.369
 30         832040      2692537         3.964
 35        9227465     29860703        44.404
 40      102334155    331160281       523.244

F(90), average of 100000 runs
  memoized recursion :    582.2 ns
  two variables      :     13.9 ns
  tail recursion     :    246.9 ns
  fast doubling      :     52.0 ns
  Binet, double      :     14.6 ns
```

and Python 3.11 (the naive recursion stops at $n = 35$, because $n = 40$ would take about 20 seconds):

```
Naive recursion
  n           F(n)        calls     time (ms)
 20           6765        21891         1.156
 25          75025       242785        12.928
 30         832040      2692537       147.673
 35        9227465     29860703      1769.737

F(90), average of 20000 runs
  memoized recursion :  11122.2 ns
  two variables      :   2074.0 ns
  tail recursion     :   4318.0 ns
  fast doubling      :   1116.2 ns
  Binet, double      :    184.2 ns
```

Figure 10.5(a) plots the three call counts on a logarithmic scale: the naive count, $2F(n+1)-1$, is a straight line, which is what exponential growth looks like on such a scale, while $2n-1$ and $\lfloor \lg n \rfloor + 2$ hug the bottom. **Figure 10.5**(b) plots the measured naive times: three nearly parallel straight lines, each rising by a factor of about 11 every five steps of n, exactly φ^5. The languages

differ by a constant factor (C++ is about 2.5 times faster than Java here and about 100 times faster than CPython) but the *slope*, which is set by the algorithm, is the same for all three.

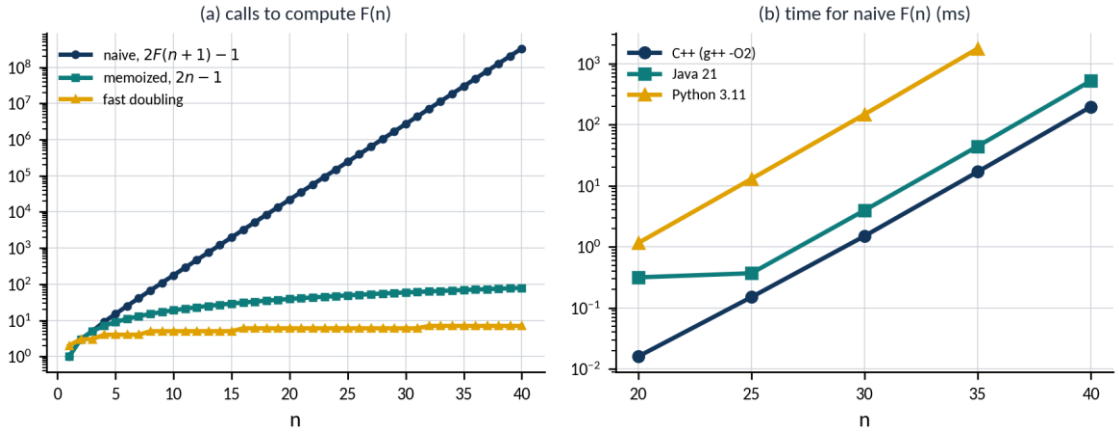


Figure 10.5 (a) Calls made to compute $F(n)$ by the three recursive methods, logarithmic scale. (b) Measured time of the naive recursion in C++, Java and Python, September 2026, logarithmic scale.

Table 10.1 summarizes the methods. The measured times for $F(90)$ repay a careful look, because they contain three small surprises. Memoization is the slowest of the exact linear methods in all three languages, even though it makes only 179 calls: most of its time goes into allocating and filling the table. The tail recursion in C++ is at least as fast as the loop, because (Section 10.11) GCC turns it *into* a loop; in Java it is about 17 times slower than the loop and in Python about twice as slow, because neither the JVM nor CPython does that. And in C++ fast doubling and Binet's formula take about the same 16 ns: at $n = 90$ the logarithmic method needs only 8 calls, and `pow` is not free either. The asymptotic analysis tells you which method wins as n grows; only measurement tells you what happens at a particular 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 $F(70)$

One more remark: the values in this section stop at $F(92) = 7{,}540{,}113{,}804{,}746{,}346{,}429$, the largest Fibonacci number that fits in a signed 64-bit integer. Beyond that, all the exact methods need big integers (Java's `BigInteger`, Python's `int`), and the cost of a multiplication then grows with the number of digits, which is why serious libraries use fast doubling with fast multiplication [6].

10.6 Euclid's Greatest Common Divisor

10.6.1 The problem and the idea

The *greatest common divisor* of two non-negative integers a and b, not both zero, written $\gcd(a, b)$, is the largest integer that divides both. For example, the divisors of 228 are 1, 2, 3, 4, 6, 12, 19, 38, 57, 76, 114 and 228, the divisors of 90 are 1, 2, 3, 5, 6, 9, 10, 15, 18, 30, 45 and 90, and the largest number in both lists is 6, so $\gcd(228,90) = 6$. Listing divisors is slow for large numbers. About 300 BC, Euclid's *Elements* (Book VII, Propositions 1 and 2) described a far better method [8], which Knuth calls "the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day" [9]. In modern form it rests on one equation:

$$\gcd(a, 0) = a, \qquad \gcd(a, b) = \gcd(b,\ a \bmod b) \quad (b > 0), \tag{10.11}$$

where $a \bmod b$ is the remainder of dividing a by b (`a % b` in all three of our languages, for non-negative operands).

Why is Equation (10.11) **true?** Write $a = qb + r$ with quotient $q = \lfloor a/b \rfloor$ and remainder $r = a \bmod b$, $0 \le r < b$. If a number d divides both a and b, it divides $a - qb = r$; so every common divisor of a and b is a common divisor of b and r. Conversely, if d divides b and r, it divides $qb + r = a$; so every common divisor of b and r is a common divisor of a and b. The two pairs have *exactly the same* common divisors, hence the same greatest one. And $\gcd(a, 0) = a$ because every number divides 0, so the common divisors of a and 0 are just the divisors of a, the largest of which is a itself.

Why does it stop? The second argument goes from b to $a \bmod b$, which is strictly smaller than b and never negative. A non-negative integer cannot decrease forever, so after finitely many steps the second argument is 0 and the base case answers. That is rule 2 of Section 10.2.5, with the measure "the second argument".

10.6.2 By hand, and as a picture

Here is $\gcd(228,90)$, one division per line:

$$228 = 2 \cdot 90 + 48 \qquad \Rightarrow \gcd(228,90) = \gcd(90,48)$$

$$90 = 1 \cdot 48 + 42 \qquad \Rightarrow \gcd(90,48) = \gcd(48,42)$$

$$48 = 1 \cdot 42 + 6 \qquad \Rightarrow \gcd(48,42) = \gcd(42,6)$$

$$42 = 7 \cdot 6 + 0 \qquad \Rightarrow \gcd(42,6) = \gcd(6,0) = 6$$

The answer is the last non-zero remainder, 6, after four divisions. **Figure 10.6** shows the same computation as geometry, which is closer to how Euclid thought about it. Take a 228×90 rectangle and cut off the largest squares you can: two 90×90 squares, leaving a 48×90 strip. From that, cut one 48×48 square, leaving 48×42; then one 42×42, leaving 6×42; and that strip is tiled exactly by seven 6×6 squares. The side of the smallest square, 6, measures both sides of the original rectangle a whole number of times ($228 = 38 \cdot 6$, $90 = 15 \cdot 6$), and no larger square does.

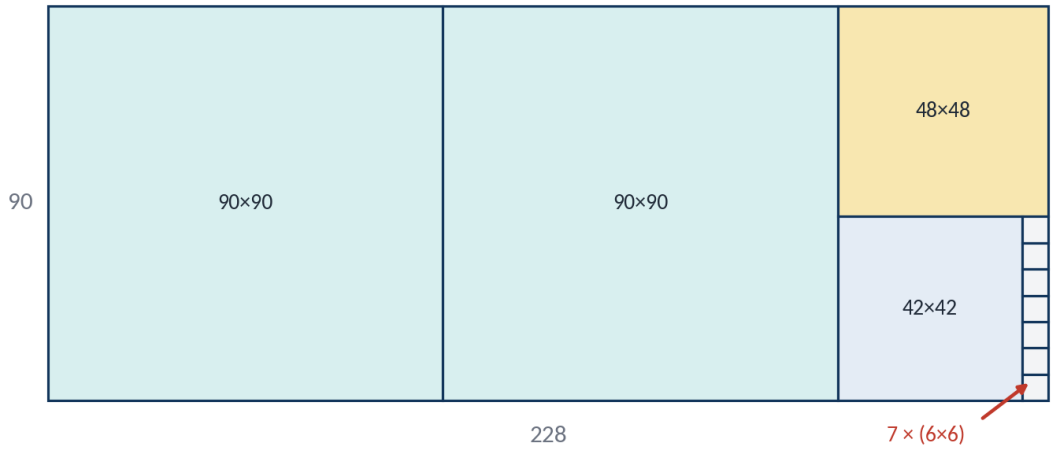


Figure 10.6 Euclid's algorithm on 228 and 90 as cutting squares from a rectangle: each division removes as many squares of the shorter side as fit; the last square, 6×6, tiles what remains exactly, so $\gcd(228,90) = 6$.

10.6.3 The program

Pseudocode 10.3 gives the recursive and the iterative form. The recursive call is a tail call: its result is returned unchanged. That is why the iterative form is so easy to write: it replaces "call yourself with $(b, a \bmod b)$" by "set $(a, b) \leftarrow (b, a \bmod b)$ and go back to the top".

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
```

Listing 10.6 traces two computations, compares the two forms on a few pairs, finds the pair below 100 that needs the most divisions, and runs the algorithm on consecutive Fibonacci numbers.

Listing 10.6 `05_gcd.cpp`: Euclid's algorithm recursively and iteratively, traced, and its worst case.

```cpp
// 05_gcd.cpp -- Euclid's algorithm, recursive and iterative, a traced run,
// and Lame's worst case: consecutive Fibonacci numbers.
#include <cstdio>

long gcdRec(long a, long b) {
    if (b == 0) return a;     // gcd(a, 0) = a
    return gcdRec(b, a % b);  // gcd(a, b) = gcd(b, a mod b)
}

long gcdIter(long a, long b) {
    while (b != 0) {
        long r = a % b;
        a = b;
        b = r;
    }
    return a;
}

long gcdTraced(long a, long b) {
    if (b == 0) {
        std::printf("  gcd(%ld, 0): b = 0, the answer is %ld\n", a, a);
        return a;
    }
    std::printf("  gcd(%ld, %ld): %ld = %ld * %ld + %ld\n", a, b, a, a / b, b,
                a % b);
    return gcdTraced(b, a % b);
}

int steps(long a, long b) {  // number of divisions performed
    return b == 0 ? 0 : 1 + steps(b, a % b);
}

int main() {
    std::printf("Traced:\n");
    gcdTraced(228, 90);
    gcdTraced(1071, 462);

    long pairs[][2] = {{228, 90}, {1071, 462}, {90, 228},
                       {17, 5},    {12, 0},     {144, 89}};
    std::printf("\n     a     b   recursive  iterative  divisions\n");
    for (auto &p : pairs)
        std::printf("%6ld %5ld %11ld %10ld %10d\n", p[0], p[1], gcdRec(p[0], p[1]),
                    gcdIter(p[0], p[1]), steps(p[0], p[1]));
}
```

Listing 10.6 (Continued) *05_gcd.cpp: Euclid's algorithm recursively and iteratively, traced, and its worst case.*

```
    int best = 0;
    long bestA = 0, bestB = 0;
    for (long a = 1; a <= 100; ++a)
        for (long b = 1; b < a; ++b)
            if (steps(a, b) > best) {
                best = steps(a, b);
                bestA = a;
                bestB = b;
            }
    std::printf("\nAmong all 1 <= b < a <= 100, the most divisions is %d, "
                "first for gcd(%ld, %ld)\n", best, bestA, bestB);

    std::printf("\nConsecutive Fibonacci numbers:\n");
    long x = 1, y = 2;  // F(2), F(3)
    for (int k = 3; k <= 12; ++k) {
        std::printf("  gcd(F(%d), F(%d)) = gcd(%ld, %ld) = %ld "
                    "after %d divisions\n",
                    k + 1, k, x + y, y, gcdRec(x + y, y), steps(x + y, y));
        long z = x + y;
        x = y;
        y = z;
    }
    return 0;
}
```

```
Traced:
  gcd(228, 90): 228 = 2 * 90 + 48
  gcd(90, 48): 90 = 1 * 48 + 42
  gcd(48, 42): 48 = 1 * 42 + 6
  gcd(42, 6): 42 = 7 * 6 + 0
  gcd(6, 0): b = 0, the answer is 6
  gcd(1071, 462): 1071 = 2 * 462 + 147
  gcd(462, 147): 462 = 3 * 147 + 21
  gcd(147, 21): 147 = 7 * 21 + 0
  gcd(21, 0): b = 0, the answer is 21

     a     b   recursive  iterative  divisions
   228    90           6          6          4
  1071   462          21         21          3
    90   228           6          6          5
    17     5           1          1          3
    12     0          12         12          0
   144    89           1          1         10

Among all 1 <= b < a <= 100, the most divisions is 9, first for gcd(89, 55)

Consecutive Fibonacci numbers:
  gcd(F(4), F(3)) = gcd(3, 2) = 1 after 2 divisions
  gcd(F(5), F(4)) = gcd(5, 3) = 1 after 3 divisions
  gcd(F(6), F(5)) = gcd(8, 5) = 1 after 4 divisions
  gcd(F(7), F(6)) = gcd(13, 8) = 1 after 5 divisions
  gcd(F(8), F(7)) = gcd(21, 13) = 1 after 6 divisions
  gcd(F(9), F(8)) = gcd(34, 21) = 1 after 7 divisions
  gcd(F(10), F(9)) = gcd(55, 34) = 1 after 8 divisions
  gcd(F(11), F(10)) = gcd(89, 55) = 1 after 9 divisions
  gcd(F(12), F(11)) = gcd(144, 89) = 1 after 10 divisions
  gcd(F(13), F(12)) = gcd(233, 144) = 1 after 11 divisions
```

Notice the third row of the table: with the arguments the “wrong” way round, gcd(90,228), the first division is $90 = 0 \cdot 228 + 90$, which simply swaps them, and the algorithm continues as before with one extra division. There is no need to sort the arguments first. The row gcd(12,0) = 12 is the base case alone.

10.6.4 How many divisions? Lamé's theorem

The search over all pairs below 100 finds that the most divisions, 9, are needed for gcd(89,55), two consecutive Fibonacci numbers, and the last block of the output shows that gcd$(F(k+1), F(k))$ always takes exactly $k-1$ divisions. That is no accident. Dividing $F(k+1)$ by $F(k)$ gives quotient 1 and remainder $F(k+1) - F(k) = F(k-1)$, so the algorithm walks down the Fibonacci sequence one step per division, the slowest possible descent. In 1844 Gabriel Lamé turned this into the first theorem ever proved about the running time of an algorithm [10]:

> **THEOREM (Lamé, 1844).** If $a > b \geq 1$ and Euclid's algorithm performs k divisions on (a, b), then $a \geq F(k+2)$ and $b \geq F(k+1)$.

Proof by induction on k. If $k = 1$, then $b \geq 1 = F(2)$ and $a > b$ gives $a \geq 2 = F(3)$. If $k \geq 2$, the first division turns (a, b) into (b, r) with $r = a \bmod b \geq 1$ (otherwise it would stop after one division), and on (b, r) the algorithm performs $k-1$ divisions. By the induction hypothesis $b \geq F(k+1)$ and $r \geq F(k)$. Since $a \geq b + r$ (the quotient is at least 1), $a \geq F(k+1) + F(k) = F(k+2)$. ▫

Combine the theorem with Equation (10.8): if k divisions are needed then $b \geq F(k+1) \geq \varphi^{k-1}$, so

$$k \;\leq\; 1 + \log_{\varphi} b \;\approx\; 1 + 4.785 \log_{10} b. \tag{10.12}$$

By Equation (10.12), the number of divisions is at most about five times the number of decimal digits of the smaller number: $O(\log b)$. For numbers below 100 it bounds the count by $1 + \log_{\varphi} 99 \approx 10.5$, and the true worst case is 9. For two 64-bit numbers it is at most 93. Euclid's algorithm is fast enough to be used inside the arithmetic of cryptography on numbers with thousands of digits.

The recursion depth is the number of divisions plus one, so the recursive form uses $O(\log b)$ stack: never more than a hundred frames for 64-bit numbers. This is a recursion you may write without a second thought. Knuth [9] analyzes the *average* number of divisions too; it is about $0.843 \ln b + 1.47$.

10.7 Fast Exponentiation

10.7.1 Subtracting one: n multiplications

To compute m^n for a non-negative integer n, the definition "multiply m by itself n times" gives a loop of n multiplications. There is an equally simple recursive definition:

$$m^0 = 1, \qquad m^n = m \cdot m^{n-1} \quad (n \geq 1) \tag{10.13}$$

Written as code, Equation (10.13) gives a recursive function, `powerNaive` in **Listing 10.7**, with the same shape as factorial: $n+1$ calls, n multiplications, $\Theta(n)$ time and stack. For 2^{10} that is 10 multiplications; for $7^{1{,}000{,}000{,}000}$ it is a billion.

10.7.2 Halving: about $2 \lg n$ multiplications

A much better recursion comes from squaring. If n is even, $m^n = (m^{n/2})^2$; if n is odd, $m^n = (m^{(n-1)/2})^2 \cdot m$. With integer division both cases read

$$m^n = \begin{cases} 1 & \text{if } n = 0, \\ (m^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is even}, \\ (m^{\lfloor n/2 \rfloor})^2 \cdot m & \text{if } n \text{ is odd}. \end{cases} \tag{10.14}$$

The recursive call in Equation (10.14) is made *once*, on half the exponent, and its result is squared; making it twice, as `power(m, n/2) * power(m, n/2)`, would give back all the work saved (Exercise 10.9 asks why). **Pseudocode 10.4** writes it down.

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
```

Trace 3^{13}. Since $13 = 2 \cdot 6 + 1$, $3^{13} = (3^6)^2 \cdot 3$; $3^6 = (3^3)^2$; $3^3 = (3^1)^2 \cdot 3$; $3^1 = (3^0)^2 \cdot 3$; and $3^0 = 1$. On the way back up: $3^1 = 1 \cdot 1 \cdot 3 = 3$, $3^3 = 3 \cdot 3 \cdot 3 = 27$, $3^6 = 27 \cdot 27 = 729$, $3^{13} = 729 \cdot 729 \cdot 3 = 1{,}594{,}323$. Five calls instead of fourteen, as **Figure 10.7** shows.

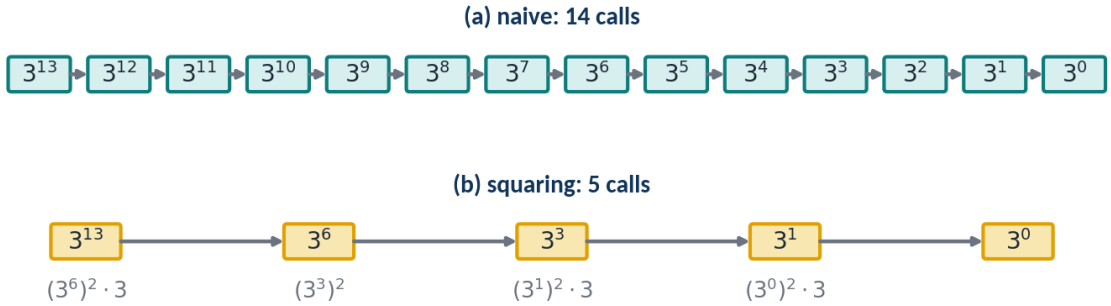


Figure 10.7 Computing 3^{13}. (a) Subtracting one from the exponent: 14 calls. (b) Halving the exponent: 5 calls; below each call, how its result is built from the next.

The exponent halves at every call until it reaches 0. Starting from $n \geq 1$, it takes $\lfloor \lg n \rfloor + 1$ halvings to reach 0 (the number of binary digits of n), so

$$\text{calls}(n) = \lfloor \lg n \rfloor + 2 \quad (n \geq 1), \qquad \text{multiplications} \leq 2(\lfloor \lg n \rfloor + 1). \tag{10.15}$$

Seen in binary, the method is transparent: $13 = 1101_2$, and each level of the recursion handles one binary digit, squaring for every digit and multiplying by m for every 1. Knuth traces the idea back more than two thousand years, to the Indian scholar Piṅgala [9].

10.7.3 Modular exponentiation

The numbers themselves grow too fast to be useful: $7^{1{,}000{,}000{,}000}$ has about 845 million decimal digits. What cryptography, hashing (Chapter 15) and many programming problems actually need is $m^n \bmod p$ for some modulus p. Because $(x \cdot y) \bmod p = ((x \bmod p)(y \bmod p)) \bmod p$, we may reduce after every multiplication, and every intermediate value stays below p. With $p = 1{,}000{,}000{,}007$, a common choice, every product of two reduced numbers is below $10^{18} < 2^{63}$ and fits in a 64-bit integer. RSA, the public-key system used every day in the TLS connections of web browsers, computes exactly such powers with exponents of hundreds of digits; without repeated squaring it would be impossible [3].

Listing 10.7 traces 3^{13}, compares the two recursions on five exact powers, and then computes $7^n \bmod 1{,}000{,}000{,}007$ for exponents up to 10^{18}.

Listing 10.7 `06_power.cpp`: exponentiation by n multiplications and by repeated squaring, with call counts, and modular exponentiation.

```
// 06_power.cpp -- m^n by n multiplications, and by repeated squaring,
// with call counts and modular exponentiation.
#include <cstdio>
typedef unsigned long long ull;

long long calls = 0;

ull powerNaive(ull m, int n) {  // m^n = m * m^(n-1)
    ++calls;
    if (n == 0) return 1;
    return m * powerNaive(m, n - 1);
}

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;
}

ull powTraced(ull m, int n, int depth) {
    if (n == 0) {
        std::printf("%*spow(%llu, 0) = 1\n", 4 * depth, "", m);
        return 1;
    }
    std::printf("%*spow(%llu, %d) needs pow(%llu, %d)\n", 4 * depth, "", m, n, m,
                n / 2);
    ull half = powTraced(m, n / 2, depth + 1);
    ull result = n % 2 == 0 ? half * half : half * half * m;
    if (n % 2 == 0)
        std::printf("%*spow(%llu, %d) = %llu^2 = %llu\n", 4 * depth, "", m, n, half,
                    result);
    else
        std::printf("%*spow(%llu, %d) = %llu^2 * %llu = %llu\n", 4 * depth, "", m,
                    n,
                    half, m, result);
    return result;
}

const ull MOD = 1000000007ULL;  // a prime; every product stays < 2^63
long long mcalls = 0;
```

Listing 10.7 (Continued) *06_power.cpp: exponentiation by n multiplications and by repeated squaring, with call counts, and modular exponentiation.*

```
ull powerNaiveMod(ull m, long long n) {
    ++mcalls;
    if (n == 0) return 1;
    return m % MOD * powerNaiveMod(m, n - 1) % MOD;
}

ull powerFastMod(ull m, long long n) {
    ++mcalls;
    if (n == 0) return 1;
    ull half = powerFastMod(m, n / 2);
    ull sq = half * half % MOD;
    return n % 2 == 0 ? sq : sq * (m % MOD) % MOD;
}

int main() {
    std::printf("Tracing pow(3, 13):\n");
    powTraced(3, 13, 0);

    std::printf("\n m   n            naive  calls            fast  calls\n");
    int cases[][2] = {{2, 10}, {3, 13}, {2, 63}, {3, 40}, {7, 22}};
    for (auto &c : cases) {
        calls = 0;
        ull a = powerNaive(c[0], c[1]);
        long long ca = calls;
        calls = 0;
        ull b = powerFast(c[0], c[1]);
        std::printf("%2d %3d %20llu %6lld %20llu %6lld\n", c[0], c[1], a, ca, b,
                    calls);
    }

    std::printf("\n7^n mod 1000000007\n");
    std::printf("                    n       naive  calls         fast  calls\n");
    long long ns[] = {10, 100, 900, 1000000000LL, 10000000000000000000LL};
    for (long long n : ns) {
        mcalls = 0;
        ull b = powerFastMod(7, n);
        long long cb = mcalls;
        if (n <= 900) {  // deeper naive recursion: see Section 10.12
            mcalls = 0;
            ull a = powerNaiveMod(7, n);
            std::printf("%20lld %11llu %6lld %12llu %6lld\n", n, a, mcalls, b, cb);
        } else {
            std::printf("%20lld %11s %6s %12llu %6lld\n", n, "-", "-", b, cb);
        }
    }
    return 0;
```

Listing 10.7 (Continued) *06_power.cpp: exponentiation by n multiplications and by repeated squaring, with call counts, and modular exponentiation.*

```
}
```

```
Tracing pow(3, 13):
pow(3, 13) needs pow(3, 6)
    pow(3, 6) needs pow(3, 3)
        pow(3, 3) needs pow(3, 1)
            pow(3, 1) needs pow(3, 0)
                pow(3, 0) = 1
            pow(3, 1) = 1^2 * 3 = 3
        pow(3, 3) = 3^2 * 3 = 27
    pow(3, 6) = 27^2 = 729
pow(3, 13) = 729^2 * 3 = 1594323

 m   n            naive  calls            fast  calls
 2  10                1024     11                1024      5
 3  13             1594323     14             1594323      5
 2  63  9223372036854775808     64  9223372036854775808      7
 3  40 12157665459056928801     41 12157665459056928801      7
 7  22  3909821048582988049     23  3909821048582988049      6

7^n mod 1000000007
                   n       naive  calls        fast  calls
                  10   282475249     11   282475249      5
                 100   946501044    101   946501044      8
                 900   928292332    901   928292332     11
          1000000000           -      -   312556845     31
 1000000000000000000           -      -   259616729     61
```

The call counts agree with Equation (10.15) on every row: 5 for $n = 10$ and $n = 13$, 7 for $n = 63$ (six binary digits, $\lfloor \lg 63 \rfloor = 5$), 31 for $n = 10^9$ and 61 for $n = 10^{18}$. The naive method needs $n + 1$ calls and is not even attempted beyond $n = 900$: at a billion calls it would take seconds in C++, and in Java and Python the recursion would crash long before, for the reasons of Section 10.12 (in Python, 900 is just below the default limit of 1,000). The row $2^{63} = 9{,}223{,}372{,}036{,}854{,}775{,}808$ fits only because the program uses *unsigned* 64-bit integers; a signed `long long` stops at $2^{63} - 1$. Python's built-in `pow(7, n, 1000000007)` uses the same method and gives the same results.

10.8 The Towers of Hanoi, Solved Recursively

10.8.1 The recursive idea

Section 5.2 introduced Édouard Lucas's puzzle of 1883 [11]: move a tower of n disks from peg A to peg C, one disk at a time, never placing a disk on a smaller one, using peg B as a helper. Walking through the three-disk solution there, we noticed that moves 1–3 carry the two small disks to B, move 4 carries the big disk to C, and moves 5–7 carry the two small disks from B onto it. Solving three disks is solving two disks twice, with one move in between.

That observation *is* the recursive algorithm. To move n disks from peg `from` to peg `to`, with `via` as the helper:

1. move the $n - 1$ smaller disks from `from` to `via`, using `to` as the helper;
2. move the largest disk, disk n, from `from` to `to`;
3. move the $n - 1$ smaller disks from `via` to `to`, using `from` as the helper.

Figure 10.8 shows the three steps with the $n - 1$ smaller disks drawn as one block. The step that feels like cheating is steps 1 and 3: "move $n - 1$ disks" is the very problem we are trying to solve. But it is a *smaller* instance of it, and that is all recursion needs. While the $n - 1$ disks move, disk n sits at the bottom of a peg and, being the largest, never blocks any move, so the smaller problem really is the same puzzle with one disk fewer. The base case is $n = 0$: nothing to move.

(**Figure 10.8** and Section 5.2 use $n = 1$, one move, as the base case; both are correct, and $n = 0$ makes the code shorter.)

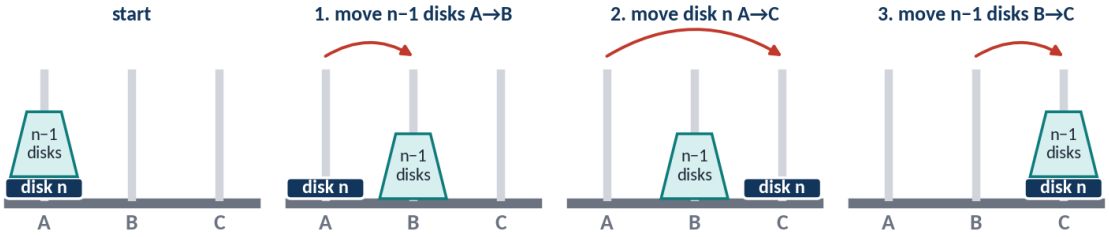


Figure 10.8 The recursive solution of the Towers of Hanoi: move the $n - 1$ smaller disks out of the way, move the largest disk, and move the $n - 1$ disks back on top of it.

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
```

10.8.2 The program

Listing 10.8 is **Pseudocode 10.5** in C++, with one addition that makes the program check itself: each peg is a `std::stack<int>`, the stack of Chapter 5, and `moveDisk` really pops the disk from one peg and pushes it on another, printing `ILLEGAL MOVE` if the disk is not on top of its peg or would land on a smaller disk. The program prints the seven moves for three disks, then solves the puzzle for 1 to 20 disks, checking the number of moves against $2^n - 1$, measuring the deepest recursion, and confirming that all disks end on C.

Listing 10.8 `07_hanoi.cpp`: the recursive Towers of Hanoi, with every move checked on three real stacks.

```
// 07_hanoi.cpp -- the Towers of Hanoi solved by recursion. Every move is also
// played on three real stacks, which check that no rule is ever broken.
#include <cstdio>
#include <stack>

std::stack<int> peg[3];  // peg[0] = A, peg[1] = B, peg[2] = C
long long moves = 0;
int depth = 0, maxDepth = 0;
bool printing = false;

void moveDisk(int disk, char from, char to) {
    std::stack<int> &src = peg[from - 'A'], &dst = peg[to - 'A'];
    if (src.empty() || src.top() != disk || (!dst.empty() && dst.top() < disk)) {
        std::printf("ILLEGAL MOVE of disk %d from %c to %c\n", disk, from, to);
        return;
    }
    src.pop();
    dst.push(disk);
    ++moves;
    if (printing)
        std::printf("%2lld. Move disk %d from rod %c to rod %c\n", moves, disk,
                    from, to);
}

// Move the n smallest disks from rod `from` to rod `to`, with rod `via` as helper.
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;
}

void solve(int n) {
    for (auto &p : peg) p = std::stack<int>();
    for (int d = n; d >= 1; --d) peg[0].push(d);
    moves = 0;
    maxDepth = 0;
    hanoi(n, 'A', 'C', 'B');
}
```

Listing 10.8 (Continued) *07_hanoi.cpp: the recursive Towers of Hanoi, with every move checked on three real stacks.*

```
int main() {
    printing = true;
    std::printf("Three disks, from A to C:\n");
    solve(3);
    printing = false;

    std::printf("\n n      moves    2^n - 1  depth  all on C\n");
    for (int n = 1; n <= 20; ++n) {
        solve(n);
        bool done = (int)peg[2].size() == n && peg[0].empty() && peg[1].empty();
        std::printf("%2d %10lld %10lld %6d  %s\n", n, moves, (1LL << n) - 1,
                    maxDepth,
                    done ? "yes" : "NO");
    }
    unsigned long long legend = ~0ULL;  // 2^64 - 1, all 64 bits set
    std::printf("\nThe legend's 64 disks need %llu moves.\n", legend);
    return 0;
}
```

```
Three disks, from A to C:
 1. Move disk 1 from rod A to rod C
 2. Move disk 2 from rod A to rod B
 3. Move disk 1 from rod C to rod B
 4. Move disk 3 from rod A to rod C
 5. Move disk 1 from rod B to rod A
 6. Move disk 2 from rod B to rod C
 7. Move disk 1 from rod A to rod C

 n      moves    2^n - 1  depth  all on C
 1          1          1      1  yes
 2          3          3      2  yes
 3          7          7      3  yes
 4         15         15      4  yes
 5         31         31      5  yes
 6         63         63      6  yes
 7        127        127      7  yes
 8        255        255      8  yes
 9        511        511      9  yes
10       1023       1023     10  yes
11       2047       2047     11  yes
12       4095       4095     12  yes
13       8191       8191     13  yes
14      16383      16383     14  yes
15      32767      32767     15  yes
16      65535      65535     16  yes
17     131071     131071     17  yes
18     262143     262143     18  yes
19     524287     524287     19  yes
20    1048575    1048575     20  yes

The legend's 64 disks need 18446744073709551615 moves.
```

The seven moves are exactly those of Section 5.2.1 (compare Figure 5.2), and they are the same moves the three-stack loop of Section 5.2.3 made: the recursive and the iterative method find the same unique shortest solution. No `ILLEGAL MOVE` line appears anywhere in 2,097,130 moves (the total for 1 to 20 disks), which is an experimental check of the argument in Section 10.8.1 that the largest disk never gets in the way. The recursion depth equals n: a 20-disk tower needs only 20 frames at a time, even though it makes over a million moves. In Java (**Listing 10.9**) each peg is an `ArrayDeque<Integer>`; in Python (**Listing 10.10**) each peg is a list used as a stack, as in Section 5.16.3.

Listing 10.9 `Hanoi.java`: the recursive Towers of Hanoi in Java.

```java
// Hanoi.java -- the Towers of Hanoi solved by recursion. Every move is also
// played on three real stacks, which check that no rule is ever broken.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.List;

public class Hanoi {
    static List<ArrayDeque<Integer>> peg = new ArrayList<>();  // A, B, C
    static long moves = 0;
    static int depth = 0, maxDepth = 0;
    static boolean printing = false;

    static void moveDisk(int disk, char from, char to) {
        ArrayDeque<Integer> src = peg.get(from - 'A'), dst = peg.get(to - 'A');
        if (src.isEmpty() || src.peek() != disk
                || (!dst.isEmpty() && dst.peek() < disk)) {
            System.out.printf("ILLEGAL MOVE of disk %d from %c to %c%n", disk, from,
                              to);
            return;
        }
        src.pop();
        dst.push(disk);
        ++moves;
        if (printing)
            System.out.printf("%2d. Move disk %d from rod %c to rod %c%n", moves,
                              disk,
                              from, to);
    }

    // Move the n smallest disks from rod `from` to rod `to`, with `via` as helper.
    static 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;
    }
```

Listing 10.9 (Continued) *Hanoi.java: the recursive Towers of Hanoi in Java.*

```
    static void solve(int n) {
        peg.clear();
        for (int i = 0; i < 3; ++i) peg.add(new ArrayDeque<>());
        for (int d = n; d >= 1; --d) peg.get(0).push(d);
        moves = 0;
        maxDepth = 0;
        hanoi(n, 'A', 'C', 'B');
    }

    public static void main(String[] args) {
        printing = true;
        System.out.println("Three disks, from A to C:");
        solve(3);
        printing = false;

        System.out.println("\n n      moves    2^n - 1  depth  all on C");
        for (int n = 1; n <= 20; ++n) {
            solve(n);
            boolean done = peg.get(2).size() == n && peg.get(0).isEmpty()
                           && peg.get(1).isEmpty();
            System.out.printf("%2d %10d %10d %6d  %s%n", n, moves, (1L << n) - 1,
                              maxDepth,
                              done ? "yes" : "NO");
        }
        long legend = ~0L;  // all 64 bits set: 2^64 - 1 unsigned
        System.out.printf("%nThe legend's 64 disks need %s moves.%n",
                          Long.toUnsignedString(legend));
    }
}
```

Listing 10.10 `07_hanoi.py`: the recursive Towers of Hanoi in Python.

```
# 07_hanoi.py -- the Towers of Hanoi solved by recursion. Every move is also
# played on three real stacks, which check that no rule is ever broken.

peg = {"A": [], "B": [], "C": []}  # each peg is a list used as a stack
moves = 0
depth = max_depth = 0
printing = False


def move_disk(disk, frm, to):
    global moves
    src, dst = peg[frm], peg[to]
    if not src or src[-1] != disk or (dst and dst[-1] < disk):
        print(f"ILLEGAL MOVE of disk {disk} from {frm} to {to}")
        return
    dst.append(src.pop())
    moves += 1
    if printing:
        print(f"{moves:2d}. Move disk {disk} from rod {frm} to rod {to}")


def hanoi(n, frm, to, via):
    """Move the n smallest disks from rod frm to rod to, with rod via as helper."""
    global depth, max_depth
    if n == 0:
        return  # base case: nothing to move
    depth += 1
    max_depth = max(max_depth, depth)
    hanoi(n - 1, frm, via, to)  # 1. clear the n-1 smaller disks onto via
    move_disk(n, frm, to)       # 2. move the largest disk
    hanoi(n - 1, via, to, frm)  # 3. bring the n-1 disks back on top of it
    depth -= 1


def solve(n):
    global moves, max_depth
    peg["A"], peg["B"], peg["C"] = list(range(n, 0, -1)), [], []
    moves = max_depth = 0
    hanoi(n, "A", "C", "B")
```

Listing 10.10 (Continued) *07_hanoi.py: the recursive Towers of Hanoi in Python.*

```
def main():
    global printing
    printing = True
    print("Three disks, from A to C:")
    solve(3)
    printing = False

    print("\n n      moves    2^n - 1  depth  all on C")
    for n in range(1, 21):
        solve(n)
        done = len(peg["C"]) == n and not peg["A"] and not peg["B"]
        print(f"{n:2d} {moves:10d} {2**n - 1:10d} {max_depth:6d}  "
              f"{'yes' if done else 'NO'}")
    print(f"\nThe legend's 64 disks need {2**64 - 1} moves.")


if __name__ == "__main__":
    main()
```

Both print the output of **Listing 10.8** line for line.

10.8.3 The order of the calls: a recursion tree

Figure 10.9 draws every call of `hanoi(3, A, C, B)` as a node of a tree; the calls with $n = 0$, which return at once, are left out. Each node has the two recursive calls of lines 3 and 5 as its children, and it makes its own move (line 4) *between* them. The red numbers give the order in which the moves happen: first everything in the left subtree, then the node's own move, then everything in the right subtree. Reading the tree in that order (Chapter 11 calls it an *in-order traversal*) lists the moves of the output: disk 1 A→C, disk 2 A→B, disk 1 C→B, disk 3 A→C, disk 1 B→A, disk 2 B→C, disk 1 A→C.

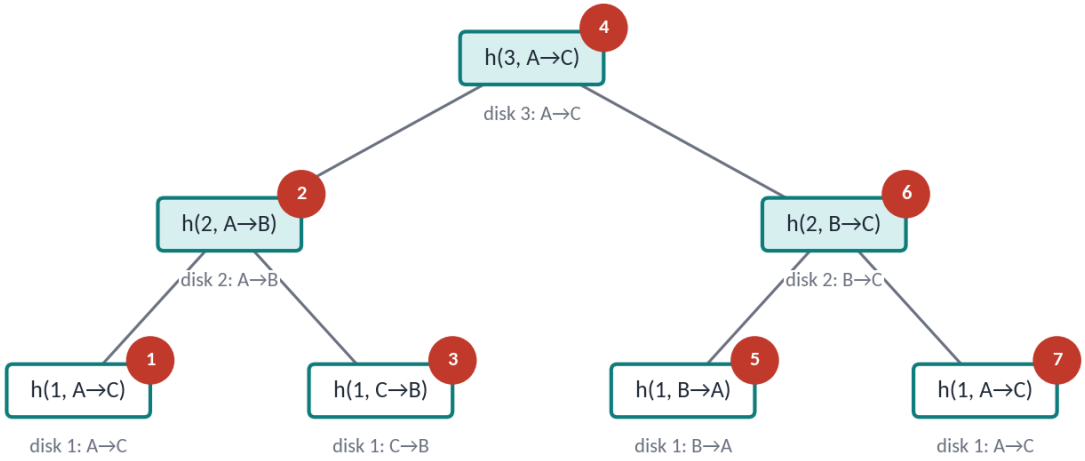


Figure 10.9 The recursion tree of `hanoi(3, A, C, B)`; `h(k, X→Y)` moves k disks from X to Y. Each call makes its own move between the moves of its two children; the red numbers give the order of the moves.

10.8.4 Why exactly $2^n - 1$ moves, and why no fewer

Let $M(n)$ be the number of moves `HANOI` makes for n disks. Line 4 makes one move and lines 3 and 5 make $M(n-1)$ each:

$$M(0) = 0, \qquad M(n) = 2M(n-1) + 1 \quad (n \geq 1). \tag{10.16}$$

This is the recurrence of Section 5.2.2, now read off the program. Its solution is $M(n) = 2^n - 1$, and here is the induction proof that Section 5.2.2 postponed. Base case: $M(0) = 0 = 2^0 - 1$. Inductive step: if $M(n-1) = 2^{n-1} - 1$, then $M(n) = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$. ▫

That the recursive algorithm makes $2^n - 1$ moves does not yet show that no cleverer algorithm could make fewer. Here is the proof that none can. Let $L(n)$ be the smallest number of moves with which *any* sequence of legal moves can transfer n disks from one peg to another. We prove $L(n) \geq 2^n - 1$ by induction. For $n = 0$ it is trivial. For $n \geq 1$, consider any solution and the *first* move of the largest disk, disk n. At that moment disk n is the top disk of its peg, so every smaller disk is elsewhere, and it moves to a peg that holds no smaller disk, so all $n - 1$ smaller disks are stacked on the third peg. Getting them there from the starting peg is a transfer of $n - 1$ disks (disk n lies below them all the while and never matters), which takes at least $L(n-1)$ moves. Now consider the *last* move of disk n, onto the target peg. Afterwards, the $n - 1$ smaller disks, which at that moment are all on the third peg for the same reason, must be carried onto disk n: at least $L(n-1)$ more moves. Together with at least one move of disk n, any solution makes at least $2L(n-1) + 1 \geq 2(2^{n-1} - 1) + 1 = 2^n - 1$ moves. ▫

So the recursive algorithm is *optimal*: its exponential cost belongs to the problem, not to the algorithm. That is the essential difference from naive Fibonacci. Both recursions make an exponential number of calls, and both have the shape "two recursive calls plus a little work". But

naive Fibonacci's calls overlap (its two subproblems share almost everything), so remembering answers removes the waste. The two calls of `HANOI` move *different disks between different pegs*: there is nothing to remember, and every one of the $2^n - 1$ moves is necessary. For the 64 disks of the legend, the program's last line gives the count, 18,446,744,073,709,551,615, about 585 billion years at one move per second (Section 5.2.2), and no algorithm will ever do better. Graham, Knuth and Patashnik [12] open their book *Concrete Mathematics* with this puzzle and use it to introduce recurrences in general.

10.9 Recursion on Arrays and Linked Lists

10.9.1 Two ways to make an array smaller

An array $A[0..n-1]$ is not obviously recursive, but it becomes so as soon as we choose how to make it smaller. There are two standard choices.

- **Peel off one element.** An array of n elements is its first $n-1$ elements followed by its last element (or its first element followed by the rest). The base case is the empty array. This gives recursions with $n+1$ calls and depth n, like factorial.
- **Cut it in half.** An array $A[lo..hi]$ is its left half $A[lo..mid]$ followed by its right half $A[mid+1..hi]$, with $mid = lo + \lfloor (hi - lo)/2 \rfloor$. The base case is one element (or none). This gives recursions of depth $\lceil \lg n \rceil + 1$, like binary search (Section 2.5.5) and merge sort (Section 3.8).

The first part of **Listing 10.11** applies both to a small array of daily loan counts at Pustaka Ceria. `sumArr(A, n)` peels off the last element: the sum of $A[0..n-1]$ is the sum of $A[0..n-2]$ plus $A[n-1]$, and the sum of the empty array is 0. `maxArr(A, lo, hi)` cuts in half: the maximum of the whole is the larger of the maxima of the halves, and the maximum of a single element is that element. `searchArr` is linear search written recursively: look at $A[i]$, and if it is not the key, search from $i+1$. `binSearch` is the recursive binary search of Section 2.5.5. `reverseArr` swaps the two ends and reverses what lies between them. `isPalindrome` asks whether a word reads the same backward: the two end letters must match, and the word between them must be a palindrome; a word of zero or one letter always is.

10.9.2 A linked list is a recursive structure

Section 7.6.2 gave the recursive definition of a list: *a list is either empty, or a first node followed by a list*. Every recursion over a singly linked list follows that definition: the base case is the null pointer, and the recursive case handles the first node and recurses on `p->next`. The second part of **Listing 10.11** collects the most useful ones.

- `length(p)` and `sumList(p)`: 0 for the empty list, otherwise $1 + \text{length}(p.\text{next})$, or $p.\text{data} + \text{sum}(p.\text{next})$.
- `contains(p, key)`: false for the empty list; otherwise true if the first node holds the key, and else the answer for the rest. The `||` stops at the first match.
- `printForward(p)` prints the first node and *then* recurses; `printBackward(p)` recurses *first* and prints afterward. That one change of order prints the list backward, without a `prev` pointer (Chapter 9) and without reversing anything (Section 7.6.6): every call waits on the stack, holding its node, until the calls for all later nodes have printed, exactly like the frames of **Figure 10.2**. The stack plays the role of the missing backward links.
- `copyList(p)` makes an independent copy: a new node holding `p->data` whose `next` is the copy of the rest.

- `removeAll(p, key)` deletes every node holding `key` and returns the new first node: it first cleans the rest of the list, then either deletes the first node (returning the cleaned rest) or relinks it to the cleaned rest. Because the function *returns* the new head, the caller writes `head = removeAll(head, 40);`, and the case "the first node must go" needs no special code; compare the special cases of `deleteKey` in Section 7.6.5.
- `destroy(p)` frees the rest of the list first and then the first node, and counts the nodes freed.

Listing 10.11 `08_arrays_lists.cpp`: recursion over an array (peeling one element or halving) and over a singly linked list (the first node and the rest).

```
// 08_arrays_lists.cpp -- recursion on arrays (a first element and the rest, or two
// halves) and on singly linked lists (a first node and the rest of the list).
#include <cstdio>
#include <cstring>

// ---------- arrays: A[lo..hi] ----------
int sumArr(const int A[], int n) {       // A[0..n-1]
    if (n == 0) return 0;                // the empty array
    return sumArr(A, n - 1) + A[n - 1];  // the first n-1, plus the last
}

int maxArr(const int A[], int lo, int hi) {  // A[lo..hi], lo <= hi
    if (lo == hi) return A[lo];
    int mid = lo + (hi - lo) / 2;
    int left = maxArr(A, lo, mid), right = maxArr(A, mid + 1, hi);
    return left > right ? left : right;
}

int searchArr(const int A[], int n, int i, int key) {  // first index >= i
    if (i == n) return -1;                             // ran off the end: absent
    if (A[i] == key) return i;
    return searchArr(A, n, i + 1, key);
}

int binSearch(const int A[], int lo, int hi, int key) {  // A sorted ascending
    if (lo > hi) return -1;
    int mid = lo + (hi - lo) / 2;
    if (A[mid] == key) return mid;
    if (key < A[mid]) return binSearch(A, lo, mid - 1, key);
    return binSearch(A, mid + 1, hi, key);
}

void reverseArr(int A[], int lo, int hi) {
    if (lo >= hi) return;  // zero or one element left
    int t = A[lo]; A[lo] = A[hi]; A[hi] = t;
    reverseArr(A, lo + 1, hi - 1);
}

bool isPalindrome(const char *s, int lo, int hi) {
    if (lo >= hi) return true;
    if (s[lo] != s[hi]) return false;
    return isPalindrome(s, lo + 1, hi - 1);
}
```

Listing 10.11 (Continued) *08_arrays_lists.cpp: recursion over an array (peeling one element or halving) and over a singly linked list (the first node and the rest).*

```
void printArr(const char *label, const int A[], int n) {
    std::printf("%s", label);
    for (int i = 0; i < n; ++i) std::printf(" %d", A[i]);
    std::printf("\n");
}

// ---------- singly linked lists ----------
struct Node {
    int data;
    Node *next;
};

Node *push(Node *head, int x) { return new Node{x, head}; }

int length(const Node *p) { return p == nullptr ? 0 : 1 + length(p->next); }
int sumList(const Node *p) { return p == nullptr ? 0 : p->data + sumList(p->next); }

bool contains(const Node *p, int key) {
    if (p == nullptr) return false;
    return p->data == key || contains(p->next, key);
}

void printForward(const Node *p) {  // print, then recurse
    if (p == nullptr) return;
    std::printf(" %d", p->data);
    printForward(p->next);
}

void printBackward(const Node *p) {  // recurse, then print
    if (p == nullptr) return;
    printBackward(p->next);
    std::printf(" %d", p->data);
}

Node *copyList(const Node *p) {  // a new node for every node
    if (p == nullptr) return nullptr;
    return new Node{p->data, copyList(p->next)};
}
```

Listing 10.11 (Continued) *08_arrays_lists.cpp: recursion over an array (peeling one element or halving) and over a singly linked list (the first node and the rest).*

```
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;
}

int destroy(Node *p) {  // returns the number freed
    if (p == nullptr) return 0;
    int freed = destroy(p->next);  // free the rest first
    delete p;
    return freed + 1;
}

int main() {
    int loans[] = {12, 7, 31, 4, 25, 18};  // loans per day at Pustaka Ceria
    int n = 6;
    printArr("loans:", loans, n);
    std::printf("sum = %d, max = %d\n", sumArr(loans, n), maxArr(loans, 0, n - 1));
    std::printf("search 4 -> index %d, search 9 -> index %d\n",
                searchArr(loans, n, 0, 4), searchArr(loans, n, 0, 9));
    reverseArr(loans, 0, n - 1);
    printArr("reversed:", loans, n);

    int sorted[] = {4, 7, 12, 18, 25, 31};
    std::printf("binary search 25 -> index %d, 5 -> index %d\n",
                binSearch(sorted, 0, 5, 25), binSearch(sorted, 0, 5, 5));
    const char *words[] = {"racecar", "rotor", "library", "a", ""};
    for (const char *w : words)
        std::printf("\"%s\" is %sa palindrome\n", w,
                    isPalindrome(w, 0, (int)std::strlen(w) - 1) ? "" : "not ");
```

Listing 10.11 (Continued) *08_arrays_lists.cpp: recursion over an array (peeling one element or halving) and over a singly linked list (the first node and the rest).*

```
    Node *head = nullptr;
    int values[] = {50, 40, 30, 40, 20};
    for (int x : values) head = push(head, x);  // pushed at the front
    std::printf("\nlist forward :");
    printForward(head);
    std::printf("\nlist backward:");
    printBackward(head);
    std::printf("\nlength = %d, sum = %d, contains 30: %s, contains 35: %s\n",
                length(head), sumList(head), contains(head, 30) ? "yes" : "no",
                contains(head, 35) ? "yes" : "no");
    Node *copy = copyList(head);
    head = removeAll(head, 40);
    std::printf("after removing 40:");
    printForward(head);
    std::printf("\nthe copy is untouched:");
    printForward(copy);
    std::printf("\nfreed %d + %d nodes\n", destroy(head), destroy(copy));
    return 0;
}
```

```
loans: 12 7 31 4 25 18
sum = 97, max = 31
search 4 -> index 3, search 9 -> index -1
reversed: 18 25 4 31 7 12
binary search 25 -> index 4, 5 -> index -1
"racecar" is a palindrome
"rotor" is a palindrome
"library" is not a palindrome
"a" is a palindrome
"" is a palindrome

list forward : 20 40 30 40 50
list backward: 50 40 30 40 20
length = 5, sum = 180, contains 30: yes, contains 35: no
after removing 40: 20 30 50
the copy is untouched: 20 40 30 40 50
freed 3 + 5 nodes
```

The palindrome test is correct for the empty string as well: with hi $= -1$ the condition `lo >= hi` holds at once. The linked-list part builds 20, 40, 30, 40, 50 by pushing 50, 40, 30, 40, 20 at the front, prints it forward and backward, and removes both 40s while the copy keeps them. Because the program allocates and frees nodes, it was run under Valgrind and under AddressSanitizer with UndefinedBehaviorSanitizer (both introduced in Section 7.11). Valgrind's summary:

```
==PID== HEAP SUMMARY:
==PID==     in use at exit: 0 bytes in 0 blocks
==PID==   total heap usage: 12 allocs, 12 frees, 77,984 bytes allocated
==PID==
==PID== All heap blocks were freed -- no leaks are possible
==PID==
==PID== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

(The 12 allocations are the 10 nodes and two blocks that the C and C++ runtime libraries allocate for themselves.) AddressSanitizer reported nothing either. The Java program `ArraysLists.java` and the Python program `08_arrays_lists.py` print the same output; there `destroy` merely unlinks the nodes and counts them, and the garbage collector frees them.

10.9.3 Costs, and a caution

Every list function above makes one call per node plus one for the null pointer: $\Theta(n)$ time and $\Theta(n)$ stack. That is the caution of Section 7.6.2 again. A recursion over a list of a million nodes needs a million frames, which is too many for the default stacks of all three languages (Section 10.12 has the numbers), while the loop versions of Chapter 7 need none. Recursion over a list is excellent for thinking, for proving and for short lists; for long lists, write the loop, or use an explicit stack as Section 10.13 does for `printBackward`. The halving recursions on arrays have no such problem: a depth of $\lceil \lg n \rceil + 1$ is 31 for a billion elements.

10.10 Recursion Trees

10.10.1 Drawing the calls

We have now drawn several pictures of the calls a recursion makes: a single chain for factorial (**Figure 10.2**), a bushy binary tree for naive Fibonacci (**Figure 10.4**), a chain of halving exponents (**Figure 10.7**) and a complete binary tree for Hanoi (**Figure 10.9**). Such a picture, one node per call, each call's recursive calls as its children, is a *recursion tree*. If we write in every node the work that call does *itself* (not counting its children), the total running time is the sum of all the

numbers in the tree, and the recursion depth, which decides the stack memory, is the height of the tree.

The sum is easiest to compute level by level. That is how Section 3.8.4 found the cost of merge sort, and it is the general method promised there.

10.10.2 Merge sort and Hanoi, level by level

Figure 10.10(a) is the recursion tree of merge sort on n elements, where a call on k elements does ck work to merge (for some constant c) and makes two calls on $k/2$ elements; assume n is a power of two. The root does cn. Level 1 has two nodes of $cn/2$: again cn in total. Level 2 has four nodes of $cn/4$: cn. In general level i has 2^i nodes of $cn/2^i$, total cn, and the leaves are at level $\lg n$, where the subarrays have one element. There are $\lg n + 1$ levels, each costing cn, so

$$T(n) = 2T(n/2) + cn \quad \Rightarrow \quad T(n) = cn(\lg n + 1) = \Theta(n \lg n). \tag{10.17}$$

Figure 10.10(b) is the recursion tree of Hanoi for n disks, where each call makes one move and two calls on $n-1$ disks. Level i has 2^i nodes, each making one move, and the levels run from 0 to $n-1$: the total is $1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1$, which is Equation (10.16) solved again, this time without guessing the answer.

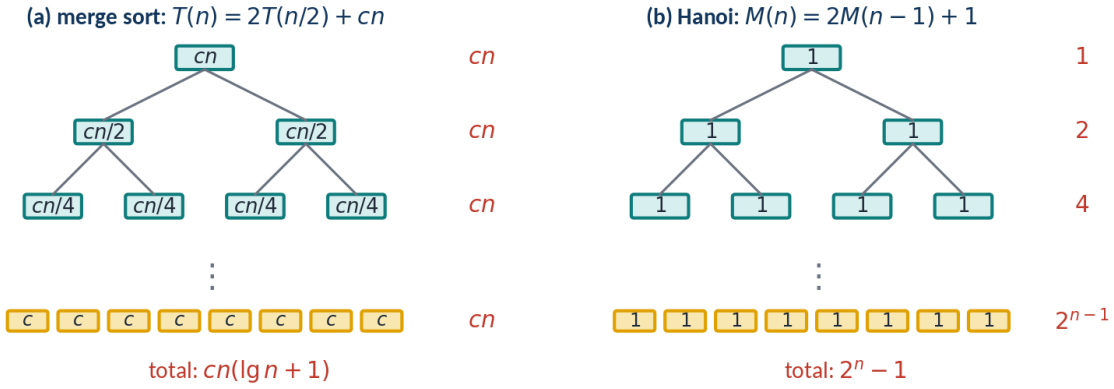


Figure 10.10 Recursion trees with the cost of each level on the right. (a) Merge sort: $\lg n + 1$ levels of cost cn each. (b) Hanoi: level i costs 2^i, and the levels add up to $2^n - 1$.

Why do these two trees, both binary, give such different totals? In merge sort the problem *halves* at each level, so the tree has only $\lg n + 1$ levels, and the work per level stays at cn. In Hanoi the problem shrinks by *one* at each level, so the tree has n levels, and the number of nodes doubles on every one of them. The shape of the recurrence decides everything: how many subproblems, how much smaller, and how much work outside the calls.

10.10.3 A table of recurrences

Table 10.2 collects the recurrences of this chapter and their solutions, each of which has been proved above by unfolding, by a recursion tree or by induction. They cover most of the recursive algorithms in a first course. (Recurrences of the form $T(n) = aT(n/b) + f(n)$ in general are solved by the *master theorem* [3], which a course on algorithm analysis treats in full.)

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))
$T(n) = T(n-1) + T(n-2) + c$	$\Theta(\varphi^n)$	naive Fibonacci (Equation (10.6))

The third row deserves a remark: `maxArr` halves the array but still takes $\Theta(n)$ time, because it must look at every element; the recursion tree has n leaves and $n-1$ internal nodes, each doing constant work. Halving saves time only when one of the halves can be *discarded*, as in binary search, or when the combining work is shared cleverly, as in fast power.

10.11 Tail Recursion and What Compilers Actually Do

10.11.1 Tail calls

A call is a *tail call* when it is the last action of the calling function: the caller returns the callee's result as it is, with nothing left to do afterwards. In

```
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
}
```

`sumPlain` must still add `n` after its recursive call returns, so its frame, holding `n`, has to survive the call. `sumAcc` has done all its work *before* the call: the partial sum travels down in the parameter `acc`, an *accumulator*, and when the call returns there is nothing left to do but return. Its correctness follows from an invariant that plays the role of the induction hypothesis: for every call, `acc` $+(1+2+\cdots+n)$ equals the result we want, which is true for the first call `sumAcc(n, 0)` and is preserved by each call because $(\text{acc}+n)+(1+\cdots+(n-1)) = \text{acc}+(1+\cdots+n)$.

Since the frame of `sumAcc(n, acc)` is never needed again once it calls `sumAcc(n - 1, acc + n)`, the new call could simply *reuse* it: overwrite `n` and `acc` with the new values and jump back to the start of the function. That transformation is *tail-call elimination* (or tail-call optimization), and it turns a tail-recursive function into exactly the loop

```
while (n != 0) { acc = acc + n; n = n - 1; }
return acc;
```

using one frame however large n is. Guy Steele argued in 1977 that a procedure call in tail position should be compiled as a jump, "LAMBDA: the ultimate GOTO" [13], and the programming language Scheme has required it ever since. C, C++, Java and Python do not require it. What their implementations actually do is a question of fact, and the rest of this section answers it by experiment.

10.11.2 What GCC does: an experiment

Listing 10.12 contains `sumPlain` and `sumAcc` and a third function, `sumKeep`, that is written so that no compiler can remove its recursion: it stores `n` in a `volatile` local variable and reads it back *after* the recursive call returns, so its frame genuinely has to survive the call. The program takes the function's name and n on the command line.

Listing 10.12 `09_tail.cpp`: a non-tail recursion, a tail recursion, and a recursion whose frames must survive, selectable from the command line.

```
// 09_tail.cpp -- three ways to add 1 + 2 + ... + n recursively. Run it as
//   ./09_tail plain|acc|keep n
// and compare builds made with -O0 and with -O2 (Section 10.11).
#include <cstdio>
#include <cstdlib>
#include <cstring>

long long sumPlain(long long n) {  // NOT a tail call: the addition
    if (n == 0) return 0;          // happens after the call returns
    return n + sumPlain(n - 1);
}

long long sumAcc(long long n, long long acc) {  // a tail call: the answer so far
    if (n == 0) return acc;                     // travels down in acc
    return sumAcc(n - 1, acc + n);
}

long long sumKeep(long long n) {  // the frame must survive the call:
    volatile long long mine = n;  // `mine` is read after it returns
    if (n == 0) return 0;
    long long rest = sumKeep(n - 1);
    return mine + rest;
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        std::printf("usage: %s plain|acc|keep n\n", argv[0]);
        return 1;
    }
    long long n = std::atoll(argv[2]);
    long long s;
    if (std::strcmp(argv[1], "plain") == 0) s = sumPlain(n);
    else if (std::strcmp(argv[1], "acc") == 0) s = sumAcc(n, 0);
    else s = sumKeep(n);
    std::printf("%s(%lld) = %lld\n", argv[1], n, s);
    return 0;
}
```

We built it twice, without optimization (`-O0`) and with the usual optimization (`-O2`), and ran each function with $n = 100{,}000$ and $n = 10{,}000{,}000$ on the Linux machine of this book, whose default stack is 8 MB:

```
$ g++ -std=c++17 -O0 -Wall -Wextra -o tail_O0 09_tail.cpp
$ g++ -std=c++17 -O2 -Wall -Wextra -o tail_O2 09_tail.cpp
$ ./tail_O0 plain 100000
plain(100000) = 5000050000
$ ./tail_O0 plain 10000000
Segmentation fault
$ ./tail_O0 acc 100000
acc(100000) = 5000050000
$ ./tail_O0 acc 10000000
Segmentation fault
$ ./tail_O0 keep 100000
keep(100000) = 5000050000
$ ./tail_O0 keep 10000000
Segmentation fault
$ ./tail_O2 plain 100000
plain(100000) = 5000050000
$ ./tail_O2 plain 10000000
plain(10000000) = 50000005000000
$ ./tail_O2 acc 100000
acc(100000) = 5000050000
$ ./tail_O2 acc 10000000
acc(10000000) = 50000005000000
$ ./tail_O2 keep 100000
keep(100000) = 5000050000
$ ./tail_O2 keep 10000000
Segmentation fault
```

At `-O0` every function crashes at ten million: the compiler translates each call literally, one frame per call, and ten million frames do not fit in 8 MB. At `-O2`, `sumAcc` runs to ten million without trouble, as tail-call elimination predicts. The surprise is that `sumPlain`, which is *not* tail recursive, also succeeds. And `sumKeep` still crashes. The assembly language that GCC generates explains all three results. (Assembly is the processor's own instruction set, written in words; you do not need to know it to follow the argument, only to notice whether the word `call` appears.) Here is `sumAcc` at `-O0`, produced with `g++ -S -masm=intel` and shown without its bookkeeping directives:

```
sumAcc(long long, long long):
    push rbp
    mov rbp, rsp
    sub rsp, 16
    mov QWORD PTR -8[rbp], rdi
    mov QWORD PTR -16[rbp], rsi
    cmp QWORD PTR -8[rbp], 0
    jne .L5
    mov rax, QWORD PTR -16[rbp]
    jmp .L6
.L5:
    mov rdx, QWORD PTR -16[rbp]
    mov rax, QWORD PTR -8[rbp]
    add rdx, rax
    mov rax, QWORD PTR -8[rbp]
    sub rax, 1
    mov rsi, rdx
    mov rdi, rax
    call sumAcc(long long, long long)
    nop
.L6:
    leave
    ret
```

The line `call sumAcc(long long, long long)` is a genuine recursive call: it pushes a return address and the called function then pushes its own frame with `push rbp` and `sub rsp, 16`. Now the same function at `-O2`:

```
sumAcc(long long, long long):
    mov rax, rsi
    test rdi, rdi
    je .L14
    lea rdx, -1[rdi]
    test dil, 1
    je .L11
    add rax, rdi
    mov rdi, rdx
    test rdx, rdx
    je .L25
.L11:
    lea rax, -1[rax+rdi*2]
    sub rdi, 2
    jne .L11
.L14:
    ret
.L25:
    ret
```

There is no `call` at all. The recursion has become the loop at label `.L11`, which even handles two values of `n` per iteration (it adds $n + (n-1) = 2n - 1$ with one `lea` instruction and subtracts 2), with a small preamble for odd n. And `sumPlain` at `-O2`:

```
sumPlain(long long):
    xor eax, eax
    test rdi, rdi
    je .L1
.L2:
    add rax, rdi
    sub rdi, 1
    jne .L2
.L1:
    ret
```

This is the loop `acc += n; n -= 1;` in three instructions. GCC recognized that the pending operation of `sumPlain` is an addition, which is associative, introduced an accumulator by itself, and then eliminated the resulting tail call. GCC's optimizer does this for additions and multiplications that wait on a single recursive call, which is also why `factRec` and the recursive `powerNaive` never came near a stack overflow at `-O2`, and why the naive Fibonacci of **Table 10.1** was faster in C++ than one might expect (one of its two calls becomes a loop; the other remains a genuine call). The option responsible is `-foptimize-sibling-calls`, "Optimize sibling and tail recursive calls", which GCC enables at `-O2`, `-O3` and `-Os` but not at `-O0` or `-O1` [14]. `sumKeep` cannot be transformed: its frame holds a `volatile` value that must be read after the call. GCC inlines three levels of the recursion into each call to save time, but a real `call` remains, and so does the frame growth.

Figure 10.11 contrasts the two situations: at `-O0` the stack holds one frame for every pending call; after tail-call elimination, one frame is reused for the whole computation.

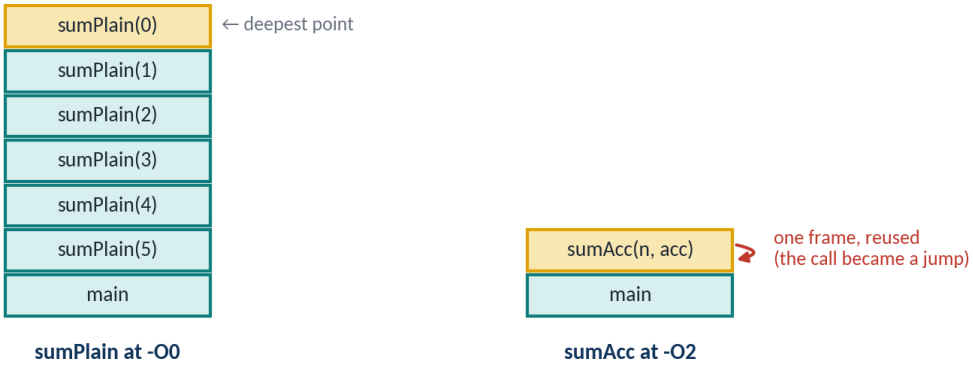


Figure 10.11 Left: without optimization, `sumPlain(5)` has six frames of its own on the stack at the deepest point. Right: at `-O2` the tail call of `sumAcc` is compiled as a jump, and a single frame serves the whole recursion (GCC does the same for `sumPlain` after introducing an accumulator).

10.11.3 Why you must not rely on it in C++

The experiment shows what GCC 13.3 did on one machine. It is *not* a promise. The C++ standard says nothing about tail calls, so the transformation is an optimization that the compiler may or may not apply. It disappears at `-O0`, which is how most people build while debugging; it can disappear when a function has a local object with a destructor (the destructor must run *after* the call, so the call is no longer the last action), when the arguments are passed on the stack in certain calling conventions, when the address of a local variable escapes, or simply in a different compiler or version. A program whose correctness depends on tail-call elimination may therefore work in the release build and crash in the debug build, which is the least pleasant kind of bug.

For code that must rely on it, both GCC (since version 15) and Clang offer an explicit attribute, `[[gnu::musttail]]` or `[[clang::musttail]]`, placed on a `return` statement: "It asserts that the call must be a tail call that does not allocate extra stack space, so it is safe to use tail recursion to implement long-running loops. If the compiler cannot generate a `musttail` tail call it reports an error" [15]. The current GCC series as of September 2026 is GCC 16, whose first release, 16.1, appeared in April 2026 [16]. The attribute is not part of standard C++, so this book does not use it; when a recursion may be deep, we write the loop.

10.11.4 The JVM and CPython do not eliminate tail calls

The Java HotSpot virtual machine does not eliminate tail calls, and neither does its JIT compiler. **Listing 10.13** runs `sumPlain` and the tail-recursive `sumAcc` with growing n and catches the error.

Listing 10.13 `TailCalls.java`: in Java a tail-recursive method uses one frame per call, exactly like any other.

```java
// TailCalls.java -- the JVM does not eliminate tail calls: a tail-recursive method
// still uses one stack frame per call, exactly like a non-tail-recursive one.
public class TailCalls {
    static long sumPlain(long n) {  // not a tail call
        if (n == 0) return 0;
        return n + sumPlain(n - 1);
    }

    static long sumAcc(long n, long acc) {  // a tail call
        if (n == 0) return acc;
        return sumAcc(n - 1, acc + n);
    }

    static long sumLoop(long n) {  // what a tail call "means"
        long acc = 0;
        while (n != 0) {    // sumAcc(n, acc) becomes
            acc = acc + n;  //   acc = acc + n; n = n - 1;
            n = n - 1;      // and start again from the top
        }
        return acc;
    }

    static String attempt(boolean tail, long n) {
        try {
            return String.valueOf(tail ? sumAcc(n, 0) : sumPlain(n));
        } catch (StackOverflowError e) {
            return "StackOverflowError";
        }
    }

    public static void main(String[] args) {
        System.out.printf("%10s %22s %22s %16s%n", "n", "sumPlain", "sumAcc",
                          "sumLoop");
        for (long n : new long[] {1000, 10000, 1000000, 10000000})
            System.out.printf("%10d %22s %22s %16d%n", n, attempt(false, n),
                              attempt(true, n), sumLoop(n));
    }
}
```

```
         n              sumPlain               sumAcc          sumLoop
      1000                 500500               500500           500500
     10000               50005000             50005000         50005000
   1000000    StackOverflowError   StackOverflowError     500000500000
  10000000    StackOverflowError   StackOverflowError   50000005000000
```

Both methods fail at the same n; being tail recursive makes no difference. The method `sumLoop` shows what the tail call "means": the parameters become variables that are overwritten, and the call becomes a jump to the top of the loop. CPython behaves the same way, deliberately. Guido van Rossum, Python's creator, explained in 2009 that he would not add tail-call elimination, chiefly because it destroys the stack traces that Python programmers rely on for debugging, and because it would encourage a style of programming foreign to the language [17]. **Listing 10.14** makes the same experiment.

Listing 10.14 `09_tail.py`: in Python a tail-recursive function hits the recursion limit like any other.

```python
# 09_tail.py -- CPython does not eliminate tail calls: a tail-recursive function
# still uses one frame per call and hits the recursion limit like any other.
import sys


def sum_plain(n):  # not a tail call
    if n == 0:
        return 0
    return n + sum_plain(n - 1)


def sum_acc(n, acc):  # a tail call
    if n == 0:
        return acc
    return sum_acc(n - 1, acc + n)


def sum_loop(n):  # what a tail call "means"
    acc = 0
    while n != 0:                # sum_acc(n, acc) becomes
        acc, n = acc + n, n - 1  #   acc = acc + n; n = n - 1
    return acc                   # and start again from the top


def attempt(fn, n):
    try:
        return str(fn(n))
    except RecursionError:
        return "RecursionError"


def main():
    print(f"recursion limit: {sys.getrecursionlimit()}")
    print(f"{'n':>10s} {'sum_plain':>16s} {'sum_acc':>16s} {'sum_loop':>16s}")
    for n in [900, 1000, 10000, 1000000]:
        print(f"{n:10d} {attempt(sum_plain, n):>16s} "
              f"{attempt(lambda k: sum_acc(k, 0), n):>16s} {sum_loop(n):16d}")


if __name__ == "__main__":
    main()
```

```
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
```

The rule is simple. In Java and Python, a tail-recursive function is as deep as its input; if you want a loop, write the loop. In C++ with optimization, GCC and Clang *usually* turn tail recursion (and simple accumulations) into loops; enjoy it, but do not depend on it.

10.12 How Deep Can Recursion Go? Limits in C++, Java and Python

10.12.1 C++: the size of the stack

In C++ the depth limit is simply the size of the thread's stack divided by the size of a frame. If the stack has S bytes and each call occupies a frame of s bytes, a recursion can go about

$$d_{\max} \approx \frac{S}{s} \tag{10.18}$$

levels deep before it overflows. On Linux the stack size of the main thread is set by the shell's limit, printed by `ulimit -s` in kilobytes; it is 8,192 KB (8 MB) by default. On Windows it is fixed by the linker when the program is built, 1 MB by default for Microsoft's linker [18] and for MinGW's. On macOS the main thread's stack is 8 MB. Threads created by a program usually get less, often 512 KB to 8 MB depending on the system.

Listing 10.15 measures both numbers of Equation (10.18) on a real machine. The function `probe` recurses n levels with a frame of realistic size (it holds a 64-byte `volatile` array and makes a non-tail call, so the optimizer can remove neither). It records the address of its local array at depth 0 and at depth 1,000; since the stack grows toward lower addresses, the difference divided by 1,000 is the average frame size.

Listing 10.15 `10_depth.cpp`: measuring the size of a stack frame and recursing to a chosen depth.

```
// 10_depth.cpp -- how deep can a C++ recursion go? Run it as  ./10_depth n
// It measures the size of one stack frame, then recurses n levels deep.
#include <cstdint>
#include <cstdio>
#include <cstdlib>

const int FAR = 1000;
std::uintptr_t top = 0, far = 0;  // a local's address at depth 0 and FAR

long long probe(long long n, int depth) {
    volatile char scratch[64];  // a frame of realistic size
    scratch[0] = (char)n;
    if (depth == 0) top = (std::uintptr_t)&scratch[0];
    if (depth == FAR) far = (std::uintptr_t)&scratch[0];
    if (n == 0) return 0;
    return 1 + probe(n - 1, depth + 1) + scratch[0] * 0;  // not a tail call
}

int main(int argc, char *argv[]) {
    long long n = argc > 1 ? std::atoll(argv[1]) : FAR;
    long long reached = probe(n, 0);
    // the stack grows toward lower addresses on the usual machines
    double frame = n >= FAR ? (double)(top - far) / FAR : 0.0;
    std::printf("one frame = %.0f bytes; reached depth %lld, using about %.1f MB\n",
                frame, reached, frame * reached / (1024.0 * 1024.0));
    return 0;
}
```

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o depth 10_depth.cpp
$ ulimit -s
8192
$ ./depth 1000
one frame = 72 bytes; reached depth 1000, using about 0.1 MB
$ ./depth 100000
one frame = 72 bytes; reached depth 100000, using about 6.9 MB
$ ./depth 116000
one frame = 72 bytes; reached depth 116000, using about 8.0 MB
$ ./depth 117000
Segmentation fault
$ ./depth 1000000
Segmentation fault
$ ulimit -s 262144
$ ./depth 1000000
one frame = 72 bytes; reached depth 1000000, using about 68.7 MB
$ g++ -std=c++17 -O0 -Wall -Wextra -o depth0 10_depth.cpp
$ ./depth0 100000
Segmentation fault
$ ./depth0 1000
one frame = 112 bytes; reached depth 1000, using about 0.1 MB
```

With `-O2` each frame takes 72 bytes, and Equation (10.18) predicts 8,388,608/72 ≈ 116,508 levels. The runs agree to the thousand: 116,000 levels succeed, 117,000 crash. Raising the limit with `ulimit -s 262144` (256 MB) lets a million levels through, using about 69 MB. Built with `-O0`, the same function uses 112 bytes per frame, and 100,000 levels, which were fine at `-O2`, already crash: the depth limit of a C++ program depends on its optimization level.

The stack can also be enlarged from inside the program's build, which is the portable choice for programs you distribute:

- **Windows (MSVC):** link with `/STACK:268435456` for a 256 MB stack [18]; **Windows (MinGW g++):** `g++ ... -Wl,--stack,268435456`.
- **Linux and macOS:** run the deep recursion in a thread whose stack size is set explicitly with `pthread_attr_setstacksize` (the standard `std::thread` offers no way to choose it); or ask users to raise `ulimit -s`.

These are remedies for recursions that are deep *by nature*, such as a recursive walk of a very deep structure. For a recursion that is deep only because it peels one element at a time from a long list, the better remedy is the loop.

10.12.2 Java: `-Xss`, and threads with a stack of their own

The Java virtual machine gives each thread a stack of a fixed size, set by the option `-Xss`. For JDK 25 the default is 1,024 KB on Linux and macOS on x64, 2,048 KB on AArch64 (for example Apple silicon), and system-dependent on Windows [19]. A thread created in the program can ask for a different size: `Thread.ofPlatform().stackSize(bytes)`, available since Java 21, or the older constructor `Thread(group, task, name, stackSize)`. The documentation warns that the virtual machine is free to treat the value as a suggestion [20]. **Listing 10.16** measures the depth at which the main thread overflows, three times, and then runs `sumPlain(1000000)` in a thread with a 256 MB stack.

Listing 10.16 `Depth.java`: the recursion depth reachable in Java's main thread, and a deep recursion in a thread with a larger stack.

```java
// Depth.java -- how deep can a Java recursion go? First in the main thread,
// then in a thread created with a much larger stack.
public class Depth {
    static int depth;

    static void dive() {  // recurses until the stack is full
        ++depth;
        dive();
    }

    static int measure() {
        depth = 0;
        try {
            dive();
        } catch (StackOverflowError e) {
            // the stack is unwound when the error propagates; we may continue
        }
        return depth;
    }

    static long sumPlain(long n) { return n == 0 ? 0 : n + sumPlain(n - 1); }

    public static void main(String[] args) throws InterruptedException {
        for (int trial = 1; trial <= 3; ++trial)
            System.out.printf("main thread, trial %d: "
                              + "StackOverflowError at depth %d%n",
                              trial, measure());

        long[] result = new long[1];
        Runnable deep = () -> result[0] = sumPlain(1000000);
        Thread t = Thread.ofPlatform().stackSize(256L * 1024 * 1024)   // 256 MB
                         .unstarted(deep);
        t.start();
        t.join();
        System.out.printf("in a big thread, sumPlain(1000000) = %d%n", result[0]);
    }
}
```

```
main thread, trial 1: StackOverflowError at depth 10730
main thread, trial 2: StackOverflowError at depth 23608
main thread, trial 3: StackOverflowError at depth 23608
in a big thread, sumPlain(1000000) = 500000500000
```

The three trials do not agree, and the disagreement is instructive. The first trial runs in the interpreter, whose frames are large, and reaches 10,730 levels. By the second trial the JIT compiler has compiled `dive`, whose compiled frames are about 44 bytes, and the depth more than doubles. Across several runs of this program we observed values from 10,730 to 59,020 for the same code: in Java the maximum depth is not a fixed number, and a program that recurses 20,000 levels deep may work on one run and fail on the next. The thread with a 256 MB stack computes the million-level `sumPlain` without complaint. `StackOverflowError` can be caught, as `measure` does, because the stack has already been unwound by the time the `catch` block runs; but catching it is a diagnostic tool, not a design.

10.12.3 Python: the recursion limit

CPython limits recursion by counting, not by measuring memory. The function `sys.getrecursionlimit()` returns the limit, 1,000 by default, and `sys.setrecursionlimit(limit)` changes it; the documentation explains that “this limit prevents infinite recursion from causing an overflow of the C stack and crashing Python”, and warns that “a too-high limit can lead to a crash” [21]. The protection has changed in recent versions. Since Python 3.11 a call from Python code to a Python function no longer uses the C stack of the

interpreter: “Most Python function calls now consume no C stack space … This also means recursive functions can recurse significantly deeper (if the user increases the recursion limit)” [22]. Since Python 3.12 the recursion limit “applies only to Python code”; built-in functions are protected by a separate mechanism [23]. **Listing 10.17** measures the default, raises the limit, and finally runs a million-level recursion in a thread with a larger stack.

Listing 10.17 `10_depth.py`: Python’s recursion limit, raising it, and a deep recursion in a thread with a larger stack.

```
# 10_depth.py -- how deep can a Python recursion go? The recursion limit, how to
# raise it, and a thread with a bigger stack for the cases that need one.
import sys
import threading

depth = 0


def dive():  # recurses until Python stops it
    global depth
    depth += 1
    dive()


def measure():
    global depth
    depth = 0
    try:
        dive()
    except RecursionError:
        pass  # the frames are unwound; we may continue
    return depth


def sum_plain(n):
    return 0 if n == 0 else n + sum_plain(n - 1)


def main():
    print(f"Python {sys.version_info.major}.{sys.version_info.minor}")
    print(f"default limit {sys.getrecursionlimit()}: "
          f"RecursionError at depth {measure()}")

    sys.setrecursionlimit(200_000)
    print(f"limit {sys.getrecursionlimit()}: RecursionError at depth {measure()}")
    print(f"sum_plain(150000) = {sum_plain(150_000)}")
```

Listing 10.17 (Continued) *10_depth.py: Python's recursion limit, raising it, and a deep recursion in a thread with a larger stack.*

```
    # Deeper still: raise the limit and run the work in a thread with a large
    # stack, so that the C stack cannot run out first on any Python version.
    sys.setrecursionlimit(1_100_000)
    threading.stack_size(256 * 1024 * 1024)  # 256 MB for new threads
    result = []
    t = threading.Thread(target=lambda: result.append(sum_plain(1_000_000)))
    t.start()
    t.join()
    print(f"in a big thread, sum_plain(1000000) = {result[0]}")


if __name__ == "__main__":
    main()
```

```
Python 3.11
default limit 1000: RecursionError at depth 997
limit 200000: RecursionError at depth 199997
sum_plain(150000) = 11250075000
in a big thread, sum_plain(1000000) = 500000500000
```

The default limit stops the recursion at depth 997 rather than 1,000, because three frames (the module, `main` and `measure`) are already on the stack. With the limit raised to 200,000, a recursion 150,000 levels deep succeeds; on Python 3.11 and later that works even in the main thread, because pure Python calls keep their frames in heap memory. The last part raises the limit above a million and uses `threading.stack_size` to give a new thread a 256 MB stack, which protects the program on any version and any platform in case some C code in the call chain does use the C stack. Deep recursion in Python is also slow: each call costs tens of nanoseconds, and a million-level recursion takes a noticeable fraction of a second.

10.12.4 The limits at a glance

Table 10.3 summarizes the three languages.

Table 10.3 Recursion depth limits in C++, Java and Python, as measured on the machine of this book in September 2026.

	C++	Java	Python
Default limit	stack size: 8 MB on Linux and macOS main threads, 1 MB on Windows	thread stack: 1,024 KB (Linux/macOS x64), 2,048 KB (AArch64)	1,000 nested calls
Measured here	116,000 levels of a 72-byte frame (`-O2`)	10,730 to 59,020 levels, varying with the JIT	997 levels
What happens	segmentation fault (Windows: stack overflow exception)	`StackOverflowError`	`RecursionError`
How to raise it	`ulimit -s`; linker `/STACK` or `-Wl,--stack`; thread with `pthread_attr_setstacksize`	`java -Xss512m`; `Thread.ofPlatform().stackSize(...)`	`sys.setrecursionlimit`; `threading.stack_size` for a new thread
Tail calls eliminated?	often at `-O2`, never guaranteed; `[[gnu::musttail]]`/`[[clang::musttail]]` to insist	no	no

10.13 From Recursion to Iteration with an Explicit Stack

10.13.1 When to convert

A recursion should become a loop when its depth can be large, meaning proportional to the size of the input rather than to its logarithm, and the input may be large: a recursion over a linked list of a million nodes, a walk over a structure nested to an unknown depth that comes from outside the program (a file, a web page, a user), or a naive recursion that would also be exponential. It may stay a recursion when its depth is logarithmic (binary search, fast power, Euclid, merge sort, and quicksort once it recurses on the smaller part first), or bounded by something small and known (the 20 disks of a Hanoi tower, the height of a balanced tree), and when the recursive version is clearly easier to get right. **Table 10.4** gives the decisions for this chapter's algorithms.

Table 10.4 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

10.13.2 The general method

Every recursion can be turned into a loop with a stack of our own, because that is exactly what the machine does to run it. The recipe mirrors Section 10.3:

1. Define a `Frame` record holding everything one call needs to remember: its parameters, the local variables that must survive a recursive call, and a *stage* (or program counter) saying where in the function body the call must continue when a recursive call returns.
2. Replace the first call by pushing its frame on an explicit stack, a `std::vector`, `ArrayDeque` or Python list.
3. Loop while the stack is not empty: look at the top frame and continue it from its stage. A base case pops its frame. A recursive call records the next stage in the current frame and pushes a new frame.
4. Returned values, if any, are passed down through a variable or through a field of the frame below.

The explicit stack lives in the heap, so its size is limited by the computer's memory, gigabytes rather than megabytes. And we may store less than the machine would: only the fields we really need, not the return address and saved registers of a real frame.

10.13.3 Hanoi with a stack of frames

For `HANOI` (**Pseudocode 10.5**) a frame needs n, the three peg names and a stage: stage 0 means "just called: make the first recursive call", stage 1 means "the first recursive call has returned: make the move and the second call". The second call is a tail call: once the move is made, nothing in the frame is needed any more. So, exactly as tail-call elimination does, we *pop* the frame before pushing the frame of the second call. That keeps the explicit stack as short as the real recursion's depth. The first part of **Listing 10.18** implements this; **Figure 10.12** shows the stack, drawn as triples (n, from and to, stage), for $n = 2$.

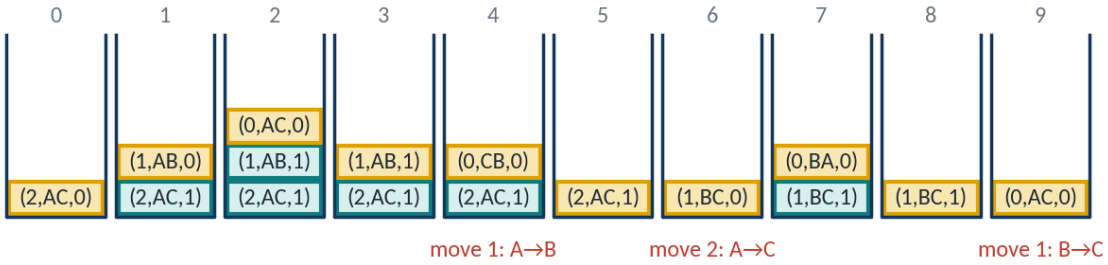


Figure 10.12 The explicit stack while `hanoiIter(2)` runs; each frame shows (n, source and target peg, stage), and the top frame is highlighted. Each column is the stack after one pass of the loop; the moves appear below the pass that makes them.

Follow the columns of **Figure 10.12**. The frame (2, AC, 0) is pushed first. Pass 1 sets its stage to 1 and pushes the first recursive call, (1, AB, 0), which moves one disk from A to B. Pass 2 does the same for it and pushes (0, AC, 0), a base case, which pass 3 pops. In pass 4 the frame (1, AB, 1) is on top at stage 1: it is popped, it makes its move (disk 1 from A to B), and it pushes its second call (0, CB, 0). And so on: after pass 9 the stack is empty and all three moves have been made.

10.13.4 A long list walked backward

The second part of **Listing 10.18** walks a list of 1,000,000 nodes backward. The recursive `printBackward` of Section 10.9 would need a million frames, far more than the 116,000 that fit on the default 8 MB stack. With an explicit stack we push a pointer to every node while walking forward, and then pop them, which yields the nodes in reverse order. The stack is a `vector` of a million pointers, 8 MB in the heap, which is no problem at all.

Listing 10.18 `11_explicit_stack.cpp`: the Towers of Hanoi with an explicit stack of frames, and a million-node list visited backward.

```cpp
// 11_explicit_stack.cpp -- recursion replaced by a stack that we manage ourselves:
// the Towers of Hanoi with a stack of frames, and a backward walk of a long list.
#include <cstdio>
#include <vector>
using namespace std;

struct Frame {  // everything one call of hanoi() needs to remember
    int n;
    char from, to, via;
    int stage;  // 0: just called, 1: first half done
};

long long hanoiIter(int n, bool print, size_t &maxStack) {
    vector<Frame> st;  // our own call stack
    long long moves = 0;
    st.push_back({n, 'A', 'C', 'B', 0});
    maxStack = 1;
    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();
    }
    return moves;
}

struct Node {
    int data;
    Node *next;
};
```

Listing 10.18 (Continued) *11_explicit_stack.cpp: the Towers of Hanoi with an explicit stack of frames, and a million-node list visited backward.*

```
int main() {
    size_t maxStack;
    printf("Three disks, with an explicit stack:\n");
    hanoiIter(3, true, maxStack);
    for (int n : {10, 16, 20}) {
        long long m = hanoiIter(n, false, maxStack);
        printf("n = %2d: %7lld moves, at most %zu frames on our stack\n", n, m,
               maxStack);
    }

    const int N = 1000000;  // far too deep for a recursive walk (Section 10.12)
    Node *head = nullptr;
    for (int i = N; i >= 1; --i) head = new Node{i, head};  // 1, 2, ..., N
    vector<const Node *> st;
    for (const Node *p = head; p != nullptr; p = p->next) st.push_back(p);
    printf("\nThe list 1..%d visited backward starts with:", N);
    for (int k = 0; k < 5; ++k) {
        printf(" %d", st.back()->data);
        st.pop_back();
    }
    printf("\n");
    while (head != nullptr) {  // release the list, iteratively as well
        Node *next = head->next;
        delete head;
        head = next;
    }
    return 0;
}
```

```
Three disks, with an explicit stack:
 1. Move disk 1 from rod A to rod C
 2. Move disk 2 from rod A to rod B
 3. Move disk 1 from rod C to rod B
 4. Move disk 3 from rod A to rod C
 5. Move disk 1 from rod B to rod A
 6. Move disk 2 from rod B to rod C
 7. Move disk 1 from rod A to rod C
n = 10:    1023 moves, at most 11 frames on our stack
n = 16:   65535 moves, at most 17 frames on our stack
n = 20: 1048575 moves, at most 21 frames on our stack

The list 1..1000000 visited backward starts with: 1000000 999999 999998 999997 999996
```

The moves for three disks are identical to those of **Listing 10.8**, and for n disks the explicit stack never holds more than $n + 1$ frames, the same as the recursion's depth plus its base-case frame. The program was also checked with AddressSanitizer, which found no errors. The Java and Python versions, `ExplicitStack.java` and `11_explicit_stack.py`, print the same lines; in Python the explicit stack is an ordinary list, and the million-node walk that would need `sys.setrecursionlimit` as a recursion runs as a loop within the default limit.

> **KEY INSIGHT:** Recursion does not add power that loops lack; any recursion can be replaced by a loop and a stack, and the machine itself does exactly that. What recursion adds is *clarity*: the machine keeps the stack for you, correctly, every time. Convert to an explicit stack when the depth may be large, and keep the recursion when it is not.

10.14 Profitina in Practice

Profitina is an AI visibility (GEO) platform: it measures how ready a brand's website is to be cited by AI answer engines, from signals such as content that machines can extract, structured data and

technical credibility, and helps fix what it finds [24]. Its server is a Python application built with FastAPI. Several of its tasks have exactly the shape studied in this chapter; we describe the shape and the class of solution, not the product's own implementation.

- **Walking a site.** A website is a set of pages joined by links, and "visit a page, then visit every page it links to" is a recursive description of a crawl. Written as a recursive function it is also a dangerous one: a site's link structure can be thousands of pages deep (a pagination chain "next, next, next…", a calendar with a link to every following day), and each level would be a Python frame, so the default limit of 1,000 would be reached quickly. A crawler therefore keeps its own stack or queue of pages still to visit, together with a set of pages already seen and a limit on depth: the explicit-stack pattern of Section 10.13, with the order of visiting as a design choice (Chapter 13 studies exactly this choice as depth-first and breadth-first search).
- **Nested structured data.** The structured data that answer engines read, schema.org descriptions embedded in a page as JSON-LD, is nested: an `Organization` contains a `PostalAddress`, an `Article` has a `publisher` that is an `Organization` with a `logo`, and so on to any depth. Checking it is a recursive walk, one function that handles an object by handling each of its fields, exactly like `countServices` below. Because the input comes from outside, the walk needs a depth guard: a hostile or broken page could nest arrays a hundred thousand levels deep, and a recursive parser or checker would then fail with `RecursionError` (Python's own `json` module does so on such input).
- **Remembering results.** Some scoring work is expensive and requested repeatedly for the same input. Remembering results by their input, the memoization of Section 10.5.3, with the bounded least-recently-used cache of Section 9.9, avoids recomputing them; the same rule as for Fibonacci applies: it pays off exactly when the same subproblem really recurs.

The runnable version of these ideas uses the course's case study, Profitina Laundry. Its service menu is nested: categories hold services and smaller categories, as **Figure 10.13** shows. The menu is a *tree*, the subject of Chapter 11, and it is the natural input for a recursive program.

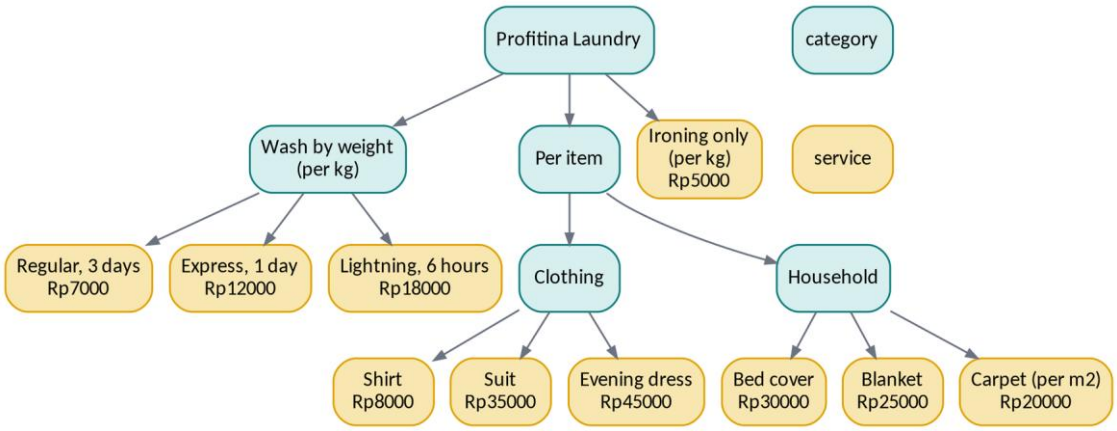


Figure 10.13 The service menu of Profitina Laundry: categories (teal) contain services (amber) and smaller categories, to any depth.

Listing 10.19 stores each menu entry as an `Item` with a name, a price (0 for a category) and a list of children. Four short recursive functions answer four questions: `printMenu` prints the menu as an indented outline (the item first, then its children, one level deeper), `countServices` counts the services (a service counts 1; a category counts the sum over its children), `height` measures how deep the nesting goes, and `cheapest` finds the cheapest service. Finally `countServicesIter` counts the services again with an explicit stack, as a crawler or a JSON checker would when the input is untrusted.

Listing 10.19 `12_laundry_menu.cpp`: recursive functions over Profitina Laundry's nested service menu, and the same count with an explicit stack.

```
// 12_laundry_menu.cpp -- Profitina Laundry's service menu is a nested structure:
// a category holds services and smaller categories. Recursion walks it naturally;
// an explicit stack walks it without using the call stack.
#include <algorithm>
#include <cstdio>
#include <string>
#include <utility>
#include <vector>
using namespace std;

struct Item {
    string name;
    long price;             // rupiah; 0 for a category
    vector<Item> children;  // empty for a service
};

Item service(string name, long price) { return Item{move(name), price, {}}; }
Item category(string name,
              vector<Item> kids) { return Item{move(name), 0, move(kids)}; }

void printMenu(const Item &it, int depth) {  // pre-order: item, then children
    if (it.children.empty())
        printf("%*s%-*s Rp%6ld\n", 2 * depth, "", 30 - 2 * depth, it.name.c_str(),
               it.price);
    else
        printf("%*s%s\n", 2 * depth, "", it.name.c_str());
    for (const Item &c : it.children) printMenu(c, depth + 1);
}

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;
}
```

Listing 10.19 (Continued) *12_laundry_menu.cpp: recursive functions over Profitina Laundry's nested service menu, and the same count with an explicit stack.*

```
const Item *cheapest(const Item &it) {  // the cheapest service
    if (it.children.empty()) return &it;
    const Item *best = nullptr;
    for (const Item &c : it.children) {
        const Item *b = cheapest(c);
        if (best == nullptr || b->price < best->price) best = b;
    }
    return best;
}

int countServicesIter(const Item &root) {  // the same count, no recursion
    vector<const Item *> st = {&root};
    int total = 0;
    while (!st.empty()) {
        const Item *it = st.back();
        st.pop_back();
        if (it->children.empty()) ++total;
        for (const Item &c : it->children) st.push_back(&c);
    }
    return total;
}

int main() {
    Item menu = category("Profitina Laundry", {
        category("Wash by weight (per kg)", {
            service("Regular, 3 days", 7000),
            service("Express, 1 day", 12000),
            service("Lightning, 6 hours", 18000)}),
        category("Per item", {
            category("Clothing", {
                service("Shirt", 8000),
                service("Suit", 35000),
                service("Evening dress", 45000)}),
            category("Household", {
                service("Bed cover", 30000),
                service("Blanket", 25000),
                service("Carpet (per m2)", 20000)})}),
        service("Ironing only (per kg)", 5000)});
```

Listing 10.19 (Continued) *12_laundry_menu.cpp: recursive functions over Profitina Laundry's nested service menu, and the same count with an explicit stack.*

```
    printMenu(menu, 0);
    const Item *c = cheapest(menu);
    printf("\nservices: %d recursively, %d with an explicit stack\n",
           countServices(menu), countServicesIter(menu));
    printf("levels below the top: %d\n", height(menu));
    printf("cheapest service: %s, Rp%ld\n", c->name.c_str(), c->price);
    return 0;
}
```

```
Profitina Laundry
  Wash by weight (per kg)
    Regular, 3 days            Rp  7000
    Express, 1 day             Rp 12000
    Lightning, 6 hours         Rp 18000
  Per item
    Clothing
      Shirt                    Rp  8000
      Suit                     Rp 35000
      Evening dress            Rp 45000
    Household
      Bed cover                Rp 30000
      Blanket                  Rp 25000
      Carpet (per m2)          Rp 20000
  Ironing only (per kg)        Rp  5000

services: 10 recursively, 10 with an explicit stack
levels below the top: 3
cheapest service: Ironing only (per kg), Rp5000
```

Each recursive function has the base case “a service, with no children” and the recursive case “a category: combine the answers of the children”. None of them needs to know how deep the menu is, which is the strength of recursion on nested data; the explicit-stack version gives the same count without using the call stack at all, which is its strength when the depth is not under our control. The Java version stores items in a `record`, and the Python version in a `dataclass`; both print the same outline.

10.15 Common Mistakes, and How to Avoid Them

No base case, or a base case that is never reached. `fact(n)` with the test `n == 0` loops forever for a negative argument, and a function that subtracts 2 from an odd argument jumps over 0. Test the smallest inputs (0, 1, 2 and any negative values the function might receive) before any others, and write the base-case test to cover everything below the smallest valid input (`n <= 1`, not `n == 1`).

A recursive call that does not shrink the problem. Passing `n` instead of `n - 1`, or `lo, hi` instead of `lo, mid - 1`, recurses on the same problem forever. For every recursive call, name the quantity that decreases (Section 10.2.5).

Expecting a stack overflow to be polite. In C++ it kills the program without a message (Section 10.3.4). Know the depth of your recursions: n for peeling one element, $\lg n$ for halving.

Recursing over long lists. `length`, `printBackward` and friends need one frame per node; a million nodes overflow every default stack (Sections 10.9.3 and 10.12). Use a loop or an explicit stack.

Naive recursion with overlapping subproblems. `fibNaive(40)` makes 331 million calls (Equation (10.7)). When the same arguments recur, memoize, or compute bottom-up (Section 10.5).

Calling twice what should be called once. Writing `power(m, n/2) * power(m, n/2)` makes two identical calls and loses all the benefit of halving; store the result in a variable (Section 10.7.2).

Trusting floating point for exact answers. Binet’s formula in `double` is wrong from $F(71)$ on (Section 10.5.7).

Relying on tail-call elimination. It happens at `-O2` in GCC and Clang, not at `-O0`, never in Java or CPython, and never by guarantee of the C++ standard (Section 10.11).

Forgetting overflow. 21! and $F(93)$ do not fit in 64 bits, and C++ does not warn (Sections 10.3.3 and 10.5.8). Use Python integers, `BigInteger`, or a modulus.

Raising Python's recursion limit without thinking. `sys.setrecursionlimit(10**9)` removes the protection, not the stack. Raise the limit only as far as the recursion really needs, and for very deep recursions run it in a thread with a larger stack, or convert it to a loop (Sections 10.12.3 and 10.13).

Losing the returned value. A recursive function that returns a new list head, such as `removeAll`, must be called as `head = removeAll(head, key)`; calling it as `removeAll(head, key)` leaves `head` pointing at a node that may have been deleted.

10.16 Chapter Summary

- A **recursive definition** or **recursive function** defines an answer in terms of answers to strictly smaller instances. It needs a **base case** that answers directly and a **recursive case** that makes progress toward it.
- **Induction** proves recursive functions correct: check the base case, assume the smaller calls are right, and check the combination. Closed forms such as $f(n) = 6 \cdot 2^n - 3$ (Equation (10.2)) are proved the same way.
- Each pending call is a **stack frame**. The stack winds up to the base case and unwinds as the calls return (**Figure 10.2**); the time of a recursion is the work of all its calls, and its extra memory is the largest number of frames present at once.
- **Factorial** by loop and by recursion both take $\Theta(n)$ time (Equation (10.4)); the loop uses $O(1)$ memory, the recursion $\Theta(n)$. 20! is the largest factorial that fits in 64 bits.
- **Fibonacci's** naive recursion makes $2F(n+1) - 1$ calls, $\Theta(\varphi^n)$ (Equation (10.7), Equation (10.8)): 331,160,281 calls for $F(40)$. Memoization makes $2n - 1$ calls, the bottom-up loop n additions in $O(1)$ memory, fast doubling $\lfloor \lg n \rfloor + 2$ calls (Equation (10.9)), and Binet's formula (Equation (10.10)) in `double` is exact only up to $F(70)$ (**Table 10.1**).
- **Euclid's algorithm** $\gcd(a, b) = \gcd(b, a \bmod b)$ (Equation (10.11)) needs at most about $1 + \log_\varphi b$ divisions (Lamé, Equation (10.12)); consecutive Fibonacci numbers are its worst case.
- **Fast exponentiation** halves the exponent: $\lfloor \lg n \rfloor + 2$ calls (Equation (10.15)) instead of $n + 1$, and with a modulus it computes $7^{10^{18}} \bmod (10^9 + 7)$ in 61 calls.
- The recursive **Towers of Hanoi** makes $M(n) = 2^n - 1$ moves (Equation (10.16)), and no algorithm can make fewer: its exponential cost belongs to the problem.
- **Arrays** recurse by peeling one element (depth n) or by halving (depth $\lg n$); **linked lists** recurse on the first node and the rest, which makes printing backward a one-line change and costs one frame per node.
- **Recursion trees** add the work level by level: $T(n) = 2T(n/2) + cn$ gives $\Theta(n \lg n)$ (Equation (10.17)), and the other common recurrences are in **Table 10.2**.
- A **tail call** is a call whose result is returned unchanged. GCC at `-O2` compiled the tail-recursive `sumAcc`, and even the accumulating `sumPlain`, into loops, but not at `-O0`; the JVM and CPython never do. Do not depend on it; `[[gnu::musttail]]` and `[[clang::musttail]]` exist for code that must.

- **Depth limits** (**Table 10.3**, Equation (10.18)): about 116,000 frames of 72 bytes in an 8 MB C++ stack; 10,000 to 60,000 in Java's default 1 MB thread stack; 1,000 in Python by default. Each can be raised, and each language offers threads with larger stacks.
- **Any recursion can become a loop with an explicit stack** of frames holding parameters, surviving locals and a stage (**Figure 10.12**); do this when the depth may be large (**Table 10.4**).

10.17 Exercises

10.1 The function g is defined by $g(0) = 0$ and $g(n) = g(n-1) + 2n - 1$ for $n \geq 1$. Compute $g(1)$ to $g(5)$ by unfolding the definition, guess a closed form, and prove it by induction.

10.2 Draw the call stack, as in **Figure 10.2**, at the moment `f(0)` is about to return during the computation of `f(3)` in **Listing 10.1**. How many frames of `f` are on the stack, and what value of `n` does each hold?

10.3 Write a recursive function `digitSum(n)` that returns the sum of the decimal digits of a non-negative integer, for example digitSum$(2026) = 2 + 0 + 2 + 6 = 10$. State its base case, and give its running time in terms of the number of digits.

10.4 Write a recursive function `printBinary(n)` that prints the binary representation of a non-negative integer, for example 1101 for 13. Explain why the recursive call must come *before* the `print` statement.

10.5 Write a recursive function `isSorted(A, n)` that decides whether $A[0..n-1]$ is in ascending order. What are its base cases, its running time and its recursion depth?

10.6 Prove by induction that the naive recursion computes $F(k)$ exactly $F(n-k+1)$ times while computing $F(n)$, for $1 \leq k \leq n$. Check the claim against the output of **Listing 10.5** for $n = 6$.

10.7 John McCarthy's *91 function* is defined for integers by $M(n) = n - 10$ if $n > 100$, and $M(n) = M(M(n+11))$ otherwise [25]. Program it, count its calls for a few arguments, and show that $M(n) = 91$ for every $n \leq 100$ that you test. Why is the proof that it always stops harder than for any other function in this chapter?

10.8 The *extended* Euclidean algorithm returns, besides $g = \gcd(a, b)$, integers x and y with $ax + by = g$. Derive its recursive case from Equation (10.11): if $bx_1 + (a \bmod b)y_1 = g$, express x and y through x_1 and y_1. Program it and find x, y for (228,90) and (1071,462).

10.9 (a) Compute $3^{13} \bmod 7$ by hand with repeated squaring, reducing modulo 7 after every multiplication. (b) Program modular fast power and compute $7^{222} \bmod 1000$. (c) A student writes the recursive case of **Pseudocode 10.4** as `return power(m, n/2) * power(m, n/2) * (n % 2 == 1 ? m : 1);`. Write the recurrence for the number of calls this version makes when n is a power of two, and solve it.

10.10 In the solution of the Towers of Hanoi for n disks, how many times does disk k move ($1 \leq k \leq n$)? Prove your answer by induction on n, and check that the total over all disks is $2^n - 1$.

10.11 Write recursive functions that print (a) all subsets of {1,2,3} and (b) all permutations of the letters of `ABC`. How many lines does each print for a set of n elements, and how deep does each recursion go?

10.12 For each of these functions, say whether its recursive call is a tail call, and if not, rewrite it with an accumulator so that it is: (a) `factRec` of **Listing 10.2**; (b) `gcdRec` of **Listing 10.6**; (c)

`length` of **Listing 10.11**; (d) `printBackward` of **Listing 10.11**. For which one is that impossible without an extra data structure, and why?

10.13 A recursive C++ function has a frame of 48 bytes. (a) Using Equation (10.18), estimate the deepest recursion possible on an 8 MB Linux stack and on a 1 MB Windows stack. (b) A Python program calls a recursive function on a list of 5,000 elements and fails with `RecursionError`. Give two ways to fix it, and say which you would choose.

10.14 A robot on a grid of $r \times c$ squares moves only right or down, from the top-left corner to the bottom-right. The number of paths satisfies $P(r,c) = P(r-1,c) + P(r,c-1)$ with $P(0,c) = P(r,0) = 1$ (counting the moves still to make). Program the naive recursion and a memoized one, and compare their call counts for $r = c = 2,8,12$. What closed form do the path counts suggest?

10.15 Rewrite `printBackward` of **Listing 10.11** without recursion, using an explicit stack. How much extra memory does it use for a list of n nodes, compared with the recursive version?

10.18 Solutions to the Exercises

10.1 Unfolding: $g(1) = 0 + 1 = 1$, $g(2) = 1 + 3 = 4$, $g(3) = 4 + 5 = 9$, $g(4) = 9 + 7 = 16$, $g(5) = 16 + 9 = 25$. The values are squares, so guess $g(n) = n^2$. Base case: $g(0) = 0 = 0^2$. Inductive step: if $g(n-1) = (n-1)^2$, then $g(n) = (n-1)^2 + 2n - 1 = n^2 - 2n + 1 + 2n - 1 = n^2$. ▫ (The definition says that the sum of the first n odd numbers is n^2.)

10.2 At that moment the stack holds, from bottom to top, the frames of `main`, `fTraced(3, 0)`, `fTraced(2, 1)`, `fTraced(1, 2)` and `fTraced(0, 3)`: four frames of the function, holding $n = 3,2,1,0$. The frames with $n = 3,2,1$ are each waiting inside the statement `long long smaller = fTraced(n - 1, depth + 1);` for their call to return; `fTraced(0, 3)` has set `result = 3` and is printing its return line. The trace of **Listing 10.1** shows the same thing: four `call` lines have appeared and only one `returns` line.

The programs for 10.3, 10.4, 10.5, 10.7, 10.8, 10.9, 10.11, 10.14 and 10.15 are collected in **Listing 10.20**; its output follows the listing.

Listing 10.20 `13_exercises.cpp`: solutions to the programming exercises of Chapter 10.

```cpp
// 13_exercises.cpp -- solutions to the programming exercises of Chapter 10.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;
typedef long long ll;

int digitSum(ll n) {  // Exercise 10.3
    if (n < 10) return (int)n;
    return (int)(n % 10) + digitSum(n / 10);
}

void printBinary(ll n) {             // Exercise 10.4
    if (n >= 2) printBinary(n / 2);  // the higher bits first
    printf("%lld", n % 2);
}

bool isSorted(const int A[], int n) {  // Exercise 10.5
    if (n <= 1) return true;
    return A[n - 2] <= A[n - 1] && isSorted(A, n - 1);
}

ll m91calls = 0;
int mc91(int n) {  // Exercise 10.7
    ++m91calls;
    if (n > 100) return n - 10;
    return mc91(mc91(n + 11));
}

ll extGcd(ll a, ll b, ll &x, ll &y) {  // Exercise 10.8: a x + b y = gcd
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    ll x1, y1;
    ll g = extGcd(b, a % b, x1, y1);  // b x1 + (a mod b) y1 = g
    x = y1;
    y = x1 - (a / b) * y1;
    return g;
}
```

Listing 10.20 (Continued) *13_exercises.cpp: solutions to the programming exercises of Chapter 10.*

```
ll powMod(ll m, ll n, ll mod) {  // Exercise 10.9
    if (n == 0) return 1 % mod;
    ll half = powMod(m, n / 2, mod);
    ll sq = half * half % mod;
    return n % 2 == 0 ? sq : sq * (m % mod) % mod;
}

// Exercise 10.11
void subsets(const vector<int> &s, size_t i, vector<int> &chosen) {
    if (i == s.size()) {
        printf("{");
        for (size_t k = 0; k < chosen.size(); ++k) printf(k ? ", %d" : "%d",
             chosen[k]);
        printf("} ");
        return;
    }
    subsets(s, i + 1, chosen);  // without s[i]
    chosen.push_back(s[i]);
    subsets(s, i + 1, chosen);  // with s[i]
    chosen.pop_back();
}

void permutations(string &s, size_t i) {  // Exercise 10.11
    if (i == s.size()) {
        printf("%s ", s.c_str());
        return;
    }
    for (size_t j = i; j < s.size(); ++j) {
        swap(s[i], s[j]);  // choose s[j] for position i
        permutations(s, i + 1);
        swap(s[i], s[j]);  // undo the choice
    }
}

ll gridCalls = 0;
ll paths(int r, int c) {  // Exercise 10.14, naive
    ++gridCalls;
    if (r == 0 || c == 0) return 1;
    return paths(r - 1, c) + paths(r, c - 1);
}
ll pathsMemo(int r, int c, vector<vector<ll>> &memo) {
    ++gridCalls;
    if (r == 0 || c == 0) return 1;
    if (memo[r][c] != 0) return memo[r][c];
    return memo[r][c] = pathsMemo(r - 1, c, memo) + pathsMemo(r, c - 1, memo);
}
```

Listing 10.20 (Continued) *13_exercises.cpp: solutions to the programming exercises of Chapter 10.*

```
struct Node {
    int data;
    Node *next;
};

void printBackwardIter(const Node *head) {  // Exercise 10.15
    vector<const Node *> st;
    for (const Node *p = head; p != nullptr; p = p->next) st.push_back(p);
    while (!st.empty()) {
        printf(" %d", st.back()->data);
        st.pop_back();
    }
    printf("\n");
}

int main() {
    printf("10.3  digitSum(2026) = %d, digitSum(9876543210) = %d\n",
           digitSum(2026), digitSum(9876543210LL));
    printf("10.4  13 = ");
    printBinary(13);
    printf(", 2026 = ");
    printBinary(2026);
    printf("\n");
    int a[] = {4, 7, 12, 18, 25, 31}, b[] = {4, 7, 18, 12, 25, 31};
    printf("10.5  isSorted(a) = %s, isSorted(b) = %s\n",
           isSorted(a, 6) ? "true" : "false",
           isSorted(b, 6) ? "true" : "false");

    bool all91 = true;
    for (int n = -50; n <= 100; ++n) all91 = all91 && mc91(n) == 91;
    printf("10.7  M(n) = 91 for every n from -50 to 100: %s\n",
           all91 ? "yes" : "no");
    for (int n : {87, 99, 100, 101, 150}) {
        m91calls = 0;
        int v = mc91(n);
        printf("      M(%d) = %d, calls made: %lld\n", n, v, m91calls);
    }

    ll x, y, g = extGcd(228, 90, x, y);
    printf("10.8  gcd(228, 90) = %lld = 228 * (%lld) + 90 * (%lld)\n", g, x, y);
    g = extGcd(1071, 462, x, y);
    printf("      gcd(1071, 462) = %lld = 1071 * (%lld) + 462 * (%lld)\n", g, x, y);
```

Listing 10.20 (Continued) *13_exercises.cpp: solutions to the programming exercises of Chapter 10.*

```
    printf("10.9  3^13 mod 7 = %lld, 7^222 mod 1000 = %lld\n", powMod(3, 13, 7),
           powMod(7, 222, 1000));

    vector<int> s = {1, 2, 3}, chosen;
    printf("10.11 subsets: ");
    subsets(s, 0, chosen);
    string abc = "ABC";
    printf("\n      permutations: ");
    permutations(abc, 0);
    printf("\n");

    for (int n : {2, 8, 12}) {
        gridCalls = 0;
        ll naive = paths(n, n);
        ll naiveCalls = gridCalls;
        vector<vector<ll>> memo(n + 1, vector<ll>(n + 1, 0));
        gridCalls = 0;
        ll fast = pathsMemo(n, n, memo);
        printf("10.14 %2d x %-2d grid: %7lld paths; naive %7lld calls, "
               "memo %3lld calls%s\n", n, n, naive, naiveCalls, gridCalls,
               fast == naive ? "" : " MISMATCH");
    }

    Node *head = nullptr;
    int values[] = {50, 40, 30, 40, 20};
    for (int v : values) head = new Node{v, head};
    printf("10.15 backward:");
    printBackwardIter(head);
    while (head != nullptr) {
        Node *next = head->next;
        delete head;
        head = next;
    }
    return 0;
}
```

```
10.3  digitSum(2026) = 10, digitSum(9876543210) = 45
10.4  13 = 1101, 2026 = 11111101010
10.5  isSorted(a) = true, isSorted(b) = false
10.7  M(n) = 91 for every n from -50 to 100: yes
      M(87) = 91, calls made: 29
      M(99) = 91, calls made: 5
      M(100) = 91, calls made: 3
      M(101) = 91, calls made: 1
      M(150) = 140, calls made: 1
10.8  gcd(228, 90) = 6 = 228 * (2) + 90 * (-5)
      gcd(1071, 462) = 21 = 1071 * (-3) + 462 * (7)
10.9  3^13 mod 7 = 3, 7^222 mod 1000 = 49
10.11 subsets: {} {3} {2} {2, 3} {1} {1, 3} {1, 2} {1, 2, 3}
      permutations: ABC ACB BAC BCA CBA CAB
10.14  2 x 2  grid:       6 paths; naive      11 calls, memo   9 calls
10.14  8 x 8  grid:   12870 paths; naive   25739 calls, memo 129 calls
10.14 12 x 12 grid: 2704156 paths; naive 5408311 calls, memo 289 calls
10.15 backward: 50 40 30 40 20
```

10.3 `digitSum` in **Listing 10.20**: base case $n < 10$ (a single digit is its own sum); recursive case: the last digit, n mod 10, plus the digit sum of $\lfloor n/10 \rfloor$, which has one digit fewer. For a number with d digits it makes d calls: $\Theta(d) = \Theta(\log n)$ time and stack. The output gives 10 for 2026 and 45 for 9,876,543,210.

10.4 `printBinary(n)` first prints the binary representation of $\lfloor n/2 \rfloor$ (all the digits except the last) and then the last digit, n mod 2. The last binary digit is the one we can compute first, but it must be printed last; making the recursive call first delays the `print` until all the higher digits have

been printed, exactly as `printBackward` delays each node. The output shows $13 = 1101_2$ and $2026 = 11111101010_2$.

10.5 `isSorted(A, n)`: an array of 0 or 1 elements is sorted (base case); a longer one is sorted if its last two elements are in order and its first $n-1$ elements are sorted. It makes at most n calls, stopping early at the first inversion because `&&` does not evaluate its right side when the left is false: $O(n)$ time and recursion depth up to $n-1$. The output shows `true` for 4, 7, 12, 18, 25, 31 and `false` for 4, 7, 18, 12, 25, 31.

10.6 Let $N_n(k)$ be the number of times $F(k)$ is computed inside `fibNaive(n)`, for $1 \le k \le n$. We prove $N_n(k) = F(n-k+1)$ by induction on n. For $n = 1$: $N_1(1) = 1 = F(1)$. For $n = 2$: $F(2)$ is computed once and $F(1)$ once, and $F(1) = F(2) = 1$. For $n \ge 3$: the call on n computes $F(n)$ once, which is $F(1)$, and otherwise every computation happens inside `fibNaive(n-1)` or `fibNaive(n-2)`, so for $k \le n-1$, $N_n(k) = N_{n-1}(k) + N_{n-2}(k) = F(n-k) + F(n-k-1) = F(n-k+1)$, where $N_{n-2}(n-1) = 0 = F(0)$ is consistent with the formula. For $n = 6$: $F(6)$ once ($F(1)$), $F(5)$ once ($F(2)$), $F(4)$ twice ($F(3)$), $F(3)$ three times ($F(4)$), $F(2)$ five times ($F(5)$), $F(1)$ eight times ($F(6)$), exactly as printed by **Listing 10.5**. ($F(0)$, which is not covered by the formula, is computed $F(n-1) = 5$ times.)

10.7 The output of **Listing 10.20** confirms $M(n) = 91$ for every n from -50 to 100, with 29 calls for $M(87)$, 5 for $M(99)$, 3 for $M(100)$ and one for any argument above 100. The difficulty is rule 2 of Section 10.2.5: the argument of the inner call, $n+11$, is *larger*, not smaller, and the outer call's argument is whatever the inner call returns. There is no simple quantity that decreases on every call; the proof that the recursion stops (by induction on $101-n$, in steps of 11) was one of the early showcases of formal program verification [25].

10.8 Write $a = qb + r$ with $q = \lfloor a/b \rfloor$ and $r = a \bmod b$. If $bx_1 + ry_1 = g$, substitute $r = a - qb$: $bx_1 + (a-qb)y_1 = ay_1 + b(x_1 - qy_1) = g$. So $x = y_1$ and $y = x_1 - qy_1$, with the base case $\gcd(a, 0) = a = a \cdot 1 + 0 \cdot 0$, that is $x = 1$, $y = 0$. The output gives $6 = 228 \cdot 2 + 90 \cdot (-5)$ and $21 = 1071 \cdot (-3) + 462 \cdot 7$; both check: $456 - 450 = 6$ and $-3213 + 3234 = 21$.

10.9 (a) $13 = 1101_2$. $3^1 \equiv 3$; $3^3 = (3^1)^2 \cdot 3 = 27 \equiv 6$; $3^6 = 6^2 = 36 \equiv 1$; $3^{13} = (3^6)^2 \cdot 3 \equiv 1 \cdot 3 = 3 \pmod 7$. (b) The program prints $7^{222} \bmod 1000 = 49$. (c) The two identical calls give $C(n) = 2C(n/2) + 1$ with $C(1) = 1 + 2C(0) = 3$ and $C(0) = 1$; the recursion tree has 2^i nodes on level i for $\lg n + 2$ levels, so $C(n) = 2^{\lg n + 2} - 1 = 4n - 1$: linear, worse than the naive method's $n+1$. The halving only helps if the half is computed once.

10.10 Disk k moves 2^{n-k} times. Induction on n: for $n = 1$, disk 1 moves once, that is 2^0 times. For $n \ge 2$, disk n moves once, that is 2^0 times (line 4 of **Pseudocode 10.5**), and every disk $k < n$ moves only inside the two recursive calls on $n-1$ disks, $2^{(n-1)-k}$ times in each by the hypothesis, 2^{n-k} times in total. The total is $\sum_{k=1}^{n} 2^{n-k} = 2^{n-1} + \cdots + 2 + 1 = 2^n - 1$, which is Equation (10.16) once more. (Section 5.22, Solution 5.2(c), found the same count from the move numbers.)

10.11 `subsets` decides for each element in turn whether to leave it out or take it, recursing on the remaining elements; at the end of each chain of n decisions it prints the chosen set. It prints 2^n sets (8 for $n = 3$) and recurses n levels deep. `permutations` chooses each remaining letter for position i by a swap, recurses on position $i+1$, and swaps back ("undoes the choice"), a pattern called *backtracking*. It prints $n!$ arrangements (6 for $n = 3$), also at depth n. The output lists `{}` `{3}` `{2}` `{2, 3}` `{1}` `{1, 3}` `{1, 2}` `{1, 2, 3}` and ABC ACB BAC BCA CBA CAB.

10.12 (a) `factRec` is not tail recursive (the multiplication waits); with an accumulator: `factAcc(n, acc)` returns `acc` when $n = 0$ and otherwise `factAcc(n - 1, acc * n)`, called as `factAcc(n, 1)`. (b) `gcdRec` is already tail recursive: its result is returned unchanged. (c) `length` is not (the `1 +` waits); `lengthAcc(p, acc)` returns `acc` for the null pointer and otherwise `lengthAcc(p->next, acc + 1)`. (d) `printBackward` is not, and it cannot be made so with a number as the accumulator: the nodes must be printed in the reverse of the order in which the recursion reaches them, so something must remember all the nodes not yet printed. That something is the call stack, or an explicit stack (Exercise 10.15), or a reversed copy of the list. An accumulator removes a pending *operation*; it cannot remove pending *data*.

10.13 (a) 8,388,608/48 ≈ 174,763 levels on an 8 MB stack and 1,048,576/48 ≈ 21,845 on a 1 MB stack, in practice somewhat less, because `main` and the runtime use part of the stack. (b) Either raise the limit, for example with `sys.setrecursionlimit(10_000)`, which is safe on Python 3.11 and later for a depth of 5,000; or rewrite the function as a loop, which needs no limit at all. The loop is the better choice: the list may grow, and a function that fails on a list of 5,000 elements will fail again on one of 50,000.

10.14 The program prints 6 paths for the 2×2 grid, 12,870 for 8×8 and 2,704,156 for 12×12. The naive recursion makes $2P(n, n) - 1$ calls (11; 25,739; 5,408,311), the same structure as Equation (10.7), because its leaves are exactly the calls that return 1. The memoized version makes 9, 129 and 289 calls: each of the n^2 inner pairs (r, c) is computed once, at the cost of two calls, plus the first call, so $2n^2 + 1$. The counts are binomial coefficients: $P(n, n) = \binom{2n}{n}$, because a path is a choice of which n of its $2n$ moves go down: $\binom{4}{2} = 6$, $\binom{16}{8} = 12{,}870$, $\binom{24}{12} = 2{,}704{,}156$.

10.15 `printBackwardIter` in **Listing 10.20** pushes a pointer to every node while walking forward and then pops and prints them: 50 40 30 40 20 for the list 20, 40, 30, 40, 50. It uses n pointers, 8 bytes each on a 64-bit machine, in the heap. The recursive version uses $n + 1$ frames on the call stack, each much larger than a pointer (at least a return address, a saved frame pointer and the parameter, typically 32 to 64 bytes). The explicit stack is therefore several times smaller, and above all it is not limited by the size of the call stack.

References

[1] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, Part I,” *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960.

[2] E. W. Dijkstra, “Recursive programming,” *Numerische Mathematik*, vol. 2, no. 1, pp. 312–318, 1960.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 2, 4, and 31.

[4] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 1.2.1 and 1.2.8.

[5] L. E. Sigler, *Fibonacci’s Liber Abaci: A Translation into Modern English of Leonardo Pisano’s Book of Calculation*. New York, NY, USA: Springer, 2002, pp. 404–405.

[6] D. Takahashi, “A fast algorithm for computing large Fibonacci numbers,” *Information Processing Letters*, vol. 75, no. 6, pp. 243–246, Nov. 2000.

[7] J. P. M. Binet, “Mémoire sur l’intégration des équations linéaires aux différences finies, d’un ordre quelconque, à coefficients variables,” *Comptes Rendus de l’Académie des Sciences*, vol. 17, pp. 559–567, 1843.

[8] T. L. Heath, *The Thirteen Books of Euclid’s Elements*, 2nd ed., vol. 2. Cambridge, U.K.: Cambridge University Press, 1926, Book VII, Propositions 1–2.

[9] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 4.5.2, 4.5.3, and 4.6.3.

[10] G. Lamé, "Note sur la limite du nombre des divisions dans la recherche du plus grand commun diviseur entre deux nombres entiers," *Comptes Rendus de l'Académie des Sciences*, vol. 19, pp. 867–870, 1844.

[11] É. Lucas, *Récréations mathématiques*, vol. 3. Paris, France: Gauthier-Villars, 1893, pp. 55–59 ("La Tour d'Hanoï").

[12] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994, ch. 1 and sec. 6.6.

[13] G. L. Steele Jr., "Debunking the 'expensive procedure call' myth, or, procedure call implementations considered harmful, or, LAMBDA: The Ultimate GOTO," in *Proc. 1977 ACM Annual Conference*, Seattle, WA, USA, Oct. 1977, pp. 153–162.

[14] GNU Project, "Options that control optimization," *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. [Accessed: 24-Sep-2026].

[15] GNU Project, "Statement attributes: musttail," *Using the GNU Compiler Collection (GCC 15.1)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-15.1.0/gcc/Statement-Attributes.html. [Accessed: 24-Sep-2026].

[16] GNU Project, "GCC 16 release series." [Online]. Available: https://gcc.gnu.org/gcc-16/. [Accessed: 24-Sep-2026].

[17] G. van Rossum, "Tail recursion elimination," *Neopythonic* (blog), Apr. 22, 2009. [Online]. Available: http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html

[18] Microsoft, "/STACK (Stack allocations)," *Microsoft Learn — C/C++ Build Reference*. [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/reference/stack-stack-allocations. [Accessed: 24-Sep-2026].

[19] Oracle, "The java command: -Xss," *Java SE 25 Tools Reference*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/specs/man/java.html. [Accessed: 24-Sep-2026].

[20] Oracle, "Interface Thread.Builder.OfPlatform: stackSize," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Thread.Builder.OfPlatform.html. [Accessed: 24-Sep-2026].

[21] Python Software Foundation, "sys — System-specific parameters and functions: setrecursionlimit," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/sys.html. [Accessed: 24-Sep-2026].

[22] Python Software Foundation, "What's new in Python 3.11: inlined Python function calls." [Online]. Available: https://docs.python.org/3/whatsnew/3.11.html. [Accessed: 24-Sep-2026].

[23] Python Software Foundation, "What's new in Python 3.12: sys." [Online]. Available: https://docs.python.org/3/whatsnew/3.12.html. [Accessed: 24-Sep-2026].

[24] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

[25] Z. Manna and J. McCarthy, "Properties of programs and partial function logic," in *Machine Intelligence 5*, B. Meltzer and D. Michie, Eds. Edinburgh, U.K.: Edinburgh University Press, 1970, pp. 27–37.

Chapter 11
Trees and Binary Search Trees

By the end of this chapter you will be able to say what a tree is and use its vocabulary precisely (root, parent, child, sibling, leaf, internal node, ancestor, descendant, path, level, depth, height, degree, subtree); draw and write a tree in four notations; store a tree in an array and in linked nodes and say when each fits; classify a binary tree as full, perfect, complete or skewed and bound its number of nodes and its height; traverse a binary tree in preorder, inorder, postorder and level order, recursively and with an explicit stack or queue, and use the three depth-first orders to turn an expression tree into prefix, infix and postfix notation; state the binary search tree property and implement, trace and analyze insertion (on the running example `insert(65, 5, 70, 3, 10)`), search, minimum, maximum, successor, predecessor, deletion in all three cases with both the successor and the predecessor variant, counting of nodes and leaves, height, and printing a tree sideways; measure how the order of insertion decides whether a tree is balanced or degenerate; recognize the rotation that self-balancing trees (AVL and red-black) use to keep every operation logarithmic; convert a general tree to a binary tree and rebuild a tree from its preorder listing and degrees; and choose between `std::map`, Java's `TreeMap` and the Python alternatives. Every operation is drawn state by state, every cost is derived, and every program is complete, compiled or run in C++, Java and Python, and memory-checked.

11.1 Where We Left Off

Every structure in this book so far has been a *line*. The array of Chapter 1 keeps its elements in one row; the stack and the queue of Chapter 5 add and remove at the ends of a row; the linked lists of Chapters 7 and 9 are rows joined by pointers. In a line, every element has at most one element after it, and the whole structure can be walked from one end to the other in a single pass.

A line forces a hard choice, which Chapters 2 and 7 made visible. A *sorted array* can be searched by binary search in $O(\lg n)$ steps (Section 2.5), but inserting a new element in its place shifts, on average, half of the array. A *linked list* inserts in $O(1)$ once it holds the right position (Section 7.6.3), but it can only find that position by walking, one node at a time, in $\Theta(n)$ steps. We want both at once: a structure that is as easy to search as a sorted array and as easy to change as a linked list.

This chapter gets there by giving up the line. A *tree* lets each element point to *several* elements below it. The pointers of Chapter 6 and the nodes of Chapter 7 are still the building material; what changes is the shape. And the shape is naturally recursive, which is why Chapter 10 came first: a tree is a root with smaller trees hanging from it, and almost every algorithm in this chapter is three or four lines of recursion that follow that definition.

The first half of the chapter is about trees in general and binary trees in particular: their vocabulary, their pictures, their representations and their traversals. The second half is about the *binary search tree*, which keeps its keys in an order that lets every search throw away a whole subtree at each step. We build it on the running example `insert(65, 5, 70, 3, 10)`, search it, count it, print it, delete from it in all three cases, and then measure what happens when the same

keys arrive in a bad order. Chapter 12 works problems of exactly this kind, on trees built the same way.

11.2 What a Tree Is

11.2.1 Four trees from everyday life

Before any definition, look at the four drawings of **Figure 11.1**. They come from very different places, yet they have the same shape.

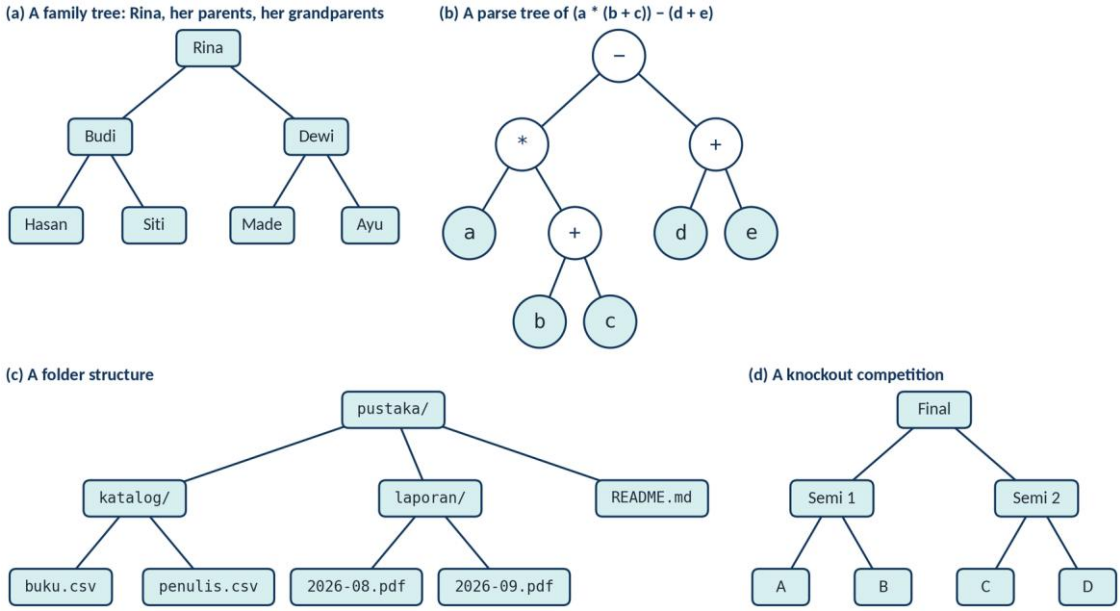


Figure 11.1 Four trees: (a) a family tree of ancestors, (b) the parse tree a compiler builds for an arithmetic expression, (c) the folders and files on a disk, (d) the draw of a knockout competition.

- **A family tree (a).** Rina is drawn at the top. Below her are her two parents, and below each parent are that parent's two parents. Every person in the drawing except Rina is reached from Rina along exactly one chain of lines.
- **A parse tree (b).** A compiler that reads `(a * (b + c)) - (d + e)` does not keep it as a string of characters. It builds this tree: the operator that is applied *last*, the subtraction, sits at the top; its two operands are the two smaller trees below it; and the letters, which are not operators, sit at the bottom. Evaluating the expression means evaluating the two lower trees and then applying the operator at the top. Section 11.6.6 builds such trees and reads them back in three notations.
- **A folder structure (c).** The folder `pustaka/` holds two folders and a file; each folder holds files of its own. The operating system finds `buku.csv` by walking from the top folder down the path `pustaka/katalog/buku.csv`. Section 10.14 met the same shape in Profitina Laundry's service menu.
- **A knockout competition (d).** Four teams play two semifinals; the two winners meet in the deciding match. Read from the bottom up, each drawing of two lines meeting at a box is one match.

Each drawing has one element at the top, each other element hangs below exactly one element, and nothing hangs in a circle. That is all a tree is.

11.2.2 The definition

A tree is a *non-linear* data structure: it describes a *hierarchical*, one-to-many relationship among its elements. Every element may have many elements directly below it, but only one directly above it. The pictures always put the special element with nothing above it at the top, so computer scientists draw their trees upside down, with the root in the sky and the leaves on the ground.

> **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* of T, and (2) the remaining nodes are partitioned into $m \geq 0$ disjoint sets $T_1, T_2, \ldots, T_m$, each of which is itself a tree. $T_1, \ldots, T_m$ are the *subtrees* of the root. The line drawn from the root to the root of each subtree is an *edge*.

This definition, which follows Knuth [1], is *recursive*: it defines a tree in terms of smaller trees. That is not circular, for the same reason the recursive definitions of Section 10.2.2 are not circular: every subtree has fewer nodes than the tree it belongs to, so the unfolding stops at trees of one node, which have $m = 0$ subtrees. The recursion in the definition is the reason why recursion is the natural way to program trees.

Two facts follow at once. First, from the root to any node there is exactly one path, because each node belongs to exactly one subtree of the root, and inside that subtree the same is true again. Second, a tree with n nodes has exactly

$$e = n - 1 \tag{11.1}$$

edges. The proof is by induction on n, the method of Section 10.2.3. A tree of one node has no edges, and $1 - 1 = 0$. For a larger tree, suppose the claim holds for all smaller trees; the root's subtrees have $n_1, \ldots, n_m$ nodes with $n_1 + \cdots + n_m = n - 1$, so by the hypothesis they have $(n_1 - 1) + \cdots + (n_m - 1) = n - 1 - m$ edges, and the root adds one edge to each of its m subtrees, for a total of $n - 1 - m + m = n - 1$. Equation (11.1) is worth remembering, because it gives a quick check on every tree you draw: count the nodes, count the lines, and the lines must be one fewer. Another way to see it: every node except the root has exactly one edge going *up* to its parent, so there are as many edges as non-root nodes.

A collection of zero or more disjoint trees is called a *forest*. Removing the root of a tree leaves a forest: its subtrees.

11.2.3 The vocabulary

Trees come with a large vocabulary, much of it borrowed from family trees. **Figure 11.2** shows the tree on which we define it, and **Table 11.1** lists each term with its value in that tree.

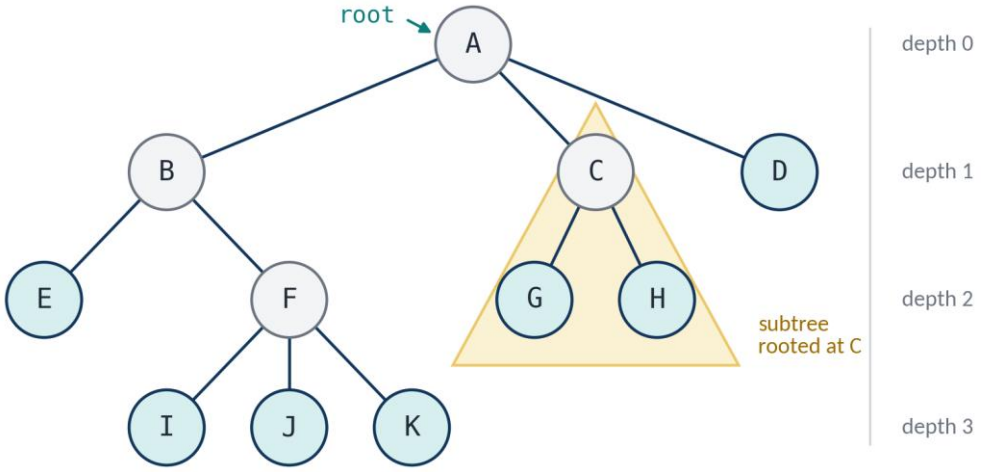


Figure 11.2 A tree of eleven nodes. Internal nodes are grey, leaves are teal; the amber triangle marks the subtree rooted at C, and the depth of each level is given on the right.

- The **root** is the one node with no node above it: A.
- If an edge joins node u to a node v directly below it, u is the **parent** of v and v is a **child** of u. Every node except the root has exactly one parent. Children of the same parent are **siblings**.
- A node with no children is a **leaf**, or *external node*. A node with at least one child is an **internal node**.
- A **path** is a sequence of nodes $v_0, v_1, \ldots, v_k$ in which each v_{i+1} is a child of v_i; its **length** is k, the number of edges on it.
- The **ancestors** of a node are the nodes on the path from the root to it, other than the node itself (its parent, grandparent, and so on up to the root). The **descendants** of a node are the nodes of which it is an ancestor (its children, grandchildren, and so on).
- The **subtree rooted at** v is v together with all its descendants. Any node can be viewed as the root of the subtree below it; in **Figure 11.2** the subtree rooted at C holds C, G and H.
- The **depth** of a node is the length of the path from the root to it, which is also its number of ancestors. The root has depth 0. The set of nodes with the same depth forms a **level**; in this book the root is on level 0, so a node's level equals its depth.
- The **height** of a node is the length of the *longest* path from it down to a leaf. A leaf has height 0. The **height of the tree** is the height of its root, which equals the largest depth of any node. By convention the empty tree has height -1, which makes the recursive formulas of Section 11.7.5 come out without a special case.
- The **degree** of a node is its number of children. The **degree of a tree** is the largest degree of any of its nodes.
- The **size** of a tree is its number of nodes.

Table 11.1 The vocabulary of trees, evaluated on the tree of **Figure 11.2**.

Term	Value in
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

DEPTH IS NOT HEIGHT. Depth is measured *down from the root* to a node: "how far below the top am I?" Height is measured *up from the lowest leaf* below a node: "how tall is the tree that hangs from me?" In **Figure 11.2**, F has depth 2 and height 1, and B has depth 1 and height 2. Only for the whole tree do the two agree: the height of the tree is the largest depth in it. Some books start counting levels at 1 instead of 0; when you read another text, check its convention before you use its formulas.

11.2.4 A worked example on a small tree

To make the vocabulary automatic, work through the seven-node tree whose root A has children B and C, where B has children D and E, and C has children F and G (**Figure 11.7** (b) draws it).

- The ancestors of F are its parent C and C's parent A, so ancestors(F) $=\{C, A\}$ and F has depth 2.
- The descendants of C are its children F and G, which have no children of their own: descendants(C) $=\{F, G\}$.
- The parent of D is B. The children of A are B and C. The only sibling of F is G.
- The leaves are D, E, F and G; the internal nodes are A, B and C.
- Every internal node has degree 2, so the tree has degree 2; its height is 2, and its size is 7, with $7 - 1 = 6$ edges.

11.2.5 Static and dynamic trees

Two kinds of use recur. In a **static tree** the shape is decided once and the contents of the nodes do not change afterward: the parse tree of an expression, the draw of a tournament, the table of contents of a printed book. In a **dynamic tree** nodes are added (*inserted*) and removed (*deleted*) while the program runs, and the shape changes with them: a library catalog that gains and loses books, an index that grows with every new record. Static trees can be stored compactly, often in an array (Section 11.4.1). Dynamic trees are almost always built from linked nodes (Section 11.4.2), for the same reason Chapter 7 built dynamic lists from nodes: nothing has to be reserved in advance, and a change touches only a few pointers.

The root is special in both kinds. It is the node from which every other node can be traced, and in a dynamic tree it is the node created first: the first insertion into an empty tree creates the root, and every later node is attached somewhere below it.

11.3 Drawing and Writing a Tree

The top-down drawing is not the only way to show a tree. **Figure 11.3** shows the same ten-node tree in three pictures, and a fourth notation writes it on one line.

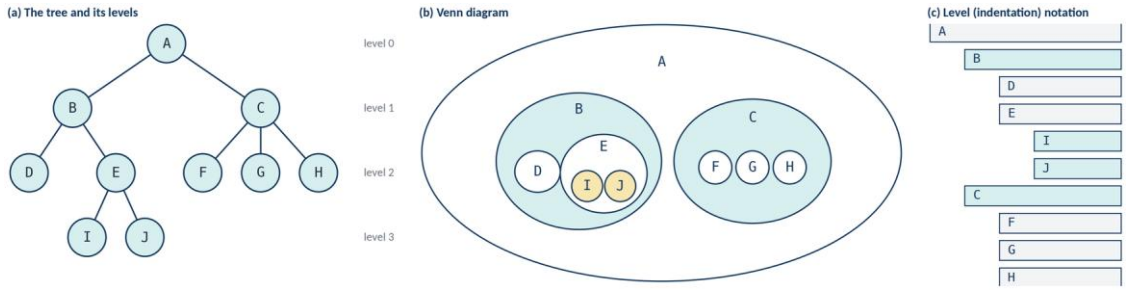


Figure 11.3 One tree in three notations: (a) the usual drawing with its levels, (b) a Venn diagram, in which every subtree is a set drawn inside its parent's set, (c) level notation, in which each node is a bar indented by its depth.

- **The drawing with levels (a)** is the one used in the rest of this book. The height of this tree is 3, and each level is labeled on the right.
- **A Venn diagram (b)** draws each subtree as a closed region, nested inside the region of its parent. The region A contains everything; inside it, B and C are disjoint regions, because their subtrees share no node; inside B are D and the region E, which holds I and J. A Venn diagram shows at a glance that the subtrees of a node never overlap, which is part (2) of the definition.
- **Level notation (c)** lists the nodes one per line in preorder (the order of Section 11.6.1) and indents each line by its depth, so that a node's children are the lines directly beneath it that are indented one step further. It is the notation of a file browser, of the outline view of a word processor and of Section 11.7.6, which prints a binary tree sideways in exactly this spirit.
- **Parenthesis notation** writes a node, then, in parentheses and separated by commas, the notation of each of its subtrees. For the tree of **Figure 11.3**:

```
(A(B(D,E(I,J)),C(F,G,H)))
```

The outer pair of parentheses encloses the whole tree; each leaf is just its name. Reading from left to right visits the nodes in preorder. Parenthesis notation is how LISP programs, and the JSON documents of the web, write trees in text: every nested pair of brackets is a subtree. Exercise 11.2 asks you to produce the four notations for another tree, and the program of Section 11.10.3 prints trees in this notation.

11.4 Representing a Tree in Memory

A tree in memory needs two things for each node: its data, and a way to reach its children. There are two classic answers, which mirror the two representations of a list: a *static* one in an array and a *dynamic* one with pointers.

11.4.1 The static representation: an array

For a binary tree (a tree whose nodes have at most two children, Section 11.5) there is a neat way to use no pointers at all. Number the positions of the tree level by level, from left to right, starting at 0 for the root, and store the node at position i in slot i of an array. Then the children and the parent of the node in slot i are found by arithmetic:

$$left(i) = 2i + 1, \qquad right(i) = 2i + 2, \qquad parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor \tag{11.2}$$

Why does this work? Level ℓ holds at most 2^ℓ positions (Section 11.5.3), so levels $0,1,\ldots,\ell-1$ occupy slots 0 to $2^\ell - 2$ and level ℓ begins at slot $2^\ell - 1$. The node at slot i on level ℓ is the $(i - 2^\ell + 1)$-th node of its level, counting from 0; its children are the $2(i - 2^\ell + 1)$-th and the next node of level $\ell + 1$, which start at slot $2^{\ell+1} - 1$. Adding, the left child is at slot $2^{\ell+1} - 1 + 2(i - 2^\ell + 1) = 2i + 1$, and the right child one slot later. The formula for the parent simply inverts both.

Figure 11.4 shows a six-node tree in an array of seven slots. Node C has a right child G but no left child, so slot $2 \cdot 2 + 1 = 5$ stays empty, marked `-`.

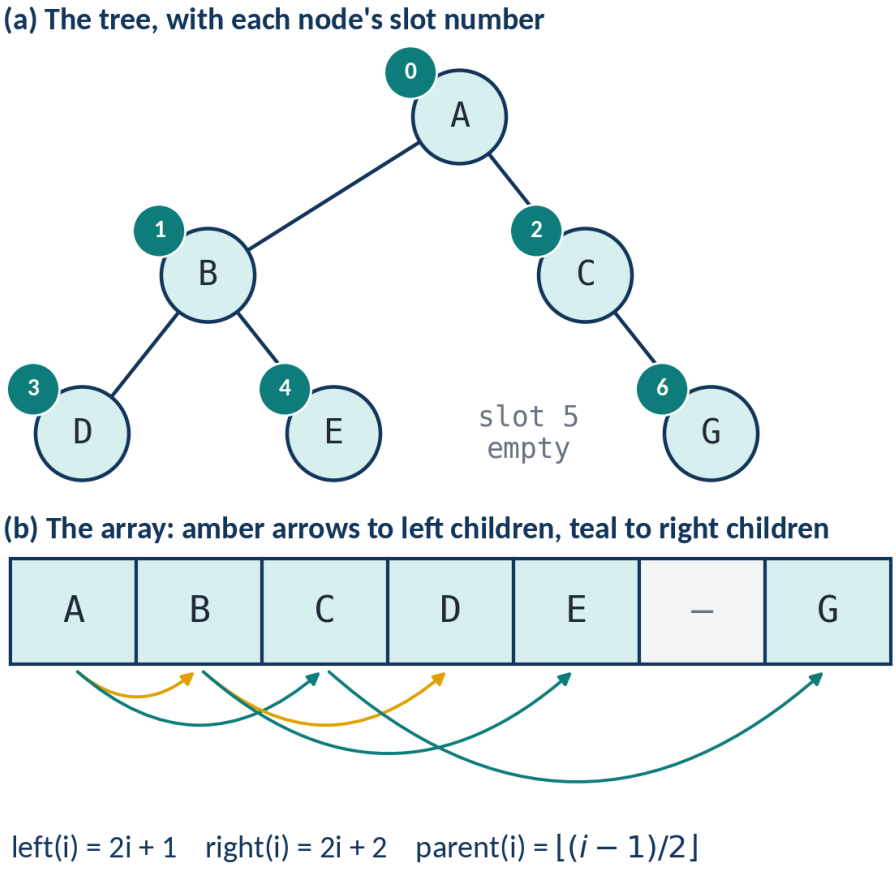


Figure 11.4 The static representation: (a) each node is numbered level by level, (b) node i is stored in slot i, and its children are found at slots $2i + 1$ and $2i + 2$ by Equation (11.2).

Listing 11.1 stores this tree in a string, one character per slot, prints every node with its parent and children computed by Equation (11.2), and walks the tree in preorder and inorder by following the index arithmetic instead of pointers.

Listing 11.1 `array_tree.cpp`: a binary tree stored in an array; children and parents are found by index arithmetic.

```cpp
// array_tree.cpp -- a binary tree stored in an array (the static representation).
#include <cstdio>
#include <string>

// Slot i holds a node; its children live in slots 2i+1 and 2i+2, its parent in
// slot (i-1)/2. A '-' marks an empty slot (no node there).
const std::string TREE = "ABCDE-G";

bool exists(const std::string &t, int i) {
    return i >= 0 && i < (int) t.size() && t[i] != '-';
}

void preOrder(const std::string &t, int i) {       // root, left, right
    if (!exists(t, i)) return;
    std::printf("%c ", t[i]);
    preOrder(t, 2 * i + 1);
    preOrder(t, 2 * i + 2);
}

void inOrder(const std::string &t, int i) {        // left, root, right
    if (!exists(t, i)) return;
    inOrder(t, 2 * i + 1);
    std::printf("%c ", t[i]);
    inOrder(t, 2 * i + 2);
}

int main() {
    std::printf("slot  node  parent  left  right\n");
    for (int i = 0; i < (int) TREE.size(); ++i) {
        if (!exists(TREE, i)) {
            std::printf("%4d     -   (empty slot)\n", i);
            continue;
        }
        int p = (i - 1) / 2, l = 2 * i + 1, r = 2 * i + 2;
        std::printf("%4d  %4c  %6c  %4c  %5c\n", i, TREE[i],
                    i == 0 ? '-' : TREE[p],
                    exists(TREE, l) ? TREE[l] : '-',
                    exists(TREE, r) ? TREE[r] : '-');
    }
    std::printf("preorder: ");
    preOrder(TREE, 0);
    std::printf("\ninorder:  ");
    inOrder(TREE, 0);
    std::printf("\n");
```

Listing 11.1 (Continued) *array_tree.cpp: a binary tree stored in an array; children and parents are found by index arithmetic.*

```cpp
    // A right-skewed chain of h+1 nodes needs 2^(h+1)-1 slots.
    for (int h = 1; h <= 4; ++h) {
        int slots = (1 << (h + 1)) - 1;
        std::printf("right chain of %d nodes needs %2d slots (%d wasted)\n",
                    h + 1, slots, slots - (h + 1));
    }
    return 0;
}
```

```
slot  node  parent  left  right
   0     A       -     B      C
   1     B       A     D      E
   2     C       A     -      G
   3     D       B     -      -
   4     E       B     -      -
   5     -   (empty slot)
   6     G       C     -      -
preorder: A B D E C G
inorder:  D B E A C G
right chain of 2 nodes needs  3 slots (1 wasted)
right chain of 3 nodes needs  7 slots (4 wasted)
right chain of 4 nodes needs 15 slots (11 wasted)
right chain of 5 nodes needs 31 slots (26 wasted)
```

The array representation costs no memory for pointers and finds a parent, which the linked representation of the next section cannot do, in $O(1)$. Its weakness is the empty slots. A tree in which every level is full except possibly the last, filled from the left, which Section 11.5.2 calls *complete*, has no empty slots at all, and that is why Chapter 14 stores its heaps this way. A tree with gaps wastes a slot for every missing position, and the last lines of the output show how badly: a chain of $h + 1$ nodes that always goes right needs $2^{h+1} - 1$ slots, so a chain of just 21 nodes would need more than two million. The array suits static trees of a known, full shape; a tree whose shape changes with every insertion needs pointers.

11.4.2 The dynamic representation: linked nodes

The dynamic representation gives each node two pointers, one to its left child and one to its right child, with a null pointer where a child is missing. It is the doubly linked node of Chapter 9 with its two pointers given a new meaning: instead of `prev` and `next`, which point along a line, we have `left` and `right`, which point *down* to two different subtrees. The node declaration, in the classic C style of Section 7.4.1, is:

```
typedef struct MyNode {
    int data;
    struct MyNode *left;
    struct MyNode *right;
} Node;
```

and the whole tree is reached through one pointer to the root, `Node *root`, which is `nullptr` when the tree is empty. **Figure 11.5** draws the five-node tree that Section 11.7.2 will build.

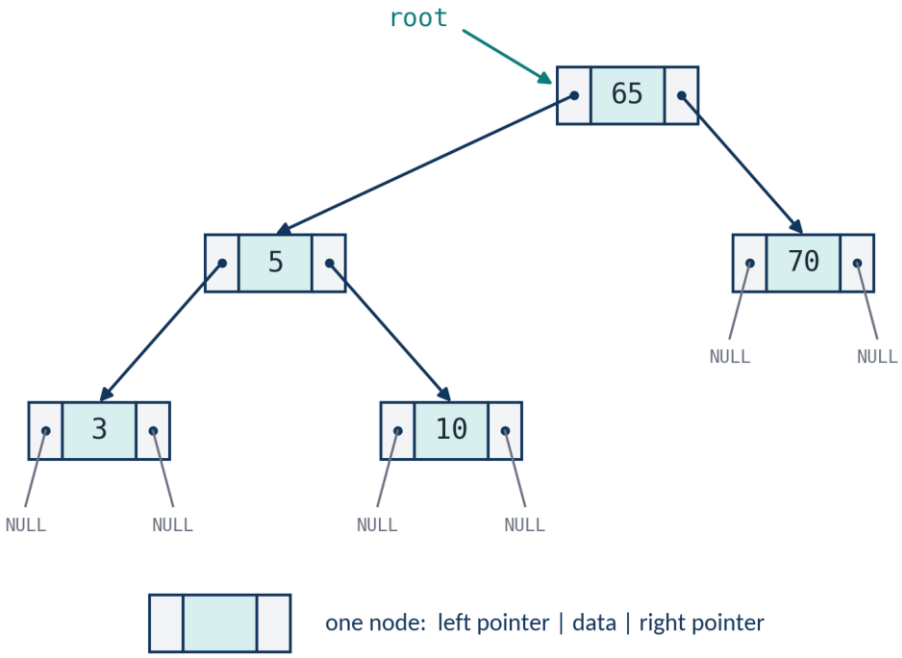


Figure 11.5 The dynamic representation of a binary tree: every node has a left pointer, a data field and a right pointer; `root` points at the top node, and every missing child is a NULL pointer.

Count the NULL pointers in the figure: there are six, for five nodes. That is no accident. Each of the n nodes has two pointer fields, $2n$ in all, and by Equation (11.1) exactly $n - 1$ of them hold an edge, so the number of null pointers in a binary tree of n nodes is

$$2n - (n - 1) = n + 1 \tag{11.3}$$

More than half of the pointer fields of every binary tree are null. That is the price of the flexible shape, and it is usually a price worth paying. (Some implementations reuse the null fields as "threads" that point to the inorder successor, which makes traversal possible without a stack; Knuth [1] describes these *threaded trees*.)

11.4.3 Trees with any number of children

When a node may have any number of children, as in the family, folder and competition trees of **Figure 11.1**, neither of the representations above fits directly. Three common answers are:

- **A parent array.** Store, for each node, the index of its parent: `parent[i]`. It is compact and finds the path to the root easily, which is exactly what a file system does to build a full path name, but finding the children of a node needs a scan of the whole array.
- **A list of children.** Give each node a dynamic array of pointers to its children (`std::vector<Node*>` in C++, a `List` in Java, a `list` in Python). This is how the DOM tree of a web page and the menu tree of Section 10.14 are usually stored.
- **First child, next sibling.** Give each node exactly two pointers: one to its *first child* and one to its *next sibling*. Every node of any tree then has two pointers, just like a binary tree node, and Section 11.10 shows that this is in fact a way of turning any tree into a binary tree.

11.5 Binary Trees

11.5.1 Definition

> **DEFINITION.** A *binary tree* is either *empty* (it has no nodes), or it consists of a root node together with two binary trees, called its *left subtree* and its *right subtree*, which have no node in common.

Two differences from the general tree of Section 11.2.2 matter. A binary tree may be empty. And the two subtrees are *distinguished*: a node with only a left child is a different binary tree from a node with only a right child, even though as general trees they would be the same. So every node of a binary tree has at most two children, each labeled left or right. **Figure 11.6** labels the parts of one.

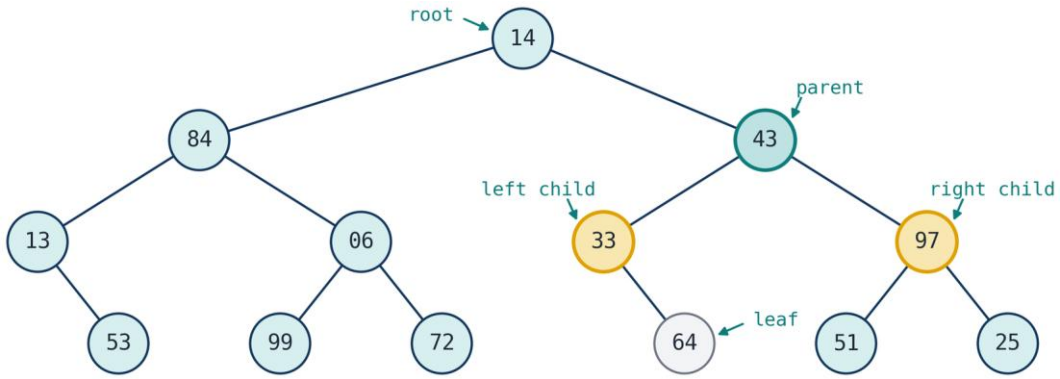


Figure 11.6 A binary tree of fourteen nodes: node 43 is the parent of its left child 33 and its right child 97, and 64, which has no children, is a leaf.

The binary tree matters far more than its restriction suggests. Two children per node is exactly what a yes/no decision needs, so every comparison-based search and every expression with binary operators is a binary tree. And Section 11.10 shows that any tree whatsoever can be stored as a binary tree, so nothing is lost by studying binary trees first.

11.5.2 Full, perfect, complete and skewed

Binary trees are classified by their shape, and the shape decides how fast the operations of Section 11.7 run. **Figure 11.7** shows the four classes and one counterexample.

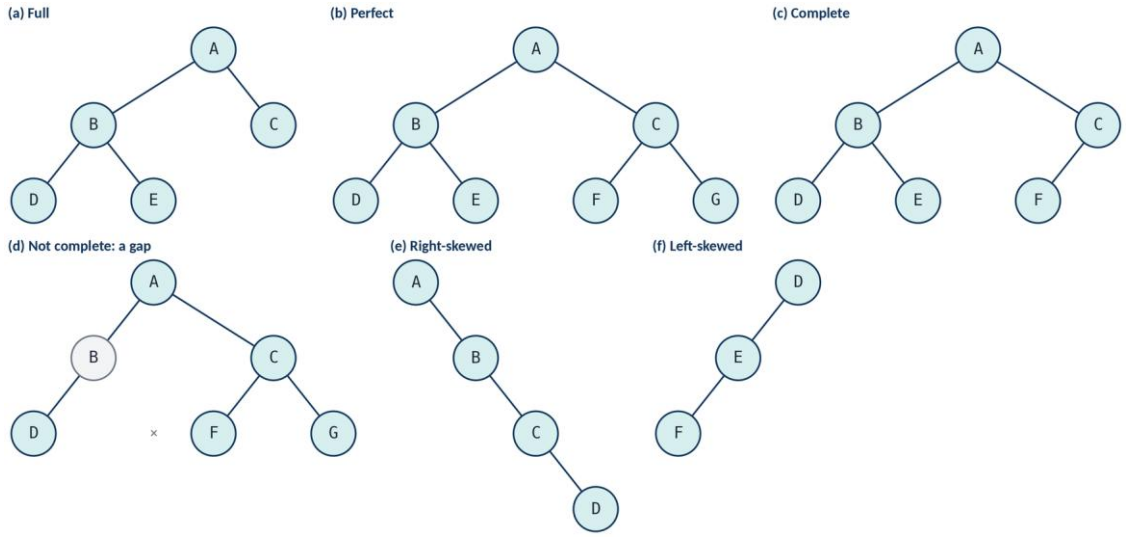


Figure 11.7 Shapes of binary trees: (a) full, (b) perfect, (c) complete, (d) not complete, because position 4 on the last level is empty while position 5 is used, (e) right-skewed, (f) left-skewed.

> **DEFINITION.** A binary tree is **full** (also called *proper* or *strictly binary*) if every node has either 0 or 2 children. It is **perfect** if it is full and all its leaves have the same depth. It is **complete** if every level except possibly the last is completely filled and the nodes of the last level are as far left as possible. It is **skewed** (or *degenerate*) if every internal node has exactly one child; a skewed tree whose children are all right children is *right-skewed*, and one whose children are all left children is *left-skewed*.

The classes overlap in a definite way: every perfect tree is both full and complete; a complete tree need not be full (**Figure 11.7** (c): C has only one child); a full tree need not be complete (**Figure 11.7** (a) is full and, as it happens, complete; move D and E under C and it would be full but not complete). A skewed tree of n nodes is a chain: it has height $n - 1$, the largest possible, and it is really a linked list drawn diagonally.

> **THE NAMES ARE NOT UNIVERSAL.** Older textbooks, and some course notes in Indonesia, use *full binary tree* for what this book calls *perfect* ("all nodes except the leaves have two children and every path has the same length") and *complete binary tree* for what this book calls *full* ("every internal node has two children, paths may differ in length"). This book follows Cormen et al. [2] and the current standard usage, summarized in **Table 11.2**. When you read a definition elsewhere, check which one is meant.

Table 11.2 The four shapes of binary tree, and the other names they go by.

This book	Condition	Also called
Full	every node has 0 or 2 children	proper, strictly binary; "complete" in some older texts
Perfect	full, and all leaves at the same depth	"full" in some older texts
Complete	all levels full except the last, which is filled from the left	nearly complete, heap-shaped
Skewed	every internal node has exactly one child	degenerate, a chain

11.5.3 How many nodes fit, and how low a tree can be

Since each node has at most two children, level 0 holds at most 1 node, level 1 at most 2, level 2 at most 4, and each level at most twice the level above. By induction on ℓ:

$$\text{nodes on level } \ell \;\le\; 2^{\ell} \tag{11.4}$$

A binary tree of height h has levels 0 to h, so by Equation (11.4) and the sum of a geometric series it has at most

$$n \;\le\; 1 + 2 + 4 + \cdots + 2^{h} = \sum_{\ell=0}^{h} 2^{\ell} = 2^{h+1} - 1 \tag{11.5}$$

nodes, with equality exactly for the perfect tree. (To check the sum: $2 \cdot (1 + 2 + \cdots + 2^{h}) - (1 + 2 + \cdots + 2^{h}) = 2^{h+1} - 1$, because all the middle terms cancel.) In the convention that numbers the root level 1, the same facts read "level k holds at most 2^{k-1} nodes, and a tree of k levels holds at most $2^{k} - 1$"; the content is identical.

Turn Equation (11.5) around and it becomes the most important inequality of this chapter. If a tree of height h holds n nodes, then $n \le 2^{h+1} - 1$, so $2^{h+1} \ge n + 1$, $h + 1 \ge \lg(n + 1)$ and, because h is a whole number,

$$h \;\ge\; \lceil \lg(n + 1) \rceil - 1 \;=\; \lfloor \lg n \rfloor \tag{11.6}$$

No binary tree of n nodes can be lower than $\lfloor \lg n \rfloor$; a complete tree achieves this bound, and a skewed tree has the opposite extreme, $n - 1$. For $n = 1{,}000{,}000$ that is the difference between height 19 and height 999,999. The whole second half of this chapter is about which of the two extremes a binary search tree ends up near, because every operation on it costs time proportional to the height.

One more counting fact is useful. In a full binary tree, the number of leaves L and the number of internal nodes I satisfy

$$L = I + 1 \tag{11.7}$$

Proof: each internal node has exactly two children, so there are $2I$ edges; by Equation (11.1) there are also $n - 1 = L + I - 1$ edges; setting $2I = L + I - 1$ gives $L = I + 1$. The knockout competition of **Figure 11.1** (d) is a full binary tree: four teams (leaves) need three matches (internal nodes). A tournament of 64 teams needs 63 matches, whatever the draw.

11.6 Traversing a Binary Tree

To *traverse* a tree is to visit every node exactly once, doing something at each visit: printing it, counting it, adding it up. A list has one natural order; a tree has several, and each one is useful.

11.6.1 Three depth-first orders

The recursive definition of a binary tree suggests the recursive traversal: visit the root, traverse the left subtree, traverse the right subtree. The only choice is *when* to visit the root relative to its two subtrees, and the three choices give the three classic orders:

- **Preorder:** visit the root, then traverse the left subtree, then the right subtree.
- **Inorder:** traverse the left subtree, then visit the root, then traverse the right subtree.
- **Postorder:** traverse the left subtree, then the right subtree, then visit the root.

The names say where the root goes: *pre*, before its subtrees; *in*, between them; *post*, after them. In all three, the left subtree is traversed before the right one. **Pseudocode 11.1** states them.

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
```

Each procedure has the shape Section 10.2.5 asked of every recursion: the base case is the empty tree (`x = NIL`), which needs no work, and each recursive call receives a strictly smaller tree. **Figure 11.8** numbers the visits on the tree built by `insert(65, 5, 70, 3, 10)`.

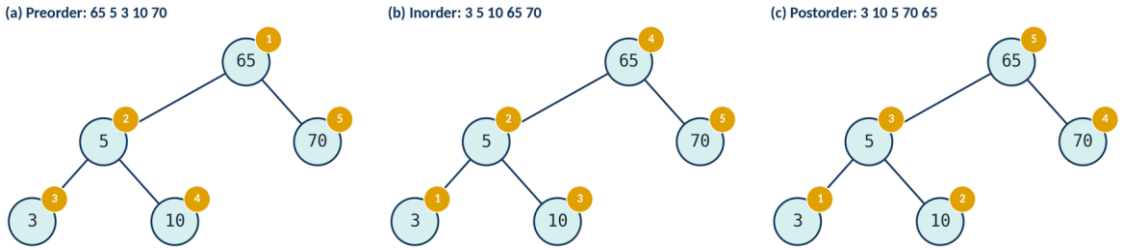


Figure 11.8 The order in which each traversal visits the nodes of the same tree; the amber badge on a node is its position in the visit sequence.

Trace preorder by hand, one call per line, to see the recursion at work:

1. PREORDER(65): visit 65, then call PREORDER(5).
2. PREORDER(5): visit 5, then call PREORDER(3).
3. PREORDER(3): visit 3; both children are NIL, so both calls return at once; return to 5.
4. Back in PREORDER(5): call PREORDER(10): visit 10; return; PREORDER(5) is finished; return to 65.
5. Back in PREORDER(65): call PREORDER(70): visit 70; return. Done.

The visits were 65, 5, 3, 10, 70. The same kind of trace gives inorder 3, 5, 10, 65, 70 and postorder 3, 10, 5, 70, 65. Look at the inorder sequence: it is *sorted*. This is not luck. Section 11.7.1 proves that inorder visits the keys of every binary search tree in ascending order, and much of the usefulness of search trees comes from it.

Each order has its natural jobs.

- **Preorder** visits a node before anything below it, so it is the order for *copying* a tree (create the node, then copy its subtrees), for *printing* it as an outline (Section 11.3) and for *saving* it so it can be rebuilt (Section 11.10.2).

- **Inorder** visits a binary search tree in sorted order, so it is the order for listing, for range queries, and for checking that a tree really is a search tree.
- **Postorder** visits a node after everything below it, so it is the order for *deleting* a tree (release the children before the parent, or the parent's pointers to them are lost) and for *computing* anything that depends on the subtrees: the size, the height, and the value of an expression (Section 11.6.6).

11.6.2 A walk around the tree

There is a single picture that contains all three orders. Imagine walking around the outside of the tree, starting to the left of the root, keeping the tree always on your right hand and going down into every gap between branches, as **Figure 11.9** shows. This walk, called an *Euler tour*, passes each node three times: first on its left side, then underneath it (between its two subtrees), and last on its right side.

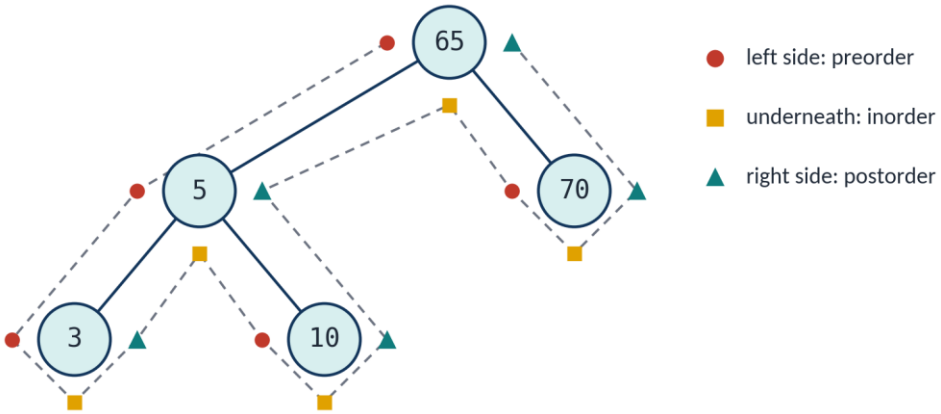


Figure 11.9 The walk around the tree touches each node three times: listing the nodes when they are passed on the left gives preorder, underneath gives inorder, and on the right gives postorder.

Listing each node the first time the walk passes it, on its left, gives preorder; listing it when the walk passes underneath gives inorder; listing it on its right, the last time, gives postorder. Following the dashed line in **Figure 11.9** and reading only the red dots gives 65, 5, 3, 10, 70; the amber squares give 3, 5, 10, 65, 70; the teal triangles give 3, 10, 5, 70, 65. The recursion of **Pseudocode 11.1** *is* this walk: each call arrives at its node from the left (line 1), goes down the left subtree and comes back underneath, goes down the right subtree and comes back on the right, and returns.

11.6.3 Level order with a queue

The three orders above are *depth-first*: they go as deep as possible down one branch before coming back. **Level order** is *breadth-first*: it visits the root, then every node on level 1 from left to right, then every node on level 2, and so on. On the tree of **Figure 11.8** it gives 65, 5, 70, 3, 10.

Level order cannot follow the recursion, because consecutive visits (70, then 3) are in different subtrees. It needs a queue, the FIFO structure of Section 5.11: the nodes waiting to be visited join at the back, and the next node to visit leaves from the front. **Pseudocode 11.2** states the algorithm.

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)
```

Figure 11.10 follows the queue on a perfect tree whose nodes are numbered level by level, so that the visit order must come out as 1, 2, ..., 7.

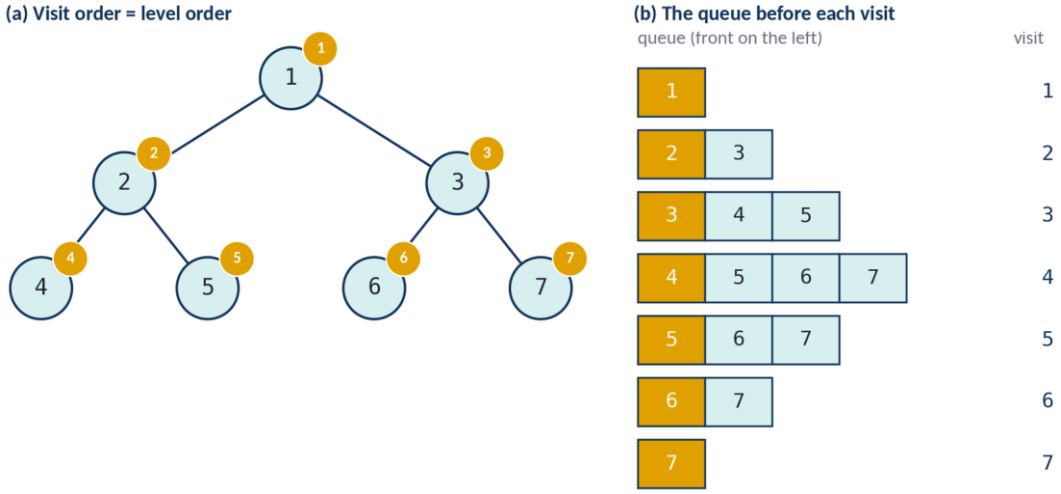


Figure 11.10 Level order on a perfect tree of seven nodes: (a) the visit order, (b) the queue just before each visit, with the node about to leave it in amber.

1. The queue starts as [1]. Dequeue 1, visit it, enqueue its children 2 and 3: [2, 3].
2. Dequeue 2, visit it, enqueue 4 and 5: [3, 4, 5].
3. Dequeue 3, visit it, enqueue 6 and 7: [4, 5, 6, 7].
4. Dequeue 4, 5, 6 and 7 in turn; they are leaves, so nothing more is enqueued, and the queue empties.

Why is the order correct? The queue always holds the unvisited nodes of at most two consecutive levels, those of the current level in front of those of the next, because every node of level $\ell + 1$ is enqueued by its parent on level ℓ, and the parents are dequeued in left-to-right order. So the nodes of each level are visited, left to right, before any node of the next level. Chapter 13 generalizes exactly this algorithm to graphs as *breadth-first search*.

11.6.4 Traversals without recursion

The recursive traversals rely on the call stack of Section 10.3 to remember where to return. That stack can be replaced by an explicit one, the technique of Section 10.13, and there are two reasons to do it. For a very deep tree (a degenerate tree of a million nodes is a chain a million calls deep) the explicit stack lives on the heap and cannot overflow the call stack (Section 10.12). And an iterative traversal can be paused and resumed, which is how the iterators of `std::map` and `TreeMap` (Section 11.11) walk a tree one step at a time.

Preorder is the easiest. Push the root; then repeatedly pop a node, visit it, and push its right child and then its left child. The left child is pushed last, so it is popped first, and the left subtree is finished before the right one is started.

Inorder needs more care, because a node must wait until its whole left subtree has been visited. **Pseudocode 11.3** keeps a pointer `cur` and a stack of nodes whose left subtrees are being traversed.

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
```

Figure 11.11 shows the stack at each visit on the tree of **Figure 11.8**. The first inner loop pushes 65, 5 and 3 and stops at 3's empty left subtree; 3 is popped and visited (a). Its right subtree is empty, so the outer loop pops 5 and visits it (b), then moves to 5's right child 10, pushes it, and pops and visits it (c). Now the stack holds only 65, which is popped and visited (d), and finally 70 is pushed and visited (e).

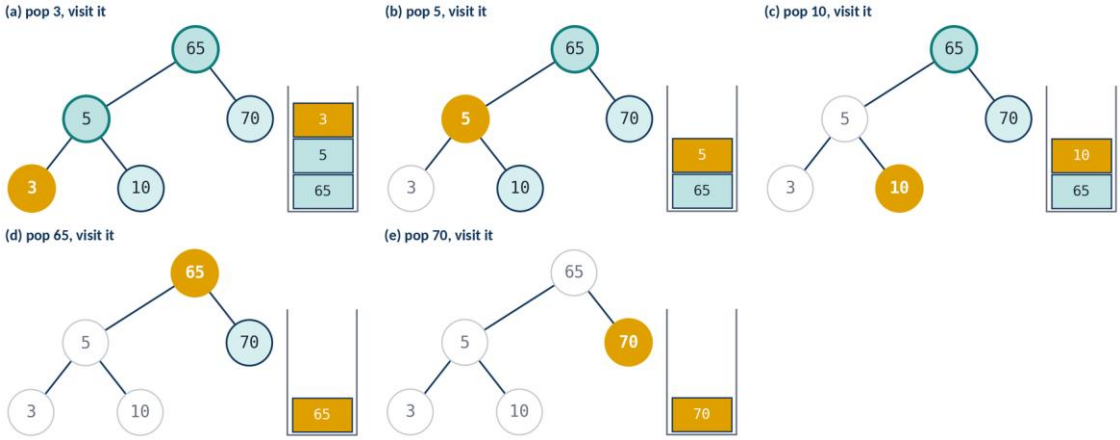


Figure 11.11 The explicit stack of **Pseudocode 11.3** at each of the five visits: teal nodes are on the stack, the amber node is being visited, and grey nodes have already been visited.

Postorder is the hardest to do with one stack, because a node must be visited after *both* its subtrees. A simple trick uses two: a modified preorder that visits root, right, left pushes every node onto a second stack, and popping the second stack then yields left, right, root, which is postorder. (A one-stack version is also possible: it remembers the last node visited, to tell whether it is coming back up from the left subtree or from the right one.) **Listing 11.2** implements all four traversals without recursion; its queue is built from linked nodes, exactly the linked queue of Section 7.10, holding pointers to tree nodes.

Listing 11.2 `traversals_iter.cpp`: preorder, inorder and postorder with an explicit stack, and level order with a queue built from linked nodes, with traces of the stack and the queue.

```cpp
// traversals_iter.cpp -- the four traversals without recursion: preorder,
// inorder and postorder with an explicit stack, level order with a linked queue.
#include <cstdio>
#include <stack>
#include <vector>

struct Node {
    int data;
    Node *left, *right;
};

Node *node(int d, Node *l = nullptr, Node *r = nullptr) {
    return new Node{d, l, r};
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

// ---- a queue of Node* built from linked nodes (Chapter 7) ----
struct QNode {
    Node *node;
    QNode *next;
};
struct Queue {
    QNode *front = nullptr, *rear = nullptr;
};
bool isEmptyQ(const Queue &q) { return q.front == nullptr; }
void enQueue(Queue &q, Node *n) {
    QNode *qn = new QNode{n, nullptr};
    if (isEmptyQ(q)) q.front = qn;
    else q.rear->next = qn;
    q.rear = qn;
}
Node *deQueue(Queue &q) {
    QNode *del = q.front;
    Node *n = del->node;
    q.front = del->next;
    if (q.front == nullptr) q.rear = nullptr;   // the queue became empty
    delete del;
    return n;
}
void showQ(const Queue &q) {
```

Listing 11.2 (Continued) *traversals_iter.cpp: preorder, inorder and postorder with an explicit stack, and level order with a queue built from linked nodes, with traces of the stack and the queue.*

```
    std::printf("[");
    for (QNode *p = q.front; p != nullptr; p = p->next)
        std::printf(" %d", p->node->data);
    std::printf(" ]");
}

void levelOrder(Node *root, bool trace) {
    if (root == nullptr) return;
    Queue q;
    enQueue(q, root);
    std::printf(trace ? "\n" : "");
    while (!isEmptyQ(q)) {
        if (trace) {
            std::printf("  queue ");
            showQ(q);
        }
        Node *n = deQueue(q);
        std::printf(trace ? "  visit %d\n" : "%d ", n->data);
        if (n->left != nullptr) enQueue(q, n->left);
        if (n->right != nullptr) enQueue(q, n->right);
    }
}

void preOrderIter(Node *root) {
    std::stack<Node *> s;
    if (root != nullptr) s.push(root);
    while (!s.empty()) {
        Node *n = s.top();
        s.pop();
        std::printf("%d ", n->data);
        if (n->right != nullptr) s.push(n->right);  // pushed first, popped later
        if (n->left != nullptr) s.push(n->left);
    }
}

void inOrderIter(Node *root, bool trace) {
    std::vector<Node *> s;              // a vector as a stack, so it can be printed
    Node *cur = root;
    std::printf(trace ? "\n" : "");
    while (cur != nullptr || !s.empty()) {
        while (cur != nullptr) {        // go as far left as possible
            s.push_back(cur);
            cur = cur->left;
        }
        cur = s.back();                 // the leftmost node not yet visited
        s.pop_back();
```

Listing 11.2 (Continued) *traversals_iter.cpp: preorder, inorder and postorder with an explicit stack, and level order with a queue built from linked nodes, with traces of the stack and the queue.*

```
        if (trace) {
            std::printf("  visit %-3d stack [", cur->data);
            for (Node *x : s) std::printf(" %d", x->data);
            std::printf(" ]\n");
        } else {
            std::printf("%d ", cur->data);
        }
        cur = cur->right;               // then its right subtree
    }
}

void postOrderIter(Node *root) {        // two stacks
    std::stack<Node *> s1, s2;
    if (root != nullptr) s1.push(root);
    while (!s1.empty()) {
        Node *n = s1.top();
        s1.pop();
        s2.push(n);                     // s2 collects root, right, left
        if (n->left != nullptr) s1.push(n->left);
        if (n->right != nullptr) s1.push(n->right);
    }
    while (!s2.empty()) {               // popping reverses it: left, right, root
        std::printf("%d ", s2.top()->data);
        s2.pop();
    }
}

int main() {
    // The tree built by insert(65, 5, 70, 3, 10).
    Node *t = node(65, node(5, node(3), node(10)), node(70));
    std::printf("PreOrder (stack):   ");
    preOrderIter(t);
    std::printf("\nInOrder (stack):    ");
    inOrderIter(t, false);
    std::printf("\nPostOrder (stacks): ");
    postOrderIter(t);
    std::printf("\nLevelOrder (queue): ");
    levelOrder(t, false);
    std::printf("\nInOrder, traced:");
    inOrderIter(t, true);
    destroy(t);
```

Listing 11.2 (Continued) *traversals_iter.cpp: preorder, inorder and postorder with an explicit stack, and level order with a queue built from linked nodes, with traces of the stack and the queue.*

```
    // A perfect tree numbered level by level, 1 to 7.
    Node *p = node(1, node(2, node(4), node(5)), node(3, node(6), node(7)));
    std::printf("LevelOrder of the tree 1..7, traced:");
    levelOrder(p, true);
    destroy(p);
    return 0;
}
```

```
PreOrder (stack):   65 5 3 10 70
InOrder (stack):    3 5 10 65 70
PostOrder (stacks): 3 10 5 70 65
LevelOrder (queue): 65 5 70 3 10
InOrder, traced:
  visit 3   stack [ 65 5 ]
  visit 5   stack [ 65 ]
  visit 10  stack [ 65 ]
  visit 65  stack [ ]
  visit 70  stack [ ]
LevelOrder of the tree 1..7, traced:
  queue [ 1 ]  visit 1
  queue [ 2 3 ]  visit 2
  queue [ 3 4 5 ]  visit 3
  queue [ 4 5 6 7 ]  visit 4
  queue [ 5 6 7 ]  visit 5
  queue [ 6 7 ]  visit 6
  queue [ 7 ]  visit 7
```

The first four lines agree with the recursive orders of **Figure 11.8**. The traced inorder run shows the stack of **Figure 11.11**, and the traced level order prints the queue of **Figure 11.10** line by line. The Java and Python versions (`TraversalsIter.java`, `traversals_iter.py`) print the same output.

11.6.5 What traversals cost

Every traversal visits each node once and does a constant amount of work per visit, plus a constant amount per empty subtree it looks at. We can prove this for the recursive versions with a recurrence. Let $T(n)$ be the time of PREORDER on a tree of n nodes whose left subtree has k nodes (so the right one has $n-k-1$). Then, for constants c and d,

$$T(0) = c, \qquad T(n) = T(k) + T(n-k-1) + d \quad (n \geq 1) \tag{11.8}$$

Claim: $T(n) = (c+d)\,n + c$, whatever the shape. For $n = 0$ this gives c, which is right. For $n \geq 1$, assume it for smaller trees (induction again): $T(n) = (c+d)k + c + (c+d)(n-k-1) + c + d = (c+d)(n-1) + 2c + d = (c+d)n + c$. So every traversal is $\Theta(n)$, which is optimal, since each of the n nodes must be visited. The same holds for inorder, postorder and level order.

Space is a different matter, and it depends on the shape. The recursive traversals and the explicit stacks hold at most one entry per level of the path being explored, so they need $O(h)$ extra space, where h is the height: $O(\lg n)$ for a balanced tree but $O(n)$ for a skewed one. Level order holds up to one full level in its queue, which is $O(w)$ for the width w of the widest level; for a perfect tree that is about $n/2$ nodes, for a chain just one.

11.6.6 Expression trees

The parse tree of **Figure 11.1** (b) is an example of an *expression tree*: a binary tree whose leaves are operands and whose internal nodes are binary operators, each applied to the values of its two subtrees. **Figure 11.12** shows the tree of `3 + 2 * 5 - 4`. Multiplication binds more tightly than addition, and addition and subtraction are performed left to right (Section 5.8.2), so the expression means $(3 + (2 \cdot 5)) - 4$, and the operator applied last, the subtraction, is at the root.

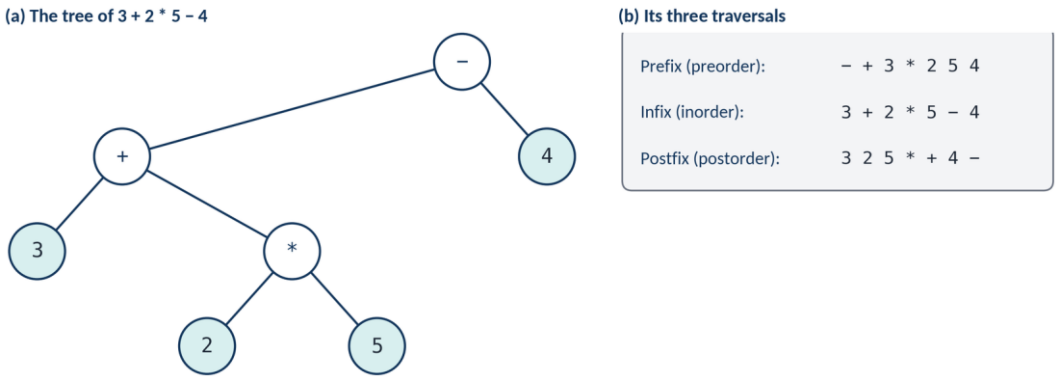


Figure 11.12 The expression tree of `3 + 2 * 5 - 4` (a) and its three depth-first traversals (b): preorder gives prefix notation, inorder gives infix notation, and postorder gives postfix notation.

The three depth-first traversals of an expression tree are exactly the three notations of Section 5.8.1:

- **Preorder** writes each operator before its operands: `- + 3 * 2 5 4`, the *prefix* (Polish) notation.
- **Inorder** writes each operator between its operands: `3 + 2 * 5 - 4`, the ordinary *infix* notation. Because infix needs precedence rules or parentheses to be unambiguous, a program that prints infix from a tree puts parentheses around every operator's subtree: `( ( 3 + ( 2 * 5 ) ) - 4 )`.
- **Postorder** writes each operator after its operands: `3 2 5 * + 4 -`, the *postfix* (reverse Polish) notation that the calculator of Section 5.9 evaluates with a stack.

Evaluating the tree is a postorder traversal: to know the value of a node, first compute the values of its two subtrees, then apply the node's operator. At the leaves the value is the operand itself.

Building the tree from postfix is the stack algorithm of Section 5.9 with trees on the stack instead of numbers. Read the postfix expression from left to right. An operand becomes a one-node tree and is pushed. An operator pops two trees, the right operand first (it was pushed last) and then the left operand, makes them the children of a new node holding the operator, and pushes that tree. At the end, exactly one tree remains, and it is the expression tree. **Pseudocode 11.4** states it, and **Figure 11.13** follows the stack for `3 2 5 * + 4 -`.

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
```

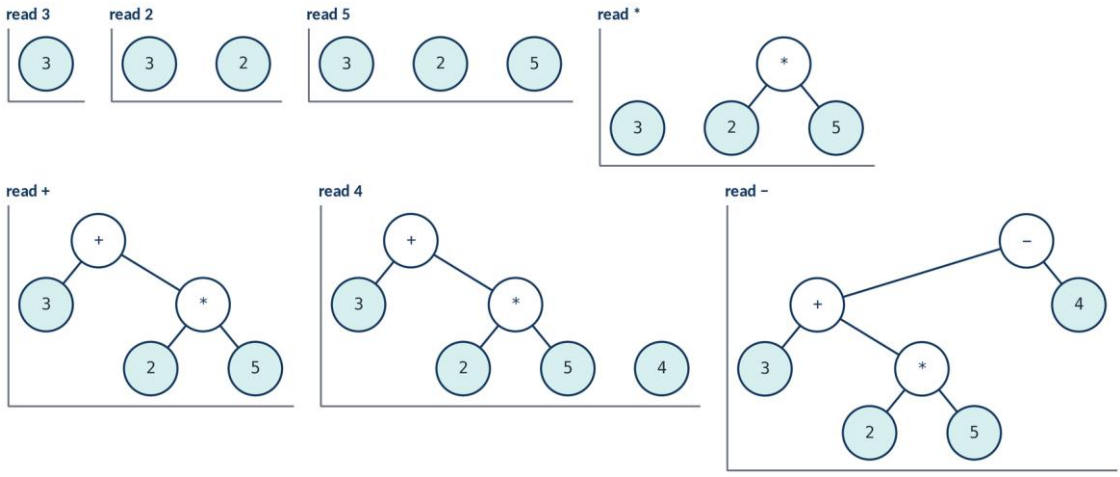


Figure 11.13 The stack of trees while `3 2 5 * + 4 -` is read: operands push one-node trees, and each operator combines the two newest trees into one.

Listing 11.3 builds the tree, prints it in the three notations and evaluates it; it also rebuilds the parse tree of **Figure 11.1** (b) from its postfix form `abc+*de+-`.

Listing 11.3 `expr_tree.cpp`: an expression tree built from postfix notation, printed in prefix, infix and postfix, and evaluated by a postorder traversal.

```
// expr_tree.cpp -- an expression tree built from postfix with a stack, printed
// in prefix, infix and postfix notation, and evaluated.
#include <cstdio>
#include <stack>
#include <string>

struct Node {
    char data;              // an operator or a one-character operand
    Node *left, *right;
};

Node *createNode(char data) { return new Node{data, nullptr, nullptr}; }

bool isOperator(char c) { return c == '+' || c == '-' || c == '*' || c == '/'; }

Node *constructExpressionTree(const std::string &postfix, bool trace) {
    std::stack<Node *> st;
    for (char c : postfix) {
        Node *newNode = createNode(c);
        if (isOperator(c)) {                // the two newest trees become children
            newNode->right = st.top();      // popped first: the RIGHT operand
            st.pop();
            newNode->left = st.top();
            st.pop();
        }
        st.push(newNode);
        if (trace)
            std::printf("  read %c: stack holds %zu tree(s)\n", c, st.size());
    }
    return st.top();                        // exactly one tree is left
}

void displayPrefix(const Node *t) {         // preorder
    if (t == nullptr) return;
    std::printf("%c ", t->data);
    displayPrefix(t->left);
    displayPrefix(t->right);
}
```

Listing 11.3 (Continued) *expr_tree.cpp: an expression tree built from postfix notation, printed in prefix, infix and postfix, and evaluated by a postorder traversal.*

```
void displayInfix(const Node *t) {          // inorder, with parentheses
    if (t == nullptr) return;
    if (isOperator(t->data)) std::printf("( ");
    displayInfix(t->left);
    std::printf("%c ", t->data);
    displayInfix(t->right);
    if (isOperator(t->data)) std::printf(") ");
}

void displayPostfix(const Node *t) {        // postorder
    if (t == nullptr) return;
    displayPostfix(t->left);
    displayPostfix(t->right);
    std::printf("%c ", t->data);
}

double evaluate(const Node *t) {            // postorder again: operands first
    if (!isOperator(t->data)) return t->data - '0';
    double a = evaluate(t->left), b = evaluate(t->right);
    switch (t->data) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        default:  return a / b;
    }
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

void show(const std::string &postfix, bool numeric) {
    std::printf("Postfix input: %s\n", postfix.c_str());
    Node *root = constructExpressionTree(postfix, numeric);
    std::printf("Prefix Notation:  ");
    displayPrefix(root);
    std::printf("\nInfix Notation:   ");
    displayInfix(root);
    std::printf("\nPostfix Notation: ");
    displayPostfix(root);
    if (numeric) std::printf("\nValue: %g", evaluate(root));
    std::printf("\n\n");
    destroy(root);
```

Listing 11.3 (Continued) *expr_tree.cpp: an expression tree built from postfix notation, printed in prefix, infix and postfix, and evaluated by a postorder traversal.*

```
}

int main() {
    show("325*+4-", true);          // 3 + 2 * 5 - 4
    show("abc+*de+-", false);       // (a * (b + c)) - (d + e)
    return 0;
}
```

```
Postfix input: 325*+4-
  read 3: stack holds 1 tree(s)
  read 2: stack holds 2 tree(s)
  read 5: stack holds 3 tree(s)
  read *: stack holds 2 tree(s)
  read +: stack holds 1 tree(s)
  read 4: stack holds 2 tree(s)
  read -: stack holds 1 tree(s)
Prefix Notation:  - + 3 * 2 5 4
Infix Notation:   ( ( 3 + ( 2 * 5 ) ) - 4 )
Postfix Notation: 3 2 5 * + 4 -
Value: 9

Postfix input: abc+*de+-
Prefix Notation:  - * a + b c + d e
Infix Notation:   ( ( a * ( b + c ) ) - ( d + e ) )
Postfix Notation: a b c + * d e + -
```

The value 9 checks by hand: $2 \cdot 5 = 10$, $3 + 10 = 13$, $13 - 4 = 9$. Compilers do the same thing on a grand scale: the parser builds a tree of the whole program, and later phases traverse it, in postorder to compute types and generate code, in preorder to print it.

11.7 The Binary Search Tree

11.7.1 The BST property

Think back to binary search on a sorted array (Section 2.5). The first probe looks at the middle element; if the key is smaller, the whole right half is thrown away, and the next probe looks at the middle of the left half. Draw every possible sequence of probes and you get a binary tree: the middle element at the root, the middle of the left half as its left child, the middle of the right half as its right child, and so on. **Figure 11.14** (b) and (c) draw it for the five keys 3, 5, 10, 65, 70. In that tree every key in the left subtree of a node is smaller than the node's key, and every key in the right subtree is larger. A *binary search tree* is exactly this tree, taken out of the array and made of nodes, so that it can grow and shrink.

> **DEFINITION.** A *binary search tree* (BST) is a binary tree in which every node x holds a key $x.key$, and for every node x: every key in the left subtree of x is less than $x.key$, and every key in the right subtree of x is greater than $x.key$.

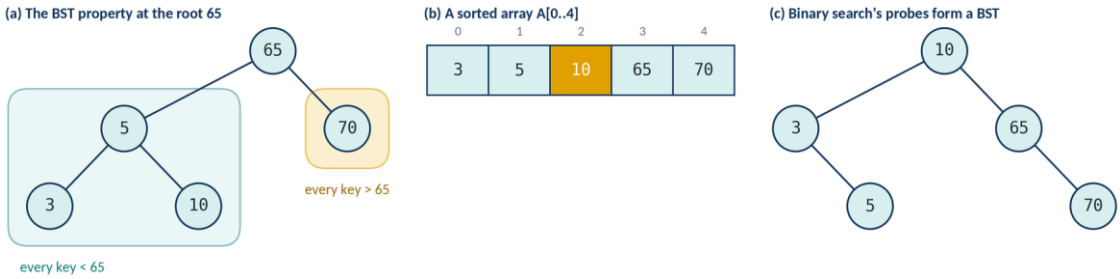


Figure 11.14 (a) The BST property at the root: every key of the left subtree is smaller than 65 and every key of the right subtree is larger; (b) and (c) the probes of binary search on a sorted array form a binary search tree.

The idea was published independently by several people around 1960, among them P. F. Windley [3], A. D. Booth and A. J. T. Colin, and T. N. Hibbard [4], all looking for a way to keep a table sorted while it grows; Knuth [8] tells the story.

Notice the words *every key in the subtree*. It is not enough that the left *child* is smaller and the right *child* is larger; the condition must hold for every descendant. Exercise 11.8 shows a tree that

passes the local test at every node and still is not a search tree. The property must also hold at *every* node, not only at the root: 5 is larger than every key in its left subtree (3) and smaller than every key in its right subtree (10).

The definition above uses strict inequalities, so a BST holds each key at most once, which is what a catalog of unique book codes or a set needs. Trees that must hold duplicates either keep a count in each node, or agree to send equal keys consistently to one side (the laundry program of Section 11.12 sends them right).

The BST property has an immediate consequence that we saw in **Figure 11.8**:

> **INORDER IS SORTED.** An inorder traversal of a binary search tree visits its keys in ascending order. *Proof* by induction on the number of nodes. The empty tree gives the empty sequence, which is sorted. For a nonempty tree with root r, inorder lists the left subtree, then r, then the right subtree. By the induction hypothesis each subtree is listed in ascending order; by the BST property every key of the first list is less than $r.key$ and every key of the last list is greater. So the concatenation is ascending.

This is why a BST is a *dictionary that stays sorted*: it answers "is this key here?" like a hash table (Chapter 15), but it can also list its keys in order, report the smallest and the largest, and find the neighbors of a key, which a hash table cannot.

11.7.2 Insertion: `insert(65, 5, 70, 3, 10)`

To insert a key z, walk down from the root as if searching for z: at each node go left if z is smaller than the node's key and right if it is larger. When the walk falls off the tree, at an empty subtree, the new node goes exactly there, as a new leaf. If the walk meets a node that already holds z, the key is a duplicate and is not inserted again. The recursive formulation is three lines:

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"
```

Line 2 must change the *pointer* that was NIL, that is, the parent's `left` or `right` field, or the root pointer itself when the tree is empty. That is why the parameter x stands for "the pointer to the subtree", not for a copy of it. C expresses this with a pointer to a pointer, `Node **root`, exactly the technique of `init(Student **p)` in Section 7.4.4; C++ can also use a reference, `Node *&root`; and Java and Python, which have neither, write the function so that it *returns* the root of the subtree after the insertion and let the caller store it: `t.left = insert(t.left, z)`. All three styles appear in this chapter's programs.

Now the running example. Starting from an empty tree, insert 65, 5, 70, 3 and 10, in that order; **Figure 11.15** shows the tree after each call, with the path followed in teal and amber and the new node in amber.

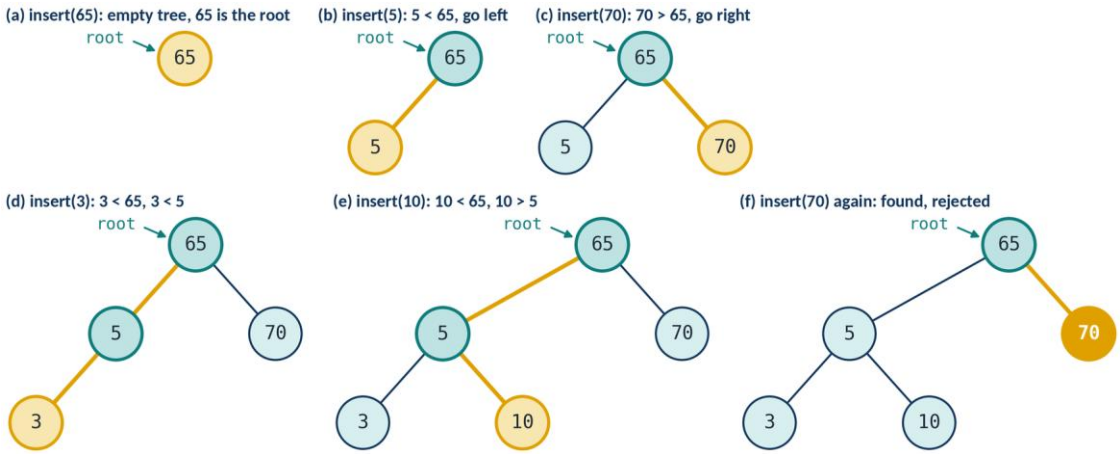


Figure 11.15 The tree after each call of `insert(65, 5, 70, 3, 10)` and a repeated `insert(70)`: teal nodes and amber edges are the path of the search, the amber node is the new leaf, and the duplicate 70 is found and rejected.

1. `insert(65)` (a). The tree is empty, so `root` is NIL: line 2 creates the node 65, and it becomes the root.
2. `insert(5)` (b). $5 < 65$, so go left. The left subtree of 65 is empty: 5 becomes the left child of 65.
3. `insert(70)` (c). $70 > 65$, go right; the right subtree is empty: 70 becomes the right child of 65.
4. `insert(3)` (d). $3 < 65$, go left to 5; $3 < 5$, go left; empty: 3 becomes the left child of 5.
5. `insert(10)` (e). $10 < 65$, go left to 5; $10 > 5$, go right; empty: 10 becomes the right child of 5.
6. `insert(70)` **again** (f). $70 > 65$, go right to 70; $70 = 70$: line 8 reports that 70 already exists, and the tree does not change.

Check the BST property on the result: every key below 5 on the left is smaller than 5 (3), on the right larger (10); everything under 65 on the left (3, 5, 10) is smaller than 65, and on the right (70) larger. The inorder traversal is 3, 5, 10, 65, 70, as it must be.

The same keys in another order give another tree. Inserting 10, 5, 3, 65, 70 would make 10 the root. The shape of a BST records the *history* of its insertions, and Section 11.8 shows how much that matters.

Insertion can also be written without recursion, with a pointer that walks down and a *trailing* pointer to its parent, as `deleteKey` trailed `prev` behind `cur` in Section 7.6.5. The height experiment of Section 11.8.2 uses that form, with the pointer-to-the-link technique of Section 7.7.3, because a recursive insertion into a degenerate tree of 32,000 nodes would nest 32,000 calls.

Cost. The walk visits one node per level, so an insertion into a tree of height h makes at most $h + 1$ comparisons and does $O(1)$ other work: $O(h)$ time.

11.7.3 Search

Search is insertion without the insert. At each node, compare the key with the node's key: equal means found; smaller means the key can only be in the left subtree, so go left; larger means go right. Reaching NIL means the key is not in the tree. **Pseudocode 11.6** gives the recursive and the iterative form; the iterative one needs no stack, because each step moves to a single child and never has to come back.

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
```

Each comparison throws away an entire subtree, the same way each probe of binary search throws away half of the array. **Figure 11.16** traces two searches on the tree built by inserting 6, 3, 10, 1, 5, 7, 8, 9.

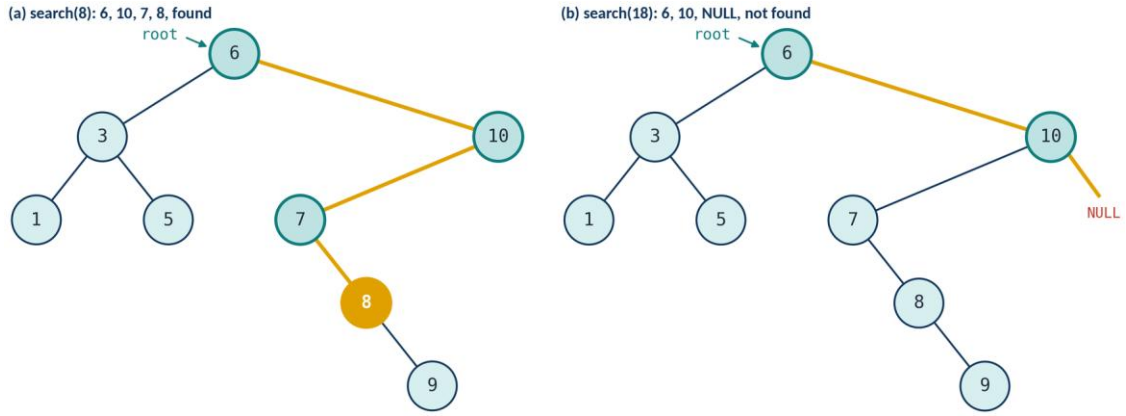


Figure 11.16 Two searches: (a) `search(8)` follows 6, 10, 7 and finds 8; (b) `search(18)` follows 6 and 10, falls off the tree at the empty right subtree of 10, and reports that 18 is absent.

- `search(8)`. At the root, 6: $8 > 6$, so 8 can only be in the right subtree; go right. At 10: $8 < 10$, go left. At 7: $8 > 7$, go right. At 8: equal, found. Four comparisons, although the tree has eight nodes; the subtree rooted at 3 and the empty right subtree of 10 were never looked at.
- `search(18)`. At 6: $18 > 6$, go right. At 10: $18 > 10$, go right. The right subtree of 10 is empty: 18 is not in the tree. Two comparisons were enough to prove it.

Cost. One comparison per level on a single root-to-leaf path: $O(h)$ in the worst case, where the worst case is an unsuccessful search that goes all the way down the longest path.

11.7.4 Minimum and maximum

The smallest key of a BST is in its *leftmost* node: starting from the root, follow `left` pointers until the next one is NIL. Why? Any node with a left child has a smaller key in its left subtree, so it cannot be the minimum; the leftmost node has no left child, and every other node is either one of its ancestors (whose keys are larger, since the leftmost node is in their left subtrees) or in the right subtree of one of them (larger still). Symmetrically, the largest key is in the rightmost node. **Figure 11.17** shows both walks.

Pseudocode 11.7 The minimum is the leftmost node and the maximum the rightmost node of a nonempty tree.

```
TREE-MINIMUM(x):                    TREE-MAXIMUM(x):
 1.  while x.left ≠ NIL              1.  while x.right ≠ NIL
 2.      x ← x.left                  2.      x ← x.right
 3.  return x                        3.  return x
```

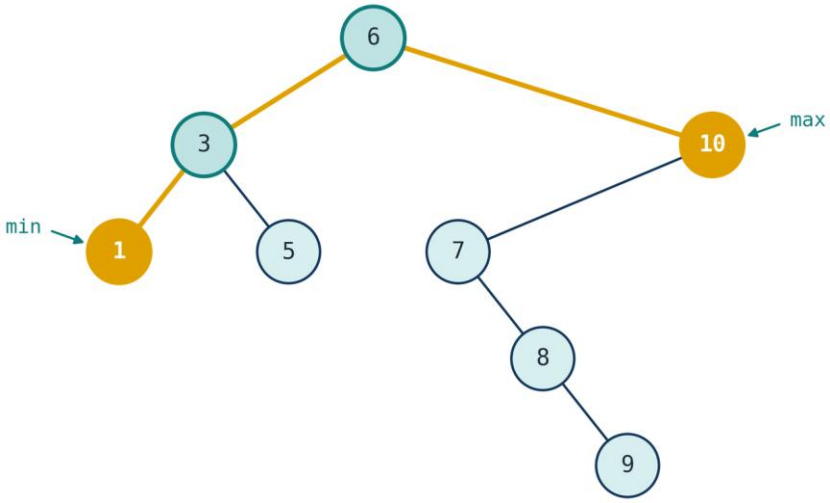


Figure 11.17 The minimum, 1, is reached by going left from the root until there is no left child; the maximum, 10, by going right until there is no right child.

Note the maximum, 10, in **Figure 11.17**: it is not a leaf. It has a left subtree (7, 8, 9), but no right child, and that is all the maximum needs. Both walks follow one path: $O(h)$.

11.7.5 Counting nodes, leaves and height

Three quantities follow the recursive definition directly, each a postorder computation: first compute the answer for the two subtrees, then combine.

- The **number of nodes** of an empty tree is 0; otherwise it is the number in the left subtree plus the number in the right subtree plus 1 for the root.
- The **number of leaves** of an empty tree is 0; a single node with no children is 1 leaf; otherwise it is the sum over the two subtrees.
- The **height** of an empty tree is -1; otherwise it is one more than the larger of the two subtrees' heights:

$$height(\text{NIL}) = -1, \qquad height(x) = 1 + \max(height(x.left),\ height(x.right)) \tag{11.9}$$

The convention $height(\text{NIL}) = -1$ is what makes Equation (11.9) right for a leaf: $1 + \max(-1, -1) = 0$. With the other common convention, height counted in *nodes* rather than edges, the empty tree has height 0 and a leaf height 1; the recursion is the same with 0 in the base case, and every answer is one larger.

Trace Equation (11.9) on the tree of `insert(65, 5, 70, 3, 10)`: the leaves 3, 10 and 70 have height 0; node 5 has height $1 + \max(0,0) = 1$; the root has height $1 + \max(1,0) = 2$. On the tree of **Figure 11.16**, the longest path is 6, 10, 7, 8, 9, four edges, so the height is 4.

All three functions visit every node once: $\Theta(n)$ each, by Equation (11.8). A function that prints the leaves, from left to right, is the same walk with a print instead of a count.

11.7.6 Printing a tree sideways

A terminal prints text line by line, from the top down, which is awkward for a tree that grows downward. A simple trick is to print the tree rotated 90 degrees counterclockwise, with the root on the left and each level one indentation step further right. The rightmost node must then come out on the *first* line, so the traversal is a *reverse inorder*: right subtree, node, left subtree, each node indented by four spaces per level of depth.

Pseudocode 11.8 Printing a tree sideways: reverse inorder, indented by depth.

```
PRINT-SIDEWAYS(x, depth):
 1.  if x ≠ NIL
 2.      PRINT-SIDEWAYS(x.right, depth + 1)
 3.      print 4 · depth spaces, then x.key
 4.      PRINT-SIDEWAYS(x.left, depth + 1)
```

To read the output, tilt your head to the left: the column on the left is the root, the next column is level 1, and so on; each node's right child is the nearest line above it one column further right, and its left child the nearest line below it. Section 11.7.7 shows the result for both example trees.

11.7.7 The complete program

Listing 11.4 puts Sections 11.7.2 to 11.7.6 together. Its `insert` receives the address of the pointer to change, `Node **root`, as `insert(&(*root)->left, newData)` shows: `(*root)->left` is the left pointer field of the current node, and `&` takes its address, so the recursive call can store the new node into the parent's field. Its `main` builds the running example, including the rejected duplicate 70, prints all four traversals and all the counts, prints the tree sideways, and releases it; then it builds the tree of **Figure 11.16** and runs the two searches of that figure, printing each decision.

Listing 11.4 `bst.cpp`: a binary search tree with insertion, the four traversals, search, count, leaf count, height, minimum, maximum, leaf listing, sideways printing, and release of all nodes.

```
// bst.cpp -- a binary search tree: insert, traversals, search, count, height,
// minimum, maximum, leaves, printing sideways, and clearing.
#include <cstdio>
#include <queue>

typedef struct MyNode {
    int data;
    struct MyNode *left;    // subtree of smaller keys
    struct MyNode *right;   // subtree of larger keys
} Node;

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);
    }
}

void preOrder(const Node *root) {           // root, left, right
    if (root != nullptr) {
        std::printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

void inOrder(const Node *root) {            // left, root, right
    if (root != nullptr) {
        inOrder(root->left);
        std::printf("%d ", root->data);
        inOrder(root->right);
    }
}
```

Listing 11.4 (Continued) *bst.cpp: a binary search tree with insertion, the four traversals, search, count, leaf count, height, minimum, maximum, leaf listing, sideways printing, and release of all nodes.*

```
void postOrder(const Node *root) {          // left, right, root
    if (root != nullptr) {
        postOrder(root->left);
        postOrder(root->right);
        std::printf("%d ", root->data);
    }
}

void levelOrder(Node *root) {               // breadth first, with a queue
    if (root == nullptr) {
        std::printf("Tree is empty.");
        return;
    }
    std::queue<Node *> q;
    q.push(root);
    while (!q.empty()) {
        Node *n = q.front();
        q.pop();
        std::printf("%d ", n->data);                    // visit the node
        if (n->left != nullptr) q.push(n->left);        // children join the back
        if (n->right != nullptr) q.push(n->right);
    }
}

Node *search(Node *root, int data) {        // prints the path it takes
    if (root == nullptr) {
        std::printf("  reached NULL: %d cannot be found by search()\n", data);
        return nullptr;
    } else if (data < root->data) {
        std::printf("  %d < %d, go left\n", data, root->data);
        return search(root->left, data);
    } else if (data > root->data) {
        std::printf("  %d > %d, go right\n", data, root->data);
        return search(root->right, data);
    } else {
        std::printf("  %d has been found by search()\n", data);
        return root;
    }
}

int count(const Node *root) {
    if (root == nullptr) return 0;
    return count(root->left) + count(root->right) + 1;
}
```

Listing 11.4 (Continued) *bst.cpp: a binary search tree with insertion, the four traversals, search, count, leaf count, height, minimum, maximum, leaf listing, sideways printing, and release of all nodes.*

```
int countLeaves(const Node *root) {
    if (root == nullptr) return 0;
    if (root->left == nullptr && root->right == nullptr) return 1;
    return countLeaves(root->left) + countLeaves(root->right);
}

int height(const Node *root) {              // edges on the longest downward path
    if (root == nullptr) return -1;         // the empty tree has height -1
    int u = height(root->left), v = height(root->right);
    return (u > v ? u : v) + 1;
}

Node *min(Node *root) {                     // the leftmost node
    if (root == nullptr) return nullptr;
    if (root->left == nullptr) return root;
    return min(root->left);
}

Node *max(Node *root) {                     // the rightmost node
    if (root == nullptr) return nullptr;
    if (root->right == nullptr) return root;
    return max(root->right);
}

void leaf(const Node *root) {               // prints the leaves, left to right
    if (root == nullptr) return;
    if (root->left == nullptr && root->right == nullptr)
        std::printf("%d ", root->data);
    leaf(root->left);
    leaf(root->right);
}

void printSideways(const Node *root, int depth) {   // right subtree on top
    if (root == nullptr) return;
    printSideways(root->right, depth + 1);
    std::printf("%*s%d\n", 4 * depth, "", root->data);
    printSideways(root->left, depth + 1);
}
```

Listing 11.4 (Continued) *bst.cpp: a binary search tree with insertion, the four traversals, search, count, leaf count, height, minimum, maximum, leaf listing, sideways printing, and release of all nodes.*

```
int clear(Node **root) {                    // postorder: children before parent
    if (*root == nullptr) return 0;
    int freed = clear(&(*root)->left) + clear(&(*root)->right);
    delete *root;
    *root = nullptr;
    return freed + 1;
}

void report(Node *root) {
    std::printf("PreOrder:   ");
    preOrder(root);
    std::printf("\nInOrder:    ");
    inOrder(root);
    std::printf("\nPostOrder:  ");
    postOrder(root);
    std::printf("\nLevelOrder: ");
    levelOrder(root);
    std::printf("\nNodes: %d, leaves: %d, height: %d, min: %d, max: %d\n",
                count(root), countLeaves(root), height(root),
                min(root)->data, max(root)->data);
    std::printf("Leaves: ");
    leaf(root);
    std::printf("\nThe tree, printed sideways (root on the left):\n");
    printSideways(root, 0);
}

int main() {
    std::printf("Create a root, it's an empty node...\n");
    Node *root = nullptr;
    int first[] = {65, 5, 70, 3, 10, 70};
    for (int x : first) {
        std::printf("Insert a node %d ... ", x);
        insert(&root, x);
    }
    std::printf("\n");
    report(root);
    std::printf("Clear the tree ... %d nodes deleted\n\n", clear(&root));
```

Listing 11.4 (Continued) *bst.cpp: a binary search tree with insertion, the four traversals, search, count, leaf count, height, minimum, maximum, leaf listing, sideways printing, and release of all nodes.*

```
    std::printf("Build a tree ...\n");
    int second[] = {6, 3, 10, 1, 5, 7, 8, 9};
    for (int x : second) insert(&root, x);
    std::printf("\n");
    report(root);
    std::printf("Searching 8 ...\n");
    search(root, 8);
    std::printf("Searching 18 ...\n");
    search(root, 18);
    std::printf("Clear the tree ... %d nodes deleted\n", clear(&root));
    return 0;
}
```

```
Create a root, it's an empty node...
Insert a node 65 ... 65 has been inserted by insert()
Insert a node 5 ... 5 has been inserted by insert()
Insert a node 70 ... 70 has been inserted by insert()
Insert a node 3 ... 3 has been inserted by insert()
Insert a node 10 ... 10 has been inserted by insert()
Insert a node 70 ... 70 has existed in the tree

PreOrder:   65 5 3 10 70
InOrder:    3 5 10 65 70
PostOrder:  3 10 5 70 65
LevelOrder: 65 5 70 3 10
Nodes: 5, leaves: 3, height: 2, min: 3, max: 70
Leaves: 3 10 70
The tree, printed sideways (root on the left):
    70
65
        10
    5
        3
Clear the tree ... 5 nodes deleted

Build a tree ...
6 has been inserted by insert()
3 has been inserted by insert()
10 has been inserted by insert()
1 has been inserted by insert()
5 has been inserted by insert()
7 has been inserted by insert()
8 has been inserted by insert()
9 has been inserted by insert()

PreOrder:   6 3 1 5 10 7 8 9
InOrder:    1 3 5 6 7 8 9 10
PostOrder:  1 5 3 9 8 7 10 6
LevelOrder: 6 3 10 1 5 7 8 9
Nodes: 8, leaves: 3, height: 4, min: 1, max: 10
Leaves: 1 5 9
The tree, printed sideways (root on the left):
    10
            9
        8
        7
6
        5
    3
        1
Searching 8 ...
  8 > 6, go right
  8 < 10, go left
  8 > 7, go right
  8 has been found by search()
Searching 18 ...
  18 > 6, go right
  18 > 10, go right
  reached NULL: 18 cannot be found by search()
Clear the tree ... 8 nodes deleted
```

Read the output against the figures. The depth-first traversals of the first tree are those of **Figure 11.8**, and its level order is 65 5 70 3 10; the counts are 5 nodes, 3 leaves (3, 10, 70) and height 2; the sideways print, tilted, is **Figure 11.15** (e). For the second tree, the level order 6 3 10 1 5 7 8 9 is the tree of **Figure 11.16** read level by level; its height, 4, is the path 6, 10, 7, 8, 9; and the two search traces are exactly the paths drawn in the figure.

Two details deserve a closer look. First, `clear` must be a *postorder* traversal: it releases both subtrees before it deletes the node itself, because once the node is deleted its `left` and `right` fields can no longer be read. It then sets the caller's pointer to `nullptr`, so the tree is a valid empty tree again. A `clear` that merely sets `root = nullptr` without deleting anything compiles, runs and prints the right thing, and leaks every node; Valgrind reports "All heap blocks were freed – no leaks are possible" for **Listing 11.4**, which is the proof that ours does not. Second, `min(root)->data` assumes the tree is not empty; on an empty tree `min` returns `nullptr` and the `->` would be undefined behavior (Section 7.11), so `report` is called only on nonempty trees.

11.7.8 The same tree in Java and Python

Java and Python have no pointer to a pointer, so their `insert` returns the root of the subtree it was given, and the caller stores the result: `root = insert(root, x)` at the top, `t.left = insert(t.left, newData)` inside. If the subtree was empty, the returned root is the new node; otherwise it is the same node as before, and the assignment changes nothing. **Listing 11.5** is the Java version; `clear` simply drops the reference to the root, and the garbage collector reclaims every node that has become unreachable.

Listing 11.5 `Bst.java`: the binary search tree of **Listing 11.4** in Java; `insert` returns the new root of each subtree.

```java
// Bst.java -- a binary search tree: insert, traversals, search, count, height,
// minimum, maximum, leaves, printing sideways, and clearing.
import java.util.ArrayDeque;

public class Bst {
    static class Node {
        int data;
        Node left, right;   // subtrees of smaller and of larger keys

        Node(int data) {
            this.data = data;
        }
    }

    private Node root;      // null means the tree is empty

    void insert(int newData) {
        root = insert(root, newData);
    }

    // Returns the root of the subtree after the insertion (Java has no Node **).
    private Node insert(Node t, int newData) {
        if (t == null) {
            System.out.println(newData + " has been inserted by insert()");
            return new Node(newData);
        } else if (newData < t.data) {
            t.left = insert(t.left, newData);
        } else if (newData > t.data) {
            t.right = insert(t.right, newData);
        } else {
            System.out.println(newData + " has existed in the tree");
        }
        return t;
    }

    static void preOrder(Node t) {
        if (t != null) {
            System.out.print(t.data + " ");
            preOrder(t.left);
            preOrder(t.right);
        }
    }
```

Listing 11.5 (Continued) *Bst.java: the binary search tree of Listing 11.4 in Java; insert returns the new root of each subtree.*

```
    static void inOrder(Node t) {
        if (t != null) {
            inOrder(t.left);
            System.out.print(t.data + " ");
            inOrder(t.right);
        }
    }

    static void postOrder(Node t) {
        if (t != null) {
            postOrder(t.left);
            postOrder(t.right);
            System.out.print(t.data + " ");
        }
    }

    static void levelOrder(Node t) {
        if (t == null) {
            System.out.print("Tree is empty.");
            return;
        }
        ArrayDeque<Node> q = new ArrayDeque<>();
        q.add(t);
        while (!q.isEmpty()) {
            Node n = q.remove();
            System.out.print(n.data + " ");
            if (n.left != null) q.add(n.left);
            if (n.right != null) q.add(n.right);
        }
    }

    static Node search(Node t, int data) {
        if (t == null) {
            System.out.println("  reached NULL: " + data
                    + " cannot be found by search()");
            return null;
        } else if (data < t.data) {
            System.out.println("  " + data + " < " + t.data + ", go left");
            return search(t.left, data);
        } else if (data > t.data) {
            System.out.println("  " + data + " > " + t.data + ", go right");
            return search(t.right, data);
        } else {
            System.out.println("  " + data + " has been found by search()");
            return t;
        }
```

Listing 11.5 (Continued) *Bst.java: the binary search tree of Listing 11.4 in Java; insert returns the new root of each subtree.*

```
    }

    static int count(Node t) {
        if (t == null) return 0;
        return count(t.left) + count(t.right) + 1;
    }

    static int countLeaves(Node t) {
        if (t == null) return 0;
        if (t.left == null && t.right == null) return 1;
        return countLeaves(t.left) + countLeaves(t.right);
    }

    static int height(Node t) {
        if (t == null) return -1;
        return Math.max(height(t.left), height(t.right)) + 1;
    }

    static Node min(Node t) {
        if (t == null) return null;
        return t.left == null ? t : min(t.left);
    }

    static Node max(Node t) {
        if (t == null) return null;
        return t.right == null ? t : max(t.right);
    }

    static void leaf(Node t) {
        if (t == null) return;
        if (t.left == null && t.right == null) System.out.print(t.data + " ");
        leaf(t.left);
        leaf(t.right);
    }

    static void printSideways(Node t, int depth) {
        if (t == null) return;
        printSideways(t.right, depth + 1);
        System.out.println(" ".repeat(4 * depth) + t.data);
        printSideways(t.left, depth + 1);
    }
```

Listing 11.5 (Continued) *Bst.java: the binary search tree of Listing 11.4 in Java; insert returns the new root of each subtree.*

```
    int clear() {                   // the garbage collector frees the nodes
        int n = count(root);
        root = null;
        return n;
    }

    void report() {
        System.out.print("PreOrder:   ");
        preOrder(root);
        System.out.print("\nInOrder:    ");
        inOrder(root);
        System.out.print("\nPostOrder:  ");
        postOrder(root);
        System.out.print("\nLevelOrder: ");
        levelOrder(root);
        System.out.printf("%nNodes: %d, leaves: %d, height: %d, min: %d, max: %d%n",
                count(root), countLeaves(root), height(root), min(root).data,
                max(root).data);
        System.out.print("Leaves: ");
        leaf(root);
        System.out.println("\nThe tree, printed sideways (root on the left):");
        printSideways(root, 0);
    }

    public static void main(String[] args) {
        System.out.println("Create a root, it's an empty node...");
        Bst tree = new Bst();
        for (int x : new int[] {65, 5, 70, 3, 10, 70}) {
            System.out.print("Insert a node " + x + " ... ");
            tree.insert(x);
        }
        System.out.println();
        tree.report();
        System.out.println("Clear the tree ... " + tree.clear()
                + " nodes deleted\n");
```

Listing 11.5 (Continued) *Bst.java: the binary search tree of Listing 11.4 in Java; insert returns the new root of each subtree.*

```
        System.out.println("Build a tree ...");
        for (int x : new int[] {6, 3, 10, 1, 5, 7, 8, 9}) tree.insert(x);
        System.out.println();
        tree.report();
        System.out.println("Searching 8 ...");
        search(tree.root, 8);
        System.out.println("Searching 18 ...");
        search(tree.root, 18);
        System.out.println("Clear the tree ... " + tree.clear() + " nodes deleted");
    }
}
```

Listing 11.6 is the Python version. The traversals append to a list instead of printing directly, which keeps the printing in one place; `__slots__` fixes the three attributes of a node, as in Section 7.5.9.

Listing 11.6 `bst.py`: the same binary search tree in Python.

```
# bst.py -- a binary search tree: insert, traversals, search, count, height,
# minimum, maximum, leaves, printing sideways, and clearing.
from collections import deque


class Node:
    __slots__ = ("data", "left", "right")

    def __init__(self, data):
        self.data = data
        self.left = None        # subtree of smaller keys
        self.right = None       # subtree of larger keys


def insert(t, new_data):
    """Returns the root of the subtree after the insertion."""
    if t is None:
        print(f"{new_data} has been inserted by insert()")
        return Node(new_data)
    if new_data < t.data:
        t.left = insert(t.left, new_data)
    elif new_data > t.data:
        t.right = insert(t.right, new_data)
    else:
        print(f"{new_data} has existed in the tree")
    return t


def pre_order(t, out):          # root, left, right
    if t is not None:
        out.append(t.data)
        pre_order(t.left, out)
        pre_order(t.right, out)


def in_order(t, out):           # left, root, right
    if t is not None:
        in_order(t.left, out)
        out.append(t.data)
        in_order(t.right, out)
```

Listing 11.6 (Continued) *bst.py: the same binary search tree in Python.*

```
def post_order(t, out):         # left, right, root
    if t is not None:
        post_order(t.left, out)
        post_order(t.right, out)
        out.append(t.data)


def level_order(t, out):        # breadth first, with a queue
    if t is None:
        return
    q = deque([t])
    while q:
        n = q.popleft()
        out.append(n.data)
        if n.left is not None:
            q.append(n.left)
        if n.right is not None:
            q.append(n.right)


def search(t, data):
    if t is None:
        print(f"  reached NULL: {data} cannot be found by search()")
        return None
    if data < t.data:
        print(f"  {data} < {t.data}, go left")
        return search(t.left, data)
    if data > t.data:
        print(f"  {data} > {t.data}, go right")
        return search(t.right, data)
    print(f"  {data} has been found by search()")
    return t


def count(t):
    return 0 if t is None else count(t.left) + count(t.right) + 1


def count_leaves(t):
    if t is None:
        return 0
    if t.left is None and t.right is None:
        return 1
    return count_leaves(t.left) + count_leaves(t.right)
```

Listing 11.6 (Continued) *bst.py: the same binary search tree in Python.*

```
def height(t):                  # the empty tree has height -1
    return -1 if t is None else max(height(t.left), height(t.right)) + 1


def tree_min(t):                # the leftmost node
    if t is None:
        return None
    return t if t.left is None else tree_min(t.left)


def tree_max(t):                # the rightmost node
    if t is None:
        return None
    return t if t.right is None else tree_max(t.right)


def leaves(t, out):
    if t is None:
        return
    if t.left is None and t.right is None:
        out.append(t.data)
    leaves(t.left, out)
    leaves(t.right, out)


def print_sideways(t, depth=0):
    if t is None:
        return
    print_sideways(t.right, depth + 1)
    print(" " * (4 * depth) + str(t.data))
    print_sideways(t.left, depth + 1)


def show(label, walk, t):
    out = []
    walk(t, out)
    print(label + "".join(f"{x} " for x in out))
```

Listing 11.6 (Continued) *bst.py: the same binary search tree in Python.*

```
def report(root):
    show("PreOrder:   ", pre_order, root)
    show("InOrder:    ", in_order, root)
    show("PostOrder:  ", post_order, root)
    show("LevelOrder: ", level_order, root)
    print(f"Nodes: {count(root)}, leaves: {count_leaves(root)}, "
          f"height: {height(root)}, min: {tree_min(root).data}, "
          f"max: {tree_max(root).data}")
    show("Leaves: ", leaves, root)
    print("The tree, printed sideways (root on the left):")
    print_sideways(root)


def main():
    print("Create a root, it's an empty node...")
    root = None
    for x in (65, 5, 70, 3, 10, 70):
        print(f"Insert a node {x} ... ", end="")
        root = insert(root, x)
    print()
    report(root)
    print(f"Clear the tree ... {count(root)} nodes deleted\n")
    root = None                 # the garbage collector frees the nodes

    print("Build a tree ...")
    for x in (6, 3, 10, 1, 5, 7, 8, 9):
        root = insert(root, x)
    print()
    report(root)
    print("Searching 8 ...")
    search(root, 8)
    print("Searching 18 ...")
    search(root, 18)
    print(f"Clear the tree ... {count(root)} nodes deleted")


if __name__ == "__main__":
    main()
```

Both programs print exactly the output of **Listing 11.4**. One warning applies to the Python version in particular: its recursion depth equals the height of the tree, and CPython's default recursion limit is 1,000 (Section 10.12.3). Inserting 2,000 keys in sorted order into this `insert` builds a chain of height 1,999 and raises `RecursionError`. Section 11.8 explains why such trees arise and how to avoid them; the iterative insertion of `height_experiment.py` has no such limit.

11.7.9 Successor and predecessor

The *successor* of a key x in a BST is the smallest key larger than x: the key that comes right after x in the inorder sequence. The *predecessor* is the largest key smaller than x. On the tree of **Figure 11.18**, whose inorder sequence is 20, 30, 35, 40, 45, 50, 60, 65, 70, 80, the successor of 30 is 35 and the successor of 45 is 50. Both can be found in $O(h)$ time without looking at the other keys, and there are two cases.

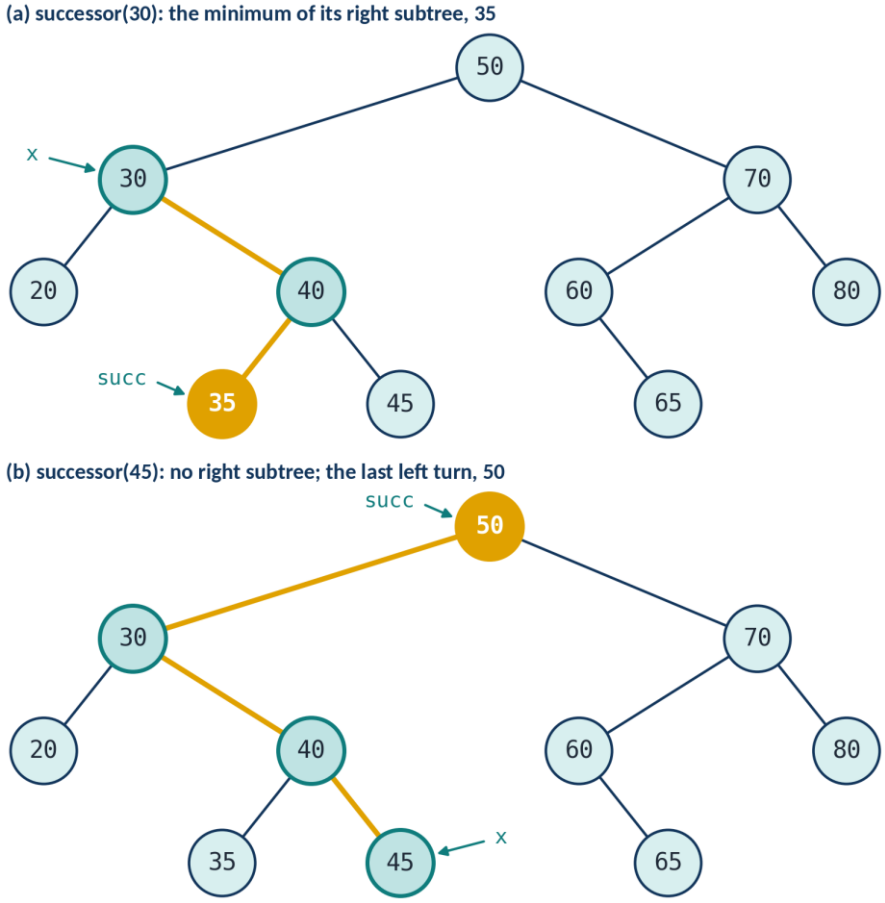


Figure 11.18 The two cases of the successor: (a) 30 has a right subtree, and its successor is that subtree's minimum, 35; (b) 45 has no right subtree, and its successor is the ancestor at which the path from the root last turned left, 50.

- **Case 1: x has a right subtree.** Every key larger than x that is close to x lies in that subtree, and the smallest of them is its minimum: go right once, then left as far as possible. In **Figure 11.18** (a), 30's right subtree is rooted at 40, whose leftmost node is 35.
- **Case 2: x has no right subtree.** Then the successor is above x. Walk up the path from x toward the root: while you arrive at a node from its right child, that node is smaller than x (you came from its right subtree); the first time you arrive at a node from its *left* child, that node is larger than x and is the successor. Seen from the root, it is the last node on the search path at which the path *turned left*. In **Figure 11.18** (b), the path to 45 goes 50 (turn left), 30 (turn right), 40 (turn right), 45; the last left turn was at 50.
- If x has no right subtree and the path never turned left, x is the maximum and has no successor.

Our nodes have no pointer to their parent, so we cannot walk up. Instead we walk *down*, remembering the last node at which we turned left, as **Pseudocode 11.9** does. The predecessor is the mirror image: the maximum of the left subtree if there is one, otherwise the last node at which the search path turned right.

Pseudocode 11.9 `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)
```

Trees that store a `parent` pointer in each node (as the red-black trees of `std::map` and `TreeMap` do) can find the successor of a node they already hold by walking up, without a search from the root; Cormen et al. [2] give that version. Either way the cost is $O(h)$. Walking a whole tree with repeated successor calls, from the minimum on, costs only $\Theta(n)$ in total, because each edge is crossed at most twice; that is how the `++it` of a `std::map` iterator works.

11.7.10 Deletion: three cases

Deletion is the one BST operation with real subtlety, because the node to be deleted may have children that must stay in the tree, and the BST property must still hold afterward. First search for the node z holding the key. Then there are three cases, according to how many children z has; **Figure 11.19** applies them in turn to the running example.

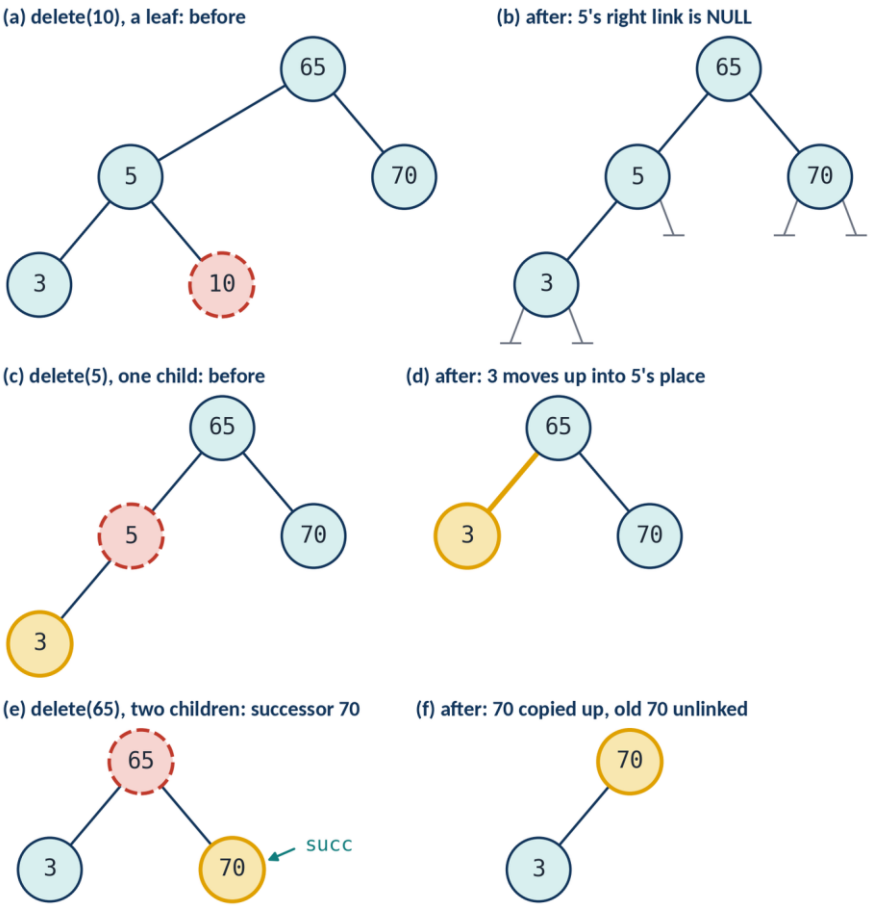


Figure 11.19 The three cases of deletion, applied one after the other to the tree of `insert(65, 5, 70, 3, 10)`: a leaf is unlinked (a, b); a node with one child is replaced by that child (c, d); a node with two children takes the key of its successor, which is then deleted from the right subtree (e, f).

Case 1: z is a leaf. Nothing hangs below z, so simply set its parent's pointer to z to NIL and release the node. Deleting 10, the right child of 5 (**Figure 11.19** (a)), makes 5's right pointer NIL (b). The inorder sequence loses 10 and stays sorted.

Case 2: z has exactly one child. Splice z out: make z's parent point directly to z's only child, and release z. The child moves up one level, taking its whole subtree with it. The BST property survives, because every key in that subtree was already on the same side of every ancestor of z as z itself. Deleting 5, which now has only the left child 3 ((c)), makes 65's left pointer point to 3 ((d)).

Case 3: z has two children. Now neither child can simply take z's place, because there would be nowhere to hang the other one. The trick, due to Hibbard [4], is to find a key that can *replace* z's key without breaking the order: the successor of z, the minimum of its right subtree. It is larger than everything in z's left subtree and smaller than everything else in the right subtree, so it fits exactly where z is. Copy the successor's key into z, and then delete the successor from the right subtree. That second deletion is always easy, because the successor, being the leftmost node of the right subtree, has *no left child*: it is case 1 or case 2. Deleting 65 ((e)): its successor is 70, the minimum of its right subtree; 70 is copied into the root, and the old 70, a leaf, is unlinked ((f)).

Pseudocode 11.10 states the procedure in the recursive style that returns the new root of the subtree, the style of the Java and Python `insert`.

Pseudocode 11.10 `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
```

The recursion does the parent bookkeeping for us: each call returns the root of its subtree *after* the deletion, and the caller stores it into the pointer it followed (lines 4 and 6). When a node is removed, the call that removed it returns NIL or the surviving child, and that value lands in the parent's pointer. When the root itself is deleted, the top-level call returns the new root. No special case for the root is needed, just as the pointer-to-the-link technique of Section 7.7.3 needed none for the first node of a list.

One more subtlety: in case 3 the node z is not released; its *key* is overwritten and the successor's *node* is released. That is fine when a node holds nothing but a key. When nodes carry large records, or when other parts of a program hold pointers to particular nodes (the iterators of `std::map` promise to stay valid for all elements that are not erased), copying is not acceptable, and the successor node is instead *moved* into z's position by relinking pointers. Cormen et al. [2] describe that version, with a helper they call TRANSPLANT; its cost is the same.

11.7.11 Two variants: successor or predecessor

The successor was not the only possible replacement in case 3. The *predecessor* of z, the maximum of its left subtree, fits just as well: it is smaller than everything else in the left subtree and larger than everything in the right subtree. It has no right child, so deleting it is again case 1 or case 2. **Figure 11.20** deletes the root 50 of a ten-node tree both ways.

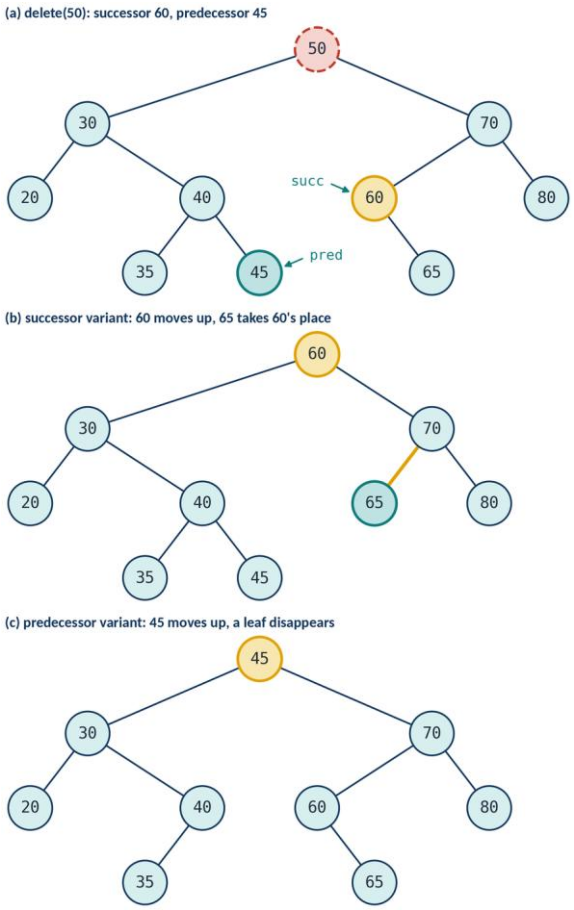


Figure 11.20 Deleting a node with two children, both ways: (a) the root 50 has successor 60 and predecessor 45; (b) with the successor, 60 moves up and its right child 65 takes its old place (case 2); (c) with the predecessor, 45 moves up and a leaf disappears (case 1).

- **Successor variant.** The minimum of 50's right subtree is found by going right to 70 and then left to 60. This time the successor is not a child of 50 but two levels down, and it has a right child, 65. Its key replaces 50, and deleting the old 60 is case 2: 65 moves up into 60's place as the left child of 70.
- **Predecessor variant.** The maximum of 50's left subtree is found by going left to 30 and right to 40 and 45. 45 replaces 50, and deleting the old 45, a leaf, is case 1.

Both results are correct binary search trees with the same keys; the inorder sequences agree. Which variant should a program use? For a single deletion it does not matter. Over many deletions it can: a program that always uses the successor keeps taking nodes from right subtrees, and experiments by Eppinger [9] showed that long random sequences of insertions and successor-only deletions make the tree measurably taller than a tree built by insertions alone. A simple and widely used remedy is to alternate between the two variants, or to choose at random.

11.7.12 The deletion program

Listing 11.7 implements successor and predecessor without parent pointers (**Pseudocode 11.9** and its mirror image) and `deleteNode` with a flag that selects the variant. Its `main` replays **Figure 11.19** on the running example, queries successors and predecessors on the ten-node tree of **Figure 11.18**, deletes that tree's root with each variant as in **Figure 11.20**, and finally tries to delete a key that is not there.

Listing 11.7 `bst_delete.cpp`: successor and predecessor without parent pointers, and deletion in all three cases with the successor and the predecessor variant.

```
// bst_delete.cpp -- successor, predecessor, and deletion from a binary search
// tree in all three cases, with the successor variant and the predecessor variant.
#include <cstdio>

struct Node {
    int data;
    Node *left, *right;
};

Node *insert(Node *root, int x) {           // returns the new subtree root
    if (root == nullptr) return new Node{x, nullptr, nullptr};
    if (x < root->data) root->left = insert(root->left, x);
    else if (x > root->data) root->right = insert(root->right, x);
    return root;                            // a duplicate changes nothing
}

Node *minNode(Node *t) {
    while (t != nullptr && t->left != nullptr) t = t->left;
    return t;
}

Node *maxNode(Node *t) {
    while (t != nullptr && t->right != nullptr) t = t->right;
    return t;
}

// The node holding the smallest key larger than x (x must be in the tree).
Node *successor(Node *root, int x) {
    Node *cand = nullptr;                   // last ancestor where we went left
    Node *cur = root;
    while (cur != nullptr && cur->data != x) {
        if (x < cur->data) {
            cand = cur;
            cur = cur->left;
        } else {
            cur = cur->right;
        }
    }
    if (cur == nullptr) return nullptr;     // x is not in the tree
    if (cur->right != nullptr) return minNode(cur->right);
    return cand;
}
```

Listing 11.7 (Continued) *bst_delete.cpp: successor and predecessor without parent pointers, and deletion in all three cases with the successor and the predecessor variant.*

```
// The node holding the largest key smaller than x (x must be in the tree).
Node *predecessor(Node *root, int x) {
    Node *cand = nullptr;                   // last ancestor where we went right
    Node *cur = root;
    while (cur != nullptr && cur->data != x) {
        if (x > cur->data) {
            cand = cur;
            cur = cur->right;
        } else {
            cur = cur->left;
        }
    }
    if (cur == nullptr) return nullptr;
    if (cur->left != nullptr) return maxNode(cur->left);
    return cand;
}

// Deletes x and returns the new root of this subtree. With useSucc the
// two-children case borrows the successor, otherwise the predecessor.
Node *deleteNode(Node *root, int x, bool useSucc) {
    if (root == nullptr) {
        std::printf("  %d is not in the tree\n", x);
        return nullptr;
    }
    if (x < root->data) {
        root->left = deleteNode(root->left, x, useSucc);
    } else if (x > root->data) {
        root->right = deleteNode(root->right, x, useSucc);
    } else if (root->left == nullptr && root->right == nullptr) {
        std::printf("  case 1: %d is a leaf, unlink it\n", x);
        delete root;
        return nullptr;
    } else if (root->left == nullptr || root->right == nullptr) {
        Node *child = (root->left != nullptr) ? root->left : root->right;
        std::printf("  case 2: %d has one child, %d moves up\n", x, child->data);
        delete root;
        return child;
    } else if (useSucc) {
        Node *s = minNode(root->right);     // smallest key of the right subtree
        std::printf("  case 3: %d has two children, successor %d replaces it\n",
                    x, s->data);
        root->data = s->data;
        root->right = deleteNode(root->right, s->data, useSucc);
    } else {
        Node *p = maxNode(root->left);      // largest key of the left subtree
        std::printf("  case 3: %d has two children, predecessor %d replaces it\n",
```

Listing 11.7 (Continued) *bst_delete.cpp: successor and predecessor without parent pointers, and deletion in all three cases with the successor and the predecessor variant.*

```
                    x, p->data);
        root->data = p->data;
        root->left = deleteNode(root->left, p->data, useSucc);
    }
    return root;
}

void printSideways(const Node *t, int depth) {
    if (t == nullptr) return;
    printSideways(t->right, depth + 1);
    std::printf("%*s%d\n", 4 * depth + 4, "", t->data);
    printSideways(t->left, depth + 1);
}

void inOrder(const Node *t) {
    if (t == nullptr) return;
    inOrder(t->left);
    std::printf("%d ", t->data);
    inOrder(t->right);
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

Node *build(const int *keys, int n) {
    Node *root = nullptr;
    for (int i = 0; i < n; ++i) root = insert(root, keys[i]);
    return root;
}

void show(const char *title, const Node *t) {
    std::printf("%s  inorder: ", title);
    inOrder(t);
    std::printf("\n");
    printSideways(t, 0);
}
```

Listing 11.7 (Continued) *bst_delete.cpp: successor and predecessor without parent pointers, and deletion in all three cases with the successor and the predecessor variant.*

```
int main() {
    int a[] = {65, 5, 70, 3, 10};
    Node *t = build(a, 5);
    show("Tree A:", t);
    int dels[] = {10, 5, 65};
    for (int x : dels) {
        std::printf("delete %d\n", x);
        t = deleteNode(t, x, true);
        show("after:", t);
    }
    destroy(t);

    int b[] = {50, 30, 70, 20, 40, 60, 80, 35, 45, 65};
    t = build(b, 10);
    show("\nTree B:", t);
    int qs[] = {45, 50, 35, 65, 80, 20};
    for (int x : qs) {
        Node *s = successor(t, x), *p = predecessor(t, x);
        std::printf("  key %2d: predecessor ", x);
        if (p != nullptr) std::printf("%2d", p->data); else std::printf("--");
        std::printf(", successor ");
        if (s != nullptr) std::printf("%2d\n", s->data); else std::printf("--\n");
    }
    std::printf("delete 50 (successor variant)\n");
    t = deleteNode(t, 50, true);
    show("after:", t);
    destroy(t);

    t = build(b, 10);
    std::printf("\nTree B again; delete 50 (predecessor variant)\n");
    t = deleteNode(t, 50, false);
    show("after:", t);
    std::printf("delete 99\n");
    t = deleteNode(t, 99, false);
    destroy(t);
    return 0;
}
```

```
Tree A:  inorder: 3 5 10 65 70
        70
    65
            10
        5
            3
delete 10
  case 1: 10 is a leaf, unlink it
after:  inorder: 3 5 65 70
        70
    65
        5
            3
delete 5
  case 2: 5 has one child, 3 moves up
after:  inorder: 3 65 70
        70
    65
        3
delete 65
  case 3: 65 has two children, successor 70 replaces it
  case 1: 70 is a leaf, unlink it
after:  inorder: 3 70
    70
        3

Tree B:  inorder: 20 30 35 40 45 50 60 65 70 80
            80
        70
                65
            60
    50
                45
            40
                35
        30
            20
  key 45: predecessor 40, successor 50
  key 50: predecessor 45, successor 60
  key 35: predecessor 30, successor 40
  key 65: predecessor 60, successor 70
  key 80: predecessor 70, successor --
  key 20: predecessor --, successor 30
delete 50 (successor variant)
  case 3: 50 has two children, successor 60 replaces it
  case 2: 60 has one child, 65 moves up
after:  inorder: 20 30 35 40 45 60 65 70 80
            80
        70
            65
    60
                45
```

```
            40
                35
        30
            20

Tree B again; delete 50 (predecessor variant)
  case 3: 50 has two children, predecessor 45 replaces it
  case 1: 45 is a leaf, unlink it
after:  inorder: 20 30 35 40 45 60 65 70 80
            80
        70
                65
            60
    45
            40
                35
        30
            20
delete 99
  99 is not in the tree
```

The trace lines name the case that applied. In the first tree, deleting 10, then 5, then 65 triggers cases 1, 2 and 3 in that order, and case 3 finishes with a case 1 deletion of the old 70, exactly as in **Figure 11.19**. In the second tree the six queries cover every situation: 45 and 35 have no right subtree (case 2 of **Pseudocode 11.9**), 50 and 65 have one, 80 has no successor and 20 no predecessor. The two deletions of 50 produce the trees of **Figure 11.20** (b) and (c), and the last line shows that deleting an absent key falls off the tree and changes nothing. Under Valgrind the program ends with all heap blocks freed and no errors; the Java and Python versions (`BstDelete.java`, `bst_delete.py`) print the same output.

11.7.13 Costs

Every BST operation except the traversals follows one path down from the root, possibly followed by one more path down (the minimum of a subtree in deletion case 3), and does constant work per node on the path. So:

$$T_{\text{search}},\ T_{\text{insert}},\ T_{\text{delete}},\ T_{\min},\ T_{\max},\ T_{\text{succ}},\ T_{\text{pred}} \ = \ O(h) \tag{11.10}$$

where h is the height of the tree. **Table 11.3** combines Equation (11.10) with the two extreme heights of Section 11.5.3.

Table 11.3 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 11.10**)	$O(\lg n)$	$O(n)$
minimum, maximum	$O(h)$ (**Pseudocode 11.7**)	$O(\lg n)$	$O(n)$
successor, predecessor	$O(h)$ (**Pseudocode 11.9**)	$O(\lg n)$	$O(n)$
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	$O(n \lg n)$	$\Theta(n^2)$

The table puts the whole question in one column: a BST is as good as binary search when its height is logarithmic, and no better than the linked list of Chapter 7 when its height is linear. The height is decided by the order in which the keys arrived. The next section measures it.

11.8 Balanced and Degenerate Trees

11.8.1 The order of insertion decides the shape

Insert the seven keys 1, 2, …, 7 into an empty BST in ascending order. Each new key is larger than every key already in the tree, so every insertion walks right all the way down and hangs the new node as the right child of the previous one. The result, **Figure 11.21** (a), is a right-skewed chain of height 6: a linked list, drawn diagonally, and every search in it is the sequential search of Chapter 2. Insert the same seven keys in the order 4, 2, 6, 1, 3, 5, 7 and you get the perfect tree of **Figure 11.21** (b), of height 2, the lowest possible by Equation (11.6).

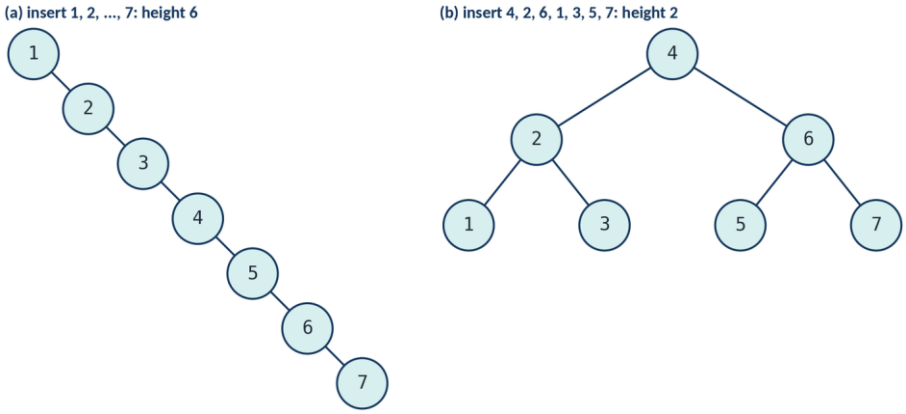


Figure 11.21 The same seven keys in two insertion orders: (a) in sorted order they form a chain of height 6; (b) inserted middle first they form a perfect tree of height 2.

Sorted input is the worst case, and in practice it is the *common* case: records arrive in order of their creation time, codes are handed out in increasing order, a file is often already sorted. How many of the $n!$ insertion orders produce a chain? The first key must be the smallest or the largest remaining key (otherwise it would have keys on both sides and the tree would branch at the root), and the same holds for every later key among those still to come, until one key is left. That is two choices for each of the first $n-1$ keys:

$$\#\{\text{orders giving height } n-1\} \;=\; 2^{n-1} \tag{11.11}$$

For $n=7$ that is 64 of the $7!=5040$ orders, about 1.3%. For random orders, degenerate trees are rare. The question is what a *typical* random tree looks like, and the answer is: good.

11.8.2 Measured heights

The theory of random binary search trees says that if all $n!$ insertion orders are equally likely, the average depth of a node is

$$\bar{d}(n) \;=\; 2\left(1+\frac{1}{n}\right)H_n - 4 \;\approx\; 2\ln n \;\approx\; 1.39\lg n \tag{11.12}$$

where $H_n = 1+\frac{1}{2}+\cdots+\frac{1}{n} \approx \ln n + 0.577$ is the n-th harmonic number [8]. So a random BST is only about 39% deeper, on average, than the best possible tree. Its *height*, the depth of its deepest node, is larger, about $4.311\ln n$ for large n [10], which is still logarithmic.

Listing 11.8 checks both claims and measures the worst case. Part 1 inserts the keys $0,1,\ldots,n-1$ in a random order (a fixed seed makes the run repeatable) for n from 2^{10} to 2^{20} and reports the height, the average depth, the value of Equation (11.12) and the time. Part 2 inserts the same keys

in sorted order and in random order and compares. Insertion is iterative, with the pointer-to-the-link technique of Section 7.7.3, and the height is measured level by level, so that a chain of 32,000 nodes cannot overflow the call stack.

Listing 11.8 `height_experiment.cpp`: the height and average depth of binary search trees built from random and from sorted insertion orders, with the time to build them.

```
// height_experiment.cpp -- how tall does a binary search tree grow? Keys inserted
// in random order against keys inserted in sorted order, measured.
#include <algorithm>
#include <chrono>
#include <cmath>
#include <cstdio>
#include <numeric>
#include <random>
#include <vector>

struct Node {
    int key;
    Node *left, *right;
};

void insert(Node *&root, int key) {        // iterative: no recursion depth limit
    Node **link = &root;
    while (*link != nullptr)
        link = (key < (*link)->key) ? &(*link)->left : &(*link)->right;
    *link = new Node{key, nullptr, nullptr};
}

// Height and total depth by level-order, with no recursion.
void measure(Node *root, int &height, double &totalDepth) {
    std::vector<Node *> level{root}, next;
    height = -1;
    totalDepth = 0;
    while (!level.empty()) {
        ++height;
        totalDepth += (double) height * level.size();
        next.clear();
        for (Node *n : level) {
            if (n->left) next.push_back(n->left);
            if (n->right) next.push_back(n->right);
        }
        level.swap(next);
    }
}
```

Listing 11.8 (Continued) *height_experiment.cpp: the height and average depth of binary search trees built from random and from sorted insertion orders, with the time to build them.*

```
void destroy(Node *root) {                  // iterative, with an explicit stack
    std::vector<Node *> st;
    if (root) st.push_back(root);
    while (!st.empty()) {
        Node *n = st.back();
        st.pop_back();
        if (n->left) st.push_back(n->left);
        if (n->right) st.push_back(n->right);
        delete n;
    }
}

double buildMs(const std::vector<int> &keys, int &h, double &avg) {
    auto t0 = std::chrono::steady_clock::now();
    Node *root = nullptr;
    for (int k : keys) insert(root, k);
    auto t1 = std::chrono::steady_clock::now();
    double total;
    measure(root, h, total);
    avg = total / keys.size();
    destroy(root);
    return std::chrono::duration<double, std::milli>(t1 - t0).count();
}

int main() {
    std::mt19937 rng(2026);
    std::printf("Part 1: keys 0..n-1 inserted in random order\n");
    std::printf("%9s %7s %8s %10s %10s %9s\n",
                "n", "floor", "height", "avg depth", "expected", "build ms");
    std::printf("%9s %7s %8s %10s %10s\n", "", "lg n", "", "", "avg depth");
    for (int n = 1024; n <= (1 << 20); n *= 4) {
        std::vector<int> keys(n);
        std::iota(keys.begin(), keys.end(), 0);
        std::shuffle(keys.begin(), keys.end(), rng);
        int h;
        double avg;
        double ms = buildMs(keys, h, avg);
        double H = 0;                       // harmonic number H_n
        for (int i = 1; i <= n; ++i) H += 1.0 / i;
        double expected = 2.0 * (1.0 + 1.0 / n) * H - 4.0;
        std::printf("%9d %7d %8d %10.2f %10.2f %9.1f\n",
                    n, (int) std::floor(std::log2(n)), h, avg, expected, ms);
    }
```

Listing 11.8 (Continued) *height_experiment.cpp: the height and average depth of binary search trees built from random and from sorted insertion orders, with the time to build them.*

```
    std::printf("\nPart 2: the same keys in sorted order against random order\n");
    std::printf("%9s %14s %12s %14s %12s\n",
                "n", "sorted height", "sorted ms", "random height", "random ms");
    for (int n = 2000; n <= 32000; n *= 2) {
        std::vector<int> keys(n);
        std::iota(keys.begin(), keys.end(), 0);
        int hs, hr;
        double avg;
        double ms = buildMs(keys, hs, avg);
        std::shuffle(keys.begin(), keys.end(), rng);
        double mr = buildMs(keys, hr, avg);
        std::printf("%9d %14d %12.1f %14d %12.2f\n", n, hs, ms, hr, mr);
    }
    return 0;
}
```

```
Part 1: keys 0..n-1 inserted in random order
        n   floor   height  avg depth   expected  build ms
             lg n                      avg depth
     1024      10       19      10.70      11.03       0.1
     4096      12       27      14.86      13.79       0.5
    16384      14       31      15.42      16.56       2.1
    65536      16       36      19.42      19.34      17.0
   262144      18       43      22.66      22.11      83.2
  1048576      20       51      24.83      24.88     530.1

Part 2: the same keys in sorted order against random order
        n  sorted height    sorted ms  random height    random ms
     2000           1999         3.0             23         0.11
     4000           3999        11.0             27         0.30
     8000           7999        52.4             31         0.71
    16000          15999       203.6             32         1.59
    32000          31999       806.0             35         3.44
```

Figure 11.22 plots the run. The times depend on the machine (this run used a shared 2.1 GHz Intel Xeon server); the heights do not, because the seed fixes the insertion order.

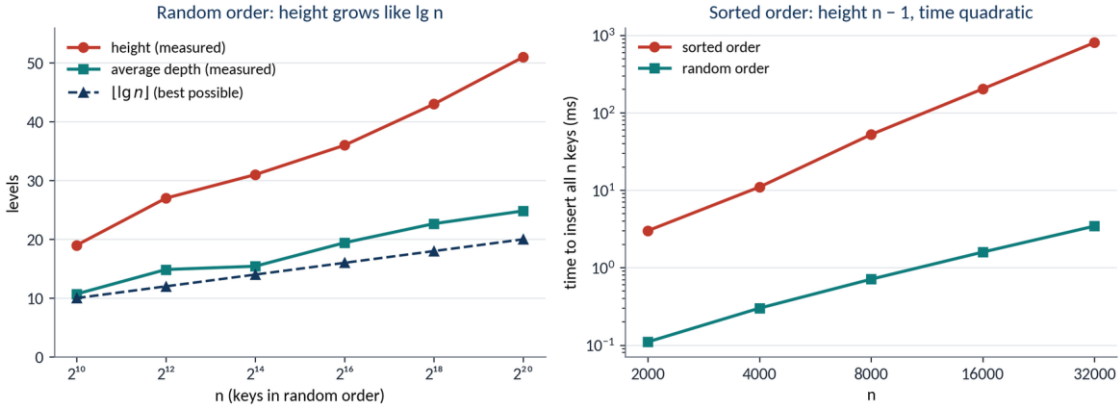


Figure 11.22 The measurements of **Listing 11.8**: in random order the height and the average depth grow in step with lg$n$ (left); in sorted order the tree is a chain, and the time to build it grows four-fold with every doubling of n (right, logarithmic scale).

- **Random order is logarithmic.** Each time n is multiplied by 4, lg$n$ grows by 2. Over the five steps from 2^{10} to 2^{20}, lg$n$ grows by 10 and the measured average depth by $24.83 - 10.70 = 14.13$, the ratio of about 1.39 that Equation (11.12) predicts. Individual steps vary, because the formula is an average over all insertion orders and each run is one order, but every measured value is within about one level of the formula. The height grows faster than lg$n$ but stays small: a million keys give a tree of height 51, compared with the minimum possible 19 and the chain's 1,048,575. Any search in it takes at most 52 comparisons.
- **Sorted order is linear.** The height is exactly $n - 1$ each time, as predicted. Building the tree costs $0 + 1 + 2 + \cdots + (n-1) = n(n-1)/2$ steps, $\Theta(n^2)$, and the times show it: each doubling of n multiplies the time by about 4 (3.0, 11.0, 52.4, 203.6, 806 ms). At $n = 32{,}000$ the sorted build is about 230 times slower than the random one, which takes 3.44 ms, and the gap grows without bound.

The Java and Python versions in the code folder (`HeightExperiment.java`, `height_experiment.py`) show the same shapes with their own timings. In one Python run the sorted build of 32,000 keys took about 16 seconds against 50 ms for the random build; in Java, 1.4 seconds against 4 ms.

11.8.3 What "balanced" means

A family of trees is **balanced** if its height is $O(\lg n)$. Several precise conditions guarantee it; the best known is the one AVL trees maintain:

> **DEFINITION.** A binary tree is *height-balanced* if, at every node, the heights of the left and right subtrees differ by at most 1. The difference $height(x.left) - height(x.right)$ is the *balance factor* of x; in a height-balanced tree it is -1, 0 or $+1$ everywhere.

Every perfect and every complete tree is height-balanced; the chain of **Figure 11.21** (a) is not (at node 1 the left subtree has height -1 and the right height 5). Chapter 12 writes a function that checks this condition by computing heights with Equation (11.9). Section 11.9.2 shows why the condition forces logarithmic height.

There are three ways to live with the fact that a plain BST's shape depends on its input.

1. **Hope the input is random.** Often reasonable, and **Listing 11.8** shows how good random is. Shuffling a batch of keys before inserting them makes it true. But a program cannot shuffle keys that arrive one at a time, and an adversary can send sorted keys on purpose.
2. **Rebuild when the tree is known.** If all keys are available at once, sort them and build the tree from the middle outward: the middle key becomes the root, and the two halves are built the same way, recursively. The result has the minimum height $\lfloor \lg n \rfloor$ and the construction takes $O(n)$ after sorting; Exercise 11.10 writes it.
3. **Rebalance on every change.** Make every insertion and deletion repair the shape locally so that the height stays $O(\lg n)$ whatever the input. That is what self-balancing trees do, and what every standard library's ordered map does.

11.9 A First Look at Self-Balancing Trees

This section is a preview: it shows the one operation all self-balancing trees are built on, runs one of them on the input that ruins a plain BST, and names the two families that the standard libraries use. The full algorithms belong to a more advanced study of algorithms; Cormen et al. [2] give them in detail.

11.9.1 Rotations

A *rotation* changes the shape of a small part of a BST in $O(1)$ time without breaking the BST property. **Figure 11.23** shows the two directions. In the left picture x is the left child of y; α and β are the subtrees of x and γ is the right subtree of y. A *right rotation at y* makes x the root of this part, with y as its right child; β, which was between x and y in key order, moves across to become y's left subtree. A *left rotation at x* undoes it.

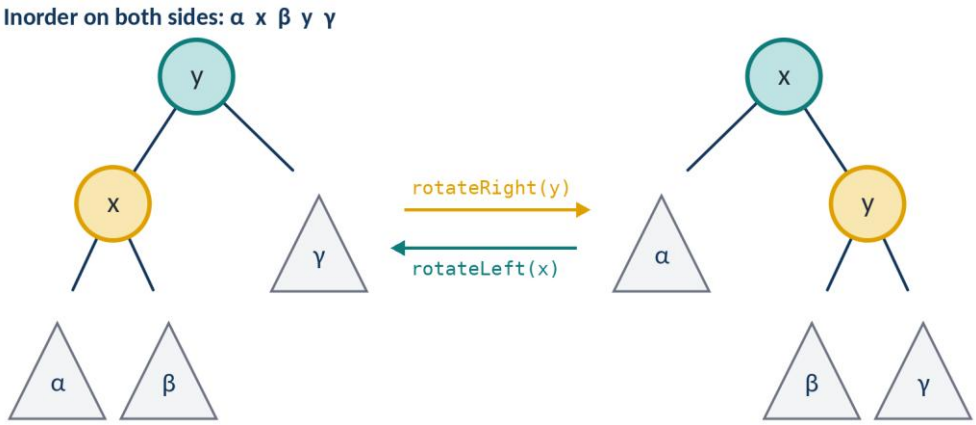


Figure 11.23 Right and left rotation: three pointers change, the subtree rooted at the top becomes one level lower on one side and one level higher on the other, and the inorder sequence $\alpha, x, \beta, y, \gamma$ is unchanged.

A rotation preserves the BST property because it preserves the inorder sequence: on both sides it is $\alpha, x, \beta, y, \gamma$. And it changes only three pointers (`y->left`, `x->right`, and the pointer from above that pointed to y and now points to x), so it costs $O(1)$. What it changes is the height: in a right rotation, everything in α moves up one level and everything in γ moves down one level. A self-balancing tree detects, after an insertion or a deletion, a place where one side has become too deep, and rotates there.

11.9.2 AVL trees

The first self-balancing BST was published in 1962 by Georgy Adelson-Velsky and Evgenii Landis [5], and bears their initials. An **AVL tree** keeps every node height-balanced (Section 11.8.3). Each node stores its height; after an insertion walks down and adds a leaf, the recursion walks back up, recomputes the heights, and wherever a balance factor has become $+2$ or -2 it repairs the node with one rotation or, when the deep part is on the inner side, with two (first at the child, then at the node). **Figure 11.24** inserts 1, 2, ..., 7 in sorted order, the input that made **Figure 11.21** (a).

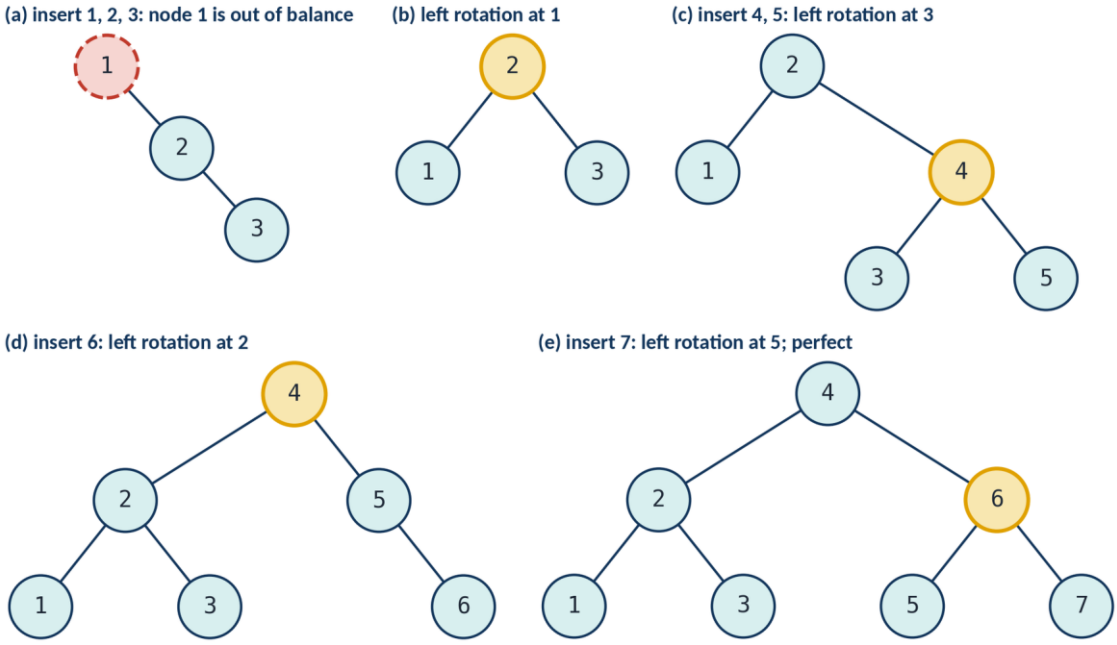


Figure 11.24 Inserting 1 to 7 in sorted order into an AVL tree: (a) after 3, node 1 has balance factor -2; (b) a left rotation at 1 repairs it; (c) to (e) three more left rotations keep the tree balanced, and the result is perfect.

Why does height-balance imply logarithmic height? Let $N(h)$ be the *fewest* nodes an AVL tree of height h can have. Such a tree has a root and two subtrees, one of height $h-1$ and, to have as few nodes as possible, the other of height $h-2$, both themselves as sparse as possible: $N(h) = N(h-1) + N(h-2) + 1$, with $N(0) = 1$ and $N(1) = 2$. This is the Fibonacci recurrence of Section 10.5, plus one, and indeed $N(h) = F_{h+3} - 1$. Fibonacci numbers grow like φ^h with $\varphi \approx 1.618$, so $n \geq N(h)$ forces h to be at most about $\log_\varphi n$. The exact bound [8] is

$$h \;<\; 1.4405\lg(n+2) - 0.3277 \tag{11.13}$$

An AVL tree is never more than 44% taller than the best possible tree, whatever order the keys come in. **Listing 11.9** implements AVL insertion (deletion is similar but longer) and feeds it sorted keys.

Listing 11.9 `avl_first_look.cpp`: AVL insertion with single and double rotations, fed the sorted keys that turn a plain binary search tree into a chain.

```
// avl_first_look.cpp -- a first look at a self-balancing tree: AVL insertion
// with rotations, fed the sorted keys that ruin a plain binary search tree.
#include <cmath>
#include <cstdio>

struct Node {
    int key;
    int h;                  // height of the subtree rooted here
    Node *left, *right;
};

bool verbose = true;
int ht(const Node *t) { return t == nullptr ? -1 : t->h; }
void fix(Node *t) {            // recompute the height from the children
    int a = ht(t->left), b = ht(t->right);
    t->h = 1 + (a > b ? a : b);
}
int balance(const Node *t) { return ht(t->left) - ht(t->right); }

Node *rotateRight(Node *y) {    // y's left child x becomes the subtree root
    Node *x = y->left;
    y->left = x->right;
    x->right = y;
    fix(y);
    fix(x);
    return x;
}

Node *rotateLeft(Node *x) {     // x's right child y becomes the subtree root
    Node *y = x->right;
    x->right = y->left;
    y->left = x;
    fix(x);
    fix(y);
    return y;
}
```

Listing 11.9 (Continued) *avl_first_look.cpp: AVL insertion with single and double rotations, fed the sorted keys that turn a plain binary search tree into a chain.*

```
Node *insert(Node *t, int key) {
    if (t == nullptr) return new Node{key, 0, nullptr, nullptr};
    if (key < t->key) t->left = insert(t->left, key);
    else if (key > t->key) t->right = insert(t->right, key);
    else return t;
    fix(t);
    int b = balance(t);
    if (b > 1 && key > t->left->key) t->left = rotateLeft(t->left);    // left-right
    if (b < -1 && key < t->right->key) t->right = rotateRight(t->right);
    if (b > 1) {
        if (verbose)
            std::printf("  insert %d: right rotation at %d\n", key, t->key);
        return rotateRight(t);
    }
    if (b < -1) {
        if (verbose) std::printf("  insert %d: left rotation at %d\n", key, t->key);
        return rotateLeft(t);
    }
    return t;
}

void printSideways(const Node *t, int depth) {
    if (t == nullptr) return;
    printSideways(t->right, depth + 1);
    std::printf("%*s%d\n", 4 * depth + 4, "", t->key);
    printSideways(t->left, depth + 1);
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

int main() {
    Node *root = nullptr;
    std::printf("AVL tree, keys 1..7 inserted in sorted order:\n");
    for (int k = 1; k <= 7; ++k) root = insert(root, k);
    printSideways(root, 0);
    std::printf("height %d (a plain BST would have height 6)\n\n", ht(root));
    destroy(root);
```

Listing 11.9 (Continued) *avl_first_look.cpp: AVL insertion with single and double rotations, fed the sorted keys that turn a plain binary search tree into a chain.*

```
    verbose = false;
    std::printf("%9s %12s %12s %14s\n", "n", "AVL height", "plain BST",
                "1.44 lg(n+2)");
    for (int n = 1000; n <= 1000000; n *= 10) {
        root = nullptr;
        for (int k = 1; k <= n; ++k) root = insert(root, k);
        std::printf("%9d %12d %12d %14.1f\n", n, ht(root), n - 1,
                    1.44 * std::log2(n + 2.0));
        destroy(root);
    }
    return 0;
}
```

```
AVL tree, keys 1..7 inserted in sorted order:
  insert 3: left rotation at 1
  insert 5: left rotation at 3
  insert 6: left rotation at 2
  insert 7: left rotation at 5
            7
        6
            5
    4
            3
        2
            1
height 2 (a plain BST would have height 6)

        n   AVL height    plain BST   1.44 lg(n+2)
     1000            9           999           14.4
    10000           13          9999           19.1
   100000           16         99999           23.9
  1000000           19        999999           28.7
```

The rotations printed for 1 to 7 are those of **Figure 11.24**, and the resulting tree is perfect. The table compares, for sorted input, the AVL tree's height with the plain BST's height $n-1$ (measured in **Listing 11.8**) and with the bound of Equation (11.13). With a million sorted keys the AVL tree has height 19, which equals $\lfloor \lg n \rfloor$: sorted input, the worst case for a plain BST, happens to be an easy case for an AVL tree. (The Java and Python versions, `AvlFirstLook.java` and `avl_first_look.py`, print the same.)

11.9.3 Red-black trees

The other great family is the **red-black tree**. Rudolf Bayer introduced the idea in 1972 as the "symmetric binary B-tree" [6], and Leonidas Guibas and Robert Sedgewick gave it its colors and its name in 1978 [7]. Each node carries one extra bit, its color, and the tree obeys five rules [2]:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (the NIL pointers) is black.
4. A red node has two black children: no two reds in a row on any path.
5. For every node, all paths from it down to NIL leaves contain the same number of black nodes, called its *black-height*.

Figure 11.25 colors the ten-node tree of **Figure 11.18** as a valid red-black tree: every path from the root to a NIL passes exactly two black nodes, and no red node has a red child.

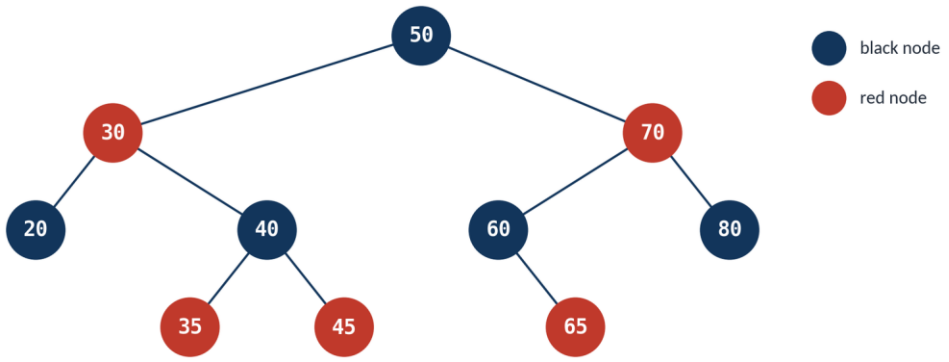


Figure 11.25 The tree of **Figure 11.18** colored as a red-black tree: the root is black, red nodes have black children, and every root-to-NIL path contains two black nodes.

Rules 4 and 5 bound the height. By rule 5 there are the same number of black nodes on every path, and by rule 4 at most as many red nodes as black ones on any path, so the longest path is at most twice the shortest. A counting argument then gives [2]

$$h \leq 2\lg(n+1) \tag{11.14}$$

which is a weaker guarantee than Equation (11.13): a red-black tree can be somewhat taller than an AVL tree. In exchange, it needs fewer rotations to repair itself: at most two rotations per insertion and at most three per deletion, plus some recoloring, whereas an AVL deletion may rotate at every level on the way up. That makes red-black trees the usual choice for general-purpose libraries, where insertions and deletions are as frequent as searches: `std::map` and `std::set` in all three major C++ standard libraries [11], Java's `TreeMap` and `TreeSet` [12], and the Linux kernel's own balanced-tree library [13] are all red-black trees. AVL trees, being lower, are preferred when searches dominate. **Table 11.4** compares the three kinds of tree met in this chapter.

Table 11.4 Plain, AVL and red-black binary search trees compared.

	Plain BST	AVL tree	Red-black tree
Extra data per node	none	height (or balance factor)	one color bit
Height, worst case	$n-1$	$<1.4405\lg(n+2)$ (Equation (11.13))	$\leq 2\lg(n+1)$ (Equation (11.14))
Search, insert, delete	$O(n)$ worst, $O(\lg n)$ average	$O(\lg n)$ worst	$O(\lg n)$ worst
Rotations per insertion	0	at most 2	at most 2
Rotations per deletion	0	up to $O(\lg n)$	at most 3
Typical use	teaching, random keys	search-heavy workloads	`std::map`, `TreeMap`, kernels

11.9.4 Other balanced trees

Three more members of the family are worth knowing by name. **B-trees**, introduced by Bayer and McCreight in 1972 [14], give each node hundreds of keys and children, so that a tree of a billion keys is only four or five levels deep; since each level costs one disk or SSD read, they are the index structure of almost every database, including the PostgreSQL indexes of Section 11.12 [15]. **Splay trees** (Sleator and Tarjan [16]) do not keep any balance information at all, but move every accessed node to the root with rotations; the cost of a long sequence of operations is $O(\lg n)$ each on average. **Treaps** (Seidel and Aragon [17]) give every node a random priority and keep the tree a heap on the priorities, which makes its shape that of a random BST (Section 11.8.2) whatever the insertion order. Chapter 14's heaps are the other half of that idea.

11.10 General Trees: Conversion and Reconstruction

11.10.1 First child, next sibling

Section 11.4.3 promised that any tree can be stored with two pointers per node. Give each node a pointer to its **first child** and a pointer to its **next sibling** (the next child of the same parent). Now read those two pointers as the `left` and `right` pointers of a binary tree: the first child becomes the left child, and the next sibling becomes the right child. Equivalently: the first child of a node

becomes its left child, the second child becomes the right child of that left child (the node's "right grandchild"), the third child the right child of the second, and so on down a chain of right pointers. **Figure 11.26** carries out the conversion on a thirteen-node general tree.

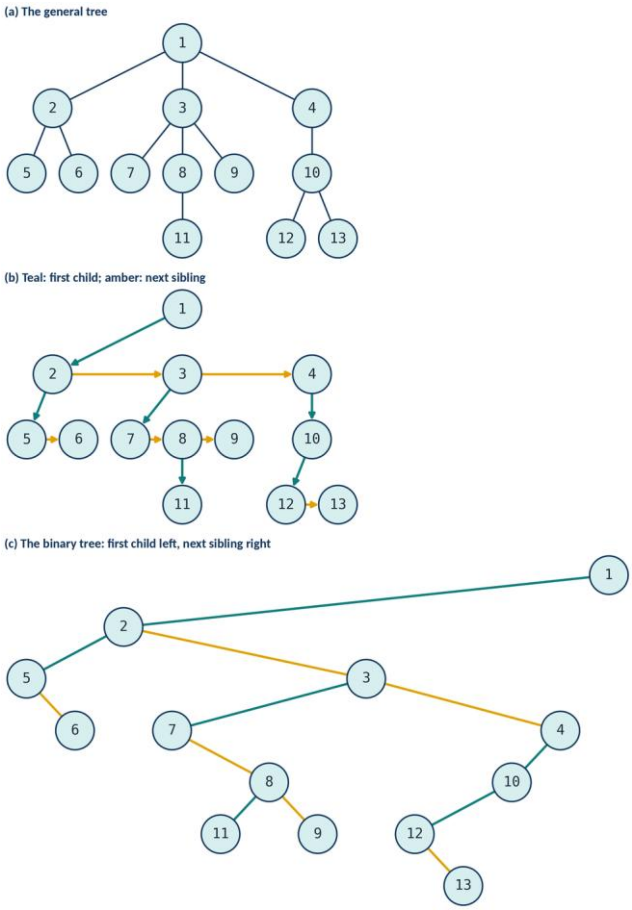


Figure 11.26 Converting a general tree to a binary tree: (a) the general tree; (b) each node keeps only a link to its first child (teal) and to its next sibling (amber); (c) the same links drawn as a binary tree, first child on the left and next sibling on the right.

The conversion loses nothing: the general tree can be read back by following, from each node, its left pointer and then the chain of right pointers from there, which lists the node's children in order. And the traversals correspond exactly:

- the **preorder** of the general tree (a node, then each child's subtree in order) equals the **preorder** of the binary tree;
- the **postorder** of the general tree (each child's subtree in order, then the node) equals the **inorder** of the binary tree.

To see the second fact, note that in the binary tree a node's left subtree holds exactly its general-tree descendants, and its right subtree holds its later siblings and their descendants. An inorder traversal therefore lists all of a node's descendants, then the node, then its later siblings: which is precisely postorder in the general tree. The postorder of the binary tree corresponds to no natural order of the general tree, which is why the program below shows it only for comparison.

11.10.2 Rebuilding a tree from preorder and degrees

A preorder listing alone does not determine a tree: the listing A, B, C could be a root with two children or a chain of three. But the preorder listing together with the *degree* of every node (its number of children) does. The rule is recursive, like everything else in this chapter: read the next

node and its degree d from the listing; it is the root of the next subtree; then build its d subtrees, one after another, from the rest of the listing.

A second way to rebuild the tree works from the end, and it is convenient by hand. Scan the listing from right to left, looking for a node whose degree is greater than 0. When you find one with degree d, the d subtrees immediately to its right (each already assembled, or a single leaf) are its children: attach them and replace them in the listing by the combined tree. Continue leftward until the root has taken its children. **Figure 11.27** applies it to two listings.

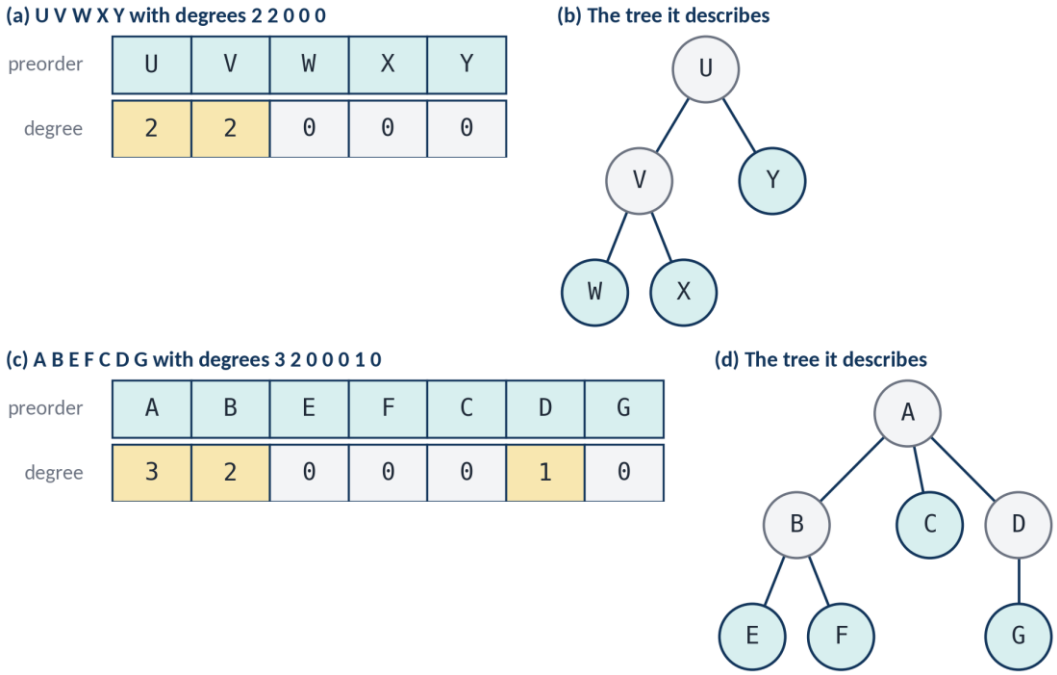


Figure 11.27 Two trees rebuilt from their preorder listing and the degree of each node: U V W X Y with degrees 2 2 0 0 0 (a, b), and A B E F C D G with degrees 3 2 0 0 0 1 0 (c, d).

- For **U V W X Y** with degrees **2 2 0 0 0**, the first nonzero degree from the right is V's, 2: its children are the two nodes to its right, W and X. Remove them; the listing is now U V(W,X) Y. The next nonzero degree is U's, 2: its children are V (with its subtree) and Y. The tree is U(V(W,X),Y).
- For **A B E F C D G** with degrees **3 2 0 0 0 1 0**: D has degree 1, so G is its child; B has degree 2, so E and F are its children; A has degree 3, so its children are B, C and D. The tree is A(B(E,F),C,D(G)). As a check, the degrees add up to $3 + 2 + 1 = 6 = 7 - 1$, the number of edges, as Equation (11.1) requires: every edge is counted once, as some node's child.

Preorder with degrees is a compact way to *save* a tree in a file or send it over a network: one line of labels, one line of numbers, and the receiver rebuilds the exact shape. Exercise 11.6 shows a second pair of listings that determines a binary tree: preorder together with inorder.

11.10.3 The program

Listing 11.10 stores the tree of **Figure 11.26** with first-child and next-sibling pointers, prints it in parenthesis notation, traverses it in preorder and postorder, converts it to a binary tree and traverses that three ways, and then rebuilds the two trees of **Figure 11.27** from their preorder listings and degrees.

Listing 11.10 `general_tree.cpp`: a general tree in first-child/next-sibling form, its conversion to a binary tree, and trees rebuilt from preorder listings and degrees.

```cpp
// general_tree.cpp -- a general tree in first-child/next-sibling form, its
// conversion to a binary tree, and a tree rebuilt from preorder plus degrees.
#include <cstdio>
#include <string>

struct GTN {                    // General Tree Node
    std::string data;
    GTN *firstChild = nullptr;
    GTN *nextSibling = nullptr;
    explicit GTN(const std::string &d) : data(d) {}
};

struct BTN {                    // Binary Tree Node
    std::string data;
    BTN *left = nullptr, *right = nullptr;
    explicit BTN(const std::string &d) : data(d) {}
};

GTN *addChild(GTN *parent, const std::string &d) {   // appends a last child
    GTN *c = new GTN(d);
    if (parent->firstChild == nullptr) {
        parent->firstChild = c;
    } else {
        GTN *s = parent->firstChild;
        while (s->nextSibling != nullptr) s = s->nextSibling;
        s->nextSibling = c;
    }
    return c;
}

void gtPreOrder(const GTN *t) {         // the node, then each child's subtree
    std::printf("%s ", t->data.c_str());
    for (GTN *c = t->firstChild; c != nullptr; c = c->nextSibling) gtPreOrder(c);
}

void gtPostOrder(const GTN *t) {        // each child's subtree, then the node
    for (GTN *c = t->firstChild; c != nullptr; c = c->nextSibling) gtPostOrder(c);
    std::printf("%s ", t->data.c_str());
}
```

Listing 11.10 (Continued) *general_tree.cpp: a general tree in first-child/next-sibling form, its conversion to a binary tree, and trees rebuilt from preorder listings and degrees.*

```
void parenthesis(const GTN *t) {        // e.g. A(B(D,E),C)
    std::printf("%s", t->data.c_str());
    if (t->firstChild == nullptr) return;
    std::printf("(");
    for (GTN *c = t->firstChild; c != nullptr; c = c->nextSibling) {
        parenthesis(c);
        if (c->nextSibling != nullptr) std::printf(",");
    }
    std::printf(")");
}

// First child -> left, next sibling -> right.
BTN *convertToBinaryTree(const GTN *node) {
    if (node == nullptr) return nullptr;
    BTN *b = new BTN(node->data);
    b->left = convertToBinaryTree(node->firstChild);
    b->right = convertToBinaryTree(node->nextSibling);
    return b;
}

void btPreOrder(const BTN *t) {
    if (t == nullptr) return;
    std::printf("%s ", t->data.c_str());
    btPreOrder(t->left);
    btPreOrder(t->right);
}

void btInOrder(const BTN *t) {
    if (t == nullptr) return;
    btInOrder(t->left);
    std::printf("%s ", t->data.c_str());
    btInOrder(t->right);
}

void btPostOrder(const BTN *t) {
    if (t == nullptr) return;
    btPostOrder(t->left);
    btPostOrder(t->right);
    std::printf("%s ", t->data.c_str());
}
```

Listing 11.10 (Continued) *general_tree.cpp: a general tree in first-child/next-sibling form, its conversion to a binary tree, and trees rebuilt from preorder listings and degrees.*

```
// Rebuilds a general tree from its preorder listing and every node's degree.
GTN *buildFromPreorder(const std::string *pre, const int *deg, int &index) {
    GTN *node = new GTN(pre[index]);
    int children = deg[index];
    ++index;                              // the next unread position
    for (int i = 0; i < children; ++i) {
        GTN *c = buildFromPreorder(pre, deg, index);
        if (node->firstChild == nullptr) {
            node->firstChild = c;
        } else {
            GTN *s = node->firstChild;
            while (s->nextSibling != nullptr) s = s->nextSibling;
            s->nextSibling = c;
        }
    }
    return node;
}

void destroy(GTN *t) {
    if (t == nullptr) return;
    destroy(t->firstChild);
    destroy(t->nextSibling);
    delete t;
}

void destroy(BTN *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

int main() {
    GTN *root = new GTN("1");
    GTN *n2 = addChild(root, "2");
    GTN *n3 = addChild(root, "3");
    GTN *n4 = addChild(root, "4");
    addChild(n2, "5");
    addChild(n2, "6");
    addChild(n3, "7");
    addChild(addChild(n3, "8"), "11");
    addChild(n3, "9");
    GTN *n10 = addChild(n4, "10");
    addChild(n10, "12");
    addChild(n10, "13");
```

Listing 11.10 (Continued) *general_tree.cpp: a general tree in first-child/next-sibling form, its conversion to a binary tree, and trees rebuilt from preorder listings and degrees.*

```
    std::printf("General tree: ");
    parenthesis(root);
    std::printf("\n  PreOrder:  ");
    gtPreOrder(root);
    std::printf("\n  PostOrder: ");
    gtPostOrder(root);

    BTN *b = convertToBinaryTree(root);
    std::printf("\nConverted binary tree:\n  PreOrder:  ");
    btPreOrder(b);
    std::printf("\n  InOrder:   ");
    btInOrder(b);
    std::printf("\n  PostOrder: ");
    btPostOrder(b);
    std::printf("\n");
    destroy(b);
    destroy(root);

    std::string pre1[] = {"U", "V", "W", "X", "Y"};
    int deg1[] = {2, 2, 0, 0, 0};
    std::string pre2[] = {"A", "B", "E", "F", "C", "D", "G"};
    int deg2[] = {3, 2, 0, 0, 0, 1, 0};
    int index = 0;
    GTN *t1 = buildFromPreorder(pre1, deg1, index);
    std::printf("\nFrom U V W X Y with degrees 2 2 0 0 0: ");
    parenthesis(t1);
    index = 0;
    GTN *t2 = buildFromPreorder(pre2, deg2, index);
    std::printf("\nFrom A B E F C D G with degrees 3 2 0 0 0 1 0: ");
    parenthesis(t2);
    std::printf("\n  PostOrder: ");
    gtPostOrder(t2);
    std::printf("\n");
    destroy(t1);
    destroy(t2);
    return 0;
}
```

```
General tree: 1(2(5,6),3(7,8(11),9),4(10(12,13)))
  PreOrder:  1 2 5 6 3 7 8 11 9 4 10 12 13
  PostOrder: 5 6 2 7 11 8 9 3 12 13 10 4 1
Converted binary tree:
  PreOrder:  1 2 5 6 3 7 8 11 9 4 10 12 13
  InOrder:   5 6 2 7 11 8 9 3 12 13 10 4 1
  PostOrder: 6 5 11 9 8 7 13 12 10 4 3 2 1

From U V W X Y with degrees 2 2 0 0 0: U(V(W,X),Y)
From A B E F C D G with degrees 3 2 0 0 0 1 0: A(B(E,F),C,D(G))
  PostOrder: E F B C G D A
```

The output confirms both correspondences of Section 11.10.1: the two preorder lines are identical, and the general tree's postorder line equals the binary tree's inorder line, 5 6 2 7 11 8 9 3 12 13 10 4 1. The binary tree's postorder, 6 5 11 9 8 7 13 12 10 4 3 2 1, is different, as expected. The two rebuilt trees print in parenthesis notation as U(V(W,X),Y) and A(B(E,F),C,D(G)), the trees of **Figure 11.27**.

11.11 Search Trees in the Standard Libraries

Production programs rarely write their own search tree; they use the balanced one their language provides. The three languages of this book provide very different things.

11.11.1 C++: `std::set` and `std::map`

`std::set<K>` keeps a set of unique keys in sorted order, and `std::map<K, V>` keeps key-value pairs sorted by key. The C++ standard does not say how they are implemented, but it requires $O(\lg n)$ search, insertion and deletion and in-order iteration, and in practice all three major implementations (GNU libstdc++, LLVM libc++ and Microsoft's) use a red-black tree [11]. Every operation of Section 11.7 has a name:

- `insert` and `m[key] = value` insert; `find` searches and returns `end()` if the key is absent; `erase` deletes;
- iterating from `begin()` to `end()` is an inorder traversal, so the keys come out sorted, and `++it` and `--it` are successor and predecessor;
- `begin()` and `rbegin()` give the minimum and the maximum;
- `lower_bound(k)` returns the first key $\geq k$ and `upper_bound(k)` the first key $>k$, the bound searches of Section 2.6.3 on a tree; together they answer range queries.

Listing 11.11 keeps the Pustaka Ceria catalog of Chapter 1 in a `std::map` keyed by title.

Listing 11.11 `std_map.cpp`: the Pustaka Ceria catalog in `std::map`, with in-order listing, search, a range query, successor and predecessor, minimum and maximum, and deletion; and a `std::set` that rejects a duplicate.

```
// std_map.cpp -- the Pustaka Ceria catalog in std::map, an ordered map that the
// standard libraries implement as a red-black tree; and std::set.
#include <cstdio>
#include <iterator>
#include <map>
#include <set>
#include <string>

int main() {
    std::map<std::string, std::string> byTitle;     // title -> book code
    const char *books[][2] = {
        {"B1001", "Laskar Pelangi"},     {"B1002", "Bumi Manusia"},
        {"B1003", "Filosofi Kopi"},      {"B1004", "Sejarah Nusantara"},
        {"B1005", "Algoritma & Struktur Data"},
        {"B1006", "Cerita Rakyat Nusantara"},
        {"B1007", "Fisika Dasar"},       {"B1008", "Negeri 5 Menara"}};
    for (auto &b : books) byTitle[b[1]] = b[0];     // O(lg n) each

    std::printf("Catalog by title (an inorder walk):\n");
    for (const auto &[title, code] : byTitle)
        std::printf("  %-27s %s\n", title.c_str(), code.c_str());

    for (const char *t : {"Fisika Dasar", "Ronggeng Dukuh Paruk"}) {
        auto it = byTitle.find(t);                  // a BST search
        std::printf("find \"%s\": %s\n", t,
                    it != byTitle.end() ? it->second.c_str()
                                        : "not in the catalog");
    }

    std::printf("Titles from \"F\" up to but not including \"N\":\n ");
    auto lo = byTitle.lower_bound("F"), hi = byTitle.lower_bound("N");
    for (auto it = lo; it != hi; ++it) std::printf(" [%s]", it->first.c_str());
    std::printf("\n");

    auto it = byTitle.find("Filosofi Kopi");
    std::printf("After \"Filosofi Kopi\" comes \"%s\"\n",
                std::next(it)->first.c_str());
    std::printf("Before \"Filosofi Kopi\" comes \"%s\"\n",
                std::prev(it)->first.c_str());
    std::printf("First title \"%s\", last title \"%s\"\n",
                byTitle.begin()->first.c_str(), byTitle.rbegin()->first.c_str());

    std::size_t removed = byTitle.erase("Bumi Manusia");
    std::printf("erase \"Bumi Manusia\": %zu removed, %zu titles left\n",
                removed, byTitle.size());
```

Listing 11.11 (Continued) *std_map.cpp: the Pustaka Ceria catalog in std::map, with in-order listing, search, a range query, successor and predecessor, minimum and maximum, and deletion; and a std::set that rejects a duplicate.*

```
    std::set<int> keys;
    for (int x : {65, 5, 70, 3, 10}) keys.insert(x);
    std::printf("set from 65 5 70 3 10:");
    for (int x : keys) std::printf(" %d", x);
    bool added = keys.insert(70).second;            // false: already present
    std::printf("\ninsert 70 again: %s\n", added ? "added" : "rejected");
    return 0;
}
```

```
Catalog by title (an inorder walk):
  Algoritma & Struktur Data   B1005
  Bumi Manusia                B1002
  Cerita Rakyat Nusantara     B1006
  Filosofi Kopi               B1003
  Fisika Dasar                B1007
  Laskar Pelangi              B1001
  Negeri 5 Menara             B1008
  Sejarah Nusantara           B1004
find "Fisika Dasar": B1007
find "Ronggeng Dukuh Paruk": not in the catalog
Titles from "F" up to but not including "N":
  [Filosofi Kopi] [Fisika Dasar] [Laskar Pelangi]
After "Filosofi Kopi" comes "Fisika Dasar"
Before "Filosofi Kopi" comes "Cerita Rakyat Nusantara"
First title "Algoritma & Struktur Data", last title "Sejarah Nusantara"
erase "Bumi Manusia": 1 removed, 7 titles left
set from 65 5 70 3 10: 3 5 10 65 70
insert 70 again: rejected
```

The books were inserted in order of their codes, B1001 to B1008, and come out in alphabetical order of titles, because iteration is an inorder walk. The range query from “F” up to but not including “N” uses two `lower_bound` searches, $O(\lg n)$ each, plus one step per title reported. `std::multimap` and `std::multiset` allow repeated keys. (C++ also has `std::unordered_map`, a hash table, Chapter 15: faster for exact lookup, but with no order, no range query and no neighbors.)

11.11.2 Java: `TreeMap` and `TreeSet`

Java’s `TreeMap<K, V>` is, in the words of its documentation, “a Red-Black tree based `NavigableMap` implementation” that “provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations”, with algorithms adapted from Cormen et al. [12]. `TreeSet<K>` is a set backed by a `TreeMap`. The `NavigableMap` interface names the neighbor queries directly: `firstKey` and `lastKey` (minimum, maximum), `higherKey` and `lowerKey` (successor and predecessor of any key, present or not), `ceilingKey` and `floorKey` (smallest key $\geq k$, largest key $\leq k$), and `subMap`, `headMap` and `tailMap` for ranges. **Listing 11.12** is **Listing 11.11** in Java and prints the same output.

Listing 11.12 `CatalogTreeMap.java`: the same catalog in `java.util.TreeMap`, with `subMap`, `higherKey`, `lowerKey`, `firstKey` and `lastKey`; and a `TreeSet`.

```
// CatalogTreeMap.java -- the Pustaka Ceria catalog in java.util.TreeMap, a
// red-black tree; and TreeSet.
import java.util.Map;
import java.util.TreeMap;
import java.util.TreeSet;

public class CatalogTreeMap {
    public static void main(String[] args) {
        TreeMap<String, String> byTitle = new TreeMap<>();     // title -> book code
        String[][] books = {
            {"B1001", "Laskar Pelangi"},     {"B1002", "Bumi Manusia"},
            {"B1003", "Filosofi Kopi"},      {"B1004", "Sejarah Nusantara"},
            {"B1005", "Algoritma & Struktur Data"},
            {"B1006", "Cerita Rakyat Nusantara"},
            {"B1007", "Fisika Dasar"},       {"B1008", "Negeri 5 Menara"}};
        for (String[] b : books) byTitle.put(b[1], b[0]);      // O(lg n) each

        System.out.println("Catalog by title (an inorder walk):");
        for (Map.Entry<String, String> e : byTitle.entrySet())
            System.out.printf("  %-27s %s%n", e.getKey(), e.getValue());

        for (String t : new String[] {"Fisika Dasar", "Ronggeng Dukuh Paruk"}) {
            String code = byTitle.get(t);                      // a BST search
            System.out.println("find \"" + t + "\": "
                    + (code != null ? code : "not in the catalog"));
        }

        System.out.println("Titles from \"F\" up to but not including \"N\":");
        StringBuilder sb = new StringBuilder(" ");
        for (String t : byTitle.subMap("F", "N").keySet())
            sb.append(" [").append(t).append(']');
        System.out.println(sb);

        System.out.println("After \"Filosofi Kopi\" comes \""
                + byTitle.higherKey("Filosofi Kopi") + "\"");
        System.out.println("Before \"Filosofi Kopi\" comes \""
                + byTitle.lowerKey("Filosofi Kopi") + "\"");
        System.out.println("First title \"" + byTitle.firstKey()
                + "\", last title \"" + byTitle.lastKey() + "\"");

        int removed = byTitle.remove("Bumi Manusia") != null ? 1 : 0;
        System.out.println("erase \"Bumi Manusia\": " + removed + " removed, "
                + byTitle.size() + " titles left");
```

Listing 11.12 (Continued) *CatalogTreeMap.java: the same catalog in java.util.TreeMap, with subMap, higherKey, lowerKey, firstKey and lastKey; and a TreeSet.*

```
        TreeSet<Integer> keys = new TreeSet<>();
        for (int x : new int[] {65, 5, 70, 3, 10}) keys.add(x);
        System.out.print("set from 65 5 70 3 10:");
        for (int x : keys) System.out.print(" " + x);
        boolean added = keys.add(70);            // false: already present
        System.out.println("\ninsert 70 again: " + (added ? "added" : "rejected"));
    }
}
```

Where `std::map` needs an iterator and `std::next`, `TreeMap` answers `higherKey("Filosofi Kopi")` in one call. As with `std::map`, the hash-based alternative is `HashMap`, which has no order.

11.11.3 Python: no built-in search tree

Python's standard library has no balanced search tree. Its `dict` and `set` are hash tables (Chapter 15): exact lookup in $O(1)$ average time, and a `dict` remembers insertion order, but neither is

sorted and neither can answer “the next key after this one” or “all keys between F and N” without scanning or sorting everything. Python programs get ordered behavior in two ways.

A sorted list with `bisect`. The `bisect` module performs binary search on a sorted list: `bisect_left(a, x)` is the lower bound of Section 2.6.3, and `insort(a, x)` inserts x in its place [18]. Search is $O(\lg n)$; insertion and deletion are $O(n)$, because the list must shift, just as Section 7.3 found for arrays. That sounds bad, but the shift is a single `memmove` of pointers in C, and for lists of up to some hundreds of thousands of elements it is usually faster than a tree written in Python would be. **Listing 11.13** is the catalog again, with a sorted list of titles for the order and a `dict` for the codes, and prints the same output as **Listing 11.11**.

Listing 11.13 `library_trees.py`: the catalog kept in order with a sorted list and the `bisect` module, with a `dict` for the codes.

```
# library_trees.py -- Python has no built-in search tree. The Pustaka Ceria
# catalog kept in order with a sorted list and the bisect module instead.
from bisect import bisect_left, insort

BOOKS = [("B1001", "Laskar Pelangi"), ("B1002", "Bumi Manusia"),
         ("B1003", "Filosofi Kopi"), ("B1004", "Sejarah Nusantara"),
         ("B1005", "Algoritma & Struktur Data"),
         ("B1006", "Cerita Rakyat Nusantara"),
         ("B1007", "Fisika Dasar"), ("B1008", "Negeri 5 Menara")]


def main():
    titles = []                     # always sorted
    code_of = {}                    # title -> book code (a hash table, Chapter 15)
    for code, title in BOOKS:
        insort(titles, title)       # binary search, then an O(n) shift
        code_of[title] = code

    print("Catalog by title (an inorder walk):")
    for t in titles:
        print(f"  {t:<27} {code_of[t]}")

    for t in ("Fisika Dasar", "Ronggeng Dukuh Paruk"):
        i = bisect_left(titles, t)
        found = i < len(titles) and titles[i] == t
        print(f'find "{t}": {code_of[t] if found else "not in the catalog"}')

    print('Titles from "F" up to but not including "N":')
    lo, hi = bisect_left(titles, "F"), bisect_left(titles, "N")
    print(" " + "".join(f" [{t}]" for t in titles[lo:hi]))

    i = bisect_left(titles, "Filosofi Kopi")
    print(f'After "Filosofi Kopi" comes "{titles[i + 1]}"')
    print(f'Before "Filosofi Kopi" comes "{titles[i - 1]}"')
    print(f'First title "{titles[0]}", last title "{titles[-1]}"')

    i = bisect_left(titles, "Bumi Manusia")
    removed = 0
    if i < len(titles) and titles[i] == "Bumi Manusia":
        del titles[i]               # another O(n) shift
        del code_of["Bumi Manusia"]
        removed = 1
    print(f'erase "Bumi Manusia": {removed} removed, {len(titles)} titles left')
```

Listing 11.13 (Continued) *library_trees.py: the catalog kept in order with a sorted list and the bisect module, with a dict for the codes.*

```
    keys = sorted({65, 5, 70, 3, 10})
    print("set from 65 5 70 3 10:" + "".join(f" {x}" for x in keys))
    added = 70 not in keys
    print(f"insert 70 again: {'added' if added else 'rejected'}")


if __name__ == "__main__":
    main()
```

The `sortedcontainers` package. The third-party package `sortedcontainers` (version 2.4.0, installed with `pip install sortedcontainers`) provides `SortedList`, `SortedDict` and `SortedSet` [19]. Interestingly, it is not a tree either: it keeps a list of sorted sublists of about a thousand elements each, which makes insertions cheap (only one short sublist shifts) and keeps the memory-friendly layout of arrays. Its interface offers exactly the tree operations: `irange` for ranges, `bisect_left` and `bisect_right` for neighbors, `peekitem` for the i-th smallest key. **Listing 11.14** shows it on the same catalog.

Listing 11.14 `sorted_catalog.py`: the catalog in a `SortedDict` from the `sortedcontainers` package.

```
# sorted_catalog.py -- the same catalog with the third-party sortedcontainers
# package (pip install sortedcontainers), whose SortedDict keeps keys in order.
from sortedcontainers import SortedDict

catalog = SortedDict({"Laskar Pelangi": "B1001", "Bumi Manusia": "B1002",
                      "Filosofi Kopi": "B1003", "Sejarah Nusantara": "B1004",
                      "Algoritma & Struktur Data": "B1005",
                      "Cerita Rakyat Nusantara": "B1006",
                      "Fisika Dasar": "B1007", "Negeri 5 Menara": "B1008"})

print("First two titles:", list(catalog.keys()[:2]))
print("From F up to N:", list(catalog.irange("F", "N", inclusive=(True, False))))
i = catalog.bisect_right("Filosofi Kopi")            # position after the key
print("After Filosofi Kopi:", catalog.peekitem(i))
catalog["Ronggeng Dukuh Paruk"] = "B1009"            # stays in order
print("Last title:", catalog.peekitem(-1))
```

```
First two titles: ['Algoritma & Struktur Data', 'Bumi Manusia']
From F up to N: ['Filosofi Kopi', 'Fisika Dasar', 'Laskar Pelangi']
After Filosofi Kopi: ('Fisika Dasar', 'B1007')
Last title: ('Sejarah Nusantara', 'B1004')
```

Table 11.5 summarizes the three libraries.

Table 11.5 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`

11.12 Profitina in Practice

Profitina is an AI visibility (GEO) platform: it scans a brand's website, measures how ready the site is to be quoted by AI answer engines, from signals such as content that machines can extract, structured data and technical credibility, and helps fix what it finds [20]. Its server is a Python application built with FastAPI on PostgreSQL. Trees are everywhere in that work, because almost everything a scanner reads is a tree. We describe the shape of each problem and the class of structure that fits it; the product's own implementation choices are its own.

- **A web page is a tree.** A browser, a crawler and an answer engine all turn HTML into the *Document Object Model* (DOM), a general tree in which each element is a node whose children are the elements nested inside it [21]. "Document order", the order in which text appears, is the preorder of that tree. Whether a page's main content can be extracted cleanly is largely a question about the tree's shape: is the article text a compact subtree, or scattered among hundreds of sibling nodes of navigation and advertising?
- **Headings form an outline.** The headings `h1` to `h6` of a page define an outline tree, the table of contents of Section 11.3's level notation. A page that jumps from `h1` straight to `h4`, or has three `h1` elements, has a malformed outline, and readers and machines alike lose the structure. Checking it is a preorder walk with a stack of open headings, exactly **Pseudocode 11.3**'s pattern of pushing while going deeper and popping while coming back up.
- **Structured data is a tree.** The schema.org JSON-LD that a site embeds for answer engines is a nested object: an `Organization` with an `address`, which has a `streetAddress`, and so on. JSON is parenthesis notation with braces, and validating it is a recursive traversal that checks each subtree against the type expected at that depth.
- **A site is a tree of links.** Starting from the home page and following links breadth-first (level order, **Pseudocode 11.2**) assigns every page a *crawl depth*, its level in the resulting tree. Pages many clicks deep are found later, or not at all, by crawlers; a scanner that reports crawl depth is reporting node depths.
- **Discussion threads are trees.** In the real-time community, a reply hangs below the post it answers, and replies to replies below those. Displaying a thread is a preorder traversal printed in level notation: each post indented by its depth.
- **Ordered data lives in B-trees.** The weekly score trend of a brand and the "scans between July and September" export are range queries on time. In PostgreSQL such a query is answered by a B-tree index (Section 11.9.4) on the timestamp column [15]: two descents to the ends of the

range and an in-order scan between them, the same shape as the range query of the program below.

The runnable version of these ideas uses the Profitina Laundry case study. Its twelve orders of one day (the orders of Section 2.13.2) are each ready for pickup a fixed time after they arrive: one hour plus 20 minutes per kilogram. The pickup board keeps the orders in a BST keyed by ready time, and answers the counter's questions: which orders will be ready between 10:00 and 12:59, which order will be ready next at 13:05, and which is the last order that is already ready. **Listing 11.15** implements the board; its range query visits only the subtrees that can contain times in the range, `ceiling` and `floorNode` are the successor-style searches of Section 11.7.9 for a key that need not be in the tree, and picked-up orders are removed with the successor variant of deletion.

Listing 11.15 `laundry_tree.cpp`: Profitina Laundry's pickup board, a binary search tree of orders keyed by ready time, with a pruned range query, ceiling, floor and deletion.

```
// laundry_tree.cpp -- Profitina Laundry's pickup board: orders in a binary search
// tree keyed by the time they are ready, with range, floor and ceiling queries.
#include <cstdio>
#include <string>

struct Order {
    int number;
    std::string customer;
    int ready;              // minutes after midnight
};

struct Node {
    Order order;
    Node *left, *right;
};

std::string hhmm(int m) {
    char buf[16];
    std::snprintf(buf, sizeof buf, "%02d:%02d", m / 60 % 24, m % 60);
    return buf;
}

Node *insert(Node *t, const Order &o) {
    if (t == nullptr) return new Node{o, nullptr, nullptr};
    if (o.ready < t->order.ready) t->left = insert(t->left, o);
    else t->right = insert(t->right, o);        // equal times go right
    return t;
}

void board(const Node *t) {                     // inorder: earliest first
    if (t == nullptr) return;
    board(t->left);
    std::printf("  %s  #%d %s\n", hhmm(t->order.ready).c_str(), t->order.number,
                t->order.customer.c_str());
    board(t->right);
}
```

Listing 11.15 (Continued) *laundry_tree.cpp: Profitina Laundry's pickup board, a binary search tree of orders keyed by ready time, with a pruned range query, ceiling, floor and deletion.*

```
// 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);
}

const Node *ceiling(const Node *t, int x) {    // earliest order ready at or after x
    const Node *best = nullptr;
    while (t != nullptr) {
        if (t->order.ready >= x) {
            best = t;                           // a candidate; try earlier
            t = t->left;
        } else {
            t = t->right;
        }
    }
    return best;
}

const Node *floorNode(const Node *t, int x) {  // latest order ready at or before x
    const Node *best = nullptr;
    while (t != nullptr) {
        if (t->order.ready <= x) {
            best = t;
            t = t->right;
        } else {
            t = t->left;
        }
    }
    return best;
}
```

Listing 11.15 (Continued) *laundry_tree.cpp: Profitina Laundry's pickup board, a binary search tree of orders keyed by ready time, with a pruned range query, ceiling, floor and deletion.*

```
Node *remove(Node *t, int ready) {              // successor variant
    if (t == nullptr) return nullptr;
    if (ready < t->order.ready) {
        t->left = remove(t->left, ready);
    } else if (ready > t->order.ready) {
        t->right = remove(t->right, ready);
    } else if (t->left == nullptr || t->right == nullptr) {
        Node *child = (t->left != nullptr) ? t->left : t->right;
        delete t;
        return child;
    } else {
        Node *s = t->right;
        while (s->left != nullptr) s = s->left;
        t->order = s->order;
        t->right = remove(t->right, s->order.ready);
    }
    return t;
}

int height(const Node *t) {
    if (t == nullptr) return -1;
    int a = height(t->left), b = height(t->right);
    return (a > b ? a : b) + 1;
}

int count(const Node *t) {
    return t == nullptr ? 0 : count(t->left) + count(t->right) + 1;
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}
```

Listing 11.15 (Continued) *laundry_tree.cpp: Profitina Laundry's pickup board, a binary search tree of orders keyed by ready time, with a pruned range query, ceiling, floor and deletion.*

```
int main() {
    struct Row {
        int number;
        const char *received, *customer;
        double kg;
    };
    Row in[] = {
        {1001, "07:45", "Sari", 3.5},  {1002, "08:10", "Budi", 5.0},
        {1003, "08:55", "Dewi", 2.0},  {1004, "09:30", "Andi", 7.5},
        {1005, "10:05", "Rina", 4.0},  {1006, "10:40", "Joko", 6.0},
        {1007, "11:15", "Maya", 3.0},  {1008, "11:50", "Tono", 8.0},
        {1009, "12:20", "Lina", 2.5},  {1010, "13:05", "Agus", 5.5},
        {1011, "14:30", "Fitri", 4.5}, {1012, "15:10", "Hadi", 3.0}};
    Node *root = nullptr;
    for (auto &r : in) {                        // ready = received + 60 + 20 per kg
        int h, m;
        std::sscanf(r.received, "%d:%d", &h, &m);
        int ready = h * 60 + m + 60 + (int) (20 * r.kg);
        root = insert(root, Order{r.number, r.customer, ready});
    }
    std::printf("Pickup board (%d orders, tree height %d):\n",
                count(root), height(root));
    board(root);

    int visited = 0;
    std::printf("Ready between 10:00 and 12:59:\n");
    range(root, 600, 779, visited);
    std::printf("  (%d of %d nodes visited)\n", visited, count(root));

    int now = 13 * 60 + 5;
    const Node *c = ceiling(root, now), *f = floorNode(root, now);
    std::printf("At %s the next order to be ready is #%d %s at %s\n",
                hhmm(now).c_str(), c->order.number, c->order.customer.c_str(),
                hhmm(c->order.ready).c_str());
    std::printf("The last order already ready is #%d %s at %s\n",
                f->order.number, f->order.customer.c_str(),
                hhmm(f->order.ready).c_str());
```

Listing 11.15 (Continued) *laundry_tree.cpp: Profitina Laundry's pickup board, a binary search tree of orders keyed by ready time, with a pruned range query, ceiling, floor and deletion.*

```
    std::printf("Picked up: #%d %s and #1003 Dewi\n", f->order.number,
                f->order.customer.c_str());
    root = remove(root, f->order.ready);
    root = remove(root, 10 * 60 + 35);
    std::printf("Pickup board (%d orders, tree height %d):\n",
                count(root), height(root));
    board(root);
    destroy(root);
    return 0;
}
```

```
Pickup board (12 orders, tree height 7):
  09:55  #1001 Sari
  10:35  #1003 Dewi
  10:50  #1002 Budi
  12:25  #1005 Rina
  13:00  #1004 Andi
  13:15  #1007 Maya
  13:40  #1006 Joko
  14:10  #1009 Lina
  15:30  #1008 Tono
  15:55  #1010 Agus
  17:00  #1011 Fitri
  17:10  #1012 Hadi
Ready between 10:00 and 12:59:
  10:35  #1003 Dewi
  10:50  #1002 Budi
  12:25  #1005 Rina
  (5 of 12 nodes visited)
At 13:05 the next order to be ready is #1007 Maya at 13:15
The last order already ready is #1004 Andi at 13:00
Picked up: #1004 Andi and #1003 Dewi
Pickup board (10 orders, tree height 7):
  09:55  #1001 Sari
  10:50  #1002 Budi
  12:25  #1005 Rina
  13:15  #1007 Maya
  13:40  #1006 Joko
  14:10  #1009 Lina
  15:30  #1008 Tono
  15:55  #1010 Agus
  17:00  #1011 Fitri
  17:10  #1012 Hadi
```

Figure 11.28 draws the board's tree and marks the nodes the range query visited.



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.

Two lessons come out of the run. The range query visited 5 of the 12 nodes: at 13:00 it did not go right, because nothing in that subtree can be ready before 12:59, and at 09:55 it did not go left. And the tree's height is 7, although 12 nodes fit in a tree of height 3 by Equation (11.6). The reason is Section 11.8's warning in a real setting: orders arrive in time order, and ready times mostly follow arrival times, so the keys come in nearly sorted and the tree leans to the right. For a

board of a dozen orders that does not matter; for a year of orders it would. A real system would keep the board in a balanced structure, a `std::map` or `TreeMap` keyed by ready time, whose `lower_bound`, `subMap`, `ceilingKey` and `floorKey` are exactly the queries of this program. The Java and Python versions (`LaundryTree.java`, `laundry_tree.py`) print the same output.

11.13 Common Mistakes, and How to Avoid Them

Confusing depth and height. Depth counts edges down from the root to a node; height counts edges from a node down to its deepest leaf. The root has depth 0 and the height of the tree; a leaf has height 0. Check which convention a text uses: some count levels from 1 and heights in nodes, which adds 1 to every answer (Sections 11.2.3 and 11.7.5).

Checking the BST property only between parent and children. Every key in the left *subtree* must be smaller, not only the left child. The tree 10(5(2, 12), 15) passes every parent-child test and is not a BST. Check with bounds passed down from the ancestors (Exercise 11.8).

Losing the change to a pointer. `void insert(Node *root, int x)` receives a *copy* of the pointer; assigning the new node to it changes the copy, and the tree never grows. Pass `Node **root` or `Node *&root`, or return the new subtree root and store it: `t->left = insert(t->left, x)` (Section 11.7.2). In Java and Python, forgetting the assignment `root = insert(root, x)` loses the first node the same way.

Deleting a node with two children by splicing. Case 2's "replace the node by its child" does not work when there are two children. Replace the key by the successor's (or predecessor's) key and delete that node from the subtree, where it has at most one child (Section 11.7.10).

Forgetting the base case, or dereferencing NIL. Every recursive tree function starts with `if (root == nullptr)`. `min(root)->data` on an empty tree, or `root->left->data` on a node without a left child, is undefined behavior (Section 7.11).

Releasing a tree in the wrong order, or not at all. Delete the children before the parent (postorder); deleting the parent first loses the pointers to its children. Setting `root = nullptr` without deleting leaks every node. Run tree programs under Valgrind or AddressSanitizer (Section 11.7.7).

Assuming $O(\lg n)$ for a plain BST. The cost is $O(h)$, and h can be $n - 1$: sorted or nearly sorted input, which is common, turns a BST into a list and building it into a $\Theta(n^2)$ job (Section 11.8.2). Use a balanced tree (`std::map`, `TreeMap`) when the input order is not under your control.

Recursing on a tree that may be degenerate. Recursion depth equals height. A chain of a million nodes overflows the C++ stack, and Python's default limit stops at about a thousand (Section 10.12). Use iterative insertion and search, and explicit stacks for traversals, when the tree is not known to be balanced (Section 11.6.4).

Using a traversal for a job that needs another. Deleting needs postorder; copying and saving need preorder; sorted output needs inorder; "nearest to the root first" needs level order with a queue.

Mixing up the names of shapes. "Full", "complete" and "perfect" mean different things in different books (Section 11.5.2); say what you mean by the condition, not only by the name.

11.14 Chapter Summary

- A **tree** is a root with zero or more disjoint subtrees; it is non-linear and hierarchical. A tree of n nodes has $n-1$ edges (Equation (11.1)) and exactly one path from the root to each node. Its vocabulary (parent, child, sibling, leaf, internal node, ancestor, descendant, path, level, depth, height, degree, subtree, size) is summarized in **Table 11.1**.
- A tree can be drawn with levels, as a Venn diagram, in level (indentation) notation, or in parenthesis notation such as `(A(B(D,E(I,J)),C(F,G,H)))`.
- **Representations.** A binary tree fits an array with children at $2i+1$ and $2i+2$ (Equation (11.2)), which is ideal for complete trees and wasteful for others; the dynamic representation uses `left` and `right` pointers and has $n+1$ null pointers (Equation (11.3)). Any tree can be stored with first-child and next-sibling pointers, which makes it a binary tree.
- **Binary trees** are full, perfect, complete or skewed (**Table 11.2**). Level ℓ holds at most 2^{ℓ} nodes (Equation (11.4)), a tree of height h at most $2^{h+1}-1$ (Equation (11.5)), so $h \geq \lfloor \lg n \rfloor$ (Equation (11.6)); a full tree has one more leaf than internal nodes (Equation (11.7)).
- **Traversals.** Preorder, inorder and postorder (**Pseudocode 11.1**) are the three times an Euler tour passes a node; level order uses a queue (**Pseudocode 11.2**); all four can run with explicit stacks or queues (**Pseudocode 11.3**) and all take $\Theta(n)$ time (Equation (11.8)). The three depth-first orders of an expression tree give prefix, infix and postfix notation, and postfix builds the tree with a stack (**Pseudocode 11.4**).
- A **binary search tree** keeps smaller keys in the left subtree and larger ones in the right subtree of every node, so inorder is sorted. Insertion (**Pseudocode 11.5**, running example `insert(65, 5, 70, 3, 10)`), search (**Pseudocode 11.6**), minimum and maximum (**Pseudocode 11.7**), successor and predecessor (**Pseudocode 11.9**) and deletion (**Pseudocode 11.10**: a leaf is unlinked, a node with one child is replaced by it, a node with two children takes its successor's or its predecessor's key) all cost $O(h)$ (Equation (11.10), **Table 11.3**). Count, leaves and height (Equation (11.9)) cost $\Theta(n)$; printing sideways is a reverse inorder (**Pseudocode 11.8**).
- **Shape.** Random insertion orders give an average depth of about $1.39 \lg n$ (Equation (11.12)); sorted input, and 2^{n-1} orders in all (Equation (11.11)), give a chain of height $n-1$, measured in **Listing 11.8** as a $\Theta(n^2)$ build.
- **Self-balancing trees** repair their shape with $O(1)$ rotations that preserve the inorder sequence. AVL trees keep every balance factor in $\{-1,0,1\}$ and have height below $1.4405 \lg(n+2)$ (Equation (11.13)); red-black trees keep five color rules and have height at most $2 \lg(n+1)$ (Equation (11.14)) with fewer rotations (**Table 11.4**).
- **General trees** convert to binary trees (first child left, next sibling right), with general preorder equal to binary preorder and general postorder equal to binary inorder; a preorder listing with degrees determines a tree.
- **Libraries.** C++ `std::set` and `std::map` and Java `TreeSet` and `TreeMap` are red-black trees with successor, predecessor and range queries; Python has no built-in search tree and uses a sorted list with `bisect` or the third-party `sortedcontainers` (**Table 11.5**).

11.15 Exercises

11.1 For the tree `(P(Q(T,U(W)),R,S(V)))` in parenthesis notation, give: the root; the internal nodes and the leaves; the children of Q; the siblings of R; the ancestors of W; the descendants of Q; the depth of U and the height of Q; the height and the degree of the tree; its size and number of edges.

11.2 Draw the tree of Exercise 11.1 with its levels, and describe its Venn diagram and its level (indentation) notation. Which nodes are on each level?

11.3 Insert 40, 20, 60, 10, 30, 50, 70 into an empty binary search tree. Draw the tree after every insertion, and give its height and its preorder, inorder, postorder and level-order traversals. What is the height if the same keys are inserted in ascending order?

11.4 Store the tree built in Exercise 11.3 in an array as in Section 11.4.1. Which key is in slot 5, and where is its parent by Equation (11.2)? Store the tree of `insert(65, 5, 70, 3, 10)` in the same way: how many slots are empty, and why?

11.5 (a) In a binary tree, let n_0, n_1 and n_2 be the numbers of nodes with 0, 1 and 2 children. Prove that $n_0 = n_2 + 1$, whatever the shape. (b) Between which two values must the number of nodes of a complete binary tree of height h lie? Deduce that its height is $\lfloor \lg n \rfloor$.

11.6 A binary tree has preorder A B D G C E F H and inorder D G B A E C H F. Draw it, give its postorder, and write `Node *buildPreIn(...)`, which rebuilds a binary tree from its preorder and inorder listings. Then show with a tree of two nodes that preorder and postorder together do *not* determine a binary tree.

11.7 In the ten-node tree of **Figure 11.18** (inserted as 50, 30, 70, 20, 40, 60, 80, 35, 45, 65), delete 30, first with the successor variant and then, starting again from the original tree, with the predecessor variant. Draw both results. Then delete 70 from the original tree with each variant.

11.8 Write `bool isBST(const Node *t, long long lo, long long hi)`, which decides whether a binary tree is a binary search tree. Explain why the check "each node's left child is smaller and right child larger" is not enough, using the tree 10(5(2, 12), 15).

11.9 Write a function that returns the node holding the k-th smallest key of a BST ($k = 1$ is the minimum), or `nullptr` if the tree has fewer than k nodes. What is its running time?

11.10 Write `Node *buildBalanced(const std::vector<int> &a, int lo, int hi)`, which builds a BST of minimum height from the sorted keys `a[lo..hi]`. What is its running time, and what height does it produce for $n = 7$, 1000 and 1,000,000 keys?

11.11 The *lowest common ancestor* of two keys x and y in a BST is the deepest node that has both in its subtree (a node counts as in its own subtree). Write a function that finds it in $O(h)$ time without recursion, and give the lowest common ancestors of (35, 45), (35, 65), (20, 35) and (60, 65) in the tree of Exercise 11.7.

11.12 Draw every binary search tree that holds the keys 1, 2, 3, and for each one list the insertion orders that produce it. Check Equation (11.11) for $n = 3$.

11.13 Build the expression tree of the postfix expression `8 2 / 3 - 4 5 * +` with **Pseudocode 11.4**, drawing the stack after each symbol. Give the prefix and fully parenthesized infix forms and the value.

11.14 `bst.cpp` (**Listing 11.4**) works for the trees of this chapter. What happens if its `main` inserts the keys 1 to 1,000,000 in ascending order and then calls `height`? Estimate the stack memory needed if each call frame takes 48 bytes, and compare with the usual 8 MB limit on Linux. What happens in `bst.py` with 2,000 ascending keys? Suggest two fixes.

11.15 For each Pustaka Ceria task, choose a structure among a sorted array with binary search (Chapter 2), a hash table (`std::unordered_map`, Chapter 15), and a balanced search tree (`std::map`), and justify the choice: (a) look up a book by its exact code, many times per second,

with frequent additions; (b) show the catalog in alphabetical order of titles after every addition; (c) list all titles that begin with "Ne"; (d) a fixed reference list of 5,000 subject headings, loaded once a year and searched constantly.

11.16 Insert 10, 20, 15 into an empty AVL tree. At which node does the balance factor become ± 2, why is a single rotation not enough, and which two rotations repair it? Draw the result.

11.16 Solutions to the Exercises

11.1 The root is P. Internal nodes: P, Q, U, S; leaves: T, W, R, V. The children of Q are T and U; the siblings of R are Q and S; the ancestors of W are U, Q and P; the descendants of Q are T, U and W. U has depth 2; Q has height 2 (the path Q, U, W). The tree has height 3 (P, Q, U, W) and degree 3 (P has three children). It has 9 nodes and, by Equation (11.1), 8 edges.

11.2 Level 0: P. Level 1: Q, R, S. Level 2: T, U, V. Level 3: W. In the Venn diagram the region P contains three disjoint regions Q, R and S; Q contains T and a region U, which contains W; S contains V; R is a single small region. In level notation, one node per line in preorder, indented by depth: P; Q (one step); T and U (two steps); W (three steps); R (one step); S (one step); V (two steps). The parenthesis notation is the one given, `(P(Q(T,U(W)),R,S(V)))`.

11.3 40 becomes the root; 20 goes left of 40 and 60 right; 10 goes left of 20, 30 right of 20, 50 left of 60 and 70 right of 60. The result is the perfect tree of height 2. Preorder: 40 20 10 30 60 50 70. Inorder: 10 20 30 40 50 60 70. Postorder: 10 30 20 50 70 60 40. Level order: 40 20 60 10 30 50 70. In ascending order the keys build a right-skewed chain of height 6.

11.4 The array is 40, 20, 60, 10, 30, 50, 70 in slots 0 to 6. Slot 5 holds 50; its parent is at $\lfloor (5-1)/2 \rfloor = 2$, which holds 60, correct. The tree of `insert(65, 5, 70, 3, 10)` is stored as 65, 5, 70, 3, 10 in slots 0 to 4 with no empty slot, because the tree is complete: its last level (3, 10) is filled from the left.

11.5 (a) The tree has $n = n_0 + n_1 + n_2$ nodes and, counting each node's edges to its children, $n_1 + 2n_2$ edges. By Equation (11.1) that is $n - 1$, so $n_0 + n_1 + n_2 - 1 = n_1 + 2n_2$, which gives $n_0 = n_2 + 1$. Equation (11.7) is the special case $n_1 = 0$. (b) Levels 0 to $h-1$ are full, which is $2^h - 1$ nodes by Equation (11.5), and the last level holds between 1 and 2^h nodes, so $2^h \le n \le 2^{h+1} - 1$. Taking logarithms, $h \le \lg n < h + 1$, so $h = \lfloor \lg n \rfloor$.

11.6 The first preorder key, A, is the root. In the inorder listing, the keys left of A (D G B) form the left subtree and those right of it (E C H F) the right subtree. Repeat on each part: the next preorder key, B, is the root of the left part, and D G are to its left in inorder, so B has only a left subtree, rooted at the next preorder key, D, which has G to its right. On the right, C is the root with E on its left and H F on its right; F is the root of that part with H to its left. So A has children B and C; B has left child D; D has right child G; C has children E and F; F has left child H. The postorder is G D B E H F C A, and the height is 3. The function `buildPreIn` in **Listing 11.16** does exactly this: a shared index walks through the preorder listing, and the position of each root in the inorder listing splits the remaining keys. The program prints the postorder and the height of the rebuilt tree, confirming the hand solution.

Listing 11.16 `exercises.cpp`: solutions to Exercises 11.6 and 11.8 to 11.11.

```cpp
// exercises.cpp -- solutions to the programming exercises of Chapter 11.
#include <climits>
#include <cstdio>
#include <string>
#include <vector>

struct Node {
    int data;
    Node *left, *right;
};

Node *insert(Node *t, int x) {
    if (t == nullptr) return new Node{x, nullptr, nullptr};
    if (x < t->data) t->left = insert(t->left, x);
    else if (x > t->data) t->right = insert(t->right, x);
    return t;
}

int height(const Node *t) {
    if (t == nullptr) return -1;
    int a = height(t->left), b = height(t->right);
    return (a > b ? a : b) + 1;
}

void destroy(Node *t) {
    if (t == nullptr) return;
    destroy(t->left);
    destroy(t->right);
    delete t;
}

void postOrderChars(const Node *t) {
    if (t == nullptr) return;
    postOrderChars(t->left);
    postOrderChars(t->right);
    std::printf("%c ", (char) t->data);
}
```

Listing 11.16 (Continued) *exercises.cpp: solutions to Exercises 11.6 and 11.8 to 11.11.*

```
// Exercise 11.6: rebuild a tree from its preorder and inorder listings.
Node *buildPreIn(const std::string &pre, const std::string &in, int &p,
                 int lo, int hi) {
    if (lo > hi) return nullptr;            // no keys for this subtree
    char rootKey = pre[p++];                // the next preorder key is this root
    int m = (int) in.find(rootKey, lo);     // inorder: left | root | right
    Node *t = new Node{rootKey, nullptr, nullptr};
    t->left = buildPreIn(pre, in, p, lo, m - 1);
    t->right = buildPreIn(pre, in, p, m + 1, hi);
    return t;
}

// Exercise 11.8: is every key inside the open interval its ancestors allow?
bool isBST(const Node *t, long long lo, long long hi) {
    if (t == nullptr) return true;
    if (t->data <= lo || t->data >= hi) return false;
    return isBST(t->left, lo, t->data) && isBST(t->right, t->data, hi);
}

bool isBSTLocalOnly(const Node *t) {        // the tempting but WRONG check
    if (t == nullptr) return true;
    if (t->left != nullptr && t->left->data > t->data) return false;
    if (t->right != nullptr && t->right->data < t->data) return false;
    return isBSTLocalOnly(t->left) && isBSTLocalOnly(t->right);
}

// Exercise 11.9: the k-th smallest key (k counts from 1), by an inorder walk.
const Node *kthSmallest(const Node *t, int &k) {
    if (t == nullptr) return nullptr;
    const Node *r = kthSmallest(t->left, k);
    if (r != nullptr) return r;
    if (--k == 0) return t;                 // t is the k-th node visited
    return kthSmallest(t->right, k);
}

// Exercise 11.10: a balanced BST from sorted keys a[lo..hi].
Node *buildBalanced(const std::vector<int> &a, int lo, int hi) {
    if (lo > hi) return nullptr;
    int mid = lo + (hi - lo) / 2;           // the middle key becomes the root
    return new Node{a[mid], buildBalanced(a, lo, mid - 1),
                    buildBalanced(a, mid + 1, hi)};
}
```

Listing 11.16 (Continued) *exercises.cpp: solutions to Exercises 11.6 and 11.8 to 11.11.*

```
// Exercise 11.11: lowest common ancestor of keys x and y in a BST.
const Node *lca(const Node *t, int x, int y) {
    while (t != nullptr) {
        if (x < t->data && y < t->data) t = t->left;         // both on the left
        else if (x > t->data && y > t->data) t = t->right;   // both on the right
        else return t;                                       // they split here
    }
    return nullptr;
}

int main() {
    int p = 0;
    std::string pre = "ABDGCEFH", in = "DGBAECHF";
    Node *t = buildPreIn(pre, in, p, 0, (int) in.size() - 1);
    std::printf("11.6  rebuilt tree, postorder: ");
    postOrderChars(t);
    std::printf("(height %d)\n", height(t));
    destroy(t);

    int keys[] = {50, 30, 70, 20, 40, 60, 80, 35, 45, 65};
    Node *b = nullptr;
    for (int x : keys) b = insert(b, x);
    Node *bad = new Node{10, new Node{5, new Node{2, nullptr, nullptr},
                                         new Node{12, nullptr, nullptr}},
                             new Node{15, nullptr, nullptr}};
    std::printf("11.8  tree B: isBST %s; bad tree: isBST %s, local check says %s\n",
                isBST(b, LLONG_MIN, LLONG_MAX) ? "true" : "false",
                isBST(bad, LLONG_MIN, LLONG_MAX) ? "true" : "false",
                isBSTLocalOnly(bad) ? "true" : "false");
    destroy(bad);

    std::printf("11.9  tree B:");
    for (int k : {1, 4, 10, 11}) {
        int kk = k;
        const Node *r = kthSmallest(b, kk);
        if (r != nullptr) std::printf(" k=%d -> %d;", k, r->data);
        else std::printf(" k=%d -> none", k);
    }
    std::printf("\n");
```

Listing 11.16 (Continued) *exercises.cpp: solutions to Exercises 11.6 and 11.8 to 11.11.*

```
    for (int n : {7, 1000, 1000000}) {
        std::vector<int> a(n);
        for (int i = 0; i < n; ++i) a[i] = i + 1;
        Node *r = buildBalanced(a, 0, n - 1);
        std::printf("11.10 n=%d: root %d, height %d\n", n, r->data, height(r));
        destroy(r);
    }

    std::printf("11.11 tree B:");
    int pairs[][2] = {{35, 45}, {35, 65}, {20, 35}, {60, 65}};
    for (auto &q : pairs)
        std::printf(" lca(%d,%d)=%d", q[0], q[1], lca(b, q[0], q[1])->data);
    std::printf("\n");
    destroy(b);
    return 0;
}
```

```
11.6  rebuilt tree, postorder: G D B E H F C A (height 3)
11.8  tree B: isBST true; bad tree: isBST false, local check says true
11.9  tree B: k=1 -> 20; k=4 -> 40; k=10 -> 80; k=11 -> none
11.10 n=7: root 4, height 2
11.10 n=1000: root 500, height 9
11.10 n=1000000: root 500000, height 19
11.11 tree B: lca(35,45)=40 lca(35,65)=50 lca(20,35)=30 lca(60,65)=60
```

For the second part: the trees “A with left child B” and “A with right child B” both have preorder A B and postorder B A, so the two listings cannot tell them apart. (Inorder can: it is B A for the first and A B for the second.)

11.7 Deleting 30 (two children: 20, and 40 with children 35 and 45). *Successor variant:* the minimum of the right subtree is 35, a leaf; 35 replaces 30 and the leaf is removed, so the left subtree of 50 becomes 35 with children 20 and 40, and 40 keeps only its right child 45. *Predecessor variant:* the maximum of the left subtree is 20, a leaf; 20 replaces 30 and the leaf is removed, so 20 has no left child and has right child 40 (with 35 and 45). Deleting 70 from the original tree (two children: 60 with right child 65, and 80). *Successor variant:* the successor is 80, a leaf; 80 replaces 70 and 70’s old right position becomes empty, so 80 has left child 60, which keeps its right child 65. *Predecessor variant:* the predecessor is 65, the maximum of the left subtree, a leaf; 65 replaces 70, and 65 has children 60 and 80. All four results have sorted inorder sequences.

11.8 `isBST` in **Listing 11.16** passes down an open interval (lo, hi) that every key of the current subtree must fall into; going left narrows hi to the node’s key, going right narrows lo. The tree 10(5(2, 12), 15) fails at 12: it is in the left subtree of 10, so it must be below 10, but it is 12. The local check sees only that $12 > 5$ (a correct right child of 5) and $5 < 10$ (a correct left child of 10), and accepts the tree; the output shows `isBST false` and `local check says true`. The type `long long` with `LLONG_MIN` and `LLONG_MAX` makes the initial bounds wider than any `int` key.

11.9 `kthSmallest` performs an inorder traversal and counts the visits down from k; the node at which the count reaches 0 is the answer, and the search stops at once. It visits the k smallest keys plus the nodes on the path to them, so its time is $O(h + k)$, and $\Theta(n)$ in the worst case. On Tree B the output gives 20, 40, 80 for $k = 1,4,10$ and no node for $k = 11$. (Storing in each node the size of its subtree reduces this to $O(h)$, the idea of the *order-statistic trees* of Cormen et al. [2].)

11.10 `buildBalanced` makes the middle key the root and builds the two halves recursively; each key is placed once with $O(1)$ work, so the time is $\Theta(n)$ after the keys are sorted. The recursion always halves, so its depth is about lg$n$, and the tree has the minimum height of Equation (11.6): the output shows height 2 for 7 keys, 9 for 1,000 and 19 for 1,000,000, which are $\lfloor \lg n \rfloor$.

11.11 Walk down from the root: if both keys are smaller than the current key, the common ancestor is in the left subtree; if both are larger, in the right subtree; otherwise the keys are on different sides (or one of them is the current key), and the current node is the lowest common ancestor. This follows one path, $O(h)$, and needs no recursion. The output gives lca(35, 45) = 40, lca(35, 65) = 50, lca(20, 35) = 30 and lca(60, 65) = 60 (65 is in the subtree of 60).

11.12 There are five trees. 1, 2, 3 gives a right chain; 3, 2, 1 a left chain; 1, 3, 2 gives 1 with right child 3, whose left child is 2; 3, 1, 2 gives 3 with left child 1, whose right child is 2; and both 2, 1, 3 and 2, 3, 1 give the perfect tree with root 2. Four of the six orders give a tree of height $n - 1 = 2$, and $2^{3-1} = 4$, as Equation (11.11) says. (In general the number of binary trees with n nodes is the Catalan number $\frac{1}{n+1}\binom{2n}{n}$, which is 5 for $n = 3$.)

11.13 Reading `8 2 / 3 - 4 5 * +`: push 8, push 2; `/` pops 2 (right) and 8 (left) and pushes the tree 8 / 2; push 3; `-` combines (8 / 2) and 3; push 4, push 5; `*` combines 4 and 5; `+` combines ((8 / 2) - 3) and (4 * 5). The root is `+`, with left subtree `-` (whose left subtree is `/` over 8 and 2, and right child 3) and right subtree `*` over 4 and 5. Prefix: `+ - / 8 2 3 * 4 5`. Infix: `( ( ( 8 / 2 ) - 3 ) + ( 4 * 5 ) )`. Value: $8/2 = 4$, $4 - 3 = 1$, $4 \cdot 5 = 20$, $1 + 20 = 21$.

11.14 Ascending keys build a right chain of height 999,999. Each `insert` then recurses once per level, so the last insertions, and `height`, `count` and `clear`, nest up to a million calls. At 48 bytes per frame that is about 48 MB of stack, six times the usual 8 MB limit on Linux (and 48 times the 1 MB default on Windows), so the program crashes with a stack overflow (Section 10.12.1); long before that, the $\Theta(n^2)$ insertion time of Section 11.8.2 would take hours. In `bst.py` the recursion limit of about 1,000 is reached at the 1,000th key or so, and Python raises `RecursionError`. Fixes: write insertion, search and height iteratively (as `height_experiment.cpp` does); or, better, keep the tree balanced (an AVL tree, or `std::map`), so that the depth never exceeds about $1.44 \lg n$.

11.15 (a) A hash table: exact lookups in $O(1)$ average time and cheap additions; nothing in the task needs order. (b) A balanced search tree: each addition is $O(\lg n)$ and the alphabetical listing is an inorder walk; a sorted array would pay $\Theta(n)$ shifting per addition, and a hash table would have to sort for every listing. (c) A balanced search tree (or a sorted array): the titles beginning with "Ne" are those from `lower_bound("Ne")` up to `lower_bound("Nf")`, two $O(\lg n)$ searches plus one step per result; a hash table cannot answer it without scanning everything. (d) A sorted array with binary search: the data never changes, so insertion cost is irrelevant, and an array has no pointer overhead and the best cache behavior (Section 7.3.2).

11.16 After inserting 10 and 20, the tree is 10 with right child 20. Inserting 15 goes right of 10 and left of 20. Now node 10 has a left subtree of height -1 and a right subtree of height 1: balance factor -2. The deep part is 15, on the *inner* side (the left subtree of 10's right child), so a single left rotation at 10 would only move the problem: 20 would become the root with left child 10, and 15 would become 10's right child, giving balance factor $+2$ at 20. The fix is the double rotation of **Listing 11.9**'s "right-left" line: first a right rotation at 20, which makes 15 the right child of 10 with right child 20 (a straight chain), then a left rotation at 10. The result is 15 with children 10 and 20, height 1.

References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, sec. 2.3.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 12–13.

[3] P. F. Windley, "Trees, forests and rearranging," *The Computer Journal*, vol. 3, no. 2, pp. 84–88, 1960.

[4] T. N. Hibbard, "Some combinatorial properties of certain trees with applications to searching and sorting," *Journal of the ACM*, vol. 9, no. 1, pp. 13–28, Jan. 1962.

[5] G. M. Adelson-Velsky and E. M. Landis, "An algorithm for the organization of information," *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1263, 1962.

[6] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance algorithms," *Acta Informatica*, vol. 1, no. 4, pp. 290–306, 1972.

[7] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Annual Symposium on Foundations of Computer Science*, Ann Arbor, MI, USA, Oct. 1978, pp. 8–21.

[8] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, secs. 6.2.2–6.2.3.

[9] J. L. Eppinger, "An empirical study of insertion and deletion in binary search trees," *Communications of the ACM*, vol. 26, no. 9, pp. 663–669, Sep. 1983.

[10] B. Reed, "The height of a random binary search tree," *Journal of the ACM*, vol. 50, no. 3, pp. 306–332, May 2003.

[11] GNU Project, "libstdc++-v3/include/bits/stl_tree.h," GCC source repository. [Online]. Available: https://gcc.gnu.org/git/?p=gcc.git;a=blob;f=libstdc%2B%2B-v3/include/bits/stl_tree.h. [Accessed: 24-Sep-2026].

[12] Oracle, "Class TreeMap<K,V>," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/TreeMap.html. [Accessed: 24-Sep-2026].

[13] The Linux Kernel Documentation, "Red-black Trees (rbtree) in Linux." [Online]. Available: https://docs.kernel.org/core-api/rbtree.html. [Accessed: 24-Sep-2026].

[14] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.

[15] PostgreSQL Global Development Group, "B-Tree Indexes," *PostgreSQL Documentation* (current version). [Online]. Available: https://www.postgresql.org/docs/current/btree.html. [Accessed: 24-Sep-2026].

[16] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, vol. 32, no. 3, pp. 652–686, Jul. 1985.

[17] R. Seidel and C. R. Aragon, "Randomized search trees," *Algorithmica*, vol. 16, no. 4–5, pp. 464–497, 1996.

[18] Python Software Foundation, "bisect — Array bisection algorithm," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/bisect.html. [Accessed: 24-Sep-2026].

[19] G. Jenks, "Sorted Containers," version 2.4.0. [Online]. Available: https://grantjenks.com/docs/sortedcontainers/ and https://pypi.org/project/sortedcontainers/. [Accessed: 24-Sep-2026].

[20] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

[21] WHATWG, "DOM Living Standard." [Online]. Available: https://dom.spec.whatwg.org/. [Accessed: 24-Sep-2026].

Chapter 12

Evaluation and Discussion III: Recursion at Work on Lists and Trees

This chapter adds no new structure. It puts the doubly linked and circular lists of Chapter 9, the recursion of Chapter 10 and the trees and binary search trees of Chapter 11 to work on sixteen problems, each solved from a blank page, one small step at a time, in C++, Java and Python. Ten of them share one small tree: walking it depth first, printing one level, adding up and tripling its values, mirroring it, checking its balance, deleting a value, finding a node's parent and children, and flattening the whole tree into a doubly linked list without allocating a single node. Six more are new: reversing a doubly linked list, the Josephus circle, choosing books by backtracking, checking that a tree really is a binary search tree, the lowest common ancestor, and printing a tree level by level. For every problem you will draw the tree, trace the recursion by hand, read three complete programs and their real output, watch a tempting wrong solution fail (for memory bugs, in a memory checker's own report), and finish with a *check yourself* variant and its answer.

12.1 Where We Stand

Three chapters lie behind us since the last chapter of this kind, and each gave us one tool.

Chapter 9 gave every node a second link. In a *doubly linked list* each node points both to the node after it (`next`) and to the node before it (`prev`), so the list can be walked in both directions and a node can be removed when all we hold is a pointer to that node (Section 9.2). We kept a `head` and a `tail` (Section 9.4), wrote insertion and deletion at both ends, and met the *circular* list, whose last node points back to the first, so that there is no end at all, only a place where we agreed to start (Section 9.6). Every operation came with its rule of two arrows: whenever a forward arrow changes, the matching backward arrow must change too.

Chapter 10 taught *recursion*: a function that solves a problem by calling itself on a smaller instance of the same problem. We met the two parts that every recursive function has, the *base case*, which is answered directly, and the *recursive case*, which reduces the problem and trusts the smaller call to be right (Section 10.2). We traced factorial, Fibonacci, the greatest common divisor and the Towers of Hanoi, watched the *call stack* grow and shrink (Section 10.3), saw how the naive Fibonacci repeats the same work exponentially often, and wrote recurrences such as $T(n) = 2T(n-1) + 1$ to count the moves of Hanoi (Section 10.8).

Chapter 11 put the two ideas together. A *tree* is a structure of nodes in which every node except one, the *root*, has exactly one *parent*, and a *binary tree* is a tree in which every node has at most two children, a *left* and a *right*. A *binary search tree* (BST) adds one rule about the values: everything in a node's left subtree is smaller than the node, and everything in its right subtree is larger. We wrote `insert`, the three traversals `preOrder`, `inOrder` and `postOrder` (Section 11.6.1), the level-order traversal with a queue (Section 11.6.3), `search`, `count`, `height`, `min`, `max` and `leaf`, nearly all of them recursive (Section 11.7), because a tree is itself a recursive thing: a root with a left tree and a right tree hanging from it.

This chapter uses all three tools, very often in the same function. **Table 12.1** lists the sixteen problems, where each is solved, and which chapter supplies its main idea.

Table 12.1 The sixteen 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
12.1	12.4	Print a depth-first traversal, left branch first	Ch. 10, 11
12.2	12.5	Print every node on level k	Ch. 10, 11
12.3	12.6	Add up all the values in the tree	Ch. 10, 11
12.4	12.7	Multiply every value by three	Ch. 10, 11
12.5	12.8	Turn the tree into its mirror image	Ch. 10, 11
12.6	12.9	Decide whether every node is height-balanced	Ch. 10, 11
12.7	12.10	Delete a value from a binary search tree	Ch. 11
12.8	12.11	Find the parent of the node holding a value	Ch. 11
12.9	12.12	Report the children of the node holding a value	Ch. 11
12.10	12.13	Flatten the tree into a doubly linked list, in place	Ch. 9, 10, 11
12.11	12.14	Reverse a doubly linked list in place	Ch. 9
12.12	12.14	The Josephus circle: who is left?	Ch. 9, 10
12.13	12.15	Every way to choose k of n books	Ch. 10
12.14	12.16	Is this binary tree a binary search tree?	Ch. 10, 11
12.15	12.16	The lowest common ancestor of two values	Ch. 11
12.16	12.16	Print a tree level by level	Ch. 5, 11

Throughout the chapter we keep the running example of this book, the catalog of the small library *Pustaka Ceria*, whose books have identifiers such as `B1001`. The first ten problems share one small tree of integers, the tree you get by inserting 6, 3, 10, 1, 5, 7, 8 and 9 into an empty binary search tree, because the point of those problems is to watch many different functions walk the same picture.

> **THE ONE IDEA BEHIND ALMOST EVERY PROBLEM HERE.** *A tree is empty, or it is a root with a left tree and a right tree; so answer the empty tree directly, ask the same question of the two smaller trees, and combine the two answers with the root.* The sum of a tree is the root plus the sums of its two trees; its mirror is the root with its two trees swapped and mirrored; its height is one more than the taller of its two trees. When a program in this chapter goes wrong, one of the three parts was broken: the empty tree answered wrongly (Problems 12.1 and 12.6), the question changed on the way down (Problem 12.2), or a smaller tree's answer thrown away instead of combined (Problems 12.3 and 12.7).

12.2 How to Approach a Recursive Problem on a Linked Structure

A recursive solution on a tree or a list is a promise: "give me the answers for the smaller pieces, and I will build the answer for the whole". The code is often five lines; the difficulty is trusting that promise, and keeping the pointers right when the function changes the structure. The method has seven steps.

Step 1 — Restate the problem in your own words, with one example. Write down what goes in, what comes out, and one small tree or list with its answer worked out by hand.

Step 2 — Say what the function returns for the empty structure. This is the *base case*, where most recursive functions go wrong. The sum of no values is 0; the height of no tree is -1 (so that a single node has height 0, Section 11.2.3); an empty tree *is* balanced and *is* a binary search tree; printing it prints nothing. Check that the answer makes the combining step of Step 4 right for a single node.

Step 3 — Draw the tree, and mark what each call receives. Next to each node, write the arguments its call receives: the level still to go, the limits, the node before it in a list. A child that receives the wrong question is then visible at a glance.

Step 4 — Write the recursive case as "root, left answer, right answer". Assume that the calls on the two subtrees return the right answers, the *recursive leap of faith* of Section 10.2, justified by induction [1], and write only how the node combines them. Decide *when* the node does its own work: before the calls (*preorder*, as in printing a depth-first order), between them (*inorder*, as in flattening a BST into sorted order) or after them (*postorder*, as in freeing a tree).

Step 5 — If the function changes links, make it return the new root. A function that may delete or replace the node it was given returns what now stands there, and the parent re-attaches it: `root->left = f(root->left)`. Problems 12.7 and 12.10 rely on this, and their wrong solutions break it.

Step 6 — List the edge cases. For trees: the empty tree, one node, a node with only a left or only a right child, an absent value, and a long chain. For lists: empty, one node, first node, last node.

Step 7 — Test, including under a memory checker. Run the example, then every edge case; in C++, run once more with the sanitizers, because a function that reads a freed node can print the right answer by luck.

Pseudocode 12.1 writes Steps 2, 4 and 5 as a template. Nearly every tree function in this chapter is this template with the three blanks filled in.

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
```

How much does such a function cost? Each call on a non-empty tree does a fixed amount of its own work, say c steps, and makes two calls on subtrees; each call on an empty tree does a fixed amount c_0. If the tree has n nodes, there are exactly n calls on non-empty trees, one per node, and exactly $n + 1$ calls on empty trees, because every node has two child pointers, which makes $2n$ pointers, and $n - 1$ of them point to real nodes (every node except the root has one parent pointing at it), so $2n - (n - 1) = n + 1$ of them are `nullptr`. So the total cost is

$$T(n) = c\,n + c_0\,(n + 1) = \Theta(n). \tag{12.1}$$

Equation (12.1) is the reason almost every function in this chapter is linear: it touches each node once and each empty child once. The exceptions are worth noticing when we meet them: `printKthLevel` stops early, `search` and the BST functions walk only one path, and the balance check of Problem 12.6 calls `height` again and again.

Table 12.2 condenses the method onto one page. Every problem below applies it, and the labels in bold inside each problem follow its order.

Table 12.2 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?

How the programs of this chapter were checked. Every program below is complete and was run. The C++ programs compile with GCC in C++17 mode and all the usual warnings switched on, without a single warning; each was compiled again with AddressSanitizer and UndefinedBehaviorSanitizer [2], [3], which watch every memory access while the program runs, and run under Valgrind [4]; every run was clean, with no invalid access and no leaked byte. The commands are these (on Windows, run `p1201` instead of `./p1201`):

```
g++ -std=c++17 -O2 -Wall -Wextra -o p1201 p1201_depth_first.cpp      (normal build)
./p1201
g++ -std=c++17 -g -Wall -Wextra -fsanitize=address,undefined \
    -o p1201_asan p1201_depth_first.cpp
./p1201_asan                  (same output, plus a report if memory is misused)
valgrind --leak-check=full ./p1201          (Linux: the same check, no recompile)
```

The Java programs were compiled with `javac *.java` in their folder and the Python programs run with `python3`; they target JDK 25 and Python 3.14 but need only JDK 21 and Python 3.10. For every problem the three programs print exactly the same lines, checked byte for byte. The chapter's code folder and its `README.md` hold every program and every wrong solution shown below.

One tree, three languages. **Table 12.3** is a small dictionary for reading the three versions of each program side by side. It extends the table of the same kind in Chapter 8 with the rows that trees need.

Table 12.3 The tree ideas of this chapter in the three languages of the book. Only C++ lets a function receive the address of the caller's pointer; Java and Python return the new root instead, or keep the variables to be changed inside an object.

Idea	C++	Java	Python
A tree node	`struct Node { int data; Node *left; Node *right; };`	`class Node { int data; Node left, right; }`	`class Node:` with `data`, `left`, `right`
The empty tree	`nullptr`	`null`	`None`
Insert, changing the caller's root	`insert(&root, 6)` with `Node **root`	`root = insert(root, 6)`	`root = insert(root, 6)`
Free a whole tree	`clear(&root)`: children first, then the node	`root = null` (garbage collector)	`root = None` (automatic)
A helper that must change two of the caller's variables	pass `Node **head, Node **prev`	put them in one small object	put them in one small object
Share the toolkit	`#include "tree_kit.hpp"`	`class P1201... extends TreeKit`	`from tree_kit import *`
Library stack / queue	`std::stack`, `std::queue`	`ArrayDeque`	`list`, `collections.deque`

Two rows deserve a sentence each. Only C++ makes *you* free the nodes, so only the C++ programs can leak a subtree or read a freed node, and only they are run under the memory checkers; every one of them ends by calling `clear`. And only C++ can pass the address of the caller's pointer, a `Node **` (Chapter 6); Java and Python *return* the new root instead, as their versions in Chapter 8 did [5], [6], [7].

12.3 The Tree We Work On

12.3.1 The starting point

The first ten problems share one starting point, which we state first in its original form.

> **PROBLEM.** Starting from a binary search tree program that already has `insert`, `clear`, the traversals `preOrder`, `inOrder` and `postOrder`, `search`, `count` and `height`, write the functions of Problems 12.1 to 12.10. As a case study, use the tree built by this driver: `clear(&root); insert(&root, 6); insert(&root, 3); insert(&root, 10); insert(&root, 1); insert(&root, 5); insert(&root, 7); insert(&root, 8); insert(&root, 9);`

The driver builds the tree of **Figure 12.1**. Build it by hand once, because every problem below walks it. The first value, 6, goes into the empty tree and becomes the root. Then 3 is smaller than 6, so it goes to the left of 6. 10 is larger than 6, so it goes to the right. 1 is smaller than 6 (go left, to 3) and smaller than 3 (go left again), and the left of 3 is empty, so 1 settles there. 5 is smaller than 6 (to 3) but larger than 3, so it becomes the right child of 3. 7 is larger than 6 (to 10) and smaller than 10, so it becomes the left child of 10. 8 goes right at 6, left at 10, right at 7. And 9 goes right at 6, left at 10, right at 7 and right at 8. Eight values, eight nodes.

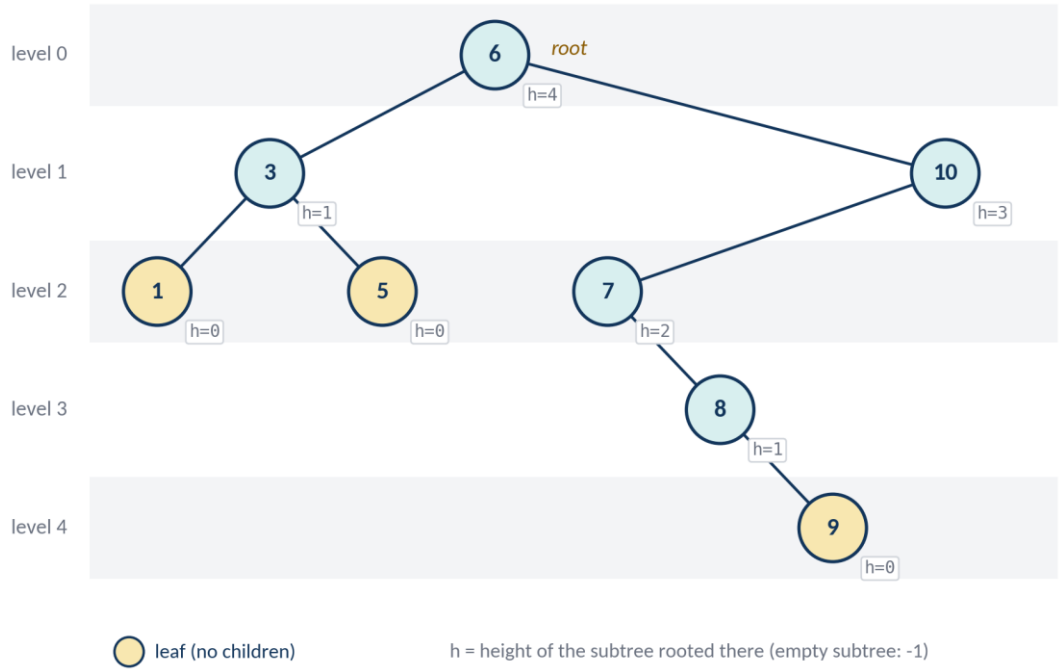


Figure 12.1 The tree of this chapter, built by inserting 6, 3, 10, 1, 5, 7, 8, 9 into an empty binary search tree. The levels are counted from 0 at the root, the leaves are shaded, and h is the height of the subtree rooted at each node.

Read three words off the figure, because the problems use them constantly. The **level** (or *depth*) of a node is the number of edges from the root down to it: 6 is on level 0, 3 and 10 on level 1, 1, 5 and 7 on level 2, 8 on level 3 and 9 on level 4. The **height** of a node is the number of edges on the longest path from it down to a leaf; the tree's height is the root's height, 4 here. By the convention of Section 11.2.3 the empty tree has height -1, which makes the rule for every node the same:

$$h(\varnothing) = -1, \qquad h(T) = 1 + \max(h(T.left),\, h(T.right)). \tag{12.2}$$

Check Equation (12.2) on a leaf: both children are empty, so its height is $1 + \max(-1, -1) = 0$, as it should be. On node 8: its left is empty (-1) and its right is the leaf 9 (0), so its height is $1 + 0 = 1$. The **leaves** are the nodes with no children at all: 1, 5 and 9.

The tree is a binary search tree, and the rule that makes it one is worth writing precisely, because Problems 12.4, 12.5 and 12.14 all turn on it:

$$\text{for every node } x\text{:}\quad y < x.key \text{ for every } y \text{ in } x.left, \qquad y > x.key \text{ for every } y \text{ in } x.right. \tag{12.3}$$

Note the words *every* in Equation (12.3). The rule is not only about a node and its two children; it is about a node and *all* of its descendants. Node 5 is in the left subtree of 6, so it must be smaller than 6, even though it is the *right* child of its own parent 3. A consequence we will use again and again is that an inorder traversal (left, node, right) of a binary search tree prints the values in increasing order [1], [8].

How tall can a binary tree with n nodes be, and how short? Section 11.5.3 answered this, and we recall the argument. Level k can hold at most 2^k nodes, because each node on level $k-1$ has at most two children. So a tree of height h holds at most $1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$ nodes, and turning that around,

$$n \le 2^{h+1} - 1 \quad \Rightarrow \quad h \ge \lceil \lg(n+1) \rceil - 1, \qquad \text{and always} \quad h \le n - 1. \tag{12.4}$$

Here lg is the logarithm to base 2 and $\lceil x \rceil$ rounds x up to a whole number. For our $n = 8$, Equation (12.4) says the height is at least $\lceil \lg 9 \rceil - 1 = 4 - 1 = 3$ and at most 7. Ours is 4, near the short end. A tree built by inserting 1, 2, 3, …, 8 in increasing order would be a single chain leaning to the right, with height 7, the tall end. That chain is the worst case for every function in this chapter that walks one path, and we will use it as an edge case.

12.3.2 The toolkit, in three languages

Rather than repeat the insertion, traversal, search, count and height functions in every program, this chapter keeps them in one small file per language, the *toolkit*, and every tree program uses it. **Listing 12.1** is the C++ toolkit. The functions are those of Section 11.7, with three changes: `clear` really frees the nodes (a version that only sets the root to `nullptr` would leak every node), `search` returns `nullptr` silently instead of printing, and two helpers build trees quickly: `insertAll` inserts a list of values, and `buildSample` builds the tree of **Figure 12.1**. The last function, `showSideways`, prints a tree as text, turned a quarter turn to the left, so that a program can show us the shape of a tree it has changed.

Listing 12.1 `tree_kit.hpp`: the C++ toolkit shared by the tree programs of this chapter.

```
// tree_kit.hpp - the binary search tree toolkit shared by Problems 12.1-12.10.
// Every tree program of Chapter 12 starts with #include "tree_kit.hpp".
#ifndef TREE_KIT_HPP
#define TREE_KIT_HPP

#include <cstdio>
#include <initializer_list>

struct Node {
    int data;
    Node *left;     // the subtree of smaller values
    Node *right;    // the subtree of larger values
};

// Insert newData into the BST whose root pointer lives at *root.
// A value that is already in the tree is ignored.
inline void insert(Node **root, int newData) {
    if (*root == nullptr) {
        *root = new Node{newData, nullptr, nullptr};
    } else if (newData < (*root)->data) {
        insert(&(*root)->left, newData);
    } else if (newData > (*root)->data) {
        insert(&(*root)->right, newData);
    }
}

// Free every node (both subtrees first, then the node) and empty the tree.
inline void clear(Node **root) {
    if (*root == nullptr) return;
    clear(&(*root)->left);
    clear(&(*root)->right);
    delete *root;
    *root = nullptr;
}

inline void preOrder(Node *root) {      // node, left, right
    if (root == nullptr) return;
    printf("%d ", root->data);
    preOrder(root->left);
    preOrder(root->right);
}
```

Listing 12.1 (Continued) *tree_kit.hpp: the C++ toolkit shared by the tree programs of this chapter.*

```
inline void inOrder(Node *root) {       // left, node, right
    if (root == nullptr) return;
    inOrder(root->left);
    printf("%d ", root->data);
    inOrder(root->right);
}

inline void postOrder(Node *root) {     // left, right, node
    if (root == nullptr) return;
    postOrder(root->left);
    postOrder(root->right);
    printf("%d ", root->data);
}

// The node that holds data, or nullptr if there is none.
inline Node *search(Node *root, int data) {
    if (root == nullptr) return nullptr;
    if (data < root->data) return search(root->left, data);
    if (data > root->data) return search(root->right, data);
    return root;
}

inline int count(Node *root) {
    if (root == nullptr) return 0;
    return count(root->left) + count(root->right) + 1;
}

// Height in edges: -1 for the empty tree, 0 for a single node.
inline int height(Node *root) {
    if (root == nullptr) return -1;
    int u = height(root->left);
    int v = height(root->right);
    return (u > v ? u : v) + 1;
}

// Insert several values, from left to right.
inline void insertAll(Node **root, std::initializer_list<int> values) {
    for (int v : values) insert(root, v);
}

// The tree used throughout the chapter: 6, 3, 10, 1, 5, 7, 8, 9 in this order.
inline void buildSample(Node **root) {
    insertAll(root, {6, 3, 10, 1, 5, 7, 8, 9});
}
```

Listing 12.1 (Continued) *tree_kit.hpp: the C++ toolkit shared by the tree programs of this chapter.*

```
// Print the tree turned a quarter turn to the left: the root at the margin,
// its right subtree above it and its left subtree below it.
inline void showSideways(Node *root, int depth = 0) {
    if (root == nullptr) return;
    showSideways(root->right, depth + 1);
    printf("%*s%d\n", 4 * depth, "", root->data);
    showSideways(root->left, depth + 1);
}

#endif
```

Three details of the toolkit deserve a word before we rely on it. `insert` **receives** `Node **root`, the address of the caller's pointer, because inserting into an empty tree must change the caller's `root` from `nullptr` to the new node; the call `insert(&(*root)->left, newData)` passes the address of the `left` field inside the current node, so the recursive call can fill that field when it reaches the

bottom. Chapter 6 explained this double pointer, and Problem 12.10 will need it again. **`clear` is a postorder function**: it frees both subtrees before the node itself, because once a node is freed its `left` and `right` fields may no longer be read. **`showSideways` is the "right, node, left" traversal of Section 11.7.6** that indents each value by four spaces per level. Printing the right subtree first puts it at the top, so if you turn the page a quarter turn clockwise, the printout looks like **Figure 12.1**.

Listing 12.2 and **Listing 12.3** are the same toolkit in Java and Python. Both return the new root from `insert`, as **Table 12.3** explained. In Java every tree program of this chapter is declared `extends TreeKit`, which lets it use `Node`, `insert` and the others by their short names; in Python every program starts with `from tree_kit import *`.

Listing 12.2 `TreeKit.java`: the Java toolkit.

```
// TreeKit.java - the binary search tree toolkit shared by Problems 12.1-12.10.
// Every tree program of Chapter 12 is declared as "extends TreeKit" to use it.
public class TreeKit {

    static class Node {
        int data;
        Node left;      // the subtree of smaller values
        Node right;     // the subtree of larger values

        Node(int data, Node left, Node right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }

    // Insert newData and return the root (a new root if the tree was empty).
    // A value that is already in the tree is ignored.
    static Node insert(Node root, int newData) {
        if (root == null) {
            return new Node(newData, null, null);
        }
        if (newData < root.data) {
            root.left = insert(root.left, newData);
        } else if (newData > root.data) {
            root.right = insert(root.right, newData);
        }
        return root;
    }

    // Insert several values, from left to right.
    static Node insertAll(Node root, int... values) {
        for (int v : values) root = insert(root, v);
        return root;
    }

    // The tree used throughout the chapter: 6, 3, 10, 1, 5, 7, 8, 9 in this order.
    static Node buildSample() {
        return insertAll(null, 6, 3, 10, 1, 5, 7, 8, 9);
    }
```

Listing 12.2 (Continued) *TreeKit.java: the Java toolkit.*

```
    static void preOrder(Node root) {      // node, left, right
        if (root == null) return;
        System.out.print(root.data + " ");
        preOrder(root.left);
        preOrder(root.right);
    }

    static void inOrder(Node root) {       // left, node, right
        if (root == null) return;
        inOrder(root.left);
        System.out.print(root.data + " ");
        inOrder(root.right);
    }

    static void postOrder(Node root) {     // left, right, node
        if (root == null) return;
        postOrder(root.left);
        postOrder(root.right);
        System.out.print(root.data + " ");
    }

    // The node that holds data, or null if there is none.
    static Node search(Node root, int data) {
        if (root == null) return null;
        if (data < root.data) return search(root.left, data);
        if (data > root.data) return search(root.right, data);
        return root;
    }

    static int count(Node root) {
        if (root == null) return 0;
        return count(root.left) + count(root.right) + 1;
    }

    // Height in edges: -1 for the empty tree, 0 for a single node.
    static int height(Node root) {
        if (root == null) return -1;
        return Math.max(height(root.left), height(root.right)) + 1;
    }
```

Listing 12.2 (Continued) *TreeKit.java: the Java toolkit.*

```
    // Print the tree turned a quarter turn to the left: the root at the margin,
    // its right subtree above it and its left subtree below it.
    static void showSideways(Node root, int depth) {
        if (root == null) return;
        showSideways(root.right, depth + 1);
        System.out.println(" ".repeat(4 * depth) + root.data);
        showSideways(root.left, depth + 1);
    }

    static void showSideways(Node root) {
        showSideways(root, 0);
    }
}
```

Listing 12.3 `tree_kit.py`: the Python toolkit.

```
# tree_kit.py - the binary search tree toolkit shared by Problems 12.1-12.10.
# Every tree program of Chapter 12 starts with "from tree_kit import *".


class Node:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left        # the subtree of smaller values
        self.right = right      # the subtree of larger values


def insert(root, new_data):
    """Insert new_data and return the root (a new root if the tree was empty).
    A value that is already in the tree is ignored."""
    if root is None:
        return Node(new_data)
    if new_data < root.data:
        root.left = insert(root.left, new_data)
    elif new_data > root.data:
        root.right = insert(root.right, new_data)
    return root


def insert_all(root, values):
    """Insert several values, from left to right."""
    for v in values:
        root = insert(root, v)
    return root


def build_sample():
    """The tree used throughout the chapter: 6, 3, 10, 1, 5, 7, 8, 9, in order."""
    return insert_all(None, [6, 3, 10, 1, 5, 7, 8, 9])


def pre_order(root):            # node, left, right
    if root is None:
        return
    print(root.data, end=" ")
    pre_order(root.left)
    pre_order(root.right)
```

Listing 12.3 (Continued) *tree_kit.py: the Python toolkit.*

```
def in_order(root):             # left, node, right
    if root is None:
        return
    in_order(root.left)
    print(root.data, end=" ")
    in_order(root.right)


def post_order(root):           # left, right, node
    if root is None:
        return
    post_order(root.left)
    post_order(root.right)
    print(root.data, end=" ")


def search(root, data):
    """The node that holds data, or None if there is none."""
    if root is None:
        return None
    if data < root.data:
        return search(root.left, data)
    if data > root.data:
        return search(root.right, data)
    return root


def count(root):
    if root is None:
        return 0
    return count(root.left) + count(root.right) + 1


def height(root):
    """Height in edges: -1 for the empty tree, 0 for a single node."""
    if root is None:
        return -1
    return max(height(root.left), height(root.right)) + 1
```

Listing 12.3 (Continued) *tree_kit.py: the Python toolkit.*

```
def show_sideways(root, depth=0):
    """Print the tree turned a quarter turn to the left: the root at the margin,
    its right subtree above it and its left subtree below it."""
    if root is None:
        return
    show_sideways(root.right, depth + 1)
    print("    " * (4 * depth) + str(root.data))
    show_sideways(root.left, depth + 1)
```

12.3.3 Looking at the tree

Listing 12.4 builds the tree and looks at it in every way the toolkit allows. Its output is our reference for the whole chapter.

Listing 12.4 `p1200_sample_tree.cpp`: the tree of this chapter, built and inspected.

```cpp
// p1200_sample_tree.cpp - the tree of Chapter 12, built and inspected.
#include "tree_kit.hpp"

int main() {
    Node *root = nullptr;
    buildSample(&root);                       // 6, 3, 10, 1, 5, 7, 8, 9
    printf("The tree, turned on its side (root at the left):\n");
    showSideways(root);
    printf("preorder : ");
    preOrder(root);
    printf("\ninorder  : ");
    inOrder(root);
    printf("\npostorder: ");
    postOrder(root);
    printf("\nnodes %d, height %d\n", count(root), height(root));
    printf("search 8 : %s\n", search(root, 8) != nullptr ? "found" : "not found");
    printf("search 18: %s\n", search(root, 18) != nullptr ? "found" : "not found");
    clear(&root);
    printf("after clear: nodes %d, height %d\n", count(root), height(root));
    return 0;
}
```

```
The tree, turned on its side (root at the left):
    10
                9
            8
        7
6
        5
    3
        1
preorder : 6 3 1 5 10 7 8 9
inorder  : 1 3 5 6 7 8 9 10
postorder: 1 5 3 9 8 7 10 6
nodes 8, height 4
search 8 : found
search 18: not found
after clear: nodes 0, height -1
```

Turn the page a quarter turn clockwise and the first eight lines become **Figure 12.1**: 6 at the root, 10 above it on the right, 7, 8 and 9 hanging below 10, and 3 with its children 1 and 5 on the left. The preorder line visits each node before its subtrees; the inorder line is sorted, as Equation (12.3) promises; the postorder line visits each node after its subtrees, so the root 6 comes last. The count is 8 and the height 4, both as computed by hand. After `clear`, both functions see an empty tree and give 0 and -1, and Valgrind confirms that `clear` freed all eight nodes. **Listing 12.5** and **Listing 12.6** print exactly the same lines.

Listing 12.5 `P1200SampleTree.java`: the tree inspected in Java.

```java
// P1200SampleTree.java - the tree of Chapter 12, built and inspected.
public class P1200SampleTree extends TreeKit {

    static String found(Node root, int x) {
        return search(root, x) != null ? "found" : "not found";
    }

    public static void main(String[] args) {
        Node root = buildSample();              // 6, 3, 10, 1, 5, 7, 8, 9
        System.out.println("The tree, turned on its side (root at the left):");
        showSideways(root);
        System.out.print("preorder : ");
        preOrder(root);
        System.out.print("\ninorder  : ");
        inOrder(root);
        System.out.print("\npostorder: ");
        postOrder(root);
        System.out.printf("%nnodes %d, height %d%n", count(root), height(root));
        System.out.println("search 8 : " + found(root, 8));
        System.out.println("search 18: " + found(root, 18));
        root = null;  // the garbage collector frees the nodes
        System.out.printf("after clear: nodes %d, height %d%n",
                count(root), height(root));
    }
}
```

Listing 12.6 `p1200_sample_tree.py`: the tree inspected in Python.

```python
# p1200_sample_tree.py - the tree of Chapter 12, built and inspected.
from tree_kit import *


def main():
    root = build_sample()                    # 6, 3, 10, 1, 5, 7, 8, 9
    print("The tree, turned on its side (root at the left):")
    show_sideways(root)
    print("preorder : ", end="")
    pre_order(root)
    print("\ninorder  : ", end="")
    in_order(root)
    print("\npostorder: ", end="")
    post_order(root)
    print(f"\nnodes {count(root)}, height {height(root)}")
    print("search 8 :", "found" if search(root, 8) is not None else "not found")
    print("search 18:", "found" if search(root, 18) is not None else "not found")
    root = None                              # Python frees the nodes by itself
    print(f"after clear: nodes {count(root)}, height {height(root)}")


if __name__ == "__main__":
    main()
```

12.4 Problem 12.1 — Depth-First Traversal, Left Branch First

12.4.1 The problem

> **PROBLEM.** Write a recursive function to print the depth-first traversal of a tree, i.e., choosing left branches first. `void printDepthFirst(Node *root)`

12.4.2 In our own words

Depth first means: once you step down into a branch, finish that whole branch before you look anywhere else. *Left branches first* means: at every node, the left branch is the one you go into first.

And to *print* the traversal is to print each node at the moment you first arrive at it. Put together: print the node, then print everything in its left subtree, then everything in its right subtree. That is exactly the *preorder* of Section 11.6.1, and for our tree the expected output is `6 3 1 5 10 7 8 9`.

The base case (Step 2 of the method) is the empty tree, and printing an empty tree prints nothing. The recursive case (Step 4) is "node, left, right" with the node's own work, the `printf`, done *before* the two calls.

12.4.3 By hand, step by step

Figure 12.2 numbers the nodes of the tree in the order they are printed. Follow the recursion, remembering that each call finishes its left subtree completely before it starts its right one.

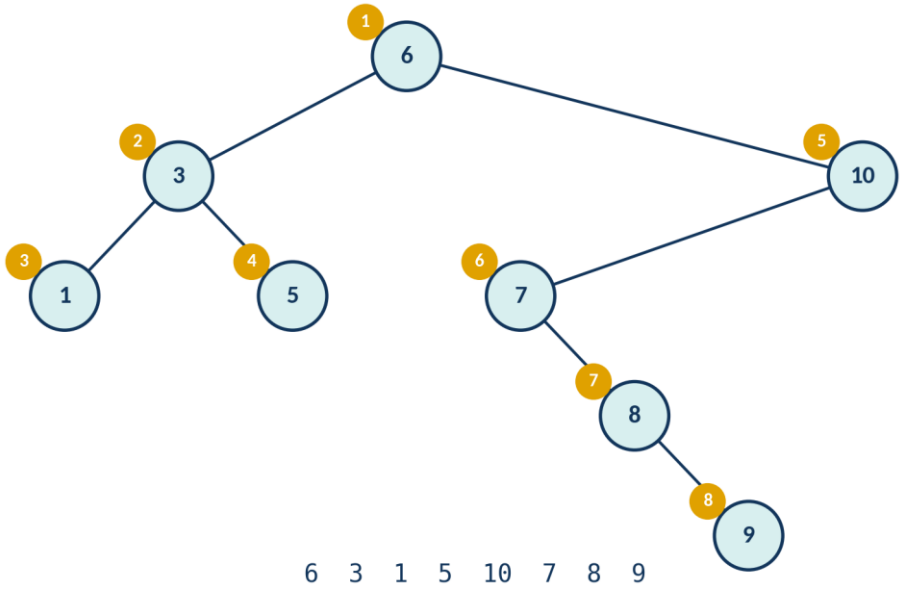


Figure 12.2 The depth-first order of Problem 12.1: each badge gives the moment the node is printed. The whole left subtree of 6 (badges 2 to 4) is finished before the right subtree (badges 5 to 8) is entered.

1. The call on 6 prints **6**, then calls itself on the left child 3.
2. The call on 3 prints **3**, then calls itself on 1.
3. The call on 1 prints **1**. Its left child is `nullptr`, so that call returns at once; so does the call on its right child. The call on 1 is finished and returns to the call on 3.
4. The call on 3 now calls itself on its right child 5, which prints **5** and, having two empty children, returns. The call on 3 is finished; the whole left subtree of 6 has been printed.
5. The call on 6 calls itself on 10, which prints **10** and goes left to 7, which prints **7**. The left of 7 is empty; its right is 8, which prints **8**, and the right of 8 is 9, which prints **9**.

The printed sequence is `6 3 1 5 10 7 8 9`. How deep does the call stack get? At the moment 9 is printed, the calls on 6, 10, 7, 8 and 9 are all waiting on the stack, five of them: one more than the height. In general a traversal needs stack space proportional to $h + 1$, which is small for a short tree and as large as n for a chain.

12.4.4 The C++ solution

Listing 12.7 gives the recursive function and, next to it, a second version without recursion, to show where the recursion's hidden work goes. Every recursive call is, underneath, a push onto the call stack; the iterative version pushes the nodes still to be visited onto a `std::stack` of its

own (Chapter 5), the technique of Sections 10.13 and 11.6.4. It pushes the right child *before* the left one, so that the left child comes off the stack first, which is what "left branches first" demands.

Listing 12.7 `p1201_depth_first.cpp`: Problem 12.1, recursively and with an explicit stack.

```
// p1201_depth_first.cpp - Problem 12.1: depth-first traversal, left branch first.
#include <stack>
#include "tree_kit.hpp"

// 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
}

// The same order without recursion: a stack holds the nodes still to visit.
void printDepthFirstStack(Node *root) {
    std::stack<Node *> todo;
    if (root != nullptr) todo.push(root);
    while (!todo.empty()) {
        Node *cur = todo.top();
        todo.pop();
        printf("%d ", cur->data);
        if (cur->right != nullptr) todo.push(cur->right);   // pushed first,
        if (cur->left != nullptr) todo.push(cur->left);     // so visited second
    }
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    printf("depth-first, recursive : ");
    printDepthFirst(root);
    printf("\ndepth-first, with stack: ");
    printDepthFirstStack(root);
    Node *empty = nullptr;
    printf("\nempty tree             : [");
    printDepthFirst(empty);
    printf("]\n");
    clear(&root);
    return 0;
}
```

```
depth-first, recursive : 6 3 1 5 10 7 8 9
depth-first, with stack: 6 3 1 5 10 7 8 9
empty tree             : []
```

Line by line. Both versions print the expected `6 3 1 5 10 7 8 9`. The third line calls the function on an empty tree; the brackets around its output show that it prints nothing and, more importantly, does not crash. The recursive version has three lines of real work: the base case, the print, and two calls. **Cost.** By Equation (12.1), $\Theta(n)$ time: one print per node and a constant amount of work per empty child. The extra memory is the stack: $O(h)$ frames for the recursive version, and at most $O(h)$ nodes waiting on the explicit stack for the iterative one (each level contributes at most one waiting right child).

12.4.5 The Java and Python solutions

Listing 12.8 and **Listing 12.9** print exactly the same three lines. Java uses `ArrayDeque` as the stack, whose `push` and `pop` work at the front [9]; Python uses a plain list, whose `append` and `pop` work at the end [10].

Listing 12.8 `P1201DepthFirst.java`: Problem 12.1 in Java.

```java
// P1201DepthFirst.java - Problem 12.1: depth-first traversal, left branch first.
import java.util.ArrayDeque;

public class P1201DepthFirst extends TreeKit {

    // Visit the node, then everything on its left, then everything on its right.
    static void printDepthFirst(Node root) {
        if (root == null) return;  // base case: an empty tree prints nothing
        System.out.print(root.data + " ");    // 1. the node itself
        printDepthFirst(root.left);           // 2. the whole left subtree
        printDepthFirst(root.right);          // 3. the whole right subtree
    }

    // The same order without recursion: a stack holds the nodes still to visit.
    static void printDepthFirstStack(Node root) {
        ArrayDeque<Node> todo = new ArrayDeque<>();
        if (root != null) todo.push(root);
        while (!todo.isEmpty()) {
            Node cur = todo.pop();
            System.out.print(cur.data + " ");
            if (cur.right != null) todo.push(cur.right);   // pushed first,
            if (cur.left != null) todo.push(cur.left);     // so visited second
        }
    }

    public static void main(String[] args) {
        Node root = buildSample();
        System.out.print("depth-first, recursive : ");
        printDepthFirst(root);
        System.out.print("\ndepth-first, with stack: ");
        printDepthFirstStack(root);
        System.out.print("\nempty tree             : [");
        printDepthFirst(null);
        System.out.println("]");
    }
}
```

Listing 12.9 `p1201_depth_first.py`: Problem 12.1 in Python.

```python
# p1201_depth_first.py - Problem 12.1: depth-first traversal, left branch first.
from tree_kit import *


def print_depth_first(root):
    """Visit the node, then everything on its left, then on its right."""
    if root is None:                    # base case: an empty tree prints nothing
        return
    print(root.data, end=" ")           # 1. the node itself
    print_depth_first(root.left)        # 2. the whole left subtree
    print_depth_first(root.right)       # 3. the whole right subtree


def print_depth_first_stack(root):
    """The same order without recursion: a stack holds the nodes still to visit."""
    todo = [root] if root is not None else []
    while todo:
        cur = todo.pop()
        print(cur.data, end=" ")
        if cur.right is not None:
            todo.append(cur.right)      # pushed first,
        if cur.left is not None:
            todo.append(cur.left)       # so visited second


def main():
    root = build_sample()
    print("depth-first, recursive : ", end="")
    print_depth_first(root)
    print("\ndepth-first, with stack: ", end="")
    print_depth_first_stack(root)
    print("\nempty tree             : [", end="")
    print_depth_first(None)
    print("]")


if __name__ == "__main__":
    main()
```

12.4.6 A tempting wrong solution

The tempting shortcut is to leave out the base case, on the thought that "the root is never empty":

```
  void printDepthFirst(Node *root) {
      printf("%d ", root->data);          // WRONG: no base case at all
      printDepthFirst(root->left);
      printDepthFirst(root->right);
  }
```

The compiler already smells trouble and warns `infinite recursion detected [-Winfinite-recursion]`: a function that calls itself unconditionally can never stop by itself. What actually stops it is worse. Running the program `wrong/p1201_wrong.cpp` under the sanitizers prints this (the long hexadecimal addresses differ from run to run):

```
depth-first, recursive : 6 3 1 p1201_wrong.cpp:7:11: runtime error: member access within
null pointer of type 'struct Node'
==27788==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x55b82ce5f5d4
bp 0x7ffe08c512b0 sp 0x7ffe08c512a0 T0)
==27788==The signal is caused by a READ memory access.
==27788==Hint: address points to the zero page.
    #0 0x55b82ce5f5d4 in printDepthFirst(Node*) p1201_wrong.cpp:7
    #1 0x55b82ce5f66c in printDepthFirst(Node*) p1201_wrong.cpp:8
SUMMARY: AddressSanitizer: SEGV p1201_wrong.cpp:7 in printDepthFirst(Node*)
```

The walk prints 6, 3 and 1, and then the call on 1 calls itself on 1's left child, which is `nullptr`, and line 7 reads `root->data` through it: the program reads address 0, which no program may read, and dies. Read the report from the top: *what* happened (a read through a null pointer), *where* (line 7 of `printDepthFirst`), and *how we got there* (line 8, the call on the left child). Section 10.3.4 showed the same failure for a function whose base case is missing. The lesson is Step 2 of the method: every recursive function on a tree must say what happens at `nullptr`, because every tree, however large, ends in `nullptr` children, $n + 1$ of them.

12.4.7 Check yourself

> **CHECK YOURSELF.** Change one line of `printDepthFirst` so that it prints the *right* branch first, and give its output on our tree. Then say what the iterative version must change.
>
> *Answer.* Swap the two recursive calls: `printDepthFirst(root->right);` `printDepthFirst(root->left);`. The output becomes `6 10 7 8 9 3 5 1`. In the iterative version, swap the two `push` lines, so that the left child is pushed first and the right child comes off the stack first. (This output is also the preorder of the mirror image of the tree, which Problem 12.5 will produce.)

12.5 Problem 12.2 — Printing One Level

12.5.1 The problem

> **PROBLEM.** Write a recursive function to print all nodes at the k-th level in the binary tree. `void printKthLevel(Node *root, int k)`

12.5.2 In our own words

The root is on level 0, its children on level 1, their children on level 2, and so on (Section 12.3.1). We must print, from left to right, every node whose level is exactly k. Looking at **Figure 12.1**: level 0 is `6`, level 1 is `3 10`, level 2 is `1 5 7`, level 3 is `8`, level 4 is `9`, and level 5 and deeper are empty.

The trick that makes this recursive is to change the question on the way down. "Print the nodes k levels below *me*" is, for each of my children, "print the nodes $k - 1$ levels below *you*". When the number reaches 0, the node that receives it is exactly on the wanted level and prints itself. When a call falls off the tree (it receives `nullptr`) before the number reaches 0, that path is too short, and it prints nothing.

12.5.3 By hand, step by step

Figure 12.3 writes, next to each node, the value of k its call receives when we ask for level 2.

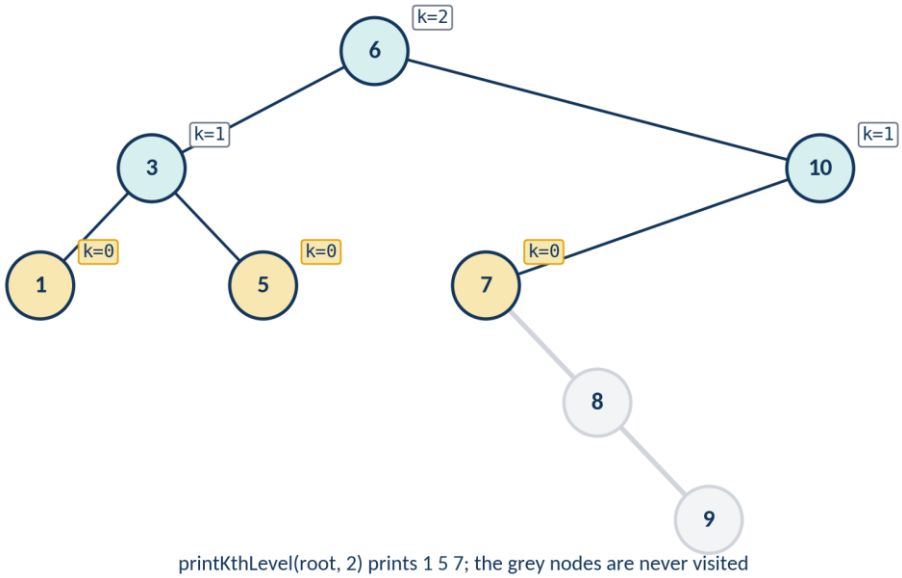


Figure 12.3 The call `printKthLevel(root, 2)`: each node's tag is the k its call receives. The nodes that receive $k = 0$ print themselves and stop, so 8 and 9, below them, are never visited.

1. The call on 6 receives $k = 2$. It is not 0, so it calls itself on 3 and on 10 with $k = 1$.
2. The call on 3 receives $k = 1$ and calls itself on 1 and on 5 with $k = 0$. Both print themselves: **1**, **5**.
3. The call on 10 receives $k = 1$ and calls itself on 7 with $k = 0$, which prints **7**, and on its right child, `nullptr`, which returns at once.
4. The calls with $k = 0$ do not go further down, so 8 and 9 are never touched.

The output is `1 5 7`, in left-to-right order, because every call handles its left child before its right child.

12.5.4 The C++ solution

Listing 12.10 `p1202_kth_level.cpp`: Problem 12.2, printing every level from 0 to one past the height.

```cpp
// p1202_kth_level.cpp - Problem 12.2: print every node on level k (root: level 0).
#include "tree_kit.hpp"

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);
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    for (int k = 0; k <= height(root) + 1; k++) {
        printf("level %d: ", k);
        printKthLevel(root, k);
        printf("\n");
    }
    clear(&root);
    return 0;
}
```

```
level 0: 6
level 1: 3 10
level 2: 1 5 7
level 3: 8
level 4: 9
level 5:
```

Line by line. The loop in `main` asks for every level from 0 to `height(root) + 1`, one more than exists, to test the case of a level below the deepest leaf: it prints an empty line, as it should. Every other line agrees with the hand computation. Notice that the function tests `root == nullptr` *before* it tests `k == 0`: the other order would read `root->data` of a null pointer whenever a path is shorter than k.

Cost. A call with a given k visits only the nodes on levels 0 to k, so it costs O(number of nodes on levels $0..k$), at most $\Theta(n)$. Printing *all* levels this way, as `main` does, costs the sum over k, which for a chain of n nodes is $1 + 2 + \cdots + n = \Theta(n^2)$. Problem 12.16 prints all levels in $\Theta(n)$ with a queue.

12.5.5 The Java and Python solutions

Listing 12.11 and **Listing 12.12** are the Java and Python versions; both print exactly the same lines.

Listing 12.11 `P1202KthLevel.java`: Problem 12.2 in Java.

```java
// P1202KthLevel.java - Problem 12.2: print every node on level k (root: level 0).
public class P1202KthLevel extends TreeKit {

    static void printKthLevel(Node root, int k) {
        if (root == null) return;  // fell off the tree: nothing on this path
        if (k == 0) {                          // this node is exactly k levels down
            System.out.print(root.data + " ");
            return;
        }
        printKthLevel(root.left, k - 1);  // one level lower, one step fewer to go
        printKthLevel(root.right, k - 1);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        for (int k = 0; k <= height(root) + 1; k++) {
            System.out.print("level " + k + ": ");
            printKthLevel(root, k);
            System.out.println();
        }
    }
}
```

Listing 12.12 `p1202_kth_level.py`: Problem 12.2 in Python.

```python
# p1202_kth_level.py - Problem 12.2: print every node on level k (root: level 0).
from tree_kit import *


def print_kth_level(root, k):
    if root is None:                    # fell off the tree: nothing on this path
        return
    if k == 0:                          # this node is exactly k levels down
        print(root.data, end=" ")
        return
    print_kth_level(root.left, k - 1)    # one level lower, one step fewer to go
    print_kth_level(root.right, k - 1)


def main():
    root = build_sample()
    for k in range(height(root) + 2):
        print(f"level {k}: ", end="")
        print_kth_level(root, k)
        print()


if __name__ == "__main__":
    main()
```

Both print the same six lines as **Listing 12.10**. In Python, `range(height(root) + 2)` gives $0, 1, \ldots, h + 1$, the same levels as the C++ loop.

12.5.6 A tempting wrong solution

"k must go down by one on each level", thinks the programmer, and writes the decrement into the calls:

```
    printKthLevel(root->left, --k);      // WRONG: --k changes k itself,
    printKthLevel(root->right, --k);     // so the right side gets k - 2
```

It compiles without a warning, and `wrong/p1202_wrong.cpp` prints this:

```
level 0: 6
level 1: 3
level 2: 1 10
level 3: 5 7
level 4:
level 5: 8
```

`--k` is not "k minus one"; it is "*make* k one smaller, then use it". The first call receives $k-1$, as intended, but it has also changed the local variable k, so the second call receives $k-2$. Every right child is therefore asked for a level one too shallow, and the right side of the tree appears one line early: 10, on level 1, is printed as part of level 2. The correct expression, `k - 1`, computes a new value and leaves `k` untouched, so both children receive the same question. This is Step 3 of the method: write the arguments next to each node, and the two children of 6 visibly receive different numbers.

12.5.7 Check yourself

CHECK YOURSELF. Write `int countKthLevel(Node *root, int k)` that *returns* the number of nodes on level k instead of printing them. What does it return for $k = 0,1,2,3,4,5$ on our tree, and what do the six numbers add up to?

Answer. `if (root == nullptr) return 0; if (k == 0) return 1; return countKthLevel(root->left, k - 1) + countKthLevel(root->right, k - 1);`. It returns 1, 2, 3, 1, 1, 0, which add up to 8, the number of nodes: every node is on exactly one level.

12.6 Problem 12.3 — The Sum of All Nodes

12.6.1 The problem

PROBLEM. Write a recursive function to calculate the sum of all nodes in the binary tree. `int sumOfNodes(Node *root)`

12.6.2 In our own words

Add up the values of all the nodes. By hand, $6+3+10+1+5+7+8+9=49$. The recursive definition follows the insight box of Section 12.1 word for word:

$$S(\varnothing)=0, \qquad S(T)=T.key+S(T.left)+S(T.right). \tag{12.5}$$

The base case is 0 because 0 is the number that changes nothing when added: the sum of a leaf must come out as the leaf's own value, and by Equation (12.5) it is $key+0+0$.

12.6.3 By hand, step by step

Figure 12.4 writes, next to each node, the value its call returns. Read it from the bottom up, because that is the order in which the calls *finish*: a call can only return when both of its children's calls have returned.

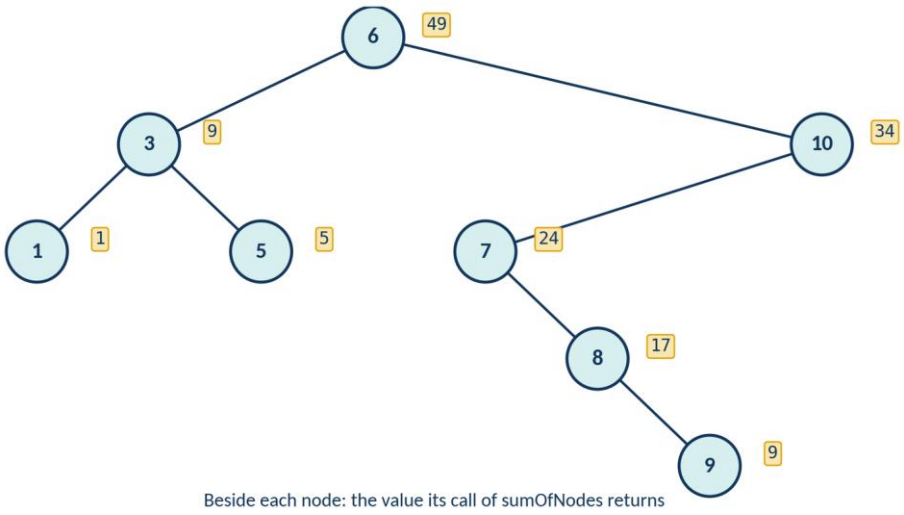


Figure 12.4 The values returned by `sumOfNodes` on each subtree. Each is the node's own value plus the two values returned below it; the root returns 49.

The leaves return their own values: 1, 5 and 9. Node 8 returns $8 + 0 + 9 = 17$, node 7 returns $7 + 0 + 17 = 24$, node 10 returns $10 + 24 + 0 = 34$, node 3 returns $3 + 1 + 5 = 9$, and the root returns $6 + 9 + 34 = 49$. As a trace of the calls, indented by depth:

```
sumOfNodes(6)
    sumOfNodes(3)
        sumOfNodes(1) = 1 + 0 + 0 = 1
        sumOfNodes(5) = 5 + 0 + 0 = 5
    = 3 + 1 + 5 = 9
    sumOfNodes(10)
        sumOfNodes(7)
            sumOfNodes(8)
                sumOfNodes(9) = 9 + 0 + 0 = 9
            = 8 + 0 + 9 = 17
        = 7 + 0 + 17 = 24
    = 10 + 24 + 0 = 34
= 6 + 9 + 34 = 49
```

12.6.4 The C++ solution

Listing 12.13 is the C++ program for Problem 12.3.

Listing 12.13 `p1203_sum.cpp`: Problem 12.3, one line of recursion.

```
// p1203_sum.cpp - Problem 12.3: the sum of all the values in a binary tree.
#include "tree_kit.hpp"

int sumOfNodes(Node *root) {
    if (root == nullptr) return 0;       // the empty tree adds nothing
    return root->data + sumOfNodes(root->left) + sumOfNodes(root->right);
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    printf("sum of the whole tree  : %d\n", sumOfNodes(root));
    printf("sum again (same answer): %d\n", sumOfNodes(root));
    printf("left subtree of 6      : %d\n", sumOfNodes(root->left));
    printf("right subtree of 6     : %d\n", sumOfNodes(root->right));
    printf("empty tree             : %d\n", sumOfNodes(nullptr));
    clear(&root);
    return 0;
}
```

```
sum of the whole tree  : 49
sum again (same answer): 49
left subtree of 6      : 9
right subtree of 6     : 34
empty tree             : 0
```

Line by line. The whole tree sums to 49. The second line calls the function again and gets 49 again; that looks too obvious to test, and the wrong solution below shows why it is not. The left and right subtrees of 6 give 9 and 34, and $6 + 9 + 34 = 49$ checks the answer a second way. The empty tree gives 0. **Cost.** $\Theta(n)$ by Equation (12.1), and $O(h)$ stack. (For very large trees with large values, an `int` could overflow; a `long long` sum would be the careful choice. Our values are small.)

12.6.5 The Java and Python solutions

Listing 12.14 and **Listing 12.15** are the Java and Python versions; both print exactly the same lines.

Listing 12.14 `P1203Sum.java`: Problem 12.3 in Java.

```java
// P1203Sum.java - Problem 12.3: the sum of all the values in a binary tree.
public class P1203Sum extends TreeKit {

    static int sumOfNodes(Node root) {
        if (root == null) return 0;           // the empty tree adds nothing
        return root.data + sumOfNodes(root.left) + sumOfNodes(root.right);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        System.out.println("sum of the whole tree  : " + sumOfNodes(root));
        System.out.println("sum again (same answer): " + sumOfNodes(root));
        System.out.println("left subtree of 6      : " + sumOfNodes(root.left));
        System.out.println("right subtree of 6     : " + sumOfNodes(root.right));
        System.out.println("empty tree             : " + sumOfNodes(null));
    }
}
```

Listing 12.15 `p1203_sum.py`: Problem 12.3 in Python.

```python
# p1203_sum.py - Problem 12.3: the sum of all the values in a binary tree.
from tree_kit import *


def sum_of_nodes(root):
    if root is None:                    # the empty tree adds nothing
        return 0
    return root.data + sum_of_nodes(root.left) + sum_of_nodes(root.right)


def main():
    root = build_sample()
    print("sum of the whole tree  :", sum_of_nodes(root))
    print("sum again (same answer):", sum_of_nodes(root))
    print("left subtree of 6      :", sum_of_nodes(root.left))
    print("right subtree of 6     :", sum_of_nodes(root.right))
    print("empty tree             :", sum_of_nodes(None))


if __name__ == "__main__":
    main()
```

12.6.6 A tempting wrong solution

Programmers who learned to add numbers with a loop often carry the loop's habit into recursion: keep a running total in a variable outside the function, and let every call add to it.

```
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;
}
```

The first call even gives the right answer, which is what makes this mistake so tempting. The program `wrong/p1203_wrong.cpp` prints:

```
sum of the whole tree  : 49
sum again (same answer): 98
left subtree of 6      : 107
right subtree of 6     : 141
empty tree             : 141
```

The variable `total` lives *outside* every call, so it remembers everything that any earlier call added. The second call on the whole tree starts from 49 and ends at 98; the call on the left subtree adds 9 to 98 and reports 107; even the empty tree reports 141. A function whose answer depends on what was called before is not computing a property of the tree any more. The correct version keeps nothing between calls: each call *returns* its sum, and the parent adds the returned values. The general rule is Step 4 of the method: a recursive function combines what its calls *return*; it does not communicate through variables they share.

12.6.7 Check yourself

CHECK YOURSELF. Write `int maxValue(Node *root)` for *any* binary tree (not necessarily a search tree), returning `INT_MIN` for the empty tree. Why is `INT_MIN` the right base case here, when 0 was right for the sum?

Answer. `if (root == nullptr) return INT_MIN; return max(root->data, max(maxValue(root->left), maxValue(root->right)));`. The base case must be the value that changes nothing under the combining operation. For $+$ that is 0; for max it is the smallest possible value, because $\max(x,\text{INT\_MIN}) = x$ for every `int` x. A base case of 0 would give the wrong maximum for a tree whose values are all negative. (In a *search* tree the maximum is simply the rightmost node, found in $O(h)$ by the `max` function of Section 11.7.4.)

12.7 Problem 12.4 — Tripling Every Node

12.7.1 The problem

PROBLEM. Write a recursive function which multiplies by three every node of the tree. `void triple(Node *root)`

12.7.2 In our own words

Visit every node once and replace its value by three times that value. The function returns nothing; its whole effect is on the nodes. Since each node is changed independently of the others, the order of the visit does not matter; we use preorder (change the node, then its two subtrees)

because it reads naturally. Expected result: the preorder `6 3 1 5 10 7 8 9` becomes `18 9 3 15 30 21 24 27`.

One question is worth asking before we write anything: **is the tripled tree still a binary search tree?** It is, and the reason is one line of algebra. Multiplying by a positive number keeps the order of any two numbers:

$$a < b \quad \Rightarrow \quad 3a < 3b. \tag{12.6}$$

Every comparison that Equation (12.3) requires between a node and its descendants was true before, so by Equation (12.6) it is still true after. **Figure 12.5** shows the tree before and after: same shape, new values, still in order. (Multiplying by -3 would reverse every comparison and destroy the search tree; Problem 12.5 destroys it in a different way.)

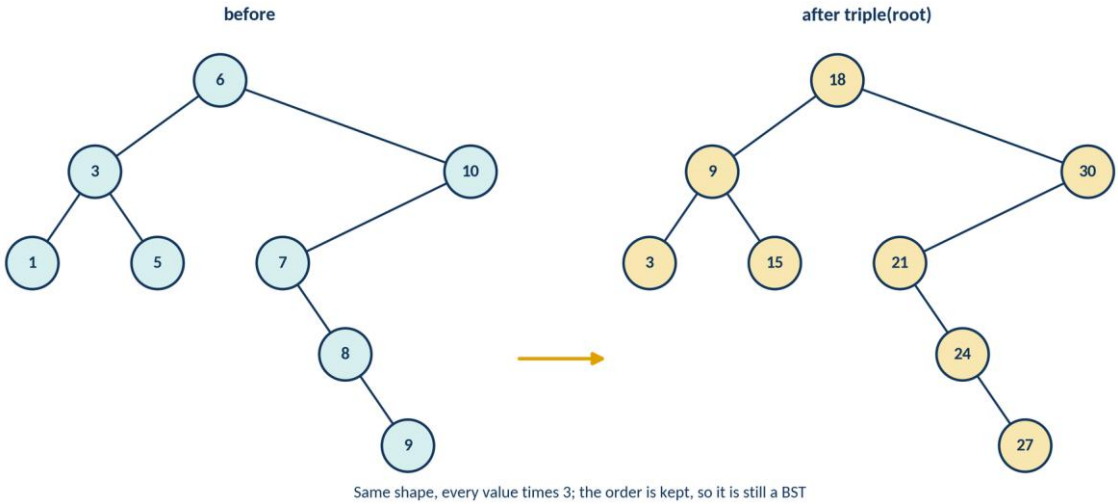


Figure 12.5 Problem 12.4: the tree before and after `triple(root)`. The shape does not change, and since $a < b$ implies $3a < 3b$, the tripled tree is still a binary search tree.

12.7.3 The C++ solution

Listing 12.16 is the C++ program for Problem 12.4.

Listing 12.16 `p1204_triple.cpp`: Problem 12.4, changing the nodes through the pointer.

```cpp
// p1204_triple.cpp - Problem 12.4: multiply every value in the tree by three.
#include "tree_kit.hpp"

void triple(Node *root) {
    if (root == nullptr) return;
    root->data *= 3;  // change the node itself, through the pointer
    triple(root->left);
    triple(root->right);
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    printf("before, preorder: ");
    preOrder(root);
    triple(root);
    printf("\nafter,  preorder: ");
    preOrder(root);
    printf("\nafter,  inorder : ");
    inOrder(root);
    printf("\nsearch 24: %s, search 8: %s\n",
           search(root, 24) != nullptr ? "found" : "not found",
           search(root, 8) != nullptr ? "found" : "not found");
    clear(&root);
    return 0;
}
```

```
before, preorder: 6 3 1 5 10 7 8 9
after,  preorder: 18 9 3 15 30 21 24 27
after,  inorder : 3 9 15 18 21 24 27 30
search 24: found, search 8: not found
```

Line by line. The preorder after tripling is exactly three times the preorder before. The inorder line is still increasing, which is the program's way of confirming Equation (12.6); and because the tree is still a search tree, `search` finds 24 (it follows $24 > 18$, $24 < 30$, $24 > 21$, found) and correctly fails to find 8, which is no longer in the tree. The key line is `root->data *= 3`: the function received a *pointer* to the node, and writing through the pointer changes the node itself, the one that lives in the tree. **Cost.** $\Theta(n)$.

12.7.4 The Java and Python solutions

Listing 12.17 and **Listing 12.18** are the Java and Python versions; both print exactly the same lines.

Listing 12.17 `P1204Triple.java`: Problem 12.4 in Java.

```java
// P1204Triple.java - Problem 12.4: multiply every value in the tree by three.
public class P1204Triple extends TreeKit {

    static void triple(Node root) {
        if (root == null) return;
        root.data *= 3;  // change the node itself, through the reference
        triple(root.left);
        triple(root.right);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        System.out.print("before, preorder: ");
        preOrder(root);
        triple(root);
        System.out.print("\nafter,  preorder: ");
        preOrder(root);
        System.out.print("\nafter,  inorder : ");
        inOrder(root);
        System.out.printf("%nsearch 24: %s, search 8: %s%n",
                search(root, 24) != null ? "found" : "not found",
                search(root, 8) != null ? "found" : "not found");
    }
}
```

Listing 12.18 `p1204_triple.py`: Problem 12.4 in Python.

```python
# p1204_triple.py - Problem 12.4: multiply every value in the tree by three.
from tree_kit import *


def triple(root):
    if root is None:
        return
    root.data *= 3                      # change the node object itself
    triple(root.left)
    triple(root.right)


def main():
    root = build_sample()
    print("before, preorder: ", end="")
    pre_order(root)
    triple(root)
    print("\nafter,  preorder: ", end="")
    pre_order(root)
    print("\nafter,  inorder : ", end="")
    in_order(root)
    found = lambda x: "found" if search(root, x) is not None else "not found"
    print(f"\nsearch 24: {found(24)}, search 8: {found(8)}")


if __name__ == "__main__":
    main()
```

In Java and Python a variable of type `Node` holds a *reference* to the node, never a copy of it, so `root.data *= 3` changes the node in the tree, exactly as the C++ pointer does. That is why this function needs no extra care in those languages.

12.7.5 A tempting wrong solution

In C++, however, a small change to the parameter's type makes the function do nothing at all:

```
void triple(Node node) {                // WRONG: receives a copy of the node
    node.data *= 3;                     // triples the copy only
    if (node.left != nullptr) triple(*node.left);
    if (node.right != nullptr) triple(*node.right);
}
```

Called as `triple(*root)`, it compiles cleanly and `wrong/p1204_wrong.cpp` prints:

```
before, preorder: 6 3 1 5 10 7 8 9
after,  preorder: 6 3 1 5 10 7 8 9
after,  inorder : 1 3 5 6 7 8 9 10
search 24: not found, search 8: found
```

Nothing changed. A parameter of type `Node` is received *by value* (Chapter 6, and Problem 8.9): the function gets a fresh copy of the node's three fields, triples the copy's `data`, and throws the copy away when it returns. The copies' `left` and `right` fields still point to the real children, so the recursion reaches every node; it just never changes one. The cure is to receive the node by pointer, as the problem's signature `triple(Node *root)` does, or by reference, `triple(Node &node)`.

12.7.6 Check yourself

> **CHECK YOURSELF.** Write `void addToAll(Node *root, int d)` that adds `d` to every value. For which values of `d` is the result still a binary search tree? And for `void negate(Node *root)`, which replaces every value x by $-x$, what one extra step turns the result back into a search tree?
>
> *Answer.* Adding the same `d` keeps every comparison ($a < b \Rightarrow a + d < b + d$), so the result is a search tree for *every* `d`. Negating reverses every comparison, so after `negate` every left subtree holds larger values; swapping the two children of every node, which is exactly the mirror of Problem 12.5, turns it back into a search tree.

12.8 Problem 12.5 — The Mirror Image

12.8.1 The problem

> **PROBLEM.** Write a recursive function to create a mirror image of the binary tree. `void mirror(Node *root)`

12.8.2 In our own words

Hold the tree up to a mirror: what was on the left is now on the right, at every level. So at every node, the left subtree and the right subtree change places, and each of them is itself mirrored. By the insight box, that is the whole recursive case: *swap the two children of the root, then mirror each of them.* The base case: the mirror of the empty tree is the empty tree, so there is nothing to do. The function changes the tree in place and returns nothing.

12.8.3 By hand, step by step

Figure 12.6 shows the tree before and after.

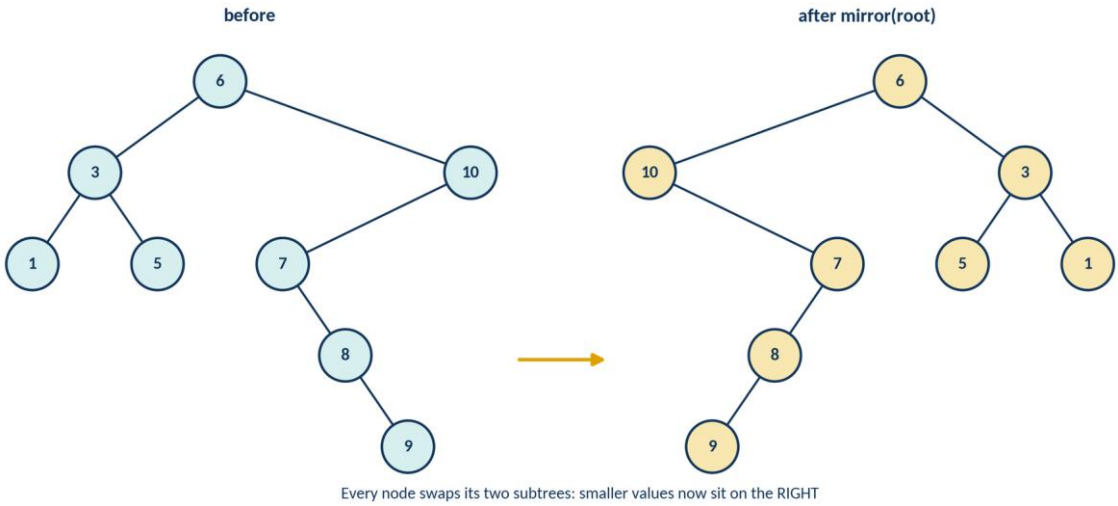


Figure 12.6 Problem 12.5: the tree before and after `mirror(root)`. Every node has swapped its two subtrees, so the smaller values now hang on the right; the result is no longer a binary search tree.

1. At 6: swap. Now 6's left is the subtree of 10 and its right is the subtree of 3.
2. Mirror the new left subtree, the one rooted at 10. At 10: its left was 7 and its right was empty; after the swap its left is empty and its right is 7. Mirror 7: its left was empty and its right 8; after the swap, 8 is its *left*. Mirror 8: 9 moves from its right to its left. Mirror 9: a leaf, nothing to swap.
3. Mirror the new right subtree, rooted at 3: 1 and 5 swap places, and both are leaves.

The preorder of the mirror is `6 10 7 8 9 3 5 1` (the answer of the check-yourself of Problem 12.1), and its inorder is the original inorder reversed, `10 9 8 7 6 5 3 1`, because inorder visits "left, node, right" and the mirror has exchanged left and right everywhere.

12.8.4 The C++ solution

Listing 12.19 is the C++ program for Problem 12.5.

Listing 12.19 `p1205_mirror.cpp`: Problem 12.5, a swap with a saved pointer at every node.

```cpp
// p1205_mirror.cpp - Problem 12.5: turn a binary tree into its mirror image.
#include "tree_kit.hpp"

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);
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    mirror(root);
    printf("The mirrored tree, turned on its side:\n");
    showSideways(root);
    printf("preorder: ");
    preOrder(root);
    printf("\ninorder : ");
    inOrder(root);
    printf("\nsearch 3 in the mirror: %s\n",
           search(root, 3) != nullptr ? "found" : "not found");
    mirror(root);                       // mirroring twice gives the original back
    printf("mirrored twice, inorder: ");
    inOrder(root);
    printf("\n");
    clear(&root);
    return 0;
}
```

```
The mirrored tree, turned on its side:
        1
    3
        5
6
        7
            8
                9
    10
preorder: 6 10 7 8 9 3 5 1
inorder : 10 9 8 7 6 5 3 1
search 3 in the mirror: not found
mirrored twice, inorder: 1 3 5 6 7 8 9 10
```

Line by line. The sideways picture, turned a quarter turn clockwise, is the right half of **Figure 12.6**. Preorder and inorder agree with the hand computation. Then comes a line to think about: **searching for 3 in the mirror fails**, although 3 is certainly in the tree. `search` relies on Equation (12.3): it compares $3 < 6$ and goes *left*, where the mirror now keeps the larger values 10, 7, 8, 9, and it falls off the tree without ever seeing 3. The mirror of a search tree is a perfectly good binary tree, but it is not a search tree, so the BST functions of Chapter 11 must not be used on it. The last line mirrors the tree a second time and gets the original inorder back: mirroring is its own inverse. **Cost.** $\Theta(n)$, three assignments per node.

12.8.5 The Java and Python solutions

Listing 12.20 and **Listing 12.21** are the Java and Python versions; both print exactly the same lines.

Listing 12.20 `P1205Mirror.java`: Problem 12.5 in Java.

```java
// P1205Mirror.java - Problem 12.5: turn a binary tree into its mirror image.
public class P1205Mirror extends TreeKit {

    static void mirror(Node root) {
        if (root == null) 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);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        mirror(root);
        System.out.println("The mirrored tree, turned on its side:");
        showSideways(root);
        System.out.print("preorder: ");
        preOrder(root);
        System.out.print("\ninorder : ");
        inOrder(root);
        System.out.println("\nsearch 3 in the mirror: "
                + (search(root, 3) != null ? "found" : "not found"));
        mirror(root);  // mirroring twice gives the original back
        System.out.print("mirrored twice, inorder: ");
        inOrder(root);
        System.out.println();
    }
}
```

Listing 12.21 `p1205_mirror.py`: Problem 12.5 in Python, with the swap written as one tuple assignment.

```python
# p1205_mirror.py - Problem 12.5: turn a binary tree into its mirror image.
from tree_kit import *


def mirror(root):
    if root is None:
        return
    root.left, root.right = root.right, root.left  # both old values read first
    mirror(root.left)                    # now mirror each of them
    mirror(root.right)


def main():
    root = build_sample()
    mirror(root)
    print("The mirrored tree, turned on its side:")
    show_sideways(root)
    print("preorder: ", end="")
    pre_order(root)
    print("\ninorder : ", end="")
    in_order(root)
    found = "found" if search(root, 3) is not None else "not found"
    print("\nsearch 3 in the mirror:", found)
    mirror(root)                         # mirroring twice gives the original back
    print("mirrored twice, inorder: ", end="")
    in_order(root)
    print()


if __name__ == "__main__":
    main()
```

The Python version swaps with `root.left, root.right = root.right, root.left`. This is safe because Python evaluates the whole right-hand side, both old values, *before* it assigns anything [10]; it is the same saved pointer as in C++, written for us by the language.

12.8.6 A tempting wrong solution

The classic error is to swap two variables with two assignments and no saved copy:

```
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 line copies that same pointer back. The left subtree is now unreachable, and the right subtree hangs from *both* sides. Running `wrong/p1205_wrong.cpp` under the sanitizers gives this (the report is shortened to its most useful lines):

```
The mirrored tree, turned on its side:
    10
6
    10
preorder: 6 10 10
inorder : 10 6 10
search 3 in the mirror: not found
mirrored twice, inorder: 10 6 10
==27839==ERROR: AddressSanitizer: heap-use-after-free on address 0x5030000000a8 at pc
0x55e840816e4f bp 0x7ffd7e794e60 sp 0x7ffd7e794e50
READ of size 8 at 0x5030000000a8 thread T0
    #0 0x55e840816e4e in clear(Node**) ../tree_kit.hpp:29
    #3 0x55e8408168d4 in main p1205_wrong.cpp:28
freed by thread T0 here:
    #1 0x55e840817006 in clear(Node**) ../tree_kit.hpp:32
SUMMARY: AddressSanitizer: heap-use-after-free ../tree_kit.hpp:29 in clear(Node**)
```

Three things went wrong, and the output shows all of them. The same mistake at node 10 made 10 lose its only child 7, so 7, 8 and 9 vanished too, and the "tree" printed is 6 with the node 10 hanging from both sides. Six nodes, 3, 1 and 5 and then 7, 8 and 9, are unreachable: a leak. And when `clear` frees the tree, it frees node 10 through the left pointer of 6 and then, following the right pointer of 6, reads the node it has just freed: a *use after free*, which is what the report says, with the line that read it (`clear`, line 29) and the line that had freed it (line 32). The cure is the rule of Chapter 8 in a new place: before overwriting a pointer, save what it points to.

12.8.7 Check yourself

> **CHECK YOURSELF.** Does the order matter? Rewrite `mirror` so that it first mirrors the two subtrees and *then* swaps them (a postorder version). Is the result the same? Then write `bool isMirror(Node *a, Node *b)`, which says whether tree `b` is the mirror image of tree `a`.
>
> *Answer.* The postorder version gives exactly the same tree: whether you mirror two subtrees before or after exchanging them, each ends up mirrored and on the other side. `isMirror`: `if (a == nullptr || b == nullptr) return a == b; return a->data == b->data && isMirror(a->left, b->right) && isMirror(a->right, b->left);`. Note how it compares a left with a right: that crossing is the whole definition of a mirror.

12.9 Problem 12.6 — Is the Tree Height-Balanced?

12.9.1 The problem

> **PROBLEM.** Write a recursive function to check if the binary tree is height-balanced (the difference between the heights of the left and right subtrees of any node is not more than 1). `int isBalanced(Node *root)` You may use the helper function `int isBalancedHelper(Node *root)`, where you use the function `int height(Node *root)` that is already defined.

12.9.2 In our own words

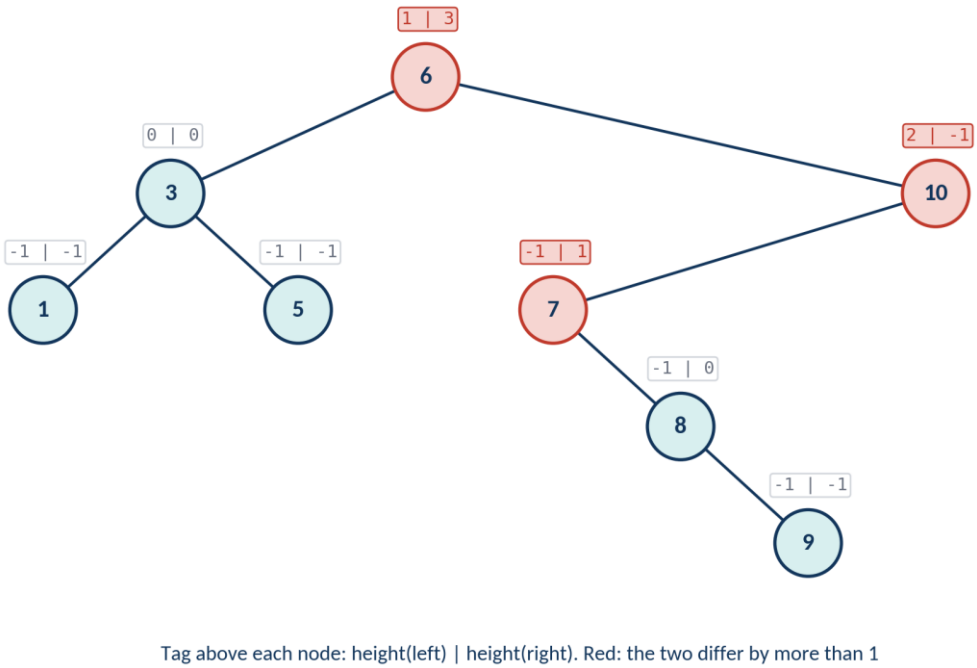
A node is *balanced* when the heights of its two subtrees differ by at most 1. The *tree* is height-balanced when **every** node is balanced, not only the root. The function answers 1 (yes) or 0 (no), as the `int` return type of the statement suggests.

> **DEFINITION.** A binary tree T is *height-balanced* if T is empty, or if $|h(T.left) - h(T.right)| \leq 1$ and both $T.left$ and $T.right$ are height-balanced. Trees with this property are the shape kept by the AVL trees of Section 11.9.2, due to Adelson-Velsky and Landis [11].

The definition is already a recursive function, with its base case in the first three words: **the empty tree is balanced**. Keep that sentence in mind; the wrong solution below gets it wrong.

12.9.3 By hand, step by step

Figure 12.7 writes above each node the heights of its two subtrees, computed with Equation (12.2).



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.

- The leaves 1, 5 and 9 have two empty subtrees: $|-1-(-1)| = 0$. Balanced.
- Node 3 has two leaves: $|0-0| = 0$. Balanced.
- Node 8 has an empty left and the leaf 9 on the right: $|-1-0| = 1$. Balanced, just.
- Node 7 has an empty left and the subtree 8–9, of height 1, on the right: $|-1-1| = 2$. **Not balanced.**

- Node 10 has the subtree 7–8–9, of height 2, on the left and nothing on the right: $|2 - (-1)| = 3$. **Not balanced.**
- The root 6 has heights 1 and 3: $|1 - 3| = 2$. **Not balanced.**

So the answer for our tree is 0. The tree `6 3 9 1 5 8 10`, on the other hand, is complete (every level full), every node's two heights are equal, and the answer is 1. The most instructive test is a tree whose root is balanced but which hides an unbalanced node lower down, such as the tree built from 5, 3, 8, 1, 4, 7, 9, 0, 10, 11: its root has subtrees of heights 2 and 3, which is fine, but node 8 has heights 0 and 2. A function that checked only the root would wrongly say yes.

12.9.4 The C++ solution

Listing 12.22 contains two solutions. The first is the one the statement asks for: `isBalancedHelper` checks the node with two calls of `height`, then checks both subtrees. The second, `checkedHeight`, computes the height and the verdict in the same walk: it returns the height of a balanced subtree, or a special value, `UNBALANCED`, as soon as it finds a node that breaks the rule.

Listing 12.22 `p1206_balanced.cpp`: Problem 12.6, the helper that uses `height`, and a one-pass version.

```
// p1206_balanced.cpp - Problem 12.6: is every node height-balanced?
#include <cstdlib>
#include "tree_kit.hpp"

// 1 if every node of this subtree has |height(left) - height(right)| <= 1.
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);
}

int isBalanced(Node *root) {
    return isBalancedHelper(root);
}

// 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;
}

void report(const char *name, Node *root) {
    printf("%-22s isBalanced = %d, one-pass = %d\n", name, isBalanced(root),
           checkedHeight(root) != UNBALANCED ? 1 : 0);
}

int main() {
    Node *sample = nullptr, *full = nullptr, *chain = nullptr, *empty = nullptr;
    Node *deep = nullptr;
    buildSample(&sample);                            // 6 3 10 1 5 7 8 9
    insertAll(&full, {6, 3, 9, 1, 5, 8, 10});
    insertAll(&chain, {1, 2, 3});
    insertAll(&deep, {5, 3, 8, 1, 4, 7, 9, 0, 10, 11});
    report("sample tree", sample);
    report("6 3 9 1 5 8 10", full);
    report("chain 1 2 3", chain);
    report("empty tree", empty);
    report("5 3 8 1 4 7 9 0 10 11", deep);  // the root is fine, node 8 is not
```

Listing 12.22 (Continued) *p1206_balanced.cpp: Problem 12.6, the helper that uses height, and a one-pass version.*

```
    clear(&sample);
    clear(&full);
    clear(&chain);
    clear(&deep);
    return 0;
}
```

```
sample tree            isBalanced = 0, one-pass = 0
6 3 9 1 5 8 10         isBalanced = 1, one-pass = 1
chain 1 2 3            isBalanced = 0, one-pass = 0
empty tree             isBalanced = 1, one-pass = 1
5 3 8 1 4 7 9 0 10 11  isBalanced = 0, one-pass = 0
```

Line by line. The two versions agree on all five trees. Our sample tree is not balanced; the complete tree is; the chain `1 2 3` is not (its root has heights -1 and 1); the empty tree is, by definition; and the tree with the hidden unbalanced node 8 is correctly rejected.

Why `UNBALANCED` is -2 and not -1. Some books signal "unbalanced" with -1, but here -1 is the height of the empty tree, so every empty subtree would look unbalanced. A height is never below -1, so -2 is safe.

Cost. The one-pass version visits every node once: $\Theta(n)$ by Equation (12.1). The helper version is slower, because at every node it checks it calls `height` on both subtrees, and `height` walks the whole subtree. Writing $size(x)$ for the number of nodes in the subtree of x, its cost is proportional to

$$C = \sum_{x \text{ checked}} size\,(x) = \sum_{x \text{ checked}} (d(x) + 1) \le n\,(h + 1), \tag{12.7}$$

where $d(x)$ is the depth of x. The middle step counts the same thing a second way: a node y is inside the subtree of each of its ancestors and of itself, $d(y) + 1$ subtrees in all. Equation (12.7) says the helper version costs $O(nh)$. That is not as bad as it looks, because the `&&` stops the checking at the first unbalanced node, and a tree in which all the checked nodes are balanced has a height of at most about $1.44 \lg n$ [11]; so the helper version costs $O(n \log n)$, against $\Theta(n)$ for the one-pass version. On a tree of a million nodes, that is the difference between about twenty million steps and one million.

12.9.5 The Java and Python solutions

Listing 12.23 and **Listing 12.24** are the Java and Python versions; both print exactly the same lines.

Listing 12.23 `P1206Balanced.java`: Problem 12.6 in Java.

```java
// P1206Balanced.java - Problem 12.6: is every node height-balanced?
public class P1206Balanced extends TreeKit {

    // true if every node of this subtree has |height(left) - height(right)| <= 1.
    static boolean isBalancedHelper(Node root) {
        if (root == null) return true;        // the empty tree is balanced
        int diff = Math.abs(height(root.left) - height(root.right));
        if (diff > 1) return false;           // this node breaks the rule
        return isBalancedHelper(root.left) && isBalancedHelper(root.right);
    }

    static int isBalanced(Node root) {
        return isBalancedHelper(root) ? 1 : 0;
    }

    // One pass: the height, or UNBALANCED as soon as a node breaks the rule.
    // UNBALANCED is -2, not -1: -1 already means "empty tree".
    static final int UNBALANCED = -2;

    static int checkedHeight(Node root) {
        if (root == null) return -1;
        int u = checkedHeight(root.left);
        if (u == UNBALANCED) return UNBALANCED;
        int v = checkedHeight(root.right);
        if (v == UNBALANCED) return UNBALANCED;
        if (Math.abs(u - v) > 1) return UNBALANCED;
        return Math.max(u, v) + 1;
    }

    static void report(String name, Node root) {
        System.out.printf("%-22s isBalanced = %d, one-pass = %d%n", name,
                isBalanced(root), checkedHeight(root) != UNBALANCED ? 1 : 0);
    }

    public static void main(String[] args) {
        report("sample tree", buildSample());
        report("6 3 9 1 5 8 10", insertAll(null, 6, 3, 9, 1, 5, 8, 10));
        report("chain 1 2 3", insertAll(null, 1, 2, 3));
        report("empty tree", null);
        report("5 3 8 1 4 7 9 0 10 11",
                insertAll(null, 5, 3, 8, 1, 4, 7, 9, 0, 10, 11));
    }
}
```

Listing 12.24 `p1206_balanced.py`: Problem 12.6 in Python.

```python
# p1206_balanced.py - Problem 12.6: is every node height-balanced?
from tree_kit import *


def is_balanced_helper(root):
    """True if every node here has |height(left) - height(right)| <= 1."""
    if root is None:                    # the empty tree is balanced
        return True
    if abs(height(root.left) - height(root.right)) > 1:
        return False                    # this node breaks the rule
    return is_balanced_helper(root.left) and is_balanced_helper(root.right)


def is_balanced(root):
    return 1 if is_balanced_helper(root) else 0


UNBALANCED = -2                         # not -1: -1 already means "empty tree"


def checked_height(root):
    """One pass: the height, or UNBALANCED as soon as a node breaks the rule."""
    if root is None:
        return -1
    u = checked_height(root.left)
    if u == UNBALANCED:
        return UNBALANCED
    v = checked_height(root.right)
    if v == UNBALANCED:
        return UNBALANCED
    if abs(u - v) > 1:
        return UNBALANCED
    return max(u, v) + 1


def report(name, root):
    one_pass = 1 if checked_height(root) != UNBALANCED else 0
    print(f"{name:<22} isBalanced = {is_balanced(root)}, one-pass = {one_pass}")
```

Listing 12.24 (Continued) *p1206_balanced.py: Problem 12.6 in Python.*

```python
def main():
    report("sample tree", build_sample())
    report("6 3 9 1 5 8 10", insert_all(None, [6, 3, 9, 1, 5, 8, 10]))
    report("chain 1 2 3", insert_all(None, [1, 2, 3]))
    report("empty tree", None)
    report("5 3 8 1 4 7 9 0 10 11",
           insert_all(None, [5, 3, 8, 1, 4, 7, 9, 0, 10, 11]))


if __name__ == "__main__":
    main()
```

Both return `true`/`True` from the helper and turn it into 1 or 0 in `isBalanced`, so that the printed lines match the C++ program, where the helper returns an `int`.

12.9.6 A tempting wrong solution

The mistake that is easiest to make is in the very first line. The helper returns an `int` used as a truth value, and "nothing there" feels like 0:

```
    if (root == nullptr) return 0;       // WRONG: "empty tree: not balanced"
```

Everything else is unchanged, and `wrong/p1206_wrong.cpp` prints:

```
sample tree            isBalanced = 0, one-pass = 0
6 3 9 1 5 8 10         isBalanced = 0, one-pass = 1
chain 1 2 3            isBalanced = 0, one-pass = 0
empty tree             isBalanced = 0, one-pass = 1
5 3 8 1 4 7 9 0 10 11  isBalanced = 0, one-pass = 0
```

Every tree is now "not balanced", even the perfect one. The reason is that every tree ends in empty subtrees: at a leaf, the helper checks the leaf's two heights (fine) and then asks for `isBalancedHelper(leaf->left)`, which returns 0, and the `&&` carries that 0 all the way up to the root. The worst feature of this bug is that it passes the test of the problem's own example: the sample tree really is unbalanced, so the answer 0 looks right. Only the edge cases of Step 6, the empty tree and a tree that *is* balanced, expose it. The rule behind the correct base case is the same as for the sum: the base case is the value that changes nothing under the combining operation, and for `&&` that value is *true*.

12.9.7 Check yourself

> **CHECK YOURSELF.** (a) Is the chain `2 1 3`, that is 2 with children 1 and 3, balanced? Is the tree `4 2 6 1 3 5 7 0` balanced? (b) What is the smallest number of values you must insert, in some order, to get a tree of height 3 that is still height-balanced?
>
> *Answer.* (a) `2 1 3` is complete, so yes. In `4 2 6 1 3 5 7 0`, node 1 has heights 0 and −1, node 2 has 1 and 0, the root has 2 and 1: every difference is 1 at most, so yes. (b) Seven. A balanced tree of height 3 needs a subtree of height 2 and one of height at least 1; the smallest such trees have 4 and 2 nodes, so the smallest has $1 + 4 + 2 = 7$ nodes, for example by inserting 5, 3, 7, 2, 4, 6, 1. (The smallest balanced trees of each height are the *Fibonacci trees*, and their sizes, 1, 2, 4, 7, 12, 20, …, grow like the Fibonacci numbers of Section 10.5 [8], [11].)

12.10 Problem 12.7 — Deleting a Value

12.10.1 The problem

> **PROBLEM.** Write a recursive function to delete a particular data. `Node *delNode(Node *root, int delData)` You may use the helper function `Node *minValueNode(Node *node)`, where you find the minimum value of the node from the tree.

12.10.2 In our own words

Remove the node that holds `delData` from the binary search tree, free its memory, and leave a tree that is still a binary search tree and still holds every other value. If `delData` is not in the tree, change nothing. The function returns a `Node *`: the root of the tree after the deletion. That return value is the key to the whole problem, because the root itself may be the node that disappears.

Finding the node is `search` again: smaller goes left, larger goes right. The work is in removing it, and there are three cases, depending on how many children the node has. **Table 12.4** lists them and **Figure 12.8** draws one of each.

Table 12.4 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

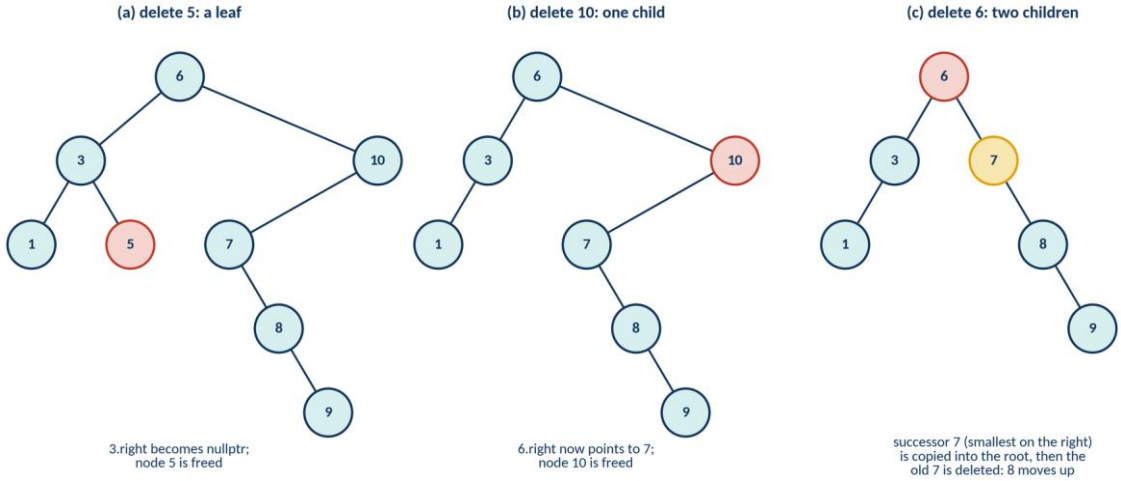


Figure 12.8 The three cases of Problem 12.7 on our tree: (a) the leaf 5 is simply removed; (b) 10 has one child, and 7 takes its place; (c) the root 6 has two children, so its successor 7 is copied into the root and the old 7 is deleted below.

Case 3 needs a word of explanation. We cannot simply remove a node with two children, because its parent has room for only one pointer where the node was, and there are two subtrees to hang. Instead we keep the node where it is and change its *value*. Which value can stand at that spot without breaking Equation (12.3)? It must be larger than everything in the left subtree and smaller than everything in the right subtree. The smallest value of the right subtree has exactly that property: it is larger than the node's old value (it is in the right subtree), hence larger than everything on the left, and it is smaller than every other value on the right. That value is called the *successor*, because it is the next value in sorted order, and it lives in the leftmost node of the right subtree, found by `minValueNode`. The leftmost node has no left child (otherwise it would not be leftmost), so deleting it from the right subtree afterwards is case 1 or case 2. This scheme is due to Hibbard [12], is the one of Section 11.7.10, and is the standard one in textbooks [1], [5].

12.10.3 By hand, step by step

We delete 5, then 10, then 6, then 42, from the same tree, one after the other.

1. **Delete 5.** $5 < 6$: go left, and whatever the left call returns becomes the new `6->left`. $5 > 3$: go right; whatever comes back becomes the new `3->right`. At 5: found, and it has no left child, so the call saves its right child (`nullptr`), frees node 5 and returns `nullptr`. Back at 3: `3->right = nullptr`. Back at 6: `6->left = 3`, unchanged. The preorder is now `6 3 1 10 7 8 9`.
2. **Delete 10.** $10 > 6$: go right. At 10: found; its left child is 7 but its right child is `nullptr`, so the call saves 7, frees 10 and returns 7. Back at 6: `6->right = 7`. The subtree 7–8–9 has moved up one level as a block, and it is still on the right side of 6, where it belongs. Preorder: `6 3 1 7 8 9`.

3. **Delete 6**, the root, which has two children, 3 and 7. `minValueNode(6->right)` starts at 7 and goes left as far as it can: 7 has no left child, so the successor is 7. Copy 7 into the root. Now there are two 7s, so delete 7 from the *right* subtree: at the old 7, found, no left child, so it is replaced by its right child 8. The root's right becomes 8. Preorder: `7 3 1 8 9`. The value 6 is gone, and the node that held it is still the root.
4. **Delete 42.** 42 > 7, 42 > 8, 42 > 9, and the right of 9 is `nullptr`: the base case returns `nullptr`, which 9 re-attaches as its right child, which it already was. Nothing changes.

12.10.4 The C++ solution

Listing 12.25 is the C++ program for Problem 12.7.

Listing 12.25 `p1207_delete.cpp`: Problem 12.7, deletion in all three cases, with every result re-attached by the parent.

```
// p1207_delete.cpp - Problem 12.7: delete a value from a binary search tree.
#include "tree_kit.hpp"

// The node with the smallest value in a non-empty subtree: its leftmost node.
Node *minValueNode(Node *node) {
    if (node->left == nullptr) return node;
    return minValueNode(node->left);
}

// Delete delData from the subtree and return the subtree's (possibly new) root.
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 if (root->right == nullptr) {              // no right child:
        Node *child = root->left;                     //   the left child takes
        delete root;                                  //   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;
}

void show(const char *what, Node *root) {
    printf("%-10s preorder: ", what);
    preOrder(root);
    printf("| inorder: ");
    inOrder(root);
    printf("\n");
}
```

Listing 12.25 (Continued) *p1207_delete.cpp: Problem 12.7, deletion in all three cases, with every result re-attached by the parent.*

```
int main() {
    Node *root = nullptr;
    buildSample(&root);
    show("start", root);
    root = delNode(root, 5);          // a leaf
    show("delete 5", root);
    root = delNode(root, 10);         // one child (7)
    show("delete 10", root);
    root = delNode(root, 6);          // the root, two children
    show("delete 6", root);
    root = delNode(root, 42);         // not in the tree
    show("delete 42", root);
    printf("The tree now, turned on its side:\n");
    showSideways(root);
    for (int v : {7, 3, 1, 8, 9}) root = delNode(root, v);
    printf("after deleting the rest: %s\n",
           root == nullptr ? "empty" : "not empty");
    return 0;
}
```

```
start      preorder: 6 3 1 5 10 7 8 9 | inorder: 1 3 5 6 7 8 9 10
delete 5   preorder: 6 3 1 10 7 8 9 | inorder: 1 3 6 7 8 9 10
delete 10  preorder: 6 3 1 7 8 9 | inorder: 1 3 6 7 8 9
delete 6   preorder: 7 3 1 8 9 | inorder: 1 3 7 8 9
delete 42  preorder: 7 3 1 8 9 | inorder: 1 3 7 8 9
The tree now, turned on its side:
        9
    8
7
    3
        1
after deleting the rest: empty
```

Line by line. Each line of the output agrees with the hand computation, and every inorder line is still increasing: the tree remained a search tree after every deletion. The sideways picture shows the resulting five-node tree, with 7 at the root. The last line deletes the remaining five values, including the root several times over, and ends with an empty tree; under Valgrind this program frees every one of its eight nodes.

Look at the shape of `delNode`. It never looks at its parent and never changes it. It only *returns* what should stand where it was, and every caller writes `root->left = delNode(root->left, delData)` or `root->right = ...`, and `main` writes `root = delNode(root, v)`. This is Step 5 of the method, and it is what makes the three cases so short: case 1 and case 2 are the same three lines, "save the only child, free the node, return the child", because in case 1 the "only child" is `nullptr`.

Cost. The function walks one path from the root down to the node, and in case 3 continues down to the successor, so it costs $O(h)$, where h is the height: between $O(\log n)$ for a short tree and $O(n)$ for a chain, by Equation (12.4).

12.10.5 The Java and Python solutions

Listing 12.26 and **Listing 12.27** are the Java and Python versions; both print exactly the same lines.

Listing 12.26 `P1207Delete.java`: Problem 12.7 in Java.

```
// P1207Delete.java - Problem 12.7: delete a value from a binary search tree.
public class P1207Delete extends TreeKit {

    // The node with the smallest value in a non-empty subtree: its leftmost node.
    static Node minValueNode(Node node) {
        if (node.left == null) return node;
        return minValueNode(node.left);
    }

    // Delete delData from the subtree; return the subtree's (maybe new) root.
    static Node delNode(Node root, int delData) {
        if (root == null) return null;              // 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 == null) {             // no left child:
            return root.right;                      //   the right child replaces it
        } else if (root.right == null) {            // no right child:
            return root.left;                       //   the left child replaces it
        } 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;
    }

    static void show(String what, Node root) {
        System.out.printf("%-10s preorder: ", what);
        preOrder(root);
        System.out.print("| inorder: ");
        inOrder(root);
        System.out.println();
    }
```

Listing 12.26 (Continued) *P1207Delete.java: Problem 12.7 in Java.*

```
    public static void main(String[] args) {
        Node root = buildSample();
        show("start", root);
        root = delNode(root, 5);          // a leaf
        show("delete 5", root);
        root = delNode(root, 10);         // one child (7)
        show("delete 10", root);
        root = delNode(root, 6);          // the root, two children
        show("delete 6", root);
        root = delNode(root, 42);         // not in the tree
        show("delete 42", root);
        System.out.println("The tree now, turned on its side:");
        showSideways(root);
        for (int v : new int[] {7, 3, 1, 8, 9}) root = delNode(root, v);
        System.out.println("after deleting the rest: "
                + (root == null ? "empty" : "not empty"));
    }
}
```

Listing 12.27 `p1207_delete.py`: Problem 12.7 in Python.

```
# p1207_delete.py - Problem 12.7: delete a value from a binary search tree.
from tree_kit import *


def min_value_node(node):
    """The node with the smallest value in a non-empty subtree: the leftmost."""
    if node.left is None:
        return node
    return min_value_node(node.left)


def del_node(root, del_data):
    """Delete del_data from the subtree; return the subtree's (maybe new) root."""
    if root is None:                                  # not found: nothing changes
        return None
    if del_data < root.data:
        root.left = del_node(root.left, del_data)     # re-attach the left side
    elif del_data > root.data:
        root.right = del_node(root.right, del_data)   # re-attach the right side
    elif root.left is None:                           # no left child:
        return root.right                             #   the right child replaces it
    elif root.right is None:                          # no right child:
        return root.left                              #   the left child replaces it
    else:                                             # two children:
        succ = min_value_node(root.right)             #   the next larger value
        root.data = succ.data                         #   is copied up here,
        root.right = del_node(root.right, succ.data)  # then deleted below
    return root


def show(what, root):
    print(f"{what:<10} preorder: ", end="")
    pre_order(root)
    print("| inorder: ", end="")
    in_order(root)
    print()
```

Listing 12.27 (Continued) *p1207_delete.py: Problem 12.7 in Python.*

```
def main():
    root = build_sample()
    show("start", root)
    root = del_node(root, 5)         # a leaf
    show("delete 5", root)
    root = del_node(root, 10)        # one child (7)
    show("delete 10", root)
    root = del_node(root, 6)         # the root, two children
    show("delete 6", root)
    root = del_node(root, 42)        # not in the tree
    show("delete 42", root)
    print("The tree now, turned on its side:")
    show_sideways(root)
    for v in (7, 3, 1, 8, 9):
        root = del_node(root, v)
    print("after deleting the rest:", "empty" if root is None else "not empty")


if __name__ == "__main__":
    main()
```

In Java and Python there is no `delete`: a node that nothing refers to any more is reclaimed by the runtime, so cases 1 and 2 shrink to “return the only child”. The structure of the function is otherwise identical.

12.10.6 A tempting wrong solution

The deletion itself is usually written correctly. What gets lost is the re-attachment on the way back:

```
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
```

It compiles without a warning. Under the sanitizers, `wrong/p1207_wrong.cpp` prints this (shortened):

```
start      preorder: 6 3 1 5 10 7 8 9 | inorder: 1 3 5 6 7 8 9 10
delete 5   preorder: 6 3 1 ==27864==ERROR: AddressSanitizer: heap-use-after-free on address
0x503000000100 at pc 0x55d504641918 bp 0x7ffebfd9d330 sp 0x7ffebfd9d320
READ of size 4 at 0x503000000100 thread T0
    #0 0x55d504641917 in preOrder(Node*) ../tree_kit.hpp:38
freed by thread T0 here:
    #1 0x55d50464081f in delNode(Node*, int) p1207_wrong.cpp:19
previously allocated by thread T0 here:
    #1 0x55d50464149f in insert(Node**, int) ../tree_kit.hpp:19
SUMMARY: AddressSanitizer: heap-use-after-free ../tree_kit.hpp:38 in preOrder(Node*)
```

The call on node 5 freed the node and returned `nullptr`, as it should, but node 3 ignored the returned value, so `3->right` still holds the address of the freed node, a *dangling pointer*. The next traversal follows it and reads freed memory. Without the sanitizer, the program might print a stale 5, or garbage, or crash, or appear to work; that uncertainty is exactly what "undefined behavior" means. The report names the three places that matter: where the freed memory was read (`preOrder`), where it was freed (`delNode`, line 19), and where it was allocated (`insert`). The fix is one assignment in each branch: re-attach what the child returns.

A second, quieter mistake is to free the node with `free(root)` when it was created with `new`. C++ requires `delete` for memory from `new` and `free` for memory from `malloc`; mixing them is undefined behavior, and AddressSanitizer reports it as an *alloc-dealloc-mismatch* [13], [2].

12.10.7 Check yourself

> **CHECK YOURSELF.** Case 3 can equally use the *predecessor*, the largest value in the left subtree. Delete 6 from the original tree that way: which value moves into the root, and what is the preorder afterwards? Why does the predecessor never have a right child?
>
> *Answer.* The largest value on the left is found by going right from 3 as far as possible: 5. Copy 5 into the root and delete the old 5, a leaf. The preorder is `5 3 1 10 7 8 9`. The predecessor is the rightmost node of the left subtree; if it had a right child, that child would be further right and larger, so it would not be the rightmost.

12.11 Problem 12.8 — Finding the Parent

12.11.1 The problem

> **PROBLEM.** Write a recursive function to find the parent node of a given child data. `Node *findParent(Node *root, int *childData)`

12.11.2 In our own words

Given a value, return the node whose left or right child holds that value. The root has no parent, and a value that is not in the tree has none either; in both cases we return `nullptr`, and a caller

who wants to tell them apart can ask `search` whether the value exists. For our tree: the parent of 8 is 7, of 10 is 6, of 9 is 8, of 1 is 3; 6 is the root; 4 is not in the tree.

The statement passes the value as `int *childData`, a pointer to an `int`. Chapter 6 explained when a function needs a pointer parameter: when it must *change* the caller's variable, or when the variable is too large to copy. Neither applies to one `int` that the function only reads, so our function takes `int childData` by value, which also lets the caller write `findParent(root, 8)` instead of declaring a variable just to take its address. If you must keep the given signature, the body is the same with `*childData` wherever we write `childData`, and the call becomes `findParent(root, &x)`.

12.11.3 By hand, step by step

Because the tree is a binary search tree, the node holding the value lies on the single path that `search` would follow, and so does its parent: the parent is simply the *last node before* the value on that path. **Figure 12.9** follows the path for 8.

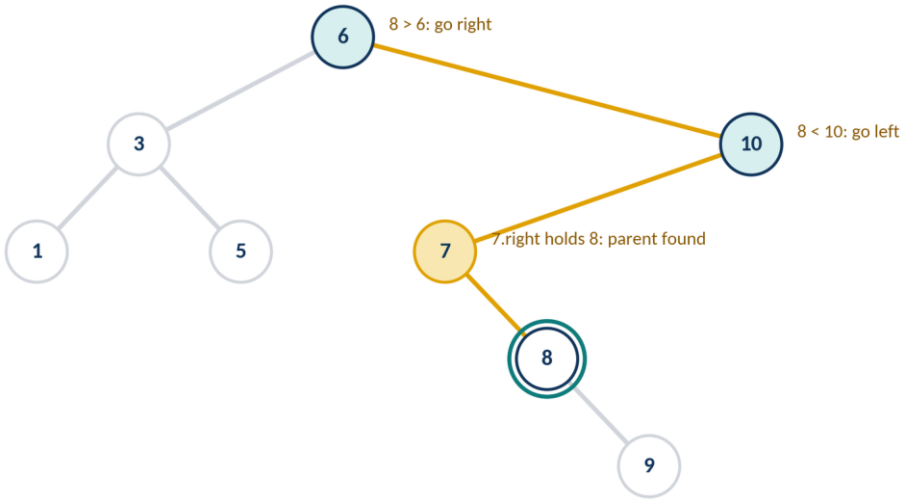


Figure 12.9 `findParent(root, 8)` follows the search path for 8. At every node it looks one step ahead; at 7 the next node on the path holds 8, so 7 is the parent.

1. At 6: $8 > 6$, so the next node on the path is the right child, 10. It does not hold 8, so ask the same question of 10.
2. At 10: $8 < 10$, so the next node is 7. Not 8; ask 7.
3. At 7: $8 > 7$, so the next node is 7's right child, which holds 8. **7 is the parent.**

For 6, the very first test finds the value at the root, and the answer is `nullptr`. For 4, the path is 6, 3, 5, and the left child of 5 is `nullptr`: the path ends, 4 is not there, and the answer is `nullptr`.

12.11.4 The C++ solution

Listing 12.28 gives two versions. `findParent` uses the order of the search tree and walks a single path. `findParentAnyTree` works on any binary tree, ordered or not, for example on the mirror of Problem 12.5: it checks the node's own children and, if neither holds the value, looks for the parent in the left subtree and then in the right one. The program runs both and checks that they agree.

Listing 12.28 `p1208_find_parent.cpp`: Problem 12.8, the search-tree version and the any-tree version.

```cpp
// p1208_find_parent.cpp - Problem 12.8: the parent of the node holding a value.
#include "tree_kit.hpp"

// BST version: walk the single path that childData would follow.
// Returns nullptr when childData is the root or is not in the tree.
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
}

// Any binary tree, ordered or not: look on both sides.
Node *findParentAnyTree(Node *root, int childData) {
    if (root == nullptr) return nullptr;
    if ((root->left != nullptr && root->left->data == childData) ||
        (root->right != nullptr && root->right->data == childData)) {
        return root;
    }
    Node *found = findParentAnyTree(root->left, childData);
    if (found != nullptr) return found;
    return findParentAnyTree(root->right, childData);
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    for (int x : {8, 10, 9, 1, 6, 4}) {
        Node *p = findParent(root, x);
        Node *q = findParentAnyTree(root, x);
        printf("parent of %d: ", x);
        if (p != nullptr) {
            printf("%d", p->data);
        } else if (search(root, x) != nullptr) {
            printf("none (%d is the root)", x);
        } else {
            printf("none (%d is not in the tree)", x);
        }
        printf("  [both versions agree: %s]\n", p == q ? "yes" : "no");
    }
    clear(&root);
    return 0;
}
```

```
parent of 8: 7  [both versions agree: yes]
parent of 10: 6  [both versions agree: yes]
parent of 9: 8  [both versions agree: yes]
parent of 1: 3  [both versions agree: yes]
parent of 6: none (6 is the root)  [both versions agree: yes]
parent of 4: none (4 is not in the tree)  [both versions agree: yes]
```

Line by line. All six answers match the hand computation, and the two versions agree on every one (two `nullptr`s compare equal too). The BST version checks `next == nullptr` before it reads `next->data`. **Cost.** The BST version walks one path, $O(h)$; the any-tree version may look everywhere, $\Theta(n)$ when the value is absent.

12.11.5 The Java and Python solutions

Listing 12.29 and **Listing 12.30** are the Java and Python versions; both print exactly the same lines.

Listing 12.29 `P1208FindParent.java`: Problem 12.8 in Java.

```java
// P1208FindParent.java - Problem 12.8: the parent of the node holding a value.
public class P1208FindParent extends TreeKit {

    // BST version: walk the single path that childData would follow.
    // Returns null when childData is the root or is not in the tree.
    static Node findParent(Node root, int childData) {
        if (root == null || root.data == childData) return null;
        Node next = (childData < root.data) ? root.left : root.right;
        if (next == null) return null;  // 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
    }

    // Any binary tree, ordered or not: look on both sides.
    static Node findParentAnyTree(Node root, int childData) {
        if (root == null) return null;
        if ((root.left != null && root.left.data == childData)
                || (root.right != null && root.right.data == childData)) {
            return root;
        }
        Node found = findParentAnyTree(root.left, childData);
        if (found != null) return found;
        return findParentAnyTree(root.right, childData);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        for (int x : new int[] {8, 10, 9, 1, 6, 4}) {
            Node p = findParent(root, x);
            Node q = findParentAnyTree(root, x);
            System.out.print("parent of " + x + ": ");
            if (p != null) {
                System.out.print(p.data);
            } else if (search(root, x) != null) {
                System.out.print("none (" + x + " is the root)");
            } else {
                System.out.print("none (" + x + " is not in the tree)");
            }
            System.out.println("  [both versions agree: "
                    + (p == q ? "yes" : "no") + "]");
        }
    }
}
```

Listing 12.30 `p1208_find_parent.py`: Problem 12.8 in Python.

```
# p1208_find_parent.py - Problem 12.8: find the parent of the node holding a value.
from tree_kit import *


def find_parent(root, child_data):
    """BST version: walk the single path that child_data would follow.
    Returns None when child_data is the root or is not in the tree."""
    if root is None or root.data == child_data:
        return None
    nxt = root.left if child_data < root.data else root.right
    if nxt is None:                                 # the path ends: not in the tree
        return None
    if nxt.data == child_data:                      # root is the parent
        return root
    return find_parent(nxt, child_data)             # ask the same question below


def find_parent_any_tree(root, child_data):
    """Any binary tree, ordered or not: look on both sides."""
    if root is None:
        return None
    if (root.left is not None and root.left.data == child_data) or \
       (root.right is not None and root.right.data == child_data):
        return root
    found = find_parent_any_tree(root.left, child_data)
    if found is not None:
        return found
    return find_parent_any_tree(root.right, child_data)


def main():
    root = build_sample()
    for x in (8, 10, 9, 1, 6, 4):
        p = find_parent(root, x)
        q = find_parent_any_tree(root, x)
        if p is not None:
            answer = str(p.data)
        elif search(root, x) is not None:
            answer = f"none ({x} is the root)"
        else:
            answer = f"none ({x} is not in the tree)"
        agree = "yes" if p is q else "no"
        print(f"parent of {x}: {answer}  [both versions agree: {agree}]")
```

Listing 12.30 (Continued) *p1208_find_parent.py: Problem 12.8 in Python.*

```
if __name__ == "__main__":
    main()
```

The Java program compares the two results with `p == q`, and the Python program with `p is q`: both ask whether the two variables refer to the *same* node, which is the right question here.

12.11.6 A tempting wrong solution

The natural first attempt asks “does one of my children hold the value?” in one line:

```
if (root->left->data == childData || root->right->data == childData) {
    return root;                          // WRONG: a child may be nullptr
}
```

On our tree, `wrong/p1208_wrong.cpp` dies on its very first question:

```
p1208_wrong.cpp:8:55: runtime error: member access within null pointer of type 'struct
Node'
==27876==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x558d9ab50593
bp 0x7ffc1a5e0460 sp 0x7ffc1a5e0430 T0)
    #0 0x558d9ab50593 in findParent(Node*, int) p1208_wrong.cpp:8
    #1 0x558d9ab5070d in findParent(Node*, int) p1208_wrong.cpp:12
SUMMARY: AddressSanitizer: SEGV p1208_wrong.cpp:8 in findParent(Node*, int)
```

Looking for 8, the call on 6 survives (both of its children exist), moves to 10, and there reads `root->right->data`, but 10 has no right child. Column 55 of line 8 is exactly that expression. Half of the nodes of a typical tree have a missing child, so a function that follows a child pointer without testing it will crash on almost any input. Test the pointer before you follow it, as both correct versions do.

12.11.7 Check yourself

> **CHECK YOURSELF.** Write `Node *findGrandparent(Node *root, int x)` using `findParent` twice. What does it return for 9, for 5 and for 3?
>
> *Answer.* `Node *p = findParent(root, x); return p == nullptr ? nullptr : findParent(root, p->data);`. For 9 the parent is 8 and 8's parent is 7. For 5 the parent is 3 and 3's parent is 6. For 3 the parent is 6, the root, which has no parent, so the result is `nullptr`.

12.12 Problem 12.9 — Reporting the Children

12.12.1 The problem

> **PROBLEM.** Write a recursive function to find the children node of a given parent data. `Node *findChild(Node *root, int parentData)` where you use the function `Node *search(Node *root, int data)` that is already defined.

12.12.2 In our own words

Find the node that holds `parentData` and report its left and right children, saying clearly when a child is missing and when the value is not in the tree at all. Return the node that was found, or `nullptr`. The recursion lives in `search`, which the statement tells us to use; `findChild` itself calls it once and then only looks at two pointers.

By hand, from **Figure 12.1**: the children of 3 are 1 (left) and 5 (right); 7 has no left child and 8 on the right; 9 is a leaf; 4 is not in the tree.

12.12.3 The C++ solution

Listing 12.31 is the C++ program for Problem 12.9.

Listing 12.31 `p1209_find_child.cpp`: Problem 12.9, one call of `search` and two careful looks.

```cpp
// p1209_find_child.cpp - Problem 12.9: the children of the node holding a value.
#include "tree_kit.hpp"

// Print the children of parentData and return its node (nullptr if absent).
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;
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    for (int x : {3, 7, 9, 4}) {
        Node *n = findChild(root, x);
        printf("  -> returned %s\n", n != nullptr ? "the node" : "no node");
    }
    clear(&root);
    return 0;
}
```

```
left child of 3: 1
right child of 3: 5
  -> returned the node
7 has no left child
right child of 7: 8
  -> returned the node
9 has no left child
9 has no right child
  -> returned the node
4 is not in the tree
  -> returned no node
```

Line by line. Each value produces two lines about its children, or one line saying it is absent, and the indented line shows what the function returned. The three tests `parent == nullptr`, `parent->left != nullptr` and `parent->right != nullptr` are the whole of the difficulty: each guards a pointer before it is followed. **Cost.** One `search`, $O(h)$, plus constant work.

12.12.4 The Java and Python solutions

Listing 12.32 and **Listing 12.33** are the Java and Python versions; both print exactly the same lines.

Listing 12.32 `P1209FindChild.java`: Problem 12.9 in Java.

```java
// P1209FindChild.java - Problem 12.9: the children of the node holding a value.
public class P1209FindChild extends TreeKit {

    // Print the children of parentData and return its node (null if absent).
    static Node findChild(Node root, int parentData) {
        Node parent = search(root, parentData);   // the recursive walk
        if (parent == null) {
            System.out.println(parentData + " is not in the tree");
            return null;
        }
        if (parent.left != null) {
            System.out.println("left child of " + parentData + ": "
                    + parent.left.data);
        } else {
            System.out.println(parentData + " has no left child");
        }
        if (parent.right != null) {
            System.out.println("right child of " + parentData + ": "
                    + parent.right.data);
        } else {
            System.out.println(parentData + " has no right child");
        }
        return parent;
    }

    public static void main(String[] args) {
        Node root = buildSample();
        for (int x : new int[] {3, 7, 9, 4}) {
            Node n = findChild(root, x);
            System.out.println("  -> returned "
                    + (n != null ? "the node" : "no node"));
        }
    }
}
```

Listing 12.33 `p1209_find_child.py`: Problem 12.9 in Python.

```python
# p1209_find_child.py - Problem 12.9: the children of the node holding a value.
from tree_kit import *


def find_child(root, parent_data):
    """Print the children of parent_data and return its node (None if absent)."""
    parent = search(root, parent_data)       # the recursive walk
    if parent is None:
        print(f"{parent_data} is not in the tree")
        return None
    if parent.left is not None:
        print(f"left child of {parent_data}: {parent.left.data}")
    else:
        print(f"{parent_data} has no left child")
    if parent.right is not None:
        print(f"right child of {parent_data}: {parent.right.data}")
    else:
        print(f"{parent_data} has no right child")
    return parent


def main():
    root = build_sample()
    for x in (3, 7, 9, 4):
        n = find_child(root, x)
        print("  -> returned", "the node" if n is not None else "no node")


if __name__ == "__main__":
    main()
```

12.12.5 A tempting wrong solution

The tempting version trusts `search` to succeed and goes straight to the children. It works for 3, 7 and 9, and `wrong/p1209_wrong.cpp` then crashes on 4:

```
9 has no left child
9 has no right child
  -> returned the node
p1209_wrong.cpp:8:17: runtime error: member access within null pointer of type 'struct
Node'
==27888==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000008 (pc 0x55b2d5ad13e7
bp 0x7fffa4ac8290 sp 0x7fffa4ac8260 T0)
    #0 0x55b2d5ad13e7 in findChild(Node*, int) p1209_wrong.cpp:8
SUMMARY: AddressSanitizer: SEGV p1209_wrong.cpp:8 in findChild(Node*, int)
```

The address in the report is `0x8`, not 0. The program read the field `left` of a node that does not exist: `search` returned `nullptr`, address 0, and the field `left` sits 8 bytes after the start of a `Node` (after the 4-byte `int` and 4 bytes of padding), so the program tried to read address 8. An address a few bytes above zero in a crash report is almost always a field of a null pointer. The fix is to test what `search` returned.

12.12.6 Check yourself

> **CHECK YOURSELF.** Write `int degree(Node *root, int x)`, the number of children of the node holding `x` (0, 1 or 2), returning -1 if `x` is absent. Which nodes of our tree have degree 1?
>
> *Answer.* `Node *p = search(root, x); if (p == nullptr) return -1; return (p->left != nullptr) + (p->right != nullptr);` (in C++ a `true` counts as 1). The nodes of degree 1 are 10, 7 and 8, the three nodes on the path down to 9 that have only one child; 6 and 3 have degree 2 and the leaves 1, 5 and 9 degree 0. Check: a binary tree with n_2 nodes of degree 2 has exactly $n_2 + 1$ leaves, and here $n_2 = 2$ and there are 3 leaves.

12.13 Problem 12.10 — From a Tree to a Doubly Linked List, in Place

12.13.1 The problem

> **PROBLEM.** Write a recursive function to convert the binary tree to a doubly linked list in place. The left pointer of the tree should act as the previous pointer and the right pointer should act as the next pointer. `Node *convertToDLL(Node *root)` You may use the helper function `void convertToDLLHelper(Node *root, Node **head, Node **prev)`. You may also use the following function for the traversal of the list: `void preOrderDLL(Node *head) { Node *current = head; while (current != nullptr) { printf("%d ", current->data); current = current->right; } }`

12.13.2 In our own words

A tree node and a doubly linked list node (Section 9.2) have the same shape: a value and two pointers. In the tree the two pointers are called `left` and `right` and mean "smaller subtree" and "larger subtree"; in a list they would be called `prev` and `next`. The problem asks us to *reuse* the tree's nodes, without creating any new ones (that is what *in place* means), and to rewire their two pointers so that `left` means "previous node in the list" and `right` means "next node in the list". The function returns the first node of the list, its *head*.

In what order should the nodes stand in the list? The statement does not say, but there is one natural answer for a binary search tree: sorted order, `1 3 5 6 7 8 9 10`, which is the inorder of the tree. **Figure 12.10** shows the tree and the list it becomes. (The given walking function is called `preOrderDLL`, but it does not do a preorder of anything; it walks the finished list from the head along the `right` pointers, and prints the list in list order.)

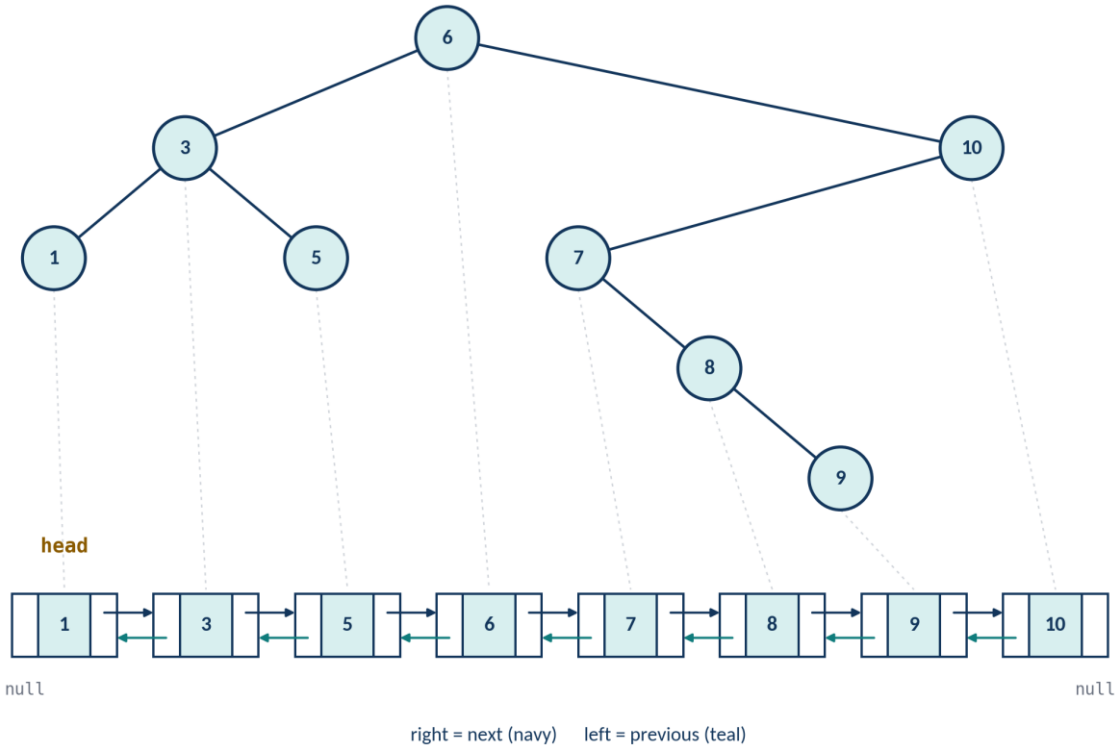


Figure 12.10 Problem 12.10: the same eight nodes, rewired. In the list, `right` points to the next larger value and `left` to the next smaller one; the dashed guides show that the list order is the inorder of the tree.

12.13.3 The idea, by hand

An inorder traversal visits the nodes in exactly the list order. So we walk the tree in inorder and, at each node we visit, *append* it to the list built so far. To append a node we need to know the last node of the list so far; call it `prev`. Appending `root` after `prev` means two assignments, the rule of two arrows from Chapter 9:

- `prev->right = root` (the forward arrow: the old last node now points to the new one), and
- `root->left = prev` (the backward arrow: the new node points back to the old last one).

Then `root` becomes the new last node: `prev = root`. The very first node visited has no `prev` (it is `nullptr`); that node is the head of the list, so we remember it in `head`.

There are two variables, `head` and `prev`, that every call of the helper must see *and change*, and the changes made deep inside the recursion must still be visible when the calls return. That is why the helper receives their addresses, `Node **head` and `Node **prev`: every call reads and writes the same two variables, which live in `convertToDLL`.

Here is the start of the walk on our tree, with `prev` after each step:

1. Inorder goes down the left side to 1. `prev` is `nullptr`, so `head = 1`; `1->left = nullptr`; `prev = 1`.
2. Back at 3: `1->right = 3`, `3->left = 1`, `prev = 3`.
3. Then 3's right subtree, 5: `3->right = 5`, `5->left = 3`, `prev = 5`.
4. Back at 6: `5->right = 6`, `6->left = 5`, `prev = 6`. Then 6's right subtree: 7, 8, 9, 10 are appended in the same way, and the last one, 10, keeps `right = nullptr`, the end of the list.

Is it safe to overwrite pointers we are still walking? This is the question that makes the problem hard, and it deserves a careful answer. We overwrite `root->left` only after the call on the left subtree has *finished*, so nobody needs the old left pointer any more. And `root->right` is overwritten only later, when the next node is appended with `prev->right = ...`; but by then the helper has already *read* `root->right` and passed it to the call on the right subtree, because a function's arguments are evaluated before the function starts. The old right pointer is used exactly once, before it is replaced. This is the rule of Chapter 8 again, *keep what you need before you change a link*, obeyed here by the order of the three steps of an inorder walk.

12.13.4 The C++ solution

Listing 12.34 is the C++ program for Problem 12.10.

Listing 12.34 `p1210_to_dll.cpp`: Problem 12.10, an inorder walk that appends every node to the list.

```
// p1210_to_dll.cpp - Problem 12.10: turn a binary tree into a doubly linked list
// in place: left acts as "previous" and right acts as "next".
#include "tree_kit.hpp"

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;
}

// As given in the problem: follow the "next" links from the head.
void preOrderDLL(Node *head) {
    Node *current = head;
    while (current != nullptr) {
        printf("%d ", current->data);
        current = current->right;
    }
}

// Go to the last node, then follow the "prev" links back.
void printBackward(Node *head) {
    Node *tail = head;
    while (tail != nullptr && tail->right != nullptr) tail = tail->right;
    for (Node *cur = tail; cur != nullptr; cur = cur->left) {
        printf("%d ", cur->data);
    }
}
```

Listing 12.34 (Continued) *p1210_to_dll.cpp: Problem 12.10, an inorder walk that appends every node to the list.*

```
void freeList(Node *head) {  // the tree is gone; free a list now
    while (head != nullptr) {
        Node *next = head->right;
        delete head;
        head = next;
    }
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    Node *head = convertToDLL(root);
    printf("forward : ");
    preOrderDLL(head);
    printf("\nbackward: ");
    printBackward(head);
    printf("\nhead %d, anything before the head? %s\n", head->data,
           head->left == nullptr ? "no" : "yes");
    freeList(head);

    Node *one = nullptr;
    insert(&one, 42);
    Node *h1 = convertToDLL(one);
    printf("one node: ");
    preOrderDLL(h1);
    printf("\nempty   : %s\n", convertToDLL(nullptr) == nullptr ? "no list" : "?");
    freeList(h1);
    return 0;
}
```

```
forward : 1 3 5 6 7 8 9 10
backward: 10 9 8 7 6 5 3 1
head 1, anything before the head? no
one node: 42
empty   : no list
```

Line by line. Walking forward along `right` gives the values in increasing order; walking backward along `left` from the last node gives them in decreasing order, which proves that *both* arrows of every node are right. The head is 1, and nothing stands before it. A one-node tree becomes a one-node list, and an empty tree becomes no list at all. Once converted, the nodes form a list and no longer a tree, so the tree's `clear` must not be used on them: it would follow `left` pointers backward and free nodes twice. The program frees them with `freeList`, a walk along `right`, and Valgrind confirms that all nine nodes (eight plus the single 42) are freed exactly once.

Cost. One inorder walk, $\Theta(n)$ time, and no new nodes at all: the only extra memory is the $O(h)$ call stack.

12.13.5 The Java and Python solutions

Java and Python have no `Node **`. The helper must still change two variables of its caller, so **Listing 12.35** and **Listing 12.36** put both in one small object, `Ends`, and pass that object to every call. All calls then share the same `head` and `prev`, exactly as the C++ calls share the two variables whose addresses they received.

Listing 12.35 `P1210ToDll.java`: Problem 12.10 in Java, with `head` and `prev` kept in one shared object.

```
// P1210ToDll.java - Problem 12.10: turn a binary tree into a doubly linked list
// in place: left acts as "previous" and right acts as "next".
public class P1210ToDll extends TreeKit {

    // Java has no Node**: the two references to change live in one object.
    static class Ends {
        Node head;                          // the first node of the list
        Node prev;                          // the last node linked so far
    }

    static void convertToDLLHelper(Node root, Ends e) {
        if (root == null) return;
        convertToDLLHelper(root.left, e);     // 1. every smaller value first
        if (e.prev == null) {
            e.head = root;                   // 2. nothing before: root is the head
        } else {
            e.prev.right = root;              //    link   prev -> root
        }
        root.left = e.prev;                   //    and    prev <- root
        e.prev = root;                        //    root is now the last node
        convertToDLLHelper(root.right, e);    // 3. every larger value after
    }

    static Node convertToDLL(Node root) {
        Ends e = new Ends();                  // head and prev both start as null
        convertToDLLHelper(root, e);
        return e.head;
    }

    static void preOrderDLL(Node head) {      // as given: follow the "next" links
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.right;
        }
    }

    static void printBackward(Node head) {    // go to the end, then follow "prev"
        Node tail = head;
        while (tail != null && tail.right != null) tail = tail.right;
        for (Node cur = tail; cur != null; cur = cur.left) {
            System.out.print(cur.data + " ");
        }
    }
```

Listing 12.35 (Continued) *P1210ToDll.java: Problem 12.10 in Java, with head and prev kept in one shared object.*

```
    public static void main(String[] args) {
        Node head = convertToDLL(buildSample());
        System.out.print("forward : ");
        preOrderDLL(head);
        System.out.print("\nbackward: ");
        printBackward(head);
        System.out.println("\nhead " + head.data + ", anything before the head? "
                + (head.left == null ? "no" : "yes"));
        Node h1 = convertToDLL(insert(null, 42));
        System.out.print("one node: ");
        preOrderDLL(h1);
        System.out.println("\nempty   : "
                + (convertToDLL(null) == null ? "no list" : "?"));
    }
}
```

Listing 12.36 `p1210_to_dll.py`: Problem 12.10 in Python.

```
# p1210_to_dll.py - Problem 12.10: turn a binary tree into a doubly linked list
# in place: left acts as "previous" and right acts as "next".
from tree_kit import *


class Ends:
    """The two references the helper must change: Python has no Node**."""
    def __init__(self):
        self.head = None                    # the first node of the list
        self.prev = None                    # the last node linked so far


def convert_to_dll_helper(root, e):
    if root is None:
        return
    convert_to_dll_helper(root.left, e)     # 1. every smaller value first
    if e.prev is None:
        e.head = root                       # 2. nothing before: root is the head
    else:
        e.prev.right = root                 #    link   prev -> root
    root.left = e.prev                      #    and    prev <- root
    e.prev = root                           #    root is now the last node
    convert_to_dll_helper(root.right, e)    # 3. every larger value after


def convert_to_dll(root):
    e = Ends()
    convert_to_dll_helper(root, e)
    return e.head


def pre_order_dll(head):                    # as given: follow the "next" links
    current = head
    while current is not None:
        print(current.data, end=" ")
        current = current.right
```

Listing 12.36 (Continued) *p1210_to_dll.py: Problem 12.10 in Python.*

```
def print_backward(head):                    # go to the end, then follow "prev"
    tail = head
    while tail is not None and tail.right is not None:
        tail = tail.right
    cur = tail
    while cur is not None:
        print(cur.data, end=" ")
        cur = cur.left


def main():
    head = convert_to_dll(build_sample())
    print("forward : ", end="")
    pre_order_dll(head)
    print("\nbackward: ", end="")
    print_backward(head)
    before = "no" if head.left is None else "yes"
    print(f"\nhead {head.data}, anything before the head? {before}")
    h1 = convert_to_dll(insert(None, 42))
    print("one node: ", end="")
    pre_order_dll(h1)
    print("\nempty   :", "no list" if convert_to_dll(None) is None else "?")


if __name__ == "__main__":
    main()
```

12.13.6 A tempting wrong solution

The given helper takes `Node **prev`, and the double pointer looks like needless trouble. Pass `prev` by value instead, and every call gets its own copy:

```
void convertToDLLHelper(Node *root, Node **head, Node *prev) {  // WRONG: a copy
    ...
    prev = root;                                  // changes the copy only
    convertToDLLHelper(root->right, head, prev);
}
```

`wrong/p1210_wrong.cpp` prints this and then a leak report:

```
forward : 6 10
backward: 10 6
head 6, anything before the head? no
one node: 42
empty   : no list

==27900==ERROR: LeakSanitizer: detected memory leaks
SUMMARY: AddressSanitizer: 144 byte(s) leaked in 6 allocation(s).
```

When the call on 1 sets its `prev` to 1, the change dies with that call; the call on 3, which called it, still has `prev == nullptr`, so it believes *it* is the first node, sets `head = 3` and `3->left = nullptr`, cutting 1 off. The same happens at 6, which becomes the head and cuts off the whole left side. On the right, 7, 8 and 9 are linked to each other but 10's own copy of `prev` is still 6, so 10 is linked directly after 6. The list is `6 10`, and six nodes (1, 3, 5, 7, 8, 9, at 24 bytes each: 144 bytes) are unreachable. A variable that the recursive calls must *share* has to be passed by pointer (or held in a shared object); a copy only carries information down, never back up.

12.13.7 Check yourself

> **CHECK YOURSELF.** Make the list *circular* (Section 9.6): the last node's `right` should point to the head, and the head's `left` to the last node. Which two lines must `convertToDLL` add after the helper returns? How would you then print the list once, without looping forever?
>
> *Answer.* After the helper, `prev` points to the last node, so: `if (head != nullptr) { head->left = prev; prev->right = head; }`. To print, use a `do`–`while` loop that starts at `head` and stops when it comes back to `head`: `Node *p = head; do { printf("%d ", p->data); p = p->right; } while (p != head);` (guarded by `head != nullptr`). Freeing must also stop at the head, for instance by first breaking the circle with `prev->right = nullptr`.

12.14 New Worked Problems on Doubly and Circular Lists

The ten problems so far walked trees. The next two go back to the lists of Chapter 9, one with two links per node and one with no end, because the same questions come up there in a simpler setting: which pointer must be saved before which assignment, and when does a walk stop.

12.14.1 Problem 12.11 — Reversing a doubly linked list in place

> **PROBLEM.** Pustaka Ceria keeps each member's reading history as a doubly linked list of book identifiers, oldest first, with a `head` and a `tail`. Reverse the list in place, so that the newest book comes first, without creating any node and without moving any book identifier from one node to another.

In our own words. In a singly linked list, reversal needed three pointers and a careful order (Problem 8.10). A doubly linked list makes it simpler, because every node already knows both of its neighbors. Reversing the list means that, for every node, "the node after me" and "the node before me" exchange roles. So the whole algorithm is: *at every node, swap `prev` and `next`; then swap `head` and `tail`.* **Figure 12.11** shows the list before and after.

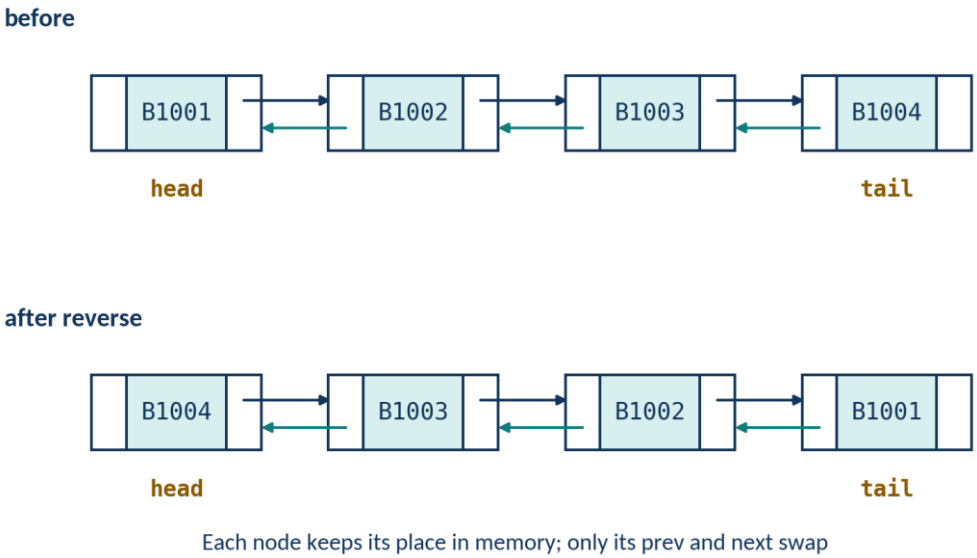


Figure 12.11 Problem 12.11: every node stays where it is in memory and keeps its book; only its two arrows exchange roles, and `head` and `tail` swap.

By hand. The only subtle point is how to move to the next node once its arrows are swapped. Take the list `B1001 B1002 B1003 B1004`. At `B1001`, first save `saved = next`, which is `B1002`; then swap, so that `next` becomes `nullptr` (the old `prev`) and `prev` becomes `B1002`. Now move to `saved`,

the *old* next, `B1002`. There, save `B1003`, swap, move to `B1003`; and so on. At `B1004`, `saved` is `nullptr` and the loop ends. Finally the old head `B1001` becomes the tail and the old tail `B1004` the head.

The solution. **Listing 12.37** is the C++ program; the Java and Python versions, **Listing 12.38** and **Listing 12.39**, print the same lines.

Listing 12.37 `p1211_reverse_dll.cpp`: Problem 12.11, swapping the two arrows of every node.

```
// p1211_reverse_dll.cpp - Problem 12.11: reverse a doubly linked list in place.
#include <cstdio>
#include <string>
using namespace std;

struct DNode {
    string book;
    DNode *prev;
    DNode *next;
};

struct DList {
    DNode *head = nullptr;
    DNode *tail = nullptr;
};

void pushBack(DList &list, const string &book) {
    DNode *fresh = new DNode{book, list.tail, nullptr};
    if (list.tail == nullptr) {
        list.head = fresh;                  // the list was empty
    } else {
        list.tail->next = fresh;
    }
    list.tail = fresh;
}

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 12.37 (Continued) *p1211_reverse_dll.cpp: Problem 12.11, swapping the two arrows of every node.*

```
void printBoth(const char *what, const DList &list) {
    printf("%-9s forward: ", what);
    for (DNode *p = list.head; p != nullptr; p = p->next) {
        printf("%s ", p->book.c_str());
    }
    printf("| backward: ");
    for (DNode *p = list.tail; p != nullptr; p = p->prev) {
        printf("%s ", p->book.c_str());
    }
    printf("\n");
}

void clear(DList &list) {
    while (list.head != nullptr) {
        DNode *next = list.head->next;
        delete list.head;
        list.head = next;
    }
    list.tail = nullptr;
}

int main() {
    DList history, one, none;
    for (const char *b : {"B1001", "B1002", "B1003", "B1004"}) {
        pushBack(history, b);
    }
    printBoth("start", history);
    reverse(history);
    printBoth("reversed", history);
    reverse(history);
    printBoth("twice", history);
    pushBack(one, "B2001");
    reverse(one);
    printBoth("one node", one);
    reverse(none);
    printBoth("empty", none);
    clear(history);
    clear(one);
    return 0;
}
```

```
start     forward: B1001 B1002 B1003 B1004 | backward: B1004 B1003 B1002 B1001
reversed  forward: B1004 B1003 B1002 B1001 | backward: B1001 B1002 B1003 B1004
twice     forward: B1001 B1002 B1003 B1004 | backward: B1004 B1003 B1002 B1001
one node  forward: B2001 | backward: B2001
empty     forward: | backward:
```

Every line prints the list twice, once along `next` from the head and once along `prev` from the tail, so that a single broken arrow would show. Reversing twice gives the original back; a one-node list and an empty list are handled by the same code with no special case, because the loop simply runs once or not at all, and swapping `head` with `tail` is harmless when they are equal. **Cost.** $\Theta(n)$ time, $O(1)$ extra memory.

Listing 12.38 `P1211ReverseDll.java`: Problem 12.11 in Java.

```java
// P1211ReverseDll.java - Problem 12.11: reverse a doubly linked list in place.
public class P1211ReverseDll {

    static class DNode {
        String book;
        DNode prev;
        DNode next;

        DNode(String book, DNode prev, DNode next) {
            this.book = book;
            this.prev = prev;
            this.next = next;
        }
    }

    static class DList {
        DNode head;
        DNode tail;
    }

    static void pushBack(DList list, String book) {
        DNode fresh = new DNode(book, list.tail, null);
        if (list.tail == null) {
            list.head = fresh;                    // the list was empty
        } else {
            list.tail.next = fresh;
        }
        list.tail = fresh;
    }

    static void reverse(DList list) {
        DNode cur = list.head;
        while (cur != null) {
            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 12.38 (Continued) *P1211ReverseDll.java: Problem 12.11 in Java.*

```
    static void printBoth(String what, DList list) {
        System.out.printf("%-9s forward: ", what);
        for (DNode p = list.head; p != null; p = p.next) {
            System.out.print(p.book + " ");
        }
        System.out.print("| backward: ");
        for (DNode p = list.tail; p != null; p = p.prev) {
            System.out.print(p.book + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        DList history = new DList(), one = new DList(), none = new DList();
        for (String b : new String[] {"B1001", "B1002", "B1003", "B1004"}) {
            pushBack(history, b);
        }
        printBoth("start", history);
        reverse(history);
        printBoth("reversed", history);
        reverse(history);
        printBoth("twice", history);
        pushBack(one, "B2001");
        reverse(one);
        printBoth("one node", one);
        reverse(none);
        printBoth("empty", none);
    }
}
```

Listing 12.39 `p1211_reverse_dll.py`: Problem 12.11 in Python.

```python
# p1211_reverse_dll.py - Problem 12.11: reverse a doubly linked list in place.


class DNode:
    def __init__(self, book, prev=None, nxt=None):
        self.book = book
        self.prev = prev
        self.next = nxt


class DList:
    def __init__(self):
        self.head = None
        self.tail = None


def push_back(lst, book):
    fresh = DNode(book, lst.tail, None)
    if lst.tail is None:
        lst.head = fresh                    # the list was empty
    else:
        lst.tail.next = fresh
    lst.tail = fresh


def reverse(lst):
    cur = lst.head
    while cur is not None:
        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"
    lst.head, lst.tail = lst.tail, lst.head  # 4. the two ends swap roles


def print_both(what, lst):
    print(f"{what:<9} forward: ", end="")
    p = lst.head
    while p is not None:
        print(p.book, end=" ")
        p = p.next
    print("| backward: ", end="")
    p = lst.tail
    while p is not None:
        print(p.book, end=" ")
        p = p.prev
```

Listing 12.39 (Continued) *p1211_reverse_dll.py: Problem 12.11 in Python.*

```
    print()


def main():
    history, one, none = DList(), DList(), DList()
    for b in ("B1001", "B1002", "B1003", "B1004"):
        push_back(history, b)
    print_both("start", history)
    reverse(history)
    print_both("reversed", history)
    reverse(history)
    print_both("twice", history)
    push_back(one, "B2001")
    reverse(one)
    print_both("one node", one)
    reverse(none)
    print_both("empty", none)


if __name__ == "__main__":
    main()
```

A tempting wrong solution. After swapping the arrows, it feels natural to move on with `cur = cur->next`, as every list walk in this book does. But `next` has just been swapped: it now holds the *old* `prev`. On the first node that is `nullptr`, and the loop stops after one node. `wrong/p1211_wrong.cpp` prints (its backward walk is limited to nine steps, so that it stops):

```
start     forward: B1001 B1002 B1003 B1004 | backward: B1004 B1003 B1002 B1001
reversed  forward: B1004 | backward: B1001 B1002 B1001 B1002 B1001 B1002 B1001 B1002 B1001
==27913==ERROR: LeakSanitizer: detected memory leaks
SUMMARY: AddressSanitizer: 144 byte(s) leaked in 3 allocation(s).
```

Only `B1001` was changed. The head is now `B1004`, whose `next` is still `nullptr`, so the forward walk prints one book. The backward walk starts at `B1001`, whose `prev` now points to `B1002`, whose `prev` still points to `B1001`: the two nodes point at each other, and a walk without the nine-step guard would never end. Finally, the program frees the list by following `next` from the head, reaches only `B1004`, and leaks the other three nodes. The rule is the one of Section 8.1: save the pointer you will need (`saved`), and use the saved copy, not the field you have just overwritten.

> **CHECK YOURSELF.** Write the reversal *recursively*: `DNode *reverseRec(DNode *cur)` swaps the arrows of `cur` and of every node after it and returns the new head. What is the base case?
>
> *Answer.* `if (cur == nullptr) return nullptr; DNode *old = cur->next; cur->next = cur->prev; cur->prev = old; if (old == nullptr) return cur; return reverseRec(old);`. The base cases are the empty list and the last node: the last node of the old list, after its swap, is the new head. (The caller must still set the new `tail` to the old head.) This recursion is a *tail call*: nothing happens after the recursive call returns, which is why the loop version is so easy to write (Section 10.11).

12.14.2 Problem 12.12 — The Josephus circle

> **PROBLEM.** The Pustaka Ceria reading club has received one advance copy of a new novel. The n members, numbered 1 to n, sit in a circle. Starting from member 1, they count around the circle, and every k-th member counted leaves the circle; the counting goes on from the next member. The last member left gets the copy. Using a circular linked list, print the order in which the members leave and the number of the survivor. Check the survivor with a recursive formula.

In our own words. This is the *Josephus problem*, named after the historian Flavius Josephus, and a classic of both circular lists and recursion [14]. A circular list (Section 9.6 builds the doubly linked kind; one link per node is enough here) is the natural model: the last member's `next` points back to member 1, so counting "around the circle" is simply following `next` forever. Removing a member is the unlink of Chapter 7: to remove node `out`, we need the node *before* it, `prev`, and set `prev->next = out->next`.

The one design decision is where `prev` starts. To remove the k-th member counted from member 1, we must stand on the member *before* it, which is $k-1$ steps after the member before member 1, the last node of the circle. So `buildCircle` returns the last node, and we start there. Then each round moves `prev` forward $k-1$ times, removes `prev->next`, and the next count starts from the member after the removed one, which is again `prev->next`. The circle is down to one member when that member's `next` is itself.

By hand, with $n = 7$ and $k = 3$, as in **Figure 12.12**. `prev` starts at 7. Two steps (to 1, to 2) and 3 leaves. From 2, two steps (4, 5) and 6 leaves. From 5, two steps (7, 1) and 2 leaves. From 1, two steps (4, 5) and 7 leaves. From 5, two steps (1, 4) and 5 leaves. From 4, two steps (1, 4) and 1 leaves. Member 4 is alone: the survivor.

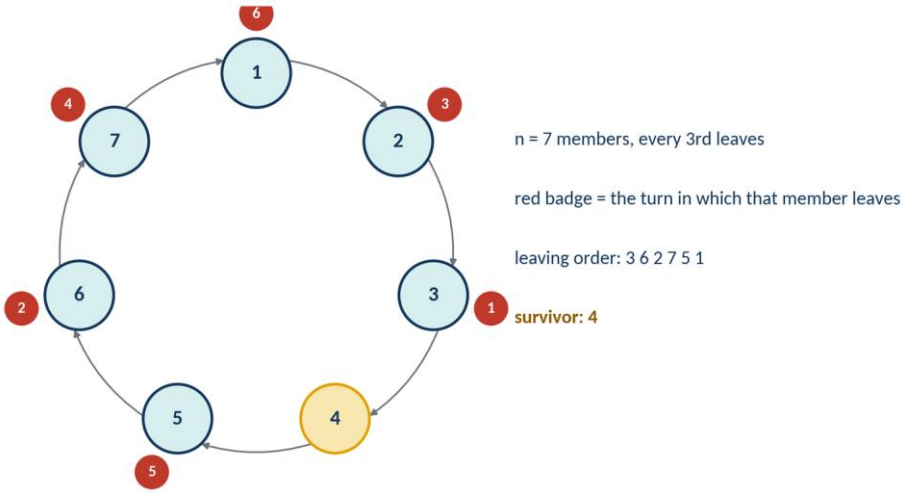


Figure 12.12 The Josephus circle for $n = 7$, $k = 3$: the red badge beside each member is the round in which that member leaves. Member 4 is never counted third and keeps the book.

The formula. Number the members from 0 instead of 1, and let $J(n)$ be the position of the survivor in a circle of n members when counting starts at position 0. With one member, the survivor is at position 0. With n members, the first member to leave is at position $k-1$ (taken modulo n), and the count continues from position k with a circle of $n-1$ members. That smaller circle is the same problem, just renumbered: its position 0 is our position k, its position 1 is our position $k+1$, and so on around. So the survivor of the smaller circle, at *its* position $J(n-1)$, is at *our* position $J(n-1)+k$, taken modulo n:

$$J(1) = 0, \qquad J(n) = (J(n-1) + k) \bmod n \quad (n \geq 2). \tag{12.8}$$

For $n = 7$ and $k = 3$: $J(1) = 0$, $J(2) = 3 \bmod 2 = 1$, $J(3) = 4 \bmod 3 = 1$, $J(4) = 4 \bmod 4 = 0$, $J(5) = 3 \bmod 5 = 3$, $J(6) = 6 \bmod 6 = 0$, $J(7) = 3 \bmod 7 = 3$. Position 3 counted from 0 is member 4, as the circle said. Equation (12.8) turns an $O(nk)$ simulation into $O(n)$ arithmetic, and it is a fine example of Chapter 10's central idea: solve the problem by solving a smaller copy of itself.

The solution. Listing 12.40 runs the circle and the formula for five cases and compares them.

Listing 12.40 `p1212_josephus.cpp`: Problem 12.12, a circular list and the recursive formula, compared.

```
// p1212_josephus.cpp - Problem 12.12: who is left when every k-th member of a
// circle leaves? A circular list, and a recursive formula to check it.
#include <cstdio>

struct CNode {
    int member;
    CNode *next;
};

// Build the circle 1, 2, ..., n and return its LAST node (whose next is 1).
CNode *buildCircle(int n) {
    CNode *first = new CNode{1, nullptr};
    CNode *last = first;
    for (int m = 2; m <= n; m++) {
        last->next = new CNode{m, nullptr};
        last = last->next;
    }
    last->next = first;                          // close the circle
    return last;
}

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;
}

// J(1) = 0 and J(n) = (J(n-1) + k) mod n, counting members from 0.
int josephusFormula(int n, int k) {
    if (n == 1) return 0;
    return (josephusFormula(n - 1, k) + k) % n;
}
```

Listing 12.40 (Continued) *p1212_josephus.cpp: Problem 12.12, a circular list and the recursive formula, compared.*

```
int main() {
    const int cases[][2] = {{7, 3}, {5, 2}, {6, 1}, {1, 4}, {10, 3}};
    for (const auto &c : cases) {
        int a = josephusList(c[0], c[1]);
        int b = josephusFormula(c[0], c[1]) + 1;  // +1: members count from 1
        printf("  formula says %d: %s\n", b, a == b ? "agree" : "DISAGREE");
    }
    return 0;
}
```

```
n=7, k=3, leaving in order: 3 6 2 7 5 1 | survivor 4
  formula says 4: agree
n=5, k=2, leaving in order: 2 4 1 5 | survivor 3
  formula says 3: agree
n=6, k=1, leaving in order: 1 2 3 4 5 | survivor 6
  formula says 6: agree
n=1, k=4, leaving in order: | survivor 1
  formula says 1: agree
n=10, k=3, leaving in order: 3 6 9 2 7 1 8 5 10 | survivor 4
  formula says 4: agree
```

The first case is the one computed by hand. $k = 1$ removes the members in order and leaves the last; $n = 1$ leaves the only member without a single count; and in every case the list and Equation (12.8) agree. The C++ program frees every node it removes and, finally, the survivor, so Valgrind reports no leak. **Cost.** The list simulation does $k - 1$ steps for each of the $n - 1$ removals: $\Theta(nk)$. The formula makes n calls: $\Theta(n)$ time and $O(n)$ stack.

Listing 12.41 and **Listing 12.42** are the Java and Python versions; both print exactly the same lines.

Listing 12.41 `P1212Josephus.java`: Problem 12.12 in Java.

```java
// P1212Josephus.java - Problem 12.12: who is left when every k-th member of a
// circle leaves? A circular list, and a recursive formula to check it.
public class P1212Josephus {

    static class CNode {
        int member;
        CNode next;

        CNode(int member) {
            this.member = member;
        }
    }

    // Build the circle 1, 2, ..., n and return its LAST node (whose next is 1).
    static CNode buildCircle(int n) {
        CNode first = new CNode(1);
        CNode last = first;
        for (int m = 2; m <= n; m++) {
            last.next = new CNode(m);
            last = last.next;
        }
        last.next = first;                          // close the circle
        return last;
    }

    static int josephusList(int n, int k) {
        CNode prev = buildCircle(n);  // the node BEFORE the next one counted
        System.out.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
            System.out.print(" " + out.member);
        }
        int survivor = prev.member;
        System.out.println(" | survivor " + survivor);
        return survivor;
    }

    // J(1) = 0 and J(n) = (J(n-1) + k) mod n, counting members from 0.
    static int josephusFormula(int n, int k) {
        if (n == 1) return 0;
        return (josephusFormula(n - 1, k) + k) % n;
    }
```

Listing 12.41 (Continued) *P1212Josephus.java: Problem 12.12 in Java.*

```java
    public static void main(String[] args) {
        int[][] cases = {{7, 3}, {5, 2}, {6, 1}, {1, 4}, {10, 3}};
        for (int[] c : cases) {
            int a = josephusList(c[0], c[1]);
            // +1: members are numbered from 1
            int b = josephusFormula(c[0], c[1]) + 1;
            System.out.printf("  formula says %d: %s%n", b,
                    a == b ? "agree" : "DISAGREE");
        }
    }
}
```

Listing 12.42 `p1212_josephus.py`: Problem 12.12 in Python.

```
# p1212_josephus.py - Problem 12.12: who is left when every k-th member of a
# circle leaves? A circular list, and a recursive formula to check it.


class CNode:
    def __init__(self, member):
        self.member = member
        self.next = None


def build_circle(n):
    """Build the circle 1, 2, ..., n and return its LAST node (whose next is 1)."""
    first = CNode(1)
    last = first
    for m in range(2, n + 1):
        last.next = CNode(m)
        last = last.next
    last.next = first                           # close the circle
    return last


def josephus_list(n, k):
    prev = build_circle(n)  # the node BEFORE the next one counted
    print(f"n={n}, k={k}, leaving in order:", end="")
    while prev.next is not prev:                # more than one member remains
        for _ in range(k - 1):                  # count k-1 members
            prev = prev.next
        out = prev.next                         # the k-th member leaves
        prev.next = out.next                    # link around it
        print(f" {out.member}", end="")
    survivor = prev.member
    prev.next = None                            # break the last self-loop
    print(f" | survivor {survivor}")
    return survivor


def josephus_formula(n, k):
    """J(1) = 0 and J(n) = (J(n-1) + k) mod n, counting members from 0."""
    if n == 1:
        return 0
    return (josephus_formula(n - 1, k) + k) % n
```

Listing 12.42 (Continued) *p1212_josephus.py: Problem 12.12 in Python.*

```
def main():
    for n, k in ((7, 3), (5, 2), (6, 1), (1, 4), (10, 3)):
        a = josephus_list(n, k)
        b = josephus_formula(n, k) + 1          # +1: members are numbered from 1
        print(f"  formula says {b}: {'agree' if a == b else 'DISAGREE'}")


if __name__ == "__main__":
    main()
```

The Python version sets `prev.next = None` at the end. A single node whose `next` is itself is a *reference cycle*; Python's garbage collector would reclaim it eventually, but breaking the cycle lets reference counting free it at once [10].

A tempting wrong solution. Start `prev` at member 1 instead of at the member before it: `CNode *prev = buildCircle(n)->next;`. Every count is now one member late, and `wrong/p1212_wrong.cpp` prints:

```
n=7, k=3, leaving in order: 4 7 3 1 6 2 | survivor 5
  formula says 4: DISAGREE
n=5, k=2, leaving in order: 3 5 2 1 | survivor 4
  formula says 3: DISAGREE
n=6, k=1, leaving in order: 2 3 4 5 6 | survivor 1
  formula says 6: DISAGREE
n=1, k=4, leaving in order: | survivor 1
  formula says 1: agree
n=10, k=3, leaving in order: 4 7 10 3 8 2 9 6 1 | survivor 5
  formula says 4: DISAGREE
```

The program runs, prints plausible sequences and never crashes; only the independent check exposes it, and only for $n \geq 2$. The case $k = 1$ makes the error plainest: "every first member leaves" should remove 1 first, and it removes 2. The lesson is the value of a second, independent computation of the same answer.

CHECK YOURSELF. Use Equation (12.8) to find the survivor for $n = 10$, $k = 2$. Then find it for $n = 16$ and for $n = 17$, $k = 2$, and guess the pattern.

Answer. For $k = 2$: $J(1..10) = 0,0,2,0,2,4,6,0,2,4$, so the survivor of 10 is member 5. $J(16) = 0$ (member 1) and $J(17) = 2$ (member 3). The pattern for $k = 2$: write $n = 2^m + \ell$ with $0 \leq \ell < 2^m$; the survivor is member $2\ell + 1$ [14]. For $n = 10 = 8 + 2$ that is 5.

12.15 A New Worked Problem on Recursion

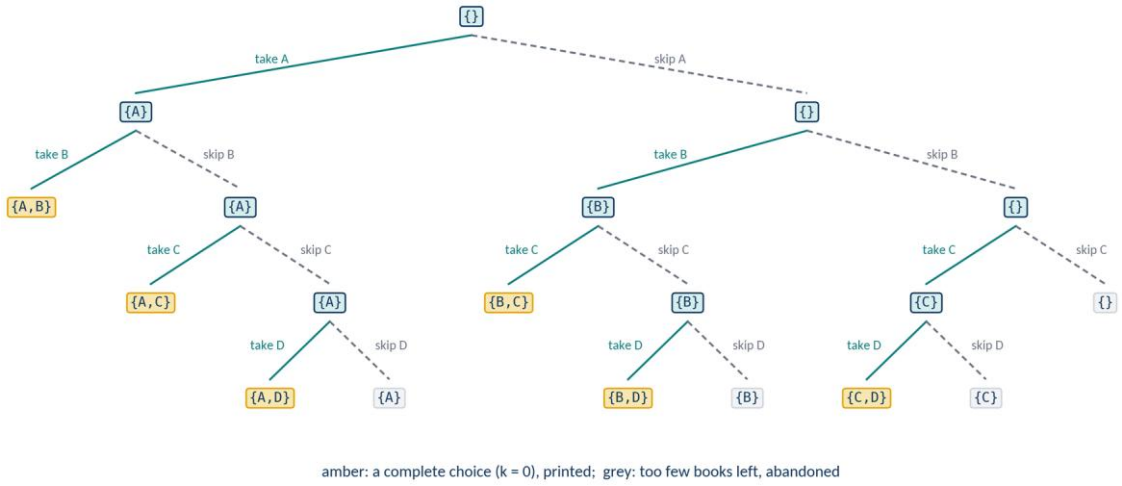
12.15.1 Problem 12.13 — Choosing books

PROBLEM. A member may borrow three of the five new books `B1001` to `B1005`. Write a recursive function that prints every possible choice of three books, each choice once, with the books of each choice in catalog order, and count the choices. Check the count with the recursive formula for the binomial coefficient $\binom{n}{k}$.

In our own words. Order does not matter (taking `B1001` and then `B1002` is the same choice as the other way round), so we want *combinations*. The recursive idea is a yes-or-no decision about one book at a time. Look at the first book not yet decided, `books[start]`. Every choice either **takes** it, and then needs $k - 1$ more books from the ones after it, or **skips** it, and then needs all k books from the ones after it. The two branches never produce the same choice (one contains the book, the other does not) and together they produce every choice. Counting the two branches gives the famous recurrence of Pascal's triangle:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \qquad \binom{n}{0} = \binom{n}{n} = 1. \tag{12.9}$$

The base cases. If $k = 0$, the choice is complete: print it. If fewer than k books remain, no choice can be completed down this branch: give up (this is a dead end, and the early return is called *pruning*). **Figure 12.13** draws the whole recursion for a smaller case, two books out of four, so that it fits on a page.



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.

Backtracking. The choices made so far are kept in one shared list, `picked`. Before the "take" branch the function appends the book; after that branch returns, it *removes* the book again, so that the "skip" branch starts from exactly the state it would have had if the book had never been taken. This append, explore, remove pattern is called *backtracking*, and the removal is the step that is easiest to forget.

Listing 12.43 is the C++ program for Problem 12.13.

Listing 12.43 `p1213_choose_books.cpp`: Problem 12.13, every choice of three books by take-or-skip recursion with backtracking.

```cpp
// p1213_choose_books.cpp - Problem 12.13: all ways to choose k of n books.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

// Choose k more books from books[start..]; picked holds the choices made so far.
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]
}

// C(n, k) = C(n-1, k-1) + C(n-1, k): the same two branches, only counted.
long long binom(int n, int k) {
    if (k == 0 || k == n) return 1;
    return binom(n - 1, k - 1) + binom(n - 1, k);
}

int main() {
    vector<string> books = {"B1001", "B1002", "B1003", "B1004", "B1005"};
    vector<string> picked;
    int ways = 0;
    choose(books, 0, 3, picked, ways);
    printf("ways printed: %d, C(5,3) = %lld, C(10,4) = %lld, C(20,10) = %lld\n",
           ways, binom(5, 3), binom(10, 4), binom(20, 10));
    return 0;
}
```

```
 1: B1001 B1002 B1003
 2: B1001 B1002 B1004
 3: B1001 B1002 B1005
 4: B1001 B1003 B1004
 5: B1001 B1003 B1005
 6: B1001 B1004 B1005
 7: B1002 B1003 B1004
 8: B1002 B1003 B1005
 9: B1002 B1004 B1005
10: B1003 B1004 B1005
ways printed: 10, C(5,3) = 10, C(10,4) = 210, C(20,10) = 184756
```

The ten choices come out in catalog order, because the “take” branch is explored before the “skip” branch, and the count agrees with $\binom{5}{3} = 10$. The same program also evaluates $\binom{10}{4} = 210$ and $\binom{20}{10} = 184\,756$ by Equation (12.9). That last number hides a warning from Chapter 10: the recursive `binom` returns 1 at each of its leaves, so to add up to 184 756 it makes $2 \cdot 184\,756 - 1 = 369\,511$ calls, many of them repeating the same subproblem, exactly like the naive Fibonacci of Section 10.5. Storing each $\binom{i}{j}$ once computed, in a table, would reduce that to about $n \cdot k$ calls.

Listing 12.44 and **Listing 12.45** are the Java and Python versions; both print exactly the same lines.

Listing 12.44 `P1213ChooseBooks.java`: Problem 12.13 in Java.

```java
// P1213ChooseBooks.java - Problem 12.13: all ways to choose k of n books.
import java.util.ArrayList;
import java.util.List;

public class P1213ChooseBooks {

    static int ways = 0;  // how many choices were printed

    // Choose k more books from books[start..]; picked holds the choices so far.
    static void choose(String[] books, int start, int k, List<String> picked) {
        if (k == 0) {                                  // a complete choice: print it
            ways++;
            System.out.printf("%2d:", ways);
            for (String b : picked) System.out.print(" " + b);
            System.out.println();
            return;
        }
        int left = books.length - start;               // books not yet considered
        if (left < k) return;                          // too few left: a dead end
        picked.add(books[start]);                      // branch 1: take books[start]
        choose(books, start + 1, k - 1, picked);
        picked.remove(picked.size() - 1);              // undo that choice
        choose(books, start + 1, k, picked);           // branch 2: skip books[start]
    }

    // C(n, k) = C(n-1, k-1) + C(n-1, k): the same two branches, only counted.
    static long binom(int n, int k) {
        if (k == 0 || k == n) return 1;
        return binom(n - 1, k - 1) + binom(n - 1, k);
    }

    public static void main(String[] args) {
        String[] books = {"B1001", "B1002", "B1003", "B1004", "B1005"};
        choose(books, 0, 3, new ArrayList<>());
        System.out.printf("ways printed: %d, C(5,3) = %d, C(10,4) = %d, "
                + "C(20,10) = %d%n",
                ways, binom(5, 3), binom(10, 4), binom(20, 10));
    }
}
```

Listing 12.45 `p1213_choose_books.py`: Problem 12.13 in Python.

```python
# p1213_choose_books.py - Problem 12.13: all ways to choose k of n books.


def choose(books, start, k, picked, ways):
    """Choose k more books from books[start:]; picked holds the choices so far.
    ways is a one-element list, so that every call adds to the same counter."""
    if k == 0:                                  # a complete choice: print it
        ways[0] += 1
        print(f"{ways[0]:2d}:", " ".join(picked))
        return
    left = len(books) - start                   # books not yet considered
    if left < k:                                # too few left: a dead end
        return
    picked.append(books[start])                 # branch 1: take books[start]
    choose(books, start + 1, k - 1, picked, ways)
    picked.pop()                                # undo that choice
    choose(books, start + 1, k, picked, ways)   # branch 2: skip books[start]


def binom(n, k):
    """C(n, k) = C(n-1, k-1) + C(n-1, k): the same two branches, only counted."""
    if k == 0 or k == n:
        return 1
    return binom(n - 1, k - 1) + binom(n - 1, k)


def main():
    books = ["B1001", "B1002", "B1003", "B1004", "B1005"]
    ways = [0]
    choose(books, 0, 3, [], ways)
    print(f"ways printed: {ways[0]}, C(5,3) = {binom(5, 3)}, "
          f"C(10,4) = {binom(10, 4)}, C(20,10) = {binom(20, 10)}")


if __name__ == "__main__":
    main()
```

The counter is shared by all calls: a reference `int &ways` in C++, a `static` field in Java, a one-element list in Python.

Cost. There are $\binom{n}{k}$ choices, each printed in $O(k)$, and with the pruning the dead ends add at most a constant factor; no algorithm that prints every choice can do better.

A tempting wrong solution. Forget the `pop_back` after the "take" branch. `wrong/p1213_wrong.cpp` prints ten lines, but they are nonsense:

```
 1: B1001 B1002 B1003
 2: B1001 B1002 B1003 B1004
 3: B1001 B1002 B1003 B1004 B1005
 4: B1001 B1002 B1003 B1004 B1005 B1003 B1004
```

(and so on, each line longer than the one before). The count of complete choices is still 10, because the counting depends only on k and `start`, which are passed by value and restored automatically when a call returns. The list `picked` is *shared*, so nothing restores it but our own `pop_back`. Every book ever taken stays in it. The general rule: whatever a recursive call receives by value is undone for free; whatever it changes in shared state, it must undo itself.

> **CHECK YOURSELF.** How many calls of `choose` are made for $n = 4$, $k = 2$ (count them in **Figure 12.13**)? And what must change in `choose` to print every choice of *up to* k books, including the empty choice?
>
> *Answer.* The figure has 16 boxes: 16 calls, of which 6 print and 3 are dead ends. For "up to k books", print `picked` at *every* call, before the two branches, and stop the recursion when $k = 0$

or when no books remain: that prints $\binom{4}{0} + \binom{4}{1} + \binom{4}{2} = 1 + 4 + 6 = 11$ choices for $n = 4$, $k = 2$.

12.16 New Worked Problems on Trees

12.16.1 Problem 12.14 — Is it really a binary search tree?

PROBLEM. A tree of catalog numbers has been built by hand, not by `insert`. Write a function that decides whether it is a binary search tree, with no two equal values.

In our own words. We must check Equation (12.3) at every node, for *all* descendants, not only the children. The trick that makes this one walk instead of many is to pass limits down. The root may hold any value. Everything in its left subtree must be below the root's value; everything in its right subtree must be above it. Going down, each call knows an open interval $(low, high)$ that its value must lie in: going left, the node's value becomes the new `high`; going right, it becomes the new `low`. The empty tree is a search tree (base case: true).

By hand. **Figure 12.14** shows a tree built to fool careless checks: 6 with children 3 and 10, and 3 with children 1 and 8. Every node is correct *compared with its own children*: $1 < 3 < 8$ and $3 < 6 < 10$. But 8 is in the left subtree of 6, so it must be below 6, and it is not. With limits, the call on 8 receives the interval (3,6): it came left from 6 (so $high = 6$) and then right from 3 (so $low = 3$), and $8 \notin (3,6)$.

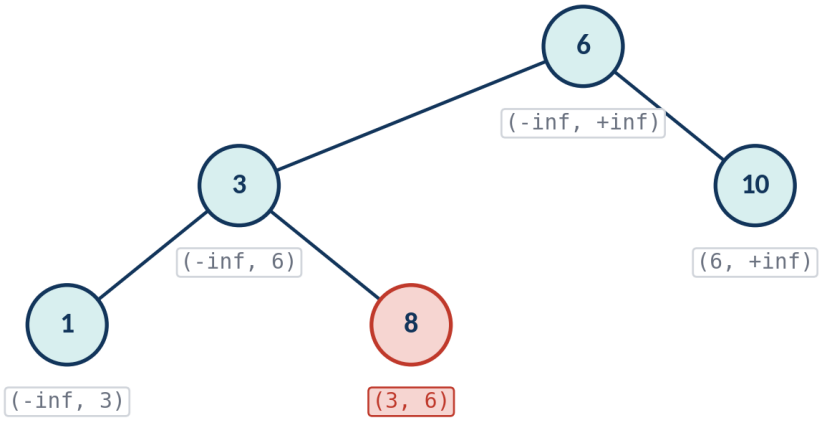


Figure 12.14 The interval each call of `inRange` receives. Every parent–child pair is in order, but the value 8 lies outside its interval (3,6), so the tree is not a binary search tree.

Listing 12.46 is the C++ program for Problem 12.14.

Listing 12.46 `p1214_is_bst.cpp`: Problem 12.14, every node checked against the interval inherited from its ancestors.

```cpp
// p1214_is_bst.cpp - Problem 12.14: is this binary tree a binary search tree?
#include <climits>
#include "tree_kit.hpp"

// Every value in this subtree must lie strictly between low and high.
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
}

Node *make(int data, Node *left, Node *right) {        // build a tree by hand
    return new Node{data, left, right};
}

int main() {
    Node *sample = nullptr;
    buildSample(&sample);
    // 6 with left child 3 (children 1 and 8) and right child 10:
    // the 8 is on the wrong side of the 6.
    Node *sneaky = make(6, make(3, make(1, nullptr, nullptr),
                                   make(8, nullptr, nullptr)),
                           make(10, nullptr, nullptr));
    // 6 whose left child is another 6: equal values are not allowed.
    Node *twin = make(6, make(6, nullptr, nullptr), nullptr);
    Node *empty = nullptr;
    printf("sample tree  : %s\n", isBST(sample) ? "BST" : "not a BST");
    printf("sneaky tree  : %s\n", isBST(sneaky) ? "BST" : "not a BST");
    printf("twin sixes   : %s\n", isBST(twin) ? "BST" : "not a BST");
    printf("empty tree   : %s\n", isBST(empty) ? "BST" : "not a BST");
    printf("sneaky inorder: ");
    inOrder(sneaky);
    printf("\n");
    clear(&sample);
    clear(&sneaky);
    clear(&twin);
    return 0;
}
```

```
sample tree  : BST
sneaky tree  : not a BST
twin sixes   : not a BST
empty tree   : BST
sneaky inorder: 1 3 8 6 10
```

The sample tree passes; the sneaky tree fails, as does a tree with two equal values (the intervals are open, so equality is refused); the empty tree passes. The last line prints the inorder of the sneaky tree, `1 3 8 6 10`, which is not increasing: that is a second, equivalent test, and the check-yourself below uses it. The limits start at `LLONG_MIN` and `LLONG_MAX`, which are outside the range of any `int` value, so they never reject a real value; Java uses `Long.MIN_VALUE` and `Long.MAX_VALUE`, and Python `-math.inf` and `math.inf`. **Cost.** $\Theta(n)$ in the worst case: every node is checked once.

Listing 12.47 and **Listing 12.48** are the Java and Python versions; both print exactly the same lines.

Listing 12.47 `P1214IsBst.java`: Problem 12.14 in Java.

```java
// P1214IsBst.java - Problem 12.14: is this binary tree a binary search tree?
public class P1214IsBst extends TreeKit {

    // Every value in this subtree must lie strictly between low and high.
    static boolean inRange(Node root, long low, long high) {
        if (root == null) 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
    }

    static boolean isBST(Node root) {
        return inRange(root, Long.MIN_VALUE, Long.MAX_VALUE);   // no limits yet
    }

    static Node make(int data, Node left, Node right) {   // build a tree by hand
        return new Node(data, left, right);
    }

    static String verdict(Node root) {
        return isBST(root) ? "BST" : "not a BST";
    }

    public static void main(String[] args) {
        Node sample = buildSample();
        // 6 with left child 3 (children 1 and 8) and right child 10:
        // the 8 is on the wrong side of the 6.
        Node sneaky = make(6, make(3, make(1, null, null), make(8, null, null)),
                make(10, null, null));
        // 6 whose left child is another 6: equal values are not allowed.
        Node twin = make(6, make(6, null, null), null);
        System.out.println("sample tree  : " + verdict(sample));
        System.out.println("sneaky tree  : " + verdict(sneaky));
        System.out.println("twin sixes   : " + verdict(twin));
        System.out.println("empty tree   : " + verdict(null));
        System.out.print("sneaky inorder: ");
        inOrder(sneaky);
        System.out.println();
    }
}
```

Listing 12.48 `p1214_is_bst.py`: Problem 12.14 in Python.

```python
# p1214_is_bst.py - Problem 12.14: is this binary tree a binary search tree?
import math
from tree_kit import *


def in_range(root, low, high):
    """Every value in this subtree must lie strictly between low and high."""
    if root is None:
        return True
    if root.data <= low or root.data >= high:
        return False
    return (in_range(root.left, low, root.data) and      # left: below this value
            in_range(root.right, root.data, high))       # right: above this value


def is_bst(root):
    return in_range(root, -math.inf, math.inf)            # no limits at the root


def main():
    sample = build_sample()
    # 6 with left child 3 (children 1 and 8) and right child 10:
    # the 8 is on the wrong side of the 6.
    sneaky = Node(6, Node(3, Node(1), Node(8)), Node(10))
    # 6 whose left child is another 6: equal values are not allowed.
    twin = Node(6, Node(6))
    verdict = lambda t: "BST" if is_bst(t) else "not a BST"
    print("sample tree  :", verdict(sample))
    print("sneaky tree  :", verdict(sneaky))
    print("twin sixes   :", verdict(twin))
    print("empty tree   :", verdict(None))
    print("sneaky inorder: ", end="")
    in_order(sneaky)
    print()


if __name__ == "__main__":
    main()
```

A tempting wrong solution. Compare each node only with its own children, and recurse:

```
if (root->left != nullptr && root->left->data >= root->data) return false;
if (root->right != nullptr && root->right->data <= root->data) return false;
return isBST(root->left) && isBST(root->right);
```

`wrong/p1214_wrong.cpp` answers `sneaky tree  : BST`. This check only enforces the rule for parent and child, and the rule of Equation (12.3) is about ancestor and descendant. The consequence is not academic: in the sneaky tree, `search(root, 8)` goes right at 6 (since $8 > 6$), finds 10, goes left into `nullptr`, and reports that 8 is absent, although it is in the tree.

> **CHECK YOURSELF.** Write the inorder test: walk the tree in inorder and check that every value is larger than the previously visited one. What must the "previous value" be, and how is it shared between calls?
>
> *Answer.* The previous value must be shared by all calls and survive their returns, exactly like `prev` in Problem 12.10: pass a `long long *prev` (or `long long &prev`), starting at `LLONG_MIN`. At each node, after the left subtree: `if (root->data <= *prev) return false; *prev = root->data;` then check the right subtree. It visits the values in the order 1, 3, 8, 6, and fails at 6 because $6 \leq 8$.

12.16.2 Problem 12.15 — The lowest common ancestor

> **PROBLEM.** In a binary search tree that contains both values a and b, find their *lowest common ancestor*: the deepest node that has both a and b in its subtree (a node counts as being in its own subtree).

In our own words. Walk down from the root. As long as a and b are both smaller than the current node, both are in its left subtree, and so is their lowest common ancestor: go left. As long as both are larger, go right. At the first node where they are *not* on the same side (one is smaller and one larger, or one of them *is* the node) their paths part, and that node is the answer. **Figure 12.15** shows the case $a = 5$, $b = 9$: at 6, 5 goes left and 9 goes right, so the answer is 6 at once.

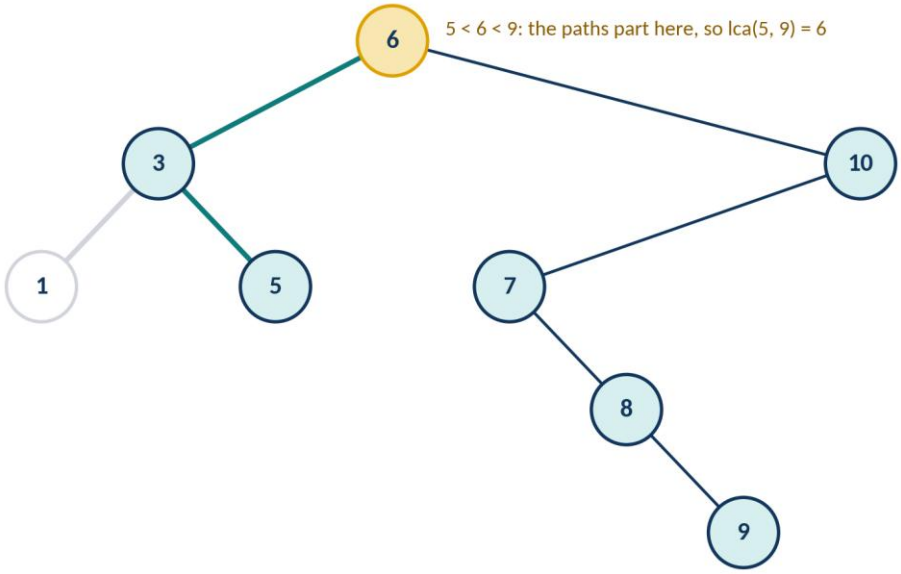


Figure 12.15 The paths from the root to 5 (teal) and to 9 (navy) share only the root: since $5 < 6 < 9$, the two values part at 6, which is their lowest common ancestor.

By hand on the other pairs: for 1 and 5, both are below 6 (go left), and at 3 they part: 3. For 8 and 9, both are above 6, then both below 10, then both above 7; at 8 one of them *is* the node: 8. For 7 and 9: 7. For 1 and 10: they part at the root: 6. For 9 and 9: the node 9 itself.

Listing 12.49 is the C++ program for Problem 12.15.

Listing 12.49 `p1215_lca.cpp`: Problem 12.15, one walk down until the two values part.

```cpp
// p1215_lca.cpp - Problem 12.15: the lowest common ancestor of two BST values.
#include "tree_kit.hpp"

// The deepest node with both a and b in its subtree (both must be in the tree).
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
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    const int pairs[][2] = {{1, 5}, {5, 9}, {8, 9}, {7, 9}, {1, 10}, {9, 9}};
    for (const auto &p : pairs) {
        printf("lca(%d, %d) = %d\n", p[0], p[1], lca(root, p[0], p[1])->data);
    }
    clear(&root);
    return 0;
}
```

```
lca(1, 5) = 3
lca(5, 9) = 6
lca(8, 9) = 8
lca(7, 9) = 7
lca(1, 10) = 6
lca(9, 9) = 9
```

All six answers agree with the hand computation. The function walks one path, so it costs $O(h)$; in a general binary tree, without the order of a search tree to steer by, finding the lowest common ancestor needs a walk of the whole tree, $\Theta(n)$.

Listing 12.50 and **Listing 12.51** are the Java and Python versions; both print exactly the same lines.

Listing 12.50 `P1215Lca.java`: Problem 12.15 in Java.

```java
// P1215Lca.java - Problem 12.15: the lowest common ancestor of two BST values.
public class P1215Lca extends TreeKit {

    // The deepest node with both a and b in its subtree (both must be present).
    static Node lca(Node root, int a, int b) {
        if (root == null) return null;
        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
    }

    public static void main(String[] args) {
        Node root = buildSample();
        int[][] pairs = {{1, 5}, {5, 9}, {8, 9}, {7, 9}, {1, 10}, {9, 9}};
        for (int[] p : pairs) {
            Node answer = lca(root, p[0], p[1]);
            System.out.printf("lca(%d, %d) = %d%n", p[0], p[1], answer.data);
        }
    }
}
```

Listing 12.51 `p1215_lca.py`: Problem 12.15 in Python.

```python
# p1215_lca.py - Problem 12.15: the lowest common ancestor of two values in a BST.
from tree_kit import *


def lca(root, a, b):
    """The deepest node with both a and b in its subtree (both must be in it)."""
    if root is None:
        return None
    if a < root.data and b < root.data:          # both on the left
        return lca(root.left, a, b)
    if a > root.data and b > root.data:          # both on the right
        return lca(root.right, a, b)
    return root                    # they part here, or one of them is here


def main():
    root = build_sample()
    for a, b in ((1, 5), (5, 9), (8, 9), (7, 9), (1, 10), (9, 9)):
        print(f"lca({a}, {b}) = {lca(root, a, b).data}")


if __name__ == "__main__":
    main()
```

A tempting wrong solution. Steer by one value only, as `search` does: `if (a < root->data) go left; if (a > root->data) go right;`. `wrong/p1215_wrong.cpp` answers `lca(1, 5) = 1`, `lca(5, 9) = 5` and `lca(1, 10) = 1`, and gets the other three right by luck. Steering by a alone simply walks to a and returns it; the answer is right only when b happens to lie below a. The walk must continue only while *both* values point the same way.

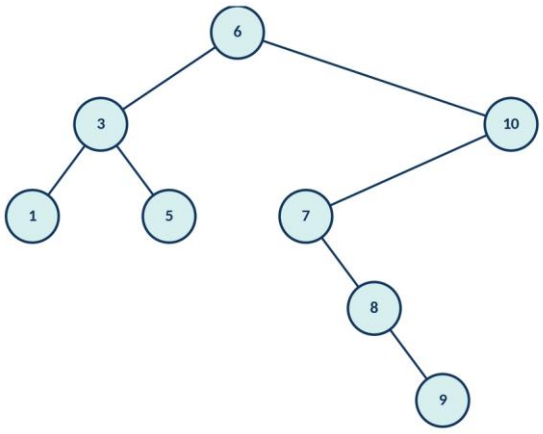
> **CHECK YOURSELF.** Use the lowest common ancestor to compute the number of edges between two nodes, $dist(a, b)$, using the depth $d(x)$ of a node. What is $dist(5,9)$?
>
> *Answer.* $dist(a, b) = d(a) + d(b) - 2\,d(lca(a, b))$: go up from a to the common ancestor and down to b. For 5 and 9: $2 + 4 - 2 \cdot 0 = 6$ edges (5, 3, 6, 10, 7, 8, 9).

12.16.3 Problem 12.16 — Printing a tree level by level

> **PROBLEM.** Print the tree one level per line, each line starting with its level number, using one walk of the tree. Also report the width of the widest level.

In our own words. Printing every level with Problem 12.2 cost up to $\Theta(n^2)$. The level-order traversal of Section 11.6.3 uses a queue (Chapter 5): take a node from the front, print it, put its children at the back; so all of level d leaves the queue before any node of level $d+1$. To know *where a level ends*, notice that at the start of each level the queue holds exactly that level, so its size at that moment is the level's width: take exactly that many nodes, and the level is done.

By hand. **Figure 12.16** follows 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.

At the start the queue holds `6`, width 1: take 6, enqueue 3 and 10. The queue now holds `3 10`, width 2: take both, enqueue 1 and 5 (children of 3) and 7 (child of 10). Width 3: take `1 5 7`, enqueue 8. Then `8`, enqueue 9. Then `9`, and the queue is empty.

Listing 12.52 is the C++ program for Problem 12.16.

Listing 12.52 `p1216_levels.cpp`: Problem 12.16, a level-order walk that records the width of each level.

```cpp
// p1216_levels.cpp - Problem 12.16: print a tree level by level with one queue.
#include <queue>
#include "tree_kit.hpp"

// Breadth-first: all of level d leave the queue before any node of level d+1.
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;
}

int main() {
    Node *root = nullptr;
    buildSample(&root);
    printf("widest level has %d nodes\n", printLevels(root));
    Node *full = nullptr;
    insertAll(&full, {8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15});
    printf("widest level has %d nodes\n", printLevels(full));
    clear(&root);
    clear(&full);
    return 0;
}
```

```
level 0: 6
level 1: 3 10
level 2: 1 5 7
level 3: 8
level 4: 9
widest level has 3 nodes
level 0: 8
level 1: 4 12
level 2: 2 6 10 14
level 3: 1 3 5 7 9 11 13 15
widest level has 8 nodes
```

The sample tree's widest level is level 2, with 3 nodes. The second tree, built from 8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15, is a *perfect* tree of height 3, and its levels have 1,2,4,8 nodes, the maximum 2^k that Section 12.3.1 allowed on level k. **Cost.** Every node enters and leaves the queue once: $\Theta(n)$ time, and the queue holds at most one level plus the start of the next, $O(\text{width})$ memory, which for a perfect tree is about $n/2$.

Listing 12.53 and **Listing 12.54** are the Java and Python versions; both print exactly the same lines.

Listing 12.53 `P1216Levels.java`: Problem 12.16 in Java.

```java
// P1216Levels.java - Problem 12.16: print a tree level by level with one queue.
import java.util.ArrayDeque;

public class P1216Levels extends TreeKit {

    // Breadth-first: all of level d leave the queue before any node of level d+1.
    static int printLevels(Node root) {  // returns the width of the widest level
        int widest = 0;
        if (root == null) return widest;
        ArrayDeque<Node> q = new ArrayDeque<>();
        q.add(root);
        for (int level = 0; !q.isEmpty(); level++) {
            int width = q.size();                // exactly the nodes of this level
            if (width > widest) widest = width;
            System.out.print("level " + level + ":");
            for (int i = 0; i < width; i++) {
                Node cur = q.poll();
                System.out.print(" " + cur.data);
                if (cur.left != null) q.add(cur.left);     // they belong to
                if (cur.right != null) q.add(cur.right);   // the next level
            }
            System.out.println();
        }
        return widest;
    }

    public static void main(String[] args) {
        Node root = buildSample();
        System.out.println("widest level has " + printLevels(root) + " nodes");
        Node full = insertAll(null, 8, 4, 12, 2, 6, 10, 14,
                1, 3, 5, 7, 9, 11, 13, 15);
        System.out.println("widest level has " + printLevels(full) + " nodes");
    }
}
```

Listing 12.54 `p1216_levels.py`: Problem 12.16 in Python.

```
# p1216_levels.py - Problem 12.16: print a tree level by level with one queue.
from collections import deque
from tree_kit import *


def print_levels(root):
    """Breadth-first: every node of level d leaves the queue before level d+1.
    Returns the width of the widest level."""
    widest = 0
    if root is None:
        return widest
    q = deque([root])
    level = 0
    while q:
        width = len(q)                          # exactly the nodes of this level
        widest = max(widest, width)
        print(f"level {level}:", end="")
        for _ in range(width):
            cur = q.popleft()
            print(f" {cur.data}", end="")
            if cur.left is not None:            # they belong to
                q.append(cur.left)
            if cur.right is not None:           # the next level
                q.append(cur.right)
        print()
        level += 1
    return widest


def main():
    root = build_sample()
    print(f"widest level has {print_levels(root)} nodes")
    full = insert_all(None, [8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15])
    print(f"widest level has {print_levels(full)} nodes")


if __name__ == "__main__":
    main()
```

A tempting wrong solution. Read the size inside the loop condition: `for (int i = 0; i < (int)q.size(); i++)`. The size changes while the loop runs, because every node taken out puts its children in, and `wrong/p1216_wrong.cpp` prints:

```
level 0: 6 3 10
level 1: 1 5
level 2: 7
level 3: 8
level 4: 9
```

After 6 is taken, the queue grows to two, so the loop runs again and takes 3; the queue grows again, and so on until *i* catches up with the size. Levels run together. The width must be read *once*, before the level starts, into a variable that does not change.

> **CHECK YOURSELF.** Change `printLevels` so that it prints the levels in *zigzag* order: level 0 left to right, level 1 right to left, level 2 left to right, and so on. What does it print for the sample tree?
>
> *Answer.* Collect the nodes of each level into a small array exactly as now, and print the array forward on even levels and backward on odd ones; the queue itself does not change. For the sample tree: `6`, `10 3`, `1 5 7`, `8`, `9`.

12.17 The Problems of This Chapter Compared

Table 12.5 lays the sixteen problems side by side. Read down the last column: most tempting wrong solutions fail on an input the problem's own example does not contain, and some even pass the example (the wrong balance check, the wrong sum on its first call, the wrong Josephus order). That is the strongest argument for Steps 6 and 7 of the method.

Table 12.5 The sixteen problems of the chapter: the structure each uses, its cost, and the input on which its tempting wrong solution fails. n is the number of nodes and h the height of the tree.

Problem	Structure	Time	Extra memory	The input the wrong solution gets wrong
12.1	tree, preorder	$\Theta(n)$	$O(h)$ stack	every tree (a leaf has `nullptr` children)
12.2	tree, depth-limited	O(nodes above level k)	$O(k)$	any level below the root
12.3	tree, postorder sum	$\Theta(n)$	$O(h)$	any second call
12.4	tree, preorder update	$\Theta(n)$	$O(h)$	every tree (only copies change)
12.5	tree, swap at each node	$\Theta(n)$	$O(h)$	any node with a left child
12.6	tree, heights	$O(n\log n)$; one pass $\Theta(n)$	$O(h)$	any balanced tree, the empty tree
12.7	BST, one path	$O(h)$	$O(h)$	any deletion below the root
12.8	BST, one path	$O(h)$	$O(h)$	any node with one missing child on the path
12.9	BST, `search`	$O(h)$	$O(h)$	an absent value
12.10	tree to list, inorder	$\Theta(n)$	$O(h)$, no new nodes	every tree of two or more levels
12.11	doubly linked list	$\Theta(n)$	$O(1)$	any list of two or more nodes
12.12	circular list; recurrence	$\Theta(nk)$; $\Theta(n)$	$O(n)$	every circle of two or more
12.13	recursion, backtracking	$\Theta\left(k\binom{n}{k}\right)$	$O(n)$	every input (nothing is undone)
12.14	tree with limits	$\Theta(n)$	$O(h)$	a value on the wrong side of a grandparent
12.15	BST, one path	$O(h)$	$O(h)$	two values on different sides
12.16	queue, level order	$\Theta(n)$	O(width)	any level with more than one node

The costs repeat three patterns. **Every node once** is $\Theta(n)$, by Equation (12.1). **One path from the root** is $O(h)$, between $\lg n$ and n by Equation (12.4), which is why balanced trees matter. **A walk inside a walk**, such as the naive balance check or printing every level with Problem 12.2, is the pattern to watch for: ask which value the inner walk recomputes, and let the outer walk *return* it instead, as `checkedHeight` returns the height.

12.18 Profitina in Practice

Profitina is an AI-visibility (GEO) platform, in production at app.profitina.com, that measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds. We describe the shapes of this chapter's problems in such a product, not how Profitina itself is built.

A threaded discussion is a tree. In Profitina's real-time community every reply answers exactly one post, its parent, and a top-level post is a root: a forest of trees, one per thread. Showing a thread in reading order is the depth-first order of Problem 12.1; a reply's indentation is its level, Problem 12.2; counting the replies under a post is the sum of Problem 12.3 with every value 1, and the "N new" counters on followed channels count only the roots. Deleting a post that has replies raises the question of Problem 12.7, what happens to the children: remove the subtree, move the replies up, or leave a placeholder.

A website is walked level by level. A site scan starts at the home page and follows links; pages one click away are level 1, two clicks level 2, and a scanner that stops at a fixed depth walks the site with a queue, as in Problem 12.16. (Real sites link back and across, so they are graphs, the subject of Chapter 13.)

Recursion has a depth limit. Profitina's server is written in Python, whose default recursion limit is about 1000 calls (Section 10.12) [10]. A recursive renderer needs one call per level, so a thread nested a thousand replies deep would raise `RecursionError`; the defenses are those of Problem 12.1, an explicit stack, or a cap on nesting depth.

The runnable practice version of these ideas is Profitina Laundry, the case study of this book series; Exercise 12.12 keeps its orders in a binary search tree keyed by order number, and asks for the queries a front desk needs.

12.19 Common Mistakes, and How to Avoid Them

No base case, or the wrong one. Every recursive tree function must say what happens at `nullptr` (Problem 12.1), and the answer must be the value that changes nothing under the combining step: 0 for $+$, true for `&&`, `INT_MIN` for max, -1 for the height (Problems 12.3 and 12.6).

Testing `k == 0` before `root == nullptr`. Test the pointer first, then use it (Problem 12.2). The same holds for every `root->left->data`: a child may be missing (Problem 12.8), and `search` may return `nullptr` (Problem 12.9).

Communicating through a global variable. A running total outside the function survives between calls (Problem 12.3). Return values and combine them; share a variable only on purpose, by pointer or in an object, when the calls really must (Problems 12.10 and 12.13).

Passing a node by value. `triple(Node node)` changes a copy (Problem 12.4). Receive nodes by pointer or by reference.

Swapping without saving. `a = b; b = a;` loses the old `a` (Problem 12.5). Save first, or let Python's tuple assignment save it for you.

Throwing away the returned subtree. A function that may replace the node it was given must return the replacement, and every caller must re-attach it: `root->left = delNode(root->left, x)` (Problem 12.7). The same applies to `root = delNode(root, x)` in `main`.

Using BST functions on a tree that is not a BST. A mirror, a negated tree or a tree built by hand may break Equation (12.3); `search` then misses values that are there (Problems 12.5 and 12.14).

Checking only parent and child. The BST rule is about all descendants; pass limits down (Problem 12.14).

Following a pointer you have just overwritten. After swapping `prev` and `next`, move to the saved old `next` (Problem 12.11).

Forgetting to undo in backtracking. Whatever a call adds to a shared list, it must remove before it returns (Problem 12.13).

Recursing too deep. Recursion depth equals the height of the tree; on a long chain, use an explicit stack or a queue, especially in Python, whose default limit is about 1000 calls.

Testing only the example. The wrong solutions of Problems 12.3, 12.6 and 12.12 pass their own examples. Test the empty tree, one node, one child on either side, an absent value and a long chain, and run the C++ tests once under AddressSanitizer or Valgrind.

12.20 Chapter Summary

- Seven steps solve a recursive problem on a list or tree (Section 12.2): an example, the answer for the empty structure, the tree with each call's arguments, "root, left answer, right answer", re-attaching returned subtrees, the edge cases, and tests under a memory checker.
- Every node once costs $\Theta(n)$, Equation (12.1); one path costs $O(h)$, with h between $\lceil \lg(n+1) \rceil - 1$ and $n-1$, Equation (12.4).
- Simple recursions combine returned values: depth first is preorder (Problem 12.1), a level is reached by passing $k-1$ down (Problem 12.2), sums and heights follow Equation (12.5) and Equation (12.2) (Problem 12.3); tripling keeps the BST order by Equation (12.6) (Problem 12.4); the mirror needs a saved pointer and is no longer a search tree (Problem 12.5).
- The empty tree is balanced; the helper that calls `height` costs Equation (12.7), and a height with a -2 sentinel checks balance in $\Theta(n)$ (Problem 12.6).
- Deletion has three cases and every parent re-attaches what its child returns (Problem 12.7); a parent is the last node before the value on its search path (Problem 12.8); a search result is tested before use (Problem 12.9); an inorder walk with a shared `prev` flattens a tree into a sorted doubly linked list in place (Problem 12.10).
- A doubly linked list reverses by swapping each node's arrows (Problem 12.11); the Josephus circle is a circular list and the recurrence Equation (12.8) (Problem 12.12); take-or-skip recursion lists combinations, counted by Equation (12.9), if shared state is undone (Problem 12.13).
- A BST check passes intervals down, because Equation (12.3) concerns all descendants (Problem 12.14); the lowest common ancestor is where two search paths part (Problem 12.15); a queue whose size is read once per level prints a tree level by level (Problem 12.16).
- Graphs, where cycles are allowed, come in Chapter 13; heaps in Chapter 14.

12.21 Exercises

Exercises 12.1 to 12.4 ask for code; their solutions are the programs of Section 12.22. The others ask for reasoning and are answered in words.

1. **(Leaves.)** Write `int countLeaves(Node *root)`, the number of nodes with no children. What does it return for the sample tree and for the empty tree?

2. **(Recursive list reversal.)** Reverse a *singly* linked list recursively: `LNode *reverseList(LNode *head)` returns the new head. Test it on 1, 2, 3, 4, 5.
3. **(k-th smallest.)** Return the k-th smallest value of a binary search tree ($k = 1$ is the minimum), or -1 if the tree has fewer than k nodes, stopping as soon as it is found. Give the results for $k = 1, \ldots, 9$ on the sample tree.
4. **(Same tree.)** Write `bool sameTree(Node *a, Node *b)`: true if the two trees have the same shape and the same values in the same places. Compare the sample tree with a second copy of itself, and with the tree built from 6, 3, 10, 1, 5, 7, 8.
5. **(Traversals by hand.)** Insert 50, 30, 70, 20, 40, 60, 80, 35 into an empty binary search tree. Draw it, and give its preorder, inorder, postorder and level order, its height, and its leaves.
6. **(Deletion by hand.)** In the tree of Exercise 12.5, delete 30, then 50, with the successor rule of Problem 12.7. Draw the tree after each deletion and give its preorder.
7. **(Balanced?)** Is the tree of Exercise 12.5 height-balanced? Which single value, inserted into it, would make it unbalanced, and at which node?
8. **(The worst order.)** Values $1, 2, \ldots, n$ are inserted in increasing order. What is the height of the tree, and how many comparisons do the n insertions make in total? What does the helper version of Problem 12.6 cost on this tree, and why?
9. **(A circular doubly linked list.)** In a circular doubly linked list with a pointer `head` to one node, write the pointer assignments that insert a new node `p` *before* `head` (that is, at the end of the list), and say what changes when the list is empty.
10. **(Josephus by the formula.)** Use Equation (12.8) to find the survivor for $n = 8$, $k = 3$, and check your answer by simulating the circle by hand.
11. **(Back from a list to a tree.)** Problem 12.10 turned a tree into a sorted doubly linked list. Describe a recursive algorithm that turns a sorted array of n values into a binary search tree of height $\lceil \lg(n+1) \rceil - 1$, and say its running time.
12. **(Profitina Laundry.)** Profitina Laundry keeps the day's orders in a binary search tree keyed by order number, each node holding the order number and its weight in kilograms. Using the functions of this chapter, say how to (a) find an order, (b) list all orders in increasing order number, (c) compute the total weight of the day, (d) count the orders whose numbers lie between a and b without visiting the whole tree, and (e) remove an order that has been collected. Give the cost of each.

12.22 Solutions to the Exercises

Exercises 1 to 4. **Listing 12.55**, **Listing 12.56** and **Listing 12.57** contain all four functions and print the same lines.

Listing 12.55 `ex12_solutions.cpp`: the programming exercises of Chapter 12 in C++.

```
// ex12_solutions.cpp - the programming exercises of Chapter 12 (12.1 to 12.4).
#include "tree_kit.hpp"

// Exercise 12.1: the number of leaves (nodes with no children).
int countLeaves(Node *root) {
    if (root == nullptr) return 0;
    if (root->left == nullptr && root->right == nullptr) return 1;
    return countLeaves(root->left) + countLeaves(root->right);
}

// Exercise 12.2: reverse a singly linked list recursively; return the new head.
struct LNode {
    int data;
    LNode *next;
};

LNode *reverseList(LNode *head) {
    if (head == nullptr || head->next == nullptr) return head;
    LNode *newHead = reverseList(head->next);   // reverse everything after head
    head->next->next = head;                    // the old second node points back
    head->next = nullptr;                       // head is now the last node
    return newHead;
}

// Exercise 12.3: the k-th smallest value (k = 1, 2, ...), or -1 if none.
int kthSmallest(Node *root, int &k) {           // k counts down during the walk
    if (root == nullptr) return -1;
    int found = kthSmallest(root->left, k);
    if (found != -1) return found;
    if (--k == 0) return root->data;            // this is the k-th node in order
    return kthSmallest(root->right, k);
}

// Exercise 12.4: same shape and same values in the same places?
bool sameTree(Node *a, Node *b) {
    if (a == nullptr || b == nullptr) return a == b;   // both empty, or not
    return a->data == b->data && sameTree(a->left, b->left) &&
           sameTree(a->right, b->right);
}

int main() {
    Node *root = nullptr, *other = nullptr, *third = nullptr;
    buildSample(&root);
    printf("12.1 leaves of the sample tree: %d, of the empty tree: %d\n",
           countLeaves(root), countLeaves(nullptr));
```

Listing 12.55 (Continued) *ex12_solutions.cpp: the programming exercises of Chapter 12 in C++.*

```
    LNode *head = nullptr;
    for (int v = 5; v >= 1; v--) head = new LNode{v, head};   // 1 2 3 4 5
    head = reverseList(head);
    printf("12.2 reversed:");
    for (LNode *p = head; p != nullptr; p = p->next) printf(" %d", p->data);
    printf("\n");
    while (head != nullptr) {
        LNode *next = head->next;
        delete head;
        head = next;
    }

    printf("12.3 k-th smallest:");
    for (int k = 1; k <= 9; k++) {
        int left = k;
        printf(" %d", kthSmallest(root, left));
    }
    printf("\n");

    buildSample(&other);
    insertAll(&third, {6, 3, 10, 1, 5, 7, 8});
    printf("12.4 sample vs same inserts: %s, sample vs one node fewer: %s\n",
           sameTree(root, other) ? "same" : "different",
           sameTree(root, third) ? "same" : "different");
    clear(&root);
    clear(&other);
    clear(&third);
    return 0;
}
```

Listing 12.56 `Ex12Solutions.java`: the programming exercises of Chapter 12 in Java.

```java
// Ex12Solutions.java - the programming exercises of Chapter 12 (12.1 to 12.4).
public class Ex12Solutions extends TreeKit {

    // Exercise 12.1: the number of leaves (nodes with no children).
    static int countLeaves(Node root) {
        if (root == null) return 0;
        if (root.left == null && root.right == null) return 1;
        return countLeaves(root.left) + countLeaves(root.right);
    }

    // Exercise 12.2: reverse a singly linked list recursively; return new head.
    static class LNode {
        int data;
        LNode next;

        LNode(int data, LNode next) {
            this.data = data;
            this.next = next;
        }
    }

    static LNode reverseList(LNode head) {
        if (head == null || head.next == null) return head;
        LNode newHead = reverseList(head.next);   // reverse everything after head
        head.next.next = head;                    // the old second node points back
        head.next = null;                         // head is now the last node
        return newHead;
    }

    // Exercise 12.3: the k-th smallest value (k = 1, 2, ...), or -1 if none.
    static int remaining;                         // counts down during the walk

    static int kthSmallest(Node root) {
        if (root == null) return -1;
        int found = kthSmallest(root.left);
        if (found != -1) return found;
        if (--remaining == 0) return root.data;   // this is the k-th node in order
        return kthSmallest(root.right);
    }
```

Listing 12.56 (Continued) *Ex12Solutions.java: the programming exercises of Chapter 12 in Java.*

```
    // Exercise 12.4: same shape and same values in the same places?
    static boolean sameTree(Node a, Node b) {
        if (a == null || b == null) return a == b;    // both empty, or not
        return a.data == b.data && sameTree(a.left, b.left)
                && sameTree(a.right, b.right);
    }

    public static void main(String[] args) {
        Node root = buildSample();
        System.out.printf("12.1 leaves of the sample tree: %d, "
                + "of the empty tree: %d%n", countLeaves(root), countLeaves(null));

        LNode head = null;
        for (int v = 5; v >= 1; v--) head = new LNode(v, head);    // 1 2 3 4 5
        head = reverseList(head);
        System.out.print("12.2 reversed:");
        for (LNode p = head; p != null; p = p.next) System.out.print(" " + p.data);
        System.out.println();

        System.out.print("12.3 k-th smallest:");
        for (int k = 1; k <= 9; k++) {
            remaining = k;
            System.out.print(" " + kthSmallest(root));
        }
        System.out.println();

        Node other = buildSample();
        Node third = insertAll(null, 6, 3, 10, 1, 5, 7, 8);
        System.out.printf("12.4 sample vs same inserts: %s, "
                + "sample vs one node fewer: %s%n",
                sameTree(root, other) ? "same" : "different",
                sameTree(root, third) ? "same" : "different");
    }
}
```

Listing 12.57 `ex12_solutions.py`: the programming exercises of Chapter 12 in Python.

```python
# ex12_solutions.py - the programming exercises of Chapter 12 (12.1 to 12.4).
from tree_kit import *


def count_leaves(root):
    """Exercise 12.1: the number of leaves (nodes with no children)."""
    if root is None:
        return 0
    if root.left is None and root.right is None:
        return 1
    return count_leaves(root.left) + count_leaves(root.right)


class LNode:
    def __init__(self, data, nxt=None):
        self.data = data
        self.next = nxt


def reverse_list(head):
    """Exercise 12.2: reverse a singly linked list recursively; return new head."""
    if head is None or head.next is None:
        return head
    new_head = reverse_list(head.next)       # reverse everything after head
    head.next.next = head                    # the old second node points back
    head.next = None                         # head is now the last node
    return new_head


def kth_smallest(root, k):
    """Exercise 12.3: the k-th smallest value (k = 1, 2, ...), or -1 if none.
    k is a one-element list, so that every call counts down the same number."""
    if root is None:
        return -1
    found = kth_smallest(root.left, k)
    if found != -1:
        return found
    k[0] -= 1
    if k[0] == 0:                            # this is the k-th node in order
        return root.data
    return kth_smallest(root.right, k)
```

Listing 12.57 (Continued) *ex12_solutions.py: the programming exercises of Chapter 12 in Python.*

```
def same_tree(a, b):
    """Exercise 12.4: same shape and same values in the same places?"""
    if a is None or b is None:
        return a is b                       # both empty, or not
    return (a.data == b.data and same_tree(a.left, b.left)
            and same_tree(a.right, b.right))


def main():
    root = build_sample()
    print(f"12.1 leaves of the sample tree: {count_leaves(root)}, "
          f"of the empty tree: {count_leaves(None)}")

    head = None
    for v in range(5, 0, -1):
        head = LNode(v, head)               # 1 2 3 4 5
    head = reverse_list(head)
    print("12.2 reversed:", end="")
    p = head
    while p is not None:
        print(f" {p.data}", end="")
        p = p.next
    print()

    print("12.3 k-th smallest:", end="")
    for k in range(1, 10):
        print(f" {kth_smallest(root, [k])}", end="")
    print()

    other = build_sample()
    third = insert_all(None, [6, 3, 10, 1, 5, 7, 8])
    word = lambda same: "same" if same else "different"
    print(f"12.4 sample vs same inserts: {word(same_tree(root, other))}, "
          f"sample vs one node fewer: {word(same_tree(root, third))}")


if __name__ == "__main__":
    main()
```

```
12.1 leaves of the sample tree: 3, of the empty tree: 0
12.2 reversed: 5 4 3 2 1
12.3 k-th smallest: 1 3 5 6 7 8 9 10 -1
12.4 sample vs same inserts: same, sample vs one node fewer: different
```

A few words on each. **(1)** A leaf is recognized by two `nullptr` children and counts 1; the empty tree counts 0; any other node counts its subtrees' leaves. The sample tree has three leaves, 1, 5 and 9. **(2)** Reverse everything after the head first; the old second node is then the *last* node of the reversed rest, so `head->next->next = head` hangs the old head after it, and `head->next = nullptr` ends the list there. The base case, a list of zero or one node, is already reversed. The list 1 to 5 becomes 5, 4, 3, 2, 1. The recursion depth is n, so for long lists the loop of Problem 8.10 is safer. **(3)** An inorder walk visits the values in increasing order (Section 12.3.1), so the k-th node visited holds the k-th smallest value. The counter must be shared by all calls (a reference in C++, a field in Java, a one-element list in Python), and the walk stops as soon as the value is found, costing $O(h + k)$. For $k = 9$ the tree has only eight values, so the answer is -1. **(4)** Two empty trees are the same, an empty and a non-empty tree are not, and two non-empty trees are the same when their roots agree and their left and right subtrees are the same. The two trees built from the same insertions are the same; removing the last insertion, 9, makes them different.

Exercise 5. The root is 50; 30 and 70 are its children; 20 and 40 are the children of 30; 60 and 80 of 70; 35 is the left child of 40. Preorder `50 30 20 40 35 70 60 80`; inorder `20 30 35 40 50 60`

`70 80`; postorder `20 35 40 30 60 80 70 50`; level order `50 30 70 20 40 60 80 35`. The height is 3 (the path 50, 30, 40, 35), and the leaves are 20, 35, 60 and 80.

Exercise 6. Deleting 30 (two children): its successor is the smallest value in its right subtree, 40, 35: go left from 40 to 35, which has no left child, so the successor is 35. Copy 35 into the node of 30 and delete the old 35, a leaf. Preorder `50 35 20 40 70 60 80`. Deleting 50 (two children): the successor is 60, the leftmost node of the right subtree. Copy 60 into the root, delete the old 60, a leaf. Preorder `60 35 20 40 70 80`.

Exercise 7. Heights of the subtrees: node 40 has 0 and -1; node 30 has 0 and 1; node 70 has 0 and 0; the root has 2 and 1. Every difference is at most 1, so the tree is height-balanced. Inserting 36 (or any value between 35 and 40, or below 35 but above 30, such as 33) hangs a node below 35: then node 40 has heights 1 and -1, a difference of 2, and it is unbalanced (as is node 30, with heights 0 and 2).

Exercise 8. Each new value goes right at every node, so the tree is a chain of height $n-1$, the upper bound of Equation (12.4), and the insertions make $0+1+\cdots+(n-1)=\frac{n(n-1)}{2}$ comparisons, $\Theta(n^2)$. The helper of Problem 12.6 costs only $\Theta(n)$ here: for $n \geq 3$ the root itself has heights -1 and $n-2$, and the `&&` stops at once.

Exercise 9. Let `last = head->prev` (in a circular list the node before the head is the last one). Then `p->next = head; p->prev = last; last->next = p; head->prev = p;`, four assignments, the two arrows on each side. The head does not change. If the list is empty (`head == nullptr`), there is no `last`: the new node becomes the whole circle, `p->next = p; p->prev = p; head = p;`.

Exercise 10. $J(1)=0$, $J(2)=3 \bmod 2=1$, $J(3)=4 \bmod 3=1$, $J(4)=4 \bmod 4=0$, $J(5)=3 \bmod 5=3$, $J(6)=6 \bmod 6=0$, $J(7)=3 \bmod 7=3$, $J(8)=6 \bmod 8=6$: the survivor is member 7. Simulating 1 to 8 with $k=3$: 3, 6, 1, 5, 2, 8, 4 leave, in that order, and 7 remains.

Exercise 11. Take the middle element as the root, and build the left and right subtrees recursively from the elements before and after it; an empty range gives the empty tree. Each element becomes one node with constant work per call, so the time is $\Theta(n)$, as in Equation (12.1). The two halves differ in size by at most one at every node, so the tree is height-balanced with the smallest possible height, $\lceil \lg(n+1) \rceil - 1$. With Problem 12.10, this rebuilds any search tree into a perfectly short one in linear time.

Exercise 12. (a) Search by order number: $O(h)$. (b) Inorder traversal: $\Theta(n)$. (c) The sum of Problem 12.3 on the weights, in a `double`: $\Theta(n)$. (d) At a node with number x, go left only if $a < x$, go right only if $x < b$, and count x if $a \leq x \leq b$; only the two boundary paths and the nodes inside the range are visited, $O(h+m)$ for m orders counted. (e) The deletion of Problem 12.7: $O(h)$. Orders usually arrive in increasing number, which builds the chain of Exercise 8 and turns every $O(h)$ into $O(n)$: the practical reason why `std::map` and `TreeMap` keep their trees balanced [1], [5].

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] Free Software Foundation, "Program Instrumentation Options," in *Using the GNU Compiler Collection (GCC)*, GCC 16.2, Aug. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://gcc.gnu.org/onlinedocs/

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *Proc. USENIX Annual Technical Conf.*, Boston, MA, USA, 2012, pp. 309–318.

[4] Valgrind Developers, *Valgrind User Manual*, release 3.27.1, May 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://valgrind.org/docs/manual/

[5] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Boston, MA, USA: Pearson, 2014.

[6] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Python*. Hoboken, NJ, USA: Wiley, 2013.

[7] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

[8] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[9] Oracle, “Class ArrayDeque,” *Java SE 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/

[10] Python Software Foundation, “Evaluation order,” “sys.getrecursionlimit,” “collections.deque,” and “gc: Garbage Collector interface,” *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/

[11] G. M. Adelson-Velsky and E. M. Landis, “An algorithm for the organization of information,” *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1263, 1962.

[12] T. N. Hibbard, “Some combinatorial properties of certain trees with applications to searching and sorting,” *Journal of the ACM*, vol. 9, no. 1, pp. 13–28, 1962.

[13] cppreference.com, “new expression,” “delete expression,” and “std::queue.” Accessed: Sep. 24, 2026. [Online]. Available: https://en.cppreference.com/w/cpp

[14] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994.

Binary search trees, their traversals, deletion by the successor, and the recursion-tree way of counting calls are treated in Cormen et al. [1], and trees with their history in Knuth [8]; deletion by copying the successor goes back to Hibbard [12], and height-balanced trees to Adelson-Velsky and Landis [11]. The Josephus problem and its recurrence are worked out in full in Graham, Knuth and Patashnik [14]. Weiss [5], Goodrich et al. [6] and Sedgewick and Wayne [7] give the same structures in C++, Python and Java respectively. The C++ rules for `new` and `delete` follow cppreference [13], the Java stack and queue the API specification [9], and Python’s evaluation order and recursion limit its documentation [10]. The memory checkers are described in the GCC manual [2], the Valgrind manual [4] and the original AddressSanitizer paper [3].

Chapter 13
Graphs

By the end of this chapter you will be able to say precisely what a graph is and use its vocabulary without hesitation: vertex, edge, directed and undirected, weighted, degree, path, cycle, connected component, and directed acyclic graph; prove and use the handshake lemma; store one and the same graph as an adjacency matrix, an adjacency list and an edge list, and choose among them by counting memory and time; search a graph breadth first with a queue, and use that search to find hop distances and fewest-hop routes; search it depth first, both recursively and with an explicit stack, record discovery and finish times, and use them to find cycles; split a graph into its connected components; put the tasks of a workflow into an order that respects every prerequisite, by Kahn's algorithm and by depth-first search, and detect when no such order exists; and find shortest weighted routes with Dijkstra's algorithm and a priority queue. One graph, the campus map of the Pustaka Ceria library and its branch libraries, runs through the whole chapter, so that every representation and every algorithm is seen working on the same picture. Every procedure is traced by hand, drawn, costed, and then programmed in C++, Java and Python, and every printed output is real.

13.1 Where We Left Off

Every structure in this book so far has arranged its elements in a very particular shape. An array (Chapter 1) and a linked list (Chapters 7 and 9) put them in a line: every element has at most one element before it and at most one after it. A stack and a queue (Chapter 5) keep that line but restrict where we may touch it. A tree (Chapter 11) relaxes the line into a hierarchy: an element may have many children, but it still has at most one parent, and there is exactly one way to walk from the root down to any element.

The world is not always so tidy. The footpaths between the buildings of a campus do not form a line or a hierarchy; from the main library you can walk to the cafeteria directly, or around through the administration office and the engineering building, and some paths close into loops. Friendships, flight routes, web pages that link to one another, the tasks of a project that must wait for other tasks: all of these are collections of things together with *connections* between pairs of things, with no restriction on how many connections a thing may have or on whether the connections form loops. The data structure that captures exactly this is the *graph*, and it is the most general structure in this book. A list is a graph, a tree is a graph, and so is almost everything else.

Two tools from earlier chapters carry most of the weight in this one. The *queue* of Chapter 5, first in, first out, is the engine of breadth-first search in Section 13.6: it makes the search spread out from its starting point in rings of equal distance. The *stack* of Chapter 5, last in, first out, together with the recursion of Chapter 10, which is a stack in disguise (Section 5.10 showed the call stack at work), is the engine of depth-first search in Section 13.7. A third tool arrives near the end: the *priority queue*, which Section 5.15 previewed and Chapter 14 builds properly as a heap. Section 13.10 uses the priority queue that each language's library provides, so that Dijkstra's shortest-path algorithm can be understood now and its engine studied in the next chapter.

The questions a graph lets us answer are surprisingly practical. Can I get from here to there at all? What is the route with the fewest changes? What is the shortest route in meters? Which buildings are cut off from the rest? In what order must a new book be processed so that no step comes before a step it depends on? Does this set of dependencies contain a loop, so that no order is possible? Each section of this chapter answers one of these questions on the Pustaka Ceria campus map, and Section 13.11 shows the same questions being asked, and answered with the same algorithms, on the road network of a whole planet and on a social network of more than three and a half billion people.

13.2 A Story to Begin: Seven Bridges and a Campus Map

13.2.1 The bridges of Königsberg

Graph theory has an unusually precise birthday. In the eighteenth century the Prussian city of Königsberg (today Kaliningrad in Russia) stood on both banks of the river Pregel and on two islands in the middle of it. Seven bridges joined the four pieces of land: two from the north bank to the larger island, two from the south bank to the larger island, one between the two islands, and one from each bank to the smaller island. The citizens liked to walk on Sunday afternoons, and a puzzle went around: is there a walk through the city that crosses every bridge exactly once?

Nobody could find one, and nobody could prove there was none, until the Swiss mathematician Leonhard Euler took up the question and presented his answer to the Academy of Sciences in Saint Petersburg in 1736; it was printed in the Academy's proceedings a few years later [1]. Euler's decisive step was to throw away everything that did not matter. The shape of the islands, the length of the bridges, the width of the river, the streets on each bank: none of it affects the answer. What matters is only *which* pieces of land exist and *which pairs* of them are joined by bridges, and how many times. **Figure 13.1** shows the city reduced in two steps. Panel (a) is a schematic map with the four land masses labeled A (north bank), B (the larger island), C (the smaller island) and D (south bank) and the seven bridges drawn in amber. Panel (b) keeps only what Euler kept: one point for each land mass and one line for each bridge.

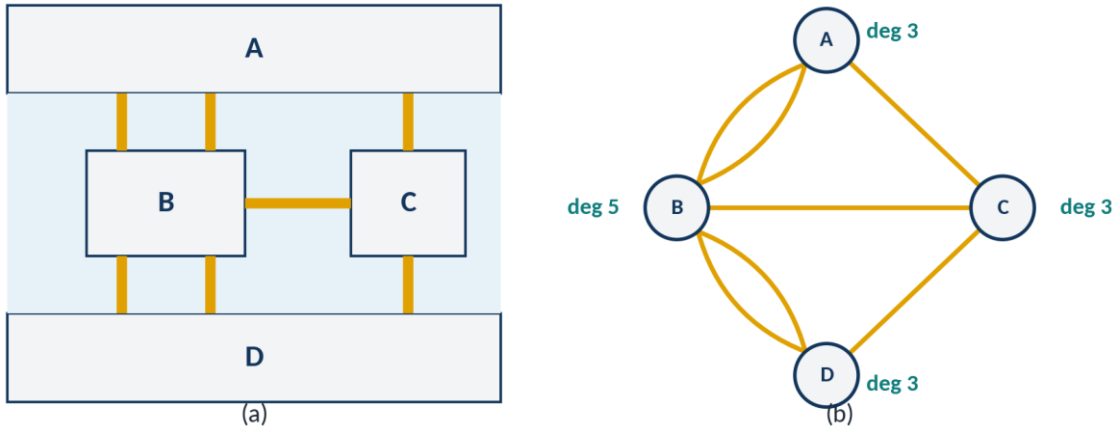


Figure 13.1 The seven bridges of Königsberg: (a) a schematic map with the four land masses A to D and the seven bridges in amber; (b) the same situation as a graph, with the number of bridges touching each land mass.

In panel (b) we can reason about walks without looking at the city at all. Suppose a walk crosses every bridge exactly once. Every time the walk passes *through* a land mass, it arrives over one bridge and leaves over another, so it uses the bridges of that land mass two at a time. The only places where a bridge can be used "alone" are the land mass where the walk starts and the one

where it ends. So every land mass other than the start and the end must touch an *even* number of bridges. But in Königsberg all four land masses touch an odd number: A, C and D touch three bridges each, and B touches five. At most two of them can be the start and the end, which leaves at least two land masses with an odd count and no way to satisfy them. The walk is impossible, and the argument is a proof, not a failure to search hard enough.

Euler's argument contains, in miniature, almost everything this chapter does. It replaced a messy real situation by *points and connections*. It counted, for each point, how many connections touch it, a number we will call the *degree*. And it reasoned about *walks* along the connections. That is graph theory, and 290 years later it describes road networks, social networks, the web, and the dependencies between the tasks of a project.

13.2.2 From bridges to library branches

The Pustaka Ceria library, which has kept us company since Chapter 1, has grown. Besides its main building it now runs two branch libraries on campus, one in the engineering building and one in the medical faculty, and a third branch downtown, in the city annex, next to the warehouse where new books are received. Books travel between the branches every day, readers walk from one building to another, and the librarians keep asking questions that sound like Euler's puzzle: how do I get a book from the main library to the medical branch, which route is the shortest, which buildings can I not reach on foot at all, and in which order must a newly delivered book be processed before it can stand on a shelf?

To answer them, we do what Euler did. We forget the shapes of the buildings and the curves of the paths and keep only the buildings and the footpaths between them, each footpath labeled with its length in meters. The result, the *campus map*, appears in Section 13.4, and it is the running example of the whole chapter. First, though, we need the vocabulary.

13.3 The Vocabulary of Graphs

13.3.1 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. Formally, a graph is written as in Equation (13.1).

$$G = (V, E), \qquad E \subseteq \big\{\{u, v\} : u, v \in V,\ u \neq v\big\} \tag{13.1}$$

In words, Equation (13.1) says: a graph is a pair made of a vertex set and an edge set, and each edge is an unordered pair $\{u, v\}$ of two different vertices. The vertices are the "things" (land masses, buildings, people, web pages, tasks); the edges are the connections between them (bridges, footpaths, friendships, links, prerequisites). Vertices are also called *nodes*, and edges are also called *arcs* or *links*; this chapter says *vertex* and *edge* throughout, and keeps the word *node* for the nodes of linked lists. The number of vertices is written $|V|$ and the number of edges $|E|$; in running text and in formulas about running time we often write simply V and E for these numbers, as Cormen et al. do [2].

When an edge $\{u, v\}$ exists, u and v are called *adjacent*, or *neighbors*, and the edge is said to be *incident* on both of them. The edges in Equation (13.1) have two further properties worth naming. First, $u \neq v$: an edge from a vertex to itself, called a *self-loop*, is not allowed. Second, E is a *set*, so the same pair cannot appear twice: there are no *parallel edges*. A graph with these two restrictions is called *simple*. The Königsberg graph of **Figure 13.1**(b) is *not* simple, because A and

B are joined by two bridges; such a graph is called a *multigraph*. Every graph from here on is simple, which is what almost all programs assume.

EXAMPLE. In **Figure 13.2**(a), $V = \{A, B, C, D, E\}$ and $E = \big\{\{A, B\}, \{B, C\}, \{C, D\}, \{D, A\}, \{A, C\}\big\}$, so $|V| = 5$ and $|E| = 5$. A and B are adjacent; A and E are not. Vertex E has no neighbors at all: it is an *isolated vertex*, and it is still a vertex of the graph.

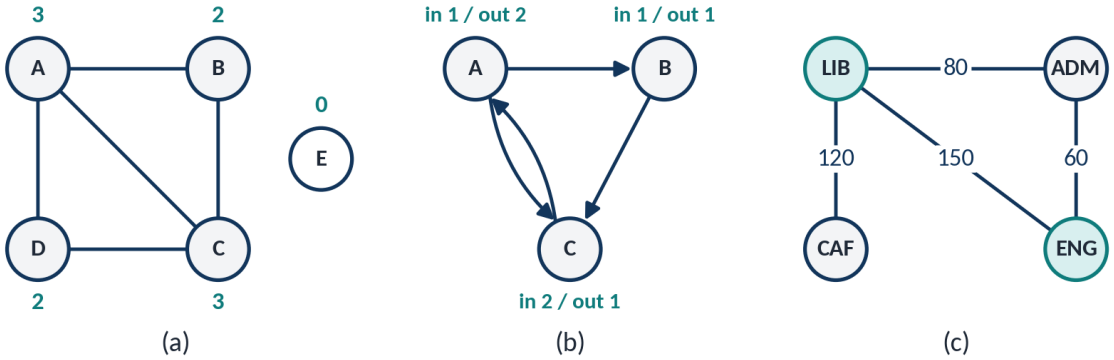


Figure 13.2 Three small graphs: (a) an undirected graph with the degree of each vertex written beside it, including the isolated vertex E; (b) a directed graph with the in-degree and out-degree of each vertex; (c) a weighted graph, a corner of the campus map with footpath lengths in meters.

13.3.2 Directed and undirected

The edges of Equation (13.1) have no direction: $\{u, v\}$ and $\{v, u\}$ are the same set, so the edge can be traveled either way. Such a graph is *undirected*. A footpath is undirected: if you can walk from the library to the cafeteria, you can walk back.

Many connections, however, run one way only. A one-way street, a hyperlink from one web page to another, "follows" on a social network, and "task u must be finished before task v starts" all have a direction. In a *directed graph*, or *digraph*, each edge is an *ordered* pair (u, v), drawn as an arrow from u to v; u is its *tail* and v its *head*. The edge (u, v) does not imply the edge (v, u). In **Figure 13.2**(b) there are arrows A to B, A to C, C to A and B to C; you can go from B to C, but not from C to B in one step.

An undirected graph can always be viewed as a directed graph in which every edge appears in both directions: the footpath $\{u, v\}$ becomes the two arrows (u, v) and (v, u). This is exactly how the programs of this chapter store undirected graphs, and it is why many results come with a factor of 2.

13.3.3 Weighted graphs

Often an edge carries a number: the length of a footpath, the travel time of a road, the price of a flight, the capacity of a pipe. A *weighted graph* has a *weight function* w that gives every edge (u, v) a number $w(u, v)$. In **Figure 13.2**(c), $w(\text{LIB,ADM}) = 80$: the footpath from the main library to the administration office is 80 meters long. An *unweighted* graph can be seen as a weighted one in which every edge has weight 1, so that the "length" of a route is simply the number of edges it uses. Sections 13.6 and 13.10 show that these two views of length, counting edges and adding weights, lead to two different algorithms.

13.3.4 Degree and the handshake lemma

The *degree* of a vertex v in an undirected graph, written $\deg(v)$, is the number of edges incident on it, which in a simple graph is the number of its neighbors. In **Figure 13.2**(a), $\deg(A) = 3$

(neighbors B, C, D), $\deg(B) = 2$, $\deg(C) = 3$, $\deg(D) = 2$ and $\deg(E) = 0$. Add them up: $3 + 2 + 3 + 2 + 0 = 10$, which is exactly twice the number of edges, 5. That is no coincidence.

> **THE HANDSHAKE LEMMA.** In every undirected graph, the sum of the degrees of all vertices is twice the number of edges:

$$\sum_{v \in V} \deg(v) = 2|E| \tag{13.2}$$

Proof. Count the pairs (vertex, edge incident on that vertex) in two ways. Counting vertex by vertex, vertex v contributes $\deg(v)$ pairs, so the total is the left-hand side of Equation (13.2). Counting edge by edge, every edge $\{u, v\}$ contributes exactly two pairs, one for u and one for v, so the total is $2|E|$. Both counts count the same pairs, so they are equal. ■

The name comes from a party: if every guest counts the hands they shook, the total is twice the number of handshakes, because every handshake involves two hands. Two consequences are worth remembering. First, the sum of degrees is always even, so *the number of vertices of odd degree is even*: a graph with degrees 3, 3, 3, 3, 3 cannot exist, since the sum 15 is odd. Second, the lemma is a free test for a program that builds a graph: add up the lengths of all adjacency lists and compare with twice the number of edges. The program in Section 13.5.5 does exactly that.

In a directed graph one number is not enough. The *in-degree* $\deg^-(v)$ counts the arrows that end at v, and the *out-degree* $\deg^+(v)$ counts the arrows that start at v. In **Figure 13.2**(b), A has in-degree 1 (from C) and out-degree 2 (to B and C). Every arrow has exactly one tail and one head, so the same double-counting argument gives

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E| \tag{13.3}$$

For **Figure 13.2**(b): in-degrees $1 + 1 + 2 = 4$, out-degrees $2 + 1 + 1 = 4$, and there are four arrows. Section 13.9 builds a whole algorithm, Kahn's topological sort, on in-degrees.

13.3.5 Paths and cycles

A *path* from u to v is a sequence of vertices $v_0, v_1, \ldots, v_k$ with $v_0 = u$ and $v_k = v$ in which every two consecutive vertices are joined by an edge (in a directed graph, by an arrow pointing forward). The *length* of the path is the number of edges it uses, k. A single vertex is a path of length 0 from itself to itself. If a path from u to v exists, v is *reachable* from u. A path is *simple* if no vertex appears twice. In a weighted graph we also speak of the *weight* of a path, the sum of the weights of its edges; to avoid confusion, this chapter calls the number of edges the number of *hops* whenever weights are around.

A *cycle* is a path of length at least 1 that starts and ends at the same vertex; in an undirected graph we also require at least three distinct vertices and no repeated edge, so that walking along an edge and straight back does not count. In **Figure 13.2**(a), A, B, C, A is a cycle of length 3, and A, B, C, D, A is a cycle of length 4. A graph with no cycle is *acyclic*. The trees of Chapter 11 (Section 11.2) are exactly the undirected graphs that are connected and acyclic, which is why every tree algorithm can walk from the root to any node without ever meeting a node twice. A general graph has no such guarantee, and every algorithm in this chapter must remember which vertices it has already seen, or it will walk around a cycle forever.

13.3.6 Connectivity and connected components

An undirected graph is *connected* if every vertex is reachable from every other vertex. **Figure 13.2**(a) is not connected, because nothing reaches E. A disconnected graph falls apart into pieces:

> **DEFINITION.** A *connected component* of an undirected graph is a maximal set of vertices in which every vertex is reachable from every other. "Maximal" means that no vertex outside the set can be added without breaking that property.

Figure 13.2(a) has two components, $\{A, B, C, D\}$ and $\{E\}$. Every vertex lies in exactly one component, because "is reachable from" is an *equivalence relation* on the vertices of an undirected graph: every vertex reaches itself (a path of length 0), if u reaches v then v reaches u (walk the path backward), and if u reaches v and v reaches x then u reaches x (glue the two paths). The components are its equivalence classes. A connected graph is simply a graph with exactly one component. Section 13.8 computes components with one search per component.

13.3.7 Directed acyclic graphs

A directed graph with no directed cycle is a *directed acyclic graph*, universally shortened to *DAG*. DAGs model dependencies: an arrow (u, v) says "u must come before v", and a cycle would say that some task must come before itself, which is impossible. The prerequisites of university courses, the build order of the files of a program, the steps of a recipe and, in Section 13.9, the steps that turn a delivered book into a shelved one are all DAGs. The central fact about DAGs, proved in Section 13.9, is that their vertices can always be listed so that every arrow points forward in the list, and that a directed graph has such a listing only if it is acyclic.

13.3.8 How many edges can a graph have? Sparse and dense

In a simple undirected graph, each edge is a pair of different vertices, and there are $\binom{|V|}{2}$ such pairs, so

$$|E| \le \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} \tag{13.4}$$

and in a simple directed graph each ordered pair may carry an arrow, so $|E| \le |V|(|V|-1)$. A graph with close to the maximum number of edges is called *dense*; one with far fewer, typically a small constant times $|V|$, is called *sparse*. A useful single number is the *density*

$$\rho = \frac{|E|}{|V|(|V|-1)/2} \tag{13.5}$$

which is 0 for a graph with no edges and 1 for the *complete graph*, in which every pair of vertices is adjacent. The distinction is not academic: Section 13.5 shows that the best way to store a graph depends on it, and Section 13.11 shows that the graphs that matter most in practice, road networks and social networks, are overwhelmingly sparse. In a road network almost every junction has at most four or five roads, whatever the size of the country, so $|E|$ grows like $|V|$ and not like $|V|^2$.

> **IN PLAIN LANGUAGE.** A graph is a set of dots and a set of lines between pairs of dots. Lines may have arrows (directed) and numbers (weighted). The degree of a dot is how many lines touch it, and all degrees together count every line twice. A path is a way to walk along lines; a

cycle is a walk that comes back to where it started. The dots you can reach from each other form a component.

13.4 The Running Example: the Pustaka Ceria Campus Map

Figure 13.3 is the graph that the rest of the chapter works on. It has ten vertices, the buildings of the Pustaka Ceria network, numbered 0 to 9 in the order the programs use, each with a three-letter code: the main library LIB (0), the cafeteria CAF (1), the administration office ADM (2), the engineering branch library ENG (3), the dormitory DRM (4), the medical branch library MED (5), the auditorium AUD (6), the parking area PRK (7), the city annex branch library ANX (8) and the book warehouse WHS (9). The four libraries are drawn in teal. The eleven edges are the footpaths, each labeled with its length in meters.

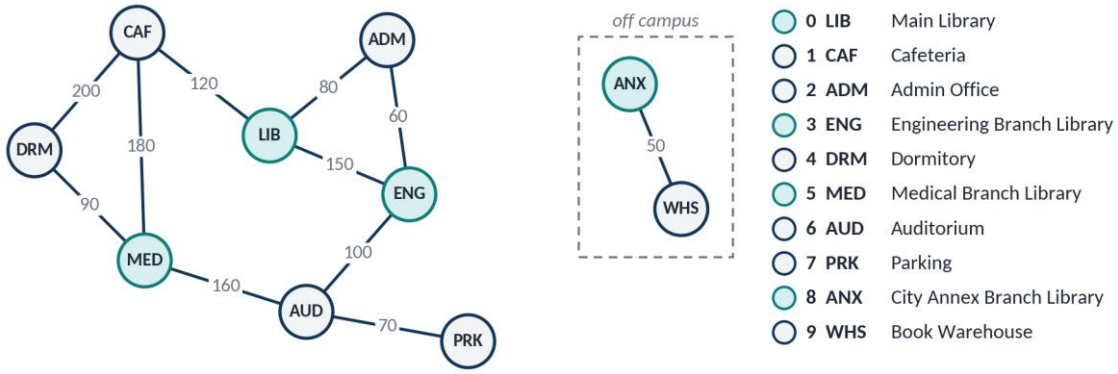


Figure 13.3 The Pustaka Ceria campus map, the running example of this chapter: ten buildings (libraries in teal) and eleven footpaths with their lengths in meters. The city annex and the warehouse are off campus and not yet connected to it.

Several features of the map were chosen so that each algorithm has something to show.

- **It is undirected and weighted.** Footpaths can be walked both ways, and each has a length. Sections 13.6 to 13.8 ignore the lengths; Section 13.10 uses them.
- **It contains cycles.** LIB, ADM, ENG, LIB is a triangle, CAF, DRM, MED, CAF is another, and the long loop LIB, CAF, MED, AUD, ENG, LIB goes around the middle. Any search that forgot which buildings it had visited would circle forever.
- **It is disconnected.** The city annex (ANX) and the warehouse (WHS) are joined to each other by a 50-meter path but to nothing on campus. The map has two connected components, and Section 13.8 joins them with a shuttle road.
- **Fewest hops and fewest meters disagree.** From LIB to PRK, the route LIB, ENG, AUD, PRK uses 3 footpaths and $150 + 100 + 70 = 320$ meters, while LIB, ADM, ENG, AUD, PRK uses 4 footpaths but only $80 + 60 + 100 + 70 = 310$ meters. Breadth-first search (Section 13.6) finds the first; Dijkstra's algorithm (Section 13.10) finds the second.

Let us apply the vocabulary of Section 13.3 to the map. $|V| = 10$ and $|E| = 11$. The degrees are $\deg(\text{LIB}) = 3$, $\deg(\text{CAF}) = 3$, $\deg(\text{ADM}) = 2$, $\deg(\text{ENG}) = 3$, $\deg(\text{DRM}) = 2$, $\deg(\text{MED}) = 3$, $\deg(\text{AUD}) = 3$, $\deg(\text{PRK}) = 1$, $\deg(\text{ANX}) = 1$ and $\deg(\text{WHS}) = 1$. Their sum is 22, which is 2×11, as Equation (13.2) demands. Eight vertices (LIB, CAF, ENG, MED, AUD, PRK, ANX and WHS) have odd degree, an even number of them, as the first consequence of the lemma demands. By Equation (13.4) a simple graph on 10 vertices can have at most $10 \times 9/2 = 45$ edges, so by Equation (13.5) the density of the map is $\rho = 11/45 \approx 0.24$. The map is small, but already three quarters of the possible footpaths are missing, and real maps are far sparser than that.

13.5 Representing a Graph in Memory

A drawing is for people. A program needs the graph in memory, in a form that answers its questions quickly. The two questions every graph algorithm asks, over and over, are:

1. *Is there an edge from u to v, and if so, what is its weight?*
2. *What are all the neighbors of u?*

There are three classic representations, and each answers these questions at a different price. **Figure 13.4** shows all three for the campus map.

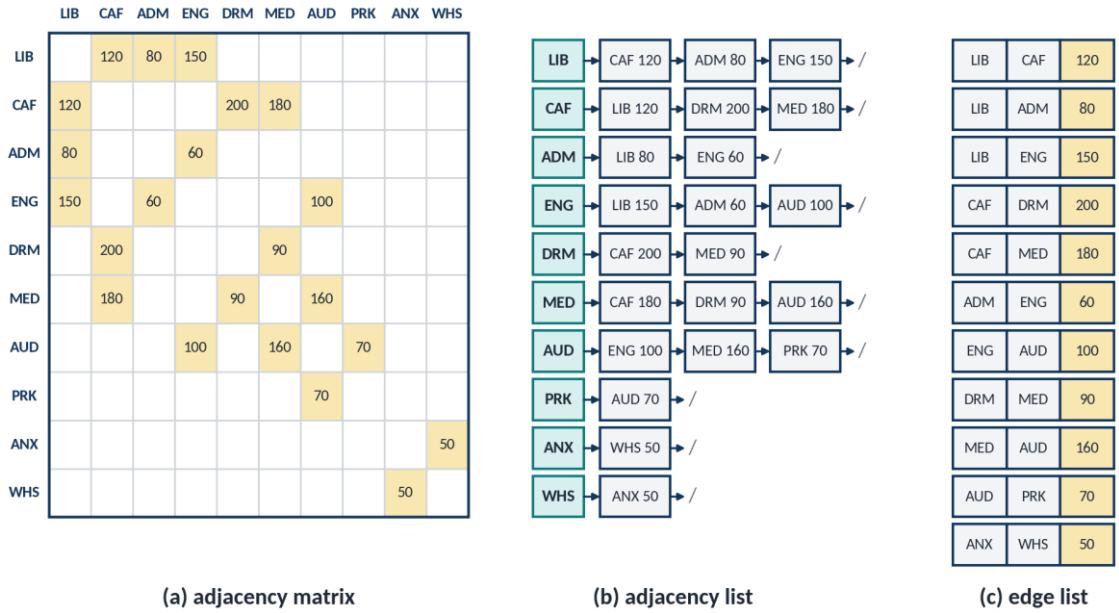


Figure 13.4 The campus map stored three ways: (a) the adjacency matrix, with footpath lengths in the shaded cells; (b) the adjacency list, one chain of (neighbor, length) entries per vertex; (c) the edge list, one record per footpath.

13.5.1 The adjacency matrix

Number the vertices $0, 1, \ldots, |V| - 1$. The *adjacency matrix* is a $|V| \times |V|$ table `mat` in which the cell in row u and column v describes the edge from u to v. For an unweighted graph the cell holds 1 if the edge exists and 0 if not; for the campus map it holds the length of the footpath, with 0 meaning "no footpath" (a safe choice because every real length is positive). **Figure 13.4**(a) shows the matrix. Row LIB has entries in columns CAF (120), ADM (80) and ENG (150), the three neighbors of the main library, and nothing else.

Three properties can be read directly off the picture.

- **The diagonal is empty**, because a simple graph has no self-loops.
- **The matrix is symmetric**: the cell in row u, column v equals the cell in row v, column u, because an undirected edge is stored in both directions. A directed graph fills only the cell for its arrow, so its matrix is in general not symmetric.
- **Most cells are empty.** Of the 100 cells, only 22 are shaded, two for each of the 11 footpaths.

The matrix answers question 1 in a single step: to know whether ENG and AUD are joined, look at one cell, which takes $O(1)$ time, exactly like indexing an array (Chapter 1). Question 2 is slower: to list the neighbors of u, the program must scan the whole row u, all $|V|$ cells, even if u has only one neighbor. So listing the neighbors of u takes $\Theta(|V|)$ time, and computing a degree takes

$\Theta(|V|)$ time too. The memory is $|V|^2$ cells whatever the number of edges; for a graph with $|V| = 10^6$ vertices, that is 10^{12} cells, a terabyte even at one byte per cell.

13.5.2 The adjacency list

The *adjacency list* stores, for each vertex u, only the neighbors that u actually has: an array `adj` of $|V|$ lists, where `adj[u]` holds one entry per edge leaving u. For a weighted graph each entry is a pair (neighbor, weight). **Figure 13.4**(b) draws each list as a chain in the style of Chapter 7, and a linked list is indeed one way to implement it; the programs of this chapter use a growable array instead (`std::vector` in C++, `ArrayList` in Java, `list` in Python), which is faster to walk for the cache reasons Section 7.3 explained. An undirected edge $\{u, v\}$ is stored twice, once in `adj[u]` and once in `adj[v]`, so the lists together hold $2|E|$ entries, which by Equation (13.2) is exactly the sum of all degrees.

Now the prices are reversed. Question 2 is as cheap as it can be: the neighbors of u are simply the entries of `adj[u]`, listed in $\Theta(\deg(u))$ time, and the degree is the length of the list, available in $O(1)$. Question 1 costs a scan of `adj[u]`, which is $O(\deg(u))$: cheap when degrees are small, as they are in road maps, and slow for a vertex with millions of neighbors, such as a celebrity in a social network. The memory is one list header per vertex plus one entry per edge end, $\Theta(|V| + |E|)$ in total.

13.5.3 The edge list

The simplest representation of all is the *edge list*: one record (u, v, w) per edge, in any order, as in **Figure 13.4**(c). It is how graphs usually arrive: a file of road segments, a table of friendships, the list of footpaths typed in by a librarian. It uses $\Theta(|E|)$ memory, storing each undirected edge only once. But it answers both questions slowly: to find whether u and v are joined, or to list the neighbors of u, a program must scan all $|E|$ records. Algorithms that visit the edges in one pass, in whatever order they come, are happy with an edge list; the minimum spanning tree algorithm of Kruskal, which first sorts all edges by weight, is the classic example [2]. The traversals of this chapter are not, and every program here converts the edge list into an adjacency list first, which takes $\Theta(|V| + |E|)$ time.

13.5.4 The costs side by side

The memory needed by the two main representations of an undirected graph is

$$S_{\text{matrix}} = |V|^2 \text{ cells}, \qquad S_{\text{list}} = |V| + 2|E| \text{ entries} \tag{13.6}$$

For the campus map, $S_{\text{matrix}} = 100$ cells, $S_{\text{list}} = 10 + 22 = 32$ (ten list headers and 22 entries), and the edge list needs 11 records. When is the matrix the smaller? Setting $|V|^2 < |V| + 2|E|$ and using $|E| \le |V|(|V| - 1)/2$ from Equation (13.4), the matrix can only win when $|E|$ is close to its maximum, that is, for dense graphs; and since a matrix cell of one bit is smaller than a list entry of 8 bytes, the break-even point for the pure 0/1 matrix is around a density of $1/64$ of the maximum when entries are 8 bytes wide. **Table 13.1** collects every cost.

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

The row that decides the matter for this chapter is “visit every edge once”. Breadth-first search, depth-first search, component labeling, topological sort and Dijkstra’s algorithm all do essentially one thing, over and over: take a vertex and look at all its neighbors. With an adjacency list, looking at the neighbors of every vertex once costs $\sum_u \Theta\,(1+\deg(u)) = \Theta(V+E)$ by Equation (13.2). With a matrix, the same work costs $\sum_u \Theta\,(V) = \Theta(V^2)$, because every row is scanned in full, empty cells included. On the campus map that is 32 steps against 100; on a road network of 24 million junctions (Section 13.11), it is the difference between a fraction of a second and many hours. This is why every traversal in this chapter is written against the adjacency list.

> **RULE OF THUMB.** Use an adjacency list unless the graph is dense or the algorithm mostly asks “is there an edge between u and v?”. Keep an edge list when the graph arrives as a list of edges and the algorithm processes edges one by one.

13.5.5 The program in C++

Listing 13.1 builds all three representations of the campus map from the same list of eleven footpaths, prints them, computes every degree twice (from the matrix row and from the list length, and complains if they differ), checks the handshake lemma, answers the edge question “is there a footpath ENG-AUD, and LIB-DRM?” in all three, and reports the memory each one used.

Listing 13.1 `graph_repr.cpp`: the campus map as an adjacency matrix, an adjacency list and an edge list.

```cpp
// graph_repr.cpp -- the Pustaka Ceria campus map stored three ways:
// adjacency matrix, adjacency list and edge list.
#include <cstdio>
#include <vector>

const int V = 10;
const char *CODE[V] = {"LIB", "CAF", "ADM", "ENG", "DRM",
                       "MED", "AUD", "PRK", "ANX", "WHS"};

struct Edge {            // one footpath: endpoints u, v and length in meters
    int u, v, w;
};

// The eleven footpaths of the campus map, in the order they are added.
const std::vector<Edge> PATHS = {
    {0, 1, 120}, {0, 2, 80},  {0, 3, 150}, {1, 4, 200}, {1, 5, 180}, {2, 3, 60},
    {3, 6, 100}, {4, 5, 90},  {5, 6, 160}, {6, 7, 70},  {8, 9, 50}};

struct Nbr {             // one entry of an adjacency list
    int to, w;
};

int main() {
    // 1. Adjacency matrix: mat[u][v] = length of the path u-v, 0 = no path.
    std::vector<std::vector<int>> mat(V, std::vector<int>(V, 0));
    // 2. Adjacency list: adj[u] holds one entry per neighbor of u.
    std::vector<std::vector<Nbr>> adj(V);
    // 3. Edge list: simply the edges themselves.
    std::vector<Edge> edges;

    for (const Edge &e : PATHS) {
        mat[e.u][e.v] = e.w;             // undirected: fill both cells
        mat[e.v][e.u] = e.w;
        adj[e.u].push_back({e.v, e.w});  // undirected: both lists
        adj[e.v].push_back({e.u, e.w});
        edges.push_back(e);              // stored once
    }
```

Listing 13.1 (Continued) *graph_repr.cpp: the campus map as an adjacency matrix, an adjacency list and an edge list.*

```
    std::printf("Adjacency matrix (meters, . = no footpath)\n     ");
    for (int v = 0; v < V; ++v) std::printf("%5s", CODE[v]);
    std::printf("\n");
    for (int u = 0; u < V; ++u) {
        std::printf("%5s", CODE[u]);
        for (int v = 0; v < V; ++v) {
            if (mat[u][v] == 0) std::printf("%5s", ".");
            else std::printf("%5d", mat[u][v]);
        }
        std::printf("\n");
    }

    std::printf("\nAdjacency list\n");
    for (int u = 0; u < V; ++u) {
        std::printf("  %s:", CODE[u]);
        for (const Nbr &n : adj[u]) std::printf(" %s(%d)", CODE[n.to], n.w);
        std::printf("\n");
    }

    std::printf("\nEdge list");
    for (std::size_t i = 0; i < edges.size(); ++i) {
        if (i % 6 == 0) std::printf("\n ");   // six edges per printed line
        std::printf(" %s-%s(%d)", CODE[edges[i].u], CODE[edges[i].v], edges[i].w);
    }
    std::printf("\n");

    // Degree: count a row of the matrix, or read the size of a list.
    std::printf("\nDegrees:");
    int sum = 0;
    for (int u = 0; u < V; ++u) {
        int dm = 0;
        for (int v = 0; v < V; ++v) if (mat[u][v] != 0) ++dm;   // O(V)
        int dl = static_cast<int>(adj[u].size());                 // O(1)
        if (dm != dl) std::printf(" MISMATCH");
        std::printf(" %s=%d", CODE[u], dl);
        sum += dl;
    }
    std::printf("\nSum of degrees = %d = 2 x %zu edges\n", sum, edges.size());
```

Listing 13.1 (Continued) *graph_repr.cpp: the campus map as an adjacency matrix, an adjacency list and an edge list.*

```
    // Is there a footpath ENG-AUD? And LIB-DRM?
    int qs[2][2] = {{3, 6}, {0, 4}};
    for (auto &q : qs) {
        int u = q[0], v = q[1];
        bool inMatrix = mat[u][v] != 0;               // one cell: O(1)
        bool inList = false;                          // scan u's list: O(deg u)
        for (const Nbr &n : adj[u]) if (n.to == v) inList = true;
        bool inEdges = false;                         // scan all edges: O(E)
        for (const Edge &e : edges)
            if ((e.u == u && e.v == v) || (e.u == v && e.v == u)) inEdges = true;
        std::printf("Footpath %s-%s? matrix: %s, list: %s, edge list: %s\n",
                    CODE[u], CODE[v], inMatrix ? "yes" : "no",
                    inList ? "yes" : "no", inEdges ? "yes" : "no");
    }

    std::printf("\nStorage: matrix %d cells, list %d entries, "
                "edge list %zu records\n", V * V, sum, edges.size());
    return 0;
}
```

```
Adjacency matrix (meters, . = no footpath)
       LIB  CAF  ADM  ENG  DRM  MED  AUD  PRK  ANX  WHS
  LIB    .  120   80  150    .    .    .    .    .    .
  CAF  120    .    .    .  200  180    .    .    .    .
  ADM   80    .    .   60    .    .    .    .    .    .
  ENG  150    .   60    .    .    .  100    .    .    .
  DRM    .  200    .    .    .   90    .    .    .    .
  MED    .  180    .    .   90    .  160    .    .    .
  AUD    .    .    .  100    .  160    .   70    .    .
  PRK    .    .    .    .    .    .   70    .    .    .
  ANX    .    .    .    .    .    .    .    .    .   50
  WHS    .    .    .    .    .    .    .    .   50    .

Adjacency list
  LIB: CAF(120) ADM(80) ENG(150)
  CAF: LIB(120) DRM(200) MED(180)
  ADM: LIB(80) ENG(60)
  ENG: LIB(150) ADM(60) AUD(100)
  DRM: CAF(200) MED(90)
  MED: CAF(180) DRM(90) AUD(160)
  AUD: ENG(100) MED(160) PRK(70)
  PRK: AUD(70)
  ANX: WHS(50)
  WHS: ANX(50)

Edge list
  LIB-CAF(120) LIB-ADM(80) LIB-ENG(150) CAF-DRM(200) CAF-MED(180) ADM-ENG(60)
  ENG-AUD(100) DRM-MED(90) MED-AUD(160) AUD-PRK(70) ANX-WHS(50)

Degrees: LIB=3 CAF=3 ADM=2 ENG=3 DRM=2 MED=3 AUD=3 PRK=1 ANX=1 WHS=1
Sum of degrees = 22 = 2 x 11 edges
Footpath ENG-AUD? matrix: yes, list: yes, edge list: yes
Footpath LIB-DRM? matrix: no, list: no, edge list: no

Storage: matrix 100 cells, list 22 entries, edge list 11 records
```

A few details deserve comment. The footpaths are written once, as the array `PATHS` of `Edge` records, and everything else is derived from it; this is the usual way to build a graph, because edge lists are how graphs arrive. The matrix is a `std::vector` of `std::vector<int>`, $|V|$ rows of $|V|$ zeros, so that `mat[u][v]` reads like the mathematics. The adjacency list `adj` is a `std::vector` of `std::vector<Nbr>`; `push_back` appends a neighbor in amortized $O(1)$ time, the doubling trick of Section 5.5.1. The order of the entries in each list is the order in which the footpaths were added, which is why `adj[LIB]` reads CAF, ADM, ENG. That order is arbitrary, but it matters: it decides the order in which the searches of the next sections visit neighbors, and therefore their exact output.

The output confirms the counts of Section 13.4: degrees 3, 3, 2, 3, 2, 3, 3, 1, 1, 1, sum 22, twice the 11 edges. The last line puts Equation (13.6) into numbers: 100 cells, 22 list entries (plus the ten list headers), 11 edge records.

13.5.6 The same program in Java and Python

Listing 13.2 and **Listing 13.3** build the same three representations. In Java, the edge and the neighbor entry are `record` types, compact immutable classes available since Java 16; in Python, plain tuples serve the same purpose.

Listing 13.2 `GraphRepr.java`: the three representations in Java.

```java
// GraphRepr.java -- the Pustaka Ceria campus map stored three ways:
// adjacency matrix, adjacency list and edge list.
import java.util.ArrayList;
import java.util.List;

public class GraphRepr {
    static final int V = 10;
    static final String[] CODE = {"LIB", "CAF", "ADM", "ENG", "DRM",
                                  "MED", "AUD", "PRK", "ANX", "WHS"};

    record Edge(int u, int v, int w) {}     // one footpath, length w meters
    record Nbr(int to, int w) {}            // one entry of an adjacency list

    static final Edge[] PATHS = {
        new Edge(0, 1, 120), new Edge(0, 2, 80), new Edge(0, 3, 150),
        new Edge(1, 4, 200), new Edge(1, 5, 180), new Edge(2, 3, 60),
        new Edge(3, 6, 100), new Edge(4, 5, 90), new Edge(5, 6, 160),
        new Edge(6, 7, 70), new Edge(8, 9, 50)};

    public static void main(String[] args) {
        int[][] mat = new int[V][V];                // 0 = no footpath
        List<List<Nbr>> adj = new ArrayList<>();
        for (int u = 0; u < V; u++) adj.add(new ArrayList<>());
        List<Edge> edges = new ArrayList<>();

        for (Edge e : PATHS) {
            mat[e.u()][e.v()] = e.w();              // undirected: both cells
            mat[e.v()][e.u()] = e.w();
            adj.get(e.u()).add(new Nbr(e.v(), e.w()));   // and both lists
            adj.get(e.v()).add(new Nbr(e.u(), e.w()));
            edges.add(e);                           // stored once
        }

        System.out.print("Adjacency matrix (meters, . = no footpath)\n     ");
        for (int v = 0; v < V; v++) System.out.printf("%5s", CODE[v]);
        System.out.println();
        for (int u = 0; u < V; u++) {
            System.out.printf("%5s", CODE[u]);
            for (int v = 0; v < V; v++) {
                if (mat[u][v] == 0) System.out.printf("%5s", ".");
                else System.out.printf("%5d", mat[u][v]);
            }
            System.out.println();
        }
```

Listing 13.2 (Continued) *GraphRepr.java: the three representations in Java.*

```
        System.out.println("\nAdjacency list");
        for (int u = 0; u < V; u++) {
            System.out.print("  " + CODE[u] + ":");
            for (Nbr n : adj.get(u))
                System.out.printf(" %s(%d)", CODE[n.to()], n.w());
            System.out.println();
        }

        System.out.print("\nEdge list");
        for (int i = 0; i < edges.size(); i++) {
            if (i % 6 == 0) System.out.print("\n ");  // six edges per printed line
            Edge e = edges.get(i);
            System.out.printf(" %s-%s(%d)", CODE[e.u()], CODE[e.v()], e.w());
        }
        System.out.println();

        System.out.print("\nDegrees:");
        int sum = 0;
        for (int u = 0; u < V; u++) {
            int dm = 0;
            for (int v = 0; v < V; v++) if (mat[u][v] != 0) dm++;   // O(V)
            int dl = adj.get(u).size();                              // O(1)
            if (dm != dl) System.out.print(" MISMATCH");
            System.out.printf(" %s=%d", CODE[u], dl);
            sum += dl;
        }
        System.out.printf("%nSum of degrees = %d = 2 x %d edges%n",
                sum, edges.size());

        int[][] qs = {{3, 6}, {0, 4}};             // ENG-AUD? LIB-DRM?
        for (int[] q : qs) {
            int u = q[0], v = q[1];
            boolean inMatrix = mat[u][v] != 0;
            boolean inList = false;
            for (Nbr n : adj.get(u)) if (n.to() == v) inList = true;
            boolean inEdges = false;
            for (Edge e : edges)
                if ((e.u() == u && e.v() == v) || (e.u() == v && e.v() == u))
                    inEdges = true;
            System.out.printf(
                    "Footpath %s-%s? matrix: %s, list: %s, edge list: %s%n",
                    CODE[u], CODE[v], inMatrix ? "yes" : "no",
                    inList ? "yes" : "no", inEdges ? "yes" : "no");
        }
```

Listing 13.2 (Continued) *GraphRepr.java: the three representations in Java.*

```
        System.out.printf("%nStorage: matrix %d cells, list %d entries, "
                + "edge list %d records%n", V * V, sum, edges.size());
    }
}
```

Listing 13.3 `graph_repr.py`: the three representations in Python.

```
# graph_repr.py -- the Pustaka Ceria campus map stored three ways:
# adjacency matrix, adjacency list and edge list.

V = 10
CODE = ["LIB", "CAF", "ADM", "ENG", "DRM", "MED", "AUD", "PRK", "ANX", "WHS"]
# The eleven footpaths (u, v, meters), in the order they are added.
PATHS = [(0, 1, 120), (0, 2, 80), (0, 3, 150), (1, 4, 200), (1, 5, 180),
         (2, 3, 60), (3, 6, 100), (4, 5, 90), (5, 6, 160), (6, 7, 70),
         (8, 9, 50)]


def main():
    mat = [[0] * V for _ in range(V)]        # 0 = no footpath
    adj = [[] for _ in range(V)]             # adj[u] = list of (neighbor, meters)
    edges = []
    for u, v, w in PATHS:
        mat[u][v] = mat[v][u] = w            # undirected: both cells
        adj[u].append((v, w))                # and both lists
        adj[v].append((u, w))
        edges.append((u, v, w))              # stored once

    print("Adjacency matrix (meters, . = no footpath)")
    print("     " + "".join(f"{c:>5}" for c in CODE))
    for u in range(V):
        cells = "".join(f"{'.' if x == 0 else x:>5}" for x in mat[u])
        print(f"{CODE[u]:>5}{cells}")

    print("\nAdjacency list")
    for u in range(V):
        print(f"  {CODE[u]}:" + "".join(f" {CODE[v]}({w})" for v, w in adj[u]))

    print("\nEdge list", end="")
    for i, (u, v, w) in enumerate(edges):
        if i % 6 == 0:
            print("\n ", end="")             # six edges per printed line
        print(f" {CODE[u]}-{CODE[v]}({w})", end="")
    print()
```

Listing 13.3 (Continued) *graph_repr.py: the three representations in Python.*

```
    print("\nDegrees:", end="")
    total = 0
    for u in range(V):
        dm = sum(1 for x in mat[u] if x != 0)    # O(V)
        dl = len(adj[u])                         # O(1)
        if dm != dl:
            print(" MISMATCH", end="")
        print(f" {CODE[u]}={dl}", end="")
        total += dl
    print(f"\nSum of degrees = {total} = 2 x {len(edges)} edges")

    for u, v in [(3, 6), (0, 4)]:            # ENG-AUD? LIB-DRM?
        in_matrix = mat[u][v] != 0
        in_list = any(x == v for x, _ in adj[u])
        in_edges = any({a, b} == {u, v} for a, b, _ in edges)
        yn = lambda b: "yes" if b else "no"
        print(f"Footpath {CODE[u]}-{CODE[v]}? matrix: {yn(in_matrix)}, "
              f"list: {yn(in_list)}, edge list: {yn(in_edges)}")

    print(f"\nStorage: matrix {V * V} cells, list {total} entries, "
          f"edge list {len(edges)} records")


if __name__ == "__main__":
    main()
```

Both print exactly the output of **Listing 13.1**. One remark about Python: `[[0] * V for _ in range(V)]` creates V separate rows. The shorter-looking `[[0] * V] * V` would create *one* row and V references to it, so that writing `mat[0][1]` would appear to change every row at once, a classic trap. Another about memory: a Python list of lists stores a pointer per cell and an object per distinct number, so a Python matrix costs far more than $|V|^2$ bytes. For large graphs in Python, the matrix belongs in a NumPy array or, far more often, is not built at all.

13.6 Breadth-First Search

13.6.1 The idea: ripples in a pond

Drop a stone into a still pond. A ring of ripples spreads out, and every point of the water is reached by the ring at a time proportional to its distance from the stone. *Breadth-first search* (BFS) spreads through a graph the same way. Starting from a source vertex s, it first visits s, then all the neighbors of s (the vertices one edge away), then all the vertices two edges away, then three, and so on, finishing each ring before it starts the next. The rings are called *layers* or *levels*, and the number of the ring a vertex lies in is its *hop distance* from s: the fewest number of edges on any path from s to it.

How does a program finish one ring before starting the next? With the queue of Chapter 5. Vertices are put into the queue in the order they are discovered and taken out in the same order, first in, first out. All the vertices of ring 1 are discovered while the source is being processed, so they enter the queue before any vertex of ring 2; the vertices of ring 2 are discovered while ring 1 is being processed, so they enter before any vertex of ring 3; and so on. The queue quietly keeps the rings in order. The algorithm was published independently by Edward F. Moore in 1959, who used it to find the shortest path through a maze [3], and by C. Y. Lee in 1961, who used it to route the wires on circuit boards [4].

13.6.2 The procedure

BFS keeps two numbers for every vertex v: $d[v]$, the hop distance from the source found so far, with -1 (or ∞) meaning "not yet discovered", and $\pi[v]$, the *parent* of v, the vertex from which v was discovered, with NIL for "none". **Pseudocode 13.1** gives the procedure; $G.adj[u]$ is the adjacency list of u and Q is a queue.

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)
```

Three points about **Pseudocode 13.1** are easy to miss.

- **A vertex is marked when it enters the queue, not when it leaves.** Line 11 sets $v.d$ the moment v is first seen, and line 10 refuses any vertex whose d is already set. So every vertex enters the queue at most once. If the mark were set only at line 8, when a vertex is dequeued, a vertex with several already-visited neighbors could be enqueued several times; on the campus map, AUD would be enqueued both from ENG and from MED. The answer would still come out right, but the queue could hold up to $|E|$ entries instead of $|V|$, and the work would grow accordingly.
- $d[v]$ **doubles as the "visited" mark.** A separate Boolean array would work equally well; using d saves memory and keeps the two facts, "seen" and "how far", from ever disagreeing.
- **The parents form a tree.** Every discovered vertex other than s gets exactly one parent, the vertex that discovered it, and that parent was discovered earlier. Following parents from any vertex therefore climbs, one edge at a time, back to s without ever repeating a vertex. The edges $\{\pi[v], v\}$ form a tree rooted at s, the *BFS tree*, and the path in this tree from s to v is a shortest path in hops (Section 13.6.4 proves it).

13.6.3 Tracing BFS from the main library

Let us run **Pseudocode 13.1** on the campus map with source LIB, looking at neighbors in the order of the adjacency list of **Figure 13.4**(b). **Table 13.2** records every step: which vertex leaves the queue, which of its neighbors are discovered (with their new distance and parent), and what the queue holds afterward.

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)

Read the steps slowly. At step 1 the three neighbors of LIB are discovered together and enter the queue in list order. At step 2 CAF is dequeued; its neighbor LIB is already discovered and is ignored, while DRM and MED are new and join the *back* of the queue, behind ADM and ENG, which are still waiting. That is the queue keeping the rings apart: ADM and ENG (ring 1) will be processed before DRM and MED (ring 2), although DRM and MED were discovered first. At step 3, ADM's neighbors LIB and ENG are both already known, so nothing happens; notice that ENG is *in the queue*, not yet processed, and is still correctly refused, because it was marked when it entered. Step 6 is the interesting refusal: MED sees AUD, which ENG discovered one step earlier, and leaves it alone. The queue empties after step 8, and the search stops.

Figure 13.5 shows the queue after every step as a strip of cells, with the vertices that have just entered in amber. **Figure 13.6** redraws the map by rings: LIB at distance 0; CAF, ADM and ENG at 1; DRM, MED and AUD at 2; PRK at 3. The amber edges are the BFS tree, one edge from each

vertex to its parent; the dashed grey edges are the footpaths that BFS looked at and refused because both ends were already discovered. Two vertices, ANX and WHS, never enter the queue at all: no path leads to them from LIB, so their distance stays −1 (drawn dashed at the right).

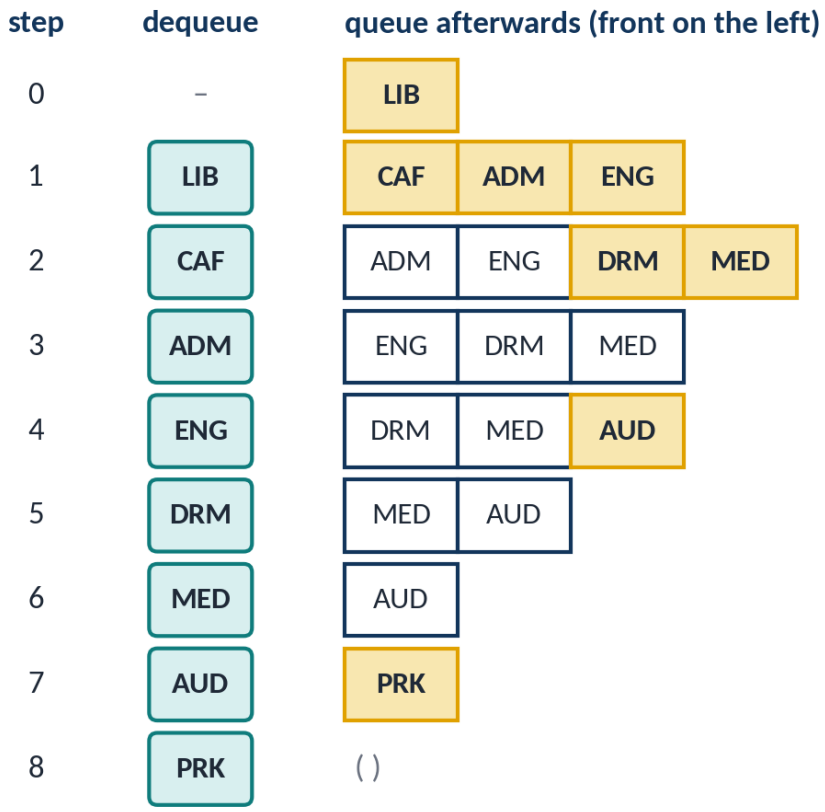


Figure 13.5 The BFS queue after each step of **Table 13.2**: the dequeued vertex on the left, the queue contents on the right with newly enqueued vertices in amber.

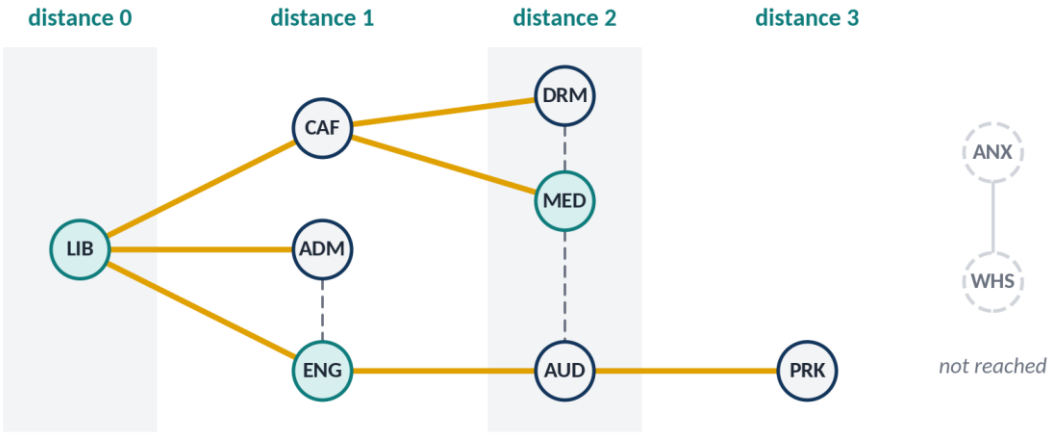


Figure 13.6 The campus map arranged by hop distance from LIB. Amber edges form the BFS tree; dashed edges were examined but led to vertices already discovered; ANX and WHS are not reached.

The dashed edges of **Figure 13.6** are revealing. ADM-ENG and DRM-MED join two vertices of the *same* ring, and MED-AUD joins two vertices of ring 2. No edge skips a ring: no edge joins ring 0 to ring 2, or ring 1 to ring 3. That is not an accident of this map but a law of BFS, and it is the heart of the proof that follows.

13.6.4 Why BFS finds fewest-hop routes

Write $\delta(s, v)$ for the true hop distance, the fewest edges on any path from s to v (infinite if there is none). We want to show that BFS computes $d[v] = \delta(s, v)$ for every v. The argument has three short steps; Cormen et al. give a fully formal version [2].

Step 1: $d[v]$ is never too small. Whenever line 11 sets $v.d \leftarrow u.d + 1$, the parent chain of v is an actual path from s of exactly $v.d$ edges. A real path of $d[v]$ edges exists, so the shortest path has at most that many: $\delta(s, v) \leq d[v]$.

Step 2: the queue is always sorted by d, with at most two neighboring values. At every moment, if the queue holds $v_1, v_2, \ldots, v_r$ from front to back, then $d[v_1] \leq d[v_2] \leq \cdots \leq d[v_r] \leq d[v_1] + 1$. It holds at the start (one vertex). A dequeue removes the front and cannot break it. An enqueue appends a vertex with $d = d[u] + 1$, where u was the old front, so $d[u]$ was at most every value in the queue, and the new value is at least every value in the queue and at most $d[v_2] + 1$. In the trace this is visible in the last column of **Table 13.2**: the queue always holds one ring, or the end of one ring followed by the start of the next. It follows that vertices are dequeued in nondecreasing order of d.

Step 3: $d[v]$ is never too large. Suppose some reachable vertex got $d[v] > \delta(s, v)$, and take such a v with the smallest $\delta(s, v)$. Let u be the vertex just before v on a shortest path, so $\delta(s, u) = \delta(s, v) - 1$, and by the choice of v, $d[u] = \delta(s, u)$. When u was dequeued, BFS looked at the edge to v. If v was undiscovered, it got $d[v] = d[u] + 1 = \delta(s, v)$, a contradiction. If v had already been discovered from some earlier x, then by Step 2 x was dequeued no later than u, so $d[x] \leq d[u]$, and $d[v] = d[x] + 1 \leq d[u] + 1 = \delta(s, v)$, again a contradiction. So no such v exists. Every edge (u, v) of the graph with u reachable therefore satisfies

$$d[v] \leq d[u] + 1 \tag{13.7}$$

which is the "no edge skips a ring" law we saw in **Figure 13.6**. Together, Steps 1 and 3 give $d[v] = \delta(s, v)$, and walking the parents from v back to s gives a path of exactly that many edges, a shortest one.

> **IN PLAIN LANGUAGE.** The queue makes BFS finish every ring before it starts the next, so the first time a vertex is reached is along a route with as few edges as possible. Remember who reached each vertex first, and you can read the route backward.

13.6.5 The running time

Count the work of **Pseudocode 13.1** on a graph stored as adjacency lists. Lines 1 to 3 touch every vertex once: $\Theta(V)$. Every vertex is enqueued at most once (because it is marked on entry) and so dequeued at most once, and each queue operation is $O(1)$ (Section 5.13): $O(V)$ in total. When a vertex u is dequeued, the loop of line 9 runs $\deg(u)$ times in an undirected graph (or $\deg^+(u)$ times in a directed one), each iteration $O(1)$. Summed over all vertices, by Equation (13.2),

$$T_{\text{BFS}} = \Theta(V) + \sum_{u \in V} O\left(1 + \deg(u)\right) = O(V + E) \tag{13.8}$$

This is called *linear time* for graphs: it is proportional to the size of the input, the vertices plus the edges, and no algorithm that looks at the whole graph can do better. With an adjacency matrix, line 9 would scan a whole row of $|V|$ cells for every dequeued vertex, and BFS would cost $\Theta(V^2)$

instead, which is why **Table 13.1** recommends the list. The extra memory is the arrays d and π and a queue of at most $|V|$ vertices: $\Theta(V)$.

13.6.6 The program in C++

Listing 13.4 implements **Pseudocode 13.1** with `std::queue`, the queue adaptor of Section 5.16.1. The function `bfs` fills `dist` and `parent` and returns the visiting order; with `trace` switched on it prints one line per dequeued vertex, reproducing **Table 13.2**. The function `printPath` reconstructs a route by walking the parents from the target back to the source and pushing each vertex onto the *front* of a `std::deque`, so that the route comes out in the forward direction.

Listing 13.4 `bfs.cpp`: breadth-first search on the campus map, with a trace, hop distances, parents and fewest-hop routes.

```
// bfs.cpp -- breadth-first search on the Pustaka Ceria campus map:
// visiting order, hop distances, parents, and fewest-hop routes.
#include <cstdio>
#include <deque>
#include <queue>
#include <string>
#include <vector>

const int V = 10;
const char *CODE[V] = {"LIB", "CAF", "ADM", "ENG", "DRM",
                       "MED", "AUD", "PRK", "ANX", "WHS"};
const int PATHS[][3] = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                        {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                        {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

std::vector<std::vector<int>> buildCampus() {   // weights are not needed by BFS
    std::vector<std::vector<int>> adj(V);
    for (const auto &p : PATHS) {
        adj[p[0]].push_back(p[1]);
        adj[p[1]].push_back(p[0]);
    }
    return adj;
}

std::string show(const std::queue<int> &q) {    // print a copy, front first
    std::queue<int> c = q;
    std::string s = "[";
    while (!c.empty()) {
        s += CODE[c.front()];
        c.pop();
        if (!c.empty()) s += " ";
    }
    return s + "]";
}

// Fills dist (-1 = not reached) and parent (-1 = none); returns visiting order.
std::vector<int> bfs(const std::vector<std::vector<int>> &adj, int s,
                     std::vector<int> &dist, std::vector<int> &parent, bool trace) {
    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()) {
```

Listing 13.4 (Continued) *bfs.cpp: breadth-first search on the campus map, with a trace, hop distances, parents and fewest-hop routes.*

```
        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];
            }
        }
        if (trace)
            std::printf("%4d  %s(d=%d)  enqueued:%-12s queue now %s\n", ++step,
                        CODE[u], dist[u], added.empty() ? " -" : added.c_str(),
                        show(q).c_str());
    }
    return order;
}

void printPath(const std::vector<int> &parent, const std::vector<int> &dist,
               int t) {
    if (dist[t] == -1) {
        std::printf("  to %s: no route\n", CODE[t]);
        return;
    }
    std::deque<int> path;        // walk the parents back from t, then read forward
    for (int v = t; v != -1; v = parent[v]) path.push_front(v);
    std::printf("  to %s (%d hops):", CODE[t], dist[t]);
    for (int v : path) std::printf(" %s", CODE[v]);
    std::printf("\n");
}

int main() {
    auto adj = buildCampus();
    std::vector<int> dist, parent;
    std::printf("BFS from LIB\nstep  dequeued\n");
    std::vector<int> order = bfs(adj, 0, dist, parent, true);
```

Listing 13.4 (Continued) *bfs.cpp: breadth-first search on the campus map, with a trace, hop distances, parents and fewest-hop routes.*

```
    std::printf("\nVisiting order:");
    for (int v : order) std::printf(" %s", CODE[v]);
    std::printf("\n\nvertex  dist  parent\n");
    for (int v = 0; v < V; ++v) {
        if (dist[v] == -1) std::printf("  %s     -     -\n", CODE[v]);
        else std::printf("  %s  %4d  %5s\n", CODE[v], dist[v],
                         parent[v] == -1 ? "-" : CODE[parent[v]]);
    }
    std::printf("\nFewest-hop routes from LIB\n");
    printPath(parent, dist, 7);  // PRK
    printPath(parent, dist, 4);  // DRM
    printPath(parent, dist, 9);  // WHS
    return 0;
}
```

```
BFS from LIB
step  dequeued
   1  LIB(d=0)  enqueued: CAF ADM ENG queue now [CAF ADM ENG]
   2  CAF(d=1)  enqueued: DRM MED     queue now [ADM ENG DRM MED]
   3  ADM(d=1)  enqueued: -           queue now [ENG DRM MED]
   4  ENG(d=1)  enqueued: AUD         queue now [DRM MED AUD]
   5  DRM(d=2)  enqueued: -           queue now [MED AUD]
   6  MED(d=2)  enqueued: -           queue now [AUD]
   7  AUD(d=2)  enqueued: PRK         queue now [PRK]
   8  PRK(d=3)  enqueued: -           queue now []

Visiting order: LIB CAF ADM ENG DRM MED AUD PRK

vertex  dist  parent
  LIB     0      -
  CAF     1    LIB
  ADM     1    LIB
  ENG     1    LIB
  DRM     2    CAF
  MED     2    CAF
  AUD     2    ENG
  PRK     3    AUD
  ANX     -     -
  WHS     -     -

Fewest-hop routes from LIB
  to PRK (3 hops): LIB ENG AUD PRK
  to DRM (2 hops): LIB CAF DRM
  to WHS: no route
```

The trace is **Table 13.2**, line for line, and the table of distances and parents is **Figure 13.6** in numbers. The route to PRK, LIB, ENG, AUD, PRK, has 3 hops, the fewest possible; Section 13.10 will find a route to PRK that is 10 meters shorter but uses 4 hops. The route to WHS does not exist, and the program says so rather than printing nonsense: `dist[WHS]` is still -1.

The function `show` copies the queue before printing it, because `std::queue` offers no way to look at any element except the front; printing destroys a queue, so it prints a copy. This is the honest price of using an adaptor that promises only queue behavior.

13.6.7 The same program in Java and Python

In Java, `ArrayDeque` serves as the queue, with `addLast` to enqueue and `pollFirst` to dequeue (Section 5.16.2), as **Listing 13.5** shows; in Python, `collections.deque` with `append` and `popleft` (Section 5.16.3). A Python `list` with `pop(0)` would also work, but each `pop(0)` shifts the whole list and costs $\Theta(n)$, which would turn the linear-time BFS of Equation (13.8) into a quadratic one.

Listing 13.5 `Bfs.java`: breadth-first search in Java, using `ArrayDeque` as the queue.

```java
// Bfs.java -- breadth-first search on the Pustaka Ceria campus map:
// visiting order, hop distances, parents, and fewest-hop routes.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Bfs {
    static final int V = 10;
    static final String[] CODE = {"LIB", "CAF", "ADM", "ENG", "DRM",
                                  "MED", "AUD", "PRK", "ANX", "WHS"};
    static final int[][] PATHS = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                                  {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                                  {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

    static List<List<Integer>> buildCampus() {
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        for (int[] p : PATHS) {
            adj.get(p[0]).add(p[1]);
            adj.get(p[1]).add(p[0]);
        }
        return adj;
    }

    static String show(ArrayDeque<Integer> q) {       // front first
        StringBuilder sb = new StringBuilder("[");
        for (int v : q) {
            if (sb.length() > 1) sb.append(' ');
            sb.append(CODE[v]);
        }
        return sb.append(']').toString();
    }

    static List<Integer> bfs(List<List<Integer>> adj, int s, int[] dist,
                             int[] parent, boolean trace) {
        Arrays.fill(dist, -1);
        Arrays.fill(parent, -1);
        List<Integer> order = new ArrayList<>();
        ArrayDeque<Integer> q = new ArrayDeque<>();
        dist[s] = 0;                          // mark s when it ENTERS the queue
        q.addLast(s);
        int step = 0;
        while (!q.isEmpty()) {
            int u = q.pollFirst();
            order.add(u);
```

Listing 13.5 (Continued) *Bfs.java: breadth-first search in Java, using ArrayDeque as the queue.*

```
            StringBuilder added = new StringBuilder();
            for (int v : adj.get(u)) {
                if (dist[v] == -1) {        // first time we see v
                    dist[v] = dist[u] + 1;
                    parent[v] = u;
                    q.addLast(v);
                    added.append(' ').append(CODE[v]);
                }
            }
            if (trace)
                System.out.printf("%4d  %s(d=%d)  enqueued:%-12s queue now %s%n",
                        ++step, CODE[u], dist[u],
                        added.length() == 0 ? " -" : added, show(q));
        }
        return order;
    }

    static void printPath(int[] parent, int[] dist, int t) {
        if (dist[t] == -1) {
            System.out.printf("  to %s: no route%n", CODE[t]);
            return;
        }
        ArrayDeque<Integer> path = new ArrayDeque<>();
        for (int v = t; v != -1; v = parent[v]) path.addFirst(v);
        System.out.printf("  to %s (%d hops):", CODE[t], dist[t]);
        for (int v : path) System.out.print(" " + CODE[v]);
        System.out.println();
    }

    public static void main(String[] args) {
        List<List<Integer>> adj = buildCampus();
        int[] dist = new int[V], parent = new int[V];
        System.out.println("BFS from LIB\nstep  dequeued");
        List<Integer> order = bfs(adj, 0, dist, parent, true);

        System.out.print("\nVisiting order:");
        for (int v : order) System.out.print(" " + CODE[v]);
        System.out.println("\n\nvertex  dist  parent");
        for (int v = 0; v < V; v++) {
            if (dist[v] == -1) System.out.printf("  %s     -     -%n", CODE[v]);
            else System.out.printf("  %s  %4d  %5s%n", CODE[v], dist[v],
                    parent[v] == -1 ? "-" : CODE[parent[v]]);
        }
        System.out.println("\nFewest-hop routes from LIB");
        printPath(parent, dist, 7);
        printPath(parent, dist, 4);
```

Listing 13.5 (Continued) *Bfs.java: breadth-first search in Java, using ArrayDeque as the queue.*

```
        printPath(parent, dist, 9);
    }
}
```

Listing 13.6 `bfs.py`: breadth-first search in Python, using `collections.deque` as the queue.

```
# bfs.py -- breadth-first search on the Pustaka Ceria campus map:
# visiting order, hop distances, parents, and fewest-hop routes.
from collections import deque

V = 10
CODE = ["LIB", "CAF", "ADM", "ENG", "DRM", "MED", "AUD", "PRK", "ANX", "WHS"]
PATHS = [(0, 1, 120), (0, 2, 80), (0, 3, 150), (1, 4, 200), (1, 5, 180),
         (2, 3, 60), (3, 6, 100), (4, 5, 90), (5, 6, 160), (6, 7, 70),
         (8, 9, 50)]


def build_campus():                          # weights are not needed by BFS
    adj = [[] for _ in range(V)]
    for u, v, _ in PATHS:
        adj[u].append(v)
        adj[v].append(u)
    return adj


def bfs(adj, s, trace=False):
    """Return (order, dist, parent); dist is -1 for vertices not reached."""
    dist = [-1] * V
    parent = [-1] * V
    order = []
    q = deque()
    dist[s] = 0                              # mark s when it ENTERS the queue
    q.append(s)
    step = 0
    while q:
        u = q.popleft()
        order.append(u)
        added = ""
        for v in adj[u]:
            if dist[v] == -1:                # first time we see v
                dist[v] = dist[u] + 1
                parent[v] = u
                q.append(v)
                added += " " + CODE[v]
        if trace:
            step += 1
            shown = "[" + " ".join(CODE[x] for x in q) + "]"
            print(f"{step:4}  {CODE[u]}(d={dist[u]})  "
                  f"enqueued:{added or ' -':<12} queue now {shown}")
    return order, dist, parent
```

Listing 13.6 (Continued) *bfs.py: breadth-first search in Python, using collections.deque as the queue.*

```
def print_path(parent, dist, t):
    if dist[t] == -1:
        print(f"  to {CODE[t]}: no route")
        return
    path = deque()                          # walk the parents back from t
    v = t
    while v != -1:
        path.appendleft(v)
        v = parent[v]
    print(f"  to {CODE[t]} ({dist[t]} hops): " + " ".join(CODE[x] for x in path))


def main():
    adj = build_campus()
    print("BFS from LIB\nstep  dequeued")
    order, dist, parent = bfs(adj, 0, trace=True)
    print("\nVisiting order: " + " ".join(CODE[v] for v in order))
    print("\nvertex  dist  parent")
    for v in range(V):
        if dist[v] == -1:
            print(f"  {CODE[v]}     -     -")
        else:
            p = "-" if parent[v] == -1 else CODE[parent[v]]
            print(f"  {CODE[v]}  {dist[v]:4}  {p:>5}")
    print("\nFewest-hop routes from LIB")
    for t in (7, 4, 9):                     # PRK, DRM, WHS
        print_path(parent, dist, t)


if __name__ == "__main__":
    main()
```

Both print exactly the output of **Listing 13.4**. Iterating over an `ArrayDeque` in Java, or over a `deque` in Python, visits the elements from front to back without removing them, so neither needs the copy that the C++ `show` had to make.

13.7 Depth-First Search

13.7.1 The idea: exploring a maze with a ball of thread

In the Greek legend, Theseus entered the Labyrinth of the Minotaur unwinding a ball of thread given to him by Ariadne, so that he could always find his way back. That thread is the idea of *depth-first search* (DFS). Standing in a room, you take any corridor you have not tried yet and follow it to the next room, and from there again an untried corridor, going deeper and deeper. When you arrive in a room whose corridors all lead to rooms you have already seen, you follow the thread back one room and try that room's next untried corridor. When you have backed up all the way to the entrance and it has no untried corridor left, you have explored everything reachable.

BFS explores cautiously, ring by ring; DFS explores boldly, as far as it can along one route before it backtracks. The thread is a *stack*: its end is the room you are in, going deeper pushes a room, and backing up pops one. The most natural way to program it is with recursion (Chapter 10), because the call stack keeps the thread for us: each recursive call is one step deeper, and each return is one step back. The modern form of DFS, together with the discovery and finish times described next, was set out by Robert Tarjan in 1972 [5], building on work with John Hopcroft [6]; the maze-exploring idea itself goes back to the nineteenth century.

13.7.2 Recursive DFS with discovery and finish times

DFS gives every vertex a color, which records where it stands in the search: *white* (not discovered yet), *gray* (discovered, but the search is still exploring below it, so it is on the thread) and *black* (finished: everything reachable from it has been explored). It also keeps a global clock, `time`, and stamps each vertex twice: its *discovery time* $v.d$ when it turns gray and its *finish time* $v.f$ when it turns black. (Note that d now means discovery time, not distance; the letters follow Cormen et al. [2].) **Pseudocode 13.2** gives the procedure. The outer procedure `DFS` starts a new search from every vertex that is still white, so that the whole graph is explored even when it is disconnected.

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
```

The recursion has no explicit base case line; it stops by itself. A call on a vertex whose neighbors are all non-white simply runs through its loop without recursing and returns. Because every call turns its vertex gray before looking at any neighbor, and only white vertices are visited, `DFS-VISIT` is called exactly once per vertex, even in a graph full of cycles.

The running time is found as for BFS. Lines 1 to 4 and the loop of lines 5 to 7 of `DFS` cost $\Theta(V)$ plus the calls. There is one call of `DFS-VISIT` per vertex, and the call on u spends $O(1 + \deg(u))$ in its own loop, not counting the recursive calls it makes. Summing, DFS costs $\Theta(V + E)$ with adjacency lists, the same as Equation (13.8). The extra memory is $\Theta(V)$ for the colors, parents and times, plus the recursion stack, which can be $|V|$ calls deep, a point Section 13.7.5 returns to.

13.7.3 Tracing DFS on the campus map

Run `DFS` on the campus map, trying start vertices in the order LIB, CAF, ..., WHS and neighbors in adjacency-list order.

- Time 1: LIB is discovered. Its first neighbor, CAF, is white: go deeper.
- Time 2: CAF is discovered (parent LIB). Its first neighbor, LIB, is gray (on the thread): skip. Next, DRM is white: go deeper.
- Time 3: DRM is discovered (parent CAF). CAF is gray: skip. MED is white: go deeper.
- Time 4: MED is discovered (parent DRM). CAF and DRM are gray: skip both. AUD is white: go deeper.
- Time 5: AUD is discovered (parent MED). Its first neighbor, ENG, is white: go deeper.

- Time 6: ENG is discovered (parent AUD). LIB is gray: skip. ADM is white: go deeper.
- Time 7: ADM is discovered (parent ENG). LIB and ENG are gray. ADM has nothing left: at time 8 it is finished.
- Back in ENG: AUD, its last neighbor, is gray. ENG finishes at time 9.
- Back in AUD: MED is gray; PRK is white. PRK is discovered at time 10; its only neighbor, AUD, is gray; PRK finishes at time 11. AUD has no neighbors left and finishes at time 12.
- The thread unwinds: MED finishes at 13, DRM at 14, CAF at 15 (its last neighbor MED is black), LIB at 16 (ADM and ENG are black).
- The outer loop of `DFS` finds the next white vertex, ANX: discovered at 17, WHS at 18, finished at 19 and 20.

The visiting order is LIB, CAF, DRM, MED, AUD, ENG, ADM, PRK, then ANX, WHS. Compare it with BFS's order LIB, CAF, ADM, ENG, DRM, MED, AUD, PRK: DFS ran straight around the long loop of the map, six buildings deep, before it came back for PRK, while BFS never went more than three steps from LIB. **Figure 13.7** draws the result. The label beside each vertex is d/f; the amber edges are the *tree edges*, the edges along which DFS discovered a new vertex, and they form the *DFS forest*, here two trees, one rooted at LIB and one at ANX. The dashed edges are the *back edges*: edges from a vertex to a gray vertex, an ancestor still on the thread.

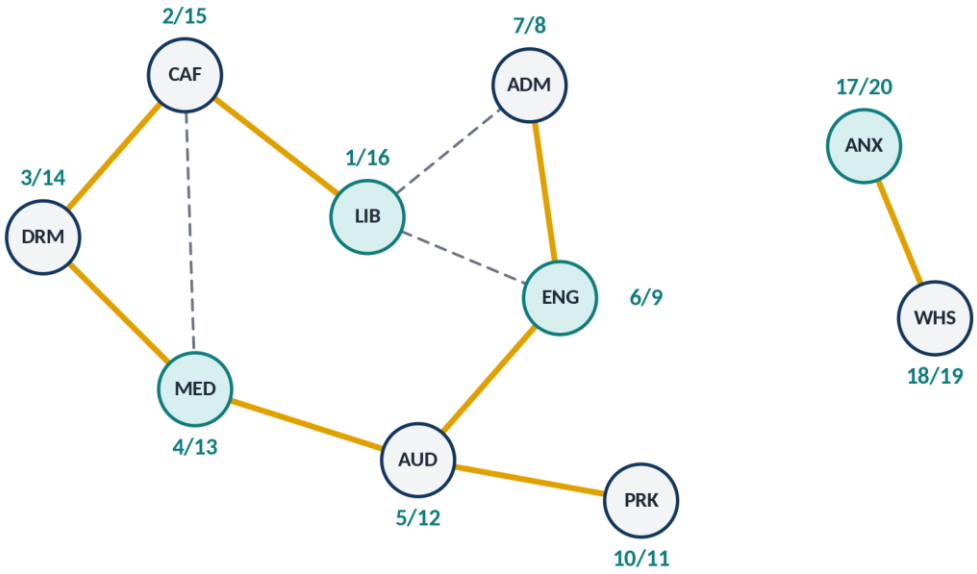


Figure 13.7 Depth-first search of the campus map. Each vertex is labeled with its discovery and finish times d/f; amber edges are tree edges and dashed edges are back edges.

Every non-tree edge in **Figure 13.7** is a back edge, and every back edge closes a cycle: CAF-MED closes CAF, DRM, MED, CAF; LIB-ENG closes LIB, CAF, DRM, MED, AUD, ENG, LIB; LIB-ADM closes LIB, CAF, DRM, MED, AUD, ENG, ADM, LIB. In an undirected graph this is always so. When DFS examines an edge $\{u, v\}$ and v is not white, v is either u's parent (the tree edge seen from the other end) or a gray ancestor of u; it can never be a black vertex in some other branch, because if v had been finished before u was discovered, the edge $\{v, u\}$ would have been examined from v while u was still white, and u would have become v's child. So in an undirected graph there are only tree edges and back edges, and the graph has a cycle exactly when DFS finds a back edge. Section 13.7.6 turns this into a cycle detector.

13.7.4 The parenthesis structure

Write each vertex's discovery as an opening parenthesis and its finish as a closing one, on a line of time. **Figure 13.8** does this for the campus map: each bar runs from d to f.

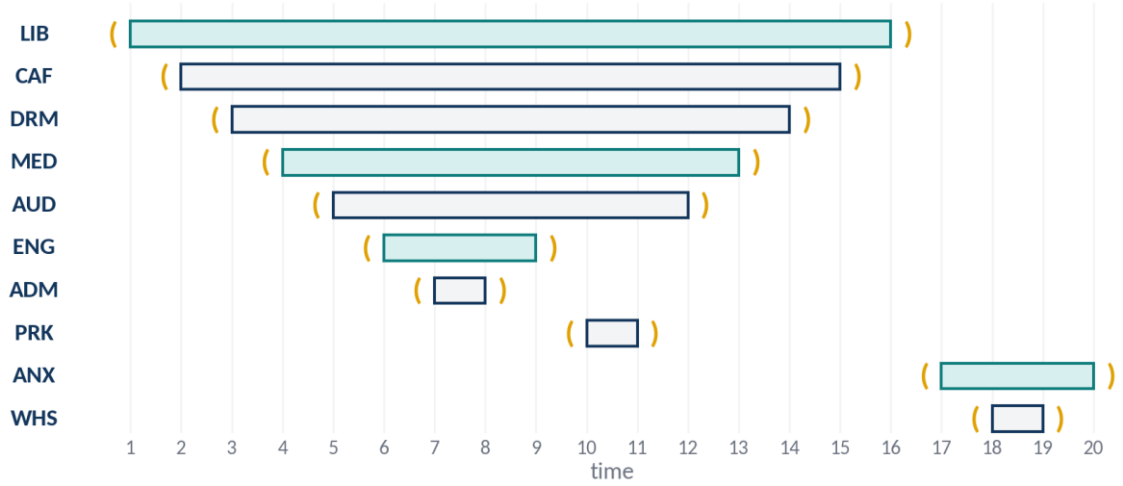


Figure 13.8 The discovery-to-finish interval of every vertex in the DFS of **Figure 13.7**: any two intervals are either nested or disjoint, like correctly matched parentheses.

Reading the bars from top to bottom gives "(LIB (CAF (DRM (MED (AUD (ENG (ADM ADM) ENG) (PRK PRK) AUD) MED) DRM) CAF) LIB) (ANX (WHS WHS) ANX)", a perfectly matched expression, like the brackets of Section 5.6. This is the *parenthesis theorem* [2]: for any two vertices u and v, exactly one of the following holds.

$$[u.d, u.f] \cap [v.d, v.f] = \emptyset \quad \text{or} \quad [u.d, u.f] \subset [v.d, v.f] \quad \text{or} \quad [v.d, v.f] \subset [u.d, u.f] \tag{13.9}$$

The reason is the stack discipline of the thread. If v is discovered while u is gray, then v is pushed on top of u, so it must be popped (finished) before u is: v's interval lies inside u's, and v is a descendant of u in the DFS forest. If v is discovered after u has finished, the intervals are disjoint. Nothing else can happen. In **Figure 13.8**, PRK's bar (10 to 11) lies inside AUD's (5 to 12) because PRK is a descendant of AUD, and it is disjoint from ENG's (6 to 9) because the two lie in different branches below AUD. Finish times will be the key to topological sorting in Section 13.9.3.

13.7.5 DFS without recursion

Recursion is elegant, but every call occupies a frame on the call stack, and the call stack is small: typically 8 MB on Linux and 1 MB for the main thread on Windows (Section 10.12 measured these limits in all three languages). A DFS along a path of a million vertices, which happens easily in a road network, makes a million nested calls and overflows the stack. The cure, as Section 10.13 showed for recursion in general, is to keep the thread in an explicit stack in ordinary memory.

The subtle part is to make the explicit stack remember what a recursive call remembers. When `DFS-VISIT(u)` calls itself on a neighbor and later returns, it resumes its loop at *the next neighbor* of u. So each stack entry must hold not just a vertex but also a position in its adjacency list: the pair (u, i) meaning "we are in vertex u and the next neighbor to try is `adj[u][i]`". **Pseudocode 13.3** is the result.

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)
```

Each pass of the loop does one of two things. If the top vertex still has an untried neighbor, it tries it and, if the neighbor is white, pushes it (going deeper). Otherwise it finishes the top vertex and pops it (backing up). Because the stack holds exactly the chain of gray vertices, with exactly the loop positions that the recursive calls would hold, the discovery and finish times are identical to those of **Pseudocode 13.2**, a claim the program below checks. The stack never holds more than $|V|$ entries, since every vertex is pushed once, when it turns gray.

A shorter iterative DFS is often seen: push the source; repeatedly pop a vertex, skip it if already visited, otherwise mark it and push all its unvisited neighbors (in reverse order, so that the first neighbor ends on top). It visits the vertices in a depth-first order, but the stack can hold up to $|E|$ entries, some of them duplicates, and it does not produce correct finish times, because a vertex is "finished" at no identifiable moment. For discovery order alone it is fine; for anything that needs finish times, such as the topological sort of Section 13.9.3, use **Pseudocode 13.3**. Exercise 13.6 compares the two.

13.7.6 Finding a cycle in an undirected graph

Section 13.7.3 showed that an undirected graph has a cycle exactly when DFS meets a back edge: an edge from the current vertex u to an already visited vertex v that is *not* u's own parent. (The parent is excluded because the tree edge $\{\pi[u], u\}$ appears again in `adj[u]`, seen from the other end, and walking along an edge and straight back is not a cycle.) When the back edge is found, the cycle itself can be read off the parents: start at u, follow π up to v, and close the loop with the edge $\{u, v\}$. On the campus map the first back edge DFS meets is MED-CAF, found at time 4 while MED is scanning its neighbors, and the parents give MED, DRM, CAF, then back to MED, the triangle at the left of the map.

The function `findCycle` in the next listing does exactly this. It passes the vertex it came from as `from`, rather than consulting the parent array, which is the common way to write it. (This test assumes a simple graph. In a multigraph, two parallel edges between u and v already form a cycle, and the test would have to exclude the particular edge it arrived on rather than the vertex.)

13.7.7 The programs

Listing 13.7 contains the recursive `dfsVisit`, which prints each discovery indented by its depth in the DFS forest; the iterative `dfsIterative` of **Pseudocode 13.3**, with `std::stack` holding pairs of a vertex and a neighbor index; a check that both produce the same discovery and finish times; and `findCycle`. The reference `std::size_t &i = st.top().second` is worth a look: incrementing `i` changes the index stored *inside* the top entry, which is how line 7 of **Pseudocode 13.3** updates the top of the stack in place.

Listing 13.7 `dfs.cpp`: recursive and iterative depth-first search with discovery and finish times, and cycle detection in an undirected graph.

```
// dfs.cpp -- depth-first search on the Pustaka Ceria campus map:
// recursive with discovery/finish times, iterative with an explicit stack,
// and cycle detection in an undirected graph.
#include <cstdio>
#include <stack>
#include <utility>
#include <vector>

const int V = 10;
const char *CODE[V] = {"LIB", "CAF", "ADM", "ENG", "DRM",
                       "MED", "AUD", "PRK", "ANX", "WHS"};
const int PATHS[][3] = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                        {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                        {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

std::vector<std::vector<int>> adj(V);
std::vector<int> d(V), f(V), parent(V);   // discovery, finish, DFS-tree parent
int timer = 0;

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
}

// The same search without recursion. Each stack entry is a vertex plus the
// index of the next neighbor to try, exactly what a recursive call remembers.
std::vector<int> dfsIterative(int s, std::vector<int> &dd, std::vector<int> &ff) {
    std::vector<int> order;
    int t = 0;
    dd.assign(V, 0);
    ff.assign(V, 0);
    std::stack<std::pair<int, std::size_t>> st;
    dd[s] = ++t;
    order.push_back(s);
    st.push({s, 0});
    while (!st.empty()) {
        int u = st.top().first;
        std::size_t &i = st.top().second;  // reference: advancing i updates the top
        if (i < adj[u].size()) {
```

Listing 13.7 (Continued) *dfs.cpp: recursive and iterative depth-first search with discovery and finish times, and cycle detection in an undirected graph.*

```
            int v = adj[u][i++];
            if (dd[v] == 0) {              // "call" dfsVisit(v)
                dd[v] = ++t;
                order.push_back(v);
                st.push({v, 0});
            }
        } else {                           // "return" from u
            ff[u] = ++t;
            st.pop();
        }
    }
    return order;
}

// Undirected cycle: a visited neighbor that is not the vertex we came from.
bool findCycle(int u, int from, std::vector<int> &par, std::vector<bool> &seen) {
    seen[u] = true;
    for (int v : adj[u]) {
        if (!seen[v]) {
            par[v] = u;
            if (findCycle(v, u, par, seen)) return true;
        } else if (v != from) {            // an edge back to an ancestor
            std::printf("Cycle found at edge %s-%s:", CODE[u], CODE[v]);
            for (int x = u; x != v; x = par[x]) std::printf(" %s", CODE[x]);
            std::printf(" %s %s\n", CODE[v], CODE[u]);
            return true;
        }
    }
    return false;
}

int main() {
    for (const auto &p : PATHS) {
        adj[p[0]].push_back(p[1]);
        adj[p[1]].push_back(p[0]);
    }
    std::printf("Recursive DFS over all vertices (indentation = depth)\n");
    parent.assign(V, -1);
    for (int s = 0; s < V; ++s)
        if (d[s] == 0) dfsVisit(s, 0);
    std::printf("\nvertex   d   f  parent\n");
    for (int v = 0; v < V; ++v)
        std::printf("  %s  %3d %3d  %s\n", CODE[v], d[v], f[v],
                    parent[v] == -1 ? "-" : CODE[parent[v]]);
```

Listing 13.7 (Continued) *dfs.cpp: recursive and iterative depth-first search with discovery and finish times, and cycle detection in an undirected graph.*

```
    std::vector<int> dd, ff;
    std::vector<int> order = dfsIterative(0, dd, ff);
    std::printf("\nIterative DFS from LIB:");
    for (int v : order) std::printf(" %s", CODE[v]);
    bool same = true;
    for (int v : order) same = same && dd[v] == d[v] && ff[v] == f[v];
    std::printf("\nSame d and f as the recursive search: %s\n",
                same ? "yes" : "no");

    std::vector<int> par(V, -1);
    std::vector<bool> seen(V, false);
    if (!findCycle(0, -1, par, seen)) std::printf("No cycle reachable from LIB\n");

    std::vector<bool> seen2(V, false);   // the ANX-WHS piece is a single edge
    if (!findCycle(8, -1, par, seen2)) std::printf("No cycle reachable from ANX\n");
    return 0;
}
```

```
Recursive DFS over all vertices (indentation = depth)
LIB d=1
  CAF d=2
    DRM d=3
      MED d=4
        AUD d=5
          ENG d=6
            ADM d=7
          PRK d=10
ANX d=17
  WHS d=18

vertex   d   f  parent
  LIB    1  16  -
  CAF    2  15  LIB
  ADM    7   8  ENG
  ENG    6   9  AUD
  DRM    3  14  CAF
  MED    4  13  DRM
  AUD    5  12  MED
  PRK   10  11  AUD
  ANX   17  20  -
  WHS   18  19  ANX

Iterative DFS from LIB: LIB CAF DRM MED AUD ENG ADM PRK
Same d and f as the recursive search: yes
Cycle found at edge MED-CAF: MED DRM CAF MED
No cycle reachable from ANX
```

The indented listing is the DFS forest of **Figure 13.7** lying on its side: each vertex appears under the vertex that discovered it. The indentation grows to six levels on the way to ADM, while BFS never went beyond distance 3. The table of d, f and parents agrees with **Figure 13.7** and **Figure 13.8**, and the iterative search reproduces every time stamp. The cycle detector reports the triangle MED, DRM, CAF, and correctly finds no cycle in the ANX-WHS component, which is a single edge.

Listing 13.8 and **Listing 13.9** are the same program in Java and Python. Java has no pair of a vertex and a mutable index, so each stack entry is a two-element `int` array whose second element is incremented in place; `ArrayDeque` with `push`, `peek` and `pop` is the stack (Section 5.16.2). In Python a two-element list plays the same role, and the stack is an ordinary `list` with `append` and `pop`.

Listing 13.8 `Dfs.java`: depth-first search, recursive and iterative, in Java.

```
// Dfs.java -- depth-first search on the Pustaka Ceria campus map:
// recursive with discovery/finish times, iterative with an explicit stack,
// and cycle detection in an undirected graph.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Dfs {
    static final int V = 10;
    static final String[] CODE = {"LIB", "CAF", "ADM", "ENG", "DRM",
                                  "MED", "AUD", "PRK", "ANX", "WHS"};
    static final int[][] PATHS = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                                  {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                                  {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

    static List<List<Integer>> adj = new ArrayList<>();
    static int[] d = new int[V], f = new int[V], parent = new int[V];
    static int timer = 0;

    static void dfsVisit(int u, int depth) {
        d[u] = ++timer;                                  // discovered
        System.out.printf("%s%s d=%d%n", "  ".repeat(depth), CODE[u], d[u]);
        for (int v : adj.get(u)) {
            if (d[v] == 0) {                             // go deeper
                parent[v] = u;
                dfsVisit(v, depth + 1);
            }
        }
        f[u] = ++timer;                                  // finished
    }

    // Each stack entry is {vertex, index of the next neighbor to try}.
    static List<Integer> dfsIterative(int s, int[] dd, int[] ff) {
        List<Integer> order = new ArrayList<>();
        int t = 0;
        Arrays.fill(dd, 0);
        Arrays.fill(ff, 0);
        ArrayDeque<int[]> st = new ArrayDeque<>();
        dd[s] = ++t;
        order.add(s);
        st.push(new int[] {s, 0});
        while (!st.isEmpty()) {
            int[] top = st.peek();
            int u = top[0];
            if (top[1] < adj.get(u).size()) {
```

Listing 13.8 (Continued) *Dfs.java: depth-first search, recursive and iterative, in Java.*

```
                int v = adj.get(u).get(top[1]++);
                if (dd[v] == 0) {                       // "call" dfsVisit(v)
                    dd[v] = ++t;
                    order.add(v);
                    st.push(new int[] {v, 0});
                }
            } else {                                    // "return" from u
                ff[u] = ++t;
                st.pop();
            }
        }
        return order;
    }

    static boolean findCycle(int u, int from, int[] par, boolean[] seen) {
        seen[u] = true;
        for (int v : adj.get(u)) {
            if (!seen[v]) {
                par[v] = u;
                if (findCycle(v, u, par, seen)) return true;
            } else if (v != from) {                     // an edge back to an ancestor
                System.out.printf("Cycle found at edge %s-%s:", CODE[u], CODE[v]);
                for (int x = u; x != v; x = par[x]) System.out.print(" " + CODE[x]);
                System.out.printf(" %s %s%n", CODE[v], CODE[u]);
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        for (int[] p : PATHS) {
            adj.get(p[0]).add(p[1]);
            adj.get(p[1]).add(p[0]);
        }
        System.out.println("Recursive DFS over all vertices (indentation = depth)");
        Arrays.fill(parent, -1);
        for (int s = 0; s < V; s++)
            if (d[s] == 0) dfsVisit(s, 0);
        System.out.println("\nvertex   d   f  parent");
        for (int v = 0; v < V; v++)
            System.out.printf("  %s  %3d %3d  %s%n", CODE[v], d[v], f[v],
                    parent[v] == -1 ? "-" : CODE[parent[v]]);
```

Listing 13.8 (Continued) *Dfs.java: depth-first search, recursive and iterative, in Java.*

```
        int[] dd = new int[V], ff = new int[V];
        List<Integer> order = dfsIterative(0, dd, ff);
        System.out.print("\nIterative DFS from LIB:");
        for (int v : order) System.out.print(" " + CODE[v]);
        boolean same = true;
        for (int v : order) same = same && dd[v] == d[v] && ff[v] == f[v];
        System.out.printf("%nSame d and f as the recursive search: %s%n",
                same ? "yes" : "no");

        int[] par = new int[V];
        Arrays.fill(par, -1);
        if (!findCycle(0, -1, par, new boolean[V]))
            System.out.println("No cycle reachable from LIB");
        if (!findCycle(8, -1, par, new boolean[V]))
            System.out.println("No cycle reachable from ANX");
    }
}
```

Listing 13.9 `dfs.py`: depth-first search, recursive and iterative, in Python.

```
# dfs.py -- depth-first search on the Pustaka Ceria campus map:
# recursive with discovery/finish times, iterative with an explicit stack,
# and cycle detection in an undirected graph.

V = 10
CODE = ["LIB", "CAF", "ADM", "ENG", "DRM", "MED", "AUD", "PRK", "ANX", "WHS"]
PATHS = [(0, 1, 120), (0, 2, 80), (0, 3, 150), (1, 4, 200), (1, 5, 180),
         (2, 3, 60), (3, 6, 100), (4, 5, 90), (5, 6, 160), (6, 7, 70),
         (8, 9, 50)]

adj = [[] for _ in range(V)]
d = [0] * V                                     # discovery time, 0 = unvisited
f = [0] * V                                     # finish time
parent = [-1] * V
timer = 0


def dfs_visit(u, depth):
    global timer
    timer += 1
    d[u] = timer                                # discovered
    print(f"{'  ' * depth}{CODE[u]} d={d[u]}")
    for v in adj[u]:
        if d[v] == 0:                           # unvisited neighbor: go deeper
            parent[v] = u
            dfs_visit(v, depth + 1)
    timer += 1
    f[u] = timer                                # finished


def dfs_iterative(s):
    """Each stack entry is [vertex, index of the next neighbor to try]."""
    dd, ff, order, t = [0] * V, [0] * V, [], 1
    dd[s] = t
    order.append(s)
    stack = [[s, 0]]
    while stack:
        top = stack[-1]
        u = top[0]
        if top[1] < len(adj[u]):
            v = adj[u][top[1]]
            top[1] += 1
            if dd[v] == 0:                      # "call" dfs_visit(v)
                t += 1
                dd[v] = t
                order.append(v)
```

Listing 13.9 (Continued) *dfs.py: depth-first search, recursive and iterative, in Python.*

```
                stack.append([v, 0])
        else:                                   # "return" from u
            t += 1
            ff[u] = t
            stack.pop()
    return order, dd, ff


def find_cycle(u, came_from, par, seen):
    seen[u] = True
    for v in adj[u]:
        if not seen[v]:
            par[v] = u
            if find_cycle(v, u, par, seen):
                return True
        elif v != came_from:                    # an edge back to an ancestor
            walk, x = [], u
            while x != v:
                walk.append(CODE[x])
                x = par[x]
            print(f"Cycle found at edge {CODE[u]}-{CODE[v]}: "
                  + " ".join(walk + [CODE[v], CODE[u]]))
            return True
    return False


def main():
    for u, v, _ in PATHS:
        adj[u].append(v)
        adj[v].append(u)
    print("Recursive DFS over all vertices (indentation = depth)")
    for s in range(V):
        if d[s] == 0:
            dfs_visit(s, 0)
    print("\nvertex   d   f  parent")
    for v in range(V):
        p = "-" if parent[v] == -1 else CODE[parent[v]]
        print(f"  {CODE[v]}  {d[v]:3} {f[v]:3}  {p}")

    order, dd, ff = dfs_iterative(0)
    print("\nIterative DFS from LIB: " + " ".join(CODE[v] for v in order))
    same = all(dd[v] == d[v] and ff[v] == f[v] for v in order)
    print(f"Same d and f as the recursive search: {'yes' if same else 'no'}")
```

Listing 13.9 (Continued) *dfs.py: depth-first search, recursive and iterative, in Python.*

```
    par = [-1] * V
    if not find_cycle(0, -1, par, [False] * V):
        print("No cycle reachable from LIB")
    if not find_cycle(8, -1, par, [False] * V):   # ANX-WHS is a single edge
        print("No cycle reachable from ANX")


if __name__ == "__main__":
    main()
```

Both print exactly the output of **Listing 13.7**. A warning for Python: CPython limits recursion depth to 1000 calls by default, so the recursive `dfs_visit` fails with `RecursionError` on any graph with a path of more than about a thousand vertices. Raising the limit with `sys.setrecursionlimit` only moves the problem toward a genuine crash of the interpreter; for large graphs in Python, the iterative version is the only safe one.

13.8 Connectivity and Connected Components

13.8.1 One search per component

Both BFS and DFS started from a vertex s visit exactly the vertices reachable from s, that is, exactly the connected component of s in an undirected graph. So the components can be found by the simplest possible loop: go through the vertices in order; whenever a vertex has not been labeled yet, it starts a new component, and one search from it labels the whole component. **Pseudocode 13.4** writes this with BFS; DFS would do equally well, and the outer loop of `DFS` in **Pseudocode 13.2** is in fact already this algorithm.

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
```

It is tempting to think that the nested loops make this quadratic, but they do not. Every vertex is labeled once and enqueued once in the whole run, not once per component, and every adjacency list is scanned once, when its vertex is dequeued. The searches of the different components touch disjoint parts of the graph, so their costs add up to one pass over everything: $\Theta(V + E)$, the same as a single BFS. Hopcroft and Tarjan made this observation the basis of a whole family of linear-time graph algorithms [6].

Once every vertex carries its component number, several questions become $O(1)$ lookups. Are u and v connected? Exactly when `comp[u] = comp[v]`. Is the whole graph connected? Exactly when `count` is 1. How large is the component of v? Count the vertices with the same label, or better, record the sizes during the searches.

13.8.2 The shuttle road

On the campus map, **Pseudocode 13.4** starts at LIB, whose BFS labels the eight campus buildings with component 0; the loop then skips CAF, ADM and the others, which are already labeled, reaches ANX, and labels ANX and WHS with component 1. The map has two components, as **Figure 13.9**(a) shows in two colors. The library management now opens a shuttle road of 400 meters from the parking area to the city annex. One new edge, PRK-ANX, is added to the adjacency lists and the algorithm is run again: its BFS from LIB now crosses the new road, and all ten buildings fall into component 0 (**Figure 13.9**(b)).

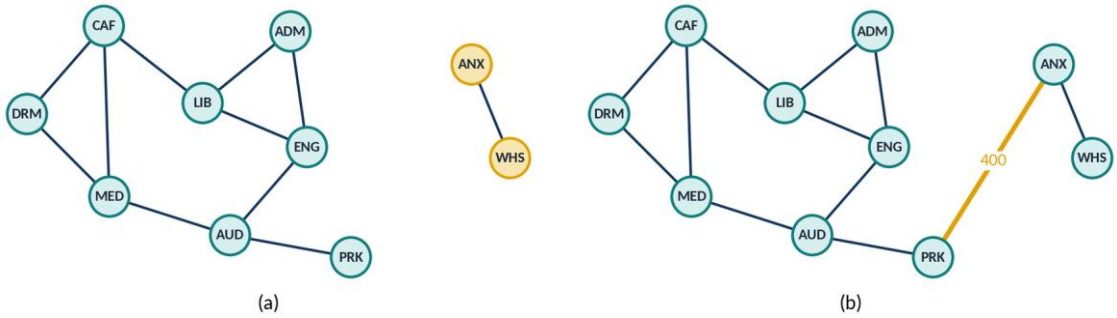


Figure 13.9 Connected components of the campus map: (a) as built, two components (teal and amber); (b) after the 400-meter shuttle road PRK-ANX is added, one component.

A single edge can change the number of components by at most one: adding an edge between two different components merges them, and adding an edge inside a component changes nothing. Removing an edge is the mirror image. An edge whose removal would disconnect its component, such as AUD-PRK or the new PRK-ANX, is called a *bridge*, in memory of Königsberg; LIB-CAF is not a bridge, because the cycles around it provide other routes. Finding all bridges efficiently is a classic application of DFS finish and discovery times [5], and a reliable network, of roads or of computers, is one with no bridges.

13.8.3 The programs

Listing 13.10 implements **Pseudocode 13.4** and a report that prints the components, whether the map is connected, and whether LIB and WHS lie in the same component; it then adds the shuttle road and reports again.

Listing 13.10 `components.cpp`: connected components of the campus map, before and after the shuttle road.

```cpp
// components.cpp -- connected components of the Pustaka Ceria campus map,
// before and after a shuttle road joins the city annex to the campus.
#include <cstdio>
#include <queue>
#include <vector>

const int V = 10;
const char *CODE[V] = {"LIB", "CAF", "ADM", "ENG", "DRM",
                       "MED", "AUD", "PRK", "ANX", "WHS"};
const int PATHS[][3] = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                        {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                        {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

// comp[v] = number of v's component (0, 1, 2, ...); returns how many there are.
int components(const std::vector<std::vector<int>> &adj, std::vector<int> &comp) {
    comp.assign(adj.size(), -1);
    int count = 0;
    for (int s = 0; s < static_cast<int>(adj.size()); ++s) {
        if (comp[s] != -1) continue;       // already inside an earlier component
        std::queue<int> q;                 // one BFS sweeps out one component
        comp[s] = count;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : adj[u])
                if (comp[v] == -1) {
                    comp[v] = count;
                    q.push(v);
                }
        }
        ++count;
    }
    return count;
}
```

Listing 13.10 (Continued) *components.cpp: connected components of the campus map, before and after the shuttle road.*

```
void report(const char *title, const std::vector<std::vector<int>> &adj) {
    std::vector<int> comp;
    int k = components(adj, comp);
    std::printf("%s: %d component(s)\n", title, k);
    for (int c = 0; c < k; ++c) {
        std::printf("  component %d:", c);
        for (int v = 0; v < V; ++v)
            if (comp[v] == c) std::printf(" %s", CODE[v]);
        std::printf("\n");
    }
    std::printf("  connected? %s\n", k == 1 ? "yes" : "no");
    std::printf("  same component LIB and WHS? %s\n",
                comp[0] == comp[9] ? "yes" : "no");
}

int main() {
    std::vector<std::vector<int>> adj(V);
    for (const auto &p : PATHS) {
        adj[p[0]].push_back(p[1]);
        adj[p[1]].push_back(p[0]);
    }
    report("Campus map as built", adj);

    adj[7].push_back(8);                    // new shuttle road PRK-ANX, 400 m
    adj[8].push_back(7);
    std::printf("\nAdding the shuttle road PRK-ANX\n\n");
    report("After the shuttle road", adj);
    return 0;
}
```

```
Campus map as built: 2 component(s)
  component 0: LIB CAF ADM ENG DRM MED AUD PRK
  component 1: ANX WHS
  connected? no
  same component LIB and WHS? no

Adding the shuttle road PRK-ANX

After the shuttle road: 1 component(s)
  component 0: LIB CAF ADM ENG DRM MED AUD PRK ANX WHS
  connected? yes
  same component LIB and WHS? yes
```

The same program in Java (**Listing 13.11**) and Python (**Listing 13.12**) follows. As before, only the queue class changes.

Listing 13.11 `Components.java`: connected components in Java.

```java
// Components.java -- connected components of the Pustaka Ceria campus map,
// before and after a shuttle road joins the city annex to the campus.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Components {
    static final int V = 10;
    static final String[] CODE = {"LIB", "CAF", "ADM", "ENG", "DRM",
                                  "MED", "AUD", "PRK", "ANX", "WHS"};
    static final int[][] PATHS = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                                  {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                                  {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};

    // comp[v] = number of v's component; returns how many components there are.
    static int components(List<List<Integer>> adj, int[] comp) {
        Arrays.fill(comp, -1);
        int count = 0;
        for (int s = 0; s < adj.size(); s++) {
            if (comp[s] != -1) continue;             // inside an earlier component
            ArrayDeque<Integer> q = new ArrayDeque<>();   // one BFS, one component
            comp[s] = count;
            q.addLast(s);
            while (!q.isEmpty()) {
                int u = q.pollFirst();
                for (int v : adj.get(u))
                    if (comp[v] == -1) {
                        comp[v] = count;
                        q.addLast(v);
                    }
            }
            count++;
        }
        return count;
    }
```

Listing 13.11 (Continued) *Components.java: connected components in Java.*

```
    static void report(String title, List<List<Integer>> adj) {
        int[] comp = new int[V];
        int k = components(adj, comp);
        System.out.printf("%s: %d component(s)%n", title, k);
        for (int c = 0; c < k; c++) {
            System.out.printf("  component %d:", c);
            for (int v = 0; v < V; v++)
                if (comp[v] == c) System.out.print(" " + CODE[v]);
            System.out.println();
        }
        System.out.printf("  connected? %s%n", k == 1 ? "yes" : "no");
        System.out.printf("  same component LIB and WHS? %s%n",
                comp[0] == comp[9] ? "yes" : "no");
    }

    public static void main(String[] args) {
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        for (int[] p : PATHS) {
            adj.get(p[0]).add(p[1]);
            adj.get(p[1]).add(p[0]);
        }
        report("Campus map as built", adj);

        adj.get(7).add(8);                          // shuttle road PRK-ANX, 400 m
        adj.get(8).add(7);
        System.out.println("\nAdding the shuttle road PRK-ANX\n");
        report("After the shuttle road", adj);
    }
}
```

Listing 13.12 `components.py`: connected components in Python.

```
# components.py -- connected components of the Pustaka Ceria campus map,
# before and after a shuttle road joins the city annex to the campus.
from collections import deque

V = 10
CODE = ["LIB", "CAF", "ADM", "ENG", "DRM", "MED", "AUD", "PRK", "ANX", "WHS"]
PATHS = [(0, 1, 120), (0, 2, 80), (0, 3, 150), (1, 4, 200), (1, 5, 180),
         (2, 3, 60), (3, 6, 100), (4, 5, 90), (5, 6, 160), (6, 7, 70),
         (8, 9, 50)]


def components(adj):
    """Return (count, comp) where comp[v] is the number of v's component."""
    comp = [-1] * len(adj)
    count = 0
    for s in range(len(adj)):
        if comp[s] != -1:                    # already inside an earlier component
            continue
        q = deque([s])                       # one BFS sweeps out one component
        comp[s] = count
        while q:
            u = q.popleft()
            for v in adj[u]:
                if comp[v] == -1:
                    comp[v] = count
                    q.append(v)
        count += 1
    return count, comp


def report(title, adj):
    k, comp = components(adj)
    print(f"{title}: {k} component(s)")
    for c in range(k):
        print(f"  component {c}: "
              + " ".join(CODE[v] for v in range(V) if comp[v] == c))
    print(f"  connected? {'yes' if k == 1 else 'no'}")
    print(f"  same component LIB and WHS? {'yes' if comp[0] == comp[9] else 'no'}")
```

Listing 13.12 (Continued) *components.py: connected components in Python.*

```
def main():
    adj = [[] for _ in range(V)]
    for u, v, _ in PATHS:
        adj[u].append(v)
        adj[v].append(u)
    report("Campus map as built", adj)

    adj[7].append(8)                         # new shuttle road PRK-ANX, 400 m
    adj[8].append(7)
    print("\nAdding the shuttle road PRK-ANX\n")
    report("After the shuttle road", adj)


if __name__ == "__main__":
    main()
```

Both print exactly the output of **Listing 13.10**. The programs re-run the whole labeling after adding one edge, which costs $\Theta(V + E)$ each time. When edges are added one at a time and the question "are u and v connected?" is asked in between, as in a network that grows, the *disjoint-set* or *union-find* structure answers each operation in almost constant amortized time [2]; it belongs to a later course, but it is worth knowing that it exists.

13.8.4 Directed graphs: a note on strong connectivity

In a directed graph, "reachable" is no longer symmetric: a one-way street can take you from u to v with no way back. A directed graph is *strongly connected* if every vertex can reach every other *following the arrows*, and its *strongly connected components* are the maximal sets of vertices that can all reach one another. A search from s in a directed graph finds everything reachable *from* s, which is generally more than s's strongly connected component. The standard algorithms for strongly connected components, by Tarjan [5] and by Kosaraju and Sharir [2], are built on two passes of DFS and its finish times, and run in $\Theta(V + E)$; they are beyond the scope of this book. The web crawler of Section 13.13 meets the simpler directed question, "which pages can be reached from the home page?", and BFS answers it directly.

13.9 Directed Acyclic Graphs and Topological Sort

13.9.1 The new-book workflow

Before a newly delivered book reaches a shelf at Pustaka Ceria, nine tasks must be done: the book is *ordered* from the publisher, *received* at the warehouse, *inspected* for damage, *cataloged* (given its classification number and catalog record), *stamped* with the library's stamp, *labeled* with its spine label, *covered* in protective plastic, *registered* in the online catalog, and finally *shelved*. The tasks cannot be done in just any order. A book cannot be inspected before it is received, and it cannot be labeled before it is cataloged, because the label shows the classification number. The dependencies form the directed graph of **Figure 13.10**(a), in which an arrow (u, v) means "task u must be finished before task v starts".

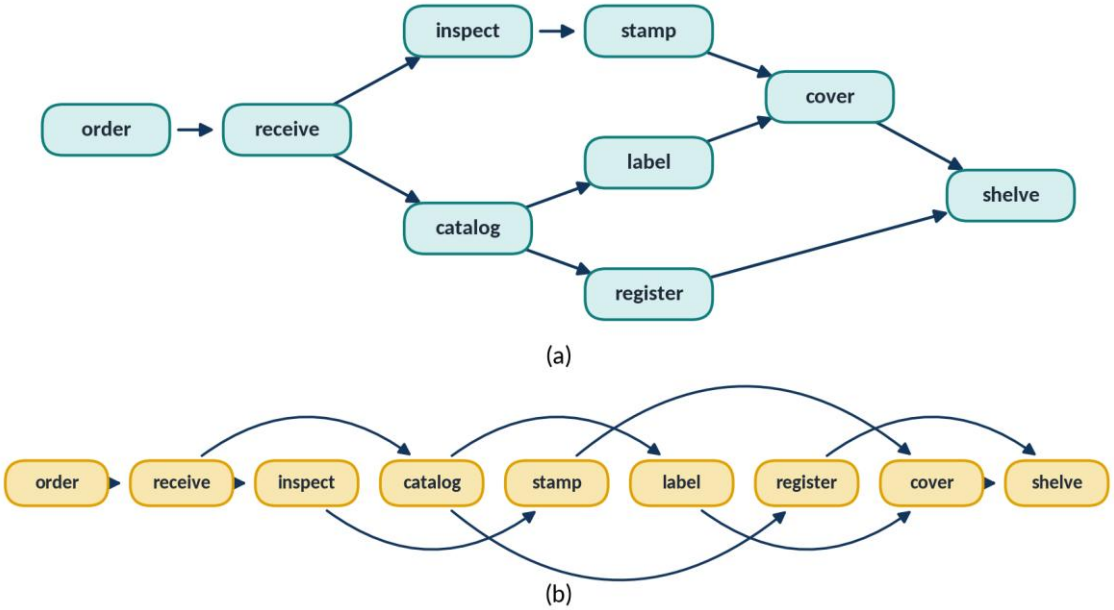


Figure 13.10 The new-book workflow: (a) the dependency DAG, with an arrow from each task to a task that depends on it; (b) the same DAG in the topological order found by Kahn's algorithm, where every arrow points to the right.

The graph is a DAG: following the arrows you can never return to a task you have left. A student helper who processes one book at a time needs an order of the nine tasks in which every arrow points *forward*, so that each task comes after all its prerequisites.

> **DEFINITION.** A *topological order* (or *topological sort*) of a directed graph is a listing of all its vertices such that for every arrow (u, v), u appears before v.

Figure 13.10(b) shows one topological order: laid out in a line, all ten arrows point to the right. A topological order exists if and only if the graph is a DAG. One direction is easy: if there is a cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1$, then v_1 must come before v_2, which must come before v_3, and so on, until v_k must come before v_1, so v_1 must come before itself, which no listing can achieve. The other direction, that every DAG has a topological order, is proved by the two algorithms that follow, each of which constructs one. Topological orders are usually not unique: in the workflow, *inspect* and *catalog* can be done in either order once the book is received.

13.9.2 Kahn's algorithm

The first algorithm, published by Arthur B. Kahn in 1962 for scheduling the tasks of large engineering projects [7], does what a sensible person with a to-do list does: repeatedly pick a task that has no unfinished prerequisite, do it, and cross it off. "No unfinished prerequisite" is "in-degree 0 among the remaining tasks". The algorithm keeps the current in-degree of every vertex and a queue of the vertices whose in-degree has dropped to 0.

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
```

Why does it work? A vertex enters L only when its in-degree has dropped to 0, which means that every one of its prerequisites was appended to L before it: every arrow points forward in L. And why does it not get stuck on a DAG? As long as some vertices remain, the remaining part of a DAG is itself a DAG, and every DAG with at least one vertex has a vertex of in-degree 0 (otherwise, start anywhere and keep walking backward along incoming arrows; with finitely many vertices, the walk must eventually repeat a vertex, which would be a cycle). So the queue never runs dry while vertices remain, and all $|V|$ vertices are output. Conversely, on a graph with a cycle, no vertex of the cycle can ever reach in-degree 0, because each has a prerequisite on the cycle that is never output; the algorithm stops early, and line 15 detects it.

The running time is $\Theta(V + E)$: lines 1 to 5 look at every vertex and arrow once, every vertex is enqueued and dequeued once, and lines 11 to 14 look at every arrow once more.

Table 13.3 traces Kahn's algorithm on the workflow. The initial in-degrees are 0 for *order*, 2 for *cover* and *shelve* (each waits for two tasks), and 1 for every other task.

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 → 0	receive	receive
2	receive	inspect 1 → 0, catalog 1 → 0	inspect, catalog	inspect catalog
3	inspect	stamp 1 → 0	stamp	catalog stamp
4	catalog	label 1 → 0, register 1 → 0	label, register	stamp label register
5	stamp	cover 2 → 1	none	label register
6	label	cover 1 → 0	cover	register cover
7	register	shelve 2 → 1	none	cover
8	cover	shelve 1 → 0	shelve	shelve
9	shelve	none	none	(empty)

The output is order, receive, inspect, catalog, stamp, label, register, cover, shelve, which is the order laid out in **Figure 13.10**(b). Steps 5 and 7 show the in-degree doing its job: *stamp* is done, but *cover* still waits for *label*, and only when *label* is done at step 6 does *cover* become available.

13.9.3 Topological sort by DFS

The second algorithm uses the finish times of DFS, and it is remarkably short: run DFS on the whole graph and list the vertices in *decreasing order of finish time*. Equivalently, whenever a vertex finishes, put it at the front of the list [2], [5].

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
```

Why is it correct? Take any arrow (u, v) and look at the moment DFS examines it, from u, while u is gray. There are three possibilities for v. If v is white, it becomes a descendant of u and, by the parenthesis structure Equation (13.9), finishes before u does. If v is black, it has already finished, so again $v.f < u.f$. And v cannot be gray, because a gray v is an ancestor of u on the thread, and the arrow (u, v) together with the tree path from v down to u would form a directed cycle, which a DAG does not have. So in every case $v.f < u.f$: every arrow goes from a later-finishing vertex to an earlier-finishing one, and listing by decreasing finish time makes every arrow point forward.

On the workflow, DFS starting at *order* runs order, receive, inspect, stamp, cover, shelve before anything finishes. *shelve* finishes first, then *cover*, *stamp* and *inspect*; back in *receive*, the search goes to *catalog* and *label* (whose neighbor *cover* is already black), then *register* (whose neighbor *shelve* is black). Finishing order: shelve, cover, stamp, inspect, label, register, catalog, receive, order. Reversed: order, receive, catalog, register, label, inspect, stamp, cover, shelve. It is a different topological order from Kahn's, and an equally valid one: check that every arrow of **Figure 13.10**(a) points forward in it.

13.9.4 Detecting a cycle in a directed graph

The proof above contains a cycle detector. In a directed graph, an arrow from the current vertex u to a *gray* vertex v is a back edge, and a back edge means a directed cycle: v reaches u along tree edges, and u reaches v along the back edge. Conversely, if the graph has a cycle, DFS finds a back

edge: let v be the first vertex of the cycle to be discovered; all the others are white at that moment and are reachable from v, so they become descendants of v, and the arrow from the last of them into v is examined while v is still gray. So *a directed graph is acyclic exactly when DFS finds no arrow into a gray vertex*. Note that an arrow into a *black* vertex is harmless in a directed graph: in the workflow, *label* to *cover* arrives at a black *cover* in the DFS above, and there is no cycle.

This is why DFS needs three colors in a directed graph rather than just "visited or not": "visited" lumps gray and black together, and the difference between them is the difference between a cycle and a harmless second route to the same task.

Suppose a librarian, meaning to say that shelving is followed by a final catalog check, enters a mistaken arrow from *shelve* to *catalog*. Now catalog, label, cover, shelve, catalog is a cycle, and no order can satisfy the dependencies. Both algorithms notice. Kahn's algorithm outputs order, receive, inspect and stamp and then stops with an empty queue: *catalog* still waits for *shelve*, which waits for *cover*, which waits for *label*, which waits for *catalog*. Only 4 of the 9 tasks are output, and line 15 of **Pseudocode 13.5** reports the cycle. DFS goes order, receive, inspect, stamp, cover, shelve, and then along the mistaken arrow to catalog and on to label, whose arrow leads to *cover*, which is gray. **Figure 13.11** shows the moment: every task on the current path is gray, only *register* is still white, and the red arrows are the cycle, closed by the dashed mistaken arrow.

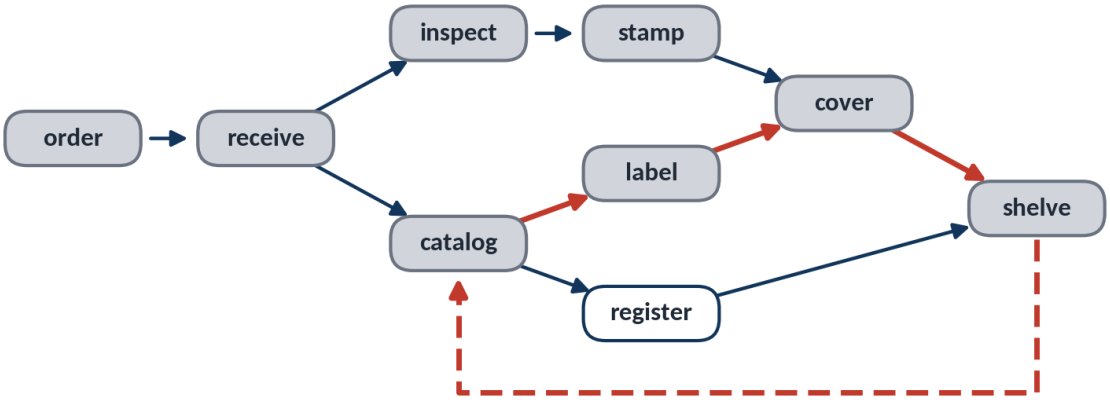


Figure 13.11 The workflow with a mistaken arrow from shelve to catalog (dashed). When DFS examines the arrow from label to cover, every task on the current path is gray and cover is one of them: a back edge, which proves the red cycle.

The back edge DFS reports, label to cover, is not the mistaken arrow itself but another arrow of the same cycle; which arrow of a cycle is caught depends on where the search enters the cycle. The practical lesson: a cycle detector proves that a cycle exists and can print it (by following parents from u up to v, as in Section 13.7.6), but deciding which dependency is the wrong one is up to the librarian.

13.9.5 The programs

Listing 13.13 runs Kahn's algorithm with a trace, then the DFS version with three colors, then adds the mistaken arrow and runs both again.

Listing 13.13 `topo.cpp`: topological sort by Kahn's algorithm and by depth-first search, with cycle detection.

```
// topo.cpp -- topological sort of the Pustaka Ceria new-book workflow,
// by Kahn's algorithm and by depth-first search, with cycle detection.
#include <algorithm>
#include <cstdio>
#include <queue>
#include <vector>

const int V = 9;
const char *TASK[V] = {"order", "receive", "inspect", "catalog", "stamp",
                       "label", "cover", "register", "shelve"};
// Arc u -> v means "task u must be finished before task v starts".
const int ARCS[][2] = {{0, 1}, {1, 2}, {1, 3}, {2, 4}, {3, 5},
                       {3, 7}, {4, 6}, {5, 6}, {6, 8}, {7, 8}};

using Graph = std::vector<std::vector<int>>;

// Kahn: repeatedly output a task with no unfinished prerequisite.
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]);
            }
        }
        if (trace) std::printf("\n");
    }
    return order;                           // shorter than V <=> there is a cycle
}

// DFS with three colors: WHITE unvisited, GRAY on the current path, BLACK done.
enum Color { WHITE, GRAY, BLACK };
```

Listing 13.13 (Continued) *topo.cpp: topological sort by Kahn's algorithm and by depth-first search, with cycle detection.*

```
bool dfsTopo(const Graph &g, int u, std::vector<Color> &c, std::vector<int> &out) {
    c[u] = GRAY;
    for (int v : g[u]) {
        if (c[v] == GRAY) {                // back edge: v is an ancestor of u
            std::printf("  back edge %s -> %s closes a cycle\n", TASK[u], TASK[v]);
            return false;
        }
        if (c[v] == WHITE && !dfsTopo(g, v, c, out)) return false;
    }
    c[u] = BLACK;
    out.push_back(u);                      // u finishes after all it leads to
    return true;
}

bool dfsOrder(const Graph &g, std::vector<int> &order) {
    std::vector<Color> c(V, WHITE);
    order.clear();
    for (int u = 0; u < V; ++u)
        if (c[u] == WHITE && !dfsTopo(g, u, c, order)) return false;
    std::reverse(order.begin(), order.end());   // decreasing finish time
    return true;
}

void print(const char *label, const std::vector<int> &order) {
    std::printf("%s", label);
    for (int u : order) std::printf(" %s", TASK[u]);
    std::printf("\n");
}

int main() {
    Graph g(V);
    for (const auto &a : ARCS) g[a[0]].push_back(a[1]);

    std::printf("Kahn's algorithm\n");
    std::vector<int> k = kahn(g, true);
    print("Kahn order:", k);

    std::vector<int> d;
    if (dfsOrder(g, d)) print("DFS order: ", d);
```

Listing 13.13 (Continued) *topo.cpp: topological sort by Kahn's algorithm and by depth-first search, with cycle detection.*

```
    std::printf("\nA mistaken arc shelve -> catalog is added\n");
    g[8].push_back(3);
    k = kahn(g, false);
    print("Kahn outputs only:", k);
    std::printf("  %zu of %d tasks: the rest wait on each other in a cycle\n",
                k.size(), V);
    if (!dfsOrder(g, d)) std::printf("DFS: no topological order exists\n");
    return 0;
}
```

```
Kahn's algorithm
  output order    frees: receive
  output receive  frees: inspect catalog
  output inspect  frees: stamp
  output catalog  frees: label register
  output stamp    frees:
  output label    frees: cover
  output register frees:
  output cover    frees: shelve
  output shelve   frees:
Kahn order: order receive inspect catalog stamp label register cover shelve
DFS order:  order receive catalog register label inspect stamp cover shelve

A mistaken arc shelve -> catalog is added
Kahn outputs only: order receive inspect stamp
  4 of 9 tasks: the rest wait on each other in a cycle
  back edge label -> cover closes a cycle
DFS: no topological order exists
```

The trace is **Table 13.3**, the DFS order is the one derived in Section 13.9.3, and after the mistaken arrow both algorithms refuse, exactly as **Figure 13.11** predicts. In `dfsTopo`, finished vertices are appended to the *end* of `out` and the whole vector is reversed once at the end; appending is $O(1)$ amortized, whereas inserting at the front of a `std::vector` would cost $\Theta(n)$ every time (Section 7.3). The Java and Python versions, **Listing 13.14** and **Listing 13.15**, follow.

Listing 13.14 `Topo.java`: topological sort in Java.

```java
// Topo.java -- topological sort of the Pustaka Ceria new-book workflow,
// by Kahn's algorithm and by depth-first search, with cycle detection.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Topo {
    static final int V = 9;
    static final String[] TASK = {"order", "receive", "inspect", "catalog", "stamp",
                                  "label", "cover", "register", "shelve"};
    // Arc u -> v means "task u must be finished before task v starts".
    static final int[][] ARCS = {{0, 1}, {1, 2}, {1, 3}, {2, 4}, {3, 5},
                                 {3, 7}, {4, 6}, {5, 6}, {6, 8}, {7, 8}};

    static List<Integer> kahn(List<List<Integer>> g, boolean trace) {
        int[] indeg = new int[V];
        List<Integer> order = new ArrayList<>();
        for (int u = 0; u < V; u++)
            for (int v : g.get(u)) indeg[v]++;
        ArrayDeque<Integer> q = new ArrayDeque<>();
        for (int u = 0; u < V; u++)
            if (indeg[u] == 0) q.addLast(u);
        while (!q.isEmpty()) {
            int u = q.pollFirst();
            order.add(u);
            if (trace) System.out.printf("  output %-8s frees:", TASK[u]);
            for (int v : g.get(u)) {
                if (--indeg[v] == 0) {              // last prerequisite done
                    q.addLast(v);
                    if (trace) System.out.print(" " + TASK[v]);
                }
            }
            if (trace) System.out.println();
        }
        return order;                                // shorter than V: a cycle
    }

    static final int WHITE = 0, GRAY = 1, BLACK = 2;
```

Listing 13.14 (Continued) *Topo.java: topological sort in Java.*

```
    static boolean dfsTopo(List<List<Integer>> g, int u, int[] c,
                           List<Integer> out) {
        c[u] = GRAY;
        for (int v : g.get(u)) {
            if (c[v] == GRAY) {                       // back edge
                System.out.printf("  back edge %s -> %s closes a cycle%n",
                        TASK[u], TASK[v]);
                return false;
            }
            if (c[v] == WHITE && !dfsTopo(g, v, c, out)) return false;
        }
        c[u] = BLACK;
        out.add(u);                                   // u finishes
        return true;
    }

    static boolean dfsOrder(List<List<Integer>> g, List<Integer> order) {
        int[] c = new int[V];                         // all WHITE
        order.clear();
        for (int u = 0; u < V; u++)
            if (c[u] == WHITE && !dfsTopo(g, u, c, order)) return false;
        Collections.reverse(order);                   // decreasing finish time
        return true;
    }

    static void print(String label, List<Integer> order) {
        System.out.print(label);
        for (int u : order) System.out.print(" " + TASK[u]);
        System.out.println();
    }

    public static void main(String[] args) {
        List<List<Integer>> g = new ArrayList<>();
        for (int i = 0; i < V; i++) g.add(new ArrayList<>());
        for (int[] a : ARCS) g.get(a[0]).add(a[1]);

        System.out.println("Kahn's algorithm");
        List<Integer> k = kahn(g, true);
        print("Kahn order:", k);

        List<Integer> d = new ArrayList<>();
        if (dfsOrder(g, d)) print("DFS order: ", d);
```

Listing 13.14 (Continued) *Topo.java: topological sort in Java.*

```
        System.out.println("\nA mistaken arc shelve -> catalog is added");
        g.get(8).add(3);
        k = kahn(g, false);
        print("Kahn outputs only:", k);
        System.out.printf(
                "  %d of %d tasks: the rest wait on each other in a cycle%n",
                k.size(), V);
        if (!dfsOrder(g, d)) System.out.println("DFS: no topological order exists");
    }
}
```

Listing 13.15 `topo.py`: topological sort in Python.

```
# topo.py -- topological sort of the Pustaka Ceria new-book workflow,
# by Kahn's algorithm and by depth-first search, with cycle detection.
from collections import deque

V = 9
TASK = ["order", "receive", "inspect", "catalog", "stamp",
        "label", "cover", "register", "shelve"]
# Arc (u, v) means "task u must be finished before task v starts".
ARCS = [(0, 1), (1, 2), (1, 3), (2, 4), (3, 5),
        (3, 7), (4, 6), (5, 6), (6, 8), (7, 8)]


def kahn(g, trace=False):
    indeg = [0] * V
    for u in range(V):
        for v in g[u]:
            indeg[v] += 1
    q = deque(u for u in range(V) if indeg[u] == 0)
    order = []
    while q:
        u = q.popleft()
        order.append(u)
        freed = []
        for v in g[u]:
            indeg[v] -= 1
            if indeg[v] == 0:               # v's last prerequisite is done
                q.append(v)
                freed.append(TASK[v])
        if trace:
            print(f"  output {TASK[u]:<8} frees:" + "".join(" " + t for t in freed))
    return order                            # shorter than V <=> a cycle


WHITE, GRAY, BLACK = 0, 1, 2
```

Listing 13.15 (Continued) *topo.py: topological sort in Python.*

```
def dfs_topo(g, u, color, out):
    color[u] = GRAY
    for v in g[u]:
        if color[v] == GRAY:                # back edge: v is an ancestor of u
            print(f"  back edge {TASK[u]} -> {TASK[v]} closes a cycle")
            return False
        if color[v] == WHITE and not dfs_topo(g, v, color, out):
            return False
    color[u] = BLACK
    out.append(u)                           # u finishes after all it leads to
    return True


def dfs_order(g):
    """Return a topological order, or None if the graph has a cycle."""
    color = [WHITE] * V
    out = []
    for u in range(V):
        if color[u] == WHITE and not dfs_topo(g, u, color, out):
            return None
    return out[::-1]                        # decreasing finish time


def show(label, order):
    print(label + "".join(" " + TASK[u] for u in order))


def main():
    g = [[] for _ in range(V)]
    for u, v in ARCS:
        g[u].append(v)

    print("Kahn's algorithm")
    show("Kahn order:", kahn(g, trace=True))
    d = dfs_order(g)
    if d is not None:
        show("DFS order: ", d)

    print("\nA mistaken arc shelve -> catalog is added")
    g[8].append(3)
    k = kahn(g)
    show("Kahn outputs only:", k)
    print(f"  {len(k)} of {V} tasks: the rest wait on each other in a cycle")
    if dfs_order(g) is None:
        print("DFS: no topological order exists")
```

Listing 13.15 (Continued) *topo.py: topological sort in Python.*

```
if __name__ == "__main__":
    main()
```

Both print exactly the output of **Listing 13.13**. Topological sorting is everywhere in software. The build tools `make` and `ninja` compile the files of a program in a topological order of their dependencies; package managers such as `apt` and `pip` install libraries in a topological order of what requires what, and report a “dependency cycle” when there is none; spreadsheet programs recompute cells in a topological order of the formulas that refer to them, and report a “circular reference” in exactly the situation of **Figure 13.11**. Python even ships the algorithm in its standard library, as the class `graphlib.TopologicalSorter`.

13.10 Weighted Shortest Paths: a Gentle Dijkstra

13.10.1 Why BFS is not enough

BFS answered "which route uses the fewest footpaths?". The librarian who pushes a trolley of books from the main library to the parking area cares about a different question: which route is *shortest in meters*? The two questions have different answers on the campus map. BFS's route LIB, ENG, AUD, PRK has 3 footpaths and $150 + 100 + 70 = 320$ meters; the route LIB, ADM, ENG, AUD, PRK has 4 footpaths but $80 + 60 + 100 + 70 = 310$ meters. BFS cannot find the second route, because it looks only at the number of edges: it reaches ENG directly from LIB in one hop and never considers that the two-hop detour through ADM (140 meters) is shorter than the direct path (150 meters).

When weights differ, the vertex to settle next is not the one discovered first but the one that is currently *closest*. Replacing BFS's first-in, first-out queue by a queue that always hands out the closest vertex gives the algorithm published by Edsger W. Dijkstra in 1959 [8], one of the most used algorithms in computing. It requires every weight to be nonnegative, which lengths, times and costs naturally are.

13.10.2 Relaxation

Dijkstra's algorithm keeps, for every vertex v, a *tentative distance* $d[v]$: the length of the shortest route from the source s to v found so far, ∞ if none has been found yet. It starts with $d[s] = 0$ and $d[v] = \infty$ for every other v, and improves the estimates by one basic step, called *relaxing* the edge (u, v):

$$\text{if } d[u] + w(u,v) < d[v]: \qquad d[v] \leftarrow d[u] + w(u,v), \quad \pi[v] \leftarrow u \tag{13.10}$$

In words: "I know a route to u of length $d[u]$. Going on from u to v costs $w(u, v)$ more. If that beats the best route to v I know, remember the new route, and remember that it arrives from u." Relaxation never makes an estimate wrong: $d[v]$ is always the length of some real route (or ∞), so it can only be too large, never too small. The whole art is to relax the edges in an order that makes every estimate exact as quickly as possible.

13.10.3 Dijkstra's algorithm

Dijkstra's order is greedy: repeatedly take the not-yet-final vertex with the smallest tentative distance, declare that distance *final*, and relax every edge leaving it. A *priority queue* (Section 5.15) is the structure that hands out "the entry with the smallest key" quickly: it supports *insert* and *extract-min*, each in $O(\log n)$ time when built as a binary heap, which is the subject of Chapter 14. This chapter uses the priority queues the three languages already provide: `std::priority_queue` in C++ [9], `java.util.PriorityQueue` in Java [10] and the `heapq` module in Python [11].

The simplest correct way to use such a library queue is the *lazy* version in **Pseudocode 13.7**. When relaxation lowers $d[v]$, we do not try to find and update v's old entry in the queue (library queues cannot do that); we simply insert a new entry $(d[v], v)$. The old, larger entry stays behind, and when it eventually comes out of the queue, v is already final and the entry is recognized as *stale* and skipped.

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))
```

Put **Pseudocode 13.7** next to **Pseudocode 13.1** and the family resemblance is plain. Both take a vertex out of a queue and look at its neighbors; BFS uses a FIFO queue and adds 1 per edge, Dijkstra uses a priority queue and adds $w(u, v)$. On a graph in which every weight is 1, Dijkstra's algorithm visits the vertices in exactly the order of BFS and computes the same distances.

13.10.4 Tracing Dijkstra from the main library

Table 13.4 runs **Pseudocode 13.7** on the campus map from LIB. Priority-queue entries are written (distance, vertex); ties between equal distances are broken by the smaller vertex number, as the programs do.

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[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 → 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)

Step 2 is where Dijkstra beats BFS. ADM is extracted with distance 80, and relaxing ADM-ENG finds a route to ENG of $80 + 60 = 140$ meters, better than the 150 meters of the direct footpath found at step 1. ENG's estimate drops and a new entry (140, ENG) is inserted; the old entry (150, ENG) stays in the queue. At step 4 the new entry comes out first and makes ENG final at 140; at step 5 the old entry comes out and is skipped as stale. Every later route through ENG (to AUD, and on to PRK) inherits the better 140. At step 6 relaxation also *fails* for once: from AUD, MED would cost $240 + 160 = 400$, worse than the 300 already known through CAF.

Figure 13.12 shows the result. Each vertex carries its final distance from LIB; the amber edges are the *shortest-path tree*, formed by the edges $\{\pi[v], v\}$. The dashed red footpath LIB-ENG is the

one BFS's tree used and Dijkstra's avoids. ANX and WHS keep the distance ∞: no route reaches them.

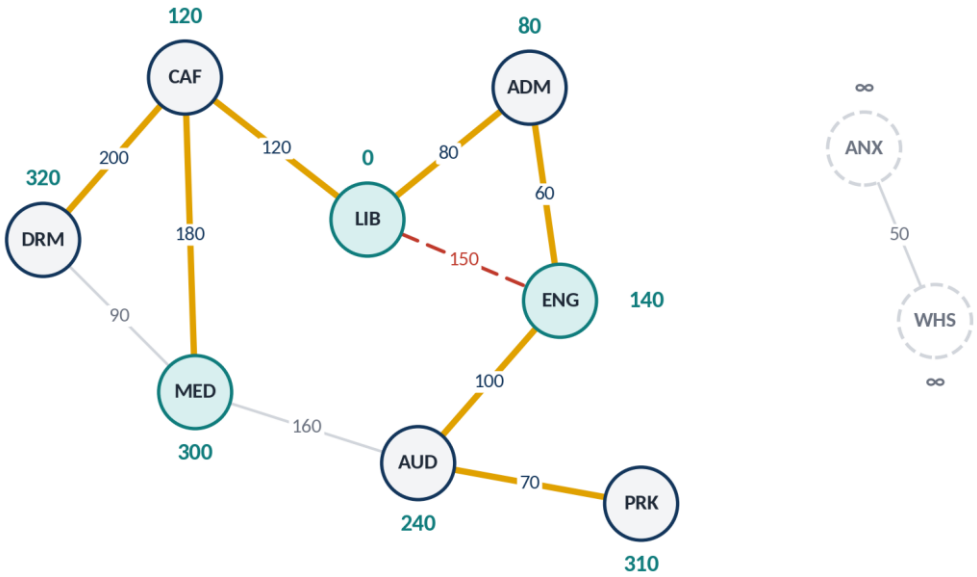


Figure 13.12 Shortest walking distances in meters from LIB. Amber edges form the shortest-path tree; the dashed red edge LIB-ENG is on the fewest-hop route to PRK but not on the shortest one.

13.10.5 Why it works, and when it does not

The key claim is that when a vertex u is extracted for the first time, its distance $d[u]$ is final, that is, equal to the true shortest distance $\delta(s,u)$. Here is the idea of the proof; Cormen et al. give it in full [2]. Suppose, to the contrary, that some shorter route P from s to u exists at that moment. P starts at s, which is final, and ends at u, which is not yet final, so somewhere along P there is a first vertex y that is not final, reached from a final vertex x. When x became final, the edge (x,y) was relaxed, so $d[y] \le \delta(s,x) + w(x,y)$, which is the length of the part of P up to y. Since all weights are nonnegative, the rest of P from y to u cannot have negative length, so $d[y] \le \text{length}(P) < d[u]$. But then the priority queue would have handed out y before u: a contradiction. So $d[u] = \delta(s,u)$.

The step "the rest of P cannot have negative length" is exactly where the assumption $w \ge 0$ is used, and it cannot be dropped. Consider three vertices s, a, b with $w(s,a) = 2$, $w(s,b) = 3$ and $w(b,a) = -2$. Dijkstra's algorithm extracts s, then a with distance 2, and declares it final; later it extracts b with distance 3, but a is already done, so the route s, b, a of length $3 - 2 = 1$ is never recorded. Negative weights do occur, for example as rebates in a cost model, and graphs with them need a different algorithm, such as Bellman-Ford [2], which is slower but correct.

13.10.6 Running time, and a look ahead to Chapter 14

Count the priority-queue operations of **Pseudocode 13.7**. Every successful relaxation inserts one entry, and each edge can be relaxed successfully at most once from each end, so there are at most $|E|$ insertions for a directed graph (or $2|E|$ for an undirected one) plus the one for s, and the same number of extractions. With a binary heap, each costs $O(\log E) = O(\log V)$, because $|E| \le |V|^2$ means $\log|E| \le 2\log|V|$. The rest of the work is $\Theta(V + E)$ as in BFS. Altogether,

$$T_{\text{Dijkstra}} = O((V + E)\log V) \tag{13.11}$$

For the campus map that is a handful of operations; for the 58 million arcs of a national road network it is still well under a second on a laptop. Two refinements are worth knowing by name. Dijkstra's original 1959 formulation used no priority queue at all, only an array scanned for the minimum, which gives $\Theta(V^2)$ and is in fact the better choice for dense graphs, where E is close to V^2. And with a more elaborate heap, the Fibonacci heap of Fredman and Tarjan [12], the bound improves to $O(E + V \log V)$.

> **A LOOK AHEAD.** Everything in Equation (13.11) that is not linear comes from the priority queue. Chapter 14 opens the box: it builds the binary heap that stands behind `std::priority_queue`, `PriorityQueue` and `heapq`, shows why *insert* and *extract-min* take $O(\log n)$, and adds the *decrease-key* operation that lets Dijkstra's algorithm update an entry instead of inserting a new one.

13.10.7 The programs

Listing 13.16 implements **Pseudocode 13.7**. The C++ `std::priority_queue` is a *max*-heap by default: its `top` is the largest element. Declaring it with the comparator `std::greater<Item>` turns it into a min-heap on pairs (distance, vertex), and because pairs compare first by their first component, `top` is the entry with the smallest distance, ties going to the smaller vertex number. The line `auto [du, u] = pq.top();` uses a structured binding (C++17) to unpack the pair.

Listing 13.16 `dijkstra.cpp`: Dijkstra's algorithm on the campus map with `std::priority_queue` as a min-heap.

```cpp
// dijkstra.cpp -- shortest walking distances from the main library,
// with a priority queue (std::priority_queue as a min-heap).
#include <climits>
#include <cstdio>
#include <deque>
#include <functional>
#include <queue>
#include <utility>
#include <vector>

const int V = 10;
const char *CODE[V] = {"LIB", "CAF", "ADM", "ENG", "DRM",
                       "MED", "AUD", "PRK", "ANX", "WHS"};
const int PATHS[][3] = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                        {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                        {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};
const int INF = INT_MAX;

struct Nbr {
    int to, w;
};
using Item = std::pair<int, int>;          // (tentative distance, vertex)

void dijkstra(const std::vector<std::vector<Nbr>> &adj, int s,
              std::vector<int> &dist, std::vector<int> &parent) {
    dist.assign(V, INF);
    parent.assign(V, -1);
    std::vector<bool> done(V, false);
    // greater<Item> turns the default max-heap into a min-heap
    std::priority_queue<Item, std::vector<Item>, std::greater<Item>> pq;
    dist[s] = 0;
    pq.push({0, s});
    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});
```

Listing 13.16 (Continued) *dijkstra.cpp: Dijkstra's algorithm on the campus map with std::priority_queue as a min-heap.*

```
                std::printf(" %s=%d", CODE[e.to], dist[e.to]);
            }
        }
        std::printf("\n");
    }
}

int main() {
    std::vector<std::vector<Nbr>> adj(V);
    for (const auto &p : PATHS) {
        adj[p[0]].push_back({p[1], p[2]});
        adj[p[1]].push_back({p[0], p[2]});
    }
    std::vector<int> dist, parent;
    std::printf("Dijkstra from LIB (relaxed distances after each pop)\n");
    dijkstra(adj, 0, dist, parent);

    std::printf("\nvertex  dist(m)  route\n");
    for (int v = 0; v < V; ++v) {
        if (dist[v] == INF) {
            std::printf("  %s   unreachable\n", CODE[v]);
            continue;
        }
        std::deque<int> path;
        for (int x = v; x != -1; x = parent[x]) path.push_front(x);
        std::printf("  %s  %7d  ", CODE[v], dist[v]);
        for (std::size_t i = 0; i < path.size(); ++i)
            std::printf("%s%s", i ? "-" : "", CODE[path[i]]);
        std::printf("\n");
    }
    return 0;
}
```

```
Dijkstra from LIB (relaxed distances after each pop)
  pop (  0, LIB)  final. CAF=120 ADM=80 ENG=150
  pop ( 80, ADM)  final. ENG=140
  pop (120, CAF)  final. DRM=320 MED=300
  pop (140, ENG)  final. AUD=240
  pop (150, ENG)  stale, skipped
  pop (240, AUD)  final. PRK=310
  pop (300, MED)  final.
  pop (310, PRK)  final.
  pop (320, DRM)  final.

vertex  dist(m)  route
  LIB        0  LIB
  CAF      120  LIB-CAF
  ADM       80  LIB-ADM
  ENG      140  LIB-ADM-ENG
  DRM      320  LIB-CAF-DRM
  MED      300  LIB-CAF-MED
  AUD      240  LIB-ADM-ENG-AUD
  PRK      310  LIB-ADM-ENG-AUD-PRK
  ANX   unreachable
  WHS   unreachable
```

The trace is **Table 13.4**, including the stale entry for ENG at step 5, and the route table is **Figure 13.12**. In Java (**Listing 13.17**), `PriorityQueue` is already a min-heap; it is given a comparator on the record `Item` that compares distances first and vertex numbers second. In Python, `heapq` keeps an ordinary list in heap order, and tuples `(distance, vertex)` compare exactly like C++ pairs.

Listing 13.17 `Dijkstra.java`: Dijkstra's algorithm with `java.util.PriorityQueue`.

```
// Dijkstra.java -- shortest walking distances from the main library,
// with a priority queue (java.util.PriorityQueue is a min-heap).
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.PriorityQueue;

public class Dijkstra {
    static final int V = 10;
    static final String[] CODE = {"LIB", "CAF", "ADM", "ENG", "DRM",
                                  "MED", "AUD", "PRK", "ANX", "WHS"};
    static final int[][] PATHS = {{0, 1, 120}, {0, 2, 80}, {0, 3, 150}, {1, 4, 200},
                                  {1, 5, 180}, {2, 3, 60}, {3, 6, 100}, {4, 5, 90},
                                  {5, 6, 160}, {6, 7, 70}, {8, 9, 50}};
    static final int INF = Integer.MAX_VALUE;

    record Nbr(int to, int w) {}
    record Item(int dist, int v) {}                 // (tentative distance, vertex)

    static void dijkstra(List<List<Nbr>> adj, int s, int[] dist, int[] parent) {
        Arrays.fill(dist, INF);
        Arrays.fill(parent, -1);
        boolean[] done = new boolean[V];
        PriorityQueue<Item> pq = new PriorityQueue<>((a, b) ->
                a.dist() != b.dist() ? Integer.compare(a.dist(), b.dist())
                                     : Integer.compare(a.v(), b.v()));
        dist[s] = 0;
        pq.add(new Item(0, s));
        while (!pq.isEmpty()) {
            Item it = pq.poll();
            int du = it.dist(), u = it.v();
            if (done[u]) {                          // an old, longer entry
                System.out.printf("  pop (%3d, %s)  stale, skipped%n", du, CODE[u]);
                continue;
            }
            done[u] = true;                         // du is now final
            System.out.printf("  pop (%3d, %s)  final.", du, CODE[u]);
            for (Nbr e : adj.get(u)) {
                if (!done[e.to()] && du + e.w() < dist[e.to()]) {   // relax
                    dist[e.to()] = du + e.w();
                    parent[e.to()] = u;
                    pq.add(new Item(dist[e.to()], e.to()));
                    System.out.printf(" %s=%d", CODE[e.to()], dist[e.to()]);
                }
            }
```

Listing 13.17 (Continued) *Dijkstra.java: Dijkstra's algorithm with java.util.PriorityQueue.*

```
            System.out.println();
        }
    }

    public static void main(String[] args) {
        List<List<Nbr>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        for (int[] p : PATHS) {
            adj.get(p[0]).add(new Nbr(p[1], p[2]));
            adj.get(p[1]).add(new Nbr(p[0], p[2]));
        }
        int[] dist = new int[V], parent = new int[V];
        System.out.println("Dijkstra from LIB (relaxed distances after each pop)");
        dijkstra(adj, 0, dist, parent);

        System.out.println("\nvertex  dist(m)  route");
        for (int v = 0; v < V; v++) {
            if (dist[v] == INF) {
                System.out.printf("  %s   unreachable%n", CODE[v]);
                continue;
            }
            ArrayDeque<Integer> path = new ArrayDeque<>();
            for (int x = v; x != -1; x = parent[x]) path.addFirst(x);
            StringBuilder sb = new StringBuilder();
            for (int x : path)
                sb.append(sb.length() > 0 ? "-" : "").append(CODE[x]);
            System.out.printf("  %s  %7d  %s%n", CODE[v], dist[v], sb);
        }
    }
}
```

Listing 13.18 `dijkstra.py`: Dijkstra's algorithm with the `heapq` module.

```
# dijkstra.py -- shortest walking distances from the main library,
# with a priority queue (the heapq module keeps a list as a min-heap).
import heapq
from collections import deque

V = 10
CODE = ["LIB", "CAF", "ADM", "ENG", "DRM", "MED", "AUD", "PRK", "ANX", "WHS"]
PATHS = [(0, 1, 120), (0, 2, 80), (0, 3, 150), (1, 4, 200), (1, 5, 180),
         (2, 3, 60), (3, 6, 100), (4, 5, 90), (5, 6, 160), (6, 7, 70),
         (8, 9, 50)]
INF = float("inf")


def dijkstra(adj, s):
    dist = [INF] * V
    parent = [-1] * V
    done = [False] * V
    dist[s] = 0
    pq = [(0, s)]                           # (tentative distance, vertex)
    while pq:
        du, u = heapq.heappop(pq)
        if done[u]:                         # an old, longer entry: ignore it
            print(f"  pop ({du:3}, {CODE[u]})  stale, skipped")
            continue
        done[u] = True                      # du is now final
        line = f"  pop ({du:3}, {CODE[u]})  final."
        for v, w in adj[u]:
            if not done[v] and du + w < dist[v]:     # relax u -> v
                dist[v] = du + w
                parent[v] = u
                heapq.heappush(pq, (dist[v], v))
                line += f" {CODE[v]}={dist[v]}"
        print(line)
    return dist, parent


def main():
    adj = [[] for _ in range(V)]
    for u, v, w in PATHS:
        adj[u].append((v, w))
        adj[v].append((u, w))
    print("Dijkstra from LIB (relaxed distances after each pop)")
    dist, parent = dijkstra(adj, 0)
```

Listing 13.18 (Continued) *dijkstra.py: Dijkstra's algorithm with the heapq module.*

```
    print("\nvertex  dist(m)  route")
    for v in range(V):
        if dist[v] == INF:
            print(f"  {CODE[v]}   unreachable")
            continue
        path, x = deque(), v
        while x != -1:
            path.appendleft(CODE[x])
            x = parent[x]
        print(f"  {CODE[v]}  {dist[v]:7}  " + "-".join(path))


if __name__ == "__main__":
    main()
```

Both print exactly the output of **Listing 13.16**. One detail differs silently: C++ and Java use `INT_MAX` and `Integer.MAX_VALUE` for ∞, which is safe here only because line 11 of **Pseudocode 13.7** never adds a weight to the distance of a vertex that has not been reached (an

extracted vertex always has a finite distance). Python uses `float("inf")`, for which $\infty + w$ is still ∞ and no overflow is possible.

13.11 Graphs at Real-World Scale

The campus map has ten vertices. The algorithms of this chapter are used every day on graphs a billion times larger, and it is instructive to see what changes and what does not.

13.11.1 Road networks

OpenStreetMap, the collaboratively edited map of the world, stores geography as a graph in all but name: *nodes* are points with coordinates, and *ways* are sequences of nodes that form roads, paths, rivers and building outlines. On September 23, 2026 its database held 10,882,374,688 nodes, 1,219,683,308 ways and 14,759,380 relations [13]. A way through k nodes contributes $k - 1$ segments, so the whole planet file describes roughly $10.9 - 1.2 \approx 9.7$ billion segments between consecutive nodes, not all of them roads. Research on route planning has long used a smaller and fixed benchmark, the road network of the United States prepared for the 9th DIMACS Implementation Challenge in 2006: 23,947,347 vertices (road junctions) and 58,333,344 arcs [14]. That benchmark is historical, but because it is fixed, it lets algorithms published twenty years apart be compared on exactly the same graph.

Both graphs are extremely sparse. In the DIMACS network the average out-degree is 58,333,344 divided by 23,947,347, about 2.4, and by Equation (13.5) the density is about $2 \times 29.2 \times 10^6 / (23.9 \times 10^6)^2 \approx 10^{-7}$. The consequences for representation are dramatic, and **Figure 13.13** plots them on a logarithmic scale, using the most generous assumptions for the matrix (one bit per cell, no weights) and a plain compact adjacency list (8 bytes per list entry and per vertex).

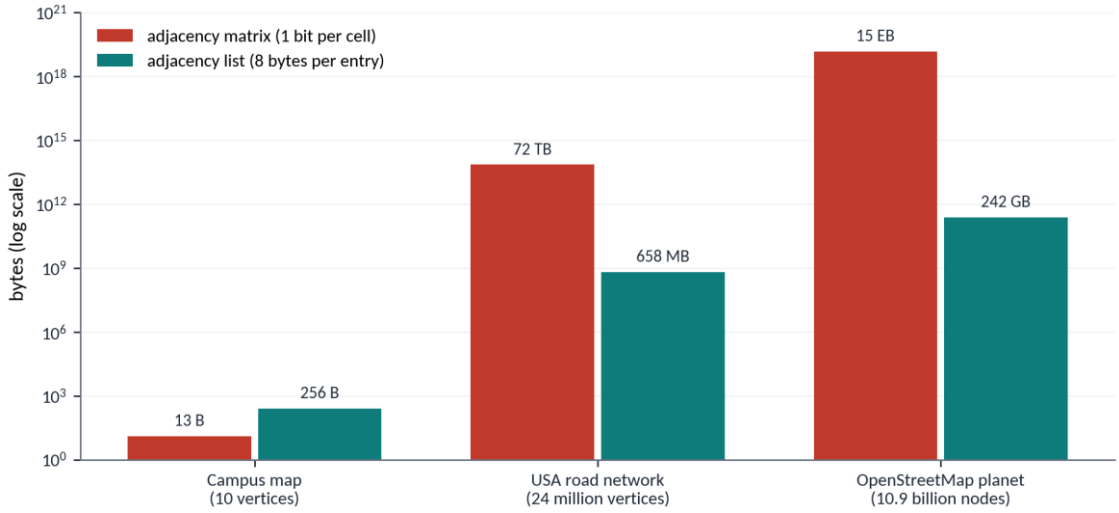


Figure 13.13 Memory for an adjacency matrix (one bit per cell) and an adjacency list (8 bytes per entry) of three road graphs: the campus map, the DIMACS USA network (2006 benchmark) and the OpenStreetMap planet (nodes and segments as of September 23, 2026). Logarithmic scale.

For the USA network, by Equation (13.6), the matrix needs $(2.39 \times 10^7)^2 \approx 5.7 \times 10^{14}$ cells, about 72 terabytes even at one bit per cell, while the adjacency list needs about 658 megabytes, which fits in the memory of a phone. For the OpenStreetMap planet the matrix would need about 1.2×10^{20} cells, some 15 exabytes, more than all the storage of a large data center, while the list needs about 240 gigabytes, a large but ordinary server. The matrix is not merely slower at this scale; it cannot exist. **Table 13.5** collects the figures.

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 [15]	3.60 billion daily active people	not published	not published	≈ 1.6 EB	depends on edges

What about time? BFS, DFS and the other linear-time algorithms of this chapter touch each vertex and edge a constant number of times, so on the USA network they need on the order of 10^8 elementary steps, well under a second on a current laptop. Dijkstra's algorithm, by Equation (13.11), is only a logarithmic factor slower: a single query from one junction to all others takes a few seconds. That is fast for one question, but a navigation service answers millions of questions a minute and must reply in milliseconds. Real route planners therefore *preprocess* the road network once, for example by *contraction hierarchies*, which add shortcut edges that let a slightly modified Dijkstra skip most of the graph, and then answer each query thousands of times faster than plain Dijkstra would [16]. Underneath the shortcuts, though, the algorithm is still the Dijkstra of Section 13.10, and the graph is still an adjacency list.

13.11.2 Social graphs

A social network is a graph whose vertices are people and whose edges are friendships (undirected) or "follows" (directed). The largest is operated by Meta, whose family of apps (Facebook, Instagram, WhatsApp and Messenger) had 3.60 billion *daily active people* on average in June 2026, according to its report for the second quarter of 2026 [15]. The friendship graph behind those numbers is not published, but a well-known historical measurement gives an idea of its shape: in February 2016, Facebook's researchers computed that the average distance between two of its then 1.6 billion users was 4.57 hops, that is, 3.57 intermediate people, "three and a half degrees of separation" [17]. Computing an average distance over billions of people needs, in principle, a BFS from every vertex, and the researchers used an approximate counting technique instead, precisely because even linear time per source is too much when there are billions of sources.

Social graphs differ from road networks in one important way: their degrees are very uneven. Most people have a few hundred connections, but a few accounts have hundreds of millions of followers. For such a vertex, the $O(\deg(u))$ edge test of an adjacency list in **Table 13.1** becomes expensive, and real systems keep the neighbor lists of large vertices sorted or hashed (Chapter 15) so that "does u follow v?" is answered without scanning. Everything else in this chapter carries over unchanged: "people you may know" suggestions start from the vertices at distance 2 in a BFS, and whether two accounts are connected at all is a component question.

13.11.3 What changes at scale, and what does not

Three lessons carry over from these numbers to any graph program you write.

- **The representation is chosen by E, not by V.** Real graphs are sparse, so adjacency lists (or their compact array form, in which all lists are concatenated into one array with an index of where each list starts) are the default.
- **Linear time is the goal.** An algorithm that is $\Theta(V + E)$ runs on a planet; one that is $\Theta(V^2)$ does not, whatever the constant.
- **Recursion depth becomes a real limit.** A DFS on a road network can go millions of vertices deep, far beyond any call stack, so production code uses the explicit stack of Section 13.7.5.

13.12 Choosing an Algorithm: Costs at a Glance

Table 13.6 summarizes the algorithms of this chapter. All costs assume adjacency lists; with an adjacency matrix, replace every $V + E$ by V^2.

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	$O((V + E)\log V)$	$O(V + E)$

The middle column tells the story of the whole book in miniature. The same loop, "take a vertex out of a container and look at its neighbors", becomes breadth-first search with the queue of Chapter 5, depth-first search with the stack of Chapter 5 or the recursion of Chapter 10, and Dijkstra's algorithm with the priority queue of Chapter 14. The container decides the order of the search, and the order decides what the search computes.

13.13 Profitina in Practice

Profitina is an AI visibility (GEO) platform: it scans a brand's website and measures how ready the site is to be found and cited by AI answer engines, and then helps fix what it finds [18]. A website is a directed graph: every page is a vertex, and every hyperlink from page u to page v is an arrow (u, v). Crawlers, both the classic search-engine kind and the ones that feed AI answer engines, discover a site by following those arrows from the pages they already know, which since the early days of web search has meant that the link structure of a site decides what gets found at all [19]. Several questions a site scanner asks are therefore exactly the questions of this chapter; we

describe the shape of each and the class of algorithm that fits it, not the product's internal implementation.

- **Can every page be reached from the home page?** This is reachability in a directed graph, answered by one BFS from the home page. A page that is listed in the site's sitemap but not reachable by links is an *orphan page*: a visitor cannot click their way to it, and a crawler that follows links will not find it either.
- **How many clicks deep is each page?** The hop distance from the home page, which BFS computes as a by-product. Pages buried many clicks deep tend to be visited, and crawled, less often, so the depth is worth reporting.
- **Which pages are linked to most?** The in-degree of each vertex, counted in one pass over the arrows. A page with in-degree 0, apart from the home page, is an orphan by definition.
- **Do the links form loops?** Of course they do, and that is fine: a website is not a DAG, and a crawler must mark visited pages exactly as BFS does, or it will follow a menu link back to the home page forever.

The runnable version of this scenario uses the website of Profitina Laundry, the case study used throughout this book, with twelve pages taken from its sitemap and fifteen links. **Figure 13.14** draws the site graph with the click depth of each page, and **Listing 13.19** computes it.

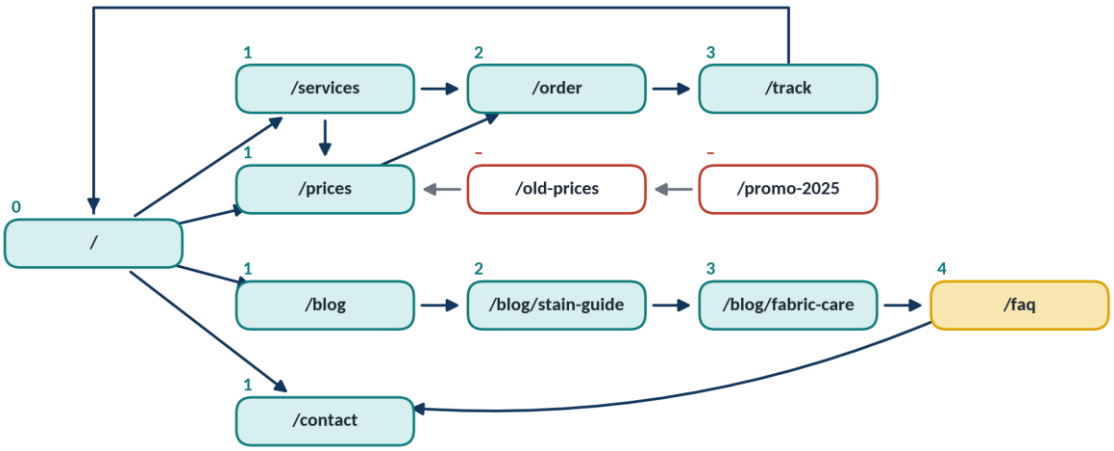


Figure 13.14 The Profitina Laundry website as a directed graph. The number above each page is its click depth from the home page; /faq (amber) is four clicks deep, and the two pages outlined in red are orphans that no link path reaches.

Listing 13.19 `laundry_site.cpp`: click depth, inbound links and orphan pages of the Profitina Laundry website, by breadth-first search.

```cpp
// laundry_site.cpp -- the Profitina Laundry website as a directed graph:
// click depth from the home page (BFS), orphan pages and inbound links.
#include <cstdio>
#include <queue>
#include <string>
#include <vector>

const std::vector<std::string> PAGE = {
    "/",          "/services",   "/prices",           "/order",
    "/track",     "/blog",       "/blog/stain-guide", "/blog/fabric-care",
    "/faq",       "/contact",    "/promo-2025",       "/old-prices"};
// Link u -> v: page u contains a hyperlink to page v.
const int LINKS[][2] = {{0, 1}, {0, 2}, {0, 5},  {0, 9},  {1, 2}, {1, 3},
                        {2, 3}, {3, 4}, {4, 0},  {5, 6},  {6, 7}, {7, 8},
                        {8, 9}, {10, 11}, {11, 2}};
const int MAX_DEPTH = 3;   // a page more than 3 clicks from home is hard to find

int main() {
    const int n = static_cast<int>(PAGE.size());
    std::vector<std::vector<int>> out(n);
    std::vector<int> inbound(n, 0);
    for (const auto &l : LINKS) {
        out[l[0]].push_back(l[1]);
        ++inbound[l[1]];
    }

    std::vector<int> depth(n, -1);          // BFS from the home page
    std::queue<int> q;
    depth[0] = 0;
    q.push(0);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : out[u])
            if (depth[v] == -1) {
                depth[v] = depth[u] + 1;
                q.push(v);
            }
    }
```

Listing 13.19 (Continued) *laundry_site.cpp: click depth, inbound links and orphan pages of the Profitina Laundry website, by breadth-first search.*

```cpp
    std::printf("%-20s %6s %8s\n", "page", "clicks", "inbound");
    for (int v = 0; v < n; ++v) {
        if (depth[v] == -1) std::printf("%-20s %6s %8d\n", PAGE[v].c_str(), "-",
                                        inbound[v]);
        else std::printf("%-20s %6d %8d\n", PAGE[v].c_str(), depth[v], inbound[v]);
    }
    std::printf("\nOrphan pages (in the sitemap, unreachable from /):");
    for (int v = 0; v < n; ++v)
        if (depth[v] == -1) std::printf(" %s", PAGE[v].c_str());
    std::printf("\nPages deeper than %d clicks:", MAX_DEPTH);
    for (int v = 0; v < n; ++v)
        if (depth[v] > MAX_DEPTH) std::printf(" %s", PAGE[v].c_str());
    std::printf("\n");
    return 0;
}
```

```
page                 clicks  inbound
/                         0        1
/services                 1        1
/prices                   1        3
/order                    2        2
/track                    3        1
/blog                     1        1
/blog/stain-guide         2        1
/blog/fabric-care         3        1
/faq                      4        1
/contact                  1        2
/promo-2025               -        0
/old-prices               -        1

Orphan pages (in the sitemap, unreachable from /): /promo-2025 /old-prices
Pages deeper than 3 clicks: /faq
```

The results read like a site audit. The page `/prices` has three inbound links, the most of any page, because it is reached from the home page, from `/services` and from the old price page. The FAQ page is four clicks deep, one more than the threshold of three the program uses, because the only path to it runs through the whole blog; one link from the home page or from `/services` would bring it to depth 1 or 2. The pages `/promo-2025` and `/old-prices` are orphans: the old promotion still links to the old price list, and the old price list even links to the current prices, but nothing links *to* the promotion, so BFS from `/` never gets there. The fix is a decision for the site owner, not for the program: link the pages from somewhere, or retire them. The program only makes the structure visible, which is the right division of labor between a tool and its user. The Java and Python versions, **Listing 13.20** and **Listing 13.21**, print the same audit.

Listing 13.20 `LaundrySite.java`: the website audit in Java.

```java
// LaundrySite.java -- the Profitina Laundry website as a directed graph:
// click depth from the home page (BFS), orphan pages and inbound links.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class LaundrySite {
    static final String[] PAGE = {
        "/",      "/services", "/prices",           "/order",
        "/track", "/blog",     "/blog/stain-guide", "/blog/fabric-care",
        "/faq",   "/contact",  "/promo-2025",       "/old-prices"};
    // Link u -> v: page u contains a hyperlink to page v.
    static final int[][] LINKS = {{0, 1}, {0, 2}, {0, 5}, {0, 9}, {1, 2}, {1, 3},
                                  {2, 3}, {3, 4}, {4, 0}, {5, 6}, {6, 7}, {7, 8},
                                  {8, 9}, {10, 11}, {11, 2}};
    static final int MAX_DEPTH = 3;

    public static void main(String[] args) {
        int n = PAGE.length;
        List<List<Integer>> out = new ArrayList<>();
        for (int i = 0; i < n; i++) out.add(new ArrayList<>());
        int[] inbound = new int[n];
        for (int[] l : LINKS) {
            out.get(l[0]).add(l[1]);
            inbound[l[1]]++;
        }

        int[] depth = new int[n];                    // BFS from the home page
        Arrays.fill(depth, -1);
        ArrayDeque<Integer> q = new ArrayDeque<>();
        depth[0] = 0;
        q.addLast(0);
        while (!q.isEmpty()) {
            int u = q.pollFirst();
            for (int v : out.get(u))
                if (depth[v] == -1) {
                    depth[v] = depth[u] + 1;
                    q.addLast(v);
                }
        }
```

Listing 13.20 (Continued) *LaundrySite.java: the website audit in Java.*

```java
        System.out.printf("%-20s %6s %8s%n", "page", "clicks", "inbound");
        for (int v = 0; v < n; v++)
            System.out.printf("%-20s %6s %8d%n", PAGE[v],
                    depth[v] == -1 ? "-" : String.valueOf(depth[v]), inbound[v]);
        System.out.print("\nOrphan pages (in the sitemap, unreachable from /):");
        for (int v = 0; v < n; v++)
            if (depth[v] == -1) System.out.print(" " + PAGE[v]);
        System.out.printf("%nPages deeper than %d clicks:", MAX_DEPTH);
        for (int v = 0; v < n; v++)
            if (depth[v] > MAX_DEPTH) System.out.print(" " + PAGE[v]);
        System.out.println();
    }
}
```

Listing 13.21 `laundry_site.py`: the website audit in Python.

```
# laundry_site.py -- the Profitina Laundry website as a directed graph:
# click depth from the home page (BFS), orphan pages and inbound links.
from collections import deque

PAGE = ["/", "/services", "/prices", "/order",
        "/track", "/blog", "/blog/stain-guide", "/blog/fabric-care",
        "/faq", "/contact", "/promo-2025", "/old-prices"]
# Link (u, v): page u contains a hyperlink to page v.
LINKS = [(0, 1), (0, 2), (0, 5), (0, 9), (1, 2), (1, 3),
         (2, 3), (3, 4), (4, 0), (5, 6), (6, 7), (7, 8),
         (8, 9), (10, 11), (11, 2)]
MAX_DEPTH = 3        # a page more than 3 clicks from home is hard to find


def main():
    n = len(PAGE)
    out = [[] for _ in range(n)]
    inbound = [0] * n
    for u, v in LINKS:
        out[u].append(v)
        inbound[v] += 1

    depth = [-1] * n                          # BFS from the home page
    depth[0] = 0
    q = deque([0])
    while q:
        u = q.popleft()
        for v in out[u]:
            if depth[v] == -1:
                depth[v] = depth[u] + 1
                q.append(v)

    print(f"{'page':<20} {'clicks':>6} {'inbound':>8}")
    for v in range(n):
        shown = "-" if depth[v] == -1 else str(depth[v])
        print(f"{PAGE[v]:<20} {shown:>6} {inbound[v]:>8}")
    print("\nOrphan pages (in the sitemap, unreachable from /):"
          + "".join(" " + PAGE[v] for v in range(n) if depth[v] == -1))
    print(f"Pages deeper than {MAX_DEPTH} clicks:"
          + "".join(" " + PAGE[v] for v in range(n) if depth[v] > MAX_DEPTH))


if __name__ == "__main__":
    main()
```

Graphs appear elsewhere in the product as well. Profitina's real-time community lets users follow channels, which is a directed graph from users to channels, and the "N new" counters on the dashboard are computed along exactly those arrows. And the Content Engine's advice about internal linking amounts to advice about the in-degrees and depths computed above.

13.14 Common Mistakes, and How to Avoid Them

Forgetting to mark visited vertices. A traversal without a "visited" mark walks around the first cycle forever. Trees never needed a mark (Chapter 11); graphs always do.

Marking in BFS when a vertex leaves the queue instead of when it enters. The result is still correct, but a vertex can be enqueued once per neighbor, and the queue can grow to $|E|$ entries. Mark on entry, as line 11 of **Pseudocode 13.1** does (Section 13.6.2).

Storing an undirected edge in only one direction. `adj[u].push_back(v)` without `adj[v].push_back(u)` silently turns the graph into a directed one, and searches then miss vertices. The handshake check of Section 13.5.5, sum of list lengths equals $2|E|$, catches it.

Using a matrix for a large sparse graph. It wastes $\Theta(V^2)$ memory and turns every traversal into $\Theta(V^2)$ time (**Table 13.1**, **Figure 13.13**).

Building a Python matrix as `[[0] * n] * n`. All rows are the same list object; writing one cell changes a whole column. Use `[[0] * n for _ in range(n)]` (Section 13.5.6).

Using `list.pop(0)` as a queue in Python. Each call costs $\Theta(n)$, making BFS quadratic. Use `collections.deque` (Section 13.6.7).

Expecting BFS to find the shortest route in a weighted graph. BFS minimizes the number of edges, not their total weight; on the campus map it picks a route 10 meters longer than the shortest (Section 13.10.1).

Using Dijkstra's algorithm with negative weights. Its "extracted means final" rule is then false, and it can return wrong distances without any warning (Section 13.10.5).

Treating every visited neighbor as a cycle in an undirected graph. The parent is always a visited neighbor; exclude it, or every edge will look like a cycle (Section 13.7.6).

Using two colors for cycle detection in a directed graph. An arrow into a finished (black) vertex is not a cycle; only an arrow into a gray vertex is (Section 13.9.4).

Recursing too deep. Recursive DFS overflows the call stack on long paths, and Python stops at 1000 levels by default. Use the explicit stack of **Pseudocode 13.3** for large graphs.

Relying on one particular BFS or DFS order. The visiting order depends on the order of the adjacency lists, which depends on the input. Distances, components and the existence of a cycle do not, but the exact trees and time stamps do; a test that expects one exact order is testing the input file as much as the algorithm.

13.15 Chapter Summary

- A **graph** $G = (V, E)$ is a set of vertices and a set of edges (Equation (13.1)); edges may be undirected or directed and may carry weights. It generalizes lists and trees: a tree is a connected acyclic graph.
- The **degree** of a vertex counts its edges; the **handshake lemma** says the degrees add up to $2|E|$ (Equation (13.2)), and in a directed graph in-degrees and out-degrees each add up to $|E|$ (Equation (13.3)). A simple graph has at most $|V|(|V|-1)/2$ edges (Equation (13.4)); real graphs are sparse (Equation (13.5)).
- **Paths, cycles, components.** Reachability splits an undirected graph into **connected components**; a **DAG** is a directed graph without directed cycles.
- **Representations.** The **adjacency matrix** answers edge queries in $O(1)$ but needs $\Theta(V^2)$ memory and time to list neighbors; the **adjacency list** needs $\Theta(V + E)$ memory and lists the neighbors of u in $\Theta(\deg(u))$; the **edge list** is the natural input format (Equation (13.6), **Table 13.1**). Traversals want lists.
- **BFS** uses a queue, visits vertices in rings of equal hop distance, and computes fewest-hop distances and routes in $\Theta(V + E)$ (**Pseudocode 13.1**, Equation (13.8)); every edge satisfies $d[v] \leq d[u] + 1$ (Equation (13.7)).
- **DFS** uses recursion or an explicit stack of (vertex, next-neighbor) pairs, records discovery and finish times whose intervals nest like parentheses (Equation (13.9)), and runs in $\Theta(V + E)$. In an undirected graph, a back edge to a non-parent means a cycle.

- **Connected components** are found by one search per unlabeled vertex, in $\Theta(V + E)$ in total (**Pseudocode 13.4**).
- **Topological sort** orders a DAG so that every arrow points forward: **Kahn's algorithm** repeatedly removes a vertex of in-degree 0, and the **DFS method** lists vertices by decreasing finish time. Kahn outputting fewer than $|V|$ vertices, or DFS meeting a gray vertex, proves a directed cycle.
- **Dijkstra's algorithm** finds shortest weighted routes when all weights are nonnegative by repeatedly finalizing the closest vertex and **relaxing** its edges (Equation (13.10)); with a binary-heap priority queue it runs in $O((V + E)\log V)$ (Equation (13.11)). Chapter 14 builds that heap.
- **At scale**, road networks of tens of millions to billions of vertices and social networks of billions of people are handled by exactly these algorithms on adjacency lists; matrices and deep recursion are out of the question.

13.16 Exercises

13.1 For the undirected graph with $V = \{1,2,3,4,5,6\}$ and edges $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, $\{3,4\}$, $\{5,6\}$: (a) give the degree of every vertex and check the handshake lemma; (b) list a cycle; (c) give the connected components; (d) compute the density.

13.2 Can a simple undirected graph have the degree sequence (a) 3, 3, 3, 3, 3; (b) 3, 3, 3, 3; (c) 5, 1, 1, 1? For each, either draw such a graph or explain why none exists.

13.3 A graph has $|V| = 10^5$ vertices and $|E| = 3 \times 10^5$ undirected edges. Using Equation (13.6), how many cells does an adjacency matrix need, and how many entries does an adjacency list need? If a matrix cell takes 1 byte and a list entry 8 bytes, how many bytes does each use?

13.4 Run BFS by hand on the campus map from PRK, with neighbors in the adjacency-list order of **Figure 13.4**(b). Give the visiting order, the distance and parent of every vertex, and the fewest-hop route from PRK to LIB.

13.5 Change lines 10 and 11 of **Pseudocode 13.1** so that a vertex is marked only when it is dequeued (and skipped if already marked). Run the changed algorithm from LIB on the campus map and list the queue contents after each step. Which vertices are enqueued more than once? Are the resulting distances still correct?

13.6 The "simple" iterative DFS pushes the source; then, while the stack is not empty, it pops a vertex, skips it if visited, and otherwise marks it and pushes its unvisited neighbors in *reverse* list order. Run it from LIB on the campus map, listing the stack after each pop. Does it visit the vertices in the same order as **Pseudocode 13.2**? How many entries does the stack hold at most, and how does that compare with **Pseudocode 13.3**?

13.7 Run `DFS-VISIT` of **Pseudocode 13.2** from ADM (only that one start) on the campus map and give the discovery and finish time of every vertex it reaches. Draw the parenthesis structure.

13.8 How many connected components does the campus map (without the shuttle road) have after removing (a) the footpath ENG-AUD; (b) the footpath AUD-PRK; (c) both LIB-CAF and MED-AUD? Which footpaths of the original map are bridges?

13.9 Give two topological orders of the new-book workflow that differ from both orders of Section 13.9. How would you change Kahn's algorithm so that, whenever several tasks are available, it always picks the one that comes first alphabetically? What order does it then produce?

13.10 A graph is *bipartite* if its vertices can be colored with two colors so that every edge joins two vertices of different colors. Write a function that decides this with BFS, giving every newly discovered vertex the color opposite to its parent. Test it on the campus map and on a cycle of four vertices. Explain why a graph that contains a cycle of odd length can never be bipartite.

13.11 The reading room of Pustaka Ceria is drawn as a grid in which `#` is a bookshelf, `S` the entrance and `T` the reading desk:

```
S..#....
.#.#.##.
.#...#..
.####.#.
......#T
```

A reader moves one cell up, down, left or right per step and cannot pass through shelves. Treating each free cell as a vertex, find the fewest steps from S to T with BFS, and mark one shortest route.

13.12 Write a function that counts, for every vertex of a DAG, the number of different directed paths from a source s to it, by processing the vertices in a topological order. How many paths lead from *order* to *shelve* in the new-book workflow, and from *order* to *cover*?

13.13 Run Dijkstra's algorithm by hand from PRK on the campus map, in the style of **Table 13.4**. Give the resulting distances and the shortest route from PRK to CAF. Is any entry stale?

13.14 Using the three-vertex example of Section 13.10.5, show step by step how Dijkstra's algorithm computes a wrong distance. Then show that adding the same constant to every weight to make them all nonnegative does *not* fix the problem in general, using a small example.

13.15 For each situation, choose an adjacency matrix, an adjacency list or an edge list, and justify the choice with **Table 13.1**: (a) the 200 airports of a country, where almost every pair has a direct flight and the program mostly asks whether a direct flight exists; (b) the footpaths of a campus with 3,000 buildings, used for route planning; (c) a file of 50 million road segments to be sorted by length; (d) the follower graph of a social network.

13.16 Euler showed that a connected graph has a walk that uses every edge exactly once if and only if it has zero or two vertices of odd degree. (a) Which of the Königsberg land masses would need an extra bridge, and how few new bridges would make such a walk possible? (b) Does the campus map (restricted to its main component) have such a walk?

13.17 Solutions to the Exercises

13.1 (a) $\deg(1)=2$, $\deg(2)=2$, $\deg(3)=3$, $\deg(4)=1$, $\deg(5)=1$, $\deg(6)=1$; the sum is $10=2\times5$. (b) 1, 2, 3, 1. (c) $\{1,2,3,4\}$ and $\{5,6\}$. (d) By Equation (13.5), $\rho=5/(6\times5/2)=5/15=1/3$.

13.2 (a) Impossible: the degree sum 15 is odd, contradicting Equation (13.2). (b) Possible: the complete graph on four vertices, in which every vertex is adjacent to the other three. (c) Impossible: in a simple graph on four vertices a degree is at most 3, so no vertex can have degree 5. (Note that the sum, 8, is even; the handshake lemma is necessary but not sufficient.)

13.3 Matrix: $|V|^2=10^{10}$ cells, 10 GB at one byte each. List: $|V|+2|E|=10^5+6\times10^5=7\times10^5$ entries, 5.6 MB at 8 bytes each. The matrix is about 1,800 times larger, and the density is only $3\times10^5/(10^5\times99999/2)\approx6\times10^{-5}$.

13.4 The queue evolves as follows: PRK; AUD; ENG, MED; MED, LIB, ADM; LIB, ADM, CAF, DRM; and then the remaining vertices leave one by one. Visiting order: PRK, AUD, ENG, MED, LIB, ADM, CAF, DRM. Distances: PRK 0; AUD 1 (parent PRK); ENG 2 and MED 2 (parent AUD); LIB 3 and

ADM 3 (parent ENG); CAF 3 and DRM 3 (parent MED); ANX and WHS unreached. Route to LIB: PRK, AUD, ENG, LIB, 3 hops. The result was checked with the program of **Listing 13.6**.

13.5 With marking at dequeue time, the queue receives every not-yet-*marked* neighbor, including those already waiting. From LIB: after LIB, the queue is CAF ADM ENG; after CAF, ADM ENG DRM MED; after ADM (whose neighbor ENG is waiting but not marked), ENG DRM MED ENG; after ENG, DRM MED ENG AUD; after DRM (MED is waiting, unmarked), MED ENG AUD MED; after MED, ENG AUD MED AUD; the second ENG is skipped; after AUD, MED AUD PRK; then MED and AUD are skipped and PRK is processed. ENG, MED and AUD are each enqueued twice. The distances remain correct in this run because a vertex's distance is fixed when it is first marked, and the queue is still in nondecreasing order of distance; but 11 vertices were enqueued instead of 8, and in a dense graph the queue can hold $\Theta(E)$ entries instead of at most $|V|$.

13.6 Stack after each pop (top on the right): start LIB; pop LIB, push ENG ADM CAF (reverse order, so CAF on top): ENG ADM CAF; pop CAF, push MED DRM: ENG ADM MED DRM; pop DRM, push MED: ENG ADM MED MED; pop MED, push AUD: ENG ADM MED AUD; pop AUD, push PRK and ENG (ENG is not yet visited): ENG ADM MED PRK ENG; pop ENG, push ADM: ENG ADM MED PRK ADM; pop ADM: ENG ADM MED PRK; pop PRK; then MED, ADM and ENG are popped and skipped. The visiting order LIB, CAF, DRM, MED, AUD, ENG, ADM, PRK is the same as the recursive one here, but the stack held up to 5 entries including duplicates (MED twice, ADM twice, ENG twice), whereas **Pseudocode 13.3** holds only the current chain of gray vertices, one entry each. In general the simple version can hold $\Theta(E)$ entries and produces no finish times.

13.7 ADM 1/16, LIB 2/15, CAF 3/14, DRM 4/13, MED 5/12, AUD 6/11, ENG 7/8, PRK 9/10; ANX and WHS are not reached. Parentheses: (ADM (LIB (CAF (DRM (MED (AUD (ENG ENG) (PRK PRK) AUD) MED) DRM) CAF) LIB) ADM). These times were checked by program.

13.8 (a) Still 2: AUD remains connected to the campus through MED. (b) 3: PRK becomes isolated, alongside the main campus and ANX-WHS. (c) 3: with LIB-CAF and MED-AUD both gone, no footpath leaves the triangle CAF, DRM, MED, so the components are {LIB,ADM,ENG,AUD,PRK}, {CAF,DRM,MED} and {ANX,WHS}. The bridges of the original map are AUD-PRK and ANX-WHS; every other footpath lies on a cycle.

13.9 For example, order, receive, catalog, label, register, inspect, stamp, cover, shelve; and order, receive, inspect, catalog, register, label, stamp, cover, shelve. (Check that every arrow of **Figure 13.10**(a) points forward in each.) To always pick the alphabetically first available task, replace the FIFO queue of **Pseudocode 13.5** by a min-priority queue keyed on the task name. The order becomes: order, receive, catalog (before inspect), inspect, label (before register and stamp), register, stamp, cover, shelve.

13.10 See `bipartite` in **Listing 13.22**. The campus map is not bipartite; the four-cycle is, with colors 0, 1, 0, 1, as the output shows. Along any cycle the colors must alternate, so a cycle returns to its starting color only after an even number of edges; a cycle of odd length, such as the triangle CAF, DRM, MED, would need its first vertex to have both colors.

Listing 13.22 `exercises.cpp`: solutions to Exercises 13.10 to 13.12.

```cpp
// exercises.cpp -- solutions to the programming exercises of Chapter 13:
// two-coloring, a grid maze by BFS, and counting paths in a DAG.
#include <cstdio>
#include <queue>
#include <string>
#include <vector>

using Graph = std::vector<std::vector<int>>;

Graph undirected(int n, const std::vector<std::pair<int, int>> &es) {
    Graph g(n);
    for (auto [u, v] : es) {
        g[u].push_back(v);
        g[v].push_back(u);
    }
    return g;
}

// Exercise 13.10: can the vertices be colored 0/1 so every edge joins two
// different colors? BFS gives each new vertex the color opposite its parent.
bool bipartite(const Graph &g, std::vector<int> &color) {
    color.assign(g.size(), -1);
    for (int s = 0; s < static_cast<int>(g.size()); ++s) {
        if (color[s] != -1) continue;
        std::queue<int> q;
        color[s] = 0;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : g[u]) {
                if (color[v] == -1) {
                    color[v] = 1 - color[u];
                    q.push(v);
                } else if (color[v] == color[u]) {
                    return false;          // an edge inside one color: odd cycle
                }
            }
        }
    }
    return true;
}
```

Listing 13.22 (Continued) *exercises.cpp: solutions to Exercises 13.10 to 13.12.*

```
// Exercise 13.11: fewest steps from S to T in a grid; '#' is a bookshelf.
int gridBfs(std::vector<std::string> &grid) {
    int R = static_cast<int>(grid.size()), C = static_cast<int>(grid[0].size());
    std::vector<std::vector<int>> dist(R, std::vector<int>(C, -1));
    std::vector<std::vector<int>> from(R, std::vector<int>(C, -1));
    std::queue<int> q;                    // a cell (r, c) is stored as r*C + c
    int s = -1, t = -1;
    for (int r = 0; r < R; ++r)
        for (int c = 0; c < C; ++c) {
            if (grid[r][c] == 'S') s = r * C + c;
            if (grid[r][c] == 'T') t = r * C + c;
        }
    dist[s / C][s % C] = 0;
    q.push(s);
    const int dr[4] = {-1, 1, 0, 0}, dc[4] = {0, 0, -1, 1};
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        int r = u / C, c = u % C;
        for (int k = 0; k < 4; ++k) {      // the four neighbors of a cell
            int nr = r + dr[k], nc = c + dc[k];
            if (nr < 0 || nr >= R || nc < 0 || nc >= C) continue;
            if (grid[nr][nc] == '#' || dist[nr][nc] != -1) continue;
            dist[nr][nc] = dist[r][c] + 1;
            from[nr][nc] = u;
            q.push(nr * C + nc);
        }
    }
    int steps = dist[t / C][t % C];
    for (int u = from[t / C][t % C]; u != -1 && u != s; u = from[u / C][u % C])
        grid[u / C][u % C] = '*';          // mark the route
    return steps;
}

// Exercise 13.12: number of different paths from s to every vertex of a DAG,
// processed in a topological order (Kahn's).
std::vector<long long> countPaths(const Graph &g, int s) {
    int n = static_cast<int>(g.size());
    std::vector<int> indeg(n, 0);
    for (int u = 0; u < n; ++u)
        for (int v : g[u]) ++indeg[v];
    std::queue<int> q;
    for (int u = 0; u < n; ++u)
        if (indeg[u] == 0) q.push(u);
    std::vector<long long> ways(n, 0);
    ways[s] = 1;
```

Listing 13.22 (Continued) *exercises.cpp: solutions to Exercises 13.10 to 13.12.*

```
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : g[u]) {
            ways[v] += ways[u];            // every path to u extends to v
            if (--indeg[v] == 0) q.push(v);
        }
    }
    return ways;
}

int main() {
    Graph campus = undirected(10, {{0, 1}, {0, 2}, {0, 3}, {1, 4}, {1, 5}, {2, 3},
                                  {3, 6}, {4, 5}, {5, 6}, {6, 7}, {8, 9}});
    Graph square = undirected(4, {{0, 1}, {1, 2}, {2, 3}, {3, 0}});
    std::vector<int> color;
    std::printf("13.10 campus map bipartite? %s\n",
                bipartite(campus, color) ? "yes" : "no");
    bool sq = bipartite(square, color);
    std::printf("13.10 4-cycle bipartite? %s, colors", sq ? "yes" : "no");
    for (int c : color) std::printf(" %d", c);
    std::printf("\n");

    std::vector<std::string> room = {"S..#....",
                                     ".#.#.##.",
                                     ".#...#..",
                                     ".####.#.",
                                     "......#T"};
    int steps = gridBfs(room);
    std::printf("\n13.11 fewest steps from S to T: %d\n", steps);
    for (const std::string &row : room) std::printf("  %s\n", row.c_str());

    Graph work(9);
    const int arcs[][2] = {{0, 1}, {1, 2}, {1, 3}, {2, 4}, {3, 5},
                           {3, 7}, {4, 6}, {5, 6}, {6, 8}, {7, 8}};
    for (const auto &a : arcs) work[a[0]].push_back(a[1]);
    std::vector<long long> ways = countPaths(work, 0);
    std::printf("\n13.12 paths from order to shelve: %lld\n", ways[8]);
    std::printf("      paths from order to cover:  %lld\n", ways[6]);
    return 0;
}
```

```
13.10 campus map bipartite? no
13.10 4-cycle bipartite? yes, colors 0 1 0 1

13.11 fewest steps from S to T: 15
  S**#****
  .#*#*##*
  .#***#.*
  .####.#*
  ......#T

13.12 paths from order to shelve: 3
      paths from order to cover:  2
```

13.11 See `gridBfs` in **Listing 13.22**. Each free cell is a vertex with up to four neighbors, computed from its row and column rather than stored, and BFS gives 15 steps. The route marked with `*` in the output goes along the top row, down through the gap in column 2, along row 2 to column 4, up to the top row again, and down the right-hand column to T. The two shelves in rows 3 and 4 force the detour; there is no shorter way.

13.12 See `countPaths` in **Listing 13.22**. In topological order, every path to v ends with an arrow (u, v) from a vertex u processed earlier, so `ways[v]` is the sum of `ways[u]` over the arrows into v.

There are 3 paths from *order* to *shelve* (via inspect and stamp, via catalog and label, and via catalog and register) and 2 to *cover*.

The Java and Python versions of the exercise program, **Listing 13.23** and **Listing 13.24**, follow; they print the same output.

Listing 13.23 `Exercises.java`: solutions to Exercises 13.10 to 13.12 in Java.

```
// Exercises.java -- solutions to the programming exercises of Chapter 13:
// two-coloring, a grid maze by BFS, and counting paths in a DAG.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Exercises {
    static List<List<Integer>> undirected(int n, int[][] es) {
        List<List<Integer>> g = new ArrayList<>();
        for (int i = 0; i < n; i++) g.add(new ArrayList<>());
        for (int[] e : es) {
            g.get(e[0]).add(e[1]);
            g.get(e[1]).add(e[0]);
        }
        return g;
    }

    // Two-coloring by BFS: each new vertex gets the color opposite its parent.
    static boolean bipartite(List<List<Integer>> g, int[] color) {
        Arrays.fill(color, -1);
        for (int s = 0; s < g.size(); s++) {
            if (color[s] != -1) continue;
            ArrayDeque<Integer> q = new ArrayDeque<>();
            color[s] = 0;
            q.addLast(s);
            while (!q.isEmpty()) {
                int u = q.pollFirst();
                for (int v : g.get(u)) {
                    if (color[v] == -1) {
                        color[v] = 1 - color[u];
                        q.addLast(v);
                    } else if (color[v] == color[u]) {
                        return false;                // odd cycle
                    }
                }
            }
        }
        return true;
    }
```

Listing 13.23 (Continued) *Exercises.java: solutions to Exercises 13.10 to 13.12 in Java.*

```
    // Fewest steps from S to T in a grid; '#' is a bookshelf.
    static int gridBfs(char[][] grid) {
        int R = grid.length, C = grid[0].length;
        int[][] dist = new int[R][C], from = new int[R][C];
        for (int[] row : dist) Arrays.fill(row, -1);
        for (int[] row : from) Arrays.fill(row, -1);
        ArrayDeque<Integer> q = new ArrayDeque<>();  // cell (r, c) is r*C + c
        int s = -1, t = -1;
        for (int r = 0; r < R; r++)
            for (int c = 0; c < C; c++) {
                if (grid[r][c] == 'S') s = r * C + c;
                if (grid[r][c] == 'T') t = r * C + c;
            }
        dist[s / C][s % C] = 0;
        q.addLast(s);
        int[] dr = {-1, 1, 0, 0}, dc = {0, 0, -1, 1};
        while (!q.isEmpty()) {
            int u = q.pollFirst();
            int r = u / C, c = u % C;
            for (int k = 0; k < 4; k++) {
                int nr = r + dr[k], nc = c + dc[k];
                if (nr < 0 || nr >= R || nc < 0 || nc >= C) continue;
                if (grid[nr][nc] == '#' || dist[nr][nc] != -1) continue;
                dist[nr][nc] = dist[r][c] + 1;
                from[nr][nc] = u;
                q.addLast(nr * C + nc);
            }
        }
        int steps = dist[t / C][t % C];
        for (int u = from[t / C][t % C]; u != -1 && u != s; u = from[u / C][u % C])
            grid[u / C][u % C] = '*';                // mark the route
        return steps;
    }

    // Number of different paths from s to every vertex of a DAG (Kahn order).
    static long[] countPaths(List<List<Integer>> g, int s) {
        int n = g.size();
        int[] indeg = new int[n];
        for (int u = 0; u < n; u++)
            for (int v : g.get(u)) indeg[v]++;
        ArrayDeque<Integer> q = new ArrayDeque<>();
        for (int u = 0; u < n; u++)
            if (indeg[u] == 0) q.addLast(u);
        long[] ways = new long[n];
        ways[s] = 1;
        while (!q.isEmpty()) {
```

Listing 13.23 (Continued) *Exercises.java: solutions to Exercises 13.10 to 13.12 in Java.*

```
            int u = q.pollFirst();
            for (int v : g.get(u)) {
                ways[v] += ways[u];                  // paths to u extend to v
                if (--indeg[v] == 0) q.addLast(v);
            }
        }
        return ways;
    }

    public static void main(String[] args) {
        List<List<Integer>> campus = undirected(10, new int[][] {
            {0, 1}, {0, 2}, {0, 3}, {1, 4}, {1, 5}, {2, 3},
            {3, 6}, {4, 5}, {5, 6}, {6, 7}, {8, 9}});
        List<List<Integer>> square = undirected(4, new int[][] {
            {0, 1}, {1, 2}, {2, 3}, {3, 0}});
        int[] color = new int[10];
        System.out.printf("13.10 campus map bipartite? %s%n",
                bipartite(campus, color) ? "yes" : "no");
        color = new int[4];
        boolean sq = bipartite(square, color);
        System.out.printf("13.10 4-cycle bipartite? %s, colors", sq ? "yes" : "no");
        for (int c : color) System.out.print(" " + c);
        System.out.println();

        String[] rows = {"S..#....",
                         ".#.#.##.",
                         ".#...#..",
                         ".####.#.",
                         "......#T"};
        char[][] room = new char[rows.length][];
        for (int i = 0; i < rows.length; i++) room[i] = rows[i].toCharArray();
        int steps = gridBfs(room);
        System.out.printf("%n13.11 fewest steps from S to T: %d%n", steps);
        for (char[] row : room) System.out.println("  " + new String(row));

        List<List<Integer>> work = new ArrayList<>();
        for (int i = 0; i < 9; i++) work.add(new ArrayList<>());
        int[][] arcs = {{0, 1}, {1, 2}, {1, 3}, {2, 4}, {3, 5},
                        {3, 7}, {4, 6}, {5, 6}, {6, 8}, {7, 8}};
        for (int[] a : arcs) work.get(a[0]).add(a[1]);
        long[] ways = countPaths(work, 0);
        System.out.printf("%n13.12 paths from order to shelve: %d%n", ways[8]);
        System.out.printf("      paths from order to cover:  %d%n", ways[6]);
    }
}
```

Listing 13.24 `exercises.py`: solutions to Exercises 13.10 to 13.12 in Python.

```
# exercises.py -- solutions to the programming exercises of Chapter 13:
# two-coloring, a grid maze by BFS, and counting paths in a DAG.
from collections import deque


def undirected(n, es):
    g = [[] for _ in range(n)]
    for u, v in es:
        g[u].append(v)
        g[v].append(u)
    return g


def bipartite(g):
    """Two-coloring by BFS; returns (ok, color)."""
    color = [-1] * len(g)
    for s in range(len(g)):
        if color[s] != -1:
            continue
        color[s] = 0
        q = deque([s])
        while q:
            u = q.popleft()
            for v in g[u]:
                if color[v] == -1:
                    color[v] = 1 - color[u]
                    q.append(v)
                elif color[v] == color[u]:
                    return False, color      # an edge inside one color
    return True, color


def grid_bfs(grid):
    """Fewest steps from S to T; '#' is a bookshelf. Marks the route with '*'."""
    R, C = len(grid), len(grid[0])
    dist = [[-1] * C for _ in range(R)]
    came = [[None] * C for _ in range(R)]
    s = next((r, c) for r in range(R) for c in range(C) if grid[r][c] == "S")
    t = next((r, c) for r in range(R) for c in range(C) if grid[r][c] == "T")
    dist[s[0]][s[1]] = 0
    q = deque([s])
    while q:
        r, c = q.popleft()
        for nr, nc in ((r - 1, c), (r + 1, c), (r, c - 1), (r, c + 1)):
            if not (0 <= nr < R and 0 <= nc < C):
                continue
```

Listing 13.24 (Continued) *exercises.py: solutions to Exercises 13.10 to 13.12 in Python.*

```
            if grid[nr][nc] == "#" or dist[nr][nc] != -1:
                continue
            dist[nr][nc] = dist[r][c] + 1
            came[nr][nc] = (r, c)
            q.append((nr, nc))
    cell = came[t[0]][t[1]]
    while cell is not None and cell != s:
        grid[cell[0]][cell[1]] = "*"          # mark the route
        cell = came[cell[0]][cell[1]]
    return dist[t[0]][t[1]]


def count_paths(g, s):
    """Number of different paths from s to every vertex of a DAG (Kahn order)."""
    n = len(g)
    indeg = [0] * n
    for u in range(n):
        for v in g[u]:
            indeg[v] += 1
    q = deque(u for u in range(n) if indeg[u] == 0)
    ways = [0] * n
    ways[s] = 1
    while q:
        u = q.popleft()
        for v in g[u]:
            ways[v] += ways[u]                # every path to u extends to v
            indeg[v] -= 1
            if indeg[v] == 0:
                q.append(v)
    return ways


def main():
    campus = undirected(10, [(0, 1), (0, 2), (0, 3), (1, 4), (1, 5), (2, 3),
                             (3, 6), (4, 5), (5, 6), (6, 7), (8, 9)])
    square = undirected(4, [(0, 1), (1, 2), (2, 3), (3, 0)])
    ok, _ = bipartite(campus)
    print(f"13.10 campus map bipartite? {'yes' if ok else 'no'}")
    ok, color = bipartite(square)
    print(f"13.10 4-cycle bipartite? {'yes' if ok else 'no'}, colors "
          + " ".join(str(c) for c in color))
```

Listing 13.24 (Continued) *exercises.py: solutions to Exercises 13.10 to 13.12 in Python.*

```
    room = [list(row) for row in ["S..#....",
                                  ".#.#.##.",
                                  ".#...#..",
                                  ".####.#.",
                                  "......#T"]]
    steps = grid_bfs(room)
    print(f"\n13.11 fewest steps from S to T: {steps}")
    for row in room:
        print("  " + "".join(row))

    work = [[] for _ in range(9)]
    for u, v in [(0, 1), (1, 2), (1, 3), (2, 4), (3, 5),
                 (3, 7), (4, 6), (5, 6), (6, 8), (7, 8)]:
        work[u].append(v)
    ways = count_paths(work, 0)
    print(f"\n13.12 paths from order to shelve: {ways[8]}")
    print(f"      paths from order to cover:  {ways[6]}")


if __name__ == "__main__":
    main()
```

13.13 Extractions (distance, vertex): (0, PRK), relax AUD 70; (70, AUD), relax ENG 170, MED 230; (170, ENG), relax LIB 320, ADM 230; (230, ADM), relax LIB 320 → 310; (230, MED), relax CAF 410, DRM 320; (310, LIB), CAF would be 430, no; (320, LIB) stale, skipped; (320, DRM), CAF would be 410, not better; (410, CAF). Final distances: PRK 0, AUD 70, ENG 170, ADM 230, MED 230, LIB 310, DRM 320, CAF 410; ANX and WHS unreachable. Route to CAF: PRK, AUD, MED, CAF (70 + 160 + 180 = 410). Yes, one stale entry, the old (320, LIB). This trace was produced by running **Listing 13.18** from PRK.

13.14 Initially $d[s] = 0$, $d[a] = d[b] = \infty$. Extract s: relax to $d[a] = 2$, $d[b] = 3$. Extract a (2): final. Extract b (3): final; the edge (b, a) would give $3 - 2 = 1 < 2$, but a is done and is not updated. Result $d[a] = 2$, true distance 1. Adding a constant fails because it penalizes routes with more edges: in the example add 2 to every weight, so $w(s, a) = 4$, $w(s, b) = 5$, $w(b, a) = 0$; now the shortest route to a is the direct edge (4 against 5), whereas in the original graph it was s, b, a. The shifted graph has a different shortest route, so solving it does not solve the original.

13.15 (a) Adjacency matrix: the graph is dense (E near $V^2/2$) and small ($200^2 = 40000$ cells), and the frequent question is an $O(1)$ edge query. (b) Adjacency list: sparse, and route planning is traversal. (c) Edge list: the algorithm processes edges, not neighborhoods, and sorting a list of records is exactly Chapter 3's problem. (d) Adjacency list, with the lists of high-degree accounts kept sorted or hashed so that "does u follow v" is fast; a matrix over billions of accounts is impossible (Section 13.11.2).

13.16 (a) All four land masses have odd degree (A 3, B 5, C 3, D 3). A walk needs zero or two odd vertices, and one new bridge changes the parity of exactly two land masses. So one new bridge, joining any two of the land masses (for example A and D), leaves exactly two odd vertices, and a walk using every bridge once then exists, starting at one odd land mass and ending at the other. (b) In the main component, LIB, CAF, ENG, MED, AUD and PRK have odd degree: six odd vertices, so no such walk exists; at least two new footpaths would be needed.

References

[1] L. Euler, "Solutio problematis ad geometriam situs pertinentis," *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, vol. 8, pp. 128–140, 1741 (presented 1736).

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 20–22.

[3] E. F. Moore, "The shortest path through a maze," in *Proc. Int. Symp. Theory of Switching*, Cambridge, MA, USA: Harvard Univ. Press, 1959, pp. 285–292.

[4] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, Sep. 1961.

[5] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[6] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973.

[7] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

[8] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[9] cppreference.com, "std::priority_queue." [Online]. Available: https://en.cppreference.com/w/cpp/container/priority_queue. [Accessed: 24-Sep-2026].

[10] Oracle, "Class PriorityQueue," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/PriorityQueue.html. [Accessed: 24-Sep-2026].

[11] Python Software Foundation, "heapq — Heap queue algorithm," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/heapq.html. [Accessed: 24-Sep-2026].

[12] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[13] OpenStreetMap Taginfo, "Database statistics," data until 23-Sep-2026. [Online]. Available: https://taginfo.openstreetmap.org/reports/database_statistics. [Accessed: 24-Sep-2026].

[14] C. Demetrescu, A. Goldberg, and D. Johnson, "9th DIMACS Implementation Challenge — Shortest Paths: USA road networks," 2006. [Online]. Available: https://www.diag.uniroma1.it/challenge9/download.shtml. [Accessed: 24-Sep-2026].

[15] Meta Platforms, Inc., "Meta reports second quarter 2026 results," press release, Jul. 29, 2026. [Online]. Available: https://investor.atmeta.com. [Accessed: 24-Sep-2026].

[16] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, "Route planning in transportation networks," in *Algorithm Engineering*, LNCS vol. 9220. Cham, Switzerland: Springer, 2016, pp. 19–80.

[17] S. Bhagat, M. Burke, C. Diuk, I. O. Filiz, and S. Edunov, "Three and a half degrees of separation," Facebook Research, Feb. 4, 2016. [Online]. Available: https://research.facebook.com/blog/2016/02/three-and-a-half-degrees-of-separation/

[18] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

[19] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1–7, pp. 107–117, 1998.

Chapter 14
Heaps and Priority Queues

By the end of this chapter you will be able to say what a priority queue is and why neither a sorted nor an unsorted array makes a good one; store a complete binary tree in a plain array and move between a node, its parent and its children with nothing but arithmetic, in both the 0-based and the 1-based numbering; state the max-heap and min-heap properties precisely and prove what they guarantee; carry out sift-up and sift-down by hand, drawing the tree and the array side by side at every step; implement `insert`, `getMax`, `extractMax`, `getMin`, `increaseKey`, `decreaseKey`, `buildMaxHeap` and `isMaxHeap`, and give the exact cost of each, including the reason why the minimum of a max-heap costs a scan of the leaves; prove that building a heap bottom-up takes only linear time, and confirm the proof by counting; write heap sort in full, in place, and explain why it is not stable; use the priority queues that C++, Java and Python already provide, including the max-heap functions that Python 3.14 added to `heapq`; and solve three classic problems with a heap: the k largest items of a stream, the merging of k sorted lists, and the shortest paths of Dijkstra's algorithm. Along the way we build one complete `MaxHeap` class, in C++, Java and Python, on which Chapter 16 builds further.

14.1 Where We Left Off

Several earlier chapters have already knocked on the door of this one. Section 3.10 sorted an array with *heap sort*: it read the array as a tree, arranged it so that every node was at least as large as its children, and then repeatedly moved the largest element to the end. That section showed just enough of the heap to sort, and promised that this chapter would build the heap properly, prove its costs and use it for more than sorting. Section 5.15 met the *priority queue*, an emergency room where the most urgent patient is served first rather than the one who came first, and concluded that the obvious array implementations each make one operation slow. Section 5.16 then used the ready-made priority queues of the three standard libraries without saying how they work. And Chapter 11 introduced trees, the vocabulary of roots, parents, children, leaves, levels and height that we use throughout, and even stored a binary tree in an array without pointers (Section 11.4.1). Finally, Section 13.10 ran Dijkstra's shortest-path algorithm on a library priority queue and left open how that queue works.

This chapter keeps all three promises. The structure at its center, the *binary heap*, is one of the most useful ideas in the whole book, and one of the most economical. It needs no pointers at all: the tree lives inside an ordinary array, and moving from a node to its parent or to its children is a multiplication or a division by two. With that single array it gives us the largest element in constant time and lets us insert a new element or remove the largest one in time proportional to the height of the tree, which is about $\lg n$, 20 steps for a million elements.

Our running example stays the Pustaka Ceria library. Readers place *hold requests* on books, and not every request is equally urgent: a lecturer who needs a book for tomorrow's class, a student with a thesis deadline this week and a reader who simply wants to browse next month all wait in the same list. The circulation desk must always serve the most urgent request next, while new requests keep arriving. That is a priority queue, and by Section 14.12 we will have built one.

14.2 Serving the Most Urgent First

14.2.1 The priority queue as an abstract data type

A *priority queue* holds a collection of items, each with a *key* (its priority), and supports the following operations. We state them for a *max-priority queue*, in which a larger key means more urgent; a *min-priority queue* is the mirror image, in which a smaller key means more urgent, such as an earlier deadline or a shorter distance.

> **DEFINITION.** A *max-priority queue* is a collection of keys with the operations `insert(x)` (add the key x), `getMax()` (return the largest key without removing it), `extractMax()` (remove and return the largest key), `increaseKey(i, k)` (raise the key of an item already in the queue to k) and `isEmpty()`. A *min-priority queue* has `getMin`, `extractMin` and `decreaseKey` instead.

Table 14.1 lists the operations of the `MaxHeap` class that this chapter builds, with the cost each one will turn out to have. The costs are the point of the whole chapter, so it is worth seeing them before we earn them.

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

Despite its name, a priority queue is not a queue in the first-in, first-out sense of Chapter 5. Arrival order matters only to break ties between equal keys, and only if we ask for it (Section 14.14 shows how).

14.2.2 Why the obvious structures are not enough

Suppose the circulation desk keeps its hold requests in an array. There are two natural ways to do it, and a third that Chapter 11 would suggest.

- **An unsorted array.** A new request is appended at the end: $O(1)$. But finding the most urgent request means scanning all n of them, and removing it means scanning and then filling the hole: $\Theta(n)$ every time.
- **A sorted array.** Keep the requests sorted by urgency, the most urgent at the end. Then `getMax` and `extractMax` are $O(1)$: the answer is the last element. But `insert` must find the new request's place and shift everything after it one slot to the right, $\Theta(n)$ moves in the worst case, as Section 1.6.3 measured for insertion into an array.

- **A binary search tree.** A balanced search tree does both in $O(\lg n)$, but it needs two pointers per node, keeps a full sorted order that the priority queue never asks for, and a plain (unbalanced) search tree can degenerate into a chain of length n, as Chapter 11 showed.

Table 14.2 puts the options side by side.

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 two array versions fail in opposite ways. The unsorted array does too little work when an item arrives, so it must do a lot when the maximum is wanted; the sorted array does too much work on arrival, keeping an order far stronger than anyone needs. The heap sits in between: it keeps *just enough* order to find the maximum at once, and that small amount of order is cheap to restore after every change. Keeping just enough order is the whole idea of this chapter.

> **IN PLAIN LANGUAGE.** A priority queue never needs the whole collection sorted. It only needs to know, at every moment, which item comes first. A heap keeps exactly that much order and no more, and that is why it is fast.

14.2.3 Where the heap came from

The binary heap was published in June 1964 by J. W. J. Williams, as the data structure behind a new sorting algorithm he called *heapsort* [1]. Williams's short paper already contained the array layout of this chapter and the procedures for inserting an element and removing the largest one. Six months later Robert W. Floyd published an improved version, *Treesort 3*, whose main contribution was to build the heap from an arbitrary array bottom-up, in linear time [2], the method of Section 14.10. Knuth's *The Art of Computer Programming* gave the analysis that most later texts follow [3], and Cormen, Leiserson, Rivest and Stein give the modern textbook treatment [4]. Sixty years later, the standard libraries of C++, Java and Python still implement their priority queues as binary heaps stored in arrays, essentially as Williams described them (Section 14.14).

14.3 A Complete Binary Tree in an Array

14.3.1 Complete binary trees

Chapter 11 defined a binary tree as a tree in which every node has at most two children, a left one and a right one. A heap uses binary trees of one special shape.

> **DEFINITION.** A binary tree of height h is *complete* when every level $0, 1, \ldots, h-1$ is completely filled, and the nodes of the last level h are packed as far to the left as possible, with no gaps.

Figure 14.1 shows one complete tree and two that are not. Tree (a) has ten nodes: levels 0, 1 and 2 are full (1, 2 and 4 nodes), and the three nodes of level 3 sit at its left end. Tree (b) fails because its last level has a gap: the second position is empty while a later one is filled. Tree (c) fails because level 2, which is not the last level, is missing a node.

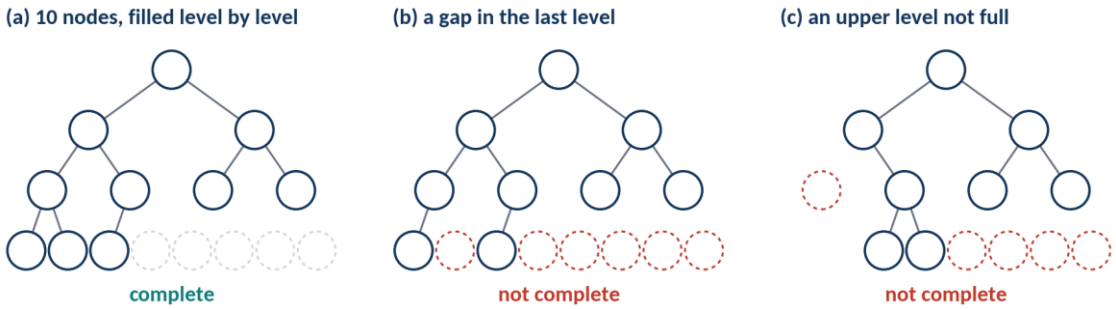


Figure 14.1 (a) A complete binary tree of ten nodes; dashed circles mark the free places of the last level, which fills from the left. (b) and (c) are not complete: (b) has a gap in its last level, and (c) has a missing node above the last level.

Section 11.5.2 met this shape among the full, perfect, complete and skewed trees, and warned that the names are not universal: Cormen et al. [4], for instance, call it *nearly complete* and reserve *complete* for the tree that Section 11.5.2 calls perfect. The shape is what matters: every level full except possibly the last, and the last filled from the left.

The shape has a remarkable consequence. A complete tree with n nodes is determined entirely by the number n: there is exactly one complete tree shape with ten nodes, the one in **Figure 14.1**(a). So if we number the nodes in a fixed order, the number of a node tells us exactly where it sits, and we never need to store the shape at all.

14.3.2 Numbering the nodes level by level

Number the nodes of a complete tree *level by level, from left to right within each level*, starting at the root. This is the order in which a level-order traversal (Chapter 11) visits them, and it is the order in which the tree grows: the next node to be added always receives the next number. Store the node numbered i in the array element with index i. **Figure 14.2** shows the ten-node tree of **Figure 14.1**(a) with keys A to J, its 0-based numbers in teal and its 1-based numbers in amber, and the array that holds it.

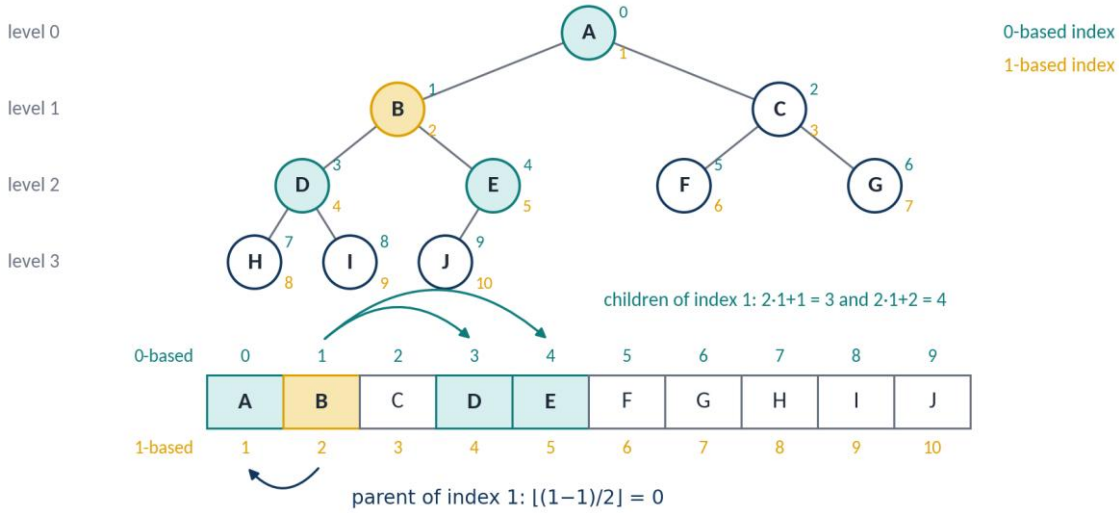


Figure 14.2 A complete tree stored level by level in an array. Each node carries its 0-based index (teal) and its 1-based index (amber); the arcs show that the children of index 1 are at indices 3 and 4, and its parent at index 0.

With the root at index 0, the relatives of the node at index i are

$$\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \qquad \text{left}(i) = 2i + 1, \qquad \text{right}(i) = 2i + 2. \tag{14.1}$$

The symbol $\lfloor x \rfloor$, the *floor* of x, is the largest integer not greater than x; in C++, Java and Python the integer division `(i - 1) / 2` computes it for every $i \geq 1$. These are the formulas of Section 11.4.1, which Section 3.10 used for heap sort. Check them on **Figure 14.2**: node B has index 1; its children D and E have indices $2 \cdot 1 + 1 = 3$ and $2 \cdot 1 + 2 = 4$; its parent A has index $\lfloor (1-1)/2 \rfloor = 0$. Node E has index 4; its left child J has index $2 \cdot 4 + 1 = 9$, and its right child would have index 10, which is outside an array of ten elements, so E has only one child.

Many books, including Cormen et al. [4] and Williams's original paper [1], number from 1 instead, leaving array slot 0 unused. With the root at index 1, the relatives of the node at index k are

$$\text{parent}(k) = \left\lfloor \frac{k}{2} \right\rfloor, \qquad \text{left}(k) = 2k, \qquad \text{right}(k) = 2k + 1. \tag{14.2}$$

The two numberings describe the same tree: the node with 0-based index i has 1-based index $k = i + 1$. Substituting $k = i + 1$ into Equation (14.2) gives back Equation (14.1): the left child is $2k = 2i + 2$ in 1-based numbering, which is $2i + 1$ in 0-based numbering, and so on. The 1-based formulas are a little prettier, since a shift left by one bit gives the left child and a shift right gives the parent, but they waste one slot and clash with the 0-based arrays of C++, Java and Python. Every program in this chapter uses the 0-based formulas of Equation (14.1); **Table 14.3** collects both.

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	$2i + 1 < n$	$2k \leq n$
is a leaf	$i \geq \lfloor n/2 \rfloor$	$k > \lfloor n/2 \rfloor$

14.3.3 Why the formulas are right

A formula that is merely checked on one example is a guess. Section 11.4.1 proved the 0-based formulas by counting the slots above each level. The 1-based version has an even shorter proof, and it shows why the formulas are so tidy; Equation (14.1) then follows again by the substitution above.

Level d of a complete tree holds up to 2^d nodes, so the full levels $0, 1, \ldots, d-1$ above it hold $1 + 2 + \cdots + 2^{d-1} = 2^d - 1$ nodes. In the 1-based numbering the first node of level d therefore gets the number 2^d, and the nodes of level d are numbered $2^d, 2^d + 1, \ldots, 2^{d+1} - 1$. Take the node k that is the j-th node of its level, counting from $j = 0$, so that $k = 2^d + j$. The j nodes to its left on level d each have two children on level $d + 1$, and those $2j$ children come first on level $d + 1$, in the same left-to-right order. So the left child of k is the node with position $2j$ on level $d + 1$, whose number is

$$2^{d+1} + 2j = 2(2^d + j) = 2k,$$

and the right child is the next one, $2k + 1$. Conversely, a node $m \geq 2$ is the left child of $m/2$ if m is even and the right child of $(m - 1)/2$ if m is odd; in both cases its parent is $\lfloor m/2 \rfloor$. The argument uses only the fact that every level above the last is full and that the last is filled from the left, which is exactly the definition of a complete tree. For a tree with gaps the formulas fail, which is why a heap must always stay complete.

14.3.4 Height, and where the leaves are

Two more facts about complete trees are used in every cost analysis of this chapter. The first was already met in Section 11.5.3, which showed that no binary tree of n nodes is lower than $\lfloor \lg n \rfloor$ and that a complete tree reaches this bound; we derive it again directly for complete trees.

The height. A complete tree of height h has all of the levels 0 to $h - 1$ full, $2^h - 1$ nodes, plus between 1 and 2^h nodes on level h. So its number of nodes satisfies

$$2^h \leq n \leq 2^{h+1} - 1. \tag{14.3}$$

Taking the base-2 logarithm, lg, of the left inequality gives $h \leq \lg n$, and of the right one gives $\lg(n + 1) \leq h + 1$, so $h > \lg n - 1$. The only integer in that range is

$$h = \lfloor \lg n \rfloor. \tag{14.4}$$

A heap of ten keys has height $\lfloor \lg 10 \rfloor = 3$, a heap of a thousand keys height 9, and a heap of a million keys height 19. Every operation of this chapter that walks along one path between the root and a leaf therefore makes at most about 20 steps on a million keys.

The leaves. In the 0-based numbering, the node i has a child exactly when its left child exists, that is, when $2i + 1 \leq n - 1$, or $i \leq n/2 - 1$. Since i is an integer this means $i \leq \lfloor n/2 \rfloor - 1$. So

$$\text{the internal nodes are } 0, 1, \ldots, \lfloor n/2 \rfloor - 1, \quad \text{and the leaves are } \lfloor n/2 \rfloor, \ldots, n - 1. \tag{14.5}$$

In a complete tree, then, the leaves always occupy the second half of the array, and there are $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ of them: at least half of all the nodes are leaves. For $n = 10$, the internal nodes are 0 to 4 and the leaves are 5 to 9, as **Figure 14.2** shows. Section 14.8 uses this to find the minimum of a max-heap, and Section 14.10 uses it to know where to start building a heap.

Listing 14.1 prints the whole index table for $n = 10$ in both numberings, together with the height and the range of leaves.

Listing 14.1 `heap_index.cpp`: parent and children of every node of a ten-node complete tree, in the 0-based and 1-based numberings.

```cpp
// heap_index.cpp -- the index arithmetic of a complete binary tree in an array.
#include <cstdio>

// 0-based: the root is at index 0.
int parent0(int i) { return (i - 1) / 2; }
int left0(int i) { return 2 * i + 1; }
int right0(int i) { return 2 * i + 2; }

// 1-based: the root is at index 1 (slot 0 is left unused).
int parent1(int k) { return k / 2; }
int left1(int k) { return 2 * k; }
int right1(int k) { return 2 * k + 1; }

void cell(int v, int n, int lo) {           // prints v, or "-" if outside
    if (v >= lo && v < n + lo) std::printf("%7d", v);
    else std::printf("%7s", "-");
}

int main() {
    const int n = 10;
    std::printf("n = %d nodes\n", n);
    std::printf("%-12s%7s%7s%7s | %-11s%7s%7s%7s\n", "0-based", "parent",
                "left", "right", "1-based", "parent", "left", "right");
    for (int i = 0; i < n; ++i) {
        int k = i + 1;                      // the same node, numbered from 1
        std::printf("i=%2d        ", i);
        if (i == 0) std::printf("%7s", "-"); else cell(parent0(i), n, 0);
        cell(left0(i), n, 0);
        cell(right0(i), n, 0);
        std::printf(" | k=%2d      ", k);
        if (k == 1) std::printf("%7s", "-"); else cell(parent1(k), n, 1);
        cell(left1(k), n, 1);
        cell(right1(k), n, 1);
        std::printf("\n");
    }
    int h = 0;
    while ((2 << h) <= n) ++h;              // largest h with 2^h <= n
    std::printf("height = floor(lg %d) = %d\n", n, h);
    std::printf("internal nodes: 0..%d, leaves: %d..%d (0-based)\n",
                n / 2 - 1, n / 2, n - 1);
    return 0;
}
```

```
n = 10 nodes
0-based      parent   left  right | 1-based     parent   left  right
i= 0              -      1      2 | k= 1             -      2      3
i= 1              0      3      4 | k= 2             1      4      5
i= 2              0      5      6 | k= 3             1      6      7
i= 3              1      7      8 | k= 4             2      8      9
i= 4              1      9      - | k= 5             2     10      -
i= 5              2      -      - | k= 6             3      -      -
i= 6              2      -      - | k= 7             3      -      -
i= 7              3      -      - | k= 8             4      -      -
i= 8              3      -      - | k= 9             4      -      -
i= 9              4      -      - | k=10            5      -      -
height = floor(lg 10) = 3
internal nodes: 0..4, leaves: 5..9 (0-based)
```

The two halves of every row describe the same node, and each entry in the right half is one more than the entry on its left, as the substitution $k = i + 1$ predicts. The height is computed without floating point: the loop finds the largest h with $2^h \le n$, which is $\lfloor \lg n \rfloor$ by Equation (14.3). The Java and Python versions, `HeapIndex.java` and `heap_index.py`, print the same table.

14.4 The Heap Property

14.4.1 Max-heaps and min-heaps

A complete tree in an array is only the container. What makes it a heap is an order between each node and its parent.

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:

$$A[\text{parent}(i)] \geq A[i] \qquad \text{for every } i = 1,2,\ldots,n-1. \tag{14.6}$$

It is a *min-heap* when every node other than the root is no smaller than its parent:

$$A[\text{parent}(i)] \leq A[i] \qquad \text{for every } i = 1,2,\ldots,n-1. \tag{14.7}$$

Figure 14.3 shows the same seven keys arranged as a max-heap and as a min-heap, each drawn as a tree next to the array that stores it. In the max-heap, 90 at the root is at least as large as its children 65 and 70; 65 is at least as large as 55 and 25; 70 is at least as large as 40 and 30. In the min-heap every arrow of comparison points the other way.

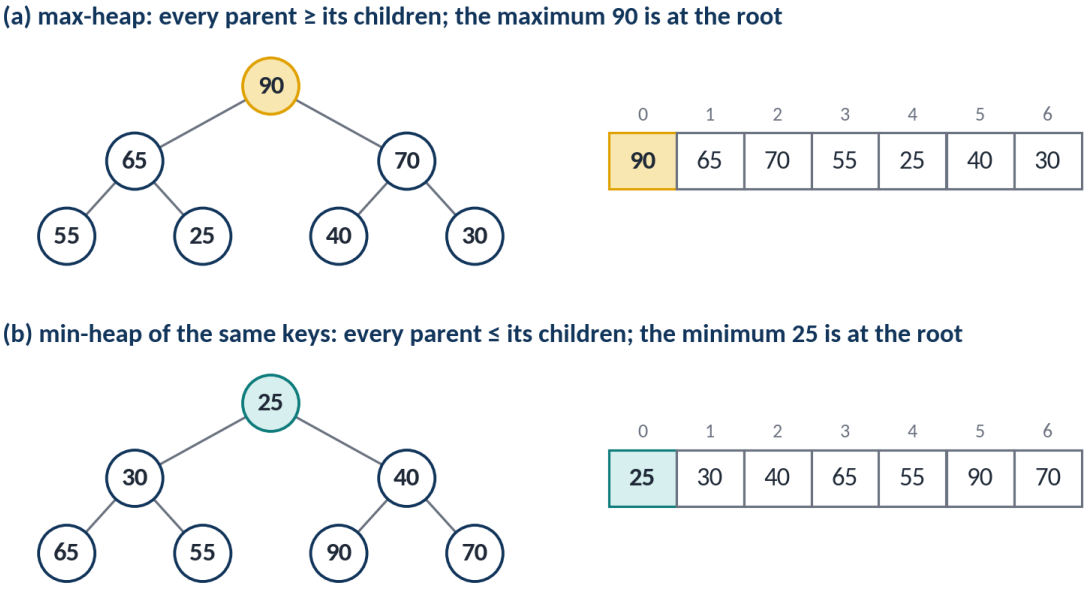


Figure 14.3 The keys 25, 30, 40, 55, 65, 70 and 90 arranged (a) as a max-heap, with the maximum at the root, and (b) as a min-heap, with the minimum at the root.

Everything we prove for max-heaps holds for min-heaps with every comparison reversed. This chapter works with max-heaps, because our `MaxHeap` class and heap sort use them; the library priority queues of Java and Python and the applications of Section 14.15 use min-heaps, and we will simply flip the comparisons there.

14.4.2 What the property guarantees

The heap property compares only a node with its parent, yet it has a global consequence.

Claim. In a non-empty max-heap, the root holds a largest key: $A[0] \geq A[i]$ for every i.

Proof. Take any node i and follow parents up to the root: i, parent(i), parent(parent(i)), and so on. Each step goes to a smaller index, so the walk reaches index 0 after finitely many steps. By

Equation (14.6) the keys along the walk never decrease: $A[i] \le A[\text{parent}(i)] \le \cdots \le A[0]$. So $A[0] \ge A[i]$. ▫

The same argument, started at any node, shows more: every node is at least as large as every node in its subtree. In other words, *every root-to-leaf path in a max-heap is sorted in non-increasing order*. In **Figure 14.3**(a) the path 90, 65, 55 is decreasing, and so is 90, 70, 30.

14.4.3 What the property does not say

It is just as important to see what a heap does *not* promise.

- **Siblings are not ordered.** Nothing relates a left child to its right sibling. In **Figure 14.3**(a) the left child of the root, 65, is smaller than the right child, 70; exchanging the two subtrees would give another valid max-heap.
- **Levels are not ordered.** A node deep in the tree can be larger than a node higher up in another branch. In **Figure 14.3**(a), 55 on level 2 is larger than 40 on the same level and smaller than 65 on the level above, while 40 is smaller than both of its level-1 neighbors 65 and 70; the heap property compares a node only with its own parent, never with the rest of a level.
- **The array is not sorted.** Read left to right, the max-heap of **Figure 14.3**(a) is 90, 65, 70, 55, 25, 40, 30: not in any order. A sorted array (in descending order) *is* a max-heap, but a max-heap is almost never sorted.
- **A heap is not a search tree.** In the binary search tree of Chapter 11 the left subtree holds smaller keys and the right subtree larger ones, so a key can be found by one walk from the root. In a heap a key can be in either subtree, so finding an arbitrary key needs a scan of up to all n elements.

This weaker order is the price of speed. A search tree must maintain the full sorted order, and a sorted array even more so; a heap maintains a partial order that is cheap to repair, and it is enough to answer the one question a priority queue asks.

> **WARNING.** The word *heap* has a second, unrelated meaning. Section 1.3.5 used it for the region of memory from which `new` allocates. The data structure of this chapter has nothing to do with that region; the two meanings simply share a name.

14.5 Sift-Up: Repairing a Key That Is Too Large

Every operation that changes a heap breaks the heap property in at most one place and then repairs it. There are only two ways in which one key can be out of place: it can be *too large* for its position, larger than its parent, or *too small*, smaller than one of its children. The first is repaired by moving the key up, the second by moving it down. These two repair procedures, *sift-up* and *sift-down*, are the engine of everything that follows; `insert`, `extractMax`, `increaseKey`, `decreaseKey`, `buildMaxHeap` and heap sort are each a few lines around one of them. Some books call them *heapify-up* and *heapify-down*, *bubble-up* and *percolate-down*, or *swim* and *sink* [5]; the ideas are the same.

14.5.1 The procedure

Suppose the heap property holds everywhere except that the key at index i may be larger than its parent. Sift-up compares the key with its parent and, if the key is larger, exchanges the two; the key has now moved up one level, and the same question is asked again at its new position. It stops when the key reaches the root or meets a parent at least as large. **Pseudocode 14.1** gives the procedure.

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)
```

Figure 14.4 follows sift-up after the key 80 has been placed at index 8 of the heap 90, 65, 70, 55, 25, 40, 30, 10. At every step the figure draws the tree and the array side by side: the key that moves is amber, and the key it is compared with is teal. The parent of index 8 is index $\lfloor 7/2 \rfloor = 3$, holding 55; since $80 > 55$ the two are exchanged. At index 3 the parent is index 1, holding 65; $80 > 65$, so they are exchanged again. At index 1 the parent is the root, 90, and $80 < 90$, so the loop stops. Two exchanges, three comparisons, and the array is a heap again.

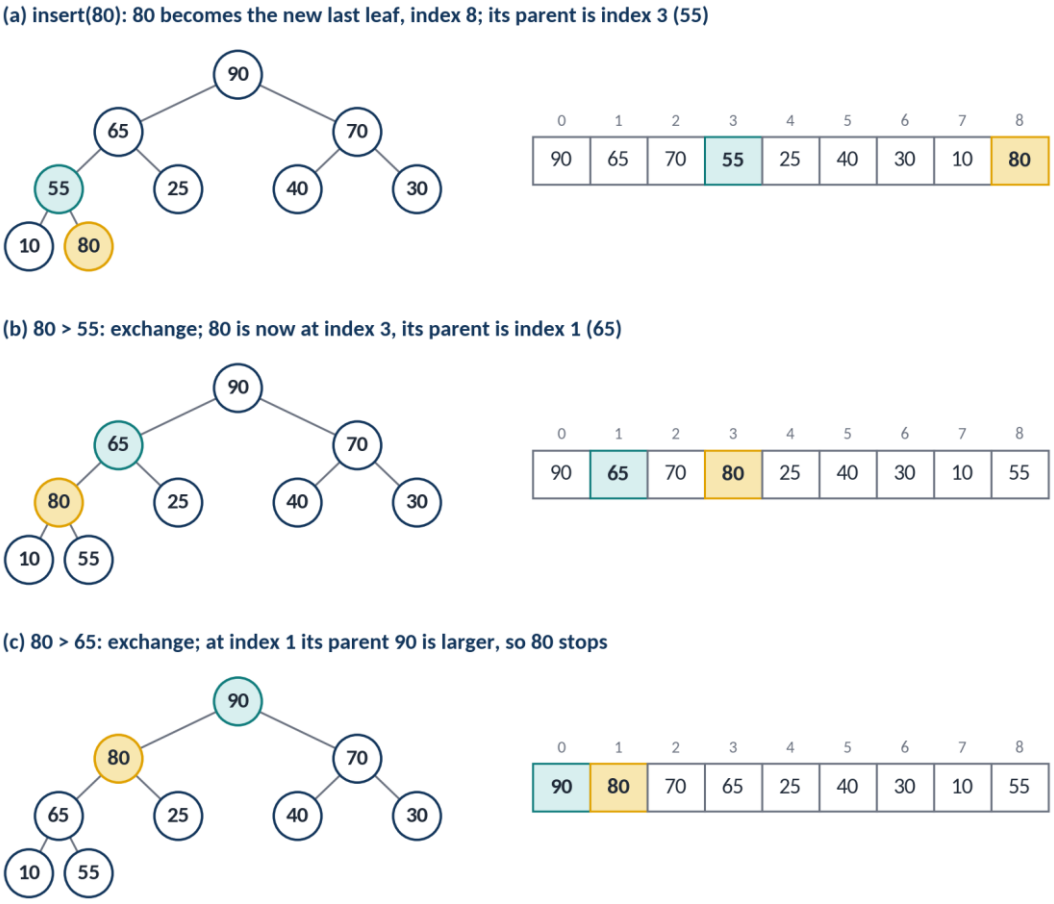


Figure 14.4 Sift-up of the key 80 from index 8. Each panel shows the tree and its array side by side; the moving key is amber and the parent it is compared with is teal.

14.5.2 Why it is correct

Sift-up is correct because of a slightly stronger fact that stays true at every step, a *loop invariant* in the sense of Section 2.5.3, proved in the same three parts.

Invariant. At the test of line 1, the heap property $A[\text{parent}(j)] \geq A[j]$ holds for every $j \neq i$; and if i has a parent, that parent is at least as large as the children of i.

Initially, the only possible violation is at i by assumption. When sift-up is called after appending a new leaf, i has no children at all; when it is called after raising the key at i (Section 14.9), the children of i were no larger than the old key at i, which was no larger than its parent.

Maintenance. Let $p = \text{parent}(i)$, let $x = A[i]$ and $y = A[p]$, with $y < x$ so that line 2 runs. After the exchange, index p holds x and index i holds y. At index i: its children are no larger than y by the second half of the invariant, so the property holds below i. At index p: its other child s (the sibling of i) satisfied $s \le y$ before, and $y < x$, so $s \le x$; and $y < x$ for the child i. So the only pair that may still be wrong is p with its own parent g. And g is at least as large as the children of p: $g \ge y$ by the heap property before the exchange, $y \ge s$, and y is the new $A[i]$. Setting $i = p$ restores the invariant.

Termination. The loop ends when $i = 0$ or $A[\text{parent}(i)] \ge A[i]$. In either case the only possible violation is gone, so the whole array is a max-heap.

14.5.3 What it costs

Each iteration moves i one level up, from index i to $\lfloor (i-1)/2 \rfloor$, so the loop runs at most as many times as the depth of the starting node, which is at most the height $\lfloor \lg n \rfloor$ by Equation (14.4). Each iteration makes one comparison and one exchange, and one more comparison ends the loop. So sift-up makes at most $\lfloor \lg n \rfloor$ exchanges and $\lfloor \lg n \rfloor + 1$ comparisons: $O(\lg n)$ time. It may also stop at once, when the key is already small enough: then it costs a single comparison.

A small refinement, used by several library implementations, avoids most of the writes. Instead of exchanging at every level, the moving key is saved in a variable, each smaller parent is copied one level *down* into the hole, and the key is written once into the last hole. This does about half the memory writes of an exchange. Our class keeps plain exchanges, which are easier to draw and to check; Exercise 14.14 asks you to write the refined version.

14.6 Sift-Down: Repairing a Key That Is Too Small

14.6.1 The procedure

Now suppose the key at index i may be *smaller* than one of its children, while the two subtrees below i are max-heaps. Sift-down compares the key with its children. If neither child is larger, we are done. Otherwise the key is exchanged with the **larger** of the two children, and the question is asked again one level lower. **Pseudocode 14.2** gives the procedure for a heap occupying $A[0..n-1]$; the size n is a parameter because heap sort (Section 14.13) runs sift-down on a heap that is only the front part of the array.

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
```

Why the *larger* child? After the exchange, the child that moved up becomes the parent of the other child. If we had moved up the smaller child, it would now sit above a larger sibling, a new violation. Moving up the larger child makes it at least as large as both the key that went down and its sibling, so the property holds at index i for good.

Figure 14.5 shows sift-down at work inside `extractMax`, which Section 14.7 explains: the root 90 has been removed, and the last key, 55, has been moved to the root of the heap 80, 70, 65, 25, 40, 30, 10. The children of the root are 80 and 70; the larger is 80, and $55 < 80$, so they are exchanged. At index 1 the children are 65 (index 3) and 25 (index 4); the larger is 65 and $55 < 65$, so they are exchanged. At index 3 the only child is 10 at index 7 (index 8 is no longer inside the heap), and $55 > 10$, so sift-down stops.

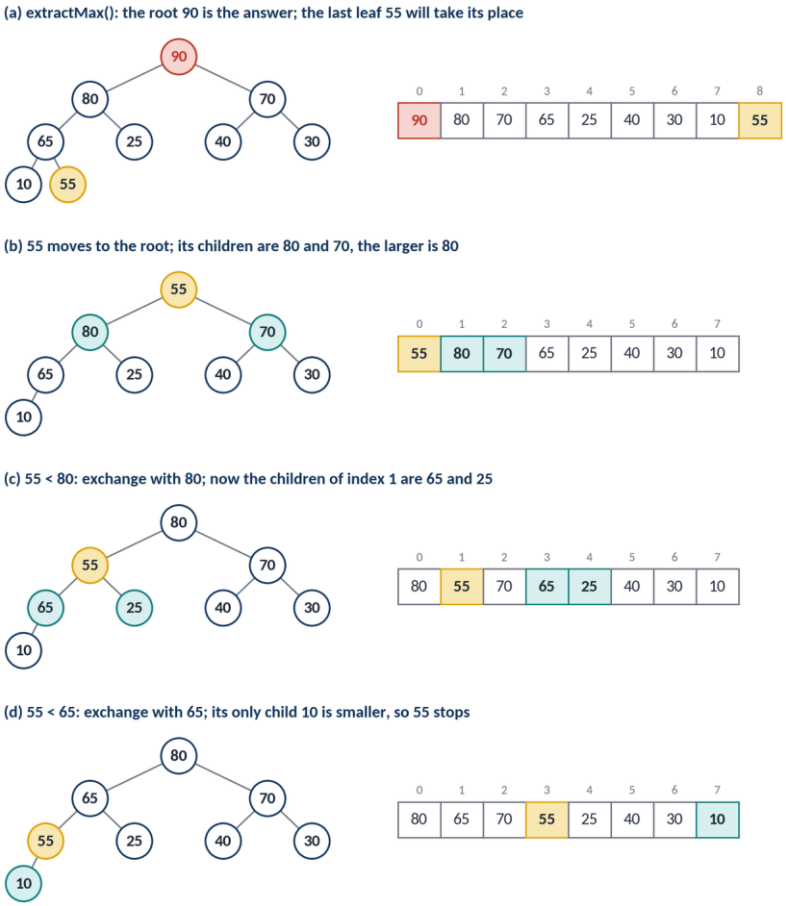


Figure 14.5 Sift-down of the key 55 from the root during `extractMax`: at each level it is compared with both children (teal) and exchanged with the larger one, until no child is larger.

14.6.2 Why it is correct, and what it costs

The argument mirrors that of sift-up. At the start of each iteration, the subtrees rooted at the children of i are max-heaps, and every node above i is at least as large as everything in the subtree of i. If $A[i]$ is at least as large as both children, the subtree of i is a heap and we stop. Otherwise, let c be the larger child. After the exchange, index i holds the old $A[c]$, which is at least as large as its sibling, as everything in the sibling's subtree (a heap below its root), and as the old $A[i]$ now at c; the sibling's subtree is untouched and still a heap; and the subtree at c is a heap except possibly at c itself, which is exactly the situation we started from, one level lower. The loop must stop, at the latest, when i becomes a leaf, because then lines 4 and 5 find no child.

Each iteration moves one level down, so there are at most $\lfloor \lg n \rfloor$ exchanges. Each iteration makes up to two comparisons, one per child, so

$$\text{comparisons of SIFT-DOWN} \le 2\lfloor \lg n \rfloor, \qquad \text{exchanges} \le \lfloor \lg n \rfloor. \tag{14.8}$$

That is again $O(\lg n)$, but with twice the comparisons of sift-up, because going down there are two candidates at each level and going up there is only one parent. More precisely, a sift-down that starts at a node of height h (the length of the longest path from it down to a leaf) makes at most h exchanges and $2h$ comparisons. This finer count is what makes the linear-time build of Section 14.10 possible.

14.7 Insert, GetMax and ExtractMax

With the two repair procedures in hand, the three central operations of a priority queue take a few lines each.

14.7.1 `getMax`

By the claim of Section 14.4.2, the root holds a largest key. So `getMax` returns $A[0]$: one array access, $O(1)$ time, whatever n is. It is an error to ask an empty heap for its maximum, and our class reports it with an exception rather than returning a made-up value.

14.7.2 `insert`

A new key must go somewhere, and a complete tree of n nodes can grow in exactly one place: the next free position of the last level, which is array index n. So `insert` appends the key at the end of the array, where the shape is automatically still complete, and then repairs the order with sift-up, since the new key may be larger than its parent. **Pseudocode 14.3** gives both steps.

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)
```

Its cost is that of sift-up, $O(\lg n)$, plus the $O(1)$ append (amortized $O(1)$ when the array grows by doubling, Section 1.6.5). **Figure 14.4** is exactly `insert(80)`. The whole sequence of Pustaka Ceria hold requests of Section 14.12 builds a heap by nine insertions of the urgency scores 40, 25, 70, 10, 55, 90, 30, 65 and 80; here is the heap after each one, as the program of Section 14.12 prints it:

```
  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
  insert(30)  -> 90 55 70 10 25 40 30
  insert(65)  -> 90 65 70 55 25 40 30 10
  insert(80)  -> 90 80 70 65 25 40 30 10 55
```

Work through a few lines by hand. `insert(70)` puts 70 at index 2, whose parent is index 0 holding 40; $70 > 40$, so they are exchanged, giving 70, 25, 40. `insert(55)` puts 55 at index 4, parent index 1 holding 25, exchanged; at index 1 its parent 70 is larger, so it stops: 70, 55, 40, 10, 25. `insert(90)` puts 90 at index 5, parent index 2 holding 40, exchanged; then parent index 0 holding 70, exchanged; 90 has reached the root. `insert(30)` puts 30 at index 6 under 70 and nothing moves. Most insertions move the new key only a level or two, or not at all; only a key larger than everything travels the whole height.

14.7.3 `extractMax`

Removing the maximum is the mirror image. The maximum is the root, but we cannot simply delete the root: that would leave two separate subtrees and a hole at the top. The only node whose removal keeps the tree complete is the *last* leaf, at index $n-1$. So `extractMax` (**Pseudocode 14.4**) saves the root's key to return it, moves the last key into the root, shrinks the heap by one, and repairs the root, where the moved key is almost certainly too small, with sift-down.

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
```

Figure 14.5 is `extractMax` on the heap of nine hold requests: it returns 90 and leaves 80, 65, 70, 55, 25, 40, 30, 10. A second `extractMax` returns 80: the last key 10 moves to the root, is exchanged with 70 (the larger of 65 and 70), then with 40 (the larger of 40 and 30 at indices 5 and 6), and the heap becomes 70, 65, 40, 55, 25, 10, 30. The cost is one sift-down from the root: at most $2\lfloor \lg n \rfloor$ comparisons, $O(\lg n)$ time.

Why move the *last* key to the root and not, say, the larger child of the root? Because moving the last key is what keeps the shape complete. Promoting children one by one would fill the hole at the root but move the hole down some path to a leaf that is generally not the last one, leaving a gap in the tree. Keeping the shape and the order separate, first the shape by moving the last leaf, then the order by sifting, is the method behind every heap operation.

> **IN PLAIN LANGUAGE.** To add a key, put it at the end and let it climb while it beats its parent. To remove the top, put the last key on top and let it sink, always trading places with its bigger child. Both walks follow one path of the tree, so both take about $\lg n$ steps.

14.8 GetMin in a Max-Heap

A max-heap is organized around its maximum. What if the circulation desk also wants the *least* urgent request, for example to decide which request to drop when the list is full? The heap property says nothing directly about the minimum, but it does say where the minimum cannot hide.

Claim. In a non-empty max-heap, some leaf holds a smallest key. If all keys are distinct, the smallest key is *only* at a leaf.

Proof. Let m be the smallest key, stored at some index j. If j is internal, it has a child c, and by the heap property $A[c] \le A[j] = m$; since m is the smallest, $A[c] = m$. Repeat from c: each step goes down one level, so after at most h steps we reach a leaf holding m. If the keys are distinct, the equality $A[c] = m$ is impossible, so j had no child: it was a leaf. ▫

By Equation (14.5) the leaves are the indices $\lfloor n/2 \rfloor$ to $n-1$, the second half of the array. So `getMin` needs to look only at them, as **Pseudocode 14.5** does.

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
```

Figure 14.6 highlights the leaves of the nine-key heap: indices $\lfloor 9/2 \rfloor = 4$ to 8, holding 25, 40, 30, 10 and 55. The scan finds 10 after four comparisons, one fewer than the number of leaves.

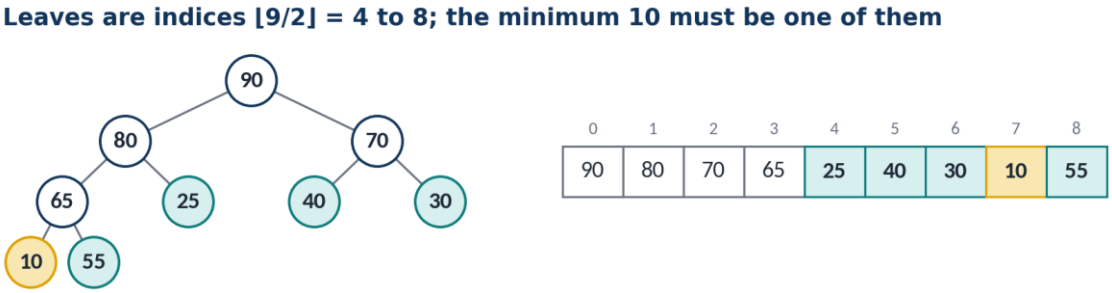


Figure 14.6 The leaves of a max-heap of nine keys are the indices 4 to 8, the second half of the array (teal); the minimum, 10 (amber), is one of them.

Scanning half of the array is still linear: $\lceil n/2 \rceil - 1$ comparisons, which is $\Theta(n)$. Could a cleverer method do better, say $O(\lg n)$, as `getMax` does in $O(1)$? No, and the reason is instructive. **Every leaf can be the minimum.** Take any max-heap and any leaf ℓ, and lower the key at ℓ to a value smaller than every other key. The array is still a max-heap, because lowering a leaf can only make it smaller than its parent, which the property allows, and a leaf has no children to violate. So an algorithm that never looks at leaf ℓ cannot tell this heap from the original one, and it would return the wrong minimum for one of them. Every correct algorithm must therefore examine all $\lceil n/2 \rceil$ leaves, and `getMin` in a max-heap needs $\Omega(n)$ time. Our scan is optimal.

This is a structural limit, not a weakness of the code. A max-heap answers "largest" in $O(1)$ and "smallest" in $\Theta(n)$. When a program needs both quickly, it uses another structure: two heaps kept side by side (one max-heap and one min-heap of the same items), a single *min-max heap*, which alternates max levels and min levels and gives both extremes in $O(1)$ and both extractions in $O(\lg n)$ [6], or a balanced search tree. If only the minimum over the whole history is wanted and items are never removed, a single variable updated at every `insert` is enough.

14.9 IncreaseKey and DecreaseKey

Priorities change. A reader's hold request becomes more urgent when a deadline moves closer; a request becomes less urgent when the reader says the book can wait. A priority queue that supports such changes needs two more operations. In a max-heap, raising a key can only make it too large for its position, and lowering it can only make it too small. So each change is one assignment followed by one of our two repairs, as **Pseudocode 14.6** shows.

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)
```

The guards on lines 1 and 2 matter. An index outside $0..n-1$ is simply a bug in the caller. A "new value" below the current key in `increaseKey` is more treacherous: the code would still run, but sift-up would never move the smaller key down, and the heap would be silently broken below index i. Rejecting such a call is safer than guessing what the caller meant. (A general `changeKey` could compare the new value with the old one and call sift-up or sift-down accordingly; Exercise 14.10 needs exactly that logic.)

Figure 14.7 traces both operations on the heap 70, 65, 40, 55, 25, 10, 30, which is what remains of the hold requests after two calls of `extractMax`. `increaseKey(6, 75)` raises the key 30 at index 6 to 75. Its parent, index 2, holds 40, so they are exchanged; at index 2 its parent is the root, 70, so they are exchanged again, and 75 is the new maximum: 75, 65, 70, 55, 25, 10, 40. Then `decreaseKey(0, 20)` lowers the root to 20. Its children are 65 and 70; the larger is 70, and they are exchanged; at index 2 the children are 10 and 40, and 20 is exchanged with 40; at index 6 it is a leaf. The result is 70, 65, 40, 55, 25, 10, 20.

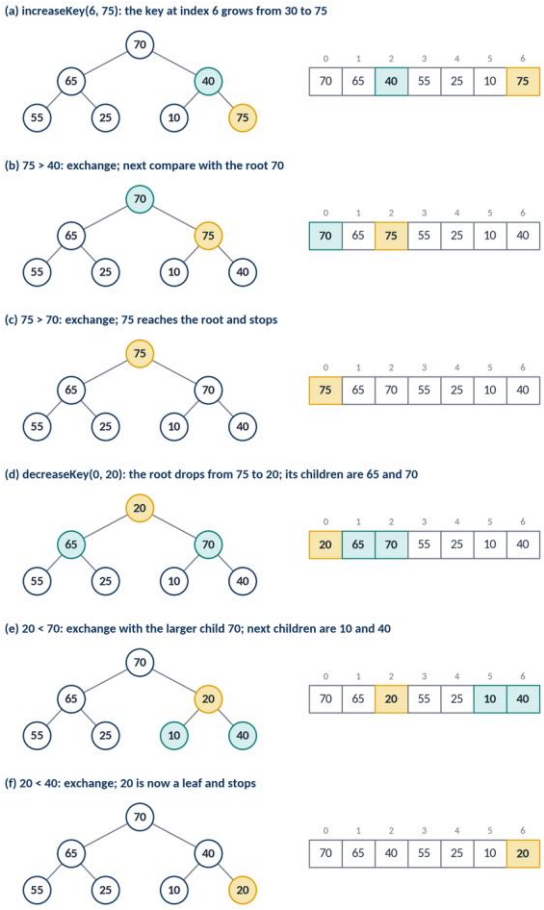


Figure 14.7 (a) to (c) `increaseKey(6, 75)`: the raised key rises by sift-up. (d) to (f) `decreaseKey(0, 20)`: the lowered key sinks by sift-down, always toward the larger child.

Both operations cost one sift, $O(\lg n)$. They have one practical catch: the caller must know the *index* of the item whose key changes. The index of an item in a heap changes whenever sift-up or sift-down moves it, so a program that changes priorities of named items (a reader, a vertex of a graph) must keep a second table from each item to its current index, updated at every exchange. Such an *indexed priority queue* is exactly what Dijkstra's algorithm with `decreaseKey` needs; Section 14.15.3 shows a simpler alternative that the standard libraries make possible.

14.10 Building a Heap Bottom-Up in Linear Time

14.10.1 Two ways to build

Often all the keys are known at once: the circulation desk opens in the morning with the whole list of yesterday's requests, or heap sort receives an array to sort. One way to make a heap of them is to start from an empty heap and call `insert` n times. That costs $O(\lg n)$ per insertion and $O(n \lg n)$ in total, and we will see that this bound can really be reached.

Floyd's method [2] is better. It takes the array *as it is*, already a complete tree by Equation (14.1), and repairs the order from the bottom up. The leaves, the second half of the array by Equation (14.5), are one-node heaps already: a node without children cannot violate anything. So **Pseudocode 14.7** visits the internal nodes from the last one, index $\lfloor n/2 \rfloor - 1$, back to the root, and run sift-down on each. When we reach a node, both of its subtrees have already been made into heaps, which is exactly the precondition of sift-down.

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)
```

Correctness follows from an invariant: *at the start of the iteration for index i, every node* $i+1, i+2, \ldots, n-1$ *is the root of a max-heap.* Initially $i = \lfloor n/2 \rfloor - 1$ and the nodes after it are leaves, one-node heaps. The children of i have larger indices, so they are roots of heaps, and sift-down makes i the root of a heap without disturbing any node outside the subtree of i. When the loop ends, $i = -1$, so node 0, the root, is the root of a max-heap: the whole array is one.

Figure 14.8 traces `buildMaxHeap` on the array 12, 30, 45, 7, 60, 18, 52, 25, 3, 41 of the program in Section 14.12. With $n = 10$ the internal nodes are 0 to 4. Node 4 holds 60, larger than its only child 41: nothing moves. Node 3 holds 7, smaller than its children 25 and 3: 7 sinks below 25. Node 2 holds 45, and its larger child is 52: they change places. Node 1 holds 30; its larger child is 60 at index 4; after the exchange, 30 is at index 4 with the child 41 at index 9, and sinks once more. Finally the root 12 sinks three levels, past 60, 41 and 30, and the heap 60, 41, 52, 25, 30, 18, 45, 7, 3, 12 is complete. Seven exchanges in all.

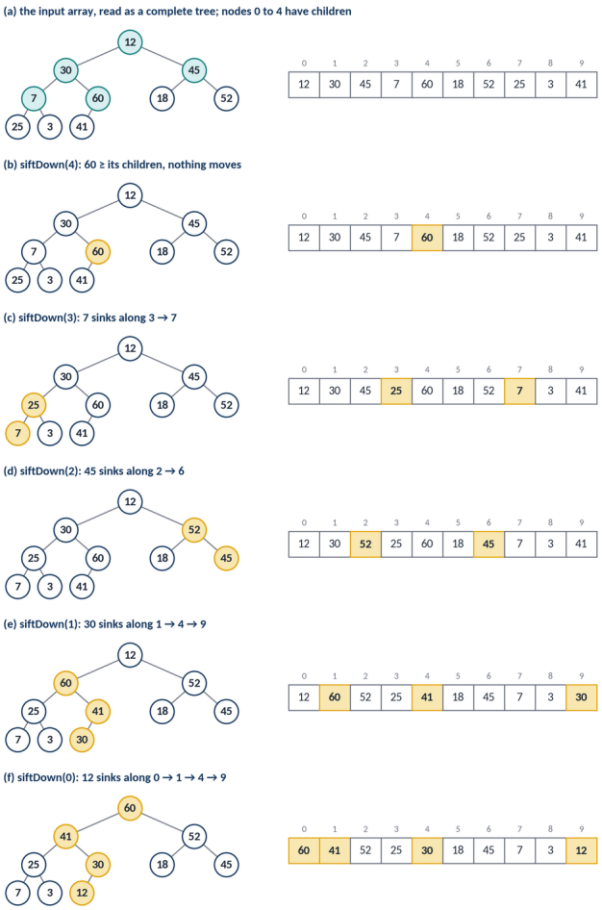


Figure 14.8 `buildMaxHeap` on 12, 30, 45, 7, 60, 18, 52, 25, 3, 41: sift-down is applied to the internal nodes 4, 3, 2, 1 and 0 in that order; amber marks the path along which each key sinks.

14.10.2 Why it takes only linear time

A quick estimate says: $n/2$ calls of sift-down, each $O(\lg n)$, so $O(n \lg n)$. That bound is true but far from tight. The estimate charges every sift-down the height of the whole tree, while most nodes are near the bottom and can sink only a little. Half of all nodes are leaves and are never visited; a quarter are one level above the leaves and can sink at most one level; an eighth can sink at most two; only the root can sink $\lfloor \lg n \rfloor$ levels. **Figure 14.9** shows the picture for a tree with 15 nodes, each labeled with its *height*, the largest number of exchanges a sift-down starting there can make.

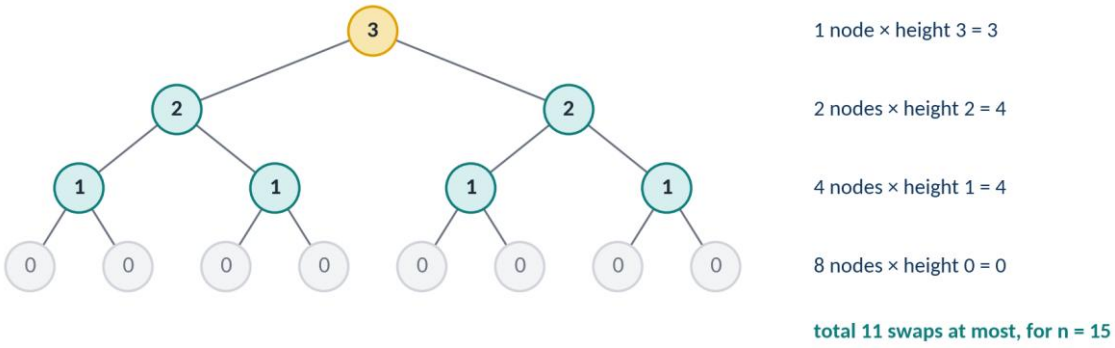


Figure 14.9 Each node of a 15-node heap labeled with its height, the most levels a sift-down from it can travel. Summing level by level gives at most 11 exchanges for the whole build, fewer than the 15 nodes.

Let us make this exact. Take first a *perfect* heap of height H, one whose last level is also full, so that $n = 2^{H+1} - 1$. Level d holds 2^d nodes, each of height $H - d$. The total number of exchanges is at most

$$S = \sum_{d=0}^{H} 2^d\,(H-d) = \sum_{h=0}^{H} h\;2^{H-h} = 2^H \sum_{h=0}^{H} \frac{h}{2^h}, \tag{14.9}$$

where the middle form substitutes $h = H - d$. For $H = 3$ this is $8 \cdot 0 + 4 \cdot 1 + 2 \cdot 2 + 1 \cdot 3 = 11$, the total of **Figure 14.9**.

Now we need the sum of $h/2^h$. Write $T = \sum_{h\ge 1} h\,/2^h = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots$ and halve it: $T/2 = \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \cdots$. Subtracting term by term,

$$T - \frac{T}{2} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = 1, \qquad \text{so} \qquad \sum_{h=0}^{\infty} \frac{h}{2^h} = T = 2. \tag{14.10}$$

The finite sum in Equation (14.9) is smaller than the infinite one, so $S < 2^H \cdot 2 = 2^{H+1} = n + 1$. (Working the finite sum exactly gives $S = 2^{H+1} - H - 2 = n - \lg(n+1)$; for $n = 15$ that is $15 - 4 = 11$.) For a complete heap that is not perfect, the same count applies level by level: in any heap of n nodes there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h [4], and the same series gives a total of $O(n)$. Each exchange is accompanied by at most two comparisons, so the numbers of exchanges E and comparisons C made by `BUILD-MAX-HEAP`, and its running time T, satisfy

$$E_{\text{build}}(n) < n, \qquad C_{\text{build}}(n) < 2n, \qquad T_{\text{build}}(n) = \Theta(n), \tag{14.11}$$

It is $\Theta(n)$ and not only $O(n)$ because the loop visits $\lfloor n/2 \rfloor$ nodes and makes at least one comparison at each.

> **IN PLAIN LANGUAGE.** Building bottom-up is cheap because the work is done where there is little to do. The many nodes near the bottom can each sink only a step or two; the few nodes near the top that could sink far are, well, few. The sum of "number of nodes times distance" stays below n.

14.10.3 Building by insertion is really slower

Why not simply insert the keys one by one? Because insertion puts the work in the wrong place: most keys are inserted at the bottom, where the path to the root is *longest*. If the keys arrive in ascending order, every new key is the largest so far and climbs all the way to the root. The key inserted at index i (for $i \ge 1$) sits at depth $\lfloor \lg(i+1) \rfloor$, so the total number of comparisons is

$$\sum_{m=2}^{n} \lfloor \lg m \rfloor = (n+1)\lfloor \lg n \rfloor - 2^{\lfloor \lg n \rfloor + 1} + 2 = \Theta(n \lg n). \tag{14.12}$$

(Exercise 14.13 asks you to prove the closed form.) For $n = 1000$ it gives $1001 \cdot 9 - 1024 + 2 = 7987$.

Listing 14.2 checks both analyses by counting. It builds heaps of 1,000, 10,000 and 100,000 keys, from an ascending array, a pseudo-random one and a descending one, once by n insertions and once bottom-up, and prints the number of key comparisons and that number divided by n. The pseudo-random keys come from a simple linear congruential generator, $x \leftarrow (1103515245\,x + 12345) \bmod 2^{31}$, written out in the program so that the C++, Java and Python versions (`BuildCount.java`, `build_count.py`) produce exactly the same keys and exactly the same counts.

Listing 14.2 `build_count.cpp`: key comparisons made by n insertions and by one bottom-up build, on ascending, pseudo-random and descending keys.

```
// build_count.cpp -- comparisons made by n inserts versus one bottom-up build.
#include <cstdio>
#include <utility>
#include <vector>
using namespace std;

long long cmps = 0;

void siftUp(vector<long long>& a, int i) {
    while (i > 0) {
        int p = (i - 1) / 2;
        ++cmps;
        if (!(a[p] < a[i])) return;
        swap(a[p], a[i]);
        i = p;
    }
}

void siftDown(vector<long long>& a, int i, int n) {
    while (true) {
        int largest = i, l = 2 * i + 1, r = 2 * i + 2;
        if (l < n && (++cmps, a[l] > a[largest])) largest = l;
        if (r < n && (++cmps, a[r] > a[largest])) largest = r;
        if (largest == i) return;
        swap(a[i], a[largest]);
        i = largest;
    }
}

long long byInserts(const vector<long long>& in) {
    vector<long long> a;
    cmps = 0;
    for (long long x : in) {
        a.push_back(x);
        siftUp(a, static_cast<int>(a.size()) - 1);
    }
    return cmps;
}

long long byBuild(vector<long long> a) {
    cmps = 0;
    int n = static_cast<int>(a.size());
    for (int i = n / 2 - 1; i >= 0; --i) siftDown(a, i, n);
    return cmps;
}
```

Listing 14.2 (Continued) *build_count.cpp: key comparisons made by n insertions and by one bottom-up build, on ascending, pseudo-random and descending keys.*

```
int main() {
    printf("%-11s%8s%13s%9s%12s%9s\n", "input", "n", "n inserts", "per n",
           "build", "per n");
    for (int n : {1000, 10000, 100000}) {
        vector<long long> up(n), down(n), rnd(n);
        unsigned long long x = 2026;                  // a fixed pseudo-random seed
        for (int i = 0; i < n; ++i) {
            up[i] = i;
            down[i] = n - i;
            x = (x * 1103515245ULL + 12345ULL) % 2147483648ULL;
            rnd[i] = static_cast<long long>(x % 1000000);
        }
        pair<const char*, vector<long long>*> inputs[] = {
            {"ascending", &up}, {"random", &rnd}, {"descending", &down}};
        for (auto& [name, v] : inputs) {
            long long ins = byInserts(*v), bld = byBuild(*v);
            printf("%-11s%8d%13lld%9.2f%12lld%9.2f\n", name, n, ins,
                   static_cast<double>(ins) / n, bld, static_cast<double>(bld) / n);
        }
    }
    return 0;
}
```

```
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
```

Read the table column by column.

- **Bottom-up build.** Comparisons per key stay at or below 2.00 for every input and every size, exactly as Equation (14.11) promises. The ascending array is the worst case (every internal key is smaller than its children and sinks as far as it can) and reaches 1.98 to 2.00; the descending array is already a heap and costs exactly $n - 1$ comparisons, one per child.
- **Insertion, ascending keys.** Comparisons per key grow with n: 7.99, 11.36, 14.69. The totals, 7,987, 113,631 and 1,468,946, are exactly the values of Equation (14.12). At $n = 100{,}000$ building by insertion makes more than seven times as many comparisons as building bottom-up.
- **Insertion, random keys.** About 2.28 comparisons per key at every size. A random new key usually belongs near the bottom, where most of the nodes are, so on average it climbs only a constant number of levels, a fact proved by Porter and Simon [7]. On random data n insertions are linear on average; the $\Theta(n \lg n)$ of Equation (14.12) is the worst case.

The lesson for practice: when all keys are available at once, use `buildMaxHeap`; when they arrive one at a time, `insert` is the only choice, and it is usually cheap anyway.

14.11 Checking the Heap Property: `isMaxHeap`

A heap that is silently broken gives wrong answers without crashing, which is the worst kind of bug. A function that *checks* the heap property is therefore worth having, both as an operation in

its own right (is this array, received from elsewhere, a max-heap?) and as a testing tool that a program can call after every operation while it is being developed.

Equation (14.6) is already an algorithm: compare every node except the root with its parent.

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
```

The loop makes at most $n - 1$ comparisons, $O(n)$ time, and stops at the first violation. Run on the array of **Figure 14.8**(a), 12, 30, 45, 7, 60, 18, 52, 25, 3, 41, it compares index 1 (30) with its parent index 0 (12), finds $12 < 30$, and returns `false` at once. Run on the heap that `buildMaxHeap` produced, it checks all nine parent-child pairs and returns `true`. Two more examples: in 10, 7, 8, 5, 2, 3, 1 every node is no larger than its parent, so it is a heap; in 10, 7, 6, 5, 1, 8, 2 the key 8 at index 5 has the parent 6 at index 2, so it is not, even though the root is larger than everything. No algorithm can do better than linear time in the worst case: when the answer is `true`, every one of the $n - 1$ parent-child pairs must have been compared, because a single unchecked pair could be the violation.

Looping over children, as **Pseudocode 14.8** does, has a quiet advantage over the equally natural loop over *parents*, `for i = 0 to ⌊n/2⌋ - 1`, which checks the left child and, if it exists, the right child. The child loop has no special cases: the right child that may be missing, the empty array and the one-element array all take care of themselves, because the loop simply does not run. The parent loop needs every one of them handled, and in C++ it hides a trap. With `v.size()` of the unsigned type `size_t`, a bound written as `(v.size() - 2) / 2` does not become negative for an array of 0 or 1 elements: the subtraction wraps around to about 1.8×10^{19}, the loop runs, and it reads far outside the array. Section 1.5.4 showed the damage an index outside the array can do; the defense is to test the empty and one-element cases, and to prefer loop forms in which they cannot go wrong.

14.12 The Complete `MaxHeap` Class

14.12.1 The interface

We now put every operation together into one class, in each of the three languages. **Table 14.4** lists its public interface. The C++ names are the ones used throughout this chapter; Java uses the same names, and Python follows its own naming convention with underscores.

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()`	$O(n)$	—

A few design decisions deserve a word.

- **The storage** is a growable array: `std::vector<int>` in C++, which grows by itself; an `int[]` in Java that the class doubles when it is full, the technique of Section 1.6.5; and a Python `list`. The keys are plain `int` values, which keeps the algorithm in view; Section 14.14 shows how the library versions carry whole records.
- **Errors are exceptions.** Asking an empty heap for its maximum has no sensible answer, and returning a made-up value such as -1 would be indistinguishable from a real key -1. The C++ class throws `std::underflow_error` for an empty heap, `std::out_of_range` for a bad index and `std::invalid_argument` for a key change in the wrong direction; Java uses `NoSuchElementException`, `IndexOutOfBoundsException` and `IllegalArgumentException`; Python uses `IndexError` and `ValueError`. The messages are the same in all three.
- **`isMaxHeap` and `heapSort` are static**, since they work on any array given to them and not on the heap's own contents. In C++ a static member can still be called through an object, `h.isMaxHeap(v)`, as well as through the class, `MaxHeap::isMaxHeap(v)`. The member `isValid()` applies the check to the heap itself, which is how the demonstration confirms that every operation left a valid heap.
- **Sift-down takes the array and its size as parameters**, so that the same code serves the heap's own storage, `buildMaxHeap`, and `heapSort` working on a caller's array.

14.12.2 The C++ class

Listing 14.3 is the complete program: the class and a `main` that exercises every operation on the Pustaka Ceria hold requests.

Listing 14.3 `max_heap.cpp`: the complete `MaxHeap` class and a demonstration of every operation.

```cpp
// max_heap.cpp -- our MaxHeap class: a max-heap of int keys kept in a vector.
#include <algorithm>
#include <iostream>
#include <stdexcept>
#include <string>
#include <utility>
#include <vector>
using namespace std;

class MaxHeap {
public:
    MaxHeap() = default;

    int size() const { return static_cast<int>(heap.size()); }
    bool empty() const { return heap.empty(); }
    void clear() { heap.clear(); }
    const vector<int>& data() const { return heap; }

    // Prints the array that stores the heap, in index order.
    void printHeap() const {
        for (int i = 0; i < size(); ++i) cout << (i ? " " : "") << heap[i];
        cout << "\n";
    }

    // 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 14.3 (Continued) *max_heap.cpp: the complete MaxHeap class and a demonstration of every operation.*

```
    // Returns the smallest key; it is on a leaf, so scan the leaves: O(n).
    int getMin() const {
        requireNonEmpty("getMin");
        int n = size();
        int smallest = heap[n / 2];         // index n/2 is the first leaf
        for (int i = n / 2 + 1; i < n; ++i) smallest = min(smallest, heap[i]);
        return smallest;
    }

    // Raises the key at index to newValue, then lets it rise: O(lg n).
    void increaseKey(int index, int newValue) {
        checkIndex(index, "increaseKey");
        if (newValue < heap[index])
            throw invalid_argument("increaseKey: new value is below the key");
        heap[index] = newValue;
        siftUp(index);
    }

    // Lowers the key at index to newValue, then lets it sink: O(lg n).
    void decreaseKey(int index, int newValue) {
        checkIndex(index, "decreaseKey");
        if (newValue > heap[index])
            throw invalid_argument("decreaseKey: new value is above the key");
        heap[index] = newValue;
        siftDown(heap, index, size());
    }

    // Replaces the contents by arr and rearranges it bottom-up: O(n).
    void buildMaxHeap(const vector<int>& arr) {
        heap = arr;
        for (int i = size() / 2 - 1; i >= 0; --i) siftDown(heap, i, size());
    }

    // True when every element is no larger than its parent: O(n).
    static bool isMaxHeap(const vector<int>& arr) {
        int n = static_cast<int>(arr.size());
        for (int i = 1; i < n; ++i)
            if (arr[(i - 1) / 2] < arr[i]) return false;
        return true;
    }
    bool isValid() const { return isMaxHeap(heap); }
```

Listing 14.3 (Continued) *max_heap.cpp: the complete MaxHeap class and a demonstration of every operation.*

```
    // Sorts arr into ascending order in place with heap sort: O(n lg n).
    static void heapSort(vector<int>& arr) {
        int n = static_cast<int>(arr.size());
        for (int i = n / 2 - 1; i >= 0; --i) siftDown(arr, i, n);
        for (int end = n - 1; end > 0; --end) {
            swap(arr[0], arr[end]);         // the maximum goes to its final slot
            siftDown(arr, 0, end);          // repair the heap arr[0..end-1]
        }
    }

private:
    vector<int> heap;                       // heap[0] is the root

    static int parent(int i) { return (i - 1) / 2; }
    static int left(int i) { return 2 * i + 1; }
    static int right(int i) { return 2 * i + 2; }

    // 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 14.3 (Continued) *max_heap.cpp: the complete MaxHeap class and a demonstration of every operation.*

```
    void requireNonEmpty(const string& op) const {
        if (heap.empty()) throw underflow_error(op + ": the heap is empty");
    }
    void checkIndex(int index, const string& op) const {
        if (index < 0 || index >= size())
            throw out_of_range(op + ": index " + to_string(index)
                               + " is outside the heap");
    }
};

void show(const string& label, const vector<int>& a) {
    cout << label;
    for (size_t i = 0; i < a.size(); ++i) cout << (i ? " " : "") << a[i];
    cout << "\n";
}

int main() {
    MaxHeap h;
    cout << "Urgent hold requests arrive (urgency scores):\n";
    for (int key : {40, 25, 70, 10, 55, 90, 30, 65, 80}) {
        h.insert(key);
        cout << "  insert(" << key << ")  -> ";
        h.printHeap();
    }
    cout << "size = " << h.size() << ", getMax = " << h.getMax()
         << ", getMin = " << h.getMin() << ", valid = " << boolalpha
         << h.isValid() << "\n\n";

    cout << "extractMax = " << h.extractMax() << "  -> ";
    h.printHeap();
    cout << "extractMax = " << h.extractMax() << "  -> ";
    h.printHeap();

    h.increaseKey(6, 75);
    cout << "increaseKey(6, 75)  -> ";
    h.printHeap();
    h.decreaseKey(0, 20);
    cout << "decreaseKey(0, 20)  -> ";
    h.printHeap();
    cout << "valid = " << h.isValid() << "\n\n";
```

Listing 14.3 (Continued) *max_heap.cpp: the complete MaxHeap class and a demonstration of every operation.*

```
    vector<int> arr = {12, 30, 45, 7, 60, 18, 52, 25, 3, 41};
    show("array:                ", arr);
    cout << "isMaxHeap(array)     = " << MaxHeap::isMaxHeap(arr) << "\n";
    h.buildMaxHeap(arr);
    cout << "buildMaxHeap(array) -> ";
    h.printHeap();
    cout << "isMaxHeap(heap)      = " << h.isMaxHeap(h.data()) << "\n";
    MaxHeap::heapSort(arr);
    show("heapSort(array)     -> ", arr);
    cout << "\n";

    h.clear();
    cout << "after clear: size = " << h.size() << ", empty = " << h.empty() << "\n";
    try {
        h.extractMax();
    } catch (const exception& e) {
        cout << "error: " << e.what() << "\n";
    }
    h.insert(5);
    try {
        h.increaseKey(0, 3);
    } catch (const exception& e) {
        cout << "error: " << e.what() << "\n";
    }
    try {
        h.increaseKey(4, 99);
    } catch (const exception& e) {
        cout << "error: " << e.what() << "\n";
    }
    return 0;
}
```

The program prints:

```
Urgent hold requests arrive (urgency scores):
  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
  insert(30)  -> 90 55 70 10 25 40 30
  insert(65)  -> 90 65 70 55 25 40 30 10
  insert(80)  -> 90 80 70 65 25 40 30 10 55
size = 9, getMax = 90, getMin = 10, valid = true

extractMax = 90  -> 80 65 70 55 25 40 30 10
extractMax = 80  -> 70 65 40 55 25 10 30
increaseKey(6, 75)  -> 75 65 70 55 25 10 40
decreaseKey(0, 20)  -> 70 65 40 55 25 10 20
valid = true

array:                12 30 45 7 60 18 52 25 3 41
isMaxHeap(array)     = false
buildMaxHeap(array) -> 60 41 52 25 30 18 45 7 3 12
isMaxHeap(heap)      = true
heapSort(array)     -> 3 7 12 18 25 30 41 45 52 60

after clear: size = 0, empty = true
error: extractMax: the heap is empty
error: increaseKey: new value is below the key
error: increaseKey: index 4 is outside the heap
```

Every line can be checked against the earlier sections. The nine insertions are those of Section 14.7.2, ending with the heap of **Figure 14.4**(c). `getMax` is 90, the root, and `getMin` is 10, found among the leaves as in **Figure 14.6**. The two calls of `extractMax` are those of Section 14.7.3; the first is **Figure 14.5**. `increaseKey(6, 75)` and `decreaseKey(0, 20)` are the six panels of **Figure 14.7**. The array 12, 30, 45, 7, 60, 18, 52, 25, 3, 41 is not a heap, `buildMaxHeap` turns it into the heap of **Figure 14.8**(f), and `heapSort` sorts it into ascending order, as Section 14.13 explains. The last three lines show the three kinds of error: an empty heap, a key change in the wrong direction, and an index outside a one-element heap.

Two small points in the code. `getMin` starts its scan at `heap[n / 2]`, the first leaf by Equation (14.5), rather than at `heap[0]`: the root is never the minimum of a heap with two or more distinct keys, and starting at the first leaf saves one comparison. And `extractMax` calls sift-down only when the heap is not empty after the removal, because a heap that had one key has nothing left to repair.

We tested the class under AddressSanitizer and UndefinedBehaviorSanitizer (Section 7.11.3) and under Valgrind (Section 7.11.2): no invalid access, no leak. A `std::vector` manages its own memory, so the class needs no destructor of its own.

14.12.3 The same class in Java and Python

Listing 14.4 and **Listing 14.5** are the Java and Python versions. They print exactly the output above.

Listing 14.4 `MaxHeap.java`: the `MaxHeap` class in Java, with an array that doubles when it is full.

```java
// MaxHeap.java -- our MaxHeap class: a max-heap of int keys in a growable array.
import java.util.Arrays;
import java.util.NoSuchElementException;

public class MaxHeap {
    private int[] heap = new int[8];        // heap[0] is the root
    private int n = 0;                      // number of keys in use

    public int size() { return n; }
    public boolean empty() { return n == 0; }
    public void clear() { n = 0; }
    public int[] data() { return Arrays.copyOf(heap, n); }

    // Prints the array that stores the heap, in index order.
    public void printHeap() { System.out.println(join(heap, n)); }

    // Appends key as the last leaf, then lets it rise: O(lg n).
    public void insert(int key) {
        if (n == heap.length) heap = Arrays.copyOf(heap, 2 * n);
        heap[n++] = key;
        siftUp(n - 1);
    }

    // Returns the largest key without removing it: O(1).
    public int getMax() {
        requireNonEmpty("getMax");
        return heap[0];
    }

    // Removes and returns the largest key: O(lg n).
    public int extractMax() {
        requireNonEmpty("extractMax");
        int top = heap[0];
        heap[0] = heap[--n];                // the last leaf fills the hole
        if (n > 0) siftDown(heap, 0, n);
        return top;
    }

    // Returns the smallest key; it is on a leaf, so scan the leaves: O(n).
    public int getMin() {
        requireNonEmpty("getMin");
        int smallest = heap[n / 2];         // index n/2 is the first leaf
        for (int i = n / 2 + 1; i < n; i++) smallest = Math.min(smallest, heap[i]);
        return smallest;
    }
```

Listing 14.4 (Continued) *MaxHeap.java: the MaxHeap class in Java, with an array that doubles when it is full.*

```
    // Raises the key at index to newValue, then lets it rise: O(lg n).
    public void increaseKey(int index, int newValue) {
        checkIndex(index, "increaseKey");
        if (newValue < heap[index])
            throw new IllegalArgumentException(
                    "increaseKey: new value is below the key");
        heap[index] = newValue;
        siftUp(index);
    }

    // Lowers the key at index to newValue, then lets it sink: O(lg n).
    public void decreaseKey(int index, int newValue) {
        checkIndex(index, "decreaseKey");
        if (newValue > heap[index])
            throw new IllegalArgumentException(
                    "decreaseKey: new value is above the key");
        heap[index] = newValue;
        siftDown(heap, index, n);
    }

    // Replaces the contents by arr and rearranges it bottom-up: O(n).
    public void buildMaxHeap(int[] arr) {
        heap = Arrays.copyOf(arr, Math.max(8, arr.length));
        n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--) siftDown(heap, i, n);
    }

    // True when every element is no larger than its parent: O(n).
    public static boolean isMaxHeap(int[] arr) {
        for (int i = 1; i < arr.length; i++)
            if (arr[(i - 1) / 2] < arr[i]) return false;
        return true;
    }
    public boolean isValid() { return isMaxHeap(data()); }

    // Sorts arr into ascending order in place with heap sort: O(n lg n).
    public static void heapSort(int[] arr) {
        int len = arr.length;
        for (int i = len / 2 - 1; i >= 0; i--) siftDown(arr, i, len);
        for (int end = len - 1; end > 0; end--) {
            swap(arr, 0, end);              // the maximum goes to its final slot
            siftDown(arr, 0, end);          // repair the heap arr[0..end-1]
        }
    }
```

Listing 14.4 (Continued) *MaxHeap.java: the MaxHeap class in Java, with an array that doubles when it is full.*

```
    private static int parent(int i) { return (i - 1) / 2; }
    private static int left(int i) { return 2 * i + 1; }
    private static int right(int i) { return 2 * i + 2; }

    private static void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }

    // Lets heap[i] rise while it is larger than its parent.
    private void siftUp(int i) {
        while (i > 0 && heap[parent(i)] < heap[i]) {
            swap(heap, i, parent(i));
            i = parent(i);
        }
    }

    // Lets a[i] sink inside the heap a[0..len-1] until no child is larger.
    private static void siftDown(int[] a, int i, int len) {
        while (true) {
            int largest = i, l = left(i), r = right(i);
            if (l < len && a[l] > a[largest]) largest = l;
            if (r < len && a[r] > a[largest]) largest = r;
            if (largest == i) return;
            swap(a, i, largest);
            i = largest;
        }
    }

    private void requireNonEmpty(String op) {
        if (n == 0) throw new NoSuchElementException(op + ": the heap is empty");
    }
    private void checkIndex(int index, String op) {
        if (index < 0 || index >= n)
            throw new IndexOutOfBoundsException(op + ": index " + index
                                                + " is outside the heap");
    }

    static String join(int[] a, int len) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < len; i++) sb.append(i > 0 ? " " : "").append(a[i]);
        return sb.toString();
    }
```

Listing 14.4 (Continued) *MaxHeap.java: the MaxHeap class in Java, with an array that doubles when it is full.*

```
    public static void main(String[] args) {
        MaxHeap h = new MaxHeap();
        System.out.println("Urgent hold requests arrive (urgency scores):");
        for (int key : new int[] {40, 25, 70, 10, 55, 90, 30, 65, 80}) {
            h.insert(key);
            System.out.print("  insert(" + key + ")  -> ");
            h.printHeap();
        }
        System.out.println("size = " + h.size() + ", getMax = " + h.getMax()
                + ", getMin = " + h.getMin() + ", valid = " + h.isValid() + "\n");

        System.out.print("extractMax = " + h.extractMax() + "  -> ");
        h.printHeap();
        System.out.print("extractMax = " + h.extractMax() + "  -> ");
        h.printHeap();

        h.increaseKey(6, 75);
        System.out.print("increaseKey(6, 75)  -> ");
        h.printHeap();
        h.decreaseKey(0, 20);
        System.out.print("decreaseKey(0, 20)  -> ");
        h.printHeap();
        System.out.println("valid = " + h.isValid() + "\n");

        int[] arr = {12, 30, 45, 7, 60, 18, 52, 25, 3, 41};
        System.out.println("array:                " + join(arr, arr.length));
        System.out.println("isMaxHeap(array)      = " + isMaxHeap(arr));
        h.buildMaxHeap(arr);
        System.out.print("buildMaxHeap(array) -> ");
        h.printHeap();
        System.out.println("isMaxHeap(heap)       = " + isMaxHeap(h.data()));
        heapSort(arr);
        System.out.println("heapSort(array)     -> " + join(arr, arr.length));
        System.out.println();

        h.clear();
        System.out.println("after clear: size = " + h.size()
                + ", empty = " + h.empty());
        try {
            h.extractMax();
        } catch (RuntimeException e) {
            System.out.println("error: " + e.getMessage());
        }
        h.insert(5);
        try {
            h.increaseKey(0, 3);
```

Listing 14.4 (Continued) *MaxHeap.java: the MaxHeap class in Java, with an array that doubles when it is full.*

```
        } catch (RuntimeException e) {
            System.out.println("error: " + e.getMessage());
        }
        try {
            h.increaseKey(4, 99);
        } catch (RuntimeException e) {
            System.out.println("error: " + e.getMessage());
        }
    }
}
```

Listing 14.5 `max_heap.py`: the `MaxHeap` class in Python.

```
# max_heap.py -- our MaxHeap class: a max-heap of int keys kept in a list.


class MaxHeap:
    def __init__(self):
        self._heap = []                  # _heap[0] is the root

    def size(self):
        return len(self._heap)

    def empty(self):
        return not self._heap

    def clear(self):
        self._heap.clear()

    def data(self):
        return list(self._heap)

    def print_heap(self):
        """Prints the list that stores the heap, in index order."""
        print(" ".join(map(str, self._heap)))

    def insert(self, key):
        """Appends key as the last leaf, then lets it rise: O(lg n)."""
        self._heap.append(key)
        self._sift_up(len(self._heap) - 1)

    def get_max(self):
        """Returns the largest key without removing it: O(1)."""
        self._require_non_empty("getMax")
        return self._heap[0]

    def extract_max(self):
        """Removes and returns the largest key: O(lg n)."""
        self._require_non_empty("extractMax")
        h = self._heap
        top = h[0]
        last = h.pop()                   # the last leaf fills the hole
        if h:
            h[0] = last
            MaxHeap._sift_down(h, 0, len(h))
        return top
```

Listing 14.5 (Continued) *max_heap.py: the MaxHeap class in Python.*

```
    def get_min(self):
        """Returns the smallest key; it is on a leaf, so scan the leaves: O(n)."""
        self._require_non_empty("getMin")
        return min(self._heap[len(self._heap) // 2:])

    def increase_key(self, index, new_value):
        """Raises the key at index to new_value, then lets it rise: O(lg n)."""
        self._check_index(index, "increaseKey")
        if new_value < self._heap[index]:
            raise ValueError("increaseKey: new value is below the key")
        self._heap[index] = new_value
        self._sift_up(index)

    def decrease_key(self, index, new_value):
        """Lowers the key at index to new_value, then lets it sink: O(lg n)."""
        self._check_index(index, "decreaseKey")
        if new_value > self._heap[index]:
            raise ValueError("decreaseKey: new value is above the key")
        self._heap[index] = new_value
        MaxHeap._sift_down(self._heap, index, len(self._heap))

    def build_max_heap(self, arr):
        """Replaces the contents by arr and rearranges it bottom-up: O(n)."""
        self._heap = list(arr)
        n = len(self._heap)
        for i in range(n // 2 - 1, -1, -1):
            MaxHeap._sift_down(self._heap, i, n)

    @staticmethod
    def is_max_heap(arr):
        """True when every element is no larger than its parent: O(n)."""
        return all(arr[(i - 1) // 2] >= arr[i] for i in range(1, len(arr)))

    def is_valid(self):
        return MaxHeap.is_max_heap(self._heap)

    @staticmethod
    def heap_sort(arr):
        """Sorts arr into ascending order in place with heap sort: O(n lg n)."""
        n = len(arr)
        for i in range(n // 2 - 1, -1, -1):
            MaxHeap._sift_down(arr, i, n)
        for end in range(n - 1, 0, -1):
            arr[0], arr[end] = arr[end], arr[0]   # the maximum goes to its slot
            MaxHeap._sift_down(arr, 0, end)       # repair the heap arr[0..end-1]
```

Listing 14.5 (Continued) *max_heap.py: the MaxHeap class in Python.*

```
    def _sift_up(self, i):
        """Lets _heap[i] rise while it is larger than its parent."""
        h = self._heap
        while i > 0 and h[(i - 1) // 2] < h[i]:
            p = (i - 1) // 2
            h[i], h[p] = h[p], h[i]
            i = p

    @staticmethod
    def _sift_down(a, i, n):
        """Lets a[i] sink inside the heap a[0..n-1] until no child is larger."""
        while True:
            largest, left, right = i, 2 * i + 1, 2 * i + 2
            if left < n and a[left] > a[largest]:
                largest = left
            if right < n and a[right] > a[largest]:
                largest = right
            if largest == i:
                return
            a[i], a[largest] = a[largest], a[i]
            i = largest

    def _require_non_empty(self, op):
        if not self._heap:
            raise IndexError(f"{op}: the heap is empty")

    def _check_index(self, index, op):
        if not 0 <= index < len(self._heap):
            raise IndexError(f"{op}: index {index} is outside the heap")


def join(a):
    return " ".join(map(str, a))


if __name__ == "__main__":
    h = MaxHeap()
    print("Urgent hold requests arrive (urgency scores):")
    for key in (40, 25, 70, 10, 55, 90, 30, 65, 80):
        h.insert(key)
        print(f"  insert({key})  -> ", end="")
        h.print_heap()
    print(f"size = {h.size()}, getMax = {h.get_max()}, getMin = {h.get_min()}, "
          f"valid = {str(h.is_valid()).lower()}\n")
```

Listing 14.5 (Continued) *max_heap.py: the MaxHeap class in Python.*

```
    print(f"extractMax = {h.extract_max()}  -> ", end="")
    h.print_heap()
    print(f"extractMax = {h.extract_max()}  -> ", end="")
    h.print_heap()

    h.increase_key(6, 75)
    print("increaseKey(6, 75)  -> ", end="")
    h.print_heap()
    h.decrease_key(0, 20)
    print("decreaseKey(0, 20)  -> ", end="")
    h.print_heap()
    print(f"valid = {str(h.is_valid()).lower()}\n")

    arr = [12, 30, 45, 7, 60, 18, 52, 25, 3, 41]
    print("array:                " + join(arr))
    print(f"isMaxHeap(array)     = {str(MaxHeap.is_max_heap(arr)).lower()}")
    h.build_max_heap(arr)
    print("buildMaxHeap(array) -> ", end="")
    h.print_heap()
    print(f"isMaxHeap(heap)      = {str(MaxHeap.is_max_heap(h.data())).lower()}")
    MaxHeap.heap_sort(arr)
    print("heapSort(array)     -> " + join(arr) + "\n")

    h.clear()
    print(f"after clear: size = {h.size()}, empty = {str(h.empty()).lower()}")
    try:
        h.extract_max()
    except IndexError as e:
        print(f"error: {e}")
    h.insert(5)
    try:
        h.increase_key(0, 3)
    except ValueError as e:
        print(f"error: {e}")
    try:
        h.increase_key(4, 99)
    except IndexError as e:
        print(f"error: {e}")
```

The Java class manages its own array, so `clear()` merely sets the size to zero, and `extractMax` shrinks the heap by decrementing `n` rather than by removing an element; the old value stays in the array, unreachable, until it is overwritten. The Python `get_min` uses slicing and the built-in `min` over the leaves, `min(self._heap[len(self._heap) // 2:])`, which is the same scan in one line. Python's `heap_sort` and `_sift_down` are static methods because they work on any list.

14.13 Heap Sort in Full

14.13.1 The algorithm

Section 3.10 introduced heap sort and ran it on six elements. We now have everything needed to state it precisely, prove it correct and analyze it. Heap sort sorts an array into ascending order *inside the array itself*, in two phases, which **Pseudocode 14.9** states in full.

1. **Build.** Make the whole array a max-heap with `BUILD-MAX-HEAP`.
2. **Extract.** Repeat for `end` $= n - 1$ down to 1: the maximum of the heap $A[0..end]$ is at the root; exchange it with $A[end]$, its final place; the heap is now $A[0..end - 1]$ with a new root that may be too small; repair it with sift-down.

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]
```

The second phase is `extractMax` with one twist: instead of throwing the maximum away, it puts it in the slot that the shrinking heap has just given up. The array is thus always divided in two: a heap at the front and a sorted tail at the back. **Figure 14.10** shows it on the heap that **Figure 14.8** built.

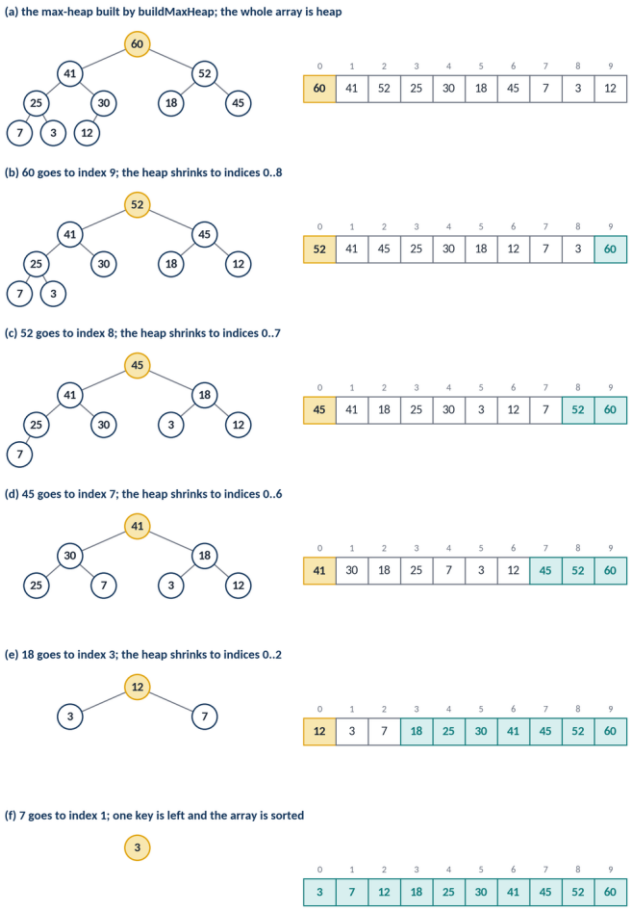


Figure 14.10 Heap sort on the heap of **Figure 14.8**: after each exchange and sift-down the heap (the tree, root in amber) shrinks by one and the sorted tail (teal) grows by one. Panels (e) and (f) skip the intermediate steps shown in **Listing 14.6**'s output.

Listing 14.6 prints every state of the sort and counts its key comparisons. Its `heapSort` is a function template that takes the comparison as a parameter, `less(x, y)` meaning "x comes before y", so the same code sorts numbers and, in the second half of the program, book records by year of publication.

Listing 14.6 `heap_sort_full.cpp`: heap sort with a trace of every state, and the same sort applied to book records to show that it is not stable.

```cpp
// heap_sort_full.cpp -- heap sort in place, traced; and why it is not stable.
#include <algorithm>
#include <cstdio>
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

long long comparisons = 0;

// Lets a[i] sink inside the heap a[0..n-1]; less(x, y) means "x before y".
template <class T, class Less>
void siftDown(vector<T>& a, int i, int n, Less less) {
    while (true) {
        int largest = i, l = 2 * i + 1, r = 2 * i + 2;
        if (l < n && (++comparisons, less(a[largest], a[l]))) largest = l;
        if (r < n && (++comparisons, less(a[largest], a[r]))) largest = r;
        if (largest == i) return;
        swap(a[i], a[largest]);
        i = largest;
    }
}

template <class T, class Less>
void heapSort(vector<T>& a, Less less,
              void (*show)(const vector<T>&, int) = nullptr) {
    int n = static_cast<int>(a.size());
    for (int i = n / 2 - 1; i >= 0; --i) siftDown(a, i, n, less);   // phase 1
    if (show) show(a, n);
    for (int end = n - 1; end > 0; --end) {                         // phase 2
        swap(a[0], a[end]);
        siftDown(a, 0, end, less);
        if (show) show(a, end);
    }
}

void printInts(const vector<int>& a, int heapSize) {
    cout << "  heap [";
    for (int i = 0; i < heapSize; ++i) cout << (i ? " " : "") << a[i];
    cout << "]  sorted [";
    for (int i = heapSize; i < static_cast<int>(a.size()); ++i)
        cout << (i > heapSize ? " " : "") << a[i];
    cout << "]\n";
}
```

Listing 14.6 (Continued) *heap_sort_full.cpp: heap sort with a trace of every state, and the same sort applied to book records to show that it is not stable.*

```
struct Book {
    int year;
    string title;
};

string cellOf(const Book& b) { return to_string(b.year) + " " + b.title; }

void printColumns(const vector<Book>& in, const vector<Book>& h,
                  const vector<Book>& s) {
    printf("%-27s%-27s%s\n", "input", "heap sort", "stable sort");
    for (size_t i = 0; i < in.size(); ++i)
        printf("%-27s%-27s%s\n", cellOf(in[i]).c_str(), cellOf(h[i]).c_str(),
               cellOf(s[i]).c_str());
}

int main() {
    vector<int> a = {12, 30, 45, 7, 60, 18, 52, 25, 3, 41};
    cout << "Heap sort of 12 30 45 7 60 18 52 25 3 41\n";
    heapSort(a, [](int x, int y) { return x < y; }, printInts);
    cout << "comparisons = " << comparisons << "\n\n";

    vector<Book> books = {{2012, "Pulang"}, {1980, "Bumi Manusia"},
                          {2012, "Amba"}, {1998, "Saman"},
                          {1980, "Ronggeng Dukuh Paruk"}, {2005, "Laskar Pelangi"}};
    auto byYear = [](const Book& x, const Book& y) { return x.year < y.year; };
    vector<Book> h = books, s = books;
    heapSort(h, byYear);
    stable_sort(s.begin(), s.end(), byYear);
    printColumns(books, h, s);
    return 0;
}
```

```
Heap sort of 12 30 45 7 60 18 52 25 3 41
  heap [60 41 52 25 30 18 45 7 3 12]  sorted []
  heap [52 41 45 25 30 18 12 7 3]  sorted [60]
  heap [45 41 18 25 30 3 12 7]  sorted [52 60]
  heap [41 30 18 25 7 3 12]  sorted [45 52 60]
  heap [30 25 18 12 7 3]  sorted [41 45 52 60]
  heap [25 12 18 3 7]  sorted [30 41 45 52 60]
  heap [18 12 7 3]  sorted [25 30 41 45 52 60]
  heap [12 3 7]  sorted [18 25 30 41 45 52 60]
  heap [7 3]  sorted [12 18 25 30 41 45 52 60]
  heap [3]  sorted [7 12 18 25 30 41 45 52 60]
comparisons = 38

input                       heap sort                   stable sort
2012 Pulang                 1980 Bumi Manusia           1980 Bumi Manusia
1980 Bumi Manusia           1980 Ronggeng Dukuh Paruk   1980 Ronggeng Dukuh Paruk
2012 Amba                   1998 Saman                  1998 Saman
1998 Saman                  2005 Laskar Pelangi         2005 Laskar Pelangi
1980 Ronggeng Dukuh Paruk   2012 Amba                   2012 Pulang
2005 Laskar Pelangi         2012 Pulang                 2012 Amba
```

The first line after the heading is the heap after phase 1, the same as **Figure 14.8**(f). Each following line is one iteration of phase 2: 60 goes to index 9, and sift-down turns 12, 41, 52, 25, 30, 18, 45, 7, 3 into the heap 52, 41, 45, 25, 30, 18, 12, 7, 3; then 52 goes to index 8; and so on until one key is left.

14.13.2 Correctness

Invariant. At the start of each iteration of the loop of lines 2 to 4: $A[0..end]$ is a max-heap containing the $end + 1$ smallest keys of the array, and $A[end + 1..n - 1]$ contains the $n - end - 1$ largest keys in ascending order.

Initially $end = n - 1$: the whole array is a heap after line 1, and the tail is empty. *Maintenance:* the root $A[0]$ is the largest key of the heap, hence the largest of the $end + 1$ smallest keys, and so no larger than any key in the tail. Line 3 puts it at index end, directly in front of the tail, which therefore stays sorted and grows by one. Line 3 also put the old $A[end]$ at the root, the only possible violation in $A[0..end - 1]$, since both subtrees of the root are still heaps; line 4 repairs it. *Termination:* the loop ends with $end = 0$, so $A[1..n - 1]$ holds the $n - 1$ largest keys in order and $A[0]$ holds the smallest. The array is sorted.

14.13.3 Cost

Phase 1 costs $\Theta(n)$ by Equation (14.11). Phase 2 performs $n - 1$ exchanges and $n - 1$ sift-downs from the root of heaps of sizes $n - 1, n - 2, \ldots, 1$, each making at most $2\lfloor \lg n \rfloor$ comparisons by Equation (14.8). So

$$C_{\text{heap sort}}(n) \leq 2n + 2(n - 1)\lfloor \lg n \rfloor = O(n \lg n) \tag{14.13}$$

comparisons, in every case. On the ten keys above the program counted 38 comparisons, well below the bound $20 + 2 \cdot 9 \cdot 3 = 74$. There is no bad input: unlike quicksort, heap sort never degrades to quadratic time. Since any sort that works by comparing keys needs $\Omega(n \lg n)$ comparisons in the worst case (Section 3.11), heap sort is asymptotically optimal. It needs $O(1)$ extra memory, a few index variables, because the heap and the sorted tail share the one array; it does not even need the recursion stack of quicksort, since our sift-down is a loop.

Table 14.5 places heap sort among the $O(n \lg n)$ sorts of Chapter 3. It is the only one that combines a worst-case guarantee with sorting in place. Its weaknesses are practical rather than asymptotic: sift-down jumps from index i to $2i + 1$, far away in memory once i is large, which uses the processor's cache poorly, so in practice heap sort is usually slower than a good quicksort (Section 3.13 measured this); and it is not stable. For the same reason, the introsort behind `std::sort` in the major C++ libraries starts as a quicksort and switches to heap sort only when the recursion becomes suspiciously deep, getting quicksort's speed on ordinary inputs and heap sort's worst-case guarantee on bad ones [8] (Section 3.14.1).

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

14.13.4 Why heap sort is not stable

A sort is *stable* when keys that compare equal keep the order they had in the input (Section 3.12). Heap sort breaks this, and the smallest example shows why. **Figure 14.11** sorts the three keys 5, 5, 3, where the two 5s are labeled a and b to tell them apart. The input is already a max-heap. The first exchange sends 5a, the root, to the *last* position, and 5b later lands in front of it: the output 3, 5b, 5a has the two equal keys in the opposite order.

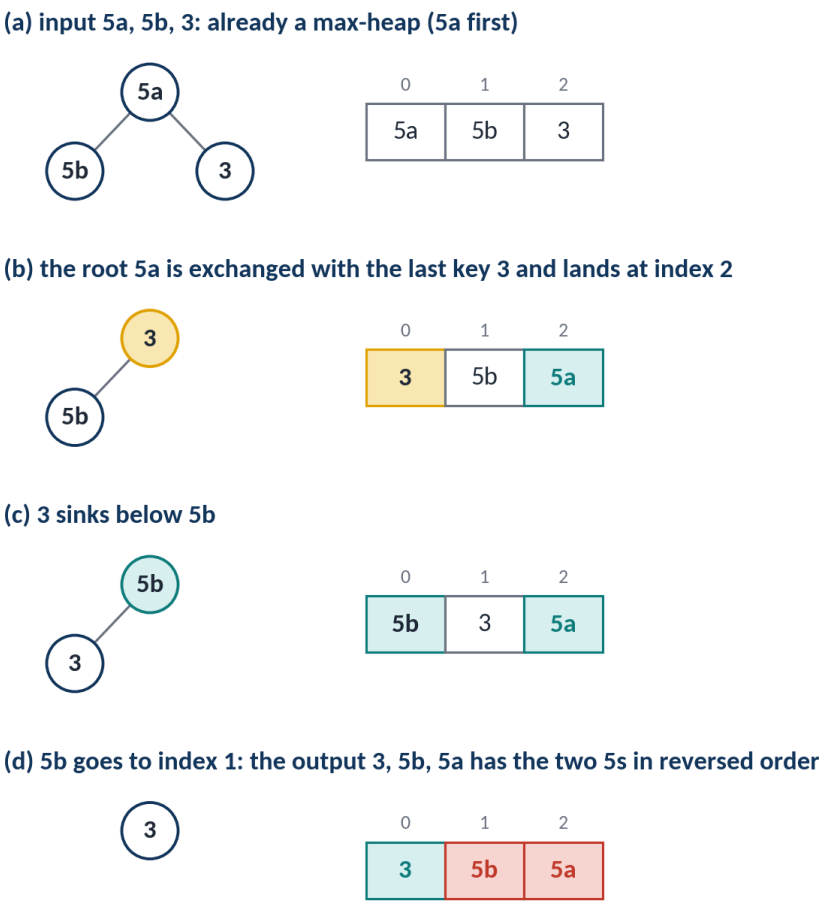


Figure 14.11 Heap sort of 5a, 5b, 3 by key: the first exchange moves 5a from the front to the back, past 5b, and the output ends with 5b before 5a.

The long-distance exchanges are the cause: every iteration moves a key from the root straight to the end of the heap, jumping over everything in between, and equal keys have no way of keeping their order. The second half of **Listing 14.6** shows the effect on real records. Sorting six books by year, heap sort puts *Amba* before *Pulang* although *Pulang* came first in the input, while `std::stable_sort` keeps them in input order. (The two books of 1980 happen to keep their order in this run; stability is a guarantee for *every* input, and heap sort gives none.) Heap sort can be made stable by sorting pairs (key, original position) instead of keys, so that no two elements are equal; this costs an extra $\Theta(n)$ memory, which removes heap sort's main advantage. Exercise 14.8 asks you to work it out.

14.14 Priority Queues in the Standard Libraries

Having built a heap by hand, you have earned the right to use the library versions, and in real programs you should, unless you need an operation they lack. Section 5.16 already used them briefly. Here we look at them again with the knowledge of how they work, using the Pustaka Ceria hold requests as records: each request has an urgency, an arrival number, a reader and a title, and the desk serves the highest urgency first and, among equal urgencies, the earliest arrival.

14.14.1 C++: `std::priority_queue`

`std::priority_queue<T, Container, Compare>` is a *container adaptor*: it keeps its elements in a `std::vector<T>` by default and maintains them as a binary heap, exactly as our class does [9]. `top()` reads the root in $O(1)$; `push` appends and sifts up; `pop` moves the last element to the root

and sifts down, both in $O(\lg n)$. Since C++23 there is also `push_range`, which inserts a whole range at once [9]. The same algorithms are available on any random-access range as free functions, `std::make_heap` (Floyd's bottom-up build), `std::push_heap`, `std::pop_heap`, `std::sort_heap` and `std::is_heap`, which correspond to our `buildMaxHeap`, `insert`, `extractMax`, `heapSort` and `isMaxHeap` [10].

Three details trip up nearly everyone.

- **It is a max-heap by default**, because the default comparison is `std::less<T>`: the element that is "largest" under `<` is on top. For a min-heap, pass `std::greater<T>`.
- **The comparison means "is served later than".** `Compare(a, b)` returns `true` when `a` has *lower* priority than `b`, that is, when `a` belongs below `b`. This is backward from how most people think of priority, and a comparison written the other way round silently reverses the queue.
- **`pop()` returns nothing.** Read `top()` first, then `pop()`. And calling either on an empty queue is undefined behavior, not an exception: check `empty()` first.

Listing 14.7 puts the hold requests into a `priority_queue` with a comparison `ServedLater`, which says that request `a` is served later than `b` when its urgency is lower, or when the urgencies are equal and `a` arrived later. It then shows a plain max-queue and a min-queue of integers.

Listing 14.7 `library_pq.cpp`: `std::priority_queue` with a custom comparison serving hold requests, and the default max-queue and a `std::greater` min-queue of integers.

```cpp
// library_pq.cpp -- std::priority_queue serving Pustaka Ceria hold requests.
#include <cstdio>
#include <functional>
#include <queue>
#include <string>
#include <vector>
using namespace std;

struct Request {
    int urgency;      // higher is more urgent
    int arrival;      // order of arrival: 1, 2, 3, ...
    string member;
    string title;
};

// "a is served later than b": the comparison priority_queue expects.
struct ServedLater {
    bool operator()(const Request& a, const Request& b) const {
        if (a.urgency != b.urgency) return a.urgency < b.urgency;
        return a.arrival > b.arrival;   // equal urgency: earlier arrival first
    }
};

int main() {
    priority_queue<Request, vector<Request>, ServedLater> holds;
    holds.push({70, 1, "Ayu", "Laskar Pelangi"});
    holds.push({90, 2, "Budi", "Bumi Manusia"});
    holds.push({40, 3, "Citra", "Saman"});
    holds.push({90, 4, "Dewi", "Pulang"});
    holds.push({55, 5, "Eko", "Amba"});
    holds.push({70, 6, "Fajar", "Cantik Itu Luka"});

    printf("Serving %zu hold requests (urgency, then arrival):\n", holds.size());
    while (!holds.empty()) {
        const Request& r = holds.top();              // O(1): the root
        printf("  urgency %2d  #%d  %-6s %s\n", r.urgency, r.arrival,
               r.member.c_str(), r.title.c_str());
        holds.pop();                                 // O(lg n): extract
    }
```

Listing 14.7 (Continued) *library_pq.cpp: std::priority_queue with a custom comparison serving hold requests, and the default max-queue and a std::greater min-queue of integers.*

```cpp
    priority_queue<int> maxq;                        // largest on top
    priority_queue<int, vector<int>, greater<int>> minq;  // smallest on top
    for (int d : {12, 3, 8, 14, 5}) {
        maxq.push(d);
        minq.push(d);
    }
    printf("Days until due, largest first:");
    while (!maxq.empty()) { printf(" %d", maxq.top()); maxq.pop(); }
    printf("\nDays until due, smallest first:");
    while (!minq.empty()) { printf(" %d", minq.top()); minq.pop(); }
    printf("\n");
    return 0;
}
```

14.14.2 Java: `java.util.PriorityQueue`

Java's `PriorityQueue<E>` is a binary heap in a growable array, and it is a **min**-heap: its head is the *least* element according to the natural order or to the `Comparator` given to the constructor [11]. `offer` and `add` insert, `poll` removes the head, `peek` reads it; the documentation states $O(\lg n)$ for insertion and removal of the head, constant time for `peek` and `size`, and linear time for

`remove(Object)` and `contains(Object)`, which must search the array [11]. For a max-queue, pass `Collections.reverseOrder()` or a reversed comparator. Two more facts from the documentation: ties between equal elements are broken arbitrarily, so if arrival order matters it must be part of the comparator; and the iterator visits the elements in *array* order, which is heap order, not sorted order. The class is not thread-safe; `PriorityBlockingQueue` is the concurrent version. **Listing 14.8** serves the same hold requests with a comparator that sorts by urgency, reversed, and then by arrival.

Listing 14.8 `LibraryPQ.java`: `java.util.PriorityQueue` with a comparator built from `comparingInt`, `reversed` and `thenComparingInt`.

```
// LibraryPQ.java -- java.util.PriorityQueue serving Pustaka Ceria hold requests.
import java.util.Collections;
import java.util.Comparator;
import java.util.PriorityQueue;

public class LibraryPQ {
    record Request(int urgency, int arrival, String member, String title) {}

    public static void main(String[] args) {
        // Highest urgency first; equal urgency: earlier arrival first.
        Comparator<Request> served = Comparator.comparingInt(Request::urgency)
                .reversed().thenComparingInt(Request::arrival);
        PriorityQueue<Request> holds = new PriorityQueue<>(served);
        holds.add(new Request(70, 1, "Ayu", "Laskar Pelangi"));
        holds.add(new Request(90, 2, "Budi", "Bumi Manusia"));
        holds.add(new Request(40, 3, "Citra", "Saman"));
        holds.add(new Request(90, 4, "Dewi", "Pulang"));
        holds.add(new Request(55, 5, "Eko", "Amba"));
        holds.add(new Request(70, 6, "Fajar", "Cantik Itu Luka"));

        System.out.printf("Serving %d hold requests (urgency, then arrival):%n",
                holds.size());
        while (!holds.isEmpty()) {
            Request r = holds.poll();                    // O(lg n): extract
            System.out.printf("  urgency %2d  #%d  %-6s %s%n", r.urgency(),
                    r.arrival(), r.member(), r.title());
        }

        PriorityQueue<Integer> maxq =
                new PriorityQueue<>(Collections.reverseOrder());  // largest first
        PriorityQueue<Integer> minq = new PriorityQueue<>();   // smallest first
        for (int d : new int[] {12, 3, 8, 14, 5}) {
            maxq.add(d);
            minq.add(d);
        }
        StringBuilder a = new StringBuilder("Days until due, largest first:");
        while (!maxq.isEmpty()) a.append(" ").append(maxq.poll());
        StringBuilder b = new StringBuilder("Days until due, smallest first:");
        while (!minq.isEmpty()) b.append(" ").append(minq.poll());
        System.out.println(a);
        System.out.println(b);
    }
}
```

14.14.3 Python: `heapq`, including the new max-heap functions

Python has no priority queue class in its core; instead the module `heapq` provides functions that treat an ordinary `list` as a binary min-heap [12]. `heapq.heappush(h, x)` and `heapq.heappop(h)` insert and extract the minimum, `h[0]` is the minimum, `heapq.heapify(h)` builds a heap in linear time, `heapq.heapreplace(h, x)` pops the minimum and pushes `x` with a single sift, and

`heapq.heappushpop(h, x)` pushes and then pops. The module also offers the applications of Section 14.15 ready-made: `heapq.nlargest(k, it)`, `heapq.nsmallest(k, it)` and `heapq.merge(*iterables)`.

For more than twenty years the module had no public max-heap. Programmers stored negated keys, $-x$, in a min-heap, which works for numbers and not at all for strings. **Python 3.14 added max-heap versions of the five core functions**: `heapify_max`, `heappush_max`, `heappop_max`, `heappushpop_max` and `heapreplace_max`, all marked "Added in version 3.14" in the documentation [12]. **Listing 14.9** uses them when they exist and falls back to negated keys on older versions, so it runs, with identical output, from Python 3.11 upward. (We ran it on Python 3.11 and on Python 3.14.)

Listing 14.9 `library_pq.py`: `heapq` with tuples as records, and the max-heap functions of Python 3.14 with a fallback for older versions.

```
# library_pq.py -- heapq serving Pustaka Ceria hold requests.
import heapq

try:                                    # Python 3.14 and newer: max-heap functions
    from heapq import heappop_max, heappush_max
except ImportError:                     # older Pythons: a min-heap of negated keys
    def heappush_max(heap, item):
        heapq.heappush(heap, -item)

    def heappop_max(heap):
        return -heapq.heappop(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}")

maxq, minq = [], []
for d in (12, 3, 8, 14, 5):
    heappush_max(maxq, d)               # largest on top
    heapq.heappush(minq, d)             # smallest on top
print("Days until due, largest first:",
      " ".join(str(heappop_max(maxq)) for _ in range(len(maxq))))
print("Days until due, smallest first:",
      " ".join(str(heapq.heappop(minq)) for _ in range(len(minq))))
```

The records are tuples `(-urgency, arrival, member, title)`. Python compares tuples element by element, from the left, so the heap orders them by negated urgency first (highest urgency on top of a min-heap) and by arrival number second. The arrival number also does a second job: it guarantees that two tuples are never fully equal, so Python never needs to compare the remaining fields. Without it, two requests with the same urgency would be compared by reader name, which is not what the desk wants; and a record type with no order at all would raise an error. Using a counter as the second element is the standard idiom [12].

All three programs print:

```
Serving 6 hold requests (urgency, then arrival):
  urgency 90  #2  Budi   Bumi Manusia
  urgency 90  #4  Dewi   Pulang
  urgency 70  #1  Ayu    Laskar Pelangi
  urgency 70  #6  Fajar  Cantik Itu Luka
  urgency 55  #5  Eko    Amba
  urgency 40  #3  Citra  Saman
Days until due, largest first: 14 12 8 5 3
Days until due, smallest first: 3 5 8 12 14
```

The two requests with urgency 90 are served in order of arrival, #2 before #4, and so are the two with urgency 70; the tie-breaking rule works in all three languages. **Table 14.6** summarizes the three 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

The last row matters for the next section. None of the libraries lets you change the key of an element already inside, because none of them tells you where the element is. Section 14.15.3 shows how Dijkstra’s algorithm lives without it.

14.15 Applications: Top-k, Merging k Sorted Lists, and Dijkstra

A heap earns its keep whenever a program must repeatedly find the best of a changing set. Three classic problems show three different ways of using it.

14.15.1 The k largest items of a stream

Pustaka Ceria wants the three most borrowed books of the year for its front window. The borrow counts arrive as a stream, possibly millions of them, and we want the k largest. Sorting everything costs $O(n \lg n)$ time and $O(n)$ memory. Building a max-heap and extracting k times costs $O(n + k \lg n)$ but still $O(n)$ memory.

The surprising solution uses a **min**-heap of size k. It holds the k best items seen so far, with the weakest of them at the root. For every new item, compare it with the root: if it is not better than the weakest of the current top k, it cannot be among the top k at the end and is discarded; if it is better, it replaces the root, which sinks into place.

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
```

Each item costs at most one sift in a heap of k elements, so

$$T_{\text{top-}k}(n) = O(n \lg k) \text{ time}, \qquad O(k) \text{ memory}. \tag{14.14}$$

For $k = 3$ and a million items that is a million comparisons with the root and, in the worst case, a few exchanges each, in a heap that never holds more than three items. The memory bound is what makes the method indispensable: it works on a stream too large to keep.

14.15.2 Merging k sorted lists

Pustaka Ceria has three branches, and each keeps its catalog sorted by book number. The central catalog needs one sorted list of all of them. Merging two sorted lists is the merge step of merge sort (Section 3.8). For k lists, comparing the k front elements at every step costs $O(k)$ per output element. A min-heap of the k front elements does better (**Pseudocode 14.11**): the smallest of them is at the root; output it, and replace it by the next element of the same list.

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))
```

The heap never holds more than k entries, and each of the N elements in total passes through it once, so

$$T_{\text{merge}}(N, k) = O(N \lg k). \tag{14.15}$$

The same idea sorts data too large for memory (*external sorting*): sort chunks that fit, write them out as sorted runs, then merge all the runs at once with a heap. **Figure 14.12** pictures both applications, and **Listing 14.10** runs them.

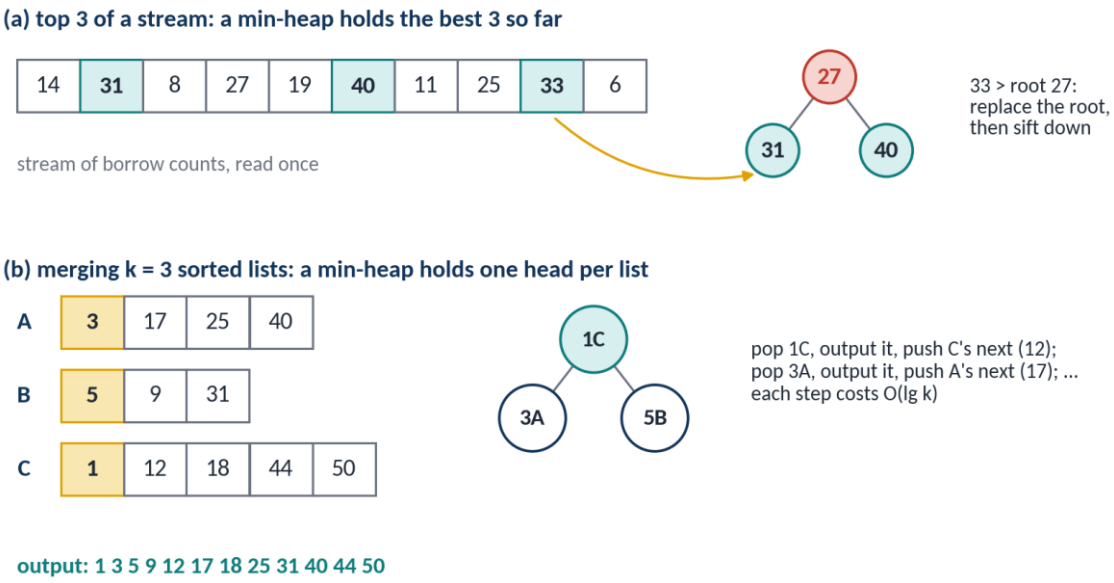


Figure 14.12 (a) Top 3 of a stream: when 33 arrives, the min-heap holds 27, 31 and 40; since 33 > 27, it replaces the root. Teal marks the top 3 at the end of the stream. (b) Merging three sorted lists: the min-heap holds the front of each list (amber), and the root is always the next output.

Listing 14.10 `topk_merge.cpp`: the three most borrowed books of a stream with a min-heap of size 3, and the merge of three sorted branch catalogs.

```cpp
// topk_merge.cpp -- two classic heap applications: top-k and k-way merge.
#include <algorithm>
#include <cstdio>
#include <functional>
#include <queue>
#include <string>
#include <tuple>
#include <vector>
using namespace std;

int main() {
    // Part 1: the k most borrowed books, read as a stream, with a min-heap of k.
    vector<pair<int, string>> borrows = {
        {14, "Saman"}, {31, "Laskar Pelangi"}, {8, "Amba"},
        {27, "Bumi Manusia"}, {19, "Pulang"}, {40, "Negeri 5 Menara"},
        {11, "Supernova"}, {25, "Cantik Itu Luka"}, {33, "Ayat-Ayat Cinta"},
        {6, "Ronggeng Dukuh Paruk"}};
    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);
    }
    vector<pair<int, string>> best;
    while (!top.empty()) { best.push_back(top.top()); top.pop(); }
    reverse(best.begin(), best.end());
    printf("Top %zu:", k);
    for (size_t i = 0; i < best.size(); ++i)
        printf("%s %s (%d)", i ? "," : "", best[i].second.c_str(), best[i].first);
    sort(borrows.rbegin(), borrows.rend());               // check: sort everything
    bool same = equal(best.begin(), best.end(), borrows.begin());
    printf("\ncheck against a full sort: %s\n\n", same ? "yes" : "no");
```

Listing 14.10 (Continued) *topk_merge.cpp: the three most borrowed books of a stream with a min-heap of size 3, and the merge of three sorted branch catalogs.*

```cpp
    // Part 2: merge k sorted lists of book numbers with a min-heap of k heads.
    vector<vector<int>> branch = {{3, 17, 25, 40}, {5, 9, 31}, {1, 12, 18, 44, 50}};
    const char* name = "ABC";
    using Head = tuple<int, int, int>;                    // value, list, position
    priority_queue<Head, vector<Head>, greater<Head>> heads;
    for (int b = 0; b < static_cast<int>(branch.size()); ++b)
        heads.push({branch[b][0], b, 0});
    printf("Merged catalog:");
    while (!heads.empty()) {
        auto [value, b, pos] = heads.top();
        heads.pop();
        printf(" %d%c", value, name[b]);
        if (pos + 1 < static_cast<int>(branch[b].size()))
            heads.push({branch[b][pos + 1], b, pos + 1});  // that list's next
    }
    printf("\n");
    return 0;
}
```

```
read 14 Saman                -> weakest kept: 14
read 31 Laskar Pelangi       -> weakest kept: 14
read  8 Amba                 -> weakest kept: 8
read 27 Bumi Manusia         -> weakest kept: 14
read 19 Pulang               -> weakest kept: 19
read 40 Negeri 5 Menara      -> weakest kept: 27
read 11 Supernova            -> weakest kept: 27
read 25 Cantik Itu Luka      -> weakest kept: 27
read 33 Ayat-Ayat Cinta      -> weakest kept: 31
read  6 Ronggeng Dukuh Paruk -> weakest kept: 31
Top 3: Negeri 5 Menara (40), Ayat-Ayat Cinta (33), Laskar Pelangi (31)
check against a full sort: yes

Merged catalog: 1C 3A 5B 9B 12C 17A 18C 25A 31B 40A 44C 50C
```

The “weakest kept” column is the root of the min-heap after each item. It rises from 8 to 14, 19, 27 and finally 31 as better items push weaker ones out; items 11, 25 and 6 never enter because they do not beat the root. The merged catalog labels each number with its branch. In C++ the min-heap is a `priority_queue` with `std::greater`; pairs and tuples compare lexicographically, which orders the heads by value. The Python version, `topk_merge.py`, uses `heapq.heapreplace` for line 5 of **Pseudocode 14.10**, and it confirms its answer against `heapq.nlargest`, which implements the same algorithm; `heapq.merge` does all of Part 2 in one call. `TopKMerge.java` gives the Java version.

14.15.3 The hook to Chapter 13: Dijkstra’s shortest paths

Section 13.10 found shortest routes on the weighted campus map with Dijkstra’s algorithm [13]: it repeatedly takes the not-yet-final vertex with the smallest tentative distance out of a min-priority queue, declares that distance final, and *relaxes* the edges leaving it, lowering the tentative distance $d[v]$ of a neighbor v when the route through u is shorter. Section 13.10.6 then observed that everything non-linear in its running time comes from the priority queue, and passed the question to this chapter. We can now answer it from the heap’s side.

Lowering $d[v]$ is a `decreaseKey` in a min-heap. As Section 14.14 showed, the library queues have no `decreaseKey`, which is why Section 13.10.3 used the *lazy* version: instead of changing the entry of v, push a *new* entry $(d[v], v)$ and leave the old, larger entry in the heap; when an old entry reaches the top later, its vertex is already final, and the entry is simply skipped. What does this cost the heap? To see it, we run the lazy algorithm on a second small map. The Pustaka Ceria delivery van carries books from the main library to five other buildings of the campus; **Figure 14.13** shows the roads with their travel times in minutes.

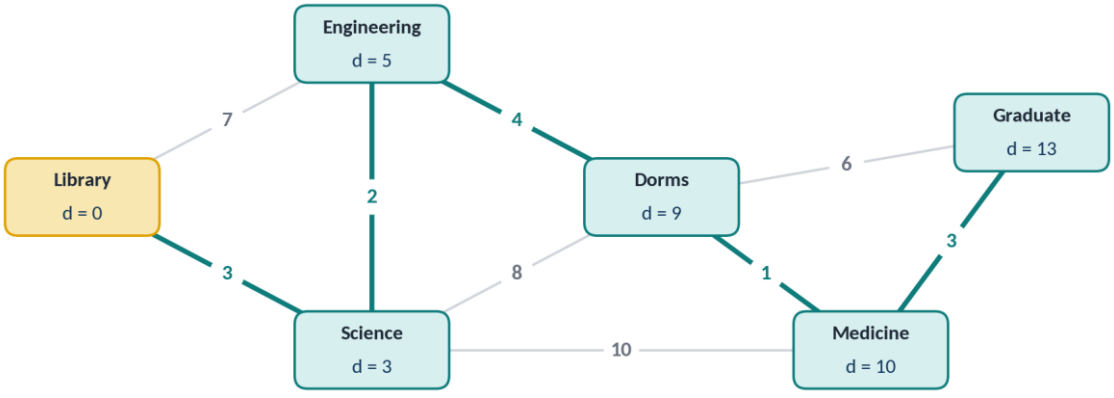


Figure 14.13 The delivery roads with travel times in minutes, and the shortest delivery time d from the Library to each building. The teal edges form the shortest-path tree that the priority queue finds.

Listing 14.11 prints every entry that leaves the heap.

Listing 14.11 `dijkstra_pq.cpp`: Dijkstra's algorithm on the delivery roads with `std::priority_queue` as a min-heap and lazy deletion.

```
// dijkstra_pq.cpp -- shortest delivery times with a min-priority queue.
#include <cstdio>
#include <functional>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

const char* place[] = {"Library", "Engineering", "Science", "Dorms",
                       "Medicine", "Graduate"};

int main() {
    const int n = 6, INF = 1000000000;
    vector<vector<pair<int, int>>> adj(n);          // adj[u] = {(v, minutes)}
    auto road = [&](int u, int v, int w) {
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
    };
    road(0, 1, 7); road(0, 2, 3); road(2, 1, 2); road(1, 3, 4); road(2, 3, 8);
    road(3, 4, 1); road(2, 4, 10); road(4, 5, 3); road(3, 5, 6);

    vector<int> dist(n, INF), prev(n, -1);
    vector<bool> done(n, false);
    using Entry = pair<int, int>;                    // (distance, vertex)
    priority_queue<Entry, vector<Entry>, greater<Entry>> pq;   // smallest first
    dist[0] = 0;
    pq.push({0, 0});
    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();
        if (done[u]) {                               // an out-of-date entry
            printf("pop %2d %-12s stale, skipped\n", d, place[u]);
            continue;
        }
        done[u] = true;
        printf("pop %2d %-12s settled\n", d, place[u]);
        for (auto [v, w] : adj[u])
            if (!done[v] && d + w < dist[v]) {        // a shorter way to v
                dist[v] = d + w;
                prev[v] = u;
                pq.push({dist[v], v});
            }
    }
    printf("\nFastest delivery from the Library:\n");
    for (int v = 0; v < n; ++v) {
        printf("  %-11s %2d min  via", place[v], dist[v]);
```

Listing 14.11 (Continued) *dijkstra_pq.cpp: Dijkstra's algorithm on the delivery roads with std::priority_queue as a min-heap and lazy deletion.*

```
        vector<int> path;
        for (int x = v; x != -1; x = prev[x]) path.push_back(x);
        for (int i = static_cast<int>(path.size()) - 1; i >= 0; --i)
            printf(" %s", place[path[i]]);
        printf("\n");
    }
    return 0;
}
```

```
pop  0 Library      settled
pop  3 Science      settled
pop  5 Engineering  settled
pop  7 Engineering  stale, skipped
pop  9 Dorms        settled
pop 10 Medicine     settled
pop 11 Dorms        stale, skipped
pop 13 Medicine     stale, skipped
pop 13 Graduate     settled
pop 15 Graduate     stale, skipped

Fastest delivery from the Library:
  Library      0 min  via Library
  Engineering  5 min  via Library Science Engineering
  Science      3 min  via Library Science
  Dorms        9 min  via Library Science Engineering Dorms
  Medicine    10 min  via Library Science Engineering Dorms Medicine
  Graduate    13 min  via Library Science Engineering Dorms Medicine Graduate
```

Follow the queue. The Library is settled at 0 and pushes Engineering at 7 and Science at 3. Science, at 3, comes out next and finds a shorter way to Engineering, $3 + 2 = 5 < 7$: a new entry (5, Engineering) is pushed, and the old entry (7, Engineering) stays behind. Engineering is settled at 5, and later the stale entry (7, Engineering) comes out and is skipped. The same happens to Dorms (11 is superseded by 9), Medicine (13 by 10) and Graduate (15 by 13). Four stale entries in all, and every shortest time agrees with **Figure 14.13**.

With lazy deletion every successful relaxation pushes one entry, at most one per edge direction, so the heap holds $O(|E|)$ entries, and each push or pop costs $O(\lg|E|) = O(\lg|V|)$, since $|E| < |V|^2$. The total is the bound of Section 13.10.6, now with every factor explained:

$$T_{\text{Dijkstra}} = O((|V| + |E|)\lg|V|). \tag{14.16}$$

With an indexed heap supporting a true `decreaseKey`, Section 14.9, the bound is the same but the heap never holds more than $|V|$ entries. The ten pushes of our run, for nine roads and six buildings, show the price of laziness: four of the ten entries were stale. More advanced heaps improve the bound: the Fibonacci heap of Fredman and Tarjan makes `decreaseKey` cost $O(1)$ amortized and brings Dijkstra's algorithm to $O(|E| + |V|\lg|V|)$ [14], as Section 13.10.6 mentioned. On ordinary graphs the simple binary heap with lazy deletion is usually the fastest in practice, which is why it is what most programs use. The Java and Python versions, `DijkstraPQ.java` and `dijkstra_pq.py`, print the same trace; in Python the queue is a list of tuples `(distance, vertex)` managed by `heapq`.

14.16 Profitina in Practice

Profitina is an AI visibility (GEO) platform: it measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds [15]. Its server is a Python application built with FastAPI, so the `heapq` module of Section 14.14.3 is the heap closest at hand. Several of its features have exactly the shapes of this chapter. We describe the shape of each problem and the class of structure that fits it; the actual implementation choices inside the product are its own.

- **What is due next.** The historical analytics feature re-scans each brand automatically, at a cadence that depends on its subscription tier. A scheduler facing many brands with different cadences asks one question over and over: *which scan is due soonest?* That is a min-priority queue keyed by the next due time. `extractMin` gives the next job in $O(\lg n)$; after the scan, the brand is pushed back with its next due time, one `insert`. Compared with scanning every brand's

due time at every tick, $\Theta(n)$ per tick, the heap makes the scheduler's cost depend on the number of scans actually run, not on the number of brands. The same shape, keyed by urgency instead of time, appears when some work is more urgent than other work, as in the Profitina Laundry orders of Exercise 14.12.

- **The top of a leaderboard.** Users earn PROFIT by improving AI responses, and the leaderboard shows who leads. Displaying the top k of many users is the top-k problem of Section 14.15.1: a min-heap of size k finds it in one pass in $O(n\lg k)$ time without sorting everyone, and a periodic recomputation of a small top list stays cheap however many users there are.
- **A dashboard merged from many feeds.** The dashboard gathers, in one call, notifications of several kinds: replies, likes, reposts, direct messages, pins, feedback status, PROFIT purchases. Each kind can be read from its own source already ordered by time, newest first. Showing them in one timeline is the k-way merge of Section 14.15.2 with k equal to the number of sources: $O(N\lg k)$ for N notifications, and if only the newest 20 are shown, the merge can stop after 20 outputs.
- **A crawl frontier.** Scanning a site means discovering pages and deciding which to fetch next. A crawler that prefers the most important pages, for example those closest to the home page or linked most often, keeps its frontier in a priority queue; one that simply takes pages in order of discovery is performing the breadth-first search of Chapter 13 with a FIFO queue.

In every case the heap replaces a repeated full scan or a repeated full sort by a logarithmic step, which is the whole message of this chapter.

14.17 Common Mistakes, and How to Avoid Them

Mixing the 0-based and 1-based formulas. With the root at index 0 the children of i are $2i+1$ and $2i+2$; with the root at 1 they are $2k$ and $2k+1$. Using $2i$ and $2i+1$ in a 0-based array makes the root its own left child and silently corrupts the tree. Pick one numbering (**Table 14.3**) and use it everywhere.

Sifting down toward the wrong child. Sift-down must exchange with the *larger* child in a max-heap (the smaller in a min-heap). Exchanging with the left child, or with the first child found to be larger, can put a smaller key above a larger sibling (Section 14.6.1).

Forgetting a child that does not exist. The right child of the last internal node may be missing; test `l < n` and `r < n` before reading `A[l]` and `A[r]`. In heap sort, n is the size of the *shrinking* heap, not of the array, or the sorted tail is pulled back into the heap.

Removing the root without moving the last leaf. `extractMax` must put the last element into the root before sifting down; deleting the root and promoting children leaves a gap and breaks completeness (Section 14.7.3).

Assuming a heap is sorted. Only the root is known; the array as a whole is in no useful order, and iterating over a `PriorityQueue` in Java or over a `heapq` list in Python visits the elements in heap order, not in sorted order. To get sorted output, extract repeatedly.

Searching a heap like a search tree. A key can be in either subtree; finding an arbitrary key takes $\Theta(n)$. This is also why `getMin` in a max-heap is $\Theta(n)$ (Section 14.8).

Building by n insertions when all keys are known. Use the bottom-up build, $\Theta(n)$ instead of $\Theta(n\lg n)$ in the worst case (**Listing 14.2**); start it at index $\lfloor n/2 \rfloor - 1$, not at $n-1$ (the leaves need no work) and not at 0 going upward (sift-down needs both subtrees to be heaps already).

Changing a key without repairing. Assigning `A[i] = k` inside a heap must be followed by sift-up (larger key) or sift-down (smaller key). An `increaseKey` that accepts a smaller value and sifts up leaves the heap broken without any error (Section 14.9).

Writing the `std::priority_queue` comparison backward. The comparison answers "is `a` served later than `b`?"; with `less` the queue is a max-heap. For "earliest deadline first", use `greater`, or a comparison that returns `true` when `a` has the later deadline. And read `top()` before `pop()`, which returns nothing.

Leaving ties to chance. Heaps are not stable, and none of the libraries promises an order among equal keys. If equal priorities must be served in arrival order, make the arrival number part of the key (Section 14.14).

Unsigned arithmetic in loop bounds. In C++, `v.size() - 1` or `(v.size() - 2) / 2` wraps around for small arrays; test the empty and one-element cases (Section 14.11).

Confusing the two heaps. The memory region of Section 1.3.5 and the data structure of this chapter share only a name.

14.18 Chapter Summary

- A **priority queue** serves the item with the highest (or lowest) key first. Unsorted and sorted arrays each make one of `insert` and `extractMax` linear; the **binary heap** makes both $O(\lg n)$ and `getMax` $O(1)$, with no extra memory (**Table 14.2**). It was introduced by Williams in 1964, with the linear-time build by Floyd.
- A **complete binary tree**, every level full except the last, which fills from the left, is stored level by level in an array. With the root at index 0, the parent of i is $\lfloor (i-1)/2 \rfloor$ and its children are $2i+1$ and $2i+2$ (Equation (14.1)); with the root at 1, the parent of k is $\lfloor k/2 \rfloor$ and its children are $2k$ and $2k+1$ (Equation (14.2)). The height is $\lfloor \lg n \rfloor$ (Equation (14.4)), and the leaves are the indices $\lfloor n/2 \rfloor$ to $n-1$ (Equation (14.5)).
- A **max-heap** has every node no larger than its parent (Equation (14.6)), so the root is the maximum and every root-to-leaf path is non-increasing; a **min-heap** is the mirror image (Equation (14.7)). Siblings, levels and the array as a whole are not ordered.
- **Sift-up** moves a key that is too large toward the root: at most $\lfloor \lg n \rfloor$ exchanges. **Sift-down** moves a key that is too small toward the leaves, always exchanging with the larger child: at most $2\lfloor \lg n \rfloor$ comparisons (Equation (14.8)).
- `insert` appends and sifts up; `extractMax` moves the last leaf to the root and sifts down; both $O(\lg n)$. `getMax` is $O(1)$. `getMin` in a max-heap scans the leaves and is $\Theta(n)$, and no method can do better, because any leaf can hold the minimum.
- `increaseKey` sifts up and `decreaseKey` sifts down, $O(\lg n)$ each, but the caller must know the item's index.
- `buildMaxHeap` sifts down the internal nodes from the last to the root, in $\Theta(n)$ time: fewer than n exchanges and $2n$ comparisons (Equation (14.9), Equation (14.10), Equation (14.11)). Building by n insertions costs $\Theta(n \lg n)$ in the worst case (Equation (14.12)), though only linear time on random keys.
- `isMaxHeap` checks every child against its parent in $O(n)$.
- **Heap sort** builds a max-heap and repeatedly moves the root to the end of the shrinking heap: $O(n \lg n)$ comparisons in every case (Equation (14.13)), $O(1)$ extra memory, **not stable** (**Table 14.5**).

- The **libraries**: `std::priority_queue` (max first, comparison means "served later", `pop` returns `void`), Java's `PriorityQueue` (min first), and Python's `heapq` (min first, with `heapify_max`, `heappush_max`, `heappop_max`, `heappushpop_max` and `heapreplace_max` since Python 3.14) (**Table 14.6**).
- **Applications**: the top k of a stream with a min-heap of size k, $O(n \lg k)$ (Equation (14.14)); the merge of k sorted lists, $O(N \lg k)$ (Equation (14.15)); and Dijkstra's shortest paths with lazy deletion, $O((|V| + |E|) \lg |V|)$ (Equation (14.16)).

14.19 Exercises

14.1 (a) In a 0-based heap, give the parent and the two children of index 13. (b) Give the same for the same node in the 1-based numbering. (c) In a heap of $n = 20$ keys, is index 13 (0-based) a leaf? Which indices are the leaves?

14.2 Which of the following arrays are max-heaps? For each one that is not, name a pair of indices that violates Equation (14.6). (a) 50, 30, 40, 10, 20, 35, 5; (b) 50, 40, 30, 45, 10; (c) 9, 9, 9; (d) 10, 9, 8, 7, 6, 5; (e) 20, 15, 18, 16, 14, 17.

14.3 Starting from an empty max-heap, insert 15, 40, 20, 35, 50, 10 and 45 in that order. Give the array after each insertion, and count the exchanges made by each sift-up.

14.4 On the heap that Exercise 14.3 produces, call `extractMax` three times. Give the value returned and the array after each call.

14.5 Run `BUILD-MAX-HEAP` by hand on the array 1, 2, 3, 4, 5, 6, 7. Give the array after each call of sift-down and the total number of exchanges, and compare that total with the bound $n - \lg(n + 1)$ of Section 14.10.2.

14.6 (a) What are the smallest and the largest number of keys in a heap of height h? (b) In a max-heap with distinct keys, at which indices can the second-largest key be? The third-largest? (c) Can the smallest key of a max-heap with distinct keys be at an internal node?

14.7 A *d-ary heap* is like a binary heap, except that every node has up to d children. (a) With the root at index 0, give the indices of the children and of the parent of index i. (b) What is the height of a d-ary heap of n keys? (c) How many comparisons do sift-up and sift-down make in the worst case? (d) Why might a 4-ary heap be preferred for Dijkstra's algorithm on a graph with many more edges than vertices?

14.8 (a) Sort 4, 10, 3, 5, 1 with heap sort by hand, giving the array after the build and after every iteration of phase 2. (b) Explain how to make heap sort stable, and what it costs.

14.9 Write (a) `bool isMinHeap(const vector<int>& a)`, and (b) `int kthLargest(const vector<int>& heap, int k)`, which returns the k-th largest key of a max-heap without modifying it, in $O(k \lg k)$ time. (Hint: the largest key is the root; the next candidate is always a child of a key already taken.)

14.10 Write `void removeAt(vector<int>& heap, int index)`, which removes the key at any index of a max-heap in $O(\lg n)$ time. Test it on a case where the moved key must rise and on one where it must sink.

14.11 A stream of numbers arrives one at a time. After each number, print the median of all numbers so far, in $O(\lg n)$ time per number. (Hint: keep the smaller half in a max-heap and the larger half in a min-heap.) Use the stream 5, 15, 1, 3, 8, 7, 9, 10.

14.12 The Profitina Laundry counter receives orders. Each has an identifier, a kind (express or regular), a pickup time and an arrival number. The washing machine must take express orders before all regular ones, then the earliest pickup time, then the earliest arrival. Write the queue with a library priority queue and print the washing order for: 101 regular, pickup 17:00, arrival 1; 102 express, 15:00, 2; 103 regular, 12:00, 3; 104 express, 11:30, 4; 105 regular, 12:00, 5; 106 express, 15:00, 6.

14.13 Prove that $\sum_{m=1}^{n} \lfloor \lg m \rfloor = (n+1)\lfloor \lg n \rfloor - 2^{\lfloor \lg n \rfloor + 1} + 2$, the closed form of Equation (14.12), and check it against the three "ascending" totals of **Listing 14.2**.

14.14 Rewrite `SIFT-UP` so that it moves the key into place with one final write instead of an exchange at every level (Section 14.5.3). How many array writes does it make when the key rises s levels, compared with **Pseudocode 14.1**?

14.15 (a) In the lazy Dijkstra algorithm of Section 13.10.3 and **Listing 14.11**, explain why skipping an entry whose vertex is already settled is correct. (b) How many entries can the priority queue hold at most, for a graph with $|V|$ vertices and $|E|$ edges? (c) What would go wrong if a max-heap were used by mistake?

14.20 Solutions to the Exercises

14.1 (a) By Equation (14.1): parent $\lfloor 12/2 \rfloor = 6$, children $2 \cdot 13 + 1 = 27$ and 28. (b) The same node has 1-based index 14; by Equation (14.2): parent $\lfloor 14/2 \rfloor = 7$, children 28 and 29. Every answer is one more than in (a), as it must be. (c) By Equation (14.5) the leaves of a 20-key heap are the indices $\lfloor 20/2 \rfloor = 10$ to 19, so index 13 is a leaf; indeed its left child, 27, would lie outside the array.

14.2 (a) Yes: 30 and 40 are at most 50; 10 and 20 are at most 30; 35 and 5 are at most 40. (b) No: index 3 holds 45, its parent index 1 holds 40, and $40 < 45$. (c) Yes: equal keys do not violate $\geq$. (d) Yes: an array in descending order is always a max-heap, since every parent comes earlier. (e) No: index 3 holds 16 and its parent, index 1, holds 15.

14.3 The arrays are: 15; 40, 15 (one exchange); 40, 15, 20 (none); 40, 35, 20, 15 (one: 35 passes 15); 50, 40, 20, 15, 35 (two: 50 is placed at index 4, passes 35 at index 1, then 40 at the root); 50, 40, 20, 15, 35, 10 (none); 50, 40, 45, 15, 35, 10, 20 (one: 45 passes 20). Five exchanges in all.

14.4 The first call returns 50: 20 moves to the root, sinks below 45, and the heap is 45, 40, 20, 15, 35, 10. The second returns 45: 10 moves to the root, sinks below 40 and then below 35, giving 40, 35, 20, 15, 10. The third returns 40: 10 moves to the root, sinks below 35 and then below 15, giving 35, 15, 20, 10.

14.5 The internal nodes are 2, 1 and 0. `siftDown(2)`: 3 is smaller than its children 6 and 7 and sinks below 7: 1, 2, 7, 4, 5, 6, 3. `siftDown(1)`: 2 sinks below the larger of 4 and 5: 1, 5, 7, 4, 2, 6, 3. `siftDown(0)`: 1 sinks below 7, then below 6: 7, 5, 6, 4, 2, 1, 3. Four exchanges in total, which equals $n - \lg(n+1) = 7 - 3 = 4$: an ascending array is the worst case for the build, and the bound of Section 14.10.2 is reached exactly.

14.6 (a) At least 2^h (the levels above the last are full, $2^h - 1$ keys, and the last has at least one) and at most $2^{h+1} - 1$ (all levels full), by Equation (14.3). (b) The second-largest key is larger than everything except the root, so its parent must be the root: it is at index 1 or 2. The third-largest has only the root and the second-largest above it, so it is a child of the root or a child of the

second-largest: any index from 1 to 6 (depth 1 or 2). (c) No; Section 14.8 proves that with distinct keys the minimum is only at a leaf.

14.7 (a) The children of i are $di + 1, di + 2, \ldots, di + d$, and the parent of $i \geq 1$ is $\lfloor (i - 1)/d \rfloor$; for $d = 2$ these are Equation (14.1). (b) Level j holds up to d^j keys, so the height is about $\log_d n$; exactly, it is the smallest h with $1 + d + \cdots + d^h \geq n$. (c) Sift-up compares with one parent per level: about $\log_d n$ comparisons. Sift-down must find the largest of up to d children per level, d comparisons per level (including the comparison with the key itself): about $d \log_d n$. (d) Dijkstra's algorithm performs one extraction per vertex but up to one insertion (or `decreaseKey`) per edge. Insertion is a sift-up, which becomes cheaper as d grows, while extraction becomes more expensive; with many more edges than vertices, the cheaper insertions win. A 4-ary heap also places the four children of a node side by side in memory, which suits the processor's cache.

14.8 (a) The build makes 10, 5, 3, 4, 1 (4 sinks below 10 and then below 5). Phase 2: exchange 10 and 1 and sift down: 5, 4, 3, 1 | 10. Exchange 5 and 1 and sift down: 4, 1, 3 | 5, 10. Exchange 4 and 3: 3, 1 | 4, 5, 10. Exchange 3 and 1: 1 | 3, 4, 5, 10. The result is 1, 3, 4, 5, 10. (b) Sort the pairs (key, original index) with the comparison "larger key, or equal key and larger index, comes later". No two pairs are then equal, and equal keys come out in order of their original index. The extra index costs $\Theta(n)$ memory, so the sort is no longer in place.

14.9 **Listing 14.12** contains both functions. `isMinHeap` is **Pseudocode 14.8** with the comparison reversed. `kthLargest` keeps a small max-heap of *candidates*, pairs (key, index), starting with the root. It extracts the best candidate $k - 1$ times, each time pushing that key's children as new candidates, and then the best remaining candidate is the answer. It is correct because every key's parent is larger, so the j-th largest key is always a child of one of the $j - 1$ keys already taken (or the root). The candidate heap holds at most k entries, so each of the k steps costs $O(\lg k)$: $O(k \lg k)$ in total, independent of n. On the heap 90, 80, 70, 65, 25, 40, 30, 10, 55 it returns 90, 80, 70, 65, 55 for $k = 1$ to 5.

Listing 14.12 `exercises.cpp`: solutions to Exercises 14.9 to 14.12.

```cpp
// exercises.cpp -- solutions to the programming exercises of Chapter 14.
#include <cstdio>
#include <functional>
#include <queue>
#include <string>
#include <tuple>
#include <utility>
#include <vector>
using namespace std;

void show(const char* label, const vector<int>& a) {
    printf("%s", label);
    for (size_t i = 0; i < a.size(); ++i) printf("%s%d", i ? " " : "", a[i]);
    printf("\n");
}

// Exercise 14.9 (a): every element is at least as large as its parent.
bool isMinHeap(const vector<int>& a) {
    for (size_t i = 1; i < a.size(); ++i)
        if (a[(i - 1) / 2] > a[i]) return false;
    return true;
}

// Exercise 14.9 (b): the k-th largest key of a max-heap in O(k lg k) time.
int kthLargest(const vector<int>& heap, int k) {
    priority_queue<pair<int, int>> cand;          // (key, index), largest first
    cand.push({heap[0], 0});
    for (int step = 1; step < k; ++step) {
        int i = cand.top().second;                // the step-th largest leaves
        cand.pop();
        for (int c : {2 * i + 1, 2 * i + 2})      // its children become candidates
            if (c < static_cast<int>(heap.size())) cand.push({heap[c], c});
    }
    return cand.top().first;
}
```

Listing 14.12 (Continued) *exercises.cpp: solutions to Exercises 14.9 to 14.12.*

```
// Exercise 14.10: remove the key at any index of a max-heap in O(lg n).
void removeAt(vector<int>& a, int index) {
    int n = static_cast<int>(a.size()) - 1;       // size after the removal
    a[index] = a[n];                              // the last leaf fills the hole
    a.pop_back();
    if (index == n) return;                       // we removed the last leaf
    int i = index;
    while (i > 0 && a[(i - 1) / 2] < a[i]) {      // it may have to rise ...
        swap(a[i], a[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
    while (true) {                                // ... or it may have to sink
        int largest = i, l = 2 * i + 1, r = 2 * i + 2;
        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;
    }
}

// Exercise 14.11: the median of a stream with two heaps.
void runningMedian(const vector<int>& stream) {
    priority_queue<int> low;                              // smaller half, max top
    priority_queue<int, vector<int>, greater<int>> high; // larger half, min top
    printf("median after each value:");
    for (int x : stream) {
        if (low.empty() || x <= low.top()) low.push(x); else high.push(x);
        if (low.size() > high.size() + 1) { high.push(low.top()); low.pop(); }
        else if (high.size() > low.size()) { low.push(high.top()); high.pop(); }
        double m = low.size() > high.size() ? low.top()
                                            : (low.top() + high.top()) / 2.0;
        printf(" %.1f", m);
    }
    printf("\n");
}

// Exercise 14.12: Profitina Laundry orders -- express first, then the earliest
// pickup time, then the earliest arrival.
struct Order {
    int id;
    bool express;
    int pickup;       // minutes after midnight
    int arrival;
};
```

Listing 14.12 (Continued) *exercises.cpp: solutions to Exercises 14.9 to 14.12.*

```
struct ServedLater {
    bool operator()(const Order& a, const Order& b) const {
        return make_tuple(!a.express, a.pickup, a.arrival) >
               make_tuple(!b.express, b.pickup, b.arrival);
    }
};

int main() {
    vector<int> h = {90, 80, 70, 65, 25, 40, 30, 10, 55};
    show("heap: ", h);
    printf("isMinHeap(heap) = %s, isMinHeap({3, 5, 4, 9, 6}) = %s\n",
           isMinHeap(h) ? "true" : "false",
           isMinHeap({3, 5, 4, 9, 6}) ? "true" : "false");
    printf("k-th largest for k = 1..5:");
    for (int k = 1; k <= 5; ++k) printf(" %d", kthLargest(h, k));
    printf("\n\n");

    vector<int> a = {90, 20, 80, 10, 15, 70, 75};
    show("heap:          ", a);
    removeAt(a, 3);
    show("removeAt(3) -> ", a);
    removeAt(a, 0);
    show("removeAt(0) -> ", a);
    printf("\n");

    runningMedian({5, 15, 1, 3, 8, 7, 9, 10});
    printf("\n");

    priority_queue<Order, vector<Order>, ServedLater> laundry;
    laundry.push({101, false, 17 * 60, 1});
    laundry.push({102, true, 15 * 60, 2});
    laundry.push({103, false, 12 * 60, 3});
    laundry.push({104, true, 11 * 60 + 30, 4});
    laundry.push({105, false, 12 * 60, 5});
    laundry.push({106, true, 15 * 60, 6});
    printf("Profitina Laundry, washing order:\n");
    while (!laundry.empty()) {
        Order o = laundry.top();
        laundry.pop();
        printf("  order %d  %-7s  pickup %02d:%02d  arrived #%d\n", o.id,
               o.express ? "express" : "regular", o.pickup / 60, o.pickup % 60,
               o.arrival);
    }
    return 0;
}
```

```
heap: 90 80 70 65 25 40 30 10 55
isMinHeap(heap) = false, isMinHeap({3, 5, 4, 9, 6}) = true
k-th largest for k = 1..5: 90 80 70 65 55

heap:          90 20 80 10 15 70 75
removeAt(3) -> 90 75 80 20 15 70
removeAt(0) -> 80 75 70 20 15

median after each value: 5.0 10.0 5.0 4.0 5.0 6.0 7.0 7.5

Profitina Laundry, washing order:
  order 104  express  pickup 11:30  arrived #4
  order 102  express  pickup 15:00  arrived #2
  order 106  express  pickup 15:00  arrived #6
  order 103  regular  pickup 12:00  arrived #3
  order 105  regular  pickup 12:00  arrived #5
  order 101  regular  pickup 17:00  arrived #1
```

14.10 `removeAt` in **Listing 14.12** moves the last leaf into the hole, as `extractMax` does for the root. The moved key comes from elsewhere in the tree, so it may be too small *or too large* for its new place: the function first lets it rise, then lets it sink (at most one of the two loops moves anything). In the heap 90, 20, 80, 10, 15, 70, 75, `removeAt(3)` removes 10; the last key 75 lands at index 3 under 20 and must *rise*, giving 90, 75, 80, 20, 15, 70. `removeAt(0)` removes 90; the last key 70 lands at the root and must *sink* below 80, giving 80, 75, 70, 20, 15. Both loops follow one path, so the time is $O(\lg n)$. The special case `index == n` (the removed key was the last leaf) needs no repair.

14.11 `runningMedian` keeps the smaller half of the numbers in the max-heap `low` and the larger half in the min-heap `high`, with `low` holding the same number of keys as `high` or one more. Each new number goes into `low` if it is at most `low`'s maximum and into `high` otherwise; then one key moves across if the sizes differ by more than allowed. The median is `low`'s top when the count is odd, and the average of the two tops when it is even. Each step is a constant number of heap operations, $O(\lg n)$. For 5, 15, 1, 3, 8, 7, 9, 10 the medians are 5, 10, 5, 4, 5, 6, 7 and 7.5: for example, after 1, 3, 5, 15 the halves are $\{1, 3\}$ and $\{5, 15\}$ and the median is $(3 + 5)/2 = 4$.

14.12 The key of an order is the triple (not express, pickup time, arrival). Tuples compare from the left, and `false` is less than `true`, so an express order (whose first component is `false`) comes before every regular one; then the earlier pickup; then the earlier arrival. In C++ the comparison `ServedLater` returns `true` when `a`'s triple is *greater*, the "served later" convention of Section 14.14.1; in Python the triple is pushed directly into a `heapq` list, and in Java a `Comparator` is chained with `thenComparingInt`. The output takes the express orders 104, 102, 106 first (11:30, then two at 15:00 in arrival order) and then the regular orders 103, 105 (both 12:00, in arrival order) and 101.

14.13 Group the terms by the value of $\lfloor \lg m \rfloor$. For $j < L = \lfloor \lg n \rfloor$, the 2^j values $m = 2^j, \ldots, 2^{j+1} - 1$ each contribute j; the values $m = 2^L, \ldots, n$, of which there are $n - 2^L + 1$, each contribute L. So the sum is $\sum_{j=0}^{L-1} j\,2^j + L(n - 2^L + 1)$. The first part is $(L - 2)2^L + 2$ (a finite version of the trick of Equation (14.10): write $U = \sum j2^j$ and subtract it from $2U$). Adding, $(L - 2)2^L + 2 + Ln - L2^L + L = (n + 1)L - 2^{L+1} + 2$. The term $m = 1$ contributes 0, so the sum from $m = 2$ is the same. Check: $n = 1000$, $L = 9$: $9009 - 1024 + 2 = 7987$; $n = 10{,}000$, $L = 13$: $130{,}013 - 16{,}384 + 2 = 113{,}631$; $n = 100{,}000$, $L = 16$: $1{,}600{,}016 - 131{,}072 + 2 = 1{,}468{,}946$. All three agree with the program.

14.14 Save the key, move smaller parents down into the hole, and write the key once at the end:

Pseudocode 14.12 `SIFT-UP-HOLE`: the key rises through a moving hole and is written once at the end.

```
SIFT-UP-HOLE(A, i)
 1  key = A[i]
 2  while i > 0 and A[parent(i)] < key
 3      A[i] = A[parent(i)]          // the parent moves down into the hole
 4      i = parent(i)
 5  A[i] = key
```

If the key rises s levels, **Pseudocode 14.1** makes s exchanges, which is $2s$ writes (three assignments per exchange through a temporary, of which two go to the array), while **Pseudocode 14.12** makes $s + 1$ writes. The comparisons are the same. The same technique applies to sift-down.

14.15 (a) Dijkstra's algorithm settles vertices in order of non-decreasing distance, and a settled vertex's distance is final (the correctness argument of Section 13.10.5, which needs non-negative weights [4]). An entry (d, u) popped after u is settled carries a distance at least as large as the true shortest distance, so it cannot improve anything and may be discarded. (b) Each relaxation that succeeds pushes one entry, and each directed edge is relaxed at most once, when its tail is settled; with the initial entry, the queue holds at most $|E| + 1$ entries (for an undirected graph, where each road appears in both adjacency lists, at most $2|E| + 1$). In **Listing 14.11**, 9 roads led to 10 pushes. (c) A max-heap would settle the vertex with the *largest* tentative distance first; distances would be declared final before shorter routes had been examined, and the results would be wrong. In **Figure 14.13**, Engineering would be settled at 7 before the route through Science (5) was found.

References

[1] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, Jun. 1964.

[2] R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, Dec. 1964.

[3] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, sec. 5.2.3.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 6.

[5] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011, sec. 2.4.

[6] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte, "Min-max heaps and generalized priority queues," *Communications of the ACM*, vol. 29, no. 10, pp. 996–1000, Oct. 1986.

[7] T. Porter and I. Simon, "Random insertion into a priority queue structure," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 3, pp. 292–298, Sep. 1975.

[8] D. R. Musser, "Introspective sorting and selection algorithms," *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, Aug. 1997.

[9] cppreference.com, "std::priority_queue." [Online]. Available: https://en.cppreference.com/w/cpp/container/priority_queue. [Accessed: 24-Sep-2026].

[10] cppreference.com, "std::make_heap," "std::push_heap," "std::pop_heap," "std::sort_heap," "std::is_heap." [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/make_heap. [Accessed: 24-Sep-2026].

[11] Oracle, "Class PriorityQueue," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/PriorityQueue.html. [Accessed: 24-Sep-2026].

[12] Python Software Foundation, "heapq — Heap queue algorithm," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/heapq.html. [Accessed: 24-Sep-2026].

[13] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[14] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[15] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Chapter 15
Hashing and Hash Tables

By the end of this chapter you will be able to explain why a hash table can find a key in a constant number of steps on average when every comparison-based search needs at least a logarithmic number; design and judge hash functions for integers and strings (the division method, the multiplication method, the polynomial hash computed by Horner's rule and its rolling update), and measure the avalanche property that separates a good mixer from a poor one; compute how soon random keys collide with the birthday bound; build a hash table with separate chaining out of the linked nodes of Chapter 7, and another with open addressing by linear probing, quadratic probing and double hashing, including deletion with tombstones; predict and measure how the load factor governs the number of probes, and why rehashing by doubling costs only a constant per insertion; recognize the worst case, the hash-flooding attack that provokes it, and the randomized hashing that defends against it; choose correctly between `std::map` and `std::unordered_map`, and explain what Java's `HashMap` and Python's `dict` do inside; and use a hash table to count words, detect duplicates, solve two-sum and build a cache. Every table is drawn, every formula is derived and then checked against a measurement, and every program is complete, compiled or run in C++, Java and Python, and memory-checked.

15.1 Where We Left Off

This book has been answering one question again and again: *given a key, is it here, and if so, where?* Chapter 2 answered it by sequential search, which looks at the elements one by one and needs up to n probes for n elements, and by binary search, which needs about $\lg n$ probes but only works on sorted data. Section 2.12.2 then proved something stronger and slightly disappointing: no algorithm that learns about the data only by *comparing* keys can beat $\lceil \lg(n+1) \rceil$ probes in the worst case. The same section left an escape clause. The proof applies only to algorithms that compare. An algorithm that does *arithmetic* on the key itself is not bound by it.

Chapter 11 met the same limit from another side. A binary search tree keeps its keys in order and finds one in $O(h)$ steps, where h is the height; the balanced trees of Section 11.9 keep h in $O(\lg n)$. That is excellent, and it comes with a bonus the array did not give us so cheaply: insertions and deletions also cost $O(\lg n)$. But it is still a logarithm, because a search tree is a comparison machine.

This chapter takes the escape clause. A *hash table* computes, directly from the key, the position where the key should be stored, and goes there. When the computation spreads the keys well, the position is usually right on the first try, and a search takes a constant number of steps on average, however many keys are stored. We have already met the idea twice without its name. In Section 4.4.3 an array of 256 counters, indexed by the character itself, found the first repeated character in one pass, because "the value you are looking for tells you where to look". In Section 8.7.3 a set of values seen so far removed duplicates from a list in linear time. In Section 9.9 a hash map pointed from each title to its node in the "recently borrowed" list. All three leaned on a structure this book had not built yet. Now we build it, twice, measure it, break it on purpose, and then read what the standard libraries of C++, Java and Python actually do.

The price must be said at once, because it shapes every decision later in the chapter. A hash table gives up *order*. It cannot list its keys from smallest to largest without sorting them first, cannot answer "which codes lie between B1003 and B1006?" without looking at every key, and cannot find the next key after a given one. It also gives up the *guarantee*: the constant time is an average, and an unlucky or malicious set of keys can make every operation linear. Section 15.8 shows how bad that gets and how real systems defend themselves. Chapter 16 works problems of exactly this kind, mixing the graphs of Chapter 13, the heaps of Chapter 14 and the hash tables of this chapter.

15.2 From $O(n)$ to $O(\lg n)$ to $O(1)$: the Idea of Hashing

15.2.1 The librarian who never searches

Picture the returns desk at Pustaka Ceria at closing time. A reader asks whether *Filosofi Kopi* has come back today. The new volunteer looks through the pile of returned books one by one; with forty books on the pile, that can take forty looks. The student who has worked the desk for a year keeps the returned books sorted by title on a shelf and opens the shelf in the middle, then in the middle of the correct half, and so on; forty books take at most six looks. The head librarian does something different. Under the desk stand twenty-six trays labeled A to Z. Every returned book goes into the tray of the first letter of its title. Asked about *Filosofi Kopi*, she does not search at all: she reaches into tray F, where two or three books lie, and answers at once.

The three people are the three rows of this book's progression. The volunteer is sequential search, $O(n)$. The student is binary search, $O(\lg n)$. The head librarian is hashing: she *computes* where the book must be, from the book's own title, and looks only there. Her rule "first letter of the title" is a *hash function*, the trays are *buckets*, and the fact that *Fisika Dasar* and *Filosofi Kopi* share tray F is a *collision*. Her method stays fast only while no tray grows large: if the library received a thousand books whose titles all begin with "Sejarah", tray S would become a pile to search, and she would be back to being the volunteer. Everything in this chapter is a refinement of these trays: a better rule than the first letter, a sensible number of trays, a plan for books that land in the same tray, and a way to add trays when the pile grows.

15.2.2 Direct addressing

The simplest possible version of the trays needs no rule at all. Suppose every key is a small non-negative integer, say the member numbers 0 to 999 of the library's readers. Keep an array `T` of 1000 cells and store the record of member k in `T[k]`. Searching, inserting and deleting are then one array access each, $O(1)$ in the worst case, with no comparison of keys whatsoever. This is *direct addressing*, and **Figure 15.1**(a) draws it. The 256 counters of Section 4.4.3 were exactly a direct-address table whose keys were the 256 possible byte values.

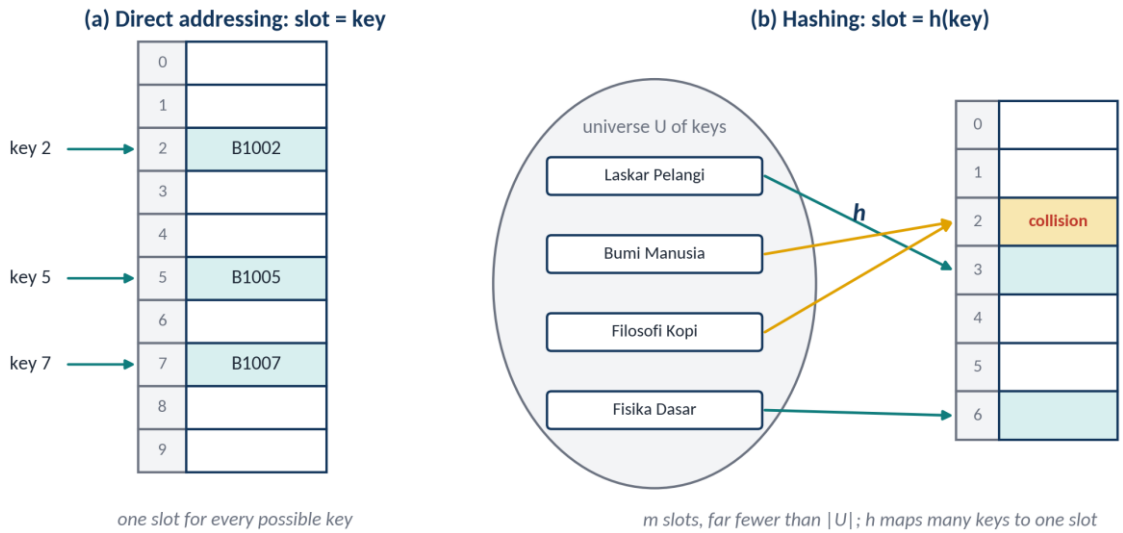


Figure 15.1 (a) A direct-address table has one slot for every possible key, so the key itself is the index. (b) A hash table has far fewer slots than possible keys; the hash function h maps each key to a slot, and two different keys may be sent to the same slot, a collision.

Direct addressing fails as soon as the set U of *possible* keys, the *universe*, is much larger than the set of keys actually stored. The universe of book titles is every string of letters, which is infinite for practical purposes. The universe of 64-bit integers has $2^{64} \approx 1.8 \times 10^{19}$ members; an array with one byte per possible key would need 18 exabytes. Even the modest universe of Indonesian national identity numbers (sixteen digits) would need 10^{16} cells to hold perhaps three hundred million people. The array would be almost entirely empty.

15.2.3 Hashing: the vocabulary

Hashing keeps the fast array access of direct addressing and throws away its huge array. We choose a table of m slots, with m in proportion to the number of keys we expect to store, and a function that squeezes the universe into it.

> **DEFINITION.** A *hash function* for a table of m slots is a function $h: U \to \{0,1,\ldots,m-1\}$. The number $h(k)$ is the *hash value* (or *home slot*) of key k, and we say that k *hashes to* slot $h(k)$. A *hash table* is an array $T[0..m-1]$ in which a key k is stored at, or starting from, slot $h(k)$. Two distinct keys $k_1 \neq k_2$ with $h(k_1) = h(k_2)$ *collide*. The *load factor* of a table holding n keys in m slots is $\alpha = n/m$.

A slot is also called a *bucket*, especially when it can hold several keys. **Figure 15.1**(b) shows four titles hashed into seven slots: *Bumi Manusia* and *Filosofi Kopi* collide in slot 2. The rest of the chapter answers three questions that the definition leaves open. What is a good h (Section 15.3)? How often do keys collide (Section 15.4)? And what do we do when they do (Sections 15.5 and 15.6)?

The idea is almost as old as random-access storage itself. Knuth traces it to an internal IBM memorandum of January 1953 by Hans Peter Luhn, which already proposed chaining colliding records together, and to the IBM 701 assembler of 1954 by Gene Amdahl, Elaine McGraw (later Boehme), Nathaniel Rochester and Arthur Samuel, which used what we now call linear probing [1]. The first published description, with its analysis for disk files, is W. Wesley Peterson's paper of 1957 on "addressing for random-access storage" [2], and the word *hashing*, from the everyday sense of chopping something into small pieces and mixing it up, first appeared in print in Robert Morris's survey of 1968 [1]. Seventy years later hash tables sit inside every compiler (the table of identifiers), every database (hash joins and hash indexes), every web browser (the cache of

recently seen host names that Chapter 1 described), and the core of the Python language, whose objects, modules and classes are all dictionaries.

15.2.4 The progression in one table

Table 15.1 places hashing next to the search structures of the earlier chapters. The column that matters most for choosing is the last one: what each structure costs you besides time.

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 like playing "higher or lower": each answer halves what is left, so you need about $\lg n$ questions. Hashing is like knowing the house number: you walk straight to the door. You only knock on a few wrong doors when two people were given the same number.

15.3 Hash Functions

15.3.1 What a good hash function must do

A hash function has one job: to turn a key into a slot number so that the keys we actually store spread evenly over the m slots. Four properties follow from that job.

1. **Deterministic.** The same key must always give the same slot, within one table's lifetime. A function that consulted the clock would store a key in one slot and look for it in another.
2. **Fast.** The whole point is to beat $\lg n$ comparisons; a hash function that costs more than a binary search has defeated its own purpose. Good hash functions cost a few machine instructions per byte of key.
3. **Uniform.** Each slot should receive about the same share of the keys that are *actually* stored, which are rarely random: book codes are consecutive, prices are round thousands, identifiers share prefixes.
4. **Uses the whole key.** Every part of the key should influence the slot. The head librarian's "first letter" rule ignores everything after the first letter, which is exactly why all the *Sejarah* books pile up in tray S.

The third property is usually stated as an idealized assumption, the yardstick against which real functions are measured and under which all the averages of this chapter are derived.

DEFINITION. Under the assumption of *simple uniform hashing*, each key is equally likely to hash to any of the m slots, independently of where the other keys hash:

$$\Pr[h(k) = j] = \frac{1}{m} \quad \text{for every slot } j \in \{0,1,\ldots,m-1\}. \tag{15.1}$$

No fixed function can satisfy Equation (15.1) for every possible set of keys, because an adversary who knows h can always pick keys that collide (Section 15.8 does exactly that). The assumption is still the right one to design for: a good function behaves *as if* Equation (15.1) held for the keys that real programs store, and Section 15.7 checks that our measurements agree with the averages it predicts.

15.3.2 The division method

The oldest and simplest hash function for integer keys takes the remainder after dividing by the table size:

$$h(k) = k \bmod m. \tag{15.2}$$

It costs one division. For $m = 11$ and the key 45000 it gives $45000 = 4090 \cdot 11 + 10$, so $h(45000) = 10$. The whole difficulty lies in choosing m, because the remainder depends only on part of the key's information, and which part depends on m.

Suppose Pustaka Ceria hashes the prices of eight books, all round thousands of rupiah: 45000, 60000, 75000, 85000, 90000, 120000, 55000, 70000. The first part of **Listing 15.1**'s output, shown a little later in this section, hashes them with three table sizes:

```
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
```

With $m = 10$ every price lands in slot 0. The remainder modulo 10 is the last decimal digit, and every price ends in 0; the function sees only the part of the key that never changes. With $m = 16 = 2^4$ the remainder is the last four *binary* digits, and a multiple of $1000 = 2^3 \cdot 125$ always ends in three binary zeros, so only slots 0 and 8 are possible. With the prime $m = 11$ all eight prices land in different slots. The general rule follows from this example.

- **Avoid $m = 2^p$ with the division method.** Then $k \bmod m$ is just the lowest p bits of k, and keys that differ only in their higher bits all collide.
- **Avoid $m = 10^p$** for decimal data, for the same reason.
- **Prefer a prime m not too close to a power of 2.** A prime shares no factor with the regular strides that real keys have (multiples of 1000, of 8, of 12), so each stride walks through all the slots before repeating.

The last point deserves one line of proof. If the keys are $k_0, k_0 + d, k_0 + 2d, \ldots$ with a common difference d, their slots are $k_0 + id \bmod m$. Two of them, i and j with $0 \le i < j < m$, coincide exactly when m divides $(j - i)d$. If m is prime and does not divide d, it must divide $j - i$, which is impossible for $0 < j - i < m$. So the first m keys of the stride fill all m slots. If instead $g = \gcd(d, m) > 1$, the stride only ever reaches m/g slots, as the multiples of 1000 in a table of 16 reached only $16/8 = 2$ of them.

15.3.3 The multiplication method

The division method makes us pick m carefully. The *multiplication method* works for any m, in particular for powers of 2, which are convenient because the table can double its size. Choose a constant A with $0 < A < 1$, multiply the key by it, keep the fractional part and scale it up to the table:

$$h(k) = \lfloor m \, (kA \bmod 1) \rfloor, \tag{15.3}$$

where $kA \bmod 1 = kA - \lfloor kA \rfloor$ is the fractional part of kA. Knuth recommends the golden-ratio constant $A = (\sqrt{5}-1)/2 \approx 0.6180339887$ [1], which spreads consecutive keys particularly evenly: the fractional parts of $A, 2A, 3A, \ldots$ always fall into the largest remaining gap of the interval $[0,1)$.

For the key $k = 45000$ and $m = 16$: $kA = 45000 \times 0.6180339887\ldots = 27811.529\ldots$, the fractional part is $0.529\ldots$, and $\lfloor 16 \times 0.529\ldots \rfloor = \lfloor 8.47\ldots \rfloor = 8$.

On a real machine we avoid floating point. With a word of w bits and $m = 2^p$, let $s = \lfloor A \cdot 2^w \rfloor$. The product $k \cdot s$ computed in w-bit unsigned arithmetic (which silently discards everything above bit $w-1$) is the fractional part of kA scaled by 2^w, and its top p bits are the slot:

$$h(k) = (k \cdot s \bmod 2^w) \gg (w - p), \qquad s = \lfloor 2^w A \rfloor. \tag{15.4}$$

For $w = 64$, $s = 0\text{x}9\text{E}3779\text{B}97\text{F}4\text{A}7\text{C}15$, and the method is often called *Fibonacci hashing*, because A is the limit of ratios of consecutive Fibonacci numbers. Equation (15.4) is one multiplication and one shift, faster than a division, and it is what the function `hashMultInt` of **Listing 15.1** computes. The same eight prices that collapsed into two slots under $k \bmod 16$ spread over six of the sixteen slots, and the floating-point and integer forms agree on every one (part 2 of the output).

15.3.4 Hashing strings

Most real keys are strings: titles, names, codes, words, URLs. A string hash function must first turn the characters into one integer. The obvious idea, adding up the character codes, is a trap:

$$h_{\text{sum}}(s) = \sum_{i=0}^{L-1} s_i,$$

where $s_0, s_1, \ldots, s_{L-1}$ are the codes of the L characters. Addition does not care about order, so every *anagram* collides. In Indonesian *kasur* (mattress) and *rusak* (broken) both sum to 550, and *kita* and *kiat* both to 425. Worse, sums of short strings are small numbers crowded into a narrow range: the 2000 book codes B0000 to B1999 have sums between only 258 (for B0000) and 286 (for B1999), so in a table of 101 buckets at most 29 buckets can ever be used; the output below shows exactly 72 of the 101 buckets empty and one of them holding 150 codes.

The cure is to let each character's *position* matter, by treating the string as a number written in base B:

$$h(s) = s_0 B^{L-1} + s_1 B^{L-2} + \cdots + s_{L-2} B + s_{L-1} = \sum_{i=0}^{L-1} s_i \, B^{L-1-i}. \tag{15.5}$$

This is the *polynomial hash*. Java's `String.hashCode` uses exactly Equation (15.5) with $B = 31$ and 32-bit arithmetic [3]; our programs use $B = 31$ with 64-bit arithmetic. A small odd base works well because multiplying by it mixes each new character into all the bits above it, and $31 = 2^5 - 1$ is also cheap (a shift and a subtraction). Now order matters: *kasur* hashes to 101820730 and *rusak* to 108880570.

Computing Equation (15.5) power by power would waste multiplications. *Horner's rule*, the standard way to evaluate a polynomial with one multiplication per coefficient, factors it instead:

$$h(s) = (\cdots((s_0 \cdot B + s_1) \cdot B + s_2) \cdot B + \cdots) \cdot B + s_{L-1},$$

one multiplication and one addition per character, as **Pseudocode 15.1** shows.

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
```

Trace it on the code "B1001", whose characters have codes 66, 49, 48, 48, 49:

```
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
```

The final value, 62459810, is also exactly what Java's `"B1001".hashCode()` returns (Section 15.10.1 prints it), because for a string this short no reduction modulo 2^{32} has happened yet. The reduction modulo M in line 3 of **Pseudocode 15.1** keeps the numbers inside a machine word; with $M = 2^{64}$ and unsigned arithmetic in C++ it costs nothing, since the hardware discards the overflow bits by itself. In Java the `long` type wraps around the same way; in Python, whose integers never overflow, we write `& MASK` with MASK $= 2^{64} - 1$ to get the same bits. The table index is then $h(s)$ mod m as in Equation (15.2).

15.3.5 The rolling update

The polynomial hash has a second talent. Suppose we need the hash of *every* window of W consecutive characters of a long text, for instance to find where a phrase occurs. Recomputing each window from scratch costs W steps per window. But neighboring windows share $W - 1$ characters, and Equation (15.5) lets us move from the window starting at $i - 1$ to the window starting at i in constant time: remove the contribution of the character leaving on the left, shift everything one place up, and add the character entering on the right.

$$h_i = (h_{i-1} - s_{i-1}B^{W-1}) \cdot B + s_{i+W-1}. \tag{15.6}$$

This *rolling hash* is the heart of the Rabin-Karp string-matching algorithm and of the "content-defined chunking" that backup programs use to find repeated blocks in files. Part 4 of **Listing 15.1** moves a window of four letters along "pustakaceria" with Equation (15.6) and checks each result against a fresh computation: all nine windows agree. The subtraction may go below zero in the middle of the computation; with unsigned arithmetic modulo 2^{64} that is harmless, because the end result is still correct modulo 2^{64}.

15.3.6 Avalanche: how well does a function mix?

Uniformity is about sets of keys. There is a sharper test that looks at single bits. A hash function has the *avalanche property* if flipping any one bit of the input flips each output bit with probability $1/2$, so that on average half of the output bits change. With 64-bit outputs, the ideal is 32 flipped bits for every input bit. Why care? Because tables take only a few bits of the hash: a table of 2^{10} slots uses ten. If some input bits cannot reach those ten, keys that differ only in those input bits always collide.

Part 5 of **Listing 15.1** measures this for four integer functions. For 2000 random 64-bit keys it flips each of the 64 input bits in turn and counts how many output bits change. It reports the average over all keys and bits, and the *weakest* input bit, the one whose flip changes the fewest output bits on average:

```
function    average  weakest input   bit
identity       1.00           1.00     0
k*31+7         3.67           1.00    63
Fibonacci     16.02           1.00    63
SplitMix64    31.97          31.80    25
```

The identity function, $h(k) = k$, is what GCC's `std::hash<int>` actually computes (Section 15.9 prints `std::hash<int>{}(42) = 42`): flipping a bit flips exactly that bit and nothing else. Multiplying by 31 carries each change only upward, so the top bit, bit 63, still affects nothing but itself. Fibonacci multiplication is much better on average, but it has the same blind spot: a change in a high input bit can only move bits above it. This is why Equation (15.4) takes the *top* bits of the product, never the bottom ones. The *SplitMix64 finalizer*, three xor-shifts interleaved with two multiplications by odd constants, is essentially perfect: every input bit changes about 32 output bits. Such a *mixer* is cheap (about five instructions) and turns even the identity into a good hash, which is why many modern hash-table libraries finish every hash value with a mixing step of this kind.

15.3.7 The program

Listing 15.1 contains all the functions of this section and the six experiments quoted in it. Read `hashPoly` next to **Pseudocode 15.1**, and the rolling loop next to Equation (15.6).

Listing 15.1 `hash_functions.cpp`: the division and multiplication methods, the sum and polynomial string hashes, the rolling update and the avalanche measurement.

```cpp
// hash_functions.cpp -- the division, multiplication and string hash functions
// of Section 15.3, the rolling update, and a measurement of the avalanche effect.
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <string>
#include <vector>

typedef std::uint64_t u64;

// Division method: h(k) = k mod m.
u64 hashDivision(u64 k, u64 m) { return k % m; }

// Multiplication method, textbook form: h(k) = floor(m * frac(k * A)).
u64 hashMultFloat(u64 k, u64 m) {
    const double A = (std::sqrt(5.0) - 1.0) / 2.0;   // 0.6180339887...
    double prod = (double)k * A;
    return (u64)std::floor((double)m * (prod - std::floor(prod)));
}

// 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 (Continued) *hash_functions.cpp: the division and multiplication methods, the sum and polynomial string hashes, the rolling update and the avalanche measurement.*

```
// Four integer-to-integer functions whose avalanche we measure.
u64 fIdentity(u64 k) { return k; }
u64 fTimes31(u64 k) { return k * 31 + 7; }
u64 fFibonacci(u64 k) { return k * 0x9E3779B97F4A7C15ULL; }
u64 fMix(u64 k) {                                       // the SplitMix64 finalizer
    k = (k ^ (k >> 30)) * 0xBF58476D1CE4E5B9ULL;
    k = (k ^ (k >> 27)) * 0x94D049BB133111EBULL;
    return k ^ (k >> 31);
}

u64 rngState = 2026;                                    // a fixed seed: same run
u64 nextRandom() {                                      // every time (SplitMix64)
    rngState += 0x9E3779B97F4A7C15ULL;
    return fMix(rngState);
}

int popcount64(u64 x) {
    int c = 0;
    while (x != 0) { x &= x - 1; ++c; }
    return c;
}

void avalanche(const char *name, u64 (*f)(u64)) {
    const int KEYS = 2000;
    long long total = 0;
    int worstBit = 0;
    double worstAvg = 64.0;
    std::vector<long long> perBit(64, 0);
    rngState = 2026;
    for (int t = 0; t < KEYS; ++t) {
        u64 k = nextRandom(), hk = f(k);
        for (int b = 0; b < 64; ++b) {
            int flips = popcount64(hk ^ f(k ^ (1ULL << b)));
            perBit[b] += flips;
            total += flips;
        }
    }
    for (int b = 0; b < 64; ++b) {
        double avg = (double)perBit[b] / KEYS;
        if (avg < worstAvg) { worstAvg = avg; worstBit = b; }
    }
    std::printf("%-14s %7.2f %14.2f %5d\n", name, (double)total / (64.0 * KEYS),
                worstAvg, worstBit);
}
```

Listing 15.1 (Continued) *hash_functions.cpp: the division and multiplication methods, the sum and polynomial string hashes, the rolling update and the avalanche measurement.*

```
int main() {
    std::printf("1. Division method: book prices (rupiah) into m buckets\n");
    const u64 price[8] = {45000, 60000, 75000, 85000, 90000, 120000, 55000, 70000};
    const u64 ms[3] = {10, 16, 11};
    for (u64 m : ms) {
        std::printf("   m = %2llu:", (unsigned long long)m);
        for (u64 p : price)
            std::printf(" %2llu", (unsigned long long)hashDivision(p, m));
        std::printf("\n");
    }

    std::printf("\n2. Multiplication method, m = 16 (p = 4)\n");
    std::printf("   price    float  integer\n");
    for (u64 p : price)
        std::printf("   %6llu %6llu %8llu\n", (unsigned long long)p,
                    (unsigned long long)hashMultFloat(p, 16),
                    (unsigned long long)hashMultInt(p, 4));

    std::printf("\n3. Strings: sum of codes versus polynomial (base 31)\n");
    const char *words[6] = {"kasur", "rusak", "kita", "kiat", "B1001", "B1010"};
    for (const char *w : words)
        std::printf("   %-6s sum = %4llu   poly = %llu\n", w,
                    (unsigned long long)hashSum(w),
                    (unsigned long long)hashPoly(w));

    std::printf("\n   Horner's rule for \"B1001\":\n");
    u64 h = 0;
    for (char c : std::string("B1001")) {
        u64 before = h;
        h = h * 31 + (unsigned char)c;
        std::printf("   h = %8llu * 31 + %2d ('%c') = %llu\n",
                    (unsigned long long)before, c, c, (unsigned long long)h);
    }

    std::printf("\n4. Rolling hash of every 4-letter window of \"pustakaceria\"\n");
    const std::string text = "pustakaceria";
    const int W = 4;
    u64 pw = 1;                                          // 31^(W-1)
    for (int i = 1; i < W; ++i) pw *= 31;
    u64 roll = hashPoly(text.substr(0, W));
    int agree = 0, windows = 0;
    for (std::size_t i = 0; i + W <= text.size(); ++i) {
        if (i > 0)                                       // drop the old first letter,
            roll = (roll - (unsigned char)text[i - 1] * pw) * 31
                   + (unsigned char)text[i + W - 1];     // shift, add the new last
        u64 direct = hashPoly(text.substr(i, W));
```

Listing 15.1 (Continued) *hash_functions.cpp: the division and multiplication methods, the sum and polynomial string hashes, the rolling update and the avalanche measurement.*

```
        ++windows;
        if (roll == direct) ++agree;
        std::printf("   %s %9llu\n", text.substr(i, W).c_str(),
                    (unsigned long long)roll);
    }
    std::printf("   rolling equals direct in %d of %d windows\n", agree, windows);

    std::printf("\n5. Avalanche: output bits that flip when one input bit flips\n");
    std::printf("   (2000 random 64-bit keys x 64 input bits; ideal = 32.00)\n");
    std::printf("%-14s %7s %14s %5s\n", "   function", "average",
                "weakest input", "bit");
    avalanche("   identity", fIdentity);
    avalanche("   k*31+7", fTimes31);
    avalanche("   Fibonacci", fFibonacci);
    avalanche("   SplitMix64", fMix);

    std::printf("\n6. Codes B0000..B1999 into m = 101 buckets\n");
    std::printf("   hash    empty  largest\n");
    for (int which = 0; which < 2; ++which) {
        std::vector<int> count(101, 0);
        for (int i = 0; i < 2000; ++i) {
            char code[16];
            std::snprintf(code, sizeof code, "B%04d", i);
            u64 v = which == 0 ? hashSum(code) : hashPoly(code);
            ++count[v % 101];
        }
        int empty = 0, largest = 0;
        for (int c : count) {
            if (c == 0) ++empty;
            if (c > largest) largest = c;
        }
        std::printf("   %-6s %6d %8d\n", which == 0 ? "sum" : "poly", empty,
                    largest);
    }
    return 0;
}
```

Its output, identical for the Java and Python versions in the chapter's code folder:

```
1. Division method: book prices (rupiah) into m buckets
   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

2. Multiplication method, m = 16 (p = 4)
   price    float  integer
    45000      8        8
    60000      0        0
    75000      8        8
    85000     14       14
    90000      0        0
   120000      1        1
    55000     13       13
    70000      6        6

3. Strings: sum of codes versus polynomial (base 31)
   kasur  sum =  550   poly = 101820730
   rusak  sum =  550   poly = 108880570
   kita   sum =  425   poly = 3292235
   kiat   sum =  425   poly = 3291665
   B1001  sum =  260   poly = 62459810
   B1010  sum =  260   poly = 62459840

   Horner's rule for "B1001":
   h =        0 * 31 + 66 ('B') = 66
   h =       66 * 31 + 49 ('1') = 2095
   h =     2095 * 31 + 48 ('0') = 64993
   h =    64993 * 31 + 48 ('0') = 2014831
   h =  2014831 * 31 + 49 ('1') = 62459810

4. Rolling hash of every 4-letter window of "pustakaceria"
   pust   3452710
   usta   3599755
   stak   3540555
   taka   3552387
   akac   2995660
   kace   3284024
   acer   2988111
   ceri   3050009
   eria   3121797
   rolling equals direct in 9 of 9 windows

5. Avalanche: output bits that flip when one input bit flips
   (2000 random 64-bit keys x 64 input bits; ideal = 32.00)
   function    average  weakest input   bit
   identity       1.00          1.00     0
   k*31+7         3.67          1.00    63
   Fibonacci     16.02          1.00    63
   SplitMix64    31.97         31.80    25

6. Codes B0000..B1999 into m = 101 buckets
   hash    empty  largest
   sum        72      150
   poly        0       23
```

The last experiment summarizes the section. The same 2000 codes, the same 101 buckets: the sum of characters leaves 72 buckets empty and piles 150 codes into one; the polynomial hash uses every bucket and never puts more than 23 codes into one, close to the $2000/101 \approx 19.8$ that perfect spreading would give.

> **A HASH FOR TABLES IS NOT A HASH FOR PASSWORDS.** Everything in this section aims at speed. A *cryptographic* hash (SHA-256) aims instead at making it infeasible to find two inputs with the same output, and a *password* hash (Argon2id, bcrypt) is deliberately slow and

> memory-hungry so that guessing billions of passwords becomes expensive. Never store passwords with `hashPoly` or `std::hash`, and never use a password hash to index a table.

15.4 Collisions and the Birthday Bound

15.4.1 Collisions cannot be avoided

A collision is not a sign of a bad hash function. As soon as the universe U has more keys than the table has slots, the *pigeonhole principle* guarantees that some two keys share a slot: if $|U| > m$ keys are placed into m slots, some slot receives at least two. For book titles and $m = 7$ this is obvious. The surprising part is how *soon* collisions appear even when only a few keys are stored in a large table.

15.4.2 The birthday problem

How many people must be in a room before two of them probably share a birthday? Many people guess about 183, half of 365. The answer is 23. The same calculation tells us how soon n keys collide in a table of m slots under simple uniform hashing, Equation (15.1).

Insert the keys one at a time and ask for the probability that *no* collision has happened. The first key can go anywhere. The second avoids the first with probability $(m-1)/m = 1 - 1/m$. The third must avoid two occupied slots, probability $1 - 2/m$. In general the $(i+1)$-st key avoids the i occupied slots with probability $1 - i/m$, independently of how the earlier keys fell. Multiplying,

$$\Pr[\text{no collision among } n \text{ keys}] = \prod_{i=1}^{n-1} \left(1 - \frac{i}{m}\right). \tag{15.7}$$

This product is exact but hides its shape. For small x we have $1 - x \approx e^{-x}$ (the tangent line of e^{-x} at 0, and $1 - x \le e^{-x}$ always), so each factor is about $e^{-i/m}$, and the product of exponentials is the exponential of the sum:

$$\Pr[\text{no collision}] \approx \exp\left(-\sum_{i=1}^{n-1} \frac{i}{m}\right) = \exp\left(-\frac{n(n-1)}{2m}\right). \tag{15.8}$$

The probability of a collision reaches one half when $n(n-1)/(2m) = \ln 2$, that is, when $n^2 \approx 2m \ln 2$:

$$n_{1/2} \approx \sqrt{2 \ln 2 \cdot m} \approx 1.1774 \sqrt{m}. \tag{15.9}$$

For $m = 365$ this gives $1.1774 \times 19.10 = 22.5$, so 23 people. The *square root* is the whole story: collisions become likely when the number of keys reaches about the square root of the number of slots, not a constant fraction of it.

There is an even simpler way to see the same number. Among n keys there are $\binom{n}{2} = n(n-1)/2$ pairs, and each pair collides with probability $1/m$ by Equation (15.1). By linearity of expectation,

$$E[\text{number of colliding pairs}] = \frac{n(n-1)}{2m}. \tag{15.10}$$

For 23 keys in 365 slots that is $253/365 \approx 0.69$ expected colliding pairs, the exponent of Equation (15.8).

15.4.3 The numbers, computed

Listing 15.2 computes Equation (15.7) exactly, compares it with Equation (15.8), finds the smallest n that reaches 50% and 99% for several table sizes, and checks the arithmetic with a simulation of 100,000 rooms of 23 random keys.

Listing 15.2 `birthday.cpp`: exact and approximate collision probabilities, the number of keys for even and near-certain odds, and a simulation.

```
// birthday.cpp -- how soon do random keys collide? The birthday bound of
// Section 15.4: exact probabilities, the exponential approximation, the number
// of keys that gives even odds, and a simulation that checks the arithmetic.
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <vector>

typedef std::uint64_t u64;

u64 rngState = 15;                                      // SplitMix64, fixed seed
u64 nextRandom() {
    u64 z = (rngState += 0x9E3779B97F4A7C15ULL);
    z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9ULL;
    z = (z ^ (z >> 27)) * 0x94D049BB133111EBULL;
    return z ^ (z >> 31);
}

// Exact probability that n keys thrown into m slots are all different.
double allDistinct(long long n, double m) {
    double p = 1.0;
    for (long long i = 1; i < n; ++i) p *= 1.0 - (double)i / m;
    return p;
}

// Smallest n for which the probability of a collision reaches `target`.
long long keysFor(double m, double target) {
    double p = 1.0;                                     // P(all distinct) so far
    long long n = 1;
    while (1.0 - p < target) {
        p *= 1.0 - (double)n / m;
        ++n;
    }
    return n;
}
```

Listing 15.2 (Continued) *birthday.cpp: exact and approximate collision probabilities, the number of keys for even and near-certain odds, and a simulation.*

```
int main() {
    std::printf("1. m = 365 slots: probability of at least one collision\n");
    std::printf("    n    exact   approx  expected pairs\n");
    const int ns[8] = {10, 20, 23, 30, 40, 50, 57, 70};
    for (int n : ns) {
        double exact = 1.0 - allDistinct(n, 365.0);
        double approx = 1.0 - std::exp(-(double)n * (n - 1) / (2.0 * 365.0));
        std::printf("  %3d   %.4f   %.4f   %6.3f\n", n, exact, approx,
                    (double)n * (n - 1) / (2.0 * 365.0));
    }

    std::printf("\n2. Keys needed for a 50%% and a 99%% chance of a collision\n");
    std::printf("  %-8s %12s %14s %12s\n", "m", "n (50%)", "1.1774*sqrt(m)",
                "n (99%)");
    const double msz[5] = {365.0, 1000.0, 1e6, 4294967296.0,
                           18446744073709551616.0};
    const char *mname[5] = {"365", "1000", "10^6", "2^32", "2^64"};
    for (int i = 0; i < 5; ++i) {
        double m = msz[i];
        double rule = std::sqrt(2.0 * m * std::log(2.0));
        if (i < 4) {
            std::printf("  %-8s %12lld %14.1f %12lld\n", mname[i], keysFor(m, 0.5),
                        rule, keysFor(m, 0.99));
        } else {                            // too many steps: formula only
            std::printf("  %-8s %12s %14.3e %12.3e\n", mname[i], "(formula)", rule,
                        std::sqrt(2.0 * m * std::log(100.0)));
        }
    }

    std::printf("\n3. Simulation: 23 random keys into 365 slots, 100000 trials\n");
    const int TRIALS = 100000;
    int withCollision = 0;
    std::vector<int> seen(365, -1);
    for (int t = 0; t < TRIALS; ++t) {
        bool hit = false;
        for (int k = 0; k < 23; ++k) {
            int slot = (int)(nextRandom() % 365);
            if (seen[slot] == t) hit = true;   // slot already used this trial
            seen[slot] = t;
        }
        if (hit) ++withCollision;
    }
    std::printf("  trials with a collision: %d  (%.4f; exact %.4f)\n",
                withCollision, (double)withCollision / TRIALS,
                1.0 - allDistinct(23, 365.0));
```

Listing 15.2 (Continued) *birthday.cpp: exact and approximate collision probabilities, the number of keys for even and near-certain odds, and a simulation.*

```
    std::printf("\n4. n = m = 1000 keys into 1000 buckets: "
                "how many buckets stay empty?\n");
    std::vector<int> count(1000, 0);
    for (int k = 0; k < 1000; ++k) ++count[nextRandom() % 1000];
    int empty = 0, largest = 0;
    for (int c : count) {
        if (c == 0) ++empty;
        if (c > largest) largest = c;
    }
    std::printf("  empty buckets: %d  (expected 1000*(1-1/1000)^1000 = %.1f)\n",
                empty, 1000.0 * std::pow(1.0 - 1.0 / 1000.0, 1000.0));
    std::printf("  largest bucket: %d keys\n", largest);
    return 0;
}
```

```
1. m = 365 slots: probability of at least one collision
    n    exact   approx  expected pairs
   10   0.1169   0.1160    0.123
   20   0.4114   0.4058    0.521
   23   0.5073   0.5000    0.693
   30   0.7063   0.6963    1.192
   40   0.8912   0.8820    2.137
   50   0.9704   0.9651    3.356
   57   0.9901   0.9874    4.373
   70   0.9992   0.9987    6.616

2. Keys needed for a 50% and a 99% chance of a collision
  m             n (50%) 1.1774*sqrt(m)      n (99%)
  365               23          22.5           57
  1000              38          37.2           95
  10^6            1178        1177.4         3034
  2^32           77164       77162.7       198892
  2^64       (formula)     5.057e+09    1.303e+10

3. Simulation: 23 random keys into 365 slots, 100000 trials
  trials with a collision: 50650  (0.5065; exact 0.5073)

4. n = m = 1000 keys into 1000 buckets: how many buckets stay empty?
  empty buckets: 353  (expected 1000*(1-1/1000)^1000 = 367.7)
  largest bucket: 5 keys
```

Table 15.2 collects the second part of the output. Read its last line slowly: even with 2^{64} possible hash values, about five billion random keys are enough for an even chance of a collision. Five billion is the number of pages a large web crawler handles, which is why systems that identify objects by a 64-bit hash (and assume no collisions) must either use 128 bits or check the full keys.

Table 15.2 How many random keys make a collision likely, computed by **Listing 15.2** 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}

Figure 15.2 draws the probability for $m = 365$. The exact product and the approximation are nearly indistinguishable; the approximation is always slightly low, because $1 - x \le e^{-x}$ makes the true probability of *no* collision slightly smaller than the approximation says.

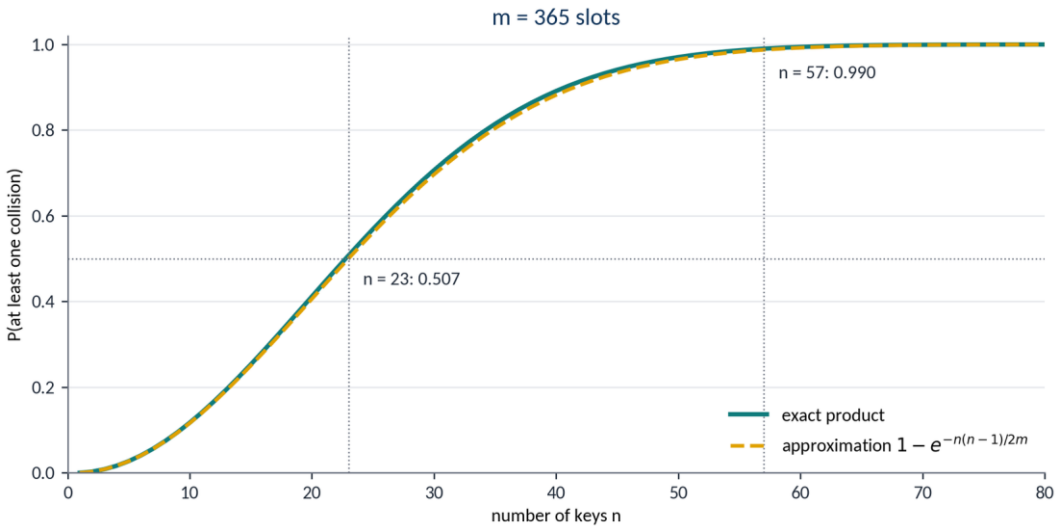


Figure 15.2 The probability of at least one collision among n random keys in 365 slots: 0.507 at $n = 23$ and 0.990 at $n = 57$. The dashed curve is the approximation Equation (15.8).

15.4.4 What this means for a hash table

A hash table does not try to avoid collisions; the birthday bound shows that it could not, since a table would need about n^2 slots for n keys. It *plans* for them. Part 4 of the output makes the plan concrete: 1000 keys in 1000 buckets left 353 buckets empty (the expectation is $1000 \cdot (1 - 1/1000)^{1000} \approx 1000/e \approx 367.7$) and put five keys into the fullest bucket. A table with $n = m$ therefore has about 37% of its buckets empty and a few buckets holding several keys, and by Equation (15.10) it contains about $1000 \cdot 999/2000 \approx 500$ colliding pairs. The next two sections give the two classic ways to live with that: let a bucket hold a list (chaining), or let a key move to another slot (open addressing).

15.5 Separate Chaining

15.5.1 A linked list in every bucket

In *separate chaining* each slot of the table holds the head pointer of a singly linked list, the list of Chapter 7, and every key that hashes to that slot is a node of that list. **Figure 15.3** shows the table that **Listing 15.3** builds from seven Pustaka Ceria titles and a table of $m = 7$ buckets. The keys are titles and the values are book codes, so the table answers “which code does this title have?”

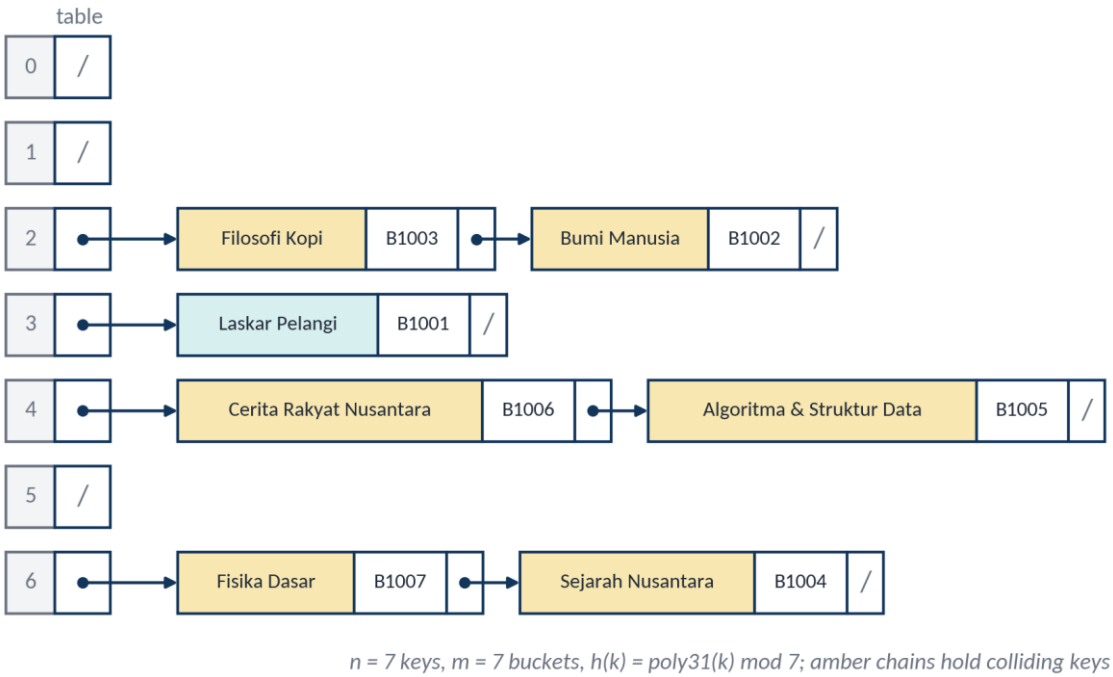


Figure 15.3 Separate chaining: seven titles in seven buckets with $h(k) = \text{poly31}(k) \bmod 7$. Each bucket holds the head of a singly linked list; buckets 2, 4 and 6 hold two colliding titles each, and buckets 0, 1 and 5 are empty.

The three operations reduce to the list operations of Chapter 7, applied to one short list.

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
```

Insertion goes at the *front* of the chain, the $O(1)$ `frontInsertion` of Section 7.5.2, but only after a search, because a table maps each key to *one* value: putting an existing key again must update it, not add a second node. Deletion is `deleteKey` of Section 7.6.5 with its trailing pointer `prev`. Line 7 of `CHAINED-PUT` watches the load factor; Section 15.7 explains why and when to rehash.

15.5.2 The program

Listing 15.3 implements **Pseudocode 15.2** as a class. The member `table_` is a `std::vector<Node *>` of chain heads, so `table_[b]` plays the role of `head` in Chapter 7, one per bucket. The destructor walks every chain and deletes every node, and copying is forbidden, the rule of three of Section

7.10.2. The rehash moves nodes rather than copying them: each node is unlinked from its old chain and pushed on the front of its new chain, so no `new` or `delete` happens during a rehash at all.

Listing 15.3 `chaining.cpp`: a hash table with separate chaining built from singly linked nodes, with search, update, deletion and rehashing.

```
// chaining.cpp -- a hash table with separate chaining, built by hand from the
// singly linked nodes of Chapter 7. Keys are book titles, values are book codes.
#include <cstdint>
#include <cstdio>
#include <string>
#include <vector>

std::uint64_t hashPoly(const std::string &s) {       // Section 15.3.4, base 31
    std::uint64_t h = 0;
    for (unsigned char c : s) h = h * 31 + c;
    return h;
}

struct Node {
    std::string key;
    std::string value;
    Node *next;
};

class ChainedHashTable {
public:
    explicit ChainedHashTable(std::size_t m = 7, double maxLoad = 1.0)
        : table_(m, nullptr), n_(0), maxLoad_(maxLoad) {}

    ~ChainedHashTable() {                            // release every node
        for (Node *head : table_) {
            while (head != nullptr) {
                Node *del = head;
                head = head->next;
                delete del;
            }
        }
    }

    ChainedHashTable(const ChainedHashTable &) = delete;   // no copies (rule of
    ChainedHashTable &operator=(const ChainedHashTable &) = delete;   // three)

    std::size_t bucketOf(const std::string &key) const {
        return hashPoly(key) % table_.size();
    }
```

Listing 15.3 (Continued) *chaining.cpp: a hash table with separate chaining built from singly linked nodes, with search, update, deletion and rehashing.*

```
    // 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()));
    }

    // Return a pointer to the value stored for key, or nullptr if key is absent.
    // `compared` receives the number of keys compared along the way.
    const std::string *get(const std::string &key, int &compared) const {
        compared = 0;
        for (Node *p = table_[bucketOf(key)]; p != nullptr; p = p->next) {
            ++compared;
            if (p->key == key) return &p->value;
        }
        return nullptr;
    }

    bool erase(const std::string &key) {            // deleteKey of Section 7.6.5
        std::size_t b = bucketOf(key);
        Node *prev = nullptr, *cur = table_[b];
        while (cur != nullptr && cur->key != key) {
            prev = cur;
            cur = cur->next;
        }
        if (cur == nullptr) return false;           // not present
        if (prev == nullptr) table_[b] = cur->next;  // first node of the chain
        else prev->next = cur->next;
        delete cur;
        --n_;
        return true;
    }
```

Listing 15.3 (Continued) *chaining.cpp: a hash table with separate chaining built from singly linked nodes, with search, update, deletion and rehashing.*

```
    void show() const {
        int empty = 0, longest = 0;
        for (std::size_t b = 0; b < table_.size(); ++b) {
            int len = 0;
            std::printf("  %2zu:", b);
            for (Node *p = table_[b]; p != nullptr; p = p->next, ++len)
                std::printf(" %s=%s ->", p->key.c_str(), p->value.c_str());
            std::printf(len == 0 ? " (empty)\n" : " NULL\n");
            if (len == 0) ++empty;
            if (len > longest) longest = len;
        }
        std::printf("  n = %zu, m = %zu, load factor = %.2f, longest chain = %d, "
                    "empty buckets = %d\n", n_, table_.size(),
                    (double)n_ / table_.size(), longest, empty);
    }

private:
    static bool isPrime(std::size_t x) {
        if (x < 2) return false;
        for (std::size_t d = 2; d * d <= x; ++d)
            if (x % d == 0) return false;
        return true;
    }
    static std::size_t nextPrime(std::size_t x) {
        while (!isPrime(x)) ++x;
        return x;
    }

    // Move every node into a new array of newM buckets. No node is created or
    // deleted: each one is unlinked from its old chain and pushed on the front
    // of its new chain, because its bucket number depends on m.
    void rehash(std::size_t newM) {
        std::printf("  -- load factor %.2f > %.2f: rehash from %zu to %zu "
                    "buckets\n",
                    (double)n_ / table_.size(), maxLoad_, table_.size(), newM);
        std::vector<Node *> old(newM, nullptr);
        old.swap(table_);                           // table_ is now the new array
        for (Node *head : old) {
            while (head != nullptr) {
                Node *moving = head;
                head = head->next;
                std::size_t b = bucketOf(moving->key);
                moving->next = table_[b];
                table_[b] = moving;
            }
        }
```

Listing 15.3 (Continued) *chaining.cpp: a hash table with separate chaining built from singly linked nodes, with search, update, deletion and rehashing.*

```
    }

    std::vector<Node *> table_;                     // table_[b] = head of chain b
    std::size_t n_;                                 // number of keys stored
    double maxLoad_;                                // rehash when n/m exceeds it
};

void lookup(const ChainedHashTable &t, const std::string &title) {
    int compared;
    const std::string *code = t.get(title, compared);
    std::printf("  get(\"%s\") -> %s, bucket %zu, %d key(s) compared\n",
                title.c_str(), code ? code->c_str() : "not found",
                t.bucketOf(title), compared);
}

int main() {
    const char *title[8] = {"Laskar Pelangi", "Bumi Manusia", "Filosofi Kopi",
                            "Sejarah Nusantara", "Algoritma & Struktur Data",
                            "Cerita Rakyat Nusantara", "Fisika Dasar",
                            "Negeri 5 Menara"};
    ChainedHashTable t(7, 1.0);

    std::printf("1. Seven titles into m = 7 buckets, h(k) = poly31(k) mod 7\n");
    for (int i = 0; i < 7; ++i) {
        std::string code = "B100" + std::to_string(i + 1);
        std::printf("  put(\"%s\", %s) -> bucket %zu\n", title[i], code.c_str(),
                    t.bucketOf(title[i]));
        t.put(title[i], code);
    }
    t.show();

    std::printf("\n2. Searching\n");
    lookup(t, "Filosofi Kopi");
    lookup(t, "Bumi Manusia");
    lookup(t, "Ayat-Ayat Cinta");
    lookup(t, "Pulang");

    std::printf("\n3. Updating and deleting\n");
    t.put("Laskar Pelangi", "B1001+B1011");          // a second copy arrived
    lookup(t, "Laskar Pelangi");
    std::printf("  erase(\"Algoritma & Struktur Data\") -> %s\n",
                t.erase("Algoritma & Struktur Data") ? "deleted" : "not found");
    std::printf("  erase(\"Pulang\") -> %s\n",
                t.erase("Pulang") ? "deleted" : "not found");
    t.show();
```

Listing 15.3 (Continued) *chaining.cpp: a hash table with separate chaining built from singly linked nodes, with search, update, deletion and rehashing.*

```
    std::printf("\n4. Two more titles: the eighth key pushes n/m above 1\n");
    t.put("Algoritma & Struktur Data", "B1005");
    t.put("Negeri 5 Menara", "B1008");
    t.show();
    lookup(t, "Bumi Manusia");
    return 0;
}
```

```
1. Seven titles into m = 7 buckets, h(k) = poly31(k) mod 7
  put("Laskar Pelangi", B1001) -> bucket 3
  put("Bumi Manusia", B1002) -> bucket 2
  put("Filosofi Kopi", B1003) -> bucket 2
  put("Sejarah Nusantara", B1004) -> bucket 6
  put("Algoritma & Struktur Data", B1005) -> bucket 4
  put("Cerita Rakyat Nusantara", B1006) -> bucket 4
  put("Fisika Dasar", B1007) -> bucket 6
   0: (empty)
   1: (empty)
   2: Filosofi Kopi=B1003 -> Bumi Manusia=B1002 -> NULL
   3: Laskar Pelangi=B1001 -> NULL
   4: Cerita Rakyat Nusantara=B1006 -> Algoritma & Struktur Data=B1005 -> NULL
   5: (empty)
   6: Fisika Dasar=B1007 -> Sejarah Nusantara=B1004 -> NULL
  n = 7, m = 7, load factor = 1.00, longest chain = 2, empty buckets = 3

2. Searching
  get("Filosofi Kopi") -> B1003, bucket 2, 1 key(s) compared
  get("Bumi Manusia") -> B1002, bucket 2, 2 key(s) compared
  get("Ayat-Ayat Cinta") -> not found, bucket 4, 2 key(s) compared
  get("Pulang") -> not found, bucket 1, 0 key(s) compared

3. Updating and deleting
  get("Laskar Pelangi") -> B1001+B1011, bucket 3, 1 key(s) compared
  erase("Algoritma & Struktur Data") -> deleted
  erase("Pulang") -> not found
   0: (empty)
   1: (empty)
   2: Filosofi Kopi=B1003 -> Bumi Manusia=B1002 -> NULL
   3: Laskar Pelangi=B1001+B1011 -> NULL
   4: Cerita Rakyat Nusantara=B1006 -> NULL
   5: (empty)
   6: Fisika Dasar=B1007 -> Sejarah Nusantara=B1004 -> NULL
  n = 6, m = 7, load factor = 0.86, longest chain = 2, empty buckets = 3

4. Two more titles: the eighth key pushes n/m above 1
  -- load factor 1.14 > 1.00: rehash from 7 to 17 buckets
   0: Bumi Manusia=B1002 -> NULL
   1: Fisika Dasar=B1007 -> NULL
   2: (empty)
   3: (empty)
   4: Negeri 5 Menara=B1008 -> NULL
   5: Cerita Rakyat Nusantara=B1006 -> NULL
   6: (empty)
   7: Laskar Pelangi=B1001+B1011 -> NULL
   8: (empty)
   9: Filosofi Kopi=B1003 -> NULL
  10: (empty)
  11: (empty)
  12: (empty)
  13: (empty)
  14: Sejarah Nusantara=B1004 -> NULL
  15: Algoritma & Struktur Data=B1005 -> NULL
  16: (empty)
  n = 8, m = 17, load factor = 0.47, longest chain = 1, empty buckets = 9
  get("Bumi Manusia") -> B1002, bucket 0, 1 key(s) compared
```

Follow the run with **Figure 15.3**. Part 1 inserts seven titles; *Filosofi Kopi* went to bucket 2 after *Bumi Manusia*, so front insertion put it first in the chain. Part 2 shows the cost of a search in *key comparisons*: *Filosofi Kopi* is found after one comparison, *Bumi Manusia*, second in the same chain, after two. An absent title costs the length of its chain: *Ayat-Ayat Cinta* hashes to bucket 4, compares with both titles there and gives up, while *Pulang* hashes to the empty bucket 1 and is rejected with no comparison at all. Part 3 updates the value of an existing key (the size stays 7)

and deletes the *second* node of chain 4, which exercises the `prev->next = cur->next` branch. In part 4 the eighth key makes $n/m = 8/7 \approx 1.14$ exceed 1, the table grows to 17 buckets, the next prime after 14, and every title moves to a new bucket computed modulo 17. After the rehash no chain is longer than 1.

15.5.3 The same table in Java and Python

Listing 15.4 and **Listing 15.5** are line-by-line translations. They print exactly the output above, because they use the same polynomial hash (`long` arithmetic in Java, `& MASK` in Python) rather than the languages' built-in `hashCode` or `hash`, which differ between languages and, in Python, between runs.

Listing 15.4 `Chaining.java`: the chained hash table in Java.

```
// Chaining.java -- a hash table with separate chaining, built by hand from the
// singly linked nodes of Chapter 7. Keys are book titles, values are book codes.
import java.util.Arrays;

public class Chaining {
    static long hashPoly(String s) {                // Section 15.3.4, base 31
        long h = 0;
        for (int i = 0; i < s.length(); i++) h = h * 31 + s.charAt(i);
        return h;
    }

    static class Node {
        final String key;
        String value;
        Node next;

        Node(String key, String value, Node next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    static class ChainedHashTable {
        private Node[] table;                       // table[b] = head of chain b
        private int n;                              // number of keys stored
        private final double maxLoad;               // rehash when n/m exceeds it

        ChainedHashTable(int m, double maxLoad) {
            table = new Node[m];
            this.maxLoad = maxLoad;
        }

        int bucketOf(String key) {
            return (int) Long.remainderUnsigned(hashPoly(key), table.length);
        }
```

Listing 15.4 (Continued) *Chaining.java: the chained hash table in Java.*

```
    // Insert key with value, or replace the value if key is already present.
    void put(String key, String value) {
        int b = bucketOf(key);
        for (Node p = table[b]; p != null; p = p.next) {
            if (p.key.equals(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.length > maxLoad)
            rehash(nextPrime(2 * table.length));
    }

    // Return the value stored for key, or null; compared[0] gets the count.
    String get(String key, int[] compared) {
        compared[0] = 0;
        for (Node p = table[bucketOf(key)]; p != null; p = p.next) {
            compared[0]++;
            if (p.key.equals(key)) return p.value;
        }
        return null;
    }

    boolean erase(String key) {                 // deleteKey of Section 7.6.5
        int b = bucketOf(key);
        Node prev = null, cur = table[b];
        while (cur != null && !cur.key.equals(key)) {
            prev = cur;
            cur = cur.next;
        }
        if (cur == null) return false;          // not present
        if (prev == null) table[b] = cur.next;  // first node of the chain
        else prev.next = cur.next;
        n--;
        return true;
    }
```

Listing 15.4 (Continued) *Chaining.java: the chained hash table in Java.*

```
    void show() {
        int empty = 0, longest = 0;
        for (int b = 0; b < table.length; b++) {
            int len = 0;
            StringBuilder sb = new StringBuilder(String.format("  %2d:", b));
            for (Node p = table[b]; p != null; p = p.next, len++)
                sb.append(' ').append(p.key).append('=').append(p.value)
                  .append(" ->");
            sb.append(len == 0 ? " (empty)" : " NULL");
            System.out.println(sb);
            if (len == 0) empty++;
            if (len > longest) longest = len;
        }
        System.out.printf("  n = %d, m = %d, load factor = %.2f, longest "
                        + "chain = %d, "
                          + "empty buckets = %d%n", n, table.length,
                          (double) n / table.length, longest, empty);
    }

    private static boolean isPrime(int x) {
        if (x < 2) return false;
        for (int d = 2; d * d <= x; d++)
            if (x % d == 0) return false;
        return true;
    }

    private static int nextPrime(int x) {
        while (!isPrime(x)) x++;
        return x;
    }

    // Move every node into a new array of newM buckets, relinking, not copying.
    private void rehash(int newM) {
        System.out.printf("  -- load factor %.2f > %.2f: rehash from %d to %d "
                          + "buckets%n", (double) n / table.length, maxLoad,
                          table.length, newM);
        Node[] old = table;
        table = new Node[newM];
        for (Node head : old) {
            while (head != null) {
                Node moving = head;
                head = head.next;
                int b = bucketOf(moving.key);
                moving.next = table[b];
                table[b] = moving;
            }
```

Listing 15.4 (Continued) *Chaining.java: the chained hash table in Java.*

```
            }
        }
    }

    static void lookup(ChainedHashTable t, String title) {
        int[] compared = new int[1];
        String code = t.get(title, compared);
        System.out.printf("  get(\"%s\") -> %s, bucket %d, %d key(s) compared%n",
                          title, code != null ? code : "not found",
                          t.bucketOf(title), compared[0]);
    }

    public static void main(String[] args) {
        String[] title = {"Laskar Pelangi", "Bumi Manusia", "Filosofi Kopi",
                          "Sejarah Nusantara", "Algoritma & Struktur Data",
                          "Cerita Rakyat Nusantara", "Fisika Dasar",
                          "Negeri 5 Menara"};
        ChainedHashTable t = new ChainedHashTable(7, 1.0);

        System.out.println("1. Seven titles into m = 7 buckets, h(k) = poly31(k) "
                           + "mod 7");
        for (int i = 0; i < 7; i++) {
            String code = "B100" + (i + 1);
            System.out.printf("  put(\"%s\", %s) -> bucket %d%n", title[i], code,
                              t.bucketOf(title[i]));
            t.put(title[i], code);
        }
        t.show();

        System.out.println("\n2. Searching");
        lookup(t, "Filosofi Kopi");
        lookup(t, "Bumi Manusia");
        lookup(t, "Ayat-Ayat Cinta");
        lookup(t, "Pulang");

        System.out.println("\n3. Updating and deleting");
        t.put("Laskar Pelangi", "B1001+B1011");      // a second copy arrived
        lookup(t, "Laskar Pelangi");
        boolean gone = t.erase("Algoritma & Struktur Data");
        System.out.println("  erase(\"Algoritma & Struktur Data\") -> "
                           + (gone ? "deleted" : "not found"));
        System.out.println("  erase(\"Pulang\") -> "
                           + (t.erase("Pulang") ? "deleted" : "not found"));
        t.show();
```

Listing 15.4 (Continued) *Chaining.java: the chained hash table in Java.*

```
        System.out.println("\n4. Two more titles: the eighth key pushes n/m "
                           + "above 1");
        t.put("Algoritma & Struktur Data", "B1005");
        t.put("Negeri 5 Menara", "B1008");
        t.show();
        lookup(t, "Bumi Manusia");
    }
}
```

Listing 15.5 `chaining.py`: the chained hash table in Python.

```
# chaining.py -- a hash table with separate chaining, built by hand from the
# singly linked nodes of Chapter 7. Keys are book titles, values are book codes.
MASK = (1 << 64) - 1


def hash_poly(s):                                   # Section 15.3.4, base 31
    h = 0
    for c in s:
        h = (h * 31 + ord(c)) & MASK
    return h


class Node:
    __slots__ = ("key", "value", "next")

    def __init__(self, key, value, nxt):
        self.key, self.value, self.next = key, value, nxt


def is_prime(x):
    if x < 2:
        return False
    d = 2
    while d * d <= x:
        if x % d == 0:
            return False
        d += 1
    return True


def next_prime(x):
    while not is_prime(x):
        x += 1
    return x


class ChainedHashTable:
    def __init__(self, m=7, max_load=1.0):
        self.table = [None] * m                     # table[b] = head of chain b
        self.n = 0                                  # number of keys stored
        self.max_load = max_load                    # rehash when n/m exceeds it

    def bucket_of(self, key):
        return hash_poly(key) % len(self.table)
```

Listing 15.5 (Continued) *chaining.py: the chained hash table in Python.*

```
    def put(self, key, value):
        """Insert key with value, or replace the value if key is present."""
        b = self.bucket_of(key)
        p = self.table[b]
        while p is not None:
            if p.key == key:                          # present: update in place
                p.value = value
                return
            p = p.next
        self.table[b] = Node(key, value, self.table[b])   # absent: front insertion
        self.n += 1
        if self.n / len(self.table) > self.max_load:
            self._rehash(next_prime(2 * len(self.table)))

    def get(self, key):
        """Return (value or None, number of keys compared)."""
        compared = 0
        p = self.table[self.bucket_of(key)]
        while p is not None:
            compared += 1
            if p.key == key:
                return p.value, compared
            p = p.next
        return None, compared

    def erase(self, key):                             # deleteKey of Section 7.6.5
        b = self.bucket_of(key)
        prev, cur = None, self.table[b]
        while cur is not None and cur.key != key:
            prev, cur = cur, cur.next
        if cur is None:                               # not present
            return False
        if prev is None:                              # first node of the chain
            self.table[b] = cur.next
        else:
            prev.next = cur.next
        self.n -= 1
        return True
```

Listing 15.5 (Continued) *chaining.py: the chained hash table in Python.*

```
    def show(self):
        empty = longest = 0
        for b, head in enumerate(self.table):
            parts, p = [], head
            while p is not None:
                parts.append(" %s=%s ->" % (p.key, p.value))
                p = p.next
            tail = " NULL" if parts else " (empty)"
            print("  %2d:" % b + "".join(parts) + tail)
            if not parts:
                empty += 1
            longest = max(longest, len(parts))
        m = len(self.table)
        print("  n = %d, m = %d, load factor = %.2f, longest chain = %d, "
              "empty buckets = %d" % (self.n, m, self.n / m, longest, empty))

    def _rehash(self, new_m):
        """Move every node into a new list of new_m buckets, relinking."""
        print("  -- load factor %.2f > %.2f: rehash from %d to %d buckets"
              % (self.n / len(self.table), self.max_load, len(self.table), new_m))
        old, self.table = self.table, [None] * new_m
        for head in old:
            while head is not None:
                moving, head = head, head.next
                b = self.bucket_of(moving.key)
                moving.next = self.table[b]
                self.table[b] = moving


def lookup(t, title):
    code, compared = t.get(title)
    print('  get("%s") -> %s, bucket %d, %d key(s) compared'
          % (title, code if code is not None else "not found", t.bucket_of(title),
             compared))


def main():
    title = ["Laskar Pelangi", "Bumi Manusia", "Filosofi Kopi", "Sejarah Nusantara",
             "Algoritma & Struktur Data", "Cerita Rakyat Nusantara", "Fisika Dasar",
             "Negeri 5 Menara"]
    t = ChainedHashTable(7, 1.0)
```

Listing 15.5 (Continued) *chaining.py: the chained hash table in Python.*

```
    print("1. Seven titles into m = 7 buckets, h(k) = poly31(k) mod 7")
    for i in range(7):
        code = "B100%d" % (i + 1)
        print('  put("%s", %s) -> bucket %d' % (title[i], code,
                                                t.bucket_of(title[i])))
        t.put(title[i], code)
    t.show()

    print("\n2. Searching")
    for x in ("Filosofi Kopi", "Bumi Manusia", "Ayat-Ayat Cinta", "Pulang"):
        lookup(t, x)

    print("\n3. Updating and deleting")
    t.put("Laskar Pelangi", "B1001+B1011")          # a second copy arrived
    lookup(t, "Laskar Pelangi")
    print('  erase("Algoritma & Struktur Data") -> '
          + ("deleted" if t.erase("Algoritma & Struktur Data") else "not found"))
    gone = t.erase("Pulang")
    print('  erase("Pulang") -> ' + ("deleted" if gone else "not found"))
    t.show()

    print("\n4. Two more titles: the eighth key pushes n/m above 1")
    t.put("Algoritma & Struktur Data", "B1005")
    t.put("Negeri 5 Menara", "B1008")
    t.show()
    lookup(t, "Bumi Manusia")


if __name__ == "__main__":
    main()
```

In Java and Python a deleted node simply becomes unreachable and the garbage collector reclaims it; in C++ `erase` must `delete` it, and the destructor must free the rest. Valgrind confirms that **Listing 15.3** releases all 25 blocks it allocates, including the strings inside the nodes and both bucket arrays:

```
==32382== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==32382== Command: /tmp/ch_g
==32382==
==32382==
==32382== HEAP SUMMARY:
==32382==     in use at exit: 0 bytes in 0 blocks
==32382==   total heap usage: 25 allocs, 25 frees, 78,946 bytes allocated
==32382==
==32382== All heap blocks were freed -- no leaks are possible
==32382==
==32382== For lists of detected and suppressed errors, rerun with: -s
==32382== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

15.5.4 How long are the chains?

Under simple uniform hashing each of the n keys lands in a given bucket with probability $1/m$, so the expected length of any one chain is $n \cdot (1/m) = \alpha$, the load factor. An *unsuccessful* search computes $h(k)$ in $O(1)$ time and then walks a whole chain, which a random absent key picks uniformly:

$$E[\text{comparisons, unsuccessful}] = \alpha, \qquad \text{time } \Theta(1 + \alpha). \tag{15.11}$$

A *successful* search for a key x compares with x itself and with every key in front of it. With front insertion, the keys in front of x are the keys inserted *after* x into the same bucket. If x was the i-th

key inserted, $n-i$ keys came after it, and each landed in x's bucket with probability $1/m$. Averaging over the n keys,

$$E[\text{comparisons, successful}] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right) = 1+\frac{n-1}{2m} \approx 1+\frac{\alpha}{2}. \tag{15.12}$$

(The sum $\sum_{i=1}^{n}(n-i) = 0+1+\cdots+(n-1) = n(n-1)/2$ is the one that counted the comparisons of bubble sort in Section 3.3.5.) Both costs depend on α only, not on n. If the table keeps α below a constant, say 1, by growing m as n grows, every operation takes $O(1)$ time on average. Section 15.7 measures Equation (15.11) and Equation (15.12) on 16,384 buckets and finds them accurate to two decimals.

15.6 Open Addressing

15.6.1 Keys that move instead of chains that grow

Chaining stores colliding keys *outside* the table, in nodes. *Open addressing* stores every key *inside* the table array itself, one key per slot, and when a key's home slot is taken the key tries other slots in a fixed order until it finds a free one. There are no pointers, no nodes and no allocations per key, which saves memory and keeps the data in one cache-friendly block. The price is that the table can fill up: $n \le m$ always, so $\alpha \le 1$.

The order in which a key tries the slots is its *probe sequence*. We extend the hash function with a second argument, the attempt number $i = 0,1,2,\ldots$:

$$h(k,0),\ h(k,1),\ \ldots,\ h(k,m-1), \tag{15.13}$$

and require that for every key this sequence visits each of the m slots exactly once, so that a free slot is always found if one exists. Insertion walks the sequence until it finds an empty slot. Search walks the *same* sequence until it finds the key or reaches an empty slot: an empty slot proves the key is absent, because insertion would have stopped there. The three classic choices of Equation (15.13) differ in how they step from one probe to the next. The trace in **Listing 15.6** inserts the same eight keys, 1003, 1019, 1035, 1012, 1028, 1013, 1044, 1051, into a table of $m = 16$ slots with each of them. Their home slots $h_1(k) = k \bmod 16$ are 11, 11, 11, 4, 4, 5, 4, 11: four keys want slot 11 and three want slot 4, a deliberately crowded example.

15.6.2 Linear probing

The simplest probe sequence tries the next slot, then the next, wrapping around at the end:

$$h(k,i) = (h_1(k)+i) \bmod m. \tag{15.14}$$

In our example 1003 takes slot 11, 1019 finds 11 taken and takes 12, 1035 tries 11 and 12 and takes 13, and 1051 later needs four probes to reach 14. Keys whose home slots are *different* start to interfere too: 1013 wants slot 5, which 1028 took when it was pushed away from slot 4, so 1013 goes to 6; then 1044, whose home is 4, must walk 4, 5, 6 and 7. Occupied slots merge into long runs, and every key that hashes anywhere into a run must walk to its end, making the run longer still. This is *primary clustering*. It makes linear probing degrade quickly as the table fills (Section 15.7 measures 123 probes per unsuccessful search at $\alpha = 0.95$), yet at moderate loads

linear probing is often the fastest method in practice, because its probes touch consecutive memory that the processor has already loaded into its cache.

15.6.3 Quadratic probing

Quadratic probing jumps further with each attempt, so that the keys pushed away from one crowded slot scatter instead of piling up next to it. With a table of $m = 2^p$ slots a convenient choice uses the triangular numbers 0,1,3,6,10, … as offsets:

$$h(k, i) = \left(h_1(k) + \frac{i(i+1)}{2}\right) \bmod m. \tag{15.15}$$

For 1051, with home slot 11, the offsets 0, 1, 3, 6 give the slots 11, 12, 14 and 17 mod 16 = 1. Does this sequence really visit every slot? Suppose attempts i and j, with $0 \le i < j < m$, hit the same slot. Then $m = 2^p$ divides $\frac{j(j+1)}{2} - \frac{i(i+1)}{2} = \frac{(j-i)(j+i+1)}{2}$, so 2^{p+1} divides $(j-i)(j+i+1)$. The two factors differ by $2i+1$, an odd number, so one of them is odd and the whole power 2^{p+1} must divide the other. But $0 < j - i < 2^p$ and $0 < j + i + 1 < 2^{p+1}$, so neither can be divisible by 2^{p+1}, a contradiction. Hence the first m attempts visit m different slots, every slot once. (With the plain squares i^2 and a prime m, only about half the slots are guaranteed to be reached, which is why a table using squares must stay less than half full.)

Quadratic probing removes primary clustering but keeps a milder *secondary clustering*: keys with the same home slot still follow the same sequence. In the trace, 1003, 1019, 1035 and 1051 all start at 11 and follow 11, 12, 14, 1 one after the other.

15.6.4 Double hashing

Double hashing removes that last clustering by letting the *step* depend on the key, through a second hash function h_2:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m. \tag{15.16}$$

The sequence visits every slot exactly when $h_2(k)$ and m have no common factor, by the argument of Section 15.3.2 with the stride $d = h_2(k)$. With $m = 2^p$ it is enough to make $h_2(k)$ odd; our programs use $h_2(k) = 1 + 2 \cdot (\lfloor k/m \rfloor \bmod (m/2))$, which uses the key bits that h_1 ignored. For 1051, $\lfloor 1051/16 \rfloor = 65$, $65 \bmod 8 = 1$, so $h_2(1051) = 3$ and the probes are 11,14,17 mod 16 = 1, …; slot 14 is free, and 1051 is stored after two probes. Keys that share a home slot now have different steps and separate immediately: 1019 steps by $h_2 = 15$, which is -1 modulo 16, and goes to slot 10; 1035 steps by 1 and goes to slot 12. With a well-mixed h_2, double hashing behaves almost like the ideal *uniform hashing*, in which every probe sequence is equally likely to be any of the $m!$ orderings of the slots.

Figure 15.4 shows the three tables after the eight insertions, with the probe path of 1051 drawn in each.

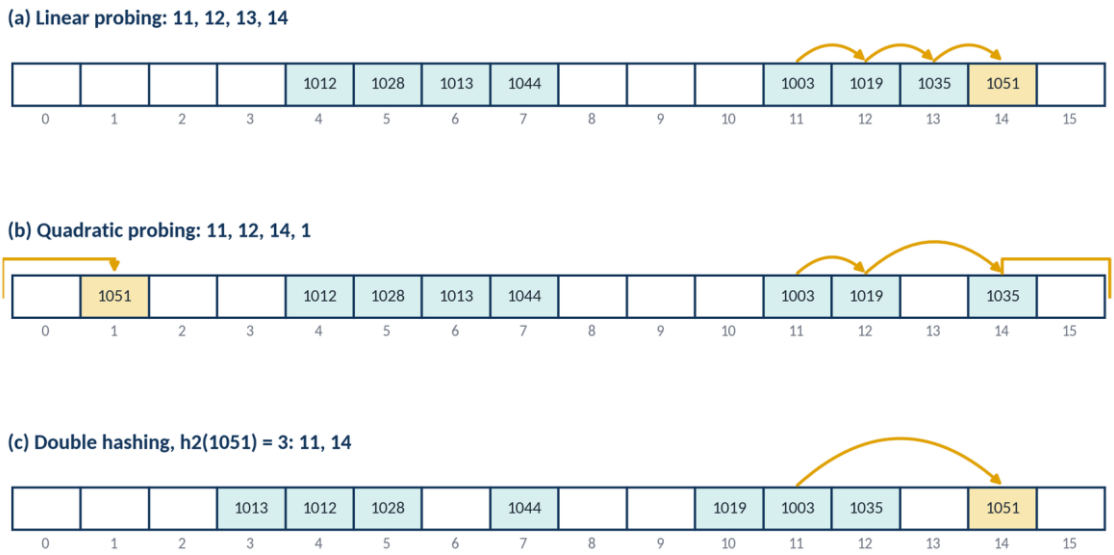


Figure 15.4 The same eight keys in 16 slots with (a) linear probing, (b) quadratic probing and (c) double hashing. The arrows are the probes made to insert (and later to find) 1051: four, four and two.

15.6.5 Deletion and tombstones

Deleting from an open-addressed table is harder than it looks. Suppose we delete 1019 from the linear-probing table by simply marking slot 12 empty. A later search for 1035 starts at its home slot 11, finds 1003, moves to 12, finds it empty, and concludes that 1035 is absent, although 1035 sits one slot further, in 13. The empty slot has cut the probe sequence along which 1035 was inserted. **Figure 15.5** draws exactly this failure, which part 4 of the output of **Listing 15.6** reproduces ("find 1035: not found, 2 probe(s)").

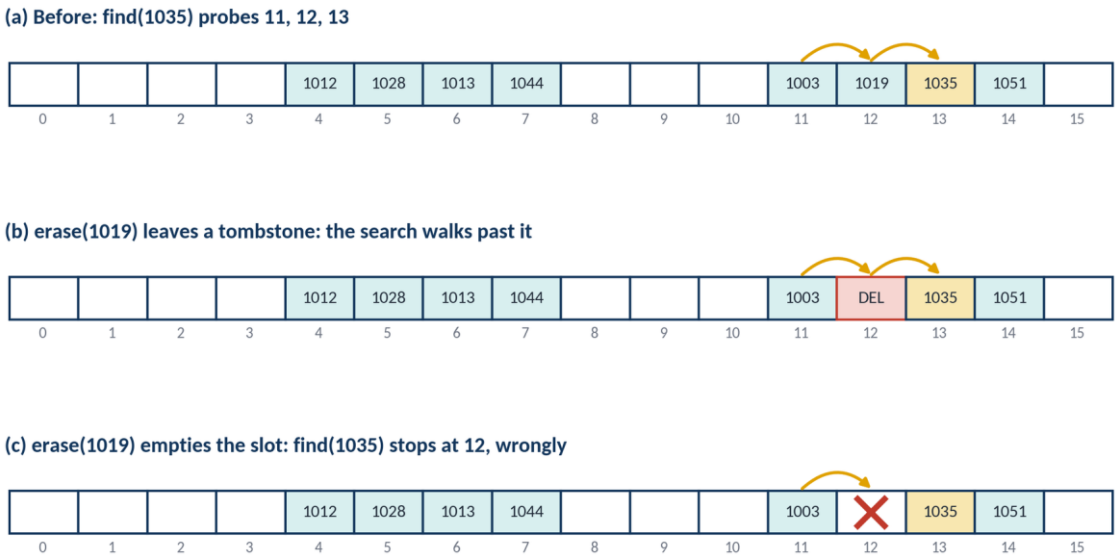


Figure 15.5 Deleting 1019 from the linear-probing table. (a) Before, a search for 1035 probes 11, 12, 13. (b) With a tombstone in slot 12 the search walks past it and still finds 1035. (c) If slot 12 is simply emptied, the search stops there and wrongly reports 1035 absent.

The standard remedy is a *tombstone*: a special marker, DELETED, that means "a key used to live here". Search treats a tombstone as occupied and walks past it; insertion treats it as free and may reuse it. **Pseudocode 15.3** gives the three operations with the three slot states EMPTY, OCCUPIED and DELETED.

Pseudocode 15.3 Search, insertion and deletion with open addressing and tombstones.

```
OPEN-SEARCH(T, k)
1  for i = 0 to m - 1
2      j = h(k, i)
3      if T[j].state == EMPTY
4          return NIL                 // the probe sequence ends: k is absent
5      if T[j].state == OCCUPIED and T[j].key == k
6          return j
7  return NIL                         // DELETED slots are walked past

OPEN-INSERT(T, k)
1  tomb = NIL
2  for i = 0 to m - 1
3      j = h(k, i)
4      if T[j].state == OCCUPIED and T[j].key == k
5          return j                   // already present
6      if T[j].state == DELETED and tomb == NIL
7          tomb = j                   // remember the first reusable slot
8      if T[j].state == EMPTY
9          if tomb != NIL then j = tomb
10         T[j].key = k; T[j].state = OCCUPIED
11         return j
12 error "table full"

OPEN-DELETE(T, k)
1  j = OPEN-SEARCH(T, k)
2  if j != NIL
3      T[j].state = DELETED           // never EMPTY
```

Notice line 8 of `OPEN-INSERT`. Even after finding a tombstone, insertion keeps walking until it reaches an EMPTY slot, because only then does it know that k is not stored further along; stopping at the tombstone could store a second copy of k. The last lines of the output show this: inserting 1067 walks 11, 12, 13, 14, 15, five probes, and then stores 1067 in the tombstone at slot 12.

Tombstones have a cost. They never shorten a search, so a table with many deletions slows down even when it holds few keys. Real implementations count tombstones along with keys when they compute the load, and rebuild the table (a rehash to the same size) when tombstones pile up. Linear probing also admits a tombstone-free deletion, *backward-shift deletion*, which moves later keys of the run back into the gap; Exercise 15.11 asks you to think it through.

15.6.6 The program

Listing 15.6 implements the three probe sequences Equation (15.14), Equation (15.15) and Equation (15.16) in one class, parameterized by the mode, with the tombstones of **Pseudocode 15.3**. The member `verbose_` prints each probe sequence as it happens.

Listing 15.6 `open_addressing.cpp`: linear probing, quadratic probing and double hashing in a table of 16 slots, with deletion by tombstones.

```
// open_addressing.cpp -- open addressing with linear probing, quadratic probing
// and double hashing (Section 15.6), with tombstones for deletion.
#include <cstdio>
#include <vector>

enum Probe { LINEAR, QUADRATIC, DOUBLE };
enum State { EMPTY, OCCUPIED, DELETED };             // DELETED = tombstone

class OpenTable {
public:
    OpenTable(int m, Probe mode, bool verbose = true)
        : m_(m), mode_(mode), verbose_(verbose), key_(m, 0), state_(m, EMPTY) {}

    // The i-th slot tried for key k, i = 0, 1, 2, ... (m is a power of 2).
    int slot(long long k, int i) const {
        long long h1 = k % m_;
        switch (mode_) {
        case LINEAR:    return (int)((h1 + i) % m_);
        case QUADRATIC: return (int)((h1 + (long long)i * (i + 1) / 2) % m_);
        default: {
            long long h2 = 1 + 2 * ((k / m_) % (m_ / 2));   // odd, so coprime to m
            return (int)((h1 + i * h2) % m_);
        }
        }
    }

    void setVerbose(bool v) { verbose_ = v; }

    // Insert k; print the slots tried. Returns the number of probes.
    int insert(long long k) {
        int firstTomb = -1;
        if (verbose_) std::printf("  insert %lld:", k);
        for (int i = 0; i < m_; ++i) {
            int s = slot(k, i);
            if (verbose_) std::printf(" %d", s);
            if (state_[s] == OCCUPIED && key_[s] == k) {
                if (verbose_) std::printf("  already present\n");
                return i + 1;
            }
            if (state_[s] == DELETED && firstTomb < 0) firstTomb = s;
            if (state_[s] == EMPTY) {                // k is certainly absent
                int target = firstTomb >= 0 ? firstTomb : s;
                key_[target] = k;
                state_[target] = OCCUPIED;
                if (verbose_)
                    std::printf("  -> slot %d (%d probe%s)\n", target, i + 1,
```

Listing 15.6 (Continued) *open_addressing.cpp: linear probing, quadratic probing and double hashing in a table of 16 slots, with deletion by tombstones.*

```
                                i == 0 ? "" : "s");
                return i + 1;
            }
        }
        if (verbose_) std::printf("  table full\n");
        return m_;
    }

    // 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;
    }

    bool eraseWrongly(long long k) {                 // just empties the slot: a bug
        int probes, s = find(k, probes);
        if (s < 0) return false;
        state_[s] = EMPTY;
        return true;
    }

    void show() const {
        for (int row = 0; row < m_; row += 8) {
            std::printf("  %2d-%-2d:", row, row + 7);
            for (int s = row; s < row + 8; ++s) {
                if (state_[s] == OCCUPIED) std::printf(" %4lld", key_[s]);
                else std::printf(" %4s", state_[s] == DELETED ? "DEL" : "--");
            }
            std::printf("\n");
        }
    }
```

Listing 15.6 (Continued) *open_addressing.cpp: linear probing, quadratic probing and double hashing in a table of 16 slots, with deletion by tombstones.*

```
private:
    int m_;
    Probe mode_;
    bool verbose_;                                  // print each probe sequence
    std::vector<long long> key_;
    std::vector<State> state_;
};

void search(const OpenTable &t, long long k) {
    int probes, s = t.find(k, probes);
    if (s >= 0) std::printf("  find %lld: slot %d, %d probe(s)\n", k, s, probes);
    else std::printf("  find %lld: not found, %d probe(s)\n", k, probes);
}

int main() {
    const long long keys[8] = {1003, 1019, 1035, 1012, 1028, 1013, 1044, 1051};
    const char *name[3] = {"Linear probing: h(k,i) = (k + i) mod 16",
                           "Quadratic probing: h(k,i) = (k + i(i+1)/2) mod 16",
                           "Double hashing: h(k,i) = (k + i*h2(k)) mod 16"};
    for (int mode = LINEAR; mode <= DOUBLE; ++mode) {
        OpenTable t(16, (Probe)mode);
        std::printf("%d. %s\n", mode + 1, name[mode]);
        int total = 0;
        for (long long k : keys) total += t.insert(k);
        t.show();
        std::printf("  total probes for 8 insertions: %d\n", total);
        search(t, 1051);
        search(t, 1067);                            // absent, 1067 mod 16 = 11
        std::printf("\n");
    }

    std::printf("4. Deleting from the linear-probing table\n");
    OpenTable a(16, LINEAR, false), b(16, LINEAR, false);
    for (long long k : keys) {
        a.insert(k);
        b.insert(k);
    }
    std::printf("  with a tombstone: erase 1019\n");
    a.erase(1019);
    a.show();
    search(a, 1035);
    std::printf("  by emptying the slot (wrong): erase 1019\n");
    b.eraseWrongly(1019);
    b.show();
    search(b, 1035);
    std::printf("  a new key may reuse the tombstone:\n");
```

Listing 15.6 (Continued) *open_addressing.cpp: linear probing, quadratic probing and double hashing in a table of 16 slots, with deletion by tombstones.*

```
    a.setVerbose(true);
    a.insert(1067);
    a.show();
    return 0;
}
```

```
1. Linear probing: h(k,i) = (k + i) mod 16
  insert 1003: 11  -> slot 11 (1 probe)
  insert 1019: 11 12  -> slot 12 (2 probes)
  insert 1035: 11 12 13  -> slot 13 (3 probes)
  insert 1012: 4  -> slot 4 (1 probe)
  insert 1028: 4 5  -> slot 5 (2 probes)
  insert 1013: 5 6  -> slot 6 (2 probes)
  insert 1044: 4 5 6 7  -> slot 7 (4 probes)
  insert 1051: 11 12 13 14  -> slot 14 (4 probes)
   0-7 :   --   --   --   -- 1012 1028 1013 1044
   8-15:   --   --   -- 1003 1019 1035 1051   --
  total probes for 8 insertions: 19
  find 1051: slot 14, 4 probe(s)
  find 1067: not found, 5 probe(s)

2. Quadratic probing: h(k,i) = (k + i(i+1)/2) mod 16
  insert 1003: 11  -> slot 11 (1 probe)
  insert 1019: 11 12  -> slot 12 (2 probes)
  insert 1035: 11 12 14  -> slot 14 (3 probes)
  insert 1012: 4  -> slot 4 (1 probe)
  insert 1028: 4 5  -> slot 5 (2 probes)
  insert 1013: 5 6  -> slot 6 (2 probes)
  insert 1044: 4 5 7  -> slot 7 (3 probes)
  insert 1051: 11 12 14 1  -> slot 1 (4 probes)
   0-7 :   -- 1051   --   -- 1012 1028 1013 1044
   8-15:   --   --   -- 1003 1019   -- 1035   --
  total probes for 8 insertions: 18
  find 1051: slot 1, 4 probe(s)
  find 1067: not found, 6 probe(s)

3. Double hashing: h(k,i) = (k + i*h2(k)) mod 16
  insert 1003: 11  -> slot 11 (1 probe)
  insert 1019: 11 10  -> slot 10 (2 probes)
  insert 1035: 11 12  -> slot 12 (2 probes)
  insert 1012: 4  -> slot 4 (1 probe)
  insert 1028: 4 5  -> slot 5 (2 probes)
  insert 1013: 5 4 3  -> slot 3 (3 probes)
  insert 1044: 4 7  -> slot 7 (2 probes)
  insert 1051: 11 14  -> slot 14 (2 probes)
   0-7 :   --   --   -- 1013 1012 1028   -- 1044
   8-15:   --   -- 1019 1003 1035   -- 1051   --
  total probes for 8 insertions: 15
  find 1051: slot 14, 2 probe(s)
  find 1067: not found, 2 probe(s)

4. Deleting from the linear-probing table
  with a tombstone: erase 1019
   0-7 :   --   --   --   -- 1012 1028 1013 1044
   8-15:   --   --   -- 1003  DEL 1035 1051   --
  find 1035: slot 13, 3 probe(s)
  by emptying the slot (wrong): erase 1019
   0-7 :   --   --   --   -- 1012 1028 1013 1044
   8-15:   --   --   -- 1003   -- 1035 1051   --
  find 1035: not found, 2 probe(s)
  a new key may reuse the tombstone:
  insert 1067: 11 12 13 14 15  -> slot 12 (5 probes)
   0-7 :   --   --   --   -- 1012 1028 1013 1044
   8-15:   --   --   -- 1003 1067 1035 1051   --
```

The totals summarize the three methods on this crowded example: 19 probes to insert the eight keys by linear probing, 18 by quadratic probing, 15 by double hashing. The absent key 1067, whose home is the crowded slot 11, costs 5, 6 and 2 probes. Eight keys prove nothing statistically, but the ranking is the one Section 15.7 confirms on 16,384 slots. **Listing 15.7** and **Listing 15.8** print the same output.

Listing 15.7 `OpenAddressing.java`: the three probe sequences and tombstones in Java.

```
// OpenAddressing.java -- open addressing with linear probing, quadratic probing
// and double hashing (Section 15.6), with tombstones for deletion.
public class OpenAddressing {
    enum Probe { LINEAR, QUADRATIC, DOUBLE }

    static final int EMPTY = 0, OCCUPIED = 1, DELETED = 2;   // DELETED = tombstone

    static class OpenTable {
        private final int m;
        private final Probe mode;
        private boolean verbose;                    // print each probe sequence
        private final long[] key;
        private final int[] state;

        OpenTable(int m, Probe mode, boolean verbose) {
            this.m = m;
            this.mode = mode;
            this.verbose = verbose;
            key = new long[m];
            state = new int[m];                     // all EMPTY
        }

        void setVerbose(boolean v) { verbose = v; }

        // The i-th slot tried for key k, i = 0, 1, 2, ... (m is a power of 2).
        int slot(long k, int i) {
            long h1 = k % m;
            switch (mode) {
                case LINEAR:
                    return (int) ((h1 + i) % m);
                case QUADRATIC:
                    return (int) ((h1 + (long) i * (i + 1) / 2) % m);
                default:
                    long h2 = 1 + 2 * ((k / m) % (m / 2));   // odd: coprime to m
                    return (int) ((h1 + i * h2) % m);
            }
        }
```

Listing 15.7 (Continued) *OpenAddressing.java: the three probe sequences and tombstones in Java.*

```
    // Insert k; print the slots tried. Returns the number of probes.
    int insert(long k) {
        int firstTomb = -1;
        StringBuilder sb = new StringBuilder("  insert " + k + ":");
        for (int i = 0; i < m; i++) {
            int s = slot(k, i);
            sb.append(' ').append(s);
            if (state[s] == OCCUPIED && key[s] == k) {
                if (verbose) System.out.println(sb + "  already present");
                return i + 1;
            }
            if (state[s] == DELETED && firstTomb < 0) firstTomb = s;
            if (state[s] == EMPTY) {             // k is certainly absent
                int target = firstTomb >= 0 ? firstTomb : s;
                key[target] = k;
                state[target] = OCCUPIED;
                if (verbose)
                    System.out.printf("%s  -> slot %d (%d probe%s)%n", sb,
                                      target, i + 1, i == 0 ? "" : "s");
                return i + 1;
            }
        }
        if (verbose) System.out.println(sb + "  table full");
        return m;
    }

    // Return the slot holding k, or -1; probes[0] receives the probes used.
    int find(long k, int[] probes) {
        probes[0] = 0;
        for (int i = 0; i < m; i++) {
            int s = slot(k, i);
            probes[0] = 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;
    }

    boolean erase(long k) {                      // leave a tombstone
        int s = find(k, new int[1]);
        if (s < 0) return false;
        state[s] = DELETED;
        return true;
    }
```

Listing 15.7 (Continued) *OpenAddressing.java: the three probe sequences and tombstones in Java.*

```
        boolean eraseWrongly(long k) {              // just empties the slot: a bug
            int s = find(k, new int[1]);
            if (s < 0) return false;
            state[s] = EMPTY;
            return true;
        }

        void show() {
            for (int row = 0; row < m; row += 8) {
                String label = String.format("  %2d-%-2d:", row, row + 7);
                StringBuilder sb = new StringBuilder(label);
                for (int s = row; s < row + 8; s++) {
                    String cell = state[s] == OCCUPIED ? Long.toString(key[s])
                                : state[s] == DELETED ? "DEL" : "--";
                    sb.append(String.format(" %4s", cell));
                }
                System.out.println(sb);
            }
        }
    }

    static void search(OpenTable t, long k) {
        int[] probes = new int[1];
        int s = t.find(k, probes);
        if (s >= 0)
            System.out.printf("  find %d: slot %d, %d probe(s)%n", k, s, probes[0]);
        else
            System.out.printf("  find %d: not found, %d probe(s)%n", k, probes[0]);
    }

    public static void main(String[] args) {
        long[] keys = {1003, 1019, 1035, 1012, 1028, 1013, 1044, 1051};
        String[] name = {"Linear probing: h(k,i) = (k + i) mod 16",
                         "Quadratic probing: h(k,i) = (k + i(i+1)/2) mod 16",
                         "Double hashing: h(k,i) = (k + i*h2(k)) mod 16"};
        Probe[] modes = Probe.values();
        for (int mode = 0; mode < 3; mode++) {
            OpenTable t = new OpenTable(16, modes[mode], true);
            System.out.printf("%d. %s%n", mode + 1, name[mode]);
            int total = 0;
            for (long k : keys) total += t.insert(k);
            t.show();
            System.out.printf("  total probes for 8 insertions: %d%n", total);
            search(t, 1051);
            search(t, 1067);                        // absent, 1067 mod 16 = 11
            System.out.println();
```

Listing 15.7 (Continued) *OpenAddressing.java: the three probe sequences and tombstones in Java.*

```
        }

        System.out.println("4. Deleting from the linear-probing table");
        OpenTable a = new OpenTable(16, Probe.LINEAR, false);
        OpenTable b = new OpenTable(16, Probe.LINEAR, false);
        for (long k : keys) {
            a.insert(k);
            b.insert(k);
        }
        System.out.println("  with a tombstone: erase 1019");
        a.erase(1019);
        a.show();
        search(a, 1035);
        System.out.println("  by emptying the slot (wrong): erase 1019");
        b.eraseWrongly(1019);
        b.show();
        search(b, 1035);
        System.out.println("  a new key may reuse the tombstone:");
        a.setVerbose(true);
        a.insert(1067);
        a.show();
    }
}
```

Listing 15.8 `open_addressing.py`: the three probe sequences and tombstones in Python; `None` is not needed because the slot states are kept in their own list.

```
# open_addressing.py -- open addressing with linear probing, quadratic probing
# and double hashing (Section 15.6), with tombstones for deletion.
LINEAR, QUADRATIC, DOUBLE = 0, 1, 2
EMPTY, OCCUPIED, DELETED = 0, 1, 2                    # DELETED = tombstone


class OpenTable:
    def __init__(self, m, mode, verbose=True):
        self.m, self.mode, self.verbose = m, mode, verbose
        self.key = [0] * m
        self.state = [EMPTY] * m

    def slot(self, k, i):
        """The i-th slot tried for key k, i = 0, 1, 2, ... (m is a power of 2)."""
        h1 = k % self.m
        if self.mode == LINEAR:
            return (h1 + i) % self.m
        if self.mode == QUADRATIC:
            return (h1 + i * (i + 1) // 2) % self.m
        h2 = 1 + 2 * ((k // self.m) % (self.m // 2))   # odd, so coprime to m
        return (h1 + i * h2) % self.m

    def insert(self, k):
        """Insert k; print the slots tried. Returns the number of probes."""
        first_tomb = -1
        tried = "  insert %d:" % k
        for i in range(self.m):
            s = self.slot(k, i)
            tried += " %d" % s
            if self.state[s] == OCCUPIED and self.key[s] == k:
                if self.verbose:
                    print(tried + "  already present")
                return i + 1
            if self.state[s] == DELETED and first_tomb < 0:
                first_tomb = s
            if self.state[s] == EMPTY:                # k is certainly absent
                target = first_tomb if first_tomb >= 0 else s
                self.key[target] = k
                self.state[target] = OCCUPIED
                if self.verbose:
                    print("%s  -> slot %d (%d probe%s)"
                          % (tried, target, i + 1, "" if i == 0 else "s"))
                return i + 1
        if self.verbose:
            print(tried + "  table full")
        return self.m
```

Listing 15.8 (Continued) *open_addressing.py: the three probe sequences and tombstones in Python; None is not needed because the slot states are kept in their own list.*

```
    def find(self, k):
        """Return (slot holding k or -1, probes used)."""
        probes = 0
        for i in range(self.m):
            s = self.slot(k, i)
            probes = i + 1
            if self.state[s] == EMPTY:              # the probe sequence ends here
                return -1, probes
            if self.state[s] == OCCUPIED and self.key[s] == k:
                return s, probes
        return -1, probes                           # DELETED: we kept going

    def erase(self, k):                             # leave a tombstone
        s, _ = self.find(k)
        if s < 0:
            return False
        self.state[s] = DELETED
        return True

    def erase_wrongly(self, k):                     # just empties the slot: a bug
        s, _ = self.find(k)
        if s < 0:
            return False
        self.state[s] = EMPTY
        return True

    def show(self):
        for row in range(0, self.m, 8):
            cells = []
            for s in range(row, row + 8):
                if self.state[s] == OCCUPIED:
                    cells.append(" %4d" % self.key[s])
                else:
                    mark = "DEL" if self.state[s] == DELETED else "--"
                    cells.append(" %4s" % mark)
            print("  %2d-%-2d:" % (row, row + 7) + "".join(cells))


def search(t, k):
    s, probes = t.find(k)
    if s >= 0:
        print("  find %d: slot %d, %d probe(s)" % (k, s, probes))
    else:
        print("  find %d: not found, %d probe(s)" % (k, probes))
```

Listing 15.8 (Continued) *open_addressing.py: the three probe sequences and tombstones in Python; None is not needed because the slot states are kept in their own list.*

```
def main():
    keys = [1003, 1019, 1035, 1012, 1028, 1013, 1044, 1051]
    name = ["Linear probing: h(k,i) = (k + i) mod 16",
            "Quadratic probing: h(k,i) = (k + i(i+1)/2) mod 16",
            "Double hashing: h(k,i) = (k + i*h2(k)) mod 16"]
    for mode in (LINEAR, QUADRATIC, DOUBLE):
        t = OpenTable(16, mode)
        print("%d. %s" % (mode + 1, name[mode]))
        total = sum(t.insert(k) for k in keys)
        t.show()
        print("  total probes for 8 insertions: %d" % total)
        search(t, 1051)
        search(t, 1067)                              # absent, 1067 mod 16 = 11
        print()

    print("4. Deleting from the linear-probing table")
    a, b = OpenTable(16, LINEAR, False), OpenTable(16, LINEAR, False)
    for k in keys:
        a.insert(k)
        b.insert(k)
    print("  with a tombstone: erase 1019")
    a.erase(1019)
    a.show()
    search(a, 1035)
    print("  by emptying the slot (wrong): erase 1019")
    b.erase_wrongly(1019)
    b.show()
    search(b, 1035)
    print("  a new key may reuse the tombstone:")
    a.verbose = True
    a.insert(1067)
    a.show()


if __name__ == "__main__":
    main()
```

15.6.7 Chaining or open addressing?

Table 15.3 compares the two families. Neither wins everywhere, and both appear in the standard libraries: C++'s `std::unordered_map` and Java's `HashMap` chain (Sections 15.9 and 15.10), while Python's `dict` probes.

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

The last row explains a design decision of C++. The standard promises that a reference to an element of an `unordered_map` stays valid when the table rehashes, and it exposes a bucket interface (`bucket_count`, `bucket`, `begin(n)`); both are natural for chaining and nearly impossible for open addressing [4], [5]. That is why faster open-addressing tables live outside the standard library.

15.7 Load Factor and Rehashing, Measured

15.7.1 The one number that matters

Every cost we have derived depends on the table only through its load factor,

$$\alpha = \frac{n}{m}, \tag{15.17}$$

the number of stored keys per slot. For chaining α is the average chain length and may exceed 1; for open addressing it is the fraction of occupied slots and must stay below 1. A hash table's promise of $O(1)$ is really a promise that α stays bounded by a constant, and keeping that promise is the table's own responsibility, because the user only ever inserts.

15.7.2 What the theory predicts

For chaining, Equation (15.11) and Equation (15.12) give α and $1 + \alpha/2$ key comparisons.

For open addressing we first analyze the ideal, *uniform hashing*, in which each probe sequence is a random ordering of the slots, the behavior that double hashing approximates. In an unsuccessful search the first probe hits an occupied slot with probability $n/m = \alpha$; given that, the second probe hits one of the remaining $n - 1$ occupied slots among the remaining $m - 1$ with probability $(n-1)/(m-1) \le \alpha$; and so on. The search makes an i-th probe only if the first $i - 1$ all hit occupied slots, which happens with probability at most α^{i-1}. Adding up the probabilities of making each probe,

$$E[\text{probes, unsuccessful}] \le 1 + \alpha + \alpha^2 + \cdots = \frac{1}{1-\alpha}. \tag{15.18}$$

A successful search for a key follows the same probes as the insertion that stored it. If the key was the $(i+1)$-st inserted, the table then held i keys, and by Equation (15.18) the insertion took at most $1/(1 - i/m)$ probes. Averaging over the n keys and replacing the sum by an integral,

$$E[\text{probes, successful}] \le \frac{1}{n}\sum_{i=0}^{n-1} \frac{1}{1 - i/m} \approx \frac{1}{\alpha}\int_0^{\alpha} \frac{dx}{1-x} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}. \tag{15.19}$$

Linear probing is worse because of primary clustering, and its analysis is one of the famous results of the field. Knuth derived it in 1962, in his first analysis of an algorithm, and published it in *The Art of Computer Programming* [1]:

$$E[\text{probes, unsuccessful}] \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right), \tag{15.20}$$

$$E[\text{probes, successful}] \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right). \tag{15.21}$$

The square in the unsuccessful cost is primary clustering expressed as a formula. At $\alpha = 0.5$ it is harmless, 2.5 probes against 2 for uniform hashing; at $\alpha = 0.9$ it is 50.5 probes against 10.

15.7.3 The measurements

Listing 15.9 tests all of these predictions. It fills a table of $m = 2^{14} = 16384$ slots with random 62-bit keys up to thirteen load factors, measures the average probes of a successful search (for every stored key) and of an unsuccessful search (for 10,000 fresh keys), for chaining and for the three probe sequences of Section 15.6, and prints the predictions of Equation (15.11), Equation (15.12), Equation (15.18), Equation (15.19), Equation (15.20) and Equation (15.21) next to them. It then looks at the distribution of chain lengths, and finally grows a chained table from 8 buckets to 200,000 keys, counting the keys moved by rehashing.

Listing 15.9 `load_factor.cpp`: probes per search against the load factor, measured and predicted; the distribution of chain lengths; and the cost of rehashing by doubling.

```
// load_factor.cpp -- measuring what the load factor does (Section 15.7):
// probes per search for chaining and three kinds of open addressing, the
// distribution of chain lengths, and the cost of rehashing as a table grows.
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <vector>

typedef std::uint64_t u64;

u64 rngState = 7;                                   // SplitMix64, fixed seed
u64 nextKey() {                                     // a random 62-bit key
    u64 z = (rngState += 0x9E3779B97F4A7C15ULL);
    z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9ULL;
    z = (z ^ (z >> 27)) * 0x94D049BB133111EBULL;
    return (z ^ (z >> 31)) >> 2;
}

const int P = 14, M = 1 << P;                       // m = 16384 slots

// ---------- open addressing: 0 = linear, 1 = quadratic, 2 = double hashing
u64 slotOf(u64 k, u64 i, int mode) {
    u64 h1 = k % M;
    if (mode == 0) return (h1 + i) % M;
    if (mode == 1) return (h1 + i * (i + 1) / 2) % M;
    u64 h2 = 1 + 2 * ((k >> P) % (M / 2));          // odd, so coprime to M
    return (h1 + i * h2) % M;
}

// Number of probes to find k (present) or to reach an empty slot (absent).
int probes(const std::vector<u64> &t, const std::vector<char> &used, u64 k,
           int mode) {
    for (u64 i = 0;; ++i) {
        u64 s = slotOf(k, i, mode);
        if (!used[s] || t[s] == k) return (int)i + 1;
    }
}
```

Listing 15.9 (Continued) *load_factor.cpp: probes per search against the load factor, measured and predicted; the distribution of chain lengths; and the cost of rehashing by doubling.*

```
void measureOpen(int n, int mode, double &succ, double &unsucc) {
    std::vector<u64> t(M, 0), keys(n);
    std::vector<char> used(M, 0);
    for (int j = 0; j < n; ++j) {
        u64 k = keys[j] = nextKey();
        for (u64 i = 0;; ++i) {
            u64 s = slotOf(k, i, mode);
            if (!used[s]) { t[s] = k; used[s] = 1; break; }
        }
    }
    long long sum = 0;
    for (u64 k : keys) sum += probes(t, used, k, mode);
    succ = (double)sum / n;
    sum = 0;
    for (int j = 0; j < 10000; ++j) sum += probes(t, used, nextKey(), mode);
    unsucc = sum / 10000.0;
}

// ---------- separate chaining: count key comparisons
void measureChain(int n, double &succ, double &unsucc) {
    std::vector<std::vector<u64>> b(M);
    std::vector<u64> keys(n);
    for (int j = 0; j < n; ++j) b[(keys[j] = nextKey()) % M].push_back(keys[j]);
    long long sum = 0;
    for (u64 k : keys) {                              // position in its chain
        const std::vector<u64> &c = b[k % M];
        for (std::size_t p = 0; p < c.size(); ++p)
            if (c[p] == k) { sum += (long long)p + 1; break; }
    }
    succ = (double)sum / n;
    sum = 0;
    for (int j = 0; j < 10000; ++j) sum += (long long)b[nextKey() % M].size();
    unsucc = sum / 10000.0;
}

int main() {
    std::printf("1. Average probes (key comparisons for chaining), m = %d\n", M);
    std::printf("          chaining      linear     quadratic     double\n");
    std::printf("  alpha  succ unsucc  succ unsucc  succ unsucc  succ unsucc\n");
    const double alphas[13] = {0.10, 0.25, 0.50, 0.60, 0.70, 0.75, 0.80, 0.85,
                               0.90, 0.95, 1.50, 2.00, 3.00};
    for (double a : alphas) {
        int n = (int)std::lround(a * M);
        double s, u;
        measureChain(n, s, u);
        std::printf("  %4.2f %5.2f %6.2f", a, s, u);
```

Listing 15.9 (Continued) *load_factor.cpp: probes per search against the load factor, measured and predicted; the distribution of chain lengths; and the cost of rehashing by doubling.*

```
        for (int mode = 0; mode < 3 && a < 1.0; ++mode) {
            measureOpen(n, mode, s, u);
            std::printf(" %5.2f %6.2f", s, u);
        }
        std::printf("\n");
    }

    std::printf("\n2. The same quantities predicted by the formulas of "
                "Section 15.7\n");
    std::printf("              chaining           linear          uniform\n");
    std::printf("  alpha   succ unsucc    succ   unsucc    succ   unsucc\n");
    for (double a : alphas) {
        std::printf("  %5.2f %6.3f %6.3f", a, 1.0 + a / 2.0, a);
        if (a < 1.0)
            std::printf(" %7.3f %8.3f %7.3f %8.3f", 0.5 * (1.0 + 1.0 / (1.0 - a)),
                        0.5 * (1.0 + 1.0 / ((1.0 - a) * (1.0 - a))),
                        std::log(1.0 / (1.0 - a)) / a, 1.0 / (1.0 - a));
        std::printf("\n");
    }

    std::printf("\n3. Chain lengths for n = m = %d random keys\n", M);
    std::vector<int> len(M, 0);
    for (int j = 0; j < M; ++j) ++len[nextKey() % M];
    std::vector<int> hist(9, 0);
    int longest = 0;
    for (int L : len) {
        ++hist[L < 8 ? L : 8];
        if (L > longest) longest = L;
    }
    std::printf("  length  buckets  Poisson(1) prediction\n");
    double fact = 1.0, tail = M;
    for (int L = 0; L <= 8; ++L) {
        if (L > 0) fact *= L;
        double expect = M * std::exp(-1.0) / fact;
        if (L < 8) {
            tail -= expect;
            std::printf("  %6d %8d %12.1f\n", L, hist[L], expect);
        } else {
            std::printf("  %5d+ %8d %12.1f\n", L, hist[L], tail);
        }
    }
    std::printf("  longest chain: %d\n", longest);
```

Listing 15.9 (Continued) *load_factor.cpp: probes per search against the load factor, measured and predicted; the distribution of chain lengths; and the cost of rehashing by doubling.*

```
    std::printf("\n4. Growing a chained table from m = 8, doubling when "
                "n/m > 0.75\n");
    std::vector<std::vector<u64>> b(8);
    long long n = 0, moved = 0;
    int rehashes = 0;
    for (int j = 0; j < 200000; ++j) {
        u64 k = nextKey();
        b[k % b.size()].push_back(k);
        ++n;
        if ((double)n / b.size() > 0.75) {           // rehash: every key moves
            std::vector<std::vector<u64>> nb(2 * b.size());
            for (const std::vector<u64> &chain : b)
                for (u64 x : chain) {
                    nb[x % nb.size()].push_back(x);
                    ++moved;
                }
            b.swap(nb);
            ++rehashes;
        }
    }
    std::size_t maxChain = 0;
    for (const std::vector<u64> &chain : b)
        if (chain.size() > maxChain) maxChain = chain.size();
    std::printf("  keys inserted: %lld, rehashes: %d, final m = %zu, load = %.3f\n",
                n, rehashes, b.size(), (double)n / b.size());
    std::printf("  keys moved by all rehashes: %lld (%.3f per insertion)\n", moved,
                (double)moved / n);
    std::printf("  longest chain at the end: %zu\n", maxChain);
    return 0;
}
```

```
1. Average probes (key comparisons for chaining), m = 16384
         chaining      linear     quadratic     double
  alpha  succ unsucc  succ unsucc  succ unsucc  succ unsucc
  0.10  1.05   0.10  1.06   1.12  1.06   1.12  1.05   1.11
  0.25  1.13   0.24  1.18   1.38  1.15   1.36  1.16   1.33
  0.50  1.25   0.50  1.51   2.52  1.43   2.15  1.39   2.03
  0.60  1.30   0.60  1.76   3.65  1.61   2.76  1.51   2.48
  0.70  1.34   0.70  2.16   5.97  1.87   3.82  1.72   3.37
  0.75  1.38   0.76  2.50   8.81  2.01   4.56  1.86   4.01
  0.80  1.39   0.80  3.18  13.51  2.18   5.94  2.01   4.95
  0.85  1.42   0.85  3.50  20.41  2.45   7.85  2.24   6.66
  0.90  1.46   0.90  5.60  50.07  2.81  11.74  2.56   9.87
  0.95  1.48   0.95  8.45 123.04  3.63  23.92  3.11  19.93
  1.50  1.74   1.49
  2.00  2.02   2.02
  3.00  2.50   3.02

2. The same quantities predicted by the formulas of Section 15.7
              chaining           linear          uniform
  alpha   succ unsucc    succ   unsucc    succ   unsucc
   0.10  1.050  0.100   1.056    1.117   1.054    1.111
   0.25  1.125  0.250   1.167    1.389   1.151    1.333
   0.50  1.250  0.500   1.500    2.500   1.386    2.000
   0.60  1.300  0.600   1.750    3.625   1.527    2.500
   0.70  1.350  0.700   2.167    6.056   1.720    3.333
   0.75  1.375  0.750   2.500    8.500   1.848    4.000
   0.80  1.400  0.800   3.000   13.000   2.012    5.000
   0.85  1.425  0.850   3.833   22.722   2.232    6.667
   0.90  1.450  0.900   5.500   50.500   2.558   10.000
   0.95  1.475  0.950  10.500  200.500   3.153   20.000
   1.50  1.750  1.500
   2.00  2.000  2.000
   3.00  2.500  3.000

3. Chain lengths for n = m = 16384 random keys
  length  buckets  Poisson(1) prediction
       0     6108       6027.3
       1     5924       6027.3
       2     3008       3013.7
       3     1002       1004.6
       4      282        251.1
       5       51         50.2
       6        8          8.4
       7        1          1.2
      8+        0          0.2
  longest chain: 7

4. Growing a chained table from m = 8, doubling when n/m > 0.75
  keys inserted: 200000, rehashes: 16, final m = 524288, load = 0.381
  keys moved by all rehashes: 393226 (1.966 per insertion)
  longest chain at the end: 6
```

Figure 15.6 draws the first two tables: the markers are measurements, the curves are the formulas.

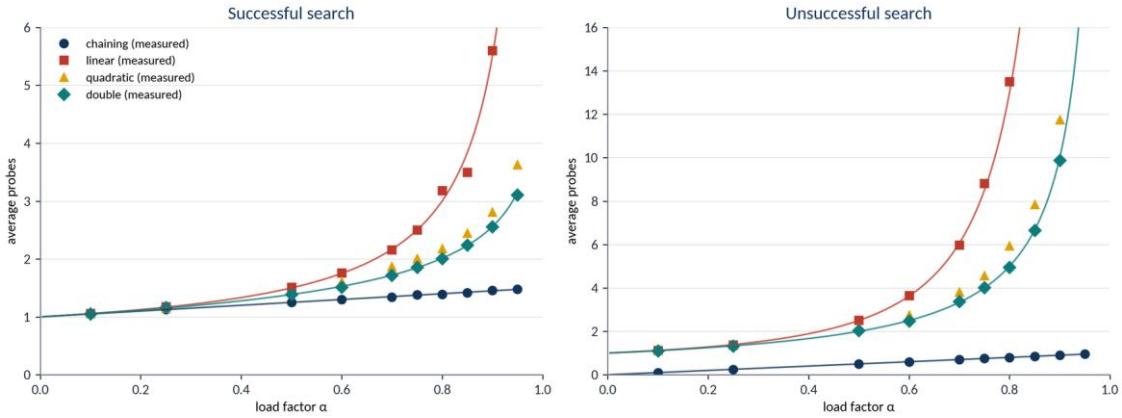


Figure 15.6 Average probes per search in a table of 16,384 slots, measured by **Listing 15.9** (markers) and predicted by Equation (15.12), Equation (15.11), Equation (15.20), Equation (15.19) and Equation (15.18) (curves). Chaining stays flat; linear probing explodes first; double hashing follows the uniform-hashing curve.

Four lessons are visible in the numbers.

- **The theory is right.** Chaining matches $1 + \alpha/2$ and α to within 0.02 at every load. Double hashing matches the uniform-hashing formulas closely (1.86 measured against 1.848 predicted at $\alpha = 0.75$; 9.87 against 10.0 unsuccessful at $\alpha = 0.9$). Linear probing matches Equation (15.20) up to about $\alpha = 0.9$ (50.07 against 50.5).
- **Near-full linear probing is unpredictable.** At $\alpha = 0.95$ the measured 123 probes are well below the predicted 200.5, and another seed would give a quite different figure: the average is dominated by the few absent keys that land at the start of the longest runs, and Equation (15.20) is itself a limit for infinitely large tables. The lesson is not the exact number but the warning: never run linear probing that full.
- **Quadratic probing sits between.** It removes primary clustering, so it follows double hashing closely until about $\alpha = 0.8$, then pays for secondary clustering.
- **Chaining degrades gracefully.** At $\alpha = 3$, three keys per bucket, a successful search still costs only 2.5 comparisons. Open addressing cannot even reach $\alpha = 1$.

This is why real tables pick a maximum load factor and rehash when they reach it: 1.0 for `std::unordered_map` [5], 0.75 for Java's `HashMap` [6], 2/3 for Python's `dict` [7], and typically 0.5 to 0.875 for tables with open addressing, depending on the probe sequence.

15.7.4 How chain lengths are distributed

The *average* chain length is α, but how often does a bucket hold 5 or 7 keys? Under Equation (15.1) the number of keys in a given bucket is a sum of n independent trials, each succeeding with probability $1/m$: a binomial distribution. When n and m are large with $n/m = \alpha$ fixed, it approaches the *Poisson distribution*:

$$\Pr[\text{a bucket holds exactly } j \text{ keys}] \approx \frac{e^{-\alpha}\alpha^{j}}{j!}. \tag{15.22}$$

For $\alpha = 1$ the fractions are $e^{-1} \approx 0.368$ for $j = 0$ and for $j = 1$, $e^{-1}/2 \approx 0.184$ for $j = 2$, $e^{-1}/6 \approx 0.061$ for $j = 3$, and they fall off faster than exponentially after that. Part 3 of the output counts the buckets of a table with $n = m = 16384$ and **Figure 15.7** sets them beside Equation (15.22): 6108 empty buckets against 6027 predicted, 3008 of length 2 against 3014, and a longest chain of 7, where the prediction for 8 or more is 0.2 buckets.

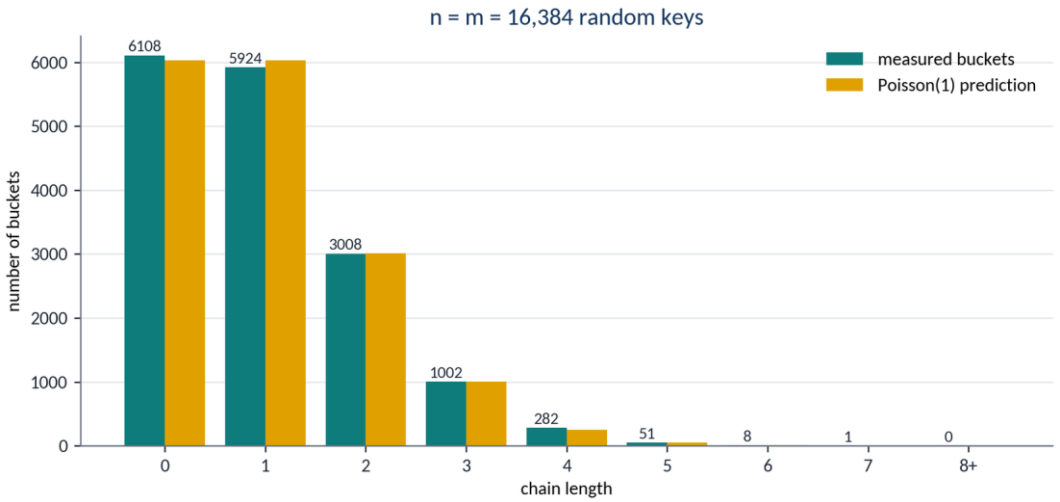


Figure 15.7 Chain lengths for 16,384 random keys in 16,384 buckets (teal) against the Poisson prediction Equation (15.22) with $\alpha = 1$ (amber). More than a third of the buckets are empty; none holds more than 7 keys.

Equation (15.22) explains a detail of Java's `HashMap` that Section 15.10 describes: a bucket is converted into a tree only when it reaches 8 entries, because with random hash codes and the default load of at most 0.75 a bucket of 8 has a probability of about one in ten million. The comment in the JDK source makes exactly this argument [8].

15.7.5 Rehashing, and why doubling is cheap

To keep α bounded, a table that crosses its maximum load allocates a larger array and reinserts every key, because the slot of each key depends on m through $h(k)$ mod m. This is *rehashing*. One rehash costs $\Theta(n)$, which sounds alarming: are we back to linear time?

No, provided the table grows by a constant *factor*. Suppose it doubles whenever n exceeds $\alpha_{\max} m$, starting from m_0 buckets. The rehashes happen at sizes of roughly $\alpha_{\max} m_0, 2\alpha_{\max} m_0, 4\alpha_{\max} m_0, \ldots$, each moving all the keys present, and the last one happens with at most n keys. Read backward from the last rehash, the numbers of keys moved are at most $n, n/2, n/4, \ldots$, so

$$\text{total keys moved} \le n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) < 2n. \tag{15.23}$$

Spread over the n insertions, rehashing adds fewer than two moves per insertion: $O(1)$ *amortized*, the same argument that makes `std::vector::push_back` cheap (Chapter 1). Part 4 of the output measures it: 200,000 insertions, 16 rehashes, 393,226 keys moved, 1.966 moves per insertion, just under the bound of 2. The exact count can be checked by hand. With $\alpha_{\max} = 0.75$ a table of m buckets rehashes when its $(0.75m + 1)$-st key arrives, so the tables of 8, 16, …, 262,144 buckets moved $0.75(8 + 16 + \cdots + 262144) + 16 = 0.75 \cdot 524280 + 16 = 393226$ keys.

Growing by a constant *amount* instead, say 1000 buckets at a time, would be a disaster: the table would rehash every few hundred insertions, and the total work would be $\Theta(n^2)$ (Exercise 15.13). Two practical refinements are common. Tables that must never pause for long (a game loop, a network switch) rehash *incrementally*, moving a few buckets at each operation while two arrays coexist. And a program that knows how many keys are coming should say so in advance with `reserve(n)` in C++ or the capacity argument of `new HashMap<>(capacity)` in Java, so that no rehash happens at all.

15.8 Complexity, Worst Cases and Hash Flooding

15.8.1 The complete bill

Table 15.4 states what a hash table costs, assuming a hash function that is computed in $O(1)$ time (for a string key of length L, add $O(L)$ to every operation, for hashing it and comparing it).

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	$\Theta(n + m)$	$\Theta(m)$	$\Theta(n + m)$

The worst case is always the same picture: every key hashes to the same slot. The chained table degenerates into one linked list of Chapter 7, the probed table into one long run, and inserting n keys costs $0 + 1 + \cdots + (n-1) = n(n-1)/2$ comparisons. With a fixed, well-designed hash function and ordinary data this essentially never happens by accident. The danger is that it can happen on purpose.

15.8.2 Hash flooding

If an attacker knows the hash function, the attacker can choose keys that all collide and send them to a program that stores them in a hash table. Scott Crosby and Dan Wallach described these *algorithmic complexity attacks* in 2003, slowing the hash tables of the Perl language, the Squid web proxy and the Bro intrusion-detection system to a crawl with modest amounts of crafted input [9]. In December 2011 Alexander Klink and Julian Wälde showed that almost every web platform of the day (PHP, ASP.NET, Java application servers, Python, Ruby and others) stored the parameters of an HTTP request in a hash table with a public, unkeyed hash function, so that one crafted request could keep a server's processor busy for minutes [10]. The attack became known as *hash flooding* or HashDoS.

Building the colliding keys is surprisingly easy for the polynomial hash Equation (15.5) with $B = 31$. The two-letter strings "Aa" and "BB" collide: $65 \cdot 31 + 97 = 2112 = 66 \cdot 31 + 66$. And if two blocks of equal length collide, then so do any two strings built from those blocks in the same positions, since by Equation (15.5) $h(xy) = h(x) \cdot 31^{|y|} + h(y)$. With 11 blocks, each "Aa" or "BB", we get $2^{11} = 2048$ different 22-letter strings with one single hash value. **Listing 15.10** builds them and inserts them into a chained table of 4096 buckets, next to 2048 ordinary keys, first with the public hash and then with a *keyed* hash.

Listing 15.10 `flooding.cpp`: 2048 keys that all collide under the public 31-based polynomial hash, and a keyed polynomial hash with a secret base that spreads them again.

```cpp
// flooding.cpp -- hash flooding (Section 15.8): keys built so that they all
// collide under a fixed, publicly known hash function, and a keyed (seeded)
// hash function that the same keys cannot defeat.
#include <cstdint>
#include <cstdio>
#include <string>
#include <vector>

// The 32-bit polynomial hash with base 31 (the formula of Java's String).
std::uint32_t hash31(const std::string &s) {
    std::uint32_t h = 0;
    for (unsigned char c : s) h = h * 31 + c;
    return h;
}

// A keyed polynomial hash modulo the prime 2^31 - 1, with a secret base r.
const std::uint64_t PRIME = 2147483647ULL;
std::uint64_t secretBase = 0;
std::uint32_t hashKeyed(const std::string &s) {
    std::uint64_t h = 0;
    for (unsigned char c : s) h = (h * secretBase + c) % PRIME;
    return (std::uint32_t)h;
}

// Insert every key into a chained table of m buckets (checking for duplicates,
// as put must), then search for every key. Report the work done.
void experiment(const char *label, const std::vector<std::string> &keys,
                std::uint32_t (*h)(const std::string &), std::size_t m) {
    std::vector<std::vector<std::string>> table(m);
    long long insertCmp = 0, searchCmp = 0;
    for (const std::string &k : keys) {
        std::vector<std::string> &chain = table[h(k) % m];
        bool present = false;
        for (const std::string &x : chain) {
            ++insertCmp;
            if (x == k) { present = true; break; }
        }
        if (!present) chain.push_back(k);
    }
    for (const std::string &k : keys) {
        for (const std::string &x : table[h(k) % m]) {
            ++searchCmp;
            if (x == k) break;
        }
    }
    std::size_t longest = 0;
```

Listing 15.10 (Continued) *flooding.cpp: 2048 keys that all collide under the public 31-based polynomial hash, and a keyed polynomial hash with a secret base that spreads them again.*

```
    for (const std::vector<std::string> &c : table)
        if (c.size() > longest) longest = c.size();
    std::printf("  %-28s %8zu %12lld %12lld\n", label, longest, insertCmp,
                searchCmp);
}

int main() {
    std::printf("1. Different strings, equal hash31 values\n");
    const char *pairs[6] = {"Aa", "BB", "AaAa", "AaBB", "BBAa", "BBBB"};
    for (const char *s : pairs)
        std::printf("  hash31(\"%s\") = %u\n", s, hash31(s));

    // 2^11 = 2048 attack keys: every string of 11 blocks, each "Aa" or "BB".
    std::vector<std::string> attack, ordinary;
    for (int mask = 0; mask < 2048; ++mask) {
        std::string s;
        for (int b = 10; b >= 0; --b) s += (mask >> b & 1) ? "BB" : "Aa";
        attack.push_back(s);
    }
    for (int i = 0; i < 2048; ++i) {
        char buf[24];
        std::snprintf(buf, sizeof buf, "reader%04d", i);
        ordinary.push_back(buf);
    }
    std::printf("  first attack keys: %s %s ...\n", attack[0].c_str(),
                attack[1].c_str());
    std::printf("  all 2048 attack keys hash to %u\n", hash31(attack[2047]));

    std::printf("\n2. n = 2048 keys into m = 4096 buckets\n");
    std::printf("  %-28s %8s %12s %12s\n", "keys and hash function", "longest",
                "insert cmp", "search cmp");
    experiment("ordinary keys, hash31", ordinary, hash31, 4096);
    experiment("attack keys, hash31", attack, hash31, 4096);
    secretBase = 1234567891ULL % PRIME;      // in real life: random at start
    experiment("attack keys, keyed hash", attack, hashKeyed, 4096);
    experiment("ordinary keys, keyed hash", ordinary, hashKeyed, 4096);
    return 0;
}
```

```
1. Different strings, equal hash31 values
  hash31("Aa") = 2112
  hash31("BB") = 2112
  hash31("AaAa") = 2031744
  hash31("AaBB") = 2031744
  hash31("BBAa") = 2031744
  hash31("BBBB") = 2031744
  first attack keys: AaAaAaAaAaAaAaAaAaAaAa AaAaAaAaAaAaAaAaAaAaBB ...
  all 2048 attack keys hash to 2738481856

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
  ordinary keys, keyed hash           3          752         2800
```

The attack keys build a single chain of 2048 nodes, and insertion costs 2,096,128 comparisons, exactly $2048 \cdot 2047/2$, against 935 for the same number of ordinary keys: more than two thousand times the work. With a million keys instead of two thousand the ratio would be half a million, and the table would take hours to fill.

15.8.3 Randomized hashing: the defense

The defense is to take the hash function out of the attacker's knowledge. Instead of one fixed function, the table uses a function chosen at random, when the program starts, from a large family. The theory behind this is Carter and Wegman's *universal hashing* of 1979 [11]:

> **DEFINITION.** A family $\mathcal{H}$ of hash functions from U to $\{0, \dots, m-1\}$ is *universal* if for every pair of distinct keys $x \neq y$,

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}, \tag{15.24}$$

the probability being over the random choice of h from $\mathcal{H}$. This is the collision bound of simple uniform hashing, but now it holds for *every* set of keys, including one chosen by an adversary who knows $\mathcal{H}$ but not which member was chosen. All the averages of this chapter, derived under Equation (15.1), then hold in expectation for every input [12].

The keyed hash of **Listing 15.10** is a simple family that comes close to Equation (15.24). It evaluates the polynomial of the string at a secret point r modulo the prime $p = 2^{31} - 1$:

$$h_r(s) = \left(\sum_{i=0}^{L-1} s_i \, r^{L-1-i} \right) \bmod p. \tag{15.25}$$

For two different strings x and y of the same length L, $h_r(x) - h_r(y)$ is a nonzero polynomial in r of degree at most $L-1$ over the field of integers modulo p, and such a polynomial has at most $L-1$ roots. So a random secret r makes x and y collide with probability at most $(L-1)/p$: for the 22-letter attack keys, about one in a hundred million. The attack keys were built for $r = 31$; with any other secret base they scatter, and the output shows a longest chain of 4 and 429 comparisons, less than for the ordinary keys.

Production systems use stronger keyed functions. SipHash, designed by Jean-Philippe Aumasson and Daniel J. Bernstein in 2012 as a direct answer to hash flooding, is a fast keyed pseudorandom function with a 128-bit key [13]. Python adopted it for strings and bytes in version 3.4 (PEP 456 [14]) and switched to its faster variant SipHash-1-3 in version 3.11 [15]; the key is drawn at random every time the interpreter starts, which is why Section 15.10.2 prints three different hash values for the same string. Java chose a different path, because `String.hashCode` is fixed by the language specification and many programs depend on its value: since Java 8 a `HashMap` bucket that collects too many entries is converted into a balanced tree, so that even a flood of colliding keys costs $O(\lg n)$ per operation instead of $O(n)$ [16]. And web servers began limiting the number of parameters a single request may carry.

> **THE HASH OF A LIBRARY IS NOT ALWAYS KEYED.** GCC's `std::hash<int>` is the identity (Section 15.3.6), and C++ does not require `std::hash` for strings to be randomized. A C++ program that stores keys chosen by strangers (request parameters, user names, file names from an upload) in an `unordered_map` should supply its own keyed hash function as the third template argument, exactly as Section 15.9 does for a two-field key.

15.9 `std::unordered_map` versus `std::map`

15.9.1 Same interface, different machines

C++ offers two associative containers that look almost identical in code. Both store key-value pairs, both have `operator[]`, `find`, `count`, `insert`, `erase` and `at`, and a program can often switch from one to the other by changing one word. Inside, they have nothing in common.

- `std::map` is a *balanced binary search tree*, in every major implementation the red-black tree of Section 11.9.3. It needs keys that can be compared with `<`. It keeps them in order, so iteration visits them from smallest to largest, and `lower_bound`, `upper_bound` and in-order neighbors are available. Every operation is $O(\lg n)$ in the worst case.
- `std::unordered_map` is a *hash table with separate chaining*: an array of buckets, each a chain of nodes. It needs a hash function (`std::hash<Key>` by default) and `==`. Its iteration order is unspecified. Its operations are $O(1)$ on average and $O(n)$ in the worst case [4].

The names are historical. `std::map` was part of the first C++ standard in 1998, where *map* meant any associative container; hash tables were added only in C++11, under the careful name *unordered* so as not to break existing programs that had defined their own `hash_map`. The consequence is a trap that every C++ programmer meets once: a program that uses `std::map` is using a *tree*, not a hash table, whatever the problem statement calls it. Chapter 16 reconstructs a travel itinerary from a pile of tickets with a map from each origin to its destination and a reverse map from each destination to its origin; written with `std::map` it is correct and runs in $O(n \lg n)$, and changing the one word to `std::unordered_map` makes it a hashing solution with expected $O(n)$ time.

Figure 15.8 shows both containers after the same eight book codes are inserted in the same shuffled order.

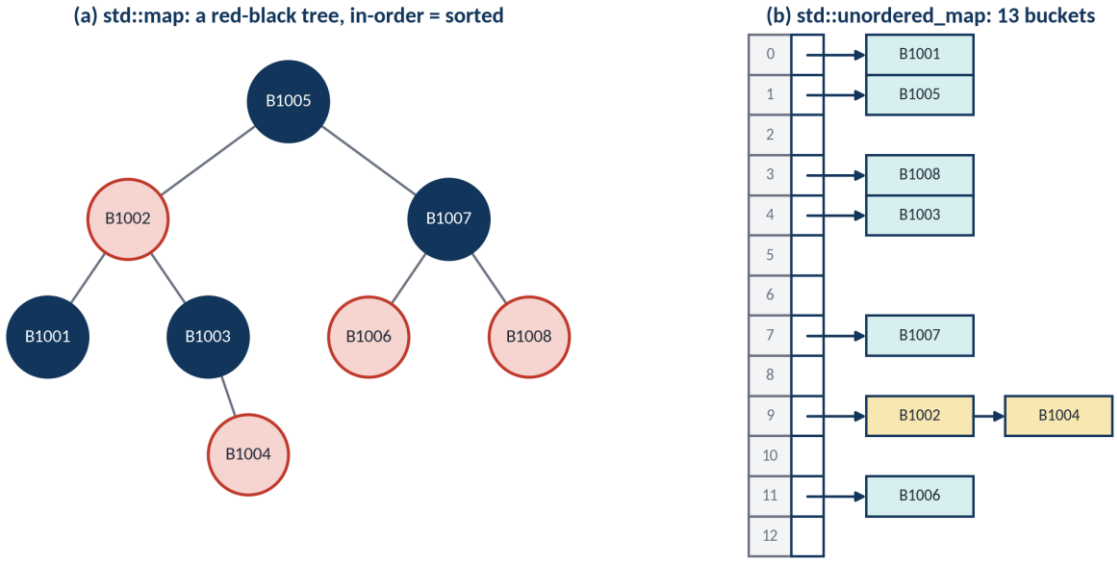


Figure 15.8 The eight codes inserted in the order B1005, B1001, B1008, B1003, B1002, B1007, B1004, B1006. (a) `std::map` keeps a red-black tree, whose in-order traversal is sorted. (b) `std::unordered_map` keeps 13 buckets; B1002 and B1004 share bucket 9, and no order survives.

15.9.2 The program

Listing 15.11 makes the difference concrete: iteration order, the bucket interface, questions that only the tree can answer, a user-defined key, and a stopwatch.

Listing 15.11 `maps.cpp`: `std::map` against `std::unordered_map` on the same data, the bucket interface, range queries, a hash function for a two-field key, and a timing of one million keys.

```
// maps.cpp -- std::map (a balanced binary search tree) against
// std::unordered_map (a hash table): ordering, buckets, range queries,
// a user-defined key, and a stopwatch (Section 15.9).
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <functional>
#include <map>
#include <random>
#include <string>
#include <unordered_map>
#include <vector>

struct Edition {                                    // a key made of two fields
    std::string title;
    int year;
    bool operator==(const Edition &o) const {
        return title == o.title && year == o.year;
    }
};

struct EditionHash {                                // how to hash an Edition
    std::size_t operator()(const Edition &e) const {
        std::size_t h = std::hash<std::string>{}(e.title);
        return h ^ (std::hash<int>{}(e.year) + 0x9E3779B97F4A7C15ULL + (h << 6)
                    + (h >> 2));                    // mix the two hashes
    }
};

template <typename F>
double ms(F f) {
    auto t0 = std::chrono::steady_clock::now();
    f();
    auto t1 = std::chrono::steady_clock::now();
    return std::chrono::duration<double, std::milli>(t1 - t0).count();
}
```

Listing 15.11 (Continued) *maps.cpp: std::map against std::unordered_map on the same data, the bucket interface, range queries, a hash function for a two-field key, and a timing of one million keys.*

```
int main() {
    const char *code[8] = {"B1005", "B1001", "B1008", "B1003",
                           "B1002", "B1007", "B1004", "B1006"};
    const char *title[8] = {"Algoritma & Struktur Data", "Laskar Pelangi",
                            "Negeri 5 Menara", "Filosofi Kopi", "Bumi Manusia",
                            "Fisika Dasar", "Sejarah Nusantara",
                            "Cerita Rakyat Nusantara"};
    std::map<std::string, std::string> tree;
    std::unordered_map<std::string, std::string> hash;
    for (int i = 0; i < 8; ++i) {
        tree[code[i]] = title[i];
        hash[code[i]] = title[i];
    }

    std::printf("1. Inserted in the order B1005 B1001 B1008 B1003 B1002 B1007 "
                "B1004 B1006\n  std::map          :");
    for (const auto &kv : tree) std::printf(" %s", kv.first.c_str());
    std::printf("\n  std::unordered_map:");
    for (const auto &kv : hash) std::printf(" %s", kv.first.c_str());

    std::printf("\n\n2. Inside the unordered_map\n");
    std::printf("  bucket_count = %zu, load_factor = %.3f, "
                "max_load_factor = %.1f\n", hash.bucket_count(),
                hash.load_factor(), hash.max_load_factor());
    for (int i = 0; i < 8; ++i)
        std::printf("  %s -> bucket %zu%s", code[i], hash.bucket(code[i]),
                    i % 2 == 1 ? "\n" : "   ");
    std::printf("  std::hash<int>{}(42) = %zu\n", std::hash<int>{}(42));
    std::unordered_map<int, int> grow;
    std::size_t last = grow.bucket_count();
    std::printf("  bucket counts while inserting 1..1000:  %zu", last);
    for (int i = 1; i <= 1000; ++i) {
        grow[i] = i;
        if (grow.bucket_count() != last) {
            last = grow.bucket_count();
            std::printf(" %zu", last);
        }
    }
```

Listing 15.11 (Continued) *maps.cpp: std::map against std::unordered_map on the same data, the bucket interface, range queries, a hash function for a two-field key, and a timing of one million keys.*

```
    std::printf("\n\n3. Questions only the ordered map can answer\n");
    std::printf("  first and last code: %s, %s\n", tree.begin()->first.c_str(),
                tree.rbegin()->first.c_str());
    std::printf("  codes from B1003 to B1006:");
    for (auto it = tree.lower_bound("B1003"); it != tree.upper_bound("B1006"); ++it)
        std::printf(" %s", it->first.c_str());
    std::printf("\n  first code after B1004: %s\n",
                tree.upper_bound("B1004")->first.c_str());

    std::printf("\n4. A key with two fields needs its own hash function\n");
    std::unordered_map<Edition, int, EditionHash> copies;
    copies[{"Laskar Pelangi", 2005}] = 3;
    copies[{"Laskar Pelangi", 2020}] = 2;
    copies[{"Bumi Manusia", 1980}] = 1;
    std::printf("  copies of Laskar Pelangi (2020): %d\n",
                copies.at({"Laskar Pelangi", 2020}));
    std::printf("  copies of Bumi Manusia (2005) recorded: %zu\n",
                copies.count({"Bumi Manusia", 2005}));

    std::printf("\n5. One million random integer keys (times vary by machine)\n");
    std::mt19937_64 gen(15);
    std::vector<std::uint64_t> keys(1000000);
    for (std::uint64_t &k : keys) k = gen();
    std::map<std::uint64_t, int> m;
    std::unordered_map<std::uint64_t, int> u;
    long long found = 0;
    double mi = ms([&] { for (std::uint64_t k : keys) m[k] = 1; });
    double mf = ms([&] { for (std::uint64_t k : keys) found += m.count(k); });
    double ui = ms([&] { for (std::uint64_t k : keys) u[k] = 1; });
    double uf = ms([&] { for (std::uint64_t k : keys) found += u.count(k); });
    std::printf("  %-20s %10s %10s\n", "", "insert", "find");
    std::printf("  %-20s %7.0f ms %7.0f ms\n", "std::map", mi, mf);
    std::printf("  %-20s %7.0f ms %7.0f ms\n", "std::unordered_map", ui, uf);
    std::printf("  (keys found: %lld)\n", found);
    return 0;
}
```

```
1. Inserted in the order B1005 B1001 B1008 B1003 B1002 B1007 B1004 B1006
  std::map          : B1001 B1002 B1003 B1004 B1005 B1006 B1007 B1008
  std::unordered_map: B1006 B1007 B1004 B1002 B1003 B1008 B1001 B1005

2. Inside the unordered_map
  bucket_count = 13, load_factor = 0.615, max_load_factor = 1.0
  B1005 -> bucket 1     B1001 -> bucket 0
  B1008 -> bucket 3     B1003 -> bucket 4
  B1002 -> bucket 9     B1007 -> bucket 7
  B1004 -> bucket 9     B1006 -> bucket 11
  std::hash<int>{}(42) = 42
  bucket counts while inserting 1..1000:  1 13 29 59 127 257 541 1109

3. Questions only the ordered map can answer
  first and last code: B1001, B1008
  codes from B1003 to B1006: B1003 B1004 B1005 B1006
  first code after B1004: B1005

4. A key with two fields needs its own hash function
  copies of Laskar Pelangi (2020): 2
  copies of Bumi Manusia (2005) recorded: 0

5. One million random integer keys (times vary by machine)
                          insert       find
  std::map                561 ms     621 ms
  std::unordered_map      153 ms      24 ms
  (keys found: 2000000)
```

Part 1 is the essential difference: the tree returns the codes sorted, whatever order they came in; the hash table returns them in an order that is neither sorted nor the insertion order, and that may change after the next rehash or with another compiler. Part 2 opens the hash table. It started with 13 buckets for 8 keys, a load factor of $8/13 \approx 0.615$, and it will rehash when the load exceeds `max_load_factor() = 1.0`, the default the standard prescribes [5]. The bucket counts during 1000 insertions, 1, 13, 29, 59, 127, 257, 541, 1109, are primes at roughly twice the previous size: GCC's implementation uses prime bucket counts, which is how it gets away with the identity as `std::hash<int>` (Section 15.3.2 showed that a prime m tolerates regular keys). Part 3 lists the queries that need order: minimum and maximum, the codes in a range, the next code after a given one. The tree answers each in $O(\lg n)$ (plus the size of the answer); the hash table can answer them only by examining every key.

Part 4 shows what a hash table needs from a user-defined key: `==` and a hash function. `EditionHash` combines the hashes of the title and the year with a formula taken from the Boost library, which shifts and adds the golden-ratio constant of Equation (15.4) so that the two parts mix instead of cancelling. A tree would need `operator<` instead. Part 5 is a real race on one million random 64-bit keys: the hash table inserts about 4 times faster and finds about 25 times faster in this run. Most of the tree's time goes into cache misses: each of its roughly 20 levels is a node somewhere else in memory, while the hash table touches one bucket and usually one node.

Table 15.5 summarizes the choice.

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

The same pair exists for sets (`std::set`, `std::unordered_set`), and Java has the same split between `TreeMap` and `HashMap`, which Chapter 11 introduced in Section 11.11.2.

15.10 Inside Java's `HashMap` and Python's `dict`

15.10.1 Java: `HashMap` and its treeified bins

Java's `java.util.HashMap` is a chained table whose design we can read directly in the JDK 25 source [8], [6].

- **The table.** An array of *bins* whose length is always a power of two, 16 by default, created on the first insertion. The default load factor is 0.75, so the table doubles when it holds more than 12 entries, then 24, and so on. Doubling a power-of-two table has a pleasant property: each bin's entries either stay at index i or move to $i + m_{\text{old}}$, decided by one bit of the hash, so a resize splits each chain in two without recomputing any hash.
- **Choosing a bin.** A power-of-two table takes the low bits of the hash, which is dangerous (Section 15.3.2). `HashMap` therefore first *spreads* the key's `hashCode()` by folding its high half into its low half, `h ^ (h >>> 16)`, and then takes `(n - 1) & hash`, the low bits. **Listing 15.12** prints this computation for three codes.
- **Treeified bins.** A bin starts as a linked list of `Node` objects. When one bin reaches `TREEIFY_THRESHOLD = 8` entries, it is converted into a red-black tree of `TreeNode` objects, provided the table has at least `MIN_TREEIFY_CAPACITY = 64` bins; a smaller table resizes instead, because a crowded bin in a small table is more likely a sign that the table is too small. A tree bin shrinks back to a list at `UNTREEIFY_THRESHOLD = 6`. Tree bins are ordered by hash value and, among equal hashes, by `compareTo` if the keys are `Comparable` (as `String` is). **Figure 15.9** draws a table with one treeified bin.

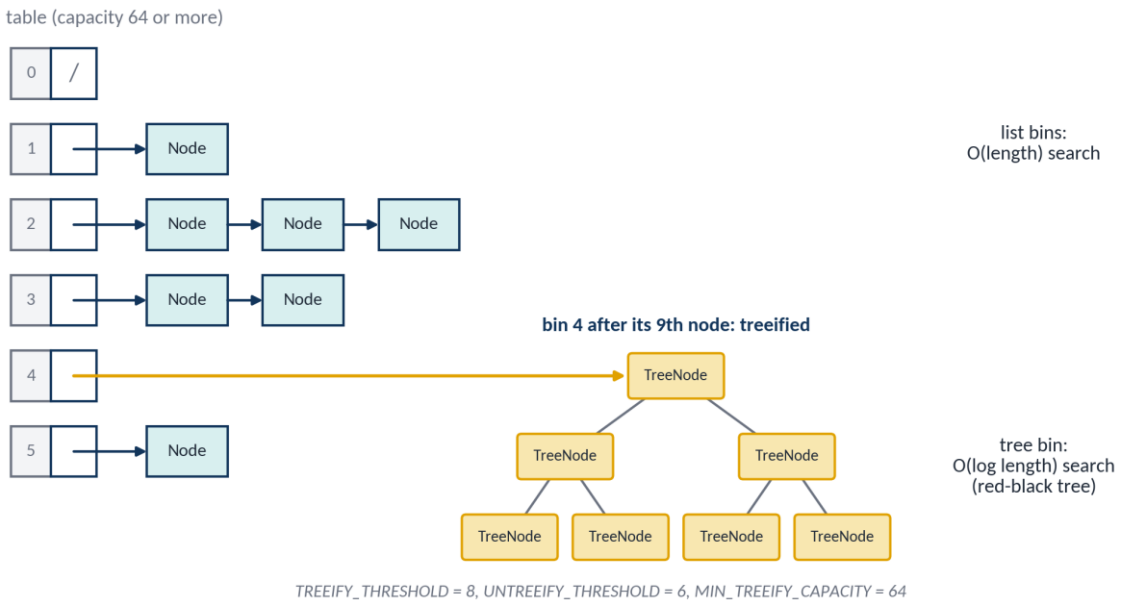


Figure 15.9 A `HashMap` table with list bins and one treeified bin. A bin becomes a red-black tree of `TreeNode`s when it collects 8 entries in a table of at least 64 bins, so a search inside it takes $O(\lg \ell)$ instead of $O(\ell)$ steps for a bin of ℓ entries.

Treeification was added in Java 8 by JEP 180 as the answer to hash flooding [16]. With random hash codes it almost never triggers: by Equation (15.22) with $\alpha = 0.5$, the typical load of a `HashMap`, a given bin holds 8 entries with probability $e^{-0.5}0.5^8/8! \approx 6 \times 10^{-8}$, which is the figure quoted in the source comment [8]. Its purpose is to cap the damage when hash codes are bad or hostile.

Listing 15.12 `LibraryMaps.java`: `HashMap`, `TreeMap` and `LinkedHashMap` on the same data, how a bin is chosen, a record as a key, 65,536 colliding keys, and a timing.

```
// LibraryMaps.java -- java.util.HashMap against TreeMap and LinkedHashMap:
// ordering, how HashMap picks a bin, range queries, a record as a key,
// colliding keys, and a stopwatch (Section 15.10).
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.SplittableRandom;
import java.util.TreeMap;

public class LibraryMaps {
    record Edition(String title, int year) {}       // equals and hashCode for free

    static double ms(Runnable r) {
        long t0 = System.nanoTime();
        r.run();
        return (System.nanoTime() - t0) / 1e6;
    }

    public static void main(String[] args) {
        String[] code = {"B1005", "B1001", "B1008", "B1003",
                         "B1002", "B1007", "B1004", "B1006"};
        Map<String, Integer> hash = new HashMap<>();
        TreeMap<String, Integer> tree = new TreeMap<>();
        Map<String, Integer> linked = new LinkedHashMap<>();
        for (int i = 0; i < code.length; i++) {
            hash.put(code[i], i);
            tree.put(code[i], i);
            linked.put(code[i], i);
        }
        System.out.println("1. Inserted in the order B1005 B1001 B1008 B1003 B1002 "
                           + "B1007 B1004 B1006");
        System.out.println("  HashMap      : " + String.join(" ", hash.keySet()));
        System.out.println("  TreeMap      : " + String.join(" ", tree.keySet()));
        System.out.println("  LinkedHashMap: " + String.join(" ", linked.keySet()));

        System.out.println("\n2. How HashMap picks a bin (table of 16 bins)");
        for (String c : new String[] {"B1001", "B1002", "B1008"}) {
            int h = c.hashCode();
            int spread = h ^ (h >>> 16);             // HashMap.hash(key)
            System.out.printf("  %s  hashCode = %d  bin = %d%n", c, h, spread & 15);
        }
        System.out.printf("  \"Aa\".hashCode() = %d, \"BB\".hashCode() = %d%n",
                          "Aa".hashCode(), "BB".hashCode());
```

Listing 15.12 (Continued) *LibraryMaps.java: HashMap, TreeMap and LinkedHashMap on the same data, how a bin is chosen, a record as a key, 65,536 colliding keys, and a timing.*

```
        System.out.println("\n3. Questions only the TreeMap can answer");
        System.out.println("  first and last code: " + tree.firstKey() + ", "
                           + tree.lastKey());
        Map<String, Integer> range = tree.subMap("B1003", true, "B1006", true);
        System.out.println("  codes from B1003 to B1006: "
                           + String.join(" ", range.keySet()));
        System.out.println("  first code after B1004: " + tree.higherKey("B1004"));

        System.out.println("\n4. A record is a key with two fields");
        Map<Edition, Integer> copies = new HashMap<>();
        copies.put(new Edition("Laskar Pelangi", 2005), 3);
        copies.put(new Edition("Laskar Pelangi", 2020), 2);
        System.out.println("  copies of Laskar Pelangi (2020): "
                           + copies.get(new Edition("Laskar Pelangi", 2020)));

        System.out.println("\n5. 65,536 keys whose hashCode values are all equal "
                           + "(times vary)");
        List<String> attack = new ArrayList<>(), ordinary = new ArrayList<>();
        for (int mask = 0; mask < 65536; mask++) {
            StringBuilder sb = new StringBuilder();
            for (int b = 15; b >= 0; b--)
                sb.append((mask >> b & 1) != 0 ? "BB" : "Aa");
            attack.add(sb.toString());
            ordinary.add(String.format("reader%05d", mask));
        }
        long distinct = attack.stream().map(String::hashCode).distinct().count();
        System.out.println("  distinct hashCode values among the attack keys: "
                           + distinct);
        for (int round = 0; round < 2; round++) {    // round 0 warms up the JIT
            Map<String, Integer> a = new HashMap<>(), o = new HashMap<>();
            double ta = ms(() -> {
                for (String s : attack) a.put(s, 1);
                for (String s : attack) a.get(s);
            });
            double to = ms(() -> {
                for (String s : ordinary) o.put(s, 1);
                for (String s : ordinary) o.get(s);
            });
            if (round == 1)
                System.out.printf("  insert + find all: attack keys %.0f ms, "
                                  + "ordinary keys %.0f ms%n", ta, to);
        }
```

Listing 15.12 (Continued) *LibraryMaps.java: HashMap, TreeMap and LinkedHashMap on the same data, how a bin is chosen, a record as a key, 65,536 colliding keys, and a timing.*

```
        System.out.println("\n6. One million random long keys "
                           + "(times vary by machine)");
        SplittableRandom rnd = new SplittableRandom(15);
        long[] keys = new long[1_000_000];
        for (int i = 0; i < keys.length; i++) keys[i] = rnd.nextLong();
        Map<Long, Integer> tm = new TreeMap<>(), hm = new HashMap<>();
        long[] found = {0};
        double ti = ms(() -> { for (long k : keys) tm.put(k, 1); });
        double tf = ms(() -> {
            for (long k : keys) if (tm.containsKey(k)) found[0]++;
        });
        double hi = ms(() -> { for (long k : keys) hm.put(k, 1); });
        double hf = ms(() -> {
            for (long k : keys) if (hm.containsKey(k)) found[0]++;
        });
        System.out.printf("  %-10s %10s %10s%n", "", "insert", "find");
        System.out.printf("  %-10s %7.0f ms %7.0f ms%n", "TreeMap", ti, tf);
        System.out.printf("  %-10s %7.0f ms %7.0f ms%n", "HashMap", hi, hf);
        System.out.println("  (keys found: " + found[0] + ")");
    }
}
```

```
1. Inserted in the order B1005 B1001 B1008 B1003 B1002 B1007 B1004 B1006
  HashMap      : B1008 B1007 B1002 B1001 B1004 B1003 B1006 B1005
  TreeMap      : B1001 B1002 B1003 B1004 B1005 B1006 B1007 B1008
  LinkedHashMap: B1005 B1001 B1008 B1003 B1002 B1007 B1004 B1006

2. How HashMap picks a bin (table of 16 bins)
  B1001  hashCode = 62459810  bin = 11
  B1002  hashCode = 62459811  bin = 10
  B1008  hashCode = 62459817  bin = 0
  "Aa".hashCode() = 2112, "BB".hashCode() = 2112

3. Questions only the TreeMap can answer
  first and last code: B1001, B1008
  codes from B1003 to B1006: B1003 B1004 B1005 B1006
  first code after B1004: B1005

4. A record is a key with two fields
  copies of Laskar Pelangi (2020): 2

5. 65,536 keys whose hashCode values are all equal (times vary)
  distinct hashCode values among the attack keys: 1
  insert + find all: attack keys 122 ms, ordinary keys 9 ms

6. One million random long keys (times vary by machine)
                 insert       find
  TreeMap        733 ms     572 ms
  HashMap        255 ms     140 ms
  (keys found: 2000000)
```

The output repeats the lessons of Section 15.9 in Java's vocabulary. `HashMap` iterates in bin order, `TreeMap` (a red-black tree, Section 11.11.2) in sorted order, and `LinkedHashMap`, a hash table with a doubly linked list threaded through its entries (Section 9.12.2), in insertion order. The codes B1001 and B1002, whose `hashCode` values differ by one, land in different bins. A `record` gets `equals` and `hashCode` automatically, so it can be a key without any extra code. Part 5 is the flooding experiment of Section 15.8 run against the real library: 65,536 keys with one single `hashCode` value are all inserted and found in about a tenth of a second, only about thirteen times slower than ordinary keys, because they live in a red-black tree ordered by `compareTo`. With plain

list bins the same work would take about $65536^2/2 \approx 2 \times 10^9$ string comparisons. In part 6, `HashMap` beats `TreeMap` by a factor of about 3 to 4.

15.10.2 Python: the compact, insertion-ordered `dict`

Python's `dict` is the most heavily used hash table in the world's most popular teaching language, and every Python object's attributes, every module's globals and every class's methods live in one. Its current design is described at length in the comments of the CPython source file `Objects/dictobject.c` [7].

- **Open addressing with perturbation.** A `dict` is a probed table of 2^k slots, at least 8. Its first probe is the low k bits of the hash, and it continues with the recurrence

$$j \leftarrow (5j + 1 + \text{perturb}) \bmod 2^k, \qquad \text{perturb} \leftarrow \text{perturb} \gg 5, \tag{15.26}$$

where *perturb* starts as the full hash value. The shifted-in high bits make the probe sequence depend on the whole hash, like double hashing; once *perturb* reaches zero, the recurrence $j \to 5j + 1 \bmod 2^k$ alone visits every slot, so an empty slot is always found.

- **Two arrays instead of one.** Since Python 3.6 the table is *compact*, an idea Raymond Hettinger proposed in 2012 [17]. The hash table proper, `indices`, holds only small integers (one byte each while the table has at most 128 slots); each points into a dense array `entries` of (hash, key, value) triples, appended in insertion order. **Figure 15.10** draws it. The probing happens in the small `indices` array, and the large entries are never moved around by collisions.
- **Order for free.** Iteration simply walks `entries` from the first to the last, so a `dict` returns its keys in insertion order. This began as a side effect of the compact layout in 3.6 and was declared a guarantee of the language in Python 3.7 [18]. A `set` has no `entries` array, and its order is *not* insertion order.
- **Load and growth.** A `dict` may fill at most 2/3 of its slots (`USABLE_FRACTION`), and when it is full it grows to hold about three times the current number of items (`GROWTH_RATE`).
- **Hash values.** Integers hash to themselves modulo the prime $2^{61} - 1$ (except that `hash(-1)` is -2, because -1 signals an error in CPython's C interface). Strings and bytes use SipHash-1-3 with a random key chosen when the interpreter starts [15], [14]; the environment variable `PYTHONHASHSEED` fixes the key for reproducible runs.

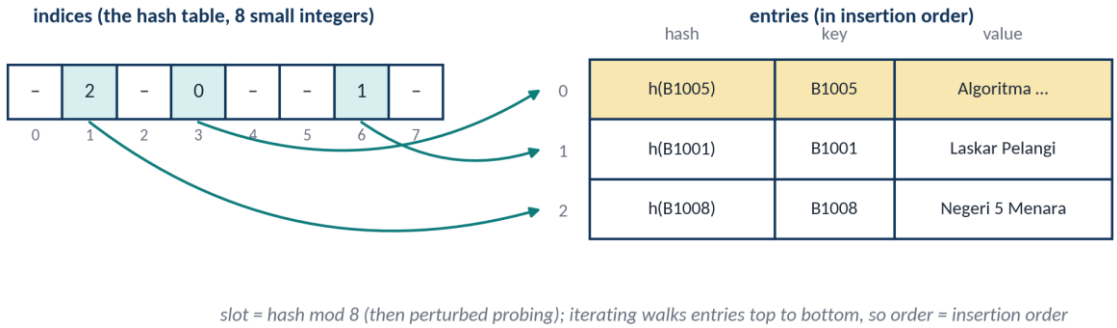


Figure 15.10 The compact `dict` layout with three items: a small `indices` table of 8 one-byte slots is probed by hash, and each used slot holds the position of an entry in the dense `entries` array, which is kept in insertion order. (The slot positions shown are illustrative; real hash values change from run to run.)

Listing 15.13 observes each of these points from Python code.

Listing 15.13 `library_maps.py`: insertion order, integer and string hash values, hash randomization across interpreter runs, the growth of a `dict`, and a timing against binary search.

```
# library_maps.py -- Python's dict and set: insertion order, how small
# integers and strings hash, hash randomization between runs, how a dict
# grows, and a stopwatch against binary search (Section 15.10).
import bisect
import os
import random
import subprocess
import sys
import time

code = ["B1005", "B1001", "B1008", "B1003", "B1002", "B1007", "B1004", "B1006"]
catalog = {c: i for i, c in enumerate(code)}

print("1. Inserted in the order", " ".join(code))
print("  dict         :", " ".join(catalog))
print("  sorted(dict) :", " ".join(sorted(catalog)))

print("\n2. Hash values")
print("  hash(42) =", hash(42), "  hash(-1) =", hash(-1),
      "  hash(2**61) =", hash(2**61))
print("  sys.hash_info.algorithm =", sys.hash_info.algorithm,
      "  modulus = 2**61 - 1:", sys.hash_info.modulus == 2**61 - 1)
script = "print(hash('B1001'), ' '.join({'B1001', 'B1002', 'B1003', 'B1004'}))"
for seed in ("0", "1", "2"):                          # three separate interpreters
    env = dict(os.environ, PYTHONHASHSEED=seed)
    out = subprocess.run([sys.executable, "-c", script], env=env,
                         capture_output=True, text=True).stdout.strip()
    print("  PYTHONHASHSEED=%s: %s" % (seed, out))

print("\n3. Memory of a dict as it grows (sizes vary by Python version)")
d, last, steps = {}, sys.getsizeof({}), []
for i in range(1, 100):
    d[i] = i
    size = sys.getsizeof(d)
    if size != last:
        steps.append("%2d key(s) -> %4d bytes" % (i, size))
        last = size
print("  " + "\n  ".join(steps))
```

Listing 15.13 (Continued) *library_maps.py: insertion order, integer and string hash values, hash randomization across interpreter runs, the growth of a dict, and a timing against binary search.*

```
print("\n4. 200,000 random keys: dict against binary search (times vary)")
rng = random.Random(15)
keys = [rng.getrandbits(62) for _ in range(200_000)]
t0 = time.perf_counter()
table = {k: 1 for k in keys}
found = sum(1 for k in keys if k in table)
t1 = time.perf_counter()
ordered = sorted(keys)
found += sum(1 for k in keys if ordered[bisect.bisect_left(ordered, k)] == k)
t2 = time.perf_counter()
print("  dict (build + find all)           : %6.0f ms" % ((t1 - t0) * 1000))
print("  sorted list + bisect (sort + find): %6.0f ms" % ((t2 - t1) * 1000))
print("  (keys found: %d)" % found)
```

```
1. Inserted in the order B1005 B1001 B1008 B1003 B1002 B1007 B1004 B1006
  dict         : B1005 B1001 B1008 B1003 B1002 B1007 B1004 B1006
  sorted(dict) : B1001 B1002 B1003 B1004 B1005 B1006 B1007 B1008

2. Hash values
  hash(42) = 42   hash(-1) = -2   hash(2**61) = 1
  sys.hash_info.algorithm = siphash13   modulus = 2**61 - 1: True
  PYTHONHASHSEED=0: -1661493604806230238 B1004 B1001 B1002 B1003
  PYTHONHASHSEED=1: -7535737535382290098 B1002 B1003 B1001 B1004
  PYTHONHASHSEED=2: -4607229078860313946 B1002 B1003 B1004 B1001

3. Memory of a dict as it grows (sizes vary by Python version)
   1 key(s) ->  224 bytes
   6 key(s) ->  352 bytes
  11 key(s) ->  632 bytes
  22 key(s) -> 1168 bytes
  43 key(s) -> 2264 bytes
  86 key(s) -> 4688 bytes

4. 200,000 random keys: dict against binary search (times vary)
  dict (build + find all)           :     44 ms
  sorted list + bisect (sort + find):    164 ms
  (keys found: 400000)
```

Part 2 runs three separate interpreters with three values of `PYTHONHASHSEED`: the hash of "B1001" is different every time, and so is the iteration order of a four-element *set*, while a `dict` keeps insertion order regardless. Part 3 shows the table growing when the 6th, 11th, 22nd, 43rd and 86th keys arrive, exactly when the 2/3 limit of tables of 8, 16, 32, 64 and 128 slots (usable sizes 5, 10, 21, 42 and 85) is exceeded; the byte counts belong to Python 3.11 and differ slightly between versions. Part 4 compares building a `dict` of 200,000 keys and finding each one with sorting a list and finding each key by binary search: the hash table wins by a factor of about 4.

Table 15.6 gathers the facts of Sections 15.9 and 15.10.

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`)

15.11 Applications

A hash table earns its place whenever a program keeps asking "have I seen this before?" or "what do I know about this key?". **Listing 15.14** solves four such problems for Pustaka Ceria; `Applications.java` and `applications.py` in the chapter's code folder print the same output.

Listing 15.14 `applications.cpp`: counting the words of a catalog, detecting duplicate returns, two-sum on book prices, and an LRU cache of catalog records.

```
// applications.cpp -- four everyday jobs for a hash table (Section 15.11):
// counting words, detecting duplicates, two-sum, and an LRU cache.
#include <algorithm>
#include <cctype>
#include <cstdio>
#include <list>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <vector>

const std::vector<std::string> TITLES = {
    "Laskar Pelangi", "Sang Pemimpi", "Edensor", "Maryamah Karpov",
    "Bumi Manusia", "Anak Semua Bangsa", "Jejak Langkah", "Rumah Kaca",
    "Filosofi Kopi", "Perahu Kertas", "Negeri 5 Menara", "Ranah 3 Warna",
    "Rantau 1 Muara", "Cantik Itu Luka", "Lelaki Harimau", "Ronggeng Dukuh Paruk",
    "Sejarah Nusantara", "Cerita Rakyat Nusantara", "Algoritma & Struktur Data",
    "Struktur Data dengan C++", "Dasar-Dasar Pemrograman", "Fisika Dasar",
    "Kimia Dasar", "Matematika Diskrit", "Pengantar Basis Data",
    "Sistem Basis Data", "Cerita Anak Nusantara", "Dongeng Anak Indonesia"};

void countWords() {
    std::unordered_map<std::string, int> count;
    int total = 0;
    for (const std::string &t : TITLES) {
        std::string w;
        for (std::size_t i = 0; i <= t.size(); ++i) {
            if (i < t.size() && std::isalnum((unsigned char)t[i])) {
                w += (char)std::tolower((unsigned char)t[i]);
            } else if (!w.empty()) {                // a word just ended
                ++count[w];                         // absent key starts at 0
                ++total;
                w.clear();
            }
        }
    }
    std::vector<std::pair<std::string, int>> v(count.begin(), count.end());
    std::sort(v.begin(), v.end(), [](const auto &a, const auto &b) {
        return a.second != b.second ? a.second > b.second : a.first < b.first;
    });
    std::printf("1. Words in %zu titles: %d in all, %zu different\n", TITLES.size(),
                total, count.size());
    std::printf("   most frequent:");
    for (int i = 0; i < 6; ++i)
        std::printf(" %s (%d)", v[i].first.c_str(), v[i].second);
```

Listing 15.14 (Continued) *applications.cpp: counting the words of a catalog, detecting duplicate returns, two-sum on book prices, and an LRU cache of catalog records.*

```
    std::printf("\n");
}

void duplicates() {
    const std::vector<std::string> returned = {"B1003", "B1017", "B1008", "B1003",
                                               "B1022", "B1017", "B1030", "B1003"};
    std::unordered_set<std::string> seen, reported;
    std::printf("\n2. Books returned today:");
    for (const std::string &c : returned) std::printf(" %s", c.c_str());
    std::printf("\n");
    for (std::size_t i = 0; i < returned.size(); ++i) {
        const std::string &c = returned[i];
        if (!seen.insert(c).second && reported.insert(c).second)   // seen before?
            std::printf("  %s appears again at position %zu\n", c.c_str(), i);
    }
    std::printf("  %zu different books, %zu set lookups (a double loop would "
                "compare up to %zu pairs)\n", seen.size(), returned.size(),
                returned.size() * (returned.size() - 1) / 2);
}

void twoSum() {
    const std::vector<long> price = {45000, 72000, 38000, 91000,
                                     64000, 27000, 83000, 55000};
    const long budget = 119000;
    std::unordered_map<long, std::size_t> where;     // price -> index seen so far
    std::printf("\n3. Two books that cost exactly Rp%ld together\n", budget);
    for (std::size_t i = 0; i < price.size(); ++i) {
        auto it = where.find(budget - price[i]);
        if (it != where.end()) {
            std::printf("  books %zu and %zu: Rp%ld + Rp%ld (found after %zu "
                        "lookups)\n", it->second, i, price[it->second], price[i],
                        i + 1);
            return;
        }
        where[price[i]] = i;
    }
    std::printf("  no such pair\n");
}

class LruCache {                                     // a hash map + a linked list
public:
    explicit LruCache(std::size_t capacity) : capacity_(capacity) {}
```

Listing 15.14 (Continued) *applications.cpp: counting the words of a catalog, detecting duplicate returns, two-sum on book prices, and an LRU cache of catalog records.*

```
    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;
    }

    std::string contents() const {
        std::string s;
        for (const std::string &k : order_) s += (s.empty() ? "" : " ") + k;
        return s;
    }

private:
    std::size_t capacity_;
    std::list<std::string> order_;                  // most recent first
    std::unordered_map<std::string, std::list<std::string>::iterator> pos_;
};

void cache() {
    const char *req[10] = {"B1001", "B1002", "B1001", "B1003", "B1001",
                           "B1002", "B1004", "B1001", "B1002", "B1005"};
    LruCache c(3);
    int hits = 0;
    std::printf("\n4. An LRU cache of 3 catalog records (most recent first)\n");
    for (const char *r : req) {
        bool hit = c.get(r);
        hits += hit;
        std::printf("  get %s: %-4s  cache: %s\n", r,
                    hit ? "hit" : "miss", c.contents().c_str());
    }
    std::printf("  %d hits out of 10 requests\n", hits);
}
```

Listing 15.14 (Continued) *applications.cpp: counting the words of a catalog, detecting duplicate returns, two-sum on book prices, and an LRU cache of catalog records.*

```
int main() {
    countWords();
    duplicates();
    twoSum();
    cache();
    return 0;
}
```

```
1. Words in 28 titles: 70 in all, 57 different
  most frequent: dasar (4) data (4) anak (3) nusantara (3) basis (2) cerita (2)

2. Books returned today: B1003 B1017 B1008 B1003 B1022 B1017 B1030 B1003
  B1003 appears again at position 3
  B1017 appears again at position 5
  5 different books, 8 set lookups (a double loop would compare up to 28 pairs)

3. Two books that cost exactly Rp119000 together
  books 4 and 7: Rp64000 + Rp55000 (found after 8 lookups)

4. An LRU cache of 3 catalog records (most recent first)
  get B1001: miss  cache: B1001
  get B1002: miss  cache: B1002 B1001
  get B1001: hit   cache: B1001 B1002
  get B1003: miss  cache: B1003 B1001 B1002
  get B1001: hit   cache: B1001 B1003 B1002
  get B1002: hit   cache: B1002 B1001 B1003
  get B1004: miss  cache: B1004 B1002 B1001
  get B1001: hit   cache: B1001 B1004 B1002
  get B1002: hit   cache: B1002 B1001 B1004
  get B1005: miss  cache: B1005 B1002 B1001
  5 hits out of 10 requests
```

15.11.1 Counting words in the catalog

The library wants to know which words occur most often in its titles, to build a subject index. `countWords` splits each title into words (letters and digits, lower-cased) and increments a counter per word in an `unordered_map<string, int>`. The line `++count[w]` does all the work: `operator[]` finds the word's counter, or creates it with the value 0 if the word is new, and the increment follows. Each word costs one expected $O(1)$ lookup, so counting W words takes $O(W)$ expected time, where sorting the words first would take $O(W \lg W)$. Only the *report* needs order: the D distinct words are copied into a vector and sorted by count, $O(D \lg D)$. The 28 titles contain 70 words, 57 different; *dasar* and *data* each occur 4 times. Java writes the counter update as `count.merge(w, 1, Integer::sum)`, and Python has a ready-made counting dictionary, `collections.Counter`.

15.11.2 Detecting duplicates

At closing time the returns desk scans every returned book, and the same code appearing twice means a scanning error. With an `unordered_set` of codes seen so far, each scan is one expected $O(1)$ insertion; `insert` returns a pair whose `.second` is `false` when the code was already present, which detects the duplicate in the same step. A second set, `reported`, makes each duplicate reported once. For n scans that is $O(n)$ expected time, where comparing every pair would take $n(n-1)/2$ comparisons, 28 for these 8 codes but about 500 million for 32,000. This is the method Section 8.7.3 used to remove duplicates from a linked list, and the method of Section 4.4.3's 256 counters, now for keys that are not small integers.

15.11.3 Two-sum

A donor offers Rp119,000 for exactly two books. Which two prices add up to the budget? Trying every pair costs $\Theta(n^2)$. With a hash map the question for each book becomes a lookup: is the *complement*, `budget - price[i]`, the price of a book we have already seen? `twoSum` stores each price with its index after checking for its complement, so it finds a pair in one pass, $O(n)$ expected time. The output shows books 4 and 7, Rp64,000 and Rp55,000, found at the eighth lookup. Checking before inserting also avoids pairing a book with itself when its price is exactly half the

budget. The same pattern, "store what you have seen, look up what you need", solves a family of problems: Exercise 15.9 counts the stretches of days whose borrowings add up to a given number by storing prefix sums.

15.11.4 Caches

A *cache* keeps the results of expensive operations (a slow catalog query, a page fetched over the network, a computed score) so that a repeated request is answered at once. A cache must answer "is this key here?" in $O(1)$, which needs a hash map, and it must decide what to throw away when it is full. The *least recently used* (LRU) policy throws away the entry that has gone unused for the longest time, which needs the entries ordered by last use and moved to the front on every access, which needs a doubly linked list. Section 9.9 built exactly this pair, a sentinel list plus a hash map from key to node. `LruCache` in **Listing 15.14** uses the library versions: a `std::list` of keys, most recent first, and an `unordered_map` from each key to its *iterator* in the list. On a hit, `splice` moves the node to the front in $O(1)$ without copying it; on a miss with a full cache, the key at the back is removed from both structures; then the new key goes in front. In the output, ten requests to a cache of three records give five hits; the miss for B1004 evicts B1003, which had gone unused longest. Java's `LinkedHashMap` in access order and Python's `OrderedDict` are this very combination packaged as one class, and Python's `functools.lru_cache` wraps it around a function.

15.11.5 Reverse maps

Sometimes the question runs backward: not "where does this ticket go?" but "which ticket arrives here?". Storing a second map with keys and values exchanged answers it in $O(1)$ too, at the price of twice the memory. The itinerary problem of Chapter 16 uses exactly this: the one city that never appears as a destination in the reverse map is where the journey starts.

15.12 Profitina in Practice

Profitina is an AI visibility (GEO) platform: it scans a brand's website, scores how ready the site is to be cited by AI answer engines, explains each signal, and helps the owner fix what it finds [19]. It is a Python application built with FastAPI on PostgreSQL, and hash tables appear wherever it must recognize something it has seen before. We describe the shape of each problem and the class of structure that fits it; the implementation choices inside the product are its own.

- **One page, many addresses.** A crawler meets the same page under many URLs: with and without `www.`, with `http` or `https`, with a trailing slash, with tracking parameters appended. Before scanning, each URL is *canonicalized* (scheme and host lower-cased, default port and fragment removed, known tracking parameters dropped, parameters sorted), and the canonical string goes into a set of URLs already visited. The test "visited before?" is then one expected $O(1)$ lookup, the duplicate detection of Section 15.11.2, and it is what keeps a crawl of a site with thousands of links from revisiting pages endlessly.
- **Pages that are almost the same.** Many sites publish near-copies: the same article under two categories, product pages that differ in one line. Comparing every pair of pages word by word is quadratic. Instead, each page can be cut into overlapping windows of a few words, each window hashed with a rolling hash (Section 15.3.5), and the page represented by a small set of these fingerprints. Two pages whose fingerprint sets overlap heavily are near-duplicates, and the overlap of two sets is computed with hash-set lookups.

- **Screening links.** Every link posted in Profitina's community is screened by a rule floor that combines several signals. Several of those signals are membership questions, such as whether a domain appears in a list of known shorteners or known-bad hosts, and a hash set answers each in constant time however long the lists grow.
- **Counting per user.** The login rate limit, the daily caps on earning PROFIT and the "N new" counters of followed channels are all counters keyed by an account or a channel: the word-counting pattern of Section 15.11.1, usually held in a hash map with an expiry time.
- **Caching expensive answers.** Scans and the local Qwen2.5 model are expensive, and the same questions recur. A bounded cache with the LRU policy of Section 15.11.4 avoids repeating the work.
- **The other kind of hash.** Profitina stores passwords with Argon2id, as its UU PDP 27/2022 compliance work requires. That is a *password* hash, deliberately slow, salted and memory-hard, the opposite of the fast table hashes of this chapter; the warning in Section 15.3.7 explains why the two must never be confused.

A small runnable version of the first pattern is a good exercise for the Profitina Laundry case study: store each order under its order code in an `unordered_map`, reject a second order with the same code, and count the garments per customer in a second map.

15.13 Common Mistakes, and How to Avoid Them

Taking the remainder by a power of two with a weak hash. `k % 16` keeps only four bits of k; for round prices or aligned addresses they may all be zero. Use a prime table size with the division method, or mix the hash first (Sections 15.3.2 and 15.3.6).

Adding up characters. A sum of character codes ignores order, so anagrams collide, and it crowds short strings into a narrow range. Use the polynomial hash Equation (15.5) or a library hash.

Negative remainders. In C++ and Java, `-7 % 5` is -2, not 3, so `h % m` of a negative hash is a negative index. Hash into an unsigned type, or use `Math.floorMod` in Java; Python's `%` is already non-negative.

Changing a key while it is in the table. The table filed the key under the hash it had when it was inserted. If a Java object used as a key changes a field that `hashCode` depends on, the table searches the wrong bucket and the entry is lost for good. Keys must be immutable, or at least not modified while stored.

Defining equality without the matching hash. Two keys that are equal must have equal hashes. A Java class that overrides `equals` but not `hashCode`, or a C++ hash functor that ignores a field that `==` compares, breaks this rule, and equal keys end up in different buckets. Java records and Python's `@dataclass(frozen=True)` generate both together.

Emptying a slot on deletion in open addressing. It cuts the probe sequences of later keys, which then cannot be found (**Figure 15.5**). Use tombstones or backward-shift deletion.

Stopping an insertion at the first tombstone. The key may already be stored further along its probe sequence; keep walking to an empty slot, then reuse the first tombstone (**Pseudocode 15.3**).

Letting the load factor grow. A chained table without rehashing turns into m long lists, and a probed table near $\alpha = 1$ turns every search into a scan. Rehash by *doubling*, never by adding a constant (Equation (15.23)).

Relying on iteration order. An `unordered_map` or `HashMap` may return its keys in a different order after one more insertion, on another compiler or JDK, or in Python sets from one run to the next. If the output must be sorted, sort it, or use `std::map`, `TreeMap` or `sorted(dict)`.

Calling `std::map` a hash table. It is a red-black tree with $O(\lg n)$ operations. Choose `std::unordered_map` when only exact lookups matter, and `std::map` when order or ranges do (Section 15.9).

Using `operator[]` to test for presence. In C++, `if (m[key] == 0)` *inserts* `key` with a default value when it is absent. Use `m.find(key) != m.end()`, `m.count(key)` or, in C++20, `m.contains(key)`.

Trusting a public hash with keys chosen by strangers. Anyone who knows the function can make every key collide (**Listing 15.10**). Use a keyed hash with a random secret, or a structure with guaranteed bounds.

Using a table hash for passwords, or a password hash for tables. They are built for opposite goals (Section 15.3.7).

15.14 Chapter Summary

- **Hashing** computes a slot from the key itself instead of comparing keys, and so escapes the $\lceil \lg(n+1) \rceil$ lower bound of comparison searching: search, insertion and deletion take $O(1)$ time on average. It gives up order and worst-case guarantees (**Table 15.1**).
- **Direct addressing** uses the key as the index; it is perfect for small integer universes and impossible for large ones. A **hash function** $h: U \to \{0, \ldots, m-1\}$ squeezes a large universe into m slots; two keys with the same slot **collide**.
- **Good hash functions** are deterministic, fast, uniform on real data and use the whole key. The **division method** k mod m wants a prime m; the **multiplication method** works for $m = 2^p$ and takes the *top* bits of $k \cdot \lfloor 2^w A \rfloor$; strings use the **polynomial hash** computed by Horner's rule, which also **rolls** along a text in $O(1)$ per window. The **avalanche** test shows how well a function mixes: a mixer such as the SplitMix64 finalizer flips about half the output bits for any one-bit change.
- **Collisions come early**: by the **birthday bound**, n random keys in m slots collide with probability about $1 - e^{-n(n-1)/2m}$, even odds at $n \approx 1.1774\sqrt{m}$ (23 for 365, about five billion for 2^{64}).
- **Separate chaining** keeps a linked list per bucket: a search makes about $1 + \alpha/2$ comparisons if successful and α if not, where $\alpha = n/m$ is the **load factor**; chain lengths follow the Poisson distribution.
- **Open addressing** stores keys in the array and follows a **probe sequence**: **linear** (primary clustering, $\frac{1}{2}(1 + 1/(1-\alpha)^2)$ unsuccessful probes), **quadratic** (secondary clustering; triangular offsets reach every slot of a power-of-two table) or **double hashing** (close to uniform hashing, $1/(1-\alpha)$). Deletion needs **tombstones**.
- **Measurements** on 16,384 slots agree with all these formulas; linear probing near $\alpha = 1$ is both slow and erratic.
- **Rehashing** by doubling keeps α bounded at an amortized cost of fewer than two moves per insertion (measured: 1.966).

- The **worst case** is $\Theta(n)$ per operation, and **hash flooding** provokes it on purpose; **universal** and **keyed** hashing (SipHash in Python, keyed at startup) and **treeified bins** (Java 8 onward) are the defenses.
- **Libraries**: `std::unordered_map` is a chained hash table, `std::map` a red-black tree; Java's `HashMap` chains into power-of-two tables and turns crowded bins into trees; Python's `dict` probes a compact index array and keeps insertion order (**Table 15.6**).
- **Applications**: counting, duplicate detection, two-sum and its prefix-sum variants, reverse maps, and caches (hash map plus doubly linked list for LRU).

15.15 Exercises

15.1 Insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10, in this order, into a hash table with separate chaining, $m = 9$ buckets and $h(k) = k \bmod 9$, inserting each key at the front of its chain. Draw the table. What is the load factor, and how many comparisons does a successful search make on average?

15.2 Insert the keys 10, 22, 31, 4, 15, 28, 17, 88, 59, in this order, into an open-addressed table of $m = 11$ slots with $h_1(k) = k \bmod 11$, using (a) linear probing; (b) quadratic probing with $h(k, i) = (h_1(k) + i + 3i^2) \bmod 11$; (c) double hashing with $h_2(k) = 1 + (k \bmod 10)$. Show each final table and count the probes.

15.3 Compute the multiplication-method hash of $k = 123456$ for $m = 2^{14}$ with $A = (\sqrt{5} - 1)/2$, first with Equation (15.3), then with the integer form Equation (15.4) for $w = 32$, where $s = \lfloor 2^{32} A \rfloor = 2654435769$.

15.4 Compute the polynomial hash of the string "abc" with base 31 modulo 101 by Horner's rule, reducing after every step. The codes of a, b, c are 97, 98, 99.

15.5 A library issues random 32-bit identifiers to its members and never checks for duplicates. Using Equation (15.8), estimate how many members it can register before the probability that two of them share an identifier reaches 1%. What about 64-bit identifiers?

15.6 Write a function that returns the first word of a list that occurs exactly once in the list, in $O(n)$ expected time. Test it on "kopi teh kopi susu teh jahe susu".

15.7 Write a function that groups words that are anagrams of each other, keeping the groups in order of their first word. (Hint: two words are anagrams exactly when their sorted letters are equal.) Test it on kasur, rusak, kita, kiat, sukar, taki, buku, kubu.

15.8 Given an unsorted array of integers, find the length of the longest run of consecutive integers in it (for 100, 4, 200, 1, 3, 2, 101, 102, 103, 5 the answer is 5, the run 1 to 5) in $O(n)$ expected time, without sorting.

15.9 The number of books borrowed on eight consecutive days, with returns counted as negative corrections, was 3, 4, 7, 2, −3, 1, 4, 2. How many stretches of consecutive days add up to exactly 7? Solve it in $O(n)$ expected time with prefix sums and a hash map.

15.10 A table is expected to hold 1500 keys. (a) What is the smallest power-of-two size m that keeps $\alpha \le 0.75$? (b) At that load factor, how many probes does Equation (15.20) predict for an unsuccessful search with linear probing, and Equation (15.18) with double hashing? (c) How many comparisons with chaining?

15.11 *Backward-shift deletion.* With linear probing, a deletion can avoid tombstones: after emptying slot j, scan the following slots of the run; any key whose home slot does not lie cyclically in $(j, i]$, where i is its current slot, is moved back into slot j, and the scan continues from its old position. Apply this to the linear-probing table of **Figure 15.5**(a) when 1019 is deleted, and explain why every remaining key can still be found.

15.12 For each task, choose `std::map` or `std::unordered_map`, and explain: (a) print the catalog in order of book code; (b) check whether a scanned code belongs to the catalog, a million times a day; (c) list all loans whose due date falls in the next seven days; (d) count how many times each reader borrowed a book this year, printed in any order.

15.13 A chained table starts with $m = 1$ and rehashes when $n > m$. (a) Show that if it doubles m at each rehash, the total number of keys moved while inserting n keys is $O(n)$, less than about $2n$. (b) Show that if it instead adds 1000 buckets at each rehash, the total is $\Theta(n^2)$.

15.14 (a) Why does a keyed hash with a secret chosen at startup defeat the attack of **Listing 15.10**, while a secret fixed in the source code does not? (b) Java's treeified bins order colliding keys by `compareTo`. What happens to a treeified bin whose keys all have the same `hashCode` and do *not* implement `Comparable`?

15.15 In Python, why is `hash(-1)` equal to -2? Why does **Listing 15.13** print a set of four codes in a different order under each `PYTHONHASHSEED`, while a `dict` of the same codes always prints in insertion order?

15.16 Solutions to the Exercises

The hand exercises 15.1 to 15.4 are checked by **Listing 15.15**, which also contains the programming solutions 15.6 to 15.9; `Exercises.java` and `exercises.py` print the same output.

Listing 15.15 `exercises.cpp`: checks for Exercises 15.1 to 15.4 and solutions to Exercises 15.6 to 15.9.

```
// exercises.cpp -- checks for the hand exercises and solutions to the
// programming exercises of Chapter 15.
#include <algorithm>
#include <cmath>
#include <cstdint>
#include <cstdio>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>

// Exercise 15.1: chaining, h(k) = k mod 9, insertion at the front of a chain.
void exercise1() {
    const int keys[9] = {5, 28, 19, 15, 20, 33, 12, 17, 10};
    std::vector<std::vector<int>> t(9);
    for (int k : keys) t[k % 9].insert(t[k % 9].begin(), k);
    std::printf("15.1 chaining, m = 9\n");
    for (int b = 0; b < 9; ++b) {
        std::printf("  %d:", b);
        for (int k : t[b]) std::printf(" %d", k);
        std::printf("\n");
    }
}

// Exercise 15.2: open addressing in m = 11 slots with three probe sequences.
int probe2(int k, int i, int mode) {
    if (mode == 0) return (k + i) % 11;                    // linear
    if (mode == 1) return (k + i + 3 * i * i) % 11;        // quadratic
    return (k + i * (1 + k % 10)) % 11;                    // double
}

void exercise2() {
    const int keys[9] = {10, 22, 31, 4, 15, 28, 17, 88, 59};
    const char *name[3] = {"linear   ", "quadratic", "double   "};
    std::printf("15.2 open addressing, m = 11 (- = empty)\n");
    for (int mode = 0; mode < 3; ++mode) {
        std::vector<int> t(11, -1);
        int probes = 0;
        for (int k : keys) {
            for (int i = 0; i < 11; ++i) {
                ++probes;
                int s = probe2(k, i, mode);
                if (t[s] < 0) { t[s] = k; break; }
            }
        }
        std::printf("  %s:", name[mode]);
```

Listing 15.15 (Continued) *exercises.cpp: checks for Exercises 15.1 to 15.4 and solutions to Exercises 15.6 to 15.9.*

```
        for (int x : t) {
            if (x < 0) std::printf("  -");
            else std::printf(" %2d", x);
        }
        std::printf("   (%d probes)\n", probes);
    }
}

// Exercise 15.3: the multiplication method for k = 123456, m = 2^14.
void exercise3() {
    const double A = (std::sqrt(5.0) - 1.0) / 2.0;
    double prod = 123456.0 * A;
    int h = (int)std::floor(16384.0 * (prod - std::floor(prod)));
    std::uint32_t s = 2654435769u;                          // floor(2^32 * A)
    std::uint32_t r0 = (std::uint32_t)123456u * s;          // low 32 bits of k*s
    std::printf("15.3 k*A = %.6f, h = %d; integer form: r0 = %u, h = %u\n", prod, h,
                r0, r0 >> 18);
}

// Exercise 15.4: Horner's rule for "abc", base 31, modulo 101.
void exercise4() {
    long h = 0;
    std::printf("15.4 ");
    for (char c : std::string("abc")) {
        h = (h * 31 + c) % 101;
        std::printf("%ld ", h);
    }
    std::printf("-> h(\"abc\") = %ld\n", h);
}

// Exercise 15.6: the first word that occurs exactly once.
std::string firstUnique(const std::vector<std::string> &w) {
    std::unordered_map<std::string, int> count;
    for (const std::string &x : w) ++count[x];              // pass 1: count
    for (const std::string &x : w)                          // pass 2: first with 1
        if (count[x] == 1) return x;
    return "";
}
```

Listing 15.15 (Continued) *exercises.cpp: checks for Exercises 15.1 to 15.4 and solutions to Exercises 15.6 to 15.9.*

```
// Exercise 15.7: group words that are anagrams of each other.
std::vector<std::vector<std::string>>
groupAnagrams(const std::vector<std::string> &w) {
    std::unordered_map<std::string, std::size_t> groupOf;   // letters -> group
    std::vector<std::vector<std::string>> groups;
    for (const std::string &x : w) {
        std::string key = x;
        std::sort(key.begin(), key.end());
        auto it = groupOf.find(key);
        if (it == groupOf.end()) {
            groupOf[key] = groups.size();
            groups.push_back({x});
        } else {
            groups[it->second].push_back(x);
        }
    }
    return groups;
}

// Exercise 15.8: the longest run of consecutive integers, in O(n) expected time.
int longestRun(const std::vector<int> &a) {
    std::unordered_set<int> s(a.begin(), a.end());
    int best = 0;
    for (int x : s) {
        if (s.count(x - 1)) continue;                  // x does not start a run
        int len = 1;
        while (s.count(x + len)) ++len;
        best = std::max(best, len);
    }
    return best;
}

// Exercise 15.9: how many contiguous stretches of days add up to exactly k?
int countSum(const std::vector<int> &a, int k) {
    std::unordered_map<long, int> seen{{0, 1}};             // prefix sum -> times
    long prefix = 0;
    int count = 0;
    for (int x : a) {
        prefix += x;
        auto it = seen.find(prefix - k);
        if (it != seen.end()) count += it->second;
        ++seen[prefix];
    }
    return count;
}
```

Listing 15.15 (Continued) *exercises.cpp: checks for Exercises 15.1 to 15.4 and solutions to Exercises 15.6 to 15.9.*

```
int main() {
    exercise1();
    exercise2();
    exercise3();
    exercise4();

    std::printf("15.6 first unique: %s\n",
                firstUnique({"kopi", "teh", "kopi", "susu", "teh", "jahe", "susu"})
                    .c_str());
    std::printf("15.7 anagram groups:");
    for (const auto &g : groupAnagrams({"kasur", "rusak", "kita", "kiat", "sukar",
                                        "taki", "buku", "kubu"})) {
        std::printf(" [");
        for (std::size_t i = 0; i < g.size(); ++i)
            std::printf("%s%s", i ? " " : "", g[i].c_str());
        std::printf("]");
    }
    std::printf("\n15.8 longest run: %d\n",
                longestRun({100, 4, 200, 1, 3, 2, 101, 102, 103, 5}));
    std::printf("15.9 stretches summing to 7: %d\n",
                countSum({3, 4, 7, 2, -3, 1, 4, 2}, 7));
    return 0;
}
```

```
15.1 chaining, m = 9
  0:
  1: 10 19 28
  2: 20
  3: 12
  4:
  5: 5
  6: 33 15
  7:
  8: 17
15.2 open addressing, m = 11 (- = empty)
  linear   : 22 88  -  -  4 15 28 17 59 31 10   (16 probes)
  quadratic: 22  - 88 17  4  - 28 59 15 31 10   (23 probes)
  double   : 22  - 59 17  4 15 28 88  - 31 10   (16 probes)
15.3 k*A = 76300.004115, h = 67; integer form: r0 = 17612864, h = 67
15.4 97 75 0 -> h("abc") = 0
15.6 first unique: jahe
15.7 anagram groups: [kasur rusak sukar] [kita kiat taki] [buku kubu]
15.8 longest run: 5
15.9 stretches summing to 7: 4
```

15.1 The home buckets are $5 \bmod 9 = 5$, $28 \bmod 9 = 1$, $19 \to 1$, $15 \to 6$, $20 \to 2$, $33 \to 6$, $12 \to 3$, $17 \to 8$, $10 \to 1$. With front insertion, bucket 1 holds 10, 19, 28 (the last inserted first), bucket 6 holds 33, 15, and buckets 2, 3, 5, 8 hold 20, 12, 5, 17; buckets 0, 4, 7 are empty, as the output shows. The load factor is $9/9 = 1$. A successful search compares with the key and all keys in front of it: 1 for each of the six keys at the front of a chain, 2 for 19 and for 15, 3 for 28, in total $6 + 2 + 2 + 3 = 13$, an average of $13/9 \approx 1.44$, against $1 + (n-1)/(2m) = 1 + 8/18 \approx 1.44$ from Equation (15.12).

15.2 The home slots modulo 11 are 10, 0, 9, 4, 4, 6, 6, 0, 4. (a) Linear: 10, 22, 31, 4 go home; 15 finds 4 taken and takes 5; 28 takes 6; 17 tries 6 and takes 7; 88 tries 0 and takes 1; 59 tries 4, 5, 6, 7 and takes 8. Final table, slots 0 to 10: 22, 88, –, –, 4, 15, 28, 17, 59, 31, 10; 16 probes. (b) Quadratic, with offsets $i + 3i^2 = 0,4,14,30,52, \ldots$: 15 tries 4, then $4 + 4 = 8$, free; 28 goes home to 6; 17 tries 6, 10, 9 and then $6 + 30 = 36 \bmod 11 = 3$; 88 tries 0, 4, 3, 8, 8, 3, 4, 0 and finally 2; 59 tries 4, 8 and takes 7. Final table: 22, –, 88, 17, 4, –, 28, 59, 15, 31, 10; 23 probes. The repetitions in

88's sequence show that this quadratic sequence does not visit every slot of an 11-slot table, so it could fail to find a free slot in a fuller table. (c) Double: 15 has $h_2 = 6$ and tries 4, 10, then 16 mod 11 = 5; 17 has $h_2 = 8$ and tries 6, then 3; 88 has $h_2 = 9$ and tries 0, 9, then 18 mod 11 = 7; 59 has $h_2 = 10$ and tries 4, 3, then 2. Final table: 22, –, 59, 17, 4, 15, 28, 88, –, 31, 10; 16 probes. All three tables agree with the output.

15.3 $kA = 123456 \times 0.6180339887\ldots = 76300.0041151\ldots$; the fractional part is 0.0041151 ...; $16384 \times 0.0041151\ldots = 67.42\ldots$, so $h(k) = 67$. In integer form, $k \cdot s = 327706022297664 = 76300 \cdot 2^{32} + 17612864$, so the low 32 bits are $r_0 = 17612864$, and its top 14 bits are $\lfloor 17612864/2^{18} \rfloor = 67$. Both forms agree.

15.4 $h = 97 \bmod 101 = 97$; then $(97 \cdot 31 + 98) \bmod 101 = 3105 \bmod 101 = 75$; then $(75 \cdot 31 + 99) \bmod 101 = 2424 \bmod 101 = 0$. So $h(\text{"abc"}) = 0$, since $2424 = 24 \cdot 101$.

15.5 Set $1 - e^{-n^2/2m} = 0.01$, so $n^2/(2m) = -\ln 0.99 \approx 0.01005$ and $n \approx \sqrt{0.0201 \cdot 2^{32}} \approx \sqrt{8.63 \times 10^7} \approx 9300$. A 1% chance of a duplicate already arises after about 9300 members, which a university library passes easily. With 64-bit identifiers, $n \approx \sqrt{0.0201 \cdot 1.84 \times 10^{19}} \approx 6.1 \times 10^8$, more than six hundred million members.

15.6 `firstUnique` makes two passes: the first counts each word in a hash map, the second returns the first word whose count is 1. Both passes are $O(n)$ expected time. For the example the counts are kopi 2, teh 2, susu 2, jahe 1, and the answer is *jahe*.

15.7 `groupAnagrams` uses the word's letters in sorted order as the key: *kasur*, *rusak* and *sukar* all give "aksru". A hash map sends each key to the index of its group in a vector, which preserves the order in which groups first appear. Sorting a word of length L costs $O(L \lg L)$, so the total is $O(\sum L \lg L)$ expected. The groups are [kasur rusak sukar] [kita kiat taki] [buku kubu].

15.8 Put all values into a hash set. A value x starts a run exactly when $x - 1$ is not in the set; only from such starts does `longestRun` count upward while $x +$ len is present. Every value is counted in at most one upward walk (the one from its run's start), so the total work is $O(n)$ expected, although there are two nested loops. For the example the starts are 100, 200 and 1, with runs of 4, 1 and 5.

15.9 Let P_j be the sum of the first j days, with $P_0 = 0$. The days $i + 1$ to j add up to $P_j - P_i$, which equals 7 exactly when $P_i = P_j - 7$. So walk the days, keep a hash map from each prefix sum to the number of times it has occurred, and at each j add the number of earlier prefix sums equal to $P_j - 7$. The prefix sums are 0, 3, 7, 14, 16, 13, 14, 18, 20, and the four stretches are days 1 to 2 (3 + 4), day 3 (7), days 3 to 6 (7 + 2 − 3 + 1) and days 6 to 8 (1 + 4 + 2). Negative values are why a two-pointer window does not work here and the hash map does.

15.10 (a) $1500/0.75 = 2000$, so $m = 2048$, giving $\alpha = 1500/2048 \approx 0.732$. (b) Linear probing: $\frac{1}{2}(1 + 1/(1 - 0.7324)^2) = \frac{1}{2}(1 + 1/0.0716) \approx 7.48$ probes; double hashing: $1/(1 - 0.7324) \approx 3.74$ probes. (c) Chaining: $\alpha \approx 0.73$ comparisons, plus the $O(1)$ hash computation.

15.11 In **Figure 15.5**(a) the run is 1003, 1019, 1035, 1051 in slots 11 to 14, all with home slot 11. Deleting 1019 empties slot 12. The scan looks at slot 13: 1035 has home 11, which is not in the cyclic interval (12,13], so 1035 moves back to 12, and slot 13 becomes the gap. At slot 14, 1051 has home 11, not in (13,14], so it moves to 13 and slot 14 becomes the gap. Slot 15 is empty, which ends the run. Final: 1003, 1035, 1051 in slots 11 to 13. A key is moved back only if its home slot lies at or before the gap, so after the move it is still reachable from its home by consecutive occupied slots; a key whose home lies after the gap stays where it is, because a search for it never

passes through the gap. Hence no search sequence is cut, and no tombstone is needed. (The method relies on the fixed step of linear probing; it does not work for quadratic probing or double hashing.)

15.12 (a) `std::map`: its in-order iteration is the code order, with no sorting. (b) `std::unordered_map` (or `unordered_set`): only exact membership is needed, in expected $O(1)$. (c) `std::map` keyed by due date (or `std::multimap` if dates repeat): the next seven days are a range, found with `lower_bound` and `upper_bound`. (d) `std::unordered_map<reader, int>`: exact-key counting, output order irrelevant.

15.13 (a) A rehash happens when the table of 2^j buckets receives its $(2^j + 1)$-st key, that is, when n reaches 2,3,5,9,17, ..., and it moves those $2^j + 1$ keys. If 2^J is the largest table size that has overflowed, then $2^J < n$ and the total is $\sum_{j=0}^{J}(2^j + 1) = 2^{J+1} - 1 + (J + 1) < 2n + \lg n + 1$, which is $O(n)$, fewer than about two moves per insertion, as in Equation (15.23). (b) With m growing by 1000, a rehash happens roughly every 1000 insertions, the t-th one moving about $1000t$ keys; reaching n keys takes about $n/1000$ rehashes, moving in total about $1000(1 + 2 + \cdots + n/1000) \approx n^2/2000$ keys, which is $\Theta(n^2)$: an average of $\Theta(n)$ moves per insertion.

15.14 (a) The attack keys were computed for one known function. A secret chosen at random when the program starts (or per table) is unknown to the attacker, and by Equation (15.25) any fixed pair of keys collides with probability at most $(L - 1)/p$ over the secret's random choice, so a precomputed list of keys is useless. A secret written into the source code is the same in every copy of the program; anyone who reads the code or the binary can compute collisions for it, and the attack is back. (b) Without `compareTo`, the tree cannot order keys with equal hashes. Insertion then orders them by an arbitrary tie-breaker (their class names and identity hash codes), which a later search cannot reproduce from the key, so a search may have to look into both subtrees at each node, and the bin behaves like a list again: $O(\ell)$ for a bin of ℓ entries. The defense works fully only for keys such as `String` that are `Comparable`.

15.15 In CPython's C interface a hash function returns -1 to signal that an error occurred, so no object may have the hash -1; the integer -1 is given -2 instead. A `set` is a plain probed table with no separate `entries` array, so iterating it walks the slots in index order, and the slots depend on the hash values, which for strings change with the random key; a `dict` iterates over its insertion-ordered `entries` array, which does not depend on the hashes at all (**Figure 15.10**).

References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, sec. 6.4.

[2] W. W. Peterson, "Addressing for random-access storage," *IBM Journal of Research and Development*, vol. 1, no. 2, pp. 130–146, Apr. 1957.

[3] Oracle, "Class String: hashCode()," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/String.html. [Accessed: 24-Sep-2026].

[4] cppreference.com, "std::unordered_map." [Online]. Available: https://en.cppreference.com/w/cpp/container/unordered_map. [Accessed: 24-Sep-2026].

[5] ISO/IEC 14882:2017, *Programming Languages — C++*, 5th ed. Geneva, Switzerland: ISO, 2017, sec. 26.2.7 (unordered associative containers).

[6] Oracle, "Class HashMap<K,V>," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/HashMap.html. [Accessed: 24-Sep-2026].

[7] Python Software Foundation, "Objects/dictobject.c," CPython source repository. [Online]. Available: https://github.com/python/cpython/blob/main/Objects/dictobject.c. [Accessed: 24-Sep-2026].

[8] OpenJDK, "src/java.base/share/classes/java/util/HashMap.java," JDK 25 update repository. [Online]. Available: https://github.com/openjdk/jdk25u/blob/master/src/java.base/share/classes/java/util/HashMap.java. [Accessed: 24-Sep-2026].

[9] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proc. 12th USENIX Security Symposium*, Washington, DC, USA, Aug. 2003, pp. 29–44.

[10] A. Klink and J. Wälde, "Efficient denial of service attacks on web application platforms," presented at the 28th Chaos Communication Congress (28C3), Berlin, Germany, Dec. 2011. [Online]. Available: https://fahrplan.events.ccc.de/congress/2011/Fahrplan/attachments/2007_28C3_Effective_DoS_on_web_application_platforms.pdf

[11] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, Apr. 1979.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 11.

[13] J.-P. Aumasson and D. J. Bernstein, "SipHash: A fast short-input PRF," in *Progress in Cryptology – INDOCRYPT 2012*, LNCS, vol. 7668. Berlin, Germany: Springer, 2012, pp. 489–508.

[14] C. Heimes, "PEP 456 – Secure and interchangeable hash algorithm," Python Enhancement Proposals, 2013. [Online]. Available: https://peps.python.org/pep-0456/. [Accessed: 24-Sep-2026].

[15] Python Software Foundation, "What's New In Python 3.11: Other language changes (siphash13, bpo-29410)," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/whatsnew/3.11.html. [Accessed: 24-Sep-2026].

[16] M. Duigou and B. Christian, "JEP 180: Handle frequent HashMap collisions with balanced trees," OpenJDK, delivered in JDK 8. [Online]. Available: https://openjdk.org/jeps/180. [Accessed: 24-Sep-2026].

[17] R. Hettinger, "More compact dictionaries with faster iteration," python-dev mailing list, Dec. 10, 2012. [Online]. Available: https://mail.python.org/pipermail/python-dev/2012-December/123028.html

[18] Python Software Foundation, "What's New In Python 3.7: Summary — release highlights (insertion-order preservation of dict objects)," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/whatsnew/3.7.html. [Accessed: 24-Sep-2026].

[19] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 24-Sep-2026].

Chapter 16

Evaluation and Discussion IV: Graphs, Heaps, and Hash Tables at Work

This last chapter adds no new structure. It gathers every tool of the book, with the weight on the three most recent ones, the graph of Chapter 13, the heap and priority queue of Chapter 14 and the hash table of Chapter 15, and puts them to work on nine problems, each solved from a blank page, one small step at a time, in C++, Java and Python. Two of the problems form a connected pair: completing a max-heap class with five new operations, and putting a pile of travel tickets in order with a hash map. Seven more are new, and each aims at one idea that beginners find slippery: the fewest hops between two places, an order in which to read topics that depend on each other, the quickest route when roads have lengths, the k largest values of a long stream, the middle value of a stream that keeps growing, a hash table built by hand with its chains, and the delicate business of deleting from a table that stores its keys in the array itself. For every problem you will draw the structure before you write code, trace the program by hand, read three complete programs and the output they really print, study a tempting wrong solution and see exactly how it fails, and finish with a *check yourself* variant and its answer. The chapter closes with the question that the whole book has been preparing you to answer: faced with a real task, *which data structure should I choose?* We answer it with a decision table that spans all fifteen chapters and with measurements of every structure on the same ten thousand books of the Pustaka Ceria catalog.

16.1 Where We Stand

Fifteen chapters lie behind us, and it is worth naming what each of them gave us, because half of solving a problem is recognizing which tool it asks for.

The first part of the book was about the array and what can be done with it. Chapter 1 stored the Pustaka Ceria catalog in an array and showed why reading element i costs the same for every i, while inserting in the middle costs a shift of everything after it. Chapter 2 searched: sequential search walks the array, binary search halves a *sorted* array at every step and needs at most $\lfloor \lg n \rfloor + 1$ probes. Chapter 3 sorted, with the elementary sorts, merge sort, quicksort, a first look at heap sort, and the counting sorts. Chapter 4 put those ideas to work.

The second part replaced “a row of cells” with “boxes that point at other boxes”. Chapter 5 built the stack (last in, first out) and the queue (first in, first out). Chapter 6 explained the pointer, and Chapter 7 used it to build the singly linked list; Chapter 8 put stacks, pointers and lists to work.

The third part added direction and hierarchy. Chapter 9 gave every node a second link, backward, and closed the list into a circle. Chapter 10 taught recursion, a function that solves a problem by solving a smaller copy of it. Chapter 11 grew the binary search tree, in which every key in the left subtree of a node is smaller than the node’s key and every key in the right subtree is larger, so that a search goes down one path instead of along the whole collection. Chapter 12 put them to work.

The fourth part, the one this chapter concentrates on, gave us three structures that between them power a large share of the software in the world.

- **Chapter 13, graphs.** A *graph* is a set of *vertices* joined by *edges*; the edges may have a direction and a *weight*. We stored graphs as an adjacency matrix or as adjacency lists, and explored them in two ways: *breadth-first search* (BFS), which visits the vertices in rings of increasing distance with the help of a queue and therefore finds routes with the fewest edges, and *depth-first search* (DFS), which goes as deep as it can before backing up, with the help of a stack or of recursion. On top of those two searches we built connected components, topological sort for graphs without directed cycles, and a first, gentle version of Dijkstra's algorithm for shortest routes when edges have lengths.
- **Chapter 14, heaps and priority queues.** A *priority queue* hands out its most urgent item first. The *binary heap* implements it in an ordinary array that we read as a complete binary tree: the children of index i live at $2i + 1$ and $2i + 2$, and every parent is at least as large as its children (a max-heap) or at most as large (a min-heap). Two repair operations, *sift-up* and *sift-down*, each cost at most the height of the tree, which is $\lfloor \lg n \rfloor$; a whole heap can be built from an unsorted array in linear time; and the standard libraries of all three languages provide one.
- **Chapter 15, hashing.** A *hash function* turns a key into an array index, so that a *hash table* can find a key by computing where it must be instead of searching for it. Two keys that land in the same place *collide*; *separate chaining* keeps a small linked list in every slot, *open addressing* moves the second key to another slot of the same array. When the tables are kept at most moderately full, a lookup, an insertion and a deletion each take constant time on average.

This chapter uses all of these tools, often in one program. **Table 16.1** lists its nine problems, where each is solved, and which chapters supply its main idea. Section 16.8 then steps back and compares every structure of the book on one real task.

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

The running example is still the small library *Pustaka Ceria*. Its books have codes such as `B1001`, it has branches in several neighborhoods of Surabaya linked by a book van, and, for the measurements of Sections 16.6 and 16.8, it has the real catalog of ten thousand books that Chapter 2 introduced.

> **THE ONE IDEA BEHIND ALMOST EVERY PROBLEM HERE.** Chapter 8 found that most pointer bugs break one rule: keep a way to reach everything you still need before you change a link. The structures of this chapter have a rule of the same kind, and it is this: *every structure*

> *promises something, an invariant, and every operation must leave the promise true.* A heap promises that no child is larger than its parent; `extractMax`, `increaseKey` and `buildMaxHeap` are each a small repair of a promise that one change has broken. BFS promises that a vertex leaves the queue in order of distance, so the first time we see a vertex we know its distance. Dijkstra promises that the vertex taken out of the heap has its final distance. A hash table promises that a key is either in the slot its hash names or further along the path that the search will follow. When a program in this chapter goes wrong, it is almost always because an operation left the promise broken, and the wrong solutions below break it in all of these ways.

16.2 How to Approach a Problem About a Structure

The method of Chapter 8, restate, choose, draw, write the invariant, list the edge cases, change things in a safe order, test, carries over unchanged. The structures of this chapter add two habits of their own, so the method gains two steps and becomes nine.

Step 1 — Restate the problem in your own words, with one example. Write down what goes in and what comes out, and work one small input by hand. If you cannot do the example by hand, you are not ready to write code.

Step 2 — Choose the structure, and say why in one sentence. "Always serve the largest" means a heap. "Look things up by name" means a hash table. "Things connected to other things" means a graph. The sentence tells you which operations you need, and Section 16.8 turns such sentences into a table.

Step 3 — Draw it. A heap is drawn twice: as a tree, where the promise is easy to see, and as the array it really is, with the index under every cell. A graph is drawn as circles and lines, with the adjacency lists written beside it in the order the program stores them. A hash table is drawn as a column of slots with its chains or its probe paths.

Step 4 — Write the invariant. For a max-heap: "`heap[(i-1)/2] >= heap[i]` for every $i \geq 1$". For BFS: "the queue holds vertices in order of distance, and the distances in it differ by at most one". For a chained hash table: "a key k, if present, is in the chain of slot $h(k)$".

Step 5 — List the edge cases. For heaps: the empty heap, one element, equal keys, an index outside the array. For graphs: an isolated vertex, a vertex that cannot be reached, a cycle, an edge from a vertex to itself. For hash tables: a key that is absent, two keys that collide, a key that is inserted twice, a deletion followed by a search.

Step 6 — Order the changes safely. Save the root before overwriting it; mark a vertex as seen before it waits in the queue; unlink a node from its chain before freeing it.

Step 7 — Count the cost. New in this chapter, because the whole point of these three structures is speed. Write down how many times each loop runs in the worst case, and compare with the obvious slow solution. If the fast structure is not faster on paper, it will not be faster on the computer.

Step 8 — Decide what happens to input that breaks the promise of the problem. A ticket pile with a loop, a prerequisite list with a cycle, an index past the end of the heap. Choose a behavior (a message, an exception, a special return value) and test it.

Step 9 — Test, including under a memory checker, and against a slow but obviously correct version. Run the example, then every edge case. In C++, compile once more with the sanitizers switched on. And whenever a fast algorithm has a slow, simple twin, sorting everything instead of

keeping a heap, scanning everything instead of hashing, run both and compare; several programs below do exactly that and print the result of the comparison.

Table 16.2 condenses the method onto one page. The labels in bold inside each problem below follow its order.

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?

How the programs of this chapter were checked. Every program below is complete: you can copy it into a file, compile it and run it. The C++ programs were compiled with GCC in C++17 mode with `-Wall -Wextra` and produce no warning; each was compiled a second time with AddressSanitizer and UndefinedBehaviorSanitizer [1], [2] and run under Valgrind [3], and every run was clean, with no invalid access and no leaked byte. The Java programs were compiled with `javac` with all lint warnings enabled and produce none; the Python programs run under Python 3.10 or later. They are written for JDK 25 and Python 3.14 but use nothing newer than JDK 21 and Python 3.10. For every problem the three programs print exactly the same lines, which was checked by comparing their outputs byte for byte; the only exceptions are the measurements of Section 16.8, whose times differ by language, as they should. The commands are those of Chapter 8:

```
g++ -std=c++17 -O2 -Wall -Wextra -o p1601 p1601_max_heap.cpp
./p1601
g++ -std=c++17 -g -Wall -Wextra -fsanitize=address,undefined -o p1601_asan
p1601_max_heap.cpp
./p1601_asan                                    same output, plus a report if memory is
misused
valgrind --leak-check=full ./p1601              Linux: the same check without recompiling
javac P1601MaxHeap.java && java P1601MaxHeap
python3 p1601_max_heap.py
```

Three structures, three languages. The heap, the hash table and the graph exist in the standard library of every language of this book, under different names and with different defaults. **Table 16.3** is a small dictionary; two of its rows cause more bugs than all the others together, and the problems below come back to them.

Table 16.3 The structures of Chapters 13 to 15 in the three languages of this book. The C++ priority queue is a max-heap by default, the Java and Python ones are min-heaps; and 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` functions on a `list`
Which end comes out first	the **largest**	the **smallest**	the **smallest**
A min-heap in C++	`priority_queue<T, vector<T>, greater<T>>`	(the default)	(the default)
A max-heap	(the default)	`new PriorityQueue<>(Collections.reverseOrder())`	store `-x`, or `heapq.heappush_max` (Python 3.14)
Hash table (map)	`std::unordered_map<K, V>`	`HashMap<K, V>`	`dict`
Hash set	`std::unordered_set<K>`	`HashSet<K>`	`set`
Ordered map (a balanced search tree)	`std::map<K, V>`	`TreeMap<K, V>`	none in the standard library
FIFO queue for BFS	`std::queue<int>`	`ArrayDeque<Integer>`	`collections.deque`
Adjacency lists	`vector<vector<int>>`	`List<List<Integer>>`	`list` of `list`

The first dangerous row is the direction of the priority queue: `std::priority_queue` hands out the largest element, `PriorityQueue` and `heapq` the smallest, and code copied from one language to another quietly turns into its opposite (Problem 16.6 shows the consequence). The second is the word *map*: in Java, `HashMap` says what it is, but in C++ the short, natural name `std::map` belongs to a balanced binary search tree, and the hash table has the longer name `std::unordered_map` [4], [5], [6], [7]. Problem 16.2 is where that distinction matters.

16.3 Problem 16.1 — Completing a Max-Heap Class

16.3.1 The problem

> **PROBLEM.** A C++ class `MaxHeap` stores a max-heap of integers in a `vector<int> heap` and already provides `insert`, `deleteRoot`, `getMax`, `heapifyArray`, `heapSort`, `printHeap` and two private helpers, `heapifyDown(index)` and `heapifyUp(index)`. Add five member functions and a `clear()` that empties the heap: (1) `int extractMax()`, which removes and returns the maximum element; (2) `void increaseKey(int index, int newValue)`, which increases the value of the key at a given index; (3) `int getMin()`, which returns the minimum element of the max-heap without removing it; (4) `void buildMaxHeap(const vector<int>& arr)`, which builds a max-heap from an unsorted array; (5) `bool isMaxHeap(const vector<int>& arr)`, which checks whether a given array is a max-heap. The driver clears the heap, inserts 3, 1, 4, 1,

5, 9, 2, 6, 5, 3, 5, calls `extractMax()`, `increaseKey(2, 8)` and `getMin()`, and then, for each of the arrays {10, 7, 8, 5, 2, 3, 1} and {10, 7, 6, 5, 1, 8, 2}, calls `buildMaxHeap` and `isMaxHeap` and prints the results.

The class to be completed is the one that Chapter 14 builds in full, and it helps to recall its two private helpers exactly, because every new operation is made of them. `heapifyDown(i)` looks at the element at index i and its two children; if a child is larger, it swaps the element with the *larger* child and continues from there, so a too-small element sinks until both of its children are no larger. `heapifyUp(i)` compares the element with its parent; if the element is larger, it swaps them and continues from the parent, so a too-large element rises until its parent is no smaller. Chapter 14 calls these *sift-down* and *sift-up*; the class keeps the names `heapifyDown` and `heapifyUp`, and so shall we.

16.3.2 In our own words

We have an array `heap` whose element 0 is the root of a complete binary tree, and the relatives of the element at index i are found by arithmetic:

$$\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \qquad \text{left}(i) = 2i + 1, \qquad \text{right}(i) = 2i + 2. \tag{16.1}$$

DEFINITION. The *max-heap invariant*: for every index i with $1 \le i < n$, `heap[parent(i)] >= heap[i]`. In words, no element is larger than its parent. It follows, by walking up from any element to the root, that `heap[0]` is the largest element of the heap.

Each of the five functions has a one-sentence job, and each sentence can be read off the invariant. `extractMax` must take out the root and leave a heap behind. `increaseKey` makes one element larger, which can only offend its parent, never its children, so it needs a repair *upward*. `getMin` must find the smallest element, which the invariant does not place at any fixed index, but, as we shall prove, always places among the leaves. `buildMaxHeap` must turn an arbitrary array into a heap, faster than inserting its elements one by one. `isMaxHeap` must check the invariant for every i, which is a single loop.

16.3.3 By hand, step by step

Building the heap by insertion. The driver inserts eleven values. Each insertion appends the value at the end of the array, the next free position of the complete tree, and lets it rise. Follow the three insertions that move something. Inserting 4 into `[3, 1]` appends it at index 2, whose parent is index $\lfloor 1/2 \rfloor = 0$ holding 3; since $4 > 3$ they swap: `[4, 1, 3]`. Inserting 5 into `[4, 1, 3, 1]` appends it at index 4, parent index 1 holding 1: swap; now at index 1, parent index 0 holding 4: swap again, and 5 is the root: `[5, 4, 3, 1, 1]`. Inserting 9 appends it at index 5, parent index 2 holding 3: swap; at index 2, parent index 0 holding 5: swap; `[9, 4, 5, 1, 1, 3]`. The other eight insertions go the same way, and after all eleven the array is the one in **Figure 16.1**.

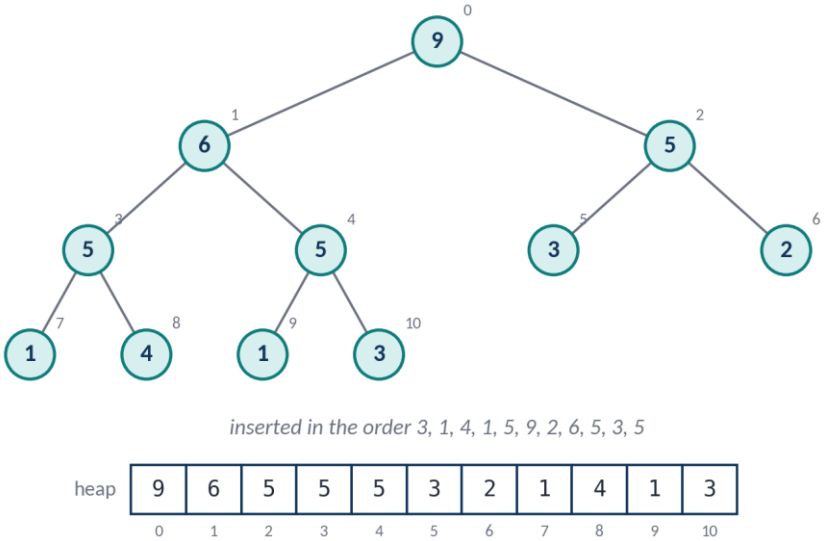


Figure 16.1 The max-heap after the eleven insertions of the driver, drawn as a tree (small grey numbers are array indices) and as the array it really is.

Check the invariant on the picture: 9 is above 6 and 5; 6 is above 5 and 5; the 5 at index 3 is above 1 and 4; the 5 at index 4 is above 1 and 3; the 5 at index 2 is above 3 and 2. Every parent is at least its children. Notice that the array is *not* sorted, and need not be: 4 at index 8 is larger than 1 at index 7 and larger than 3 at index 5, and nobody minds, because they are not parent and child.

1. `extractMax`. Figure 16.2 shows the three moves. The answer is `heap[0]`, 9, and the first thing to do is to *save it in a local variable*, because the next move overwrites it. The tree must shrink by one element, and the only element that can disappear without making a hole in the middle of a complete tree is the last one, index 10 holding 3. So we copy that 3 into index 0 and remove the last cell. The tree is complete again, but its root, 3, is smaller than both of its children, 6 and 5; the invariant is broken at exactly one place, the top, and a broken promise at the top is what `heapifyDown(0)` repairs. The larger child is 6 at index 1: swap. Now 3 sits at index 1 with children 5 (index 3) and 5 (index 4); the larger is the first 5 (the code takes the left child when the two are equal): swap. Now 3 sits at index 3 with children 1 and 4: swap with 4. At index 8 it has no children, and the repair stops. The function returns the saved 9, and the array is `[6, 5, 5, 4, 5, 3, 2, 1, 3, 1]`.

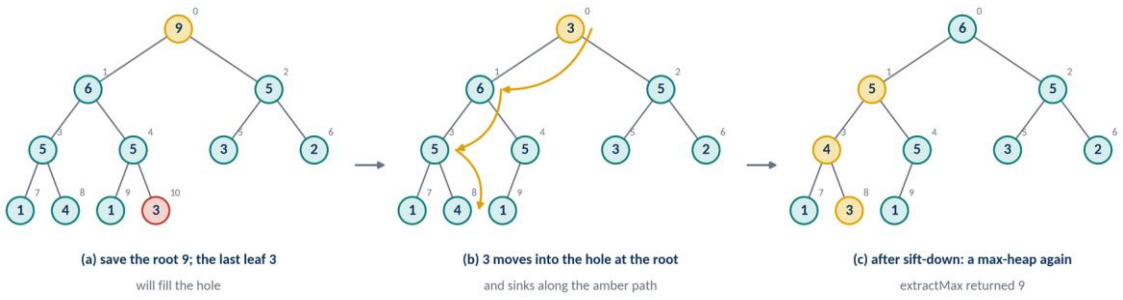


Figure 16.2 `extractMax` in three moves: save the root, move the last leaf into the root, and let it sink along the path of larger children until the max-heap invariant holds again.

How many swaps can this take? Each swap moves the sinking element down one level, and a complete tree with n elements has only so many levels. Level d can hold 2^d elements, so the levels 0 to h together hold at most $2^{h+1} - 1$ elements, and the last level must hold at least one. Hence $2^h \leq n$, and

$$h = \lfloor \lg n \rfloor. \tag{16.2}$$

By Equation (16.2), a heap of 10 elements has height 3, and indeed the 3 above made exactly three swaps. Every operation that sinks or rises along one path therefore costs $O(\lg n)$.

2. `increaseKey(2, 8)`**.** The element at index 2 is 5, and it becomes 8. A larger value cannot harm the relation with its children, who were at most 5 and are certainly at most 8. It can harm only the relation with its parent: index $\lfloor (2-1)/2 \rfloor = 0$ holds 6, and $8 > 6$. So the repair goes *up*, with `heapifyUp(2)`: one swap, and 8 is the root. **Figure 16.3** shows it. The result is `[8, 5, 6, 4, 5, 3, 2, 1, 3, 1]`.

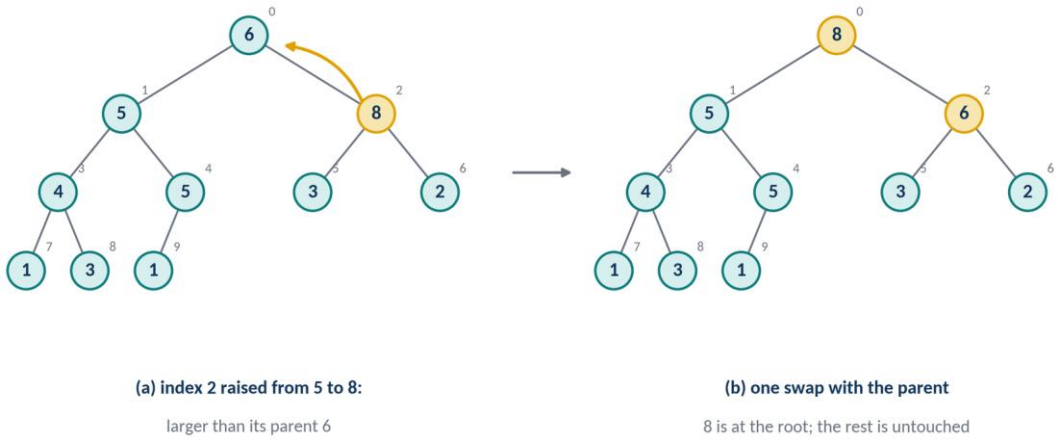


Figure 16.3 `increaseKey(2, 8)`: the raised key can only be too large for its parent, so it rises; one swap restores the invariant.

What if the caller passes a *smaller* value? Then the element could become smaller than its children, and rising would be the wrong repair. The name of the function promises an increase, so our version refuses a decrease with an exception rather than silently doing the wrong thing; Exercise 16.1 writes the mirror function `decreaseKey`, whose repair goes down.

3. `getMin`**.** Where can the smallest element be? Not at any index that has a child, because that child would be at most as large, so the element would not be smaller than everything. The minimum is therefore a *leaf*, an element without children. Which indices are leaves? Index i has a left child exactly when $2i + 1 \le n - 1$, that is, when $i \le (n-2)/2$. So the leaves are all the indices from $\lfloor n/2 \rfloor$ up:

$$\text{leaves} = \{\, \lfloor n/2 \rfloor,\ \lfloor n/2 \rfloor + 1,\ \ldots,\ n-1 \,\}, \qquad \text{about } n/2 \text{ of them.} \tag{16.3}$$

Figure 16.4 marks them for our heap of $n = 10$: indices 5 to 9, holding 3, 2, 1, 3, 1. The smallest is 1. By Equation (16.3), `getMin` looks at $\lceil n/2 \rceil$ elements instead of n: half the work of a plain scan, but still $\Theta(n)$. A max-heap simply does not keep its minimum anywhere special. If both extremes are needed quickly, one keeps two heaps, as Problem 16.7 does for a different reason.

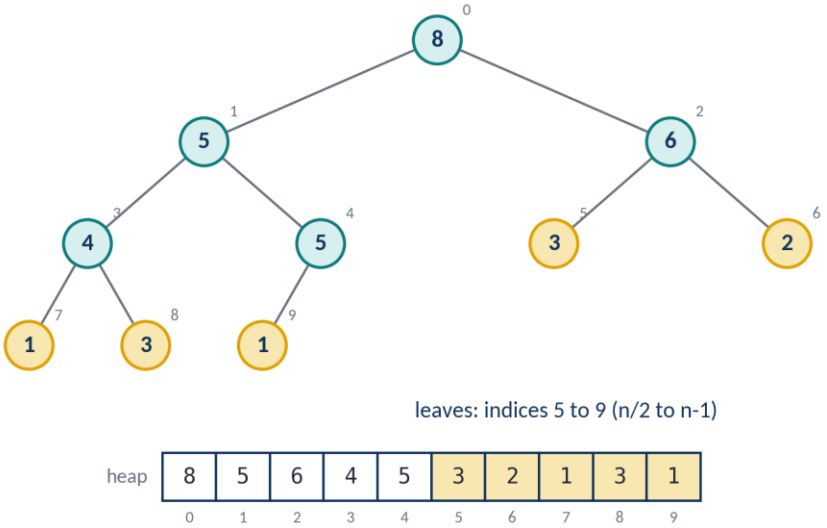


Figure 16.4 The minimum of a max-heap is always a leaf, and the leaves are the last half of the array: indices $\lfloor n/2 \rfloor$ to $n-1$.

4. `buildMaxHeap`. The simplest way to build a heap from an array is to insert its elements one by one, at a cost of up to lg$n$ swaps each, $O(n\lg n)$ in all. Floyd's bottom-up method [8] is faster. Copy the array as it is, and then call `heapifyDown` on every element that has a child, *from the last such element back to the root*. Why that order? `heapifyDown(i)` assumes that the two subtrees below i are already heaps; working from the bottom up guarantees it, since every subtree below i was repaired before i. By Equation (16.3) the elements with children are indices $\lfloor n/2 \rfloor - 1$ down to 0.

Take the second array of the driver, {10, 7, 6, 5, 1, 8, 2}, $n = 7$, drawn in **Figure 16.5**. The internal indices are 2, 1, 0. At index 2 the element 6 has children 8 and 2; 8 is larger: swap. At index 1 the element 7 has children 5 and 1: nothing to do. At index 0 the element 10 has children 7 and 8: nothing to do. The result is `[10, 7, 8, 5, 1, 6, 2]`. The first array, {10, 7, 8, 5, 2, 3, 1}, is already a heap, and every call finds nothing to do.

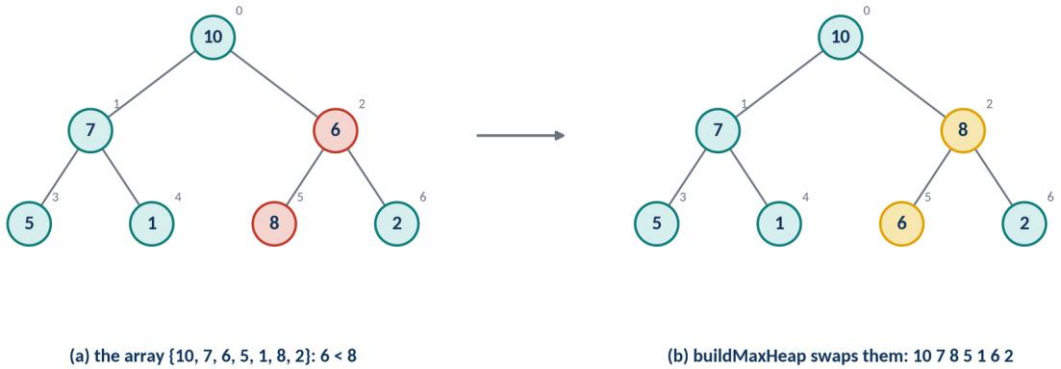


Figure 16.5 `buildMaxHeap` on {10, 7, 6, 5, 1, 8, 2}: only index 2 violates the invariant, and one swap repairs it.

Why is this linear? Most elements are near the bottom, where sinking is cheap. At most $n/2^{j+1}$ elements sit at height j, and each sinks at most j levels, so the total number of swaps is at most

$$\sum_{j=0}^{\lfloor \lg n \rfloor} \frac{n}{2^{j+1}}\, j \;=\; \frac{n}{2}\sum_{j\geq 0} \frac{j}{2^{j}} \;\leq\; \frac{n}{2}\cdot 2 \;=\; n. \tag{16.4}$$

The sum $\sum_{j \geq 0} j / 2^j$ equals 2 (Chapter 14 derives it), so Equation (16.4) bounds the work of `buildMaxHeap` by $O(n)$ [9].

5. `isMaxHeap`. The invariant quantifies over every i from 1 to $n - 1$, so the check is one loop that compares every element with its parent and stops at the first offender. For {10, 7, 8, 5, 2, 3, 1}: $7 \leq 10$, $8 \leq 10$, $5 \leq 7$, $2 \leq 7$, $3 \leq 8$, $1 \leq 8$: a heap. For {10, 7, 6, 5, 1, 8, 2}: at $i = 5$, the parent index 2 holds 6 and $8 > 6$: not a heap. Note what the driver asks. It passes the *original* array to `isMaxHeap`, not the heap built from it; so the second answer is "No", even though `buildMaxHeap` has just produced a perfectly good heap from it. Our program asks both questions, to make the difference visible. An empty array and an array of one element have no i in the range $1 \leq i < n$ at all, so for them the answer is "Yes" without a single comparison, as it should be: there is nothing to violate.

16.3.4 The C++ solution

Listing 16.1 is the completed class. The given operations are kept with their names; we changed only what they do on an empty heap (`getMax` throws instead of returning a made-up -1, for the same reason as the stack of Chapter 8: -1 is a perfectly good key) and used `size_t` for indices so that the compiler has no signed/unsigned comparison to warn about. `main` runs the given driver, labels its second half "The five new operations", and adds edge cases at the end.

Listing 16.1 `p1601_max_heap.cpp`: the max-heap class of Problem 16.1 with `extractMax`, `increaseKey`, `getMin`, `buildMaxHeap`, `isMaxHeap` and `clear`.

```cpp
// p1601_max_heap.cpp - Problem 16.1: a max-heap class with five new operations.
#include <iostream>
#include <stdexcept>
#include <vector>
using namespace std;

class MaxHeap {
private:
    vector<int> heap;   // heap[0] is the root; the children of i are 2i+1 and 2i+2

    // Moves heap[index] down until it is at least as large as both children.
    void heapifyDown(size_t index) {
        size_t largest = index;
        size_t left = 2 * index + 1;
        size_t right = 2 * index + 2;
        if (left < heap.size() && heap[left] > heap[largest]) largest = left;
        if (right < heap.size() && heap[right] > heap[largest]) largest = right;
        if (largest != index) {
            swap(heap[index], heap[largest]);
            heapifyDown(largest);
        }
    }

    // Moves heap[index] up until its parent is at least as large as it.
    void heapifyUp(size_t index) {
        if (index > 0 && heap[(index - 1) / 2] < heap[index]) {
            swap(heap[index], heap[(index - 1) / 2]);
            heapifyUp((index - 1) / 2);
        }
    }

public:
    // ---------------- the operations that were already there ----------------
    void insert(int key) {
        heap.push_back(key);
        heapifyUp(heap.size() - 1);
    }
```

Listing 16.1 (Continued) *p1601_max_heap.cpp: the max-heap class of Problem 16.1 with extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap and clear.*

```
    void deleteRoot() {
        if (heap.empty()) {
            cout << "Heap is empty\n";
            return;
        }
        heap[0] = heap.back();
        heap.pop_back();
        if (!heap.empty()) heapifyDown(0);
    }

    int getMax() const {
        if (heap.empty()) throw underflow_error("getMax on an empty heap");
        return heap[0];
    }

    void heapifyArray(const vector<int>& arr) {
        heap = arr;
        for (size_t i = heap.size() / 2; i-- > 0;) heapifyDown(i);
    }

    vector<int> heapSort() {              // empties the heap, largest first
        vector<int> sorted;
        while (!heap.empty()) {
            sorted.push_back(getMax());
            deleteRoot();
        }
        return sorted;
    }

    void printHeap() const {
        if (heap.empty()) cout << "(empty)";
        for (size_t i = 0; i < heap.size(); i++) cout << (i ? " " : "") << heap[i];
        cout << "\n";
    }

    // ---------------- the five new operations, and clear ----------------
    // 1. Removes and returns the largest element.
    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;
    }
```

Listing 16.1 (Continued) *p1601_max_heap.cpp: the max-heap class of Problem 16.1 with extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap and clear.*

```
    // 2. Raises heap[index] to newValue; a larger key can only move UP.
    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);
    }

    // 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;
    }

    // 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;
    }

    void clear() { heap.clear(); }

    const vector<int>& data() const { return heap; }
};

static const char* yesNo(bool b) { return b ? "Yes" : "No"; }
```

Listing 16.1 (Continued) *p1601_max_heap.cpp: the max-heap class of Problem 16.1 with extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap and clear.*

```
int main() {
    MaxHeap h;
    for (int x : {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}) h.insert(x);
    cout << "Max-Heap array: ";
    h.printHeap();
    cout << "The max element is " << h.getMax() << "\n";
    h.deleteRoot();
    cout << "After deleting the root, heap array: ";
    h.printHeap();
    h.heapifyArray({10, 20, 15, 30, 40});
    cout << "Heapified array: ";
    h.printHeap();
    cout << "Heap sorted array:";
    for (int x : h.heapSort()) cout << " " << x;
    cout << "\n";

    cout << "\n=== The five new operations ===\n\n";
    cout << "Clearing the existing heap ...\n";
    h.clear();
    cout << "Heap after clearing: ";
    h.printHeap();
    for (int x : {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}) h.insert(x);
    cout << "Max-Heap array: ";
    h.printHeap();

    cout << "\n1. Extracted Max: " << h.extractMax() << "\n";
    cout << "Max-Heap array: ";
    h.printHeap();

    h.increaseKey(2, 8);
    cout << "\n2. Heap after increasing key at index 2 to 8: ";
    h.printHeap();

    cout << "\n3. Minimum value in the heap: " << h.getMin() << "\n\n";

    vector<int> a1 = {10, 7, 8, 5, 2, 3, 1};
    cout << "4a. BuildMaxHeap()\nArray: {10, 7, 8, 5, 2, 3, 1}\n";
    h.clear();
    h.buildMaxHeap(a1);
    cout << "Max-Heap built from unsorted array: ";
    h.printHeap();
    cout << "\n5a. IsMaxHeap()\n";
    cout << "Is the array {10, 7, 8, 5, 2, 3, 1} a Max-Heap? "
         << yesNo(MaxHeap::isMaxHeap(a1)) << "\n";
```

Listing 16.1 (Continued) *p1601_max_heap.cpp: the max-heap class of Problem 16.1 with extractMax, increaseKey, getMin, buildMaxHeap, isMaxHeap and clear.*

```
    vector<int> a2 = {10, 7, 6, 5, 1, 8, 2};
    cout << "\n4b. BuildMaxHeap()\nArray: {10, 7, 6, 5, 1, 8, 2}\n";
    h.clear();
    h.buildMaxHeap(a2);
    cout << "Max-Heap built from unsorted array: ";
    h.printHeap();
    cout << "\n5b. IsMaxHeap()\n";
    cout << "Is the array {10, 7, 6, 5, 1, 8, 2} a Max-Heap? "
         << yesNo(MaxHeap::isMaxHeap(a2)) << "\n";
    cout << "Is the heap built from it a Max-Heap? "
         << yesNo(MaxHeap::isMaxHeap(h.data())) << "\n";

    cout << "\n=== Edge cases ===\n";
    cout << "isMaxHeap({}) -> " << yesNo(MaxHeap::isMaxHeap({})) << ", "
         << "isMaxHeap({7}) -> " << yesNo(MaxHeap::isMaxHeap({7})) << "\n";
    try {
        h.increaseKey(0, 1);
    } catch (const invalid_argument& e) {
        cout << "caught: " << e.what() << "\n";
    }
    try {
        h.increaseKey(7, 99);
    } catch (const out_of_range& e) {
        cout << "caught: " << e.what() << "\n";
    }
    h.clear();
    try {
        h.extractMax();
    } catch (const underflow_error& e) {
        cout << "caught: " << e.what() << "\n";
    }
    return 0;
}
```

```
Max-Heap array: 9 6 5 5 5 3 2 1 4 1 3
The max element is 9
After deleting the root, heap array: 6 5 5 4 5 3 2 1 3 1
Heapified array: 40 30 15 10 20
Heap sorted array: 40 30 20 15 10

=== The five new operations ===

Clearing the existing heap ...
Heap after clearing: (empty)
Max-Heap array: 9 6 5 5 5 3 2 1 4 1 3

1. Extracted Max: 9
Max-Heap array: 6 5 5 4 5 3 2 1 3 1

2. Heap after increasing key at index 2 to 8: 8 5 6 4 5 3 2 1 3 1

3. Minimum value in the heap: 1

4a. BuildMaxHeap()
Array: {10, 7, 8, 5, 2, 3, 1}
Max-Heap built from unsorted array: 10 7 8 5 2 3 1

5a. IsMaxHeap()
Is the array {10, 7, 8, 5, 2, 3, 1} a Max-Heap? Yes

4b. BuildMaxHeap()
Array: {10, 7, 6, 5, 1, 8, 2}
Max-Heap built from unsorted array: 10 7 8 5 1 6 2

5b. IsMaxHeap()
Is the array {10, 7, 6, 5, 1, 8, 2} a Max-Heap? No
Is the heap built from it a Max-Heap? Yes

=== Edge cases ===
isMaxHeap({}) -> Yes, isMaxHeap({7}) -> Yes
caught: increaseKey: new value is smaller
caught: increaseKey: no element at that index
caught: extractMax on an empty heap
```

Line by line. The first five lines come from the operations that were already there: the eleven insertions give the array of **Figure 16.1**; deleting the root gives the same array as `extractMax` will; `heapifyArray` turns {10, 20, 15, 30, 40} into `40 30 15 10 20`; and heap sort, which repeatedly takes the maximum, prints the five values largest first. The second half follows Section 16.3.3 exactly: 9 extracted, 8 at the root after `increaseKey(2, 8)`, 1 as the minimum, the two built heaps, and the two answers of `isMaxHeap`. The extra line "Is the heap built from it a Max-Heap? Yes" confirms that `buildMaxHeap` did its job on the array that is not a heap. The edge cases show that an empty and a one-element array count as heaps, that a decrease passed to `increaseKey` is refused, that index 7 is refused on a heap of seven elements (valid indices are 0 to 6), and that `extractMax` on an empty heap throws instead of reading `heap[0]`, which does not exist.

Three details deserve a sentence each. The loop `for (size_t i = heap.size() / 2; i-- > 0;)` counts down from $\lfloor n/2 \rfloor - 1$ to 0 with an *unsigned* counter; the test `i-- > 0` checks before decrementing, so the body sees $\lfloor n/2 \rfloor - 1, \ldots, 0$ and the loop ends without ever computing -1, which an unsigned variable cannot hold. `isMaxHeap` is `static` because it inspects the array it is given, not the object's own heap. And `extractMax` calls `heapifyDown(0)` only if the heap is not empty after the removal, because a heap that had one element is now empty and has no index 0.

Cost. By Equation (16.2), `extractMax`, `increaseKey` and `insert` cost $O(\lg n)$; `getMin` and `isMaxHeap` cost $\Theta(n)$ in the worst case; `buildMaxHeap` costs $O(n)$ by Equation (16.4); `clear` costs

$O(n)$ because `vector::clear` destroys every element (for `int`, the compiler makes that essentially free).

16.3.5 The Java and Python solutions

Listing 16.2 and **Listing 16.3** print exactly the lines above. Java keeps the heap in an `ArrayList<Integer>` and throws `NoSuchElementException` on an empty heap, the exception its own collections use; Python keeps a `list`, raises `IndexError` like its own `list.pop()`, and writes `getMin` as `min(self.heap[len(self.heap) // 2:])`, the slice of Equation (16.3). In real programs you would reach for the library heap of **Table 16.3**; the point here is to write the operations once yourself.

Listing 16.2 `P1601MaxHeap.java`: Problem 16.1 in Java.

```
// P1601MaxHeap.java - Problem 16.1: a max-heap class with five new operations.
import java.util.ArrayList;
import java.util.List;
import java.util.NoSuchElementException;

public class P1601MaxHeap {

    static class MaxHeap {
        // heap.get(0) is the root; the children of i are 2i+1 and 2i+2
        private final ArrayList<Integer> heap = new ArrayList<>();

        private void swap(int i, int j) {
            int t = heap.get(i);
            heap.set(i, heap.get(j));
            heap.set(j, t);
        }

        private void heapifyDown(int index) {
            int largest = index, left = 2 * index + 1, right = 2 * index + 2;
            int n = heap.size();
            if (left < n && heap.get(left) > heap.get(largest)) largest = left;
            if (right < n && heap.get(right) > heap.get(largest)) largest = right;
            if (largest != index) {
                swap(index, largest);
                heapifyDown(largest);
            }
        }

        private void heapifyUp(int index) {
            if (index > 0 && heap.get((index - 1) / 2) < heap.get(index)) {
                swap(index, (index - 1) / 2);
                heapifyUp((index - 1) / 2);
            }
        }

        // ---------------- the operations that were already there ----------------
        void insert(int key) {
            heap.add(key);
            heapifyUp(heap.size() - 1);
        }
```

Listing 16.2 (Continued) *P1601MaxHeap.java: Problem 16.1 in Java.*

```
        void deleteRoot() {
            if (heap.isEmpty()) {
                System.out.println("Heap is empty");
                return;
            }
            heap.set(0, heap.get(heap.size() - 1));
            heap.remove(heap.size() - 1);
            if (!heap.isEmpty()) heapifyDown(0);
        }

        int getMax() {
            if (heap.isEmpty())
                throw new NoSuchElementException("getMax on an empty heap");
            return heap.get(0);
        }

        void heapifyArray(List<Integer> arr) {
            buildMaxHeap(arr);
        }

        List<Integer> heapSort() {             // empties the heap, largest first
            List<Integer> sorted = new ArrayList<>();
            while (!heap.isEmpty()) {
                sorted.add(getMax());
                deleteRoot();
            }
            return sorted;
        }

        void printHeap() {
            System.out.println(heap.isEmpty() ? "(empty)" : join(heap));
        }

        // ---------------- the five new operations, and clear ----------------
        int extractMax() {
            if (heap.isEmpty())
                throw new NoSuchElementException("extractMax on an empty heap");
            int maxVal = heap.get(0);                  // SAVE the answer first
            heap.set(0, heap.get(heap.size() - 1));    // last leaf fills the hole
            heap.remove(heap.size() - 1);
            if (!heap.isEmpty()) heapifyDown(0);
            return maxVal;
        }
```

Listing 16.2 (Continued) *P1601MaxHeap.java: Problem 16.1 in Java.*

```
        void increaseKey(int index, int newValue) {
            if (index < 0 || index >= heap.size())
                throw new IndexOutOfBoundsException(
                        "increaseKey: no element at that index");
            if (newValue < heap.get(index))
                throw new IllegalArgumentException(
                        "increaseKey: new value is smaller");
            heap.set(index, newValue);
            heapifyUp(index);
        }

        int getMin() {                  // the minimum is a leaf: heap[n/2 .. n-1]
            if (heap.isEmpty())
                throw new NoSuchElementException("getMin on an empty heap");
            int n = heap.size();
            int minVal = heap.get(n / 2);
            for (int i = n / 2 + 1; i < n; i++)
                minVal = Math.min(minVal, heap.get(i));
            return minVal;
        }

        void buildMaxHeap(List<Integer> arr) {
            heap.clear();
            heap.addAll(arr);
            for (int i = heap.size() / 2 - 1; i >= 0; i--) heapifyDown(i);
        }

        static boolean isMaxHeap(List<Integer> arr) {
            for (int i = 1; i < arr.size(); i++)
                if (arr.get((i - 1) / 2) < arr.get(i)) return false;
            return true;
        }

        void clear() {
            heap.clear();
        }

        List<Integer> data() {
            return heap;
        }
    }
```

Listing 16.2 (Continued) *P1601MaxHeap.java: Problem 16.1 in Java.*

```
    static String join(List<Integer> xs) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < xs.size(); i++)
            sb.append(i > 0 ? " " : "").append(xs.get(i));
        return sb.toString();
    }

    static String yesNo(boolean b) {
        return b ? "Yes" : "No";
    }

    public static void main(String[] args) {
        MaxHeap h = new MaxHeap();
        for (int x : new int[] {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}) h.insert(x);
        System.out.print("Max-Heap array: ");
        h.printHeap();
        System.out.println("The max element is " + h.getMax());
        h.deleteRoot();
        System.out.print("After deleting the root, heap array: ");
        h.printHeap();
        h.heapifyArray(List.of(10, 20, 15, 30, 40));
        System.out.print("Heapified array: ");
        h.printHeap();
        System.out.println("Heap sorted array: " + join(h.heapSort()));

        System.out.println("\n=== The five new operations ===\n");
        System.out.println("Clearing the existing heap ...");
        h.clear();
        System.out.print("Heap after clearing: ");
        h.printHeap();
        for (int x : new int[] {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}) h.insert(x);
        System.out.print("Max-Heap array: ");
        h.printHeap();

        System.out.println("\n1. Extracted Max: " + h.extractMax());
        System.out.print("Max-Heap array: ");
        h.printHeap();

        h.increaseKey(2, 8);
        System.out.print("\n2. Heap after increasing key at index 2 to 8: ");
        h.printHeap();

        System.out.println("\n3. Minimum value in the heap: " + h.getMin() + "\n");
```

Listing 16.2 (Continued) *P1601MaxHeap.java: Problem 16.1 in Java.*

```
        List<Integer> a1 = List.of(10, 7, 8, 5, 2, 3, 1);
        System.out.println("4a. BuildMaxHeap()\nArray: {10, 7, 8, 5, 2, 3, 1}");
        h.clear();
        h.buildMaxHeap(a1);
        System.out.print("Max-Heap built from unsorted array: ");
        h.printHeap();
        System.out.println("\n5a. IsMaxHeap()");
        System.out.println("Is the array {10, 7, 8, 5, 2, 3, 1} a Max-Heap? "
                + yesNo(MaxHeap.isMaxHeap(a1)));

        List<Integer> a2 = List.of(10, 7, 6, 5, 1, 8, 2);
        System.out.println("\n4b. BuildMaxHeap()\nArray: {10, 7, 6, 5, 1, 8, 2}");
        h.clear();
        h.buildMaxHeap(a2);
        System.out.print("Max-Heap built from unsorted array: ");
        h.printHeap();
        System.out.println("\n5b. IsMaxHeap()");
        System.out.println("Is the array {10, 7, 6, 5, 1, 8, 2} a Max-Heap? "
                + yesNo(MaxHeap.isMaxHeap(a2)));
        System.out.println("Is the heap built from it a Max-Heap? "
                + yesNo(MaxHeap.isMaxHeap(h.data())));

        System.out.println("\n=== Edge cases ===");
        System.out.println("isMaxHeap({}) -> " + yesNo(MaxHeap.isMaxHeap(List.of()))
                + ", isMaxHeap({7}) -> " + yesNo(MaxHeap.isMaxHeap(List.of(7))));
        try {
            h.increaseKey(0, 1);
        } catch (IllegalArgumentException e) {
            System.out.println("caught: " + e.getMessage());
        }
        try {
            h.increaseKey(7, 99);
        } catch (IndexOutOfBoundsException e) {
            System.out.println("caught: " + e.getMessage());
        }
        h.clear();
        try {
            h.extractMax();
        } catch (NoSuchElementException e) {
            System.out.println("caught: " + e.getMessage());
        }
    }
}
```

Listing 16.3 `p1601_max_heap.py`: Problem 16.1 in Python.

```python
# p1601_max_heap.py - Problem 16.1: a max-heap class with five new operations.


class MaxHeap:
    def __init__(self):
        self.heap = []              # heap[0] is the root; children of i: 2i+1, 2i+2

    def _heapify_down(self, index):
        largest, left, right = index, 2 * index + 1, 2 * index + 2
        h = self.heap
        if left < len(h) and h[left] > h[largest]:
            largest = left
        if right < len(h) and h[right] > h[largest]:
            largest = right
        if largest != index:
            h[index], h[largest] = h[largest], h[index]
            self._heapify_down(largest)

    def _heapify_up(self, index):
        h = self.heap
        if index > 0 and h[(index - 1) // 2] < h[index]:
            p = (index - 1) // 2
            h[index], h[p] = h[p], h[index]
            self._heapify_up(p)

    # ---------------- the operations that were already there ----------------
    def insert(self, key):
        self.heap.append(key)
        self._heapify_up(len(self.heap) - 1)

    def delete_root(self):
        if not self.heap:
            print("Heap is empty")
            return
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        if self.heap:
            self._heapify_down(0)

    def get_max(self):
        if not self.heap:
            raise IndexError("getMax on an empty heap")
        return self.heap[0]
```

Listing 16.3 (Continued) *p1601_max_heap.py: Problem 16.1 in Python.*

```
    def heapify_array(self, arr):
        self.build_max_heap(arr)

    def heap_sort(self):                    # empties the heap, largest first
        out = []
        while self.heap:
            out.append(self.get_max())
            self.delete_root()
        return out

    def print_heap(self):
        print(" ".join(map(str, self.heap)) if self.heap else "(empty)")

    # ---------------- the five new operations, and clear ----------------
    def extract_max(self):
        if not self.heap:
            raise IndexError("extractMax on an empty heap")
        max_val = self.heap[0]              # SAVE the answer first
        self.heap[0] = self.heap[-1]        # the last leaf fills the hole
        self.heap.pop()
        if self.heap:
            self._heapify_down(0)
        return max_val

    def increase_key(self, index, new_value):
        if not 0 <= index < len(self.heap):
            raise IndexError("increaseKey: no element at that index")
        if new_value < self.heap[index]:
            raise ValueError("increaseKey: new value is smaller")
        self.heap[index] = new_value
        self._heapify_up(index)

    def get_min(self):              # the minimum is a leaf: heap[n//2 .. n-1]
        if not self.heap:
            raise IndexError("getMin on an empty heap")
        return min(self.heap[len(self.heap) // 2:])

    def build_max_heap(self, arr):
        self.heap = list(arr)
        for i in range(len(self.heap) // 2 - 1, -1, -1):
            self._heapify_down(i)

    @staticmethod
    def is_max_heap(arr):
        return all(arr[(i - 1) // 2] >= arr[i] for i in range(1, len(arr)))
```

Listing 16.3 (Continued) *p1601_max_heap.py: Problem 16.1 in Python.*

```
    def clear(self):
        self.heap.clear()


def yes_no(b):
    return "Yes" if b else "No"


def main():
    h = MaxHeap()
    for x in (3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5):
        h.insert(x)
    print("Max-Heap array: ", end="")
    h.print_heap()
    print("The max element is", h.get_max())
    h.delete_root()
    print("After deleting the root, heap array: ", end="")
    h.print_heap()
    h.heapify_array([10, 20, 15, 30, 40])
    print("Heapified array: ", end="")
    h.print_heap()
    print("Heap sorted array:", " ".join(map(str, h.heap_sort())))

    print("\n=== The five new operations ===\n")
    print("Clearing the existing heap ...")
    h.clear()
    print("Heap after clearing: ", end="")
    h.print_heap()
    for x in (3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5):
        h.insert(x)
    print("Max-Heap array: ", end="")
    h.print_heap()

    print("\n1. Extracted Max:", h.extract_max())
    print("Max-Heap array: ", end="")
    h.print_heap()

    h.increase_key(2, 8)
    print("\n2. Heap after increasing key at index 2 to 8: ", end="")
    h.print_heap()

    print("\n3. Minimum value in the heap:", h.get_min(), end="\n\n")
```

Listing 16.3 (Continued) *p1601_max_heap.py: Problem 16.1 in Python.*

```
    a1 = [10, 7, 8, 5, 2, 3, 1]
    print("4a. BuildMaxHeap()\nArray: {10, 7, 8, 5, 2, 3, 1}")
    h.clear()
    h.build_max_heap(a1)
    print("Max-Heap built from unsorted array: ", end="")
    h.print_heap()
    print("\n5a. IsMaxHeap()")
    print("Is the array {10, 7, 8, 5, 2, 3, 1} a Max-Heap?",
          yes_no(MaxHeap.is_max_heap(a1)))

    a2 = [10, 7, 6, 5, 1, 8, 2]
    print("\n4b. BuildMaxHeap()\nArray: {10, 7, 6, 5, 1, 8, 2}")
    h.clear()
    h.build_max_heap(a2)
    print("Max-Heap built from unsorted array: ", end="")
    h.print_heap()
    print("\n5b. IsMaxHeap()")
    print("Is the array {10, 7, 6, 5, 1, 8, 2} a Max-Heap?",
          yes_no(MaxHeap.is_max_heap(a2)))
    print("Is the heap built from it a Max-Heap?",
          yes_no(MaxHeap.is_max_heap(h.heap)))

    print("\n=== Edge cases ===")
    print(f"isMaxHeap({{}}) -> {yes_no(MaxHeap.is_max_heap([]))}, "
          f"isMaxHeap({{7}}) -> {yes_no(MaxHeap.is_max_heap([7]))}")
    try:
        h.increase_key(0, 1)
    except ValueError as e:
        print("caught:", e)
    try:
        h.increase_key(7, 99)
    except IndexError as e:
        print("caught:", e)
    h.clear()
    try:
        h.extract_max()
    except IndexError as e:
        print("caught:", e)


if __name__ == "__main__":
    main()
```

16.3.6 A tempting wrong solution

Two tempting shortcuts are worth running, because one of them *passes the given driver*. The first is to think that the smallest element of a max-heap is at the end of the array; the second is to write the loop bound of `isMaxHeap` with the familiar `arr.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
```

The program `wrong/p1601_wrong.cpp` calls both, first on the heap of the driver, then on a three-element heap, then on a one-element array, compiled with the sanitizers:

```
getMinWrong(sample) = 1
getMinWrong({5, 3, 4}) = 4
```

```
==ERROR: AddressSanitizer: heap-buffer-overflow ...
READ of size 4 ... in isMaxHeapWrong ... p1601_wrong.cpp:12
0x... is located 0 bytes after 4-byte region
SUMMARY: AddressSanitizer: heap-buffer-overflow ... p1601_wrong.cpp:12 in isMaxHeapWrong
```

On the driver's heap `getMinWrong` answers 1, which is right, by luck: the last element happens to be one of the two 1s. Insert 5, 3, 4 instead, and the heap is `[5, 3, 4]`; the last element is 4 but the minimum is 3. The last element is always a leaf, but, by Equation (16.3), it is only one of about $n/2$ leaves. The second mistake is worse. `arr.size()` is an unsigned `size_t`, so for an array of one element `arr.size() - 2` is not -1 but the largest `size_t`, about 1.8×10^{19}; the loop runs, reads `arr[1]`, which does not exist, and AddressSanitizer stops the program with the report above (the report has been shortened; the full one names every stack frame). Without the sanitizer the program reads whatever byte happens to follow the array and may print either answer. Our `isMaxHeap` avoids both problems by looping over children, `for (size_t i = 1; i < arr.size(); i++)`, whose bound can never underflow.

16.3.7 Check yourself

> **CHECK YOURSELF.** Start from the heap `[8, 5, 6, 4, 5, 3, 2, 1, 3, 1]` (the heap after step 2). Call `extractMax()` twice. What does each call return, and what is the array after each? Then call `getMin()`: which indices does it look at?
>
> *Answer.* First call: returns 8; the last element 1 moves to the root and sinks past 6 (the larger child) and then past 3 (at index 5, larger than 2): `[6, 5, 3, 4, 5, 1, 2, 1, 3]`. Second call: returns 6; the last element 3 moves to the root, sinks past 5 (index 1, larger than 3), then past 5 (index 4, larger than 4): `[5, 5, 3, 4, 3, 1, 2, 1]`. Now $n = 8$, so by Equation (16.3) `getMin` looks at indices 4 to 7, holding 3, 1, 2, 1, and returns 1.

16.4 Problem 16.2 — Putting Tickets in Travel Order

16.4.1 The problem

> **PROBLEM.** Given a list of tickets, find the itinerary in travel order, using hashing. (1) Create a hash map `dataset` of the tickets: every entry has the form *from* → *to*. (2) Find the starting point of the itinerary: (a) create a reverse hash map `reverseMap` whose entries have the form *to* → *from*; (b) traverse `dataset`; for every key, check whether it exists in `reverseMap`; the key that cannot be found is the starting point. (3) From the starting point, traverse `dataset` and print the itinerary. Input: Cianjur → Bandung, Badung → Denpasar, Gianyar → Cianjur, Denpasar → Gianyar. Output: `Badung -> Denpasar, Denpasar -> Gianyar, Gianyar -> Cianjur, Cianjur -> Bandung`.

(Badung and Bandung are two different places: Badung is a regency in Bali, next to Denpasar and Gianyar, and Bandung is the capital of West Java, next to Cianjur. The one-letter difference is part of what makes the sample a good test.)

16.4.2 In our own words

A traveler has a pile of one-way tickets that were shuffled. Each ticket leaves one city and arrives at another. The tickets form a single trip: every city is left at most once, and the trip never comes back to a city it has already visited. We must print the tickets in the order they are used.

What does the first ticket have that the others do not? Its departure city is *never an arrival city*. Every other ticket leaves a city that the previous ticket arrived at. So the start is the only departure city that does not appear on the arrival side, and a hash set or hash map of the arrival cities answers "is this city ever arrived at?" in constant time. Once we have the start, the next ticket is the one that leaves the city where we are, and a hash map from departure city to arrival

city answers "where does the ticket that leaves this city go?" in constant time. Walking from the start, one lookup per ticket, prints the trip.

> **DEFINITION.** A pile of n tickets *forms one chain* when the n departure cities are all different and exactly one of them, the *start*, is not an arrival city, and when following the tickets from the start uses all n of them. The chain is a path in the directed graph whose vertices are the cities and whose arcs are the tickets (Chapter 13).

The definition tells us what can go wrong, and Step 8 of the method asks us to decide what to do about it: two tickets leaving the same city (then `dataset` cannot even be built, since a map keeps one value per key), no start at all (the tickets form a loop), several starts (two separate trips), or one start but a walk that stops before using every ticket (a chain plus a separate loop). Our program reports each of these instead of printing a partial trip.

16.4.3 By hand, step by step

Figure 16.6 follows the three steps of the problem on the sample.

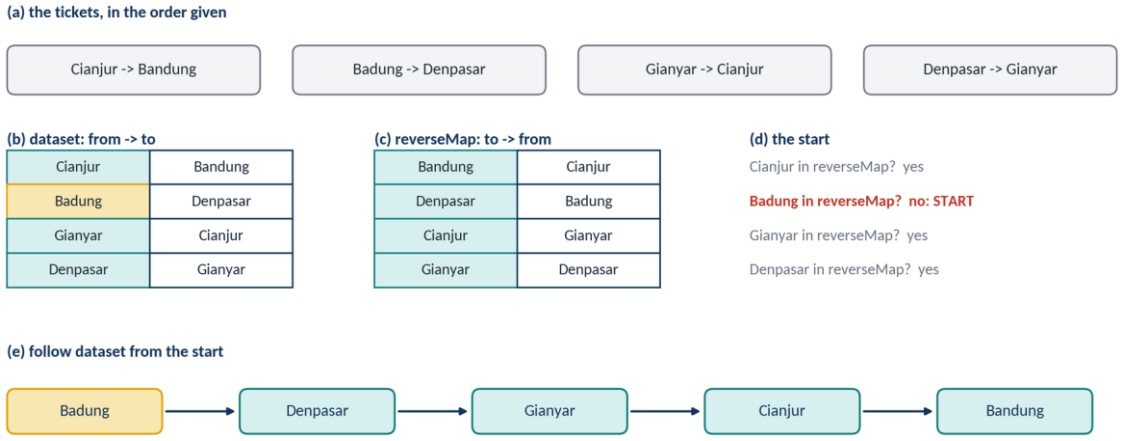


Figure 16.6 Problem 16.2 by hand: the tickets as given (a), the map from departure to arrival (b) and its reverse (c), the search for the one departure city that is never an arrival (d), and the walk from it (e).

1. **Build `dataset`.** Cianjur → Bandung, Badung → Denpasar, Gianyar → Cianjur, Denpasar → Gianyar. Four insertions, each $O(1)$ on average.
2. **Build `reverseMap`.** Bandung → Cianjur, Denpasar → Badung, Cianjur → Gianyar, Gianyar → Denpasar.
3. **Find the start.** Is Cianjur an arrival? Yes (from Gianyar). Badung? *No*. Badung is the start. (We keep checking the other two, Gianyar and Denpasar, both arrivals, so that we can report more than one start if the input is broken.)
4. **Walk.** From Badung, `dataset` gives Denpasar; from Denpasar, Gianyar; from Gianyar, Cianjur; from Cianjur, Bandung; Bandung is not a key of `dataset`, so the trip ends. Four tickets used out of four: the output is the expected line.

Every step is a constant number of hash-map operations per ticket, so the whole algorithm costs $O(n)$ on average for n tickets. The obvious alternative, "for each ticket, search the pile for the ticket that leaves its arrival city", costs $O(n)$ per search and $O(n^2)$ in all: for ten thousand tickets, a hundred million comparisons against about forty thousand hash operations.

16.4.4 Which map is a hash map?

The problem says *hash map*, and in C++ it is tempting to reach for the container whose name is simply `std::map`. That container is not a hash table. The C++ standard requires `std::map` to keep

its keys in sorted order and to find, insert and erase in logarithmic time, and every standard library meets those requirements with a balanced binary search tree, a *red-black tree*: the balanced cousin of the binary search tree of Chapter 11 [9], [5]. The hash table is `std::unordered_map`. A program that solves this problem with `std::map` prints the right itinerary, because both containers can answer "where does this key lead?", but it does not use hashing, it pays $O(\lg n)$ comparisons per lookup instead of $O(1)$ on average, and it visits its keys in alphabetical order, which a hash table never promises to do.

Listing 16.4 makes the difference visible. It puts the four departure cities in both containers, prints the order in which each one visits its keys, asks the hash table which bucket each key landed in, and counts the comparisons that `std::map` makes to find one key.

Listing 16.4 `p1602_map_vs_hash.cpp`: the same four keys in `std::map`, a search tree, and in `std::unordered_map`, a hash table.

```
// p1602_map_vs_hash.cpp - std::map is a search tree; unordered_map a hash table.
#include <iostream>
#include <map>
#include <string>
#include <unordered_map>
using namespace std;

static int comparisons = 0;
struct CountingLess {                       // an ordinary "<" that counts its calls
    bool operator()(const string& a, const string& b) const {
        comparisons++;
        return a < b;
    }
};

int main() {
    const char* from[] = {"Cianjur", "Badung", "Gianyar", "Denpasar"};
    const char* to[] = {"Bandung", "Denpasar", "Cianjur", "Gianyar"};

    map<string, string, CountingLess> tree;     // a red-black tree, ordered by key
    unordered_map<string, string> table;        // a hash table with chained buckets
    for (int i = 0; i < 4; i++) {
        tree[from[i]] = to[i];
        table[from[i]] = to[i];
    }

    cout << "std::map visits the keys in sorted order:\n ";
    for (const auto& kv : tree) cout << " " << kv.first;
    cout << "\nstd::unordered_map visits them bucket by bucket:\n ";
    for (const auto& kv : table) cout << " " << kv.first;
    cout << "\n\nThe hash table has " << table.bucket_count() << " buckets:\n";
    for (int i = 0; i < 4; i++)
        cout << "  " << from[i] << " is in bucket " << table.bucket(from[i])
             << "\n";

    comparisons = 0;
    tree.find("Gianyar");
    cout << "\nmap.find(\"Gianyar\") used " << comparisons << " key comparisons\n";
    cout << "unordered_map.find(\"Gianyar\") hashes the key once and looks in "
         << "bucket " << table.bucket("Gianyar") << "\n";
    return 0;
}
```

```
std::map visits the keys in sorted order:
  Badung Cianjur Denpasar Gianyar
std::unordered_map visits them bucket by bucket:
  Gianyar Badung Denpasar Cianjur

The hash table has 13 buckets:
  Cianjur is in bucket 6
  Badung is in bucket 3
  Gianyar is in bucket 0
  Denpasar is in bucket 6

map.find("Gianyar") used 4 key comparisons
unordered_map.find("Gianyar") hashes the key once and looks in bucket 0
```

Figure 16.7 draws both containers as the output describes them. The tree visits Badung, Cianjur, Denpasar, Gianyar: alphabetical, because an in-order walk of a search tree always is (Chapter 11). The hash table visits them in the order of its buckets, which depends on the hash function of the library and on the number of buckets, 13 here; Cianjur and Denpasar *collide* in bucket 6 and share a chain. A different compiler, or the same one after a few more insertions trigger a rehash, may print a different order. Finding "Gianyar" in the tree took 4 comparisons of whole strings; finding it in the table takes one computation of a hash and a look into one bucket.

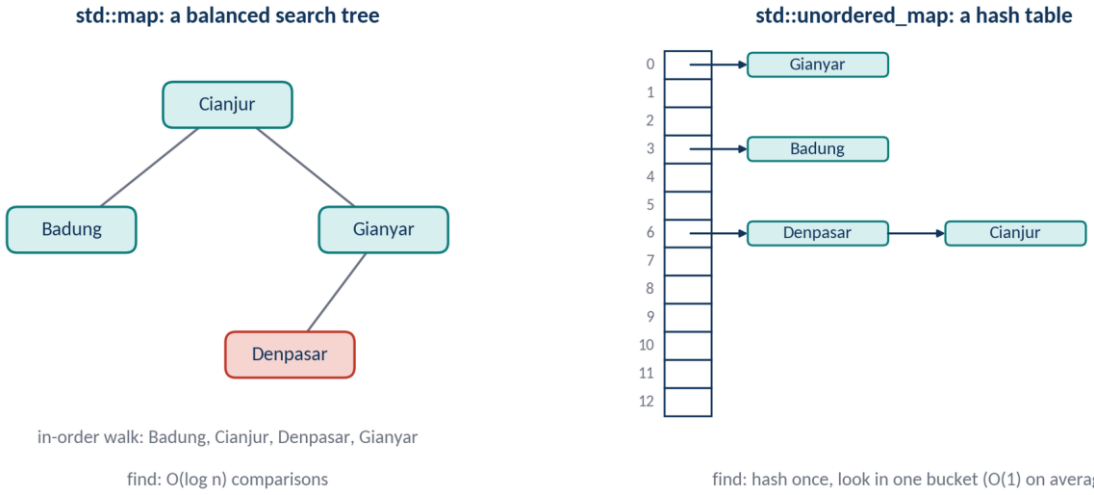


Figure 16.7 The same keys in `std::map`, a red-black tree kept in sorted order (the red node is a red node of the red-black scheme), and in `std::unordered_map`, a hash table with 13 buckets in which Cianjur and Denpasar collide.

Neither container is "better"; they answer different questions. When the task needs keys in order, the smallest key, or every key between two bounds, the tree is the right choice and the hash table the wrong one. When it needs only "is this key here, and what goes with it?", the hash table is the right choice. Section 16.8 measures exactly this trade-off on ten thousand books. In Java the names say it plainly, `HashMap` against `TreeMap`; in Python, `dict` is a hash table and there is no tree map in the standard library at all [6], [7].

16.4.5 The C++ solution

Listing 16.5 follows the three steps with two `unordered_map`s. It checks the input as the definition of Section 16.4.2 requires and runs six piles of tickets: the given one, a round of the Pustaka Ceria book van through East Java, a single ticket, and three broken piles.

Listing 16.5 `p1602_itinerary.cpp`: Problem 16.2 with two hash maps, and a report for every way a pile of tickets can fail to form one chain.

```cpp
// p1602_itinerary.cpp - Problem 16.2: put tickets in travel order, with hashing.
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>
using namespace std;

struct Ticket {
    string from, to;
};

// Fills route with the tickets in travel order. Returns "" on success,
// otherwise a sentence that says why the tickets cannot be put in order.
string findItinerary(const vector<Ticket>& tickets, vector<Ticket>& route) {
    unordered_map<string, string> dataset;      // from -> to  (a hash table)
    unordered_map<string, string> reverseMap;   // to -> from  (a hash table)
    for (const Ticket& t : tickets) {
        if (!dataset.emplace(t.from, t.to).second)
            return "two tickets leave " + t.from;
        reverseMap[t.to] = t.from;
    }

    // The start is the only city that is left but never arrived at.
    vector<string> starts;
    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)";
    if (starts.size() > 1) {
        string s = to_string(starts.size()) + " starting points:";
        for (const string& c : starts) s += " " + c;
        return s;
    }

    // Follow the tickets: from the start, look up where each city leads.
    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";
    return "";
}
```

Listing 16.5 (Continued) *p1602_itinerary.cpp: Problem 16.2 with two hash maps, and a report for every way a pile of tickets can fail to form one chain.*

```
void solve(const string& name, const vector<Ticket>& tickets) {
    cout << "== " << name << "\nInput:\n";
    for (size_t i = 0; i < tickets.size(); i++)
        cout << (i ? "; " : "") << tickets[i].from << " -> " << tickets[i].to;
    vector<Ticket> route;
    string error = findItinerary(tickets, route);
    if (!error.empty()) {
        cout << "\nInvalid input: " << error << "\n";
        return;
    }
    cout << "\nStart: " << route[0].from << "\nOutput:\n";
    for (size_t i = 0; i < route.size(); i++)
        cout << (i ? ", " : "") << route[i].from << " -> " << route[i].to;
    cout << "\n";
}

int main() {
    solve("The given tickets", {{"Cianjur", "Bandung"}, {"Badung", "Denpasar"},
                                {"Gianyar", "Cianjur"}, {"Denpasar", "Gianyar"}});
    solve("The book van of Pustaka Ceria",
          {{"Malang", "Batu"}, {"Surabaya", "Sidoarjo"},
           {"Pasuruan", "Malang"}, {"Sidoarjo", "Pasuruan"}});
    solve("A single ticket", {{"Kediri", "Blitar"}});
    solve("A loop", {{"Bandung", "Jakarta"}, {"Jakarta", "Bandung"}});
    solve("Two separate trips", {{"Solo", "Yogyakarta"}, {"Semarang", "Kudus"}});
    solve("A chain plus a loop",
          {{"Bogor", "Depok"}, {"Bekasi", "Tangerang"}, {"Tangerang", "Bekasi"}});
    return 0;
}
```

```
== The given tickets
Input:
Cianjur -> Bandung; Badung -> Denpasar; Gianyar -> Cianjur; Denpasar -> Gianyar
Start: Badung
Output:
Badung -> Denpasar, Denpasar -> Gianyar, Gianyar -> Cianjur, Cianjur -> Bandung
== The book van of Pustaka Ceria
Input:
Malang -> Batu; Surabaya -> Sidoarjo; Pasuruan -> Malang; Sidoarjo -> Pasuruan
Start: Surabaya
Output:
Surabaya -> Sidoarjo, Sidoarjo -> Pasuruan, Pasuruan -> Malang, Malang -> Batu
== A single ticket
Input:
Kediri -> Blitar
Start: Kediri
Output:
Kediri -> Blitar
== A loop
Input:
Bandung -> Jakarta; Jakarta -> Bandung
Invalid input: no starting point (the tickets form a loop)
== Two separate trips
Input:
Solo -> Yogyakarta; Semarang -> Kudus
Invalid input: 2 starting points: Solo Semarang
== A chain plus a loop
Input:
Bogor -> Depok; Bekasi -> Tangerang; Tangerang -> Bekasi
Invalid input: the tickets do not form one chain
```

Line by line. `dataset.emplace(t.from, t.to)` inserts only if the key is new and returns a pair whose `.second` says whether it did; a `false` there means two tickets leave the same city, and the

function reports it at once. The start is found by looping over the tickets *in input order* and asking `reverseMap.find(t.from)`. We loop over the input rather than over `dataset` for a reason that matters beyond this problem: the order in which a hash table visits its keys is unspecified, and a program whose output depends on it may print different lines on different compilers. By looping over the input we make every line of output deterministic, which is also why the Java and Python versions print exactly the same lines. The walk looks up the current city with `dataset.find` and stops when the city has no outgoing ticket, or when it has used as many tickets as there are, which protects it against a loop hidden behind the start (the last pile). The closing comparison `route.size() != tickets.size()` catches a chain that ends early.

Cost. Building the two maps is n insertions each, finding the start is n lookups, and the walk is at most n lookups: $O(n)$ on average. The printing in `solve` is $O(n)$ as well. The memory is two maps of n entries each.

16.4.6 The Java and Python solutions

Listing 16.6 and **Listing 16.7** print the same six results. Java's `putIfAbsent` returns the old value, or `null` if there was none, which is how it reports a second ticket from the same city; Python tests `if frm in dataset` first.

Listing 16.6 `P1602Itinerary.java`: Problem 16.2 in Java with two `HashMap`s.

```java
// P1602Itinerary.java - Problem 16.2: put tickets in travel order, with hashing.
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class P1602Itinerary {

    record Ticket(String from, String to) {}

    // Fills route with the tickets in travel order. Returns "" on success,
    // otherwise a sentence that says why the tickets cannot be put in order.
    static String findItinerary(List<Ticket> tickets, List<Ticket> route) {
        HashMap<String, String> dataset = new HashMap<>();     // from -> to
        HashMap<String, String> reverseMap = new HashMap<>();  // to -> from
        for (Ticket t : tickets) {
            if (dataset.putIfAbsent(t.from(), t.to()) != null)
                return "two tickets leave " + t.from();
            reverseMap.put(t.to(), t.from());
        }

        // The start is the only city that is left but never arrived at.
        List<String> starts = new ArrayList<>();
        for (Ticket t : tickets)
            if (!reverseMap.containsKey(t.from())) starts.add(t.from());
        if (starts.isEmpty()) return "no starting point (the tickets form a loop)";
        if (starts.size() > 1)
            return starts.size() + " starting points: " + String.join(" ", starts);

        // Follow the tickets: from the start, look up where each city leads.
        String city = starts.get(0);
        while (dataset.containsKey(city) && route.size() < tickets.size()) {
            String next = dataset.get(city);
            route.add(new Ticket(city, next));
            city = next;
        }
        if (route.size() != tickets.size())
            return "the tickets do not form one chain";
        return "";
    }
```

Listing 16.6 (Continued) *P1602Itinerary.java: Problem 16.2 in Java with two HashMaps.*

```
    static void solve(String name, List<Ticket> tickets) {
        StringBuilder sb = new StringBuilder("== " + name + "\nInput:\n");
        for (int i = 0; i < tickets.size(); i++)
            sb.append(i > 0 ? "; " : "").append(tickets.get(i).from())
              .append(" -> ").append(tickets.get(i).to());
        List<Ticket> route = new ArrayList<>();
        String error = findItinerary(tickets, route);
        if (!error.isEmpty()) {
            System.out.println(sb + "\nInvalid input: " + error);
            return;
        }
        sb.append("\nStart: ").append(route.get(0).from()).append("\nOutput:\n");
        for (int i = 0; i < route.size(); i++)
            sb.append(i > 0 ? ", " : "").append(route.get(i).from())
              .append(" -> ").append(route.get(i).to());
        System.out.println(sb);
    }

    static Ticket t(String from, String to) {
        return new Ticket(from, to);
    }

    public static void main(String[] args) {
        solve("The given tickets",
                List.of(t("Cianjur", "Bandung"), t("Badung", "Denpasar"),
                        t("Gianyar", "Cianjur"), t("Denpasar", "Gianyar")));
        solve("The book van of Pustaka Ceria",
                List.of(t("Malang", "Batu"), t("Surabaya", "Sidoarjo"),
                        t("Pasuruan", "Malang"), t("Sidoarjo", "Pasuruan")));
        solve("A single ticket", List.of(t("Kediri", "Blitar")));
        solve("A loop", List.of(t("Bandung", "Jakarta"), t("Jakarta", "Bandung")));
        solve("Two separate trips",
                List.of(t("Solo", "Yogyakarta"), t("Semarang", "Kudus")));
        solve("A chain plus a loop", List.of(t("Bogor", "Depok"),
                t("Bekasi", "Tangerang"), t("Tangerang", "Bekasi")));
    }
}
```

Listing 16.7 `p1602_itinerary.py`: Problem 16.2 in Python with two `dict`s.

```
# p1602_itinerary.py - Problem 16.2: put tickets in travel order, with hashing.


def find_itinerary(tickets):
    """Returns (route, "") on success, or ([], reason) if there is no chain."""
    dataset = {}                    # from -> to   (a dict is a hash table)
    reverse_map = {}                # to -> from
    for frm, to in tickets:
        if frm in dataset:
            return [], "two tickets leave " + frm
        dataset[frm] = to
        reverse_map[to] = frm

    # The start is the only city that is left but never arrived at.
    starts = [frm for frm, _ in tickets if frm not in reverse_map]
    if not starts:
        return [], "no starting point (the tickets form a loop)"
    if len(starts) > 1:
        return [], f"{len(starts)} starting points: " + " ".join(starts)

    # Follow the tickets: from the start, look up where each city leads.
    route, city = [], starts[0]
    while city in dataset and len(route) < len(tickets):
        route.append((city, dataset[city]))
        city = dataset[city]
    if len(route) != len(tickets):
        return [], "the tickets do not form one chain"
    return route, ""


def solve(name, tickets):
    print(f"== {name}\nInput:")
    print("; ".join(f"{a} -> {b}" for a, b in tickets))
    route, error = find_itinerary(tickets)
    if error:
        print("Invalid input:", error)
        return
    print("Start:", route[0][0])
    print("Output:")
    print(", ".join(f"{a} -> {b}" for a, b in route))
```

Listing 16.7 (Continued) *p1602_itinerary.py: Problem 16.2 in Python with two dicts.*

```
def main():
    solve("The given tickets", [("Cianjur", "Bandung"), ("Badung", "Denpasar"),
                                ("Gianyar", "Cianjur"), ("Denpasar", "Gianyar")])
    solve("The book van of Pustaka Ceria",
          [("Malang", "Batu"), ("Surabaya", "Sidoarjo"),
           ("Pasuruan", "Malang"), ("Sidoarjo", "Pasuruan")])
    solve("A single ticket", [("Kediri", "Blitar")])
    solve("A loop", [("Bandung", "Jakarta"), ("Jakarta", "Bandung")])
    solve("Two separate trips", [("Solo", "Yogyakarta"), ("Semarang", "Kudus")])
    solve("A chain plus a loop",
          [("Bogor", "Depok"), ("Bekasi", "Tangerang"), ("Tangerang", "Bekasi")])


if __name__ == "__main__":
    main()
```

16.4.7 A tempting wrong solution

The tempting shortcut is to skip Step 2 altogether and start from the first ticket in the pile, reasoning that the pile is "a list of tickets" and lists start at the beginning:

```
    string city = tickets[0].first;      // WRONG: "the first ticket is the first leg"
```

Run on the sample (`wrong/p1602_wrong.cpp`), it prints

```
Cianjur -> Bandung
```

and stops. The first ticket in the pile is Cianjur → Bandung, which is the *last* leg of the trip; from Bandung there is no ticket, and three quarters of the trip is lost without a word. The whole point of the problem is that the pile is shuffled, so the position of a ticket in the input says nothing about its position in the trip. A second, subtler slip lives in the printing loop of many solutions: a counter that decides whether to print the separating comma is declared as `int a;` and never set to 0. Its first value is whatever was left in that memory, so the first ticket may or may not get a comma in front of it, and Valgrind reports “Conditional jump or move depends on uninitialised value(s)”. Initialize every counter where you declare it, or, as our solution does, decide on the separator from the loop index.

16.4.8 Check yourself

> **CHECK YOURSELF.** The tickets are Semarang → Solo, Jakarta → Cirebon, Solo → Madiun, Cirebon → Semarang, Madiun → Surabaya. Build `reverseMap`, find the start, and write the itinerary. Then add the ticket Surabaya → Jakarta: what does our program report, and why?
>
> *Answer.* `reverseMap`: Solo → Semarang, Cirebon → Jakarta, Madiun → Solo, Semarang → Cirebon, Surabaya → Madiun. The departure cities are Semarang, Jakarta, Solo, Cirebon, Madiun; only Jakarta is not a key of `reverseMap`, so Jakarta is the start. Itinerary: `Jakarta -> Cirebon, Cirebon -> Semarang, Semarang -> Solo, Solo -> Madiun, Madiun -> Surabaya`. With Surabaya → Jakarta added, Jakarta becomes an arrival too, every departure city is an arrival, and the program reports `no starting point (the tickets form a loop)`. A loop has no first ticket; any city could be the start.

16.5 New Worked Problems on Graphs

The three problems of this section share one map, drawn in **Figure 16.8**: eight branches of the Pustaka Ceria network in Surabaya and the roads that the book van may use. Pusat is the central library; Benowo, a new branch in the far west of the city, is not yet connected to any road the van uses. The map is small enough to trace by hand and has everything the algorithms need to show their character: two cycles, a vertex that cannot be reached, and, in Problem 16.5, a direct road that is *longer* than a detour.

16.5.1 Problem 16.3 — The fewest hops between branches

> **PROBLEM.** The book van moves between neighboring branches along the roads Pusat–Gubeng, Pusat–Darmo, Pusat–Kenjeran, Gubeng–Sukolilo, Gubeng–Rungkut, Darmo–Wonokromo, Wonokromo–Rungkut and Kenjeran–Sukolilo; Benowo has no road yet. Every road counts as one hop. For every branch, find the fewest hops from Pusat and the branch from which the van arrives on such a route, and print the route to Rungkut, to Wonokromo and to Benowo.

In our own words. The vertices are the eight branches, the edges the eight roads, undirected and unweighted. We want the *distance* from Pusat to every vertex, measured in edges, and one shortest route to each. That is exactly what breadth-first search computes (Chapter 13): it discovers the vertices in rings, first those one edge away, then those two edges away, and so on, and the first time it discovers a vertex it has found a shortest route to it.

> **DEFINITION.** The *BFS invariant*: the vertices in the queue are in nondecreasing order of `dist`, and the first and last differ by at most 1. Consequently, when a vertex v is discovered from u, no shorter route to v can be found later, and `dist[v] = dist[u] + 1` is final.

The invariant explains the single most important line of BFS: we set `dist[v]` *when we first see* v, not when v leaves the queue, and we never change it again. A vertex whose `dist` is still -1 has not been seen; any other value means "seen, and final". So the same array serves as the distance table and as the "visited" marks.

By hand, step by step. The adjacency lists, in the order the program builds them (each road is added to both of its ends, in the order the roads are listed), are:

```
Adjacency lists:
  Pusat     : Gubeng Darmo Kenjeran
  Gubeng    : Pusat Sukolilo Rungkut
  Darmo     : Pusat Wonokromo
  Kenjeran  : Pusat Sukolilo
  Rungkut   : Gubeng Wonokromo
  Wonokromo : Darmo Rungkut
  Sukolilo  : Gubeng Kenjeran
  Benowo    :
```

Now follow the queue.

1. `dist[Pusat] = 0`; queue: Pusat.
2. Take Pusat. Its neighbors Gubeng, Darmo, Kenjeran are unseen: each gets distance 1 and parent Pusat, and joins the queue. Queue: Gubeng, Darmo, Kenjeran.
3. Take Gubeng. Pusat is seen; Sukolilo and Rungkut get distance 2 and parent Gubeng. Queue: Darmo, Kenjeran, Sukolilo, Rungkut.
4. Take Darmo. Pusat is seen; Wonokromo gets distance 2, parent Darmo. Queue: Kenjeran, Sukolilo, Rungkut, Wonokromo.
5. Take Kenjeran: Pusat and Sukolilo are already seen. Nothing changes. Notice that Kenjeran–Sukolilo is also a route to Sukolilo of length 2, but Sukolilo already has distance 2 from Gubeng, and the invariant tells us it cannot improve.
6. Take Sukolilo, Rungkut, Wonokromo in turn: all their neighbors are seen. The road Wonokromo–Rungkut is examined twice, once from each end, and changes nothing.
7. The queue is empty. Benowo was never seen: its distance stays -1, "unreachable".

Figure 16.8 draws the result: the dark edges are the *BFS tree*, the edges through which each vertex was first discovered, and the pale edges were examined but not used.

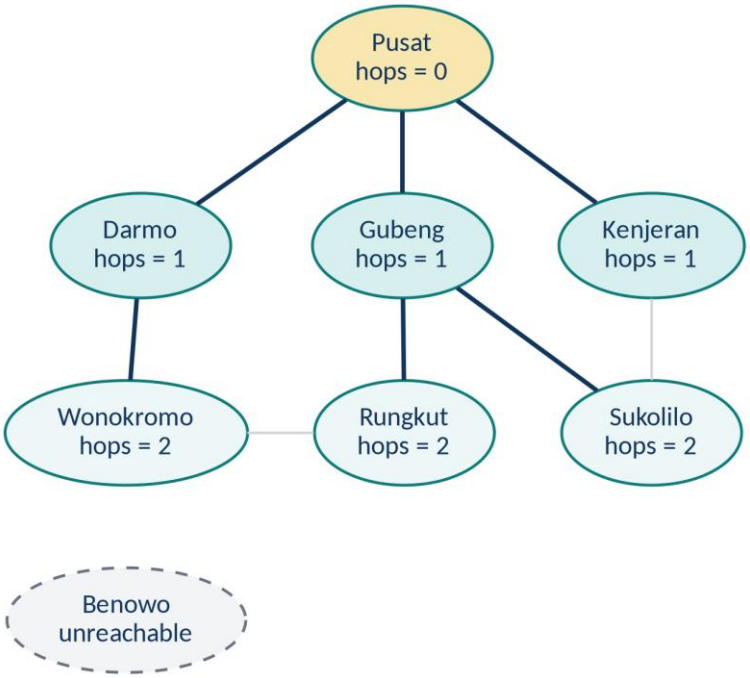


Figure 16.8 The branch map searched breadth-first from Pusat: vertices in rings of equal distance, the BFS tree in dark lines, the two unused roads pale, and Benowo unreachable.

To print the route to Rungkut, start at Rungkut and follow the parents back: Rungkut came from Gubeng, Gubeng from Pusat, Pusat has no parent. Reversed, that is Pusat → Gubeng → Rungkut, two hops.

Cost. Every vertex enters the queue at most once, because it is marked on entry, and when it leaves, its adjacency list is read once. The adjacency lists of an undirected graph hold $2|E|$ entries in all (each edge appears at both ends), so the total work is

$$T_{\text{BFS}} = O(|V| + |E|). \tag{16.5}$$

For our map, Equation (16.5) is $8 + 2 \cdot 8 = 24$ basic steps. An adjacency matrix would force the search to look at all $|V|$ cells of a row for every vertex, $O(|V|^2)$, which is why BFS is written with adjacency lists (Chapter 13).

The C++ solution. Listing 16.8 builds the adjacency lists from the road list and runs BFS from Pusat.

Listing 16.8 `p1603_bfs_hops.cpp`: breadth-first search from Pusat, with the fewest hops, the parent of every branch, and three routes.

```cpp
// p1603_bfs_hops.cpp - Problem 16.3: fewest hops between library branches (BFS).
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <queue>
#include <string>
#include <vector>
using namespace std;

const vector<string> NAME = {"Pusat", "Gubeng", "Darmo", "Kenjeran",
                             "Rungkut", "Wonokromo", "Sukolilo", "Benowo"};

// Breadth-first search from s. dist[v] = fewest edges from s (-1: unreachable),
// parent[v] = the vertex from which v was first reached (-1: none).
void bfs(const vector<vector<int>>& adj, int s, vector<int>& dist,
         vector<int>& parent, vector<int>& order) {
    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);
            }
        }
    }
}

void printRoute(const vector<int>& dist, const vector<int>& parent, int t) {
    cout << "Fewest hops to " << NAME[t] << ": ";
    if (dist[t] == -1) {
        cout << "no route\n";
        return;
    }
    vector<int> path;
    for (int v = t; v != -1; v = parent[v]) path.push_back(v);   // walk back
    reverse(path.begin(), path.end());
    for (size_t i = 0; i < path.size(); i++)
        cout << (i ? " -> " : "") << NAME[path[i]];
```

Listing 16.8 (Continued) *p1603_bfs_hops.cpp: breadth-first search from Pusat, with the fewest hops, the parent of every branch, and three routes.*

```
    cout << " (" << dist[t] << " hops)\n";
}

int main() {
    int n = NAME.size();
    vector<pair<int, int>> roads = {{0, 1}, {0, 2}, {0, 3}, {1, 6},
                                    {1, 4}, {2, 5}, {5, 4}, {3, 6}};
    vector<vector<int>> adj(n);
    for (auto [u, v] : roads) {         // an undirected road is two arcs
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    cout << "Adjacency lists:\n";
    for (int u = 0; u < n; u++) {
        cout << "  " << left << setw(10) << NAME[u] << ":";
        for (int v : adj[u]) cout << " " << NAME[v];
        cout << "\n";
    }

    vector<int> dist, parent, order;
    bfs(adj, 0, dist, parent, order);
    cout << "\nOrder in which BFS takes the branches out of the queue:\n ";
    for (int u : order) cout << " " << NAME[u] << "(" << dist[u] << ")";
    cout << "\n\n" << left << setw(12) << "Branch" << setw(6) << "Hops"
         << "Reached from\n";
    for (int v = 0; v < n; v++) {
        cout << setw(12) << NAME[v];
        if (dist[v] == -1) {
            cout << "unreachable\n";
            continue;
        }
        cout << setw(6) << dist[v] << (parent[v] == -1 ? "-" : NAME[parent[v]])
             << "\n";
    }
    cout << "\n";
    printRoute(dist, parent, 4);
    printRoute(dist, parent, 5);
    printRoute(dist, parent, 7);
    return 0;
}
```

```
Adjacency lists:
  Pusat     : Gubeng Darmo Kenjeran
  Gubeng    : Pusat Sukolilo Rungkut
  Darmo     : Pusat Wonokromo
  Kenjeran  : Pusat Sukolilo
  Rungkut   : Gubeng Wonokromo
  Wonokromo : Darmo Rungkut
  Sukolilo  : Gubeng Kenjeran
  Benowo    :

Order in which BFS takes the branches out of the queue:
  Pusat(0) Gubeng(1) Darmo(1) Kenjeran(1) Sukolilo(2) Rungkut(2) Wonokromo(2)

Branch      Hops  Reached from
Pusat       0     -
Gubeng      1     Pusat
Darmo       1     Pusat
Kenjeran    1     Pusat
Rungkut     2     Gubeng
Wonokromo   2     Darmo
Sukolilo    2     Gubeng
Benowo      unreachable

Fewest hops to Rungkut: Pusat -> Gubeng -> Rungkut (2 hops)
Fewest hops to Wonokromo: Pusat -> Darmo -> Wonokromo (2 hops)
Fewest hops to Benowo: no route
```

Line by line. `for (auto [u, v] : roads)` uses C++17 *structured bindings* to name the two ends of each road; the two `push_back`s add the road in both directions. In `bfs`, `dist.assign(n, -1)` both sizes and fills the arrays; `dist[s] = 0` is set before `s` enters the queue, the same rule as for every other vertex. The inner `if (dist[v] == -1)` is the whole algorithm: an unseen neighbor gets its final distance and its parent and joins the queue. `printRoute` walks the parents back into a vector and reverses it, because parents lead from the target to the source and we want to print from the source.

The Java and Python solutions. **Listing 16.9** and **Listing 16.10** print the same lines. Java uses `ArrayDeque` as the queue (`add` at the back, `poll` from the front); Python uses `collections.deque` (`append`, `popleft`), never a plain `list` with `pop(0)`, which shifts every remaining element and would turn the $O(1)$ dequeue into $O(n)$ [6], [7].

Listing 16.9 `P1603BfsHops.java`: Problem 16.3 in Java.

```java
// P1603BfsHops.java - Problem 16.3: fewest hops between library branches (BFS).
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class P1603BfsHops {

    static final String[] NAME = {"Pusat", "Gubeng", "Darmo", "Kenjeran",
                                  "Rungkut", "Wonokromo", "Sukolilo", "Benowo"};

    // Breadth-first search from s. dist[v] = fewest edges from s (-1: unreachable),
    // parent[v] = the vertex from which v was first reached (-1: none).
    static List<Integer> bfs(List<List<Integer>> adj, int s, int[] dist,
                             int[] parent) {
        Arrays.fill(dist, -1);
        Arrays.fill(parent, -1);
        List<Integer> order = new ArrayList<>();
        ArrayDeque<Integer> q = new ArrayDeque<>();
        dist[s] = 0;                          // mark s BEFORE it waits in the queue
        q.add(s);
        while (!q.isEmpty()) {
            int u = q.poll();
            order.add(u);
            for (int v : adj.get(u)) {
                if (dist[v] == -1) {          // first sight of v: distance is final
                    dist[v] = dist[u] + 1;
                    parent[v] = u;
                    q.add(v);
                }
            }
        }
        return order;
    }
```

Listing 16.9 (Continued) *P1603BfsHops.java: Problem 16.3 in Java.*

```
    static void printRoute(int[] dist, int[] parent, int t) {
        System.out.print("Fewest hops to " + NAME[t] + ": ");
        if (dist[t] == -1) {
            System.out.println("no route");
            return;
        }
        List<String> path = new ArrayList<>();
        for (int v = t; v != -1; v = parent[v]) path.add(NAME[v]);   // walk back
        Collections.reverse(path);
        System.out.println(String.join(" -> ", path) + " (" + dist[t] + " hops)");
    }

    public static void main(String[] args) {
        int n = NAME.length;
        int[][] roads = {{0, 1}, {0, 2}, {0, 3}, {1, 6},
                         {1, 4}, {2, 5}, {5, 4}, {3, 6}};
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < n; i++) adj.add(new ArrayList<>());
        for (int[] r : roads) {             // an undirected road is two arcs
            adj.get(r[0]).add(r[1]);
            adj.get(r[1]).add(r[0]);
        }
        System.out.println("Adjacency lists:");
        for (int u = 0; u < n; u++) {
            StringBuilder sb = new StringBuilder(
                    String.format("  %-10s:", NAME[u]));
            for (int v : adj.get(u)) sb.append(" ").append(NAME[v]);
            System.out.println(sb);
        }

        int[] dist = new int[n], parent = new int[n];
        List<Integer> order = bfs(adj, 0, dist, parent);
        System.out.println(
                "\nOrder in which BFS takes the branches out of the queue:");
        StringBuilder sb = new StringBuilder(" ");
        for (int u : order)
            sb.append(" ").append(NAME[u]).append("(").append(dist[u]).append(")");
        System.out.println(sb);
        System.out.printf("%n%-12s%-6s%s%n", "Branch", "Hops", "Reached from");
        for (int v = 0; v < n; v++) {
            if (dist[v] == -1) {
                System.out.printf("%-12sunreachable%n", NAME[v]);
                continue;
            }
            System.out.printf("%-12s%-6d%s%n", NAME[v], dist[v],
                    parent[v] == -1 ? "-" : NAME[parent[v]]);
```

Listing 16.9 (Continued) *P1603BfsHops.java: Problem 16.3 in Java.*

```
        }
        System.out.println();
        printRoute(dist, parent, 4);
        printRoute(dist, parent, 5);
        printRoute(dist, parent, 7);
    }
}
```

Listing 16.10 `p1603_bfs_hops.py`: Problem 16.3 in Python.

```
# p1603_bfs_hops.py - Problem 16.3: fewest hops between library branches (BFS).
from collections import deque

NAME = ["Pusat", "Gubeng", "Darmo", "Kenjeran",
        "Rungkut", "Wonokromo", "Sukolilo", "Benowo"]


def bfs(adj, s):
    """dist[v] = fewest edges from s (-1: unreachable); parent[v] = where v was
    first reached from (-1: none); order = the order vertices leave the queue."""
    dist = [-1] * len(adj)
    parent = [-1] * len(adj)
    order = []
    dist[s] = 0                          # mark s BEFORE it waits in the queue
    q = deque([s])
    while q:
        u = q.popleft()
        order.append(u)
        for v in adj[u]:
            if dist[v] == -1:            # first time we see v: its distance is final
                dist[v] = dist[u] + 1
                parent[v] = u
                q.append(v)
    return dist, parent, order


def print_route(dist, parent, t):
    print(f"Fewest hops to {NAME[t]}: ", end="")
    if dist[t] == -1:
        print("no route")
        return
    path, v = [], t
    while v != -1:                       # walk back along the parents
        path.append(NAME[v])
        v = parent[v]
    print(" -> ".join(reversed(path)), f"({dist[t]} hops)")
```

Listing 16.10 (Continued) *p1603_bfs_hops.py: Problem 16.3 in Python.*

```
def main():
    n = len(NAME)
    roads = [(0, 1), (0, 2), (0, 3), (1, 6), (1, 4), (2, 5), (5, 4), (3, 6)]
    adj = [[] for _ in range(n)]
    for u, v in roads:                  # an undirected road is two arcs
        adj[u].append(v)
        adj[v].append(u)
    print("Adjacency lists:")
    for u in range(n):
        print(f"  {NAME[u]:<10}:" + "".join(" " + NAME[v] for v in adj[u]))

    dist, parent, order = bfs(adj, 0)
    print("\nOrder in which BFS takes the branches out of the queue:")
    print(" " + "".join(f" {NAME[u]}({dist[u]})" for u in order))
    print(f"\n{'Branch':<12}{'Hops':<6}Reached from")
    for v in range(n):
        if dist[v] == -1:
            print(f"{NAME[v]:<12}unreachable")
            continue
        came_from = "-" if parent[v] == -1 else NAME[parent[v]]
        print(f"{NAME[v]:<12}{dist[v]:<6}{came_from}")
    print()
    print_route(dist, parent, 4)
    print_route(dist, parent, 5)
    print_route(dist, parent, 7)


if __name__ == "__main__":
    main()
```

A tempting wrong solution. Depth-first search is shorter to write, recursive and elegant, and it also visits every reachable vertex. So why not let the depth at which DFS reaches a vertex be its 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);
  }
```

`wrong/p1603_wrong.cpp` prints

```
Pusat 0
Gubeng 1
Darmo 4
Kenjeran 3
Rungkut 2
Wonokromo 3
Sukolilo 2
```

DFS from Pusat goes to Gubeng (1), then, before looking at Pusat's other neighbors, dives on to Sukolilo (2) and Kenjeran (3), backs up, goes to Rungkut (2), Wonokromo (3) and finally Darmo (4). Darmo is one road away from Pusat, but DFS happened to reach it last, along the longest way round. DFS answers "can I get there?" correctly, and it is the right tool for cycles, components and topological order; it does not answer "how few hops?". Only the ring-by-ring order of a *queue* guarantees that the first route found is a shortest one.

> **CHECK YOURSELF.** The city opens a road Darmo–Benowo, appended after the eight roads. Give the new BFS order, the distance and parent of Benowo, and the route to it.
>
> *Answer.* Darmo's adjacency list becomes Pusat, Wonokromo, Benowo, so Benowo is discovered from Darmo right after Wonokromo, with distance 2. The BFS order is Pusat, Gubeng, Darmo,

> Kenjeran, Sukolilo, Rungkut, Wonokromo, Benowo, and the route is Pusat → Darmo → Benowo, 2 hops. Nothing else changes.

16.5.2 Problem 16.4 — An order in which to read the topics

> **PROBLEM.** The topics of this book depend on each other: arrays come before searching, sorting, stacks and pointers; pointers and stacks come before linked lists; linked lists before recursion, trees, graphs and hashing; recursion and searching before trees; trees and sorting before heaps; stacks before graphs; searching before hashing. Print an order in which the eleven topics can be read so that every topic comes after all the topics it needs. Then add the requirement "graphs before pointers" and report what happens.

In our own words. Each requirement "read u before v" is an *arc* $u \to v$ of a directed graph. An order of the vertices in which every arc points forward is a *topological order* (Chapter 13). It exists exactly when the graph has no directed cycle, and such a graph is a *directed acyclic graph*, a DAG. We use Kahn's algorithm [10]: count, for every topic, how many prerequisites it still waits for (its *in-degree*); put every topic with count 0 in a queue; repeatedly take a topic from the queue, append it to the order, and subtract 1 from the count of every topic that needs it; a topic whose count reaches 0 joins the queue.

> **DEFINITION.** The *Kahn invariant*: every topic already in the order has all its prerequisites earlier in the order, and `indeg[v]` equals the number of prerequisites of v not yet in the order. If the queue empties while some topics are still missing, each missing topic waits for another missing topic, so following prerequisites backward among them must eventually repeat a topic: there is a directed cycle among them or in front of them, and no order exists.

The last sentence of the definition is the reason for the one line that careless solutions leave out: *after* the loop, compare the length of the order with the number of topics.

By hand, step by step. **Figure 16.9** shows the eleven topics and their sixteen arcs, each topic labeled with its position in the order that Kahn's algorithm produces. The in-degrees at the start are: Arrays 0; Searching, Sorting, Stacks, Pointers, Recursion 1; Lists, Graphs, Heaps, Hashing 2; Trees 3.

1. Queue: Arrays. Take it. Searching, Sorting, Stacks, Pointers drop to 0. Queue: Searching, Sorting, Stacks, Pointers.
2. Take Searching: Trees 2, Hashing 1. Take Sorting: Heaps 1. Take Stacks: Lists 1, Graphs 1. Take Pointers: Lists 0, joins. Queue: Lists.
3. Take Lists: Recursion 0, Trees 1, Graphs 0, Hashing 0; three join. Queue: Recursion, Graphs, Hashing.
4. Take Recursion: Trees 0, joins. Take Graphs, Hashing. Take Trees: Heaps 0, joins. Take Heaps. The queue is empty and all eleven topics are placed.

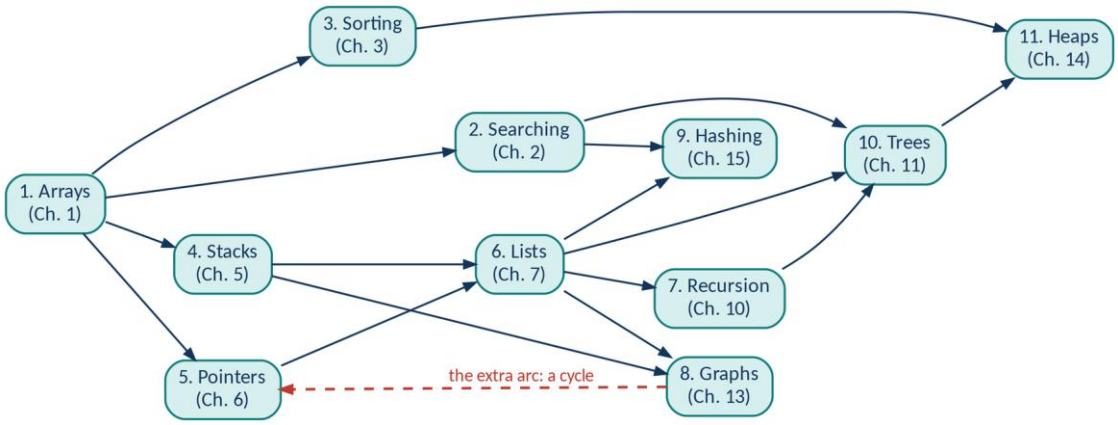


Figure 16.9 The eleven topics and the sixteen "read before" arcs, each topic numbered by its place in the order Kahn's algorithm finds; the dashed red arc "graphs before pointers" closes the cycle Pointers, Lists, Graphs, Pointers.

The order puts Graphs and Hashing before Trees, which is not the order of the chapters of this book. It is still a valid order for these sixteen arcs: nothing in the list says that trees must precede graphs. A DAG usually has many topological orders, and Kahn's algorithm returns the one that its queue discipline produces. Now add the dashed arc Graphs → Pointers. Pointers now waits for two topics, Arrays and Graphs; Graphs waits for Lists, which waits for Pointers. After Arrays, Searching, Sorting and Stacks, the queue is empty, and seven topics are left: the three on the cycle and the four (Recursion, Trees, Heaps, Hashing) that depend on it.

Cost. Counting in-degrees reads every arc once; every topic enters and leaves the queue at most once; every arc is followed once when its tail leaves the queue. The total is $O(|V| + |E|)$, as for BFS in Equation (16.5).

The C++ solution. Listing 16.11 builds the graph, prints each topic with the topics that need it, and reports a reading order or the topics that cannot be placed.

Listing 16.11 `p1604_reading_order.cpp`: Kahn's algorithm on the eleven topics of the book, with and without the arc that creates a cycle.

```cpp
// p1604_reading_order.cpp - Problem 16.4: an order to read the topics (Kahn).
#include <iomanip>
#include <iostream>
#include <queue>
#include <string>
#include <vector>
using namespace std;

const vector<string> TOPIC = {"Arrays", "Searching", "Sorting", "Stacks",
                              "Pointers", "Lists", "Recursion", "Trees",
                              "Graphs", "Heaps", "Hashing"};
const vector<int> CHAPTER = {1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15};

// Kahn's algorithm. Returns the topics in an order in which every arc u -> v
// has u before v. If some topics are missing from the result, they lie on
// or behind a cycle and no valid order exists.
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;
}

void report(const vector<vector<int>>& adj) {
    vector<int> order = kahn(adj);
    int n = adj.size();
    if ((int)order.size() == n) {
        cout << "A reading order:\n";
        for (int i = 0; i < n; i++)
            cout << setw(4) << i + 1 << ". " << left << setw(10) << TOPIC[order[i]]
                 << right << " (Chapter " << CHAPTER[order[i]] << ")\n";
        return;
    }
    vector<bool> placed(n, false);
```

Listing 16.11 (Continued) *p1604_reading_order.cpp: Kahn's algorithm on the eleven topics of the book, with and without the arc that creates a cycle.*

```
    for (int u : order) placed[u] = true;
    cout << "No reading order exists: " << n - order.size()
         << " topics wait for each other in a cycle or behind one:\n ";
    for (int u = 0; u < n; u++)
        if (!placed[u]) cout << " " << TOPIC[u];
    cout << "\n";
}

int main() {
    int n = TOPIC.size();
    vector<pair<int, int>> needs = {           // {u, v}: read u before v
        {0, 1}, {0, 2}, {0, 3}, {0, 4}, {4, 5}, {3, 5}, {5, 6}, {5, 7}, {6, 7},
        {1, 7}, {3, 8}, {5, 8}, {7, 9}, {2, 9}, {5, 10}, {1, 10}};
    vector<vector<int>> adj(n);
    for (auto [u, v] : needs) adj[u].push_back(v);

    cout << "Each topic, and the topics that need it:\n";
    for (int u = 0; u < n; u++) {
        cout << "  " << left << setw(10) << TOPIC[u] << right << "->";
        for (int v : adj[u]) cout << " " << TOPIC[v];
        cout << "\n";
    }
    cout << "\n";
    report(adj);

    cout << "\nNow suppose someone insists that Graphs must come before "
            "Pointers.\n";
    adj[8].push_back(4);
    report(adj);
    return 0;
}
```

```
Each topic, and the topics that need it:
  Arrays    -> Searching Sorting Stacks Pointers
  Searching -> Trees Hashing
  Sorting   -> Heaps
  Stacks    -> Lists Graphs
  Pointers  -> Lists
  Lists     -> Recursion Trees Graphs Hashing
  Recursion -> Trees
  Trees     -> Heaps
  Graphs    ->
  Heaps     ->
  Hashing   ->

A reading order:
   1. Arrays     (Chapter 1)
   2. Searching  (Chapter 2)
   3. Sorting    (Chapter 3)
   4. Stacks     (Chapter 5)
   5. Pointers   (Chapter 6)
   6. Lists      (Chapter 7)
   7. Recursion  (Chapter 10)
   8. Graphs     (Chapter 13)
   9. Hashing    (Chapter 15)
  10. Trees      (Chapter 11)
  11. Heaps      (Chapter 14)

Now suppose someone insists that Graphs must come before Pointers.
No reading order exists: 7 topics wait for each other in a cycle or behind one:
  Pointers Lists Recursion Trees Graphs Heaps Hashing
```

Line by line. `kahn` returns the order and nothing else; it is `report` that decides what the order means, by comparing `order.size()` with `n`. The expression `--indeg[v] == 0` decrements and tests in one step: it is true exactly when the topic just lost its last missing prerequisite. The topics that never reach the queue are found by marking the placed ones and listing the rest.

The Java and Python solutions. **Listing 16.12** and **Listing 16.13** print the same lines.

Listing 16.12 `P1604ReadingOrder.java`: Problem 16.4 in Java.

```java
// P1604ReadingOrder.java - Problem 16.4: an order to read the topics (Kahn).
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.List;

public class P1604ReadingOrder {

    static final String[] TOPIC = {"Arrays", "Searching", "Sorting", "Stacks",
                                   "Pointers", "Lists", "Recursion", "Trees",
                                   "Graphs", "Heaps", "Hashing"};
    static final int[] CHAPTER = {1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15};

    // Kahn's algorithm. Returns the topics in an order in which every arc u -> v
    // has u before v. If some topics are missing from the result, they lie on
    // or behind a cycle and no valid order exists.
    static List<Integer> kahn(List<List<Integer>> adj) {
        int n = adj.size();
        int[] indeg = new int[n];
        for (List<Integer> out : adj)
            for (int v : out) indeg[v]++;
        ArrayDeque<Integer> ready = new ArrayDeque<>();   // no unread prerequisite
        for (int u = 0; u < n; u++)
            if (indeg[u] == 0) ready.add(u);
        List<Integer> order = new ArrayList<>();
        while (!ready.isEmpty()) {
            int u = ready.poll();
            order.add(u);
            for (int v : adj.get(u))
                if (--indeg[v] == 0) ready.add(v);    // v's last prerequisite read
        }
        return order;
    }

    static void report(List<List<Integer>> adj) {
        List<Integer> order = kahn(adj);
        int n = adj.size();
        if (order.size() == n) {
            System.out.println("A reading order:");
            for (int i = 0; i < n; i++)
                System.out.printf("%4d. %-10s (Chapter %d)%n", i + 1,
                        TOPIC[order.get(i)], CHAPTER[order.get(i)]);
            return;
        }
        boolean[] placed = new boolean[n];
        for (int u : order) placed[u] = true;
        System.out.println("No reading order exists: " + (n - order.size())
```

Listing 16.12 (Continued) *P1604ReadingOrder.java: Problem 16.4 in Java.*

```
                + " topics wait for each other in a cycle or behind one:");
        StringBuilder sb = new StringBuilder(" ");
        for (int u = 0; u < n; u++)
            if (!placed[u]) sb.append(" ").append(TOPIC[u]);
        System.out.println(sb);
    }

    public static void main(String[] args) {
        int n = TOPIC.length;
        int[][] needs = {                          // {u, v}: read u before v
            {0, 1}, {0, 2}, {0, 3}, {0, 4}, {4, 5}, {3, 5}, {5, 6}, {5, 7}, {6, 7},
            {1, 7}, {3, 8}, {5, 8}, {7, 9}, {2, 9}, {5, 10}, {1, 10}};
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < n; i++) adj.add(new ArrayList<>());
        for (int[] e : needs) adj.get(e[0]).add(e[1]);

        System.out.println("Each topic, and the topics that need it:");
        for (int u = 0; u < n; u++) {
            StringBuilder sb = new StringBuilder(
                    String.format("  %-10s->", TOPIC[u]));
            for (int v : adj.get(u)) sb.append(" ").append(TOPIC[v]);
            System.out.println(sb);
        }
        System.out.println();
        report(adj);

        System.out.println("\nNow suppose someone insists that Graphs must come "
                + "before Pointers.");
        adj.get(8).add(4);
        report(adj);
    }
}
```

Listing 16.13 `p1604_reading_order.py`: Problem 16.4 in Python.

```
# p1604_reading_order.py - Problem 16.4: an order to read the topics (Kahn).
from collections import deque

TOPIC = ["Arrays", "Searching", "Sorting", "Stacks", "Pointers", "Lists",
         "Recursion", "Trees", "Graphs", "Heaps", "Hashing"]
CHAPTER = [1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15]


def kahn(adj):
    """Returns the topics in an order in which every arc u -> v has u before v.
    Topics missing from the result lie on or behind a cycle."""
    n = len(adj)
    indeg = [0] * n
    for out in adj:
        for v in out:
            indeg[v] += 1
    ready = deque(u for u in range(n) if indeg[u] == 0)   # no unread prerequisite
    order = []
    while ready:
        u = ready.popleft()
        order.append(u)
        for v in adj[u]:
            indeg[v] -= 1
            if indeg[v] == 0:           # v's last prerequisite has been read
                ready.append(v)
    return order


def report(adj):
    order = kahn(adj)
    n = len(adj)
    if len(order) == n:
        print("A reading order:")
        for i, u in enumerate(order, 1):
            print(f"{i:4d}. {TOPIC[u]:<10} (Chapter {CHAPTER[u]})")
        return
    placed = set(order)
    print(f"No reading order exists: {n - len(order)} topics wait for each other"
          " in a cycle or behind one:")
    print(" " + "".join(" " + TOPIC[u] for u in range(n) if u not in placed))
```

Listing 16.13 (Continued) *p1604_reading_order.py: Problem 16.4 in Python.*

```
def main():
    n = len(TOPIC)
    needs = [(0, 1), (0, 2), (0, 3), (0, 4), (4, 5), (3, 5), (5, 6), (5, 7), (6, 7),
             (1, 7), (3, 8), (5, 8), (7, 9), (2, 9), (5, 10), (1, 10)]  # u before v
    adj = [[] for _ in range(n)]
    for u, v in needs:
        adj[u].append(v)

    print("Each topic, and the topics that need it:")
    for u in range(n):
        print(f"  {TOPIC[u]:<10}->" + "".join(" " + TOPIC[v] for v in adj[u]))
    print()
    report(adj)

    print("\nNow suppose someone insists that Graphs must come before Pointers.")
    adj[8].append(4)
    report(adj)


if __name__ == "__main__":
    main()
```

A tempting wrong solution. Leave out the closing comparison and print whatever the loop produced. With the cycle, `wrong/p1604_wrong.cpp` prints

```
Reading order: Arrays Searching Sorting Stacks
```

without any warning. A reader who trusts it would read four topics and then wonder where the other seven went. On an input without a cycle the program 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.

> **CHECK YOURSELF.** Is Arrays, Pointers, Stacks, Lists, Searching, Hashing, Sorting, Recursion, Trees, Heaps, Graphs a valid reading order for the sixteen arcs (without the extra one)? Could any topic other than Arrays be read first?
>
> *Answer.* Yes, it is valid: check each arc, for example Lists (4th) comes after Pointers (2nd) and Stacks (3rd), Hashing (6th) after Lists and Searching (5th), Trees (9th) after Lists, Recursion (8th) and Searching, Heaps (10th) after Trees and Sorting (7th), and Graphs (11th) after Stacks and Lists. No other topic can come first: Arrays is the only topic with in-degree 0.

16.5.3 Problem 16.5 — The quickest route for the book van

> **PROBLEM.** The roads of Problem 16.3 now have travel times in 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 road Pusat–Rungkut of 20 minutes. Find the quickest time from Pusat to every branch, the branch from which the van arrives, and the quickest routes to Rungkut, Sukolilo and Benowo, using a priority queue.

In our own words. Fewest hops is no longer the question: the direct road to Rungkut is one hop but 20 minutes, while Pusat → Gubeng → Rungkut is two hops and 11 minutes. With nonnegative lengths, Dijkstra's algorithm [11] answers the question. It keeps a tentative time `dist[v]` for every branch, starting at ∞ except `dist[Pusat] = 0`, and repeats: take the branch u with the smallest tentative time that is not yet *done*, declare it done, and *relax* every road u–v: if `dist[u] + w < dist[v]`, the road offers a quicker way to v, so lower `dist[v]` and remember u as its parent. Chapter 13 did "take the smallest" with a plain scan over all branches; Chapter 14 showed that a min-heap does it in $O(\lg n)$. We use the heap, with *lazy deletion*: when `dist[v]` drops we simply push a new entry $(\text{dist}[v], v)$, and when an entry for a branch that is already done comes out of the heap, we recognize it as stale and skip it.

> **DEFINITION.** The *Dijkstra invariant*: for every branch marked done, `dist` is the length of a quickest route from the source, and every tentative `dist` of a branch not yet done is the length of a quickest route whose last branch is the only one not done. The branch taken out of the heap with the smallest tentative time can then be declared done, because every other route to it would have to leave the done region through a branch whose time is already at least as large, and lengths are never negative.

The phrase "lengths are never negative" is doing real work: with a negative road, a route could leave the done region at a larger time and still arrive earlier, and the invariant would fail. Chapter 13 shows the failure on a small example; Bellman–Ford is the algorithm for that case [9].

By hand, step by step. The program prints its own trace: every branch it takes out of the heap, and every improvement it makes.

```
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
```

Read it with the invariant in mind. Pusat is taken at 0 and offers 5, 9, 14 and 20. The smallest tentative time is Gubeng's 5, so Gubeng is done: no route through Darmo (9 already) or Kenjeran (14) could reach Gubeng in less. From Gubeng, Rungkut improves from 20 to $5 + 6 = 11$; the entry (20, Rungkut) stays in the heap but is now stale. Darmo is taken at 9 and offers Wonokromo 13. Rungkut is taken at 11, and its road to Wonokromo offers $11 + 3 = 14$, not better than 13. Wonokromo at 13, Kenjeran at 14 (whose road offers Sukolilo $14 + 4 = 18$, not better than 17), Sukolilo at 17. Last, the stale (20, Rungkut) comes out and is skipped. **Figure 16.10** shows the settled times and, in dark lines, the *shortest-path tree*: each branch with the road through which it got its final time.

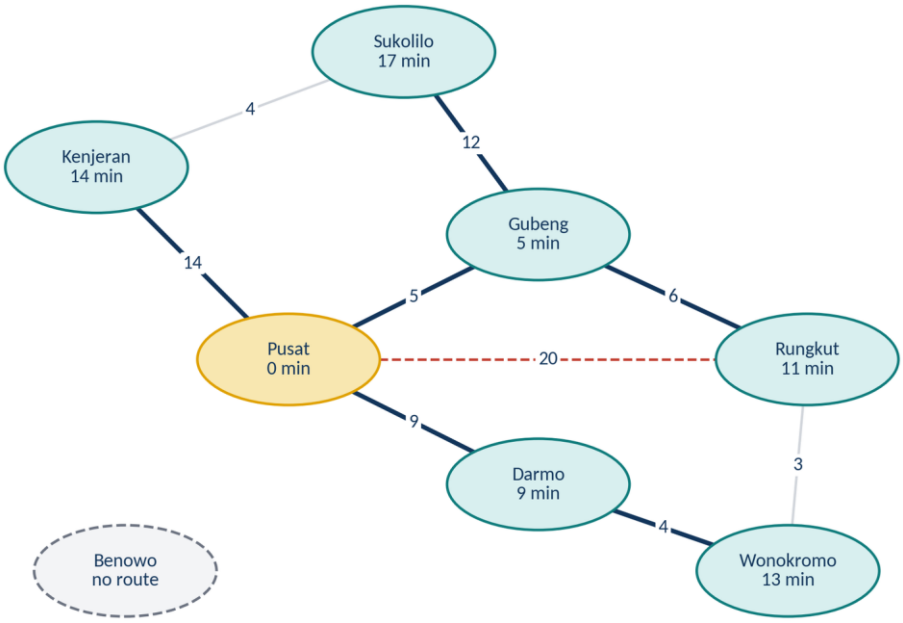


Figure 16.10 Quickest times from Pusat. Dark roads form the shortest-path tree; the dashed red direct road Pusat–Rungkut (20 minutes) loses to the detour through Gubeng (11 minutes).

Cost. Each successful relaxation pushes one entry, so the heap receives at most $1 + 2|E|$ entries, and each push or pop costs $O(\lg|E|) = O(\lg|V|)$, since $|E| < |V|^2$. Reading every adjacency list once adds $O(|V| + |E|)$, and so

$$T_{\text{Dijkstra}} = O((|V| + |E|)\lg|V|). \tag{16.6}$$

The version of Chapter 13 that scans for the smallest time costs $O(|V|^2)$ instead. For a road map, where $|E|$ is a small multiple of $|V|$, Equation (16.6) is far smaller: for a million intersections, about twenty million steps against a trillion.

The C++ solution. **Listing 16.14** is Dijkstra's algorithm with `std::priority_queue`. The C++ priority queue is a max-heap by default (**Table 16.3**); `greater<Entry>` turns it into a min-heap on pairs (time, branch).

Listing 16.14 `p1605_dijkstra.cpp`: Dijkstra's algorithm with a binary min-heap and lazy deletion, tracing every step.

```
// p1605_dijkstra.cpp - Problem 16.5: quickest routes for the book van (Dijkstra).
#include <algorithm>
#include <climits>
#include <functional>
#include <iomanip>
#include <iostream>
#include <queue>
#include <string>
#include <vector>
using namespace std;

const vector<string> NAME = {"Pusat", "Gubeng", "Darmo", "Kenjeran",
                             "Rungkut", "Wonokromo", "Sukolilo", "Benowo"};
const int INF = INT_MAX;

struct Arc {
    int to, minutes;
};

string show(int d) { return d == INF ? "inf" : to_string(d); }

// Dijkstra's algorithm with a binary min-heap and "lazy deletion":
// a vertex may sit in the heap several times; only its smallest entry counts.
void dijkstra(const vector<vector<Arc>>& adj, int s, vector<int>& dist,
              vector<int>& parent) {
    dist.assign(adj.size(), INF);
    parent.assign(adj.size(), -1);
    vector<bool> done(adj.size(), false);
    using Entry = pair<int, int>;                       // (distance, vertex)
    priority_queue<Entry, vector<Entry>, greater<Entry>> heap;   // smallest on top
    dist[s] = 0;
    heap.push({0, s});
    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]) {
                cout << "  " << NAME[a.to] << ": " << show(dist[a.to]) << " -> "
                     << d + a.minutes << "\n";
                dist[a.to] = d + a.minutes;
```

Listing 16.14 (Continued) *p1605_dijkstra.cpp: Dijkstra's algorithm with a binary min-heap and lazy deletion, tracing every step.*

```
                parent[a.to] = u;
                heap.push({dist[a.to], a.to});
            }
        }
    }
}

void printRoute(const vector<int>& dist, const vector<int>& parent, int t) {
    cout << "Quickest route to " << NAME[t] << ": ";
    if (dist[t] == INF) {
        cout << "no route\n";
        return;
    }
    vector<int> path;
    for (int v = t; v != -1; v = parent[v]) path.push_back(v);
    reverse(path.begin(), path.end());
    for (size_t i = 0; i < path.size(); i++)
        cout << (i ? " -> " : "") << NAME[path[i]];
    cout << " (" << dist[t] << " minutes)\n";
}

int main() {
    int n = NAME.size();
    int roads[][3] = {{0, 1, 5}, {0, 2, 9}, {0, 3, 14}, {1, 6, 12}, {1, 4, 6},
                      {2, 5, 4}, {5, 4, 3}, {3, 6, 4}, {0, 4, 20}};
    vector<vector<Arc>> adj(n);
    for (auto& r : roads) {                          // two-way roads
        adj[r[0]].push_back({r[1], r[2]});
        adj[r[1]].push_back({r[0], r[2]});
    }
    vector<int> dist, parent;
    dijkstra(adj, 0, dist, parent);

    cout << "\n" << left << setw(12) << "Branch" << setw(9) << "Minutes" << "Via\n";
    for (int v = 0; v < n; v++)
        cout << setw(12) << NAME[v] << setw(9) << show(dist[v])
             << (parent[v] == -1 ? "-" : NAME[parent[v]]) << "\n";
    cout << "\n";
    printRoute(dist, parent, 4);
    printRoute(dist, parent, 6);
    printRoute(dist, parent, 7);
    return 0;
}
```

```
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

Branch      Minutes  Via
Pusat       0        -
Gubeng      5        Pusat
Darmo       9        Pusat
Kenjeran    14       Pusat
Rungkut     11       Gubeng
Wonokromo   13       Darmo
Sukolilo    17       Gubeng
Benowo      inf      -

Quickest route to Rungkut: Pusat -> Gubeng -> Rungkut (11 minutes)
Quickest route to Sukolilo: Pusat -> Gubeng -> Sukolilo (17 minutes)
Quickest route to Benowo: no route
```

Line by line. `INT_MAX` plays ∞, and `show` prints it as `inf`; the program never adds anything to `INT_MAX`, because a branch whose time is infinite is never taken out of the heap (it was never pushed). `if (done[u])` is the stale-entry test. The condition `!done[a.to] && d + a.minutes < dist[a.to]` is the relaxation; the first half saves a pointless comparison, since a done branch can never improve. Pairs compare by their first member and then their second, so equal times are broken by the smaller branch number, in all three languages alike.

The Java and Python solutions. **Listing 16.15** and **Listing 16.16** print the same trace. Java's `PriorityQueue` is already a min-heap, and entries are `int[]` pairs with an explicit comparator; Python's `heapq` works on tuples, which compare element by element.

Listing 16.15 `P1605Dijkstra.java`: Problem 16.5 in Java.

```java
// P1605Dijkstra.java - Problem 16.5: quickest routes for the book van (Dijkstra).
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.PriorityQueue;

public class P1605Dijkstra {

    static final String[] NAME = {"Pusat", "Gubeng", "Darmo", "Kenjeran",
                                   "Rungkut", "Wonokromo", "Sukolilo", "Benowo"};
    static final int INF = Integer.MAX_VALUE;

    record Arc(int to, int minutes) {}

    static String show(int d) {
        return d == INF ? "inf" : Integer.toString(d);
    }

    // Dijkstra's algorithm with a binary min-heap and "lazy deletion":
    // a vertex may sit in the heap several times; only its smallest entry counts.
    static void dijkstra(List<List<Arc>> adj, int s, int[] dist, int[] parent) {
        Arrays.fill(dist, INF);
        Arrays.fill(parent, -1);
        boolean[] done = new boolean[adj.size()];
        PriorityQueue<int[]> heap = new PriorityQueue<>(      // {distance, vertex}
                (a, b) -> a[0] != b[0] ? Integer.compare(a[0], b[0])
                                       : Integer.compare(a[1], b[1]));
        dist[s] = 0;
        heap.add(new int[] {0, s});
        while (!heap.isEmpty()) {
            int[] top = heap.poll();
            int d = top[0], u = top[1];
            if (done[u]) {                                    // an old, larger entry
                System.out.println("skip " + NAME[u] + " (" + d
                        + "): an old, stale entry");
                continue;
            }
            done[u] = true;                                   // d is now final for u
            System.out.println("take " + NAME[u] + " (" + d + ")");
            for (Arc a : adj.get(u)) {
                if (!done[a.to()] && d + a.minutes() < dist[a.to()]) {
                    System.out.println("  " + NAME[a.to()] + ": "
                            + show(dist[a.to()]) + " -> " + (d + a.minutes()));
                    dist[a.to()] = d + a.minutes();
                    parent[a.to()] = u;
```

Listing 16.15 (Continued) *P1605Dijkstra.java: Problem 16.5 in Java.*

```
                    heap.add(new int[] {dist[a.to()], a.to()});
                }
            }
        }
    }

    static void printRoute(int[] dist, int[] parent, int t) {
        System.out.print("Quickest route to " + NAME[t] + ": ");
        if (dist[t] == INF) {
            System.out.println("no route");
            return;
        }
        List<String> path = new ArrayList<>();
        for (int v = t; v != -1; v = parent[v]) path.add(NAME[v]);
        Collections.reverse(path);
        System.out.println(String.join(" -> ", path) + " (" + dist[t]
                + " minutes)");
    }

    public static void main(String[] args) {
        int n = NAME.length;
        int[][] roads = {{0, 1, 5}, {0, 2, 9}, {0, 3, 14}, {1, 6, 12}, {1, 4, 6},
                         {2, 5, 4}, {5, 4, 3}, {3, 6, 4}, {0, 4, 20}};
        List<List<Arc>> adj = new ArrayList<>();
        for (int i = 0; i < n; i++) adj.add(new ArrayList<>());
        for (int[] r : roads) {                          // two-way roads
            adj.get(r[0]).add(new Arc(r[1], r[2]));
            adj.get(r[1]).add(new Arc(r[0], r[2]));
        }
        int[] dist = new int[n], parent = new int[n];
        dijkstra(adj, 0, dist, parent);

        System.out.printf("%n%-12s%-9s%s%n", "Branch", "Minutes", "Via");
        for (int v = 0; v < n; v++)
            System.out.printf("%-12s%-9s%s%n", NAME[v], show(dist[v]),
                    parent[v] == -1 ? "-" : NAME[parent[v]]);
        System.out.println();
        printRoute(dist, parent, 4);
        printRoute(dist, parent, 6);
        printRoute(dist, parent, 7);
    }
}
```

Listing 16.16 `p1605_dijkstra.py`: Problem 16.5 in Python.

```
# p1605_dijkstra.py - Problem 16.5: quickest routes for the book van (Dijkstra).
import heapq

NAME = ["Pusat", "Gubeng", "Darmo", "Kenjeran",
        "Rungkut", "Wonokromo", "Sukolilo", "Benowo"]
INF = float("inf")


def show(d):
    return "inf" if d == INF else str(d)


def dijkstra(adj, s):
    """Dijkstra's algorithm with a binary min-heap and lazy deletion: a vertex
    may sit in the heap several times; only its smallest entry counts."""
    dist = [INF] * len(adj)
    parent = [-1] * len(adj)
    done = [False] * len(adj)
    dist[s] = 0
    heap = [(0, s)]                                  # (distance, vertex)
    while heap:
        d, u = heapq.heappop(heap)
        if done[u]:                                  # an old, larger entry
            print(f"skip {NAME[u]} ({d}): an old, stale entry")
            continue
        done[u] = True                               # d is now final for u
        print(f"take {NAME[u]} ({d})")
        for v, minutes in adj[u]:
            if not done[v] and d + minutes < dist[v]:
                print(f"  {NAME[v]}: {show(dist[v])} -> {d + minutes}")
                dist[v] = d + minutes
                parent[v] = u
                heapq.heappush(heap, (dist[v], v))
    return dist, parent
```

Listing 16.16 (Continued) *p1605_dijkstra.py: Problem 16.5 in Python.*

```
def print_route(dist, parent, t):
    print(f"Quickest route to {NAME[t]}: ", end="")
    if dist[t] == INF:
        print("no route")
        return
    path, v = [], t
    while v != -1:
        path.append(NAME[v])
        v = parent[v]
    print(" -> ".join(reversed(path)), f"({dist[t]} minutes)")


def main():
    n = len(NAME)
    roads = [(0, 1, 5), (0, 2, 9), (0, 3, 14), (1, 6, 12), (1, 4, 6),
             (2, 5, 4), (5, 4, 3), (3, 6, 4), (0, 4, 20)]
    adj = [[] for _ in range(n)]
    for u, v, w in roads:                          # two-way roads
        adj[u].append((v, w))
        adj[v].append((u, w))
    dist, parent = dijkstra(adj, 0)

    print(f"\n{'Branch':<12}{'Minutes':<9}Via")
    for v in range(n):
        via = "-" if parent[v] == -1 else NAME[parent[v]]
        print(f"{NAME[v]:<12}{show(dist[v]):<9}{via}")
    print()
    print_route(dist, parent, 4)
    print_route(dist, parent, 6)
    print_route(dist, parent, 7)


if __name__ == "__main__":
    main()
```

A tempting wrong solution. The tempting simplification treats Dijkstra like BFS: give a branch its time the first time it is seen, and never look back.

```
   if (!seen[v]) { seen[v] = true; dist[v] = d + w; heap.push({dist[v], v}); }   // WRONG
```

`wrong/p1605_wrong.cpp` prints

```
Pusat 0
Gubeng 5
Darmo 9
Kenjeran 14
Rungkut 20
Wonokromo 13
Sukolilo 17
```

Rungkut is first seen from Pusat, along the 20-minute road, and is sealed at 20 before Gubeng, which offers 11, has even been taken. In BFS, first sight is final because every edge counts 1 and the queue meets vertices in distance order. With lengths, a vertex seen early may still be reached more quickly later; a time is final only when its branch comes out of the heap as the smallest. That is precisely the difference between the two invariants.

> **CHECK YOURSELF.** A new one-minute lane opens between Darmo and Rungkut. Which times and parents change?
>
> *Answer.* Darmo is taken at 9 and now offers Rungkut $9 + 1 = 10 < 11$, so Rungkut becomes 10 with parent Darmo. When Rungkut is taken at 10 its road to Wonokromo offers $10 + 3 = 13$, which is not *less* than 13, so Wonokromo keeps 13 with parent Darmo. Every other time is

> unchanged: Pusat 0, Gubeng 5, Darmo 9, Kenjeran 14, Wonokromo 13, Sukolilo 17, Benowo unreachable.

16.6 New Worked Problems on Heaps

Problem 16.1 built a heap operation by operation. The two problems of this section use a heap as a tool, and both use it in a way that surprises beginners: the first keeps a *min*-heap in order to find the *largest* values, and the second keeps two heaps facing each other in order to find the value in the *middle*.

16.6.1 Problem 16.6 — The k largest values with a heap of size k

> **PROBLEM.** (a) Eight books were borrowed 12, 45, 7, 30, 18, 3, 26 and 40 times this month (codes B1001 to B1008). Find the three most borrowed with a heap that never holds more than three entries, and show the heap after each book. (b) Find the five most-rated books of the ten-thousand-book catalog the same way, and check the answer against sorting the whole catalog.

In our own words. Sorting everything and taking the first k works, at a cost of $O(n\lg n)$ time and a copy of all n values. When k is much smaller than n, most of that work orders values we will throw away. Instead, keep the best k seen so far in a heap, and ask of every new value one question: *is it better than the worst of the ones I am keeping?* The worst of the kept ones must be found at once, so the heap is a **min**-heap: its root is the smallest of the k largest values so far, the value on the edge of being thrown out. A new value that is not larger than the root cannot be in the top k and is ignored in $O(1)$; a larger one replaces the root, which sinks into place in $O(\lg k)$.

> **DEFINITION.** The *top-k invariant*: after the first i values have been offered, the heap holds the $\min(i, k)$ largest of them, and its root is the smallest value it holds.

By hand, step by step. **Figure 16.11** follows part (a). The first three books fill the heap unconditionally: 12; 12, 45; 7, 12, 45. From then on each book is compared with the root. B1004 (30) beats the root 7: drop 7, keep 30. B1005 (18) beats 12. B1006 (3) does not beat 18 and is ignored at the cost of one comparison. B1007 (26) beats 18; B1008 (40) beats 26. The heap ends with 30, 40, 45, and read largest first that is B1002 (45), B1008 (40), B1004 (30).

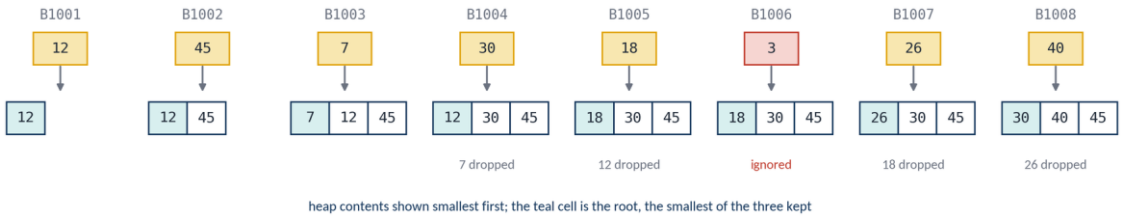


Figure 16.11 The three most borrowed books found with a min-heap of size 3. Each new count is compared with the root, the smallest of the three kept; if it is larger, the root is dropped.

Cost. Every value costs one comparison with the root, and at most one replacement of $O(\lg k)$, so

$$T_{\text{top-}k} = O(n\lg k), \qquad \text{extra memory } O(k). \tag{16.7}$$

For $n = 10{,}000$ and $k = 5$, $\lg k$ is about 2.3 against $\lg n \approx 13.3$ for sorting, and the memory is five entries instead of ten thousand. How many replacements happen depends on the order of the input: in the worst order, increasing, *every* value after the first k is a new record and causes a replacement; in the best order, decreasing, none does; and in a random order the i-th value is among the k largest so far with probability k/i, so the expected number of replacements is

$$\sum_{i=k+1}^{n} \frac{k}{i} \;\approx\; k \ln \frac{n}{k}, \tag{16.8}$$

which by Equation (16.8) is about $5\ln 2000 \approx 38$ for our catalog: a few dozen repairs among ten thousand values.

The C++ solution. **Listing 16.17** keeps the heap in a `vector` with the standard algorithms `push_heap` and `pop_heap`, so that it can print the heap's contents as it goes; `greater<Item>` makes them build a min-heap. Part (b) reads the catalog three times in different orders and counts the replacements each time.

Listing 16.17 `p1606_top_k.cpp`: the k largest values with a min-heap of size k, traced on eight loan counts and run on the whole catalog in three orders.

```
// p1606_top_k.cpp - Problem 16.6: the k largest values with a min-heap of size k.
#include <algorithm>
#include <fstream>
#include <functional>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

using Item = pair<long long, string>;              // (value, name)

// A min-heap kept in a vector: heap.front() is the SMALLEST of the k kept so far.
struct TopK {
    size_t k;
    vector<Item> heap;
    int replaced = 0;
    // Returns true if the item was kept.
    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;
    }
    vector<Item> largestFirst() const {
        vector<Item> v = heap;
        sort(v.rbegin(), v.rend());
        return v;
    }
    string contents() const {                      // smallest first, for tracing
        vector<Item> v = heap;
        sort(v.begin(), v.end());
        string s;
        for (const Item& x : v) s += " " + to_string(x.first);
        return s;
    }
};
```

Listing 16.17 (Continued) *p1606_top_k.cpp: the k largest values with a min-heap of size k, traced on eight loan counts and run on the whole catalog in three orders.*

```
int main() {
    cout << "Part 1: the three most-borrowed books this month (k = 3)\n";
    vector<Item> loans = {{12, "B1001"}, {45, "B1002"}, {7, "B1003"},
                          {30, "B1004"}, {18, "B1005"}, {3, "B1006"},
                          {26, "B1007"}, {40, "B1008"}};
    TopK small{3, {}};
    for (const Item& x : loans) {
        long long before = small.heap.size() == 3 ? small.heap.front().first : -1;
        bool kept = small.offer(x);
        cout << "  " << x.second << " (" << x.first << "): ";
        if (!kept)
            cout << "not more than the smallest kept (" << before << "), ignored\n";
        else
            cout << "heap holds" << small.contents() << "\n";
    }
    cout << "Top 3:";
    for (const Item& x : small.largestFirst())
        cout << " " << x.second << " (" << x.first << ")";
    cout << "\n\nPart 2: the five most-rated books of the catalog (k = 5)\n";

    ifstream in("../data/catalog.tsv");
    if (!in) {
        cout << "cannot open ../data/catalog.tsv\n";
        return 1;
    }
    string line;
    getline(in, line);                              // skip the header
    vector<pair<long long, Item>> books;            // (id, (ratings, title))
    while (getline(in, line)) {
        stringstream ss(line);
        string id, year, ratings, title;
        getline(ss, id, '\t');
        getline(ss, year, '\t');
        getline(ss, ratings, '\t');
        getline(ss, title, '\t');
        books.push_back({stoll(id), {stoll(ratings), title}});
    }
    cout << "Read " << books.size() << " books.\n";
```

Listing 16.17 (Continued) *p1606_top_k.cpp: the k largest values with a min-heap of size k, traced on eight loan counts and run on the whole catalog in three orders.*

```
    TopK inFileOrder{5, {}};                        // the file: most-rated first
    for (const auto& b : books) inFileOrder.offer(b.second);
    sort(books.begin(), books.end());               // now in order of identifier
    TopK top{5, {}};
    for (const auto& b : books) top.offer(b.second);
    vector<Item> rising;                            // the worst order: fewest first
    for (const auto& b : books) rising.push_back(b.second);
    sort(rising.begin(), rising.end());
    TopK worst{5, {}};
    for (const Item& x : rising) worst.offer(x);
    cout << "Changes to the heap after it filled up:\n  " << inFileOrder.replaced
         << " in file order (most-rated first), " << top.replaced
         << " in identifier order,\n  " << worst.replaced
         << " in increasing order of ratings.\n";

    vector<Item> best = top.largestFirst();
    for (size_t i = 0; i < best.size(); i++)
        cout << "  " << i + 1 << ". " << best[i].first << "  " << best[i].second
             << "\n";

    vector<Item>& all = rising;                     // the slow way: sort everything
    reverse(all.begin(), all.end());
    bool same = equal(best.begin(), best.end(), all.begin());
    cout << "Same five as sorting all " << all.size() << " books: "
         << (same ? "yes" : "no") << "\n";
    return 0;
}
```

```
Part 1: the three most-borrowed books this month (k = 3)
  B1001 (12): heap holds 12
  B1002 (45): heap holds 12 45
  B1003 (7): heap holds 7 12 45
  B1004 (30): heap holds 12 30 45
  B1005 (18): heap holds 18 30 45
  B1006 (3): not more than the smallest kept (18), ignored
  B1007 (26): heap holds 26 30 45
  B1008 (40): heap holds 30 40 45
Top 3: B1002 (45) B1008 (40) B1004 (30)

Part 2: the five most-rated books of the catalog (k = 5)
Read 10000 books.
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)
  3. 3916824  Twilight (Twilight, #1)
  4. 3340896  To Kill a Mockingbird
  5. 2773745  The Great Gatsby
Same five as sorting all 10000 books: yes
```

Line by line. `offer` is the whole algorithm. While the heap has fewer than k entries it pushes unconditionally. Afterwards `x <= heap.front()` rejects anything that does not beat the root; `pop_heap` moves the root to the back of the vector and restores the heap on the rest, the back cell is overwritten with the new item, and `push_heap` lets it rise into place. The catalog results show Equation (16.8) and its extremes in real numbers. The file lists the books from the most rated down, the best possible order: after the first five, nothing beats the root, and the heap never changes. In increasing order of ratings, the worst order, every one of the other 9,995 books causes a replacement. In order of identifier the heap changed only 9 times, even fewer than the 38 a random order would give, because the catalog's oldest and most famous books have the smallest

identifiers. The last line confirms that the five kept books are exactly the first five of the fully sorted catalog.

The Java and Python solutions. **Listing 16.18** and **Listing 16.19** print the same lines. Java's `PriorityQueue` and Python's `heapq` are min-heaps already; Python's `heapq.heapreplace` pops the root and pushes the new item in one sift-down, exactly our "replace the root" [7].

Listing 16.18 `P1606TopK.java`: Problem 16.6 in Java.

```
// P1606TopK.java - Problem 16.6: the k largest values with a min-heap of size k.
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.PriorityQueue;

public class P1606TopK {

    record Item(long value, String name) {}

    static final Comparator<Item> ORDER =
            Comparator.comparingLong(Item::value).thenComparing(Item::name);

    // A min-heap of at most k items: heap.peek() is the SMALLEST of those kept.
    static class TopK {
        final int k;
        final PriorityQueue<Item> heap = new PriorityQueue<>(ORDER);
        int replaced = 0;

        TopK(int k) {
            this.k = k;
        }

        boolean offer(Item x) {                    // true if the item was kept
            if (heap.size() < k) {                 // not full yet: always keep
                heap.add(x);
                return true;
            }
            if (ORDER.compare(x, heap.peek()) <= 0)
                return false;                      // not above the smallest kept
            heap.poll();                           // drop the smallest
            heap.add(x);
            replaced++;
            return true;
        }

        List<Item> largestFirst() {
            List<Item> v = new ArrayList<>(heap);
            v.sort(ORDER.reversed());
            return v;
        }
```

Listing 16.18 (Continued) *P1606TopK.java: Problem 16.6 in Java.*

```
        String contents() {                         // smallest first, for tracing
            List<Item> v = new ArrayList<>(heap);
            v.sort(ORDER);
            StringBuilder sb = new StringBuilder();
            for (Item x : v) sb.append(" ").append(x.value());
            return sb.toString();
        }
    }

    public static void main(String[] args) throws IOException {
        System.out.println(
                "Part 1: the three most-borrowed books this month (k = 3)");
        long[] counts = {12, 45, 7, 30, 18, 3, 26, 40};
        TopK small = new TopK(3);
        for (int i = 0; i < counts.length; i++) {
            Item x = new Item(counts[i], "B100" + (i + 1));
            long before = small.heap.size() == 3 ? small.heap.peek().value() : -1;
            boolean kept = small.offer(x);
            System.out.print("  " + x.name() + " (" + x.value() + "): ");
            if (!kept)
                System.out.println("not more than the smallest kept (" + before
                        + "), ignored");
            else
                System.out.println("heap holds" + small.contents());
        }
        StringBuilder sb = new StringBuilder("Top 3:");
        for (Item x : small.largestFirst())
            sb.append(" ").append(x.name()).append(" (").append(x.value())
              .append(")");
        System.out.println(sb);
        System.out.println(
                "\nPart 2: the five most-rated books of the catalog (k = 5)");

        List<String> lines = Files.readAllLines(Path.of("../data/catalog.tsv"),
                                                StandardCharsets.UTF_8);
        List<long[]> ids = new ArrayList<>();       // {id, index into items}
        List<Item> items = new ArrayList<>();
        for (String line : lines.subList(1, lines.size())) {    // skip the header
            String[] f = line.split("\t");
            ids.add(new long[] {Long.parseLong(f[0]), items.size()});
            items.add(new Item(Long.parseLong(f[2]), f[3]));
        }
        System.out.println("Read " + items.size() + " books.");
```

Listing 16.18 (Continued) *P1606TopK.java: Problem 16.6 in Java.*

```
        TopK inFileOrder = new TopK(5);            // the file: most-rated first
        for (Item x : items) inFileOrder.offer(x);
        ids.sort(Comparator.comparingLong(a -> a[0]));   // now in identifier order
        TopK top = new TopK(5);
        for (long[] a : ids) top.offer(items.get((int) a[1]));
        List<Item> rising = new ArrayList<>(items);      // the worst order
        rising.sort(ORDER);
        TopK worst = new TopK(5);
        for (Item x : rising) worst.offer(x);
        System.out.println("Changes to the heap after it filled up:\n  "
                + inFileOrder.replaced + " in file order (most-rated first), "
                + top.replaced + " in identifier order,\n  " + worst.replaced
                + " in increasing order of ratings.");

        List<Item> best = top.largestFirst();
        for (int i = 0; i < best.size(); i++)
            System.out.println("  " + (i + 1) + ". " + best.get(i).value() + "  "
                    + best.get(i).name());
        List<Item> all = new ArrayList<>(rising);  // the slow way: sort everything
        all.sort(ORDER.reversed());
        boolean same = best.equals(all.subList(0, 5));
        System.out.println("Same five as sorting all " + all.size() + " books: "
                + (same ? "yes" : "no"));
    }
}
```

Listing 16.19 `p1606_top_k.py`: Problem 16.6 in Python.

```python
# p1606_top_k.py - Problem 16.6: the k largest values with a min-heap of size k.
import heapq


class TopK:
    """A min-heap of at most k items: heap[0] is the SMALLEST of those kept."""

    def __init__(self, k):
        self.k, self.heap, self.replaced = k, [], 0

    def offer(self, x):                     # True if the item was kept
        if len(self.heap) < self.k:         # not full yet: always keep
            heapq.heappush(self.heap, x)
            return True
        if x <= self.heap[0]:               # not above the smallest kept
            return False
        heapq.heapreplace(self.heap, x)     # drop the smallest, add x
        self.replaced += 1
        return True

    def largest_first(self):
        return sorted(self.heap, reverse=True)

    def contents(self):                     # smallest first, for tracing
        return "".join(f" {v}" for v, _ in sorted(self.heap))


def main():
    print("Part 1: the three most-borrowed books this month (k = 3)")
    counts = [12, 45, 7, 30, 18, 3, 26, 40]
    small = TopK(3)
    for i, c in enumerate(counts, 1):
        x = (c, f"B100{i}")
        before = small.heap[0][0] if len(small.heap) == 3 else -1
        kept = small.offer(x)
        print(f"  {x[1]} ({c}): ", end="")
        if not kept:
            print(f"not more than the smallest kept ({before}), ignored")
        else:
            print("heap holds" + small.contents())
    print("Top 3:" + "".join(f" {name} ({v})" for v, name in small.largest_first()))
    print("\nPart 2: the five most-rated books of the catalog (k = 5)")
```

Listing 16.19 (Continued) *p1606_top_k.py: Problem 16.6 in Python.*

```
    books = []                              # (id, (ratings, title))
    with open("../data/catalog.tsv", encoding="utf-8") as f:
        next(f)                             # skip the header
        for line in f:
            fields = line.rstrip("\n").split("\t")
            books.append((int(fields[0]), (int(fields[2]), fields[3])))
    print(f"Read {len(books)} books.")

    in_file_order = TopK(5)                 # the file: most-rated first
    for _, x in books:
        in_file_order.offer(x)
    books.sort()                            # now in order of identifier
    top = TopK(5)
    for _, x in books:
        top.offer(x)
    rising = sorted(x for _, x in books)    # the worst order: fewest first
    worst = TopK(5)
    for x in rising:
        worst.offer(x)
    print(f"Changes to the heap after it filled up:\n  {in_file_order.replaced} in "
          f"file order (most-rated first), {top.replaced} in identifier order,\n  "
          f"{worst.replaced} in increasing order of ratings.")

    best = top.largest_first()
    for i, (v, title) in enumerate(best, 1):
        print(f"  {i}. {v}  {title}")
    everything = sorted(rising, reverse=True)   # the slow way: sort everything
    same = best == everything[:5]
    print(f"Same five as sorting all {len(everything)} books:",
          "yes" if same else "no")


if __name__ == "__main__":
    main()
```

A tempting wrong solution. "The largest values, so a max-heap": in C++ that is what `std::priority_queue<int>` gives by default, and it is tempting to push everything and pop whenever the size exceeds k:

```
   priority_queue<int> heap;            // WRONG: a MAX-heap; top() is the largest
   heap.push(c); if (heap.size() > 3) heap.pop();
```

`wrong/p1606_wrong.cpp` prints

```
Top 3: 12 7 3
```

the three *smallest* counts. Every `pop` removes the largest value seen so far, which is exactly the one we wanted to keep. The same code with Java's `PriorityQueue` or Python's `heapq` happens to be right, because they are min-heaps; this is the first dangerous row of **Table 16.3** at work. The rule to remember is the one in the invariant: to keep the k *largest*, you must be able to throw away the *smallest* of them quickly, so the heap's root must be the smallest.

> **CHECK YOURSELF.** With $k = 3$ and the counts 5, 9, 2, 9, 7, 1, 8, what does the heap hold after each value, and how many replacements happen? What changes if equal values are allowed to replace the root?
>
> *Answer.* 5; 5, 9; 2, 5, 9; then 9 beats 2: 5, 9, 9; 7 beats 5: 7, 9, 9; 1 is ignored; 8 beats 7: 8, 9, 9. Three replacements; the top three are 9, 9, 8. If equal values replaced the root, nothing would change here, because no value ever equals the root at the moment it arrives; with $k = 1$ and the counts 9, 9, the second 9 would cause a pointless replacement. Rejecting values equal to the root saves work and, for records with names, keeps the one that arrived first.

16.6.2 Problem 16.7 — The median so far, with two heaps

> **PROBLEM.** Visitors to the Pustaka Ceria reading desk waited 5, 15, 1, 3, 8, 7, 9, 10, 20 and 2 minutes, in that order. After each visitor, print the median of the waiting times so far (for an even count, the average of the two middle values). Each update must cost $O(\lg n)$, not a sort.

In our own words. The median is the value in the middle of the sorted list. Sorting after every visitor costs $O(n \lg n)$ each time. But we never need the whole sorted list, only its middle, so split the values into a lower half and an upper half. The middle values are the *largest of the lower half* and the *smallest of the upper half*: keep the lower half in a max-heap `low` and the upper half in a min-heap `high`, and both middle values sit at the roots.

> **DEFINITION.** The *two-heap invariant*: (i) every value in `low` is at most every value in `high`, which holds when `low.top() <= high.top()`; (ii) `low.size()` equals `high.size()` or `high.size() + 1`. Then the median is `low.top()` when the count is odd, and `(low.top() + high.top()) / 2` when it is even.

A new value goes into `low` if it is at most `low.top()`, otherwise into `high`; that keeps part (i). It may break part (ii), and one move repairs it: if `low` is two larger, move its top to `high`; if `high` is larger, move its top to `low`. Moving the *top* keeps part (i), because the top of `low` is the largest value of the lower half and becomes the smallest of the upper half, and symmetrically.

By hand, step by step. The program prints the two heaps (each listed in the order its values would come out) after every visitor:

```
add  5  median  5.0   low {5}  high {}
add 15  median 10.0   low {5}  high {15}
add  1  median  5.0   low {5 1}  high {15}
add  3  median  4.0   low {3 1}  high {5 15}
add  8  median  5.0   low {5 3 1}  high {8 15}
add  7  median  6.0   low {5 3 1}  high {7 8 15}
add  9  median  7.0   low {7 5 3 1}  high {8 9 15}
add 10  median  7.5   low {7 5 3 1}  high {8 9 10 15}
add 20  median  8.0   low {8 7 5 3 1}  high {9 10 15 20}
add  2  median  7.5   low {7 5 3 2 1}  high {8 9 10 15 20}
Two heaps agree with sorting every time: yes
```

Look at the fourth line. After 5, 15, 1 we have `low` = {5, 1} and `high` = {15}. The value 3 is at most 5, so it goes into `low`, which now has three values against one: too many. Its top, 5, moves to `high`: `low` = {3, 1}, `high` = {5, 15}, and the median is $(3 + 5)/2 = 4$. Check: the sorted list 1, 3, 5, 15 has middle values 3 and 5. After all ten visitors, **Figure 16.12** shows the two heaps; the median is $(7 + 8)/2 = 7.5$, which is the middle of the sorted list 1, 2, 3, 5, 7, 8, 9, 10, 15, 20. The last line of the output confirms that the two heaps agreed with a full sort after every single visitor.

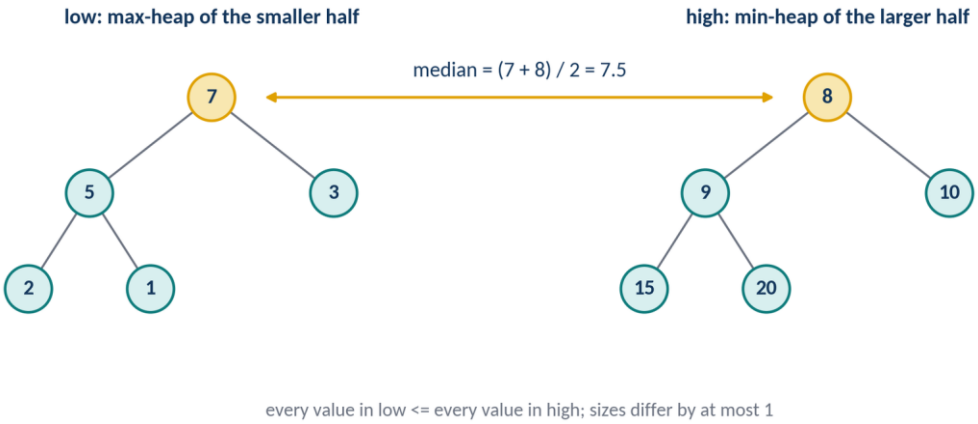


Figure 16.12 The two heaps after all ten waiting times. The roots are the two middle values of the sorted list, so the median is read off in constant time.

Cost. Each new value costs one push and at most one move between the heaps, a pop and a push: $O(\lg n)$. Reading the median is $O(1)$. For n values the whole stream costs $O(n \lg n)$, against $O(n^2 \lg n)$ for sorting after each one.

The C++ solution. **Listing 16.20** holds `low` as the default `priority_queue<int>`, which is a max-heap, and `high` with `greater<int>`. Here the C++ default is exactly what we want for the lower half.

Listing 16.20 `p1607_running_median.cpp`: the running median with a max-heap for the lower half and a min-heap for the upper half, checked against sorting after every value.

```cpp
// p1607_running_median.cpp - Problem 16.7: the median so far, with two heaps.
#include <algorithm>
#include <functional>
#include <iomanip>
#include <iostream>
#include <queue>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

class RunningMedian {
    priority_queue<int> low;                                // max-heap: smaller half
    priority_queue<int, vector<int>, greater<int>> high;  // min-heap: larger half
    // Invariant: every value in low <= every value in high, and
    //            low.size() == high.size() or low.size() == high.size() + 1.
public:
    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;
    }
    template <class PQ>
    static string list(PQ pq) {                             // a copy, emptied
        string s;
        while (!pq.empty()) {
            s += (s.empty() ? "" : " ") + to_string(pq.top());
            pq.pop();
        }
        return "{" + s + "}";
    }
    string show() const { return "low " + list(low) + "  high " + list(high); }
};
```

Listing 16.20 (Continued) *p1607_running_median.cpp: the running median with a max-heap for the lower half and a min-heap for the upper half, checked against sorting after every value.*

```
double bruteForceMedian(vector<int> v) {                  // sort everything: slow
    sort(v.begin(), v.end());
    size_t n = v.size();
    return n % 2 ? v[n / 2] : (v[n / 2 - 1] + v[n / 2]) / 2.0;
}

int main() {
    vector<int> waits = {5, 15, 1, 3, 8, 7, 9, 10, 20, 2};
    RunningMedian rm;
    vector<int> seen;
    bool agree = true;
    cout << fixed << setprecision(1);
    for (int x : waits) {
        rm.add(x);
        seen.push_back(x);
        agree = agree && rm.median() == bruteForceMedian(seen);
        cout << "add " << setw(2) << x << "  median " << setw(4) << rm.median()
             << "    " << rm.show() << "\n";
    }
    cout << "Two heaps agree with sorting every time: " << (agree ? "yes" : "no")
         << "\n";
    return 0;
}
```

Line by line. `add` does the three steps: place, then at most one of the two repairs. `median` uses `2.0` so that the average of 7 and 8 is 7.5 and not the integer 7. `list` is a *template* that receives a copy of either heap (the parameter `PQ pq` is passed by value) and empties the copy to print it in order; the real heaps are untouched. The comparison with `bruteForceMedian`, which sorts a copy of all values seen, is Step 9 of the method: a slow twin that is obviously right.

The Java and Python solutions. **Listing 16.21** and **Listing 16.22** print the same lines. Java builds the max-heap with `Collections.reverseOrder()`. Python's `heapq` has been min-heap only for most of its life, so `low` stores the *negated* values: the smallest negative number is the largest value. (Python 3.14 added `heapq.heappush_max` and its relatives, and Chapter 14 shows them; the negation trick runs on every version.)

Listing 16.21 `P1607RunningMedian.java`: Problem 16.7 in Java.

```java
// P1607RunningMedian.java - Problem 16.7: the median so far, with two heaps.
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.PriorityQueue;

public class P1607RunningMedian {

    static class RunningMedian {
        // max-heap for the smaller half, min-heap for the larger half.
        // Invariant: every value in low <= every value in high, and
        //            low.size() == high.size() or low.size() == high.size() + 1.
        final PriorityQueue<Integer> low =
                new PriorityQueue<>(Collections.reverseOrder());
        final PriorityQueue<Integer> high = new PriorityQueue<>();

        void add(int x) {
            if (low.isEmpty() || x <= low.peek()) low.add(x);
            else high.add(x);
            if (low.size() > high.size() + 1) high.add(low.poll());   // low too big
            else if (high.size() > low.size()) low.add(high.poll()); // high too big
        }

        double median() {
            if (low.size() > high.size()) return low.peek();
            return (low.peek() + high.peek()) / 2.0;
        }

        static String list(PriorityQueue<Integer> pq) {   // a copy, emptied
            PriorityQueue<Integer> copy = new PriorityQueue<>(pq.comparator());
            copy.addAll(pq);
            List<String> s = new ArrayList<>();
            while (!copy.isEmpty()) s.add(String.valueOf(copy.poll()));
            return "{" + String.join(" ", s) + "}";
        }

        String show() {
            return "low " + list(low) + "  high " + list(high);
        }
    }
```

Listing 16.21 (Continued) *P1607RunningMedian.java: Problem 16.7 in Java.*

```
    static double bruteForceMedian(List<Integer> seen) {  // sort everything: slow
        List<Integer> v = new ArrayList<>(seen);
        Collections.sort(v);
        int n = v.size();
        return n % 2 == 1 ? v.get(n / 2) : (v.get(n / 2 - 1) + v.get(n / 2)) / 2.0;
    }

    public static void main(String[] args) {
        int[] waits = {5, 15, 1, 3, 8, 7, 9, 10, 20, 2};
        RunningMedian rm = new RunningMedian();
        List<Integer> seen = new ArrayList<>();
        boolean agree = true;
        for (int x : waits) {
            rm.add(x);
            seen.add(x);
            agree = agree && rm.median() == bruteForceMedian(seen);
            System.out.printf("add %2d  median %4.1f   %s%n", x, rm.median(),
                    rm.show());
        }
        System.out.println("Two heaps agree with sorting every time: "
                + (agree ? "yes" : "no"));
    }
}
```

Listing 16.22 `p1607_running_median.py`: Problem 16.7 in Python, with the lower half stored negated.

```
# p1607_running_median.py - Problem 16.7: the median so far, with two heaps.
import heapq


class RunningMedian:
    """low holds the smaller half as a max-heap (values stored negated, because
    heapq is a min-heap); high holds the larger half as a min-heap.
    Invariant: every value in low <= every value in high, and
    len(low) == len(high) or len(low) == len(high) + 1."""

    def __init__(self):
        self.low, self.high = [], []

    def add(self, x):
        if not self.low or x <= -self.low[0]:
            heapq.heappush(self.low, -x)
        else:
            heapq.heappush(self.high, x)
        if len(self.low) > len(self.high) + 1:        # low too big: move its top
            heapq.heappush(self.high, -heapq.heappop(self.low))
        elif len(self.high) > len(self.low):          # high too big: move its top
            heapq.heappush(self.low, -heapq.heappop(self.high))

    def median(self):
        if len(self.low) > len(self.high):
            return float(-self.low[0])
        return (-self.low[0] + self.high[0]) / 2

    def show(self):
        low = " ".join(str(v) for v in sorted((-x for x in self.low), reverse=True))
        high = " ".join(str(v) for v in sorted(self.high))
        return f"low {{{low}}}  high {{{high}}}"


def brute_force_median(seen):                         # sort everything: slow
    v = sorted(seen)
    n = len(v)
    return float(v[n // 2]) if n % 2 else (v[n // 2 - 1] + v[n // 2]) / 2
```

Listing 16.22 (Continued) *p1607_running_median.py: Problem 16.7 in Python, with the lower half stored negated.*

```
def main():
    waits = [5, 15, 1, 3, 8, 7, 9, 10, 20, 2]
    rm, seen, agree = RunningMedian(), [], True
    for x in waits:
        rm.add(x)
        seen.append(x)
        agree = agree and rm.median() == brute_force_median(seen)
        print(f"add {x:2d}  median {rm.median():4.1f}   {rm.show()}")
    print("Two heaps agree with sorting every time:", "yes" if agree else "no")


if __name__ == "__main__":
    main()
```

A tempting wrong solution. Placing each value on the correct side feels like enough; the rebalancing looks like decoration. Without it (`wrong/p1607_wrong.cpp`):

```
add 5  median 5
add 15  median 10
add 1  median 5
add 3  median 10
add 8  median 5
add 7  median 6
add 9  median 5
add 10  median 6
add 20  median 5
add 2  median 6
```

The first three medians are right, then the fourth is 10 instead of 4, and from there on the answers are wrong more often than not. After 5, 15, 1, 3 the halves are {5, 3, 1} and {15}: part (i) of the invariant holds, but part (ii) does not, so the roots are no longer the middle values. The median is a statement about *positions*, and only the size condition ties the roots to the middle positions.

> **CHECK YOURSELF.** Continue the stream with 6 and then 30. Give the two heaps and the median after each.
>
> *Answer.* 6 is at most `low.top()` = 7, so it goes into `low`, which now has six values against five; the sizes are allowed (low may be one larger), so no move: `low` = {7, 6, 5, 3, 2, 1}, `high` = {8, 9, 10, 15, 20}, median 7 (eleven values; the sorted list 1, 2, 3, 5, 6, 7, 8, 9, 10, 15, 20 has 7 in the middle). Then 30 > 7 goes into `high`, sizes 6 and 6: median $(7 + 8)/2 = 7.5$.

16.7 New Worked Problems on Hashing

Problem 16.2 used a hash table from the library. The two problems of this section build hash tables by hand, one for each of the two collision strategies of Chapter 15, and each aims at the mistake that the library hides from us: an index computed with an integer that overflows, and a deletion that breaks the search for a key that is still there.

16.7.1 Problem 16.8 — A hash table with separate chaining, by hand

> **PROBLEM.** Store the eight books of Pustaka Ceria, keyed by title with the book code as value, in a hash table with separate chaining that starts with 5 buckets and doubles to $2m + 1$ buckets whenever the number of keys exceeds the number of buckets. Use the polynomial hash $h = (h \cdot 31 + c) \bmod m$ over the characters c of the title. Print the table before and after it grows; look up two titles that are present and one that is not; remove one title twice.

In our own words. An array of m *buckets*; bucket i is the head of a singly linked list (Chapter 7) of the entries whose key hashes to i. To find a key, compute its bucket and walk that one chain. To insert, walk the chain to check that the key is new, then put a node at the front. To remove, find the node and unlink it. The *load factor* is

$$\alpha = \frac{n}{m}, \tag{16.9}$$

the average chain length. With a hash function that spreads keys evenly, a search that fails walks a whole chain and a search that succeeds walks about half of one, so both cost [9]

$$T_{\text{chain}} = \Theta(1 + \alpha). \tag{16.10}$$

By Equation (16.9) and Equation (16.10), keeping $\alpha \le 1$, as our rule does, keeps every operation $O(1)$ on average; the price is an occasional *rehash*, which moves every node into a larger array. Because the array roughly doubles each time, the total cost of all rehashes over n insertions is $O(n)$, so each insertion pays $O(1)$ amortized (Chapter 15).

> **DEFINITION.** The *chaining invariant*: every stored key k is in the chain of bucket $h(k)$, and appears there exactly once.

By hand, step by step. The hash of a title is computed one character at a time, and reduced modulo m after every step so that the running value stays small. The reduction does not change the result, because $(a \bmod m) \cdot 31 + c \equiv a \cdot 31 + c \pmod m$. For "Fisika Dasar" and $m = 11$, starting from $h = 0$: `F` (code 70) gives $70 \bmod 11 = 4$; `i` (105) gives $(4 \cdot 31 + 105) \bmod 11 = 229 \bmod 11 = 9$; `s` (115) gives $394 \bmod 11 = 9$; and so on through the space (32) and `D`, `a`, `s`, `a`, `r`, ending at 9. So "Fisika Dasar" lives in bucket 9, which is what the program prints. The program shows every bucket and every chain:

```
put "Laskar Pelangi" -> bucket 2
put "Bumi Manusia" -> bucket 2
put "Filosofi Kopi" -> bucket 2
put "Sejarah Nusantara" -> bucket 0
put "Algoritma & Struktur Data" -> bucket 4
m = 5, n = 5, load factor 1.00
  [ 0] Sejarah Nusantara
  [ 1] -
  [ 2] Filosofi Kopi -> Bumi Manusia -> Laskar Pelangi
  [ 3] -
  [ 4] Algoritma & Struktur Data
put "Cerita Rakyat Nusantara" -> bucket 0
  (load factor 6/5 > 1: rehash to 11 buckets)
put "Fisika Dasar" -> bucket 9
put "Negeri 5 Menara" -> bucket 7
m = 11, n = 8, load factor 0.73
  [ 0] -
  [ 1] Algoritma & Struktur Data -> Cerita Rakyat Nusantara
  [ 2] -
  [ 3] Sejarah Nusantara
  [ 4] -
  [ 5] -
  [ 6] -
  [ 7] Negeri 5 Menara -> Laskar Pelangi
  [ 8] Bumi Manusia
  [ 9] Fisika Dasar
  [10] Filosofi Kopi
get "Fisika Dasar" -> B1007
get "Laskar Pelangi" -> B1001
get "Harry Potter" -> not found
remove "Bumi Manusia" -> true
remove "Bumi Manusia" -> false
m = 11, n = 7, load factor 0.64
  [ 0] -
  [ 1] Algoritma & Struktur Data -> Cerita Rakyat Nusantara
  [ 2] -
  [ 3] Sejarah Nusantara
  [ 4] -
  [ 5] -
  [ 6] -
  [ 7] Negeri 5 Menara -> Laskar Pelangi
  [ 8] -
  [ 9] Fisika Dasar
  [10] Filosofi Kopi
```

The first table is the interesting one. With only 5 buckets, three of the first five titles land in bucket 2 and form a chain of three; the newest, "Filosofi Kopi", is at the front because insertion is at the front. The sixth title raises n to 6 > 5, and the table grows to 11 buckets. Every node is *relinked*, not copied, into the bucket its key hashes to modulo 11. After the eighth title, $\alpha = 8/11 \approx 0.73$, the longest chain has two nodes, and six buckets are empty. That is normal:

throwing 8 keys at 11 buckets at random leaves about $11(1 - 1/11)^8 \approx 5$ of them empty. **Figure 16.13** draws the table before and after the rehash.

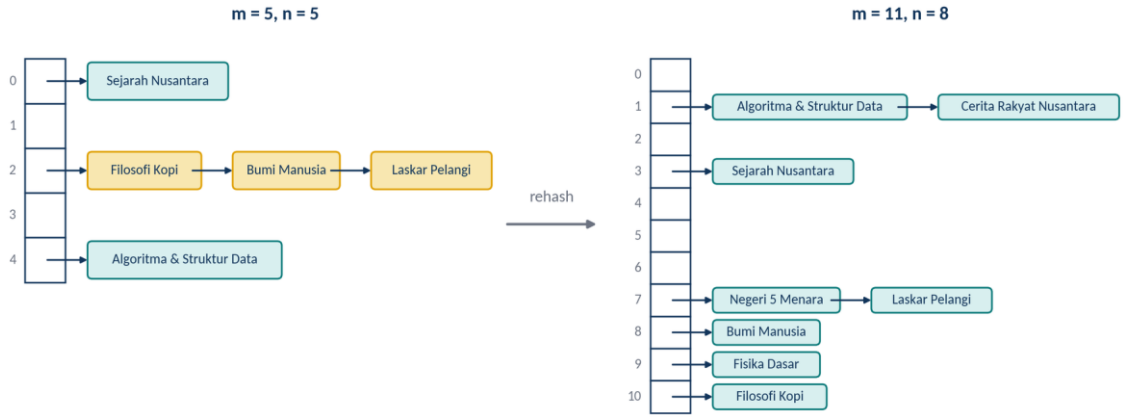


Figure 16.13 The chained table with 5 buckets (three titles collide in bucket 2) and after rehashing to 11 buckets, where no chain is longer than two.

The C++ solution. **Listing 16.23** builds the nodes by hand with `new` and frees them in the destructor, as in Chapter 7; copying the table is forbidden (`= delete`) because a copy would share the nodes and free them twice.

Listing 16.23 `p1608_chaining.cpp`: a hash table with separate chaining, front insertion, removal through a pointer to the link, and rehashing when the load factor exceeds 1.

```
// p1608_chaining.cpp - Problem 16.8: a hash table with separate chaining, by hand.
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class ChainedTable {
    struct Node {
        string key, value;
        Node* next;
    };
    vector<Node*> bucket;          // bucket[i]: first node of chain i, or nullptr
    size_t n = 0;                  // number of keys stored

    // 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;
    }

    void rehash(size_t newM) {     // relink every node into a bigger table
        vector<Node*> old = bucket;
        bucket.assign(newM, nullptr);
        for (Node* head : old) {
            while (head != nullptr) {
                Node* next = head->next;                  // save the way on first
                size_t i = hashOf(head->key, newM);
                head->next = bucket[i];                   // front insertion
                bucket[i] = head;
                head = next;
            }
        }
    }
```

Listing 16.23 (Continued) *p1608_chaining.cpp: a hash table with separate chaining, front insertion, removal through a pointer to the link, and rehashing when the load factor exceeds 1.*

```
public:
    explicit ChainedTable(size_t m) : bucket(m, nullptr) {}
    ~ChainedTable() {
        for (Node* head : bucket)
            while (head != nullptr) {
                Node* next = head->next;
                delete head;
                head = next;
            }
    }
    ChainedTable(const ChainedTable&) = delete;
    ChainedTable& operator=(const ChainedTable&) = delete;

    size_t indexOf(const string& key) const { return hashOf(key, bucket.size()); }

    void put(const string& key, const string& value) {
        size_t i = indexOf(key);
        for (Node* p = bucket[i]; p != nullptr; p = p->next)
            if (p->key == key) {                          // already there: update
                p->value = value;
                return;
            }
        bucket[i] = new Node{key, value, bucket[i]};      // front of chain i
        n++;
        if (n > bucket.size()) {                          // load factor above 1
            cout << "  (load factor " << n << "/" << bucket.size()
                 << " > 1: rehash to " << 2 * bucket.size() + 1 << " buckets)\n";
            rehash(2 * bucket.size() + 1);
        }
    }

    const string* get(const string& key) const {          // nullptr if absent
        for (Node* p = bucket[indexOf(key)]; p != nullptr; p = p->next)
            if (p->key == key) return &p->value;
        return nullptr;
    }
```

Listing 16.23 (Continued) *p1608_chaining.cpp: a hash table with separate chaining, front insertion, removal through a pointer to the link, and rehashing when the load factor exceeds 1.*

```
    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;
    }

    void print() const {
        cout << "m = " << bucket.size() << ", n = " << n << ", load factor "
             << fixed << setprecision(2) << (double)n / bucket.size() << "\n";
        for (size_t i = 0; i < bucket.size(); i++) {
            cout << "  [" << setw(2) << i << "]";
            if (bucket[i] == nullptr) cout << " -";
            for (Node* p = bucket[i]; p != nullptr; p = p->next)
                cout << (p != bucket[i] ? " -> " : " ") << p->key;
            cout << "\n";
        }
    }
};

int main() {
    const char* books[][2] = {
        {"Laskar Pelangi", "B1001"}, {"Bumi Manusia", "B1002"},
        {"Filosofi Kopi", "B1003"}, {"Sejarah Nusantara", "B1004"},
        {"Algoritma & Struktur Data", "B1005"},
        {"Cerita Rakyat Nusantara", "B1006"},
        {"Fisika Dasar", "B1007"}, {"Negeri 5 Menara", "B1008"}};
    ChainedTable t(5);
    for (int i = 0; i < 8; i++) {
        cout << "put \"" << books[i][0] << "\" -> bucket " << t.indexOf(books[i][0])
             << "\n";
        t.put(books[i][0], books[i][1]);
        if (i == 4) t.print();
    }
    t.print();
    for (const char* q : {"Fisika Dasar", "Laskar Pelangi", "Harry Potter"}) {
        const string* v = t.get(q);
        cout << "get \"" << q << "\" -> " << (v ? *v : "not found") << "\n";
    }
    cout << "remove \"Bumi Manusia\" -> " << boolalpha << t.remove("Bumi Manusia")
         << "\nremove \"Bumi Manusia\" -> " << t.remove("Bumi Manusia") << "\n";
    t.print();
```

Listing 16.23 (Continued) *p1608_chaining.cpp: a hash table with separate chaining, front insertion, removal through a pointer to the link, and rehashing when the load factor exceeds 1.*

```
    return 0;
}
```

Line by line. `hashOf` uses `unsigned char` so that a byte above 127 (an accented letter in UTF-8) counts as a positive number, and `size_t` arithmetic reduced at every step, so no intermediate value ever exceeds $31 \cdot m + 255$. `rehash` walks every old chain, saving `head->next` before relinking the node into the new array, the rule of Chapter 8. `remove` uses a *pointer to the link*, `Node** link`, which first points at the bucket's head pointer and then at the `next` field of each node in turn; when the key is found, `*link = dead->next` unlinks the node whether it was first in its chain or not, without a special case, and only then is the node deleted. Valgrind and AddressSanitizer report no error and no leak.

The Java and Python solutions. **Listing 16.24** and **Listing 16.25** print the same lines. Neither language exposes a pointer to a link, so `remove` walks with the usual pair `prev`, `p` and handles the first node separately.

Listing 16.24 `P1608Chaining.java`: Problem 16.8 in Java.

```
// P1608Chaining.java - Problem 16.8: a hash table with separate chaining, by hand.
public class P1608Chaining {

    static class ChainedTable {
        static class Node {
            final String key;
            String value;
            Node next;

            Node(String key, String value, Node next) {
                this.key = key;
                this.value = value;
                this.next = next;
            }
        }

        Node[] bucket;             // bucket[i]: first node of chain i, or null
        int n = 0;                 // number of keys stored

        ChainedTable(int m) {
            bucket = new Node[m];
        }

        // Polynomial hash, reduced modulo m at every step so it never overflows.
        static int hashOf(String key, int m) {
            int h = 0;
            for (int i = 0; i < key.length(); i++) h = (h * 31 + key.charAt(i)) % m;
            return h;
        }

        int indexOf(String key) {
            return hashOf(key, bucket.length);
        }

        void rehash(int newM) {    // relink every node into a bigger table
            Node[] old = bucket;
            bucket = new Node[newM];
            for (Node head : old) {
                while (head != null) {
                    Node next = head.next;              // save the way on first
                    int i = hashOf(head.key, newM);
                    head.next = bucket[i];              // front insertion
                    bucket[i] = head;
                    head = next;
                }
            }
```

Listing 16.24 (Continued) *P1608Chaining.java: Problem 16.8 in Java.*

```
    }

    void put(String key, String value) {
        int i = indexOf(key);
        for (Node p = bucket[i]; p != null; p = p.next)
            if (p.key.equals(key)) {                // already there: update
                p.value = value;
                return;
            }
        bucket[i] = new Node(key, value, bucket[i]); // front of chain i
        n++;
        if (n > bucket.length) {                    // load factor above 1
            System.out.println("  (load factor " + n + "/" + bucket.length
                    + " > 1: rehash to " + (2 * bucket.length + 1)
                    + " buckets)");
            rehash(2 * bucket.length + 1);
        }
    }

    String get(String key) {                        // null if absent
        for (Node p = bucket[indexOf(key)]; p != null; p = p.next)
            if (p.key.equals(key)) return p.value;
        return null;
    }

    boolean remove(String key) {
        int i = indexOf(key);
        Node prev = null, p = bucket[i];
        while (p != null && !p.key.equals(key)) {
            prev = p;
            p = p.next;
        }
        if (p == null) return false;
        if (prev == null) bucket[i] = p.next;       // unlink the first node
        else prev.next = p.next;                    // unlink a later node
        n--;
        return true;
    }
```

Listing 16.24 (Continued) *P1608Chaining.java: Problem 16.8 in Java.*

```
        void print() {
            System.out.printf("m = %d, n = %d, load factor %.2f%n", bucket.length,
                    n, (double) n / bucket.length);
            for (int i = 0; i < bucket.length; i++) {
                StringBuilder sb = new StringBuilder(String.format("  [%2d]", i));
                if (bucket[i] == null) sb.append(" -");
                for (Node p = bucket[i]; p != null; p = p.next)
                    sb.append(p != bucket[i] ? " -> " : " ").append(p.key);
                System.out.println(sb);
            }
        }
    }

    public static void main(String[] args) {
        String[][] books = {
            {"Laskar Pelangi", "B1001"}, {"Bumi Manusia", "B1002"},
            {"Filosofi Kopi", "B1003"}, {"Sejarah Nusantara", "B1004"},
            {"Algoritma & Struktur Data", "B1005"},
            {"Cerita Rakyat Nusantara", "B1006"},
            {"Fisika Dasar", "B1007"}, {"Negeri 5 Menara", "B1008"}};
        ChainedTable t = new ChainedTable(5);
        for (int i = 0; i < 8; i++) {
            System.out.println("put \"" + books[i][0] + "\" -> bucket "
                    + t.indexOf(books[i][0]));
            t.put(books[i][0], books[i][1]);
            if (i == 4) t.print();
        }
        t.print();
        for (String q : new String[] {"Fisika Dasar", "Laskar Pelangi",
                                      "Harry Potter"}) {
            String v = t.get(q);
            System.out.println("get \"" + q + "\" -> "
                    + (v != null ? v : "not found"));
        }
        for (int i = 0; i < 2; i++)
            System.out.println("remove \"Bumi Manusia\" -> "
                    + t.remove("Bumi Manusia"));
        t.print();
    }
}
```

Listing 16.25 `p1608_chaining.py`: Problem 16.8 in Python.

```
# p1608_chaining.py - Problem 16.8: a hash table with separate chaining, by hand.


class Node:
    def __init__(self, key, value, next_node):
        self.key, self.value, self.next = key, value, next_node


class ChainedTable:
    def __init__(self, m):
        self.bucket = [None] * m    # bucket[i]: first node of chain i, or None
        self.n = 0                  # number of keys stored

    @staticmethod
    def hash_of(key, m):
        """Polynomial hash, reduced modulo m at every step (as in C++ and Java)."""
        h = 0
        for c in key:
            h = (h * 31 + ord(c)) % m
        return h

    def index_of(self, key):
        return self.hash_of(key, len(self.bucket))

    def _rehash(self, new_m):       # relink every node into a bigger table
        old, self.bucket = self.bucket, [None] * new_m
        for head in old:
            while head is not None:
                nxt = head.next                         # save the way on first
                i = self.hash_of(head.key, new_m)
                head.next = self.bucket[i]              # front insertion
                self.bucket[i] = head
                head = nxt

    def put(self, key, value):
        i = self.index_of(key)
        p = self.bucket[i]
        while p is not None:
            if p.key == key:                            # already there: update
                p.value = value
                return
            p = p.next
        self.bucket[i] = Node(key, value, self.bucket[i])   # front of chain i
        self.n += 1
        m = len(self.bucket)
        if self.n > m:                                  # load factor above 1
```

Listing 16.25 (Continued) *p1608_chaining.py: Problem 16.8 in Python.*

```
            print(f"  (load factor {self.n}/{m} > 1: "
                  f"rehash to {2 * m + 1} buckets)")
            self._rehash(2 * m + 1)

    def get(self, key):                                  # None if absent
        p = self.bucket[self.index_of(key)]
        while p is not None:
            if p.key == key:
                return p.value
            p = p.next
        return None

    def remove(self, key):
        i = self.index_of(key)
        prev, p = None, self.bucket[i]
        while p is not None and p.key != key:
            prev, p = p, p.next
        if p is None:
            return False
        if prev is None:
            self.bucket[i] = p.next                      # unlink the first node
        else:
            prev.next = p.next                           # unlink a later node
        self.n -= 1
        return True

    def print(self):
        m = len(self.bucket)
        print(f"m = {m}, n = {self.n}, load factor {self.n / m:.2f}")
        for i, head in enumerate(self.bucket):
            keys = []
            while head is not None:
                keys.append(head.key)
                head = head.next
            print(f"  [{i:2d}] " + (" -> ".join(keys) if keys else "-"))
```

Listing 16.25 (Continued) *p1608_chaining.py: Problem 16.8 in Python.*

```
def main():
    books = [("Laskar Pelangi", "B1001"), ("Bumi Manusia", "B1002"),
             ("Filosofi Kopi", "B1003"), ("Sejarah Nusantara", "B1004"),
             ("Algoritma & Struktur Data", "B1005"),
             ("Cerita Rakyat Nusantara", "B1006"),
             ("Fisika Dasar", "B1007"), ("Negeri 5 Menara", "B1008")]
    t = ChainedTable(5)
    for i, (title, code) in enumerate(books):
        print(f'put "{title}" -> bucket {t.index_of(title)}')
        t.put(title, code)
        if i == 4:
            t.print()
    t.print()
    for q in ("Fisika Dasar", "Laskar Pelangi", "Harry Potter"):
        v = t.get(q)
        print(f'get "{q}" -> {v if v is not None else "not found"}')
    print('remove "Bumi Manusia" ->', str(t.remove("Bumi Manusia")).lower())
    print('remove "Bumi Manusia" ->', str(t.remove("Bumi Manusia")).lower())
    t.print()


if __name__ == "__main__":
    main()
```

A tempting wrong solution. The textbook formula $h = \sum c_i \cdot 31^{L-1-i}$ invites a direct transcription with a plain `int`, reducing modulo m only at the end:

```
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
```

`wrong/p1608_wrong.cpp` computes the bucket of "Laskar Pelangi" this way and stores the title there. Compiled normally, it prints

```
Laskar Pelangi -> bucket -1
Segmentation fault
```

and, compiled with the sanitizers, it reports both of its bugs:

```
p1608_wrong.cpp:9:30: runtime error: signed integer overflow:
    73191252 * 31 cannot be represented in type 'int'
Laskar Pelangi -> bucket -1
==ERROR: AddressSanitizer: heap-buffer-overflow ... READ of size 8 ...
    #1 ... in main ... p1608_wrong.cpp:18
```

31^6 is already about 8.9×10^8, so after six or seven characters $h \cdot 31$ passes $2^{31} - 1 \approx 2.1 \times 10^9$. In C++, overflow of a *signed* integer is undefined behavior; in practice the value wraps to a negative number, and the remainder of a negative number is negative in C++ and Java (Chapter 8), so the index is -1 and `bucket[-1]` lies before the array. The fixes are the two things our `hashOf` does: unsigned arithmetic, whose wraparound is defined, and reduction at every step. In Java the same slip gives an `ArrayIndexOutOfBoundsException`; use `Math.floorMod(h, m)`. Python's integers never overflow and `%` never returns a negative result for a positive m, so the bug cannot happen there, but the unreduced value grows to hundreds of digits for a long key and slows every hash down.

> **CHECK YOURSELF.** In the 11-bucket table, which buckets are searched, and how many titles are compared, for `get("Negeri 5 Menara")`, `get("Laskar Pelangi")` and `get("Harry Potter")`? (The hash of "Harry Potter" modulo 11 is 4.)
>
> *Answer.* "Negeri 5 Menara": bucket 7, one comparison (it is at the front). "Laskar Pelangi": bucket 7, two comparisons. "Harry Potter": bucket 4, which is empty, so zero comparisons and the answer "not found". By Equation (16.10), the cost depends on the chain, never on the eight titles as a whole.

16.7.2 Problem 16.9 — Open addressing, and deleting safely

> **PROBLEM.** Insert the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a table of 11 slots with $h(k) = k \bmod 11$ and linear probing. Then remove 15, search for 59 and for 26, and insert 37. Show every probe sequence and the table after each change.

In our own words. *Open addressing* keeps the keys in the array itself, one per slot. If slot $h(k)$ is taken, *linear probing* tries $h(k)+1$, $h(k)+2$, and so on, wrapping around at the end. A search follows the same sequence and may stop as soon as it reaches an *empty* slot, because an insertion of k would have stopped there too. That stopping rule is exactly what makes deletion delicate: emptying the slot of a deleted key can cut the probe path of another key that was pushed past it.

> **DEFINITION.** The *probing invariant*: if key k is stored in slot j, then every slot on the probe path from $h(k)$ up to j (exclusive) is occupied or is a *tombstone*, a slot marked "deleted". A search therefore skips tombstones and stops only at a truly empty slot; an insertion may reuse the first tombstone it passes, after checking that the key is not present further on.

By hand, step by step. This is a classic exercise of Cormen et al. [9]. The program prints every probe sequence:

```
h(k) = k mod 11, linear probing
insert 10: probes 10 -> slot 10
insert 22: probes 0 -> slot 0
insert 31: probes 9 -> slot 9
insert  4: probes 4 -> slot 4
insert 15: probes 4 5 -> slot 5
insert 28: probes 6 -> slot 6
insert 17: probes 6 7 -> slot 7
insert 88: probes 0 1 -> slot 1
insert 59: probes 4 5 6 7 8 -> slot 8
  slot:   0   1   2   3   4   5   6   7   8   9  10
  key:   22  88   .   .   4  15  28  17  59  31  10

remove 15: true
  slot:   0   1   2   3   4   5   6   7   8   9  10
  key:   22  88   .   .   4 del  28  17  59  31  10
search 59: probes 4 5 6 7 8 -> slot 8
search 26: probes 4 5 6 7 8 9 10 0 1 2 -> not found
insert 37: probes 4 5 6 7 8 9 10 0 1 2 -> slot 5 (the tombstone)
  slot:   0   1   2   3   4   5   6   7   8   9  10
  key:   22  88   .   .   4  37  28  17  59  31  10
```

The first four keys land at home: $10 \to 10$, $22 \to 0$, $31 \to 9$, $4 \to 4$. Then $15 \bmod 11 = 4$ collides with 4 and moves to 5. $28 \to 6$. $17 \bmod 11 = 6$ collides and moves to 7. $88 \bmod 11 = 0$ collides with 22 and moves to 1. And $59 \bmod 11 = 4$ finds 4, 5, 6, 7 all taken and lands in 8 after five probes. Keys that hash near each other have merged into one *cluster*, slots 4 to 10 and around to 0 and 1, and every new key hashing into it must walk to its end: this *primary clustering* is the weakness of linear probing.

Now the deletion. Slot 5 held 15, and 59's probe path runs through it. Our `remove` marks slot 5 `del`, a tombstone. The search for 59 probes 4 (4, not 59), 5 (a tombstone: keep going), 6, 7, 8: found. The search for 26 ($26 \bmod 11 = 4$) walks the whole cluster, 4 to 10 and 0 and 1, and stops at the empty slot 2: not present, after ten probes. Finally 37 ($37 \bmod 11 = 4$): the insertion remembers the tombstone at 5 as the first free slot, but keeps probing until the empty slot 2 to make sure 37 is not already stored further on; it is not, so 37 goes into slot 5, and the tombstone is reused. **Figure 16.14** shows the table before and after the removal.

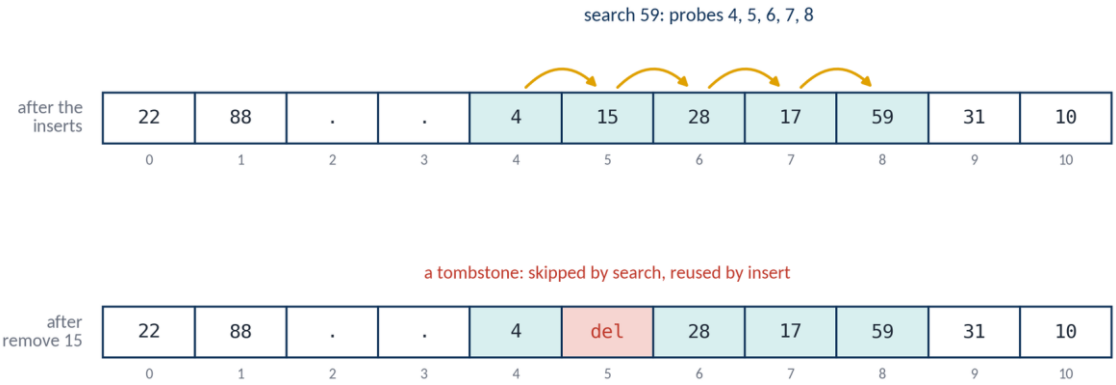


Figure 16.14 Linear probing with $m = 11$: the probe path of 59 runs through slots 4 to 8. Removing 15 leaves a tombstone in slot 5, which a search skips and an insertion reuses.

Cost. For a load factor $\alpha < 1$ and a hash function that spreads keys evenly, Knuth's classic analysis of linear probing [12] gives the average number of probes of a successful search, S, and of an unsuccessful one, U:

$$S \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right), \qquad U \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right). \tag{16.11}$$

At $\alpha = 0.5$, Equation (16.11) gives 1.5 and 2.5 probes; at our $\alpha = 9/11 \approx 0.82$ it gives about 3.3 and 15.6, and the unsuccessful search for 26 indeed needed ten. The quadratic term is why open-addressing tables are kept well below full: Python's `dict`, which uses open addressing, keeps its load at most two thirds, and even Java's chained `HashMap` grows at 0.75 [6], [7]. Tombstones count toward the load until a rehash sweeps them away.

The C++ solution. Listing 16.26 keeps a key array and a state array with three values, `EMPTY`, `FULL` and `DELETED`.

Listing 16.26 `p1609_linear_probing.cpp`: open addressing with linear probing and tombstones, tracing every probe sequence.

```
// p1609_linear_probing.cpp - Problem 16.9: open addressing, and deleting safely.
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class ProbingTable {
    enum State { EMPTY, FULL, DELETED };                // DELETED = a "tombstone"
    vector<int> key;
    vector<State> state;
    int m;

    int h(int k) const { return k % m; }                // keys are non-negative

public:
    explicit ProbingTable(int size) : key(size, 0), state(size, EMPTY), m(size) {}

    // 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
    }

    // Returns the slot used, or -1 if k is already present or the table is full.
    int insert(int k, string& trace) {
        int firstFree = -1;
        for (int j = 0; j < m; j++) {
            int i = (h(k) + j) % m;
            trace += " " + to_string(i);
            if (state[i] == FULL) {
                if (key[i] == k) return -1;             // already present
                continue;
            }
            if (firstFree == -1) firstFree = i;         // remember the first hole
            if (state[i] == EMPTY) break;               // k is certainly absent
        }
        if (firstFree == -1) return -1;                 // full
        key[firstFree] = k;
        state[firstFree] = FULL;
        return firstFree;
```

Listing 16.26 (Continued) *p1609_linear_probing.cpp: open addressing with linear probing and tombstones, tracing every probe sequence.*

```
    }

    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;
    }

    void print() const {
        cout << "  slot:";
        for (int i = 0; i < m; i++) cout << setw(4) << i;
        cout << "\n  key: ";
        for (int i = 0; i < m; i++) {
            if (state[i] == FULL) cout << setw(4) << key[i];
            else cout << setw(4) << (state[i] == EMPTY ? "." : "del");
        }
        cout << "\n";
    }
};

int main() {
    ProbingTable t(11);
    cout << "h(k) = k mod 11, linear probing\n";
    for (int k : {10, 22, 31, 4, 15, 28, 17, 88, 59}) {
        string trace;
        int slot = t.insert(k, trace);
        cout << "insert " << setw(2) << k << ": probes" << trace << " -> slot "
             << slot << "\n";
    }
    t.print();

    cout << "\nremove 15: " << boolalpha << t.remove(15) << "\n";
    t.print();
    for (int k : {59, 26}) {
        string trace;
        int slot = t.search(k, trace);
        cout << "search " << k << ": probes" << trace << " -> "
             << (slot == -1 ? string("not found") : "slot " + to_string(slot))
             << "\n";
    }
    string trace;
    int slot = t.insert(37, trace);
    cout << "insert 37: probes" << trace << " -> slot " << slot
         << " (the tombstone)\n";
```

Listing 16.26 (Continued) *p1609_linear_probing.cpp: open addressing with linear probing and tombstones, tracing every probe sequence.*

```
    t.print();
    return 0;
}
```

Line by line. `search` stops at `EMPTY` and ignores `DELETED`. `insert` remembers the first non-`FULL` slot in `firstFree` but continues until an `EMPTY` slot proves the key absent, and gives up after m probes if the table is full. `remove` sets the state to `DELETED`, never to `EMPTY`; the comment says why.

The Java and Python solutions. **Listing 16.27** and **Listing 16.28** print the same lines; Java uses an `enum` for the three states.

Listing 16.27 `P1609LinearProbing.java`: Problem 16.9 in Java.

```
// P1609LinearProbing.java - Problem 16.9: open addressing, and deleting safely.
public class P1609LinearProbing {

    enum State { EMPTY, FULL, DELETED }                  // DELETED = a "tombstone"

    static class ProbingTable {
        final int[] key;
        final State[] state;
        final int m;

        ProbingTable(int size) {
            m = size;
            key = new int[m];
            state = new State[m];
            java.util.Arrays.fill(state, State.EMPTY);
        }

        int h(int k) {
            return k % m;                                // keys are non-negative
        }

        // Returns the slot of k, or -1. A tombstone does NOT stop the search.
        int search(int k, StringBuilder trace) {
            for (int j = 0; j < m; j++) {
                int i = (h(k) + j) % m;
                trace.append(" ").append(i);
                if (state[i] == State.EMPTY) return -1;  // k would have been here
                if (state[i] == State.FULL && key[i] == k) return i;
            }
            return -1;                                   // looked at every slot
        }

        // Returns the slot used, or -1 if k is present or the table is full.
        int insert(int k, StringBuilder trace) {
            int firstFree = -1;
            for (int j = 0; j < m; j++) {
                int i = (h(k) + j) % m;
                trace.append(" ").append(i);
                if (state[i] == State.FULL) {
                    if (key[i] == k) return -1;          // already present
                    continue;
                }
                if (firstFree == -1) firstFree = i;      // remember the first hole
                if (state[i] == State.EMPTY) break;      // k is certainly absent
            }
            if (firstFree == -1) return -1;              // full
```

Listing 16.27 (Continued) *P1609LinearProbing.java: Problem 16.9 in Java.*

```
            key[firstFree] = k;
            state[firstFree] = State.FULL;
            return firstFree;
        }

        boolean remove(int k) {
            int i = search(k, new StringBuilder());
            if (i == -1) return false;
            state[i] = State.DELETED;             // NOT EMPTY: keep the runs whole
            return true;
        }

        void print() {
            StringBuilder a = new StringBuilder("  slot:");
            StringBuilder b = new StringBuilder("  key: ");
            for (int i = 0; i < m; i++) {
                a.append(String.format("%4d", i));
                if (state[i] == State.FULL) b.append(String.format("%4d", key[i]));
                else b.append(String.format("%4s",
                                            state[i] == State.EMPTY ? "." : "del"));
            }
            System.out.println(a + "\n" + b);
        }
    }

    public static void main(String[] args) {
        ProbingTable t = new ProbingTable(11);
        System.out.println("h(k) = k mod 11, linear probing");
        for (int k : new int[] {10, 22, 31, 4, 15, 28, 17, 88, 59}) {
            StringBuilder trace = new StringBuilder();
            int slot = t.insert(k, trace);
            System.out.printf("insert %2d: probes%s -> slot %d%n", k, trace, slot);
        }
        t.print();

        System.out.println("\nremove 15: " + t.remove(15));
        t.print();
        for (int k : new int[] {59, 26}) {
            StringBuilder trace = new StringBuilder();
            int slot = t.search(k, trace);
            System.out.println("search " + k + ": probes" + trace + " -> "
                    + (slot == -1 ? "not found" : "slot " + slot));
        }
        StringBuilder trace = new StringBuilder();
        int slot = t.insert(37, trace);
        System.out.println("insert 37: probes" + trace + " -> slot " + slot
```

Listing 16.27 (Continued) *P1609LinearProbing.java: Problem 16.9 in Java.*

```
                + " (the tombstone)");
        t.print();
    }
}
```

Listing 16.28 `p1609_linear_probing.py`: Problem 16.9 in Python.

```
# p1609_linear_probing.py - Problem 16.9: open addressing, and deleting safely.
EMPTY, FULL, DELETED = 0, 1, 2              # DELETED = a "tombstone"


class ProbingTable:
    def __init__(self, m):
        self.m = m
        self.key = [0] * m
        self.state = [EMPTY] * m

    def h(self, k):
        return k % self.m                   # keys are non-negative

    def search(self, k, trace):
        """Returns the slot of k, or -1. A tombstone does NOT stop the search."""
        for j in range(self.m):
            i = (self.h(k) + j) % self.m
            trace.append(i)
            if self.state[i] == EMPTY:      # k would have been here
                return -1
            if self.state[i] == FULL and self.key[i] == k:
                return i
        return -1                           # looked at every slot

    def insert(self, k, trace):
        """Returns the slot used, or -1 if k is present or the table is full."""
        first_free = -1
        for j in range(self.m):
            i = (self.h(k) + j) % self.m
            trace.append(i)
            if self.state[i] == FULL:
                if self.key[i] == k:        # already present
                    return -1
                continue
            if first_free == -1:            # remember the first hole
                first_free = i
            if self.state[i] == EMPTY:      # k is certainly absent
                break
        if first_free == -1:                # full
            return -1
        self.key[first_free] = k
        self.state[first_free] = FULL
        return first_free
```

Listing 16.28 (Continued) *p1609_linear_probing.py: Problem 16.9 in Python.*

```
    def remove(self, k):
        i = self.search(k, [])
        if i == -1:
            return False
        self.state[i] = DELETED             # NOT EMPTY: keep chains whole
        return True

    def print(self):
        print("  slot:" + "".join(f"{i:4d}" for i in range(self.m)))
        cells = [f"{self.key[i]:4d}" if self.state[i] == FULL else
                 f"{'.' if self.state[i] == EMPTY else 'del':>4}"
                 for i in range(self.m)]
        print("  key: " + "".join(cells))


def probes(trace):
    return "".join(f" {i}" for i in trace)


def main():
    t = ProbingTable(11)
    print("h(k) = k mod 11, linear probing")
    for k in (10, 22, 31, 4, 15, 28, 17, 88, 59):
        trace = []
        slot = t.insert(k, trace)
        print(f"insert {k:2d}: probes{probes(trace)} -> slot {slot}")
    t.print()

    print("\nremove 15:", str(t.remove(15)).lower())
    t.print()
    for k in (59, 26):
        trace = []
        slot = t.search(k, trace)
        print(f"search {k}: probes{probes(trace)} -> "
              + ("not found" if slot == -1 else f"slot {slot}"))
    trace = []
    slot = t.insert(37, trace)
    print(f"insert 37: probes{probes(trace)} -> slot {slot} (the tombstone)")
    t.print()


if __name__ == "__main__":
    main()
```

A tempting wrong solution. Deleting by emptying the slot is the natural first idea, and it seems to work, because the deleted key is indeed no longer found. `wrong/p1609_wrong.cpp` removes 15 that way and then looks for 59:

```
before: search 59 -> slot 8
after removing 15: search 59 -> slot -1
```

59 has not been deleted; it is still in slot 8. But its search starts at 4, finds 4, moves to 5, sees an empty slot, and concludes by the stopping rule that 59 was never inserted. The table now *loses* a key that it holds, which in a library catalog means a book on the shelf that the computer swears is not there. The same mistake cannot happen with chaining, where removing a node from a chain affects no other key; that is one of the reasons Chapter 15 compares the two strategies.

> **CHECK YOURSELF.** Starting from the table after the nine insertions, remove 17 (with a tombstone) and then insert 50 (50 mod 11 = 6). Which slots does the insertion probe, and where does 50 go? Would the answer change if 17 had been removed by emptying its slot?

Answer. 50 probes 6 (28), 7 (tombstone, remembered), 8 (59), 9 (31), 10 (10), 0 (22), 1 (88), 2 (empty: 50 is certainly absent) and goes into slot 7, the remembered tombstone. With the slot emptied instead, 50 would stop at 7 at once and go there, which happens to be the same place, but the table would be broken for any key whose path ran through 7: 59 would still be found (its path 4, 5, 6, 7 now reaches 50 at 7 and continues to 8), yet a later removal of 50 by emptying would lose 59 again. Tombstones make every sequence of operations safe, not just this one.

16.8 Choosing the Right Data Structure

Every chapter of this book answered a question of the form "how does this structure work?". The question a working programmer meets is the reverse: *here is a task; which structure?* This section answers it twice, first with a table that condenses the whole book, and then with measurements of every structure on the same real task, so that the table is not a matter of faith.

16.8.1 Start from the operations, not from the structure

A data structure is chosen by the operations the task performs most often, and by how often it performs each. The same catalog of ten thousand books needs a different structure if the librarian mostly looks up a book by its code, mostly lists the books between two codes, mostly asks for the ten most popular, or mostly follows "people who wrote with this person". So the first step is always Step 2 of the method: write the one sentence that names the dominant operation. **Table 16.4** then turns the sentence into a structure.

Table 16.4 Choosing a data structure by the operation the task performs most often: the whole book on one page. n is the number of items, r the number of results of a range query.

The task mostly needs to…	Choose	Cost of that operation	Watch out for	Ch.
read or write item number i	array / `vector`	$O(1)$	insertion in the middle is $O(n)$	1
find an item, few times, no preparation	unsorted array + sequential search	$O(n)$	fine for small n or a single search	2
find by key, and also list keys in order or in a range, with rare changes	sorted array + binary search	$O(\lg n)$ search, $O(\lg n + r)$ for r results	each insertion shifts $O(n)$ items	2, 3
add and remove only at the most recent end (undo, brackets, DFS)	stack	$O(1)$	popping an empty stack	5
serve in order of arrival (waiting lines, BFS)	queue, or deque for both ends	$O(1)$	a linear array drifts; use a circular one	5, 9
insert or delete in the middle at a node you already hold	linked list	$O(1)$	finding the node is still $O(n)$; poor cache locality	7, 9
find by key, and also the smallest, largest, predecessor, successor, or a range, with frequent changes	balanced BST (`std::map`, `TreeMap`)	$O(\lg n)$	an unbalanced BST degrades to $O(n)$ on sorted input	11
repeatedly take the largest or smallest item while items keep arriving	heap / priority queue	$O(\lg n)$ insert and extract, $O(1)$ peek	a heap cannot search; C++ default is a max-heap	14
the k largest of many, or a running median	min-heap of size k; two heaps	$O(n \lg k)$; $O(\lg n)$ per item	the direction of the heap (**Table 16.3**)	14
find, insert or delete by key, nothing else	hash table (`unordered_map`, `HashMap`, `dict`)	$O(1)$ on average	no order, no ranges; bad hash or full table degrade to $O(n)$	15
model connections: routes, dependencies, "who is linked to whom"	graph with adjacency lists	BFS/DFS $O(\|V\| + \|E\|)$	an adjacency matrix costs $\|V\|^2$ memory	13
fewest steps / an order respecting dependencies / quickest route	BFS / Kahn / Dijkstra with a heap	$O(\|V\| + \|E\|)$ / same / $O((\|V\| + \|E\|)\lg\|V\|)$	DFS depth is not a distance; negative weights break Dijkstra	13, 14

Figure 16.15 draws the same reasoning as a decision tree. It asks first *what you will do most often*, and only then one or two follow-up questions. Its most important branch is the one in the middle: "find by key" leads to a hash table only if you *never* need order; the moment the task also needs "the smallest", "the next one" or "everything between", the answer becomes a search tree or, if the data rarely changes, a sorted array.

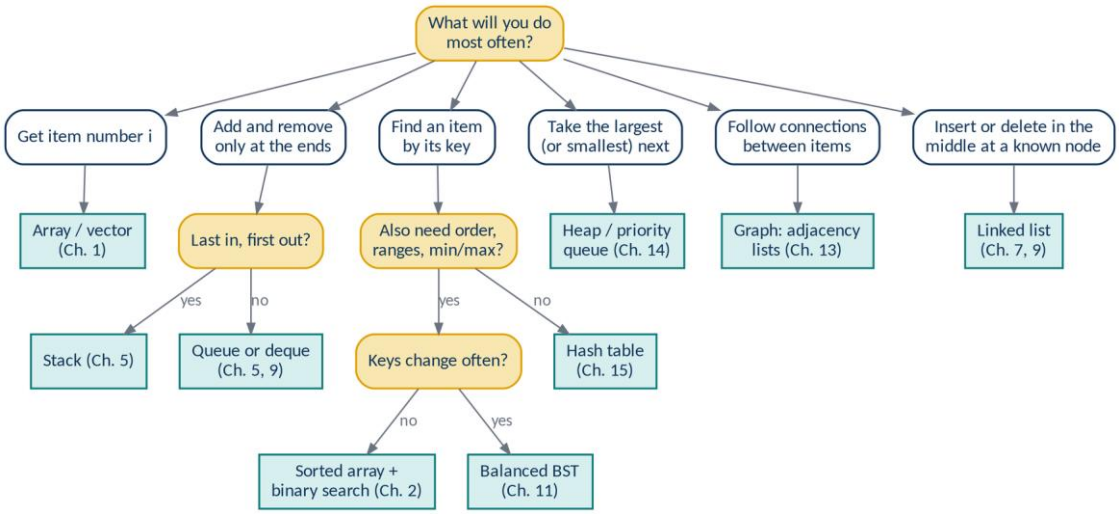


Figure 16.15 Choosing a structure: begin with the operation the task performs most often, then answer at most two follow-up questions. The chapter that teaches each structure is in brackets.

Two cautions keep the table honest. First, the costs are *growth rates*, and for small n constant factors decide; a sequential search of twenty items is often faster than hashing them. Second, real tasks rarely have a single operation, and real programs combine structures, as Problem 16.2 combined two hash maps with a walk along a chain, and Problem 16.5 combined adjacency lists with a heap. The table tells you which structure should carry the *dominant* operation.

16.8.2 The experiment

Pustaka Ceria's catalog of ten thousand books (Chapter 2, from the goodbooks-10k data set [13]) is a realistic test bed: real identifiers, which run from 1 to over 33 million and are far from evenly spread, real ratings counts, and real names of the people credited on each book. **Listing 16.29** loads it and measures six structures on four operations.

- **Build**: store the 10,000 identifiers, in the order of the file, which is the order of popularity.
- **Lookup**: 2,000 identifiers, 1,000 of them present (every tenth book) and 1,000 absent (the next free number after every tenth book, offset by five).
- **Range**: 200 questions "how many books have an identifier in $[a, a + 50{,}000)$?", with a spread over the whole range.
- **Insert**: 1,000 new books (the 1,000 absent identifiers) into a fresh copy of the structure.

The six structures are the unsorted array (sequential search), the sorted array (binary search, `lower_bound` for ranges), the singly linked list, the unbalanced binary search tree of Chapter 11 with the keys inserted in file order, the balanced search tree of the library, and the hash table of the library. Every structure must find the same 1,000 books and count the same number of books in the ranges; the program checks that. Each measurement is the best of five runs, to reduce the noise of other programs on the computer. Two further experiments follow: the ten most-rated books with a heap of size 10 against a full sort, and a graph of the people credited on the same book.

Listing 16.29 `p1610_choose.cpp`: six structures measured on the same catalog (build, lookup, range query, insert), the ten most-rated books by heap and by sorting, and the graph of people credited on the same book.

```cpp
// p1610_choose.cpp - one catalog, every structure of the book, measured.
#include <algorithm>
#include <chrono>
#include <cstdio>
#include <forward_list>
#include <fstream>
#include <functional>
#include <map>
#include <queue>
#include <sstream>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>
using namespace std;

struct Book {
    long long id, ratings;
    int year;
    string title, authors;
};

// ------------- the binary search tree of Chapter 11 (not balanced) -------------
struct BST {
    struct Node {
        long long key;
        const Book* book;
        Node *left = nullptr, *right = nullptr;
    };
    Node* root = nullptr;
    size_t count = 0;
    BST() = default;
    BST(const BST&) = delete;
    BST& operator=(const BST&) = delete;
    ~BST() { destroy(root); }
    static void destroy(Node* p) {
        if (!p) return;
        destroy(p->left);
        destroy(p->right);
        delete p;
    }
    void insert(long long key, const Book* b) {
        Node** link = &root;
        while (*link) link = key < (*link)->key ? &(*link)->left : &(*link)->right;
        *link = new Node{key, b};
        count++;
```

Listing 16.29 (Continued) *p1610_choose.cpp: six structures measured on the same catalog (build, lookup, range query, insert), the ten most-rated books by heap and by sorting, and the graph of people credited on the same book.*

```
    }
    const Book* find(long long key) const {
        Node* p = root;
        while (p && p->key != key) p = key < p->key ? p->left : p->right;
        return p ? p->book : nullptr;
    }
    static long long countRange(const Node* p, long long lo, long long hi) {
        if (!p) return 0;
        if (p->key < lo) return countRange(p->right, lo, hi);     // skip left side
        if (p->key >= hi) return countRange(p->left, lo, hi);     // skip right side
        return 1 + countRange(p->left, lo, hi) + countRange(p->right, lo, hi);
    }
    static int height(const Node* p) {
        return p ? 1 + max(height(p->left), height(p->right)) : 0;
    }
};

// Runs f several times and returns the best time in milliseconds.
double bestMs(int reps, const function<void()>& f) {
    double best = 1e300;
    for (int r = 0; r < reps; r++) {
        auto t0 = chrono::steady_clock::now();
        f();
        auto t1 = chrono::steady_clock::now();
        best = min(best, chrono::duration<double, milli>(t1 - t0).count());
    }
    return best;
}

// Like bestMs, but make() builds a fresh structure first, and only op() is timed.
template <class S>
double bestMsAfter(int reps, const function<S()>& make,
                   const function<void(S&)>& op) {
    double best = 1e300;
    for (int r = 0; r < reps; r++) {
        S c = make();
        auto t0 = chrono::steady_clock::now();
        op(c);
        auto t1 = chrono::steady_clock::now();
        best = min(best, chrono::duration<double, milli>(t1 - t0).count());
    }
    return best;
}

volatile long long sink;          // a result the optimizer cannot throw away
```

Listing 16.29 (Continued) *p1610_choose.cpp: six structures measured on the same catalog (build, lookup, range query, insert), the ten most-rated books by heap and by sorting, and the graph of people credited on the same book.*

```
int main() {
    ifstream in("../data/catalog.tsv");
    if (!in) {
        printf("cannot open ../data/catalog.tsv\n");
        return 1;
    }
    vector<Book> books;
    string line;
    getline(in, line);
    while (getline(in, line)) {
        stringstream ss(line);
        string f[5];
        for (auto& x : f) getline(ss, x, '\t');
        books.push_back({stoll(f[0]), stoll(f[2]), stoi(f[1]), f[3], f[4]});
    }
    size_t n = books.size();

    // 1,000 identifiers that are present and 1,000 that are absent.
    unordered_set<long long> present;
    for (const Book& b : books) present.insert(b.id);
    vector<long long> queries, fresh;
    for (size_t i = 0; i < n; i += 10) queries.push_back(books[i].id);
    for (size_t i = 5; i < n; i += 10) {
        long long q = books[i].id + 1;
        while (present.count(q)) q++;
        queries.push_back(q);
        fresh.push_back(q);                 // also used as the 1,000 new books
    }
    vector<pair<long long, long long>> ranges;       // 200 ranges [lo, lo + 50,000)
    for (long long i = 0; i < 200; i++)
        ranges.push_back({i * 150000, i * 150000 + 50000});

    printf("Catalog: %zu books; %zu lookups (half absent); %zu range queries; "
           "%zu inserts\n\n", n, queries.size(), ranges.size(), fresh.size());
    printf("%-28s%10s%12s%12s%12s\n", "structure", "build ms", "lookup us",
           "range us", "insert us");
    const int R = 5;
    vector<long long> answers;              // all structures must give the same
    auto row = [&](const char* name, double b, double l, double r, double ins,
                   long long found, long long inRange) {
        printf("%-28s%10.3f%12.3f%12.3f%12.3f\n", name, b,
               1000 * l / queries.size(), 1000 * r / ranges.size(),
               1000 * ins / fresh.size());
        answers.push_back(found);
        answers.push_back(inRange);
    };
```

Listing 16.29 (Continued) *p1610_choose.cpp: six structures measured on the same catalog (build, lookup, range query, insert), the ten most-rated books by heap and by sorting, and the graph of people credited on the same book.*

```
    {   // unsorted array: append in arrival order, search by walking
        vector<long long> a;
        double tb = bestMs(R, [&] {
            a.clear();
            for (const Book& b : books) a.push_back(b.id);
        });
        long long found = 0, cnt = 0;
        double tl = bestMs(R, [&] {
            found = 0;
            for (long long q : queries)
                for (long long x : a)
                    if (x == q) { found++; break; }
        });
        double tr = bestMs(R, [&] {
            cnt = 0;
            for (auto [lo, hi] : ranges)
                for (long long x : a) cnt += (x >= lo && x < hi);
        });
        double ti = bestMsAfter<vector<long long>>(R, [&] { return a; },
            [&](vector<long long>& c) {
                for (long long q : fresh) c.push_back(q);
            });
        row("unsorted array", tb, tl, tr, ti, found, cnt);
    }
    {   // sorted array: sort once, binary search (Chapter 2)
        vector<long long> a;
        double tb = bestMs(R, [&] {
            a.clear();
            for (const Book& b : books) a.push_back(b.id);
            sort(a.begin(), a.end());
        });
        long long found = 0, cnt = 0;
        double tl = bestMs(R, [&] {
            found = 0;
            for (long long q : queries)
                found += binary_search(a.begin(), a.end(), q);
        });
        double tr = bestMs(R, [&] {
            cnt = 0;
            for (auto [lo, hi] : ranges)
                cnt += lower_bound(a.begin(), a.end(), hi) -
                       lower_bound(a.begin(), a.end(), lo);
        });
        double ti = bestMsAfter<vector<long long>>(R, [&] { return a; },
            [&](vector<long long>& c) {
```

Listing 16.29 (Continued) *p1610_choose.cpp: six structures measured on the same catalog (build, lookup, range query, insert), the ten most-rated books by heap and by sorting, and the graph of people credited on the same book.*

```
                for (long long q : fresh)
                    c.insert(upper_bound(c.begin(), c.end(), q), q);
            });
        row("sorted array + binary search", tb, tl, tr, ti, found, cnt);
    }
    {   // singly linked list: push at the front (Chapter 7)
        forward_list<long long> L;
        double tb = bestMs(R, [&] {
            L.clear();
            for (const Book& b : books) L.push_front(b.id);
        });
        long long found = 0, cnt = 0;
        double tl = bestMs(R, [&] {
            found = 0;
            for (long long q : queries)
                for (long long x : L)
                    if (x == q) { found++; break; }
        });
        double tr = bestMs(R, [&] {
            cnt = 0;
            for (auto [lo, hi] : ranges)
                for (long long x : L) cnt += (x >= lo && x < hi);
        });
        double ti = bestMsAfter<forward_list<long long>>(R, [&] { return L; },
            [&](forward_list<long long>& c) {
                for (long long q : fresh) c.push_front(q);
            });
        row("singly linked list", tb, tl, tr, ti, found, cnt);
    }
    int bstHeight = 0;
    {   // binary search tree of Chapter 11, keys inserted in arrival order
        double tb = bestMs(R, [&] {
            BST t;
            for (const Book& b : books) t.insert(b.id, &b);
            sink = t.count;
        });
        BST t;
        for (const Book& b : books) t.insert(b.id, &b);
        bstHeight = BST::height(t.root);
        long long found = 0, cnt = 0;
        double tl = bestMs(R, [&] {
            found = 0;
            for (long long q : queries) found += t.find(q) != nullptr;
        });
        double tr = bestMs(R, [&] {
            cnt = 0;
```

Listing 16.29 (Continued) *p1610_choose.cpp: six structures measured on the same catalog (build, lookup, range query, insert), the ten most-rated books by heap and by sorting, and the graph of people credited on the same book.*

```
            for (auto [lo, hi] : ranges) cnt += BST::countRange(t.root, lo, hi);
        });
        double ti = 1e300;                  // a BST is not copied: rebuild, untimed
        for (int r = 0; r < R; r++) {
            BST c;
            for (const Book& b : books) c.insert(b.id, &b);
            ti = min(ti, bestMs(1, [&] {
                for (long long q : fresh) c.insert(q, nullptr);
            }));
        }
        row("BST (Chapter 11, unbalanced)", tb, tl, tr, ti, found, cnt);
    }
    {   // balanced search tree: std::map is a red-black tree
        map<long long, const Book*> t;
        double tb = bestMs(R, [&] {
            t.clear();
            for (const Book& b : books) t.emplace(b.id, &b);
        });
        long long found = 0, cnt = 0;
        double tl = bestMs(R, [&] {
            found = 0;
            for (long long q : queries) found += t.count(q);
        });
        double tr = bestMs(R, [&] {
            cnt = 0;
            for (auto [lo, hi] : ranges)
                cnt += distance(t.lower_bound(lo), t.lower_bound(hi));
        });
        using M = map<long long, const Book*>;
        double ti = bestMsAfter<M>(R, [&] { return t; },
            [&](M& c) { for (long long q : fresh) c.emplace(q, nullptr); });
        row("balanced BST (std::map)", tb, tl, tr, ti, found, cnt);
    }
    {   // hash table (Chapter 15): std::unordered_map
        unordered_map<long long, const Book*> h;
        double tb = bestMs(R, [&] {
            h.clear();
            for (const Book& b : books) h.emplace(b.id, &b);
        });
        long long found = 0, cnt = 0;
        double tl = bestMs(R, [&] {
            found = 0;
            for (long long q : queries) found += h.count(q);
        });
        double tr = bestMs(R, [&] {         // no order: every range scans it all
            cnt = 0;
```

Listing 16.29 (Continued) *p1610_choose.cpp: six structures measured on the same catalog (build, lookup, range query, insert), the ten most-rated books by heap and by sorting, and the graph of people credited on the same book.*

```
            for (auto [lo, hi] : ranges)
                for (const auto& kv : h) cnt += (kv.first >= lo && kv.first < hi);
        });
        using H = unordered_map<long long, const Book*>;
        double ti = bestMsAfter<H>(R, [&] { return h; },
            [&](H& c) { for (long long q : fresh) c.emplace(q, nullptr); });
        row("hash table (unordered_map)", tb, tl, tr, ti, found, cnt);
    }
    bool agree = true;
    for (size_t i = 2; i < answers.size(); i++)
        agree = agree && answers[i] == answers[i % 2];
    printf("All %zu structures found %lld of the %zu lookups and counted "
           "%lld books\nin the ranges: %s\n", answers.size() / 2, answers[0],
           queries.size(), answers[1], agree ? "yes" : "no");
    printf("The BST built in arrival order has %d levels; a balanced one has 14.\n",
           bstHeight);

    // ------------- the 10 most-rated books: heap against sorting -------------
    vector<long long> top1, top2;
    double th = bestMs(R, [&] {             // min-heap of size 10 (Chapter 14)
        priority_queue<long long, vector<long long>, greater<long long>> heap;
        for (const Book& b : books) {
            if (heap.size() < 10) heap.push(b.ratings);
            else if (b.ratings > heap.top()) { heap.pop(); heap.push(b.ratings); }
        }
        top1.clear();
        while (!heap.empty()) { top1.push_back(heap.top()); heap.pop(); }
        reverse(top1.begin(), top1.end());
    });
    double ts = bestMs(R, [&] {             // sort all 10,000 (Chapter 3)
        vector<long long> r;
        for (const Book& b : books) r.push_back(b.ratings);
        sort(r.rbegin(), r.rend());
        top2.assign(r.begin(), r.begin() + 10);
    });
    printf("\nTop 10 by ratings: heap of size 10 %.3f ms, sorting everything "
           "%.3f ms\n  (same answer: %s)\n", th, ts, top1 == top2 ? "yes" : "no");
```

Listing 16.29 (Continued) *p1610_choose.cpp: six structures measured on the same catalog (build, lookup, range query, insert), the ten most-rated books by heap and by sorting, and the graph of people credited on the same book.*

```
    // ------------- the graph of people credited on the same book -------------
    unordered_map<string, int> idOf;        // name -> vertex number (hashing again)
    vector<string> name;
    vector<vector<int>> adj;
    auto t0 = chrono::steady_clock::now();
    for (const Book& b : books) {
        vector<int> who;
        stringstream ss(b.authors);
        string a;
        while (getline(ss, a, ',')) {
            if (!a.empty() && a[0] == ' ') a.erase(0, 1);
            auto [it, isNew] = idOf.emplace(a, (int)name.size());
            if (isNew) {
                name.push_back(a);
                adj.emplace_back();
            }
            if (find(who.begin(), who.end(), it->second) == who.end())
                who.push_back(it->second);
        }
        for (int u : who)
            for (int v : who)
                if (u != v && find(adj[u].begin(), adj[u].end(), v) == adj[u].end())
                    adj[u].push_back(v);
    }
    size_t V = name.size(), E = 0;
    for (auto& a : adj) E += a.size();
    E /= 2;
    vector<int> comp(V, -1);                // label every connected component (BFS)
    int comps = 0;
    size_t largest = 0;
    for (size_t s = 0; s < V; s++) {
        if (comp[s] != -1) continue;
        size_t size = 0;
        queue<int> q;
        q.push(s);
        comp[s] = comps;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            size++;
            for (int v : adj[u])
                if (comp[v] == -1) { comp[v] = comps; q.push(v); }
        }
        largest = max(largest, size);
        comps++;
    }
```

Listing 16.29 (Continued) *p1610_choose.cpp: six structures measured on the same catalog (build, lookup, range query, insert), the ten most-rated books by heap and by sorting, and the graph of people credited on the same book.*

```
    auto t1 = chrono::steady_clock::now();
    double tg = chrono::duration<double, milli>(t1 - t0).count();
    printf("\nPeople graph: %zu people, %zu co-credit edges, %d components,\n"
           "  the largest has %zu people\n", V, E, comps, largest);
    printf("  adjacency lists hold %zu entries; an adjacency matrix would need\n"
           "  %zu cells; built and explored in %.3f ms\n", 2 * E, V * V, tg);

    int s = idOf.at("Stephen King"), t = idOf.at("Charles Dickens");
    vector<int> parent(V, -2);              // BFS from s, remembering parents
    queue<int> q;
    q.push(s);
    parent[s] = -1;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : adj[u])
            if (parent[v] == -2) { parent[v] = u; q.push(v); }
    }
    vector<string> path;
    for (int v = t; v != -1; v = parent[v]) path.push_back(name[v]);
    reverse(path.begin(), path.end());
    printf("  fewest hops from Stephen King to Charles Dickens: %zu\n   ",
           path.size() - 1);
    for (size_t i = 0; i < path.size(); i++)   // three names to a line
        printf(" %s%s", path[i].c_str(),
               i + 1 == path.size() ? "\n" : (i % 3 == 2 ? " ->\n   " : " ->"));
    return 0;
}
```

```
Catalog: 10000 books; 2000 lookups (half absent); 200 range queries; 1000 inserts

structure                     build ms   lookup us    range us   insert us
unsorted array                   0.023       2.280       6.021       0.004
sorted array + binary search     0.481       0.073       0.048       0.634
singly linked list               0.136      13.652      18.038       0.007
BST (Chapter 11, unbalanced)     1.262       0.053       0.168       0.131
balanced BST (std::map)          1.459       0.088       0.252       0.124
hash table (unordered_map)       0.314       0.006      44.913       0.089
All 6 structures found 1000 of the 2000 lookups and counted 4702 books
in the ranges: yes
The BST built in arrival order has 35 levels; a balanced one has 14.

Top 10 by ratings: heap of size 10 0.006 ms, sorting everything 0.420 ms
  (same answer: yes)

People graph: 5841 people, 6997 co-credit edges, 3408 components,
  the largest has 605 people
  adjacency lists hold 13994 entries; an adjacency matrix would need
  34117281 cells; built and explored in 3.578 ms
  fewest hops from Stephen King to Charles Dickens: 5
    Stephen King -> Dave McKean -> Richard Dawkins ->
    Thomas Hardy -> Patricia Ingham -> Charles Dickens
```

Listing 16.30 and **Listing 16.31** run the same experiment in Java and Python. The Java version stores the identifiers of its two arrays in a plain `long[]` (a `LongArray` of our own), because an `ArrayList<Long>` would hold ten thousand separate objects and measure Java's object overhead rather than the array. Python has no balanced search tree in its standard library, so that row is marked as missing.

Listing 16.30 `P1610Choose.java`: the experiment of Section 16.8 in Java, with `TreeMap` as the balanced search tree and `HashMap` as the hash table.

```java
// P1610Choose.java - one catalog, every structure of the book, measured.
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.TreeMap;
import java.util.function.Supplier;

public class P1610Choose {

    record Book(long id, int year, long ratings, String title, String authors) {}

    // A growable array of long values: the array holds numbers, not objects.
    static class LongArray {
        long[] a = new long[16];
        int n = 0;

        void add(long x) {
            if (n == a.length) a = Arrays.copyOf(a, 2 * n);
            a[n++] = x;
        }

        void insertAt(int i, long x) {          // shift a[i..n-1] one place right
            if (n == a.length) a = Arrays.copyOf(a, 2 * n);
            System.arraycopy(a, i, a, i + 1, n - i);
            a[i] = x;
            n++;
        }

        LongArray copy() {
            LongArray c = new LongArray();
            c.a = Arrays.copyOf(a, a.length);
            c.n = n;
            return c;
        }
    }
```

Listing 16.30 (Continued) *P1610Choose.java: the experiment of Section 16.8 in Java, with TreeMap as the balanced search tree and HashMap as the hash table.*

```
    // First index i with a[i] >= key (Chapter 2's lower bound).
    static int lowerBound(LongArray s, long key) {
        int lo = 0, hi = s.n;
        while (lo < hi) {
            int mid = (lo + hi) >>> 1;
            if (s.a[mid] < key) lo = mid + 1;
            else hi = mid;
        }
        return lo;
    }

    // ----------- the binary search tree of Chapter 11 (not balanced) -----------
    static class BST {
        static class Node {
            final long key;
            Node left, right;

            Node(long key) {
                this.key = key;
            }
        }

        Node root;

        void insert(long key) {
            if (root == null) {
                root = new Node(key);
                return;
            }
            Node p = root;
            while (true) {
                if (key < p.key) {
                    if (p.left == null) { p.left = new Node(key); return; }
                    p = p.left;
                } else {
                    if (p.right == null) { p.right = new Node(key); return; }
                    p = p.right;
                }
            }
        }
```

Listing 16.30 (Continued) *P1610Choose.java: the experiment of Section 16.8 in Java, with TreeMap as the balanced search tree and HashMap as the hash table.*

```
        boolean contains(long key) {
            Node p = root;
            while (p != null && p.key != key) p = key < p.key ? p.left : p.right;
            return p != null;
        }

        static long countRange(Node p, long lo, long hi) {
            if (p == null) return 0;
            if (p.key < lo) return countRange(p.right, lo, hi);    // skip left side
            if (p.key >= hi) return countRange(p.left, lo, hi);   // skip right side
            return 1 + countRange(p.left, lo, hi) + countRange(p.right, lo, hi);
        }

        static int height(Node p) {
            return p == null ? 0 : 1 + Math.max(height(p.left), height(p.right));
        }
    }

    // Runs f several times and returns the best time in milliseconds.
    static double bestMs(int reps, Runnable f) {
        double best = Double.MAX_VALUE;
        for (int r = 0; r < reps; r++) {
            long t0 = System.nanoTime();
            f.run();
            best = Math.min(best, (System.nanoTime() - t0) / 1e6);
        }
        return best;
    }

    interface Op<S> {
        void run(S s);
    }

    // Like bestMs, but make() builds a fresh structure first, and only op is timed.
    static <S> double bestMsAfter(int reps, Supplier<S> make, Op<S> op) {
        double best = Double.MAX_VALUE;
        for (int r = 0; r < reps; r++) {
            S c = make.get();
            long t0 = System.nanoTime();
            op.run(c);
            best = Math.min(best, (System.nanoTime() - t0) / 1e6);
        }
        return best;
    }
```

Listing 16.30 (Continued) *P1610Choose.java: the experiment of Section 16.8 in Java, with TreeMap as the balanced search tree and HashMap as the hash table.*

```
    static final int R = 5;
    static List<Book> books = new ArrayList<>();
    static List<Long> queries = new ArrayList<>(), fresh = new ArrayList<>();
    static long[][] ranges = new long[200][];
    static List<Long> answers = new ArrayList<>();
    static long found, cnt;

    static void row(String name, double b, double l, double r, double ins) {
        System.out.printf("%-28s%10.3f%12.3f%12.3f%12.3f%n", name, b,
                1000 * l / queries.size(), 1000 * r / ranges.length,
                1000 * ins / fresh.size());
        answers.add(found);
        answers.add(cnt);
    }

    public static void main(String[] args) throws IOException {
        List<String> lines = Files.readAllLines(Path.of("../data/catalog.tsv"),
                                                StandardCharsets.UTF_8);
        for (String line : lines.subList(1, lines.size())) {
            String[] f = line.split("\t", -1);
            books.add(new Book(Long.parseLong(f[0]), Integer.parseInt(f[1]),
                               Long.parseLong(f[2]), f[3], f[4]));
        }
        int n = books.size();

        // 1,000 identifiers that are present and 1,000 that are absent.
        HashSet<Long> present = new HashSet<>();
        for (Book b : books) present.add(b.id());
        for (int i = 0; i < n; i += 10) queries.add(books.get(i).id());
        for (int i = 5; i < n; i += 10) {
            long q = books.get(i).id() + 1;
            while (present.contains(q)) q++;
            queries.add(q);
            fresh.add(q);                       // also used as the 1,000 new books
        }
        for (int i = 0; i < 200; i++)
            ranges[i] = new long[] {i * 150000L, i * 150000L + 50000};

        System.out.printf("Catalog: %d books; %d lookups (half absent); %d range "
                + "queries; %d inserts%n%n", n, queries.size(), ranges.length,
                fresh.size());
        System.out.printf("%-28s%10s%12s%12s%12s%n", "structure", "build ms",
                "lookup us", "range us", "insert us");
```

Listing 16.30 (Continued) *P1610Choose.java: the experiment of Section 16.8 in Java, with TreeMap as the balanced search tree and HashMap as the hash table.*

```
        {   // unsorted array: append in arrival order, search by walking
            LongArray[] a = {null};
            double tb = bestMs(R, () -> {
                a[0] = new LongArray();
                for (Book b : books) a[0].add(b.id());
            });
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries)
                    for (int i = 0; i < a[0].n; i++)
                        if (a[0].a[i] == q) { found++; break; }
            });
            double tr = bestMs(R, () -> {
                cnt = 0;
                for (long[] g : ranges)
                    for (int i = 0; i < a[0].n; i++)
                        if (a[0].a[i] >= g[0] && a[0].a[i] < g[1]) cnt++;
            });
            double ti = bestMsAfter(R, a[0]::copy, c -> {
                for (long q : fresh) c.add(q);
            });
            row("unsorted array", tb, tl, tr, ti);
        }
        {   // sorted array: sort once, binary search (Chapter 2)
            LongArray[] a = {null};
            double tb = bestMs(R, () -> {
                a[0] = new LongArray();
                for (Book b : books) a[0].add(b.id());
                Arrays.sort(a[0].a, 0, a[0].n);
            });
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries) {
                    int i = lowerBound(a[0], q);
                    if (i < a[0].n && a[0].a[i] == q) found++;
                }
            });
            double tr = bestMs(R, () -> {
                cnt = 0;
                for (long[] g : ranges)
                    cnt += lowerBound(a[0], g[1]) - lowerBound(a[0], g[0]);
            });
            double ti = bestMsAfter(R, a[0]::copy, c -> {
                for (long q : fresh) c.insertAt(lowerBound(c, q), q);
            });
            row("sorted array + binary search", tb, tl, tr, ti);
```

Listing 16.30 (Continued) *P1610Choose.java: the experiment of Section 16.8 in Java, with TreeMap as the balanced search tree and HashMap as the hash table.*

```
        }
        {   // linked list (Chapter 7; java.util.LinkedList is doubly linked)
            List<LinkedList<Long>> L = new ArrayList<>(List.of(new LinkedList<>()));
            double tb = bestMs(R, () -> {
                LinkedList<Long> x = new LinkedList<>();
                for (Book b : books) x.addFirst(b.id());
                L.set(0, x);
            });
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries)
                    for (long x : L.get(0))
                        if (x == q) { found++; break; }
            });
            double tr = bestMs(R, () -> {
                cnt = 0;
                for (long[] g : ranges)
                    for (long x : L.get(0))
                        if (x >= g[0] && x < g[1]) cnt++;
            });
            double ti = bestMsAfter(R, () -> new LinkedList<>(L.get(0)), c -> {
                for (long q : fresh) c.addFirst(q);
            });
            row("linked list", tb, tl, tr, ti);
        }
        int bstHeight;
        {   // binary search tree of Chapter 11, keys inserted in arrival order
            BST[] t = {null};
            double tb = bestMs(R, () -> {
                t[0] = new BST();
                for (Book b : books) t[0].insert(b.id());
            });
            bstHeight = BST.height(t[0].root);
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries) if (t[0].contains(q)) found++;
            });
            double tr = bestMs(R, () -> {
                cnt = 0;
                for (long[] g : ranges)
                    cnt += BST.countRange(t[0].root, g[0], g[1]);
            });
            double ti = bestMsAfter(R, () -> {
                BST c = new BST();
                for (Book b : books) c.insert(b.id());
                return c;
```

Listing 16.30 (Continued) *P1610Choose.java: the experiment of Section 16.8 in Java, with TreeMap as the balanced search tree and HashMap as the hash table.*

```
            }, c -> {
                for (long q : fresh) c.insert(q);
            });
            row("BST (Chapter 11, unbalanced)", tb, tl, tr, ti);
        }
        {   // balanced search tree: java.util.TreeMap is a red-black tree
            List<TreeMap<Long, Book>> t = new ArrayList<>(List.of(new TreeMap<>()));
            double tb = bestMs(R, () -> {
                TreeMap<Long, Book> x = new TreeMap<>();
                for (Book b : books) x.put(b.id(), b);
                t.set(0, x);
            });
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries) if (t.get(0).containsKey(q)) found++;
            });
            double tr = bestMs(R, () -> {
                cnt = 0;
                for (long[] g : ranges) cnt += t.get(0).subMap(g[0], g[1]).size();
            });
            double ti = bestMsAfter(R, () -> new TreeMap<>(t.get(0)), c -> {
                for (long q : fresh) c.put(q, null);
            });
            row("balanced BST (TreeMap)", tb, tl, tr, ti);
        }
        {   // hash table (Chapter 15): java.util.HashMap
            List<HashMap<Long, Book>> h = new ArrayList<>(List.of(new HashMap<>()));
            double tb = bestMs(R, () -> {
                HashMap<Long, Book> x = new HashMap<>();
                for (Book b : books) x.put(b.id(), b);
                h.set(0, x);
            });
            double tl = bestMs(R, () -> {
                found = 0;
                for (long q : queries) if (h.get(0).containsKey(q)) found++;
            });
            double tr = bestMs(R, () -> {       // no order: each range scans it all
                cnt = 0;
                for (long[] g : ranges)
                    for (long k : h.get(0).keySet())
                        if (k >= g[0] && k < g[1]) cnt++;
            });
            double ti = bestMsAfter(R, () -> new HashMap<>(h.get(0)), c -> {
                for (long q : fresh) c.put(q, null);
            });
            row("hash table (HashMap)", tb, tl, tr, ti);
```

Listing 16.30 (Continued) *P1610Choose.java: the experiment of Section 16.8 in Java, with TreeMap as the balanced search tree and HashMap as the hash table.*

```
        }
        boolean agree = true;
        for (int i = 2; i < answers.size(); i++)
            agree = agree && answers.get(i).equals(answers.get(i % 2));
        System.out.printf("All %d structures found %d of the %d lookups and "
                + "counted %d books%nin the ranges: %s%n", answers.size() / 2,
                answers.get(0), queries.size(), answers.get(1),
                agree ? "yes" : "no");
        System.out.printf("The BST built in arrival order has %d levels; a "
                + "balanced one has 14.%n", bstHeight);

        // ------------- the 10 most-rated books: heap against sorting -------------
        List<Long> top1 = new ArrayList<>(), top2 = new ArrayList<>();
        double th = bestMs(R, () -> {           // min-heap of size 10 (Chapter 14)
            PriorityQueue<Long> heap = new PriorityQueue<>();
            for (Book b : books) {
                if (heap.size() < 10) heap.add(b.ratings());
                else if (b.ratings() > heap.peek()) {
                    heap.poll();
                    heap.add(b.ratings());
                }
            }
            top1.clear();
            while (!heap.isEmpty()) top1.add(heap.poll());
            Collections.reverse(top1);
        });
        double ts = bestMs(R, () -> {           // sort all 10,000 (Chapter 3)
            long[] r = new long[books.size()];
            for (int i = 0; i < r.length; i++) r[i] = books.get(i).ratings();
            Arrays.sort(r);
            top2.clear();
            for (int i = r.length - 1; i >= r.length - 10; i--) top2.add(r[i]);
        });
        System.out.printf("%nTop 10 by ratings: heap of size 10 %.3f ms, sorting "
                + "everything %.3f ms%n  (same answer: %s)%n", th, ts,
                top1.equals(top2) ? "yes" : "no");
```

Listing 16.30 (Continued) *P1610Choose.java: the experiment of Section 16.8 in Java, with TreeMap as the balanced search tree and HashMap as the hash table.*

```
        // ------------- the graph of people credited on the same book -------------
        long t0 = System.nanoTime();
        HashMap<String, Integer> idOf = new HashMap<>();    // name -> vertex number
        List<String> name = new ArrayList<>();
        List<List<Integer>> adj = new ArrayList<>();
        for (Book b : books) {
            List<Integer> who = new ArrayList<>();
            for (String a : b.authors().split(",")) {
                if (a.startsWith(" ")) a = a.substring(1);
                Integer id = idOf.get(a);
                if (id == null) {
                    id = name.size();
                    idOf.put(a, id);
                    name.add(a);
                    adj.add(new ArrayList<>());
                }
                if (!who.contains(id)) who.add(id);
            }
            for (int u : who)
                for (int v : who)
                    if (u != v && !adj.get(u).contains(v)) adj.get(u).add(v);
        }
        int V = name.size();
        long E = 0;
        for (List<Integer> a : adj) E += a.size();
        E /= 2;
        int[] comp = new int[V];               // label each connected component
        Arrays.fill(comp, -1);
        int comps = 0, largest = 0;
        for (int s = 0; s < V; s++) {
            if (comp[s] != -1) continue;
            int size = 0;
            ArrayDeque<Integer> q = new ArrayDeque<>();
            q.add(s);
            comp[s] = comps;
            while (!q.isEmpty()) {
                int u = q.poll();
                size++;
                for (int v : adj.get(u))
                    if (comp[v] == -1) { comp[v] = comps; q.add(v); }
            }
            largest = Math.max(largest, size);
            comps++;
        }
        double tg = (System.nanoTime() - t0) / 1e6;
        System.out.printf("%nPeople graph: %d people, %d co-credit edges, "
```

Listing 16.30 (Continued) *P1610Choose.java: the experiment of Section 16.8 in Java, with TreeMap as the balanced search tree and HashMap as the hash table.*

```
                + "%d components,%n"
                + "  the largest has %d people%n", V, E, comps, largest);
        System.out.printf("  adjacency lists hold %d entries; an adjacency matrix "
                + "would need%n  %d cells; built and explored in %.3f ms%n", 2 * E,
                (long) V * V, tg);

        int s = idOf.get("Stephen King"), t = idOf.get("Charles Dickens");
        int[] parent = new int[V];              // BFS from s, remembering parents
        Arrays.fill(parent, -2);
        ArrayDeque<Integer> q = new ArrayDeque<>();
        q.add(s);
        parent[s] = -1;
        while (!q.isEmpty()) {
            int u = q.poll();
            for (int v : adj.get(u))
                if (parent[v] == -2) { parent[v] = u; q.add(v); }
        }
        List<String> path = new ArrayList<>();
        for (int v = t; v != -1; v = parent[v]) path.add(name.get(v));
        Collections.reverse(path);
        StringBuilder sb = new StringBuilder("  fewest hops from Stephen King to "
                + "Charles Dickens: " + (path.size() - 1) + "\n    ");
        for (int i = 0; i < path.size(); i++)   // three names to a line
            sb.append(" ").append(path.get(i)).append(i + 1 == path.size() ? ""
                    : (i % 3 == 2 ? " ->\n    " : " ->"));
        System.out.println(sb);
    }
}
```

```
Catalog: 10000 books; 2000 lookups (half absent); 200 range queries; 1000 inserts

structure                     build ms   lookup us    range us   insert us
unsorted array                   1.046       1.546       9.619       0.085
sorted array + binary search     1.654       0.206       0.182       0.814
linked list                      1.184      19.141      29.328       0.111
BST (Chapter 11, unbalanced)     1.803       0.158       0.333       0.187
balanced BST (TreeMap)           3.057       0.269       1.763       0.318
hash table (HashMap)             1.966       0.169     113.180       0.272
All 6 structures found 1000 of the 2000 lookups and counted 4702 books
in the ranges: yes
The BST built in arrival order has 35 levels; a balanced one has 14.

Top 10 by ratings: heap of size 10 1.750 ms, sorting everything 1.890 ms
  (same answer: yes)

People graph: 5841 people, 6997 co-credit edges, 3408 components,
  the largest has 605 people
  adjacency lists hold 13994 entries; an adjacency matrix would need
  34117281 cells; built and explored in 75.398 ms
  fewest hops from Stephen King to Charles Dickens: 5
    Stephen King -> Dave McKean -> Richard Dawkins ->
    Thomas Hardy -> Patricia Ingham -> Charles Dickens
```

Listing 16.31 `p1610_choose.py`: the experiment of Section 16.8 in Python, with `bisect` for the sorted array, our own nodes for the list and the tree, and `dict` as the hash table.

```
# p1610_choose.py - one catalog, every structure of the book, measured.
import heapq
import sys
import time
from bisect import bisect_left, insort
from collections import deque

sys.setrecursionlimit(10000)
R = 5


class ListNode:                          # a singly linked list node (Chapter 7)
    __slots__ = ("key", "next")

    def __init__(self, key, nxt):
        self.key, self.next = key, nxt


class TreeNode:                          # a binary search tree node (Chapter 11)
    __slots__ = ("key", "left", "right")

    def __init__(self, key):
        self.key, self.left, self.right = key, None, None


class BST:
    """The unbalanced binary search tree of Chapter 11."""

    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
            return
        p = self.root
        while True:
            if key < p.key:
                if p.left is None:
                    p.left = TreeNode(key)
                    return
                p = p.left
            else:
                if p.right is None:
                    p.right = TreeNode(key)
                    return
```

Listing 16.31 (Continued) *p1610_choose.py: the experiment of Section 16.8 in Python, with bisect for the sorted array, our own nodes for the list and the tree, and dict as the hash table.*

```
                p = p.right

    def contains(self, key):
        p = self.root
        while p is not None and p.key != key:
            p = p.left if key < p.key else p.right
        return p is not None

    @staticmethod
    def count_range(p, lo, hi):
        if p is None:
            return 0
        if p.key < lo:                          # skip the left side
            return BST.count_range(p.right, lo, hi)
        if p.key >= hi:                         # skip the right side
            return BST.count_range(p.left, lo, hi)
        return (1 + BST.count_range(p.left, lo, hi)
                + BST.count_range(p.right, lo, hi))

    @staticmethod
    def height(p):
        return 0 if p is None else 1 + max(BST.height(p.left), BST.height(p.right))


def best_ms(f, reps=R):
    """Runs f several times and returns the best time in milliseconds."""
    best = float("inf")
    for _ in range(reps):
        t0 = time.perf_counter()
        f()
        best = min(best, (time.perf_counter() - t0) * 1000)
    return best


def best_ms_after(make, op, reps=R):
    """Like best_ms, but make() builds a fresh structure first; only op is timed."""
    best = float("inf")
    for _ in range(reps):
        c = make()
        t0 = time.perf_counter()
        op(c)
        best = min(best, (time.perf_counter() - t0) * 1000)
    return best
```

Listing 16.31 (Continued) *p1610_choose.py: the experiment of Section 16.8 in Python, with bisect for the sorted array, our own nodes for the list and the tree, and dict as the hash table.*

```
def main():
    books = []                          # (id, year, ratings, title, authors)
    with open("../data/catalog.tsv", encoding="utf-8") as f:
        next(f)
        for line in f:
            i, y, r, t, a = line.rstrip("\n").split("\t")
            books.append((int(i), int(y), int(r), t, a))
    n = len(books)

    # 1,000 identifiers that are present and 1,000 that are absent.
    present = {b[0] for b in books}
    queries = [books[i][0] for i in range(0, n, 10)]
    fresh = []
    for i in range(5, n, 10):
        q = books[i][0] + 1
        while q in present:
            q += 1
        fresh.append(q)                 # also used as the 1,000 new books
    queries += fresh
    ranges = [(i * 150000, i * 150000 + 50000) for i in range(200)]

    print(f"Catalog: {n} books; {len(queries)} lookups (half absent); "
          f"{len(ranges)} range queries; {len(fresh)} inserts\n")
    print(f"{'structure':<28}{'build ms':>10}{'lookup us':>12}{'range us':>12}"
          f"{'insert us':>12}")
    answers = []

    def row(name, b, look, rng, ins, found, cnt):
        print(f"{name:<28}{b:10.3f}{1000 * look / len(queries):12.3f}"
              f"{1000 * rng / len(ranges):12.3f}{1000 * ins / len(fresh):12.3f}")
        answers.extend([found, cnt])

    res = {}

    # unsorted array: append in arrival order, search by walking
    def build_unsorted():
        res["a"] = [b[0] for b in books]
```

Listing 16.31 (Continued) *p1610_choose.py: the experiment of Section 16.8 in Python, with bisect for the sorted array, our own nodes for the list and the tree, and dict as the hash table.*

```
    def look_unsorted():
        found = 0
        for q in queries:
            for x in res["a"]:
                if x == q:
                    found += 1
                    break
        res["found"] = found

    def range_unsorted():
        res["cnt"] = sum(1 for lo, hi in ranges for x in res["a"] if lo <= x < hi)

    tb, tl = best_ms(build_unsorted), best_ms(look_unsorted)
    tr = best_ms(range_unsorted)
    ti = best_ms_after(lambda: list(res["a"]),
                       lambda c: [c.append(q) for q in fresh])
    row("unsorted array", tb, tl, tr, ti, res["found"], res["cnt"])

    # sorted array: sort once, binary search (Chapter 2)
    def build_sorted():
        res["s"] = sorted(b[0] for b in books)

    def look_sorted():
        s, found = res["s"], 0
        for q in queries:
            i = bisect_left(s, q)
            found += i < len(s) and s[i] == q
        res["found"] = found

    def range_sorted():
        s = res["s"]
        res["cnt"] = sum(bisect_left(s, hi) - bisect_left(s, lo)
                         for lo, hi in ranges)

    tb, tl, tr = best_ms(build_sorted), best_ms(look_sorted), best_ms(range_sorted)
    ti = best_ms_after(lambda: list(res["s"]),
                       lambda c: [insort(c, q) for q in fresh])
    row("sorted array + binary search", tb, tl, tr, ti, res["found"], res["cnt"])

    # singly linked list: push at the front (Chapter 7)
    def build_list():
        head = None
        for b in books:
            head = ListNode(b[0], head)
        res["head"] = head
```

Listing 16.31 (Continued) *p1610_choose.py: the experiment of Section 16.8 in Python, with bisect for the sorted array, our own nodes for the list and the tree, and dict as the hash table.*

```
    def look_list():
        found = 0
        for q in queries:
            p = res["head"]
            while p is not None:
                if p.key == q:
                    found += 1
                    break
                p = p.next
        res["found"] = found

    def range_list():
        cnt = 0
        for lo, hi in ranges:
            p = res["head"]
            while p is not None:
                cnt += lo <= p.key < hi
                p = p.next
        res["cnt"] = cnt

    def push_fresh(box):
        for q in fresh:
            box[0] = ListNode(q, box[0])

    tb, tl, tr = best_ms(build_list), best_ms(look_list), best_ms(range_list)
    ti = best_ms_after(lambda: [res["head"]], push_fresh)
    row("singly linked list", tb, tl, tr, ti, res["found"], res["cnt"])

    # binary search tree of Chapter 11, keys inserted in arrival order
    def build_bst():
        t = BST()
        for b in books:
            t.insert(b[0])
        return t

    tb = best_ms(lambda: res.__setitem__("t", build_bst()))
    height = BST.height(res["t"].root)

    def look_bst():
        res["found"] = sum(1 for q in queries if res["t"].contains(q))

    def range_bst():
        res["cnt"] = sum(BST.count_range(res["t"].root, lo, hi)
                         for lo, hi in ranges)
```

Listing 16.31 (Continued) *p1610_choose.py: the experiment of Section 16.8 in Python, with bisect for the sorted array, our own nodes for the list and the tree, and dict as the hash table.*

```
    tl, tr = best_ms(look_bst), best_ms(range_bst)
    ti = best_ms_after(build_bst, lambda c: [c.insert(q) for q in fresh])
    row("BST (Chapter 11, unbalanced)", tb, tl, tr, ti, res["found"], res["cnt"])
    print(f"{'balanced BST':<28}  (none in the Python standard library)")

    # hash table (Chapter 15): dict
    def build_dict():
        res["h"] = {b[0]: b for b in books}

    def look_dict():
        h = res["h"]
        res["found"] = sum(1 for q in queries if q in h)

    def range_dict():                   # no order: every range scans it all
        h = res["h"]
        res["cnt"] = sum(1 for lo, hi in ranges for k in h if lo <= k < hi)

    def put_fresh(c):
        for q in fresh:
            c[q] = None

    tb, tl, tr = best_ms(build_dict), best_ms(look_dict), best_ms(range_dict)
    ti = best_ms_after(lambda: dict(res["h"]), put_fresh)
    row("hash table (dict)", tb, tl, tr, ti, res["found"], res["cnt"])

    agree = all(answers[i] == answers[i % 2] for i in range(2, len(answers)))
    print(f"All {len(answers) // 2} structures found {answers[0]} of the "
          f"{len(queries)} lookups and counted {answers[1]} books\nin the ranges: "
          f"{'yes' if agree else 'no'}")
    print(f"The BST built in arrival order has {height} levels; "
          "a balanced one has 14.")

    # ------------- the 10 most-rated books: heap against sorting -------------
    def top_heap():                     # min-heap of size 10 (Chapter 14)
        heap = []
        for b in books:
            if len(heap) < 10:
                heapq.heappush(heap, b[2])
            elif b[2] > heap[0]:
                heapq.heapreplace(heap, b[2])
        res["top1"] = sorted(heap, reverse=True)

    def top_sort():                     # sort all 10,000 (Chapter 3)
        res["top2"] = sorted((b[2] for b in books), reverse=True)[:10]
```

Listing 16.31 (Continued) *p1610_choose.py: the experiment of Section 16.8 in Python, with bisect for the sorted array, our own nodes for the list and the tree, and dict as the hash table.*

```
    th, ts = best_ms(top_heap), best_ms(top_sort)
    same = "yes" if res["top1"] == res["top2"] else "no"
    print(f"\nTop 10 by ratings: heap of size 10 {th:.3f} ms, sorting everything "
          f"{ts:.3f} ms\n  (same answer: {same})")

    # ------------- the graph of people credited on the same book -------------
    t0 = time.perf_counter()
    id_of, name, adj = {}, [], []       # name -> vertex number (hashing again)
    for b in books:
        who = []
        for a in b[4].split(","):
            a = a[1:] if a.startswith(" ") else a
            if a not in id_of:
                id_of[a] = len(name)
                name.append(a)
                adj.append([])
            if id_of[a] not in who:
                who.append(id_of[a])
        for u in who:
            for v in who:
                if u != v and v not in adj[u]:
                    adj[u].append(v)
    V = len(name)
    E = sum(len(a) for a in adj) // 2
    comp = [-1] * V                     # label every connected component (BFS)
    comps = largest = 0
    for s in range(V):
        if comp[s] != -1:
            continue
        size, q = 0, deque([s])
        comp[s] = comps
        while q:
            u = q.popleft()
            size += 1
            for v in adj[u]:
                if comp[v] == -1:
                    comp[v] = comps
                    q.append(v)
        largest = max(largest, size)
        comps += 1
    tg = (time.perf_counter() - t0) * 1000
    print(f"\nPeople graph: {V} people, {E} co-credit edges, {comps} components,\n"
          f"  the largest has {largest} people")
    print(f"  adjacency lists hold {2 * E} entries; an adjacency matrix "
          "would need\n"
          f"  {V * V} cells; built and explored in {tg:.3f} ms")
```

Listing 16.31 (Continued) *p1610_choose.py: the experiment of Section 16.8 in Python, with bisect for the sorted array, our own nodes for the list and the tree, and dict as the hash table.*

```
    s, t = id_of["Stephen King"], id_of["Charles Dickens"]
    parent = [-2] * V                   # BFS from s, remembering parents
    parent[s] = -1
    q = deque([s])
    while q:
        u = q.popleft()
        for v in adj[u]:
            if parent[v] == -2:
                parent[v] = u
                q.append(v)
    path, v = [], t
    while v != -1:
        path.append(name[v])
        v = parent[v]
    path.reverse()
    out = ("  fewest hops from Stephen King to Charles Dickens: "
           f"{len(path) - 1}\n   ")
    for i, p in enumerate(path):        # three names to a line
        sep = "" if i + 1 == len(path) else " ->\n   " if i % 3 == 2 else " ->"
        out += " " + p + sep
    print(out)


if __name__ == "__main__":
    main()
```

```
Catalog: 10000 books; 2000 lookups (half absent); 200 range queries; 1000 inserts

structure                     build ms   lookup us    range us   insert us
unsorted array                   0.124      70.476     190.958       0.020
sorted array + binary search     1.512       0.336       0.451       2.092
singly linked list               1.547     139.348     503.491       0.090
BST (Chapter 11, unbalanced)     6.350       0.556       3.624       0.698
balanced BST                  (none in the Python standard library)
hash table (dict)                0.512       0.034     255.484       0.085
All 5 structures found 1000 of the 2000 lookups and counted 4702 books
in the ranges: yes
The BST built in arrival order has 35 levels; a balanced one has 14.

Top 10 by ratings: heap of size 10 0.265 ms, sorting everything 1.114 ms
  (same answer: yes)

People graph: 5841 people, 6997 co-credit edges, 3408 components,
  the largest has 605 people
  adjacency lists hold 13994 entries; an adjacency matrix would need
  34117281 cells; built and explored in 8.582 ms
  fewest hops from Stephen King to Charles Dickens: 5
    Stephen King -> Dave McKean -> Richard Dawkins ->
    Thomas Hardy -> Patricia Ingham -> Charles Dickens
```

Your times will differ from these: they depend on the processor, the compiler, the other programs running, and, in Java, on how far the just-in-time compiler has warmed up. The *ratios* between rows, and the answers, will not.

16.8.3 Reading the numbers

Figure 16.16 plots the C++ measurements on logarithmic axes, where every grid line is ten times the previous one. **Table 16.5** puts the lookup and range results of the three languages side by side.

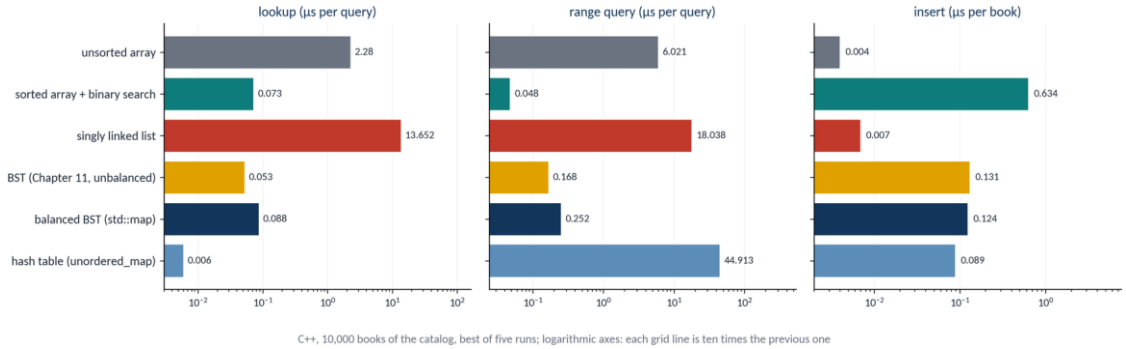


Figure 16.16 Six structures on the Pustaka Ceria catalog (C++, microseconds per operation, best of five runs, logarithmic axes). No structure wins every column.

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

Five lessons can be read off the table and the figure, and each confirms a row of **Table 16.4**.

1. For lookup by key, the hash table wins, by a lot. In C++ it needs 0.006 microseconds per lookup, twelve times faster than binary search and 380 times faster than sequential search. That is Equation (16.10) at work: one hash, one short chain, regardless of n. In Python the gap to the sorted array is smaller, about ten times, because every step of Python costs more and a lookup is dominated by the interpreter; but `dict` is still the fastest lookup Python has.

2. For ranges, the hash table is the worst structure of all. A hash table keeps no order, so "every identifier between a and $a + 50{,}000$" forces it to look at all 10,000 keys for every query: 45 microseconds, against 0.048 for the sorted array, which needs two binary searches and a subtraction. This is the trade-off of Section 16.4.4 in numbers: the structure that answers "is it here?" fastest cannot answer "what is near it?" at all.

3. The linked list is the slowest at almost everything except inserting at the front. Its lookup visits the same number of items as the unsorted array but is about six times slower in C++, because the array's items sit next to each other in memory, while the list's nodes are scattered and every step to the next node is a potential cache miss (Chapter 7). Its one win is insertion at the front, 0.007 microseconds, as **Table 16.4** promises.

4. The unbalanced BST did well here, and that is luck worth understanding. Its lookups are as fast as binary search, and its range queries are fast because the tree prunes whole subtrees. It built a tree of 35 levels where a perfectly balanced one has 14, because the identifiers arrive in order of popularity, which is nearly random with respect to the identifiers. Feed it the same keys in *sorted* order, and every node gets only a right child: 10,000 levels, lookups as slow as the linked list (Chapter 11). The library's balanced tree is a little slower on this friendly input and immune to

the unfriendly one; that insurance is why it is the default choice when you need order and changes.

5. The sorted array is excellent until the data changes. It has the best range queries and the second-best lookups, with no pointers at all. But every insertion shifts on average half of the array: 0.634 microseconds per book, the slowest insertion in the C++ table, and the gap grows linearly with n. For a catalog that is rebuilt once a night it is ideal; for one that changes all day, the balanced tree is the better compromise.

The heap. The ten most-rated books took 0.006 ms with a heap of size 10 and 0.420 ms with a full sort in C++: seventy times less, as Equation (16.7) predicts for $\lg 10 \approx 3.3$ against $\lg 10{,}000 \approx 13.3$ plus the cost of copying and sorting everything. In Python the heap is about four times faster. In the Java run the two are close, because the heap loop ran only five times, too few for the just-in-time compiler to optimize it; timing Java code reliably requires a warm-up phase, which production benchmark tools such as JMH provide.

The graph. The catalog's column of credited people names 5,841 different people (writers, illustrators, editors, translators); joining two people whenever they are credited on the same book gives 6,997 edges. A graph that sparse is exactly what adjacency lists are for: the lists hold $2|E| = 13{,}994$ entries, while an adjacency matrix would need

$$|V|^2 = 5841^2 = 34{,}117{,}281 \text{ cells} \qquad \text{against} \qquad |V| + 2|E| = 19{,}835 \text{ list entries}, \tag{16.12}$$

more than 1,700 times as much memory, almost all of it zeros. By Equation (16.12), even at one bit per cell the matrix would take about 4 megabytes to record fewer than seven thousand connections. BFS from every unvisited person labels 3,408 connected components; the largest joins 605 people, from Stephen King to Charles Dickens, who are five hops apart along the chain the program prints. The whole graph, including building the name-to-number hash map, took under 4 ms in C++. One task used three structures of this book at once: a hash table to turn names into vertex numbers, adjacency lists to store the graph, and a queue to search it.

16.8.4 The choice, in one paragraph

Name the dominant operation. If it is "item number i", use an array. If it is "find by key" and nothing else, use a hash table. If it is "find by key" plus anything about order, use a balanced search tree, or a sorted array if the data seldom changes. If it is "the largest (or smallest) next", use a heap. If the items are connected to each other and the questions are about the connections, use a graph with adjacency lists and the search that matches the question: BFS for fewest steps, DFS or Kahn for structure and order, Dijkstra with a heap for weighted routes. Use stacks and queues when the order of processing is last-in-first-out or first-in-first-out, and linked lists when you hold a position and must insert or delete there. Then, as this section did, measure; and when the measurement disagrees with the table, find out why before you trust either.

16.9 The Problems of This Chapter Compared

Table 16.6 lays the nine problems side by side. As in Chapter 8, read down the last column: most tempting wrong solutions pass the example they were written for. The DFS "distances" are right for four of the seven reachable branches; Kahn without the closing check is perfect on every input without a cycle; `getMin` from the last element is right on the given heap; a hash computed in an

`int` works for short keys; and deletion by emptying a slot works until some key's probe path runs through the emptied slot.

Table 16.6 The nine problems of the chapter: the structure each uses, its cost, and the input on which its tempting wrong solution fails.

Problem	Structure	Time	Extra memory	The input the wrong solution gets wrong
16.1	array max-heap	$O(\lg n)$ extract and increase; $O(n)$ build, `getMin`, check	$O(1)$	a heap whose last element is not the minimum; an array of 0 or 1 elements
16.2	two hash maps	$O(n)$ average	$O(n)$	any pile whose first ticket is not the first leg
16.3	adjacency lists + queue	$O(\|V\| + \|E\|)$	$O(\|V\|)$	any graph where DFS reaches a vertex by a long way first
16.4	adjacency lists + queue	$O(\|V\| + \|E\|)$	$O(\|V\|)$	a graph with a directed cycle
16.5	adjacency lists + min-heap	$O((\|V\| + \|E\|)\lg\|V\|)$	$O(\|V\| + \|E\|)$	a direct road longer than a detour
16.6	min-heap of size k	$O(n \lg k)$	$O(k)$	any input, if the heap is a max-heap
16.7	max-heap + min-heap	$O(\lg n)$ per value	$O(n)$	the fourth value of the sample, without rebalancing
16.8	chained hash table	$O(1 + \alpha)$ average	$O(n + m)$	a key long enough to overflow an `int`
16.9	linear probing	$O(1/(1-\alpha)^2)$ average	$O(m)$	a search whose path crosses a deleted slot

The costs group into three families. **One path down a tree** costs $O(\lg n)$, by Equation (16.2): every heap operation, every lookup in a balanced tree. **One pass over a structure** costs $O(n)$ or $O(|V| + |E|)$: building a heap, BFS, Kahn, the itinerary walk, finding the minimum of a max-heap. **One computed jump** costs $O(1)$ on average: every hash-table operation at a moderate load factor, by Equation (16.10) and Equation (16.11). Seeing which family an operation belongs to is most of the analysis.

16.10 Profitina in Practice

Profitina is an AI-visibility (GEO) platform, in production at app.profitina.com, that measures how ready a brand's website is to be cited by AI answer engines and helps fix what it finds. The three structures of this chapter have obvious places in a product of that kind; we describe the shape of each problem and the class of solution that fits it, not how Profitina itself is built.

A website is a graph. When a site is scanned, its pages are the vertices and its internal links are the arcs. The questions a readiness scan asks are graph questions. Which pages can be reached from the home page at all? That is a search from one vertex, and a page that no search reaches is an orphan that an answer engine's crawler is unlikely to find either. How many clicks away from the home page is an important page? That is BFS distance, exactly Problem 16.3, and pages buried many clicks deep tend to be crawled less. Crawling itself is a BFS whose queue is the *frontier* of discovered but not yet fetched pages, with a "seen" set so that no page is fetched twice.

The seen set is a hash table, and its keys need care. The same page can hide behind many spellings: `https://Example.com/a`, `https://example.com/a/`, `https://example.com/a?utm_source=x`. Before a URL goes into the seen set it is *canonicalized*,

turned into one standard form (lower-case host, trailing slash rule, tracking parameters removed), and only then hashed; otherwise the set of Problem 16.8 would store three entries for one page and the crawl would fetch it three times.

Rescans are a priority queue. Profitina re-scans each brand's site automatically at a cadence that depends on the subscription tier, and raises alerts when a score changes. A scheduler of that kind holds, for every site, the time at which its next scan is due, and repeatedly asks "which scan is due soonest?": a min-heap keyed by due time, the structure of Problems 16.5 and 16.6. When a user asks for an immediate rescan, the due time of that site decreases: the `decreaseKey` of Exercise 16.1, or, more simply, a new entry with lazy deletion as in Problem 16.5.

Top lists are heaps of size k. The community leaderboard and "top posts this week" lists need the k best of many, and need them often; a heap of size k (Problem 16.6) answers without sorting everyone.

Profitina's server is written in Python, so these structures appear as `dict`, `set`, `collections.deque` and `heapq`, the rows of **Table 16.3**, and like every other part of the product they are guarded by the full automated test suite, well over a thousand tests, that must pass before any release. The runnable practice version of this chapter's ideas is Profitina Laundry, the case study of this book series: Exercise 16.14 builds its service desk from a hash map of orders and a priority queue of promised times.

16.11 Common Mistakes, and How to Avoid Them

Using the wrong end of a priority queue. `std::priority_queue` is a max-heap; `PriorityQueue` and `heapq` are min-heaps. Code copied between languages silently inverts (Problem 16.6). Write the direction in a comment where the heap is declared.

Keeping a max-heap to find the k largest. To keep the k largest you must throw away the smallest of them, so the heap is a *min*-heap (Problem 16.6).

Forgetting to save the root before overwriting it. `extractMax` must copy `heap[0]` into a local variable before the last leaf moves in (Problem 16.1).

Repairing in the wrong direction. A larger key can only rise; a smaller key can only sink. `increaseKey` sifts up, `decreaseKey` sifts down (Problem 16.1, Exercise 16.1).

Unsigned arithmetic that underflows. `arr.size() - 2` is huge when the array has fewer than two elements. Loop over children from index 1 instead, or compare before subtracting (Problem 16.1).

Looking for the minimum of a max-heap in one place. It is somewhere among the $\lceil n/2 \rceil$ leaves, never at a fixed index (Problem 16.1).

Calling `std::map` a hash map. It is a balanced search tree with $O(\lg n)$ operations and sorted iteration; the hash table is `std::unordered_map` (Problem 16.2).

Relying on the iteration order of a hash table. It is unspecified and changes after a rehash. Iterate over the input, or sort, when output order matters (Problem 16.2).

Using DFS depth as a distance. Only BFS, with its queue, finds fewest-edge routes (Problem 16.3).

Marking a vertex when it leaves the queue instead of when it enters. The distances stay correct, but a vertex can enter the queue many times; mark on entry.

Trusting a topological sort without counting. If fewer than $|V|$ vertices were output, there is a cycle (Problem 16.4).

Fixing a Dijkstra distance on first sight. A distance is final only when its vertex leaves the heap as the smallest (Problem 16.5). And Dijkstra requires nonnegative weights.

Forgetting to rebalance two heaps. The median is a statement about positions; the sizes must stay within one of each other (Problem 16.7).

Computing a hash in a signed `int`. It overflows (undefined behavior in C++) and can give a negative index. Use unsigned arithmetic and reduce modulo m at every step, or `Math.floorMod` in Java (Problem 16.8).

Deleting from an open-addressing table by emptying the slot. Use a tombstone (Problem 16.9).

Letting a hash table fill up. Costs grow like $1 + \alpha$ for chaining and like $1/(1-\alpha)^2$ for linear probing; rehash before α gets large (Equation (16.10), Equation (16.11)).

Starting a chain from the wrong place. The first item of the input is not the first item of the answer (Problem 16.2).

Choosing a structure before naming the operation. Name the dominant operation first, then consult **Table 16.4**, then measure (Section 16.8).

16.12 Chapter Summary

- A problem about a data structure is solved with nine steps (Section 16.2): restate it, choose the structure with a reason, draw it, write its invariant, list the edge cases, order the changes safely, count the cost, decide what to do with bad input, and test against a memory checker and a slow twin.
- Every structure promises an invariant, and every operation must leave the promise true; most bugs of this chapter are operations that do not.
- Heaps (Problems 16.1, 16.6, 16.7): the children of index i are $2i+1$ and $2i+2$, Equation (16.1); the height is $\lfloor \lg n \rfloor$, Equation (16.2), so insert, extract and increase cost $O(\lg n)$; the minimum of a max-heap is among the leaves, Equation (16.3); building bottom-up costs $O(n)$, Equation (16.4); the k largest need a *min*-heap of size k, Equation (16.7), with about $k \ln(n/k)$ replacements on random input, Equation (16.8); a running median needs a max-heap and a min-heap of balanced sizes.
- Hash tables (Problems 16.2, 16.8, 16.9): the load factor $\alpha = n/m$, Equation (16.9), governs the cost, $\Theta(1+\alpha)$ for chaining, Equation (16.10), and about $\frac{1}{2}(1 + 1/(1-\alpha)^2)$ probes for an unsuccessful search with linear probing, Equation (16.11); compute hashes without overflow; delete with tombstones in open addressing; and remember that `std::map` is a tree, not a hash table.
- Graphs (Problems 16.3, 16.4, 16.5): BFS finds fewest-edge routes in $O(|V| + |E|)$, Equation (16.5); Kahn's algorithm finds a topological order or proves a cycle; Dijkstra with a heap finds quickest routes with nonnegative weights in $O((|V| + |E|) \lg |V|)$, Equation (16.6); sparse graphs belong in adjacency lists, Equation (16.12).
- Choosing a structure (Section 16.8): name the dominant operation and consult **Table 16.4**. On the real catalog, the hash table was the fastest for lookup and the slowest for ranges, the sorted

array the best for ranges and the slowest to insert into, the linked list the slowest at almost everything but front insertion, and the heap seventy times faster than sorting for the top ten.

16.13 Exercises

Exercises 16.1 to 16.6 and 16.14 ask for code; their solutions are the programs of Section 16.14. The others ask for reasoning and are answered in words.

1. **(Decrease a key.)** Write `decreaseKey(i, newValue)` for the max-heap of Problem 16.1. In which direction does the repair go, and why? Apply `decreaseKey(0, 2)` to `[8, 5, 6, 4, 5, 3, 2, 1, 3, 1]`.
2. **(Merge k sorted lists.)** Merge the sorted lists {3, 9, 14}, {1, 10}, {2, 4, 8, 20} and an empty list into one sorted list using a min-heap that holds at most one entry per list. What is the cost for k lists with N values in total?
3. **(Two sides.)** A graph is *bipartite* if its vertices can be colored with two colors so that every edge joins different colors. Decide it with BFS. Is the branch map of Problem 16.3 bipartite? Is a square (four vertices in a cycle)?
4. **(Components.)** Count the connected components of a graph with an iterative DFS that uses an explicit stack. How many does the branch map have?
5. **(First unique character.)** Find the first character of a string that occurs exactly once, in two passes with a hash map. Try "borrowed books" and "aabb".
6. **(Two prices.)** Given the prices 35, 80, 42, 17, 60, 23 and a voucher value, find two prices that add up to it in one pass with a hash set. Try 77 and 100. Why must a price not be paired with itself?
7. **(Heap by hand.)** Insert 20, 5, 15, 22, 40, 11, 30 one by one into an empty max-heap. Give the array after each insertion, then the array after one `extractMax`.
8. **(Build by hand.)** Apply `buildMaxHeap` to {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}. List the calls to `heapifyDown` in order and the array after each.
9. **(BFS and DFS by hand.)** For the directed graph with arcs 0→1, 0→2, 1→3, 2→3, 3→4, 2→4, list the BFS order and the recursive DFS order from 0 (neighbors in increasing order), and the BFS distance of every vertex.
10. **(Dijkstra and a negative road.)** On the graph with arcs A→B 2, A→C 5, C→B −4, what does Dijkstra's algorithm report as the distance to B, and what is the true shortest distance? Which step of the invariant fails?
11. **(Probing by hand.)** With $m = 7$, $h(k) = k \bmod 7$ and linear probing, insert 12, 19, 5, 26, 33. Then remove 19 with a tombstone and search for 26 and for 40. Give every probe sequence.
12. **(Which structure?)** Name the structure of **Table 16.4** for each Pustaka Ceria task, with a one-sentence reason: (a) the members' records, looked up by membership number at the desk all day; (b) the list of overdue loans, always processed from the most overdue; (c) the shelf catalog, printed in order of call number every night and changed a few times a week; (d) the "customers who borrowed this also borrowed" links between books; (e) the undo history of the catalog editor; (f) the books whose codes lie between B2000 and B2999.
13. **(Matrix or lists?)** A city has 20,000 intersections and 52,000 road segments. How many cells has an adjacency matrix, how many entries have adjacency lists, and how many bytes does each need with 4-byte integers (and, for the matrix, with one bit per cell)? Which would you choose for BFS?

14. **(Profitina Laundry.)** Profitina Laundry's service desk receives orders, each with an order number, a customer name and a promised hour. The washer always takes the order with the earliest promised hour (ties by order number), and the desk must answer "what is the status of order x?" instantly. Choose the two structures, and write `receive`, `washNext` and `status`. Run: receive 501 (Ani, 17:00), 502 (Budi, 10:00), 503 (Citra, 12:00); wash next; receive 504 (Dewi, 9:00, an express order); wash next; show the status of 503, 502 and 999.

16.14 Solutions to the Exercises

Exercises 16.1 to 16.6 and 16.14. Listing 16.32, **Listing 16.33** and **Listing 16.34** contain all the requested functions and print the same lines.

Listing 16.32 `ex16_solutions.cpp`: solutions to the programming exercises of Chapter 16.

```
// ex16_solutions.cpp - solutions to the programming exercises of Chapter 16.
#include <functional>
#include <iostream>
#include <queue>
#include <stdexcept>
#include <string>
#include <tuple>
#include <unordered_map>
#include <unordered_set>
#include <vector>
using namespace std;

void print(const string& label, const vector<int>& v) {
    cout << label;
    for (size_t i = 0; i < v.size(); i++) cout << (i ? " " : "") << v[i];
    cout << "\n";
}

// ---------- Exercise 16.1: decreaseKey in a max-heap (a smaller key sinks) ----
void siftDown(vector<int>& h, size_t i) {
    while (true) {
        size_t l = 2 * i + 1, r = 2 * i + 2, big = i;
        if (l < h.size() && h[l] > h[big]) big = l;
        if (r < h.size() && h[r] > h[big]) big = r;
        if (big == i) return;
        swap(h[i], h[big]);
        i = big;
    }
}

void decreaseKey(vector<int>& h, size_t i, int newValue) {
    if (i >= h.size()) throw out_of_range("decreaseKey: no element at that index");
    if (newValue > h[i]) throw invalid_argument("decreaseKey: new value is larger");
    h[i] = newValue;
    siftDown(h, i);
}
```

Listing 16.32 (Continued) *ex16_solutions.cpp: solutions to the programming exercises of Chapter 16.*

```
// ---------- Exercise 16.2: merge k sorted lists with a min-heap ----------
vector<int> mergeSorted(const vector<vector<int>>& lists) {
    using E = tuple<int, size_t, size_t>;          // (value, which list, position)
    priority_queue<E, vector<E>, greater<E>> heap;
    for (size_t j = 0; j < lists.size(); j++)
        if (!lists[j].empty()) heap.push({lists[j][0], j, 0});
    vector<int> out;
    while (!heap.empty()) {
        auto [v, j, p] = heap.top();
        heap.pop();
        out.push_back(v);
        if (p + 1 < lists[j].size()) heap.push({lists[j][p + 1], j, p + 1});
    }
    return out;
}

// ---------- Exercise 16.3: is the graph bipartite? (BFS two-coloring) ----------
bool bipartite(const vector<vector<int>>& adj) {
    vector<int> color(adj.size(), -1);
    for (size_t s = 0; s < adj.size(); s++) {
        if (color[s] != -1) continue;
        color[s] = 0;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : adj[u]) {
                if (color[v] == -1) {
                    color[v] = 1 - color[u];       // the other side
                    q.push(v);
                } else if (color[v] == color[u]) {
                    return false;                  // an edge inside one side
                }
            }
        }
    }
    return true;
}
```

Listing 16.32 (Continued) *ex16_solutions.cpp: solutions to the programming exercises of Chapter 16.*

```
// ---------- Exercise 16.4: count connected components (DFS, a stack) ----------
int components(const vector<vector<int>>& adj) {
    vector<bool> seen(adj.size(), false);
    int count = 0;
    for (size_t s = 0; s < adj.size(); s++) {
        if (seen[s]) continue;
        count++;
        vector<int> stack = {(int)s};
        seen[s] = true;
        while (!stack.empty()) {
            int u = stack.back();
            stack.pop_back();
            for (int v : adj[u])
                if (!seen[v]) {
                    seen[v] = true;
                    stack.push_back(v);
                }
        }
    }
    return count;
}

// ---------- Exercise 16.5: the first character that occurs only once ----------
char firstUnique(const string& s) {
    unordered_map<char, int> count;
    for (char c : s) count[c]++;                  // pass 1: count
    for (char c : s)                              // pass 2: in the original order
        if (count[c] == 1) return c;
    return '\0';
}

// ---------- Exercise 16.6: two prices that add up to a voucher ----------
bool twoSum(const vector<int>& price, int target, int& a, int& b) {
    unordered_set<int> seen;
    for (int p : price) {
        if (seen.count(target - p)) {             // its partner came earlier
            a = target - p;
            b = p;
            return true;
        }
        seen.insert(p);
    }
    return false;
}
```

Listing 16.32 (Continued) *ex16_solutions.cpp: solutions to the programming exercises of Chapter 16.*

```
// ---------- Exercise 16.14: the Profitina Laundry service desk ----------
struct Order {
    int number;
    string customer;
    int readyBy;                                    // promised hour, 0..23
    string status;
};

class LaundryDesk {
    unordered_map<int, Order> byNumber;             // look up any order in O(1)
    using Key = pair<int, int>;                     // (promised hour, order number)
    priority_queue<Key, vector<Key>, greater<Key>> next;   // earliest promise first
public:
    void receive(int number, const string& customer, int readyBy) {
        byNumber[number] = {number, customer, readyBy, "waiting"};
        next.push({readyBy, number});
    }
    void washNext() {
        if (next.empty()) {
            cout << "  nothing to wash\n";
            return;
        }
        auto [hour, number] = next.top();
        next.pop();
        byNumber[number].status = "washed";
        cout << "  wash order " << number << " (" << byNumber[number].customer
             << ", ready by " << hour << ":00)\n";
    }
    void status(int number) const {
        auto it = byNumber.find(number);
        if (it == byNumber.end()) cout << "  order " << number << ": unknown\n";
        else cout << "  order " << number << ": " << it->second.status << "\n";
    }
};

int main() {
    cout << "Exercise 16.1\n";
    vector<int> h = {8, 5, 6, 4, 5, 3, 2, 1, 3, 1};
    decreaseKey(h, 0, 2);
    print("  after decreaseKey(0, 2): ", h);

    cout << "Exercise 16.2\n";
    print("  merged: ", mergeSorted({{3, 9, 14}, {1, 10}, {2, 4, 8, 20}, {}}));
```

Listing 16.32 (Continued) *ex16_solutions.cpp: solutions to the programming exercises of Chapter 16.*

```
    cout << "Exercise 16.3\n";
    vector<vector<int>> branches = {{1, 2, 3}, {0, 6, 4}, {0, 5}, {0, 6},
                                    {1, 5}, {2, 4}, {1, 3}, {}};
    vector<vector<int>> square = {{1, 3}, {0, 2}, {1, 3}, {2, 0}};
    cout << boolalpha << "  branch map bipartite: " << bipartite(branches)
         << "\n  a square bipartite: " << bipartite(square) << "\n";

    cout << "Exercise 16.4\n";
    cout << "  components of the branch map: " << components(branches) << "\n";

    cout << "Exercise 16.5\n";
    for (string s : {"borrowed books", "aabb"}) {
        char c = firstUnique(s);
        cout << "  \"" << s << "\" -> " << (c ? string(1, c) : string("none"))
             << "\n";
    }

    cout << "Exercise 16.6\n";
    int a, b;
    vector<int> price = {35, 80, 42, 17, 60, 23};
    for (int target : {77, 100}) {
        if (twoSum(price, target, a, b))
            cout << "  " << target << " = " << a << " + " << b << "\n";
        else
            cout << "  " << target << ": no pair\n";
    }

    cout << "Exercise 16.14\n";
    LaundryDesk desk;
    desk.receive(501, "Ani", 17);
    desk.receive(502, "Budi", 10);
    desk.receive(503, "Citra", 12);
    desk.washNext();
    desk.receive(504, "Dewi", 9);                  // an express order
    desk.washNext();
    desk.status(503);
    desk.status(502);
    desk.status(999);
    return 0;
}
```

Listing 16.33 `Ex16Solutions.java`: the same solutions in Java.

```java
// Ex16Solutions.java - solutions to the programming exercises of Chapter 16.
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;

public class Ex16Solutions {

    static String join(List<Integer> v) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < v.size(); i++)
            sb.append(i > 0 ? " " : "").append(v.get(i));
        return sb.toString();
    }

    // ---------- Exercise 16.1: decreaseKey in a max-heap (a smaller key sinks) --
    static void siftDown(List<Integer> h, int i) {
        while (true) {
            int l = 2 * i + 1, r = 2 * i + 2, big = i;
            if (l < h.size() && h.get(l) > h.get(big)) big = l;
            if (r < h.size() && h.get(r) > h.get(big)) big = r;
            if (big == i) return;
            int t = h.get(i);
            h.set(i, h.get(big));
            h.set(big, t);
            i = big;
        }
    }

    static void decreaseKey(List<Integer> h, int i, int newValue) {
        if (i < 0 || i >= h.size())
            throw new IndexOutOfBoundsException(
                    "decreaseKey: no element at that index");
        if (newValue > h.get(i))
            throw new IllegalArgumentException("decreaseKey: new value is larger");
        h.set(i, newValue);
        siftDown(h, i);
    }
```

Listing 16.33 (Continued) *Ex16Solutions.java: the same solutions in Java.*

```
    // ---------- Exercise 16.2: merge k sorted lists with a min-heap ----------
    static List<Integer> mergeSorted(int[][] lists) {
        // entries are {value, which list, position in that list}
        PriorityQueue<int[]> heap = new PriorityQueue<>((x, y) ->
                x[0] != y[0] ? Integer.compare(x[0], y[0])
              : x[1] != y[1] ? Integer.compare(x[1], y[1])
              : Integer.compare(x[2], y[2]));
        for (int j = 0; j < lists.length; j++)
            if (lists[j].length > 0) heap.add(new int[] {lists[j][0], j, 0});
        List<Integer> out = new ArrayList<>();
        while (!heap.isEmpty()) {
            int[] e = heap.poll();
            out.add(e[0]);
            int j = e[1], p = e[2];
            if (p + 1 < lists[j].length)
                heap.add(new int[] {lists[j][p + 1], j, p + 1});
        }
        return out;
    }

    // ---------- Exercise 16.3: is the graph bipartite? (BFS two-coloring) ------
    static boolean bipartite(int[][] adj) {
        int[] color = new int[adj.length];
        java.util.Arrays.fill(color, -1);
        for (int s = 0; s < adj.length; s++) {
            if (color[s] != -1) continue;
            color[s] = 0;
            ArrayDeque<Integer> q = new ArrayDeque<>();
            q.add(s);
            while (!q.isEmpty()) {
                int u = q.poll();
                for (int v : adj[u]) {
                    if (color[v] == -1) {
                        color[v] = 1 - color[u];   // the other side
                        q.add(v);
                    } else if (color[v] == color[u]) {
                        return false;              // an edge inside one side
                    }
                }
            }
        }
        return true;
    }
```

Listing 16.33 (Continued) *Ex16Solutions.java: the same solutions in Java.*

```
    // ---------- Exercise 16.4: count connected components (DFS, a stack) ------
    static int components(int[][] adj) {
        boolean[] seen = new boolean[adj.length];
        int count = 0;
        for (int s = 0; s < adj.length; s++) {
            if (seen[s]) continue;
            count++;
            ArrayDeque<Integer> stack = new ArrayDeque<>();
            stack.push(s);
            seen[s] = true;
            while (!stack.isEmpty()) {
                int u = stack.pop();
                for (int v : adj[u])
                    if (!seen[v]) {
                        seen[v] = true;
                        stack.push(v);
                    }
            }
        }
        return count;
    }

    // ---------- Exercise 16.5: the first character that occurs only once ------
    static String firstUnique(String s) {
        HashMap<Character, Integer> count = new HashMap<>();
        for (char c : s.toCharArray()) count.merge(c, 1, Integer::sum);    // pass 1
        for (char c : s.toCharArray())                                     // pass 2
            if (count.get(c) == 1) return String.valueOf(c);
        return "none";
    }

    // ---------- Exercise 16.6: two prices that add up to a voucher ----------
    static int[] twoSum(int[] price, int target) {
        HashSet<Integer> seen = new HashSet<>();
        for (int p : price) {
            if (seen.contains(target - p)) return new int[] {target - p, p};
            seen.add(p);
        }
        return null;
    }

    // ---------- Exercise 16.14: the Profitina Laundry service desk ----------
    static class LaundryDesk {
        record Order(int number, String customer, int readyBy) {}
```

Listing 16.33 (Continued) *Ex16Solutions.java: the same solutions in Java.*

```
        final HashMap<Integer, Order> byNumber = new HashMap<>();    // O(1) lookup
        final HashMap<Integer, String> status = new HashMap<>();
        // entries are {promised hour, order number}; earliest promise first
        final PriorityQueue<int[]> next = new PriorityQueue<>((x, y) ->
                x[0] != y[0] ? Integer.compare(x[0], y[0])
                             : Integer.compare(x[1], y[1]));

        void receive(int number, String customer, int readyBy) {
            byNumber.put(number, new Order(number, customer, readyBy));
            status.put(number, "waiting");
            next.add(new int[] {readyBy, number});
        }

        void washNext() {
            if (next.isEmpty()) {
                System.out.println("  nothing to wash");
                return;
            }
            int[] top = next.poll();
            status.put(top[1], "washed");
            System.out.println("  wash order " + top[1] + " ("
                    + byNumber.get(top[1]).customer() + ", ready by " + top[0]
                    + ":00)");
        }

        void showStatus(int number) {
            System.out.println("  order " + number + ": "
                    + status.getOrDefault(number, "unknown"));
        }
    }

    public static void main(String[] args) {
        System.out.println("Exercise 16.1");
        List<Integer> h = new ArrayList<>(List.of(8, 5, 6, 4, 5, 3, 2, 1, 3, 1));
        decreaseKey(h, 0, 2);
        System.out.println("  after decreaseKey(0, 2): " + join(h));

        System.out.println("Exercise 16.2");
        System.out.println("  merged: " + join(mergeSorted(
                new int[][] {{3, 9, 14}, {1, 10}, {2, 4, 8, 20}, {}})));
```

Listing 16.33 (Continued) *Ex16Solutions.java: the same solutions in Java.*

```
        System.out.println("Exercise 16.3");
        int[][] branches = {{1, 2, 3}, {0, 6, 4}, {0, 5}, {0, 6},
                            {1, 5}, {2, 4}, {1, 3}, {}};
        int[][] square = {{1, 3}, {0, 2}, {1, 3}, {2, 0}};
        System.out.println("  branch map bipartite: " + bipartite(branches)
                + "\n  a square bipartite: " + bipartite(square));

        System.out.println("Exercise 16.4");
        System.out.println("  components of the branch map: "
                + components(branches));

        System.out.println("Exercise 16.5");
        for (String s : new String[] {"borrowed books", "aabb"})
            System.out.println("  \"" + s + "\" -> " + firstUnique(s));

        System.out.println("Exercise 16.6");
        int[] price = {35, 80, 42, 17, 60, 23};
        for (int target : new int[] {77, 100}) {
            int[] ab = twoSum(price, target);
            if (ab != null)
                System.out.println("  " + target + " = " + ab[0] + " + " + ab[1]);
            else
                System.out.println("  " + target + ": no pair");
        }

        System.out.println("Exercise 16.14");
        LaundryDesk desk = new LaundryDesk();
        desk.receive(501, "Ani", 17);
        desk.receive(502, "Budi", 10);
        desk.receive(503, "Citra", 12);
        desk.washNext();
        desk.receive(504, "Dewi", 9);                  // an express order
        desk.washNext();
        desk.showStatus(503);
        desk.showStatus(502);
        desk.showStatus(999);
    }
}
```

Listing 16.34 `ex16_solutions.py`: the same solutions in Python.

```python
# ex16_solutions.py - solutions to the programming exercises of Chapter 16.
import heapq
from collections import Counter, deque


# ---------- Exercise 16.1: decreaseKey in a max-heap (a smaller key sinks) ----
def sift_down(h, i):
    while True:
        l, r, big = 2 * i + 1, 2 * i + 2, i
        if l < len(h) and h[l] > h[big]:
            big = l
        if r < len(h) and h[r] > h[big]:
            big = r
        if big == i:
            return
        h[i], h[big] = h[big], h[i]
        i = big


def decrease_key(h, i, new_value):
    if not 0 <= i < len(h):
        raise IndexError("decreaseKey: no element at that index")
    if new_value > h[i]:
        raise ValueError("decreaseKey: new value is larger")
    h[i] = new_value
    sift_down(h, i)


# ---------- Exercise 16.2: merge k sorted lists with a min-heap ----------
def merge_sorted(lists):
    # entries are (value, which list, position in that list)
    heap = [(lst[0], j, 0) for j, lst in enumerate(lists) if lst]
    heapq.heapify(heap)
    out = []
    while heap:
        v, j, p = heapq.heappop(heap)
        out.append(v)
        if p + 1 < len(lists[j]):
            heapq.heappush(heap, (lists[j][p + 1], j, p + 1))
    return out
```

Listing 16.34 (Continued) *ex16_solutions.py: the same solutions in Python.*

```
# ---------- Exercise 16.3: is the graph bipartite? (BFS two-coloring) ----------
def bipartite(adj):
    color = [-1] * len(adj)
    for s in range(len(adj)):
        if color[s] != -1:
            continue
        color[s] = 0
        q = deque([s])
        while q:
            u = q.popleft()
            for v in adj[u]:
                if color[v] == -1:
                    color[v] = 1 - color[u]         # the other side
                    q.append(v)
                elif color[v] == color[u]:
                    return False                    # an edge inside one side
    return True


# ---------- Exercise 16.4: count connected components (DFS, a stack) ----------
def components(adj):
    seen = [False] * len(adj)
    count = 0
    for s in range(len(adj)):
        if seen[s]:
            continue
        count += 1
        stack = [s]
        seen[s] = True
        while stack:
            u = stack.pop()
            for v in adj[u]:
                if not seen[v]:
                    seen[v] = True
                    stack.append(v)
    return count


# ---------- Exercise 16.5: the first character that occurs only once ----------
def first_unique(s):
    count = Counter(s)                              # pass 1: count
    for c in s:                                     # pass 2: in the original order
        if count[c] == 1:
            return c
    return "none"
```

Listing 16.34 (Continued) *ex16_solutions.py: the same solutions in Python.*

```
# ---------- Exercise 16.6: two prices that add up to a voucher ----------
def two_sum(price, target):
    seen = set()
    for p in price:
        if target - p in seen:                      # its partner came earlier
            return target - p, p
        seen.add(p)
    return None


# ---------- Exercise 16.14: the Profitina Laundry service desk ----------
class LaundryDesk:
    def __init__(self):
        self.by_number = {}             # number -> [customer, hour, status]
        self.next = []                  # min-heap of (promised hour, number)

    def receive(self, number, customer, ready_by):
        self.by_number[number] = [customer, ready_by, "waiting"]
        heapq.heappush(self.next, (ready_by, number))

    def wash_next(self):
        if not self.next:
            print("  nothing to wash")
            return
        hour, number = heapq.heappop(self.next)
        self.by_number[number][2] = "washed"
        customer = self.by_number[number][0]
        print(f"  wash order {number} ({customer}, ready by {hour}:00)")

    def status(self, number):
        order = self.by_number.get(number)
        print(f"  order {number}: {order[2] if order else 'unknown'}")


def main():
    print("Exercise 16.1")
    h = [8, 5, 6, 4, 5, 3, 2, 1, 3, 1]
    decrease_key(h, 0, 2)
    print("  after decreaseKey(0, 2):", " ".join(map(str, h)))

    print("Exercise 16.2")
    print("  merged:", " ".join(map(str, merge_sorted([[3, 9, 14], [1, 10],
                                                       [2, 4, 8, 20], []]))))
```

Listing 16.34 (Continued) *ex16_solutions.py: the same solutions in Python.*

```
    print("Exercise 16.3")
    branches = [[1, 2, 3], [0, 6, 4], [0, 5], [0, 6], [1, 5], [2, 4], [1, 3], []]
    square = [[1, 3], [0, 2], [1, 3], [2, 0]]
    print("  branch map bipartite:", str(bipartite(branches)).lower())
    print("  a square bipartite:", str(bipartite(square)).lower())

    print("Exercise 16.4")
    print("  components of the branch map:", components(branches))

    print("Exercise 16.5")
    for s in ("borrowed books", "aabb"):
        print(f'  "{s}" -> {first_unique(s)}')

    print("Exercise 16.6")
    price = [35, 80, 42, 17, 60, 23]
    for target in (77, 100):
        ab = two_sum(price, target)
        print(f"  {target} = {ab[0]} + {ab[1]}" if ab else f"  {target}: no pair")

    print("Exercise 16.14")
    desk = LaundryDesk()
    desk.receive(501, "Ani", 17)
    desk.receive(502, "Budi", 10)
    desk.receive(503, "Citra", 12)
    desk.wash_next()
    desk.receive(504, "Dewi", 9)                    # an express order
    desk.wash_next()
    desk.status(503)
    desk.status(502)
    desk.status(999)


if __name__ == "__main__":
    main()
```

```
Exercise 16.1
  after decreaseKey(0, 2): 6 5 3 4 5 2 2 1 3 1
Exercise 16.2
  merged: 1 2 3 4 8 9 10 14 20
Exercise 16.3
  branch map bipartite: false
  a square bipartite: true
Exercise 16.4
  components of the branch map: 2
Exercise 16.5
  "borrowed books" -> w
  "aabb" -> none
Exercise 16.6
  77 = 35 + 42
  100: no pair
Exercise 16.14
  wash order 502 (Budi, ready by 10:00)
  wash order 504 (Dewi, ready by 9:00)
  order 503: waiting
  order 502: washed
  order 999: unknown
```

16.1. A smaller key can break the invariant only with its children, never with its parent, so the repair goes *down*: `siftDown` from the index. On the given heap the root becomes 2, sinks past 6 (the larger child) and then past 3 (at index 5, larger than 2 at index 6): `6 5 3 4 5 2 2 1 3 1`. Cost $O(\lg n)$ by Equation (16.2).

16.2. The heap holds the smallest remaining value of each list, with the list and position it came from. Each pop outputs the global minimum, and the next value of the same list replaces it. The heap never holds more than k entries, so each of the N values costs $O(\lg k)$: $O(N \lg k)$ in all. The empty list is simply never entered into the heap.

16.3. Color the start 0 and every newly discovered vertex the opposite color of its discoverer; an edge between two vertices of the same color proves an odd cycle, and a graph with an odd cycle cannot be two-colored. The branch map contains the five-cycle Pusat, Gubeng, Rungkut, Wonokromo, Darmo, Pusat, so it is not bipartite; the square is. Cost $O(|V| + |E|)$.

16.4. Each search from an unseen vertex marks one whole component; the number of searches started is the number of components. The branch map has 2: the seven connected branches, and Benowo alone.

16.5. The first pass counts every character in a hash map; the second walks the string in its original order and returns the first character with count 1. In "borrowed books", b, o, r appear more than once and w is the first single one. "aabb" has none. Two passes, $O(L)$ for a string of length L.

16.6. For each price p, the partner $t - p$ must have appeared *earlier*, so we look it up before inserting p. For 77: at 42 the set holds {35, 80}, and $77 - 42 = 35$ is there. For 100 there is no pair. Looking up before inserting is also what stops a price from being paired with itself: for a voucher of 70 and a single price 35, the lookup of 35 happens while 35 is not yet in the set. (Chapter 4 solved the same problem with two indices on a *sorted* array in $O(n \lg n)$; the hash set does it in $O(n)$ on average without sorting.)

16.7. After each insertion: `[20]`; `[20, 5]`; `[20, 5, 15]`; `[22, 20, 15, 5]` (22 rises past 5 and 20); `[40, 22, 15, 5, 20]` (40 rises past 20 and 22); `[40, 22, 15, 5, 20, 11]`; `[40, 22, 30, 5, 20, 11, 15]` (30 rises past 15). `extractMax` returns 40; 15 moves to the root and sinks past 30: `[30, 22, 15, 5, 20, 11]`.

16.8. With $n = 10$ the internal indices are 4 down to 0. `heapifyDown(4)`: 16 is larger than its child 7, nothing to do. `heapifyDown(3)`: 2 against 14 and 8, swap with 14: `4 1 3 14 16 9 10 2 8 7`. `heapifyDown(2)`: 3 against 9 and 10, swap with 10: `4 1 10 14 16 9 3 2 8 7`. `heapifyDown(1)`: 1 against 14 and 16, swap with 16, then 1 against 7, swap: `4 16 10 14 7 9 3 2 8 1`. `heapifyDown(0)`: 4 against 16 and 10, swap with 16; then against 14 and 7, swap with 14; then against 2 and 8, swap with 8: `16 14 10 8 7 9 3 2 4 1`. This is the example of Cormen et al. [9], and the result matches theirs.

16.9. BFS from 0: taking 0 discovers 1 and 2 (distance 1); taking 1 discovers 3 (distance 2); taking 2 finds 3 already seen and discovers 4 (distance 2, from 2); the order is 0, 1, 2, 3, 4 with distances 0, 1, 1, 2, 2. Recursive DFS from 0 goes 0, 1, 3, 4, then backs up to 0 and visits 2, whose neighbors 3 and 4 are already visited: the order is 0, 1, 3, 4, 2.

16.10. Dijkstra takes A (0), relaxes B to 2 and C to 5, takes B at 2 and declares it done; then takes C at 5, whose arc offers B $5 - 4 = 1$, but B is already done, so the reported distance is 2. The true distance is 1, along A→C→B. The invariant's argument "every other route must leave the done region at a time at least as large, and lengths are never negative" fails at its last clause: the route through C left at 5 and *decreased* afterward. Bellman–Ford handles negative weights [9].

16.11. 12 → 5; 19 → 5 taken, 6; 5 → 5 taken, 6 taken, 0; 26 → 5, 6, 0 taken, 1; 33 → 5, 6, 0, 1 taken, 2. Table: 0: 5, 1: 26, 2: 33, 5: 12, 6: 19. Remove 19: slot 6 becomes a tombstone. Search 26: probes 5 (12), 6 (tombstone, continue), 0 (5), 1 (26): found. Search 40 (40 mod 7 = 5): 5, 6, 0, 1, 2, 3

(empty): not found after six probes. Had slot 6 been emptied, the search for 26 would have stopped at 6 and missed it.

16.12. (a) Hash table: lookup by number only, all day. (b) Heap (priority queue) keyed by days overdue: always the largest next. (c) Sorted array: printed in order nightly, rarely changed, so the cost of insertion does not matter. (d) Graph with adjacency lists: connections between items. (e) Stack: undo takes the most recent change first. (f) A range query: sorted array or balanced search tree with `lower_bound`, never a hash table.

16.13. Matrix: $20,000^2 = 4 \times 10^8$ cells, 1.6 GB with 4-byte integers, 50 MB with one bit per cell. Lists: $2 \times 52{,}000 = 104{,}000$ entries plus 20,000 list headers, well under 1 MB. BFS on the lists costs $O(|V| + |E|) \approx 124{,}000$ steps, on the matrix $|V|^2 = 4 \times 10^8$. Choose the lists; by Equation (16.12) the matrix only pays for dense graphs.

16.14. A hash map from order number to order answers `status` in $O(1)$; a min-heap of (promised hour, order number) gives the next order to wash in $O(\lg n)$. Budi (10:00) is washed first; Dewi's express order (9:00), received afterwards, jumps ahead of Citra (12:00) and is washed second; 503 is still waiting, 502 is washed, and 999 is unknown. The output is the last block of the program output above.

References

[1] Free Software Foundation, "Program Instrumentation Options," in *Using the GNU Compiler Collection (GCC)*, GCC 16.2, Aug. 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://gcc.gnu.org/onlinedocs/

[2] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annual Technical Conf.*, Boston, MA, USA, 2012, pp. 309–318.

[3] Valgrind Developers, *Valgrind User Manual*, release 3.27.1, May 2026. Accessed: Sep. 24, 2026. [Online]. Available: https://valgrind.org/docs/manual/

[4] B. Stroustrup, *A Tour of C++*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2022.

[5] cppreference.com, "std::priority_queue," "std::map," "std::unordered_map," and "std::push_heap." Accessed: Sep. 24, 2026. [Online]. Available: https://en.cppreference.com/w/cpp

[6] Oracle, "Class PriorityQueue," "Class HashMap," "Class TreeMap," and "Class ArrayDeque," *Java SE 25 API Specification*. Accessed: Sep. 24, 2026. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/

[7] Python Software Foundation, "heapq — Heap queue algorithm," "bisect," "collections.deque," and "Mapping Types — dict," *Python 3.14 Documentation*. Accessed: Sep. 24, 2026. [Online]. Available: https://docs.python.org/3/

[8] R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[10] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[11] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[13] Z. Zajac, "goodbooks-10k: A new dataset for book recommendations," 2017. [Online]. Available: https://github.com/zygmuntz/goodbooks-10k. License CC BY-SA 4.0. Accessed: Sep. 24, 2026.

[14] J. W. J. Williams, “Algorithm 232: Heapsort,” *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[15] E. F. Moore, “The shortest path through a maze,” in *Proc. Int. Symp. on the Theory of Switching*, Cambridge, MA, USA: Harvard Univ. Press, 1959, pp. 285–292.

[16] W. W. Peterson, “Addressing for random-access storage,” *IBM Journal of Research and Development*, vol. 1, no. 2, pp. 130–146, 1957.

[17] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

The heap and heap sort are due to Williams [14], and the linear-time bottom-up construction to Floyd [8]; breadth-first search as a shortest-path method appears in Moore’s work on mazes [15], Dijkstra’s algorithm in his 1959 note [11], and topological sorting of large networks in Kahn [10]. Open addressing with linear probing was introduced by Peterson [16] and analyzed by Knuth [12], whose formulas Equation (16.11) reproduces. Cormen et al. [9] and Sedgewick and Wayne [17] treat every structure of this chapter with proofs; the library behavior follows cppreference [5], the Java API specification [6] and the Python documentation [7], and the C++ language rules Stroustrup [4]. The catalog of Section 16.8 is the goodbooks-10k data set [13]. The memory checkers are described in the GCC manual [1], the Valgrind manual [3] and the AddressSanitizer paper [2].