An Intuitive Introduction to Data Structures

Brian Heinold
Department of Mathematics and Computer Science
Mount St. Mary’s University
## Contents

1 Running times of algorithms
   1.1 Analyzing algorithms ................................................. 4
   1.2 Brief review of logarithms ........................................... 7
   1.3 Big O notation ......................................................... 7
   1.4 Examples ............................................................... 9
   1.5 A catalog of common running times ................................. 12
   1.6 Exercises ............................................................. 16

2 Lists
   2.1 Dynamic arrays ....................................................... 21
   2.2 Implementing a dynamic array ...................................... 22
   2.3 Linked lists ........................................................... 27
   2.4 Implementing a linked list .......................................... 29
   2.5 Comparison of dynamic arrays and linked lists ................ 36
   2.6 Making the linked list class generic .............................. 38
   2.7 The Java Collections Framework .................................. 41
   2.8 Applications of lists ................................................ 47
   2.9 Iterators* ............................................................... 48
   2.10 Exercises ............................................................. 52

3 Stacks and Queues
   3.1 A stack class ......................................................... 56
   3.2 Queues .................................................................. 58
   3.3 Stacks, queues, and deques in the Collections Framework ...... 61
   3.4 Applications ........................................................... 63
   3.5 Exercises ............................................................. 65

4 Recursion ................................................................. 67
# CONTENTS

4.1 Reversing a string ......................................................... 67
4.2 Length of a string and sum of a list ................................. 68
4.3 Power function .............................................................. 69
4.4 Printing the contents of a directory .................................. 69
4.5 The Sierpinski triangle .................................................... 70
4.6 Permutations ................................................................. 72
4.7 Combinations ............................................................... 73
4.8 Factoring numbers ......................................................... 74
4.9 Fibonacci numbers ......................................................... 75
4.10 Towers of Hanoi ............................................................ 77
4.11 Using recursion ............................................................. 80
4.12 Exercises ................................................................. 81

5 Binary Trees ................................................................. 85
   5.1 Implementing a binary tree ........................................... 86
   5.2 A different approach .................................................. 96
   5.3 Applications ............................................................ 98
   5.4 Exercises ............................................................... 98

6 Binary Search Trees, Heaps, and Priority Queues ................. 101
   6.1 Binary Search Trees .................................................. 101
   6.2 Implementing a BST .................................................. 102
   6.3 More about BSTs ..................................................... 111
   6.4 Comparable interface ................................................. 112
   6.5 Heaps ................................................................. 117
   6.6 Implementing a heap .................................................. 117
   6.7 Running time and applications of heaps ......................... 123
   6.8 Priority Queues ....................................................... 124
   6.9 Implementing a priority queue .................................... 124
   6.10 Priority queues and heaps in the Collections Framework .... 126
   6.11 Applications of priority queues ................................. 127
   6.12 Exercises ............................................................. 128

7 Sets, Maps and Hashing .................................................. 131
   7.1 Sets ................................................................. 131
   7.2 Hashing ............................................................... 135
# CONTENTS

7.3 Implementing a set with hashing ........................................ 136
7.4 Sets in the Collections Framework ..................................... 141
7.5 Applications of sets .......................................................... 141
7.6 Java's `hashCode` method .................................................. 143
7.7 Maps ............................................................................. 144
7.8 Implementing a map class .................................................... 145
7.9 Using hashing to implement a map ........................................ 149
7.10 Maps in the Collections Framework .................................... 151
7.11 Applications of maps ........................................................ 152
7.12 Exercises ................................................................. 155

8 Graphs ........................................................................... 159
8.1 Terminology ...................................................................... 160
8.2 Implementing a graph class ................................................. 161
8.3 Searching ........................................................................ 169
8.4 Digraphs .......................................................................... 172
8.5 An application of digraphs .................................................... 178
8.6 Weighted graphs ............................................................... 181
8.7 Kruskal's algorithm for minimum spanning trees ...................... 184
8.8 Dijkstra's shortest path algorithm ......................................... 187
8.9 Exercises ......................................................................... 191

9 Sorting .............................................................................. 195
9.1 Selection Sort .................................................................... 196
9.2 Bubble Sort ....................................................................... 197
9.3 Insertion Sort ..................................................................... 198
9.4 Shellsort ........................................................................... 199
9.5 Heapsort ........................................................................... 201
9.6 Treesort ............................................................................ 201
9.7 Merge Sort ......................................................................... 202
9.8 Quicksort ........................................................................... 205
9.9 Counting Sort ..................................................................... 208
9.10 Comparison of sorting algorithms ....................................... 209
9.11 Sorting with generics and lists .......................................... 211
9.12 Sorting in Java ................................................................... 212
9.13 Exercises ......................................................................... 213
CONTENTS

A A few Java prerequisites 217
  A.1 Interfaces ........................................................................... 217
  A.2 Abstract classes ............................................................... 220
  A.3 Referencing .................................................................... 220
  A.4 Throwing exceptions ....................................................... 222

Index 225
Preface

This book is based off the Data Structures and Algorithms class I taught in the fall of 2011. I used a bunch of data structures and introductory programming books as references, and while I liked parts of all the books, none of them approached things quite in the way I wanted, so I decided to write my own book. This book is just meant to be a simple introduction to basic data structures.

I've tried to keep explanations short and to the point. A guiding theme of this book is that the best way to understand a data structure or algorithm is to implement it. Generally, for each data structure and algorithm I provide a succinct explanation of how it works, followed by an implementation and discussion of the issues that arise in implementing it. After that there is some discussion of efficiency and applications. The implementations are designed with simplicity and understanding in mind. They are not the most efficient implementations and there is no exception-handling. In general, there are a lot of subtle issues about data structures that I chose not to cover in this book. In addition, my approach to topics is a lot more intuitive than it is formal. Anyone looking for a more formal approach could easily jump into a lot of other data structures and algorithms books after reading this one.

The first five chapters form the basis for the other chapters and should be covered first and probably in the order given. The other chapters can be covered in any order.

This is an early version, probably with lots of typos and some other mistakes. Please send comments, corrections, and suggestions to heinold@msmary.edu.

Last updated January 4, 2013.
Chapter 1

Running times of algorithms

Consider the problem of finding all the words that can be made from the letters of a given word. For instance, from the main word *computer*, one can make *put, mop, rope,* and *compute,* and many others. An algorithm to do this would be useful for programming a *Scrabble* computer player or just for cheating at word games.

One approach would be to systematically generate all combinations of letters from the main word and check each to see if each is a real word. For instance, from *computer,* we would start by generating *co,* *cm,* *cp,* etc., going all the way to *omputer,* looking for each in a word list to see which are real words.

A different approach would be to loop through the words in the word list and check if each can be made from the letters of the main word. So we would start with *aardvark, abaci, aback,* etc., checking each to see if it comes from the main word.

Which one is better? We could go with whatever approach is easier to program. Sometimes, this is a good idea, but not here. There is a huge difference in performance between the first and second algorithms. The first approach chokes on large main words. On my machine, it is very slow on 10-letter main words and impossible to use on 20-letter words. On the other hand, the second approach runs in a few tenths of a second for any nearly word size.

The reason that the first approach is slow is that the number of ways to arrange the letters explodes as the number of letters increases. For example, there are 24 ways to rearrange the letters of *bear,* 40,320 ways to rearrange the letters of *computer* and 119,750,400 ways to rearrange the letters of *encyclopedia.* For a 20-letter word, if all the letters are different, there are 20! (roughly 2.5 quintillion) different rearrangements. And this isn't all, as these are only rearrangements of all the letters, and we also have to consider all the two-letter, three-letter, etc. combinations of the letters. And for each and every combination of letters, we have to check if it is in the dictionary.

On the other hand, with the second approach we scan through the dictionary just once. For each word, we check to see if it can be made from the letters of the main word. And the key here is that this can be done very quickly. One way is to count the number of occurrences of each letter in the word and compare it to the number of occurrences of that letter in the main word. Even for large words, this can
be done in a split second. A typical dictionary would have maybe 100,000 words, so looping through
the dictionary performing a very quick operation for each word adds up to a running time on the order
of a fraction of a second up to maybe a second or so, depending on your system. The algorithm will
have nearly same running time for a main word like bear as for a main word like encyclopedia.

This is why it is important to be able to analyze algorithms. We need to be able to tell which algorithms
will run in a reasonable amount of time and which ones will take too long.

1.1 Analyzing algorithms

One important way to analyze an algorithm is to pick out the most important parameter(s) affecting the
running time of the algorithm and find a function that describes how the running time varies with the
parameter(s). Here is an example.

Comparison of two algorithms to find the mode of an array

Consider the problem of finding the most common element (the mode) in an array of integers. The
parameter of interest here is the size of the array. We are interested in how our algorithm will perform
as the array size is increased. Here is one algorithm:

```java
public static int mode(int[] a)
{
    int maxCount=0, mode=0;
    for (int i=0; i<a.length; i++)
    {
        int count = 0;
        for (int j=0; j<a.length; j++)
            if (a[j]==a[i])
                count++;
        if (count>maxCount)
        {
            maxCount = count;
            mode = a[i];
        }
    }
    return mode;
}
```

This code loops through the entire array, element by element, and for each element, it counts how many
other array elements are equal to it. The algorithm keeps track of the current maximum as it loops
through the array, updating it as necessary.

To analyze the algorithm, we notice that it loops through all \( n \) elements of the array, and for each of
those \( n \) times through the loop, the algorithm performs another loop through the entire array. For each
of those we do a comparison, for a total of \( n^2 \) comparisons. Note that each comparison is a fast operation
that has no dependence on the size of the array. Therefore, the running time of this algorithm is on the
order of \( n^2 \). This is a rough estimate. The actual running time is some function of the form \( cn^2 + d \),
where \( c \) and \( d \) are constants. While knowing those constants can be useful, we are not concerned with
that here. We are only concerned with the \( n^2 \) part of the formula. We say that this is an \( O(n^2) \) algorithm. The big \( O \) stands for “order of,” and this read as “big O of \( n^2 \)” or just “\( O \) \( n^2 \).”

The \( n^2 \) tells us how the running time changes as the number of elements increases. For a 100-element array, \( n^2 \) is 10,000. For a 1000-element array, \( n^2 \) is 1,000,000. For a 10,000-element array, \( n^2 \) is 100,000,000. The key to this big \( O \) notation is that it doesn’t tell us the exact running time, but instead it captures how the running time grows as the parameter varies.

Consider now a different algorithm for finding the mode: For simplicity’s sake, suppose that the array contains only integers between 0 and 999. Create an array called \( \text{counts} \) that keeps track of how many zeros, ones, twos, etc. are in the array. Loop through the array once, updating the \( \text{counts} \) array each time through the loop. Then loop through the \( \text{counts} \) array to find the maximum count. Here is the code:

```java
public static int mode(int[] a) {
    int[] counts = new int[1000];
    for (int i=0; i<a.length; i++)
        counts[a[i]]++;

    int maxCount=0, mode=0;
    for (int i=0; i<counts.length; i++)
        if (counts[i]>maxCount)
            {maxCount = counts[i]; mode = i;}

    return mode;
}
```

The first loop runs \( n \) times, doing a quick assignment at each step. The second loop runs 1000 times, regardless of the size of the array. So the running time of the algorithm is of the form \( an + b \) for some constants \( c \) and \( d \). Here the growth the function is linear instead of quadratic, \( n \) instead of \( n^2 \). Whereas the quadratic algorithm would require on the order of an unmanageable one trillion operations for a million-element array, this linear algorithm will require on the order of one million operations.

We note something here that happens quite often. The second algorithm traded memory for speed. It runs a lot faster than the first algorithm, but it requires more memory to do so (an array of 1000 counts versus a single counting variable). It often happens that we can speed up an algorithm at the cost of using more memory.

**Comparison of two searching algorithms**

Let’s examine another problem, that of determining whether an array contains a given element. Here is a simple search:

```java
public static boolean linearSearch(int[] a, int item) {
    for (int i=0; i<a.length; i++)
        if (a[i]==item)
            return true;
    return false;
}
```
In the best-case scenario, the element we are searching for (called item) will be the first element of the array. But algorithms are usually not judged on their best-case performance. They are judged on average-case or worst-case performance. For this algorithm, the worst case occurs if the item is not in the array or is the last element in the array. In this case, the loop will run \( n \) times, where \( n \), the parameter of interest, is the size of the array. So the worst case running time is \( O(n) \). Moreover, on average, we will have to search through about half of the elements (\( n/2 \) of them) before finding item. So the average case running time is \( O(n) \) also. Remember that we don’t care about constants; we just care about the variables. So instead of \( O(n/2) \), we just say \( O(n) \) here.

Here is another algorithm for this problem, called a *binary search*. It runs much faster than the previous algorithm, though it does require that the array be sorted.

```java
public static boolean binarySearch(int[] a, int item) {
    int start = 0, end = a.length - 1;

    while (end >= start) {
        int mid = (start + end) / 2;
        if (a[mid] == item)
            return true;
        if (a[mid] > item)
            end = mid - 1;
        else
            start = mid + 1;
    }
    return false;
}
```

The way the algorithm works is we start by examining the element in the middle of the array and comparing it to the item. If it actually equals item, then we’re done. Otherwise, if the middle element is greater than item, then we know that since the elements are in order, then item must not be in the second half of the array. So we just have to search through the first half of the array in that case. On the other hand, if the middle element is less than item, then we just have to search the second half of the array. We repeat this process on the appropriate half of the array, and repeat again as necessary until we home in on the location of the item or decide it isn’t in the array.

This is like searching through an alphabetical list of names for a specific name, say *Ralph*: We might start by looking in the middle of the list for *Ralph*. If the middle element is *Lewis*, then we know that *Ralph* must be in the second half of the list, so we just look the second half. We then look at the middle element of that list and throw out the appropriate half of that list. We keep that up as long as necessary.

Each time through the algorithm, we cut the search space in half. For instance, if we have 1000 elements to search through, after throwing away the appropriate half, we cut the search space down to 500. We then cut it in half again to 250, and then to 125, 63, 32, 16, 8, 4, 2, and 1. So in the worst case, it will take only 10 steps.

Suppose we have one million elements to search through. Repeatedly cutting things in half, we get 500,000, 250,000, 125,000, 63,000, 32,000, 16,000, 8000, 4000, 2000, 1000, \ldots, 4, 2, 1, for a total of...
20 steps. So increasing the array from 1000 to 1,000,000 elements only adds 11 steps. If we have one billion elements to search through, only 30 steps are required. With one trillion elements, only 40 steps are needed.

This is the hallmark of a logarithmic running time—multiplying the size of the search space by a certain factor adds a constant amount of steps to the running time. The binary search has an $O(\log n)$ running time.

### 1.2 Brief review of logarithms

Logarithms are for when we want to know what power to raise a number to in order to get another number. For example, if we want to know to what power we have to raise 2 to get 55, we can write that in equation form as $2^x = 55$. The solution is $x = \log_2(55)$, where $\log_2$ is called the logarithm base 2. In general, $x = \log_b(a)$ is the solution to $b^x = a$.

For example, $\log_2(8)$ is 3 because $2^3 = 8$. Similarly, $\log_2(16)$ is 4 because $2^4 = 16$. And we have that $\log_2(14)$ is about 3.807 (using a calculator). We can use bases other than 2. For example, $\log_5(25) = 2$ because $5^2 = 25$, and $\log_{1/27}(3) = -3$ because $3^{-3} = 1/27$. In computer science, the most important base is 2. In calculus, the most important base is base $e = 2.718\ldots$, which gives the natural logarithm. Instead of $\log_e(x)$ people usually write $\ln(x)$.

One particularly useful rule is that for any base $b$, we have $\log_b(x) = \ln(x)/\ln(b)$. So for example, $\log_2(14) = \ln(14)/\ln(2)$. This is useful because most calculators and many programming languages have $\ln(x)$ built in but not $\log_2(x)$.

With the binary search, we cut our search space in half at each step. This can keep going until the search space is reduced to only one item. So if we start with 1000 items, we cut things in half 10 times to get down to 1. We can get this by computing $\log_2(1000) \approx 9.96$ and rounding up to get 10. This is because asking how many times we have to cut things in half starting at 1000 to get to 1 is the same as asking how many times we have to double things starting at 1 to get 1000, which is the same as asking what power of 2 1000 is. So this is where the logarithm comes from in the binary search.

### 1.3 Big O notation

Say we fix a parameter $n$ (like array size or string length) that is important to the running time of algorithm. When we say that the running time of an algorithm is $O(f(n))$ (where $f(n)$ is a function like $n^2$ or $\log n$), we mean that the running time grows roughly like the function $f(n)$ as $n$ gets larger and larger.

At this point, it’s most important to have a good intuitive understanding of what big O notation means, but it is also helpful to have a formal definition. The reason for this is to clear up any ambiguities in case our intuition fails us. Formally, we say that a function $g(n)$ is $O(f(n))$ if there exist a real number $M$ and an integer $N$ such that $g(n) \leq Mf(n)$ for all $n \geq N$. 
Put into plainer (but slightly more ambiguous) English, when we say that an algorithm’s running time is $O(f(n))$, we are saying that it takes no more time than a constant multiple of $f(n)$ for large enough values of $n$. Roughly speaking, this means that a function which is $O(f(n))$ is of the following form:

$$cf(n) + [\text{terms that are small compared to } f(n) \text{ when } n \text{ is sufficiently large}].$$

For example, $4n^2 + 3n + 2$ is $O(n^2)$. For large values of $n$, the dominant term is $4n^2$, a constant times $n^2$. For large $n$, the other terms, $3n$ and $2$ are small compared to $n^2$. Some other functions that are $O(n^2)$ include $.5n^2$, $n^2 + 2n + \frac{1}{n}$, and $3n^2 + 1000$.

As another example, $10 \cdot 2^n + n^5 + n^2$ is $O(2^n)$. This is because the function is of the form a constant times $2^n$ plus some terms ($n^5$ and $n^2$), which are small compared to $2^n$ when $n$ is sufficiently large.

A good rule of thumb for writing a function in big $O$ notation is to drop all the terms except the most dominant one and to ignore any constants.

### Some technicalities

*This part can be skipped on a first reading.*

Note that by the formal definition, because of the $\leq$, a function with running time $n$ or $\log(n)$ or $1$ is technically $O(n^2)$. This may seem a little odd. There is another notation $\Theta(f(n))$ that captures more of a notion of equality. Formally, we say that a function $g(n)$ is $\Theta(f(n))$ if there exist real numbers $m$ and $M$ and an integer $N$ such that $mf(n) \leq g(n) \leq Mf(n)$ for all $n \geq N$. With this definition, only functions whose dominant term is $n^2$ are $\Theta(n)$. The big theta notation is closer to the intuitive idea we built up earlier, and probably should be used, but often programmers, mathematicians, and computer scientists will “abuse notation” and use big $O$ in place of big theta. It’s not anything to be overly concerned about.

There is one other notation that shows up from time to time (though not in this book), and that is little $o$. Formally, a function $g(n)$ is $o(f(n))$ if there exists an integer $N$ such that for all $M > 0$ and for all $n > N$, $g(n) \leq Mf(n)$. This notation behaves like a string less than, where a function is, for instance $o(n^2)$, if its dominant term is less than $n^2$. For example, running times of $n$, $\log(n)$, and $1$ are all $o(n)$, while $n^2$ is not $o(n^2)$.

So in summary, little $o$ acts like a less than, big theta acts like equality, and big $O$ acts like less than or equal to, though people often use it to mean the same thing as big theta. Some people are more careful about things than others. In this book we will take the looser approach and use big $O$.

### Important note

An $O(n)$ algorithm is usually, but not always, better than an $O(n^2)$ algorithm. For example, consider algorithm $A$ with running time $1000000n$ and algorithm $B$ with running time $.001n^2$. Algorithm $B$ is an $O(n^2)$ algorithm, yet it will outperform the $O(n)$ algorithm $A$ for all input sizes less than one billion, which may likely include all the values of $n$ that are likely to occur in practice.
1.4. EXAMPLES

Big O notation says that an \( O(n) \) algorithm must eventually outperform any \( O(n^2) \) algorithm, but it doesn’t say anything about how large \( n \) has to be before “eventually” takes effect. That value of \( n \) might be larger than any value that a program is likely to encounter.

Analyzing the big O growth of an algorithm is not the only or best way of analyzing the running time of an algorithm. It just gives a picture of how the running time will grow as the parameter grows.

1.4 Examples

Example 1  Here is code that sums up the entries in an array:

```java
public static int sum(int[] a)
{
    int total = 0;
    for (int i=0; i<a.length; i++)
        total += a[i];
    return total;
}
```

The running time of this algorithm is of the form \( cn + d \), where \( n \) is the size of the array. The \( d \) part of the running time comes from the constant amount of time it takes to create and initialize the variable total and return the result. The \( c \) part comes from the constant amount of time it takes to update the total, and as that operation is repeated \( n \) times, we get the \( cn \) term. Keeping only the dominant term, \( cn \), and ignoring the constant, we see that, overall, this is an \( O(n) \) algorithm.

Example 2  Here is some code involving nested for loops:

```java
public static int func(int[] a)
{
    for (int i=0; i<a.length; i++)
        for (int j=0; j<a.length; j++)
            System.out.println(a[i]+a[j]);
}
```

The outer loop runs \( n \) times and the inner loop runs \( n \) times for each of run of the outer loop, for a total of \( n \cdot n = n^2 \) times through the loop. The print statement takes a constant amount of time. So this is an \( O(n^2) \) algorithm.

Example 3  Below is a simple sorting algorithm called Selection Sort.

```java
public static int selectionSort(int[] a)
{
    for (int i=0; i<a.length-1; i++)
    {
        for (int j=i; j<a.length; j++)
        {
            if (a[i]>a[j])
            {
                int hold = a[i];
                a[i] = a[j];
            }
        }
    }
}
```
The outer loop runs \( n - 1 \) times, where \( n \) is the size of the array. The inner loop runs \( n \) times when \( i \) is 0, \( n - 1 \) times when \( i \) is 1, \( n - 2 \) times when \( i \) is 2, etc., down to 1 time when \( i \) is \( n - 2 \). Using a fairly well-known algebraic formula, the inner loop thus runs a total of \( 1 + 2 + 3 + \cdots + n - 1 = \frac{n(n-1)}{2} \) times. The operations in the loop have no dependence on \( n \), so overall this is an \( O(n^2) \) algorithm.

**Example 4** Here is a short code segment that doesn’t do anything particularly interesting.

```java
public static int func(int n)
{
    int sum=0;
    while (n>1)
    {
        sum += n;
        n /= 2;
    }
    return sum;
}
```

This is an \( O(\log n) \) algorithm. We notice that at each step \( n \) is cut in half and the algorithm terminates when \( n = 1 \). Nothing else of consequence happens inside the loop, so the number of times the loop runs is determined by how many times we can cut \( n \) in half before we get to 1, which is \( \log n \) times.

**Example 5** Here is a related example:

```java
public static int func(int n)
{
    for (int i=0; i<n; i++)
    {
        int sum=0;
        int x=n;
        while (x>1)
        {
            sum += x;
            x /= 2;
        }
    }
    return sum;
}
```

This example contains, more or less, the same code as the previous example wrapped inside of a for loop that runs \( n \) times. So each of those \( n \) times through the loop, the inside \( O(\log n) \) code is run, for a total running time given by \( O(n \log n) \).

**Example 6** Consider the following code:
1.4. EXAMPLES

\[ c = 0; \]
\[ \text{stop} = 100; \]
\[ \text{for} \ (\text{int} \ i=0; \ i<\text{stop}; \ i++) \]
\[ c+=a[i]; \]

The running time does not depend on the length of the array a. The code will take essentially the same amount of time, regardless of the size of a. This is therefore an \(O(1)\) algorithm. The 1 stands for the constant function \(f(n) = 1\).

Example 7  The following code consists of two unnested for loops.

```
int count=0, count2=0;
for (int i=0; i<a.length; i++)
    count++;
for (int i=0; i<a.length; i++)
    count2+=2;
```

This is an \(O(n)\) algorithm. We are summing the running times of two loops that each run in \(O(n)\) time.

Example 8  Here is an algorithm that runs in exponential time. It returns a list of all the subsets of the set \(\{1, 2, \ldots, n\}\).

```
public static List<List<Integer>> subsets(int n)
{
    List<List<Integer>> list = new ArrayList<List<Integer>>();
    if (n==0)
        list.add(new ArrayList<Integer>());
    else
    {
        List<List<Integer>> oneDown = subsets(n-1);
        list.addAll(oneDown);
        for (List<Integer> set : oneDown)
        {
            List<Integer> newSet = new ArrayList<Integer>(set);
            newSet.add(n);
            list.add(newSet);
        }
    }
    return list;
}
```

This is a recursive method—that is, a method that calls itself. See Chapter 4 for more on recursion. The algorithm here is pretty straightforward, though the syntax is a little verbose. The way the algorithm works is that it uses the subsets from one step down to generate the next ones. For instance, to generate the subsets of \(\{1, 2, 3\}\), we find the subsets of \(\{1, 2\}\), namely \(\emptyset, \{1\}, \{2\}\), and \(\{1, 2\}\). The subsets of \(\{1, 2, 3\}\) consist of these subsets as well as these subsets with a 3 added to each. We could describe the algorithm in pseudocode as follows:

```
function subsets(n)
{
```
if (n==0) return list containing just the empty set else
    oneDown = list of subsets of \{1,2,...,n-1\}
    newSet = list of subsets of oneDown with n added to end of each
    return list consisting of subsets from oneDown and newSet
\}

Since the subsets at step \(n\) come from two copies of the subsets at step \(n-1\), there will be twice as many of size \(n\) as there are of size \(n-1\). So the amount of work to generate the subsets doubles at each step, for an exponential \(O(2^n)\) running time.

Generating all the subsets of \{1, 2, ..., \(n\)\} isn’t necessarily all that useful, but this same technique can be used to generate the subsets of other sets to be used in a brute force search for something or other.

1.5 A catalog of common running times

In this section we look at some of the common functions that are used with big \(O\) notation. To get a good feel for the functions, it helps to compare them side by side. The table below compares the values of several common functions for varying values of \(n\).

<table>
<thead>
<tr>
<th>(1)</th>
<th>(10)</th>
<th>(100)</th>
<th>(1000)</th>
<th>(10000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(\log n)</td>
<td>0</td>
<td>3.3</td>
<td>6.6</td>
<td>10.0</td>
</tr>
<tr>
<td>(n)</td>
<td>1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>(n \log n)</td>
<td>0</td>
<td>33</td>
<td>664</td>
<td>9966</td>
</tr>
<tr>
<td>(n^2)</td>
<td>1</td>
<td>100</td>
<td>10,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>(n^3)</td>
<td>1</td>
<td>1000</td>
<td>1,000,000</td>
<td>1,000,000,000</td>
</tr>
<tr>
<td>(2^n)</td>
<td>2</td>
<td>1024</td>
<td>(1.2 \times 10^{30})</td>
<td>(1.1 \times 10^{301})</td>
</tr>
<tr>
<td>(n!)</td>
<td>1</td>
<td>3,628,800</td>
<td>(9.3 \times 10^{157})</td>
<td>(4.0 \times 10^{2567})</td>
</tr>
</tbody>
</table>

**Constant time** \(O(1)\)

The first entry in the table above corresponds to the constant function \(f(n) = 1\). This is a function whose value is always 1, no matter what \(n\) is.

<table>
<thead>
<tr>
<th>(1)</th>
<th>(10)</th>
<th>(100)</th>
<th>(1000)</th>
<th>(10000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Functions that are \(O(1)\) are of the form \(c + \text{[terms small compared to } c]\) for some constant \(c\). In practice, \(O(1)\) algorithms are usually ones whose running time does not depend on the size of the parameter. For example, consider a function that returns the minimum of an array of integers. If the array is sorted, then the function simply has to return the first element of the array. This operation takes practically the same amount of time whether the array has 10 elements or 10 million elements.
1.5. A CATALOG OF COMMON RUNNING TIMES

Logarithmic time \( O(\log n) \)

In the table below, and throughout the book, \( \log \) refers to \( \log_2 \). In other books you may see \( \lg \) for \( \log_2 \).

<table>
<thead>
<tr>
<th>( \log n )</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log n )</td>
<td>0</td>
<td>3.3</td>
<td>6.6</td>
<td>10.0</td>
<td>13.3</td>
</tr>
</tbody>
</table>

The key here is that a multiplicative increase in \( n \) corresponds to only a constant increase in the running time. For example, suppose in the table above the values of \( n \) are the size of an array and the table entries are running times of an algorithm in seconds. We see then that if we make the array 10 times as large, the running time will increase by 3.3 seconds. Multiplying the array size by a factor of 1000 increases the running time by 10 seconds, as in the table below:

<table>
<thead>
<tr>
<th>( \log n )</th>
<th>( 10^9 )</th>
<th>( 10^6 )</th>
<th>( 10^9 )</th>
<th>( 10^{12} )</th>
<th>( 10^{15} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log n )</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
</tr>
</tbody>
</table>

We see that even with a ridiculous array size of 1 quadrillion elements (over 1000 terabytes worth of memory), the running time of a logarithmic algorithm will only increase by a factor of 5 over a 1000-element array, from 10 seconds to 50 seconds.

We have seen already that the binary search runs in logarithmic time. The key to that is that we are constantly cutting the search space in half. In general, when we are cutting the amount of work in half (or by some other percentage) at each step, that is where logarithms come in.

Linear time \( O(n) \)

Linear time is one of the easier ones to grasp. It is what we are most familiar with in everyday life.

<table>
<thead>
<tr>
<th>( n )</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>10,000</td>
</tr>
</tbody>
</table>

Linear growth means that a constant change in the input results in a constant change in the output. For instance, if you get paid a fixed amount per hour, the amount of money you earn grows linearly with the number of hours you work—work twice as long, earn twice as much; work three times as long, earn three times as much; etc.

Some example linear algorithms include summing up the elements in an array and counting the number of occurrences of an element in an array.

Loglinear time \( O(n \log n) \)

Loglinear (or linarithmic) growth is a combination of linear and logarithmic growth. It is like linear growth of the form \( cn + d \) except that the constant \( c \) is not really constant, but slowly growing. For comparison, the table below compares linear and loglinear growth.
Loglinear growth often occurs as a combination of a linear and logarithmic algorithm. For instance, say we want to find all the words in a word list that are real words when written backwards. The linear part of the algorithm is that we have to loop once through the entire list. The logarithmic part of the algorithm is that for each word, we perform a binary search (assuming the list is alphabetized) to see if the backwards word is in the list. So overall, we perform \( n \) searches that take \( O(\log n) \) time, giving us an \( O(n \log n) \) algorithm.

Many of the sorting algorithms we will see in Chapter 9 are loglinear.

**Quadratic time** \( O(n^2) \)

Quadratic growth is a little less familiar in everyday life than linear time, but it shows up a lot in practice.

<table>
<thead>
<tr>
<th>( n^2 )</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>10,000</td>
<td>1,000,000</td>
<td>100,000,000</td>
</tr>
</tbody>
</table>

With quadratic growth, doubling the input size quadruples the running time. Tripling the input size corresponds to a ninefold increase in running time. As shown in the table, increasing the input size by a factor of ten corresponds to increasing the running time by a factor of 100. The table below shows how quickly quadratic growth can get out of hand compared to linear growth.

<table>
<thead>
<tr>
<th>( n )</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n^2 )</td>
<td>1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>10,000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>10,000</td>
<td>1,000,000</td>
<td>100,000,000</td>
</tr>
</tbody>
</table>

One place quadratic growth shows up is in operations on two-dimensional arrays as an \( n \times n \) array has \( n^2 \) elements. An other example is removing duplicates from an unsorted list. To do this, we loop over all elements and then for each element, loop over list to see if it is in there. Since the list is unsorted, the binary search can’t be used, so we must use a linear search giving us a running time on the order of \( n \cdot n = n^2 \).

There are a lot of cases, like with the mode algorithms from earlier in the chapter, where the first algorithm you think of may be \( O(n^2) \), but with some refinement or a more clever approach, things can be improved to \( O(n \log n) \) or \( O(n) \) (or better).

**Polynomial time in general** \( O(n^p) \)

Linear and quadratic growth are two examples of polynomial growth, growth of the form \( O(n^p) \), where \( p \) can be any positive power. Linear growth corresponds to \( p = 1 \) and quadratic to \( p = 2 \). There are many common algorithms where \( p \) is between 1 and 2. Values larger than 2 also occur fairly often. For example, the ordinary algorithm for multiplying two matrices is \( O(n^3) \), though there are other algorithms that run as low as about \( O(n^{2.37}) \).
Exponential time \( O(2^n) \)

With exponential time, a constant increase in \( n \) translates to a multiplicative increase in running time. For example, consider \( 2^n \):

<table>
<thead>
<tr>
<th>( 2^n )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
</table>

We see that each time \( n \) increases by 1, the running time doubles. Exponential growth is far faster than polynomial growth. While a polynomial may start out faster than an exponential, an exponential will always eventually overtake a polynomial. The table below shows just how much faster exponential growth can be than polynomial growth.

| \( n^3 \) | 1 | 1000 | 1,000,000 | 1,000,000,000 | \( 10^{12} \) |
| \( 2^n \) | 2 | 1024 | \( 1.2 \times 10^{30} \) | \( 1.1 \times 10^{301} \) | \( 2.0 \times 10^{3010} \) |

Already for \( n = 100 \), the running time (if measured in seconds) is so large as to be totally impossible to ever compute.

There is an old riddle that demonstrates exponential growth. It boils down to this: would you rather be paid \$1000 every day for a month or be given a penny on the first day of the month that is doubled to 2 cents the second day, doubled again to 4 cents the third day, and so on, doubling every day for the whole month? The first way translates to \$30,000 at the end of the month. The second way translates to \( 2^{29}/100 \) dollars, i.e., \$5,368,709.12. If we were to keep doubling for another 30 days, it would amount to over 5 quadrillion dollars.

Exponential growth shows up in many places. For example, a lot of times computing something by a brute force check of all the possibilities will lead to an exponential algorithm. As another example, Moore’s law, which has proven remarkably accurate, says that the number of transistors that can fit on a chip doubles every 18 months. That is, a constant increase in time (18 months) translates to a multiplicative increase of two in the number of transistors. This continuous doubling has meant that the number of transistors that can fit on a chip has increased from a few thousand in 1970 to over a billion by 2010.

Exponential growth is the opposite of logarithmic growth (exponentials and logs are inverse functions of each other). Whereas for logarithmic growth (say \( \log_2 n \)) an increase of ten times in \( n \) translates to only a constant increase of 3.3 in the running time, for exponential growth (say \( 2^n \)) a constant increase of 3.3 in \( n \) translates to a ten times increase in running time.

Factorial time \( O(n!) \)

The factorial \( n! \) is defined as \( n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 \). For example, \( 5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120 \). With exponential growth the running time usually grows so fast as to make things intractable for all but the smallest values of \( n \). Factorial growth is even worse. See the table below for a comparison.
CHAPTER 1. RUNNING TIMES OF ALGORITHMS

<table>
<thead>
<tr>
<th>Function</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n^3 )</td>
<td>1</td>
<td>1000</td>
<td>1,000,000</td>
<td>1,000,000,000</td>
<td>( 10^{12} )</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>2</td>
<td>1024</td>
<td>( 1.2 \times 10^{30} )</td>
<td>( 1.1 \times 10^{301} )</td>
<td>( 2.0 \times 10^{3010} )</td>
</tr>
<tr>
<td>( n! )</td>
<td>1</td>
<td>3,628,800</td>
<td>( 9.3 \times 10^{157} )</td>
<td>( 4.0 \times 10^{2567} )</td>
<td>( 2.8 \times 10^{35559} )</td>
</tr>
</tbody>
</table>

Just like with exponential growth, factorial growth often shows up when checking all the possibilities in a brute force search, especially when order matters. There are \( n! \) rearrangements (permutations) of a set of size \( n \). One famous problem that requires \( n! \) steps is a brute force search in the traveling salesman problem. In that problem, a salesman can travel from any city to any other city, and each route has its own cost. The goal is to find cheapest route that visits all the cities exactly once. If there are \( n \) cities, then there are \( n! \) possible routes (ways to rearrange the cities). A very simple algorithm would be to check all possible routes, and that is an \( O(n!) \) algorithm.

Other types of growth

These are just a sampling of the most common types of growth. There are infinitely many other types of growth. For example, beyond \( n! \) one can have \( n^n \) or \( n^n \) or \( n^n \).

1.6 Exercises

1. Write the following using big O notation:
   (a) \( n^2 + 2n + 3 \)
   (b) \( n + \log(n) \)
   (c) \( 3 + 5n^2 \)
   (d) \( 100 \cdot 2^n + n!/5 \)

2. Give the asymptotic running times of the following segments of code using big O notation in terms of \( n = a.length \).

   (a) \begin{verbatim}
   int sum=0;
   for (int i=0; i<a.length; i++)
   {
       for (int j=0; j<a.length; j+=2)
       sum += a[i]*j;
   }
   \end{verbatim}

   (b) \begin{verbatim}
   int sum=0;
   for (int i=0; i<a.length; i++)
   {
       for (int j=0; j<100; j+=2)
       sum += a[i]*j;
   }
   \end{verbatim}

   (c) \begin{verbatim}
   int sum=0;
   System.out.println((a[0]+a[a.length-1])/2.0);
   \end{verbatim}
1.6. EXERCISES

(d) int sum=0;
    for (int i=0; i<a.length; i++)
    {
        for (int j=0; j<a.length; j++)
        {
            for (int k=0; i<a.length; k++)
                sum += a[i]*j;
        }
    }

(e) int i = a.length-1;
    while (i>=1)
    {
        System.out.println(a[i]);
        i/=3;
    }

(f) for (int i=0; i<1000; i++)
    System.out.println(Math.pow(a[0], i));

(g) int i = a.length-1;
    while (i>=1)
    {
        a[i]=0;
        i/=3;
    }
    for (int i=0; i<a.length; i++)
    System.out.println(a);

(h) int i = a.length-1;
    while (i>=1)
    {
        for (int j=0; j<a.length; j++)
            a[j]+=i;
        i/=3;
    }

3. Give the asymptotic running times of the following methods using big O notation in terms of n.

(a) public int f(int n)
    {
        for (int i=0; i<n; i++)
        {
            for (int j=i; j<n; j++)
            {
                for (int k=j; k<n; k++)
                {
                    if (i+j+j=k+k)
                        count++;  
                }
            }
        }
        return count;
    }
(b) public int f(int n)
{
    int i=0, count=0;
    while (i<100 && n%5!=0)
    {
        i++;
        count+=n;
    }
    return count;
}

(c) public int f(int n)
{
    int value = n, count=0;
    while (value>1)
    {
        value = value/2;
        count++;
    }
    return count;
}

(d) public int f(int n)
{
    int count=0;
    for (int i=0; i<n; i++)
    {
        int value = n;
        while (value>1)
        {
            value = value/2;
            count++;
        }
    }
    return count;
}

(e) public int f(int n)
{
    int count=0;
    for (int i=0; i<n; i++)
        count++;
    int value = n;
    while (value>1)
    {
        value = value/2;
        count++;
    }
    return count;
}

4. Describe a situation where we have two algorithms, A and B, which both solve the same problem, with A having a better big O running time than algorithm B, yet B runs faster.

5. Show that $\log_b(n)$ is $O(\log_2 n)$ for any $b$. This means we don't have to worry about naming a particular base when we say an algorithm runs in $O(\log n)$ time.
6. Rewrite the linear search and binary search functions to return the first index of the item in the array and -1 if it is not in the array.

7. (a) Implement the two algorithms discussed in the beginning of the chapter for finding all the words that can be made from a word.
   (b) Analyze the running time of those two algorithms.
Chapter 2

Lists

In computer science, a list is a collection of objects, such as [5, 31, 19, 27, 16], where order matters and there may be repeated elements. Each element is referred to by its index in the list. By convention, indices start at 0, so that for the list above, 5 is at index 0, 31 is at index 1, etc.

We will consider mutable lists, lists that allows for operations like insertion and deletion of elements. There are two common approaches to implementing lists: dynamic arrays and linked lists. The former uses arrays to represent a list. The latter uses an approach of nodes and links. Each has its benefits and drawbacks, as we shall see.

In the next two sections we will implement lists each way. Our lists will have the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append(value)</td>
<td>adds a new element value to the end of the list</td>
</tr>
<tr>
<td>contains(value)</td>
<td>returns true iff (if and only if) value is in the list</td>
</tr>
<tr>
<td>delete(index)</td>
<td>deletes the element at index</td>
</tr>
<tr>
<td>get(index)</td>
<td>returns the value at index</td>
</tr>
<tr>
<td>insert(value, index)</td>
<td>adds a new element value at index</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>returns true iff the list is empty</td>
</tr>
<tr>
<td>set(value, index)</td>
<td>sets the value at index to value</td>
</tr>
<tr>
<td>size()</td>
<td>returns how many elements are in the list</td>
</tr>
<tr>
<td>toString()</td>
<td>returns a printable string representing the list</td>
</tr>
</tbody>
</table>

There are lots of other methods that we can give our lists. See the exercises at the end of the chapter.

2.1 Dynamic arrays

An array is stored in a contiguous block of memory like in the figure below.
In this case, each array element is a 4-byte integer. To get the element at index 3 the compiler knows that it must be $3 \times 4 = 12$ bytes past the start of the array, which would be location $@61de3f$. Regardless of the size of the array, a similar quick computation can be used to get the value at any index. This makes arrays are fast at random access, accessing a particular element anywhere in the array.

But this contiguous arrangement makes it difficult to insert and delete array elements. Suppose, for example, we want to insert the value 99 at index 2 so that the new array is [5, 99, 31, 19, 27, 16]. To make room, we have to shift most of the elements one slot to the right. This can take a while if the array is large. Moreover, how do we know we have room for another element? There is a chance that the space immediately past the end of the array is being used for something else. If that is the case, then we would have to move the array somewhere else in memory.

One way to deal with running out of room in the array when adding elements is to just make the array really large. The problem with this is how large is large enough? And we can’t go around making all our arrays have millions of elements because we would be wasting a lot of memory.

Instead, what is typically done is whenever more space is needed we increase the capacity of the array by a constant factor (say by doubling it), create a new array of that size, and then copy the contents from the old array over into the new one. By doubling the capacity, we insure that we will only periodically have to incur the expensive operation of creating a new array and moving the contents of the old one over. This type of data structure is called a dynamic array.

### 2.2 Implementing a dynamic array

Our implementation will have three class variables: an array $a$ where we store all the elements, a variable $capacity$ that is the capacity of the array, and a variable $numElements$ that keeps track of how many elements are in the list.

For instance, consider the list [77, 24, 11]. It has three elements, so $numElements$ is 3. However, the capacity of the list will usually be larger so that it has room for additional elements. In the figure below, we show the array with a capacity of 8. We can add five more elements before we have to increase the capacity.
Constructors

The simple constructor below sets the list to have a capacity of 100 elements and sets numElements to 0. This creates an empty list.

```java
public AList()
{
    this.capacity = 100;
    a = new int[capacity];
    numElements = 0;
}
```

It is not much more difficult to make a constructor creates that a list from an already-existing array. We initialize a to be an array with double the capacity of the user's array and then copy the elements from the user's array over, one by one.

```java
public AList(int[] array)
{
    capacity = array.length*2;
    a = new int[capacity];
    numElements = array.length;
    for (int i=0; i<array.length; i++)
        a[i] = array[i];
}
```

Getting and setting elements

Getting and setting elements of the list is straightforward.

```java
public int get(int index)
{
    return a[index];
}

public void set(int index, int value)
{
    a[index] = value;
}
```

size and isEmpty

Returning the size of the list is easy since we have a variable, numElements, keeping track of the size:

```java
public int size()
{
    return numElements;
}
```

Returning whether the list is empty or not is also straightforward—just return true if numElements is 0 and false otherwise. The code below does this in one line.

```java
return numElements == 0;
```
public boolean isEmpty()
{
    return numElements==0;
}

contains

To determine if the list contains an element or not, we can loop through the list, checking each element in turn. If we find the element we're looking for, return true. If we get all the way through the list without finding what we're looking for, then return false.

public boolean contains(int value)
{
    for (int i=0; i<numElements; i++)
        if (a[i]==value)
            return true;
    return false;
}

toString

Our toString method returns the elements of the list separated by commas and enclosed in square brackets, like [1,2,3]. The way we do this is to build up a string as we loop through the array. As we encounter each element, we add it along with a comma and a space to the string. When we're through, we have an extra comma and space at the end of the string. The final line below uses the substring method to take all of the string except for the extra comma and space and then add a closing bracket.

There is one place we have to be a little careful, and that is if the list is empty. In that case, the substring call will fail because the ending index will be negative. So we just take care of the empty list separately.

public String toString()
{
    if (numElements==0)
        return "[]";
    String s = "[";
    for (int i=0; i<numElements; i++)
        s += a[i] + ", ";
    return s.substring(0,s.length()-2)+"]";
}

Adding and deleting things

First, we will occasionally need to increase the capacity of the list as we run out of room. We have a private method (that won't be visible to users of the class since it is only intended to be used by our methods themselves) that doubles the capacity of the list.

private void enlarge()
{
2.2. IMPLEMENTING A DYNAMIC ARRAY

```java
capacity *= 2;
int[] newArray = new int[capacity];
for (int i=0; i<a.length; i++)
    newArray[i] = a[i];
a = newArray;
}
```

To append an element to a list we first make room, if necessary. Then we set the element right past the end of the list to the desired value and increase `numElements` by 1.

```java
public void append(int value)
{
    if (numElements==capacity)
        enlarge();
    a[numElements] = value;
    numElements++;
}
```

To delete an element from the list at a specified index we shift everything to the right of that index left by one element and reduce `numElements` by 1.

```java
public void delete(int index)
{
    for (int i=index; i<numElements-1; i++)
        a[i] = a[i+1];
    numElements--;
}
```

Finally, we consider inserting a value at a specified index. We start, just like with append, by increasing the capacity if necessary. We then shift everything right by one element to make room for the new element. Then we add in the new element and increase `numElements` by 1.

```java
public void insert(int value, int index)
{
    if (numElements==capacity)
        enlarge();
    for (int i=numElements; i>index; i--)
        a[i] = a[i-1];
a[index] = value;
    numElements++;
}
```

The entire `AList` class

Here is the entire class.

```java
public class AList
{
    private int[] a;
    private int numElements;
    private int capacity;

    public AList()
    {
```
this.capacity = 100;
a = new int[capacity];
numElements = 0;
}

public AList(int[] array)
{
capacity = array.length * 2;
a = new int[capacity];
numElements = array.length;

for (int i = 0; i < array.length; i++)
a[i] = array[i];
}

public int get(int index)
{
    return a[index];
}

public void set(int index, int value)
{
a[index] = value;
}

public boolean isEmpty()
{
    return numElements == 0;
}

public int size()
{
    return numElements;
}

public boolean contains(int value)
{
    for (int i = 0; i < numElements; i++)
        if (a[i] == value)
            return true;
    return false;
}

public String toString()
{
    if (numElements == 0)
        return "[]";

    String s = "[";
    for (int i = 0; i < numElements; i++)
        s += a[i] + ", ";
    return s.substring(0, s.length()-2) + "]";
}

private void enlarge()
{
capacity *= 2;
int[] newArray = new int[capacity];
for (int i = 0; i < a.length; i++)
    newArray[i] = a[i];
a = newArray;
}

public void append(int value)
{
2.3. LINKED LISTS

```java
if (numElements==capacity)
    enlarge();
a[numElements] = value;
numElements++;
}

public void delete(int index)
{
    for (int i=index; i<numElements-1; i++)
        a[i] = a[i+1];
    numElements--;
}

public void insert(int value, int index)
{
    if (numElements==capacity)
        enlarge();

    for (int i=numElements; i>index; i--)
        a[i] = a[i-1];
    a[index] = value;
    numElements++;
}
```

Important notes

A few notes about this class (and most of the others we will develop later in the book): First, this is just the bare bones of a class whose purpose is to demonstrate how dynamic arrays work. If we were going to use this class for something important we should add some error checking. For example, the get and set methods should raise exceptions if the user were to try to get or set an index less than 0 or greater than listSize-1. See Appendix A.4 for how to throw exceptions.

Second, after writing code for a class like this, it is important to check edge cases. For example, we should test not only how the insert method works when inserting at the middle of a list, but we should also check how it works when inserting an element at the left or right end or how insertion works on an empty list. We should also check how insert and append work when the list size bumps up against the capacity.

### 2.3 Linked lists

In the dynamic array approach to lists, the key to finding an element at a specific index is that the elements of the array are laid out one after another, so we know where everything is. Each element takes up the same amount of memory and an easy computation tells how far from the start the desired element is.

The problem with this is when we insert or delete things, we have to maintain this property of array elements being all right next to each other. This can involve moving quite a few elements around. Also, once the array backing our list fills up, we have find somewhere else in memory to hold our list and copy all the elements over.
A linked list is a different approach to lists. List elements are allowed to be widely separated in memory. This eliminates the problems mentioned above for dynamic arrays, but then how do we get from one list element to the next? That’s what the “link” in linked list is about. Along with each data element in the list, we also have a link, which points to where the next element in the list is located. See the figure below.

The linked list shown above is often drawn like this:

Each item in a linked list is called a node. Each node consists of a data value as well as a link to the next node in the list. The last node’s link is null; in other words, it is a link going nowhere. This indicates the end of the list.

Because the nodes don’t have to be next to each other in memory like with an array, inserting and deleting elements is simpler. In fact, to insert and delete things, all we have to do is reroute one or two links, as shown below:

Inserting:

Deleting:
On the other hand, getting an element at a specific index is slower. Since the nodes can be anywhere in memory, we can't do a simple computation to locate one. Instead we have to walk along the list from element to element, following the links, until we get to the desired location.

### 2.4 Implementing a linked list

To implement a linked list we will first create a class to represent a node. The Node class will have two variables, one to hold the node's data, and one for the link to the next node. The tricky part here is the link is a Node object itself, so the class is self-referential.

```java
private class Node {
    public int data;
    public Node link;

    public Node(int data, Node link) {
        this.data = data;
        this.link = link;
    }
}
```

There are a few things to note here. First, we have declared Node to be a nested class, a part of the LList class. This is because only the LList class needs to deal with nodes; nothing outside of the LList class needs the Node class. Secondly, notice that the class variables are declared public. We could declare them private and use getters and setters, but it's partly traditional with linked lists, and frankly a little easier, to just make Node's fields public. Since Node is such a small class and it is private to the LList class, this will not cause any problems.

The LList class actually has only one class variable, a LLNode object called front, which represents the first element in the list. This is all we need. If we want anything further down in the list, we just start at front and keep following links until we get where we want.

Here is the LList constructor:

```java
public LList() {
    front = null;
}
```
CHAPTER 2. LISTS

The isEmpty, size, and contains methods

Determining if the list is empty is as simple as checking whether the front of the list is set to null.

```java
public boolean isEmpty()
{
    return front==null;
}
```

To find the size of the list, we walk through the list, following links and keeping count, until we get to the end of the list, which is marked by a null link.

```java
public int size()
{
    int count = 0;
    Node n = front;
    while (n!=null)
    {
        count++;
        n = n.link;
    }
    return count;
}
```

Determining if a value is in the list can be done similarly:

```java
public boolean contains(int value)
{
    Node n = front;
    while (n!=null)
    {
        if (n.data==value)
            return true;
        n = n.link;
    }
    return false;
}
```

It is worth stopping to understand how we are looping through the list, as most of the other linked list methods will use a similar kind of loop. We start at the front node and move through the list with the line n=n.link, which moves to the next element. We know we have moved past the end of the list when our node is equal to null.

To picture a linked list, imagine a phone chain. Phone chains aren’t in use much anymore, but the way they used to work is if school or work was canceled because of a snowstorm, the person in charge would call someone and tell them about the cancelation. That person would then call the next person in the chain and tell them. Then that person would call the next person, etc. The process of each person dialing the other person’s number and calling them to tell the message is akin to the process we follow in our loops of following links.

And if we imagine that each person in the phone chain only has the next person’s phone number, we have the same situation as a linked list, where each node only knows the location of the next node. This is different from arrays, where we know the location of everything. An array would be the equivalent of everyone having phone numbers in order, like 345-0000, 345-0001, 345-0002, and looping through
the array would be the equivalent of having a robo-dialer call each of the numbers in order, giving the
message to each person.

**Getting and setting elements**

Getting and setting elements at a specific index is trickier with linked lists than with dynamic list lists.
To get an element we have to walk the list, link by link, until we get to the node we want. We use a
count variable to keep track of how many nodes we've visited. We will know that we have reached our
desired node when the count equals the desired index. Here is the `get` method:

```java
public int get(int index)
{
    int count = 0;
    Node n = front;
    while (count<index)
    {
        count++;
        n = n.link;
    }
    return n.data;
}
```

Setting an element is similar.

```java
public void set(int index, int value)
{
    int count = 0;
    Node n = front;
    while (count<index)
    {
        count++;
        n = n.link;
    }
    n.data = value;
}
```

**Adding things to the front and back of the list**

Adding an element to the front of a linked list is very quick. We create a new node that contains the
value and point it to the front of the list. This node then becomes our new front, so we set `front`
equal to it. We can do this all in one line:

```java
public void addToFront(int value)
{
    front = new Node(value, front);
}
```

To append to a linked list, we create a new node, walk the list until we get to the end, and set the end
node's link to point to the new node. If the list is empty, this won't work, so we instead just set `front`
equal to the new node.
public void append(int value) {
    if (front==null) {
        front = new Node(value, null);
        return;
    }

    Node n = front;
    while (n.link != null)
        n = n.link;
    n.link = new Node(value, null);
}

We see that adding to the front is $O(1)$, while appending is $O(n)$. We could make appending into $O(1)$ if we have another class variable, back, that keeps track of the last node in the list. This would add a few lines of code to some of the other methods as we would occasionally need to modify back.

Deleting a node

To delete an index, we reroute a few links as shown below.

We first have to get to the node before the one to be deleted. We do this with a loop like we used for the get and set methods, except that we stop right before the deleted node. Once we get there, we can use the following line to reroute around the deleted node:

    n.link = n.link.link;

This sets the current node’s link to point to its neighbor’s link, essentially removing its neighbor from the list. Here is the code of the method. Note the special case for deleting the first element of the list.

    public void delete(int index) {
        if (index==0) {
            front = front.link;
            return;
        }

        int count = 0;
        Node n = front;
        while (count<index-1) {
            count++;
            n = n.link;
        }

        n.next = n.next.next;  // Now remove the node next to it
        }
One other thing that is worth mentioning is that the deleted node is still there in memory but we don’t have any way to get to it, so it is essentially gone and forgotten about. Eventually Java’s garbage collector will recognize that nothing is pointing to it and recycle its memory for future use.

**Inserting a node**

To insert a node, we first create the node to be inserted and then reroute two links to put it into the list. The rerouting code is very similar to what we did for deleting a node. Here is how the rerouting works:

```java
n.link = new Node(value, n.link);
```

Here is the code for the `insert` method.

```java
public void insert(int index, int value)
{
    if (index==0)
    {
        front = new Node(value, front);
        return;
    }

    int count = 0;
    Node n = front;
    while (count<index-1)
    {
        count++;
        n = n.link;
    }
    n.link = new Node(value, n.link);
}
```

**The entire LList class**

Here is the entire linked list class:
public class LList
{
  private class Node
  {
    public int data;
    public Node link;

    public Node(int data, Node link)
    {
      this.data = data;
      this.link = link;
    }
  }

  private Node front;

  public LList()
  {
    front = null;
  }

  public boolean isEmpty()
  {
    return front == null;
  }

  public int size()
  {
    int count = 0;
    Node n = front;
    while (n != null)
    {
      count++;
      n = n.link;
    }
    return count;
  }

  public int get(int index)
  {
    int count = 0;
    Node n = front;
    while (count < index)
    {
      count++;
      n = n.link;
    }
    return n.data;
  }

  public void set(int index, int value)
  {
    int count = 0;
    Node n = front;
    while (count < index)
    {
      count++;
      n = n.link;
    }
    n.data = value;
  }

  public boolean contains(int value)
  {
    return true;
  }
}
Node n = front;
while (n!=null)
{
    if (n.data==value)
        return true;
    n = n.link;
}
return false;

public String toString()
{
    if (front==null)
        return "[]";

    Node n = front;
    String s = "[";
    while (n != null)
    {
        s += n.data + "", ";
        n = n.link;
    }
    return s.substring(0,s.length()-2)+"]";
}

public void append(int value)
{
    if (front==null)
    {
        front = new Node(value, null);
        return;
    }

    Node n = front;
    while (n.link != null)
        n = n.link;
    n.link = new Node(value, null);
}

public void addToFront(int value)
{
    front = new Node(value, front);
}

public void delete(int index)
{
    if (index==0)
    {
        front = front.link;
        return;
    }

    int count = 0;
    Node n = front;
    while (count<index-1)
    {
        count++;
        n = n.link;
    }
    n.link = n.link.link;
}

public void insert(int index, int value)
{
    if (index==0)
Null pointer exceptions

When working with nodes, there is a good chance of making a mistake and getting a NullPointerException from Java. This usually happens if you use an object without first initializing it. For example, the following will cause a null pointer exception:

```java
Node myNode;
// some code not involving myNode might go here
if (myNode.value==3)
    // some more code goes here...
```

The correction is to either set myNode equal to some other Node object or initialize it like below:

```java
Node myNode = new Node();
```

In summary, if you get a null pointer exception, it usually indicates an object that has been declared but not initialized.

Doubly- and circularly-linked lists

A **doubly-linked list** is one where each node has two links, one to the next node and one to the previous node. On the one hand, this can be useful for methods that need to know the predecessor of a node. On the other hand, some of the list methods become a little more complex because we now have to keep track of predecessors. The backwards link is also useful for iterating through the list in reverse order.

A **circularly-linked list** is one where the link of the last node in the list points back to the start of the list.

2.5 Comparison of dynamic arrays and linked lists

First, let’s look at running times of the methods. Let \( n \) denote the size of the list.
1. Getting/setting a specific index

*Dynamic array* — $O(1)$: Getting and setting both consist of a single array operation, which translates to a quick memory address calculation at the machine level. The size of the list has no effect on the calculation.

*Linked list* — $O(n)$: To get or set a specific index requires us to follow links until we get to the desired index. In the worst case, we have to go through all $n$ links (when the index is the end of the list), and on average we have to go through $n/2$ links.

2. `isEmpty`

*Dynamic array* — $O(1)$: We just check if the class variable `numElements` is 0 or not.

*Linked list* — $O(1)$: We just check if the class variable `front` is equal to `null` or not.

3. `size`

*Dynamic array* — $O(1)$: We just return the value of the class variable `numElements`. If we did not have that class variable, then it would be an $O(n)$ operation because we would have to loop through and count how many elements there are. The tradeoff here is that the `insert`, `delete`, and `append` methods have to update `numElements` each time they are called, adding a little bit of time to each of them. Essentially, by having the class variable, we have spread out the `size` method's work over all those other methods.

*Linked list* — $O(n)$: Our code must loop through the entire list to count how many elements there are. On the other hand, we could make this $O(1)$ by adding a class variable `numElements` to the list and updating it whenever we add or delete elements, just like we did for our dynamic array class.

4. `contains`

*Dynamic array* — $O(n)$: Our code searches through the list, element-by-element, until it finds what it's looking for or gets to the end of the list. If the desired value is at the end of the list, this will take $n$ steps. On average, it will take $n/2$ steps.

*Linked list* — $O(n)$: Our linked-list code behaves similarly to the dynamic list code.

5. Adding and deleting elements

*Dynamic array* — First, increasing the capacity of the list is an $O(n)$ operation, but this has to be done only rarely. Appending to the list is an $O(1)$ operation. So overall, appending is said to be *amortized* $O(1)$, which is to say that most of the time it appending is an $O(1)$ operation, but it can on rare occasions be $O(n)$.

On the other hand, inserting or deleting an element in the middle of the list is $O(n)$ on average because we have to move around big chunks of the array either to make room or to make sure the elements stay contiguous.

*Linked list* — The actual operation of inserting or deleting a node is an $O(1)$ operation as it just involves moving around one or two links. However, we have to walk through the list to get to the location where the insertion or deletion happens, and this takes $O(n)$ operations on average. So, the way we've implemented things, a single insert or delete will run in $O(n)$ time.
However, once you're at the proper location, inserting or deleting is very quick. Consider a `removeAll` method that removes all occurrences of a specific value. To implement this we could walk through the list, deleting each occurrence of the value as we encounter it. Since we are already walking through the list, no further searching is needed to find a node and the deletions will all be $O(1)$ operations. So `removeAll` will run in $O(n)$ time.

Also, inserting or deleting at the front of the list is an $O(1)$ operation as we don't have to go anywhere to find the front. And, as noted earlier, appending to a list is currently $O(n)$ because we have to walk through the entire list to get there, but it can be made into $O(1)$ operation if we add another class variable to keep track of the end of the list.

In terms of which is better—dynamic arrays or linked lists—there is no definite answer as they each have their uses. In fact, for most applications either will work fine. For applications where speed is critical, dynamic arrays are better when a lot of random access is involved, whereas linked lists can be better when it comes to insertions and deletions. In terms of memory usage, either one is fine for most applications. For really large lists, remember that linked lists require extra memory for all the links, an extra five or six bytes per list element. Dynamic arrays have their own potential problems in that not every space allocated for the array is used and for really large arrays, finding a contiguous block of memory could be tricky.

We also have to consider cache performance. Not all RAM is created equal. RAM access can be slow, so modern processors include a small amount cache memory that is faster to access than regular RAM. Arrays, occupying contiguous chunks of memory, generally give better cache performance than linked lists whose nodes can be spread out in memory. Chunks of the array can be copied into the cache, making it faster to iterate through a dynamic array than a linked lists.

### 2.6 Making the linked list class generic

The linked list class we created in this chapter works only for integer lists. We could modify it to work with strings by going through and replacing the `int` declarations with `String` declarations, but that would be tedious. And then if we wanted to modify it to work with doubles or longs or some object, we would have to do the same tedious replacements all over again. It would be nice if we could modify the class once so that it works with any type. This is where Java's generics come in.

The process of making a class work with generics is pretty straightforward. We will add a little syntax to the class to indicate we are using generics and then replace the `int` declarations with `T` declarations. The name `T`, short for “type,” acts as a placeholder for a generic type. (You can use other names instead of `T`, but the `T` is somewhat standard.)

First, we have to introduce the generic type in the class declaration:

```java
public class LList<T>
```

The slanted brackets are used to indicate a generic type. After that, the only thing we have to do is anywhere we refer to the data type of the values stored in the list, we have to change it from `int` to `T`. 
For example, the get method changes as shown below:

```java
public T get(int index) {
    int count = 0;
    Node n = front;
    while (count<index)
    {
        count++;
        n = n.link;
    }
    return n.data;
}
```

Notice that the return type of the method is now type T instead of int. Notice also that index stays an int. Indices are still integers, so they don’t change. It’s only the data type stored in the list that changes to type T.

Here is the entire class:

```java
public class LList<T> {
    private class Node {
        public T data;
        public Node link;
    }
    private Node front;
    public LList() {
        front = null;
    }
    public boolean isEmpty() {
        return front==null;
    }
    public int size() {
        int count = 0;
        Node n = front;
        while (n!=null)
        {
            count++;
            n = n.link;
        }
        return count;
    }
    public T get(int index) {
        int count = 0;
        Node n = front;
        while (count<index)
        {
            count++;
            n = n.link;
        }
        return n.data;
    }
}
```
```java
{  
    count++;  
    n = n.link;
}
return n.data;
}

public void set(int index, T value)  
{  
    int count = 0;  
    Node n = front;  
    while (count<index)  
    {  
        count++;  
        n = n.link;
    }  
    n.data = value;
}

public boolean contains(T value)  
{  
    Node n = front;  
    while (n!=null)  
    {  
        if (n.data==value)  
            return true;
        n = n.link;
    }  
    return false;
}

public String toString()  
{  
    if (front==null)  
        return "[]";
    Node n = front;  
    String s = "[";  
    while (n !=null)  
    {  
        s += n.data + ", ";
        n = n.link;
    }  
    return s.substring(0, s.length()-2)+"]";
}

public void append(T value)  
{  
    if (front==null)  
    {  
        front = new Node(value, null);
        return;
    }  
    Node n = front;  
    while (n.link != null)  
    {  
        n = n.link;
        n.link = new Node(value, null);
    }
}

public void addToFront(T value)  
{  
    front = new Node(value, front);
}
```
2.7 The Java Collections Framework

While it is instructive to write our own list classes, it is best to use the ones provided by Java for most practical applications. Java's list classes have been extensively tested and optimized, whereas ours are mostly designed to help us understand how the two types of lists work. On the other hand, it's nice to know that if we need something slightly different from what Java provides, we could code it.

Java's Collections Framework contains, among many other things, an interface called List. There are
two classes implementing this interface: a dynamic array class called \texttt{ArrayList} a linked list class called \texttt{LinkedList}.

Here are some sample declarations:

\begin{verbatim}
List<Integer> list = new ArrayList<Integer>();
List<Integer> list = new LinkedList<Integer>();
\end{verbatim}

The \texttt{Integer} in the slanted braces indicates the data type that the list will hold. This is an example of Java generics. We can put any class name here. Here are some examples of other types of lists we could have:

\begin{verbatim}
List<String> — list of strings
List<Double> — list of doubles
List<Card> — list of Card objects (assuming you've created a Card class)
List<List<Integer>> — list of integer lists
\end{verbatim}

Here is an example of lists in action. The following program contains a method \texttt{getPrimeList} that returns a list of the primes less than the parameter \(n\).

\begin{verbatim}
import java.util.ArrayList;
import java.util.List;

public class Primes
{
    public static void main(String[] args)
    {
        System.out.println(getPrimeList(80));
    }

    public static List<Integer> getPrimeList(int n)
    {
        List<Integer> primes = new ArrayList<Integer>();
        primes.add(2);
        for (int i=3; i<=n; i+=2)
        {
            boolean prime = true;
            for (int j=2; j<Math.sqrt(n); j++)
            {
                if (i%j==0)
                {
                    prime=false;
                    break;
                }
            }
            if (prime)
                primes.add(i);
        }
        return primes;
    }
}
\end{verbatim}

Here is the output of the program:

\begin{verbatim}
[2, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79]
\end{verbatim}
List methods

Here are the most useful methods of the List interface. They are available to both ArrayLists and LinkedLists.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)</td>
<td>appends x to the list</td>
</tr>
<tr>
<td>add(i, x)</td>
<td>adds x to the list at index i</td>
</tr>
<tr>
<td>clear()</td>
<td>deletes everything in the list</td>
</tr>
<tr>
<td>clone()</td>
<td>makes a shallow copy of the list</td>
</tr>
<tr>
<td>contains(x)</td>
<td>returns true iff x is in the list</td>
</tr>
<tr>
<td>containsAll(a)</td>
<td>returns true iff the entire collection a is in the list</td>
</tr>
<tr>
<td>equals(a)</td>
<td>returns true iff the list equals a</td>
</tr>
<tr>
<td>get(i)</td>
<td>returns the value at index i</td>
</tr>
<tr>
<td>indexOf(x)</td>
<td>returns the first index (location) of x in the list</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>returns true iff the list is empty</td>
</tr>
<tr>
<td>lastIndexOf(x)</td>
<td>like indexOf, but returns the last index</td>
</tr>
<tr>
<td>remove(i)</td>
<td>removes the item at index i</td>
</tr>
<tr>
<td>remove(x)</td>
<td>removes the first occurrence of the object x</td>
</tr>
<tr>
<td>removeAll(x)</td>
<td>removes all occurrences of x</td>
</tr>
<tr>
<td>retainAll(a)</td>
<td>removes everything in the list that is not in the collection a</td>
</tr>
<tr>
<td>set(i, x)</td>
<td>sets the value at index i to x</td>
</tr>
<tr>
<td>size()</td>
<td>returns the number of elements in the list</td>
</tr>
<tr>
<td>subList(i, j)</td>
<td>returns a slice of a list from index i to j-1, sort of like substring</td>
</tr>
<tr>
<td>toString()</td>
<td>returns a string representation of the list</td>
</tr>
<tr>
<td>toArray()</td>
<td>returns an array representation of the list</td>
</tr>
</tbody>
</table>

A few notes about the methods:

1. The `add` method is useful for filling up a list. Here is a simple example that fills up a list with random integers.

   ```java
   List<Integer> list = new ArrayList<Integer>();
   Random random = new Random();
   for (int i=0; i<10000; i++)
       list.add(Random.nextInt(100));
   ```

2. The copy that is made by the `clone` method is a shallow copy. This means that if we clone a list of objects, the clone will not contain copies of the objects, but rather it will contain references to
the objects from the original list. So if we change one of the objects in the clone, the change will affect the original list as well, since they essentially share the same elements.

A lot of times as an alternative to clone you can make a shallow copy of a list (say an ArrayList of integers called list) like this:

```java
List<Integer> copy = new ArrayList<Integer>(list);
```

The opposite of a shallow copy is a deep copy. A deep copy would make copies of all the objects in the list so that the cloned list would have its copies, separate from the original list's objects. Unfortunately, there is no deep copy method in the Collections Framework. There are, however, ways to implement a deep copy. Here is a simple example. Say we have a card class called Card with fields suit and value. Here is a deep copy method using the class's getters and constructor:

```java
public ArrayList<Card> deepCopy(ArrayList<Card> cards)
{
    ArrayList<Card> copy = new ArrayList<Card>();
    for (Card card:cards)
        copy.add(new Card(card.getValue(), card.getSuit()));
    return copy;
}
```

Another option would be to use the Card class's clone method (if it has one). The copy.add line would change to the following:

```java
copy.add(card.clone());
```

3. The equals method is used to check if two lists have the exact same elements in the same order. The == operator can only check if two lists occupy the same place in memory.

4. The get and set methods are used to get and set specific locations in the list. Where we would use a[i] to get the item at index i in an array, we use list.get(i) to get the item at index i in a list.

5. A note about the remove method: Say we're working with a list a of integers. If we call a.remove(3), does it remove the first occurrence of the integer 3 in the list or does it remove the integer at index 3 in the list? It turns out to be the latter. If we want the former, we must use a.remove((Integer)3). Basically, if remove is called with an int argument, it assumes the argument is an index, while if it is called with an object (especially an Integer object), then it removes the first occurrence of that object.

6. The Java documentation points out a nice way to use the subList and clear methods to delete part of a list:

```java
list.subList(i, j).clear();
```

7. The toString method works together with print statements to print out the list. For example, the following are equivalent:

```java
System.out.println(list);
System.out.println(list.toString());
```
8. The ArrayList class also has an additional method called `ensureCapacity` that sets the capacity of an ArrayList. If you know how many elements should be in your list, setting the capacity ahead of time removes the need for the ArrayList to have to resize itself when it runs out of room. The capacity can also be specified in the ArrayList constructor, like below:

   ```java
   List<Integer> list = new ArrayList<Integer>(100);
   ```

   Just be careful, though. Even if you specify an initial capacity, the initial list size is 0 because there is nothing in the list to start.

9. Java's LinkedList class has additional methods like `addFirst`, `addLast`, `push`, and `pop` that are useful for implementing stacks and queues (see Chapter 3).

### Methods of `java.util.Collections`

There are some useful static Collections methods that operate on lists (and, in some cases, other Collections objects that we will see later):

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addAll(a, x1,..., xn)</code></td>
<td>adds $x_1, x_2, ..., x_n$ to the list $a$</td>
</tr>
<tr>
<td><code>binarySearch(a,x)</code></td>
<td>returns <code>true</code> iff x is in the collection $a$</td>
</tr>
<tr>
<td><code>copy(a,b)</code></td>
<td>copies everything from $a$ into $b$</td>
</tr>
<tr>
<td><code>fill(a,x)</code></td>
<td>sets every element of $a$ to $x$</td>
</tr>
<tr>
<td><code>frequency(a,x)</code></td>
<td>returns a count of how many times $x$ occurs in $a$</td>
</tr>
<tr>
<td><code>indexOfSubList(a,b)</code></td>
<td>returns the index of the first occurrence of collection $b$ in $a$</td>
</tr>
<tr>
<td><code>lastIndexOfSubList(a,b)</code></td>
<td>like <code>indexOfSubList</code>, but returns the last occurrence</td>
</tr>
<tr>
<td><code>max(a)</code></td>
<td>returns the largest element in $a$</td>
</tr>
<tr>
<td><code>min(a)</code></td>
<td>returns the smallest element in $a$</td>
</tr>
<tr>
<td><code>replaceAll(a,x,y)</code></td>
<td>replaces all occurrences of $x$ in $a$ with $y$</td>
</tr>
<tr>
<td><code>reverse(a)</code></td>
<td>reverses the elements of $a$</td>
</tr>
<tr>
<td><code>rotate(a,n)</code></td>
<td>rotates all the elements of $a$ forward $n$ positions</td>
</tr>
<tr>
<td><code>shuffle(a)</code></td>
<td>puts the contents of $a$ in a random order</td>
</tr>
<tr>
<td><code>sort(a)</code></td>
<td>sorts the collection $a$</td>
</tr>
<tr>
<td><code>swap(a,i,j)</code></td>
<td>swaps elements at indices $i$ and $j$</td>
</tr>
</tbody>
</table>

A few notes about these methods:

1. The `addAll` method can take an arbitrary number of arguments. It is useful for quickly adding some initial data to a list. Here is an example:
List<Integer> primes = new ArrayList<Integer>();
Collections.addAll(primes, 2, 3, 5, 7, 11, 13, 17, 19);

2. The binarySearch method requires the collection to be sorted because of the way binary search works. If the collection is not (or cannot be) sorted, then use its contains method, may be slower than binarySearch.

3. The fill method is useful for quickly setting everything in a list equal to something, like zeroing out all the elements of a list of counts. Unfortunately, it cannot be used to initialize the elements of a list all to the same value because when a list is created, it is initially empty. The fill method only changes values; it doesn't add anything new to the list.

4. If the second parameter of the rotate method is negative, the collection will be rotated backwards.

5. The shuffle method is very useful. For instance, a quick way to come up with a random ordering of players for a game is shown below:

```
List<String> players = new ArrayList<String>();
Collections.addAll(players, "Andrew", "Bob", "Carol", "Deb", "Erin");
Collections.shuffle(players);
System.out.println(players);
```

6. We can specify a comparator as a second parameter to the sort method in order to sort according to some other criterion, like sorting strings by length instead of alphabetically. See Section 9.12.

**Iterating**

We can use a foreach loop to iterate through the items of a list. For example, suppose list is a list of strings. The following code prints all the strings in the list that start with the letter 'a':

```
for (String s:list)
  if (s.charAt(0)=='a')
    System.out.println(s);
```

This is much easier to read than the more verbose way using an ordinary for loop:

```
for (int i=0; i<list.size(); i++)
  if (list.get(i).charAt(0)=='a')
    System.out.println(list.get(i));
```

**Working with arrays**

Most of the time it is easier to work with lists than with arrays. However, if you find yourself working with arrays, java.util.Arrays has a few useful methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asList(x1,...,xn)</td>
<td>converts the array ([x_1,...,x_n]) into a list</td>
</tr>
</tbody>
</table>
# 2.8 Applications of lists

It’s hard to write a program of any length that doesn’t use lists (or arrays) somewhere. Lists of names, words, numbers, and so on are so common in everyday life that it’s no surprise that they show up in programs. Exercise 3 is about some practical applications of lists.

One exercise in particular demonstrates the usefulness of lists. Here is the statement of the problem:

Write a program that asks the user to enter a length in feet. The program should then give the user the option to convert from feet into inches, yards, miles, millimeters, centimeters, meters, or kilometers. Say if the user enters a 1, then the program converts to inches, if they enter a 2, then the program converts to yards, etc.

While this could certainly be written using if statements or a switch statement, if there are a lot of conversions to make, the code will be long and messy. The code can be made short and elegant by storing all the conversion factors in a list. This would also make it easy to add new conversions as we would just have to add some values to the list, as opposed to copying and pasting chunks of code. Moreover, the conversion factors could be put into a file and read into the list, meaning that the program

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarySearch(a, x)</td>
<td>returns <code>true</code> iff <code>x</code> is in the sorted array <code>a</code></td>
</tr>
<tr>
<td>copyOf(a, n)</td>
<td>returns a copy of <code>a</code> that is <code>n</code> elements long.</td>
</tr>
<tr>
<td>copyOfRange(a, m, n)</td>
<td>returns a copy of all the elements of <code>a</code> from index <code>m</code> to index <code>n-1</code></td>
</tr>
<tr>
<td>deepToString()</td>
<td>like <code>toString</code> but for printing out multidimensional arrays</td>
</tr>
<tr>
<td>equals(a, b)</td>
<td>returns <code>true</code> iff arrays <code>a</code> and <code>b</code> are equal</td>
</tr>
<tr>
<td>fill(a, x)</td>
<td>sets every element of <code>a</code> to <code>x</code></td>
</tr>
<tr>
<td>sort(a)</td>
<td>sorts the array <code>a</code> in place</td>
</tr>
<tr>
<td>toString(a)</td>
<td>returns a string representation of the array <code>a</code></td>
</tr>
</tbody>
</table>
itself need not be modified at all. This way, our users (who may not be programmers) could easily add conversions.

This is of course a really simple example, but it hopefully demonstrates the main point—that lists can eliminate repetitious code and make programs easier to maintain.

2.9 Iterators*

This section can be skipped. It is about how to make a class work with Java’s foreach loop.

We can add this feature to our own classes, like our linked list class. To start, we need the following import statement.

```java
import java.util.Iterator;
```

Next, our class needs to implement the `Iterable` interface.

```java
public class LList<T> implements Iterable<T>
```

We then create a private nested class that tells how to traverse the list. This class implements the `Iterator` interface. It must have methods `hasNext`, `next`, and `remove`. Its code is shown below.

```java
private class LListIterator implements Iterator<T> {
    private Node location;

    public LListIterator(LList<T> d) {
        location = front;
    }

    public T next() {
        T value = location.data;
        location = location.link;
        return value;
    }

    public boolean hasNext() {
        return location!=null;
    }

    public void remove() {}
}
```

There is a class variable `location` that keeps track of where in the list we are. The constructor sets up the iterator and starts it at the front of the list. The `next` method returns the next element in the list and advances the iterator. The `hasNext` method returns `true` or `false` depending on whether there is anything left in the list to be iterated over. To do this we just check if `location` is `null`. The `remove` method is part of the interface, so we must include it, but we have decided not to do anything with it.

The one other thing we need is a constructor for the iterator:
public Iterator<T> iterator()
{
    return new LListIterator(this);
}

Here is the linked list class, complete with the iterator:

```java
import java.util.Iterator;

public class LList<T> implements Iterable<T>
{
    private class LListIterator implements Iterator<T>
    {
        private Node location;
        public LListIterator(LList<T> d)
        {
            location = front;
        }

        public T next()
        {
            T value = location.data;
            location = location.link;
            return value;
        }

        public boolean hasNext()
        {
            return location!=null;
        }

        public void remove() {}
    }

    public Iterator<T> iterator()
    {
        return new LListIterator(this);
    }

    private class Node
    {
        public T data;
        public Node link;
        public Node(T data, Node link)
        {
            this.data = data;
            this.link = link;
        }
    }

    private Node front;

    public LList()
    {
        front = null;
    }

    public boolean isEmpty()
    {
        return front==null;
    }
```


public int size()
{
    int count = 0;
    Node n = front;
    while (n!=null)
    {
        count++;
        n = n.link;
    }
    return count;
}

public T get(int index)
{
    int count = 0;
    Node n = front;
    while (count<index)
    {
        count++;
        n = n.link;
    }
    return n.data;
}

public void set(int index, T value)
{
    if (front==null)
        front = new Node(value, null);

    int count = 0;
    Node n = front;
    while (count<index)
    {
        count++;
        n = n.link;
    }
    n.data = value;
}

public boolean contains(T value)
{
    Node n = front;
    while (n!=null)
    {
        if (n.data==value)
            return true;
        n = n.link;
    }
    return false;
}

public String toString()
{
    if (front==null)
        return "[]";

    Node n = front;
    String s = "[";
    while (n != null)
    {
        s += n.data + ", ";
        n = n.link;
    }
    return s.substring(0, s.length()-2) + "]";
}
public void append(T value) {
    if (front == null) {
        front = new Node(value, null);
        return;
    }

    Node n = front;
    while (n.link != null)
        n = n.link;

    n.link = new Node(value, null);
}

public void addToFront(T value) {
    front = new Node(value, front);
}

public void delete(int index) {
    if (index == 0) {
        front = front.link;
        return;
    }

    int count = 0;
    Node n = front;
    while (count < index - 1)
        count++;
        n = n.link;

    n.link = n.link.link;
}

public void insert(int index, T value) {
    if (index == 0) {
        front = new Node(value, front);
        return;
    }

    int count = 0;
    Node n = front;
    while (count < index - 1)
        count++;
        n = n.link;

    Node newNode = new Node(value, n.link);
    n.link = newNode;
}

We could then use the class as below:

LList<Integer> list = new LList<Integer>();
list.append(2);
list.append(3);
for (Integer x:list)
    System.out.println(x);

### 2.10 Exercises

1. Add the following methods to the dynamic array class from this chapter.
   
   (a) addAll(a) — Adds all the elements of the array a to the end of the list.
   (b) addSorted(x) — Assuming the list is in sorted order, this method adds x to the list in the appropriate place so that the list remains sorted.
   (c) allIndicesOf(x) — Returns an integer ArrayList consisting of every index in the linked-list where x occurs.
   (d) clear() — Empties all the elements from the list.
   (e) count(x) — Returns the number of occurrences of x in the list.
   (f) equals(L) — Returns true if the list is equal to L and false otherwise. Lists are considered equal if they have the same elements in the same order.
   (g) indexOf(x) — Returns the first index of x in the list and -1 if x is not in the list.
   (h) lastIndexOf(x) — Returns the last index of x in the list and -1 if x is not in the list.
   (i) remove(x) — Removes the first occurrence of x from the list.
   (j) removeAll(x) — Removes all occurrences of x from the list.
   (k) removeDuplicates() — Removes all duplicate items from the list so that each element in the list occurs no more than once
   (l) reverse() — Reverses the elements of the list
   (m) rotate() — Shifts the elements of the list so that the last item in the list is now first and everything else is shifted right by one position.
   (n) rotate(n) — Like above but every elements shifts forward by n units.
   (o) sublist(i,j) — Returns a new list consisting of the elements of the original list starting at index i and ending at index j-1.

2. Add the methods above to the linked list class. Try to implement them as efficiently as possible, creating as few new nodes as possible and instead relying on moving around links.

3. Use lists from the Collections Framework to do the following:
   
   (a) Write a method primes that takes an integer n and returns a list of all the primes less than or equal to n.
   (b) Write a method called merge that takes two sorted lists and merges them together into a single, sorted list.
2.10. EXERCISES

(c) Write a program that asks the user to enter a length in feet. The program should then give the user the option to convert from feet into inches, yards, miles, millimeters, centimeters, meters, or kilometers. Say if the user enters a 1, then the program converts to inches, if they enter a 2, then the program converts to yards, etc. While this can be done with if statements, it is much shorter with lists and it is also easier to add new conversions if you use lists.

(d) Write a method that takes and list list containing integers between 1 and 100 and returns a new list whose first element is how many ones are in list, whose second element is how many twos are in list, etc.

(e) Write a method getRandom that takes a list and returns a random element from the list.

(f) Write a method sample(list, n) that returns a list of n random elements from list.

(g) Write a method breakIntoTeams that takes a list containing an even number of people's names, randomly breaks all the people into teams of two, and returns a list of the teams.

(h) Write a program that estimates the average number of drawings it takes before the user's numbers are picked in a lottery that consists of correctly picking six different numbers that are between 1 and 10. To do this, run a loop 1000 times that randomly generates a set of user numbers and simulates drawings until the user's numbers are drawn. Find the average number of drawings needed over the 1000 times the loop runs.

(i) Write a program that simulates drawing names out of a hat. In this drawing, the number of hat entries each person gets may vary. Allow the user to input a list of names and a list of how many entries each person has in the drawing, and print out who wins the drawing.

(j) Write a method runLengthEncode that performs run-length encoding on a list. It should return a list consisting of all lists of the form [x,n], where x is an element from the list and n is its frequency. For example, the encoding of the list [a,a,b,b,b,c,c,c,d] is [[2,a],[4,b],[3,c],[1,d]].

(k) Write a simple quiz program that reads a bunch of questions and answers from a file and randomly chooses five questions. It should ask the user those questions and print out the user's score.

4. Following what we did in Section 2.9, do the same for the ALi st class.

5. Add a field called back to the LLi st class that keeps track of the back of the list. To do this properly, you will have to modify many of the methods to keep track of the new field. Use this new field to write a more efficient append method.

6. Implement a doubly-linked list, giving it the same methods as the LLi st class.

7. Implement a circularly-linked list, giving it the same methods as the LLi st class.

8. Add exception-handling to the ALi st class. The class should throw exceptions if the user tries to access an index outside of the list.
Chapter 3

Stacks and Queues

This chapter is about two of the simplest and most useful data structures—stacks and queues. Stacks and queues behave like lists of items, each with its own rules for how items are added and removed.

A stack is a LIFO structure, short for last in, first out. Think of a stack of dishes, where the last dish placed on the stack is the first one to be taken off. Or as another example, think of a discard pile in a game of cards where the last card put on the pile is the first one to be picked. Stacks have two main operations: push and pop. The push operation puts a new element on the top of the stack and the pop element removes an element from the top of the stack and returns it to the caller.

To demonstrate how a stack works, suppose we have a stack of integers. Say we push 1, 2, and 3 onto the stack, then do a pop operation, then push 4 onto the stack. The stack will grow as follows, where the left side of each list is the “top” of the stack:

\[
\begin{align*}
[] & \xrightarrow{\text{push } 1} [1] \\
[1] & \xrightarrow{\text{push } 2} [2, 1] \\
[2, 1] & \xrightarrow{\text{push } 3} [3, 2, 1] \\
[3, 2, 1] & \xrightarrow{\text{pop}} [2, 1] \\
[2, 1] & \xrightarrow{\text{push } 4} [4, 2, 1]
\end{align*}
\]

A queue is a FIFO structure, short for first in, first out. A queue is like a line (or queue) at a store. The first person in line is the first person to check out. If a line at a store were implemented as a stack, on the other hand, the last person in line would be the first to check out, which would lead to lots of angry customers. Queues have two main operations, enqueue and dequeue. The enqueue operation adds a new element to the back of the queue, and the dequeue operation removes the element at the front of the queue and returns it to the caller.

To demonstrate a queue, consider a queue of integers and say we do a similar sequence of operations to what we showed with stacks. We will enqueue 1, 2, and 3, then dequeue, and finally enqueue 4. The queue grows as follow, where the left side of each list is the “front” of the queue.

\[
\begin{align*}
[] & \xrightarrow{\text{push } 1} [1] \\
[1] & \xrightarrow{\text{enqueue } 2} [1, 2] \\
[1, 2] & \xrightarrow{\text{enqueue } 3} [1, 2, 3] \\
[1, 2, 3] & \xrightarrow{\text{dequeue}} [2, 3] \\
[2, 3] & \xrightarrow{\text{enqueue } 4} [2, 3, 4]
\end{align*}
\]
3.1 A stack class

Our class will have a push method that puts an item on the top of the stack and a pop method that removes and returns the value of the top item on the stack. It will also have a peek method, which returns the value of the top item without removing it from the stack, and an isEmpty method.

We implement the stack as a linked list. The class will have an inner class called Node, just like with linked lists. We will have one class variable, a node called top to keep track of what is at the top of the stack. The constructor sets it to null, so we start with an empty stack. Also, we will use generics to allow us to use any datatype with our stack. Here is the start of the class:

```java
public class LLStack<T> {
    private class Node {
        public T data;
        public Node link;
        public Node(T data, Node link) {
            this.data = data;
            this.link = link;
        }
    }
    private Node top;
    public LLStack() {
        top = null;
    }
}
```

The isEmpty and peek methods

The isEmpty method is nearly the same as it is for linked lists:

```java
public boolean isEmpty() {
    return top==null;
}
```

The peek method is also very short as we just have to return the data value stored at the top of the stack.

```java
public T peek() {
    return top.data;
}
```

The method also should make sure that the stack is not empty and raise an exception if it is, but for simplicity, we will not do that.
3.1. A STACK CLASS

Pushing and popping things

Pushing things onto and popping things off of the stack is a matter of moving around links and where the top variable points to. See the figure below:

To push an item onto the stack, we create a new node for it that points to the current top and then change the top variable to point to the new node. This very similar to the addToFront method for a linked list, and it can be done in one line:

```java
public void push(T value)
{
    top = new Node(value, top);
}
```

To pop an item from the stack, we have to both return the value of the item and delete it. To delete the top item, we set top equal to top.link. This points the top variable to the second item from the top (or null if there is only one element). But before we do this, we save the value of the old top item so we can later return it.

```java
public T pop()
{
    T data = top.data;
    top = top.link;
    return data;
}
```

Here is the entire stack class:

```java
public class LLStack<T>
{
    private class Node
    {
        public T data;
        public Node link;

        public Node(T data, Node link)
        {
            this.data = data;
            this.link = link;
        }
    }
}
```
private Node top;

public LLStack()
{
    top = null;
}

public boolean isEmpty()
{
    return top==null;
}

public T peek()
{
    return top.data;
}

public void push(T value)
{
    top = new Node(value, top);
}

public T pop()
{
    T data = top.data;
    top = top.link;
    return data;
}

Here is a short demonstration of the class in use:

```
LLStack<Integer> stack = new LLStack<Integer>();
stack.push(3);
stack.push(4);
stack.push(5);
System.out.print(stack.pop() + " " + stack.pop() + " " + stack.pop());
```

5 4 3

Finally, it is worth noting that all of the methods are $O(1)$.

### 3.2 Queues

Our queue class will have an enqueue method that puts an item at the back of the queue and a dequeue method that removes and returns the item at the front of the queue. It will also have isEmpty and peek methods. Like with stacks, we will implement the queue as a linked list. Here we have two variables, front and back, to keep track of the two ends of the queue. The constructor sets both to null, so we start with an empty queue. Other than that, the queue class starts out quite similarly to the stack class. Here is the start of the class:

```
public class LLQueue<T>
{
    private class Node
    {
        public T data;
```
3.2. QUEUES

```java
public Node link;

public Node(T data, Node link)
{
    this.data = data;
    this.link = link;
}

private Node front, back;

public LLQueue()
{
    front = back = null;
}
```

Queue methods

The peek and isEmpty methods are practically the same as for stacks. We just use front in place of top.

Just like with stacks, adding and removing things with queues comes down to rearranging a few links and whatnot. However, unlike queues, we do now have special cases to deal with empty queues. The figure below shows how the regular cases work:

To enqueue something we create a new node for it and place it at the back of the queue by setting the current back.next to point to the new node. We then set back equal to the new node. There is an edge case we have to consider, which is if the queue is empty. In that case, setting back.next would raise a null pointer exception, so instead we just set back equal to the new node. We also have to set front to that new node because there is only one node and it is at both the back and the front of the queue.

```java
public void enqueue(T value)
{
    Node newNode = new Node(value, null);
    if (front==null)
        front = back = newNode;
    else
        back = back.link = newNode;
}```
To dequeue we return the value of front.data and set front to front.link. If this operation empties the list, then we've deleted the back of the list and so we update back to null.

```java
public T dequeue()
{
    T data = front.data;
    front = front.link;
    if (front==null)
        back = null;
    return data;
}
```

Here is the entire class:

```java
public class LLQueue<T>
{
    private class Node
    {
        public T data;
        public Node link;
        public Node(T data, Node link)
        {
            this.data = data;
            this.link = link;
        }
    }
    private Node front, back;
    public LLQueue()
    {
        front = back = null;
    }
    public boolean isEmpty()
    {
        return front==null;
    }
    public T peek()
    {
        return front.data;
    }
    public void enqueue(T value)
    {
        Node newNode = new Node(value, null);
        if (front==null)
            front = back = newNode;
        else
            back = back.link = newNode;
    }
    public T dequeue()
    {
        T data = front.data;
        front = front.link;
        if (front==null)
            back = null;
        return data;
    }
}
```
3.3. STACKS, QUEUES, AND DEQUES IN THE COLLECTIONS FRAMEWORK

Here is an example of the class in action.

```java
LLQueue<Integer> queue = new LLQueue<Integer>();
queue.enqueue(3);
queue.enqueue(4);
queue.enqueue(5);
System.out.println(queue.dequeue() + " " + queue.dequeue() + " " + queue.dequeue());
```

3 4 5

Finally, just like with stacks, it is worth noting that all of the queue methods are \( O(1) \).

Deques

A deque behaves like a list that supports adding and removing elements on either end. The name deque is short for double-ended queue. Since it supports addition and deletion at both ends, it can also be used as a stack or a queue. Exercise 8 is about implementing a deque class.

3.3 Stacks, queues, and deques in the Collections Framework

The Collections Framework supports all three of these data structures. According to Java's documentation, all three are best implemented using the `ArrayDeque` class, which is kind of a workhorse class, able to implement a variety of data structures.

Queues

For queues, there is an interface called `Queue` which we usually implement with the `ArrayDeque` class. The queue's enqueue and dequeue operations are called `add` and `remove`, and its peek operation is called `element`. Here is a simple example:

```java
import java.util.ArrayDeque;
import java.util.Queue;

public class QueueExample
{
    public static void main(String[] args)
    {
        Queue<Integer> queue = new ArrayDeque<Integer>();
        queue.add(3);
        queue.add(4);
        queue.add(5);
        queue.remove();
        queue.add(6);
        System.out.println(queue);
    }
}
```
The front of the queue here is on the left.

Stacks

The Collections Framework contains a Stack interface, but it is old and flawed and only kept around to avoid breaking old code. The Java documentation recommends against using it. Instead, it recommends using the Deque interface, which is usually implemented by the ArrayDeque class.

Using this, we have the expected push, pop methods. The peek method is called element. Here is a simple example:

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class StackExample {
    public static void main(String[] args) {
        Deque<Integer> stack = new ArrayDeque<Integer>();
        stack.push(3);
        stack.push(4);
        stack.push(5);
        stack.pop();
        stack.push(6);
        System.out.println(stack);
    }
}
```

The top of the stack here is on the left.

Deques

We use the Deque interface for deques in the Collections Framework and implement it using the ArrayDeque class. The class has the following methods:

- `addFirst` — add something to the beginning of the list.
- `addLast` — add something to the end of the list.
- `getFirst` — returns the first element of the list without removing it.
- `getLast` — returns the last element of the list without removing it.
- `removeFirst` — removes the first thing from list.
3.4. APPLICATIONS

- `removeLast` — removes the last thing from list.

Here is an example:

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeExample
{
    public static void main(String[] args)
    {
        Deque<Integer> deque = new ArrayDeque<Integer>();
        deque.addFirst(3);
        deque.addFirst(4);
        deque.addFirst(5);
        deque.addLast(10);
        deque.addLast(11);
        deque.removeLast();
        deque.removeFirst();
        System.out.println(deque);
    }
}
```

```
[4, 3, 10]
```

The front of the deque here is on the left.

**More about using the Collections Framework**

Each of the three data structures can also be implemented using the `LinkedList` class, though Java’s documentation says that `ArrayDeque` is faster.

The `ArrayDeque` class and `LinkedList` classes contain a number of other methods like `isEmpty`, `size` and `contains` and `remove` that may be useful.

Finally, all of the methods mentioned—`pop`, `remove`, `element` etc.—will raise an exception if they fail (like trying to pop from an empty stack). There are other methods provided by the `ArrayDeque` and `LinkedList` classes that behave the same as these except that instead of raising an exception, they will return `false` or `null` upon failure. For example, the non-exception-raising equivalents for queues of `add`, `remove`, and `element` are `offer`, `poll`, and `peek`. See the Java documentation for further details.

### 3.4 Applications

Stacks and queues have many applications in algorithms and the architecture of computers. Here are a few.

1. **Breadth-first search/Depth-first search** — These are both useful brute-force searching algorithms that are discussed in Section 8.3. They can be implemented with nearly the same algorithm, the
only major change being that breadth-first search uses a queue, whereas depth-first search uses a stack.

2. Queues are used in operating systems for scheduling tasks. Often, if the system is busy doing something, tasks will pile up waiting to be executed. They are usually done in the order in which they are received, and so a queue is an appropriate data structure to handle this.

3. Most programming languages use a call stack to handle function calls. When a program makes a function call, it needs to store information about where in the program to return to after the function is done. To see why a stack is used, suppose the main program $P$ calls function $A$ which in turn calls function $B$. Once $B$ is done, we need to return to $A$, which was the most recent function called before $B$. Since we always need to go back to the most recent function, a stack is appropriate. The call stack is important for recursion as we will see in Chapter 4.

4. Stacks are useful for parsing formulas. Suppose, for example, we need to write a program to evaluate user-entered expressions like $2 \ast (3+4) + 5 \ast 6$. One technique is to convert the expression into what is called postfix notation (or Reverse Polish Notation) and then use a stack. In postfix, an expression is written with its operands first followed by the operators. Ordinary notation, with the operator between its operands is called infix notation.

For example, $1 + 2$ in infix becomes $12+$ in postfix. The expression $1 + 2 + 3$ becomes $123++$, and the expression $(3 + 4) \ast (5 + 6)$ becomes $3456+\ast$. 

To evaluate a postfix expression, we use a stack of operands and results. We move through the formula left to right. Every time we meet an operand, we push it onto the stack. Every time we meet an operator, we pop the top two items from the stack, apply the operand to them, and push the result back onto the stack.

For example, let's evaluate $(3 + 4) \ast (5 + 6)$, which is $3456+\ast$ in postfix. Here is the sequence of operations.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 3 | push it onto stack. | Stack = [3]  
| 4 | push it onto stack. | Stack = [4,3]  
| + | pop 4 and 3 from stack, compute $3 + 4 = 7$, push 7 onto stack. | Stack = [7]  
| 5 | push it onto stack. | Stack = [5,7]  
| 6 | push it onto stack. | Stack = [6,5,7]  
| + | pop 6 and 5 from stack, compute $5 + 6 = 11$, push 11 onto stack. | Stack = [11,7]  
| * | pop 11 and 7 from stack, compute $7 \ast 11 = 77$, push 77 onto stack | Stack = [77]  

The result is 77, the only thing left on the stack at the end.

Postfix notation has some benefits over infix notation in that parentheses are never needed and it is straightforward to write a program to evaluate postfix expressions. In fact, many calculators of the ’70s and ’80s used postfix for just this reason. Calculator users would enter their formulas using postfix notation. To convert from infix to postfix there are several algorithms. A popular one is Dijkstra’s Shunting Yard Algorithm.

In general, queues are useful when things must be processed in the order in which they were received, and stacks are useful if things must be processed in the opposite order, where the most recent thing must be processed first.
3.5 Exercises

1. The following sequence of stack operations is performed. What does the stack look like after all the operations have been performed? Please indicate where the stack top is in your answer.

   ```
   push(33);
   pop();
   push(28);
   push(19);
   push(14);
   pop();
   push(11);
   push(32);
   ```

2. Suppose instead that the stack from the problem above is replaced with a queue and that the `push/pop` operations are replaced with `enqueue/dequeue` operations. What would the queue look like after all the operations have been performed? Please indicate where the queue front is in your answer.

3. Add exception-handling to the stack and queue classes. An exception should be raised if the user tries a `peek` operation or a `pop/dequeue` when the data structure is empty.

4. The Java documentation recommends using the `ArrayDeque` class for a stack. A problem with this is that the `ArrayDeque` class is very general and has a lot of methods that are not needed for stacks. Create a stack class that acts as a wrapper around the `ArrayDeque` class. It should provide the same methods as our stack class from this chapter and implement them by calling the appropriate `ArrayDeque` methods. There should be no other methods.

5. Do the same as the previous exercise, but for queues.

6. Implement a stack class using dynamic arrays instead of linked lists. The stack class should have the same methods as the one from this chapter.

7. Implement a queue class using dynamic arrays instead of linked lists. The queue class should have the same methods as the one from this chapter.

8. Implement a deque class from scratch using a linked list approach like in this chapter. It should have methods `addFront`, `removeFront`, `addBack`, `removeBack`, `size`, `isEmpty`, `toString`, and a constructor that creates an empty deque.

9. Implement a queue class by using two stacks. The queue class should have the same methods as the one from this chapter.

10. Write a method `reverse(List<Integer> list)` that uses a stack to reverse `list`. The stack should be implemented using Java’s `ArrayDeque` class.

11. Write a postfix expression evaluator. For simplicity, assume the valid operators are `+`, `−`, `∗`, and `/ and that operands are integers.
12. Write a simple version of the card game *War*. Instead of cards with suits, just consider cards with integer values 2 through 14, with four copies of each card in the deck. Each player has a shuffled deck of cards. On each turn of the simplified game, both players compare the top card of their deck. The player with the higher-valued card takes both cards and puts them on the bottom of his deck. If both cards have the same value, then each player puts their card on the bottom of their deck (this part differs from the usual rules). Use queues to implement the decks.

13. For this problem you will be working with strings that consist of capital and lowercase letters. Such a string is considered valid if it satisfies the following rules:

- The letters of the string always come in pairs—one capital and one lowercase. The capital letter always comes first followed by its lowercase partner somewhere later in the string.
- Letters may come between the capital and lowercase members of a pair $\xi$, but any capital letter that does so must have its lowercase partner come before the lowercase member of $\xi$.

Here are a few valid strings: ZYyz, ABCcba, ABCcCcaba, AaBb, AAaa and ABBbCDEedcba.

Here are a few invalid strings:

- $ABb$ (because $A$ has no partner)
- $BaAb$ (because $A$ should come before $a$)
- $ABab$ (breaks the second rule — $b$ should come before $a$)
- $IAEea$ (breaks the second rule — $a$ should come before $i$)
Chapter 4

Recursion

Informally, recursion is the process where a function calls itself. Recursion takes a little getting used to, and it can often provide simple and elegant solutions to problems. The way to use recursion to solve problems is to formulate the problem in terms of smaller versions of itself.

4.1 Reversing a string

Here is a simple example of a recursive function to reverse a string:

```java
public static String reverse(String s)
{
    if (s.equals(""))
        return "";
    return reverse(s.substring(1)) + s.charAt(0);
}
```

To understand how it works, consider the string `abcde`. Pull off the first letter to break the string into `a` and `bcde`. If we reverse `bcde` and add `a` to the end of it, we will have reversed the string. That is what the following line does:

```java
return reverse(s.substring(1)) + s.charAt(0);
```

The key here is that `bcde` is smaller than the original string. The function gets called on this smaller string and it is reversed by breaking off `b` from the front and reversing `cde`. Then `cde` is reversed in a similar way. We continue breaking the string down until we can’t break it down any further, as shown below:

```
reverse(abcde) = reverse(bcde) + a
reverse(bcde) = reverse(cde) + b
reverse(cde) = reverse(de) + c
reverse(de) = reverse(e) + d
```
reverse(e) = reverse("") + e  
reverse("") = ""

Building back up from the bottom, we go from "" to e to ed to edc, etc. All of this is encapsulated in the single line of code shown below again, where the reverse function keeps calling itself:

```java
return reverse(s.substring(1)) + s.charAt(0);
```

We stop the recursive process when we get to the empty string "". This is our base case. We need it to prevent the recursion from continuing forever. In the function it corresponds to the following two lines:

```java
if (s.equals(""))
    return "";
```

Another way to think of this is as a series of nested function calls, like this:

reverse("abcde") = reverse(reverse(reverse(reverse(reverse("")+"e")+"d")+"c")+"b")+"a";

In general, to create a recursive algorithm, there are two questions that are useful to ask:

1. How can I solve the problem in terms of smaller cases of itself?
2. How will I stop the recursion? (In other words, what are the small cases that don't need recursion?)

### 4.2 Length of a string and sum of a list

Here is a recursive function that returns the length of a string.

```java
public static int length(String s)
{
    if (s.equals(""))
        return 0;
    return 1 + length(s.substring(1));
}
```

Here again the idea is that we break the problem into a smaller version of itself. We do the same breakdown as for reversing the string, where we pull of the first character. The length of the string is 1 plus the length of the remainder of the string. The base case that stops the recursion is when we get down to the empty string.

Here is another recursive function that sums a list of integers. We break down the list by pulling off the first element and adding it to the sum of the remaining elements. The base case is the empty list, which has a sum of 0.

```java
public static int sum(List<Integer> a)
{
    if (a.size()==0)
        return 0;
    return a.get(0) + sum(a.subList(1, a.size()));
}
```
4.3 Power function

Here is a recursive function that raises a number to a nonnegative integer power.

```java
public static double pow(double x, int n)
{
    if (n==0)
        return 1;
    return x*pow(x, n-1);
}
```

For example, `pow(3,5)` computes $3^5 = 729$.

To see how things get broken down here, consider $3^5 = 3 \cdot 3 \cdot 3 \cdot 3$. If we pull the first 3 off, we have $3^4 = 3 \cdot 3^3$. In general, $x^n = x \cdot x^{n-1}$. This translates to $\text{pow}(x,n) = x \cdot \text{pow}(x,n-1)$.

We can keep breaking $n$ down by 1 until we get to $n = 0$, which is our base case. In that case we return 1, since anything (except maybe 0) to the 0 power is 1.

It is worth comparing this with the iterative code that computes the power:

```java
int prod = 1;
for (int i=0; i<n; i++)
    prod = prod * x;
```

The base case of the recursion corresponds to the statement that sets `prod` equal to 1. The recursive case $x \cdot \text{pow}(x, n-1)$ corresponds to the line `prod = prod * x`, where `prod` holds the product computed thus far, which is similar to how `\text{pow}(x, n-1)` holds the product from the next level down.

4.4 Printing the contents of a directory

The previous examples we've looked at could be done iteratively (that is, using loops). In fact, for reasons we'll talk about a little later, they should be done iteratively in Java, not recursively. We did them recursively just to demonstrate how recursion works on something simple. We'll now look at a more practical use of recursion to print out the contents of a directory.

We can see why recursion might be a natural solution here: the directories may have subdirectories which themselves may be subdirectories, and so on to some arbitrary level. To print the contents of those subdirectories we can use the same function. Here is the recursive solution:

```java
public static void printContents(File file)
{
    if (!file.isDirectory())
        System.out.println(file.getName());
    else
        for (File f : file.listFiles())
            printContents(f);
}
```

Basically, if the file is not a directory (i.e., it is a normal file), then we just print out its name. This acts as the base case of the recursion. Otherwise if we have a directory, then we loop over all the files in the...
directory (some of which may be directories themselves) and call `printContents` on them.

For contrast, an iterative solution is shown below. It was harder for me to write than the recursive solution and it is harder to read. The way it works is to start by building up a list of files in the directory. It then loops through the list, printing out the names of the regular files, and whenever it encounters a directory, it adds all of its files to the list.

```java
public static void printDirectory(File dir)
{
    ArrayList<File> files = new ArrayList<File>();
    for (File f : dir.listFiles())
        files.add(f);
    for (int i=0; i<files.size(); i++)
    {
        File current = files.get(i);
        if (current.isDirectory())
            for (File f : current.listFiles())
                files.add(i+1,f);
        else
            System.out.println(current.getName());
    }
}
```

### 4.5 The Sierpinski triangle

The Sierpinski triangle is a fractal object formed as follows: Start with an equilateral triangle and cut out a triangle from the middle such that it creates three equal triangles like in first part of the figure below. Then for each of those three triangle do the same thing. This gives the top middle object in the figure below. Then for each of the nine (upright) triangles created from the previous step, do the same thing. Keep repeating this process.

The same procedure that we apply at the start is repeatedly applied to all the smaller triangles. This makes it a candidate for recursion. Here is a program to draw an approximation to the Sierpinski triangle:
Here is the recursive part of the code:

```java
public void sierpinski(Graphics g, int x1, int y1, int x2, int y2,
                        int x3, int y3, int depth)
{
    if (depth>0)
    {
        sierpinski(g,x1,y1,(x1+x2)/2,(y1+y2)/2,(x1+x3)/2,(y1+y3)/2,depth-1);
        sierpinski(g,x2,y2,(x2+x3)/2,(y2+y3)/2,(x2+x1)/2,(y2+y1)/2,depth-1);
        sierpinski(g,x3,y3,(x3+x1)/2,(y3+y1)/2,(x3+x2)/2,(y3+y2)/2,depth-1);
    }
    else
    {
        drawTriangle(g,x1,y1,x2,y2,x3,y3);
    }
}
```
The key to writing the code is just to worry about one case, namely the first one.

When we cut out the middle triangle, we create three new triangles, whose coordinates are shown in the figure. We then want to perform the Sierpinski triangle process on each of them, so we call sierpinski on each triangle. When we make the call, we pass the coordinates of the three vertices of the triangle (the midpoint formula is used to get them). We also have a depth variable that keeps track of how many iterations of the process we’ve got left. Each time we call sierpinski, we decrease that depth by 1 until we get to the base case of depth 0, which is when we actually draw the triangles to the screen.

The Sierpinski triangle is one example of a fractal, an endlessly self-similar figure. Recursive fractal processes like this are used to draw realistic landscapes in video games and movies. To see some of the amazing things people do with fractals, try a web search for fractal landscapes.

### 4.6 Permutations

A permutation of a set of objects is a rearrangement of those objects. For example, the permutations of 123 are 123, 132, 213, 231, 312, and 321. We can use recursion to create a function that returns all the permutations of 123…n.

The key is that we can build up the permutations from the permutations the next level down. For example, the twenty-four permutations of 1234 come from inserting a 4 in all the possible locations of each of the permutations of 123. For example, building off of 123, we get 4123, 1423, 1243, and 1234;
4.7 Combinations

To implement this, we just need a concrete example to work from, like going from permutations of 123 to permutations of 1234. Working off of this, we know that we have to loop over the list of the permutations of 123. For each one, there are four places to add a 4. So we run another loop over those places. For example, let's say we are looking at 213. Here is how we construct the four new permutations from it:

\[
\begin{align*}
 i=0 & \quad 4123 = "" + 4 + 123 = \text{substring}(0,1) + 4 + \text{substring}(1) \\
 i=1 & \quad 1423 = 1 + 4 + 23 = \text{substring}(0,2) + 4 + \text{substring}(2) \\
 i=2 & \quad 1243 = 12 + 4 + 3 = \text{substring}(0,3) + 4 + \text{substring}(3) \\
 i=3 & \quad 1234 = 123 + 4 + "" = \text{substring}(0,4) + 4 + \text{substring}(4)
\end{align*}
\]

In Java, assuming we are storing our permutations in a list called list and s is the permutation we are working off of, we would do the following:

```java
for (int i=0; i<=s.length(); i++)
    list.add(s.substring(0,i) + n + s.substring(i));
```

This is the core of the function. Other than that, we just have to set up our lists and take care of the base case \( n = 1 \), in which the only permutation is the original set itself. Here the entire function:

```java
public static List<String> permutations(int n)
{
    List<String> list = new ArrayList<String>();
    if (n==1)
    {
        list.add("1");
        return list;
    }

    for (String s:permutations(n-1))
    for (int i=0; i<=s.length(); i++)
        list.add(s.substring(0,i) + n + s.substring(i));

    return list;
}
```

4.7 Combinations

Related to permutations are combinations. Given the set \{1, 2, 3\} (1234 in shorthand), the two-element combinations are 12, 13, 14, 23, 24, and 34, all the groups of two elements from the set. Order doesn't matter, so, for instance, 34 and 43 are considered the same combination. The three-element combinations of 1234 are 123, 124, 134, and 234.

To compute all the \( k \)-element combinations from the set \{1, 2, \ldots, n\}, we can use a recursive process. Let's say we want the two-element subsets of 1234. We first generate the two-elements combinations of 123, which are 12, 13, and 23. These are half of the two-elements combinations of 1234. For the other half, we take the one-element combinations of 123, namely 1, 2, and 3, and append a 4 to the end of them to get 14, 24, and 34.
In general, we build up the $k$-element combinations of $\{1, 2, \ldots, n\}$, we take the $k$-element combinations of $\{1, 2, \ldots, n - 1\}$ and the $k - 1$ element combinations of $\{1, 2, \ldots, n - 1\}$ with $n$ appended to the end of them.

Since we are actually going down in two directions, by decreasing $n$ and $k$, we have a base case for $n = 0$ and a base case for $k = 1$. In the $n = 0$ case, we return an empty list and in the $k = 1$ case, we return the elements $1, 2, \ldots, n$.

```java
public static List<String> combinations(int n, int k)
{
    if (n==0)
        return new ArrayList<String>();
    if (k==1)
    {
        List<String> list = new ArrayList<String>();
        for (int i=1; i<=n; i++)
            list.add(String.valueOf(i));
        return list;
    }
    List<String> list = combinations(n-1,k);
    for (String s: combinations(n-1, k-1))
        list.add(s + n);
    return list;
}
```

### 4.8 Factoring numbers

In math it is often important to factor integers into their prime factors. For example, 54 factors into $2 \times 3 \times 3 \times 3$. One way to factor is to search for a factor and once we find one, we divide the number by that factor and then factor the smaller number. For example, with 54 we first find a factor of 2, divide 54 by 2 to get 27 and then factor 27. To factor 27 we find a factor of 3, divide 27 by 3 to get 9 and then factor 9. This continues until we get down to a prime number. Since we are repeating the same process over and over, each time on smaller numbers, recursion seems like a natural choice. Here is the recursive solution:

```java
public static List<Integer> factors(int n, List<Integer> factorList)
{
    for (int i=2; i<n/2; i++)
    {
        if (n%i==0)
        {
            factorList.add(i);
            factors(n/i, factorList);
            return factorList;
        }
    }
    factorList.add(n);
    return factorList;
}
```
The code is very short. To factor a number, we loop through potential factors starting with 2. If we find one, we add it to a list of found factors, and then recursively call factors on the number divided by the factor. Also note that, once a factor is found and the recursive call returns, we stop searching for factors and jump out of the function.

It may seem like we may end up adding a factor to the list that is not a prime, but that will actually never happen, as a prime factor $p$ will always be checked first, before any nonprimes with a factor of $p$, and remember that once we find a factor, we make the recursive call and return, so no larger nonprime factors involving $p$ will be checked.

The base cases for the recursion are prime numbers. If the for loop makes it all the way through without finding a factor, then we know the number is prime and add it to the list of factors.

**A small improvement**

One problem with this code is that it requires the caller to create and feed a list to the function. But the user just wants the factors and doesn’t want to be bothered with details of our implementation. An easy fix is shown below:

```java
public static List<Integer> factors(int n) {
    return factorsHelper(n, new ArrayList<Integer>());
}

public static List<Integer> factorsHelper(int n, List<Integer> factorList) {
    for (int i=2; i<n/2; i++)
    {
        if (n%i==0)
        {
            factorList.add(i);
            factorsHelper(n/i, factorList);
            return factorList;
        }
    }
    factorList.add(n);
    return factorList;
}
```

What we do is create a helper function, which actually does all the work. The user calls the factors function and that function creates the empty list to hold the factors and calls the factorsHelper function to fill up the list. The factors function then returns the list. Our factors function acts as a wrapper around the factorsHelper function to hide the list parameter from the user.

**4.9 Fibonacci numbers**

The Fibonacci numbers are the numbers 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . . , where each number in the sequence is the sum of the two previous numbers. For example, $2 = 1 + 1$, $3 = 1 + 2$, and $5 = 2 + 3$. 
The formal definition is $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$, which says that each Fibonacci number is the sum of the two previous Fibonacci numbers. We can translate this definition directly into a recursive function, like below:

```java
public static long fib(int n)
{
    if (n==1 || n==2)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

Notice how close to the mathematical definition the code is. On the other hand, here is an example of an iterative Fibonacci function:

```java
public static long fib(int n)
{
    if (n==1 || n==2)
        return 1;
    long c=1, p=1;
    for (int i=0; i<n-2; i++)
    {
        long hold = c;
        c = p+c;
        p = hold;
    }
    return c;
}
```

It's not as easy to see what's going on, is it? So it would seem that recursion has iteration beat. But there's a problem. Try running the both versions to find $F_{50}$. You will most likely get a garbage value because $F_{50}$ causes an overflow, but don’t worry about that. Instead, notice that the iterative version returns a result almost immediately, whereas the recursive one just keeps chugging and chugging. The problem with the recursive solution is that, though it is very simple, it is also very wasteful. When it computes $F_{50}$, it gets it by adding $F_{49}$ and $F_{48}$. To get $F_{49}$ it computes $F_{48}$ and $F_{47}$. But we had already computed $F_{48}$ and here we are computing it again. This waste actually grows exponentially as the tree diagram below shows:

![Tree Diagram]

This just shows the first few levels. We see that the number of computations doubles at each step so that by the 25th level, we are doing $2^{25} = 33,554,432$ calls to the function. By contrast, the iterative code runs a loop from 0 to 47, does three quick operations per iteration and is done almost instantly. The recursive method runs in $O(2^n)$ time, while the iterative method runs in $O(n)$ time.
However, there are more efficient ways to recursively find Fibonacci numbers. One way, called memoization, involves saving the values of $F_{49}$, $F_{48}$, etc. as they are computed and using instead of recomputing things.

Here is another way to recursively find Fibonacci numbers. It is essentially a direct translation of the iterative code.

```java
public static long fib(int n) {
    if (n==1 || n==2) return 1;
    return helperFib(n, 0, 1, 1);
}

public static long helperFib(int n, int i, long c, long p) {
    if (i==n-2) return c;
    return helperFib(n, i+1, c+p, c);
}
```

We have a helper function here, just like in the last section because our recursive method has a lot of parameters that the caller shouldn't be bothered with. The `fib` function just handles the simple cases that $n$ equals 1 or 2 and then sets the `helperFib` function in motion by essentially “initializing” the variable $i$ to 0, and $c$ and $p$ to 1.

Notice how the variables in the recursive code (including the loop variable $i$) become parameters to the function. Also, notice how the for loop in the iterative code runs until $i$ equals $n-2$, which is the base case of the `helperFib` function.

Let's compare the iterative code's for loop with the recursive call to `helperFib`.

```java
    for (int i=0; i<n-2; i++) return helperFib(n, i+1, c+p, c);
    long hold = c;
    c = p+c;
    p = hold;
```

Notice that the recursive function calls itself with the $i$ parameter set to $i+1$, the $c$ parameter set to $c+p$ and the $p$ parameter set to $c$. This is the equivalent of what happens in the iterative code's for loop, where we do $i++$, $c=c+p$, and $p=hold$, where $hold$ holds the old value of $c$.

In general, it is possible to translate iterative code to recursive code in a way similar to what we did here, where variables in the iterative code become function parameters in the recursive code.

### 4.10 Towers of Hanoi

The Towers of Hanoi is a classic problem. There are three pegs, and the one on the left has several rings of varying sizes on it. The goal is to move all the rings from that peg to the peg on the right. Only one ring may be moved at a time, and a larger ring can never be placed on a smaller ring. See the figure below.
At this point, it would be good to stop reading and get out four objects of varying sizes and try the problem. Rings and pegs are not needed—any four objects of different sizes will do. The minimum number of moves needed is 15. It’s also worth trying with just three objects. In that case, the minimum number of moves is 7.

We will solve the problem using recursion. The key idea is that the solution to the problem builds off the solution with one ring less. To start, here is the optimal solution with two rings:

The solution takes three moves and is pretty obvious. Shown below is the optimal seven-move solution with three rings.

Looking carefully, we can see that the solution for two rings is used to solve the three-ring problem. Steps 1, 2, and 3 are the two-ring solution for moving two rings from the left to the center. Steps 5, 6, and 7 are the two-ring solution for moving two rings from the center to the right. And overall, the three-ring solution itself is structured like the two-ring solution. Let's compare the solutions.

The two-ring solution can be described thusly:

Move the top ring to the center, move the bottom ring to the right, move the top ring to the right.

The three-ring solution can be described in a similar way:

Move the top two rings to the center, move the bottom ring to the right, move the top two rings to the right.

And the same can be said about the four-ring solution:

Move the top three rings to the center, move the bottom ring to the right, move the top three rings to the right.
And to move the top three rings, we use the three-ring solution. Shown below are parts of the two-, three-, and four-ring solutions, lined up, showing how they are similar.

The general solution is a similar combination of the \( n - 1 \) solution and the two-ring solution. We use the \( n - 1 \) solution to move the top \( n - 1 \) rings to the center, then move the bottom ring to the right, and finally use the \( n - 1 \) solution to move the top three rings to the right.

It's not too hard to count the number of moves needed in general. The solution for two rings requires 3 moves. The solution for three rings requires two applications of the two-ring solution plus one more move, for a total of \( 2 \cdot 3 + 1 = 7 \) moves. Similarly, the solution for four rings requires two applications of the three-ring solution plus one additional move, for \( 2 \cdot 7 + 1 = 15 \) moves. In general, if \( h_n \) is the number of moves in the optimal solution with \( n \) rings, then we have \( h_n = 2h_{n-1} + 1 \). And in general, \( h_n = 2^n - 1 \) moves.

This is exponential growth. Already, the solution for 10 pegs requires \( 2^{10} - 1 = 1023 \) moves. This is a lot, but doable by hand in under an hour. The solution for 20 pegs requires over a million moves, and the solution for 30 pegs requires over a billion moves. The solution for 100 pegs requires roughly \( 10^{30} \) moves, which just might be able to be completed if every computer in existence worked round the clock for millions of years on the problem.

Here is a program that returns the optimal solution to the Towers of Hanoi problem.

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class Hanoi {
    private Deque<Integer> s1, s2, s3;

    public Hanoi(int n) {
        s1 = new ArrayDeque<Integer>();
        s2 = new ArrayDeque<Integer>();
        s3 = new ArrayDeque<Integer>();

        for (int i=n; i>=1; i--)
            s1.push(i);

        System.out.println(s1 + " " + s2 + " " + s3);
        recursiveHanoi(s1,s3,s2,n);
    }

    public void recursiveHanoi(Deque<Integer> start, Deque<Integer> end, Deque<Integer> other, int size) {
    }
```
if (size==1)
{
    end.push(start.pop());
    System.out.println(s1 + " " + s2 + " " + s3);
}
else
{
    recursiveHanoi(start, other, end, size-1);
    recursiveHanoi(start, end, other, 1);
    recursiveHanoi(other, end, start, size-1);
}

public static void main(String[] args)
{
    new Hanoi(3);
}

Here is the output for the three-ring solution:

```
[1, 2, 3] [] []
[2, 3] [] [1]
[3] [2] [1]
[3] [1, 2] []
[] [1, 2] [3]
[1] [2] [3]
[1] [] [2, 3]
[] [] [1, 2, 3]
```

The code uses three stacks to represent the pegs. The class's constructor initializes the stacks and fills the left stack. All of the work is done by the recursiveHanoi method. The method takes three stacks—start, end, and other—as parameters, as well as an integer size. Any move can be thought of as moving size number of rings from start to end, using other as a helper peg.

The base case of the recursion is size=1, which is when we are moving a single ring from one peg to another. We accomplish that by popping the “peg” from the start stack and pushing it onto the end stack. At this point we also print out that stacks, so that every time a switch is made, the stacks are printed.

The recursive part of the method moves size-1 rings from the start stack to the helper stack (other), then moves the bottom ring from the start stack to the end stack, and finally moves size-1 rings from the helper stack to the end stack.

### 4.11 Using recursion

As we've seen, recursive solutions are often simpler and more straightforward than iterative solutions.

Sometimes recursion almost seems like magic. All you have to do is specify how to use the solution to the smaller problem to solve the larger one, take it on faith that the smaller one works, and you're essentially done (once you do the base case, which is often simple).
For example, here again is recursive solution to reversing a string: break off the first character, reverse
the remaining characters and add the first character to the end. The code for this is:

```java
return reverse(s.substring(1)) + s.charAt(0);
```

We just assume that `reverse` correctly reverses the smaller substring and then add the first character to
the end. This, plus the base case to stop the recursion, are all that we need.

But there is a downside to recursion. If we try running the `reverse` function on a really long string, we
will get a stack overflow error. The reason for this is every time we make a recursive call, the program
uses a certain amount of memory for local variables and remembering where in the code to return to
after the function returns. This is saved on the call stack. The problem with recursive calls is they are
nested—each call happens before the previous one returns—which can cause the call stack to fill up, or
overflow. This is a problem in most imperative languages.

There is another class of programming languages, called functional languages, which have better support
for recursion. They use what is called tail call optimization to eliminate most of the overhead of the call
stack. A function is said to be tail-recursive if it is written so that the last thing the function does is call
itself without making any use of the result of that call other than to return it. Many of the examples from
this chapter could be rewritten to be tail-recursive. There are a lot of sources that explain more about
this.

It’s actually the call stack that makes recursive solutions seem simpler than iterative ones. The call stack
saves the value of the local variables, whereas with an iterative solution, we have to keep track of the
variables ourselves. In fact, we can convert a recursive solution into an iterative by managing a stack
ourselves. In fact, anything that can be done iteratively can also be done recursively and vice-versa.

In Java, a simple rule for when to use recursion might be to use it when it gives a simpler, easier-to-code
solution than an iterative approach and doesn’t risk a stack overflow error or other inefficiency (like with
the first Fibonacci function we did). A good example of an appropriate use of recursion is for calculating
permutations. Iterative code to generate permutations is somewhat more complicated than the recursive
code. And there is no risk of a stack overflow as we couldn’t generate permutations for very large values
of `n` recursively (or iteratively) because there are so many permutations (`n!` of them).

The take-home lesson is that in Java, most code is done iteratively, and recursion is useful on occasion.

---

### 4.12 Exercises

1. Describe the benefits and drawbacks of recursive algorithms versus iterative ones.

2. Rewrite the reverse function from Section 4.1 so that it works by by trimming off the last character
   instead of the first character.

3. Rewrite the sum function from Section 4.2 so that it uses a parameter to the function called `total`
   to keep a running total.

4. Write recursive implementations of the following. In many cases an iterative solution would be
   much easier, but this exercise is for practice thinking recursively.
(a) contains(s, c) — returns true iff the string s contains the character c
(b) isPalindrome(s) — returns true iff the string s is a palindrome
(c) indexOf(s, c) — returns the index of the first occurrence of the character c in the string s and -1 if c is not in s.
(d) count(s, c) — returns a how many times the character c occurs in the string s
(e) repeat(s, n) — returns a string consisting of n copies of the string s
(f) removeAll(s, c) — returns the string obtained by removing all occurrences of the character c from the string s
(g) range(i, j) — returns a list containing the integers from i through j-1
(h) min(list) — returns the minimum element of the integer list list
(i) isSorted(list) — returns true iff the elements of the integer list list are in increasing order
(j) equals(list, list2) — returns true iff lists list1 and list2 are equal
(k) evenList(list) — returns true iff the list list has an even number of elements. Do this without actually counting how many elements are in the list.
(l) sameElements(list1, list2) — returns true iff list1 and list2 have the same elements with the same frequencies, though possibly in different orders. For instance, the function would return true [1,1,2,3] and [2,1,3,1] but false for [1,1,2,1] and [2,2,1,2].
(m) count_start(list) — counts the number of repeat elements at the start of the list list
(n) remove_start(list) — removes any repeated elements from the start of the list list
(o) duplicate(list, n) — duplicates every item in list n times. For instance, with a=[1,2,2,3] and n=3, the method would return [1,1,1,2,2,2,2,2,2,3,3,3].
(p) gcd(m, n) — returns the greatest common divisor of m and n
(q) factorial(n) — returns n!, where n! is the product of all the integers between 1 and n. For example, 5! = 5 · 4 · 3 · 2 · 1 = 120.
(r) flatten(list) — returns a flattened version of the list list. What this means is that if list is a list of lists (and maybe the lists are nested even deeper), then the result will be just a flat list of all the individual elements. For example, the flattened version of [1, [2, 3], [4, 5], [6, [7, 8, [9, 10]]]] is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].
(s) compress(list) — compresses the list so that all duplicates are reduced to a single copy. For instance, [1, 2, 3, 4, 4, 4, 5, 5, 6] gets reduced to [1, 2, 3, 4, 5, 6].
(t) runLengthEncode — performs run-length encoding on a list. Given a list it should return a list consisting of all lists of the form [x, n], where x is an element the list and n is its frequency. For example, the encoding of [a, a, b, b, b, b, c, c, c, c, d] is [[2, a], [4, b], [3, c], [1, d]].
(u) superRepeat(list, n) — Takes a list called list and an integer n and returns a new list which consists of each element of list repeated n times, in the same order as in list. For example, if list=[6, 7, 8, 8, 9] and we call superRepeat(list, 4), the method should return the list [6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9].
4.12. EXERCISES

(v) partitions(n) — returns a list of all the partitions of the integer n. A partition of n is a breakdown of n into positive integers less than or equal to n that sum to n, where order doesn't matter. For instance, all the partitions of 5 are 5, 4 + 1, 3 + 2, 3 + 1 + 1, 2 + 2 + 1, 2 + 1 + 1 + 1, and 1 + 1 + 1 + 1 + 1. Each partition should be represented as a string, like "2+1+1".

5. Translate the following code into recursive code using the technique outlined in Section 4.9.

(a) public static ArrayList<Integer> f(int n) {
    ArrayList<Integer> a = new ArrayList<Integer>();
    for (int i=2; i<n; i++) {
        if (i*i%5==1)
            a.add(i);
    }
}

(b) public int f(int n) {
    for (int i=0; i<n; i++) {
        for (int j=i; j<n; j++) {
            for (int k=j; k<n; k++) {
                if (i*i+j*j=k*k)
                    count++;
            }
        }
    return count;
}

6. What does the following recursive method do?

   public static boolean f(String s) {
       if (s.length()==0 || s.length()==1)
           return true;
       if (s.charAt(0)!=s.charAt(s.length()-1))
           return false;
       return f(s.substring(1,s.length()-1));
   }

7. Rewrite the binary search algorithm from Section 1.1 recursively.

8. Rewrite the permutations function from Section 4.6 so that it returns lists of integer lists instead of strings. For instance, when called with n = 2, it should return [[1,2], [2,1]].

9. Rewrite the permutations function from Section 4.6 to return a list of the permutations of an arbitrary string. For instance, when called on "abc", it should return [cba, bca, bac, cab, acb, abc].

10. Rewrite the permutations function from Section 4.6 iteratively.
11. Rewrite the combinations function from Section 4.7 so that it returns lists of integer lists instead of strings.

12. Rewrite the combinations function from Section 4.7 so that it returns a list of all the $k$-element combinations of an arbitrary string.

13. Rewrite the combinations function from Section 4.7 iteratively.

14. Write a recursive function called combinationsWithReplacement($n$, $k$) that returns all the $k$-element combinations with replacement of the set $\{1, 2, \ldots, n\}$ as a list of strings, like the combinations function from Section 4.7. Combinations with replacement are like combinations, but repeats are allowed. For instance, the two-element combinations with replacement of $\{1, 2, 3, 4\}$ (shorthand 1234) are 11, 12, 13, 14, 22, 23, 24, 33, 34, and 44.

15. Rewrite the factor function from Section 4.8 iteratively.

16. Modify the program from Section 4.5 to recursively generate approximations to the Sierpinski carpet. The first few approximations are shown below.
Chapter 5

Binary Trees

Tree structures are fairly common in everyday life. For example, shown below on the left is a tree of the outcomes of flipping a coin three times. On the right is a simplified version of part of a phylogenetic tree, one that shows the family relationships of animal species.

These are just two simple examples. Trees are especially important in computer science. Of particular interest are binary trees, trees where each node has no more than two children.

First, a little terminology. Consider the tree below.

The circles are called nodes. Each node has a data value associated to it. The node at the top of the tree is called the root. Each node has up to two children, which we will refer to as its left child and right child. The nodes at the end of the tree are called leaves.
5.1 Implementing a binary tree

Our implementation will be similar to our linked list implementation. Each node of the tree will be represented as a Node object. The Node class will have four fields: a field to hold the node's data, a link to the left child, a link to the right child, and a link to the parent. The last link is not really necessary, but it's not too difficult to maintain and having that link will make certain other methods easier to write. To indicate that a node has no left child or right child or parent, we will use a null link. Here is the Node class.

```java
private class Node{
    T data;
    Node parent, left, right;
    public Node(T data, Node parent, Node left, Node right){
        this.data = data;
        this.parent = parent;
        this.left = left;
        this.right = right;
    }
}
```

The BinaryTree class has a class variable, root, to represent the root of the tree. This is analogous to our linked list implementation, where the only class variable was front, the front node of the list. The constructor creates an empty tree as below:

```java
public BinaryTree(){
    root = null;
}
```

Adding things to the tree

Next, we have a method add that adds a new node to the tree.

```
public void add(T data, String location){
    // Implementation...
}
```

We need to be able to specify where to add the node. One way is to specify the node's parent. We will take another approach, where we specify the location with a string of L's and R's, that describe the path we take to get to the added node. For instance, the node to be added in the figure above is specified by LRL. Here is the add function:

```java
public void add(T data, String location){
    // Implementation...
}
```
5.1. IMPLEMENTING A BINARY TREE

```java
if (location.equals(""))
{
    root = new Node(data, null, null, null);
    return;
}
String parentPart = location.substring(0, location.length()-1);
char childPart = location.charAt(location.length()-1);

Node n = root;
for (char c:parentPart.toCharArray())
    n = c=='L' ? n.left : n.right;
if (childPart=='L')
    n.left = new Node(data, n, null, null);
else
    n.right = new Node(data, n, null, null);
```

The first part of the add function considers the case where the location is the empty string. Remember that our string consists of L's and R's. A string that has none of those means we don’t go anywhere. In other words, we stay at the root.

Otherwise, we break the location string into two pieces: all the characters but the last, and the last character. That last character represents whether the new node will be the left or the right child of its parent. The rest of the string represents the route we take to get to the parent. That's what the loop is used for. We loop through the location string, moving left if we encounter an L and right otherwise. Note that the code uses Java's ternary operator, which essentially acts like an if/then/else compressed into a single line.

When the for loop ends, we are at the parent of the new node. The remainder of the function uses the last character of the location string to determine whether the new node is the left or right child of the parent and then it creates the new node.

One thing to consider here is what happens if the user passes the method an invalid string, like one that contains letters other than L and R. Our code does not handle that case. A good thing to do would be to raise an illegal argument exception.

Deleting from the tree

Our method will be called prune because it is like pruning a branch from a real-life tree. Not only is the node removed, but the entire subtree starting at that node is removed. See the figure below.
We use a string to specify the location to be pruned, just like we did with the add method. And just like with that method, we use loop through the location string, moving left or right down the tree as we encounter L's and R's. The difference is in what we do when we get to our location. There are two cases. First, if we are pruning the root, then we are essentially removing everything, and we can accomplish that by simply setting root to null. Otherwise, once we get to our location, to delete the node there, we just have to sever the link from its parent. We do that by looking at the last character in the location string—if it's an L, we set the parent's left link to null and otherwise set the parent's right link to null.

Here is the code for the method:

```java
public void prune(String location)
{
    if (location.equals"")
    {
        root = null;
        return;
    }

    Node n = root;
    for (char c:location.toCharArray())
    {
        if (c=='L')
            n = n.left;
        else
            n = n.right;
    }

    if (location.charAt(location.length()-1)=='L')
        n.parent.left = null;
    else
        n.parent.right = null;
}
```

**Printing the contents of the tree**

Let's now add a toString method for printing the tree. There are a number of different ways we can print the contents of the tree. What we'll do here is print out an in-order traversal of the tree. The way that works is we always visit nodes in the order left-center-right. That is, from any given node, we will always print out everything in the left subtree, then the node's own value, and finally everything in its right subtree. This is a recursive process. From a given node we first print out the in-order traversal of the left subtree, then the value of the node, then the in-order traversal of the right subtree.

```
3
   /
  5   6
 /     |
3     1   2
     /     |
   4     2   6
```

Let's look at how this works on the tree above. Start at the root. According the the in-order rule, we first look at its left subtree. So we're now at the node labeled with 5. Again, recursively following the
rule, we look at this node's left subtree. We're now at the node labeled 9. Its left subtree is empty, so following the in-order rule, we then look at the node itself, which has data value 9, and that's the first thing that gets printed. The node labeled 9 has no right subtree, so we back up to the previous node. Having finished its left subtree, we visit the node itself, and print out a 5. We then visit its right subtree and apply the same process. In the end, we get 9 5 1 4 3 6 2.

As the process described is recursive, we will use recursion here. The toString method is shown below:

```java
public String toString()
{
    return toStringHelper(root);
}

private String toStringHelper(Node n)
{
    if (n==null)
        return "";
    return toStringHelper(n.left)+ n.data + " " + toStringHelper(n.right);
}
```

Notice that the toString method needs a helper method. The recursion requires us to add a parameter to the method to keep track of where we are in the tree. However, the toString method (as per Java's definition of it) should have no parameters. So we create a separate method that has a parameter and does all the work, and we call that method from toString. The toString method just gets the recursion started by feeding toStringHelper a starting node, the tree's root.

We make the helper method a private method because it is not something users of our class need to see. It is just something our class needs to get some work done internally.

The toStringHelper method has a base case, which is a null node. In that case, there is no data value to print, so it returns the empty string. For the recursive part, we follow the prescription, left-center-right. All we do here is tell how to combine the results from the left, center, and right subtrees, and let recursion take care of the rest.

Here is an alternate toString method that prints out the path to each node as a string of L's and R's. To get the path, we add a parameter location to the helper function. In the recursion, we add an "L" or "R" to the previous location depending on whether we are moving on to the left or right subtree.

```java
public String toString()
{
    return toStringHelper(root, "");
}

private String toStringHelper(Node n, String location)
{
    // in-order traversal
    if (n==null)
        return "";
    if (n==root)
        return toStringHelper(n.left, location+"L") + " root: " + n.data + 
                  "\n" + toStringHelper(n.right, location+"R");
    return toStringHelper(n.left, location+"L") + " location + ": " + 
                  n.data + "$n$" + toStringHelper(n.right, location+"R");
}
```
For the tree shown earlier in this section, here is the output of this `toString` method:

<table>
<thead>
<tr>
<th>LL: 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>L:  5</td>
</tr>
<tr>
<td>LR: 1</td>
</tr>
<tr>
<td>LRR:4</td>
</tr>
<tr>
<td>root:3</td>
</tr>
<tr>
<td>RL: 2</td>
</tr>
<tr>
<td>R:  6</td>
</tr>
</tbody>
</table>

There are two other similar ways of traversing trees: *pre-order* and *post-order*. For pre-order, we go center-left-right, and for post-order, we go left-right-center. Two different traversals of a tree—breadth-first and depth-first—are the subjects of part of Exercise 1, as well as Section 8.3.

**The size of a tree**

The recursive approach we took to traversing the tree sets the pattern for most binary tree methods. For example, here is a method that returns the number of nodes in the tree:

```java
public int size() {
    return recursiveSize(root);
}

private int recursiveSize(Node n) {
    if (n==null) return 0;
    return 1 + recursiveSize(n.left) + recursiveSize(n.right);
}
```

We have a public/private pair. The public method `size` is the one that the user sees. It acts as a wrapper around the private method `recursiveSize` that does all the work. Again, the reason for two methods is that the recursive algorithm we use requires the method to have a parameter keeping track of where we are in the tree. To get the size of the tree, the user should just have to use `tree.size()`, not `tree.size(tree.root)`.

The way the recursive method works is to break the tree down into smaller subtrees, namely the left subtree and the right subtree from each node. The number of elements in the part of the tree including the current node and all its descendants is equal to the size of the left subtree plus the size of the right subtree plus 1 (for the current node). The base case of the recursion is a null node, which contributes nothing to the size of the tree. Thus we return 0 in that case. Here is an example of how a tree is broken down:
The key to writing a method like this is to think about how to combine the size of the left subtree and the right subtree to get the answer for the entire tree. In the example above, we see that the size of the tree is $4 + 2 + 1$, which is the size of the left subtree plus the size of the right subtree plus 1 for the node itself.

The number of leaves on the tree

Here is another example to get a feel for how recursion is useful on binary trees. The method below returns a count of how many leaves the tree has.

```java
public int leafCount() {
    return leafCountHelper(root);
}

private int leafCountHelper(Node n) {
    if (n == null) {
        return 0;
    }
    if (n.left == null && n.right == null) {
        return 1;
    }
    return leafCountHelper(n.left) + leafCountHelper(n.right);
}
```

Again we have a public/private pair. Our recursion has two base cases, one for if we've reached a null node and one for if we've found a leaf. To check to see if a node is a leaf, we just have to check to see if both of its children are null. The other base case is needed in the case that a node has only one child. In that case the node is not a leaf and when we recursively check its left and right children, one of them is a null node.

The recursive process we use here is similar to the size method in that the number of leaves is the sum of the leaves in the left subtree and the right subtree. Notice that we don't add an additional 1 here, like we did in the size method, because the current node is not a leaf itself.

The height of a tree

Let's look at a method that returns the height of a tree, which is how many levels deep the tree goes. Formally, an element is said to be at level $k$ of the tree if we have to follow $k$ links to get to it from the
CHAPTER 5. BINARY TREES

root. The root is at level 0, the root’s two children are at level 1, their children are at level 2, etc. Here is the code for the method:

```java
public int height()
{
    return heightHelper(root);
}

private int heightHelper(Node n)
{
    if (n==null)
        return 0;
    return 1 + Math.max(heightHelper(n.left), heightHelper(n.right));
}
```

The key is to imagine how to determine the height of a tree if all we know is the height of its left and right subtrees. To do this, we have to notice that the height of the tree is equal to one more than the height of either the left or the right subtree, whichever is larger. This is what the last line of the helper function does. See the figure below.

This is how to think recursively—figure out how the problem can be solved in terms of smaller versions of itself. You don’t actually have to specify how to move through the tree. Instead specify how to combine the solutions to the subproblems to get a solution to the whole, and specify some base cases to stop the recursion.

The exercises ask you to write a number of other methods for the binary tree class. Many of them follow this pattern, consisting of a public/private pair, and a recursive algorithm that combines the results from the left and right subtrees of each node.

A contains method

We’ll try one more method, a contains method that tells if the tree contains a certain value. Again, we approach this by trying to figure out how we take the answers for the left and right subtrees and combine them to get the answer for the whole tree. Here is the code:

```java
public boolean contains(T x)
{
    return containsHelper(root, x);
}

private boolean containsHelper(Node n, T x)
{
    return;
}
```
The way it works is we basically return true if the current node’s data equals the data we’re looking for or if the value is in either the node’s left or the right subtree. The last line of the helper checks the left and right subtrees. If either of them contains the value, then the entire OR expression will be true and otherwise it will be false. See the figure below.

The entire binary tree class

Here is the entire class:

```java
public class BinaryTree<T>
{
    private class Node
    {
        T data;
        Node parent, left, right;

        public Node(T data, Node parent, Node left, Node right)
        {
            this.data = data;
            this.parent = parent;
            this.left = left;
            this.right = right;
        }
    }

    Node root;

    public BinaryTree()
    {
        root = null;
    }

    public void add(T data, String location)
    {
        if (location.equals(""))
        {
            root = new Node(data, null, null, null);
            return;
        }
    }
}
```
String parentPart = location.substring(0, location.length()-1);
char childPart = location.charAt(location.length()-1);

Node n = root;
for (char c: parentPart.toCharArray())
    n = c=='L' ? n.left : n.right;
if (childPart=='L')
    n.left = new Node(data, n, null, null);
else
    n.right = new Node(data, n, null, null);

public String toString()
{
    return toStringHelper(root, "");
}

private String toStringHelper(Node n, String location)
{
    // in-order traversal
    if (n==null)
        return "";
    if (n==root)
        return toStringHelper(n.left, location+"L") + " " + "root: " + n.data + 
        "\n" + toStringHelper(n.right, location+"R");
    return toStringHelper(n.left, location+"L") + " " + location + ": " + 
        n.data + "\n" + toStringHelper(n.right, location+"R");
}
/* The following simpler toStringHelper that just prints out the elements of 
the tree using an in-order traversal without labeling where they are from. */

private String toStringHelper(Node n)
{
    // in-order traversal
    if (n==null)
        return "";
    return toStringHelper(n.left) + " " + n.data + " " + 
        toStringHelper(n.right);
}

public boolean isEmpty()
{
    return root==null;
}
public int size()
{
    return recursiveSize(root);
}

private int recursiveSize(Node n)
{
    if (n==null)
        return 0;
    return 1 + recursiveSize(n.left) + recursiveSize(n.right);
}
public void prune(String location)
if (location.equals(""))
{
    root = null;
    return;
}

Node n = root;
for (char c: location.toCharArray())
{
    if (c == 'L')
        n = n.left;
    else
        n = n.right;
}

if (location.charAt(location.length()-1)=='L')
    n.parent.left = null;
else
    n.parent.right = null;

public int leafCount()
{
    return leafCountHelper(root);
}

private int leafCountHelper(Node n)
{
    if (n == null)
        return 0;
    if (n.left == null && n.right == null)
        return 1;
    return leafCountHelper(n.left) + leafCountHelper(n.right);
}

public int height()
{
    return heightHelper(root);
}

private int heightHelper(Node n)
{
    if (n == null)
        return 0;
    return 1 + Math.max(heightHelper(n.left), heightHelper(n.right));
}

public boolean contains(T x)
{
    return containsHelper(root, x);
}

private boolean containsHelper(Node n, T x)
{
    if (n == null)
        return false;
    if (n.data.equals(x))
        return true;
    return containsHelper(n.left, x) || containsHelper(n.right, x);
}
Running times of the methods

The add method runs in $O(n)$ in the worst case. The worst case here is if the tree is a path, where each node, except the last, has exactly one child. If we try to add to the bottom of a tree of this form, we have walk through all $n$ nodes to get to the bottom of the tree. On the other hand, if the tree is more-or-less balanced, where for any given node both its left and right subtrees have roughly the same size, then adding is $O(\log n)$. This is because with each left or right turn we take to get to the desired node, we essentially are removing half of the tree from consideration. These ideas will be especially important in the next chapter.

The analysis above also applies also to prune. The rest of the methods all run in $O(n)$ time because they visit each of the nodes exactly once, doing a few $O(1)$ operations at each node.

5.2 A different approach

There is another approach we can take to implementing binary trees that is simpler to code, but less user-friendly to work with. The key observation here is to think of a tree as a recursive data structure. Each node of the tree consists of a data value and links to a left and a right subtree. And those subtrees are of course binary trees themselves. Here is how such a class might look (using integer data for simplicity):

```java
public class Tree {
    private int data;
    private Tree left, right;
    public Tree(int data, Tree left, Tree right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
```

Here is how we could declare the binary tree shown above. The declaration is spread across several lines for readability.

```java
Tree tree = new Tree(3,
    new Tree(5,
        new Tree(9, null, null),
        new Tree(1, null, null)),
    new Tree(6, null, null));
```

We could then add methods to our class if we want. Here’s a toString method:
public String toString()
{
    return toStringHelper(this);
}

public String toStringHelper(Tree tree)
{
    if (tree==null)
        return "";
    return toStringHelper(tree.left) + " " + tree.data + toStringHelper(tree.right);
}

We can see that this is very similar to the earlier toString method. In fact, the other recursive algorithms
we wrote would change very little with this new approach.

To add a node to a binary tree, we specify its data value, its parent, and whether to go left or right. Here
is the add method:

    public void add(int data, Tree parent, char leftRight)
    {
        if (leftRight=='L')
            parent.left = new Tree(data, null, null);
        else
            parent.right = new Tree(data, null, null);
    }

To add the nodes indicated in the tree above, we would use the following:

tree.add(2, tree.right, 'L');
tree.add(4, tree.left.right, 'R');

The prune method is quite similar:

    public void prune(Tree parent, char leftRight)
    {
        if (leftRight=='L')
            parent.left = null;
        else
            parent.right = null;
    }

This approach and the earlier node-based approach are actually quite similar. In the earlier approach
the recursion is hidden in the Node class, which has references to itself. That earlier approach is a little
more user-friendly, but that friendliness came at a cost of more work to code the methods.

Either of the two implementations of binary trees gives a good basis to work from if you need a binary
tree class for something. This is important as there is no binary tree or general tree class in the Collections
CHAPTER 5. BINARY TREES

Framework, so if you need an explicit binary tree class, you either have to code one up yourself or look around for one.

5.3 Applications

Binary trees and more general trees are a fundamental building block of other data structures and algorithms and form a useful way to represent data. They show up in networking algorithms, compression algorithms, and databases. In Chapter 6 we will see binary search trees and heaps, which are specialized binary trees used for maintaining data in order.

As a specific example, a computer's file system can be thought of as a tree with the directories and files being the nodes. The files are leaf nodes, and the directory nodes are nodes whose children are subdirectories and files in the directory.

Another place binary trees show up is in parsing expressions. An expression is broken down into its component parts and is represented using a tree like in the example below.

```
      3+4*5-6
        /
       /
      +   6
     /
    3    x
   /
  4    5
```

Trees also provide a natural way to structure moves from a game. For example, the potential moves in a game of tic-tac-toe can be represented with the following tree: The root node has nine children corresponding to nine possible opening moves. Each of those nodes has eight children corresponding to the eight possible places the second player can go. Each of those nodes has children representing the first player's next move. And those children have children, and so on to the end of the tree. A computer player can be implemented by simply searching through the tree looking for moves that are good according to some criterion.

Binary trees are also one of the simplest examples of graphs, covered in Chapter 8.

5.4 Exercises

1. Add the following methods to the binary tree class of Section 5.1.
   (a) preOrderPrint and postOrderPrint — which are analogous to inOrderPrint except they use preorder and postorder traversals, respectively.
   (b) get(location) returns the data value at the node specified by the string location
5.4. EXERCISES

(c) set(location, data) changes the data value at the node specified by the string location to the new value data.

(d) leaves() — returns a list containing the data in all the leaves of the tree

(e) level(location) — returns the level of the node specified by the string location, assuming the root is at level 0, its children are at level 1, their children are at level 2, etc.

(f) printLevel(k) — prints all of the elements at level k

(g) descendants(location) — returns the number of descendants of the node specified by the string location

(h) isUnbalanced() — returns true if there is some node for which either its left subtree or right subtree is more than one level deeper than the other

(i) equals(t) — returns true if the tree is equal to the tree t. Two trees are considered equal if they have the same values in the exact same tree structure.

(j) parenthesizedRep — returns a string containing a parenthesized representation of a tree. The basic structure of the representation at a node n is (data stored in n, representation of left subtree, representation of right subtree). If a node doesn’t have a left or a right subtree, the representation for the missing subtree is *. As an example, the tree below has the following representation:

   (3, (5, (9, *, *), (1, *, (4, *, *)) ), (6, (2, *, *), *))

(k) breadthFirstSearch(x) — determines if the tree contains x using a breadth-first search. Such a search starts at the root of the tree, and then examines all of the nodes at level 2, then all of the nodes at level 3, etc.

(l) depthFirstSearch(x) — determines if the tree contains the x using a depth-first search. Such a search starts at the root and always follows each branch to the end before backing up.

2. Our binary tree class works with generic types.

(a) Make the class less general. Write it so that it only works with integer data.

(b) Add to the class from part (a) a method, min, that returns the minimum element in the tree.

(c) Add to the class from part (a) a method depthSum that returns the sum of the elements in the tree, weighted by depth. The root is weighted by 1, each of its children is weighted by 2, each of their children is weighted by 3, etc.

(d) Read ahead to Section 6.4 and rewrite the original binary tree class to implement the Comparable interface. Then add in a method min that returns the minimum element in the tree.
3. Rewrite the size method using an iterative, instead of recursive, algorithm.

4. Rewrite the binary tree class to represent a general tree where each node can have an arbitrary number of children. To do this, use a list of nodes in place of the left and right variables.

5. Write a binary tree class that uses a dynamic array instead of a linked structure. Use the following technique: The root is stored at index 0. Its children are stored at indices 1 and 2. The children of the root’s left child are stored at indices 3 and 4, and its right children are stored at indices 5 and 6. In general, if the index of a node is \( i \), then its left and right children are stored at indices \( 2i + 1 \) and \( 2i + 2 \), respectively. The class should have three methods:
   - `setRoot(value)` — sets the value of the root node
   - `set(location, value)` — takes a string, location, of zeros and ones representing the location, as in Exercise 2, and sets the corresponding node to value
   - `toString()` — returns a string representing an in-order traversal of the tree

6. Add the methods from Exercise 1 to the dynamic array implementation from the previous exercise.
Chapter 6

Binary Search Trees, Heaps, and Priority Queues

This chapter is about data structures that are efficient at storing ordered data.

6.1 Binary Search Trees

Let’s suppose we have a list containing some data in sorted order. The data changes from time to time as things are removed from the list and new things are added to the list, but the list needs to remain sorted at all times.

It’s not hard to write an algorithm to insert an element in the proper place in the list—just step through the list element by element until you reach an element that is larger than the one you want to insert and insert the new element there. For example, the following method will do just that:

```java
public static void addSorted(int element, List<Integer> list)
{
    int i=0;
    while (i<list.size() && element>list.get(i))
        i++;
    list.add(i, element);
}
```

The only problem is that the worst-case and average-case running times of this algorithm are $O(n)$ because we have to step through all $n$ items of the list in the worst case and $n/2$ on average. This can add up if have a list of millions of items that we are continually adding and deleting from.

But, by storing our data in a more clever way, in something called a binary search tree (BST), we can get the running time down to $O(\log n)$. It is worth noting once more how big a difference there is between $n$ and $\log n$: if $n = 1,000,000$, then $\log n \approx 20$. There is a huge difference between a program that takes 1,000,000 milliseconds to run and a program that takes 20 milliseconds to run.
Recall from Section 1.1 that the binary search algorithm is an $O(\log n)$ algorithm for finding an element in a list. It works by breaking continually breaking the data into halves and essentially throwing away the half that doesn't contain the value we are looking for. It is similar to how we might look through a long list for a value—look at an entry somewhere in the middle. Assuming we didn't find it on the first try, if the entry is bigger than the one we want we know not to bother with the upper half of the list and to just look at the lower half of the list, and if the entry is smaller than the one we want, we can just focus on the upper half of the list. We then apply the same procedure to the appropriate half of the list and keep doing so, cutting the search space in half at each step until we find the value we want. This continuous cutting in half is characteristic of logarithmic algorithms.

A BST can be thought of as combination of a binary tree and the binary search algorithm. Formally, a BST is a binary tree with one special property:

For any given node $N$, each element in its left subtree is less than or equal to the value stored in $N$ and each element in its right subtree is greater than the value stored in $N$.

To add an element to a BST we start at the root and traverse the tree comparing the value to be added to the value stored in the current node. If the value to be added is less than or equal to the value in the current node, we move left; otherwise, we move right. We continue this until we reach the end of the tree (reach a null link).

For example, suppose we add the integers 5, 11, 19, 6, 2, 4, 2, 3, 26 to the BST in exactly that order. The figure below shows the evolution of the tree.

Try this on paper for yourself to get a good sense of how the BST property works.

### 6.2 Implementing a BST

As a BST is a binary tree, we will use a lot of the code from our binary tree class. In particular, we will use a very similar node class and the same `toString` method. In fact, by printing out our BST, we see where “in-order traversal” gets its name: the values will be printed in order. This is because an in-order traversal does things in the order left-middle-right, and the BST property guarantees that the data values
of everything to the left of a node will be less than or equal its value and everything to the right will be greater.

One change we will make for now is to just make our class work with integers. The reason for this is we have to compare values, and not all data types can be compared. A little later in the chapter we will see how to make it work for more general types using Java comparators. Here is the start of the class:

```java
public class BinarySearchTree {
    private class Node {
        int data;
        Node left, right;

        public Node(int data, Node left, Node right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }

    private Node root;

    public BinarySearchTree() {
        root = null;
    }

    public String toString() {
        return toStringHelper(root);
    }

    private String toStringHelper(Node n) {
        // in-order traversal
        if (n == null) {
            return "";
        }
        return toStringHelper(n.left) + " " + n.data + " " + toStringHelper(n.right);
    }
}
```

One change that we notice is our node class does not have a parent node like we had for ordinary binary trees. We could add one in, but doing so will actually not turn out to make our methods any easier to code.

**Adding things to a BST**

The idea behind adding things to a BST is simply to follow the BST property node by node. Start at the root and compare the value to be added with the root's value. If the value is less than or equal to the root's value, then go left and otherwise go right. Then do the same thing at the new node, moving left or right by comparing data values. We continue this until we read the end of the tree and add the new node there.
For instance, in the example below, let’s suppose we are adding a 7 into the BST.

![Binary Search Tree Diagram]

We start by comparing 7 to the root’s value 5. Since $7 > 5$, we move right. Then we compare 7 to the next node’s value 11 and since $7 \leq 11$, we move left. Then comparing 7 to 6, we see $7 > 6$ so we move right. At this point we stop and add the new node because the moving right leads to a null node.

To code this, we first need a special case if the root is null. With that out of the way, we run a loop to move through the tree. At each stage, we compare the value being added with the current node’s value. At each step there are four possibilities: move left, move right, add the new node at the left link of the current node, or add the new node at the right. The latter two possibilities are used when we reach the end of the tree. This is coded below:

```java
public void add(int data) {
    if (root == null) {
        root = new Node(data, null, null);
        return;
    }

    Node n = root;
    while (true) {
        if (data <= n.data && n.left == null) {
            n.left = new Node(data, null, null);
            return;
        }

        else if (data <= n.data) 
            n = n.left;

        else if (data > n.data && n.right == null) {
            n.right = new Node(data, null, null);
            return;
        }

        else 
            n = n.right;
    }
}
```

Here is some code to test the add method:

```java
BinarySearchTree bst = new BinarySearchTree();
List<Integer> list = new ArrayList<Integer>();
```
6.2. IMPLEMENTING A BST

```java
Collections.addall(list, 5, 11, 19, 6, 2, 4, 2, 3, 26);
for (int x: list)
    bst.add(x);
System.out.println(bst);
```

2 2 3 4 5 6 11 19 26

Notice that the data is displayed in sorted order.

A contains method

For a little more practice, here is a contains method that tests if a BST contains a value or not:

```java
public boolean contains(int data)
{
    Node n = root;
    while (n != null)
    {
        if (data==n.data)
            return true;
        else if (data<=n.data)
            n = n.left;
        else
            n = n.right;
    }
    return false;
}
```

The code is similar to the add method. We use a while loop to visit the nodes. If the data value is less than the current node's data value, then move left down the tree, if it is greater, move right, and if it's equal return true. If we reach a null node, then we've fallen off the end of the tree and we know the value is not there, so we return false.

Removing values

Removing a data value from a BST is a little tricky as we can't just delete its corresponding node from the tree as that would be cut off its descendants as well, essentially removing them their data values from the tree. We will break the problem into three separate cases. All of them require that we first locate the node to be deleted and its parent. Here is the code to do that:

```java
Node n = root, parent = null;
while (data!=n.data)
{
    parent = n;
    if (data>n.data)
        n = n.right;
    else
        n = n.left;
}
```
This very similar to the code we used for the add method. The difference is we also maintain a variable called parent that will end up being the parent of the node to be deleted.

Now that we've found the node, here are the cases for deleting it.

**Case 1: Deleting a leaf**

To delete a leaf node, we simply set the link from its parent to it to null. See the figure above. Here is the code to take care of this case:

```java
if (n.left==null && n.right==null)
{
    if (parent.left==n)
        parent.left=null;
    else
        parent.right=null;
    return;
}
```

**Case 2: Deleting a node with one child**

Deleting a node with one child is also pretty straightforward. As shown in the figure above, we simply reroute its parent's link around it to point to its child. The code is a little messy to deal with whether to move left or right links, but it is straightforward. Here it is:

```java
// case 2a: n has a left child and no right child
if (n.right==null)
{
    if (parent.left==n)
        parent.left=n.left;
    else
        parent.right=n.left;
    return;
}
```
6.2. IMPLEMENTING A BST

} // case 2b: n has a right child and no left child
if (n.left==null)
{
  if (parent.left==n)
    parent.left=n.right;
  else
    parent.right=n.right;
  return;
}

Deleting a node with two children

This is the tricky part because we can't just reroute a link like we did above without potentially messing up the BST property. But there is a clever trick that we can use here that will allow us to preserve the BST property without too much difficulty. What we do is look for the largest value in the node's left subtree and replace the deleted node's value with that value.

For example, suppose we are deleting 9 in the tree below. We would replace 9 with 8, which is the largest value in the node's left subtree. This will preserve the BST property because (1) we know that 8 is definitely less than anything in 9's right subtree, since 8 came from 9's left subtree, and (2) everything in 9's left subtree will be no larger than 8.

![Image of a BST tree with node 9 being deleted and 8 replacing it.]

To find the largest value in the left subtree, we step through the left subtree always following the link to the right until we can't go any farther right. The code to do that is below. Note that we will need to know the parent of the node we find. The code to find the node and its parent is below:

```java
Node m = n.left, parent2 = null;
while (m.right!=null)
{
  parent2 = m;
  m = m.right;
}
```

Once we find the node, we replace n's data value with m's data value. This is simple:

```java
n.data = m.data;
```

Finally, we have to remove m from the tree. We might have a problem is m itself has two children, but, fortunately, that can't happen. We found m by moving as far right as possible, so we are guaranteed
that \( m \) has no right child. Therefore, deleting \( m \) involves routing its parent's link around it to its child. However, we still have to be careful about how we do this because there is the possibility that \( m \) is \( n \)'s child. In that case, the while loop condition would never have been true and \( \text{parent2} \) would be \( \text{null} \). We take care of this like below:

```java
if (parent2==null)
    n.left = m.left;
else
    parent2.right = m.left;
```

Finally, there is one more thing to worry about, and that is if the value to be removed is stored in the root and the root has 0 or 1 children. These are edge cases that are not handled by our method. So we'll add the following special cases at the beginning of the method:

```java
if (root.data==data & root.left==null)
{
    root = root.right;
    return;
}
if (root.data==data & root.right==null)
{
    root = root.left;
    return;
}
```

The entire BST class

Here is the entire BST class:

```java
public class BinarySearchTree
{
    private class Node
    {
        int data;
        Node left, right;
        public Node(int data, Node left, Node right)
        {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }
    private Node root;
    public BinarySearchTree()
    {
        root = null;
    }
    public void add(int data)
    {
        if (root==null)
        {
            root = new Node(data, null, null);
            return;
        }
```
Node n = root;
while (true)
{
    if (data <= n.data && n.left==null)
    {
        n.left = new Node(data, null, null);
        return;
    }

    else if (data <= n.data)
    n = n.left;

    else if (data>n.data && n.right==null)
    {
        n.right = new Node(data, null, null);
        return;
    }

    else
    n = n.right;
}

public String toString()
{
    return toStringHelper(root);
}

private String toStringHelper(Node n)
{
    // in-order traversal
    if (n==null)
        return "";
    return toStringHelper(n.left) + " " + n.data + " " + toStringHelper(n.right);
}

public boolean isEmpty()
{
    return root==null;
}

public boolean contains(int data)
{
    Node n = root;
    while (n != null)
    {
        if (data==n.data)
            return true;
        else if (data<=n.data)
            n = n.left;
        else
            n = n.right;
    }
    return false;
}

public void remove(int data)
{
// special cases needed if data value to be removed is in root
// and root has 0 or 1 child
    if (root.data==data && root.left==null)
    {
        root = root.right;
    }
}
// find the node to be removed
Node n = root, parent = null;
while (data!=n.data)
{
    parent = n;
    if (data>n.data)
        n = n.right;
    else
        n = n.left;
}

// case 1: n has no children
if (n.left==null && n.right==null)
{
    if (parent.left==n)
        parent.left=null;
    else
        parent.right=null;
    return;
}

// case 2a: n has a left child and no right child
if (n.right==null)
{
    if (parent.left==n)
        parent.left=n.left;
    else
        parent.right=n.left;
    return;
}

// case 2b: n has a right child and no left child
if (n.left==null)
{
    if (parent.left==n)
        parent.left=n.right;
    else
        parent.right=n.right;
    return;
}

// case 3: n has two children (only remaining case)
// find the node with the largest value in n's left subtree
// and find its parent
Node m = n.left, parent2 = null;
while (m.right!=null)
{
    parent2 = m;
    m = m.right;
}

// set n's data value to m's value
n.data = m.data;
// delete m from the tree
6.3 More about BSTs

Take a look at the two trees below. The one on the left is perfectly balanced, whereas the one on the right is unbalanced.

Roughly speaking, a tree is balanced if all of the paths from the root to the leaves (end-nodes) are roughly the same length. If there are some long and some short paths, the tree is unbalanced. In the perfectly balanced case, like the graph above on the left, each of the paths from the root to the leaves has length $\log n$, where $n$ is the number of nodes. In a case like this, the add and contains methods work in $O(\log n)$ time as they both walk down the tree one level at a time, until they reach their goal or the bottom of the tree. The remove method will also work in $O(\log n)$ time.

The reason for the appearance of the logarithm is this: Suppose we have a perfectly balanced BST with $n$ elements in total. At the root level there is one node. At the next level there are two nodes, the root’s two children. At the next level, each child has two children for a total of four nodes. In general, as we go from level to the next, the number of nodes doubles, so we have $2^{k-1}$ nodes at level $k$, where level 1 is the root level. If the tree is completely full to level $k$, then, using the geometric series formula, we find that there are $1 + 2 + 4 + \cdots + 2^{k-1} = 2^k - 1$ total nodes in the tree. Thus see that a tree with $n$ nodes and $k$ levels must satisfy the equation $2^k - 1 = n$. Solving for $k$, we get $k = \log(n + 1)$. So the number of levels is the logarithm of the number of nodes. Another way to look at things is that each time we choose a left or right child and step down the tree, we cutting off half of the tree. This continual dividing in half is just like the binary search algorithm, which has a logarithmic running time.

As long as the tree is relatively balanced, the methods will work in more or less $O(\log n)$ time. If there is a lot of data and it’s pretty random, the tree will be fairly well balanced. On the other hand, suppose the data we’re putting into a BST already happens to be sorted. Then each item as its added to the BST will be added as the right child of its parent. The result is a degenerate tree, a long string of nodes,
arranged like in a list. In this case the BST behaves a lot like a linked list and all the operations are reduced to $O(n)$. This is a serious problem because in a lot of real life situations, data is often sorted or nearly sorted.

In a case like this, there are techniques to balance the tree. One of these techniques, AVL trees, involves rebalancing the tree after each addition of a node. This involves moving around a few nodes in the area of the added node. Another technique is red-black trees. These techniques are nice to know, but we will not cover them here. They are covered in other data structures books and on the web.

### 6.4 Comparable interface

We chose to implement the binary search tree only with integer data instead of generics. The reason for this is that we have to compare the data values, and values from any old generic type might not be comparable. But there is a way to use generics here if we use Java's `Comparable` interface.

First, here is an example of the using the `Comparable` interface to make the objects of a class comparable. Say we have a class `Record` that contains two fields `name` and `score`. Initially, we have no natural way of comparing two of these objects. But, let's say we want to be able to compare them based on the value of `score`. Here is how to do that:

```java
public class Record implements Comparable<Record> {
    private String name;
    private int score;
    public Record(String name, int score) {
        this.score = score;
        this.name = name;
    }
    public String toString() {
        return (name + ":" + score);
    }
    public int compareTo(Record record) {
        if (this.score<record.score)
            return -1;
        else if (this.score==record.score)
            return 0;
        else
            return 1;
    }
}
```

We implement the `Comparable` interface. To do that we have to define a method `compareTo` that describes how to compare two objects. The method must return either a negative value, 0, or a positive value depending on whether current object is smaller than, equal to, or larger than the object it is being compared to. Note that since we are comparing things based on the score field, which is an integer, we can actually piggyback off of Java's `Integer` class `compareTo` function and rewrite our `compareTo` as...
6.4. **COMPARABLE INTERFACE**

follows:

```java
public int compareTo(Record2 record)
{
    return ((Integer)score1).compareTo(record.score);
}
```

Here is an example of how to use the Record class:

```java
Record rec1 = new Record("Bob", 88);
Record rec2 = new Record("Sally", 92);
if (rec1.compareTo(rec2)>0)
    System.out.println("rec1 is larger than rec2.");
```

To rewrite the binary search tree class using generics, we have only a few changes to make. First, here is the declaration of the class:

```java
public class BinarySearchTreeGeneric<T extends Comparable<T>>
```

The exact syntax is important here. Then we go through and replace all occurrences of `int` data with `T` data and replace comparisons like `data<n.data` with `data.compareTo(n.data)<0`. For example, here is how the contains method changes:

```java
public boolean contains(int data)
{
    Node n = root;
    while (n!=null)
    {
        if (data==n.data)
            return true;
        else if (data<n.data)
            n = n.left;
        else
            n = n.right;
    }
    return false;
}
```

Here is the entire generic BST. For convenience, we have also added a `toString` method and an `isEmpty` method.

```java
public class GenericBST<T extends Comparable<T>>
{
    private class Node
    {
        T data;
        Node left, right;

        public Node(T data, Node left, Node right)
        {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }

    private Node root;
```
```java
public GenericBST()
{
    root = null;
}

public void add(T data)
{
    if (root == null)
    {
        root = new Node(data, null, null);
        return;
    }

    Node n = root;
    while (true)
    {
        if (data.compareTo(n.data) <= 0 && n.left == null)
        {
            n.left = new Node(data, null, null);
            return;
        }
        else if (data.compareTo(n.data) <= 0)
            n = n.left;
        else if (data.compareTo(n.data) >= 1 && n.right == null)
        {
            n.right = new Node(data, null, null);
            return;
        }
        else
            n = n.right;
    }
}

public String toString()
{
    return toStringHelper(root);
}

private String toStringHelper(Node n)
{
    // in-order traversal
    if (n == null)
        return "";
    return toStringHelper(n.left) + " " + n.data + " " + toStringHelper(n.right);
}

public boolean isEmpty()
{
    return root == null;
}

public boolean contains(T data)
{
    Node n = root;
    while (n != null)
    {
        if (data.compareTo(n.data) == 0)
            return true;
        if (data.compareTo(n.data) >= 1)
            n = n.right;
        else
            n = n.left;
```
} return false;
}

public void remove(T data) {
    // special cases needed if data value to be removed is in root 
    // and root has 0 or 1 child
    if (root.data.compareTo(data)==0 && root.left==null) {
        root = root.right;
        return;
    }

    if (root.data.compareTo(data)==0 && root.right==null) {
        root = root.left;
        return;
    }

    // find the node to be removed
    Node n = root;
    Node parent = null;
    while (data!=n.data) {
        parent = n;
        if (data.compareTo(n.data)>=1)
            n = n.right;
        else
            n = n.left;
    }

    // case 1: n has no children
    if (n.left==null && n.right==null) {
        if (parent.left==n) 
            parent.left=null;
        else
            parent.right=null;
        return;
    }

    // case 2a: n has a left child and no right child
    if (n.right==null) {
        if (parent.left==n) 
            parent.left=n.left;
        else
            parent.right=n.left;
        return;
    }

    // case 2b: n has a right child and no left child
    if (n.left==null) {
        if (parent.left==n) 
            parent.left=n.right;
        else
            parent.right=n.right;
        return;
    }

    // case 3: n has two children (only remaining case)
// find the node with the largest value in n's left subtree
// and find its parent
Node m = n.left;
Node parent2 = null;
while (m.right!=null) {
    parent2 = m;
    m = m.right;
}

// set n's data value to m's value
n.data = m.data;

// delete m from the tree
if (parent2==null)
    n.left = m.left;
else
    parent2.right = m.left;
}

Here is an example test of the class using strings:

GenericBST<String> bst = new GenericBST<Record>();
bst.add("e");
bst.add("a");
bst.add("o");
bst.add("u");
bst.add("i");
System.out.println(bst);

a e i o u

Here is an example test of the class using the Record class from earlier in the section.

GenericBST<Record> bst = new GenericBST<Record>();
List<String> names = new ArrayList<String>();
List<Integer> scores = new ArrayList<Integer>();
Collections.addAll(names, "Al", "Bob", "Carol", "Deb", "Elly");
Collections.addAll(scores, 88, 77, 68, 95, 56);
for (int i=0; i<names.size(); i++)
    bst.add(new Record(names.get(i), scores.get(i)));
System.out.println(bst);

Elly:56 Carol:68 Bob:77 Al:88 Deb:95

Java's Collections Framework does not have a dedicated BST class, though it does use BSTs to implement some other data structures (like sets and maps, which we'll see in Chapter 7).
6.5 Heaps

A heap is a relative of a BST. Like BSTs, heaps maintain their elements in a kind of sorted order. In particular, finding the minimum element in a heap is very fast, an \(O(1)\) operation, compared with \(O(\log n)\) for a BST. Heaps are also used to implement other data structures.

A heap is a binary tree with two special properties:

1. The value of a parent node is less than or equal to the values of both its children.

2. The nodes are entered in the tree in the order shown in the figure below. Start at the root and work down the tree from left to right, typewriter style, completely filling a level before moving on to the next.

Notice the difference in property 1 between BSTs and heaps. In a BST, values to the left of the parent are always less than or equal to the parent's value and values to the right are larger, whereas in a heap, left and right are not important. All that matters is that the values of the parent be less than or equal to the values of its children. Property 2 guarantees that the heap is a balanced binary tree.

The heap described above is a min-heap. We can also have a max-heap, where the “less than” in property 1 is changed to “greater than.”

6.6 Implementing a heap

Looking at the heap shown above, we notice an interesting relationship between the parent and its left child. The “index” of the left child is exactly twice the “index” of the parent. Because of this and the fact the tree has no gaps, we can implement a heap with a dynamic array instead of as a linked structure with nodes. In particular, the root will be at list index 0, its two children will be at indices 1 and 2, its left child's children will be at indices 3 and 4, etc. The figure below shows the indices of the nodes in the heap.
This is the same as the first figure in this section, but with all the indices shifted by 1. The relationship we noted between parents and children still holds, more or less. In particular, if $p$ is the index of the parent, then its children are at indices $2p + 1$ and $2p + 2$. Shown below is an example heap and its dynamic array representation:

Here is the start of the class. For simplicity, we will assume the heap’s data are integers. Later, we will rewrite it using generics.

```java
public class Heap
{
    private ArrayList<Integer> data;

    public Heap()
    {
        data = new ArrayList<Integer>();
    }
}
```

Adding to a heap

When we add an item to a heap, we have to make sure it is inserted in a spot that maintains the heap's two properties. To do this we first insert it at the end of the heap. This takes care of the second property (completely filling the heap left to right, typewriter style), but not necessarily the first property (parent's data less than or equal to its children's data). To take care of that, we compare the inserted value with its parent. If it is less than its parent, then we swap child and parent. We then compare the inserted value to the parent's parent and swap if necessary. We keep moving upward going until the first property is satisfied or until we get to the root. The new item sort of bubbles up until it gets into a suitable place in the heap.

Here is an example where we add the value 3 (shown in green) to the rest of the heap. We start by placing it at the end of the heap. Its value is out of place, however. We compare its value to its parent, which has value 8. Since $3 < 8$, a swap is necessary. We then compare 3 with its new parent, which has value 5. Since $3 < 5$, another swap is needed. We again compare 3 with its new parent and this time $3 > 1$, so we can stop.
Here is the code for the method:

```java
public void add(int value) {
    data.add(value);
    int c = data.size() - 1, p = (c - 1) / 2;
    while (data.get(c) < data.get(p)) {
        int hold = data.get(p);
        data.set(p, data.get(c));
        data.set(c, hold);
        c = p;
        p = (c - 1) / 2;
    }
}
```

The code is a direct implementation of the preceding discussion. The only thing worth mentioning is how to use the child's index in the list to find its parent's index. If the parent's index is \( p \), then we know its children are located at \( c_1 = 2p + 1 \) and \( c_2 = 2p + 2 \). Solving each of these for \( p \), we get \( p = (c_1 - 1)/2 \) and \( (c_2 - 2)/2 \). But the code has just the formula \((\text{childIndex-1})/2\) that works for both children. The reason for this is that the division is integer division. For example, if \( p = 4 \), its children are at \( c_1 = 9 \) and \( c_2 = 10 \), and using the formula from the code, we get \((9 - 1)/2 = 4\) and \((10 - 1)/2 = 4\), so one formula actually covers both cases because of the floor operation implicit in the division.

### Removing the top of the heap

The other important heap operations are `peek` and `pop`. The `peek` method returns the value of the minimum element in the heap. Since that value is always stored at the top of the heap, this is an easy thing to do:

```java
public int peek() {
    return data.get(0);
}
```

The `pop` method not only returns the minimum value in the heap, but it also removes it from the heap. This is a little tricky because we have to move things around in the heap to fill in the gap left by removing the top. The idea here is similar to what we did with adding an element, just done in reverse. What we do is take the last element, \( x \), in the heap and move it to the top. This ensures that property 2 is satisfied. But doing so will probably break property 1, so we compare \( x \) with its children and swap if necessary. We then compare \( x \) with its new children and swap again if necessary. We continue this until property 1 is satisfied. Essentially, the item sinks until it is at a suitable place in the heap. Note that when doing the comparisons, we always swap with the smaller of the two children to ensure that property 1 holds. Here is an example:
Here is the code for the method:

```java
public int pop()
{
    // remove the top entry and replace it with the last entry
    int saveFirst = data.get(0);
    data.set(0, data.get(data.size()-1));
    data.remove(data.size()-1);

    // bubble the new top down to its proper place in the tree
    int p = 0, c1=2*p+1, c2=2*p+2;
    while (c2<data.size() && (data.get(p)>data.get(c1) || data.get(p)>data.get(c2)))
    {
        if (data.get(c1)>data.get(c2))
        {
            int hold = data.get(p);
            data.set(p, data.get(c2));
            data.set(c2, hold);
            p = c2;
        }
        else
        {
            int hold = data.get(p);
            data.set(p, data.get(c1));
            data.set(c1, hold);
            p = c1;
        }
        c1 = 2*p+1;
        c2 = 2*p+2;
    }

    // special case if a swap needs to be made at end of tree where parent has
    // a left child but no right child
    if (c2==data.size() && data.get(p)>data.get(c1))
    {
        int hold = data.get(p);
        data.set(p, data.get(c1));
        data.set(c1, hold);
    }

    return saveFirst;
}
```

The method starts by saving the value of the top of the tree for the return statement at the end of the function. Then we remove the last element from the heap and use it to replace the top element.

We then step through the tree looking for where to place the new value. The loop is set to run until we either reach the end of the tree or find a suitable position. At each step of the loop, we check to see if the value is greater than either of its children’s values and swap if so. The condition on the first if statement, \(p > c_1 \land c_1 = c_2\), guarantees that we will swap with the smaller child. If the value is not greater than either child, then property 1 is satisfied and we stop looping.

Other than that, there is one special case to consider, which is if the parent has a left child at the end of the tree and no right child. Since there is no right child, the loop will have ended, so we need this
special case after the loop to see if the parent and left child should be swapped.

Making the class generic

Just like with BSTs, since we are comparing values, to use generics we use the Comparable interface. Mostly this just involves replacing things like \( p > c_2 \) with \( p \text{.compareTo}(c_2) > 0 \), similarly to how we made the BST class generic. However, we can take things a little further. If we want to make our heap a max-heap, where the parent is always greater than or equal to its children (and the root stores the maximum value instead of the minimum), the only changes we need to make are to reverse all the greater than and less than symbols. For instance, \( p > c_2 \) becomes \( p < c_2 \), or with generic code it becomes \( p \text{.compareTo}(c_2) > 0 \) becomes \( p \text{.compareTo}(c_2) < 0 \).

It seems like a waste to copy and paste all the code to create a max-heap class when all we have to change is a few symbols, and, fortunately, there is a better way. Here is the trick: the expression \( a < 0 \) is equivalent to \( a \cdot -1 > 0 \). This seemingly trivial algebraic fact will make it so that we don’t have to change any of the \( < \) and \( > \) symbols in our code. Instead, we will create a class variable, called heapType that is set to 1 or -1 depending on whether we have a min-heap or max-heap. For example, on the left below are two conditions and on the right is what they will change to:

\[
\begin{align*}
p > c_2 \quad & \quad p \text{.compareTo}(c_2) \ast \text{heapType} > 0 \\
c_1 \leq c_2 \quad & \quad c_1 \text{.compareTo}(c_2) \ast \text{heapType} \leq 0
\end{align*}
\]

Below is the code for the generic heap class. For convenience, we have also added a toString method and an isEmpty method.

```java
import java.util.ArrayList;

public class Heap<T extends Comparable<T>> {
    public static final int MAXHEAP = -1, MINHEAP = 1;
    private int heapType;
    private ArrayList<T> data;

    public Heap(int heapType) {
        data = new ArrayList<T>();
        this.heapType = heapType;
    }

    public void add(T value) {
        data.add(value);
        int c = data.size()-1, p = (c-1)/2;
        while (data.get(c).compareTo(data.get(p))*heapType<0) {
            T hold = data.get(p);
            data.set(p, data.get(c));
            data.set(c, hold);
            c = p;
            p = (c-1)/2;
        }
    }

    public T peek()
```
\begin{verbatim}
{  return data.get(0);
}

public T pop()
{
  // remove the top entry and replace it with the last entry
  T saveFirst = data.get(0);
  data.set(0, data.get(data.size()-1));
  data.remove(data.size()-1);
  // bubble the new top down to its proper place in the tree
  int p = 0, c1=2*p+1, c2=2*p+2;
  while (c2<data.size() &&
      (data.get(p).compareTo(data.get(c1))*heapType>0
       || data.get(p).compareTo(data.get(c2))*heapType>0))
  {
    if (data.get(c1).compareTo(data.get(c2))*heapType>0)
    {
      T hold = data.get(p);
      data.set(p, data.get(c2));
      data.set(c2, hold);
      p = c2;
    }
    else
    {
      T hold = data.get(p);
      data.set(p, data.get(c1));
      data.set(c1, hold);
      p = c1;
    }
    c1 = 2*p+1;
    c2 = 2*p+2;
  }
  // special case if a swap needs to be made at end of tree where parent has
  // a left child but no right child
  if (c2==data.size() && data.get(p).compareTo(data.get(c1))*heapType>0)
  {
    T hold = data.get(p);
    data.set(p, data.get(c1));
    data.set(c1, hold);
  }
  return saveFirst;
}

public boolean isEmpty()
{
  return data.isEmpty();
}

public String toString()
{
  return data.toString();
}
\end{verbatim}

Here is a simple test of the class. It will print out the strings in alphabetical order.

```java
Heap<String> heap = new Heap<String>(Heap.MINHEAP);
heap2.add("abc");```
6.7. Running time and applications of heaps

Running time of the methods

The peek method, which returns (but does not remove) the minimum value of a min-heap, is $O(1)$. This is because the minimum value is always stored at the top of the heap and it is a quick `data.get(0)` to return that value.

The add and pop methods of heaps each run in $O(\log n)$ time. In the worst case scenario, each method will require a number of swaps equal to the number of levels in the tree. The heap is complete binary tree, so the number of levels is the logarithm of the number of nodes in the tree.

Since the add method runs in $O(\log n)$ time, building a heap from a list $n$ elements will run in $O(n \log n)$ time. (There does exist an $O(n)$ algorithm for doing that, but it is beyond the scope of this book.)

Applications of heaps

As we saw in the examples at the end of the last section, repeatedly popping elements of the heap returns the values in order. So heaps can be used to sort data. This sort is called Heapsort and it is one of the faster sorts out there. We will have more to say about it in Chapter 9. Heaps are also used to implement certain graph algorithms in Chapter 8. In general, heaps are useful when you are continually adding things a data structure and need quick access to the smallest or largest value added.

Just like with BSTs, Java's Collections Framework does not have a dedicated heap class, but it does use heaps to implement other data structures. In fact, Section 6.10 shows how to use one of those data structures as a heap.
6.8 Priority Queues

A priority queue is a data structure related to a queue. The difference is that each element is given a priority and when elements are removed from the queue, they are removed in order of priority. Lots of real-life situations can be described by priority queues. For instance, in our everyday lives we have a bunch of things to do, some of which have higher priorities than others. In computers, operating systems use priority queues to determine which tasks to run when, as certain tasks are more important than others.

6.9 Implementing a priority queue

Heaps provide a straightforward and efficient way to implement a priority queue. Our priority queue class will have four methods:

1. enqueue — takes an item and its priority and puts the item onto the queue based off its priority
2. dequeue — removes and returns the highest priority item
3. peek — returns the value of the highest priority item without removing it
4. isEmpty — returns true or false depending on whether the queue is empty or not

There is one trick here. We want the heap to sort things based on their priority. To do this we will create a subclass called PQItem that has two fields: the data and its priority. The class will implement Comparable and compare PQItem objects based off their priority field. Here is the class:

```java
private class PQItem implements Comparable<PQItem> {
    public T data;
    public int priority;
    public PQItem(T data, int priority) {
        this.data = data;
        this.priority = priority;
    }
    public int compareTo(PQItem other) {
        return ((Integer)priority).compareTo(other.priority);
    }
}
```

Implementing the priority queue methods is very quick since we will use a heap to do all the work. We will have one class variable, heap, which is of type Heap<PQItem>. Each of our four methods will simply call the appropriate heap method. Here is the entire class:

```java
public class PQueue<T> {
    private class PQItem implements Comparable<PQItem> {
        public T data;
        public int priority;
        public PQItem(T data, int priority) {
            this.data = data;
            this.priority = priority;
        }
        public int compareTo(PQItem other) {
            return ((Integer)priority).compareTo(other.priority);
        }
    }
    private Heap<PQItem> heap;
    // constructor and methods
}
```
6.9. IMPLEMENTING A PRIORITY QUEUE

```java
private class PQItem implements Comparable<PQItem>
{
    public T data;
    public int priority;

    public PQItem(T data, int priority)
    {
        this.data = data;
        this.priority = priority;
    }

    public int compareTo(PQItem other)
    {
        return ((Integer)priority).compareTo(other.priority);
    }
}

private Heap<PQItem> heap;

public PQueue()
{
    heap = new Heap<PQItem>(Heap.MAXHEAP);
}

public void enqueue(T data, int priority)
{
    heap.add(new PQItem(data, priority));
}

public T dequeue()
{
    return heap.pop().data;
}

public T peek()
{
    return heap.peek().data;
}

public boolean isEmpty()
{
    return heap.isEmpty();
}
```

Here is a small example of how to use it:

```java
Q.enqueue("Low priority event", 1);
Q.enqueue("High priority event", 8);
Q.enqueue("Medium priority event", 5);
Q.enqueue("Super high priority event", 10);

System.out.println(Q.dequeue());
System.out.println(Q.dequeue());
System.out.println(Q.dequeue());
System.out.println(Q.dequeue());
```

Super high priority event
High priority event
Medium priority event
Low priority event
6.10 Priority queues and heaps in the Collections Framework

The Collections Framework has an interface called PriorityQueue and a class called PriorityQueue implementing it. It has the following methods, which are the same as for ordinary queues:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add something to the queue</td>
</tr>
<tr>
<td>remove</td>
<td>remove something from the queue</td>
</tr>
<tr>
<td>element</td>
<td>returns the next thing to come off the queue without removing it</td>
</tr>
</tbody>
</table>

Here is an example of using PriorityQueue to schedule events, like in the previous section. To do so, we will create a new class PQItem. It is essentially the same as the one from the previous section.

```java
public class PQItem<T> implements Comparable<PQItem<T>> {
    private T data;
    private int priority;

    public PQItem(T data, int priority) {
        this.data = data;
        this.priority = priority;
    }

    public T getData() {
        return data;
    }

    public int compareTo(PQItem<T> other) {
        Integer p = priority;
        return p.compareTo(other.priority);
    }
}
```

Here is the PriorityQueue tester:

```java
PriorityQueue<PQItem<String>> pq = new PriorityQueue<PQItem<String>>();

pq.add(new PQItem<String>("Low priority event", 1));
pq.add(new PQItem<String>("High priority event", 8));
pq.add(new PQItem<String>("Medium priority event", 5));
pq.add(new PQItem<String>("Super high priority event", 10));

System.out.println(pq.remove().getData());
System.out.println(pq.remove().getData());
System.out.println(pq.remove().getData());
System.out.println(pq.remove().getData());
```

PriorityQueue can be used with any class that implements Comparable. The remove method will remove things in the order based on the class’s compareTo method. As a simple example, Java’s String class implements Comparable, comparing strings alphabetically. The following code puts a few strings onto the queue and they are removed in alphabetical order. It prints out ACX.
PriorityQueue<String> Q = new PriorityQueue<String>()
Q.add("C");
Q.add("X");
Q.add("A");
System.out.println(Q.remove() + Q.remove() + Q.remove());

To get the queue to work in reverse, we can define our own comparator and pass it as an argument to the priority queue constructor. This constructor also requires us to specify the initial capacity of the queue. Here is the code. It prints out XCA.

PriorityQueue<String> P = new PriorityQueue<String>(3,
    new Comparator<String>() {
        public int compare(String x, String y) {
            return y.compareTo(x);}});
P.add("C");
P.add("X");
P.add("A");
System.out.println(P.remove() + P.remove() + P.remove());

The comparator we define is an example of an anonymous class in Java. Further, to keep the comparison code short, we piggyback off of the String class's comparator, but it reverse the order (y.compareTo(x)) to cause things to work in reverse.

Note: According to Java's documentation, the remove method of PriorityQueue runs in $O(n)$ time, not $O(\log n)$ time.

Heaps

Heaps and priority queues are very similar, and we can use Java's PriorityQueue class as a heap. Here is how to use it with the same test program as the one we wrote to test our own heap class.

PriorityQueue<Integer> heap = new PriorityQueue<Integer>();
Random random = new Random();
for (int i=0; i<10; i++)
    heap.add(random.nextInt(30));
System.out.println(heap);
for (int i=0; i<10; i++)
    System.out.print(heap.remove() + " ");

6.11 Applications of priority queues

The key use for heaps and priority queues is that they are an efficient way to efficiently store sorted data. One can easily write a priority queue implementation that runs in $O(n)$ time—just store the objects in a list and do a linear search for the item with highest priority. This may be fine in many cases, but consider a list of $n = 1,000,000$ items. Adding and popping from a heap-based priority queue is an $O(\log n)$ operation. Comparing one million to $\log(1000000) = 20$, we see quite a difference. This difference can be important in an application that adds and pops from the queue a lot.
The take-home message is that heaps and priority queues are useful whenever we are continually storing
data and need quickly retrieve the smallest or largest values stored so far. One place this is important
is in implementing some useful algorithms like Dijkstra’s shortest path algorithm, Prim’s algorithm, and
the $A^*$ search algorithm. Dijkstra’s algorithm is covered in Section 8.8. Prim’s is mentioned briefly in
Section 8.7. The $A^*$ is a smarter version of the breadth-first search algorithm (breadth-first search is
covered in Section 8.3).

Priority queues are useful in a number of other algorithms. Here’s one example: Suppose we have a list
of integers and we want a function that returns a list of the $n$ largest integers. One way to get them is
to sort the list and then return the last $n$ elements of the sorted list. But what if we are reading a lot of
data, say from a file, and there is too much data to put into a list without running out of memory? We
need a different approach. One such approach is to use a priority queue. As we read the data from the
file, we add it to the priority queue, and once the queue has $n$ elements in it, we start popping things off
the queue. At each step, the minimum value is popped off, leaving us with only the $n$ largest. Exercise 6
is about this.

Here’s one more example of priority queues. An important type of simulation is simulating a large num-
ber of particles that can collide with one another. This shows up in computer games and in simulations
of molecules, among other places. Let’s say we have $n$ particles. Usually simulations work by setting a
small time step, say .001 seconds, and at every time step we advance all the particles forward by the
appropriate amount. At each time step we then have to check for collisions. One way to go about check-
ing for collisions is to loop through the particles and for each particle, loop through the other particles,
checking to see which are about to collide with the current particle. This requires a total of $n(n - 1)/2$
checks. This is a lot of checks to be making every .001 seconds.

We can greatly reduce the number of checks using a priority queue. The way this approach works is as
follows: First assume that the particles are moving in straight line trajectories. We can calculate ahead
of time when two particles will collide. We then use a priority queue of collisions, with priority given by
the collision time. After each time step we pop off any imminent collisions from the queue and address
them. It may happen that a collision we calculated ahead of time may not happen because one of the
particles got deflected in the meantime, so we also keep track of which collisions won’t happen. After
addressing collisions, we have to recalculate new collisions since the colliding particles are now moving
in different directions, but the total number of these calculations is usually a lot less than $n(n - 1)/2$.

6.12 Exercises

1. The following elements are added to a binary search tree in the order given.

$$16, 10, 24, 4, 9, 13, 22, 30, 23, 11, 14, 4$$

(a) Draw what the tree will look like after everything has been added.

(b) The element 22 is removed. Draw what the tree looks like now.

(c) The element 10 is removed. Draw what the tree looks like now.
2. The following elements are added to a heap in the order given.

\[16, 10, 24, 4, 9, 13, 22, 30, 23, 11, 14, 4\]

(a) Draw what the heap will look like after everything has been added.

(b) The top is removed. Draw what the heap looks like now.

(c) The new top is removed. Draw what the heap looks like now.

3. This exercise involves adding some methods to the BST class. Your methods must make use of the order property of binary search trees. In other words, they should work with binary search trees, but not on an arbitrary binary tree. A method that has to look at every node in the tree, regardless of the input data, is not good enough here. Binary search trees are ordered to make searching faster than with arbitrary trees and this problem is about taking advantage of that ordering.

(a) min() — returns the minimum item in the BST

(b) max() — returns the maximum item in the BST

(c) get(k) — returns the kth smallest item in the BST

(d) numberLess(n) — returns the number of elements in the tree whose values are less than n.

Do not use the toArrayList method from the next exercise.

(e) toArrayList() — returns an ArrayList containing the elements of the tree in increasing order

4. Write a function isHeap(list) that checks if the ArrayList list represents a heap.

5. Add a method called size to the heap class that returns the number of elements in the heap.

6. Suppose we have a list of integers and we want a function that returns a list of the n largest integers. One way to get them is to sort the list and then return the last n elements of the sorted list. But what if we are reading a lot of data, say from a file, and there is too much data to put into a list without running out of memory? We need a different approach. One such approach is to use a priority queue. As we read the data from the file, we add it to the priority queue and once the queue has n elements in it, we start popping things off the queue. At each step, the minimum value is popped off, leaving us with only the n largest. Write a method whose parameters are a data file (or file name) and an integer n. Assume the data file contains integer data. The method should use the approach described here to find the n largest values in the file.

7. Do the same as the previous exercise, but find the n smallest values.
Chapter 7

Sets, Maps and Hashing

This chapter is two useful data structures—sets and maps—and a useful technique called hashing that we will use to implement those data structures.

7.1 Sets

A set is a collection of objects with no repeats. Common operations with sets $A$ and $B$ include the union, $A \cup B$, which consists of all the elements in $A$ or $B$ (or both); the intersection, $A \cap B$, which consists of all the elements in both $A$ and $B$; and the difference $A - B$, which consists of all the elements of $A$ that are not in $B$.

We will first implement a set in a simple and natural way, using a list to hold its data. It will be quick to code and understand, but it will not be very efficient. However, it’s good practice to try implementing a set like this before we get to a more sophisticated method later in the chapter. Here is the beginning of the class.

```java
public class SimpleSet<T>
{
    private ArrayList<T> data;

    public SimpleSet()
    {
        data = new ArrayList<T>();
    }
}
```

Here is the add method for adding things to the set. The important thing here is that a set can’t have repeats, so before adding an element, we to make sure it is not already in the set.

```java
public void add(T x)
{
    if (!data.contains(x))
        data.add(x);
}
```
Here is another constructor that creates a set from a list by calling our class’s add method for each element of the list:

```java
public SimpleSet(List<T> list)
{
    data = new ArrayList<T>();
    for (T x:list)
        this.add(x);
}
```

The running time of this algorithm can be \( O(n^2) \), which is not ideal. We get a factor of \( n \) because we loop through the entire list and another factor of \( n \) (on average) from the contains method.

Next, we can implement remove, toString, size, isEmpty, and contains methods very quickly just by piggybacking off of the ArrayList methods of the same name. For example, here is the toString method (the others are implemented similarly).

```java
public String toString()
{
    return data.toString();
}
```

We also have a method that returns the data of the set as a list. Here it is:

```java
public ArrayList<T> getData()
{
    return new ArrayList<T>(data);
}
```

Note that we don’t just return `data` as then any changes the caller makes to returned list will affect `data` itself. Instead we make a copy of the list and return that. There is still one problem that we won’t address here. This returns a shallow copy of the list. If the elements in `data` are mutable objects, the caller can still change them.

**Union, intersection, and difference**

With this approach to sets it is straightforward to implement union, intersection, and difference. For union, we create an empty set, add the values from `data`, then add the values from the second set. The add method will make sure there are no repeats.

```java
public SimpleSet<T> union(SimpleSet<T> set2)
{
    SimpleSet<T> returnSet = new SimpleSet<T>();
    for (T x : data)
        returnSet.add(x);
    for (T x : set2.data)
        returnSet.add(x);
    return returnSet;
}
```
The intersection method works somewhat similarly to union. What we do here is create an empty set, loop through the data of the second set and only add those values that are also in data. The difference method is very similar, the main difference being we loop through data and only add things that are not in the second set.

The entire class

Here is the entire class:

```java
import java.util.ArrayList;
import java.util.List;

public class SimpleSet<T>
{
    private List<T> data;

    public SimpleSet()
    {
        data = new ArrayList<T>();
    }

    public SimpleSet(List<T> list)
    {
        data = new ArrayList<T>();
        for (T x:list)
            if (!data.contains(x))
                data.add(x);
    }

    public String toString()
    {
        return data.toString();
    }

    public int size()
    {
        return data.size();
    }

    public boolean isEmpty()
    {
        return data.isEmpty();
    }

    public boolean contains(T x)
    {
        return data.contains(x);
    }

    public List<T> getData()
    {
        ArrayList<T> returnList = new ArrayList<T>();
        for (T x:data)
            returnList.add(x);
        return returnList;
    }

    public void add(T x)
    {
        if (!data.contains(x))
            data.add(x);
    }
```


```java
public void remove(T x)
{
    data.remove(x);
}

public SimpleSet<T> union(SimpleSet<T> set2)
{
    SimpleSet<T> returnSet = new SimpleSet<T>();
    for (T x : data)
        returnSet.add(x);
    for (T x : set2.data)
        returnSet.add(x);
    return returnSet;
}

public SimpleSet<T> intersection(SimpleSet<T> set2)
{
    SimpleSet<T> returnSet = new SimpleSet<T>();
    for (T x : data)
        if (set2.contains(x))
            returnSet.add(x);
    return returnSet;
}

public SimpleSet<T> difference(SimpleSet<T> set2)
{
    SimpleSet<T> returnSet = new SimpleSet<T>();
    for (T x : data)
        if (!set2.contains(x))
            returnSet.add(x);
    return returnSet;
}
```

### Analysis of running time

The methods here are pretty slow. They are fine for small data sets, but they don’t scale well. The add method is $O(n)$ because it uses the ArrayList contains method, which is $O(n)$. That method essentially has to loop through the list to see if a value is in the list. The constructor that builds a set from a list runs in more or less $O(n^2)$ time, as it loops through the list and calls the $O(n)$ add method at each step. The union method is $O(n^2 + m^2)$, where $n$ and $m$ are the sizes of the two sets, as the method consists of a loop with a call to the add method inside. The intersection is even worse, being cubic in the worst case (specifically $O(n^2m)$). It consists essentially of three nested loops, namely a loop over all $n$ elements of the set, with a call to contains nested inside (which loops over the $m$-elements of set2) and a call to add inside the if statement (which has to loop again over the $n$ elements of the set to check for repeats).

The next section is about hashing, a technique that will give us $O(1)$ add and remove methods and bring the other operations down to $O(n)$. The main benefit of the approach here is that it is simple to
implement and does not contain a lot that can go wrong.

7.2  Hashing

Consider first the following approach: Suppose we know our set will only consist of integers from 0 to 999. We could create a list, called data, consisting of 1000 booleans. If the ith entry of data is true, then i is in the set and otherwise it’s not. Inserting and deleting elements comes down to setting a single element of data to true or false, an \(O(1)\) operation. Checking if the set contains a certain element comes down to checking if the corresponding entry in data is true, also an \(O(1)\) operation. This will also bring down the union, intersection, and difference operations down to linear time.

This type of set is called a bit set and it is the subject of Exercise 1. This approach has a problem, though. If our set could consist of any 32-bit integer, then we would need an ArrayList of about four billion booleans, which is not practical. Suppose, however, that we do the following: when adding an integer to the set, mod it by 1000, use that result as the integer's index (called a hash), and store that integer at the index. For instance, to add 123456 to the set, we compute 123456 mod 1000 to get 456, and then call data.set(456, 123456).

The problem with this is we could get a collision, where different integers end up with the same hash. For example, 123456, 1456, and 10456, all yield the same value when modded by 1000. There are a couple of different things we can do here.

The first thing we can do is to find a better operation than modding by 1000. The operation where we mod the integer by 1000 is called a hash function. A hash function converts a data value (called a key) into another data value, called a hash, which often serves as an index into an array or list. A good hash function to use with integers is one of the form \(h(x) = ax \mod b\), where a and b are relatively large primes. A simple, but not particularly good, hash function operating on strings of letters would be a function which converts the individual letters into numerical values, adds the numerical values of the letters and then mods by 100. For example, the hash of “message” would be \(13 + 5 + 19 + 19 + 1 + 7 + 5 \mod 100 = 69\).

There is a lot that goes into constructing good hash functions. Most hash functions, even very good ones, suffer from collisions, where multiple keys get mapped to the same hash value. A lot of thought goes into designing hash functions so that collisions are rare. The web and some other data structures books have more information on designing hash functions. But we will not focus on constructing hash functions here.

Instead, we will focus on what to do when collisions occur. There are several approaches. We will focus here on the approach called chaining. Here is how chaining works: Since several items could hash to the same index, instead of storing a single item at that index, we will store a list of items there. These lists are called buckets. For example, let’s say we are using the simple hash function \(h(x) = x \mod 5\) and we add 2, 9, 14, 12, 6, and 29 to the set. The hash table will look like this:

\[
\begin{align*}
0: & \ [2, 12] \\
1: & \ [6]
\end{align*}
\]
To add a new element to the set, we compute its hash and add it to the list at that hash (index). To determine if the set contains an element, we compute its hash and search the corresponding list for the element.

### 7.3 Implementing a set with hashing

In this section we will implement a very basic integer set using hashing.

First, we need a hash function. Here is a simple one:

```java
public int hashfunc(int key)
{
    return 7927*key % 17393;
}
```

As mentioned earlier, a good hash function to use with integers is one of the form $h(x) = ax \mod b$, where $a$ and $b$ are relatively large primes. In this case, we picked $a = 7927$ (the 1000th prime) and $b = 17393$ (the 2000th prime). Since we are modding by 17393, we will have that many buckets in our hash table.

In our implementation, the number of buckets is fixed and cannot be changed. A better approach, such as the one used by the `HashSet` class in the Collections Framework, would be to resize the hash table if things get too full. This would involve creating a new list of buckets and populating that list from the old one.

The class has two fields, an integer `numElements` that keeps track of the size of the set, and an list `data` whose elements are themselves lists. It is declared like this:

```java
private List<List<Integer>> data;
```

Below is the constructor that creates an empty set. It creates the list of buckets called `data`. Each element of data is a list itself and needs to be declared before we can add stuff to it.

```java
public HSet()
{
    data = new ArrayList<List<Integer>>(17393);
    for (int i=0; i<17393; i++)
        data.add(new ArrayList<Integer>(i));
}
```

We will implement most of the set methods in a very similar way to our simple set implementation from earlier, just with an added hashing step. To add an item to the set, we compute its hash and insert it into the bucket at the hash’s index in `data`. We don’t want any repeats, so before adding it to the bucket, we make sure it is not already there.
public void add(int x)
{
    int hash = hashfunc(x);
    if (!data.get(hash).contains(x))
    {
        data.get(hash).add(x);
        numElements++;
    }
}

Removing items from the set is quick:

public void remove(int x)
{
    data.get(hashfunc(x)).remove(x);
    numElements--;
}

The contains method is also quick:

public boolean contains(int x)
{
    return data.get(hashfunc(x)).contains(x);
}

Because we have a class variable keeping track of the number of elements, the isEmpty and size methods are simple:

public boolean isEmpty()
{
    return numElements==0;
}

public int size()
{
    return numElements;
}

Most of the rest of the methods we implement require a way to get a list of the set's elements. Here is a method to do that. It loops through each of the lists that comprise data, adding each of their elements to a master list that is returned.

public List<Integer> getData()
{
    List<Integer> returnList = new ArrayList<Integer>();
    for (List<Integer> list:data)
        for (int x:list)
            returnList.add(x);
    return returnList;
}

Armed with this method, the toString method is simple:

public String toString()
{
    return this.getData().toString();
}
The union, intersection, and difference methods work much like the method from our earlier implementation of sets. Here is the union method. It makes use of the getData method.

```java
public HSet union(HSet set2)
{
    HSet returnSet = new HSet();
    for (int x:this.getData())
        returnSet.add(x);
    for (int x:set2.getData())
        returnSet.add(x);
    return returnSet;
}
```

Here is the entire class:

```java
import java.util.ArrayList;
import java.util.List;
public class HSet
{
    private List<List<Integer>> data;
    private int numElements;
    public int hashfunc(int key)
    {
        return 7927*key % 17393;  // 1000th and 2000th primes
    }
    public HSet()
    {
        data = new ArrayList<List<Integer>>(17393);
        for (int i=0; i<17393; i++)
            data.add(new ArrayList<Integer>());
        numElements = 0;
    }
    public HSet(List<Integer> list)
    {
        data = new ArrayList<List<Integer>>(17393);
        for (int i=0; i<17393; i++)
            data.add(new ArrayList<Integer>());
        numElements = 0;
        for (int x:list)
            this.add(x);
        numElements++;
    }
    public void add(int x)
    {
        int hash = hashfunc(x);
        if (!data.get(hash).contains(x))
        {
            data.get(hash).add(x);
            numElements++;
        }
    }
}```
```java
public void remove(int x) {
    data.get(hashfunc(x)).remove(x);
    numElements--;
}

global boolean contains(int x) {
    return data.get(hashfunc(x)).contains(x);
}

global boolean isEmpty() {
    return numElements==0;
}

global int size() {
    return numElements;
}

global List<Integer> getData() {
    List<Integer> returnList = new ArrayList<Integer>();
    for (List<Integer> list: data) {
        for (int x: list) {
            returnList.add(x);
        }
    }
    return returnList;
}

global String toString() {
    return this.getData().toString();
}

global HSet union(HSet set2) {
    HSet returnSet = new HSet();
    for (int x: this.getData()) {
        returnSet.add(x);
    }
    for (int x: set2.getData()) {
        returnSet.add(x);
    }
    return returnSet;
}

global HSet intersection(HSet set2) {
    HSet returnSet = new HSet();
    for (int x: set2.getData()) {
        if (this.contains(x)) {
            returnSet.add(x);
        }
    }
    return returnSet;
}

global HSet difference(HSet set2) {
    HSet returnSet = new HSet();
    for (int x: this.getData()) {
        if (!set2.contains(x)) {
            returnSet.add(x);
        }
    }
    return returnSet;
}
```
More about hashing

The approach we took here to deal with collisions is called chaining. If the hash function is a good one, then things will be well spread out through the list and none of the buckets will be very full. In this case, the add and remove methods will run in $O(1)$ time. With a poorly chosen hash function, or if we have a lot of data values and too few buckets, things could degenerate to $O(n)$ time, as we would waste a lot of time when inserting and deleting things from the buckets.

An important consideration in using a hash table is the **load factor**, which is computed by $N/M$, where $M$ is the number of buckets and $N$ is the number of elements in the set. As mentioned, if $N$ is large and $M$ is small (a high load factor), then things could degenerate to $O(n)$. If things are reversed—if there are a lot of buckets but the set is small—then there will be few collisions, but lots of wasted space, and iterating through the set will be slowed. Generally, a load factor between about .6 and .8 is considered good.

There is a common alternative to chaining called **probing** or **open addressing**. Instead of using lists of elements with the same hash, (linear) probing does the following: Calculate the item’s hash. If there is already an element at that hash index, then check the next index in the list. If it is empty, then store the item there, and otherwise keep going until a suitable index is found. There are pros and cons to probing that we won’t get into here. More information can be found in other books on data structures or on the web.

In this section we used hashing to implement a set class. Hashing is used for a whole lot more than just this. For example, we will use it to implement a map class later in this chapter. Hashing is also important for secure communications. One particularly important example is password hashing. On a secure system, user passwords are never stored in plain text. Instead, the user’s password is run through a hash function and only the hash itself is stored on the system. When the user logs onto the system, the hash value of the password is compared to the hash value stored in the system. The hash function used is a one-way (non-invertible) function, so that it is not feasible to go backwards from a hash to a password.

As a concrete example, consider the hashing function that converts each character in the user’s password to a numerical value, adds up those values, and mods by 100. For example, say A=1, B=2, etc. The hash of the password `PASSWORD` is $(16 + 1 + 19 + 19 + 23 + 15 + 18 + 4) \mod 100 = 17$. Only the value 17 is stored on the system. When the user logs on and types in their password, its hash is computed and compared to 17.

Notice that 17 can be the hash value of lots of different passwords, like `CCCCCB` or `EEAAA`. There are in fact infinitely many passwords that hash to 17, so you can’t really be sure what the user’s password is if all you know is the hash value. On the other hand, any password with a hash value of 17 can be used to log on to the system. Now this hash function is poor one since it has a lot of collisions, but more secure
7.4 Sets in the Collections Framework

The Collections Framework contains an interface called Set. There are three classes that implement the interface: TreeSet, HashSet, and LinkedHashSet. HashSet is a hash table implementation of a set using chaining. The TreeSet class is an implementation of a set using a BST. It is useful as a set whose elements are stored in sorted order. The LinkedHashSet class is similar to HashSet except that it preserves the order in which items were added to the set. The HashSet class does not do this (neither does our HSet class). The hash function tends to scramble elements. In terms of performance, HashSet and LinkedHashSet are both fast, with $O(1)$ running time for its add, remove, and contains methods. TreeSet is a bit slower, with $O(\log n)$ running time for those methods.

Here are three sample declarations:

```java
Set<Integer> set = new TreeSet<Integer>();
Set<Integer> set = new HashSet<Integer>();
Set<Integer> set = new LinkedHashSet<Integer>();
```

Here are some useful methods of the Set interface:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)</td>
<td>adds x to the set</td>
</tr>
<tr>
<td>contains(x)</td>
<td>returns true iff x is in the set</td>
</tr>
<tr>
<td>difference(t)</td>
<td>returns a set of the elements of s not in the set t</td>
</tr>
<tr>
<td>intersection(t)</td>
<td>returns a set of the elements in both s and t</td>
</tr>
<tr>
<td>remove(x)</td>
<td>removes x from the set</td>
</tr>
<tr>
<td>union(t)</td>
<td>returns a set of all the elements that are in s or t</td>
</tr>
</tbody>
</table>

We can iterate through the items of a set with a foreach loop. Here is a loop through a set of integers:

```java
for (int x: set)
    System.out.println(x);
```

We can use the Set and List constructors to convert between sets and lists. For example,

```java
List<Integer> list = new ArrayList<Integer>(set);
Set<Integer> set = new HashSet<Integer>(list);
```

7.5 Applications of sets

Sets give an easy way to remove the duplicates from a list. Just convert the list to a set and then back to a list again:
Sets are particularly handy if you have a collection of objects and you need to do repeated checks to see if things are in the collection. A simple example is a spell checker. One way to implement it is to use a list of dictionary words and read through the document to be spell-checked word by word, checking each word to see if it is in the dictionary wordlist. Once we break the document into its individual words, we could use the following code to find misspellings:

```java
List<String> words = new ArrayList<String>();
// ... code that fills up list would go here
for (String word:documentWords)
    if (!wordlist.contains(word))
        System.out.println(word);
```

This uses a linear search. And it is slow. On my computer, with a wordlist of 300,000 words and 10-million words to check, the spell-check was on pace to finish in about 12 hours before I stopped it.

An easy improvement is to make sure the wordlist is arranged alphabetically and use a binary search, which runs in logarithmic time.

```java
List<String> words = new ArrayList<String>();
// ... code that fills up list would go here
for (String word:documentWords)
    if (Collections.binarySearch(wordlist, word)<=0)
        System.out.println(word);
```

This is a massive improvement. Tested on the same document as the previous algorithm, this one ran in 4 seconds, once again demonstrating how big a difference there is between linear and logarithmic growth.

But using a HashSet, we can still do better.

```java
List<String> words = new ArrayList<String>();
// ... code that fills up list would go here
for (String word:documentWords)
    if (Collections.binarySearch(wordlist, word)<=0)
        System.out.println(word);
```

Tested on the same document, the HashSet implementation took 0.6 seconds, about seven times faster than the binary search algorithm. And when I tested Java's LinkedHashSet, it ran a slight bit faster even than this.

Sets are also useful if for storing some elements where we don’t want repeats. For example, if we are running a search and as we find certain things, we add them to a list of things to be checked later. We don’t want repeats in the list as that will waste space and time, so in place of the list, we could use a set. There are a few examples of this in Chapter 8.
7.6 Java’s hashCode method

Every Java class has a hashCode method that is used by, among other things, HashSet. Java provides hashCode methods for common classes like String, Integer, etc., and it combines these to create the hashCode method for custom classes.

It's possible to override hashCode if you have something better in mind than Java's default. In fact, there is one time when it's important to override hashCode, and that if you are overriding the equals method. Here is an example. Suppose we have the following class:

```java
public class Pair {
    public int x, y;
    public Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

This is a very useful class representing an ordered pair \((x, y)\). One use for it is for returning multiple values from a function. Another is if you need a list that stores two pieces of information. We use this very class in Section 8.5 in a solution of a classic puzzle problem.

Let's say we try the following:

```java
Pair a = new Pair(1,2);
Pair b = new Pair(1,2);
if (a.equals(b))
    System.out.println("Equal!");
else
    System.out.println("Not equal");
```

We would certainly like the result to be Equal!, but it's not. By default, Java's equals method and the == operator compare purely based on whether a and b occupy the same memory location. We would like them to compare based on the fields of those objects. There's not anything we can do about getting the == operator to work with our class, but we can override equals to get that method to work properly. Here is a simple way to do it:

```java
public boolean equals(Object obj) {
    Pair other = (Pair) obj;
    return other.x==x && other.y==y;
}
```

```java
public int hashCode() {
    Integer x = this.x;
    Integer y = this.y;
```
return x.hashCode()-y.hashCode();
}

The reason hashCode must be implemented as well as equals is that Java needs hashCode and equals to work in sync internally. Note that the simple example above is not very robust. A better solution is to use an IDE to generate the code. Here is the code generated by Eclipse:

```java
@Override
public int hashCode()
{
    final int prime = 31;
    int result = 1;
    result = prime * result + x;
    result = prime * result + y;
    return result;
}

@Override
public boolean equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    testHashCode other = (testHashCode) obj;
    if (x != other.x)
        return false;
    if (y != other.y)
        return false;
    return true;
}
```

One further note: It is possible to have HashSets of custom objects, but care must be taken if the object is mutable, that is one whose fields can be changed. If we add such an object to a HashSet and then change one of its fields, the value that hashCode returns will likely change, and when we go looking for the object in the HashSet, we won’t be able to find it because its hash has changed.

### 7.7 Maps

A map is a data structure that is a generalization of a list, where indices may be objects other than integers, such as strings. Maps are known by several other names, including dictionaries, associative arrays, hashes, and symbol tables.

Here is an example. Let’s say we want to write a simple program where the user types in the name of a month and we output the number of days in the month. One way to approach this is with a bunch of if/else statements or a switch statement. That would work fine since there are only 12 months, but we could come up with a similar type of problem with maybe 300 cases instead of 12, where the if/else/switch approach would be too tedious. Another way to do it is with lists is shown below.

```java
List<String> months = new ArrayList<String>();
```
List<Integer> days = new ArrayList<Integer>();
Collections.addAll(months, "jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov", "dec");
Collections.addAll(days, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
int answer = days.get(months.indexOf("feb")); // number of days in February

This is not too bad of a solution, but using maps, we can do better. Keeping the two lists in sync in problems like this can sometimes be a problem. Maps allow us to synchronize the two lists together into one structure. Here is how to do the above using maps in the Collections Framework:

Map<String, Integer> monthDays = new HashMap<String,Integer>();
monthDays.put("jan", 31);
monthDays.put("feb", 28);
//...
monthDays.put("dec", 31);
int answer = monthDays.get("feb"); // number of days in February

The months are called keys and the days are called values. Together, a month and its number of days form a key-value pair (or binding). The keys are like generalized indices of a list.

### 7.8 Implementing a map class

To better understand how maps work, we will provide two implementations of them. The first implementation will use two lists, one of keys and one of values. This is just like the approach to the month-days problem described above, except we will manage all the synchronization and indexing details in the class itself, so the user does not have to.

Here are the methods that our class will have.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(key, value)</td>
<td>add a key-value pair to the map</td>
</tr>
<tr>
<td>remove(key)</td>
<td>remove a key (and its associated value) from the map</td>
</tr>
<tr>
<td>set(key, value)</td>
<td>change the value associated with a key</td>
</tr>
<tr>
<td>get(key)</td>
<td>get the value associated with a key</td>
</tr>
<tr>
<td>size()</td>
<td>get the number of keys in the map</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>returns true iff the map is empty</td>
</tr>
<tr>
<td>containsKey()</td>
<td>returns true iff the map contains the given key</td>
</tr>
<tr>
<td>containsValue()</td>
<td>returns true iff the map contains the given key</td>
</tr>
<tr>
<td>getKeys()</td>
<td>returns a list of the keys in the map</td>
</tr>
<tr>
<td>getValues()</td>
<td>returns a list of the values in the map</td>
</tr>
</tbody>
</table>

Our first implementation using two lists will not be particularly efficient, but it is quick to code and gives us a good understanding of how maps work.
To start, our class will be generic. The user can specify any data type for the keys and any data type for the values. The class has two list fields, keys and values.

```java
private List<T> keys;
private List<U> values;

public LMap()
{
    keys = new ArrayList<T>();
    values = new ArrayList<U>();
}
```

The `toString` method is straightforward. We loop through the keys and values lists separately and add their contents to a string to be returned. There is a special case for an empty map.

```java
public String toString()
{
    if (keys.size()==0)
        return "[]";

    String s = "[");
    for (int i=0; i<keys.size(); i++)
        s += keys.get(i) + ":" + values.get(i) + ", ";
    return s.substring(0,s.length()-2) + "]";
}
```

The methods `containsKey`, `containsValue`, `size`, and `isEmpty` are all easy to implement:

```java
public Boolean containsKey(T key)
{
    return keys.contains(key);
}

public Boolean containsValue(U value)
{
    return values.contains(value);
}

public int size()
{
    return keys.size();
}

public Boolean isEmpty()
{
    return keys.isEmpty();
}
```

The `getKeys` and `getValues` methods are implemented return copies of the keys and values lists.

```java
public List<T> getKeys()
{
    return new ArrayList<T>(keys);
}

public List<U> getValues()
{
    return new ArrayList<U>(values);
}
```
To add a key-value pair to the map, we check to make sure the key is not already in the map and then update the keys and values lists:

```java
public void add(T key, U value) {
    if (keys.contains(key))
        return;
    keys.add(key);
    values.add(value);
}
```

To remove a key, we find its index in the keys array. That index will be the same as the index of the key's corresponding value in the values array, so we can use that index to remove both the key and its value.

```java
public void remove(T key) {
    int index = keys.indexOf(key);
    keys.remove(index);
    values.remove(index);
}
```

Finally, we have get and set methods for returning and setting the value stored at a specific key.

```java
public U get(T key) {
    int index = keys.indexOf(key);
    return values.get(index);
}
```

```java
public void set(T key, U value) {
    int index = keys.indexOf(key);
    values.set(index, value);
}
```

Here is the entire class:

```java
import java.util.ArrayList;
import java.util.List;

public class LMap<T, U> {
    private List<T> keys;
    private List<U> values;

    public LMap() {
        keys = new ArrayList<T>();
        values = new ArrayList<U>();
    }

    public String toString() {
        if (keys.size() == 0)
            return "[]";
        String s = "[");
        for (int i = 0; i < keys.size(); i++) {
            s += keys.get(i) + " ";
        }
        return s + "]";
    }
}
```
for (int i=0; i<keys.size(); i++)
    s += keys.get(i) + "":"" + values.get(i) + ", ";

return s.substring(0, s.length()-2) + "]";
}

public Boolean containsKey(T key)
{
    return keys.contains(key);
}

public Boolean containsValue(U value)
{
    return values.contains(value);
}

public int size()
{
    return keys.size();
}

public Boolean isEmpty()
{
    return keys.isEmpty();
}

public List<T> getKeys()
{
    return new ArrayList<T>(keys);
}

public List<U> getValues()
{
    return new ArrayList<U>(values);
}

public void add(T key, U value)
{
    if (keys.contains(key))
        return;
    keys.add(key);
    values.add(value);
}

public void remove(T key)
{
    int index = keys.indexOf(key);
    keys.remove(index);
    values.remove(index);
}

public U get(T key)
{
    int index = keys.indexOf(key);
    return values.get(index);
}

public void set(T key, U value)
{
    int index = keys.indexOf(key);
    values.set(index, value);
}
}
Here is an example of how to use the class:

```java
LMap<String, Integer> map = new LMap<String, Integer>();
map.add("jan", 31);
map.add("feb", 28);
System.out.println(map.get("feb"));
```

```
[January:31, February:28]
```

Running time of the methods

The `containsKey` and `containsValue` methods both run in \(O(n)\) time because they have to search their corresponding lists using a linear search. The binary search can’t be used because the keys and values may not be orderable. If they are, we could a BST in place of a list to get things down to \(O(\log n)\). Adding a key-value pair to the map is also an \(O(n)\) operation because it calls `containsKey` and adding to an `ArrayList` is amortized \(O(1)\). The `remove` method is also \(O(n)\). Even the `get` and `set` methods are \(O(n)\) because they call `indexOf`, which runs in \(O(n)\) time.

We can get all of these down to \(O(1)\) using hashing.

### 7.9 Using hashing to implement a map

Using hashing to implement a map takes ideas from the two-list approach and from the hashing implementation of sets. We will use generics here for both the keys and the values.

To insert a key-value pair into the map, we compute the hash of the key and use that as the index into the list of buckets. Each bucket will store both the key and its value. In order to do this, we need a class to represent the key-value pair:

```java
private class KVPair
{
    public T key;
    public U value;

    public KVPair(T key, U value)
    {
        this.key = key;
        this.value = value;
    }
}
```

We have two class variables: a list called `data` whose elements are lists of key-value pairs, and an integer, `numElements`, to keep track of how many total key-value pairs are in the list. Here is the constructor:

```java
public HMap()
{
    data = new ArrayList<List<KVPair>>(10000);
    for (int i=0; i<10000; i++)
        data.add(new ArrayList<KVPair>());
    numElements = 0;
}
```
This implementation is just to help us learn how maps and hashing work, so we will just go with 10000 buckets. A more robust implementation would vary the number of buckets based on the how much is being stored in the map.

Figuring out a hash function for the keys is a little tricky since we don't know the data type of the key, so we'll just make use of Java's $hashCode$ method. Here it is:

```java
public int hashfunc(T key)
{
    return (key.hashCode() & Integer.MAX_VALUE) % 10000;
}
```

The use of the bitwise AND operator, &, might seem a bit bizarre. The reason has to do with the fact that the $hashCode$ function often produces negatives. We can't use negatives as indices in a list, so we must do something. One possibility would be to take the absolute value of the result returned by $hashCode$, but there is about a one in a four billion chance of a bug using this code. If the hashcode turns out to be $Integer.MIN_VALUE$, then by a quirk of how the absolute value function behaves, it will actually return a negative value. The solution to this is to bitwise and the hash code with $Integer.MAX_VALUE$, which consists of all F's in hexadecimal. This will guarantee a positive number. It might seem like overkill to worry about this, but considering that we might have maps containing millions of elements, this one-in-a-billion error will eventually show up in practice.

To add a key-value pair to the map, we compute the hash of the key, check to make sure the key is not already in the map, and then add the key-value pair to list of buckets.

```java
public void add(T key, U value)
{
    int hash = hashfunc(key);
    for (KVPair p:data.get(hash))
        if (p.key.equals(key))
            return;
    data.get(hash).add(new KVPair(key, value));
    numElements++;
}
```

To get the value associated with a key we compute the key's hash, and loop through the list associated with that hash until we find the key.

```java
public U get(T key)
{
    for (KVPair p: data.get(hashfunc(key)))
        if (p.key.equals(key))
            return p.value;
    return null;
}
```

This is all we will implement here. It's just the bare bones of a class. See Exercise 7 for more. Here is the entire class:

```java
import java.util.ArrayList;
import java.util.List;

public class HMapNew<T, U>
```
7.10 Maps in the Collections Framework

The Collections Framework has an interface called Map. There are three classes implementing this interface: TreeMap, HashMap and LinkedHashMap. These are analogous to the three classes that implement the Set interface: Just like with sets, TreeMap stores the keys in sorted order, LinkedHashMap stores them in the order they are added, and HashMap is all about speed, though LinkedHashMap has comparable performance. Here are some sample declarations:

```java
Map<String, Integer> map = new TreeMap<String, Integer>();
```
Map<String, Integer> map = new HashMap<String, Integer>();
Map<String, Integer> map = new LinkedHashMap<String, Integer>();

The first generic type is the key type, and the second is the value type. Here are some useful methods in the Map interface:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>containsKey(k)</td>
<td>returns true iff k is a key in the map</td>
</tr>
<tr>
<td>containsValue(v)</td>
<td>returns true iff v is a value in the map</td>
</tr>
<tr>
<td>keySet()</td>
<td>returns the keys of a as a set</td>
</tr>
<tr>
<td>values()</td>
<td>returns a Collection object of the values of the map</td>
</tr>
<tr>
<td>get(k)</td>
<td>returns the value associated with the key k</td>
</tr>
<tr>
<td>put(k, v)</td>
<td>adds the key-value pair (k, v) into the map</td>
</tr>
<tr>
<td>putAll(m)</td>
<td>adds all the key-value pairs from the map m into the map</td>
</tr>
<tr>
<td>remove(k)</td>
<td>removes a key k and its associated value from the map</td>
</tr>
</tbody>
</table>

We can iterate through the keys of a map by using the keySet method. For example, if the keys of map are strings, the following will print out all the values in the array:

```java
for (String s:map.keySet())
    System.out.println(map.get(s));
```

### 7.11 Applications of maps

Maps are useful tools for solving programming problems. Here is one example. Suppose we are writing a program to count the frequencies of words in a document. Our goal is to print out a list of words in the document and their frequencies, like so:

```
the: 213
a: 158
and: 53
...
```

We can solve this with a map whose keys are the words of the document and whose values are the frequencies of those words. If the map is called wordfreq, we could call wordfreq.get("the") to get the frequency of the word the. Exercise asks you to implement this approach.

### A Roman Numerals example

A nice place where a map can be used is to convert from Roman numerals to ordinary numbers. The keys are the Roman numerals and the values are their numerical values. We would like to then compute the numerical value by looping through the string and counting like below:
7.11. APPLICATIONS OF MAPS

```java
int count = 0;
for (char c : s.toCharArray())
    count += romanMap.get(c);
```

The problem with this is it doesn’t work with things like IV, which is 4. The trick we use to take care of this is to replace each occurrence of IV with some unused character, like a and add an entry to the map for it. The code is below:

```java
public static int fromRoman(String s) {
    Map<Character, Integer> romanMap = new HashMap<Character, Integer>;
    romanMap.put('M', 1000);
    romanMap.put('D', 500);
    romanMap.put('C', 100);
    romanMap.put('L', 50);
    romanMap.put('X', 10);
    romanMap.put('V', 5);
    romanMap.put('I', 1);
    romanMap.put('a', 900); // CM
    romanMap.put('b', 400); // CD
    romanMap.put('c', 90); // XC
    romanMap.put('d', 40); // XL
    romanMap.put('e', 9); // IX
    romanMap.put('f', 4); // IV

    s = s.replace("CM", "a");
    s = s.replace("CD", "b");
    s = s.replace("XC", "c");
    s = s.replace("XL", "d");
    s = s.replace("IX", "e");
    s = s.replace("IV", "f");

    int count = 0;
    for (char c : s.toCharArray())
        count += romanMap.get(c);
    return count;
}
```

A wordplay example

What we want to do is given a word, find all the words that can be formed from its letters. For instance, given water, we can form the words we, wear, are, tea, etc. This problem was described in Chapter 1.

One solution is, for a given word, form a map whose keys are the characters of the word and whose values are the number of times the character appears in the word. For instance, the map corresponding to java would be (a:2, j:1, v:1). A word w can be made from the letters of another word W if each key in the map of w also occurs in the map of W and the corresponding values for the letters of w are less than or equal to the values for the letters of W. Here is a program implementing this approach:

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Scanner;
```
public class WordFinder {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner file = new Scanner(new File("wordlist.txt"));
        List<String> wordlist = new ArrayList<String>();
        while (file.hasNextLine())
            wordlist.add(file.nextLine());
        System.out.println(wordsFromWord("water", wordlist));
    }
    public static List<String> wordsFromWord(String word, List<String> wordlist) {
        List<String> returnList = new ArrayList<String>();
        Map<Character, Integer> wordMap = getWordMap(word);
        for (String w: wordlist) {
            Map<Character, Integer> wMap = getWordMap(w);
            boolean goodWord = true;
            for (char c : wMap.keySet()) {
                if (!wordMap.containsKey(c) || wordMap.get(c) < wMap.get(c)) {
                    goodWord = false;
                    break;
                }
            }
            if (goodWord)
                returnList.add(w);
        }
        return returnList;
    }
    public static Map<Character, Integer> getWordMap(String word) {
        Map<Character, Integer> wordMap = new HashMap<Character, Integer>();
        for (char c: word.toCharArray()) {
            if (wordMap.containsKey(c))
                wordMap.put(c, wordMap.get(c) + 1);
            else
                wordMap.put(c, 1);
        }
        return wordMap;
    }
}

To understand how the program works, we will work from the bottom up. The getWordMap function builds up a map of the frequencies of the letters in a given word. The key part is the for loop below:

    for (char c: word.toCharArray()) {
        if (wordMap.containsKey(c))
            wordMap.put(c, wordMap.get(c) + 1);
        else
            wordMap.put(c, 1);
    }

It loops through the word, counting the number of occurrences of each character. For each character in
the word, the code checks to see if the character has been found before. If it has, it increments one to the character's count. If it hasn't, then it sets the character's count to 1.

Next, the `wordsFromWord` function finds the words that can be made from the main word. It starts by creating the map `wordMap` for the main word and then looping through a list of words (which was passed to it as a parameter), looking for those that work. The code below is what finds the words that work:

```java
for (String w:wordlist) {
    Map<Character, Integer> wMap = getWordMap(w);
    boolean goodWord = true;
    for (char c:wMap.keySet()) {
        if (!wordMap.containsKey(c) || wordMap.get(c)<wMap.get(c)) {
            goodWord = false;
            break;
        }
    }
    if (goodWord) returnList.add(w);
}
```

To check if an individual word works, we loop through its characters, checking to make sure that the frequency count in the word is less than the frequency count in the main word. If that isn't true for any character, then we break set a flag variable to `false` and break out of the character loop.

Finally, the `main` function just fills up the wordlist and calls `wordsFromWord`.

In summary, maps provide elegant ways to solve a number of tricky programming problems.

---

### 7.12 Exercises

1. In Section 7.2, we described a bit set. The way it works is the set's data is stored in an array or list of booleans such that index `i` in the arraylist is set to `true` if `i` is in the set and `false` otherwise.

   For instance, the set `{0, 1, 4, 6}` is represented by the following arraylist:

   ```java
   [true, true, false, false, true, false, true]
   ```

   The indices 0, 1, 4, and 6 are true, and the others are false.

   Write a `BitSet` class that works for sets of integers that contains the following methods:

   - `BitSet(n)` — a constructor that creates an empty set. The parameter `n` specifies the largest element the set will be able to hold.
   - `BitSet(list)` — a constructor that is given an `ArrayList` of integers and builds a set from it
   - `toString()` — returns a string containing the contents of the set bracketed by curly braces with individual elements separated by commas and spaces. For example: `{0, 1, 4, 6}` is how the set above would be returned.
   - `toArrayList()` — returns an `ArrayList` of integers containing the elements of the set
contains(x) — returns true or false depending on whether the set contains the element x
add(x) — inserts x into the set
remove(x) — removes x from the set
size() — returns the number of elements in the set
isEmpty() — returns true if the set is empty and false otherwise
getMax() — returns the largest element that the set is able to hold
union(s2) — returns the union of the set with the set s2 (return a set)
intersection(s2) — returns the intersection of the set with the set s2 (return a set)
difference(s2) — returns the difference of the set with the set s2 (return a set)

2. Add the following methods to the SimpleSet class.
   (a) isSubset(s) — returns true if s is a subset of the set and false otherwise
   (b) isSuperset(s) — returns true if s is a superset of the set and false otherwise
   (c) symmetricDifference(s) — returns the symmetric difference of the set with the set s. The symmetric difference \( A \sim B \) is defined as the elements in A but not B together with the elements in B but not in A.

3. This exercise is about an implementation of a set that combines ideas from bit sets and hash sets. In place of a list of buckets, it uses a list of booleans. To add an element to the set, compute its hash value (index in the list) and store true at that location. To check if an element is in the set, compute its hash and see if there is a true or false at that location. We will call this set FastAndDangerousSet because, while it is very efficient, it does nothing to deal with collisions. However, if the load factor is low and a few mistakes can be tolerated, then this is a useful implementation. The set should have add, remove, and contains methods.
   For this exercise, assume the set holds integer data and use a hash function of the form \( ax \mod b \), where \( a \) and \( b \) are relatively large prime numbers.

4. Implement a set using a binary search tree. Provide add, remove, contains, and union methods.

5. Using a set of dictionary words, find all the words that are also words backwards, like bard/drab or drawer/reward.

6. Add an iterator to the HSet class to make it possible to iterate over the elements using a foreach loop.

7. Add the following methods to the Map class from Section 7.8.
   (a) A constructor that builds map from a list of keys and a list of values
   (b) remove(key)
   (c) toString()
   (d) set(key, value)
   (e) containsKey(key)
7.12. **EXERCISES**

(f) `containsValue(value)`

(g) `size()`

(h) `isEmpty()`

(i) `getKeys()`

(j) `getValues()`

8. One use for maps is for implementing sparse integer arrays. A sparse integer array is one where most of the entries are 0, and only a few are nonzero. We can save a lot of space by using a map whose keys are the indices of nonzero entries and whose values are the corresponding entries. Create a class called `SparseArray` that uses this approach to store a sparse array. The class should have `get`, `set`, and `contains` methods.

9. The file `scores.txt` contains the results of every 2009-10 NCAA Division I basketball game (from [http://kenpom.com](http://kenpom.com)). Each line of that file looks like below:

11/09/2009 AlcornSt. 60 OhioSt. 100

Write a program that finds all the teams with winning records (more wins than losses) who were collectively (over the course of the season) outscored by their opponents. To solve the problem, use maps `wins`, `losses`, `pointsFor`, and `pointsAgainst` whose keys are the teams.

10. The substitution cipher encodes a word by replacing every letter of a word with a different letter. For instance every `a` might be replaced with an `e`, every `b` might be replaced with an `a`, etc. Write a program that asks the user to enter two strings. Then determine if the second string could be an encoded version of the first one with a substitution cipher. For instance, `CXYZ` is not an encoded version of `BOOK` because `O` got mapped to two separate letters. Also, `CXXK` is not an encoded version of `BOOK`, because `K` got mapped to itself. On the other hand, `CXXZ` would be an encoding of `BOOK`.

11. Maps are useful for frequency counting. Using a map, write a program that reads through a document and creates a map whose keys are the document's words and whose values are the frequencies of those words.

12. This problem will give you a tool for cheating at crossword puzzles. Prompt the user for a string that consists of the letters they know from the word and asterisks as placeholders for the letters they don't know. Then print out all the words that could work. For instance, if the user enters `w***er`, the program should print out `water`, `wafer`, `wider`, etc. Do this by creating a map from the user's word whose keys are the indices of the non-asterisk characters and whose values are the characters themselves. Then loop through the words in the wordlist and use the map to find the words that work.

13. As noted in Section 7.9, our implementation of a hash map uses a fixed number of buckets. Add a method to the class called `rehash` that doubles the number of buckets, and places the elements from the old buckets into the new buckets.


15. Add an iterator to the `HMap` class to make it possible to iterate over the keys using a `foreach` loop.
Chapter 8

Graphs

Graphs are generalizations of trees which allow for more complex relationships than parent-child ones. The figure below shows a tree graph and two other graphs.

![Graphs Diagram]

We see that a graph is a network of points and lines. The points are called vertices or nodes, and the lines are called edges or arcs. A lot of real-life problems can be modeled with graphs. For example, we can model a social network with a graph. The vertices are the people and there is an edge between two vertices if their corresponding people know each other. Asking questions about the social network, like the average number of people that people know or the average number of connections it takes to get from any one person to another can be answered by computing properties of the graph. In this chapter, we will write a graph class that will make computing these properties straightforward. So fundamental questions about a social network can be answered by representing the people by vertices, representing the relationship of knowing one another by edges, and running some simple graph algorithms on the resulting graph.

Another example of a use of graphs is modeling games. In the game Tic-tac-toe, we can create a game tree to represent the game. The vertices are the states of the game and there is an edge between states if there is single move that gets us from one state to another. Part of the tree is shown below.
CHAPTER 8. GRAPHS

Having represented the game this way, a computer player can scan through the game tree to find an optimal move. More complicated games, where it is possible to go back to a previous state can be represented by more general graphs.

One more example involves scheduling. Suppose we have several people who have to be at certain meetings at a conference. We want to schedule the meetings so that everyone who has to be at a meeting can be there. The main constraint then is that two meetings can’t be scheduled for the same time if there is someone who has to be at both. We can represent this as a graph if we let the vertices be the meetings, with an edge between two vertices if there is someone that has to be at both meetings. We then have to assign times to the vertices such that vertices that are connected by an edge must get different times. This is an example of what is called graph coloring and it can be used to model a lot of problems like this. If we implement an algorithm to find proper colorings of graphs, then all we have to do is represent our real-life problem with a graph and then run the algorithm on that.

In general, to model a problem with a graph, identify the fundamental units of the problem (people, game states, meetings) and the fundamental relationship between the units (acquaintance, can get from one state to the next by a single move, one person has to be at both meetings).

8.1 Terminology

As mentioned, the dots can be called vertices or nodes and the lines can be called edges or arcs. Different authors use different terminology. We will stick with vertices and edges. Here is a list of the most important terms:

- If there is an edge between two vertices, we say the vertices are adjacent
- The neighbors of a vertex are the vertices it is adjacent to. The neighborhood of a vertex is the set of all its neighbors.
- The degree of a vertex is the number of neighbors it has.
8.2 Implementing a graph class

A path in a graph is an alternating sequence of vertices and edges, with no repeated vertices or edges.

A cycle in a graph is a path that starts and ends at the same vertex.

A graph is said to be connected if it is possible to get from every vertex to every other vertex by a path.

To illustrate these definitions, consider the graph below.

![Graph Illustration]

The vertex c has three neighbors: a, b, and c. Therefore it has degree 3. On the other hand, vertex e has two neighbors, b and d, so it has degree 2. Some example paths are abe and acdeb. Some example cycles are acba, adcba, and adeba.

8.2 Implementing a graph class

We will examine two ways to implement graphs. The first, using an adjacency matrix is one that we will investigate only briefly. The idea is that we use a matrix (in this case a two-dimensional array of booleans) to keep track of which vertices are adjacent to which other vertices. We will assume the vertices are labeled by integers 0, 1, 2, …. If entry $a[i][j]$ in the adjacency matrix is true, then there is an edge between vertices i and j, and otherwise not. Here is the class:

```java
import java.util.List;
import java.util.ArrayList;

public class AGraph {
    private boolean[][] a;

    public AGraph() {
        a = new boolean[100][100];
    }

    public void addEdge(int u, int v) {
        a[u][v] = a[v][u] = true;
    }

    public void removeEdge(int u, int v) {
        a[u][v] = a[v][u] = false;
    }
}
```
Our implementation does not have many methods, just a few to give an idea of how the approach works. It is simple to code and works fine for small graphs. For a graph with \( n \) vertices, the array will have \( n^2 \) entries. Many real-life problems that are modeled with graphs have thousands or even millions of vertices, so having \( n^2 \) entries will not be feasible. In addition, it quite often happens that most vertices are adjacent to very few of the other vertices, so this implementation wastes a lot of space.

The second approach to implementing a graph is to use adjacency lists. For each vertex of the graph we keep a list of the vertices it is adjacent to. This is similar to the way we implemented binary trees. Each vertex had two nodes, left and right, which were links to its children. Here, instead of just two things, left and right, we will have a whole list of things that the vertex is adjacent to. The difference, though, is we will not use a linked structure. Instead we will use a map. We will allow the vertices to be of a generic type \( T \) so that they can represent strings, or integers, or some other object. These will be the keys of the map, and the values will be lists of neighbors of the key vertex.

Our graph class will have the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(v)</td>
<td>adds a vertex</td>
</tr>
<tr>
<td>addEdge(u,v)</td>
<td>adds an edge between vertices u and v</td>
</tr>
<tr>
<td>contains(v)</td>
<td>returns whether or not v is a vertex in the graph</td>
</tr>
<tr>
<td>containsEdge(u,v)</td>
<td>returns whether or not there is an edge between u and v</td>
</tr>
<tr>
<td>remove(v)</td>
<td>removes vertex v from the graph</td>
</tr>
<tr>
<td>removeEdge(u,v)</td>
<td>removes the edge between u and v from the graph</td>
</tr>
<tr>
<td>size()</td>
<td>returns how many vertices are in the graph</td>
</tr>
<tr>
<td>numberOfEdges()</td>
<td>returns how many edges are in the graph</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>returns whether or not there are any vertices in the graph</td>
</tr>
<tr>
<td>degree(v)</td>
<td>returns how many neighbors vertex v has</td>
</tr>
<tr>
<td>neighbors(v)</td>
<td>return a list of the neighbors of vertex v</td>
</tr>
</tbody>
</table>

In addition, it will be occasionally be convenient to work with edges as objects themselves. A little later we will create an Edge object and we will add methods like addEdge(e) where the user can specify an Edge object to add.

To start, the class will have one variable, a map called neighbors whose keys are vertices (of type \( T \)) and whose values are lists of the vertices that each key vertex is adjacent to. Here is the constructor that creates an empty graph:

```java
public List<Integer> neighbors(int v)
{
    List<Integer> neighborList = new ArrayList<Integer>();
    for (int i=0; i<100; i++)
        if (a[v][i]==true)
            neighborList.add(i);
    return neighborList;
}
```
8.2. IMPLEMENTING A GRAPH CLASS

```java
public Graph()
{
    neighbors = new LinkedHashMap<T, List<T>>();
}
```

We use a LinkedHashMap to store the vertices in the order they are added. To add a vertex we add it to the map, doing a check to make sure the vertex is not already in the graph:

```java
public void add(T v)
{
    if (!neighbors.containsKey(v))
        neighbors.put(v, new ArrayList<T>());
}
```

To remove a vertex is a little trickier. We not only have to remove it from the map, but we also have to remove all the edges that use that vertex. To do that, we loop through the map and remove the vertex from all the neighbor lists. If the vertex isn't in the list, remove doesn't change the list.

```java
public void remove(T v)
{
    neighbors.remove(v);
    for (T u:neighbors.keySet())
        neighbors.get(u).remove(v);
}
```

Many of the graph methods are easy to implement. In fact, the contains, size, neighbors, degree, and vertices methods can all be done in one line. Shown below are the one-line solutions:

<table>
<thead>
<tr>
<th>Method</th>
<th>Return statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td><code>return neighbors.containsKey(v);</code></td>
</tr>
<tr>
<td>size</td>
<td><code>return neighbors.size();</code></td>
</tr>
<tr>
<td>neighbors</td>
<td><code>return neighbors.get(v);</code></td>
</tr>
<tr>
<td>degree</td>
<td><code>return neighbors.get(v).size();</code></td>
</tr>
<tr>
<td>vertices</td>
<td><code>return new List&lt;T&gt;(neighbors.keySet());</code></td>
</tr>
</tbody>
</table>

To add an edge between u and v, we add u to the neighbor list of v and vice-versa. It is important that we do both.

```java
public void addEdge(T u, T v)
{
    neighbors.get(v).add(u);
    neighbors.get(u).add(v);
}
```

Removing an edge is the inverse of adding it:

```java
public void removeEdge(T u, T v)
{
    neighbors.get(u).remove(v);
    neighbors.get(v).remove(u);
}
```

Returning whether the graph contains an edge between u and v is straightforward:
public boolean containsEdge(T u, T v) {
    return neighbors.get(u).contains(v);
}

Earlier, we mentioned that it is convenient to work with the edges as objects themselves. So we will create an Edge class. An edge is described by its endpoints, so the class will have two fields, one for each endpoint. The trick with this class is that we would like to use it in lists and do operations like list.contains(e). If we add an edge between vertices u and v to the list, checking later to see if that edge is in the list with something like the following will not work:

    list.contains(new Edge<T>(u, v)

The reason is that the contains method will rely on the edge class’s equals method to determine if the desired edge is in the list. The equals method will only return true if the two edges are the exact same object, i.e. occupy the same space in memory, which will not usually be true. We must override the equals method describing how to compare to edges. The proper comparison is that the two edges share the same endpoints. Also, because we may want to use the edges in a HashSet or HashMap, we will have to override the hashCode method. Our approach here won’t be particularly creative. We just have to make sure that the returned hash does not depend on the order in which the endpoints are specified. Finally, our class includes a toString method. Here is the entire Edge class:

```java
public class Edge<T> {
    public T u, v;

    public Edge(T u, T v) {
        this.u = u;
        this.v = v;
    }

    public boolean equals(Object obj) {
        Edge<T> other = (Edge<T>) obj;
        if (u.equals(other.u) && v.equals(other.v) ||
            u.equals(other.v) && v.equals(other.u))
            return true;
        return false;
    }

    public int hashCode() {
        return u.hashCode() + v.hashCode();
    }

    public String toString() {
        return "(" + u + ", " + v + ")";
    }
}
```

With the Edge class completed, we make versions of addEdge, removeEdge, and containsEdge to work with the new class. The methods are implemented very similarly to their equivalents above.

```java
public void addEdge(Edge<T> e) {
```
8.2. IMPLEMENTING A GRAPH CLASS

neighbors.get(e.v).add(e.u);
neighbors.get(e.u).add(e.v);
}

public void removeEdge(Edge<T> e)
{
    neighbors.get(e.v).remove(e.u);
    neighbors.get(e.u).remove(e.v);
}

public boolean containsEdge(Edge<T> e)
{
    return neighbors.get(e.u).contains(e.v);
}

We also would like a method that returns a list of the edges of the graph. To do that, we loop through all
the vertices in neighbors and through each vertex's list of neighbors. The one trick here is that by doing
this, each edge will end up in the list twice. We take care of this by building a set of neighbors and then
converting that set to a list.

public List<Edge<T>> edges()
{
    Set<Edge<T>> edgeSet = new LinkedHashSet<Edge<T>>();
    for (T u:neighbors.keySet())
        for (T v:neighbors.get(u))
            edgeSet.add(new Edge<T>(u, v));
    return new ArrayList<Edge<T>>(edgeSet);
}

The last method we will consider for now is to return the number of edges in the graph. We can use the
edges method we just wrote for this:

public int numberOfEdges()
{
    return edges().size();
}

Finally, it would be nice to make the class iterable so that if we have a graph whose vertices are, say,
strings, we could loop over the vertices of the graph like so:

for (String s:graph)
    System.out.println(s);

To do this, we implement Iterable. Recall from Section 2.9 that we need a method called iterator. How-
ever, unlike the linked list code where we had to create a class that specified how to iterate through
the list, here things are much easier. The vertices are the keys of the neighbors map, and we can get
at them by calling neighbors.keySet() which returns a set. Java's sets are already iterable and we can
piggyback off of that to make our Graph class iterable. All we have to do is the following:

public Iterator<T> iterator()
{
    return neighbors.keySet().iterator();
}

Here the complete class:
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.LinkedHashMap;
import java.util.LinkedHashSet;
import java.util.Iterator;
import java.util.Set;

public class Graph<T> implements Iterable<T> {
    private Map<T, List<T>> neighbors;

    public Graph() {
        neighbors = new LinkedHashMap<T, List<T>>();
    }

    public void add(T v) {
        if (!neighbors.containsKey(v))
            neighbors.put(v, new ArrayList<T>());
    }

    public void remove(T v) {
        neighbors.remove(v);
        for (T u : neighbors.keySet())
            neighbors.get(u).remove(v);
    }

    public boolean contains(T v) {
        return neighbors.containsKey(v);
    }

    public int size() {
        return neighbors.size();
    }

    public List<T> neighbors(T v) {
        return neighbors.get(v);
    }

    public int degree(T v) {
        return neighbors.get(v).size();
    }

    public List<T> vertices() {
        return new ArrayList<T>(neighbors.keySet());
    }

    public void addEdge(T u, T v) {
        neighbors.get(v).add(u);
        neighbors.get(u).add(v);
    }

    public void removeEdge(T u, T v) {
        neighbors.get(u).remove(v);
    }
}
8.2. IMPLEMENTING A GRAPH CLASS

```java
neighbors.get(v).remove(u);
}

public void addEdge(Edge<T> e)
{
    neighbors.get(e.v).add(e.u);
    neighbors.get(e.u).add(e.v);
}

public void removeEdge(Edge<T> e)
{
    neighbors.get(e.v).remove(e.u);
    neighbors.get(e.u).remove(e.v);
}

public boolean containsEdge(T u, T v)
{
    return
    neighbors.get(u).contains(v);
}

public boolean containsEdge(Edge<T> e)
{
    return
    neighbors.get(e.u).contains(e.v);
}

public List<Edge<T>> edges()
{
    Set<Edge<T>> edgeSet = new LinkedHashSet<Edge<T>>;
    for (T u:neighbors.keySet())
        for (T v:neighbors.get(u))
            edgeSet.add(new Edge<T>(u, v));
    return new ArrayList<Edge<T>>(edgeSet);
}

public int numberOfEdges()
{
    return edges().size();
}

public Iterator<T> iterator()
{
    return neighbors.keySet().iterator();
}
```

An application

Here is an application of the graph class. A word ladder is a type of puzzle where you are given two words and you have to get from one to another by changing a single letter at a time, with each intermediate word being a real word. For example to get from time to rise, we could do time → tide → ride → rise. We can represent this as a graph with the words being the vertices and an edge between vertices if their corresponding words are one letter away from each other. Solving a word ladder amounts to finding a path in the graph between two vertices. In a coming section we will learn how to find the path, but for now we will just create the graph.

The first thing we need is a function to tell if two strings are one character away from each other.

```java
public static boolean oneAway(String s1, String s2)
{
```
if (s1.length()!=s2.length())
    return false;

int count=0;
for (int i=0; i<s1.length(); i++)
{
    if (s1.charAt(i)!=s2.charAt(i))
    {
        count++;
        if (count>1)
            return false;
    }
}
if (count==0)
    return false;
return true;
}

The way it works is we first return false if the strings are not of the same length. Otherwise, we count the number of letters in which the strings differ and only return true if that count is 1. To speed things up a little, when we do the counting, we return false as soon as the count is greater than 1, as it is not necessary to keep scanning the rest of the string.

We will focus only on five-letter words here, though the program is easy to modify to work with other word sizes. The approach we will take is too slow to include all the words of the wordlist in the graph.

To create the graph, can start by looping through a wordlist file, adding each five-letter word to the graph as a vertex.

Scanner file = new Scanner(new File("wordlist.txt"));
while (file.hasNextLine())
{
    String word = file.nextLine();
    if (word.length()==5)
        g.add(word);
}

We then use nested for loops to loop over all pairs of vertices, calling oneAway on each pair to see if we need to add an edge.

for (String word1:g)
    for (String word2:g)
        if (oneAway(word1, word2))
            g.addEdge(word1,word2);

This is not the most efficient way to do things, but it is quick to code and works reasonably quickly. Finally, to get the neighbors of a word, we call the neighbors method. For example, with the word water, we get the following list of neighbors:

[wafer, rater, wager, later, hater, eater, tater, dater, waver, wader, cater]

Here is the entire program:
8.3 Searching

In a graph, searching is the process of finding a path from one vertex to another. The two most basic types of search are breadth-first search (BFS) and depth-first search (DFS). To get a handle on these we will consider them first on trees. The figure below shows the order that each search visits the vertices of a tree, starting at the root.
BFS first checks all of the children of the starting vertex, then all of their children, etc. It always completes a level before moving on to the next level. DFS works its way down the tree until it can't go any further then backs up to the first place where there is another route heading down that hasn't already been checked. It keeps this up until all routes have been checked.

We can extend these algorithms to graphs in general. We just need to keep a list of all the vertices that have previously been visited by the searching algorithm to prevent a cycle in the graph from causing an infinite loop.

### Implementing BFS

A standard way to implement BFS is with a queue. The queue holds all the vertices that are waiting to be checked. The queue starts off with the starting vertex and the algorithm finishes when either the ending vertex is found or when the queue is empty (indicating there is no path from start to end).

At each step of the algorithm, we get the next item from the queue and look at all its neighbors. If one of those is the end vertex, then we're done. Otherwise we add to the queue any of those neighbors that haven't already been checked. Because we are using a queue, vertices will be checked in the order they are added to the queue, and since all the neighbors of a vertex are all added at once, we will end up checking an entire level before moving to the next one, which is how a BFS is defined.

We also maintain a map parentMap that keeps track of the parents of each vertex that has been reached. The vertices are the keys and the parents are the values. So we can use the keys as our list of vertices that have been found. And, when we reach the goal vertex, we can use the parentMap to trace the shortest path back to the start and return that.

Here is the algorithm:

```java
public List<T> bfs(T start, T end) {
    Deque<T> queue = new ArrayDeque<T>();
    HashMap<T, T> parentMap = new HashMap<T, T>();
    queue.add(start);
    parentMap.put(start, null);
    while (!queue.isEmpty()) {
        T v = queue.remove();
        for (T n: neighbors(v)) {
            if (n == end) {
                return tracePath(v, end, parentMap);
            }
            queue.add(n);
            parentMap.put(n, v);
        }
    }
    return null;
}
```

BFS first checks all of the children of the starting vertex, then all of their children, etc. It always completes a level before moving on to the next level. DFS works its way down the tree until it can't go any further then backs up to the first place where there is another route heading down that hasn't already been checked. It keeps this up until all routes have been checked.

We can extend these algorithms to graphs in general. We just need to keep a list of all the vertices that have previously been visited by the searching algorithm to prevent a cycle in the graph from causing an infinite loop.
Implementing DFS

Implementing DFS is similar to implementing BFS. The main difference is the queue becomes a stack. With a queue, vertices were checked in the order they were added, which meant that since all the neighbors of a vertex were added at once, they would be checked one after another. With a stack, the next vertex to be checked is the most recently added vertex. So when we are checking a vertex and we add its children, its children put are higher in the stack than its siblings and so they are checked first. In short, BFS will always move to a brother or sister before checking a child, whereas DFS will do the opposite.

Like the BFS implementation, we keep checking vertices until the stack is empty or the goal vertex has been reached. At each step, we pop the top item off the stack and add its unchecked neighbors to the stack. Instead of a map of parents, we can get away here using a simple list that keeps track of the path from the start to the end. Each time we pop something off the stack we add it to the path, and we remove an item from the path if all of its neighbors turn out to have no unchecked neighbors.

Here is the code for the algorithm:

```java
public List<T> dfs(T start, T end)
{
    Deque<T> stack = new ArrayDeque<T>();
    List<T> path = new ArrayList<T>();
    stack.push(start);
    while (!stack.isEmpty())
    {
        T v = stack.pop();
        path.add(v);
        boolean foundNewOne = false;
        for (T n : neighbors(v))
        {
            if (!path.contains(n))
            {
                foundNewOne = true;
                queue.add(n);
                parentMap.put(n, v);
            }
        }
        if (n.equals(end))
        {
            List<T> returnList = new ArrayList<T>();
            T u = end;
            while (u != null)
            {
                returnList.add(u);
                u = parentMap.get(u);
            }
            Collections.reverse(returnList);
            return returnList;
        }
    }
    return new ArrayList<T>();
}
```
It would be straightforward to modify both of these methods to find a path from the start to any vertex in a list of possible end vertices. See Exercise 4.

When I ran both of these searches to find a word ladder from time to rise, BFS returned the ladder time → rime → rise, this being the shortest possible word ladder. DFS returned a list of 698 words, more than a third of the number of words in the list. This illustrates a general principle: BFS will find the shortest, most direct solution, while DFS usually will not.

Which search to use depends on the structure of the graph. If each vertex has a high degree, then since BFS searches every neighbor before moving on to the next level, BFS might take too long to find a solution that is several layers deep. However, if there are many possible end vertices, DFS might have a better chance of finding them. But in general, especially for small problems, DFS and BFS run quickly and the solution given by BFS will most be the most direction solution.

BFS and DFS are among the most straightforward searching algorithms and are the building blocks for more sophisticated searches. Do a web search or consult a textbook on algorithms for more.

8.4 Digraphs

A digraph is a graph in which the edges can be one-way (directed). The figure below shows a digraph.

Digraphs model in many real-life situations. For instance, a road network can be represented as a graph with the edges being roads and vertices being intersections or dead-ends. Directed edges could be used for one-way roads. Another example would be the graph of a game where the vertices are the states, with an edge between two vertices if it is possible to go between their corresponding states. In many games, once you make a move, you can’t go back, so directed edges would be appropriate here.

To implement a digraph, we can use most of our graph class, but we have to change around a few
things. For one, we have to change the Edge class. We will create a new class EdgeD. The difference is in the equals and hashCode methods. In the Edge class, two edges are equal if they share the same endpoints, with the order of the endpoints not mattering. In the EdgeD class, order now matters. Here is the relevant line in both methods:

Edge — \( u.\text{equals}(other.u) \&\& v.\text{equals}(other.v) \) || \( u.\text{equals}(other.v) \&\& v.\text{equals}(other.u) \)

EdgeD — \( u.\text{equals}(other.u) \&\& v.\text{equals}(other.v) \)

A similar concern applies to the hashCode method:

Edge — return \( u.\text{hashCode()} + v.\text{hashCode()} \);

EdgeD — return \( u.\text{hashCode()} - v.\text{hashCode()} \);

This isn’t the most sophisticated approach, but using a minus sign instead of a plus sign means that if the order of the endpoints is different, then the hashes will likely be different.

Here is the whole EdgeD class:

```java
public class EdgeD<T>
{
    public T u, v;

    public EdgeD(T u, T v)
    {
        this.u = u;
        this.v = v;
    }

    public boolean equals(Object obj)
    {
        EdgeD<T> other = (EdgeD<T>) obj;
        if (u.equals(other.u) \&\& v.equals(other.v))
            return true;
        return false;
    }

    public int hashCode()
    {
        return u.hashCode() - v.hashCode();
    }

    public String toString()
    {
        return "(" + u + ", " + v + ")";
    }
}
```

In terms of the Digraph class itself, all of the vertex code is the same as in Graph. It’s only the edge code that is different. In the next section we will be implementing one other type of graph, a weighted graph, that also shares its vertex code with Graph, but has different edge code. So it might seem like it would be good to have the Digraph and WeightedGraph classes inherit from the Graph class. However, the Digraph class’s edge methods work with EdgeD instead of Edge and the call signatures of the WeightedEdge class’s edge methods will be different still. So inheriting from Graph won’t work so well. Instead, we will create an abstract class that each of the three classes will inherit from. The abstract class consists of all of the vertex methods as well as BFS and DFS. Each of the child classes will implement their own edge methods. The neighbors field is now protected instead of private so that the child classes can easily
use it. Here is the abstract class:

```java
import java.util.ArrayDeque;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Deque;
import java.util.Map;
import java.util.LinkedHashMap;
import java.util.Iterator;

public abstract class AbstractGraph<T> implements Iterable<T> {
    protected Map<T, List<T>> neighbors;

    public AbstractGraph() {
        neighbors = new LinkedHashMap<T, List<T>>();
    }

    public void add(T v) {
        if (!neighbors.containsKey(v))
            neighbors.put(v, new ArrayList<T>());
    }

    public void remove(T v) {
        neighbors.remove(v);
        for (T u : neighbors.keySet())
            neighbors.get(u).remove(v);
    }

    public boolean contains(T v) {
        return neighbors.containsKey(v);
    }

    public int size() {
        return neighbors.size();
    }

    public List<T> neighbors(T v) {
        return neighbors.get(v);
    }

    public int degree(T v) {
        return neighbors.get(v).size();
    }

    public List<T> vertices() {
        return new ArrayList<T>(neighbors.keySet());
    }

    public boolean containsEdge(T u, T v) {
        return neighbors.get(u).contains(v);
    }

    public boolean containsEdge(Edge<T> e) {
        return containsEdge(e.getSource(), e.getTarget());
    }
}
```
8.4. DIGRAPHS

```java
return neighbors.get(e.u).contains(e.v);
}

public Iterator<T> iterator()
{
    return neighbors.keySet().iterator();
}

public List<T> dfs(T start, T end)
{
    Deque<T> stack = new ArrayDeque<T>();
    List<T> path = new ArrayList<T>();
    stack.push(start);

    while (!stack.isEmpty())
    {
        T v = stack.pop();
        path.add(v);

        boolean foundNewOne = false;
        for (T n:neighbors(v))
        {
            if (!path.contains(n))
            {
                stack.push(n);
                foundNewOne = true;
            }
            if (n.equals(end))
            {
                path.add(end);
                return path;
            }
        }
    }
    if (!foundNewOne)
    path.remove(path.size()-1);
    return new ArrayList<T>();
}

public List<T> bfs(T start, T end)
{
    Deque<T> queue = new ArrayDeque<T>();
    HashMap<T, T> parentMap = new HashMap<T, T>();
    queue.add(start);
    parentMap.put(start, null);

    while (!queue.isEmpty())
    {
        T v = queue.remove();
        for (T n:neighbors(v))
        {
            if (!parentMap.containsKey(n))
            {
                queue.add(n);
                parentMap.put(n, v);
            }
            if (n.equals(end))
            {
                List<T> returnList = new ArrayList<T>();
                T u = end;
                while (u!=null)
                {
                    returnList.add(u);
                }
            }
        }
    }
    return new ArrayList<T>();
}
```
Here is what the Graph class now becomes.

```java
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedHashSet;
import java.util.Set;

public class Graph<T> extends AbstractGraph<T> {
    public Graph() {
        super();
    }

    public void addEdge(T u, T v) {
        if (neighbors.get(v).contains(u))
            return;
        neighbors.get(v).add(u);
        neighbors.get(u).add(v);
    }

    public void removeEdge(T u, T v) {
        neighbors.get(u).remove(v);
        neighbors.get(v).remove(u);
    }

    public void addEdge(Edge<T> e) {
        if (neighbors.get(e.v).contains(e.u))
            return;
        neighbors.get(e.v).add(e.u);
        neighbors.get(e.u).add(e.v);
    }

    public void removeEdge(Edge<T> e) {
        neighbors.get(e.v).remove(e.u);
        neighbors.get(e.u).remove(e.v);
    }

    public boolean containsEdge(T u, T v) {
        return neighbors.get(u).contains(v);
    }

    public boolean containsEdge(Edge<T> e) {
        return neighbors.get(e.u).contains(e.v);
    }

    public List<Edge<T>> edges()
```
Now to implement the `Digraph` class, we just have to change around a few of the edge methods of the `Graph` class. For example, here is how the `addEdge` method changes from the `Graph` class to the `Digraph` class.

First the method in the `Graph` class:
```java
public void addEdge(T u, T v)
{
    if (neighbors.get(v).contains(u))
        return;
    neighbors.get(v).add(u);
    neighbors.get(u).add(v);
}
```

Here is the method in the `Digraph` class.
```java
public void addEdge(T u, T v)
{
    if (neighbors.get(u).contains(v))
        return;
    neighbors.get(u).add(v);
}
```

The only difference is since the edge is only directed from `u` to `v`, we don't add `u` as a neighbor of `v`, like in the `Graph` class. Other than some similar changes in other methods and replacing some references to `Edge` with references to `EdgeD`, there are very few changes from the `Graph` class. Here is the entire `Digraph` class:
```java
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedHashSet;
import java.util.Set;

public class Digraph<T> extends AbstractGraph<T>
{
    public Digraph()
    {
        super();
    }

    public void addEdge(T u, T v)
    {
        if (neighbors.get(u).contains(v))
            return;
        neighbors.get(u).add(v);
    }
}
```
8.5 An application of digraphs

Here is an application of digraphs to a classic puzzle problem. Suppose we have two pails, one that holds 7 gallons and another that holds 11 gallons. We have a fountain of water where we can fill up and dump out our pails. Our goal is to get exactly 6 gallons just using these two pails. The rules are that you can only 1) fill up a pail to the top, 2) dump a pail completely out, or 3) dump the contents of one pail into another until the pail is empty or the other is full. There is no guestimating allowed.

For example, maybe we could start by filling up the 7-gallon pail, then dumping it into the 11-gallon pail. We could then fill up the 7-gallon pail and then dump it into the 11-gallon again, leaving 3 gallons in the 7-gallon pail. Maybe then we could dump the 11-gallon pail out completely. We could keep going doing operations like this until we 6 gallons in one of the pails.

We can represent this problem with a digraph. The vertices are the possible states, represented as pairs \((x, y)\), where \(x\) and \(y\) are the amount of water in the 7- and 11-gallon pails, respectively. There is an edge between two vertices if it is possible to go from their corresponding states, following the rules of the problem. For example, there is an edge between \((3, 0)\) and \((0, 0)\) because starting at the \((3, 0)\) state, we can dump out the 7-gallon pail to get to the \((0, 0)\) state. However, there is not an edge going in the opposite direction because there is no way to go from \((0, 0)\) to \((3, 0)\) in one step following the rules.
To create the graph, we first need a class to represent the states. The class has two integer fields, \( x \) and \( y \). We also override the `equals` method because we will need to test states for equality, and while we're at it, we will also override the `hashCode` and `toString` methods. This helps to make the class a little more usable if we want to use it elsewhere.

```java
public class Pair
{
    public int x, y;

    public Pair(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object obj)
    {
        Pair other = (Pair) obj;
        return other.x==x && other.y==y;
    }

    public int hashCode()
    {
        Integer x = this.x, y = this.y;
        return x.hashCode()-y.hashCode();
    }

    public String toString()
    {
        return "(" + x + ", " + y + ")";
    }
}
```

The states for the problem are all pairs \((x, y)\), where \(0 \leq x \leq 7\) and \(0 \leq y \leq 11\). Here is how we add those states to our digraph \( g \):

```java
for (int x=0; x<=7; x++)
    for (int y=0; y<=11; y++)
        g.add(new Pair(x,y));
```

To add the edges, we will loop over all vertices in the graph and add the appropriate edges for each. Here is the start of the loop:

```java
for (Pair p: g)
{
    int x=p.x, y=p.y;
    g.addEdge(p, new Pair(x,0));
    g.addEdge(p, new Pair(0,y));
    g.addEdge(p, new Pair(x,11));
    g.addEdge(p, new Pair(7,y));
    // loop continues...
```

The four edges added above come from completely dumping out or completely filling one of the pails. There are other edges to add that come from dumping one of the pails into the other. These are a little trickier because there are a couple of cases to consider depending on whether the dumping completely empties or completely fills a pail. For instance, suppose we have 3 gallons in the 7-gallon pail. If there
are 5 gallons in the 11-gallon pail, then $(3, 5) \rightarrow (0, 8)$ empties the 7-gallon pail, whereas if there are 9 gallons in the 11-gallon pail, then $(3, 9) \rightarrow (1, 11)$ completely fills the 11-gallon pail but doesn't empty the 7-gallon. We also have to avoid trying to pour into an already full pail or trying to pour from an empty pail. Here is the code that takes care of all of this:

```java
// continuing the loop
if (x>0 && y<11)
{
    if (x+y>11)
        g.addEdge(p, new Pair(x-(11-y),11));
    else
        g.addEdge(p, new Pair(0, x+y));
}
if (y>0 && x<7)
{
    if (x+y>7)
        g.addEdge(p, new Pair(7,y-(7-x)));
    else
        g.addEdge(p, new Pair(x+y,0));
}
```

Once the digraph is complete, solving the puzzle boils down to using a BFS or DFS. Here is the complete class. Note that we use variables \(m\) and \(n\) in place of 7 and 11 so that the program will work with any size pails.

```java
public class WaterFountain
{
    public static void main(String[] args)
    {
        WaterFountain wf = new WaterFountain();
        Digraph<Pair> g = new Digraph<Pair>();

        int m=7, n=11;
        for (int x=0; x<=m; x++)
            for (int y=0; y<=n; y++)
                g.add(new Pair(x,y));

        for (Pair p: g)
        {
            int x=p.x, y=p.y;
            g.addEdge(p, new Pair(x,0));
            g.addEdge(p, new Pair(0,y));
            g.addEdge(p, new Pair(x,n));
            g.addEdge(p, new Pair(m,y));

            if (x>0 && y<n)
            {
                if (x+y>n)
                    g.addEdge(p, new Pair(x-(n-y),n));
                else
                    g.addEdge(p, new Pair(0, x+y));
            }
            if (y>0 && x<m)
            {
                if (x+y>m)
                    g.addEdge(p, new Pair(m,y-(m-x)));
                else
```
8.6. WEIGHTED GRAPHS

When we run it, we get the following output:

```
[(0, 0), (7, 0), (0, 7), (7, 7), (3, 11), (3, 0), (0, 3), (7, 3), (0, 10),
 (7, 10), (6, 11), (6, 0)]
```

This solution, given by BFS, is the most direct solution. DFS gives a longer solution:

```
[(0, 0), (7, 0), (0, 7), (7, 7), (3, 11), (7, 11), (0, 11), (7, 4), (0, 4),
 (4, 0), (4, 11), (7, 8), (0, 8), (7, 1), (0, 1), (1, 0), (1, 11), (7, 5),
 (0, 5), (5, 0), (5, 11), (7, 9), (0, 9), (7, 2), (0, 2), (2, 0), (2, 11),
 (7, 6), (0, 6), (6, 0)]
```

8.6 Weighted graphs

There is one more type of graph that we will consider, called a weighted graph. An example weighted graph is shown below. A weighted graph is like an ordinary graph except that the edges are given weights. We can think of edge weights as the cost of using edges.

As an example, using a graph to model a network of cities, edge weights could represent the cost to get from one city to the other.

Our implementation of a weighted graph will also rely on the abstract graph class, but it will be a little more involved than the other classes. One option is to create a new edge class that has an extra field for the weight. Instead of that, we will add a map to our class called `edgeMap` whose keys are edges (of type Edge) and whose values are the weights. With this, the edge methods are very similar to the edge methods from the Graph class, just with an additional line for the map. For example:

```java
public void addEdge(T u, T v, int weight)
```
The edges method is simpler than with the Graph class because we can make use of the map:

```java
public List<Edge<T>> edges()
{
    return new ArrayList<Edge<T>>(edgeMap.keySet());
}
```

Our class will have a few methods that are not present in the Graph class. One returns a copy of edgeMap. In making the copy, it is good to make copies of all the edges rather than sending along references to the original edge objects.

```java
public Map<Edge<T>, Integer> getEdgeMap()
{
    Map<Edge<T>, Integer> returnMap = new LinkedHashMap<Edge<T>, Integer>();
    for (Edge<T> key: edgeMap.keySet())
    {
        returnMap.put(new Edge<T>(key.u, key.v), edgeMap.get(key));
    }
    return returnMap;
}
```

We also have methods to get and set the weight of an edge.

```java
public int getWeight(Edge<T> e)
{
    return edgeMap.get(e);
}

public int getWeight(T u, T v)
{
    return edgeMap.get(new Edge<T>(u, v));
}

public void setWeight(T u, T v, int weight)
{
    if (this.containsEdge(new Edge<T>(u, v)))
    {
        edgeMap.put(new Edge<T>(u, v), weight);
    }
}

public void setWeight(Edge<T> e, int weight)
{
    if (this.containsEdge(e))
    {
        edgeMap.put(e, weight);
    }
}
```

These are mostly simple. The only trick is to make sure the edge exists before we set the weight; otherwise, we would be adding an edge to the edge map without updating the neighbors map. Here is the entire class:

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
```
```java
import java.util.Map;
import java.util.LinkedHashMap;

public class WeightedGraph<T> extends AbstractGraph<T> {
    private Map<Edge<T>, Integer> edgeMap;

    public WeightedGraph() {
        super();
        edgeMap = new LinkedHashMap<Edge<T>, Integer>();
    }

    public void addEdge(T u, T v, int weight) {
        if (neighbors.get(v).contains(u))
            return;
        neighbors.get(v).add(u);
        neighbors.get(u).add(v);
        edgeMap.put(new Edge<T>(u, v), weight);
    }

    public void removeEdge(T u, T v) {
        neighbors.get(u).remove(v);
        neighbors.get(v).remove(u);
        edgeMap.remove(new Edge<T>(u, v));
    }

    public void addEdge(Edge<T> e, int weight) {
        if (neighbors.get(e.v).contains(e.u))
            return;
        neighbors.get(e.v).add(e.u);
        neighbors.get(e.u).add(e.v);
        edgeMap.put(e, weight);
    }

    public void removeEdge(Edge<T> e) {
        neighbors.get(e.v).remove(e.u);
        neighbors.get(e.u).remove(e.v);
        edgeMap.remove(e);
    }

    public boolean containsEdge(T u, T v) {
        return neighbors.get(u).contains(v);
    }

    public boolean containsEdge(Edge<T> e) {
        return neighbors.get(e.u).contains(e.v);
    }

    public List<Edge<T>> edges() {
        return new ArrayList<Edge<T>>(edgeMap.keySet());
    }

    public int numberOfEdges() {
        return edgeMap.keySet().size();
    }
}
```
```java
public HashMap<Edge<T>, Integer> getEdgeMap() {
    HashMap<Edge<T>, Integer> returnMap = new HashMap<Edge<T>, Integer>();
    for (Edge<T> key: edgeMap.keySet())
        returnMap.put(new Edge<T>(key.u, key.v), edgeMap.get(key));
    return returnMap;
}

public int getWeight(Edge<T> e) {
    return edgeMap.get(e);
}

public int getWeight(T u, T v) {
    return edgeMap.get(new Edge<T>(u, v));
}

public void setWeight(T u, T v, int weight) {
    if (this.containsEdge(new Edge<T>(u, v)))
        edgeMap.put(new Edge<T>(u, v), weight);
}

public void setWeight(Edge<T> e, int weight) {
    if (this.containsEdge(e))
        edgeMap.put(e, weight);
}
```

### 8.7 Kruskal’s algorithm for minimum spanning trees

One thing people are interested in about weighted graphs is finding a *minimum spanning tree*. A spanning tree of a weighted graph is a subgraph of the graph that includes all the vertices and has no cycles. A minimum spanning tree is a spanning tree where the sum of the weights of the edges in the tree is as small as possible. A minimum spanning tree is highlighted in the graph below.

One way to think of this is that the vertices are cities, the edges are potential roads, and the weights are the costs of building the roads. Say we want to connect all the cities so that it is possible to travel along the network from any city to any other. A minimum spanning tree will give the cheapest way of doing this. There are many other applications of minimum spanning trees.
There are two common algorithms for finding minimum spanning trees. The one we will look at is Kruskal's algorithm. It is a surprising algorithm in that it do just about the simplest thing you can think of and it ends up giving the optimal answer. At each step in the algorithm, pick the remaining edge that has the smallest weight (breaking ties arbitrarily) and add it to the tree, provided adding it doesn't create a cycle. We don't want cycles because trees cannot have cycles, and thinking in terms of the road network, having a cycle would mean we have a redundant edge.

In the graph above, we would start by adding edges \(ce\) and \(dg\) of weight 1. We then add edges \(bd\), \(ae\), and \(fh\), each of weight 2. Then we add edge \(eg\) of weight 4. We don't add edge \(ab\) of weight 4 because that would create a cycle. Alternatively, we could have added \(ab\) and not \(eg\). Either way would work. There are often several spanning trees of the same total weight. Finally, we add edges \(ci\) and \(hi\) to get a minimum spanning tree of total weight 22. No other spanning tree has a smaller weight.

To implement this algorithm, we have two things to do. First, we have to order the edges by weight. One way to do this would have been to have stored the edges in a data structure like a TreeSet or a BST. However, this technique of using an anonymous Comparator class is useful when you need to sort something in an unusual way. See Section 9.12 for more on this topic.

The other thing we need to do to implement the algorithm is to make sure that adding an edge does not create a cycle. This can be a little tricky. One clever way to do this is as follows: Create a list of sets corresponding to the endpoints of the edge. For example, in the graph shown earlier, the list of sets will change as follows:

- \([a], \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}\)
- \([a], \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{h\}, \{i\}\)
- \([a], \{b, d, g\}, \{c, e\}, \{f\}, \{h\}, \{i\}\)
- \([a, c, e], \{b, d, g\}, \{f\}, \{h\}, \{i\}\)
- \([a, c, e], \{b, d, g\}, \{f, h\}, \{i\}\)
- \([a, b, c, d, e, g], \{f, h\}, \{i\}\)
- \([a, b, c, d, e, g], \{f, h\}, \{i\}\)
- \([a, b, c, d, e, f, g, h, i]\]

The idea is that the sets stand for the connected components of the soon-to-be tree. Initially, it is
completely disconnected, with all the vertices disconnected from each other. We then add edge \(dg\), which connects \(d\) and \(g\), and we now have the set \(\{d, g\}\) in the list. Two steps later, we add the edge \(bd\), which connects \(b\) to \(d\), and therefore to \(g\), and so we end up with the set \(\{b, d, g\}\) in the list. The process continues until there is only one set in the list, meaning that all the vertices have been connected. To check to see if adding an edge creates a cycle, we just have to check to see if the start and endpoints are in the same set. If they are, then they are already connected and so we don’t want to add that edge. If they are not in the same set, then we merge the sets corresponding to the two endpoints of the edge to indicate that now everything connected to the one endpoint and everything connected to the other are now all connected together. The code for the algorithm is shown below.

```java
public List<Edge<T>> kruskal()
{
    List<Edge<T>> returnList = new ArrayList<Edge<T>>();
    List<Set<T>> sets = new ArrayList<Set<T>>(edgeMap.keySet());
    // sort edges by weight
    Collections.sort(edges,
        new Comparator<Edge<T>>() {
            public int compare(Edge<T> e1, Edge<T> e2) {
                Integer w1 = edgeMap.get(e1), w2 = edgeMap.get(e2);
                return w1.compareTo(w2);});
    }

    // create the list of sets, one set containing each vertex
    for (T v:neighbors.keySet())
        sets.add(new HashSet<T>(Collections.singleton(v)));
    while (!edges.isEmpty() && sets.size()!=1)
    {
        Edge<T> edge = edges.remove(0);
        // find the sets that contain the two endpoints
        Set<T> set1 = null, set2 = null;
        for (Set<T> set:sets)
        {
            if (set.contains(edge.u))
                set1 = set;
            if (set.contains(edge.v))
                set2 = set;
        }
        if (!set1.equals(set2))
        {
            set1.addAll(set2);
            sets.remove(set2);
            returnList.add(edge);
        }
    }
    if (edges.isEmpty()) // graph is disconnected, no spanning tree possible
        return new ArrayList<Edge<T>>();
    return returnList;
}
```

Kruskal’s algorithm is called a **greedy algorithm** because at each step you choose the minimum edge available. In general, greedy algorithms work such that at each step you always take the least-cost or most-cost move.

A second algorithm for finding a minimum spanning tree is called Prim’s algorithm. The main difference
between it and Kruskal's algorithm is that Prim's maintains a tree at each step. Start the algorithm by picking any vertex. Then add to the tree the edge from that vertex that has the least weight. Next, we pick the edge leading from either of the two vertices currently in the tree that has minimum weight and add it to the tree. We keep doing this, looking at all the edges leading from vertices currently in the tree to vertices not in the tree and add the one with minimum weight.

8.8 Dijkstra's shortest path algorithm

One other useful and interesting thing people do with weighted graphs is find a minimum weight path between two vertices. Thinking of the vertices as cities and the edges as the cost of traveling between cities, we would be interested in the cheapest route between two cities. We could also think of the vertices as states of a game and the edges as possible moves, with weights being the cost of making the move. We would be interested in the cheapest way to get from one state to another. Finally, if all the weights are equal, then finding a minimum weight path between two vertices amounts to finding a shortest path between them.

The most well-known algorithm for finding a minimum weight path is Dijkstra's algorithm. We will go through an example of Dijkstra's algorithm on the graph below, looking for the shortest path from $a$ to $g$.

We start the algorithm at the starting vertex, $a$. We look at all the edges from that vertex and mark their endpoints with the cost to get to them from $a$ and also with the start vertex indicating that we got to those vertices from $a$.

Next, we pick the vertex that was marked with the smallest cost from $a$ and examine all the edges out from that vertex. In this example it is vertex $b$. When we follow an edge, if the other endpoint is unmarked, we mark the other endpoint with the total cost to get to it from the start vertex $a$ (so the cost from $a$ to $b$ and from $b$ to the vertex) and we also mark it with the letter $b$ to indicate the path came from $b$. If the other endpoint is already marked, we only mark it if the total cost of getting to it from $b$ is less than the current cost it is marked at.
We continue this same process. We look at all the currently marked vertices and pick the one that is marked with the least total cost to get to it from \( a \), breaking ties arbitrarily. From the picked vertex, we explore all the edges out from it, marking each opposite endpoint as before.

We continue the process until the vertex with the least total cost is the goal vertex. This may take until the last step, but it may happen sooner.

One we get there, we can trace back using the vertices each vertex was marked by. Here, the goal vertex \( g \) was marked with \( f \), and \( f \) was marked with \( d \), which in turn was marked by \( b \), which was itself marked by \( a \). So our minimum weight path goes \( a \rightarrow b \rightarrow d \rightarrow f \rightarrow g \), for a total cost of 8.

To implement the algorithm we will use two maps and a priority queue. The maps totalCosts and parents keep track of two things we mark the vertices with. The priority queue is used to decide which vertex to use at each step.

```
Map<T, Integer> totalCost = new HashMap<T, Integer>();
Map<T, T> parents = new HashMap<T, T>();
```
PriorityQueue<VCPair> Q = new PriorityQueue<VCPair>();
List<T> searched = new ArrayList<T>();

The way we choose which vertex is based on its current total cost. So that is how we prioritize vertices that go into the priority queue. To do this, we create a separate class VCPair which has a field for the vertex and a field for its total cost. It implements Comparable and the comparisons are done by the total cost.

private class VCPair implements Comparable<VCPair>{
    public T vertex;
    public int totalCost;

    public VCPair(T vertex, int totalCost){
        this.vertex = vertex;
        this.totalCost = totalCost;
    }

    public int compareTo(VCPair other){
        Integer d1=totalCost, d2=other.totalCost;
        return d1.compareTo(d2);
    }
}

The way the algorithm runs is we loop as long as the priority queue has something in it. If it ever runs out, then there is no path at all between the vertices and we just return an empty list. At each step we dequeue the next vertex/cost pair from the queue into a variable called current. We use a priority queue because it allows us to efficiently store things in the right order.

The first thing we do with current is check to see if we've reached the end. If so, we trace back through the parents to return the path. This code is exactly the same as the BFS code that does the same thing.

    if (current.vertex.equals(end))
    {
        List<T> returnList = new ArrayList<T>();
        T u = end;
        while (u!=null)
        {
            returnList.add(u);
            u = parents.get(u);
        }
        Collections.reverse(returnList);
        return returnList;
    }

For the main part of the algorithm, we explore all the edges from our current vertex. We compute the new cost to each vertex as the cost to get to it, plus the cost of the edge. If the other endpoint of the edge has not yet been found or if the new cost is cheaper than that endpoints old cost, then we mark the vertex with its new parent and total cost. We then add this new information to the priority queue.

    for (T n:neighbors.get(current.vertex))
    {
        int newCost = current.totalCost + this.getWeight(current.vertex, n);
        }
if (!totalCosts.containsKey(n) || newCost < totalCosts.get(n))
{
    totalCosts.put(n, newCost);
    parents.put(n, current.vertex);
    Q.add(new VCPair(n, newCost));
}

There are a few things that are a little inefficient here. For one, when we add the new info to the priority queue, the old info for that vertex is still there. We may end up dequeuing a vertex cost/pair for a vertex more than once. This won’t affect the answer, however, as the first time that vertex’s vertex/cost pair is dequeued will be the one where the cost is as small as possible. However, we could spend some time pointlessly searching. One way to fix this would be to introduce a list of vertices that have already been searched. We choose not to do that here as checking through this list every time would be more code and would have its own time cost.

Here is the complete algorithm:

```java
public List<T> shortestPath(T start, T end)
{
    Map<T, Integer> totalCosts = new HashMap<T, Integer>();
    Map<T, T> parents = new HashMap<T, T>();
    PriorityQueue<VCPair> Q = new PriorityQueue<VCPair>();
    parents.put(start, null);
    totalCosts.put(start, 0);
    Q.add(new VCPair(start, 0));

    while (!Q.isEmpty())
    {
        VCPair current = Q.remove();
        if (current.vertex.equals(end))
        {
            List<T> returnList = new ArrayList<T>();
            T u = end;
            while (u!=null)
            {
                returnList.add(u);
                u = parents.get(u);
            }
            Collections.reverse(returnList);
            return returnList;
        }
        for (T n:neighbors.get(current.vertex))
        {
            int d = current.totalCost + this.getWeight(current.vertex, n);
            if (!totalCosts.containsKey(n) || d < totalCosts.get(n))
            {
                totalCosts.put(n, d);
                parents.put(n, current.vertex);
                Q.add(new VCPair(n, d));
            }
        }
    }
    return new ArrayList<T>();
}
```
8.9 Exercises

1. Here is a puzzle that is at least 1200 years old: A traveler has to get a wolf, a goat, and a cabbage across a river. The problem is that the wolf can’t be left alone with the goat, the goat can’t be left alone with the cabbage, and the boat can only hold the traveler and a single animal/cabbage at once. How can the traveler get all three items safely across the river?

Solve this problem by representing it with a graph and using a breadth-first search to find a solution.

2. The Internet Movie Database (IMDB) provides text files containing info on which actors and actresses were in which movies. One such file consists of thousands of lines, one line per movie. Each line starts with the title of the movie and is followed by a list of the people that acted in the movie. Each entry (including the title) is separated from the other entries by a slash, /. Read through the text file and put the information into a graph (you will have to decide how to structure the graph). Then prompt the user for the names of two actors and output the shortest path of the form (actor → movie → actor → movie → ... → actor) that runs between the two actors. For instance, here is the shortest path between Meryl Streep and Sly Stallone:

   [Streep, Meryl, Adaptation. (2002), Beaver, Jim (I), Nighthawks (1981), Stallone, Sylvester]

3. Add a method `addAll` to the graph class that takes a list of vertices as a parameter and adds each vertex to the graph.

4. Modify the depth-first search and breadth-first search methods so that instead of a single goal vertex as a parameter, they receive a list of goal vertices as a parameter. The search is successful if any of the vertices in the list are found.

5. Write a recursive version of the depth first search algorithm.

6. In Section 8.5, we created a class `Pair` that represents an ordered pair \((x, y)\). Generalize this idea to a class `Tuple` that allows us represent a tuple of arbitrary length, like \((x, y, z)\), or \((x, y, z, w, v)\). The `Tuple` class should have the same methods as the `Pair` class.

7. In this chapter we created a digraph class and a weighted graph class. Combine the ideas used in those classes to created a weighted digraph class.

8. Add the following methods to the graph class.

   (a) `distance(u, v)` — returns the distance between two vertices. The distance is the length of the shortest path between the vertices.

   (b) `findConnectedComponent(v)` — returns a list of all the vertices that can be reached from vertex \(v\)

   (c) `isConnected()` — returns `true` if the graph is connected and `false` otherwise
(d) `isCutVertex(v)` — returns `true` if `v` is a cut vertex, a vertex whose deletion disconnects the graph.

(e) `isHamiltonianCycle(a)` — returns `true` if the list of vertices `a` represent a Hamiltonian cycle. A Hamiltonian cycle is a cycle that includes every vertex.

(f) `isValidColoring(m)` — returns `true` if the map `m` represents a valid coloring of the graph and `false` otherwise. A coloring of a graph is an assignment of colors to the vertices of the graph such that adjacent vertices are not given the same color. The argument `m` is a map whose keys are the vertices of the graph and whose values are the colors (integers) assigned to the vertices. Usually positive integers are used for the color names.

(g) `findValidColoring()` — returns a map like in the previous problem that represents a valid coloring of the graph. Create the map using the following greedy algorithm: Loop through the vertices in any order, assign each vertex the smallest color (starting with the number 1) that has not already been assign to one of its neighbors.

(h) `isEulerian()` — returns `true` if the graph is Eulerian and false otherwise. A graph is Eulerian if one can traverse the graph using every edge exactly once and end up back at the starting vertex. A nice theorem says that a graph is Eulerian if and only if every vertex has even degree.

(i) `inducedSubgraph(list)` — given a list of vertices, returns a new graph object which consists of the vertices in the list and all the edges in the original graph between those vertices

(j) `addAll(list)` — adds all the vertices in `list` to the graph

(k) `addAll(v1, v2, ..., vn)` — adds the variable number of vertices, `v1, v2, ..., vn` to the graph. You may need to look up Java’s syntax for functions with a variable number of arguments.

9. For the tree below, we are starting at vertex `a` and searching for vertex `j`.

   ![Tree Diagram]

   (a) Indicate the order in which a depth-first search visits the nodes, assuming that it visits the descendants of a node from left to right.

   (b) Indicate the order in which a breadth-first search visits the nodes, assuming that it visits the descendants of a node from left to right.

10. For this problem you will do a few steps of Dijkstra’s Shortest Path Algorithm on the graph below. We are looking for the shortest path from `a` to `e`. The first step is done for you. Please fill in the next two steps. At each step indicate the contents of the list of finished vertices and the map of parents.
11. For the graphs below, find a minimum spanning tree using Kruskal's algorithm. Indicate the order in which the edges are added.

12. The table below gives the cost of building a road between various cities. An entry of $\infty$ indicates that the road cannot be built. The goal is to find the minimum cost of a network of roads such that it is possible to get from any city to any other. This problem can be solved with Kruskal's algorithm. Set up the graph in Java and use the method we wrote in Section 8.7 to find the solution.
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>∞</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>11</td>
<td>9</td>
<td>∞</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>e</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
Chapter 9

Sorting

This chapter is about a variety of different algorithms for sorting arrays and lists. Most higher level languages have sorting built in to the language (for instance, Java has Collections.sort and Arrays.sort), so it not as useful now to know how to sort things as it once was, when programmers often had to implement their own sorts. However, it is still useful to know how some of the different sorts work. It is also good practice to implement the algorithms. Knowledge of these things is considered foundational computer science knowledge, and moreover, some of the ideas behind the algorithms are useful in other situations.

A few notes:

- We will implement all the searches on integer arrays instead of on generic lists. The reason for this is that array syntax is simpler and more readable than list syntax. In practice, though, it is usually better to use lists in place of arrays. It is not hard to translate the array algorithms to list algorithms. This is covered in Section 9.11.

- Our sorts will directly modify caller’s array. The other option would be to leave the array untouched and return a new sorted array.

- One consideration to keep in mind when going through the different algorithms is whether or not they are stable. This concept comes in when sorting objects based on a field. For example, suppose we have a list of person objects with fields for name and age, and we are sorting based on age. Suppose entries in the list are already alphabetized. A stable sort will maintain the alphabetical order while sorting by age, whereas an unstable on may not. In some cases, our algorithms will not be stable but may be able to be modified to be stable.

- Another important consideration is how the algorithms perform on data that is already sorted or nearly so. Though it might seem silly at first, sorts are often used on already sorted data. For example, consider a program where the user has to enter data and the program needs the data to be sorted and sorts it. If the user is typing in the data off of a sheet that is already in sorted order, then the program would be performing a sort on already sorted data. Another important use case
is data entered in reverse order. Some algorithms do well with data in these use cases and some do quite poorly. Data in reversed order is actually the worst case scenario for some of algorithms.

### 9.1 Selection Sort

The Selection Sort is one of this simplest sorts to implement, but it is also slow. The way it works is shown in the figure below.

```
1 CBDEA  5 ACDEB  8 ABDEC
2 BCDEA  6 ACDEB  9 ABDEC
3 BCDEA  7 ACDEB 10 ABCED
4 BCDEA
```

We start by comparing the first element in the array to the second, third, fourth, etc. If the elements being compared are out of order, we swap them. By the time we are done with this first step, we are guaranteed that the first position of the array will contain the correct element.

We then move on to the second element in the array and compare it to the third, fourth, fifth, etc. elements, swapping as necessary. At the end of this step, we are guaranteed that the second position in the array will contain the correct element.

From here, we continue this way, comparing the third element to everything after, then comparing the fourth element to everything after, etc., until we reach the end of the array.

To implement this, we use nested for loops. The outer loop runs with \( i \) going from 1 to one element short of the end of the array, while the inner loop has \( j \) running from \( i+1 \) to the end of the array. The inner loop loops over all the array elements to the right of the position we are currently working on. In it we check to see if elements are out of order and swap if needed.

```java
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        for (int j = i + 1; j < a.length; j++) {
            if (a[i] > a[j]) {
                int hold = a[i];
                a[i] = a[j];
                a[j] = hold;
            }
        }
    }
}
```

The total number of comparisons made is \( n(n - 1)/2 \). See Section 1.4 for the calculation. And it always makes this number of comparisons, no matter if the array is already sorted or completely jumbled up.
So it is very consistent, but as an $O(n^2)$ algorithm, it is very slow for large arrays.

The Selection Sort is fine for sorting small arrays. If you find you need to quickly code a sort and you don’t care about its running time, then the Selection Sort fits the bill. It is just two for loops, an if statement, and a swap.

### 9.2 Bubble Sort

Bubble Sort is a relative of Selection Sort. Like Selection Sort, Bubble Sort uses two for loops, an if statement, and a swap. The difference is in what is compared. The figure below shows the order in which elements are compared.

We start at the end of the array and compare the last and second-to-last elements, swapping if necessary. We then compare the second-to-last element with the third-to-last. Then we compare that one with the one before it. We keep going until we reach the front of the array. At this point, the first position in the array will be correct. The correct element “bubbles up” to the front of the array.

We then start at the end of the array and repeat the process, but this time instead of going all the way to the front, we stop at the second element (because we know the first element is already correct). After this stage, the second element is correct.

We then continue the process, where at each step, we start at the end of the array and compare adjacent elements and continue doing that until we get to the stopping point. At each step, the stopping point moves one index forward until we reach the end of the array.

We use nested for loops to implement the algorithm. The outer loop keeps track of the stopping point. It starts at the front of the array and works its way to the end. The inner loop always starts at the end of the array and ends at the stopping point. Inside the loop we compare adjacent elements and swap if necessary.

```java
public static void bubbleSort(int[] a)
{
    for (int i=0; i<a.length-1; i++)
    {
        for(int j=a.length-1; j>i; j--)
        {
            if (a[j-1]>a[j])
            {
                int hold = a[j-1];
                a[j-1] = a[j];
                a[j] = hold;
            }  
        }
    }
}
```
Just like Selection Sort, Bubble Sort will make $n(n-1)/2$ comparisons, regardless of whether the array is already sorted or completely jumbled. However, we can actually improve on this. If on one of the passes through the array (going from the end to the current stopping point), there are no swaps made, then the array must be sorted and we can stop. Exercise 4 is about implementing this. This improvement, however, is still not enough to make the Bubble Sort useful for sorting large arrays. In fact, its performance can be worse than the ordinary Bubble Sort for large, well-mixed arrays.

Bubble Sort is useful exactly where Selection Sort is useful—as a quick and simple algorithm to sort small arrays.

### 9.3 Insertion Sort

Say we have a stack of papers to sort in alphabetical order. Start by putting the first two papers in order. Then put the third paper where it fits in order with the other two. Then put the fourth paper where it fits in order with the first three. Keep doing this. This is essentially the Insertion Sort. At each step the papers already looked at are in sorted order.

With an array, we loop over the indices of the array running from 1 to the end of the array. For each index, we take the element at that index and loop back through the array towards the front, looking for where the element belongs among the earlier elements. These elements are in sorted order, so to find the location we run the loop until we find an element that is smaller than the one we or looking at or fall off the front of the array. The figure below shows it in action.

Here is the code:

```java
public static void insertionSort(int[] a)
{
    for (int i=1; i<a.length; i++)
    {
        int hold = a[i];
        int j = i;
        while (j>=1 && a[j-1]>hold)
        {
            a[j] = a[j-1];
            j--;
        }
        a[j] = hold;
    }
}
```

Here is the code:
With the outer for loop we look at a new element at each time. The inner while loop looks for the right place to put that element. Notice that, unlike Bubble Sort and Selection Sort, there is no need to swap elements. Instead, we can use \( a[j] = a[j-1] \) to move things down the line and then \( a[j] = \text{hold} \) to finish.

'Insertion Sort is faster than both Selection Sort and Bubble Sort, but it is still an \( O(n^2) \) algorithm. Insertion Sort will usually make less comparisons than the other two, but we still have nested loops and on average, the number of comparisons needed is still quadratic and hence unsuitable for sorting large arrays. However, for small arrays it is very fast, faster in fact than just about anything else including the \( O(n \log n) \) algorithms in the upcoming sections. The key is that those algorithms require a certain amount of setup work or overhead due to recursion that Insertion Sort doesn’t. Once they get going, however, they will usually outperform Insertion Sort.

If the list is already sorted, the condition on the inner while loop will always be false and Insertion Sort runs in \( O(n) \) time, which is the best any sorting algorithm can manage. On the other hand, if the list is in reverse order, the while loop condition is always true and we have \( n(n - 1)/2 \) swaps, just as bad as Selection Sort and Bubble Sort.

9.4 Shellsort

Shellsort builds off of Insertion Sort. Here is an example of it in action. Say we have have the following string to rearrange into alphabetical order:

\[
A \ Q \ D \ O \ M \ C \ R \ B \ J \ I \ H \ P \ L \ K \ F \ G \ E \ N
\]

We start by pulling out every fourth character starting with the first, and using Insertion Sort to sort just those characters. We then pull out every fourth character starting with the second character, and use Insertion Sort to sort those characters. We do the same for every fourth character starting with the third and then every fourth starting with the fourth. See the figure below:

\[
A \ Q \ D \ O \ M \ C \ R \ B \ J \ I \ H \ P \ L \ K \ F \ G \ E \ N
\]

\[
A \ Q \ D \ O \ E \ C \ R \ B \ J \ I \ H \ P \ L \ K \ F \ G \ M \ N
\]

\[
A \ C \ D \ O \ E \ I \ R \ B \ J \ K \ H \ P \ L \ N \ F \ G \ M \ Q
\]

\[
A \ C \ D \ O \ E \ I \ F \ B \ J \ K \ H \ P \ L \ N \ R \ G \ M \ Q
\]

We then do something similar except we break up the array at a finer scale, first breaking it up at every second character starting with the first, sorting those characters, and then breaking it up at every second
character starting with the second and sorting, as shown below:

\begin{verbatim}
A C D B E I F G J K H O L N R P M Q
A C D B E I F G H K J O I N M P R Q
\end{verbatim}

At this point the array is nearly sorted:

\begin{verbatim}
A B D C E G F I H K J N L O M P R Q
\end{verbatim}

We run a final insertion sort on the entire array to sort it. This step is pretty quick because only a few swaps need to be made before the array is sorted. And it may seem like we did so many individual sorts that it would have been more efficient to just have run a single insertion sort in the first place, but that is not the case. The total number of comparisons and swaps needed will often be quite a bit less for Shellsort than for Insertion Sort.

In the example above, we used gaps of length 4, 2, and 1. The sequence (4, 2, 1) is called a gap-sequence. There are a variety of different gap sequences one could use, the only requirement being that the last gap is 1. The standard implementation of Shellsort is shown below. It surprisingly short and very efficient as it essentially works on all the subarrays for a given gap at once. In the example above, with a gap of 4, we did four separate steps, but the code below would handle them all together.

```java
public static void shellsort(int[] a) {
    int[] gaps = {1391376, 463792, 198768, 86961, 33936, 13776, 4592, 1968, 861, 336, 112, 48, 21, 7, 3, 1};
    for (int gap:gaps) {
        for (int i=gap; i<a.length; i+=1) {
            int hold = a[i];
            int j = i;
            while (j>=gap && a[j-gap]>hold) {
                a[j] = a[j-gap];
                j-=gap;
            }
            a[j] = hold;
        }
    }
}
```

Different gap sequences produce different running times. Some produce an $O(n^2)$ running time, while others can get somewhat closer to $O(n)$, with running times like $O(n^{5/4})$ or $O(n^{3/2})$. Determining the running time for a given gap sequence can be tricky. In fact the running times for some popular gap sequences are unknown. Another open problem would be to find better gap sequences than what are currently known.

Shellsort doesn’t get a lot of attention, but it is a good sort, competitive even with the $O(n \log n)$ algorithms that have better asymptotics.
9.5 Heapsort

Heapsort uses a heap to sort an array. Recall that a heap is a data structure for storing data in order. Once we have a heap data structure constructed, using it to sort an array is very easy—we just run through the array, putting the elements one-by-one onto the heap and then one-by-one we remove the top (minimum) element from the heap. Here is the code using the heap class we created in Chapter 6.

```java
public static void heapsort(int[] a)
{
    Heap<Integer> heap = new Heap<Integer>(Heap.MINHEAP);
    for (int i=0; i<a.length; i++)
        heap.add(a[i]);
    for (int i=0; i<a.length; i++)
        a[i] = heap.pop();
}
```

The PriorityQueue class in Java’s Collections Framework is implemented using a heap. So we can use PriorityQueue to implement Heapsort. This will give better performance as PriorityQueue is more efficient than our simple heap implementation.

```java
public static void heapsort(int[] a)
{
    PriorityQueue<Integer> Q = new PriorityQueue<Integer>();
    for (int i=0; i<a.length; i++)
        Q.add(a[i]);
    for (int i=0; i<a.length; i++)
        a[i] = Q.remove();
}
```

Heapsort has an $O(n \log n)$ running time. When we insert the elements into the heap we have to loop through the array, so that’s where the $n$ comes from. Adding each element to the heap is an $O(\log n)$ operation, so overall adding the elements of the array into the heap takes $O(n \log n)$ time. Removing them also takes $O(n \log n)$ time.

Heapsort is one of the faster sorts. It runs in $O(n \log n)$ time regardless of whether the data is already sorted, in reverse order, or whatever.

9.6 Treesort

Along the same lines as Heapsort, we have Treesort, which uses a binary search tree instead of a heap to sort an array. It works in a similar way. We first add the elements of an array to the BST one at a time. The BST stores them in sorted order. We then traverse the tree using an in-order traversal (that’s where you visit a node’s left subtree, then the node itself, then the node’s right subtree). The BST we implemented in Chapter 6 had a method called inOrderPrint that did this, but it returned a string. We need something that returns an array or a list. Here is a pair of BST methods to do that:
public List<Integer> inOrderList()
{
    List<Integer> list = new ArrayList<Integer>();
    inOrderList(root, list);
    return list;
}

private void inOrderList(Node n, List<Integer> list)
{
    if (n == null)
        return;

    inOrderList(n.left, list);
    list.add(n.data);
    inOrderList(n.right, list);
}

The Treesort method is then very simple:

    public static void treesort(int[] a)
    {
        BinarySearchTree bst = new BinarySearchTree();
        for (int i=0; i<a.length; i++)
            bst.add(a[i]);

        List<Integer> list = bst.inOrderList();
        for (int i=0; i<a.length; i++)
            a[i] = list.get(i);
    }

As long as the BST is relatively well balanced, inserting and removing elements from the BST is a log n operation. So the running time of the algorithm in that case will be O(n log n). Our BST implementation is simple and if the array is already sorted or almost sorted, the BST will be very unbalanced, resulting in a worst case of O(n^2). Using a more sophisticated BST implementation like AVL trees or red-black trees would bring the worst case down to O(n log n).

Treesort is a decent search for large arrays.

9.7 Merge Sort

Merge Sort works as follows: We break the array into two halves, sort them, and then merge them together. The halves themselves are sorted with Merge Sort, making this a recursive algorithm. For instance, say we want to sort the string CEIGJBFDHA. We break it up into two halves CEIGJ and BFDHA. We then sort those halves to get CEGIJ and ABDFH and merge the sorted halves back together to get ABCDEFGHILJ. See the figure below:
We will break up the work into sorting and merging. Here is the sorting code:

```java
public static void mergeSort(int[] a)
{
    if (a.length<=1)
        return;
    int[] left = Arrays.copyOfRange(a, 0, a.length/2);
    int[] right = Arrays.copyOfRange(a, a.length/2, a.length);
    mergeSort(left);
    mergeSort(right);
    int[] m = merge(left, right);
    for (int i=0; i<a.length; i++)
        a[i] = m[i];
}
```

The base case of the recursion is a list of size 0 or 1, which doesn't require any sorting. We use the java.util.Arrays method copyOfRange to get the left and right halves of the array. We then call Merge Sort on each, merge the two sorted halves, and copy the result back into a.

Here is the merging code. The way it works is we maintain counters to keep track of where we are in the left and right arrays. We loop through, looking at the elements at the current location in each array, adding the smaller of the two to the merged array and then updating the appropriate counter. We stop when we reach the end of either array. At this point, there still may be some elements left over in the other array, so we add all of them to the merged array.

```java
private static int[] merge(int[] left, int[] right)
{
    int[] mergedArray = new int[left.length + right.length];
    int x=0, y=0, z=0;
    while (x<left.length && y<right.length)
    {
        if (left[x]<right[y])
            mergedArray[z++] = left[x++];
        else
            mergedArray[z++] = right[y++];
    }
    if (x<left.length)
    {
        for (int i=x; i<left.length; i++)
            mergedArray[z++] = left[i];
    }
    if (y<right.length)
    {
        for (int i=y; i<right.length; i++)
            mergedArray[z++] = right[i];
    }
    return mergedArray;
}
```
else if (y<right.length)
{
    for (int i=y; i<right.length; i++)
        mergedArray[z++] = right[i];
}

return mergedArray;

One thing to note is that to keep the code short, we use the ++ operator. When used like this, the incrementing happens after the assignment. So, for instance, the code on the left is equivalent to the code on the right.

if (left[x]<right[y])
    mergedArray[z++] = left[x++];

if (left[x]<right[y])
{
    mergedArray[z] = left[x];
    z++;
    x++;

Overall, Merge Sort has an \(O(n \log n)\) running time. Intuitively, the logarithm comes from the fact that we are cutting things in half at each step. The \(n\) comes from the fact that the merging algorithm is \(O(n)\).

Because of the overhead involved in recursion, Merge Sort can be outperformed by some of the \(O(n^2)\) sorts, like Insertion Sort, for small arrays. This suggests a way to speed Merge Sort up a little. Instead of our current base case taking effect at length 0 or 1, what we can do is if the length of the array smaller than some size, we call Insertion Sort, like below:

if (a.length<=100)
    insertionSort(a);

Even with this improvement, our version of the sort is a little slow. There are several places that it can be improved, but one in particular stands out—we waste a lot of time creating new arrays. It’s hard to avoid having to use at least some extra memory, but we can get away with just creating a single array \(b\) which has the same size as \(a\). The key is that instead of creating the left and right arrays, we will work completely within \(a\), using indices called left, mid, right to keep track of where in the array we are currently sorting. These become parameters to the merge function itself. The array auxiliary array \(b\) is used when merging. The faster code is below:

```
public static void mergeSort(int[] a)
{
    int[] b = new int[a.length];
    mergeSortHelper(a, b, 0, a.length);
}

private static void mergeSortHelper(int[] a, int[] b, int left, int right)
{
    // base case
    if (right-left<=1)
        return;

    // recursive part
    int mid = (left+right)/2;
    mergeSortHelper(a, b, left, mid);
    mergeSortHelper(a, b, mid, right);
```

9.8. QUICKSORT

Merge Sort is one of the faster sorting algorithms. The sort method built in to Python and Java 7 is a variant of Merge Sort called Timsort.

9.8 Quicksort

Quicksort, like Merge Sort, is a sort that works by breaking the array into two subarrays, sorting them and then recombining. One subarray, smaller, consists of all the elements less than the first element in the array and the other subarray, larger, consists of all the elements greater than or equal to it. For instance, if we are sorting CEIGJBFDHA, the subarray smaller is BA (everything less than the first element, C) and the subarray larger is EIGJFDH (everything greater than C). We do this by scanning through the array and putting all the elements that are less than the first element (called the pivot) into smaller and putting the rest into larger.

We then sort the two subarrays using Quicksort again, making this a recursive algorithm. Finally, we combine the subarrays. Because we know that all the things in smaller are less than the pivot and all the things in larger are greater than or equal to the pivot, we build our sorted array by adding the elements of smaller (which are now sorted), then adding the pivot, and then adding the elements of larger. See the figure below:
Notice that the way Quicksort breaks up the array is more complicated way than the way Merge Sort does, but then Quicksort has an easier time putting things back together than Merge Sort does.

Here is the Quicksort code:

```java
public static void quicksort(int[] a) {
    if (a.length <= 1)
        return;

    int[] smaller = new int[a.length - 1];
    int[] larger = new int[a.length - 1];
    int pivot = a[0];

    // build the subarrays
    int d = 0, e = 0;
    for (int i = 1; i < a.length; i++)
        if (a[i] < pivot)
            smaller[d++] = a[i];
        else
            larger[e++] = a[i];

    smaller = Arrays.copyOf(smaller, d);
    larger = Arrays.copyOf(larger, e);

    // recursive part
    quicksort(smaller);
    quicksort(larger);

    // put back together
    int c = 0;
    for (int x : smaller)
        a[c++] = x;
    a[c++] = pivot;
    for (int x : larger)
        a[c++] = x;
}
```

There's one trick here with building the subarrays. Initially, we don't know how much space to allocate for them. So we allocate the maximum amount we would need, then build up the arrays, and then use the `java.util.Arrays` method `copyOf` the data in `smaller` and `larger` into appropriately sized arrays.

Quicksort generally has an $O(n \log n)$ running time, but there are some special cases where it degenerates to $O(n^2)$. One of these would be if the array was already sorted. In that case the subarray `smaller` will be empty, while the subarray `larger` will have size $n - 1$. When we call Quicksort on `larger`, we will
again have the same problem, so we will end up with $n$ total recursive calls. Since real-life arrays are often sorted or nearly so, this is an important problem. One way around this problem is to use a different pivot, such as one in the middle of the array.

But there is a bigger problem with our implementation and that is that, contrary to its name, our algorithm is slow. Creating the subarrays is very time-consuming, particularly because we create two new arrays, recopy them, and then put them back together. We can get a considerable speedup by doing the partition \textit{in-place}. That is, we will not create any new arrays, but instead we will move things around in the main array. This is not unlike how we sped up our Merge Sort implementation.

There are a few different ways to do this. The approach we will take is to have two indices, $i$ and $j$, with $i$ starting at the left end of the array and $j$ starting at the right. Both indices will move towards the middle of the array. We first advance $i$ until we meet an element that is greater than or equal to the pivot. We then advance $j$ until we meet an element that is less than or equal to the pivot. When this happens, we swap the elements at each index and repeat the process until the indices meet up.

Here is an example. We will assume the pivot is the middle element of the list, in this case the letter $P$. The index $i$ starts out at 0 and advances forward until it meets an element larger than the pivot, which happens at index 3, a $Z$. The index $j$ starts out at the end of the array and moves left until it meets an element smaller than the pivot, which happens right away. We then swap the elements at these indices. The indices then start moving towards each other again until $i$ meets something greater than the pivot and $j$ meets something less than the pivot. We swap the elements at location and continue the process. The process stops when $i$ and $j$ cross paths.

Once we partition the array, we then make the recursive calls to Quicksort, using the positions of the indices $i$ and $j$ to indicate where the two subarrays are located in the main array. Also, because we are using the same array for everything, we need to have two additional parameters to the Quicksort function, \texttt{left} and \texttt{right}, which indicate which part of the array we are currently working on. Here is the code:

```java
public static void quicksort(int[] a)
```

CHAPTER 9. SORTING

public static void quicksort_helper(int[] a, int left, int right)
{
    if (left>=right)
        return;
    int i=left, j=right, pivot=a[(left+right)/2];
    while (i<=j)
    {
        while (a[i]<pivot) i++;
        while (a[j]>pivot) j--;
        if (i>j)
            break;
        int hold = a[i];
        a[i] = a[j];
        a[j] = hold;
        i++;
        j--;
    }
    quicksort_helper(a, i, right);
    quicksort_helper(a, left, j);
}

This gives a big improvement in speed. The only real problem here is that the algorithm is susceptible to a stack overflow error when used to sort large arrays. One way to help this is (just like with Merge Sort) to use an Insertion Sort whenever one of the subarrays is small, like less than 100 elements. Another way is to manage a stack ourselves instead of using recursion.

Quicksort is one of the fastest sorting algorithms around. It is used in a number of programming language libraries.

9.9 Counting Sort

If we know something about the data in our array, then we can do better than the \( O(n \log n) \) algorithms we've seen in the last few sections. For instance, if we know that we know the array contains only values in a relatively small range, then we can use an \( O(n) \) sort known as Counting Sort.

Counting Sort works by keeping an array of counts of how many times each element occurs in the array. We scan through the array and each time we meet an element, we add 1 to its count. So for instance, suppose we know that our arrays will only contain integers between 0 and 9. If we have the array \[1,2,5,0,1,5,1,3,5\], the array of counts would be \[1,3,1,1,0,3,0,0,0\] because there is 1 zero, 3 ones, 1 two, 1 three, no fours, 3 fives, and no sixes, sevens, eights, or nines.

We can then use this array of counts to construct the sorted list \[0,1,1,1,2,3,5,5,5\] by repeating 0 one time, repeating 1 three times, repeating 2 one time, etc., starting from 0 and repeating each element according to its count. Here is the Counting Sort algorithm implemented for arrays with values between 0 and 99:
public static void countingSort(int[] a)
{
    int[] counts = new int[100];
    for (int x:a)
        counts[x]++;
    int c=0;
    for (int i=0; i<counts.length; i++)
        for (int j=0; j<counts[i]; j++)
            a[c++] = i;
}

This simple sort has an $O(n)$ running time. The only drawback is that it only works if the array consists of integers between 0 and 99. We can change the size of the counts array to something larger, like 10,000 or 100,000 to get it to work with arrays that contain larger numbers, but this approach becomes infeasible if the array could contain very large integers.

9.10 Comparison of sorting algorithms

First, in terms of asymptotics, the running time of a sort must always be at least $O(n)$ because you have to look at every element before you know if the array is sorted. We have seen that Counting Sort achieves the $O(n)$ optimum, but it is limited in where it can be used. On any sort that uses comparisons, it can be shown that the running time must be at least $O(n \log n)$. The $O(n \log n)$ sorts that we have seen are Heapsort, Treesort, Merge Sort, and Quicksort. Of these, Merge Sort and Quicksort are regarded as the fastest.

But there is more to the story. We won't focus too much on this, and a lot more info can be found elsewhere. One thing that matters is exactly how the algorithms are implemented on what sort of data they will be used. Another consideration is memory usage. Most of the $O(n \log n)$ algorithms have memory usage associated with either a data structure, recursive overhead, or using auxiliary arrays to hold intermediate results.

Also, we need to consider how the algorithms work with memory. For instance, it is important whether the algorithm reads from memory sequentially or randomly. While it is not important with arrays small enough to fit in RAM, it is important when sorting large files that require reads from a hard disc. Repositioning the read head on a hard drive is slow, whereas once it is positioned, reading the next elements in sequence is not as slow. So, since Merge Sort does sequential reads and Quicksort does random ones, Merge Sort may be better when a lot of reading from a hard disc is required. Some of these same considerations apply to how well the algorithm makes use of cache memory.

Another important consideration is whether the algorithm is stable or not. Of the ones we have looked at, Bubble Sort, Insertion Sort, Treesort, and Merge Sort are stable, while Selection Sort, Shellsort, Heapsort, and Quicksort are not, though some of them can be modified to be stable. Another consideration is whether the sort is done in-place, that is whether all the work is done on the array itself or whether auxiliary arrays are needed. And beyond this, there are still further considerations that we won't get into here. In summary, each of the algorithms has its good and bad points. There are even situations
where an $O(n^2)$ algorithm would be better than an $O(n \log n)$ algorithm. But of all the algorithms, Quicksort and Merge Sort are probably the most used.

Here is a table of the running times of the sorts:

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Worst</th>
<th>Already sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)^*$</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Shell</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Tree**</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quick</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n \log n)^{**}$</td>
</tr>
<tr>
<td>Count</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

* An easy modification makes this $O(n)$.
** The worst-case and already-sorted-case for treesort can be improved to $O(n \log n)$ using AVL or red-black trees, but using our implementation they are $O(n^2)$.
*** Depends on the pivot. Choosing the first element as the pivot will make this $O(n^2)$, but choosing the middle element will make it $O(n \log n)$.

Finally, to close out the section, we have a comparison of the sorting algorithms from this chapter. For each array size from 10 through 10,000,000, going up by a factor of 10 at each step, each algorithm was given the same 100 arrays consisting of random integers between 0 and 99999. This is mostly for fun and is not meant to be a scientific study. Here are the results. Times are given in milliseconds.

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select</td>
<td>0.0052</td>
<td>Shell</td>
<td>0.013</td>
<td>Quick</td>
<td>0.09</td>
<td>Count</td>
<td>0.6</td>
</tr>
<tr>
<td>Bubble</td>
<td>0.0056</td>
<td>Quick</td>
<td>0.013</td>
<td>Shell</td>
<td>0.11</td>
<td>Quick</td>
<td>1.1</td>
</tr>
<tr>
<td>Insert</td>
<td>0.0057</td>
<td>Insert</td>
<td>0.014</td>
<td>Merge</td>
<td>0.13</td>
<td>Shell</td>
<td>1.4</td>
</tr>
<tr>
<td>Quick</td>
<td>0.0071</td>
<td>Merge</td>
<td>0.018</td>
<td>Tree</td>
<td>0.25</td>
<td>Merge</td>
<td>1.6</td>
</tr>
<tr>
<td>Shell</td>
<td>0.0080</td>
<td>Select</td>
<td>0.030</td>
<td>Heap</td>
<td>0.36</td>
<td>Tree</td>
<td>3.1</td>
</tr>
<tr>
<td>Merge</td>
<td>0.0101</td>
<td>Bubble</td>
<td>0.031</td>
<td>Insert</td>
<td>0.49</td>
<td>Heap</td>
<td>4.8</td>
</tr>
<tr>
<td>Tree</td>
<td>0.0450</td>
<td>Heap</td>
<td>0.071</td>
<td>Count</td>
<td>0.52</td>
<td>Insert</td>
<td>49.0</td>
</tr>
<tr>
<td>Heap</td>
<td>0.0455</td>
<td>Tree</td>
<td>0.073</td>
<td>Bubble</td>
<td>2.59</td>
<td>Bubble</td>
<td>252.8</td>
</tr>
<tr>
<td>Count</td>
<td>0.4651</td>
<td>Count</td>
<td>0.494</td>
<td>Select</td>
<td>2.64</td>
<td>Select</td>
<td>255.6</td>
</tr>
</tbody>
</table>

* Only averaged over 10 trials instead of 100 to save time
** Not attempted because they would take too long
*** Treesort and Heapsort ran out of memory at $n = 10,000,000$

Here are the results sorted by algorithm:

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>0.0056</td>
<td>0.031</td>
<td>2.59</td>
<td>252.8</td>
<td>25343.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Count</td>
<td>0.4851</td>
<td>0.494</td>
<td>0.52</td>
<td>0.6</td>
<td>1.7</td>
<td>8</td>
<td>61</td>
</tr>
<tr>
<td>Heap</td>
<td>0.0455</td>
<td>0.071</td>
<td>0.36</td>
<td>4.8</td>
<td>83.8</td>
<td>2002</td>
<td>—</td>
</tr>
<tr>
<td>Insert</td>
<td>0.0057</td>
<td>0.014</td>
<td>0.49</td>
<td>49.0</td>
<td>5388.0</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Merge</td>
<td>0.0101</td>
<td>0.018</td>
<td>0.13</td>
<td>1.6</td>
<td>20.1</td>
<td>280</td>
<td>2587</td>
</tr>
<tr>
<td>Quick</td>
<td>0.0071</td>
<td>0.013</td>
<td>0.09</td>
<td>1.1</td>
<td>12.8</td>
<td>131</td>
<td>1302</td>
</tr>
<tr>
<td>Select</td>
<td>0.0052</td>
<td>0.030</td>
<td>2.64</td>
<td>255.6</td>
<td>26771.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Shell</td>
<td>0.0080</td>
<td>0.013</td>
<td>0.11</td>
<td>1.4</td>
<td>19.9</td>
<td>248</td>
<td>3040</td>
</tr>
<tr>
<td>Tree</td>
<td>0.0450</td>
<td>0.073</td>
<td>0.25</td>
<td>3.1</td>
<td>86.8</td>
<td>2027</td>
<td>—</td>
</tr>
</tbody>
</table>

The tests for large arrays were not run on Insertion Sort, Bubble Sort, or Selection Sort because they would take far too long. In fact, when looking at the times for these arrays, we can see the $O(n^2)$ growth.
Look for instance, at the Bubble Sort times. We see that as $n$ goes from 100 to 1000 (up by a factor of 10), the time goes from .031 to 2.59, an increase of roughly 100 times. From $n = 1000$ to 10000 we see a similar near hundredfold increase from 2.59 to 252.8. From $n = 10000$ to 100000, it again goes up by a factor of nearly 100, from 252.8 to 25343.1. This is quadratic growth — a tenfold increase in $n$ translates to a hundredfold ($100 = 10^2$) increase in running time. Were we to try running Bubble Sort with $n = 1,000,000$, we would expect a running time on the order of about 2500000 milliseconds (2500 seconds or about 40 minutes). At $n = 10,000,000$, Bubble Sort would take about 4000 minutes (roughly 3 days).

There are dozens of other sorting algorithms out there. There are a number of places on the web that have good information and interesting graphics and animations showing how various sorting algorithms work.

9.11 Sorting with generics and lists

As mentioned, we wrote the sorting methods to work just on integer arrays in order to keep the code simple, without too much syntax getting in the way. It’s not too much work to modify the methods to work other data types. We use lists instead of arrays and use generics to support multiple data types.

Here is the Selection Sort algorithm on an integer array from Section 9.1:

```java
public static void selectionSort(int[] a)
{
   for (int i=0; i<a.length-1; i++)
   {
      for (int j=i+1; j<a.length; j++)
      {
         if (a[i]>a[j])
         {
            int hold = a[i];
            a[i] = a[j];
            a[j] = hold;
         }
      }
   }
}
```

Here is its modification to run with a list of objects:

```java
public static <T extends Comparable<T>> void selectionSort(List<T> a)
{
   for (int i=0; i<a.size()-1; i++)
   {
      for (int j=i; j<a.size(); j++)
      {
         if (a.get(i).compareTo(a.get(j))>0)
         {
            T hold = a.get(i);
            a.set(i, a.get(j));
            a.set(j, hold);
         }
      }
   }
}
```
There is an array sorting method in `java.util.Arrays`. Here is an example of it:

```java
int[] a = {3,9,4,1,3,2};
Arrays.sort(a);
```

When working with lists, use the sorting method in the Collections Framework. Here is an example:

```java
List<Integer> list = new ArrayList<Integer>();
Collections.addAll(list, 3,9,4,1,3,2);
Collections.sort(list);
```

### Sorting by a special criterion

Sometimes, we have objects that we want to sort according to a certain rule. For example, it is often useful to sort strings by length rather than alphabetically. Java's sort methods allow us to specify a second argument, a `Comparator` that tells how the sorting is to be done.

Here is a comparator that compares strings by length.

```java
import java.util.Comparator;
public class StringLengthComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        Integer len1=s1.length(), len2=s2.length();
        return len1.compareTo(len2);
    }
}
```

We can use it as follows:

```java
List<String> list = new ArrayList<String>();
Collections.addAll(list, "this", "is", "a", "test");
Collections.sort(list, new StringLengthComparator());
System.out.println(list);
```

We can also use an anonymous Comparator class to avoid having to create a new class. Here is what that looks like:

```java
Collections.sort(M, new Comparator<String>() {
    public int compare(String s1, String s2) {
        Integer len1=s1.length(), len2=s2.length();
        return len1.compareTo(len2); }});
```
Here is another example of using a Comparator to sort. Say we have a class called Record that has three fields—firstName, lastName, and age—and we want to sort a list of records by age. Here is the Record class:

```java
private class Record {
    private int age;
    private String firstName, lastName;

    public Record(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String toString() {
        return "(" + firstName + " " + lastName + ", " + age + ")";
    }
}
```

To sort a list of records by age, we could use the following:

```java
Collections.sort(list, new Comparator<Record>() {
    public int compare(Record r1, Record r2) {
        Integer age1 = r1.age, age2 = r2.age;
        return age1.compareTo(age2);
    }
});
```

Suppose we want to sort first by last name and then by first name. We could use the following:

```java
Collections.sort(P, new Comparator<Record>() {
    public int compare(Record r1, Record r2) {
        String last1 = r1.lastName, last2 = r2.lastName;
        if (!last1.equals(last2))
            return last1.compareTo(last2);
        else {
            String first1 = r1.firstName, first2 = r2.firstName;
            return first1.compareTo(first2);
        }
    }
});
```

9.13 Exercises

1. Show the steps of the selection sort, bubble sort, and insertion sort when applied to the array [R, A, K, C, M]. Specifically, trace through each algorithm and each time a swap is made, indicate the new state of the array after the swap.

2. Show the steps that Quicksort and Mergesort take when sorting [K, E, A, E, O, P, R, S, V, Y, U, B].

3. Explain how Quicksort can degenerate to $O(n^2)$ if the array consists of just a few values repeated quite a bit, like an array of 1000 ones, 1500 twos, and 80 threes, all jumbled up.

4. At the end of Section 9.2 was a suggestion for improving the bubble sort by checking to see if any swaps were made on a pass through the array. If no swaps are made, then the array must be sorted and the algorithm can be stopped. Implement this.
5. In Section 9.11, we showed how to write the Selection Sort to work with a generic list. Rewrite the Quicksort method in the same way.

6. In Section 8.5 we had a class called `Pair` that represents an ordered pair \((x, y)\). Create a list of pairs and sort the pairs by their \(x\) coordinate. Then sort them by their \(y\) coordinate.

7. Exercise 11 of Chapter 7 is about creating a map of words and their frequencies from a document. Using that map, do the following:
   
   (a) Print out the words and their frequencies, ordered alphabetically.
   
   (b) Print out the words and their frequencies, ordered from most to least frequent.
   
   (c) Print out the words and their frequencies, ordered from least to most frequent.

8. Implement the following sorting algorithms to sort integer arrays.

   (a) Cocktail Sort — This is a relative of Bubble Sort. Assume the \(n\) elements are in indices 0, 1, \(\ldots\) \(n - 1\). Start by comparing elements 0 and 1, then elements 1 and 2, and so on down to the end of the array, swapping each time if the elements are out of order. When we’re done, the largest element will be in place at the end of the array. Now go backwards. Start at the second to last element (index \(n - 2\)) and compare it with the one before it (index \(n - 3\)). Then compare the elements at \(n - 3\) and \(n - 4\), the elements at \(n - 4\) and \(n - 5\), etc. down to the front of the list. After this, the smallest element will be in place at the front of the array. Now proceed forward again, starting by comparing elements 2 and 3, then 3 and 4, etc., down to \(n - 3\) and \(n - 2\). After this the second to last element will be correct. Then go backwards again, then forwards, then backwards, each time stopping one position short of the previous stopping place, until there is nowhere left to move.

   (b) Hash Table Counting Sort — This is like Counting Sort, except instead of using an array of counts, it uses HashMap of counts.

   (c) Introsort — This is a combination of Quicksort and Heapsort. Introsort uses the Quicksort algorithm to sort the array until some maximum recursion depth is reached (say 16 levels deep), at which point it switches to Heapsort to finish the sorting.

   (d) Odd/even sort — Implement the following sorting algorithm: Start by comparing the elements at indices 1 and 2 and swapping if they are out of order. Then compare the elements at indices 3 and 4, 5 and 6, 7 and 8, etc., swapping whenever necessary. This is called the “odd” part of the algorithm. Then compare the elements at indices 0 and 1, 2 and 3, 4 and 5, 6 and 7, etc., swapping whenever necessary. This is called the “even” part of the algorithm. What you do is do the odd part, then the even part, then the odd part, then the even part, etc. repeating this process until the list is sorted. One way to tell if the list is sorted is that once you do an odd part followed by an even part, and no swaps are made in either part, then the list is sorted.

   (e) Comb Sort — Comb Sort is a modification of Bubble Sort that instead of comparing adjacent elements, instead compares elements that are a certain gap apart. There are several passes with the gap size gradually decreasing. Essentially, Comb Sort is to Bubble Sort as Shell Sort is to Insertion Sort. The starts at size \(n/1.3\) and by a factor of 1.3 until the gap size is 1, which is what is used for the final pass. On each pass, an algorithm analogous to Bubble Sort
is performed except that instead of comparing adjacent elements, we compare elements that are one gap apart.

(f) Bucket Sort — This sort puts the data into different “buckets,” sorts the buckets individually, and then puts the buckets back together. Write a simple version of this algorithm that assumes the data consists of integers from 0 to 999999, sorts it into buckets of size 100 (0-99, 100-199, 200-299), and uses Insertion Sort on each bucket.

(g) Radix Sort — This sort works by sorting integers based on their ones digit, then on their tens digit, etc. A simple version of this can be implemented as follows: Put the elements of the array into one of ten buckets, based off their ones digit. Put the elements back into the array by starting with all the elements in the 0 bucket, followed by all the elements in the 1 bucket, etc. Then put the elements back into buckets by their tens digit and fill the array back up in order from the buckets. Repeat this process until there are no more digits to work with.

(h) Permutation Sort — This is a really bad sort, but it is a good exercise to try to implement it. It works by looking at all the permutations of the elements until it finds the one where all the elements are in sorted order. One way to implement it is to generate all the permutations of the indices 0, 1, …, n − 1 (see Section 4.6), create a new array with the elements in the order specified by the permutation, and check to see if the elements of the new array are in order.

(i) Bogosort — This is another really bad sort—it runs in worse than \(O(n!)\) time. The way it works is it randomly shuffles the elements of the array until they are in order. One way to implement this is to put the elements of the array into a list and use \texttt{Collections.shuffle}.

9. Analyze the running times of the following sorts from the previous exercise.

(a) Cocktail Sort
(b) Introsort
(c) Permutation Sort
Appendix A

A few Java prerequisites

This appendix reviews a few object-oriented topics that show up from time to time in the book.

A.1 Interfaces

Here is an example to help explain what an interface is and what it is useful for.

Say we have a program that has a bunch of on-screen widgets. Each of the widgets does something different when clicked, but other than that, the widgets are similar. Our program also has a method that handles widgets. It would be nice if we could just pass that method a general Widget object, but since each of the widgets has a different onClick method, it would seem that each one would have to be a different class, with no general Widget class possible.

The solution to this is an interface. An interface is kind of like a template for a class. An interface specifies a set of methods. It doesn’t specify the code for those methods, just what the methods are and their signatures. Any class implementing the interface must provide the code for those methods, and the key is that each class can have its own different implementation of the method.

So, in our example above, we can create a Widget interface that acts as a template for the on-screen widgets. Each one of those widget classes will implement the interface and specify its own code for the onClick method. The interface might look something like this:

```java
public interface Widget {
    void onClick();
}
```

It specifies the signature of the function. In this case, the function returns nothing and takes two integer parameters. Notice that we don’t specify public/private here. A specialized widget might implement the interface as follows:

```java
public class drawingWidget implements Widget {
    void onClick();
}
```
Roughly speaking, a class is to an interface as an object is to a class. A class is a template for objects, and an interface is a template for classes.

Interfaces are also useful if you are creating a data structure that may have several different implementations. As an example, the Java Collections Framework contains an interface called List (lists are covered in Chapter 2). There are two different classes that implement the List interface: ArrayList and LinkedList. The difference between the two is how they are implemented, and there are some places where an ArrayList will perform better and some places where a LinkedList will perform better. The reason for the interface is that we have a bunch of methods that take ArrayList parameters, like the one below:

```java
ArrayList<Integer> listThatGetsPassedToABunchOfMethods = new ArrayList<Integer>();
public void method1(ArrayList<Integer> a) { ... }
public void method2(ArrayList<Integer> a) { ... }
public void method3(ArrayList<Integer> a) { ... }
public void method4(ArrayList<Integer> a) { ... }
```

Suppose requirements change and some of those ArrayLists need to be LinkedLists. We have to go through and manually change all the method parameters. On the other hand, if we used the list interface in the parameter lists, the only place we would have to change is the declaration of the list that gets passed to the methods. This is what our code should look like:

```java
List<Integer> listThatGetsPassedToABunchOfMethods = new ArrayList<Integer>();
public void method1(List<Integer> a) { ... }
public void method2(List<Integer> a) { ... }
public void method3(List<Integer> a) { ... }
public void method4(List<Integer> a) { ... }
```

If requirements change, only the first line needs to be changed.

One place that interfaces show up is GUI programming in Java. To get a button to do something when clicked for instance, the ActionListener interface is used. There are lots of other interfaces, like MouseListener or KeyboardListener that are used.

A class can implement more than one interface. One the other hand, a class cannot inherit from more than one class in Java, so interfaces are the closest we can get to multiple inheritance in Java.

Here is a simple example of how interfaces work. We will create an interface called Pair that represents a pair \((x, y)\) of integers. The interface will have two methods, `getX` and `getY`.

```java
public interface Pair
{
    int getX();
    int getY();
}
```
We will create two classes implementing this interface. One class will implement it using two variables for \( x \) and \( y \) and the other will implement it using an array. Here is the approach using two variables:

```java
public class VPair implements Pair {
    int x, y;
    public VPair(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
    public int getY() { return y; }
}
```

Here is the approach using an array:

```java
public class APair implements Pair {
    int[] vars;
    public APair(int x, int y) {
        vars = new int[2];
        vars[0] = x;
        vars[1] = y;
    }
    public int getX() { return vars[0]; }
    public int getY() { return vars[1]; }
}
```

We can then use these classes as follows:

```java
Pair p = new APair(3, 4);
Pair q = new VPair(4, 9);
System.out.println(p.getX() + " " + q.getX());
```

This is admittedly a little too simple of an example, as an interface is overkill here. The purpose is just to show with a really simple example how to create an interface.
A.2 Abstract classes

Abstract classes are closely related to interfaces. To see where they’re useful, let’s go back to the widget example from the previous section. In that example, we had a bunch of on-screen widgets, each with a different onClick method, so that each widget did something different when clicked. Suppose that our widgets have a bunch of other methods and the code for those methods is the exact same thing for every widget. For example, maybe there are methods to reposition a widget that are the same for every widget. If we use an interface, we will have to copy and paste the same repositioning code into all of the widget classes.

An abstract class allows us to avoid the copying and pasting. The way it works is that an abstract class still acts like a template for a class, but we can actually include in it the code that will be the same in all the widget classes. We can also specify variables. Here is how it might look in the widget example.

```java
public abstract class Widget {
    protected int xLocation, yLocation;
    public void reposition(int x, int y) {
        // repositioning code goes here
    }
    abstract void onClick();
}
```

Our specialized widgets will be subclasses of this class. Here is an example:

```java
public class drawingWidget {
    public void onClick() {
        // code for what to do when clicked would go here
    }
}
```

A.3 Referencing

Let’s say we have a card class with fields value and suit, along with some getters and setters. In a card game, say we use lists of card objects to represent players’ hands. Suppose we try the following to check if the player’s hand contains the 3 of clubs.

```java
if (hand.contains(new Card(3, "C")))
    // do something
```

This will always return `false`, whether or not the card is actually in the user’s hand. The reason for this is that the card in the user’s hand and the new card we create are different objects residing in different parts of memory, even though they have the same fields. The contains method internally calls the Card object’s equals method to check for equality while it is searching through the list. The equals method
A.3. REFERENCE

by default just compares objects' locations in memory and will only return true if two objects that reside in the same place in memory are being compared.

There are a couple of solutions to this. The first is to write our own contains method and use that. It might look something like this:

```java
public static boolean contains(List<Card> list, Card card)
{
    for (Card c : list)
        if (c.value == card.value && c.suit.equals(card.suit))
            return true;
    return false;
}
```

A second way is to override the Card class's equals method. A simple way to do that is shown below:

```java
// this is part of the ~Card class
public boolean equals(Object obj)
{
    Card other = (Card) obj;
    return value == other.value && suit.equals(other.suit);
}
```

When overriding equals, the Java documentation recommends overriding another method called hashCode. This is covered in Section 7.6. The easiest way to do this is to let an IDE generate the equals and hashCode methods for you. Here is the code that Eclipse generates:

```java
public int hashCode()
{
    final int prime = 31;
    int result = 1;
    result = prime * result + ((suit == null) ? 0 : suit.hashCode());
    result = prime * result + value;
    return result;
}

@Override
public boolean equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Card other = (Card) obj;
    if (suit == null)
    {
        if (other.suit != null)
            return false;
    }
    else if (!suit.equals(other.suit))
        return false;
    if (value != other.value)
        return false;
    return true;
}
```
Overriding equals is better than writing our own contains method because by overriding equals, we have made it possible to compare our card objects in other settings, too.

The lesson to remember here is when you are working with custom objects, if you want to compare them (even implicitly like with a list’s contains method), you have to specify how the objects are compared, and that is best done by overriding the object’s equals method.

A.4 Throwing exceptions

When writing classes that other people might use, it's nice a good idea to throw exceptions when the class is misused in some way. For example, when implementing a list class, it would be good to throw an exception if the user tries to access a list element at an index that doesn't exist. If we don't do this, then the user may be faced with bizarre undefined behavior or a cryptic exception referring to something internal to our class.

One way to handle exceptions is to use one of Java's predefined exceptions. For example, the following code might be useful if the user of a list class tries to access a negative list index:

```java
public int get(int index)
{
    if (index<0)
        throw new IndexOutOfBoundsException();
}
```

We can be even nicer and provide some information to the user, like below:

```java
public int get(int index)
{
    if (index<0)
        throw new IndexOutOfBoundsException("Bad index was: " + i);
}
```

Java contains a lot of predefined exceptions, such as IllegalArgumentException or NamingException. A quick web search will turn up a list of all the exceptions. If none of the predefined exceptions fits, we can define our own exception class.

As an example, suppose we have a list class and we want to raise an exception if someone tries to remove elements from an empty list. We can define our class as follows:

```java
public class EmptyListException extends RuntimeException
{
    public EmptyListException()
    {
        super();
    }

    public EmptyListException(String message)
    {
        super(message);
    }
}
```
Our class provides two constructors, one of which allows the exception-thrower to specify a message and the other that allows just a plain exception to be thrown. Here is how the class might be used:

```java
if (size==0)
    throw new EmptyListException();

if (size==0)
    throw new EmptyListException("Why did you try deleting from an empty list?");
```

On the other hand, if we just want something quick and dirty, we can skip creating a new exception class and just do the following:

```java
if (size==0)
    throw new RuntimeException("The list is empty.");
```

Though, for any sort of bigger project, you would want to avoid this.

**Checked versus unchecked exceptions**

One thing to note about the EmptyListException class is we inherited from RuntimeException. We could also have inherited from Exception. The difference is that the former way is **unchecked**, whereas the latter is **checked**.

A checked exception is one that anyone using our class must take care of in their code, using `throws` statements or `try/catch`. If the user doesn't deal with a checked exception, the compiler issues an error. For example, the IOException that one has to worry about when working with text files in Java is a checked exception. An unchecked exception is one that will not result in any errors until something goes wrong while the program is running.

While it is definitely easier to use unchecked exceptions, it is often better practice to use checked exceptions so that people using your class know what sorts of things can go wrong and how to deal with them. The Java documentation has the following to say: “If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.”
APPENDIX A. A FEW JAVA PREREQUISITES
Index

abstract classes, 220
adjacent vertices, 160
ArrayDeque, 63
ArrayList, 42
Arrays library, 46, 212
big Θ notation, 8
big O notation, 5, 7–16
binary search, 6, 45, 47
binary search trees, 101–112, 201
running time, 111
binary trees, 85–95
applications, 98
running time, 96
breadth-first search, 64, 169–172
Bubble Sort, 197–198
cache, 38, 209
call stack, 64, 81
Collections methods, 45
combinations, 73–74
Comparable interface, 112–116, 123
Comparators, 212
connected graph, 161
Counting sort, 208–209
cycle, 161
deep copy, 44
degree, 160
depth-first search, 64, 169–172
deques, 61
Collections Framework, 62
digraphs, 172–181
Dijkstra's shortest path algorithm, 187–191
Dijkstra's Shunting Yard Algorithm, 64
dynamic arrays, 21–27
    compared to linked lists, 36–38
    running time, 36–38
degree, 159
equals, 221
exponential growth, 15, 79
factorial, 15
foreach loop, 46, 48–52
generating the subsets of a set, 11
generics, 38, 42, 211
graphs, 98, 159–191
    applications, 167, 172, 178
    weighted, 181–184
hashCode, 143–144
hashcode, 221
hashing, 135–141, 149–151
    chaining, 140
    load factor, 140
    open addressing, 140
    probing, 140
HashMap, 151
HashSet, 141
heaps, 117–123, 201
    applications, 123
    running time, 123
Heapsort, 201
Insertion Sort, 198–199
interfaces, 217–219
Iterable interface, 48, 165
Iterating, 46
    iterating, 48–52
Java Collections Framework, 41–46, 61–63, 98, 116,
    123, 126–127, 141, 151–152, 212
    iterating, 46
Java Iterators, 48–52
Kruskal’s algorithm, 184–187
linear search, 5
linked lists, 27–36
  circularly-linked, 36
  compared to dynamic arrays, 36–38
  doubly-linked, 36
  generic class, 36–41
  running time, 36–38
LinkedHashMap, 151
LinkedHashSet, 141
LinkedList, 42, 63
list, 21
List interface, 41
lists
  applications, 47
little o notation, 8
logarithms, 7, 13
Map interface, 151
maps, 144–155
  applications, 152–155
  running time, 149
Merge Sort, 202–205
mode of an array, 4
Moore’s law, 15
neighbor, 160
path, 161
permutations, 72–73
postfix notation, 64
priority queues, 124–127, 188
  applications, 127
queues, 55, 58–61, 170
  applications, 63–64
  Collections Framework, 61
  running time, 61
Quicksort, 205–208
recursion, 67–81
Selection Sort, 9, 196–197
Set interface, 141
sets, 131–134, 136–142
  applications, 141–142
  running time, 134, 140
shallow copy, 43
Shellsort, 199–200
Sierpinski triangle, 70–72
spanning tree, 184
stable sort, 195
stacks, 55–58, 80, 171
  applications, 63–64
  Collections Framework, 62
  running time, 58
Towers of Hanoi, 77–80
traveling salesman problem, 16
TreeMap, 151
TreeSet, 141
Treesort, 201–202
vertex, 159