ALGORITHMS

IN ACTION

FIRST EDITION

Victor Savvich

8.00 x 10.00
254 mm x 203 mm

ALGORITHMS IN ACTION

ALGORITHMS
IN ACTION

First Edition

Victor Savvich
Univeristy of Southern California

cognella
SAN DIEGO

Bassim Hamadeh, CEO and Publisher
Mieka Portier, Field Acquisitions Editor
Tony Paese, Project Editor
Alia Bales, Production Editor
Jess Estrella, Senior Graphic Designer
Trey Soto, Licensing Coordinator
Natalie Piccotti, Director of Marketing
Kassie Graves, Vice President of Editorial
Jamie Giganti, Director of Academic Publishing

Copyright © 2020 by Cognella, Inc. All rights reserved. No part of this publication may be reprinted, reproduced, transmitted, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information retrieval system without the written permission of Cognella, Inc. For inquiries regarding permissions, translations, foreign rights, audio rights, and any other forms of reproduction, please contact the Cognella Licensing Department at rights@cognella.com.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

Cover image copyright © 2018 iStockphoto LP/monsitj.

Printed in the United States of America.

cognella® | ACADEMIC PUBLISHING
3970 Sorrento Valley Blvd., Ste. 500, San Diego, CA 9212

Contents

Preface vii

Chapter 1 Review 1

1.1 Runtime Complexity 1
1.2 Lower Bound for Sorting 3
1.3 Trees and Graphs 4

Chapter 2 Amortized Analysis 15

2.1 Unbounded Array 15
2.2 Binary Counter 18
2.3 Amortized Dictionary 19
2.4 Amortized Trees 20

Chapter 3 Heaps 25

3.1 Binary Heaps 25
3.2 Binomial Heaps 33
3.3 Fibonacci Heaps 40

Chapter 4 Greedy Algorithms 45

4.1 The Money Changing Problem 46
4.2 Scheduling Problem 47
4.3 Huffman Code 49
4.4 Minimum Spanning Trees 56
4.5 Shortest Path Problem 62

Chapter 5 Divide-and-Conquer Algorithms 69

5.1 Solving Divide-and-Conquer Recurrences 71
5.2 Integer Multiplication 76
5.3 Matrix Multiplication 78
5.4 The Maximum Subsequence Sum Problem 80
5.5 Computing Fibonacci Numbers 81

Chapter 6 Dynamic Programming 87

6.1 Introduction 87
6.2 Knapsack Problem 89
6.3 Static Optimal Binary Search Tree 94
6.4 The Bellman-Ford Algorithm 98
6.5 The Shortest Path in DAGs 101

Chapter 7 Network Flow 107

7.1 Introduction 107
7.2 The Ford–Fulkerson Algorithm 110
7.3 Reduction to Network Flow 116
7.4 Augmenting Path Heuristics 120
7.5 The Circulation Problem 123
7.6 Reduction to Circulation 130

Chapter 8 Linear Programming 137

8.1 Introduction: A Production Problem 137
8.2 The Standard Maximum Problem 141
8.3 A Few Applications 144
8.4 The Dual Linear Program 148

Chapter 9 NP Completeness 157

9.1 A Brief Introduction to the Turing Machines 157
9.2 Computational Intractability 159
9.3 NP-Complete Problems 163

Preface

THIS BOOK IS AN INTRODUCTION TO algorithm design, intended to teach a variety of techniques for designing algorithms, with a focus on an intuitive understanding of algorithms. The book will train you on many basic algorithms, so you should be able to employ them in future algorithmic courses (like machine learning) and eventually apply them in your professional work. The book does not intend to be comprehensive nor complete.

The genesis of this book came about through the lecture notes I developed while teaching undergraduate and graduate computer science courses at Carnegie Mellon University and the University of Southern California. The book covers roughly a semester's worth of coursework, though some chapters go far beyond the standard lecture material, letting students dive deeper into the concepts and thus providing material to stimulate further thought and discussion. The book is intended for advanced undergraduate or master-level students in computer science and/or related technical disciplines. A foundation of undergraduate coursework in discrete mathematics, data structures, and calculus is highly recommended as a prerequisite. The book does not emphasize nor require programming, just pseudocode to encourage readers on conceptual understanding.

Analysis of algorithms is challenging for the most students, as they have not yet developed an experience in algorithmic problem solving. Students often easily come up with an erroneous intuitive solution, demonstrating their overconfidence in understanding material. My approach is to explain how to design algorithms, focusing on providing fundamental concepts, with a detailed, visual step-by-step algorithm execution. The book contains over 160 figures that help the reader to visualize the process. While I provide proofs of algorithm correctness, my goal is not to overwhelm the reader with rigorous mathematical proofs.

My hope with this book is to offer a reader-friendly approach to algorithms, with the numerous review questions and exercises at the end of each chapter (122 short review questions and 129 exercises in total) allowing readers to practice and apply the concepts taught.

Victor Savvich
Playa Vista, California
April 2019

Chapter 1

Review

IN THIS CHAPTER WE REVIEW BASIC concepts, from asymptotic complexity, graph theory, and mathematical proof techniques, as they are required for better understanding for the chapters that follow. If the reader has some previous acquaintance with these topics, the chapter should be enough to get started. If the reader has no previous background in these, we suggest a more thorough introduction such as *Mathematics for Computer Science* by Eric Lehman, Thomson Leighton and Albert Meyer.[1]

1.1 Runtime Complexity

The term *analysis of algorithms* is used to describe approaches to study the performance of algorithms. With each algorithm we associate a sequence of steps comprising this algorithm. We measure the run time of an algorithm by counting the number of steps and therefore define an algorithmic complexity as a numerical function $T(n)$, where n is the input size. Consider a problem of addition of two n-bit binary numbers. Let $T(n)$ represent an amount of time addition used to add two n-bit numbers. We want to define "time" $T(n)$ taken by the method of addition without having to worry about implementation details. The process of addition consists of the following two steps:

- Adding 3 bits (one bit is a carry bit)
- Writing down 2 bits (again, one bit is a carry bit)

1 Eric Lehman, Thomson Leighton, and Albert Meyer, *Mathematics for Computer Science,* ([Great Britain: Samurai Media Limited, 2017).

On any computer, adding and writing two bits can be done in constant time. By constant time we mean that the time is independent of the input size. Therefore, the total time of addition of two n-bit binary numbers is $T(n) = n \cdot c$, where the constant c can be different on different computers. We say that bit addition is a linear time algorithm. The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science. In this course we will perform the following types of analysis:

1. The worst-case complexity (the maximum number of steps taken on any input)
2. The best-case complexity (the minimum number of steps taken on any input)
3. The average case complexity (the average number of steps taken on a random input)
4. The amortized time complexity (the average complexity over a sequence of operations)

We measure the runtime of an algorithm using following asymptotic notations: O, Ω, Θ.

1.1.1 Upper Bound (Big-O)

For any monotonic functions, f, g from the positive integers to the positive integers, we say $f(n) = O(g(n))$ (or $f(n) \in O(g(n))$) if $g(n)$ eventually dominates $f(n)$. Figure 1.1 helps you to visualize this relationship.

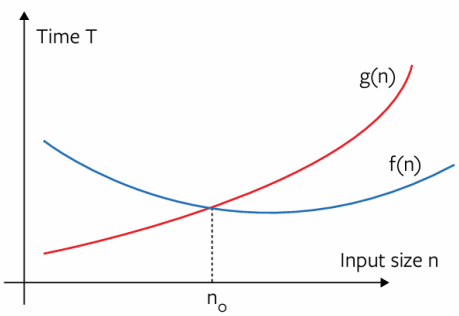


FIGURE 1.1 $f(n) = O(g(n))$.

Formally, there exists a positive real number, $c > 0$, and a real number, n_0, such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Example: $n^2 + 2n + 1 = O(n^2)$. Since $n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$ for $n \geq 1$, we choose $c = 4$ and $n_0 = 1$.

1.1.2 Lower Bound (Big-Omega)

For any monotonic functions, f, g from the positive integers to the positive integers, we say $f(n) = \Omega(g(n))$ (or $f(n) \in \Omega(g(n))$) if $f(n)$ eventually dominates $g(n)$. Formally, there exists a positive real number, $c > 0$, and a real number, n_0, such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Example: $n^2 + 2n + 1 = \Omega(n^2)$. Since $n^2 + 2n + 1 \geq n^2$ for $n \geq 1$, we choose $c = 1$ and $n_0 = 1$.

1.1.3 Exact Bound (Big-Theta)

For any monotonic functions, f, g from the positive integers to the positive integers, we say $f(n) = \Theta(g(n))$ (or $f(n) \in \Theta(g(n))$) if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Formally, there exists positive real numbers, c_1 and c_2, and a real number, n_0, such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.

Example: $n^2 + 2n + 1 = \Theta(n^2)$.

1.2 Lower Bound for Sorting

We will show here that any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time to sort an array of n elements in the worst case. Comparison-based sorting algorithms operate on the input by comparing pairs of elements. For example, Mergesort and insertion sort are comparison-based sorting algorithms. But bucket sort and radix sort are not. In order to show $\Omega(n \log n)$ bound we will play the Guess-a-Number game. The computer will select a number, x, between 1 and 10, and you need to determine x by asking questions. You'll keep guessing numbers until you find x. The guessing game can be viewed abstractly as a binary search tree (also called a decision tree). Figure 1.2 shows that we can guess any number between 1 and 10 by asking at most four questions.

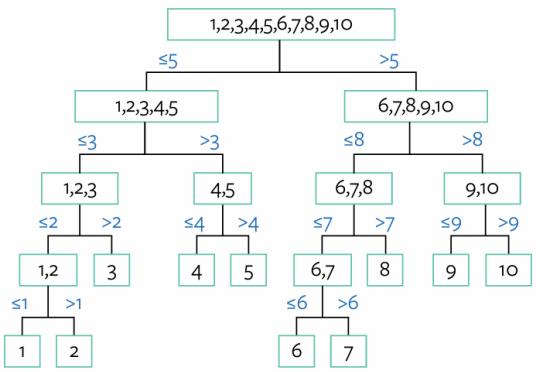


FIGURE 1.2 A binary search tree to guess a number between 1 and 10.

We will be using a decision tree to model the execution of any comparison-based sort. The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node we make a comparison; based on that comparison we proceed further down to either the left or right subtree. Figure 1.3 depicts sorting an array of three elements $[a, b, c]$. In that tree each leaf represents a permutation of $[a, b, c]$. Generally, for sorting an array of n elements, each leaf in the

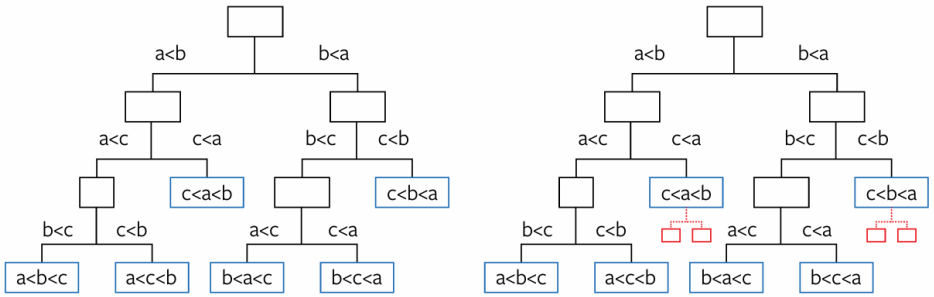


FIGURE 1.3 A binary search tree to guess a number between 1 and 10.

decision tree represents a permutation of the n elements. Hence, there are $n!$ leaves in the tree. Let us denote the tree by B. Next, we make the tree complete by adding extra nodes, as seen in figure 1.3. We will denote the new tree by B^* and its height by h. Since tree B^* is a complete binary tree, it has 2^h leaves. By construction, B^* has more leaves than B. It follows, $2^h \geq n!$, or, after taking the log of both parts, $h \geq \log(n!)$.

The height of the decision tree is the number of comparisons in a sorting algorithm; in other words, h is the runtime complexity $T(n)$ of sorting. Thus, $T(n) \geq \log(n!)$. Lastly, we simplify $\log(n!)$ as follows:

$$\begin{aligned}\log(n!) &= \log(n(n-1)(n-2)\ldots 1)\\ &\geq \log(n(n-1)\ldots(n-n/2))\\ &\geq \log((n/2)^{n/2}) = \Omega(n \log n)\end{aligned}$$

We have proved that any comparison-based sorting algorithm needs $\Omega(n \log n)$ comparisons. This holds even for a quantum computer!

1.3 Trees and Graphs

A graph G is a pair (V, E), where V is a set of vertices (or nodes) and E is a set of edges connecting the vertices. A self-loop is an edge that connects to the same vertex twice. A multi-edge is a set of two or more edges that have the same two vertices. A graph is simple if it has no multi-edges or self-loops. We always assume simple graphs unless otherwise noted. Graphs could be directed and undirected and weighted and unweighted (weights will usually be edge weights).

Theorem. *Let G be a graph with V vertices and E edges. The following statements are equivalent:*

1. *G is a tree.*
2. *Every two vertices of G are connected by a unique path.*
3. *G is connected and $V = E + 1$.*

4. *G is acyclic and* $V = E + 1$.
5. *G is acyclic and if any two non-adjacent vertices are joined by an edge, the resulting graph has exactly one cycle.*

To prove this, it suffices to show $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5 \Rightarrow 1$. We'll just show $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4$ and leave the rest to the reader.

Proof of 1 ⇒ 2. We prove it by contradiction. Assume G is a tree that has two vertices connected by two different paths like in figure 1.4.

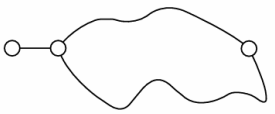


FIGURE 1.4 Two vertices connected by two different paths.

Then there exists a cycle! It follows that G cannot be a tree: a contradiction. ■

Proof of 2 ⇒ 3. Since every two vertices in G are connected by a path, G is a connected graph. We prove that in G the number of nodes and edges are related by $V = E + 1$. The proof is by strong induction on the number of nodes.

Base case: $V = 2$. Since a graph is simple, $E = 1$. Thus, $V = E + 1$.

Inductive hypothesis: Assume $V = E + 1$ for every graph with $V < n$ vertices.

Inductive step: Prove $V = E + 1$ for every graph with $V = n$ vertices.

Graph G has n vertices. We will use notation $V(G) = n$. We choose two adjacent vertices, x and y. We know that every two vertices in G are connected by a unique path. It follows that x and y are joined by an edge, like in figure 1.5.

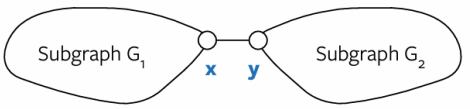


FIGURE 1.5 Graph G consists of two subgraphs G_1 and G_2.

Note in both subgraphs G_1 and G_2 the number of vertices is less than n. Indeed, G_1 does not contain vertex y, and G_2 does not contain vertex x. We can apply the inductive hypothesis to G_1 and G_2. It follows,

$$V(G) = V(G_1) + V(G_2) = E(G_1) + 1 + E(G_2) + 1 = E(G) + 1.$$

This concludes the proof. ■

Proof of 3 ⇒ 4. We prove that G is an acyclic graph by contradiction. Assume that G has a cycle with k vertices in it. This cycle also contains k edges. Now let us count edges in

the whole graph. We claim the number of edges in the graph will be at least V. Indeed, there are k edges in the cycle and at least $V - k + 1$ outside the cycle. ■

Theorem. *In an undirected simple graph $G = (V, E)$, there are at most $V(V - 1)/2$ edges. In short, by using the asymptotic notation, $E = O(V^2)$.*

Proof. Choose any vertex in G. The possible number of edges leaving this vertex is $V - 1$. Take another vertex (different from the previous one). The possible number of edges leaving that vertex is $V - 2$ (don't count the edge between two vertices twice!), and so on. We have that the total number of edges is at most

$$(V - 1) + (V - 2) + \ldots + 2 + 1 = V(V - 1)/2$$

This concludes the proof. ■

We define a *dense* graph $G = (V, E)$ as a graph in which the number of edges is $E = \Omega(V^2)$. We say that a graph is *sparse* if $E = O(V)$.

1.3.1 Graph Traversals

Graphs traversal means visiting all vertices in a systematic order. We can choose any vertex as a starting point. Then we will systematically enumerate all vertices accessible from it. Because a graph might contain cycles, we need some way for marking a vertex as having been visited. To do so we will keep a Boolean array, with all elements initially set to false. We will set a correspondent element to true as soon as we visit a particular vertex. Also, we need to keep in mind that the graph might be disconnected. There are two most common traversals:

- Depth-first search (DFS)
- Breadth-first search (BFS)

DFS uses a stack data structure for backtracking. BFS uses a FIFO queue for bookkeeping. Here is a pseudocode:

BFS : FIFO 队列
先进先出

DFS : LIFO 栈
后进先出

```
for all v in V do visited[v] = false
for all v in V do if !visited[v] traversal(v)
  traversal(v) {
    visited[v] = true
    for all w in adj(v)
        do if !visited[w] traversal(w)
  }
```

```
DFS(s):
  初始化 S 为具有一个元素 s 的栈
  While S 非空
    从 S 中取一个结点 u
    If Explored[u]=false then
      置 Explored[u]=true
      For 每条与 u 关联的边(u,v)
        把 v 加到栈 S
      Endfor
    Endif
  Endwhile
```

The runtime complexity of traversal is $O(V+E)$. There are two important properties of traversal: (1) It visits all the vertices in the connected component; (2) edges labeled by traversal form a spanning tree of the connected component.

1.3.2 Topological Sort

Suppose each vertex represents a task that must be completed and a directed edge (u, v) indicates that task v depends on task u. That is, u must be completed before v. If G is a direct acyclic graph (DAG), then there exists a valid order in which you can complete the tasks. This is called topological order or topological sort. If the graph is cyclic, no topological order exists. Consider the graph in figure 1.6.

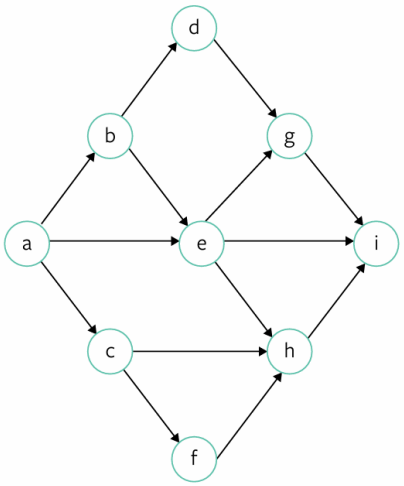


FIGURE 1.6 A directed acyclic graph.

The following sequence {a, b, c, d, e, f, g, h, i} is a valid topological sort. In other words, a topological order means arranging the vertices along a line so that all edges go from left to right. It should be evident from figure 1.6 that a topological sort is not unique. The following list {a, c, b, f, e, d, h, g, i} is another topological order.

The algorithm of finding a topological sort is based on traversal: run DFS (from any vertex) and return a vertex that has no undiscovered leaving edges. In figure 1.6, the possible DFS run may be $a \rightarrow e \rightarrow i$, making i the first vertex with no undiscovered leaving edges. From i, backtrack to e and then go to g. This makes g another vertex with no undiscovered leaving edges. From g, backtrack to e and then go to h. This makes h the third vertex with no undiscovered leaving edges. And so on. The algorithm will produce a topological order in reverse. Note, if we start DFS at any other vertex but a, we will need another run of DFS. The runtime complexity of the algorithm is linear $O(V+E)$.

1.3.3 Planar Graphs

A connected graph is planar if it can be drawn in the plane with each edge drawn as a continuous curve such that no two edges cross. There are many examples of planar graphs: any tree is planar, every cycle is planar, a complete graph K_4 is planar.

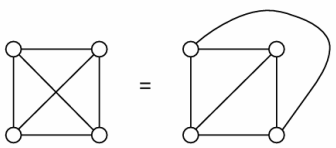


FIGURE 1.7 An example of a planar graph.

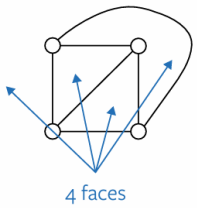


FIGURE 1.8 A planar graph with four faces.

A planar graph in addition to vertices and edges also has disjoint faces.

Theorem. (Euler's formula) *If G is a connected planar graph with V vertices, E edges, and F faces, then*

$$V - E + F = 2.$$

Proof. The proof by induction on the number of edges.

Base case: $E = 1$. The identity holds, since $V = 2$ and $F = 1$.

Inductive hypothesis: Assume it's true for any graph with no more than E edges.

Inductive step: Prove it for graphs with E edges.

Start with a graph G that has E edges and remove one edge. There are two cases to consider:

1. The edge to remove lies on a cycle. See figure 1.9.
2. The edge to remove does not lie on a cycle. See figure 1.10.

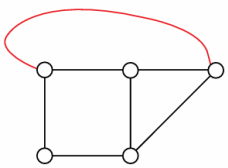


FIGURE 1.9 A graph in case 1.

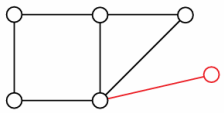


FIGURE 1.10 A graph in case 2.

In case (1) by removing an edge (in red) on a cycle, we obtain a new graph G' with $E - 1$ edges and F-1 faces. Since G' has $E - 1$ edges, the relation works by the induction hypothesis. That is $V - (E - 1) + (F - 1) = 2$. This simplifies to $V - E + F = 2$.

In case (2) by removing an edge (in red) that does not lies on a cycle, we obtain a new graph G' with $E - 1$ edges and $V - 1$ vertices. Since G' has $E - 1$ edges, the relation works by the induction hypothesis. That is $(V - 1) - (E - 1) + F = 2$. This simplifies to $V - E + F = 2$.

This completes the proof. ■

if remove 1 E , F --

∴ V-E+F=2 ✓

Theorem. *In any simple connected planar graph with at least 3 vertices, $E \leq 3V - 6$.*

Proof. If a graph has no cycles, then

$$E = V - 1 \leq V \leq V + (2V - 6) = 3V - 6,$$

since $V \geq 3$, and therefore $2V - 6 \geq 0$.

Assume a graph with cycles. We will count the number of pairs (edge, face) (i.e., Σ(edge, face)). Since each face is bounded by at least 3 edges, then $\Sigma(\text{edge, face}) \geq 3F$. Since each edge is associated with at most 2 faces, then $\Sigma(\text{edge, face}) \leq 2E$. Combining these two inequalities, we find $3F \leq 2E$. But we know from the previous theorem that $V - E + F = 2$. It follows,

$$6 = 3V - 3E + 3F \leq 3V - 3E + 2E = 3V - E.$$

Thus, we conclude $E \leq 3V - 6$. ■

Corollary. *A simple connected planar graph with at least 3 vertices has a vertex of degree 5 or less.*

Proof. We know that in any graph, the sum of the degrees of all vertices is equal to twice the number of edges, $\Sigma\text{degree(v)} = 2E$. From the previous theorem, $2E \leq 6V - 12$. Thus, the average degree is at most 6:

$$\frac{1}{V}\sum_{v \in V}\text{degree}(v) \leq \frac{6\ V - 12}{V} = 6 - \frac{12}{V}$$

It follows there exists a vertex of degree 5 or less. ■

1.3.4 Coloring Planar Graphs

Given a planar graph, how many colors do you need in order to color the vertices so that no two adjacent vertices get the same color? Back in the 1880s, Francis Guthrie conjectured that four colors suffice. In 1976 K. Appel and W. Haken, using a special-purpose computer program, have proved that conjecture.

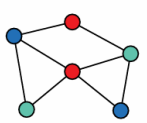


FIGURE 1.11 A graph coloring problem.

We won't prove the conjecture but let us prove the six-color theorem.

Theorem. (6-color theorem) *Every planar graph can be colored with at most six colors.*

Proof. By induction on the number of vertices.

Base case: If a graph has six or less vertices, then the result is obvious.

Inductive hypothesis: Assume that all graphs with $V - 1$ vertices are six-colorable.

Inductive step: Prove it for any graph with V vertices.

By previous corollary any graph G with V vertices has at least one vertex of degree 5 or less. Remove it from G. The remaining graph is planar and by induction can be colored with at most 6 colors. Now insert that vertex back. Since this vertex has at most 5 neighbors then at least one of 6 colors is not used. We color the vertex with one of the unused colors. ■

1.3.5 Bipartite Graphs

A graph is bipartite if the vertices can be partitioned into two disjoint sets, X and Y, such that all edges go only between X and Y (no edges go from X to X or from Y to Y).

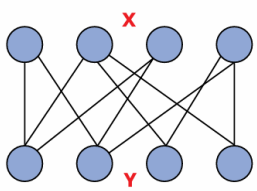


FIGURE 1.12 A bipartite graph.

A complete bipartite graph (denoted by $K_{n,m}$, where n and m are sizes of two sets) is a special kind of *bipartite graph* where every vertex of the first partition is connected to every vertex of the second partition.

Theorem. *A graph is bipartite if and only if it does not contain an odd length cycle.*

Proof. Note, the length of a cycle is the number of vertices in the cycle.

⇒) In a bipartite graph every cycle has vertices that must alternate between two partitions. Since the number of vertices in such a cycle is even, it is an even cycle.

⇐) Assume connected graph G has no odd cycle. Pick any vertex v. Define two sets of vertices based on parity of distance (even or odd) from v:

$$X = \{u \in V \mid \mathrm{d}(v, u) \text{ is even}\}$$
$$Y = \{u \in V \mid \mathrm{d}(v, u) \text{ is odd}\}$$

These sets provide a bipartition. If G had an odd cycle, then there will be a vertex present on both sets. Finish the proof for disconnected graph G. ■

One of the famous problems on bipartite graphs is a matching problem. A subset of edges is a *matching* if no two edges have a common vertex. A *maximum matching* is a matching with the largest possible number of edges. Our goal is to find the maximum matching in a graph. We will show in chapter 7 that the problem of finding the maximum matching can be reduced to the maximum flow problem.

1.3.6 Other Famous Graph Problems

A Euler path is a path that uses each edge of a graph exactly once. A Euler cycle is a cycle that uses every edge of a graph exactly once. A graph that contains a Euler cycle is called a *Eulerian graph*.

Theorem. *A connected graph G is a Eulerian graph if and only if all vertices of G are of even degree.*

Proof. $\Rightarrow$) Let $G = (V, E)$ be a Euler graph. Thus, G contains a Euler path. Let us walk that path. Visiting an intermediate vertex in the path contributes two to the degree of that vertex. It follows that every intermediate vertex has an even degree.

$\Leftarrow$) Assume that all vertices of G are of even degree. We construct a cycle starting at an arbitrary vertex v, going through the edges of G only once. Since every vertex is of even degree, we eventually come back to v. If this cycle includes all the edges of G, then G is a Eulerian graph. If not, we remove from G the cycle we have constructed. We will get a connected subgraph G_0 in which all vertices are of even degree. We again construct a new cycle in G_0. This process is repeated until we obtain a cycle that traces all the edges of G. We showed that G is a collection of cycles, hence G is a Eulerian graph. ■

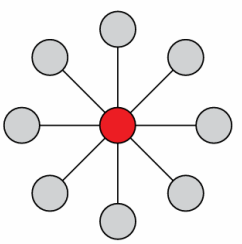


FIGURE 1.13 The *vertex cover* problem.

A *Hamiltonian path* is a path that visits each vertex of a graph exactly once. A Hamiltonian cycle is a cycle that visits every vertex in a graph exactly once (except for the start and end vertices). A graph that contains a Hamiltonian cycle is called a *Hamiltonian graph*. A problem closely related to Hamiltonian cycle is the famous *traveling salesman problem. Given a* weighted graph, find the shortest weighted Hamiltonian cycle. We will prove in chapter 9 that both problems are *unlikely to be solved in polynomial time.* The *traveling salesman problem* is one of the most intensively studied problems in computer science.

A *vertex cover* of an undirected graph is a subset of *vertices* such that for every edge (u, v) either u or v is in a *vertex cover*. The *vertex cover* problem is to find the minimum size *vertex cover*.

Given a graph, we say that a subset of vertices is independent if no two of them are joined by an edge. Given an undirected graph, the *independent set* problem is to find the largest independent set. Vertex cover and independent set are very closely related graph problems; see Exercise 15.

Two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are *isomorphic* if there is a bijective function $f: V_1 \to V_2$ such that an edge $(u, v) \in E_1$, if and only if an edge $(f(u), f(v)) \in E_2$. Two isomorphic graphs look differently but are structurally the same, up to the renaming of the vertices.

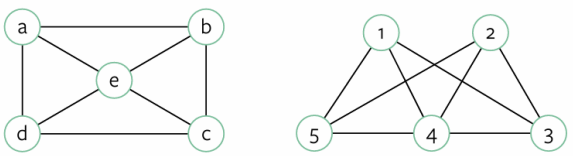


FIGURE 1.14 Two isomorphic graphs.

In figure 1.14 we can match vertices as follows: $a - 1$, $b - 2$, $c - 3$, $d - 5$, $e - 4$. Although we matched vertices in one particular way, there could be several ways to do it. Determining whether two graphs are isomorphic is not an easy task; however, computer scientists believe the problem can be solved in polynomial time.

REVIEW QUESTIONS

1. Mark the following assertions as TRUE or FALSE. No need to provide any justification.
   a. $n = O(n^2)$ T
   b. $n = O(\sqrt{n})$ F
   c. $\log n = \Omega(n)$ F
   d. $n^2 = \Omega(n \log n)$ T
   e. $n^2 \log n = \Theta(n^2)$ F
   f. $7 \log^2 n + 2n \log n = \Omega(\log n)$ T
   g. $5n \log n + 1024\, n \log(\log n) = \Theta(n \log n)$ T
   h. $2^n + 100n^2 + n^{100} = O(n^{101})$ F
   i. $(1/3)^n + 100 = O(1)$ T

T 2. (**T/F**) Any function which is $\Omega(\log n)$ is also $\Omega(\log(\log n))$. $\exists\, c,\ f(n) \geq c \cdot \log n \geq c \cdot \log(\log n)$

T 3. (**T/F**) If $f(n) = \Theta(g(n))$ then $g(n) = \Theta(f(n))$. $\exists\ c_1, c_2$

$c_1 \cdot g(n) \leq f(x) \leq c_2 \cdot g(n)$

$g(n) \geq \frac{1}{c_2} f(x)$ $\quad g(n) \leq \frac{1}{c_1} f(x)$

$\leq f(x) \leq$

T 4. (**T/F**) If $f(n) = \Theta(g(n))$ then $f(n) = \Omega(g(n))$.

F T 5. (**T/F**) If $f(n) = \Omega(g(n))$ then $2^{f(n)} = \Omega(2^{g(n)})$. $\exists c,\ f(n) \geq c \cdot g(n) \Rightarrow 2^{f(n)} \geq 2^{c \cdot g(n)}$ $f(n)=n$ $g(n)=5n$

T 6. (**T/F**) BFS can be used to find the shortest path between any two nodes in a non-weighted graph.

F 7. (**T/F**) A DFS tree is never the same as a BFS tree.

F 8. (**T/F**) Algorithm A has a running time of $O(n^2)$ and algorithm B has a running time of $O(n \log n)$. From this we conclude that A can never run faster than B on the same input.

T 9. (**T/F**) Planar graph is a sparse graph.

T 10. (**T/F**) Every DAG contains a vertex with no incoming edges.

EXERCISES

1. Prove $g(n) = \Omega(f(n))$ if and only if $f(n) = O(g(n))$.
2. Prove or disprove $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.
3. Arrange the following functions

$$\log n^n,\ n^2,\ n^{\log n},\ n \log\log n,\ 2^{\log n},\ \log^2 n,\ n^{\sqrt{2}}$$

in increasing order of growth rate, with $g(n)$ following $f(n)$ in your list if and only if $f(n) = O(g(n))$.

$\log_b a = \frac{\log a}{\log b}$ ①

$\log_n 2 = \frac{\log 2}{\log n} = \frac{1}{\log n}$

$\sqrt{2\log n}$ ③

$2 < \sqrt{n}$

$\sqrt{2\log n} < \log\sqrt{n} = \frac{1}{2}\log n$

True

$\log\log \ldots \log n$? $\log[(\log n)!]$

4. Arrange the following functions

⑤ ② ⑦ ④ ③ ① ⑥

$$4^{\log n},\ \sqrt{\log n},\ n^{\log\log n},\ (\sqrt{2})^{\log n},\ 2^{\sqrt{2\log n}},\ n^{1/\log n},\ (\log n)!$$

n^2 $\sqrt{n}$ 2

in increasing order of growth rate with $g(n)$ following $f(n)$ in your list if and only if $f(n) = O(g(n))$.

5. What is the Big-O runtime complexity of the following function?

⑦ ? ⑥

```
void bigOh1 (int n):
  for i=1 to n
    j=1;
    while j < n
      j = j*2;
```

換底公式 $n^{\log(c)} = c^{\log n}$

$= (\log n)^{\log n}$

6. What is the Big-O runtime complexity of the following function?

$(\log_2 4)^2 = 4$

$\log_2 16 = 4$

$(\log n + \log n + \cdots)$

12 12 12 12

次4 次4

n n

```
void bigOh2 (int n):
  if(n == 0) return "a";
  string str = bigOh2(n-1);
  return str + str;
```

7. What is the Big-Theta runtime complexity of the following function? Here `find_max` finds the maximum element in the array L[0], L[1], ..., L[n − 1].

```
void bigTheta (int[] L, int n):
  while (n > 0)
    find_max(L, n);
        n = n/4
```

8. The complete graph on n vertices, denoted K_n, is a simple graph in which there is an edge between every pair of distinct vertices. What is the height of the DFS tree for the complete graph K_n? What is the height of the BFS tree for the complete graph K_n?
9. We are interested in finding a simple path in a directed acyclic graph that visits all vertices once and only once. Design a linear time algorithm to determine if there is such a path in a given DAG.
10. Prove that a complete graph K_5 is not a planar graph.
11. Prove that a complete bipartite graph $K_{3,3}$ is not a planar graph.
12. In a connected bipartite graph, is the bipartition unique? Justify your answer.
13. Given a directed graph $G = (V, E)$ and a particular node $v \in V$, design a linear time algorithm to determine whether v is in a triangle of edges (a cycle of length 3).
14. Design a linear time algorithm which, given an undirected graph $G = (V, E)$ and a particular edge $e \in E$, determines whether G has a cycle containing e.
15. Given an undirected graph $G = (V, E)$, prove that S is an independent set if and only if $V - S$ is a vertex cover.

Chapter 2

Amortized Analysis

分摊分析

In a sequence of operations, the worst-case time does not occur often in each operation; some operations may be cheap, some may be expensive. Therefore, a traditional worst-case per operation analysis can give an overly pessimistic bound. When the same operation takes different times, how can we accurately calculate the runtime complexity? Amortized analysis gives the average performance (over time) of each operation in the worst case. Amortized analysis is not average case analysis. In average case analysis we compute the expected cost of each operation. Amortization is a technique used by accountants to average a large one-time expense over a long period of time. There are generally three methods for performing amortized analysis:

1. The aggregate method computes the upper bound $T(n)$ on the total cost of n operations. The amortized cost is given by $T(n)/n$. In this method each operation will get the same amortized cost, even if there are several types of operations in the sequence.
2. The accounting method (or the banker's method) computes the individual cost of each operation. We assign different charges to each operation; some operations may charge more or less than they actually cost. The amount we charge an operation is called its amortized cost.
3. The potential method (or the physicist's method). We won't use a potential method in this course.

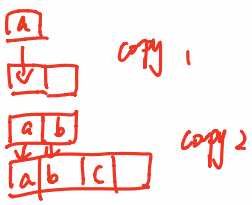
2.1 Unbounded Array

The general implementation strategy: We maintain an array of a fixed length limit and an internal index size, which tracks how many elements are actually used in the array. When

we add a new element, we increment size; when we remove an element, we decrement size. How do we proceed when the array is full and we need to add another element? At that point, we allocate a new array twice as large and copy the elements we already have to the new array. So, if the current array is full, the cost of insertion is linear; if it is not full, insertion takes a constant time. In order to make the analysis as concrete as possible, we will count the total number of inserts and the number of copy operations. In this model we won't analyze deletions (see exercise 3 for insertions and deletions). In table 2.1, we record the current size of the array, its new size, the number of insets, and the number of copies. The table shows that 9 inserts require $1 + 2 + 4 + 8 = 15$ copy operations. Therefore, the amortized cost of a single insert is the total cost $(9 + 15 = 24)$ over 9 inserts, which is 2.67.

TABLE 2.1 The cost of insertions

Insert	Old size	New size	Copy
1	1	—	—
2	1	2	1
3	2	4	2
4	4	—	—
5	4	8	4
6	8	—	—
7	8	—	—
8	8	—	—
9	8	16	8



Let us generalize the pattern. Assume we start with the array of size 1 and make $2^n + 1$ inserts. These inserts will require $1 + 2 + 4 + \ldots + 2^n = 2^{n+1} - 1$ copy operations. Thus, the total work (inserts plus copies) is given by $(2^n + 1) + (2^{n+1} - 1) = 3 \times 2^n$. Next, we compute the average cost per insert as a limit when the input size tends to infinity:

$$\lim_{n \to \infty} \frac{3 \cdot 2^n}{1 + 2^n} = 3$$

We say that the amortized cost of insertion is constant, namely $O(3)$. Such method of analysis is called an *aggregate* method. The aggregate method seeks an upper bound on the total running time of a sequence of operations.

Let us compute the amortized cost of insertion using the *accounting* method. This method seeks a payment for each individual operation. Intuitively, we maintain a bank account and each operation is charged to it. Some operations are charged very little but also generate a surplus. Others drain the savings. The balance in the bank account must always remain positive.

We will assign a dollar token to each operation. It costs a token to insert an element and another token to move it when we need to double the array size. It follows we have to assign at least 2 tokens to each insert: we pay one token to perform an operation, and we put one token into the bank. Figure 2.1 demonstrates the insertion process starting with an array of size one and an empty bank account.

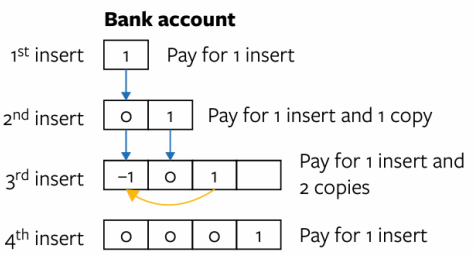


FIGURE 2.1 Out-of-pocket cost per insert is 2 tokens.

In that picture we see that after third insertion, the bank account is empty, and after fourth insertion, the bank account has only one token. On the next, fifth insertion, we need to double the array size from 4 to 8. Clearly, we do not have enough money in the bank to pay for it.

Let us increase the number of tokens for each insert to three tokens: we pay one token to perform an operation, and we put two tokens into the bank. Figure 2.2 demonstrates the insertion process.

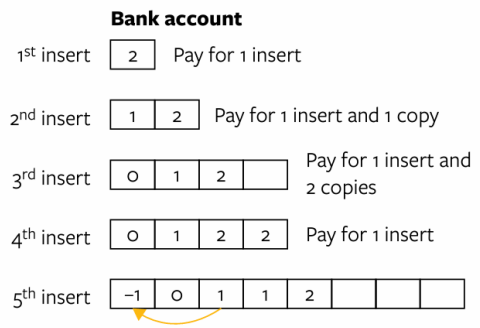


FIGURE 2.2 Out-of-pocket cost per insert is 3 tokens.

Now the bank has enough money to perform fifth insertion. In the next few inserts (6th–8th) we generate surplus.

In the next insert, we drain our savings. How do we know there will be enough money in the bank to pay for moving when we need to double the array size?

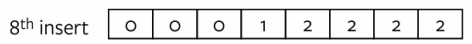


FIGURE 2.3 Bank account after 8th insertion.

Doubling the array size say from N to $2N$, we need at least N tokens in the bank. Those N extra tokens will be generated by $N/2$ new inserts. Therefore, assigning three tokens per insert, we were able to pay for all the operations. This proves that our amortized cost is at most three.

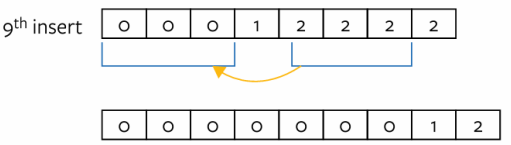
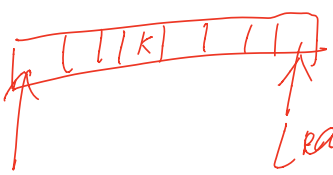


FIGURE 2.4 Bank account after 9th insertion.

number n with $\log n$ bits



Least significant Bit
$\parallel$
$2^0 = 1$

Most Significant Bit
$\parallel$
$2^{\log n} = n$

cost of each flips is 2^k

find AC (average cost) of each flips

1. $O(n)$

$$n \cdot 2^0 + \frac{n}{2} \cdot 2 + \frac{n}{4} \cdot 2^2 + \cdots + 2 \cdot n = O(n \log n)$$

$$\text{AC (per flip)} = \frac{O(n \log n)}{n} = O(\log n)$$

2.2 Binary Counter

Given a binary number with $\log(n)$ bits, stored as an array, where each entry $A[i]$ stores the i-th bit, the cost of incrementing a binary number is the number of bits flipped. We use the standard way of incrementing the counter, which is to toggle the lowest order bit. What is the amortized cost per increment? As an example, consider 3-bit numbers and count the number of flips for each increment.

Table 2.2 shows that incrementing 000 requires a single flip, incrementing 001 results in two flips, and incrementing 111 results in 3 flips.

Clearly, in the worst-case all bits are flipped, so the cost per increment is $O(\log n)$. Now suppose we increment n times, starting with a zero-binary number. If we only use the worst-case running time for each increment, we get an upper bound of $O(n \log n)$. Although this bound is correct, we can do better.

flips

000	1
001	2
010	1
011	3
100	1
101	2
110	1
111	3

TABLE 2.2 The number of flips for 3-bit numbers

	# of flips
000	1
001	2
010	1
011	3
100	1
101	2
110	1
111	3

2.2.1 The Aggregate Method

Let us think about how often we flip a single bit. Consider the least significant bit. Each time we increment a binary number, that bit is changed. Thus, the number of times the bit changes is n. Consider the next significant bit. How often is it toggled? $n/2$ times. The next bit is toggled $n/4$ times, and so on. The most significant bit is toggled only twice. Thus, the total cost is given by

+16 变1 +4 变一 +1 变1

+2 变1 +8 变一 +2 变一次

$$\sum_{i=0}^{\log n} \frac{n}{2^i} < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

$$n+\frac{n}{2}+\frac{n}{4}+\ldots+2=n\left(1+\frac{1}{2}+\frac{1}{4}+\ldots+\frac{2}{n}\right)\leq n\sum_{k=0}^{\infty}\frac{1}{2^k}=2n.$$

It follows the amortized cost per increment is $O(2)$.

TABLE 2.3 Bank account

	Bank
000	0
001	1
010	1
011	2
100	1
101	2
110	2
111	3

2.2.2 The Accounting Method

run time 为这段资源

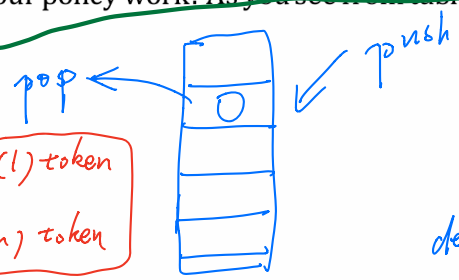
The key point to observe is that each increment has exactly one $0 \to 1$ flip. But different increments have different numbers of $1 \to 0$ flips. Our accounting policy is the following: Every time you flip $0 \to 1$, pay the actual cost of 1, plus put 1 into a bank; every time you flip $1 \to 0$, use the money in the bank to pay for that flip. Consider 3-bit numbers and count the number of $1 \to 0$ flips for each increment. Why does our policy work? As you see from table 2.3, our bank

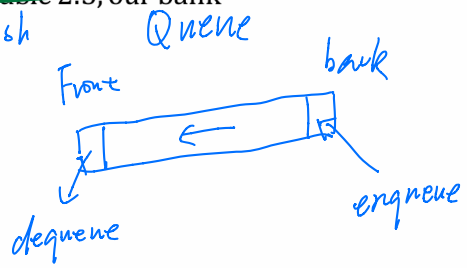
$0 \to 1$: +2 → 1 pay the move; → store in bank for pop pay

$1 \to 0$: -1

Method

1. How many tokens O(3) to enqueue? O(1) token
2. How many tokens O(1) to dequeue? O(n) token

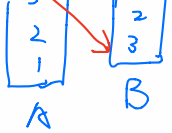




enqueue: 1, 2, 3

A.push(...)

Amortized Dictionary



dequeue : A.pop B.push loop

Cost O(n)

dequeue : B.pop()

account has as many tokens as the number of 1 bit. This shows that we have enough tokens in the bank to pay for future $1 \rightarrow 0$ flips.

2.3 Amortized Dictionary

One of the most important structures in computer science is the *dictionary* data structure that supports fast insert and search operations. Here we will discuss a dictionary based on linked lists and sorted arrays. The idea of this data structure is as follows. We will have a linked list of arrays, where array k has size 2^k, and each array has a unique size and is in sorted order. Whether arrays are full or empty is based on the binary representation of the number of items we are storing. For example, with 11 items our dictionary might look like this: $(11 = 1 + 2 + 8)$.

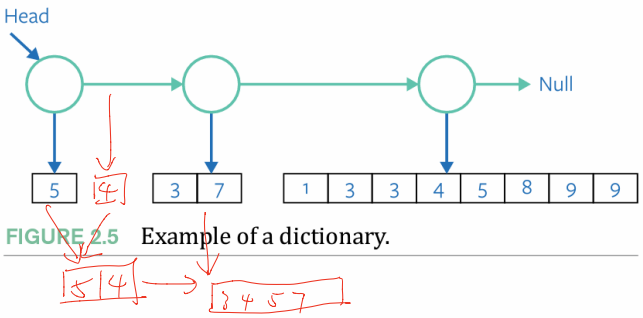


FIGURE 2.5 Example of a dictionary.

In general, we need at most ceiling(log(n)) arrays to store n items. How do we insert into this data structure? We create an array of size one and add to the linked list. Since each array must have a different length, insertion requires merging arrays of the same size. As an example, consider inserting 4 into the dictionary in figure 2.5. We get two arrays of size one; thus, we have to merge them. After merging, the dictionary will have two arrays of size two: [4, 5] and [3, 7]. We have to merge them into an array of size four. Figure 2.6 demonstrates the final dictionary.

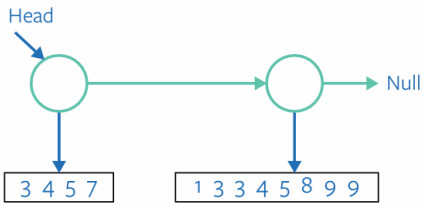


FIGURE 2.6 A dictionary after inserting 4 to it.

In the worst case we may merge all $O(\log(n))$ sorted arrays. Each pair of sorted arrays can be merged in linear time. The total cost model is the following: Creating the initial array of size 1 costs 1, and merging two arrays of size k costs $2k$. The total cost of this insert is $1 + 2 + 4 = 7$. In the general case, the total cost per insert is given by

$$1 + 2 \cdot 2^0 + 2 \cdot 2^1 + 2 \cdot 2^2 + \ldots + 2 \cdot 2^{\log n} = O(n).$$

1+2+4+...+2·n =

Therefore, the worst-case runtime complexity of a single insert is $O(n)$. However, on average we do not merge all $O(\log n)$ arrays. What is the amortized cost of insertion? First, we note that each array is a power of 2. Then we observe that adding a new item to the dictionary is equivalent to a bit increment. However, the cost of incrementing is not a constant anymore. Its cost equals to a cost of merging two sorted arrays. It follows that the cost of flipping the k-th bit is 2^k. Consider the least significant bit ($k = 0$). The number of times this bit changes is n, with the cost 2^0. For the next bit ($k = 1$), the cost is 2^1. For the most significant bit ($k = \log(n)$), the cost is $2^{\log n}$. Thus, the total cost of n inserts is given by

$$n \cdot 2^0 + n/2 \cdot 2^1 + n/4 \cdot 2^2 + \ldots + 2 \cdot 2^{\log n} = n + n + \ldots + n = O(n \log n).$$

We have proved that the amortized dictionary data structure has amortized cost $O(\log n)$ per insert.

2.4 Amortized Trees

二分树

Recall that a binary search tree is not necessary balanced; therefore, it does not guarantee $O(\log n)$ insertion and searching time that could be in the worst case as bad as $O(n^2)$. There are several ways to make a search tree balanced, though in this section we consider a different approach. Suppose we search a tree multiple times. Don't we want a previous search somehow to affect the next search? Ideally, we want a data structure that adjusts itself to accommodate the observed sequence of operations. The splay tree is a variant of a binary search tree that is designed to do exactly that. The intuition behind splay trees is based on the following observation: If an item was searched once, it is most likely to be searched again. Therefore, the splay tree heuristic is to move a searched item to the root, so that next time the item is searched it would take almost a constant time. Splay trees give up a tree balance in favor of taking advantage of the fact that a large percentage of the searches is caused by only a small subset of data. Splay trees have been introduced by D. Sleater and R. Tarjan in 1985.

伸展树

The key operation performed on a splay tree is the *splay* operation. `splay(N)` is moving a node `N` to the root via a sequence of rotations that preserves the binary search

tree ordering invariant. Every time a node is accessed in a splay tree, it is moved to the root of the tree. However, splaying is done in a very special way that guarantees $O(\log n)$ amortized bound.

The rotation depends on the positions of the current node N, its parent P, and its grandparent G. There are six types of rotations:

Zig (Zag): A single right (left) rotation. It can only occur when the N node's parent is the root of the tree. This rotation moves the current node N one level up, so N becomes the root.

only P is root

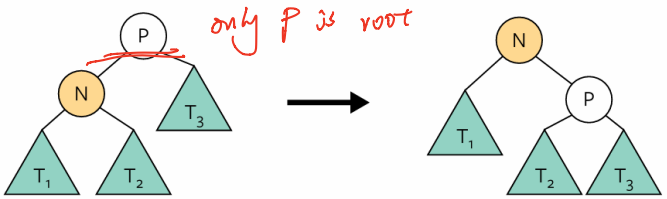


FIGURE 2.7 Zig rotation.

Zig-Zag (Zag-Zig): A double rotation formed by a single Zig (Zag) followed by Zag (Zig) rotation. In the first rotation we rotate N and P. Node N moves a level up and becomes a child of G node. In the next rotation we rotate N and G. Node N moves again a level up, so that nodes G and P become children of N.

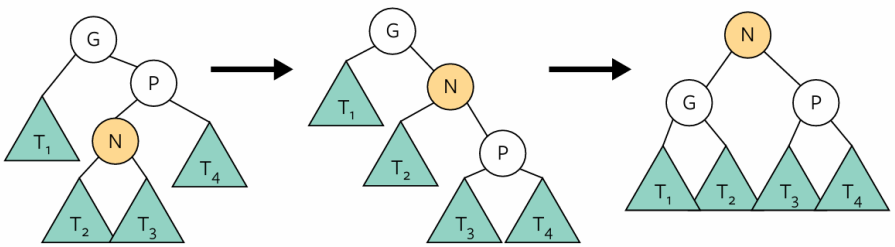


FIGURE 2.8 Zig-Zag rotation.

Zig-Zig (Zag-Zag): A double rotation formed by two single Zig (Zag) rotations in special order. This rotation occurs when N and its parent P are both left (right) children. First, we rotate P and G, and then N and P.

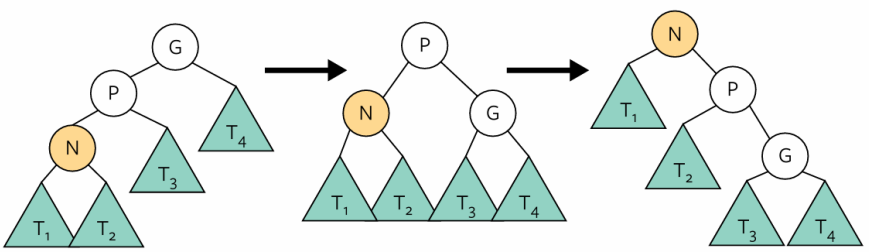


FIGURE 2.9 Zig-Zig rotation.

Note the order of rotations in Zig-Zig rule does matter. Alternatively, we may think to first rotate N and P and then N and G. These rotations will form the tree in figure 2.10. The difference between figures 2.9 and 2.10 might not seem to be that important, but the next example will demonstrate that without this rule, the amortized cost of search is linear.

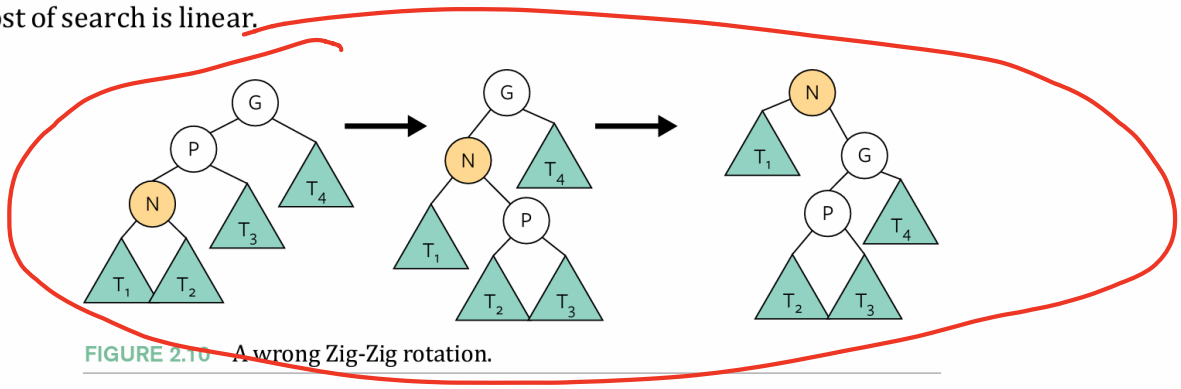


FIGURE 2.10 A wrong Zig-Zig rotation.

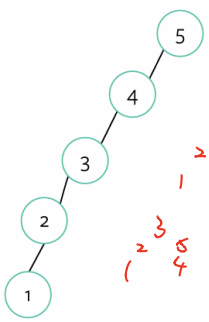


FIGURE 2.11 A linked list of 5 elements.

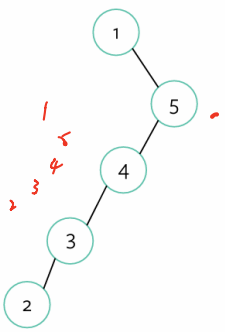


FIGURE 2.12 The tree after splay(1) has taken place.

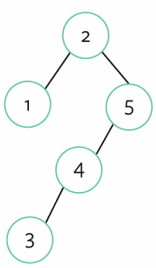


FIGURE 2.13 The tree after splay(1) and splay(2) have taken place.

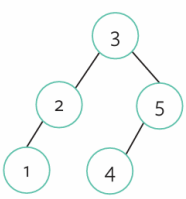


FIGURE 2.14 The tree after splay(1), splay(2), and splay(3) has taken place.

Let us consider an example (figure 2.11), where a splay tree is a linked list of ordered elements from 1 to 5. On this tree we will perform the following sequence of operations: `splay(1)`, `splay(2)`, `splay(3)`, `splay(4)`, `splay(5)`. Each splay operation will move a node to the root by performing a chain of single rotations, Zig or Zag. We will use the aggregate method to compute the amortized cost per splay.

To promote node 1 to the root we have performed 4 single Zig rotations.

To promote node 2 to the root we have performed 3 single Zig rotations and one Zag rotation.

To promote node 3 to the root we have performed 2 single Zig rotations and one Zag rotation.

The next two splays, `splay(4)` and `splay(5)`, will require two and one rotations, respectively. Therefore, the total number of single rotations for 5 splay operations is $4 + 4 + 3 + 2 + 1 = 14$.

Let us generalize this example. Suppose we started with a linked list of ordered items from 1 to n. We run a sequence of n splay operations: `splay(1)`, `splay(2)`, ..., `splay(n)`. Proceeding as in the previous example, the first splay takes $n-1$ single rotations, `splay(2)` also takes $n-1$ single rotations, `splay(3)` takes $n-2$ rotations, and so on. The total number of rotations is given by

$$(n-1)+(n-1)+(n-2)+\ldots+2+1=-1+\sum_{k=1}^{n}k=-1+\frac{n(n+1)}{2}=O(n^2).$$

It follows that the amortized cost per splay is $O(n^2)/n = O(n)$. This example demonstrates that in order to achieve $O(\log n)$ amortized bound per splay, we have to have Zig-Zig (Zag-Zag) rotation.

The analysis of running time of splay trees is quite difficult. Any single insert or search might take a linear time in the worst case. But any sequence of m operations on a tree with n nodes takes $O(m \log n)$ time. The proof is far beyond the scope of this book.

average for single input
amortized for sequence input

REVIEW QUESTIONS

1. What is the definition of the amortized cost using the aggregate method?
2. F **(T/F)** Amortized analysis is used to determine the average runtime complexity of an algorithm. False
3. **(T/F)** Compared to the worst-case analysis, amortized analysis provides a more accurate upper bound on the performance of an algorithm. True
4. F **(T/F)** The total amortized cost of a sequence of n operations gives a ~~lower~~ upper bound on the total actual cost of the sequence. False
5. T **(T/F)** Amortized constant time for a dynamic array is still guaranteed if we increase the array size by 5%. True
6. F **(T/F)** If an operation takes $O(1)$ expected time, then it takes $O(1)$ amortized time. False
7. Suppose you have a data structure such that a sequence of n operations has an amortized cost of $O(n \log n)$. What could be the highest actual time of a single operation? amor $= \frac{n\log n}{n} = \log n$ $\leq n\log n$ 就可以
8. What is the worst-case runtime complexity of searching in an amortized dictionary?

A. $O(n\log n)$
~~B $O(n^2)$~~
~~C $O(n^2\log n)$~~
D $O(\log n)$
E $O(n)$

$n-1+n^2$ $n^2\log n$.
$n\cdot n\log n$
n $n^2\log n$

2.

0	1	2	...	log n-1
1	2	3	...	log n

bit cost

$n \quad \frac{n}{2} \quad \frac{n}{4} \quad 2$ flips

$n \cdot 1 + \frac{n}{2} \cdot 2 + \frac{n}{4} \cdot 3 + \dots + 2 \log n$

$= n \sum_{k=0}^{\log n - 1} \frac{(k+1)}{2^k}$

$\leq n \cdot \sum_{k=0}^{\infty} = O(n)$

ac per $O(n)/n = O(1)$

EXERCISES

1. You have a stack data type, and you need to implement a FIFO queue. The stack has the usual POP and PUSH operations, and the cost of each operation is 1. The FIFO has two operations: ENQUEUE and DEQUEUE. We can implement a FIFO queue using two stacks. What is the amortized cost of ENQUEUE and DEQUEUE operations?
2. We are incrementing a binary counter, where flipping the i-th bit costs $i + 1$. Flipping the lowest-order bit costs $0 + 1 = 1$, the next bit costs $1 + 1 = 2$, the next bit costs $2 + 1 = 3$, and so on. What is the amortized cost per operation for a sequence of n increments, starting from zero?
3. We have argued in the lecture that if the table size is doubled when it's full, then the amortized cost per insert is acceptable. Fred Hacker claims that this consumes too much space. He wants to try to increase the size with every insert by just two over the previous size. What is the amortized cost per insertion in Fred's table?
4. This table supports inserts as well as deletions. The protocol is the following: If an array is full, we double its size on insertion; if an array is 1/4 full, we halve the array size on deletion. Show that the amortized cost of insert and delete is 5.
5. Suppose we perform a sequence of n operations on a data structure in which the i-th operation costs i if i is an exact power of 2 and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.
6. Suppose we perform a sequence of n operations on a data structure in which the i-th operation costs i if i is an exact power of 4 and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.
7. A MultiStack data structure has the usual POP and PUSH operations, and the cost of each operation is one unit. Additionally, it has MULTIPOP(k) operation that removes k recently pushed items. If k is bigger than the stack size, it removes all items. We wish to analyze the running time for a sequence of n PUSH, POP, and MULTIPOP operations, starting with an empty stack. What is the worst-case complexity for a sequence of n operations? What is the amortized cost per operation? Use the accounting method.
8. Consider a singly linked list as a dictionary that we always insert at the beginning of the list. Now assume that you may perform any number of insert operations but will only ever perform at most one lookup operation. What is the amortized cost per operation?

Chapter 3

Heaps

1. sort (nlogn) greedy algorithm
2. find Max O(n)
3. heap O(logn)

PQ
1. insert
2. delete Min

Heap implement

HEAPS ARE ONE OF THE MOST important data structures, especially for implementing greedy algorithms using a priority queue. Heaps provide a great option over sorting when input data changes during an algorithm execution. Sorting as we know is a process of arranging elements according to their priorities. However, in many applications we do not need a full sorted order, just the ability to access an element with the highest priority. We start the chapter with classical binary heaps and then extend the definition to amortized heaps that provide the constant amortized cost for insertion and decreaseKey operations. We will be using heaps in a few applications, namely finding the shortest path in graphs, building the minimum spanning tree, and constructing Huffman encoding.

3.1 Binary Heaps

Binary heaps are based on the notion of a complete binary tree. A complete binary tree is a binary tree where each level is completely filled with nodes, except the gap at the bottom level, which is filled from left to right, as illustrated in figure 3.1.

A complete tree with n nodes has a height of floor(log($n+1$)). In this example, the tree height is 3. Note that the height of the root is 0.

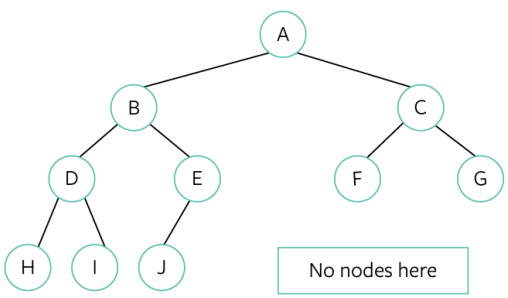


FIGURE 3.1 Example of a complete tree.

height = log (n-1)

A binary heap satisfies one of the following two heap ordering properties:

- *The min-heap property*: The value of each node is greater than or equal to the value of its parent
- *The max-heap property*: The value of each node is less than or equal to the value of its parent

In this course the word *heap* will always refer to a min-heap, unless otherwise noted. Note that a heap may have duplicate elements. To sum up, formally a binary heap can be defined as a collection of items 集合 that satisfy the following invariants:

- *Structural property*: States that a heap is a complete tree
- *Ordering property*: The key of the parent node less or equal than the key of children nodes

A heap supports the following operations:

- insert
- deleteMin
- decreaseKey
- build
- meld (merge two heaps)

These operations will be discussed in the subsequent sections. But first, let us discuss heap implementation.

3.1.1 Implementation.

A heap is uniquely represented by storing its data in an array by running a level-order traversal on a tree, with the root at index 1. This allows fast access to each heap element.

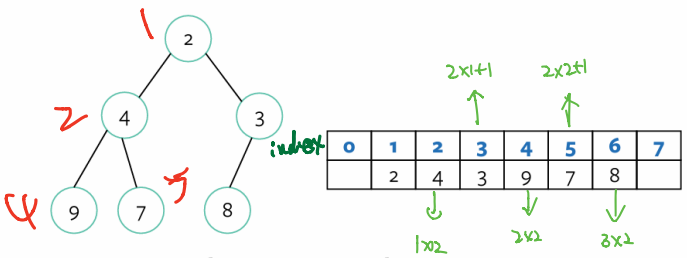
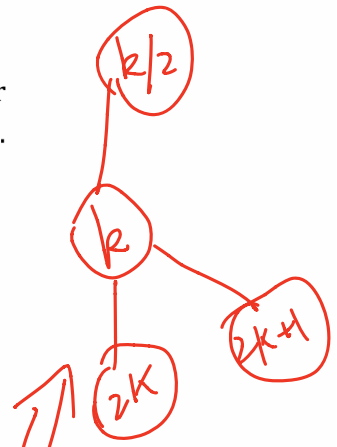


FIGURE 3.2 A heap represented as an array.



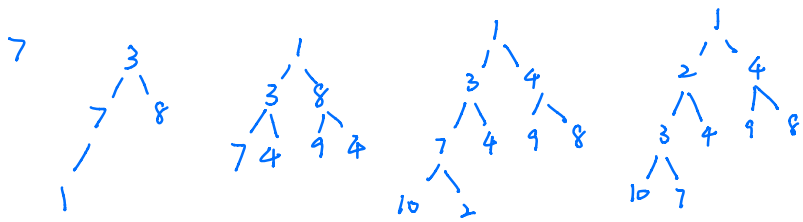
Observe that if a node's index is k, its left child is located at $2k$ index, its right child is located at $2k+1$ index, and its parent is located at $k/2$ index. Array index 0 is left empty to make the indexing work easily.

Current node is k { it's left child is located at 2*k index
right child is located at 2*k+1 index
parents is located at k/2 index

Build Heap

Given an array, turn it into heap

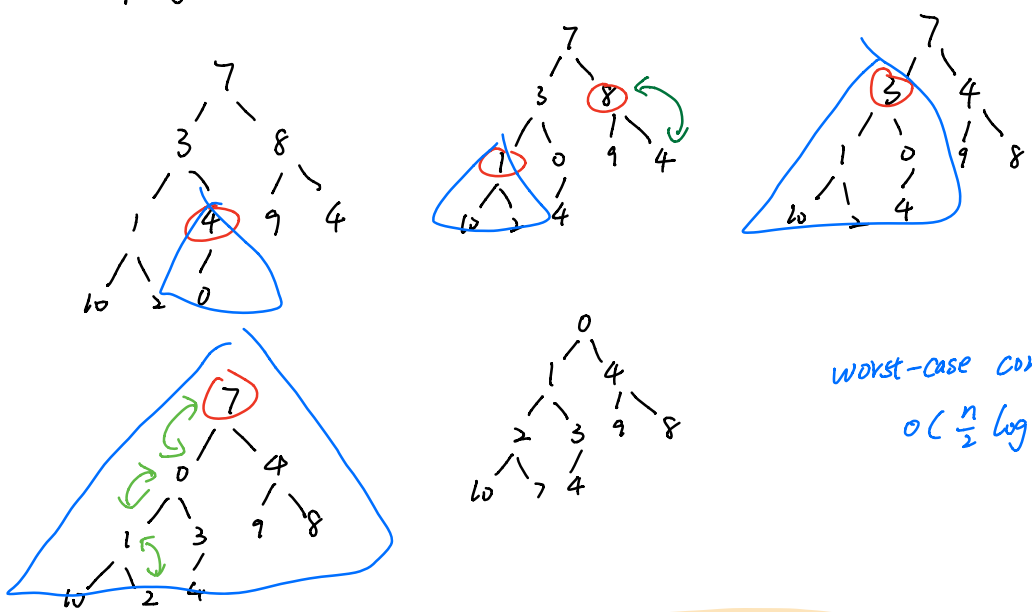
7, 3, 8, 1, 4, 9, 4, 10, 2, 0



worst-case complexity $= O(n \log n)$

$\log 1 + \log 2 \cdots + \log n = \log(n!) = \Omega(n \log n)$

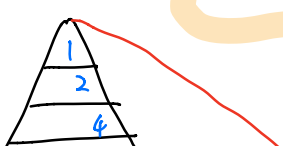
Heapify = 7, 3, 8, 1, 4, 9, 4, 10, 2, 0



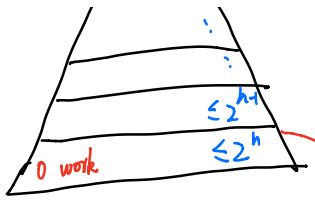
worst-case complexity

$O(\frac{n}{2} \log n)$

Complexity of heapify



we will count the max number of swaps at each level $h = \log(n)$



height	# of nodes	# of swaps
0	1	h
1	2	h-1
2	4	h-2
...	...	
h-1	2^{h-1}	1

$$h\cdot 1 + (h-1)\cdot 2 + (h-2)\cdot 3 + \cdots + 2^{h-1}\cdot 1$$

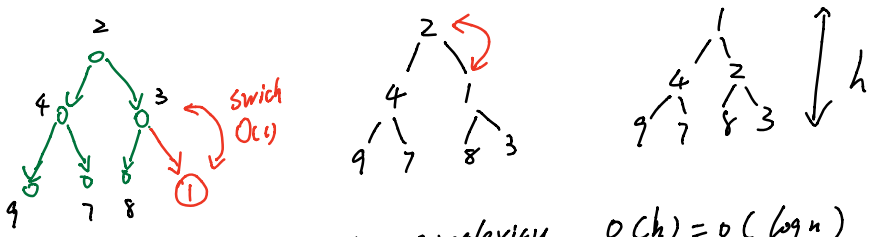
$$= \sum_{k=1}^{h} k\cdot 2^{h-k} = 2^h \sum_{k=1}^{h} \frac{k}{2^k} \le 2^h \sum_{k=1}^{\infty} = O(2^h)$$

$h = \log n$, $2^h = n$

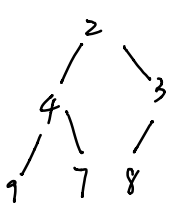
$T(n) = O(n)$

(total complexity..)

insert (tree reps)



worst-case complexity $O(h) = O(\log n)$



0	1	2	3	4	5	6	7
X	2	4	3	9	7	8	1
X	2	4	1	9	7	8	3
X	1	4	2	9	7	8	3

3.1.2 Insert

O(log n)

The new element is initially appended to the bottom level. If the level is full, we start a new one. In an array-based implementation, we place a new item to the end of the array. This will preserve the structural property.

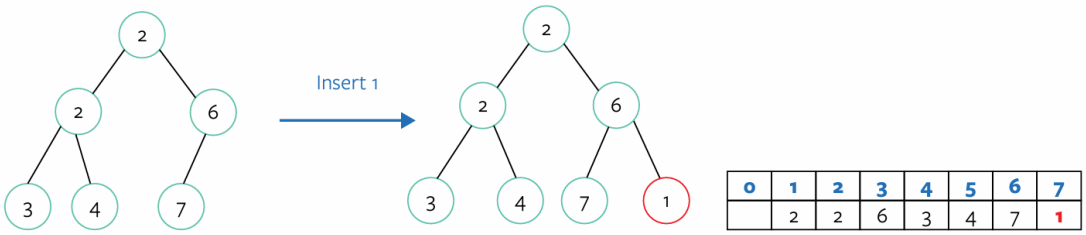


FIGURE 3.3 Inserting 1 into a heap.

违反

At this step, the inserted item may violate the ordering property. We fix this by percolating the item up the tree by swapping positions with the parent, if it's necessary. In figure 3.3 we swap 1 and 6, as shown in figure 3.4.

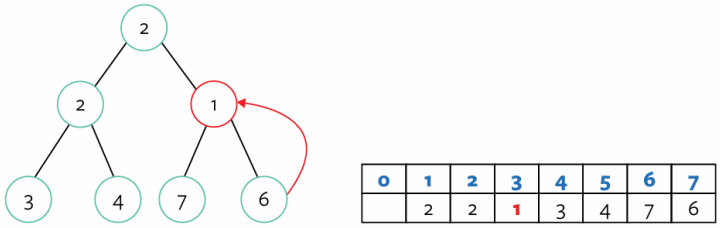


FIGURE 3.4 Swapping 1 and 6.

Again, we observe that new placement of 1 still violates the heap-ordering property. Thus, we swap 1 and 2, as shown in figure 3.5.

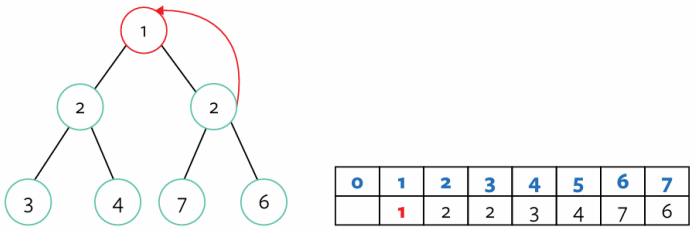


FIGURE 3.5 Swapping 1 and 2.

The worst-case runtime complexity of insertion is $O(\log n)$. This is because a complete tree is a balanced tree and in the worst-case scenario it may require a single swap on each tree level.

3.1.3 DeleteMin

O(log n)

The minimum element can be found at the root of the heap, which is the first element of the array. Clearly, we cannot delete it, since otherwise a tree will be split into two trees. Instead, we move the last element of the heap to the root (this step preserves the structural invariant) and then restore the heap property by percolating it down the tree (this step preserves the ordering invariant). In figure 3.6, we move 8 to the root and then percolate it down by swapping it with the smallest child.

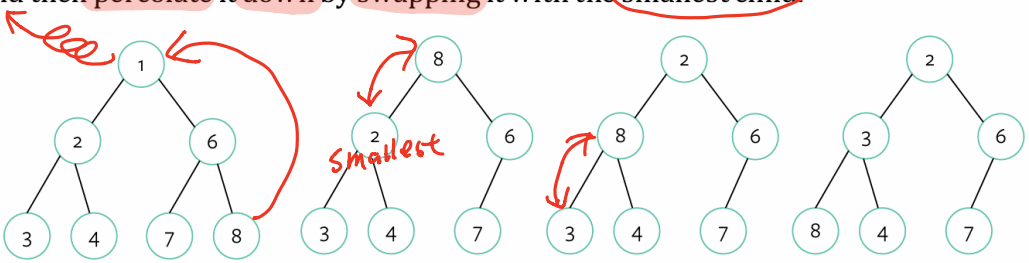


FIGURE 3.6 Deleting the minimum.

This continues until it is less or equal to its children or it reaches the last level. The worst-case runtime complexity of deleteMin is $\boxed{O(\log n)}$, since during percolation it may require a swap on each tree level.

3.1.4 Heapsort

wcr = O(n log n)

If we run deleteMin n times we will get all heap elements in sorted order. This could be implemented in place by storing the deleted element at the end of the array. Figure 3.7 demonstrates one step of the algorithm; we swap 1 with 8, and then percolate 8 down the tree.

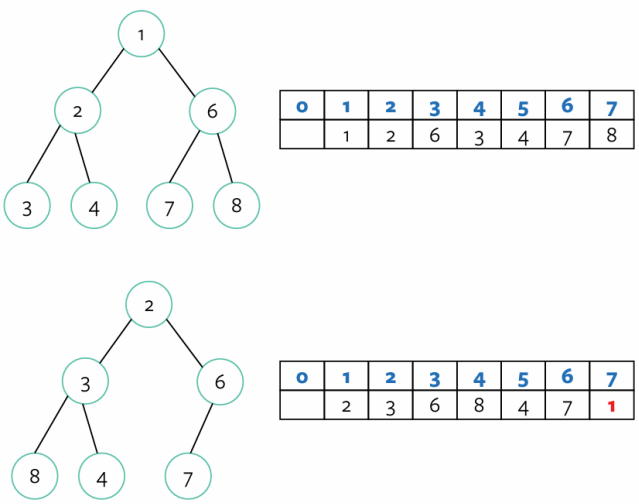


FIGURE 3.7 One iteration of Heapsort.

The worst-case runtime complexity of heapsort is $O(n \log n)$. Heapsort is in place but not stable.

3.1.5 DecreaseKey

wrc = O(log n)

In some algorithms we may require changing the key (value) of one of the heap elements. To restore a heap-ordering property, we may need to percolate this item up. The worst-case runtime complexity of decreaseKey is $O(\log n)$. In figure 3.8, we decrease 7 to 2.

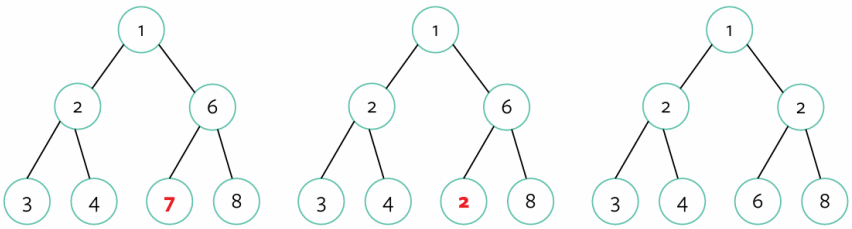


FIGURE 3.8 Demonstration of the decreaseKey operation.

3.1.6 Building a Heap

wcr = Θ(n log n)

1 There are two algorithms to build a heap. The first one is the online algorithm, when the data is not known to us in advance. In this case we build a heap by insertion, starting with an empty array. We will resize the array once it is full. Read about the resizing policy in chapter 2. If we insert n elements, the total cost $T(n)$ is bounded by

$$T(n) = \log 1 + \log 2 + \ldots + \log(n-1) + \log n \leq \log n + \log n + \ldots + \log n + \log n = n \log n.$$

On the other hand (see chapter 1.2 for the proof),

$$T(n) = \log 1 + \log 2 + \ldots + \log(n-1) + \log n = \log(n!) = \Omega(n \log n).$$

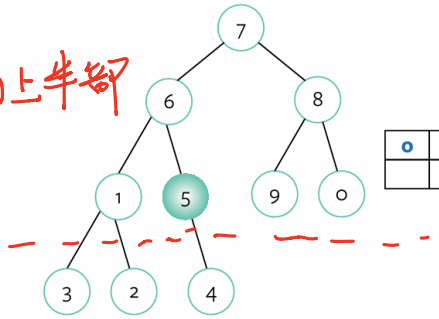
2 Thus, the worst-case runtime complexity of building a heap is $\Theta(n \log n)$. wcr = O(n)

The second algorithm is offline, when the data is known to us in advance. In this case we can develop a faster algorithm. We will call it "heapify," a process of converting a complete tree into a heap. We begin by placing all the elements into an array in given order. Next, starting at position $n/2$ and working toward position 1, we percolate each element down the tree by swapping it with its smallest child.

Let us consider an example of building a heap on the following set of data: 7, 6, 8, 1, 5, 9, 0, 3, 2, 4. First we place the numbers into a complete tree. This will satisfy the structural invariant.

only if we know n

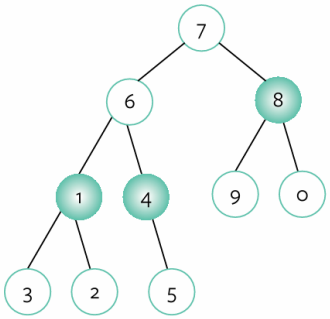
只动上半部



0	1	2	3	4	5	6	7	8	9	10
	7	6	8	1	5	9	0	3	2	4

FIGURE 3.9

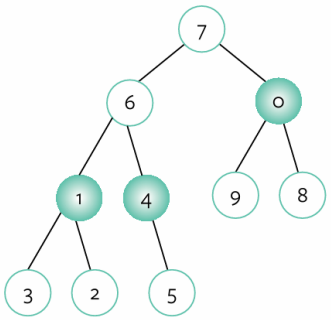
Then we start at the middle (node 5) and swap it with the child 4. Next, we move to 1. There is nothing to swap for that element, so we move to 8.



0	1	2	3	4	5	6	7	8	9	10
	7	6	8	1	4	9	0	3	2	5

FIGURE 3.10 Heapify nodes at depth 2.

The smallest child of 8 is 0, so we swap 8 with 0 and then move to 6.



0	1	2	3	4	5	6	7	8	9	10
	7	6	0	1	4	9	8	3	2	5

FIGURE 3.11 Heapify node 8 at depth 1.

We swap 6 with the left child 1, and then swap 6 again with the right child 2.

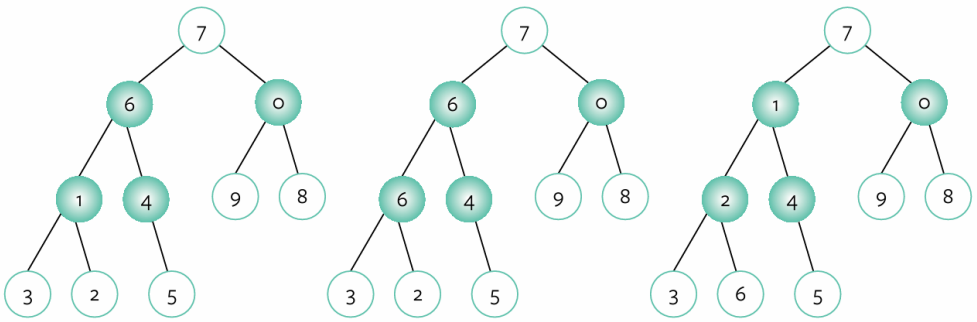
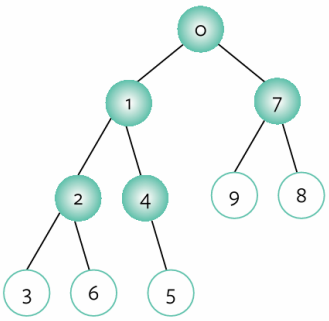


FIGURE 3.12 Heapify node 6 at depth 1.

Finally, we move to the root and swap it with the right child 0. Figure 3.13 shows the final heap.



0	1	2	3	4	5	6	7	8	9	10
	0	1	7	2	4	9	8	3	6	5

FIGURE 3.13 The final heap.

Now we analyze the worst-case complexity of heapify. During the algorithm execution at most $n/2$ heap elements percolate down the heap. Since the each percolation is $O(\log n)$, the total cost is bounded by $O(n \log n)$. But let us note that not each element was percolated down to a leaf. Thus, we shall derive an asymptotically tight bound. We will count the exact number of swaps (in the worst case) at each level. At the root, we may percolate down h times, where h is the tree height. At a level below, we may

not the best way

have at most $(h - 1)$ swaps. And so on. At the last level, there are zero swaps. Figure 3.14 replicates the number of swaps per level:

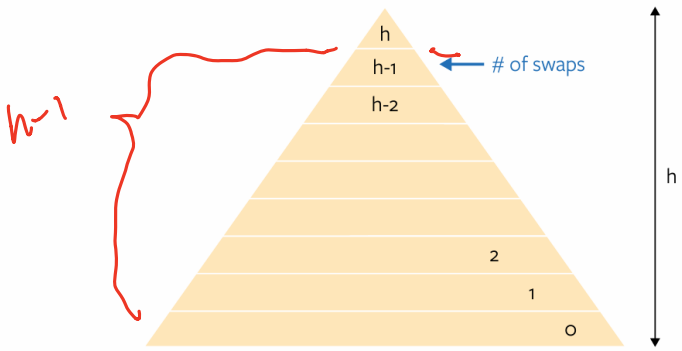


FIGURE 3.14 Demonstrates the number of swaps per level.

Next, we take into account the number of nodes on each level.

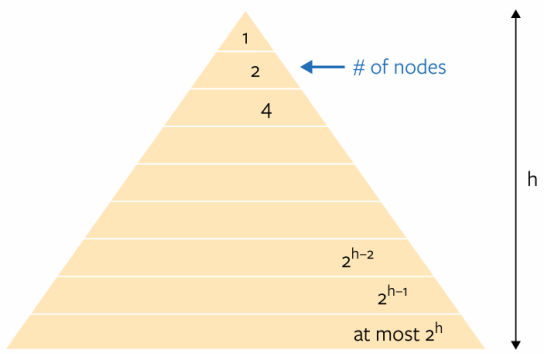


FIGURE 3.15 Demonstrates the number of nodes per level.

We summarize the total number of swaps during the heapification in table 3.1.

TABLE 3.1 The total number of swaps

Height	# of nodes	# of swaps K
0	1	h
1	2	$h - 1$
—	—	—
$h - 2$	2^{h-2}	2
$h - 1$	2^{h-1}	1

Finally, we compute the total work by multiplying the number of swaps by the number of nodes on each level. Let *T(n)* denote the total number of swaps in the worst case. Then, as one can see from table 3.1,

$$T(n) = \sum_{k=1}^{h} k\ 2^{h-k},$$

k = # of swaps

where $h = \log n$. The finite sum can be further simplified as it follows

$$T(n) = \sum_{k=1}^{h} k\ 2^{h-k} = 2^h \sum_{k=1}^{h} \frac{k}{2^k} \leq 2^h \sum_{k=1}^{\infty} \frac{k}{2^k} = 2^h\, 2 = O(n).$$

?

1

This proves that building a heap by running the heapify operation has a linear runtime complexity.

Table 3.2 shows the binary heap operations and their runtime complexities:

TABLE 3.2 Running times for heap operations

Operation	Complexity
findMin	$O(1)$
deleteMin	$O(\log n)$
insert	$O(\log n)$
decreaseKey	$O(\log n)$
buildHeap	$O(n)$

3.2 Binomial Heaps

In the previous section we have proved that insertion takes $O(\log n)$ in the worst case. However, as it easy to see, not all inserts require $\log n$ swaps; some inserts can be performed in constant time. This observation implies that a binary heap *may* exhibit a better amortized complexity of insertion. Let us consider an example of inserting n items in sorted decreasing order from n to 1 into an empty min-heap. We will count the total number of swaps required to insert all the items. For simplicity, we assume that $n = 2^k - 1$. The process of insertion will create a binary heap of height $k - 1 = \log(n+1) - 1$. Obviously, it takes no swaps to insert the first item n. It takes a single swap to each insert for the next two items $n - 1$ and $n - 2$. Finally, it takes $k - 1$ swaps for each element on the last level (the last level contains 2^{k-1} items.) The total work by inserting n items is given by

$$\sum_{m=0}^{k-1} m2^m = O(k2^k) = O(n \log n).$$

每个数m次　　都一层有 2^m 个

$2^k - 1 = n$

$k = \log(n+1)$

height $= k-1 = \log(n+1) - 1$

Then, the amortized cost per insertion in

$$\frac{O(n\log n)}{n} = O(\log n).$$

worst-case complexity

This is the same as the worst-case complexity. This simple mathematical computation demonstrates that a binary heap is not suitable for amortized operations and therefore requires creating a different type of heap.

2

In this section we describe another heap data structure that has a slight improvement in amortized cost over a binary heap. This data structure was introduced by J. Vuillemin in 1978. Each binomial heap is a collection of binomial trees. A binomial tree B_k, of rank k, is defined recursively as follows:

1. B_0 is a single node
2. B_k is formed by joining two B_{k-1} trees

Here are the first four binomial trees:

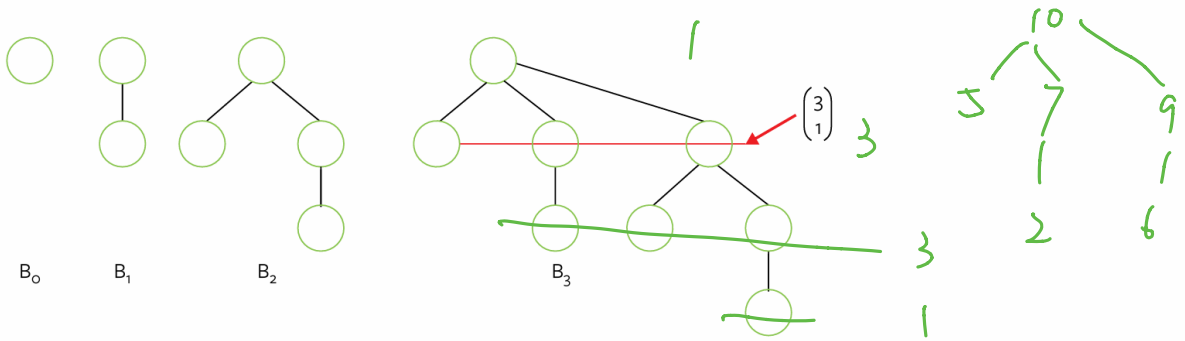


FIGURE 3.16 Example of binomial trees.

B_3 every level = $(1+x)^3 = 1+3x+3x^2+x^3$

The number of nodes on each level l in a binomial tree B_k is defined by binomial coefficients $\binom{k}{l}$, where $0 \le l \le k$. The term *binomial* tree comes exactly from this property. The total number of nodes in B_k is 2^k, as it follows from

$$\binom{k}{0} + \binom{k}{1} + \ldots + \binom{k}{k-1} + \binom{k}{k} = 2^k$$

Another interesting property of a binomial tree B_k is that when we remove the root, the tree will break into k binomial trees $B_0, B_1, \ldots, B_{k-1}$. Figure 3.17 shows a new way of looking at B_3.

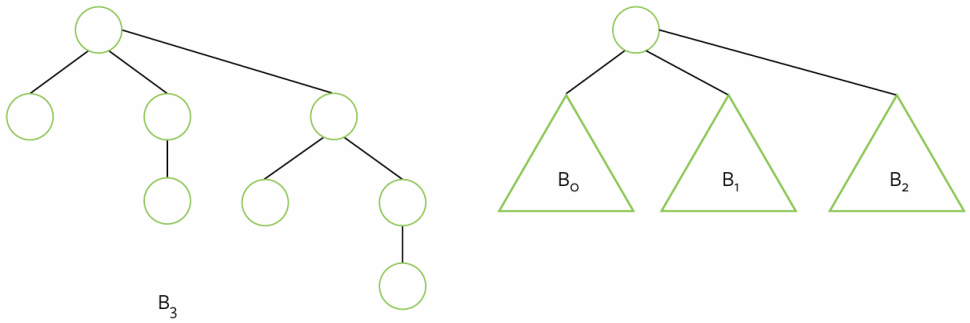


FIGURE 3.17 Binomial tree of rank three.

A binomial heap is a collection (a linked list) of at most *ceiling*(log n) binomial trees in increasing order of size, where each tree has a heap ordering property. In a binomial heap there is at most one binomial tree of any given rank. In order to have constant time access to the top element, we store the pointer to the smallest root. Figure 3.18 demonstrates a binomial heap of 13 elements:

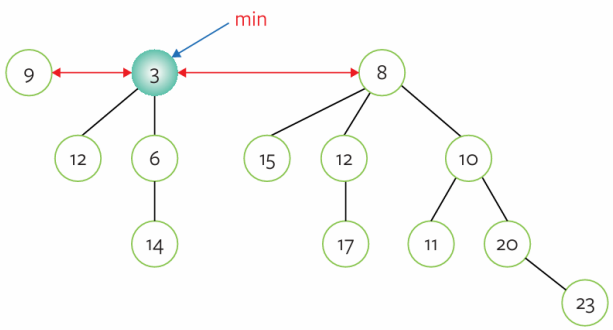


FIGURE 3.18 A binomial heap of size 13.

Observe that the number of elements that can be stored in a heap relates to its binary expansion. To store 13 elements, we need B_0, B_2, and B_3 binomial trees. This is due to the binary expansion $13_{10} = 1101_2$. If we need to store 25 items, the heap will be a collection of B_0, B_3, and B_4, since $25_{10} = 11001_2$. Thus, a binomial heap with n nodes has number of binomial trees equal to the number of 1's bits in binary representation of n. Having this in mind, we always will assume in the worst-case analysis that there are $O(\log n)$ binomial trees in a binomial heap with n nodes.

3.2.1 Merging

wcr = O(log n)

Binomial heaps allow faster merging, compared to binary heaps. Note, binary heaps are complete binary trees, and two complete binary trees cannot easily be linked to

one another. Consider the merging of two binomial heaps on the following example in figure 3.19.

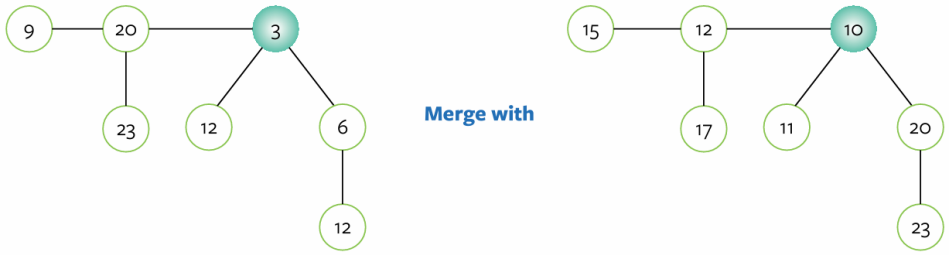


FIGURE 3.19 Merging two Binomial heaps.

First we merge two heaps as we merge two linked lists; it takes $O(1)$ time. We get the heap shown in figure 3.20.

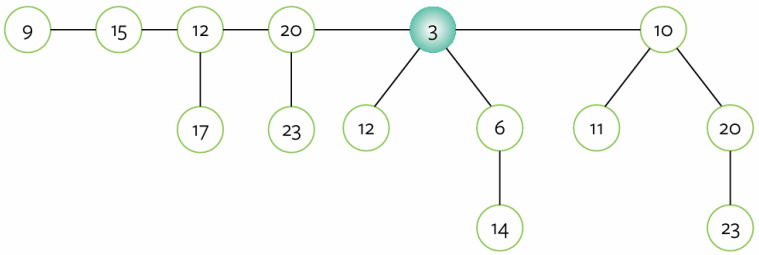


FIGURE 3.20 The result of joining two top linked lists.

This heap is not a binomial heap yet, since it has trees of the same ranks. Thus, we need to combine binomial trees of the same rank. This can be done by making the smaller root the child of the larger root. It also takes $O(1)$ time; however, it may require to merge $O(\log n)$ trees in total. Thus, the worst-case runtime complexity of merging is $O(\log n)$. In our example we need to merge two trees of rank 0 and two trees of rank 1.

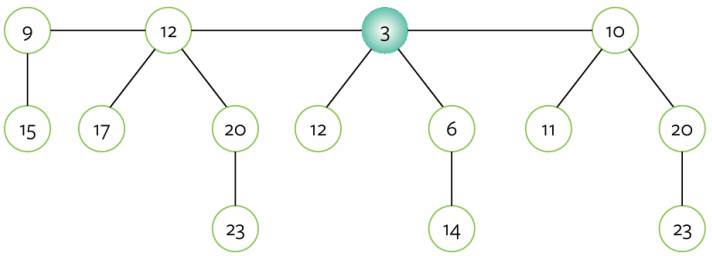


FIGURE 3.21 The result of merging two trees of rank 0 and two trees of rank 1.

Finally, we combine two trees of rank 2 to get the heap in figure 3.22.

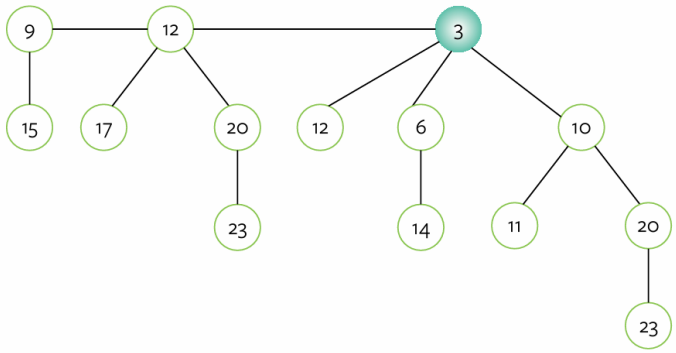


FIGURE 3.22 The result of merging two trees of rank 2.

In conclusion, we note that merging two binomial heaps is related to a binary addition. In the previous example, the result of merging B0B1B2 with B0B1B2 is a binomial heap B1B2B3. This can be viewed as a binary addition

$$\begin{array}{r} 111 \\ 111 \\ \hline 1110 \end{array}$$

where the result of addition 1110 translates into a heap $B_1B_2B_3$.

3.2.2. DeleteMin

wcr = O(log n)

The algorithm is as follows:

1. Find the binomial tree that contains the minimum element
2. Delete the root and move all subtrees to the top list
3. Merge the binomial trees of the same rank
4. Set a pointer to the new minimum

Note that deleting the root of B_k results in $B_0, B_1, \ldots, B_{k-1}$ binomial trees. It follows that the worst-case complexity of deleteMin is $O(\log n)$, which is the same as merging two heaps. Let us execute the algorithm on the heap in figure 3.23.

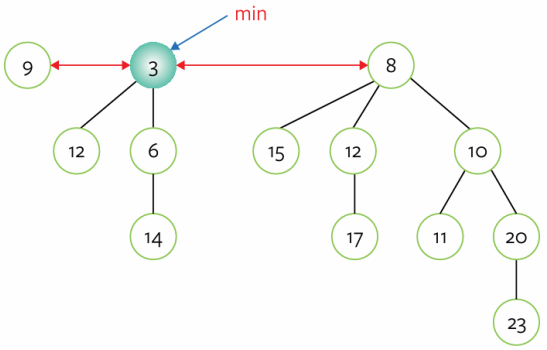


FIGURE 3.23 We will perform deleteMin on this heap.

After deleting the minimum, the heap transforms into what is shown in figure 3.24.

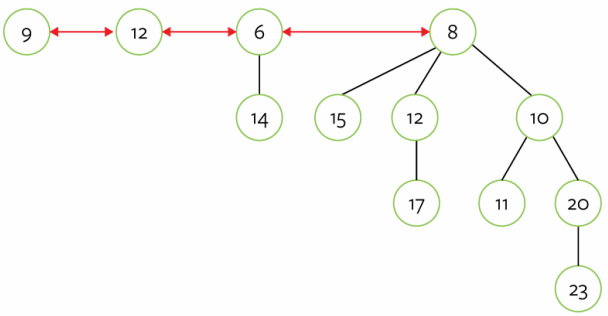


FIGURE 3.24 The result of deleting the min.

Next, we merge two B_0 and B_1 to get what is shown in figure 3.25.

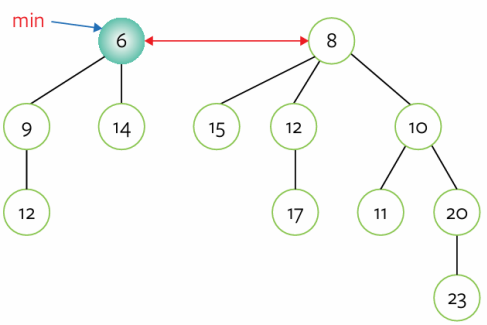


FIGURE 3.25 The result of merging B_0 and B_1.

3.2.3. Insert

Essentially, insertion is merging two heaps, one of size 1 and the other of size n. Therefore, it takes $O(\log n)$ in the worst case. This case occurs only if a binomial heap of size $n = 2^m - 1$ contains binomial trees of all orders, namely the following trees B_0, B_1, ..., B_{m-1}. Then, inserting a new item into this heap will case m binomial trees to merge. The first merge will result in a heap $B_1\, B_1\, B_2 \ldots B_{m-1}$. The second merge will result in a heap $B_2\, B_2\, B_3 \ldots B_{m-1}$. And so on. After $m = O(\log n)$ merges we will get a binomial heap that contains only a single binomial tree of order m.

It should be clear that not each insertion requires merging all $O(\log n)$ binomial trees. Let us compute amortized cost per insertion. We will use the accounting method. We show that assigning two tokens to a single insert is sufficient. Here is our assignment: One token is paid for creating a single binomial tree, and the other is for future tree merging. In this model each binomial tree in a heap has a single token associated with it. When we merge two trees of the same rank, we use one token to pay for merging and keep the second token for the next merger (if it will ever be required.) It follows that single insertion into a binomial heap has a constant amortized cost.

Here is another way to prove that amortized cost of a single insert is constant. Recall that a binomial heap of size n is associated with a binary expansion of n. When we insert a new item into it, we merge two heaps, one of size 1 and the other of size n. This is equivalent to a binary addition, namely incrementing a binary representation of n. We have proved in chapter 2 that amortized cost of binary increment is $O(2)$.

3.2.4 Building a Binomial Heap

We have studied two algorithms of building a binary heap. One is an offline algorithm (building by insertion) with the runtime complexity $O(n \log n)$; the other is an online algorithm (building by heapifying) with the runtime complexity $O(n)$. The cost of building a binomial heap of n elements by insertion is $O(n)$, even if the data is not known to us in advance.

Finally, we summarize runtime complexities of binary and binomial heaps in table 3.3 (here, "ac" stands for amortized cost).

TABLE 3.3 Running times for heap operations

	Binary	Binomial
findMin	$O(1)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$
insert	$O(\log n)$	$O(1)$ (ac)
decreaseKey	$O(\log n)$	$O(\log n)$
merge	$O(n)$	$O(\log n)$

3.3 Fibonacci Heaps

The Fibonacci heap data structure was invented by Fredman and Tarjan in 1987. The general idea is to have a more relaxed structure (compared to binomial heaps) that will improve `decreaseKey` complexity to constant amortized time. The trees in a Fibonacci heap are not constrained to be binomial trees. Figure 3.26 shows an example of a Fibonacci heap.

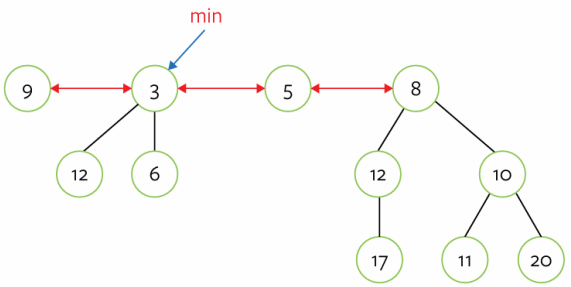


FIGURE 3.26 Example of a Fibonacci heap.

The high-level idea of a `decreaseKey` algorithm is to take the node you want to decrease, change its value, disconnect it and its entire subtree from where it is, and attach it to the tree root list. This is clearly $O(1)$ time. This attractive feature of Fibonacci heaps allows a performance improvement to many algorithms, in particular, the Dijkstra's shortest path algorithm, (see chapter 4.5.1.2) bringing its runtime complexity to $O(E + V \log V)$.

Let us discuss a bird's-eye view of a `decreaseKey` algorithm on the heap in figure 3.27.

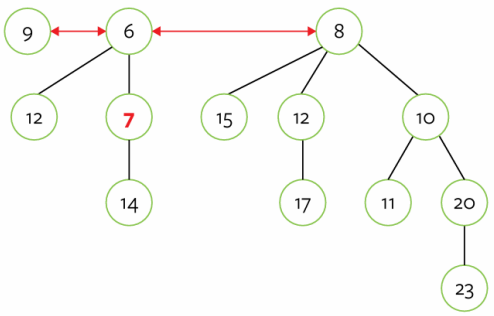


FIGURE 3.27 We will perform decreasekey on this heap.

Suppose we want to change 7 to 5. Since 7 is not the root of the tree, its value decrementing will break the heap order. We cut the tree rooted at 7 from its parent and move it to the top level.

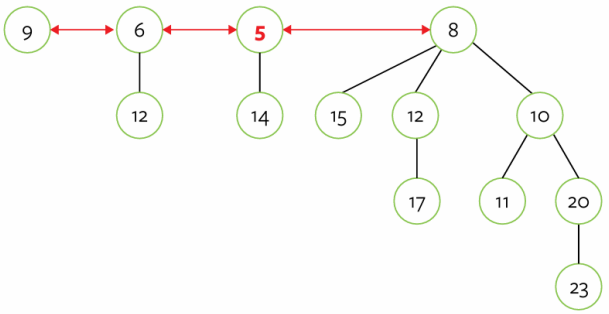


FIGURE 3.28 cut(7).

Running a new function `cut(7)`, we are guaranteed that changing its value to 5 surely does not break the heap order. However, we may end up with having extremely sparse trees of high ranks as well as with several trees of the same rank. In order to avoid this problem, we limit the number of cuts among the children of any vertex to two. This is done by implementing another function `marked(v)` that keeps a track of cuts of all children of v. Clearly, after a call to `decreaseKey` we won't have a binomial tree anymore. We will fix the binomial heap property when `deleteMin` is called. However, the problem is that we can no longer prove the bound on the time of `deleteMin`. That heap may contain more than $O(\log n)$ binomial trees, and some of them are not necessary binomial. There's a clever way to fix this by implementing "cascading cuts." The algorithm was designed by M. Fredman and R. Tarjan. The algorithm is beyond the scope of this course.

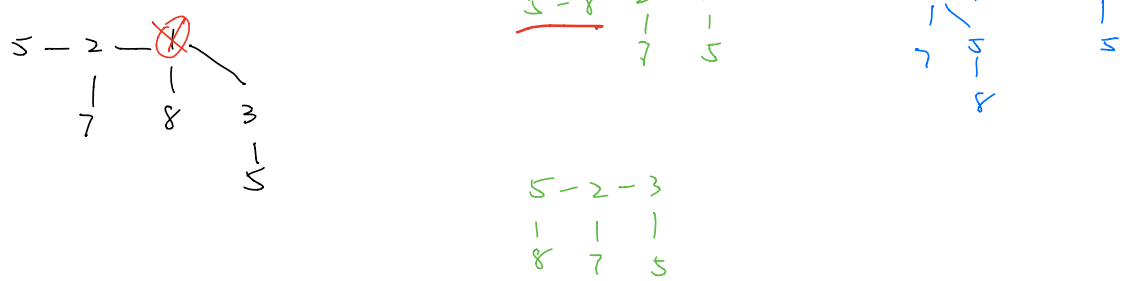
Fibonacci heaps have another advantage: The worst-case time complexity of the insert is $O(1)$. How do we insert into the Fibonacci heap? Just add a single node to the top level. Do not merge binomial trees! We may have several trees of the same rank. We will fix the heap when deleteMin is called. Clearly, lazy insertion runs in $O(1)$ time in the worst case.

In table 3.4, we summarize runtime complexities of different heaps. There "ac" stands for the amortized time complexity.

TABLE 3.4 Running times for heap operations

	Binary	Binomial	Fibonacci
findMin	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
deleteMin	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$ (ac)
insert	$\Theta(\log n)$	$\Theta(1)$ (ac)	$\Theta(1)$
decreaseKey	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$ (ac)
merge	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$ (ac)

Review Question



Review Question

Prove that it is impossible construct a min-heap (not nasecarrily binary) in a comparision-based model with both the following properties:

a delete Min() runs in $O(1)$

b buildHeap() runs in $O(n)$

Sorting $\Omega(n \log n)$

$O(n) + O(1) \cdot n = O(n)$

REVIEW QUESTIONS

1. What is the worst-case runtime complexity of finding the smallest item in a binary min-heap? O(1)
2. What is the worst-case runtime complexity of finding the largest item in a binary min-heap? O(n)
3. How many binomial trees does a binomial heap with 31 elements contain? 5 11111
4. How many binomial trees are in a binomial heap of size n?
5. What is the worst-case runtime complexity of inserting into a binomial heap? 1
6. What is the worst-case runtime complexity of searching in a binomial heap? n log n
7. What is the amortized cost of inserting into a binomial heap? O(2)
8. What is the worst-case runtime complexity of deleteMin() from a binomial heap? log n
9. F (**T/F**) The following array is a max heap: [10, 3, 5, 1, 4, 2]. F
10. ? T (**T/F**) In a binary max-heap with n elements, the worst-case runtime complexity of finding the second largest element is $O(1)$.
11. F (**T/F**) If item A is an ancestor of item B in a heap then it must be the case that the insert(A) operation occurred before insert(B).
12. F (**T/F**) Using a binary heap we can sort any array of size n in $O(n)$ time.
13. T (**T/F**) In a binomial min-heap with n elements, the worst-case runtime complexity of finding the smallest element is $O(1)$.
14. F (**T/F**) In a binomial min-heap with n elements, the worst-case runtime complexity of finding the second smallest element is $O(1)$.
15. ? F (**T/F**) By using a binomial heap we can sort data of size n in $O(n)$ time.
16. T (**T/F**) Given a Fibonacci heap of size n, the maximum number of trees is that heap is n.

EXERCISES

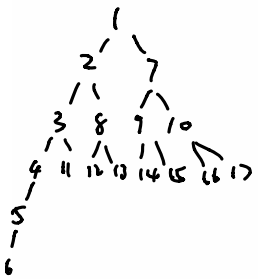
1. Given a sequence of numbers, `3, 5, 2, 8, 1, 5, 2,`
   a. draw a binary min-heap (in an array form) by inserting these numbers, reading them from left to right; and
   b. show an array that would be the result after the call to deleteMin() on this heap.

$6-1$

$n = 2^{h+1} - 1$ $\quad 7 = 8-1$?

$n+1 = 64 = 2^{h+1}$ $\quad h = \log(n+1) - 1 = 7-1 = 6$

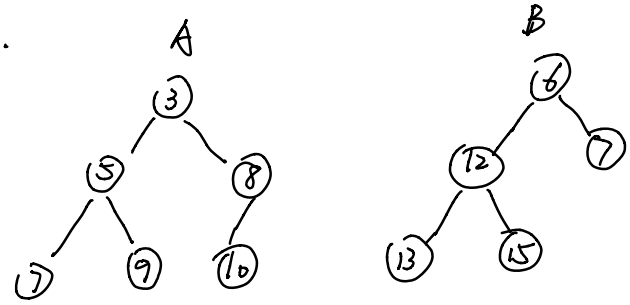
2. Devise an algorithm of merging two binary heaps. What is its runtime complexity?
3. Suppose you have two binary min-heaps, A and B, with a total of n elements between them. You want to discover if A and B have a key in common. Devise an algorithm to this problem that takes $O(n \log n)$ time.



(A) 32
(B) 33
(C) 1
(D) 2
(E) 6

4. The values 1, 2, 3, ..., 63 are all inserted (in any order) into an initially empty min-heap. What is the smallest number that could be a leaf node?
5. Prove that it is impossible construct a min-heap (not necessarily binary) in a comparison-based model with *both* the following properties:
   a. deleteMin() runs in $O(1)$
   b. buildHeap() runs in $O(n)$, where n is the input size
6. Given an unsorted array of size n, devise a heap-based algorithm that finds the k-th largest element in the array. What is its runtime complexity?
7. Recall that two sorted arrays of size n can be merged into a single sorted list in linear time $O(n)$. Suppose there are $k > 2$ sorted arrays, each of size n. Devise a heap-based algorithm that merges k arrays and requires at most $O(k)$ extra space.
8. Given a stream of data (its size is unknown in advance), devise a heap-based algorithm that finds the k-th largest element in the array. Your algorithm must take at most $O(k)$ extra space. What is its runtime complexity?
9. Given a stream of data (its size is unknown in advance), devise a heap-based algorithm that finds the median of elements read so far. What is its runtime complexity?
10. Given a sequence of numbers, `3, 5, 2, 8, 1, 5, 2, 7,`
    a. draw a binomial heap by inserting these numbers, reading them from left to right; and
    b. show a heap that would be the result after the call to deleteMin() on this heap.
11. Discuss the relationship between inserting into a binomial heap and binary increment.
12. Discuss the relationship between merging two binomial heaps and adding two binary numbers.
13. Discuss the relationship between inserting into a binomial heap and a Fibonacci heap.
14. Devise an algorithm of deleting any item from a binomial heap. What is its runtime complexity?
15. Devise an algorithm to find all nodes less than some given value X in a binomial heap. Analyze its complexity.

2.



A. delMin

A. delMin

B. delMin

BST, inorder $O(n)$

Discussion problem 3

How would you sort using a binary heap?

What is it runtime complexity?

Run deleteMin n times, $O(n \log n)$

Space Complexity = to store sorted data $O(n)$

$\boxed{\text{in-place}}$

stable sort?

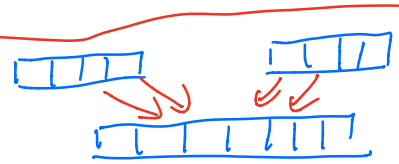
Discussion Problem 4

How would you merge two binary min-heaps?

What is it runtime complexity?

#1 B. insert (A. delete Min) $O(n \log n)$

#2



$O(n)$

heapify $O(n)$

Discussion Problem 5

Given a unstored array of size n. Devise a heap-based algorithm that finds the k-th largest element in the array. What is its runtime complexity?

offline algorithm:

1. build max-heap of n elements $O(n)$
2. call deleteMax k times $O(\log n)$

$O(n + k \cdot \log n)$

Online algorithm:

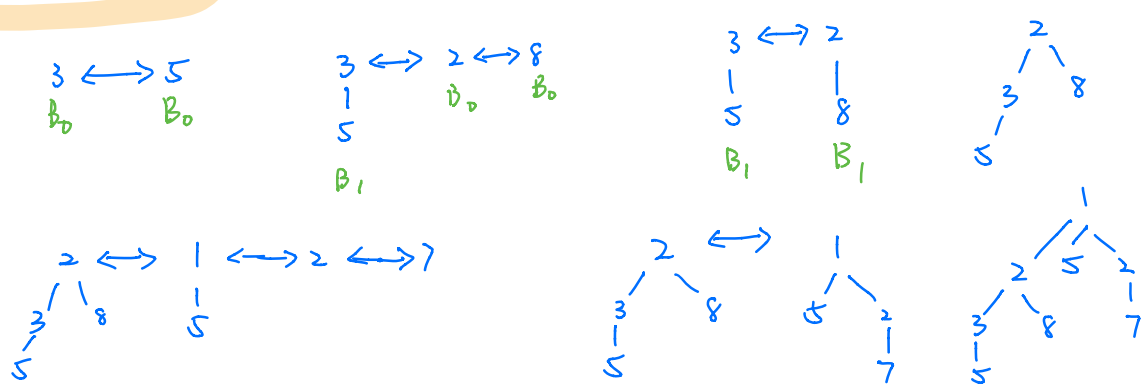
1. build min-heap of k elements $O(k)$

wait for $(k+1)$

root > $(k+1)$st , then do nothing

(otherwise) o.w. delMin, insert $O((n-k)\log k)$

10



Discussion Problem 7

How many binomial trees does a binomial heap with 25 elements contain?

What are the ranks of those trees?

$$25_2 = (16 + 8 + 1) = \underset{B_4\,B_3}{11}\underset{B_0}{001}$$

Insertion is binary addition by 1

Discussion Problem 8

Devise an algorithm for merging two binomial heaps and discuss its complexity. Merge $B_0, B_1, B_2 B_4$ with $B_1 B_4$

Binary heap $O(n)$

Binomial heap $O(\log n)$

$$\begin{array}{r} 10111 \\ +\ 10010 \\ \hline 101001 \end{array}$$

Chapter 4

Greedy Algorithms

Greedy algorithms do not have a formal definition, but all of them possess the following characteristics:

- They make a sequence of choices.
- Each choice is the best available at each step.
- Earlier decisions made during execution are never undone.
- They do not always yield the optimal solution.

Greedy algorithms have several advantages over other algorithmic approaches. The first one is simplicity: Greedy algorithms are often easier to describe and implement. The second is efficiency: The greedy approach can often produce more efficient solutions. At the same time, they have a drawback: Showing that a greedy algorithm is correct often requires a non-trivial proof.

How can we tell if a greedy approach will solve a particular problem? There is no guarantee that such a greedy algorithm exists; however, a problem to be solved must obey the following two common properties:

- Optimal substructure
- Greedy-choice property

An optimal substructure means that an optimal solution to the original problem contains optimal solutions to all of its subproblems. The proof of optimal substructure correctness is usually by induction.

A greedy-choice property means that a globally optimal solution is obtained by making a locally optimal (greedy) choice. This choice is made to solve each subproblem and may depend on choices that have been made to date, but it cannot depend on any future choices. The proof that a greedy choice for each subproblem yields a globally optimal solution is usually by contradiction.

Where does greedy approach efficiency come from? A greedy algorithm can be described as a multistage decision-making process, and therefore we can construct a tree to enumerate all possible decisions. During the algorithm execution, we don't consider all available choices at any given node, but use a greedy heuristic to pick just one, the highest-ranking child. In this model a greedy technique can be viewed as finding a set of paths from the root to a leaf node. Consider a board game where two players alternately take turns. We can use a tree to represent all possible moves until the game ends. Each node corresponds to a position, and each edge corresponds to a move. In this game tree, the number of nodes on each level is exponential in the tree height. Thus, a brute-force algorithm will have an exponential (in height) runtime complexity. A greedy algorithm, in contrast, will make only a single greedy choice at each tree level; therefore, its runtime complexity will be proportional to the height.

We conclude the introduction with a few remarks on implementation. In order to make greedy choices efficiently we have to use a certain data structure. One simple choice is an unsorted array. The better choice may be a priority queue that allows accessing the highest-ranking choice in constant time. Alternatively, we could use a sorted array, though this may be more expensive compared to heaps. In this chapter we start with two greedy algorithms when a single sorting is sufficient and then proceed to other algorithms when use of a priority queue is advantageous.

4.1 The Money Changing Problem

In this problem we are to compute the minimum number of coins needed to make change for a given amount m and given set of n denominations. Assume that we have an unlimited supply of coins. As an example, let us use US currency (pennies, nickels, dimes, and quarters) and the amount to change is $m = \$0.40$. There are several ways to make change, $0.40 = 4*0.10$ (four dimes) or $0.40 = 2*0.10 + 4*0.05$ (two dimes and four nickels). But intuitively we can get the smaller number of coins if we start with the largest coin first: $0.40 = 0.25 + 0.10 + 0.05$. This suggests the following greedy algorithm to make change: Start with the largest coin and use it as many times as possible; then use the second largest coin, and so on. Will we get the least number of coins? We prove the algorithm correct by contradiction. If we do not choose the largest coin, is there a

0.10 = dimes

0.05 = nickels

better solution? Assume that our algorithm does not take the largest coin (which is a quarter). Then we will need a combination of smaller coins (pennies, nickels, or dimes) to add up to a quarter. It follows that the change won't be optimal, since we will end up with more coins.

Let us discuss the algorithm efficiency. We will visualize the algorithm as a quad choice tree (figure 4.1) where each vertex contains an amount to change and each edge is a denomination.

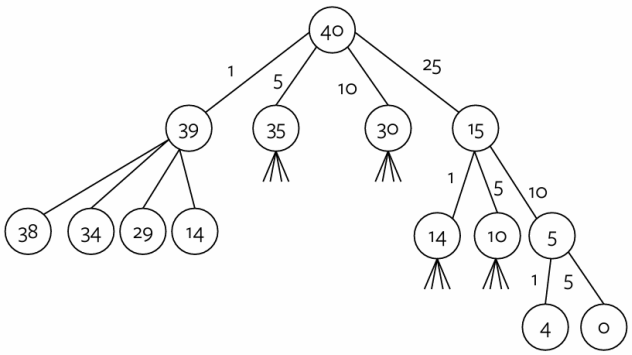


FIGURE 4.1 A choice tree to change 40 cents using pennies, nickels, dimes and quarters.

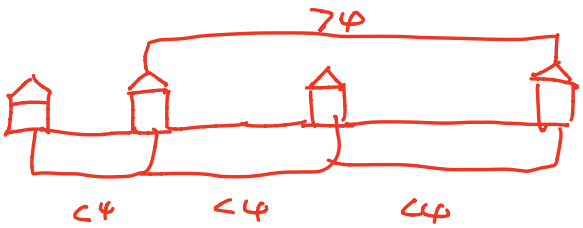
面额

In the brute-force approach we will have to try each available denomination. This will lead to $O(4^h)$ runtime, where h is the tree height. In the greedy approach, we will always choose the largest available denomination. It is clear that the complexity of this approach is $O(h)$. Here we assumed that we could get the largest coin in the constant time. We can always do that by sorting all denominations in descending order and then traversing them in that order. In a general case, when we have a set of n denominations, this choice tree has height $h = n$; therefore, the algorithm runtime complexity is $O(n)$.

Lastly, we demonstrate on the example that a greedy choice does not necessarily yield the optimal solution. Let us imagine a different denomination system, where in addition to pennies, nickels, dimes, and quarters we have a 20-cent coin. Then, running a greedy approach, we still get three coins: $0.40 = 0.25 + 0.10 + 0.05$. However, the optimal solution contains only two coins: $0.40 = 2*0.20$. This example emphasizes an importance of proving the algorithm correctness.

4.2 Scheduling Problem

There is a set of n requests. Each i-th request has a starting time $s(i)$ and finish time $f(i)$. Assume that all requests are equally important and $s(i) \leq f(i)$. Our goal is to develop a


>4
<4
<4
<4

1. sort s(i)

2. sort f(i)-s(i)

3. sort f(i)

greedy algorithm that finds the largest compatible (non-overlapping) subset of requests. This problem is interesting because among many available greedy strategies it is not obvious which one to choose. One approach is to sort requests with respect to $s(i)$ in ascending order. This one is not going to work (see figure 4.2). In that example the solution will consist of one request.

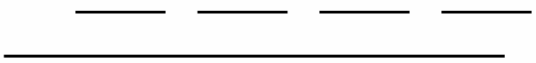


FIGURE 4.2 The earliest starting time strategy.

You may think that starting with the shortest $f(i) - s(i)$ request first will be the right strategy. See figure 4.3 for the counterexample.

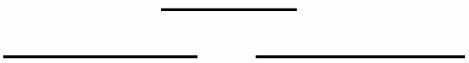


FIGURE 4.3 The shortest interval strategy.

Another possible strategy is to take into consideration the number of overlapping intervals. In this strategy we start with an interval that has the smallest number of overlaps with other intervals. See figure 4.4 for the counterexample. It demonstrates that using this strategy we get only three intervals; however, the optimal solution has four intervals.

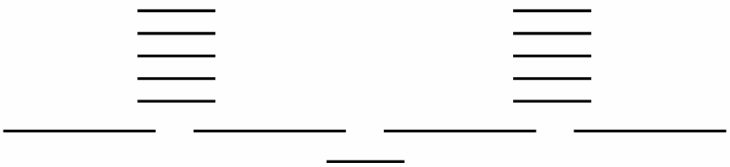


FIGURE 4.4 The smallest number of overlaps.

f(1) f(2) f(3) f(i)

k

n

nlogn

Finally, we consider a strategy of taking intervals with respect to finish time $f(i)$, a request with the earliest finish time first. In this approach we sort requests with respect to $f(i)$ in ascending order. Pick a request that has the earliest finish time. Delete all requests that overlap with it. Repeat.

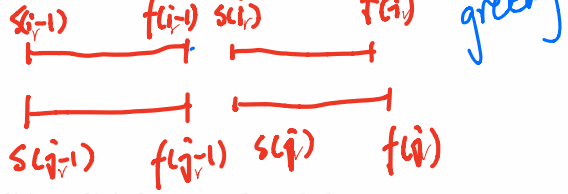
The running time is $O(n \log n)$ for sorting plus $O(n)$ for the greedy collection of activities. Does it always find an optimum?

sort

s[i]

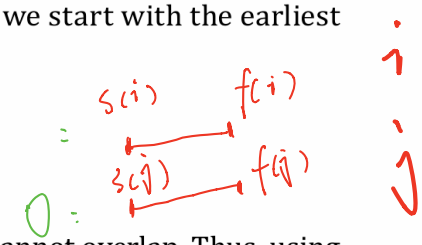
4.2.1 Proof of Optimality

We assume that all intervals are sorted with respect to the finish time. Let $\{i_1, i_2, \ldots, i_k\}$ be a subset of intervals chosen by our greedy algorithm and $\{j_1, j_2, \ldots, j_m\}$ be the optimal



subset of intervals. We will prove by induction that $f(i_r) \leq f(j_r)$ for $\forall r \leq k$ and thus our solution cannot be worse than the optimal one.

> *Base case*: $r = 1$. This is true, $f(i_1) = f(j_1)$, because we start with the earliest finish time.
>
> *Inductive hypothesis*: Let us assume $f(i_{r-1}) \leq f(j_{r-1})$.
>
> *Inductive step*: We need to prove $f(i_r) \leq f(j_r)$.



Note, $f(j_{r-1}) \leq s(j_r)$ since in the solution the intervals cannot overlap. Thus, using the inductive hypothesis $f(i_{r-1}) \leq f(j_{r-1}) \leq s(j_r)$ we arrive at $f(i_{r-1}) \leq s(j_r)$. Since j_r is in the optimal set, then in the next step our greedy algorithm must pick the j_r interval. It follows $f(i_r) \leq f(j_r)$.

Next, we need to prove that our solution $\{i_1, i_2, \ldots, i_k\}$ has the same size as the optimal solution (i.e., $k = m$). We prove this by contradiction. Let us assume that $k < m$. There must be a request j_{k+1} such that $f(j_k) \leq s(j_{k+1})$ and $f(i_k) \leq f(j_k)$. Combining these two inequalities, it follows $f(i_k) \leq s(j_{k+1})$. This means that a request j_{k+1} does not overlap with any $i_1, i_2, \ldots, i_k$ requests. So, our greedy algorithm would not stop at i_k and choose j_{k+1} as the next request. Contradiction, the size of our solution, is bigger than k.

4.3 Huffman Code

In 1948 Claude Shannon established that there is a fundamental limit to lossless data compression. This limit is called the entropy rate H. Entropy is a measure of the amount of information contained in the source. It is possible to compress the source, in a lossless manner, with a compression rate close to the entropy H. But it is mathematically impossible to do better than H. The ASCII table is the simplest example of data compression. In that model we assign a fixed number of bits (called a codeword) to a character, namely 8 bits. But we can achieve a better compression ratio if we assign a variable number of bits to each character. It is known (statistically) that the character "e" is much more likely to appear than "u." In this model each letter in the alphabet of size n has a certain probability p_k. We can define probabilities by counting frequencies of each character in the input text. The entropy H is given by

$$H = \sum_{k=0}^{n} p_k \log \frac{1}{p_k}.$$

a codeword for C is 100
a codeword for H is 11

H is the lower bound on the average number of bits to code a character. Using standard distribution of characters in the English language, we get $H = 4.07$ bits/char.

This theoretical result cannot be directly used to compute the number of bits per letter in a data compression algorithm, due to the fact that the log values are not integers. If we round up the logs, then the solution is not an optimal. In 1952 David Huffman developed a greedy algorithm to assign a prefix-free codeword to each character in the text according to their frequencies. A prefix-free code is one where no codeword is a prefix of another codeword.

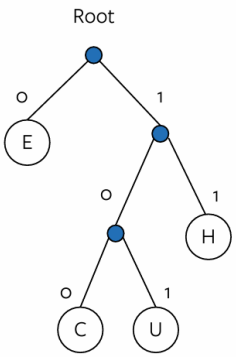


FIGURE 4.5 Prefix-free binary codes.

We will be using a full tree to map each character to a binary string. A codeword is a path from the root to the character. In figure 4.5 a codeword for C is 100 and a codeword for H is 11. Using a prefix-free code it is easy to encode and decode data. To encode, we need only to concatenate the codewords for each character. To decode, we scan the text from left to right, and as soon as we recognize a codeword, we print the corresponding character.

In general, we want to minimize the overall length of encoding, namely

$$\text{cost}(T) = \min \sum_{k=1}^{n} f(x_k)\, d(x_k)$$

where $f(x)$ is a frequency of x_k character and $d(x_k)$ is a depth of x_k in the tree T. This suggests a greedy approach to constructing a tree. We need to put characters with the lowest frequencies to the bottom of a tree. This will guarantee longer binary strings assigned to them. Characters with the high frequency should be at the top, so they will have shorter codewords. Such a tree is called a Huffman tree.

4.3.1 Example: Building a Huffman tree

Let us draw a Huffman tree for the following table of frequencies:

TABLE 4.1 A table of frequencies

char	A	M	L	E	K	B	U	X
freq	34	21	14	13	11	9	8	7

Example : encode

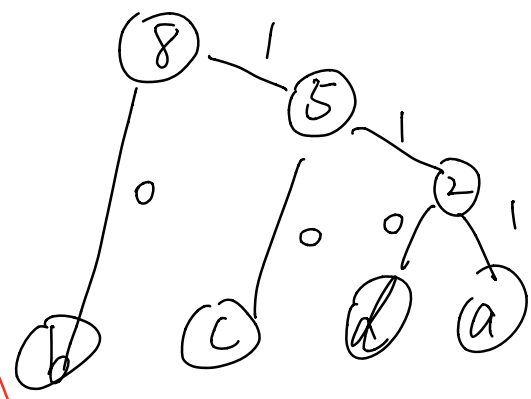
a b c b c d b c $8 \times 8 = 64$ bits

a - 1

b - 3

c - 3

d - 1



a = 111

b = 0

c = 10

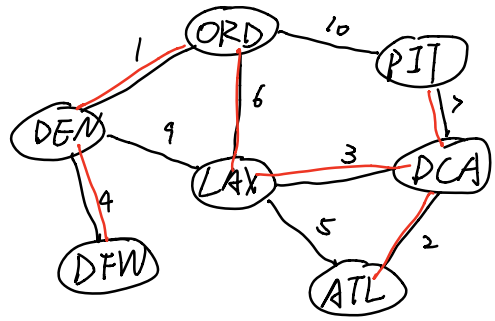
d = 110

$3 \times 1 = 3+3+6+3 = 15$ bits

111 0 10 0 10 110 0 10

The Minimum Spanning Tree

Find a spanning tree of the minimum total weight.



Initially, there are only single-node trees: one for each character.

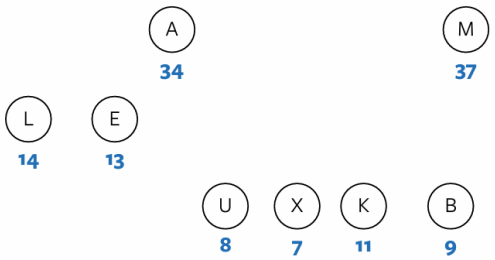


FIGURE 4.6 Single-node trees.

Next we select two characters of the smallest frequencies (they are U and X) and form a new parent node with the frequency $8 + 7 = 15$, and connect it to U and X.

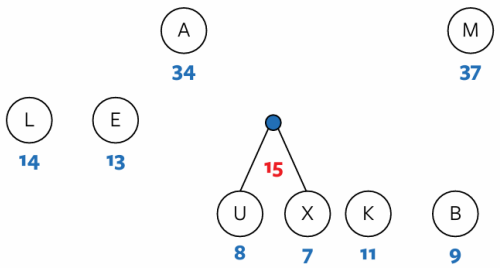


FIGURE 4.7 A parent node for U and X.

Once two nodes in a tree are connected, they are removed from consideration. However, their parent node is still in the game. Again, select two characters of the smallest frequencies (they are K and B), form a new node with the frequency $11 + 9 = 20$, and connect it to K and B. In the next step we connect L and E. The result is depicted in figure 4.8.

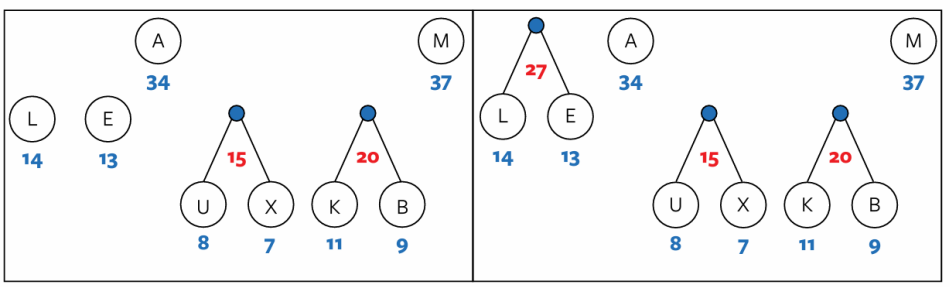


FIGURE 4.8 The result of joining K and B, and then L and E.

Next, we join 15 and 20, and then 27 and 34 as in figure 4.9

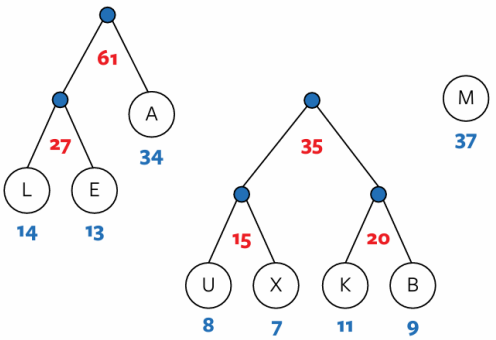


FIGURE 4.9 The result of joining 15 and 20, and then 27 and 34.

Continue connecting nodes until there is only one tree left. That tree is the optimal Huffman coding tree. Lastly, we assign 0's and 1's to the edges.

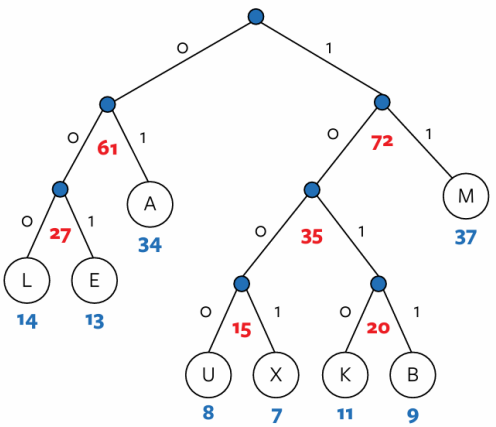


FIGURE 4.10 A Huffman tree.

Table 4.2 is a table of codewords.

TABLE 4.2 A table of codewords

char	A	M	L	E	K	B	U	X
freq	34	37	14	13	11	9	8	7
codeword	01	11	000	001	1010	1011	1000	1001

To get the total number of bits needed to compress a text (given the frequency table) we multiply the frequency of each character by the codeword length in bits:

$$34*2 + 37*2 + 14*3 + 13*3 + 11*4 + 9*4 + 8*4 + 7*4 = 363.$$

4.3.2 Proof of Optimality

We will prove it by induction on the number of characters.

Base case: Two characters. The tree is unique; therefore, it is optimal.

Inductive hypothesis: Assume that a Huffman tree of any $\boxed{n-1}$ characters is optimal.

Inductive step: We need to prove that a Huffman tree of n characters is optimal.

Given a set A of n characters x_k with some frequencies $f(x_k)$, where $k = 1,2, \ldots, n$. If we run our greedy algorithm we will get a tree T over A. We will prove that T is optimal in a sense that T minimizes the overall length of encoding:

$$\text{cost}(T) = \min \sum_{k=1}^{n} f(x_k)\ d(x_k),$$

where $d(x_k)$ is a depth of x_k in the tree T. Note that $d(x_k)$ is the number of bits of a codeword associated with x_k. Let us run a single step of our greedy algorithm. We choose two characters, x_1 and x_2, with the lowest frequencies $f(x_1)$ and $f(x_2)$. Then we join them to create a parent node x^* with a frequency $f(x^*) = f(x_1) + f(x_2)$. After this step the number of characters in consideration is decreased by one (we removed x_1 and x_2 and added x^*). Let us call this new set of characters by A^*:

$$A^* = A \setminus \{x_1, x_2\} \cap \{x^*\}.$$

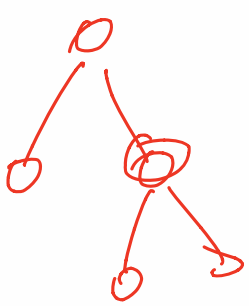
$|A| = n \quad |A^*| = n-2+1 = n-1$

Since the size of A^* is $n - 1$, we can apply the inductive hypothesis and thus build an optimal Huffman tree over A^*. We will call this tree by T^*. Let us summarize the construction; we say that we build a tree T (over a set A) by running one step of the algorithm for two characters and then using inductive hypothesis for the rest of characters (a set A^*):

$$\text{cost}(T) = \sum_{k=1}^{n} f(x_k)\ d(x_k) = f(x_1)\ d(x_1) + f(x_2)\ d(x_2) + \sum_{k=1}^{n-2} f(x_k) d(x_k).$$

By construction the cost of T is related to cost (T^*) as follows:

$$\text{cost}(T) = f(x_1)d(x_1) + f(x_2)d(x_2) + \text{cost}(T^*) - f(x^*)d(x^*).$$



Here, $f(x^*) = f(x_1) + f(x_2)$ and $d(x_1) = d(x^*) + 1$ and $d(x_2) = d(x^*) + 1$. Substituting these into the previous equation, we get

$$\text{cost}(T) = \text{cost}(T^*) + f(x_1) + f(x_2).$$

It is important to observe that we can reverse the construction process; namely, we can get an optimal tree T^* from T by removing two lowest frequency nodes if they are siblings (see 4.3.2.1 for details)

$$\text{cost}(T^*) = \text{cost}(T) - f(x_1) - f(x_2)$$

Having this in mind we proceed to the next step. We prove optimality of T by contradiction. Assume that there is another tree T_1 over the set A such that cost $(T_1) <$ cost (T). For the tree T_1 we can perform the same reverse process to get another optimal tree T_1^*:

$$\text{cost}(T_1^*) = \text{cost}(T_1) - f(x_1) - f(x_2).$$

Since $\text{cost}(T_1) < \text{cost}(T)$, it follows $\text{cost}(T_1^*) < \text{cost}(T^*)$. But tree T^* is optimal. This is a contradiction. ■

4.3.2.1 What if x_1 and x_2 are not siblings?

Lemma. *Let x and y be two characters such that f(x) and f(y) are minimal. Then there is an optimal prefix code such that x and y are siblings.*

Proof. Let T be the optimal tree and z be a sibling of x such that $d(x) = d(z) \geq d(y)$. Consider two cases.

Case 1. $d(z) = d(y)$. If z and y are at the same depth, we can swap them. The cost of the optimal tree T won't change.

Case 2. $d(z) > d(y)$. Since z is located deeper within the tree T than $f(z) < f(y)$, we swap z and y and call that tree T_1 as in figure 4.11.

Next, we compute the cost of T:

$$\text{cost}(T) = f(x)d(x) + f(z)d(x) + f(y)d(y) + \ldots$$

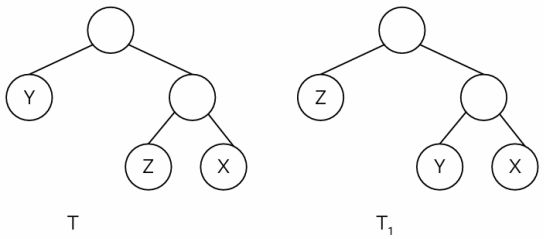


FIGURE 4.11 The result of swapping z and y.

the cost of T_1

$$\text{cost}(T_1) = f(x)d(x) + f(z)d(y) + f(y)d(x) + \ldots$$

and subtract them to get

$$\text{cost}(T_1) - \text{cost}(T) = f(z)d(y) + f(y)d(x) - f(z)d(x) - f(y)d(y) = (f(y) - f(z))\,(d(x) - d(y)).$$

Since $f(y) > f(z)$ and $d(x) < d(y)$, it follows that $\text{cost}(T_1) \cancel{\leq} > \text{cost}(T)$. But T is the optimal tree, thus $\text{cost}(T_1) = \text{cost}(T)$. ■

4.3.3 Runtime Complexity of Building a Huffman Tree

Let us assume that a frequency table is given to us and its size is n. The algorithm works by repeatedly connecting a pair of nodes that have the smallest frequencies. The frequency of the new node is the sum of the frequencies of the connected nodes. We keep (node, frequency) in a min-heap. In each step of the algorithm we extract two nodes with the smallest frequencies, create a new parent node, and insert it back into the heap. The whole process takes $O(n \log n)$ time. Observe that using an unsorted array instead of a min-heap is less efficient. It will take us $O(n)$ to find the minimum, and $O(1)$ to insert a parent node. This will lead to $O(n^2)$ time.

if in min-heap
WRC = $O(n \log n)$
if in unsorted array
WRC = $O(n^2$

4.3.4 Storing a Huffman Tree

In this section we will discuss decompression—a process of translating the stream of prefix codes back to the characters. It should be clear that in order to decompress an encoded text file we have to have the same Huffman tree that was used to compress. Therefore, every compressed file must have the whole Huffman tree stored in a binary form. How to store a full tree? It turns out a single bit per node is sufficient: a 0 bit for an internal node and a 1 bit for a leaf. We output nodes in preorder traversal. Once we hit a

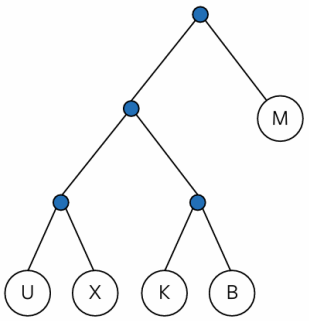


FIGURE 4.12 The Huffman tree to store.

leaf we output a binary ASCII code (8 bits) for that character. As an example, consider the tree in figure 4.12

The preorder traversal yields 0001U1X01K1B1M. The ASCII code for character U is 85, or 01010101 in binary. Taking into account ASCII codes for all other characters, we get the following encoded string: 000101010101101011000010100101111010000101001101. The total number of bits required to store that tree is $9 + 8*5 = 49$.

The Huffman tree is always stored at the beginning of a compressed file and is called a header. During decompression we read the header and restore the Huffman tree recursively from a preorder traversal.

4.4 Minimum Spanning Trees

Given a weighted undirected connected graph $G = (V, E)$, a spanning tree is a subgraph of G that contains all vertices and it is a tree. The cost of a tree is the sum of the weights of its edges. A minimum spanning tree (MST) is a spanning tree with the minimum cost.

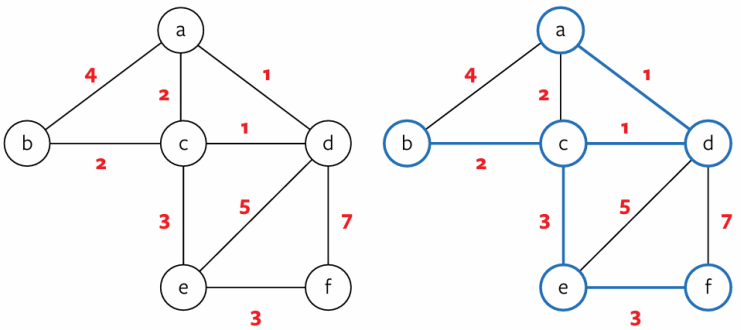


FIGURE 4.13 A graph on the left and its MST (blue edges) on the right.

It is important to note that a given graph is undirected! For directed graphs an MST problem is defined in a different way and called an arborescence problem (which we won't cover in this text). Before we proceed with a greedy approach, let us discuss a brute-force approach. We find all spanning trees (using a BFS, for example) for a given graph and then choose the one with the smallest cost. It turns out this approach is quite expensive. In 1889 Arthur Cayley proved that the number of spanning trees in K_n (a complete graph on n vertices) is n^{n-2}, and thus the brute force approach has an exponential runtime complexity. We will omit a proof of Cayley's theorem, and instead we will discuss a polynomial time algorithm for finding an MST.

Prim算法从任意一个顶点开始，每次选择一个与当前顶点集最近的一个顶点，并将两顶点之间的边加入

算法描述：

1. 在一个加权连通图中，顶点集合V，边集合为E

2. 任意选出一个点作为初始顶点,标记为visit,计算所有与之相连接的点的距离，选择距离最短的，标记visit.

3. 重复以下操作，直到所有点都被标记为visit：

在剩下的点钟，计算与已标记visit点距离最小的点，标记visit,证明加入了最小生成树。

4.4.1 Prim's Algorithm (1957)

For any weighted undirected graph $G = (V, E)$, the algorithm builds a minimum spanning tree T one vertex at a time. Here are the algorithm steps:

增加最小边

1. Start with an arbitrary vertex and add it to an empty tree T. This vertex will be the root of T.
2. Expand T by adding a vertex from $V \backslash T$, having the minimum weight edge and having exactly one end point in T.
3. Update distances from all vertices in T to adjacent vertices in $V \backslash T$.
4. Continue to grow the tree until T gets all vertices, $T = V$.

Step 2 is a greedy choice: Among all adjacent vertices to T we pick the one that has the minimum weight edge. Step 3 is the most important step; we update only the shorter edges from T to $V \backslash T$. The greedy choice implies that we have to use an intermediate data structure, which will allow us to find a vertex with the shortest edge in the most efficient way. This suggests use of a priority queue.

4.4.1.1 Example: Building an MST

Let us run Prim's algorithm on the graph in figure 4.14. We will keep a binary min-heap H as an intermediate data structure. Every element of the min-heap contains a vertex number and a key value of the vertex, which is an edge weight from a tree T to the vertex. In min-heap H we won't show vertices that are not yet connected to T by an edge.

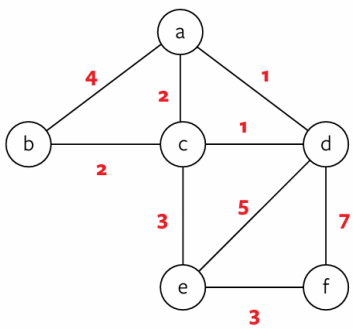


FIGURE 4.14 The Prim algorithm illustrated on this graph.

We start at vertex a, so $T = \{a\}$. Edge weights from T to all adjacent vertices are $|ab| = 4$, $|ac| = 2$, $|ad| = 1$. We update adjacent vertices in a heap $H = \{d_1, c_2, b_4\}$, where the subscripts denote an edge weight from a tree T to the vertex. We pick the shortest vertex in H (which is d) and add it to T. After this step $T = \{a, d\}$ and $H = \{c_2, b_4\}$.

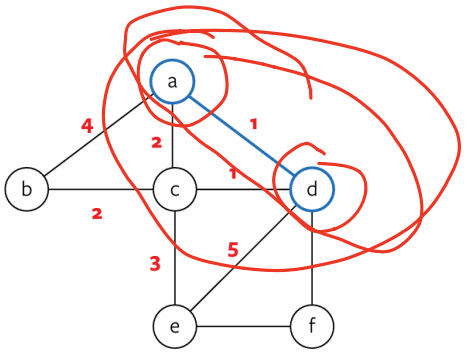


FIGURE 4.15 The first iteration of Prim's algorithm.

In the next iteration we update edges from T to all adjacent vertices b, c, e, and f. The binary heap becomes $H = \{c_1, b_4, e_5, f_7\}$. Note that the edge (a, c) to vertex c gets replaced by a shorter one (d, c). Now vertex c in H is the closest one to T. We remove it from the heap H and add it to tree T. After this step $T = \{a, d, c\}$ and $H = \{b_4, e_5, f_7\}$.

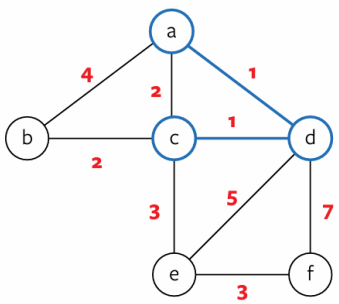


FIGURE 4.16 The second iteration of Prim's algorithm.

Next, we update edges from T to all adjacent vertices b, e, and f. The binary heap becomes $H = \{b_2, e_3, f_7\}$. Since vertex b is the shortest one, we remove it from H and add it to T. After this step $T = \{a, d, c, b\}$ and $H = \{e_5, f_7\}$.

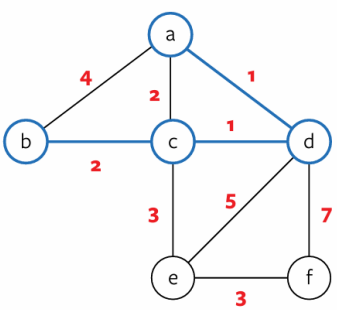


FIGURE 4.17 The third iteration of Prim's algorithm.

Again we update edges in H, so $H = \{e_3, f_7\}$. Vertex e is the shortest one; we remove it from H and add it to T. After this step $T = \{a, d, c, b, e\}$ and $H = \{f_7\}$.

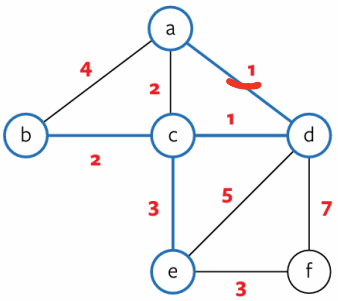


FIGURE 4.18 The fourth iteration of Prim's algorithm.

Update heap $H = \{f_3\}$. Vertex f is the shortest one; we remove it from H and add it to T. After this step $T = \{a, d, c, b, e, f\}$ and $H = \{\}$.

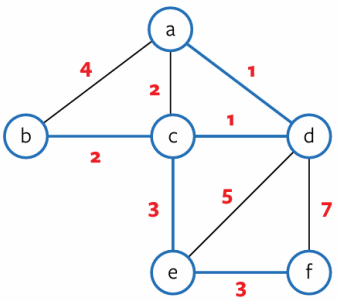


FIGURE 4.19 The minimum spanning tree.

We have constructed the minimum spanning tree of the total weight $1 + 1 + 2 + 3 + 3 = 10$.

4.4.1.2 *Complexity of Prim's Algorithm*

The Prim's algorithm complexity depends heavily on the chosen graph representation. The following analysis assumes using an adjacency list structure for graph representation. Here is a pseudocode for the algorithm:

Build binary heap $O(V)$

Loop:

(a) delete min $O(V \log V)$

(b) decreasekey $O(E \log V)$

1. delMin $O(V \log V)$

```
1. H = minHeap(V);
2. insert(s, H);                    // start at vertex s
3. while (H is not empty)
4. {
5.  u = deleteMin(H);               // O(log V)
6.  for each w in adj(u)
7.  {
8.      if(weight(w,u) < key(u))
9.          key(u) = weight(w,u)    // update edge weight
10.         decreaseKey(u, H);      // O(log V)
11.  }
12. }
```

2. decreaseKey $O(E \log V)$

3. total 1+2

usually people write $O(E \log V)$

We maintain a min-heap of V vertices (line 1). In each step of the algorithm we delete a vertex (line 5) with the smallest weight (this takes $O(\log V)$ by `deleteMin` operation). We also update edges (lines 8–10) to all adjacent vertices (this takes $O(\log V)$ by `decreaseKey` operation). We run `deleteMin` operation once on each vertex, so the time required is $O(V \log V)$. We run `decreaseKey` operation once on each edge, so the time required is $O(E \log V)$. The latter requires an explanation. Consider the inner loop, lines 6–11. The number of steps in that loop depends on the degree of vertex u, which we will denote by $\deg(u)$. Thus, the complexity of the inner loop is $O(\deg(u) \log V)$. If we add the outer loop, the total runtime complexity is given by

$$\sum_{u \in V}(O(\log V)+O(\deg(u) \log V)) = V\ O(\log V)+O(\log V)\sum_{u \in V}O(\deg(u)) =$$

$$= O(V \log V)+O(\log V)\ E = O(V \log V+E \log V).$$

Prim's algorithm can be further improved by using Fibonacci heaps that provide the best runtime in theory. In this case, the algorithm complexity is $O(V \log V + E)$ amortized.

after use Fibonacci heap improve it

4.4.1.3 *Prim's Algorithm Using an Array*

This is the simplest implementation of Prim's algorithm. We use an unsorted array instead of a priority queue. Assuming the pseudocode from the previous section, let us analyze the runtime complexity. Each `deleteMin` (line 5) will take $O(V)$ times to find the minimum in an unsorted array. Each `decreaseKey` (line 10) will take $O(1)$ to update the edge weight. The total runtime complexity is given by

$$\sum_{u \in V}(O(V)+O(\deg(u))) = V\ O(V)+\sum_{u \in V}O(\deg(u)) = O(V^2)+E = O(V^2+E).$$