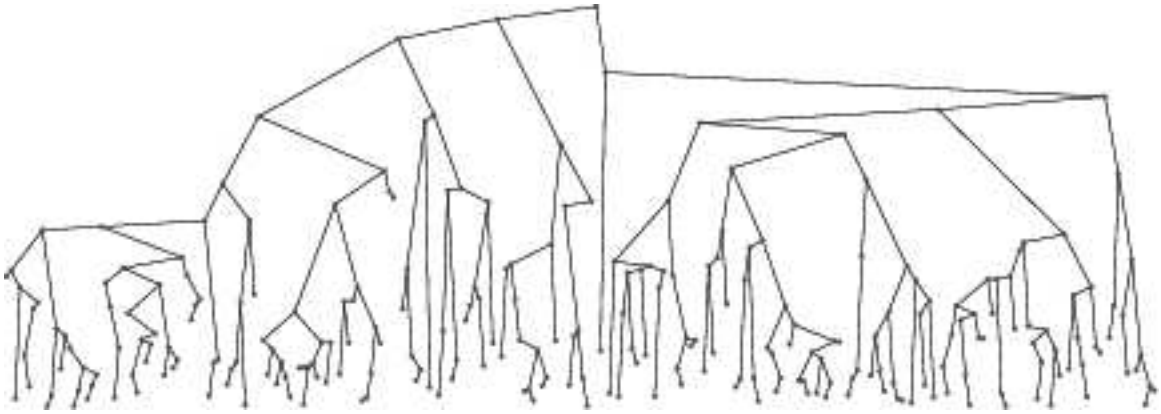


Open Data Structures (in C++)

Pat Morin

Edition 0.1β

Acknowledgments

I am grateful to Nima Hoda, who spent a summer tirelessly proofreading many of the chapters in this book, and to the students in the Fall 2011 offering of COMP2402/2002, who put up with the first draft of this book and spotted many typographic, grammatical, and factual errors in the first draft.

Preface to the C++ Edition

This book is intended to teach the design and analysis of basic data structures and their implementation in an object-oriented language. In this edition, the language happens to be C++.

This book is not intended to act as an introduction to the C++ programming language. Readers of this book need only be familiar with the basic syntax of C++ and similar languages. Those wishing to work with the accompanying source code should have some experience programming in C++.

This book is also not intended as an introduction to the C++ Standard Template Library or the generic programming paradigm that the STL embodies. This book describes implementations of several different data structures, many of which are used in implementations of the STL. The contents of this book may help an STL programmer understand how some of the STL data structures are implemented and why these implementations are efficient.

Why This Book?

There are plenty of books that teach introductory data structures. Some of them are very good. Most of them cost money, and the vast majority of computer science undergraduate students will shell-out at least some cash on a data structures book.

There are a few free data structures books available online. Some are very good, but most of them are getting old. The majority of these books became free when the author and/or publisher decided to stop updating them. Updating these books is usually not possible, for two reasons: (1) The copyright belongs to the author or publisher, who may not allow it. (2) The *source code* for these books is often not available. That is, the Word, WordPerfect, FrameMaker, or LaTeX source for the book is not available, and the version of the software that handles this source may not even be available.

The goal of this project is to forever free undergraduate computer science students from having to pay for an introductory data structures book. I have decided to implement this goal by treating this book like an Open Source software project. The LaTeX source, C++ source, and build scripts for the book are available for download on the book's website (`opendatastructures.org`) and also — more importantly — on a reliable source code management site (`https://github.com/patmorin/ods`).

This source code is released under a Creative Commons Attribution license, meaning that anyone is free

- to Share — to copy, distribute and transmit the work; and
- to Remix — to adapt the work.

This includes the right to make commercial use of the work. The only condition on these rights is

- Attribution — You must attribute the work by displaying a notice stating the derived work contains code and/or text from the Open Data Structures Project and/or linking to `opendatastructures.org`.

Anyone can contribute corrections/fixes using the `git` source-code management system. Anyone can fork from the current version of the book and develop their own version (for example, in another programming language). They can then ask that their changes be merged back into my version. My hope is that, by doing things this way, this book will continue to be a useful textbook long after my interest in the project (or my pulse, whichever comes first) has waned.

Contents

1 Introduction **1**

1.1 Interfaces 1

1.1.1 The `Queue`, `Stack`, and `Deque` Interfaces 2

1.1.2 The `List` Interface: Linear Sequences 2

1.1.3 The `USet` Interface: Unordered Sets 3

1.1.4 The `SSet` Interface: Sorted Sets 4

1.2 Mathematical Background 5

1.2.1 Logarithms 5

1.2.2 Factorials 6

1.2.3 Asymptotic Notation 6

1.2.4 Randomization and Probability 8

1.3 The Model of Computation 10

1.4 Code Samples 11

1.5 List of Data Structures 11

1.6 References 12

2 Array-Based Lists **15**

2.1 `ArrayStack`: Fast Stack Operations Using an Array 17

2.1.1 The Basics 17

2.1.2 Growing and Shrinking 19

2.1.3 Summary 21

2.2 `FastArrayStack`: An Optimized ArrayStack 21

2.3 `ArrayQueue`: An Array-Based Queue 22

2.3.1 Summary 25

2.4 `ArrayDeque`: Fast Deque Operations Using an Array 25
2.4.1 Summary 27
2.5 `DualArrayDeque`: Building a Deque from Two Stacks 28
2.5.1 Balancing 31
2.5.2 Summary 33
2.6 `RootishArrayStack`: A Space-Efficient Array Stack 33
2.6.1 Analysis of Growing and Shrinking 37
2.6.2 Space Usage 37
2.6.3 Summary 38
2.6.4 Computing Square Roots 39
2.7 Discussion and Exercises 41

3 Linked Lists **45**
3.1 `SLList`: A Singly-Linked List 45
3.1.1 Queue Operations 47
3.1.2 Summary 48
3.2 `DLList`: A Doubly-Linked List 48
3.2.1 Adding and Removing 50
3.2.2 Summary 51
3.3 `SEList`: A Space-Efficient Linked List 52
3.3.1 Space Requirements 53
3.3.2 Finding Elements 53
3.3.3 Adding an Element 55
3.3.4 Removing an Element 57
3.3.5 Amortized Analysis of Spreading and Gathering 59
3.3.6 Summary 60
3.4 Discussion and Exercises 61

4 Skiplists **63**
4.1 The Basic Structure 63
4.2 `SkiplistSSet`: An Efficient `SSet` Implementation 65
4.2.1 Summary 68

4.3 `SkiplistList`: An Efficient Random-Access `List` Implementation 68
4.3.1 Summary 73
4.4 Analysis of Skiplists 73
4.5 Discussion and Exercises 76

5 Hash Tables **79**
5.1 `ChainedHashTable`: Hashing with Chaining 79
5.1.1 Multiplicative Hashing 81
5.1.2 Summary 85
5.2 `LinearHashTable`: Linear Probing 85
5.2.1 Analysis of Linear Probing 88
5.2.2 Summary 90
5.2.3 Tabulation Hashing 91
5.3 Hash Codes 92
5.3.1 Hash Codes for Primitive Data Types 92
5.3.2 Hash Codes for Compound Objects 93
5.3.3 Hash Codes for Arrays and Strings 94
5.4 Discussion and Exercises 97

6 Binary Trees **101**
6.1 `BinaryTree`: A Basic Binary Tree 102
6.1.1 Recursive Algorithms 103
6.1.2 Traversing Binary Trees 103
6.2 `BinarySearchTree`: An Unbalanced Binary Search Tree 106
6.2.1 Searching 106
6.2.2 Inserting 108
6.2.3 Deleting 110
6.2.4 Summary 111
6.3 Discussion and Exercises 112

7 Random Binary Search Trees **115**
7.1 Random Binary Search Trees 115

7.1.1 Proof of Lemma 7.1 117
7.1.2 Summary 119
7.2 `Treap`: A Randomized Binary Search Tree 119
7.2.1 Summary 127
7.3 Summary and Exercises 128

8 Scapegoat Trees **131**
8.1 `ScapegoatTree`: A Binary Search Tree with Partial Rebuilding 132
8.1.1 Analysis of Correctness and Running-Time 135
8.1.2 Summary 137
8.2 Discussion and Exercises 137

9 Red-Black Trees **141**
9.1 2-4 Trees 142
9.1.1 Adding a Leaf 142
9.1.2 Removing a Leaf 144
9.2 `RedBlackTree`: A Simulated 2-4 Tree 144
9.2.1 Red-Black Trees and 2-4 Trees 146
9.2.2 Left-Leaning Red-Black Trees 149
9.2.3 Addition 150
9.2.4 Removal 153
9.3 Summary 158
9.4 Discussion and Exercises 159

10 Heaps **163**
10.1 `BinaryHeap`: An Implicit Binary Tree 163
10.1.1 Summary 168
10.2 `MeldableHeap`: A Randomized Meldable Heap 168
10.2.1 Analysis of `merge(h1,h2)` 170
10.2.2 Summary 172
10.3 Discussion and Exercises 172

11 Sorting Algorithms **173**

11.1 Comparison-Based Sorting 173

11.1.1 Merge-Sort 173

11.1.2 Quicksort 176

11.1.3 Heap-sort 179

11.1.4 A Lower-Bound for Comparison-Based Sorting 181

11.2 Counting Sort and Radix Sort 184

11.2.1 Counting Sort 184

11.2.2 Radix-Sort 186

11.3 Discussion and Exercises 187

Chapter 1

Introduction

This chapter briefly reviews some of the main concepts used throughout the rest of the book. Section 1.1 describes the interfaces implemented by all the data structures described in this book. It should be considered required reading. The remaining sections discuss asymptotic (big-Oh) notation, probability and randomization, the model of computation, and the sample code and typesetting conventions. A reader with or without a background in these areas can easily skip them now and come back to them later if necessary.

1.1 Interfaces

In discussing data structures, it is important to understand the difference between a data structure's interface and its implementation. An interface describes what a data structure does, while an implementation describes how the data structure does it.

An *interface*, sometimes also called an *abstract data type*, defines the set of operations supported by a data structure and the semantics, or meaning, of those operations. An interface tells us nothing about how the data structure implements these operations, it only provides the list of supported operations along with specifications about what types of arguments each operation accepts and the value returned by each operation.

A data structure *implementation* on the other hand, includes the internal representation of the data structure as well as the definitions of the algorithms that implement the operations supported by the data structure. Thus, there can be many implementations of a single interface. For example, in Chapter 2, we will see implementations of the `List` interface using arrays and in Chapter 3 we will see implementations of the `List` interface using pointer-based data structures. Each implements the same interface, `List`, but in different ways.

1.1.1 The `Queue`, `Stack`, and `Deque` Interfaces

The `Queue` interface represents a collection of elements to which we can add elements and remove the next element. More precisely, the operations supported by the `Queue` interface are

- `add(x)`: add the value `x` to the `Queue`
- `remove()`: remove the next (previously added) value, `y`, from the `Queue` and return `y`

Notice that the `remove()` operation takes no argument. The `Queue`'s *queueing discipline* decides which element should be removed. There are many possible queueing disciplines, the most common of which include FIFO, priority, and LIFO.

A *FIFO (first-in-first-out)* `Queue` removes items in the same order they were added, much in the same way a queue (or line-up) works when checking out at a cash register in a grocery store.

A *priority* `Queue` always removes the smallest element from the `Queue`, breaking ties arbitrarily. This is similar to the way many airlines manage upgrades to the business class on their flights. When a business-class seat becomes available it is given to the most important customer waiting on an upgrade.

A very common queueing discipline is the LIFO (last-in-first-out) discipline. In a *LIFO Queue*, the most recently added element is the next one removed. This is best visualized in terms of a stack of plates; plates are placed on the top of the stack and also removed from the top of the stack. This structure is so common that it gets its own name: `Stack`. Often, when discussing a `Stack`, the names of `add(x)` and `remove()` are changed to `push(x)` and `pop()`; this is to avoid confusing the LIFO and FIFO queueing disciplines.

A `Deque` is a generalization of both the FIFO `Queue` and LIFO `Queue` (`Stack`). A `Deque` represents a sequence of elements, with a front and a back. Elements can be added at the front of the sequence or the back of the sequence. The names of the operations on a `Deque` are self-explanatory: `addFirst(x)`, `removeFirst()`, `addLast(x)`, and `removeLast()`. Notice that a `Stack` can be implemented using only `addFirst(x)` and `removeFirst()` while a FIFO `Queue` can be implemented using only `addLast(x)` and `removeFirst()`.

1.1.2 The `List` Interface: Linear Sequences

This book will talk very little about the FIFO `Queue`, `Stack`, or `Deque` interfaces. This is because these interfaces are subsumed by the `List` interface. A `List` represents a sequence, $x_0, \ldots, x_{n-1}$, of values. The `List` interface includes the following operations:

1. `size`(): return `n`, the length of the list
2. `get`(`i`): return the value $\mathtt{x}_\mathtt{i}$
3. `set`(`i`, `x`): set the value of $\mathtt{x}_\mathtt{i}$ equal to `x`
4. `add`(`i`, `x`): add `x` at position `i`, displacing $\mathtt{x}_\mathtt{i}, \ldots, \mathtt{x}_{\mathtt{n}-1}$;
   Set $\mathtt{x}_{j+1} = \mathtt{x}_j$, for all $j \in \{\mathtt{n}-1, \ldots, \mathtt{i}\}$, increment `n`, and set $\mathtt{x}_i = \mathtt{x}$
5. `remove`(`i`) remove the value $\mathtt{x}_\mathtt{i}$, displacing $\mathtt{x}_{\mathtt{i}+1}, \ldots, \mathtt{x}_{\mathtt{n}-1}$;
   Set $\mathtt{x}_j = \mathtt{x}_{j+1}$, for all $j \in \{\mathtt{i}, \ldots, \mathtt{n}-2\}$ and decrement `n`

Notice that these operations are easily sufficient to implement the `Deque` interface:

$$\begin{aligned} \texttt{addFirst}(\texttt{x}) &\Rightarrow \texttt{add}(0, \texttt{x}) \\ \texttt{removeFirst}(\texttt{x}) &\Rightarrow \texttt{remove}(0) \\ \texttt{addLast}(\texttt{x}) &\Rightarrow \texttt{add}(\texttt{size}(), \texttt{x}) \\ \texttt{removeLast}(\texttt{x}) &\Rightarrow \texttt{remove}(\texttt{size}() - 1) \end{aligned}$$

Although we will normally not discuss the `Stack`, `Deque` and FIFO `Queue` interfaces very often in subsequent chapters, the terms `Stack` and `Deque` are sometimes used in the names of data structures that implement the `List` interface. When this happens, it is to highlight the fact that these data structures can be used to implement the `Stack` or `Deque` interface very efficiently. For example, the `ArrayDeque` class is an implementation of the `List` interface that can implement the `Deque` operations in constant (amortized) time per operation.

1.1.3 The `USet` Interface: Unordered Sets

The `USet` interface represents an unordered set of elements. This is a *set* in the mathematical sense. A `USet` contains `n` *distinct* elements; no element appears more than once; the elements are in no specific order. A `USet` supports the following operations:

1. `size`(): return the number, `n`, of elements in the set
2. `add`(`x`): add the element `x` to the set if not already present;
   Add `x` to the set provided that there is no element `y` in the set such that `x` equals `y`.
   Return `true` if `x` was added to the set and `false` otherwise.

3. `remove(x)`: remove `x` from the set;
   Find an element `y` in the set such that `x` equals `y` and remove `y`. Return `y`, or `null` if no such element exists.

4. `find(x)`: find `x` in the set if it exists;
   Find an element `y` in the set such that `y` equals `x`. Return `y`, or `null` if no such element exists.

These definitions are a bit fussy about distinguishing `x`, the element we are removing or finding, from `y`, the element we remove or find. This is because `x` and `y` might actually be distinct objects that are nevertheless treated as equal. This is a very useful distinction since it allows for the creation of *dictionaries* or *maps* that map keys onto values. This is done by creating a compound object called a `Pair` that contains a *key* and a *value*. Two `Pair`s are treated as equal if their keys are equal. By storing `Pair`s in a `USet`, we can find the value associated with any key `k` by creating a `Pair`, `x`, with key `k` and using the `find(x)` method.

1.1.4 The `SSet` Interface: Sorted Sets

The `SSet` interface represents a sorted set of elements. An `SSet` stores elements from some total order, so that any two elements `x` and `y` can be compared. In code examples, this will be done with a method called `compare(x, y)` in which

$$\texttt{compare}(\texttt{x},\texttt{y}) \begin{cases} < 0 & \text{if } \texttt{x} < \texttt{y} \\ > 0 & \text{if } \texttt{x} > \texttt{y} \\ = 0 & \text{if } \texttt{x} = \texttt{y} \end{cases}$$

An `SSet` supports the `size()`, `add(x)`, and `remove(x)` methods with exactly the same semantics as in the `USet` interface. The difference between a `USet` and an `SSet` is in the `find(x)` method:

4. `find(x)`: locate `x` in the sorted set;
   Find the smallest element `y` in the set such that $\texttt{y} > \texttt{x}$. Return `y` or `null` if no such element exists.

This version of the `find(x)` operation is sometimes referred to as a *successor search*. It differs in a fundamental way from `USet.find(x)` since it returns a meaningful result even when there is no element in the set that is equal to `x`.

The distinction between the `USet` and `SSet` `find(x)` operations is very important and is very often missed. The extra functionality provided by an `SSet` usually comes with a price that includes both a larger running time and a higher implementation complexity. For example, the `SSet` implementations discussed in this book all have `find(x)` operations with running times that are at least logarithmic in the size of the set. On the other hand, the implementation of a `USet` as a `ChainedHashTable` in Chapter 5 has a `find(x)` operation that runs in constant expected time. When choosing which of these structures to use, one should always use a `USet` unless the extra functionality offered by an `SSet` is really needed.

1.2 Mathematical Background

In this section, we review some mathematical notations and tools used throughout this book, including logarithms, big-Oh notation, and probability theory.

1.2.1 Logarithms

In this book, the expression $\log_b k$ denotes the *base-b logarithm* of k. That is, the unique value x that satisfies

$$b^x = k \ .$$

Most of the logarithms in this book are base 2 (*binary logarithms*), in which case we drop the base, so that $\log k$ is shorthand for $\log_2 k$.

Another logarithm that comes up several times in this book is the *natural logarithm*. Here we use the notation $\ln k$ to denote $\log_e k$, where e — *Euler's constant* — is given by

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \approx 2.71828 \ .$$

The natural logarithm comes up frequently because it is the value of a particularly common integral:

$$\int_1^k 1/x \, \mathrm{d}x = \ln k \ .$$

Two of the most common manipulations we do with logarithms are removing them from an exponent:

$$b^{\log_b k} = k$$

and changing the base of a logarithm:

$$\log_b k = \frac{\log_a k}{\log_a b} \ .$$

For example, we can use these two manipulations to compare the natural and binary logarithms

$$\ln k = \frac{\log k}{\log e} = \frac{\log k}{(\ln e)/(\ln 2)} = (\ln 2)(\log k) \approx 0.693147 \log k \ .$$

1.2.2 Factorials

In one or two places in this book, the *factorial* function is used. For a non-negative integer n, the notation $n!$ (pronounced "n factorial") denotes

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \ .$$

Factorials appear because $n!$ counts the number of distinct permutations, i.e., orderings, of n distinct elements. For the special case $n = 0$, $0!$ is defined as 1.

The quantity $n!$ can be approximated using *Stirling's Approximation*:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha(n)} \ ,$$

where

$$\frac{1}{12n+1} < \alpha(n) < \frac{1}{12n} \ .$$

Stirling's Approximation also approximates $\ln(n!)$:

$$\ln(n!) = n \ln n - n + \frac{1}{2}\ln(2\pi n) + \alpha(n)$$

(In fact, Stirling's Approximation is most easily proven by approximating $\ln(n!) = \ln 1 + \ln 2 + \cdots + \ln n$ by the integral $\int_1^n \ln n \, \mathrm{d}n = n \ln n - n + 1$.)

Related to the factorial function are the *binomial coefficients*. For a non-negative integer n and an integer $k \in \{0, \ldots, n\}$, the notation $\binom{n}{k}$ denotes:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \ .$$

The binomial coefficient $\binom{n}{k}$ (pronounced "n choose k") counts the number of subsets of an n element set that have size k, i.e., the number of ways of choosing k distinct integers from the set $\{1, \ldots, n\}$.

1.2.3 Asymptotic Notation

When analyzing data structures in this book, we will want to talk about the running times of various operations. The exact running times will, of course, vary from computer to computer and even from run to run on an individual computer. Therefore, instead of

analyzing running times exactly, we will use the so-called *big-Oh notation*: For a function $f(n)$, $O(f(n))$ denotes a set of functions,

$$O(f(n)) = \{g(n) : \text{there exists } c > 0, \text{ and } n_0 \text{ such that } g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0\} \ .$$

Thinking graphically, this set consists of the functions $g(n)$ where $c \cdot f(n)$ starts to dominate $g(n)$ when n is sufficiently large.

We generally use asymptotic notation to simplify functions. For example, in place of $5n \log n + 8n - 200$ we can write, simply, $O(n \log n)$. This is proven as follows:

$$\begin{aligned} 5n \log n + 8n - 200 &\leq 5n \log n + 8n \\ &\leq 5n \log n + 8n \log n \qquad \text{for } n \geq 2 \text{ (so that } \log n \geq 1) \\ &\leq 13n \log n \end{aligned}$$

which demonstrates that the fundtion $f(n) = 5n \log n + 8n - 200$ is in the set $O(\log n)$ using the constants $c = 13$ and $n_0 = 2$.

There are a number of useful shortcuts when using asymptotic notation. First:

$$O(n^{c_1}) \subset O(n^{c_2}) \ ,$$

for any $c_1 < c_2$. Second: For any constants $a, b, c > 0$,

$$O(a) \subset O(\log n) \subset O(n^b) \subset O(c^n) \ .$$

These inclusion relations can be multiplied by any positive value, and they still hold. For example, multiplying by n yields:

$$O(n) \subset O(n \log n) \subset O(n^{1+b}) \subset O(nc^n) \ .$$

Continuing in a long and distinguished tradition, we will abuse this notation by writing things like $f_1(n) = O(f(n))$ when what we really mean is $f_1(n) \in O(f(n))$. We will also make statements like "the running time of this operation is $O(f(n))$" when this statement should be "the running time of this operation is a member of $O(f(n))$." These shortcuts are mainly to avoid awkward language and to make it easier to use asymptotic notation within strings of equations.

A particularly strange example of this comes when we write statements like

$$T(n) = 2 \log n + O(1) \ .$$

Again, this would be more correctly written as

$$T(n) \leq 2\log n + [\text{some member of } O(1)] \ .$$

The expression $O(1)$ also brings up another issue. Since there is no variable in this expression, it may not be clear what variable is getting arbitrarily large. Without context, there is no way to tell. In the example above, since the only variable in the rest of the equation is n, we can assume that this should be read as $T(n) = 2\log n + O(f(n))$, where $f(n) = 1$.

In a few cases, we will use asymptotic notation on functions with more than one variable. There seems to be no standard for this, but for our purposes, the following definition is sufficient:

$$O(f(n_1, \ldots, n_k)) = \left\{ \begin{array}{l} g(n_1, \ldots, n_k) : \text{there exists } c > 0, \text{ and } z \text{ such that} \\ \quad g(n_1, \ldots, n_k) \leq c \cdot f(n_1, \ldots, n_k) \\ \quad\quad \text{for all } n_1, \ldots, n_k \text{ such that } g(n_1, \ldots, n_k) \geq z \end{array} \right\} \ .$$

This definition captures the situation we really care about: when the arguments $n_1, \ldots, n_k$ make g take on large values. This agrees with the univariate definition of $O(f(n))$ when $f(n)$ is an increasing function of n. The reader should be warned that, although this works for our purposes, other texts may treat multivariate functions and asymptotic notation differently.

1.2.4 Randomization and Probability

Some of the data structures presented in this book are *randomized*; they make random choices that are independent of the data being stored in them or the operations being performed on them. For this reason, performing the same set of operations more than once using these structures could result in different running times. When analyzing these data structures we are interested in their average or *expected* running times.

Formally, the running time of an operation on a randomized data structure is a random variable and we want to study its *expected value*. For a discrete random variable X taking on values in some countable universe U, the expected value of X, denoted by $\mathrm{E}[X]$ is given the by the formula

$$\mathrm{E}[X] = \sum_{x \in U} x \cdot \Pr\{X = x\} \ .$$

Here $\Pr\{\mathcal{E}\}$ denotes the probability that the event $\mathcal{E}$ occurs. In all the examples in this book, these probabilities are only with respect to whatever random choices are made by the

randomized data structure; there is no assumption that the data stored in the structure is random or that the sequence of operations performed on the data structure is random.

One of the most important properties of expected values is *linearity of expectation*: For any two random variables X and Y,

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y] \ .$$

More generally, for any random variables $X_1, \ldots, X_k$,

$$\mathrm{E}\left[\sum_{i=1}^{k} X_k\right] = \sum_{i=1}^{k} \mathrm{E}[X_i] \ .$$

Linearity of expectation allows us to break down complicated random variables (like the left hand sides of the above equations) into sums of simpler random variables (the right hand sides).

A useful trick, that we will use repeatedly, is that of defining *indicator random variables*. These binary variables are useful when we want to count something and are best illustrated by an example. Suppose we toss a fair coin k times and we want to know the expected number of times the coin comes up heads. Intuitively, we know the answer is $k/2$, but if we try to prove it using the definition of expected value, we get

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=0}^{k} i \cdot \Pr\{X = i\} \\
&= \sum_{i=0}^{k} i \cdot \binom{k}{i} / 2^k \\
&= k \cdot \sum_{i=0}^{k-1} \binom{k-1}{i} / 2^k \\
&= k/2 \ .
\end{aligned}$$

This requires that we know enough to calculate that $\Pr\{X = i\} = \binom{k}{i}/2^k$, that we know the binomial identity $i\binom{k}{i} = k\binom{k-1}{i}$, and that we know the binomial identity $\sum_{i=0}^{k} \binom{k}{i} = 2^k$.

Using indicator variables and linearity of expectation makes things much easier: For each $i \in \{1, \ldots, k\}$, define the indicator random variable

$$I_i = \begin{cases} 1 & \text{if the } i\text{th coin toss is heads} \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$\mathrm{E}[I_i] = (1/2)1 + (1/2)0 = 1/2 \ .$$

Now, $X = \sum_{i=1}^{k} I_i$, so

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k} I_i\right] \\ &= \sum_{i=1}^{k} \mathrm{E}[I_i] \\ &= \sum_{i=1}^{k} 1/2 \\ &= k/2 \ . \end{aligned}$$

This is a bit more long-winded, but doesn't require that we know any magical identities or compute any non-trivial probabilities. Even better: It agrees with the intuition that we expect half the coins to come up heads precisely because each individual coin has probability $1/2$ of coming up heads.

1.3 The Model of Computation

In this book, we will analyze the theoretical running times of operations on the data structures we study. To do this precisely, we need a mathematical model of computation. For this, we use the $\mathtt{w}$-*bit word-RAM* model. In this model, we have access to a random access memory consisting of *cells*, each of which stores a $\mathtt{w}$-bit *word*. This implies a memory cell can represent, for example, any integer in the set $\{0, \ldots, 2^{\mathtt{w}} - 1\}$.

In the word-RAM model, basic operations on words take constant time. This includes arithmetic operations ($+$, $-$, $*$, $/$, $\%$), comparisons ($<$, $>$, $=$, $\leq$, $\geq$), and bitwise boolean operations (bitwise-AND, OR, and exclusive-OR).

Any cell can be read or written in constant time. Our computer's memory is managed by a memory management system from which we can allocate or deallocate a block of memory of any size we like. Allocating a block of memory of size k takes $O(k)$ time and returns a reference to the newly-allocated memory block. This reference is small enough to be represented by a single word.

The word-size, $\mathtt{w}$, is a very important parameter of this model. The only assumption we will make on $\mathtt{w}$ is that it is at least $\mathtt{w} \geq \log \mathtt{n}$, where $\mathtt{n}$ is the number of elements stored in any of our data structures. This is a fairly modest assumption, since otherwise a word is not even big enough to count the number of elements stored in the data structure.

Space is measured in words so that, when we talk about the amount of space used by a data structure, we are referring to the number of words of memory used by the structure.

All our data structures store values of a generic type `T` and we assume an element of type `T` occupies one word of memory. (In reality, we are storing references to objects of type `T`, and these references occupy only one word of memory.)

The `w`-bit word-RAM model is a fairly close match for modern desktop computers when $w = 32$ or $w = 64$. The data structures presented in this book don't use any special tricks that are not implementable on the JVM and most other architectures.

1.4 Code Samples

The code samples in this book are written in the C++ programming language. However to make the book accessible even to readers not familiar with all of C++'s constructs and keywords, the code samples have been simplified. For example, a reader won't find any of the keywords `public`, `protected`, `private`, or `static`. A reader also won't find much discussion about class hierarchies. Which interfaces a particular class implements or which class it extends, if relevant to the discussion, will be clear from the accompanying text.

These conventions should make most of the code samples understandable by anyone with a background in any of the languages from the ALGOL tradition, including B, C, C++, C#, D, Java, JavaScript, and so on. Readers who want the full details of all implementations are encouraged to look at the C++ source code that accompanies this book.

This book mixes mathematical analysis of running times with C++ source code for the algorithms being analyzed. This means that some equations contain variables also found in the source code. These variables are typeset consistently, both within the source code and within equations. The most common such variable is the variable `n` that, without exception, always refers to the number of items currently stored in the data structure.

1.5 List of Data Structures

The following table summarize the performance of data structures described in this book that implement each of the interfaces, `List`, `USet`, and `SSet`, described in Section 1.1.

List implementations			
	get(i)/set(i,x)	add(i,x)/remove(i)	
ArrayStack	$O(1)$	$O(1+\mathtt{n}-\mathtt{i})^{\mathrm{A}}$	§ 2.1
ArrayDeque	$O(1)$	$O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})^{\mathrm{A}}$	§ 2.4
DualArrayDeque	$O(1)$	$O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})^{\mathrm{A}}$	§ 2.5
RootishArrayStack	$O(1)$	$O(1+\mathtt{n}-\mathtt{i})^{\mathrm{A}}$	§ 2.6
DLList	$O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$	$O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$	§ 3.2
SEList	$O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\}/\mathtt{b})$	$O(\mathtt{b}+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\}/\mathtt{b})^{\mathrm{A}}$	§ 3.3
SkiplistList	$O(\log \mathtt{n})^{\mathrm{E}}$	$O(\log \mathtt{n})^{\mathrm{E}}$	§ 4.3

USet implementations			
	find(x)	add(x)/remove(x)	
ChainedHashTable	$O(1)^{\mathrm{E}}$	$O(1)^{\mathrm{AE}}$	§ 5.1
LinearHashTable	$O(1)^{\mathrm{E}}$	$O(1)^{\mathrm{AE}}$	§ 5.2

SSet implementations			
	find(x)	add(x)/remove(x)	
SkiplistSSet	$O(\log \mathtt{n})^{\mathrm{E}}$	$O(\log \mathtt{n})^{\mathrm{E}}$	§ 4.2
Treap	$O(\log \mathtt{n})^{\mathrm{E}}$	$O(\log \mathtt{n})^{\mathrm{E}}$	§ 7.2
ScapegoatTree	$O(\log \mathtt{n})$	$O(\log \mathtt{n})^{\mathrm{A}}$	§ 8.1
RedBlackTree	$O(\log \mathtt{n})$	$O(\log \mathtt{n})$	§ 9.2

(Priority) Queue implementations			
	findMin()	add(x)/remove()	
BinaryHeap	$O(1)$	$O(\log \mathtt{n})^{\mathrm{A}}$	§ 10.1
MeldableHeap	$O(1)$	$O(\log \mathtt{n})^{\mathrm{E}}$	§ 10.2

1.6 References

The List, USet, and SSet interfaces described in Section 1.1 are influenced by the Java Collections Framework [40]. These are essentially simplified versions of the List, Set/Map, and SortedSet/SortedMap interfaces found in the Java Collections Framework.

[A] Denotes an *amortized* running time. See Chapter 2.
[E] Denotes an *expected* running time. See Section 1.2.4.

For more information on basic probability, especially as it relates to computer science, see the textbook by Ross [48]. Another good reference, that covers both asymptotic notation and probability, is the textbook by Graham, Knuth, and Patashnik [30].

Chapter 2

Array-Based Lists

In this chapter, we study implementations of the `List` and `Queue` interfaces where the underlying data is stored in an array, called the *backing array*. The following table summarizes the running times of operations for the data structures presented in this chapter:

	`get(i)`/`set(i,x)`	`add(i,x)`/`remove(i)`
`ArrayStack`	$O(1)$	$O(\texttt{n}-\texttt{i})$
`ArrayDeque`	$O(1)$	$O(\min\{\texttt{i},\texttt{n}-\texttt{i}\})$
`DualArrayDeque`	$O(1)$	$O(\min\{\texttt{i},\texttt{n}-\texttt{i}\})$
`RootishArrayStack`	$O(1)$	$O(\texttt{n}-\texttt{i})$

Data structures that work by storing data in a single array have many advantages and limitations in common:

- Arrays offer constant time access to any value in the array. This is what allows `get(i)` and `set(i,x)` to run in constant time.

- Arrays are not very dynamic. Adding or removing an element near the middle of a list means that a large number of elements in the array need to be shifted to make room for the newly added element or to fill in the gap created by the deleted element. This is why the operations `add(i,x)` and `remove(i)` have running times that depend on `n` and `i`.

- Arrays cannot expand or shrink. When the number of elements in the data structure exceeds the size of the backing array, a new array needs to be allocated and the data from the old array needs to be copied into the new array. This is an expensive operation.

The third point is important. The running times cited in the table above do not include the cost of growing and shrinking the backing array. We will see that, if carefully managed, the cost of growing and shrinking the backing array does not add much to the cost of an *average* operation. More precisely, if we start with an empty data structure, and perform any sequence of m `add(i,x)` or `remove(i)` operations, then the total cost of growing and shrinking the backing array, over the entire sequence of m operations is $O(m)$. Although some individual operations are more expensive, the amortized cost, when amortized over all m operations, is only $O(1)$ per operation.

In this chapter, and throughout this book, it will be convenient to have arrays that keep track of their size. The usual C++ arrays do not do this, so we have defined a class, `array`, that keeps track of its length. The implementation of this class is straightforward. It is implemented as a standard C++ array, `a`, and an integer, `length`:

array
```
T *a;
int length;
```

The size of an `array` is specified at the time of creation:

array
```
array(int len) {
  length = len;
  a = new T[length];
}
```

The elements of an array can be indexed:

array
```
T& operator[](int i) {
  assert(i >= 0 && i < length);
  return a[i];
}
```

Finally, when one array is assigned to another, this is just a pointer manipulation that takes constant time:

array
```
array<T>& operator=(array<T> &b) {
  if (a != NULL) delete[] a;
  a = b.a;
  b.a = NULL;
  length = b.length;
  return *this;
}
```

2.1 ArrayStack: Fast Stack Operations Using an Array

An ArrayStack implements the list interface using an array a, called the *backing array*. The list element with index i is stored in a[i]. At most times, a is larger than strictly necessary, so an integer n is used to keep track of the number of elements actually stored in a. In this way, the list elements are stored in a[0],...,a[n − 1] and, at all times, a.length ≥ n.

ArrayStack

```
array<T> a;
int n;
int size() {
  return n;
}
```

2.1.1 The Basics

Accessing and modifying the elements of an ArrayStack using get(i) and set(i, x) is trivial. After performing any necessary bounds-checking we simply return or set, respectively, a[i].

ArrayStack

```
T get(int i) {
  return a[i];
}
T set(int i, T x) {
  T y = a[i];
  a[i] = x;
  return y;
}
```

The operations of adding and removing elements from an ArrayStack are illustrated in Figure 2.1. To implement the add(i, x) operation, we first check if a is already full. If so, we call the method resize() to increase the size of a. How resize() is implemented will be discussed later. For now, it is sufficient to know that, after a call to resize(), we can be sure that a.length > n. With this out of the way, we now shift the elements a[i], ..., a[n − 1] right by one position to make room for x, set a[i] equal to x, and increment n.

ArrayStack

```
void add(int i, T x) {
  if (n + 1 > a.length) resize();
  for (int j = n; j > i; j--)
    a[j] = a[j - 1];
  a[i] = x;
  n++;
}
```

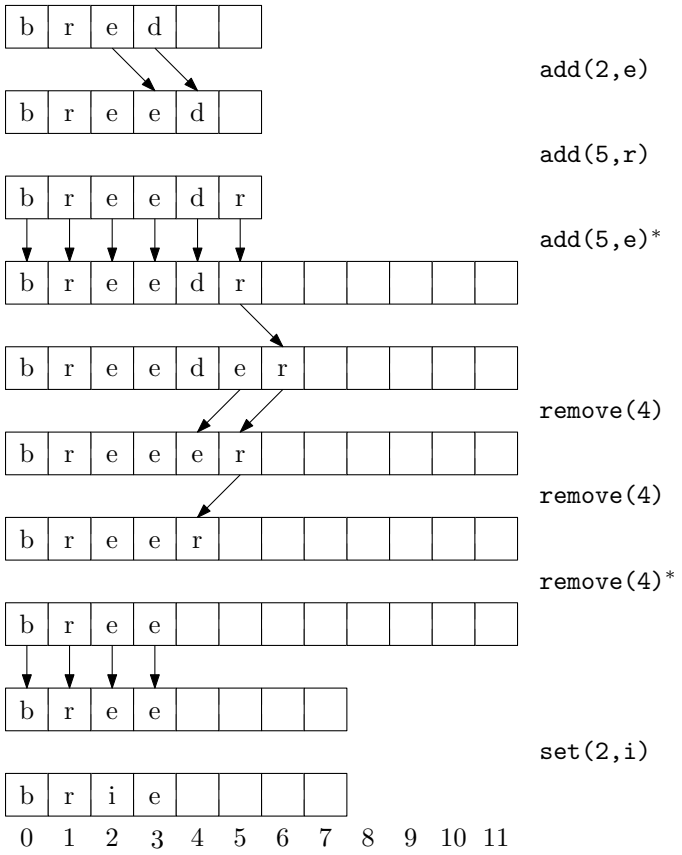


Figure 2.1: A sequence of `add(i,x)` and `remove(i)` operations on an `ArrayStack`. Arrows denote elements being copied. Operations that result in a call to `resize()` are marked with an asterisk.

If we ignore the cost of the potential call to `resize`(), the cost of the `add`(`i`, `x`) operation is proportional to the number of elements we have to shift to make room for `x`. Therefore the cost of this operation (ignoring the cost of resizing `a`) is $O(\texttt{n} - \texttt{i} + 1)$.

Implementing the `remove`(`i`) operation is similar. We shift the elements $\texttt{a}[\texttt{i}+1], \ldots, \texttt{a}[\texttt{n}-1]$ left by one position (overwriting `a`[`i`]) and decrease the value of `n`. After doing this, we check if `n` is getting much smaller than `a.length` by checking if $\texttt{a.length} \geq 3\texttt{n}$. If so, we call `resize`() to reduce the size of `a`.

ArrayStack

```
T remove(int i) {
    T x = a[i];
  for (int j = i; j < n - 1; j++)
    a[j] = a[j + 1];
  n--;
  if (a.length >= 3 * n) resize();
  return x;
}
```

If we ignore the cost of the `resize`() method, the cost of a `remove`(`i`) operation is proportional to the number of elements we shift, which is $O(\texttt{n} - \texttt{i})$.

2.1.2 Growing and Shrinking

The `resize`() method is fairly straightforward; it allocates a new array `b` whose size is $2\texttt{n}$ and copies the `n` elements of `a` into the first `n` positions in `b`, and then sets `a` to `b`. Thus, after a call to `resize`(), $\texttt{a.length} = 2\texttt{n}$.

ArrayStack

```
void resize() {
  array<T> b(max(2 * n, 1));
  for (int i = 0; i < n; i++)
    b[i] = a[i];
  a = b;
}
```

Analyzing the actual cost of the `resize`() operation is easy. It allocates an array `b` of size $2\texttt{n}$ and copies the `n` elements of `a` into `b`. This takes $O(\texttt{n})$ time.

The running time analysis from the previous section ignored the cost of calls to `resize`(). In this section we analyze this cost using a technique known as *amortized analysis*. This technique does not try to determine the cost of resizing during each individual `add`(`i`, `x`) and `remove`(`i`) operation. Instead, it considers the cost of all calls to `resize`() during a sequence of m calls to `add`(`i`, `x`) or `remove`(`i`). In particular, we will show:

Lemma 2.1. *If an empty* `ArrayList` *is created and any sequence of* $m \geq 1$ *calls to* `add(i,x)` *and* `remove(i)` *are performed, then the total time spent during all calls to* `resize()` *is* $O(m)$.

Proof. We will show that anytime `resize()` is called, the number of calls to `add` or `remove` since the last call to `resize()` is at least $\texttt{n}/2 - 1$. Therefore, if $\texttt{n}_i$ denotes the value of `n` during the ith call to `resize()` and r denotes the number of calls to `resize()`, then the total number of calls to `add(i,x)` or `remove(i)` is at least

$$\sum_{i=1}^{r}(\texttt{n}_i/2 - 1) \leq m \ ,$$

which is equivalent to

$$\sum_{i=1}^{r}\texttt{n}_i \leq 2m + 2r \ .$$

On the other hand, the total time spent during all calls to `resize()` is

$$\sum_{i=1}^{r} O(\texttt{n}_i) \leq O(m + r) = O(m) \ ,$$

which will prove the lemma since r is not more than m. All that remains is to show that the number of calls to `add(i,x)` or `remove(i)` between the $(i-1)$th and the ith call to `resize()` is at least $\texttt{n}_i/2$.

There are two cases to consider. In the first case, `resize()` is being called by `add(i,x)` because the backing array `a` is full, i.e., `a.length` $= \texttt{n} = \texttt{n}_i$. Consider the previous call to `resize()`: After this previous call, the size of `a` was `a.length`, but the number of elements stored in `a` was at most `a.length`$/2 = \texttt{n}_i/2$. But now the number of elements stored in `a` is $\texttt{n}_i =$ `a.length`, so there must have been at least $\texttt{n}_i/2$ calls to `add(i,x)` since the previous call to `resize()`.

The second case to consider is when `resize()` is being called by `remove(i)` because `a.length` $\geq 3\texttt{n} = 3\texttt{n}_i$. Again, after the previous call to `resize()` the number of elements stored in `a` was at least `a.length`$/2 - 1$.[1] Now there are $\texttt{n}_i \leq$ `a.length`$/3$ elements stored in `a`. Therefore, the number of `remove(i)` operations since the last call to `resize()` is at least

$$\texttt{a.length}/2 - 1 - \texttt{a.length}/3 = \texttt{a.length}/6 - 1 = (\texttt{a.length}/3)/2 - 1 \geq \texttt{n}_i/2 - 1 \ .$$

In either case, the number of calls to `add(i,x)` or `remove(i)` that occur between the $(i-1)$th call to `resize()` and the ith call to `resize()` is at least $\texttt{n}_i/2 - 1$, as required to complete the proof. □

[1] The -1 in this formula accounts for the special case that occurs when $\texttt{n} = 0$ and `a.length` $= 1$.

2.1.3 Summary

The following theorem summarizes the performance of an ArrayStack:

Theorem 2.1. *An* ArrayStack *implements the* List *interface. Ignoring the cost of calls to* resize(), *an* ArrayStack *supports the operations*

- get(i) *and* set(i, x) *in* $O(1)$ *time per operation; and*
- add(i, x) *and* remove(i) *in* $O(1 + n - i)$ *time per operation.*

Furthermore, beginning with an empty ArrayStack, *any sequence of* m add(i, x) *and* remove(i) *operations results in a total of* $O(m)$ *time spent during all calls to* resize().

The ArrayStack is an efficient way to implement a Stack. In particular, we can implement push(x) as add(n, x) and pop() as remove(n − 1), in which case these operations will run in $O(1)$ amortized time.

2.2 FastArrayStack: An Optimized ArrayStack

Much of the work done by an ArrayStack involves shifting (by add(i, x) and remove(i)) and copying (by resize()) of data. In the implementations shown above, this was done using for loops. It turns out that many programming environments have specific functions that are very efficient at copying and moving blocks of data. In the C and C++ programming languages there is the memcpy(d, s, n) function. In Java there is the System.arraycopy(s, i, d, j, n) method.

FastArrayStack

```
void resize() {
  array<T> b(max(1, 2*n));
  memcpy(b+0, a+0, n*sizeof(T));
  a = b;
}
void add(int i, T x) {
  if (n + 1 > a.length) resize();
  memcpy(a + i + 1, a + i, (n-i)*sizeof(T));
  a[i] = x;
  n++;
}
```

These functions are usually highly optimized and may even use special machine instructions that can do this copying much faster than we could do using a for loop.

Although using these functions does not asymptotically decrease the running times, it can still be a worthwhile optimization. In the Java implementations here, the use of `System.arraycopy(s, i, d, j, n)` resulted in speedups of a factor of 2-3 depending on the types of operations performed.

2.3 `ArrayQueue`: An Array-Based Queue

In this section, we present the `ArrayQueue` data structure, which implements a FIFO (first-in-first-out) queue; elements are removed (using the `remove()` operation) from the queue in the same order they are added (using the `add(x)` operation).

Notice that an `ArrayStack` is a poor choice for an implementation of a FIFO queue. The reason is that we must choose one end of the list to add to and then remove from the other end. One of the two operations must work on the head of the list, which involves calling `add(i, x)` or `remove(i)` with a value of `i` = 0. This gives a running time of $\Theta(n)$.

To obtain an efficient array-based implementation of a queue, we first notice that the problem would be easy if we had an infinite array `a`. We could maintain one index `j` that keeps track of the next element to remove and an integer `n` that counts the number of elements in the queue. The queue elements would always be stored in

$$\texttt{a}[\texttt{j}], \texttt{a}[\texttt{j}+1], \ldots, \texttt{a}[\texttt{j}+\texttt{n}-1] \ .$$

Initially, both `j` and `n` would be set to 0. To add an element, we would place it in $\texttt{a}[\texttt{j}+\texttt{n}]$ and increment `n`. To remove an element, we would remove it from `a[j]`, increment `j`, and decrement `n`.

Of course, the problem with this solution is that it requires an infinite array. An `ArrayQueue` simulates this by using a finite array `a` and *modular arithmetic*. This is the kind of arithmetic used when we are talking about the time of day. For example 10 o'clock plus 5 hours gives 3 o'clock. Formally, we say that

$$10 + 5 = 15 \equiv 3 \pmod{12} \ .$$

We read the latter part of this equation as "15 is congruent to 3 modulo 12." We can also treat mod as a binary operator, so that

$$15 \bmod 12 = 3 \ .$$

More generally, for an integer a and positive integer m, $a \bmod m$ is the unique integer $r \in \{0, \ldots, m-1\}$ such that $a = r + km$ for some integer k. Less formally, the

value r is the remainder we get when we divide a by m. In many programming languages, including C++, the mod operator is represented using the % symbol.[2]

Modular arithmetic is useful for simulating an infinite array, since `i` mod `a.length` always gives a value in the range $0, \ldots,$ `a.length` $-$ `1`. Using modular arithmetic we can store the queue elements at array locations

$$\texttt{a[j\%a.length]}, \texttt{a[(j + 1)\%a.length]}, \ldots, \texttt{a[(j + n - 1)\%a.length]} \ .$$

This treats `a` like a *circular array* in which array indices exceeding `a.length` $-$ 1 "wrap around" to the beginning of the array.

The only remaining thing to worry about is taking care that the number of elements in the `ArrayQueue` does not exceed the size of `a`.

ArrayQueue

```
array<T> a;
int j;
int n;
```

A sequence of `add`(`x`) and `remove`() operations on an `ArrayQueue` is illustrated in Figure 2.2. To implement `add`(`x`), we first check if `a` is full and, if necessary, call `resize`() to increase the size of `a`. Next, we store `x` in `a`[(`j` + `n`)%`a.length`] and increment `n`.

ArrayQueue

```
bool add(T x) {
   if (n + 1 > a.length) resize();
   a[(j+n) % a.length] = x;
   n++;
   return true;
 }
```

To implement `remove`() we first store `a`[`j`] so that we can return it later. Next, we decrement `n` and increment `j` (modulo `a.length`) by setting `j` = (`j` + 1) mod `a.length`. Finally, we return the stored value of `a`[`j`]. If necessary, we may call `resize`() to decrease the size of `a`.

ArrayQueue

```
T remove() {
  T x = a[j];
  j = (j + 1) % a.length;
```

[2]This is sometimes referred to as the *brain-dead* mod operator since it does not correctly implement the mathematical mod operator when the first argument is negative.

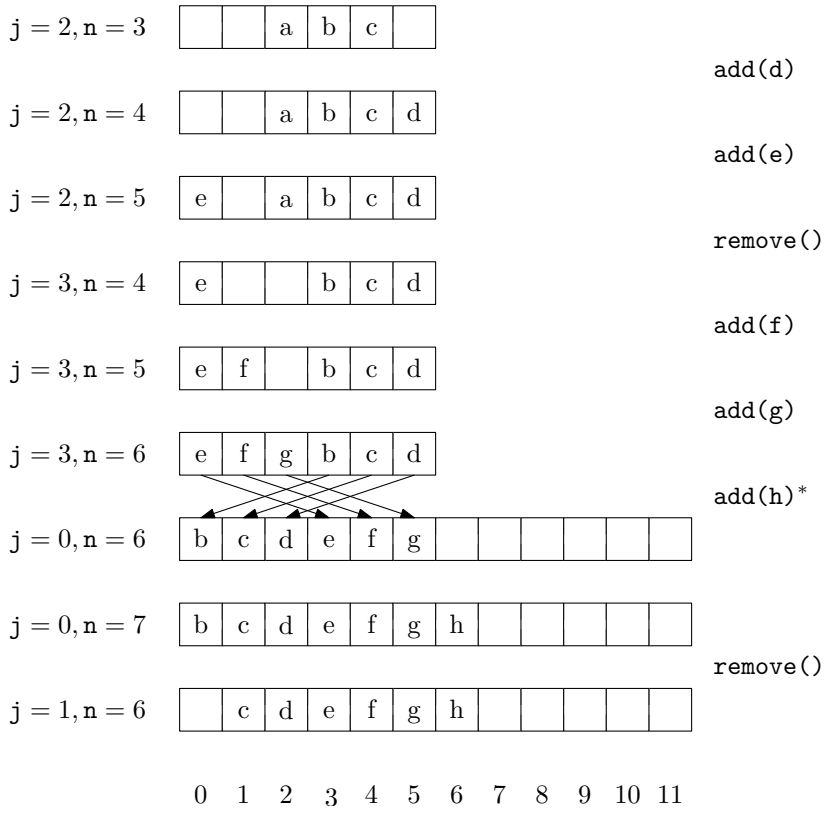


Figure 2.2: A sequence of `add(x)` and `remove(i)` operations on an `ArrayQueue`. Arrows denote elements being copied. Operations that result in a call to `resize()` are marked with an asterisk.

```
    n--;
    if (a.length >= 3*n) resize();
    return x;
  }
```

Finally, the `resize()` operation is very similar to the `resize()` operation of `ArrayStack`. It allocates a new array `b` of size $2n$ and copies

$$a[j], a[(j+1)\%a.length], \ldots, a[(j+n-1)\%a.length]$$

onto

$$b[0], b[1], \ldots, b[n-1]$$

and sets $j = 0$.

ArrayQueue

```
  void resize() {
    array<T> b(max(1, 2*n));
    for (int k = 0; k < n; k++)
      b[k] = a[(j+k)%a.length];
    a = b;
  }
```

2.3.1 Summary

The following theorem summarizes the performance of the `ArrayQueue` data structure:

Theorem 2.2. *An* `ArrayQueue` *implements the (FIFO)* `Queue` *interface. Ignoring the cost of calls to* `resize()`*, an* `ArrayQueue` *supports the operations* `add(x)` *and* `remove()` *in* $O(1)$ *time per operation. Furthermore, beginning with an empty* `ArrayQueue`*, any sequence of* m `add(i,x)` *and* `remove(i)` *operations results in a total of* $O(m)$ *time spent during all calls to* `resize()`*.*

2.4 `ArrayDeque`: Fast Deque Operations Using an Array

The `ArrayQueue` from the previous section is a data structure for representing a sequence that allows us to efficiently add to one end of the sequence and remove from the other end. The `ArrayDeque` data structure allows for efficient addition and removal at both ends. This structure implements the `List` interface using the same circular array technique used to represent an `ArrayQueue`.

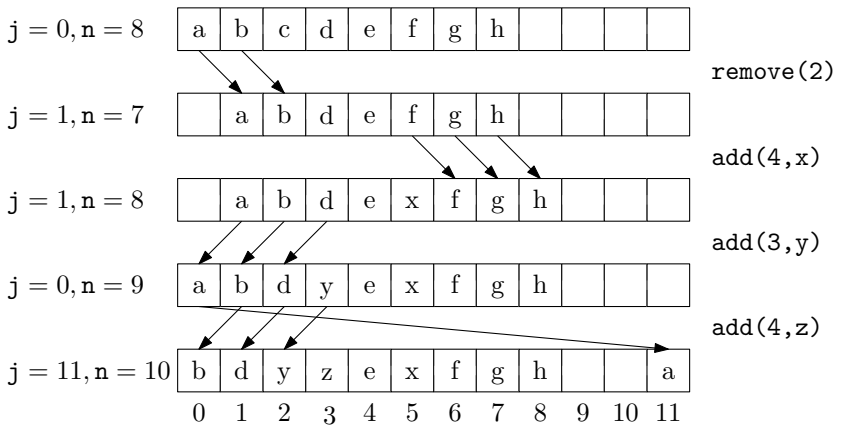


Figure 2.3: A sequence of `add(i,x)` and `remove(i)` operations on an `ArrayDeque`. Arrows denote elements being copied.

ArrayDeque
```
array<T> a;
int j;
int n;
```

The `get(i)` and `set(i,x)` operations on an `ArrayDeque` are straightforward. They get or set the array element `a[(j + i) mod a.length]`.

ArrayDeque
```
T get(int i) {
  return a[(j + i) % a.length];
}
T set(int i, T x) {
  T y = a[(j + i) % a.length];
  a[(j + i) % a.length] = x;
  return y;
}
```

The implementation of `add(i,x)` is a little more interesting. As usual, we first check if `a` is full and, if necessary, call `resize()` to resize `a`. Remember that we want this operation to be fast when `i` is small (close to 0) or when `i` is large (close to `n`). Therefore, we check if $\texttt{i} < \texttt{n}/2$. If so, we shift the elements $\texttt{a}[0], \ldots, \texttt{a}[\texttt{i}-1]$ left by one position. Otherwise ($\texttt{i} \geq \texttt{n}/2$), we shift the elements $\texttt{a}[\texttt{i}], \ldots, \texttt{a}[\texttt{n}-1]$ right by one position. See Figure 2.3 for an illustration of `add(i,x)` and `remove(x)` operations on an `ArrayDeque`.

ArrayDeque
```
void add(int i, T x) {
  if (n + 1 > a.length)  resize();
  if (i < n/2) { // shift a[0],..,a[i-1] left one position
    j = (j == 0) ? a.length - 1 : j - 1;
    for (int k = 0; k <= i-1; k++)
      a[(j+k)%a.length] = a[(j+k+1)%a.length];
  } else { // shift a[i],..,a[n-1] right one position
    for (int k = n; k > i; k--)
      a[(j+k)%a.length] = a[(j+k-1)%a.length];
  }
  a[(j+i)%a.length] = x;
  n++;
}
```

By doing the shifting in this way, we guarantee that $\texttt{add}(\texttt{i},\texttt{x})$ never has to shift more than $\min\{\texttt{i},\texttt{n}-\texttt{i}\}$ elements. Thus, the running time of the $\texttt{add}(\texttt{i},\texttt{x})$ operation (ignoring the cost of a `resize`() operation) is $O(1+\min\{\texttt{i},\texttt{n}-\texttt{i}\})$.

The $\texttt{remove}(\texttt{i})$ operation is similar. It either shifts elements $\texttt{a}[0],\ldots,\texttt{a}[\texttt{i}-1]$ right by one position or shifts the elements $\texttt{a}[\texttt{i}+1],\ldots,\texttt{a}[\texttt{n}-1]$ left by one position depending on whether $\texttt{i} < \texttt{n}/2$. Again, this means that $\texttt{remove}(\texttt{i})$ never spends more than $O(1+\min\{\texttt{i},\texttt{n}-\texttt{i}\})$ time to shift elements.

ArrayDeque
```
T remove(int i) {
    T x = a[(j+i)%a.length];
    if (i < n/2) {  // shift a[0],..,[i-1] right one position
      for (int k = i; k > 0; k--)
      a[(j+k)%a.length] = a[(j+k-1)%a.length];
    j = (j + 1) % a.length;
    } else {        // shift a[i+1],..,a[n-1] left one position
    for (int k = i; k < n-1; k++)
      a[(j+k)%a.length] = a[(j+k+1)%a.length];
    }
    n--;
    if (3*n < a.length) resize();
    return x;
}
```

2.4.1 Summary

The following theorem summarizes the performance of the `ArrayDeque` data structure:

Theorem 2.3. *An* `ArrayDeque` *implements the* `List` *interface. Ignoring the cost of calls to* `resize()`*, an* `ArrayDeque` *supports the operations*

- `get(i)` *and* `set(i,x)` *in* $O(1)$ *time per operation; and*
- `add(i,x)` *and* `remove(i)` *in* $O(1+\min\{i, n-i\})$ *time per operation.*

Furthermore, beginning with an empty `ArrayDeque`*, any sequence of* m `add(i,x)` *and* `remove(i)` *operations results in a total of* $O(m)$ *time spent during all calls to* `resize()`*.*

2.5 `DualArrayDeque`: Building a Deque from Two Stacks

Next, we present another data structure, the `DualArrayDeque` that achieves the same performance bounds as an `ArrayDeque` by using two `ArrayStack`s. Although the asymptotic performance of the `DualArrayDeque` is no better than that of the `ArrayDeque`, it is still worth studying since it offers a good example of how to make a sophisticated data structure by combining two simpler data structures.

A `DualArrayDeque` represents a list using two `ArrayStack`s. Recall that an `ArrayStack` is fast when the operations on it modify elements near the end. A `DualArrayDeque` places two `ArrayStack`s, called `front` and `back`, back-to-back so that operations are fast at either end.

DualArrayDeque
```
ArrayStack<T> front;
ArrayStack<T> back;
```

A `DualArrayDeque` does not explicitly store the number, `n`, of elements it contains. It doesn't need to, since it contains `n` = `front.size()`+`back.size()` elements. Nevertheless, when analyzing the `DualArrayDeque` we will still use `n` to denote the number of elements it contains.

DualArrayDeque
```
int size() {
  return front.size() + back.size();
}
```

The `front` `ArrayStack` contains list elements with indices $0, \ldots,$ `front.size()` -1, but stores them in reverse order. The `back` `ArrayStack` contains list elements with indices `front.size()`, $\ldots,$ `size()`-1 in the normal order. In this way, `get(i)` and `set(i,x)` translate into appropriate calls to `get(i)` or `set(i,x)` on either `front` or `back`, which take $O(1)$ time per operation.

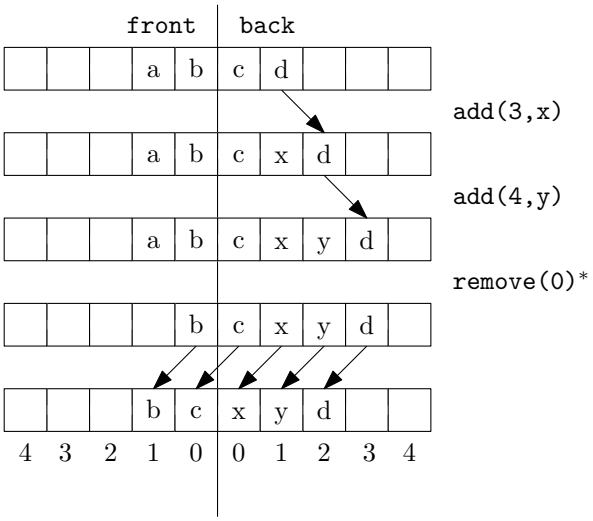


Figure 2.4: A sequence of `add(i,x)` and `remove(i)` operations on a `DualArrayDeque`. Arrows denote elements being copied. Operations that result in a rebalancing by `balance()` are marked with an asterisk.

DualArrayDeque

```
T get(int i) {
  if (i < front.size()) {
    return front.get(front.size() - i - 1);
  } else {
    return back.get(i - front.size());
  }
}
T set(int i, T x) {
  if (i < front.size()) {
    return front.set(front.size() - i - 1, x);
  } else {
    return back.set(i - front.size(), x);
  }
}
```

Note that, if an index $\texttt{i} < \texttt{front.size()}$, then it corresponds to the element of `front` at position $\texttt{front.size()} - \texttt{i} - 1$, since the elements of `front` are stored in reverse order.

Adding and removing elements from a `DualArrayDeque` is illustrated in Figure 2.4. The `add(i,x)` operation manipulates either `front` or `back`, as appropriate:

DualArrayDeque

```
void add(int i, T x) {
```

```
    if (i < front.size()) {
      front.add(front.size() - i, x);
    } else {
      back.add(i - front.size(), x);
    }
    balance();
  }
```

The add(i,x) method performs rebalancing of the two ArrayStacks front and back, by calling the balance() method. The implementation of balance() is described below, but for now it is sufficient to know that balance() ensures that, unless size() < 2, front.size() and back.size() do not differ by more than a factor of 3. In particular, $3 \cdot$ front.size() $\geq$ back.size() and $3 \cdot$ back.size() $\geq$ front.size().

Next we analyze the cost of add(i,x), ignoring the cost of the balance() operation. If i < front.size(), then add(i,x) becomes front.add(front.size() − i − 1,x). Since front is an ArrayStack, the cost of this is

$$O(\texttt{front.size}() - (\texttt{front.size}() - \texttt{i} - 1) + 1) = O(\texttt{i} + 1) \ . \tag{2.1}$$

On the other hand, if i $\geq$ front.size(), then add(i,x) becomes back.add(i − front.size(),x). The cost of this is

$$O(\texttt{back.size}() - (\texttt{i} - \texttt{front.size}()) + 1) = O(\texttt{n} - \texttt{i} + 1) \ . \tag{2.2}$$

Notice that the first case (2.1) occurs when i < n/4. The second case (2.2) occurs when i ≥ 3n/4. When n/4 $\leq$ i < 3n/4, we can't be sure whether the operation affects front or back, but in either case, the operation takes $O(\texttt{n}) = O(\texttt{i}) = O(\texttt{n} - \texttt{i})$ time, since i $\geq$ n/4 and n − i > n/4. Summarizing the situation, we have

$$\text{Running time of } \texttt{add}(\texttt{i},\texttt{x}) \leq \begin{cases} O(1+\texttt{i}) & \text{if } \texttt{i} < \texttt{n}/4 \\ O(\texttt{n}) & \text{if } \texttt{n}/4 \leq \texttt{i} < 3\texttt{n}/4 \\ O(1+\texttt{n}-\texttt{i}) & \text{if } \texttt{i} \geq 3\texttt{n}/4 \end{cases}$$

Thus, the running time of add(i,x) (ignoring the cost of the call to balance()) is $O(1 + \min\{\texttt{i}, \texttt{n} - \texttt{i}\})$.

The remove(i) operation, and its analysis, is similar to the add(i,x) operation.

DualArrayDeque

```
  T remove(int i) {
      T x;
```

```
    if (i < front.size()) {
            x = front.remove(front.size()-i-1);
    } else {
            x = back.remove(i-front.size());
    }
    balance();
    return x;
}
```

2.5.1 Balancing

Finally, we study the `balance()` operation performed by `add(i,x)` and `remove(i)`. This operation is used to ensure that neither `front` nor `back` gets too big (or too small). It ensures that, unless there are fewer than 2 elements, each of `front` and `back` contain at least `n/4` elements. If this is not the case, then it moves elements between them so that `front` and `back` contain exactly $\lfloor n/2 \rfloor$ elements and $\lceil n/2 \rceil$ elements, respectively.

DualArrayDeque

```
void balance() {
  if (3*front.size() < back.size()
      || 3*back.size() < front.size()) {
    int n = front.size() + back.size();
    int nf = n/2;
    array<T> af(max(2*nf, 1));
    for (int i = 0; i < nf; i++) {
      af[nf-i-1] = get(i);
    }
    int nb = n - nf;
    array<T> ab(max(2*nb, 1));
    for (int i = 0; i < nb; i++) {
      ab[i] = get(nf+i);
    }
    front.a = af;
    front.n = nf;
    back.a = ab;
    back.n = nb;
  }
}
```

There is not much to analyze. If the `balance()` operation does rebalancing, then it moves $\Theta(n)$ elements and this takes $O(n)$ time. This is bad, since `balance()` is called with each call to `add(i,x)` and `remove(i)`. However, the following lemma shows that, on average, `balance()` only spends a constant amount of time per operation.

Lemma 2.2. *If an empty* `DualArrayDeque` *is created and any sequence of* $m \geq 1$ *calls to* `add(i,x)` *and* `remove(i)` *are performed, then the total time spent during all calls to* `balance()` *is* $O(m)$.

Proof. We will show that, if `balance()` is forced to shift elements, then the number of `add(i,x)` and `remove(i)` operations since the last time `balance()` shifted any elements is at least $\texttt{n}/2 - 1$. As in the proof of Lemma 2.1, this is sufficient to prove that the total time spent by `balance()` is $O(m)$.

We will perform our analysis using the *potential method*. Define the *potential* of the `DualArrayDeque` as

$$\Phi = |\texttt{front.size()} - \texttt{back.size()}| \ .$$

The interesting thing about this potential is that a call to `add(i,x)` or `remove(i)` that does not do any balancing can increase the potential by at most 1.

Observe that, immediately after a call to `balance()` that shifts elements, the potential, Φ_0, is at most 1, since

$$\Phi_0 = |\lfloor \texttt{n}/2 \rfloor - \lceil \texttt{n}/2 \rceil| \leq 1 \ .$$

Consider the situation immediately before a call to `balance()` that shifts elements and suppose, without loss of generality, that `balance()` is shifting elements because $3\texttt{front.size()} < \texttt{back.size()}$. Notice that, in this case,

$$\begin{aligned} \texttt{n} &= \texttt{front.size()} + \texttt{back.size()} \\ &< \texttt{back.size()}/3 + \texttt{back.size()} \\ &= \frac{4}{3}\texttt{back.size()} \end{aligned}$$

Furthermore, the potential at this point in time is

$$\begin{aligned} \Phi_1 &= \texttt{back.size()} - \texttt{front.size()} \\ &> \texttt{back.size()} - \texttt{back.size()}/3 \\ &= \frac{2}{3}\texttt{back.size()} \\ &> \frac{2}{3} \times \frac{3}{4}\texttt{n} \\ &= \texttt{n}/2 \end{aligned}$$

Therefore, the number of calls to `add(i,x)` or `remove(i)` since the last time `balance()` shifted elements is at least $\Phi_1 - \Phi_0 > \texttt{n}/2 - 1$. This completes the proof. □

2.5.2 Summary

The following theorem summarizes the performance of a DualArrayStack

Theorem 2.4. *A* DualArrayDeque *implements the* List *interface. Ignoring the cost of calls to* resize() *and* balance(), *a* DualArrayDeque *supports the operations*

- get(i) *and* set(i, x) *in* $O(1)$ *time per operation; and*
- add(i, x) *and* remove(i) *in* $O(1 + \min\{\mathtt{i}, \mathtt{n} - \mathtt{i}\})$ *time per operation.*

Furthermore, beginning with an empty DualArrayDeque, *any sequence of* m add(i, x) *and* remove(i) *operations results in a total of* $O(m)$ *time spent during all calls to* resize() *and* balance().

2.6 RootishArrayStack: A Space-Efficient Array Stack

One of the drawbacks of all previous data structures in this chapter is that, because they store their data in one or two arrays, and they avoid resizing these arrays too often, the arrays are frequently not very full. For example, immediately after a resize() operation on an ArrayStack, the backing array a is only half full. Even worse, there are times when only 1/3 of a contains data.

In this section, we discuss a data structure, the RootishArrayStack, that addresses the problem of wasted space. The RootishArrayStack stores n elements using $O(\sqrt{\mathtt{n}})$ arrays. In these arrays, at most $O(\sqrt{\mathtt{n}})$ array locations are unused at any time. All remaining array locations are used to store data. Therefore, these data structures waste at most $O(\sqrt{\mathtt{n}})$ space when storing n elements.

A RootishArrayStack stores its elements in a list of r arrays called *blocks* that are numbered $0, 1, \ldots, \mathtt{r} - 1$. See Figure 2.5. Block b contains $b + 1$ elements. Therefore, all r blocks contain a total of

$$1 + 2 + 3 + \cdots + \mathtt{r} = \mathtt{r}(\mathtt{r} + 1)/2$$

elements. The above formula (allegedly discovered by the mathematician Gauß at the age of 9) can be obtained as shown in Figure 2.6.

RootishArrayStack

```
ArrayStack<T*> blocks;
int n;
```

The elements of the list are laid out in the blocks as we might expect. The list element with index 0 is stored in block 0, the elements with list indices 1 and 2 are stored in

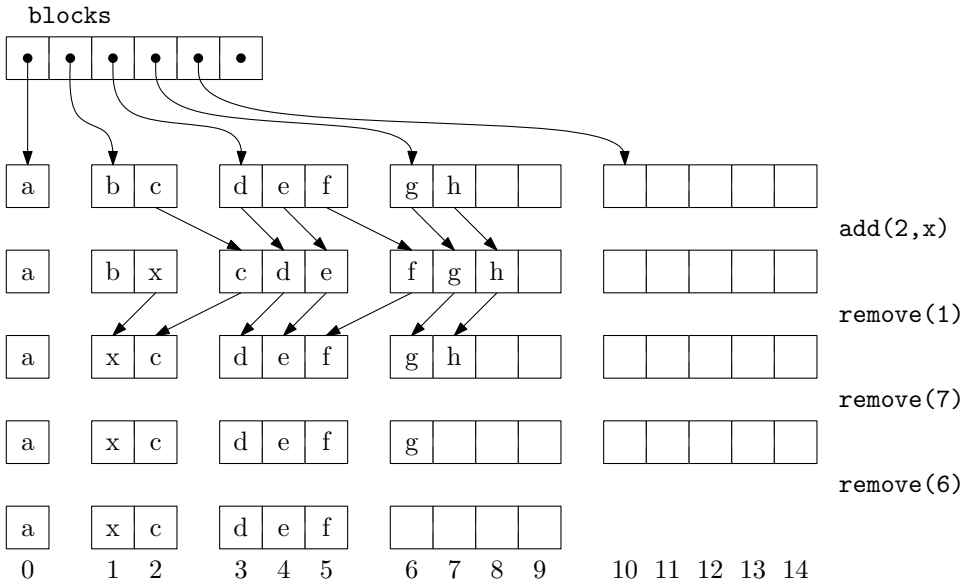


Figure 2.5: A sequence of `add(i,x)` and `remove(i)` operations on a `RootishArrayStack`. Arrows denote elements being copied.

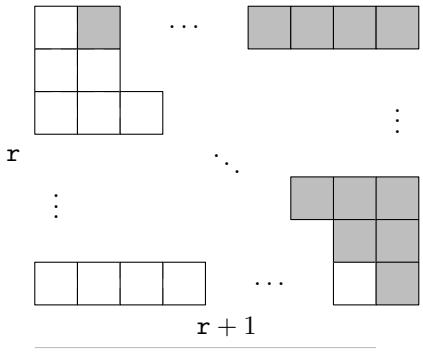


Figure 2.6: The number of white squares is $1 + 2 + 3 + \cdots + \mathtt{r}$. The number of shaded squares is the same. Together the white and shaded squares make a rectangle consisting of $\mathtt{r}(\mathtt{r}+1)$ squares.

block 1, the elements with list indices 3, 4, and 5 are stored in block 2, and so on. The main problem we have to address is that of determining, given an index i, which block contains i as well as the index corresponding to i within that block.

Determining the index of i within its block turns out to be easy. If index i is in block b, then the number of elements in blocks $0, \ldots, b-1$ is $b(b+1)/2$. Therefore, i is stored at location

$$j = i - b(b+1)/2$$

within block b. Somewhat more challenging is the problem of determining the value of b. The number of elements that have indices less than or equal to i is $i+1$. On the other hand, the number of elements in blocks 0,...,b is $(b+1)(b+2)/2$. Therefore, b is the smallest integer such that

$$(b+1)(b+2)/2 \geq i+1 \ .$$

We can rewrite this equation as

$$b^2 + 3b - 2i \geq 0 \ .$$

The corresponding quadratic equation $b^2 + 3b - 2i = 0$ has two solutions: $b = (-3 + \sqrt{9+8i})/2$ and $b = (-3 - \sqrt{9+8i})/2$. The second solution makes no sense in our application since it always gives a negative value. Therefore, we obtain the solution $b = (-3+\sqrt{9+8i})/2$. In general, this solution is not an integer, but going back to our inequality, we want the smallest integer b such that $b \geq (-3+\sqrt{9+8i})/2$. This is simply

$$b = \left\lceil (-3+\sqrt{9+8i})/2 \right\rceil \ .$$

RootishArrayStack

```
int i2b(int i) {
    double db = (-3.0 + sqrt(9 + 8*i)) / 2.0;
    int b = (int)ceil(db);
    return b;
}
```

With this out of the way, the get(i) and set(i,x) methods are straightforward. We first compute the appropriate block b and the appropriate index j within the block and then perform the appropriate operation:

RootishArrayStack

```
T get(int i) {
    int b = i2b(i);
    int j = i - b*(b+1)/2;
```

```
    return blocks.get(b)[j];
}
T set(int i, T x) {
    int b = i2b(i);
    int j = i - b*(b+1)/2;
    T y = blocks.get(b)[j];
    blocks.get(b)[j] = x;
    return y;
}
```

If we use any of the data structures in this chapter for representing the `blocks` list, then `get(i)` and `set(i,x)` will each run in constant time.

The `add(i,x)` method will, by now, look familiar. We first check if our data structure is full, by checking if the number of blocks `r` is such that $\texttt{r}(\texttt{r}+1)/2 = \texttt{n}$ and, if so, we call `grow()` to add another block. With this done, we shift elements with indices $\texttt{i},\ldots,\texttt{n}-1$ to the right by one position to make room for the new element with index `i`:

RootishArrayStack

```
void add(int i, T x) {
    int r = blocks.size();
    if (r*(r+1)/2 < n + 1) grow();
    n++;
    for (int j = n-1; j > i; j--)
            set(j, get(j-1));
    set(i, x);
}
```

The `grow()` method does what we expect. It adds a new block:

RootishArrayStack

```
void grow() {
    blocks.add(blocks.size(), new T[blocks.size()+1]);
}
```

Ignoring the cost of the `grow()` operation, the cost of an `add(i,x)` operation is dominated by the cost of shifting and is therefore $O(1+\texttt{n}-\texttt{i})$, just like an `ArrayStack`.

The `remove(i)` operation is similar to `add(i,x)`. It shifts the elements with indices $\texttt{i}+1,\ldots,\texttt{n}$ left by one position and then, if there is more than one empty block, it calls the `shrink()` method to remove all but one of the unused blocks:

RootishArrayStack

```
T remove(int i) {
    T x = get(i);
```

```
    for (int j = i; j < n-1; j++)
            set(j, get(j+1));
    n--;
    int r = blocks.size();
    if ((r-2)*(r-1)/2 >= n) shrink();
    return x;
}
```

RootishArrayStack

```
void shrink() {
    int r = blocks.size();
    while (r > 0 && (r-2)*(r-1)/2 >= n) {
            delete [] blocks.remove(blocks.size()-1);
            r--;
    }
}
```

Once again, ignoring the cost of the `shrink`() operation, the cost of a `remove`(`i`) operation is dominated by the cost of shifting and is therefore $O(\mathtt{n} - \mathtt{i})$.

2.6.1 Analysis of Growing and Shrinking

The above analysis of `add`(`i`, `x`) and `remove`(`i`) does not account for the cost of `grow`() and `shrink`(). Note that, unlike the `ArrayStack.resize`() operation, `grow`() and `shrink`() do not do any copying of data. They only allocate or free an array of size `r`. In some environments, this takes only constant time, while in others, it may require $\Theta(\mathtt{r})$ time.

We note that, immediately after a call to `grow`() or `shrink`(), the situation is clear. The final block is completely empty and all other blocks are completely full. Another call to `grow`() or `shrink`() will not happen until at least $\mathtt{r}-1$ elements have been added or removed. Therefore, even if `grow`() and `shrink`() take $O(\mathtt{r})$ time, this cost can be amortized over at least $\mathtt{r} - 1$ `add`(`i`, `x`) or `remove`(`i`) operations, so that the amortized cost of `grow`() and `shrink`() is $O(1)$ per operation.

2.6.2 Space Usage

Next, we analyze the amount of extra space used by a `RootishArrayStack`. In particular, we want to count any space used by a `RootishArrayStack` that is not an array element currently used to hold a list element. We call all such space *wasted space*.

The `remove`(`i`) operation ensures that a `RootishArrayStack` never has more than 2 blocks that are not completely full. The number of blocks, `r`, used by a `RootishArrayStack`

that stores n elements therefore satisfies

$$(r-2)(r-1) \le n$$

Again, using the quadratic equation on this gives

$$r \le (3+\sqrt{1+4n})/2 = O(\sqrt{n})$$

The last two blocks have sizes r and $r-1$, so the space wasted by these two blocks is at most $2r-1 = O(\sqrt{n})$. If we store the blocks in (for example) an ArrayList, then the amount of space wasted by the List that stores those r blocks is also $O(r) = O(\sqrt{n})$. The other space needed for storing n and other accounting information is $O(1)$. Therefore, the total amount of wasted space in a RootishArrayStack is $O(\sqrt{n})$.

Next, we argue that this space usage is optimal for any data structure that starts out empty and can support the addition of one item at a time. More precisely, we will show that, at some point during the addition of n items, the data structure is wasting an amount of space at least in $\sqrt{n}$ (though it may be only wasted for a moment).

Suppose we start with an empty data structure and we add n items one at a time. At the end of this process, all n items are stored in the structure and they are distributed among a collection of r memory blocks. If $r \ge \sqrt{n}$, then the data structure must be using r pointers (or references) to keep track of these r blocks, and this is wasted space. On the other hand, if $r < \sqrt{n}$ then, by the pigeonhole principle, some block must have size at least $n/r > \sqrt{n}$. Consider the moment at which this block was first allocated. Immediately after it was allocated, this block was empty, and was therefore wasting $\sqrt{n}$ space. Therefore, at some point in time during the insertion of n elements, the data structure was wasting $\sqrt{n}$ space.

2.6.3 Summary

The following theorem summarizes the performance of the RootishArrayStack data structure:

Theorem 2.5. *A* RootishArrayStack *implements the* List *interface. Ignoring the cost of calls to* grow() *and* shrink(), *a* RootishArrayStack *supports the operations*

- get(i) *and* set(i, x) *in* $O(1)$ *time per operation; and*
- add(i, x) *and* remove(i) *in* $O(1+n-i)$ *time per operation.*

Furthermore, beginning with an empty `RootishArrayStack`*, any sequence of* m `add(i,x)` *and* `remove(i)` *operations results in a total of* $O(m)$ *time spent during all calls to* `grow()` *and* `shrink()`*.*

The space (measured in words)[3] *used by a* `RootishArrayStack` *that stores* `n` *elements is* $\mathtt{n} + O(\sqrt{\mathtt{n}})$*.*

2.6.4 Computing Square Roots

A reader who has had some exposure to models of computation may notice that the `RootishArrayStack`, as described above, does not fit into the usual word-RAM model of computation (Section 1.3) because it requires taking square roots. The square root operation is generally not considered a basic operation and is therefore not usually part of the word-RAM model.

In this section, we take time to show that the square root operation can be implemented efficiently. In particular, we show that for any integer $\mathtt{x} \in \{0, \ldots, \mathtt{n}\}$, $\lfloor\sqrt{\mathtt{x}}\rfloor$ can be computed in constant-time, after $O(\sqrt{\mathtt{n}})$ preprocessing that creates two arrays of length $O(\sqrt{\mathtt{n}})$. The following lemma shows that we can reduce the problem of computing the square root of $\mathtt{x}$ to the square root of a related value $\mathtt{x}'$.

Lemma 2.3. *Let* $\mathtt{x} \geq 1$ *and let* $\mathtt{x}' = \mathtt{x} - a$*, where* $0 \leq a \leq \sqrt{\mathtt{x}}$*. Then* $\sqrt{x'} \geq \sqrt{\mathtt{x}} - 1$*.*

Proof. It suffices to show that

$$\sqrt{\mathtt{x} - \sqrt{\mathtt{x}}} \geq \sqrt{\mathtt{x}} - 1 \ .$$

Square both sides of this inequality to get

$$\mathtt{x} - \sqrt{\mathtt{x}} \geq \mathtt{x} - 2\sqrt{\mathtt{x}} + 1$$

and gather terms to get

$$\sqrt{\mathtt{x}} \geq 1$$

which is clearly true for any $\mathtt{x} \geq 1$. □

Start by restricting the problem a little, and assume that $2^{\mathtt{r}} \leq \mathtt{x} < 2^{\mathtt{r}+1}$, so that $\lfloor \log \mathtt{x} \rfloor = \mathtt{r}$, i.e., $\mathtt{x}$ is an integer having $\mathtt{r}+1$ bits in its binary representation. We can take $\mathtt{x}' = \mathtt{x} - (\mathtt{x} \bmod 2^{\lfloor r/2 \rfloor})$. Now, $\mathtt{x}'$ satisfies the conditions of Lemma 2.3, so $\sqrt{\mathtt{x}} - \sqrt{\mathtt{x}'} \leq 1$. Furthermore, $\mathtt{x}'$ has all of its lower-order $\lfloor \mathtt{r}/2 \rfloor$ bits equal to 0, so there are only

$$2^{\mathtt{r}+1-\lfloor \mathtt{r}/2 \rfloor} \leq 4 \cdot 2^{\mathtt{r}/2} \leq 4\sqrt{\mathtt{x}}$$

[3]Recall Section 1.3 for a discussion of how memory is measured.

possible values of x'. This means that we can use an array, sqrttab, that stores the value of $\lfloor\sqrt{x'}\rfloor$ for each possible value of x'. A little more precisely, we have

$$\texttt{sqrttab}[i] = \left\lfloor \sqrt{i2^{\lfloor r/2\rfloor}} \right\rfloor \ .$$

In this way, sqrttab$[i]$ is within 2 of $\sqrt{x}$ for all $x \in \{i2^{\lfloor r/2\rfloor}, \ldots, (i+1)2^{\lfloor r/2\rfloor} - 1\}$. Stated another way, the array entry $s =$ sqrttab[x>>$\lfloor r/2\rfloor$] is either equal to $\lfloor\sqrt{x}\rfloor$, $\lfloor\sqrt{x}\rfloor - 1$, or $\lfloor\sqrt{x}\rfloor - 2$. From s we can determine the value of $\lfloor\sqrt{x}\rfloor$ by incrementing s until $(s+1)^2 > x$.

FastSqrt

```
int sqrt(int x, int r) {
  int s = sqrtab[x>>r/2];
  while ((s+1)*(s+1) <= x) s++; // executes at most twice
  return s;
}
```

Now, this only works for $x \in \{2^r, \ldots, 2^{r+1} - 1\}$ and sqrttab is a special table that only works for a particular value of $r = \lfloor\log x\rfloor$. To overcome this, we could compute $\lfloor\log n\rfloor$ different sqrttab arrays, one for each possible value of $\lfloor\log x\rfloor$. The sizes of these tables form an exponential sequence whose largest value is at most $4\sqrt{n}$, so the total size of all tables is $O(\sqrt{n})$.

However, it turns out that more than one sqrttab array is unnecessary; we only need one sqrttab array for the value $r = \lfloor\log n\rfloor$. Any value x with $\log x = r' < r$ can be *upgraded* by multiplying x by $2^{r-r'}$ and using the equation

$$\sqrt{2^{r-r'}x} = 2^{(r-r')/2}\sqrt{x} \ .$$

The quantity $2^{r-r'}x$ is in the range $\{2^r, \ldots, 2^{r+1} - 1\}$ so we can look up its square root in sqrttab. The following code implements this idea to compute $\lfloor\sqrt{x}\rfloor$ for all non-negative integers x in the range $\{0, \ldots, 2^{30} - 1\}$ using an array sqrttab of size 2^{16}.

FastSqrt

```
int sqrt(int x) {
  int rp = log(x);
  int upgrade = ((r-rp)/2) * 2;
  int xp = x << upgrade;  // xp has r or r-1 bits
  int s = sqrtab[xp>>(r/2)] >> (upgrade/2);
  while ((s+1)*(s+1) <= x) s++;  // executes at most twice
  return s;
}
```

Something we have taken for granted thus far is the question of how to compute $r' = \lfloor\log x\rfloor$. Again, this is a problem that can be solved with an array logtab of size

$2^{r/2}$. In this case, the code is particularly simple, since $\lfloor \log x \rfloor$ is just the index of the most significant 1 bit in the binary representation of x. This means that, for $x > 2^{r/2}$, we can right-shift the bits of x by r/2 positions before using it as an index into logtab. The following code does this using an array logtab of size 2^{16} to compute $\lfloor \log x \rfloor$ for all x in the range $\{1, \ldots, 2^{32} - 1\}$

FastSqrt
```
int log(int x) {
  if (x >= halfint)
    return 16 + logtab[x>>16];
  return logtab[x];
}
```

Finally, for completeness, we include the following code that initializes logtab and sqrttab:

FastSqrt
```
void inittabs() {
  sqrtab = new int[1<<(r/2)];
  logtab = new int[1<<(r/2)];
  for (int d = 0; d < r/2; d++)
    for (int k = 0; k < 1<<d; k++)
      logtab[1<<d+k] = d;
  int s = 1<<(r/4);                    // sqrt(2^(r/2))
  for (int i = 0; i < 1<<(r/2); i++) {
    if ((s+1)*(s+1) <= i << (r/2)) s++; // sqrt increases
    sqrtab[i] = s;
  }
}
```

To summarize, the computations done by the i2b(i) method can be implemented in constant time on the word-RAM using $O(\sqrt{n})$ extra memory to store the sqrttab and logtab arrays. These arrays can be rebuilt when n increases or decreases by a factor of 2, and the cost of this rebuilding can be amortized over the number of add(i, x) and remove(i) operations that caused the change in n in the same way that the cost of resize() is analyzed in the ArrayStack implementation.

2.7 Discussion and Exercises

Most of the data structures described in this chapter are folklore. They can be found in implementations dating back over 30 years. For example, implementations of stacks, queues, and deques which generalize easily to the ArrayStack, ArrayQueue and ArrayDeque structures described here are discussed by Knuth [36, Section 2.2.2].

Brodnik *et al.* [10] seem to have been the first to describe the `RootishArrayStack` and prove a $\sqrt{n}$ lower-bound like that in Section 2.6.2. They also present a different structure that uses a more sophisticated choice of block sizes in order to avoid computing square roots in the `i2b(i)` method. With their scheme, the block containing `i` is block $\lfloor \log(\mathtt{i}+1) \rfloor$, which is just the index of the leading 1 bit in the binary representation of $\mathtt{i}+1$. Some computer architectures provide an instruction for computing the index of the leading 1-bit in an integer.

A structure related to the `RootishArrayStack` is the 2-level *tiered-vector* of Goodrich and Kloss [29]. This structure supports `get(i,x)` and `set(i,x)` in constant time and `add(i,x)` and `remove(i)` in $O(\sqrt{\mathtt{n}})$ time. These running times are similar to what can be achieved with the more careful implementation of a `RootishArrayStack` discussed in Exercise 2.9.

Exercise 2.1. The `List` method `addAll(i,c)` inserts all elements of the `Collection c` into the list at position `i`. (The `add(i,x)` method is a special case where $\mathtt{c} = \{\mathtt{x}\}$.) Explain why, for the data structures in this chapter, it is not efficient to implement `addAll(i,c)` by repeated calls to `add(i,x)`. Design and implement a more efficient implementation.

Exercise 2.2. Design and implement a `RandomQueue`. This is an implementation of the `Queue` interface in which the `remove()` operation removes an element that is chosen uniformly at random among all the elements in the queue. The `add(x)` and `remove()` operations in a `RandomQueue` should take constant time.

Exercise 2.3. Design and implement a `Treque` (triple-ended queue). This is a `List` implementation in which `get(i)` and `set(i,x)` run in constant time and `add(i,x)` and `remove(i)` run in time

$$O(1 + \min\{\mathtt{i}, \mathtt{n}-\mathtt{i}, |\mathtt{n}/2 - \mathtt{i}|\}) \ .$$

With this running time, modifications are fast if they are near either end or near the middle of the list.

Exercise 2.4. Implement a method `rotate(r)` that "rotates" a `List` so that list item `i` becomes list item $(\mathtt{i} + \mathtt{r}) \bmod n$. When run on an `ArrayDeque`, or a `DualArrayDeque`, `rotate(r)` should run in $O(1 + \min\{r, n-r\})$.

Exercise 2.5. Modify the `ArrayDeque` implementation so that the shifting done by `add(i,x)`, `remove(i)`, and `resize()` is done using `System.arraycopy(s,i,d,j,n)`.

Exercise 2.6. Modify the `ArrayDeque` implementation so that it does not use the % operator (which is expensive on some systems). Instead, it should make use of the fact that, if

`a.length` is a power of 2, then `k%a.length=k&(a.length − 1)`. (Here, & is the bitwise-and operator.)

Exercise 2.7. Design and implement a variant of `ArrayDeque` that does not do any modular arithmetic at all. Instead, all the data sits in a consecutive block, in order, inside an array. When the data overruns the beginning or the end of this array, a modified `rebuild()` operation is performed. The amortized cost of all operations should be the same as in an `ArrayDeque`.

Hint: Making this work is really all about how a `rebuild()` operation is performed. You would like `rebuild()` to put the data structure into a state where the data cannot run off either end until at least `n/2` operations have been performed.

Test the performance of your implementation against the `ArrayDeque`. Optimize your implementation (by using `System.arraycopy(a, i, b, i, n)`) and see if you can get it to outperform the `ArrayDeque` implementation.

Exercise 2.8. Design and implement a version of a `RootishArrayStack` that has only $O(\sqrt{n})$ wasted space, but that can perform `add(i, x)` and `remove(i, x)` operations in $O(1+\min\{i, n-i\})$ time.

Exercise 2.9. Design and implement a version of a `RootishArrayStack` that has only $O(\sqrt{n})$ wasted space, but that can perform `add(i, x)` and `remove(i, x)` operations in $O(1 + \min\{\sqrt{n}, n - i\})$ time. (For an idea on how to do this, see Section 3.3.)

Exercise 2.10. Design and implement a version of a `RootishArrayStack` that has only $O(\sqrt{n})$ wasted space, but that can perform `add(i, x)` and `remove(i, x)` operations in $O(1 + \min\{i, \sqrt{n}, n - i\})$ time. (See Section 3.3 for ideas on how to achieve this.)

Chapter 3

Linked Lists

In this chapter, we continue to study implementations of the `List` interface, this time using pointer-based data structures rather than arrays. The structures in this chapter are made up of nodes that contain the list items. The nodes are linked together into a sequence using references (pointers). We first study singly-linked lists, which can implement `Stack` and (FIFO) `Queue` operations in constant time per operation.

Linked lists have advantages and disadvantages relative to array-based implementations of the `List` interface. The primary disadvantage is that we lose the ability to access any element using `get(i)` or `set(i, x)` in constant time. Instead, we have to walk through the list, one element at a time, until we reach the `i`th element. The primary advantage is that they are more dynamic: Given a reference to any list node `u`, we can delete `u` or insert a node adjacent to `u` in constant time. This is true no matter where `u` is in the list.

3.1 `SLList`: A Singly-Linked List

An `SLList` (singly-linked list) is a sequence of `Node`s. Each node `u` stores a data value `u.x` and a reference `u.next` to the next node in the sequence. For the last node `w` in the sequence, `w.next` = `null`

SLList

```
struct Node {
  T x;
  Node *next;
};
```

For efficiency, an `SLList` uses variables `head` and `tail` to keep track of the first and last node in the sequence, as well as an integer `n` to keep track of the length of the sequence:

SLList

```
Node *head;
Node *tail;
```

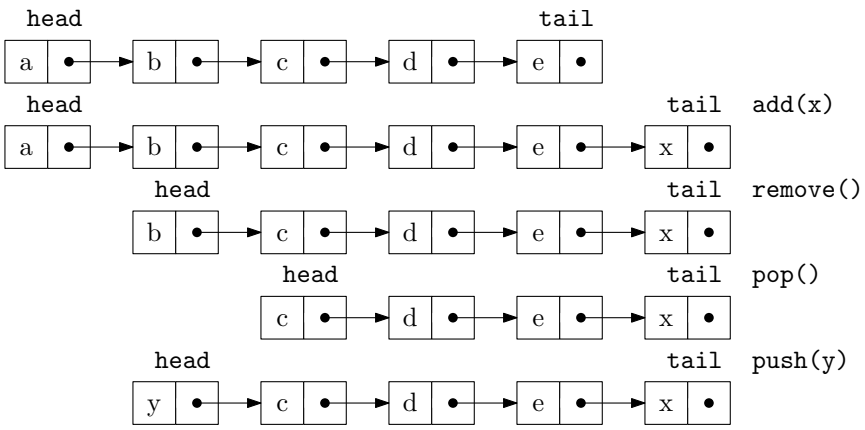


Figure 3.1: A sequence of `Queue` (`add(x)` and `remove()`) and `Stack` (`push(x)` and `pop()`) operations on an `SLList`.

```
int n;
```

A sequence of `Stack` and `Queue` operations on an `SLList` is illustrated in Figure 3.1.

An `SLList` can efficiently implement the `Stack` operations `push()` and `pop()` by adding and removing elements at the head of the sequence. The `push()` operation simply creates a new node `u` with data value `x`, sets `u.next` to the old head of the list and makes `u` the new head of the list. Finally, it increments `n` since the size of the `SLList` has increased by one:

SLList

```
T push(T x) {
  Node *u = new Node;
  u->x = x;
  u->next = head;
  head = u;
  if (n == 0)
    tail = u;
  n++;
  return x;
}
```

The `pop()` operation, after checking that the `SLList` is not empty, removes the head by setting `head = head.next` and decrementing `n`. A special case occurs when the last element is being removed, in which case `tail` is set to `null`:

SLList

```
T pop() {
  if (n == 0)  return NULL;
  T x = head->x;
  Node *u = head;
  head = head->next;
  delete u;
  if (--n == 0) tail = NULL;
  return x;
}
```

Clearly, both the push(x) and pop() operations run in $O(1)$ time.

3.1.1 Queue Operations

An SLList can also efficiently implement the FIFO queue operations add(x) and remove(). Removals are done from the head of the list, and are identical to the pop() operation:

SLList

```
T remove() {
  if (n == 0)  return NULL;
  T x = head->x;
  Node *u = head;
  head = head->next;
  delete u;
  if (--n == 0) tail = NULL;
  return x;
}
```

Additions, on the other hand, are done at the tail of the list. In most cases, this is done by setting tail.next = u, where u is the newly created node that contains x. However, a special case occurs when n = 0, in which case tail = head = null. In this case, both tail and head are set to u.

SLList

```
bool add(T x) {
  Node *u = new Node;
  u->x = x;
  if (n == 0) {
    head = u;
  } else {
    tail->next = u;
  }
  tail = u;
  n++;
```

```
    return true;
  }
```

Clearly, both add(x) and remove() take constant time.

3.1.2 Summary

The following theorem summarizes the performance of an SLList:

Theorem 3.1. *An* SLList *implements the* Stack *and (FIFO)* Queue *interfaces. The* push(x), pop(), add(x) *and* remove() *operations run in* $O(1)$ *time per operation.*

An SLList comes very close to implementing the full set of Deque operations. The only missing operation is removal from the tail of an SLList. Removing from the tail of an SLList is difficult because it requires updating the value of tail so that it points to the node w that precedes tail in the SLList; this is the node w such that w.next = tail. Unfortunately, the only way to get to w is by traversing the SLList starting at head and taking $n - 2$ steps.

3.2 DLList: A Doubly-Linked List

A DLList (doubly-linked list) is very similar to an SLList except that each node u in a DLList has references to both the node u.next that follows it and the node u.prev that precedes it.

DLList

```
struct Node {
  T x;
  Node *prev, *next;
};
```

When implementing an SLList, we saw that there were always some special cases to worry about. For example, removing the last element from an SLList or adding an element to an empty SLList requires special care so that head and tail are correctly updated. In a DLList, the number of these special cases increases considerably. Perhaps the cleanest way to take care of all these special cases in a DLList is to introduce a dummy node. This is a node that does not contain any data, but acts as a placeholder so that there are no special nodes; every node has both a next and a prev, with dummy acting as the node that follows the last node in the list and that precedes the first node in the list. In this way, the nodes of the list are (doubly-)linked into a cycle, as illustrated in Figure 3.2.

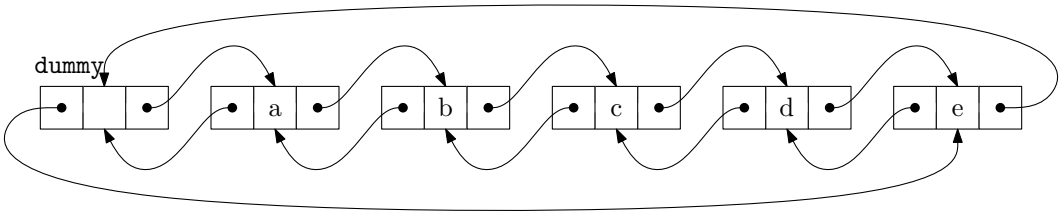


Figure 3.2: A DLList containing a,b,c,d,e.

DLList

```
Node dummy;
int n;
DLList() {
  dummy.next = &dummy;
  dummy.prev = &dummy;
  n = 0;
}
```

Finding the node with a particular index in a DLList is easy; we can either start at the head of the list (dummy.next) and work forward, or start at the tail of the list (dummy.prev) and work backward. This allows us to reach the ith node in $O(1+\min\{\texttt{i}, \texttt{n}-\texttt{i}\})$ time:

DLList

```
Node* getNode(int i) {
  Node* p;
  if (i < n / 2) {
    p = dummy.next;
    for (int j = 0; j < i; j++)
      p = p->next;
  } else {
    p = &dummy;
    for (int j = n; j > i; j--)
      p = p->prev;
  }
  return (p);
}
```

The get(i) and set(i, x) operations are now also easy. We first find the ith node and then get or set its x value:

DLList

```
T get(int i) {
    return getNode(i)->x;
```

```
}
T set(int i, T x) {
  Node* u = getNode(i);
  T y = u->x;
  u->x = x;
  return y;
}
```

The running time of these operations is dominated by the time it takes to find the $\texttt{i}$th node, and is therefore $O(1 + \min\{\texttt{i}, \texttt{n} - \texttt{i}\})$.

3.2.1 Adding and Removing

If we have a reference to a node $\texttt{w}$ in a DLList and we want to insert a node $\texttt{u}$ before $\texttt{w}$, then this is just a matter of setting $\texttt{u.next} = \texttt{w}$, $\texttt{u.prev} = \texttt{w.prev}$, and then adjusting $\texttt{u.prev.next}$ and $\texttt{u.next.prev}$. Thanks to the dummy node, there is no need to worry about $\texttt{w.prev}$ or $\texttt{w.next}$ not existing.

DLList

```
Node* addBefore(Node *w, T x) {
  Node *u = new Node;
  u->x = x;
  u->prev = w->prev;
  u->next = w;
  u->next->prev = u;
  u->prev->next = u;
  n++;
  return u;
}
```

Now, the list operation $\texttt{add}(\texttt{i}, \texttt{x})$ is trivial to implement. We find the $\texttt{i}$th node in the DLList and insert a new node $\texttt{u}$ that contains $\texttt{x}$ just before it.

DLList

```
void add(int i, T x) {
    addBefore(getNode(i), x);
}
```

The only non-constant part of the running time of $\texttt{add}(\texttt{i}, \texttt{x})$ is the time it takes to find the $\texttt{i}$th node (using $\texttt{getNode}(\texttt{i})$). Thus, $\texttt{add}(\texttt{i}, \texttt{x})$ runs in $O(1 + \min\{\texttt{i}, \texttt{n} - \texttt{i}\})$ time.

Removing a node $\texttt{w}$ from a DLList is easy. We need only adjust pointers at $\texttt{w.next}$ and $\texttt{w.prev}$ so that they skip over $\texttt{w}$. Again, the use of the dummy node eliminates the need to consider any special cases:

DLList
```
void remove(Node *w) {
  w->prev->next = w->next;
  w->next->prev = w->prev;
  delete w;
  n--;
}
```

Now the remove(i) operation is trivial. We find the node with index i and remove it:

DLList
```
T remove(int i) {
  Node *w = getNode(i);
  T x = w->x;
  remove(w);
  return x;
}
```

Again, the only expensive part of this operation is finding the ith node using getNode(i), so remove(i) runs in $O(1 + \min\{\mathtt{i}, \mathtt{n} - \mathtt{i}\})$ time.

3.2.2 Summary

The following theorem summarizes the performance of a DLList:

Theorem 3.2. *A* DLList *implements the* List *interface. The* get(i)*,* set(i, x)*,* add(i, x) *and* remove(i) *operations run in* $O(1 + \min\{\mathtt{i}, \mathtt{n} - \mathtt{i}\})$ *time per operation.*

It is worth noting that, if we ignore the cost of the getNode(i) operation, then all operations on a DLList take constant time. Thus, the only expensive part of operations on a DLList is finding the relevant node. Once we have the relevant node, adding, removing, or accessing the data at that node takes only constant time.

This is in sharp contrast to the array-based List implementations of Chapter 2; in those implementations, the relevant array item can be found in constant time. However, addition or removal requires shifting elements in the array and, in general, takes non-constant time.

For this reason, linked list structures are well-suited to applications where references to list nodes can be obtained through external means. For example, pointers to the nodes of a linked list could be stored in a USet. Then, to remove an item x from the linked list, the node that contains x can be found quickly using the Uset and the node can be removed from the list in constant time.

3.3 SEList: A Space-Efficient Linked List

One of the drawbacks of linked lists (besides the time it takes to access elements that are deep within the list) is their space usage. Each node in a DLList requires an additional two references to the next and previous nodes in the list. Two thirds of the fields in a Node are dedicated to maintaining the list and only one third of the fields are for storing data!

An SEList (space-efficient list) reduces this wasted space using a simple idea: Rather than store individual elements in a DLList, we store a block (array) containing several items. More precisely, an SEList is parameterized by a *block size* b. Each individual node in an SEList stores a block that can hold up to b + 1 elements.

It will turn out, for reasons that become clear later, that it will be helpful if we can do Deque operations on each block. The data structure we choose for this is a BDeque (bounded deque), derived from the ArrayDeque structure described in Section 2.4. The BDeque differs from the ArrayDeque in one small way: When a new BDeque is created, the size of the backing array a is fixed at b + 1 and it never grows or shrinks. The important property of a BDeque is that it allows for the addition or removal of elements at either the front or back in constant time. This will be useful as elements are shifted from one block to another.

SEList

```
class BDeque : public ArrayDeque<T> {
public:
  BDeque(int b) {
    n = 0;
    j = 0;
    array<int> z(b+1);
    a = z;
  }
  ~BDeque() { }
  // C++ Question: Why is this necessary?
  void add(int i, T x) {
    ArrayDeque<T>::add(i, x);
  }
  bool add(T x) {
    ArrayDeque<T>::add(size(), x);
    return true;
  }
  void resize() {}
};
```

An SEList is then a doubly-linked list of blocks:

SEList
```
class Node {
public:
  BDeque d;
  Node *prev, *next;
  Node(int b) : d(b) { }
};
```

SEList
```
int n;
Node dummy;
```

3.3.1 Space Requirements

An SEList places very tight restrictions on the number of elements in a block: Unless a block is the last block, then that block contains at least $b-1$ and at most $b+1$ elements. This means that, if an SEList contains n elements, then it has at most

$$n/(b-1)+1 = O(n/b)$$

blocks. The BDeque for each block contains an array of length $b+1$ but, for all blocks except the last, at most a constant amount of space is wasted in this array. The remaining memory used by a block is also constant. This means that the wasted space in an SEList is only $O(b+n/b)$. By choosing an appropriate value of b (ideally in $\Theta(\sqrt{n})$) we can make the space-overhead of an SEList approach the $\Omega(\sqrt{n})$ lower bound.[1]

3.3.2 Finding Elements

The first challenge we face with an SEList is finding the list item with a given index i. Note that the location of an element consists of two parts: The node u that contains the block that contains the element as well as the index j of the element within its block.

SEList
```
class Location {
public:
  Node *u;
  int j;
  Location() { }
  Location(Node *u, int j) {
    this->u = u;
    this->j = j;
  }
};
```

[1] See Section 2.6.2 for a proof of this lower bound.

To find the block that contains a particular element, we proceed in the same way as in a DLList. We either start at the front of the list and traverse in the forward direction or at the back of the list and traverse backwards until we reach the node we want. The only difference is that, each time we move from one node to the next, we skip over a whole block of elements.

SEList

```
void getLocation(int i, Location &ell) {
  if (i < n / 2) {
    Node *u = dummy.next;
    while (i >= u->d.size()) {
      i -= u->d.size();
      u = u->next;
    }
    ell.u = u;
    ell.j = i;
  } else {
    Node *u = &dummy;
    int idx = n;
    while (i < idx) {
      u = u->prev;
      idx -= u->d.size();
    }
    ell.u = u;
    ell.j = i - idx;
  }
}
```

Remember that, with the exception of at most one block, each block contains at least $\mathtt{b}-1$ elements, so each step in our search gets us $\mathtt{b}-1$ elements closer to the element we are looking for. If we are searching forward, this means we reach the node we want after $O(1+\mathtt{i}/\mathtt{b})$ steps. If we search backwards, we reach the node we want after $O(1+(\mathtt{n}-\mathtt{i})/\mathtt{b})$ steps. The algorithm takes the smaller of these two quantities depending on the value of i, so the time to locate the item with index i is $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\}/\mathtt{b})$.

Once we know how to locate the item with index i, the get(i) and set(i, x) operations translate into getting or setting a particular index in the correct block:

SEList

```
T get(int i) {
  Location l;
  getLocation(i, l);
  return l.u->d.get(l.j);
}
```

```
T set(int i, T x) {
  Location l;
  getLocation(i, l);
  T y = l.u->d.get(l.j);
  l.u->d.set(l.j, x);
  return y;
}
```

These operations are dominated by the time it takes to locate the item, so they also run in $O(1 + \min\{\mathtt{i}, \mathtt{n} - \mathtt{i}\}/\mathtt{b})$ time.

3.3.3 Adding an Element

Things start to get complicated when adding elements to an SEList. Before considering the general case, we consider the easier operation, add(x), in which x is added to the end of the list. If the last block is full (or does not exist because there are no blocks yet), then we first allocate a new block and append it to the list of blocks. Now that we are sure that the last block exists and is not full, we append x to the last block.

SEList

```
void add(T x) {
  Node *last = dummy.prev;
  if (last == &dummy || last->d.size() == b+1) {
    last = addBefore(&dummy);
  }
  last->d.add(x);
  n++;
}
```

Things get more complicated when we add to the interior of the list using add(i, x). We first locate i to get the node u whose block contains the ith list item. The problem is that we want to insert x into u's block, but we have to be prepared for the case where u's block already contains $\mathtt{b} + 1$ elements, so that it is full and there is no room for x.

Let $\mathtt{u}_0, \mathtt{u}_1, \mathtt{u}_2, \ldots$ denote u, u.next, u.next.next, and so on. We explore $\mathtt{u}_0, \mathtt{u}_1, \mathtt{u}_2, \ldots$ looking for a node that can provide space for x. Three cases can occur during our space exploration (see Figure 3.3):

1. We quickly (in $r + 1 \leq \mathtt{b}$ steps) find a node $\mathtt{u}_r$ whose block is not full. In this case, we perform r shifts of an element from one block into the next, so that the free space in $\mathtt{u}_r$ becomes a free space in $\mathtt{u}_0$. We can then insert x into $\mathtt{u}_0$'s block.

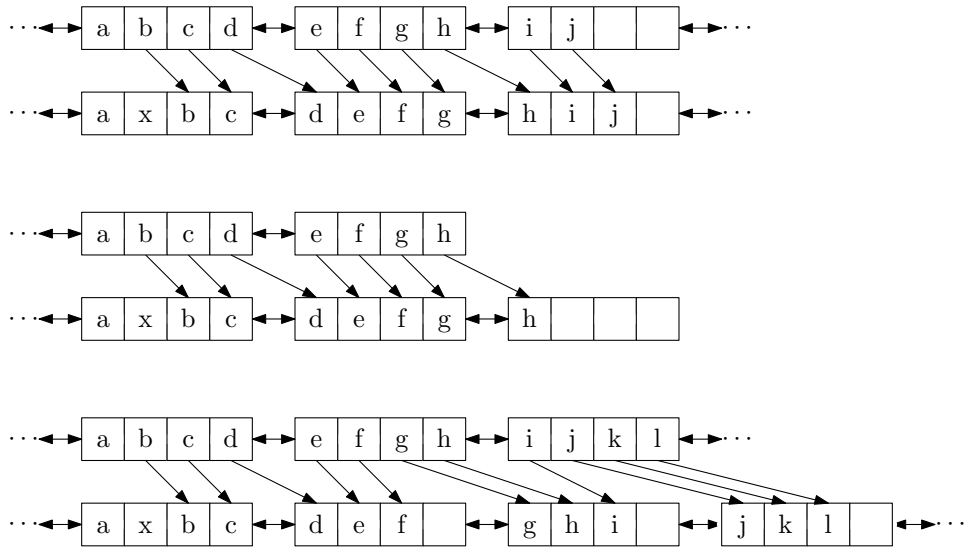


Figure 3.3: The three cases that occur during the addition of an item x in the interior of an SEList. (This SEList has block size b = 3.)

2. We quickly (in $r + 1 \leq b$ steps) run off the end of the list of blocks. In this case, we add a new empty block to the end of the list of blocks and proceed as in the first case.

3. After b steps we do not find any block that is not full. In this case, $u_0, \ldots, u_{b-1}$ is a sequence of b blocks that each contain $b + 1$ elements. We insert a new block u_b at the end of this sequence and *spread* the original $b(b + 1)$ elements so that each block of $u_0, \ldots, u_b$ contains exactly b elements. Now u_0's block contains only b elements so it has room for us to insert x.

SEList

```
void add(int i, T x) {
  if (i == n) {
    add(x);
    return;
  }
  Location l; getLocation(i, l);
  Node *u = l.u;
  int r = 0;
  while (r < b && u != &dummy && u->d.size() == b+1) {
    u = u->next;
    r++;
  }
  if (r == b) {      // found b blocks each with b+1 elements
    spread(l.u);
```

```
    u = l.u;
  }
  if (u == &dummy) {  // ran off the end of the list - add new node
    u = addBefore(u);
  }
  while (u != l.u) { // work backwards, shifting an element at each step
    u->d.add(0, u->prev->d.remove(u->prev->d.size()-1));
    u = u->prev;
  }
  u->d.add(l.j, x);
  n++;
}
```

The running time of the `add(i,x)` operation depends on which of the three cases above occurs. Cases 1 and 2 involve examining and shifting elements through at most `b` blocks and take $O(\texttt{b})$ time. Case 3 involves calling the `spread(u)` method, which moves $\texttt{b}(\texttt{b}+1)$ elements and takes $O(\texttt{b}^2)$ time. If we ignore the cost of Case 3 (which we will account for later with amortization) this means that the total running time to locate `i` and perform the insertion of `x` is $O(\texttt{b}+\min\{\texttt{i},\texttt{n}-\texttt{i}\}/\texttt{b})$.

3.3.4 Removing an Element

Removing an element, using the `remove(i)` method from an `SEList` is similar to adding an element. We first locate the node `u` that contains the element with index `i`. Now, we have to be prepared for the case where we cannot remove an element from `u` without causing `u`'s block to have size less than $\texttt{b}-1$, which is not allowed.

Again, let $\texttt{u}_0,\texttt{u}_1,\texttt{u}_2,\ldots$ denote `u`, `u.next`, `u.next.next`, We examine $\texttt{u}_0,\texttt{u}_1,\texttt{u}_2,\ldots$ in order looking for a node from which we can borrow an element to make the size of $\texttt{u}_0$'s block larger than $\texttt{b}-1$. There are three cases to consider (see Figure 3.4):

1. We quickly (in $r+1\le\texttt{b}$ steps) find a node whose block contains more than $\texttt{b}-1$ elements. In this case, we perform r shifts of an element from one block into the previous, so that the extra element in $\texttt{u}_r$ becomes an extra element in $\texttt{u}_0$. We can then remove the appropriate element from $\texttt{u}_0$'s block.

2. We quickly (in $r+1\le\texttt{b}$ steps) run off the end of the list of blocks. In this case, $\texttt{u}_r$ is the last block, and there is no requirement that $\texttt{u}_r$'s block contain at least $\texttt{b}-1$ elements. Therefore, we proceed as above, borrowing an element from $\texttt{u}_r$ to make an extra element in $\texttt{u}_0$. If this causes $\texttt{u}_r$'s block to become empty, then we remove it.

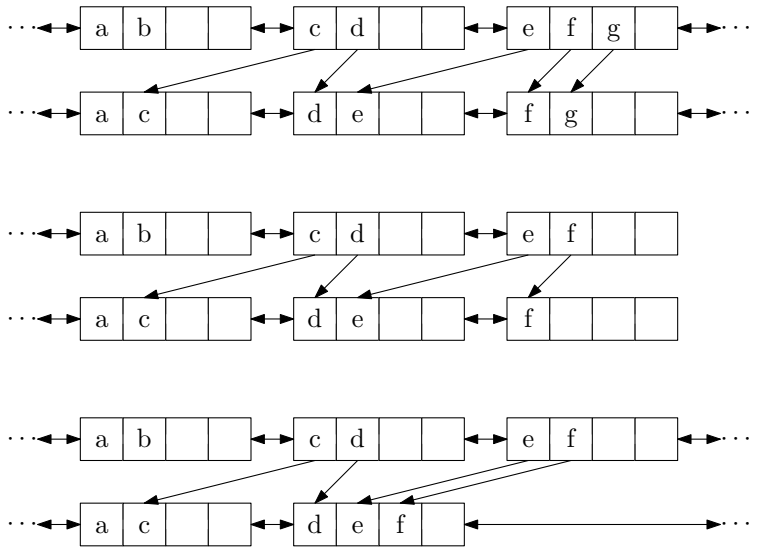


Figure 3.4: The three cases that occur during the removal of an item x in the interior of an SEList. (This SEList has block size b = 3.)

3. After b steps we do not find any block containing more than b − 1 elements. In this case, $u_0, \ldots, u_{b-1}$ is a sequence of b blocks that each contain b−1 elements. We *gather* these b(b − 1) elements into $u_0, \ldots, u_{b-2}$ so that each of these b − 1 blocks contains exactly b elements and we remove u_{b-1}, which is now empty. Now u_0's block contains b elements so we can remove the appropriate element from it.

SEList

```
T remove(int i) {
  Location l; getLocation(i, l);
  T y = l.u->d.get(l.j);
  Node *u = l.u;
  int r = 0;
  while (r < b && u != &dummy && u->d.size() == b - 1) {
    u = u->next;
    r++;
  }
  if (r == b) { // found b blocks each with b-1 elements
    gather(l.u);
  }
  u = l.u;
  u->d.remove(l.j);
  while (u->d.size() < b - 1 && u->next != &dummy) {
    u->d.add(u->next->d.remove(0));
    u = u->next;
```

```
    }
    if (u->d.size() == 0)
      remove(u);
    n--;
    return y;
  }
```

Like the add(i, x) operation, the running time of the remove(i) operation is $O(\mathtt{b} + \min\{\mathtt{i}, \mathtt{n} - \mathtt{i}\}/\mathtt{b})$ if we ignore the cost of the gather(u) method that occurs in Case 3.

3.3.5 Amortized Analysis of Spreading and Gathering

Next, we consider the cost of the gather(u) and spread(u) methods that may be executed by the add(i, x) and remove(i) methods. For completeness, here they are:

SEList

```
void spread(Node *u) {
  Node *w = u;
  for (int j = 0; j < b; j++) {
    w = w->next;
  }
  w = addBefore(w);
  while (w != u) {
    while (w->d.size() < b)
      w->d.add(0, w->prev->d.remove(w->prev->d.size()-1));
    w = w->prev;
  }
}
```

SEList

```
void gather(Node *u) {
  Node *w = u;
  for (int j = 0; j < b-1; j++) {
    while (w->d.size() < b)
      w->d.add(w->next->d.remove(0));
    w = w->next;
  }
  remove(w);
}
```

The running time of each of these methods is dominated by the two nested loops. Both the inner loop and outer loop execute at most $\mathtt{b}+1$ times, so the total running time of each of these methods is $O((\mathtt{b}+1)^2) = O(\mathtt{b}^2)$. However, the following lemma shows that these methods execute on at most one out of every b calls to add(i, x) or remove(i).

Lemma 3.1. *If an empty* SEList *is created and any sequence of* $m \geq 1$ *calls to* add(i, x) *and* remove(i) *are performed, then the total time spent during all calls to* spread() *and* gather() *is* $O(\mathtt{b}m)$.

Proof. We will use the potential method of amortized analysis. We say that a node u is *fragile* if u's block does not contain b elements (so that u is either the last node, or contains $\mathtt{b}-1$ or $\mathtt{b}+1$ elements). Any node whose block contains b elements is *rugged*. Define the *potential* of an SEList as the number of fragile nodes it contains. We will consider only the add(i, x) operation and its relation to the number of calls to spread(u). The analysis of remove(i) and gather(u) is identical.

Notice that, if Case 1 occurs during the add(i, x) method, then only one node, $\mathtt{u}_r$ has the size of its block changed. Therefore, at most one node, namely $\mathtt{u}_r$, goes from being rugged to being fragile. If Case 2 occurs, then a new node is created, and this node is fragile, but no other node changes sizes, so the number of fragile nodes increases by one. Thus, in either Case 1 or Case 2 the potential of the SEList increases by at most 1.

Finally, if Case 3 occurs, it is because $\mathtt{u}_0, \ldots, \mathtt{u}_{\mathtt{b}-1}$ are all fragile nodes. Then spread(u_0) is called and these b fragile nodes are replaced with $\mathtt{b}+1$ rugged nodes. Finally, x is added to $\mathtt{u}_0$'s block, making $\mathtt{u}_0$ fragile. In total the potential decreases by $\mathtt{b}-1$.

In summary, the potential starts at 0 (there are no nodes in the list). Each time Case 1 or Case 2 occurs, the potential increases by at most 1. Each time Case 3 occurs, the potential decreases by $\mathtt{b}-1$. The potential (which counts the number of fragile nodes) is never less than 0. We conclude that, for every occurrence of Case 3, there are at least $\mathtt{b}-1$ occurrences of Case 1 or Case 2. Thus, for every call to spread(u) there are at least b calls to add(i, x). This completes the proof. □

3.3.6 Summary

The following theorem summarizes the performance of the SEList data structure:

Theorem 3.3. *An* SEList *implements the* List *interface. Ignoring the cost of calls to* spread(u) *and* gather(u)*, an* SEList *with block size* b *supports the operations*

- get(i) *and* set(i, x) *in* $O(1+\min\{\mathtt{i}, \mathtt{n}-\mathtt{i}\}/\mathtt{b})$ *time per operation; and*
- add(i, x) *and* remove(i) *in* $O(\mathtt{b}+\min\{\mathtt{i}, \mathtt{n}-\mathtt{i}\}/\mathtt{b})$ *time per operation.*

Furthermore, beginning with an empty SEList*, any sequence of* m add(i, x) *and* remove(i) *operations results in a total of* $O(\mathtt{b}m)$ *time spent during all calls to* spread(u) *and* gather(u).

The space (measured in words)[2] *used by an* `SEList` *that stores* `n` *elements is* $\texttt{n} + O(\texttt{b} + \texttt{n}/\texttt{b})$.

The `SEList` is a tradeoff between an `ArrayList` and a `DLList` where the relative mix of these two structures depends on the block size `b`. At the extreme $\texttt{b} = 2$, each `SEList` node stores at most 3 values, which is really not much different than a `DLList`. At the other extreme, $\texttt{b} > \texttt{n}$, all the elements are stored in a single array, just like in an `ArrayList`. In between these two extremes lies a tradeoff between the time it takes to add or remove a list item and the time it takes to locate a particular list item.

3.4 Discussion and Exercises

Both singly-linked and doubly-linked lists are folklore, having been used in programs for over 40 years. They are discussed, for example, by Knuth [36, Sections 2.2.3–2.2.5]. Even the `SEList` data structure seems to be a well-known data structures exercise.

Exercise 3.1. Why is it not possible, in an `SLList` to use a dummy node to avoid all the special cases that occur in the operations `push(x)`, `pop()`, `add(x)`, and `remove()`?

Exercise 3.2. Describe and implement the `List` operations `get(i)`, `set(i,x)`, `add(i,x)` and `remove(i)` on an `SLList`. Each of these operations should run in $O(1 + \texttt{i})$ time.

[2] Recall Section 1.3 for a discussion of how memory is measured.

Chapter 4

Skiplists

In this chapter, we discuss a beautiful data structure: the skiplist, which has a variety of applications. Using a skiplist we can implement a `List` that is fast for all the operations `get(i)`, `set(i,x)`, `add(i,x)`, and `remove(i)`. We can also implement an `SSet` in which all operations run in $O(\log \mathtt{n})$ expected time.

Skiplists rely on *randomization* for their efficiency. In particular, a skiplist uses random coin tosses when an element is inserted to determine the height of that element. The performance of skiplists is expressed in terms of *expected* running times and lengths of paths. This expectation is taken over the random coin tosses used by the skiplist. In the implementation, the random coin tosses used by a skiplist are simulated using a pseudo-random number (or bit) generator.

4.1 The Basic Structure

Conceptually, a skiplist is a sequence of singly-linked lists $L_0, \ldots, L_h$, where each L_r contains a subset of the items in L_{r-1}. We start with the input list L_0 that contains `n` items and construct L_1 from L_0, L_2 from L_1, and so on. The items in L_r are obtained by tossing a coin for each element, `x`, in L_{r-1} and including `x` in L_r if the coin comes up heads. This process ends when we create a list L_r that is empty. An example of a skiplist is shown in Figure 4.1.

For an element, `x`, in a skiplist, we call the *height* of `x` the largest value r such that `x` appears in L_r. Thus, for example, elements that only appear in L_0 have height 0. Notice that the height of `x` corresponds to the following experiment: Toss a coin repeatedly until the first time it comes up tails. How many times did it come up heads? The answer, not surprisingly, is that the expected height of a node is 1. (We expect to toss the coin twice before getting tails, but we don't count the last toss.) The *height* of a skiplist is the height of its tallest node.

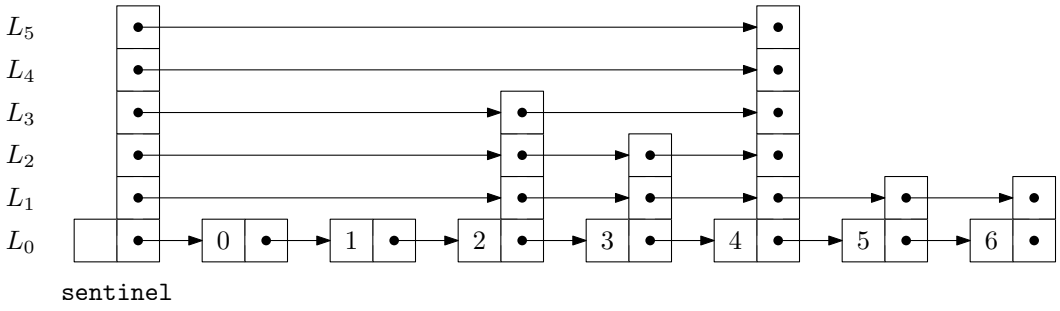


Figure 4.1: A skiplist containing seven elements.

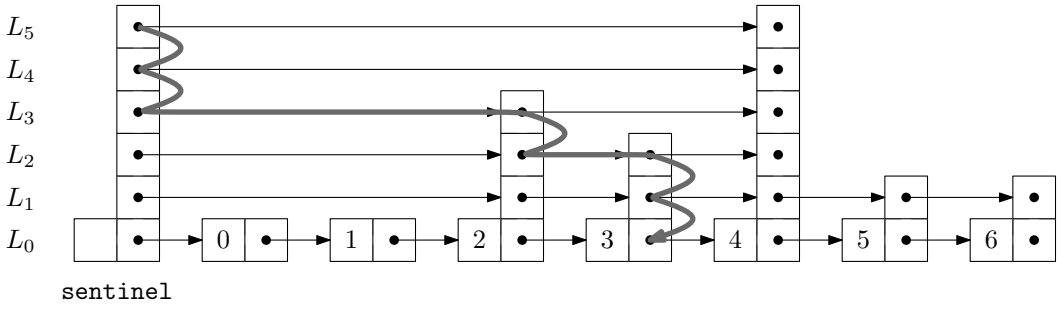


Figure 4.2: The search path for the node containing 4 in a skiplist.

At the head of every list is a special node, called the *sentinel*, that acts as a dummy node for the list. The key property of skiplists is that there is a short path, called the *search path*, from the sentinel in L_h to every node in L_0. Remembering how to construct a search path for a node, `u`, is easy (see Figure 4.2) : Start at the top left corner of your skiplist (the sentinel in L_h) and always go right unless that would overshoot `u`, in which case you should take a step down into the list below.

More precisely, to construct the search path for the node `u` in L_0 we start at the sentinel, `w`, in L_h. Next, we examine `w.next`. If `w.next` contains an item that appears before `u` in L_0, then we set `w` = `w.next`. Otherwise, we move down and continue the search at the occurrence of `w` in the list L_{h-1}. We continue this way until we reach the predecessor of `u` in L_0.

The following result, which we will prove in Section 4.4, shows that the search path is quite short:

Lemma 4.1. *The expected length of the search path for any node,* `u`*, in* L_0 *is at most* $2\log \mathtt{n} + O(1) = O(\log \mathtt{n})$.

A space-efficient way to implement a Skiplist is to define a Node, u, as consisting of a data value, x, and an array, next, of pointers, where u.next[i] points to u's successor in the list L_i. In this way, the data, x, in a node is stored only once, even though x may appear in several lists.

SkiplistSSet
```
struct Node {
  T x;
  int height;     // length of next
  Node *next[];
};
```

The next two sections of this chapter discuss two different applications of skiplists. In each of these applications, L_0 stores the main structure (a list of elements or a sorted set of elements). The primary difference between these structures is in how a search path is navigated; in particular, they differ in how they decide if a search path should go down into L_{r-1} or go right within L_r.

4.2 SkiplistSSet: An Efficient SSet Implementation

A SkiplistSSet uses a skiplist structure to implement the SSet interface. When used this way, the list L_0 stores the elements of the SSet in sorted order. The find(x) method works by following the search path for the smallest value y such that $y \geq x$:

SkiplistSSet
```
Node* findPredNode(T x) {
  Node *u = sentinel;
  int r = h;
  while (r >= 0) {
    while (  u->next[r] != NULL && compare(u->next[r]->x, x) < 0)
      u = u->next[r]; // go right in list r
    r--; // go down into list r-1
  }
  return u;
}
T find(T x) {
  Node *u = findPredNode(x);
  return u->next[0] == NULL ? NULL : u->next[0]->x;
}
```

Following the search path for y is easy: when situated at some node, u, in L_r, we look right to u.next[r].x. If $x > u.next[r].x$, then we take a step to the right in L_r, otherwise

we move down into L_{r-1}. Each step (right or down) in this search takes only constant time so, by Lemma 4.1, the expected running time of `find(x)` is $O(\log n)$.

Before we can add an element to a `SkipListSet`, we need a method to simulate tossing coins to determine the height, `k`, of a new node. We do this by picking a random integer, `z`, and counting the number of trailing 1s in the binary representation of `z`:[1]

SkiplistSSet

```
int pickHeight() {
  int z = rand();
  int k = 0;
  int m = 1;
  while ((z & m) != 0) {
    k++;
    m <<= 1;
  }
  return k;
}
```

To implement the `add(x)` method in a `SkiplistSSet` we search for `x` and then splice `x` into a few lists $L_0, \ldots, L_k$, where `k` is selected using the `pickHeight()` method. The easiest way to do this is to use an array, `stack`, that keeps track of the nodes at which the search path goes down from some list L_r into L_{r-1}. More precisely, `stack[r]` is the node in L_r where the search path proceeded down into L_{r-1}. The nodes that we modify to insert `x` are precisely the nodes `stack[0]`, ..., `stack[k]`. The following code implements this algorithm for `add(x)`:

SkiplistSSet

```
bool add(T x) {
  Node *u = sentinel;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u->next[r] != NULL && (comp = compare(u->next[r]->x, x)) < 0)
      u = u->next[r];
    if (u->next[r] != NULL && comp == 0)
      return false;
    stack[r--] = u; // going down, store u
  }
  Node *w = newNode(x, pickHeight());
  while (h < w->height)
```

[1] This method does not exactly replicate the coin-tossing experiment since the value of `k` will always be less than the number of bits in an `int`. However, this will have negligible impact unless the number of elements in the structure is much greater than $2^{32} = 4294967296$.

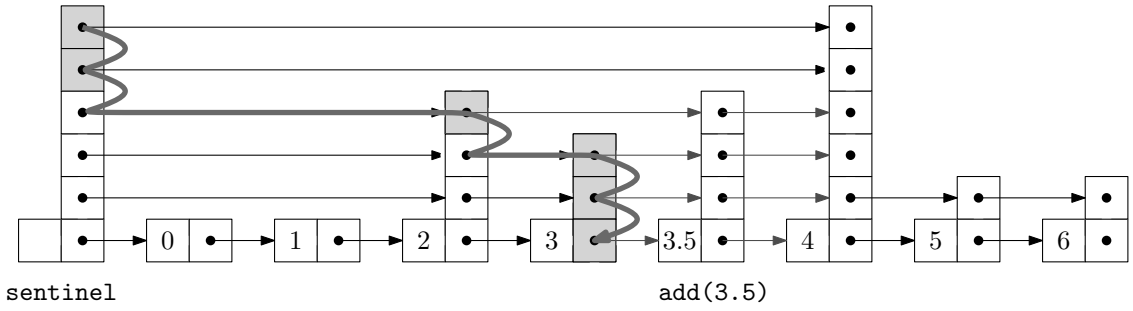


Figure 4.3: Adding the node containing 3.5 to a skiplist. The nodes stored in stack are highlighted.

```
    stack[++h] = sentinel; // increasing height of skiplist
  for (int i = 0; i < w->height; i++) {
    w->next[i] = stack[i]->next[i];
    stack[i]->next[i] = w;
  }
  n++;
  return true;
}
```

Removing an element, x, is done in a similar way, except that there is no need for stack to keep track of the search path. The removal can be done as we are following the search path. We search for x and each time the search moves downward from a node u, we check if u.next.x = x and if so, we splice u out of the list:

SkiplistSSet

```
bool remove(T x) {
  bool removed = false;
  Node *u = sentinel, *del;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u->next[r] != NULL && (comp = compare(u->next[r]->x, x)) < 0) {
      u = u->next[r];
    }
    if (u->next[r] != NULL && comp == 0) {
      removed = true;
      del = u->next[r];
      u->next[r] = u->next[r]->next[r];
      if (u == sentinel && u->next[r] == NULL)
        h--; // skiplist height has gone down
    }
```

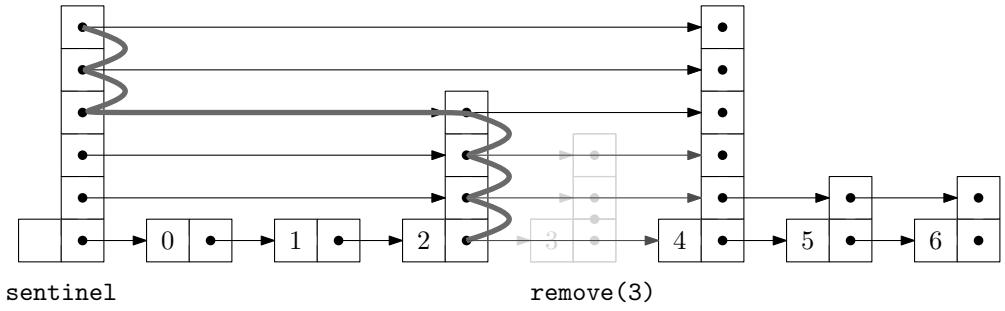


Figure 4.4: Removing the node containing 3 from a skiplist.

```
    r--;
  }
  if (removed) {
    delete del;
    n--;
  }
  return removed;
}
```

4.2.1 Summary

The following theorem summarizes the performance of skiplists when used to implement sorted sets:

Theorem 4.1. *A* `SkiplistSSet` *implements the* `SSet` *interface. A* `SkiplistSSet` *supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log n)$ *expected time per operation.*

4.3 `SkiplistList`: An Efficient Random-Access `List` Implementation

A `SkiplistList` implements the `List` interface on top of a skiplist structure. In a `SkiplistList`, L_0 contains the elements of the list in the order they appear in the list. Just like with a `SkiplistSSet`, elements can be added, removed, and accessed in $O(\log n)$ time.

For this to be possible, we need a way to follow the search path for the `i`th element in L_0. The easiest way to do this is to define the notion of the *length* of an edge in some list, L_r. We define the length of every edge in L_0 as 1. The length of an edge, `e`, in L_r, $r > 0$, is defined as the sum of the lengths of the edges below `e` in L_{r-1}. Equivalently, the length of `e` is the number of edges in L_0 below `e`. See Figure 4.5 for an example of a skiplist with the lengths of its edges shown. Since the edges of skiplists are stored in arrays, the lengths can be stored the same way:

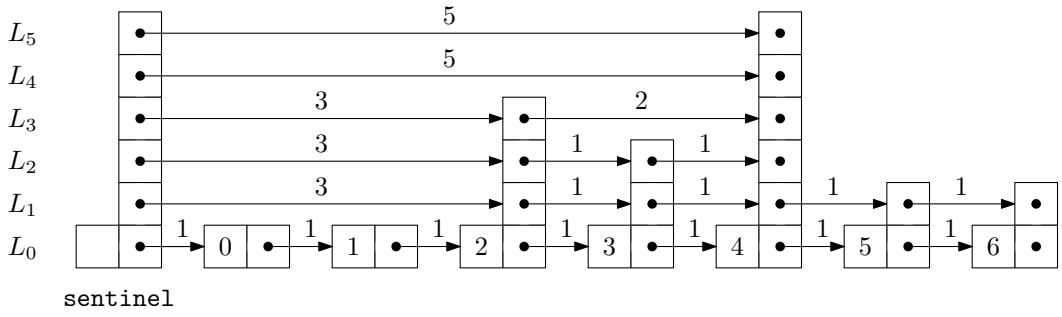


Figure 4.5: The lengths of the edges in a skiplist.

SkiplistList
```
struct Node {
  T x;
  int height;     // length of next
  int *length;
  Node **next;
};
```

The useful property of this definition of length is that, if we are currently at a node that is at position `j` in L_0 and we follow an edge of length ℓ, then we move to a node whose position, in L_0, is $\texttt{j} + \ell$. In this way, while following a search path, we can keep track of the position, `j`, of the current node in L_0. When at a node, `u`, in $L_{\texttt{r}}$, we go right if `j` plus the length of the edge `u.next[r]` is less than `i`, otherwise we go down into $L_{\texttt{r}-1}$.

SkiplistList
```
Node* findPred(int i) {
  Node *u = sentinel;
  int r = h;
  int j = -1;   // the index of the current node in list 0
  while (r >= 0) {
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    r--;
  }
  return u;
}
```

SkiplistList
```
T get(int i) {
  return findPred(i)->next[0]->x;
```

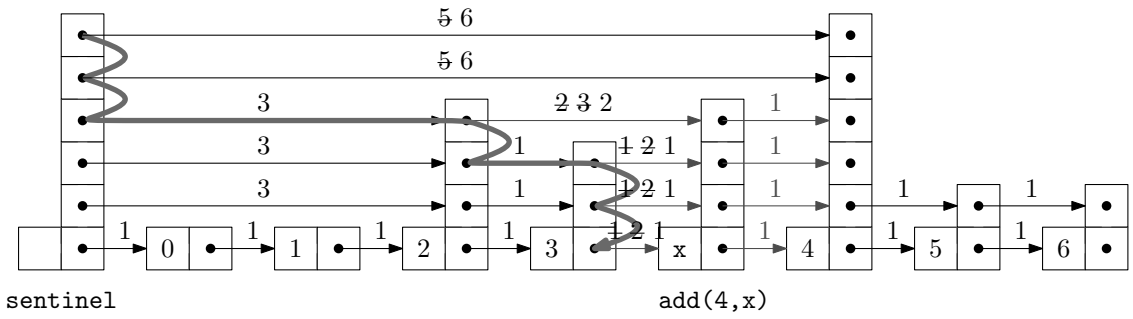


Figure 4.6: Adding an element to a SkiplistList.

```
}
T set(int i, T x) {
  Node *u = findPred(i)->next[0];
  T y = u->x;
  u->x = x;
  return y;
}
```

Since the hardest part of the operations get(i) and set(i,x) is finding the ith node in L_0, these operations run in $O(\log n)$ time.

Adding an element to a SkiplistList at a position, i, is fairly straightforward. Unlike in a SkiplistSSet, we are sure that a new node will actually be added, so we can do the addition at the same time as we search for the new node's location. We first pick the height, k, of the newly inserted node, w, and then follow the search path for i. Anytime the search path moves down from L_r with $r \leq k$, we splice w into L_r. The only extra care needed is to ensure that the lengths of edges are updated properly. See Figure 4.6.

Note that, each time the search path goes down at a node, u, in L_r, the length of the edge u.next[r] increases by one, since we are adding an element below that edge at position i. Splicing the node w between two nodes, u and z, works as shown in Figure 4.7. While following the search path we are already keeping track of the position, j, of u in L_0. Therefore, we know that the length of the edge from u to w is $i - j$. We can also deduce the length of the edge from w to z from the length, ℓ, of the edge from u to z. Therefore, we can splice in w and update the lengths of the edges in constant time.

This sounds more complicated than it actually is and the code is actually quite simple:

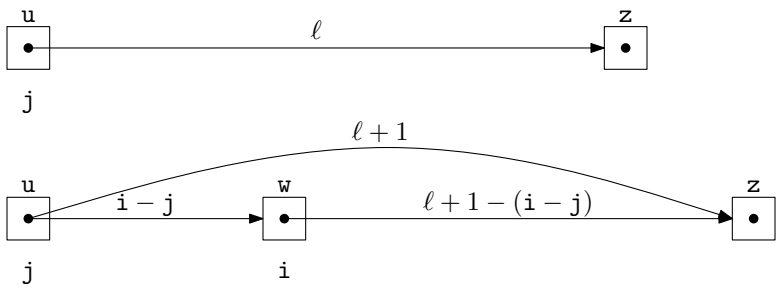


Figure 4.7: Updating the lengths of edges while splicing a node w into a skiplist.

SkiplistList

```
void add(int i, T x) {
  Node *w = newNode(x, pickHeight());
  if (w->height > h)
    h = w->height;
  add(i, w);
}
```

SkiplistList

```
Node* add(int i, Node *w) {
  Node *u = sentinel;
  int k = w->height;
  int r = h;
  int j = -1; // index of u
  while (r >= 0) {
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    u->length[r]++; // to account for new node in list 0
    if (r <= k) {
      w->next[r] = u->next[r];
      u->next[r] = w;
      w->length[r] = u->length[r] - (i - j);
      u->length[r] = i - j;
    }
    r--;
  }
  n++;
  return u;
}
```

By now, the implementation of the `remove(i)` operation in a `SkiplistList` should

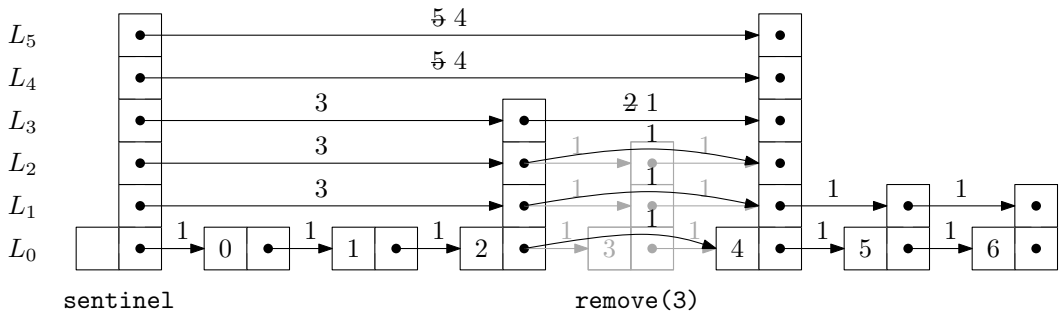


Figure 4.8: Removing an element from a SkiplistList.

be obvious. We follow the search path for the node at position i. Each time the search path takes a step down from a node, u, at level r we decrement the length of the edge leaving u at that level. We also check if u.next[r] is the element of rank i and, if so, splice it out of the list at that level. An example is shown in Figure 4.8.

SkiplistList

```
T remove(int i) {
  T x = NULL;
  Node *u = sentinel, *del;
  int r = h;
  int j = -1; // index of node u
  while (r >= 0) {
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    u->length[r]--; // for the node we are removing
    if (j + u->length[r] + 1 == i && u->next[r] != NULL) {
      x = u->next[r]->x;
      u->length[r] += u->next[r]->length[r];
      del = u->next[r];
      u->next[r] = u->next[r]->next[r];
      if (u == sentinel && u->next[r] == NULL)
        h--;
    }
    r--;
  }
  deleteNode(del);
  n--;
  return x;
}
```

4.3.1 Summary

The following theorem summarizes the performance of the `SkiplistList` data structure:

Theorem 4.2. *A* `SkiplistList` *implements the* `List` *interface. A* `SkiplistList` *supports the operations* `get(i)`, `set(i,x)`, `add(i,x)`, *and* `remove(i)` *in* $O(\log \mathtt{n})$ *expected time per operation.*

4.4 Analysis of Skiplists

In this section, we analyze the expected height, size, and length of the search path in a skiplist. This section requires a background in basic probability. Several proofs are based on the following basic observation about coin tosses.

Lemma 4.2. *Let T be the number of times a fair coin is tossed up to and including the first time the coin comes up heads. Then* $\mathrm{E}[T] = 2$.

Proof. Suppose we stop tossing the coin the first time it comes up heads. Define the indicator variable

$$I_i = \begin{cases} 0 & \text{if the coin is tossed less than } i \text{ times} \\ 1 & \text{if the coin is tossed } i \text{ or more times} \end{cases}$$

Note that $I_i = 1$ if and only if the first $i-1$ coin tosses are tails, so $\mathrm{E}[I_i] = \Pr\{I_i = 1\} = 1/2^{i-1}$. Observe that T, the total number of coin tosses, can be written as $T = \sum_{i=1}^{\infty} I_i$. Therefore,

$$\begin{aligned} \mathrm{E}[T] &= \mathrm{E}\left[\sum_{i=1}^{\infty} I_i\right] \\ &= \sum_{i=1}^{\infty} \mathrm{E}\left[I_i\right] \\ &= \sum_{i=1}^{\infty} 1/2^{i-1} \\ &= 1 + 1/2 + 1/4 + 1/8 + \cdots \\ &= 2 \ . \end{aligned}$$

□

The next two lemmata tell us that skiplists have linear size:

Lemma 4.3. *The expected number of nodes in a skiplist containing* `n` *elements, not including occurrences of the sentinel, is* $2\mathtt{n}$.

Proof. The probability that any particular element, $\mathtt{x}$, is included in list $L_{\mathtt{r}}$ is $1/2^{\mathtt{r}}$, so the expected number of nodes in $L_{\mathtt{r}}$ is $\mathtt{n}/2^{\mathtt{r}}$. Therefore, the total number of nodes in all lists is

$$\sum_{\mathtt{r}=0}^{\infty} \mathtt{n}/2^{\mathtt{r}} = \mathtt{n}(1 + 1/2 + 1/4 + 1/8 + \cdots) = 2\mathtt{n} \ . \qquad \square$$

Lemma 4.4. *The expected height of a skiplist containing* $\mathtt{n}$ *elements is at most* $\log \mathtt{n} + 2$.

Proof. For each $\mathtt{r} \in \{1, 2, 3, \ldots, \infty\}$, define the indicator random variable

$$I_{\mathtt{r}} = \begin{cases} 0 & \text{if } L_{\mathtt{r}} \text{ is empty} \\ 1 & \text{if } L_{\mathtt{r}} \text{ is non-empty} \end{cases}$$

The height $\mathtt{h}$ of the skiplist is then given by

$$\mathtt{h} = \sum_{i=1}^{\infty} I_{\mathtt{r}} \ .$$

Note that $I_{\mathtt{r}}$ is never more than the length, $|L_{\mathtt{r}}|$, of $L_{\mathtt{r}}$, so

$$\mathrm{E}[I_{\mathtt{r}}] \leq \mathrm{E}[|L_{\mathtt{r}}|] = \mathtt{n}/2^{\mathtt{r}} \ .$$

Therefore, we have

$$\begin{aligned} \mathrm{E}[\mathtt{h}] &= \mathrm{E}\left[\sum_{r=1}^{\infty} I_{\mathtt{r}}\right] \\ &= \sum_{\mathtt{r}=1}^{\infty} E[I_{\mathtt{r}}] \\ &= \sum_{\mathtt{r}=1}^{\lfloor \log \mathtt{n} \rfloor} E[I_{\mathtt{r}}] + \sum_{r=\lfloor \log \mathtt{n} \rfloor + 1}^{\infty} E[I_{\mathtt{r}}] \\ &\leq \sum_{\mathtt{r}=1}^{\lfloor \log \mathtt{n} \rfloor} 1 + \sum_{r=\lfloor \log \mathtt{n} \rfloor + 1}^{\infty} \mathtt{n}/2^{\mathtt{r}} \\ &\leq \log \mathtt{n} + \sum_{\mathtt{r}=0}^{\infty} 1/2^{\mathtt{r}} \\ &= \log \mathtt{n} + 2 \ . \end{aligned}$$

$\square$

Lemma 4.5. *The expected number of nodes in a skiplist containing* $\mathtt{n}$ *elements, including all occurrences of the sentinel, is* $2\mathtt{n} + O(\log \mathtt{n})$.

Proof. By Lemma 4.3, the expected number of nodes, not including the sentinel, is $2\mathtt{n}$. The number of occurrences of the sentinel is equal to the height, $\mathtt{h}$, of the skiplist so, by Lemma 4.4 the expected number of occurrences of the sentinel is at most $\log \mathtt{n} + 2 = O(\log \mathtt{n})$. □

Lemma 4.6. *The expected length of a search path in a skiplist is at most* $2 \log \mathtt{n} + O(1)$.

Proof. The easiest way to see this is to consider the *reverse search path* for a node, $\mathtt{x}$. This path starts at the predecessor of $\mathtt{x}$ in L_0. At any point in time, if the path can go up a level, then it does. If it cannot go up a level then it goes left. Observe that the reverse search path for $\mathtt{x}$ is identical to the search path for $\mathtt{x}$, except that it is reversed.

The number of nodes that the reverse search path visits at a particular level, $\mathtt{r}$, is related to the following experiment: Toss a coin. If the coin comes up heads then go up and stop, otherwise go left and repeat the experiment. The number of coin tosses before the heads then represents the number of steps to the left that a reverse search path takes at a particular level.[2] Lemma 4.2 tells us that the expected number of coin tosses before the first heads is 1.

Let $S_{\mathtt{r}}$ denote the number of steps the forward search path takes at level $\mathtt{r}$ that go to the right. We have just argued that $\mathrm{E}[S_{\mathtt{r}}] \le 1$. Furthermore, $S_{\mathtt{r}} \le |L_{\mathtt{r}}|$, since we can't take more steps in $L_{\mathtt{r}}$ than the length of $L_{\mathtt{r}}$, so

$$\mathrm{E}[S_{\mathtt{r}}] \le E[|L_{\mathtt{r}}|] = \mathtt{n}/2^{\mathtt{r}} \ .$$

We can now finish as in the proof of Lemma 4.4. Let S be the length of the search path for some node, $\mathtt{u}$, in a skiplist, and let $\mathtt{h}$ be the height of the skiplist. Then

$$\begin{aligned}
\mathrm{E}[S] &= \mathrm{E}\left[\mathtt{h} + \sum_{\mathtt{r}=0}^{\infty} S_{\mathtt{r}}\right] \\
&= \mathrm{E}[\mathtt{h}] + \sum_{\mathtt{r}=0}^{\infty} \mathrm{E}[S_{\mathtt{r}}] \\
&= \mathrm{E}[\mathtt{h}] + \sum_{\mathtt{r}=0}^{\lfloor \log \mathtt{n} \rfloor} \mathrm{E}[S_{\mathtt{r}}] + \sum_{\mathtt{r}=\lfloor \log \mathtt{n} \rfloor + 1}^{\infty} \mathrm{E}[S_{\mathtt{r}}] \\
&\le \mathrm{E}[\mathtt{h}] + \sum_{\mathtt{r}=0}^{\lfloor \log \mathtt{n} \rfloor} 1 + \sum_{r=\lfloor \log \mathtt{n} \rfloor + 1}^{\infty} \mathtt{n}/2^{\mathtt{r}}
\end{aligned}$$

[2]Note that this might overcount the number of steps to the left, since the experiment should end either at the first heads or when the search path reaches the sentinel, whichever comes first. This is not a problem since the lemma is only stating an upper bound.

$$\begin{aligned}
&\le \mathrm{E}[\mathtt{h}] + \sum_{\mathtt{r}=0}^{\lfloor \log \mathtt{n} \rfloor} 1 + \sum_{\mathtt{r}=0}^{\infty} 1/2^{\mathtt{r}} \\
&\le \mathrm{E}[\mathtt{h}] + \sum_{\mathtt{r}=0}^{\lfloor \log \mathtt{n} \rfloor} 1 + \sum_{\mathtt{r}=0}^{\infty} 1/2^{\mathtt{r}} \\
&\le \mathrm{E}[\mathtt{h}] + \log \mathtt{n} + 3 \\
&\le 2 \log \mathtt{n} + 5 \ . \qquad \square
\end{aligned}$$

The following theorem summarizes the results in this section:

Theorem 4.3. *A skiplist containing* $\mathtt{n}$ *elements has expected size* $O(\mathtt{n})$ *and the expected length of the search path for any particular element is at most* $2 \log \mathtt{n} + O(1)$.

4.5 Discussion and Exercises

Skiplists were introduced by Pugh [45] who also presented a number of applications of skiplists [44]. Since then they have been studied extensively. Several researchers have done very precise analysis of the expected length and variance in length of the search path for the `i`th element in a skiplist [35, 34, 42]. Deterministic versions [39], biased versions [6, 21], and self-adjusting versions [9] of skiplists have all been developed. Skiplist implementations have been written for various languages and frameworks and have seen use in open-source database systems [52, 46]. A variant of skiplists is used in the HP-UX operating system kernel's process management structures [33].

Exercise 4.1. Show that, during an `add(x)` or a `remove(x)` operation, the expected number of pointers in the structure that get changed is constant.

Exercise 4.2. Suppose that, instead of promoting an element from L_{i-1} into L_i based on a coin toss, we promote it with some probability p, $0 < p < 1$. Show that the expected length of the search path in this case is at most $(1/p) \log_{1/p} \mathtt{n} + O(1)$. What is the value of p that minimizes this expression? What is the expected height of the skiplist? What is the expected number of nodes in the skiplist?

Exercise 4.3. Design and implement a `find(x)` method for `SkiplistSSet` that avoids *locally-redundant comparisons*; these are comparisons that have already been done and occur because `u.next[r]` = `u.next[r - 1]`. Analyze the expected number of comparisons done by your modified `find(x)` method.

Exercise 4.4. Design and implement a version of a skiplist that implements the `SSet` interface, but also allows fast access to elements by rank. That is, it also supports the function

`get(i)`, which returns the element whose rank is `i` in $O(\log \mathtt{n})$ expected time. (The rank of an element `x` in an `SSet` is the number of elements in the `SSet` that are less than `x`.)

Exercise 4.5. Using the ideas from the space-efficient linked-list, `SEList`, design and implement a space-efficient `SSet`, `SESSet`. Do this by storing the data, in order, in an `SEList` and then storing the blocks of this `SEList` in an `SSet`. If the original `SSet` implementation uses $O(\mathtt{n})$ space to store `n` elements, then the `SESSet` will use enough space for `n` elements plus $O(\mathtt{n}/\mathtt{b} + \mathtt{b})$ wasted space.

Exercise 4.6. Using an `SSet` as your underlying structure, design and implement an application that reads a (large) text file and allow you to search, interactively, for any substring contained in the text.

Hint 1: Every substring is a prefix of some suffix, so it suffices to store all suffixes of the text file.

Hint 2: Any suffix can be represented compactly as a single integer indicating where the suffix begins in the text.

Test your application on some large texts like some of the books available at Project Gutenberg [1].

Chapter 5

Hash Tables

Hash tables are an efficient method of storing a small number, `n`, of integers from a large range $U = \{0, \ldots, 2^{\mathtt{w}} - 1\}$. The term *hash table* includes a broad range of data structures. This chapter focuses on one of the most common implementations of hash tables, namely hashing with chaining.

Very often hash tables store data that are not integers. In this case, an integer *hash code* is associated with each data item and this hash code is used in the hash table. The second part of this chapter discusses how such hash codes are generated.

Some of the methods used in this chapter require random choices of integers in some specific range. In the code samples, some of these "random" integers are hard-coded constants. These constants were obtained using random bits generated from atmospheric noise.

5.1 `ChainedHashTable`: Hashing with Chaining

A `ChainedHashTable` data structure uses *hashing with chaining* to store data as an array, `t`, of lists. An integer, `n`, keeps track of the total number of items in all lists:

`ChainedHashTable`

```
array<List> t;
int n;
```

The *hash value* of a data item `x`, denoted `hash(x)` is a value in the range $\{0, \ldots, \mathtt{t.length} - 1\}$. All items with hash value `i` are stored in the list at `t[i]`. To ensure that lists don't get too long, we maintain the invariant

$$\mathtt{n} \leq \mathtt{t.length}$$

so that the average number of elements stored in one of these lists is $\mathtt{n}/\mathtt{t.length} \leq 1$.

To add an element, x, to the hash table, we first check if the length of t needs to be increased and, if so, we grow t. With this out of the way we hash x to get an integer, i, in the range $\{0, \ldots, \texttt{t.length} - 1\}$ and we append x to the list t[i]:

ChainedHashTable
```
bool add(T x) {
  if (find(x) != null) return false;
  if (n+1 > t.length) resize();
  t[hash(x)].add(x);
  n++;
  return true;
}
```

Growing the table, if necessary, involves doubling the length of t and reinserting all elements into the new table. This is exactly the same strategy used in the implementation of ArrayStack and the same result applies: The cost of growing is only constant when amortized over a sequence of insertions (see Lemma 2.1 on page 20).

Besides growing, the only other work done when adding x to a ChainedHashTable involves appending x to the list t[hash(x)]. For any of the list implementations described in Chapters 2 or 3, this takes only constant time.

To remove an element x from the hash table we iterate over the list t[hash(x)] until we find x so that we can remove it:

ChainedHashTable
```
T remove(T x) {
  int j = hash(x);
  for (int i = 0; i < t[j].size(); i++) {
    T y = t[j].get(i);
    if (x == y) {
      t[j].remove(i);
      n--;
      return y;
    }
  }
  return null;
}
```

This takes $O(\texttt{n}_{\texttt{hash(x)}})$time, where $\texttt{n}_\texttt{i}$ denotes the length of the list stored at t[i].

Searching for the element x in a hash table is similar. We perform a linear search on the list t[hash(x)]:

ChainedHashTable
```
T find(T x) {
  int j = hash(x);
```

```
    for (int i = 0; i < t[j].size(); i++)
      if (x == t[j].get(i))
        return t[j].get(i);
    return null;
  }
```

Again, this takes time proportional to the length of the list $\texttt{t}[\texttt{hash}(\texttt{x})]$.

The performance of a hash table depends critically on the choice of the hash function. A good hash function will spread the elements evenly among the `t.length` lists, so that the expected size of the list $\texttt{t}[\texttt{hash}(\texttt{x})]$ is $O(\texttt{n}/\texttt{t.length}) = O(1)$. On the other hand, a bad hash function will hash all values (including `x`) to the same table location, in which case the size of the list $\texttt{t}[\texttt{hash}(\texttt{x})]$ will be `n`. In the next section we describe a good hash function.

5.1.1 Multiplicative Hashing

Multiplicative hashing is an efficient method of generating hash values based on modular arithmetic (discussed in Section 2.3) and integer division. It uses the div operator, which calculates the integral part of a quotient, while discarding the remainder. Formally, for any integers $a \geq 0$ and $b \geq 1$, $a \operatorname{div} b = \lfloor a/b \rfloor$.

In multiplicative hashing, we use a hash table of size $2^{\texttt{d}}$ for some integer `d` (called the *dimension*). The formula for hashing an integer $\texttt{x} \in \{0, \ldots, 2^{\texttt{w}} - 1\}$ is

$$\texttt{hash}(\texttt{x}) = ((\texttt{z} \cdot \texttt{x}) \bmod 2^{\texttt{w}}) \operatorname{div} 2^{\texttt{w}-\texttt{d}} \ .$$

Here, `z` is a randomly chosen *odd* integer in $\{1, \ldots, 2^{\texttt{w}} - 1\}$. This hash function can be realized very efficiently by observing that, by default, operations on integers are already done modulo $2^{\texttt{w}}$ where `w` is the number of bits in an integer. (See Figure 5.1.) Furthermore, integer division by $2^{\texttt{w}-\texttt{d}}$ is equivalent to dropping the rightmost $\texttt{w} - \texttt{d}$ bits in a binary representation (which is implemented by shifting the bits right by $\texttt{w} - \texttt{d}$). In this way, the code that implements the above formula is simpler than the formula itself:

ChainedHashTable

```
int hash(T x) {
  return ((unsigned)(z * hashCode(x))) >> (w-d);
}
```

The following lemma, whose proof is deferred until later in this section, shows that multiplicative hashing does a good job of avoiding collisions:

Lemma 5.1. *Let* `x` *and* `y` *be any two values in* $\{0, \ldots, 2^{\texttt{w}}-1\}$ *with* $\texttt{x} \neq \texttt{y}$. *Then* $\Pr\{\texttt{hash}(\texttt{x}) = \texttt{hash}(\texttt{y})\} \leq 2/2^{\texttt{d}}$.

$2^{\mathtt{w}}$ (4294967296)	10000000000000000000000000000000
z (4102541685)	11110100100001111101000101110101
x (42)	00000000000000000000000000101010
$\mathtt{z} \cdot \mathtt{x}$	10100000011110010010000101110100110010
$(\mathtt{z} \cdot \mathtt{x}) \bmod 2^{\mathtt{w}}$	00011110010010000101110100110010
$((\mathtt{z} \cdot \mathtt{x}) \bmod 2^{\mathtt{w}}) \operatorname{div} 2^{\mathtt{w}-\mathtt{d}}$	00011110

Figure 5.1: The operation of the multiplicative hash function with $\mathtt{w} = 32$ and $\mathtt{d} = 8$.

With Lemma 5.1, the performance of `remove`(x), and `find`(x) are easy to analyze:

Lemma 5.2. *For any data value* x*, the expected length of the list* t[hash(x)] *is at most* $\mathtt{n}_\mathtt{x} + 2$*, where* $\mathtt{n}_\mathtt{x}$ *is the number of occurrences of* x *in the hash table.*

Proof. Let S be the (multi-)set of elements stored in the hash table that are not equal to x. For an element $\mathtt{y} \in S$, define the indicator variable

$$I_\mathtt{y} = \begin{cases} 1 & \text{if } \mathtt{hash}(\mathtt{x}) = \mathtt{hash}(\mathtt{y}) \\ 0 & \text{otherwise} \end{cases}$$

and notice that, by Lemma 5.1, $\mathrm{E}[I_\mathtt{y}] \le 2/2^{\mathtt{d}} = 2/\mathtt{t.length}$. The expected length of the list t[hash(x)] is given by

$$\begin{aligned} \mathrm{E}\left[\mathtt{t}[\mathtt{hash}(\mathtt{x})].\mathtt{size}()\right] &= \mathrm{E}\left[\mathtt{n}_\mathtt{x} + \sum_{\mathtt{y} \in S} I_\mathtt{y}\right] \\ &= \mathtt{n}_\mathtt{x} + \sum_{\mathtt{y} \in S} \mathrm{E}[I_\mathtt{y}] \\ &\le \mathtt{n}_\mathtt{x} + \sum_{\mathtt{y} \in S} 2/\mathtt{t.length} \\ &\le \mathtt{n}_\mathtt{x} + \sum_{\mathtt{y} \in S} 2/\mathtt{n} \\ &\le \mathtt{n}_\mathtt{x} + (\mathtt{n} - \mathtt{n}_\mathtt{x})2/\mathtt{n} \\ &\le \mathtt{n}_\mathtt{x} + 2 \ , \end{aligned}$$

as required. □

Now, we want to prove Lemma 5.1, but first we need a result from number theory. In the following proof, we use the notation $(b_r, \ldots, b_0)_2$ to denote $\sum_{i=0}^{r} b_i 2^i$, where each b_i

is a bit either 0 or 1, i.e., the integer whose binary representation is given by $b_r, \ldots, b_0$. We use $\star$ to denote a bit of unknown value.

Lemma 5.3. *Let S be the set of odd integers in $\{1, \ldots, 2^{\mathtt{w}} - 1\}$, Let q and i be any two elements in S. Then there is exactly one value $\mathtt{z} \in S$ such that $\mathtt{z}q \bmod 2^{\mathtt{w}} = i$.*

Proof. Since the number of choices for $\mathtt{z}$ and i is the same, it is sufficient to prove that there is *at most* one value $\mathtt{z} \in S$ that satisfies $\mathtt{z}q \bmod 2^{\mathtt{w}} = i$.

Suppose, for the sake of contradiction, that there are two such values $\mathtt{z}$ and $\mathtt{z}'$, with $\mathtt{z} > \mathtt{z}'$. Then

$$\mathtt{z}q \bmod 2^{\mathtt{w}} = \mathtt{z}'q \bmod 2^{\mathtt{w}} = i$$

So

$$(\mathtt{z} - \mathtt{z}')q \bmod 2^{\mathtt{w}} = 0$$

But this means that

$$(\mathtt{z} - \mathtt{z}')q = k2^{\mathtt{w}} \tag{5.1}$$

for some integer k. Thinking in terms of binary numbers, we have

$$(\mathtt{z} - \mathtt{z}')q = k \cdot (1, \underbrace{0, \ldots, 0}_{\mathtt{w}})_2 \ ,$$

so that the $\mathtt{w}$ trailing bits in the binary representation of $(\mathtt{z} - \mathtt{z}')q$ are all 0's.

Furthermore $k \neq 0$ since $q \neq 0$ and $\mathtt{z} - \mathtt{z}' \neq 0$. Since q is odd, it has no trailing 0's in its binary representation:

$$q = (\star, \ldots, \star, 1)_2 \ .$$

Since $|\mathtt{z} - \mathtt{z}'| < 2^{\mathtt{w}}$, $\mathtt{z} - \mathtt{z}'$ has fewer than $\mathtt{w}$ trailing 0's in its binary representation:

$$\mathtt{z} - \mathtt{z}' = (\star, \ldots, \star, 1, \underbrace{0, \ldots, 0}_{<\mathtt{w}})_2 \ .$$

Therefore, the product $(\mathtt{z} - \mathtt{z}')q$ has fewer than $\mathtt{w}$ trailing 0's in its binary representation:

$$(\mathtt{z} - \mathtt{z}')q = (\star, \cdots, \star, 1, \underbrace{0, \ldots, 0}_{<\mathtt{w}})_2 \ .$$

Therefore $(\mathtt{z} - \mathtt{z}')q$ cannot satisfy (5.1), yielding a contradiction and completing the proof. □

The utility of Lemma 5.3 comes from the following observation: If z is chosen uniformly at random from S, then zt is uniformly distributed over S. In the following proof, it helps to think of the binary representation of z, which consists of $w-1$ random bits followed by a 1.

Proof of Lemma 5.1. First we note that the condition hash(x) = hash(y) is equivalent to the statement "the highest-order d bits of zx mod 2^w and the highest-order d bits of zy mod 2^w are the same." A necessary condition of that statement is that the highest-order d bits in the binary representation of z(x − y) mod 2^w are either all 0's or all 1's. That is,

$$z(x-y) \bmod 2^w = (\underbrace{0,\ldots,0}_{d},\underbrace{\star,\ldots,\star}_{w-d})_2 \tag{5.2}$$

when zx mod 2^w > zy mod 2^w or

$$z(x-y) \bmod 2^w = (\underbrace{1,\ldots,1}_{d},\underbrace{*,\ldots,*}_{w-d})_2 \ . \tag{5.3}$$

when zx mod 2^w < zy mod 2^w. Therefore, we only have to bound the probability that z(x − y) mod 2^w looks like (5.2) or (5.3).

Let q be the unique odd integer such that (x − y) mod $2^w = q2^r$ for some integer $r \geq 0$. By Lemma 5.3, the binary representation of zq mod 2^w has $w-1$ random bits, followed by a 1:

$$zq \bmod 2^w = (\underbrace{b_{w-1},\ldots,b_1}_{w-1},1)_2$$

Therefore, the binary representation of z(x−y) mod $2^w = zq2^r$ mod 2^w has $w-r-1$ random bits, followed by a 1, followed by r 0's:

$$z(x-y) \bmod 2^w = zq2^r \bmod 2^w = (\underbrace{b_{w-r-1},\ldots,b_1}_{w-r-1},1,\underbrace{0,0,\ldots,0}_{r})_2$$

We can now finish the proof: If $r > w-d$, then the d higher order bits of z(x − y) mod 2^w contain both 0's and 1's, so the probability that z(x − y) mod 2^w looks like (5.2) or (5.3) is 0. If $r = w-d$, then the probability of looking like (5.2) is 0, but the probability of looking like (5.3) is $1/2^{d-1} = 2/2^d$ (since we must have $b_1,\ldots,b_{d-1} = 1,\ldots,1$). If $r < w-d$ then we must have $b_{w-r-1},\ldots,b_{w-r-d} = 0,\ldots,0$ or $b_{w-r-1},\ldots,b_{w-r-d} = 1,\ldots,1$. The probability of each of these cases is $1/2^d$ and they are mutually exclusive, so the probability of either of these cases is $2/2^d$. This completes the proof. □

5.1.2 Summary

The following theorem summarizes the performance of the ChainedHashTable data structure:

Theorem 5.1. *A* ChainedHashTable *implements the* USet *interface. Ignoring the cost of calls to* grow(), *a* ChainedHashTable *supports the operations* add(x), remove(x), *and* find(x) *in* $O(1)$ *expected time per operation.*

Furthermore, beginning with an empty ChainedHashTable, *any sequence of* m add(x) *and* remove(x) *operations results in a total of* $O(m)$ *time spent during all calls to* grow().

5.2 LinearHashTable: Linear Probing

The ChainedHashTable data structure uses an array of lists, where the ith list stores all elements x such that hash(x) = i. An alternative, called *open addressing* is to store the elements directly in an array, t, with each array location in t storing at most one value. This is the approach taken by the LinearHashTable described in this section. In some places, this data structure is described as *open addressing with linear probing*.

The main idea behind a LinearHashTable is that we would, ideally, like to store the element x with hash value i = hash(x) in the table location t[i]. If we can't do this (because some element is already stored there) then we try to store it at location t[(i + 1) mod t.length]; if that's not possible, then we try t[(i + 2) mod t.length], and so on, until we find a place for x.

There are three types of entries stored in t:

1. data values: actual values in the USet that we are representing;
2. null values: at array locations where no data has ever been stored; and
3. del values: at array locations where data was once stored but that has since been deleted.

In addition to the counter, n, that keeps track of the number of elements in the LinearHashTable, a counter, q, keeps track of the number of elements of Types 1 and 3. That is, q is equal to n plus the number of del values in t. To make this work efficiently, we need t to be considerably larger than q, so that there are lots of null values in t. The operations on a LinearHashTable therefore maintain the invariant that t.length ≥ 2q.

To summarize, a LinearHashTable contains an array, t, that stores data elements, and integers n and q that keep track of the number of data elements and non-null values

of t, respectively. Because many hash functions only work for table sizes that are a power of 2, we also keep an integer d and maintain the invariant that t.length = 2^d.

LinearHashTable
```
array<T> t;
int n;   // number of values in T
int q;   // number of non-null entries in T
int d;   // t.length = 2^d
```

The find(x) operation in a LinearHashTable is simple. We start at array entry t[i] where i = hash(x) and search entries t[i], t[(i+1) mod t.length], t[(i+2) mod t.length], and so on, until we find an index i′ such that, either, t[i′] = x, or t[i′] = null. In the former case we return t[i′]. In the latter case, we conclude that x is not contained in the hash table and return null.

LinearHashTable
```
T find(T x) {
  int i = hash(x);
  while (t[i] != null) {
    if (t[i] != del && t[i] == x) return t[i];
    i = (i == t.length-1) ? 0 : i + 1; // increment i (mod t.length)
  }
  return null;
}
```

The add(x) operation is also fairly easy to implement. After checking that x is not already stored in the table (using find(x)), we search t[i], t[(i + 1) mod t.length], t[(i+2) mod t.length], and so on, until we find a null or del and store x at that location, increment n, and q, if appropriate.:

LinearHashTable
```
bool add(T x) {
  if (find(x) != null) return false;
  if (2*(q+1) > t.length) resize();   // max 50% occupancy
  int i = hash(x);
  while (t[i] != null && t[i] != del)
    i = (i == t.length-1) ? 0 : i + 1; // increment i (mod t.length)
  if (t[i] == null) q++;
  n++;
  t[i] = x;
  return true;
}
```

By now, the implementation of the remove(x) operation should be obvious. we search t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], and so on until we find an

index i′ such that t[i′] = x or t[i′] = null. In the former case, we set t[i′] = del and return true. In the latter case we conclude that x was not stored in the table (and therefore cannot be deleted) and return false.

LinearHashTable

```
T remove(T x) {
  int i = hash(x);
  while (t[i] != null) {
    T y = t[i];
    if (y != del && x == y) {
      t[i] = del;
      n--;
      if (8*n < t.length) resize(); // min 12.5% occupancy
      return y;
    }
    i = (i == t.length-1) ? 0 : i + 1;  // increment i (mod t.length)
  }
  return null;
}
```

The correctness of the find(x), add(x), and remove(x) methods is easy to verify, though it relies on the use of del values. Notice that none of these operations ever sets a non-null entry to null. Therefore, when we reach an index i′ such that t[i′] = null, this is a proof that the element, x, that we are searching for is not stored in the table; t[i′] has always been null, so there is no reason that a previous add(x) operation would have proceeded beyond index i′.

The resize() method is called by add(x) when the number of non-null entries exceeds n/2 or by remove(x) when the number of data entries is less than t.length/8. The resize() method works like the resize() methods in other array-based data structures. We find the smallest non-negative integer d such that $2^d \geq 3n$. We reallocate the array t so that it has size 2^d and then we insert all the elements in the old version of t into the newly-resized copy of t. While doing this we reset q equal to n since the newly-allocated t has no del values.

LinearHashTable

```
void resize() {
  d = 1;
  while ((1<<d) < 3*n) d++;
  array<T> tnew(1<<d, null);
  q = n;
  // insert everything in told
  for (int k = 0; k < t.length; k++) {
    if (t[k] != null && t[k] != del) {
```

```
        int i = hash(t[k]);
        while (tnew[i] != null)
          i = (i == tnew.length-1) ? 0 : i + 1;
        tnew[i] = t[k];
      }
    }
    t = tnew;
  }
```

5.2.1 Analysis of Linear Probing

Notice that each operation, `add(x)`, `remove(x)`, or `find(x)`, finishes as soon as (or before) it discovers the first `null` entry in `t`. The intuition behind the analysis of linear probing is that, since at least half the elements in `t` are equal to `null`, an operation should not take long to complete because it will very quickly come across a `null` entry. We shouldn't rely too heavily on this intuition though, because it would lead us to (the incorrect) conclusion that the expected number of locations in `t` examined by an operation is at most 2.

For the rest of this section, we will assume that all hash values are independently and uniformly distributed in $\{0, \ldots, \texttt{t.length}-1\}$. This is not a realistic assumption, but it will make it possible for us to analyze linear probing. Later in this section we will describe a method, called tabulation hashing, that produces a hash function that is "good enough" for linear probing. We will also assume that all indices into the positions of `t` are taken modulo `t.length`, so that `t[i]` is really a shorthand for `t[i mod t.length]`.

A *run of length k that starts at* `i` occurs when $\texttt{t}[\texttt{i}], \texttt{t}[\texttt{i}+1], \ldots, \texttt{t}[\texttt{i}+k-1]$ are all non-`null` and $\texttt{t}[\texttt{i}-1] = \texttt{t}[\texttt{i}+k] = \texttt{null}$. The number of non-`null` elements of `t` is exactly `q` and the `add(x)` method ensures that, at all times, $\texttt{q} \le \texttt{t.length}/2$. There are `q` elements $\texttt{x}_1, \ldots, \texttt{x}_\texttt{q}$ that have been inserted into `t` since the last `rebuild()` operation. By our assumption, each of these has a hash value, $\texttt{hash}(\texttt{x}_j)$, that is uniform and independent of the rest. With this setup, we can prove the main lemma required to analyze linear probing.

Lemma 5.4. *For any* $\texttt{i} \in \{0, \ldots, \texttt{t.length}-1\}$*, the probability that a run of length k starts at* `i` *is $O(c^k)$ for some constant $0 < c < 1$.*

Proof. If a run of length k starts at `i`, then there are exactly k elements $\texttt{x}_j$ such that $\texttt{hash}(\texttt{x}_j) \in \{\texttt{i}, \ldots, \texttt{i}+k-1\}$. The probability that this occurs is exactly

$$p_k = \binom{\texttt{q}}{k} \left(\frac{k}{\texttt{t.length}}\right)^k \left(\frac{\texttt{t.length}-k}{\texttt{t.length}}\right)^{\texttt{q}-k} ,$$

since, for each choice of k elements, these k elements must hash to one of the k locations and the remaining $\mathtt{q} - k$ elements must hash to the other $\mathtt{t.length} - k$ table locations.[1]

In the following derivation we will cheat a little and replace $r!$ with $(r/e)^r$. Stirling's Approximation (Section 1.2.2) shows that this is only a factor of $O(\sqrt{r})$ from the truth. This is just done to make the derivation simpler; Exercise 5.2 asks the reader to redo the calculation more rigorously using Stirling's Approximation in its entirety.

The value of p_k is maximized when $\mathtt{t.length}$ is minimum, and the data structure maintains the invariant that $\mathtt{t.length} \geq 2\mathtt{q}$, so

$$\begin{aligned}
p_k &\leq \binom{\mathtt{q}}{k}\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} \\
&= \left(\frac{\mathtt{q}!}{(\mathtt{q}-k)!k!}\right)\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} \\
&\approx \left(\frac{\mathtt{q}^{\mathtt{q}}}{(\mathtt{q}-k)^{\mathtt{q}-k}k^k}\right)\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} && \text{[Stirling's approximation]} \\
&= \left(\frac{\mathtt{q}^k\mathtt{q}^{\mathtt{q}-k}}{(\mathtt{q}-k)^{\mathtt{q}-k}k^k}\right)\left(\frac{k}{2\mathtt{q}}\right)^k\left(\frac{2\mathtt{q}-k}{2\mathtt{q}}\right)^{\mathtt{q}-k} \\
&= \left(\frac{\mathtt{q}k}{2\mathtt{q}k}\right)^k\left(\frac{\mathtt{q}(2\mathtt{q}-k)}{2\mathtt{q}(\mathtt{q}-k)}\right)^{\mathtt{q}-k} \\
&= \left(\frac{1}{2}\right)^k\left(\frac{(2\mathtt{q}-k)}{2(\mathtt{q}-k)}\right)^{\mathtt{q}-k} \\
&= \left(\frac{1}{2}\right)^k\left(1+\frac{k}{2(\mathtt{q}-k)}\right)^{\mathtt{q}-k} \\
&= \left(\frac{\sqrt{e}}{2}\right)^k .
\end{aligned}$$

(In the last step, we use the inequality $(1 + 1/x)^x \leq e$, which holds for all $x > 0$.) Since $\sqrt{e}/2 < 0.824360636 < 1$, this completes the proof. □

Using Lemma 5.4 to prove upper-bounds on the expected running time of `find(x)`, `add(x)`, and `remove(x)` is now fairly straight-forward. Consider the simplest case, where we execute `find(x)` for some value `x` that has never been stored in the `LinearHashTable`. In this case, $\mathtt{i} = \mathtt{hash(x)}$ is a random value in $\{0, \ldots, \mathtt{t.length} - 1\}$ independent of the contents of `t`. If `i` is part of a run of length k then the time it takes to execute the `find(x)`

[1] Note that p_k is greater than the probability that a run of length k starts at `i`, since the definition of p_k does not include the requirement $\mathtt{t}[\mathtt{i}-1] = \mathtt{t}[\mathtt{i}+k] = \mathtt{null}$.

operation is at most $O(1+k)$. Thus, the expected running time can be upper-bounded by

$$O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k\Pr\{\texttt{i} \text{ is part of a run of length } k\}\right) .$$

Note that each run of length k contributes to the inner sum k times for a total contribution of k^2, so the above sum can be rewritten as

$$\begin{aligned}
&O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k^2\Pr\{\texttt{i} \text{ starts a run of length } k\}\right)\\
&\le O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty} k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty} k^2\cdot O(c^k)\right)\\
&= O(1) .
\end{aligned}$$

The last step in this derivation comes from the fact that $\sum_{k=0}^{\infty} k^2\cdot O(c^k)$ is an exponentially decreasing series.[2] Therefore, we conclude that the expected running time of the `find(x)` operation for a value `x` that is not contained in a `LinearHashTable` is $O(1)$.

If we ignore the cost of the `resize()` operation, the above analysis gives us all we need to analyze the cost of operations on a `LinearHashTable`.

First of all, the analysis of `find(x)` given above applies to the `add(x)` operation when `x` is not contained in the table. To analyze the `find(x)` operation when `x` is contained in the table, we need only note that this is the same as the cost of the `add(x)` operation that previously added `x` to the table. Finally, the cost of a `remove(x)` operation is the same as the cost of a `find(x)` operation.

In summary, if we ignore the cost of calls to `resize()`, all operations on a `LinearHashTable` run in $O(1)$ expected time. Accounting for the cost of resize can be done using the same type of amortized analysis performed for the `ArrayStack` data structure in Section 2.1.

5.2.2 Summary

The following theorem summarizes the performance of the `LinearHashTable` data structure:

[2] In the terminology of many calculus texts, this sum passes the ratio test: There exists a positive integer k_0 such that, for all $k \ge k_0$, $\frac{(k+1)^2 c^{k+1}}{k^2 c^k} < 1$.

Theorem 5.2. *A* `LinearHashTable` *implements the* `USet` *interface. Ignoring the cost of calls to* `resize()`*, a* `LinearHashTable` *supports the operations* `add(x)`*,* `remove(x)`*, and* `find(x)` *in* $O(1)$ *expected time per operation.*

Furthermore, beginning with an empty `LinearHashTable`*, any sequence of* m `add(x)` *and* `remove(x)` *operations results in a total of* $O(m)$ *time spent during all calls to* `resize()`*.*

5.2.3 Tabulation Hashing

While analyzing the `LinearHashTable` structure, we made a very strong assumption: That for any set of elements, $\{x_1, \ldots, x_n\}$, the hash values $\texttt{hash}(x_1), \ldots, \texttt{hash}(x_n)$ are independently and uniformly distributed over $\{0, \ldots, \texttt{t.length} - 1\}$. One way to imagine getting this is to have a giant array, called `tab` of length 2^w, where each entry is a random `w`-bit integer, independent of all the other entries. In this way, we could implement `hash(x)` by extracting a `d`-bit integer from `tab[x.hashCode()]`:

LinearHashTable
```
int idealHash(T x) {
  return tab[hashCode(x) >> w-d];
}
```

Unfortunately, storing an array of size 2^w is prohibitive in terms of memory usage. The approach used by *tabulation hashing* is to, instead, store `w/r` arrays each of length 2^r. All the entries in these arrays are indepent `w`-bit integers. To obtain the value of `hash(x)` we split `x.hashCode()` up into `w/r` `r`-bit integers and use these as indices into these arrays. We then combine all these values with the bitwise exclusive-or operator to obtain `hash(x)`. The following code shows how this works when `w` = 32 and `r` = 4:

LinearHashTable
```
int hash(T x) {
  unsigned h = hashCode(x);
  return (tab[0][h&0xff]
       ^ tab[1][(h>>8)&0xff]
       ^ tab[2][(h>>16)&0xff]
       ^ tab[3][(h>>24)&0xff])
      >> (w-d);
}
```

In this case, `tab` is a 2-dimensional array with 4 columns and $2^{32/4} = 256$ rows.

One can easily verify that, for any `x`, `hash(x)` is uniformly distributed over $\{0, \ldots, 2^d - 1\}$. With a little work, one can even verify that any pair of values have independent hash

values. This implies tabulation hashing could be used in place of multiplicative hashing for the `ChainedHashTable` implementation.

However, it is not true that any set of `n` distinct values gives a set of `n` independent hash values. Nevertheless, when tabulation hashing is used, the bound of Theorem 5.2 still holds. References for this are provided at the end of this chapter.

5.3 Hash Codes

The hash tables discussed in the previous section are used to associate data with integer keys consisting of `w` bits. In many cases, we have keys that are not integers. They may be strings, objects, arrays, or other compound structures. To use hash tables for these types of data, we must map these data types to `w`-bit hash codes. Hash code mappings should have the following properties:

1. If `x` and `y` are equal, then `x.hashCode()` and `y.hashCode()` are equal.

2. If `x` and `y` are not equal, then the probability that `x.hashCode()` = `y.hashCode()` should be small (close to $1/2^w$).

The first property ensures that if we store `x` in a hash table and later look up a value `y` equal to `x`, then we will find `x`—as we should. The second property minimizes the loss from converting our objects to integers. It ensures that unequal objects usually have different hash codes and so are likely to be stored at different locations in our hash table.

5.3.1 Hash Codes for Primitive Data Types

Small primitive data types like `char`, `byte`, `int`, and `float` are usually easy to find hash codes for. These data types always have a binary representation and this binary representation usually consists of `w` or fewer bits. (For example, in C++ `char` is typically an 8-bit type and `float` is a 32-bit type.) In these cases, we just treat these bits as the representation of an integer in the range $\{0, \ldots, 2^w - 1\}$. If two values are different, they get different hash codes. If they are the same, they get the same hash code.

A few primitive data types are made up of more than `w` bits, usually cw bits for some constant integer c. (Java's `long` and `double` types are examples of this with $c = 2$.) These data types can be treated as compound objects made of c parts, as described in the next section.

5.3.2 Hash Codes for Compound Objects

For a compound object, we want to create a hash code by combining the individual hash codes of the object's constituent parts. This is not as easy as it sounds. Although one can find many hacks for this (for example, combining the hash codes with bitwise exclusive-or operations), many of these hacks turn out to be easy to foil (see Exercises 5.4–5.6). However, if one is willing to do arithmetic with $2\mathtt{w}$ bits of precision, then there are simple and robust methods available. Suppose we have an object made up of several parts $P_0, \ldots, P_{r-1}$ whose hash codes are $\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}$. Then we can choose mutually independent random $\mathtt{w}$-bit integers $\mathtt{z}_0, \ldots, \mathtt{z}_{r-1}$ and a random $2\mathtt{w}$-bit odd integer $\mathtt{z}$ and compute a hash code for our object with

$$h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = \left(\left(\mathtt{z} \sum_{i=0}^{r-1} \mathtt{z}_i \mathtt{x}_i \right) \bmod 2^{2\mathtt{w}} \right) \operatorname{div} 2^{\mathtt{w}} \ .$$

Note that this hash code has a final step (multiplying by $\mathtt{z}$ and dividing by $2^{\mathtt{w}}$) that uses the multiplicative hash function from Section 5.1.1 to take the $2\mathtt{w}$-bit intermediate result and reduce it to a $\mathtt{w}$-bit final result. Here is an example of this method applied to a simple compound object with 3 parts `x0`, `x1`, and `x2`:

Point3D

```
unsigned hashCode() {
  long long z[] = {0x2058cc50L, 0xcb19137eL, 0x2cb6b6fdL}; // random
  long zz = 0xbea0107e5067d19dL;                           // random
  long h0 = ods::hashCode(x0);
  long h1 = ods::hashCode(x1);
  long h2 = ods::hashCode(x2);
  return (int)(((z[0]*h0 + z[1]*h1 + z[2]*h2)*zz) >> 32);
}
```

The following theorem shows that, in addition to being straightforward to implement, this method is provably good:

Theorem 5.3. *Let* $\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}$ *and* $\mathtt{y}_0, \ldots, \mathtt{y}_{r-1}$ *each be sequences of* $\mathtt{w}$ *bit integers in* $\{0, \ldots, 2^{\mathtt{w}} - 1\}$ *and assume* $\mathtt{x}_i \neq \mathtt{y}_i$ *for at least one index* $i \in \{0, \ldots, r-1\}$*. Then*

$$\Pr\{h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = h(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1})\} \leq 3/2^{\mathtt{w}} \ .$$

Proof. We will first ignore the final multiplicative hashing step and see how that step contributes later. Define:

$$h'(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = \left(\sum_{j=0}^{r-1} \mathtt{z}_j \mathtt{x}_j \right) \bmod 2^{2\mathtt{w}} \ .$$

Suppose that $h'(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = h'(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1})$. We can rewrite this as:

$$\mathtt{z}_i(\mathtt{x}_i - \mathtt{y}_i) \bmod 2^{2\mathtt{w}} = t \tag{5.4}$$

where

$$t = \left(\sum_{j=0}^{i-1} \mathtt{z}_j(\mathtt{y}_j - \mathtt{x}_j) + \sum_{j=i+1}^{r-1} \mathtt{z}_j(\mathtt{y}_j - \mathtt{x}_j) \right) \bmod 2^{2\mathtt{w}}$$

If we assume, without loss of generality that $\mathtt{x}_i > \mathtt{y}_i$, then (5.4) becomes

$$\mathtt{z}_i(\mathtt{x}_i - \mathtt{y}_i) = t \ , \tag{5.5}$$

since each of $\mathtt{z}_i$ and $(\mathtt{x}_i - \mathtt{y}_i)$ is at most $2^{\mathtt{w}} - 1$, so their product is at most $2^{2\mathtt{w}} - 2^{\mathtt{w}+1} + 1 < 2^{2\mathtt{w}} - 1$. By assumption, $\mathtt{x}_i - \mathtt{y}_i \neq 0$, so (5.5) has at most one solution in $\mathtt{z}_i$. Therefore, since $\mathtt{z}_i$ and t are independent ($\mathtt{z}_0, \ldots, \mathtt{z}_{r-1}$ mutually independent), the probability that we select $\mathtt{z}_i$ so that $h'(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = h'(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1})$ is at most $1/2^{\mathtt{w}}$.

The final step of the hash function is to apply multiplicative hashing to reduce our $2\mathtt{w}$-bit intermediate result $h'(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1})$ to a $\mathtt{w}$-bit final result $h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1})$. By Theorem 5.3, if $h'(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) \neq h'(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1})$, then $\Pr\{h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = h(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1})\} \leq 2/2^{\mathtt{w}}$.

To summarize,

$$\begin{aligned}
&\Pr\left\{ \begin{array}{l} h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) \\ \quad = h(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1}) \end{array} \right\} \\
&= \Pr\left\{ \begin{array}{l} h'(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = h'(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1}) \text{ or} \\ h'(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) \neq h'(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1}) \\ \quad \text{and } \mathtt{z}h'(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) \operatorname{div} 2^{\mathtt{w}} = \mathtt{z}h'(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1}) \operatorname{div} 2^{\mathtt{w}} \end{array} \right\} \\
&\leq 1/2^{\mathtt{w}} + 2/2^{\mathtt{w}} = 3/2^{\mathtt{w}} \ .
\end{aligned}$$

□

5.3.3 Hash Codes for Arrays and Strings

The method from the previous section works well for objects that have a fixed, constant, number of components. However, it breaks down when we want to use it with objects that have a variable number of components since it requires a random $\mathtt{w}$-bit integer $\mathtt{z}_i$ for each component. We could use a pseudorandom sequence to generate as many $\mathtt{z}_i$'s as we need, but then the $\mathtt{z}_i$'s are not mutually independent, and it becomes difficult to prove that the pseudorandom numbers don't interact badly with the hash function we are using. In particular, the values of t and $\mathtt{z}_i$ in the proof of Theorem 5.3 are no longer independent.

A more rigorous approach is to base our hash codes on polynomials over prime fields. This method is based on the following theorem, which says that polynomials over prime fields behave pretty-much like usual polynomials:

Theorem 5.4. *Let* $\mathtt{p}$ *be a prime number, and let* $f(\mathtt{z}) = \mathtt{x}_0\mathtt{z}^0 + \mathtt{x}_1\mathtt{z}^1 + \cdots + \mathtt{x}_{r-1}\mathtt{z}^{r-1}$ *be a non-trivial polynomial with coefficients* $\mathtt{x}_i \in \{0, \ldots, \mathtt{p}-1\}$*. Then the equation* $f(\mathtt{z}) \bmod \mathtt{p} = 0$ *has at most* $r-1$ *solutions for* $\mathtt{z} \in \{0, \ldots, p-1\}$*.*

To use Theorem 5.4, we hash a sequence of integers $\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}$ with each $\mathtt{x}_i \in \{0, \ldots, \mathtt{p}-2\}$ using a random integer $\mathtt{z} \in \{0, \ldots, \mathtt{p}-1\}$ via the formula

$$h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = \left(\mathtt{x}_0\mathtt{z}^0 + \cdots + \mathtt{x}_{r-1}\mathtt{z}^{r-1} + (\mathtt{p}-1)\mathtt{z}^r\right) \bmod \mathtt{p} \ .$$

Note the extra $(\mathtt{p}-1)\mathtt{z}^r$ term at the end of the formula. It helps to think of $(\mathtt{p}-1)$ as the last element, $\mathtt{x}_r$, in the sequence $\mathtt{x}_0, \ldots, \mathtt{x}_r$. Note that this element differs from every other element in the sequence (each of which is in the set $\{0, \ldots, \mathtt{p}-2\}$). We can think of $\mathtt{p}-1$ as an end-of-sequence marker.

The following theorem, which considers the case of two sequences of the same length, shows that this hash function gives a good return for the small amount of randomization needed to choose $\mathtt{z}$:

Theorem 5.5. *Let* $\mathtt{p} > 2^{\mathtt{w}}+1$ *be a prime, let* $\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}$ *and* $\mathtt{y}_0, \ldots, \mathtt{y}_{r-1}$ *each be sequences of* $\mathtt{w}$*-bit integers in* $\{0, \ldots, 2^{\mathtt{w}}-1\}$*, and assume* $\mathtt{x}_i \neq \mathtt{y}_i$ *for at least one index* $i \in \{0, \ldots, r-1\}$*. Then*

$$\Pr\{h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = h(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1})\} \leq (r-1)/\mathtt{p} \ .$$

Proof. The equation $h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = h(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1})$ can be rewritten as

$$\left((\mathtt{x}_0 - \mathtt{y}_0)\mathtt{z}^0 + \cdots + (\mathtt{x}_{r-1} - \mathtt{y}_{r-1})\mathtt{z}^{r-1}\right) \bmod \mathtt{p} = 0. \tag{5.6}$$

Since $\mathtt{x_i} \neq \mathtt{y_i}$, this polynomial is non-trivial. Therefore, by Theorem 5.4, it has at most $r-1$ solutions in $\mathtt{z}$. The probability that we pick $\mathtt{z}$ to be one of these solutions is therefore at most $(r-1)/\mathtt{p}$. □

Note that this hash function also deals with the case in which two sequences have different lengths, even when one of the sequences is a prefix of the other. This is because this function effectively hashes the infinite sequence

$$\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}, \mathtt{p}-1, 0, 0, \ldots \ .$$

This guarantees that if we have two sequences of length r and r' with $r > r'$, then these two sequences differ at index $i = r$. In this case, (5.6) becomes

$$\left(\sum_{i=0}^{i=r'-1} (\mathtt{x}_i - \mathtt{y}_i)\mathtt{z}^i + (\mathtt{x}_{r'} - \mathtt{p} + 1)\mathtt{z}^{r'} + \sum_{i=r'+1}^{i=r-1} \mathtt{x}_i\mathtt{z}^i + (\mathtt{p}-1)\mathtt{z}^r \right) \bmod \mathtt{p} = 0 \ ,$$

which, by Theorem 5.4, has at most r solutions in $\mathtt{z}$. This combined with Theorem 5.5 suffice to prove the following more general theorem:

Theorem 5.6. *Let* $\mathtt{p} > 2^{\mathtt{w}} + 1$ *be a prime, let* $\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}$ *and* $\mathtt{y}_0, \ldots, \mathtt{y}_{r'-1}$ *be distinct sequences of* $\mathtt{w}$*-bit integers in* $\{0, \ldots, 2^{\mathtt{w}} - 1\}$*. Then*

$$\Pr\{h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = h(\mathtt{y}_0, \ldots, \mathtt{y}_{r-1})\} \leq \max\{r, r'\}/\mathtt{p}\} \ .$$

The following example code shows how this hash function is applied to an object that contains an array, `x`, of values:

GeomVector

```
unsigned hashCode() {
  long p = (1L<<32)-5;   // prime: 2^32 - 5
  long z = 0x64b6055aL;  // 32 bits from random.org
  int z2 = 0x5067d19d;   // random odd 32 bit number
  long s = 0;
  long zi = 1;
  for (int i = 0; i < x.length; i++) {
    long long xi = (ods::hashCode(x[i]) * z2) >> 1; // reduce to 31 bits
    s = (s + zi * xi) % p;
    zi = (zi * z) % p;
  }
  s = (s + zi * (p-1)) % p;
  return (int)s;
}
```

The above code sacrifices some collision probability for implementation convenience. In particular, it applies the multiplicative hash function from Section 5.1.1, with $\mathtt{d} = 31$ to reduce `x[i].hashCode()` to a 31-bit value. This is so that the additions and multiplications that are done modulo the prime $\mathtt{p} = 2^{32} - 5$ can be carried out using unsigned 63-bit arithmetic. This means that the probability of two different sequences, the longer of which has length r, having the same hash code is at most

$$2/2^{31} + r/(2^{32} - 5)$$

rather than the $r/(2^{32} - 5)$ specified in Theorem 5.6.

5.4 Discussion and Exercises

Hash tables and hash codes are an enormous and active area of research that is just touched upon in this chapter. The online Bibliography on Hashing [7] contains nearly 2000 entries.

A variety of different hash table implementations exist. The one described in Section 5.1 is known as *hashing with chaining* (each array entry contains a chain (`List`) of elements). Hashing with chaining dates back to an internal IBM memorandum authored by H. P. Luhn and dated January 1953. This memorandum also seems to be one of the earliest references to linked lists.

An alternative to hashing with chaining is that used by *open addressing* schemes, where all data is stored directly in an array. These schemes include the `LinearHashTable` structure of Section 5.2. This idea was also proposed, independently, by a group at IBM in the 1950s. Open addressing schemes must deal with the problem of *collision resolution*: the case where two values hash to the same array location. Different strategies exist for collision resolution and these provide different performance guarantees and often require more sophisticated hash functions than the ones described here.

Yet another category of hash table implementations are the so-called *perfect hashing* methods. These are methods in which `find(x)` operations take $O(1)$ time in the worst-case. For static data sets, this can be accomplished by finding *perfect hash functions* for the data; these are functions that map each piece of data to a unique array location. For data that changes over time, perfect hashing methods include *FKS two-level hash tables* [24, 19] and *cuckoo hashing* [41].

The hash functions presented in this chapter are probably among the most practical currently known methods that can be proven to work well for any set of data. Other provably good methods date back to the pioneering work of Carter and Wegman who introduced the notion of *universal hashing* and described several hash functions for different scenarios [11]. Tabulation hashing, described in Section 5.2.3, is due to Carter and Wegman [11], but its analysis, when applied to linear probing (and several other hash table schemes) is due to Pătraşcu and Thorup [43].

The idea of *multiplicative hashing* is very old and seems to be part of the hashing folklore [38, Section 6.4]. However, the idea of choosing the multiplier `z` to be a random *odd* number, and the analysis in Section 5.1.1 is due to Dietzfelbinger *et al.* [18]. This version of multiplicative hashing is one of the simplest, but its collision probability of $2/2^{\mathtt{d}}$ is a factor of 2 larger than what one could expect with a random function from $2^{\mathtt{w}} \to 2^{\mathtt{d}}$. The

multiply-add hashing method uses the function

$$h(\mathtt{x}) = ((\mathtt{z}\mathtt{x} + b) \bmod 2^{2\mathtt{w}}) \operatorname{div} 2^{2\mathtt{w}-\mathtt{d}}$$

where $\mathtt{z}$ and $\mathtt{b}$ are each randomly chosen from $\{0, \ldots, 2^{2\mathtt{w}} - 1\}$. Multiply-add hashing has a collision probability of only $1/2^{\mathtt{d}}$ [16], but requires $2\mathtt{w}$-bit precision arithmetic.

There are a number of methods of obtaining hash codes from fixed-length sequences of $\mathtt{w}$-bit integers. One particularly fast method [8] is the function

$$h(\mathtt{x}_0, \ldots, \mathtt{x}_{r-1}) = \left(\sum_{i=0}^{r/2-1} ((\mathtt{x}_{2i} + \mathtt{a}_{2i}) \bmod 2^{\mathtt{w}})((\mathtt{x}_{2i+1} + \mathtt{a}_{2i+1}) \bmod 2^{\mathtt{w}}) \right) \bmod 2^{2\mathtt{w}}$$

where r is even and $\mathtt{a}_0, \ldots, \mathtt{a}_{r-1}$ are randomly chosen from $\{0, \ldots, 2^{\mathtt{w}}\}$. This yields a $2\mathtt{w}$-bit hash code that has collision probability $1/2^{\mathtt{w}}$. This can be reduced to a $\mathtt{w}$-bit hash code using multiplicative (or multiply-add) hashing. This method is fast because it requires only $r/2$ $2\mathtt{w}$-bit multiplications whereas the method described in Section 5.3.2 requires r multiplications. (The mod operations occur implicitly by using $\mathtt{w}$ and $2\mathtt{w}$-bit arithmetic for the additions and multiplications, respectively.)

The method from Section 5.3.3 of using polynomials over prime fields to hash variable-length arrays and strings is due to Dietzfelbinger *et al.* [17]. It is, unfortunately, not very fast. This is due to its use of the mod operator which relies on a costly machine instruction. Some variants of this method choose the prime $\mathtt{p}$ to be one of the form $2^{\mathtt{w}} - 1$, in which case the mod operator can be replaced with addition (+) and bitwise-and (&) operations [37, Section 3.6]. Another option is to apply one of the fast methods for fixed-length strings to blocks of length c for some constant $c > 1$ and then apply the prime field method to the resulting sequence of $\lceil r/c \rceil$ hash codes.

Exercise 5.1. Prove that the bound $2/2^{\mathtt{d}}$ in Lemma 5.1 is the best possible by showing that, if $x = 2^{\mathtt{w}-\mathtt{d}-2}$ and $\mathtt{y} = 3\mathtt{x}$, then $\Pr\{\mathtt{hash}(\mathtt{x}) = \mathtt{hash}(\mathtt{y})\} = 2/2^{\mathtt{d}}$. (Hint look at the binary representations of $\mathtt{z}\mathtt{x}$ and $\mathtt{z}3\mathtt{x}$ and use the fact that $\mathtt{z}3\mathtt{x} = \mathtt{z}x + 2z\mathtt{x}$.)

Exercise 5.2. Reprove Lemma 5.4 using the full version of Stirling's Approximation given in Section 1.2.2.

Exercise 5.3. Consider the following the simplified version of the code for adding an element $\mathtt{x}$ to a `LinearHashTable`. This code simply stores $\mathtt{x}$ in the first `null` array entry it finds. Explain why this could be very slow by giving an example of a sequence of $O(\mathtt{n})$ $\mathtt{add}(\mathtt{x})$, $\mathtt{remove}(\mathtt{x})$, and $\mathtt{find}(\mathtt{x})$ operations that would take on the order of $\mathtt{n}^2$ time to execute.

LinearHashTable

```
bool addSlow(T x) {
  if (2*(q+1) > t.length) resize();   // max 50% occupancy
  int i = hash(x);
  while (t[i] != null) {
      if (t[i] != del && x.equals(t[i])) return false;
      i = (i == t.length-1) ? 0 : i + 1; // increment i (mod t.length)
  }
  t[i] = x;
  n++; q++;
  return true;
}
```

Exercise 5.4. Suppose you have an object made up of two `w`-bit integers `x` and `y`. Show why $\texttt{x} \oplus \texttt{y}$ does not make a good hash code for your object. Give an example of a large set of objects that would all have hash code 0.

Exercise 5.5. Suppose you have an object made up of two `w`-bit integers `x` and `y`. Show why $\texttt{x} + \texttt{y}$ does not make a good hash code for your object. Give an example of a large set of objects that would all have the same hash code.

Exercise 5.6. Suppose you have an object made up of two `w`-bit integers `x` and `y`. Suppose that the hash code for your object is defined by some deterministic function $h(x, y)$. Prove that there exists a large set of objects that have the same hash code.

Exercise 5.7. Let $p = 2^{\texttt{w}} - 1$ for some positive integer `w`. Explain why, for a positive integer x

$$(x \bmod 2^{\texttt{w}}) + (x \operatorname{div} 2^{\texttt{w}}) \equiv x \bmod (2^{\texttt{w}} - 1) \ .$$

(This gives an algorithm for computing $x \bmod (2^{\texttt{w}} - 1)$ by repeatedly setting

$$\texttt{x} = \texttt{x}\&((\texttt{1} << \texttt{w}) - \texttt{1}) + \texttt{x} >>> \texttt{w}$$

until $\texttt{x} \leq 2^{\texttt{w}} - 1$.)

Chapter 6

Binary Trees

This chapter introduces one of the most fundamental structures in computer science: binary trees. There are lots of ways of defining binary trees. Mathematically, a binary tree is a connected undirected finite graph with no cycles, and no vertex of degree greater than three.

For most computer science applications, binary trees are *rooted*: A special node, `r`, of degree at most two is called the *root* of the tree. For every node, `u` $\neq$ `r`, the second node on the path from `u` to `r` is called the *parent* of `u`. Each of the other nodes adjacent to `u` is called a *child* of `u`. Most of the binary trees we are interested in are *ordered*, so we distinguish between the *left child* and *right child* of `u`.

In illustrations, binary trees are usually drawn from the root downward, with the root at the top of the drawing and the left and right children respectively given by left and right positions in the drawing (Figure 6.1). A binary tree with nine nodes is drawn this way in Figure 6.2.a.

Binary trees are so important that a terminology has developed around them: The *depth* of a node, `u`, in a binary tree is the length of the path from `u` to the root of the tree.

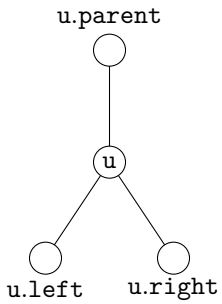


Figure 6.1: The parent, left child, and right child of the node `u` in a `BinaryTree`.

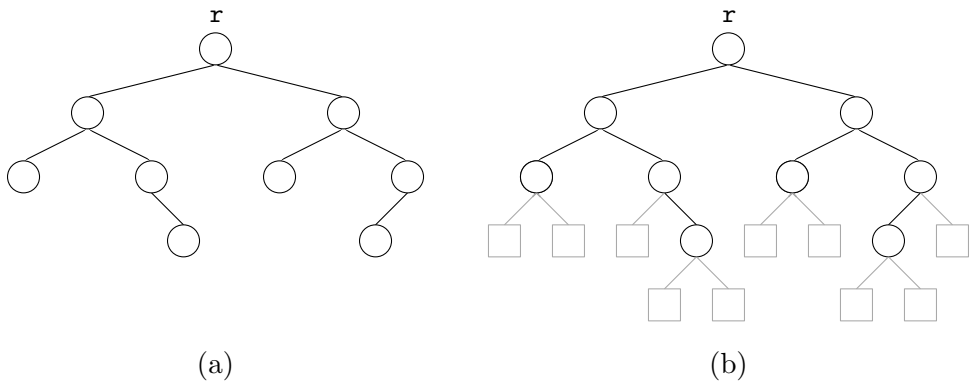


Figure 6.2: A binary tree with (a) nine real nodes and (b) ten external nodes.

If a node, w, is on the path from u to r then w is called an *ancestor* of u and u a *descendant* of w. The *subtree* of a node, u, is the binary tree that is rooted at u and contains all of u's descendants. The *height* of a node, u, is the length of the longest path from u to one of its descendants. The height of a tree is the height of its root. A node, u, is a *leaf* if it has no children.

We sometimes think of the tree as being augmented with *external nodes*. Any node that does not have a left child has an external node as its left child and any node that does not have a right child has an external node as its right child (see Figure 6.2.b). It is easy to verify, by induction, that a binary tree having $n \geq 1$ real nodes has $n + 1$ external nodes.

6.1 BinaryTree: A Basic Binary Tree

The simplest way to represent a node, u, in a binary tree is to store the (at most three) neighbours of u explicitly:

BinaryTree

```
class BTNode {
  N *left;
  N *right;
  N *parent;
  BTNode() {
    left = right = parent = NULL;
  }
};
```

When one of these three neighbours is not present, we set it to nil. In this way, external nodes in the tree as well as the parent of the root correspond to the value nil.

The binary tree itself can then be represented by a pointer to its root node, r:

BinaryTree
```
Node *r;    // root node
```

We can compute the depth of a node, u, in a binary tree by counting the number of steps on the path from u to the root:

BinaryTree
```
int depth(Node *u) {
  int d = 0;
  while (u != r) {
    u = u->parent;
    d++;
  }
  return d;
}
```

6.1.1 Recursive Algorithms

It is very easy to compute facts about binary trees using recursive algorithms. For example, to compute the size of (number of nodes in) a binary tree rooted at node u, we recursively compute the sizes of the two subtrees rooted at the children of u, sum these sizes, and add one:

BinaryTree
```
int size(Node *u) {
  if (u == nil) return 0;
  return 1 + size(u->left) + size(u->right);
}
```

To compute the height of a node u we can compute the height of u's two subtrees, take the maximum, and add one:

BinaryTree
```
int height(Node *u) {
  if (u == nil) return -1;
  return 1 + max(height(u->left), height(u->right));
}
```

6.1.2 Traversing Binary Trees

The two algorithms from the previous section use recursion to visit all the nodes in a binary tree. Each of them visits the nodes of the binary tree in the same order as the following code:

BinaryTree
```
void traverse(Node *u) {
    if (u == nil) return;
    traverse(u->left);
    traverse(u->right);
}
```

Using recursion this way produces very short, simple code, but can be problematic. The maximum depth of the recursion is given by the maximum depth of a node in the binary tree, i.e., the tree's height. If the height of the tree is very large, then this could very well use more stack space than is available, causing a crash.

Luckily, traversing a binary tree can be done without recursion. This is done using an algorithm that uses where it came from to decide where it will go next. See Figure 6.3. If we arrive at a node `u` from `u.parent`, then the next thing to do is to visit `u.left`. If we arrive at `u` from `u.left`, then the next thing to do is to visit `u.right`. If we arrive at `u` from `u.right`, then we are done visiting `u`'s subtree, so we return to `u.parent`. The following code implements this idea, with code included for handling the cases where any of `u.left`, `u.right`, or `u.parent` is `nil`:

BinaryTree
```
void traverse2() {
  Node *u = r, *prev = nil, *next;
  while (u != nil) {
    if (prev == u->parent) {
      if (u->left != nil) next = u->left;
      else if (u->right != nil) next = u->right;
      else next = u->parent;
    } else if (prev == u->left) {
      if (u->right != nil) next = u->right;
      else next = u->parent;
    } else {
      next = u->parent;
    }
    prev = u;
    u = next;
  }
}
```

The same things that can be computed with recursive algorithms can also be done this way. For example, to compute the size of the tree we keep a counter, `n`, and increment `n` whenever visiting a node for the first time:

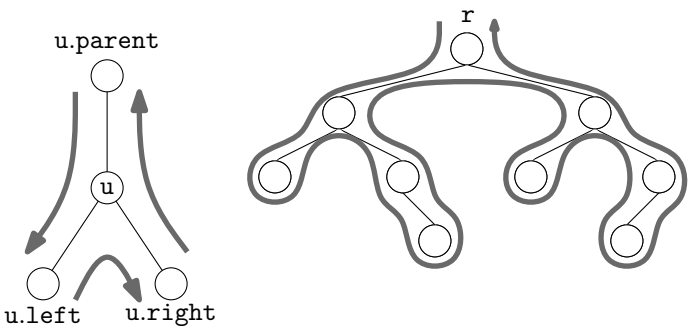


Figure 6.3: The three cases that occur at node u when traversing a binary tree non-recursively, and the resulting traversal of the tree.

BinaryTree

```
int size2() {
    Node *u = r, *prev = nil, *next;
    int n = 0;
    while (u != nil) {
      if (prev == u->parent) {
        n++;
        if (u->left != nil) next = u->left;
        else if (u->right != nil) next = u->right;
        else next = u->parent;
      } else if (prev == u->left) {
        if (u->right != nil) next = u->right;
        else next = u->parent;
      } else {
        next = u->parent;
      }
      prev = u;
      u = next;
    }
    return n;
  }
```

In some implementations of binary trees, the `parent` field is not used. When this is the case, a non-recursive implementation is still possible, but the implementation has to use a `List` (or `Stack`) to keep track of the path from the current node to the root.

A special kind of traversal that does not fit the pattern of the above functions is the *breadth-first traversal*. In a breadth-first traversal, the nodes are visited level-by-level starting at the root and working our way down, visiting the nodes at each level from left

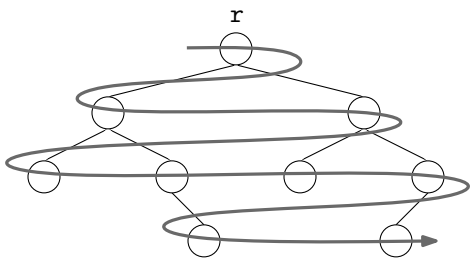


Figure 6.4: During a breadth-first traversal, the nodes of a binary tree are visited level-by-level, and left-to-right within each level.

to right. This is similar to the way we would read a page of English text. (See Figure 6.4.) This is implemented using a queue, q, that initially contains only the root, r. At each step, we extract the next node, u, from q, process u and add u.left and u.right (if they are non-nil) to q:

BinaryTree

```
void bfTraverse() {
  ArrayDeque<Node*> q;
  if (r != nil) q.add(q.size(),r);
  while (q.size() > 0) {
    Node *u = q.remove(q.size()-1);
    if (u->left != nil) q.add(q.size(),u->left);
    if (u->right != nil) q.add(q.size(),u->right);
  }
}
```

6.2 BinarySearchTree: An Unbalanced Binary Search Tree

A BinarySearchTree is a special kind of binary tree in which each node, u, also stores a data value, u.x, from some total order. The data values in a binary search tree obey the *binary search tree property*: For a node, u, every data value stored in the subtree rooted at u.left is less than u.x and every data value stored in the subtree rooted at u.right is greater than u.x. An example of a BinarySearchTree is shown in Figure 6.5.

6.2.1 Searching

The binary search tree property is extremely useful because it allows us to quickly locate a value, x, in a binary search tree. To do this we start searching for x at the root, r. When examining a node, u, there are three cases:

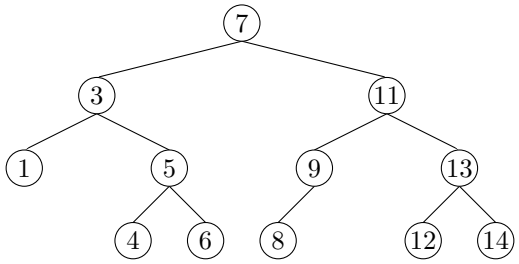


Figure 6.5: A binary search tree.

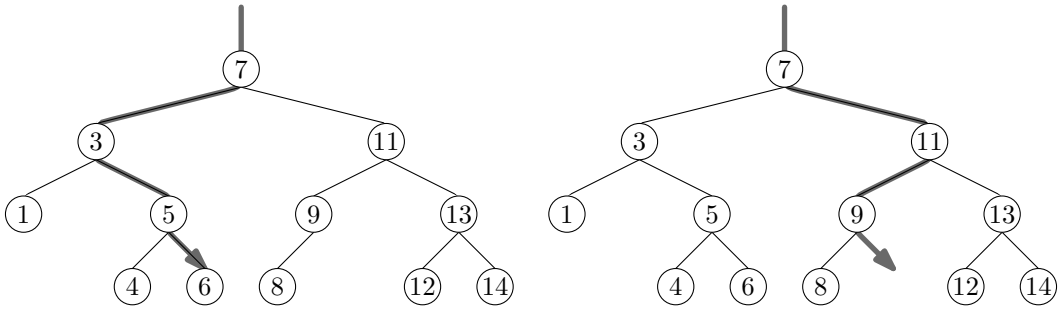
1. If x < u.x then the search proceeds to u.left;
2. If x > u.x then the search proceeds to u.right;
3. If x = u.x then we have found the node u containing x.

The search terminates when Case 3 occurs or when u = nil. In the former case, we found x. In the latter case, we conclude that x is not in the binary search tree.

BinarySearchTree

```
T findEQ(T x) {
  Node *w = r;
  while (w != nil) {
    int comp = compare(x, w->x);
    if (comp < 0) {
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
      return w->x;
    }
  }
  return null;
}
```

Two examples of searches in a binary search tree are shown in Figure 6.6. As the second example shows, even if we don't find x in the tree, we still gain some valuable information. If we look at the last node, u, at which Case 1 occurred, we see that u.x is the smallest value in the tree that is greater than x. Similarly, the last node at which Case 2



(a) (b)

Figure 6.6: An example of (a) a successful search (for 6) and (b) an unsuccessful search (for 10) in a binary search tree.

occurred contains the largest value in the tree that is less than x. Therefore, by keeping track of the last node, z, at which Case 1 occurs, a BinarySearchTree can implement the find(x) operation that returns the smallest value stored in the tree that is greater than or equal to x:

BinarySearchTree

```
T find(T x) {
  Node *w = r, *z = nil;
  while (w != nil) {
    int comp = compare(x, w->x);
    if (comp < 0) {
      z = w;
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
      return w->x;
    }
  }
  return z == nil ? null : z->x;
}
```

6.2.2 Inserting

To add a new value, x, to a BinarySearchTree, we first search for x. If we find it, then there is no need to insert it. Otherwise, we store x at a leaf child of the last node, p, encountered during the search for x. Whether the new node is the left or right child of p depends on the

result of comparing x and p.x.

BinarySearchTree

```
bool add(T x) {
  Node *p = findLast(x);
  Node *u = new Node;
  u->x = x;
  return addChild(p, u);
}
```

BinarySearchTree

```
Node* findLast(T x) {
  Node *w = r, *prev = nil;
  while (w != nil) {
    prev = w;
    int comp = compare(x, w->x);
    if (comp < 0) {
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
      return w;
    }
  }
  return prev;
}
```

BinarySearchTree

```
bool addChild(Node *p, Node *u) {
    if (p == nil) {
      r = u;              // inserting into empty tree
    } else {
      int comp = compare(u->x, p->x);
      if (comp < 0) {
        p->left = u;
      } else if (comp > 0) {
        p->right = u;
      } else {
        return false;   // u.x is already in the tree
      }
      u->parent = p;
    }
    n++;
    return true;
  }
```

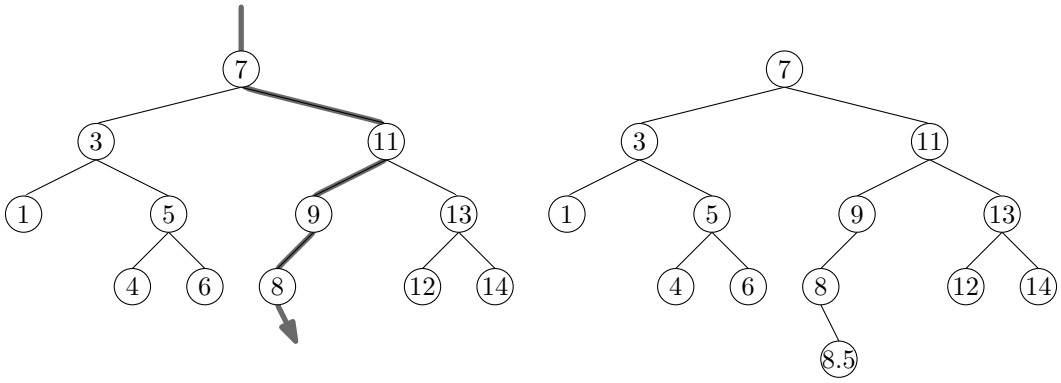


Figure 6.7: Inserting the value 8.5 into a binary search tree.

An example is shown in Figure 6.7. The most time-consuming part of this process is the initial search for x, which takes time proportional to the height of the newly added node u. In the worst case, this is equal to the height of the BinarySearchTree.

6.2.3 Deleting

Deleting a value stored in a node, u, of a BinarySearchTree is a little more difficult. If u is a leaf, then we can just detach u from its parent. Even better: If u has only one child, then we can splice u from the tree by having u.parent adopt u's child:

BinarySearchTree

```
void splice(Node *u) {
  Node *s, *p;
  if (u->left != nil) {
    s = u->left;
  } else {
    s = u->right;
  }
  if (u == r) {
    r = s;
    p = nil;
  } else {
    p = u->parent;
    if (p->left == u) {
      p->left = s;
    } else {
      p->right = s;
    }
  }
  if (s != nil) {
    s->parent = p;
```

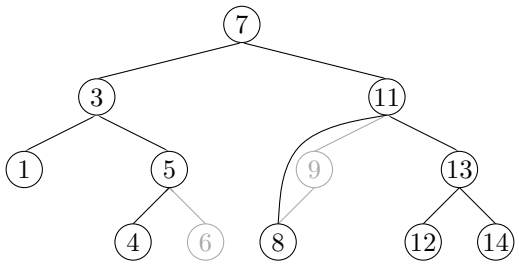


Figure 6.8: Deleting a leaf (6) or a node with only one child (9) is easy.

```
  }
  n--;
}
```

Things get tricky, though, when u has two children. In this case, the simplest thing to do is to find a node, w, that has less than two children such that we can replace u.x with w.x. To maintain the binary search tree property, the value w.x should be close to the value of u.x. For example, picking w such that w.x is the smallest value greater than u.x will do. Finding the node w is easy; it is the smallest value in the subtree rooted at u.right. This node can be easily removed because it has no left child. (See Figure 6.9)

BinarySearchTree

```
void remove(Node *u) {
  if (u->left == nil || u->right == nil) {
    splice(u);
    delete u;
  } else {
    Node *w = u->right;
    while (w->left != nil)
      w = w->left;
    u->x = w->x;
    splice(w);
    delete w;
  }
}
```

6.2.4 Summary

The find(x), add(x), and remove(x) operations in a BinarySearchTree each involve following a path from the root of the tree to some node in the tree. Without knowing more about the shape of the tree it is difficult to say much about the length of this path, except

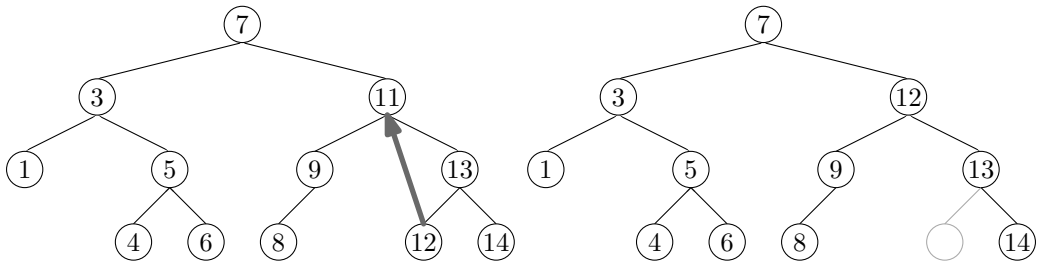


Figure 6.9: Deleting a value (11) from a node, u, with two children is done by replacing u's value with the smallest value in the right subtree of u.

that it is less than n, the number of nodes in the tree. The following (unimpressive) theorem summarizes the performance of the BinarySearchTree data structure:

Theorem 6.1. *A* BinarySearchTree *implements the* SSet *interface. A* BinarySearchTree *supports the operations* add(x)*,* remove(x)*, and* find(x) *in* $O(n)$ *time per operation.*

Theorem 6.1 compares poorly with Theorem 4.1, which shows that the SkiplistSSet structure can implement the SSet interface with $O(\log n)$ expected time per operation. The problem with the BinarySearchTree structure is that it can become *unbalanced*. Instead of looking like the tree in Figure 6.5 it can look like a long chain of n nodes, all but the last having exactly one child.

There are a number of ways of avoiding unbalanced binary search trees, all of which lead to data structures that have $O(\log n)$ time operations. In Chapter 7 we show how $O(\log n)$ *expected* time operations can be achieved with randomization. In Chapter 8 we show how $O(\log n)$ *amortized* time operations can be achieved with partial rebuilding operations. In Chapter 9 we show how $O(\log n)$ *worst-case* time operations can be achieved by simulating a tree that is not binary: a tree in which nodes can have up to four children.

6.3 Discussion and Exercises

Binary trees have been used to model relationships for literally thousands of years. One reason for this is that binary trees naturally model (pedigree) family trees. These are the family trees in which the root is a person, the left and right children are the person's parents, and so on, recursively. In more recent centuries binary trees have also been used to model species-trees in biology, where the leaves of the tree represent extant species and the internal nodes of the tree represent *speciation events* in which two populations of a single species evolve into two separate species.

Binary search trees appear to have been discovered independently by several groups in the 1950s [38, Section 6.2.2]. Further references to specific kinds of binary search trees are provided in subsequent chapters.

Exercise 6.1. Write a non-recursive variant of the `size2`() method, `size`(`u`), that computes the size of the subtree rooted at node `u`.

Exercise 6.2. Write a non-recursive method, `height2`(`u`), that computes the height of node `u` in a binary search tree.

Exercise 6.3. A *pre-order* traversal of a binary tree is a traversal that visits each node, `u`, before any of its children. An *in-order* traversal visits `u` after visiting all the nodes in `u`'s left subtree but before visiting any of the nodes in `u`'s right subtree. A *post-order* traversal visits `u` only after visiting all other nodes in `u`'s subtree.

Write non-recursive functions `nextPreOrder`(`u`), `nextInOrder`(`u`), and `nextPostOrder`(`u`) that return the node that follows `u` in a pre-order, in-order, or post-order traversal, respectively.

Exercise 6.4. Describe a sequence of `n` operations on an initially empty `BinarySearchTree` that results in a tree of height $\texttt{n} - 1$.

Exercise 6.5. Design and implement a version of `BinarySearchTree` in which each node, `u`, maintains values `u.size` (the size of the subtree rooted at `u`), `u.depth` (the depth of `u`), and `u.height` (the height of the subtree rooted at `u`).

These values should be maintained, even during the `add`(`x`) and `remove`(`x`) operations, but this should not increase the cost of these operations by more than a constant factor.

Chapter 7

Random Binary Search Trees

In this chapter, we present a binary search tree structure that uses randomization to achieve $O(\log \mathtt{n})$ expected time for all operations.

7.1 Random Binary Search Trees

Consider the two binary search trees shown in Figure 7.1. The one on the left is a list and the other is a perfectly balanced binary search tree. The one on the left has height $\mathtt{n} - 1 = 14$ and the one on the right has height three.

Imagine how these two trees could have been constructed. The one on the left occurs if we start with an empty `BinarySearchTree` and add the sequence

$$\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 \rangle \ .$$

No other sequence of additions will create this tree (as you can prove by induction on $\mathtt{n}$). On the other hand, the tree on the right can be created by the sequence

$$\langle 7, 3, 11, 1, 5, 9, 13, 0, 2, 4, 6, 8, 10, 12, 14 \rangle \ .$$

Other sequences work as well, including

$$\langle 7, 3, 1, 5, 0, 2, 4, 6, 11, 9, 13, 8, 10, 12, 14 \rangle \ ,$$

and

$$\langle 7, 3, 1, 11, 5, 0, 2, 4, 6, 9, 13, 8, 10, 12, 14 \rangle \ .$$

In fact, there are $21,964,800$ addition sequences that generate the tree on the right and only one that generates the tree on the left.

The above example gives some anecdotal evidence that, if we choose a random permutation of $0, \ldots, 14$, and add it into a binary search tree then we are more likely to get

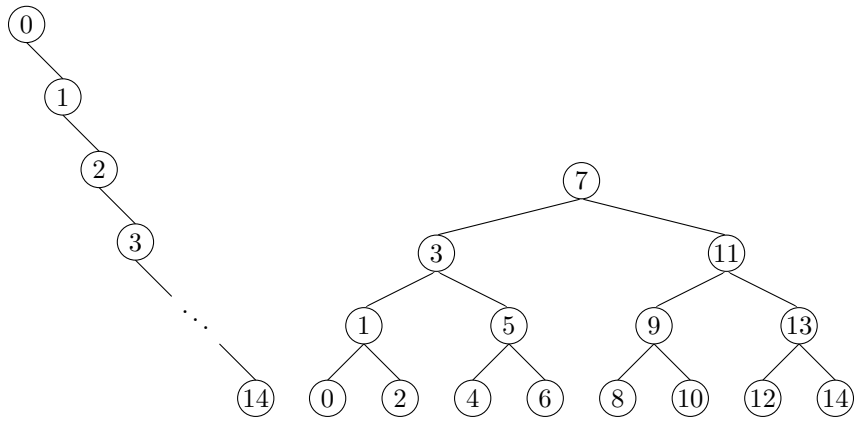


Figure 7.1: Two binary search trees containing the integers $0, \ldots, 14$.

a very balanced tree (the right hand of Figure 7.1) than we are to get a very unbalanced tree (the left hand of Figure 7.1).

We can formalize this notion by studying random binary search trees. A *random binary search tree* of size $\mathtt{n}$ is obtained in the following way: Take a random permutation $\mathtt{x}_0, \ldots, \mathtt{x}_{\mathtt{n}-1}$ of $0, \ldots, \mathtt{n}-1$ and add its elements, one by one, into a `BinarySearchTree`.

Note that the values $0, \ldots, \mathtt{n}-1$ could be replaced by any ordered set of $\mathtt{n}$ elements without changing any of the properties of the random binary search tree. The element $\mathtt{x} \in \{0, \ldots, \mathtt{n}-1\}$ is simply standing in for the element of rank $\mathtt{x}$ in an ordered set of size $\mathtt{n}$.

Before we can present our main result about random binary search trees, we must take some time for a short digression to discuss a type of number that comes up frequently when studying randomized structures. For a non-negative integer, k, the k-th *harmonic number*, denoted H_k, is defined as

$$H_k = 1 + 1/2 + 1/3 + \cdots + 1/k \ .$$

The harmonic number H_k has no simple closed form, but it is very closely related to the natural logarithm of k. In particular,

$$\ln k \le H_k \le \ln k + 1 \ .$$

Readers who have studied calculus might notice that this is because the integral $\int_1^k (1/x)\,\mathrm{d}x = \ln k$. Keeping in mind that an integral can be interpreted as the area between a curve and the x-axis, the value of H_k can be lower-bounded by the integral $\int_1^k (1/x)\,\mathrm{d}x$ and upper-bounded by $1 + \int_1^k (1/x)\,\mathrm{d}x$. (See Figure 7.2 for a graphical explanation.)

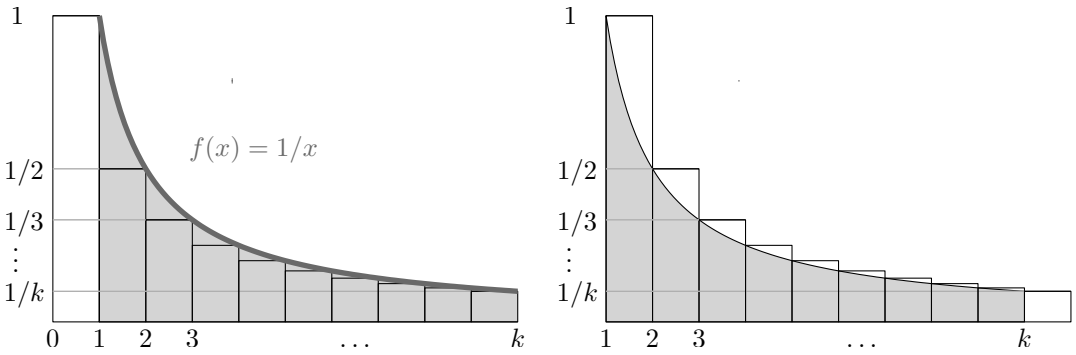


Figure 7.2: The kth harmonic number $H_k = \sum_{i=1}^{k} 1/i$ is upper-bounded by $1 + \int_1^k (1/x)\,\mathrm{d}x$ and lower-bounded by $\int_1^k (1/x)\,\mathrm{d}x$.

Lemma 7.1. *In a random binary search tree of size* $\mathtt{n}$*, the following statements hold:*

1. *For any* $\mathtt{x} \in \{0, \ldots, \mathtt{n}-1\}$*, the expected length of the search path for* $\mathtt{x}$ *is* $H_{\mathtt{x}+1} + H_{\mathtt{n}-\mathtt{x}} - O(1)$.[1]
2. *For any* $\mathtt{x} \in (-1, n) \setminus \{0, \ldots, \mathtt{n}-1\}$*, the expected length of the search path for* $\mathtt{x}$ *is* $H_{\lceil \mathtt{x} \rceil} + H_{\mathtt{n}-\lceil \mathtt{x} \rceil}$.

We will prove Lemma 7.1 in the next section. For now, we will consider what the two parts of Lemma 7.1 tell us. The first part tells us that if we search for an element in a tree of size $\mathtt{n}$, then the expected length of the search path is at most $2 \ln n + O(1)$. The second part tells us the same thing about searching for a value not stored in the tree. When we compare the two parts of the lemma, we see that it is only slightly faster to search for something that is in a tree compared to something that is not in a tree.

7.1.1 Proof of Lemma 7.1

The key observation needed to prove Lemma 7.1 is the following: The search path for a value $\mathtt{x}$ in the open interval $(-1, \mathtt{n})$ in a random binary search tree, T, contains the node with key $i < \mathtt{x}$ if and only if, in the random permutation used to create T, i appears before any of $\{i+1, i+2, \ldots, \lfloor \mathtt{x} \rfloor\}$.

To see this, refer to Figure 7.3 and notice that, until some value in $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ is added, the search paths for each value in the open interval $(i-1, \lfloor \mathtt{x} \rfloor + 1)$ are identical. (Remember that for two search values to have different search paths, there must be some

[1] The expressions $\mathtt{x}+1$ and $\mathtt{n}-\mathtt{x}$ can be interpreted respectively as the number of elements in the tree less than or equal to $\mathtt{x}$ and the number of elements in the tree greater than or equal to $\mathtt{x}$.

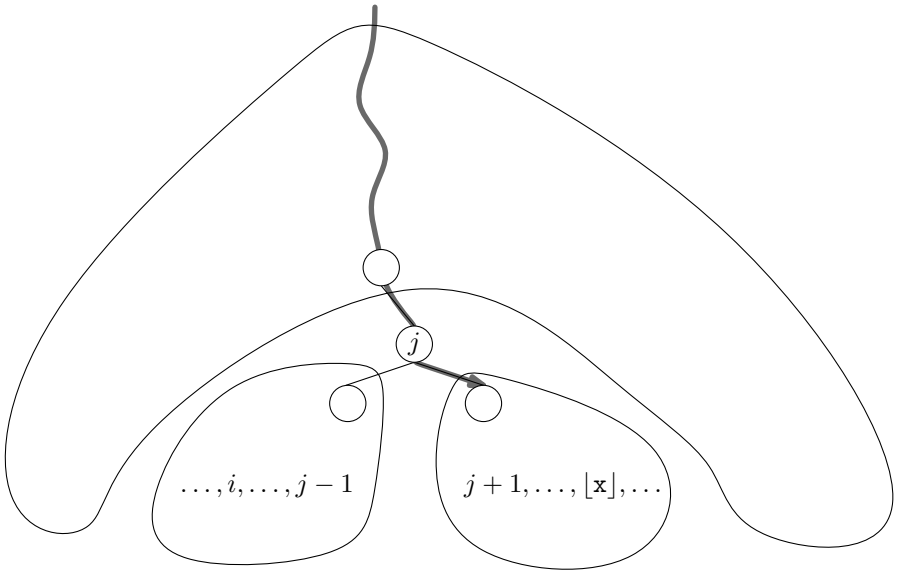


Figure 7.3: The value $i < \mathtt{x}$ is on the search path for $\mathtt{x}$ if and only if i is the first element among $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ added to the tree.

element in the tree that compares differently with them.) Let j be the first element in $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ to appear in the random permutation. Notice that j is now and will always be on the search path for $\mathtt{x}$. If $j \neq i$ then the node $\mathtt{u}_j$ containing j is created before the node $\mathtt{u}_i$ that contains i. Later, when i is added, it will be added to the subtree rooted at $\mathtt{u}_j.\mathtt{left}$, since $i < j$. On the other hand, the search path for $\mathtt{x}$ will never visit this subtree because it will proceed to $\mathtt{u}_j.\mathtt{right}$ after visiting $\mathtt{u}_j$.

Similarly, for $i > \mathtt{x}$, i appears in the search path for $\mathtt{x}$ if and only if i appears before any of $\{\lceil \mathtt{x} \rceil, \lceil \mathtt{x} \rceil + 1, \ldots, i-1\}$ in the random permutation used to create T.

Notice that, if we start with a random permutation of $\{0, \ldots, \mathtt{n}\}$, then the subsequences containing only $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ and $\{\lceil \mathtt{x} \rceil, \lceil \mathtt{x} \rceil + 1, \ldots, i-1\}$ are also random permutations of their respective elements. Each element, then, in the subsets $\{i, i+1, \ldots, \lfloor \mathtt{x} \rfloor\}$ and $\{\lceil \mathtt{x} \rceil, \lceil \mathtt{x} \rceil + 1, \ldots, i-1\}$ is equally likely to appear before any other in its subset in the random permutation used to create T. So we have

$$\Pr\{i \text{ is on the search path for } \mathtt{x}\} = \begin{cases} 1/(\lfloor \mathtt{x} \rfloor - i + 1) & \text{if } i < \mathtt{x} \\ 1/(i - \lceil \mathtt{x} \rceil + 1) & \text{if } i > \mathtt{x} \end{cases} .$$

With this observation, the proof of Lemma 7.1 involves some simple calculations with harmonic numbers:

Proof of Lemma 7.1. Let I_i be the indicator random variable that is equal to one when i appears on the search path for $\mathtt{x}$ and zero otherwise. Then the length of the search path is given by

$$\sum_{i\in\{0,\ldots,\mathtt{n}-1\}\setminus\{\mathtt{x}\}} I_i$$

so, if $\mathtt{x} \in \{0,\ldots,\mathtt{n}-1\}$, the expected length of the search path is given by

$$\begin{aligned}
\mathrm{E}\left[\sum_{i=0}^{\mathtt{x}-1} I_i + \sum_{i=\mathtt{x}+1}^{\mathtt{n}-1} I_i\right] &= \sum_{i=0}^{\mathtt{x}-1} \mathrm{E}\left[I_i\right] + \sum_{i=\mathtt{x}+1}^{\mathtt{n}-1} \mathrm{E}\left[I_i\right] \\
&= \sum_{i=0}^{\mathtt{x}-1} 1/(\lfloor \mathtt{x} \rfloor - i + 1) + \sum_{i=\mathtt{x}+1}^{\mathtt{n}-1} 1/(i - \lceil \mathtt{x} \rceil + 1) \\
&= \sum_{i=0}^{\mathtt{x}-1} 1/(\mathtt{x} - i + 1) + \sum_{i=\mathtt{x}+1}^{\mathtt{n}-1} 1/(i - \mathtt{x} + 1) \\
&= \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\mathtt{x}+1} \\
&+ \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\mathtt{n}-\mathtt{x}} \\
&= H_{\mathtt{x}+1} + H_{\mathtt{n}-\mathtt{x}} - 2 \ .
\end{aligned}$$

The corresponding calculations for a search value $\mathtt{x} \in (-1, n) \setminus \{0,\ldots,\mathtt{n}-1\}$ are almost identical. □

7.1.2 Summary

The following theorem summarizes the performance of a random binary search tree:

Theorem 7.1. *A random binary search tree can be constructed in $O(\mathtt{n}\log\mathtt{n})$ time. In a random binary search tree, the* `find(x)` *operation takes $O(\log\mathtt{n})$ expected time.*

7.2 `Treap`: A Randomized Binary Search Tree

The problem with random binary search trees is, of course, that they are not dynamic. They don't support the `add(x)` or `remove(x)` operations needed to implement the `SSet` interface. In this section we describe a data structure called a `Treap` that uses Lemma 7.1 to implement the `SSet` interface.

A node in a `Treap` is like a node in a `BinarySearchTree` in that it has a data value, `x`, but it also contains a unique numerical *priority*, `p`, that is assigned at random:

`Treap`

```
class TreapNode : public BSTNode<Node, T> {
  friend class Treap<Node,T>;
```

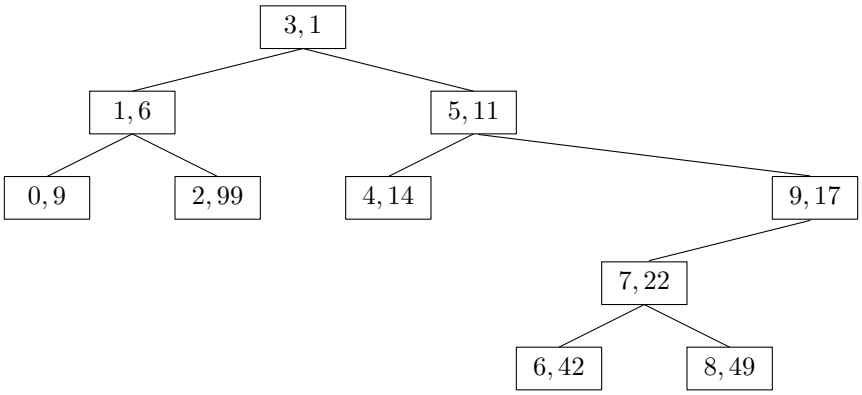


Figure 7.4: An example of a `Treap` containing the integers $0, \ldots, 9$. Each node, `u`, is illustrated with `u.x`, `u.p`.

```
    int p;
};
```

In addition to being a binary search tree, the nodes in a `Treap` also obey the *heap property*: At every node `u`, except the root, `u.parent.p` $<$ `u.p`. That is, each node has a priority smaller than that of its two children. An example is shown in Figure 7.4.

The heap and binary search tree conditions together ensure that, once the key (`x`) and priority (`p`) for each node are defined, the shape of the `Treap` is completely determined. The heap property tells us that the node with minimum priority has to be the root, `r`, of the `Treap`. The binary search tree property tells us that all nodes with keys smaller than `r.x` are stored in the subtree rooted at `r.left` and all nodes with keys larger than `r.x` are stored in the subtree rooted at `r.right`.

The important point about the priority values in a `Treap` is that they are unique and assigned at random. Because of this, there are two equivalent ways we can think about a `Treap`. As defined above, a `Treap` obeys the heap and binary search tree properties. Alternatively, we can think of a `Treap` as a `BinarySearchTree` whose nodes were added in increasing order of priority. For example, the `Treap` in Figure 7.4 can be obtained by adding the sequence of $(\texttt{x}, \texttt{p})$ values

$$\langle (3,1), (1,6), (0,9), (5,11), (4,14), (9,17), (7,22), (6,42), (8,49), (2,99) \rangle$$

into a `BinarySearchTree`.

Since the priorities are chosen randomly, this is equivalent to taking a random per-

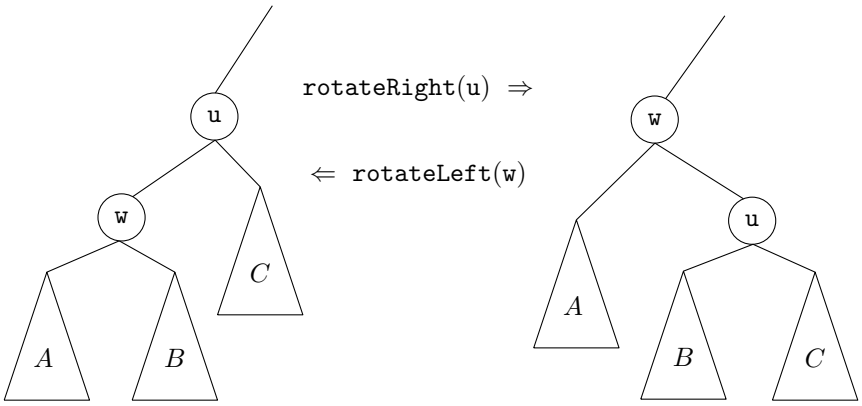


Figure 7.5: Left and right rotations in a binary search tree.

mutation of the keys — in this case the permutation is

$$\langle 3, 1, 0, 5, 9, 4, 7, 6, 8, 2 \rangle$$

— and adding these to a `BinarySearchTree`. But this means that the shape of a treap is identical to that of a random binary search tree. In particular, if we replace each key `x` by its rank,[2] then Lemma 7.1 applies. Restating Lemma 7.1 in terms of `Treap`s, we have:

Lemma 7.2. *In a* `Treap` *that stores a set* S *of* `n` *keys, the following statements hold:*

1. *For any* $\mathtt{x} \in S$*, the expected length of the search path for* `x` *is* $H_{r(\mathtt{x})+1} + H_{\mathtt{n}-r(\mathtt{x})} - O(1)$.
2. *For any* $\mathtt{x} \notin S$*, the expected length of the search path for* `x` *is* $H_{r(\mathtt{x})} + H_{\mathtt{n}-r(\mathtt{x})}$.

Here, $r(\mathtt{x})$ *denotes the rank of* `x` *in the set* $S \cup \{\mathtt{x}\}$.

Lemma 7.2 tells us that `Treap`s can implement the `find(x)` operation efficiently. However, the real benefit of a `Treap` is that it can support the `add(x)` and `delete(x)` operations. To do this, it needs to perform rotations in order to maintain the heap property. Refer to Figure 7.5. A *rotation* in a binary search tree is a local modification that takes a parent `u` of a node `w` and makes `w` the parent of `u`, while preserving the binary search tree property. Rotations come in two flavours: *left* or *right* depending on whether `w` is a right or left child of `u`, respectively.

The code that implements this has to handle these two possibilities and be careful of a boundary case (when `u` is the root) so the actual code is a little longer than Figure 7.5 would lead a reader to believe:

[2]The rank of an element `x` in a set S of elements is the number of elements in S that are less than `x`.

BinarySearchTree

```
void rotateLeft(Node *u) {
  Node *w = u->right;
  w->parent = u->parent;
  if (w->parent != nil) {
    if (w->parent->left == u) {
      w->parent->left = w;
    } else {
      w->parent->right = w;
    }
  }
  u->right = w->left;
  if (u->right != nil) {
    u->right->parent = u;
  }
  u->parent = w;
  w->left = u;
  if (u == r) { r = w; r->parent = nil; }
}
void rotateRight(Node *u) {
  Node *w = u->left;
  w->parent = u->parent;
  if (w->parent != nil) {
    if (w->parent->left == u) {
      w->parent->left = w;
    } else {
      w->parent->right = w;
    }
  }
  u->left = w->right;
  if (u->left != nil) {
    u->left->parent = u;
  }
  u->parent = w;
  w->right = u;
  if (u == r) { r = w; r->parent = nil; }
}
```

In terms of the `Treap` data structure, the most important property of a rotation is that the depth of `w` decreases by one while the depth of `u` increases by one.

Using rotations, we can implement the `add(x)` operation as follows: We create a new node, `u`, and assign `u.x` = `x` and pick a random value for `u.p`. Next we add `u` using the usual `add(x)` algorithm for a `BinarySearchTree`, so that `u` is now a leaf of the `Treap`. At

this point, our Treap satisfies the binary search tree property, but not necessarily the heap property. In particular, it may be the case that u.parent.p > u.p. If this is the case, then we perform a rotation at node w=u.parent so that u becomes the parent of w. If u continues to violate the heap property, we will have to repeat this, decreasing u's depth by one every time, until u either becomes the root or u.parent.p < u.p.

Treap

```
bool add(T x) {
  Node *u = new Node;
  u->x = x;
  u->p = rand();
  if (BinarySearchTree<Node,T>::add(u)) {
    bubbleUp(u);
    return true;
  }
  return false;
}
void bubbleUp(Node *u) {
  while (u->parent != nil && u->parent->p > u->p) {
    if (u->parent->right == u) {
      rotateLeft(u->parent);
    } else {
      rotateRight(u->parent);
    }
  }
  if (u->parent == nil) {
    r = u;
  }
}
```

An example of an add(x) operation is shown in Figure 7.6.

The running time of the add(x) operation is given by the time it takes to follow the search path for x plus the number of rotations performed to move the newly-added node, u, up to its correct location in the Treap. By Lemma 7.2, the expected length of the search path is at most $2\ln n + O(1)$. Furthermore, each rotation decreases the depth of u. This stops if u becomes the root, so the expected number of rotations cannot exceed the expected length of the search path. Therefore, the expected running time of the add(x) operation in a Treap is $O(\log n)$. (Exercise 7.3 asks you to show that the expected number of rotations performed during an insertion is actually only $O(1)$.)

The remove(x) operation in a Treap is the opposite of the add(x) operation. We search for the node, u, containing x and then perform rotations to move u downwards until

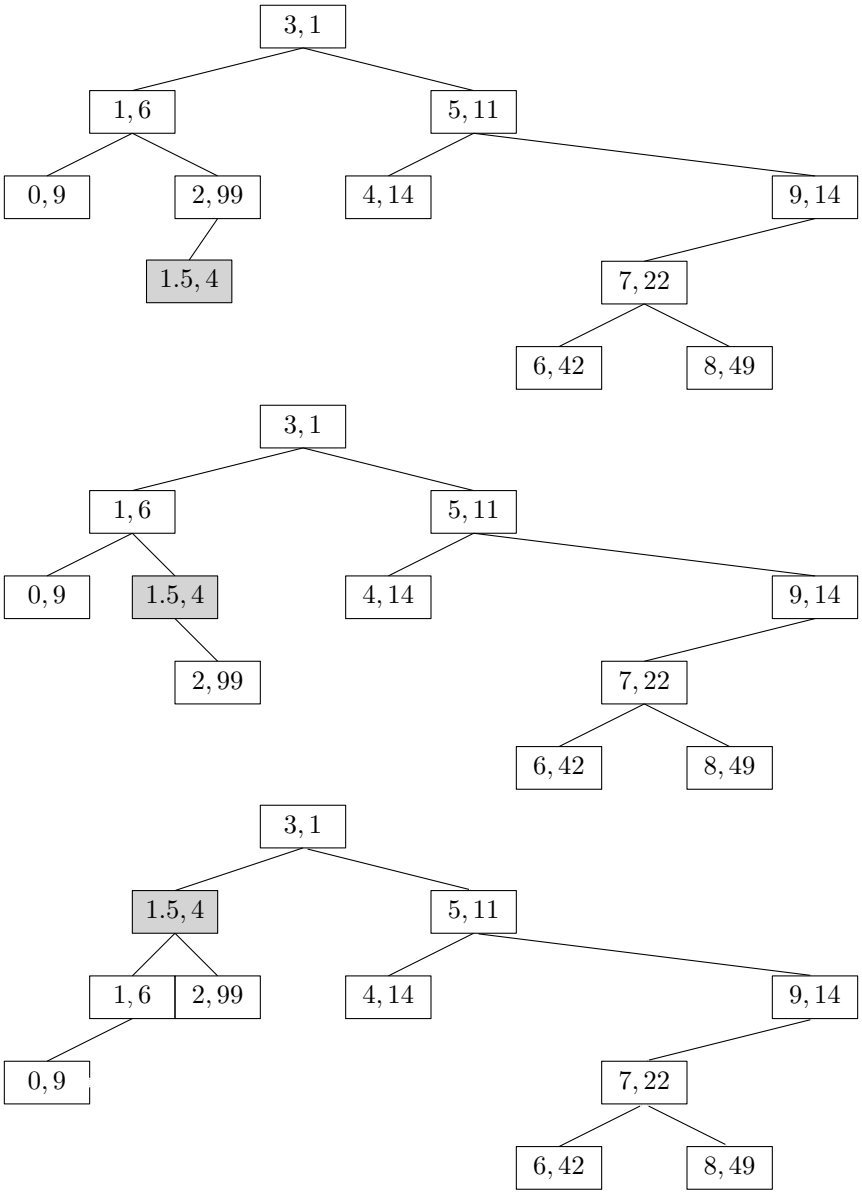


Figure 7.6: Adding the value 1.5 into the `Treap` from Figure 7.4.

it becomes a leaf and then we splice u from the Treap. Notice that, to move u downwards, we can perform either a left or right rotation at u, which will replace u with u.right or u.left, respectively. The choice is made by the first of the following that apply:

1. If u.left and u.right are both null, then u is a leaf and no rotation is performed.
2. If u.left (or u.right) is null, then perform a right (or left, respectively) rotation at u.
3. If u.left.p < u.right.p (or u.left.p > u.right.p), then perform a right rotation (or left rotation, respectively) at u.

These three rules ensure that the Treap doesn't become disconnected and that the heap property is maintained once u is removed.

Treap

```
bool remove(T x) {
  Node *u = findLast(x);
  if (u != nil && compare(u->x, x) == 0) {
    trickleDown(u);
    splice(u);
    delete u;
    return true;
  }
  return false;
}
void trickleDown(Node *u) {
  while (u->left != nil || u->right != nil) {
    if (u->left == nil) {
      rotateLeft(u);
    } else if (u->right == nil) {
      rotateRight(u);
    } else if (u->left->p < u->right->p) {
      rotateRight(u);
    } else {
      rotateLeft(u);
    }
    if (r == u) {
      r = u->parent;
    }
  }
}
```

An example of the remove(x) operation is shown in Figure 7.7.

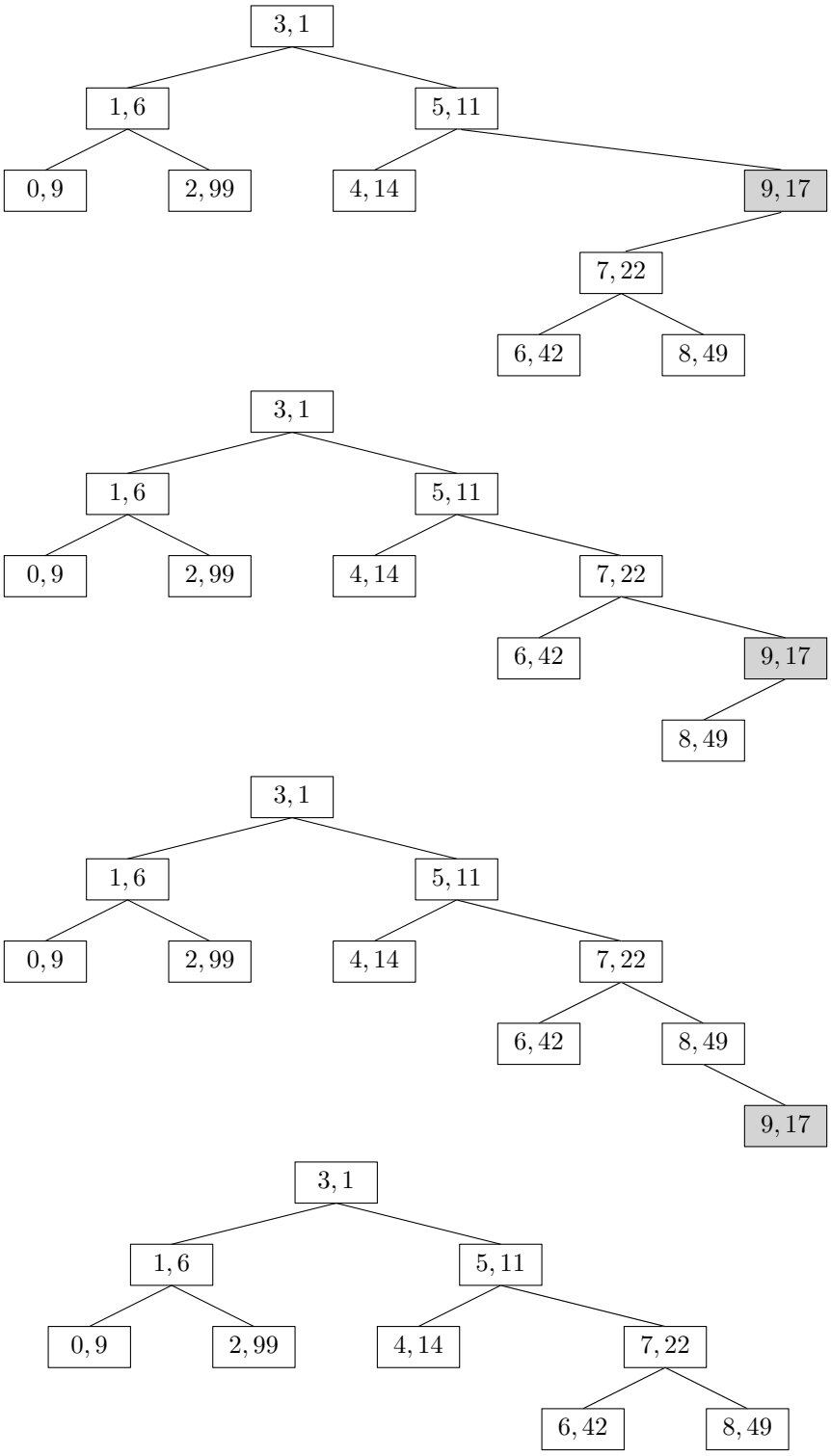


Figure 7.7: Removing the value 9 from the `Treap` in Figure 7.4.

The trick to analyze the running time of the `remove(x)` operation is to notice that this operation is the reverse of the `add(x)` operation. In particular, if we were to reinsert `x`, using the same priority `u.p`, then the `add(x)` operation would do exactly the same number of rotations and would restore the `Treap` to exactly the same state it was in before the `remove(x)` operation took place. (Reading from bottom-to-top, Figure 7.7 illustrates the insertion of the value 9 into a `Treap`.) This means that the expected running time of the `remove(x)` on a `Treap` of size `n` is proportional to the expected running time of the `add(x)` operation on a `Treap` of size $\texttt{n}-1$. We conclude that the expected running time of `remove(x)` is $O(\log \texttt{n})$.

7.2.1 Summary

The following theorem summarizes the performance of the `Treap` data structure:

Theorem 7.2. *A* `Treap` *implements the* `SSet` *interface. A* `Treap` *supports the operations* `add(x)`*,* `remove(x)`*, and* `find(x)` *in* $O(\log \texttt{n})$ *expected time per operation.*

It is worth comparing the `Treap` data structure to the `SkiplistSSet` data structure. Both implement the `SSet` operations in $O(\log \texttt{n})$ expected time per operation. In both data structures, `add(x)` and `remove(x)` involve a search and then a constant number of pointer changes (see Exercise 7.3 below). Thus, for both these structures, the expected length of the search path is the critical value in assessing their performance. In a `SkiplistSSet`, the expected length of a search path is

$$2\log \texttt{n} + O(1) \ ,$$

In a `Treap`, the expected length of a search path is

$$2\ln \texttt{n} + O(1) \approx 1.386\log \texttt{n} + O(1) \ .$$

Thus, the search paths in a `Treap` are considerably shorter and this translates into noticeably faster operations on `Treap`s than `Skiplist`s. Exercise 4.2 in Chapter 4 shows how the expected length of the search path in a `Skiplist` can be reduced to

$$e\ln \texttt{n} + O(1) \approx 1.884\log \texttt{n} + O(1)$$

by using biased coin tosses. Even with this optimization, the expected length of search paths in a `SkiplistSSet` is noticeably longer than in a `Treap`.

7.3 Summary and Exercises

Random binary search trees have been studied extensively. Devroye [14] gives a proof of Lemma 7.1 and related results. There are much stronger results in the literature as well. The most impressive of which is due to Reed [47], who shows that the expected height of a random binary search tree is

$$\alpha \ln n - \beta \ln \ln n + O(1)$$

where $\alpha \approx 4.31107$ is the unique solution on $[2, \infty)$ of the equation $\alpha \ln((2e/\alpha)) = 1$ and $\beta = \frac{3}{2\ln(\alpha/2)}$. Furthermore, the variance of the height is constant.

The name `Treap` was coined by Aragon and Seidel [50] who discussed `Treap`s and some of their variants. However, their basic structure was studied much earlier by Vuillemin [55] who called them Cartesian trees.

One space-optimization of the `Treap` data structure that is sometimes performed is the elimination of the explicit storage of the priority `p` in each node. Instead, the priority of a node, `u`, is computed by hashing `u`'s address in memory. Although a number of hash functions will probably work well for this in practice, for the important parts of the proof of Lemma 7.1 to remain valid, the hash function should be randomized and have the *min-wise independent property*: For any distinct values $x_1, \ldots, x_k$, each of the hash values $h(x_1), \ldots, h(x_k)$ should be distinct with high probability and, for each $i \in \{1, \ldots, k\}$,

$$\Pr\{h(x_i) = \min\{h(x_1), \ldots, h(x_k)\}\} \leq c/k$$

for some constant c. One such class of hash functions that is easy to implement and fairly fast is *tabulation hashing* [43].

Exercise 7.1. Prove the assertion that there are $21,964,800$ sequences that generate the tree on the right hand side of Figure 7.1. (Hint: Give a recursive formula for the number of sequences that generate a complete binary tree of height h and evaluate this formula for $h = 3$.)

Exercise 7.2. Design and implement the `permute(a)` method that takes as input an array, `a`, containing `n` distinct values and randomly permutes `a`. The method should run in $O(n)$ time and you should prove that each of the `n`! possible permutations of `a` is equally probable.

Exercise 7.3. Use both parts of Lemma 7.2 to prove that the expected number of rotations performed by an `add(x)` operation (and hence also a `remove(x)` operation) is $O(1)$.

Exercise 7.4. Design and implement a version of a `Treap` that includes a `get(i)` operation that returns the key with rank `i` in the `Treap`. (Hint: Have each node, `u`, keep track of the size of the subtree rooted at `u`.)

Exercise 7.5. Design and implement a version of a `Treap` that supports the `split(x)` operation. This operation removes all values from the `Treap` that are greater than `x` and returns a second `Treap` that contains all the removed values.

For example, the code `t2 = t.split(x)` removes from `t` all values greater than `x` and returns a new `Treap t2` containing all these values. The `split(x)` operation should run in $O(\log \mathtt{n})$ expected time.

Chapter 8

Scapegoat Trees

In this chapter, we study a binary search tree data structure, the `ScapegoatTree`, that keeps itself balanced by *partial rebuilding operations*. During a partial rebuilding operation, an entire subtree is deconstructed and rebuilt into a perfectly balanced subtree.

There are many ways of rebuilding a subtree rooted at node `u` into a perfectly balanced tree. One of the simplest is to traverse `u`'s subtree, gathering all its nodes into an array `a` and then to recursively build a balanced subtree using `a`. If we let $\texttt{m} = \texttt{a.length}/2$, then the element $\texttt{a}[\texttt{m}]$ becomes the root of the new subtree, $\texttt{a}[0], \ldots, \texttt{a}[\texttt{m}-1]$ get stored recursively in the left subtree and $\texttt{a}[\texttt{m}+1], \ldots, \texttt{a}[\texttt{a.length}-1]$ get stored recursively in the right subtree.

ScapegoatTree

```
void rebuild(Node *u) {
  int ns = BinaryTree<Node>::size(u);
  Node *p = u->parent;
  Node **a = new Node*[ns];
  packIntoArray(u, a, 0);
  if (p == nil) {
    r = buildBalanced(a, 0, ns);
    r->parent = nil;
  } else if (p->right == u) {
    p->right = buildBalanced(a, 0, ns);
    p->right->parent = p;
  } else {
    p->left = buildBalanced(a, 0, ns);
    p->left->parent = p;
  }
  delete[] a;
}
int packIntoArray(Node *u, Node **a, int i) {
  if (u == nil) {
    return i;
```

```
    }
    i = packIntoArray(u->left, a, i);
    a[i++] = u;
    return packIntoArray(u->right, a, i);
  }
```

A call to `rebuild(u)` takes $O(\texttt{size(u)})$ time. The subtree built by `rebuild(u)` has minimum height; there is no tree of smaller height that has `size(u)` nodes.

8.1 ScapegoatTree: A Binary Search Tree with Partial Rebuilding

A `ScapegoatTree` is a `BinarySearchTree` that, in addition to keeping track of the number, `n`, of nodes in the tree also keeps a counter, `q`, that maintains an upper-bound on the number of nodes.

ScapegoatTree

```
int q;
```

At all times, `n` and `q` obey the following inequalities:

$$\texttt{q}/2 \leq \texttt{n} \leq \texttt{q} \ .$$

In addition, a `ScapegoatTree` has logarithmic height; at all times, the height of the scapegoat tree does not exceed:

$$\log_{3/2} \texttt{q} \leq \log_{3/2} 2\texttt{n} < \log_{3/2} \texttt{n} + 2 \ . \tag{8.1}$$

Even with this constraint, a `ScapegoatTree` can look surprisingly unbalanced. The tree in Figure 8.1 has $\texttt{q} = \texttt{n} = 10$ and height $5 < \log_{3/2} 10 \approx 5.679$.

Implementing the `find(x)` operation in a `ScapegoatTree` is done using the standard algorithm for searching in a `BinarySearchTree` (see Section 6.2). This takes time proportional to the height of the tree which, by (8.1) is $O(\log \texttt{n})$.

To implement the `add(x)` operation, we first increment `n` and `q` and then use the usual algorithm for adding `x` to a binary search tree; we search for `x` and then add a new leaf `u` with `u.x` = `x`. At this point, we may get lucky and the depth of `u` might not exceed $\log_{3/2} \texttt{q}$. If so, then we leave well enough alone and don't do anything else.

Unfortunately, it will sometimes happen that $\texttt{depth(u)} > \log_{3/2} \texttt{q}$. In this case we need to do something to reduce the height. This isn't a big job; there is only one node, namely `u`, whose depth exceeds $\log_{3/2} \texttt{q}$. To fix `u`, we walk from `u` back up to the root

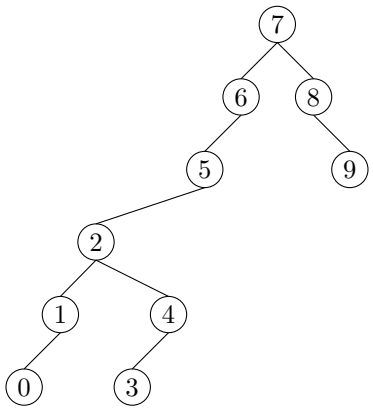


Figure 8.1: A ScapegoatTree with 10 nodes and height 5.

looking for a *scapegoat*, w. The scapegoat, w, is a very unbalanced node. It has the property that

$$\frac{\texttt{size(w.child)}}{\texttt{size(w)}} > \frac{2}{3} \ , \tag{8.2}$$

where w.child is the child of w on the path from the root to u. We'll very shortly prove that a scapegoat exists. For now, we can take it for granted. Once we've found the scapegoat w, we completely destroy the subtree rooted at w and rebuild it into a perfectly balanced binary search tree. We know, from (8.2), that, even before the addition of u, w's subtree was not a complete binary tree. Therefore, when we rebuild w, the height decreases by at least 1 so that height of the ScapegoatTree is once again at most $\log_{3/2} \texttt{q}$.

ScapegoatTree

```
bool add(T x) {
  // first do basic insertion keeping track of depth
  Node *u = new Node;
  u->x = x;
  u->left = u->right = u->parent = nil;
  int d = addWithDepth(u);
  if (d > log32(q)) {
    // depth exceeded, find scapegoat
    Node *w = u->parent;
    int a = BinaryTree<Node>::size(w);
    int b = BinaryTree<Node>::size(w->parent);
    while (3*a <= 2*b) {
      w = w->parent;
      a = BinaryTree<Node>::size(w);
```

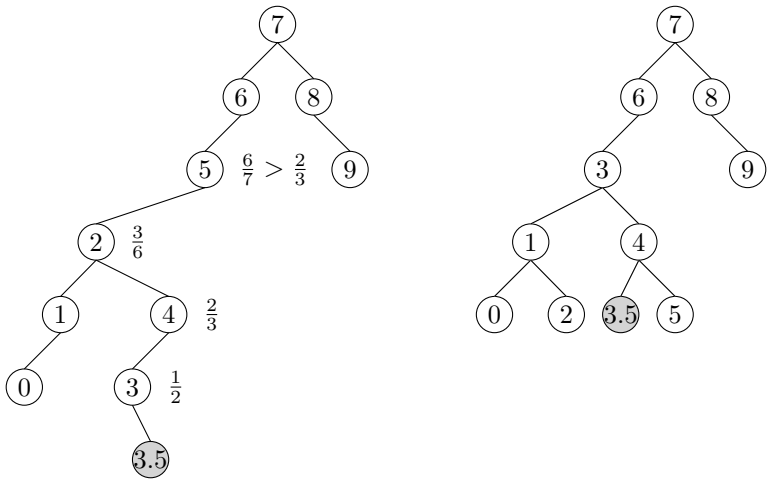


Figure 8.2: Inserting 3.5 into a ScapegoatTree increases its depth to 6, which violates (8.1) since $6 > \log_{3/2} 11 \approx 5.914$. A scapegoat is found at the node containing 5.

```
      b = BinaryTree<Node>::size(w->parent);
    }
    rebuild(w->parent);
  }
  return d >= 0;
}
```

If we ignore the cost of finding the scapegoat `w` and rebuilding the subtree rooted at `w`, then the running time of `add(x)` is dominated by the initial search, which takes $O(\log \mathtt{q}) = O(\log \mathtt{n})$ time. We will account for the cost of finding the scapegoat and rebuilding using amortized analysis in the next section.

The implementation of `remove(x)` in a `ScapegoatTree` is very simple. We search for `x` and remove it using the usual algorithm for removing a node from a `BinarySearchTree`. (Note that this can never increase the height of the tree.) Next, we decrement `n` but leave `q` unchanged. Finally, we check if $\mathtt{q} > 2\mathtt{n}$ and, if so, we *rebuild the entire tree* into a perfectly balanced binary search tree and set $\mathtt{q} = \mathtt{n}$.

ScapegoatTree

```
bool remove(T x) {
  if (BinarySearchTree<Node,T>::remove(x)) {
    if (2*n < q) {
      rebuild(r);
      q = n;
    }
```

```
      return true;
    }
    return false;
  }
```

Again, if we ignore the cost of rebuilding, the running time of the `remove(x)` operation is proportional to the height of the tree, and is therefore $O(\log \mathtt{n})$.

8.1.1 Analysis of Correctness and Running-Time

In this section we analyze the correctness and amortized running time of operations on a `ScapegoatTree`. We first prove the correctness by showing that, when the `add(x)` operation results in a node that violates Condition (8.1), then we can always find a scapegoat:

Lemma 8.1. *Let* $\mathtt{u}$ *be a node of depth* $h > \log_{3/2} \mathtt{q}$ *in a* `ScapegoatTree`*. Then there exists a node* $\mathtt{w}$ *on the path from* $\mathtt{u}$ *to the root such that*

$$\frac{\mathtt{size}(\mathtt{w})}{\mathtt{size}(\mathtt{parent}(\mathtt{w}))} > 2/3 \ .$$

Proof. Suppose, for the sake of contradiction, that this is not the case, and

$$\frac{\mathtt{size}(\mathtt{w})}{\mathtt{size}(\mathtt{parent}(\mathtt{w}))} \leq 2/3 \ .$$

for all nodes $\mathtt{w}$ on the path from $\mathtt{u}$ to the root. Denote the path from the root to $\mathtt{u}$ as $\mathtt{r} = \mathtt{u}_0, \ldots, \mathtt{u}_h = \mathtt{u}$. Then, we have $\mathtt{size}(\mathtt{u}_0) = \mathtt{n}$, $\mathtt{size}(\mathtt{u}_1) \leq \frac{2}{3}\mathtt{n}$, $\mathtt{size}(\mathtt{u}_2) \leq \frac{4}{9}\mathtt{n}$ and, more generally,

$$\mathtt{size}(\mathtt{u}_i) \leq \left(\frac{2}{3}\right)^i \mathtt{n} \ .$$

But this gives a contradiction, since $\mathtt{size}(\mathtt{u}) \geq 1$, hence

$$1 \leq \mathtt{size}(\mathtt{u}) \leq \left(\frac{2}{3}\right)^h \mathtt{n} < \left(\frac{2}{3}\right)^{\log_{3/2} \mathtt{q}} \mathtt{n} \leq \left(\frac{2}{3}\right)^{\log_{3/2} \mathtt{n}} \mathtt{n} = \left(\frac{1}{\mathtt{n}}\right)\mathtt{n} = 1 \ . \qquad \square$$

Next, we analyze the parts of the running time that we have not yet accounted for. There are two parts: The cost of calls to `size(u)` when search for scapegoat nodes, and the cost of calls to `rebuild(w)` when we find a scapegoat `w`. The cost of calls to `size(u)` can be related to the cost of calls to `rebuild(w)`, as follows:

Lemma 8.2. *During a call to* `add(x)` *in a* `ScapegoatTree`*, the cost of finding the scapegoat* $\mathtt{w}$ *and rebuilding the subtree rooted at* $\mathtt{w}$ *is* $O(\mathtt{size}(\mathtt{w}))$*.*

Proof. The cost of rebuilding the scapegoat node w, once we find it, is $O(\texttt{size}(\texttt{w}))$. When searching for the scapegoat node, we call size(u) on a sequence of nodes $\texttt{u}_0, \ldots, \texttt{u}_k$ until we find the scapegoat $\texttt{u}_k = \texttt{w}$. However, since $\texttt{u}_k$ is the first node in this sequence that is a scapegoat, we know that

$$\texttt{size}(\texttt{u}_i) < \frac{2}{3}\texttt{size}(\texttt{u}_{i+1})$$

for all $i \in \{0, \ldots, k-2\}$. Therefore, the cost of all calls to size(u) is

$$\begin{aligned} O\left(\sum_{i=0}^{k} \texttt{size}(\texttt{u}_{k-i})\right) &= O\left(\texttt{size}(\texttt{u}_k) + \sum_{i=0}^{k-1} \texttt{size}(\texttt{u}_{k-i-1})\right) \\ &= O\left(\texttt{size}(\texttt{u}_k) + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i \texttt{size}(\texttt{u}_k)\right) \\ &= O\left(\texttt{size}(\texttt{u}_k)\left(1 + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i\right)\right) \\ &= O(\texttt{size}(\texttt{u}_k)) = O(\texttt{size}(\texttt{w})) \ , \end{aligned}$$

where the last line follows from the fact that the sum is a geometrically decreasing series. □

All that remains is to prove an upper-bound on the cost of calls to rebuild(u):

Lemma 8.3. *Starting with an empty* ScapegoatTree *any sequence of* m add(x) *and* remove(x) *operations causes at most* $O(m \log m)$ *time to be used by* rebuild(u) *operations.*

Proof. To prove this, we will use a credit scheme. Each node stores a number of credits. Each credit can pay for some constant, c, units of time spent rebuilding. The scheme gives out a total of $O(m \log m)$ credits and every call to rebuild(u) is paid for with credits stored at u.

During an insertion or deletion, we give one credit to each node on the path to the inserted node, or deleted node, u. In this way we hand out at most $\log_{3/2} \texttt{q} \leq \log_{3/2} m$ credits per operation. During a deletion we also store an additional 1 credit "on the side." Thus, in total we give out at most $O(m \log m)$ credits. All that remains is to show that these credits are sufficient to pay for all calls to rebuild(u).

If we call rebuild(u) during an insertion, it is because u is a scapegoat. Suppose, without loss of generality, that

$$\frac{\texttt{size}(\texttt{u.left})}{\texttt{size}(\texttt{u})} > \frac{2}{3} \ .$$

Using the fact that

$$\texttt{size(u)} = 1 + \texttt{size(u.left)} + \texttt{size(u.right)}$$

we deduce that

$$\frac{1}{2}\texttt{size(u.left)} > \texttt{size(u.right)}$$

and therefore

$$\texttt{size(u.left)} - \texttt{size(u.right)} > \frac{1}{2}\texttt{size(u.left)} > \frac{1}{3}\texttt{size(u)} \ .$$

Now, the last time a subtree containing u was rebuilt (or when u was inserted, if a subtree containing u was never rebuilt), we had

$$\texttt{size(u.left)} - \texttt{size(u.right)} \leq 1 \ .$$

Therefore, the number of add(x) or remove(x) operations that have affected u.left or u.right since then is at least

$$\frac{1}{3}\texttt{size(u)} - 1 \ .$$

and there are therefore at least this many credits stored at u that are available to pay for the $O(\texttt{size(u)})$ time it takes to call rebuild(u).

If we call rebuild(u) during a deletion, it is because $\texttt{q} > 2\texttt{n}$. In this case, we have $\texttt{q} - \texttt{n} > \texttt{n}$ credits stored "on the side" and we use these to pay for the $O(\texttt{n})$ time it takes to rebuild the root. This completes the proof. □

8.1.2 Summary

The following theorem summarizes the performance of the ScapegoatTree data structure:

Theorem 8.1. *A* ScapegoatTree *implements the* SSet *interface. Ignoring the cost of* rebuild(u) *operations, a* ScapegoatTree *supports the operations* add(x), remove(x), *and* find(x) *in* $O(\log \texttt{n})$ *time per operation.*

Furthermore, beginning with an empty ScapegoatTree, *any sequence of* m add(x) *and* remove(x) *operations results in a total of* $O(m \log m)$ *time spent during all calls to* rebuild(u).

8.2 Discussion and Exercises

The term *scapegoat tree* is due to Galperin and Rivest [27], who define and analyze these trees. However, the same structure was discovered earlier by Andersson [3, 5], who called them *general balanced trees* since they can have any shape as long as their height is small.

Experimenting with the `ScapegoatTree` implementation will reveal that it is often considerably slower than the other `SSet` implementations in this book. This may be somewhat surprising, since height bound of

$$\log_{3/2} \mathtt{q} \approx 1.709 \log \mathtt{n} + O(1)$$

is better than the expected length of a search path in a `Skiplist` and not too far from that of a `Treap`. The implementation could be optimized by storing the sizes of subtrees explicitly at each node (or at least reusing already computed subtree sizes). Even with these optimizations, there will always be sequences of `add(x)` and `delete(x)` operation for which a `ScapegoatTree` takes longer than other `SSet` implementations.

This gap in performance is due to the fact that, unlike the other `SSet` implementations discussed in this book, a `ScapegoatTree` can spend a lot of time restructuring itself. Exercise 8.1 asks you to prove that there are sequences of `n` operations in which a `ScapegoatTree` will spend on the order of $\mathtt{n} \log \mathtt{n}$ time in calls to `rebuild(u)`. This is in contrast to other `SSet` implementations discussed in this book that only make $O(\mathtt{n})$ structural changes during a sequence of `n` operations. This is, unfortunately, a necessary consequence of the fact that a `ScapegoatTree` does all its restructuring by calls to `rebuild(u)` [15].

Despite their lack of performance, there are applications in which a `ScapegoatTree` could be the right choice. This would occur any time there is additional data associated with nodes that cannot be updated in constant time when a rotation is performed, but that can be updated during a `rebuild(u)` operation. In such cases, the `ScapegoatTree` and related structures based on partial rebuilding may work. An example of such an application is outlined in Exercise 8.5.

Exercise 8.1. Show that, if we start with an empty `ScapegoatTree` and call `add(x)` for $\mathtt{x} = 1, 2, 3, \ldots, \mathtt{n}$, then the total time spent during calls to `rebuild(u)` is at least $c\mathtt{n} \log \mathtt{n}$ for some constant $c > 0$.

Exercise 8.2. The `ScapegoatTree`, as described in this chapter guarantees that the length of the search path does not exceed $\log_{3/2} \mathtt{q}$. Design, analyze, and implement a modified version of `ScapegoatTree` where the length of the search path does not exceed $\log_{\mathtt{b}} \mathtt{q}$, where `b` is a parameter with $1 < \mathtt{b} < 2$.

What does your analysis and/or experiments say about the amortized cost of `find(x)`, `add(x)` and `remove(x)` as a function of `b`?

Exercise 8.3. Analyze and implement a `WeightBalancedTree`. This is a tree in which each node `u`, except the root, maintains the *balance invariant* that $\mathtt{size(u)} \leq (2/3)\mathtt{size(u.parent)}$.

The `add(x)` and `remove(x)` operations are identical to the standard `BinarySearchTree` operations, except that any time the balance invariant is violated at a node `u`, the subtree rooted at `u.parent` is rebuilt.

Your analysis should show that operations on a `WeightBalancedTree` run in $O(\log n)$ amortized time.

Exercise 8.4. Analyze and implement a `CountdownTree`. In a `CountdownTree` each node `u` keeps a *timer* `u.t`. The `add(x)` and `remove(x)` operations are exactly the same as in a standard `BinarySearchTree` except that, whenever one of these operations affects `u`'s subtree, `u.t` is decremented. When `u.t` $= 0$ the entire subtree rooted at `u` is rebuilt into a perfectly balanced binary search tree. When a node `u` is involved in a rebuilding operation (either because `u` is rebuilt or one of `u`'s ancestors is rebuilt) `u.t` is reset to `size(u)`$/3$.

Your analysis should show that operations on a countdown tree run in $O(\log n)$ amortized time. (Hint: First show that each node `u` satisfies some version of a balance invariant.)

Exercise 8.5. Design and implement a `Sequence` data structure that maintains a sequence (list) of elements. It supports these operations:

- `addAfter(e)`: Add a new element after the element `e` in the sequence. Return the newly added element. (If `e` is null, the new element is added at the beginning of the sequence.)
- `remove(e)`: Remove `e` from the sequence.
- `testBefore(e1, e2)`: return `true` if and only if `e1` comes before `e2` in the sequence.

The first two operations should run in $O(\log n)$ amortized time. The third operation should run in constant-time.

The `Sequence` data structure can be implemented by storing the elements in something like a `ScapegoatTree`, in the same order that they occur in the sequence. To implement `testBefore(e1, e2)` in constant time, each element `e` is labelled with an integer that encodes the path from the root to `e`. In this way, `testBefore(e1, e2)` can be implemented just by comparing the labels of `e1` and `e2`.

Chapter 9

Red-Black Trees

In this chapter, we present red-black trees, a version of binary search trees that have logarithmic depth. Red-black trees are one of the most widely-used data structures in practice. They appear as the primary search structure in many library implementations, including the Java Collections Framework and several implementations of the C++ Standard Template Library. They are also used within the Linux operating system kernel. There are several reasons for the popularity of red-black trees:

1. A red-black tree storing `n` values has height at most $2\log \texttt{n}$.

2. The `add(x)` and `remove(x)` operations on a red-black tree run in $O(\log \texttt{n})$ *worst-case* time.

3. The amortized number of rotations done during an `add(x)` or `remove(x)` operation is constant.

The first two of these properties already put red-black trees ahead of skiplists, treaps, and scapegoat trees. Skiplists and treaps rely on randomization and their $O(\log \texttt{n})$ running times are only expected. Scapegoat trees have a guaranteed bound on their height, but `add(x)` and `remove(x)` only run in $O(\log \texttt{n})$ amortized time. The third property is just icing on the cake. It tells us that that the time needed to add or remove an element `x` is dwarfed by the time it takes to find `x`.[1]

However, the nice properties of red-black trees come with a price: implementation complexity. Maintaining a bound of $2\log \texttt{n}$ on the height is not easy. It requires a careful analysis of a number of cases and it requires that the implementation does exactly the right

[1]Note that skiplists and treaps also have this property in the expected sense. See Exercise 4.1 and Exercise 7.3.

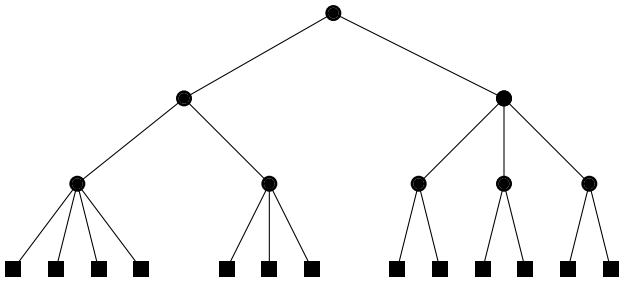


Figure 9.1: A 2-4 tree of height 3.

thing in each case. One misplaced rotation or change of color produces a bug that can be very difficult to understand and track down.

Rather than jumping directly into the implementation of red-black trees, we will first provide some background on a related data structure: 2-4 trees. This will give some insight into how red-black trees were discovered and why efficiently maintaining red-black trees is even possible.

9.1 2-4 Trees

A 2-4 tree is a rooted tree with the following properties:

Property 9.1 (height). All leaves have the same depth.

Property 9.2 (degree). Every internal node has 2, 3, or 4 children.

An example of a 2-4 tree is shown in Figure 9.1. The properties of 2-4 trees imply that their height is logarithmic in the number of leaves:

Lemma 9.1. *A 2-4 tree with* $\mathtt{n}$ *leaves has height at most* $\log \mathtt{n}$.

Proof. The lower-bound of 2 on the degree of an internal node implies that, if the height of a 2-4 tree is h, then it has at least 2^h leaves. In other words,

$$\mathtt{n} \geq 2^h \ .$$

Taking logarithms on both sides of this equation gives $h \leq \log \mathtt{n}$. □

9.1.1 Adding a Leaf

Adding a leaf to a 2-4 tree is easy (see Figure 9.2). If we want to add a leaf `u` as the child of some node `w` on the second-last level, we simply make `u` a child of `w`. This certainly

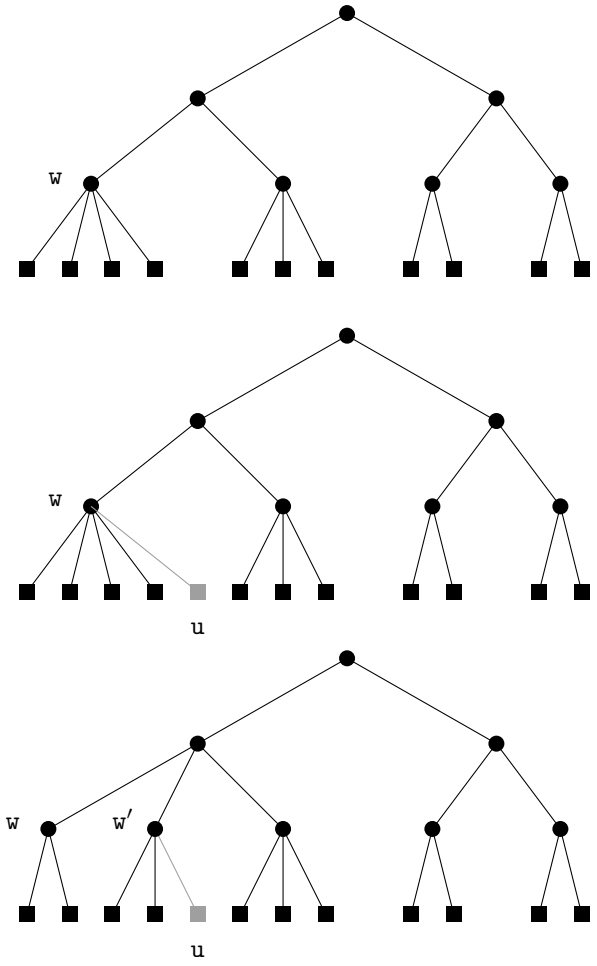


Figure 9.2: Adding a leaf to a 2-4 Tree. This process stops after one split because `w.parent` has degree less than 4 before the addition.

maintains the height property, but could violate the degree property; if `w` had 4 children prior to adding `u`, then `w` now has 5 children. In this case, we split `w` into two nodes, `w` and `w'`, having 2 and 3 children, respectively. But now `w'` has no parent, so we recursively make `w'` a child of `w`'s parent. Again, this may cause `w`'s parent to have too many children in which case we split it. This process goes on until we reach a node that has fewer than 4 children, or until we split the root, `r`, into two nodes `r` and $\texttt{r}'$. In the latter case, we make a new root that has `r` and $\texttt{r}'$ as children. This simultaneously increases the depth of all leaves and so maintains the height property.

Since the height of the 2-4 tree is never more than $\log \texttt{n}$, the process of adding a leaf finishes after at most $\log \texttt{n}$ steps.

9.1.2 Removing a Leaf

Removing a leaf from a 2-4 tree is a little more tricky (see Figure 9.3). To remove a leaf u from its parent w, we just remove it. If w had only two children prior to the removal of u, then w is left with only one child and violates the degree property.

To correct this, we look at w's sibling, w′. The node w′ is sure to exist since w's parent has at least 2 children. If w′ has 3 or 4 children, then we take one of these children from w′ and give it to w. Now w has 2 children and w′ has 2 or 3 children and we are done.

On the other hand, if w′ has only two children, then we merge w and w′ into a single node, w, that has 3 children. Next we recursively remove w′ from the parent of w′. This process ends when we reach a node, u, where u or its sibling has more than 2 children; or we reach the root. In the latter case, if the root is left with only 1 child, then we delete the root and make its child the new root. Again, this simultaneously decreases the height of every leaf and therefore maintains the height property.

Again, since the height of the tree is never more than $\log n$, the process of removing a leaf finishes after at most $\log n$ steps.

9.2 RedBlackTree: A Simulated 2-4 Tree

A red-black tree is a binary search tree in which each node, u, has a *color* which is either *red* or *black*. Red is represented by the value 0 and black by the value 1.

RedBlackTree

```
class RedBlackNode : public BSTNode<Node, T> {
  friend class RedBlackTree<Node, T>;
  char color;
};
int red = 0;
int black = 1;
```

Before and after any operation on a red-black tree, the following two properties are satisfied. Each property is defined both in terms of the colors red and black, and in terms of the numeric values 0 and 1.

Property 9.3 (black-height). There are the same number of black nodes on every root to leaf path. (The sum of the colors on any root to leaf path is the same.)

Property 9.4 (no-red-edge). No two red nodes are adjacent. (For any node u, except the root, u.color + u.parent.color ≥ 1.)

Notice that we can always color the root, r, of a red-black tree black without violating either of these two properties, so we will assume that the root is black, and the

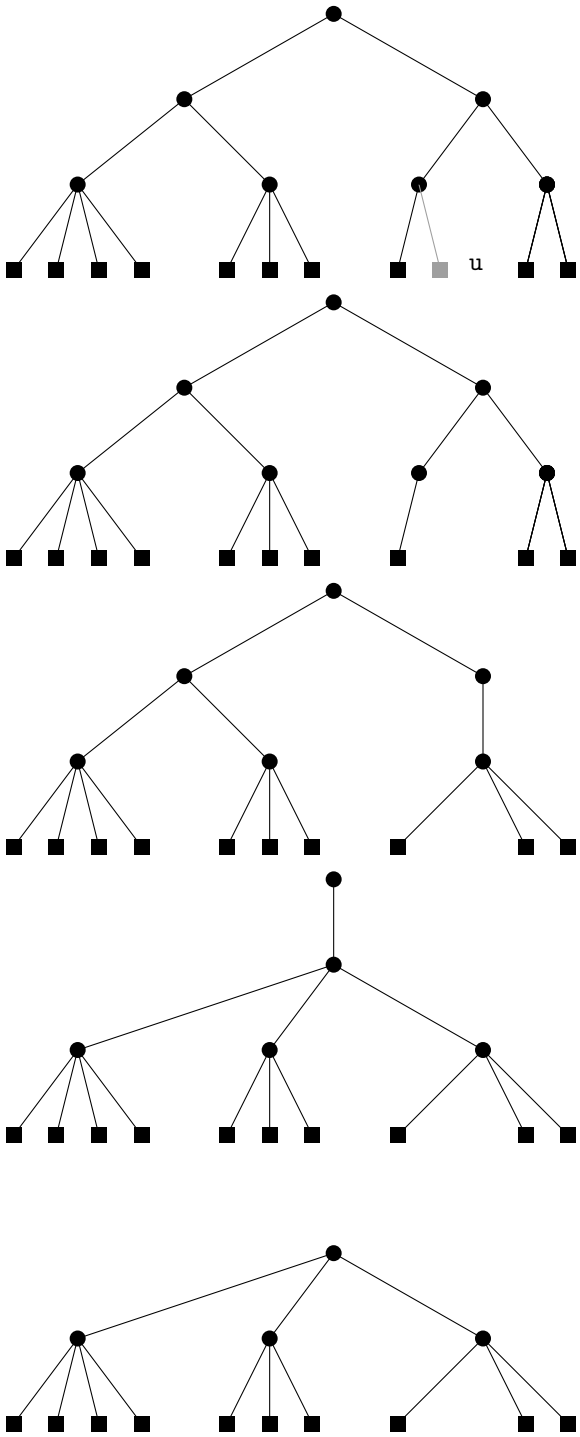


Figure 9.3: Removing a leaf from a 2-4 Tree. This process goes all the way to the root because all of u's ancestors and their siblings have degree 2.

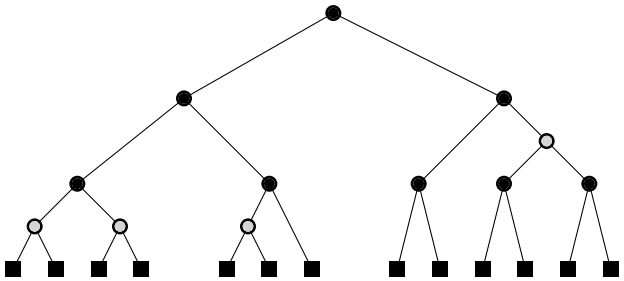


Figure 9.4: An example of a red-black tree with black-height 3. External (nil) nodes are drawn as squares.

algorithms for updating a red-black tree will maintain this. Another trick that simplifies red-black trees is to treat the external nodes (represented by nil) as black nodes. This way, every real node, u, of a red-black tree has exactly two children, each with a well-defined color. An example of a red-black tree is shown in Figure 9.4.

9.2.1 Red-Black Trees and 2-4 Trees

At first it might seem surprising that a red-black tree can be efficiently updated to maintain the black-height and no-red-edge properties, and it seems unusual to even consider these as useful properties. However, red-black trees were designed to be an efficient simulation of 2-4 trees as binary trees.

Refer to Figure 9.5. Consider any red-black tree, T, having n nodes and perform the following transformation: Remove each red node u and connect u's two children directly to the (black) parent of u. After this transformation we are left with a tree T' having only black nodes.

Every internal node in T' has 2, 3, or 4 children: A black node that started out with two black children will still have two black children after this transformation. A black node that started out with one red and one black child will have three children after this transformation. A black node that started out with two red children will have 4 children after this transformation. Furthermore, the black-height property now guarantees that every root-to-leaf path in T' has the same length. In other words, T' is a 2-4 tree!

The 2-4 tree T' has $n + 1$ leaves that correspond to the $n + 1$ external nodes of the red-black tree. Therefore, this tree has height $\log(n + 1)$. Now, every root to leaf path in the 2-4 tree corresponds to a path from the root of the red-black tree T to an external node. The first and last node in this path are black and at most one out of every two internal

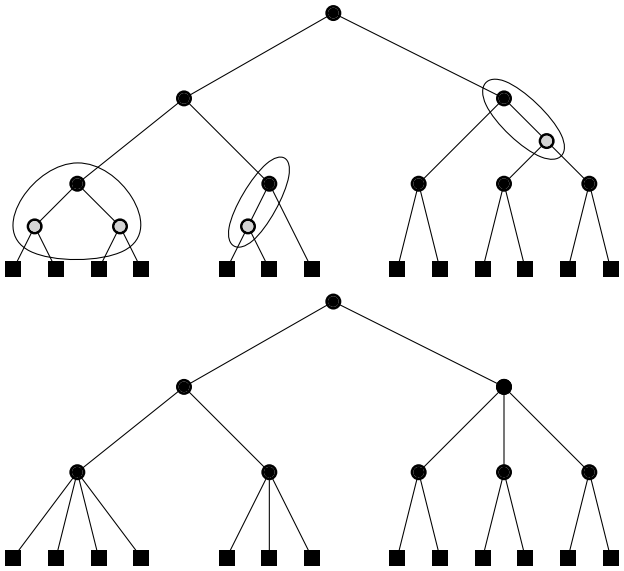


Figure 9.5: Every red-black tree has a corresponding 2-4 tree.

nodes is red, so this path has at most $\log(\mathtt{n}+1)$ black nodes and at most $\log(\mathtt{n}+1)-1$ red nodes. Therefore, the longest path from the root to any *internal* node in T is at most

$$2\log(\mathtt{n}+1) - 2 \leq 2\log \mathtt{n} \ ,$$

for any $\mathtt{n} \geq 1$. This proves the most important property of red-black trees:

Lemma 9.2. *The height of red-black tree with* n *nodes is at most* $2\log \mathtt{n}$.

Now that we have seen the relationship between 2-4 trees and red-black trees, it is not hard to believe that we can efficiently maintain a red-black tree while adding and removing elements.

We have already seen that adding an element in a `BinarySearchTree` can be done by adding a new leaf. Therefore, to implement `add(x)` in a red-black tree we need a method of simulating splitting a degree 5 node in a 2-4 tree. A degree 5 node is represented by a black node that has two red children one of which also has a red child. We can "split" this node by coloring it red and coloring its two children black. An example of this is shown in Figure 9.6.

Similarly, implementing `remove(x)` requires a method of merging two nodes and borrowing a child from a sibling. Merging two nodes is the inverse of a split (shown in Figure 9.6), and involves coloring two (black) siblings red and coloring their (red) parent

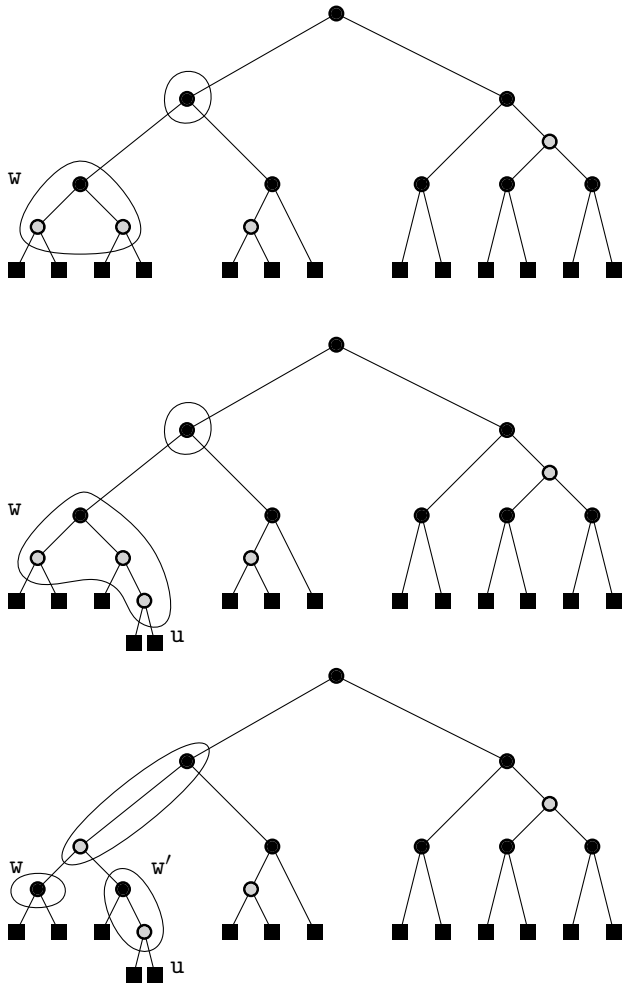


Figure 9.6: Simulating a 2-4 tree split operation during an addition in a red-black tree. (This simulates the 2-4 tree addition shown in Figure 9.2.)

black. Borrowing from a sibling is the most complicated of the procedures and involves both rotations and recoloring of nodes.

Of course, during all of this we must still maintain the no-red-edge property and the black-height property. While it is no longer surprising that this can be done, there are a large number of cases that have to be considered if we try to do a direct simulation of a 2-4 tree by a red-black tree. At some point, it just becomes simpler to forget about the underlying 2-4 tree and work directly towards maintaining the red-black tree properties.

9.2.2 Left-Leaning Red-Black Trees

There is no single definition of a red-black tree. Rather, there are a family of structures that manage to maintain the black-height and no-red-edge properties during `add(x)` and `remove(x)` operations. Different structures go about it in different ways. Here, we implement a data structure that we call a `RedBlackTree`. This structure implements a particular variant of red-black trees that satisfies an additional property:

Property 9.5 (left-leaning). At any node `u`, if `u.left` is black, then `u.right` is black.

Note that the red-black tree shown in Figure 9.4 does not satisfy the left-leaning property; it is violated by the parent of the red node in the rightmost path.

The reason for maintaining the left-leaning property is that it reduces the number of cases encountered when updating the tree during `add(x)` and `remove(x)` operations. In terms of 2-4 trees, it implies that every 2-4 tree has a unique representation: A node of degree 2 becomes a black node with 2 black children. A node of degree 3 becomes a black node whose left child is red and whose right child is black. A node of degree 4 becomes a black node with two red children.

Before we describe the implementation of `add(x)` and `remove(x)` in detail, we first present some simple subroutines used by these methods that are illustrated in Figure 9.7. The first two subroutines are for manipulating colors while preserving the black-height property. The `pushBlack(u)` method takes as input a black node `u` that has two red children and colors `u` red and its two children black. The `pullBlack(u)` method reverses this operation:

RedBlackTree

```
void pushBlack(Node *u) {
  u->color--;
  u->left->color++;
  u->right->color++;
}
void pullBlack(Node *u) {
  u->color++;
```

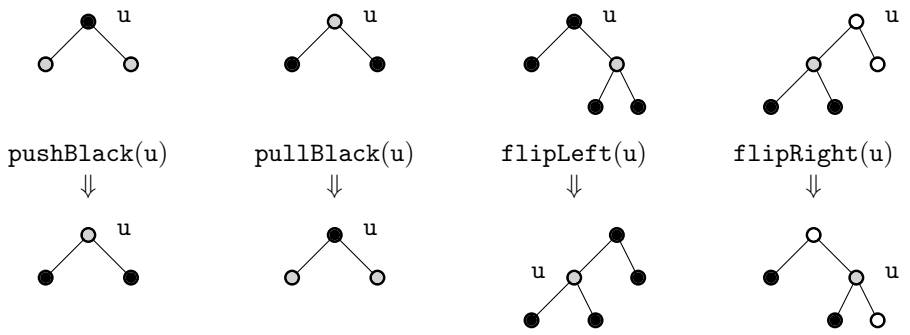


Figure 9.7: Flips, pulls and pushes

```
  u->left->color--;
  u->right->color--;
}
```

The `flipLeft(u)` method swaps the colors of `u` and `u.right` and then performs a left rotation at `u`. This reverses the colors of these two nodes as well as their parent-child relationship:

RedBlackTree

```
void flipLeft(Node *u) {
  swapColors(u, u->right);
  rotateLeft(u);
}
```

The `flipLeft(u)` operation is especially useful in restoring the left-leaning property at a node u that violates it (because `u.left` is black and `u.right` is red). In this special case, we can be assured this operation preserves both the black-height and no-red-edge properties. The `flipRight(u)` operation is symmetric to `flipLeft(u)` with the roles of left and right reversed.

RedBlackTree

```
void flipRight(Node *u) {
  swapColors(u, u->left);
  rotateRight(u);
}
```

9.2.3 Addition

To implement `add(x)` in a `RedBlackTree`, we perform a standard `BinarySearchTree` insertion, which adds a new leaf, `u`, with `u.x` = `x` and set `u.color` = `red`. Note that this does

not change the black height of any node, so it does not violate the black-height property. It may, however, violate the left-leaning property (if u is the right child of its parent) and it may violate the no-red-edge property (if u's parent is red). To restore these properties, we call the method addFixup(u).

RedBlackTree

```
bool add(T x) {
  Node *u = new Node();
  u->left = u->right = u->parent = nil;
  u->x = x;
  u->color = red;
  bool added = BinarySearchTree<Node,T>::add(u);
  if (added)
    addFixup(u);
  return added;
}
```

The addFixup(u) method, illustrated in Figure 9.8, takes as input a node u whose color is red and which may be violating the no-red-edge property and/or the left-leaning property. The following discussion is probably impossible to follow without referring to Figure 9.8 or recreating it on a piece of paper. Indeed, the reader may wish to study this figure before continuing.

If u is the root of the tree, then we can color u black and this restores both properties. If u's sibling is also red, then u's parent must be black, so both the left-leaning and no-red-edge properties already hold.

Otherwise, we first determine if u's parent, w, violates the left-leaning property and, if so, perform a flipLeft(w) operation and set u = w. This leaves us in a well-defined state: u is the left child of its parent, w, so w now satisfies the left-leaning property. All that remains is to ensure the no-red-edge property at u. We only have to worry about the case where w is red, since otherwise u already satisfies the no-red-edge property.

Since we are not done yet, u is red and w is red. The no-red-edge property (which is only violated by u and not by w) implies that u's grandparent g exists and is black. If g's right child is red, then the left-leaning property ensures that both g's children are red, and a call to pushBlack(g) makes g red and w black. This restores the no-red-edge property at u, but may cause it to be violated at g, so the whole process starts over with u = g.

If g's right child is black, then a call to flipRight(g) makes w the (black) parent of g and gives w two red children, u and g. This ensures that u satisfies the no-red-edge property and g satisfies the left-leaning property. In this case we can stop.

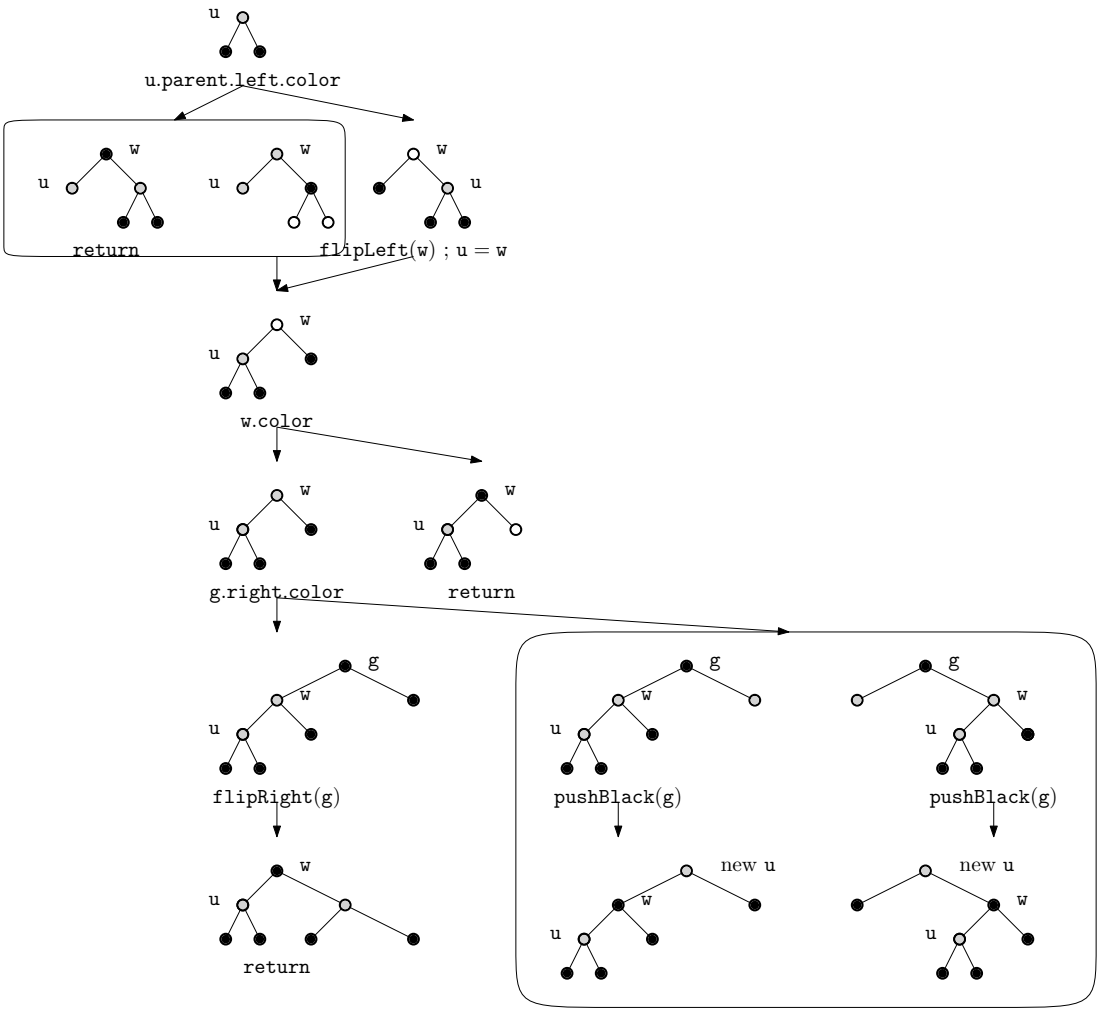


Figure 9.8: A single round in the process of fixing Property 2 after an insertion.

RedBlackTree

```
void addFixup(Node *u) {
  while (u->color == red) {
    if (u == r) { // u is the root - done
      u->color = black;
      return;
    }
    Node *w = u->parent;
    if (w->left->color == black) { // ensure left-leaning
      flipLeft(w);
      u = w;
      w = u->parent;
    }
    if (w->color == black)
      return; // no red-red edge = done
    Node *g = w->parent; // grandparent of u
    if (g->right->color == black) {
      flipRight(g);
      return;
    } else {
      pushBlack(g);
      u = g;
    }
  }
}
```

The `insertFixup(u)` method takes constant time per iteration and each iteration either finishes or moves `u` closer to the root. This implies that the `insertFixup(u)` method finishes after $O(\log n)$ iterations in $O(\log n)$ time.

9.2.4 Removal

The `remove(x)` operation in a `RedBlackTree` tree is the most complicated operation to implement, and this is true of all known implementations. Like `remove(x)` in a `BinarySearchTree`, this operation boils down to finding a node `w` with only one child, `u`, and splicing `w` out of the tree by having `w.parent` adopt `u`.

The problem with this is that, if `w` is black, then the black-height property will now be violated at `w.parent`. We get around this problem, temporarily, by adding `w.color` to `u.color`. Of course, this introduces two other problems: (1) `u` and `w` both started out black, then `u.color` + `w.color` = 2 (double black), which is an invalid color. If `w` was red, then it is replaced by a black node `u`, which may violate the left-leaning property at `u.parent`. Both of these problems are resolved with a call to the `removeFixup(u)` method.

RedBlackTree
```
bool remove(T x) {
  Node *u = findLast(x);
  if (u == nil || compare(u->x, x) != 0)
    return false;
  Node *w = u->right;
  if (w == nil) {
    w = u;
    u = w->left;
  } else {
    while (w->left != nil)
      w = w->left;
    u->x = w->x;
    u = w->right;
  }
  splice(w);
  u->color += w->color;
  u->parent = w->parent;
  delete w;
  removeFixup(u);
  return true;
}
```

The `removeFixup`(u) method takes as input a node u whose color is black (1) or double-black (2). If u is double-black, then `removeFixup`(u) performs a series of rotations and recoloring operations that move the double-black node up the tree until it can be gotten rid of. During this process, the node u changes until, at the end of this process, u refers to the root of the subtree that has been changed. The root of this subtree may have changed color. In particular, it may have gone from red to black, so the `removeFixup`(u) method finishes by checking if u's parent violates the left-leaning property and, if so, fixes it.

RedBlackTree
```
void removeFixup(Node *u) {
  while (u->color > black) {
    if (u == r) {
      u->color = black;
    } else if (u->parent->left->color == red) {
      u = removeFixupCase1(u);
    } else if (u == u->parent->left) {
      u = removeFixupCase2(u);
    } else {
      u = removeFixupCase3(u);
    }
  }
```

```
    if (u != r) { // restore left-leaning property, if necessary
      Node *w = u->parent;
      if (w->right->color == red && w->left->color == black) {
        flipLeft(w);
      }
    }
  }
```

The `removeFixup(u)` method is illustrated in Figure 9.9. Again, the following text will be very difficult, if not impossible, to follow without referring constantly to Figure 9.9. Each iteration of the loop in `removeFixup(u)` processes the double-black node `u` based on one of four cases.

Case 0: `u` is the root. This is the easiest case to treat. We recolor `u` to be black and this does not violate any of the red-black tree properties.

Case 1: `u`'s sibling, `v`, is red. In this case, `u`'s sibling is the left child of its parent, `w` (by the left-leaning property). We perform a right-flip at `w` and then proceed to the next iteration. Note that this causes `w`'s parent to violate the left-leaning property and it causes the depth of `u` to increase. However, it also implies that the next iteration will be in Case 3 with `w` colored red. When examining Case 3, below, we will see that this means the process will stop during the next iteration.

RedBlackTree

```
Node* removeFixupCase1(Node *u) {
  flipRight(u->parent);
  return u;
}
```

Case 2: `u`'s sibling, `v`, is black and `u` is the left child of its parent, `w`. In this case, we call `pullBlack(w)`, making `u` black, `v` red, and darkening the color of `w` to black or double-black. At this point, `w` does not satisfy the left-leaning property, so we call `flipLeft(w)` to fix this.

At this point, `w` is red and `v` is the root of the subtree we started with. We need to check if `w` causes no-red-edge property to be violated. We do this by inspecting `w`'s right child, `q`. If `q` is black, then `w` satisfies the no-red-edge property and we can continue to the next iteration with `u=v`.

Otherwise (`q` is red), both the no-red-edge property and the left-leaning property are violated at `q` and `w`, respectively. A call to `rotateLeft(w)` restores the left-leaning property, but the no-red-edge property is still violated. At this point, `q` is the left child of `v` and `w` is the left child of `q`, `q` and `w` are both red and `v` is black or double-black. A `flipRight(v)`

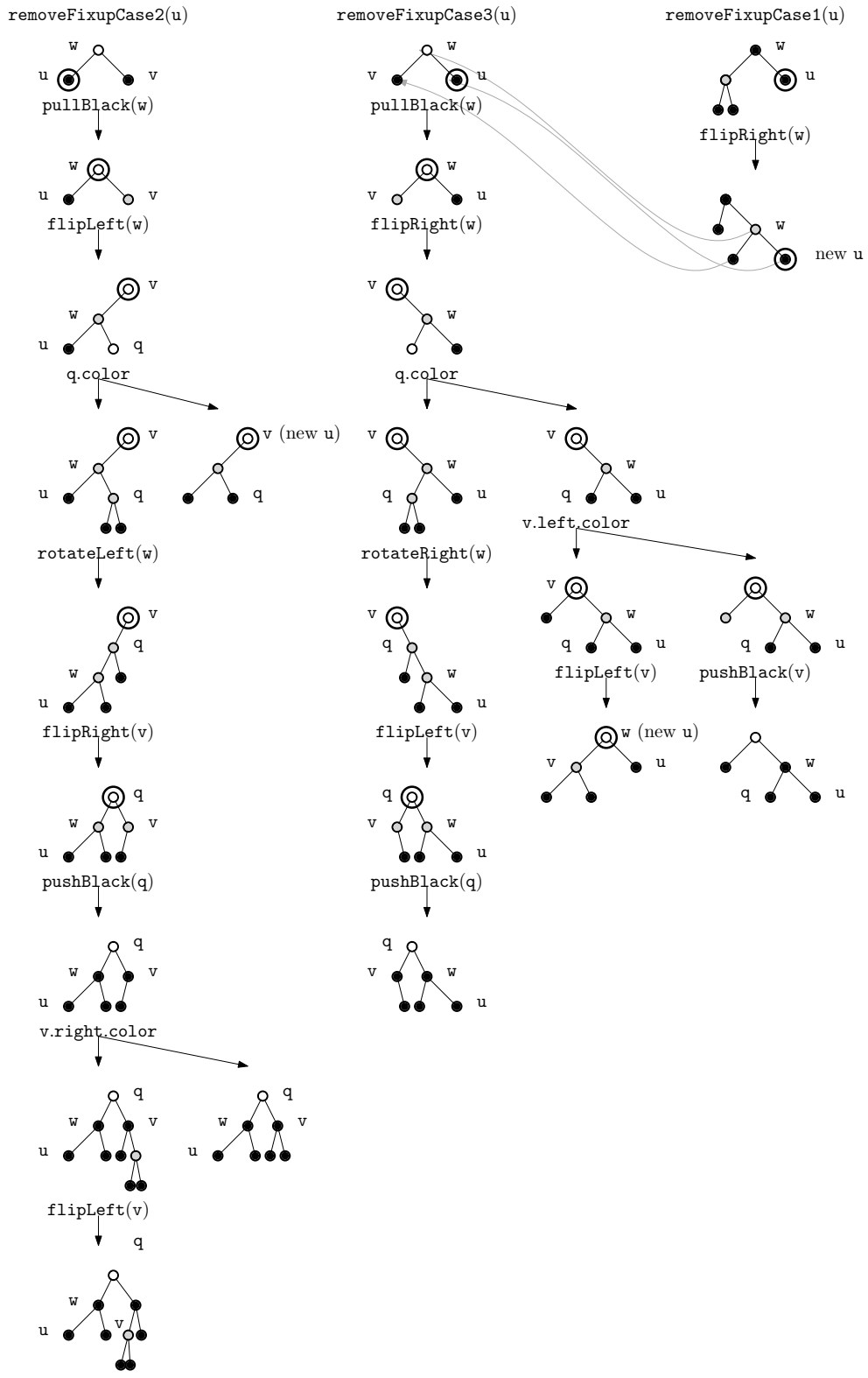


Figure 9.9: A single round in the process of eliminating a double-black node after a removal.

makes q the parent of both v and w. Following this up by a pushBlack(q) makes both v and w black and sets the color of q back to the original color of w.

At this point, there is no more double-black node and the no-red-edge and black-height properties are reestablished. The only possible problem that remains is that the right child of v may be red, in which case the left-leaning property is violated. We check this and perform a flipLeft(v) to correct it if necessary.

RedBlackTree

```
Node* removeFixupCase2(Node *u) {
  Node *w = u->parent;
  Node *v = w->right;
  pullBlack(w); // w->left
  flipLeft(w); // w is now red
  Node *q = w->right;
  if (q->color == red) { // q-w is red-red
    rotateLeft(w);
    flipRight(v);
    pushBlack(q);
    if (v->right->color == red)
      flipLeft(v);
    return q;
  } else {
    return v;
  }
}
```

Case 3: u's sibling is black and u is the right child of its parent, w. This case is symmetric to Case 2 and is handled mostly the same way. The only differences come from the fact that the left-leaning property is asymmetric, so it requires different handling.

As before, we begin with a call to pullBlack(w), which makes v red and u black. A call to flipRight(w) promotes v to the root of the subtree. At this point w is red, and the code branches two ways depending on the color of w's left child, q.

If q is red, then the code finishes up exactly the same way that Case 2 finishes up, but is even simpler since there is no danger of v not satisfying the left-leaning property.

The more complicated case occurs when q is black. In this case, we examine the color if v's left child. If it is red, then v has two red children and its extra black can be pushed down with a call to pushBlack(v). At this point, v now has w's original color and we are done.

If v's left child is black then v violates the left-leaning property and we restore this with a call to flipLeft(v). The next iteration of removeFixup(u) then continues with

u = v.

RedBlackTree

```
Node* removeFixupCase3(Node *u) {
  Node *w = u->parent;
  Node *v = w->left;
  pullBlack(w);
  flipRight(w);            // w is now red
  Node *q = w->left;
  if (q->color == red) {    // q-w is red-red
    rotateRight(w);
    flipLeft(v);
    pushBlack(q);
    return q;
  } else {
    if (v->left->color == red) {
      pushBlack(v);   // both v's children are red
      return v;
    } else {            // ensure left-leaning
      flipLeft(v);
      return w;
    }
  }
}
```

Each iteration of `removeFixup(u)` takes constant time. Cases 2 and 3 either finish or move `u` closer to the root of the tree. Case 0 (where `u` is the root) always terminates and Case 1 leads immediately to Case 3, which also terminates. Since the height of the tree is at most $2\log \mathtt{n}$, we conclude that there are at most $O(\log \mathtt{n})$ iterations of `removeFixup(u)` so `removeFixup(u)` runs in $O(\log \mathtt{n})$ time.

9.3 Summary

The following theorem summarizes the performance of the `RedBlackTree` data structure:

Theorem 9.1. *A* `RedBlackTree` *implements the* `SSet` *interface. A* `RedBlackTree` *supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log \mathtt{n})$ *worst-case time per operation.*

Not included in the above theorem is the extra bonus

Theorem 9.2. *Beginning with an empty* `RedBlackTree`, *any sequence of* m `add(x)` *and* `remove(x)` *operations results in a total of* $O(m)$ *time spent during all calls* `addFixup(u)` *and* `removeFixup(u)`.

We will only sketch a proof of Theorem 9.2. By comparing `addFixup(u)` and `removeFixup(u)` with the algorithms for adding or removing a leaf in a 2-4 tree, we can convince ourselves that this property is something that is inherited from a 2-4 tree. In particular, if we can show that the total time spent splitting, merging, and borrowing in a 2-4 tree is $O(m)$, then this implies Theorem 9.2.

The proof of this for 2-4 trees uses the potential method of amortized analysis.[2] Define the potential of an internal node `u` in a 2-4 tree as

$$\Phi(\texttt{u}) = \begin{cases} c & \text{if u has 2 children} \\ 0 & \text{if u has 3 children} \\ 3c & \text{if u has 3 children} \end{cases}$$

and the potential of a 2-4 tree as the sum of the potentials of its nodes. When a split occurs, it is because a node of degree 4 becomes two nodes, one of degree 2 and one of degree 3. This means that the overall potential drops by $3c - c - 0 = 2c$. Thus, the time spent performing a split is accounted for by the drop in potential. When a merge occurs, two nodes that used to have degree 2 are replaced by one node of degree 3. The result is a drop in potential of $2c - 0 = 2c$. This drop in potential accounts for the cost of the merge. A borrow operation happens only once per deletion and, except for merges and splits, the potential can only increase by a constant amount for each addition or removal of a leaf.

The above analysis implies that, beginning with an empty 2-4 tree, any sequence of m additions and removals of leaves results in at most $O(m)$ time spent splitting nodes, merging nodes, and borrowing children from nodes. Theorem 9.2 is a consequence of this analysis and the correspondence between 2-4 trees and red-black trees.

9.4 Discussion and Exercises

Red-black trees were first introduced by Guibas and Sedgewick [31]. Despite their high implementation complexity they are found in some of the most commonly used libraries and applications. Most algorithms and data structures discuss some variant of red-black trees.

Andersson [4] describes a left-leaning version of balanced trees that are similar to red-black trees but have the additional constraint that any node has at most one red child. This implies that these trees simulate 2-3 trees rather than 2-4 trees. They are significantly simpler, though, than the `RedBlackTree` structure presented in this chapter.

[2] See the proofs of Lemma 2.2 and Lemma 3.1 for other applications of the potential method.

Sedgewick [49] describes at least two versions of left-leaning red-black trees. These use recursion along with a simulation of top-down splitting and merging in 2-4 trees. The combination of these two techniques makes for particularly short and elegant code.

A related, and older, data structure is the AVL tree [2]. AVL trees are *height-balanced*: At each node u, the height of the subtree rooted at `u.left` and the subtree rooted at `u.right` differ by at most one. It follows immediately that, if $F(h)$ is the minimum number of leaves in a tree of height h, then $F(h)$ obeys the Fibonacci recurrence

$$F(h) = F(h-1) + F(h-2)$$

with base cases $F(0) = 1$ and $F(1) = 1$. This means $F(h)$ is approximately $\varphi^h/\sqrt{5}$, where $\varphi = (1+\sqrt{5})/2 \approx 1.61803399$ is the *golden ratio*. (More precisely, $|\varphi^h/\sqrt{5} - F(h)| \leq 1/2$.) Arguing as in the proof of Lemma 9.1, this implies

$$h \leq \log_\varphi \mathtt{n} \approx 1.440420088 \log \mathtt{n} \ ,$$

so AVL trees have smaller height than red-black trees. The height-balanced property can be maintained during `add(x)` and `remove(x)` operations by walking back up the path to the root and performing a rebalancing operation at each node `u` where the height of `u`'s left and right subtrees differ by 2. See Figure 9.10.

Andersson's variant of red-black trees, Sedgewick's variant of red-black trees, and AVL trees are all simpler to implement than the `RedBlackTree` structure defined here. Unfortunately, none of them can guarantee that the amortized time spent rebalancing is $O(1)$ per update. In particular, there is no analogue of Theorem 9.2 for those structures.

Exercise 9.1. Why does the method `remove(x)` in the `RedBlackTree` implementation perform the assignment `u.parent = w.parent`? Shouldn't this already be done by the call to `splice(w)`?

Exercise 9.2. Suppose a 2-4 tree, T, has $\mathtt{n}_\ell$ leaves and $\mathtt{n}_i$ internal nodes.

1. What is the minimum value of $\mathtt{n}_i$, as a function of $\mathtt{n}_\ell$?
2. What is the maximum value of $\mathtt{n}_i$, as a function of $\mathtt{n}_\ell$?
3. If T' is a red-black tree that represents T, then how many red nodes does T' have?

Exercise 9.3. Prove that, during an `add(x)` operation, an AVL tree must perform at most one rebalancing operation (that involves at most 2 rotations; see Figure 9.10). Give an example of an AVL tree and a `remove(x)` operation on that tree that requires on the order of $\log \mathtt{n}$ rebalancing operations.

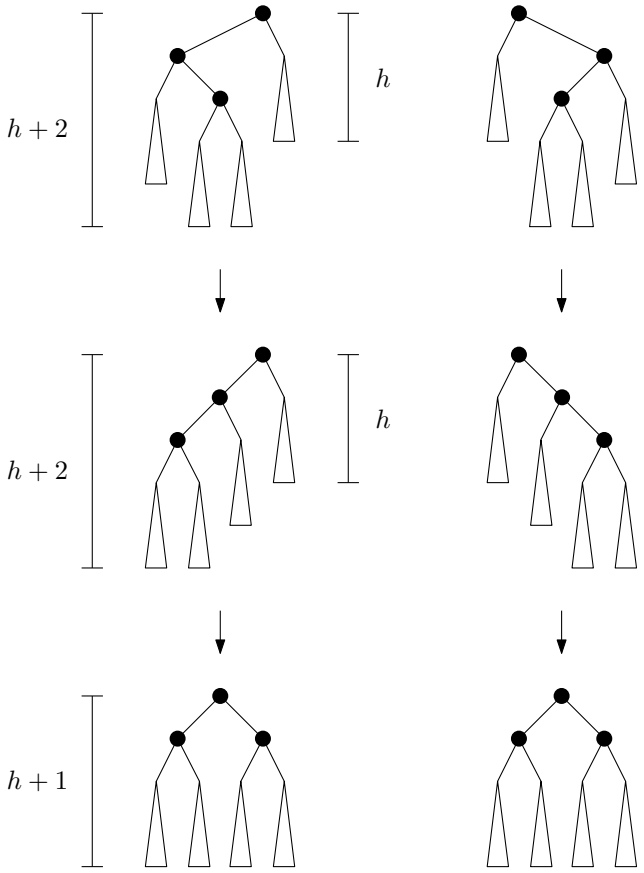


Figure 9.10: Rebalancing in an AVL tree. At most 2 rotations are required to convert a node whose subtrees have height h and $h + 2$ into a node whose subtrees each have height at most $h + 1$.

Exercise 9.4. Implement an `AVLTree` class that implements AVL trees as described above. Compare its performance to that of the `RedBlackTree` implementation. Which implementation has a faster `find(x)` operation?

Exercise 9.5. Design and implement a series of experiments that compare the relative performance of `find(x)`, `add(x)`, and `remove(x)` for `SkiplistSSet`, `ScapegoatTree`, `Treap`, and `RedBlackTree`. Be sure to include multiple test scenarios, including cases where the data is random, already sorted, is removed in random order, is removed in sorted order, and so on.

Chapter 10

Heaps

In this chapter, we discuss two implementations of the extremely useful priority `Queue` data structure. The first is an implementation based on arrays. It is very fast and is the basis of one of the fastest known sorting algorithms, namely heapsort (see Section 11.1.3). The second implementation is based on binary trees and is more flexible. In particular, it supports a `meld(h)` operation that allows the priority queue to absorb the elements of a second priority queue `h`.

10.1 `BinaryHeap`: An Implicit Binary Tree

Our first implementation of a (priority) `Queue` is based on a technique that is over 400 years old. Eytzinger's method allows us to represent a complete binary tree as an array. This is done by laying out the nodes of the tree in breadth-first order (see Section 6.1.2) in the array. In this way, the root is stored at position 0, the root's left child is stored at position 1, the root's right child at position 2, the left child of the left child of the root is stored at position 3, and so on. See Figure 10.1.

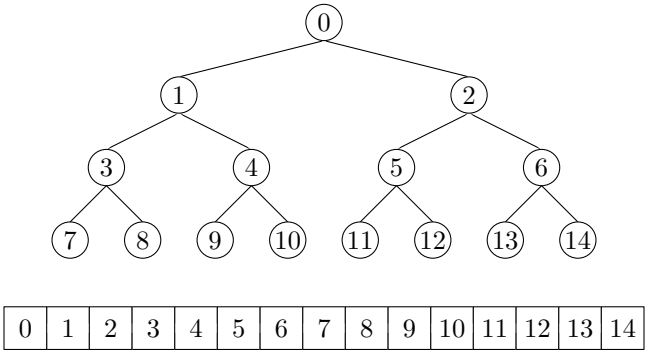


Figure 10.1: Eytzinger's method represents a complete binary tree as an array.

If we do this for a large enough tree, some patterns emerge. The left child of the node at index i is at index left(i) = 2i + 1 and the right child of the node at index i is at index right(i) = 2i + 2. The parent of the node at index i is at index parent(i) = (i − 1)/2.

BinaryHeap
```
int left(int i) {
  return 2*i + 1;
}
int right(int i) {
  return 2*i + 2;
}
int parent(int i) {
  return (i-1)/2;
}
```

A BinaryHeap uses this technique to implicitly represent a complete binary tree in which the elements are *heap-ordered*: The value stored at any index i is not smaller than the value stored at index parent(i), with the exception of the root value, i = 0. It follows that the smallest value in the priority Queue is therefore stored at position 0 (the root).

In a BinaryHeap, the n elements are stored in an array a:

BinaryHeap
```
array<T> a;
int n;
```

Implementing the add(x) operation is fairly straightforward. As with all array-based structures, we first check if a is full (because a.length = n) and, if so, we grow a. Next, we place x at location a[n] and increment n. At this point, all that remains is to ensure that we maintain the heap property. We do this by repeatedly swapping x with its parent until x is no longer smaller than its parent. See Figure 10.2.

BinaryHeap
```
bool add(T x) {
  if (n + 1 > a.length) resize();
  a[n++] = x;
  bubbleUp(n-1);
  return true;
}
void bubbleUp(int i) {
  int p = parent(i);
  while (i > 0 && compare(a[i], a[p]) < 0) {
    a.swap(i,p);
    i = p;
    p = parent(i);
```

```
    }
  }
```

Implementing the `remove`() operation, which removes the smallest value from the heap, is a little trickier. We know where the smallest value is (at the root), but we need to replace it after we remove it and ensure that we maintain the heap property.

The easiest way to do this is to replace the root with the value `a`[`n`] and decrement `n`. Unfortunately, the new element now at the root is probably not the smallest element, so it needs to be moved downwards. We do this by repeatedly comparing this element to its two children. If it is the smallest of the three then we are done. Otherwise, we swap this element with the smallest of its two children and continue.

BinaryHeap

```
T remove() {
  T x = a[0];
  a[0] = a[--n];
  trickleDown(0);
  if (3*n < a.length) resize();
  return x;
}
void trickleDown(int i) {
  do {
    int j = -1;
    int r = right(i);
    if (r < n && compare(a[r], a[i]) < 0) {
      int l = left(i);
      if (compare(a[l], a[r]) < 0) {
        j = l;
      } else {
        j = r;
      }
    } else {
      int l = left(i);
      if (l < n && compare(a[l], a[i]) < 0) {
        j = l;
      }
    }
    if (j >= 0)  a.swap(i, j);
    i = j;
  } while (i >= 0);
}
```

As with other array-based structures, we will ignore the time spent in calls to

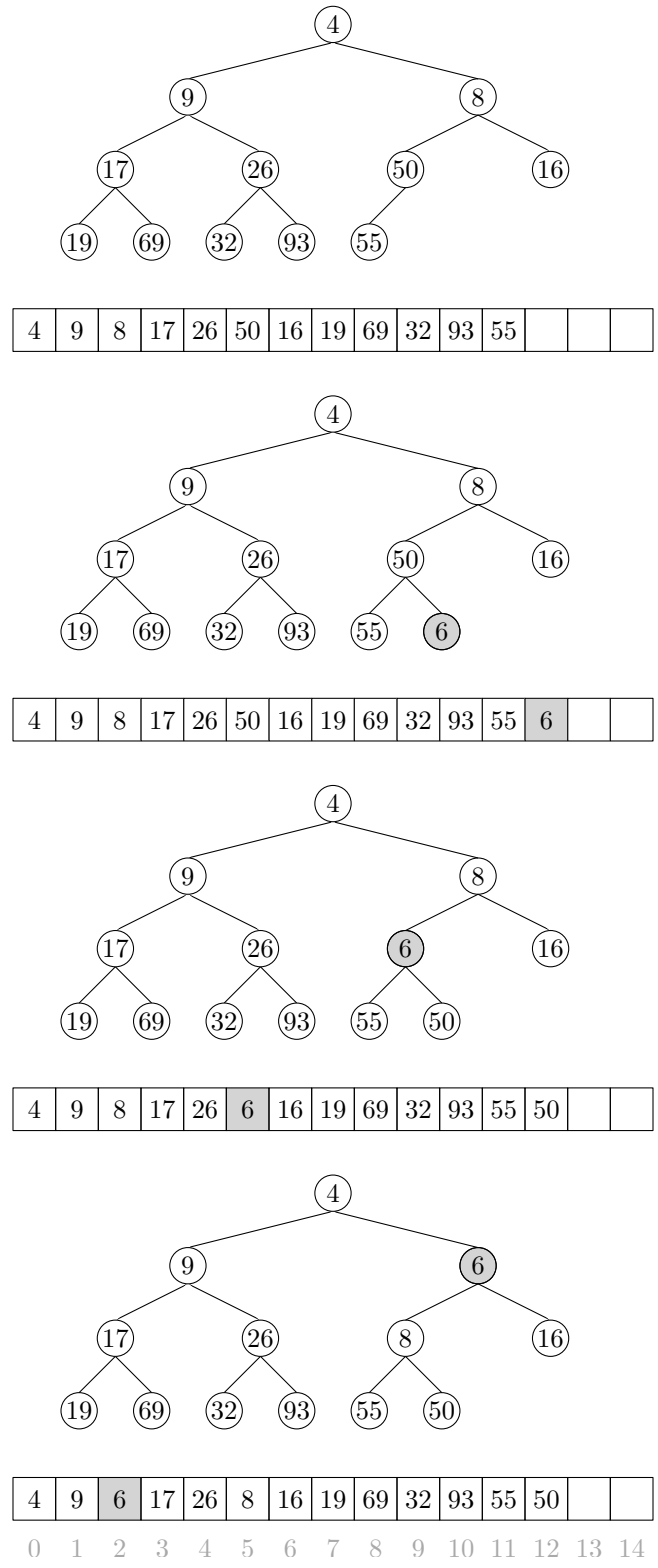


Figure 10.2: Inserting the value 6 into a `BinaryHeap`.

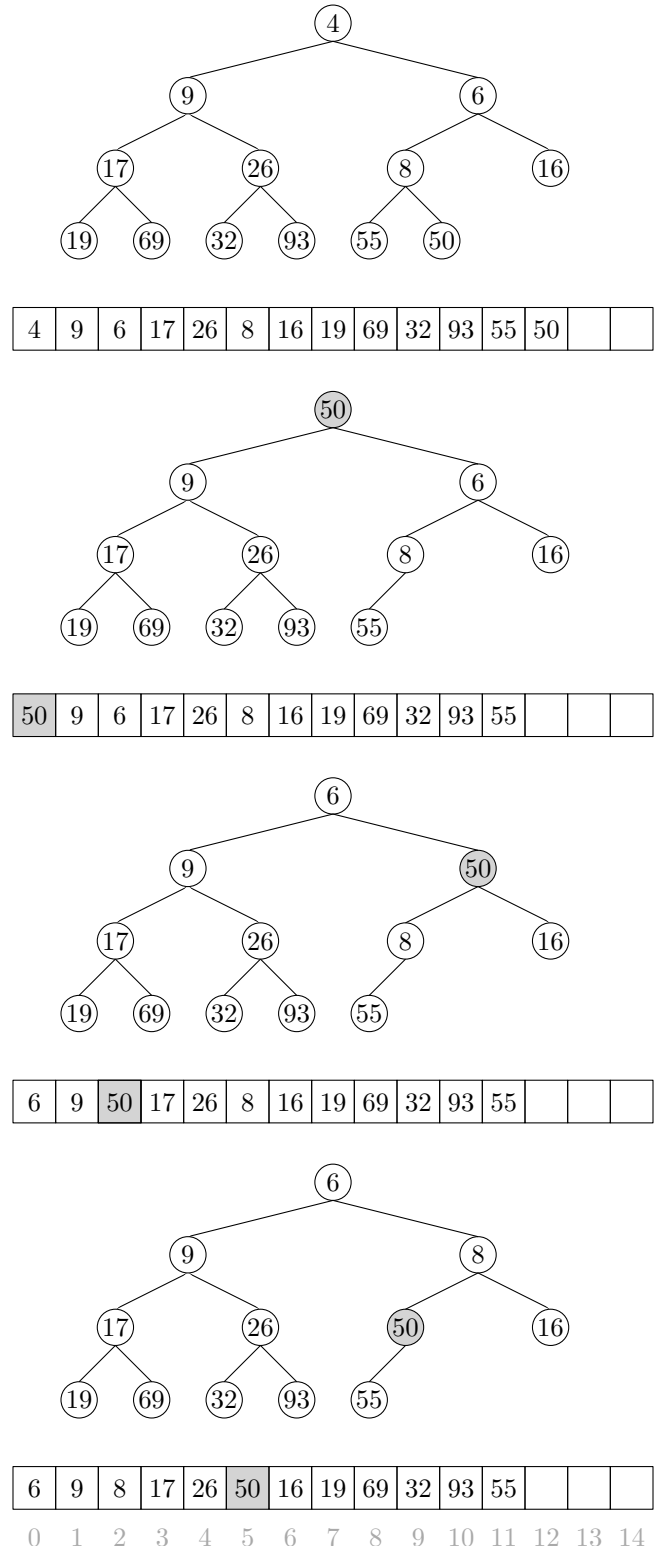


Figure 10.3: Removing the minimum value, 4, from a `BinaryHeap`.

`resize`(), since these can be accounted for with the amortization argument from Lemma 2.1. The running times of both `add`(`x`) and `remove`() then depend on the height of the (implicit) binary tree. However, this is a *complete* binary tree; every level except the last has the maximum possible number of nodes. Therefore, if the height of this tree is h, then it has at least 2^h nodes. Stated another way

$$\mathtt{n} \geq 2^h \ .$$

Taking logarithms on both sides of this equation gives

$$h \leq \log \mathtt{n} \ .$$

Therefore, both the `add`(`x`) and `remove`() operation run in $O(\log \mathtt{n})$ time.

10.1.1 Summary

The following theorem summarizes the performance of a `BinaryHeap`:

Theorem 10.1. *A* `BinaryHeap` *implements the (priority)* `Queue` *interface. Ignoring the cost of calls to* `resize`(), *a* `BinaryHeap` *supports the operations* `add`(`x`) *and* `remove`() *in* $O(\log \mathtt{n})$ *time per operation.*

Furthermore, beginning with an empty `BinaryHeap`, *any sequence of* m `add`(`x`) *and* `remove`() *operations results in a total of* $O(m)$ *time spent during all calls to* `resize`().

10.2 `MeldableHeap`: A Randomized Meldable Heap

In this section, we describe the `MeldableHeap`, a priority `Queue` implementation in which the underlying structure is also a heap-ordered binary tree. However, unlike a `BinaryHeap` in which the underlying binary tree is completely defined by the number of elements, there are no restrictions on the shape of the binary tree that underlies a `MeldableHeap`; anything goes.

The `add`(`x`) and `remove`() operations in a `MeldableHeap` are implemented in terms of the `merge`(`h1`, `h2`) operation. This operation takes two heap nodes `h1` and `h2` and merges them, returning a heap node that is the root of a heap that contains all elements in the subtree rooted at `h1` and all elements in the subtree rooted at `h2`.

The nice thing about a `merge`(`h1`, `h2`) operation is that it can be defined recursively. If either of `h1` or `h2` is `null`, then we are merging with an empty set, so we return `h2` or `h1`, respectively. Otherwise, assume `h1.x` $\leq$ `h2.x` since, if `h1.x` $>$ `h2.x`, then we can reverse the roles of `h1` and `h2`. Then we know that the root of the merged heap will contain `h1.x`

and we can recursively merge h2 with h1.left or h1.right, as we wish. This is where randomization comes in, and we toss a coin to decide whether to merge h2 with h1.left or h1.right:

MeldableHeap

```
Node* merge(Node *h1, Node *h2) {
  if (h1 == nil) return h2;
  if (h2 == nil) return h1;
  if (compare(h1->x, h2->x) > 0) {  // ensure h1->x < h2->x
    Node *tmp = h1;
    h1 = h2;
    h2 = tmp;
  }
  if (rand() % 2) {
    h1->left = merge(h1->left, h2);
    if (h1->left != nil) h1->left->parent = h1;
  } else {
    h1->right = merge(h1->right, h2);
    if (h1->right != nil) h1->right->parent = h1;
  }
  return h1;
}
```

In the next section, we show that merge(h1, h2) runs in $O(\log \mathtt{n})$ expected time, where n is the total number of elements in h1 and h2.

With access to a merge(h1, h2) operation, the add(x) operation is easy. We create a new node u containing x and then merge u with the root of our heap:

MeldableHeap

```
bool add(T x) {
  Node *u = new Node();
  u->left = u->right = u->parent = nil;
  u->x = x;
  r = merge(u, r);
  r->parent = nil;
  n++;
  return true;
}
```

This takes $O(\log(\mathtt{n}+1)) = O(\log \mathtt{n})$ expected time.

The remove() operation is similarly easy. The node we want to remove is the root, so we just merge its two children and make the result the root:

MeldableHeap

```
T remove() {
  T x = r->x;
```

```
    Node *tmp = r;
    r = merge(r->left, r->right);
    delete tmp;
    if (r != nil) r->parent = nil;
    n--;
    return x;
  }
```

Again, this takes $O(\log \mathtt{n})$ expected time.

Additionally, a `MeldableHeap` can implement many other operations in $O(\log \mathtt{n})$ expected time, including:

- `remove`(`u`): remove the node `u` (and its key `u.x`) from the heap.
- `absorb`(`h`): add all the elements of the `MeldableHeap` `h` to this heap, emptying `h` in the process.

Each of these operations can be implemented using a constant number of `merge`(`h1`, `h2`) operations that each take $O(\log \mathtt{n})$ time.

10.2.1 Analysis of `merge`(`h1`, `h2`)

The analysis of `merge`(`h1`, `h2`) is based on the analysis of a random walk in a binary tree. A *random walk* in a binary tree is a walk that starts at the root of the tree. At each step in the walk, a coin is tossed and the walk proceeds to the left or right child of the current node depending on the result of this coin toss. The walk ends when it falls off the tree (the current node becomes `null`).

The following lemma is somewhat remarkable because it does not depend at all on the shape of the binary tree:

Lemma 10.1. *The expected length of a random walk in a binary tree with* $\mathtt{n}$ *nodes is at most* $\log(\mathtt{n}+1)$.

Proof. The proof is by induction on $\mathtt{n}$. In the base case, $\mathtt{n} = 0$ and the walk has length $0 = \log(\mathtt{n}+1)$. Suppose now that the result is true for all non-negative integers $\mathtt{n}' < \mathtt{n}$.

Let $\mathtt{n}_1$ denote the size of the root's left subtree, so that $\mathtt{n}_2 = \mathtt{n} - \mathtt{n}_1 - 1$ is the size of the root's right subtree. Starting at the root, the walk takes one step and then continues in a subtree of size $\mathtt{n}_1$ or continues in a subtree of size $\mathtt{n}_2$. By our inductive hypothesis, the expected length of the walk is then

$$\mathrm{E}[W] = 1 + \frac{1}{2}\log(\mathtt{n}_1 + 1) + \frac{1}{2}\log(\mathtt{n}_2 + 1) \ ,$$

since each of $\mathtt{n}_1$ and $\mathtt{n}_2$ are less than $\mathtt{n}$. To maximize this, over all choices of $\mathtt{n}_1 \in [0, \mathtt{n}-1]$, we take the derivative and obtain

$$(\mathrm{E}[W])' = \frac{1}{2}(c/\mathtt{n}_1 - c/(\mathtt{n} - \mathtt{n}_1 - 1)) \ ,$$

which is equal to 0 when $\mathtt{n}_1 = (\mathtt{n}-1)/2$. We can establish that this is a maximum fairly easily, so the expected number of steps taken by the random walk is

$$\begin{aligned}\mathrm{E}[W] &= 1 + \frac{1}{2}\log(\mathtt{n}_1 + 1) + \frac{1}{2}\log(\mathtt{n}_2 + 1) \\ &\le 1 + \log((\mathtt{n}-1)/2 + 1) \\ &= 1 + \log((\mathtt{n}+1)/2) \\ &= \log(\mathtt{n}+1) \ .\end{aligned}$$

□

We make a quick digression to note that, for readers who know a little about information theory, the proof of Lemma 10.1 can be stated in terms of entropy.

Information Theoretic Proof of Lemma 10.1. Let d_i denote the depth of the ith external node and recall that a binary tree with $\mathtt{n}$ nodes has $\mathtt{n}+1$ external nodes. The probability of the random walk reaching the ith external node is exactly $p_i = 1/2^{d_i}$, so the expected length of the random walk is given by

$$H = \sum_{i=0}^{\mathtt{n}} p_i d_i = \sum_{i=0}^{\mathtt{n}} p_i \log(1/p_i)$$

The right hand side of this equation is easily recognizable as the entropy of a probability distribution over $\mathtt{n}+1$ elements. A basic fact about the entropy of a distribution over $\mathtt{n}+1$ elements is that it does not exceed $\log(\mathtt{n}+1)$, which proves the lemma. □

With this result on random walks, we can now easily prove that the running time of the `merge(h1,h2)` operation is $O(\log \mathtt{n})$.

Lemma 10.2. *If* `h1` *and* `h2` *are the roots of two heaps containing* $\mathtt{n}_1$ *and* $\mathtt{n}_2$ *nodes, respectively, then the expected running time of* `merge(h1,h2)` *is at most* $O(\log \mathtt{n})$*, where* $\mathtt{n} = \mathtt{n}_1 + \mathtt{n}_2$*.*

Proof. Each step of the merge algorithm takes one step of a random walk, either in the heap rooted at `h1` or the heap rooted at `h2`, depending on whether `h1.x` < `h2.x` or not. The algorithm terminates when either of these two random walks fall out of its corresponding tree (when `h1` = `null` or `h2` = `null`). Therefore, the expected number of steps performed by the merge algorithm is at most

$$\log(\mathtt{n}_1 + 1) + \log(\mathtt{n}_2 + 1) \le 2\log \mathtt{n} \ .$$

□

10.2.2 Summary

The following theorem summarizes the performance of a `MeldableHeap`:

Theorem 10.2. *A* `MeldableHeap` *implements the (priority)* `Queue` *interface. A* `MeldableHeap` *supports the operations* `add(x)` *and* `remove()` *in* $O(\log \mathtt{n})$ *expected time per operation.*

10.3 Discussion and Exercises

The implicit representation of a complete binary tree as an array, or list, seems to have been first proposed by Eytzinger [22], as a representation for pedigree family trees. The `BinaryHeap` data structure described here was first introduced by Williams [56].

The randomized `MeldableHeap` data structure described here appears to have first been proposed by Gambin and Malinowski [28]. Other meldable heap implementations exist, including leftist heaps [12, 38, Section 5.3.2], binomial heaps [54], Fibonacci heaps [26], pairing heaps [25], and skew heaps [53], although none of these are as simple as the `MeldableHeap` structure described here.

Some of the above structures also support a `decreaseKey(u, y)` operation in which the value stored at node `u` is decreased to `y`. (It is a pre-condition that $\mathtt{y} \leq \mathtt{u.x}$.) This operation can be implemented in $O(\log \mathtt{n})$ time in most of the above structures by removing node `u` and adding `y`. However, some of these structures can implement `decreaseKey(u, y)` more efficiently. In particular, `decreaseKey(u, y)` takes $O(1)$ amortized time in Fibonacci heaps and $O(\log \log \mathtt{n})$ amortized time in a special version of pairing heaps [20]. This more efficient `decreaseKey(u, y)` operation has applications in speeding up several graph algorithms including Dijkstra's shortest path algorithm [26].

Chapter 11

Sorting Algorithms

This chapter discusses algorithms for sorting a set of $\mathtt{n}$ items. This might seem like a strange topic for a book on data structures, but there are several good reasons for including it here. The most obvious reason is that two of these sorting algorithms (quicksort and heap-sort) are intimately related to two of the data structures we have already studied (random binary search trees and heaps, respectively).

The first part of this chapter discusses algorithms that sort using only comparisons and presents three algorithms that run in $O(\mathtt{n}\log\mathtt{n})$ time. As it turns out, all three algorithms are asymptotically optimal; no algorithm that uses only comparisons can avoid doing roughly $\mathtt{n}\log\mathtt{n}$ comparisons in the worst-case and even the average-case.

The second part of this chapter shows that, if we allow other operations besides comparisons, then all bets are off. Indeed, by using array-indexing, it is possible to sort a set of $\mathtt{n}$ integers in the range $\{0,\ldots,\mathtt{n}^c-1\}$ in $O(c\mathtt{n})$ time.

11.1 Comparison-Based Sorting

In this section, we present three sorting algorithms: merge-sort, Quicksort, and heap-sort. All these algorithms take an input array `a` and sort the elements of `a` into non-decreasing order in $O(\mathtt{n}\log\mathtt{n})$ (expected) time. These algorithms are all *comparison-based*. Their second second argument, `c`, is a `Comparator` that implements the `compare(a,b)` method. These algorithms don't care what type of data is being sorted, the only operation they do on the data is comparisons using the `compare(a,b)` method. Recall, from Section 1.1.4, that `compare(a,b)` returns a negative value if `a` $<$ `b`, a positive value if `a` $>$ `b`, and zero if `a` $=$ `b`.

11.1.1 Merge-Sort

The *merge-sort* algorithm is a classic example of recursive divide and conquer: If the length of `a` is at most 1, then `a` is already sorted, so we do nothing. Otherwise, we split `a` into two

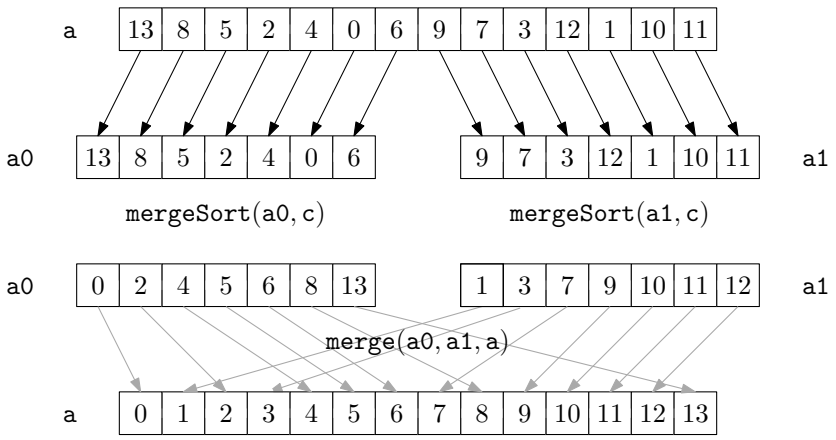


Figure 11.1: The execution of `mergeSort(a, c)`

halves, `a0` $= a[0], \ldots, a[n/2 - 1]$ and `a1` $= a[n/2], \ldots, a[n - 1]$. We recursively sort `a0` and `a1`, and then we merge (the now sorted) `a0` and `a1` to get our fully sorted array `a`:

Algorithms

```
void mergeSort(array<T> &a) {
  if (a.length <= 1) return;
  array<T> a0(0);
  array<T>::copyOfRange(a0, a, 0, a.length/2);
  array<T> a1(0);
  array<T>::copyOfRange(a1, a, a.length/2, a.length);
  mergeSort(a0);
  mergeSort(a1);
  merge(a0, a1, a);
}
```

An example is shown in Figure 11.1.

Compared to sorting, merging the two sorted arrays `a0` and `a1` fairly easy. We add elements to `a` one at a time. If `a0` or `a1` is empty we add the next element from the other (non-empty) array. Otherwise, we take the minimum of the next element in `a0` and the next element in `a1` and add it to `a`:

Algorithms

```
void merge(array<T> &a0, array<T> &a1, array<T> &a) {
  int i0 = 0, i1 = 0;
  for (int i = 0; i < a.length; i++) {
    if (i0 == a0.length)
      a[i] = a1[i1++];
    else if (i1 == a1.length)
      a[i] = a0[i0++];
```

```
    else if (compare(a0[i0], a1[i1]) < 0)
      a[i] = a0[i0++];
    else
      a[i] = a1[i1++];
  }
}
```

Notice that the `merge`(`a0`, `a1`, `a`, `c`) algorithm performs at most $\mathtt{n}-1$ comparisons before running out of elements in one of `a0` or `a1`.

The following theorem shows that `mergeSort`(`a`, `c`) is a very efficient algorithm:

Theorem 11.1. *The* `mergeSort`(`a`, `c`) *algorithm runs in* $O(\mathtt{n}\log\mathtt{n})$ *time and performs at most* $\mathtt{n}\log\mathtt{n}$ *comparisons.*

Proof. The proof is by induction on `n`. The base case, in which $\mathtt{n}=1$, is trivial.

Merging two sorted lists of total length `n` requires at most $\mathtt{n}-1$ comparisons. Let $C(\mathtt{n})$ denote the maximum number of comparisons performed by `mergeSort`(`a`, `c`) on an array `a` of length `n`. If `n` is even, then we apply the inductive hypothesis to the two subproblems and obtain

$$\begin{aligned} C(\mathtt{n}) &\leq \mathtt{n}-1+2C(\mathtt{n}/2) \\ &\leq \mathtt{n}-1+2((\mathtt{n}/2)\log(\mathtt{n}/2)) \\ &= \mathtt{n}-1+\mathtt{n}\log(\mathtt{n}/2) \\ &= \mathtt{n}-1+\mathtt{n}\log\mathtt{n}-\mathtt{n} \\ &< \mathtt{n}\log\mathtt{n}\ . \end{aligned}$$

The case where `n` is odd is slightly more complicated. For this case, we use two inequalities, that are easy to verify

$$\log(x+1) \leq \log(x)+1\ , \tag{11.1}$$

for all $x \geq 1$ and

$$\log(x+1/2)+\log(x-1/2) \leq 2\log(x)\ , \tag{11.2}$$

for all $x \geq 1/2$. Inequality (11.1) comes from the fact that $\log(x)+1=\log(2x)$ while (11.2) follows from the fact that log is a concave function. With these tools in hand we have, for

odd n,

$$\begin{aligned}
C(n) &\leq n - 1 + C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) \\
&\leq n - 1 + \lceil n/2 \rceil \log \lceil n/2 \rceil + \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor \\
&= n - 1 + (n/2 + 1/2) \log(n/2 + 1/2) + (n/2 - 1/2) \log(n/2 - 1/2) \\
&\leq n - 1 + n \log(n/2) + (1/2)(\log(n/2 + 1/2) - \log(n/2 - 1/2)) \\
&\leq n - 1 + n \log(n/2) + 1/2 \\
&< n + n \log(n/2) \\
&= n + n(\log n - 1) \\
&= n \log n \ .
\end{aligned}$$

□

11.1.2 Quicksort

The *quicksort* algorithm is another classic divide and conquer algorithm. Unlike merge-sort, which does merging after solving the two subproblems, quicksort does all its work upfront.

The algorithm is simple to describe: Pick a random *pivot* element, `x`, from `a`; partition `a` into the set of elements less than `x`, the set of elements equal to `x`, and the set of elements greater than `x`; and, finally, recursively sort the first and third sets in this partition. An example is shown in Figure 11.2.

Algorithms

```
void quickSort(array<T> &a) {
  quickSort(a, 0, a.length);
}
void quickSort(array<T> &a, int i, int n) {
  if (n <= 1) return;
  T x = a[i + rand()%n];
  int p = i-1, j = i, q = i+n;
  // a[i..p]<x,  a[p+1..q-1]??x, a[q..i+n-1]>x
  while (j < q) {
    int comp = compare(a[j], x);
    if (comp < 0) {       // move to beginning of array
      a.swap(j++, ++p);
    } else if (comp > 0) {
      a.swap(j, --q);  // move to end of array
    } else {
      j++;              // keep in the middle
    }
  }
  // a[i..p]<x,  a[p+1..q-1]=x, a[q..i+n-1]>x
  quickSort(a, i, p-i+1);
```

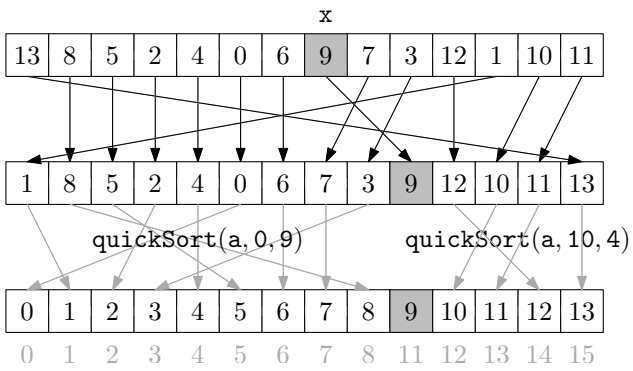


Figure 11.2: An example execution of `quickSort(a, 0, 14)`

```
  quickSort(a, q, n-(q-i));
}
```

All of this is done in-place, so that instead of making copies of subarrays being sorted, the `quickSort(a, i, n, c)` method only sorts the subarray `a[i]`, ..., `a[i + n - 1]`. Initially, this method is called as `quickSort(a, 0, a.length, c)`.

At the heart of the quicksort algorithm is the in-place partitioning that, without any extra space, swaps elements in `a` and computes indices `p` and `q` so that

$$\texttt{a[i]} \begin{cases} < \texttt{x} & \text{if } 0 \le \texttt{i} \le \texttt{p} \\ = \texttt{x} & \text{if } \texttt{p} < \texttt{i} < \texttt{q} \\ > \texttt{x} & \text{if } \texttt{q} \le \texttt{i} \le \texttt{n} - 1 \end{cases}$$

This partitioning, which is done by the `while` loop in the code, works by iteratively increasing `p` and decreasing `q` while maintaining the first and last of these conditions. At each step, the element at position `j` is either moved to the front, left where it is, or moved to the back. In the first two cases, `j` is incremented, while in the last case, `j` is not incremented since the new element at position `j` has not been processed yet.

Quicksort is very closely related to the random binary search trees studied in Section 7.1. In fact, if the input to quicksort consists of `n` distinct elements, then the quicksort recursion tree is a random binary search tree. To see this, recall that when constructing a random binary search tree the first thing we do is pick a random element `x` and make it the root of the tree. After this, every element will eventually be compared to `x`, with smaller elements going into the left subtree and larger elements going into the right subtree.

In quicksort, we select a random element x and immediately compare everything to x, putting the smaller elements at the beginning of the array and larger elements at the end of the array. Quicksort then recursively sorts the beginning of the array and the end of the array, while the random binary search tree recursively inserts smaller elements in the left subtree of the root and larger elements in the right subtree of the root.

The above correspondence between random binary search trees and quicksort means that we can translate Lemma 7.1 to a statement about quicksort:

Lemma 11.1. *When quicksort is called to sort an array containing the integers* $0, \ldots, \mathtt{n}-1$*, the expected number of times element* i *is compared to a pivot element is at most* $H_{\mathtt{i}+1} + H_{\mathtt{n}-\mathtt{i}}$*.*

A little summing of harmonic numbers gives us the following theorem about the running time of quicksort:

Theorem 11.2. *When quicksort is called to sort an array containing* n *distinct elements, the expected number of comparisons performed is* $2\mathtt{n}\ln\mathtt{n} + O(\mathtt{n})$*.*

Proof. Let T be the number of comparisons performed by quicksort when sorting n distinct elements. Using Lemma 11.1, we have:

$$\begin{aligned}
\mathrm{E}[T] &= \sum_{i=0}^{\mathtt{n}-1} (H_{\mathtt{i}+1} + H_{\mathtt{n}-\mathtt{i}}) \\
&= 2\sum_{i=1}^{\mathtt{n}} H_i \\
&\leq 2\sum_{i=1}^{\mathtt{n}} H_{\mathtt{n}} \\
&\leq 2\mathtt{n}\ln\mathtt{n} + 2\mathtt{n} = 2\mathtt{n}\ln\mathtt{n} + O(\mathtt{n})
\end{aligned}$$

□

Theorem 11.3 describes the case where the elements being sorted are all distinct. When the input array, a, contains duplicate elements, the expected running time of quicksort is no worse, and can be even better; any time a duplicate element x is chosen as a pivot, all occurrences of x get grouped together and don't take part in either of the two subproblems.

Theorem 11.3. *The* quickSort(a, c) *method runs in* $O(\mathtt{n}\log\mathtt{n})$ *expected time and the expected number of comparisons it performs is at most* $2\mathtt{n}\ln\mathtt{n} + O(\mathtt{n})$*.*

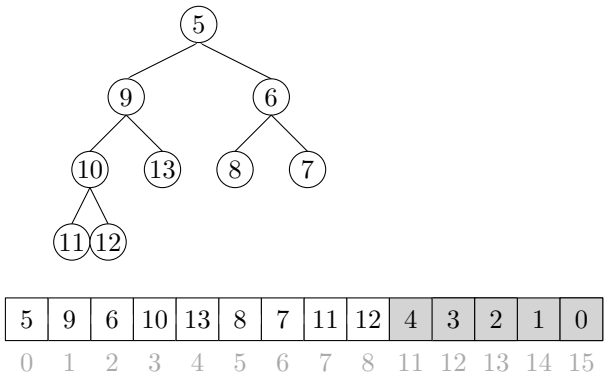


Figure 11.3: A snapshot of the execution of `heapSort(a, c)`. The shaded part of the array is already sorted. The unshaded part is a `BinaryHeap`. During the next iteration, element 5 will be placed into array location 8.

11.1.3 Heap-sort

The heap-sort algorithm is another in-place sorting algorithm. Heap-sort uses the binary heaps discussed in Section 10.1. Recall that the `BinaryHeap` data structure represents a heap using a single array. The heap-sort algorithm converts the input array `a` into a heap and then repeatedly extracts the minimum value.

More specifically, a heap stores `n` elements at array locations $\texttt{a}[0], \ldots, \texttt{a}[\texttt{n}-1]$ with the smallest value stored at the root, $\texttt{a}[0]$. After transforming `a` into a `BinaryHeap`, the heap-sort algorithm repeatedly swaps $\texttt{a}[0]$ and $\texttt{a}[\texttt{n}-1]$, decrements `n`, and calls `trickleDown(0)` so that $\texttt{a}[0], \ldots, \texttt{a}[\texttt{n}-2]$ once again are a valid heap representation. When this process ends (because $\texttt{n} = 0$) the elements of `a` are stored in decreasing order, so `a` is reversed to obtain the final sorted order.[1] Figure 11.1.3 shows an example of the execution of `heapSort(a, c)`.

BinaryHeap

```
void sort(array<T> &b) {
  BinaryHeap<T> h(b);
  while (h.n > 1) {
    h.a.swap(--h.n, 0);
    h.trickleDown(0);
  }
  b = h.a;
  b.reverse();
}
```

[1]The algorithm could alternatively redefine the `compare(x, y)` function so that the heap sort algorithm stores the elements directly in ascending order.

A key subroutine in heap sort is the constructor for turning an unsorted array a into a heap. It would be easy to do this in $O(\mathtt{n}\log\mathtt{n})$ time by repeatedly calling the BinaryHeap add(x) method, but we can do better by using a bottom-up algorithm. Recall that, in a binary heap, the children of a[i] are stored at positions a[2i + 1] and a[2i + 2]. This implies that the elements $\mathtt{a}[\lfloor \mathtt{n}/2 \rfloor], \ldots, \mathtt{a}[\mathtt{n}-1]$ have no children. In other words, each of $\mathtt{a}[\lfloor \mathtt{n}/2 \rfloor], \ldots, \mathtt{a}[\mathtt{n}-1]$ is a sub-heap of size 1. Now, working backwards, we can call trickleDown(i) for each $\mathtt{i} \in \{\lfloor \mathtt{n}/2 \rfloor - 1, \ldots, 0\}$. This works, because by the time we call trickleDown(i), each of the two children of a[i] are the root of a sub-heap so calling trickleDown(i) makes a[i] into the root of its own subheap.

BinaryHeap

```
BinaryHeap(array<T> &b) : a(0) {
  a = b;
  n = a.length;
  for (int i = n/2-1; i >= 0; i--) {
    trickleDown(i);
  }
}
```

The interesting thing about this bottom-up strategy is that it is more efficient than calling add(x) n times. To see this, notice that, for n/2 elements, we do no work at all, for n/4 elements, we call trickleDown(i) on a subheap rooted at a[i] and whose height is 1, for n/8 elements, we call trickleDown(i) on a subheap whose height is 2, and so on. Since the work done by trickleDown(i) is proportional to the height of the sub-heap rooted at a[i], this means that the total work done is at most

$$\sum_{i=1}^{\log \mathtt{n}} O((i-1)\mathtt{n}/2^i) \leq \sum_{i=1}^{\infty} O(i\mathtt{n}/2^i) = O(\mathtt{n})\sum_{i=1}^{\infty} i/2^i = O(2\mathtt{n}) = O(\mathtt{n}) \ .$$

The second-last equality follows by recognizing that the sum $\sum_{i=1}^{\infty} i/2^i$ is equal, by definition, to the expected number times we toss a coin up to and including the first time the coin comes up as heads and applying Lemma 4.2.

The following theorem describes the performance of heapSort(a, c).

Theorem 11.4. *The* heapSort(a, c) *method runs in* $O(\mathtt{n}\log\mathtt{n})$ *time and performs at most* $2\mathtt{n}\log\mathtt{n} + O(\mathtt{n})$ *comparisons.*

Proof. The algorithm runs in 3 steps: (1) Transforming a into a heap, (2) repeatedly extracting the minimum element from a, and (3) reversing the elements in a. We have just argued that step 1 takes $O(\mathtt{n})$ time and performs $O(\mathtt{n})$ comparisons. Step 3 takes $O(\mathtt{n})$

time and performs no comparisons. Step 2 performs $\texttt{n}$ calls to `trickleDown(0)`. The ith such call operates on a heap of size $\texttt{n} - i$ and performs at most $2\log(\texttt{n} - i)$ comparisons. Summing this over i gives

$$\sum_{i=0}^{\texttt{n}-i} 2\log(\texttt{n} - i) \leq \sum_{i=0}^{\texttt{n}-i} 2\log \texttt{n} = 2\texttt{n}\log \texttt{n}$$

Adding the number of comparisons performed in each of the three steps completes the proof. □

11.1.4 A Lower-Bound for Comparison-Based Sorting

We have now seen three comparison-based sorting algorithms that each run in $O(\texttt{n}\log \texttt{n})$ time. By now, we should be wondering if faster algorithms exist. The short answer to this question is no. If the only operations allowed on the elements of `a` are comparisons then no algorithm can avoid doing roughly $\texttt{n}\log \texttt{n}$ comparisons. This is not difficult to prove, but requires a little imagination. Ultimately, it follows from the fact that

$$\log(\texttt{n}!) = \log \texttt{n} + \log(\texttt{n} - 1) + \cdots + \log(1) = \texttt{n}\log \texttt{n} - O(\texttt{n}) \ .$$

(Proving this fact is left as Exercise 11.3.)

We will first focus our attention on deterministic algorithms like merge-sort and heap-sort and on a particular fixed value of $\texttt{n}$. Imagine such an algorithm is being used to sort $\texttt{n}$ distinct elements. The key to proving the lower-bound is to observe that, for a deterministic algorithm with a fixed value of $\texttt{n}$, the first pair of elements that are compared is always the same. For example, in `heapSort(a, c)`, when $\texttt{n}$ is even, the first call to `trickleDown(i)` is with $\texttt{i} = \texttt{n}/2 - 1$ and the first comparison is between elements $\texttt{a}[\texttt{n}/2 - 1]$ and $\texttt{a}[\texttt{n} - 1]$.

Since all input elements are distinct, this first comparison has only two possible outcomes. The second comparison done by the algorithm may depend on the outcome of the first comparison. The third comparison may depend on the results of the first two, and so on. In this way, any deterministic comparison-based sorting algorithm can be viewed as a rooted binary *comparison-tree*. Each internal node, $\texttt{u}$, of this tree is labelled with a pair of of indices $\texttt{u.i}$ and $\texttt{u.j}$. If $\texttt{a}[\texttt{u.i}] < \texttt{a}[\texttt{u.j}]$ the algorithm proceeds to the left subtree, otherwise it proceeds to the right subtree. Each leaf $\texttt{w}$ of this tree is labelled with a permutation $\texttt{w.p}[0], \ldots, \texttt{w.p}[\texttt{n} - 1]$ of $0, \ldots, \texttt{n} - 1$. This permutation represents the permutation that is required to sort `a` if the comparison tree reaches this leaf. That is,

$$\texttt{a}[\texttt{w.p}[0]] < \texttt{a}[\texttt{w.p}[1]] < \cdots < \texttt{a}[\texttt{w.p}[\texttt{n} - 1]] \ .$$

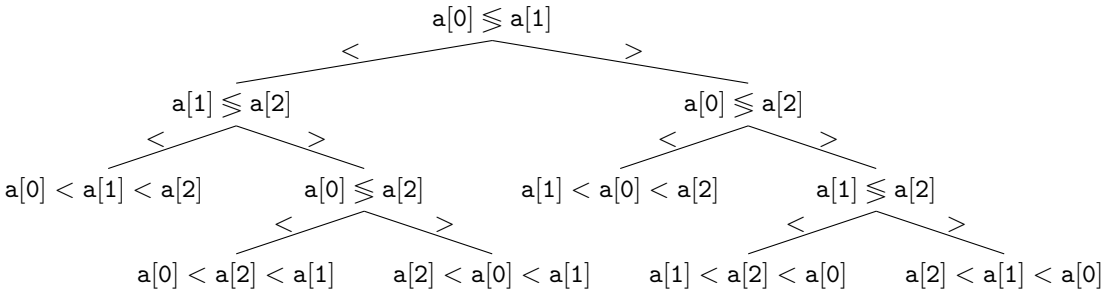


Figure 11.4: A comparison tree for sorting an array $\mathtt{a}[0], \mathtt{a}[1], \mathtt{a}[2]$ of length $\mathtt{n} = 3$.

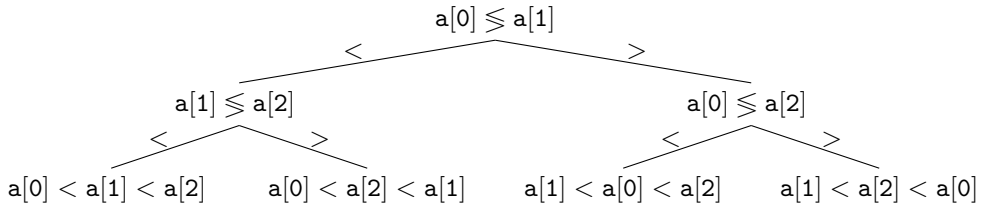


Figure 11.5: A comparison tree that does not correctly sort every input permutation.

An example of a comparison tree for an array of size $\mathtt{n} = 3$ is shown in Figure 11.4.

The comparison tree for a sorting algorithm tells us everything about the algorithm. It tells us exactly the sequence of comparisons that will be performed for any input array `a` having `n` distinct elements and it tells us how the algorithm will reorder `a` to sort it. An immediate consequence of this is that the comparison tree must have at least `n`! leaves; if not, then there are two distinct permutations that lead to the same leaf, so the algorithm does not correctly sort at least one of these permutations.

For example, the comparison tree in Figure 11.5 has only $4 < 3! = 6$ leaves. Inspecting this tree, we see that the two input arrays $3, 1, 2$ and $3, 2, 1$ both lead to the rightmost leaf. On the input $3, 1, 2$ this leaf correctly outputs $\mathtt{a}[1] = 1, \mathtt{a}[2] = 2, \mathtt{a}[0] = 3$. However, on the input $3, 2, 1$, this node incorrectly outputs $\mathtt{a}[1] = 2, \mathtt{a}[2] = 1, \mathtt{a}[0] = 3$. This discussion leads to the primary lower-bound for comparison-based algorithms.

Theorem 11.5. *For any deterministic comparison-based sorting algorithm $\mathcal{A}$ and any integer $\mathtt{n} \geq 1$, there exists an input array `a` of length `n` such that $\mathcal{A}$ performs at least $\log(\mathtt{n}!) = \mathtt{n}\log\mathtt{n} - O(\mathtt{n})$ comparisons when sorting `a`.*

Proof. By the above discussion, the comparison tree defined by $\mathcal{A}$ must have at least `n`!

leaves. An easy inductive proof shows that any binary tree with k leaves has height at least $\log k$. Therefore, the comparison tree for $\mathcal{A}$ has a leaf, $\mathtt{w}$, of depth at least $\log(\mathtt{n}!)$ and there is an input array $\mathtt{a}$ that leads to this leaf. The input array $\mathtt{a}$ is an input for which $\mathcal{A}$ does at least $\log(\mathtt{n}!)$ comparisons. □

Theorem 11.5 deals with deterministic algorithms like merge-sort and heap-sort, but doesn't tell us anything about randomized algorithms like quicksort. Could a randomized algorithm beat the $\log(\mathtt{n}!)$ lower bound on the number of comparisons? The answer, again, is no. Again, the way to prove it is to think differently about what a randomized algorithm is.

In the following discussion, we will implicitly assume that our decision trees have been "cleaned up" in the following way: Any node that can not be reached by some input array $\mathtt{a}$ is removed. This cleaning up implies that the tree has exactly $\mathtt{n}!$ leaves. It has at least $\mathtt{n}!$ leaves because, otherwise, it could not sort correctly. It has at most $\mathtt{n}!$ leaves since each of the possible $\mathtt{n}!$ permutation of $\mathtt{n}$ distinct elements follows exactly one root to leaf path in the decision tree.

We can think of a randomized sorting algorithm $\mathcal{R}$ as a deterministic algorithm that takes two inputs: The input array $\mathtt{a}$ that should be sorted and a long sequence $b = b_1, b_2, b_3, \ldots, b_m$ of random real numbers in the range $[0, 1]$. The random numbers provide the randomization. When the algorithm wants to toss a coin or make a random choice, it does so by using some element from b. For example, to compute the index of the first pivot in quicksort, the algorithm could use the formula $\lfloor nb_1 \rfloor$.

Now, notice that if we fix b to some particular sequence $\hat{b}$ then $\mathcal{R}$ becomes a deterministic sorting algorithm, $\mathcal{R}(\hat{b})$, that has an associated comparison tree, $\mathcal{T}(\hat{b})$. Next, notice that if we select $\mathtt{a}$ to be a random permutation of $\{1, \ldots, \mathtt{n}\}$, then this is equivalent to selecting a random leaf, $\mathtt{w}$, from the $\mathtt{n}!$ leaves of $\mathcal{T}(\hat{b})$.

Exercise 11.5 asks you to prove that, if we select a random leaf from any binary tree with k leaves, then the expected depth of that leaf is at least $\log k$. Therefore, the expected number of comparisons performed by the (deterministic) algorithm $\mathcal{R}(\hat{b})$ when given an input array containing a random permutation of $\{1, \ldots, n\}$ is at least $\log(\mathtt{n}!)$. Finally, notice that this is true for every choice of $\hat{b}$, therefore it holds even for $\mathcal{R}$. This completes the proof of the lower-bound for randomized algorithms.

Theorem 11.6. *For any (deterministic or randomized) comparison-based sorting algorithm $\mathcal{A}$ and any integer $\mathtt{n} \geq 1$, the expected number of comparisons done by $\mathcal{A}$ when sorting a*

random permutation of $\{1, \ldots, n\}$ *is at least* $\log(\mathtt{n}!) = \mathtt{n}\log\mathtt{n} - O(\mathtt{n})$.

11.2 Counting Sort and Radix Sort

In this section we consider two sorting algorithms that are not comparison-based. These algorithms are specialized for sorting small integers. These algorithms get around the lower-bounds of Theorem 11.5 by using (parts of) the elements of `a` as indices into an array. Consider a statement of the form

$$\mathtt{c}[\mathtt{a}[\mathtt{i}]] = 1 \ .$$

This statement executes in constant time, but has `c.length` possible different outcomes, depending on the value of `a[i]`. This means that the execution of an algorithm that makes such a statement can not be modelled as a binary tree. Ultimately, this is the reason that the algorithms in this section are able to sort faster than comparison-based algorithms.

11.2.1 Counting Sort

Suppose we have an input array `a` consisting of `n` integers, each in the range $0, \ldots, \mathtt{k}-1$. The *counting-sort* algorithm sorts `a` using an auxiliary array `c` of counters. It outputs a sorted version of `a` as an auxiliary array `b`.

The idea behind counting-sort is simple: For each $\mathtt{i} \in \{0, \ldots, \mathtt{k}-1\}$, count the number of occurrences of `i` in `a` and store this in `c[i]`. Now, after sorting, the output will look like `c[0]` occurrences of 0, followed by `c[1]` occurrences of 1, followed by `c[2]` occurrences of 2,..., followed by `c[k-1]` occurrences of `k-1`. The code that does this is very slick, and its execution is illustrated in Figure 11.6:

Algorithms

```
void countingSort(array<int> &a, int k) {
  array<int> c(k, 0);
  for (int i = 0; i < a.length; i++)
    c[a[i]]++;
  for (int i = 1; i < k; i++)
    c[i] += c[i-1];
  array<int> b(a.length);
  for (int i = a.length-1; i >= 0; i--)
    b[--c[a[i]]] = a[i];
  a = b;
}
```

The first `for` loop in this code sets each counter `c[i]` so that it counts the number of occurrences of `i` in `a`. By using the values of `a` as indices, these counters can all be computed in $O(\mathtt{n})$ time with a single for loop. At this point, we could use `c` to fill in the

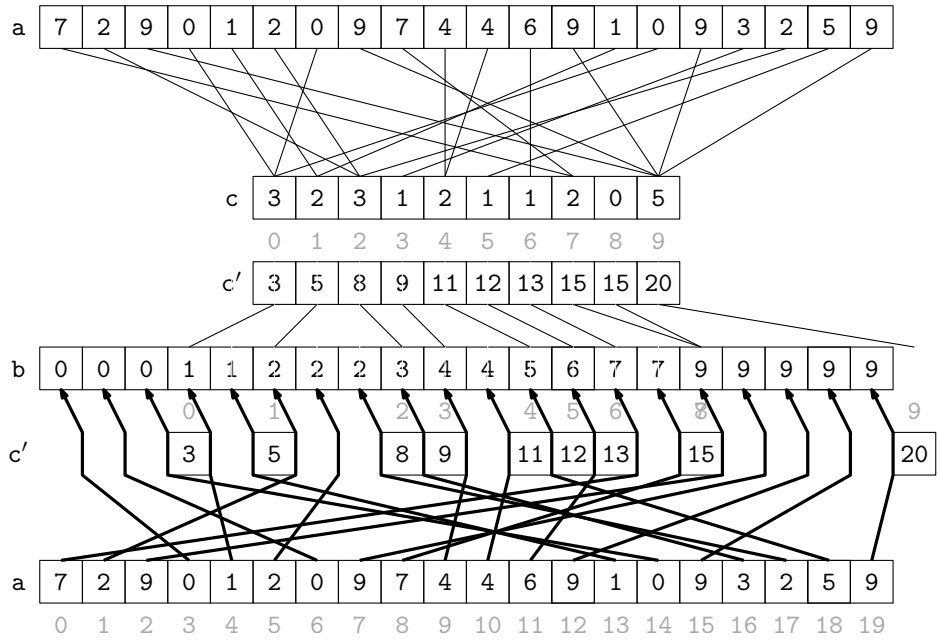


Figure 11.6: The operation of counting sort on an array of length $\texttt{n} = 20$ that stores integers $0, \ldots, \texttt{k} - 1 = 9$.

output array `b` directly. However, this would not work if the elements of `a` have associated data. Therefore we spend a little extra effort to copy the elements of `a` into `b`.

The next `for` loop, which takes $O(\texttt{k})$ time, computes a running-sum of the counters so that `c[i]` becomes the number of elements in `a` that are less than or equal to `i`. In particular, for every $\texttt{i} \in \{0, \ldots, \texttt{k} - 1\}$, the output array, `b`, will have

$$\texttt{b}[\texttt{c}[\texttt{i} - 1]] = \texttt{b}[\texttt{c}[\texttt{i} - 1] + 1] = \cdots = \texttt{b}[\texttt{c}[\texttt{i}] - 1] = \texttt{i} \ .$$

Finally, the algorithm scans `a` backwards to put its elements in order into an output array `b`. When scanning, the element $\texttt{a}[\texttt{i}] = \texttt{j}$ is placed at location $\texttt{b}[\texttt{c}[\texttt{j}] - 1]$ and the value `c[j]` is decremented.

Theorem 11.7. *The* `countingSort(a,k)` *method can sort an array* `a` *containing* `n` *integers in the set* $\{0, \ldots, \texttt{k} - 1\}$ *in* $O(\texttt{n} + \texttt{k})$ *time.*

The counting-sort algorithm has the nice property of being *stable*; it preserves the relative order of elements that are equal. If two elements `a[i]` and `a[j]` have the same value, and $\texttt{i} < \texttt{j}$ then `a[i]` will appear before `a[j]` in `b`. This will be useful in the next section.

11.2.2 Radix-Sort

Counting-sort is very efficient for sorting an array of integers when the length, n, of the array is not much smaller than the maximum value, $k-1$, that appears in the array. The *radix-sort* algorithm, which we now describe, uses several passes of counting-sort to allow for a much greater range of maximum values.

Radix-sort sorts w-bit integers by using w/d passes of counting-sort to sort these integers d bits at a time.[2] More precisely, radix sort first sorts the integers by their least significant d bits, then their next significant d bits, and so on until, in the last pass, the integers are sorted by their most significant d bits.

Algorithms

```
void radixSort(array<int> &a) {
  int d = 8, w = 32;
  for (int p = 0; p < w/d; p++) {
    array<int> c(1<<d, 0);
    // the next three for loops implement counting-sort
    array<int> b(a.length);
    for (int i = 0; i < a.length; i++)
      c[(a[i] >> d*p)&((1<<d)-1)]++;
    for (int i = 1; i < 1<<d; i++)
      c[i] += c[i-1];
    for (int i = a.length-1; i >= 0; i--)
      b[--c[(a[i] >> d*p)&((1<<d)-1)]] = a[i];
    a = b;
  }
}
```

(In this code, the expression $(\texttt{a[i]} >> \texttt{d} * \texttt{p})\&((1 << \texttt{d}) - 1)$ extracts the integer whose binary representation is given by bits $(\texttt{p}+1)\texttt{d}-1, \ldots, \texttt{pd}$ of a[i].) An example of the steps of this algorithm is shown in Figure 11.7.

This remarkable algorithm sorts correctly because counting-sort is a stable sorting algorithm. If $x < y$ are two elements of a and the most significant bit at which x differs from y has index r, then x will be placed before y during pass $\lfloor r/\texttt{d} \rfloor$ and subsequent passes will not change the relative order of x and y.

Radix-sort performs w/d passes of counting-sort. Each pass requires $O(\texttt{n}+2^{\texttt{d}})$ time. Therefore, the performance of radix-sort is given by the following theorem.

Theorem 11.8. *For any integer* $\texttt{d} > 0$*, the* radixSort(a, k) *method can sort an array* a *containing* n w*-bit integers in* $O((\texttt{w}/\texttt{d})(\texttt{n}+2^{\texttt{d}}))$ *time.*

[2] We assume that d divides w, otherwise we can always increase w to $\texttt{d}\lceil \texttt{w}/\texttt{d} \rceil$.

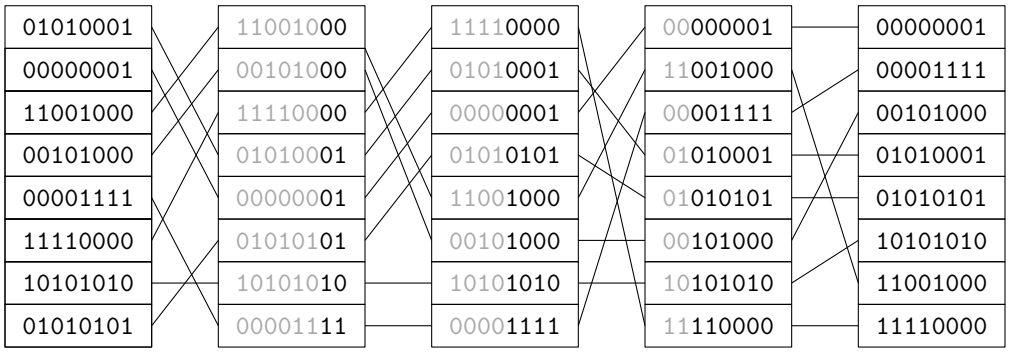


Figure 11.7: Using radixsort to sort $\mathtt{w} = 8$-bit integers by using 4 passes of counting sort on $\mathtt{d} = 2$-bit integers.

If we think, instead, of the elements of the array being in the range $\{0, \ldots, \mathtt{n}^c - 1\}$, and take $\mathtt{d} = \lceil \log \mathtt{n} \rceil$ we obtain the following version of Theorem 11.8.

Corollary 11.1. *The* `radixSort(a,k)` *method can sort an array* `a` *containing* `n` *integer values in the range* $\{0, \ldots, \mathtt{n}^c - 1\}$ *in* $O(c\mathtt{n})$ *time.*

11.3 Discussion and Exercises

Sorting is probably *the* fundamental algorithmic problem in computer science, and has a long history. Knuth [38] attributes the merge-sort algorithm to von Neumann (1945). Quicksort is due to Hoare [32]. The original heap-sort algorithm is due to Williams [56], but the version presented here (in which the heap is constructed bottom-up in $O(\mathtt{n})$ time) is due to Floyd [23]. Lower-bounds for comparison-based sorting appear to be folklore. The following table summarizes the performance of these comparison-based algorithms:

	comparisons		in-place
Merge-sort	$\mathtt{n}\log\mathtt{n}$	worst-case	No
Quicksort	$1.38\mathtt{n}\log\mathtt{n} + O(\mathtt{n})$	expected	Yes
Heap-sort	$2\mathtt{n}\log\mathtt{n} + O(\mathtt{n})$	worst-case	Yes

Each of these comparison-based algorithms has advantages and disadvantages. Merge-sort does the fewest comparisons and does not rely on randomization. Unfortunately, it uses an auxilliary array during its merge phase. Allocating this array can be expensive and is a potential point of failure if memory is limited. Quicksort is an in-place algorithm and is a close second in terms of the number of comparisons, but is randomized so this running

time is not always guaranteed. Heap-sort does the most comparisons, but it is in-place and deterministic.

There is one setting in which merge-sort is a clear-winner; this occurs when sorting a linked-list. In this case, the auxiliary array is not needed; two sorted linked lists are very easily merged into a single sorted linked-list by pointer manipulations.

The counting-sort and radix-sort algorithms described here are due to Seward [51, Section 2.4.6]. However, variants of radix-sort have been used since the 1920's to sort punch cards using punched card sorting machines. These machines can sort a stack of cards into two piles based on the existence (or not) of a hole in a specific location on the card. Repeating this process for different hole locations gives an implementation of radix-sort.

Exercise 11.1. Find another pair of permutations of $1, 2, 3$ that are not correctly sorted by the comparison-tree in Figure 11.5.

Exercise 11.2. Implement a version of the merge-sort algorithm that sorts a `DLList` without using an auxiliary array.

Exercise 11.3. Prove that $\log \mathtt{n}! = \mathtt{n} \log \mathtt{n} - O(\mathtt{n})$.

Exercise 11.4. Prove that a binary tree with k leaves has height at least $\log k$.

Exercise 11.5. Prove that, if we pick a random leaf from a binary tree with k leaves, then the expected height of this leaf is at least $\log k$. (Hint: Use induction along with the inequality $(k_1/k)\log k_1 + (k_2/k)\log k_2) \geq \log k - 1$, when $k_1 + k_2 = k$.)

Exercise 11.6. Simple floating point numbers are numbers of the form $x \cdot 10^y$, where $0 \leq x \leq 1$ and y is an integer and each of x and y can be represented by at most k bits. Describe a version of radix-sort that can be used to sort simple floating point numbers.

Extend your version of radix sort so that it can handle signed values of both x and y and implement a version of radix-sort that sorts arrays of type `float`.

Bibliography

[1] Free eBooks by Project Gutenberg. Available from: `http://www.gutenberg.org/` [cited 2011-10-12].

[2] G.M. Adelson-Velskii and E.M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(1259-1262):4, 1962.

[3] A. Andersson. Improving partial rebuilding by using simple balance criteria. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989.

[4] A. Andersson. Balanced search trees made simple. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11-13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1993.

[5] A. Andersson. General balanced trees. *Journal of Algorithms*, 30(1):1–18, 1999.

[6] Amitabha Bagchi, Adam L. Buchsbaum, and Michael T. Goodrich. Biased skip lists. In Prosenjit Bose and Pat Morin, editors, *Algorithms and Computation, 13th International Symposium, ISAAC 2002 Vancouver, BC, Canada, November 21-23, 2002, Proceedings*, volume 2518 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2002.

[7] Bibliography on hashing. Available from: `http://liinwww.ira.uka.de/bibliography/Theory/hash.html` [cited 2011-07-20].

[8] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara,*

California, USA, August 15-19, 1999, Proceedings, volume 1666 of *Lecture Notes in Computer Science*, pages 79–79. Springer, 1999.

[9] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In Shang-Hua Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 1106–1114. SIAM, 2008.

[10] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In Dehne et al. [13], pages 37–48.

[11] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[12] C.A. Crane. Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Computer Science Department, Stanford University, 1972.

[13] Frank K. H. A. Dehne, Arvind Gupta, Jörg-Rüdiger Sack, and Roberto Tamassia, editors. *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*, volume 1663 of *Lecture Notes in Computer Science*. Springer, 1999.

[14] L. Devroye. Applications of the theory of records in the study of random trees. *Acta Informatica*, 26(1):123–130, 1988.

[15] P. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In John R. Gilbert and Rolf G. Karlsson, editors, *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 11-14, 1990, Proceedings*, volume 447 of *Lecture Notes in Computer Science*, pages 173–180. Springer, 1990.

[16] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In Claude Puech and Rüdiger Reischuk, editors, *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science, Grenoble, France, February 22-24, 1996, Proceedings*, volume 1046 of *Lecture Notes in Computer Science*, pages 567–580. Springer, 1996.

[17] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In Werner Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 1992.

[18] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[19] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[20] A. Elmasry. Pairing heaps with $O(\log \log n)$ decrease cost. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 471–476. Society for Industrial and Applied Mathematics, 2009.

[21] F. Ergun, S. C. Sahinalp, J. Sharp, and R. Sinha. Biased dictionaries with fast insert/deletes. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 483–491, New York, NY, USA, 2001. ACM.

[22] M. Eytzinger. *Thesaurus principum hac aetate in Europa viventium (Cologne)*. 1590. In commentaries, 'Eytzinger' may appear in variant forms, including: Aitsingeri, Aitsingero, Aitsingerum, Eyzingern.

[23] R. W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7(12):701, 1964.

[24] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[25] M.L. Fredman, R. Sedgewick, D.D. Sleator, and R.E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[26] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[27] I. Galperin and R.L. Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1993.

[28] A. Gambin and A. Malinowski. Randomized meldable priority queues. In *SOFSEM98: Theory and Practice of Informatics*, pages 344–349. Springer, 1998.

[29] M. T. Goodrich and J. G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Dehne et al. [13], pages 205–216.

[30] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.

[31] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, 16-18 October 1978, Proceedings*, pages 8–21. IEEE Computer Society, 1978.

[32] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[33] HP-UX process management white paper, version 1.3, 1997. Available from: `http://h21007.www2.hp.com/portal/download/files/prot/files/STK/pdfs/proc_mgt.pdf` [cited 2011-07-20].

[34] P. Kirschenhofer, C. Martinez, and H. Prodinger. Analysis of an optimized search algorithm for skip lists. *Theoretical Computer Science*, 144:199–220, 1995.

[35] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 31:775–792, 1994.

[36] D. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[37] D. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[38] D. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[39] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA'92)*, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[40] Oracle. *The Collections Framework*. Available from: `http://download.oracle.com/javase/1.5.0/docs/guide/collections/` [cited 2011-07-19].

[41] R. Pagh and F.F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[42] T. Papadakis, J. I. Munro, and P. V. Poblete. Average search and update costs in skip lists. *BIT*, 32:316–332, 1992.

[43] M. Patrascu and M. Thorup. The power of simple tabulation hashing, 2010. `arXiv:1011.5200`.

[44] W. Pugh. A skip list cookbook. Technical report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, 1989. Available from: `ftp://ftp.cs.umd.edu/pub/skipLists/cookbook.pdf` [cited 2011-07-20].

[45] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[46] Redis. Available from: `http://redis.io/` [cited 2011-07-20].

[47] B. Reed. The height of a random binary search tree. *Journal of the ACM*, 50(3):306–332, 2003.

[48] S. M. Ross. *Probability Models for Computer Science*. Academic Press, Inc., Orlando, FL, USA, 2001.

[49] R. Sedgewick. Left-leaning red-black trees, September 2008. Available from: `http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf` [cited 2011-07-21].

[50] R. Seidel and C.R. Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

[51] H. H. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, Massachusetts Institute of Technology, Digital Computer Laboratory, 1954.

[52] Skipdb. Available from: `http://dekorte.com/projects/opensource/SkipDB/` [cited 2011-07-20].

[53] D.D. Sleator and R.E. Tarjan. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 235–245. ACM, ACM, 1983.

[54] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[55] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[56] J.W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.