Part II

Algorithms and Building Blocks

CHAPTER

5 Algorithm Analysis

IN Part I we examined how object-oriented programming can help in the design and implementation of large systems. We did not examine performance issues. Generally, we use a computer because we need to process a large amount of data. When we run a program on large amounts of input, we must be certain that the program terminates within a reasonable amount of time. Although the amount of running time is somewhat dependent on the programming language we use, and to a smaller extent the methodology we use (such as procedural versus object-oriented), often those factors are unchangeable constants of the design. Even so, the running time is most strongly correlated with the choice of algorithms.

An *algorithm* is a clearly specified set of instructions the computer will follow to solve a problem. Once an algorithm is given for a problem and determined to be correct, the next step is to determine the amount of resources, such as time and space, that the algorithm will require. This step is called *algorithm analysis*. An algorithm that requires several gigabytes of main memory is not useful for most current machines, even if it is completely correct.

In this chapter, we show:

- How to estimate the time required for an algorithm
- How to use techniques that drastically reduce the running time of an algorithm
- How to use a mathematical framework that more rigorously describes the running time of an algorithm
- How to write a simple *binary search* routine

5.1 What Is Algorithm Analysis?

More data means that the program takes more time.

The amount of time that any algorithm takes to run almost always depends on the amount of input that it must process. We expect, for instance, that sorting 10,000 elements requires more time than sorting 10 elements. The running time of an algorithm is thus a function of the input size. The exact value of the function depends on many factors, such as the speed of the host machine, the quality of the compiler, and in some cases, the quality of the program. For a given program on a given computer, we can plot the running time function on a graph. Figure 5.1 illustrates such a plot for four programs. The curves represent four common functions encountered in algorithm analysis: linear, $O(N \log N)$, quadratic, and cubic. The input size N ranges from 1 to 100 items, and the running times range from 0 to 10 milliseconds. A quick glance at Figure 5.1 and its companion, Figure 5.2, suggests that the linear, O(N log N), quadratic, and cubic curves represent running times in order of decreasing preference.

An example is the problem of downloading a file over the Internet. Suppose there is an initial 2-sec delay (to set up a connection), after which the download proceeds at 1.6 K/sec. Then if the file is N kilobytes, the time to download is described by the formula $T(N) = N/1.6 + 2$. This is a *linear function*. Downloading an 80K file takes approximately 52 sec, whereas downloading a file twice as large (160K) takes about 102 sec, or roughly twice as long. This property, in which time essentially is directly proportional to amount of input, is the signature of a *linear algorithm*, which is the most efficient algorithm. In contrast, as these first two graphs show, some of the nonlinear algorithms lead to large running times. For instance, the linear algorithms is much more efficient than the cubic algorithm.

Of the common functions encountered in algorithm analysis, linear represents the most efficient algorithm.

In this chapter we address several important questions:

- Is it always important to be on the most efficient curve?
- How much better is one curve than another?
- How do you decide which curve a particular algorithm lies on?
- How do you design algorithms that avoid being on less-efficient curves?

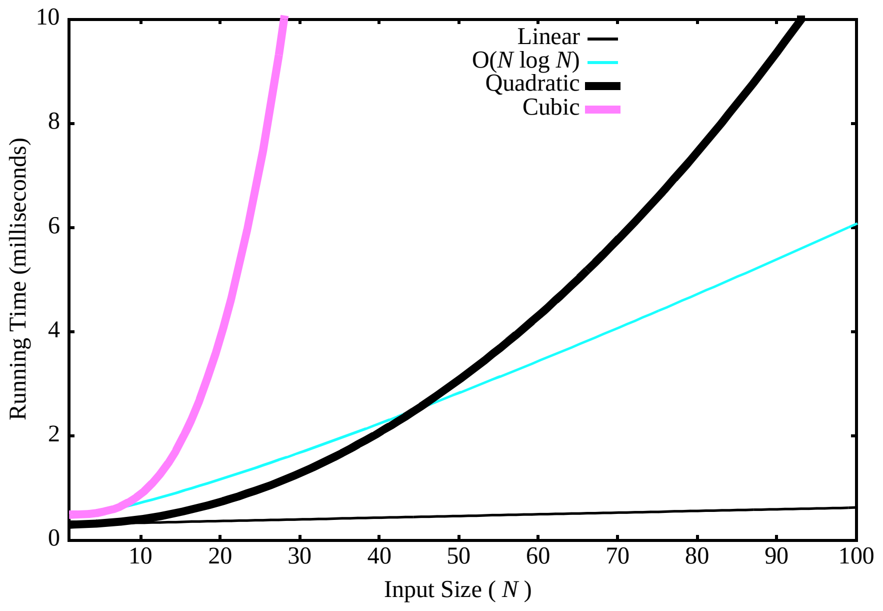

Figure 5.1 Running times for small inputs

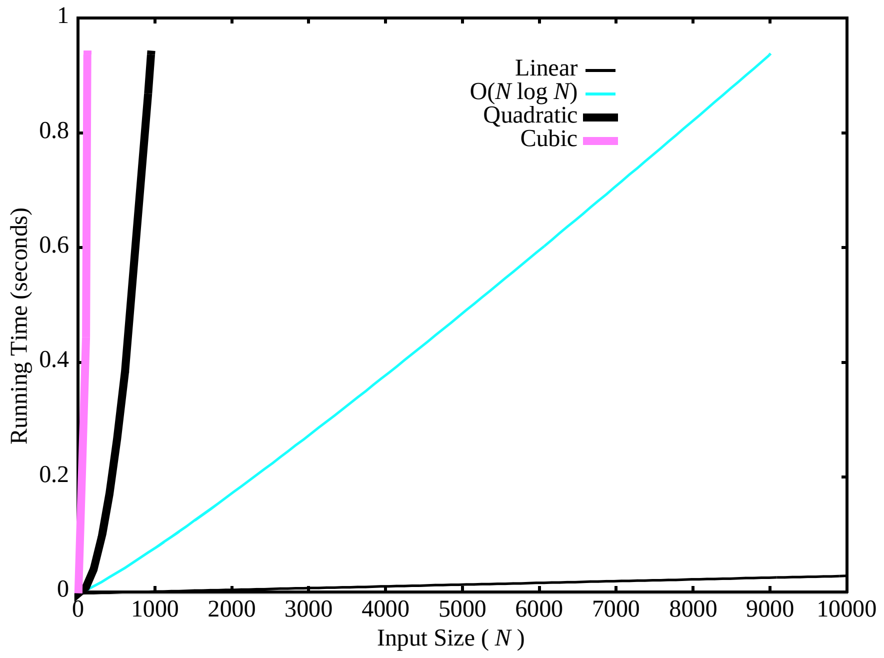

Figure 5.2 Running times for moderate inputs

A *cubic function* is a function whose dominant term is some constant times N^3. As an example, $10N^3 + N^2 + 40N + 80$ is a cubic function. Similarly, a quadratic function has a dominant term that is some constant times N^2, and a linear function has a dominant term that is some constant times N. The expression $O(N \log N)$ represents a function whose dominant term is N times the logarithm of N. The logarithm is a slowly growing function; for instance, the logarithm of 1,000,000 (with the typical base 2) is only 20. The logarithm grows more slowly than a square or cube (or any) root. We discuss the logarithm in more depth in Section 5.5.

The growth rate of a function is most important when N is sufficiently large.

Either of two functions may be smaller than the other at any given point, so claiming, for instance, that $F(N) < G(N)$ does not make sense. Instead, we measure the functions' rates of growth. This is justified for three reasons. First, for cubic functions such as the one shown in Figure 5.2, when N is 1,000 the value of the cubic function is almost entirely determined by the cubic term. In the function $10N^3 + N^2 + 40N + 80$, for $N = 1{,}000$, the value of the function is 10,001,040,080, of which 10,000,000,000 is due to the $10N^3$ term. If we were to use only the cubic term to estimate the entire function, an error of about 0.01 percent would result. For sufficiently large N, the value of a function is largely determined by its dominant term (the meaning of the term *sufficiently large* varies by function).

The second reason we measure the functions' growth rates is that the exact value of the leading constant of the dominant term is not meaningful across different machines (although the relative values of the leading constant for identically growing functions might be). For instance, the quality of the compiler could have a large influence on the leading constant. The third reason is that small values of N generally are not important. For $N = 20$, Figure 5.1 shows that all algorithms terminate within 5 ms. The difference between the best and worst algorithm is less than a blink of the eye.

We use *Big-Oh* notation to capture the most dominant term in a function and to represent the growth rate. For instance, the running time of a quadratic algorithm is specified as $O(N^2)$ (pronounced "order en-squared"). Big-Oh notation also allows us to establish a relative order among functions by comparing dominant terms. We discuss Big-Oh notation more formally in Section 5.4.

Big-Oh notation is used to capture the most dominant term in a function.

For small values of N (for instance, those less than 40), Figure 5.1 shows that one curve may be initially better than another, which doesn't hold for larger values of N. For example, initially the quadratic curve is better than the $O(N \log N)$ curve, but as N gets sufficiently large, the quadratic algorithm loses its advantage. For small amounts of input, making comparisons between functions is difficult because leading constants become very significant. The function $N + 2{,}500$ is larger than N^2 when N is less than 50. Eventually, the linear function is always less than the quadratic function. Most important, for small input sizes the running times are generally inconsequential, so we need not worry about them. For instance, Figure 5.1 shows that when N is less than 25, all four algorithms run in less than 10 ms. Consequently, when input sizes are very small, a good rule of thumb is to use the simplest algorithm.

Figure 5.2 clearly demonstrates the differences between the various curves for large input sizes. A linear algorithm solves a problem of size 10,000 in a small fraction of a second. The $O(N \log N)$ algorithm uses roughly 10 times as much time. Note that the actual time differences depend on the constants involved and thus might be more or less. Depending on these constants, an $O(N \log N)$ algorithm might be faster than a linear algorithm for fairly large input sizes. For equally complex algorithms, however, linear algorithms tend to win out over $O(N \log N)$ algorithms.

Quadratic algorithms are impractical for input sizes exceeding a few thousand.

Cubic algorithms are impractical for input sizes as small as a few hundred.

This relationship is not true, however, for the quadratic and cubic algorithms. Quadratic algorithms are almost always impractical when the input size is more than a few thousand, and cubic algorithms are impractical for input sizes as small as a few hundred. For instance, it is impractical to use a naive sorting algorithm for 100,000 items, because most simple sorting algorithms (such as bubble sort and selection sort) are quadratic algorithms. The sorting algorithms discussed in Chapter 8 run in *subquadratic* time (that is, better than $O(N^2)$), thus making sorting large arrays practical.

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Figure 5.3 Functions in order of increasing growth rate

The most striking feature of these curves is that the quadratic and cubic algorithms are not competitive with the others for reasonably large inputs. We can code the quadratic algorithm in highly efficient machine language and do a poor job coding the linear algorithm, and the quadratic algorithm will still lose badly. Even the most clever programming tricks cannot make an inefficient algorithm fast. Thus, before we waste effort attempting to optimize code, we need to optimize the algorithm. Figure 5.3 arranges functions that commonly describe algorithm running times in order of increasing growth rate.

5.2 Examples of Algorithm Running Times

In this section we examine three problems. We also sketch possible solutions and determine what kind of running times the algorithms will exhibit, without providing detailed programs. The goal in this section is to provide you with some intuition about algorithm analysis. In Section 5.3 we provide more details on the process, and in Section 5.4 we formally approach an algorithm analysis problem.

We look at the following problems in this section:

MINIMUM ELEMENT IN AN ARRAY

Given an array of N items, find the smallest item.

CLOSEST POINTS IN THE PLANE

Given N points in a plane (that is, an x-y coordinate system), find the pair of points that are closest together.

COLINEAR POINTS IN THE PLANE

Given N points in a plane (that is, an x-y coordinate system), determine if any three form a straight line.

The minimum element problem is fundamental in computer science. It can be solved as follows:

1. Maintain a variable `min` that stores the minimum element.
2. Initialize `min` to the first element.
3. Make a sequential scan through the array and update `min` as appropriate.

The running time of this algorithm will be $O(N)$, or linear, because we will repeat a fixed amount of work for each element in the array. A linear algorithm is as good as we can hope for. This is because we have to examine every element in the array, a process that requires linear time.

The closest points problem is a fundamental problem in graphics that can be solved as follows:

1. Calculate the distance between each pair of points.
2. Retain the minimum distance.

This calculation is expensive, however, because there are $N(N-1)/2$ pairs of points.[1] Thus there are roughly N^2 pairs of points. Examining all these pairs and finding the minimum distance among them takes quadratic time. A better algorithm runs in $O(N \log N)$ time and works by avoiding the computation of all distances. There is also an algorithm that is expected to take $O(N)$ time. These last two algorithms use subtle observations to provide faster results and are beyond the scope of this text.

The colinear points problem is important for many graphics algorithms. The reason is that the existence of colinear points introduces a degenerate case that requires special handling. It can be directly solved by enumerating all groups of three points. This solution is even more computationally expensive than that for the closest points problem because the number of different groups of three points is $N(N-1)(N-2)/6$ (using reasoning similar to that used for the closest points problem). This result tells us that the direct approach will yield a cubic algorithm. There is also a more clever strategy (also beyond the scope of this text) that solves the problem in quadratic time (and further improvement is an area of continuously active research).

In Section 5.3 we look at a problem that illustrates the differences among linear, quadratic, and cubic algorithms. We also show how the performance of these algorithms compares to a mathematical prediction. Finally, after discussing the basic ideas, we examine Big-Oh notation more formally.

---

1. Each of *N* points can be paired with $N-1$ points, for a total of $N(N-1)$ pairs. However, this pairing double counts pairs *A*, *B* and *B*, *A*, so we must divide by two.

5.3 The Maximum Contiguous Subsequence Sum Problem

In this section, we consider the following problem:

MAXIMUM CONTIGUOUS SUBSEQUENCE SUM PROBLEM

Given (possibly negative) integers $A_1, A_2, \ldots, A_N$, find (and identify the sequence corresponding to) the maximum value of $\sum_{k=i}^{j} A_k$. The maximum contiguous subsequence sum is zero if all the integers are negative.

As an example, if the input is {–2, **11**, **–4**, **13**, –5, 2}, then the answer is 20, which represents the contiguous subsequence encompassing items 2 through 4 (shown in boldface type). As a second example, for the input { 1, –3, **4**, **–2**, **–1**, **6** }, the answer is 7 for the subsequence encompassing the last four items.

Programming details are considered after the algorithm design.

In Java, arrays begin at zero, so a Java program would represent the input as a sequence A_0 to A_{N-1}. This is a programming detail and not part of the algorithm design.

Always consider emptiness.

Before the discussion of the algorithms for this problem, we need to comment on the degenerate case in which all input integers are negative. The problem statement gives a maximum contiguous subsequence sum of 0 for this case. One might wonder why we do this, rather than just returning the largest (that is, the smallest in magnitude) negative integer in the input. The reason is that the empty subsequence, consisting of zero integers, is also a subsequence, and its sum is clearly 0. Because the empty subsequence is contiguous, there is always a contiguous subsequence whose sum is 0. This result is analogous to the empty set being a subset of any set. Be aware that emptiness is always a possibility and that in many instances it is not a special case at all.

The maximum contiguous subsequence sum problem is interesting mainly because there are so many algorithms to solve it — and the performance of these algorithms varies drastically. In this section we discuss three such algorithms. The first is an obvious exhaustive search algorithm, but it is very inefficient. The second is an improvement on the first, which is accomplished by a simple observation. The third is a very efficient, but not obvious, algorithm. We prove that its running time is linear.

There are lots of drastically different algorithms (in terms of efficiency) that can be used to solve the maximum contiguous subsequence sum problem.

In Chapter 7 we present a fourth algorithm, which has $O(N \log N)$ running time. That algorithm is not as efficient as the linear algorithm, but it is much more efficient than the other two. It is also typical of the kinds of algorithms that result in $O(N \log N)$ running times. The graphs shown in Figures 5.1 and 5.2 are representative of these four algorithms.

5.3.1 The Obvious $O(N^3)$ Algorithm

The simplest algorithm is a direct exhaustive search, or a *brute force algorithm*, as shown in Figure 5.4. Lines 9 and 10 control a pair of loops that iterate over all possible subsequences. For each possible subsequence, the value of its sum is computed at lines 12 to 15. If that sum is the best sum encountered, we update the value of `maxSum`, which is eventually returned at line 25. Two `ints` — `seqStart` and `seqEnd` (which are static class fields) — are also updated whenever a new best sequence is encountered.

A brute force algorithm is generally the least efficient but simplest method to code.

The direct exhaustive search algorithm has the merit of extreme simplicity; the less complex an algorithm is, the more likely it is to be programmed correctly. However, exhaustive search algorithms are usually not as efficient as possible. In the remainder of this section we show that the running time of the algorithm is cubic. We count the number of times (as a function of the input size) the expressions in Figure 5.4 are evaluated. We require only a Big-Oh result, so once we have found a dominant term, we can ignore lower order terms and leading constants.

```
/**
 * Cubic maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */
public static int maxSubsequenceSum( int [ ] a )
{
    int maxSum = 0;

    for( int i = 0; i < a.length; i++ )
        for( int j = i; j < a.length; j++ )
        {
            int thisSum = 0;

            for( int k = i; k <= j; k++ )
                thisSum += a[ k ];

            if( thisSum > maxSum )
            {
                maxSum = thisSum;
                seqStart = i;
                seqEnd   = j;
            }
        }

    return maxSum;
}
```

Figure 5.4 A cubic maximum contiguous subsequence sum algorithm

The running time of the algorithm is entirely dominated by the innermost `for` loop in lines 14 and 15. Four expressions there are repeatedly executed:

1. the initialization `k = i`,
2. the test `k <= j`,
3. the increment `thisSum += a[ k ]`, and
4. the adjustment `k++`.

A mathematical analysis is used to count the number of times that certain statements are executed.

The number of times expression 3 is executed makes it the dominant term among the four expressions. Note that each initialization is accompanied by at least one test. We are ignoring constants, so we may disregard the cost of the initializations; the initializations cannot be the single dominating cost of the algorithm. Because the test given by expression 2 is unsuccessful exactly once per loop, the number of unsuccessful tests performed by expression 2 is exactly equal to the number of initializations. Consequently, it is not dominant. The number of successful tests at expression 2, the number of increments performed by expression 3, and the number of adjustments at expression 4 are all identical. Thus the number of increments (i.e., the number of times that line 15 is executed) is a dominant measure of the work performed in the innermost loop.

The number of times line 15 is executed is exactly equal to the number of ordered triplets (i, j, k) that satisfy $1 \le i \le k \le j \le N$.[2] The reason is that the index i runs over the entire array, j runs from i to the end of the array, and k runs from i to j. A quick and dirty estimate is that the number of triplets is somewhat less than

[2]. In Java, the indices run from 0 to $N - 1$. We have used the algorithmic equivalent 1 to N to simplify the analysis.

$N \times N \times N$, or N^3, because *i*, *j*, and *k* can each assume one of *N* values. The additional restriction $i \le k \le j$ reduces this number. A precise calculation is somewhat difficult to obtain and is performed in Theorem 5.1.

The most important part of Theorem 5.1 is not the proof, but rather the result. There are two ways to evaluate the number of triplets. One is to evaluate the sum $\sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=i}^{j} 1$. We could evaluate this sum inside out (see Exercise 5.9). Instead, we will use an alternative.

Theorem 5.1

The number of integer ordered triplets (i, j, k) that satisfy $1 \le i \le k \le j \le N$ *is* $N(N+1)(N+2)/6$.

Proof *Place the following* $N + 2$ *balls in a box: N balls numbered 1 to N, one unnumbered red ball, and one unnumbered blue ball. Remove three balls from the box. If a red ball is drawn, number it as the lowest of the numbered balls drawn. If a blue ball is drawn, number it as the highest of the numbered balls drawn. Notice that if we draw both a red and blue ball, then the effect is to have three balls identically numbered. Order the three balls. Each such order corresponds to a triplet solution to the equation in Theorem 5.1. The number of possible orders is the number of distinct ways to draw three balls without replacement from a collection of* $N + 2$ *balls. This is similar to the problem of selecting three points from a group of N that we evaluated in Section 5.2, so we immediately obtain the stated result.*

The result of Theorem 5.1 is that the innermost `for` loop accounts for cubic running time. The remaining work in the algorithm is inconsequential because it is done, at most, once per iteration of the inner loop. Put another way, the cost of lines 17 to 22 is inconsequential because it is done only as often as the initialization of the inner `for` loop, rather than as often as the repeated body of the inner `for` loop. Consequently, the algorithm is $O(N^3)$.

The previous combinatoric argument allows us to obtain precise calculations on the number of iterations in the inner loop. For a Big-Oh calculation, this is not really necessary; we need to know only that the leading term is some constant times N^3. Looking at the algorithm, we see a loop that is potentially of size N inside a loop that is potentially of size N inside another loop that is potentially of size N. This configuration tells us that the triple loop has the potential for $N \times N \times N$ iterations. This potential is only about six times higher than what our precise calculation of what actually occurs. Constants are ignored anyway, so we can adopt the general rule that, when we have nested loops, we should multiply the cost of the innermost statement by the size of each loop in the nest to obtain an upper bound. In most cases, the upper bound will not be a gross overestimate.[3] Thus a program with three nested loops, each running sequentially through large portions of an array, is likely to exhibit $O(N^3)$ behavior. Note that three consecutive (nonnested) loops exhibit linear behavior; it is nesting that leads to a combinatoric explosion. Consequently, to improve the algorithm, we need to remove a loop.

We do not need precise calculations for a Big-Oh estimate. In many cases, we can use the simple rule of multiplying the size of all the nested loops. Note carefully that consecutive loops do not multiply.

3. Exercise 5.16 illustrates a case in which the multiplication of loop sizes yields an overestimate in the Big-Oh result.

5.3.2 An Improved $O(N^2)$ Algorithm

When we remove a nested loop from an algorithm, we generally lower the running time.

When can remove a nested loop from the algorithm, we genereally lower the running time. How do we remove a loop? Obviously, we cannot always do so. However, the previous algorithm has many unnecessary computations. The inefficiency that the improved algorithm corrects is the unduly expensive computation in the inner `for` loop in Figure 5.4. The improved algorithm makes use of the fact that $\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$. In other words, suppose we have just calculated the sum for the subsequence $i, ..., j-1$. Then computing the sum for the subsequence $i, ..., j$ should not take long because we need only one more addition. However, the cubic algorithm throws away this information. If we use this observation, we obtain the improved algorithm shown in Figure 5.5. We have two rather than three nested loops, and the running time is $O(N^2)$.

```java
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */
public static int maxSubsequenceSum( int [ ] a )
{
    int maxSum = 0;

    for( int i = 0; i < a.length; i++ )
    {
        int thisSum = 0;

        for( int j = i; j < a.length; j++ )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
            {
                maxSum = thisSum;
                seqStart = i;
                seqEnd   = j;
            }
        }
    }

    return maxSum;
}
```

Figure 5.5 A quadratic maximum contiguous subsequence sum algorithm

5.3.3 A Linear Algorithm

If we remove another loop, we have a linear algorithm.

The algorithm is tricky. It uses a clever observation to step quickly over large numbers of subsequences that cannot be the best.

To move from a quadratic algorithm to a linear algorithm, we need to remove yet another loop. However, unlike the reduction illustrated in Figures 5.4 and 5.5, where loop removal was simple, getting rid of another loop is not so easy. The problem is that the quadratic algorithm is still an exhaustive search; that is, we are trying all possible subsequences. The only difference between the quadratic and cubic algorithms is that the cost of testing each successive subsequence is a constant $O(1)$ instead of linear $O(N)$. Because a quadratic number of subsequences are possible, the only way we can attain a subquadratic bound is to find a clever way to eliminate from consideration a large number of subsequences, without actually computing their sum and testing to see if that sum is a new maximum. This section shows how this is done.

First, we eliminate a large number of possible subsequences from consideration. Let $A_{i,j}$ be the subsequence encompassing elements from *i* to *j*, and let $S_{i,j}$ be its sum.

.

Theorem 5.2 *Let $A_{i,j}$ be any sequence with $S_{i,j} < 0$. If $q > j$, then $A_{i,q}$ is not the maximum contiguous subsequence.*

Proof *The sum of A's elements from i to q is the sum of A's elements from i to j added to the sum of A's elements from j + 1 to q. Thus we have $S_{i,q} = S_{i,j} + S_{j+1,q}$. Because $S_{i,j} < 0$, we know that $S_{i,q} < S_{j+1,q}$. Thus $A_{i,q}$ is not a maximum contiguous subsequence.*

An illustration of the sums generated by i, j, and q is shown on the first two lines in Figure 5.6. Theorem 5.2 demonstrates that we can avoid examining several subsequences by including an additional test: If `thisSum` is less than 0, we can `break` from the inner loop in Figure 5.5. Intuitively, if a subsequence's sum is negative, then it cannot be part of the maximum contiguous subsequence. The reason is that we can get a large contiguous subsequence by not including it. This observation by itself is not sufficient to reduce the running time below quadratic. A similar observation also holds: All contiguous subsequences that border the maximum contiguous subsequence must have negative (or 0) sums (otherwise, we would include them). This observation also does not reduce the running time to below quadratic. However, a third observation, illustrated in Figure 5.7, does, and we formalize it with Theorem 5.3.

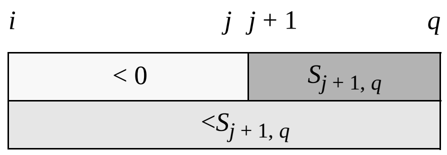

Figure 5.6 The subsequences used in Theorem 5.2

Theorem 5.3

For any i, let $A_{i,\,j}$ be the first sequence, with $S_{i,\,j} < 0$. Then, for any $i \le p \le j$ and $p \le q$, $A_{p,\,q}$ either is not a maximum contiguous subsequence or is equal to an already seen maximum contiguous subsequence.

Proof

If $p = i$, then Theorem 5.2 applies. Otherwise, as in Theorem 5.2, we have $S_{i,q} = S_{i,p-1} + S_{p,q}$. Since j is the lowest index for which $S_{i,j} < 0$, it follows that $S_{i,p-1} \geq 0$. Thus $S_{p,q} \leq S_{i,q}$. If $q > j$ (shown on the left-hand side in Figure 5.7), then Theorem 5.2 implies that $A_{i,q}$ is not a maximum contiguous subsequence, so neither is $A_{p,q}$. Otherwise, as shown on the right-hand side in Figure 5.7, the subsequence $A_{p,q}$ has a sum equal to, at most, that of the already seen subsequence $A_{i,q}$.

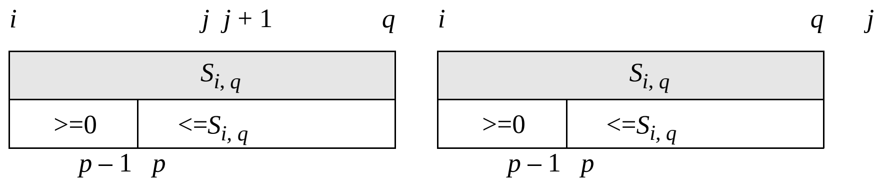

Figure 5.7 The subsequences used in Theorem 5.3. The sequence from *p* to *q* has a sum that is, at most, that of the subsequence from *i* to *q*. On the left-hand side the sequence from *i* to *q* is itself not the maximum (by Theorem 5.2). On the right-hand side, the sequence from *i* to *q* has already been seen

BOOK.mkr Page 255 Wednesday, March 14, 2001 1:11 PM

```
/**
 * Linear maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */
public static int maximumSubsequenceSum( int [ ] a )
{
    int maxSum = 0;
    int thisSum = 0;

    for( int i = 0, j = 0; j < a.length; j++ )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
        {
            maxSum = thisSum;
            seqStart = i;
            seqEnd   = j;
        }
        else if( thisSum < 0 )
        {
            i = j + 1;
            thisSum = 0;
        }
    }

    return maxSum;
}
```

Figure 5.8 A linear maximum contiguous subsequence sum algorithm

If we detect a negative sum, we can move *i* all the way past *j*.

Theorem 5.3 tells us that, when a negative subsequence is detected, not only can we `break` the inner loop; but we also can advance `i` to `j+1`. Figure 5.8 shows that we can rewrite the algorithm using only a single loop. Clearly, the running time of this algorithm is linear: At each step in the loop, we advance `j`, so the loop iterates at most N times. The correctness of this algorithm is much less obvious than for the previous algorithms, which is typical. That is, algorithms that use the structure of a problem to beat an exhaustive search generally require some sort of correctness proof. We proved that the algorithm (although not the resulting Java program) is correct using a short mathematical argument. The purpose is not to make the discussion entirely mathematical, but rather to give a flavor of the techniques that might be required in advanced work.

If an algorithm is complex, a correctness proof is required.

5.4 General Big-Oh Rules

Now that we have the basic ideas of algorithm analysis, we can adopt a slightly more formal approach. In this section we outline the general rules for using Big-Oh notation. Although we use Big-Oh notation almost exclusively throughout this text, we also define three other types of algorithm notation that are related to Big-Oh and used occasionally later on in the text.

DEFINITION: (Big-Oh) $T(N)$ is $O(F(N))$ if there are positive constants c and N_0 such that $T(N) \le cF(N)$ when $N \ge N_0$.

DEFINITION: (Big-Omega) $T(N)$ is $\Omega(F(N))$ if there are positive constants c and N_0 such that $T(N) \ge cF(N)$ when $N \ge N_0$.

DEFINITION: (Big-Theta) $T(N)$ is $\Theta(F(N))$ if and only if $T(N)$ is $O(F(N))$ and $T(N)$ is $\Omega(F(N))$.

DEFINITION: (Little-Oh) $T(N)$ is $o(F(N))$ if and only if $T(N)$ is $O(F(N))$ and $T(N)$ is not $\Theta(F(N))$.

The first definition, *Big-Oh* notation, states that there is a point N_0 such that for all values of N that are past this point, $T(N)$ is bounded by some multiple of $F(N)$. This is the sufficiently large N mentioned earlier. Thus, if the running time $T(N)$ of an algorithm is $O(N^2)$, then, ignoring constants, we are guaranteeing that at some point we can bound the running time by a quadratic function. Notice that if the true running time is linear, then the statement that the running time is $O(N^2)$ is technically correct because the inequality holds. However, $O(N)$ would be the more precise claim.

Big-Oh is similar to less than or equal to, when growth rates are being considered.

If we use the traditional inequality operators to compare growth rates, then the first definition says that the growth rate of $T(N)$ is less than or equal to that of $F(N)$.

Big-Omega is similar to greater than or equal to, when growth rates are being considered.

The second definition, $T(N) = \Omega(F(N))$, called *Big-Omega*, says that the growth rate of $T(N)$ is greater than or equal to that of $F(N)$. For instance, we might say that any algorithm that works by examining every possible subsequence in the maximum subsequence sum problem must take $\Omega(N^2)$ time because a quadratic number of subsequences are possible. This is a lower-bound argument that is used in more advanced analysis. Later in the text, we will see one example of this argument and demonstrate that any general-purpose sorting algorithm requires $\Omega(N \log N)$ time.

Big-Theta is similar to equal to, when growth rates are being considered.

The third definition, $T(N) = \Theta(F(N))$, called *Big-Theta*, says that the growth rate of $T(N)$ equals the growth rate of $F(N)$. For instance, the maximum subsequence algorithm shown in Figure 5.5 runs in $\Theta(N^2)$ time. In other words, the running time is bounded by a quadratic function, and that this bound cannot be improved because it is also lower-bounded by another quadratic function. When we use Big-Theta notation, we are providing not only an upper bound on an algorithm but also assurances that the analysis that leads to the upper bound is as good (tight) as possible. In spite of the additional precision offered by Big-Theta, however, Big-Oh is more commonly used, except by researchers in the algorithm analysis field.

Little-Oh is similar to less than, when growth rates are being considered.

The final definition, $T(N) = o(F(N))$, called *Little-Oh*, says that the growth rate of $T(N)$ is strictly less than the growth rate of $F(N)$. This function is different from Big-Oh because Big-Oh allows the possibility that the growth rates are the same. For instance, if the running time of an algorithm is $o(N^2)$, then it is guaranteed to be growing at a slower rate than quadratic (that is, it is a *subquadratic algorithm*). Thus a bound of $o(N^2)$ is a better bound than $\Theta(N^2)$. Figure 5.9 summarizes these four definitions.

Throw out leading constants, lower-order terms, and relational symbols when using Big-Oh.

A couple of stylistic notes are in order. First, including constants or low-order terms inside a Big-Oh is bad style. Do not say $T(N) = O(2N^2)$ or $T(N) = O(N^2 + N)$. In both cases, the correct form is $T(N) = O(N^2)$. Second, in any analysis that requires a Big-Oh answer, all sorts of shortcuts are possible. Lower-order terms, leading constants, and relational symbols are all thrown away.

Now that the mathematics have formalized, we can relate it to the analysis of algorithms. The most basic rule is that *the running time of a loop is at most the running time of the statements inside the loop (including tests) times the number of iterations*. As shown earlier, the initialization and testing of the loop condition is usually no more dominant than are the statements encompassing the body of the loop.

A *worst-case bound* is a guarantee over all inputs of some size.

The running time of statements inside a group of nested loops is the running time of the statements (including tests in the innermost loop) multiplied by the sizes of all the loops. The running time of a sequence of consecutive loops is equal to the running time of the dominant loop. The time difference between a nested loop in which both indices run from 1 to N and two consecutive loops that are not nested but run over the same indices is the same as the space difference between a two-dimensional array and two one-dimensional arrays. The first case is quadratic. The second case is linear because $N+N$ is $2N$, which is still $O(N)$. Occasionally, this simple rule can overestimate the running time, but in most cases it does not. Even if it does, Big-Oh does not guarantee an exact asymptotic answer — just an upper bound.

Mathematical Expression	Relative Rates of Growth
$T(N) = O(F(N))$	Growth of $T(N)$ is $\leq$ growth of $F(N)$.
$T(N) = \Omega(F(N))$	Growth of $T(N)$ is $\geq$ growth of $F(N)$.
$T(N) = \Theta(F(N))$	Growth of $T(N)$ is $=$ growth of $F(N)$.

Figure 5.9 Meanings of the various growth functions

Mathematical Expression	Relative Rates of Growth
$T(N) = o(F(N))$	Growth of $T(N)$ is < growth of $F(N)$.

Figure 5.9 Meanings of the various growth functions

	Figure 5.4	Figure 5.5	Figure 7.18	Figure 5.8
N	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0.000009	0.000004	0.000006	0.000003
100	0.002580	0.000109	0.000045	0.000006
1,000	2.281013	0.010203	0.000485	0.000031
10,000	NA	1.2329	0.005712	0.000317
100,000	NA	135	0.064618	0.003206

Figure 5.10 Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms

In an *average-case bound*, the running time is measured as an average over all of the possible inputs of size *N*.

The analyses we have performed thus far involved use of a *worst-case bound*, which is a guarantees over all inputs of some size. Another form of analysis is the *average-case bound*, in which the running time is measured as an average over all of the possible inputs of size *N*. The average might differ from the worst case if, for example, a conditional statement that depends on the particular input causes an early exit from a loop. We discuss average-case bounds in more detail in Section 5.8. For now, simply note that, the fact that one algorithm has a better worst-case bound than another algorithm implies nothing about their relative average-case bounds. However, in many cases average-case and worst-case bounds are closely correlated. When they are not, the bounds are treated separately.

The last Big-Oh item we examine is how the running time grows for each type of curve, as illustrated in Figures 5.1 and 5.2. We want a more quantitative answer to this question: If an algorithm takes $T(N)$ time to solve a problem of size N, how long does it take to solve a larger problem? For instance, how long does it take to solve a problem when there is 10 times as much input? The answers are shown in Figure 5.10. However, we want to answer the question without running the program and hope our analytical answers will agree with the observed behavior.

We begin by examining the cubic algorithm. We assume that the running time is reasonably approximated by $T(N) = cN^3$. Consequently, $T(10N) = c(10N)^3$. Mathematical manipulation yields

$$T(10N) = 1000cN^3 = 1000T(N).$$

Thus the running time of a cubic program increases by a factor of 1,000 (assuming N is sufficiently large) when the amount of input is increased by a factor of 10. This relationship is roughly confirmed by the increase in running time from $N = 100$ to 1,000 shown in Figure 5.10. Recall that we do not expect an exact answer — just a reasonable approximation. We would also expect that for $N = 10{,}000$, the running time would increase another 1,000-fold. The result would be that a cubic algorithm requires roughly 35 minutes of computation time. In general, if the amount of the input increases by a factor of f, then the cubic algorithm's running time increases by a factor of f^3.

If the size of the input increases by a factor of f, the running time of a cubic program increases by a factor of roughly f^3.

If the size of the input increases by a factor of *f*, the running time of a quadratic program increases by a factor of roughly f^2.

We can perform similar calculations for quadratic and linear algorithms. For the quadratic algorithm, we assume that $T(N) = cN^2$. It follows that $T(10N) = c(10N)^2$. When we expand, we obtain

$$T(10N) = 100cN^2 = 100T(N).$$

So when the input size increases by a factor of 10, the running time of a quadratic program increases by a factor of approximately 100. This relationship is also confirmed in Figure 5.10. In general, an *f*-fold increase in input size yields an f^2-fold increase in running time for a quadratic algorithm.

If the size of the input increases by a factor of *f*, then the running time of a linear program also increases by a factor of *f*. This is the preferred running time for an algorithm.

Finally, for a linear algorithm, a similar calculation shows that a 10-fold increase in input size results in a 10-fold increase in running time. Again, this relationship has been confirmed experimentally in Figure 5.10. Note, however, that for a linear program, the term *sufficiently large* means a somewhat higher input size than for the other programs. The reason is that of the overhead of 0.000003 sec that is used in all cases. For a linear program, this term is still significant for moderate input sizes.

The analysis used here does not work when there are logarithmic terms. When an $O(N \log N)$ algorithm is presented with 10 times as much input, the running time increases by a factor slightly larger than ten. Specifically, we have $T(10N) = c(10N)\log(10N)$. When we expand we obtain

$$T(10N) = 10cN\log(10N) = 10cN\log N + 10cN\log 10 = 10T(N) + c'N.$$

Here $c' = 10c\log 10$. As N gets very large, the ratio $T(10N)/T(N)$ gets closer and closer to 10 because $c'N/T(N) \approx (10\log 10)/\log N$ gets smaller and smaller with increasing N. Consequently, if the algorithm is competitive with a linear algorithm for very large N, it is likely to remain competitive for slightly larger N.

Does all this mean that quadratic and cubic algorithms are useless? The answer is no. In some cases, the most efficient algorithms known are quadratic or cubic. In others, the most efficient algorithm is even worse (exponential). Furthermore, when the amount of input is small, any algorithm will do. Frequently the algorithms that are not asymptotically efficient are nonetheless easy to program. For small inputs, that is the way to go. Finally, a good way to test a complex linear algorithm is to compare its output with an exhaustive search algorithm. In Section 5.8 we discuss some other limitations of the Big-Oh model.

5.5 The Logarithm

The list of typical growth rate functions includes several entries containing the logarithm. A *logarithm* is the exponent that indicates the power to which a number (the base) is raised to produce a given number. In this section we look in more detail at the mathematics behind the logarithm. In Section 5.6 we show its use in a simple algorithm.

We begin with the formal definition and then follow with more intuitive viewpoints.

The *logarithm* of N (to the base 2) is the value X such that 2 raised to the power of X equals N. By default, the base of the logarithm is 2.

DEFINITION: For any $B, N > 0$, $\log_B N = K$ if $B^K = N$.

In this definition, B is the base of the logarithm. In computer science, when the base is omitted, it defaults to 2, which is natural for several reasons, as we show later in the chapter. We will prove one mathematical theorem, Theorem 5.4, to show that, as far as Big-Oh notation is concerned, the base is unimportant, and also to show how relations that involve logarithms can be derived.

Theorem 5.4

The base does not matter. For any constant $B > 1$, $\log_B N = O(\log N)$.

Proof

Let $\log_B N = K$. Then $B^K = N$. Let $C = \log B$. Then $2^C = B$. Thus $B^K = (2^C)^K = N$. Hence, we have $2^{CK} = N$, which implies that $\log N = CK = C \log_B N$. Therefore $\log_B N = (\log N)/(\log B)$, thus completing the proof.

In the rest of the text, we use base 2 logarithms exclusively. An important fact about the logarithm is that it grows slowly. Because $2^{10} = 1{,}024$, log 1,024 = 10. Additional calculations show that the logarithm of 1,000,000 is roughly 20, and the logarithm of 1,000,000,000 is only 30. Consequently, performance of an $O(N \log N)$ algorithm is much closer to a linear $O(N)$ algorithm than to a quadratic $O(N^2)$ algorithm for even moderately large amounts of input. Before we look at a realistic algorithm whose running time includes the logarithm, let us look at a few examples of how the logarithm comes into play.

BITS IN A BINARY NUMBER
How many bits are required to represent N consecutive integers?

A 16-bit `short` integer represents the 65,536 integers in the range –32,768 to 32,767. In general, B bits are sufficient to represent 2^B different integers. Thus the number of bits B required to represent N consecutive integers satisfies the equation $2^B \geq N$. Hence, we obtain $B \geq \log N$, so the minimum number of bits is $\lceil \log N \rceil$. (Here $\lceil X \rceil$ is the ceiling function and represents the smallest integer that is at least as large as X. The corresponding floor function $\lfloor X \rfloor$ represents the largest integer that is at least as small as X.)

The number of bits required to represent numbers is logarithmic.

REPEATED DOUBLING
Starting from $X = 1$, how many times should X be doubled before it is at least as large as N?

Suppose we start with \$1 and double it every year. How long would it take to save a million dollars? In this case, after 1 yr we would have \$2; after 2 yr, \$4; after 3 yr, \$8, and so on. In general, after K years we would have 2^K dollars, so we want to find the smallest K satisfying $2^K \geq N$. This is the same equation as before, so $K = \lceil \log N \rceil$. After 20 yr, we would have over a million dollars. The *repeated doubling principle* holds that, starting from 1, we can repeatedly double only $\lceil \log N \rceil$ times until we reach N.

The *repeated doubling principle* holds that, starting at 1, we can repeatedly double only logarithmically many times until we reach *N*.

REPEATED HALVING
Starting from $X = N$, if N is repeatedly halved, how many iterations must be applied to make N smaller than or equal to 1?

If the division rounds up to the nearest integer (or is real, not integer, division), we have the same problem as with repeated doubling, except that we are going in the opposite direction. Once again the answer is $\lceil \log N \rceil$ iterations. If the division rounds down, the answer is $\lfloor \log N \rfloor$. We can show the difference by starting with $X = 3$. Two divisions are necessary, unless the division rounds down, in which case only one is needed.

The *repeated halving principle* holds that, starting at *N*, we can halve only logarithmically many times. This process is used to obtain logarithmic routines for searching.

Many of the algorithms examined in this text will have logarithms, introduced because of the *repeated halving principle*, which holds that, starting at *N*, we can halve only logarithmically many times. In other words, an algorithm is $O(\log N)$ if it takes constant ($O(1)$) time to cut the problem size by a constant fraction (which is usually $1/2$). This condition follows directly from the fact that there will be $O(\log N)$ iterations of the loop. Any constant fraction will do because the fraction is reflected in the base of the logarithm, and Theorem 5.4 tells us that the base does not matter.

The *N*th *harmonic number* is the sum of the reciprocals of the first *N* positive integers. The growth rate of the harmonic number is logarithmic.

All of the remaining occurrences of logarithms are introduced (either directly or indirectly) by applying Theorem 5.5. This theorem concerns the *N*th *harmonic number*, which is the sum of the reciprocals of the first *N* positive integers, and states that the *N*th harmonic number, H_N, satisfies $H_N = \Theta(\log N)$. The proof uses calculus, but you do not need to understand the proof to use the theorem.

Theorem 5.5 *Let* $H_N = \sum_{i=1}^{N} 1/i$. *Then* $H_N = \Theta(\log N)$. *A more precise estimate is* $\ln N + 0.577$.

Proof *The intuition of the proof is that a discrete sum is well approximated by the (continuous) integral. The proof uses a construction to show that the sum H_N can be bounded above and below by $\int \frac{dx}{x}$, with appropriate limits. Details are left as Exercise 5.18.*

The next section shows how the repeated halving principle leads to an efficient searching algorithm.

5.6 Static Searching Problem

An important use of computers is looking up data. If the data are not allowed to change (e.g., it is stored on a CD-ROM), we say that the data are static. A *static search* accesses data that are never altered. The static searching problem is naturally formulated as follows.

STATIC SEARCHING PROBLEM

Given an integer X and an array A, return the position of X in A or an indication that it is not present. If X occurs more than once, return any occurrence. The array A is never altered.

An example of static searching is looking up a person in the telephone book. The efficiency of a static searching algorithm depends on whether the array being searched is sorted. In the case of the telephone book, searching by name is fast, but searching by phone number is hopeless (for humans). In this section, we examine some solutions to the static searching problem.

5.6.1 Sequential Search

A *sequential search* steps through the data sequentially until a match is found.

When the input array is not sorted, we have little choice but to do a linear *sequential search*, that steps through the array sequentially until a match is found. The complexity of the algorithm is analyzed in three ways. First, we provide the cost of an unsuccessful search. Then, we give the worst-case cost of a successful search. Finally, we find the average cost of a successful search. Analyzing successful and unsuccessful searches separately is typical. Unsuccessful searches usually are more time consuming than are successful searches (just think about the last time you lost something in your house). For sequential searching, the analysis is straightforward.

A sequential search is linear.

An unsuccessful search requires the examination of every item in the array, so the time will be $O(N)$. In the worst case, a successful search, too, requires the examination of every item in the array because we might not find a match until the last item. Thus the worst-case running time for a successful search is also linear. On average, however, we search only half of the array. That is, for every successful search in position i, there is a corresponding successful search in position $N - 1 - i$ (assuming we start numbering from 0). However, $N/2$ is still $O(N)$. As mentioned earlier in the chapter, all these Big-Oh terms should correctly be Big-Theta terms. However, the use of Big-Oh is more popular.

5.6.2 Binary Search

If the input array has been sorted, we have an alternative to the sequential search, the *binary search*, which is performed from the middle of the array rather than the end. We keep track of `low` and `high`, which delimit the portion of the array in which an item, if present, must reside. Initially, the range is from 0 to $N-1$. If `low` is larger than `high`, we know that the item is not present, so we return `NOT_FOUND`. Otherwise, we let `mid` be the halfway point of the range (rounding down if the range has an even number of elements) and compare the item we are searching for with the item in position `mid`. If we find a match, we are done and can return. If the item we are searching for is less than the item in position `mid`, then it must reside in the range `low` to `mid-1`. If it is greater, then it must reside in the range `mid+1` to `high`. In Figure 5.11, lines 17 to 20 alter the possible range, essentially cutting it in half. By the repeated halving principle, we know that the number of iterations will be $O(\log N)$.

If the input array is sorted, we can use the *binary search*, which we perform from the middle of the array rather than the end.

For an unsuccessful search, the number of iterations in the loop is $\lfloor \log N \rfloor + 1$. The reason is that we halve the range in each iteration (rounding down if the range has an odd number of elements); we add 1 because the final range encompasses zero elements. For a successful search, the worst case is $\lfloor \log N \rfloor$ iterations because in the worst case we get down to a range of only one element. The average case is only one iteration better because half of the elements require the worst case for their search, a quarter of the elements save one iteration, and only one in 2^i elements will save i iterations from the worst case. The mathematics involves computing the weighted average by calculating the

The *binary search* is logarithmic because the search range is halved in each iteration.

sum of a finite series. The bottom line, however, is that the running time for each search is $O(\log N)$. In Exercise 5.20 you are asked to complete the calculation.

For reasonably large values of N, the binary search outperforms the sequential search. For instance, if N is 1,000, then on average a successful sequential search requires about 500 comparisons. The average binary search, using the previous formula, requires $\lfloor \log N \rfloor - 1$, or eight iterations for a successful search. Each iteration uses 1.5 comparisons on average (sometimes 1; other times, 2), so the total is 12 comparisons for a successful search. The binary search wins by even more in the worst case or when searches are unsuccessful.

```java
/**
 * Performs the standard binary search
 * using two comparisons per level.
 * @return index where item is found, or NOT_FOUND.
 */
public static int binarySearch( Comparable [ ] a,
                                Comparable x )
{
    int low = 0;
    int high = a.length - 1;
    int mid;

    while( low <= high )
    {
        mid = ( low + high ) / 2;

        if( a[ mid ].compareTo( x ) < 0 )
            low = mid + 1;
        else if( a[ mid ].compareTo( x ) > 0 )
            high = mid - 1;
        else
            return mid;
    }

    return NOT_FOUND;  // - 1
}
```

Figure 5.11 Basic binary search that uses three-way comparisons

Optimizing the binary search can cut the number of comparisons roughly in half.

If we want to make the binary search even faster, we need to make the inner loop tighter. A possible strategy is to remove the (implicit) test for a successful search from that inner loop and shrink the range down to one item in all cases. Then we can use a single test outside of the loop to determine if the item is in the array or cannot be found, as shown in Figure 5.12 (page 272). If the item we are searching for in Figure 5.12 is not larger than the item in the `mid` position, then it is in the range that includes the `mid` position. When we break the loop, the subrange is 1, and we can test to see whether we have a match.

In the revised algorithm, the number of iterations is always $\lfloor \log N \rfloor$ because we always shrink the range in half, possibly by rounding down. Thus, the number of comparisons used is always $\lfloor \log N \rfloor + 1$.

Binary search is surprisingly tricky to code. Exercise 5.6 illustrates some common errors.

Notice that for small N, such as values smaller than 6, the binary search might not be worth using. It uses roughly the same number of comparisons for a typical successful search, but it has the overhead of line 18 in each iteration. Indeed, the last few iterations of the binary search progress slowly. One can adopt a hybrid strategy in which the binary search loop terminates when the range is small and applies a sequential scan to finish. Similarly, people search a phone book nonsequentially. Once they have narrowed the range to a column, they perform a sequential scan. The scan of a telephone book is not sequential, but it also is not a binary search. Instead it is more like the algorithm discussed in the next section.

5.6.3 Interpolation Search

The binary search is very fast at searching a sorted static array. In fact, it is so fast that we would rarely use anything else. A static searching method that is sometimes faster, however, is an *interpolation search*, which has better Big-Oh performance on aveage than binary earch but has limited practicality and a bad worst case. For an interpolation search to be practical, two assumptions must be satisfied:

```
    /**
     * Performs the standard binary search
     * using one comparison per level.
     * @return index where item is found of NOT_FOUND.
     */
    public static int binarySearch( Comparable [ ] a,
                                    Comparable x )
    {
        if( a.length == 0 )
            return NOT_FOUND;

        int low = 0;
        int high = a.length - 1;
        int mid;

        while( low < high )
        {
            mid = ( low + high ) / 2;

            if( a[ mid ].compareTo( x ) < 0 )
                low = mid + 1;
            else
                high = mid;
        }

        if( a[ low ].compareTo( x ) == 0 )
            return low;

        return NOT_FOUND;
    }
}
```

Figure 5.12 Binary search using two-way comparisons

1. Each access must be very expensive compared to a typical instruction. For example, the array might be on a disk instead of in memory, and each comparison requires a disk access.
2. The data must not only be sorted; it must also be fairly uniformly distributed. For example, a phone book is fairly uniformly distributed. If the input items are {1, 2, 4, 8, 16, …}, the distribution is not uniform.

These assumptions are quite restrictive, so you might never use an interpolation search. But it is interesting to see that there is more than one way to solve a problem and that no algorithm, not even the classic binary search, is the best in all situations.

The interpolation search requires that we spend more time to make an accurate guess regarding where the item might be. The binary search always uses the midpoint. However, searching for *Hank Aaron* in the middle of the phone book would be silly; somewhere near the start clearly would be more appropriate. Thus, instead of `mid`, we use `next` to indicate the next item that we will try to access.

Here's an example of what might work well. Suppose that the range contains 1,000 items, the low item in the range is 1,000, the high item in the range is 1,000,000, and we are searching for an item of value 12,000. If the items are uniformly distributed, then we expect to find a match somewhere near the twelfth item. The applicable formula is

$$next = low + \left\lceil \frac{x - a[low]}{a[high] - a[low]} \times (high - low - 1) \right\rceil .$$

The subtraction of 1 is a technical adjustment that has been shown to perform well in practice. Clearly, this calculation is more costly then the binary search calculation. It involves an extra division (the division by 2 in the binary search is really just a bit shift, just as dividing by 10 is easy for humans), multiplication, and four subtractions. These calculations need to be done using floating-point operations. One iteration may be slower than the complete binary search. However, if the cost of these calculations is insignificant when compared to the cost of accessing an item, speed is immaterial; we care only about the number of iterations.

Interpolation search has a better Big-Oh bound on average than does binary search, but has limited practicality and a bad worst case.

In the worst case, where data is not uniformly distributed, the running time could be linear and every item might be examined. In Exercise 5.19 you are asked to construct such a case. However, if we assume that the items are reasonably distributed, as with a phone book, the average number of comparisons has been shown to be $O(\log\log N)$. In other words, we apply the logarithm twice in succession. For $N = 4$ billion, $\log N$ is about 32 and $\log\log N$ is roughly 5. Of course, there are some hidden constants in the Big-Oh notation, but the extra logarithm can lower the number of iterations considerably, so long as a bad case does not crop up. Proving the result rigorously, however, is quite complicated.

5.7 Checking an Algorithm Analysis

Once we have performed an algorithm analysis, we want to determine whether it is correct and as good as we can possibly make it. One way to do this is to code the program and see if the empirically observed running time matches the running time predicted by the analysis.

When N increases by a factor of 10, the running time goes up by a factor of ten for linear programs, 100 for quadratic programs, and 1,000 for cubic programs. Programs that run in $O(N \log N)$ take slightly more than 10 times as long to run under the same circumstances. These increases can be hard to spot if the lower-order terms have relatively large coefficients and N is not large enough. An example is the jump from $N = 10$ to $N = 100$ in the running time for the various implementations of the maximum contiguous subsequence sum problem. Differentiating linear programs from $O(N \log N)$ programs, based purely on empirical evidence, also can be very difficult.

Another commonly used trick to verify that some program is $O(F(N))$ is to compute the values $T(N)/F(N)$ for a range of N (usually spaced out by factors of two), where $T(N)$ is the empirically observed running time. If $F(N)$ is a tight answer for the running time, then the computed values converge to a positive constant. If $F(N)$ is an overestimate, the values converge to zero. If $F(N)$ is an underestimate, and hence wrong, the values diverge.

As an example, suppose that we write a program to perform N random searches using the binary search algorithm. Since each search is logarithmic, we expect the total running time of the program to be $O(N \log N)$. Figure 5.13 shows the actual observed running time for the routine for various input sizes on a real (but extremely slow) computer. The last column is most likely the converging column and thus confirms our analysis, whereas the increasing numbers for T/N suggest that $O(N)$ is an underestimate and the quickly decreasing values for T/N^2 suggest that $O(N^2)$ is an overestimate.

Note in particular that we do not have definitive convergence. One problem is that the clock that we used to time the program ticks only every 10 ms. Note also that there is not a great difference between $O(N)$ and $O(N \log N)$. Certainly an $O(N \log N)$ algorithm is much closer to being linear than being quadratic. Finally, note that the machine in this example has enough memory to store 640,000 objects (in the case of this experiment, integers). If your machine does not have this much available memory, then you will not be able to reproduce similar results.

N	**CPU Time T (milliseconds)**	T/N	T/N^2	$T/(N \log N)$
10,000	100	0.01000000	0.00000100	0.00075257
20,000	200	0.01000000	0.00000050	0.00069990
40,000	440	0.01100000	0.00000027	0.00071953
80,000	930	0.01162500	0.00000015	0.00071373
160,000	1,960	0.01225000	0.00000008	0.00070860
320,000	4,170	0.01303125	0.00000004	0.00071257
640,000	8,770	0.01370313	0.00000002	0.00071046

Figure 5.13 Empirical running time for *N* binary searches in an *N*-item array

5.8 Limitations of Big-Oh Analysis

Big-Oh analysis is a very effective tool, but it does have limitations. As already mentioned, its use is not appropriate for small amounts of input. For small amounts of input, use the simplest algorithm. Also, for a particular algorithm, the constant implied by the Big-Oh may be too large to be practical. For example, if one algorithm's running time is governed by the formula $2N \log N$ and another has a running time of $1000N$, then the first algorithm would most likely be better, even though its growth rate is larger. Large constants can come into play when an algorithm is excessively complex. They also come into play because our analysis disregards constants and thus cannot differentiate between things like memory access (which is cheap) and disk access (which typically is many thousand times more expensive). Our analysis assumes infinite memory, but in applications involving large data sets, lack of sufficient memory can be a severe problem.

Worse case is sometimes uncommon and can be safely ignored. At other times, it is very common and cannot be ignored.

Sometimes, even when constants and lower-order terms are considered, the analysis is shown empirically to be an overestimate. In this case, the analysis needs to be tightened (usually by a clever observation). Or the average-case running time bound may be significantly less than the worst-case running time bound, and so no improvement in the bound is possible. For many complicated algorithms the worst-case bound is achievable by some bad input, but in practice it is usually an overestimate. Two examples are the sorting algorithms Shellsort and quicksort (both described in Chapter 8).

Average-case analysis is almost always much more difficult than worst-case analysis.

However, worst-case bounds are usually easier to obtain than their average-case counterparts. For example, a mathematical analysis of the average-case running time of Shellsort has not been obtained. Sometimes, merely defining what *average* means is difficult. We use a worst-case analysis because it is expedient and also because, in most instances, the worst-case analysis is very meaningful. In the course of performing the analysis, we frequently can tell whether it will apply to the average case.

Summary

In this chapter we introduced algorithm analysis and showed that algorithmic decisions generally influence the running time of a program much more than programming tricks do. We also showed the huge difference between the running times for quadratic and linear programs and illustrated that cubic algorithms are, for the most part, unsatisfactory. We examined an algorithm that could be viewed as the basis for our first data structure. The binary search efficiently supports static operations (i.e., searching but not updating), thereby providing a logarithmic worst-case search. Later in the text we examine dynamic data structures that efficiently support updates (both insertion and deletion).

In Chapter 6 we discuss some of the data structures and algorithms included in Java's Collections API. We also look at some applications of data structures and discuss their efficiency.

Objects of the Game

average-case bound Measurement of running time as an average over all the possible inputs of size *N*. (260)

Big-Oh The notation used to capture the most dominant term in a function; it is similar to less than or equal to when growth rates are being considered. (239)

Big-Omega The notation similar to greater than or equal to when growth rates are being considered. (257)

Big-Theta The notation similar to equal to when growth rates are being considered. (258)

binary search The search method used if the input array has been sorted and is performed from the middle rather than the end. The binary search is logarithmic because the search range is halved in each iteration. (269)

harmonic numbers The *N*th harmonic number is the sum of the reciprocals of the first *N* positive integers. The growth rate of the harmonic numbers is logarithmic. (266)

interpolation search A static searching algorithm that has better Big-Oh performance on average than binary search but has limited practicality and a bad worst case. (274)

linear time algorithm An algorithm that causes the running time to grow as $O(N)$. If the size of the input increases by a factor of *f*, then the running time also increases by a factor of *f*. It is the preferred running time for an algorithm. (262)

Little-Oh The notation similar to less than when growth rates are being considered. (258)

logarithm The exponent that indicates the power to which a number is raised to produce a given number. For example, the logarithm of *N* (to the base 2) is the value *X* such that 2 raised to the power of *X* equals *N*. (263)

repeated-doubling principle Holds that, starting at 1, repeated doubling can occur only logarithmically many times until we reach *N*. (265)

repeated-halving principle Holds that, starting at *N*, repeated halving can occur only logarithmically many times until we reach 1. This process is used to obtain logarithmic routines for searching. (266)

sequential search A linear search method that steps through an array until a match is found. (268)

static search Accesses data that is never altered. (267)

subquadratic An algorithm whose running time is strictly slower than quadratic, which can be written as $o(N^2)$. (258)

worst-case bound A guarantee over all inputs of some size. (259)

Common Errors

1. For nested loops, the total time is affected by the product of the loop sizes. For consecutive loops, it is not.
2. Do not just blindly count the number of loops. A pair of nested loops that each run from 1 to N^2 accounts for $O(N^4)$ time.

3. Do not write expressions such as $O(2N^2)$ or $O(N^2 + N)$. Only the dominant term, with the leading constant removed, is needed.
4. Use equalities with Big-Oh, Big-Omega, and so on. Writing that the running time is $> O(N^2)$ makes no sense because Big-Oh is an upper bound. Do not write that the running time is $< O(N^2)$; if the intention is to say that the running time is strictly less than quadratic, use Little-Oh notation.
5. Use Big-Omega, not Big-Oh, to express a lower bound.
6. Use the logarithm to describe the running time for a problem solved by halving its size in constant time. If it takes more than constant time to halve the problem, the logarithm does not apply.
7. The base (if it is a constant) of the logarithm is irrelevant for the purposes of Big-Oh. To include it is an error.

On the Internet

The three maximum contiguous subsequence sum algorithms, as well as a fourth taken from Section 7.5, are available, along with a `main` that conducts the timing tests.

MaxSumTest.java	Contains four algorithms for the maximum subsequence sum problem.
BinarySearch.java	Contains the binary search shown in Figure 5.11. The code in Figure 5.12 is not provided, but a similar version that is part of the Collections API

and is implemented in Figure 6.15 is in

Arrays.java as part of `weiss.util`.

Exercises

In Short

5.1. Balls are drawn from a box as specified in Theorem 5.1 in the combinations given in (a) – (d). What are the corresponding values of *i*, *j*, and *k*?

a. Red, 5, 6

b. Blue, 5, 6

c. Blue, 3, Red

d. 6, 5, Red

5.2. Why isn't an implementation based solely on Theorem 5.2 sufficient to obtain a subquadratic running time for the maximum contiguous subsequence sum problem?

5.3. Suppose $T_1(N) = O(F(N))$ and $T_2(N) = O(F(N))$. Which of the following are true:

a. $T_1(N) + T_2(N) = O(F(N))$

b. $T_1(N) - T_2(N) = O(F(N))$

c. $T_1(N) / T_2(N) = O(1)$

d. $T_1(N) = O(T_2(N))$

5.4. Group the following into equivalent Big-Oh functions:

$$x^2, \quad x, \quad x^2 + x, \quad x^2 - x, \quad \text{and} \quad (x^3 / (x - 1)).$$

5.5. Programs *A* and *B* are analyzed and are found to have worst-case running times no greater than $150N \log N$ and N^2, respectively. Answer the following questions, if possible.

a. Which program has the better guarantee on the running time for large values of *N* (*N* > 10,000)?

b. Which program has the better guarantee on the running time for small values of *N* (*N* < 100)?

c. Which program will run faster *on average* for *N* = 1,000?

d. Can program *B* will run faster than program *A* on *all* possible inputs?

5.6. For the binary search routine in Figure 5.11, show the consequences of the following replacement code fragments:

a. Line 13: using the test `low < high`

b. Line 15: assigning `mid = low + high / 2`

c. Line 18: assigning `low = mid`

d. Line 20: assigning `high = mid`

In Theory

5.7. For the typical algorithms that you use to perform calculations by hand, determine the running time to

a. Add two *N*-digit integers.

b. Multiply two *N*-digit integers.

c. Divide two *N*-digit integers.

5.8. In terms of N, what is the running time of the following algorithm to compute X^N:

```
public static double power( double x, int n )
{
    double result = 1.0;

    for( int i = 0; i < n; i++ )
        result *= x;
    return result;
}
```

5.9. Directly evaluate the triple summation that precedes Theorem 5.1. Verify that the answers are identical.

5.10. For the quadratic algorithm for the maximum contiguous subsequence sum problem, determine precisely how many times the innermost statement is executed.

5.11. An algorithm takes 0.5 ms for input size 100. How long will it take for input size 500 (assuming that low-order terms are negligible) if the running time is

a. linear.

b. $O(N \log N)$.

c. quadratic.

d. cubic.

5.12. An algorithm takes 0.5 ms for input size 100. How large a problem can be solved in 1 min (assuming that low-order terms are negligible) if the running time is

a. linear.

b. $O(N \log N)$.

c. quadratic.

d. cubic.

5.13. Complete Figure 5.10 with estimates for the running times that were too long to simulate. Interpolate the running times for all four algorithms and estimate the time required to compute the maximum contiguous subsequence sum of 10,000,000 numbers. What assumptions have you made?

5.14. Order the following functions by growth rate: N, $\sqrt{N}$, $N^{1.5}$, N^2, $N \log N$, $N \log\log N$, $N \log^2 N$, $N \log(N^2)$, $2/N$, 2^N, $2^{N/2}$, 37, N^3, and $N^2 \log N$. Indicate which functions grow at the same rate.

5.15. For each of the following program fragments,

a. give a Big-Oh analysis of the running time.

b. implement the code and run for several values of N.

c. compare your analysis with the actual running times.

```
// Fragment #1
for( int i = 0; i < n; i++ )
    sum++;

// Fragment #2
for( int i = 0; i < n; i += 2 )
    sum++;


// Fragment #3
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ )
        sum++;
```

```
// Fragment #4
for( int i = 0; i < n; i++ )
    sum++;
for( int j = 0; j < n; j++ )
    sum++;


// Fragment #5
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n * n; j++ )
        sum++;


// Fragment #6
for( int i = 0; i < n; i++ )
    for( int j = 0; j < i; j++ )
        sum++;


// Fragment #7
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n * n; j++ )
        for( int k = 0; k < j; k++ )
            sum++;
// Fragment #8
for( int i = 0; i < n; i = i * 2 )
    sum++;
```

5.16. Occasionally, multiplying the sizes of nested loops can give an overestimate for the Big-Oh running time. This result happens when an innermost loop is infrequently executed. Repeat Exercise 5.15 for the following program fragment:

```
for( int i = 1; i <= n; i++ )
    for( int j = 1; j <= i * i; j++ )
        if( j % i == 0 )
            for( int k = 0; k < j; k++ )
                sum++;
```

5.17. In a court case, a judge cited a city for contempt and ordered a fine of \$2 for the first day. Each subsequent day, until the city followed the judge's order, the fine was squared (that is, the fine progressed as follows: \$2, \$4, \$16, \$256, \$65536, . . .).

a. What would be the fine on day N?

b. How many days would it take for the fine to reach D dollars (a Big-Oh answer will do)?

5.18. Prove Theorem 5.5. *Hint*: Show that $\sum_{2}^{N} \frac{1}{i} < \int_{1}^{N} \frac{dx}{x}$. Then show a similar lower bound.

5.19. Construct an example whereby an interpolation search examines every element in the input array.

5.20. Analyze the cost of an average successful search for the binary search algorithm in Figure 5.11.

In Practice

5.21. Give an efficient algorithm to determine whether an integer i exists such that $A_i = i$ in an array of increasing integers. What is the running time of your algorithm?

5.22. A prime number has no factors besides 1 and itself. Do the following:

a. Write a program to determine if a positive integer N is prime. In terms of N, what is the worst-case running time of your program?

b. Let B equal the number of bits in the binary representation of N. What is the value of B?

c. In terms of B, what is the worst-case running time of your program?

d. Compare the running times to determine if a 20-bit number and a 40-bit number are prime.

5.23. An important problem in numerical analysis is to find a solution to the equation $F(X) = 0$ for some arbitrary *F*. If the function is continuous and has two points *low* and *high* such that $F(low)$ and $F(high)$ have opposite signs, then a root must exist between *low* and *high* and can be found by either a binary search or an interpolation search. Write a function that takes as parameters *F*, *low*, and *high* and solves for a zero. What must you do to ensure termination?

5.24. A majority element in an array *A* of size *N* is an element that appears more than $N/2$ times (thus there is at most one such element). For example, the array

3, 3, 4, 2, 4, 4, 2, 4, 4

has a majority element (4), whereas the array
3, 3, 4, 2, 4, 4, 2, 4

does not. Give an algorithm to find a majority element if one exists, or reports that one does not. What is the running time of your algorithm? (*Hint*: There is an $O(N)$ solution.)

5.25. The input is an $N \times N$ matrix of numbers that is already in memory. Each individual row is increasing from left to right. Each individual column is increasing from top to bottom. Give an $O(N)$ worst-case algorithm that decides if a number *X* is in the matrix.

5.26. Design efficient algorithms that take an array of positive numbers `a`, and determine

a. the maximum value of `a[j]+a[i]`, for `j` ≥ `i`.

b. the maximum value of `a[j]-a[i]`, for `j` ≥ `i`.

c. the maximum value of `a[j]*a[i]`, for `j` ≥ `i`.

d. the maximum value of `a[j]/a[i]`, for `j` ≥ `i`.

Programming Projects

5.27. The Sieve of Eratosthenes is a method used to compute all primes less than N. Begin by making a table of integers 2 to N. Find the smallest integer, i, that is not crossed out. Then print i and cross out i, $2i$, $3i$, $\ldots$. When $i > \sqrt{N}$, the algorithm terminates. The running time has been shown to be $O(N \log\log N)$. Write a program to implement the Sieve and verify that the running time claim. How difficult is it to differentiate the running time from $O(N)$ and $O(N \log N)$?

5.28. The equation $A^5 + B^5 + C^5 + D^5 + E^5 = F^5$ has exactly one integral solution that satisfies $0 < A \le B \le C \le D \le E \le F \le 75$. Write a program to find the solution. *Hint:* First, precompute all values of X^5 and store them in an array. Then, for each tuple (A, B, C, D, E), you only need to verify that some F exists in the array. (There are several ways to check for F, one of which is to use a binary search to check for F. Other methods might prove to be more efficient.)

5.29. Implement the maximum contiguous subsequence sum algorithms to obtain data equivalent to the data in Figure 5.10. Compile the programs with the highest optimization settings.

References

The maximum contiguous subsequence sum problem is from [5]. References [4], [5], and [6] show how to optimize programs for speed. Interpolation search was first suggested in [14] and was analyzed in [13]. References [1], [8], and [17] provide a more rigorous treatment of algorithm analysis. The three-part series [10], [11], and [12], newly updated, remains the foremost reference work on the topic. The mathematical background required for more advanced algorithm analysis is provided by [2], [3], [7], [15], and [16]. An especially good book for advanced analysis is [9].

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
2. M. O. Albertson and J. P. Hutchinson, *Discrete Mathematics with Algorithms*, John Wiley & Sons, New York, NY, 1988.
3. Z. Bavel, *Math Companion for Computer Science*, Reston Publishing Company, Reston, Va., 1982.
4. J. L. Bentley, *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, N.J., 1982.
5. J. L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, Mass., 1986.
6. J. L. Bentley, *More Programming Pearls*, Addison-Wesley, Reading, Mass., 1988.

BOOK.mkr Page 291 Wednesday, March 14, 2001 1:11 PM

7. R. A. Brualdi, *Introductory Combinatorics*, North-Holland, New York, N.Y., 1977.

8. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1990.

9. R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, Mass., 1989.

10. D. E. Knuth, *The Art of Computer Programming, Vol 1: Fundamental Algorithms*, 3d ed., Addison-Wesley, Reading, Mass, 1997.

11. D. E. Knuth, *The Art of Computer Programming, Vol 2: Seminumerical Algorithms*, 3d ed., Addison-Wesley, Reading, Mass., 1997.

12. D. E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching*, 2d ed., Addison-Wesley, Reading, Mass., 1998.

13. Y. Pearl, A. Itai, and H. Avni, “Interpolation Search – A log log *N* Search,” *Communications of the ACM* **21** (1978), 550–554.

14. W. W. Peterson, “Addressing for Random Storage,” *IBM Journal of Research and Development* **1** (1957), 131–132.

15. F. S. Roberts, *Applied Combinatorics*, Prentice-Hall, Englewood Cliffs, N.J., 1984.

16. A. Tucker, *Applied Combinatorics*, 2d ed., John Wiley & Sons, New York, N.Y., 1984.

17. M. A. Weiss, *Data Structures and Algorithm Analysis in Java*, Addison-Wesley, Reading, Mass., 1999.

CHAPTER

6 The Collections API

Many algorithms require the use of a proper representation of data to achieve efficiency. This representation and the operations that are allowed for it are known as a *data structure*. Each data structure allows arbitrary insertion but differs in how it allows access to members in the group. Some data structures allow arbitrary access and deletions, whereas others impose restrictions, such as allowing access only to the most recently or least recently inserted item in the group.

As part of Java, a supporting library known as the *Collections API* is provided. Most of the Collections API resides in `java.util`. This API provides a collection of data structures. It also provides some generic algorithms, such as sorting. The Collections API makes heavy use of inheritance.

Our primary goal is to describe, in general terms, some examples and applications of data structures. Our secondary goal is to describe the basics of the Collections API, so that we can use it in Part III. We do not discuss the theory behind an efficient Collections API implementation until Part IV, at which point we provide simplified implementations of some core Collections API components. But delaying the discussion of the Collection API's implementation until after we use it is not a problem. We do not need to know *how* something is implemented so long as we know that it *is* implemented.

In this chapter, we show:

- common data structures, their allowed operations, and their running times;
- some applications of the data structures; and
- the organization of the Collections API, and its integration with the rest of the language

6.1 Introduction

A *data structure* is a representation of data and the operations allowed on that data.

Data structures allow us to achieve an important object-oriented programming goal: component reuse. The data structures described in this section (and implemented later in Part IV) have recurring uses. When each data structure has been implemented once, it can be used over and over in various applications.

A *data structure* is a representation of data and the operations allowed on that data. Many, but by no means all, of the common data structures store a collection of objects, and then provide methods to add a new object to, remove an existing object from, or access a contained object in the collection.

Data structures allow us to achieve component reuse.

In this chapter we examine some of the fundamental data structures and their applications. Using a high-level protocol, we describe typical operations that are usually supported by the data structures, and briefly describe their uses. When possible, we give an estimate of the cost of implementing these operations efficiently. This estimate is often based on analogy with non-computer applications of the data structure. Our high-level protocol usually supports only a core set of basic operations. Later, when describing the basics of how the data structures can be implemented (in general there are multiple competing ideas), we can more easily focus on language-independent algorithmic details if we restrict the set of operations to a minimum core set.

As an example, Figure 6.1 illustrates a generic protocol that many data structures tend to follow. We do not actually use this protocol directly in any code. However, an inheritance-based hierarchy of data structures could use this class as a starting point.

Then, we give a description of the Collections API interface that is provided for these data structures. By no means does the Collections API represent the best way of doing things. However, it represents the one library for data structures and algorithms guaranteed to be available. Its use also illustrates some of the core issues that must be dealt with once the theory is taken care of.

The Collections API is the one library for data structures and algorithms that is guaranteed to be available.

```
package weiss.nonstandard;

// SimpleContainer protocol
public interface SimpleContainer
{
    void insert( Object x );
    void remove( Object x );
    Object find( Object x );

    boolean isEmpty( );
    void makeEmpty( );
}
```

Figure 6.1 A generic protocol for many data structures

We defer consideration of efficient implementation of data structures to Part IV. At that point we will provide, as part of package `weiss.nonstandard`, some competing implementations for data structures that follow the simple protocols developed in this chapter. We will also provide one implementation for the basic Collections API components described in the chapter, in package `weiss.util`. Thus we are separating the interface of the Collections API (that is, what it does, which we describe in the chapter) from its implementation (that is, how it is done, which we describe in Part IV). This approach — the separation of the interface and implementation — is part of the object-oriented paradigm. The user of the data structure needs to see only the available operations, not the implementation. Recall this is the encapsulation and information-hiding part of object-oriented programming.

The rest of this chapter is organized as follows: First, we discuss the basics of the *iterator pattern*, which is used throughout the Collections API. Then we discuss the interface for containers and iterators in the Collections API. Next we describe some Collections API algorithms, and finally, we examine some other data structures many of which are supported in the Collections API.

6.2 The Iterator Pattern

The Collections API makes heavy use of a common technique known as the iterator pattern. So before we begin our discussion of the Collections API, we examine the ideas behind the iterator pattern.

An *iterator* object controls iteration of a collection.

Consider the problem of printing the elements in a collection. Typically, the collection is an array, so assuming that the object `v` is an array, its contents are easily printed with code like:

```
for( int i = 0; i < v.length; i++ )
    System.out.println( v[ i ] );
```

In this loop, `i` is an *iterator* object, because it is the object that is used to control the iteration. However, using the integer `i` as an iterator constrains the design: We can only store the collection in an array-like structure. A more flexible alternative is to design an iterator class that encapsulates a position inside of a collection. The iterator class provides methods to step through the collection.

The key is the concept of programming to an interface: We want the code that performs access of the container to be as independent of the type of the container as possible. This is done by using only methods that are common to all containers and their iterators.

When we program to an interface, we write code that uses the most abstract methods. These methods will be applied to the actual concrete types.

There are many different possible iterator designs. If we replace `int i` with `IteratorType itr`, then the loop above expresses

```
for( itr = v.first( ); itr.isValid( ); itr.advance( ) )
    System.out.println( itr.getData( ) );
```

This suggests an iterator class that contains methods such as `isValid`, `advance`, `getData`, etc.

We describe two designs, outside of the Collections API context, that lead to the Collections API iterator design. We discuss the specifics of the Collections iterators in Section 6.3.2, deferring implementations to Part IV.

6.2.1 Basic Iterator Design

`iterator` returns an appropriate iterator for the collection.

The first iterator design uses only three methods. The container class is required to provide an `iterator` method. `iterator` returns an appropriate iterator for the collection. The iterator class has only two methods, `hasNext` and `next`. `hasNext` returns true if the iteration has not yet been exhausted. `next` returns the next item in the collection (and in the process, advances the current position). This iterator interface is similar to the interface provided in the Collections API.

To illustrate the implementation of this design, we outline the collection class and provide an iterator class, `MyContainer` and `MyContainerIterator`, respectively. Their use is shown in Figure 6.2. The data members and `iterator` method for `MyContainer` are written in Figure 6.3. To simplify matters, we omit the constructors, and methods such as `add`, `size`, etc. The `ArrayList` class from earlier chapters can be reused to provide an implementation of these methods.

The iterator is constructed with a reference to the container that it iterates over.

`iterator` simply returns a new iterator; notice that the iterator must have information about the container that it is iterating over. Thus the iterator is constructed with a reference to the `MyContainer`.

Figure 6.4 shows the `MyContainerIterator`. The iterator keeps a variable (`current`) that represents the current position in the container, and a reference to the container. The implementation of the constructor and two methods is straightforward. The constructor initializes the container reference, `hasNext` simply compares the current position with the container size, and `next` uses the current position to index the array (and then advances the current position).

A limitation of this iterator design is the relatively limited interface. Observe that it is impossible to reset the iterator back to the beginning, and that the `next` method couples access of an item with advancing. The `next`, `hasNext` design is what is used in the Java Collections API; many people feel that the API should have provided a more flexible iterator. It is certainly possible to put more functionality in the iterator, while leaving the `MyContainer` class implementation completely unchanged. On the other hand, doing so illustrates no new principles.

The better design would put more functionality in the iterator

```
    public static void main( String [ ] args )
    {
        MyContainer v = new MyContainer( );

        v.add( "3" );
        v.add( "2" );

        System.out.println( "Container contents: " );
        MyContainerIterator itr = v.iterator( );
        while( itr.hasNext( ) )
            System.out.println( itr.next( ) );
    }
```

Figure 6.2 `main` method to illustrate iterator design #1

```
package weiss.ds;

public class MyContainer
{
    Object [ ] items;
    int size;

    public MyContainerIterator iterator( )
      { return new MyContainerIterator( this ); }

    // Other methods
}
```

Figure 6.3 The `MyContainer` class, design #1

```
// An iterator class that steps through a MyContainer.

package weiss.ds;

public class MyContainerIterator
{
    private int current = 0;
    private MyContainer container;

    MyContainerIterator( MyContainer c )
      { container = c; }

    public boolean hasNext( )
      { return current < container.size; }

    public Object next( )
      { return container.items[ current++ ]; }
}
```

Figure 6.4 Implementation of the `MyContainerIterator`, design #1

Note that in the implementation of `MyContainer`, the data members `items` and `size` are package visible, rather than being private. This unfortunate relaxation of the usual privacy of data members is necessary because these data members need to be accessed by `MyContainerIterator`. Similarly, the `MyContainerIterator` constructor is package visible, so that it can be called by `MyContainer`.

6.2.2 Inheritance-based Iterators and Factories

The iterator designed so far manages to abstract the concept of iteration into an iterator class. This is good, because it means that if the collection changes from an array-based collection to something else, the basic code such as lines 9 and 11 in Figure 6.2 does not need to change.

While this is a significant improvement, changes from an array-based collection to something else require that we change all the declarations of the iterator. For instance, in Figure 6.2, we would need to change line 9. We discuss an alternative in this section.

An inheritance-based iteration scheme defines an iterator interface. Clients program to this interface.

Our basic idea is to define an interface `Iterator`. Corresponding to each different kind of container is an iterator that implements the `Iterator` protocol. In our example, this gives three classes: `MyContainer`, `Iterator`, and `MyContainerIterator`. The relationship that holds is `MyContainerIterator` IS-A `Iterator`. The reason we do this is that each container can now create an appropriate iterator, but pass it back as an abstract `Iterator`.

Figure 6.5 shows `MyContainer`. In the revised `MyContainer`, the `iterator` method returns a reference to an `Iterator` object; the actual type turns out to be a `MyContainerIterator`. Since `MyContainerIterator` IS-A `Iterator`, this is safe to do.

```
package weiss.ds;

public class MyContainer
{
    Object [ ] items;
    int size;

    public Iterator iterator( )
      { return new MyContainerIterator( this ); }

    // Other methods not shown.
}
```

Figure 6.5 The `MyContainer` class, design #2

```java
package weiss.ds;

public interface Iterator
{
    boolean hasNext( );
    Object next( );
}
```

Figure 6.6 The `Iterator` interface, design #2

```java
// An iterator class that steps through a MyContainer.

package weiss.ds;

class MyContainerIterator implements Iterator
{
    private int current = 0;
    private MyContainer container;

    MyContainerIterator( MyContainer c )
      { container = c; }

    public boolean hasNext( )
      { return current < container.size; }

    public Object next( )
      { return container.items[ current++ ]; }
}
```

Figure 6.7 Implementation of the `MyContainerIterator`, design #2

```java
    public static void main( String [ ] args )
    {
        MyContainer v = new MyContainer( );

        v.add( "3" );
        v.add( "2" );

        System.out.println( "Container contents: " );
        Iterator itr = v.iterator( );
        while( itr.hasNext( ) )
            System.out.println( itr.next( ) );
    }
```

Figure 6.8 `main` method to illustrate iterator design #2

Because `iterator` creates and returns a new `Iterator` object, whose actual type is unknown, it is commonly known as a *factory method*. The iterator interface, which serves simply to establish the protocol by which all subclasses of `Iterator` can be accessed, is shown in Figure 6.6. There are only two changes to the implementation of `MyContainerIterator`, shown in Figure 6.7. and both changes are at line 5. First, the `implements` clause has been added. Second, `MyContainerIterator` no longer needs to be a public class.

A *factory method* creates a new concrete instance, but returns it using a reference to the interface type.

Figure 6.8 demonstrates how the inheritance-based iterators are used. At line 9, we see the declaration of `itr`: it is now a reference to an `Iterator`. Nowhere in `main` is there any mention of the actual `MyContainerIterator` type. The fact that a `MyContainerIterator` exists is not used by any clients of the `MyContainer` class. This is a very slick design, and illustrates nicely the idea of hiding an implementation and programming to an interface. The implementation can be made even slicker by use of nested classes, and a Java feature known as inner classes. Those implementation details are deferred until Chapter 15.

Nowhere in `main` is there any mention of the actual iterator type.

6.3 Collections API: Containers and Iterators

This section describes the basics of the Collection API iterators, and how they interact with containers. We know that an iterator is an object that is used to traverse a collection of objects. In the Collections API such a collection is abstracted by the `Collection` interface, and the iterator is abstracted by the `Iterator` interface.

The Collection API iterators are somewhat inflexible, in that they provide few operations. It uses an inheritance model described in Section 6.2.2.

6.3.1 The `Collection` interface

The `Collection` interface represents a group of objects, known as its elements.

The `Collection` interface represents a group of objects, known as its elements. Some implementations, such as vectors and lists, are unordered; others, such as sets and maps, may be ordered. Some implementations allow duplicates; others do not. All containers support the following operations.

`boolean isEmpty( )`
returns `true` if the container contains no elements and `false` otherwise.

`int size( )`
returns the number of elements in the container.

`boolean add( Object x )`
adds item `x` to the container. Returns `true` if this operation succeeds and `false` otherwise (e.g. if the container does not allow duplicates and `x` is already in the container).

`boolean contains( Object x )`
returns true if `x` is in the container and `false` otherwise.

`boolean remove( Object x )`
Removes item `x` from the container. Returns `true` if `x` was removed and `false` otherwise.

```
package weiss.util;

/**
 * Collection interface; the root of all 1.2 collections.
 */
public interface Collection extends java.io.Serializable
{
    /**
     * Returns the number of items in this collection.
     */
    int size( );

    /**
     * Tests if this collection is empty.
     */
    boolean isEmpty( );

    /**
     * Tests if some item is in this collection.
     */
    boolean contains( Object x );

    /**
     * Adds an item to this collection.
     */
    boolean add( Object x );

    /**
     * Removes an item from this collection.
     */
    boolean remove( Object x );

    /**
     * Change the size of this collection to zero.
     */
    void clear( );

    /**
     * Obtains an Iterator used to traverse the collection.
     */
    Iterator iterator( );

    /**
     * Obtains a primitive array view of the collection.
     */
    Object [] toArray( );
}
```

Figure 6.9 Sample specification of the `Collection` interface

`void clear( )`
makes the container empty

`Object [] toArray( )`
returns an array that contains references to all items in the container.

`Iterator iterator( )`
returns an `iterator` that can be used to begin traversing all locations in the container.

Figure 6.9 illustrates a specification of the `Collection` interface. The `Collection` interface in `java.util` contains some additional methods, but we will be content with this subset. By convention, all implementations supply both a zero-parameter constructor that creates an empty collection and a constructor that creates a collection that refers to the same elements as another collection. This is basically a shallow-copy of a collection. However, there is no syntax in the language that forces the implementation of these constructors.

The Collections API also codifies the notion of an *optional interface method*. For instance, suppose we want an immutable collection: once it is constructed, its state should never change. An immutable collection appears incompatible with `Collection`, since `add` and `remove` do not make sense for immutable collections.

However, there is an existing loophole: although the implementor of the immutable collection must implement `add` and `remove`, there is no rule that says these methods must do anything. Instead, the implementor can simply throw a runtime `UnsupportedOperationException`. In doing so, the implementor has technically implemented the interface, while not really providing `add` and `remove`.

By convention, interface methods that document that they are *optional* can be implemented in this manner. If the implementation chooses not to implement an optional method, then it should document that fact. It is up to the client user of the API to verify that the method is implemented by consulting the documentation, and if the client ignores the documentation and calls the method anyway, the runtime `UnsupportedOperationException` is thrown, signifying a programming error.

Optional methods are somewhat controversial, but they do not represent any new language additions. They are simply a convention.

We will eventually implement all methods. The most interesting of these methods is `iterator`, which is a factory method that creates and returns an `Iterator` object. The operations that can be performed by an `Iterator` are described in Section 6.3.2.

6.3.2 `Iterator` interface

As described in Section 6.2, an *iterator* is an object that allows us to iterate through all objects in a collection. The technique of using an iterator class was discussed in the context of read-only vectors in Section 6.2.

An *iterator* is an object that allows us to iterate through all objects in a collection.

The `Iterator` interface is the Collections API is small, and contains only three methods:

`boolean hasNext( )`
returns true if there are more items to view in this iteration.

`Object next( )`

returns a reference to the next object not yet seen by this iterator. The object becomes seen, and thus advances the iterator.

`void remove( )`

removes the last item viewed by `next`. This can be called only once between calls to `next`.

The `Iterator` interface contains only three methods: `next`, `hasNext`, and `remove`.

Each collection defines its own implementation of the `Iterator` interface, in a class that is invisible to users of the `java.util` package.

The iterators also expect a stable container. An important problem that occurs in the design of containers and iterators is to decide what happens if the state of container is modified while an iteration is in progress. The Collections API takes a strict view: any external structural modification of the container (adds, removes, etc.) will result in a `ConcurrentModificationException` by the iterator methods when one of the methods is called. In other words, if we have an iterator, and then an object is added to the container, and then we invoke the `next` method on the iterator, the iterator will detect that it is now invalid, and an exception will be thrown by `next`.

The `Iterator` methods throw an exception if its container has been structurally modified.

This means that it is impossible to remove an object from a container when we have seen it via an iterator, without invalidating the iterator. This is one reason why there is a `remove` method in the iterator class. Calling the iterator `remove` causes the last seen object to be removed from the container. It invalidates all other iterators that are viewing this container, but not the iterator that performed the `remove`. It is also likely to be more efficient than the container's `remove` method, at least for some collections. However, `remove` cannot be called twice in a row. Furthermore, `remove` preserves the semantics of `next` and `hasNext`, because the next unseen item in the iteration remains the same. This version of `remove` is listed as an optional method, so the programmer needs to check that it is implemented. `remove` has been criticized as poor design, but we will use it at one point in the text.

Figure 6.10 provides a sample specification of the `Iterator` interface. As an example of using the `Iterator`, the routine in Figure 6.11 prints each element in any container. If the container is an ordered set, its elements are output in sorted order.

```java
package weiss.util;

/**
 * Iterator interface.
 */
public interface Iterator
{
    /**
     * Tests if there are items not yet iterated over.
     */
    boolean hasNext( );

    /**
     * Obtains the next (as yet unseen) item in the collection.
     */
    Object next( );

    /**
     * Remove the last item returned by next.
     * Can only be called once after next.
     */
    void remove( );
}
```

Figure 6.10 Sample specification of `Iterator`

```
// Print the contents of Collection c
public static void printCollection( Collection c )
{
    Iterator itr = c.iterator( );
    while( itr.hasNext( ) )
        System.out.println( itr.next( ) );
}
```

Figure 6.11 Print the contents of any `Collection`

6.4 Generic Algorithms

The `Collections` class contains a set of static methods that operate on `Collection` objects.

The Collections API provides a few general purpose algorithms that operate on all of the containers. These are static methods in the `Collections` class (note that this is a different class than the `Collection` interface). There are also some static methods in the `Arrays` class that manipulate arrays (sorting, searching, etc.). Most of those methods are overloaded — for `Object` and once for each of the primitive types (except `boolean`).

We examine only a few of the algorithms, with the intention of showing the general ideas that pervade the Collections API, while documenting the specific algorithms that will be used in Part III.

The material in Section 4.7 is an essential prerequisite to this section.

Some of the algorithms make use of function objects. Consequently, the material in Section 4.7 is an essential prerequisite to this section.

6.4.1 `Comparator` Function Objects

Many Collections API classes and routines require the ability to order objects. There are two ways to do this. One possibility is that the objects implement the `Comparable` interface, and provide a `compareTo` method. The other possibility is that the comparison function is embedded as the `compare` method in an object that implements the `Comparator` interface. `Comparator` is defined in `java.util`; a sample implementation was shown in Figure 4.29, and is repeated in Figure 6.12.

```
package weiss.util;

/**
 * Comparator function object interface.
 */
public interface Comparator
{
    /**
     * Return the result of comparing lhs and rhs.
     * @param lhs first object.
     * @param rhs second object.
     * @return < 0 if lhs is less than rhs,
     *           0 if lhs is equal to rhs,
     *         > 0 if lhs is greater than rhs.
     * @throws ClassCastException if objects
     *         cannot be compared.
     */
    int compare( Object lhs, Object rhs );
}
```

Figure 6.12 The `Comparator` interface, originally defined in `java.util` rewritten for the `weiss.util` package.

```java
package weiss.util;

/**
 * Instanceless class contains static methods
 * that operate on collections.
 */
public class Collections
{
    private Collections( )
    {
    }

    /**
     * Returns a comparator that imposes the reverse of the
     * default ordering on a collection of objects that
     * implement the Comparable interface.
     * @return the comparator.
     */
    public static Comparator reverseOrder( )
    {
        return REVERSE_COMPARATOR;
    }

    private static class ReverseComparator
                          implements Comparator
    {
        public int compare( Object lhs, Object rhs )
        {
            return -( (Comparable) lhs ).compareTo( rhs );
        }
    }

    private static class DefaultComparator
                          implements Comparator
    {
        public int compare( Object lhs, Object rhs )
        {
            return ( (Comparable) lhs ).compareTo( rhs );
        }
    }

    private static final Comparator REVERSE_COMPARATOR =
                                    new ReverseComparator( );
            static final Comparator DEFAULT_COMPARATOR =
                                    new DefaultComparator( );
```

Figure 6.13 `Collections` class (part 1): private constructor and `reverseOrder`

```
    /*
     * Returns the maximum object in the collection
     * using default ordering.
     * @param coll the collection.
     * @return the maximum object.
     * @throws NoSuchElementException if coll is empty.
     * @throws ClassCastException if objects in collection
     *         cannot be compared.
     */
    public static Object max( Collection coll )
    {
        return max( coll, DEFAULT_COMPARATOR );
    }

    /**
     * Returns the maximum object in the collection.
     * @param coll the collection.
     * @param cmp the comparator.
     * @return the maximum object.
     * @throws NoSuchElementException if coll is empty.
     * @throws ClassCastException if objects in collection
     *         cannot be compared.
     */
    public static Object max( Collection coll, Comparator cmp )
    {
        if( coll.size( ) == 0 )
            throw new NoSuchElementException( );

        Iterator itr = coll.iterator( );
        Object maxValue = itr.next( );

        while( itr.hasNext( ) )
        {
            Object current = itr.next( );
            if( cmp.compare( current, maxValue ) > 0 )
                current = maxValue;
        }
        return maxValue;
    }
}
```

Figure 6.14 `Collections` class (part 2): `max`

6.4.2 The `Collections` Class

Although we will not make use of the `Collections` class in this text, it has two methods that are thematic of how generic algorithms for the Collections API are written. We write these methods in the `Collections` class implementation that spans Figures 6.13 and 6.14.

`reverseOrder` is a factory method that creates a `Comparator` representing the reverse natural order.

Figure 6.13 begins by illustrating the common technique of declaring a private constructor in classes that contain only static methods. This prevents instantiation of the class. It continues by providing the `reverseOrder` method. This is a factory method that returns a `Comparator` that provides the reverse of the natural ordering for `Comparable` objects. The returned object, declared at lines 42 and 43 is a shared static instance of the `ReverseComparator` class written in lines 24 to 31. In the `ReverseComparator` class, we eventually downcast `rhs` to a `Comparable` and use the `compareTo` method. If the objects are not comparable, this throws a `ClassCastException`.[1] This is an example of the type of code that might be implemented with an anonymous class. We have a similar declaration for the default comparator; since the standard API does not provide a public method to return this, we have not either, and instead declare a package visible `DEFAULT_COMPARATOR` instance that can be used by methods that need one.

Figure 6.14 illustrates the `max` method, which returns the largest element in any `Collection`. The one-parameter `max` calls the two parameter `max` by supplying the `DEFAULT_COMPARATOR`. The two-parameter `max` combines the iter-

1. Although not well-documented, the compare method in `java.util.Comparator` is allowed to throw a `NullPointerException` if either reference is `null` — even if both are `null`.

ator pattern with the function object pattern to step through the collection, and at line 80 uses calls to the function object to update the maximum item.

6.4.3 Binary Search

`binarySearch` uses binary search and returns the index of the matched item or a negative number if the item is not found.

The Collections API implementation of the binary search is the static method `Arrays.binarySearch`. There are actually seven overloaded versions — one for each of the primitive types except `boolean`, plus two more overloaded versions that work on `Objects` (one works with a comparator, one uses the default comparator). We will implement the `Object` versions; the other seven are mindless copy and paste.

As usual the array must be sorted; if it is not, the results are undefined (verifying that the array is sorted would destroy the logarithmic time bound for the operation).

If the search for the item is successful, the index of the match is returned. If the search is unsuccessful, we determine the first position that contains a larger item, add one to this position, and then return the negative of the value. Thus, the return value is always negative, because is at most `-1` (which occurs if the item we are searching for is smaller than all other items) and is at least `-a.length-1` (which occurs if the item we are searching for is larger than all other items).

The implementation is shown in Figure 6.15. As was the case for the `max` routines, the two-parameter `binarySearch` calls the three-parameter `binarySearch` (see lines 16 and 17). The three-parameter binary search routine mirrors the implementation in Figure 5.12.

We use the `binarySearch` method in Section 10.1.

```java
package weiss.util;

/**
 * Instanceless class that contains static methods
 * to manipulate arrays.
 */
public class Arrays
{
    private Arrays( ) { }

    /**
     * Search sorted array arr using default comparator.
     */
    public static int binarySearch( Object [ ] arr, Object x )
    {
        return binarySearch( arr, x,
                             Collections.DEFAULT_COMPARATOR );
    }

    /**
     * Search sorted array arr using a comparator.
     * If arr is not sorted, results are undefined.
     * @param arr the array to search.
     * @param x the object to search for.
     * @param cmp the comparator.
     * @return if x is found, returns index where it is found.
     *  otherwise, x is not found. In that case, a negative
     *  number is always returned, and this number is equal
     *  to -( p + 1 ), which p is the first position greater
     *  than x. This can range from -1 down to -(arr.length+1).
     * @throws ClassCastException if items are not comparable.
     */
    public static int binarySearch( Object [ ] arr, Object x,
                                    Comparator cmp )
    {
        int low = 0, mid = 0;
        int high = arr.length;

        while( low < high )
        {
            mid = ( low + high ) / 2;
            if( cmp.compare( x, arr[ mid ] ) > 0 )
                low = mid + 1;
            else
                high = mid;
        }
        if( cmp.compare( x, arr[ mid ] ) == 0 )
            return - ( low + 1 );
        return low;
    }
}
```

Figure 6.15 Implementation of `binarySearch` method in `Arrays` class

6.4.4 Sorting

The `Arrays` class contains a set of static methods that operate on arrays.

The Collections API provides a set of overloaded `sort` methods in the `Arrays` class. Simply pass an array of primitives, or an array of `Object` that implement `Comparable`, or an array of `Object` and a `Comparator`. We have not provided a `sort` method in our `Arrays` class.

`void sort( Object [] arr )`
rearranges the elements in the array to be in sorted order, using the natural order.

`void sort( Object [] arr, Comparator cmp )`
rearranges the elements in the array to be in sorted order, using the order specified by the comparator.

The generic sorting algorithms are required to run in $O(N \log N)$ time.

6.5 The `List` Interface

A *list* is a collection of items in which the items have a position.

A *list* is a collection of items in which the items have a position. The most obvious example of a list is an array. In an array, items are placed in position 0, 1, etc.

```
package weiss.util;

/**
 * List interface. Contains much less than java.util.
 */
public interface List extends Collection
{
    Object get( int idx );
    Object set( int idx, Object newVal );

    /**
     * Obtains a ListIterator object used to traverse
     * the collection bidirectionally.
     * @returns an iterator positioned
     *          prior to the requested element.
     * @param idx the index to start the iterator.
     *          Use size() to do complete reverse traversal.
     *          Use 0 to do complete forward traversal.
     * @throws IndexOutOfBoundsException if idx is not
     *          between 0 and size(), inclusive.
     */
    ListIterator listIterator( int pos );
}
```

Figure 6.16 Sample `List` interface

```java
package weiss.util;

/**
 * ListIterator interface for List interface.
 */
public interface ListIterator extends Iterator
{
    /**
     * Tests if there are more items in the collection
     * when iterating in reverse.
     * @return true if there are more items in the collection
     *  when traversing in reverse.
     */
    boolean hasPrevious( );

    /**
     * Obtains the previous item in the collection.
     * @return the previous (as yet unseen) item in the
     *  collection when traversing in reverse.
     */
    Object previous( );

    /**
     * Remove the last item returned by next or previous.
     * Can only be called once after next or previous.
     */
    void remove( );
}
```

Figure 6.17 Sample `ListIterator` interface

The `List` interface extends the `Collection` interface and abstracts the notion of a position.

The `List` interface extends the `Collection` interface and abstracts the notion of a position. The interface in `java.util` adds numerous methods to the `Collection` interface. We are content to add the three shown in Figure 6.16.

The first two methods are `get` and `set`, which are similar to the methods that we have already seen in `ArrayList`. The third method returns a more flexible iterator, the `ListIterator`.

6.5.1 The ListIterator Interface

As shown in Figure 6.17, ListIterator is just like an Iterator, except that it is bidirectional. Thus we can both advance and retreat. Because of this, the listIterator factory method that creates it must be given a value that is logically equal to the number of elements that have already been visited in the forward direction. If this value is zero, the ListIterator is initialized at the front, just like an Iterator. If this value is the size of the List, the iterator is initialized to have processed all elements in the forward direction. Thus in this state, hasNext returns false, but we can use hasPrevious and previous to traverse the list in reverse.

ListIterator is a bidirectional version of Iterator.

Figure 6.18 illustrates that we can use itr1 to traverse a list in the forward direction, and then once we reach the end, we can traverse the list backwards. It also illustrates itr2, which is positioned at the end, and simply processes the ArrayList in reverse.

One difficulty with the ListIterator is that the semantics for remove must change slightly. The new semantics are that remove deletes from the List the last object returned as a result of calling either next or previous, and remove can only be called once between calls to either next of previous. In order to override the Javadoc that is generated for remove, it is listed in the ListIterator interface.

```java
import java.util.ArrayList;
import java.util.ListIterator;

class TestArrayList
{
    public static void main( String [] args )
    {
        ArrayList lst = new ArrayList( );
        lst.add( "2" ); lst.add( "4" );
        ListIterator itr1 = lst.listIterator( 0 );
        ListIterator itr2 = lst.listIterator( lst.size( ) );

        System.out.print( "Forward: " );
        while( itr1.hasNext( ) )
            System.out.print( itr1.next( ) + " " );
        System.out.println( );

        System.out.print( "Backward: " );
        while( itr1.hasPrevious( ) )
            System.out.print( itr1.previous( ) + " " );
        System.out.println( );

        System.out.print( "Backward: " );
        while( itr2.hasPrevious( ) )
            System.out.print( itr2.previous( ) + " " );
        System.out.println( );
    }
}
```

Figure 6.18 Sample program that illustrates bidirectional iteration

6.5.2 `LinkedList` Class

There are two basic `List` implementations in the Collections API. One is the `ArrayList`, which we have already seen. The other is a `LinkedList`, which stores items internally in a different manner than `ArrayList`, yielding performance trade-offs. A third version is `Vector`, which is like `ArrayList`, but is from an older library, and is present mostly for compatibility with legacy (old) code. Using `Vector` is no longer in vogue.

The ArrayList may be appropriate if insertions are performed only at the high end of the array (using add), for the reasons discussed in Section 2.4.3. The ArrayList doubles the array if an insertion at the high-end would exceed an internal capacity. Although this gives good Big-Oh performance, especially if we add a constructor that allows the caller to suggest initial capacity for the internal array, the ArrayList is a poor choice if insertions are not made at the end, because then we must move items out of way.

The LinkedList class implements a linked list.

In a linked list, we store items noncontiguously rather than in the usual contiguous array. To do this, we store each object in a *node* that contains the object and a reference to the next node in the list, as shown in Figure 6.19. In this scenario, we maintain references to both the first and last node in the list.

The linked list is used to avoid large amounts of data movement. It stores items with an additional one reference per item overhead.

To be more concrete, a typical node looks like this:

```
class ListNode
{
    Object   data;   // Some element
    ListNode next;
}
```

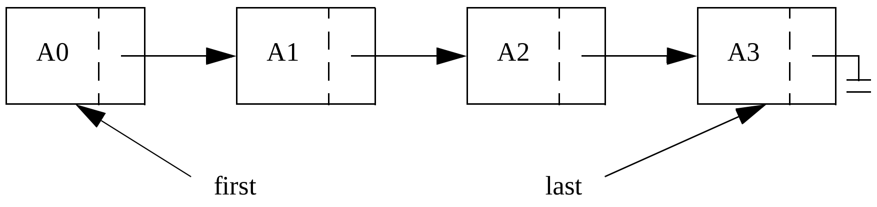

Figure 6.19 A simple linked list

At any point, we can add a new last item x by doing this:

```
last.next = new ListNode( ); // Attach a new ListNode
last = last.next;            // Adjust last
last.data = x;               // Place x in the node
last.next = null;            // It's the last; adjust next
```

The basic trade-off between `ArrayList` and `LinkedList` is that `get` is not efficient for `LinkedList`, while insertion and removal from the middle of a container is more efficiently supported by the `LinkedList`.

Now, an arbitrary item can no longer be found in one access. Instead, we must scan down the list. This is similar to the difference between accessing an item on a compact disk (one access) or a tape (sequential). While this may appear to make linked lists less attractive than arrays, they still have advantages. First, an insertion into the middle of the list does not require moving all of the items that follow the insertion point. Data movement is very expensive in practice, and the linked list allows insertion with only a constant number of assignment statements.

Insertions and deletions toward the middle of the sequence are inefficient in the `ArrayList`. An `ArrayList` allows direct access by the index, but a `LinkedList` should not. It happens, that in the Collections API, `get` and `set` are part of the `List` interface, so `LinkedList` supports these operations, slowly. Thus, the `LinkedList` can always be used unless efficient indexing is needed. The `ArrayList` may still be a better choice if insertions occur only at the end.

Access to the list is done through an iterator class.

To access items in the list, we need a reference to the corresponding node, rather than an index. The reference to the node would typically be hidden inside an iterator class.

Because `LinkedList` performs `add`s and `remove`s more efficiently, it has more operations than the `ArrayList`. Some of the additional operations available for `LinkedList` are:

`void addLast( Object element )`
appends `element` at the end of this `LinkedList`.

`void addFirst( Object element )`
prepends `element` to the front of this `LinkedList`.

`Object getFirst( )`
returns the first element in this `LinkedList`.

`Object getLast( )`
returns the last element in this `LinkedList`.

`void removeFirst( )`
removes the first element from this `LinkedList`.

`void removeLast( )`
removes the last element from this `LinkedList`.

We implement the `LinkedList` class in Part IV.

6.6 Stacks and Queues

In this section we describe two containers: the stack and queue. In principle, both have very simple interfaces (but not in the Collections API) and very efficient implementations. Even so, as we will see, they are very useful data structures.

6.6.1 Stacks

A *stack* is a data structure in which access is restricted to the most recently inserted element. It behaves very much like the common stack of bills, stack of plates, or stack of newspapers. The last item added to the stack is placed on the top and is easily accessible, whereas items that have been in the stack for a while are more difficult to access. Thus the stack is appropriate if we expect to access only the top item; all other items are inaccessible.

A *stack* restricts access to the most recently inserted item.

BOOK.mkr Page 326 Wednesday, March 14, 2001 1:11 PM

In a stack the three natural operations of `insert`, `remove`, and `find` are renamed `push`, `pop`, and `top`. These basic operations are illustrated in Figure 6.20.

The interface shown in Figure 6.21 illustrates the typical protocol. It is similar to the protocol previously seen in Figure 6.1. By pushing items and then popping them, we can use the stack to reverse the order of things.

Stack operations take a constant amount of time.

Each stack operation should take a constant amount of time, independent of the number of items in the stack. By analogy, finding today's newspaper in a stack of newspapers is fast, no matter how deep the stack is. However, arbitrary access in a stack is not efficiently supported, so we do not list it as an option in the protocol.

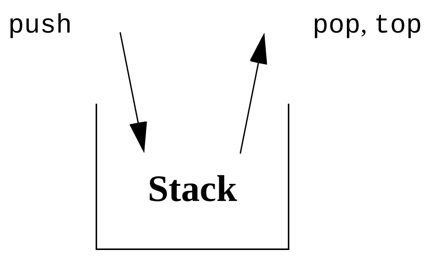

Figure 6.20 Stack model: input to a stack is by `push`, output is by `top`, deletion is by `pop`

```
// Stack protocol

package weiss.nonstandard;

public interface Stack
{
    void    push( Object x );  // insert
    void    pop( );            // remove
    Object  top( );            // find
    Object  topAndPop( );      // find + remove

    boolean isEmpty( );
    void    makeEmpty( );
}
```

Figure 6.21 Protocol for the stack

What makes the stack useful are the many applications for which we need to access only the most recently inserted item. An important use of stacks is in compiler design.

6.6.2 Stacks and Computer Languages

Compilers check your programs for syntax errors. Often, however, a lack of one symbol (e.g. a missing comment-ender `*/` or `}`) causes the compiler to spill out a hundred lines of diagnostics without identifying the real error; this is especially true when using anonymous classes.

A useful tool in this situation is a program that checks whether everything is balanced, that is, every `{` corresponds to a `}`, every `[` to a `]`, and so on. The sequence `[()]` is legal but `[(])` is not — so simply counting the numbers of each symbol is insufficient. (Assume for now that we are processing only a sequence of tokens and will not worry about problems such as the character constant `'{'` not needing a matching `'}'`.)

A stack can be used to check for unbalanced symbols.

A stack is useful for checking unbalanced symbols because we know that when a closing symbol such as `)` is seen, it matches the most-recently seen unclosed `(`. Therefore, by placing opening symbols on a stack, we can easily check that a closing symbol makes sense. Specifically, we have the following algorithm.

```
1. Make an empty stack.
2. Read symbols until the end of the file.
   a. If the token is an opening symbol, push it onto the stack.
   b. If it is a closing symbol and if the stack is empty, report an error.
   c. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, report an error.
3. At the end of the file, if the stack is not empty, report an error.
```

In Section 11.1 we will develop this algorithm to work for (almost) all Java programs. Details include error reporting, processing of comments, strings, and character constants, as well as escape sequences.

The stack is used to implement method calls in most programming languages.

The algorithm to check balanced symbols suggests a way to implement method calls. The problem is that, when a call is made to a new method, all the variables local to the calling method need to be saved by the system; otherwise, the new method would overwrite the calling routine's variables. Furthermore, the current location in the calling routine must be saved so that the new method knows where to go after it is done. The reason that this problem is similar to balancing symbols is because a method call and a method return are essentially the same as an open parenthesis and a closed parenthesis, so the same ideas should apply. This indeed is the case: as discussed in Section 7.3, the stack is used to implement method calls in most programming languages.

A final important application of the stack is the evaluation of expressions in computer languages. In the expression `1+2*3`, we see that at the point that the `*` is encountered, we have already read the operator `+` and the operands `1` and `2`. Does `*` operate on `2`, or `1+2`? Precedence rules tell us that `*` operates on `2`, which is the most recently seen operand. After the `3` is seen, we can evaluate `2*3` as `6` and then apply the `+` operator. This process suggests that operands and intermediate results should be saved on a stack. It also suggests that the operators be saved on the stack (since the `+` is held until the higher precedence `*` is evaluated). An algorithm that uses this strategy is *operator precedence parsing*, and is described in Section 11.2.

The *operator precedence parsing* algorithm uses a stack to evaluate expressions.

6.6.3 Queues

Another simple data structure is the *queue*, which restricts access to the least recently inserted item. In many cases being able to find and/or remove the most-recently inserted item is important. But in an equal number of cases, it is not only unimportant — but it is actually the wrong thing to do. In a multiprocessing system, for example, when jobs are submitted to a printer, we expect the least recent or most senior job to be printed first. This order is not only fair but it is also required to guarantee that the first job does not wait forever. Thus you can expect to find printer queues on all large systems.

The *queue* restricts access to the least recently inserted item.

The basic operations supported by queues are

- `enqueue`, or insertion at the back of the line;
- `dequeue`, or removal of the item from the front of the line; and
- `getFront`, or access of the item at the front of the line.

Queue operations take a constant amount of time.

Figure 6.22 illustrates these queue operations. Historically, `dequeue` and `getFront` have been combined into one operation; we do this by having `dequeue` return a reference to the item that it has removed.

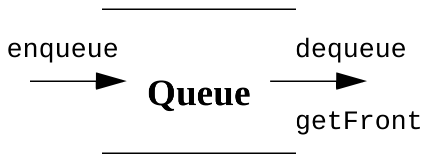

Figure 6.22 Queue model: input is by `enqueue`, output is by `getFront`, deletion is by `dequeue`

```
// Queue protocol

package weiss.nonstandard;

public interface Queue
{
    void  enqueue( Object x );    // insert
    Object getFront( );           // find
    Object dequeue( );            // remove + find

    boolean isEmpty( );
    void makeEmpty( );
}
```

Figure 6.23 Protocol for the queue

Figure 6.23 illustrates a possible protocol for queues. Because the queue operations and the stack operations are restricted similarly, we expect that they should also take a constant amount of time per query. This is indeed the case. All of the basic queue operations take $O(1)$ time. We will see several applications of queues in the case studies.

6.6.4 Stacks and Queues in the Collections API

The Collections API provides a `Stack` class but no queue class. The `Stack` methods are `push`, `pop`, and `peek`. However, the `Stack` class extends `Vector` and is slower than it needs to be; like `Vector`, its use is no longer in vogue, and can be replaced with `List` operations. The queue operations must be done using a `LinkedList` (e.g. `addLast`, `removeFirst`, `getFirst`).

The Collections API provides a `Stack` class, but no queue class.

6.7 Sets

A `Set` is a container that contains no duplicates. It supports all of the `Collection` methods. Figure 6.24 illustrates that the interface does little more than declare a type.

A `Set` contains no duplicates.

A `SortedSet` is a `Set` that maintains (internally) its items in sorted order. The objects that are added into the `SortedSet` must either be comparable, or a `Comparator` has to be provided when the container is instantiated. A `SortedSet` supports all of the `Set` methods, but its iterator is guaranteed to step through items in its sorted order. The `SortedSet` also allows us to find the smallest and largest item. The interface for our subset of `SortedSet` is shown in Figure 6.25.

The `SortedSet` is an ordered container. It allows no duplicates.

```
package weiss.util;

/**
 * Set interface.
 */
public interface Set extends Collection
{
}
```

Figure 6.24 Possible `Set` implementation.

```
package weiss.util;

/**
 * SortedSet interface.
 */
public interface SortedSet extends Set
{
    /**
     * Return the comparator used by this SortedSet.
     * @return the comparator or null if the
     * default comparator is used.
     */
    Comparator comparator( );

    /**
     * Find the smallest item in the set.
     * @return the smallest item.
     * @throws NoSuchElementException if the set is empty.
     */
    Object first( );

    /**
     * Find the largest item in the set.
     * @return the largest item.
     * @throws NoSuchElementException if the set is empty.
     */
    Object last( );
}
```

Figure 6.25 Possible `SortedSet` interface

```
    public static void main( String [] args )
    {
        Set s = new TreeSet( Collections.reverseOrder( ) );
        s.add( "joe" );
        s.add( "bob" );
        s.add( "hal" );
        printCollection( s );    // Figure 7.8
    }
```

Figure 6.26 Illustration of the `TreeSet`, using reverse order.

6.7.1 The `TreeSet` Class

The `TreeSet` is an implementation of `SortedSet`.

The `SortedSet` is implemented by a `TreeSet`. The underlying implementation of the `TreeSet` is a balanced-binary search tree, and is discussed in Chapter 19.

By default, ordering uses the default comparator. An alternate ordering can be specified by providing a `Comparator` to the constructor. As an example, Figure 6.26 illustrates how a `SortedSet` that stores strings is constructed. The call to `printCollection` will output elements in decreasing sorted order.

The `SortedSet`, like all `Set`s, does not allow duplicates. Two items are considered equal if the comparator's `compare` method returns 0.

In Section 5.6 we examined the static searching problem and saw that if the items are presented to us in sorted order, then we can support the `find` operation in logarithmic worst-case time. This is static searching because, once we are presented with the items, we cannot add or remove items. The `SortedSet`, allows us to add and remove items.

We are hoping that the worst-case cost of the `contains`, `add`, and `remove` operations is $O(\log N)$ because that would match the bound obtained for the static binary search. Unfortunately, for the simplest implementation of the `TreeSet`, this is not the case. The average case is logarithmic, but the worst case is $O(N)$ and occurs quite frequently. However, by applying some algorithmic tricks, we can obtain a more complex structure that does indeed have $O(\log N)$ cost per operation. The Collections API `TreeSet` is guaranteed to have this performance, and in Chapter 19, we discuss how to obtain it using the *binary search tree* and its variants, and provide an implementation of the `TreeSet`, with an iterator.

We can also use a binary search tree to access the *K*th smallest item in logarithmic time.

We mention in closing that although we can find the smallest and largest item in a `SortedSet` in $O(\log N)$ time, finding the Kth smallest item, where K is a parameter, is not supported in the Collections API. However, it is possible to perform this operation in $O(\log N)$ time, while preserving the running time of the other operations, if we do more work.

6.7.2 The `HashSet` Class

The `HashSet` implements the `Set` interface. It does not require a comparator.

In addition to the `TreeSet`, the Collections API provides a `HashSet` class that implements the `Set` interface. The `HashSet` differs from the `TreeSet` in that it cannot be used to enumerate items in sorted order, nor can it be used to obtain the smallest or largest item. Indeed, the items in the `HashSet` do not have to be comparable in any way. This means that the `HashSet` is less powerful than the `TreeSet`. If being able to enumerate the items in a `Set` in sorted order is not important, than it is often preferable to use the `HashSet` because not having to maintain sorted order allows the `HashSet` to obtain faster performance. To do so, elements placed in the `HashSet` must provide hints to the `HashSet` algorithms. This is done by having each element implement a special `hashCode` method; we describe this method later in this subsection.

Figure 6.27 illustrates the use of the `HashSet`. It is guaranteed that if we iterate through the entire `HashSet`, we will see each item once, but the order that the items are visited is unknown. It is almost certain that the order will not be the same as the order of insertion, nor will it be any kind of sorted order.

Like all `Set`s, the `HashSet` does not allow duplicates. Two items are considered equal if the `equals` method says so. Thus, any object that is inserted into the `HashSet` must have a properly overridden `equals` method.

Recall that in Section 4.8, we discussed that it is essential that `equals` is overridden (by providing a new version that takes an `Object` as parameter) rather than overloaded.

Implementing equals and hashCode

`equals` must be symmetric; this is tricky when inheritance is involved.

Overloading `equals` is very tricky when inheritance is involved. The contract for `equals` states that if `p` and `q` are not `null`, `p.equals(q)` should return the same value as `q.equals(p)`. This does not occur in Figure 6.28. In that example, clearly `b.equals(c)` returns true, as expected. `a.equals(b)` also returns true, because `BaseClass`'s `equals` method is used, and that only compares the `x` components. However, `b.equals(a)` returns false, because `DerivedClass`'s `equals` method is used, and the `instanceof` test will fail (`a` is not an instance of `DerivedClass`) at line 29.

```
    public static void main( String [] args )
    {
        Set s = new HashSet( );
        s.add( "joe" );
        s.add( "bob" );
        s.add( "hal" );
        printCollection( s );    // Figure 7.8
    }
```

Figure 6.27 Illustration of the `HashSet`, items are output in some order.

```java
class BaseClass
{
    public BaseClass( int i )
      { x = i; }

    public boolean equals( Object rhs )
    {
        // This is the wrong test (ok if final class)
        if( !( rhs instanceof BaseClass ) )
            return false;

        return x == ( (BaseClass) rhs ).x;
    }

    private int x;
}

class DerivedClass extends BaseClass
{
    public DerivedClass( int i, int j )
    {
        super( i );
        y = j;
    }

    public boolean equals( Object rhs )
    {
        // This is the wrong test.
        if( !( rhs instanceof DerivedClass ) )
            return false;

        return super.equals( rhs ) &&
               y == ( (DerivedClass) rhs ).y;
    }

    private int y;
}

public class EqualsWithInheritance
{
    public static void main( String [ ] args )
    {
        BaseClass a = new BaseClass( 5 );
        DerivedClass b = new DerivedClass( 5, 8 );
        DerivedClass c = new DerivedClass( 5, 8 );

        System.out.println( "b.equals(c): " + b.equals( c ) );
        System.out.println( "a.equals(b): " + a.equals( b ) );
        System.out.println( "b.equals(a): " + b.equals( a ) );
    }
}
```

Figure 6.28 Illustration of a broken implementation of `equals`

Solution 1 is to not override `equals` below the base class. Solution 2 is to require identically typed objects using `getClass`.

There are two standard solutions to this problem. One is to make the `equals` method final in `BaseClass`. This avoids the problem of conflicting `equals`.

The other solution is to strengthen the equals test to require that the types are identical, and not simply compatible, since the one-way compatibility is what breaks `equals`. In this example, a `BaseClass` and `DerivedClass` object would never be declared equal. Figure 6.29 shows a correct implementation. Line 8 contains the idiomatic test. `getClass` returns a special object of type `Class` (note the capital `C`) that represents information about any object's class. `getClass` is a final method in the `Object` class. If it returns the same `Class` instance, then the two objects have identical types.

```
class BaseClass
{
    public BaseClass( int i )
      { x = i; }

    public boolean equals( Object rhs )
    {
        if( rhs == null || getClass( ) != rhs.getClass( ) )
            return false;

        return x == ( (BaseClass) rhs ).x;
    }

    private int x;
}

class DerivedClass extends BaseClass
{
    public DerivedClass( int i, int j )
    {
        super( i );
        y = j;
    }

    public boolean equals( Object rhs )
    {
          // Class test not needed; getClass() is done
          // in superclass equals
        return super.equals( rhs ) &&
               y == ( (DerivedClass) rhs ).y;
    }

    private int y;
}
```

Figure 6.29 Correct implementation of `equals`

```
/**
 * Test program for HashSet.
 */
class IteratorTest
{
    public static void main( String [] args )
    {
        List stud1 = new ArrayList( );
        stud1.add( new SimpleStudent( "Bob", 0 ) );
        stud1.add( new SimpleStudent( "Joe", 1 ) );
        stud1.add( new SimpleStudent( "Bob", 2 ) ); // dup

           // will only have 2 items, if hashCode is
           // implemented. Otherwise will have 3 because
           // duplicate will not be detected.
        Set stud3 = new HashSet( stud1 );

        printCollection( stud1 ); // Bob Joe Bob
        printCollection( stud3 ); // 2 items in some order
    }
}

/**
 * Illustrates use of hashCode/equals for a user-defined class.
 * Students are matched on basis of name only.
 */
class SimpleStudent
{
    String name;
    int id;

    public SimpleStudent( String n, int i )
      { name = n; id = i; }

    public String toString( )
      { return name + " " + id; }

    public boolean equals( Object rhs )
    {
        if( rhs == null || getClass( ) != rhs.getClass( ) )
            return false;

        SimpleStudent other = (SimpleStudent) rhs;
        return name.equals( other.name );
    }

    public int hashCode( )
      { return name.hashCode( ); }
}
```

Figure 6.30 Illustrates the `equals` and `hashCode` methods for use in `HashSet`

When using a `HashSet`, we must also override the special `hashCode` method that is specified in `Object`; `hashcode` returns an `int`. Think of `hashCode` as providing a trusted hint of where the items are stored. If the hint is wrong, the item is not found, so if two objects are equal, they should provide identical hints. The contract for `hashCode` is that if two objects are declared equal by the `equals` method, then the `hashCode` method must return the same value for them. If this contract is violated, the `HashSet` will fail to find objects, even if `equals` declares that there is a match. If `equals` declares the objects are not equal, the `hashCode` method should return a different value for them, but this is not required. However, it is very beneficial for `HashSet` performance if `hashCode` rarely produces identical results for unequal objects. How `hash-Code` and `HashSet` interact is discussed in Chapter 20.

The `hashCode` method must be overridden if `equals` is overrid-den or the `HashSet` will not work.

Figure 6.30 illustrates a `SimpleStudent` class in which two `SimpleStudent`s are equal if they have the same name (and are both `SimpleStudent`s). This could be overridden using the techniques in Figure 6.29 as needed, or this method could be declared final. If it was declared final, then the test that is present allows only two identically-typed `SimpleStudent`s to be declared equal. If, with a final `equals`, we replace the test at line 40 with an `instanceof` test, then any two objects in the hierarchy can be declared equal if their names match.

The `hashCode` method at lines 47 and 48 simply uses the `hashCode` of the `name` field. Thus if two `SimpleStudent` objects have the same name (as declared by `equals`) they will have the same `hashCode`, since, presumably, the implementors of `String` honored the contract for `hashCode`.

The accompanying test program is part of a larger test that illustrates all the basic containers. Observe that if `hashCode` is unimplemented, all three `SimpleStudent` objects will be added to the `HashSet` because the duplicate will not be detected.

It turns out that on average, the `HashSet` operations can be performed in constant time. This seems like an astounding result because it means that the cost of a single `HashSet` operation does not depend on whether the `HashSet` contains 10 items or 10,000 items. The theory behind the `HashSet` is fascinating and is described in Chapter 20.

6.8 Maps

A `Map` is used to store a collection of entries that consists of *keys* and their *values*. The `map` maps keys to values.

A `Map` is used to store a collection of entries that consists of *keys* and their *values*. The `Map` maps keys to values. Keys must be unique, but several keys can map to the same value. Thus values need not be unique. There is a `SortedMap` interface that maintains the map logically in key-sorted order.

Not surprisingly, there are two implementations: the `HashMap` and `TreeMap`. The `HashMap` does not keep keys in sorted order, whereas the `TreeMap` does. For simplicity, we do not implement the `SortedMap` interface but we do implement `HashMap` and `TreeMap`.

```java
package weiss.util;

/**
 * Map interface.
 * A map stores key/value pairs.
 * In our implementations, duplicate keys are not allowed.
 */
public interface Map extends java.io.Serializable
{
    /**
     * Returns the number of keys in this map.
     */
    int size( );

    /**
     * Tests if this map is empty.
     */
    boolean isEmpty( );

    /**
     * Tests if this map contains a given key.
     */
    boolean containsKey( Object key );

    /**
     * Returns the value that matches the key or null
     * if the key is not found. Since null values are allowed,
     * checking if the return value is null may not be a
     * safe way to ascertain if the key is present in the map.
     */
    Object get( Object key );

    /**
     * Adds the key/value pair to the map, overriding the
     * original value if the key was already present.
     * Returns the old value associated with the key, or
     * null if the key was not present prior to this call.
     */
    Object put( Object key, Object value );

    /**
     * Removes the key and its value from the map.
     * Returns the previous value associated with the key,
     * or null if the key was not present prior to this call.
     */
    Object remove( Object key );
```

Figure 6.31 Sample `Map` interface (Part 1)

```
    /**
     * Removes all key/value pairs from the map.
     */
    void clear( );

    /**
     * Returns the keys in the map.
     */
    Set keySet( );

    /**
     * Returns the values in the map. There may be duplicates.
     */
    Collection values( );

    /**
     * Return a set of Map.Entry objects corresponding to
     * the key/value pairs in the map.
     */
    Set entrySet( );

    /**
     * Interface used to access the key/value pairs in a map.
     * From a map, use entrySet().iterator to obtain a iterator
     * over a Set of pairs. The next() method of this iterator
     * yields objects of type Map.Entry.
     */
    public interface Entry extends java.io.Serializable
    {
        /**
         * Returns this pair's key.
         */
        Object getKey( );

        /**
         * Returns this pair's value.
         */
        Object getValue( );
    }
}
```

Figure 6.32 Sample `Map` interface (Part 2)

The map can be implemented as a `Set` instantiated with a *pair* (see Section 3.7), whose comparator or `equals`/`hashCode` implementation refers only to the key. The `Map` interface does not extend `Collection`; instead it exists on its own. A sample interface that contains the most important methods is shown in Figures 6.31 and 6.32.

Most of the methods have intuitive semantics. `put` is used to add a key/value pair, `remove` is used to remove a key/value pair (only the key is specified), and `get` returns the value associated with a key. `null` values are allowed, which complicates issues for `get`, because the return value from `get` will not distinguish between a failed search and a successful search that returns `null` for the value. `containsKey` can be used if `null` values are known to be in the map.

The `Map` does not provide an `iterator` method or class. Instead it returns a `Collection` that can be used to view the contents of the map.

The `keySet` method gives a `Collection` that contains all the keys. Since duplicate keys are not allowed, the result of `keySet` is a `Set`, for which we can obtain an iterator. If the `Map` is a `SortedMap`, the `Set` is a `SortedSet`.

```
import weiss.util.Map;
import weiss.util.TreeMap;
import weiss.util.Set;
import weiss.util.Iterator;
import weiss.util.Comparator;

public class MapDemo
{
    public static void printMap( String msg, Map m )
    {
        System.out.println( msg + ":" );
        Set entries = m.entrySet( );
        Iterator itr = entries.iterator( );

        while( itr.hasNext( ) )
        {
            Map.Entry thisPair = (Map.Entry) itr.next( );
            System.out.print( thisPair.getKey( ) + ": " );
            System.out.println( thisPair.getValue( ) );
        }
    }

    // Do some inserts and printing (done in printMap).
    public static void main( String [ ] args )
    {
        Map phone1 = new TreeMap( );

        phone1.put( "John Doe", "212-555-1212" );
        phone1.put( "Jane Doe", "312-555-1212" );
        phone1.put( "Holly Doe", "213-555-1212" );

        System.out.println( "phone1.get(\"Jane Doe\"): "
                               + phone1.get( "Jane Doe" ) );
        System.out.println( );

        printMap( "phone1", phone1 );
    }
}
```

Figure 6.33 Illustration of the using the `Map` interface

Similarly, the `values` method returns a `Collection` that contains all the values. This really is a `Collection`, since duplicate values are allowed.

`Map.Entry` abstracts the notion of a pair in the map.

Finally, the `entrySet` method returns a collection of key/value pairs. Again, this is a `Set`, because the pairs must have different keys. The objects in the `Set` returned by the `entrySet` are pairs, there must be a type that represents key/value pairs. This is specified by the `Entry` interface that is nested in the `Map` interface. Thus the type of object that is in the `entrySet` is `Map.Entry`.

Figure 6.33 illustrates the use of the `Map` with a `TreeMap`. An empty map is created at line 26 and then populated with a series of `put` calls at lines 28 to 30. Lines 32 and 33 print the result of a call to `get`, which is used to obtain the value for the key `"Jane Doe"`. More interesting is the `printMap` routine that spans lines 9 to 21.

In `printMap`, at line 12, we obtain a `Set` containing `Map.Entry` pairs. From the `Set`, we can obtain an `Iterator` at line 13. The call to `next` at line 17 produces a `Map.Entry` object, and at that point we can obtain the key and value information using `getKey` and `getValue`, as shown on lines 18 and 19.

6.9 Priority Queues

The *priority queue* supports access of the minimum item only.

Although jobs sent to a printer are generally placed on a queue, that might not always be the best thing to do. For instance, one job might be particularly important, so we might want to allow that job to be run as soon as the printer is available. Conversely, when the printer finishes a job and several 1-page jobs and one 100-page job are waiting, it might be reasonable to print the long job last, even if it is not the last job submitted. (Unfortunately, most systems do not do this, which can be particularly annoying at times.)

Similarly, in a multiuser environment the operating system scheduler must decide which of several processes to run. Generally, a process is allowed to run only for a fixed period of time. A poor algorithm for such a procedure involves use of a queue. Jobs are initially placed at the end of the queue. The scheduler repeatedly takes the first job from the queue, runs it until either it finishes or its time limit is up, and places it at the end of the queue if it does not finish. Generally, this strategy is not appropriate because short jobs must wait and thus seem to take a long time to run. Clearly, users that are running an editor should not see a visible delay in the echoing of typed characters. Thus short jobs (that is, those using fewer resources) should have precedence over jobs that have already consumed large amounts of resources. Furthermore, some resource-intensive jobs, such as jobs run by the system administrator, might be important and should also have precedence.

If we give each job a number to measure its priority, then the smaller number (pages printed, resources used) tends to indicate greater importance. Thus we want to be able to access the smallest item in a collection of items and remove it from the collection. To do so we use the `findMin` and `deleteMin` operations. The data structure that supports these operations is the *priority queue* and supports access of the minimum item only. Figure 6.34 illustrates the basic priority queue operations.

Unfortunately, although the priority queue is a fundamental data structure, there is no implementation of it in the Collections API. One idea might be to use a `SortedSet`, but that is not sufficient because it is important for a priority queue to allow duplicate items.

A typical priority queue protocol is shown in Figures 6.35 and 6.36. Although it is possible to use comparators, we keep the implementation simple by assuming all items in the priority queue are `Comparable`.

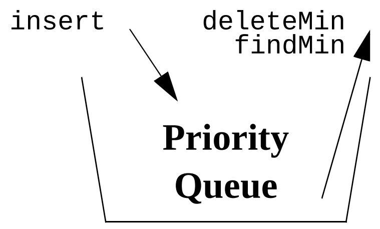

Figure 6.34 Priority queue model: only the minimum element is accessible

```
package weiss.nonstandard;

/**
 * PriorityQueue interface.
 * Some priority queues may support a decreaseKey operation,
 * but this is considered an advanced operation. If so,
 * a Position is returned by insert.
 * Note that all "matching" is based on the compareTo method.
 */
public interface PriorityQueue
{
    /**
     * Insert into the priority queue, maintaining heap order.
     * May return a Position useful for decreaseKey.
     */
    public Position insert( Comparable x );

    /**
     * Returns the smallest item in the priority queue.
     * @throws UnderflowException if empty.
     */
    public Comparable findMin( );
```

Figure 6.35 Protocol for priority queue in `weiss.nonstandard` (Part 1)

```
    /**
     * Remove and return the smallest item.
     * @throws UnderflowException if empty.
     */
    public Comparable deleteMin( );
    /**
     * Returns true if empty, false otherwise.
     */
    public boolean isEmpty( );

    /**
     * Make the priority queue logically empty.
     */
    public void makeEmpty( );

    /**
     * Returns the size.
     */
    public int size( );

    /**
     * The Position interface represents a type that can
     * be used for the decreaseKey operation.
     */
    public interface Position
    {
        /**
         * Returns the value stored at this position.
         */
        Comparable getValue( );
    }

    /**
     * Change the value of the item stored in the pairing heap.
     * This is considered an advanced operation and might not
     * be supported by all priority queues. A priority queue
     * will signal its intention to not support decreaseKey by
     * having insert return null consistently.
     * @param p any non-null Position returned by insert.
     * @param newVal the new value, which must be smaller
     *    than the currently stored value.
     * @throws IllegalArgumentException if p invalid
     */
    public void decreaseKey( Position p, Comparable newVal );
}
```

Figure 6.36 Protocol for priority queue in `weiss.nonstandard` (Part 2)

```
import weiss.nonstandard.PriorityQueue;
import weiss.nonstandard.BinaryHeap;

public class PriorityQueueDemo
{
    public static void dumpPQ( String msg, PriorityQueue pq )
    {
        System.out.println( msg + ":" );
        while( !pq.isEmpty( ) )
            System.out.println( pq.deleteMin( ) );
    }

    // Do some inserts and removes (done in dumpPQ).
    public static void main( String [ ] args )
    {
        PriorityQueue minPQ = new BinaryHeap( );

        minPQ.insert( new Integer( 4 ) );
        minPQ.insert( new Integer( 3 ) );
        minPQ.insert( new Integer( 5 ) );

        dumpPQ( "minPQ", minPQ );
    }
}
```

Figure 6.37 Routine to demonstrate the `PriorityQueue` in `weiss.nonstandard`

Most of the interface is straightforward. The tricky method, which is considered part of advanced priority queue implementations only, is `decreaseKey`. This operation reduces the value of an item that is in the priority queue. In order to do so, the entry in the priority queue must have a known position that is immutable. Essentially, this position must be established when an item is inserted, and can never change. The abstraction of a position is expressed in the `Position` nested interface, and `insert` returns a `Position` object when an item is added to the priority queue. If a priority queue does not support the `decreaseKey` operation, `insert` can simply return `null`. Figure 6.37 illustrates the use of the priority queue.

The *binary heap* implements the priority queue in logarithmic time per operation with little extra space.

As the priority queue supports only the `deleteMin` and `findMin` operations, we might expect performance that is a compromise between the constant-time queue and the logarithmic time set. Indeed, this is the case. The basic priority queue supports all operations in logarithmic worst-case time, uses only an array, supports insertion in constant average time, is simple to implement, and is known as a *binary heap*. This structure is one of the most elegant data structures known. In Chapter 21 we provide details on the implementation of the binary heap. The binary heap does not support the `decreaseKey` operation. However, the *pairing heap*, described in Chapter 23 does.

Data Structure	Access	Comments
Stack	Most recent only, `pop`, $O(1)$	Very very fast
Queue	Least recent only, `dequeue`, $O(1)$	Very very fast
List	Any item	$O(N)$
TreeSet	Any item by name or rank, $O(\log N)$	Average case easy to do; worst case requires effort
HashSet	Any item by name, $O(1)$	Average case
Priority Queue	`findMin`, $O(1)$, `deleteMin`, $O(\log N)$	`insert` is $O(1)$ on average $O(\log N)$ worst case

Figure 6.38 Summary of some data structures

An important use of priority queues is *event-driven simulation.*

An important application of the priority queue is *event-driven simulation*. Consider, for example, a system such as a bank in which customers arrive and wait in line until one of K tellers is available. Customer arrival is governed by a probability distribution function, as is the service time (the amount of time it takes a teller to provide complete service to one customer). We are interested in statistics such as how long on average a customer has to wait or how long a line might be.

With certain probability distributions and values of *K*, we can compute these statistics exactly. However, as *K* gets larger, the analysis becomes considerably more difficult, so the use of a computer to simulate the operation of the bank is appealing. In this way the bank's officers can determine how many tellers are needed to ensure reasonably smooth service. An event-driven simulation consists of processing events. The two events here are (1) a customer arriving and (2) a customer departing, thus freeing up a teller. At any point we have a collection of events waiting to happen. To run the simulation, we need to determine the *next* event, this is the event whose time of occurrence is minimum. Hence we use a priority queue that extracts the event of minimum time to process the event list efficiently. We present a complete discussion and implementation of event-driven simulation in Section 13.2.

Summary

In this chapter we examined the basic data structures that will be used throughout the book. We provided generic protocols and explained what the running time should be for each data structure. We also described the interface provided by the Collections API. In later chapters we show how these data structures are used and eventually give an implementation of each data structure that meets the time bounds we have claimed here. Figure 6.38 summarizes the results that will be obtained for the generic `insert`, `find`, `remove` sequence of operations.

Chapter 7 describes an important problem-solving tool known as *recursion*. Recursion allows many problems to be efficiently solved with short algorithms and is central to the efficient implementation of a sorting algorithm and several data structures.

Objects of the Game

`Arrays` Contains a set of static methods that operate on arrays. (318)

binary heap Implements the priority queue in logarithmic time per operation using an array. (348)

binary search tree A data structure that supports insertion, removal, and searching. We can also use it to access the *K*th smallest item. The cost is logarithmic average-case time for a simple implementation and logarithmic worst-case time for a more careful implementation. (334)

`Collection` Interface that represents a group of objects, known as its elements. (304)

`Collections` Class contains a set of static methods that operate on `Collection` objects. (310)

data structure A representation of data and the operations allowed on that data, permitting component reuse. (294)

factory method A method that creates new concrete instances, but returns them using a pointer (or reference) to an abstract class. (303)

BOOK.mkr Page 355 Wednesday, March 14, 2001 1:11 PM

`hashCode` Method used by `HashSet` that must be overridden for objects if the object's `equals` method is overridden. (341)

`HashMap` The Collections API implementation of a `Map` with unordered keys. (342)

`HashSet` The Collections API implementation of a (unordered) `Set`. (334)

iterator An object that allows access to elements in a container. (307)

`Iterator` An object that allows access to elements in a container. (307)

list a collection of items in which the items have a position. (318)

`List` The Collections API interface that specifies the protocol for a list. (320)

`ListIterator` The Collections API interface that provides bidirectional iteration. (321)

linked list A data structure that is used to avoid large amounts of data movement. It uses a small amount of extra space per item. (323)

`LinkedList` The Collections API class that implements a linked list. (323)

`Map` The Collections API interface that abstracts a collection of pairs consisting of keys and their values and maps keys to values. (342)

`Map.Entry` Abstracts the idea of a pair in a map. (346)

operator precedence parsing An algorithm that uses a stack to evaluate expressions. (329)

priority queue A data structure that supports access of the minimum item only. (347)

programming to an interface The technique of using classes by writing in terms of the most abstract interface. Attempts to hide even the name of the concrete class that is being operated on. (301)

queue A data structure that restricts access to the least-recently-inserted item. (329)

`Set` The Collections API interface that abstracts a collection with no duplicates. (331)

`SortedSet` The Collections API interface that abstracts a sorted set with no duplicates. (331)

stack A data structure that restricts access to the most-recently-inserted item. (325)

`TreeMap` The Collections API implementation of a `Map` with ordered keys. (342)

`TreeSet` The Collections API implementation of a `SortedSet`. (333)

Common Errors

1. Do not worry about low-level optimizations until after you have concentrated on basic design and algorithmic issues.
2. When you send a function object as a parameter, you must send a constructed object, and not simply the name of the class.

3. When using a `Map`, if you are not sure if a key is in the map, you may need to use `containsKey` rather than checking the result of `get`.
4. A priority queue is not a queue. It just sounds like it is.

On the Internet

There is lots of code in this chapter. Test code is in the root directory, nonstandard protocols are in package `weiss.nonstandard`, and everything else is in package `weiss.util`.

Stack.java	Contains the nonstandard protocol in Figure 6.21.
UnderflowException.java	Contains a nonstandard exception.
Queue.java	Contains the nonstandard protocol in Figure 6.23.
Collection.java	Contains the code in Figure 6.9.
Iterator.java	Contains the code in Figure 6.10.
Collections.java	Contains the code in Figures 6.13 and 6.14.
Arrays.java	Contains the code in Figure 6.15.
List.java	Contains the code in Figure 6.16.
ListIterator.java	Contains the code in Figure 6.17.
TestArrayList.java	Illustrates the `ArrayList`, as in Figure 6.18.
Set.java	Contains the code in Figure 6.24. The online code contains an extra method that is not part of Java 1.2.
Sorted.java	Contains the code in Figure 6.25.

TreeSetDemo.java	Contains the code in Figures 6.11 and 6.26.
IteratorTest.java	Contains the code that illustrates all the iterators, including code in Figures 6.11, 6.27 and 6.30.
EqualsWithInheritance.java	Contains the code in Figures 6.28 and 6.29, combined as one.
Map.java	Contains the code in Figures 6.31 and 6.32.
MapDemo.java	Contains the code in Figure 6.33.
PriorityQueue.java	Contains the nonstandard protocol in Figures 6.35 and 6.36.
PriorityQueueDemo.java	Contains the code in Figure 6.37.

Exercises

In Short

6.1. Show the results of the following sequence: `add(4)`, `add(8)`, `add(1)`, `add(6)`, `remove()`, and `remove()` when the `add` and `remove` operations correspond to the basic operations in the following:

a. stack

b. queue

c. priority queue

In Theory

6.2. Suppose that you want to support the following three operations exclusively: `insert`, `findMax`, and `deleteMax`. How fast do you think these operations can be performed?

6.3. Can all of the following be supported in logarithmic time: `insert`, `deleteMin`, `deleteMax`, `findMin`, and `findMax`?

6.4. Which of the data structures in Figure 6.38 lead to sorting algorithms that could run in less than quadratic time (by inserting all items into the data structure and then removing them in order)?

6.5. Show that the following operations can be supported in constant time simultaneously: `push`, `pop`, and `findMin`. Note that `deleteMin` is not part of the repertoire. *Hint*: Maintain two stacks — one to store items and the other to store minimums as they occur.

6.6. A double-ended queue supports insertions and deletions at both the front and end of the line. What is the running time per operation?

In Practice

6.7. Write a routine that uses the Collections API to print out the items in any `Collection` in reverse order. Do not use a `ListIterator`.

6.8. Show how to implement a `Stack` efficiently by using a `List` as a data member.

6.9. Show how to implement a `Queue` efficiently by using a `List` as a data member.

Programming Projects

6.10. A queue can be implemented by using an array and maintaining the current size. The queue elements are stored in consecutive array positions, with the front item always in position 0. Note that this is not the most efficient method. Do the following:

a. Describe the algorithms for `getFront`, `enqueue`, and `dequeue`.

b. What is the Big-Oh running time for each of `getFront`, `enqueue`, and `dequeue` using these algorithms?

c. Write an implementation that uses these algorithms using the protocol in Figure 6.23.

6.11. The operations that are supported by the `SortedSet` can also be implemented by using an array and maintaining the current size. The array elements are stored in sorted order in consecutive array positions. Thus `contains` can be implemented by a binary search. Do the following:

a. Describe the algorithms for `add` and `remove`.

b. What is the running time for these algorithms?

c. Write an implementation that uses these algorithms, using the protocol in Figure 6.1.

d. Write an implementation that uses these algorithms, using the standard `SortedSet` protocol.

6.12. A priority queue can be implemented by using a sorted array (as in Exercise 6.11). Do the following:

a. Describe the algorithms for `findMin`, `deleteMin`, and `insert`.

b. What is the Big-Oh running time for each of `findMin`, `deleteMin`, and `insert` using these algorithms?

c. Write an implementation that uses these algorithms.

6.13. A priority queue can be implemented by storing items in an unsorted array and inserting items in the next available location. Do the following:

a. Describe the algorithms for `findMin`, `deleteMin`, and `insert`.

b. What is the Big-Oh running time for each of `findMin`, `deleteMin`, and `insert` using these algorithms?

c. Write an implementation that uses these algorithms.

6.14. By adding an extra data member to the priority queue class in Exercise 6.13, you can implement both `insert` and `findMin` in constant time. The extra data member maintains the array position where the minimum is stored. However, `deleteMin` is still be expensive. Do the following:

a. Describe the algorithms for `insert`, `findMin`, and `deleteMin`.

b. What is the Big-Oh running time for `deleteMin`?

c. Write an implementation that uses these algorithms.

6.15. By maintaining the invariant that the elements in the priority queue are sorted in nonincreasing order (that is, the largest item is first, the smallest is last), you can implement both `findMin` and `deleteMin` in constant time. However, `insert` is expensive. Do the following:

a. Describe the algorithms for `insert`, `findMin`, and `deleteMin`.

b. What is the Big-Oh running time for `insert`?

c. Write an implementation that uses these algorithms.

6.16. A double-ended priority queue allows access to both the minimum and maximum elements. In other words, all of the following are supported:

`findMin`, `deleteMin`, `findMax`, and `deleteMax`. Do the following:

a. Describe the algorithms for `findMin`, `deleteMin`, `findMax`, `deleteMax`, and `insert`.

b. What is the Big-Oh running time for each of `findMin`, `deleteMin`, `findMax`, `deleteMax`, and `insert` using these algorithms?

c. Write an implementation that uses these algorithms.

6.17. A median heap supports the following operations: `insert`, `findKth`, and `removeKth`. The last two find and remove, respectively, the *K*th smallest element. The simplest implementation maintains the data in sorted order. Do the following:

a. Describe the algorithms that can be used to support median heap operations.

b. What is the Big-Oh running time for each of the basic operations using these algorithms?

c. Write an implementation that uses these algorithms.

6.18. Write a program that reads strings from input and outputs them sorted, by length, shortest string first. If a subset of strings that have the same length, output them in alphabetical order.

6.19. `Collections.fill` takes a `List` and a `value`, and places `value` in all positions in the list. Implement `fill`.

6.20. `Collections.reverse` takes a `List` and reverses its contents. Implement `reverse`.

References

References for the theory that underlies these data structures are provided in Part IV. The Collections API is described in most recent Java books (see the references in Chapter 1).

BOOK.mkr Page 364 Wednesday, March 14, 2001 1:11 PM