Fundamental to computer science is the development of algorithms. This chapter presents the standard algorithms to search for a specific item in an array and to sort the items in an array into increasing or decreasing order. As there are many sort algorithms, only a few simple ones are given in this chapter. More sorting algorithms are given in Chapter 18.

Another major topic of this chapter is the time required to execute a program. As the amount of data read into a program becomes larger, typically the amount of time required to execute this program tends to grow. For most programs, even moderately large amounts of data are not a problem as computers are extremely fast. In some applications, however, the computer controls some crucial task that requires almost immediate response time. In these applications, any growth in the time requirements is important. Also, some programs require long periods of time—as long as years—for even relatively small amounts of data. Thus, it is important to be able to estimate the time requirements of a computer program. The obvious approach to obtaining an estimate is to run the program to see how long it takes. For
small applications, this may be acceptable; however, large systems require years to develop. A company would not last long building such systems if the delivered system subsequently required too much time to be usable. Therefore, analytic techniques are needed to estimate time requirements of algorithms before they are coded into programs. This chapter presents the basic techniques to do this analysis. In later chapters, several other techniques are given to handle more complex situations.

The section on the estimation of time requirements is given first so that it can be applied to the searching and sorting algorithms in the rest of the chapter.

5.1 Algorithm Analysis

Recall that an algorithm is a sequence of steps that can be used to attain some goal. When the term algorithm is used, it is usually implied that the steps are specified in an implementation-independent manner. A program generally refers to an implementation of an algorithm or algorithms in some programming language. However, this distinction is not always followed. In particular, when doing quality analysis, the term algorithm is generally used, as often the objective is to estimate the quality of an algorithm independent of a particular implementation.

As soon as we can design algorithms and programs, it is important to learn how to analyze them. This is an important part of being a computer scientist. Analysis can take several forms. First, there is the question of the correctness of the algorithm that is ultimately implemented as a program. This can be determined by tracing the algorithm, implementing and testing it on some data, or using mathematical techniques to prove it correct. Another question relates to the simplicity of the algorithm. Perhaps the algorithm can be expressed in a simpler way so that it is easier to implement and do other analysis of it. However, the most straightforward way of solving a problem is sometimes not the best one. This usually occurs when the simplest approach involves the use of too much computer time and/or space. Time and space efficiency is especially important for algorithms on arrays and other data structures as large amounts of data are stored in them.

Various aspects of software quality were discussed in Section 1.3. Because a program is merely the implementation of many algorithms working together to accomplish some goal, much of what applies to software quality also applies to algorithm quality. To recap, the main aspects of quality software as related to algorithms are as follows:

- simplicity
- readability
- verifiability
- validity
- robustness
- modifiability
- reusability
- efficiency

The quality of efficiency is more objective than most of the others. As computers handle larger and larger problems, it is important to be able to analyze the time and space
/** Return the largest value in the array. */
public static short getMax(short[] array)
{
    short largest = array[0];
    for (int i = 1; i < array.length; i++)
        if (array[i] > largest)
            largest = array[i];
    return largest;
}

Figure 5.1. Code for finding the largest value

requirements of an algorithm to see if they are within acceptable limits. For example, if an
algorithm to control the trajectory of a rocket required several seconds for each trajectory
adjustment, it would be unacceptable. Analysis of time and space requirements is also im-
portant for comparison of algorithms to determine the best one. Measuring time and space
efficiency is the topic of this section.

Measuring the running time and space utilization of a program can be measured for
sample sets of input data. Conversely, analysis and comparisons of algorithms and programs
can be done without running the programs. Big-O notation is introduced as a vehicle for
describing the rate of growth of an algorithm’s time requirements as a function of its input
size. At the end, there is a discussion of experimental comparison of algorithms by timing
program execution.

5.1.1 Analytical Timing Analysis

This section introduces some techniques to estimate the time requirements of an algorithm
without needing to implement it. This approach has the obvious advantage of not requiring
the work to implement the algorithm. For small algorithms, such as the ones to be studied
in this section, it is a small task to implement them. However, the implementation of major
systems can require the equivalent of a person working for several years. If an analytical
analysis can show that the algorithm is too inefficient to ever be used, then the major task
of implementing the system can be avoided.

Time requirements for most algorithms can be estimated easily by analytical techniques.
In this section, techniques are developed for simple algorithms, and the way to apply them
to larger systems is presented. To show timing analysis for real programs, most of the
algorithms of this section are actually coded as Java methods. However, the same approaches
to timing can be used for algorithmic notation so that the analysis can be done before
implementation.

Can All Methods Be Run Quickly?

One reaction to timing analysis is that computers are so fast that any algorithm can be run
quickly. Hence, there is no need to bother with timing analysis. Figures 5.1, 5.2, and 5.3
give three methods. For now, it is not important what they do, but they all process values
in an array. The first two methods have arrays of type short rather than int to reduce
the space requirements when handling a large number of values. Also, all three methods
have static in their header as they are located in the main class and called from main().
The third one looks different as it calls itself. Such algorithms are called recursive and are
studied in Chapter 10. Because none of the methods has many statements, it might seem
that a computer should not take long to execute them. Nevertheless, they were timed on
/** Performs a selection sort on the array. */
public static void selectionSort(short[] array)
{
    int smallIndex, i, j;
    short temp;
    for (i = 0; i < array.length - 1; i++)
    {
        smallIndex = i;
        for (j = i + 1; j < array.length; j++)
        {
            if (array[j] < array[smallIndex])
                smallIndex = j;
        }
        if (smallIndex != i)
        {
            temp = array[i];
            array[i] = array[smallIndex];
            array[smallIndex] = temp;
        }
    }
}

Figure 5.2. Code for a selection sort

/** Find all permutations of the last p elements in the array. */
public static void permute(char[] array, int p)
{
    int i, s;
    char temp;
    if (p != 1)
    {
        s = array.length - p;
        /* For each possible character in position s, permute the remaining characters. */
        permute(array, p - 1);
        for (i = s + 1; i < array.length; i++)
        {
            temp = array[s];
            array[s] = array[i];
            array[i] = temp;
            permute(array, p - 1);
        }
        temp = array[s];
        /* Left shift each character back to its original position. */
        for (i = s; i < array.length - 2; i++)
        {
            array[i] = array[i + 1];
            array[array.length - 1] = temp;
        }
    }
}

Figure 5.3. Code for character permutations

different sizes for the array, as it seems to make sense that more values might require more time. Table 5.1 gives the results. In this table, the abbreviations ms, s, h, d, and y denote milliseconds, seconds, hours, days, and years, respectively. A question mark appears behind estimated values.

The first method was as fast as might have been expected; but even here, if a user is sitting waiting for this method to execute when the array has a size much more than 100
Table 5.1. Times Required for Three Methods

<table>
<thead>
<tr>
<th># of items</th>
<th>First routine</th>
<th>Second routine</th>
<th>Third routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0 ms</td>
<td>0 ms</td>
<td>1</td>
</tr>
<tr>
<td>500</td>
<td>0 ms</td>
<td>2 ms</td>
<td>3</td>
</tr>
<tr>
<td>1,000</td>
<td>0 ms</td>
<td>10 ms</td>
<td>4</td>
</tr>
<tr>
<td>5,000</td>
<td>0 ms</td>
<td>0.27 s</td>
<td>5</td>
</tr>
<tr>
<td>10,000</td>
<td>1 ms</td>
<td>1.06 s</td>
<td>6</td>
</tr>
<tr>
<td>20,000</td>
<td>1 ms</td>
<td>4.46 s</td>
<td>7</td>
</tr>
<tr>
<td>50,000</td>
<td>1 ms</td>
<td>28.15 s</td>
<td>8</td>
</tr>
<tr>
<td>100,000</td>
<td>2 ms</td>
<td>113 s</td>
<td>9</td>
</tr>
<tr>
<td>500,000</td>
<td>17 ms</td>
<td>1.09 h</td>
<td>10</td>
</tr>
<tr>
<td>1,000,000</td>
<td>33 ms</td>
<td>4.93 h</td>
<td>11</td>
</tr>
<tr>
<td>2,000,000</td>
<td>66 ms</td>
<td>22.3 h?</td>
<td>12</td>
</tr>
<tr>
<td>50,000,000</td>
<td>1.7 s</td>
<td>1.7 y?</td>
<td>13</td>
</tr>
<tr>
<td>100,000,000</td>
<td>3.3 s</td>
<td>7.9 y?</td>
<td>14</td>
</tr>
<tr>
<td>200,000,000</td>
<td>6.5 s?</td>
<td>32 y?</td>
<td>15</td>
</tr>
<tr>
<td>400,000,000</td>
<td>13 s?</td>
<td>120 y?</td>
<td>16</td>
</tr>
</tbody>
</table>

million, the user would start to get impatient. Although 100 million might seem like a value that would never occur in practice, think about the number of Visa credit cards in the world. The second method is slower to the extent that the number of values had better not approach 1 million. The numbers for the third method are not typing errors; this method really takes hours for a size 14 array. For a size 17 array, years would be required for the third method’s execution time (if you wanted to wait). Certainly, by a careful implementation and a faster machine, the times in the table could be reduced somewhat. However, such changes are not going to make significant improvements in the times for the third method. Thus, it is obvious that some methods require much more time than others, and some require so much time that they can only be used on small problems. Also, the hypothesis that the time required grows with the number of values being analyzed seems to be valid for these methods.

As a point of interest, the first method finds the maximum value in an array, the second sorts an array into ascending order, and the third finds all permutations of a set of characters. With this knowledge, it is not surprising that the permutation method takes a lot of time. For an array of size \( n \), there are \( n! \) permutations to be found by the method. Observe that \( n! \) grows fast with respect to \( n \) (\( 10! = 3,628,800 \), \( 14! = 87,178,291,200 \), \( 17! = 355,687,428,096,000 \)), so even a fast modern computer cannot generate (in a reasonable time) the permutations on a small-sized character set.

Basic Timing Analysis

It is important to realize that a general estimate of the time required for an algorithm is usually sufficient. Certainly no attempt is made to compute the time to the nearest millisecond. Even if it could be done, the result would change with every computer model and every compiler. The objective is to determine the order of magnitude for the time requirements so that the result is independent of a particular compiler, machine, and programming language.

As an example of the basic techniques of algorithm analysis, the method of Figure 5.1 is analyzed. It is easy to see that the algorithm finds the maximum value in an array, providing
that there is at least one value in the array.

A simple approach to timing is to count how many statements are executed. Not all statements require the same amount of time, but they are approximately equal provided that the statements do not involve method calls or particularly time-consuming operations. Thus, a count of the number of statements executed gives a good estimate of the time required (when there are no method calls or time-consuming operations). For an algorithm to find the maximum value in an array, the count obviously depends on the size of the array.

Figure 5.4 repeats the body of the method of Figure 5.1 and places beside each statement a count of the number times each statement is executed. As a shorthand, \( n \) is used instead of \( \text{array.length} \). Note that the loop body is done when \( i = 1 \) to \( i = n - 1 \) for a total of \( n - 1 \) times. The \( \textbf{for} \) loop test is done \( n \) times as it is tested every time through the loop and tested one more time when the loop is exited. From the information available, it is impossible to tell how many times the assignment in the \( \textbf{if} \) statement is done. Let us assume \( p \) is used to represent this value. Then the total number of statements executed is given by

\[
1 + 1 + n + (n - 1) + (n - 1) + p + 1 = 3n + p + 1.
\]

It is easy to see that \( 0 \leq p \leq (n - 1) \), so the most statements are executed when \( p = (n - 1) \). Let \( T S_{\text{max}}(n) \) be the function for the maximum number of statements that will ever be executed by the algorithm, where \( T \) denotes time, \( S \) stands for statement, and \( \text{max} \) is an abbreviation for the name of the algorithm. Then

\[
T S_{\text{max}}(n) = 4n.
\]

From this analysis, it is clear that the number of statements executed grows linearly with \( n \). Thus, the time requirements are also expected to grow linearly with \( n \). In fact, the times given in Table 5.1 for the time requirements of the algorithm grow linearly with \( n \). For example, note that the time required approximately doubles as \( n \) increases from 500,000 to 1,000,000 and 2,000,000. As previously remarked, this approach works because each statement requires about the same amount of time. If a statement or statements required significantly more time, as, for example, in the addition of two numbers each of length a million digits, such statements would need to be analyzed separately.

Another approach to timing is often quicker and simpler; an operation is found that is executed as often as any other operation, or at least within a constant of being the most frequent. Such an operation is called the \textit{active operation}. It is usually possible to take an operation that is central to the algorithm as the active operation. For the previous algorithm, the test \( i < \text{array.length} \) is done most frequently, whereas the test \( \text{array}[i] > \text{largest} \) is more central to the algorithm and is only done one less time. Either of these could be selected as the active operation, but \( \text{array}[i] > \text{largest} \) is used, as it is more
central to the algorithm. Let $T_{A_{\max}}(n)$ be the function for the number of times that the active operation is done as a function of $n$. Note the use of $A$ since this is an active operation count. Then

$T_{A_{\max}}(n) = n - 1$.

Because different things are being counted, the two functions $T_{S_{\max}}$ and $T_{A_{\max}}$ are not the same. However, they both grow linearly with the number of values.

Note that this measure of timing also assumes that all operations require about the same amount of time. If some operation requires a lot more time (e.g., a disk write operation), it should be the active operation, even though it might not be anywhere near the most frequently performed operation. However, most of the time, it is safe to assume that all operations require about the same amount of time. Also, sometimes it is necessary to have more than one active operation. In such a case, all active operations need to be counted. Examples are given later.

In most situations, it is good enough to know the rate of growth as a function of the size of the data. Either counting the statements executed or counting the active operations executed will measure this growth. As it is usually easier to count active operations, this approach will generally be used.

**Basic Growth Notation**

In this book, many algorithms are developed and their time requirements analyzed. These requirements are expressed as a function, whose rate of growth depends on the size of the input. It is useful to have notation to express rates of growth and to group together those functions with similar rates of growth. Note that for the function $T_{A_{\max}}(n) = 4n - 1$, the $-1$ does not affect the rate of growth, and neither does the $4$ in $T_{S_{\max}}(n) = 4n$. Both $T_{S_{\max}}(n) = 4n$ and $T_{A_{\max}} = n - 1$ should be grouped together with the functions that grow at most linearly—that is, functions $t$ such that there exists a constant $c > 0$ so that

$t(n) \leq c \cdot n$ for all $n > 0$.

Sometimes for small values of $n$, the counting functions behave differently than for large values of $n$. Generally, the time requirements only matter when $n$ is large enough that a significant amount of time is required by the algorithm. Thus, for the linear class of functions, it should contain functions $t$ such that there exists constants $c > 0$ and $n_0 > 0$ so that

$t(n) \leq c \cdot n$ for all $n \geq n_0$.

The function $t(n) = 4n + 2$ fits into this class of linear functions using $c = 5$ and $n_0 = 2$, since

$t(n) = 4n + 2 \leq 5n$ for $n \geq 2$.

Hence, the set of functions that grow at most linearly, denoted by $O(n)$, is defined as follows:

$O(n) = \{\text{function } t \mid \text{there exists constants } c > 0 \text{ and } n_0 > 0, \text{ such that } t(n) \leq c \cdot n \text{ for all } n \geq n_0\}$.

A set containing functions is somewhat unusual to many people. However, a function is a well-defined thing, and there is no reason that a collection of them cannot be put into a set. Note that the following are true:

$n - 1 \in O(n)$
4n - 1 ∈ O(n)
n + 1000 ∈ O(n)
1000n + 500 ∈ O(n)
100 ∈ O(n)  
(Notes that functions in O(n) can grow slower than n.)
log_{10}(n) + 300 ∈ O(n)
log_{10}(log_{10}(n)) ∈ O(n)
[log_{10}(n)]^2 ∈ O(n)

The last three relationships depend on properties of the log to the base 10 function. The key property is that any constant power of log_{10}(n) still grows slower than n.

Of course, not all algorithms grow linearly relative to the amount of data. Therefore, notation is needed for other rates of growth.

**Definition 5.1:** For any function f that maps nonnegative integers to nonnegative real numbers, the set of functions that grow at most as fast as f, notated by O(f(n)) and pronounced “big Oh of f(n),” is given by

\[ O(f(n)) = \{ \text{function } t \mid \text{there exists constants } c > 0 \text{ and } n_0 > 0, \]
\[ \text{such that } t(n) \leq c \times f(n) \text{ for all } n \geq n_0 \}. \]

When \( t(n) \in O(f(n)) \), it is stated that \( t(n) \) is in the order of \( f(n) \).

Sometimes, “\( t(n) \) is in the order of \( f(n) \)” is abbreviated to “\( t(n) \) is order \( f(n) \)” or “\( t(n) \) has order \( f(n) \).” Here are some examples of using this notation:

\( n^2 + 30 \in O(n^2) \)
\( 2n^3 + 100n^2 + 30 \in O(n^3) \)
\( 250 \in O(1) \)  
(Neither of these functions grow at all.)
\( 5n^2 + 100n + 500 \in O(n^3) \)  
(Again functions in \( O(n^3) \) can grow slower than \( n^3 \).)
\( n \times (\log_{10}(n)) \in O(n^2) \)
\( n \in O(n \times (\log_{10}(n))) \)
\( n^k \in O(2^n) \)  
(for any constant \( k \))
\( 2^n \in O(n!) \)
\( n! \in O(n^n) \)

As an example, consider the verification that \( 2n^3 + 100n^2 + 30 \in O(n^3) \). An easy way to do this is as follows:

\[ 100n^2 \leq 100n^3 \text{ and } 30 \leq 30n^3 \text{ for } n \geq 1, \]

therefore, \( 2n^3 + 100n^2 + 30 \leq 2n^3 + 100n^3 + 30n^3 = 132n^3 \) for \( n \geq 1 \). Thus, the criterion is satisfied with \( c = 132 \) and \( n_0 = 1 \). Note that many other values of \( c \) and \( n_0 \) will work (e.g., \( c = 3 \) and \( n_0 = 101 \)).

It is useful to have some idea how fast various functions grow. Figure 5.5 shows the growth of several common functions. To facilitate the comparison of these functions, we have used log scales for both axes in the figure. As is readily apparent, some functions grow much faster than others. The differences become even more apparent when a table is produced that computes the time required by a fast computer to do the number of operations specified by functions of varying growth rates. Assume that a computer executes 1 million operations per second. Table 5.2 gives the time required by the computer to do \( t(n) \) operations for different values of \( n \) and different functions \( t \). From this table, for a data
set of size 1 million, the rate of growth of the time requirements had better be less than $n^2$, or too much time will be required. If the growth is exponential (i.e., $n$ appears in the exponent), then the size of the data set had better be in the teens or less.

**Warning:** Our notation, $t(n) \in O(f(n))$ (first suggested by Brassard [6]) is usually written as $t(n) = O(f(n))$ and verbally expressed as “$t(n)$ is of the order of $f(n)$.” The use of the equality symbol tends to be confusing as the two are certainly not equal. In fact, the relationship is not even symmetric, as $O(f(n)) = t(n)$ is not even defined, let alone meaning the same as $t(n) = O(f(n))$. Thus, this book uses the set notation, even though it is nonstandard. In program comments, = will be used, as there is no $\in$ symbol on an English keyboard.
For those readers with some background in calculus, there is an alternative way of defining $t(n) \in O(f(n))$. The definition is $t(n) \in O(f(n))$ if

$$\lim_{n \to \infty} \frac{t(n)}{f(n)} < \infty.$$  

### Analysis of the Selection Sort Algorithm

So far, the `getMax` method of Figure 5.1 has been analyzed, and notation has been defined to express growth of a function. In this subsection, the sort algorithm of Figure 5.2, known as the selection sort algorithm, is analyzed.

The algorithm, reproduced in Figure 5.6, assumes that the index range is from 0 to $n-1$. The algorithm begins with $i = 0$, finds the smallest value in the array, and moves it to `array[0]`. The move is only done if the smallest value was not already in position 0, and the move is done by having the value initially in position 0 switch positions with the smallest value. With the smallest value in position 0, the algorithm goes on to consider position 1. This is done by finding the smallest value in positions 1 through the end and moving it to position 1. Then each of the other positions is considered in turn. In general, when considering the $i$th position, the smallest value from positions $i$ through the end is found. If this smallest value is not already in position $i$, it swaps position with the value presently in position $i$. When all the positions have been considered (except position $n-1$, which by the time $i = n-1$ is guaranteed to have the largest value), the array is sorted.

The analysis of the time requirements of the algorithm is done by counting the executions of each statement. As the algorithm has nested loops, it is a little harder to count the number of times that a statement is done. Summation notation is used to assist in this task; recall

for (int i = 0;
    i < n - 1;
    i++)
{
    smallIndex = i;
    for (int j = i+1;
        j < n;
        j++)
    {
        if (array[j] < array[smallIndex])
            smallIndex = j;
    }
    if (smallIndex != i)
    {
        temp = array[i];
        array[i] = array[smallIndex];
        array[smallIndex] = temp;
    }
}
that for any integer function \( t(n) \), the following are defined to be equal:

\[
\sum_{i=r}^{s} t(i) = t(r) + t(r + 1) + t(r + 2) + \ldots + t(s - 1) + t(s).
\]

Refer to Appendix C for a more detailed review of summation notation.

Note that the initialization statement, the first one in the header of a \texttt{for} loop, is only done once each time the loop is reached. Thus, it is outside of the loop, and the count for it should be outside the summation for the loop. The \texttt{boolean} expression that controls the loop is tested once each time through the loop, plus one more time when the loop is exited. Hence, it must be counted once inside the loop body summation and once outside. The increment statement is done once for every time through the loop.

Returning to the analysis of the selection sort, the algorithm is repeated in Figure 5.6 with an execution count beside each statement. First, consider the inner loop in which there are two \texttt{boolean} expression evaluations and one increment statement that are done each time through the loop. This loop also contains an assignment that is sometimes done. This can be expressed as \( 3 + \lceil 0 \text{ or } 1 \rceil \). The inner loop is done from \( j = i + 1 \) up to and including \( j = n - 1 \). This can be expressed as a summation. Thus, the number of statements executed in the inner loop for a certain \( i \) is given by

\[
\sum_{j=i+1}^{n-1} (3 + \lceil 0 \text{ or } 1 \rceil) \leq \sum_{j=i+1}^{n-1} 4 = 4 \times ((n - 1) - (i + 1) + 1) = 4 \times (n - i - 1).
\]

Note that this result is a function of \( i \). This type of analysis can be repeated for the outer loop to develop its summation. In this loop, \( i \) progresses from 0 to \( n - 2 \), so the sum is

\[
\sum_{i=0}^{n-2} (5 + \text{count for inner loop} + 1 + \lceil 0 \text{ or } 3 \rceil) \leq \sum_{i=0}^{n-2} (9 + \text{count for inner loop}).
\]

Finally, two statements are outside both loops and only done once. Therefore, the total number of statements executed is given by

\[
2 + \text{count for outer loop} \leq 2 + \sum_{i=0}^{n-2} (9 + \text{count for inner loop}).
\]

Let \( TS_{ssort}(n) \) be the function for the maximum number statements that will ever be exe-
cuted by the selection sort algorithm. Then

\[
T_{S_{sort}}(n) = 2 + \sum_{i=0}^{n-2} (9 + \text{count for inner loop})
\]

\[
= 2 + \sum_{i=0}^{n-2} (9 + 4 \times (n - i - 1))
\]

\[
= 2 + \sum_{i=0}^{n-2} (5 + 4n - 4i)
\]

\[
= 2 + (5 + 4n) \times \sum_{i=0}^{n-2} (1) - 4 \times \sum_{i=0}^{n-2} (i)
\]

since \(\sum_{i=1}^{n} i = \frac{n(n+1)}{2}\)

\[
= 2 + 5n - 5 + 4n^2 - 4n - 2n^2 + 2n + 4n - 4
\]

\[
= 2n^2 + 7n - 7.
\]

Therefore, using the statement count approach, \(T_{S_{sort}}(n) \in O(n^2)\).

The same result can be obtained much easier by using the active operation approach. To do this, first the active operation must be determined. The operation that occurs most frequently must be executed every time through the inner loop. Thus, the obvious operation that qualifies is the test `array[j] < array[smallIndex]`. Let \(T_{A_{sort}}(n)\) be the number of times that the active operation is done by the selection sort when there are \(n\) values in the array. Then it follows that

\[
T_{A_{sort}}(n) = \sum_{i=0}^{n-2} \left( \sum_{j=i+1}^{n-1} 1 \right) = \sum_{i=0}^{n-2} (n - i - 1)
\]

\[
= \sum_{i=0}^{n-2} (n - 1) - \sum_{i=0}^{n-2} (i)
\]

\[
= (n - 1)(n - 1) - \frac{(n - 2)(n - 1)}{2} = \frac{n^2 - n}{2}.
\]

Thus, \(T_{A_{sort}}(n) \in O(n^2)\), and both \(T_{A_{sort}}(n)\) and \(T_{S_{sort}}(n)\) have the same order, as they should. Therefore, if \(T_{ssort}(n)\) is the time required by the selection sort when there are \(n\) values in the array, then \(T_{ssort}(n)\) is in the order of \(n^2\) [i.e., \(T_{ssort}(n) \in O(n^2)\)].

**Useful Summation Expressions**

As in the foregoing analysis of the selection sort, it is often useful to use summations. A review of summation notation is given in Appendix C. The closed form of several summations is presented in Table 5.3 for reference.
Table 5.3. Some Common Summation Expressions

\[
\begin{align*}
\sum_{i=1}^{n} 1 &= n \quad (5.1) \\
\sum_{i=1}^{n} i &= \frac{n(n+1)}{2} \quad (5.2) \\
\sum_{i=1}^{n} i^2 &= \frac{n(n+1)(2n+1)}{6} \quad (5.3) \\
\sum_{i=0}^{n} a^i &= \frac{a^{n+1} - 1}{a - 1} \quad \text{when } a \neq 1 \quad (5.4) \\
\sum_{i=1}^{n} i a^i &= \frac{na^{n+1}}{a - 1} - \frac{a(a^n - 1)}{(a-1)^2} \quad \text{when } a \neq 1 \quad (5.5)
\end{align*}
\]

More on Growth Notation and Expressions

Note that it is correct to say that \(T_{\text{ssort}}(n) \in O(n^3)\) and that \(T_{\text{ssort}}(n) \in O(n^5)\), but both are misleading as a more accurate order can be given. When an accurate order can be found, it is beneficial to use another notation that specifies that accuracy.

Definition 5.2: For any function \(f\) that maps nonnegative integers to nonnegative real numbers, the set of functions that grows at the same rate as \(f\), notated by \(\Theta(f(n))\) and pronounced “big Theta of \(f(n)\),” is given by

\[
\Theta(f(n)) = \{\text{function } t \mid \text{there exists constants } c, d > 0 \text{ and } n_0 > 0, \text{ such that } d \cdot f(n) \leq t(n) \leq c \cdot f(n) \text{ for all } n \geq n_0\}
\]

When \(t(n) \in \Theta(f(n))\), it is stated that \(t(n)\) is in the exact order of \(f(n)\).

An obvious implication of this definition is that if \(t(n) \in \Theta(f(n))\), then \(t(n) \in O(f(n))\). Some examples of using this new notation are

- \(T_{\text{Amax}}(n) = n - 1 \in \Theta(n)\)
- \(T_{\text{ssort}}(n) = 2n^2 + 7n - 7 \in \Theta(n^2)\)
- \(2n^3 + 100n^2 + 30 \in \Theta(n^3)\)
- \(T_{\text{ssort}}(n) = 2n^2 + 7n - 7 \notin \Theta(n^3)\)
- \(n \log_{10}(n) \notin \Theta(n^2)\) The left side grows slower than \(n^2\)

As \(\Theta(.)\) is more expressive than \(O(.)\), \(\Theta(.)\) should be used where possible.

It is often necessary to combine two order expressions. For example, suppose that an algorithm is in the order \(n \log(n)\) for stage 1, and stage 1 is followed by a stage 2, which is in the order of \(n^2\). Then the total time for the two stages is the sum of their time requirements. To handle this situation, the standard arithmetic operations can be defined on sets of numeric functions

\[
\begin{align*}
O(f(n)) + O(g(n)) &= O(f(n) + g(n)), \\
O(f(n)) \times O(g(n)) &= O(f(n) \times g(n)), \\
\Theta(f(n)) + \Theta(g(n)) &= \Theta(f(n) + g(n)).
\end{align*}
\]
Of course, when $O(.)$ and $\Theta(.)$ are mixed, only the weaker result is obtained. For example,

$$O(f(n)) + \Theta(g(n)) = O(f(n) + g(n)),$$

There is one more result that is useful for simplifying order sets. Since all that matters is the fastest growing term, it is easy to see that

$$O(f(n) + g(n)) = O(max(f(n), g(n))).$$

Now the time for the two-stage algorithm mentioned earlier is given by

$$O(n \cdot \log(n)) + O(n^2) = O(n \cdot \log(n) + n^2) = O(max(n \cdot \log(n), n^2)) = O(n^2).$$

The $\Theta(.)$ notation can also be expressed using limits.

**Timing Method Calls**

All the examples so far have timed one method. When several methods are involved, the same techniques are used except that individual statements cannot be assumed to take the same amount of time. If a statement involves a method call, when counting the number of statements executed, the count for one execution of the method call is the time for one execution of the method rather than one unit of time. The total time for the method call is obtained by multiplying the number of times the statement is done by the amount of time for one execution of the method.

Consider timing a complex program. The full program is given in Figure 5.7; the objective is to obtain the worst-case order for the time requirements of each of the methods of the program. As can be seen from the code, the program calculates several values for a matrix of integers. The matrix is implemented via an array of arrays.

We use the notation

- $h$ // the length of the main array (i.e., the number of rows or height of the matrix).
- $w$ // the length of each subarray (i.e., the number of columns or width of the matrix).

An approach that can be applied to most systems with several methods is to handle each method individually. To select the next method to analyze, choose a method that only calls methods that have already been analyzed. It is also convenient to handle the simplest methods first.

It is instructive to attempt this timing analysis before reading our development of it. If you find the analysis too difficult, read our analysis of the first one or two methods and then attempt the remainder yourself.

As mentioned before, to obtain the total count for a statement involving a method call, it is necessary to determine how often the statement is done and how much time is required for each execution of the statement. The total time for the statement is the sum of the execution times. If all executions take the same amount of time, the total is the number of executions times the time per execution. This analysis needs to be done for each statement that might require the most time (i.e., each possible active operation).

In the example of Figure 5.7, the main method is listed first. As it calls the constructor, its analysis must be postponed until after the analysis of the constructor. The constructor is listed next, but it must be analyzed second last, since it calls each of the other methods. The simplest method is $aItem()$, which does not call any other methods. Doing the analysis for function $aItem()$, it is easy to obtain the following:

Function $aItem()$: 

import java.io.*;
/** A collection of functions and procedures for timing. */
public class TimingArray
{
    /** A two-dimensional array (matrix). */
    public int[][] a;
    public static void main(String[] args) throws Exception
    {
        TimingArray c = new TimingArray();
    }
    /** Main program. */
    public TimingArray() throws Exception
    {
        createArray();
        System.out.print("nThe array is " + aString()
            + "\nItem (1,1) is " + aItem(1, 1)
            + "\nThe special sum is " + specialSum(a[0].length / 2)
            + "\nThe different sum is " + differentSum()
            + "\nThe big sum is " + bigSum() + "\n\n");
    }
    public int aItem(int i, int j)
    {
        return a[i][j];
    }
    public String aString()
    {
        String result = new String();
        for (int i = 0; i < a.length; i++)
            result += rowString(i);
        return result;
    }
    public String rowString(int i)
    {
        String result = "Row " + (i+1) + ": ";
        for (int j = 0; j < a[0].length; j++)
            result += a[i][j] + " ";
        return result;
    }
    public int bigSum()
    {
        int result = 0;
        int i, j, k;
        for (i = 0; i < a.length; i++)
            for (j = 0; j < a[0].length; j++)
                for (k = i; k < a.length; k++)
                    result += a[k][j];
        return result;
    }
}

Figure 5.7. A complex example for timing analysis (part 1)
public void createArray() throws Exception {
    int i, j, height, width;
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("Enter the height of the matrix: ");
    height = Integer.parseInt(br.readLine());
    System.out.print("Enter the width of the matrix: ");
    width = Integer.parseInt(br.readLine());
    int[][] a = new int[height][width];
    for (i = 0; i < a.length; i++)
        for (j = 0; j < a[0].length; j++)
            a[i][j] = i + 10 * j;
}

public int specialSum(int j) {
    int result = 0;
    for (int i = 1; i < a.length; i = 2*i)
        result += a[i][j];
    return result;
}

public int differentSum() {
    int i, j;
    int result = 0;
    for (i = 0, j = 0; i < a.length && j < a[0].length; i++, j++)
        result += a[i][0] * a[0][j];
    for (; i < a.length; i++)
        result += a[i][0] * a[i][0];
    for (; j < a[0].length; j++)
        result += a[0][j] * a[0][j];
    return result;
}

Figure 5.7. A complex example for timing analysis (part 2)

active operation: return a[i][j]
done: once
time per execution: Θ(1)

Remember from Section 4.4 that an array access can be done with one multiply and one addition. As a result, only a constant number of operations are required to access the \( i \)th subarray of the main array and the \( j \)th value of the subarray.

time complexity: Θ(1)

Procedure createArray() is a little more complex, as there are two possible active operations.

Procedure createArray():
active operation: a = new int[height][width]
done: once
time per execution: $\Theta(h \times w)$ since all $h \times w$ locations need to be initialized to 0

time complexity: $\Theta(h \times w)$
or
active operation: $a[i][j] = i + 10 \times j$
done: $\Theta(h \times w)$

time per execution: $\Theta(1)$ array access only needs a constant number of operations

time complexity: $\Theta(h \times w)$

Both active operations yield the same time bound, so it is the time bound for the method.

The next method, function $aString()$, cannot be analyzed next, since it calls function $rowString()$, for which analysis has not yet been done. Thus, $rowString()$ is done next:

Function $rowString()$:
active operation: $\text{result } += a[i][j] + "\"$
done: $\Theta(w)$

time per execution: $\Theta(1)$ assuming the concatenations can be done in a constant
number of operations, since the strings being concatenated all have a constant
length. We use this result, even though it may not be valid if the Java system
is not smart enough to use a large enough $\text{StringBuffer}$ to avoid multiple
string copies.
time complexity: $\Theta(w)$

Function $aString()$:
active operation: $\text{result } += rowString(i)$
done: $\Theta(h)$
time per execution: $\Theta(w)$

This statement calls function $rowString()$, which requires time $\Theta(w)$. The result
of the function call is appended onto $\text{result}$. The amount of time for the append is
$\Theta(w)$, since that is the measure of the length of the string being appended (assuming
the system uses a large enough $\text{StringBuffer}$). Thus, two $\Theta(w)$ operations are done
for a cost of $\Theta(w)$.

time complexity: $\Theta(h \times w)$

Next, consider function $specialSum()$. It is more difficult, since it is not obvious how
many times the loop is done, as each time through the loop $i$ is multiplied by 2. Thus, if
the loop is done $t$ times, when the loop is exited $i$ has the value $2^t$. But the loop is exited
when $2^t \geq h$, and the loop was not exited the previous time when $2^{t-1} < h$. Putting these
together, we obtain

\[ 2^{t-1} < h \leq 2^t \]
\[ t - 1 < \log_2 h \leq t \] taking log to the base 2 of the terms
\[ \log_2 h \leq t < 1 + \log_2 h \] rearranging the terms
i.e., $t \in \Theta(\log_2 h)$

Function $specialSum()$:
active operation: $\text{result } += a[i][j]$
done: $\Theta(\log_2 h)$
time per execution: $\Theta(1)$
time complexity: $\Theta(\log_2 h)$
Function `differentSum()` is complicated by having three loops. The times for these loops should be added together, since the loops are not nested. As it is not immediately clear which takes the most time, an active operation is selected for each one.

Function `differentSum()`:
- active operations: three possible active operations
  1. `result += a[i][0] * a[0][j]`
  2. `result += a[i][0] * a[i][0]`
  3. `result += a[0][j] * a[0][j]`
- done: number of times that each is done
  1. \( \min(h, w) \)
  2. \( \max(0, h - \min(h, w)) \)
  3. \( \max(0, w - \min(h, w)) \)
- time per execution: \( O(1) \) each active operation does a constant number of things.
- time complexity: add together the times for each active operation
  \[
  O(\min(h, w)) + O(\max(0, h - \min(h, w))) + O(\max(0, w - \min(h, w)))
  \]
  since constants do not matter, the first term can be included twice
  \[
  = O(\max(\min(h, w) + 0, \ min(h, w) + h - \min(h, w))
  + \max(\min(h, w) + 0, \ min(h, w) + w - \min(h, w)))
  \]
  since \( z + \max(a, b) = \max(z + a, z + b) \) and use \( z = \min(h, w) \)
  \[
  = O(\max(\min(h, w), h) + \max(\min(h, w), w))
  \]
  \[
  = O(\max(\min(h, w), h, w)) = O(\max(h, w)) = O(h + w)
  \]

There are three nested loops in function `bigSum()`. Since they are nested, the number of times one must be multiplied. Since the number of times for the innermost loop depends on the index of the outermost loop, it is not as easy as simply multiplying them. Thus, the summation approach is used to calculate the correct value.

Function `bigSum()`:
- active operation: `result += a[k][j]`
- done: number of times done is given by

\[
\begin{align*}
\sum_{i=0}^{h-1} \left( \sum_{j=0}^{w-1} \sum_{k=i}^{h-1} (1) \right) &= \sum_{i=0}^{h-1} \left( \sum_{j=0}^{w-1} (h - i) \right) \\
&= \sum_{i=0}^{h-1} \left( (h - i) \sum_{j=0}^{w-1} (1) \right) \\
&= \sum_{i=0}^{h-1} ((h - i) \times w) \\
&= \sum_{i=0}^{h-1} (h) \times w - \sum_{i=0}^{h-1} (i \times w) \\
&= h \times w \times h - w \times (h - 1)h/2 \\
&\in \Theta(w \times h^2)
\end{align*}
\]
time per execution: $\Theta(1)$

time complexity: $\Theta(w \times h^2)$

Finally, the analysis of the constructor needs to be done. Its time is estimated by adding the times for each of the method calls made from it.

Constructor **TimingArray**:
- active operations: any of the function or procedure calls
- done: each call is done once
- time per execution: varies with the method called
- time complexity: (Add together the times for each of the methods called.)

\[
\Theta(T_{createArray} + T_{aString} + T_{aItem} + T_{specialSum} + T_{differentSum} + T_{bigSum})
\]
\[
= \Theta(h \times w + h \times w + 1 + \log_2 h + (h + w) + w \times h^2)
\]
\[
= \Theta(w \times h^2)
\]

The last method is **main()**. Its time requirements are the same as those of the constructor **TimingArray** that it calls.

The approach used to time these methods relied on finding a method that only calls methods whose timing has already been done. It is not possible to do this for a recursive method—a method that calls itself. It is best to have practice with the analysis of nonrecursive methods before studying the techniques to handle recursive ones. Thus, the discussion of timing recursive methods is delayed until recursion is discussed in detail in Chapter 10. However, as long as no methods are recursive, the approach discussed in this section can be used to analyze large complex systems.

Part of the documentation of every method should be a comment in the code giving its time requirements. All methods in the rest of this book have a comment to this effect. If the time is dependent on some parameter $n$, be sure to state what $n$ measures.

An important benefit of doing a timing analysis is that it indicates where improvement is required if the algorithm needs to be speeded up. The key to making significant improvements in the time requirements of an algorithm is reducing the number of times that the dominating active operations are done. A timing analysis isolates the active operations and determines the dominating one(s), so that optimization can concentrate on reducing the number of times that they are done.

For example, in the previous program, there is little value in speeding up functions **aString()** or **specialSum()**, as their time does not dominate the overall time requirements. Probably any optimization effort should go into procedure **bigSum()**, especially in optimizing the innermost loop of that method. However, there is little that can be done to speed up this method.

### Problems 5.1.1

1. Which of the following functions are $O(n^3)$?

   (a) $f(n) = n^2 + n + 1$
   (b) $f(n) = n^3 + n \log n$
   (c) $f(n) = n + 2^n$
   (d) $f(n) = n^4 + n^2 \sqrt{n}$
   (e) $f(n) = n^5 + n^3 \log n$
   (f) $f(n) = n + 3^n$
public double method1(int[] arr) {
    double sum = 0;
    for (int i = 0; i < arr.length; i++)
        sum += Math.abs(arr[i] - average(arr));
    return sum / arr.length;
}

Figure 5.8. A function using function average()
/** The maximum sum found in any contiguous subvector. */
public int maxCon1(int[] x)
{
    int maxSoFar = 0;
    for (int lower = 0; lower < x.length; lower++)
    {
        int sum = 0;
        for (int k = lower; k <= upper; k++)
            sum += x[k];
        if (sum > maxSoFar)
            maxSoFar = sum;
    }
    return maxSoFar;
}

/** The maximum sum found in any contiguous subvector. */
public int maxCon2(int[] x)
{
    int maxSoFar = 0;
    for (int lower = 0; lower < x.length; lower++)
    {
        int sum = 0;
        for (int upper = lower; upper < x.length; upper++)
        {
            sum += x[upper];
            if (sum > maxSoFar)
                maxSoFar = sum;
        }
    }
    return maxSoFar;
}

Figure 5.9. Code for maxCon1 and maxCon2

u is an array with length m
v is an array with length n
r is a method with time bound Θ(log₂ n)
v is an int function with time bound Θ(m).

5.1.2 Experimental Timing Comparison

The analytical comparison of algorithms is useful, but it has limitations. First, some algorithms are difficult to analyze analytically. This is especially true when an algorithm’s typical expected time requirement is much better than its worst-case time requirement. Typical performance is often what is wanted, but for some algorithms it is difficult to express what typical means so that it can be used in an order analysis. Another limitation is that an analytical analysis only yields the order of the algorithm. If two algorithms have the same order, the analysis given here does not indicate which is better. To determine which is better, either a more precise analytical analysis is needed, or they need to be implemented and timed. Finally, for small values of n, an algorithm with a larger asymptotic growth rate for its time may execute faster than one with a slower growth rate. Thus, if you know that the algorithm will be only run on small amounts of data, a different algorithm might be
public class NumberList
{
    int[ ] number;
    int capacity;
    /** Constructor for the class. */
    public NumberList(int size)
    {
        number = new int[size];
        capacity = size;
    }

    public void printDuplicates()
    {
        for (int i = 0; i < capacity; i++)
            for (int j = 0; j < capacity; j++)
                if (j != i && number[j] == number[i])
                    System.out.println(number[i]);
    }

    public void fill(int r)
    {
        for (int i = 0; i < capacity; i++)
            number[i] = (i * 2 + 7) % r;
    }

    public boolean findNumber(int searchNumber)
    {
        boolean found = false;
        for (int i = 0; !found & (i < capacity); i++)
            if (number[i] == searchNumber)
                found = true;
        return found;
    }

    public void summarize(NumberList otherNumbers)
    {
        for (int i = 0; i < otherNumbers.capacity; i++)
        {
            int currentNumber = otherNumbers.number[i];
            if (findNumber(currentNumber))
                System.out.println(currentNumber);
        }
        printDuplicates();
    }
}

Figure 5.10. Methods to manipulate an array of integers

chosen. Again, such a determination can only be made by a more precise analytical analysis or by timing the implementations. In this book, more precise analytical analyses are not discussed, but the results of the actual timing of algorithms are given.

Timing an algorithm is not quite as straightforward as initially it might seem. The most important consideration is the data for the algorithm. Many algorithms have quite different time requirements on different data sets, even if the data sets have the same size. Hence, if possible, an algorithm should be timed on data sets that are representative of the
public int h(int x) {
    int i = 0;
    while ((i < m) || (u[i] != x)) {
        r();
        i++;
    }
    return i;
}
public void p() {
    for (int j = 1; j <= n; j = 2 * j)
        v[j-1] = h(j) + w(j);
    q(m);
}
public void q(int k) {
    u[k] = w(2 * k + 1);
    if (k - 1 > 0)
        q(k-1);
}

Figure 5.11. A function and procedure for timing analysis

application where they are to be used. The determination of such data sets can be a difficult task. Furthermore, even if such data sets can be determined, the algorithm should be timed on several representative data sets of the same size. If the times are all about the same, their average probably gives a good indication of the expected performance of the algorithm on the type of data that occur in the data sets. If there is a large variance in the times, more work may be necessary to determine why and what can be done about it.

5.1.3 Space Comparison

Another parameter that sometimes needs to be estimated for an algorithm is its run-time memory requirements. As with time requirements, the memory requirements often grow with the size of the data set, and it is sufficient to estimate the rate of growth. Generally, the same techniques that we have been using for timing analysis can be used for memory analysis. However, memory is becoming so inexpensive that most computers have a large amount of memory and an algorithm’s memory requirement is not usually an important consideration. Consequently, it is not discussed any further here. However, it is important to realize that some algorithms have exponential memory requirements, so that memory can become the bottleneck in terms of running the algorithm on larger data sets.

5.2 Searching

Searching is one of the most basic tasks of computer science. As soon as a collection of items exists—for example, in an array—the problem of finding some item of the collection exists. An example is the problem of finding the largest item of the collection. When the items are stored in an array, an algorithm was given for this task in Figure 5.1.

Whether an array contains integers, names, or complex objects, it is frequently necessary to search for a specific item. As this is a frequent task, and it can be moderately time
The index of the first occurrence of key in array (-1 if it doesn’t exist).

/** Time = O(n), n = length of the array */
public static int linSearch(short[] array, short key)
{
    for (int i = 0; i < array.length; i++)
        if (array[i] == key)
            return i;
    return -1;
}

Figure 5.12. Code for linear search

5.2.1 Linear Search

The standard search, called the linear search, scans sequentially through the items, testing each one, until it reaches the end or finds the desired item. A Java function to do this task is given in Figure 5.12, where array is the name of the array to be searched and key contains the value being sought. The function returns the index of the location containing the value in key or else -1.

The timing of the algorithm is not so straightforward. Suppose that \( n \) represents the number of items in the array. Now we apply the active operation counting approach to timing the algorithm.

A reasonable active operation here is the comparison between key and values in array. A problem arises, however, in counting the number of active operations executed. The answer depends on the data. The count can range from 1 to \( n \). The best-case, or least-time requirement case, is when key is in the first position of the array, since only one comparison is used. The worst case is when key is only equal to \( \text{array}[i] \) with \( i = \text{array.length} - 1 \). In this case, \( n \) comparisons are used. Thus, we obtain

\[
T_{LS}^B(n) = 1 \quad \text{and} \quad T_{LS}^W(n) = n,
\]

where \( T_{LS}^B \) and \( T_{LS}^W \) denote the best-case and the worst-case times for the linear search, respectively.

The important question is, “What time can be expected on the average?” To answer this question, we need to know the probability distribution for the value key in the array; that is, the probability of key occurring in each location. We assume that key is equally likely to be in each of the locations. Yet key might not be in the array at all. Let \( q \) be the probability that key is in the array. Then we have

- probability key is in location \( j \) is \( q / n \)
- probability key is not in the array is \( 1 - q \)

The average time is given by

\[
T_{LS}^A(n) = \sum_{s \in S} (\text{probability of situation } s) \times (\text{time for situation } s),
\]

where \( S \) is the set of all possible situations. Thus, for the preceding algorithm, using the
active operation count for timing, we have

\[ T_{\text{LS}}^A(n) = \left( \text{probability of key in location 0} \right) \times 1 \\
+ \left( \text{probability of key in location 1} \right) \times 2 \\
+ \cdots \\
+ \left( \text{probability of key in location } n-1 \right) \times n \\
+ \left( \text{probability of key not in array} \right) \times n \]

\[ = \left( \sum_{j=0}^{n-1} \frac{q}{n} \times (j+1) \right) + (1-q) \times n \]

\[ = \left( \frac{q}{n} \sum_{j=0}^{n-1} j \right) + \left( \frac{q}{n} \sum_{j=0}^{n-1} 1 \right) + (1-q) \times n \]

\[ = \frac{q}{n} \times \frac{(n-1)(n)}{2} + \frac{q}{n} \times n + (1-q) \times n \]

\[ = q \times \frac{(n+1)}{2} + (1-q) \times n. \]

Hence, if \( q = 1 \) (key is guaranteed in the array), then

\[ T_{\text{LS}}^A(n) = \frac{(n+1)}{2}, \]

and if \( q = \frac{1}{2} \) (key is as likely to be found in the array as not), then

\[ T_{\text{LS}}^A(n) = \frac{(n+1)}{4} + \frac{n}{2} \approx \frac{3n}{4}. \]

In either case, \( T_{\text{LS}}^A(n) \in \Theta(n) \). Note that with a different probability distribution, a different average may be obtained, perhaps even one with a different order. Also, if \( T_{\text{LS}}(n) \) is the time to do a linear search on an array of size \( n \), then \( T_{\text{LS}}(n) \in O(n) \), but \( T_{\text{LS}}(n) \notin \Theta(n) \). This is because not all linear searches require a linear amount of time, but all functions in \( \Theta(n) \) must grow linearly.

Unfortunately, as the preceding example indicates, average-case timing analysis is generally more difficult than best-case or worst-case timing. The difficulties begin with the need to obtain a reasonable probability distribution of the possible situations. For many problems, this is difficult to do. Even if a realistic probability distribution can be obtained, it is often hard to use it to obtain active operation counts in many algorithms. As a result, only the worst-case timing analysis is done for many algorithms. Fortunately, since a worst-case analysis provides an upper bound on the execution time, worst-case analysis is the most appropriate analysis in real-time applications.

Note that in discussing the best-case time requirements of an algorithm, we are not referring to the time required when there are few or no data (\( n = 1 \) or 0). It refers to the least amount of time that can possibly be required by the algorithm as a function of the data set size.

For many algorithms, the order for the time requirements is the same in both the best and worst cases (e.g., the three algorithms of Section 5.1.1). In particular, the \texttt{getMax} algorithm has a worst-case bound of \( 4n \) and a best-case bound of \( 3n + 1 \). For such algorithms, there is no need to distinguish among best-case, worst-case, and average-case time requirements.
public static int binarySearch(short[] array, int key)
{
    int low = 0, high = array.length - 1;
    boolean found = false;
    while (low <= high)
    {
        int middle = (low + high) / 2;
        if (key == array[middle])
            return middle;
        else if (key < array[middle])
            high = middle - 1;
        else
            low = middle + 1;
    }
    return -1;
}

Figure 5.13. Code for binary search

5.2.2 Binary Search

A linear search is about as fast a search as can be done for unordered items. If the items
are ordered, however, a much faster search can be done. Thus, suppose that the data are
ordered into ascending order; that is, in order from first to last, the successive values either
increase or stay the same. An algorithm such as the selection sort algorithm can be used
to order the data. The linear search approach can be used for the search, with the speedup
of stopping the search when a value too large is found. For the average case, however, the
search would still need to go halfway through the array, so the algorithm still requires Θ(n)
time on average. This is too large to search large collections, like a book collection of a
library. Of course, you would never use a linear search to find a word in a dictionary as it
is much too slow and there is a better approach.

Suppose a search is begun by comparing the key with the value in the middle location of
an ordered array. If the middle value is too large, then all values in the last half are too large;
thus, the search can be limited to the first half. Similarly, if the middle value is too small,
then the search can be limited to the second half. This approach can be continued, each time
narrowing the search to an interval half the size of the previous interval. Eventually, either
the item will be found or else the interval will become empty, implying that the item does
not occur in the array. This technique is called a binary search as the remaining interval
is cut in half each time. A function based on this approach is given in Figure 5.13. The
parameters and result type in this function are the same as for the linear search.

For an active operation, the comparison key == array[middle] occurs as often as any
other operation and it is central to the algorithm. The active operation is done once for
every time through the loop, but how often is the loop done? In the best case, the key is
found the first time, so the best-case time is Θ(1). In the worst case, it is necessary to
determine how many times the loop can be done; that is, how many times the interval can
be divided in half until there are no locations left. Suppose n is the length of the array and
k is the number of times that n can be divided in half until the result is less than 1. Then
$k$ is the worst-case number of active operations, and the relationship between $n$ and $k$ is

\[
\frac{n}{2^k} < 1 \text{ and } 1 \leq \frac{n}{2^k-1} \\
n < 2^k \leq 2n \\
\log_2 n < k \leq \log_2(2n) = 1 + \log_2 n,
\]

where $\log_2 n$ is the logarithm to the base 2 of $n$; that is, the power to which 2 must be raised to obtain the value $n$. Therefore, the number of times through the loop is at most $1 + \lfloor \log_2 n \rfloor$, and the worst-case time is $\Theta(\log_2 n)$.

It can be shown by doing an average case analysis that $T_{BS}^A(n) = \Theta(\log_2(n))$, so that the order is the same for both the average and worst cases. Of course, both of these cases are much better than $\Theta(n)$ for the linear search (worst and average cases). However, this analysis does not apply for small $n$. In fact, for $n < 15$, the binary search actually requires more time than the linear search. The reason is that the binary search has a larger overhead to compute $\text{low}$, $\text{high}$, and $\text{middle}$ and may require two comparisons each time through the loop.

Of course, to use a binary search, the items must be sorted. If the items are initially unordered and only a small number of searches are needed, it is not worth the time for a sort just to speed up the searches. As usual, the optimal strategy depends on the situation.

Searching is an extremely important task. As a result, several of the data structures considered in the rest of the book are specifically designed to facilitate fast searches. The standard for them to beat—or at least attain—is the binary search of an ordered array.

### Problems 5.2

1. Write a method that takes as parameters an array of integers sorted in **descending** order and an integer. The method should use a binary search to find the integer in the array and return a Boolean value indicating whether the integer was found in the array.

   ★ 2. Consider an array $X$ that contains 10 elements. Assume that the distribution of requests for each element in this array is as shown in the following table.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$X_i$</th>
<th>Distribution of Requests, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$X_1$</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>$X_2$</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>$X_3$</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>$X_4$</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>$X_5$</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>$X_6$</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>$X_7$</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>$X_8$</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>$X_9$</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>$X_{10}$</td>
<td>10</td>
</tr>
</tbody>
</table>

   (a) Compute the expected number of comparisons for a sequential search of this array.

   (b) Suggest a more efficient arrangement of the data entries. Based on this arrangement, recompute the expected number of comparisons for a sequential search.

3. Do an average case analysis of the binary search algorithm.
5.3 Sorting

The task of searching for an item can be accomplished much more quickly if the items are ordered by their values. Therefore, it is useful to study techniques to order, or sort, items. One sort algorithm, the selection sort, has already been presented and studied. There are many others of varying efficiency, some of which are presented in this section and others in Chapter 18.

5.3.1 Bubble Sort

A sort called the bubble sort is probably the most natural sort. A general algorithm for its approach is as follows:

```plaintext
do
  search for two (consecutive) items that are out of order
  if such a pair is found
    interchange them
  while an interchange was done
```

This general algorithm uses the structure of the do-while loop of Java (i.e., the statements between do and while are repeatedly executed as long as the while condition is true). The best-case performance of this algorithm occurs when the array is already sorted. In this case, one pass through the array will fail to find a pair of items out of order, so the algorithm will terminate. The function for the best-case time is $\Theta(n)$, where $n$ is the size of the array. The worst case occurs when the items are distinct and initially in descending order. In this situation, there are $n \times (n - 1)/2$ pairs that need to be interchanged. Suppose whenever a pair is interchanged, the search for the next pair starts again at the beginning of the array. Then an average of $\Theta(n)$ time is taken to find the next pair, and the total time is $\Theta(n^3)$. This results in a poor algorithm; it is the result of taking a workable idea and naively implementing it. If no analysis of timing were done, a programmer might simply use it, assuming that all sort algorithms are about the same or that computers are fast enough that any workable algorithm will do.

The algorithm known as the bubble sort does not quite use this approach for searching. Rather than having each search start at the beginning of the array after each pair interchange, a complete scan is done through the array interchanging any pairs found out of order. If one or more pairs were interchanged during the scan, another scan is done. The algorithm quits when no pair is found that is out of order during a scan. Thus, the general algorithm is

```plaintext
do
  scan through all unsorted locations of the array
  if the items in the current and next locations are out of order
    interchange the items in those locations
  while one or more interchanges were done in the scan
```

Note that when the first scan reaches the largest item, it moves the largest item past other items until the largest item is moved to the last location. Similarly, the second scan moves the second largest item to the second last location. In general, the $i$th pass moves the $i$th largest item to the $i$th location from the end if it was not there already. Hence, the $i$th pass need only go the $i$th location from the end. Also, at most $n - 1$ passes are needed. Therefore, the worst-case time requirement function is $\Theta(n^2)$, since the scan loop takes $\Theta(n)$ time and at most $n - 1$ of them are done. This is a considerable improvement over the first
version. There are many other improvements that can be done to the algorithm, like keeping track of when a scan can be stopped sooner or having the scans alternate between going left to right and going right to left. However, none of these improvements improves the order of the worst-case time. The result is an algorithm with the same order as the selection sort. A more detailed analysis would show that the selection sort is somewhat faster for random data, but the bubble sort has the advantage of being fast when the data are already sorted or close to being sorted.

### 5.3.2 Merge Sort

Neither of the preceding sort algorithms is efficient for large data sets. We now examine another sort algorithm, called the merge sort, which is significantly faster. Before getting to the actual merge sort algorithm, what is meant by a merge is considered, and then a merge algorithm is given. In computer science, merging is the task of combining two or more sorted sequences into a new sorted sequence. The general approach of a merge is to find the items for the new sequence from first to last. As each item is found, it is added to the right end of the new sequence. It is easy to see how to find the next item to be added by considering an example. Suppose two sorted sequences, \( a \) and \( b \) given next, are to be merged to form a new sorted sequence \( c \). Since sequence \( c \) is to be sorted in ascending order, the items are added to it from smallest to largest:

\[
\begin{align*}
a &: 3, 5, 10, 12, 15, 23, 27, 35 \\
b &: 7, 9, 11, 17, 19, 20 
\end{align*}
\]

Since 3 and 7 are the smallest of each sequence and \( 3 < 7 \), 3 goes first into sequence \( c \). Now the next smallest of sequence \( a \), 5, needs to be compared with 7. Since 5 is smaller, it is placed at the right end of \( c \). Since the last item placed in \( c \) was from \( a \), the next item from \( a \), 10, is compared to the current item from \( b \), 7. Now the item from \( b \) is smaller, so it is moved to \( c \), and the next item from \( b \) is compared to the current item from \( a \). The general algorithm for this task is given in Figure 5.14.

This general approach can be used for a number of purposes (e.g., combining the items of two ordered arrays to obtain another ordered array, or combining two ordered files to yield another ordered file). For the purpose here, the task is to combine two consecutive ordered segments of an array that occur one immediately after the other. As they are merged, the items are placed in a temporary array. After the merge is completed, the items are copied back to the original array. A procedure to do this task is given in Figure 5.15. The procedure assumes that the two ordered segments are in vector \( array \), and \( temp \) is another vector with the same index range and type.

Now consider the time requirements of this algorithm. Suppose that there are \( m \) items to be merged—that is, \( m = \text{end2nd} - \text{start1st} + 1 \). To have an active operation that is executed the most often, it should be in each of the four loops of the algorithm. Given this requirement, the best choice is the operation of copying an item from one array to another. Each loop has an operation that does this task, even though different loops do different copying. Note that each item is copied exactly twice, once from \( array \) to \( temp \) and once from \( temp \) to \( array \). Thus, \( 2m \) active operations are done, and \( \text{Merge}(m) = 2m \in \Theta(m) \), where \( \text{Merge}(m) \) is the function for the number of active operations required by the merge procedure when \( m \) items are to be merged. (Note that, although \( n \) is often used, any variable can be used as a parameter to a timing function provided that an interpretation is given for its value in the application.)
curA refers to the first item of a
curB refers to the first item of b
while there exists a current item in both sequences
  if item curA <= item curB
    move item curA to right end of c
    move curA to the next item of a
  else
    move item curB to right end of c
    move curB to the next item of b
while the remaining sequence isn’t empty
  move the current item of the remaining sequence to the right end of c
  move the current reference of the remaining sequence to the next item of the sequence

Figure 5.14. General algorithm for a simple merge

Now return to the task of sorting a sequence of items. For ease of exposition, assume that the number of values to be sorted is a power of 2, say \( m = 2^k \). Initially, it might seem that the array does not start with any ordered sequences that can be merged, but consider the length 1 sequences. There are \( m = 2^k \) of them, and each is ordered. The merge sort algorithm begins by merging consecutive pairs of these segments. This is called pass 1. In pass 2, the \( 2^{k-1} \) ordered sequences of length 2 (from pass 1) are pairwise merged. In general during pass \( i \), the \( 2^{k-i+1} \) ordered sequences of length \( 2^{i-1} \) are pairwise merged to yield ordered sequences of length \( 2^i \). When pass \( k \) has been completed, the whole array is sorted. Note that with each pass, the length of the sequences being merged doubles. Also the passes continue to occur until there is only one sequence left; that is, the next size for the sequences to be merged is greater than or equal to the size of the array. If \( m \) is not a power of 2, some passes may have fewer than \( 2^{k-i+1} \) sequences to merge. But the merging of sequences within a pass can continue as long as two sequences exist (i.e., as long as the start of the second sequence is within the array). Figure 5.16 shows all the merges in the four passes needed to sort 11 values. The merge sort algorithm that uses these ideas is given in Figure 5.17.

To do the timing analysis of this algorithm, again assume there are \( m \) items to be sorted and \( k = \lceil \log_2(m) \rceil \). The most operations are done when there are actually \( 2^k \) items, so we assume that this is true. By the previous analysis, there are \( k \) passes where the \( i \)th pass merges \( 2^{k-i+1} \) ordered sequences of length \( 2^{i-1} \). Thus, in the \( i \)th pass, there are \( 2^{k-i} \) pairs of sequences, and each merge involves \( 2^i \) items. Since \( T_{\text{merge}}(m) = 2m \), the total number of active operations is given by

\[
\sum_{i=1}^{k} 2^{k-i} \cdot T_{\text{merge}}(2^i) = \sum_{i=1}^{k} 2^{k-i}2^{i+1} = \sum_{i=1}^{k} 2^{k+1} = 2^{k+1} \sum_{i=1}^{k} 1 = 2^{k+1}(k) = 4 \cdot 2^{k-1}(k).
\]
/** Merge start1st through start2nd - 1 with start2nd through end2nd
 in array using temp as a temporary array.
 Analysis: Time = O(end2nd - start1st) */

public static void merge(short[] array, short[] temp, int start1st,
    int start2nd, int end2nd)
{
    /* cur1 and cur2 are the indices of the current items of each sequence.
     cur3 is the index of the next location of temp. */
    /* Repeatedly copy the smaller item into temp. */
    int cur1 = start1st, cur2 = start2nd, cur3 = start1st;
    while ((cur1 < start2nd) && (cur2 <= end2nd))
    {
        if (array[cur1] <= array[cur2])
        {
            temp[cur3] = array[cur1];
            cur1++;
            cur3++;
        }
        else
        {
            temp[cur3] = array[cur2];
            cur2++;
            cur3++;
        }
    }
    /* Copy remainder of the first sequence (if any) into temp. */
    while (cur1 < start2nd)
    {
        temp[cur3] = array[cur1];
        cur1++;
        cur3++;
    }
    /* Copy remainder of the second sequence (if any) into temp. */
    while (cur2 <= end2nd)
    {
        temp[cur3] = array[cur2];
        cur2++;
        cur3++;
    }
    /* Copy items from temp back into array. */
    cur1 = start1st;
    cur3 = start1st;
    while (cur3 <= end2nd)
    {
        array[cur1] = temp[cur3];
        cur1++;
        cur3++;
    }
}

Figure 5.15. Code for simple merge
/** Sort array using the merge sort algorithm.
 * The method uses the simple merge algorithm to sort an array.
 * Analysis: Time = O(n log n), where n is the length of array */

public static void mergeSort(short[] array) {
    /* seqSize is the current size of the sequence being merged.
     * start1st, start2nd, end2nd are the ends of the consecutive sequences being merged. */
    short[] temp = new short[array.length];
    int seqSize = 1;
    while (seqSize < array.length) {
        int start1st = 0;
        int start2nd = seqSize;
        while (start2nd < array.length) {
            int end2nd = Math.min((start2nd + seqSize - 1), (array.length-1));
            merge(array, temp, start1st, start2nd, end2nd);
            start1st = end2nd + 1;
            start2nd = start1st + seqSize;
        }
        seqSize *= 2;
    }
}

But $k = \lceil \log_2(m) \rceil$, so that $k < 1 + \log_2(m)$ and $2^{k-1} < m$. Thus, the total count is bounded by $4m \ast (1 + \log_2(m))$. Note that this is also the time requirements for the average and best cases, since the items are all copied to temp and back independent of whether the values were already sorted.

Of the sort algorithms discussed so far, the merge sort algorithm has the advantage of the slowest growing worst-case time requirements. However, it has two significant disadvantages. First, it still requires $\Theta(m \ast \log_2(m))$ time even in the best case—even the poor bubble sort algorithm requires less time on sorted data. Second, the merge sort algorithm requires a temporary vector of the same size as the original vector. In most cases, this is not a problem. However, if the data set is really large, like the contents of a large file, then this becomes a limitation.
There are a couple of possible improvements that can somewhat speed up the merge sort algorithm. First, note that during a merge, sequences from *array* are merged and placed in *temp* and then simply copied back to *array*. Instead the merges can be organized so that on one pass items from *array* are merged and placed in *temp*, and on the next pass items from *temp* are merged and placed in *array*. This saves the copy operation and almost cuts the time in half. A second speed-up improvement to the algorithm is to not use the merge process on small arrays. The overhead for merging is too high for the merge sort algorithm to be efficient for small \( m \). Thus, a faster sort algorithm is obtained by using the selection sort to sort the segments of size 16 or less. Then the merge sort can be started with sorted segments of size 16 (i.e., \( \text{seqSize} = 16 \)). However, neither of these improvements affects the order for the time of the merge sort algorithm.

There is a temptation to say, “This all looks good on paper, but do actual times match the analysis?” Yes they do. Table 5.4 gives the time for six sort methods on random data. The units for the time measurement are ms for milliseconds, s for seconds, m for minutes, h for hours, and NA for not available. The six methods are the \( O(m^3) \) algorithm based on the bubble sort, the \( O(m^2) \) bubble sort, the selection sort, the merge sort as given before, and two versions of the merge sort that avoid extra copying and use the selection and bubble sorts, respectively, to sort small sequences. As the table shows, the three variations on the merge sort are much faster than the others. Thus, there is no doubt that the merge sort is the best approach of these algorithms—not just by analytical analysis, but also by experimental timing. At the other extreme, the \( O(m^3) \) algorithm based on the bubble sort is much slower than the others, requiring about six hours for \( m = 20,000 \). Of the three variations on the merge sort, the one that uses the bubble sort for the initial sequences of size 16 is slightly faster. In addition to showing that it is the fastest algorithm, it also implies that the bubble sort is slightly better than the selection sort for sequences of size 16.

Analytical analysis implies that the bubble sort should be the fastest on sorted data sets. Therefore, we tried out three algorithms—the bubble sort, standard merge sort, and the merge sort with bubble sort for initial size 16 sequences—on data that are nearly sorted. In particular, 160,000 values were placed in an array as follows:
### Table 5.4. Timing Sort Methods on Random Data

<table>
<thead>
<tr>
<th># items</th>
<th>Bubble $m^3$</th>
<th>Bubble $m^2$</th>
<th>Selection</th>
<th>Merge</th>
<th>Merge Selection</th>
<th>Merge Bubble</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3 ms</td>
<td>0 ms</td>
<td>0 ms</td>
<td>0 ms</td>
<td>0 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>500</td>
<td>327 ms</td>
<td>3 ms</td>
<td>2 ms</td>
<td>1 ms</td>
<td>0 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>1,000</td>
<td>2.5 s</td>
<td>12 ms</td>
<td>11 ms</td>
<td>1 ms</td>
<td>0 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>5,000</td>
<td>5.3 m</td>
<td>314 ms</td>
<td>268 ms</td>
<td>6 ms</td>
<td>3 ms</td>
<td>3 ms</td>
</tr>
<tr>
<td>10,000</td>
<td>42 m</td>
<td>1.27 s</td>
<td>1.1 s</td>
<td>12 ms</td>
<td>8 ms</td>
<td>7 ms</td>
</tr>
<tr>
<td>20,000</td>
<td>5.96 h</td>
<td>5.3 s</td>
<td>4.4 s</td>
<td>30 ms</td>
<td>17 ms</td>
<td>16 ms</td>
</tr>
<tr>
<td>40,000</td>
<td>NA</td>
<td>21.5 s</td>
<td>18 s</td>
<td>60 ms</td>
<td>37 ms</td>
<td>35 ms</td>
</tr>
<tr>
<td>80,000</td>
<td>NA</td>
<td>86.3 s</td>
<td>72.2 s</td>
<td>111 ms</td>
<td>70 ms</td>
<td>76 ms</td>
</tr>
<tr>
<td>160,000</td>
<td>NA</td>
<td>5.8 m</td>
<td>4.84 m</td>
<td>219 ms</td>
<td>175 ms</td>
<td>168 ms</td>
</tr>
<tr>
<td>320,000</td>
<td>NA</td>
<td>23.4 m</td>
<td>21.35 m</td>
<td>509 ms</td>
<td>434 ms</td>
<td>422 ms</td>
</tr>
<tr>
<td>640,000</td>
<td>NA</td>
<td>2.21 h</td>
<td>1.91 h</td>
<td>1.1 s</td>
<td>985 ms</td>
<td>958 ms</td>
</tr>
<tr>
<td>1,280,000</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>2.4 s</td>
<td>2.1 s</td>
<td>2.1 s</td>
</tr>
<tr>
<td>2,560,000</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>4.9 s</td>
<td>4.4 s</td>
<td>4.4 s</td>
</tr>
<tr>
<td>5,120,000</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>10.3 s</td>
<td>9.5 s</td>
<td>9.2 s</td>
</tr>
</tbody>
</table>

### Table 5.5. Timing Sorting Methods on Nearly Sorted Data (in milliseconds)

<table>
<thead>
<tr>
<th>% random</th>
<th>Bubble $m^2$</th>
<th>Merge</th>
<th>Merge Bubble</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00%</td>
<td>5</td>
<td>193</td>
<td>145</td>
</tr>
<tr>
<td>0.02%</td>
<td>75</td>
<td>188</td>
<td>145</td>
</tr>
<tr>
<td>0.04%</td>
<td>125</td>
<td>193</td>
<td>143</td>
</tr>
<tr>
<td>0.06%</td>
<td>164</td>
<td>191</td>
<td>143</td>
</tr>
<tr>
<td>0.08%</td>
<td>236</td>
<td>190</td>
<td>145</td>
</tr>
</tbody>
</table>

For each location in the array from first to last

- with probability $p$
  - enter a random value into the location that is within the range of the other values
- with probability $1.00 - p$
  - enter the next value of an ordered sequence into the location

Table 5.5 gives the results in milliseconds for sorting the array with different percentages of the values that are random (i.e., different values for $100 * p$). With no random values (i.e., 0% random), the bubble sort is the fastest, as predicted by the analytical analysis. However, with as little as 0.06% random values, the merge sort with the bubble sort on size 16 sequences is slightly faster than the bubble sort. Moreover, the merge sort algorithm can be customized to take advantage of ordered subsequences so that it runs even more efficiently on nearly sorted data. As a result, the bubble sort algorithm is almost never the most efficient algorithm.

### Problems 5.3

1. Implement the bubble sort as described previously.
2. The insertion sort works by having the values in locations 0 through \( i - 1 \) of an array already sorted. Then item \( i \) is inserted into its proper position in the sorted sequence by shifting the larger items. This process is repeated for \( i \) going from 1 to the array size. Thus, the insertion sort works just like the usual approach for sorting a hand of playing cards. Write the procedure for an insertion sort that has an array of integers for its parameter. Do a timing analysis of the method to determine the order for its time requirements.

5.4 Introduction to Object Comparison, Interfaces, and the Object Class

The previous sections on searching and sorting only have \texttt{short} data. Versions are easily defined for other integer types, but there are complications with most of the other types. In particular, for \texttt{float} and \texttt{double} values, it is not clear what definition should be used for equality. Two \texttt{float} or \texttt{double} values are seldom exactly equal because of all the digits of precision that are stored. Thus, instead of testing for exact equality, an equality test usually checks whether the two values are within an error tolerance of each other. As a result, applications seldom call for a search for a \texttt{float} or \texttt{double} value. However, there is no problem sorting \texttt{float} or \texttt{double} values.

The situation for reference types is more complex. First, recall that for reference types there is two types of equality: reference equality that compares the references (pointers) to the objects; and object equality that compares fields of the objects. Linear searching with reference equality can be done for general objects by simply changing the parameter types in the linear search method given earlier. However, reference equality is seldom what is wanted. For object equality, linear searching can only be done if the parameter types are changed, \texttt{equals()} is used rather than \texttt{==}, and \texttt{equals()} is suitably defined in the class for the objects. We previously saw \texttt{equals()} being defined in the definition of \texttt{BasicBox} in Figure 2.5 on page 38. Note that the definition of \texttt{equals()} does not necessarily just compare all fields, as often the objective is to search for an item with the same key field—say name or employee number—in order to access other fields of the object. An example of this is given later in this section, but first we consider less than, and greater than comparison.

The comparison operators \( <, <=, >, >= \) are not predefined for any reference type. Hence, a binary search or a sort cannot involve reference type unless comparison has been defined for the type. However, it is relatively easy to define a comparison method for any new class. This is a common enough requirement that Java has a special class called \texttt{Comparable} for this purpose: it is actually an interface rather than a class. The definition of the interface \texttt{Comparable} is given in Figure 5.18. Notice that the keyword \texttt{class} is replaced by \texttt{interface}. \texttt{Comparable} is an \texttt{interface} rather than a class because its method does not have a body. The header, that specifies how to call the method, is given, but there is no implementation. The body needs to be supplied by every implementor of the interface. Although this interface appears to be too trivial to be of any use, it defines a type such that all items of this type can be compared. This becomes the type stored in many of the containers discussed in this book. A further rationale for interfaces is provided in the next chapter, which deals with abstract data types.

This definition is also different from the others in that the symbols \( <T> \) appear after the interface name. \( T \) is called a generic type parameter of the interface. The symbols \( <T> \) after the interface name define \( T \) as a type that can be used in the rest of the definition of the interface; for example, see its use as the type of the parameter for \texttt{compareTo()}. Within the interface, actual type isn’t known, but \( T \) represents some unknown type. This is like an \texttt{int} parameter called \( x \) in a method; the actual value of \( x \) is unknown, but some value will be supplied when the method is invoked. Similarly, the actual type for \( T \) is unknown,
public interface Comparable<T> {
    /** Compares this object with the specified object, other, and returns a negative integer, zero, or a positive integer depending upon whether this object is less than, equal to, or greater than other, respectively. */
    public int compareTo(T other);
}

Figure 5.18. The Comparable interface

but whenever the interface is used, an argument must be supplied to specify the actual type represented by T. As the generic parameter represents a type, the argument must be a type. In the definition of Comparable, all that is known is that T is some reference type that is a descendant of Object. Hence, all the methods of type Object can be invoked on an object of type T, but no other methods. An example of using this interface will be given shortly, and most of the rest of the chapters of this book have many more examples.

As can be seen, interface Comparable has one int function compareTo(). As the comment indicates, the method compares the this object to the parameter other. Note that the reference type String is defined to implement the Comparable interface so that two strings can be compared. In class String, compareTo() is defined to compare two strings based on which comes first in dictionary order.

It is also easy to implement the interface Comparable for a user-defined class. Suppose that an application deals with books in a library, such that each book has a title, an author (for now assume only one author), and an isbn number. Assume that a class called Book already exists with these instance variables and with appropriate constructors and methods. Furthermore, assume that we want to order the books by the instance variable isbn. To define comparison for Book objects, all that is necessary is to define a descendant of Book that defines compareTo(). The class for comparable books is given in Figure 5.19. It consists of a constructor (which simply forwards the parameter values to its parent’s constructor), a definition for the method compareTo(), and a definition for equals().

When a class inherits another class, the descendant is said to extend the parent. As a result, the keyword extends is used in the header for the descendant class. When a class implements an interface, the keyword implements is used. Figure 5.19 shows ComparableBook extending Book and implementing Comparable. In Java, a class can extend at most one class, but a class can implement many interfaces. Also, an interface can extend many interfaces. Both of these—multiple interface implementation in a class and multiple interface extension in an interface—are called multiple interface inheritance.

Now consider in more detail the implements clause in the header of the class. As this class in implementing an interface with a generic parameter, we must supply an argument for the parameter. In this case, we want to compare a LibraryBook with another LibraryBook, so LibraryBook is placed in the < > after Comparable.

The definition of compareTo() is simple because the isbn field stores a string, and we can use the built-in compareTo() function for strings to determine whether one string is less than, equal to, or greater than the other. Note that the parameter for compareTo() has type LibraryBook. This is the case since we supplied that argument LibraryBook for the generic type of Comparable; see the definition of the Comparable interface in Figure 5.18.

Before describing the details of method equals(), some other concepts should be explained. First, consider the class Object. The class Object is implicitly defined to be an ancestor of every reference class. This occurs automatically even though the programmer
/** A Book that can be compared to another Book by comparing their isbn numbers using the compareTo() function. */

```java
public class ComparableBook extends Book implements Comparable<ComparableBook>
{
    /** Constructor. */
    public ComparableBook(String t, String a, String i)
    {
        super(t, a, i);
    }

    /** Compare the current book with book 'other'.
     * If this book has a smaller isbn value, then return a negative int, return 0 for equal isbns, and return a positive int for this having a larger isbn.
     * Analysis: Time = O(1) */
    public int compareTo(ComparableBook other)
    {
        return isbn.compareTo(other.isbn);
    }

    /** Is current book equal to 'other'?.
     * Analysis: Time = O(1) */
    public boolean equals(Object other)
    {
        /* Equality is defined to be consistent with compareTo. */
        if (other instanceof ComparableBook)
        {
            ComparableBook otherBook = (ComparableBook) other;
            return compareTo(otherBook) == 0;
        } else
            return false;
    }
}
```

Figure 5.19. Definition of the class for a comparable book

does not specify this inheritance. Hence, Object is an ancestor for the array of types, for the built-in type String, and also for every user-defined type/class. All of the methods of type Object—two of which are toString() and equals()—are inherited by every class. A class inheriting such methods is free to override the body of the method inherited from Object and give it a new implementation that fits the type being defined. We have seen this being done for both toString() and equals().

An important property of object-oriented languages, including Java, is that an object of a certain type T can be assigned to a variable of type V provided that either V is the same as T or V is an ancestor of T. This means that an object of type ComparableBook can be assigned to a variable of type Book. Also, since type Object is an ancestor of every reference type, this implies that any object, of whatever reference type, can be assigned to a variable of type Object. As another example, a variable of type Comparable can be assigned any object of a type that is a descendant of Comparable. The assignment rule also applies to arrays, so an object of type ComparableBook[] can be assigned to a variable of type Comparable[]. Thus, the sorting and searching methods can be rewritten to accept arrays of type Comparable.

An implication of the rule that an object can be assigned to a variable of an ancestor type is that the object referenced by a variable of type A does not necessarily refer to
an object of type A. By the rule, such a variable might have been assigned an object of a type that is a descendant of A. Hence, a variable of type A references an object whose type is A or a descendant of A. Sometimes we need to test whether a variable refers to an object of a specific type. This is done by the instanceof operator. Consider the method equals() defined in class ComparableBook. The parameter has type Object, but as just mentioned, the actual argument can be a descendant of class Object. If the argument is an instance of type ComparableBook, we know how to do the comparison — use method compareTo(). Therefore, the method first uses the instanceof operator to test whether the argument is a ComparableBook. If not, then the argument and the this object, a ComparableBook, are clearly not equal and false is returned. If the argument has type ComparableBook, then the argument needs to be converted to type ComparableBook. In Java, this action is specified by the expression (ComparableBook)other. The expression is called a typecast. The meaning of this expression is as follows: if the type of the object referenced by other is ComparableBook, then return a reference of type ComparableBook that refers to the ComparableBook object; otherwise, if the object referenced by other does not have type ComparableBook, then throw an exception. Thus, a typecast can succeed in doing the type change, or it can fail and throw an exception. In the present case, the typecast is successful because the if condition must have had the value true. Therefore, a reference to a ComparableBook is obtained and the compareTo() method is used to do the comparison.

Note that two Books could be compared by just providing a definition for compareTo() in the Book class without implementing Comparable. The advantage of implementing Comparable is that an array — or any other container — can be defined to hold objects of type Comparable. For such a container, only objects of type Comparable can be placed in the container. Thus, compareTo() can be used for item comparison within the container, say for searching or sorting. Also, the container is generic in the sense that it can contain any type of object that is a descendant of Comparable. Of course, all items in the container still need to have the same type, whatever descendant of Comparable it is, as compareTo() is usually only defined to compare two items of the same type.

Note that compareTo() can be used to test for equality and so can equals(). Java does not force a user to have consistent definitions for these two functions — a.compareTo(b) == 0 has the same Boolean value as a.equals(b). Nevertheless, a user should always define them to be consistent, unless there is a really good reason to do otherwise. Figure 5.19 shows equals() being defined in terms of compareTo(), to ensure that they are consistent in the ComparableBook class.

Note that the parameter for compareTo() has type LibraryBook, while the parameter for equals() is Object. As we have seen, the type for the parameter of compareTo() is determined by the type specified for the generic type parameter of the interface Comparable that is being implemented. On the other hand, equals() was originally defined in the class Object, that is not generic. For the Object class, the method equals() compares the this object with an instance of type Object as that is only type known to exist. Thus, the two methods compareTo() and equals() have different parameters types because of the other class/interface in which they were originated defined. Note that the parameter for method equals() could have been defined to be LibraryBook. If that were done, then the class LibraryBook would have two equals() methods: one defined in Object and inherited, and one defined within LibraryClass itself. This is not a problem in the language Java, as Java allows more than one method or constructor in the same class/interface to have the same name. This is called overloading. When overloading is used, there must be different parameters for the distinct uses of a name so that Java can determine which version is
/** The index of the first occurrence of Object `key` in `array` (-1 if it doesn't exist). 
Analysis : Time = O(n), n = length of the array */

```java
public static int linSearch(Object[] array, Object key)
{
    for (int i = 0; i < array.length; i++)
        if (array[i].equals(key))
            return i;
    return -1;
}
```

Figure 5.20. Code for linear search with Object parameters

being used. Determining the version specified is easy when there are different number of parameters, or very different types for the parameters. However, it can be confusing when the parameter for one version is an ancestor of the parameter for another. Although Java has specific rules to determine which version is specified (discussed in Section 8.4), the user (especially one not too familiar with Java) can be confused. For a method like `equals()`, it is best to have only one version of the method.

We conclude this section by considering the linear search in Figure 5.20 to see some of the complications that can arise. Note that the parameters have type `Object[]` and `Object`, and `equals()` is used for object comparison. Thus, this linear search can be used for all reference types. Also, the expression `array[i].equals(key)` is used when both `array[i]` and `key` have type `Object`. In Java, the version of `equals()` used is determined by the compile-time type of the target (the type of `array`, `Object` in this case), by the type of the actual object referenced at run time by the target (the type of the object referenced by `array[i]`), and by the compile time type of the parameter (the type of `key`, `Object` in `linSearch()`). The basics rule is to determine which method is called by analyzing the invocation at compile time. At compile time, it is easily seen that method `equals()` is invoked on an object of type `Object`, so it must be an invocation of a method defined for class `Object`. There is only one `equals()` defined for the class `Object`, the one with parameter type `Object`, so that is the one being invoked. Note that if `equals()` was overloaded in class `Object`, then the types of the parameters would be used to determine which one was being invoked. However, in our case, there is no overloading, so the version of `equals()` with parameter `Object` is wanted. At run time, the actual method invoked must belong to the actual class of the target. Thus, the question is which method of the target’s actual type is the one corresponding to the one in class `Object`.

Now, suppose that an array and a parameter, both having type `BasicBox` from Figure 2.5 on page 38, are passed into `linSearch()`. The `equals()` defined in `BasicBox` has an argument type of `BasicBox`. Hence, this is the definition of a new method by the name `equals`, and class `BasicBox` has two methods by the name `equals`. Which one would be invoked by the method in Figure 5.20? As just explained, it is the one defined in class `Object`. Therefore, the version of `equals()` used is the default one defined in `Object` that compares references. Hence, the objects are never compared by their contents, which is probably not what is wanted.

Now consider the class `BetterBasicBox` defined in Figure 5.21. The class `BetterBasicBox` actually has two versions of `equals()` — one with parameter type `BasicBox` defined in class `BasicBox`, and one with parameter type `Object` defined in itself. The latter method overrides (changes) the original definition of the method `equals()` in class `Object`. Now if an array and an argument of type `BetterBasicBox` are passed into `linSearch()`, which version
/** A class that extends BasicBox to include a version of equals that is defined for a parameter of type Object. */

```java
public class BetterBasicBox extends BasicBox {
    /** Constructor for a box. */
    public BetterBasicBox(int l, int w, int h) {
        super(l, w, h);
    }

    /** Does this box have equal variable values as object other? */
    public boolean equals(Object other) {
        BetterBasicBox otherBox = (BetterBasicBox) other;
        return length == otherBox.length & width == otherBox.width & height == otherBox.height;
    }
}
```

Figure 5.21. A descendant of BasicBox with equals() having an Object parameter

of equals() is used? As described above, the method invoked is the method defined in class Object. However, this method has been overridden (revised) in class BetterBasicBox by the version given in that class. Hence, the version of equals() invoked is the one with a parameter of type Object as defined in class BetterBasicBox; i.e., the version that compares the objects by their contents. This is exactly what we want to happen. This implies that, for object comparison to work properly in methods with parameters of type Object, the class for the objects must redefine the equals() inherited from Object so that the result is based on the field values. The body of equals() needs to use typecasting, as shown in class BetterBasicBox, in order to access the fields for comparisons.

The same reasoning applies to the definition of ComparableBox. This time it is simpler as there is only one equals() method defined for class ComparableBox—the override of the one originally defined in class Object. Thus this version is invoked which compares books by their isbn values.

**Problems 5.4**

1. Define class ComparableBox that extends class BasicBox of Chapter 2 and implements interface Comparable. Two boxes should be compared by comparing their volumes.

2. Change the definitions of binary search and merge sort so that they handle items of type Comparable. Use the resulting methods to sort and search an array of type ComparableBook.

### 5.5 Array Dictionaries

The essence of a data structure, like any object, is the values it stores and its operations. Usually a data structure is a dynamic container (i.e., it contains a collection of items that can be inserted or deleted). Therefore, the data structure needs some structure to store the current items, perhaps an array, or a linked structure that will be introduced later in this book. The data structure also needs operations to handle the items. For a dynamic container, operations are needed for insertion, deletion, and item access. Depending on the data structure, the operations are different. However, there is a data structure called a **dictionary** that occurs in many applications.
There are two variations of a dictionary: standard and keyed. A standard dictionary is a collection of items with the main operations on the collection being

- `void insert(G x)`
- `boolean has(G x)`
- `void delete(G x)`
- `G obtain(G x)`

where the items have type `G`. The last operation, `obtain()`, does not seem to be useful, as the object needs to be supplied as an argument to obtain the same item as a result. As we will see later, equality can be redefined to make this operation useful. The standard dictionary has a minimal collection of operations that seem to be necessary to maintain a dynamic collection of items.

More common in many applications is a keyed dictionary. In this data structure, each item has a unique key that is associated with the item. Common keys are a student’s student number, an employee number, a contract number, a book’s isbn number, a car’s serial number, and so on. In general, a key is an identification, often an integer, that uniquely identifies an item. Therefore, each item has a distinct key. In a keyed dictionary, each item is associated with a key at the time the item is inserted. Subsequent access of the item is by key. Thus, the operations are

- `void insert(K i, G x)`
- `boolean has(K i)`
- `void delete(K i)`
- `G obtain(K i)`

where the items have type `G` and the keys have type `K`.

We now consider arrayed implementations of both these variations of a dictionary. Future chapters consider several other implementations.

### 5.5.1 Basic Dictionary

This subsection defines a basic nonkeyed dictionary with an array implementation. The operations for a nonkeyed dictionary are

- `void insert(G x)`
- `boolean has(G x)`
- `void delete(G x)`
- `G obtain(G x)`

where the items have type `G`. To keep the algorithms a little simpler, we implement the dictionary using an unordered array. As this is the first major data structure developed in this book, we develop it in detail, one part at a time so that all the parts are understood. The full implementation is given in Figure 5.22, but do not be too intimidated by the class’ size.
For arrayed implementation, an array is needed to store the items. Since this array can only contain a fixed finite number of items, it is necessary to keep track of the current number of items in the dictionary. Suppose that instance variable `count` is used to store the number of items currently in the container and `capacity()` is a function that returns the size of the array. The use of `count` and `capacity()` immediately leads to the functions `isEmpty()` and `isFull()` of Figure 5.22.

Note that the two functions `isEmpty()` and `isFull()` have the modifier `public`, so that they can be called from an object of any type. However, the instance variable `count` has no modifier, so it can only be accessed by objects whose class is in the same directory as the dictionary class (see Section 2.11.4 on page 39 for a discussion of variable access). Direct access to `count` is precluded for objects in other directories, so that such objects cannot change `count` and give it an incorrect value. However, objects that use the dictionary often need to be able to determine the number of items in the dictionary. Therefore, the public function `count()` is added to the dictionary.

The creation of an array requires that its size be known. This value is supplied by the client by passing the size as an argument to the constructor for the dictionary. Thus, if the array is called `rep` (short for representation), the array declaration, the dictionary constructor, and the capacity function have the form shown in Figure 5.22.

Note that the type for the items stored in the array of the dictionary is `Object`. Recall from the previous section that `Object` is a special class in Java with the property that an instance variable of type `Object` can be assigned an object of any reference type. As a result, by giving an array type `Object`, an object of any reference type can be placed in its locations. Therefore, an object of any reference type can be stored in an instance of `ArrayedBasicDictionary`. Although any reference type can be stored in such a dictionary, normally all items in a dictionary have the same type. However, the flexibility of allowing any reference type means that one dictionary definition can be used to store references to items of any type. If instead we had defined the dictionary class to store items of a specific type `G`, then a new dictionary definition would be needed if type `H` were to be stored in the dictionary. This is not necessary when the dictionary is defined to store items of type `Object`.

The items contained in the dictionary are stored in locations 0 through `count - 1` of the array. This leads to procedure `insert()` in Figure 5.22. Note that the parameter for the procedure has type `Object`.

The dictionary has methods `has()`, `obtain()`, and `delete()` that test for, access, and delete a specific item, respectively. For each of these operations, it is necessary to find a specific item in the dictionary. Thus, it makes sense to design a function specifically for the search task and use it to implement each of the methods `delete()`, `has()`, and `obtain()`. Let `location()` denote this function, where the item being sought is its parameter and it returns the item’s index. Furthermore, assume that if the item does not exist in the dictionary, the function returns the value `count`. Using this specification for function `location()`, Figure 5.22 shows the methods `has()`, `obtain()`, and `delete()` that are obtained. Note that when an item is deleted, all items that follow it are moved one position towards the start so that the deleted item is overwritten. The version of the `obtain()` function given here does not appear to be useful. However, later in this book, we will show that the function can be made more useful if equality is appropriately defined for the type.

The function `location()` is given next in Figure 5.22. It is the standard linear search, except that the function `membershipEquals()` is used for item comparison. This is done as there is more than one kind of equality defined for objects. In particular, Section 4.4 describes two definitions of equality—reference and object. Even for object equality, it can
/** A basic dictionary that stores the items in an unordered array. Its capabilities include isEmpty, isFull, insert, has, delete, and obtain. */

public class ArrayedBasicDictionary
{
    /** The number of items in the data structure. */
    int count;

    /** Is the dictionary empty? Analysis: Time = O(1) */
    public boolean isEmpty()
    {
        return (count == 0);
    }

    /** Is the dictionary full? Analysis: Time = O(1) */
    public boolean isFull()
    {
        return count == capacity();
    }

    /** Current number of items in the dictionary. Analysis: Time = O(1) */
    public int count()
    {
        return count;
    }

    /** The storage for the items in the dictionary. */
    Object[] rep;

    /** Create a dictionary with capacity size. Analysis: Time = O(size) */
    public ArrayedBasicDictionary(int size)
    {
        rep = new Object[size];
        count = 0;
    }

    /** The maximum number of items allowed. Analysis: Time = O(1) */
    public int capacity()
    {
        return rep.length;
    }

    /** Insert x into the dictionary. Analysis: Time = O(1) */
    public void insert(Object x)
    {
        /* Insert at the right-hand end of the array, i.e., location = count. */
        rep[count] = x;
        count++;
    }
}

Figure 5.22. Array implementation of a basic dictionary (part 1)
/** Does the dictionary contain y? 
Analysis: Time = O(count) */
public boolean has(ImmutableList y)
{
    if (location(y) == count)
        return false;
    else
        return true;
}

/** An instance of x from the dictionary. 
Analysis: Time = O(count) */
public Object obtain(ImmutableList x)
{
    return rep[location(x)];
}

/** Delete the item x. 
Analysis: Time = O(count) */
public void delete(ImmutableList x)
{
    /* Starting at the location to the right of the deleted item, 
shift all items to the left one location. */
    for (int i = location(x) + 1; i < count; i++)
        rep[i-1] = rep[i];
    count--;
}

/** Return the location of the first occurrence of x, or count if not found. 
Analysis: Time = O(count) */
public int location(ImmutableList x)
{
    int index = 0;
    while ((index < count) && !membershipEquals(x, rep[index]))
        index++;
    return index;
}

/** Comparison operations use == rather than equals!. 
Default: use *equals* */
boolean objectReferenceComparison = false;
/** Test whether x equals y using the current comparison mode. 
Analysis: Time = O(1) */
public boolean membershipEquals(ImmutableList x, ImmutableList y)
{
    if (objectReferenceComparison)
        return x == y;
    else if ((x instanceof Comparable) && (y instanceof Comparable))
        return 0 == ((Comparable)x).compareTo(y);
    else if (x.equals(y))
        return true;
    else
        return false;
}

Figure 5.22. Array implementation of a basic dictionary (part 2)
/** Set comparison operations to use `equals`. 
Analysis: Time = \(O(1)\) */

```java
public void compareContents()
{
    objectReferenceComparison = false;
}
```

/** Set comparison operations to use `==`. 
Analysis: Time = \(O(1)\) */

```java
public void compareObjectReferences()
{
    objectReferenceComparison = true;
}
```

/** Remove all items from the data structure. 
Analysis: Time = \(O(1)\) */

```java
public void clear()
{
    count = 0;
}
```

/** String representation of the dictionary. 
Analysis: Time = \(O(\text{count})\) */

```java
public String toString()
{
    String result = new String();
    for (int i = 0; i < count; i++)
        result += rep[i].toString() + " ";
    return result;
}
```

Figure 5.22. Array implementation of a basic dictionary (part 3)

be tested via `compareTo()` or `equals()`. As implementors of the dictionary, we do not know what kind of equality a client wants to use, but we should make it possible for a client to specify what kind is wanted.

We could have the client supply an argument or function to specify which equality to use. However, as object equality is almost always what is wanted, it is more convenient to have object equality as the default equality and have commands that can be used to change the default. For object equality, if both `compareTo()` and `equals()` are defined, they should be consistent so that it does not matter which is used. The boolean field `objectReferenceComparison` of class `ArrayedBasicDictionary` is used to store the type of equality to be used. When `objectReferenceComparison` has the value true, this indicates that reference comparison is to be used (i.e., use `==`). When `objectReferenceComparison` is false, object equality is to be used. With this specification, the function `membership-Equals()` can be defined as shown in Figure 5.22. This function assumes that if both `compareTo()` and `equals()` are defined, `compareTo()` should be used. The `membership-Equals()` function is used to compare items in the function `location()`. The procedures to specify the kind of equality, `compareContents()` and `compareObjectReferences()`, are also given in Figure 5.22.

The remaining two methods should be easy to follow. However, it is worth noting that the `toString()` function relies on calling the function `toString()` for each item. The call invokes the function `toString()` that is defined in the class for the actual type of the object
/* Create a dictionary and insert three BetterBasicBoxes. */
ArrayedBasicDictionary boxDictionary = new ArrayedBasicDictionary(10);
BetterBasicBox bbox1 = new BetterBasicBox(1, 11, 21);
boxDictionary.insert(bbox1);
BetterBasicBox bbox2 = new BetterBasicBox(2, 12, 22);
boxDictionary.insert(bbox2);
BetterBasicBox bbox3 = new BetterBasicBox(3, 13, 23);
boxDictionary.insert(bbox3);
/* If the dictionary has a box with dimensions 2, 12, and 22, then delete it. */
BetterBasicBox bbox4 = new BetterBasicBox(2, 12, 22);
if (boxDictionary.has(bbox4))
    boxDictionary.delete(bbox4);
/* Obtain the box from the dictionary with dimensions 1, 11, and 21. */
Object obj = boxDictionary.obtain(new BetterBasicBox(1, 11, 21));
BetterBasicBox bbox5 = (BetterBasicBox) obj;
/* Change the definition of equality in the dictionary to compare boxes by reference. */
boxDictionary.compareObjectReferences();
if (boxDictionary.has(bbox5))
    boxDictionary.delete(bbox5);
/* The next three statements do not delete a box, as bbox6 references a new box, so it will never be found in the dictionary using reference equality. */
BetterBasicBox bbox6 = new BetterBasicBox(3, 13, 23);
if (boxDictionary.has(bbox6))
    boxDictionary.delete(bbox3);

Figure 5.23. Code to demonstrate the use of a dictionary of BetterBasicBoxes

stored in the array location. If no such function is defined in the class, the default definition is one that is supplied in the class Object. As the definition in Object is almost never what is desired, each item class should have its own definition of toString().

Prior to each method is a comment that briefly describes what the method does. It also specifies the time requirements of the method. From the discussion earlier in this chapter, the time bounds should be obvious.

Unfortunately, there are a few complications associated with using this dictionary, so an example is given. Figure 5.23 contains code to create a dictionary, insert some objects of type BetterBasicBox, and access them. Class BetterBasicBox, a descendant of BasicBox of Chapter 2, was given in Figure 5.21. In the example, after creating a dictionary object with capacity 10, three boxes are created and inserted. Next, a box with the same contents as bbox2 is created, and then it is found and deleted from the dictionary. This box is correctly found since by default the dictionary compares objects using object equality. Next in the code, a local variable of type Object is defined, and it is assigned the object in the dictionary with contents (1, 11, 21) (i.e., the object referenced by bbox1). Note that the return type of function obtain() is Object, not BetterBasicBox. Therefore, the result of a call to this function can only be assigned to a variable of type Object. Of course, we know the object actually has type BetterBasicBox, because that was the only type of object placed in the dictionary. This type inconsistency can be resolved by means of the following typecast:

BetterBasicBox bbox5 = (BetterBasicBox) obj;
Typecasting was discussed in the previous section, but it is important enough to discuss it again. It is convenient to think of a reference as having a type. Its type is obtained from either the type of the variable that stores the reference or the type of the function that returned the reference. When a reference has a certain type $A$ and it is known that the object referenced has a descendant type $B$, then a reference to the object of type $B$ can be obtained by applying $(B)$ to the reference. This is called typecasting (i.e., changing a type). The typecast operation will fail and throw an exception if the object does not have type $B$ or a type that is a descendant of $B$. In the statement displayed, a typecast is used to change the type of a reference so that the object referenced by variable $\text{obj}$ can be assigned to the variable $\text{bbox5}$. Since variable $\text{obj}$ has type $\text{Object}$ and variable $\text{bbox5}$ has type $\text{BetterBasicBox}$, the assignment cannot be done without the typecast. Therefore, by using a typecast, an object of a specific type can be obtained from a dictionary even though the dictionary only stores references of type $\text{Object}$. Of course, the item retrieved must actually have that specific type, or an exception will be thrown. Hence, all objects placed in a dictionary should have the same type so that we know what typecast is legal when retrieving items from the dictionary.

The remaining statements of Figure 5.23 change the dictionary to use reference equality and then attempt to delete two boxes. The first deletion will be successful, as $\text{bbox5}$ references $\text{bbox1}$. The second one is not successful, as $\text{bbox6}$ references a new box. It does not delete $\text{bbox3}$, even though they have the same dimensions, since $\text{bbox6}$ references a different box than $\text{bbox3}$ and the dictionary is now using reference equality.

To complete this example, note that the results would be different if the objects had type $\text{BasicBox}$ rather than $\text{BetterBasicBox}$. In function $\text{membershipEquals()}$ of class $\text{ArrayedBasicDictionary}$, the expression $x.equals(y)$ is used when both $x$ and $y$ have type $\text{Object}$. Thus, as described in the previous section, the version of $\text{equals()}$ called is one that has a parameter of type $\text{Object}$. If an instance of $\text{BasicBox}$ were stored in the dictionary, the $\text{equals()}$ invoked in $\text{membershipEquals()}$ would be the one defined in $\text{Object}$—with a parameter of type $\text{Object}$ using reference equality—rather than the one defined in $\text{BasicBox}$—with parameter of type $\text{BasicBox}$ that uses object equality. If object equality is wanted, instances of $\text{BetterBasicBox}$ should be stored in the dictionary as this class defines a version of $\text{equals()}$ that tests for object equality.

As we have seen, since the type for the array in the dictionary class is $\text{Object}$, any reference type can be stored in such an array. However, this specification precludes storing any of the primitive types in this dictionary, as they are not reference types. To place primitive values in a dictionary, the values must be converted to objects. Java provides classes, called $\text{wrapper classes}$, that provide this capability. For example, the class $\text{Integer}$ forms a wrapper for the primitive type $\text{int}$. Besides several members that are not of interest to us now, the $\text{Integer}$ class has the following members:

- a field for a value of type $\text{int}$
- a constructor with a parameter of type $\text{int}$ to initialize the field
- the function $\text{int intValue()}$ defined to return the field value
- the function $\text{boolean equals(Object other)}$ defined to typecast $\text{other}$ to type $\text{Integer}$ and then compare the value fields of the two $\text{Integer}$ objects
- the function $\text{String toString()}$ defined to return a $\text{String}$ representation of the value
/* Create a dictionary and insert four Float values. */
ArrayedBasicDictionary floatDictionary = new ArrayedBasicDictionary(10);
floatDictionary.insert(new Float(5.7));
floatDictionary.insert(new Float(9.3));
floatDictionary.insert(new Float(-4.8));
floatDictionary.insert(new Float(12.7));
/* Delete the objects with values 9.3 and 12.7. */
Float f = new Float(9.3);
if (floatDictionary.has(f))
    floatDictionary.delete(f);
floatDictionary.delete(new Float(12.7));
/* Obtain the object with value 5.7, convert it to type Float, and retrieve its value as an int. */
int i = ((Float) floatDictionary.obtain(new Float(5.7))).intValue();

Figure 5.24. Code to insert float values into a dictionary and to retrieve them

* the function int compareTo(Object other) defined to return a negative integer if the current object is less than other, 0 if the current object is equal to other, and a positive integer if the current object is greater than other.

The other wrapper classes are Boolean, Character, Long, Float, and Double. There are no wrapper classes for byte or short, but Integer can be used instead. The Boolean wrapper does not have the method compareTo(), but otherwise all of the wrapper classes have all of the preceding members, with appropriate type substitutions. Thus, for the class Character, the constructor is Character(char c), and the function char charValue() can be used to retrieve the character. Note that any of the numeric Value functions intValue(), longValue(), floatValue(), and doubleValue() can be applied to any of the numeric objects to yield a value of the appropriate primitive type. Hence, using an appropriate wrapper class, it is easy to create an object containing a primitive value and later retrieve the value from the object. Note that all the wrapper classes except Boolean are descendants of the class Comparable. Hence, any of them can be put in a container of type Comparable.

Figure 5.24 has a simple program segment to place some float values in a dictionary using the Float wrapper. This code first creates a dictionary and inserts four float values. Next it finds and deletes the objects that contain the values 9.3 and 12.7. The last statement obtains the object that contains the value 5.7. The reference returned by the function obtain() has type Object. This reference is typecast to reference type Float. This allows the value in this object to be retrieved as an int value and assigned to i. Note that the typecast to type Float is necessary as type Object does not have the function intValue().

We now summarize some of the typing issues associated with an instance of ArrayedBasicDictionary:

* A dictionary stores items of type Object, where Object is a special reference type that is implicitly an ancestor of every reference type.

* A dictionary can store an item of any reference type. This is allowed as an object of any reference type can be assigned to a variable of type Object and can be passed to a parameter of type Object.

* When retrieving an item from a dictionary, a typecast is needed to convert the reference from type Object to the type stored in the dictionary.
• Items placed in a dictionary must have the method

   public boolean equals(Object other)

appropriately defined in order to use object equality (i.e., to compare items by their contents).

• By default, objects in a dictionary are compared by object equality using the equals() method declared in the object’s class. The default can be changed to reference equality by calling the method compareObjectReferences().

• Primitive types cannot be placed in a dictionary, but each primitive type has a corresponding wrapper class that can be used to store values of the primitive type in a dictionary.

Although the ArrayedBasicDictionary class is a usable class, it does have its limitations. An obvious limitation is its fixed size. Often dictionaries can grow unpredictably large, making it hard to set a capacity when they are created. Later in this book, several other dictionary implementations are presented that avoid this limitation.

Also, the data structure is not very robust (i.e., it does not handle invalid inputs well). For example, consider what would happen if the function obtain() was passed an item that does not exist in the dictionary. First, the location() function would return value count. But array location count is beyond the segment of the array where we are presently storing items. As a result, function obtain() would either return the item at position count, which might be null or might contain some item other than the desired one. The other possibility is the obtain function will throw a run-time exception for attempting to access a location outside the index range of the array. Returning null is a reasonable thing to do, but the other options are unacceptable. Similarly, delete() would run into problems if the item to be deleted is not in the dictionary. It is left as an exercise to modify the code for these methods so that they always behave reasonably. Other approaches to robustness are discussed later in the book.

Another serious limitation of this dictionary implementation is that the methods delete(), obtain(), and has() can each take time proportional to the number of items in the dictionary. This is easy to verify and is stated in the comment at the start of each method. For a large dictionary, this means unacceptably slow performance for operations that are expected to be performed frequently. Therefore, later in this book, several other implementations are given that have better performance characteristics.

The last limitation mentioned here is that the dictionary has only a primitive set of methods. More convenient methods can be added to the dictionary for accessing and handling the items. Such methods are added later to obtain a full-featured dictionary. As this discussion only included a minimal set of methods, the present dictionary was called Basic.

### 5.5.2 Basic Keyed Dictionary

Although a standard dictionary fits some situations, often what is wanted is a keyed dictionary. In a keyed dictionary, each item in the dictionary has a distinct key. Both the item and its key are specified for the insert operation, whereas only the key is given for the other standard operations:

* void insert(K i, G x)
* boolean has(K i)
• void delete(K i)
• G obtain(K i)

where the items have type G and the keys have type K. As with the standard dictionary, the keyed dictionary is designed to be as general as possible. Thus, it would seem that both the items and keys should be designed to have type Object. However, to facilitate fast searches, the array will be kept ordered by the key. This implies that there must be operations to compare two keys and determine which is smaller. Therefore, not just any class can be used for the key type. Since many keys are integers, one obvious approach would be to force the key type to be int. However, this approach would be too restrictive, especially when in many applications the key is a name with type String (i.e., the key has a reference type). For our dictionary, all that we really require is that the key type be restricted to those types for which comparison is defined. As seen in Section 5.4, Java has the special interface Comparable for this purpose. The Comparable interface has the one function

```java
/** Return a negative integer, 0, or a positive integer if the current
 object is less than, equal to, or greater than obj, respectively. */
public int compareTo(Object obj)
```

The wrapper classes Integer, Long, Float, Double, and Character all implement Comparable. The reference type String also implements it. Therefore, any of these types can be used for the key type. In addition, any other class defined to implement Comparable can also be used.

One of the implications of having Comparable keys is that the type of equality is no longer an issue. As long as the keyed dictionary only uses compareTo() to compare keys, it can eliminate the complication of having to deal with membershipEquals() and related members. Now, given that the items have type Object and the keys have type Comparable, the signatures of the main methods for a keyed dictionary become

• void insert(Comparable i, Object x)
• boolean has(Comparable i)
• void delete(Comparable i)
• Object obtain(Comparable i)

This dictionary assumes that the key and the item are distinct objects. In some applications, the key for an item is instead a field of the item. For now, we assume that they are distinct objects and discuss the field option later.

Since both a key and an item need to be stored, the array must store pairs. We will specify that the first item of each pair is the key, and the second item is the data associated with the key. We will use class Pair (see Figure 5.25) to store a pair of items. This class has two instance variables to store the two items of a pair. These instance variables have access modifier public so that they can be accessed and set by any client. Note that the equals() function defined by Pair has an argument of type Object, and that the two pairs are compared by ensuring that they both contain references to the same objects.

The keyed dictionary class is given in Figure 5.26. The PKeyed part of the name of this class is used to indicate that the dictionary is keyed and stores pairs. As can be seen, the array to store the (key, item) pairs is declared as follows:

```java
Pair[ ] rep;
```
/** This class represents a pair of objects. */
public class Pair
{
    /** The first item of the pair. */
    public Object firstItem;
    /** The second item of the pair. */
    public Object secondItem;
    /** Create a pair with initial values v1 and v2.
Analysis: Time = O(1) */
    public Pair(Object v1, Object v2)
    {
        firstItem = v1;
        secondItem = v2;
    }
    /** String representation of the pair.
Analysis: Time = O(1) */
    public String toString()
    {
        return ("(" + firstItem + "," + secondItem + ")");
    }
    /** Does this pair have equal variable values as the pair referenced by other? 
Analysis: Time = O(1) */
    public boolean equals(Object other)
    {
        Pair otherPair = (Pair) other;
        return firstItem == otherPair.firstItem && secondItem == otherPair.secondItem;
    }
}

Figure 5.25. A class to store a pair of items

Several members of the keyed dictionary class are identical to those in the standard dictionary — count, count(), isEmpty(), capacity(), wipeOut(), and toString(). The methods has(), obtain(), and delete() are almost the same in the two dictionary classes, except for the type of the parameter and the access of the item from a pair in the keyed dictionary. The insert() methods are quite different in the two dictionaries since in the keyed dictionary the array must be kept ordered (i.e., the items with a key value larger than the key of the new item must be moved one position to the right). The two location() functions are also different as one uses a linear search with membershipEquals() and the other uses a binary search with compareTo().

The ArrayedPKeyedBasicDictionary class has many of the same limitations as the ArrayedBasicDictionary class. In particular, the keyed dictionary is no more robust than the nonkeyed one. Also, they are both of fixed size. One significant advantage of the keyed dictionary over the nonkeyed one is its search time; its binary search is much faster than the linear search of the nonkeyed dictionary. However, the insert() and delete() times for a keyed dictionary are both slow. In many situations, however, searches are much more frequent than insertions. Consequently, it is more desirable to have a fast search algorithm than a fast insertion algorithm. Nevertheless, one clear objective in developing future data structures is to strive for efficient implementations of the three methods insert(), search(), and delete().
/** A keyed basic dictionary that stores an ordered sequence of (key, item) pairs in an array. There are procedures to insert and delete keyed items, as well as the functions has and obtain. The first item of a pair is the key, and the second item of a pair is the data value associated with the key. */

public class ArrayedPKeyedBasicDictionary
{
    /** The number of items in the dictionary. */
    int count;
    /** The array storing the items. */
    Pair[ ] rep;
    /** Current number of items in the dictionary.
     * Analysis: Time = O(1) */
    public int count()
    {
        return count;
    }
    /** Is the dictionary empty?
     * Analysis: Time = O(1) */
    public boolean isEmpty()
    {
        return count == 0;
    }
    /** Is the dictionary full?
     * Analysis: Time = O(1) */
    public boolean isFull()
    {
        return count == capacity();
    }
    /** The maximum number of items allowed.
     * Analysis: Time = O(1) */
    public int capacity()
    {
        return rep.length;
    }
    /** Create a dictionary with capacity size.
     * Analysis: Time = O(size) */
    public ArrayedPKeyedBasicDictionary(int size)
    {
        rep = new Pair[size];
        count = 0;
    }
    /** Does the dictionary contain an item with the key?
     * Analysis: Time = O(log(count)) */
    public boolean has(Comparable key)
    {
        if (location(key) == count)
            return true;
        else
            return false;
    }
}

Figure 5.26. A keyed dictionary based on storing pairs in an ordered array (part 1)
/** An instance of an item with the key from the dictionary. 
Analysis: Time = \( O(\log(count)) \) */
public Object obtain(Comparable key) {
    return (rep[location(key)]).secondItem;
}

/** Delete the item with the key. 
Analysis: Time = \( O(count) \) */
public void delete(Comparable key) {
    /* Starting at the location to the right of the deleted item, 
    shift all items to the left one location. */
    for (int i = location(key) + 1; i < count; i++)
        rep[i - 1] = rep[i];
    count--;
}

/** Insert \( x \) and its key value into the keyed dictionary. 
Analysis: Time = \( O(count) \) */
public void insert(Comparable key, Object x) {
    /* Shift all higher-keyed items back one position. */
    int index = count - 1;
    while ((index >= 0) && (key.compareTo(rep[index].firstItem) < 0))
    {
        rep[index + 1] = rep[index];
        index--;
    }
    Pair newPair = new Pair(key, x);
    rep[index + 1] = newPair;
    count++;  
}

/** The location of the first occurrence of an item with the key, 
returning \( count \) if not found. 
Analysis: Time = \( O(\log(count)) \) */
public int location(Comparable key) {
    /* Perform a binary search on the sorted array of elements. */
    int low = 0, high = count - 1, middle = 0;
    boolean found = false;
    while (!found && (low <= high))
    {
        middle = (low + high) / 2;
        if (key.compareTo(rep[middle].firstItem) < 0)
            high = middle - 1;
        else if (key.compareTo(rep[middle].firstItem) > 0)
            low = middle + 1;
        else // if necessary, move high to find the first occurrence
            if (middle != low)
                high = middle;
            else
                found = true;
    }
    if (found)
        return middle;
    else
        return count;
}

Figure 5.26. A keyed dictionary based on storing pairs in an ordered array (part 2)
/** Remove all items from the data structure. 
 Analysis: Time = O(1) */

public void clear()
{
    count = 0;
}

/** String representation of the items in the dictionary. 
 Analysis: Time = O(count) */

public String toString()
{
    String result = new String();
    for (int i = 0; i < count; i++)
    {
        result += rep[i] + " ";
    }
    return result;
}

Figure 5.26. A keyed dictionary based on storing pairs in an ordered array (part 3)

Problems 5.5

1. Develop a simple system that will keep track of who is coming to a weekend party. The system should display a menu with the following operations:
   
   • add another name to the list
   • remove a name from the list
   • check if a name is in the list
   • output all names in the list

Implement the container to hold the names by an ArrayedBasicDictionary. Be sure to give the time complexity of each method. Note that for at least one method, its time will be a function of the number of operations entered.

2. Write a program to maintain a simple address book. The address book should keep information for a collection of friends and relatives. For each person, the book should record their postal address, e-mail address, and telephone number. Assume that each person has a unique surname, and the book is organized so that a person is looked up by surname.

3. Modify obtain() and delete() of ArrayedBasicDictionary so that they always do something reasonable (i.e., never fail with an exception thrown).

4. Define a descendant of ArrayedPKeyedBasicDictionary that can facilitate moving through the dictionary. At any specific time, there might or might not be a current item.

   (a) The following methods should be placed in the descendant:
   
   itemExists — a boolean function that indicates whether there is a current item
   item — a function to return the current item, if it exists
   itemKey — a function to return the key for the current item, if the item exists
   goFirst — make the first item in the dictionary the current item
   goForth — make the next item in the dictionary the current item
   after — a boolean function that indicates whether you have walked off the end of the dictionary

   (b) Add the following methods:
   
   goLast — make the last item in the dictionary the current item
   goBack — make the previous item in the dictionary the current item
   before — a boolean function which indicates whether you have walked off the beginning of the dictionary
(c) Include the following methods to modify the dictionary while iterating through it:

- `deleteItem` — delete the current item from the dictionary and make the next item the dictionary current item
- `insertNext` — insert a new item as the next item
- `insertPriorGo` — insert a new item before the current item and make the new item the current item
- `setItem` — change the item associated with the current key value

5.6 Concluding Remarks

Various data structures, including arrays, provide the means to store large numbers of items. As a result, time requirements of some algorithms can become significant or even prohibitive. Thus, techniques were presented to analytically analyze the time requirements of an algorithm. Time requirements are normally presented as a function that grows relative to the size of the data. Order notation was presented to denote such growth. Set notation is used to represent a set of functions that grow at a similar rate, for example, \( T(n) \in O(n^2) \). This is not standard notation, as it would more normally be written as \( T(n) = O(n^2) \). However, with this standard notation, it is important to realize that \( = \) is not being used to represent equality between the left and right sides. Set notation is used because it is more natural and intuitive.

Two tasks common to much of computer science are searching and sorting. Techniques were given for searching arrays. Several methods of sorting values in an array were also developed and compared. They provided evidence of contrasting time requirements for algorithms that do the same task and the need for care in the selection of an algorithm to use to accomplish a task.

In the study of searching for reference types, the concepts of reference equality, object equality, and object comparison were discussed. Also, the special class `Object` was discussed, including its role in defining methods that can operate on data of any reference type.

Dictionaries are among the most common data structures. Nonkeyed and keyed dictionaries were discussed, including their arrayed implementations. In the rest of the book, dictionaries with a larger selection of methods will be discussed, and other implementations of dictionaries will be presented that have smaller time requirements.

5.7 New Java Constructs

- `Comparable` interface: The `Comparable` interface and its method `compareTo` were described on page 123. An example of defining a user-defined `Comparable` class was given in Figure 5.19. Consistency between the functions `equals()` and `compareTo()` was discussed on page 126.

- Class `Object`: The special class `Object` was discussed on page 124. Also see Appendix A.4.3.

- Reference widening conversion: The type of a reference can be changed to an ancestor type. Such conversions were introduced on page 125, and used to store items in an array of type `Object`.

- Typecasting: Typecasting for reference types was discussed in two places. On page 126, typecasting was used in `compareTo()` to convert the type of a parameter to the type of the actual object passed into the method. When an object is retrieved from a
dictionary, typecasting was used, see page 135, to convert the reference from type `Object` to the actual type of the object. Typecasting is discussed in more detail in Appendix A.5.2.

- **Method overloading:** There can be more than one method in a class with the same name, provided that they have a different number or types of parameters. Some of the complications of calling the correct version of a method were introduced on page 127. In particular, a class for items to be stored in a standard dictionary must have `equals` defined with an `Object` parameter. Also, in an implementation of interface `Comparable`, `compareTo()` should be defined with parameter type `Object`.

- **Interfaces:** A Java program mostly consists of a collection of classes. A class can be the implementation of an interface. An initial discussion of interfaces was given on page 123 and an example given in Figure 5.18. The difference between extending a class and implementing an interface was briefly discussed on page 124.

- **Reference and object equality:** There was further discussion of reference equality, object equality, and the difference between them in Section 5.4. In the dictionary classes defined, the default is to use object equality, but there are methods to switch to a specific equality test.

- **Wrapper classes:** The wrapper classes for the primitive types were discussed on page 135. They are discussed in more detail in Appendix A.4.6.
Chapter 6

Abstract Data Types and Their Implementation

Contents

6.1 Introduction ........................................ 146
6.2 Data Types ........................................ 146
6.3 Specifying Abstract Data Types ................. 148
   6.3.1 Introduction .................................. 148
   6.3.2 The Axiomatic Approach ...................... 149
   6.3.3 The Constructive Approach ................. 157
   6.3.4 The Postcondition Approach ............. 161
6.4 Specifying and Implementing ADTs in Java .... 164
   6.4.1 ADT Syntax Specification in Java .......... 164
   6.4.2 ADT Semantics Specification in Java .... 166
   6.4.3 ADT Implementation in Java ............... 169
6.5 Assertion Checking and Exceptions ............. 171
6.6 Visibility of Classes and Class Members ....... 183
6.7 Introduction to Design by Contract .......... 185
6.8 Concluding Remarks ................................ 187
6.9 New Java Constructs ............................. 188

One of the important principles of software design is that of abstraction and information hiding. Procedural abstraction has been supported by virtually all programming languages, including ones such as FORTRAN and ALGOL. During the 1970s, the concept of an abstract data type was introduced. The basic idea behind an abstract data type is the separation of the use of the data type from its implementation (i.e., what an abstract data type does can be specified separately from how it is done through its implementation). A focus of this chapter is to present formal approaches to specifying abstract data types. Also, a notation for the specification of these data types from an object-oriented perspective is given. We also discuss to what extent Java supports the specification of abstract data types. This discussion includes interfaces, assertions, assertion checking, exceptions, and hiding members of a class.
6.1 Introduction

Developing software components that are reusable should be an important goal in object-oriented software development. The establishment of a “component reuse culture” should result in more software development efforts being spent on building generic application frameworks that make use of computer-assisted and reusable component libraries rather than constructing specific applications systems from scratch. Such a reuse culture would replace the current labor-intensive approach with a more capital-intensive one.

The use of abstract data types (ADTs) is part of a good design strategy that can lead to the production of good reusable components. A software developer only has to know what an ADT does through its specification and not how it does it, based on some chosen representation. The advantage of using the ADT approach is that the implementation of an ADT can change without affecting those methods that use the ADT (i.e., its clients). In addition, the complexities of the implementation are hidden. Classes in an object-oriented approach are extensions of the ADT approach in that classes are implementations of ADTs. Therefore, before discussing any other data structures in detail, this chapter’s primary focus is on the specification of ADTs.

The next section introduces types and data types. Section 6.3 deals with three approaches for specifying ADTs. The first approach, the axiomatic approach, is more formal and mathematical. The second approach, the constructive approach, defines a new ADT in terms of previously defined ADTs. The third approach, the postcondition approach, defines the semantics of each operation of a new ADT by specifying the logical conditions that must hold for the operation to be correct. Examples of the approaches are given in this section.

Section 6.4 deals with the Java implementation and Java specification of an ADT. In Java, an interface is the natural way to specify an ADT. To specify the methods of an interface, preconditions, postconditions, and invariants are introduced. If one of these conditions fails, an exception should be thrown. Thus, exceptions are discussed in Section 6.5. To hide the implementation of a class, Section 6.6 discusses the access modifiers of Java. Finally, Section 6.7 introduces a principle that has gained popularity in software design: Design by Contract.

6.2 Data Types

Even before the advent of computers, types were used in areas such as botany to classify plants into different categories. Traditional programming languages, such as Pascal and C, provide data types to classify various kinds of data. The advantages of using types in software development include:

- Assistance in the prevention and detection of errors
- Assistance to software developers in understanding and organizing their ideas about objects
- Assistance in identifying and describing unique properties of certain types

As all of these advantages are important in programming languages, let us examine each advantage more closely. One of the important contributions of typed languages, such as ALGOL and Pascal, is in the area of error detection and prevention. If an object of a certain type is used in an inappropriate context, a type error occurs. For example, an arithmetic expression such as
5 + 2 * "computer science"

contains an inappropriate use of a string literal.

Problem solvers use types to organize and express solutions to problems by identifying specific ideas about data. Types are an important link between the external world and the data elements that programs manipulate. The use of types allows developers to restrict their attention to particular kinds of data.

Specific types have unique properties. For example, size is a property of arrays and strings but not a property of the logical values, true and false.

Data types have received much attention since the 1970s. Although several definitions of data type exist, a widely accepted definition is the following:

**Definition 6.1:** A data type is a set of values and associated operations on those values.

A wide variety of data types are found in programming languages. The simplest data types, often referred to as primitive data types, are those types that are atomic (i.e., they do not build on other types). Therefore, primitive data types are those whose values are indivisible data elements. They include numeric, Boolean, character, and enumerated types. The Java specification of primitive types is consistent with this more general notion.

Numeric types are perhaps the easiest primitive types to understand because humans are familiar with numbers. However, numbers can also be the most difficult to comprehend because of the many ways that are used to represent them on computers. For example, binary and decimal machines have different representations for numbers. Also, as the number of bits for representing numbers is finite, only subsets of the integers and reals can be represented. As a result, overflow and underflow can occur when performing arithmetic operations. Overflow occurs in a different way for integers than for floating-point numbers. Also, floating-point operations are not always exact because the least significant excess digits are discarded. Normally, numeric ranges and accuracy vary from machine to machine. To eliminate this inconsistency, Java has precise specifications for the number of bits, the range, and the numerical accuracy of each operation for each numeric type. Although we omit this detail, as it is not relevant to our discussion, numerical precision is important in many applications.

The Boolean data type is considered by many to be the simplest type because it has two possible values: true and false. The syntactic form of these logical constants may vary from one programming language to another. Some languages offer a richer set of Boolean operations than others. For example, some languages have constructs that permit programmers to specify short-circuit or conditional operations. In Java, the conditional Boolean operators are || and &&. For these operators, the second operand is only evaluated if its value is needed to determine the value of the overall expression.

The character type consists of the set of characters available for a particular language on a particular computer. In some programming languages, a character is an indivisible symbol, whereas a string is a sequence of zero or more characters. Whereas strings can be manipulated in many ways, few manipulations involve single characters. Character types are often machine dependent. Many IBM machines from the 1970s and 1980s used the Extended Binary Coded Decimal Interchange Code (EBCDIC). However, almost all computer manufacturers now use the American Standard Code for Information Interchange (ASCII). The difference in these two codes is apparent when sorting alphanumeric items. Although both codes use the same ordering for letters, the ordering is different for other characters. This difference gives different results when performing a sort operation.

In Java, characters are represented by Unicode, an international character set that is the unification of many character sets including English, French, German, Spanish, Greek,
Latin, and Hebrew, to name a few. In Java, char is represented by a 16-bit value. Thus, the range of a char is between 0 and 65,536. It should be noted, however, that the use of Unicode is sometimes not codewise efficient for languages such as English and French, whose character sets can be represented by 8 bits. The advantage of Unicode is that it brings global portability to computing.

A simple generalization of Boolean, integer, and character types is the enumerated type. An enumerated type, first introduced in Pascal, is some finite set of values. This set is specified by either listing each value and/or giving the lower and upper bounds of some previously defined enumerated type, usually in the form of a subrange. Boolean, integer, and character types are assumed to be predefined enumerated types. However, enumerated types can be more than elements and subranges of the predefined enumerated types. An enumerated type can represent any finite list of unique names or labels that are often used in generalized conditional statements such as case or switch statements, which are available in Pascal and Java, respectively. Note that Java does not have enumerated types.

In addition to primitive types, many programming languages allow the definition and use of nonprimitive types called aggregate types. An aggregate type is a data type whose values are collections of data elements. An aggregate type is composed of previously defined data types. How an element in the collection is accessed depends on the organization and implementation of the collection. Three basic aggregate types are arrays, sequences, and records.

The most commonly used aggregate type in modeling data is the array. An array is usually a fixed-size collection of data elements each of which is accessible at run time through the evaluation of subscript expressions. All elements of an array must be of the same type.

Another popular aggregate type is the sequence or string. Essentially a sequence is an array whose size can vary at run time. Thus, sequences are similar to dynamic or flexible arrays. Because of the variability in storage requirements at run time, the storage manager for a language that allows the creation and manipulation of dynamic arrays and sequences is much more complicated than one that only deals with fixed-sized aggregates.

A record can have both aggregate and primitive data elements. Each aggregate is eventually decomposed into primitive elements called fields. A record can be viewed as a fixed-sized collection. Unlike arrays, where all data elements must be of the same type, the fields of a record can be of different types. The fields of a record are accessed or selected by the use of identifiers. Unlike arrays, the field selectors are not evaluated at run time. The concept of an object in object-oriented development is a generalization of record type. This generalization encapsulates both data elements and their associated operations.

6.3 Specifying Abstract Data Types

6.3.1 Introduction

The use of the object-oriented paradigm to develop software, in addition to its use in implementing software, requires that software developers deal with objects in all phases of the software life cycle. As a result, there is a great need to describe objects in an adequate manner. An adequate description of an object should be

- Brief, precise, and unambiguous;
- Complete as we want it to be; and
- Not overspecified.
The notation introduced in this chapter to specify ADTs is meant to satisfy these conditions.

Specifications can be given in natural language or in a more formal notation. We generally use formal notation, as specifications given in natural language tend to be verbose, ambiguous, incomplete, and randomly mixed with different levels of abstractions (i.e., not given in a structured form). Furthermore, a natural language specification tends to contain redundancy.

An ADT consists of two major parts—its use/specification and its implementation—as shown in Table 6.1. The specification part, syntax and semantics, describes what the ADT does, ideally specified independently from how it does it. The implementation part, representation and algorithms, defines a possible implementation of an ADT. From a client’s perspective, it is neither necessary nor desirable to know the implementation of an ADT—just what the ADT does. The main idea behind the word abstract in abstract data type is the separation of the specification of the data type from its implementation.

The specification of an ADT consists of two parts: syntax and semantics. The syntax of an ADT specifies the forms of its operations, often called the signatures. The signature of an operation consists of its name, the number and types of its operands, and the type of the value returned by the operation. The semantics of an ADT specify for each operation what its output should be for each possible input. Also, any input values for which an operation is not defined should be specified.

The implementation of an ADT consists of a representation and its associated algorithms. A representation specifies how ADT values are to be stored in memory. The algorithms specify how the operations of an ADT are implemented based on the chosen representation. An ADT may have several implementations, each with its own time and space requirements. The appropriate implementation to use in a given context is determined by performance requirements and by the types or volume of input data.

The separation of the use of an ADT from its implementation also assists developers in the production of correct software. By making the implementation of an ADT inaccessible to its users, errors due to knowledge of representation and algorithmic details are avoided. We refer to this type of protection as information hiding or encapsulation.

The next three subsections of this chapter describe three formal approaches to specifying ADTs: axiomatic, constructive, and postcondition.

### 6.3.2 The Axiomatic Approach

A key part of an ADT is the specification of its semantics. With the axiomatic approach, the semantics of an ADT are described by a set of axioms. An axiom is a statement of truth that always holds. The axioms of an ADT define the semantics by expressing relationships among the operations of the ADT. This approach will now be explained through the presentation of examples.

As a first example, consider the specification of an ADT for an array. Two common array operations are placing a new item into an array and accessing an item in an array. Rather than using assignment and [] to do these tasks, methods/operations are used in our ADT. The operation to place an item in an array—let us call it put—has an item, a subscript
value, and an array as parameters. It places the item in the array location associated with
the subscript value, provided that the subscript value is within the bounds of the array. The
access operation, called item, has a subscript value and an array as parameters, and returns
the item in the location of the array specified by subscript value, again provided that the
subscript is within the bounds of the array.

An array also requires an operation that creates an instance of an array. In the present
context, it is called newArray to emphasize that an array is to be created. Rather than
always having the lower bound for the index range being 0, the newArray operation is
supplied with two integer parameters that specify the lower and upper bounds of the index
range. The newArray operation returns an empty array of the proper size, providing that
the specified lower bound is less than or equal to the upper bound.

To access the two bounds of the index range, the operations lower and upper return the
lower and upper bounds. Finally, for our ADT, we assume that the value in each location
of the array is initially undefined. As it is useful to be able to test if a location has an item,
the operation hasItem is added to perform this test.

The ADT is presented first and then followed by a description of its parts. The array
ADT is specified as follows:

NAME
Array<G>

SETS
G: the set of elements
A: the set of arrays containing elements from G
Z: the set of integers
B: the set of logical constants \{false, true\}

SIGNATURES
newArray<G>: Z × Z ↦ A
A.lower: → Z
A.upper: → Z
A.put: G × Z ↦ A
A.hasItem: Z ↦ B
A.item: Z ↦ G

PRECONDITIONS
For all low, high, i, ∈ Z, g ∈ G, and a ∈ A
pre-newArray<G>(low, high) ::= (low ≤ high)
pre-a.lower ::= true
pre-a.upper ::= true
pre-a.put(g, i) ::= (a.lower ≤ i) and (i ≤ a.upper)
pre-a.hasItem(i) ::= (a.lower ≤ i) and (i ≤ a.upper) and (a.hasItem(i))

BUILD OPERATIONS
newArray, put

AXIOMS
For all g ∈ G, i, j ∈ Z, and a ∈ A
1. newArray<G>(low, high).lower = low
2. newArray<G>(low, high).upper = high
3. a.put(g, i).lower = a.lower
4. a.put(g, i).upper = a.upper
5. a.put(g, i).item(i) = g
6. a.put(g, i).item(j) = a.item(j) when j ≠ i
7. newArray<G>(low, high).hasItem(i) = false
8. a.put(g, i).hasItem(i) = true
9. a.put(g, j).hasItem(i) = a.hasItem(i) when j ≠ i

The first three parts describe the syntax of the array ADT. The NAME part is the name
given to the ADT (i.e., \texttt{Array<G>}), where \(G\) denotes a \textit{formal generic parameter}. The name \texttt{Array<G>} actually denotes a template for types. Replacing the formal generic parameter \(G\) by an \textit{actual generic argument}, such as \texttt{int}, yields the type name \texttt{Array<int>—} an array type where the elements have type integer.

The SETS part specifies the various sets used in specifying the array ADT. In the current example, four sets are used. The set of all integers, \(Z\), and the set of Boolean values, \(B\), should be obvious. The set \(G\) is the set of all possible elements that can be stored in arrays. Finally, \(A\) is the set of all possible arrays that meet this ADT specification.

The third part gives the signature of each of the six operations associated with arrays. The set of signatures for an ADT is known as its \textit{interface}.

The signature for creating an array is specified by standard mathematical notation for a function—that is, the function name, a colon, and its mapping specification

\[
Z \times Z \rightarrow A.
\]

The left part of the mapping, \(Z \times Z\), specifies that the function has two arguments, both of which are from the set of integers. The symbol \(\times\) denotes the cross product used in mathematics (see Appendix C for a short refresher on this notation). The result of the \texttt{newArray} operation is an array that belongs to the set \(A\), as specified by the \(A\) to the right of the arrow. It may seem unusual for \texttt{newArray} to be a function, because in Java creation is done by a constructor. However, when specifying the operations in an ADT, all mappings are functions. Therefore, \texttt{newArray} is a function returning a newly created empty array. When an array is created, the actual type for the elements to be stored in the array is specified in \(<G>\) after the name. The signature specifies this by \(<G>\) after the function name. The function \texttt{newArray} is only defined for inputs such that the lower bound is less than or equal to the upper bound. The function is not defined for the cases where the lower bound is greater than the upper bound. The set of admissible input ordered pairs is called the \textit{domain} of the function. If a function is defined for all possible inputs, it is said to be a \textit{total function}. However, if, as in the current case, the function is only defined for some proper subset of all possible inputs, the function is said to be a \textit{partial function}. The possible results for the function is called its \textit{range}—in this case, the set \(A\). The arrow symbol is used to denote that input values are mapped to a result value. The slash symbol across the arrow denotes a partial function.

All the other operations have an “\(A\)” before their names. This conforms to object-oriented notation, where \(A\) represents the set of possible objects on which the operation can be applied. The period is a separator between the target specification and operation name. There is no “\(A\)” prior to \texttt{newArray} as it is a constructor and not applied to an object.

The operations \texttt{lower} and \texttt{upper} do not have any sets to the left of the arrow, meaning that they have no arguments other than the target array specified to the left of the operation name. As there is no slash symbol across the arrow, these functions are total. These functions return an integer value.

The signature for the \texttt{put} operation specifies two parameters: an element and an integer. The output for this operation is an array. The signature does not specify any relationships between the original array and the returned array. It is the semantics of the operations, given afterwards, which specify the effects of the operation. In this case, the result is a \textit{copy} of the original array with one of its elements having been overwritten. Again, this is a partial function. The semantics specify that the operation is only performed if the specified index is within the bounds of the array.

The signature for the \texttt{hasItem} and \texttt{item} operations also describe partial functions. Each
operation has an integer as its parameter, and \texttt{hasItem} returns a Boolean value while \texttt{item} returns an element. Because each is partial, it is not defined for all integer values.

The semantics for the specification of an ADT consists of three parts: a PRECONDITION part, a BUILD OPERATIONS part, and an AXIOMS part. We discuss each in turn.

Each operation for the \texttt{Array} ADT has an associated precondition. The precondition specifies the relationship(s) that must exist among the input(s) for the function to be defined. Thus, the precondition for the \texttt{newArray} operation is

\[
\text{pre-newArray}<G>(\text{low}, \text{high}) ::= (\text{low} \leq \text{high})
\]

This statement says that the operation can be applied only when the condition \texttt{low} \leq \texttt{high} holds. Such a statement of truth is called an assertion. If the condition is false (i.e., \texttt{low} > \texttt{high} is true), the create operation is not executed. The symbol ::= is a meta-symbol that means “is defined as.” A precondition has the general form

\[
\text{pre-entity.operationName(arg1, arg2, \ldots, argn)} ::= \text{assertion}
\]

where its left part consists of an entity, an operation name, and its associated input data names as arguments. Constructors like \texttt{newArray} do not have an entity. The assertion part is a logical expression whose symbols are selected from the target entity, input data names, relational and logical operators, parentheses, and other operations of the ADT.

As \texttt{lower} and \texttt{upper} are total functions, they can always be applied. Thus, the assertions for their preconditions have the value \texttt{true}.

The precondition for the \texttt{put} and \texttt{hasItem} operations involve the use of the \texttt{lower} and \texttt{upper} functions of an array. As we will see from the axioms, \texttt{lower} returns the lower bound for the index range of the array and \texttt{upper} returns the upper bound. Hence, the precondition states that the operation can be applied only if the index value is within the bounds of the array. The precondition for the \texttt{item} operation is more complex. The first part of the assertion is the same as that for the \texttt{put} operation. The last condition of the assertion states that an item has been placed in location \texttt{i}. The implication is that initially the value in each location is undefined, as we wanted for our ADT.

The second part of the semantics specification deals with the abstract part of the ADT definition. In the BUILD OPERATIONS part, a minimum collection of operations is specified that can be used to build an arbitrary instance. In the \texttt{Array} case, the specified build operations are \texttt{newArray} and \texttt{put}. By doing a \texttt{newArray} operation followed by an appropriate sequence of \texttt{put} operations, any array can be built. A sequence of operations involves the application of one function to the result of another function. For example, the array of size 3 with values ‘a’, ‘q’, and ‘h’ in locations 8, 9, and 10, respectively, can be built by

\[
\text{newArray}<\text{char}>(8, 10).\text{put(‘q’, 9)}.\text{put(‘d’, 8)}.\text{put(‘h’, 10)}.\text{put(‘a’, 8)}
\]

Note that during creation the generic type must be replaced by an actual type. Also, this sequence first places ‘d’ in location 8, and later overwrites it with ‘a’. As with normal assignment to a location in an array, it is only the most recent value that is assigned to the location that can be accessed. Of course, there are an infinite number of other sequences of operations that could be used to build the same array. The key property is that any arbitrary array can be built by at least one sequence of the operations \texttt{newArray} and \texttt{put}.

Recall that one of the key concepts behind ADTs is to hide the implementation. In the axiomatic approach to ADTs, there is no specification as to how the build operations work. Their implementation is purposely hidden. The remaining operations are specified next by
stipulating what they should return when applied to an arbitrary instance created by the build operations.

The last part of the semantics is a specification of the axioms; an axiom is a statement of truth. In the present case, an axiom is a statement known to be true about a relationship that exists between operations of the ADT. In particular, each axiom states what is true about the result of an operation when applied to an instance built from the build operations. Thus, an axiom contains a sequence of operations. The two axioms

1. $\text{newArray}<G>(\text{low}, \text{high}).\text{lower} = \text{low}$
2. $\text{newArray}<G>(\text{low}, \text{high}).\text{upper} = \text{high}$

state that the functions $\text{lower}$ and $\text{upper}$ yield the lower and upper bounds of a newly created array. To complete their specification, we need to specify their results after a put operation. The next axiom,

3. $\text{a.put}(g, i).\text{lower} = \text{a.lower}$

directly does this task for the function $\text{lower}$. It specifies that the $\text{lower}$ value of an array after a put operation is the same as its $\text{lower}$ value before the put. This axiom may need to be applied several times, but repeated applications eventually eliminate all the put operations, and the value for $\text{lower}$ is obtained from the last newArray operation for the array. Therefore, $\text{lower}$ is equal to the low argument of the last newArray operation. Similarly, $\text{upper}$ equals the high argument of the last newArray operation.

The axioms for function $\text{item}$ specify that its value depends on which put operations have been done:

5. $\text{a.put}(g, i).\text{item}(i) = g$
6. $\text{a.put}(g, j).\text{item}(i) = \text{a.item}(i)$ when $j \neq i$

If the function call $\text{item}(i)$ follows immediately after the operation $\text{put}(g, i)$, the result is $g$, since $g$ is in location $i$ of the array. If the function call $\text{item}(i)$ follows immediately after the operation $\text{put}(g, j)$ for $j \neq i$, the result depends on previous puts for the array. Thus, the axiom is repeatedly applied to eliminate puts until reaching the most recent put with argument value $i$. The item placed in location $i$ by that put is returned — exactly what would be expected for an array. Note that it is not necessary to specify the result for an $\text{item}$ operation after a newArray operation as the precondition for $\text{item}$ requires that the array has an item in the location.

The remaining axioms are for the $\text{hasItem}$ operation. The first axiom

7. $\text{newArray}<G>(\text{low}, \text{high}).\text{hasItem}(i) = \text{false}$

states that a newly created array does not have a value in location $i$. The pair of axioms

8. $\text{a.put}(g, i).\text{hasItem}(i) = \text{true}$
9. $\text{a.put}(g, j).\text{hasItem}(i) = \text{a.hasItem}(i)$ when $j \neq i$

are similar to those for $\text{item}$. The first axiom should be obvious. However, when a value is placed in location $j$, it isn’t clear whether location $i$ has a value. To determine this, it is necessary to check previous put operations. This is done by applying $\text{hasItem}(i)$ to the array as it existed prior to the put in location $j$, i.e., apply it to $\text{a}$. This results in the puts being checked until one involves location $i$, or a new array is reached and axiom 7 is used.

We can use the array axioms to evaluate compositions of array operations. When used in this way, axioms are viewed as rewriting rules. For example, consider the evaluation of the expression
newArray<char>(8, 10).put('q', 9).put('d', 8).put('h', 10).put('a', 8).item(9)

Starting from the right-hand side, axioms 8 and 9 can be used to specify the result when function \texttt{item} is applied to an array when the previous operation was a \texttt{put}. In the present case, axiom 9 must be used as the location of item access is different from the location of the \texttt{put}. Applying it we obtain

\[
\text{newArray<char>(8, 10).put('q', 9).put('d', 8).put('h', 10).put('a', 8).item(9)} = \text{newArray<char>(8, 10).put('q', 9).put('d', 8).put('h', 10).item(9)} 
\]

Because the indices are still not equal, axiom 9 can be applied to the right-hand side two more times. Axiom 8 can then be applied to yield the answer. The whole sequence of replacements is

\[
\begin{align*}
\text{newArray<char>(8, 10).put('q', 9).put('d', 8).put('h', 10).put('a', 8).item(9)} &= \text{newArray<char>(8, 10).put('q', 9).put('d', 8).put('h', 10).item(9)} \\
&= \text{newArray<char>(8, 10).put('q', 9).put('d', 8).item(9)} \\
&= \text{newArray<char>(8, 10).put('q', 9).item(9)} \\
&= \text{q} \\
\end{align*}
\]

An issue that arises is whether this set of axioms is sufficient to completely specify the array ADT. The specifications for the build operations \texttt{newArray} and \texttt{put} were purposely omitted to hide any implementation details. For each of the other operations, it was defined for the last build operation that was applied to the array. The only exceptions are that \texttt{item} and \texttt{hasItem} were not defined if they occurred immediately after a \texttt{newArray} operation. This is acceptable, because these situations are precluded by their preconditions. Thus, each of the functions was fully defined.

Another important issue that arises in the axiomatic specification of ADTs concerns consistency. A set of axioms is said to be \textit{consistent} if they are not contradictory. This issue is a little more difficult to handle, so we do not pursue it.

When using the axiomatic approach, it is often necessary to add new operations to an initial basic collection of operations. In the array example, the three operations \texttt{newArray}, \texttt{put}, and \texttt{item} seem sufficient to describe the behavior of the type. However, they are not sufficient to describe the preconditions for the operations, so \texttt{lower}, \texttt{upper}, and \texttt{hasValues} were added. Sometimes it is necessary to define further operations to build the correct result. This is not unusual, just as in programming it is frequently necessary, or at least convenient, to define additional functions or procedures in order to implement an algorithm.

We now present the ADT specification for a new data structure: an unbounded generic stack. An unbounded stack is a sequence of elements, possibly empty, where all insertions and deletions for the sequence are performed at the same end—its top. This is like placing new pieces of paper on the top of a stack of papers and always taking the next piece of paper from the top of the stack. The formal specification of an unbounded stack begins with the name and the sets:

\begin{itemize}
  \item \textbf{NAME} \hspace{1cm} \texttt{Stack\langle G \rangle}
  \\
  \item \textbf{SETS} \hspace{1cm} \\
    \hspace{1cm} G: \text{ the set of all elements that may be stacked} \\
    \hspace{1cm} E: \text{ the set of logical constants } \{\text{false, true}\} \\
    \hspace{1cm} S: \text{ the set of all unbounded stacks, containing elements from } G
\end{itemize}

The name of this ADT is similar to \texttt{Array} in that it has a generic parameter. By replacing \texttt{G} by a specific type, a stack that stores items of the specific type is obtained. This facility is usual for arrays, but not as common for other data structures. Starting with Java 5, generic
classes were added to Java and the notation that we are using is similar to the notation for generic classes in Java.

The SETS part consists of three sets: G, B, and S. The SIGNATURES part contains five functions:

**SIGNATURES**

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>newStack&lt;G&gt;</td>
<td>S</td>
</tr>
<tr>
<td>S.isEmpty</td>
<td>B</td>
</tr>
<tr>
<td>S.push: G → S</td>
<td></td>
</tr>
<tr>
<td>S.pop: S</td>
<td>S</td>
</tr>
<tr>
<td>S.top: G</td>
<td></td>
</tr>
</tbody>
</table>

As with the Array ADT, each operation name except for the constructor is preceded by a set to specify the type of the target object to which the operation is to be applied. Informally, the function `newStack` creates a new empty stack, and `isEmpty` tests the target stack to determine whether it is empty. The `push` operation creates a new stack that consists of the target stack with the input element inserted as the top element. The `pop` operation returns a new stack that is a copy of the target stack with its top element removed. The `top` function returns the top element of the target stack. Note that `pop` and `top` functions are partial, as they can be applied only to nonempty stacks. (See the preconditions that follow.)

The preconditions are straightforward:

**PRECONDITIONS**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all s ∈ S and g ∈ G</td>
<td>newStack&lt;G&gt; ::= true</td>
</tr>
<tr>
<td>For all s ∈ S and g ∈ G</td>
<td>pre-s.isEmpty ::= true</td>
</tr>
<tr>
<td>For all s ∈ S and g ∈ G</td>
<td>pre-s.push(g) ::= true</td>
</tr>
<tr>
<td>For all s ∈ S and g ∈ G</td>
<td>pre-s.pop ::= not s.isEmpty</td>
</tr>
<tr>
<td>For all s ∈ S and g ∈ G</td>
<td>pre-s.top ::= not s.isEmpty</td>
</tr>
</tbody>
</table>

The operations `newStack`, `isEmpty`, and `push` can always be done, whereas `pop` and `top` can only be done when the target stack is nonempty.

For the specification of the axioms, we first determine a set of operations that can uniquely build an arbitrary stack. The two operations to do the job are `newStack` and `push`. Using these operations, an arbitrary stack has either the form `newStack` or the form `s.push(g)` for some s ∈ S and some g ∈ G:

**BUILD OPERATIONS**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newStack, push</td>
<td></td>
</tr>
</tbody>
</table>

**AXIOMS**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all g ∈ G, and s ∈ S</td>
<td>newStack&lt;G&gt;.isEmpty = true</td>
</tr>
<tr>
<td>For all g ∈ G, and s ∈ S</td>
<td>s.push(g).isEmpty = false</td>
</tr>
<tr>
<td>For all g ∈ G, and s ∈ S</td>
<td>s.push(g).top = g</td>
</tr>
<tr>
<td>For all g ∈ G, and s ∈ S</td>
<td>s.push(g).pop = s</td>
</tr>
</tbody>
</table>

The axioms specify the behavior of a structure that operates in a last-in first-out (LIFO) manner. The axioms specify the stack’s LIFO behavior by stating the relationships that exist among its operations. The first axiom states that, on creating an initial stack, it must be empty. The second axiom specifies that pushing an element onto any stack (empty or not) always results in a nonempty stack. Therefore, the `isEmpty` function is defined for each of the possible forms that an arbitrary stack can have when it is constructed from the build operations.
The remaining two axioms specify the LIFO behavior. The third axiom states that the result of applying a push operation followed immediately by a top operation always yields the element just pushed. The fourth axiom specifies that a push operation that is followed immediately by a pop operation always yields the original stack. The operations top and pop are not defined to follow after the operation newStack because their preconditions and the first axiom preclude that form.

Readers who are familiar with traditional stack terminology may notice the familiar use of stack operation names such as push, pop, and top. A stack is one of the many kinds of containers discussed throughout this text. Instead of emphasizing the differences between container types, we instead emphasize their commonalities. This is done by using a standardized naming terminology for the operations of ADTs. For example, in Section 9.2, the stack operations push, pop, and top are renamed insert, deleteItem, and item, respectively. The rationale behind the use of a consistent naming convention is part of the quest to produce libraries of reusable software components. Typically, these libraries may contain thousands of exported operations. Without some kind of consistent naming convention, clients of the library are hard pressed to remember a plethora of different operation names based on the differences of library classes. A professional reusable library should name its available operations in a consistent manner.

**Problems 6.3.2**

1. Modify the ADT specification of stacks in this section to include the operations count that returns the number of elements in a stack, changeTop, which replaces the top element of a stack by a given element, and wipeOut, which removes all elements from a stack.

2. Modify the ADT specification for stacks in this section to describe bounded stacks. The capacity of a stack can be modeled as a query function. The insert operation will now be partial because of a potential stack overflow.

3. Write an ADT specification for points in the plane, each of which consists of x and y real Cartesian coordinates. Include operations such as translate and compute the distance of a point from the origin (a point with an x and y value of 0).

4. A university registration system keeps data on students. In particular, a student’s number (a positive integer), name (a string), address (a string), and year of study (a positive integer) are recorded. In addition to operations to access any of a student’s fields, operations for changing a student’s address and year of study are to be included. Use the axiomatic approach to formally specify this data type.

5. ★ A queue is a container that is similar to a stack except for the order in which its elements are processed. Elements in a queue are processed in the same order that they were received—that is, in a first-in, first-out (FIFO) or a first-come, first-served (FCFS) basis. Deletions are performed at one end—the front of the queue. Insertions are performed at the other end—the rear of the queue.

   Give an axiomatic specification for an unbounded queue type that includes the following operations: newQueue, insert, delete, front, and isEmpty. The front operation returns the element at the front of the queue.

6. ★ Give an ADT axiomatic specification for a bounded queue type. The queue operations were given in the preceding problem. A bounded queue has a fixed capacity that is a parameter to the constructor.
6.3.3 The Constructive Approach

We now explore a second approach to specifying ADTs. Recall that in the axiomatic approach, axioms are used to describe the behavior of an ADT. These axioms define relationships that exist between an ADT’s operations. An alternative approach to specifying the semantics of an ADT’s operations, called the *constructive approach*, is to express each operation in terms of previously defined operations of another ADT. It is assumed that this other ADT has already been formally defined and, as such, forms an underlying model for specifying the new ADT. With this approach, the effects of all operations of the new ADT must be expressed in terms of operations of the ADTs in the underlying model. For example, we could specify a stack ADT using the previously defined array ADT as the underlying model. Before doing so, we explain the approach more fully and give some simpler examples.

Consider a banking application involving customer bank accounts. Each account has the following three attributes:

- **owner** — the holder of an account
- **balance** — the amount held in an account
- **lineOfCredit** — the amount of overdraft permitted for an account

The informal English description of the operations for the ADT *Account* is as follows:

- **newAccount** — an operation that takes, as input, an owner name, initial balance, and a personal line-of-credit amount and returns a new account with its attributes having been assigned their initial values
- **deposit** — an operation that takes, as input, an account and a positive deposit amount and updates the account’s balance
- **withdraw** — an operation that takes, as input, an account and a positive withdrawal amount and updates the account’s balance, provided that the amount is less than or equal to the sum of the account’s balance and approved line-of-credit amounts
- **changeLOC** — an operation that takes, as input, a new line-of-credit amount and changes the lineOfCredit attribute to this amount

The name and sets for the type follow naturally from this description. The signatures are also easy to understand. The only complexity in the signatures is that several of the functions are partial. The preconditions specify for each operation the condition that must hold prior to the execution of the operation. These conditions are necessary to ensure that there is never a negative line of credit and that the balance is never more negative than the line of credit. Thus, the following is obtained:
NAME
Account

SETS
Acc: the set of accounts
N: the set of names
M: the set of money values

SIGNATURES
newAccount: N × M × M ↦ Acc
Acc.owner: → N
Acc.balance: → M
Acc.lineOfCredit: → M
Acc.deposit: M ↦ Acc
Acc.withdraw: M ↦ Acc
Acc.changeLOC: M ↦ Acc

PRECONDITIONS
For all n ∈ N, acc ∈ Acc, and b, c, d, e, w ∈ M.
newAccount(n, b, c) ::= (b ≥ 0) and (c ≥ 0)
pre-acc.owner ::= true
pre-acc.balance ::= true
pre-acc.lineOfCredit ::= true
pre-acc.deposit(d) ::= (d ≥ 0)
pre-acc.withdraw(w) ::= (w ≥ 0) and (w ≤ (acc.balance + acc.lineOfCredit))
pre-acc.changeLOC(e) ::= (e ≥ 0)

The functions newAccount, deposit, withdraw, and changeLOC each alter the state of an account. This is indicated in the mappings by the occurrence of Acc on the right-hand side. The functions for these four operations are partial. Since there are no restrictions on the use of the operations owner, balance, and lineOfCredit, the preconditions for these operations are all true.

In the constructive approach, the operations are expressed in terms of operations in an underlying model. The underlying model should be a data type already defined or else one that is already well known. One of the data types that is already defined is the array, so we will use an array as our underlying model. The item in the first location of the array will be a name value from N. The second and third locations will have money values from M representing the balance and the line of credit, respectively. Since the different locations of the array store different types, the array type must be a common ancestor of the types placed in the array. The only common ancestor for a name value and a money value will be the class Object that is an ancestor of all reference types. Thus, an arbitrary account with name n ∈ N, balance b ∈ M, and line of credit c ∈ M is modeled as

a = newArray<Object>(1, 3).put(n, 1).put(b, 2).put(c, 3)

(i.e., create a new array of size 3 and then place n, b, and c in their correct locations). It is assumed that the set M has operations to add two instances, subtract two instances, compare two instances, and compare an instance with 0. With the underlying model specified, the following semantics are obtained:
MODEL

An array with three locations 1, 2, and 3 containing the customer name, the balance, and the line of credit, respectively.

SEMANTICS

For all acc ∈ Acc, n ∈ N, and b, c, d, e, w ∈ M.
Suppose a ∈ A is the array that models acc ∈ Acc.
newAccount(n, b, c) = newArray<Object>(1, 3).put(n, 1).put(b, 2).put(c, 3)
acc.owner = a.item(1)
acc.balance = a.item(2)
acc.lineOfCredit = a.item(3)
acc.deposit(d) = a.put(2, a.item(2)+d)
acc.withdraw(w) = a.put(2, a.item(2)-w)
acc.changeLOC(e) = a.put(3, e)

For each line of the semantics, the left-hand side is an operation in the ADT being defined, whereas the right-hand side is the same operation expressed in the underlying model. Thus, accounts occur on the left, arrays on the right. With this explanation, each of the specifications should be easy to understand.

Note that an array is not likely what would be used to implement an account. This is desirable as the ADT is not meant to suggest an implementation; it is used here to precisely define the abstract data type. If the underlying model is the same as a reasonable implementation, then the implementor may simply use this implementation rather than attempting to design a better one. This is an undesirable aspect of the constructive ADT approach.

Earlier, using the axiomatic approach, we gave a formal specification of an unbounded stack. Let us use the constructive approach to formally specify a bounded stack. A bounded stack is a stack whose size has some finite maximum value:

NAME

BoundedStack<G>

SETS

G: the set of all elements that can be stacked
B: the set of logical constants \{false, true\}
A: the set of all arrays
N: the set of natural numbers (including 0)
FS: the set of bounded (fixed-capacity) stacks that contain elements from G

SIGNATURES

newBoundedStack<G>: N → FS
FS.push: G ↠ FS
FS.pop: FS ↩ FS
FS.isEmpty: FS → B
FS.isFull: FS → B
FS.top: FS ↠ G

The SETS and SIGNATURES parts are similar to their counterparts in the axiomatic specification of the unbounded stack given earlier. However, there are differences between the two. First, the newBoundedStack function creates a stack capable of containing some fixed maximum number of elements. Note that empty bounded stacks are allowed. Second, since an overflow condition can arise in a bounded stack, we have provided a Boolean function isFull() to test whether a bounded stack is full (i.e., contains the maximum number of elements allowed).

For our underlying model, we can again choose an array to store the items. The array can be one whose capacity is the same as the capacity of the bounded stack. However, the number of items in the stack can vary, so we need to know which locations of the array contain the items. We assume that the items are stored in locations 1 through t, where
t is a variable to record the number of items in the stack. But now the underlying model requires two things—an array and an integer. A collection of two things can be represented by an ordered pair, a well-known structure from mathematics. Thus, the underlying model consists of a pair, and an arbitrary bounded stack has the form \(<a, t>\), where \(<\cdot, \cdot>\) represents an ordered pair, \(a \in A\), and \(t \in \mathbb{N}\). The specification of a bounded stack that is based on this representation is as follows:

**PRECONDITIONS**
For all \(fs \in FS\), \(n \in \mathbb{N}\), and \(g \in G\).
- \(\text{pre-newBoundedStack}<G>(n) ::= true\)
- \(\text{pre-fs.push}(g) ::= \text{not } fs\text{.isEmpty}\)
- \(\text{pre-fs.pop} ::= \text{not } fs\text{.isEmpty}\)
- \(\text{pre-fs.isFull} ::= true\)
- \(\text{pre-fs.top} ::= \text{not } fs\text{.isEmpty}\)

**MODEL**
An ordered pair consisting of an array and an integer \(t\), where the items are in locations 1, 2, \ldots, \(t\) with \(t\) having the top item.

**SEMANTICS**
For all \(fs \in FS\), \(n \in \mathbb{N}\), and \(g \in G\).
- Suppose \(<a, t>, a \in A\) and \(t \in \mathbb{N}\), models \(fs \in FS\).
- \(\text{newBoundedStack}<G>(n) = \langle\text{newArray}<G>(1, n), 0\rangle\)
- \(<a, t>\text{.isEmpty} = (t = 0)\)
- \(<a, t>\text{.isFull} = (t = a\text{.upper})\)
- \(<a, t>\text{.top} = a\text{.item}(t)\)
- \(<a, t>\text{.pop} = <a, t-1>\)
- \(<a, t>\text{.push}(g) = <a\text{.put}(g, t + 1), t + 1>\)

Note that rather than placing \(fs\text{.isEmpty}\) on the left-hand side, \(<a, t>\text{.isEmpty}\) is used as it reinforces the model being used. The preconditions are basically the same as for the unbounded stack. The only exception is the constraint that a bounded stack must not be full when a push is done. In the constructive approach, a result must be specified for every operation. For the \(\text{newBoundedStack}\) operation, the result is the empty container in the underlying model. In the present example, an empty stack in the underlying model is a new empty array and the integer zero. The zero indicates that there are no items in the stack. The \(\text{isEmpty}\) and \(\text{isFull}\) functions simply test the integer of the target to determine the appropriate result. The \(\text{top}\) operation returns the element that is at the index position of the array in the target. The \(\text{pop}\) function returns the same array as in the target, but with an index value one smaller. As a result, the element that was on the top is no longer accessible. Finally, the \(\text{push}\) function has a new array in its result; recall that operation \(\text{put}\) on an array yields a new array. This array is the same as the one in the target, except that the value in the \(t + 1\) location is set to \(g\). Also, the index of the result is one larger than the index of the target, so that the new element can be accessed as the top element.

Both of our examples used an array in the underlying model. This is certainly not a requirement; any well-defined data type can be used. For example, we used an ordered pair in the second example. Where appropriate, ordered triples, quadruples, or whatever can be used. Also, once a data type has been defined, it can be used as the underlying model of other data types.

In summary, the constructive approach is similar to the axiomatic approach. In the axiomatic approach, build operations for the type need to be specified, and the other operations are defined in terms of these build operations. In the constructive approach, the
Sec. 6.3. Specifying Abstract Data Types

161

present type is expressed in terms of some underlying model (i.e., some other type). The
target is represented by an instance of the other type, and the result of every operation is
expressed by using operations of the other type to obtain a suitable instance in the other
type.

Problems 6.3.3

1. Define the semantics for the ADT of an account using the constructive approach when the
underlying model is a triple (i.e., an ordered sequence of three items).

2. Figure 5.25 on page 139 has the code for class Pair. Give the ADT for a Pair using the
constructive approach with an array of size 2 for the underlying model.

3. Give the ADT constructive specification for a point like the one described in Problem 3 of
Section 6.3.2.

4. Give the ADT constructive specification for a student like the one described in Problem 4 of
Section 6.3.2.

5. Use the ADT constructive approach to formally specify first-in, first-out (FIFO) bounded
queue containers using the array underlying model of page 150. The capacity of a bounded
queue is a given input. A queue was described in Problem 5 of Section 6.3.2.

6.3.4 The Postcondition Approach

In the postcondition approach, the semantics of each operation for the new ADT are specified
by using preconditions and postconditions. The preconditions are the same as were used in
the axiomatic and constructive approaches to specifying ADTs. Recall that a precondition
is a condition, a statement of truth, that specifies the condition(s) that must hold
before an operation can be executed. In contrast, a postcondition is a condition that specifies what
must be true after an operation has been executed.

The use of preconditions and postconditions can be illustrated by means of a simple ex-
ample; consider the formal specification of an operation, minimum, for two given nonnegative
integer values. Informally, given an input consisting of two integers, the operation minimum
returns the smallest of those two numerical values. When both input values are equal, either
value can be returned.

The syntax of the desired operation is as follows:

\[
\text{minimum: } \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}
\]

where \(\mathbb{Z}\) denotes the set of integers. Assuming that \(x, y, r \in \mathbb{Z}\), the semantics of \(\text{minimum}\)
are given as follows:

\[
\begin{align*}
\text{pre-minimum}(x, y) &::= (x \geq 0) \text{ and } (y \geq 0) \\
\text{post-minimum}(x, y; r) &::= (r \leq x) \text{ and } (r \leq y) \text{ and } ((r = x) \text{ or } (r = y))
\end{align*}
\]

Note that the domain is defined to be \(\mathbb{Z}\), the set of all integers, but the original specification
involved the comparison of two nonnegative integers. This discrepancy is handled by the
precondition. An alternative would have been to define the domain to be a pair of natural
numbers. Note that syntax for expressing a postcondition is similar to that for a precon-
dition, including ::= to stand for “is defined as.” Because a postcondition must refer to
the result, the name of the result is defined in the left-hand side by an additional argument
separated from the others by a semicolon. As a concrete example, consider the case where
\(x = 2\) and \(y = 5\). For the condition
to hold, the result returned by the function must have a value of 2.

Observe we have specified minimum without giving details of how minimum accomplishes its task. In essence, minimum is viewed as a black box that produces the desired result.

We now consider the ADT for Account using the postcondition approach. The name, signatures, and preconditions are exactly the same as on page 158. This time, to state the semantics of the type for each operation, we specify the conditions that must be true after the operation has completed. As in the minimum example, the postconditions need to refer to the result, so it is placed after the arguments with an intervening semicolon:

**POSTCONDITIONS**

For all \( n \in \mathbb{N}, \text{acc}, r \in \text{Acc}, \text{and} b, c, d, e, w \in M \).

\[
\begin{align*}
\text{post-newAccount} & (n, b, c; r) := (r.\text{owner} = n) \quad \text{and} \quad (r.\text{balance} = b) \quad \text{and} \quad (r.\text{lineOfCredit} = c) \\
\text{post-acc.deposit} & (d; r) := (r.\text{balance} = \text{acc.\text{balance}} + d) \quad \text{and} \quad (r.\text{owner} = \text{acc.\text{owner}}) \quad \text{and} \quad (r.\text{lineOfCredit} = \text{acc.\text{lineOfCredit}}) \\
\text{post-acc.withdraw} & (w; r) := (r.\text{balance} = \text{acc.\text{balance}} - w) \quad \text{and} \quad (r.\text{owner} = \text{acc.\text{owner}}) \quad \text{and} \quad (r.\text{lineOfCredit} = \text{acc.\text{lineOfCredit}}) \\
\text{post-acc.changeLOC} & (e; r) := (r.\text{lineOfCredit} = e) \quad \text{and} \quad (r.\text{owner} = \text{acc.\text{owner}}) \\
& \quad \text{and} \quad (r.\text{balance} = \text{acc.\text{balance}})
\end{align*}
\]

Note that the conditions simply specify the properties of the result in terms of properties of the target and the arguments. The properties given include those properties that do not change as well as those that change. The conditions do not indicate how the result is obtained or stored.

Also note that no postconditions are given for the operations owner, balance, and lineOfCredit. They are the operations used to access the properties of an object. Also, they are used to specify the requirements for any object that results from the other operations. In general, a minimum set of query operations should be used, but the set should be sufficient to access all the properties of the object. The need for these operations is like the opposite of the axiomatic approach. With the axiomatic approach, a minimum collection of build operations was specified, and the result for each of the other operations was specified in terms of building the correct result. With the postcondition approach here, a minimum number of query operations is used to verify that each result has the correct properties.

As an additional example, we give the postconditions for the postcondition specification of a bounded stack. The name, sets, signatures, and preconditions were given in the previous section on the constructive approach:

**QUERY OPERATIONS**

isEmpty, top, pop, capacity, count

**POSTCONDITIONS**

For all \( g \in \mathbb{G}, n \in \mathbb{N}, \text{and} \text{fs} \in \text{FS} \).

\[
\begin{align*}
\text{post-newStack} & <G>(n; r) := (r.\text{isEmpty}) \quad \text{and} \quad (r.\text{count} = 0) \quad \text{and} \quad (r.\text{capacity} = n) \\
\text{post-fs.isFull} & (; b) := (b = (\text{fs.\text{count}} = \text{fs.\text{capacity}})) \\
\text{post-fs.push} & (g; r) := (\text{not} \ r.\text{isEmpty}) \quad \text{and} \quad (r.\text{top} = g) \\
& \quad \text{and} \quad (r.\text{capacity} = \text{fs.\text{capacity}}) \quad \text{and} \quad (r.\text{count} = \text{fs.\text{count}} + 1) \\
& \quad \text{and} \quad (r.\text{pop} = \text{fs})
\end{align*}
\]

Note that pop is treated as a query operation, since a stack can be thought of as consisting of a top element and the remainder. Operation top accesses the top element, and pop returns the remainder of the stack. Also to define the postcondition for operation isFull, two additional query functions, count and capacity, were needed. Of course, we must give the signatures for these new operations and their preconditions:
SIGNATURE

FS.count: \rightarrow N
FS.capacity: \rightarrow N

PRECONDITION

For all \( fs \in FS \).

\[
\begin{align*}
\text{pre-}fs\text{.count} & ::= \text{true} \\
\text{pre-}fs\text{.capacity} & ::= \text{true}
\end{align*}
\]

It should be noted that sometimes the difference between the axiomatic and postcondition approaches is merely a difference in perspective. In the section on the axiomatic approach, we developed an axiomatic definition of an unbounded stack. We also developed the postcondition definition of a bounded stack; we now compare these approaches. The first axiom, \( \text{newStack}\text{<G>}.\text{isEmpty} = \text{true} \), has become the first condition of the postcondition for \( \text{newStack} \). The only difference is the notation used. In the axiomatic approach, the operations were directly applied one after the other (e.g., \( \text{newStack}\text{<G>}.\text{isEmpty} = \text{true} \)), whereas in the postcondition approach, the result of the first operation is associated with a result entity and then the second operation is applied to the result entity. Returning to the axioms, the second, third, and fourth axioms, \( s\text{.push}(g).\text{isEmpty} = \text{false} \), \( s\text{.push}(g).\text{top} = g \), and \( s\text{.push}(g).\text{pop} = s \), have become the first, second, and fifth conditions of the postcondition of \( \text{push} \). Thus, every axiom showed up in the postconditions. The only extra conditions in the postcondition approach involved \( \text{count} \), \( \text{capacity} \), and \( \text{isFull} \), which were added to handle the bounded aspects of the bounded stack done here. Hence, both approaches yielded the same result. This raises the question of which approach is better. The axiomatic approach is often easier to develop, as it is more constructive. However, in Section 6.4.2 we will see that the postconditions are closer to what we want when it comes to developing an actual class from the ADT. Thus, both approaches have their benefits.

Finally, we give the array ADT using the postcondition approach (the name, sets, signatures, and preconditions for \( \text{Array} \) were given on page 150):

QUERY OPERATIONS

\( \text{hasItem, item, lower, and upper} \)

POSTCONDITIONS

For all \( g \in G, \text{low}, \text{high}, i \in Z, \text{and} a \in A \).

\[
\begin{align*}
\text{post-}\text{newArray}\text{<G>}\text{(low, high; } r) & ::= (r\text{.lower} = \text{low} \land (r\text{.upper} = \text{high}) \\
& \land (r\text{.upper} \not\preceq r\text{.hasItem}(k)) \\
\text{post-}a\text{.put}(g, i; r) & ::= (r\text{.lower} = a\text{.lower} \land (r\text{.upper} = a\text{.upper}) \\
& \land (r\text{.hasItem}(i)) \land (r\text{.item}(i) = g) \\
& \land (\forall_{r\text{.upper}, k \neq i} r\text{.hasItem}(k) = a\text{.hasItem}(k)) \\
& \land (\forall_{r\text{.upper}, k \neq i} (r\text{.hasItem}(k) \implies r\text{.item}(k) = a\text{.item}(k)))
\end{align*}
\]

The query operations are fairly obvious, as is the postcondition for \( \text{newArray} \). The postcondition for \( \text{put} \) is more complex. The first two conditions specify that the \( \text{lower} \) and \( \text{upper} \) values are not changed by a \( \text{put} \) operation. The third and fourth conditions state that location \( i \) has a value, and the item in that location is the value \( g \). Finally, the last two conditions require that the items in all the other locations are unchanged. The notation \( \forall_{r\text{.upper}, k \neq i} k \) denotes the universal quantifier “for all” that is constrained (bounded) to an index range of between \( r\text{.lower} \) and \( r\text{.upper} \), inclusive, except that \( k = i \) is excluded.

Problems 6.3.4
1. Use the ADT postcondition approach to formally specify the operation, maximum, which returns the largest of two given integer values.

2. Repeat Problem 1 for two given nonnegative integer values.

3. Repeat Problem 4 of Section 6.3.2 using the postcondition approach.

4. Give the ADT postcondition specification for each of the following:
   (a) unbounded stack
   (b) unbounded queue
   (c) bounded queue

5. Give the ADT postcondition specification for a point like the one described in Problem 3 of Section 6.3.2.

6.4 Specifying and Implementing ADTs in Java

This chapter has discussed the need to specify a data type without giving an actual implementation (i.e., to specify the Abstract Data Type). In the design of a software system, such a specification can be used in other parts of the system design before implementing the data type. Even if the data type has been implemented, as long as other parts of the system only use the ADT specification, the implementation can change without affecting the rest of the system. This promotes modifiability and ease of maintenance of the system. Thus, it is desirable to actually hide the implementation of a data type from its clients.

The specification of an ADT has two main parts: the syntax and the semantics. The syntax gives the name of the type, the sets involved, and the signatures of the methods. Hence, it gives the details on how to create the type and how to call methods on the type. It does not give a specification of what the methods do. The semantics specify what each method accomplishes.

The first two subsections consider the facilities in Java to support ADT specifications. The first subsection presents the specification of the syntax of an ADT in Java, whereas the second considers the specification of the semantics of an ADT. Of course, in a working system, it is not sufficient to have an abstract specification—a implementation is needed. Therefore, the Java implementation of an ADT is also discussed.

6.4.1 ADT Syntax Specification in Java

To give the syntax of an ADT, we need to give the name of the type, the sets involved, and the signatures of the methods for the type. The specification of the name is obvious. In Java, the sets involved in an ADT become the other types involved. These other types occur as parameter types or return types from function calls. In Java, the signature of a method is expressed in its header. Of course, the headers also specify the parameter types and return types, so the sets need not be specified separately. Therefore, the syntax for an ADT can be expressed in Java by giving the name for the ADT and the header for each method of the type. In Java, this is done by defining an interface.

We now develop a Java specification for the Account ADT given on page 158. For the name of the type, we will use AccountSimpleADT to emphasize that it is an ADT and that it is a simple version that will be refined shortly. The SETS specified for the Account ADT consist of the following sets: Acc, the set of accounts; N, the set of owner names; and M, the set of monies. The set Acc corresponds to the type being defined, AccountSimpleADT. For the set of names, N, we choose the String type. For the set of monies, M, we use the