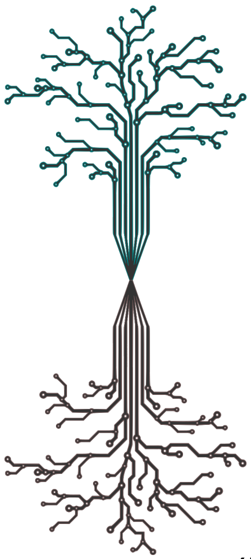


Efficient Data Structures (in pseudocode)

Compiled: Monday 2nd September, 2019

Taylor

Contents

Acknowledgments

This is a remix of the materials generously opened at:
http://opendatastructures.org
and made possible by the far-sighted UM-AOER efforts:
https://aoer.mst.edu/

Chapter 1

Introduction

Every computer science curriculum in the world includes a course on data structures and algorithms. Data structures are *that* important; they improve our quality of life and even save lives on a regular basis. Many multi-million and several multi-billion dollar companies have been built around data structures.

How can this be? If we stop to think about it, we realize that we interact with data structures constantly.

- Open a file: File system data structures are used to locate the parts of that file on disk so they can be retrieved. This isn't easy; disks contain hundreds of millions of blocks. The contents of your file could be stored on any one of them.

- Look up a contact on your phone: A data structure is used to look up a phone number in your contact list based on partial information even before you finish dialing/typing. This isn't easy; your phone may contain information about a lot of people—everyone you have ever contacted via phone or email—and your phone doesn't have a very fast processor or a lot of memory.

- Log in to your favourite social network: The network servers use your login information to look up your account information. This isn't easy; the most popular social networks have hundreds of millions of active users.

- Do a web search: The search engine uses data structures to find the web pages containing your search terms. This isn't easy; there are over 8.5 billion web pages on the Internet and each page contains a lot of potential search terms.

- Phone emergency services (9-1-1): The emergency services network looks up your phone number in a data structure that maps phone numbers to addresses so that police cars, ambulances, or fire trucks can be sent there without delay. This is important; the person making the call may not be able to provide the exact address they are calling from and a delay can mean the difference between life or death.

1.1 The Need for Efficiency

In the next section, we look at the operations supported by the most commonly used data structures. Anyone with a bit of programming experience will see that these operations are not hard to implement correctly. We can store the data in an array or a linked list and each operation can be implemented by iterating over all the elements of the array or list and possibly adding or removing an element.

This kind of implementation is easy, but not very efficient. Does this really matter? Computers are becoming faster and faster. Maybe the obvious implementation is good enough. Let's do some rough calculations to find out.

Number of operations: Imagine an application with a moderately-sized data set, say of one million (10^6), items. It is reasonable, in most applications, to assume that the application will want to look up each item at least once. This means we can expect to do at least one million (10^6) searches in this data. If each of these 10^6 searches inspects each of the 10^6 items, this gives a total of $10^6 \times 10^6 = 10^{12}$ (one thousand billion) inspections.

Processor speeds: At the time of writing, even a very fast desktop computer can not do more than one billion (10^9) operations per second.[1] This means that this application will take at least $10^{12}/10^9 = 1000$ seconds, or roughly 16 minutes and 40 seconds. Sixteen minutes is an eon in computer time, but a person might be willing to put up with it (if he or she were headed out for a coffee break).

Bigger data sets: Now consider a company like Google, that indexes over 8.5 billion web pages. By our calculations, doing any kind of query over this data would take at least 8.5 seconds. We already know that this isn't the case; web searches complete in much less than 8.5 seconds, and they do much more complicated queries than just asking if a particular page is in their list of indexed pages. At the time of writing, Google receives approximately $4,500$ queries per second, meaning that they would require at least $4,500 \times 8.5 = 38,250$ very fast servers just to keep up.

The solution: These examples tell us that the obvious implementations of data structures do not scale well when the number of items, n, in the data structure and the number of operations, m, performed on the data structure are both large. In these cases, the time (measured in, say, machine instructions) is roughly $n \times m$.

The solution, of course, is to carefully organize data within the data structure so that not every operation requires every data item to be inspected. Although it sounds impossible at first, we will see data structures where a search requires looking at only

[1] Computer speeds are at most a few gigahertz (billions of cycles per second), and each operation typically takes a few cycles.

two items on average, independent of the number of items stored in the data structure. In our billion instruction per second computer it takes only 0.000000002 seconds to search in a data structure containing a billion items (or a trillion, or a quadrillion, or even a quintillion items).

We will also see implementations of data structures that keep the items in sorted order, where the number of items inspected during an operation grows very slowly as a function of the number of items in the data structure. For example, we can maintain a sorted set of one billion items while inspecting at most 60 items during any operation. In our billion instruction per second computer, these operations take 0.00000006 seconds each.

The remainder of this chapter briefly reviews some of the main concepts used throughout the rest of the book. Section 1.2 describes the interfaces implemented by all of the data structures described in this book and should be considered required reading. The remaining sections discuss:

- some mathematical review including exponentials, logarithms, factorials, asymptotic (big-Oh) notation, probability, and randomization;
- the model of computation;
- correctness, running time, and space;
- an overview of the rest of the chapters; 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.

1.2 Interfaces

When 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 a 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

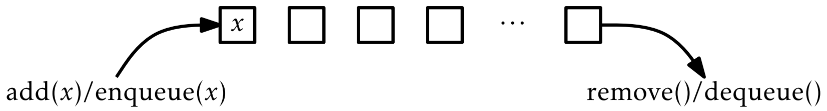


Figure 1.1: A FIFO Queue.

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.

1.2.1 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

- $\mathrm{add}(x)$: add the value x to the Queue
- $\mathrm{remove}()$: remove the next (previously added) value, y, from the Queue and return y

Notice that the $\mathrm{remove}()$ 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*, which is illustrated in Figure 1.1, 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. This is the most common kind of Queue so the qualifier FIFO is often omitted. In other texts, the $\mathrm{add}(x)$ and $\mathrm{remove}()$ operations on a FIFO Queue are often called $\mathrm{enqueue}(x)$ and $\mathrm{dequeue}()$, respectively.

A *priority Queue*, illustrated in Figure 1.2, always removes the smallest element from the Queue, breaking ties arbitrarily. This is similar to the way in which patients are triaged in a hospital emergency room. As patients arrive they are evaluated and then placed in a waiting room. When a doctor becomes available he or she first treats the patient with the most life-threatening condition. The $\mathrm{remove}()$ operation on a priority Queue is usually called $\mathrm{delete\_min}()$ in other texts.

A very common queueing discipline is the LIFO (last-in-first-out) discipline, illustrated in Figure 1.3. 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 $\mathrm{add}(x)$ and $\mathrm{remove}()$ are changed to $\mathrm{push}(x)$ and $\mathrm{pop}()$; this is to avoid confusing the LIFO and FIFO queueing disciplines.

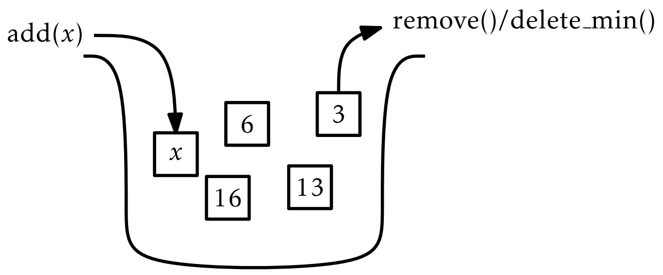


Figure 1.2: A priority Queue.

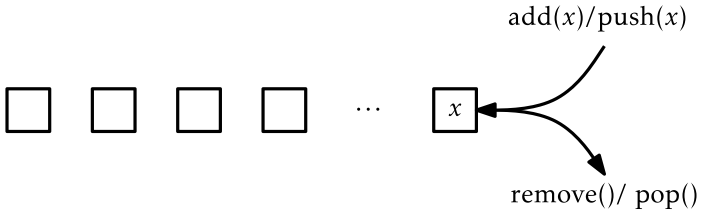


Figure 1.3: A stack.

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 Deque operations are self-explanatory: $\mathrm{add\_first}(x)$, $\mathrm{remove\_first}()$, $\mathrm{add\_last}(x)$, and $\mathrm{remove\_last}()$. It is worth noting that a Stack can be implemented using only $\mathrm{add\_first}(x)$ and $\mathrm{remove\_first}()$ while a FIFO Queue can be implemented using $\mathrm{add\_last}(x)$ and $\mathrm{remove\_first}()$.

1.2.2 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, illustrated in Figure 1.4, represents a sequence, $x_0,\ldots,x_{n-1}$, of values. The List interface includes the following operations:

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

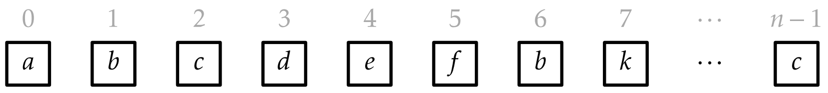


Figure 1.4: A List represents a sequence indexed by $0,1,2,\ldots,n-1$. In this List a call to $\mathrm{get}(2)$ would return the value c.

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

$$\begin{aligned}
\mathrm{add\_first}(x) &\Rightarrow \mathrm{add}(0,x)\\
\mathrm{remove\_first}() &\Rightarrow \mathrm{remove}(0)\\
\mathrm{add\_last}(x) &\Rightarrow \mathrm{add}(\mathrm{size}(),x)\\
\mathrm{remove\_last}() &\Rightarrow \mathrm{remove}(\mathrm{size}()-1)
\end{aligned}$$

Although we will normally not discuss the Stack, Deque and FIFO Queue interfaces 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 highlights 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 implements all the Deque operations in constant time per operation.

1.2.3 The USet Interface: Unordered Sets

The USet interface represents an unordered set of unique elements, which mimics a mathematical *set*. 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. $\mathrm{size}()$: return the number, n, of elements in the set
2. $\mathrm{add}(x)$: 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. $\mathrm{remove}(x)$: remove x from the set;
   Find an element y in the set such that x equals y and remove y. Return y, or *nil* if no such element exists.
4. $\mathrm{find}(x)$: find x in the set if it exists;
   Find an element y in the set such that y equals x. Return y, or *nil* 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 may remove or find. This is because x and y might actually be distinct objects that are nevertheless treated as equal. Such a distinction is useful because it allows for the creation of *dictionaries* or *maps* that map keys onto values.

To create a dictionary/map, one forms compound objects called Pairs, each of which contains a *key* and a *value*. Two Pairs are treated as equal if their keys are equal. If we store some pair (k,v) in a USet and then later call the $\mathrm{find}(x)$ method using the pair $x = (k, nil)$ the result will be $y = (k,v)$. In other words, it is possible to recover the value, v, given only the key, k.

1.2.4 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 $\mathrm{compare}(x,y)$ in which

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

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

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

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

The distinction between the USet and SSet $\mathrm{find}(x)$ operations is very important and 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, most of the SSet implementations discussed in this book all have $\mathrm{find}(x)$ operations with running times that are logarithmic in the size of the set. On the other hand, the implementation of a USet as a ChainedHashTable in Chapter 7 has a $\mathrm{find}(x)$ 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 truly needed.

1.3 Mathematical Background

In this section, we review some mathematical notations and tools used throughout this book, including logarithms, big-Oh notation, and probability theory. This review will be brief and is not intended as an introduction. Readers who feel they are missing this background are encouraged to read, and do exercises from, the appropriate sections of the very good (and free) textbook on mathematics for computer science [29].

1.3.1 Exponentials and Logarithms

The expression b^x denotes the number b raised to the power of x. If x is a positive integer, then this is just the value of b multiplied by itself $x-1$ times:

$$b^x = \underbrace{b \times b \times \cdots \times b}_{x} \enspace .$$

When x is a negative integer, $b^{-x} = 1/b^x$. When $x = 0$, $b^x = 1$. When b is not an integer, we can still define exponentiation in terms of the exponential function e^x (see below), which is itself defined in terms of the exponential series, but this is best left to a calculus text.

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

$$b^x = k \enspace .$$

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

An informal, but useful, way to think about logarithms is to think of $\log_b k$ as the number of times we have to divide k by b before the result is less than or equal to 1. For example, when one does binary search, each comparison reduces the number of possible answers by a factor of 2. This is repeated until there is at most one possible answer. Therefore, the number of comparison done by binary search when there are initially at most $n+1$ possible answers is at most $\lceil \log_2(n+1) \rceil$.

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

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

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

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

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:

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

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

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

1.3.2 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") is defined to mean

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

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 $\{1,\ldots,n\}$.

1.3.3 Asymptotic Notation

When analyzing data structures in this book, we 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. When we talk about the running time of an operation we are referring to the number of computer instructions performed during the operation. Even for simple code, this quantity can be difficult to compute exactly. 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,

$$O(f(n)) = \left\{ \begin{array}{c} g(n) : \text{there exists } c > 0 \text{, and } n_0 \text{ such that} \\ g(n) \le c \cdot f(n) \text{ for all } n \ge n_0 \end{array} \right\} .$$

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 $O(n\log n)$. This is proven as follows:

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

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

A number of useful shortcuts can be applied 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 occurs when we write statements like

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

Again, this would be more correctly written as

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

The expression $O(1)$ also brings up another issue. Since there is no variable in this expression, it may not be clear which 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 $f(n) = 1$.

Big-Oh notation is not new or unique to computer science. It was used by the number theorist Paul Bachmann as early as 1894, and is immensely useful for describing the running times of computer algorithms. Consider the following piece of code: One execution of this method involves

- 1 assignment ($int\, i \leftarrow 0$),
- $n+1$ comparisons ($i < n$),
- n increments ($i++$),
- n array offset calculations ($a[i]$), and
- n indirect assignments ($a[i] \leftarrow i$).

So we could write this running time as

$$T(n) = a + b(n+1) + cn + dn + en \ ,$$

where a, b, c, d, and e are constants that depend on the machine running the code and represent the time to perform assignments, comparisons, increment operations, array offset calculations, and indirect assignments, respectively. However, if this expression represents the running time of two lines of code, then clearly this kind of analysis will not be tractable to complicated code or algorithms. Using big-Oh notation, the running time can be simplified to

$$T(n) = O(n) \ .$$

Not only is this more compact, but it also gives nearly as much information. The fact that the running time depends on the constants a, b, c, d, and e in the above example means that, in general, it will not be possible to compare two running times to know which is faster without knowing the values of these constants. Even if we make the effort to determine these constants (say, through timing tests), then our conclusion will only be valid for the machine we run our tests on.

Big-Oh notation allows us to reason at a much higher level, making it possible to analyze more complicated functions. If two algorithms have the same big-Oh running time, then we won't know which is faster, and there may not be a clear winner. One may be faster on one machine, and the other may be faster on a different machine. However, if the two algorithms have demonstrably different big-Oh running times, then we can be certain that the one with the smaller running time will be faster *for large enough values of n*.

An example of how big-Oh notation allows us to compare two different functions is shown in Figure 1.5, which compares the rate of growth of $f_1(n) = 15n$ versus $f_2(n) = 2n\log n$. It might be that $f_1(n)$ is the running time of a complicated linear time algorithm while $f_2(n)$ is the running time of a considerably simpler algorithm based on the divide-and-conquer paradigm. This illustrates that, although $f_1(n)$ is greater than $f_2(n)$ for small values of n, the opposite is true for large values of n. Eventually $f_1(n)$ wins out, by an increasingly wide margin. Analysis using big-Oh notation told us that this would happen, since $O(n) \subset O(n\log n)$.

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:

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

This definition captures the situation we really care about: when the arguments $n_1,\ldots,n_k$ make g take on large values. This definition also 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.

1.3.4 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.

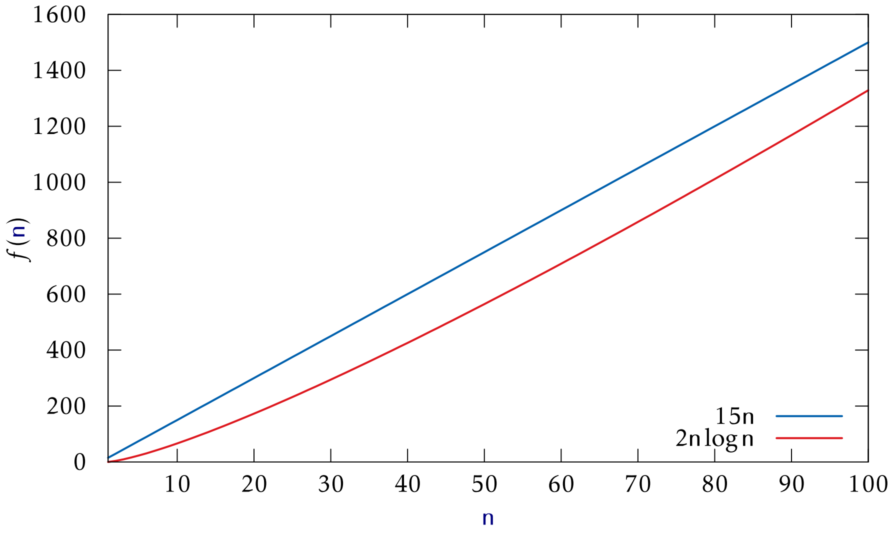
Formally, the running time of an operation on a randomized data structure is a random variable, and we want to study its *expected value*. For a discrete random variable X taking on values in some countable universe U, the expected value of X, denoted by $\mathrm{E}[X]$, is given by the formula

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

Here $\Pr\{\mathcal{E}\}$ denotes the probability that the event $\mathcal{E}$ occurs. In all of the examples in this book, these probabilities are only with respect to the random choices made by the randomized data structure; there is no assumption that the data stored in the structure, nor 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,

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



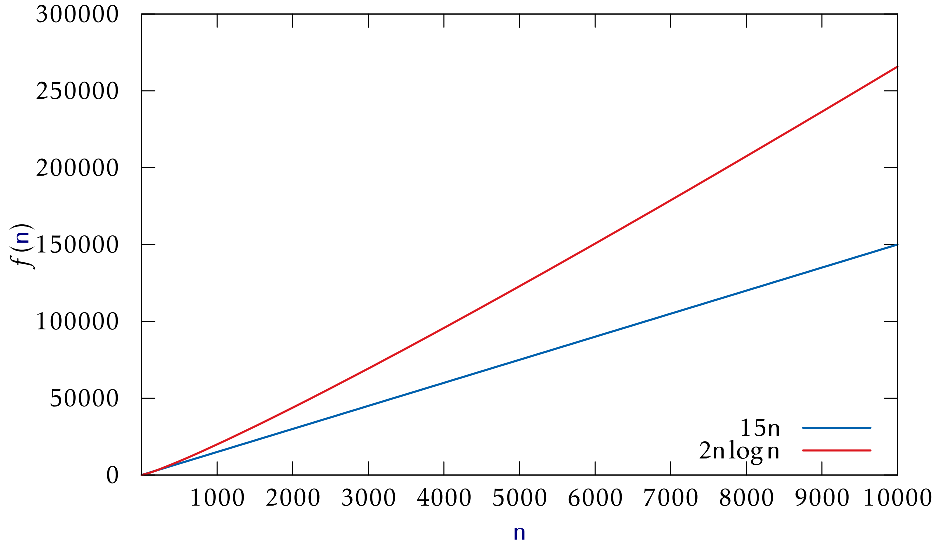


Figure 1.5: Plots of $15n$ versus $2n \log n$.

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

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

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 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 turns up as heads. Intuitively, we know the answer is $k/2$, but if we try to prove it using the definition of expected value, we get

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

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

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

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

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

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

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 turn up as heads precisely because each individual coin turns up as heads with a probability of $1/2$.

1.4 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 *w-bit word-RAM* model. RAM stands for Random Access Machine. In this model, we have access to a random access memory consisting of *cells*, each of which stores a w-bit *word*. This implies that a memory cell can represent, for example, any integer in the set $\{0,\ldots,2^w-1\}$.

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

Any cell can be read or written in constant time. A 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 would like. Allocating a block of memory of size k takes $O(k)$ time and returns a reference (a pointer) to the newly-allocated memory block. This reference is small enough to be represented by a single word.

The word-size w is a very important parameter of this model. The only assumption we will make about w is the lower-bound $w \geq \log n$, where 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 of our data structures store values of a generic type T, and we assume an element of type T occupies one word of memory.

The data structures presented in this book don't use any special tricks that are not implementable

1.5 Correctness, Time Complexity, and Space Complexity

When studying the performance of a data structure, there are three things that matter most:

Correctness: The data structure should correctly implement its interface.

Time complexity: The running times of operations on the data structure should be as small as possible.

Space complexity: The data structure should use as little memory as possible.

In this introductory text, we will take correctness as a given; we won't consider data structures that give incorrect answers to queries or don't perform updates properly. We will, however, see data structures that make an extra effort to keep space usage to a minimum. This won't usually affect the (asymptotic) running times of operations, but can make the data structures a little slower in practice.

When studying running times in the context of data structures we tend to come across three different kinds of running time guarantees:

Worst-case running times: These are the strongest kind of running time guarantees. If a data structure operation has a worst-case running time of $f(n)$, then one of these operations *never* takes longer than $f(n)$ time.

Amortized running times: If we say that the amortized running time of an operation in a data structure is $f(n)$, then this means that the cost of a typical operation is at most $f(n)$. More precisely, if a data structure has an amortized running time of $f(n)$, then a sequence of m operations takes at most $mf(n)$ time. Some individual operations may take more than $f(n)$ time but the average, over the entire sequence of operations, is at most $f(n)$.

Expected running times: If we say that the expected running time of an operation on a data structure is $f(n)$, this means that the actual running time is a random variable (see Section 1.3.4) and the expected value of this random variable is at most $f(n)$. The randomization here is with respect to random choices made by the data structure.

To understand the difference between worst-case, amortized, and expected running times, it helps to consider a financial example. Consider the cost of buying a house:

Worst-case versus amortized cost: Suppose that a home costs \$120 000. In order to buy this home, we might get a 120 month (10 year) mortgage with monthly payments of \$1 200 per month. In this case, the worst-case monthly cost of paying this mortgage is \$1 200 per month.

If we have enough cash on hand, we might choose to buy the house outright, with one payment of \$120 000. In this case, over a period of 10 years, the amortized monthly cost of buying this house is

$$\$120\,000/120 \text{ months} = \$1\,000 \text{ per month} .$$

This is much less than the \$1 200 per month we would have to pay if we took out a mortgage.

Worst-case versus expected cost: Next, consider the issue of fire insurance on our \$120 000 home. By studying hundreds of thousands of cases, insurance companies have determined that the expected amount of fire damage caused to a home like ours is \$10 per month. This is a very small number, since most homes never have fires, a few homes may have some small fires that cause a bit of smoke damage, and a tiny number of homes burn right to their foundations. Based on this information, the insurance company charges \$15 per month for fire insurance.

Now it's decision time. Should we pay the \$15 worst-case monthly cost for fire insurance, or should we gamble and self-insure at an expected cost of \$10 per month? Clearly, the \$10 per month costs less *in expectation*, but we have to be able to accept the possibility that the *actual cost* may be much higher. In the unlikely event that the entire house burns down, the actual cost will be \$120 000.

These financial examples also offer insight into why we sometimes settle for an amortized or expected running time over a worst-case running time. It is often possible to get a lower expected or amortized running time than a worst-case running time. At the very least, it is very often possible to get a much simpler data structure if one is willing to settle for amortized or expected running times.

1.6 Code Samples

The code samples in this book are written in pseudocode. These should be easy enough to read for anyone who has any programming experience in any of the most common programming languages of the last 40 years. To get an idea of what the pseudocode in this book looks like, here is a function that computes the average of an array, a:

```
average($a$)
  $s \leftarrow 0$
  for $i$ in $0, 1, 2, \ldots, \mathrm{length}(a) - 1$ do
    $s \leftarrow s + a[i]$
  return $s/\mathrm{length}(a)$
```

As this code illustrates, assigning to a variable is done using the $\leftarrow$ notation. We use the convention that the length of an array, a, is denoted by $\mathrm{length}(a)$ and array indices start at zero, so that $0, 1, 2, \ldots, \mathrm{length}(a) - 1$ are the valid indices for a. To shorten code, and sometimes make it easier to read, our pseudocode allows for (sub)array assignments. The following two functions are equivalent:

```
left_shift_a($a$)
    for $i$ in $0,1,2,\ldots,\mathrm{length}(a)-2$ do
        $a[i] \leftarrow a[i+1]$
    $a[\mathrm{length}(a)-1] \leftarrow nil$

left_shift_b($a$)
    $a[0,1,\ldots,\mathrm{length}(a)-2] \leftarrow a[1,2,\ldots,\mathrm{length}(a)-1]$
    $a[\mathrm{length}(a)-1] \leftarrow nil$
```

The following code sets all the values in an array to zero:

```
zero($a$)
    $a[0,1,\ldots,\mathrm{length}(a)-1] \leftarrow 0$
```

When analyzing the running time of code like this, we have to be careful; statements like $a[0,1,\ldots,\mathrm{length}(a)-1] \leftarrow 1$ or $a[1,2,\ldots,\mathrm{length}(a)-1] \leftarrow a[0,1,\ldots,\mathrm{length}(a)-2]$ do not run in constant time. They run in $O(\mathrm{length}(a))$ time.

We take similar shortcuts with variable assignments, so that the code $x,y \leftarrow 0,1$ sets x to zero and y to 1 and the code $x,y \leftarrow y,x$ swaps the values of the variables x and y.

Our pseudocode uses a few operators that may be unfamiliar. As is standard in mathematics, (normal) division is denoted by the / operator. In many cases, we want to do integer division instead, in which case we use the div operator, so that $a \,\mathrm{div}\, b = \lfloor a/b \rfloor$ is the integer part of a/b. So, for example, $3/2 = 1.5$ but $3 \,\mathrm{div}\, 2 = 1$. Occasionally, we also use the $\bmod$ operator to obtain the remainder from integer division, but this will be defined when it is used. Later in the book, we may use some bitwise operators including left-shift ($\ll$), right shift ($\gg$), bitwise and ($\wedge$), and bitwise exclusive-or ($\oplus$).

The pseudocode samples in this book are machine translations of Python code that can be downloaded from the book's website.[2] If you ever encounter an ambiguity in the pseudocode that you can't resolve yourself, then you can always refer to the corresponding Python code. If you don't read Python, the code is also available in Java and C++. If you can't decipher the pseudocode, or read Python, C++, or Java, then you may not be ready for this book.

[2]http://opendatastructures.org

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

USet implementations			
	find(x)	add(x) / remove(x)	
ChainedHashTable	$O(1)^{\mathsf{E}}$	$O(1)^{\mathsf{A,E}}$	§ 7.1
LinearHashTable	$O(1)^{\mathsf{E}}$	$O(1)^{\mathsf{A,E}}$	§ 7.2

$^{\mathsf{A}}$ Denotes an *amortized* running time.
$^{\mathsf{E}}$ Denotes an *expected* running time.

Table 1.1: Summary of List and USet implementations.

1.7 List of Data Structures

Tables 1.1 and 1.2 summarize the performance of data structures in this book that implement each of the interfaces, List, USet, and SSet, described in Section 1.2.

1.8 Discussion and Exercises

The List, USet, and SSet interfaces described in Section 1.2 are influenced by the Java Collections Framework [31]. These are essentially simplified versions of the List, Set, Map, SortedSet, and SortedMap interfaces found in the Java Collections Framework.

For a superb (and free) treatment of the mathematics discussed in this chapter, including asymptotic notation, logarithms, factorials, Stirling's approximation, basic probability, and lots more, see the textbook by Leyman, Leighton, and Meyer [29]. For a gentle calculus text that includes formal definitions of exponentials and logarithms, see the (freely available) classic text by Thompson [39].

For more information on basic probability, especially as it relates to computer science, see the textbook by Ross [34]. Another good reference, which covers both asymp-

SSet implementations			
	$\mathrm{find}(x)$	$\mathrm{add}(x)$ / $\mathrm{remove}(x)$	
Binary Tree	$O(\log w)$A,E	O(1) / $O(\log w)$A,E	§ 4

(Priority) Queue implementations			
	$\mathrm{find\_min}()$	$\mathrm{add}(x)$ / $\mathrm{remove}()$	
BinaryHeap	$O(1)$	$O(\log n)$A	§ 5.1
MeldableHeap	$O(1)$	$O(\log n)$E	§ 5.2

I This structure can only store w-bit integer data.

Table 1.2: Summary of SSet and priority Queue implementations.

totic notation and probability, is the textbook by Graham, Knuth, and Patashnik [21].

Exercise 1.1. This exercise is designed to help familiarize the reader with choosing the right data structure for the right problem. If implemented, the parts of this exercise should be done by making use of an implementation of the relevant interface (Stack, Queue, Deque, USet, or SSet) provided by the .

Solve the following problems by reading a text file one line at a time and performing operations on each line in the appropriate data structure(s). Your implementations should be fast enough that even files containing a million lines can be processed in a few seconds.

1. Read the input one line at a time and then write the lines out in reverse order, so that the last input line is printed first, then the second last input line, and so on.

2. Read the first 50 lines of input and then write them out in reverse order. Read the next 50 lines and then write them out in reverse order. Do this until there are no more lines left to read, at which point any remaining lines should be output in reverse order.

   In other words, your output will start with the 50th line, then the 49th, then the 48th, and so on down to the first line. This will be followed by the 100th line, followed by the 99th, and so on down to the 51st line. And so on.

   Your code should never have to store more than 50 lines at any given time.

3. Read the input one line at a time. At any point after reading the first 42 lines, if some line is blank (i.e., a string of length 0), then output the line that occured 42 lines prior to that one. For example, if Line 242 is blank, then your program should output line 200. This program should be implemented so that it never stores more than 43 lines of the input at any given time.

4. Read the input one line at a time and write each line to the output if it is not a duplicate of some previous input line. Take special care so that a file with a lot of

duplicate lines does not use more memory than what is required for the number of unique lines.

5. Read the input one line at a time and write each line to the output only if you have already read this line before. (The end result is that you remove the first occurrence of each line.) Take special care so that a file with a lot of duplicate lines does not use more memory than what is required for the number of unique lines.

6. Read the entire input one line at a time. Then output all lines sorted by length, with the shortest lines first. In the case where two lines have the same length, resolve their order using the usual "sorted order." Duplicate lines should be printed only once.

7. Do the same as the previous question except that duplicate lines should be printed the same number of times that they appear in the input.

8. Read the entire input one line at a time and then output the even numbered lines (starting with the first line, line 0) followed by the odd-numbered lines.

9. Read the entire input one line at a time and randomly permute the lines before outputting them. To be clear: You should not modify the contents of any line. Instead, the same collection of lines should be printed, but in a random order.

Exercise 1.2. A *Dyck word* is a sequence of +1's and -1's with the property that the sum of any prefix of the sequence is never negative. For example, $+1,-1,+1,-1$ is a Dyck word, but $+1,-1,-1,+1$ is not a Dyck word since the prefix $+1-1-1 < 0$. Describe any relationship between Dyck words and Stack $\mathrm{push}(x)$ and $\mathrm{pop}()$ operations.

Exercise 1.3. A *matched string* is a sequence of $\{$, $\}$, (,), [, and] characters that are properly matched. For example, "$\{\{()[]\}\}$" is a matched string, but this "$\{\{()]\}$" is not, since the second $\{$ is matched with a]. Show how to use a stack so that, given a string of length n, you can determine if it is a matched string in $O(n)$ time.

Exercise 1.4. Suppose you have a Stack, s, that supports only the $\mathrm{push}(x)$ and $\mathrm{pop}()$ operations. Show how, using only a FIFO Queue, q, you can reverse the order of all elements in s.

Exercise 1.5. Using a USet, implement a Bag. A Bag is like a USet—it supports the $\mathrm{add}(x)$, $\mathrm{remove}(x)$ and $\mathrm{find}(x)$ methods—but it allows duplicate elements to be stored. The $\mathrm{find}(x)$ operation in a Bag returns some element (if any) that is equal to x. In addition, a Bag supports the $\mathrm{find\_all}(x)$ operation that returns a list of all elements in the Bag that are equal to x.

Exercise 1.6. From scratch, write and test implementations of the List, USet and SSet interfaces. These do not have to be efficient. They can be used later to test the correctness and performance of more efficient implementations. (The easiest way to do this is to store the elements in an array.)

Exercise 1.7. Work to improve the performance of your implementations from the previous question using any tricks you can think of. Experiment and think about how you could improve the performance of $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$ in your List implementation. Think about how you could improve the performance of the $\mathrm{find}(x)$ operation in your USet and SSet implementations. This exercise is designed to give you a feel for how difficult it can be to obtain efficient implementations of these interfaces.

Chapter 2

Array-Based Lists

In this chapter, we will 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:

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

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 $\mathrm{get}(i)$ and $\mathrm{set}(i,x)$ 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 $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$ 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 associated with 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 $\mathrm{add}(i,x)$ or $\mathrm{remove}(i)$ 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.

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 $a[0],\ldots,a[n-1]$ and, at all times, $\mathrm{length}(a) \geq n$.

```
initialize()
    $a \leftarrow \mathrm{new\_array}(1)$
    $n \leftarrow 0$
```

2.1.1 The Basics

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

```
get($i$)
    **return** $a[i]$

set($i,x$)
    $y \leftarrow a[i]$
    $a[i] \leftarrow x$
    **return** $y$
```

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

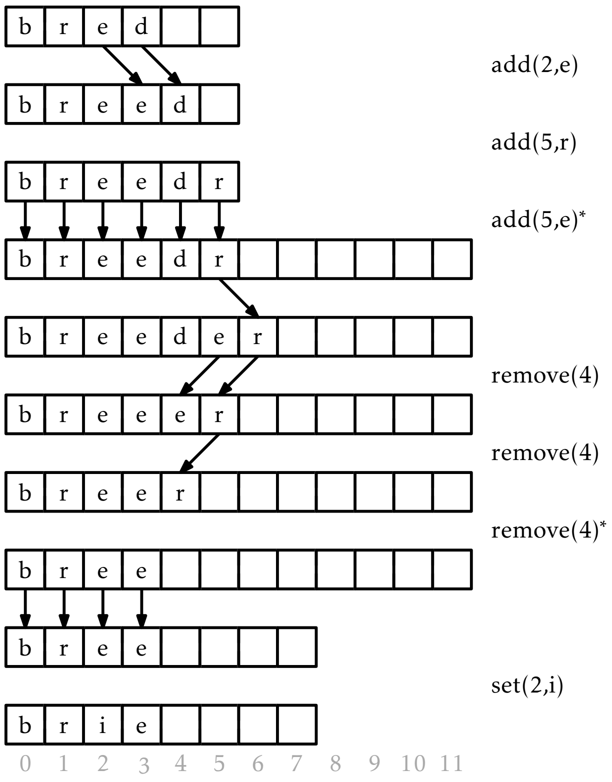


Figure 2.1: A sequence of $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$ operations on an ArrayStack. Arrows denote elements being copied. Operations that result in a call to $\mathrm{resize}()$ are marked with an asterisk.

```
add($i,x$)
    if $n = \mathrm{length}(a)$ then resize()
    $a[i+1,i+2,\ldots,n] \leftarrow a[i,i+1,\ldots,n-1]$
    $a[i] \leftarrow x$
    $n \leftarrow n+1$
```

If we ignore the cost of the potential call to resize(), then 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 $O(n-i)$.

Implementing the remove(i) operation is similar. We shift the elements $a[i+1],\ldots,a[n-1]$ 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 length(a) by checking if $\mathrm{length}(a) \geq 3n$. If so, then we call resize() to reduce the size of a.

```
remove($i$)
    $x \leftarrow a[i]$
    $a[i,i+1,\ldots,n-2] \leftarrow a[i+1,i+2,\ldots,n-1]$
    $n \leftarrow n-1$
    if $\mathrm{length}(a) \geq 3 \cdot n$ then 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 $O(n-i)$.

2.1.2 Growing and Shrinking

The resize() method is fairly straightforward; it allocates a new array b whose size is $2n$ 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(), $\mathrm{length}(a) = 2n$.

```
resize()
    $b \leftarrow \mathrm{new\_array}(\max(1, 2 \cdot n))$
    $b[0,1,\ldots,n-1] \leftarrow a[0,1,\ldots,n-1]$
    $a \leftarrow b$
```

Analyzing the actual cost of the resize() operation is easy. It allocates an array b of size $2n$ and copies the n elements of a into b. This takes $O(n)$ 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 $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$ operation. Instead, it considers the cost of all calls to $\mathrm{resize}()$ during a sequence of m calls to $\mathrm{add}(i,x)$ or $\mathrm{remove}(i)$. In particular, we will show:

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

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

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

which is equivalent to

$$\sum_{i=1}^{r} n_i \leq 2m+2r \ .$$

On the other hand, the total time spent during all calls to $\mathrm{resize}()$ is

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

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

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

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

$$\begin{aligned} R &\geq \mathrm{length}(a)/2-1-\mathrm{length}(a)/3 \\ &= \mathrm{length}(a)/6-1 \\ &= (\mathrm{length}(a)/3)/2-1 \\ &\geq n_i/2-1 \ . \end{aligned}$$

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

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

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* $\mathrm{resize}()$*, an ArrayStack supports the operations*

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

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

The ArrayStack is an efficient way to implement a Stack. In particular, we can implement $\mathrm{push}(x)$ as $\mathrm{add}(n,x)$ and $\mathrm{pop}()$ as $\mathrm{remove}(n-1)$, 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 $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$) and copying (by $\mathrm{resize}()$) of data. In a naive implementation, this would be 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 programming language, there are the $\mathrm{memcpy}(d,s,n)$ and $\mathrm{memmove}(d,s,n)$ functions. In the C++ language there is the $\mathrm{stdcopy}(a_0,a_1,b)$ algorithm. In Java there is the $\mathrm{System.arraycopy}(s,i,d,j,n)$ method.

These functions are usually highly optimized and may even use special machine instructions that can do this copying much faster than we could by using a **for** loop. Although using these functions does not asymptotically decrease the running times, it can still be a worthwhile optimization.

In our C++ and Java implementations, the use of fast array copying functions resulted in speedups of a factor between 2 and 3, depending on the types of operations performed. Your mileage may vary.

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 $\mathrm{remove}()$ operation) from the queue in the same order they are added (using the $\mathrm{add}(x)$ operation).

Notice that an ArrayStack is a poor choice for an implementation of a FIFO queue. It is not a good choice because we must choose one end of the list upon which to add elements and then remove elements from the other end. One of the two operations must work on the head of the list, which involves calling $\mathrm{add}(i,x)$ or $\mathrm{remove}(i)$ with a value of $i = 0$. This gives a running time proportional to 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

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

Initially, both j and n would be set to 0. To add an element, we would place it in $a[j+n]$ 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 10:00 plus five hours gives 3:00. 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 $\bmod$ 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, C++, and Java, the mod operate is represented using the % symbol.

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

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

This treats the array a like a *circular array* in which array indices larger than $\mathrm{length}(a)-1$ "wrap around" to the beginning of the array.

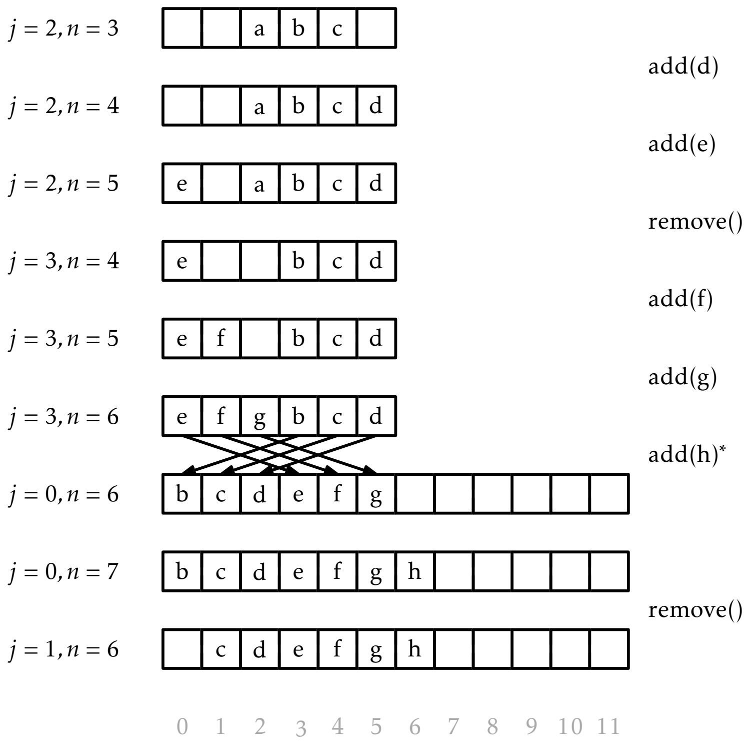


Figure 2.2: A sequence of $\mathrm{add}(x)$ and $\mathrm{remove}(i)$ operations on an ArrayQueue. Arrows denote elements being copied. Operations that result in a call to $\mathrm{resize}()$ are marked with an asterisk.

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.

```
initialize()
    $a \leftarrow \mathrm{new\_array}(1)$
    $j \leftarrow 0$
    $n \leftarrow 0$
```

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

```
add($x$)
    **if** $n + 1 > \mathrm{length}(a)$ **then** resize()
    $a[(j+n) \bmod \mathrm{length}(a)] \leftarrow x$
    $n \leftarrow n + 1$
    **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 $\text{length}(a)$) by setting $j = (j+1) \bmod \text{length}(a)$. Finally, we return the stored value of $a[j]$. If necessary, we may call resize() to decrease the size of a.

```
remove()
    $x \leftarrow a[j]$
    $j \leftarrow (j+1) \bmod \text{length}(a)$
    $n \leftarrow n-1$
    **if** $\text{length}(a) \geq 3 \cdot n$ **then** 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

$$a[j], a[(j+1) \bmod \text{length}(a)], \ldots, a[(j+n-1) \bmod \text{length}(a)]$$

onto

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

and sets $j = 0$.

```
resize()
    $b \leftarrow \text{new\_array}(\max(1, 2 \cdot n))$
    **for** $k$ **in** $0, 1, 2, \ldots, n-1$ **do**
        $b[k] \leftarrow a[(j+k) \bmod \text{length}(a)]$
    $a \leftarrow b$
    $j \leftarrow 0$
```

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 by using the same circular array technique used to represent an ArrayQueue.

```
initialize()
    $a \leftarrow \mathrm{new\_array}(1)$
    $j \leftarrow 0$
    $n \leftarrow 0$
```

The $\mathrm{get}(i)$ and $\mathrm{set}(i,x)$ operations on an ArrayDeque are straightforward. They get or set the array element $a[(j+i) \bmod \mathrm{length}(a)]$.

```
get($i$)
    return $a[(i+j) \bmod \mathrm{length}(a)]$

set($i,x$)
    $y \leftarrow a[(i+j) \bmod \mathrm{length}(a)]$
    $a[(i+j) \bmod \mathrm{length}(a)] \leftarrow x$
    return $y$
```

The implementation of $\mathrm{add}(i,x)$ is a little more interesting. As usual, we first check if a is full and, if necessary, call $\mathrm{resize}()$ 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 $i < n/2$. If so, we shift the elements $a[0],\ldots,a[i-1]$ left by one position. Otherwise ($i \geq n/2$), we shift the elements $a[i],\ldots,a[n-1]$ right by one position. See Figure 2.3 for an illustration of $\mathrm{add}(i,x)$ and $\mathrm{remove}(x)$ operations on an ArrayDeque.

```
add($i,x$)
    if $n = \mathrm{length}(a)$ then resize()
    if $i < n/2$ then
        $j \leftarrow (j-1) \bmod \mathrm{length}(a)$
        for $k$ in $0,1,2,\ldots,i-1$ do
            $a[(j+k) \bmod \mathrm{length}(a)] \leftarrow a[(j+k+1) \bmod \mathrm{length}(a)]$
    else
        for $k$ in $n,n-1,n-2,\ldots,i+1$ do
```

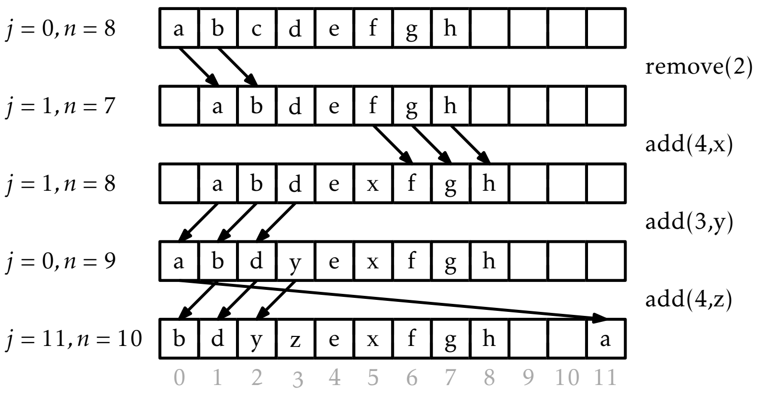


Figure 2.3: A sequence of add(i,x) and remove(i) operations on an ArrayDeque. Arrows denote elements being copied.

```
      $a[(j+k) \bmod \mathrm{length}(a)] \leftarrow a[(j+k-1) \bmod \mathrm{length}(a)]$
  $a[(j+i) \bmod \mathrm{length}(a)] \leftarrow x$
  $n \leftarrow n+1$
```

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

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

```
remove($i$)
  $x \leftarrow a[(j+i) \bmod \mathrm{length}(a)]$
  if $i < n/2$ then
    for $k$ in $i, i-1, i-2, \ldots, 1$ do
      $a[(j+k) \bmod \mathrm{length}(a)] \leftarrow a[(j+k-1) \bmod \mathrm{length}(a)]$
    $j \leftarrow (j+1) \bmod \mathrm{length}(a)$
  else
    for $k$ in $i, i+1, i+2, \ldots, n-2$ do
      $a[(j+k) \bmod \mathrm{length}(a)] \leftarrow a[(j+k+1) \bmod \mathrm{length}(a)]$
  $n \leftarrow n-1$
  if $\mathrm{length}(a) \geq 3 \cdot n$ then 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* $\mathrm{resize}()$*, an ArrayDeque supports the operations*

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

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

2.5 DualArrayDeque: Building a Deque from Two Stacks

Next, we present a data structure, the DualArrayDeque that achieves the same performance bounds as an ArrayDeque by using two ArrayStacks. 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 ArrayStacks. Recall that an ArrayStack is fast when the operations on it modify elements near the end. A DualArrayDeque places two ArrayStacks, called front and back, back-to-back so that operations are fast at either end.

```
initialize()
    $\mathit{front} \leftarrow \mathrm{ArrayStack}()$
    $\mathit{back} \leftarrow \mathrm{ArrayStack}()$
```

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

```
size()
    return $\mathit{front}.\mathrm{size}() + \mathit{back}.\mathrm{size}()$
```

The front ArrayStack stores the list elements that whose indices are $0,\ldots,\mathit{front}.\mathrm{size}()-1$, but stores them in reverse order. The back ArrayStack contains list elements with indices in $\mathit{front}.\mathrm{size}(),\ldots,\mathrm{size}()-1$ in the normal order. In this way, $\mathrm{get}(i)$ and $\mathrm{set}(i,x)$

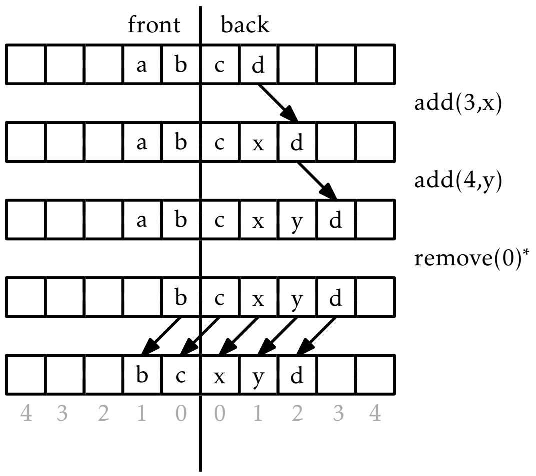


Figure 2.4: A sequence of add(i,x) and remove(i) operations on a DualArrayDeque. Arrows denote elements being copied. Operations that result in a rebalancing by balance() are marked with an asterisk.

translate into appropriate calls to get(i) or set(i,x) on either *front* or *back*, which take $O(1)$ time per operation.

```
get($i$)
  **if** $i < front$.size() **then**
    **return** $front$.get($front$.size() $- i - 1$)
  **else**
    **return** $back$.get($i - front$.size())

set($i,x$)
  **if** $i < front$.size() **then**
    **return** $front$.set($front$.size() $- i - 1, x$)
  **else**
    **return** $back$.set($i - front$.size(), $x$)
```

Note that if an index $i < front$.size(), then it corresponds to the element of *front* at position $front$.size() $- i - 1$, 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:

```
add($i,x$)
  **if** $i < front$.size() **then**
```

```
        $front.\mathrm{add}(front.\mathrm{size}() - i, x)$
    else
        $back.\mathrm{add}(i - front.\mathrm{size}(), x)$
    $\mathrm{balance}()$
```

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

Next we analyze the cost of $\mathrm{add}(i,x)$, ignoring the cost of calls to $\mathrm{balance}()$. If $i < front.\mathrm{size}()$, then $\mathrm{add}(i,x)$ gets implemented by the call to $front.\mathrm{add}(front.\mathrm{size}() - i - 1, x)$. Since $front$ is an ArrayStack, the cost of this is

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

On the other hand, if $i \geq front.\mathrm{size}()$, then $\mathrm{add}(i,x)$ gets implemented as $back.\mathrm{add}(i - front.\mathrm{size}(), x)$. The cost of this is

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

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

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

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

The $\mathrm{remove}(i)$ operation and its analysis resemble the $\mathrm{add}(i,x)$ operation and analysis.

```
$\mathrm{remove}(i)$
    if $i < front.\mathrm{size}()$ then
        $x \leftarrow front.\mathrm{remove}(front.\mathrm{size}() - i - 1)$
    else
        $x \leftarrow back.\mathrm{remove}(i - front.\mathrm{size}())$
    $\mathrm{balance}()$
    return $x$
```

2.5.1 Balancing

Finally, we turn to the $\mathrm{balance}()$ operation performed by $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$. This operation ensures that neither front nor back becomes too big (or too small). It ensures that, unless there are fewer than two 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.

```
balance()
  $n \leftarrow \mathrm{size}()$
  $\mathit{mid} \leftarrow n \operatorname{div} 2$
  if $3 \cdot \mathit{front}.\mathrm{size}() < \mathit{back}.\mathrm{size}()$ or $3 \cdot \mathit{back}.\mathrm{size}() < \mathit{front}.\mathrm{size}()$ then
    $f \leftarrow \mathrm{ArrayStack}()$
    for $i$ in $0,1,2,\ldots,\mathit{mid}-1$ do
      $f.\mathrm{add}(i, \mathrm{get}(\mathit{mid}-i-1))$
    $b \leftarrow \mathrm{ArrayStack}()$
    for $i$ in $0,1,2,\ldots,n-\mathit{mid}-1$ do
      $b.\mathrm{add}(i, \mathrm{get}(\mathit{mid}+i))$
    $\mathit{front} \leftarrow f$
    $\mathit{back} \leftarrow b$
```

Here there is little to analyze. If the $\mathrm{balance}()$ operation does rebalancing, then it moves $O(n)$ elements and this takes $O(n)$ time. This is bad, since $\mathrm{balance}()$ is called with each call to $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$. However, the following lemma shows that, on average, $\mathrm{balance}()$ 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 $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$ are performed, then the total time spent during all calls to $\mathrm{balance}()$ is $O(m)$.*

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

We will perform our analysis using a technique knows as the *potential method*. Define the *potential*, Φ, of the DualArrayDeque as the difference in size between front and back:

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

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

Observe that, immediately after a call to $\mathrm{balance}()$ that shifts elements, the potential, Φ_0, is at most 1, since

$$\Phi_0 = |\lfloor n/2 \rfloor - \lceil n/2 \rceil| \le 1 \ .$$

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

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

Furthermore, the potential at this point in time is

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

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

2.5.2 Summary

The following theorem summarizes the properties of a DualArrayDeque:

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

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

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

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

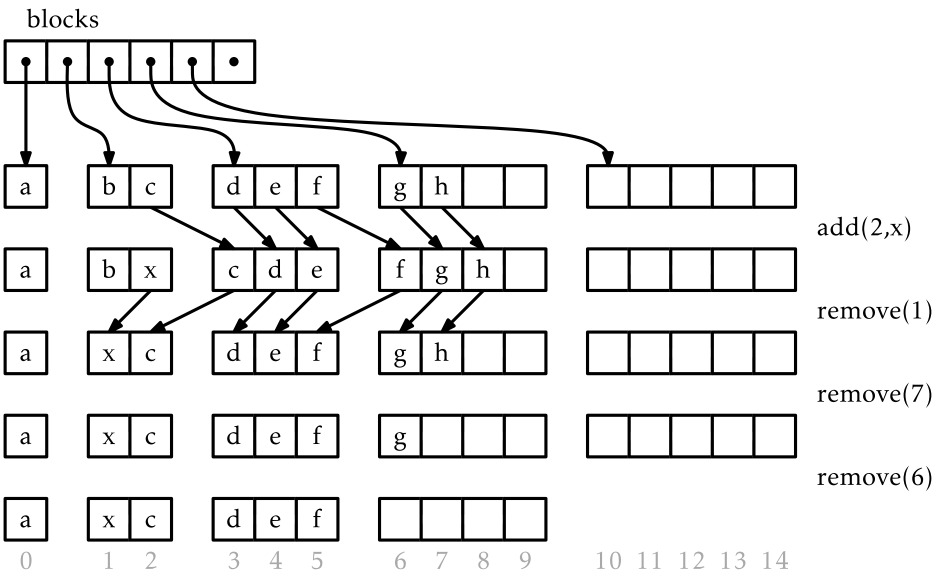


Figure 2.5: A sequence of $\text{add}(i,x)$ and $\text{remove}(i)$ operations on a RootishArrayStack. Arrows denote elements being copied.

arrays frequently are 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 one third of a contains data.

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

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

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

elements. The above formula can be obtained as shown in Figure 2.6.

```
initialize()
    $n \leftarrow 0$
    $blocks \leftarrow \text{ArrayStack}()$
```

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

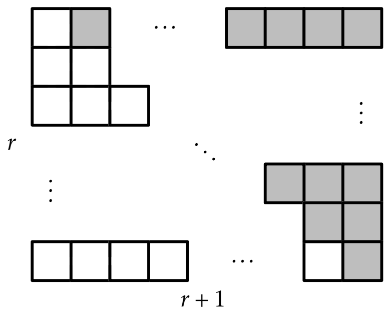


Figure 2.6: The number of white squares is $1+2+3+\cdots+r$. The number of shaded squares is the same. Together the white and shaded squares make a rectangle consisting of $r(r+1)$ squares.

stored in block 1, 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 $0,\dots,b-1$ 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 $0,\dots,b$ is $(b+1)(b+2)/2$. Therefore, b is the smallest integer such that

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

We can rewrite this equation as

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

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

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

```
i2b($i$)
    **return** int_value(ceil(($-3.0 + \sqrt{9 + 8 \cdot i}$))/2.0)
```

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

```
get($i$)
    $b \leftarrow \mathrm{i2b}(i)$
    $j \leftarrow i - b \cdot (b+1)/2$
    **return** $blocks.\mathrm{get}(b)[j]$

set($i,x$)
    $b \leftarrow \mathrm{i2b}(i)$
    $j \leftarrow i - b \cdot (b+1)/2$
    $y \leftarrow blocks.\mathrm{get}(b)[j]$
    $blocks.\mathrm{get}(b)[j] \leftarrow x$
    **return** $y$
```

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

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

```
add($i,x$)
    $r \leftarrow blocks.\mathrm{size}()$
    **if** $r \cdot (r+1)/2 < n+1$ **then** grow()
    $n \leftarrow n+1$
    **for** $j$ **in** $n-1, n-2, n-3, \ldots, i+1$ **do**
        $\mathrm{set}(j, \mathrm{get}(j-1))$
    $\mathrm{set}(i,x)$
```

The $\mathrm{grow}()$ method does what we expect. It adds a new block:

```
grow()
    $blocks.\mathrm{append}(\mathrm{new\_array}(blocks.\mathrm{size}()+1))$
```

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

The remove(i) operation is similar to add(i,x). It shifts the elements with indices $i+1,\ldots,n$ 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:

```
remove($i$)
    $x \leftarrow$ get($i$)
    **for** $j$ **in** $i, i+1, i+2, \ldots, n-2$ **do**
        set($j$, get($j+1$))
    $n \leftarrow n-1$
    $r \leftarrow$ $blocks$.size()
    **if** $(r-2)\cdot(r-1)/2 \geq n$ **then** shrink()
    **return** $x$
```

```
shrink()
    $r \leftarrow$ $blocks$.size()
    **while** $r > 0$ **and** $(r-2)\cdot(r-1)/2 \geq n$ **do**
        $blocks$.remove($blocks$.size() $-1$)
        $r \leftarrow r-1$
```

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 $O(n-i)$.

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 copy any 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 time proportional to r.

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 $r-1$ elements have been added or removed. Therefore, even if grow() and shrink() take $O(r)$ time, this cost can be amortized over at least $r-1$ 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 $\mathrm{remove}(i)$ operation ensures that a RootishArrayStack never has more than two blocks that are not completely full. The number of blocks, r, used by a RootishArrayStack that stores n elements therefore satisfies

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

Again, using the quadratic equation on this gives

$$r \le \frac{1}{2}\left(3+\sqrt{8n+1}\right) = O(\sqrt{n}) \ .$$

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 ArrayStack, 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 at least in $\sqrt{n}$ space (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 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 these pointers are 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 our discussion of the RootishArrayStack data structure:

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

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

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

The space (measured in words)[2] *used by a RootishArrayStack that stores* n *elements is* $n + O(\sqrt{n})$.

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 [25, Section 2.2.2].

Brodnik *et al.* [4] 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 $\mathrm{i2b}(i)$ method. Within their scheme, the block containing i is block $\lfloor \log(i+1) \rfloor$, which is simply the index of the leading 1 bit in the binary representation of $i+1$. 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 two-level *tiered-vector* of Goodrich and Kloss [20]. This structure supports the $\mathrm{get}(i,x)$ and $\mathrm{set}(i,x)$ operations in constant time and $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$ in $O(\sqrt{n})$ time. These running times are similar to what can be achieved with the more careful implementation of a RootishArrayStack discussed in Exercise 2.10.

Exercise 2.1. The List method $\mathrm{add\_all}(i,c)$ inserts all elements of the Collection c into the list at position i. (The $\mathrm{add}(i,x)$ method is a special case where $c = \{x\}$.) Explain why, for the data structures in this chapter, it is not efficient to implement $\mathrm{add\_all}(i,c)$ by repeated calls to $\mathrm{add}(i,x)$. 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 $\mathrm{remove}()$ operation removes an element that is chosen uniformly at random among all the elements currently in the queue. (Think of a RandomQueue as a bag in which we can add elements or reach in and blindly remove some random element.) The $\mathrm{add}(x)$ and $\mathrm{remove}()$ operations in a RandomQueue should run in amortized constant time per operation.

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

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

In other words, modifications are fast if they are near either end or near the middle of the list.

[2]Recall Section 1.4 for a discussion of how memory is measured.

Exercise 2.4. Implement a method $\mathrm{rotate}(a,r)$ that "rotates" the array a so that $a[i]$ moves to $a[(i+r) \bmod \mathrm{length}(a)]$, for all $i \in \{0,\ldots,\mathrm{length}(a)\}$.

Exercise 2.5. Implement a method $\mathrm{rotate}(r)$ that "rotates" a List so that list item i becomes list item $(i+r) \bmod n$. When run on an ArrayDeque, or a DualArrayDeque, $\mathrm{rotate}(r)$ should run in $O(1+\min\{r,n-r\})$ time.

Exercise 2.6. This exercise is left out of the pseudocode edition.

Exercise 2.7. Modify the ArrayDeque implementation so that it does not use the $\bmod$ operator (which is expensive on some systems). Instead, it should make use of the fact that, if $\mathrm{length}(a)$ is a power of 2, then

$$k \bmod \mathrm{length}(a) = k \wedge (\mathrm{length}(a)-1) \ .$$

(Here, $\wedge$ is the bitwise-and operator.)

Exercise 2.8. 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 $\mathrm{rebuild}()$ operation is performed. The amortized cost of all operations should be the same as in an ArrayDeque.
Hint: Getting this to work is really all about how you implement the $\mathrm{rebuild}()$ operation. You would like $\mathrm{rebuild}()$ 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 $\mathrm{System.arraycopy}(a,i,b,i,n)$) and see if you can get it to outperform the ArrayDeque implementation.

Exercise 2.9. Design and implement a version of a RootishArrayStack that has only $O(\sqrt{n})$ wasted space, but that can perform $\mathrm{add}(i,x)$ and $\mathrm{remove}(i,x)$ operations in $O(1+\min\{i,n-i\})$ time.

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

Exercise 2.11. Design and implement a version of a RootishArrayStack that has only $O(\sqrt{n})$ wasted space, but that can perform $\mathrm{add}(i,x)$ and $\mathrm{remove}(i,x)$ operations in $O(1+\min\{i,\sqrt{n},n-i\})$ time. (See Section 3.3 for ideas on how to achieve this.)

Exercise 2.12. Design and implement a CubishArrayStack. This three level structure implements the List interface using $O(n^{2/3})$ wasted space. In this structure, $\mathrm{get}(i)$ and $\mathrm{set}(i,x)$ take constant time; while $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$ take $O(n^{1/3})$ amortized time.

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. Using references (pointers), the nodes are linked together into a sequence. We first study singly-linked lists, which can implement Stack and (FIFO) Queue operations in constant time per operation and then move on to doubly-linked lists, which can implement Deque operations in constant time.

Linked lists have advantages and disadvantages when compared to array-based implementations of the List interface. The primary disadvantage is that we lose the ability to access any element using $\mathrm{get}(i)$ or $\mathrm{set}(i,x)$ in constant time. Instead, we have to walk through the list, one element at a time, until we reach the ith 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 Nodes. 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, $w.next = nil$

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:

```
initialize()
    $n \leftarrow 0$
    $head \leftarrow nil$
    $tail \leftarrow nil$
```

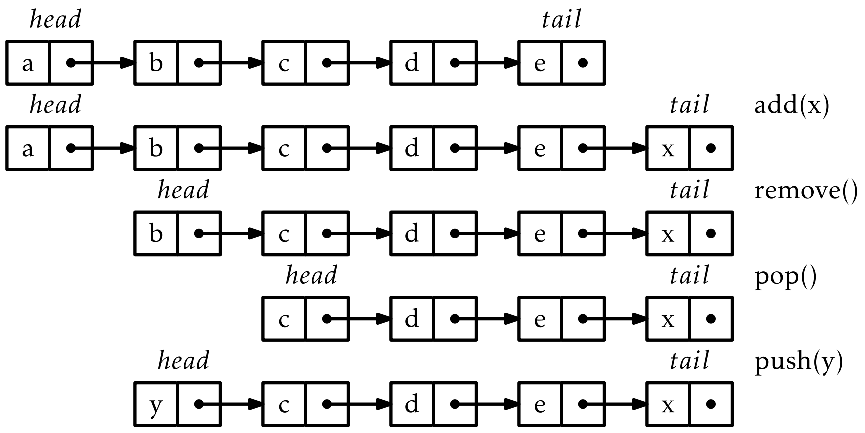


Figure 3.1: A sequence of Queue ($\mathrm{add}(x)$ and $\mathrm{remove}()$) and Stack ($\mathrm{push}(x)$ and $\mathrm{pop}()$) operations on an SLList.

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

An SLList can efficiently implement the Stack operations $\mathrm{push}()$ and $\mathrm{pop}()$ by adding and removing elements at the head of the sequence. The $\mathrm{push}()$ 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:

```
push($x$)
    $u \leftarrow \mathrm{new\_node}(x)$
    $u.next \leftarrow head$
    $head \leftarrow u$
    **if** $n = 0$ **then**
        $tail \leftarrow u$
    $n \leftarrow n + 1$
    **return** $x$
```

The $\mathrm{pop}()$ operation, after checking that the SLList is not empty, removes the head by setting $head \leftarrow head.next$ and decrementing n. A special case occurs when the last element is being removed, in which case $tail$ is set to nil:

```
pop()
    **if** $n = 0$ **then return** $nil$
    $x \leftarrow head.x$
    $head \leftarrow head.next$
    $n \leftarrow n-1$
    if $n = 0$ then
        $tail \leftarrow nil$
    return $x$
```

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

3.1.1 Queue Operations

An SLList can also implement the FIFO queue operations $\mathrm{add}(x)$ and $\mathrm{remove}()$ in constant time. Removals are done from the head of the list, and are identical to the $\mathrm{pop}()$ operation:

```
remove()
    return pop()
```

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 $tail = head = nil$. In this case, both $tail$ and $head$ are set to u.

```
add($x$)
    $u \leftarrow \mathrm{new\_node}(x)$
    if $n = 0$ then
        $head \leftarrow u$
    else
        $tail.next \leftarrow u$
    $tail \leftarrow u$
    $n \leftarrow n+1$
    return $true$
```

Clearly, both $\mathrm{add}(x)$ and $\mathrm{remove}()$ 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* $\mathrm{push}(x)$, $\mathrm{pop}()$, $\mathrm{add}(x)$ *and* $\mathrm{remove}()$ *operations run in* $O(1)$ *time per operation.*

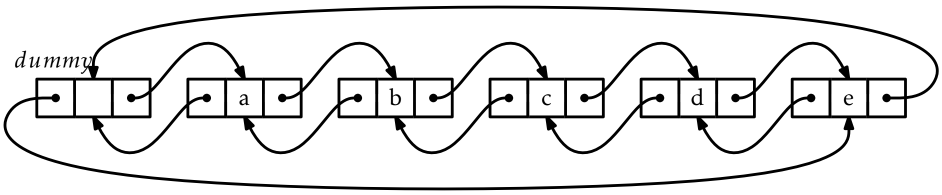


Figure 3.2: A DLList containing a,b,c,d,e.

An SLList nearly implements the full set of Deque operations. The only missing operation is removing 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.

When implementing an SLList, we saw that there were always several special cases to worry about. For example, removing the last element from an SLList or adding an element to an empty SLList requires care to ensure 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.

```
initialize()
    $n \leftarrow 0$
    $dummy \leftarrow$ DLList.Node($nil$)
    $dummy.prev \leftarrow dummy$
    $dummy.next \leftarrow dummy$
```

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 $O(1 +$

$\min\{i, n-i\})$ time:

```
get_node($i$)
    **if** $i < n/2$ **then**
        $p \leftarrow dummy.next$
        **repeat** $i$ **times**
            $p \leftarrow p.next$
    **else**
        $p \leftarrow dummy$
        **repeat** $n - i$ **times**
            $p \leftarrow p.prev$
    **return** $p$
```

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

```
get($i$)
    **return** get_node($i$).$x$

set($i,x$)
    $u \leftarrow$ get_node($i$)
    $y \leftarrow u.x$
    $u.x \leftarrow x$
    **return** $y$
```

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

3.2.1 Adding and Removing

If we have a reference to a node w in a DLList and we want to insert a node u before w, then this is just a matter of setting $u.next = w$, $u.prev = w.prev$, and then adjusting $u.prev.next$ and $u.next.prev$. (See Figure 3.3.) Thanks to the dummy node, there is no need to worry about $w.prev$ or $w.next$ not existing.

```
add_before($w,x$)
    $u \leftarrow$ DLList.Node($x$)
    $u.prev \leftarrow w.prev$
    $u.next \leftarrow w$
```

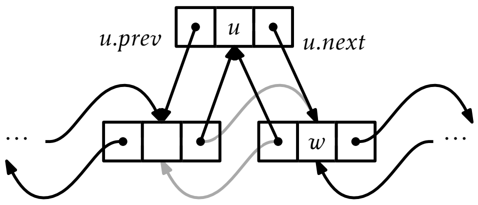


Figure 3.3: Adding the node u before the node w in a DLList.

```
    $u.next.prev \leftarrow u$
    $u.prev.next \leftarrow u$
    $n \leftarrow n+1$
    **return** $u$
```

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

```
add($i,x$)
    add_before(get_node($i$), $x$)
```

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

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

```
remove($w$)
    $w.prev.next \leftarrow w.next$
    $w.next.prev \leftarrow w.prev$
    $n \leftarrow n-1$
```

Now the $\mathrm{remove}(i)$ operation is trivial. We find the node with index i and remove it:

```
remove($i$)
    remove(get_node($i$))
```

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

3.2.2 Summary

The following theorem summarizes the performance of a DLList:

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

It is worth noting that, if we ignore the cost of the $\mathrm{get\_node}(i)$ 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.

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 of the fields in a Node are dedicated to maintaining the list, and only one of the fields is 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.

For reasons that will become clear later, it will be helpful if we can do Deque operations on each block. The data structure that 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 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.

An SEList is just a doubly-linked list of blocks. In addition to $next$ and $prev$ pointers, each node u in an SEList contains a BDeque, $u.d$.

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 every block 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 a value of b within a constant factor of $\sqrt{n}$, we can make the space-overhead of an SEList approach the $\sqrt{n}$ lower bound given in Section 2.6.2.

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:

1. The node u that contains the block that contains the element with index i; and
2. the index j of the element within its block.

To find the block that contains a particular element, we proceed the same way as we do 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.

```
get_location($i$)
    if $i < n \operatorname{div} 2$ then
        $u \leftarrow dummy.next$
        while $i \geq u.d.\mathrm{size}()$ do
            $i \leftarrow i - u.d.\mathrm{size}()$
            $u \leftarrow u.next$
        return $u, i$
    else
        $u \leftarrow dummy$
        $idx \leftarrow n$
        while $i < idx$ do
            $u \leftarrow u.prev$
            $idx \leftarrow idx - u.d.\mathrm{size}()$
    return $u, i - idx$
```

Remember that, with the exception of at most one block, each block contains at least $b-1$ elements, so each step in our search gets us $b-1$ elements closer to the element we are looking for. If we are searching forward, this means that we reach the node we want after $O(1+i/b)$ steps. If we search backwards, then we reach the node we want after $O(1+(n-i)/b)$ 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 $O(1+\min\{i, n-i\}/b)$.

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

```
get($i$)
    $u, j \leftarrow$ get_location($i$)
    return $u.d$.get($j$)

set($i, x$)
    $u, j \leftarrow$ get_location($i$)
    return $u.d$.set($j, x$)
```

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

3.3.3 Adding an Element

Adding elements to an SEList is a little more complicated. Before considering the general case, we consider the easier operation, $\mathrm{add}(x)$, 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.

```
append($x$)
    $last \leftarrow dummy.prev$
    if $last = dummy$ or $last.d$.size() $= b+1$ then
        $last \leftarrow$ add_before($dummy$)
    $last.d$.append($x$)
    $n \leftarrow n+1$
```

Things get more complicated when we add to the interior of the list using $\mathrm{add}(i,x)$. We first locate i to get the node u whose block contains the ith list item. The problem is

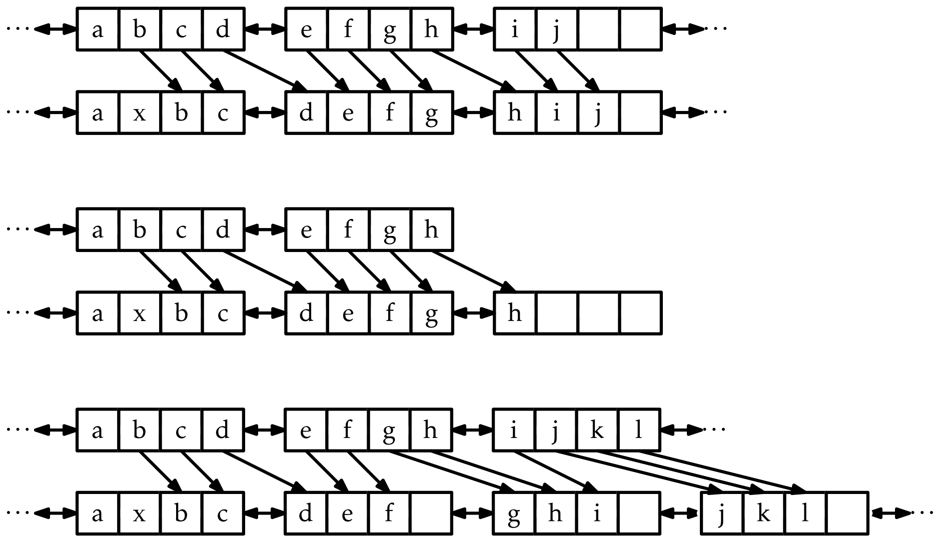


Figure 3.4: The three cases that occur during the addition of an item x in the interior of an SEList. (This SEList has block size $b = 3$.)

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 $b+1$ elements, so that it is full and there is no room for x.

Let $u_0, u_1, u_2, \ldots$ denote u, $u.next$, $u.next.next$, and so on. We explore $u_0, u_1, u_2, \ldots$ looking for a node that can provide space for x. Three cases can occur during our space exploration (see Figure 3.4):

1. We quickly (in $r+1 \le b$ steps) find a node 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 u_r becomes a free space in u_0. We can then insert x into u_0's block.

2. We quickly (in $r+1 \le b$ 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.

```
add($i$,$x$)
    **if** $i = n$ **then**
```

```
        append($x$)
        return
    $u, j \leftarrow$ get_location($i$)
    $r \leftarrow 0$
    $w \leftarrow u$
    while $r < b$ and $w \neq dummy$ and $w.d$.size() $= b+1$ do
        $w \leftarrow w.next$
        $r \leftarrow r+1$
    if $r = b$ then # b blocks, each with b+1 elements
        spread($u$)
        $w \leftarrow u$
    if $w = dummy$ then # ran off the end - add new node
        $w \leftarrow$ add_before($w$)
    while $w \neq u$ do # work backwards, shifting elements as we go
        $w.d$.add_first($w.prev.d$.remove_last())
        $w \leftarrow w.prev$
    $w.d$.add($j, x$)
    $n \leftarrow n+1$
```

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 $O(b)$ time. Case 3 involves calling the spread(u) method, which moves $b(b+1)$ elements and takes $O(b^2)$ 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 $O(b + \min\{i, n-i\}/b)$.

3.3.4 Removing an Element

Removing an element 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 become smaller than $b-1$.

Again, let $u_0, u_1, u_2, \ldots$ denote u, $u.next$, $u.next.next$, and so on. We examine $u_0, u_1, u_2, \ldots$ in order to look for a node from which we can borrow an element to make the size of u_0's block at least $b-1$. There are three cases to consider (see Figure 3.5):

1. We quickly (in $r+1 \leq b$ steps) find a node whose block contains more than $b-1$ elements. In this case, we perform r shifts of an element from one block into the previous one, so that the extra element in u_r becomes an extra element in u_0. We can then remove the appropriate element from u_0's block.

2. We quickly (in $r+1 \leq b$ steps) run off the end of the list of blocks. In this case, u_r is

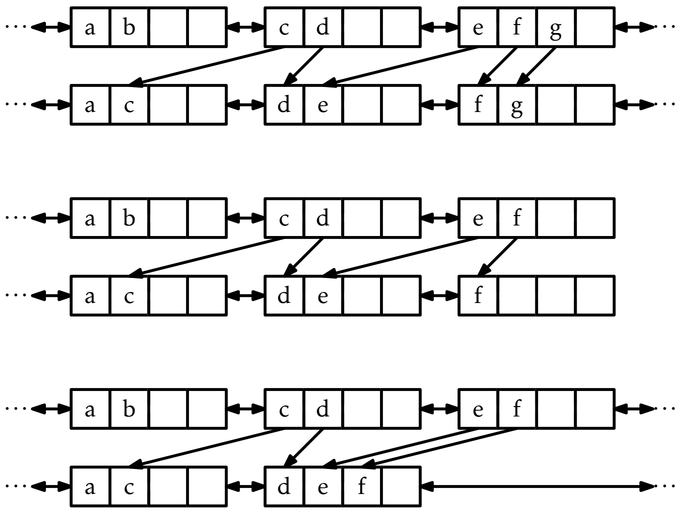


Figure 3.5: The three cases that occur during the removal of an item x in the interior of an SEList. (This SEList has block size $b = 3$.)

the last block, and there is no need for u_r's block to contain at least $b-1$ elements. Therefore, we proceed as above, borrowing an element from u_r to make an extra element in u_0. If this causes u_r's block to become empty, then we remove it.

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 and we can then remove the appropriate element from it.

```
remove($i$)
  $u, j \leftarrow$ get_location($i$)
  $y \leftarrow u.d.$get($j$)
  $w \leftarrow u$
  $r \leftarrow 0$
  while $r < b$ and $w \neq dummy$ and $w.d.$size() $= b-1$ do
    $w \leftarrow w.next$
    $r \leftarrow r+1$
  if $r = b$ then # b blocks, each with b-1 elements
    gather($u$)
  $u.d.$remove($j$)
```

```
    while $u.d.\mathrm{size}() < b-1$ and $u.next \neq dummy$ do
        $u.d.\mathrm{add\_last}(u.next.d.\mathrm{remove\_first}())$
        $u \leftarrow u.next$
    if $u.d.\mathrm{size}() = 0$ then $\mathrm{remove\_node}(u)$
    $n \leftarrow n-1$
```

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

3.3.5 Amortized Analysis of Spreading and Gathering

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

```
spread($u$)
    $w \leftarrow u$
    for $j$ in $0,1,2,\ldots,b-1$ do
        $w \leftarrow w.next$
    $w \leftarrow \mathrm{add\_before}(w)$
    while $w \neq u$ do
        while $w.d.\mathrm{size}() < b$ do
            $w.d.\mathrm{add\_first}(w.prev.d.\mathrm{remove\_last}())$
        $w \leftarrow w.prev$
```

```
gather($u$)
    $w \leftarrow u$
    for $j$ in $0,1,2,\ldots,b-2$ do
        while $w.d.\mathrm{size}() < b$ do
            $w.d.\mathrm{add\_last}(w.next.d.\mathrm{remove\_first}())$
        $w \leftarrow w.next$
    $\mathrm{remove\_node}(w)$
```

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

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

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 $b-1$ or $b+1$ 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 $\mathrm{add}(i,x)$ operation and its relation to the number of calls to $\mathrm{spread}(u)$. The analysis of $\mathrm{remove}(i)$ and $\mathrm{gather}(u)$ is identical.

Notice that, if Case 1 occurs during the $\mathrm{add}(i,x)$ method, then only one node, u_r has the size of its block changed. Therefore, at most one node, namely 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 size, 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 one.

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

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 $b-1$. 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 $b-1$ occurrences of Case 1 or Case 2. Thus, for every call to $\mathrm{spread}(u)$ there are at least b calls to $\mathrm{add}(i,x)$. 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* $\mathrm{spread}(u)$ *and* $\mathrm{gather}(u)$*, an SEList with block size b supports the operations*

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

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

The space (measured in words)[1] *used by an SEList that stores n elements is $n+O(b+n/b)$.*

[1] Recall Section 1.4 for a discussion of how memory is measured.

The SEList is a trade-off between an ArrayList and a DLList where the relative mix of these two structures depends on the block size b. At the extreme $b = 2$, each SEList node stores at most three values, which is not much different than a DLList. At the other extreme, $b > n$, all the elements are stored in a single array, just like in an ArrayList. In between these two extremes lies a trade-off 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 established techniques, having been used in programs for over 40 years. They are discussed, for example, by Knuth [25, Sections 2.2.3–2.2.5]. Even the SEList data structure seems to be a well-known data structures exercise. The SEList is sometimes referred to as an *unrolled linked list* [36].

Another way to save space in a doubly-linked list is to use so-called XOR-lists. In an XOR-list, each node, u, contains only one pointer, called $u.nextprev$, that holds the bitwise exclusive-or of $u.prev$ and $u.next$. The list itself needs to store two pointers, one to the $dummy$ node and one to $dummy.next$ (the first node, or $dummy$ if the list is empty). This technique uses the fact that, if we have pointers to u and $u.prev$, then we can extract $u.next$ using the formula

$$u.next = u.prev \hat{} u.nextprev \quad .$$

(Here ^ computes the bitwise exclusive-or of its two arguments.) This technique complicates the code a little and is not possible in some languages, like Java and Python, that have garbage collection but gives a doubly-linked list implementation that requires only one pointer per node. See Sinha's magazine article [37] for a detailed discussion of XOR-lists.

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

Exercise 3.2. Design and implement an SLList method, $\mathrm{second\_last}()$, that returns the second-last element of an SLList. Do this without using the member variable, n, that keeps track of the size of the list.

Exercise 3.3. Implement the List operations $\mathrm{get}(i)$, $\mathrm{set}(i,x)$, $\mathrm{add}(i,x)$ and $\mathrm{remove}(i)$ on an SLList. Each of these operations should run in $O(1+i)$ time.

Exercise 3.4. Design and implement an SLList method, $\mathrm{reverse}()$ that reverses the order of elements in an SLList. This method should run in $O(n)$ time, should not use recursion, should not use any secondary data structures, and should not create any new nodes.

Exercise 3.5. Design and implement SLList and DLList methods called $\mathrm{check\_size}()$. These methods walk through the list and count the number of nodes to see if this matches the value, n, stored in the list. These methods return nothing, but throw an exception if the size they compute does not match the value of n.

Exercise 3.6. Try to recreate the code for the $\mathrm{add\_before}(w)$ operation that creates a node, u, and adds it in a DLList just before the node w. Do not refer to this chapter. Even if your code does not exactly match the code given in this book it may still be correct. Test it and see if it works.

The next few exercises involve performing manipulations on DLLists. You should complete them without allocating any new nodes or temporary arrays. They can all be done only by changing the $prev$ and $next$ values of existing nodes.

Exercise 3.7. Write a DLList method $\mathrm{is\_palindrome}()$ that returns $true$ if the list is a *palindrome*, i.e., the element at position i is equal to the element at position $n-i-1$ for all $i \in \{0,\ldots,n-1\}$. Your code should run in $O(n)$ time.

Exercise 3.8. Implement a method $\mathrm{rotate}(r)$ that "rotates" a DLList so that list item i becomes list item $(i+r) \bmod n$. This method should run in $O(1+\min\{r,n-r\})$ time and should not modify any nodes in the list.

Exercise 3.9. Write a method, $\mathrm{truncate}(i)$, that truncates a DLList at position i. After executing this method, the size of the list will be i and it should contain only the elements at indices $0,\ldots,i-1$. The return value is another DLList that contains the elements at indices $i,\ldots,n-1$. This method should run in $O(\min\{i,n-i\})$ time.

Exercise 3.10. Write a DLList method, $\mathrm{absorb}(l_2)$, that takes as an argument a DLList, l_2, empties it and appends its contents, in order, to the receiver. For example, if l_1 contains a,b,c and l_2 contains d,e,f, then after calling $l_1.\mathrm{absorb}(l_2)$, l_1 will contain a,b,c,d,e,f and l_2 will be empty.

Exercise 3.11. Write a method $\mathrm{deal}()$ that removes all the elements with odd-numbered indices from a DLList and return a DLList containing these elements. For example, if l_1, contains the elements a,b,c,d,e,f, then after calling $l_1.\mathrm{deal}()$, l_1 should contain a,c,e and a list containing b,d,f should be returned.

Exercise 3.12. Write a method, $\mathrm{reverse}()$, that reverses the order of elements in a DLList.

Exercise 3.13. This exercise walks you through an implementation of the merge-sort algorithm for sorting a DLList, as discussed in Section 6.1.1.

1. Write a DLList method called $\mathrm{take\_first}(l_2)$. This method takes the first node from l_2 and appends it to the the receiving list. This is equivalent to $\mathrm{add}(\mathrm{size}(), l_2.\mathrm{remove}(0))$, except that it should not create a new node.

2. Write a DLList static method, $\mathrm{merge}(l_1,l_2)$, that takes two sorted lists l_1 and l_2, merges them, and returns a new sorted list containing the result. This causes l_1 and l_2 to be emptied in the proces. For example, if l_1 contains a,c,d and l_2 contains b,e,f, then this method returns a new list containing a,b,c,d,e,f.

3. Write a DLList method $\mathrm{sort}()$ that sorts the elements contained in the list using the merge sort algorithm. This recursive algorithm works in the following way:

   (a) If the list contains 0 or 1 elements then there is nothing to do. Otherwise,

   (b) Using the $\mathrm{truncate}(\mathrm{size}()/2)$ method, split the list into two lists of approximately equal length, l_1 and l_2;

   (c) Recursively sort l_1;

   (d) Recursively sort l_2; and, finally,

   (e) Merge l_1 and l_2 into a single sorted list.

The next few exercises are more advanced and require a clear understanding of what happens to the minimum value stored in a Stack or Queue as items are added and removed.

Exercise 3.14. Design and implement a MinStack data structure that can store comparable elements and supports the stack operations $\mathrm{push}(x)$, $\mathrm{pop}()$, and $\mathrm{size}()$, as well as the $\min()$ operation, which returns the minimum value currently stored in the data structure. All operations should run in constant time.

Exercise 3.15. Design and implement a MinQueue data structure that can store comparable elements and supports the queue operations $\mathrm{add}(x)$, $\mathrm{remove}()$, and $\mathrm{size}()$, as well as the $\min()$ operation, which returns the minimum value currently stored in the data structure. All operations should run in constant amortized time.

Exercise 3.16. Design and implement a MinDeque data structure that can store comparable elements and supports all the deque operations $\mathrm{add\_first}(x)$, $\mathrm{add\_last}(x)$ $\mathrm{remove\_first}()$, $\mathrm{remove\_last}()$ and $\mathrm{size}()$, and the $\min()$ operation, which returns the minimum value currently stored in the data structure. All operations should run in constant amortized time.

The next exercises are designed to test the reader's understanding of the implementation and analysis of the space-efficient SEList:

Exercise 3.17. Prove that, if an SEList is used like a Stack (so that the only modifications to the SEList are done using $\mathrm{push}(x) \equiv \mathrm{add}(\mathrm{size}(),x)$ and $\mathrm{pop}() \equiv \mathrm{remove}(\mathrm{size}()-1)$), then these operations run in constant amortized time, independent of the value of b.

Exercise 3.18. Design and implement of a version of an SEList that supports all the Deque operations in constant amortized time per operation, independent of the value of b.

Exercise 3.19. Explain how to use the bitwise exclusive-or operator, ^, to swap the values of two int variables without using a third variable.

Chapter 4

Binary Trees

This chapter introduces one of the most fundamental structures in computer science: binary trees. The use of the word *tree* here comes from the fact that, when we draw them, the resultant drawing often resembles the trees found in a forest. There are many ways 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 4.1). For example, Figure 4.2.a shows a binary tree with nine nodes.

Because binary trees are so important, a certain terminology has developed for 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. 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

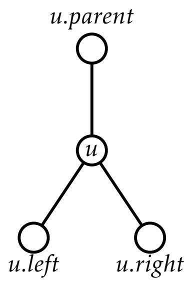


Figure 4.1: The parent, left child, and right child of the node u in a BinaryTree.

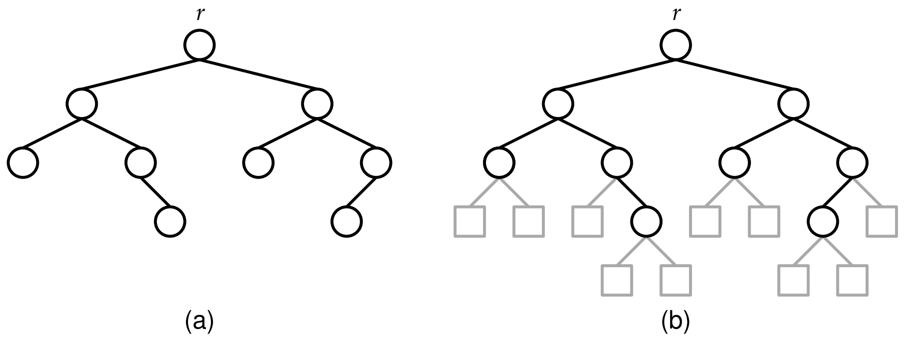


Figure 4.2: A binary tree with (a) nine real nodes and (b) ten external nodes.

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, correspondingly, any node that does not have a right child has an external node as its right child (see Figure 4.2.b). It is easy to verify, by induction, that a binary tree with $n \geq 1$ real nodes has $n+1$ external nodes.

4.1 BinaryTree: A Basic Binary Tree

The simplest way to represent a node, u, in a binary tree is to explicitly store the (at most three) neighbours of u. When one of these three neighbours is not present, we set it to nil. In this way, both external nodes of the tree and the parent of the root correspond to the value nil.

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

```
initialize()
    $r \leftarrow nil$
```

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:

```
depth($u$)
    $d \leftarrow 0$
    while ($u \neq r$) do
        $u \leftarrow u.parent$
    $d \leftarrow d+1$
  **return** $d$
```

4.1.1 Recursive Algorithms

Using recursive algorithms makes it very easy to compute facts about binary trees. 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 up these sizes, and add one:

```
size($u$)
  **if** $u = nil$ **then return** $0$
  **return** $1 + \mathrm{size}(u.left) + \mathrm{size}(u.right)$
```

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

```
height($u$)
  **if** $u = nil$ **then return** $-1$
  **return** $1 + \max(\mathrm{height}(u.left), \mathrm{height}(u.right))$
```

4.1.2 Traversing Binary Trees

The two algorithms from the previous section both 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:

```
traverse($u$)
  **if** $u = nil$ **then return**
  traverse($u.left$)
  traverse($u.right$)
```

Using recursion this way produces very short, simple code, but it can also 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 recursion could very well use more stack space than is available, causing a crash.

To traverse a binary tree without recursion, you can use an algorithm that relies on where it came from to determine where it will go next. See Figure 4.3. If we arrive

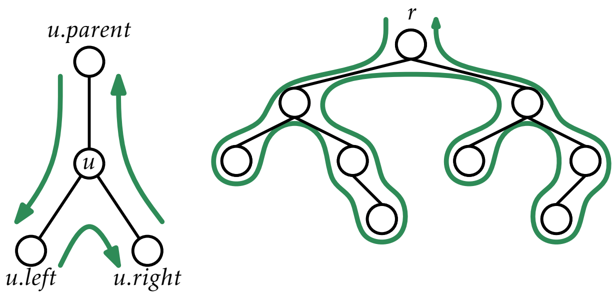


Figure 4.3: The three cases that occur at node u when traversing a binary tree non-recursively, and the resultant traversal of the tree.

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, and 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:

```
traverse2()
    $u \leftarrow r$
    $prv \leftarrow nil$
    while $u \neq nil$ do
        if $prv = u.parent$ then
            if $u.left \neq nil$ then $nxt \leftarrow u.left$
            else if $u.right \neq nil$ $nxt \leftarrow u.right$
            else $nxt \leftarrow u.parent$
        else if $prv = u.left$
            if $u.right \neq nil$ then $nxt \leftarrow u.right$
            else $nxt \leftarrow u.parent$
        else
            $nxt \leftarrow u.parent$
        $prv \leftarrow u$
        $u \leftarrow nxt$
```

The same facts that can be computed with recursive algorithms can also be computed in this way, without recursion. 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:

```
size2()
    $u \leftarrow r$
    $prv \leftarrow nil$
    $n \leftarrow 0$
    while $u \neq nil$ do
        if $prv = u.parent$ then
            $n \leftarrow n + 1$
            if $u.left \neq nil$ then $nxt \leftarrow u.left$
            else if $u.right \neq nil$ $nxt \leftarrow u.right$
            else $nxt \leftarrow u.parent$
        else if $prv = u.left$
            if $u.right \neq nil$ then $nxt \leftarrow u.right$
            else $nxt \leftarrow u.parent$
        else
            $nxt \leftarrow u.parent$
        $prv \leftarrow u$
        $u \leftarrow nxt$
    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 moving down, visiting the nodes at each level from left to right (see Figure 4.4). This is similar to the way that we would read a page of English text. Breadth-first traversal 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:

```
bf_traverse()
    $q \leftarrow$ ArrayQueue()
    if $r \neq nil$ then $q$.add($r$)
    while $q$.size() $> 0$ do
        $u \leftarrow q$.remove()
        if $u.left \neq nil$ then $q$.add($u.left$)
        if $u.right \neq nil$ then $q$.add($u.right$)
```

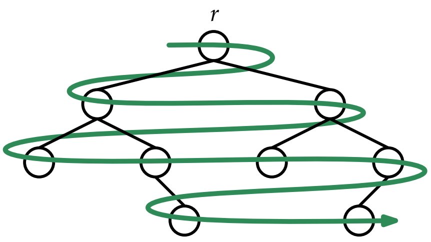


Figure 4.4: During a breadth-first traversal, the nodes of a binary tree are visited level-by-level, and left-to-right within each level.

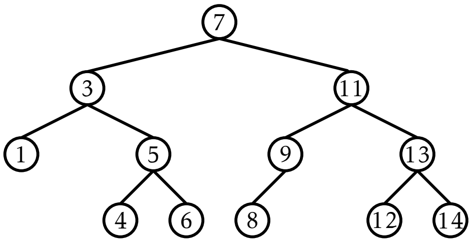


Figure 4.5: A binary search tree.

4.2 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 $u.\mathit{left}$ is less than $u.x$ and every data value stored in the subtree rooted at $u.\mathit{right}$ is greater than $u.x$. An example of a BinarySearchTree is shown in Figure 4.5.

4.2.1 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:

1. If $x < u.x$, then the search proceeds to $u.\mathit{left}$;
2. If $x > u.x$, then the search proceeds to $u.\mathit{right}$;

3. If $x = u.x$, then we have found the node u containing x.

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

```
find_eq($x$)
    $w \leftarrow r$
    while $w \neq nil$ do
        if $x < w.x$ then
            $w \leftarrow w.left$
        else if $x > w.x$
            $w \leftarrow w.right$
        else
            return $w.x$
    return $nil$
```

Two examples of searches in a binary search tree are shown in Figure 4.6. 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 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:

```
find($x$)
    $w \leftarrow r$
    $z \leftarrow nil$
    while $w \neq nil$ do
        if $x < w.x$ then
            $z \leftarrow w$
            $w \leftarrow w.left$
        else if $x > w.x$
            $w \leftarrow w.right$
        else
            return $w.x$
    if $z = nil$ then return $nil$
    return $z.x$
```

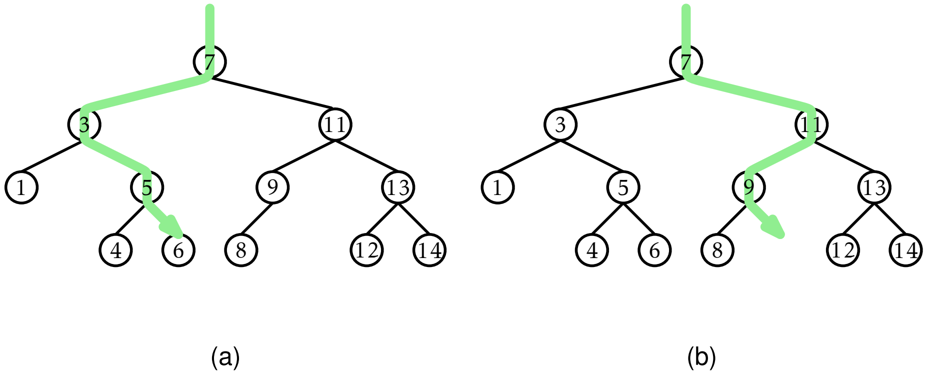


Figure 4.6: An example of (a) a successful search (for 6) and (b) an unsuccessful search (for 10) in a binary search tree.

4.2.2 Addition

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 $p.x$.

```
add($x$)
    $p \leftarrow$ find_last($x$)
    return add_child($p$, new_node($x$))
```

```
find_last($x$)
    $w \leftarrow r$
    $prev \leftarrow nil$
    while $w \neq nil$ do
        $prev \leftarrow w$
        if $(x < w.x)$ then
            $w \leftarrow w.left$
        else if $(x > w.x)$
            $w \leftarrow w.right$
        else
            return $w$
    return $prev$
```

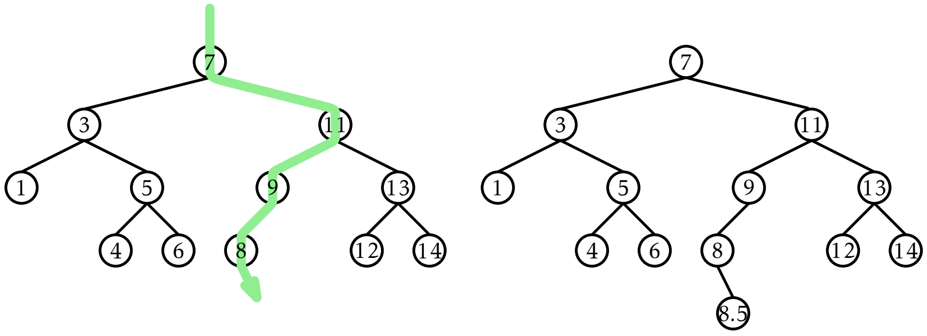


Figure 4.7: Inserting the value 8.5 into a binary search tree.

```
add_child($p, u$)
    if $p = nil$ then
        $r \leftarrow u$ # inserting into empty tree
    else
        if $u.x < p.x$ then
            $p.left \leftarrow u$
        else if $u.x > p.x$
            $p.right \leftarrow u$
        else
            return $false$ # u.x is already in the tree
        $u.parent \leftarrow p$
    $n \leftarrow n + 1$
    return $true$
```

An example is shown in Figure 4.7. The most time-consuming part of this process is the initial search for x, which takes an amount of time proportional to the height of the newly added node u. In the worst case, this is equal to the height of the BinarySearchTree. In the average case, it is O(1), because half the values are leaves, and on average, one only has to traverse halfway up the *values* in the tree.

4.2.3 Removal

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 (see Figure 4.8):

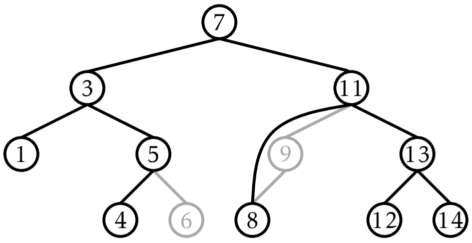


Figure 4.8: Removing a leaf (6) or a node with only one child (9) is easy.

```
splice($u$)
    if $u.left \neq nil$ then
        $s \leftarrow u.left$
    else
        $s \leftarrow u.right$
    if $u = r$ then
        $r \leftarrow s$
        $p \leftarrow nil$
    else
        $p \leftarrow u.parent$
        if $p.left = u$ then
            $p.left \leftarrow s$
        else
            $p.right \leftarrow s$
    if $s \neq nil$ then
        $s.parent \leftarrow p$
    $n \leftarrow n - 1$
```

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 and such that $w.x$ can replace $u.x$. To maintain the binary search tree property, the value $w.x$ should be close to the value of $u.x$. For example, choosing w such that $w.x$ is the smallest value greater than $u.x$ will work. 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 4.9).

```
remove_node($u$)
    if $u.left = nil$ or $u.right = nil$ then
        splice($u$)
    else
```

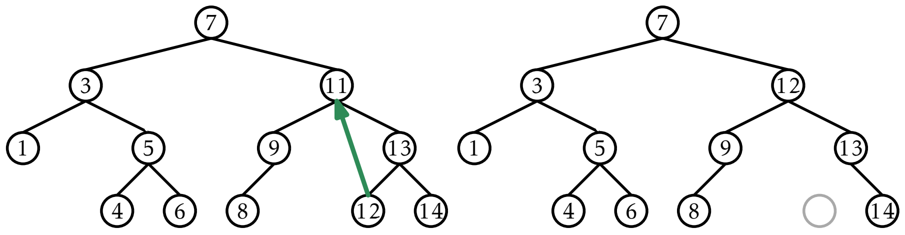


Figure 4.9: Deleting a value (11) from a node, u, with two children is done by replacing u's value with the smallest value in the right subtree of u.

```
$w \leftarrow u.right$
**while** $w.left \neq nil$ **do**
    $w \leftarrow w.left$
$u.x \leftarrow w.x$
splice$(w)$
```

4.2.4 Summary

The $\mathrm{find}(x)$, $\mathrm{add}(x)$, and $\mathrm{remove}(x)$ 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 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 4.1. *BinarySearchTree implements the SSet interface and supports the operations* $\mathrm{add}(x)$, $\mathrm{remove}(x)$, *and* $\mathrm{find}(x)$ *in* $O(n)$ *time per operation.*

The problem with the BinarySearchTree structure is that it can become *unbalanced*. Instead of looking like the tree in Figure 4.5 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. $O(\log n)$ *expected* time operations can be achieved with randomization. $O(\log n)$ *amortized* time operations can be achieved with partial rebuilding operations. $O(\log n)$ *worst-case* time operations can be achieved by simulating a tree that is not binary: one in which nodes can have up to four children.

4.3 Discussion and Exercises

Binary trees have been used to model relationships for 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 [27, Section 6.2.2]. Further references to specific kinds of binary search trees are provided in subsequent chapters.

When implementing a binary tree from scratch, there are several design decisions to be made. One of these is the question of whether or not each node stores a pointer to its parent. If most of the operations simply follow a root-to-leaf path, then parent pointers are unnecessary, waste space, and are a potential source of coding errors. On the other hand, the lack of parent pointers means that tree traversals must be done recursively or with the use of an explicit stack. Some other methods (like inserting or deleting into some kinds of balanced binary search trees) are also complicated by the lack of parent pointers.

Another design decision is concerned with how to store the parent, left child, and right child pointers at a node. In the implementation given here, these pointers are stored as separate variables. Another option is to store them in an array, p, of length 3, so that $u.p[0]$ is the left child of u, $u.p[1]$ is the right child of u, and $u.p[2]$ is the parent of u. Using an array this way means that some sequences of **if** statements can be simplified into algebraic expressions.

An example of such a simplification occurs during tree traversal. If a traversal arrives at a node u from $u.p[i]$, then the next node in the traversal is $u.p[(i+1) \bmod 3]$. Similar examples occur when there is left-right symmetry. For example, the sibling of $u.p[i]$ is $u.p[(i+1) \bmod 2]$. This trick works whether $u.p[i]$ is a left child ($i = 0$) or a right child ($i = 1$) of u. In some cases this means that some complicated code that would otherwise need to have both a left version and right version can be written only once.

Exercise 4.1. Prove that a binary tree having $n \geq 1$ nodes has $n-1$ edges.

Exercise 4.2. Prove that a binary tree having $n \geq 1$ real nodes has $n+1$ external nodes.

Exercise 4.3. Prove that, if a binary tree, T, has at least one leaf, then either (a) T's root has at most one child or (b) T has more than one leaf.

Exercise 4.4. Implement a non-recursive method, $\mathrm{size2}(u)$, that computes the size of the subtree rooted at node u.

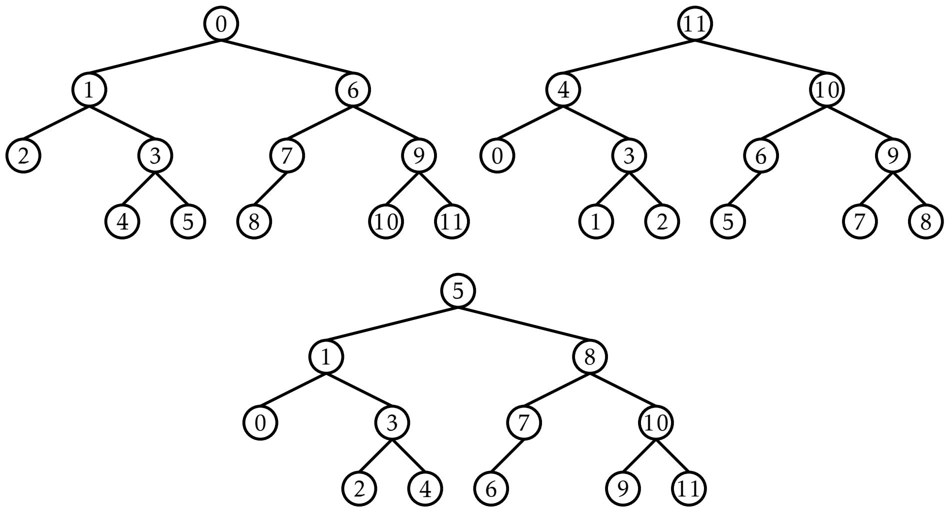


Figure 4.10: Pre-order, post-order, and in-order numberings of a binary tree.

Exercise 4.5. Write a non-recursive method, height2(u), that computes the height of node u in a BinaryTree.

Exercise 4.6. A binary tree is *size-balanced* if, for every node u, the size of the subtrees rooted at $u.left$ and $u.right$ differ by at most one. Write a recursive method, is_balanced(), that tests if a binary tree is balanced. Your method should run in $O(n)$ time. (Be sure to test your code on some large trees with different shapes; it is easy to write a method that takes much longer than $O(n)$ time.)

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. The pre/in/post-order numbering of a tree labels the nodes of a tree with the integers $0,\ldots,n-1$ in the order that they are encountered by a pre/in/post-order traversal. See Figure 4.10 for an example.

Exercise 4.7. Create a subclass of BinaryTree whose nodes have fields for storing pre-order, post-order, and in-order numbers. Write recursive methods pre_orderNumber(), in_orderNumber(), and post_orderNumbers() that assign these numbers correctly. These methods should each run in $O(n)$ time.

Exercise 4.8. Implement the non-recursive functions next_preOrder(u), next_inOrder(u), and next_postOrder(u) that return the node that follows u in a pre-order, in-order, or post-order traversal, respectively. These functions should take amortized constant time; if we start at any node u and repeatedly call one of these functions and assign the return value to u until $u = nil$, then the cost of all these calls should be $O(n)$.

Exercise 4.9. Suppose we are given a binary tree with pre-, post-, and in-order numbers assigned to the nodes. Show how these numbers can be used to answer each of the following questions in constant time:

1. Given a node u, determine the size of the subtree rooted at u.
2. Given a node u, determine the depth of u.
3. Given two nodes u and w, determine if u is an ancestor of w

Exercise 4.10. Suppose you are given a list of nodes with pre-order and in-order numbers assigned. Prove that there is at most one possible tree with this pre-order/in-order numbering and show how to construct it.

Exercise 4.11. Show that the shape of any binary tree on n nodes can be represented using at most $2(n-1)$ bits. (Hint: think about recording what happens during a traversal and then playing back that recording to reconstruct the tree.)

Exercise 4.12. Illustrate what happens when we add the values 3.5 and then 4.5 to the binary search tree in Figure 4.5.

Exercise 4.13. Illustrate what happens when we remove the values 3 and then 5 from the binary search tree in Figure 4.5.

Exercise 4.14. Implement a BinarySearchTree method, $\mathrm{get\_lE}(x)$, that returns a list of all items in the tree that are less than or equal to x. The running time of your method should be $O(n'+h)$ where n' is the number of items less than or equal to x and h is the height of the tree.

Exercise 4.15. Describe how to add the elements $\{1,\ldots,n\}$ to an initially empty BinarySearchTree in such a way that the resulting tree has height $n-1$. How many ways are there to do this?

Exercise 4.16. If we have some BinarySearchTree and perform the operations $\mathrm{add}(x)$ followed by $\mathrm{remove}(x)$ (with the same value of x) do we necessarily return to the original tree?

Exercise 4.17. Can a $\mathrm{remove}(x)$ operation increase the height of any node in a BinarySearchTree? If so, by how much?

Exercise 4.18. Can an $\mathrm{add}(x)$ operation increase the height of any node in a BinarySearchTree? Can it increase the height of the tree? If so, by how much?

Exercise 4.19. 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 calls to the $\mathrm{add}(x)$ and $\mathrm{remove}(x)$ operations, but this should not increase the cost of these operations by more than a constant factor.

Chapter 5

Heaps

In this chapter, we discuss two implementations of the extremely useful priority Queue data structure. Both of these structures are a special kind of binary tree called a *heap*, which means “a disorganized pile.” This is in contrast to binary search trees that can be thought of as a highly organized pile.

The first heap implementation uses an array to simulate a complete binary tree. This very fast implementation is the basis of one of the fastest known sorting algorithms, namely heapsort (see Section 6.1.3). The second implementation is based on more flexible binary trees. It supports a $\mathrm{meld}(h)$ operation that allows the priority queue to absorb the elements of a second priority queue h.

5.1 BinaryHeap: An Implicit Binary Tree

Our first implementation of a (priority) Queue is based on a technique that is over four hundred years old. *Eytzinger’s method* allows us to represent a complete binary tree as an array by laying out the nodes of the tree in breadth-first order (see Section 4.1.2). 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 5.1.

If we apply Eytzinger’s method to a sufficiently large tree, some patterns emerge. The left child of the node at index i is at index $\mathrm{left}(i) = 2i + 1$ and the right child of the node at index i is at index $\mathrm{right}(i) = 2i + 2$. The parent of the node at index i is at index $\mathrm{parent}(i) = (i - 1)/2$.

```
$\mathrm{left}(i)$
    return $2 \cdot i + 1$

$\mathrm{right}(i)$
    return $2 \cdot (i + 1)$
```

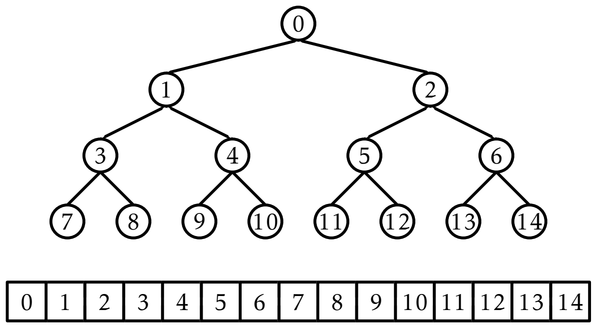


Figure 5.1: Eytzinger's method represents a complete binary tree as an array.

```
parent($i$)
    return $(i-1)\operatorname{div}2$
```

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 $\mathrm{parent}(i)$, 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:

```
initialize()
    $a \leftarrow \mathrm{new\_array}(1)$
    $n \leftarrow 0$
```

Implementing the $\mathrm{add}(x)$ operation is fairly straightforward. As with all array-based structures, we first check to see if a is full (by checking if $\mathrm{length}(a) = n$) 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 5.2.

```
add($x$)
    if $\mathrm{length}(a) < n+1$ then
        resize()
    $a[n] \leftarrow x$
    $n \leftarrow n+1$
    bubble_up($n-1$)
    **return** $true$

bubble_up($i$)
    $p \leftarrow$ parent($i$)
    **while** $i > 0$ **and** $a[i] < a[p]$ **do**
        $a[i], a[p] \leftarrow a[p], a[i]$
        $i \leftarrow p$
        $p \leftarrow$ 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 $a[n-1]$, delete that value, and decrement n. Unfortunately, the new root element is now 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.

```
remove()
    $x \leftarrow a[0]$
    $a[0] \leftarrow a[n-1]$
    $n \leftarrow n-1$
    trickle_down(0)
    **if** $3 \cdot n <$ length($a$) **then**
        resize()
    **return** $x$

trickle_down($i$)
    **while** $i \geq 0$ **do**
        $j \leftarrow -1$
        $r \leftarrow$ right($i$)
        **if** $r < n$ **and** $a[r] < a[i]$ **then**
            $\ell \leftarrow$ left($i$)
            **if** $a[\ell] < a[r]$ **then**
                $j \leftarrow \ell$
            **else**
                $j \leftarrow r$
        **else**
            $\ell \leftarrow$ left($i$)
```

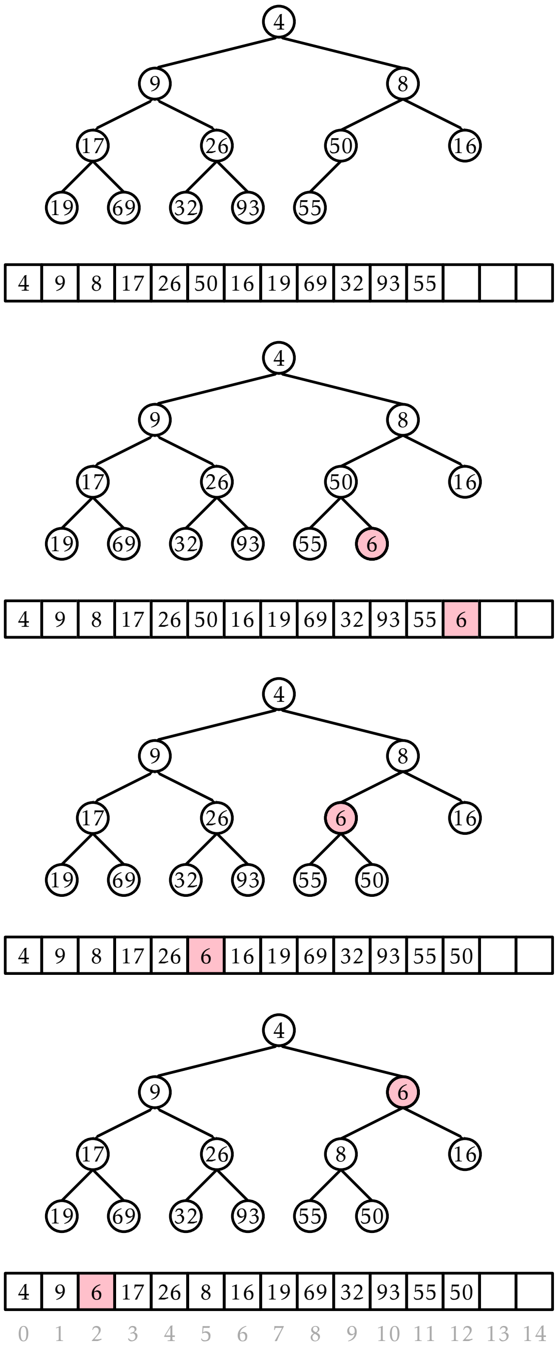


Figure 5.2: Adding the value 6 to a BinaryHeap.

```
        if $\ell < n$ and $a[\ell] < a[i]$ then
            $j \leftarrow \ell$
    if $j \geq 0$ then
        $a[j], a[i] \leftarrow a[i], a[j]$
    $i \leftarrow j$
```

As with other array-based structures, we will ignore the time spent in calls to $\mathrm{resize}()$, since these can be accounted for using the amortization argument from Lemma 2.1. The running times of both $\mathrm{add}(x)$ and $\mathrm{remove}()$ then depend on the height of the (implicit) binary tree. Luckily, 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

$$n \geq 2^h \ .$$

Taking logarithms on both sides of this equation gives

$$h \leq \log n \ .$$

Therefore, both the $\mathrm{add}(x)$ and $\mathrm{remove}()$ operation run in $O(\log n)$ time.

5.1.1 Summary

The following theorem summarizes the performance of a BinaryHeap:

Theorem 5.1. *A BinaryHeap implements the (priority) Queue interface. Ignoring the cost of calls to* $\mathrm{resize}()$*, a BinaryHeap supports the operations* $\mathrm{add}(x)$ *and* $\mathrm{remove}()$ *in* $O(\log n)$ *time per operation.*

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

5.2 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 $\mathrm{add}(x)$ and $\mathrm{remove}()$ operations in a MeldableHeap are implemented in terms of the $\mathrm{merge}(h_1, h_2)$ operation. This operation takes two heap nodes h_1 and h_2 and merges them, returning a heap node that is the root of a heap that contains all elements in the subtree rooted at h_1 and all elements in the subtree rooted at h_2.

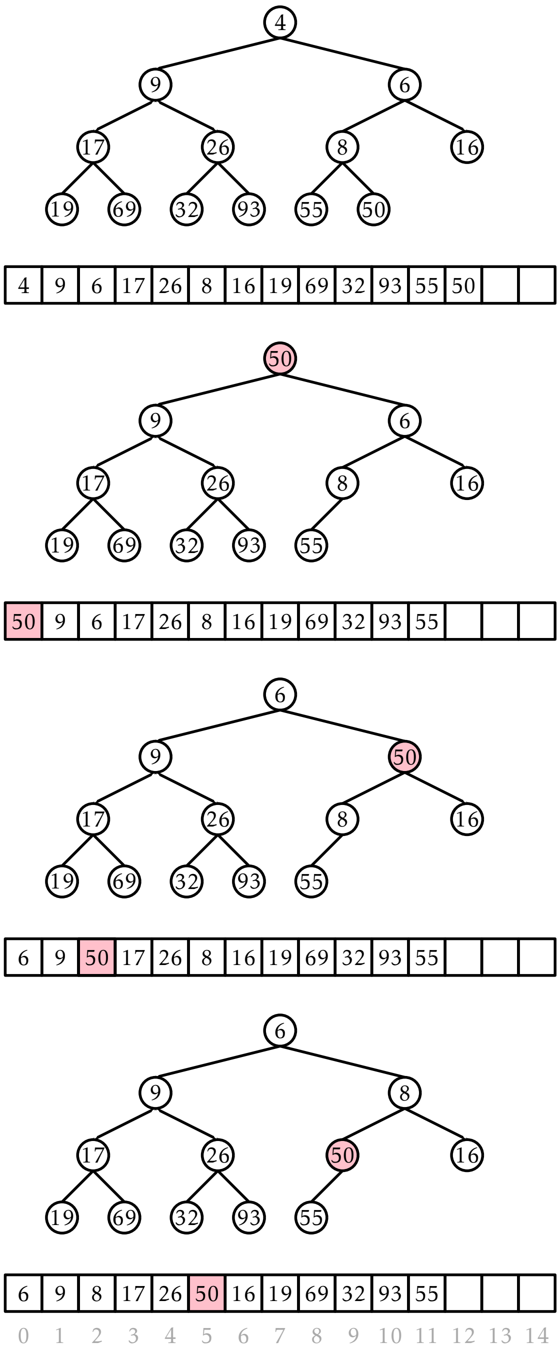


Figure 5.3: Removing the minimum value, 4, from a BinaryHeap.

The nice thing about a $\mathrm{merge}(h_1,h_2)$ operation is that it can be defined recursively. See Figure 5.4. If either h_1 or h_2 is nil, then we are merging with an empty set, so we return h_2 or h_1, respectively. Otherwise, assume $h_1.x \le h_2.x$ since, if $h_1.x > h_2.x$, then we can reverse the roles of h_1 and h_2. Then we know that the root of the merged heap will contain $h_1.x$, and we can recursively merge h_2 with $h_1.left$ or $h_1.right$, as we wish. This is where randomization comes in, and we toss a coin to decide whether to merge h_2 with $h_1.left$ or $h_1.right$:

```
merge($h_1$,$h_2$)
  if $h_1 = nil$ then return $h_2$
  if $h_2 = nil$ then return $h_1$
  if $h_2.x < h_1.x$ then $(h_1, h2) \leftarrow (h_2, h_1)$
  if random_bit() then
    $h_1.left \leftarrow \mathrm{merge}(h_1.left, h_2)$
    $h_1.left.parent \leftarrow h1$
  else
    $h_1.right \leftarrow \mathrm{merge}(h_1.right, h_2)$
    $h_1.right.parent \leftarrow h1$
  return $h_1$
```

In the next section, we show that $\mathrm{merge}(h_1,h_2)$ runs in $O(\log n)$ expected time, where n is the total number of elements in h_1 and h_2.

With access to a $\mathrm{merge}(h_1,h_2)$ operation, the $\mathrm{add}(x)$ operation is easy. We create a new node u containing x and then merge u with the root of our heap:

```
add($x$)
  $u \leftarrow$ new_node($x$)
  $r \leftarrow \mathrm{merge}(u, r)$
  $r.parent \leftarrow nil$
  $n \leftarrow n + 1$
  return $true$
```

This takes $O(\log(n+1)) = O(\log n)$ expected time.

The $\mathrm{remove}()$ 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:

```
remove()
  $x \leftarrow r.x$
  $r \leftarrow \mathrm{merge}(r.left, r.right)$
```

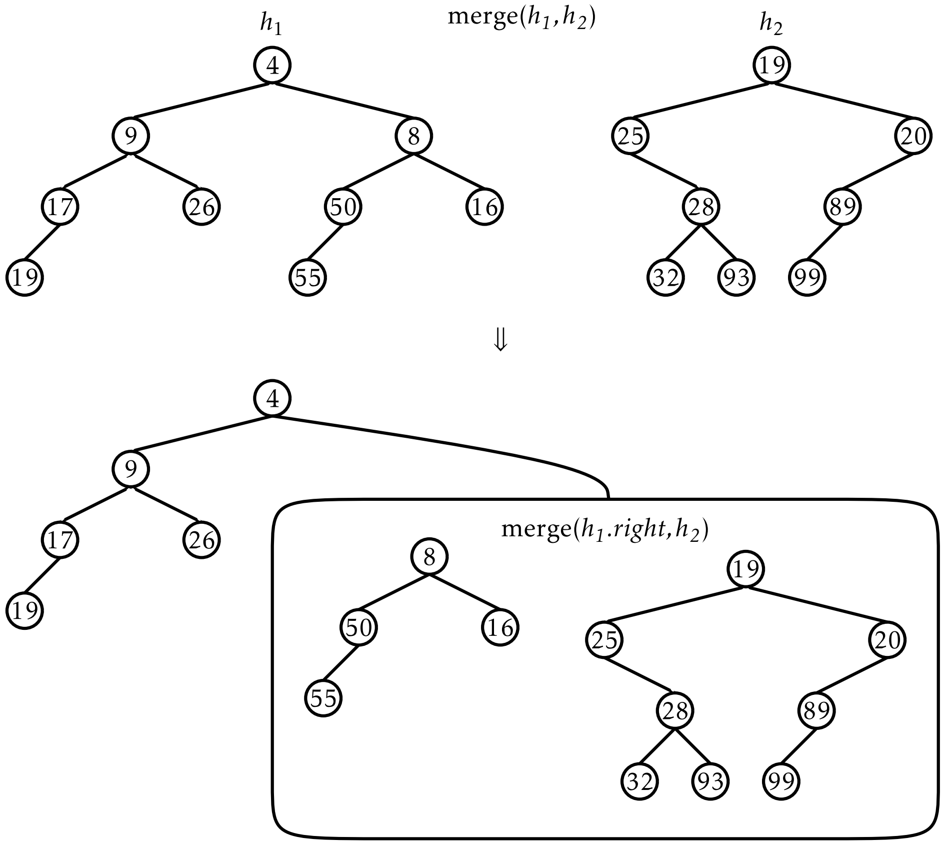


Figure 5.4: Merging h_1 and h_2 is done by merging h_2 with one of $h_1.left$ or $h_1.right$.

```
if $r \neq nil$ then $r.parent \leftarrow nil$
$n \leftarrow n - 1$
return $x$
```

Again, this takes $O(\log n)$ expected time.

Additionally, a MeldableHeap can implement many other operations in $O(\log n)$ expected time, including:

- $\mathrm{remove}(u)$: remove the node u (and its key $u.x$) from the heap.
- $\mathrm{absorb}(h)$: 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 $\mathrm{merge}(h_1,h_2)$ operations that each take $O(\log n)$ expected time.

5.2.1 Analysis of $\mathrm{merge}(h_1,h_2)$

The analysis of $\mathrm{merge}(h_1,h_2)$ is based on the analysis of a random walk in a binary tree. A *random walk* in a binary tree starts at the root of the tree. At each step in the random walk, a coin is tossed and, depending on the result of this coin toss, the walk proceeds to the left or to the right child of the current node. The walk ends when it falls off the tree (the current node becomes nil).

The following lemma is somewhat remarkable because it does not depend at all on the shape of the binary tree:

Lemma 5.1. *The expected length of a random walk in a binary tree with n nodes is at most $\log(n+1)$.*

Proof. The proof is by induction on n. In the base case, $n = 0$ and the walk has length $0 = \log(n+1)$. Suppose now that the result is true for all non-negative integers $n' < n$.

Let n_1 denote the size of the root's left subtree, so that $n_2 = n - n_1 - 1$ 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 n_1 or n_2. By our inductive hypothesis, the expected length of the walk is then

$$\mathrm{E}[W] = 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \ ,$$

since each of n_1 and n_2 are less than n. Since $\log$ is a concave function, $\mathrm{E}[W]$ is maximized when $n_1 = n_2 = (n-1)/2$. Therefore, the expected number of steps taken by the random walk is

$$\begin{aligned}
\mathrm{E}[W] &= 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \\
&\le 1 + \log((n-1)/2+1) \\
&= 1 + \log((n+1)/2) \\
&= \log(n+1) \ . \qquad \square
\end{aligned}$$

We make a quick digression to note that, for readers who know a little about information theory, the proof of Lemma 5.1 can be stated in terms of entropy.

Information Theoretic Proof of Lemma 5.1. Let d_i denote the depth of the ith external node and recall that a binary tree with n nodes has $n+1$ 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

$$H = \sum_{i=0}^{n} p_i d_i = \sum_{i=0}^{n} p_i \log\left(2^{d_i}\right) = \sum_{i=0}^{n} p_i \log(1/p_i)$$

The right hand side of this equation is easily recognizable as the entropy of a probability distribution over $n+1$ elements. A basic fact about the entropy of a distribution over $n+1$ elements is that it does not exceed $\log(n+1)$, which proves the lemma. $\square$

With this result on random walks, we can now easily prove that the running time of the $\mathrm{merge}(h_1,h_2)$ operation is $O(\log n)$.

Lemma 5.2. *If h_1 and h_2 are the roots of two heaps containing n_1 and n_2 nodes, respectively, then the expected running time of $\mathrm{merge}(h_1,h_2)$ is at most $O(\log n)$, where $n = n_1 + n_2$.*

Proof. Each step of the merge algorithm takes one step of a random walk, either in the heap rooted at h_1 or the heap rooted at h_2. The algorithm terminates when either of these two random walks fall out of its corresponding tree (when $h_1 = nil$ or $h_2 = nil$). Therefore, the expected number of steps performed by the merge algorithm is at most

$$\log(n_1+1) + \log(n_2+1) \le 2\log n \ . \qquad \square$$

5.2.2 Summary

The following theorem summarizes the performance of a MeldableHeap:

Theorem 5.2. *A MeldableHeap implements the (priority) Queue interface. A MeldableHeap supports the operations $\mathrm{add}(x)$ and $\mathrm{remove}()$ in $O(\log n)$ expected time per operation.*

5.3 Discussion and Exercises

The implicit representation of a complete binary tree as an array, or list, seems to have been first proposed by Eytzinger [14]. He used this representation in books containing pedigree family trees of noble families. The BinaryHeap data structure described here was first introduced by Williams [41].

The randomized MeldableHeap data structure described here appears to have first been proposed by Gambin and Malinowski [19]. Other meldable heap implementations exist, including leftist heaps [6, 27, Section 5.3.2], binomial heaps [40], Fibonacci heaps [17], pairing heaps [16], and skew heaps [38], although none of these are as simple as the MeldableHeap structure.

Some of the above structures also support a $\mathrm{decrease\_key}(u,y)$ operation in which the value stored at node u is decreased to y. (It is a pre-condition that $y \leq u.x$.) In most of the preceding structures, this operation can be supported in $O(\log n)$ time by removing node u and adding y. However, some of these structures can implement $\mathrm{decrease\_key}(u,y)$ more efficiently. In particular, $\mathrm{decrease\_key}(u,y)$ takes $O(1)$ amortized time in Fibonacci heaps and $O(\log\log n)$ amortized time in a special version of pairing heaps [13]. This more efficient $\mathrm{decrease\_key}(u,y)$ operation has applications in speeding up several graph algorithms, including Dijkstra's shortest path algorithm [17].

Exercise 5.1. Illustrate the addition of the values 7 and then 3 to the BinaryHeap shown at the end of Figure 5.2.

Exercise 5.2. Illustrate the removal of the next two values (6 and 8) on the BinaryHeap shown at the end of Figure 5.3.

Exercise 5.3. Implement the $\mathrm{remove}(i)$ method, that removes the value stored in $a[i]$ in a BinaryHeap. This method should run in $O(\log n)$ time. Next, explain why this method is not likely to be useful.

Exercise 5.4. A d-ary tree is a generalization of a binary tree in which each internal node has d children. Using Eytzinger's method it is also possible to represent complete d-ary trees using arrays. Work out the equations that, given an index i, determine the index of i's parent and each of i's d children in this representation.

Exercise 5.5. Using what you learned in Exercise 5.4, design and implement a *DaryHeap*, the d-ary generalization of a BinaryHeap. Analyze the running times of operations on a DaryHeap and test the performance of your DaryHeap implementation against that of the BinaryHeap implementation given here.

Exercise 5.6. Illustrate the addition of the values 17 and then 82 in the MeldableHeap h_1 shown in Figure 5.4. Use a coin to simulate a random bit when needed.

Exercise 5.7. Illustrate the removal of the next two values (4 and 8) in the MeldableHeap h_1 shown in Figure 5.4. Use a coin to simulate a random bit when needed.

Exercise 5.8. Implement the $\mathrm{remove}(u)$ method, that removes the node u from a MeldableHeap. This method should run in $O(\log n)$ expected time.

Exercise 5.9. Show how to find the second smallest value in a BinaryHeap or MeldableHeap in constant time.

Exercise 5.10. Show how to find the kth smallest value in a BinaryHeap or MeldableHeap in $O(k \log k)$ time. (Hint: Using another heap might help.)

Exercise 5.11. Suppose you are given k sorted lists, of total length n. Using a heap, show how to merge these into a single sorted list in $O(n \log k)$ time. (Hint: Starting with the case $k = 2$ can be instructive.)

Chapter 6

Sorting Algorithms

This chapter discusses algorithms for sorting a set of 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 $O(n \log n)$ time. As it turns out, all three algorithms are asymptotically optimal; no algorithm that uses only comparisons can avoid doing roughly $n \log n$ comparisons in the worst case and even the average case.

Before continuing, we should note that any of the SSet or priority Queue implementations presented in previous chapters can also be used to obtain an $O(n \log n)$ time sorting algorithm. For example, we can sort n items by performing n $\mathrm{add}(x)$ operations followed by n $\mathrm{remove}()$ operations on a BinaryHeap or MeldableHeap. Alternatively, we can use n $\mathrm{add}(x)$ operations on any of the binary search tree data structures and then perform an in-order traversal (Exercise 4.8) to extract the elements in sorted order. However, in both cases we go through a lot of overhead to build a structure that is never fully used. Sorting is such an important problem that it is worthwhile developing direct methods that are as fast, simple, and space-efficient as possible.

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 n integers in the range $\{0, \dots, n^c - 1\}$ in $O(cn)$ time.

6.1 Comparison-Based Sorting

In this section, we present three sorting algorithms: merge-sort, quicksort, and heap-sort. Each of these algorithms takes an input array a and sorts the elements of a into non-decreasing order in $O(n \log n)$ (expected) time. These algorithms are all *comparison-based*. These algorithms don't care what type of data is being sorted; the only operation they do on the data is comparisons using the $\mathrm{compare}(a, b)$ method. Recall, from

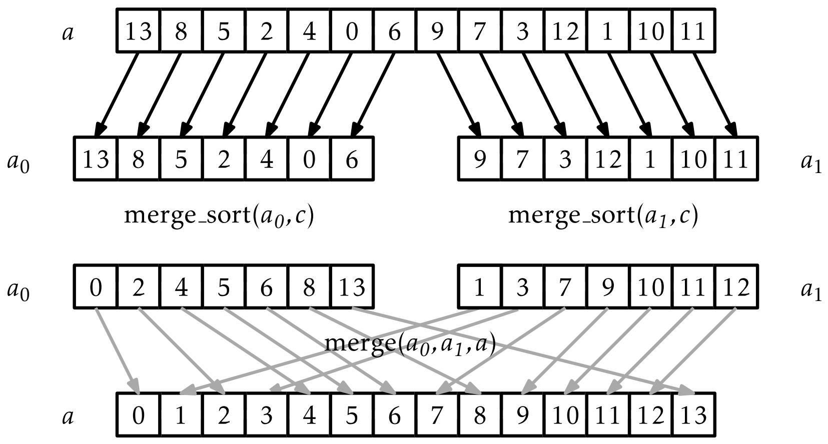


Figure 6.1: The execution of $\mathrm{merge\_sort}(a,c)$

Section 1.2.4, that $\mathrm{compare}(a,b)$ returns a negative value if $a < b$, a positive value if $a > b$, and zero if $a = b$.

6.1.1 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 halves, $a_0 = a[0],\ldots,a[n/2-1]$ and $a_1 = a[n/2],\ldots,a[n-1]$. We recursively sort a_0 and a_1, and then we merge (the now sorted) a_0 and a_1 to get our fully sorted array a:

```
merge_sort($a$)
    if length($a$) $\leq 1$ then
        return $a$
    $m \leftarrow$ length($a$) div 2
    $a_0 \leftarrow$ merge_sort($a[m]$)
    $a_1 \leftarrow$ merge_sort($a[m]$)
    merge($a_0$,$a_1$,$a$)
    return $a$
```

An example is shown in Figure 6.1.

Compared to sorting, merging the two sorted arrays a_0 and a_1 is fairly easy. We add elements to a one at a time. If a_0 or a_1 is empty, then we add the next elements from the other (non-empty) array. Otherwise, we take the minimum of the next element in a_0 and the next element in a_1 and add it to a:

```
merge($a_0, a_1, a$)
    $i_0 \leftarrow i1 \leftarrow 0$
    for $i$ in $0, 1, 2, \ldots, \text{length}(a) - 1$ do
        if $i_0 = \text{length}(a_0)$ then
            $a[i] \leftarrow a_1[i_1]$
            $i_1 \leftarrow i2$
        else if $i_1 = \text{length}(a_1)$
            $a[i] \leftarrow a_0[i_0]$
            $i_0 \leftarrow i1$
        else if $a_0[i_0] \leq a_1[i_1]$
            $a[i] \leftarrow a_0[i_0]$
            $i_0 \leftarrow i1$
        else
            $a[i] \leftarrow a_1[i_1]$
            $i_1 \leftarrow i2$
```

Notice that the $\text{merge}(a_0, a_1, a, c)$ algorithm performs at most $n-1$ comparisons before running out of elements in one of a_0 or a_1.

To understand the running-time of merge-sort, it is easiest to think of it in terms of its recursion tree. Suppose for now that n is a power of two, so that $n = 2^{\log n}$, and $\log n$ is an integer. Refer to Figure 6.2. Merge-sort turns the problem of sorting n elements into two problems, each of sorting $n/2$ elements. These two subproblem are then turned into two problems each, for a total of four subproblems, each of size $n/4$. These four subproblems become eight subproblems, each of size $n/8$, and so on. At the bottom of this process, $n/2$ subproblems, each of size two, are converted into n problems, each of size one. For each subproblem of size $n/2^i$, the time spent merging and copying data is $O(n/2^i)$. Since there are 2^i subproblems of size $n/2^i$, the total time spent working on problems of size 2^i, not counting recursive calls, is

$$2^i \times O(n/2^i) = O(n) \ .$$

Therefore, the total amount of time taken by merge-sort is

$$\sum_{i=0}^{\log n} O(n) = O(n \log n) \ .$$

The proof of the following theorem is based on preceding analysis, but has to be a little more careful to deal with the cases where n is not a power of 2.

Theorem 6.1. *The merge_sort(a) algorithm runs in $O(n \log n)$ time and performs at most $n \log n$ comparisons.*

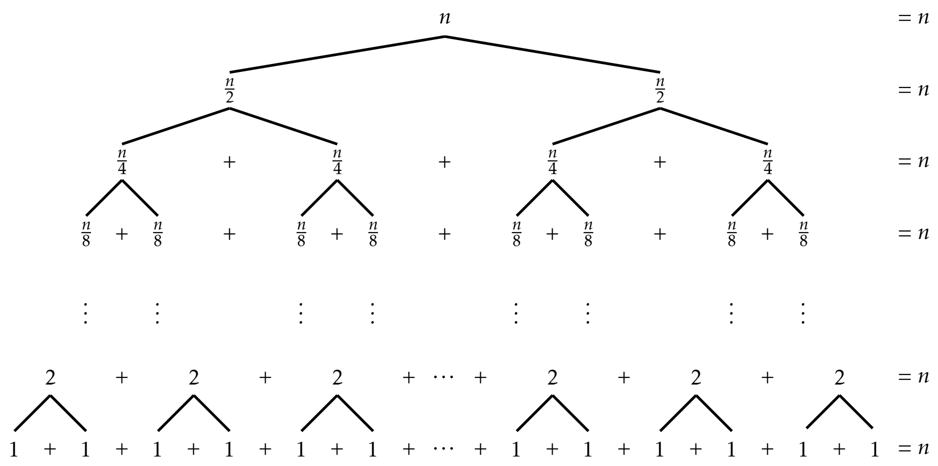


Figure 6.2: The merge-sort recursion tree.

Proof. The proof is by induction on n. The base case, in which $n = 1$, is trivial; when presented with an array of length 0 or 1 the algorithm simply returns without performing any comparisons.

Merging two sorted lists of total length n requires at most $n-1$ comparisons. Let $C(n)$ denote the maximum number of comparisons performed by $\mathrm{merge\_sort}(a,c)$ on an array a of length n. If n is even, then we apply the inductive hypothesis to the two subproblems and obtain

$$\begin{aligned} C(n) &\le n-1+2C(n/2) \\ &\le n-1+2((n/2)\log(n/2)) \\ &= n-1+n\log(n/2) \\ &= n-1+n\log n-n \\ &< n\log n \ . \end{aligned}$$

The case where n is odd is slightly more complicated. For this case, we use two inequalities that are easy to verify:

$$\log(x+1) \le \log(x)+1 \ , \tag{6.1}$$

for all $x \ge 1$ and

$$\log(x+1/2)+\log(x-1/2) \le 2\log(x) \ , \tag{6.2}$$

for all $x \ge 1/2$. Inequality (6.1) comes from the fact that $\log(x)+1 = \log(2x)$ while (6.2) follows from the fact that $\log$ is a concave function. With these tools in hand we have, for odd n,

$$\begin{aligned}
C(n) &\le n-1+C(\lceil n/2\rceil)+C(\lfloor n/2\rfloor) \\
&\le n-1+\lceil n/2\rceil\log\lceil n/2\rceil+\lfloor n/2\rfloor\log\lfloor n/2\rfloor \\
&= n-1+(n/2+1/2)\log(n/2+1/2)+(n/2-1/2)\log(n/2-1/2) \\
&\le n-1+n\log(n/2)+(1/2)(\log(n/2+1/2)-\log(n/2-1/2)) \\
&\le n-1+n\log(n/2)+1/2 \\
&< n+n\log(n/2) \\
&= n+n(\log n-1) \\
&= n\log n \ . \qquad \square
\end{aligned}$$

6.1.2 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 of its work upfront.

Quicksort 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 6.3.

```
quick_sort($a$)
    quick_sort($a, 0, \mathrm{length}(a)$)

quick_sort($a, i, n$)
    **if** $n \le 1$ **then return**
    $x \leftarrow a[i + \mathrm{random\_int}(n)]$
    $(p, j, q) \leftarrow (i-1, i, i+n)$
    **while** $j < q$ **do**
        **if** $a[j] < x$ **then**
            $p \leftarrow p+1$
            $a[j], a[p] \leftarrow a[p], a[j]$
            $j \leftarrow j+1$
        **else if** $a[j] > x$
            $q \leftarrow q-1$
            $a[j], a[q] \leftarrow a[q], a[j]$
        **else**
        $j \leftarrow j+1$
    quick_sort($a, i, p-i+1$)
    quick_sort($a, q, n-(q-i)$)
```

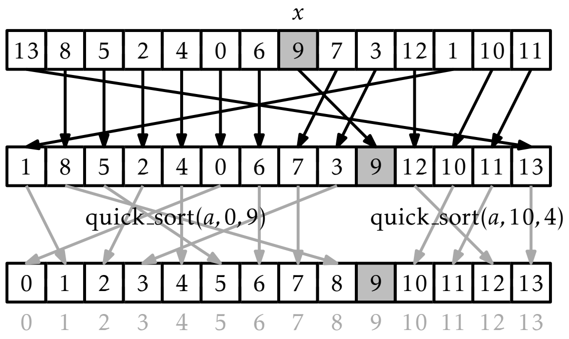


Figure 6.3: An example execution of $\mathrm{quick\_sort}(a,0,14)$

All of this is done in place, so that instead of making copies of subarrays being sorted, the $\mathrm{quick\_sort}(a,i,n,c)$ method only sorts the subarray $a[i],\ldots,a[i+n-1]$. Initially, this method is invoked with the arguments $\mathrm{quick\_sort}(a,0,\mathrm{length}(a),c)$.

At the heart of the quicksort algorithm is the in-place partitioning algorithm. This algorithm, without using any extra space, swaps elements in a and computes indices p and q so that

$$a[i]\begin{cases} < x & \text{if } 0 \le i \le p \\ = x & \text{if } p < i < q \\ > x & \text{if } q \le i \le n-1 \end{cases}$$

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 yet been processed.

Quicksort is very closely related to the random binary search trees studied in Section 4. 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 into the right.

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 to a statement about quicksort:

Lemma 6.1. *When quicksort is called to sort an array containing the integers $0,\dots,n-1$, the expected number of times element i is compared to a pivot element is at most $H_{i+1}+H_{n-i}$.*

A little summing up of harmonic numbers gives us the following theorem about the running time of quicksort:

Theorem 6.2. *When quicksort is called to sort an array containing n distinct elements, the expected number of comparisons performed is at most $2n\ln n+O(n)$.*

Proof. Let T be the number of comparisons performed by quicksort when sorting n distinct elements. Using Lemma 6.1 and linearity of expectation, we have:

$$\begin{aligned}
\mathrm{E}[T] &= \sum_{i=0}^{n-1}(H_{i+1}+H_{n-i}) \\
&= 2\sum_{i=1}^{n}H_i \\
&\le 2\sum_{i=1}^{n}H_n \\
&\le 2n\ln n + 2n = 2n\ln n + O(n)
\end{aligned}$$ □

Theorem 6.3 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 do not take part in either of the two subproblems.

Theorem 6.3. *The* $\mathrm{quick\_sort}(a,c)$ *method runs in $O(n\log n)$ expected time and the expected number of comparisons it performs is at most $2n\ln n+O(n)$.*

6.1.3 Heap-sort

The *heap-sort* algorithm is another in-place sorting algorithm. Heap-sort uses the binary heaps discussed in Section 5.1. 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 in an array, a, at array locations $a[0],\dots,a[n-1]$ with the smallest value stored at the root, $a[0]$. After transforming a into a BinaryHeap, the heap-sort algorithm repeatedly swaps $a[0]$ and $a[n-1]$, decrements n, and calls $\mathrm{trickle\_down}(0)$ so that $a[0],\dots,a[n-2]$ once again are a valid heap representation. When this process ends (because $n=0$) the elements of a are stored in decreasing

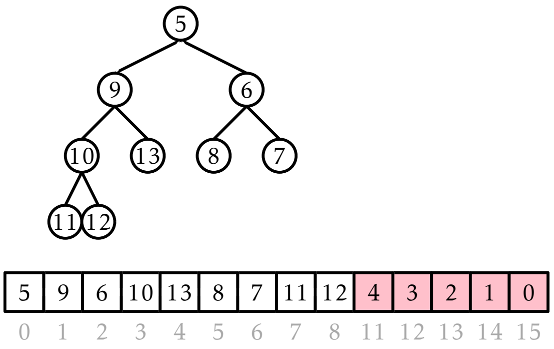


Figure 6.4: A snapshot of the execution of $\mathrm{heap\_sort}(a,c)$. The shaded part of the array is already sorted. The unshaded part is a BinaryHeap. During the next iteration, element 5 will be placed into array location 8.

order, so a is reversed to obtain the final sorted order.[1] Figure 6.4 shows an example of the execution of $\mathrm{heap\_sort}(a,c)$.

```
heap_sort($a$)
    $h \leftarrow \mathrm{BinaryHeap}()$
    $h.a \leftarrow a$
    $h.n \leftarrow \mathrm{length}(a)$
    $m \leftarrow h.n \operatorname{div} 2$
    for $i$ in $m-1, m-2, m-3, \ldots, 0$ do
        $h.\mathrm{trickle\_down}(i)$
    while $h.n > 1$ do
        $h.n \leftarrow h.n - 1$
        $h.a[h.n], h.a[0] \leftarrow h.a[0], h.a[h.n]$
        $h.\mathrm{trickle\_down}(0)$
    $a.\mathrm{reverse}()$
```

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 $O(n \log n)$ time by repeatedly calling the BinaryHeap $\mathrm{add}(x)$ 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 $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ have no children. In other words, each of $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ is a sub-heap of size 1. Now, working backwards, we can call $\mathrm{trickle\_down}(i)$ for each $i \in \{\lfloor n/2 \rfloor - 1, \ldots, 0\}$. This works, because by the time we call

[1]The algorithm could alternatively redefine the $\mathrm{compare}(x,y)$ function so that the heap sort algorithm stores the elements directly in ascending order.

$\mathrm{trickle\_down}(i)$, each of the two children of $a[i]$ are the root of a sub-heap, so calling $\mathrm{trickle\_down}(i)$ makes $a[i]$ into the root of its own subheap.

The interesting thing about this bottom-up strategy is that it is more efficient than calling $\mathrm{add}(x)$ n times. To see this, notice that, for $n/2$ elements, we do no work at all, for $n/4$ elements, we call $\mathrm{trickle\_down}(i)$ on a subheap rooted at $a[i]$ and whose height is one, for $n/8$ elements, we call $\mathrm{trickle\_down}(i)$ on a subheap whose height is two, and so on. Since the work done by $\mathrm{trickle\_down}(i)$ is proportional to the height of the sub-heap rooted at $a[i]$, this means that the total work done is at most

$$\sum_{i=1}^{\log n} O((i-1)n/2^i) \le \sum_{i=1}^{\infty} O(in/2^i) = O(n)\sum_{i=1}^{\infty} i/2^i = O(2n) = O(n) \ .$$

The second-last equality follows by recognizing that the sum $\sum_{i=1}^{\infty} i/2^i$ is equal, by definition of expected value, to the expected number of times we toss a coin up to and including the first time the coin comes up as heads.

The following theorem describes the performance of $\mathrm{heap\_sort}(a,c)$.

Theorem 6.4. *The* $\mathrm{heap\_sort}(a,c)$ *method runs in* $O(n\log n)$ *time and performs at most* $2n\log n + O(n)$ *comparisons.*

Proof. The algorithm runs in three 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 $O(n)$ time and performs $O(n)$ comparisons. Step 3 takes $O(n)$ time and performs no comparisons. Step 2 performs n calls to $\mathrm{trickle\_down}(0)$. The ith such call operates on a heap of size $n-i$ and performs at most $2\log(n-i)$ comparisons. Summing this over i gives

$$\sum_{i=0}^{n-i} 2\log(n-i) \le \sum_{i=0}^{n-i} 2\log n = 2n\log n$$

Adding the number of comparisons performed in each of the three steps completes the proof. □

6.1.4 A Lower-Bound for Comparison-Based Sorting

We have now seen three comparison-based sorting algorithms that each run in $O(n\log n)$ 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 $n\log n$ comparisons. This is not difficult to prove, but requires a little imagination. Ultimately, it follows from the fact that

$$\log(n!) = \log n + \log(n-1) + \cdots + \log(1) = n\log n - O(n) \ .$$

(Proving this fact is left as Exercise 6.10.)

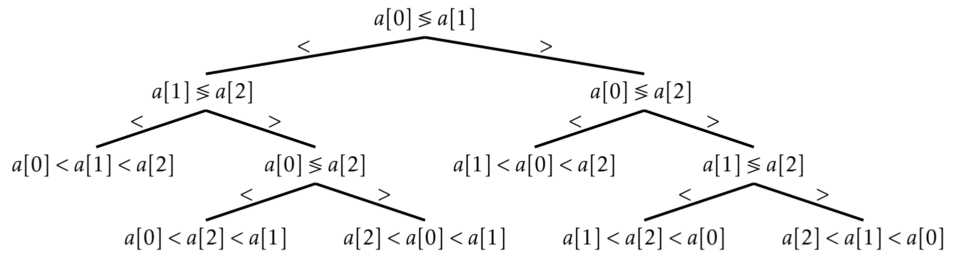


Figure 6.5: A comparison tree for sorting an array $a[0], a[1], a[2]$ of length $n \leftarrow 3$.

We will start by focusing our attention on deterministic algorithms like merge-sort and heap-sort and on a particular fixed value of n. Imagine such an algorithm is being used to sort n distinct elements. The key to proving the lower-bound is to observe that, for a deterministic algorithm with a fixed value of n, the first pair of elements that are compared is always the same. For example, in $\mathrm{heap\_sort}(a,c)$, when n is even, the first call to $\mathrm{trickle\_down}(i)$ is with $i \leftarrow n/2-1$ and the first comparison is between elements $a[n/2-1]$ and $a[n-1]$.

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, u, of this tree is labelled with a pair of indices $u.i$ and $u.j$. If $a[u.i] < a[u.j]$ the algorithm proceeds to the left subtree, otherwise it proceeds to the right subtree. Each leaf w of this tree is labelled with a permutation $w.p[0], \ldots, w.p[n-1]$ of $0, \ldots, n-1$. This permutation represents the one that is required to sort a if the comparison tree reaches this leaf. That is,

$$a[w.p[0]] < a[w.p[1]] < \cdots < a[w.p[n-1]] \enspace .$$

An example of a comparison tree for an array of size $n \leftarrow 3$ is shown in Figure 6.5.

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 in order to sort it. Consequently, the comparison tree must have at least $n!$ leaves; if not, then there are two distinct permutations that lead to the same leaf; therefore, the algorithm does not correctly sort at least one of these permutations.

For example, the comparison tree in Figure 6.6 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 $a[1] = 1, a[2] = 2, a[0] = 3$. However, on the input $3,2,1$, this node incorrectly outputs $a[1] = 2, a[2] = 1, a[0] = 3$. This discussion leads to the primary lower-bound for comparison-based algorithms.

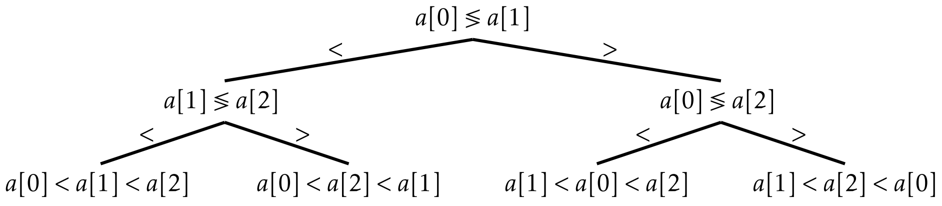


Figure 6.6: A comparison tree that does not correctly sort every input permutation.

Theorem 6.5. *For any deterministic comparison-based sorting algorithm $\mathcal{A}$ and any integer $n \geq 1$, there exists an input array a of length n such that $\mathcal{A}$ performs at least $\log(n!) = n\log n - O(n)$ comparisons when sorting a.*

Proof. By the preceding 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 a height of at least $\log k$. Therefore, the comparison tree for $\mathcal{A}$ has a leaf, w, with a depth of at least $\log(n!)$ and there is an input array a that leads to this leaf. The input array a is an input for which $\mathcal{A}$ does at least $\log(n!)$ comparisons. □

Theorem 6.5 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 $\log(n!)$ 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 assume that our decision trees have been "cleaned up" in the following way: Any node that can not be reached by some input array a is removed. This cleaning up implies that the tree has exactly $n!$ leaves. It has at least $n!$ leaves because, otherwise, it could not sort correctly. It has at most $n!$ leaves since each of the possible $n!$ permutation of 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 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 for the algorithm. 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 a to be a random permutation of $\{1,\ldots,n\}$, then this is equivalent to selecting a random leaf, w, from the $n!$ leaves of $\mathcal{T}(\hat{b})$.

Exercise 6.12 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 $\log(n!)$. 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 6.6. *For any integer $n \geq 1$ and any (deterministic or randomized) comparison-based sorting algorithm $\mathcal{A}$, the expected number of comparisons done by $\mathcal{A}$ when sorting a random permutation of $\{1,\ldots,n\}$ is at least $\log(n!) = n\log n - O(n)$.*

6.2 Counting Sort and Radix Sort

In this section we study two sorting algorithms that are not comparison-based. Specialized for sorting small integers, these algorithms elude the lower-bounds of Theorem 6.5 by using (parts of) the elements in a as indices into an array. Consider a statement of the form

$$c[a[i]] = 1 \quad .$$

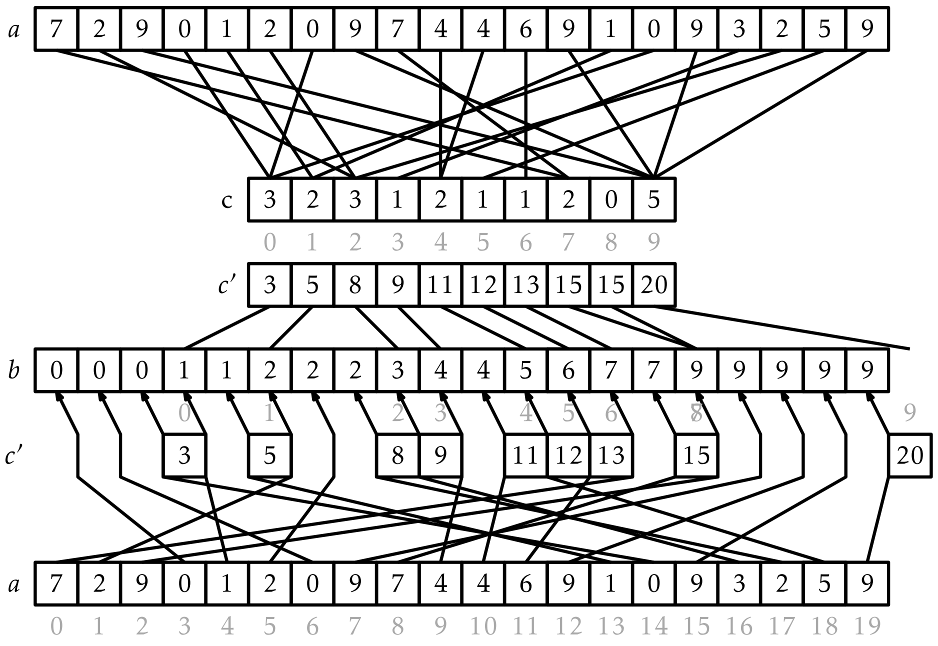
This statement executes in constant time, but has $\mathrm{length}(c)$ possible different outcomes, depending on the value of $a[i]$. This means that the execution of an algorithm that makes such a statement cannot be modelled as a binary tree. Ultimately, this is the reason that the algorithms in this section are able to sort faster than comparison-based algorithms.

6.2.1 Counting Sort

Suppose we have an input array a consisting of n integers, each in the range $0,\ldots,k-1$. 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 $i \in \{0,\ldots,k-1\}$, 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 6.7:

```
$\mathrm{counting\_sort}(a,k)$
    $c \leftarrow \mathrm{new\_zero\_array}(k)$
    for $i$ in $0,1,2,\ldots,\mathrm{length}(a)-1$ do
        $c[a[i]] \leftarrow c[a[i]]+1$
    for $i$ in $1,2,3,\ldots,k-1$ do
```



a 7 2 9 0 1 2 0 9 7 4 4 6 9 1 0 9 3 2 5 9
c 3 2 3 1 2 1 1 2 0 5
0 1 2 3 4 5 6 7 8 9
c′ 3 5 8 9 11 12 13 15 15 20
b 0 0 0 1 1 2 2 2 3 4 4 5 6 7 7 9 9 9 9 9
c′ 3 5 8 9 11 12 13 15 20
0 1 2 3 4 5 6 8 9
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Figure 6.7: The operation of counting sort on an array of length $n = 20$ that stores integers $0,\ldots,k-1 = 9$.

```
        $c[i] \leftarrow c[i] + c[i-1]$
    $b \leftarrow$ new_array(length($a$))
    **for** $i$ **in** length($a$) − 1, length($a$) − 2, length($a$) − 3, ..., 0 **do**
        $c[a[i]] \leftarrow c[a[i]] - 1$
        $b[c[a[i]]] \leftarrow a[i]$
    **return** $b$
```

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 $O(n)$ time with a single for loop. At this point, we could use c to fill in the 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 $O(k)$ 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 $i \in \{0,\ldots,k-1\}$, the output array, b, will have

$$b[c[i-1]] = b[c[i-1]+1] = \cdots = b[c[i]-1] = i \ .$$

Finally, the algorithm scans a backwards to place its elements, in order, into an output array b. When scanning, the element $a[i] \leftarrow j$ is placed at location $b[c[j]-1]$ and the value $c[j]$ is decremented.

Theorem 6.7. *The* $\mathrm{counting\_sort}(a,k)$ *method can sort an array* a *containing* n *integers in the set* $\{0,\ldots,k-1\}$ *in* $O(n+k)$ *time.*

The counting-sort algorithm has the nice property of being *stable*; it preserves the relative order of equal elements. If two elements $a[i]$ and $a[j]$ have the same value, and $i<j$ then $a[i]$ will appear before $a[j]$ in b. This will be useful in the next section.

6.2.2 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.

```
$\mathrm{radix\_sort}(a)$
    for $p$ in $0,1,2,\ldots,w \operatorname{div} d-1$ do
        $c \leftarrow new\_zero\_array 2^d$
        $b \leftarrow \mathrm{new\_array}(\mathrm{length}(a))$
        for $i$ in $0,1,2,\ldots,\mathrm{length}(a)-1$ do
            $bits \leftarrow (a[i] \gg d\cdot p) \wedge (2^d-1)$
            $c[bits] \leftarrow c[bits]+1$
        for $i$ in $1,2,3,\ldots,2^d-1$ do
            $c[i] \leftarrow c[i]+c[i-1]$
        for $i$ in $\mathrm{length}(a)-1,\mathrm{length}(a)-2,\mathrm{length}(a)-3,\ldots,0$ do
            $bits \leftarrow (a[i] \gg d\cdot p) \wedge (2^d-1)$
            $c[bits] \leftarrow c[bits]-1$
            $b[c[bits]] \leftarrow a[i]$
        $a \leftarrow b$
    return $b$
```

(In this code, the expression $(a[i] \gg d\cdot p) \wedge (2^d-1)$ extracts the integer whose binary representation is given by bits $(p+1)d-1,\ldots,pd$ of $a[i]$.) An example of the steps of this algorithm is shown in Figure 6.8.

[2]We assume that d divides w, otherwise we can always increase w to $d\lceil w/d\rceil$.

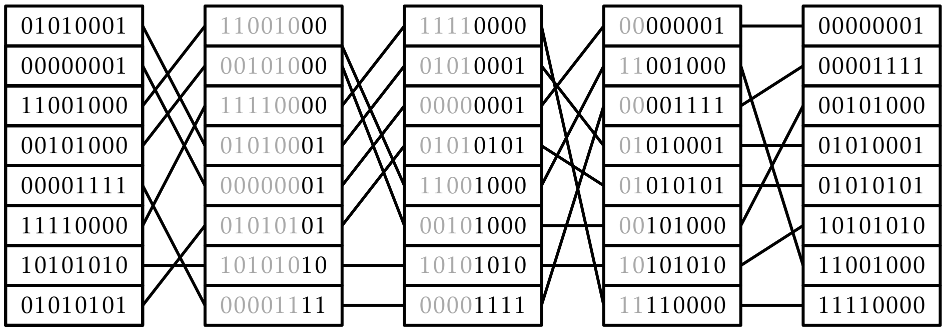


Figure 6.8: Using radixsort to sort $w = 8$-bit integers by using 4 passes of counting sort on $d = 2$-bit integers.

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 $\lfloor r/d \rfloor$ 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 $O(n + 2^d)$ time. Therefore, the performance of radix-sort is given by the following theorem.

Theorem 6.8. *For any integer $d > 0$, the $\mathrm{radix\_sort}(a,k)$ method can sort an array a containing n w-bit integers in $O((w/d)(n + 2^d))$ time.*

If we think, instead, of the elements of the array being in the range $\{0,\ldots,n^c - 1\}$, and take $d = \lceil \log n \rceil$ we obtain the following version of Theorem 6.8.

Corollary 6.1. *The $\mathrm{radix\_sort}(a,k)$ method can sort an array a containing n integer values in the range $\{0,\ldots,n^c - 1\}$ in $O(cn)$ time.*

6.3 Discussion and Exercises

Sorting is *the* fundamental algorithmic problem in computer science, and it has a long history. Knuth [27] attributes the merge-sort algorithm to von Neumann (1945). Quicksort is due to Hoare [22]. The original heap-sort algorithm is due to Williams [41], but the version presented here (in which the heap is constructed bottom-up in $O(n)$ time) is due to Floyd [15]. 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	$n \log n$	worst-case	No
Quicksort	$1.38 n \log n + O(n)$	expected	Yes
Heap-sort	$2n \log n + O(n)$	worst-case	Yes

Each of these comparison-based algorithms has its 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 (see Exercise 6.2).

The counting-sort and radix-sort algorithms described here are due to Seward [35, Section 2.4.6]. However, variants of radix-sort have been used since the 1920s 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.

Finally, we note that counting sort and radix-sort can be used to sort other types of numbers besides non-negative integers. Straightforward modifications of counting sort can sort integers, in any interval $\{a,\ldots,b\}$, in $O(n+b-a)$ time. Similarly, radix sort can sort integers in the same interval in $O(n(\log_n(b-a))$ time. Finally, both of these algorithms can also be used to sort floating point numbers in the IEEE 754 floating point format. This is because the IEEE format is designed to allow the comparison of two floating point numbers by comparing their values as if they were integers in a signed-magnitude binary representation [1].

Exercise 6.1. Illustrate the execution of merge-sort and heap-sort on an input array containing $1,7,4,6,2,8,3,5$. Give a sample illustration of one possible execution of quicksort on the same array.

Exercise 6.2. Implement a version of the merge-sort algorithm that sorts a DLList without using an auxiliary array. (See Exercise 3.13.)

Exercise 6.3. Some implementations of $\mathrm{quick\_sort}(a,i,n,c)$ always use $a[i]$ as a pivot. Give an example of an input array of length n in which such an implementation would perform $\binom{n}{2}$ comparisons.

Exercise 6.4. Some implementations of $\mathrm{quick\_sort}(a,i,n,c)$ always use $a[i+n/2]$ as a pivot. Given an example of an input array of length n in which such an implementation would perform $\binom{n}{2}$ comparisons.

Exercise 6.5. Show that, for any implementation of $\mathrm{quick\_sort}(a,i,n,c)$ that chooses a pivot deterministically, without first looking at any values in $a[i],\ldots,a[i+n-1]$, there exists an input array of length n that causes this implementation to perform $\binom{n}{2}$ comparisons.

Exercise 6.6. Design a Comparator, c, that you could pass as an argument to $\mathrm{quick\_sort}(a,i,n,c$ and that would cause quicksort to perform $\binom{n}{2}$ comparisons. (Hint: Your comparator does not actually need to look at the values being compared.)

Exercise 6.7. Analyze the expected number of comparisons done by Quicksort a little more carefully than the proof of Theorem 6.3. In particular, show that the expected number of comparisons is $2nH_n - n + H_n$.

Exercise 6.8. Describe an input array that causes heap sort to perform at least $2n\log n - O(n)$ comparisons. Justify your answer.

Exercise 6.9. Find another pair of permutations of $1,2,3$ that are not correctly sorted by the comparison tree in Figure 6.6.

Exercise 6.10. Prove that $\log n! = n\log n - O(n)$.

Exercise 6.11. Prove that a binary tree with k leaves has height at least $\log k$.

Exercise 6.12. 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 $\log k$.

Exercise 6.13. The implementation of $\mathrm{radix\_sort}(a,k)$ given here works when the input array, a contains only integers.

Chapter 7

Hash Tables

Hash tables are an efficient method of storing a small number, n, of integers from a large range $U = \{0,\ldots,2^w-1\}$. The term *hash table* includes a broad range of data structures. The first part of this chapter focuses on two of the most common implementations of hash tables: hashing with chaining and linear probing.

Very often hash tables store types of data that are not integers. In this case, an integer *hash code* is associated with each data item and 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.

7.1 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 (see Figure 7.1):

```
initialize()
    $d \leftarrow 1$
    $t \leftarrow \mathrm{alloc\_table}(2^d)$
    $z \leftarrow \mathrm{random\_odd\_int}()$
    $n \leftarrow 0$
```

The *hash value* of a data item x, denoted $\mathrm{hash}(x)$ is a value in the range $\{0,\ldots,\mathrm{length}(t)-1\}$. 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

$$n \le \mathrm{length}(t)$$

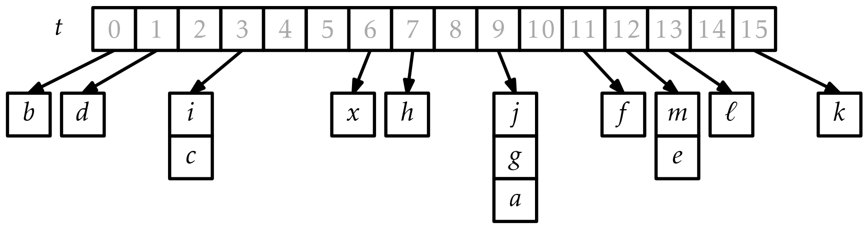


Figure 7.1: An example of a ChainedHashTable with $n = 14$ and $\mathrm{length}(t) = 16$. In this example $\mathrm{hash}(x) = 6$

so that the average number of elements stored in one of these lists is $n/\mathrm{length}(t) \le 1$.

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 $\{0,\ldots,\mathrm{length}(t)-1\}$, and we append x to the list $t[i]$:

```
add($x$)
    if find($x$) $\ne$ $nil$ then return $false$
    if $n+1 > \mathrm{length}(t)$ then resize()
    $t[\mathrm{hash}(x)]$.append($x$)
    $n \leftarrow n+1$
    return $true$
```

Growing the table, if necessary, involves doubling the length of t and reinserting all elements into the new table. This strategy is exactly the same as the one 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 27).

Besides growing, the only other work done when adding a new value x to a ChainedHashTable involves appending x to the list $t[\mathrm{hash}(x)]$. 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 $t[\mathrm{hash}(x)]$ until we find x so that we can remove it:

```
remove($x$)
    $\ell \leftarrow t[\mathrm{hash}(x)]$
    for $y$ in $\ell$ do
        if $y = x$ then
            $\ell$.remove_value($y$)
            $n \leftarrow n-1$
            if $3 \cdot n < \mathrm{length}(t)$ then resize()
            return $y$
    return $nil$
```

This takes $O(n_{\text{hash}(x)})$ time, where n_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 $t[\text{hash}(x)]$:

```
find($x$)
    for $y$ in $t[\text{hash}(x)]$ do
        if $y = x$ then
            return $y$
    return $nil$
```

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

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 $\text{length}(t)$ lists, so that the expected size of the list $t[\text{hash}(x)]$ is $O(n/\text{length}(t)) = O(1)$. 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 $t[\text{hash}(x)]$ will be n. In the next section we describe a good hash function.

7.1.1 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^d for some integer d (called the *dimension*). The formula for hashing an integer $x \in \{0, \ldots, 2^w - 1\}$ is

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

Here, z is a randomly chosen *odd* integer in $\{1, \ldots, 2^w - 1\}$. This hash function can be realized very efficiently by observing that, by default, operations on integers are already done modulo 2^w where w is the number of bits in an integer.[1] (See Figure 7.2.) Furthermore, integer division by 2^{w-d} is equivalent to dropping the rightmost $w - d$ bits in a binary representation (which is implemented by shifting the bits right by $w - d$ using the $\gg$ operator).

[1]This is true for most programming languages including C, C#, C++, and Java. Notable exceptions are Python and Ruby, in which the result of a fixed-length w-bit integer operation that overflows is upgraded to a variable-length representation.

2^w (4294967296)	100000000000000000000000000000000
z (4102541685)	11110100100001111101000101110101
x (42)	00000000000000000000000000101010
$z \cdot x$	10100000011110010010000101110100110010
$(z \cdot x) \bmod 2^w$	00011110010010000101110100110010
$((z \cdot x) \bmod 2^w) \operatorname{div} 2^{w-d}$	00011110

Figure 7.2: The operation of the multiplicative hash function with $w = 32$ and $d = 8$.

```
hash(x)
    return ((z · hash(x)) mod 2^w) ≫ (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 7.1. *Let x and y be any two values in $\{0,\dots,2^w-1\}$ with $x \neq y$. Then $\Pr\{\mathrm{hash}(x) = \mathrm{hash}(y)\} \leq 2/2^d$.*

With Lemma 7.1, the performance of $\mathrm{remove}(x)$, and $\mathrm{find}(x)$ are easy to analyze:

Lemma 7.2. *For any data value x, the expected length of the list $t[\mathrm{hash}(x)]$ is at most $n_x + 2$, where n_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 $y \in S$, define the indicator variable

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

and notice that, by Lemma 7.1, $\mathrm{E}[I_y] \leq 2/2^d = 2/\mathrm{length}(t)$. The expected length of the list $t[\mathrm{hash}(x)]$ is given by

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

as required. □

Now, we want to prove Lemma 7.1, 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. In other words, $(b_r,\ldots,b_0)_2$ is the integer whose binary representation is given by $b_r,\ldots,b_0$. We use $\star$ to denote a bit of unknown value.

Lemma 7.3. *Let S be the set of odd integers in $\{1,\ldots,2^w-1\}$; let q and i be any two elements in S. Then there is exactly one value $z \in S$ such that $zq \bmod 2^w = i$.*

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

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

$$zq \bmod 2^w = z'q \bmod 2^w = i$$

So

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

But this means that

$$(z-z')q = k2^w \tag{7.1}$$

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

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

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

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

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

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

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

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

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

Therefore $(z-z')q$ cannot satisfy (7.1), yielding a contradiction and completing the proof. □

The utility of Lemma 7.3 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 7.1. First we note that the condition $\mathrm{hash}(x) = \mathrm{hash}(y)$ is equivalent to the statement "the highest-order d bits of $zx \bmod 2^w$ and the highest-order d bits of $zy \bmod 2^w$ are the same." A necessary condition of that statement is that the highest-order d bits in the binary representation of $z(x-y) \bmod 2^w$ are either all 0's or all 1's. That is,

$$z(x-y) \bmod 2^w = (\underbrace{0,\ldots,0}_{d},\underbrace{\star,\ldots,\star}_{w-d})_2 \tag{7.2}$$

when $zx \bmod 2^w > zy \bmod 2^w$ or

$$z(x-y) \bmod 2^w = (\underbrace{1,\ldots,1}_{d},\underbrace{\star,\ldots,\star}_{w-d})_2 \ . \tag{7.3}$$

when $zx \bmod 2^w < zy \bmod 2^w$. Therefore, we only have to bound the probability that $z(x-y) \bmod 2^w$ looks like (7.2) or (7.3).

Let q be the unique odd integer such that $(x-y) \bmod 2^w = q2^r$ for some integer $r \geq 0$. By Lemma 7.3, the binary representation of $zq \bmod 2^w$ 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 $z(x-y) \bmod 2^w = zq2^r \bmod 2^w$ 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 $z(x-y) \bmod 2^w$ contain both 0's and 1's, so the probability that $z(x-y) \bmod 2^w$ looks like (7.2) or (7.3) is 0. If $r = w-d$, then the probability of looking like (7.2) is 0, but the probability of looking like (7.3) 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. □

7.1.2 Summary

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

Theorem 7.1. *A ChainedHashTable implements the USet interface. Ignoring the cost of calls to* $\mathrm{grow}()$*, a ChainedHashTable supports the operations* $\mathrm{add}(x)$*,* $\mathrm{remove}(x)$*, and* $\mathrm{find}(x)$ *in* $O(1)$ *expected time per operation.*

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

7.2 LinearHashTable: Linear Probing

The ChainedHashTable data structure uses an array of lists, where the ith list stores all elements x such that $\mathrm{hash}(x) = i$. 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 approach is 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 $i \leftarrow \mathrm{hash}(x)$ in the table location $t[i]$. If we cannot do this (because some element is already stored there) then we try to store it at location $t[(i+1) \bmod \mathrm{length}(t)]$; if that's not possible, then we try $t[(i+2) \bmod \mathrm{length}(t)]$, 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. nil 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 nil values in t. The operations on a LinearHashTable therefore maintain the invariant that $\mathrm{length}(t) \geq 2q$.

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-nil 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 $\mathrm{length}(t) = 2^d$.

```
initialize()
    $del \leftarrow \mathrm{object}()$

initialize()
    $d \leftarrow 1$
    $t \leftarrow \mathrm{new\_array}(2^d)$
    $q \leftarrow 0$
    $n \leftarrow 0$
```

The $\mathrm{find}(x)$ operation in a LinearHashTable is simple. We start at array entry $t[i]$ where $i = \mathrm{hash}(x)$ and search entries $t[i]$, $t[(i+1) \bmod \mathrm{length}(t)]$, $t[(i+2) \bmod$

$\mathrm{length}(t)]$, and so on, until we find an index i' such that, either, $t[i'] \leftarrow x$, or $t[i'] \leftarrow nil$. 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 nil.

```
find($x$)
    $i \leftarrow \mathrm{hash}(x)$
    while $t[i] \neq nil$ do
        if $t[i] \neq del$ and $x = t[i]$ then
            return $t[i]$
        $i \leftarrow (i+1) \bmod \mathrm{length}(t)$
```

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

```
add($x$)
    if $\mathrm{find}(x) \neq nil$ then return $false$
    if $2 \cdot (q+1) > \mathrm{length}(t)$ then resize()
    $i \leftarrow \mathrm{hash}(x)$
    while $t[i] \neq nil$ and $t[i] \neq del$ do
        $i \leftarrow (i+1) \bmod \mathrm{length}(t)$
    if $t[i] = nil$ then $q \leftarrow q+1$
    $n \leftarrow n+1$
    $t[i] \leftarrow x$
    return $true$
```

By now, the implementation of the $\mathrm{remove}(x)$ operation should be obvious. We search $t[i]$, $t[(i+1) \bmod \mathrm{length}(t)]$, $t[(i+2) \bmod \mathrm{length}(t)]$, and so on until we find an index i' such that $t[i'] \leftarrow x$ or $t[i'] \leftarrow nil$. In the former case, we set $t[i'] \leftarrow del$ and return $true$. In the latter case we conclude that x was not stored in the table (and therefore cannot be deleted) and return $false$.

```
remove($x$)
    $i \leftarrow \mathrm{hash}(x)$
    while $t[i] \neq nil$ do
        $y \leftarrow t[i]$
        if $y \neq del$ and $x = y$ then
            $t[i] \leftarrow del$
            $n \leftarrow n - 1$
            if $8 \cdot n < \mathrm{length}(t)$ then resize()
            return $y$
        $i \leftarrow (i+1) \bmod \mathrm{length}(t)$
    return $nil$
```

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-*nil* entry to *nil*. Therefore, when we reach an index i' such that $t[i'] \leftarrow nil$, this is a proof that the element, x, that we are searching for is not stored in the table; $t[i']$ has always been *nil*, 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-*nil* entries exceeds $\mathrm{length}(t)/2$ or by remove(x) when the number of data entries is less than $\mathrm{length}(t)/8$. 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 contains no *del* values.

```
resize()
    $d \leftarrow 1$
    while $(2^d < 3 \cdot n)$ do $d \leftarrow d + 1$
    $told \leftarrow t$
    $t \leftarrow \mathrm{new\_array}(2^d)$
    $q \leftarrow n$
    for $x$ in $told$ do
        if $x \neq nil$ and $x \neq del$ then
            $i \leftarrow \mathrm{hash}(x)$
            while $t[i] \neq nil$ do
                $i \leftarrow (i+1) \bmod \mathrm{length}(t)$
            $t[i] \leftarrow x$
```

7.2.1 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 *nil* entry in t. The intuition behind the analysis of linear probing is that, since at least half the elements in t are equal to *nil*, an operation should not take long to complete because it will very quickly come across a *nil* 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 $\{0,\ldots,\mathrm{length}(t)-1\}$. 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 $\mathrm{length}(t)$, so that $t[i]$ is really a shorthand for $t[i \bmod \mathrm{length}(t)]$.

We say that a *run of length k that starts at i* occurs when all the table entries $t[i], t[i+1],\ldots,t[i+k-1]$ are non-*nil* and $t[i-1]=t[i+k]=nil$. The number of non-*nil* elements of t is exactly q and the $\mathrm{add}(x)$ method ensures that, at all times, $q \le \mathrm{length}(t)/2$. There are q elements $x_1,\ldots,x_q$ that have been inserted into t since the last $\mathrm{resize}()$ operation. By our assumption, each of these has a hash value, $hash(x_j)$, that is uniform and independent of the rest. With this setup, we can prove the main lemma required to analyze linear probing.

Lemma 7.4. *Fix a value $i \in \{0,\ldots,\mathrm{length}(t)-1\}$. Then 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 x_j such that $hash(x_j) \in \{i,\ldots,i+k-1\}$. The probability that this occurs is exactly

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

since, for each choice of k elements, these k elements must hash to one of the k locations and the remaining $q-k$ elements must hash to the other $\mathrm{length}(t)-k$ table locations.[2]

In the following derivation we will cheat a little and replace $r!$ with $(r/e)^r$. Stirling's Approximation (Section 1.3.2) shows that this is only a factor of $O(\sqrt{r})$ from the truth. This is just done to make the derivation simpler; Exercise 7.4 asks the reader to redo the calculation more rigorously using Stirling's Approximation in its entirety.

The value of p_k is maximized when $\mathrm{length}(t)$ is minimum, and the data structure maintains the invariant that $\mathrm{length}(t) \ge 2q$, so

$$\begin{aligned} p_k &\le \binom{q}{k}\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \\ &= \left(\frac{q!}{(q-k)!k!}\right)\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \end{aligned}$$

[2]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 $t[i-1]=t[i+k]=nil$.

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

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

Using Lemma 7.4 to prove upper-bounds on the expected running time of $\mathrm{find}(x)$, $\mathrm{add}(x)$, and $\mathrm{remove}(x)$ is now fairly straightforward. Consider the simplest case, where we execute $\mathrm{find}(x)$ for some value x that has never been stored in the LinearHashTable. In this case, $i = \mathrm{hash}(x)$ is a random value in $\{0,\ldots,\mathrm{length}(t)-1\}$ independent of the contents of t. If i is part of a run of length k, then the time it takes to execute the $\mathrm{find}(x)$ operation is at most $O(1+k)$. Thus, the expected running time can be upper-bounded by

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

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

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

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.[3] Therefore, we conclude that the expected running time of the $\mathrm{find}(x)$ operation for a value x that is not contained in a LinearHashTable is $O(1)$.

If we ignore the cost of the $\mathrm{resize}()$ operation, then the above analysis gives us all we need to analyze the cost of operations on a LinearHashTable.

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

In summary, if we ignore the cost of calls to $\mathrm{resize}()$, 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.

7.2.2 Summary

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

Theorem 7.2. *A LinearHashTable implements the USet interface. Ignoring the cost of calls to* $\mathrm{resize}()$*, a LinearHashTable supports the operations* $\mathrm{add}(x)$*,* $\mathrm{remove}(x)$*, and* $\mathrm{find}(x)$ *in* $O(1)$ *expected time per operation.*

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

7.2.3 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 $hash(\mathrm{x}_1), \ldots, hash(x_n)$ are independently and uniformly distributed over the set $\{0, \ldots, \mathrm{length}(t) - 1\}$. One way to achieve this is to store a giant array, 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 $\mathrm{hash}(x)$ by extracting a d-bit integer from $tab[x.\mathrm{hash\_code}()]$:

```
ideal_hash($x$)
    return $tab[x.\mathrm{hash\_code}() \gg w - d]$
```

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

Here, $\gg$, is the *bitwise right shift* operator, so $x.\mathrm{hash\_code}() \gg w - d$ extracts the d most significant bits of x's w-bit hash code.

Unfortunately, storing an array of size 2^w is prohibitive in terms of memory usage. The approach used by *tabulation hashing* is to, instead, treat w-bit integers as being comprised of w/r integers, each having only r bits. In this way, tabulation hashing only needs w/r arrays each of length 2^r. All the entries in these arrays are independent random w-bit integers. To obtain the value of $\mathrm{hash}(x)$ we split $x.\mathrm{hash\_code}()$ 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 $\mathrm{hash}(x)$. The following code shows how this works when $w = 32$ and $r = 4$:

```
$\mathrm{hash}(x)$
  $h \leftarrow \mathrm{hash\_code}(x)$
  **return** $(tab[0][h \wedge \mathrm{ff}_{16}]$
        $\oplus\, tab[1][(h \gg 8) \wedge \mathrm{ff}_{16}]$
        $\oplus\, tab[2][(h \gg 16) \wedge \mathrm{ff}_{16}]$
        $\oplus\, tab[3][(h \gg 24) \wedge \mathrm{ff}_{16}]) \gg (w - d)$
```

In this case, tab is a two-dimensional array with four columns and $2^{32/4} = 256$ rows. Quantities like ff_{16}, used above, are *hexadecimal numbers* whose digits have 16 possible values 0–9, which have their usual meaning and a–f, which denote 10–15. The number $\mathrm{ff}_{16} = 15 \cdot 16 + 15 = 255$. The $\wedge$ symbol is the *bitwise and* operator, so code like $h \gg 8 \wedge \mathrm{ff}_{16}$ extracts bits with index 8 through 15 of h.

One can easily verify that, for any x, $\mathrm{hash}(x)$ is uniformly distributed over $\{0, \dots, 2^d - 1\}$. 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 7.2 still holds. References for this are provided at the end of this chapter.

7.3 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 $x.\mathrm{hash\_code}()$ and $y.\mathrm{hash\_code}()$ are equal.

2. If x and y are not equal, then the probability that $x.\mathrm{hash\_code}() = y.\mathrm{hash\_code}()$ 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.

7.3.1 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. In these cases, we just treat these bits as the representation of an integer in the range $\{0,\dots,2^w-1\}$. 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.

7.3.2 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 7.7–7.9). However, if one is willing to do arithmetic with $2w$ bits of precision, then there are simple and robust methods available. Suppose we have an object made up of several parts $P_0,\dots,P_{r-1}$ whose hash codes are $x_0,\dots,x_{r-1}$. Then we can choose mutually independent random w-bit integers $z_0,\dots,z_{r-1}$ and a random $2w$-bit odd integer z and compute a hash code for our object with

$$h(x_0,\dots,x_{r-1}) = \left(\left(z\sum_{i=0}^{r-1} z_i x_i\right) \bmod 2^{2w}\right) \operatorname{div} 2^w \enspace .$$

Note that this hash code has a final step (multiplying by z and dividing by 2^w) that uses the multiplicative hash function from Section 7.1.1 to take the $2w$-bit intermediate result and reduce it to a w-bit final result. Here is an example of this method applied to a simple compound object with three parts x_0, x_1, and x_2:

```
hash_code()
    $z \leftarrow [2058cc50_{16}, cb19137e_{16}, 2cb6b6fd_{16}]$
    $zz \leftarrow bea0107e5067d19d_{16}$
    $h \leftarrow [x_0.\text{hash\_code}(), x_1.\text{hash\_code}(), x_2.\text{hash\_code}()]$
    return $(((z[0]\cdot h[0] + z[1]\cdot h[1] + z[2]\cdot h[2])\cdot zz) \bmod 2^2\cdot w)) \gg w$
```

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

Theorem 7.3. *Let $x_0,\ldots,x_{r-1}$ and $y_0,\ldots,y_{r-1}$ each be sequences of w bit integers in $\{0,\ldots,2^w-1\}$ and assume $x_i \neq y_i$ for at least one index $i \in \{0,\ldots,r-1\}$. Then*

$$\Pr\{h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})\} \le 3/2^w \ .$$

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

$$h'(x_0,\ldots,x_{r-1}) = \left(\sum_{j=0}^{r-1} z_j x_j\right) \bmod 2^{2w} \ .$$

Suppose that $h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1})$. We can rewrite this as:

$$z_i(x_i - y_i) \bmod 2^{2w} = t \tag{7.4}$$

where

$$t = \left(\sum_{j=0}^{i-1} z_j(y_j - x_j) + \sum_{j=i+1}^{r-1} z_j(y_j - x_j)\right) \bmod 2^{2w}$$

If we assume, without loss of generality that $x_i > y_i$, then (7.4) becomes

$$z_i(x_i - y_i) = t \ , \tag{7.5}$$

since each of z_i and $(x_i - y_i)$ is at most $2^w - 1$, so their product is at most $2^{2w} - 2^{w+1} + 1 < 2^{2w} - 1$. By assumption, $x_i - y_i \neq 0$, so (7.5) has at most one solution in z_i. Therefore, since z_i and t are independent ($z_0,\ldots,z_{r-1}$ are mutually independent), the probability that we select z_i so that $h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1})$ is at most $1/2^w$.

The final step of the hash function is to apply multiplicative hashing to reduce our $2w$-bit intermediate result $h'(x_0,\ldots,x_{r-1})$ to a w-bit final result $h(x_0,\ldots,x_{r-1})$. By Theorem 7.3, if $h'(x_0,\ldots,x_{r-1}) \neq h'(y_0,\ldots,y_{r-1})$, then $\Pr\{h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})\} \le 2/2^w$.

To summarize,

$$
\begin{aligned}
&\Pr\left\{\begin{array}{l} h(x_0,\ldots,x_{r-1}) \\ \quad = h(y_0,\ldots,y_{r-1}) \end{array}\right\} \\
&= \Pr\left\{\begin{array}{l} h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1}) \text{ or} \\ h'(x_0,\ldots,x_{r-1}) \neq h'(y_0,\ldots,y_{r-1}) \\ \quad \text{and } zh'(x_0,\ldots,x_{r-1}) \operatorname{div} 2^w = zh'(y_0,\ldots,y_{r-1}) \operatorname{div} 2^w \end{array}\right\} \\
&\leq 1/2^w + 2/2^w = 3/2^w \ .
\end{aligned}
$$

□

7.3.3 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 w-bit integer z_i for each component. We could use a pseudorandom sequence to generate as many z_i's as we need, but then the 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 z_i in the proof of Theorem 7.3 are no longer independent.

A more rigorous approach is to base our hash codes on polynomials over prime fields; these are just regular polynomials that are evaluated modulo some prime number, p. This method is based on the following theorem, which says that polynomials over prime fields behave pretty-much like usual polynomials:

Theorem 7.4. *Let p be a prime number, and let $f(z) = x_0z^0 + x_1z^1 + \cdots + x_{r-1}z^{r-1}$ be a non-trivial polynomial with coefficients $x_i \in \{0,\ldots,p-1\}$. Then the equation $f(z) \bmod p = 0$ has at most $r-1$ solutions for $z \in \{0,\ldots,p-1\}$.*

To use Theorem 7.4, we hash a sequence of integers $x_0,\ldots,x_{r-1}$ with each $x_i \in \{0,\ldots,p-2\}$ using a random integer $z \in \{0,\ldots,p-1\}$ via the formula

$$h(x_0,\ldots,x_{r-1}) = \left(x_0z^0 + \cdots + x_{r-1}z^{r-1} + (p-1)z^r\right) \bmod p \ .$$

Note the extra $(p-1)z^r$ term at the end of the formula. It helps to think of $(p-1)$ as the last element, x_r, in the sequence $x_0,\ldots,x_r$. Note that this element differs from every other element in the sequence (each of which is in the set $\{0,\ldots,p-2\}$). We can think of $p-1$ 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 z:

Theorem 7.5. *Let $p > 2^w + 1$ be a prime, let $x_0,\ldots,x_{r-1}$ and $y_0,\ldots,y_{r-1}$ each be sequences of w-bit integers in $\{0,\ldots,2^w-1\}$, and assume $x_i \neq y_i$ for at least one index*

$i \in \{0,\dots,r-1\}$. *Then*

$$\Pr\{h(x_0,\dots,x_{r-1}) = h(y_0,\dots,y_{r-1})\} \le (r-1)/p \enspace .$$

Proof. The equation $h(x_0,\dots,x_{r-1}) = h(y_0,\dots,y_{r-1})$ can be rewritten as

$$\left((x_0-y_0)z^0 + \cdots + (x_{r-1}-y_{r-1})z^{r-1}\right) \bmod p = 0. \tag{7.6}$$

Since $x_i \neq y_i$, this polynomial is non-trivial. Therefore, by Theorem 7.4, it has at most $r-1$ solutions in z. The probability that we pick z to be one of these solutions is therefore at most $(r-1)/p$. □

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

$$x_0,\dots,x_{r-1},p-1,0,0,\dots \enspace .$$

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, (7.6) becomes

$$\left(\sum_{i=0}^{i=r'-1}(x_i-y_i)z^i + (x_{r'}-p+1)z^{r'} + \sum_{i=r'+1}^{i=r-1} x_i z^i + (p-1)z^r\right) \bmod p = 0 \enspace ,$$

which, by Theorem 7.4, has at most r solutions in z. This combined with Theorem 7.5 suffice to prove the following more general theorem:

Theorem 7.6. *Let $p > 2^w+1$ be a prime, let $x_0,\dots,x_{r-1}$ and $y_0,\dots,y_{r'-1}$ be distinct sequences of w-bit integers in $\{0,\dots,2^w-1\}$. Then*

$$\Pr\{h(x_0,\dots,x_{r-1}) = h(y_0,\dots,y_{r-1})\} \le \max\{r,r'\}/p \enspace .$$

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

```
hash_code()
  $p \leftarrow 2^{32} - 5$   # this is a prime number
  $z \leftarrow \mathrm{64b6055a}_{16}$ # 32 bits from random.org
  $z_2 \leftarrow \mathrm{5067d19d}_{16}$ # random odd 32 bit number
  $s \leftarrow 0$
  $zi \leftarrow 1$
  for $i$ in $0,1,2,\dots,\mathrm{length}(x)-1$ do
    # reduce to 31 bits
    $xi \leftarrow ((x[i].\mathrm{hash\_code}() \cdot z_2) \bmod 2^{32}) \gg 1$
```

```
        $s \leftarrow (s + zi \cdot xi) \bmod p$
        $zi \leftarrow (zi \cdot z) \bmod p$
    $s \leftarrow (s + zi \cdot (p-1)) \bmod p$
    **return** $s \bmod 2^{32}$
```

The preceding code sacrifices some collision probability for implementation convenience. In particular, it applies the multiplicative hash function from Section 7.1.1, with $d = 31$ to reduce $x[i].\mathrm{hash\_code}()$ to a 31-bit value. This is so that the additions and multiplications that are done modulo the prime $p = 2^{32} - 5$ can be carried out using unsigned 63-bit arithmetic. Thus 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 7.6.

7.4 Discussion and Exercises

Hash tables and hash codes represent an enormous and active field of research that is just touched upon in this chapter. The online Bibliography on Hashing [2] contains nearly 2000 entries.

A variety of different hash table implementations exist. The one described in Section 7.1 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 7.2. 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; 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 $\mathrm{find}(x)$ 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* [18, 12] and *cuckoo hashing* [32].

The hash functions presented in this chapter are probably among the most practical methods currently known 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 [5]. Tabulation hashing, described in Section 7.2.3, is due to Carter and Wegman [5], but its analysis, when applied to linear probing (and several other hash table schemes) is due to Pătraşcu and Thorup [33].

The idea of *multiplicative hashing* is very old and seems to be part of the hashing folklore [27, Section 6.4]. However, the idea of choosing the multiplier z to be a random *odd* number, and the analysis in Section 7.1.1 is due to Dietzfelbinger *et al.* [11]. This version of multiplicative hashing is one of the simplest, but its collision probability of $2/2^d$ is a factor of two larger than what one could expect with a random function from $2^w \to 2^d$. The *multiply-add hashing* method uses the function

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

where z and b are each randomly chosen from $\{0,\ldots,2^{2w}-1\}$. Multiply-add hashing has a collision probability of only $1/2^d$ [9], but requires $2w$-bit precision arithmetic.

There are a number of methods of obtaining hash codes from fixed-length sequences of w-bit integers. One particularly fast method [3] is the function

$$\begin{aligned} &h(x_0,\ldots,x_{r-1}) \\ &\quad = \left(\textstyle\sum_{i=0}^{r/2-1}((x_{2i}+a_{2i}) \bmod 2^w)((x_{2i+1}+a_{2i+1}) \bmod 2^w)\right) \bmod 2^{2w} \end{aligned}$$

where r is even and $a_0,\ldots,a_{r-1}$ are randomly chosen from $\{0,\ldots,2^w\}$. This yields a $2w$-bit hash code that has collision probability $1/2^w$. This can be reduced to a w-bit hash code using multiplicative (or multiply-add) hashing. This method is fast because it requires only $r/2$ $2w$-bit multiplications whereas the method described in Section 7.3.2 requires r multiplications. (The $\bmod$ operations occur implicitly by using w and $2w$-bit arithmetic for the additions and multiplications, respectively.)

The method from Section 7.3.3 of using polynomials over prime fields to hash variable-length arrays and strings is due to Dietzfelbinger *et al.* [10]. Due to its use of the $\bmod$ operator which relies on a costly machine instruction, it is, unfortunately, not very fast. Some variants of this method choose the prime p to be one of the form $2^w - 1$, in which case the $\bmod$ operator can be replaced with addition ($+$) and bitwise-and ($\wedge$) operations [26, 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 7.1. A certain university assigns each of its students student numbers the first time they register for any course. These numbers are sequential integers that started at 0 many years ago and are now in the millions. Suppose we have a class of one hundred first year students and we want to assign them hash codes based on their student numbers. Does it make more sense to use the first two digits or the last two digits of their student number? Justify your answer.

Exercise 7.2. Consider the hashing scheme in Section 7.1.1, and suppose $n = 2^d$ and $d \leq w/2$.

1. Show that, for any choice of the muliplier, z, there exists n values that all have the same hash code. (Hint: This is easy, and doesn't require any number theory.)

2. Given the multiplier, z, describe n values that all have the same hash code. (Hint: This is harder, and requires some basic number theory.)

Exercise 7.3. Prove that the bound $2/2^d$ in Lemma 7.1 is the best possible bound by showing that, if $x = 2^{w-d-2}$ and $y = 3x$, then $\Pr\{\mathrm{hash}(x) = \mathrm{hash}(y)\} = 2/2^d$. (Hint look at the binary representations of zx and $z3x$ and use the fact that $z3x = zx + 2zx$.)

Exercise 7.4. Reprove Lemma 7.4 using the full version of Stirling's Approximation given in Section 1.3.2.

Exercise 7.5. Consider the following simplified version of the code for adding an element x to a LinearHashTable, which simply stores x in the first nil array entry it finds. Explain why this could be very slow by giving an example of a sequence of $O(n)$ $\mathrm{add}(x)$, $\mathrm{remove}(x)$, and $\mathrm{find}(x)$ operations that would take on the order of n^2 time to execute.

```
add_slow($x$)
    if $2 \cdot (q+1) > \mathrm{length}(t)$ then $\mathrm{resize}()$
    $i \leftarrow \mathrm{hash}(x)$
    while $t[i] \neq nil$ do
        if $t[1] \neq del$ and $x = t[i]$ then return $false$
        $i \leftarrow (i+1) \bmod \mathrm{len}(t[i])$
    $t[i] \leftarrow x$
    $n \leftarrow n+1$
    $q \leftarrow q+1$
    return $true$
```

Exercise 7.6. Early versions of the Java $\mathrm{hash\_code}()$ method for the String class worked by not using all of the characters found in long strings. For example, for a sixteen character string, the hash code was computed using only the eight even-indexed characters. Explain why this was a very bad idea by giving an example of large set of strings that all have the same hash code.

Exercise 7.7. Suppose you have an object made up of two w-bit integers, x and y. Show why $x \oplus y$ 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 7.8. Suppose you have an object made up of two w-bit integers, x and y. Show why $x + y$ 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 7.9. 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 $h(x,y)$ that produces a single w-bit integer. Prove that there exists a large set of objects that have the same hash code.

Exercise 7.10. Let $p = 2^w - 1$ for some positive integer w. Explain why, for a positive integer x

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

(This gives an algorithm for computing $x \bmod (2^w - 1)$ by repeatedly setting until $x \leq 2^w - 1$.)

Exercise 7.11. Find some commonly used hash table implementation such as the (or the HashTable or LinearHashTable implementations in this book, and design a program that stores integers in this data structure so that there are integers, x, such that $\mathrm{find}(x)$ takes linear time. That is, find a set of n integers for which there are cn elements that hash to the same table location.

Depending on how good the implementation is, you may be able to do this just by inspecting the code for the implementation, or you may have to write some code that does trial insertions and searches, timing how long it takes to add and find particular values. (This can be, and has been, used to launch denial of service attacks on web servers [7].)

Chapter 8

Graphs

In this chapter, we study two representations of graphs and basic algorithms that use these representations.

Mathematically, a *(directed) graph* is a pair $G = (V, E)$ where V is a set of *vertices* and E is a set of ordered pairs of vertices called *edges*. An edge (i, j) is *directed* from i to j; i is called the *source* of the edge and j is called the *target*. A *path* in G is a sequence of vertices $v_0, \ldots, v_k$ such that, for every $i \in \{1, \ldots, k\}$, the edge (v_{i-1}, v_i) is in E. A path $v_0, \ldots, v_k$ is a *cycle* if, additionally, the edge (v_k, v_0) is in E. A path (or cycle) is *simple* if all of its vertices are unique. If there is a path from some vertex v_i to some vertex v_j then we say that v_j is *reachable* from v_i. An example of a graph is shown in Figure 8.1.

Due to their ability to model so many phenomena, graphs have an enormous number of applications. There are many obvious examples. Computer networks can be modelled as graphs, with vertices corresponding to computers and edges corresponding to (directed) communication links between those computers. City streets can be modelled as graphs, with vertices representing intersections and edges representing streets joining consecutive intersections.

Less obvious examples occur as soon as we realize that graphs can model any pairwise relationships within a set. For example, in a university setting we might have a timetable *conflict graph* whose vertices represent courses offered in the university and in which the edge (i, j) is present if and only if there is at least one student that is taking both class i and class j. Thus, an edge indicates that the exam for class i should not be scheduled at the same time as the exam for class j.

Throughout this section, we will use n to denote the number of vertices of G and m to denote the number of edges of G. That is, $n = |V|$ and $m = |E|$. Furthermore, we will assume that $V = \{0, \ldots, n-1\}$. Any other data that we would like to associate with the elements of V can be stored in an array of length n.

Some typical operations performed on graphs are:

- $\mathrm{add\_edge}(i, j)$: Add the edge (i, j) to E.
- $\mathrm{remove\_edge}(i, j)$: Remove the edge (i, j) from E.

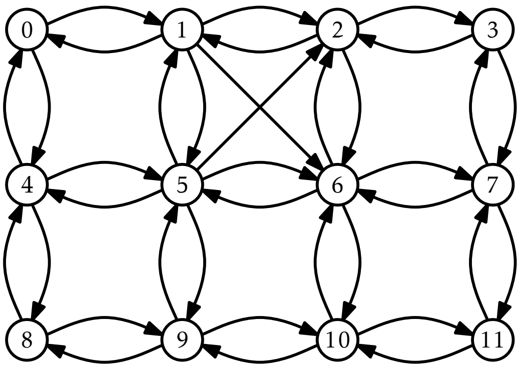


Figure 8.1: A graph with twelve vertices. Vertices are drawn as numbered circles and edges are drawn as pointed curves pointing from source to target.

- $\mathrm{has\_edge}(i,j)$: Check if the edge $(i,j) \in E$
- $\mathrm{out\_edges}(i)$: Return a List of all integers j such that $(i,j) \in E$
- $\mathrm{in\_edges}(i)$: Return a List of all integers j such that $(j,i) \in E$

Note that these operations are not terribly difficult to implement efficiently. For example, the first three operations can be implemented directly by using a USet, so they can be implemented in constant expected time using the hash tables discussed in Chapter 7. The last two operations can be implemented in constant time by storing, for each vertex, a list of its adjacent vertices.

However, different applications of graphs have different performance requirements for these operations and, ideally, we can use the simplest implementation that satisfies all the application's requirements. For this reason, we discuss two broad categories of graph representations.

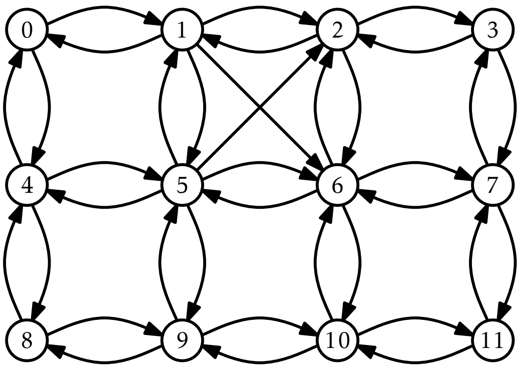
8.1 AdjacencyMatrix: Representing a Graph by a Matrix

An *adjacency matrix* is a way of representing an n vertex graph $G = (V, E)$ by an $n \times n$ matrix, a, whose entries are boolean values.

```
initialize()
    $a \leftarrow \mathrm{new\_boolean\_matrix}(n, n)$
```

The matrix entry $a[i][j]$ is defined as

$$a[i][j] = \begin{cases} \mathit{true} & \text{if } (i,j) \in E \\ \mathit{false} & \text{otherwise} \end{cases}$$



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	0	0	1	0	0	0	0	0	0	0
1	1	0	1	0	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	1	0	0	0	0
4	1	0	0	0	0	1	0	0	1	0	0	0
5	0	1	1	0	1	0	1	0	0	1	0	0
6	0	0	1	0	0	1	0	1	0	0	1	0
7	0	0	0	1	0	0	1	0	0	0	0	1
8	0	0	0	0	1	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0	1	0	1	0
10	0	0	0	0	0	0	1	0	0	1	0	1
11	0	0	0	0	0	0	0	1	0	0	1	0

Figure 8.2: A graph and its adjacency matrix.

The adjacency matrix for the graph in Figure 8.1 is shown in Figure 8.2.

In this representation, the operations add_edge(i,j), remove_edge(i,j), and has_edge(i,j) just involve setting or reading the matrix entry $a[i][j]$:

```
add_edge($i,j$)
    $a[i][j] \leftarrow true$

remove_edge($i,j$)
    $a[i][j] \leftarrow false$

has_edge($i,j$)
    return $a[i][j]$
```

These operations clearly take constant time per operation.

Where the adjacency matrix performs poorly is with the $\mathrm{out\_edges}(i)$ and $\mathrm{in\_edges}(i)$ operations. To implement these, we must scan all n entries in the corresponding row or column of a and gather up all the indices, j, where $a[i][j]$, respectively $a[j][i]$, is true. These operations clearly take $O(n)$ time per operation.

Another drawback of the adjacency matrix representation is that it is large. It stores an $n \times n$ boolean matrix, so it requires at least n^2 bits of memory. The implementation here uses a matrix of values so it actually uses on the order of n^2 bytes of memory. A more careful implementation, which packs w boolean values into each word of memory, could reduce this space usage to $O(n^2/w)$ words of memory.

Theorem 8.1. *The AdjacencyMatrix data structure implements the Graph interface. An AdjacencyMatrix supports the operations*

- $\mathrm{add\_edge}(i,j)$*,* $\mathrm{remove\_edge}(i,j)$*, and* $\mathrm{has\_edge}(i,j)$ *in constant time per operation; and*
- $\mathrm{in\_edges}(i)$*, and* $\mathrm{out\_edges}(i)$ *in* $O(n)$ *time per operation.*

The space used by an AdjacencyMatrix is $O(n^2)$*.*

Despite its high memory requirements and poor performance of the $\mathrm{in\_edges}(i)$ and $\mathrm{out\_edges}(i)$ operations, an AdjacencyMatrix can still be useful for some applications. In particular, when the graph G is *dense*, i.e., it has close to n^2 edges, then a memory usage of n^2 may be acceptable.

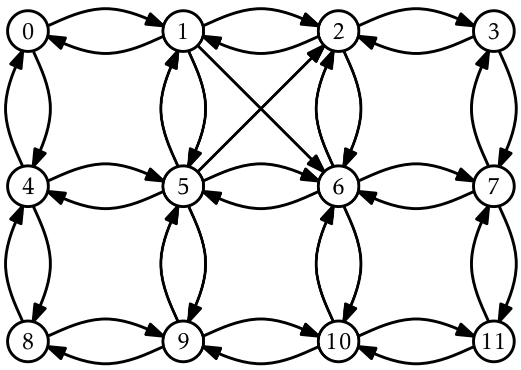
The AdjacencyMatrix data structure is also commonly used because algebraic operations on the matrix a can be used to efficiently compute properties of the graph G. This is a topic for a course on algorithms, but we point out one such property here: If we treat the entries of a as integers (1 for *true* and 0 for *false*) and multiply a by itself using matrix multiplication then we get the matrix a^2. Recall, from the definition of matrix multiplication, that

$$a \oplus 2[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot a[k][j] \ .$$

Interpreting this sum in terms of the graph G, this formula counts the number of vertices, k, such that G contains both edges (i,k) and (k,j). That is, it counts the number of paths from i to j (through intermediate vertices, k) whose length is exactly two. This observation is the foundation of an algorithm that computes the shortest paths between all pairs of vertices in G using only $O(\log n)$ matrix multiplications.

8.2 AdjacencyLists: A Graph as a Collection of Lists

Adjacency list representations of graphs take a more vertex-centric approach. There are many possible implementations of adjacency lists. In this section, we present a



0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	0	1	5	6	4	8	9	10
4	2	3	7	5	2	2	3	9	5	6	7
	6	6		8	6	7	11		10	11	
	5				9	10					
					4						

Figure 8.3: A graph and its adjacency lists

simple one. At the end of the section, we discuss different possibilities. In an adjacency list representation, the graph $G = (V,E)$ is represented as an array, adj, of lists. The list $adj[i]$ contains a list of all the vertices adjacent to vertex i. That is, it contains every index j such that $(i,j) \in E$.

```
initialize()
    $adj \leftarrow$ new_array($n$)
    **for** $i$ **in** $0, 1, 2, \ldots, n-1$ **do**
        $adj[i] \leftarrow$ ArrayStack()
```

(An example is shown in Figure 8.3.) In this particular implementation, we represent each list in adj as ArrayStack, because we would like constant time access by position. Other options are also possible. Specifically, we could have implemented adj as a DLList.

The add_edge(i,j) operation just appends the value j to the list $adj[i]$:

```
add_edge($i$,$j$)
    $adj[i]$.append($j$)
```

This takes constant time.

The $\mathrm{remove\_edge}(i,j)$ operation searches through the list $adj[i]$ until it finds j and then removes it:

```
remove_edge($i,j$)
    **for** $k$ **in** $0,1,2,\ldots,\mathrm{length}(adj[i])-1$ **do**
        **if** $adj[i].\mathrm{get}(k) = j$ **then**
            $adj[i].\mathrm{remove}(k)$
            **return**
```

This takes $O(\deg(i))$ time, where $\deg(i)$ (the *degree* of i) counts the number of edges in E that have i as their source.

The $\mathrm{has\_edge}(i,j)$ operation is similar; it searches through the list $adj[i]$ until it finds j (and returns true), or reaches the end of the list (and returns false):

```
has_edge($i,j$)
    **for** $k$ **in** $adj[i]$ **do**
        **if** $k = j$ **then**
            **return** $true$
    **return** $false$
```

This also takes $O(\deg(i))$ time.

The $\mathrm{out\_edges}(i)$ operation is very simple; it returns the list $adj[i]$:

```
out_edges($i$)
    **return** $adj[i]$
```

The $\mathrm{in\_edges}(i)$ operation is much more work. It scans over every vertex j checking if the edge (i,j) exists and, if so, adding j to the output list:

```
in_edges($i$)
    $out \leftarrow \mathrm{ArrayStack}()$
    **for** $j$ **in** $0,1,2,\ldots,n-1$ **do**
        **if** $\mathrm{has\_edge}(j,i)$ **then** $out.\mathrm{append}(j)$
    **return** $out$
```

This operation is very slow. It scans the adjacency list of every vertex, so it takes $O(n+m)$ time.

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

Theorem 8.2. *The AdjacencyLists data structure implements the Graph interface. An AdjacencyLists supports the operations*

- $\mathrm{add\_edge}(i,j)$ *in constant time per operation;*
- $\mathrm{remove\_edge}(i,j)$ *and* $\mathrm{has\_edge}(i,j)$ *in* $O(\deg(i))$ *time per operation;*
- $\mathrm{in\_edges}(i)$ *in* $O(n+m)$ *time per operation.*

The space used by a AdjacencyLists is $O(n+m)$.

As alluded to earlier, there are many different choices to be made when implementing a graph as an adjacency list. Some questions that come up include:

- What type of collection should be used to store each element of adj? One could use an array-based list, a linked-list, or even a hashtable.
- Should there be a second adjacency list, $inadj$, that stores, for each i, the list of vertices, j, such that $(j,i) \in E$? This can greatly reduce the running-time of the $\mathrm{in\_edges}(i)$ operation, but requires slightly more work when adding or removing edges.
- Should the entry for the edge (i,j) in $adj[i]$ be linked by a reference to the corresponding entry in $inadj[j]$?
- Should edges be first-class objects with their own associated data? In this way, adj would contain lists of edges rather than lists of vertices (integers).

Most of these questions come down to a tradeoff between complexity (and space) of implementation and performance features of the implementation.

8.3 Graph Traversal

In this section we present two algorithms for exploring a graph, starting at one of its vertices, i, and finding all vertices that are reachable from i. Both of these algorithms are best suited to graphs represented using an adjacency list representation. Therefore, when analyzing these algorithms we will assume that the underlying representation is an AdjacencyLists.

8.3.1 Breadth-First Search

The *breadth-first-search* algorithm starts at a vertex i and visits, first the neighbours of i, then the neighbours of the neighbours of i, then the neighbours of the neighbours of the neighbours of i, and so on.

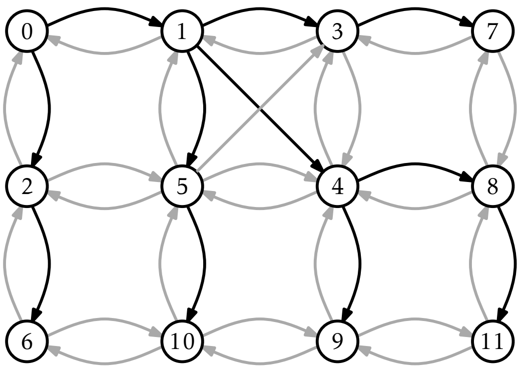


Figure 8.4: An example of breadth-first-search starting at node 0. Nodes are labelled with the order in which they are added to q. Edges that result in nodes being added to q are drawn in black, other edges are drawn in grey.

This algorithm is a generalization of the breadth-first traversal algorithm for binary trees (Section 4.1.2), and is very similar; it uses a queue, q, that initially contains only i. It then repeatedly extracts an element from q and adds its neighbours to q, provided that these neighbours have never been in q before. The only major difference between the breadth-first-search algorithm for graphs and the one for trees is that the algorithm for graphs has to ensure that it does not add the same vertex to q more than once. It does this by using an auxiliary boolean array, *seen*, that tracks which vertices have already been discovered.

```
bfs$(g,r)$
    $seen \leftarrow$ new_boolean_array$(n)$
    $q \leftarrow$ SLList()
    $q$.add$(r)$
    $seen[r] \leftarrow true$
    while $q$.size() $> 0$ do
        $i \leftarrow q$.remove()
        for $j$ in $g$.out_edges$(i)$ do
            if $seen[j] = false$ then
                $q$.add$(j)$
                $seen[j] \leftarrow true$
```

An example of running bfs$(g,0)$ on the graph from Figure 8.1 is shown in Figure 8.4. Different executions are possible, depending on the ordering of the adjacency lists; Figure 8.4 uses the adjacency lists in Figure 8.3.

Analyzing the running-time of the bfs(g,i) routine is fairly straightforward. The use of the *seen* array ensures that no vertex is added to q more than once. Adding (and later removing) each vertex from q takes constant time per vertex for a total of $O(n)$ time. Since each vertex is processed by the inner loop at most once, each adjacency list is processed at most once, so each edge of G is processed at most once. This processing, which is done in the inner loop takes constant time per iteration, for a total of $O(m)$ time. Therefore, the entire algorithm runs in $O(n+m)$ time.

The following theorem summarizes the performance of the $\mathrm{bfs}(g,r)$ algorithm.

Theorem 8.3. *When given as input a Graph, g, that is implemented using the AdjacencyLists data structure, the $\mathrm{bfs}(g,r)$ algorithm runs in $O(n+m)$ time.*

A breadth-first traversal has some very special properties. Calling $\mathrm{bfs}(g,r)$ will eventually enqueue (and eventually dequeue) every vertex j such that there is a directed path from r to j. Moreover, the vertices at distance 0 from r (r itself) will enter q before the vertices at distance 1, which will enter q before the vertices at distance 2, and so on. Thus, the $\mathrm{bfs}(g,r)$ method visits vertices in increasing order of distance from r and vertices that cannot be reached from r are never visited at all.

A particularly useful application of the breadth-first-search algorithm is, therefore, in computing shortest paths. To compute the shortest path from r to every other vertex, we use a variant of $\mathrm{bfs}(g,r)$ that uses an auxilliary array, p, of length n. When a new vertex j is added to q, we set $p[j] \leftarrow i$. In this way, $p[j]$ becomes the second last node on a shortest path from r to j. Repeating this, by taking $p[p[j]$, $p[p[p[j]]]$, and so on we can reconstruct the (reversal of) a shortest path from r to j.

8.3.2 Depth-First Search

The *depth-first-search* algorithm is similar to the standard algorithm for traversing binary trees; it first fully explores one subtree before returning to the current node and then exploring the other subtree. Another way to think of depth-first-search is by saying that it is similar to breadth-first search except that it uses a stack instead of a queue.

During the execution of the depth-first-search algorithm, each vertex, i, is assigned a colour, $c[i]$: *white* if we have never seen the vertex before, *grey* if we are currently visiting that vertex, and *black* if we are done visiting that vertex. The easiest way to think of depth-first-search is as a recursive algorithm. It starts by visiting r. When visiting a vertex i, we first mark i as *grey*. Next, we scan i's adjacency list and recursively visit any white vertex we find in this list. Finally, we are done processing i, so we colour i black and return.

```
dfs($g,r$)
    $c \leftarrow$ new_array($g.n$)
    dfs($g,r,c$)
```

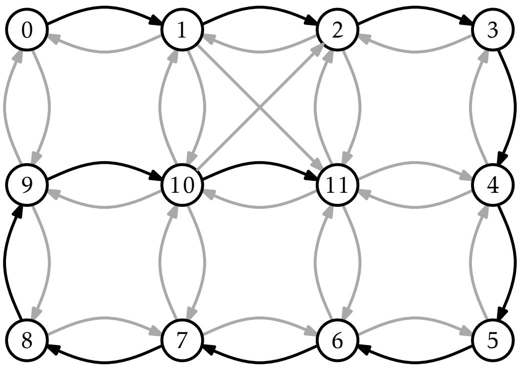


Figure 8.5: An example of depth-first-search starting at node 0. Nodes are labelled with the order in which they are processed. Edges that result in a recursive call are drawn in black, other edges are drawn in *grey*.

```
dfs($g,i,c$)
    $c[i] \leftarrow grey$
    for $j$ in $g$.out_edges($i$) do
        if $c[j] = white$ then
            $c[j] \leftarrow grey$
            dfs($g,j,c$)
    $c[i] \leftarrow black$
```

An example of the execution of this algorithm is shown in Figure 8.5.

Although depth-first-search may best be thought of as a recursive algorithm, recursion is not the best way to implement it. Indeed, the code given above will fail for many large graphs by causing a stack overflow. An alternative implementation is to replace the recursion stack with an explicit stack, s. The following implementation does just that:

```
dfs2($g,r$)
    $c \leftarrow$ new_array($g.n$)
    $s \leftarrow$ SLList()
    $s$.push($r$)
    while $s$.size() $> 0$ do
        $i \leftarrow s$.pop()
        if $c[i] = white$ then
            $c[i] \leftarrow grey$
            for $j$ in $g$.out_edges($i$) do
                $s$.push($j$)
```

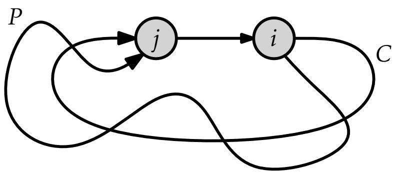


Figure 8.6: The depth-first-search algorithm can be used to detect cycles in G. The node j is coloured *grey* while i is still *grey*. This implies that there is a path, P, from i to j in the depth-first-search tree, and the edge (j,i) implies that P is also a cycle.

In the preceding code, when the next vertex, i, is processed, i is coloured *grey* and then replaced, on the stack, with its adjacent vertices. During the next iteration, one of these vertices will be visited.

Not surprisingly, the running times of $\mathrm{dfs}(g,r)$ and $\mathrm{dfs2}(g,r)$ are the same as that of $\mathrm{bfs}(g,r)$:

Theorem 8.4. *When given as input a Graph, g, that is implemented using the AdjacencyLists data structure, the $\mathrm{dfs}(g,r)$ and $\mathrm{dfs2}(g,r)$ algorithms each run in $O(n+m)$ time.*

As with the breadth-first-search algorithm, there is an underlying tree associated with each execution of depth-first-search. When a node $i \neq r$ goes from *white* to *grey*, this is because $\mathrm{dfs}(g,i,c)$ was called recursively while processing some node i'. (In the case of $\mathrm{dfs2}(g,r)$ algorithm, i is one of the nodes that replaced i' on the stack.) If we think of i' as the parent of i, then we obtain a tree rooted at r. In Figure 8.5, this tree is a path from vertex 0 to vertex 11.

An important property of the depth-first-search algorithm is the following: Suppose that when node i is coloured *grey*, there exists a path from i to some other node j that uses only white vertices. Then j will be coloured first *grey* then *black* before i is coloured *black*. (This can be proven by contradiction, by considering any path P from i to j.)

One application of this property is the detection of cycles. Refer to Figure 8.6. Consider some cycle, C, that can be reached from r. Let i be the first node of C that is coloured *grey*, and let j be the node that precedes i on the cycle C. Then, by the above property, j will be coloured *grey* and the edge (j,i) will be considered by the algorithm while i is still *grey*. Thus, the algorithm can conclude that there is a path, P, from i to j in the depth-first-search tree and the edge (j,i) exists. Therefore, P is also a cycle.

8.4 Discussion and Exercises

The running times of the depth-first-search and breadth-first-search algorithms are somewhat overstated by the Theorems 8.3 and 8.4. Define n_r as the number of vertices, i, of

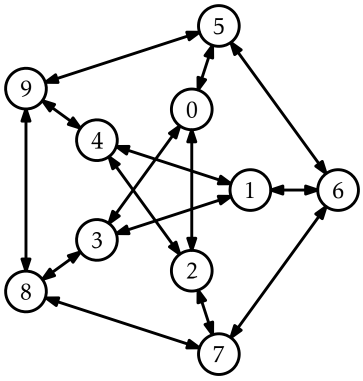


Figure 8.7: An example graph.

G, for which there exists a path from r to i. Define m_r as the number of edges that have these vertices as their sources. Then the following theorem is a more precise statement of the running times of the breadth-first-search and depth-first-search algorithms. (This more refined statement of the running time is useful in some of the applications of these algorithms outlined in the exercises.)

Theorem 8.5. *When given as input a Graph, g, that is implemented using the AdjacencyLists data structure, the* $\mathrm{bfs}(g,r)$, $\mathrm{dfs}(g,r)$ *and* $\mathrm{dfs2}(g,r)$ *algorithms each run in* $O(n_r+m_r)$ *time.*

Breadth-first search seems to have been discovered independently by Moore [30] and Lee [28] in the contexts of maze exploration and circuit routing, respectively.

Adjacency-list representations of graphs were presented by Hopcroft and Tarjan [23] as an alternative to the (then more common) adjacency-matrix representation. This representation, as well as depth-first-search, played a major part in the celebrated Hopcroft-Tarjan planarity testing algorithm that can determine, in $O(n)$ time, if a graph can be drawn, in the plane, and in such a way that no pair of edges cross each other [24].

In the following exercises, an undirected graph is one in which, for every i and j, the edge (i,j) is present if and only if the edge (j,i) is present.

Exercise 8.1. Draw an adjacency list representation and an adjacency matrix representation of the graph in Figure 8.7.

Exercise 8.2. The *incidence matrix* representation of a graph, G, is an $n\times m$ matrix, A, where

$$A_{i,j} = \begin{cases} -1 & \text{if vertex } i \text{ the source of edge } j \\ +1 & \text{if vertex } i \text{ the target of edge } j \\ 0 & \text{otherwise.} \end{cases}$$

1. Draw the incident matrix representation of the graph in Figure 8.7.

2. Design, analyze and implement an incidence matrix representation of a graph. Be sure to analyze the space, the cost of $\mathrm{add\_edge}(i,j)$, $\mathrm{remove\_edge}(i,j)$, $\mathrm{has\_edge}(i,j)$, $\mathrm{in\_edges}(i)$, and $\mathrm{out\_edges}(i)$.

Exercise 8.3. Illustrate an execution of the $\mathrm{bfs}(G,0)$ and $\mathrm{dfs}(G,0)$ on the graph, G, in Figure 8.7.

Exercise 8.4. Let G be an undirected graph. We say G is *connected* if, for every pair of vertices i and j in G, there is a path from i to j (since G is undirected, there is also a path from j to i). Show how to test if G is connected in $O(n+m)$ time.

Exercise 8.5. Let G be an undirected graph. A *connected-component labelling* of G partitions the vertices of G into maximal sets, each of which forms a connected subgraph. Show how to compute a connected component labelling of G in $O(n+m)$ time.

Exercise 8.6. Let G be an undirected graph. A *spanning forest* of G is a collection of trees, one per component, whose edges are edges of G and whose vertices contain all vertices of G. Show how to compute a spanning forest of of G in $O(n+m)$ time.

Exercise 8.7. We say that a graph G is *strongly-connected* if, for every pair of vertices i and j in G, there is a path from i to j. Show how to test if G is strongly-connected in $O(n+m)$ time.

Exercise 8.8. Given a graph $G=(V,E)$ and some special vertex $r\in V$, show how to compute the length of the shortest path from r to i for every vertex $i\in V$.

Exercise 8.9. Give a (simple) example where the $\mathrm{dfs}(g,r)$ code visits the nodes of a graph in an order that is different from that of the $\mathrm{dfs2}(g,r)$ code. Write a version of $\mathrm{dfs2}(g,r)$ that always visits nodes in exactly the same order as $\mathrm{dfs}(g,r)$. (Hint: Just start tracing the execution of each algorithm on some graph where r is the source of more than 1 edge.)

Exercise 8.10. A *universal sink* in a graph G is a vertex that is the target of $n-1$ edges and the source of no edges.[1] Design and implement an algorithm that tests if a graph G, represented as an AdjacencyMatrix, has a universal sink. Your algorithm should run in $O(n)$ time.

[1] A universal sink, v, is also sometimes called a *celebrity*: Everyone in the room recognizes v, but v doesn't recognize anyone else in the room.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008. `doi:10.1109/IEEESTD.2008.4610935`.

[2] Bibliography on hashing. URL: `http://liinwww.ira.uka.de/bibliography/Theory/hash.html` [cited 2011-07-20].

[3] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In M. 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.

[4] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In Dehne et al. [8], pages 37–48.

[5] J. Carter and M. Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[6] C. Crane. Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Computer Science Department, Stanford University, 1972.

[7] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, 2003.

[8] F. K. H. A. Dehne, A. Gupta, J.-R. Sack, and R. 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.

[9] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In C. Puech and R. 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.

[10] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In W. 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.

[11] 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.

[12] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[13] 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.

[14] M. Eytzinger. *Thesaurus principum hac aetate in Europa viventium (Cologne)*. 1590. In commentaries, ‘Eytzinger’ may appear in variant forms, including: Aitsingeri, Aitsingero, Aitsingerum, Eyzingern.

[15] R. W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7(12):701, 1964.

[16] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[17] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[18] 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.

[19] A. Gambin and A. Malinowski. Randomized meldable priority queues. In *SOFSEM’98: Theory and Practice of Informatics*, pages 344–349. Springer, 1998.

[20] M. T. Goodrich and J. G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Dehne et al. [8], pages 205–216.

[21] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.

[22] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[23] J. E. Hopcroft and R. E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[24] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[25] D. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[26] D. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[27] D. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[28] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transaction on Electronic Computers*, EC-10(3):346–365, 1961.

[29] E. Lehman, F. T. Leighton, and A. R. Meyer. *Mathematics for Computer Science*. 2011. URL: http://people.csail.mit.edu/meyer/mcs.pdf [cited 2012-09-06].

[30] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292, 1959.

[31] Oracle. *The Collections Framework*. URL: http://download.oracle.com/javase/1.5.0/docs/guide/collections/ [cited 2011-07-19].

[32] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[33] M. Pătraşcu and M. Thorup. The power of simple tabulation hashing. *Journal of the ACM*, 59(3):14, 2012.

[34] S. M. Ross. *Probability Models for Computer Science*. Academic Press, Inc., Orlando, FL, USA, 2001.

[35] 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.

[36] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM conference LISP and Functional Programming (LFP'94)*, pages 185–195, New York, 1994. ACM.

[37] P. Sinha. A memory-efficient doubly linked list. *Linux Journal*, 129, 2005. URL: http://www.linuxjournal.com/article/6828 [cited 2013-06-05].

[38] D. Sleator and R. 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.

[39] S. P. Thompson. *Calculus Made Easy*. MacMillan, Toronto, 1914. Project Gutenberg EBook 33283. URL: http://www.gutenberg.org/ebooks/33283 [cited 2012-06-14].

[40] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[41] J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

Index