Open Data Structures (in Java)

Pat Morin

Version 0.0 pre α : COMP2402 Fall 2011
# Contents

1. **Introduction** ................................................. 1
   1.1 Interfaces ............................................. 1
      1.1.1 The Queue, Stack, and Deque Interfaces .......... 1
      1.1.2 The List Interface: Linear Sequences ........... 2
      1.1.3 The USet Interface: Unordered Sets .......... 3
      1.1.4 The SSet Interface: Sorted Sets ............... 3
   1.2 Mathematical Background ................................. 4
      1.2.1 Logarithms ....................................... 4
      1.2.2 Asymptotic Notation .............................. 5
      1.2.3 Randomization and Probability ................. 6
   1.3 The Model of Computation ................................ 8
   1.4 Code Samples ........................................... 8
   1.5 References ............................................. 9

2. **Array-Based Lists and Queues** ............................ 11
   2.1 ArrayStack: Fast Stack Operations Using an Array .... 12
      2.1.1 The Basics ....................................... 12
      2.1.2 Growing and Shrinking ............................ 14
      2.1.3 Summary .......................................... 15
   2.2 FastArrayStack: An Optimized ArrayStack ............. 16
   2.3 ArrayQueue: An Array-Based Queue .................... 17
      2.3.1 Summary .......................................... 19
   2.4 ArrayDeque: Fast Deque Operations Using an Array .... 19
      2.4.1 Summary .......................................... 21
   2.5 DualArrayDeque: Building a Deque from Two Stacks ... 22
      2.5.1 Balancing ......................................... 24
      2.5.2 Summary .......................................... 26
   2.6 RootishArrayStack: A Space-Efficient Array Stack .... 26
      2.6.1 Analysis of Growing and Shrinking .............. 30
      2.6.2 Space Usage ....................................... 30
## CONTENTS

6.1.1 Recursive Algorithms ........................................ 79
6.1.2 Traversing Binary Trees ..................................... 79
7.2 **BinarySearchTree**: An Unbalanced Binary Search Tree .... 81
  6.2.1 Searching .................................................. 81
  6.2.2 Inserting .................................................... 84
  6.2.3 Deleting .................................................... 85
  6.2.4 Summary .................................................... 87
6.3 Discussion and Exercises ........................................ 88

7 Random Binary Search Trees ..................................... 89
  7.1 Random Binary Search Trees .................................. 89
    7.1.1 Proof of Lemma 7.1 ...................................... 91
    7.1.2 Summary .................................................. 92
  7.2 **Treap**: A Randomized Binary Search Tree ................. 93
    7.2.1 Summary .................................................. 98
  7.3 Summary and Exercises ........................................ 100

8 Scapegoat Trees .................................................. 103
  8.1 **ScapegoatTree**: A Binary Search Tree with Partial Rebuilding .... 104
    8.1.1 Analysis of Correctness and Running-Time .......... 106
    8.1.2 Summary .................................................. 109
  8.2 Discussion and Exercises .................................... 109

9 Red-Black Trees .................................................. 111
  9.1 2-4 Trees ..................................................... 111
    9.1.1 Adding a Leaf .......................................... 112
    9.1.2 Removing a Leaf ....................................... 112
  9.2 **RedBlackTree**: A Simulated 2-4 Tree ................. 115
    9.2.1 Red-Black Trees and 2-4 Trees ...................... 115
    9.2.2 Left-Leaning Red-Black Trees ..................... 117
    9.2.3 Addition ................................................. 120
    9.2.4 Removal ................................................ 122
  9.3 Summary .......................................................... 127
  9.4 Discussion and Exercises .................................... 128

10 Heaps .............................................................. 131
  10.1 BinaryHeap: An Implicit Binary Tree ..................... 131
    10.1.1 Summary ................................................. 136
  10.2 MeldableHeap: A Randomized Meldable Heap ............... 136
    10.2.1 Analysis of merge(h1,h2) ............................ 137
10.2.2 Summary ....................................................... 138
10.3 Discussion and Exercises ................................. 139
Chapter 1

Introduction

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

1.1 Interfaces

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

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

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

1.1.1 The Queue, Stack, and Deque Interfaces

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

- **add(x)**: add the value x to the Queue

- **remove()**: the next (previously added) value, y, from the Queue and return y
Notice that the \texttt{remove()} operation takes no argument. The \texttt{Queue}'s \textit{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 \textit{FIFO} (first-in-first-out) \texttt{Queue} removes items in the same order they were added, much in the same way a queue (or line-up) works when checking out at a cash register in a grocery store.

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

A very common queueing discipline is the \textit{LIFO} (last-in-first-out) discipline. In a \textit{LIFO Queue}, the most recently element added 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: \texttt{Stack}. Often, when discussing a \texttt{Stack} the names of \texttt{add(x)} and \texttt{remove()} are changed to \texttt{push(x)} and \texttt{pop()}, to avoid confusing the LIFO and FIFO queueing disciplines.

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

1.1.2 The \texttt{List} Interface: Linear Sequences

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

1. \texttt{size()}: return $n$, the length of the list
2. \texttt{get(i)}: return the value $x_i$
3. \texttt{set(i)}: set the value of $x_i$ equal to $x$
4. \texttt{add(i,x)}: add $x$ at position $i$, displacing $x_1, \ldots, x_{n-1}$;
   Set $x_{j+1} = x_j$, for all $j \in \{n - 1, \ldots, i\}$, increment $n$, and set $x_i = x$
5. \texttt{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 $n$

Notice that these operations are easily sufficient to implement the \texttt{Deque} interface:

\begin{align*}
\texttt{addFirst}(x) & \Rightarrow \texttt{add}(0, x) \\
\texttt{removeFirst}(x) & \Rightarrow \texttt{remove}(0) \\
\texttt{addLast}(x) & \Rightarrow \texttt{add}(\texttt{size}(), x) \\
\texttt{removeLast}(x) & \Rightarrow \texttt{remove}(\texttt{size}() - 1)
\end{align*}
Although we will normally not discuss the Stack, Deque and FIFO Queue interfaces very often in subsequent chapters, the terms Stack and Deque are sometimes used in the names of data structures that implement the List interface. When this happens, it is to highlight the face that these data structures can be used to implement the Stack or Deque interface very efficiently. For example, the ArrayDeque class is an implementation of the List interface that can implement the Deque operations in constant (amortized) time per operation.

1.1.3 The USet Interface: Unordered Sets

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

1. size(): return the number, n, of elements in the set
2. add(x): add the element x to the set if not already present;
   Add x to the set provided that there is no element y in the set such that x equals y.
   Return true if x was added to the set and false otherwise.
3. remove(x): remove x from the set;
   Find an element y in the set such that x equals y and remove y. Return y, or null if
   no such element exists.
4. find(x): find x in the set if it exists;
   Find an element y in the set such that y equals x. Return y, or null if no such element
   exists.

These definitions are a bit fussy about distinguishing x, the element we are removing or finding from y, the element we remove or find. This is because x and y may actually be distinct objects that are nevertheless treated as equal.1 This is a very useful distinction since it allows for the creation of dictionaries or maps that map keys onto values. This is done by creating a compound object called a Pair that contains a key and a value. Two Pairs are treated as equal if their keys are equal. By storing Pairs in a USet, we can find the value associated with any key k by creating a Pair, x, with key k and using the find(x)
method.

1.1.4 The SSet Interface: Sorted Sets

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

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

1In Java, this is done by overriding the class’ equals(y) and hashCode() methods.
An SSet supports the size(), add(x), and remove(x) methods with exactly the same semantics as in the USet interface. The difference between a USet and a SSet is in the find(x) method:

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

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

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

1.2 Mathematical Background

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

1.2.1 Logarithms

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

\[
b^x = k.
\]

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

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

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

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

\[
\int_1^k 1/x \, dx = \ln k.
\]

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

\[
b^{\log_b k} = k
\]

and changing the base of a logarithm:

\[
\log_b k = \frac{\log_a k}{\log_a b}.
\]
For example, we can use these two manipulations to compare the natural and binary logarithms:
\[ \ln k = \log_k \frac{\log k}{\log e} = \frac{(\ln 2)(\log k)}{\ln 2} \approx 0.693147 \log k. \]

### 1.2.2 Asymptotic Notation

When analyzing data structures in this book we will want to talk about the running times of various operations. The exact running-times will, of course, vary from computer to computer and even from run to run on an individual computer. Therefore, instead of analyzing running times exactly, we will use the so-called big-Oh notation: For a non-decreasing function \( f(n) \), \( O(f(n)) \) denotes a set of functions:

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

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

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

\[
5n \log n + 8n - 200 \leq 5n \log n + 8n \\
\leq 5n \log n + 8n \log n \quad \text{for } n \geq 2 \quad (\text{so that } \log n \geq 1) \\
\leq 13n \log n
\]

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

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

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

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

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

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

\[ O(n) \subset O(n \log n) \subset O(n^{1+b}) \subset O(ne^n) . \]

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

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

\[ T(n) = 2 \log n + O(1) . \]
Again, this would be more correctly written as

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

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

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

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

This definition captures the situation we really care about: When the arguments \( n_1, \ldots, n_k \) make \( g \) take on large values. The reader should be warned that, although this works for our purposes, other texts may treat multivariate functions and asymptotic notation differently.

### 1.2.3 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 finite universe \( U \), the expected value of \( X \), denoted by \( E[X] \) is given by the formula

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

Here \( \Pr\{E\} \) denotes the probability that the event \( E \) occurs. In all the examples in this book these probabilities are only with respect to whatever random choices are made by the randomized data structure; there is no assumption that the data stored in the structure is random or that the sequence of operations performed on the data structure are random.

One of the most important properties of expected values is linearity of expectation:

For any two random variables \( X \) and \( Y \),

\[ E[X + Y] = E[X] + E[Y] . \]

More generally, for any random variables \( X_1, \ldots, X_k \),

\[ E \left[ \sum_{i=1}^{k} X_k \right] = \sum_{i=1}^{k} E[X_i] . \]
Linearity of expectation allows us to break down complicated random variables (like the left hand sides of the above equations) into sums of simpler random variables (the right hand sides).

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

$$E[X] = \sum_{i=0}^{k} i \cdot \Pr\{X = i\}$$

$$= \sum_{i=0}^{k} i \cdot \left(\frac{k}{i}\right)/2^k$$

$$= k \cdot \sum_{i=0}^{k-1} \left(\frac{k-1}{i}\right)/2^k$$

$$= k/2 .$$

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

Using indicator variables and linearity of independence 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

$$E[I_i] = (1/2)1 + (1/2)0 = 1/2 .$$

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

$$E[X] = E\left[\sum_{i=1}^{k} I_i\right]$$

$$= \sum_{i=1}^{k} E[I_i]$$

$$= \sum_{i=1}^{k} 1/2$$

$$= k/2 .$$

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

In this book, we will analyze the theoretical running times of operations on the data structures we study. To do this precisely, we need a mathematical model of computation. For this, we use the \( w \)-bit word-RAM model. In this model, we have access to a random access memory consisting of cells, each of which stores a \( w \)-bit word. This implies 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 (\(+, -, \ast, /, \%\)), comparisons (\(<, >, =, \leq, \geq\)) and bitwise boolean operations (bitwise-AND, OR, and exclusive-OR).

Any cell can be read or written in constant time. Our computer’s memory is managed by a memory management system from which we can allocate or deallocate a block of memory of any size we like. Allocating a block of memory of size \( k \) takes \( O(k) \) time and returns a reference to 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 on \( w \) is that it is at least \( w \geq \log n \), 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 our data structures store values of a generic type \( T \) and we assume an element of type \( T \) occupies one word of memory. (In reality, we are storing references to objects of type \( T \), and these references occupy only one word of memory.)

The \( w \)-bit word-RAM model is a fairly close match for the (32-bit) Java Virtual Machine (JVM) when \( w = 32 \). The data structures presented in this book don’t use any special tricks that are not implementable on the JVM and most other architectures.

1.4 Code Samples

The code samples in this book are written in the Java programming language. However to make the book accessible even to readers not familiar with all of Java’s constructs and keywords, the code samples have been simplified. For example, a reader won’t find any of the keywords public, protected, private, or static. A reader also won’t find the extends or implements keywords, so any notion of class hierarchy is missing. Which interface a particular class implements or which class it extends, if relevant to the discussion, will be clear from the accompanying text.

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

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

1.5 References

The List, USet, and SSet interfaces described in Section 1.1 are influenced by the Java Collections Framework [36]. These are essentially simplified versions of the List, Set/Map, and SortedSet/SortedMap interfaces found in the Java Collections Framework. Indeed, the accompanying source code includes wrapper classes for making USet and SSet implementations into Set, Map, SortedSet, and SortedMap implementations.

For more information on basic probability, especially as it relates to computer science, see the textbook by Ross [46]. Another good reference that covers both asymptotic notation and probability is the textbook by Graham, Knuth, and Patashnik [27].

Readers wanting to brush up on their Java programming can find many Java tutorial online [38].
Chapter 2

Array-Based Lists and Queues

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

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>get(i)</th>
<th>set(i,x)</th>
<th>add(i,x)</th>
<th>remove(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayStack</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1 + n - i)$</td>
<td>$O(n - i)$</td>
</tr>
<tr>
<td>ArrayDeque</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1 + \min{i, n - i})$</td>
<td>$O(1 + \min{i, n - i})$</td>
</tr>
<tr>
<td>DualArrayDeque</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1 + \min{i, n - i})$</td>
<td>$O(1 + \min{i, n - i})$</td>
</tr>
<tr>
<td>RootishArrayStack</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1 + n - i)$</td>
<td>$O(n - i)$</td>
</tr>
</tbody>
</table>

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 $\text{get}(i)$ and $\text{set}(i, x)$ 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 $\text{add}(i, x)$ and $\text{remove}(i)$ have running times that depend on $n$ and $i$.

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

The third point is important. The running times cited in the table above do not include the cost of growing and shrinking the backing array. We will see that, if carefully managed, the cost of growing and shrinking the backing array does not add much to the cost of an average operation. More precisely, if we start with an empty data structure, and perform any sequence of $m$ $\text{add}(i, x)$ or $\text{remove}(i)$ 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 require growing or shrinking the backing array, and are therefore 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, \( a.length \geq n \).

```java
T[] a;
int n;
int size() {
    return n;
}
```

### 2.1.1 The Basics

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

```java
T get(int i) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    return a[i];
}
T set(int i, T x) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    T y = a[i];
    a[i] = x;
    return y;
}
```

Adding and removing elements from an ArrayStack is illustrated in Figure 2.1. To implement the \( \text{add}(i, x) \) operation, we first check if \( a \) is already full. If so, we call the method \( \text{resize}() \) to increase the size of \( a \). How \( \text{resize}() \) is implemented will be discussed later. For now, it is sufficient to know that, after a call to \( \text{resize}() \), we can be sure that \( a.length > n \). 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 \( n \).

```java
void add(int i, T x) {
    if (i < 0 || i > n) throw new IndexOutOfBoundsException();
    if (n + 1 > a.length) resize();
    for (int j = n; j > i; j--)
        a[j] = a[j-1];
    a[i] = x;
    n++;
}
```
2.1. **ARRAYSTACK: FAST STACK OPERATIONS USING AN ARRAY**

Figure 2.1: A sequence of \( \text{add}(i,x) \) and \( \text{remove}(i) \) operations on an ArrayStack. Arrows denote elements being copied. Operations that result in a call to \( \text{resize}() \) are marked with an asterisk.
If we ignore the cost of the potential call to `resize()`, the cost of the `add(i, x)` operation is proportional to the number of elements we have to shift to make room for `x`. Therefore the cost of this operation (ignoring the cost of resizing `a`) is $O(n - i + 1)$.

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 `a.length` by checking if `a.length ≥ 3n`. If so, we call `resize()` to reduce the size of `a`.

```java
ArrayStack T remove(int i) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    T x = a[i];
    for (int j = i; j < n-1; j++)
        a[j] = a[j+1];
    n--;
    if (a.length >= 3*n) resize();
    return x;
}
```

If we ignore the cost of the `resize()` method, the cost of a `remove(i)` operation is proportional to the number of the number 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()`, `a.length = 2n`.

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

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

**Lemma 2.1.** If an empty `ArrayList` is created and any sequence of $m \geq 1$ calls to `add(i, x)` and `remove(i)` are performed, then the total time spent during all calls to `resize()` is $O(m)$.
Proof. We will show that anytime `resize()` is called, the number of calls to `add` or `remove` since the last call to `resize()` is at least \( n/2 - 1 \). Therefore, if \( n_i \) denotes the value of \( n \) during the \( i \)th call to `resize()` and \( r \) denotes the number of calls to `resize()`, then the total number of calls to `add(i,x)` or `remove(i)` is at least

\[
\sum_{i=1}^{r} \left( n_i/2 - 1 \right) \leq m ,
\]

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 `resize()` is

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

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

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

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

\[
a.length/2 - 1 - a.length/3 = a.length/6 - 1 = (a.length/3)/2 - 1 \geq n_i/2 - 1 .
\]

In either case, the number of calls to `add(i,x)` or `remove(i)` that occur between the \((i-1)\)th call to `resize()` and the \(i\)th call to `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 `resize()`, an `ArrayStack` supports the operations

- `get(i)` and `set(i,x)` in \( O(1) \) time per operation; and

\(^1\)The \(-1\) in this formula accounts for the special case that occurs when \( n = 0 \) and `a.length = 1`. 
• \( \text{add}(i, x) \) and \( \text{remove}(i) \) in \( O(1 + n - i) \) time per operation.

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

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

\subsection*{2.2 FastArrayStack: An Optimized ArrayStack}

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

```java
void resize() {
    T[] b = newArray(max(2*n,1));
    System.arraycopy(a, 0, b, 0, n);
    a = b;
}
void add(int i, T x) {
    if (i < 0 || i > n) throw new IndexOutOfBoundsException();
    if (n + 1 > a.length) resize();
    System.arraycopy(a, i, a, i+1, n-i);
    a[i] = x;
    n++;
}
T remove(int i) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    T x = a[i];
    System.arraycopy(a, i+1, a, i, n-i-1);
    n--;
    if (a.length >= 3*n) resize();
    return x;
}
```

These functions are usually highly optimized and may even use special machine instructions that can do this copying much faster than we could do with \texttt{for} loop. Although using these functions does not asymptotically decrease the running times, it can still be a worthwhile optimization. In the Java implementations here, the use of \texttt{System.arraycopy(s, i, d, j, n)} resulted in speedups of a factor of 2-3 depending on the types of operations performed.
2.3 **ArrayQueue: An Array-Based Queue**

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

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

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

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

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 o'clock plus 5 hours gives 3 o'clock. Formally, we say that

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

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

\[
15 \mod 12 = 3.
\]

More generally, for an integer \( a \) and positive integer \( m \), \( a \mod m \) 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 Java, the mod operator is represented using the \( \% \) symbol.\(^2\)

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

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

This treats \( a \) like a *circular array* in which array indices exceeding \( a.length - 1 \) “wrap around” to the beginning of the array.

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

\(^2\)This is sometimes referred to as the *brain-dead* mod operator since it does not correctly implement the mathematical mod operator when the first argument is negative.
A sequence of `add(x)` and `remove(i)` operations on an `ArrayQueue`. Arrows denote elements being copied. Operations that result in a call to `resize()` are marked with an asterisk.

```java
ArrayQueue {
    T[] a;
    int j;
    int n;
}
```

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

```java
boolean add(T x) {
    if (n + 1 > a.length) resize();
    a[(j+n) % a.length] = x;
    n++;
    return true;
}
```
To implement `remove()` we first store `a[j]` so that we can return it later. Next, we decrement `n` and “increment” `j` by setting `j = (j + 1) mod a.length` and return the stored value of `a[j]`. If necessary, we may call `resize()` to decrease the size of `a`.

```java
ArrayQueue
T remove() {
    if (n == 0) throw new NoSuchElementException();
    T x = a[j];
    j = (j + 1) % a.length;
    n--;
    if (a.length >= 3*n) resize();
    return x;
}
```

Finally, the `resize()` operation is very similar to the `resize()` operation of `ArrayQueue`. It allocates a new array `b` of size `2n` and copies `a[j], a[(j + 1) % a.length], ..., a[(j + n - 1) % a.length]` onto `b[0], b[1], ..., b[n - 1]` and sets `j = 0`.

```java
ArrayQueue
void resize() {
    T[] b = newArray(max(1, n*2));
    for (int k = 0; k < n; k++)
        b[k] = a[(j+k) % a.length];
    a = b;
    j = 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 using the same circular array technique used to represent an `ArrayQueue`. 
Figure 2.3: A sequence of add(i, x) and remove(i) operations on an ArrayDeque. Arrows denote elements being copied.

```
T[] a;
int j;
int n;
```

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

```
T get(int i) {
    if (i < 0 || i > n-1) throw new IndexOutOfBoundsException();
    return a[(j+i)%a.length];
}
T set(int i, T x) {
    if (i < 0 || i > n-1) throw new IndexOutOfBoundsException();
    T y = a[(j+i)%a.length];
    a[(j+i)%a.length] = x;
    return y;
}
```

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

```
void add(int i, T x) {
    if (i < 0 || i > n) throw new IndexOutOfBoundsException();
```
2.4. **ARRAYDEQUE: FAST DEQUE OPERATIONS USING AN ARRAY**

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

By doing the shifting in this way, we guarantee that 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 remove(i) operation is similar. It either shifts elements a[0],...,a[i - 1] right by one position or shifts the elements a[i + 1],...,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.

---

### ArrayDeque

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

#### 2.4.1 Summary

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

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

- get(i) and set(i, x) in $O(1)$ time per operation; and
CHAPTER 2. ARRAY-BASED LISTS AND QUEUES

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

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

2.5 DualArrayDeque: Building a Deque from Two Stacks

Next, we present another data structure, the DualArrayDeque that achieves the same performance bounds as an `ArrayDeque` by using two `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.

```java
List<T> front;
List<T> back;
```

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

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

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

```java
T get(int i) {
    if (i < front.size()) {
        return front.get(front.size()-i-1);
    } else {
        return back.get(i-front.size());
    }
}
T set(int i, T x) {
    if (i < front.size()) {
        return front.set(front.size()-i-1, x);
    } else {
        return back.set(front.size()-i, x);
    }
}
```
2.5. DUALARRAYDEQUE: BUILDING A DEQUE FROM TWO STACKS

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 `rebalance()` are marked with an asterisk.

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

The add(i, x) method performs rebalancing of the two ArrayStacks `front` and `back`, by calling the `balance()` method. The implementation of `balance()` is described below, but for now it is sufficient to know that `balance()` ensures that, unless `size() < 2`,
CHAPTER 2. ARRAY-BASED LISTS AND QUEUES

front.size() and back.size() do not differ by more than a factor of 3. In particular, 3front.size() ≥ back.size() and 3back.size() ≥ front.size().

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

\[ O(front.size() − (front.size() − i − 1) + 1) = O(i + 1). \] (2.1)

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

\[ O(back.size() − (i − front.size()) + 1) = O(size() − i + 1) = O(n − i + 1). \] (2.2)

Notice that the first case (2.1) occurs when i < n/4. The second case (2.2) occurs when i ≥ 3n/4. When n/4 ≤ i < 3n/4, we can’t 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 ≥ n/4 and n − i > n/4. Summarizing the situation, we have

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

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

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

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

2.5.1 Balancing

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

```java
void balance() {
    int n = size();

    // Balance logic...
}
```
if (3*front.size() < back.size()) {
    int s = n/2 - front.size();
    List<T> l1 = newStack();
    List<T> l2 = newStack();
    l1.addAll(back.subList(0, s));
    Collections.reverse(l1);
    l1.addAll(front);
    l2.addAll(back.subList(s, back.size()));
    front = l1;
    back = l2;
} else if (3*back.size() < front.size()) {
    int s = front.size() - n/2;
    List<T> l1 = newStack();
    List<T> l2 = newStack();
    l1.addAll(front.subList(s, front.size()));
    l2.addAll(front.subList(0, s));
    Collections.reverse(l2);
    l2.addAll(back);
    front = l1;
    back = l2;
}

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

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

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

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

\[
\Phi = |\text{front.size()} - \text{back.size()}|.
\]

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

Observe that, immediately after a call to balance() that shifts elements, the potential, \(\Phi_0\), is at most 1, since

\[
\Phi_0 = |\lfloor n/2 \rfloor - \lceil n/2 \rceil| \leq 1.
\]


Now, consider the situation immediately before a call to balance() that shifts elements, and suppose, without loss of generality that balance() is shifting elements because 3front.size() < back.size(). Notice that, in this case,

\[ n = \text{front.size()} + \text{back.size()} \]
\[ < \text{back.size()}/3 + \text{back.size()} \]
\[ = \frac{4}{3} \text{back.size()} \]

Furthermore, the potential at this point in time is

\[ \Phi_1 = \text{back.size()} - \text{front.size()} \]
\[ > \text{back.size()} - \text{back.size()}/3 \]
\[ = \frac{2}{3} \text{back.size()} \]
\[ > \frac{2}{3} \times \frac{3}{4} n \]
\[ = \frac{n}{2} \]

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

### 2.5.2 Summary

The following theorem summarizes the performance of a DualArrayStack

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

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

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

### 2.6 RootishArrayStack: A Space-Efficient Array Stack

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

In this section, we discuss a data structure, the RootishArrayStack, that addresses this 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.
2.6. **ROOTISHARRAYSTACK: A SPACE-EFFICIENT ARRAY STACK**

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

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

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

```java
RootishArrayStack
List<T[]> blocks;
int n;
```

The elements of the list are laid out in the blocks as we might expect. The list element with index 0 is stored in block 0, the elements with list indices 1 and 2 are stored in block 1, the elements with list indices 3, 4, and 5 are stored in block 2, and so on. The main problem we have to address is that of determining, given an index $i$, which block contains $i$ as well as the index corresponding to $i$ within that block.

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

$$j = i - \frac{b(b + 1)}{2}$$

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 $0, \ldots, b$ is $(b + 1)(b + 2)/2$. Therefore, $b$ is the

![Figure 2.5: A sequence of add(i,x) and remove(i) operations on a RootishArrayStack. Arrows denote elements being copied.](image)
CHAPTER 2. ARRAY-BASED LISTS AND QUEUES

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.

The smallest integer such that

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

We can rewrite this equation as

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

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

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

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

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

```java
T get(int i) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    int b = i2b(i);
    int j = i - b*(b+1)/2;
    }
2.6. **ROOTISHARRAYSTACK: A SPACE-EFFICIENT ARRAY STACK**

```
return blocks.get(b)[j];
}
T set(int i, T x) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    int b = i2b(i);
    int j = i - b*(b+1)/2;
    T y = blocks.get(b)[j];
    blocks.get(b)[j] = x;
    return y;
}
```

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

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

```
void add(int i, T x) {
    if (i < 0 || i > n) throw new IndexOutOfBoundsException();
    int r = blocks.size();
    if (r*(r+1)/2 < n + 1) grow();
    n++;
    for (int j = n-1; j > i; j--)
        set(j, get(j-1));
    set(i, x);
}
```

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

```
void grow() {
    blocks.add(newArray(blocks.size()+1));
}
```

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

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

```
T remove(int i) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    T x = get(i);
    ```
for (int j = i; j < n-1; j++)
    set(j, get(j+1));
    n--;
    int r = blocks.size();
    if ((r-2)*(r-1)/2 >= n) shrink();
    return x;
}

RootishArrayStack

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

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

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 do any copying of data. They only allocate or free an array of size $r$. In some environments, this takes only constant time, while in others, it may require $\Theta(r)$ time.

We note that, immediately after a call to grow() or shrink(), the situation is clear. The final block is completely empty and all other blocks are completely full. Another call to grow() or shrink() will not happen until at least $r - 1$ 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 remove(i) operation ensures that a RootishArrayStack never has more than 2 blocks that are not completely full. The number of blocks, $r$, used by a RootishArrayStack that stores $n$ elements therefore satisfies

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

Again, using the quadratic equation on this gives

$$r \leq (3 + \sqrt{1 + 4n})/2 = O(\sqrt{n})$$
The last two blocks have sizes $r$ and $r-1$, so the space wasted by these two blocks is at most $2r - 1 = O(\sqrt{n})$. If we store the blocks in (for example) an ArrayList, then the amount of space wasted by the List that stores those $r$ blocks is also $O(r) = O(\sqrt{n})$. The other space needed for storing $n$ and other accounting information is $O(1)$. Therefore, the total amount of wasted space in a RootishArrayStack is $O(\sqrt{n})$.

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

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

2.6.3 Summary

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

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

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

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

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

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 [32, Section 2.2.2].

Brodnik et al. [9] 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

---

$^3$Recall Section 1.3 for a discussion of how memory is measured.
roots in the $i2b(i)$ method. With their scheme, the block containing $i$ is block $\lfloor \log(i+1) \rfloor$, which is just 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 2-level tiered-vector of Goodrich and Kloss [26]. This structure supports $\text{get}(i, x)$ and $\text{set}(i, x)$ in constant time and $\text{add}(x)$ and $\text{remove}(x)$ 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 2.9.

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

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

Exercise 2.3. Design and implement a Treque (triple-ended queue). This is a List implementation in which $\text{get}(i)$ and $\text{set}(i, x)$ run in constant time and $\text{add}(i, x)$ and $\text{remove}(i)$ run in time $O(1 + \min\{i, n - i, |n/2 - i|\})$.

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

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

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

Exercise 2.6. Modify the ArrayDeque implementation so that it does not use the $\%$ operator (which is expensive on some systems). Instead, it should make use of the fact that, if $a$ is the power of 2, then $k\%a.length = k & (a.length - 1)$. (Here, $\&$ is the bitwise-and operator.)

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

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

Test the performance of your implementation against the ArrayDeque. Optimize your implementation (by using System.arraycopy($a$, $i$, $b$, $i$, $n$)) and see if you can get it to outperform the ArrayDeque implementation.
Exercise 2.8. Design and implement a version of a RootishArrayStack that has only $O(\sqrt{n})$ wasted space, but that can perform \texttt{add}(i, x) and \texttt{remove}(i, x) operations in $O(1+\min\{i, n-i\})$ time.

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

Exercise 2.10. Design and implement a version of a RootishArrayStack that has only $O(\sqrt{n})$ wasted space, but that can perform \texttt{add}(i, x) and \texttt{remove}(i, x) operations in $O(1+\min\{i, \sqrt{n}, n-i\})$ time. (See Section 3.3 for ideas on how to achieve this.)
Chapter 3

Linked Lists

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

Compared to array-based list implementations, linked lists have their advantages and disadvantages. The primary disadvantage is that we lose the ability to access any element using get(i) or set(i, x) in constant time. Instead, we have to walk through the list, one element at a time, until we reach the 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 = null

```java
class Node {
    T x;
    Node next;
}
```

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

```java
Node head;
Node tail;
int n;
```

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

An SLList can efficiently implement the Stack operations push() and pop() by adding and removing elements at the head of the sequence. The push() operation simply
CHAPTER 3. LINKED LISTS

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

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:

```java
SLList<T> push(T x) {
    Node u = new Node();
    u.x = x;
    u.next = head;
    head = u;
    if (n == 0)
        tail = u;
    n++;
    return x;
}
```

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

```java
SLList<T> pop() {
    if (n == 0) return null;
    T x = head.x;
    head = head.next;
    if (--n == 0) tail = null;
    return x;
}
```

Clearly both the push(x) and pop() operations run in O(1) time.
3.1.1 Queue Operations

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

```java
SLList remove() {
    if (n == 0) return null;
    T x = head.x;
    head = head.next;
    if (--n == 0) tail = null;
    return x;
}
```

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

```java
boolean add(T x) {
    Node u = new Node();
    u.x = x;
    if (n == 0) {
        head = u;
    } else {
        tail.next = u;
    }
    tail = u;
    n++;
    return true;
}
```

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

3.1.2 Summary

The following theorem summarizes the performance of an SLList:

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

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

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

```
class Node {
    T x;
    Node prev, next;
}
```

When implementing an SLList, we saw that there were always some special cases to worry about. For example, removing the last element from an SLList or adding an element to an empty SLList requires special care so that \text{head} and \text{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 \text{next} and a \text{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.

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

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

![Figure 3.2: A DLList containing a,b,c,d,e.](image-url)
3.2. **DLLIST: A DOUBLY-LINKED LIST**

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

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

```java
T get(int i) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    return getNode(i).x;
}
T set(int i, T x) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    Node u = getNode(i);
    T y = u.x;
    u.x = x;
    return y;
}
```

The running time of these operations is dominated by the time it takes to find the *i*th 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 \(u.next.prev\). Thanks to the dummy node, there is no need to worry about \(w.prev\) or \(w.next\) not existing.

```java
Node addBefore(Node w, T x) {
    Node u = new Node();
    u.x = x;
    u.prev = w.prev;
    u.next = w;
    return u;
}
```
Now, the list operation \texttt{add}(i, x) is trivial to implement. We find the \textit{i}th node in 
the \texttt{DLList} and insert a new node \texttt{u} that contains \texttt{x} just before it.

```java
u.next.prev = u;
u.prev.next = u;
n++;
return u;
}
```

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

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

```java
void remove(Node w) {
    w.prev.next = w.next;
    w.next.prev = w.prev;
    n--;
}
```

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

```java
T remove(int i) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    Node w = getNode(i);
    remove(w);
    return w.x;
}
```

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

3.2.2 Summary

The following theorem summarizes the performance of a \texttt{DLList}:

**Theorem 3.2.** A \texttt{DLList} implements the \texttt{List} interface. The \texttt{get}(i), \texttt{set}(i, x), \texttt{add}(i, x) 
and \texttt{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 `getNode(i)` operation, then all operations on a `DLList` take constant time. Thus, the only expensive part of operations on a `DLList` is finding the relevant node. Once we have the relevant node, adding, removing, or accessing the data at that node take 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. An example of this is the `LinkedHashSet` data structure found in the Java Collections Framework, in which a set of items is stored in a doubly-linked list and the nodes of the doubly-linked list are stored in a hash table (discussed in Chapter 5). When elements are removed from a `LinkedHashSet`, the hash table is used to find the relevant list node in constant time and then the list node is deleted (also in constant time).

### 3.3 SEList: A Space-Efficient Linked List

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

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

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

```java
class BDeque extends ArrayDeque<T> {
    BDeque() {
        super(SEList.this.type());
        a = newArray(b+1);
    }
    void grow() { }
    void shrink() { }
}
```

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

\[
\frac{n}{b-1} + 1 = O\left(\frac{n}{b}\right)
\]

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

3.3.2 Finding Elements

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

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

\(^1\)See Section 2.6.2 for a proof of this lower bound.
Location getLocation(int i) {
    if (i < n/2) {
        Node u = dummy.next;
        while (i >= u.d.size()) {
            i -= u.d.size();
            u = u.next;
        }
        return new Location(u, i);
    } else {
        Node u = dummy;
        int idx = n;
        while (i < idx) {
            u = u.prev;
            idx -= u.d.size();
        }
        return new Location(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 we reach the node we want after \( O(1 + i/b) \) steps. If we search backwards, 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 \( \text{get}(i) \) and \( \text{set}(i, x) \) operations translate into getting or setting a particular index in the correct block:

T get(int i) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    Location l = getLocation(i);
    return l.u.d.get(l.j);
}

T set(int i, T x) {
    if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
    Location l = getLocation(i);
    T y = l.u.d.get(l.j);
    l.u.d.set(l.j,x);
    return y;
}

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

Where things start to get complicated is in adding elements. Before considering the general case, we first consider the easier operation, \texttt{add(x)}, in which \( x \) is added to the end of the list. In this case, 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.

```java
boolean add(T x) {
    Node last = dummy.prev;
    if (last == dummy || last.d.size() == b+1) {
        last = addBefore(dummy);
    }
    last.d.add(x);
    n++;
    return true;
}
```

Things get more complicated when we add to the interior of the list using \texttt{add(i,x)}. We first locate \( i \) to get the node \( u \) whose block contains the \( i \)th list item. The problem is that we want to insert \( x \) into \( u \)'s block, but we have to be prepared for the case where \( u \)'s block already contains \( b+1 \) 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 3.3):

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

```java
void add(int i, T x) {
    if (i < 0 || i > n) throw new IndexOutOfBoundsException();
    if (i == n) {
        add(x);
        return;
    }
    Location l = getLocation(i);
```
Figure 3.3: The three cases that occur during the addition of an item $x$ in the interior of an SEList. (This SEList has block size $b = 3$.)

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 \texttt{spread}(u) method, which moves
Figure 3.4: The three cases that occur during the removal of an item \(x\) in the interior of an \(SEList\). (This \(SEList\) has block size \(b = 3\).)

\[ b(b + 1) \text{ elements and takes } O(b^2) \text{ time. If we ignore the cost of Case 3 (which we will account for later with amortization) this means that the total running time to locate } i \text{ and perform the insertion of } x \text{ is } O(b + \min\{i, n - i\}/b). \]

### 3.3.4 Removing an Element

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

Again, let \(u_0, u_1, u_2, \ldots\) denote \(u, u_.next, u_.next.next\). We examine \(u_0, u_1, u_2, \ldots\) in order looking for a node from which we can borrow an element to make the size of \(u_0\)'s block larger than \(b - 1\). There are three cases to consider (see Figure 3.4):

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, 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 the last block, and there is no requirement that \(u_r\)'s block 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 \(\text{gather}\)
these \(b(b-1)\) elements into \(u_0, \ldots, u_{b-2}\) so that each of these \(b-1\) blocks contains exactly \(b\) elements and we remove \(u_{b-1}\), which is now empty. Now \(u_0\)’s block contains \(b\) elements so we can remove the appropriate element from it.

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

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

### 3.3.5 Amortized Analysis of Spreading and Gathering

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

```java
void spread(Node u) {
    Node w = u;
    for (int j = 0; j < b; j++) {
        w = w.next;
    }
    w = addBefore(w);
    while (w != u) {
        while (w.d.size() < b)
            w.d.add(0, w.prev.d.remove(w.prev.d.size()-1));
    }
}
```
The running time of each of these methods is dominated by the two nested loops. Both the inner loop and outer loop execute at most $b + 1$ 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 `add(i, x)` or `remove(i).

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

**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 **strong**. Define the **potential** of an `SEList` as the number of fragile nodes it contains. We will consider only the `add(i, x)` operation and its relation to the number of calls to `spread(u)`. The analysis of `remove(i)` and `gather(u)` is identical.

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

Finally, if Case 3 occurs, it is because $u_0, \ldots, u_{b-1}$ are all fragile nodes. Then `spread(u_0)` is called and these $b$ fragile nodes are replaced with $b + 1$ strong 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 `spread(u)` there are at least $b$ calls to `add(i, x)`. This completes the proof. □
3.3.6 Summary

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

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

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

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

The space (measured in words)\(^2\) used by an SEList that stores \(n\) elements is \(n + O(b + n/b)\).

The SEList is a tradeoff between an ArrayList and a DLList where the relative mix of these two structures depends on the block size \(b\). At the extreme \(b = 2\), each SEList node stores at most 3 values, which is really 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 tradeoff between the time it takes to add or remove a list item and the time it takes to locate a particular list item.

3.4 Discussion and Exercises

Both singly-linked and doubly-linked lists are folklore, having been used in programs for over 40 years. They are discussed, for example, by Knuth [32, Sections 2.2.3–2.2.5]. Even the SEList data structure seems to be a well-known data structures exercise.

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

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

\(^2\)Recall Section 1.3 for a discussion of how memory is measured.
Chapter 4

Skiplists

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

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

4.1 The Basic Structure

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

For an element, $x$, in a skiplist, we call the height of $x$ the largest value $r$ such that $x$ appears in $L_r$. Thus, for example, elements that only appear in $L_0$ have height 0. Notice

![Figure 4.1: A skiplist containing seven elements.](image-url)
Figure 4.2: The search path for the node containing 4 in a skiplist.

that the height of $x$ corresponds to the following experiment: Toss a coin repeatedly until the first time it comes up tails. How many times did it come up heads? The answer, not surprisingly, is that the expected height of a node is 1. (We expect to toss the coin twice before getting tails, but we don’t count the last toss.) The *height* of a skiplist is the height of its tallest node.

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

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

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

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

A space-efficient way to implement a Skiplist is to define a `Node`, $u$, as consisting of a data value, $x$, and an array, `next`, of pointers, where $u.next[i]$ points to $u$’s successor in the list $L_i$. In this way, the data, $x$, in a node is referenced only once, even though $x$ may appear in several lists.
The next two sections of this chapter discuss two different applications of skiplists. In each of these applications, \( L_0 \) stores the main structure (a list of elements or a sorted set of elements). The primary difference between these structures is in how a search path is navigated; in particular, they differ in how they decide if a search path should go down into \( L_{r-1} \) or go right within \( L_r \).

### 4.2 SkiplistSet: An Efficient SSet Implementation

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

```java
SkiplistSet findPredNode(T x) {
    Node<T> u = sentinel;
    int r = h;
    while (r >= 0) {
        while (u.next[r] != null && compare(u.next[r].x, x) < 0)
            u = u.next[r]; // go right in list r
        r--; // go down into list r-1
    }
    return u;
}

T find(T x) {
    Node<T> u = findPredNode(x);
    return u.next[0] == null ? null : u.next[0].x;
}
```

Following the search path for \( y \) is easy: when situated at some node, \( u \), in \( L_r \), we look right to \( u.next[r].x \). If \( x > u.next[r].x \), then we take a step to the right in \( L_r \), otherwise we move down into \( L_{r-1} \). Each step (right or down) in this search takes only constant time so, by Lemma 4.1, the expected running time of \( \text{find}(x) \) is \( O(\log n) \).

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

```java
int pickHeight() {
    int z = rand.nextInt();
}
```

\(^1\)This method does not exactly replicate the coin-tossing experiment since the value of \( k \) will always be less than the number of bits in an int. However, this will have negligible impact unless the number of elements in the structure is much greater than \( 2^{32} = 4294967296 \).
To implement the add\((x)\) method in a SkiplistSet we search for \(x\) and then splice \(x\) into a few lists \(L_0, \ldots, L_k\), where \(k\) is selected using the \texttt{pickHeight()} method. The easiest way to do this is to use an array, \texttt{stack}, that keeps track of the nodes at which the search path goes down from some list \(L_r\) into \(L_{r-1}\). More precisely, \texttt{stack}[r] is the node in \(L_r\) where the search path proceeded down into \(L_{r-1}\). The nodes that we modify to insert \(x\) are precisely the nodes \texttt{stack}[0], \ldots, \texttt{stack}[k]. The following code implements this algorithm for add\((x)\):

```java
boolean add(T x) {
    Node<T> u = sentinel;
    int r = h;
    int comp = 0;
    while (r >= 0) {
        while (u.next[r] != null && (comp = compare(u.next[r].x, x)) < 0)
            u = u.next[r];
        if (u.next[r] != null && comp == 0) return false;
        stack[r--] = u; // going down, store u
    }
    Node<T> w = new Node<T>(x, pickHeight());
    while (h < w.height())
        stack[++h] = sentinel; // increasing height of skiplist
    for (int i = 0; i < w.next.length; i++) {
        w.next[i] = stack[i].next[i];
        stack[i].next[i] = w;
    }
    n++;
    return true;
}
```

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

```java
boolean remove(T x) {
    boolean removed = false;
    ```
4.2. **SKIPLISTSET: AN EFFICIENT SSET IMPLEMENTATION**

![Diagram](image1)

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

![Diagram](image2)

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

```java
Node<T> u = sentinel;
int r = h;
int comp = 0;
while (r >= 0) {
    while (u.next[r] != null && (comp = compare(u.next[r].x, x)) < 0) {
        u = u.next[r];
    }
    if (u.next[r] != null && comp == 0) {
        removed = true;
        u.next[r] = u.next[r].next[r];
        if (u == sentinel && u.next[r] == null)
            h--; // skiplist height has gone down
    }
    r--;
}
if (removed) n--;
return removed;
```
4.2.1 Summary

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

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

4.3 SkiplistList: An Efficient Random-Access List Implementation

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

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

```java
class Node {
    T x;
    Node[] next;
    int[] length;
    Node(T ix, int h) {
        x = ix;
        next = (Node[])Array.newInstance(Node.class, h+1);
        length = new int[h+1];
    }
    int height() {
        return next.length - 1;
    }
}
```
The useful property of this definition of length is that, if we are currently at a node that is at position \( j \) in \( L_0 \) and we follow an edge of length \( \ell \), then we move to a node whose position, in \( L_0 \), is \( j + \ell \). In this way, while following a search path, we can keep track of the position, \( j \), of the current node in \( L_0 \). When at a node, \( u \), in \( L_r \), we go right if \( j \) plus the length of the edge \( u.\text{next}[r] \) is less than \( i \), otherwise we go down into \( L_{r-1} \).

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

```java
T get(int i) {
    if (i < 0 || i > n-1) throw new IndexOutOfBoundsException();
    return findPred(i).next[0].x;
}
T set(int i, T x) {
    if (i < 0 || i > n-1) throw new IndexOutOfBoundsException();
    Node u = findPred(i).next[0];
    T y = u.x;
    u.x = x;
    return y;
}
```

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

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

Note that, each time the search path goes down at a node, \( u \), in \( L_r \), the length of the edge \( u.\text{next}[r] \) increases by one, since we are adding an element below that edge at position \( i \). Splicing the node \( w \) between two nodes, \( u \) and \( z \), works as shown in Figure 4.7.

While following the search path we are already keeping track of the position, \( j \), of \( u \) in \( L_0 \).
Therefore, we know that the length of the edge from $u$ to $w$ is $i - j$. We can also deduce the length of the edge from $w$ to $z$ from the length, $\ell$, of the edge from $u$ to $z$. Therefore, we can splice in $w$ and update the lengths of the edges in constant time.

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

```java
void add(int i, T x) {
    if (i < 0 || i > n) throw new IndexOutOfBoundsException();
    Node w = new Node(x, pickHeight());
    if (w.height() > h)
        h = w.height();
    add(i, w);
}
```

```java
Node add(int i, Node w) {
    Node u = sentinel;
    int k = w.height();
    int r = h;
    int j = -1; // index of u
    while (r >= 0) {
        while (u.next[r] != null && j+u.length[r] < i) {
```
4.3. **SkiplistList: An Efficient Random-Access List Implementation**

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

![Figure 4.8: Removing an element from a SkiplistList.](image)

```java
T remove(int i) {
    if (i < 0 || i > n-1) throw new IndexOutOfBoundsException();
    T x = null;
    Node u = sentinel;
    int r = h;
    int j = -1; // index of node u
    while (r >= 0) {
        while (u.next[r] != null && u.length[r] < i) {
            j += u.length[r];
            u = u.next[r];
        }
        u.length[r]++;
        if (r <= k) {
            w.next[r] = u.next[r];
            u.next[r] = w;
            w.length[r] = u.length[r] - (i - j);
            u.length[r] = i - j;
        }
        r--;
    }
    n++;
    return u;
}
```
4.3.1 Summary

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

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

4.4 Analysis of Skiplists

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

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

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

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

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

\[ E[T] = E \left[ \sum_{i=1}^{\infty} I_i \right] \]
\[ = \sum_{i=1}^{\infty} E[I_i] \]
\[ = \sum_{i=1}^{\infty} \frac{1}{2^{i-1}} \]
\[ = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \]
\[ = 2 . \]

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

**Lemma 4.3.** The expected number of nodes in a skiplist containing \( n \) elements, not including occurrences of the sentinel, is \( 2n \).

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

\[ \sum_{r=0}^{\infty} \frac{n}{2^r} = n \left( 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \right) = 2n . \]

**Lemma 4.4.** The expected height of a skiplist containing \( n \) elements is at most \( \log n + 2 \).

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

\[ I_r = \begin{cases} 0 & \text{if } L_r \text{ is empty} \\ 1 & \text{if } L_r \text{ is non-empty} \end{cases} \]

The height \( h \) of the skiplist is then given by

\[ h = \sum_{i=1}^{\infty} I_r . \]

Note that \( I_r \) is never more than the length, \( |L_r| \), of \( L_r \), so

\[ E[I_r] \leq E[|L_r|] = \frac{n}{2^r} . \]
Therefore, we have

\[
E[h] = E \left[ \sum_{r=1}^{\infty} I_r \right] \\
= \sum_{r=1}^{\infty} E[I_r] \\
= \sum_{r=1}^{\lfloor \log n \rfloor} E[I_r] + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} E[I_r] \\
\leq \sum_{r=1}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} n/2^r \\
\leq \log n + \sum_{r=0}^{\infty} 1/2^r \\
= \log n + 2 .
\]

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

**Proof.** By Lemma 4.3, the expected number of nodes, not including the sentinel is \( 2n \). The number of occurrences of the sentinel is equal to the height, \( h \), of the skiplist so, by Lemma 4.4 the expected number of occurrences of the sentinel is at most \( \log n + 1 = O(\log n) \). □

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

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

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

Let \( S_r \) denote the number of steps the forward search path takes at level \( r \) that go to the right. We have just argued that \( E[S_r] \leq 1 \). Furthermore, \( S_r \leq |L_r| \), since we can’t take more steps in \( L_r \) than the length of \( L_r \), so

\[
E[S_r] \leq E[|L_r|] = n/2^r .
\]

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

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

$$E[S] = E \left[ h + \sum_{r=0}^{\infty} S_r \right]$$

$$= E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} E[S_r] + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} E[S_r]$$

$$\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} n/2^r$$

$$\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r$$

$$\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r$$

$$\leq E[h] + \log n + 3$$

$$\leq 2 \log n + 5 .$$

The following theorem summarizes the results in this section

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

### 4.5 Discussion and Exercises

Skiplists were introduced by Pugh [43] who also presented a number of applications of skiplists [42]. Since then they have been studied extensively. Several researchers have done very precise analysis of the expected length and variance in length of the search path for the $i$th element in a skiplist [31, 30, 40]. Deterministic versions [35], biased versions [5, 19], and self-adjusting versions [8] of skiplists have all been developed. Skiplist implementations have been written for various languages and frameworks and have seen use in open-source database systems [49, 44] A variant of skiplists is used in the HP-UX operating system kernel’s process management structures [29]. Skiplists are even part of the Java 1.6 API [37].

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

**Exercise 4.2.** Suppose that, instead of promoting an element from $L_{i-1}$ into $L_i$ based on a coin toss, we promote it with some probability $p$, $0 < p < 1$. Show that the expected length of the search path in this case is at most $(1/p) \log_{1/p} n + O(1)$. What is the value of $p$ that minimizes this expression? What is the expected height of the skiplist? What is the expected number of nodes in the skiplist?
Exercise 4.3. Design and implement a \texttt{find(x)} method for \texttt{SkiplistSet} that avoids \textit{locally-redundant comparisons}; these are comparisons that have already been done and occur because \( u.next[r] = u.next[r-1] \). Analyze the expected number of comparisons done by your modified \texttt{find(x)} method.

Exercise 4.4. Design and implement a version of a skiplist that implements the \texttt{SSet} interface, but also allows fast access to elements by rank. That is, it also supports the function \texttt{get(i)}, which returns the element whose rank is \( i \) in \( O(\log n) \) expected time. (The rank of an element \( x \) in a \texttt{SSet} is the number of elements in the \texttt{SSet} that are less than \( x \).)

Exercise 4.5. Using the ideas from the space-efficient linked-list, \texttt{SEList}, design and implement a space-efficient \texttt{SSet}, \texttt{SESSet}. Do this by storing the data, in order, in an \texttt{SEList} and then storing the blocks of this \texttt{SEList} in an \texttt{SSet}. If the original \texttt{SSet} implementation uses \( O(n) \) space to store \( n \) elements, then the \texttt{SESSet} will use enough space for \( n \) elements plus \( O(n/b + b) \) wasted space.
Chapter 5

Hash Tables

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

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

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

5.1 HashTable: Hashing with Chaining

A HashTable 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:

```java
    List<T>[] t;
    int n;
```

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

\[
    n \leq t.length
\]

so that the average number of elements stored in one of these lists is \( n/t.length \leq 1 \).

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

```java
    boolean add(T x) {
        if (n+1 > t.length)
            grow();
        
```
Growing the table, if necessary, involves doubling the length of t and reinserting all elements into the new table. This is exactly the same strategy used in the implementation of ArrayStack and the same result applies: The cost of growing is only constant when amortized over a sequence of insertions (see Lemma 2.1 on page 14).

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

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

```java
HashTable

    boolean remove(Object x) {
        return removeOne(x) != null;
    }

HashTable

    T removeOne(Object x) {
        Iterator<T> it = t[hash(x)].iterator();
        while (it.hasNext()) {
            T y = it.next();
            if (y.equals(x)) {
                it.remove();
                n--;
                return y;
            }
        }
        return null;
    }
```

This takes $O(n_{hash(x)})$ 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[hash(x)]:

```java
HashTable

    T find(Object x) {
        for (T y : t[hash(x)])
            if (y.equals(x))
                return y;
        return null;
    }
```
5.1. HASHTABLE: HASHING WITH CHAINING

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 $t.length$ lists, so that the expected size of the list $t[\text{hash}(x)]$ is $O(n/t.length) = O(1)$. On the other hand, a bad hash function will hash all values (including $x$) to the same table location, in which case the size of the list $t[\text{hash}(x)]$ will be $n$. In the next section we describe a good hash function.

5.1.1 Multiplicative Hashing

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

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

$$\text{hash}(x) = \left(\left(\lfloor z \cdot x \rfloor \mod 2^w\right) \text{ div } 2^{w-d}\right).$$

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. (See Figure 5.1.) 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 $w-d$). In this way, the code that implements the above formula is simpler than the formula itself:

```java
int hash(Object x) {
    return (z * x.hashCode()) >>> (w-d);
}
```

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

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

With Lemma 5.1, the performance of $\text{remove}(x)$, and $\text{find}(x)$ are easy to analyze:
Lemma 5.2. For any data value \( x \), the expected length of the list \( t[\text{hash}(x)] \) 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

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

and notice that, by Lemma 5.1, \( \mathbb{E}[I_y] \leq 2/2^d = 2/t.length \). The expected length of the list \( t[\text{hash}(x)] \) is given by

\[
\mathbb{E}[t[\text{hash}(x)].\text{size}()] = \mathbb{E} \left[ n_x + \sum_{y \in S} I_y \right] \\
= n_x + \sum_{y \in S} \mathbb{E}[I_y] \\
\leq n_x + \sum_{y \in S} 2/t.length \\
\leq n_x + \sum_{y \in S} 2/n \\
\leq n_x + (n - n_x)2/n \\
\leq n_x + 2,
\]

as required.

Now, we want to prove Lemma 5.1, but first we need a result from number theory. In the following proof, we use the notation \((b_r, \ldots, b_0)_2\) to denote \( \sum_{i=0}^{r} b_i 2^i \), where each \( b_i \) is a bit either 0 or 1, i.e., the integer whose binary representation is given by \( b_r, \ldots, b_0 \). We use * to denote a bit of unknown value.

Lemma 5.3. 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 \( zq \mod 2^w = i \).

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 \( zq \mod 2^w = i \).

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

\[
zq \mod 2^w = z'q \mod 2^w = i
\]

So

\[
(z - z')q \mod 2^w = 0
\]

But this means that

\[
(z - z')q = k 2^w \tag{5.1}
\]
for some integer \( k \). Thinking in terms of binary numbers, we have
\[
(z - z')q = k \cdot (1, 0, \ldots, 0)_2,
\]
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:
\[
q = (\ast, \ldots, \ast, 1)_2.
\]
Since \(|z - z'| < 2^w\), \( z - z' \) has fewer than \( w \) trailing 0’s in its binary representation:
\[
z - z' = (\ast, \ldots, \ast, 1, 0, \ldots, 0)_{< w}.
\]
Therefore, the product \((z - z')q\) has fewer than \( w \) trailing 0’s in its binary representation:
\[
(z - z')q = (\ast, \cdots, \ast, 1, 0, \ldots, 0)_{< w}.
\]
Therefore \((z - z')q\) cannot satisfy (5.1), yielding a contradiction and completing the proof. \( \square \)

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

\textbf{Proof of Lemma 5.1.} First we note that the condition \( \text{hash}(x) = \text{hash}(y) \) is equivalent
to the statement “the highest-order \( d \) bits of \( zx \mod 2^w \) and the highest-order \( d \) bits of
\( zy \mod 2^w \) are the same.” A necessary condition of that statement is that the highest-order
\( d \) bits in the binary representation of \( z(x - y) \mod 2^w \) are either all 0’s or all 1’s. That is,
\[
z(x - y) \mod 2^w = (0, \ldots, 0, \ast, \ldots, \ast)_{d \ w-d}
\] (5.2)
when \( zx \mod 2^w > zy \mod 2^w \) or
\[
z(x - y) \mod 2^w = (1, \ldots, 1, \ast, \ldots, \ast)_{d \ w-d}
\] (5.3)
when \( zx \mod 2^w < zy \mod 2^w \). Therefore, we only have to bound the probability that
\( z(x - y) \mod 2^w \) looks like (5.2) or (5.3).

Let \( q \) be the unique odd integer such that \((x - y) \mod 2^w = q2^r \) for some integer
\( r \geq 0 \). By Lemma 5.3, the binary representation of \( zq \mod 2^w \) has \( w - 1 \) random bits,
followed by a 1:
\[
zq \mod 2^w = (b_{w-1}, \ldots, b_1, 1)_{\ w-1}
\]
Therefore, the binary representation of \(z(x - y) \mod 2^w = zq2^r \mod 2^w\) has \(w - r - 1\) random bits, followed by a 1, followed by \(r\) 0's:

\[
z(x - y) \mod 2^w = zq2^r \mod 2^w = (b_{u-r-1}, \ldots, b_1, 0, 0, \ldots, 0)_2
\]

We can now finish the proof: If \(r > w - d\), then the \(d\) higher order bits of \(z(x - y) \mod 2^w\) contain both 0's and 1's, so the probability that \(z(x - y) \mod 2^w\) looks like (5.2) or (5.3) is 0. If \(r = w - d\), then the probability of looking like (5.3) 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 \(b_{u-r-1}, \ldots, b_{u-r-d} = 0, \ldots, 0\) or \(b_{u-r-1}, \ldots, b_{u-r-d} = 1, \ldots, 1\). 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.

\[\square\]

### 5.1.2 Summary

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

**Theorem 5.1.** A `HashTable` implements the `USet` interface. Ignoring the cost of calls to `grow()`, a `HashTable` supports the operations `add(x)`, `remove(x)`, and `find(x)` in \(O(1)\) expected time per operation. Furthermore, beginning with an empty `HashTable`, any sequence of \(m\) `add(x)` and `remove(x)` operations results in a total of \(O(m)\) time spent during all calls to `grow()`.

### 5.2 Hash Codes

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

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

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

#### 5.2.1 Hash Codes for Primitive Data Types

Small primitive data types like `char`, `byte`, `int`, and `float` are usually easy to find hash codes for. These data types always have a binary representation and this binary representation usually consists of \(w\) or fewer bits. (For example, in Java, `byte` is an 8-bit type and `float` is a 32-bit type.) In these cases, we just treat these bits as the representation of
5.2. HASH CODES

an integer in the range \{0, \ldots, 2^w - 1\}. If two values are different, they get different hash codes. If they are the same, they get the same hash code.

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

5.2.2 Hash Codes for Compound Objects

For a compound object, we want to create a hash code by combining the individual hash codes of the object’s constituent parts. This is not as easy as it sounds. Although one can find many hacks for this (for example, combining the hash codes with bitwise exclusive-or operations), many of these hacks turn out to be easy to foil (see Exercises 5.2–5.4).

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

\[
h(x_0, \ldots, x_{r-1}) = \left(\left(\sum_{i=0}^{r-1} z_i x_i \mod 2^{2w}\right) \div 2^w\right).
\]

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 5.1.1 to take the \(2^w\)-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 3 parts \(x_0, x_1, \text{ and } x_2:\)

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

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

**Theorem 5.2.** 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

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

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

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

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

$$z_i(x_i - y_i) \mod 2^{2^w} = t$$  \hspace{1cm} (5.4)

where

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

If we assume, without loss of generality that $x_i > y_i$, then (5.4) becomes

$$z_i(x_i - y_i) = t,$$  \hspace{1cm} (5.5)

since each of $z_i$ and $(x_i - y_i)$ is at most $2^w - 1$, so their product is at most $2^{2^w} - 2^{w+1} + 1 < 2^{2^w} - 1$. By assumption, $x_i - y_i \neq 0$, so (5.5) has at most one solution in $z_i$. Therefore, since $z_i$ and $t$ are independent ($z_0, \ldots, z_{r-1}$ 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 5.2, if $h'(x_0, \ldots, x_{r-1}) \neq h'(y_0, \ldots, y_{r-1})$, then $\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq 2/2^w$.

To summarize,

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

and this completes the proof.

### 5.2.3 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 5.2 are no longer independent.

A more rigorous approach is to base our hash codes on polynomials over prime fields. This method is based on the following theorem, which says that polynomials over prime fields behave pretty-much like usual polynomials:
Theorem 5.3. 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 \( f(z) \mod p = 0 \) has at most \( r - 1 \) solutions for \( z \in \{0, \ldots, p - 1\} \).

To use Theorem 5.3, 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

\[
h(x_0, \ldots, x_{r-1}) = (x_0z^0 + \cdots + x_{r-1}z^{r-1} + (p - 1)z^r) \mod p.
\]

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 5.4. 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 \( i \in \{0, \ldots, r-1\} \). Then

\[
\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq (r - 1)/p.
\]

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

\[
((x_0 - y_0)z^0 + \cdots + (x_{r-1} - y_{r-1})z^{r-1}) \mod p = 0. \tag{5.6}
\]

Since \( x_i \neq y_i \), this polynomial is non-trivial. Therefore, by Theorem 5.3, 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 \((r - 1)/p\). \(\square\)

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

\[
x_0, \ldots, x_{r-1}, p - 1, 0, 0, \ldots.
\]

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

\[
\left( \sum_{i=0}^{i=r-1} (x_i - y_i)z^i + (x_r - p - 1)z^{r'} + \sum_{i=r'+1}^{i=r-1} x_i z^i + (p - 1)z^r \right) \mod p = 0,
\]

which, by Theorem 5.3, has at most \( r \) solutions in \( z \). This combined with Theorem 5.4 suffice to prove the following more general theorem:

Theorem 5.5. Let \( p > 2^w + 1 \) be a prime, let \( x_0, \ldots, x_{r-1} \) and \( y_0, \ldots, y_{r'-1} \) be distinct sequences of \( w \)-bit integers in \( \{0, \ldots, 2^w - 1\} \). Then

\[
\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r'-1})\} \leq \max\{r, r'\}/p.
\]
The following example code shows how this hash function is applied to an object that contains an array, $x$, of values:

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

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

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

rather than the $r/(2^{32} - 5)$ specified in Theorem 5.5.

5.3 Discussion and Exercises

Hash tables and hash codes are a enormous and active area of research that is just touched upon in this chapter. The online Bibliography on Hashing [6] contains nearly 2000 entries.

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

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

Yet another category of hash table implementations are the so-called perfect hashing methods. These are methods in which $\text{find}(x)$ operations take $O(1)$ time in the worst-case.
5.3. DISCUSSION AND EXERCISES

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 [21] and cuckoo hashing [39].

The hash functions presented in this chapter are probably among the most practical currently known methods that can be proven to work well for any set of data. Other provably good methods date back to the pioneering work of Carter and Wegman who introduced the notion of universal hashing and described several hash functions for different scenarios [10].

The idea of multiplicative hashing is very old and seems to be part of the hashing folklore [34, Section 6.4]. However, the idea of choosing the multiplier z to be a random odd number, and the analysis in Section 5.1.1 is due to Dietzfelbinger et al. [17]. This version of multiplicative hashing is one of the simplest, but it’s collision probability of \(2/2^d\) is a factor of 2 larger than what one could expect with a random function from \(2^w \to 2^d\). The multiply-add hashing method uses the function

\[
h(x) = ((zx + b) \mod 2^w) \div 2^{2w-d}
\]

where \(z\) and \(b\) are each randomly chosen from \(\{0, \ldots, 2^w - 1\}\). Multiply-add hashing has a collision probability of only \(1/2^d\) [15], 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 [7] is the function

\[
h(x_0, \ldots, x_{r-1}) = \left( \sum_{i=0}^{r/2-1} ((z_{2i} + a_{2i}) \mod 2^w)((z_{2i+1} + a_{2i+1}) \mod 2^w) \right) \mod 2^{2w}
\]

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 5.2.2 requires \(r\) multiplications. (The mod operations occur implicitly by using \(w\) and \(2w\)-bit arithmetic for the additions and multiplications, respectively.)

The method from Section 5.2.3 of using polynomials over prime fields to hash variable-length arrays and strings is due to Dietzfelbinger et al. [16]. It is, unfortunately, not very fast. This is due to its use of the mod operator which relies on a costly machine instruction. Some variants of this method choose the prime \(p\) to be one of the form \(2^w - 1\), in which case the mod operator can be replaced with addition (+) and bitwise-and (\&) operations [33, 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 \([r/c]\) hash codes.

**Exercise 5.1.** Prove that the bound \(2/2^2\) in Lemma 5.1 is the best possible by showing that, if \(x = 2^{w-d-2}\) and \(y = 3x\), then \(\Pr\{\text{hash}(x) = \text{hash}(y)\} = 2/2^d\). (Hint look at the binary representations of \(zx\) and \(z3x\) and use the fact that \(z3x = zx + 2zx\).)

**Exercise 5.2.** 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 5.3. 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 5.4. Suppose you have an object made up of two \( w \)-bit integers \( x \) and \( y \). Suppose that the hash code for your object is defined by some deterministic function \( h(x, y) \). Prove that there exists a large set of objects that have the same hash code.

Exercise 5.5. Let \( p = 2^w - 1 \) for some positive integer \( w \). Explain why, for a positive integer \( x \)

\[
(x \mod 2^w) + (x \div 2^w) \equiv x \mod (2^w - 1).
\]

(This gives an algorithm for computing \( x \mod (2^w - 1) \) by repeatedly setting

\[
x = x \& ((1 \ll w) - 1) + x >> w
\]

until \( x \leq 2^w - 1 \).)
Chapter 6

Binary Trees

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

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

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

Binary trees are so important that a terminology has developed around them: The depth of a node, \( u \), in a binary tree is the length of the path from \( u \) to the root of the tree. If a node, \( w \), is on the path from \( u \) to \( r \) then \( w \) is called an ancestor of \( u \) and \( u \) a descendant of \( w \). The subtree of a node, \( u \), is the binary tree that is rooted at \( u \) and contains all of \( u \)'s descendants. The height of a node, \( u \), is the length of the longest path from \( u \) to one of its descendants. The height of a tree is the height of its root. A node, \( u \), is a leaf if it has no children.

We sometimes think of the tree as being augmented with external nodes. Any node

![Figure 6.1: The parent, left child, and right child of the node u in a BinaryTree.](image-url)
that does not have a left child has an external node as its left child and any node that does not have a right child has an external node as its right child (see Figure 6.2.b). It is easy to verify, by induction, that a binary tree having \( n \geq 1 \) real nodes has \( n + 1 \) external nodes.

### 6.1 BinaryTree: A Basic Binary Tree

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

```plaintext
BinaryTreeNode

Node left;
Node right;
Node parent;
```

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

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

```plaintext
TreeNode

Node r;
```

We can compute the depth of a node, \( u \), by counting the number of steps on the path from \( u \) to the root:

```plaintext
int depth(Node u) {
    int d = 0;
    while (u != r) {
        u = u.parent;
        d++;
    }
    return d;
}
```
6.1. **BINARY TREE: A BASIC BINARY TREE**

### 6.1.1 Recursive Algorithms

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

```c
int size(Node u) {
    if (u == nil)
        return 0;
    return 1 + size(u.left) + size(u.right);
}
```

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

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

### 6.1.2 Traversing Binary Trees

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

```c
void traverse(Node u) {
    if (u == nil) return;
    traverse(u.left);
    traverse(u.right);
}
```

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

Luckily, traversing a binary tree can be done without recursion. This is done using an algorithm that uses where it came from to decide where it will go next. See Figure 6.3. If we arrive at a node \( u \) from \( u.parent \), then the next thing to do is to visit \( u.left \). If we arrive at \( u \) from \( u.left \), then the next thing to do is to visit \( u.right \). If we arrive at \( u \) from \( u.right \), then we are done visiting \( u \)'s subtree, so we return to \( u.parent \). The following code implements this idea, with code included for handling the cases where any of \( u.left \), \( u.right \), or \( u.parent \) is \( nil \):
void traverse2() {
    Node u = r, prev = nil, next;
    while (u != nil) {
        if (prev == u.parent) {
            if (u.left != nil) next = u.left;
            else if (u.right != nil) next = u.right;
            else next = u.parent;
        } else if (prev == u.left) {
            if (u.right != nil) next = u.right;
            else next = u.parent;
        } else {
            next = u.parent;
        }
        prev = u;
        u = next;
    }
}

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

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

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

A special kind of traversal that does not fit the pattern of the above functions is
the breadth-first traversal. In a breadth-first traversal, the nodes are visited level-by-level
starting at the root and working our way down, visiting the nodes at each level from left
to right. This is similar to the way we would read a page of English text. (See Figure 6.4.)
This is implemented using a queue, q, that initially contains only the root, r. At each step,
we extract the next node, u, from q, process u and add u.left and u.right (if they are
non-nil) to q:

```java
void bfTraverse() {
    Queue<Node> q = new LinkedList<Node>();
    q.add(r);
    while (!q.isEmpty()) {
        Node u = q.remove();
        if (u.left != nil) q.add(u.left);
        if (u.right != nil) q.add(u.left);
    }
}
```

### BinarySearchTree: An Unbalanced Binary Search Tree

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

#### 6.2.1 Searching

The binary search tree property is extremely useful because it allows us to quickly locate a
value, x, in a binary search tree. To do this we start searching for x at the root, r. When
Figure 6.4: During a breadth-first traversal, the nodes of a binary tree are visited level-by-level, and left-to-right within each level.

Figure 6.5: A binary search tree.
examine a node, \( u \), there are three cases:

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

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

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

Two examples of searches in a binary search tree are shown in Figure 6.6. As the second example shows, even if we don’t find \( x \) in the tree, we still gain some valuable information. If we look at the last node, \( u \), at which Case 1 occured, 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 occured contains the largest value in the tree that is less than than \( x \). Therefore, by keeping track of the last node, \( z \), at which Case 1 occurs, a BinarySearchTree can implement the \( \text{find}(x) \) operation that returns the smallest value stored in the tree that is greater than or equal to \( x \):

```java
BinarySearchTree
T find(T x) {
    Node w = r, z = nil;
    while (w != nil) {
        int comp = compare(x, w.x);
        if (comp < 0) {
            z = w;
            w = w.left;
        } else if (comp > 0) {
            w = w.right;
        } else {
            return w.x;
        }
    }
    return null;
}
```
Figure 6.6: An example of (a) a successful search (for 6) and (b) an unsuccessful search (for 10) in a binary search tree.

6.2.2 Inserting

To add a new value, \( x \), to a `BinarySearchTree`, we first search for \( x \). If we find it, then there is no need to insert it. Otherwise, we store \( x \) at a leaf child of the last node, \( p \), encountered during the search for \( x \). Whether the new node is the left or right child of \( p \) depends on the result of comparing \( x \) and \( p.x \).
An example is shown in Figure 6.7. The most time-consuming part of this process is the initial search for \( x \), which takes time proportional to the height of the newly added node \( u \).

In the worst case, this is equal to the height of the BinarySearchTree.

### 6.2.3 Deleting

Deleting a value stored in a node, \( u \), of a BinarySearchTree is a little more difficult. If \( u \) is a leaf, then we can just detach \( u \) from its parent. Even better: If \( u \) has only one child, then we can splice \( u \) from the tree by having \( u \).parent adopt \( u \)'s child:
Figure 6.8: Deleting a leaf (6) or a node with only one child (9) is easy.

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

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

```
void remove(Node u) {
    if (u.left == nil || u.right == nil) {
```
6.2. **BINARYSEARCHTREE: AN UNBALANCED BINARY SEARCH TREE**

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

```cpp
splice(u);
} else {
    Node w = u.right;
    while (w.left != nil)
        w = w.left;
    u.x = w.x;
    splice(w);
}
```

### 6.2.4 Summary

The `find(x)`, `add(x)`, and `remove(x)` operations in a `BinarySearchTree` each involve following a path from the root of the tree to some node in the tree. Without knowing more about the shape of the tree it is difficult to say much about the length of this path, except that it is less than n, the number of nodes in the tree. The following (unimpressive) theorem summarizes the performance of the `BinarySearchTree` data structure:

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

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

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

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

Binary search trees appear to have been discovered independently by several groups in the 1950s [34, Section 6.2.2]. Further references to specific kinds of binary search trees are provided in subsequent chapters.

Exercise 6.1. Write a non-recursive variant of the size2() method, size(u), that computes the size of the subtree rooted at node u.

Exercise 6.2. Write a non-recursive method, height2(u), that computes the height of node u in a binary search tree.

Exercise 6.3. A pre-order traversal of a binary tree is a traversal that visits each node, u, before any of its children. An in-order traversal visits u after visiting all the nodes in u’s left subtree but before visiting any of the nodes in u’s right subtree. A post-order traversal visits u only after visiting all other nodes in u’s subtree.

Write non-recursive functions nextPreOrder(u), nextInOrder(u), and nextPostOrder(u) that return the node that follows u in a pre-order, in-order, or post-order traversal, respectively.

Exercise 6.4. Describe a sequence of n operations on an initially empty BinarySearchTree that results in a tree of height n – 1.

Exercise 6.5. Design and implement a version of BinarySearchTree in which each node, u, maintains values u.size (the size of the subtree rooted at u), u.depth (the depth of u), and u.height (the height of the subtree rooted at u).

These values should be maintained, even during the add(x) and remove(x) operations, but this should not increase the cost of these operations by more than a constant factor.
In this chapter, we present a binary search tree structure that uses randomization to achieve $O(\log n)$ expected time for all operations.

### 7.1 Random Binary Search Trees

Consider the two binary search trees shown in Figure 7.1. The one on the left is a list and the other is a perfectly balanced binary search tree. The one on the left has height $n - 1 = 14$ and the one on the right has height three.

Imagine how these two trees could have been constructed. The one on the left occurs if we start with an empty `BinarySearchTree` and add the sequence

$$\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 \rangle.$$  

No other sequence of additions will create this tree (as you can prove by induction on $n$). On the other hand, the tree on the right can be created by the sequence

$$\langle 7, 3, 11, 1, 5, 9, 13, 0, 2, 4, 6, 8, 10, 12, 14 \rangle.$$  

Other sequences work as well, including

$$\langle 7, 3, 1, 5, 0, 2, 4, 6, 11, 9, 13, 8, 10, 12, 14 \rangle$$  

and

$$\langle 7, 3, 1, 11, 5, 0, 2, 4, 6, 9, 13, 8, 10, 12, 14 \rangle.$$  

In fact, there are 21,964,800 addition sequences that generate the tree on the right and only one that generates the tree on the left.

The above example gives some anecdotal evidence that, if we choose a random permutation of $0, \ldots, 14$, and add it into a binary search tree then we are more likely to get a very balanced tree (the right hand of Figure 7.1) than we are to get a very unbalanced tree (the left hand of Figure 7.1).

We can formalize this notion by studying random binary search trees. A *random binary search tree* of size $n$ is obtained in the following way: Take a random permutation $x_0, \ldots, x_{n-1}$ of $0, \ldots, n - 1$ and add its elements, one by one, into a `BinarySearchTree`.

Note that the values $0, \ldots, n - 1$ could be replaced by any ordered set of $n$ elements without changing any of the properties of the random binary search tree. The element $x \in \{0, \ldots, n - 1\}$ is simply standing in for the element of rank $x$ in an ordered set of size $n$.  

89
Before we can present our main result about random binary search trees, we must take some time for a short digression to discuss a type of number that comes up frequently when studying randomized structures. For a non-negative integer, $k$, the $k$-th harmonic number, denoted $H_k$, is defined as

$$H_k = 1 + 1/2 + 1/3 + \cdots + 1/k.$$ 

The harmonic number $H_k$ has no simple closed form, but it is very closely related to the natural logarithm of $k$. In particular,

$$\ln k \leq H_k \leq \ln k + 1.$$ 

Readers who have studied calculus might notice that this is because the integral $\int_1^k (1/x) \, dx = \ln k$. Keeping in mind that an integral can be interpreted as the area between a curve and the $x$-axis, the value of $H_k$ can be lower-bounded by the integral $\int_1^k (1/x) \, dx$ and upper-bounded by $1 + \int_1^k (1/x) \, dx$. (See Figure 7.2 for an explanation.)

**Lemma 7.1.** In a random binary search tree of size $n$, the following statements hold:

1. For any $x \in \{0, \ldots, n - 1\}$, the expected length of the search path for $x$ is $H_{x+1} + H_{n-x} - O(1)$.

2. For any $x \in (-1, n) \setminus \{0, \ldots, n - 1\}$, the expected length of the search path for $x$ is $H_{|x|} + H_{n-|x|}$.

We will prove Lemma 7.1 in the next section. For now, we will consider what the two parts of Lemma 7.1 tell us. The first part tells us that if we search for an element in a tree of size $n$, then the expected length of the search path is at most $2 \ln n + O(1)$. The second part tells us the same thing about searching for a value not stored in the tree. When we compare the two parts of the lemma, we see that it is only slightly faster to search for something that is in a tree compared to something that is not in a tree.

---

1 The expressions $x+1$ and $n-x$ can be interpreted respectively as the number of elements in the tree less than or equal to $x$ and the number of elements in the tree greater than or equal to $x$. 

---

Figure 7.1: Two binary search trees containing the integers 0, \ldots, 14.
7.1. RANDOM BINARY SEARCH TREES

7.1.1 Proof of Lemma 7.1

The key observation needed to prove Lemma 7.1 is the following: The search path for a value \( x \) in the open interval \((-1, n)\) in a random binary search tree, \( T \), contains the node with key \( i < x \) if and only if, in the random permutation used to create \( T \), \( i \) appears before any of \( \{i + 1, i + 2, \ldots, \lfloor x \rfloor\} \).

To see this, refer to Figure 7.3 and notice that, until some value in \( \{i, i + 1, \ldots, \lfloor x \rfloor\} \) is added, the search paths for each value in the open interval \((i - 1, \lfloor x \rfloor + 1)\) are identical. (Remember that for two search values to have different search paths, there must be some element in the tree that compares differently with them.) Let \( j \) be the first element in \( \{i, i + 1, \ldots, \lfloor x \rfloor\} \) to appear in the random permutation. Notice that \( j \) is now and will always be on the search path for \( x \). If \( j \neq i \) then the node \( u_j \) containing \( j \) is created before the node \( u_i \) that contains \( i \). Later, when \( i \) is added, it will be added to the subtree rooted at \( u_j.left \), since \( i < j \). On the other hand, the search path for \( x \) will never visit this subtree because it will proceed to \( u_j.right \) after visiting \( u_j \).

Similarly, for \( i > x \), \( i \) appears in the search path for \( x \) if and only if \( i \) appears before any of \( \{\lceil x \rceil, \lfloor x \rfloor + 1, \ldots, i - 1\} \) in the random permutation used to create \( T \).

From probability theory, the subsequence containing any independently chosen subset of a random permutation is a random permutation of the subset. Each element, then, in the subsets \( \{i, i + 1, \ldots, \lfloor x \rfloor\} \) and \( \{\lceil x \rceil, \lfloor x \rfloor + 1, \ldots, i - 1\} \) is equally likely to appear before any other in its subset in the random permutation used to create \( T \). So we have

\[
\Pr\{i \text{ is on the search path for } x\} = \begin{cases} 
1/(\lceil x \rceil - i + 1) & \text{if } i < x \\
1/((i - \lfloor x \rfloor) + 1) & \text{if } i > x
\end{cases}
\]

With this observation, the proof of Lemma 7.1 involves some simple calculations with harmonic numbers:

\textit{Proof of Lemma 7.1.} Let \( I_i \) be the indicator random variable that is equal to one when \( i \) appears on the search path for \( x \) and zero otherwise. Then the length of the search path is
Figure 7.3: The value \( i < x \) is on the search path for \( x \) if and only if \( i \) is the first element among \( \{i, i + 1, \ldots, \lfloor x \rfloor\} \) added to the tree.

The corresponding calculations for a search value \( x \in (n - x - 1) \setminus \{0, \ldots, n - 1\} \) are almost identical.

### 7.1.2 Summary

The following theorem summarizes the performance of a random binary search tree:
7.2. Treap: A Randomized Binary Search Tree

Theorem 7.1. A random binary search tree can be constructed in \( O(n \log n) \) time. In a random binary search tree, the \texttt{find}(x) operation takes \( O(\log n) \) expected time.

7.2 Treap: A Randomized Binary Search Tree

The problem with random binary search trees is, of course, that they are not dynamic. They don't support the \texttt{add}(x) or \texttt{remove}(x) operations needed to implement the \texttt{SSet} interface. In this section we describe a data structure called a Treap that uses Lemma 7.1 to implement the \texttt{SSet} interface.

A node in a Treap is like a node in a BinarySearchTree in that it has a data value, \( x \), but it also contains a unique numerical \textit{priority}, \( p \), that is assigned at random:

```java
class Node<T> extends BSTNode<Node<T>,T> {
    int p;
}
```

In addition to being a binary search tree, the nodes in a Treap also obey the \textit{heap property}: At every node except the root, \( u.p \) < \( u.p \). That is, each node has a priority smaller than that of its two children. An example is shown in Figure 7.4.

The heap and binary search tree conditions together ensure that, once the key (\( x \)) and priority (\( p \)) for each node are defined, the shape of the Treap is completely determined. The heap property tells us that the node with minimum priority has to be the root, \( r \), of the Treap. The binary search tree property tells us that all nodes with keys smaller than \( r.x \) are stored in the subtree rooted at \( r.left \) and all nodes with keys larger than \( r.x \) are stored in the subtree rooted at \( r.right \).

The important point about the priority values in a Treap is that they are unique and assigned at random. Because of this, there are two equivalent ways we can think about a Treap. As defined above, a Treap obeys the heap and binary search tree properties. Alternatively, we can think of a Treap as a BinarySearchTree whose nodes were added.
in increasing order of priority. For example, the Treap in Figure 7.4 can be obtained by adding the sequence of \((x, p)\) values
\[
\langle (3, 1), (1, 6), (0, 9), (5, 11), (9, 12), (4, 14), (7, 22), (6, 42), (8, 49), (2, 99) \rangle
\]
into a BinarySearchTree.

Since the priorities are chosen randomly, this is equivalent to taking a random permutation of the keys — in this case the permutation is
\[
\langle 3, 1, 0, 5, 9, 4, 7, 6, 8, 2 \rangle
\]
— and adding these to a BinarySearchTree. But this means that the shape of a treap is identical to that of a random binary search tree. In particular, if we replace each key \(x\) by its rank,\(^2\) then Lemma 7.1 applies. Restating Lemma 7.1 in terms of Treaps, we have:

**Lemma 7.2.** In a Treap that stores a set \(S\) of \(n\) keys, the following statements hold:

1. For any \(x \in S\), the expected length of the search path for \(x\) is \(H_{r(x)+1} + H_{n-r(x)} - O(1)\).
2. For any \(x \not\in S\), the expected length of the search path for \(x\) is \(H_{r(x)} + H_{n-r(x)}\).

Here, \(r(x)\) denotes the rank of \(x\) in the set \(S \cup \{x\}\).

Lemma 7.2 tells us that Treaps can implement the \texttt{find}(x) operation efficiently. However, the real benefit of a Treap is that it can support the \texttt{add}(x) and \texttt{delete}(x) operations. To do this, it needs to perform rotations in order to maintain the heap property. Refer to Figure 7.5. A rotation in a binary search tree is a local modification that takes a parent \(u\) of a node \(w\) and makes \(w\) the parent of \(u\), while preserving the binary search tree property. Rotations come in two flavours: left or right depending on whether \(w\) is a right or left child of \(u\), respectively.

The code that implements this has to handle these two possibilities and be careful of a boundary case (when \(u\) is the root) so the actual code is a little longer than Figure 7.5 would lead a reader to believe:

\(^2\)The rank of an element \(x\) in a set \(S\) of elements is the number of elements in \(S\) that are less than \(x\).
7.2. **TREAP: A RANDOMIZED BINARY SEARCH TREE**

```java
void rotateLeft(Node u) {
    Node w = u.right;
    w.parent = u.parent;
    if (w.parent != nil) {
        if (w.parent.left == u) {
            w.parent.left = w;
        } else {
            w.parent.right = w;
        }
    }
    u.right = w.left;
    if (u.right != nil) {
        u.right.parent = u;
    }
    u.parent = w;
    w.left = u;
    if (u == r) r = w;
}

void rotateRight(Node u) {
    Node w = u.left;
    w.parent = u.parent;
    if (w.parent != nil) {
        if (w.parent.left == u) {
            w.parent.left = w;
        } else {
            w.parent.right = w;
        }
    }
    u.left = w.right;
    if (u.left != nil) {
        u.left.parent = u;
    }
    u.parent = w;
    w.right = u;
    if (u == r) r = w;
}
```

In terms of the Treap data structure, the most important property of a rotation is that the depth of \( w \) decreases by one while the depth of \( u \) increases by one.

Using rotations, we can implement the \texttt{add(x)} operation as follows: We create a new node, \( u \), and assign \( u.x = x \) and pick a random value for \( u.p \). Next we add \( u \) using the usual \texttt{add(x)} algorithm for a BinarySearchTree, so that \( u \) is now a leaf of the Treap. At this point, our Treap satisfies the binary search tree property, but not necessarily the heap property. In particular, it may be the case that \( u.parent.p > u.p \). If this is the case, then
we perform a rotation at node $w = u.parent$ so that $u$ becomes the parent of $w$. If $u$ continues
to violate the heap property, we will have to repeat this, decreasing $u$’s depth by one every
time, until $u$ either becomes the root or $u.parent.p < u.p$.

```java
public boolean add(T x) {
    Node<T> u = new Node<T>();
    u.x = x;
    u.p = rand.nextInt();
    if (super.add(u)) {
        bubbleUp(u);
        return true;
    }
    return false;
}

void bubbleUp(Node<T> u) {
    while (u.parent != null && u.parent.p > u.p) {
        if (u.parent.right == u) {
            rotateLeft(u.parent);
        } else {
            rotateRight(u.parent);
        }
    }
    if (u.parent == null) {
        r = u;
    }
}
```

An example of an `add(x)` operation is shown in Figure 7.6.

The running-time of the `add(x)` operation is given by the time it takes to follow
the search path for $x$ plus the number of rotations performed to move the newly-added
node, $u$, up to its correct location in the `Treap`. By Lemma 7.2, the expected length of
the search path is at most $2\ln n + O(1)$. Furthermore, each rotation decreases the depth of
$u$. This stops if $u$ becomes the root, so the expected number of rotations can not exceed
the expected length of the search path. Therefore, the expected running-time of the `add(x)`
operation in a `Treap` is $O(\log n)$. (Exercise 7.2 asks you to show that the expected number
of rotations performed during an insertion is actually only $O(1)$.)

The `remove(x)` operation in a `Treap` is the opposite of the `add(x)` operation. We
search for the node, $u$, containing $x$ and then perform rotations to move $u$ downwards until
it becomes a leaf and then we splice $u$ from the `Treap`. Notice that, to move $u$ downwards,
we can perform either a left or right rotation at $u$, which will replace $u$ with $u.right$
or $u.left$, respectively. The choice is made by the first of the following that apply:

1. If $u.left$ and $u.right$ are both null, then $u$ is a leaf and no rotation is performed.
2. If $u.left$ (or $u.right$) is null, then perform a right (or left, respectively) rotation at
   $u$. 

Figure 7.6: Adding the value 1.5 into the Treap from Figure 7.4.
98 CHAPTER 7. RANDOM BINARY SEARCH TREES

3. If u.left.p < u.right.p (or u.left.p > u.right.p), then perform a right rotation (or left rotation, respectively) at u.

These three rules ensure that the Treap doesn’t become disconnected and that the heap property is maintained once u is removed.

```java
boolean remove(T x) {
    Node<T> u = findLast(x);
    if (compare(u.x, x) == 0) {
        trickleDown(u);
        splice(u);
        return true;
    }
    return false;
}
```

A example of the remove(x) operation is shown in Figure 7.7.

The trick to analyzing the running time of the remove(x) operation is to notice that this operation is the reverse of the add(x) operation. In particular, if we were to reinser x, using the same priority u.p, then the add(x) operation would do exactly the same number of rotations and would restore the Treap to exactly the same state it was in before the remove(x) operation took place. (Reading from bottom-to-top, Figure 7.7 illustrates the insertion of the value 9 into a Treap.) This means that the expected running time of the remove(x) on a Treap of size n is proportional to the expected running time of the add(x) operation on a Treap of size n – 1. We conclude that the expected running time of remove(x) is $O(\log n)$.

7.2.1 Summary

The following theorem summarizes the performance of the Treap data structure:
Figure 7.7: Removing the value 9 from the Treap in Figure 7.4.
Theorem 7.2. A Treap implements the SSet interface. A Treap supports the operations add(x), remove(x), and find(x) in $O(\log n)$ expected time per operation.

It is worth comparing the Treap data structure to the SkiplistSet data structure. Both implement the SSet operations in $O(\log n)$ expected time per operation. In both data structures, add(x) and remove(x) involve a search and then a constant number of pointer changes (see Exercise 7.2 below). Thus, for both these structures, the expected length of the search path is the critical value in assessing their performance. In a SkiplistSet, the expected length of a search path is

$$2\log n + O(1),$$

In a treap, the expected length of a search path is

$$2\ln n + O(1) \approx 1.386 \log n + O(1).$$

Thus, the search paths in a Treap are considerably shorter and this translates into noticeably faster operations on Treaps than Skiplists. Exercise 4.2 in Chapter 4 shows how the expected length of the search path in a Skiplist can be reduced to

$$e \ln n + O(1) \approx 1.884 \log n + O(1)$$

by using biased coin tosses. Even with this optimization, the expected length of search paths in a SkiplistSet is noticeably longer than in a Treap.

7.3 Summary and Exercises

Random binary search trees have been studied extensively. Devroye [13] gives a proof of Lemma 7.1 and related results. There are much stronger results in the literature as well. The most impressive of which is due to Reed [45], who shows that the expected height of a random binary search tree is

$$\alpha \ln n - \beta \ln \ln n + O(1)$$

where $\alpha \approx 4.31107$ is the unique solution on $[2, \infty)$ of the equation $\alpha \ln((2e/\alpha)) = 1$ and $\beta = \frac{3}{27\ln(\alpha/2)}$. Furthermore, the variance of the height is constant.

The name Treap was coined by Aragon and Seidel [48] who discussed Treaps and some of their variants. However, their basic structure was studied much earlier by Vuillemin [52] who called them Cartesian trees.

One space-optimization of the Treap data structure that is sometimes performed is the elimination of the explicit storage of the priority $p$ in each node. Instead, the priority of a node, $u$, is computed by hashing $u$’s address in memory (in 32-bit Java, this is equivalent to hashing $u$.hashCode()). Although a number of hash functions will probably work well for this in practice, for the important parts of the proof of Lemma 7.1 to remain valid, the hash function should be randomized and have the min-wise independent property: For any distinct values $x_1, \ldots, x_k$, each of the hash values $h(x_1), \ldots, h(x_k)$ should be distinct with high probability and, for each $i \in \{1, \ldots, k\}$,

$$\Pr\{h(x_i) = \min\{h(x_1), \ldots, h(x_k)\}\} \leq c/k$$

for some constant $c$. One such class of hash functions that is easy to implement and fairly fast is tabulation hashing [41].
Exercise 7.1. Prove the assertion that there are 21,964,800 sequences that generate the
tree on the right hand side of Figure 7.1. (Hint: Give a recursive formula for the number
of sequences that generate a complete binary tree of height $h$ and evaluate this formula for
$h = 3$.)

Exercise 7.2. Use both parts of Lemma 7.2 to prove that the expected number of rotations
performed by an add($x$) operation (and hence also a remove($x$) operation) is $O(1)$.

Exercise 7.3. Design and implement a version of a Treap that includes a get($i$) operation
that returns the key with rank $i$ in the Treap. (Hint: Have each node, $u$, keep track of the
size of the subtree rooted at $u$.)

Exercise 7.4. Design and implement a version of a Treap that supports the split($x$) oper-
ation. This operation removes all values from the Treap that are greater than $x$ and returns
a second Treap that contains all the removed values.

For example, the code $t2 = t.split(x)$ removes from $t$ all values greater than $x$ and
returns a new Treap $t2$ containing all these values. The split($x$) operation should run in
$O(\log n)$ expected time.
Chapter 8

Scapegoat Trees

In this chapter we study a binary search tree data structure, the ScapegoatTree, that keeps itself balanced by partial rebuilding operations. During one of these operations, an entire subtree is deconstructed and rebuilt into a perfectly balanced subtree.

There are many ways of rebuilding a subtree the subtree rooted at node u into a perfectly balanced tree. One of the simplest is to traverse u’s subtree, gathering all its nodes into an array a and then to recursively build a balanced subtree using a. If we let $m = a.length/2$, then the element $a[m]$ becomes the root of the new subtree, $a[0], \ldots, a[m-1]$ get stored recursively in the left subtree and $a[m+1], \ldots, a[a.length-1]$ get stored recursively in the right subtree.

```java
void rebuild(Node<T> u) {
    int ns = size(u);
    Node<T> p = u.parent;
    Node<T>[] a = (Node<T>[]) Array.newInstance(Node.class, ns);
    packIntoArray(u, a, 0);
    if (p == null) {
        r = buildBalanced(a, 0, ns);
        r.parent = null;
    } else if (p.right == u) {
        p.right = buildBalanced(a, 0, ns);
        p.right.parent = p;
    } else {
        p.left = buildBalanced(a, 0, ns);
        p.left.parent = p;
    }
}

int packIntoArray(Node<T> u, Node<T>[] a, int i) {
    if (u == null) {
        return i;
    }
    i = packIntoArray(u.left, a, i);
    a[i++] = u;
    return packIntoArray(u.right, a, i);
}
```
A call to `rebuild(u)` takes $O(\text{size}(u))$ time. The subtree built by `rebuild(u)` has minimum height; there is no tree of smaller height that has `size(u)` nodes.

### 8.1 ScapegoatTree: A Binary Search Tree with Partial Rebuilding

A ScapegoatTree is a BinarySearchTree that, in addition to keeping track of the number, $n$, of nodes in the tree also keep a counter $q$ that maintains an upper-bound on the number of nodes.

```java
int q;
```

At all times, $n$ and $q$ obey the following inequalities:

$$q/2 \leq n \leq q.$$  

In addition, a ScapegoatTree has logarithmic height; at all times, the height of the scapegoat tree does not exceed:

$$\log_{3/2} q \leq \log_{3/2} 2n < \log_{3/2} n + 2.$$  \hspace{1cm} (8.1)

Even with this constraint, a ScapegoatTree can look surprisingly unbalanced. The tree in Figure 8.1 has $q = n = 10$ and height $5 < \log_{3/2} 10 \approx 5.679$.

Implementing the `find(x)` operation in a ScapegoatTree is done using the standard algorithm for searching in a BinarySearchTree (see Section 6.2). This takes time proportional to the height of the tree which, by (8.1) is $O(\log n)$.

To implement the `add(x)` operation, we first increment $n$ and $q$ and then use the usual algorithm for adding $x$ to a binary search tree; we search for $x$ and then add a new leaf $u$ with $u.x = x$. At this point, we may get lucky and the depth of $u$ might not exceed $\log_{3/2} q$ If so, then we leave well enough alone and don’t do anything else.

Unfortunately, it will sometimes happen that `depth(u) > \log_{3/2} q`. In this case we need to do something to reduce the height. This isn’t a big job; there is only one node, namely $u$, whose depth exceeds $\log_{3/2} q$. To fix $u$, we walk from $u$ back up to the root.
looking for a *scapegoat*, $w$. The scapegoat, $w$, is a very unbalanced node. It has the property that
\[
\frac{\text{size}(w.\text{child})}{\text{size}(w)} > \frac{2}{3}, \tag{8.2}
\]
where $w.\text{child}$ is the child of $w$ on the path from the root to $u$. We’ll very shortly prove that a scapegoat exists. For now, we can take it for granted. Once we’ve found the scapegoat $w$, we completely destroy the subtree rooted at $w$ and rebuild it into a perfectly balanced binary search tree. We know, from (8.2), that, even before the addition of $u$, $w$’s subtree was not a complete binary tree. Therefore, when we rebuild $w$, the height decreases by at least 1 so that height of the ScapegoatTree is once again at most $\log_{3/2} q$.

```java
boolean add(T x) {
    // first do basic insertion keeping track of depth
    Node<T> u = newNode(x);
    int d = addWithDepth(u);
    Node<T> w = u.parent;
    if (d > log32(q)) {
        // depth exceeded, find scapegoat
        while (3*size(w) <= 2*size(w.parent))
            w = w.parent;
        rebuild(w.parent);
    }
    return true;
}
```

If we ignore the cost of finding the scapegoat $w$ and rebuilding the subtree rooted at $w$, then the running-time of \texttt{add(x)} is dominated by the initial search, which takes $O(\log q) =$
Figure 8.2: Inserting 3.5 into a ScapegoatTree increases its depth to 6, which violates (8.1) since $6 > \log_{3/2} 11 \approx 5.914$. A scapegoat is found at the node containing 5.

$O(\log n)$ time. We will account for the cost of finding the scapegoat and rebuilding using amortized analysis in the next section.

The implementation of remove($x$) in a ScapegoatTree is very simple. We search for $x$ and remove it using the usual algorithm for removing a node from a BinarySearchTree. (Note that this can never increase the height of the tree.) Next, we decrement $n$ but leave $q$ unchanged. Finally, we check if $q > 2n$ and, if so, we rebuild the entire tree into a perfectly balanced binary search tree and set $q = n$

```java
boolean remove(T x) {
    if (super.remove(x)) {
        if (2*n < q) {
            rebuild(r);
            q = n;
        }
        return true;
    }
    return false;
}
```

Again, if we ignore the cost of rebuilding, the running-time of the remove($x$) operation is proportional to the height of the tree is therefore $O(\log n)$.

8.1.1 Analysis of Correctness and Running-Time

In this section we analyze the correctness and amortized running-time of operations on a ScapegoatTree. We first prove the correctness by showing that, when the add($x$) operation results in a node that violates Condition (8.1), then we can always find a scapegoat:
Lemma 8.1. Let $u$ be a node of depth $h > \log_{3/2} q$ in a ScapegoatTree. Then there exists a node $w$ on the path from $u$ to the root such that

$$\frac{\text{size}(w)}{\text{size(parent}(w))} > \frac{2}{3} .$$

Proof. Suppose, for the sake of contradiction, that this is not the case, and

$$\frac{\text{size}(w)}{\text{size(parent}(w))} \leq \frac{2}{3} .$$

for all nodes $w$ on the path from $u$ to the root. Denote the path from the root to $u$ as $r = u_0, \ldots, u_h = u$. Then, we have $\text{size}(u_0) = n$, $\text{size}(u_1) \leq \frac{2}{3} n$, $\text{size}(u_2) \leq \frac{4}{9} n$ and, more generally,

$$\text{size}(u_i) \leq \left(\frac{2}{3}\right)^i n .$$

But this gives a contradiction, since $\text{size}(u) \geq 1$, hence

$$1 \leq \text{size}(u) \leq \left(\frac{2}{3}\right)^h n < \left(\frac{2}{3}\right)^{\log_{3/2} q} n \leq \left(\frac{2}{3}\right)^{\log_{3/2} n} n = \left(\frac{1}{n}\right) n = 1 .$$

Next, we analyze the parts of the running time that we have not yet accounted for. There are two parts: The cost of calls to $\text{size}(u)$ when search for scapegoat nodes, and the cost of calls to $\text{rebuild}(w)$ when we find a scapegoat $w$. The cost of calls to $\text{size}(u)$ can be related to the cost of calls to $\text{rebuild}(w)$, as follows:

Lemma 8.2. During a call to $\text{add}(x)$ in a ScapegoatTree, the cost of finding the scapegoat $w$ and rebuilding the subtree rooted at $w$ is $O(\text{size}(w))$

Proof. The cost of rebuilding the scapegoat node $w$, once we find it, is $O(\text{size}(w))$. When searching for the scapegoat node, we call $\text{size}(u)$ on a sequence of nodes $u_0, \ldots, u_k$ until we find the scapegoat $u_k = w$. However, since $u_k$ is the first node in this sequence that is a scapegoat, we know that

$$\text{size}(u_i) < \frac{2}{3} \text{size}(u_{i+1})$$

for all $i \in \{0, \ldots, k-2\}$. Therefore, the cost of all calls to $\text{size}(u)$ is

$$O\left(\sum_{i=0}^{k} \text{size}(u_{k-i})\right) = O\left(\text{size}(u_k) + \sum_{i=0}^{k-1} \text{size}(u_{k-i-1})\right) = O\left(\text{size}(u_k) + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i \text{size}(u_k)\right) = O\left(\text{size}(u_k) \left(1 + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i\right)\right) = O(\text{size}(u_k)) = O(\text{size}(w)) ,$$

where the last line follows from the fact that the sum is a geometrically decreasing series.
All that remains is to prove an upper-bound on the cost of calls to `rebuild(u):

**Lemma 8.3.** Starting with an empty `ScapegoatTree` any sequence of `m add(x)` and `remove(x)` operations causes at most $O(m \log m)$ time to be used by `rebuild(u)` operations.

**Proof.** To prove this, we will use a credit scheme. Each node stores a number of credits. Each credit can pay for some constant, $c$, units of time spent rebuilding. The scheme gives out a total of $O(m \log m)$ credits and every call to `rebuild(u)` is paid for with credits stored at $u$.

During an insertion or deletion, we give one credit to each node on the path to the inserted node, or deleted node, $u$. In this way we hand out at most $\log_{3/2} q \leq \log_{3/2} m$ credits per operation. During a deletion we also store an additional 1 credit “on the side.” Thus, in total we give out at most $O(m \log m)$ credits. All that remains is to show that these credits are sufficient to pay for all calls to `rebuild(u)`.

If we call `rebuild(u)` during an insertion, it is because $u$ is a scapegoat. Suppose, without loss of generality, that

$$\frac{\text{size}(u.\text{left})}{\text{size}(u)} > \frac{2}{3}.$$  

Using the fact that

$$\text{size}(u) = 1 + \text{size}(u.\text{left}) + \text{size}(u.\text{right})$$

we deduce that

$$\frac{1}{2} \text{size}(u.\text{left}) > \text{size}(u.\text{right})$$

and therefore

$$\text{size}(u.\text{left}) - \text{size}(u.\text{right}) > \frac{1}{2} \text{size}(u.\text{left}) > \frac{1}{3} \text{size}(u).$$

Now, the last time a subtree containing $u$ was rebuilt (or when $u$ was inserted, if a subtree containing $u$ was never rebuilt), we had

$$\text{size}(u.\text{left}) - \text{size}(u.\text{right}) \leq 1.$$  

Therefore, the number of operations `add(x)` or `remove(x)` operations that have affected `u.\text{left}` or `u.\text{right}` since then is at least

$$\frac{1}{3} \text{size}(u) - 1.$$  

and there are therefore at least this many credits stored at $u$ that are available to pay for the $O(\text{size}(u))$ time it takes to call `rebuild(u)`.

If we call `rebuild(u)` during a deletion, it is because $q > 2n$. In this case, we have $q - n > n$ credits stored “on the side” and we use these to pay for the $O(n)$ time it takes to rebuild the root. This completes the proof. □
8.2. DISCUSSION AND EXERCISES

8.1.2 Summary

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

**Theorem 8.1.** A ScapegoatTree implements the SSet interface. Ignoring the cost of rebuild\((u)\) operations, a ScapegoatTree supports the operations add\((x)\), remove\((x)\), and find\((x)\) in \(O(\log n)\) time per operation.

Furthermore, beginning with an empty ScapegoatTree, any sequence of \(m\) add\((x)\) and remove\((x)\) operations results in a total of \(O(m \log m)\) time spent during all calls to rebuild\((u)\).

8.2 Discussion and Exercises

The term scapegoat tree is due to Galperin and Rivest [24], who define and analyze these trees. However, the same structure was discovered earlier by Andersson [2, 4], who called them general balanced trees since they can have any shape as long as their height is small.

Experimenting with the ScapegoatTree implementation will reveal that it is often considerably slower than the other SSet implementations in this book. This may be somewhat surprising, since height bound of

\[
\log_{3/2} q \approx 1.709 \log n + O(1)
\]

is better than the expected length of a search path in a Skiplist and not too far from that of a Treap. The implementation could be optimized by storing the sizes of subtrees explicitly at each node (or at least reusing already computed subtree sizes). Even with these optimizations, there will always be sequences of add\((x)\) and delete\((x)\) operation for which a ScapegoatTree takes longer than other SSet implementations.

This gap in performance is due to the fact that, unlike the other SSet implementations discussed in this book, a ScapegoatTree can spend a lot of time restructuring itself. Exercise 8.1 asks you to prove that there are sequences of \(n\) operations in which a ScapegoatTree will spend on the order of \(n \log n\) time in calls to rebuild\((u)\). This is in contrast to other SSet implementations discussed in this book that only make \(O(n)\) structural changes during a sequence of \(n\) operations. This is, unfortunately, a necessary consequence of the fact that a ScapegoatTree does all its restructuring by calls to rebuild\((u)\) [14].

Despite their lack of performance, there are applications in which a ScapegoatTree could be the right choice. This would occur any time there is additional data associated with nodes that can not be updated in constant time when a rotation is performed, but that can be updated during a rebuild\((u)\) operation. In such cases, the ScapegoatTree and related structures based on partial rebuilding may work. An example of such an application is outlined in Exercise 8.5.

Exercise 8.1. Show that, if we start with an empty ScapegoatTree and call add\((x)\) for \(x = 1, 2, 3, \ldots, n\), then the total time spent during calls to rebuild\((u)\) is at least \(cn \log n\) for some constant \(c > 0\).

Exercise 8.2. The ScapegoatTree, as described in this chapter guarantees that the length of the search path does not exceed \(\log_{3/2} q\). Design, analyze, and implement a modified version of ScapegoatTree where the length of the search path does not exceed \(\log_b q\), where \(b\) is a parameter with \(1 < b < 2\).
What does your analysis and/or experiments say about the amortized cost of $\text{find}(x)$, $\text{add}(x)$ and $\text{remove}(x)$ as a function of $b$?

**Exercise 8.3.** Analyze and implement a WeightBalancedTree. This is a tree in which each node $u$, except the root, maintains the balance invariant that $\text{size}(u) \leq (2/3)\text{size}(u.\text{parent})$. The $\text{add}(x)$ and $\text{remove}(x)$ operations are identical to the standard BinarySearchTree operations, except that any time the balance invariant is violated at a node $u$, the subtree rooted at $u.\text{parent}$ is rebuilt.

Your analysis should show that operations on a WeightBalancedTree run in $O(\log n)$ amortized time.

**Exercise 8.4.** Analyze and implement a CountdownTree. In a CountdownTree each node $u$ keeps a timer $u.\text{t}$. The $\text{add}(x)$ and $\text{remove}(x)$ operations are exactly the same as in a standard BinarySearchTree except that, whenever one of these operations affects $u$’s subtree, $u.\text{t}$ is decremented. When $u.\text{t} = 0$ the entire subtree rooted at $u$ is rebuilt into a perfectly balanced binary search tree. When a node $u$ is involved in a rebuilding operation (either because $u$ is rebuilt or one of $u$’s ancestors is rebuilt) $u.\text{t}$ is reset to $\text{size}(u)/3$.

Your analysis should show that operations on a countdown tree run in $O(\log n)$ amortized time. (Hint: First show that each node $u$ satisfies some version of a balance invariant.)

**Exercise 8.5.** Design and implement a Sequence data structure that maintains a sequence (list) of elements. It supports these operations:

- $\text{addAfter}(e)$: Add a new element after the element $e$ in the sequence. Return the new newly added element. (If $e$ is null, the new element is added at the beginning of the sequence.)

- $\text{remove}(e)$: Remove $e$ from the sequence.

- $\text{testBefore}(e_1, e_2)$: return $\text{true}$ if and only if $e_1$ comes before $e_2$ in the sequence.

The first two operations should run in $O(\log n)$ amortized time. The third operation should run in constant-time.

The Sequence data structure can be implemented by storing the elements in something like a ScapegoatTree, in the same order that they occur in the sequence. To implement $\text{testBefore}(e_1, e_2)$ in constant time, each element $e$ is labelled with an integer that encodes the path from the root to $e$. In this way, $\text{testBefore}(e_1, e_2)$ can be implemented just by comparing the labels of $e_1$ and $e_2$. 
Chapter 9

Red-Black Trees

In this chapter we present red-black trees, a version of binary search trees that have logarithmic depth. Red-black trees are one of the most widely-used data structures in practice. They appear as the primary search structure in many library implementations, including the Java Collections Framework and several implementations of the C++ Standard Template Library. They are also used within the Linux operating system kernel. There are several reasons for the popularity of red-black trees:

1. A red-black tree storing \( n \) values has height at most \( 2 \log n \).
2. The \( \text{add}(x) \) and \( \text{remove}(x) \) operations on a red-black tree run in \( O(\log n) \) worst-case time.
3. The amortized number of rotations done during an \( \text{add}(x) \) or \( \text{remove}(x) \) operation is constant.

The first two of these properties already put red-black trees ahead of skiplists, treaps, and scapegoat trees. Skiplists and treaps rely on randomization and their \( O(\log n) \) running times are only expected. Scapegoat trees have a guaranteed bound on their height, but \( \text{add}(x) \) and \( \text{remove}(x) \) only run in \( O(\log n) \) amortized time. The third property is just icing on the cake. It tells us that that the time needed to add or remove an element \( x \) is dwarfed by the time it takes to find \( x \).\(^1\)

However, the nice properties of red-black trees come with a price: implementation complexity. Maintaining a bound of \( 2 \log n \) on the height is not easy. It requires a careful analysis of a number of cases and it requires that the implementation do exactly the right thing in each case. One misplaced rotation or change of color produces a bug that can be very difficult to understand and track down.

Rather than jump directly into the implementation of red-black trees, we will first provide some background on a related data structure: 2-4 trees. This will give some insight into how red-black trees were discovered and why efficiently maintaining red-black trees is even possible.

9.1 2-4 Trees

A 2-4 tree is a rooted tree with the following properties:

\(^1\)Note that skiplists and treaps also have this property in the expected sense. See Exercise 4.1 and Exercise 7.2.
Property 9.1 (height). All leaves have the same depth.

Property 9.2 (degree). Every internal node has 2, 3, or 4 children.

An example of a 2-4 tree is shown in Figure 9.1. The properties of 2-4 trees imply that their height is logarithmic in the number of leaves:

Lemma 9.1. A 2-4 tree with \( n \) leaves has height at most \( \log n \).

Proof. The lower-bound of 2 on the degree of an internal node implies that, if the height of a 2-4 tree is \( h \), then it has at least \( 2^h \) leaves. In other words,

\[
n \geq 2^h.
\]

Taking logarithms on both sides of this equation gives \( h \leq \log n \). \( \square \)

9.1.1 Adding a Leaf

Adding a leaf to a 2-4 tree is easy (see Figure 9.2). If we want to add a leaf \( u \) as the child of some node \( w \) on the second-last level, we simply make \( u \) a child of \( w \). This certainly maintains the height property, but could violate the degree property; if \( w \) had 4 children prior to adding \( u \), then \( w \) now has 5 children. In this case, we split \( w \) into two nodes \( w \) and \( w' \) having 2 and 3 children, respectively. But now \( w' \) has no parent, so we recursively make \( w' \) a child of \( w \)'s parent. Again, this may cause \( w \)'s parent to have too many children in which case we split it. This process goes on until we reach a node that has fewer than 4 children, or until we split the root, \( r \), into two nodes \( r \) and \( r' \). In the latter case, we make a new root that has \( r \) and \( r' \) as children. This simultaneously increases the depth of all leaves and so maintains the height property.

Since the height of the 2-4 tree is never more than \( \log n \), the process of adding a leaf finishes after at most \( \log n \) steps.

9.1.2 Removing a Leaf

Removing a leaf from a 2-4 tree is a little more tricky (see Figure 9.3). To remove a leaf \( u \) from its parent \( w \), we just remove it. If \( w \) had only two children prior to the removal of \( u \), then \( w \) is left with only one child and violates the degree property.
Figure 9.2: Adding a leaf to a 2-4 Tree. This process stops after one split because \texttt{w.parent} has degree less than 4 before the addition.
Figure 9.3: Removing a leaf from a 2-4 Tree. This process goes all the way to the root because all of u’s ancestors and their siblings have degree 2.
To correct this, we look at \( w \)'s sibling, \( w' \). The node \( w' \) is sure to exist since \( w \)'s parent has at least 2 children. If \( w' \) has 3 or 4 children, then we take one of these children from \( w' \) and give it to \( w \). Now \( w \) has 2 children and \( w' \) has 2 or 3 children and we are done.

On the other hand, if \( w' \) has only two children, then we merge \( w \) and \( w' \) into a single node, \( w \), that has 3 children. Next we recursively remove \( w' \) from the parent of \( w' \). This process ends when we reach a node, \( u \), where \( u \) or its sibling has more than 2 children; or we reach the root. In the latter case, if the root is left with only 1 child, then we delete the root and make its child the new root. Again, this simultaneously decreases the height of every leaf and therefore maintains the height property.

Again, since the height of the tree is never more than \( \log n \), the process of removing a leaf finishes after at most \( \log n \) steps.

### RedBlackTree: A Simulated 2-4 Tree

A red-black tree is a binary search tree in which each node, \( u \), has a *color* which is either *red* or *black*. Red is represented by the value 0 and black by the value 1.

```java
public class Node<T> extends BSTNode<Node<T>, T> {
    byte color;
}
```

Before and after any operation on a red-black tree, the following two properties are satisfied. Each property is defined both in terms of the colors red and black, and in terms of the numeric values 0 and 1.

**Property 9.3** (black-height). There are the same number of black nodes on every root to leaf path. (The sum of the colors on any root to leaf path is the same.)

**Property 9.4** (no-red-edge). No two red nodes are adjacent. (For any node \( u \), except the root, \( u.color + u.parent.color \geq 1 \).)

Notice that we can always color the root, \( r \), of a red-black tree black without violating either of these two properties, so we will assume that the root is black, and the algorithms for updating a red-black tree will maintain this. Another trick that simplifies red-black trees is to treat the external nodes (represented by \( \text{nil} \)) as black nodes. This way, every real node, \( u \), of a red-black tree has exactly two children, each with a well-defined color. An example of a red-black tree is shown in Figure 9.4

### 9.2.1 Red-Black Trees and 2-4 Trees

At first it might seem surprising that a red-black tree can be efficiently updated to maintain the black-height and no-red-edge properties, and it seems unusual to even consider these as useful properties. However, red-black trees were designed to be an efficient simulation of 2-4 trees as binary trees.

Refer to Figure 9.5. Consider any red-black tree, \( T \), having \( n \) nodes and perform the following transformation: Remove each red node \( u \) and connect \( u \)'s two children directly to the (black) parent of \( u \). After this transformation we are left with a tree \( T' \) having only black nodes.
Figure 9.4: An example of a red-black tree with black-height 3. External (nil) nodes are drawn as squares.

Figure 9.5: Every red-black tree has a corresponding 2-4 tree.
Every internal node in $T'$ has 2, 3, or 4 children: A black node that started out with two black children will still have two black children after this transformation. A black node that started out with one red and one black child will have three children after this transformation. A black node that started out with two red children will have 4 children after this transformation. Furthermore, the black-height property now guarantees that every root-to-leaf path in $T'$ has the same length. In other words, $T'$ is a 2-4 tree!

The 2-4 tree $T'$ has $n + 1$ leaves that correspond to the $n + 1$ external nodes of the red-black tree. Therefore, this tree has height $\log(n + 1)$. Now, every root to leaf path in the 2-4 tree corresponds to a path from the root of the red-black tree $T$ to an external node. The first and last node in this path are black and at most one out of every two internal nodes is red, so this path has at most $\log(n + 1)$ black nodes and at most $\log(n + 1) - 1$ red nodes. Therefore, the longest path from the root to any internal node in $T$ is at most

$$2\log(n + 1) - 2 \leq 2\log n,$$

for any $n \geq 1$. This proves the most important property of red-black trees:

**Lemma 9.2.** The height of red-black tree with $n$ nodes is at most $2\log n$.

Now that we have seen the relationship between 2-4 trees and red-black trees, it is not hard to believe that we can efficiently maintain a red-black tree while adding and removing elements.

We have already seen that adding an element in a BinarySearchTree can be done by adding a new leaf. Therefore, to implement $\text{add}(x)$ in a red-black tree we need a method of simulating splitting a degree 5 node in a 2-4 tree. A degree 5 node is represented by a black node that has two red children one of which also has a red child. We can “split” this node by coloring it red and coloring its two children black. An example of this is shown in Figure 9.6.

Similarly, implementing $\text{remove}(x)$ requires a method of merging two nodes and borrowing a child from a sibling. Merging two nodes is the inverse of a split (shown in Figure 9.6), and involves coloring two (black) siblings red and coloring their (red) parent black. Borrowing from a sibling is the most complicated of the procedures and involves both rotations and recoloring of nodes.

Of course, during all of this we must still maintain the no-red-edge property and the black-height property. While it is no longer surprising that this can be done, there are a large number of cases that have to be considered if we try to do a direct simulation of a 2-4 tree by a red-black tree. At some point, it just becomes simpler to forget about the underlying 2-4 tree and work directly towards maintaining the red-black tree properties.

## 9.2.2 Left-Leaning Red-Black Trees

There is no single definition of a red-black tree. Rather, there are a family of structures that manage to maintain the black-height and no-red-edge properties during $\text{add}(x)$ and $\text{remove}(x)$ operations. Different structures go about it in different ways. Here, we implement a data structure that we call a RedBlackTree. This structure implements a particular variant of red-black trees that satisfies an additional property:

**Property 9.5 (left-leaning).** At any node $u$, if $u.\text{left}$ is black, then $u.\text{right}$ is black.
Figure 9.6: Simulating a 2-4 tree split operation during an addition in a red-black tree. (This simulates the 2-4 tree addition shown in Figure 9.2.)
Figure 9.7: Flips, pulls and pushes

Note that the red-black tree shown in Figure 9.4 does not satisfy the left-leaning property. It is violated by the parent of red node in the rightmost path.

The reason for maintaining the left-leaning property is that it reduces the number of cases encountered when updating the tree during `add(x)` and `remove(x)` operations. In terms of 2-4 trees, it implies that every 2-4 tree has a unique representation: A node of degree 2 becomes a black node with 2 black children. A node of degree 3 becomes a black node whose left child is red and whose right child is black. A node of degree 4 becomes a black node with two red children.

Before we describe the implementation of `add(x)` and `remove(x)` in detail, we first present some simple subroutines used by these methods that are illustrated in Figure 9.7.

The first two subroutines are for manipulating colors while preserving the black-height property. The `pushBlack(u)` method takes as input a black node `u` that has two red children and colors `u.red` and its two children black. The `pullBlack(u)` method reverse this operation:

```c
void pushBlack(Node<T> u) {
    u.color--;  // decrease color of current node
    u.left.color++;  // increase color of left child
    u.right.color++;  // increase color of right child
}
void pullBlack(Node<T> u) {
    u.color++;    // increase color of current node
    u.left.color--;  // decrease color of left child
    u.right.color--;  // decrease color of right child
}
```

The `flipLeft(u)` method swaps the colors of `u` and `u.right` and then performs a left rotation at `u`. This reverses the colors of these two nodes as well as their parent-child relationship:

```c
void flipLeft(Node<T> u) {
    swapColors(u, u.right);  // swap colors of u and u.right
    rotateLeft(u);  // perform left rotation at u
}
```
is especially useful in restoring the left-leaning property at a node \( u \) that violates it (because \( u.\text{left} \) is black and \( u.\text{right} \) is red). In this special case, we can be assured this operation preserves both the black-height and no-red-edge properties. The \( \text{flipRight}(u) \) operation is symmetric to \( \text{flipLeft}(u) \) with the roles of left and right reversed.

```java
void flipRight(Node<T> u) {
    swapColors(u, u.left);
    rotateRight(u);
}
```

### 9.2.3 Addition

To implement \( \text{add}(x) \) in a \texttt{RedBlackTree}, we perform a standard \texttt{BinarySearchTree} insertion, which adds a new leaf, \( u \), with \( u.x = x \) and set \( u.\text{color} = \text{red} \). Note that this does not change the black height of any node, so it does not violate the black-height property. It may, however violate the left-leaning property (if \( u \) is the right child of its parent) and it may violate the no-red-edge property (if \( u \)'s parent is \texttt{red}). To restore these properties, we call the method \( \text{addFixup}(u) \).

```java
boolean add(T x) {
    Node<T> u = newNode(x);
    u.color = red;
    boolean added = add(u);
    if (added)
        addFixup(u);
    return added;
}
```

The \( \text{addFixup}(u) \) method, illustrated in Figure 9.8, takes as input a node \( u \) whose color is \texttt{red} and which may be violating the no-red-edge property and/or the left-leaning property. The following discussion is probably impossible to follow without referring to Figure 9.8 or recreating it on a piece of paper while reading this discussion. Indeed, the reader may wish to study this figure before continuing.

If \( u \) is the root of the tree, then we can color \( u \) black and this restores both properties. If \( u \)'s sibling is also red, then \( u \)'s parent must be black, so both the left-leaning and no-red-edge properties already hold.

Otherwise, we first determine if \( u \)'s parent, \( w \) violates the left-leaning property and, if so, perform a \( \text{flipLeft}(w) \) operation and set \( u = w \). This leaves us in a well-defined state: \( u \) is the left child of its parent, \( w \), so \( w \) now satisfies the left-leaning property. All that remains is to ensure the no-red-edge property at \( u \). We only have to worry about the case where \( w \) is \texttt{red}, since otherwise \( u \) already satisfies the no-red-edge property.
9.2. RedBlackTree: A Simulated 2-4 Tree

Figure 9.8: A single round in the process of fixing Property 2 after an insertion.
Since we are not done yet, u is red and w is red. The no-red-edge property (which is
only violated by u and not by w) implies that u’s grandparent g exists and is black. If g’s
right child is red, then the left-leaning property ensures that both g’s children are red, and
a call to pushBlack(g) makes g red and w black. This restores the no-red-edge property at
u, but may cause it to be violated at g, so the whole process starts over with u = g

If g’s right child is black, then a call to flipRight(g) makes w the (black) parent
of g and gives w two red children u and g. This ensures that u satisfies the no-red-edge
property and g satisfies the left-leaning property. In this case we can stop.

```java
void addFixup(Node<T> u) {
    while (u.color == red) {
        if (u == r) { // u is the root - done
            u.color = black;
            return;
        }
        Node<T> w = u.parent;
        if (w.left.color == black) { // ensure left-leaning
            flipLeft(w);
            u = w;
            w = u.parent;
        } else {
            pushBlack(g);
            u = g;
        }
    }
}
```

The `insertFixup(u)` method takes constant time per iteration and each iteration
either finishes or moves u closer to the root. This implies that the `insertFixup(u)` method
finishes after \(O(\log n)\) iterations in \(O(\log n)\) time.

### 9.2.4 Removal

The `remove(x)` operation in a `RedBlackTree` tree is the most complicated operation to
implement, and this is true of all known implementations. Like `remove(x)` in a `Binary-
SearchTree` the operation boils down to finding a node w with only one child, u, and splicing
w out of the tree by having w.parent adopt u.

The problem with this is that, if w is black, then the black-height property will now
be violated at w.parent. We get around this problem, temporarily, by adding w.color to
u.color. Of course, this introduces two other problems: (1) u and w both started out black, then u.color + w.color = 2 (double black), which is invalid color. If w was red, then it is replaced by a black node u, which may violate the left-leaning property at u.parent. Both of these problems are resolved with a call to the `removeFixup(u)` method.

```java
RedBlackTree

boolean remove(T x) {
    Node<T> u = findLast(x);
    if (u == nil || compare(u.x, x) != 0)
        return false;
    Node<T> w = u.right;
    if (w == nil) {
        w = u;
        u = w.left;
    } else {
        while (w.left != nil)
            w = w.left;
        u.x = w.x;
        u = w.right;
    }
    splice(w);
    u.color += w.color;
    u.parent = w.parent;
    removeFixup(u);
    return true;
}
```

The `removeFixup(u)` method takes as input a node u whose color is black (1) or double-black (2). If u is double-black, then `removeFixup(u)` performs a series of rotations and recolorings that move the double-black node up the tree until it can be gotten rid of. During this process, the node u changes until, at the end of this process, u refers to the root of the subtree that has been changed. The root of this subtree may have changed color. In particular, it may have gone from red to black, so the `removeFixup(u)` method finishes by checking if u’s parent violates the left-leaning property and, if so, fixes it.

```java
RedBlackTree

void removeFixup(Node<T> u) {
    while (u.color > black) {
        if (u == r) {
            u.color = black;
        } else if (u.parent.left.color == red) {
            u = removeFixupCase1(u);
        } else if (u == u.parent.left) {
            u = removeFixupCase2(u);
        } else {
            u = removeFixupCase3(u);
        }
    }
}
```
if (u != r) { // restore left-leaning property, if necessary
    Node<T> w = u.parent;
    if (w.right.color == red && w.left.color == black) {
        flipLeft(w);
    }
}

The removeFixup(u) method is illustrated in Figure 9.9. Again, the following text will be very difficult, if not impossible, to follow without referring constantly to Figure 9.9.

Each iteration of the loop in removeFixup(u) processes the double-black node u based on one of four cases.

Case 0: u is the root. This is the easiest case to treat. We recolor u to be black and this does not violate any of the red-black tree properties.

Case 1: u’s sibling, v, is red. In this case, u’s sibling is the left child of its parent, w (by the left-leaning property). We perform a right-flip at w and then proceed to the next iteration. Note that this causes w’s parent to violate the left-leaning property and it causes the depth of u to increase. However, it also implies that the next iteration will be in Case 3 with w colored red. When examining Case 3, below, we will see that this means the process will stop during the next iteration.

```
RedBlackTree
Node<T> removeFixupCase1(Node<T> u) {
    flipRight(u.parent);
    return u;
}
```

Case 2: u’s sibling, v is black and u is the left child of its parent, w. In this case, we call pullBlack(w), making u black, v red, and darkening the color of w to black or double-black. At this point, w does not satisfy the left-leaning property, so we call flipLeft(w) to fix this.

At this point, w is red and v is the root of the subtree we started with. We need to check if w causes no-red-edge property to be violated. We do this by inspecting w’s right child, q. If q is black, then w satisfies the no-red-edge property and we can continue to the next iteration with u=v.

Otherwise (q is red), both the no-red-edge property and the left-leaning property are violated at q and v, respectively. A call to rotateLeft(w) restores the left-leaning property, but the no-red-edge property is still violated. At this point, q is the left child of v and w is the left child of q. q and w are both red and v is black or double-black. A flipRight(v) makes q the parent of both v and w. Following this up by a pushBlack(q) makes both v and w black and sets the color of q back to the original color of w.

At this point, there is no more double-black node and the no-red-edge and black-height properties are reestablished. The only possible problem that remains is that the right child of v may be red, in which case the left-leaning property is violated. We check this and perform a flipLeft(v) to correct it if necessary.
Figure 9.9: A single round in the process of eliminating a double-black node after a removal.
RedBlackTree

Node<T> removeFixupCase2(Node<T> u) {
    Node<T> w = u.parent;
    Node<T> v = w.right;
    pullBlack(w); // w.left
    flipLeft(w); // w is now red
    Node<T> q = w.right;
    if (q.color == red) { // q-w is red-red
        rotateLeft(w);
        flipRight(v);
        pushBlack(q);
        if (v.right.color == red)
            flipLeft(v);
        return q;
    } else {
        return v;
    }
}

Case 3: u’s sibling is black and u is the right child of its parent, w. This case is symmetric to Case 2 and is handled mostly the same way. The only differences come from the fact that the left-leaning property is asymmetric so requires different handling.

As before, we being with a call to pullBlack(w), which makes v red and u black. A call to flipRight(w) promotes v to the root of the subtree. At this point w is red, and the code branches two ways depending on the color of w’s left child, q.

If q is red, then the code finishes up exactly the same way that Case 2 finishes up, but is even simpler since there is no danger of v not satisfying the left-leaning property.

The more complicated case occurs when q is black. In this case, we examine the color if v’s left child. If it is red, then v has two red children and its extra black can be pushed down with a call to pushBlack(v). At this point, v now has w’s original color and we are done.

If v’s left child is black then v violates the left-leaning property and we restore this with a call to flipLeft(v). The next iteration of removeFixup(u) then continues with u = v.

RedBlackTree

Node<T> removeFixupCase3(Node<T> u) {
    Node<T> w = u.parent;
    Node<T> v = w.left;
    pullBlack(w);
    flipRight(w); // w is now red
    Node<T> q = w.left;
    if (q.color == red) { // q-w is red-red
        rotateRight(w);
        flipLeft(v);
        pushBlack(q);
    } else {
        return v;
    }
}
Each iteration of removeFixup(u) takes constant time. Cases 2 and 3 either finish or move u closer to the root of the tree. Case 0 (where u is the root) always terminates and Case 1 leads immediately to Case 3 which also terminates. Since the height of the tree is at most $2 \log n$, we conclude that there are at most $O(\log n)$ iterations of removeFixup(u) so removeFixup(u) runs in $O(\log n)$ time.

9.3 Summary

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

**Theorem 9.1.** A RedBlackTree implements the SSet interface. A RedBlackTree supports the operations add(x), remove(x), and find(x) in $O(\log n)$ worst-case time per operation.

Not included in the above theorem is the extra bonus

**Theorem 9.2.** Beginning with an empty RedBlackTree, any sequence of $m$ add(x) and remove(x) operations results in a total of $O(m)$ time spent during all calls addFixup(u) and removeFixup(u).

We will only sketch a proof of Theorem 9.2. By comparing addFixup(u) and removeFixup(u) with the algorithms for adding or removing a leaf in a 2-4 tree, we can convince ourselves that this property is something that is inherited from a 2-4 tree. In particular, if we can show that the total time spent splitting, merging, and borrowing in a 2-4 tree is $O(m)$, then this implies Theorem 9.2.

The proof of this for 2-4 trees uses the potential method of amortized analysis.\(^2\)

Define the potential of an internal node u in a 2-4 tree as

$$
\Phi(u) = \begin{cases} 
  c & \text{if } u \text{ has 2 children} \\
  0 & \text{if } u \text{ has 3 children} \\
  3c & \text{if } u \text{ has 3 children}
\end{cases}
$$

and the potential of a 2-4 tree as the sum of the potentials of its nodes. When a split occurs, it is because a node of degree 4 becomes two nodes, one of degree 2 and one of

\(^2\)See the proofs of Lemma 2.2 and Lemma 3.1 for other applications of the potential method.
degree 3. This means that the overall potential drops by $3c - c - 0 = c$. Thus, the time spent performing a split is accounted for by the drop in potential. When a merge occurs, two nodes that used to have degree 2 are replaced by one node of degree 3. The result is a drop in potential of $2c - 0 = 2c$. This drop in potential accounts for the cost of the merge.

A borrow operation happens only once per deletion and, except for merges and splits, the potential can only increase by a constant amount for each addition or removal of a leaf.

The above analysis implies that, beginning with an empty 2-4 tree, any sequence of $m$ additions and removals of leaves results in at most $O(m)$ time spent splitting nodes, merging nodes, and borrowing children from nodes. Theorem 9.2 is a consequence of this analysis and the correspondence between 2-4 trees and red-black trees.

9.4 Discussion and Exercises

Red-black trees were first introduced by Guibas and Sedgewick [28]. Despite their high implementation complexity they are found in some of the most commonly-used libraries and applications. Most algorithms and data structures discuss some variant of red-black trees.

Andersson [3] describes a left-leaning version of balanced trees that are similar to red-black trees but have the additional constraint that any node has at most one black child. This implies that these trees simulate 2-3 trees rather than 2-4 trees. They are significantly simpler, though, than the RedBlackTree structure presented in this chapter.

Sedgewick [47] describes at least two versions of left-leaning red-black trees. These use recursion along with a simulation of top-down splitting and merging in 2-4 trees. The combination of these two techniques makes for particularly short and elegant code.

A related, and older, data structure is the AVL tree [1]. AVL trees are height-balanced: At each node $u$, the height of the subtree rooted at $u$.left and the subtree rooted at $u$.right differ by at most one. It follows immediately that, if $F(h)$ is the minimum number of nodes in a tree of height $h$, then $F(h)$ obeys the Fibonacci recurrence

$$F(h) = F(h - 1) + F(h - 2)$$

with base cases $F(0) = 1$ and $F(1) = 2$. This means $F(h)$ is approximately $\varphi^h / \sqrt{5}$, where $\varphi = (1 + \sqrt{5})/2 \approx 1.61803399$ is the golden ratio. (More precisely, $|\varphi^h / \sqrt{5} - F(h)| \leq 1/2$.) Arguing as in the proof of Lemma 9.1, this implies

$$h \leq \log_\varphi n \approx 1.440420088 \log n,$$

so AVL trees have smaller height than red-black trees. The height-balanced property can be maintained during add($x$) and remove($x$) operations by walking back up the path to the root and performing a rebalancing operation at each node $u$ where the height of $u$’s left and right subtrees differ by 2. See Figure 9.10.

Andersson’s variant of red-black trees, Sedgewick’s variant of red-black trees, and AVL trees are all simpler to implement than the RedBlackTree structure defined here. Unfortunately, none of them can guarantee that the amortized time spent rebalancing is $O(1)$ per update. In particular, there is no analogue of Theorem 9.2 for those structures.
Figure 9.10: Rebalancing in an AVL tree. At most 2 rotations are required to convert a node whose subtrees have height $h$ and $h + 2$ into a node whose subtrees each have height at most $h + 1$. 
Exercise 9.1. Why does the method remove(x) in the RedBlackTree implementation perform the assignment u.parent = u.parent? Shouldn’t this already be done by the call to splice(w)?

Exercise 9.2. Suppose a 2-4 tree, T, has $n_\ell$ leaves and $n_i$ internal nodes.

1. What is the minimum value of $n_i$, as a function of $n_\ell$?
2. What is the maximum value of $n_i$, as a function of $n_\ell$?
3. If $T'$ is a red-black tree that represents $T$, then how many red nodes does $T'$ have?

Exercise 9.3. Prove that, during an add(x) operation, an AVL tree must perform at most one relancing operation (that involves at most 2 rotations; see Figure 9.10). Give an example of an AVL tree and a remove(x) operation on that tree that requires on the order of $\log n$ rebalancing operations.

Exercise 9.4. Implement an AVLTree class that implements AVL trees as described above. Compare its performance to that of the RedBlackTree implementation. Which implementation has a faster find(x) operation?

Exercise 9.5. Design and implement a series of experiments that compare the relative performance of find(x), add(x), and remove(x) for SkipListSet, ScapegoatTree, Treap, and RedBlackTree. Be sure to include multiple test scenarios, including cases where the data is random, already sorted, is removed in random order, is removed in sorted order, and so on.
Chapter 10

Heaps

In this chapter we discuss two implementations of the extremely useful priority Queue data structure. The first is an implementation based on arrays. It is very fast, and is the basis of one of the fastest known sorting algorithms, namely heapsort. The second implementation is based on binary trees and is more flexible. In particular, it support a meld(h) operation that allows the priority queue to absorb the elements of a second priority queue h.

10.1 BinaryHeap: An Implicit Binary Tree

Our first implementation of a (priority) Queue is based on a technique that is over 400 years old. Eytzinger’s method allows us to represent a complete binary tree as an array. This is done by laying out the nodes of the tree in breadth-first order (see Section 6.1.2) in the array. In this way, the root is stored at position 0, the root’s left child is stored at position 1, the root’s right child at position 2, the left child of the left child of the root is stored at position 3, and so on. See Figure 10.1.

If we do this for a large enough tree, some patterns emerge. The left child of the node at index i is at index left(i) = 2i + 1 and the right child of the node at index i is at index right(i) = 2i + 2. The parent of the node at index i is at index parent(i) = (i − 1)/2.

```c
int left(int i) {
    return 2*i + 1;
}
```

Figure 10.1: Eytzinger’s method represents a complete binary tree as an array.
A `BinaryHeap` uses this technique to implicitly represent a complete binary tree in which the elements are *heap ordered*: The value stored at any index `i` is not smaller than the value stored at index `parent(i)`, with the exception of the root value, `i = 0`. It follows that the smallest value in the priority `Queue` is therefore stored at position 0 (the root).

In a `BinaryHeap`, the `n` elements are stored in an array `a`:

```java
T[] a;
int n;
```

Implementing the `add(x)` operation is fairly straightforward. As with all array-based structures, we first check if `a` is full (because `a.length = n`) and, if so, we grow `a`. Next, we place `x` at location `a[n]` and increment `n`. At this point, all that remains is to ensure that we maintain the heap property. We do this by repeatedly swapping `x` with its parent until `x` is no smaller than its parent. See Figure 10.2.

```java
boolean add(T x) {
    if (n + 1 > a.length) resize();
    a[n++] = x;
    bubbleUp(n-1);
    return true;
}
void bubbleUp(int i) {
    int p = parent(i);
    while (i > 0 && compare(a[i], a[p]) < 0) {
        swap(i,p);
        i = p;
        p = parent(i);
    }
}
```

Implementing the `remove()` operation, which removes the smallest value from the heap, is a little trickier. We know where the smallest value is (at the root), but we need to replace it after we remove it and ensure that we maintain the heap property.

The easiest way to do this is to replace the root with the value `a[n]` and decrement `n`. Unfortunately, the new element now at the root is probably not the smallest element, so it needs to be moved downwards. We do this by repeatedly comparing this element to its
Figure 10.2: Inserting the value 6 into a BinaryHeap.
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.

```java
BinaryHeap

T remove() {
    T x = a[0];
    swap(0, --n);
    trickleDown(0);
    if (3*n < a.length) resize();
    return x;
}

void trickleDown(int i) {
    do {
        int j = -1;
        int r = right(i);
        if (r < n && compare(a[r], a[i]) < 0) {
            int l = left(i);
            if (compare(a[l], a[r]) < 0) {
                j = l;
            } else { j = r; }
        } else { int l = left(i);
            if (l < n && compare(a[l], a[i]) < 0) { j = l; }
        }
        if (j >= 0) swap(i, j);
        i = j;
    } while (i >= 0);
}
```

As with other array-based structures we will ignore the time spent in calls to
`resize()` since these can be accounted for with the amortization argument from Lemma 2.1.
The running-time of both `add(x)` and `remove()` then depends on the height of the (implicit)
binary tree. However, this is a complete binary tree; every level except the last has the
maximum possible number of nodes. Therefore, if the height of this tree is $h$, then it has
at least $2^h$ nodes. Stated another way

$$n \geq 2^h.$$  

Taking logarithms on both sides of this equation gives

$$h \leq \log n.$$  

Therefore, both the `add(x)` and `remove()` operation run in $O(\log n)$ time.
Figure 10.3: Removing the minimum value, 4, from a BinaryHeap.
10.1.1 Summary

The following theorem summarizes the performance of a BinaryHeap:

**Theorem 10.1.** A BinaryHeap implements the (priority) Queue interface. Ignoring the cost of calls to resize(), an BinaryHeap supports the operations add(x) and remove() in $O(\log n)$ time per operation.

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

10.2 MeldableHeap: A Randomized Meldable Heap

In this section, we describe the MeldableHeap, a priority Queue implementation in which the underlying structure is also a heap-ordered binary tree. However, unlike a BinaryHeap in which the underlying binary tree is completely defined by the number of elements, there are no restrictions on the shape of the binary tree that underlies a MeldableHeap; anything goes.

The add(x) and remove() operations in a MeldableHeap are implemented in terms of the merge(h1, h2) operation. This operation takes two heap nodes $h_1$ 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 $h_2$.

The nice thing about a merge($h_1, h_2$) operation is that it can be defined recursively. If either of $h_1$ or $h_2$ is null, then we are merging with an empty set, so we return $h_2$ or $h_1$, respectively. Otherwise, assume $h_1.x < h_2.x$, since otherwise 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 $h_1.right$:

```java
Node merge(Node h1, Node h2) {
    if (h1 == null) return h2;
    if (h2 == null) return h1;
    if (h2.x.compareTo(h1.x) < 0) { // ensure h1.x < h2.x
        Node tmp = h1;
        h1 = h2;
        h2 = tmp;
    }
    if (rand.nextBoolean()) {
        h1.left = merge(h1.left, h2);
        if (h1.left != null) h1.left.parent = h1;
    } else {
        h1.right = merge(h1.right, h2);
        if (h1.right != null) h1.right.parent = h1;
    }
    return h1;
}
```
In the next section, we show that $\text{merge}(h_1, h_2)$ 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 $\text{merge}(h_1, h_2)$ operation, the $\text{add}(x)$ operation is easy. We create a new node $u$ containing $x$ and then merge $u$ with the root of our heap:

```java
boolean add(T x) {
    Node u = new Node();
    u.x = x;
    r = merge(u, r);
    r.parent = null;
    n++;
    return true;
}
```

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

The $\text{remove}()$ 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:

```java
T remove() {
    T x = r.x;
    r = merge(r.left, r.right);
    if (r != null) r.parent = null;
    n--;
    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:

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

### 10.2.1 Analysis of $\text{merge}(h_1, h_2)$

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

The following lemma is remarkable because it does not depend at all on the shape of the binary tree:
Lemma 10.1. The expected length of a random walk in a binary tree with \( n \) 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 1 step and then continues in a subtree of size \( n_1 \) or continues in a subtree of size \( n_2 \). By our inductive hypothesis, the expected length of the walk is then

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

since each of \( n_1 \) and \( n_2 \) are less than \( n \). To maximize this, over the choice of \( n_1 \in [0, n-1] \), we take the derivative and obtain

\[
(E[W])' = \frac{1}{2} \left( \frac{c}{n_1} - \frac{c}{n-n_1-1} \right) ,
\]

which is equal to 0 when \( n_1 = (n-1)/2 \). We can establish that this is a maximum fairly easily, so the expected number of steps taken by the random walk is

\[
E[W] = 1 + \frac{1}{2} \log(n_1 + 1) + \frac{1}{2} \log(n_2 + 1) \\
\leq 1 + \log((n-1)/2 + 1) \\
= 1 + \log((n+1)/2) \\
= \log(n+1) .
\]

With this result on random walks, we can now easily prove that the running time of the \texttt{merge}(h1, h2) operation is \( O(\log n) \).

Lemma 10.2. 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 \texttt{merge}(h1, h2) is at most \( O(\log n) \), where \( n = n_1 + n_2 \).

Proof. Each step of the merge algorithm takes a one step of a random walk, either in the heap rooted at \( h_1 \) or the heap rooted at \( h_2 \), depending on whether \( h_1.x < h_2.x \) or not. The algorithm terminates when either of these two random walks falls out of its corresponding tree (when \( h_1 = \text{null} \) or \( h_2 = \text{null} \)). Therefore, the expected number of steps performed by the merge algorithm is at most

\[
\log(n_1 + 1) + \log(n_2 + 1) \leq 2 \log n .
\]

10.2.2 Summary

The following theorem summarizes the performance of a \texttt{MeldableHeap}:

Theorem 10.2. A \texttt{MeldableHeap} implements the (priority) \texttt{Queue} interface. A \texttt{MeldableHeap} supports the operations \texttt{add(x)} and \texttt{remove()} in \( O(\log n) \) expected time per operation.
10.3 Discussion and Exercises

The implicit representation of a complete binary tree as an array, or list, seems to have been first proposed by Eytzinger [20], as a representation for pedigree family trees. The BinaryHeap data structure described here was first introduced by Williams [53].

The randomized MeldableHeap data structure described here appears to have first been proposed by Gambin and Malinowski [25]. Other meldable heap implementations exist, including leftist heaps [11, 34, Section 5.3.2], binomial heaps [51], Fibonacci heaps [23], pairing heaps [22], and skew heaps [50], although none of these is as simple as the MeldableHeap structure described here.

Some of the above structures also support a decreaseKey(u, y) operation in which the value stored at node u is decreased to y. (It is a pre-condition that y ≤ u.x.) This operation can be implemented in $O(\log n)$ time in most of the above structures by removing node u and adding y. However, some of these structures can implement decreaseKey(u, y) more efficiently. In particular, decreaseKey(u, y) takes $O(1)$ amortized time in Fibonacci heaps and $O(\log \log n)$ amortized time in a special version of pairing heaps [18]. This more efficient decreaseKey(u, y) operation has applications in speeding up several graph algorithms including Dijkstra’s shortest path algorithm [23].


