that takes the most thought: it shows the number of comparisons each merge performs.

If you look at the pseudocode (or your implementation) of merge, you should convince yourself that in the worst case it takes $m - 1$ comparisons, where m is the total number items being merged.

The next step is to multiply the number of merges at each level by the amount of work (comparisons) per merge. The result is the total work at each level. At this point we take advantage of a small trick. We know that in the end we are only interested in the leading-order term in the result, so we can go ahead and ignore the -1 term in the comparisons per merge. If we do that, then the total work at each level is simply n.

Next we need to know the number of levels as a function of n. Well, we start with an array of n items and divide it in half until it gets to 1. That's the same as starting at 1 and multiplying by 2 until we get to n. In other words, we want to know how many times we have to multiply 2 by itself before we get to n. The answer is that the number of levels, l, is the logarithm, base 2, of n.

Finally, we multiply the amount of work per level, n, by the number of levels, $\log_2 n$ to get $n \log_2 n$, as promised. There isn't a good name for this functional form; most of the time people just say, "en log en."

It might not be obvious at first that $n \log_2 n$ is better than n^2, but for large values of n, it is. As an exercise, write a program that prints $n \log_2 n$ and n^2 for a range of values of n.

18.4 Overhead

Performance analysis takes a lot of handwaving. First we ignored most of the operations the program performs and counted only comparisons. Then we decided to consider only worst case performance. During the analysis we took the liberty of rounding a few things off, and when we finished, we casually discarded the lower-order terms.

When we interpret the results of this analysis, we have to keep all this handwaving in mind. Because mergesort is $n \log_2 n$, we consider it a better algorithm than selection sort, but that doesn't mean that mergesort is *always* faster. It just means that eventually, if we sort bigger and bigger arrays, mergesort will win.

How long that takes depends on the details of the implementation, including the additional work, besides the comparisons we counted, that each algorithm performs. This extra work is sometimes called **overhead**. It doesn't affect the performance analysis, but it does affect the run time of the algorithm.

For example, our implementation of mergesort actually allocates subarrays before making the recursive calls and then lets them get garbage collected after

they are merged. Looking again at the diagram of mergesort, we can see that the total amount of space that gets allocated is proportional to $n \log_2 n$, and the total number of objects that get allocated is about $2n$. All that allocating takes time.

Even so, it is most often true that a bad implementation of a good algorithm is better than a good implementation of a bad algorithm. The reason is that for large values of n the good algorithm is better and for small values of n it doesn't matter because both algorithms are good enough.

As an exercise, write a program that prints values of $1000n \log_2 n$ and n^2 for a range of values of n. For what value of n are they equal?

18.5 Priority Queue implementations

In Chapter 16 we looked at an implementation of a Priority Queue based on an array. The items in the array are unsorted, so it is easy to add a new item (at the end), but harder to remove an item, because we have to search for the item with the highest priority.

An alternative is an implementation based on a sorted list. In this case when we add a new item we traverse the list and put the new item in the right spot. This implementation takes advantage of a property of lists, which is that it is easy to add a new node into the middle. Similarly, removing the item with the highest priority is easy, provided that we keep it at the beginning of the list.

Performance analysis of these operations is straightforward. Adding an item to the end of an array or removing a node from the beginning of a list takes the same amount of time regardless of the number of items. So both operations are constant time.

Any time we traverse an array or list, performing a constant-time operation on each element, the run time is proportional to the number of items. Thus, removing something from the array and adding something to the list are both linear time.

So how long does it take to add and then remove n items from a Priority Queue? For the array implementation, n adds takes time proportional to n, but the removals take longer. The first removal has to traverse all n items; the second has to traverse $n - 1$, and so on, until the last removal, which only has to look at 1 item. Thus, the total time is $1 + 2 + \cdots + n$, which is $n^2/2 + n/2$. So the total for the adds and the removals is the sum of a linear function and a quadratic function, which we would characterize as quadratic.

The analysis of the list implementation is similar. The first add doesn't require any traversal, but after that we have to traverse at least part of the list each time we add a new item. In general we don't know how much of the list we will have to traverse, since it depends on the data and what order they are

added, but we can assume that on average we have to traverse half of the list. Unfortunately, even traversing half of the list is a linear operation.

So, once again, to add and remove n items takes time proportional to n^2. Thus, based on this analysis we cannot say which implementation is better; the array and list are both quadratic time implementations.

If we implement a Priority Queue using a heap, we can perform both adds and removals in time proportional to $logn$. Thus the total time for n items is $n \log n$, which is better than n^2. That's why, at the beginning of the chapter, I said that a heap is a particularly efficient implementation of a Priority Queue.

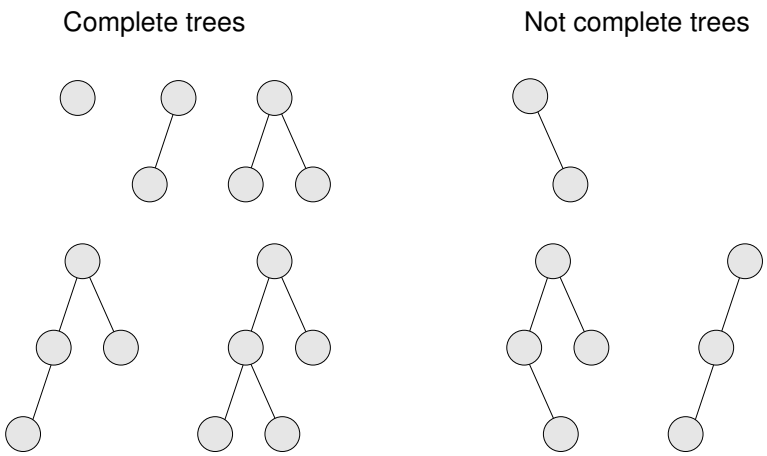
18.6 Definition of a Heap

A heap is a special kind of tree. It has two properties that are not generally true for other trees:

completeness: The tree is complete, which means that nodes are added from top to bottom, left to right, without leaving any spaces.

heapness: The item in the tree with the highest priority is at the top of the tree, and the same is true for every subtree.

Both of these properties call for a little explaining. This figure shows a number of trees that are considered complete or not complete:



An empty tree is also considered complete. We can define completeness more rigorously by comparing the height of the subtrees. Recall that the **height** of a tree is the number of levels.

Starting at the root, if the tree is complete, then the height of the left subtree and the height of the right subtree should be equal, or the left subtree may be

taller by one. In any other case, the tree cannot be complete. Furthermore, if the tree is complete, then the height relationship between the subtrees has to be true for every node in the tree.

The **heap property** is similarly recursive. In order for a tree to be a heap, the largest value in the tree has to be at the root, *and* the same has to be true for each subtree. As another exercise, write a method that checks whether a tree has the heap property.

Exercise 18.1 Write a method that takes a Tree as a parameter and checks whether it is complete.

HINT: You can use the `height` method from Exercise 17.4.

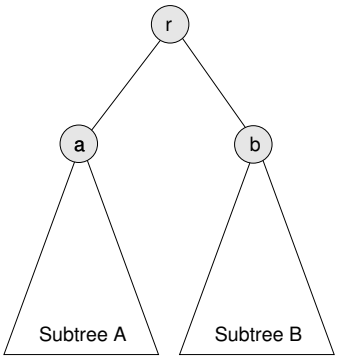
Exercise 18.2 Write a method that takes a Tree as a parameter and checks whether it has the heap property.

18.7 Heap `remove`

It might seem odd that we are going to remove things from the heap before we add any, but I think removal is easier to explain.

At first glance, we might think that removing an item from the heap is a constant time operation, since the item with the highest priority is always at the root. The problem is that once we remove the root node, we are left with something that is no longer a heap. Before we can return the result, we have to restore the heap property. We call this operation `reheapify`.

The situation is shown in the following figure:



The root node has priority `r` and two subtrees, A and B. The value at the root of Subtree A is `a` and the value at the root of Subtree B is `b`.

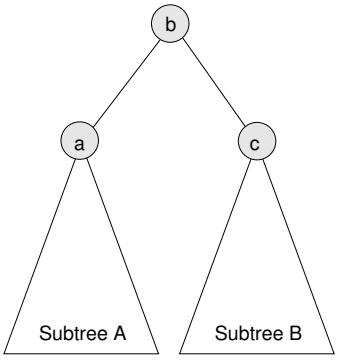
We assume that before we remove `r` from the tree, the tree is a heap. That implies that `r` is the largest value in the heap and that `a` and `b` are the largest values in their respective subtrees.

Once we remove `r`, we have to make the resulting tree a heap again. In other words we need to make sure it has the properties of completeness and heapness.

The best way to ensure completeness is to remove the bottom-most, rightmost node, which we'll call `c` and put its value at the root. In a general tree implementation, we would have to traverse the tree to find this node, but in the array implementation, we can find it in constant time because it is always the last (non-null) element of the array.

Of course, the chances are that the last value is not the highest, so putting it at the root breaks the heapness property. Fortunately it is easy to restore. We know that the largest value in the heap is either `a` or `b`. Therefore we can select whichever is larger and swap it with the value at the root.

Arbitrarily, let's say that `b` is larger. Since we know it is the highest value left in the heap, we can put it at the root and put `c` at the top of Subtree B. Now the situation looks like this:



Again, `c` is the value we copied from the last element in the array and `b` is the highest value left in the heap. Since we haven't changed Subtree A at all, we know that it is still a heap. The only problem is that we don't know if Subtree B is a heap, since we just stuck a (probably low) value at its root.

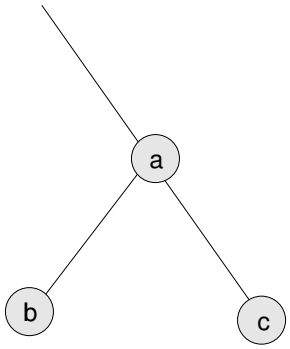
Wouldn't it be nice if we had a method that could `reheapify` Subtree B? Wait... we do!

18.8 Heap `add`

Adding a new item in a heap is a similar operation, except that instead of trickling a value down from the top, we trickle it up from the bottom.

Again, to guarantee completeness, we add the new element at the bottom-most, rightmost position in the tree, which is the next available space in the array.

Then to restore the heap property, we compare the new value with its neighbors. The situation looks like this:



The new value is `c`. We can restore the heap property of this subtree by comparing `c` to `a`. If `c` is smaller, then the heap property is satisfied. If `c` is larger, then we swap `c` and `a`. The swap satisfies the heap property because we know that `c` must also be bigger than `b`, because `c > a` and `a > b`.

Now that the subtree is reheapified, we can work our way up the tree until we reach the root.

18.9 Performance of heaps

For both add and remove, we perform a constant time operation to do the actual addion and removal, but then we have to reheapify the tree. In one case we start at the root and work our way down, comparing items and then recursively reheapifying one of the subtrees. In the other case we start at a leaf and work our way up, again comparing elements at each level of the tree.

As usual, there are several operations we might want to count, like comparisons and swaps. Either choice would work; the real issue is the number of levels of the tree we examine and how much work we do at each level. In both cases we keep examining levels of the tree until we restore the heap property, which means we might only visit one, or in the worst case we might have to visit them all. Let's consider the worst case.

At each level, we perform only constant time operations like comparisons and swaps. So the total amount of work is proportional to the number of levels in the tree, a.k.a. the height.

So we might say that these operations are linear with respect to the height of the tree, but the "problem size" we are interested in is not height, it's the number of items in the heap.

As a function of n, the height of the tree is $log_2 n$. This is not true for all trees, but it is true for complete trees. To see why, think of the number of nodes on each level of the tree. The first level contains 1, the second contains 2, the third contains 4, and so on. The ith level contains 2^i nodes, and the total number in all levels up to i is $2^i - 1$. In other words, $2^h = n$, which means that $h = log_2 n$.

Thus, both add and remove take **logarithmic** time. To add and remove n items takes time proportional to $n \log_2 n$.

18.10 Heapsort

The result of the previous section suggests yet another algorithm for sorting. Given n items, we add them into a Heap and then remove them. Because of the Heap semantics, they come out in order. We have already shown that this algorithm, which is called **heapsort**, takes time proportional to $n \log_2 n$, which is better than selection sort and the same as mergesort.

As the value of n gets large, we expect heapsort to be faster than selection sort, but performance analysis gives us no way to know whether it will be faster than mergesort. We would say that the two algorithms have the same **order of growth** because their runtimes grow with the same functional form. Another way to say the same thing is that they belong to the same **complexity class**.

Complexity classes are sometimes written in "big-O notation". For example, $\mathcal{O}(n^2)$, pronounced "oh of en squared" is the set of all functions that grow no faster than n^2 for large values of n. To say that an algorithm is $\mathcal{O}(n^2)$ is the same as saying that it is quadratic. The other complexity classes we have seen, in decreasing order of performance, are:

$\mathcal{O}(1)$	constant time
$\mathcal{O}(\log n)$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	"en log en"
$\mathcal{O}(n^2)$	quadratic
$\mathcal{O}(2^n)$	exponential

So far none of the algorithms we have looked at are **exponential**. For large values of n, these algorithms quickly become impractical. Nevertheless, the phrase "exponential growth" appears frequently in even non-technical language. It is frequently misused so I wanted to include its technical meaning.

People often use "exponential" to describe any curve that is increasing and accelerating (that is, one that has positive slope and curvature). Of course, there are many other curves that fit this description, including quadratic functions (and higher-order polynomials) and even functions as undramatic as $n \log n$. Most of these curves do not have the (often detrimental) explosive behavior of exponentials.

18.11 Glossary

selection sort: The simple sorting algorithm in Section 12.3.

mergesort: A better sorting algorithm from Section 12.6.

heapsort: Yet another sorting algorithm.

complexity class: A set of algorithms whose performance (usually run time) has the same order of growth.

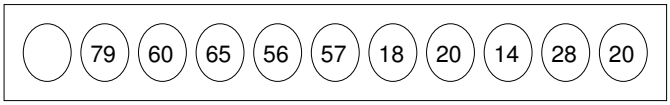
order of growth: A set of functions with the same leading-order term, and therefore the same qualitative behavior for large values of n.

overhead: Additional time or resources consumed by a program performing operations other than the abstract operations considered in performance analysis.

18.12 Exercises

Exercise 18.3

a. Draw the Heap represented by the following array.



b. Show what the array would look like after the value 68 is added to the Heap.

Exercise 18.4 Assume that there are n elements in a Heap. To find the median value of the elements, we could remove $n/2 - 1$ elements and then return the value of the $n/2$-eth element. Then we would have to put $n/2$ elements back into the Heap. What would be the order of growth of this algorithm?

Exercise 18.5 How many times will the following loop execute? Express your answer as a function of `n`:

```
while (n > 1) {
    n = n / 2;
}
```

Exercise 18.6 How many recursive calls will zippo make? Express your answer as a function of `x` or `n` or both.

```
public static double zippo (double x, int n) {
    if (n == 0) return 1.0;
    return x * zippo (x, n-1);
}
```

Exercise 18.7 Write an implementation of a Heap based on the array implementation of a tree. The run time of the `add` and `remove` operations should be proportional to $\log n$, where n is the number of elements in the Heap.

Chapter 19

Maps

19.1 Arrays, Vectors and Maps

Arrays are a generally useful data structure, but they suffer from two important limitations:

- The size of the array does not depend on the number of items in it. If the array is too big, it wastes space. If it is too small it might cause an error, or we might have to write code to resize it.
- Although the array can contain any type of item, the indices of the array have to be integers. We cannot, for example, use a String to specify an element of an array.

In Section 17.9 we saw how the built-in `Vector` class solves the first problem. As the client code adds items the Vector expands automatically. It is also possible to shrink a Vector so that the capacity is the same as the current size.

But `Vectors` don't help with the second problem. The indices are still integers. That's where the Map ADT comes in. A Map is similar to a Vector; the difference is that it can use any object type as an index. These generalized indices are called **keys**.

Just as you use an index to access a value in a Vector, you use a key to access a value in a Map. Each key is associated with a value, which is why Maps are sometimes called **associative arrays**. The association of a particular key with a particular value is called an **entry**.

A common example of a map is a dictionary, which maps words (the keys) onto their definitions (the values). Because of this example Maps are also sometimes called Dictionaries. And, just for completeness, Maps are also sometimes called Tables.

19.2 The Map ADT

Like the other ADTs we have looked at, Maps are defined by the set of operations they support:

constructor: Make a new, empty map.

`put`: Create an entry that associates a value with a key.

`get`: For a given key, find the corresponding value.

`containsKey`: Return `true` if there is an entry in the Map with the given Key.

`keySet` : Returns a Set that contains all the keys in the Map.

19.3 The built-in HashMap

Java provides an implementation of the Map ADT called `java.util.HashMap`. Later in the chapter we'll see why it is called `HashMap`.

To demonstrate the use of a HashMap we'll write a short program that traverses a String and counts the number of times each word appears.

We'll create a new class called `WordCount` that will build the Map and then print its contents. Each `WordCount` object contains a `HashMap` as an instance variable:

```
public class WordCount {
    HashMap map;

    public WordCount () {
        map = new HashMap ();
    }
}
```

The only public methods for `WordCount` are `processLine`, which takes a String and adds its words to the Map, and `print`, which prints the results at the end.

`processLine` breaks the String into words using a `StringTokenizer` and passes each word to `processWord`.

```
    public void processLine (String s) {
        StringTokenizer st = new StringTokenizer (s, " ,.");
        while (st.hasMoreTokens()) {
            String word = st.nextToken();
            processWord (word.toLowerCase ());
        }
    }
```

The interesting work is in `processWord`.

```
    public void processWord (String word) {
        if (map.containsKey (word)) {
            Integer i = (Integer) map.get (word);
            Integer j = new Integer (i.intValue() + 1);
            map.put (word, j);
        } else {
            map.put (word, new Integer (1));
        }
    }
```

If the word is already in the map, we `get` its counter, increment it, and `put` the new value. Otherwise, we just `put` a new entry in the map with the counter set to `1`.

To print the entries in the map, we need to be able to traverse the keys in the map. Fortunately, `HashMap` provides a method, `keySet`, that returns a `Set` object, and `Set` provides a method `iterator` that returns (you guessed it) an Iterator.

Here's how to use `keySet` to print the contents of the `HashMap`:

```
public void print () {
    Set set = map.keySet();
    Iterator it = set.iterator();
    while (it.hasNext ()) {
        String key = (String) it.next ();
        Integer value = (Integer) map.get (key);
        System.out.println ("{ " + key + ", " + value + " }");
    }
}
```

Each of the elements of the Iterator is an Object, but since we know they are keys, we typecast them to be Strings. When we get the values from the Map, they are also Objects, but we know they are counters, so we typecast them to be Integers.

Finally, to count the words in a string:

```
        WordCount wc = new WordCount ();
        wc.processLine ("you spin me right round baby " +
                        "right round like a record baby " +
                        "right round round round");
        wc.print ();
```

The output is

```
{ you, 1 }
{ round, 5 }
{ right, 3 }
{ me, 1 }
{ like, 1 }
{ baby, 2 }
{ spin, 1 }
```

```
{ record, 1 }
{ a, 1 }
```

The elements of the Iterator are not in any particular order. The only guarantee is that all the keys in the map will appear.

19.4 A Vector implementation

An easy way to implement the Map ADT is to use a `Vector` of entries, where each Entry is an object that contains a key and a value. A class definition for `Entry` might look like this:

```
class Entry {
    Object key, value;

    public Entry (Object key, Object value) {
        this.key = key;
        this.value = value;
    }

    public String toString () {
        return "{ " + key + ", " + value + " }";
    }
}
```

Then the implementation of Map looks like this:

```
public class MyMap {
    Vector entries;

    public Map () {
        entries = new Vector ();
    }
}
```

To put a new entry in the map, we just `add` a new `Entry` to the `Vector`:

```
    public void put (Object key, Object value) {
        entries.add (new Entry (key, value));
    }
```

Then to look up a key in the Map we have to traverse the Vector and find an Entry with a matching key:

```
    public Object get (Object key) {
        Iterator it = entries.iterator ();
        while (it.hasNext ()) {
            Entry entry = (Entry) it.next ();
            if (key.equals (entry.key)) {
                return entry.value;
            }
        }
```

```
        return null;
    }
```

The idiom to traverse a `Vector` is the one we saw in Section 17.10. When we compare keys, we use deep equality (the `equals` method) rather than shallow equality (the `==` operator). This allows the key class to specify the definition of equality. In the example, the keys are Strings, so `get` will use the `equals` method in the `String` class.

For most of the built-in classes, the `equals` method implements deep equality. For some classes, though, it is not easy to define what that means. For example, see the documentation of `equals` for `Doubles`.

Because `equals` is an object method, this implementation of `get` does not work if a key is `null`. In order to handle `null` keys, we would have to add a special case to `get` or write a class method that compares keys and handles `null` parameters safely. But we will skip this detail.

Now that we have `get`, we can write a version of `put` that is more complete. If there is already an entry in the map with the given key, `put` is supposed to update it (give it a new value), and return the old value (or `null` if there was none). Here is an implementation that provides this feature:

```
    public Object put (Object key, Object value) {
        Object result = get (key);
        if (result == null) {
            Entry entry = new Entry (key, value);
            entries.add (entry);
        } else {
            update (key, value);
        }
        return result;
    }

    private void update (Object key, Object value) {
        Iterator it = entries.iterator ();
        while (it.hasNext ()) {
            Entry entry = (Entry) it.next ();
            if (key.equals (entry.key)) {
                entry.value = value;
                break;
            }
        }
    }
```

The `update` method is not part of the Map ADT, so it is declared `private`. It traverses the vector until it finds the right `Entry` and then it updates the `value` field. Notice that we don't modify the `Vector` itself, just one of the objects it contains.

The only methods we haven't implemented are `containsKey` and `keySet`. The `containsKey` method is almost identical to `get` except that it returns `true` or

`false` instead of an object reference or `null`.

Exercise 19.1 Implement `keySet` by building and returning a `TreeSet` object. Use your implementation of `TreeSet` from Exercise 17.7 or the built-in implementation `java.util.TreeSet`.

19.5 The `List` metaclass

The `java.util` package defines a metaclass called `List` that specifies the set of operations a class has to implement in order to be considered (very abstractly) a list. This does not mean, of course, that every class that implements `List` has to be a linked list.

Not surprisingly, the built-in `LinkedList` class is a member of the `List` metaclass. Surprisingly, so is `Vector`.

The methods in the `List` definition include `add`, `get` and `iterator`. In fact, all the methods from the `Vector` class that we used to implement `Map` are defined in the `List` metaclass. That means that instead of a `Vector`, we could have used any `List` class. In our implementation of `Map` we can replace `Vector` with `LinkedList`, and the program still works!

This kind of type generality can be useful for tuning the performance of a program. You can write the program in terms of a metaclass like `List` and then test the program with several different implementations to see which yields the best performance.

19.6 HashMap implementation

The reason that the built-in implementation of the Map ADT is called `HashMap` is that it uses a particularly efficient implementation of a Map called a hash table.

In order to understand the hash table implementation, and why it is considered efficient, we'll start by analyzing the performance of the `List` implementation.

Looking at the implementation of `put`, we see that there are two cases. If the key is not already in the map, then we only have to create a new entry and `add` it to the `List`. Both of these are constant-time operations.

In the other case, we have to traverse the `List` to find the existing entry. That's a linear time operation. For the same reason, `get` and `containsKey` are also linear.

Although linear operations are often good enough, we can do better. It turns out that there is a way to implement the Map ADT so that both `put` and `get` are constant time operations!

The key is to realize that traversing a list takes time proportional to the length of the list. If we can put an upper bound on the length of the list, then we can put an upper bound on the traverse time, and anything with a fixed upper bound is considered constant time.

But how can we limit the length of the lists without limiting the number of items in the map? By increasing the number of lists. Instead of one long list, we'll keep many short lists.

As long as we know which list to search, we can put a bound on the amount of searching.

19.7 Hash Functions

And that's where hash functions come in. We need some way to look at a key and know, without searching, which list it will be in. We'll assume that the lists are in an array (or `Vector`) so we can refer to them by index.

The solution is to come up with some mapping—almost any mapping—between the key values and the indices of the lists. For every possible key there has to be a single index, but there might be many keys that map to the same index.

For example, imagine an array of 8 Lists and a map made up of keys that are `Integers` and values that are `Strings`. It might be tempting to use the `intValue` of the `Integers` as indices, since they are the right type, but there are a whole lot of integers that do not fall between 0 and 7, which are the only legal indices.

The modulus operator provides a simple (in terms of code) and efficient (in terms of run time) way to map *all* the integers into the range $(0, 7)$. The expression

```
key.intValue() % 8
```

is guaranteed to produce a value in the range from -7 to 7 (including both). If you take its absolute value (using `Math.abs`) you will get a legal index.

For other types, we can play similar games. For example, to convert a `Character` to an integer, we can use the built-in method `Character.getNumericValue` and for `Doubles` there is `intValue`.

For `Strings` we could get the numeric value of each character and add them up, or instead we might use a **shifted sum**. To calculate a shifted sum, you alternate between adding new values to the accumulator and shifting the accumulator to the left. By "shift to the left" I mean "multiply by a constant."

To see how this works, take the list of numbers $1, 2, 3, 4, 5, 6$ and compute their shifted sum as follows. First, initialize the accumulator to 0. Then,

1. Multiply the accumulator by 10.
2. Add the next element of the list to the accumulator.

3. Repeat until the list is finished.

As an exercise, write a method that calculates the shifted sum of the numeric values of the characters in a `String` using a multiplier of 16.

For each type, we can come up with a function that takes values of that type and generates a corresponding integer value. These functions are called **hash functions**, because they often involve making a hash of the components of the object. The integer value for each object is called its **hash code**.

There is one other way we might generate a hash code for Java objects. Every Java object provides a method called `hashCode` that returns an integer that corresponds to that object. For the built-in types, the `hashCode` method is implemented so that if two objects contain the same data (deep equality), they will have the same hash code. The documentation of these methods explains what the hash function is. You should check them out.

For user-defined types, it is up to the implementor to provide an appropriate hash function. The default hash function, provided in the `Object` class, often uses the location of the object to generate a hash code, so its notion of "sameness" is shallow equality. Most often when we are searching a Map for a key, shallow equality is not what we want.

Regardless of how the hash code is generated, the last step is to use modulus and absolute value functions to map the hash code into the range of legal indices.

19.8 Resizing a hash map

Let's review. A hash table consists of an array (or Vector) of `Lists`, where each `List` contains a small number of entries. To add a new entry to a map, we calculate the hash code of the new key and add the entry to the corresponding `List`.

To look up a key, we hash it again and search the corresponding list. If the lengths of the lists are bounded then the search time is bounded.

So how do we keep the lists short? Well, one goal is to keep them as balanced as possible, so that there are no very long lists at the same time that others are empty. This is not easy to do perfectly—it depends on how well we chose the hash function—but we can usually do a pretty good job.

Even with perfect balance, the average list length grows linearly with the number of entries, and we have to put a stop to that.

The solution is to keep track of the average number of entries per list, which is called the **load factor**; if the load factor gets too high, we have to resize the hash table.

To resize, we create a new hash table, usually twice as big as the original, take all the entries out of the old one, hash them again, and put them in the new hash table. Usually we can get away with using the same hash function; we just use a different value for the modulus operator.

19.9 Performance of resizing

How long does it take to resize the hash table? Clearly it is linear with the number of entries. That means that *most* of the time `put` takes constant time, but every once in a while —when we resize—it takes linear time.

At first that sounds bad. Doesn't that undermine my claim that we can perform `put` in constant time? Well, frankly, yes. But with a little wheedling, we can fix it.

Since some `put` operations take longer than others, let's figure out the *average* time for a put operation. The average is going to be c, the constant time for a simple `put`, plus an additional term of p, the percentage of the time I have to resize, times kn, the cost of resizing.

$$t(n) = c + p \cdot kn \tag{19.1}$$

We don't know what c and k are, but we can figure out what p is. Imagine that we have just resized the hash map by doubling its size. If there are n entries, then we can add an addition n entries before we have to resize again. So the percentage of the time we have to resize is $1/n$.

Plugging into the equation, we get

$$t(n) = c + 1/n \cdot kn = c + k \tag{19.2}$$

In other words, $t(n)$ is constant time!

19.10 Glossary

map: An ADT that defines operations on a collection of entries.

entry: An element in a map that contains a key and a value.

key: An index, of any type, used to look up values in a map.

value: An element, of any type, stored in a map.

dictionary: Another name for a map.

associative array: Another name for a dictionary.

hash map: A particularly efficient implementation of a map.

hash function: A function that maps values of a certain type onto integers.

hash code: The integer value that corresponds to a given value.

shifted sum: A simple hash function often used for compounds objects like `Strings`.

load factor: The number of entries in a hashmap divided by the number of lists in the hashmap; i.e. the average number of entries per list.

19.11 Exercises

Exercise 19.2

a. Compute the shifted sum of the numbers 1, 2, 3, 4, 5, 6 using a multiplier of 10.

b. Compute the shifted sum of the numbers 11, 12, 13, 14, 15, 16 using a multiplier of 100.

c. Compute the shifted sum of the numbers 11, 12, 13, 14, 15, 16 using a multiplier of 10.

Exercise 19.3 Imagine you have a hash table that contains 100 lists. If you are asked to `get` the value associated with a certain key, and the hash code for that key is 654321, what is the index of the list where the key will be (if it is in the map)?

Exercise 19.4 If there are 100 lists and there are 89 entries in the map and the longest list contains 3 entries, and the median list length is 1 and 19% of the lists are empty, what is the load factor?

Exercise 19.5 Imagine that there are a large number of people at a party. You would like to know whether any two of them have the same birthday. First, you design a new kind of Java object that represents birthdays with two instance variables: `month`, which is an integer between 1 and 12, and `day`, which is an integer between 1 and 31.

Next, you create an array of `Birthday` objects that contains one object for each person at the party, and you get someone to help type in all the birthdays.

a. Write an `equals` method for `Birthday` so that Birthdays with the same month and day are equal.

b. Write a method called `hasDuplicate` that takes an array of `Birthdays` and returns `true` if there are two or more people with the same birthday. Your algorithm should only traverse the array of `Birthdays` once.

c. Write a method called `randomBirthdays` that takes an integer `n` and returns an array with `n` random `Birthday` objects. To keep things simple, you can pretend that all months have 30 days.

d. Generate 100 random arrays of 10 `Birthdays` each, and see how many of them contain duplicates.

e. If there are 20 people at a party, what is the probability that two or more of them have the same birthday?

Exercise 19.6 A Map is said to be **invertible** if each key and each value appears only once. In a HashMap, it is always true that each key appears only once, but it is possible for the same value to appear many times. Therefore, some HashMaps are invertible and some are not.

Write a method called `isInvertible` that takes a `HashMap` and returns true if the map is invertible and false otherwise.

Exercise 19.7

The goal of this exercise is to find the 20 most common words in this book. Type in the code from the book that uses the built-in `HashMap` to count word frequencies. Download the text of the book from `http://thinkapjava.com/thinkapjava.txt`.

The plain text version of the book is generated automatically by a program called `detex` that tries to strip out all the typesetting commands, but it leaves behind assorted junk. Think about how your program should handle punctuation and any other wierdness that appears in the file.

Write programs to answer the following questions:

a. How many words are there in the book?

b. How many *different* words are there in the book? For purposes of comparison, there are (very roughly) 500,000 words in English, of which (very roughly) 50,000 are in current use. What percentage of this vast lexicon did I find fit to use?

c. How many times does the word "encapsulation" appear?

d. What are the 20 most common words in the book? HINT: there is another data structure that might help with this part of the exercise.

Exercise 19.8 Write a method for the `LinkedList` class that checks whether the list contains a loop. You should not assume that the length field is correct. Your method should be linear in the number of nodes.

Exercise 19.9 Write a hash table implementation of the Map ADT (as defined in Section 19.2). Test it by using it with any of the programs you have written so far. Hints:

a. Start with the `Vector` implementation in the book and make sure it works with existing programs.

b. Modify the implementation to use either an array of LinkedLists or a Vector of LinkedLists, whichever you prefer.

Exercise 19.10 Write an implementation of the `Set` interface using a `HashMap`.

Exercise 19.11

The `java.util` package provides two implementations of the `Map` interface, called `HashMap` and `TreeMap`. `HashMap` is based on a hash table like the one described in this chapter. `TreeMap` is based on a red-black tree, which is similar to the search tree in Exercise 17.6.

Although these implementations provide the same interface, we expect them to achieve different performance. As the number of entries, n, increases, we expect `add` and `contains` to take constant time for the hash table implementation, and logarithmic time for the tree implementation.

Perform an experiment to confirm (or refute!) these performance predictions. Write a program that adds n entries to a `HashMap` or `TreeMap`, then invokes `contains` on each of the keys. Record the runtime of the program for a range of values of n and make a plot of runtime versus n. Is the performance behavior in line with our expectations.

Chapter 20

Huffman code

20.1 Variable-length codes

If you are familiar with Morse code, you know that it is a system for encoding the letters of the alphabet as a series of dots and dashes. For example, the famous signal `...---...` represents the letters SOS, which comprise an internationally-recognized call for help. This table shows the rest of the codes:

A	.-	N	-.	1	.----	.	.-.-.-
B	-...	O	---	2	..---	,	--..--
C	-.-.	P	.--.	3	...--	?	..--..
D	-..	Q	--.-	4	....-	(	-.--.
E	.	R	.-.	5	.....	)	-.--.-
F	..-.	S	...	6	-....	-	-....-
G	--.	T	-	7	--...	"	.-..-.
H	....	U	..-	8	---..	_	..--.-
I	..	V	...-	9	----.	'	.----.
J	.---	W	.--	0	-----	:	---...
K	-.-	X	-..-	/	-..-.	;	-.-.-.
L	.-..	Y	-.--	+	.-.-.	$	...-..-
M	--	Z	--..	=	-...-		

Notice that some codes are longer than others. By design, the most common letters have the shortest codes. Since there are a limited number of short codes, that means that less common letters and symbols have longer codes. A typical message will have more short codes than long ones, which minimizes the average transmission time per letter.

Codes like this are called variable-length codes. In this chapter, we will look at an algorithm for generating a variable-length code called a **Huffman code**. It is an interesting algorithm in its own right, but it also makes a useful exercise because its implementation uses many of the data structures we have been studying.

Here is an outline of the next few sections:

- First, we will use a sample of English text to generate a frequency table. A frequency table is like a histogram; it counts the number of times each letter appears in the sample text.

- The heart of a Huffman code is the Huffman tree. We will use the frequency table to build the Huffman tree, and then use the tree to encode and decode sequences.

- Finally, we will traverse the Huffman tree and built a code table, which contains the sequence of dots and dashes for each letter.

20.2 The frequency table

Since the goal is to give short codes to common letters, we have to know how often each letter occurs. In Edgar Allen Poe's short story "The Gold Bug," one of the characters uses letter frequencies to crack a cypher. He explains,

> "Now, in English, the letter which most frequently occurs is e. Afterwards, the succession runs thus: a o i d h n r s t u y c f g l m w b k p q x z. E however predominates so remarkably that an individual sentence of any length is rarely seen, in which it is not the prevailing character."

So our first mission is to see whether Poe got it right. To check, I chose as my sample the text of "The Gold Bug" itself, which I downloaded from one of the Web sites that take advantage of its entry into the public domain.

Exercise 20.12 Write a class called `FreqTab` that counts the number of times each letter appears in a sample text. Download the text of your favorite public-domain short story, and analyse the frequency of the letters.

I found it most convenient to make `FreqTab` extend `HashMap`. Then I wrote a method called `increment` that takes a letter as a parameter and either adds or updates an entry in the HashMap for each letter.

You can use `keySet` to get the entries from the HashMap, and print a list of letters with their frequencies. Unfortunately, they will not appear in any particular order. The next exercise solves that problem.

Exercise 20.13 Write a class called `Pair` that represents a letter-frequency pair. Pair objects should contain a letter and a frequency as instance variables. `Pair` should implement `Comparable`, where the Pair with the higher frequency wins.

Now sort the letter-frequency pairs from the HashMap by traversing the set of keys, building `Pair` objects, adding the Pairs to a PriorityQueue, removing the Pairs from the PriorityQueue, and printing them in descending order by frequency.

How good was Poe's guess about the most frequent letters?

20.3 The Huffman Tree

The next step is to assemble the Huffman tree. Each node of the tree contains a letter and its frequency, and pointers to the left and right nodes.

To assemble the Huffman tree, we start by creating a set of singleton trees, one for each entry in the frequency table. Then we build the tree from the bottom up, starting with the least-frequent letters and iteratively joining subtrees until we have a single tree that contains all the letters.

Here is the algorithm in more detail.

1. For each entry in the frequency table, create a singleton Huffman tree and add it to a PriorityQueue. When we remove a tree from the PriorityQueue, we get the one with the lowest frequency.

2. Remove two trees from the PriorityQueue and join them by creating a new parent node that refers to the removed nodes. The frequency of the parent node is the sum of the frequencies of the children.

3. If the PriorityQueue is empty, we are done. Otherwise, put the new tree into the PriorityQueue and go to Step 2.

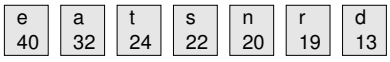
An example will make this clearer. To keep things managable, we will use a sample text that only contains the letters `adenrst`:

> Eastern Tennessee anteaters ensnare and eat red ants, detest ant antennae (a tart taste) and dread Antarean anteater-eaters. Rare Andean deer eat tender sea reeds, aster seeds and rats' ears. Dessert? Rats' asses.

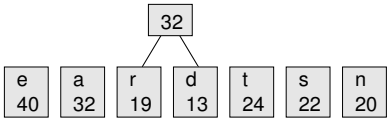
This eloquent treatise on the eating habits of various fauna yields the following frequency table:

```
e       40
a       32
t       24
s       22
n       20
r       19
d       13
```

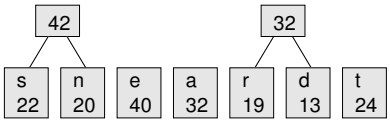
So after Step 1, the PriorityQueue looks like this:



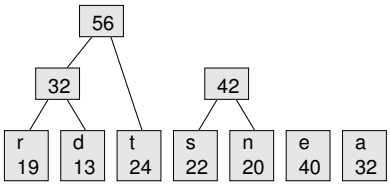
In Step 2, we remove the two trees with the lowest frequency (`r` and `d`) and join them by creating a parent node with frequency 32. The letter value for the internal nodes is irrelevant, so it is omitted from the figures. When we put the new tree back in the PriorityQueue, the result looks like this:



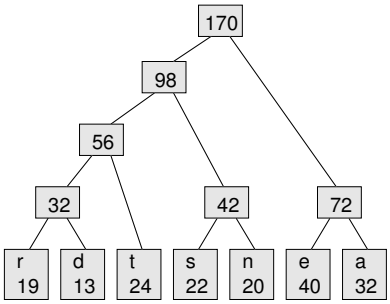
Now we repeat the process, combining `s` and `n`:



After the next interation, we have the following collection of trees. By the way, a collection of trees is called a **forest**.



After two more iterations, there is only one tree left:



This is the Huffman tree for the sample text. Actually, it is not the only one, because each time we join two trees, we choose arbitrarily which goes on the left and which on the right, and when there is a tie in the PriorityQueue, the choice is arbitrary. So there may be many possible trees for a given sample.

So how to we get a code from the Huffman tree? The code for each letter is determined by the path from the root of the tree to the leaf that contains the letter. For example, the path from the root to `s` is left-right-left. If we represent a left with `.` and a right with `-` (another arbitrary choice) we get the following code table.

```
e       -.
a       --
t       ..-
s       .-.
n       .--
r       ....
d       ...-
```

Notice that we have achieved the goal: the most frequent letters have the shortest codes.

Exercise 20.14 By hand, figure out a Huffman tree for the following frequency table:

```
e       93
s       71
r       57
t       53
n       49
i       44
d       43
o       37
```

20.4 The super method

One way to implement `HuffTree` is to extend `Pair` from Exercise 20.13.

```
public class HuffTree extends Pair implements Comparable {
    HuffTree left, right;

    public HuffTree (int freq, String letter,
                     HuffTree left, HuffTree right) {
        this.freq = freq;
        this.letter = letter;
        this.left = left;
        this.right = right;
    }
}
```

`HuffTree` implements `Comparable` so we can put HuffTrees into a PriorityQueue. In order to implement `Comparable`, we have to provide a `compareTo` method. We could write one from scratch, but it is easier to take advantage of the version of `compareTo` in the `Pair` class.

Unfortunately, the existing method doesn't do exactly what we want. For Pairs, we give priority to the higher frequency. For HuffTrees, we want to give priority to the lower frequency. Of course, we could write a new version of `compareTo`, but that would override the version in the parent class, and we would like to be able to invoke the version in the parent class.

Apparently we are not the first people to encounter this little **Catch 22**, because the nice people who invented Java have provided a solution. The keyword `super` allows us to invoke a method that has been overridden. It's called `super` because parent classes are sometimes called **superclasses**.

Here's an example from my implementation of `HuffTree`:

```
public int compareTo (Object obj) {
    return -super.compareTo (obj);
}
```

When `compareTo` is invoked on a `HuffTree`, it turns around and invokes the overridden version of `compareTo`, and then negates the result, which has the effect of reversing the order of priority.

When a child class (also called a **subclass**) overrides a constructor, it can invoke the parent's constructor using `super`:

```
public HuffTree (int freq, String letter,
                 HuffTree left, HuffTree right) {
    super (freq, letter);
    this.left = left;
    this.right = right;
}
```

In this example, the parent's constructor initializes `freq` and `letter`, and then the child's constructor initializes `left` and `right`.

Although this feature is useful, it is also error prone. There are some odd restrictions—the parent's constructor has to be invoked first, before the other instance variables are initialized—and there are some gotchas you don't even want to know about. In general, this mechanism is like a first aid kit. If you get into real trouble, it can help you out. But you know what's even better? Don't get into trouble. In this case, it is simpler to initialize all four instance variables in the child's constructor.

Exercise 20.15 Type in the `HuffTree` class definition from this section and add a class method called `build` that takes a `FreqTab` and returns a `HuffTree`. Use the algorithm in Section 20.3.

20.5 Decoding

When we receive an encoded message, we use the Huffman tree to decode it. Here is the algorithm:

1. Start at the root of the HuffTree.
2. If the next symbol is `.`, go to the left child; otherwise, go to the right child.
3. If you are at a leaf node, get the letter from the node and append it to the result. Go back to the root.
4. Go to Step 2.

Consider the code `..--.--`, as an example. Starting at the top of the tree, we go left-left-right and get to the letter `t`. Then we start at the root again, go right-left and get to the letter `e`. Back to the top, then right-right, and we get

the letter `a`. If the code is well-formed, we should be at a leaf node when the code ends. In this case the message is my beverage of choice, tea.

Exercise 20.16 Use the example HuffTree to decode the following words:

a. `.-.--.--..---`

b. `.-...--.-......-.`

c. `...--.-.....`

d. `-.--.-...-`

Notice that until you start decoding, you can't tell how many letters there are or where the boundaries fall.

Exercise 20.17 Write a class definition for `Huffman`. The constructor should take a string that contains sample text, and it should build a frequency table and a Huffman tree.

Write a method called `decode` that takes a String of dots and dashes and that uses the Huffman tree to decode the String and return the result.

Note: even if you use the sample string in Section 20.2, you won't necessarily get the same HuffTree as in Section 20.3, so you (probably) won't be able to use your program to decode the examples in the previous exercise.

20.6 Encoding

In a sense, encoding a message is harder than decoding, because for a given letter we might have to search the tree to find the leaf node that contains the letter, and then figure out the path from the root to that node.

This process is much more efficient if we traverse the tree once, compute all the codes, and build a Map from letters to codes.

By now we have seen plenty of tree traversals, but this one is unusual because as we move around the tree, we want to keep track of the path we are on. At first, that might seem hard, but there is a natural way to perform this computation recursively. Here is the key observation: if the path from the root to a given node is represented by a string of dots and dashes called `path`, then the path to the left child of the node is `path + '-'` and the path to the right child is `path + '.'`.

Exercise 20.18

a. Write a class definition for `CodeTab`, which extends `HashMap`.

b. In the CodeTab definition, write a recursive method called `getCodes` that traverses a HuffTree in any order. When it reaches a leaf node, it should print the letter in the node and the code that represents the path from the root to the node.

c. Once `getCodes` is working, modify it so that when it reaches a leaf node, it makes an entry in the HashMap, with the letter as the key and the code as the value.

d. Write a constructor for `CodeTab` that takes a HuffTree as a parameter and that invokes `getCodes` to build the code table.

e. In the `Huffman` class, write a method called `encode` that traverses a string, looks up each character in the code table, and returns the encoding of the string. Test this method by passing the result to `decode` and see if you get the original string back.

20.7 Glossary

forest: A collection of trees (duh!).

Catch 22: A situation in which the apparent solution to a problem seems inevitably to preclude any solution to the problem. The term comes from Joseph Heller's book of the same title (and the excellent movie directed by Stanley Kubrick).

superclass: Another name for a parent class.

subclass: Another name for a child class.

`super`: A keyword that can be used to invoke an overridden method from a superclass.

Appendix A

Program development plan

If you are spending a lot of time debugging, it is probably because you do not have an effective **program development plan**.

A typical, bad program development plan goes something like this:

1. Write an entire method.
2. Write several more methods.
3. Try to compile the program.
4. Spend an hour finding syntax errors.
5. Spend an hour finding run time errors.
6. Spend three hours finding semantic errors.

The problem, of course, is the first two steps. If you write more than one method, or even an entire method, before you start the debugging process, you are likely to write more code than you can debug.

If you find yourself in this situation, the *only* solution is to remove code until you have a working program again, and then gradually build the program back up. Beginning programmers are often unwilling to do this, because their carefully crafted code is precious to them. To debug effectively, you have to be ruthless!

Here is a better program development plan:

1. Start with a working program that does something visible, like printing something.
2. Add a small number of lines of code at a time, and test the program after every change.
3. Repeat until the program does what it is supposed to do.

After every change, the program should produce some visible effect that demonstrates the new code. This approach to programming can save a lot of time. Because you only add a few lines of code at a time, it is easy to find syntax errors. Also, because each version of the program produces a visible result, you are constantly testing your mental model of how the program works. If your mental model is erroneous, you will be confronted with the conflict (and have a chance to correct it) before you have written a lot of erroneous code.

One problem with this approach is that it is often difficult to figure out a path from the starting place to a complete and correct program.

I will demonstrate by developing a method called `isIn` that takes a String and a character, and that returns a boolean: `true` if the character appears in the String and `false` otherwise.

1. The first step is to write the shortest possible method that will compile, run, and do something visible:

   ```
   public static boolean isIn (char c, String s) {
       System.out.println ("isIn");
       return false;
   }
   ```

   Of course, to test the method we have to invoke it. In `main`, or somewhere else in a working program, we need to create a simple test case.

   We'll start with a case where the character appears in the String (so we expect the result to be `true`).

   ```
   public static void main (String[] args) {
       boolean test = isIn ('n', "banana");
       System.out.println (test);
   }
   ```

   If everything goes according to plan, this code will compile, run, and print the word `isIn` and the value `false`. Of course, the answer isn't correct, but at this point we know that the method is getting invoked and returning a value.

   In my programming career, I have wasted way too much time debugging a method, only to discover that it was never getting invoked. If I had used this development plan, it never would have happened.

2. The next step is to check the parameters the method receives.

   ```
   public static boolean isIn (char c, String s) {
       System.out.println ("isIn looking for " + c);
       System.out.println ("in the String " + s);
       return false;
   }
   ```