What happens if there is a value in the array that is the same as the pivot value? The index `down` will stop at such a value. If `up` has stopped prior to reaching that value, `table[up]` and `table[down]` will be exchanged, and the value equal to the pivot will be in the left partition. If `up` has passed this value and therefore passed `down`, `table[first]` will be exchanged with `table[down]` (same value as `table[first]`), and the value equal to the pivot will still be in the left partition.

What happens if the pivot value is the smallest value in the array? Since the pivot value is at `table[first]`, the loop will terminate with `down` equal to `first`. In this case, the left partition is empty. Figure 8.16 shows an array for which this is the case.

By similar reasoning, we can show that `up` will stop at `last` if there is no element in the array larger than the pivot. In this case, `down` will also stay at `last`, and the pivot value (`table[first]`) will be swapped with the last value in the array, so the right partition will be empty. Figure 8.17 shows an array for which this is the case.

FIGURE 8.16
Values of **up**, **down**, and `pivIndex` If the Pivot Is the Smallest Value

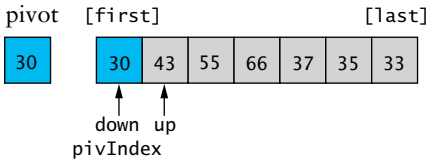
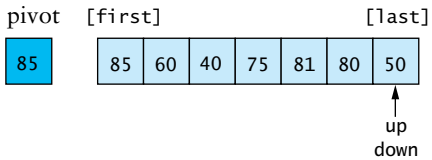
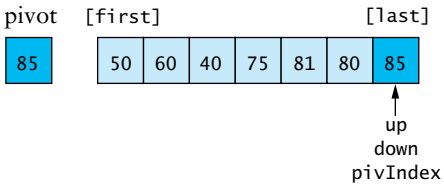


FIGURE 8.17
Values of **up**, **down**, and `pivIndex` If the Pivot Is the Largest Value



After swap



LISTING 8.9
Quicksort `partition` Method (First Version)

```
/** Partition the table so that values from first to pivIndex
    are less than or equal to the pivot value, and values from
    pivIndex to last are greater than the pivot value.
    @param table The table to be partitioned
    @param first The index of the low bound
    @param last The index of the high bound
    @return The location of the pivot value
 */
private static <T extends Comparable<T>> int partition(T[] table,
                                        int first, int last) {
    // Select the first item as the pivot value.
    T pivot = table[first];
    int up = first;
```

```
    int down = last;
    do {
        /* Invariant:
           All items in table[first . . up - 1] <= pivot
           All items in table[down + 1 . . last] > pivot
        */
        while ((up < last) && (pivot.compareTo(table[up]) >= 0)) {
            up++;
        }
        // assert: up equals last or table[up] > pivot.
        while (pivot.compareTo(table[down]) < 0) {
            down--;
        }
        // assert: down equals first or table[down] <= pivot.
        if (up < down) { // if up is to the left of down.
            // Exchange table[up] and table[down].
            swap(table, up, down);
        }
    } while (up < down); // Repeat while up is left of down.

    // Exchange table[first] and table[down] thus putting the
    // pivot value where it belongs.
    swap(table, first, down);
    // Return the index of the pivot value.
    return down;
}
```

A Revised `partition` Algorithm

We stated earlier that quicksort is $O(n^2)$ when each split yields one empty subarray. Unfortunately, that would be the case if the array was sorted. So the worst possible performance occurs for a sorted array, which is not very desirable.

A better solution is to pick the pivot value in a way that is less likely to lead to a bad split. One approach is to examine the first, middle, and last elements in the array and select the median of these three values as the pivot. We can do this by sorting the three-element subarray (shaded in color in Figure 8.18). After sorting, the smallest of the three values is in position `first`, the median is in position `middle`, and the largest is in position `last`.

At this point, we can exchange the first element with the middle element (the median) and use the partition algorithm shown earlier, which uses the first element (now the median) as the pivot value. When we exit the partitioning `loop`, `table[first]` and `table[down]` are exchanged, moving the pivot value where it belongs (back to the middle position). This revised partition algorithm follows.

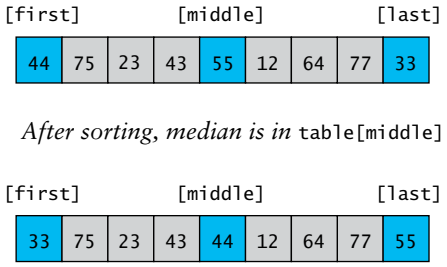


FIGURE 8.18 Sorting First, Middle, and Last Elements in Array

Algorithm for Revised partition Method

1. Sort table[first], table[middle], and table[last].
2. Move the median value to table[first] (the pivot value) by exchanging table[first] and table[middle].
3. Initialize up to first and down to last.
4. **do**
5. Increment up until up selects the first element greater than the pivot value or up has reached last.
6. Decrement down until down selects the first element less than or equal to the pivot value or down has reached first.
7. **if** up < down then
8. Exchange table[up] and table[down].
9. **while** up is to the left of down.
10. Exchange table[first] and table[down].
11. Return the value of down to pivIndex.

You may be wondering whether you can avoid the double shift (Steps 2 and 10) and just leave the pivot value at table[middle], where it belongs. The answer is "yes," but you would also need to modify the partition algorithm further if you did this. Programming Project 6 addresses this issue and the construction of an industrial-strength quicksort method.

Implementation of Revised partition Method

Listing 8.10 shows the revised version of method partition with method sort3, which uses three pairwise comparisons to sort the three selected items in table so that

```
table[first] <= table[middle] <= table[last]
```

Method partition begins with a call to method sort3 and then calls swap to make the median the pivot. The rest of the method is unchanged.

LISTING 8.10
Revised partition Method and sort3

```
private static <T extends Comparable<T>> int partition(T[] table,
                                                       int first, int last) {
    /* Put the median of table[first], table[middle], table[last]
       into table[first], and use this value as the pivot.
     */
    sort3(table, first, last);
    // Swap first element with median.
    swap(table, first, (first + last) / 2);

    // Continue as in Listing 8.9
    // . . .

}

/** Sort table[first], table[middle], and table[last].
    @param table The table to be sorted
    @param first Index of the first element
    @param last Index of the last element
 */
private static <T extends Comparable<T>> sort3(T[] table,
                                               int first, int last) {
```

```
        int middle = (first + last) / 2;
        /* Sort table[first], table[middle], table[last]. */
        if (table[middle].compareTo(table[first]) < 0) {
            swap(table, first, middle);
        }
        // assert: table[first] <= table[middle]
        if (table[last].compareTo(table[middle]) < 0) {
            swap(table, middle, last);
        }
        // assert: table[last] is the largest value of the three.
        if (table[middle].compareTo(table[first]) < 0) {
            swap(table, first, middle);
        }
        // assert: table[first] <= table[middle] <= table[last].
    }
```

EXERCISES FOR SECTION 8.9

SELF-CHECK

1. Trace the execution of quicksort on the following array, assuming that the first item in each subarray is the pivot value. Show the values of `first` and `last` for each recursive call and the array elements after returning from each call. Also, show the value of `pivot` during each call and the value returned through `pivIndex`. How many times is `sort` called, and how many times is `partition` called?

   55 50 10 40 80 90 60 100 70 80 20 50 22

2. Redo Question 1 above using the revised `partition` algorithm, which does a preliminary sort of three elements and selects their median as the pivot value.

3. Explain why the condition (`down > first`) is not necessary in the loop that decrements `down`.

PROGRAMMING

1. Insert statements to trace the quicksort algorithm. After each call to `partition`, display the values of `first`, `pivIndex`, and `last` and the array.

PITFALL

Falling Off Either End of the Array

A common problem when incrementing up or down during the partition process is falling off either end of the array. This will be indicated by an `ArrayIndexOutOfBoundsException`. We used the condition

```
((up < last) && (pivot.compareTo(table[up]) >= 0))
```

to keep up from falling off the right end of the array. Self-Check Exercise 3 asks why we don't need to write similar code to avoid falling off the left end of the array.

8.10 Testing the Sort Algorithms

To test the sorting algorithms, we need to exercise them with a variety of test cases. We want to make sure that they work and also want to get some idea of their relative performance when sorting the same array. We should test the methods with small arrays, large arrays, arrays whose elements are in random order, arrays that are already sorted, and arrays with duplicate copies of the same value. For performance comparisons to be meaningful, the methods must sort the same arrays.

Listing 8.11 shows a driver program that tests methods `Arrays.Sort` (from the API `java.util`) and `QuickSort.sort` on the same array of random integer values. Method `System.currentTimeMillis` returns the current time in milliseconds. This method is called just before a sort begins and just after the return from a sort. The elapsed time between calls is displayed in the console window. Although the numbers shown will not be precise, they give a good indication of the relative performance of two sorting algorithms if this is the only application currently executing.

LISTING 8.11
Driver to Test Sort Algorithms

```
/** Driver program to test sorting methods.
    @param args Not used
 */
public static void main(String[] args) {
    int size = Integer.parseInt(JOptionPane.showInputDialog("Enter Array size:"));
    Integer[] items = new Integer[size]; // Array to sort.
    Integer[] copy = new Integer[size];  // Copy of array.
    Random rInt = new Random();          // For random number generation

    // Fill the array and copy with random Integers.
    for (int i = 0; i < items.length; i++) {
        items[i] = rInt.nextInt();
        copy[i] = items[i];
    }

    // Sort with utility method.
    long startTime = System.currentTimeMillis();
    Arrays.sort(items);
    System.out.println("Utility sort time is "
                       + (System.currentTimeMillis()
                       - startTime) + "ms");
    System.out.println("Utility sort successful (true/false): "
                       + verify(items));

    Reload array items from array copy.
    for (int i = 0; i < items.length; i++) {
        items[i] = copy[i];
    }

    Sort with quicksort.
    startTime = System.currentTimeMillis();
    QuickSort.sort(items);
    System.out.println("QuickSort time is "
                       + (System.currentTimeMillis()
                       - startTime) + "ms");
    System.out.println("QuickSort successful (true/false): "
                       + verify(items));
    dumpTable(items);    // Display part of the array.
}
```

```
/** Verifies that the elements in array test are
    in increasing order.
    @param test The array to verify
    @return true if the elements are in increasing order;
            false if any 2 elements are not in increasing order
 */
private static boolean verify(Comparable[] test) {
    boolean ok = true;
    int i = 0;
    while (ok && i < test.length - 1) {
        ok = test[i].compareTo(test[i + 1]) <= 0;
        i++;
    }
    return ok;
}
```

Method `verify` verifies that the array elements are sorted by checking that each element in the array is not greater than its successor. Method `dumpTable` (not shown) should display the first 10 elements and last 10 elements of an array (or the entire array if the array has 20 or fewer elements).

EXERCISES FOR SECTION 8.10

SELF-CHECK

1. Explain why method `verify` will always determine whether an array is sorted. Does `verify` work if an array contains duplicate values?
2. Explain the effect of removing the second **`for`** statement in the `main` method.

PROGRAMMING

1. Write method `dumpTable`.
2. Modify the driver method to fill array `items` with a collection of integers read from a file when `args[0]` is not **`null`**.
3. Extend the driver to test all O(n log n) sorts and collect statistics on the different sorting algorithms. Test the sorts using an array of random numbers and also a data file processed by the solution to Programming Exercise 2.

8.11 The Dutch National Flag Problem (Optional Topic)

A variety of partitioning algorithms for quicksort have been published. Most are variations on the one presented in this text. There is another popular variation that uses a single left-to-right scan of the array (instead of scanning left and scanning right as we did). The following case study illustrates a partitioning algorithm that combines both scanning techniques to partition an array into three segments. The famous computer scientist Edsger W. Dijkstra described this problem in his book *A Discipline of Programming* (Prentice-Hall, 1976).

CASE STUDY The Problem of the Dutch National Flag

Problem The Dutch national flag (see Figure 8.19) consists of three stripes that are colored (from top to bottom) red, white, and blue. Unfortunately, when the flag arrived, it looked like Figure 8.20; threads of each of the colors were all scrambled together! Fortunately, we have a machine that can unscramble it, but it needs software.

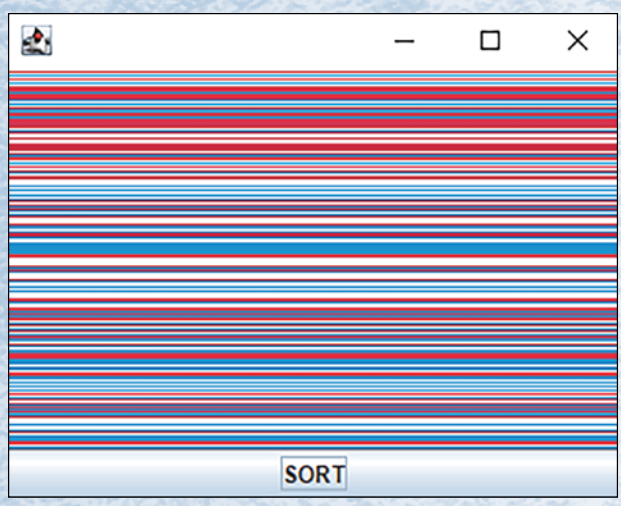
Analysis Our unscrambling machine has the following abilities:

- It can look at one thread in the flag and determine its color.
- It can swap the position of two threads in the flag.

Our machine can also execute `while` loops and `if` statements.

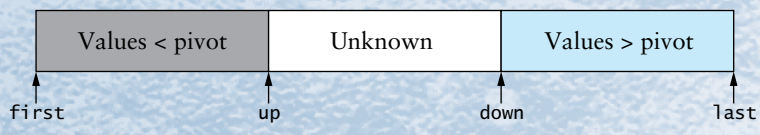
FIGURE 8.19
The Dutch National Flag

FIGURE 8.20
Scrambled Dutch National Flag



Design Loop Invariant

When we partitioned the array in quicksort, we split the array into three regions. Values between `first` and `up` were less than or equal to the pivot; values between `down` and `last` were greater than the pivot; and values between `up` and `down` were unknown. We started with the unknown region containing the whole array (`first == up`, and `down == last`). The partitioning algorithm preserves this invariant while shrinking the unknown region. The loop terminates when the unknown region becomes empty (`up > down`).



Since our goal is to have three regions when we are done, let us define four regions: the red region, the white region, the blue region, and the unknown region. Now, initially the whole flag is unknown. When we get done, however, we would like the red region on top, the white region in the middle, and the blue region on the bottom. The unknown region must be empty.

Let us assume that the threads are stored in an array `threads` and that the total number of threads is `HEIGHT`. Let us define `red` to be the upper bound of the red region, `white` to be

the lower bound of the white region, and `blue` to be the lower bound of the blue region. Then, if our flag is complete, we can say the following:

- If $0 \leq$ `i` $<$ `red`, then `threads[i]` is red.
- If `white` $<$ `i` $\leq$ `blue`, then `threads[i]` is white.
- If `blue` $<$ `i` $<$ `HEIGHT`, then `threads[i]` is blue.

What about the case where `red` $\leq$ `i` $\leq$ `white`? When the flag is all sorted, `red` should equal `white`, so this region should not exist. However, when we start, everything is in this region, so a thread in that region can have any color.

Thus, we can define the following loop invariant:

- If $0 \leq$ `i` $<$ `red`, then `threads[i]` is `red`.
- If `red` $\leq$ `i` $\leq$ `white`, then the color is unknown.
- If `white` $<$ `i` $\leq$ `blue`, then `threads[i]` is white.
- If `blue` $<$ `i` $<$ `HEIGHT`, then `threads[i]` is blue.

This is illustrated in Figure 8.21.

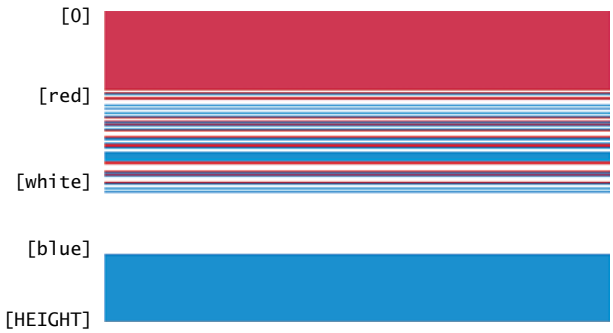


FIGURE 8.21
Dutch National Flag Loop Invariant

Algorithm

We can solve our problem by establishing the loop invariant and then executing a loop that both preserves the loop invariant and shrinks the unknown region.

1. Set `red` to `0`, `white` to `HEIGHT - 1`, and `blue` to `HEIGHT - 1`. This establishes our loop invariant with the unknown region the whole flag and the red, white, and blue regions empty.
2. `while red < white`
3. Shrink the distance between `red` and `white` while preserving the loop invariant.

Preserving the Loop Invariant

Let us assume that we now know the color of `threads[white]` (the thread at position `white`). Our goal is to either leave `threads[white]` where it is (in the white region if it is white) or "move it" to the region where it belongs. There are three cases to consider:

Case 1: The color of `threads[white]` is white. In this case, we merely decrement the value of white to restore the invariant. By doing so, we increase the size of the white region by one thread.

Case 2: The color of `threads[white]` is red. We know from our invariant that the color of `threads[red]` is unknown. Therefore, if we swap the thread at `threads[red]` with the one at `threads[white]`, we can then increment the value of red and preserve the invariant. By

doing this, we add the thread to the end of the red region and reduce the size of the unknown region by one thread.

Case 3: The color of `threads[white]` is blue. We know from our invariant that the color of `threads[blue]` is white. Thus, if we swap the thread at `threads[white]` with the thread at `threads[blue]` and then decrement both `white` and `blue`, we preserve the invariant.

By doing this, we insert the thread at the beginning of the blue region and reduce the size of the unknown region by one thread.

Implementation

We show the coding of the sort algorithm in Listing 8.12. Programming project 7 requires you to make modifications to the complete program.

LISTING 8.12
Dutch National Flag Sort

```
public void sort() {
    int red = 0;
    int white = height - 1;
    int blue = height - 1;
    /* Invariant:
       0 <= i < red ==>threads[i].getColor() == Color.RED
       red <= i <= white ==>threads[i].getColor() is unknown
       white < i < blue ==>threads[i].getColor() == Color.WHITE
       blue < i < height ==>threads[i].getColor() == Color.BLUE
     */
    while (red <= white) {
        if (threads[white].getColor() == Color.WHITE) {
            white--;
        } else if (threads[white].getColor() == Color.RED) {
            swap(red, white);
            red++;
        } else { // threads[white].getColor() == Color.BLUE
            swap(white, blue);
            white--;
            blue--;
        }
    }
    // assert: red > white so unknown region is now empty.
}
```

EXERCISES FOR SECTION 8.11

PROGRAMMING

1. Adapt the Dutch National Flag algorithm to do the quicksort partitioning. Consider the red region to be those values less than the pivot, the white region to be those values equal to the pivot, and the blue region to be those values equal to the pivot. You should initially sort the first, middle, and last items and use the middle value as the pivot value.

Chapter Review

- We analyzed several sorting algorithms; their performance is summarized in Table 8.6.
- Two quadratic algorithms, $O(n^2)$, are selection sort and insertion sort. They give satisfactory performance for small arrays (up to 100 elements). Generally, insertion sort is considered to be the best of the quadratic sorts.
- Shell sort, $O(n^{5/4})$, gives satisfactory performance for arrays up to 5000 elements.
- Quicksort has average-case performance of $O(n \log n)$, but if the pivot is picked poorly, the worst-case performance is $O(n^2)$.
- Merge sort and heapsort have $O(n \log n)$ performance.
- The Java API contains "industrial-strength" sort algorithms in the classes `java.util.Arrays` and `java.util.Collections`. The methods in `Arrays` use a mixture of quicksort and insertion sort for sorting arrays of primitive-type values and Timsort for sorting arrays of objects. For primitive types, quicksort is used until the size of the subarray reaches the point where insertion sort is quicker (seven elements or less). The sort method in `Collections` merely copies the list into an array and then calls `Arrays.sort`.

TABLE 8.6
Comparison of Sort Algorithms

	Number of Comparisons		
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Timsort	$O(n)$	$O(n \log n)$	$O(n \log n)$
Shell sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Java Classes Introduced in This Chapter

```
java.util.Arrays
java.util.Collections
```

User-Defined Classes in This Chapter

```
ComparePerson    Quicksort
HeapSort         SelectionSort
InsertionSort    ShellSort
MergeSort        Timsort
Person
```

Quick-Check Exercises

1. Name two quadratic sorts.
2. Name two sorts with n log n worst-case behavior.
3. Which algorithm is particularly good for an array that is already sorted? Which is particularly bad? Explain your answers.
4. What determines whether you should use a quadratic sort or a logarithmic sort?
5. Which quadratic sort's performance is least affected by the ordering of the array elements? Which is most affected?
6. What is a good all-purpose sorting algorithm for medium-size arrays?

Review Questions

1. When does quicksort work best, and when does it work worst?
2. Write a recursive procedure to implement the insertion sort algorithm.
3. What is the purpose of the pivot value in quicksort? How did we first select it in the text, and what is wrong with that approach for choosing a pivot value?
4. For the following array

   30 40 20 15 60 80 75 4 20

   show the new array after each pass of insertion sort and selection sort. How many comparisons and exchanges are performed by each?
5. For the array in Question 4, trace the execution of Shell sort.
6. For the array in Question 4, trace the execution of merge sort.
7. For the array in Question 4, trace the execution of quicksort.
8. For the array in Question 4, trace the execution of heapsort.

Programming Projects

1. Use the random number function to store a list of 1000 pseudorandom integer values in an array. Apply each of the sort classes described in this chapter to the array and determine the number of comparisons and exchanges. Make sure the same array is passed to each sort method.
2. Investigate the effect of array size and initial element order on the number of comparisons and exchanges required by each of the sorting algorithms described in this chapter. Use arrays with 100 and 10,000 integers. Use three initial orderings of each array (randomly ordered, inversely ordered, and ordered). Be certain to sort the same six arrays with each sort method.
3. A variation of the merge sort algorithm can be used to sort large sequential data files. The basic strategy is to take the initial data file, read in several (say, 10) data records, sort these records using an efficient array-sorting algorithm, and then write these sorted groups of records (runs) alternately to one of two output files. After all records from the initial data file have been distributed to the two output files, the runs on these output files are merged one pair of runs at a time and written to two other files. Runs no longer need to be sorted after the first distribution to the temporary output files.

   Each time runs are distributed to the alternate files; they contain twice as many records as the time before. The process stops when the length of the runs exceeds ½ the number of records in the data file. The final merge is then to the output file. Write a program that implements merge sort for sequential data files. Test your program on a file with several thousand data values.
4. Write a method that sorts a linked list.
5. Write an industrial-strength quicksort method with the following enhancements:

   If an array segment contains 20 elements or fewer, sort it using insertion sort.
6. In the early days of data processing (before computers), data was stored on punched cards. A machine to sort these cards contained 12 bins (one for each digit value and + and –). A stack of cards was fed into the machine, and the cards were placed into the appropriate bin depending on the value of the selected column. By restacking the cards so that all 0s were first, followed by the 1s, followed by

the 2s, and so forth, and then sorting on the next column, the whole deck of cards could be sorted. This process, known as *radix sort,* requires $c \times n$ passes, where c is the number of columns and n is the number of cards.

We can simulate the action of this machine using an array of queues. During the first pass, the least-significant digit (the ones digit) of each number is examined and the number is added to the queue whose subscript matches that digit. After all numbers have been processed, the elements of each queue are added to an 11th queue, starting with `queue[0]`, followed by `queue[1]`, and so forth. The process is then repeated for the next significant digit, taking the numbers out of the 11th queue. After all the digits have been processed, the 11th queue will contain the numbers in sorted order.

Write a program that implements radix sort on an array of `int` values. You will need to make 10 passes because an `int` can store numbers up to 2,147,483,648.

7. Project KW.CH08.dutchnationalflag in the student program file on the textbook website contains three Java classes that implement the solution to the Dutch National Flag problem. Execute the main method in file DutchNationaFlag wihich will display a scrambled flag in a frame on your screen similar to the one shown in Figure 8.19. Press Solve to unscramble the flag.

   Provide more complete documentation for this program. Modify it to display a flag that has a square shape instead of rectangular. Modify it to display a flag with red, white, blue, and black regions of the same size. Modify it so that the dimensions of each region is twice as large as the one above it in the flag. Finally, modify it to process a flag with vertical regions instead of horizontal.

Answers to Quick-Check Exercises

1. Selection sort, insertion sort
2. Merge sort, heapsort
3. Insertion sort—it requires $n - 1$ comparisons with no exchanges. Quicksort can be bad if the first element is picked as the pivot value because the partitioning process always creates one subarray with a single element.
4. Array size
5. Selection sort
6. Shell sort or any O(n log n) sort

Chapter 9

Self-Balancing Search Trees

Chapter Objectives

- To understand the impact that balance has on the performance of binary search trees
- To learn about the AVL tree for storing and maintaining a binary search tree in balance
- To learn about the Red–Black tree for storing and maintaining a binary search tree in balance
- To learn about 2–3 trees, 2–3–4 trees, and B-trees and how they achieve balance
- To understand the process of search and insertion in each of these trees and to be introduced to removal

In Chapter 6 we introduced the binary search tree. The performance (time required to find, insert, or remove an item) of a binary search tree is proportional to the total *height of the tree,* where we define the height of a tree as the maximum number of nodes along a path from the root to a leaf. A full binary tree of height k can hold $2^k - 1$ items. Thus, if the binary search tree were full and contained n items, the expected performance would be O(log n).

Unfortunately, if we build the binary search tree as described in Chapter 6, the resulting tree is not necessarily full or close to being full. Thus, the actual performance is worse than expected. In this chapter, we explore two algorithms for building binary search trees so that they are as full as possible. We call these trees *self-balancing* because they attempt to achieve a balance so that the height of each left subtree and right subtree is equal or nearly equal.

Finally, we look at the B-tree and its specializations, the 2–3 and 2–3–4 trees. These are not binary search trees, but they achieve and maintain balance.

In this chapter, we focus on algorithms and methods for search and insertion. We also discuss removing an item, but we have left the details of removal to the programming projects.

Self-Balancing Search Trees

9.1 Tree Balance and Rotation

Why Balance Is Important

Figure 9.1 shows an example of a valid, but extremely unbalanced, binary search tree. Searches or inserts into this tree would be O(n), not O(log n). Figure 9.2 shows the binary search tree resulting from inserting the words of the sentence "The quick brown fox jumps over the lazy dog." It too is not well balanced, having a height of 7 but containing only nine words. (Note that the string "The" is the smallest because it begins with an uppercase letter.)

FIGURE 9.1
Very Unbalanced Binary Search Tree

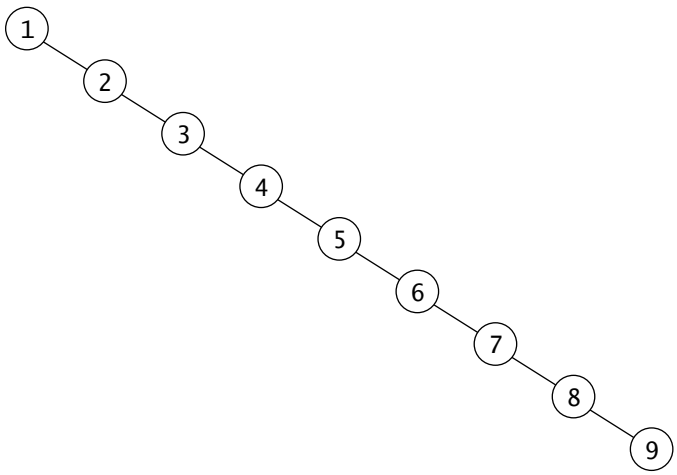
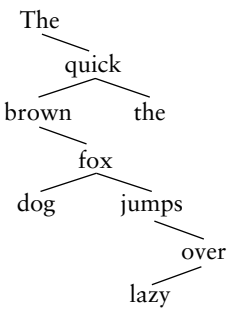


FIGURE 9.2
Realistic Example of an Unbalanced Binary Search Tree



Rotation

To achieve self-adjusting capability, we need an operation on a binary search tree that will change the relative heights of left and right subtrees but preserve the binary search tree property—that is, the items in each left subtree are less than the item at the root, and the items in each right subtree are greater than the item in the root. In Figure 9.3, we show an unbalanced binary search tree with a height of 4 right after the insertion of node 7. The height of the left subtree of the root (20) is 3, and the height of the right subtree is 1.

We can transform the tree in Figure 9.3 by doing a *right rotation* around node 20, making 10 the root and 20 the root of the right subtree of the new root (10). Because 20 is now the right subtree of 10, we need to move node 10's old right subtree (root is 15). We will make it the left subtree of 20, as shown in Figure 9.4.

FIGURE 9.3
Unbalanced Tree before Rotation

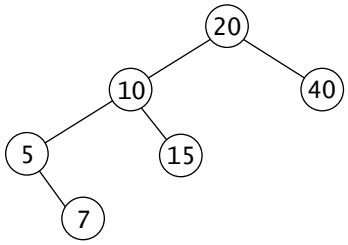


FIGURE 9.4
Right Rotation

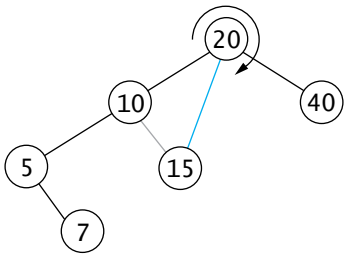
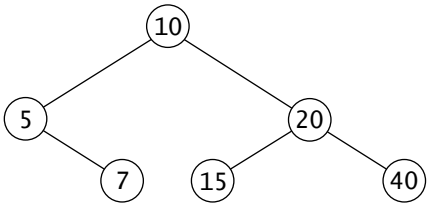


FIGURE 9.5
More Balanced Tree after Rotation



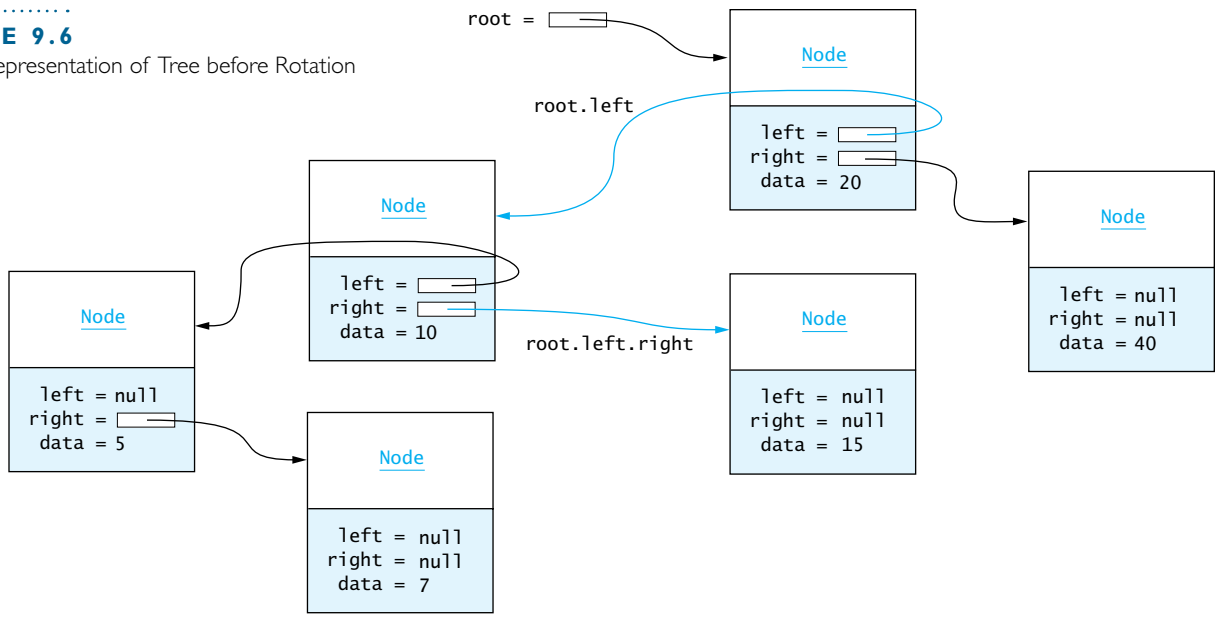
After these changes, the new binary search tree has a height of 3 (one less than before), and the left and right subtrees of the new root (10) have a height of 2, as shown in Figure 9.5. Note that the binary search tree property is maintained for all the nodes of the tree.

This result can be generalized. If node 15 had children, its children would have to be greater than 10 and less than 20 in the original tree. The left and right subtrees of node 15 would not change when node 15 was moved, so the binary search tree property would still be maintained for all children of node 15 in the new tree (>10 and < 20). We can make a similar statement for any of the other leaf nodes in the original tree.

Algorithm for Rotation

Figure 9.6 illustrates the internal representation of the nodes of our original binary search tree whose three branches (shown in color) will be changed by rotation. Initially, `root` references node 20. Rotation right is achieved by the following algorithm.

FIGURE 9.6
Internal Representation of Tree before Rotation



Algorithm for Rotation Right

1. Remember the value of `root.left` (`temp = root.left`).
2. Set `root.left` to the value of `temp.right`.
3. Set `temp.right` to `root`.
4. Set `root` to `temp`.

Figure 9.7 shows the rotated tree. Step 1 sets `temp` to reference the left subtree (node 10) of the original root. Step 2 resets the original root's left subtree to reference node 15. Step 3 resets node `temp`'s right subtree to reference the original root. Then Step 4 sets `root` to reference node `temp`. The internal representation corresponds to the tree shown in Figure 9.5.

The algorithm for rotation left is symmetric to rotation right and is left as an exercise.

Implementing Rotation

Listing 9.1 shows class `BinarySearchTreeWithRotate`. This class is an extension of the `BinarySearchTree` class described in Chapter 6, and it will be used as the base class for the other search trees discussed in this chapter. Like class `BinarySearchTree`, class `BinarySearchTreeWithRotate` must be declared as a generic class with type parameter `<E extends Comparable<E>>`. It contains the methods `rotateLeft` and `rotateRight`. These methods take a reference to a `Node` that is the root of a subtree and return a reference to the root of the rotated tree.

Figure 9.8 is a UML (Unified Modeling Language) class diagram that shows the relationships between `BinarySearchTreeWithRotate` and the other classes in the hierarchy. `BinarySearchTreeWithRotate` is a subclass of `BinaryTree` as well as `BinarySearchTree`. Class `BinaryTree` has the static inner class `Node` and the data field `root`, which references the `Node` that is the root of the tree. The figure shows that a `Node` contains a data field named `data` and two references (as indicated by the open diamond) to a `Node`. The names of the reference are `left` and `right`, as shown on the line from the `Node` to itself. We cover UML in Appendix B.

FIGURE 9.7
Effect of Rotation Right on Internal Representation

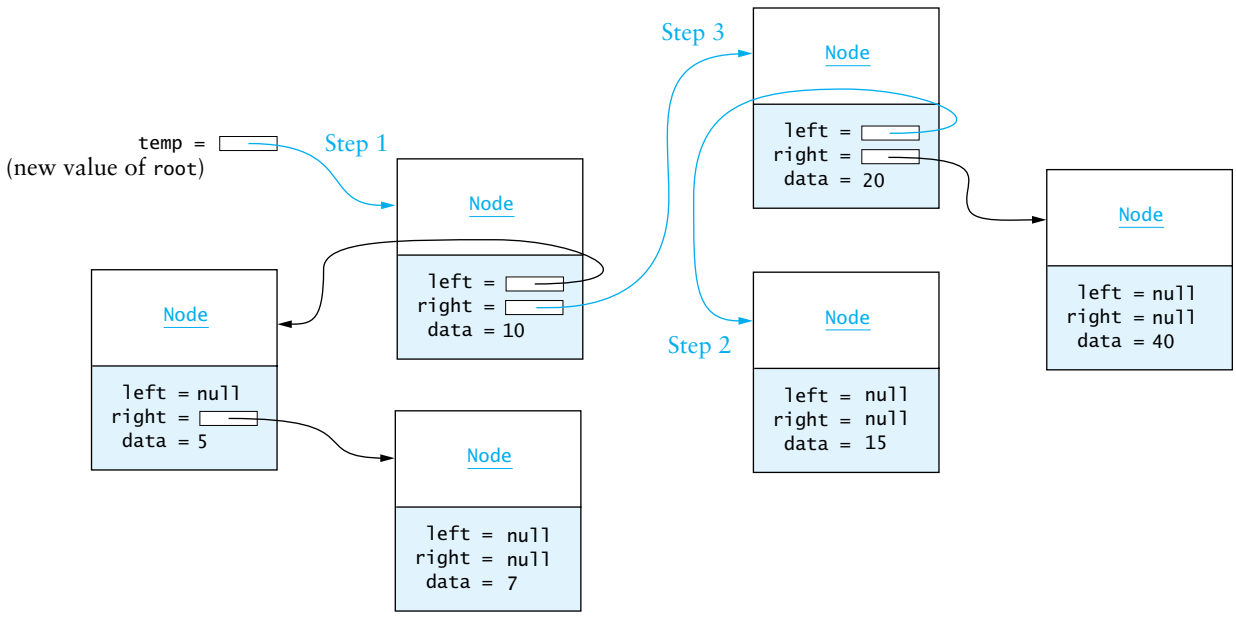
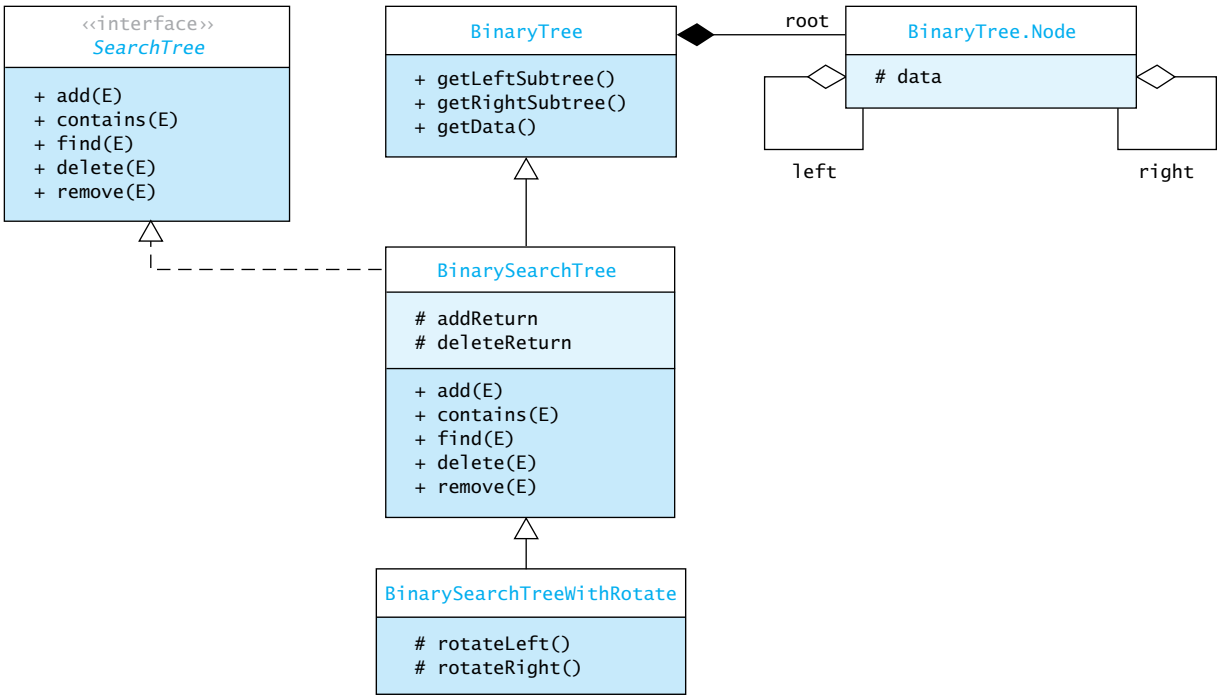


FIGURE 9.8
UML Diagram of `BinarySearchTreeWithRotate`



LISTING 9.1
BinarySearchTreeWithRotate.java

```
/** This class extends the BinarySearchTree by adding the rotate operations.
    Rotation will change the balance of a search tree while preserving the
    search tree property.
    Used as a common base class for self-balancing trees.
 */
public class BinarySearchTreeWithRotate<E extends Comparable<E>>
             extends BinarySearchTree<E> {
    // Methods
    /** Method to perform a right rotation.
        pre: root is the root of a binary search tree.
        post: root.right is the root of a binary search tree,
              root.right.right is raised one level,
              root.right.left does not change levels,
              root.left is lowered one level.
        @param root The root of the binary tree to be rotated
        @return The new root of the rotated tree
     */

    protected Node<E> rotateRight(Node<E> root) {
        Node<E> temp = root.left;
        root.left = temp.right;
        temp.right = root;
        return temp;
    }

    /** Insert method to perform a left rotation (rotateLeft).
        See Programming Exercise 1 */
}
```

EXERCISES FOR SECTION 9.1

SELF-CHECK

1. Draw the binary search tree that results from inserting the words of the sentence "Now is the time for all good people to come to the aid of others." What is its height? Compare this with 4, the smallest integer greater than $\log_2 13$, where 13 is the number of distinct words in this sentence.
2. Try to construct a binary search tree that contains the same words as in Exercise 1 but has a maximum height of 4.
3. Describe the algorithm for rotation left.

PROGRAMMING

1. Add the `rotateLeft` method to the `BinarySearchTreeWithRotate` class.

9.2 AVL Trees

Two Russian mathematicians, G. M. Adel'son-Vel'skiî and E. M. Landis, published a paper in 1962 that describes an algorithm for maintaining overall balance of a binary search tree. Their algorithm keeps track of the difference in height of each subtree. As items are added to (or removed from) the tree, the balance (i.e., the difference in the heights of the subtrees) of each subtree from the insertion point up to the root is updated. If the balance ever gets out of the range –1 ... +1, the subtree is rotated to bring it back into balance. Trees using this approach are known as *AVL trees* after the initials of the inventors. As before, we define the height of a tree as the number of nodes in the longest path from the root to a leaf node, including the root.

Balancing a Left–Left Tree

Figure 9.9 shows a binary search tree with a balance of –2 caused by an insert into its left–left subtree. This tree is called a Left–Left tree because its root and the left subtree of the root are both left-heavy. Each white triangle with label `a`, `b`, or `c` represents a tree of height k; the area in color at the bottom of the left–left triangle (tree `a`) indicates an insertion into this tree (its height is now k +1).

We use the formula

$$h_R - h_L$$

to calculate the balance for each node, where h_L and h_R are the heights of the left and right subtrees, respectively. The actual heights are not important; it is their relative difference that matters. The right subtree (`b`) of node 25 has a height of k; its left subtree (`a`) has a height of k +1, so its balance is –1. The right subtree of node 50 has a height of k; its left subtree has a height of k + 2 (adding one for node 25), so its balance is –2.

Figure 9.10 shows this same tree after a rotation right. The new tree root is node 25. Its right subtree (root 50) now has tree `b` as its left subtree. Note that balance has now been achieved. Also, the overall height has not increased. Before the insertion, the tree height was k + 2; after the rotation, the tree height is still k + 2.

FIGURE 9.9
Left-Heavy Tree

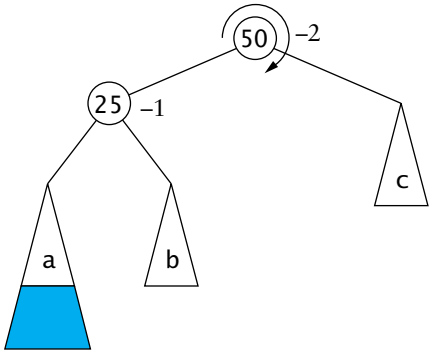
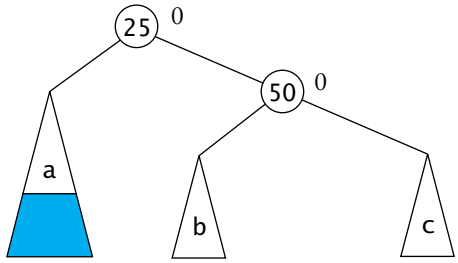


FIGURE 9.10
Left-Heavy Tree after Rotation Right



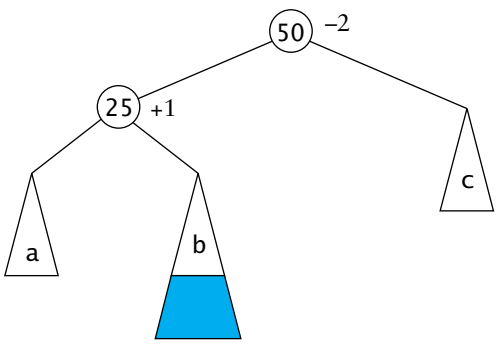
Balancing a Left–Right Tree

Figure 9.11 shows a left-heavy tree caused by an insert into the Left–Right subtree. This tree is called a Left–Right tree because its root is left-heavy but the left subtree of the root is right-heavy. We cannot fix this with a simple rotation right as in the Left–Left case. (See Self-Check Exercise 2 at the end of this section.) We show how to fix this by looking more closely at the insertion into subtree b next.

Figure 9.12 shows the same tree as in Figure 9.11 but with subtree b expanded into node 40 and its subtrees b_L and b_R, which are both height $k - 1$. There is an insertion into b_L, which makes node 40 left-heavy. This change ripples up the tree, causing node 25 to become right-heavy and node 50 left-heavy resulting in a left–right tree as shown Figure 9.12. If the left subtree is rotated left (shown in Figure 9.13) the overall tree is now a Left–Left tree, similar to the case of Figure 9.9. Now if the modified tree is rotated right, overall balance is achieved, as shown in Figure 9.14. Figures 9.15–9.17 illustrate the effect of these double rotations after insertion into b_R instead of b_L.

In both cases, the new tree root is 40; its left subtree has node 25 as its root, and its right subtree has node 50 as its root. The balance of the root is 0. If the critically unbalanced situation was due to an insertion into subtree b_L, the balance of the root's left child is 0, and the balance of the root's right child is +1 (Figure 9.14). For insertion into subtree b_R, the balance of the root's left child is –1, and the balance of the root's right child is 0 (Figure 9.17).

FIGURE 9.11
Left–Right Tree



Balance 50 = $(k - (k + 2))$

Balance 25 = $((k + 1) - k)$

FIGURE 9.12
Insertion into b_L

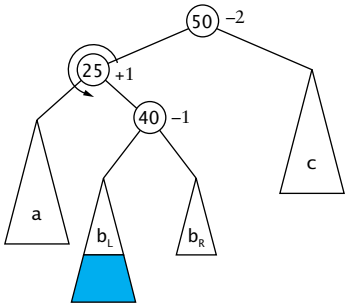


FIGURE 9.13
Left Subtree after Rotate left

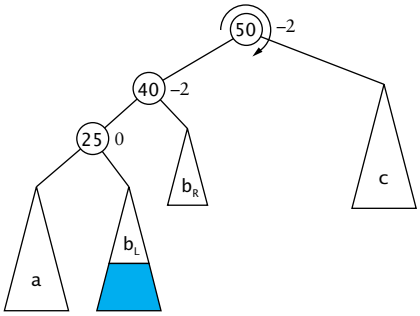


FIGURE 9.14
Tree after Rotate Right

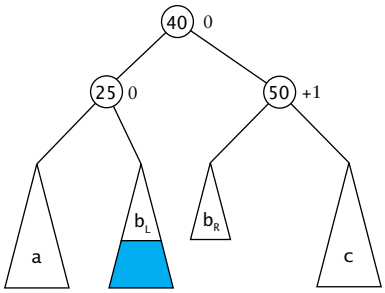


FIGURE 9.15
Insertion into b_R

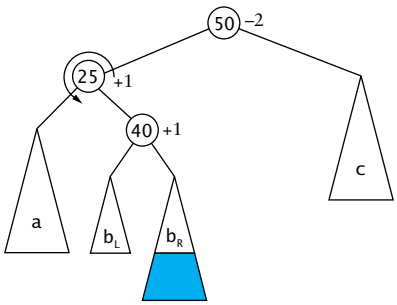


FIGURE 9.16
Left Subtree after Rotate Left

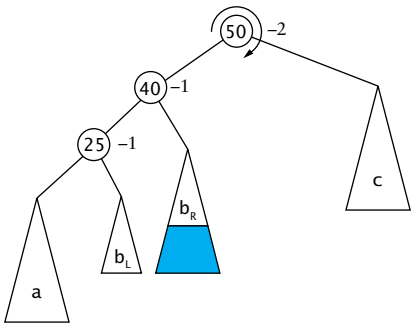
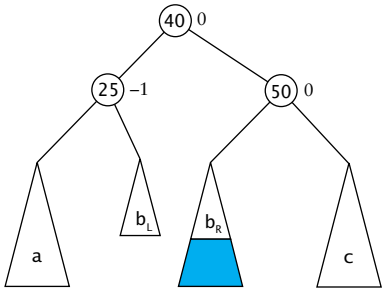


FIGURE 9.17
Tree after Rotate Right



Four Kinds of Critically Unbalanced Trees

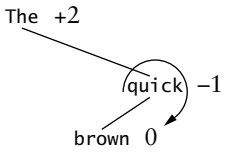
How do we recognize unbalanced trees and determine what to do to balance them? For the Left–Left tree shown in Figure 9.9 (parent and child nodes are both left-heavy, parent balance is –2, child balance is –1), the remedy is to rotate right around the parent.

For the Left–Right example shown in Figure 9.11 (parent is left-heavy with balance –2, child is right-heavy with balance +1), the remedy is to rotate left around the child and then rotate right around the parent. We list the four cases that need rebalancing and their remedies next.

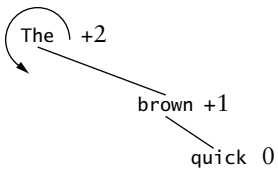
- Left–Left (parent balance is –2, left child balance is –1): rotate right around parent.
- Left–Right (parent balance is –2, left child balance is +1): rotate left around child, then rotate right around parent.
- Right–Right (parent balance is +2, right child balance is +1): rotate left around parent.
- Right–Left (parent balance is +2, right child balance is –1): rotate right around child, then rotate left around parent.

EXAMPLE 9.1 We will build an AVL tree from the words in the sentence "The quick brown fox jumps over the lazy dog".

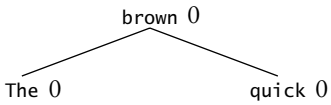
After inserting the words *The*, *quick*, and *brown*, we get the following tree.



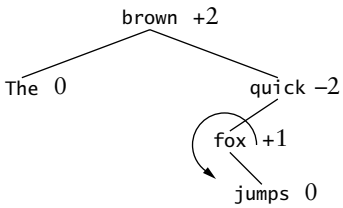
The subtree with the root *quick* is left-heavy by 1, but the overall tree with the root of *The* is right-heavy by 2 (Right–Left case). We must first rotate the subtree around *quick* to the right:

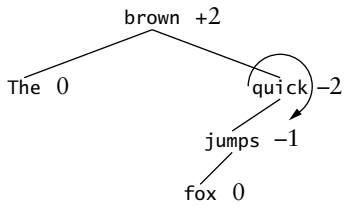


Then rotate left about *The*:

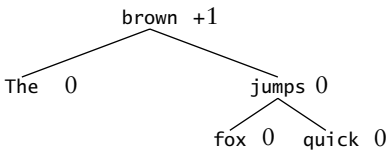


We now proceed to insert *fox* and *jumps*:

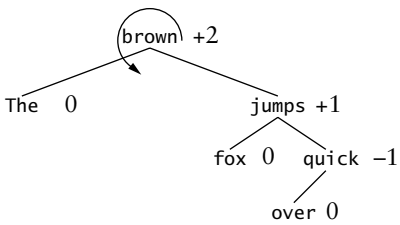




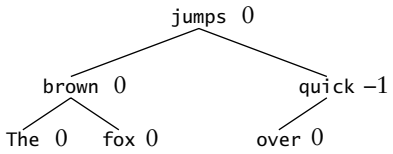
The subtree rooted about *quick* is now left-heavy by 2 and *fox* is right-heavy by 1 (Left–Right case shown above on the left). To fix this we rotate left about *fox* (shown above on the right) and then right about *quick*, giving the following result.



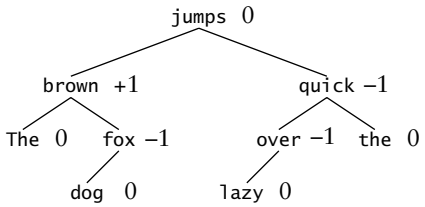
We now insert *over*:



The subtrees at *quick* and *jumps* are unbalanced by 1. The overall tree, however, is right-heavy by 2 (Right–Right case), so a rotation left solves the problem.



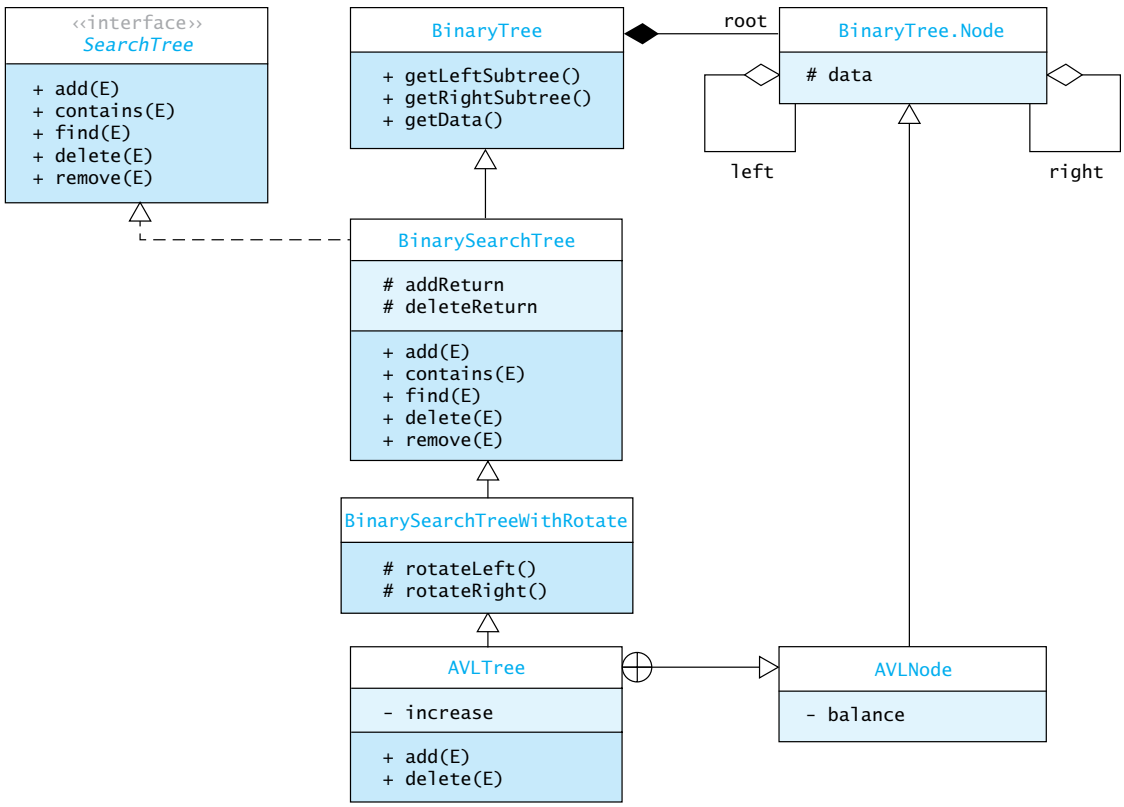
We can now insert *the*, *lazy*, and *dog* without any additional rotations being necessary.



Implementing an AVL Tree

We begin by deriving the class `AVLTree` from `BinarySearchTreeWithRotate` (see Listing 9.1). Figure 9.18 is a UML class diagram showing the relationship between `AVLTree` and `BinarySearchTreeWithRotate`. The `AVLTree` class contains the **`boolean`** data field

FIGURE 9.18
UML Class Diagram of AVLTree



SYNTAX UML Syntax

The line from the `AVLTree` class to the `AVLNode` class in the diagram in Figure 9.18 indicates that methods in the `AVLTree` class can access the private data field `balance`. The symbol ⊕ next to the `AVLTree` class indicates that the `AVLNode` class is an inner class of `AVLTree`. The arrow pointing to `AVLNode` indicates that methods in `AVLTree` access the contents of `AVLNode`, but methods in `AVLNode` do not access the contents of `AVLTree`.

Note that the `Node` class is an inner class of the `BinaryTree` class, but we do not show the ⊕. This is because an object of type `Node`, called `root`, is a component of the `BinaryTree` class, as indicated by the filled diamond next to the `BinaryTree` class. Showing both the ⊕ and the filled diamond would clutter the diagram, so only the filled diamond is shown.

`increase`, which indicates whether the current subtree height has increased as a result of the insertion.

We override the methods `add` and `delete` but inherit method `find` because searching a balanced tree is not different from searching an unbalanced tree. We also extend the inner class `BinaryTree.Node` with `AVLNode`. Within this class we add the additional field `balance`.

```
/** Self-balancing binary search tree using the algorithm defined
    by Adelson-Velskii and Landis.
 */
public class AVLTree<E extends Comparable<E>>
             extends BinarySearchTreeWithRotate<E> {
    // Insert nested class AVLNode<E> here.
    // Data Fields
    /** Flag to indicate that height of tree has increased. */
    private boolean increase;
    . . .
```

The AVLNode Class

The `AVLNode` class is shown in Listing 9.2. It is an extension of the `BinaryTree.Node` class. It adds the data field `balance` and the constants `LEFT_HEAVY`, `BALANCED`, and `RIGHT_HEAVY`.

LINSTING 9.2
The `AVLNode` Class

```
/** Class to represent an AVL Node. It extends the
    BinaryTree.Node by adding the balance field.
 */
private static class AVLNode<E> extends Node<E> {
    /** Constant to indicate left-heavy */
    public static final int LEFT_HEAVY = -1;
    /** Constant to indicate balanced */
    public static final int BALANCED = 0;
    /** Constant to indicate right-heavy */
    public static final int RIGHT_HEAVY = 1;
    /** balance is right subtree height - left subtree height */
    private int balance;
    // Methods
```

```
    /** Construct a node with the given item as the data field.
        @param item The data field
     */
    public AVLNode(E item) {
        super(item);
        balance = BALANCED;
    }

    /** Return a string representation of this object.
        The balance value precedes the contents.
        @return String representation of this object
     */
    @Override
    public String toString() {
        return balance + ": " + super.toString();
    }
}
```

Inserting into an AVL Tree

The easiest way to keep a tree balanced is never to let it become unbalanced. If any node becomes critical and needs rebalancing, rebalance immediately. You can identify critical nodes by checking the balance at the root node of a subtree as you return to each parent node along the insertion path. If the insertion was in the left subtree and the left subtree height has increased, you must check to see whether the balance for the root node of the left subtree has become critical (–2 or +2). If so, you need to fix it by calling `rebalanceLeft` (rebalance a left-heavy tree when balance is –2) or `rebalanceRight` (rebalance a right-heavy tree when balance is +2). A symmetric strategy should be followed after returning from an insertion into the right subtree. The **`boolean`** variable `increase` is set before the return from recursion to indicate to the next higher level that the height of the subtree has increased. This information is then used to adjust the balance of the next level in the tree. The following algorithm is based on the algorithm for inserting into a binary search tree, described in Chapter 6.

Algorithm for Insertion into an AVL Tree

1. **`if`** the `root` is **`null`**
2. Create a new tree with the item at the `root` and `return` **`true`**.

   `else if` the item is equal to `root.data`
3. The item is already in the tree; return **`false`**.

   `else if` the item is less than `root.data`
4. Recursively insert the item in the left subtree.
5. **`if`** the height of the left subtree has increased (`increase` is **`true`**)
6. Decrement `balance`.
7. **`if`** `balance` is zero, reset `increase` to **`false`**.
8. **`if`** `balance` is less than –1
9. Reset `increase` to **`false`**.
10. Perform a `rebalanceLeft`.

    `else if` the item is greater than `root.data`
11. The processing is symmetric to Steps 4 through 10. Note that `balance` is incremented if `increase` is **`true`**.

After returning from the recursion (Step 4), examine the global data field `increase` to see whether the left subtree has increased in height. If it did, then decrement the balance. If the balance had been +1 (current subtree was right-heavy), it is now zero, so the overall height of the current subtree is not changed. Therefore, reset `increase` to **`false`** (Steps 5–7).

If the balance was –1 (current subtree was left-heavy), it is now –2, and a `rebalanceLeft` must be performed. The rebalance operation reduces the overall height of the tree by 1, so increase is reset to **`false`**. Therefore, no more rebalancing operations will occur, so you can fix the tree by either a single rotation (Left–Left case) or a double rotation (Left–Right case) (Steps 8–10).

add Starter Method

We are now ready to implement the insertion algorithm. The `add` starter method merely calls the recursive `add` method with the root as its argument. The returned `AVLNode` is the new root.

```
/** add starter method.
    pre: the item to insert implements the Comparable interface.
    @param item The item being inserted.
    @return true if the object is inserted; false
            if the object already exists in the tree
    @throws ClassCastException if item is not Comparable
 */
@Override
public boolean add(E item) {
    increase = false;
    root = add((AVLNode<E>) root, item);
    return addReturn;
}
```

As for the `BinarySearchTree` in Chapter 6, the recursive `add` method will set the data field `addReturn` to **`true`** (inherited from class `BinarySearchTree`) if the item is inserted and **`false`** if the item is already in the tree.

Recursive add Method

The declaration for the recursive `add` method begins as follows:

```
/** Recursive add method. Inserts the given object into the tree.
    post: addReturn is set true if the item is inserted,
          false if the item is already in the tree.
    @param localRoot The local root of the subtree
    @param item The object to be inserted
    @return The new local root of the subtree with the item inserted
 */
private AVLNode<E> add(AVLNode<E> localRoot, E item)
```

We begin by seeing whether the `localRoot` is **`null`**. If it is, then we set `addReturn` and `increase` to **`true`** and return a new `AVLNode`, which contains the item to be inserted.

```
if (localRoot == null) {
    addReturn = true;
    increase = true;
    return new AVLNode<E>(item);
}
```

Next, we compare the inserted item with the data field of the current node. If it is equal, we set `addReturn` and `increase` to **`false`** and return the `localRoot` unchanged.

```
if (item.compareTo(localRoot.data) == 0) {
    // Item is already in the tree.
    increase = false;
    addReturn = false;
    return localRoot;
}
```

If it is less than this value, we recursively call the `add` method (Step 4 of the insertion algorithm), passing `localRoot.left` as the parameter and replacing the value of `localRoot.left` with the returned value.

```
else if (item.compareTo(localRoot.data) < 0) {
    // item < data
    localRoot.left = add((AVLNode<E>) localRoot.left, item);
    . . .
```

Upon return from the recursion, we examine the global data field `increase`. If `increase` is **`true`**, then the height of the left subtree has increased, so we decrement the balance by calling the `decrementBalance` method. If the balance is now less than –1, we reset `increase` to **`false`** and call the `rebalanceLeft` method. The return value from the `rebalanceLeft` method is the return value from this call to `add`. If the balance is not less than –1, or if the left subtree height did not increase, then the return from this recursive call is the same local root that was passed as the parameter.

```
if (increase) {
    decrementBalance(localRoot);
    if (localRoot.balance < AVLNode.LEFT_HEAVY) {
        increase = false;
        return rebalanceLeft(localRoot);
    }
}
// Rebalance not needed.
return localRoot;
```

If the item is not equal to `localRoot.data` and not less than `localRoot.data`, then it must be greater than `localRoot.data`. The processing is symmetric with the less-than case and is left as an exercise.

Initial Algorithm for `rebalanceLeft`

Method `rebalanceLeft` rebalances a left-heavy tree. Such a tree can be a Left–Left tree (fixed by a single right rotation) or a Left–Right tree (fixed by a left rotation followed by a right rotation). If its left subtree is right-heavy, we have a Left–Right case, so we first rotate left around the left subtree. Finally, we rotate the tree right.

1. **`if`** the left subtree has positive balance (Left–Right case)
2. Rotate left around left subtree root.
3. Rotate right.

The Effect of Rotations on Balance

The rebalancing algorithm just presented is incomplete. So far we have focused on changes to the root reference and to the internal branches of the tree being balanced, but we have not adjusted the balances of the nodes. In the beginning of this section, we showed that for a Left–Left tree, the balances of the new root node and of its right

child are 0 after a right rotation (see Figure 9.10); so the balances of all other nodes are unchanged.

The Left–Right case is more complicated. We made the following observation after studying the different cases.

The balance of the root is 0. If the critically unbalanced situation was due to an insertion into subtree b_L, the balance of the root's left child is 0 and the balance of the root's right child is + 1 (Figure 9.14). For insertion into subtree b_R, the balance of the root's left child is –1, and the balance of the root's right child is 0 (Figure 9.17). So we need to change the balances of the new root node and both its left and right children; all other balances are unchanged. We will call insertion into subtree b_L the Left–Right–Left case and insertion into subtree b_R the Left–Right–Right case.

There is a third case where the left–right subtree is balanced; this occurs when a left–right leaf is inserted into a subtree that has only a left child. In this case after the rotations are performed, the root, left child, and right child are all balanced.

Revised Algorithm for `rebalanceLeft`

Based on the foregoing discussion, we can now develop the complete algorithm for `rebalanceLeft`, including the required balance changes. It is easier to store the new balance for each node before the rotation than after.

1. **if** the left subtree has a positive balance (Left–Right case)
2. **if** the left–left subtree has a negative balance (Left–Right–Left case)
3. Set the left subtree (new left subtree) balance to 0.
4. Set the left–left subtree (new root) balance to 0.
5. Set the local root (new right subtree) balance to +1.

else (Left–Right–Right case)

6. Set the left subtree (new left subtree) balance to –1.
7. Set the left–left subtree (new root) balance to 0.
8. Set the local root (new right subtree) balance to 0.
9. Rotate the left subtree left.

else (Left–Left case)

10. Set the left subtree balance to 0.
11. Set the local root balance to 0.
12. Rotate the local root right.

The algorithm for `rebalanceRight` is symmetric. Exchange left and right and –1 and +1.

Method `rebalanceLeft`

The code for `rebalanceLeft` is shown in Listing 9.3. First, we test to see whether the left subtree is right-heavy (Left–Right case). If so, the Left–Right subtree is examined.

Depending on its balance, the balances of the left subtree and local root are set as previously described in the algorithm. The rotations will reduce the overall height of the tree by 1, so `increase` is now set to **`false`**. The left subtree is then rotated left, and the tree is rotated right.

LISTING 9.3
The `rebalanceLeft` Method

```
/** Method to rebalance left.
    pre: localRoot is the root of an AVL subtree that is critically left-heavy.
    post: Balance is restored.
    @param localRoot Root of the AVL subtree that needs rebalancing
    @return a new localRoot
 */
private AVLNode<E> rebalanceLeft(AVLNode<E> localRoot) {
    // Obtain reference to left child.
    var leftChild = (AVLNode<E>) localRoot.left;
    // See whether left-right heavy.
    if (leftChild.balance > AVLNode.BALANCED) {
        // Obtain reference to left-right child.
        var leftRightChild = (AVLNode<E>) leftChild.right;
        /** Adjust the balances to be their new values after
            the rotations are performed.
         */
        if (leftRightChild.balance < AVLNode.BALANCED) {
            leftChild.balance = AVLNode.BALANCED;
            leftRightChild.balance = AVLNode.BALANCED;
            localRoot.balance = AVLNode.RIGHT_HEAVY;
        } else {
            leftChild.balance = AVLNode.LEFT_HEAVY;
            leftRightChild.balance = AVLNode.BALANCED;
            localRoot.balance = AVLNode.BALANCED;
        }
        // Perform left rotation.
        localRoot.left = rotateLeft(leftChild);
    } else {   // Left-Left case
        /** In this case the leftChild (the new root) and the root
            (new right child) will both be balanced after the rotation.
         */
        leftChild.balance = AVLNode.BALANCED;
        localRoot.balance = AVLNode.BALANCED;
    }
    // Now rotate the local root right.
    return (AVLNode<E>) rotateRight(localRoot);
}
```

If the left child is `LEFT_HEAVY`, the rotation process will restore the balance to both the tree and its left subtree and reduce the overall height by 1; the balance for the left subtree and local root are both set to `BALANCED`, and increase is now set to **`false`**. The tree is then rotated right to correct the imbalance.

We also need a `rebalanceRight` method that is symmetric with `rebalanceLeft` (i.e., all `lefts` are changed to `rights` and all `rights` are changed to `lefts`). Coding of this method is left as an exercise.

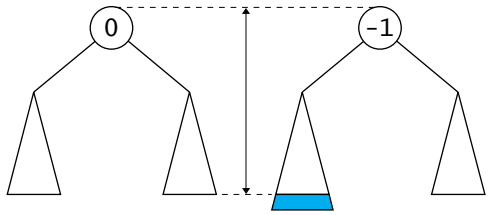
The `decrementBalance` Method

As we return from an insertion into a node's left subtree, we need to decrement the balance of the node. We also need to indicate whether the subtree height at that node has not increased, by setting `increase` (currently **`true`**) to **`false`**. There are two cases to consider: a node that is balanced and a node that is right-heavy. If a node is balanced, insertion into its left subtree will cause it to become left-heavy, and its height will also increase by 1 (see Figure 9.19). If a node is right-heavy, insertion into its left subtree will cause it to become balanced, and its height will not increase (see Figure 9.20).

```
private void decrementBalance(AVLNode<E> node) {
    // Decrement the balance.
    node.balance--;
    if (node.balance == AVLNode.BALANCED) {
        /** If now balanced, overall height has not increased. */
        increase = false;
    }
}
```

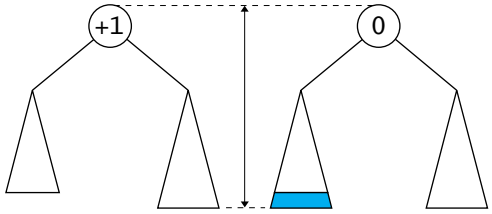
Step 11 of the insertion algorithm performs insertion into a right subtree. This can cause the height of the right subtree to increase, so we will also need an `incrementBalance` method that increments the balance and resets `increase` to **`false`** if the balance changes from left-heavy to balanced. Coding this method is left as an exercise.

FIGURE 9.19
Decrement of `balance` by Insert on Left (Height Increases)



`balance` before insert is 0 — `balance` is decreased due to insert; overall height increased

FIGURE 9.20
Decrement of `balance` by Insert on Left (Height Does Not Change)



`balance` before insert is +1 — `balance` is decreased due to insert; overall height remains the same

Removal from an AVL Tree

When we remove an item from a left subtree, the balance of the local root is increased, and when we remove an item from the right subtree, the balance of the local root is decreased. We can adapt the algorithm for removal from a binary search tree to become an algorithm for removal from an AVL tree. We need to maintain a data field `decrease` that tells the previous level in the recursion that there was a decrease in the height of the subtree that was just returned from. (This data field is analogous to the data field `increase`, which is used in the insertion to indicate that the height of the subtree has increased.) We can then increment or decrement the local root balance. If the balance is outside the threshold, then the rebalance methods (`rebalanceLeft` or `rebalanceRight`) are used to restore the balance.

We need to modify methods `decrementBalance`, `incrementBalance`, `rebalanceLeft`, and `rebalanceRight` so that they set the value of `decrease` (as well as `increase`) after a node's

balance has been decremented. When a subtree changes from either left-heavy or right-heavy to balanced, then the height has decreased, and `decrease` should be set **`true`**; when the subtree changes from balanced to either left-heavy or right-heavy, then `decrease` should be reset to **`false`**. We also need to provide methods similar to the ones needed for removal in a binary search tree. Implementing removal is left as a programming project.

Also, observe that the effect of rotations is not only to restore balance but to decrease the height of the subtree being rotated. Thus, while only one `rebalanceLeft` or `rebalanceRight` was required for insertion, during removal each recursive return could result in a further need to rebalance.

Performance of the AVL Tree

Since each subtree is kept as close to balanced as possible, one would expect that the AVL tree provides the expected O(log *n*) performance. Each subtree is allowed to be out of balance by ±1. Thus, the tree may contain some holes.

It can be shown that in the worst case, the height of an AVL tree can be 1.44 times the height of a full binary tree that contains the same number of items. However, this would still yield O(log *n*) performance because we ignore constants.

The worst-case performance is very rare. Empirical tests (see, e.g., Donald Knuth, *The Art of Computer Programming, Vol 3: Searching and Sorting* [Addison-Wesley, 1973], p. 460) show that, on the average, log *n* + 0.25 comparisons are required to insert the *n*th item into an AVL tree. Thus, the average performance is very close to that of the corresponding complete binary search tree.

EXERCISES FOR SECTION 9.2

SELF-CHECK

1. Show how the final AVL tree for the "The quick brown fox" changes as you insert "apple", "cat", and "hat" in that order.
2. Show the effect of just rotating right on the tree in Figure 9.11. Why doesn't this fix the problem?
3. Build an AVL tree that inserts the integers 30, 40, 15, 25, 90, 80, 70, 85, 15, 72 in the given order.
4. Build the AVL tree from the sentence "Now is the time for all good people to come to the aid of others".

PROGRAMMING

1. Program the `rebalanceRight` method.
2. Program the code in the `add` method for the case where `item.compareTo(localRoot.data) > 0`.
3. Program the `incrementBalance` method.

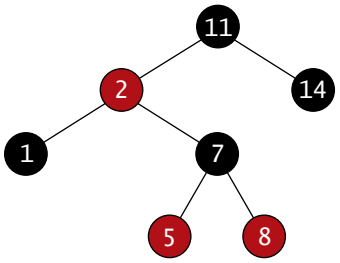
9.3 Red–Black Trees

We will now discuss another approach to keeping a tree balanced, called the *Red–Black tree*. Rudolf Bayer developed the Red–Black tree as a special case of his B-tree (the topic of Section 9.5); Leo Guibas and Robert Sedgewick refined the concept and introduced the color convention. A Red–Black tree maintains the following invariants:

1. A node is either red or black.
2. The root is always black.
3. A red node always has black children. (A `null` reference is considered to refer to a black node.)
4. The number of black nodes in any path from the root to a leaf is the same.

Figure 9.21 shows an example of a Red–Black tree. Invariant 4 states that a Red–Black tree is always balanced because the root node's left and right subtrees must be the same height where the height is determined by counting just black nodes. Note that by the standards of the AVL tree, this tree is out of balance and would be considered a Left–Right tree. However, by the standards of the Red–Black tree, it is balanced because there are two black nodes (counting the root) in any path from the root to a leaf.

FIGURE 9.21
Red–Black Tree

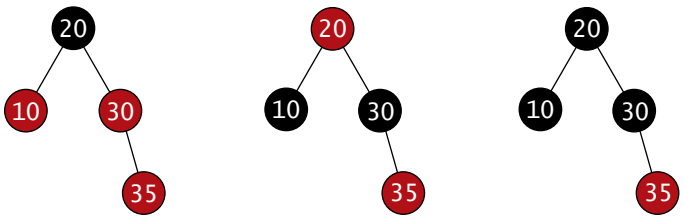


Insertion into a Red–Black Tree

The algorithm for insertion follows the same recursive search process used for all binary search trees to reach the insertion point. When a leaf is found, the new item is inserted, and it is initially given the color red, so invariant 4 will be maintained. If the parent is black, we are done.

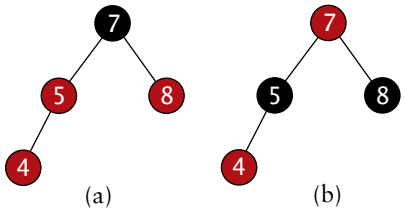
However, if the parent is also red, then invariant 3 has been violated, so we must fix this problem. The solution is relatively easy if the parent has a red sibling (Case 1). Figure 9.22(a) shows the insertion of 35 as a red child of 30. If the parent's sibling is also red, then we can change the grandparent's color to red and change both the parent and parent's sibling to black. If the grandparent is the root of the overall tree, we can change its color to black to restore invariant 2 and still maintain invariant 4 (the heights of all paths to a leaf are increased by 1) (see Figure 9.22(c)).

FIGURE 9.22
Insertion into a Red–Black Tree, Case 1



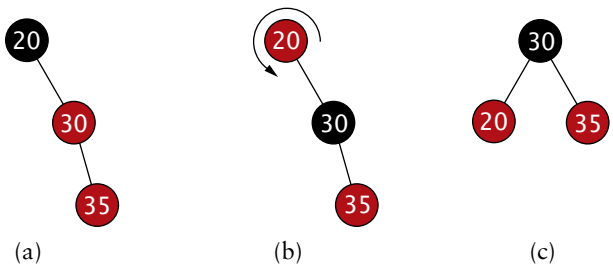
Ignoring the node values, Figure 9.23(a) is the mirror image of Figure 9.22(a) where both the parent (5) and the inserted child (4) are left children and the parent has a red sibling (8). The insertion of 4 violates invariant 3. Figure 9.23(b) shows the tree after recoloring. As in Figure 9.22c, if the grandparent (7) is the root of the overall tree, it should be recolored to black.

FIGURE 9.23
Case 1a—Parent and Child Nodes Are Left Children



Case 2 is the situation where we insert a value with a red parent, but that parent does not have a red sibling (see Figure 9.24(a)). The solution is more complicated. We change the color of the grandparent (20) to red and the parent (30) to black (see Figure 9.24(b)). However invariant 4 is violated because the right subtree of the root has more black nodes (1) than its right side (0). We correct this by rotating left around the grandparent so that the parent moves into the position where the grandparent was, thus restoring invariant 4 (see Figure 9.24(c)).

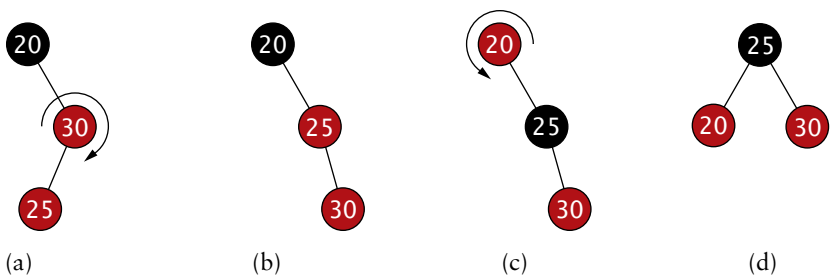
FIGURE 9.24
Insertion into a Red–Black Tree, Case 2



The mirror image of Case 2 (Case 2a) occurs when the red parent is a left child of the grandparent and the inserted node is also a left child. This would be corrected by recoloring the parent and grandparent and rotating right around the grandparent.

Figure 9.25(a) shows 25 inserted as the left child of a node that is a right child (30). This is Case 3. It requires a double rotation. First, we rotate right around the parent so that the new right child (30) and its new parent (25) are both right children (Figure 9.25(b)), which is Case 2. We resolve this Case 2 situation by recoloring the new parent and the grandparent (Figure 9.25(c)) and rotating left around the grandparent (Figure 9.25(d)).

FIGURE 9.25
Insertion into a Red–Black Tree, Case 3 (Double Rotation)



The mirror image of Case 3 (Case 3a) occurs when the red parent is a left child of its parent (the grandparent) and the inserted node is a right child. This could be corrected by first

rotating left around the parent, recoloring the new parent and grandparent, and rotating right around the grandparent.

More than one of these cases can occur as the result of a single insertion. Figure 9.26 shows the insertion of the value 4 into the Red–Black tree of Figure 9.21. Upon return from the insertion, it will be discovered that a red node (5) now has a red child (4), which is a violation of invariant 3. Because 5 has a red sibling (8), this is an example of Case 1. The next step is to recolor node 7 (the parent of 5) and its children nodes 5 and 8, but the problem of a red parent with a red child is shifted up as shown in Figure 9.27 (red nodes 2 and 7).

Looking at Figure 9.27, we see that 7 is red and that its parent, 2, is also red. However, we can't simply change 2's color as we did before because 2's sibling, 14, is black. This problem will require one or two rotations to correct.

In fact, the situation pictured in Figure 9.27 is Case 3a where the child (7) is a right child of its parent (2) and the parent is a left child of the grandparent (11) and requires a double rotation to fix. First we rotate left around the parent (see Figure 9.28), which results in a Case 2 situation: the new parent (7) and its child (2) are both left children. So we recolor the new parent and the grandparent (11) (see Figure 9.29) and then rotate right around the grandparent. Figure 9.30 shows the final Red–Black tree.

FIGURE 9.26
Red–Black Tree after Insertion of 4

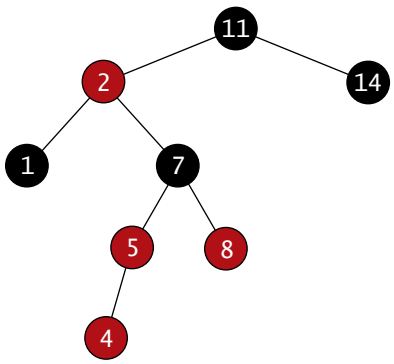


FIGURE 9.27
Moving Black Down and Red Up by Recoloring 7, 5, and 8

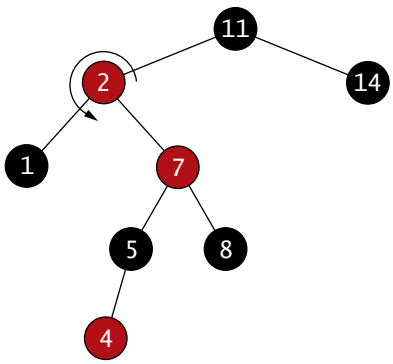


FIGURE 9.28
Node 7 Moved to Outside by Rotating Left around 2

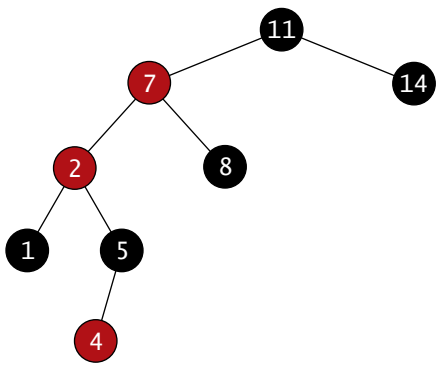


FIGURE 9.29
Recoloring Parent (7) and Grandparent (11)

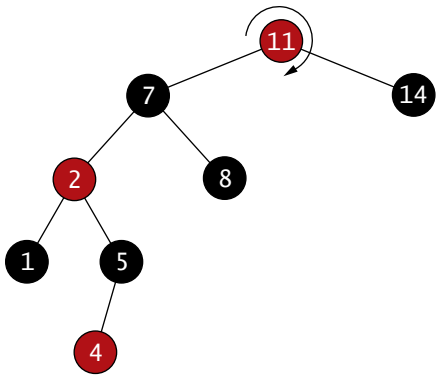
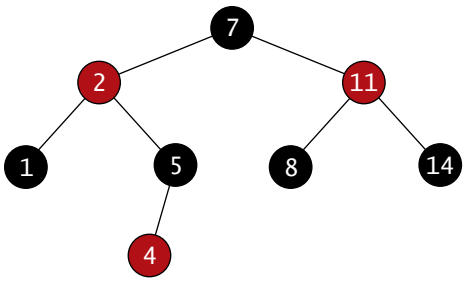
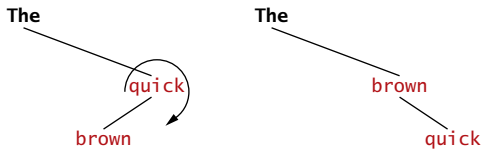


FIGURE 9.30
Final Red–Black Tree after Rotating Right around Node 11

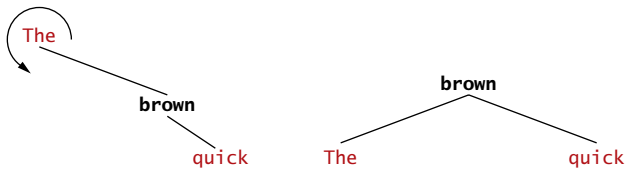


EXAMPLE 9.2 We will now build the Red–Black tree for the sentence "The quick brown fox jumps over the lazy dog".

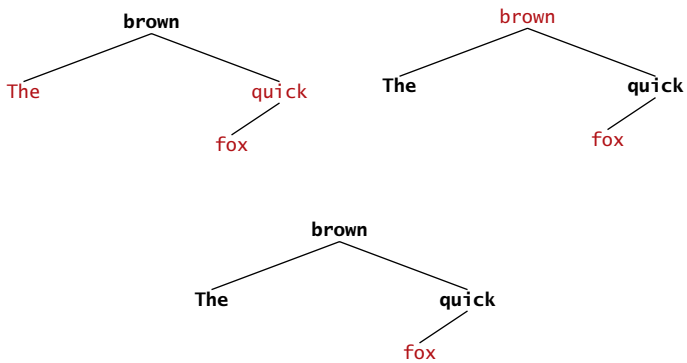
We start by inserting *The*, *quick*, and *brown*. The situation on the left below is an example of a Case 3 insertion. It is resolved by a double rotation. We start by rotating right around *quick*.



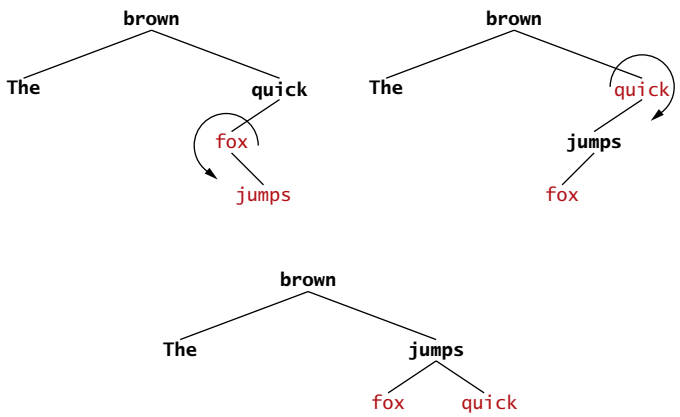
Now, we recolor *The* and *brown* and rotate left around *The*.



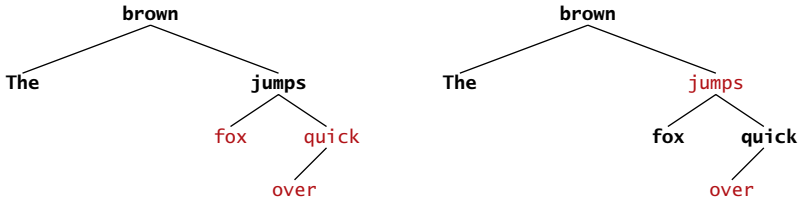
After inserting *fox* (shown on the left below) we have a Case 1 situation (*fox* has a red parent—*quick*—with a red sibling—*The*). We resolve it by recoloring the sibling, parent, and grandparent. However, the grandparent is the root of the overall tree so we change it back to black (shown on the bottom).



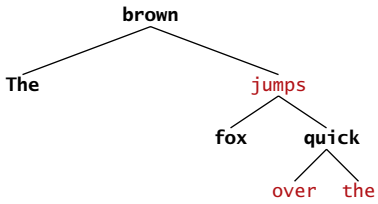
Now we add *jumps*, which gives us a Case 3a situation (*fox* is a left child with a right child of its own). This triggers a double rotation. First rotate left around *fox*, recolor *jumps* and *quick* (diagram on the right), and then rotate right around *quick*.



Next, we insert *over*, which is a Case 1 situation resolved by recoloring *fox*, *quick*, and *jumps*.

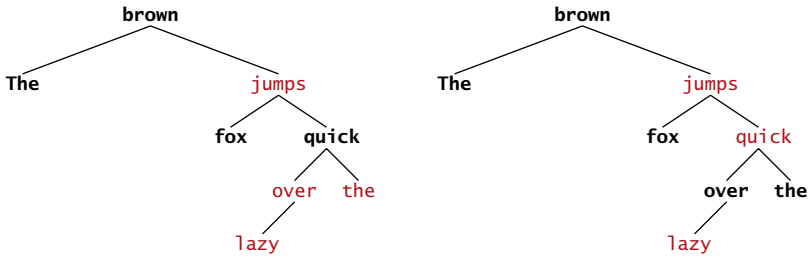


Next, we add *the*. No changes are required because its parent is black.

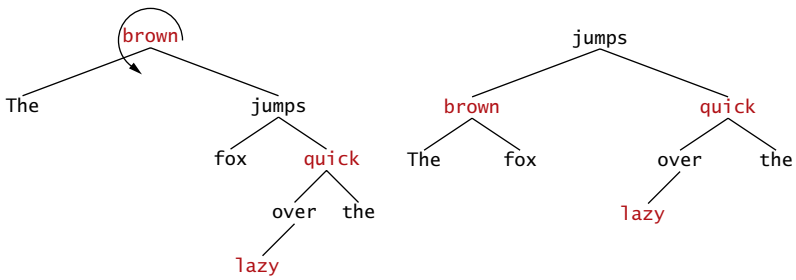


When compared to the corresponding AVL tree, this tree looks out of balance. But the black nodes are in balance (two in each path).

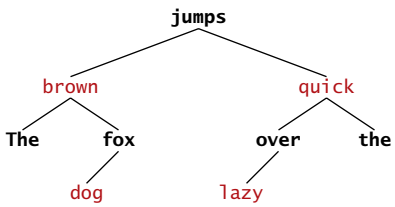
Now comes the interesting part. We insert *lazy*, which gives us a Case 1a situation resolved by recoloring *over*, *the*, and *quick*.



The problem is shifted up one level. Red node *jumps* is a right child with its own right child. The sibling of *jumps*, *The*, is black so this is a Case 2a situation, which is resolved by first recoloring the grandparent *brown* and the parent *jumps* and then rotating left around *brown*.



Finally, we can insert *dog*.



Surprisingly, the result is identical to the AVL tree for the same input, but the intermediate steps were very different.

Implementation of Red–Black Tree Class

We begin by deriving the class `RedBlackTree` from `BinarySearchTreeWithRotate` (see Listing 9.1). Figure 9.31 is a UML class diagram showing the relationship between `RedBlackTree` and `BinarySearchTreeWithRotate`. The `RedBlackTree` class overrides the `add` and `delete` methods. The nested class `RedBlackNode` extends the `BinaryTree.Node` class. Class `RedBlackNode` has the additional data field `isRed` to indicate red nodes (see Listing 9.4).

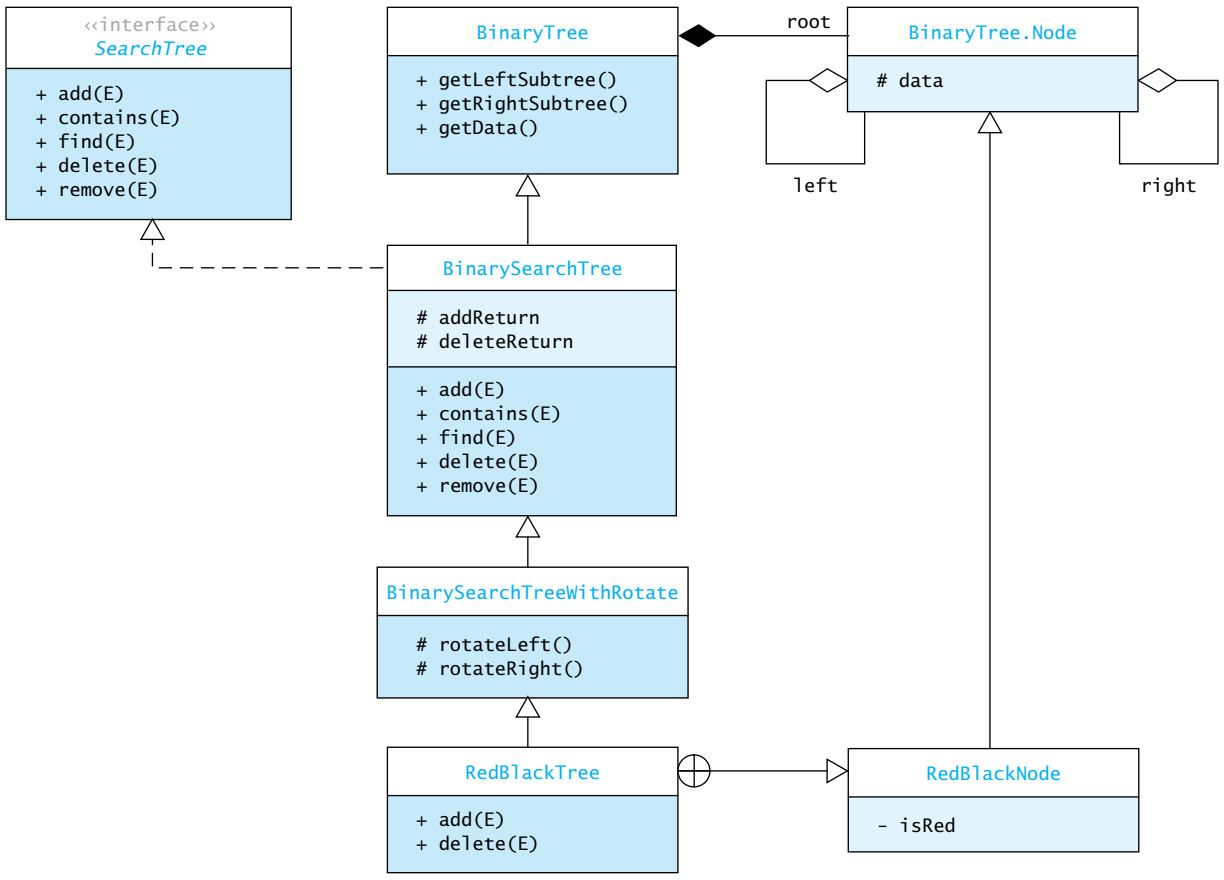
LISTING 9.4
The `RedBlackTree` and `RedBlackNode` Classes

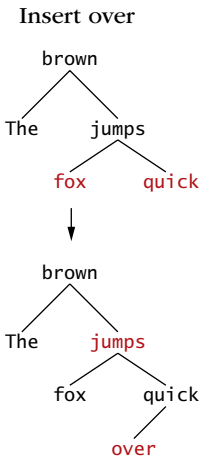
```
/** Class to represent Red-Black tree. */
public class RedBlackTree<E extends Comparable<E>>
             extends BinarySearchTreeWithRotate<E> {

    /** Nested class to represent a Red-Black node. */
    private static class RedBlackNode<E> extends Node<E> {
        // Additional data members
        /** Color indicator. true if red, false if black. */
        private boolean isRed;

        // Constructor
        /** Create a RedBlackNode with the default color of red
            and the given data field.
            @param item The data field
         */
        public RedBlackNode(E item) {
            super(item);
            isRed = true;
        }

        // Methods
        /** Return a string representation of this object.
            The color (red or black) precedes the node's contents.
            @return String representation of this object
         */
        @Override
        public String toString() {
            if (isRed) {
                return "Red : " + super.toString();
            } else {
                return "Black: " + super.toString();
            }
        }
    }
    . . .
}
```

Algorithm for Red–Black Tree Insertion

The foregoing outline of the Red–Black tree insertion algorithm is from the point of view of the node being inserted. It can be, and has been, implemented using a data structure that has a reference to the parent of each node stored in it so that, given a reference to a node, one can access the parent, grandparent, and the parent's sibling (the node's aunt or uncle).

FIGURE 9.31
UML Class Diagram of RedBlackTree





We are going to present a recursive algorithm where the need for fix-ups is detected from the grandparent level. This algorithm has one additional difference from the algorithm as presented in the foregoing examples: whenever a black node with two red children is detected on the way down the tree, it is changed to red and the children are changed to black (e.g., *jumps* and its children in the figure at left). If this change causes a problem, it is fixed on the way back up. This modification simplifies the logic a bit and improves the performance of the algorithm. This algorithm is also based on the algorithm for inserting into a binary search tree that was described in Chapter 6.

Algorithm for Red–Black Tree Insertion

1. **if** the root is **null**
2. Insert a new Red–Black node and color it black.
3. Return **true**.
4. **else if** the item is equal to `root.data`
5. The item is already in the tree; return **false**.
6. **else if** the item is less than `root.data`
7. **if** the left subtree is **null**
8. Insert a new Red–Black node as the left subtree and color it red.

9. Return **true**.
10. **else**
11. **if** both the left child and the right child are red
12. Change the color of the children to black and change local root to red.
13. Recursively insert the item into the left subtree.
14. **if** the left child is now red
15. **if** the left grandchild is now red (grandchild is an "outside node")
16. Change the color of the left child to black and change the local root to red.
17. Rotate the local root right.
18. **else if** the right grandchild is now red (grandchild is an "inside" node)
19. Rotate the left child left.
20. Change the color of the left child to black and change the local root to red.
21. Rotate the local root right.
22. **else**
23. `item` is greater than `root.data`; process is symmetric and is left as an exercise.
24. **if** the local root is the root of the tree
25. Force its color to be black.

Because Java passes the value of a reference, we have to work with a node that is a local root of a Red–Black tree. Thus, in Step 8, we replace the **`null`** reference to the left subtree with the inserted node.

If the left subtree is not **`null`** (Step 10), we recursively apply the algorithm (Step 13). But before we do so, we see whether both children are red. If they are, we change the local root to red and change the children to black (Steps 11 and 12). (If the local root's parent was red, this condition will be detected at that level during the return from the recursion.)

Upon return from the recursion (Step 14), we see whether the local root's left child is now red. If it is, we need to check its children (the local root's grandchildren). If one of them is red, then we have a red parent with a red child, and a rotation is necessary. If the left grandchild is red, a single rotation will solve the problem (Steps 15 through 17). If the right grandchild is red, a double rotation is necessary (Steps 18 through 21). Note that there may be only one grandchild or no grandchildren. However, if there are two grandchildren, they cannot both be red because they would have been changed to black by Steps 11 and 12, as described in the previous paragraph.

The add Starter Method

As with the other binary search trees we have studied, the `add` starter method checks for a **`null`** root and inserts a single new node. Since the root of a Red–Black tree is always black, we set the newly inserted node to black. The cast is necessary because `root` is a data field that was inherited from `BinaryTree` and is therefore of type `Node`.

```
public boolean add(E item) {
    if (root == null) {
        root = new RedBlackNode<>(item);
        ((RedBlackNode<E>) root).isRed = false; // root is black.
        return true;
    }
    . . .
```

Otherwise the recursive `add` method is called. This method takes two parameters: the node that is the local root of the subtree into which the item is to be inserted and the item to be inserted. The return value is the node that is the root of the subtree that now contains the inserted item. The data field `addReturn` is set to **`true`** if the insert method succeeded and to **`false`** if the item is already in the subtree.

The root is replaced by the return value from the recursive `add` method, the color of the root is set to black, and the data field `addReturn` is returned to the caller of the `add` starter method.

```
else {
    root = add((RedBlackNode<E>) root, item);
    ((RedBlackNode<E>) root).isRed = false; // root is always black.
    return addReturn;
}
```

The Recursive add Method

The recursive `add` method begins by comparing the item to be inserted with the data field of the local root. If they are equal, then the item is already in the tree; `addReturn` is set to **`false`** and the `localRoot` is returned (algorithm Step 5).

```
private Node<E> add(RedBlackNode<E> localRoot, E item) {
    if (item.compareTo(localRoot.data) == 0) {
        // item already in the tree.
        addReturn = false;
        return localRoot;
    }
    . . .
```

If it is less, then `localRoot.left` is checked to see whether it is **`null`**. If so, then we insert a new node and return (Steps 7–9).

```
else if (item.compareTo(localRoot.data) < 0) {
    // item < localRoot.data.
    if (localRoot.left == null) {
        // Create new left child.
        localRoot.left = new RedBlackNode<>(item);
        addReturn = true;
        return localRoot;
    }
    . . .
```

Otherwise, check to see whether both children are red. If so, we make them black and change the local root to red. This is done by the method `moveBlackDown`. Then we recursively call the `add` method, using `root.left` as the new local root (Steps 11-13).

```
else {   // Need to search.
    // Check for two red children, swap colors if found.
    moveBlackDown(localRoot);
    // Recursively add on the left.
    localRoot.left = add((RedBlackNode<E>) localRoot.left, item);
    . . .
```

It is upon return from the recursive `add` that things get interesting. Upon return from the recursive call, `localRoot.left` refers to the parent of a Red–Black subtree that may be violating the rule against adjacent red nodes. Therefore, we check the left child to see whether it is red (Step 14).

```
    // See whether the left child is now red
    if (((RedBlackNode<E>) localRoot.left).isRed) {
        // Check whether right grandchildren are red
        . . .
```

If the left child is red, then we need to check its two children. First, we check the left grandchild (Step 15).

```
if (localRoot.left.left != null && ((RedBlackNode<E>)
                                    localRoot.left.left).isRed) {
    // Left-left grandchild is also red.
    . . .
```

If the left–left grandchild is red, we have detected a violation of invariant 3 (no consecutive red children), and we have a left–left case. Thus, we change colors and perform a single rotation, returning the resulting local root to the caller (Steps 16–17).

```
// Single rotation is necessary.
((RedBlackNode<E>) localRoot.left).isRed = false;
localRoot.isRed = true;
return rotateRight(localRoot);
```

If the left grandchild is not red, we then check the right grandchild. If it is red, the process is symmetric to the preceding case, except that a double rotation will be required (Steps 18–21).

```
else if (localRoot.left.right != null && ((RedBlackNode<E>)
                                          localRoot.left.right).isRed) {
    // Left-right grandchild is also red.
    // Double rotation is necessary.
    localRoot.left = rotateLeft(localRoot.left);
    ((RedBlackNode<E>) localRoot.left).isRed = false;
    localRoot.isRed = true;
    return rotateRight(localRoot);
}
```

If upon return from the recursive call the left child is black, the return is immediate, and all of this complicated logic is skipped. Similarly, if neither the left nor the right grandchild is also red, nothing is done.

If the item is greater than `root.data`, the process is symmetric and is left as an exercise (Step 23 and Programming Exercise 1).

Removal from a Red–Black Tree

Removal follows the algorithm for a binary search tree that was described in Chapter 6. Recall that we remove a node only if it is a leaf or if it has only one child. Otherwise, the node that contains the inorder predecessor of the value being removed is the one that is removed. If the node that is removed is red, nothing further must be done because red nodes do not affect a Red–Black tree's balance. If the node to be removed is black and has a red child, then the red child takes its place, and we color it black. However, if we remove a black leaf, then the black height is now out of balance. There are several cases that must be considered. We will describe them in Programming Project 6 at the end of this chapter.

Performance of a Red–Black Tree

It can be shown that the upper limit in the height for a Red–Black tree is $2 \log_2 n + 2$, which is still O(log n). As with the AVL tree, the average performance is significantly better than the worst-case performance. Empirical studies (see Robert Sedgewick, *Algorithms* in C++, 3rd ed. [Addison-Wesley, 1998], p. 570) show that the average cost of a search in a Red–Black tree built from random values is $1.002 \log n$. Thus, both the AVL and Red–Black trees give performance that is close to that of a complete binary search tree.

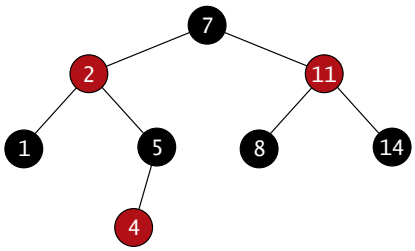
The TreeMap and TreeSet Classes

The Java API has a TreeMap class (part of the package java.util) that implements a Red–Black tree. The TreeMap class implements the SortedMap interface, so it defines methods get, put (a tree insertion), remove, and containsKey, among others. Because a Red–Black tree is used, these are all O(log *n*) operations. There is also a TreeSet class (introduced in Section 7.5) that implements the SortedSet interface. This class is implemented as an adapter of the TreeMap class using a technique like what was described in Chapter 7 to implement the HashSet as an adapter of the HashMap.

EXERCISES FOR SECTION 9.3

SELF-CHECK

For questions 1 through 4, assume each insertion is performed on the tree shown in Figure 9.30 and redrawn below.



1. If you insert 3 where will it be placed?
   a. To the right of 1
   b. To the right of 5
   c. To the left of 4
   d. To the right of 4

2. If you insert 13 where will it be placed?
   a. To the right of 8
   b. To the right of 8
   c. To the left of 14
   d. To the right of 14

3. After the insertion of 3 in question 1, what changes would need to take place to restore the Red–Black tree?
   a. None—new tree is Red–Black Tree
   b. Recolor 4 to black and 5 to red, rotate right around 5
   c. Recolor 5 to red and 2 to black, rotate right around 5
   d. Rotate left around 11

4. After the insertion of 13 in question 2, what changes would occur to restore the Red–Black tree?
   a. None—tree is Red–Black Tree
   b. Recolor 14 to red and 11 to black, rotate left around 11
   c. Recolor 14 to red and 11 to black, rotate right around 11
   d. Rotate right around 14

5. Build the Red–Black tree for the number sequence 20, 10, 15. What would be stored in the root? What would be stored in the children of the root?
6. Build the Red–Black tree for the number sequence 10, 15, 20, 12. What is the first insertion case that occurs that needs to be resolved? After this is resolved, what is the root of the tree and what are its children. What is the next case that needs to be resolved? What is the root of the tree and what are its children?
7. Build the Red–Black tree for the sentence "Now is the time for all good people to come to the aid of others". Would this be the same as the AVL tree (Yes or No)? Give the reason for your answer.
8. Insert the numbers 6, 3, and 0 in the Red–Black tree in Figure 9.21.

PROGRAMMING

1. Program step 23 in the insertion algorithm where the item to insert is greater than `root.data`.

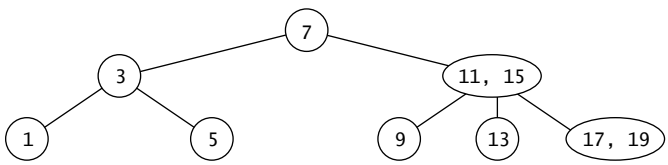
9.4 2–3 Trees

In this section, we begin our discussion of three nonbinary trees. We begin with the *2–3 tree*, named for the number of possible children from each node (either 2 or 3). A 2–3 tree is made up of nodes designated as *2-nodes* and *3-nodes*. A 2-node is the same as a binary search tree node: it consists of a data field and references to two children, one child containing values less than the data field and the other child containing values greater than the data field. A 3-node contains two data fields, ordered so that the first is less than the second, and references to three children: one child containing values less than the first data field, one child containing values between the two data fields, and one child containing values greater than the second data field.

Figure 9.32 shows the general forms of a 2-node (data item is x) and a 3-node (data items are x and y). The children are represented as subtrees. Figure 9.33 shows an example of a 2–3 tree. There are only two 3-nodes in this tree (the right and right–right nodes); the rest are 2-nodes.

FIGURE 9.32
2-Node and 3-Node

FIGURE 9.33
Example of a 2–3 Tree



A 2–3 tree has the additional property that all of the leaves are at the lowest level. This is how the 2–3 tree maintains balance. This will be further explained when we study the insertion and removal algorithms.

Searching a 2–3 Tree

Searching a 2–3 tree is very similar to searching a binary search tree.

```
1.  if the local root is null
2.      Return null; the item is not in the tree.
3.  else if this is a 2-node
4.      if the item is equal to the data1 field
5.          Return the data1 field.
6.      else if the item is less than the data1 field
7.          Recursively search the left subtree.
8.      else
9.          Recursively search the right subtree.
10. else // This is a 3-node
11.     if the item is equal to the data1 field
12.         Return the data1 field.
13.     else if the item is equal to the data2 field
14.         Return the data2 field.
15.     else if the item is less than the data1 field
16.         Recursively search the left subtree.
17.     else if the item is less than the data2 field
18.         Recursively search the middle subtree.
19.     else
20.         Recursively search the right subtree.
```

EXAMPLE 9.3 To search for 13 in Figure 9.33, we would compare 13 with 7 and see that it is greater than 7, so we would search the node that contains 11 and 15. Because 13 is greater than 11 but less than 15, we would next search the middle child, which contains 13: success! The search path is shown in color in Figure 9.34.

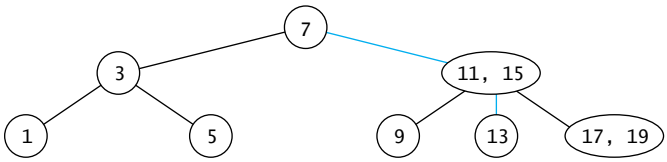


FIGURE 9.34
Searching a 2–3 Tree

Inserting an Item into a 2–3 Tree

A 2–3 tree maintains balance by being built from the bottom up, not the top down. Instead of hanging a new node onto a leaf, we insert the new node into a leaf, as discussed in the following paragraphs. We search for the insertion node using the normal process for a 2–3 tree.

Inserting into a 2-Node Leaf

Figure 9.35 (left) shows a 2–3 tree with three 2-nodes. We want to insert 15. Because the leaf we are inserting into is a 2-node, we can insert 15 directly, creating a new 3-node (Figure 9.35 right).

FIGURE 9.35
Inserting into a Tree with All 2-Nodes

Inserting into a 3-Node Leaf with a 2-Node Parent

If we want to insert a number larger than 7 (say, 17), that number will be virtually inserted into the 3-node at the bottom right of the tree, giving the virtual node in gray in Figure 9.36. Because a node can't store three values, the middle value will propagate up to the 2-node parent, and the virtual node will be split into two new 2-nodes containing the smallest and largest values. Because the parent is a 2-node, it will be changed to a 3-node, and it will reference the three 2-nodes, as shown in Figure 9.37.

FIGURE 9.36
A Virtual Insertion

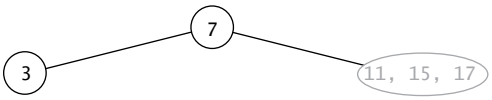
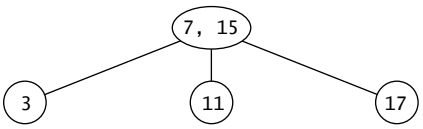
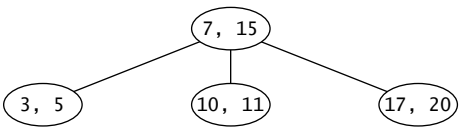


FIGURE 9.37
Result of Propagating 15 to 2-Node Parent



Let's now insert the numbers 5, 10, and 20. Each of these would go into one of the leaf nodes (all 2-nodes), changing them to 3-nodes, as shown in Figure 9.38.

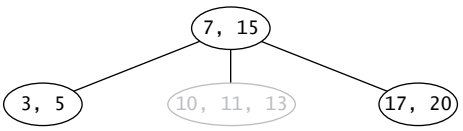
FIGURE 9.38
Inserting 5, 10, and 20



Inserting into a 3-Node Leaf with a 3-Node Parent

In the tree in Figure 9.39, all the leaf nodes are full, so if we insert any other number, one of the leaf nodes will need to be virtually split, and its middle value will propagate to the parent. Because the parent is already a 3-node, it will also need to be split.

FIGURE 9.39
Virtually Inserting 13



For example, if we were to insert 13, it would be virtually inserted into the leaf node with values 10 and 11 (see Figure 9.39). This would result in two new 2-nodes with values 10 and 13, and 11 would propagate up to be virtually inserted in the 3-node at the root (see Figure 9.40). Because the root is full, it would split into two new 2-nodes with values 7 and 15,

and 11 would propagate up to be inserted in a new root node. The net effect is an increase in the overall height of the tree, as shown in Figure 9.41.

FIGURE 9.40
Virtually Inserting 11

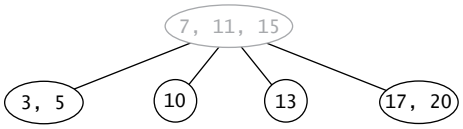
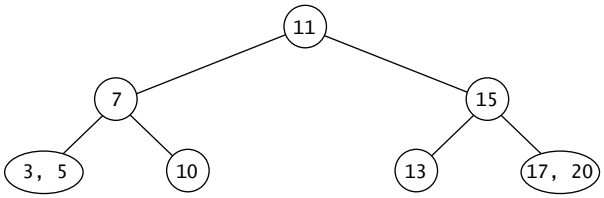


FIGURE 9.41
Result of Making 11 the New Root

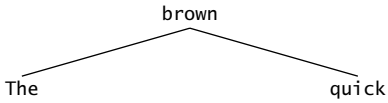


We summarize these observations in the following insertion algorithm.

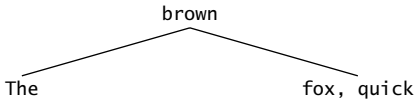
Algorithm for Insertion into a 2–3 Tree

1. **if** the root is `null`
2. Create a new 2-node that contains the new item.
3. **else if** the item is in the local root
4. Return **false**.
5. **else if** the local root is a leaf
6. **if** the local root is a 2-node
7. Expand the 2-node to a 3-node and insert the item.
8. **else**
9. Split the 3-node (creating two 2-nodes) and pass the new parent and right child back up the recursion chain.
10. **else** *// search for a leaf in a subtree of local root*
11. **if** the item is less than the smaller item in the local root
12. Recursively insert into the left child.
13. **else if** the local root is a 2-node
14. Recursively insert into the right child.
15. **else if** the item is less than the larger item in the local root
16. Recursively insert into the middle child.
17. **else**
18. Recursively insert into the right child.
19. **if** a new parent was passed up from the previous level of recursion
20. **if** the new parent will be the tree root
21. Create a 2-node whose data item is the passed-up parent, left child is the old root, and right child is the passed-up child. This 2-node becomes the new root.
22. **else**
23. Recursively insert the new parent at the local root.
24. Return **true**.

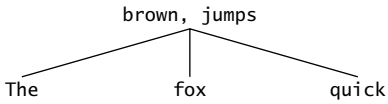
EXAMPLE 9.4 We will create a 2–3 tree using "The quick brown fox jumps over the lazy dog." The initial root contains *The*, *quick*. If we insert *brown*, we will split the root. Because *brown* is between *The* and *quick*, it gets passed up and will become the new root (Algorithm step 9).



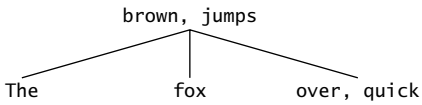
We now insert *fox* as the left neighbor of *quick*, creating a new 3-node (steps 14, 7).



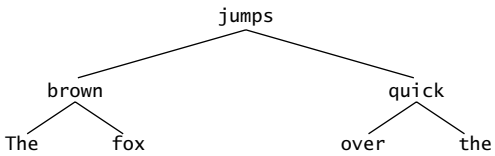
Next, *jumps* is virtually inserted between *fox* and *quick*, thus splitting this 3-node, and *jumps* gets passed up and inserted next to *brown* (steps 14, 9, 23).



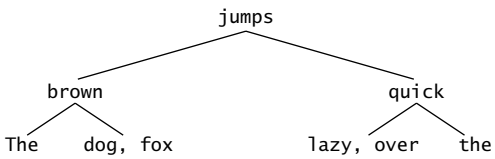
Then *over* is inserted next to *quick* (steps 14, 7).



Now we insert *the*. It will be virtually inserted to the right of *over*, *quick*, splitting that node, and *quick* will be passed up. It will be virtually inserted to the right of *brown*, *jumps*, splitting that node as well, causing *jumps* to be passed up to the new root (steps 14, 9, 23, 9).



Finally, *lazy* and *dog* are inserted next to *over* (steps 14, 12, 7) and *fox* (steps 12, 14, 7), respectively.



Analysis of 2–3 Trees and Comparison with Balanced Binary Trees

The 2–3 tree resulting from the preceding example is a balanced tree of height 3 that requires fewer complicated manipulations. There were no rotations, as were needed to build the AVL and Red–Black trees, which were both of height 4. The number of items that a 2–3 tree of height h can hold is between $2^h - 1$ (all 2-nodes) and $3^h - 1$ (all 3-nodes). Therefore, the

height of a 2–3 tree is between $\log_3 n$ and $\log_2 n$. Thus, the search time is O(log n), since logarithms are all related by a constant factor, and constant factors are ignored in big-O notation.

Removal from a 2–3 Tree

Removing an item from a 2–3 tree is somewhat the reverse of the insertion process. To remove an item, we must first search for it. If the item to be removed is in a leaf, we simply delete it. However, if the item to be removed is not in a leaf, we remove it by swapping it with its inorder predecessor in a leaf node and deleting it from the leaf node. If removing a node from a leaf causes the leaf to become empty, items from the sibling and parent can be redistributed into that leaf, or the leaf can be merged with its parent and sibling nodes. In the latter case, the height of the tree may decrease. We illustrate these cases next.

If we remove item 13 from the tree shown in Figure 9.42, its node will become empty, and item 15 in the parent node would have no left child. We can merge 15 and its right child to form the virtual leaf node {15, 17, 19}. Item 17 moves up to the parent node; item 15 is the new left child of 17 (see Figure 9.43).

We next remove 11 from the 2–3 tree. Because this is not a leaf, we replace it with its predecessor, 9, as shown in Figure 9.44. We now have the case where the left leaf node of 9 has become empty. So, we merge 9 into its right leaf node as shown in Figure 9.45.

FIGURE 9.42
Removing 13 from a 2–3 Tree

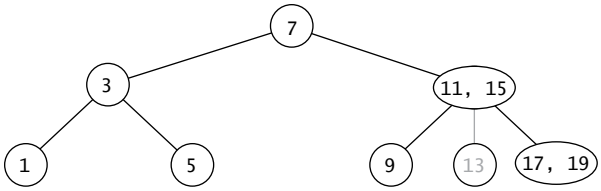


FIGURE 9.43
2–3 Tree after Redistribution of Nodes Resulting from Removal

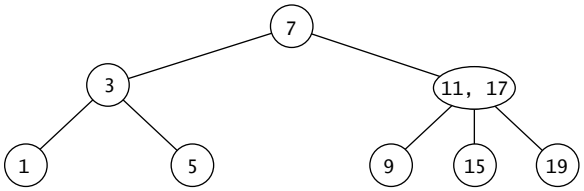
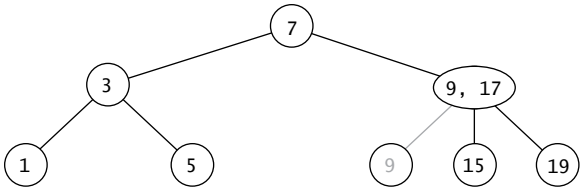


FIGURE 9.44
Removing 11 from the 2–3 Tree (Step 1)



Finally, let's consider the case in which we remove the value 1 from Figure 9.45. First, 1's parent (3) and its right sibling (5) are merged to form a 3-node, as shown in Figure 9.46. This has the effect of deleting 3 from the next higher level. Therefore, the process repeats, and 3's parent (7) and 7's right child (17) are merged, as shown in Figure 9.47. The merged node becomes the root.

The 2–3 tree served as an inspiration for the more general B-tree and 2–3–4 tree. Rather than show an implementation of the 2–3 tree, which has some rather messy complications, we will describe and implement the more general B-tree in the next section.

FIGURE 9.45
2–3 Tree after Removing 11

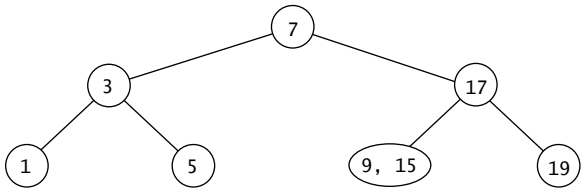


FIGURE 9.46
After Removing 1 (Intermediate Step)

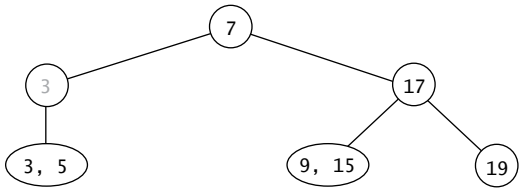
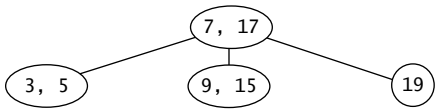


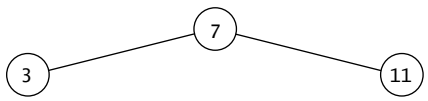
FIGURE 9.47
After Removing 1 (Final Form)



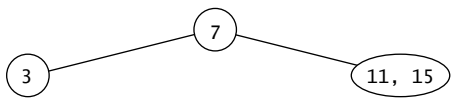
EXERCISES FOR SECTION 9.4

SELF-CHECK

1. Show the tree after inserting each of the following values one at a time: 2, 6, 10.



2. Show the tree after inserting each of the following one at a time: 8, 12.



3. Build the 2–3 tree for the sentence "Now is the time for all good people to come to the aid of others". What would be the second virtual node before splitting? The third virtual node? The fourth virtual node?

9.5 B-Trees and 2–3–4 Trees

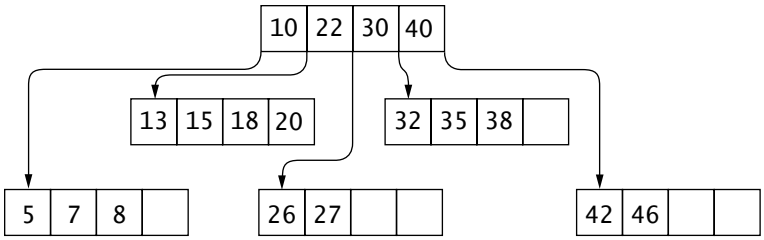
The 2–3 tree was the inspiration for the more general B-tree, which allows up to *n* children per node, where *n* may be a very large number. The B-tree was designed for building indexes to very large databases stored on a hard disk. The 2–3–4 tree is a special case of a B-tree.

B-Trees

In the 2–3 tree, a 2-node has two children and a 3-node has three children. In the B-tree, the maximum number of children is the order of the B-tree, which we will represent by the variable `order`. Other than the root, each node has between `order`/2 and `order` children. The number of data items in a node is 1 less than the number of children. The data items in each node are in increasing sequence.

Figure 9.48 shows an example of a B-tree with `order` equal to 5. The first link from a node connects to a subtree with values smaller than the parent's smallest value (10 for the root node); the last link from a node connects to a subtree with values greater than the parent's largest value (40 for the root node); the other links are to subtrees with values between each pair of consecutive values in the parent node (e.g., > 10 and < 22 or > 22 and < 30).

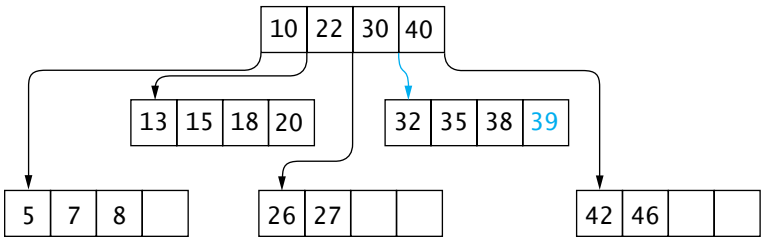
FIGURE 9.48
Example of a B-Tree



B-trees were developed to store indexes to databases on disk storage. Disk storage is broken into blocks, and the nodes of a B-tree are sized to fit in a block, so each disk access to the index retrieves exactly one B-tree node. The time to retrieve a block is large compared to the time required to process it in memory, so by making the tree nodes as large as possible, we reduce the number of disk accesses required to find an item in the index. Assuming a block can store a node for a B-tree of order 200, each node would store at least 100 items. This would enable 100^4 or 100 million items to be accessed in a B-tree of height 4.

The insertion process for a B-tree is similar to that of a 2–3 tree, and each insertion is into a leaf. For Figure 9.48, a number less than 10 would be inserted into the leftmost leaf; a number greater than 40 would be inserted into the rightmost leaf; and numbers between 11 and 39 would be inserted into one of the interior leaves. A simple case is insertion of the number 39 into the fourth child of the root node (shown in color in Figure 9.49).

FIGURE 9.49
B-Tree after Inserting 39



However, if the leaf being inserted into is full, it is split into two nodes, each containing approximately half the items, and the middle item is passed up to the split node's parent. If the parent is full, it is split and its middle item is passed up to its parent, and so on. If a node

being split is the root of the B-tree, a new root node is created, thereby increasing the height of the B-tree. The children of the new root will be the two nodes that resulted from splitting the old root.

Figure 9.50 shows the B-tree after inserting 17. Because 17 is between 10 and 22, it should be inserted into the second leaf node {13, 15, 18, 20}, with 17 as the new third or middle item {13, 15, 17, 18, 20}. However, the original leaf node was full, so it is split, and its new middle item, 17, is passed up to the root node {10, 17, 22, 30, 40}. Because the original root node was full, it is also split and its new middle item, 22, is passed up as the first item in a new root node, thereby increasing the height of the tree. It is interesting that the B-tree grows by adding nodes to the top of the tree instead of adding them at the bottom. The nodes that changed are in color.

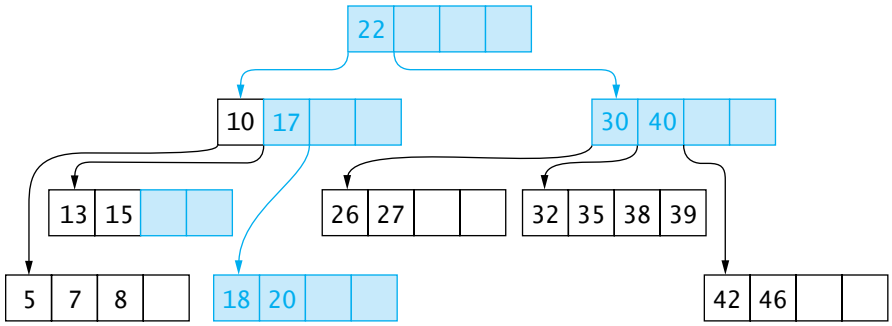


FIGURE 9.50
B-Tree after Inserting 17

Implementing the B-Tree

The `Node` class holds up to `order-1` data items and `order` references to children. The array `data` stores the data, and the array `children` stores the references to the children. The number of data items currently stored is indicated by `size`.

```
/** A Node represents a node in a B-tree. */
private static class Node<E> {
    // Data Fields

    /** The number of data items in this node */
    private int size = 0;
    /** The information */
    private E[] data;
    /** The links to the children. child[i] refers to the subtree of
        children < data[i] for i < size and to the subtree
        of children > data[size-1] for i == size
     */
    private Node<E>[] child;

    /** Create an empty node of size order
        @param order The size of a node
     */
    @SuppressWarnings("unchecked")
    public Node(int order) {
        data = (E[]) new Comparable[order-1];
        child = (Node<E>[]) new Node[order];
        size = 0;
    }
}
```

The declaration for the B-tree class begins as follows:

```
/** A B-tree is a search tree
    in which each node contains n data items where n is between
    (order-1)/2 and order-1. (For the root, n may be between 1
    and order-1.) Each node not a leaf has n+1 children. The tree is
    always balanced in that all leaves are on the same level, i.e.,
    the length of the path from the root to a leaf is constant.
    @author Koffman and Wolfgang
 */
public class BTree<E extends Comparable<E>>
    // Nested class
    /** A Node represents a node in a B-tree. */
    private static class Node<E> {
    . . .
    }

    /** The root node. */
    private Node<E> root = null;
    /** The maximum number of children of a node */
    private int order;

    /** Construct a B-tree with node size order
        @param order the size of a node
     */
    public BTree(int order) {
        this.order = order;
        root = null;
    }
```

The `insert` method is very similar to that for the 2–3 tree. Method `insert` searches the current node for the item. If the item is found, insertion is not possible, so it returns **`false`**. If the item is not found in the current node, it follows the appropriate link until it reaches a leaf and then inserts the item into that leaf. If the leaf is full, it is split. In the 2–3 tree, we described this process as a virtual insertion into the full leaf, and then the middle data value is used as the parent of the split-off node. This parent value is then inserted into the parent node during the return process of the recursion.

Algorithm for Insertion

1. **`if`** the root is **`null`**
2. Create a new `Node` that contains the inserted item.
3. **`else`** search the local root for the item
4. **`if`** the item is in the local root
5. return **`false`**
6. **`else`**
7. **`if`** the local root is a leaf
8. **`if`** the local root is not full
9. insert the new item
10. return **`null`** as the `newChild` and **`true`** to indicate successful insertion
11. **`else`**
12. split the local root
13. return the `newParent` and a `newChild` and **`true`** to indicate successful insertion
14. **`else`**

```
15.          recursively call the insert method
16.          if the returned newChild is not null
17.              if the local root is not full
18.                  insert the newParent and newChild into the local root
19.                  return null as the newChild and true to indicate successful
                     insertion
20.              else
21.                  split the local root
22.                  return newParent and newChild and true to indicate success
23.          else
24.              return the success/fail indicator for the insertion
```

In this algorithm, we showed multiple return values. There is the `boolean` return value that indicates success or failure of the insertion. There is the `newParent` of the split-off node, and there is the split-off node, which we call the `newChild`. We implement this by using the return value from the `insert` method as the success/fail indicator, and `newParent` and `newChild` are private data fields in the `BTree` class. If there is no new child, `newChild` is set to **`null`**.

Code for the `insert` Method

The code for the `insert` method is shown in Listing 9.5. We use a binary search to locate the item in the local root. The `binarySearch` method returns the index of the item if it is present or the index of the position where the item should be inserted.

We need to test to see whether the local root is full. If it is not full, we can insert the item into the local root; otherwise, we need to split the local root. In either case, we return **`true`**.

```
if (root.size < order-1) {
    insertIntoNode(root, index, item, null);
    newChild = null;
} else {
    splitNode(root, index, item, null);
}
return true;
```

If the local root is not a leaf, then we recursively call the `insert` method using `local.child[index]` as the root argument. Upon return from the recursion, we test the value of `newChild`. If it is `null`, we return the result of the insertion. If it is not **`null`** and the local root is not full, we insert the `newParent` and `newChild` into the local root; otherwise, we split the local root.

```
boolean result = insert(root.child[index], item);
if (newChild != null) {
    if (root.size < order-1) {
        insertIntoNode(root, index, newParent, newChild);
        newChild = null;
    } else {
        splitNode(root, index, newParent, newChild);
    }
}
return result;
```

The `insertIntoNode` and `splitNode` methods are described next.

LISTING 9.5
The `insert` Method Function from `BTree.java`

```
/** Recursively insert an item into the B-tree. Inserted
    item must implement the Comparable interface
    @param root The local root
    @param item The item to be inserted
    @return true if the item was inserted,
            false if the item is already in the tree
 */
private boolean insert(Node<E> root, E item) {
    int index = binarySearch(item, root.data, 0, root.size);
    if (index != root.size && item.compareTo(root.data[index]) == 0) {
        return false;
    }
    if (root.child[index] == null) {
        if (root.size < order-1) {
            insertIntoNode(root, index, item, null);
            newChild = null;
        } else {
            splitNode(root, index, item, null);
        }
        return true;
    } else {
        boolean result = insert(root.child[index], item);
        if (newChild != null) {
            if (root.size < order-1) {
                insertIntoNode(root, index, newParent, newChild);
                newChild = null;
            } else {
                splitNode(root, index, newParent, newChild);
            }
        }
        return result;
    }
}
```

The `insertIntoNode` Method

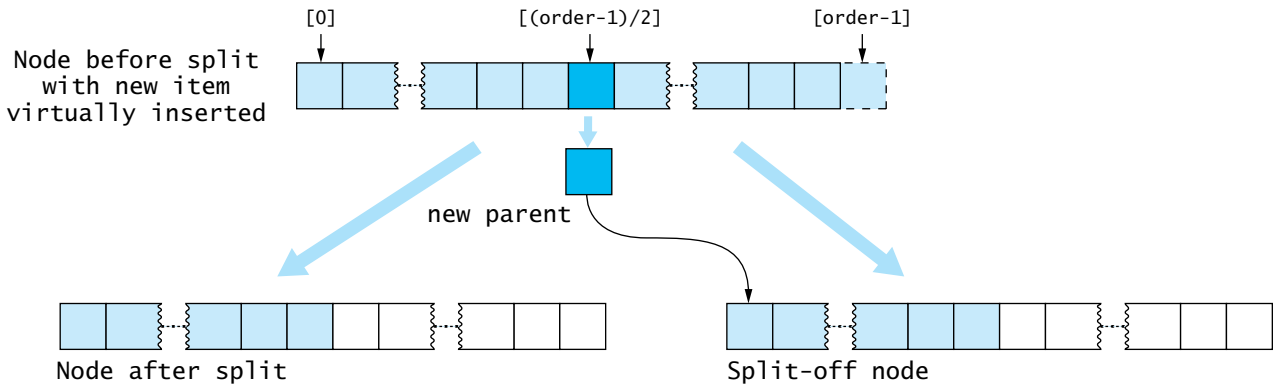
The `insertIntoNode` method shifts the data and child values to the right and inserts the new value and child at the indicated index.

```
/** Method to insert a new value into a node
    pre: node.data[index-1] < item < node.data[index];
    post: node.data[index] == item and old values are moved right one position
    @param node The node to insert the value into
    @param index The index where the inserted item is to be placed
    @param item The value to be inserted
    @param child The right child of the value to be inserted
 */
private void insertIntoNode(Node<E> node, int index, E obj, Node<E> child) {
    for (int i = node.size; i > index; i--){
        node.data[i] = node.data[i - 1];
        node.child[i + 1] = node.child[i];
    }
    node.data[index] = obj;
    node.child[index + 1] = child;
    node.size++;
}
```

The splitNode Method

The splitNode method will perform the virtual insertion of the new item (and its child) into the node and split it so that half of the items remain in the node being split and the rest are moved to the split-off node. The middle value becomes the parent of the split-off node. The middle value is passed up to the parent node, which still links to the node that was split. This is illustrated in Figure 9.51.

FIGURE 9.51
Splitting the Node



The code for the splitNode method is shown in Listing 9.6. Since we cannot insert the new item into the node before it is split, we need to do the split first in such a way that space is available in either the original node or the split-off node for the new item. After the split, we keep half of the items in the original node and move the other half to the split-off node. The number of items to keep is order-1; thus half of them is (order-1)/2, and the number of items to move is (order-1) - (order-1)/2. The reason that this is not simply (order-1)/2 is that order-1 may be an odd number. Thus, we move (order-1) - (order-1)/2 items, unless the new item is to be inserted into the split-off node, in which case we move one fewer item. The number of items to be moved is computed using the following statements:

```
// Determine number of items to move
int numToMove = (order-1) - ((order-1) / 2);
// If insertion point is in the right half, move one less item
if (index > (order-1) / 2) { numToMove--; }
```

The System.arraycopy method is then used to move the data and the corresponding children.

```
// Move items and their children
System.arraycopy(node.data, order - numToMove - 1,
                 newChild.data, 0, numToMove);
System.arraycopy(node.child, order - numToMove,
                 newChild.child, 1, numToMove);
node.size = order - numToMove - 1;
newChild.size = numToMove;
```

Now we are ready to insert the new item and set the newChild.child[0] reference. There are three cases: the item is to be inserted as the middle item, the item is to be inserted into the original node, and the item is to be inserted into the newChild. If the item is to be inserted into the middle, then it becomes the newParent, and its child becomes newChild.child[0].

```
if (index == ((order-1) / 2)) { // Insert as middle item
    newParent = item;
    newChild.child[0] = child;
}
```

Otherwise we can use the `insertIntoNode` method to insert the item into either the original node or the `newChild` node.

```
if (index < ((order-1) / 2)) { // Insert into the left half
    insertIntoNode(node, index, item, child);
} else {
    insertIntoNode(newChild, index - ((order-1) / 2) - 1, item, child);
}
```

In either case, after the insert, the last item in the original node becomes the `newParent` and its child becomes `newChild.child[0]`

```
// The rightmost item of the node is the new parent
newParent = node.data[node.size - 1];
// Its child is the left child of the split-off node
newChild.child[0] = node.child[node.size];
node.size--;
```

LISTING 9.6
Method `splitNode` from `BTree.java`

```
private void splitNode(Node<E> node, int index, E item, Node<E> child) {
    // Create new child
    newChild = new Node<E>(order);
    // Determine number of items to move
    int numToMove = (order-1) - ((order-1) / 2);
    // If insertion point is in the right half, move one less item
    if (index > (order-1) / 2) { numToMove--; }

    // Move items and their children
    System.arraycopy(node.data, order - numToMove - 1,
                     newChild.data, 0, numToMove);
    System.arraycopy(node.child, order - numToMove,
                     newChild.child, 1, numToMove);
    node.size = order - numToMove - 1;
    newChild.size = numToMove;

    // Insert new item
    if (index == ((order-1) / 2)) { // Insert as middle item
        newParent = item;
        newChild.child[0] = child;
    } else {
        if (index < ((order-1) / 2)) { // Insert into the left half
            insertIntoNode(node, index, item, child);
        } else {
            insertIntoNode(newChild, index - ((order-1) / 2) - 1, item, child);
        }
        // The rightmost item of the node is the new parent
        newParent = node.data[node.size - 1];
        // Its child is the left child of the split-off node
        newChild.child[0] = node.child[node.size]; node.size--;
    }

    // Remove items and references to moved items
    for (int i = node.size; i < node.data.length; i++) {
        node.data[i] = null;
        node.child[i + 1] = null;
    }
}
```

Removal from a B-Tree

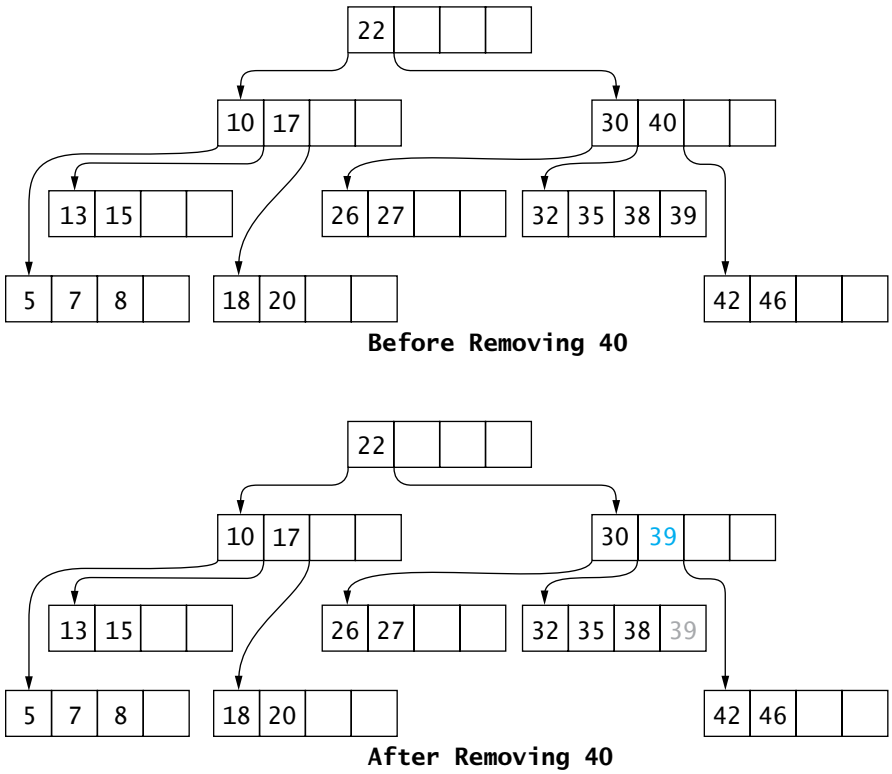
Removing an item from a B-tree is a generalization of removing an item from a 2–3 tree. The simpler case occurs when the item to be removed is in a leaf; in this case, it is deleted from the leaf. However, if the item to be removed is in an interior node, it can't be deleted simply because that would damage the B-tree. To retain the B-tree property, the item must be replaced by its inorder predecessor (or its inorder successor), which is in a leaf. As an example, Figure 9.52 shows the tree that would be formed when 40 is removed from the tree in Figure 9.50 and is replaced with its inorder predecessor (39).

In the event the removal of an item from a leaf results in a leaf node that is less than half full, this would violate a property of the B-tree (only the root node can be less than half full). To correct this, items from a sibling node and parent are redistributed into that leaf. However, if the sibling is itself exactly half full, the leaf, its parent item, and sibling are merged into a single node, deleting the parent item from the parent node. If the parent node is now half full, the process of node redistribution or merging is repeated during the recursive return process. The merging process may reduce the height of the B-tree.

We illustrate this process by deleting item 18 from the bottom B-tree in Figure 9.52. The leaf node that contained 18 would have only one item (20), so we merge it with its parent and left sibling into a new full node {13, 15, 17, 20} as shown in Figure 9.53.

The problem is that the parent of {13, 15, 17, 20} has only one item (10), so it is merged with its parent and right sibling to form a new root node {10, 22, 30, 40} as shown in Figure 9.54. Note that the height of the resulting B-tree has also been reduced by 1.

FIGURE 9.52
Removing 40 from B-Tree of Figure 9.50



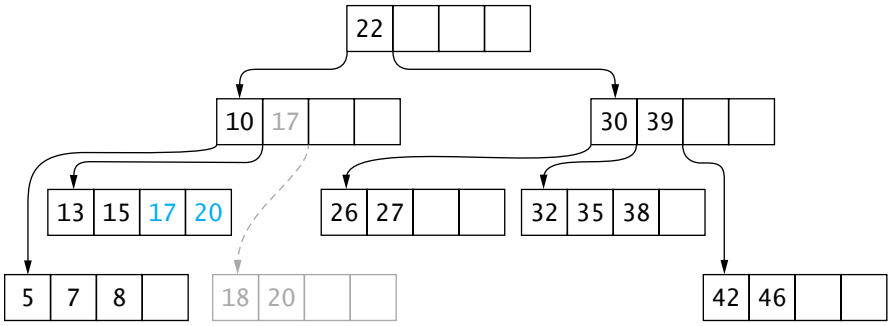


FIGURE 9.53
Step 1 of Removing 18 from Bottom B-Tree of Figure 9.52

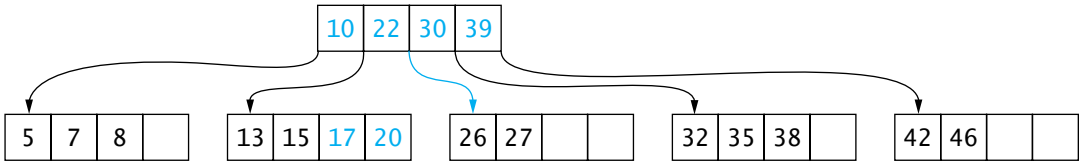


FIGURE 9.54
Step 2 of Removing 18 from B-Tree of Figure 9.52

B+ Trees

We stated earlier that B-trees were developed and are still used to create indexes for databases. The `Node` is stored on a disk block, and the pointers are pointers to disk blocks instead of being memory addresses. The `E` is a key-value pair, where the value is also a pointer to a disk block. Since in the leaf nodes all of the child pointers are **`null`**, there is a significant amount of wasted space. A modification to the B-tree, known as the B+ tree, was developed to reduce this wasted space. In the B+ tree, the leaves contain the keys and pointers to the corresponding values. The internal nodes only contain keys and pointers to children.

Like in the `B`-tree there are *order* pointers to children and *order*-1 values. The `child[0]` pointer points to nodes less than `data[0]`. The `child[i]` pointer points to nodes greater than or equal to `data[i-1]`, with the value of `data[i-1]` repeated in the child node or pointed to by the child's `child[0]` pointer. All of the keys are repeated in the leaves.

An example of a B+ tree is shown in Figure 9.55. Observe how the pointers at the root (22) point to items less than 22 and to items greater than 22. At the next level, the `child[0]` pointer of the node containing 8 and 15 points to the leaf node containing 5 and 7 (numbers < 8).

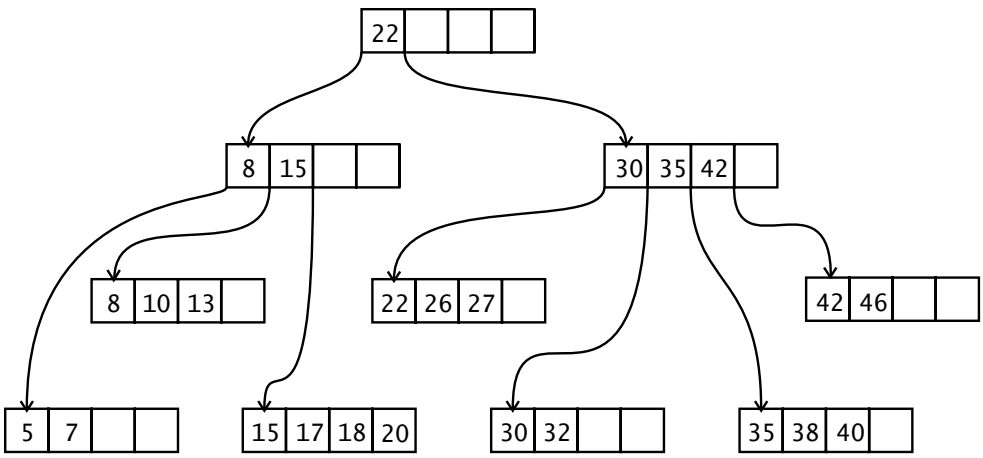


FIGURE 9.55
Example of a B+ Tree

The pointer from 8 points to the leaf containing 8, 10, and 13 (numbers >= 8 and < 15). The pointer from 15 points to the leaf containing 15, 17, 18, and 20 (numbers >= 15).

2–3–4 Trees

2–3–4 trees are a special case of the B-tree where order is fixed at 4. We refer to such a node as a 4-node (see Figure 9.56). This is a node with three data items and four children. Figure 9.57 shows an example of a 2–3–4 tree.

FIGURE 9.56
2-, 3-, and 4-Nodes

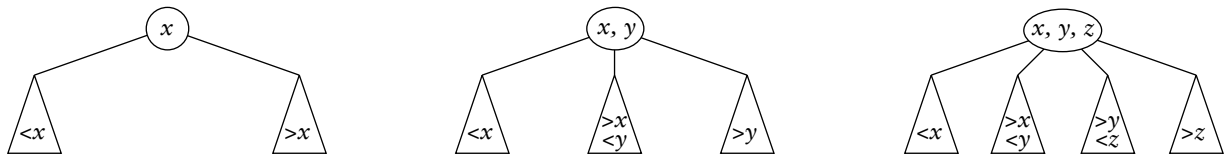
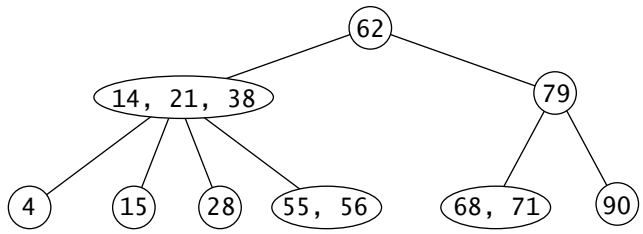


FIGURE 9.57
Example of a 2–3–4 Tree



Fixing the capacity of a node at three data items simplifies the insertion logic. We can search for a leaf in the same way as for a 2–3 tree or a B-tree. If a 4-node is encountered at any point, we will split it, as discussed subsequently. Therefore, when we reach a leaf, we are guaranteed that there will be room to insert the item.

For the 2–3–4 tree shown in Figure 9.57, a number larger than 62 would be inserted in a leaf node in the right subtree. A number between 63 and 78, inclusive, would be inserted in the 3-node (68, 71), making it a 4-node. A number larger than 79 would be inserted in the 2-node (90), making it a 3-node.

When inserting a number smaller than 62 (say, 25), we would encounter the 4-node (14, 21, 38). We would immediately split it into two 2-nodes and insert the middle value (21) into the parent (62) as shown in Figure 9.58. Doing this guarantees that there will be room to insert the new item. We perform the split from the parent level and immediately insert the middle item from the split child into the parent node. Because we are guaranteed that the parent is not a 4-node, we will always have room to do this. We do not need to propagate a child or its parent back up the recursion chain. Consequently, the recursion becomes tail recursion.

Now we can insert 25 as the left neighbor of 28 as shown in Figure 9.59.

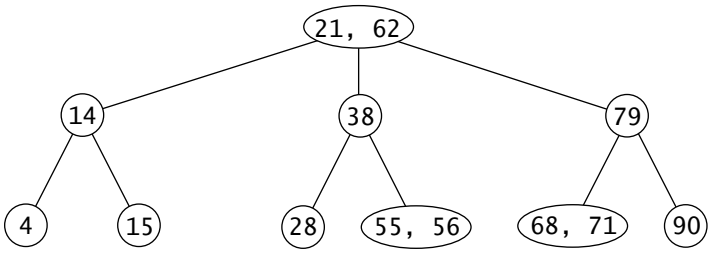


FIGURE 9.58
Result of Splitting a 4-Node

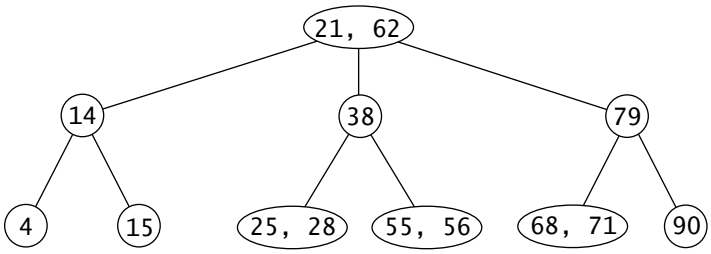


FIGURE 9.59
2–3–4 Tree after Inserting 25

In this example, splitting the 4-node was not necessary. We could have merely inserted 25 as the left neighbor of 28. However, if the leaf being inserted into was a 4-node, we would have had to split it and propagate the middle item back up the recursion chain, just as we did for the 2–3 tree. Always splitting a 4-node when it is encountered results in prematurely splitting some nodes, but it simplifies the algorithm and has minimal impact on the overall performance.

Relating 2–3–4 Trees to Red–Black Trees

A Red–Black tree is a binary-tree equivalent of a 2–3–4 tree. A 2-node is a black node (see Figure 9.60). A 4-node is a black node with two red children (see Figure 9.61). A 3-node can be represented as either a black node with a left red child or a black node with a right red child (see Figure 9.62).

FIGURE 9.60
A 2-Node as a Black Node in a Red–Black Tree

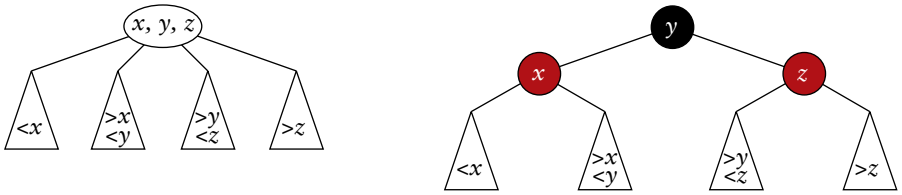


FIGURE 9.61
A 4-Node as a Black Node with Two Red Children in a Red–Black Tree

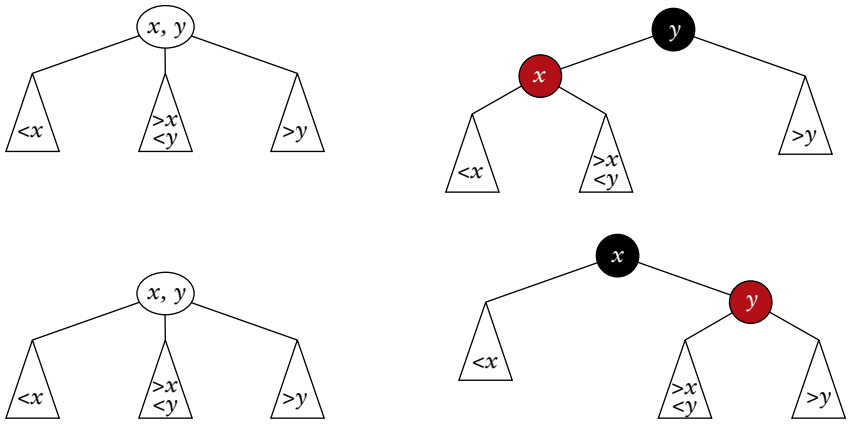


FIGURE 9.62
A 3-Node as a Black Node with One Red Child in a Red–Black Tree

Suppose we want to insert a value z that is greater than y into the 3-node shown at the top of Figure 9.62 (tree with black root y). Node z would become the red right child of black node y, and the subtree with label $> y$ would be split into two parts, giving the Red–Black tree and the 4-node shown in Figure 9.61.

Suppose, on the other hand, we want to insert a value z that is between x and y into the 3-node shown at the bottom of Figure 9.62 (tree with black root x). Node z would become the red left child of red node y (see the left diagram in Figure 9.63), and a double rotation would be required. First rotate right around y (the middle diagram) and then rotate left around x (the right diagram). This corresponds to the situation shown in Figure 9.64 (a 4-node with x, z, y).

FIGURE 9.63
Inserting into the Middle of a 3-Node (Red–Black Tree Equivalent)

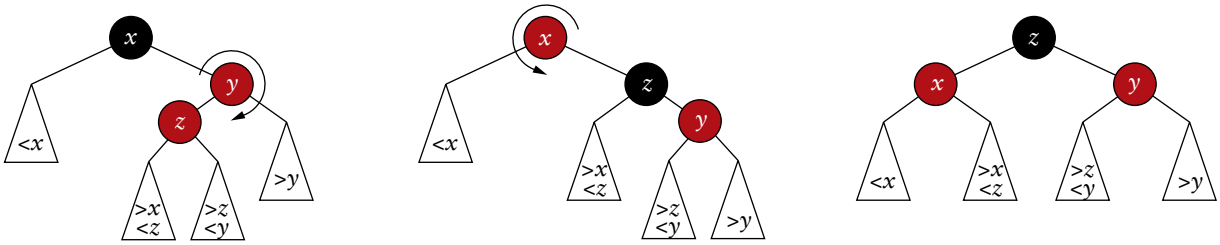
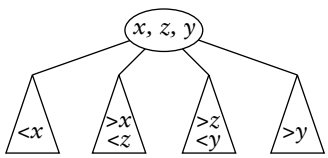


FIGURE 9.64
Inserting into the Middle of a 3-Node (2–3–4 Tree)

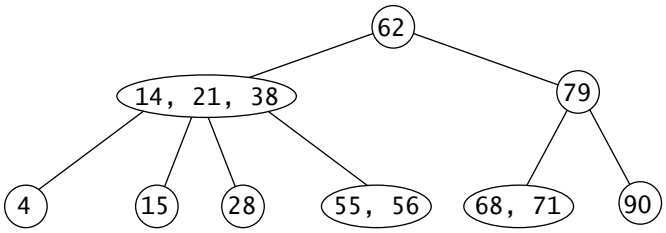


EXERCISES FOR SECTION 9.5

SELF-CHECK

1. Draw a B-tree with order 5 that stores the sequence of integers: 20, 30, 8, 10, 15, 18, 44, 26, 28, 23, 25, 43, 55, 36, 44, 39.
2. Remove items 30, 26, 15, and 17 from the B-tree in Figure 9.50.

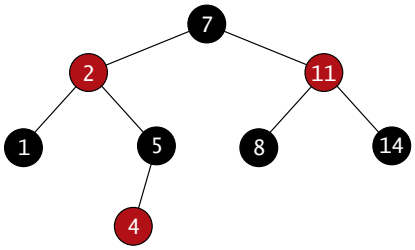
3. Draw the B+ tree that would be formed by inserting the integers shown in Exercise 1.
4. Show the tree after inserting each of the following values one at a time: 1, 5, 9, and 13.



5 Build a 2–3–4 tree to store the words in the sentence "Now is the time for all good people to come to the aid of others".

6 Draw the Red–Black tree equivalent of the 2–3–4 tree shown in Exercise 4.

7 Draw the 2–3–4 tree equivalent to the following Red–Black tree.



PROGRAMMING

1. Code the `binarySearch` method of the `BTree`.
2. Code the `insert` method for the 2–3–4 tree. The rest of the 2–3–4 tree implementation can be done by taking the B-tree implementation and fixing `order` at 4.

Chapter Review

- Tree balancing is necessary to ensure that a search tree has O(log n) behavior. Tree balancing is done as part of an insertion or removal.
- An AVL tree is a balanced binary tree in which each node has a balance value that is equal to the difference between the heights of its right and left subtrees ($h_R - h_L$). A node is balanced if it has a balance value of 0; a node is left(right)-heavy if it has a balance of –1 (+1). Tree balancing is done when a node along the insertion (or removal) path becomes critically out of balance; that is, the absolute value of the difference of the height of its two subtrees is 2. The rebalancing is done after returning from a recursive call in the `add` or `delete` method.

- For an AVL tree, there are four kinds of imbalance and a different remedy for each.
  - — Left–Left (parent balance is –2, left child balance is –1): rotate right around parent.
  - — Left–Right (parent balance is –2, left child balance is +1): rotate left around child, then rotate right around parent.
  - — Right–Right (parent balance is +2, right child balance is +1): rotate left around parent.
  - — Right–Left (parent balance is +2, right child balance is –1): rotate right around child, then rotate left around parent.
- A Red–Black tree is a balanced tree with red and black nodes. After an insertion or removal, the following invariants must be maintained for a Red–Black tree:
  - — A node is either red or black.
  - — The root is always black.
  - — A red node always has black children. (A `null` reference is considered to refer to a black node.)
  - — The number of black nodes in any path from the root to a leaf is the same.
- To maintain tree balance in a Red–Black tree, it may be necessary to recolor a node and also to rotate around a node. The rebalancing is done inside the `add` or `delete` method, right after returning from a recursive call.
- Trees whose nodes have more than two children are an alternative to balanced binary search trees. These include 2–3 and 2–3–4 trees. A 2-node has two children, a 3-node has three children, and a 4-node has four children. The advantage of these trees is that keeping the trees balanced is a simpler process. Also, the tree may be less deep because a 3-node can have three children and a 4-node can have four children, but they still have O(log *n*) behavior.
- A `B`-tree of order *n* is a tree whose nodes can store up to n – 1 items and have *n* children and is a generalization of a 2–3 tree. `B`-trees are used as indexes to large databases stored on disk. The value of *n* is chosen so that each node is as large as it can be and still fit in a disk block. The time to retrieve a block is large compared to the time required to process it in memory. By making the tree nodes as large as possible, we reduce the number of disk accesses required to find an item in the index.
- A 2–3–4 tree can be balanced on the way down the insertion path by splitting a 4-node into two 2-nodes before inserting a new item. This is easier than splitting nodes and rebalancing after returning from an insertion.

Java Classes Introduced in This Chapter

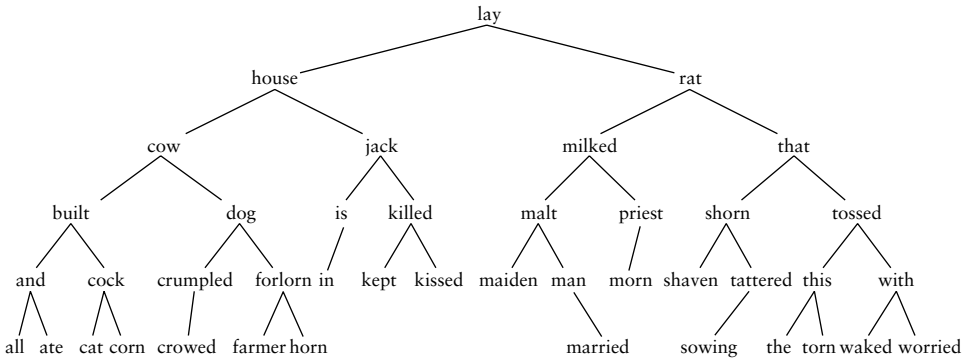
`java.util.TreeMap`

User-Defined Interfaces and Classes in This Chapter

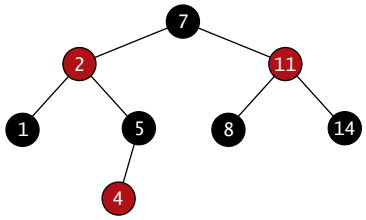
`AVLTree`
`AVLTree.AVLNode`
`BinarySearchTreeWithRotate`
`BTree`
`BTree.Node`
`RedBlackTree`
`RedBlackTree.RedBlackNode`
`TwoThreeFourTree`
`TwoThreeFourTree.Node`

Quick-Check Exercises

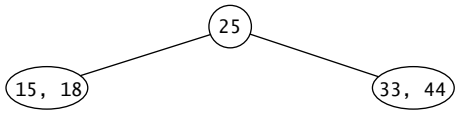
1. For the following AVL tree, where would *mouse* be inserted? What kind of imbalance occurs, and what is the remedy?



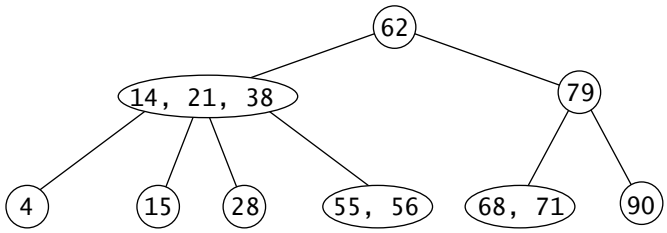
2. Show the following Red–Black tree after inserting 12 and then 13. What kind of rotation, if any, is performed?



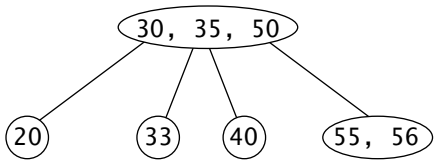
3. Show the following 2–3 tree after inserting 55 and then 22.



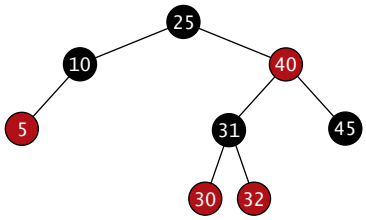
4. Show the following 2–3–4 tree after inserting 42 and then 52.



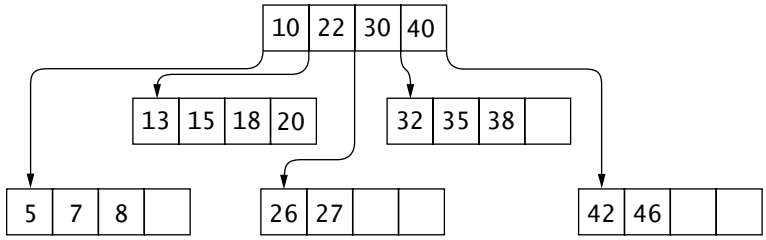
5. Draw the Red–Black tree equivalent to the following 2–3–4 tree.



6. Draw the 2–3–4 tree equivalent to the following Red–Black tree.



7. Show the following B-tree after inserting 50 and 21. What key is found in the root? What is the height of the new tree?



Review Questions

1. Draw the mirror images of the three cases for insertion into a Red–Black tree and explain how each situation is resolved.
2. Show the AVL tree that would be formed by inserting the month names (12 strings) into a tree in their normal calendar sequence.
3. Show the Red–Black tree that would be formed by inserting the month names into a tree in their normal calendar sequence.
4. Show the 2–3 tree that would be formed by inserting the month names into a tree in their normal calendar sequence.
5. Show the 2–3–4 tree that would be formed by inserting the month names into a tree in their normal calendar sequence.
6. Show a B-tree of capacity 5 that would be formed by inserting the month names into a tree in their normal calendar sequence.

Programming Projects

1. Complete the `AVLTree` class by coding the missing methods for insertion only. Use it to insert a collection of randomly generated numbers. Insert the same numbers in a binary search tree that is not balanced. Verify that each tree is correct by performing an inorder traversal. Also, display the format of each tree that was built and compare their heights.
2. Code the `RedBlackTree` class by coding the missing methods for insertion. Redo Project 1 using this class instead of the `AVLTree` class.
3. Code the `TwoThreeFourTree` class by coding the missing methods. Redo Project 1 using this class instead of the `AVLTree` class.
4. Code the `TwoThreeTree` class. Redo Project 1 using this class instead of the `AVLTree` class.
5. Complete the `AVLTree` class by providing the missing methods for removal. Demonstrate that these methods work.

   Review the changes required for methods `decrementBalance`, `incrementBalance`, `rebalanceLeft`, and `rebalanceRight` discussed at the end of Section 9.2. Also, modify `rebalanceLeft` (and `rebalanceRight`) to consider the cases where the left (right) subtree is balanced. This case can result when there is a removal from the right (left) subtree that causes the critical imbalance to occur. This

is still a Left–Left (Right–Right) case, but after the rotation the overall balances are not zero. This is illustrated in Figures 9.65 and 9.66 where an item is removed from subtree c.

FIGURE 9.65
Left–Left Imbalance with Left Subtree Balanced

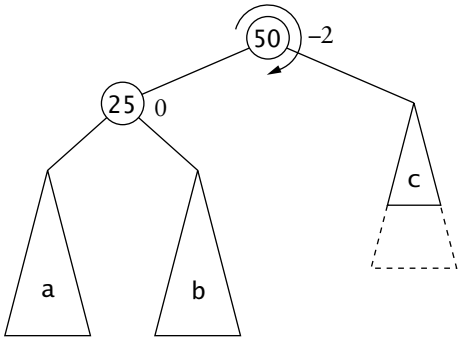
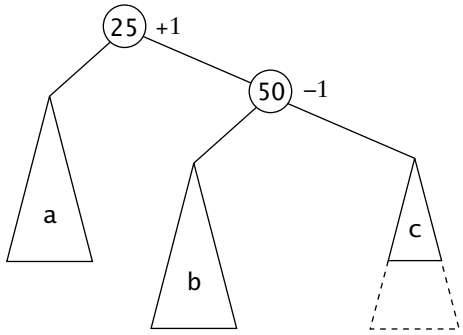


FIGURE 9.66
All Trees Unbalanced after Rotation



In addition, the Left–Right (or Right–Left) case can have a case in which the Left–Right (Right–Left) subtree is balanced. In this case, after the double rotation is performed, all balances are zero. This is illustrated in Figures 9.67 through 9.69.

FIGURE 9.67
Left–Right Case with Left–Right Subtree Balanced

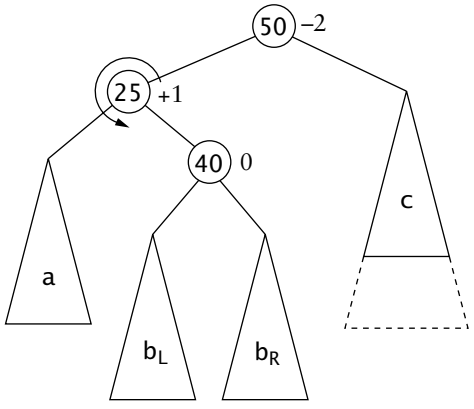


FIGURE 9.68
Imbalance after Single Rotation

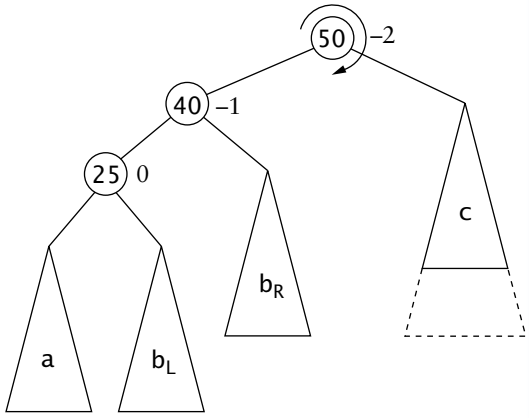
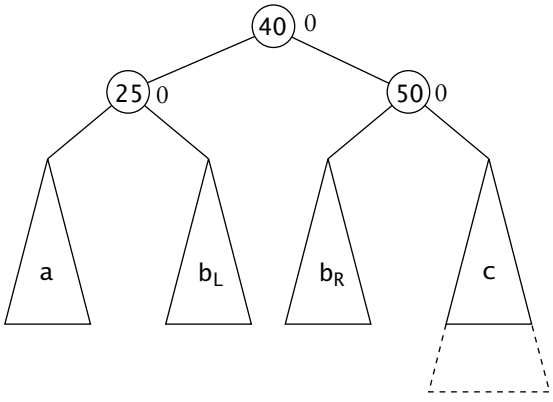


FIGURE 9.69
Complete Balance after Double Rotation

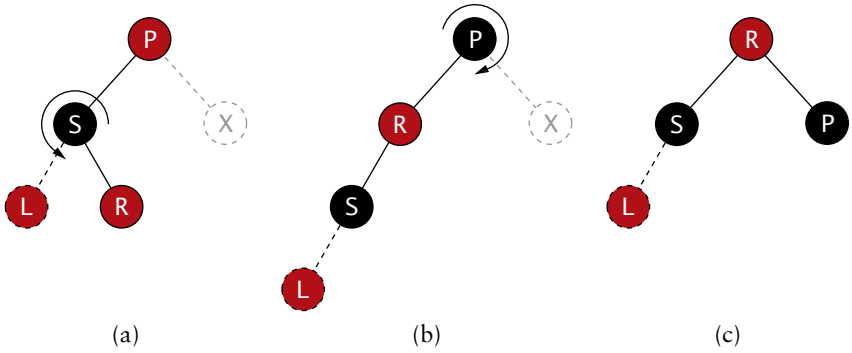


6. Complete the `RedBlackTree` class by coding the missing methods for removal. The methods `remove` and `findLargestChild` are adapted from the corresponding methods of the `BinarySearchTree` class. These adaptations are similar to those done for the AVL tree. A data field `fixupRequired` performs a role analogous to the `decrease` data field in the AVL tree. It is set when a black node is removed. Upon return from a method that can remove a node, this variable is tested. If the removal is from the right, then a new method `fixupRight` is called. If the removal is from the left, then a new method `fixupLeft` is called.

   The `fixupRight` method must consider five cases, as follows:

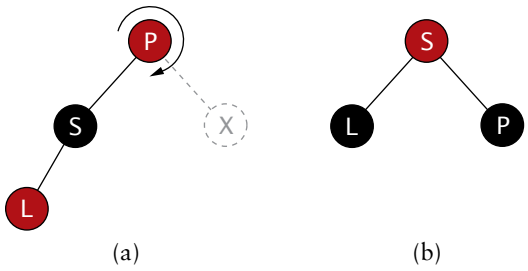
   - Case 1: Parent is red and the deleted node's sibling has a red right child. Figure 9.70(a) shows a red node P that is the root of a subtree that has lost a black node X from its right subtree. The root of the left subtree of P is S, and it must be a black node. If S has a red right child, as shown in the figure, we can restore the black balance. First we rotate the left subtree left and change the color of its parent (P) to black (see Figure 9.70(b)). Then we rotate right about the parent as shown in Figure 9.70(c). This restores the black balance. As shown in the figure, the node S may also have a left child. This does not affect the results.

FIGURE 9.70
Red–Black Removal Case 1



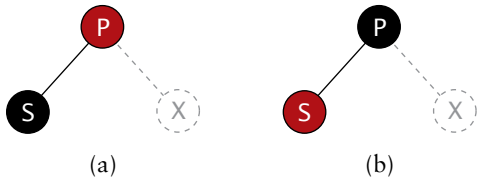
   - Case 2: Parent is red, and the deleted node's sibling has only a left red child. Figure 9.71(a) shows the case where the red parent P has a left child S that has a red left child L. In this case, we change the color of S to red and the color of P to black. Then we rotate right as shown in Figure 9.71(b).

FIGURE 9.71
Red–Black Removal Case 2



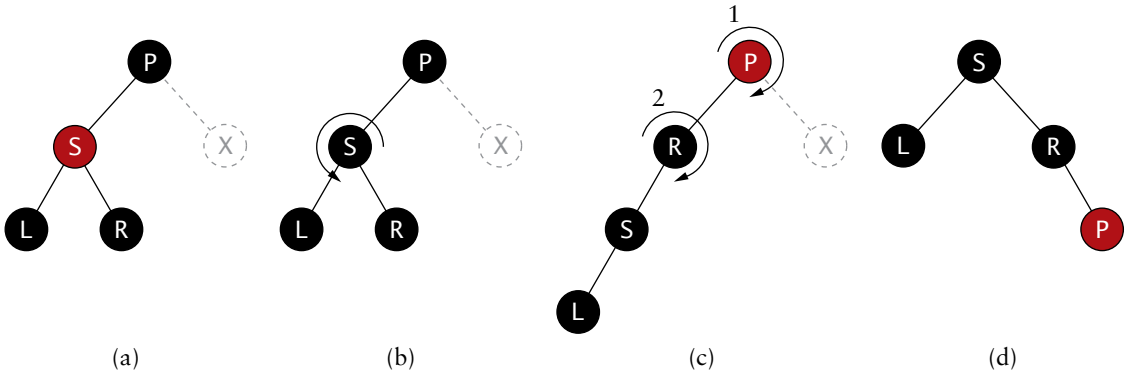
   - Case 3: Parent is red, and the left child has no red children. Figure 9.72(a) shows the case where the red parent P has a left child S that has no children. As in the next two cases, this fixup process is started at the bottom of the tree but can move up the tree. In this case, S may have black children, and X may represent the root of a subtree whose black height is one less than the black height of S. The correction is quite easy. We change P to black and S to red (see Figure 9.72(b)). Now the balance is restored, and the black height at P remains the same as it was before the black height at X was reduced.

FIGURE 9.72
Red–Black Removal Case 3



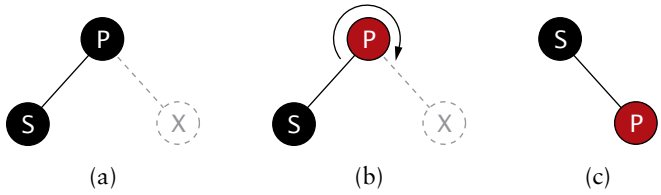
- Case 4: Parent is black and left child is red. Figure 9.73(a) shows the case where the parent P is black and the left child S is red. Since the black heights of S and X were equal before removing X, S must have two black children. We first change the color of the left child to black as shown in Figure 9.73(b). We rotate the child S left and change the color of P to red as shown in Figure 9.73(c). Then we rotate right twice, so that S is now where P was, thus restoring the black balance.

FIGURE 9.73
Red–Black Removal Case 4



- Case 5: Parent is black and left child is black. Figure 9.74(a) shows the case where P is black and S is black. We then change the color of the parent to red and rotate. The black height of P has been reduced. Thus, we repeat the process at the next level (P's parent).

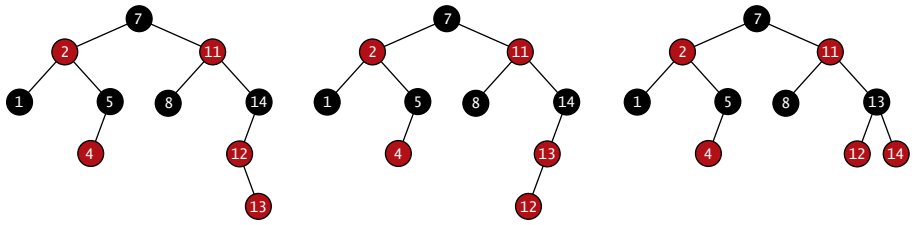
FIGURE 9.74
Red–Black Removal Case 5



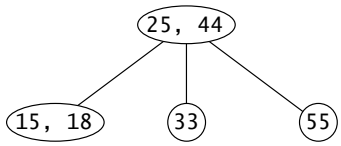
Answers to Quick-Check Exercises

1. Node *mouse* is inserted to the right of *morn*. Node *morn* has a balance of +1, and node *priest* has a balance of –2. This is a case of Left–Right imbalance. Rotate left around *morn* and right around *priest*. Node *mouse* will have *morn (priest)* as its left (right) subtree.
2. When we insert 12 as a red node, it has a black parent, so we are done. When we insert 13, we have the situation shown in the first of the following figures. This is the mirror image of Case 3 in Figure 9.25. We correct it by first rotating left around 12, giving the second of the following figures.

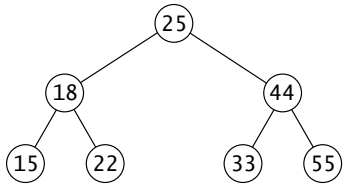
Then we change 14 to red and 13 to black and rotate right around 13, giving the tree in the third figure.



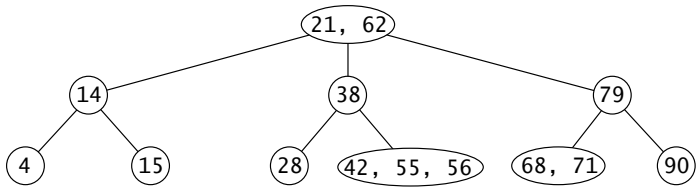
3. The 2–3 tree after inserting 55 is as follows.



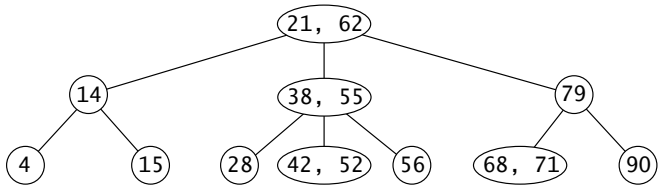
The 2–3 tree after inserting 22 is as follows.



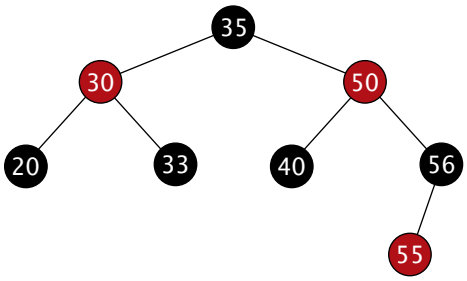
4. When 42 is inserted, the 4-node 14, 21, 38 is split and 21 is inserted into the root, 62. The node-14 has the children 4 and 15, and the node-38 has the children 28 and the 3-node 55, 56. We then insert 42 into the 3-node, making it a 4-node. The result follows.



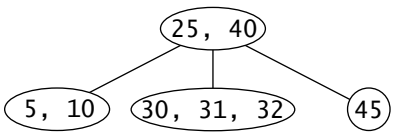
When we insert 52, the 4-node 42, 55, 56 is split and the 55 is inserted into the 2-node 38. Then 52 is inserted into the resulting 2-node, 42, making it a 3-node, as follows.



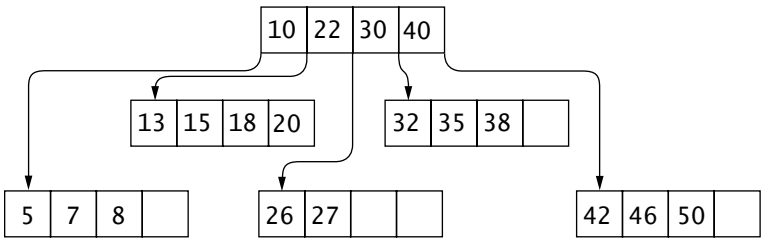
5. The equivalent Red–Black tree follows.



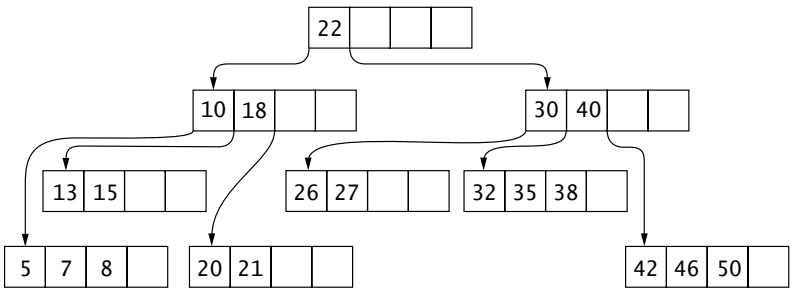
6. The equivalent 2–3–4 tree follows.



7. Insert 50 in a leaf.



To insert 21, we need to split node {13, 15, 18, 20} and pass 18 up. Then we split the root and pass 22 up to the new root.



Chapter 10

Graphs

Chapter Objectives

- To become familiar with graph terminology and the different types of graphs
- To study a Graph ADT (abstract data type) and different implementations of the Graph ADT
- To learn the breadth-first and depth-first search traversal algorithms
- To learn some algorithms involving weighted graphs
- To study some applications of graphs and graph algorithms

One of the limitations of trees is that they cannot represent information structures in which a data item has more than one parent. In this chapter, we introduce a data structure known as a *graph* that will allow us to overcome this limitation.

Graphs and graph algorithms were being studied long before computers were invented. The advent of the computer made the application of graph algorithms to real-world problems possible. Graphs are especially useful in analyzing networks. Thus, it is not surprising that much of modern graph theory and application was developed at Bell Laboratories, which needed to analyze the very large communications network that is the telephone system. Graph algorithms are also incorporated into the software that makes the Internet function. You can also use graphs to describe a road map, airline routes, or course prerequisites. Computer chip designers use graph algorithms to determine the optimal placement of components on a silicon chip.

You will learn how to represent a graph, determine the shortest path through a graph, and find the minimum subset of a graph.

Graphs

10.1 Graph Terminology

A graph is a data structure that consists of a set of *vertices* (or nodes) and a set of *edges* (relations) between the pairs of vertices. The edges represent paths or connections between the vertices. Both the set of vertices and the set of edges must be finite, and either set may be empty. If the set of vertices is empty, naturally the set of edges must also be empty. We restrict our discussion to simple graphs in which there is at most one edge from a given vertex to another vertex.

EXAMPLE 10.1 The following set of vertices, V, and set of edges, E, define a graph that has five vertices, with labels A through E, and four edges.

V = {A, B, C, D, E}

E = {{A, B}, {A, D}, {C, E}, {D, E}}

Each edge is a set of two vertices. There is an edge between A and B (the edge {A, B}), between A and D, between C and E, and between D and E. If there is an edge between any pair of vertices x, y, this means there is a path from vertex x to vertex y and vice versa. We discuss the significance of this shortly.

Visual Representation of Graphs

Visually we represent vertices as points or labeled circles and the edges as lines joining the vertices. Figure 10.1 shows the graph from Example 10.1.

There are many ways to draw any given graph. The physical layout of the vertices, and even their labeling, are not relevant. Figure 10.2 shows two ways to draw the same graph.

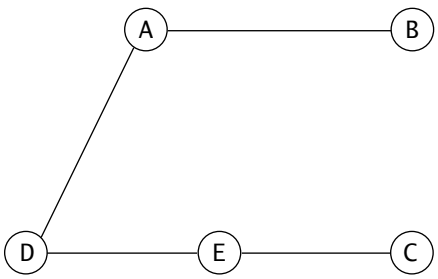


FIGURE 10.1 Graph Given in Example 10.1

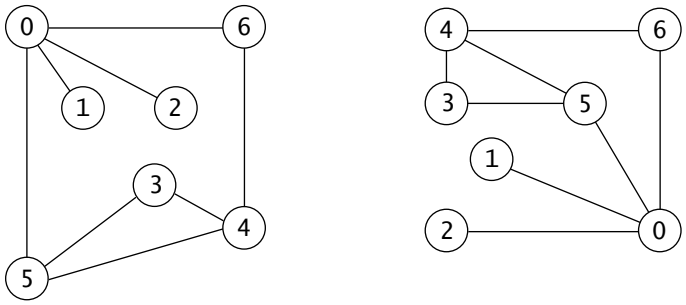


FIGURE 10.2
Two Representations of the Same Graph

Directed and Undirected Graphs

The edges of a graph are *directed* if the existence of an edge from A to B does not necessarily guarantee that there is a path in both directions. A graph that contains directed edges is known as a *directed graph* or *digraph*, and a graph that contains undirected edges is known as an *undirected graph* or simply a graph. A directed edge is like a one-way street; you can travel on it in only one direction. Directed edges are represented as lines with an arrow on one end, whereas undirected edges are represented as single lines. The graph in Figure 10.1 is undirected; Figure 10.3 shows a directed graph. The set of edges for the directed graph follows:

$$E = \{(A, B), (B, A), (B, E), (D, A), (E, A), (E, C), (E, D)\}$$

Each edge above is an ordered pair of vertices instead of a set as in an undirected graph. The edge (A, B) means there is a path from A to B. Observe that there is a path from both A to B and from B to A, but these are the only two vertices where there is an edge in both directions. Our convention will be to denote an edge for a directed graph as an ordered pair (u, v) where this notation means that v (the destination) is adjacent to u (the source). We denote an edge in an undirected graph as the set $\{u, v\}$, which means that u is adjacent to v and v is adjacent to u. Therefore, you can create a directed graph that is equivalent to an undirected graph by substituting for each edge $\{u, v\}$ the ordered pairs (u, v) and (v, u). In general, when we describe graph algorithms in this chapter, we will use the ordered pair notation (u, v) for an edge.

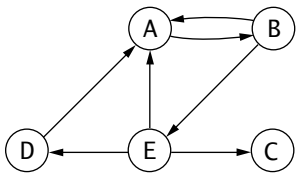


FIGURE 10.3
Example of a Directed Graph

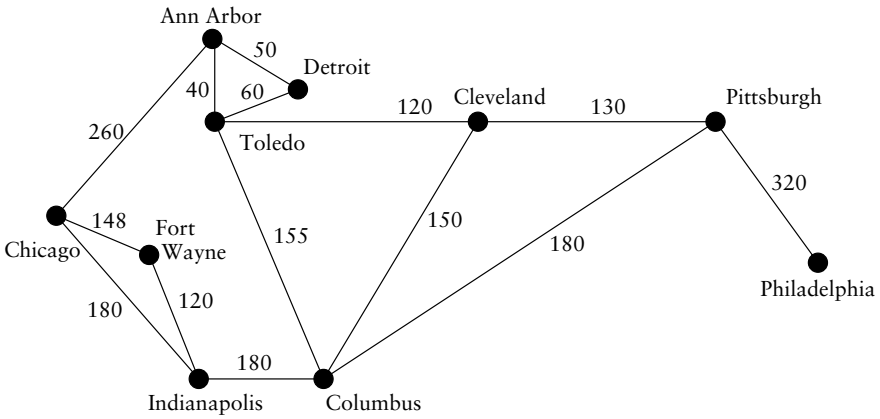
The edges in a graph may have values associated with them known as their *weights*. A graph with weighted edges is known as a *weighted graph*. In an illustration of a weighted graph, the weights are shown next to the edges. Figure 10.4 shows an example of a weighted graph. Each weight is the distance between the two cities (vertices) connected by the edge. Generally, the weights are nonnegative, but there are graph problems and graph algorithms that deal with negative weighted edges.

Paths and Cycles

One reason we study graphs is to find pathways between vertices. We use the following definitions to describe pathways between vertices.

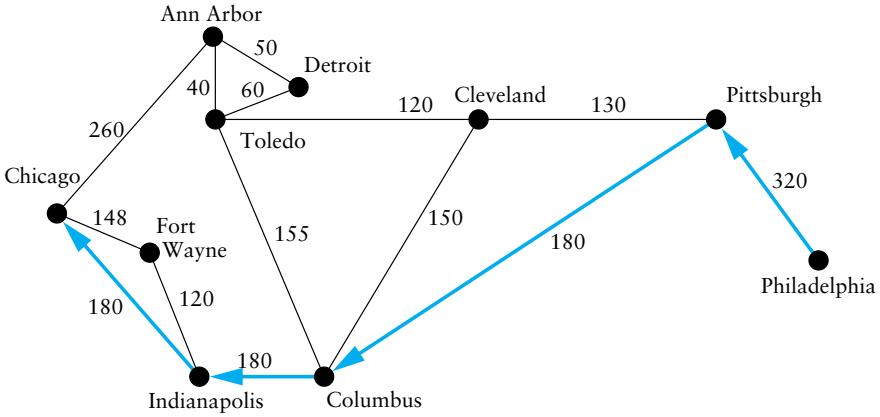
- A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex. In Figure 10.4, Philadelphia is adjacent to Pittsburgh. In Figure 10.3, A is adjacent to D, but since this is a directed graph, D is not adjacent to A.

FIGURE 10.4
Example of a Weighted Graph



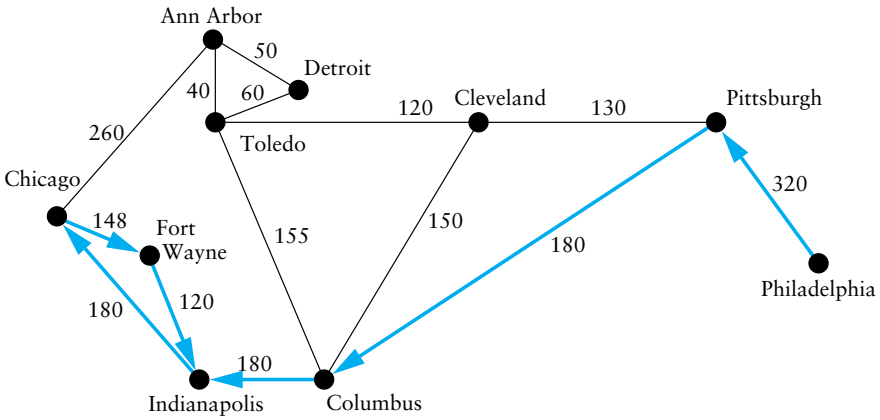
- A *path* is a sequence of vertices in which each successive vertex is adjacent to its predecessor. In Figure 10.5, the following sequence of vertices is a path: Philadelphia → Pittsburgh → Columbus → Indianapolis → Chicago.

FIGURE 10.5
A Simple Path



- In a *simple path*, the vertices and edges are distinct, except that the first and last vertices may be the same. In Figure 10.5, the path Philadelphia → Pittsburgh → Columbus → Indianapolis → Chicago is a simple path. The path Philadelphia → Pittsburgh → Columbus → Indianapolis → Chicago → Fort Wayne → Indianapolis is a path but not a simple path (see Figure 10.6).

FIGURE 10.6
Not a Simple Path



- A *cycle* is a simple path in which only the first and final vertices are the same. In Figure 10.7, the path Pittsburgh → Columbus → Toledo → Cleveland → Pittsburgh is a cycle. For an undirected graph, a cycle must contain at least three distinct vertices. Thus, Pittsburgh → Columbus → Pittsburgh is not considered a cycle.

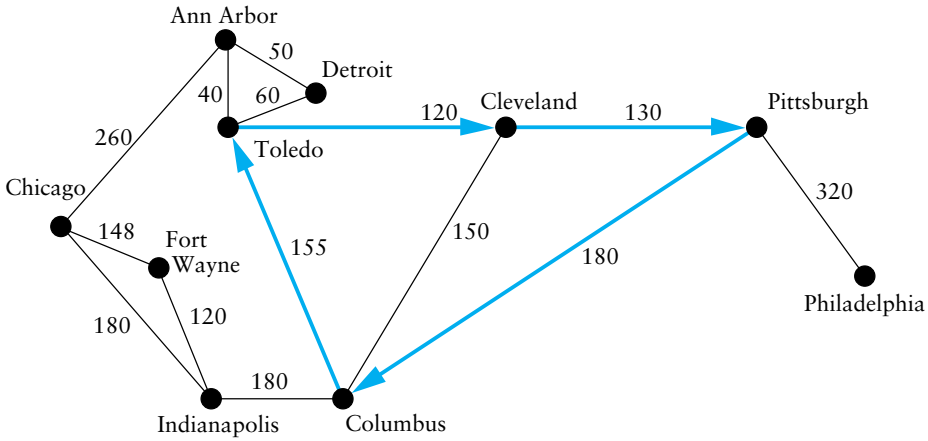


FIGURE 10.7
Ann Arbor

- An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex. Figure 10.7 is a connected graph, whereas Figure 10.8 is not.

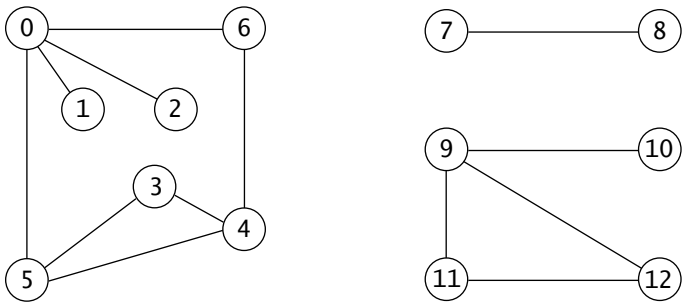


FIGURE 10.8
Example of an Unconnected Graph

- If a graph is not connected, it is considered *unconnected*, but it will still consist of *connected components*. A connected component is a subset of the vertices and the edges connected to those vertices where there is a path between every pair of vertices in the component. A single vertex with no edges is also considered a connected component. Figure 10.8 consists of the connected components {0, 1, 2, 3, 4, 5, 6}, {7, 8}, and {9, 10, 11, 12}.

Relationship between Graphs and Trees

The graph is the most general of the data structures we have studied. It allows for any conceivable relationship among the data elements (vertices). A tree is actually a special case of a graph. Any graph that is connected and contains no cycles can be viewed as a tree by picking one of its vertices (nodes) as the root. For example, the graph shown in Figure 10.1 can be viewed as a tree if we consider the node labeled D to be the root (see Figure 10.9).

Graph Applications

We can use graphs to help solve several different kinds of problems. For example, we might want to know whether there is a connection from one node in a network to all others. If

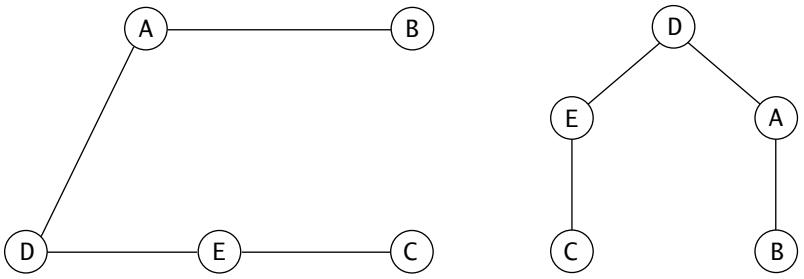


FIGURE 10.9
A Graph Viewed as a Tree

we can show that the graph is connected, then a path must exist from one node to every other node.

In college you must take some prerequisite courses before you can take others. Some courses have multiple prerequisites, and some prerequisites have prerequisites of their own. It can be quite confusing. You may even feel that there is a loop in the maze of prerequisites and that it is impossible to schedule your classes to meet the prerequisites. We can represent the set of prerequisites by a directed graph. If the graph has no cycles, then we can find a solution. We can also find the cycles.

Another application would be finding the least-cost path or shortest path from each vertex to all other vertices in a weighted graph. For example, in Figure 10.4, we might want to find the shortest path from Philadelphia to Chicago. Or we might want to create a table showing the distance (miles in the shortest route) between each pair of cities.

EXERCISES FOR SECTION 10.1

SELF-CHECK

1. In the graph shown in Figure 10.1, what vertices are adjacent to D? Also check in Figure 10.3.
2. In Figure 10.3, is it possible to get from A to all other vertices? How about from C?
3. In Figure 10.4, what is the shortest path from Philadelphia to Chicago?

10.2 The Graph ADT and Edge Class

Java does not provide a Graph ADT, so we have the freedom to design our own. To write programs for the applications mentioned at the end of the previous section, we need to be able to navigate through a graph or traverse it (visit all its vertices). To accomplish this, we need to be able to advance from one vertex in a graph to all its adjacent vertices. Therefore, we need to be able to do the following:

1. Create a graph with the specified number of vertices.
2. Iterate through all the vertices in the graph.
3. Iterate through the vertices that are adjacent to a specified vertex.
4. Determine whether an edge exists between two vertices.
5. Determine the weight of an edge between two vertices.
6. Insert an edge into the graph.

TABLE 10.1
The `Graph` Interface

Methods	Behavior
`int getNumV()`	Returns the number of vertices in the graph
`boolean isDirected()`	Returns an indicator of whether the graph is directed
`Iterator<Edge> edgeIterator(int source)`	Returns an iterator to the edges that originate from a given vertex
`Edge getEdge(int source, int dest)`	Gets the edge between two vertices
`void insert(Edge e)`	Inserts a new edge into the graph
`boolean isEdge(int source, int dest)`	Determines whether an edge exists from vertex `source` to `dest`
`default void loadEdgesFromFile(Scanner scan)`	A default method that loads the edges from a file associated with `scan`

Except for item 1 in the above list, we can specify these requirements in a Java interface. Since a Java interface cannot include a constructor, the requirements for item 1 can only be specified in the comment at the beginning of the interface. Table 10.1 describes the `Graph` interface.

The `default` method `loadEdgesFromFile` reads individual edges from lines of a file. Back in Chapter 1 we mentioned that Java permitted the use of `default` methods in interfaces. This is the first time we have taken advantage of that capability. Without it, implementations of the `Graph` ADT would be more complicated.

Representing Vertices and Edges

Before we can implement this interface, we must decide how to represent the vertices and edges of a graph. We can represent the vertices by integers from 0 up to, but not including, $|V|$. ($|V|$ means the *cardinality of* V, or the number of vertices in set V.) For edges, we will define the class `Edge` that will contain the source vertex, the destination vertex, and the weight. For unweighted edges, we will use the default value of 1.0.

Table 10.2 shows the `Edge` class. Observe that an `Edge` is directed. For undirected graphs, we will always have two `Edge` objects: one in each direction for each pair of vertices that has an edge between them. A vertex is represented by a type `int` variable. Programming Exercise 1 asks you to implement the `Edge` class.

TABLE 10.2
The `Edge` Class

Data Field	Attribute
`private int dest`	The destination vertex for an edge
`private int source`	The source vertex for an edge
`private double weight`	The weight
Constructor	**Purpose**
`public Edge(int source, int dest)`	Constructs an `Edge` from `source` to `dest`. Sets the `weight` to 1.0
`public Edge(int source, int dest, double w)`	Constructs an `Edge` from `source` to `dest`. Sets the `weight` to `w`

TABLE 10.2
The Edge Class (*Continued*)

Method	Behavior
`public boolean equals(Object o)`	Compares two edges for equality. Edges are equal if their source and destination vertices are the same. The weight is not considered
`public int getDest()`	Returns the destination vertex
`public int getSource()`	Returns the source vertex
`public double getWeight()`	Returns the weight
`public int hashCode()`	Returns the hash code for an `Edge`. The hash code depends only on the source and destination
`public String toString()`	Returns a string representation of the `Edge`

EXERCISES FOR SECTION 10.2

SELF-CHECK

1. Use the constructors in Table 10.2 to create the `Edge` objects connecting vertices 9 through 12 for the graph in Figure 10.8.

PROGRAMMING

1. Implement the `Edge` class.

10.3 Implementing the Graph ADT

Because graph algorithms have been studied and implemented throughout the history of computer science, many of the original publications of graph algorithms and their implementations did not use an object-oriented approach and did not even use ADTs. The implementation of the graph was done in terms of fundamental data structures that were used directly in the algorithm. Different algorithms would use different representations.

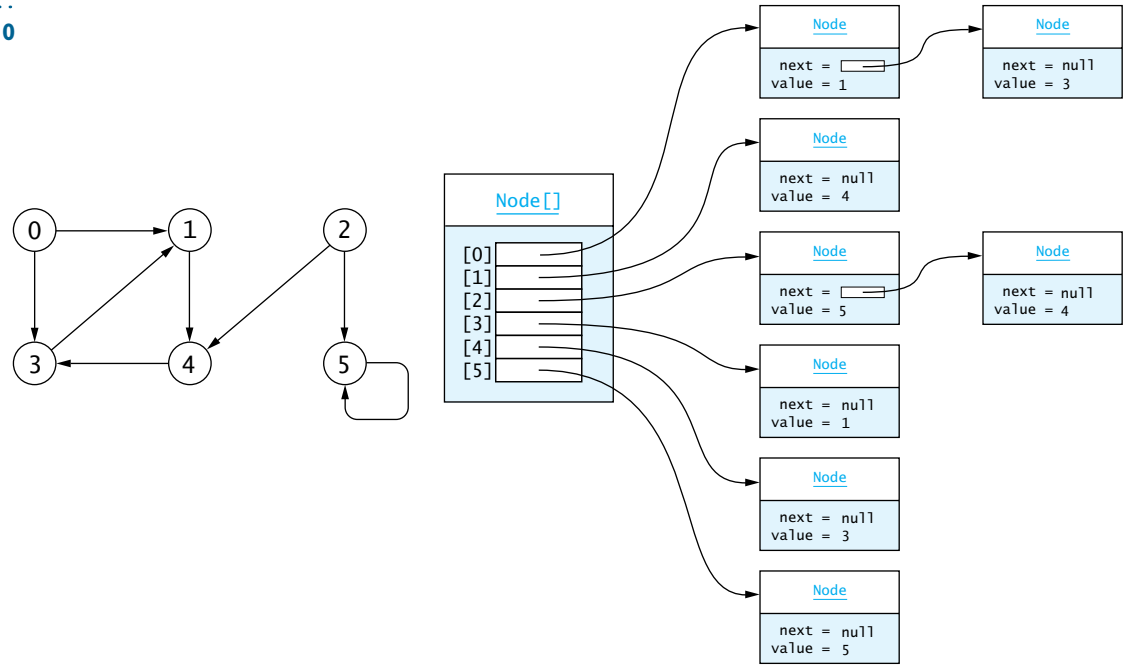
Two representations of graphs are most common:

- Edges are represented by an array of lists called *adjacency lists*, where each list stores the vertices adjacent to a particular vertex.
- Edges are represented by a two-dimensional array, called an *adjacency matrix*, with $|V|$ rows and $|V|$ columns.

Adjacency List

An adjacency list representation of a graph uses an array of lists. There is one list for each vertex. Figure 10.10 shows an adjacency list representation of a directed graph. The list referenced by array element 0 shows the vertices (**1** and 3) that are adjacent to vertex 0. The vertices are in no particular order. For simplicity, we are showing just the destination vertex as the value field in each node of the adjacency list, but in the actual implementation the

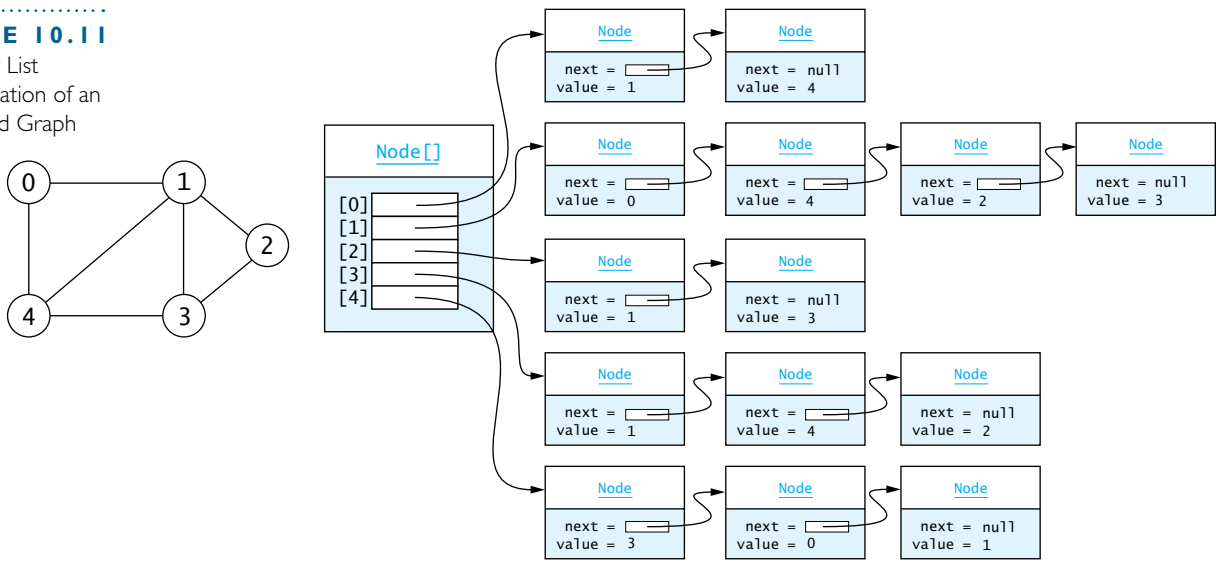
FIGURE 10.10
Adjacency List Representation of a Directed Graph



entire `Edge` will be stored. Instead of storing `value = 1` (the destination vertex) in the first vertex adjacent to 0, we will store a reference to the `Edge (0, 1, 1.0)` where 0 is the source, `1` is the destination, and `1.0` is the weight. The `Edge` must be stored (not just the destination) because weighted graphs can have different values for weights.

For an undirected graph (or simply a "graph"), symmetric entries are required. Thus, if $\{u, v\}$ is an edge, then v will appear on the adjacency list for u and u will appear on the adjacency list for v. Figure 10.11 shows the adjacency list representation for an undirected graph. The actual lists will store references to `Edges`.

FIGURE 10.11
Adjacency List Representation of an Undirected Graph



Adjacency Matrix

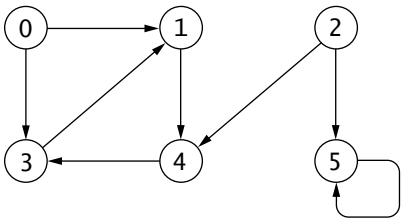
The adjacency matrix uses a two-dimensional array to represent the graph. For an unweighted graph, the entries in this matrix can be **`boolean`** values, where **`true`** represents the presence of an edge and **`false`** its absence. Another popular method is to use the value 1 for an edge and 0 for no edge. The integer coding has benefits over the **`boolean`** approach for some graph algorithms that use matrix multiplication.

For a weighted graph, the matrix would contain the weights. Since 0 is a valid weight, we will use `Double.POSITIVE_INFINITY` (a special `double` value in Java that approximates the mathematical behavior of infinity) to indicate the absence of an edge, and in an unweighted graph we will use a weight of 1.0 to indicate the presence of an edge.

Figure 10.12 shows a directed graph and the corresponding adjacency matrix. Instead of using `Edge` objects, an edge is indicated by the value 1.0, and the lack of an edge is indicated by a blank space.

If the graph is undirected, then the matrix is symmetric, and only the lower diagonal of the matrix needs be saved (the colored squares in Figure 10.13).

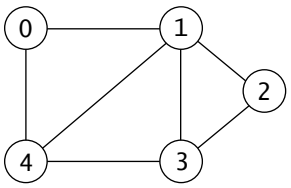
FIGURE 10.12
Directed Graph and Corresponding Adjacency Matrix



Column

Row	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1.0		1.0		
[1]					1.0	
[2]					1.0	1.0
[3]		1.0				
[4]				1.0		
[5]						1.0

FIGURE 10.13
Undirected Graph and Adjacency Matrix Representation



Column

Row	[0]	[1]	[2]	[3]	[4]
[0]		1.0			1.0
[1]	1.0		1.0	1.0	1.0
[2]		1.0		1.0	
[3]		1.0	1.0		1.0
[4]	1.0	1.0		1.0	

Overview of the Hierarchy

We will describe Java classes that use each representation. Each class will extend the interface `Graph` described earlier in Table 10.1. Class `Edge` was described in Table 10.2.

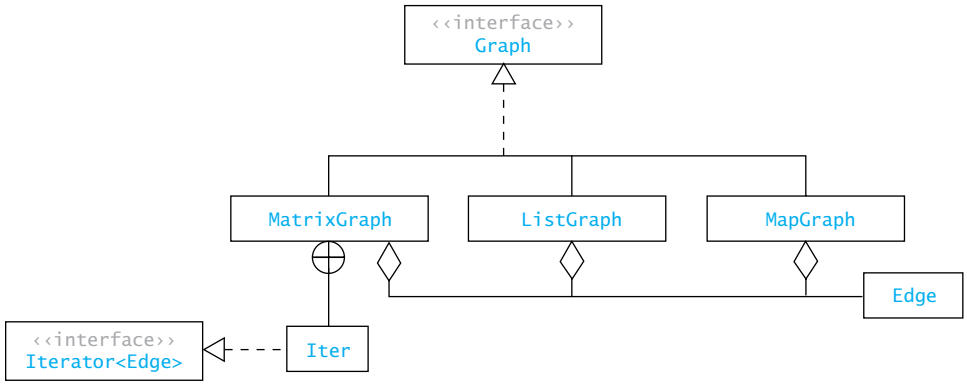


FIGURE 10.14
UML Class Diagram of `Graph` Class Hierarchy

The classes `ListGraph` and `MatrixGraph` will provide concrete representations of graphs using an adjacency list and adjacency matrix, respectively (see Figure 10.14). The `MatrixGraph` class contains an inner class (indicated by the ⊕ symbol) that we call `Iter`, which implements the `Iterator<Edge>` interface.

Declaring the Graph Interface

Listing 10.1 declares the `Graph` interface. The `default` method `loadEdgesFromFile` is left as Programming Exercise 1. Two implementations of interface `Graph`, `ListGraph` and `MatrixGraph`, follow.

LISTING 10.1
Graph.java

```
import java.util.*;

/** Interface to specify a Graph ADT. A graph is a set of vertices and
    a set of edges. Vertices are represented by integers
    from 0 to n - 1. Edges are ordered pairs of vertices.
    Each implementation of the Graph interface should
    provide a constructor that specifies the number of
    vertices and whether the graph is directed.
 */
public interface Graph {

    // Accessor Methods
    /** Return the number of vertices.
        @return The number of vertices
     */
    int getNumV();

    /** Determine whether this is a directed graph.
        @return true if this is a directed graph
     */
    boolean isDirected();

    /** Insert a new edge into the graph.
        @param edge The new edge
     */
    void insert(Edge edge);

    /** Determine whether an edge exists.
```

```
        @param source The source vertex
        @param dest The destination vertex
        @return true if there is an edge from source to dest
     */
    boolean isEdge(int source, int dest);

    /** Get the edge between two vertices.
        @param source The source vertex
        @param dest The destination vertex
        @return The edge between these two vertices
                or null if there is no edge
     */
    Edge getEdge(int source, int dest);

    /** Return an iterator to the edges connected to a given vertex.
        @param source The source vertex
        @return An Iterator<Edge> to the vertices connected to source
     */
    Iterator<Edge> edgeIterator(int source);

    /** Read source and destination vertices and weight
        (optional)for each edge from a file
         @param scan The Scanner associated with the file
     */
     default void loadEdgesFromFile(Scanner scan) { // Programming Exercise 1
   }
}
```

The ListGraph Class

The ListGraph class implements the Graph interface by providing an internal representation using an array of lists. Table 10.3 describes the ListGraph class.

The Data Fields

The class begins as follows:

```
import java.util.*;

/** A ListGraph implements the Graph interface using an
    array of lists to represent the edges.
 */
public class ListGraph implements Graph {

    // Data Fields
    /** The number of vertices */
    private final int numV;
    /** An indicator of whether the graph is directed (true) or not (false)
     */
    private final boolean directed;
    /** An array of Lists to contain the edges that
        originate with each vertex.
     */
    private List<Edge>[] edges;
    // Methods
    // Insert accessor methods getNumV and isDirected
    . . .
```

TABLE 10.3
The `ListGraph` Class

Data Field	Attribute
`private final int numV`	The number of vertices
`private final boolean directed`	Value is `true` for directed graphs
`private List<Edge>[] edges`	An array of `Lists` to contain the edges that originate from each vertex
Constructor	**Purpose**
`public ListGraph(int numV, boolean directed)`	Constructs a graph with the specified number of vertices and directionality
Method	**Behavior**
`public int getNumV()`	Returns the number of vertices in the graph
`public boolean isDirected()`	Returns an indicator of whether the graph is directed
`public Iterator<Edge> edgeIterator(int source)`	Returns an iterator to the edges that originate from vertex `source`
`public Edge getEdge(int source, int dest)`	Gets the edge between two vertices
`public void insert(Edge e)`	Inserts a new edge into the graph
`public boolean isEdge(int source, int dest)`	Determines whether an edge exists from vertex `source` to `dest`

The Constructor

The constructor allocates an array of `LinkedLists`, one for each vertex.

```
/** Construct a graph with the specified number of vertices and directionality.
    @param numV The number of vertices
    @param directed The directionality flag
 */
public ListGraph(int numV, boolean directed) {
    this.numV = numV;
    this.directed = directed;
    edges = new List[numV];
    for (int i = 0; i < numV; i++) {
        edges[i] = new LinkedList<Edge>();
    }
}
```

The `isEdge` Method

Method `isEdge` determines whether an edge exists by searching the list associated with the source vertex for an entry. This is done by calling the `contains` method for the `List`.

```
/** Determine whether an edge exists.
    @param source The source vertex
    @param dest The destination vertex
    @return true if there is an edge from source to dest
 */
public boolean isEdge(int source, int dest) {
    return edges[source].contains(new Edge(source, dest));
}
```

Observe that we had to create a dummy `Edge` object for the `contains` method to search for. The `Edge.equals` method does not check the edge weights, so the `weight` parameter is not needed.

The `insert` Method

The `insert` method inserts a new edge `(source, destination, weight)` into the graph by adding that edge's data to the list of adjacent vertices for that edge's source. If the graph is not directed, it adds a new edge in the opposite direction `(destination, source, weight)` to the list of adjacent vertices for that edge's destination.

```
/** Insert a new edge into the graph.
    @param edge The new edge
 */
public void insert(Edge edge) {
    edges[edge.getSource()].add(edge);
    if (!directed) {
        edges[edge.getDest()].add(new Edge(edge.getDest(), edge.getSource(),
                                           edge.getWeight()));
    }
}
```

The `edgeIterator` Method

The `edgeIterator` method will return an `Iterator<Edge>` object that can be used to iterate through the edges adjacent to a given vertex. Because each `LinkedList` entry in the array `edges` is a `List<Edge>`, its `iterator` method will provide the desired object. Thus, the `edgeIterator` merely calls the corresponding `iterator` method for the specified vertex.

```
public Iterator<Edge> edgeIterator(int source) {
    return edges[source].iterator();
}
```

The `getEdge` Method

Like the `isEdge` method, the `getEdge` method also requires a search. However, we need to program the search directly. We will use the enhanced `for` statement to access all edges in the list for vertex `source`. We compare each edge to a target object with source and destination set to the method arguments. The `equals` method does not compare edge weights, only the vertices.

```
/** Get the edge between two vertices.
    @param source The source
    @param dest The destination
    @return the edge between these two vertices
            or null if an edge does not exist.
 */
public Edge getEdge(int source, int dest) {
    Edge target = new Edge(source, dest, Double.POSITIVE_INFINITY);
    for (Edge edge : edges[source]) {
        if (edge.equals(target))
            return edge; // Desired edge found, return it.
    }
    // Assert: All edges for source checked.
    return null; // Desired edge not found.
}
```

The `MatrixGraph` Class

The `MatrixGraph` class implements the `Graph` interface by providing an internal representation using a two-dimensional array for storing the edge weights

```
double[][] edges;
```

When a new `MatrixGraph` object is created, the constructor sets the number of rows (vertices) in this array (`numV`) and sets the directionality flag (`directed`). It implements the same methods as class `ListGraph` and also has an inner iterator class `Iter`. It needs its own iterator class because there is no `Iterator` class associated with an array. The implementation is left as a project (Programming Project 1).

Comparing Implementations

Time Efficiency

The two implementations present a tradeoff. Which is best depends on the algorithm and the density of the graph. The density of a graph is the ratio of $|E|$ (the number of edges) to $|V|^2$. A *dense graph* is one in which $|E|$ is close to but less than $|V|^2$, and a *sparse graph* is one in which $|E|$ is much less than $|V|^2$. Therefore, for a dense graph we can assume that $|E|$ is $O(|V|^2)$, and for a sparse graph we can assume that $|E|$ is $O(|V|)$.

Many graph algorithms are of the form

```
1. for each vertex u in the graph
2.     for each vertex v adjacent to u
3.         Do something with edge (u, v).
```

For an adjacency list representation, Step 1 is $O(|V|)$ and Step 2 is $O(|E_u|)$, where $|E_u|$ is the number of edges that originate at vertex u. Thus, the combination of Steps 1 and 2 will represent examining each edge in the graph, giving $O(|E|)$. For an adjacency matrix representation, Step 2 is also $O(|V|)$, and thus the overall algorithm is $O(|V|^2)$. Thus, for a sparse graph, the adjacency list gives better performance for this type of algorithm, whereas for a dense graph, the performance is the same for either representation.

Some graph algorithms are of the form

```
1. for each vertex u in some subset of the vertices
2.     for each vertex v in some subset of the vertices
3.         if (u, v) is an edge
4.             Do something with edge (u, v).
```

For an adjacency matrix representation, Step 3 tests a matrix value and is $O(1)$, so the overall algorithm is $O(|V|^2)$. However, for an adjacency list representation, Step 3 searches a list and is $O(|E_u|)$, so the combination of Steps 2 and 3 is $O(|E|)$ and the overall algorithm is $O(|V||E|)$. For a dense graph, the adjacency matrix gives the best performance for this type of algorithm, and for a sparse graph, the performance is the same for both representations.

Thus, if a graph is dense, the adjacency matrix representation is best, and if a graph is sparse, the adjacency list representation is best. Intuitively, this makes sense because a sparse graph will lead to a sparse matrix, or one in which most entries are `POSITIVE_INFINITY`. These entries are not included in a list representation, so they will have no effect on processing time. However, they are included in a matrix representation and will have an undesirable impact on processing time.

Storage Efficiency

Note that storage is allocated for all vertex combinations (or at least half of them) in an adjacency matrix. So the storage required is proportional to $|V|^2$. If the graph is sparse (not many edges), there will be a lot of wasted space in the adjacency matrix. In an adjacency list, only the adjacent edges are stored.

On the other hand, in an adjacency list, each edge is represented by a reference to an `Edge` object containing data about the source, destination, and weight. There is also a reference to the next edge in the list. In a matrix representation, only the weight associated with an edge is stored. So each element in an adjacency list requires approximately four times the storage of an element in an adjacency matrix.

Based on this, we can conclude that the break-even point in terms of storage efficiency occurs when approximately 25 percent of the adjacency matrix is filled with meaningful data. That is, the adjacency list uses less (more) storage when less than (more than) 25 percent of the adjacency matrix would be filled.

The MapGraph Class

We can achieve the performance benefits of both the `ListGraph` and `MatrixGraph` by making a slight modification to the `ListGraph`. Replacing the array of `List<Edge>` with an array of `Map<Integer, Edge>` allows us to query the existence of an edge in O(1) time, and using the `LinkedHashMap` allows iterating through the edges adjacent to a given vertex in $O(|E_u|)$. The constructor is changed to

```
public MapGraph(int numV, boolean directed) {
    this.numV = numV;
    this.directed = directed;
    outgoingEdges = new Map[numV];
    for (int i = 0; i < numV; i++) {
        outgoingEdges[i] = new LinkedHashMap<>();
    }
}
```

The `insertEdge` method is changed to

```
public void insertEdge(Edge edge) {
    int source = edge.getSource();
    int dest = edge.getDest();
    double weight = edge.getWeight();
    outgoingEdges[source].put(dest, edge);
    if (!directed) {
        Edge reverseEdge = new Edge(dest, source, weight);
        outgoingEdges[dest].put(source, reverseEdge);
    }
}
```

The `isEdge` and `getEdge` methods are simplified to

```
public boolean isEdge(int source, int dest) {
    return outgoingEdges[source].containsKey(dest);
}

public Edge getEdge(int source, int dest) {
    return outgoingEdges[source].get(dest);
}
```

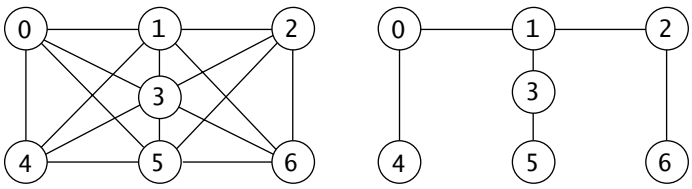
And the `edgeIterator` becomes

```
public Iterator<Edge> edgeIterator(int source) {
    return outgoingEdges[source].values().iterator();
}
```

EXERCISES FOR SECTION 10.3

SELF-CHECK

1. Represent the following graphs using adjacency lists.



2. Represent the graphs in Exercise 1 above using an adjacency matrix.
3. For each graph in Exercise 1, what are the $|V|$, the $|E|$, and the density? Which representation is best for each graph? Explain your answers.

PROGRAMMING

1. Implement the `default` method `loadEdgesFromFile` for interface `Graph`. If there are two values on a line, an edge with the default weight of 1.0 is inserted; if there are three values, the third value is the weight.
2. Implement methods `getNumV` and `isDirected` for class `ListGraph`.

10.4 Traversals of Graphs

Most graph algorithms involve visiting each vertex in a systematic order. Just as with trees, there are different ways to do this. The two most common traversal algorithms are breadth first and depth first. Although these are graph traversals, they are more commonly called *breadth-first* and *depth-first search*.

Breadth-First Search

In a breadth-first search, we visit the start node first, then all nodes that are adjacent to it next, then all nodes that can be reached by a path from the start node containing two edges, three edges, and so on. The requirement for a breadth-first search is that we must visit all nodes for which the shortest path from the start node is length k before we visit any node for which the shortest path from the start node is length $k + 1$. You can visualize a breadth-first traversal by "picking up" the graph at the vertex that is the start node, so the start node will be the highest node and the rest of the nodes will be suspended underneath it, connected by their edges. In a breadth-first search, the nodes that are higher up in the picked-up graph are visited before nodes that are lower in the graph.

Breadth-first search starts at some vertex. Unlike a tree, there is no special start vertex, so we will arbitrarily pick the vertex with label 0. We then visit it by identifying all vertices that are adjacent to the start vertex. Then we visit each of these vertices, identifying all of the vertices adjacent to them. This process continues until all vertices are visited. If the graph is not a connected graph, then the process is repeated with one of the unidentified vertices. In the discussion that follows, we use color to distinguish among three states for a node: identified (light color), visited (dark color), and not identified (white). Initially, all nodes are not identified. If a node is in the identified state, that node was encountered while visiting another, but it has not yet been visited.

Example of Breadth-First Search

Consider the graph shown in Figure 10.15. We start at vertex 0 and change it to light color (see Figure 10.16(a)). We visit 0 and see that 1 and 3 are adjacent, so we change them to light

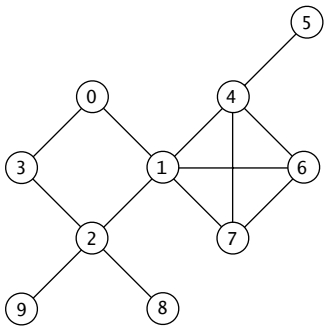
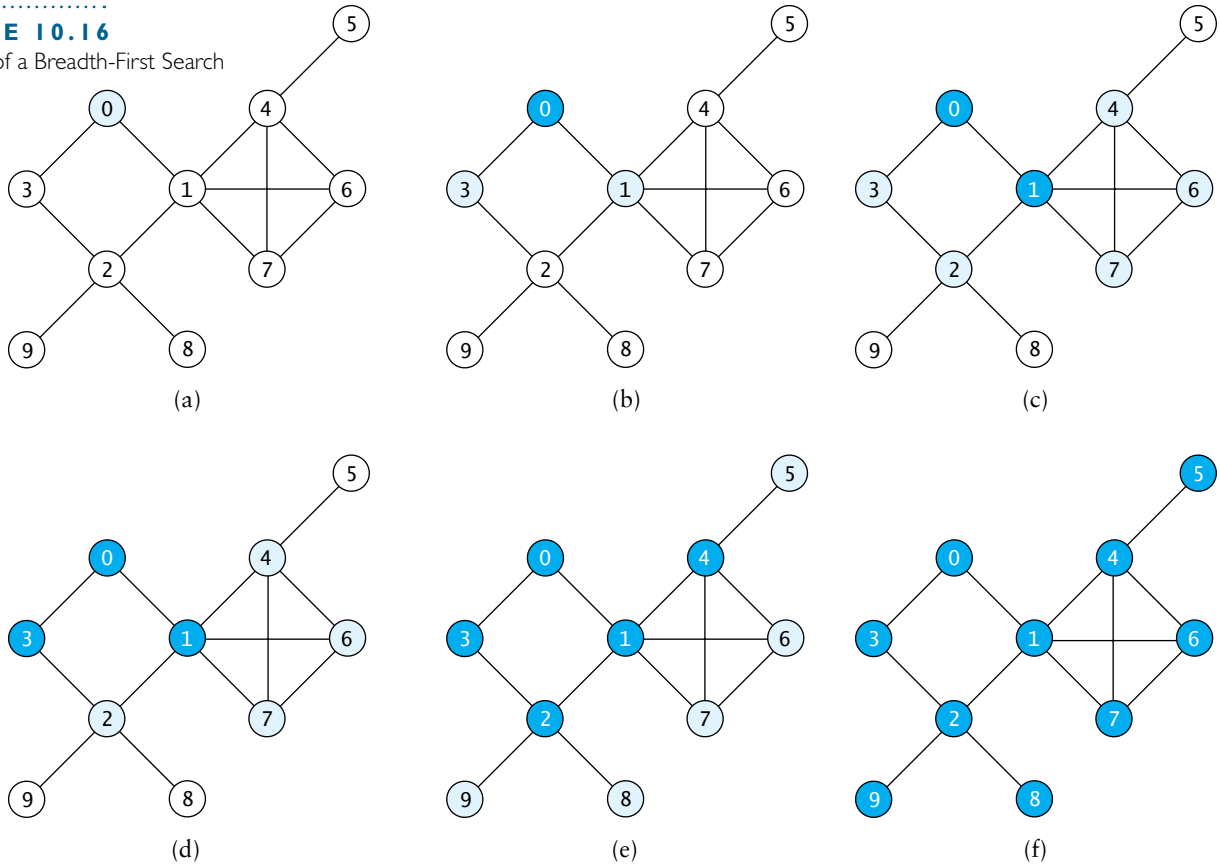


FIGURE 10.15
Graph to Be Traversed Breadth First

FIGURE 10.16
Example of a Breadth-First Search



color (to show that they have been identified). We are finished visiting 0 and now change it to dark color (see Figure 10.16(b)). So far we have visited node 0.

We always select the first node that was identified (light color) but not yet visited and visit it next. Therefore, we visit 1 and look at its adjacent vertices: 0, 2, 4, 6, and 7. We skip 0 because it is not colored white, and we change the others to light color. Then we change 1 to dark color (see Figure 10.16(c)). Now we have visited nodes 0 and 1.

Then we look at 3 (the first of the light color vertices in Figure 10.16(c) to have been identified) and see that its adjacent vertex, 2, has already been identified and 0 has been visited, so we are finished with 3 (see Figure 10.16(d)). Now we have visited nodes 0, 1, and 3, which are the starting vertex and all vertices adjacent to it.

Now we visit 2 and see that 8 and 9 are adjacent. Then we visit 4 and see that 5 is the only adjacent vertex not identified or visited (Figure 10.16(e)). Finally, we visit 6 and 7 (the last vertices that are two edges away from the starting vertex), then 8, 9, and 5, and see that there are no unidentified vertices (Figure 10.16(f)). The vertices have been visited in the sequence 0, 1, 3, 2, 4, 6, 7, 8, 9, 5.

Algorithm for Breadth-First Search

To implement breadth-first search, we need to be able to determine the first identified vertex that has not been visited so that we can visit it. To ensure that the identified vertices are visited in the correct sequence, we will store them in a queue (first-in, first-out). When we need a new node to visit, we remove it from the queue. We summarize the process in the following algorithm.

Algorithm for Breadth-First Search

1. Take an arbitrary start vertex, mark it identified (change it to light color), and place it in a queue.
2. **while** the queue is not empty
3. Take a vertex, *u*, out of the queue and visit *u*.
4. **for** all vertices, *v*, adjacent to this vertex, *u*
5. **if** *v* has not been identified or visited
6. Mark it identified (change it to light color).
7. Insert vertex *v* into the queue.
8. We are now finished visiting *u* (change it to dark color).

Table 10.4 traces this algorithm on the graph shown earlier in Figure 10.15. The initial queue contents is the start node, 0. The first line shows that after we finish visiting vertex 0, the queue contains nodes 1 and 3, which are adjacent to node 0 and are in light color in Figure 10.16(b). The second line shows that after removing 1 from the queue and visiting 1, we insert its neighbors that have not yet been identified or visited: nodes 2, 4, 6, and 7.

Table 10.4 shows that the nodes were visited in the sequence 0, 1, 3, 2, 4, 6, 7, 8, 9, 5. There are other sequences that would also be valid breadth-first traversals.

We can also build a tree that represents the order in which vertices would be visited in a breadth-first traversal, by attaching the vertices as they are identified to the vertex from which they are identified. Such a tree is shown in Figure 10.17. Observe that this tree contains

TABLE 10.4
Trace of Breadth-First Search of Graph in Figure 10.15

Vertex Being Visited	Queue Contents after Visit	Visit Sequence
0	1 3	0
1	3 2 4 6 7	0 1
3	2 4 6 7	0 1 3
2	4 6 7 8 9	0 13 2
4	6 7 8 9 5	0 13 2 4
6	7 8 9 5	0 1 3 2 4 6
7	8 9 5	0 1 3 2 4 6 7
8	9 5	0 1 3 2 4 6 7 8
9	5	0 1 3 2 4 6 7 8 9
5	Empty	0 1 3 2 4 6 7 8 9 5

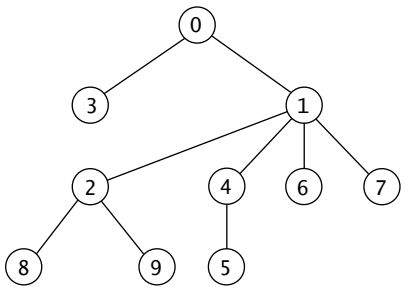


FIGURE 10.17
Breadth-First Search Tree of Graph in Figure 10.15

all of the vertices and some of the edges of the original graph. A path starting at the root to any vertex in the tree is the shortest path in the original graph from the start vertex to that vertex, where we consider all edges to have the same weight. Therefore, the *shortest path* is the one that goes through the smallest number of vertices. We can save the information we need to represent this tree by storing the parent of each vertex when we identify it (Step 7 of the breadth-first algorithm).

Refinement of Step 7 of Breadth-First Search Algorithm

7.1 Insert vertex v into the queue.
7.2 Set the parent of v to u.

Performance Analysis of Breadth-First Search

The loop at Step 2 will be performed for each vertex. The inner loop at Step 4 is performed for $|E_v|$ (the number of edges that originate at vertex v). The total number of steps is the sum of the edges that originate at each vertex, which is the total number of edges. Thus, the algorithm is $O(|E|)$.

Implementing Breadth-First Search

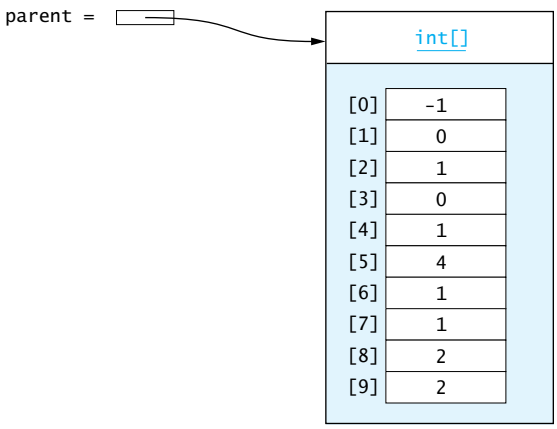
Listing 10.2 shows method `breadthFirstSearch`. Note that nothing is done when we have finished visiting a vertex (algorithm Step 8).

This method declares three data structures: `int[] parent`, `boolean[] identified`, and `Queue theQueue`. The array identified is used to keep track of the nodes that have been previously encountered, and `theQueue` is used to store nodes that are waiting to be visited.

The method returns array `parent`, which could be used to construct the breadth-first search tree. The element `parent[v]` contains the parent of vertex `v` in the tree. The statement

```
parent[neighbor] = current;
```

is used to "insert an edge into the breadth-first search tree." It does this by setting the parent of a newly identified node `(neighbor)` as the node being visited `(current)`. If we run the `breadthFirstSearch` method on the graph shown in Figure 10.15, then the array `parent` will be defined as follows:



If you compare array `parent` to Figure 10.17, you can see that `parent[i]` is the parent of vertex `i`. For example, the parent of vertex 4 is vertex 1. The entry `parent[0]` is –1 because node 0 is the start vertex.

Although array `parent` could be used to construct the breadth-first search tree, we are generally not interested in the complete tree but rather in the path from the root to a given vertex.

Using array parent to trace the path from that vertex back to the root would give you the reverse of the desired path. For example, the path derived from parent for vertex 4 to the root would be 4 to 1 to 0. If you place these vertices in a stack and then pop the stack until it is empty, you will get the path from the root: 0 to 1 to 4.

LISTING 10.2

Class BreadthFirstSearch.java

```
/** Class to implement the breadth-first search algorithm. */
public class BreadthFirstSearch {

    /** Perform a breadth-first search of a graph.
        @post The array parent will contain the predecessor of each
              vertex in the breadth-first search tree.
        @param graph The graph to be searched
        @param start The start vertex
        @return The array of parents
     */
    public static int[] breadthFirstSearch(Graph graph, int start) {
        Queue<Integer> theQueue = new LinkedList<>();
        // Declare array parent and initialize its elements to -1.
        int[] parent = new int[graph.getNumV()];
        for (int i = 0; i < graph.getNumV(); i++) {
            parent[i] = -1;
        }

        // Declare array identified and
        // initialize its elements to false.
        boolean[] identified = new boolean[graph.getNumV()];
        /* Mark the start vertex as identified and insert it into the queue */
        identified[start] = true;
        theQueue.offer(start);

        /* Perform breadth-first search until done */
        while (!theQueue.isEmpty()) {
            /* Take a vertex, current, out of the queue. and begin visiting it. */
            int current = theQueue.remove();
            /* Examine each vertex, neighbor, adjacent to current. */
            Iterator<Edge> itr = graph.edgeIterator(current);
            while (itr.hasNext()) {
                Edge edge = itr.next();
                int neighbor = edge.getDest();
                // If neighbor has not been identified
                if (!identified[neighbor]) {
                    // Mark it identified.
                    identified[neighbor] = true;
                    // Place it into the queue.
                    theQueue.offer(neighbor);
                    /** Insert the edge (current, neighbor)
                        into the tree by marking current as the
                        parent of neighbor
                    */
                    parent[neighbor] = current;
                }
            }
            // Finished visiting current.
        }
        // Finished the breadth-first traversal
        return parent;
    }
}
```

Depth-First Search

Another way to traverse a graph is depth-first search. In depth-first search, you start at a vertex, visit it, and choose one adjacent vertex to visit. Then choose a vertex adjacent to that vertex to visit, and so on until you go no further. Then back up and see whether a new vertex (one not previously visited) can be found. In the discussion that follows, we use color to distinguish among three states for a node: being visited (light color), finished visiting (dark color), and not yet visited (white). Initially, of course, all nodes are not yet visited. Note that the light color is used in depth-first search to indicate that a vertex is in the process of being visited, whereas it was used in our discussion of breadth-first search to indicate that the vertex was identified.

Example of Depth-First Search

Consider the graph shown in Figure 10.18. We can start at any vertex, but for simplicity we will start at 0. The vertices adjacent to 0 are 1, 2, 3, and 4. We mark 0 as being visited (change it to light color; see Figure 10.19(a)). Next we consider 1. We mark 1 as being visited (see Figure 10.19(b)). The vertices adjacent to 1 are 0, 3, and 4. But 0 is being visited, so we recursively apply the algorithm with 3 as the start vertex. We mark 3 as being visited (see Figure 10.19(c)). The vertices adjacent to 3 are 0, 1, and 4. Because 0 and 1 are already being visited, we recursively apply the algorithm with 4 as the start vertex. We mark 4 as being visited (see Figure 10.19(d)). The vertices adjacent to 4 are 0, 1, and 3. All of these are being visited, so we mark 4 as finished (see Figure 10.19(e)) and return from the recursion. Now all of the vertices adjacent to 3 have been visited, so we mark 3 as finished and return from the recursion. Now all of the vertices adjacent to 1 have been visited, so we mark 1 as finished and return from the recursion to the original start vertex, 0. The order in which we started to visit vertices is 0, 1, 3, 4; the order in which vertices have become finished so far is 4, 3, 1.

We now consider vertex 2, which is adjacent to 0 but has not been visited. We mark 2 as being visited (see Figure 10.19(f)) and consider the vertices adjacent to it: 5 and 6. We mark 5 as being visited (see Figure 10.19(g)) and consider the vertices adjacent to it: 2 and 6. Because 2 is already being visited, we next visit 6. We mark 6 as being visited (see Figure 10.19(h)). The vertices adjacent to 6 (2 and 5) are already being visited. Thus, we mark 6 as finished and recursively return. The vertices adjacent to 5 have all been visited, so we mark 5 as finished and return from the recursion. All of the vertices adjacent to 2 have been visited, so we mark 2 as finished and return from the recursion.

Finally, we come back to 0. Because all of the vertices adjacent to it have also been visited, we mark 0 as finished and we are done (see Figure 10.19(i)). The order in which we started to visit all vertices is 0, 1, 3, 4, 2, 5, 6; the order in which we finished visiting all vertices is 4, 3, 1, 6, 5, 2, 0. The *discovery order* is the order in which the vertices are discovered. The *finish order* is the order in which the vertices are finished. We consider a vertex to be finished when we return to it after finishing all its successors.

Figure 10.20 shows the depth-first search tree for the graph in Figure 10.18. A preorder traversal of this tree yields the sequence in which the vertices were visited: 0, 1, 3, 4, 2, 5, 6.

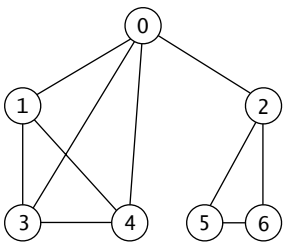


FIGURE 10.18
Graph to Be Traversed Depth First

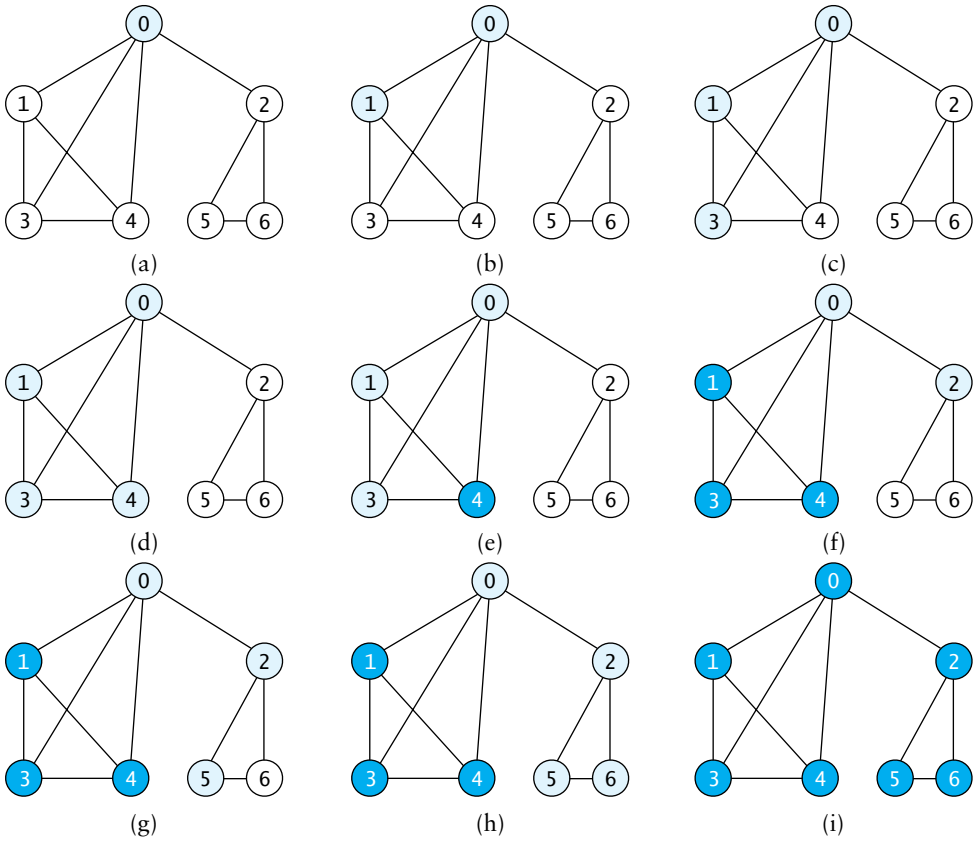


FIGURE 10.19
Example of Depth-First Search

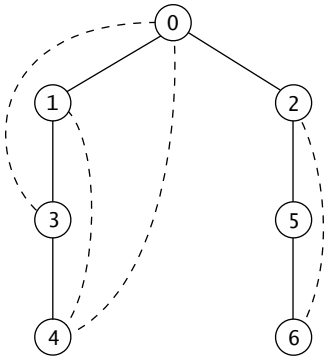


FIGURE 10.20
Depth-First Search Tree of Figure 10.18

The dashed lines are the other edges in the graph that are not part of the depth-first search tree. These edges are called *back edges* because they connect a vertex with its ancestors in the depth-first search tree. Observe that vertex 4 has two ancestors in addition to its parent 3: 1 and 0. Vertex 1 is a grandparent, and vertex 0 is a great-grandparent.

Algorithm for Depth-First Search

Depth-first search is used as the basis of other graph algorithms. However, rather than embedding the depth-first search algorithm into these other algorithms, we will implement the depth-first search algorithm to collect information about the vertices, which we can then use in these other algorithms. The information we will collect is the discovery order (or the visit order) and the finish order.

The depth-first search algorithm follows. Step 5 recursively applies this algorithm to each vertex as it is discovered.

Algorithm for Depth-First Search

1. Mark the current vertex, u, visited (change it to light color), and enter it in the discovery order list.
2. **for** each vertex, v, adjacent to the current vertex, u
3. **if** v has not been visited
4. Set parent of v to u.
5. Recursively apply this algorithm starting at v.
6. Mark u finished (change it to dark color) and enter u into the finish order list.

Observe that Step 6 is executed after the loop in Step 2 has examined all vertices adjacent to vertex u. Also, the loop at Step 2 does not select the vertices in any particular order.

Table 10.5 shows a trace of the algorithm as applied to the graph shown in Figure 10.19. We list each visit or finish step in column 1. Column 2 lists the vertices adjacent to each vertex when it begins to be visited. The discovery order (the order in which the vertices are visited) is 0, 1, 3, 4, 2, 5, 6. The finish order is 4, 3, 1, 6, 5, 2, and 0.

Performance Analysis of Depth-First Search

The loop at Step 2 is executed $|E_u|$ (the number of edges that originate at that vertex) times. The recursive call results in this loop being applied to each vertex. The total number of steps is the sum of the edges that originate at each vertex, which is the total number of edges $|E|$. Thus, the algorithm is $O(|E|)$.

There is an implicit Step 0 to the algorithm that colors all of the vertices white. This is $O(|V|)$; thus, the total running time of the algorithm is $O(|V|+|E|)$.

TABLE 10.5
Trace of Depth-First Search of Figure 10.19

Operation	Adjacent Vertices	Discovery (Visit) Order	Finish Order
Visit 0	1, 2, 3, 4	0	
Visit 1	0, 3, 4	0, 1	
Visit 3	0, 1, 4	0, 1, 3	
Visit 4	0, 1, 3	0, 1, 3, 4	
Finish 4			4
Finish 3			4, 3
Finish 1			4, 3, 1
Visit 2	0, 5, 6	0, 1, 3, 4, 2	
Visit 5	2, 6	0, 1, 3, 4, 2, 5	
Visit 6	2, 5	0, 1, 3, 4, 2, 5, 6	
Finish 6			4, 3, 1, 6
Finish 5			4, 3, 1, 6, 5
Finish 2			4, 3, 1, 6, 5, 2
Finish 0			4, 3, 1, 6, 5, 2, 0

Implementing Depth-First Search

The class `DepthFirstSearch` is designed to be used as a building block for other algorithms. When constructed, this class performs a depth-first search on a graph and records the start time, finish time, start order, and finish order. For an unconnected graph or for a directed graph (whether connected or not), a depth-first search may not visit each vertex in the graph. Thus, once the recursive method returns, the vertices need to be examined to see whether they all have been visited; if not, the recursive process repeats, starting with the next unvisited vertex. Thus, the depth-first search can generate more than one tree. We will call this collection of trees a *forest*. Also, it may be important that we control the order in which the vertices are examined to form the forest. Thus, one of the constructors for the `DepthFirstSearch` class enables its caller to specify the order in which vertices are examined to select a new start vertex. The default is normal ascending order. The class is described in Table 10.6, and part of the code is shown in Listing 10.3.

Each constructor allocates storage for the arrays `parent`, `visited`, `discoveryOrder`, and `finishOrder` and initializes all elements of `parent` to `-1 (no parent)`. In the constructor in Listing 10.3, the **`for`** statement

```
for (int i = 0; i < n; i++) {
    if (!visited[i])
       depthFirstSearch(i);
}
```

TABLE 10.6
Class `DepthFirstSearch`

Data Field	Attribute
`private int discoverIndex`	The index that indicates the discovery order of a vertex
`private int[] discoveryOrder`	The array that contains the vertices in discovery order
`private int finishIndex`	The index that indicates the finish order of a vertex
`private int[] finishOrder`	The array that contains the vertices in finish order
`private Graph graph`	A reference to the graph being searched
`private int[] parent`	The array of predecessors in the depth-first search tree
`private boolean[] visited`	An array of **`boolean`** values to indicate whether or not a vertex has been visited
Constructor	**Purpose**
`public DepthFirstSearch(Graph graph)`	Constructs the depth-first search of the specified graph selecting the start vertices in ascending vertex order
`public DepthFirstSearch(Graph graph, int[] order)`	Constructs the depth-first search of the specified graph selecting the start vertices in the specified order. The first vertex visited is `order[0]`
Method	**Behavior**
`public void depthFirstSearch(int s)`	Recursively searches the graph starting at vertex `s`
`public int[]getDiscoveryOrder()`	Gets the discovery order
`public int[] getFinishOrder()`	Gets the finish order
`public int[] getParent()`	Gets the parents in the depth-first search tree

calls the recursive depth-first search method. Method `depthFirstSearch` follows the algorithm shown earlier. If the graph is connected, all vertices will be visited after the return from the initial call to `depthFirstSearch`. If the graph is not connected, additional calls will be made using a start vertex that has not been visited.

In the constructor (not shown) that allows the client to control the order of selection for start vertices, the parameter `int[] order` specifies this sequence. To code this constructor, change the **`if`** statement in the **`for`** loop of the first constructor to

```
if (!visited[order[i]])
   depthFirstSearch(order[i]);
```

The rest of the code is the same.

LISTING 10.3

DepthFirstSearch.java

```
/** Class to implement the depth-first search algorithm. */
public class DepthFirstSearch {

    // Data Fields
    /** A reference to the graph being searched. */
    private Graph graph;
    /** Array of parents in the depth-first search tree. */
    private int[] parent;
    /** Flag to indicate whether this vertex has been visited. */
    private boolean[] visited;
    /** The array that contains each vertex in discovery order. */
    private int[] discoveryOrder;
    /** The array that contains each vertex in finish order. */
    private int[] finishOrder;
    /** The index that indicates the discovery order. */
    private int discoverIndex = 0;
    /** The index that indicates the finish order. */
    private int finishIndex = 0;

    // Constructors
    /** Construct the depth-first search of a Graph starting at
        vertex 0 and visiting the start vertices in ascending order.
        @param graph The graph
     */
    public DepthFirstSearch(Graph graph) {
        this.graph = graph;
        int n = graph.getNumV();
        parent = new int[n];
        visited = new boolean[n];
        discoveryOrder = new int[n];
        finishOrder = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = -1;
        }
        for (int i = 0; i < n; i++) {
            if (!visited[i])
                depthFirstSearch(i);
        }
    }

    /** Construct the depth-first search of a graph
        selecting the start vertices in the specified order.
        The first vertex visited is order[0].
```

```
        @param graph The graph
        @param order The array giving the order
                     in which the start vertices should be selected
     */
    public DepthFirstSearch(Graph graph, int[] order) {
        // Same as constructor above except for the if statement.
    }

    /** Recursively depth-first search the graph starting at vertex current.
        @param current The start vertex
     */
    public void depthFirstSearch(int current) {
        /* Mark the current vertex visited. */
        visited[current] = true;
        discoveryOrder[discoverIndex++] = current;
        /* Examine each vertex adjacent to the current vertex */
        Iterator<Edge> itr = graph.edgeIterator(current);
        while (itr.hasNext()) {
            int neighbor = itr.next().getDest();
            /* Process a neighbor that has not been visited */
            if (!visited[neighbor]) {
                /* Insert (current, neighbor) into the depth-first search tree. */
                parent[neighbor] = current;
                /* Recursively apply the algorithm starting at neighbor. */
                depthFirstSearch(neighbor);
            }
        }
        /* Mark current finished. */
        finishOrder[finishIndex++] = current;
    }
}
```

Testing Class `DepthFirstSearch`

Next, we show a `main` method that tests the class. It is a simple driver program that can be used to read a graph and then initiate a depth-first traversal. After the traversal, the driver program displays the arrays that represent the search results.

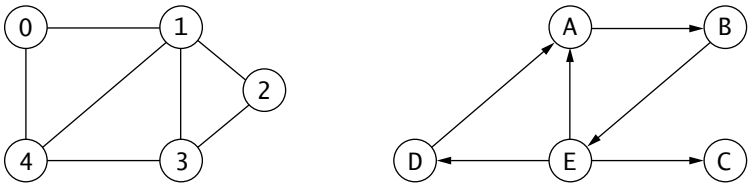
```
    /** Main method to test depth-first search
        @pre args[0] is the name of the input file.
        @param args The command line arguments
     */
    public static void main(String[] args) {
        Graph g = null;
        int n = 0;
        try (var scan = new Scanner(new File(args[0]))){
            n = scan.nextInt();
            g = new ListGraph(n, true);
            g.loadEdgesFromFile(scan);
        } catch (IOException ex) {
            ex.printStackTrace();
            System.exit(1);  // Error
        }

        // Perform depth-first search.
        DepthFirstSearch dfs = new DepthFirstSearch(g);
        int[] dOrder = dfs.getDiscoveryOrder();
```

```
        int[] fOrder = dfs.getFinishOrder();
        System.out.println("Discovery and finish order");
        for (int i = 0; i < n; i++) {
            System.out.println(dOrder[i] + " " + fOrder[i]);
        }
    }
```

EXERCISES FOR SECTION 10.4

SELF-CHECK

1. Show the breadth-first search trees for the following graphs.



2. Show the depth-first search trees for the graphs in Exercise 1 above.

PROGRAMMING

1. Provide all accessor methods for class `DepthFirstSearch` and the constructor that specifies the order of start vertices.
2. Implement method `depthFirstSearch` without using recursion. *Hint:* Use a stack to save the parent of the current vertex when you start to search one of its adjacent vertices.

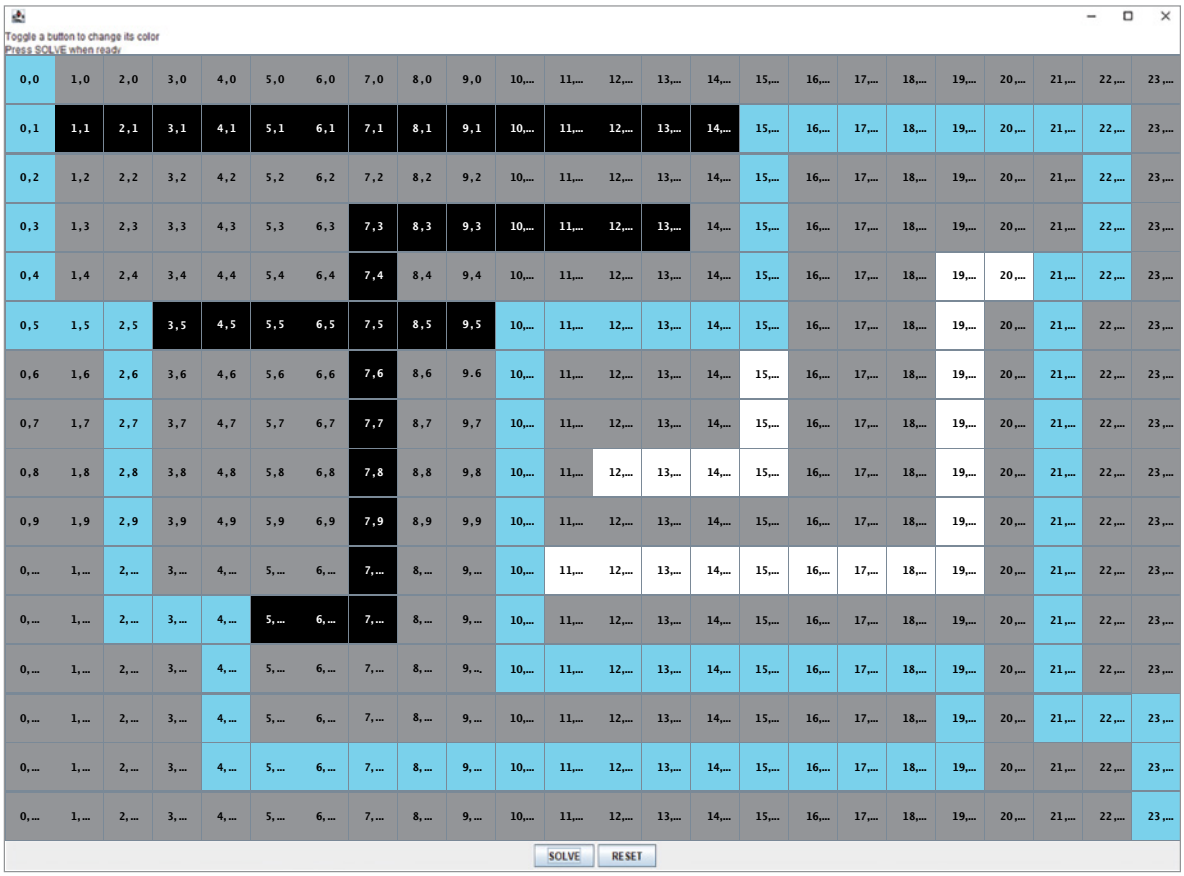
10.5 Applications of Graph Traversals

CASE STUDY Shortest Path through a Maze

Problem We want to design a program that will find the shortest path through a maze. In Chapter 5, we showed how to write a recursive program that found a solution to a maze. This program used a backtracking algorithm that visited alternate paths. When it found a dead end, it backed up and tried another path, and eventually it found a solution.

Figure 10.21 shows a maze solution generated by the backtracking program. The light-gray cells are barriers in the maze. All other squares were originally white to show that they are open squares. The color squares show the solution path, the black squares show the squares that were visited but rejected, and the squares that remain white are open squares that were not visited.

FIGURE 10.21
Backtracking Solution to Finding a Path through a Maze



As you can see, the backtracking program found a rather circuitous route from the start point to the end point, which was not an optimal solution. This is a consequence of the program advancing the solution path to the south at the cell in column 0, row 1 (white cell 1, 0) before attempting to advance it to the east.

Our goal is to find the shortest path, defined as the one with the fewest decision points in it. We will use the breadth-first algorithm to do this.

Analysis

We can represent the maze shown in Figure 10.21 by a graph, where we place a node at each decision point and at each dead end, as shown in Figure 10.22.

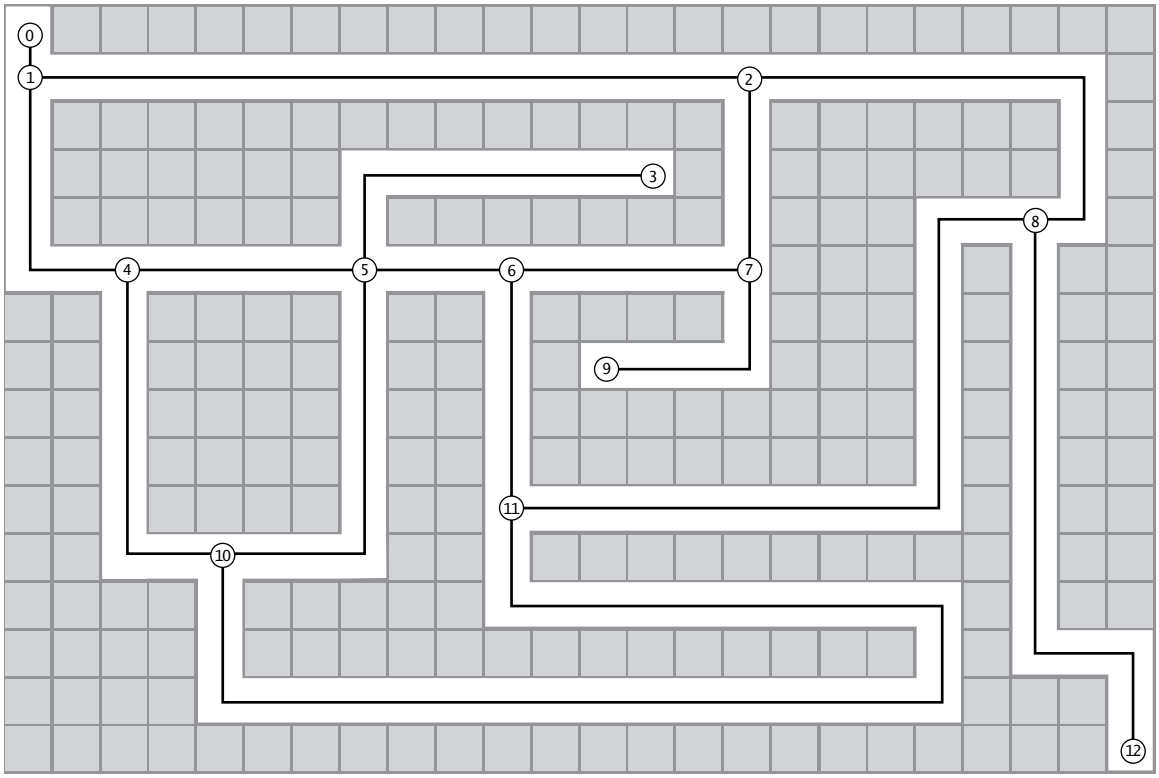
Now that we have the maze represented as a graph, we need to find the shortest path from the start point (vertex 0) to the end point (vertex 12). The breadth-first search method will return the shortest path from each vertex to its parent (the array of parent vertices), and we can use this array to find the shortest path to the end point.

Design

Your program will need the following data structures:

- An external representation of the maze, consisting of the number of vertices and the edges.
- An object of a class that implements the `Graph` interface.

FIGURE 10.22
Graph Representation of the Maze in Figure 10.21



- An array to hold the predecessors returned from the `breadthFirstSearch` method.
- A stack to reverse the path.

The algorithm is as follows:

1. Read in the number of vertices and create the graph object.
2. Read in the edges and insert the edges into the graph.
3. Call the `breadthFirstSearch` method with this graph and the starting vertex as its argument. The method returns the array `parent`.
4. Start at v, the end vertex.
5. **`while`** v is not –1
6. Push v onto the stack.
7. Set v to `parent`[v].
8. **`while`** the stack is not empty
9. Pop a vertex off the stack and output it.

Implementation

Listing 10.4 shows the program. We assume that the graph that represents the maze is stored in a text file. The first line of this file contains the number of vertices. The edges are on subsequent lines. The method `loadEdgesFromFile` reads the source and destination vertices and inserts the edge into the graph. The rest of the code follows the algorithm.

LISTING 10.4

Program to Solve a Maze Using a Breadth-First Search

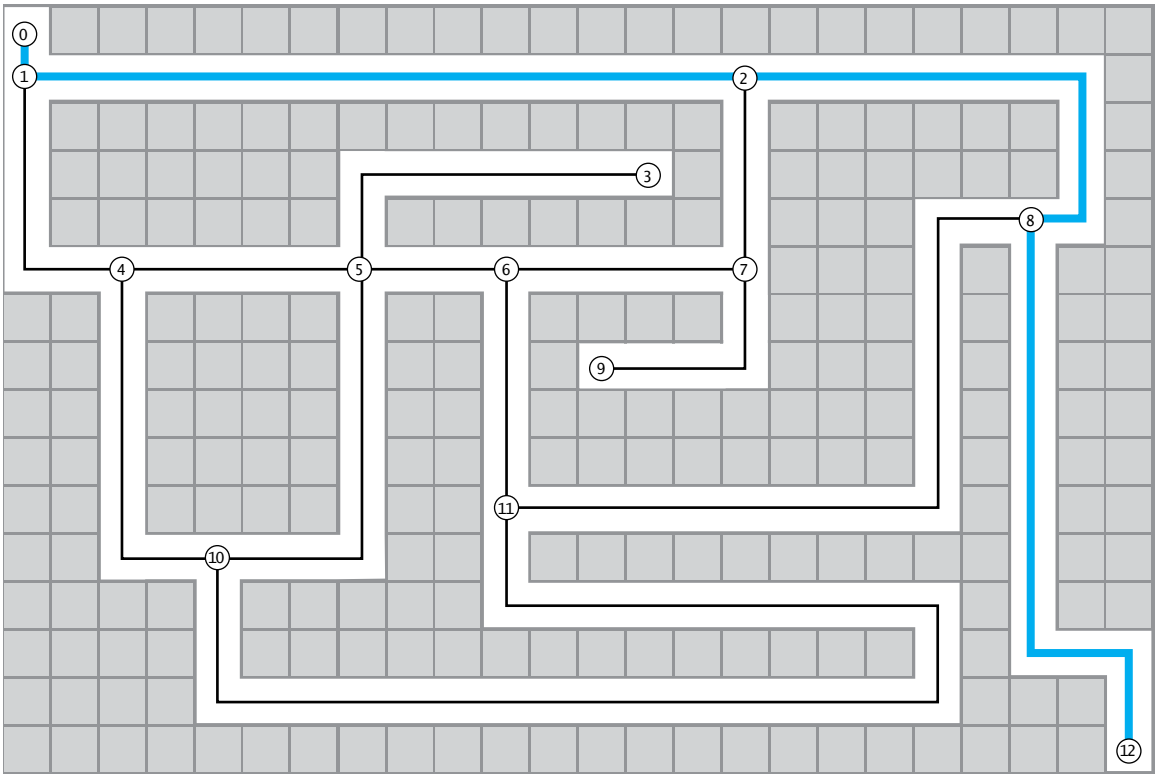
```
import java.io.*;
import java.util.*;

/** Program to solve a maze represented as a graph.
    This program performs a breadth-first search of the graph
    to find the "shortest" path from the start vertex to the
    end. It is assumed that the start vertex is 0, and the
    end vertex is numV-1.
 */
public class Maze {

    /** Main method to solve the maze.
        pre: args[0] contains the name of the input file.
        @param args Command line argument
     */
    public static void main(String[] args) {
        int numV = 0;
        // The number of vertices.
        Graph theMaze = null;
        // Load the graph data from a file.
        try (var scan = new Scanner(new File(args[0]))){
            numV = scan.nextInt();
            theMaze = new ListGraph(numV, false);
            theMaze.loadEdgesFromFile(scan);
        } catch (IOException ex) {
            System.err.println("IO Error while reading graph");
            System.err.println(ex.toString());
            System.exit(1);
        }
        // Perform breadth-first search.
        int parent[] = BreadthFirstSearch.breadthFirstSearch(theMaze, 0);
        // Construct the path.
        Deque<Integer> thePath = new ArrayDeque<>();
        int v = numV - 1;
        // Trace backwards from destination until source is reached
        while (v != -1) {
            thePath.push(v);
            v = parent[v];
        }
        // Output the path.
        System.out.println("The Shortest path is:");
        while (!thePath.isEmpty()) {
            System.out.println(thePath.pop());
        }
    }
}
```

Testing

Test this program with a variety of mazes. Use mazes for which the backtracking program finds the shortest path and mazes for which it does not. For the graph shown in Figure 10.23, the shortest path from 0 to 12 is $0 \rightarrow 1 \rightarrow 2 \rightarrow 8 \rightarrow 12$.

It so happens that this path is also the shortest path based on the number of cells. But this would not always be the case. To find the shortest path based on the number of cells, we would need to use a weighted graph as discussed in Section 10.6.

FIGURE 10.23
Solution to Maze in Figure 10.21



CASE STUDY Topological Sort of a Graph

Problem There are many problems in which one activity cannot be started before another one has been completed. One that you may have already encountered is determining the order in which you can take courses. Some courses have prerequisites. Some have more than one prerequisite. Furthermore, the prerequisites may have prerequisites. Figure 10.24 shows the courses and prerequisites of a Computer Science program at the authors' university.

Graphs such as the one shown in Figure 10.24 are known as *directed acyclic graphs* (*DAGs*). They are directed graphs that contain no cycles; that is, there are no loops, so

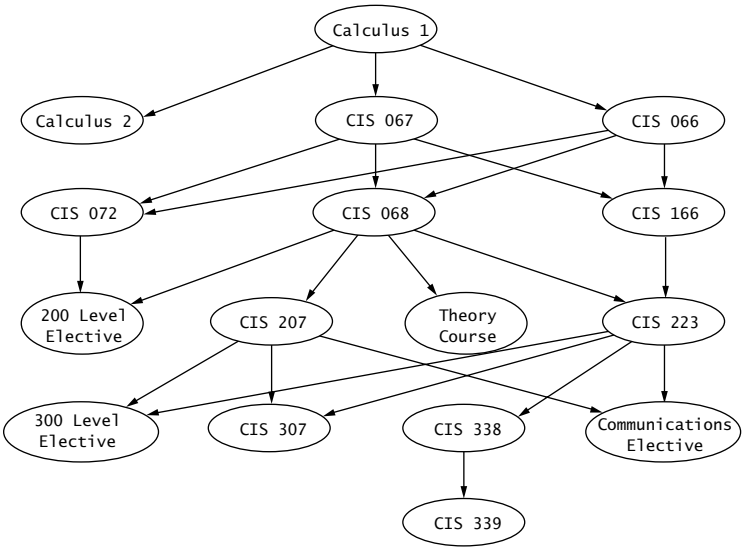


FIGURE 10.24
Prerequisites for a Computer Science Program

once you pass through a vertex, there is no path back to that vertex. Figure 10.25 shows another example of a DAG.

A *topological sort* of the vertices of a DAG is an ordering of the vertices such that if (u, v) is an edge, then u appears before v. This must be true for all edges. For example, 0, 1, 2, 3, 4, 5, 6, 7, 8 is a valid topological sort of the graph in Figure 10.25, but 0, 1, 5, 3, 4, 2, 6, 7, 8 is not because $2 \rightarrow 5$ is an edge, but 5 appears before 2. There are many valid paths through the prerequisite graph and many valid topological sorts. Another valid topological sort is 0, 3, 1, 4, 6, 2, 5, 7, 8.

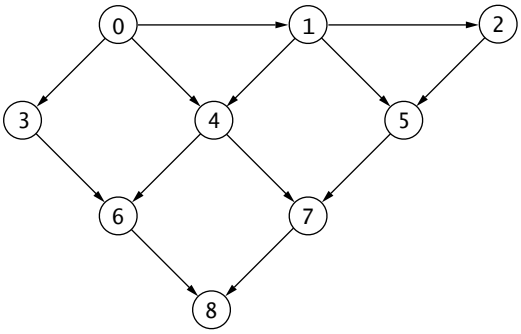


FIGURE 10.25
Example of a DAG

Analysis

If there is an edge from u to v in a DAG, then if we perform a depth-first search of this graph, the finish time of u must be after the finish time of v. When we return to u, either v has not been visited or it has finished. It is not possible that v would be visited but not finished, because if it were possible, we would discover u on a path that had passed through v. That would mean that there is a loop or cycle in the graph.

For example, in Figure 10.25, we could start the depth-first search at 0, then visit 4, followed by 6, followed by 8. Then, returning to 4, we would have to visit 7 before returning to 0. Then we would visit 1, and from 1 we would see that 4 has finished. Alternatively, we could

start at 0 and then go to 1, and we would see that 4 has not been visited. What we cannot have happen is that we start at 0, then visit 4, and eventually get to 1 before finishing 4.

Design

If we perform a depth-first search of a graph and then order the vertices by the inverse of their finish order, we will have one topological sort of a DAG. The topological sort produced by listing the vertices in the reverse of their finish order after a depth-first search of the graph in Figure 10.25 is 0, 3, 1, 4, 6, 2, 5, 7, 8.

Algorithm for Topological Sort

1. Read the graph from a data file.
2. Perform a depth-first search of the graph.
3. List the vertices in reverse of their finish order.

Implementation

We can use our `DepthFirstSearch` class to implement this algorithm. Listing 10.5 shows a program that does this. It begins by reading the graph from an input file. It then creates a `DepthFirstSearch` object `dfs`. The constructor of the `DepthFirstSearch` class performs the depth-first search and saves information about the graph. We then call the `getFinishOrder` method to get the vertices in the order in which they finished. If we output this array starting at `numVertices - 1`, we will obtain the topological sort of the graph.

LISTING 10.5

TopologicalSort.java

```
import java.util.*;

/** This program outputs the topological sort of a directed graph
    that contains no cycles.
 */
public class TopologicalSort {

    /** The main method that performs the topological sort.
        pre: arg[0] contains the name of the file
             that contains the graph. It has no cycles.
        @param args The command line arguments
     */
    public static void main(String[] args) {
        Graph theGraph = null;
        int numVertices = 0;
        // Connect a Scanner to the input file
        try (var scan = new Scanner(new File(args[0]))) {
            // Load the graph data from a file.
            numVertices = scan.nextInt();
            theGraph = new ListGraph(numVertices, true);
            theMaze.loadEdgesFromFile(scan);
        } catch (Exception ex) {
            ex.printStackTrace();
            System.exit(1);
            // Error exit
        }

        // Perform the depth-first search.
        DepthFirstSearch dfs = new DepthFirstSearch(theGraph);
        // Obtain the finish order
        int[] finishOrder = dfs.getFinishOrder();
```

```
            // Print the vertices in reverse finish order.
            System.out.println("The Topological Sort is");
            for (int i = numVertices - 1; i >= 0; i--) {
                System.out.println(finishOrder[i]);
            }
        }
    }
```

Testing Test this program using several different graphs. Use sparse graphs and dense graphs. Make sure that each graph you try has no loops or cycles. If it does, the algorithm may display an invalid output.

EXERCISES FOR SECTION 10.5

SELF-CHECK

1. Draw the depth-first search tree of the graph in Figure 10.24 and then list the vertices in reverse finish order.
2. List some alternative topological sorts for the graph in Figure 10.24.

10.6 Algorithms Using Weighted Graphs

Finding the Shortest Path from a Vertex to All Other Vertices

The breadth-first search discussed in Section 10.4 found the shortest path from the start vertex to a destination vertex, assuming that the length of each edge was the same. We now consider the problem of finding the shortest path from a start vertex to all other vertices where the length of each edge may be different—that is, in a weighted directed graph such as that shown in Figure 10.26. The computer scientist Edsger W. Dijkstra developed an algorithm, now called Dijkstra's algorithm ("A Note on Two Problems in Connection with Graphs," *Numerische Mathematik*, Vol. 1 [1959], pp. 269–271), to solve this problem. This algorithm assumes that all of the edge values are positive and that there is a path from the start vertex to all other vertices.

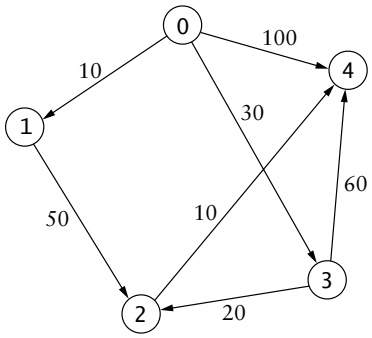


FIGURE 10.26 Weighted Directed Graph

For Dijkstra's algorithm, we need two sets, S and $V–S$, and two arrays, `d` and `p`. Set S will contain the vertices for which we have computed the shortest distance, and $V–S$ will contain the vertices that we still need to process. The entry `d`[v] will contain the shortest distance from s to v, and `p`[v] will contain the predecessor of v in the path from s to v.

We initialize S by placing the start vertex, s, into it. We initialize $V–S$ by placing the remaining vertices into it. For each v in $V–S$, we initialize `d` by setting `d`[v] equal to the weight of the edge $w(s, v)$ for each vertex, v, adjacent to s and to ∞ for each vertex that is not adjacent to s. We initialize `p`[v] to s for each v in $V–S$.

For example, given the graph shown in Figure 10.26, the set S would initially be {0}, and $V–S$ would be {1, 2, 3, 4}. The arrays `d` and `p` would be defined as follows.

v	d[v]	p[v]
1	10	0
2	∞	0
3	30	0
4	100	0

The first row shows that the distance from vertex 0 to vertex 1 is 10 and that vertex 0 is the predecessor of vertex 1. The second row shows that vertex 2 is not adjacent to vertex 0.

We now find the vertex u in $V–S$ that has the smallest value of `d`[u]. Using our example, this is 1. We now consider the vertices v that are adjacent to u. If the distance from s to u (`d`[u]) plus the distance from u to v (i.e., $w(u, v)$) is smaller than the known distance from s to v, `d`[v], then we update `d`[v] to be `d`[u] + $w(u, v)$, and we set `p`[v] to u. In our example, the value of `d`[1] is 10, and $w(1, 2)$ is 50. Since 10 + 50 = 60 is less than ∞, we set `d`[2] to 60 and `p`[2] to 1. We remove 1 from $V–S$ and place it into S. We repeat this until $V–S$ is empty.

After the first pass through this loop, S is {0, 1}, $V–S$ is {2, 3, 4}, and `d` and `p` are as follows:

v	d[v]	p[v]
1	10	0
2	60	1
3	30	0
4	100	0

We again select u from $V–S$ with the smallest `d`[u]. This is now 3. The adjacent vertices to 3 are 2 and 4. The distance from 0 to 3, `d`[3], is 30. The distance from 3 to 2 is 20. Because 30 + 20 = 50 is less than the current value of `d`[2], 60, we update `d`[2] to 50 and change `p`[2] to 3. Also, because 30 + 60 = 90 is less than 100, we update `d`[4] to 90 and set `p`[4] to 3.

Now S is {0, 1, 3}, and $V–S$ is {2, 4}. The arrays `d` and `p` are as follows:

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	90	3

Next, we select vertex 2 from V–S. The only vertex adjacent to 2 is 4. Since d[2] + $w(2, 4)$ = 50 + 10 = 60 is less than d[4], 90, we update d[4] to 60 and p[4] to 2. Now S is {0, 1, 2, 3}, V–S is {4}, and d and p are as follows:

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	60	2

Finally, we remove 4 from V–S and find that it has no adjacent vertices. We are now done. The array d shows the shortest distances from the start vertex to all other vertices, and the array p can be used to determine the corresponding paths. For example, the path from vertex 0 to vertex 4 has a length of 60, and it is the reverse of 4, 2, 3, 0; therefore, the shortest path is 0 → 3 → 2 → 4.

Dijkstra's Algorithm

1. Initialize S with the start vertex, s, and V–S with the remaining vertices.
2. **for** all v in V–S
3. Set p[v] to s.
4. **if** there is an edge (s, v)
5. Set d[v] to $w(s, v)$.
   else
6. Set d[v] to ∞.
7. **while** V–S is not empty
8. **for** all u in V–S, find the vertex u with the smallest d[u].
9. Remove u from V–S and add u to S.
10. **for** all v adjacent to u in V–S
11. **if** d[u] + $w(u, v)$ is less than d[v]
12. Set d[v] to d[u] + $w(u, v)$.
13. Set p[v] to u.

Analysis of Dijkstra's Algorithm

Step 1 requires $|V|$ steps.

The loop at Step 2 will be executed $|V - 1|$ times.

The loop at Step 7 will also be executed $|V - 1|$ times.

Within the loop at Step 7, we must consider Steps 8 and 9. For these steps, we will have to search each value in V–S. This decreases each time through the loop at Step 7, so we will have $|V| - 1 + |V| - 2 + \cdots 1$. This is $O(|V|^2)$. Therefore, Dijkstra's algorithm as stated is $O(|V|^2)$. We will look at possible improvements to this for sparse graphs when we discuss a similar algorithm in the next subsection.

Implementation

Listing 10.6 provides a straightforward implementation of Dijkstra's algorithm using `HashSet vMinusS` to represent set V–S. We chose to implement the algorithm as a **`static`** method with the inputs (the graph and starting point) and outputs (predecessor and distance array)

passed through parameters. We use iterators to traverse `vMinusS`, and the `edgeIterator` to traverse the edges adjacent to a node being added to set `S`.

LISTING 10.6

Dijkstra's Shortest-Path Algorithm

```
/** Dijkstra's Shortest-Path algorithm.
    @param graph The weighted graph to be searched
    @param start The start vertex
    @param pred Output array to contain the predecessors in the shortest path
    @param dist Output array to contain the distance in the shortest path
 */
public static void dijkstrasAlgorithm(Graph graph, int start, int[] pred,
                                       double[] dist) {
    int numV = graph.getNumV();
    HashSet<Integer> vMinusS = new HashSet<>(numV);
    // Initialize V-S.
    for (int i = 0; i < numV; i++) {
        if (i != start) {
            vMinusS.add(i);
        }
    }
    // Initialize pred and dist.
    pred[start] = -1;
    dist[start] = 0;
    for (int v : vMinusS) {
        pred[v] = start;
        dist[v] = graph.getEdge(start, v).getWeight();
    }

    // Main loop
    while (vMinusS.size() != 0) {
        // Find the vertex u in V-S with the smallest dist[u].
        double minDist = Double.POSITIVE_INFINITY;
        int u = -1;
        for (int v : vMinusS) {
            if (dist[v] < minDist) {
                minDist = dist[v];
                u = v;
            }
        }
        // Remove u from vMinusS.
        vMinusS.remove(u);
        // Update the distances.
        var itr = graph.edgeIterator(u);
        while (itr.hasNext()) {
            Edge e = itr.next();
            int v = e.getDest();
            if (vMinusS.contains(v)) {
                double weight = graph.getEdge(u, v).getWeight();
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pred[v] = u;
                }
            }
        }
    }
}
```

Minimum Spanning Trees

A *spanning tree* is a subset of the edges of a graph such that there is only one edge between each vertex, and all the vertices are connected. If we have a spanning tree for a graph, then we can access all the vertices of the graph from the start node. The *cost of a spanning tree* is the sum of the weights of the edges. We want to find the *minimum spanning tree* or the spanning tree with the smallest cost. For example, if we want to start up our own long-distance phone company and need to connect the cities shown in Figure 10.4, finding the minimum spanning tree would allow us to build the cheapest network.

We will discuss the algorithm published by R. C. Prim ("Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal*, Vol. 36 [1957], pp. 1389–1401) for finding the minimum spanning tree of a graph. It is very similar to Dijkstra's algorithm, but Prim published his algorithm in 1957, 2 years before Dijkstra's paper that contains an algorithm for finding the minimum spanning tree that is essentially the same as Prim's as well as the previously discussed algorithm for finding the shortest paths.

Overview of Prim's Algorithm

The vertices are divided into two sets: S, the set of vertices in the spanning tree, and V–S, the remaining vertices. As in Dijkstra's algorithm, we maintain two arrays: d[v] will contain the length of the shortest edge from a vertex in S to the vertex v that is in V–S, and p[v] will contain the source vertex for that edge. The only difference between the algorithm to find the shortest path and the algorithm to find the minimum spanning tree is the contents of d[v]. In the algorithm to find the shortest path, d[v] contains the total length of the path from the starting vertex. In the algorithm to find the minimum spanning tree, d[v] contains only the length of the final edge. We show the essentials of Prim's algorithm next.

Prim's Algorithm for Finding the Minimum Spanning Tree

1. Initialize S with the start vertex, s, and V–S with the remaining vertices.
2. **for** all v in V–S
3. Set p[v] to s.
4. **if** there is an edge (s, v)
5. Set d[v] to $w(s, v)$.
   else
6. Set d[v] to ∞.
7. **while** V–S is not empty
8. **for** all u in V–S, find the smallest d[u].
9. Remove u from V–S and add it to S.
10. Insert the edge (u, p[u]) into the spanning tree.
11. **for** all v in V–S
12. **if** $w(u, v)$ < d[v]
13. Set d[v] to $w(u, v)$.
14. Set p[v] to u.

In array d, d[v] contains the length of the shortest known (previously examined) edge from a vertex in S to the vertex v, while v is a member of V–S. In array p, the value p[v] is the source vertex of this shortest edge. When v is removed from V–S, we no longer update these entries in d and p.

EXAMPLE 10.2 Consider the graph shown in Figure 10.27. We initialize S to {0} and $V–S$ to {1, 2, 3, 4, 5}. The smallest edge from u to v, where u is in S and v is in $V–S$, is the edge (0, 2). We add this edge to the spanning tree and add 2 to S (see Figure 10.28(a)). The set S is now {0, 2} and $V–S$ is {1, 3, 4, 5}. We now have to consider all of the edges (u, v), where u is either 0 or 2, and v is 1, 3, 4, or 5 (there are eight possible edges). The smallest one is (2, 5). We add this to the spanning tree, and S now is {0, 2, 5} and $V–S$ is {1, 3, 4} (see Figure 10.28(b)). The next smallest edge is (5, 3). We insert that into the tree and add 3 to S (see Figure 10.28(c)). Now $V–S$ is {1, 4}. The smallest edge is (2, 1). After adding this edge (see Figure 10.28(d)), we are left with $V–S$ being {4}. The smallest edge to 4 is (1, 4). This is added to the tree, and the spanning tree is complete (see Figure 10.28(e)).

FIGURE 10.27
Graph for Example 10.2

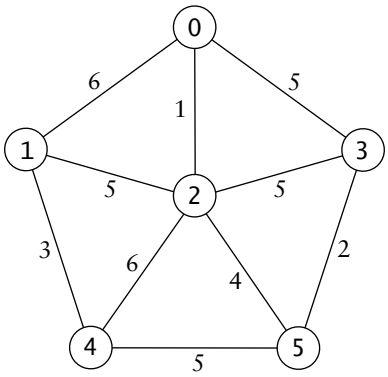
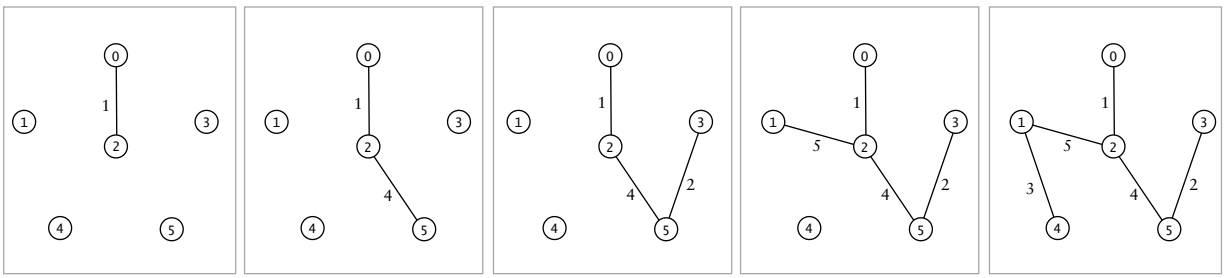


FIGURE 10.28
Building a Minimum Spanning Tree Using Prim's Algorithm



Analysis of Prim's Algorithm

Step 8 is O(|V|). Because this is within the loop at Step 7, it will be executed O(|V|) times for a total time of O(|V|^2). Step 11 is O(|Eu|), the number of edges that originate at u. Because Step 11 is inside the loop of Step 7, it will be executed for all vertices; thus, the total is O(|E|). Because |V|^2 is greater than |E|, the overall cost of the algorithm is O(|V|^2).

By using a priority queue to hold the edges from S to $V–S$, we can improve on this algorithm. Then Step 8 is O(log n), where n is the size of the priority queue. In the worst case, all of the edges are inserted into the priority queue, and the overall cost of the algorithm is then

$O(|E| \log |V|)$. We say that the algorithm is $O(|E| \log |V|)$ instead of saying that it is $O(|E| \log |E|)$, even though the maximum size of the priority queue is $|E|$, because $|E|$ is bounded by $|V|^2$ and $\log |V|^2$ is $2 \times \log |V|$.

For a dense graph, where $|E|$ is approximately $|V|^2$, this is not an improvement; however, for a sparse graph, where $|E|$ is significantly less than $|V|^2$, it is. Furthermore, computer science researchers have developed improved priority queue implementations that give $O(|E| + |V| \log |V|)$ or better performance.

Implementation

Listing 10.7 shows an implementation of Prim's algorithm using a priority queue to hold the edges from *S* to *V*–*S*. The arrays `p` and `d` given in the algorithm description above are not needed because the priority queue contains complete edges. For a given vertex `d`, if a shorter edge is discovered, we do not remove the entry containing the longer edge from the priority queue. We merely insert new edges as they are discovered. Therefore, when the next shortest edge is removed from the priority queue, it may have a destination that is no longer in *V*–*S*. In that case, we continue to remove edges from the priority queue until we find one with a destination that is still in *V*–*S*. This is done with the following loop:

```
do {
    edge = pQ.remove();
    dest = edge.getDest();
} while(!vMinusS.contains(dest));
```

LISTING 10.7

Prim's Minimum Spanning Tree Algorithm

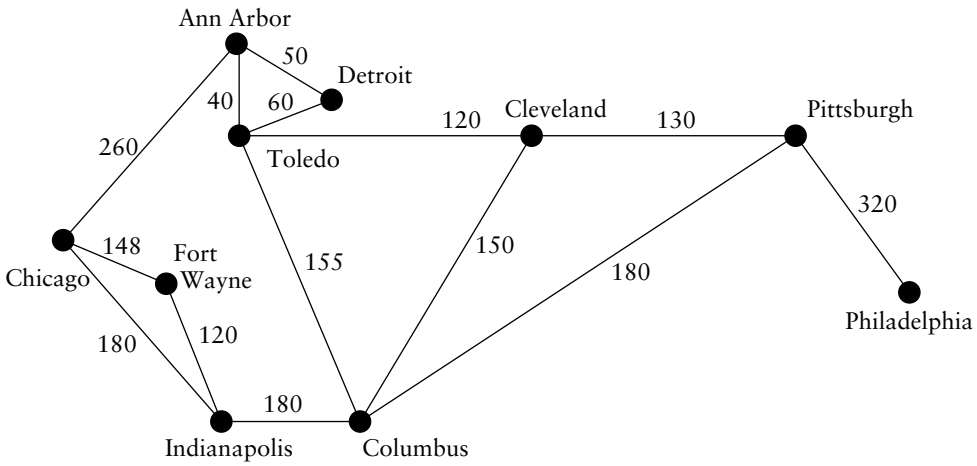
```
/** Prim's Minimum Spanning Tree (MST) algorithm.
    @param graph The weighted graph to be searched
    @param start The start vertex
    @return An ArrayList of edges that forms the MST
 */
public static ArrayList<Edge> primsAlgorithm(Graph graph, int start) {
    ArrayList<Edge> result = new ArrayList<>();
    int numV = graph.getNumV();
    // Use a HashSet to represent V-S.
    Set<Integer> vMinusS = new HashSet<>(numV);
    // Declare the priority queue.
    Queue<Edge> pQ = new PriorityQueue<>(numV,
            (e1, e2) -> Double.compare(e1.getWeight(), e2.getWeight()));
    // Initialize V-S.
    for (int i = 0; i < numV; i++) {
        if (i != start) {
            vMinusS.add(i);
        }
    }
    int current = start;
    // Main loop
    while (vMinusS.size() != 0) {
        // Update priority queue.
        var iter = graph.edgeIterator(current);
        while (iter.hasNext()) {
            Edge edge = iter.next();
            int dest = edge.getDest();
            if (vMinusS.contains(dest)) {
                pQ.add(edge);
            }
        }
```

```
            // Find the shortest edge whose source is in S and
            // destination is in V-S.
            int dest = -1;
            Edge edge = null;
            do {
                edge = pQ.remove();
                dest = edge.getDest();
            } while(!vMinusS.contains(dest));
            // Take dest out of vMinusS.
            vMinusS.remove(dest);
            // Add edge to result.
            result.add(edge);
            // Make this the current vertex.
            current = dest;
        }
        return result;
    }
```

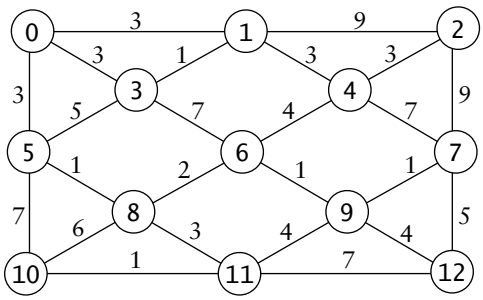
EXERCISES FOR SECTION 10.6

SELF-CHECK

1. Trace the execution of Dijkstra's algorithm to find the shortest path from Philadelphia to the other cities shown in the following graph.



2. Trace the execution of Dijkstra's algorithm to find the shortest paths from vertex 0 to the other vertices in the following graph.



3. Trace the execution of Prim's algorithm to find the minimum spanning tree for the graph shown in Exercise 2.
4. Trace the execution of Prim's algorithm to find the minimum spanning tree for the graph shown in Exercise 1.

10.7 A Heuristic Algorithm A* to Find the Best Path

In the previous section, we saw how Dijkstra's algorithm can be used to find the shortest path from a given start vertex to all other vertices in a graph. GPS and mapping software can be used to find the shortest (fastest) path from a given start position to a destination. The mapping database is a graph with many vertices and edges. If we were to use Dijkstra's algorithm, we would find the best path to the destination but also find paths to vertices that are not at all close to the desired destination.

Figure 10.29 is a weighted graph that shows the actual distance traveled on roads between adjacent vertices in black alongside the edge connecting these vertices. The shortest path from Philadelphia (the start) to Chicago (the destination) is shown in color. We will explain what the numbers in color under each vertex represent shortly.

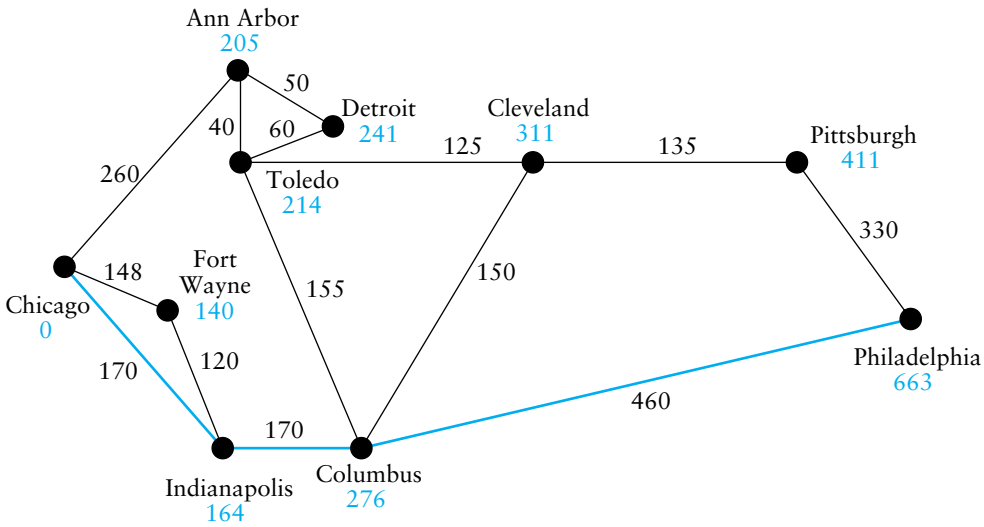


FIGURE 10.29
Path from Philadelphia to Chicago

Dijkstra's algorithm explores the graph in a modified breadth-first search in which the vertex next visited is the one that is closest to the starting point. Following Dijkstra's algorithm, we first visit Pittsburgh and determine the distance from Philadelphia to Cleveland through Pittsburgh 465. Next, we visit Columbus and determine the distance from Philadelphia to Indianapolis through Columbus is 630. Since Cleveland is the next vertex closest to Philadelphia, we visit it and determine the distance from Philadelphia to Toledo is 590. Toledo is then visited and we determine the distances to Ann Arbor and Detroit are 630 and 650, respectively. Now we visit Indianapolis and determine the distance to Chicago from Philadelphia is 800 and to Ft. Wayne is 750. Then, Ann Arbor, Detroit, and Ft. Wayne are visited, but do not provide a shorter path to Chicago than what was already found (800 going through Columbus and Indianapolis as shown in color in Figure 10.29).

A* (A-Star) an Improvement of Dijkstra's Algorithm

The A* algorithm was developed by Peter Hart, Nills Nilsson, and Bertram Raphael in 1968. (Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics*. 4 (2): 100–1070). It was developed to enable a robot to find the shortest path to a goal with obstacles between it and the goal. The A* algorithm is significant because it is one of the first examples of a *tree-pruning algorithm*, which is one of the staples of artificial intelligence. Pruning a search tree enables a program to focus on the more likely solution paths, ignoring those that are not productive, and can save considerable computational time. Chess-playing programs that compete with humans depend on tree pruning heuristics to search through a tree with millions of possible sequences of moves and countermoves in order to determine the best move to make.

Instead of searching the vertices in the order of closeness to the start, they are searched in the order of closeness to the destination. However, we do not know the actual distance from a vertex to the destination. A *heuristic function*, or *heuristic*, is used to guesstimate the distance from each vertex to the destination based on available data. In Figure 10.29, the numbers below the city names are the direct distance (as the proverbial crow flies) to Chicago; we can use this as the heuristic.

A* adds an additional array, f, to Dijkstra's algorithm. In this array, `f`[v] is the estimated distance from the start to the destination through vertex v. It is the sum of `d`[v], the actual distance from the start to vertex v, and `h`[v], the distance from vertex v to the destination calculated by the heuristic. The next vertex to visit is selected based on the smallest value of `f`. We will see that `d`[v] and `f`[v] can change as we explore new paths to a vertex v, but `h`[v] remains the same.

Using A* and the actual distances in Figure 10.29, we start at Philadelphia and initialize the distance from Philadelphia to Pittsburgh to 330 and the distance from Philadelphia to Columbus to 460 as shown in the table that follows. The direct distance as a crow flies from a vertex to Chicago is shown in color under the city name and is listed in the column with label `h`. In the row for vertex Pittsburgh, 411 is the direct distance from Pittsburgh to Chicago, 330 is the actual distance from Philadelphia to Pittsburgh, and 741 is their sum. If a vertex v is not adjacent to Philadelphia, `d`[v] and `f`[v] are shown as infinity. The column `p`[v] shows the previous city in our route to vertex v. Since we are still at the origin, `p`[v] is Philadelphia for each vertex.

Vertex	h	d	f	p
Pittsburgh	411	330	741	Philadelphia
Columbus	276	460	736	Philadelphia
Cleveland	311	∞	∞	Philadelphia
Toledo	214	∞	∞	Philadelphia
Indianapolis	164	∞	∞	Philadelphia
Detroit	241	∞	∞	Philadelphia
Ann Arbor	205	∞	∞	Philadelphia
Ft Wayne	140	∞	∞	Philadelphia
Chicago	0	∞	∞	Philadelphia

The vertex with the smallest value of f is Columbus, so we update the distances from Columbus to Indianapolis, Cleveland, and Toledo. Column d now shows the actual distance traveled to reach these three cities from Philadelphia through Columbus. For each vertex v, f[v] shows the new estimated distance to Chicago, h[v] + d[v]. Column p shows the city from which each vertex v has been reached. (The shading indicates that vertex Columbus has been removed from V–S.)

Vertex	h	d	f	p
Pittsburgh	411	330	741	Philadelphia
Columbus	276	460	736	Philadelphia
Cleveland	311	610	921	Columbus
Toledo	214	615	829	Columbus
Indianapolis	164	630	794	Columbus
Detroit	241	∞	∞	Philadelphia
Ann Arbor	205	∞	∞	Philadelphia
Ft Wayne	140	∞	∞	Philadelphia
Chicago	0	∞	∞	Philadelphia

The smallest value for f is now 741, so we visit Pittsburgh and recalculate d for its neighbor Cleveland. Because its new value for d (465) is smaller than its previous value (610), we update d, f, and p for Cleveland and remove Pittsburgh from V–S.

Vertex	h	d	f	p
Pittsburgh	411	330	741	Philadelphia
Columbus	276	460	736	Philadelphia
Cleveland	311	465	776	Pittsburgh
Toledo	214	615	829	Columbus
Indianapolis	164	630	794	Columbus
Detroit	241	∞	∞	Philadelphia
Ann Arbor	205	∞	∞	Philadelphia
Ft Wayne	140	∞	∞	Philadelphia
Chicago	0	∞	∞	Philadelphia

The smallest value for f is 776 so we visit Cleveland and recalculate d for its neighbor Toledo. Because its new value for d (590) is smaller than its previous value (615), we update d, f, and p for Toledo and remove Cleveland from V–S.

Vertex	h	d	f	p
Pittsburgh	411	330	741	Philadelphia
Columbus	276	460	736	Philadelphia
Cleveland	311	465	776	Pittsburgh

Vertex	h	d	f	p
Toledo	214	590	804	Cleveland
Indianapolis	164	630	794	Columbus
Detroit	241	∞	∞	Philadelphia
Ann Arbor	205	∞	∞	Philadelphia
Ft. Wayne	140	∞	∞	Philadelphia
Chicago	0	∞	∞	Philadelphia

If we were following Dijkstra's algorithm, the next vertex to be visited would be Toledo since it is the closest to Philadelphia, but the estimated distance to Chicago via Toledo is 804 while the estimated distance to Chicago via Indianapolis is 794. Thus, Indianapolis is the next visited vertex. The distances to Chicago and Ft. Wayne are updated, giving the following:

Vertex	h	d	f	p
Pittsburgh	411	330	741	Philadelphia
Columbus	276	460	736	Philadelphia
Cleveland	311	465	776	Pittsburgh
Toledo	214	590	804	Cleveland
Indianapolis	164	630	794	Columbus
Detroit	241	∞	∞	Philadelphia
Ann Arbor	205	∞	∞	Philadelphia
Ft. Wayne	140	750	890	Indianapolis
Chicago	0	800	800	Indianapolis

Since 800 is the smallest `f` value and vertex Chicago is the destination, we now know the shortest distance (800) and path from Philadelphia to Chicago. The results are graphically shown in Figure 10.30. The gray vertices are not visited, and the dashed edges are not examined.

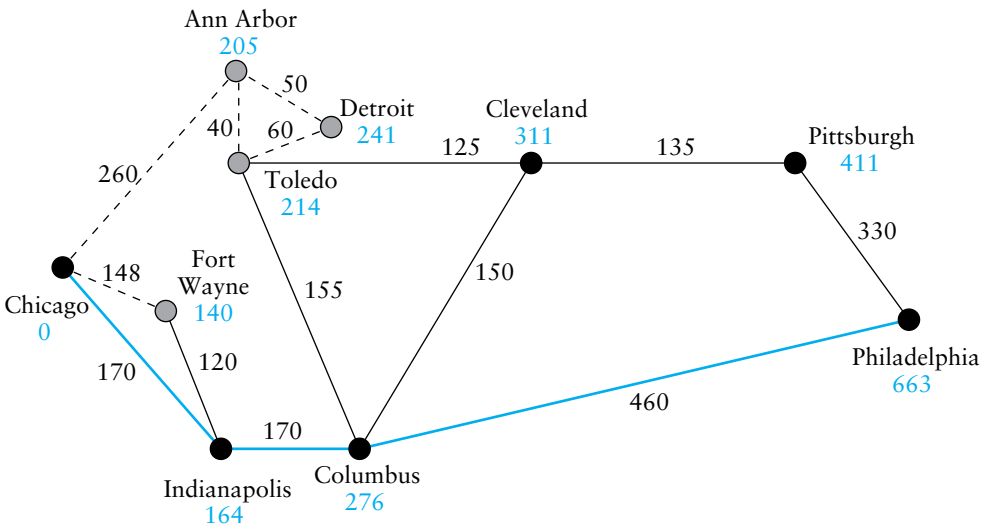


FIGURE 10.30
Path from Philadelphia to Chicago as Determined by the A* Algorithm

A* Algorithm

The A* algorithm is shown below as a modification to Dijkstra's algorithm that was given in Section 10.6. The additions to Dijkstra's algorithm are shown in color.

1. Initialize *V*–*S* with all vertices except the starting vertex *s*.
2. **for** `all` *v* in *V*–*S*
3. Set `p`[*v*] to *s*
4. **if** there is an edge (*s*, *v*)
5. Set `d`[*v*] to *w*(*s*, *v*)
6. Set `f`[*v*] to `d`[*v*] + `h`(*v*)
   else
7. Set `d`[*v*] to ∞
8. Set `f`[*v*] to ∞
9. **while** *V*–*S* is not empty
10. **for** all *u* in *V*–*S* find the smallest `f`[*u*]
11. **if** *u* is the destination `stop`.
12. Remove *u* from *V*–*S* and add it to *S*
13. **for** all *v* adjacent to *u* in *V*–*S*
14. **if** `d`[*u*] + *w*(*u*, *v*) is less than `d`[*v*]
15. Set `d`[*v*] to `d`[*u*] + *w*(*u*, *v*)
16. Set `f`[*v*] to `d`[*v*] + `h`(*v*)
17. Set `p`[*v*] to *u*.

Implementation

Listing 10.8 provides a straightforward implementation of the A* algorithm. It is like the implementation of Dijkstra's algorithm shown in Listing 10.6 with the differences shown in color. Because we don't use set *S*, it is omitted in the implementation. There are three additional parameters: the start vertex `start`, the destination vertex `dest`, the heuristic function `h`, and the array `fScore` to hold the sum of the distance plus the heuristic value. Although `start` is normally 0, we have included it as a parameter. The heuristic function `h` (shown in Listing 10.9) implements the `BiFunction` functional interface.

LISTING 10.8

A* Algorithm Implementation

```
/**
   AStar Shortest-Path algorithm.
   @param graph The weighted graph to be searched
   @param start The start vertex
   @param dest The destination vertex
   @param h The heuristic function
   @param pred Output array to contain the predecessors
          in the shortest path
   @param dist Output array to contain the distance
          in the shortest path
   @param fScore Output array to contain the estimated
          distance in the shortest path
 */
 public static void aStarAlgorithm(Graph graph,
            int start, int dest, BiFunction<Integer, Integer, Double> h,
            int[] pred,              // column p in tables
            double[] dist,           // column d in tables
```

```
            double[] fScore) {       // column f in tables
        int numV = graph.getNumV();
        Set<Integer> vMinusS = new HashSet<>(numV);
        // Initialize V-S.
        for (int i = 0; i < numV; i++) {
            if (i != start) {
                vMinusS.add(i);
            }
        }
        // Initialize pred and dist.
        for (int v : vMinusS) {
            pred[v] = start;
            var e = graph.getEdge(start, v);
            if (e != null) {
                dist[v] = e.getWeight();
                fScore[v] = dist[v] + h.apply(v, dest);
            } else {
                dist[v] = Double.POSITIVE_INFINITY;
                fScore[v] = Double.POSITIVE_INFINITY;
            }
        }

        // Main loop
        while (!vMinusS.isEmpty()) {
            // Find the value u in V-S with the smallest dist[u].
            double minDist = Double.POSITIVE_INFINITY;
            int u = -1;
            for (int v : vMinusS) {
                if (fScore[v] < minDist) {
                    minDist = fScore[v];
                    u = v;
                }
            }
            // If u is the destination, return
            if (u == dest) return;
            // Remove u from vMinusS.
            vMinusS.remove(u);
            // Update the distances.
            var itr = graph.edgeIterator(u);
            while (itr.hasNext()) {
                Edge e = itr.next();
                int v = e.getDest();
                if (vMinusS.contains(v)) {
                    double weight = e.getWeight();
                    if (dist[u] + weight < dist[v]) {
                        dist[v] = dist[u] + weight;
                        fScore[v] = dist[v] + h.apply(v, dest);
                        pred[v] = u;
                    }
                }
            }
        }
    }
```

Demonstrating the A* Algorithm

Listing 10.9 shows the main method of a program to demonstrate the A* algorithm. The method `readCityCoordinates` will return a two-dimensional `double` array containing the coordinates of each city that are stored in array `coord`. The input begins with the number of cities followed by the x, y coordinates for each city, beginning with the start city and ending

with the destination city. This input is then followed by the list of edges for each graph vertex or city, read by `loadEdgesFromFile`.

The heuristic function, `h`, is defined by the lambda expression shown in color in Listing 10.9. Each time it is called, the start and destination vertices are passed to `v` and `w`.

LISTING 10.9
A* Algorithm Demonstration

```
public static void main(String[] args) {
        try (var in = new Scanner(new FileReader(args[0]))) {
            double[][] coord = readCityCoordinates(in);
            int numVert = coord.length;
            Graph g = new ListGraph(numVert, false);
            g.loadEdgesFromFile(in);
            int start = 0;
            int dest = numVert-1;
            double[] dist = new double[numVert];
            double[] fScore = new double[numVert];
            int[] pred = new int[numVert];
            BiFunction<Integer, Integer, Double> h = (v, w) -> {
                double x1 = coord[v][0];
                double y1 = coord[v][1];
                double x2 = coord[w][0];
                double y2 = coord[w][1];
                double deltax = x1 - x2;
                double deltay = y1 - y2;
                return Math.sqrt(deltax*deltax + deltay*deltay);
            };
            // Call A* algorithm to define arrays pred, dist, and fScore
            AStarAlgorithm.aStarAlgorithm(g, start, dest, h,
                                pred, dist, fScore);
            // Construct the path.
            Deque<Integer> path = new ArrayDeque<>();
            int previous = pred[dest];
            while (previous != start) {
                 path.push(previous);
                 previous = pred[previous];
            }
            path.push(previous);

            // output the path.
            System.out.print("The shortest path is: ");
            while (!path.isEmpty())
                 System.out.print(path.pop() + " -> ");
            System.out.println(dest);
            System.out.println("Its length is " + fScore[dest]);
        } catch (FileNotFoundException ioex) {
            System.out.println(args[0] + " not found");
            System.exit(1);
        }
    }
```

Running this program with the correct city coordinates and the data in Figure 10.30 would show the following result:

```
The shortest path is: 0 -> 2 -> 5 -> 9
Its length is 800.0
```

EXERCISES FOR SECTION 10.7

SELF-CHECK

1. Use the A* algorithm to find the shortest path from Cleveland to Chicago. Use the values for `h` and `d` from Figure 10.30. Set the value of `h`(Philadelphia) to 663. Trace the execution of the algorithm by drawing a sequence of tables like those shown in this section. The first table should look like this:

Vertex	h	d	f	p
Toledo	214	125	339	Cleveland
Columbus	276	150	426	Cleveland
Pittsburgh	411	135	546	Cleveland
Philadelphia	663	∞	∞	Cleveland
Indianapolis	164	∞	∞	Cleveland
Detroit	241	∞	∞	Cleveland
Ann Arbor	205	∞	∞	Cleveland
Ft Wayne	140	∞	∞	Cleveland
Chicago	0	∞	∞	Cleveland

2. Trace the execution of method `aStarAlgorithm` in Listing 10.8 to find the shortest path from Philadelphia to Chicago by showing set `vMinusS`, and arrays `dist`, `fScores`, and `pred` at the start of each iteration of the `while` loop. Use the normal notation for a set and array.

PROGRAMMING

1. Write method `readCityCoordinates`.

Chapter Review

- A graph consists of a set of vertices and a set of edges. An edge is a pair of vertices. Graphs may be either undirected or directed. Edges may have a value associated with them known as the weight.
- In an undirected graph, if $\{u, v\}$ is an edge, then there is a path from vertex u to vertex v, and vice versa.
- In a directed graph, if (u, v) is an edge, then (v, u) is not necessarily an edge.
- If there is an edge from one vertex to another, then the second vertex is adjacent to the first.
- A path is a sequence of adjacent vertices. A path is simple if the vertices in the path are distinct except, perhaps, for the first and last vertices, which may be the same. A cycle is a path in which the first and last vertices are the same.

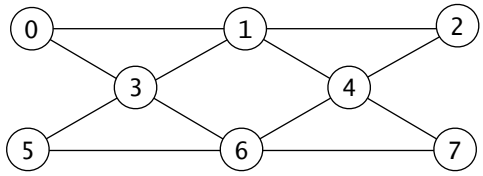
- A graph is considered connected if there is a path from each vertex to every other vertex.
- A tree is a special case of a graph. Specifically, a tree is a connected graph that contains no cycles.
- Graphs may be represented by an array of adjacency lists. There is one list for each vertex, and the list contains the edges that originate at this vertex.
- Graphs may be represented by a two-dimensional square array called an adjacency matrix. The entry [*u*][*v*] will contain a value to indicate that an edge from *u* to *v* is present or absent.
- A breadth-first search of a graph finds all vertices reachable from a given vertex via the shortest path, where the length of the path is based on the number of vertices in the path.
- A depth-first search of a graph starts at a given vertex and then follows a path of unvisited vertices until it reaches a point where there are no unvisited vertices that are reachable. It then backtracks until it finds an unvisited vertex, and then continues along the path to that vertex.
- A topological sort determines an order for starting activities that are dependent on the completion of other activities (prerequisites). The finish order derived from a depth-first traversal represents a topological sort.
- Dijkstra's algorithm finds the shortest path from a start vertex to all other vertices, where the distance from one vertex to another is determined by the weight of the edge between them.
- Prim's algorithm finds the minimum spanning tree for a graph. This consists of the subset of the edges of a connected graph whose sum of weights is the minimum and the graph consisting of only the edges in the subset is still connected.

User-Defined Classes and Interfaces in This Chapter

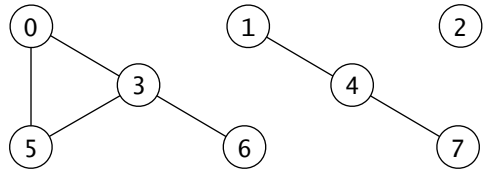
```
BreadthFirstSearch
DepthFirstSearch
Edge
Graph
ListGraph
MatrixGraph
MatrixGraph.Iter
Maze
TopologicalSort
```

Quick-Check Exercises

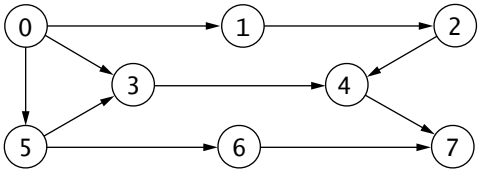
1. For the following graph:
   a. List the vertices and edges.
   b. True or false: The path 0, 1, 4, 6, 3 is a simple path.
   c. True or false: The path 0, 3, 1, 4, 6, 3, 2 is a simple path.
   d. True or false: The path 3, 1, 2, 4, 7, 6, 3 is a cycle.



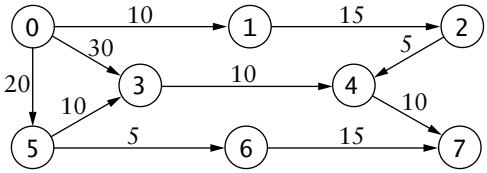
2. Identify the connected components in the following graph.



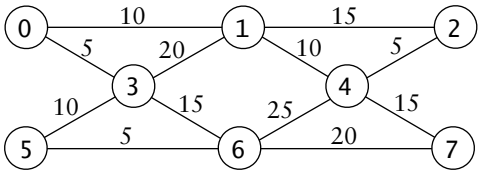
3. For the following graph:
   a. List the vertices and edges.
   b. Does this graph contain any cycles?



4. Show the adjacency matrices for the graphs shown in Questions 1, 2, and 3.
5. Show the adjacency lists for the graphs shown in Questions 1, 2, and 3.
6. Show the breadth-first search tree for the graph shown in Question 1, starting at vertex 0.
7. Show the depth-first search tree for the graph shown in Question 3, starting at vertex 0.
8. Show a topological sort of the vertices in the graph shown in Question 3.
9. In the following graph, find the shortest path from 0 to all other vertices.



10. In the following graph, find the minimum spanning tree.



Review Questions

1. What are the different types of graphs?
2. What are the different types of paths?
3. What are two common methods for representing graphs? Can you think of other methods?
4. What is a breadth-first search? What can it be used for?
5. What is a depth-first search? What can it be used for?
6. Under what circumstances are the paths found by Dijkstra's algorithm not unique?
7. Under what circumstances is the minimum spanning tree unique?
8. What is a topological sort?

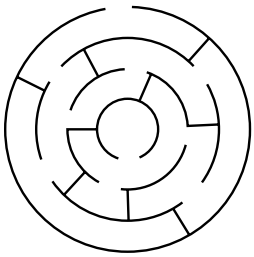
Programming Projects

1. Design and implement the `MatrixGraph` class.
2. Rewrite method `dijkstrasAlgorithm` to use a priority queue as we did for method `primsAlgorithm`. When inserting edges into the priority queue, the weight is replaced by the total distance from the source vertex to the destination vertex. The source vertex, however, remains unchanged as it is the predecessor in the shortest path.

3. In both Prim's algorithm and Dijkstra's algorithm, edges are retained in the priority queue, even though a shorter edge to a given destination vertex has been found. This can be avoided, and thus performance improved, by using a `ModifiablePriorityQueue`. Extend the `PriorityQueue` class described in Chapter 6 as follows:

```
/** A ModifiablePriorityQueue stores Comparable objects. Items
    may be inserted in any order. They are removed in priority
    order, with the smallest being removed first, based on the
    compareTo method. If the item is already in the queue,
    then it is presumed that its priority has changed,
    and it is removed and re-inserted in the heap.
*/

public class ModifiablePriorityQueue<E extends Comparable<E>>
                                    extends PriorityQueue<E> {
    /** Insert an item into the priority queue.
        @param obj The item to be inserted
        @return A locator to the item
    */
    int insert(E obj);

    /** Remove the smallest item in the priority queue.
        @return The smallest item in the priority queue
    */
    E poll();
    . . .
```

4. Implement Dijkstra's algorithm using the `ModifiablePriorityQueue`.
5. A maze can be constructed from a series of concentric circles. Between the circles there are walls placed, and around the circles there are doors. The walls divide the areas between the circles into chambers, and the doors permit movement between chambers. The positions of the doors and walls are given in degrees measured counterclockwise from the horizontal. For example, the maze shown on the left can be described as follows:



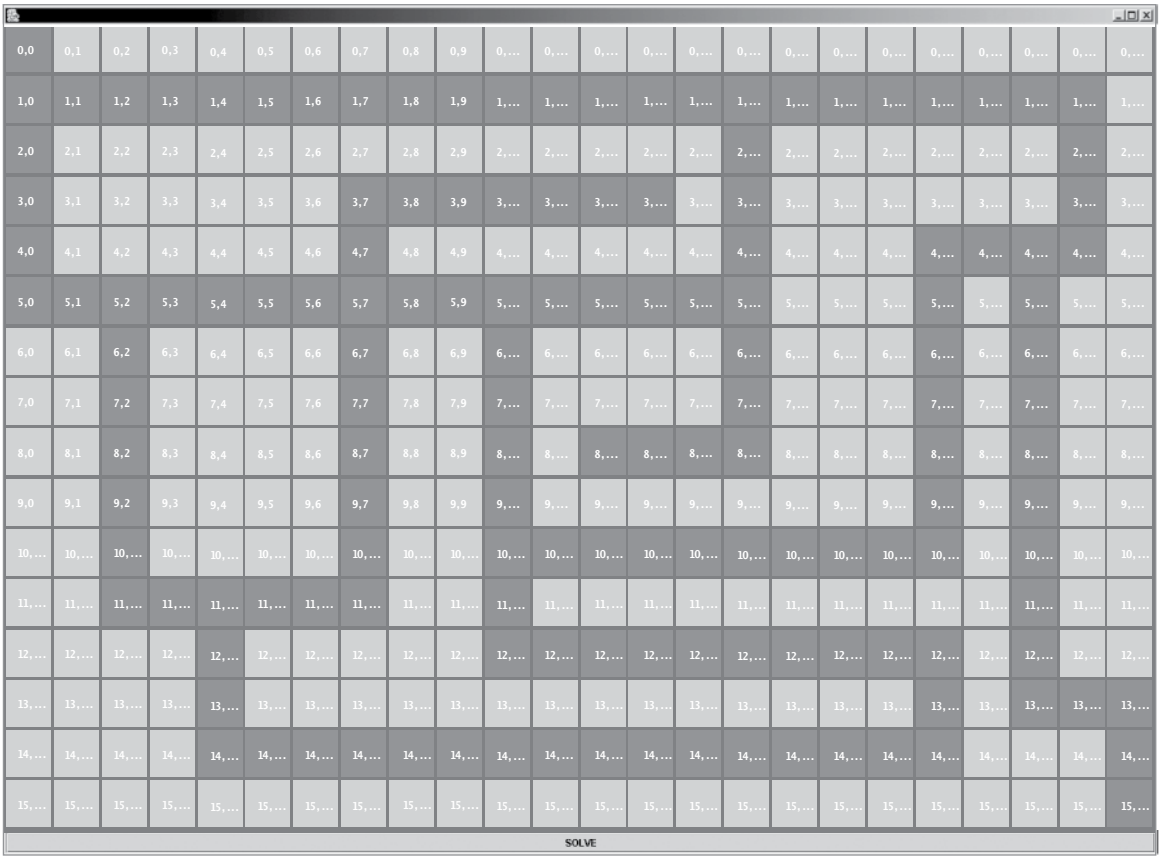
Number of circles	4	
Position of doors	Innermost circle	251–288
	Next outer circle	67–90, 161–180, 243–256, 342–360
	Next outer circle	26–40, 135–146, 198–215, 305–319
	Outer circle	90–100
Position of walls	Inner ring	65, 180
	Middle ring	0, 100, 225, 270
	Outer ring	45, 135, 300

Write a program that inputs a description of a maze in this format and finds the shortest path from the outside to the innermost circle. The shortest path is the one that goes through the smallest number of chambers.

6. In Chapter 5 we discussed the class `MazeTest`, which reads a rectangular maze as a sequence of lines consisting of 0s and 1s, where a 0 represents an open square and a 1 represents a closed one. For example, the maze shown in Figure 10.21 and reproduced here has the following input file:

```
01111111111111111111111
00000000000000000000001
01111111111111011111101
01111110000001011111101
01111101111111011100001
```

```
000000000000000011101011
110111101101111011101011
110111101101111011101011
110111101101000011101011
110111101101111111101011
110111101100000000001011
110000001101111111111011
111101111100000000001011
111101111111111111101000
111100000000000000001110
111111111111111111111110
```



Write a program that reads input in this format and finds the shortest path, where the distance along a path is defined by the number of squares covered.

Answers to Quick-Check Exercises

1. **a.** Vertices: {0, 1, 2, 3, 4, 5, 6, 7}
   Edges: {{0, 1}, {0, 3}, {1, 2}, {1, 3}, {1, 4}, {2, 4}, {3, 5}, {3, 6}, {4, 6}, {4, 7}, {5, 6}, {6, 7}}
   b. True
   c. False
   d. True
2. The connected components are {0, 3, 5, 6}, {1, 4, 7}, and {2}.

3. **a.** Vertices: {0, 1, 2, 3, 4, 5, 6, 7}
   Edges: {**(0, 1), (0, 3), (0, 5), (1, 2), (2, 4), (3, 4), (4, 7), (5, 3), (5, 6), (6, 7)**}
   b. The graph contains no cycles.
4. For the graph shown in Question 1:

Row \ Column	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]		**1**		**1**				
[1]	**1**		**1**	**1**	**1**			
[2]		**1**			**1**			
[3]	**1**	**1**				**1**	**1**	
[4]		**1**	**1**				**1**	**1**
[5]				**1**			**1**	
[6]				**1**	**1**	**1**		**1**
[7]					**1**		**1**	

For Question 2:

Row \ Column	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]				**1**		**1**		
[1]					**1**			
[2]								
[3]	**1**					**1**	**1**	
[4]		**1**						**1**
[5]	**1**			**1**				
[6]				**1**				
[7]					**1**			

For Question 3:

Row \ Column	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]		**1**		**1**		**1**		
[1]			**1**					
[2]					**1**			
[3]					**1**			
[4]								**1**
[5]				**1**			**1**	
[6]								**1**
[7]								

5. For Question 1:

[0] → 1 → 3
[1] → 0 → 2 → 3 → 4
[2] → 1 → 4
[3] → 0 → 1 → 5 → 6
[4] → 1 → 2 → 6 → 7
[5] → 3 → 6
[6] → 3 → 4 → 5 → 7
[7] → 4 → 6

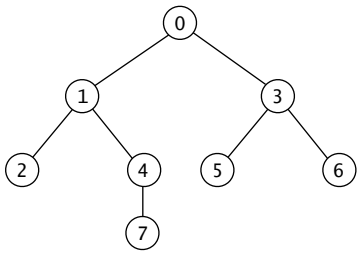
For Question 2:

[0] → 3 → 5
[1] → 4
[2] →
[3] → 0 → 5 → 6
[4] → 1 → 7
[5] → 0 → 3
[6] → 3
[7] → 4

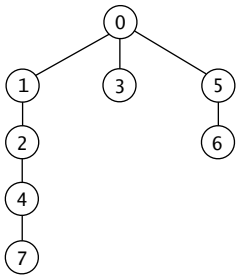
For Question 3:

[0] → 1 → 3 → 5
[1] → 2
[2] → 4
[3] → 4
[4] → 7
[5] → 3 → 6
[6] → 7
[7] →

6.



7.

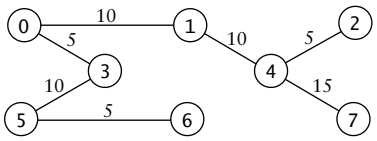


8. 0, 5, 6, 3, 1, 2, 4, 7

9.

Vertex	Distance	Path
1	10	0 → 1
2	25	0 → 1 → 2
3	30	0 → 3 (or 0 → 5 → 3)
4	30	0 → 1 → 2 → 4
5	20	0 → 5
6	25	0 → 5 → 6
7	40	0 → 5 → 6 → 7 (or 0 → 1 → 2 → 4 → 7)

10.



Appendix A

Introduction to Java

Appendix Objectives

- To understand the essentials of object-oriented programming in Java
- To learn about the primitive data types of Java
- To understand how to use the control structures of Java
- To learn how to use predefined classes such as `Math`, `String`, `StringBuilder`, `StringBuffer`, and `StringJoiner`
- To introduce regular expressions and `Pattern` and `Matcher` classes
- To learn how to write and document your own Java classes
- To understand how to use arrays in Java
- To understand how to use enumerators in Java
- To learn how to perform input/output (I/O) in Java using simple dialog windows
- To introduce the `Scanner` class for input and the `Formatter` class for output
- To learn how to perform I/O in Java using streams and readers
- To learn how to use the **try-catch-finally** sequence to catch and process exceptions
- To understand what it means to throw an exception and how to throw an exception in a method

This appendix reviews object-oriented programming in Java. It is oriented to a student who has had a first course in programming in Java or another language and who, therefore, is familiar with control statements for selection and repetition, basic data types, arrays, and methods or functions. If your first course was in Java, you can skim this appendix for review or just use it as a reference as needed. However, you should read it more carefully if your Java course did not emphasize object-oriented design.

If your first course was not in Java, you should read this appendix carefully. If your first course followed an object-oriented approach but was in another language, you should concentrate on the differences between Java syntax and the language that you know. If you have programmed only in a language that was not object-oriented, you will need to concentrate on aspects of object-oriented programming and classes as well as Java syntax.

The appendix begins with an introduction to the Java environment and the Java Virtual Machine (JVM). Next, it covers the basic data types of Java, called primitive data types, and provides an introduction to objects and classes. Control structures and methods are then discussed.

The Java Application Programming Interface (API) provides a rich collection of classes that simplify programming in Java. The first Java classes that we cover are the `String`, `StringBuilder`, `StringBuffer`, `StringJoiner`, and `Math` classes. The `String` class provides several methods and an operator `+` (concatenation) that process sequences of characters (strings). The `Math` class provides many methods for performing standard mathematical computations.

Next, we show you how to design and write your own classes consisting of data fields and methods. We also discuss the Java wrapper classes, which enable a programmer to create and process objects that contain primitive-type values.

We describe a specific format for comments in classes. Using this commenting style enables you to generate HTML pages with clear and complete documentation for classes in the same form as the Java documentation provided on the Oracle website.

We also review array objects in Java. We cover both one- and two-dimensional arrays.

Next, we discuss I/O. We show how to use the `JOptionPane` class (part of package `javax.swing`) to create dialog windows for data entry and for output. We also show how to use streams, readers, and the console for I/O.

Finally, we discuss how to handle exceptions and to throw exceptions.

Introduction to Java

A.1 The Java Environment and Classes

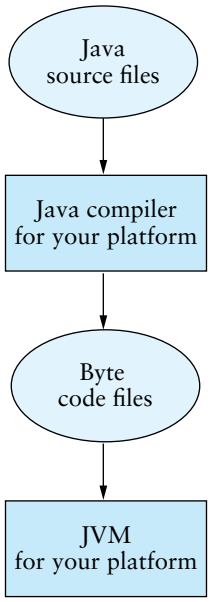
Before we talk about the Java language, we will briefly discuss the Java environment and how Java programs are executed. Java, developed by Sun Microsystems Corporation, enjoys its popularity because it is a *platform-independent*, object-oriented language and because Java facilitated developing software for the World Wide Web. Being platform independent means that a Java program will run on any kind of computer. Although platform independence is a goal for all high-level language programs, it is not always achieved. Java

comes closer to achieving this goal than most by providing implementations of the JVM (discussed next) for many platforms.

The Java Virtual Machine

Java is platform independent because the Java designers utilize the concept of a Java Virtual Machine (JVM), which is a software "computer" that runs inside an actual computer. Before you can execute a Java program, the classes in the Java program must first be translated from the Java language in which they were written into an executable form in the traditional way by a compiler program. Instead of a file of platform-dependent machine-language instructions, however, which is the normal output from a compiler, the Java compiler generates a file of platform-independent Java *byte code* instructions. When you execute the program, your computer's JVM *interprets* each byte code instruction and carries it out. The JVM for machines running Microsoft Windows is different from the JVM for UNIX or Apple machines, but they all process byte code instructions in the same way (see Figure A.1).

FIGURE A.1
Compiling and Executing a Java Program



The Java Compiler

The Java compiler is also platform specific, even though it produces the same byte code file for a given Java source program on all platforms. It must be platform specific because it executes machine-language instructions for a particular platform, and these instructions are not the same for all platforms.

Classes and Objects

In Java and object-oriented programming in general, the class is the fundamental programming unit. Every program is written as a collection of classes, and all code that you write must be part of a class. In Java, class definitions are stored in separate files with the extension `.java`. The file name must be the same as the class name defined within.

If you are new to object-oriented design, you may be confused about the differences between a class and an object. A *class* is a general description of a group of entities (called *objects* or *instances* of the class) that all have the same characteristics—that is, they can all perform the same kinds of operations (class *method*s), and the same pieces of information (class *attributes* or *data fields*) are meaningful for all of them. For example, the class `House` would describe a collection of entities that each have a number of bedrooms, a number of bathrooms, a kind of roof, and so on (but not a horsepower rating or mileage); they can all be built, remodeled, assessed for property tax, and so on (but not have their transmission fluid changed). The house where you live and the house where your best friend lives can be represented by two objects of class `House`.

Classes extend Java by providing additional data types. For example, the class `String` is a predefined class that enables the programmer to process sequences of characters easily.

The Java API

The Java programming language consists of a relatively small core language augmented by an extensive collection of *packages* (called libraries in other languages), which constitute the Java API and give Java additional capabilities. Each package contains a collection of related Java classes. We will use several of these packages in this textbook. Among them are the `javax.swing` package and the `java.util` package. You can find out about these packages by accessing the Java website maintained by Oracle corporation at `https://docs.oracle.com/en/java/javase/11/`.

Java documentation is provided as a linked collection of Web pages. In Section A.7, we will discuss how you can write your own Java documentation that follows this style.

Versions of Java

Java is constantly being updated and improved. The previous edition of this book used Java Version 8 (Java 8). In this edition, we will use some new features that have been introduced through Java 11, which is currently the Java version that has long-term support by the Java community. We will identify new features since Java 8 and show you how to manage without them if you are using an older Java version.

The `import` Statement

Next, we show a sample Java source file (`HelloWorld.java`) that contains an application program (`class HelloWorld`). Our goal in the rest of this section is to give you an overview of the process of creating and executing an application program. The statements in this program will be covered in more detail later in this chapter.

```
import java.util.Scanner;

/** A HelloWorld class.
    @author Koffman and Wolfgang
 */
public class HelloWorld {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scan.nextLine();
        System.out.println("Hello " + name + ", welcome to Java!");
    }
}
```

The Java source file begins with the statement

```
import java.util.Scanner;
```

This statement tells the Java compiler to make the class `Scanner` defined in the `java.util` package accessible to this file. The semicolon at the end of the line is used to terminate a Java statement. The next three lines bracketed, by /** and */, are comments that provide information about the class but are not executable Java statements.

Class `HelloWorld` begins with the line

```
public class HelloWorld {
```

which identifies `HelloWorld` as a public class and makes it visible to other classes (or the JVM).

Method `main`

The line

```
public static void main(String[] args) {
```

identifies the start of the definition for method `main`. This is the place where the JVM begins the execution of an application program. The words **`public static void`** tell the compiler that `main` is accessible outside of the class (**`public`**), it is a **`static`** method (explained in Section A.4), and it does not return a value (**`void`**). The part in parentheses after `main` describes the method's parameters, an array of `Strings` named `args`. Arrays discussed in Section A.8. We always write the heading for method `main` in this way.

The first statement in `main` declares a `Scanner` object named `scan` that will be used to read input from the console (`Sytem.in`). The second statement displays the message:

```
Enter your name:
```

on the console. The statement

```
String name = scan.nextLine();
```

reads the characters that are typed in before the Enter key is pressed. These characters are converted to a `String` and placed in the variable `name`. Assuming that the characters `Kathy` were typed in, the statement:

```
System.out.println("Hello " + name + ", welcome to Java!");
```

will display the output:

```
Hello Kathy, welcome to Java!
```

Execution of a Java Program

You can compile and run class `HelloWorld` using an Integrated Development Environment (IDE) or the Java Development Kit (JDK). If you are using an IDE, type this class into the edit window for class `HelloWorld.java` and select **Run**. We discuss two popular IDEs in Chapter 3.

If you are not using an IDE, you must create this file using an editor program and save it as file `HelloWorld.java`. In Java 11, you can use the single command

```
java HelloWorld.java
```

to get Java to compile and run it. In earlier versions of Java you can accomplish this in two separate steps by first using

```
javac HelloWorld.java
```

to compile it. This will create the Java byte code file called `HelloWorld.class`. The command

```
java HelloWorld
```

starts the JVM and causes it to execute the byte code instructions in file `HelloWorld.class`. It begins execution with the byte code instructions for method `main`.

Use of `jshell`

The command `jshell` (introduced in Java 9) executes single Java statements and immediately shows the result. To activate `jshell`, open a command window on your computer and type `jshell` after the prompt and press Enter. On many computers, you can type `/exit` and press Enter to exit `jshell`.

The shaded lines below that begin with `jshell >` instruct `jshell` to execute each statement that follows the symbol >. The line(s) shown below each `jshell` command will contain either data entered by the user or the result of executing the statement.

```
jshell> Scanner scan = new Scanner(System.in)
scan ==> java.util.Scanner[ . . . ]
```

The line above shows that `scan` is declared as a `Scanner` object.

The first line below identifies `name` as repository for a sequence of characters to be processed by method `scan.nextLine`.

```
jshell> String name = scan.nextLine();
Kathy
name ==> "Kathy"
```

All characters typed on the next line before Enter is pressed are stored in `name` as shown on the last line above.

The next jshell command

```
jshell> System.out.println("Hello " + name + ", welcome to Java!");
Hello Kathy, welcome to Java!
```

causes execution of method `println`, which is defined in the Java package `System.out`. Method `println` (discussed in Section A.4) displays the `String` expression shown in parentheses in the form shown on the last line above.

EXERCISES FOR SECTION A.1

SELF-CHECK

1. What is the Java Virtual Machine? Is it hardware or software? How does its role differ from that of the Java compiler?
2. Explain the statement: You can write a Java program once and run it anywhere.
3. Explain the relationship between a class and an object. Which is general and which is specific?
4. What is displayed by `jshell` after executing each statement below? The method `nextInt` reads the integer typed on the line following the `jshell` command that calls it. Data type `int` declares variables that will store integers.

```
jshell > Scanner scan = new Scanner (System.in);
jshell > int birthYear;
jshell > birthYear = scan.nextInt();
jshell > int yearNow = 2020;
jshell > int age = yearNow - birthYear;
jshell > System.out.println("Your age is " + age);
```

PROGRAMMING

1. Write a Java class that asks the user to enter the number of nickels, dimes, and quarters and then computes the value of the coins in pennies and displays the result.
2. Open a command window on your computer. Type in `jshell` commands to perform the operations in Programming Exercise 1.
3. Write a Java class that reads in the distance you plan to travel, your average speed in miles per hour (or kilometers / hour), and the fuel consumption estimate for your vehicle. Then display the time it will take to travel and the amount of fuel your vehicle will consume.

A.2 Primitive Data Types and Reference Variables

Java distinguishes between two kinds of entities: primitive types (numbers, characters) and objects. Values associated with primitive-type data are stored in primitive-type variables. Objects, however, are associated with reference variables, which store an object's address. We will discuss primitive types and introduce objects in this section; we describe objects in more detail throughout the chapter.

Primitive Data Types

The primitive data types for Java represent numbers, characters, and boolean values (**`true`**, **`false`**) (see Table A.1). Integers are represented by data types **`byte`**, **`short`**, **`int`**, and **`long`**; real numbers are represented by **`float`** and **`double`**. The range of values for the numeric data types is in increasing order in Table A.1. The bold words in this paragraph are *reserved words* in Java and cannot be used by programmers for other purposes.

TABLE A.1
Java Primitive Data Types in Increasing Order of Range

Data Type	Range of Values
`byte`	–128 through 127
`short`	–32,768 through 32,767
`int`	–2,147,483,648 through 2,147,483,647
`long`	–9,223,372,036,854,775,808 through 9,223,372,036,854,775,807
`float`	Approximately $\pm 10^{-38}$ through $\pm 10^{38}$ and 0 with 6 digits precision
`double`	Approximately $\pm 10^{-308}$ through $\pm 10^{308}$ and 0 with 15 digits precision
`char`	The Unicode character set
`boolean`	`true`, `false`

TABLE A.2
The First 128 Unicode Symbols

	000	001	002	003	004	005	006	007
0	Null		Space	0	@	P	`	p
1			!	1	A	Q	a	q
2			"	2	B	R	b	r
3			#	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7	Bell		'	7	G	W	g	w
8	Backspace		(	8	H	X	h	x
9	Tab		)	9	I	Y	i	y
A	Line feed		*	:	J	Z	j	z
B	Escape		+	;	K	[	k	{
C	Form feed		,	<	L	\	l	|
D	Return		-	=	M	]	m	}
E			.	>	N	^	n	~
F			/	?	O	_	o	delete

Type **`char`** is used in Java to represent characters. Java uses the Unicode character set (two bytes per character), which provides a much richer set of characters than the ASCII character set (one byte per character) used by many earlier languages. Table A.2 shows the first 128 Unicode characters, which correspond to the ASCII characters. These include the control characters (for example, Tab and Line feed) and the Basic Latin alphabet.

Table A.2 uses the *Hexadecimal number system,* which is a base 16 number system. In Hexadecimal, the digits 0 through 9 have their normal decimal values. The characters A though F represent decimal values 10 through 15.

Unicode for each character consists of the three-digit column number (000 through 007) followed by the row number (0 through F). For example, the Unicode for the letter Q is 0051, and the Unicode for the letter q is 0071. The characters in the first two columns of Table A.2 and the Unicode character 007F (delete) are control characters. The hexadecimal number 007F (Unicode for delete) is equivalent to the decimal number 7 × 16 + 15.

Java uses type **`boolean`** to represent logical data. The **`boolean`** data type has only two values: **`true`** and **`false`**.

Primitive-Type Variables

Java uses *declaration statements* to declare and initialize primitive-type variables.

```
int countItems;
double sum = 0.0;
char star = '*';
boolean moreData;
```

The second and third of the preceding statements initialize variables `sum` and `star` to the values after the operator =. As shown, you can use primitive-type values (such as `0.0` and `'*'`) as *literals* in Java statements. A literal is a constant value that appears directly in a statement.

Note that type **`char`** values are enclosed in apostrophes, not quotes. The statement

```
char plus = "+";
```

would cause an error.

Identifiers, such as variable names in Java, must consist of some combination of letters, digits, the underscore character, and the $ character, beginning with a letter. Identifiers can't begin with a digit.

PROGRAM STYLE

Java Convention for Identifiers

Many Java programmers use "camel notation" for variable names. All letters are in lowercase except for identifiers that are made up of more than one word. The first letter of each word, starting with the second word, is in uppercase (e.g., `thisLongIdentifier`). Camel notation gets its name from the appearance of the identifier, with the uppercase letters in the interior forming "humps."

Primitive-Type Constants

Java programmers usually use all uppercase letters for constant identifiers, with an underscore symbol between words. The keywords **`static final`** identify a constant value that is **`static`** (more on this later) and **`final`**—that is, can't be changed.

```
static final int MAX_SCORE = 999;
static final double G = 3.82;
```

Operators

Table A.3 shows the Java operators in decreasing *precedence* (order of execution). The arithmetic operators (*, /, +, –) can be used with any of the primitive numeric types or type **`char`**, but not with type **`boolean`**. This is also the case for the Java remainder operator (%) and the increment (++) and decrement (–) operators.

TABLE A.3
Operator Precedence

Rank	Operator	Operation	Associativity
1	`[]`	Array subscript	Left
	`()`	Method call	
	`.`	Member access	
	`++`	Postfix increment	
	`--`	Postfix decrement	
2	`++`	Prefix increment	Right
	`--`	Prefix decrement	
	`+ -`	Unary plus or minus	
	`!`	Complement	
	`~`	Bitwise complement	
	`(`*`type`*`)`	Type cast	
	`new`	Object creation	
3	`*, /, %`	Multiply, divide, remainder	Left
4	`+`	Addition or string concatenation	Left
	`-`	Subtraction	
5	`<<`	Signed bit shift left	Left
	`>>`	Signed bit shift right	
	`>>>`	Unsigned bit shift right	
6	`<, <=`	Less than, less than or equal	Left
	`>, >=`	Greater than, greater than or equal	
	`instanceof`	Reference test	
7	`==`	Equal to	Left
	`!=`	Not equal to	
8	`&`	Bitwise and	Left
9	`^`	Bitwise exclusive or	Left
10	`|`	Bitwise or	Left
11	`&&`	Logical and	Left
12	`||`	Logical or	Left
13	`?:`	Conditional	Left
14	`=`	Assignment	Right
	`*=, /=, %=, +=, -=, <<=, <<=, >>>=, &=, |=`	Compound assignment	

Postfix and Prefix Increment

In Java you can write statements such as

```
i = i + 1;
```

using the *increment operator*:

```
i++;
```

This form is the *postfix increment*. You can also use the *prefix increment*

```
++i;
```

but the postfix increment (or decrement) is more common.

When the postfix form is used in an expression (e.g., `x * i++`), the variable `i` is evaluated and then incremented. When the prefix form is used in an expression (e.g., `x * ++i`), the variable `i` is incremented before it is evaluated.

EXAMPLE A.1 In the assignment

```
z = i++;
```

`i` is incremented, but `z` gets the value `i` had before it was incremented. So if `i` is 3 before the assignment statement, `z` would be 3 and `i` would be 4 after the assignment. In the assignment statement

```
z = ++i;
```

`i` is incremented and `z` gets its new value, so if `i` is 3 before the assignment, `z` and `i` would both be 4 after the assignment statement.

PITFALL

Using Increment and Decrement in Expressions with Other Operators

In the preceding example, the increment operator is used with the assignment operator in the same statement. Similarly, the expression `x * i++` uses the operators for multiplication and postfix increment. In this expression, the variable `i` is evaluated and then incremented. When the prefix form is used in an expression (e.g., `x * ++i`), the variable `i` is incremented before it is evaluated. However, you should avoid writing expressions like these, which could easily be interpreted incorrectly by the human reader.

Type Compatibility and Conversion

In operations involving mixed-type operands, the numeric type of the smaller range is converted to the numeric type of the larger range. This means that if an operation involves a type **`int`** and a type **`double`** operand, the type **`int`** operand is automatically converted to type **`double`**. This is called a *widening conversion*.

In an assignment operation, a numeric type of a smaller range can be assigned to a numeric type of a larger range; for example, a type **`int`** expression can be assigned to a type **`float`** or **`double`** variable. Java performs the widening conversion automatically.

```
int item = 37 ;
double realItem = item;     // Valid - automatic widening
```

However, the converse is not true.

```
double y = -67;
int x = y; // Invalid assignment
```

This statement is invalid because it attempts to store a real value (–67.0) in an integer variable. It would cause the syntax error `possible loss of precision; double, required: int`. This means that a type **`int`** expression is required for the assignment. You can use explicit *type cast* operations to perform a *narrowing conversion* and ensure that the assignment statement will be valid. In the following statement, the expression `(int)` instructs the compiler to cast the value of `y` to type **`int`** before assigning the integer value to `x`.

```
int x = (int) y; // Cast to int before assignment
```

Referencing Objects

In Java, you can declare reference variables that can reference objects of specified types. For example, the statement

```
String greeting;
```

declares a reference variable named `greeting` that can reference a `String` object. The statement

```
greeting = "hello";
```

specifies the particular `String` object to be referenced by `greeting`: the one that contains the characters in the string literal `"hello"`. What is actually stored in the memory cell allocated to greeting is the *address* of the area in memory where this particular object of type `String` is stored. We illustrate this in Figure A.2 by drawing an arrow from variable `greeting` to the object that it references (type `String`, value is `"hello"`). In contrast, the memory cell allocated to a primitive-type variable stores a value, not an address. Just as with the primitive variable declarations shown earlier, these two statements can be combined into one.

```
String greeting = "hello";
```

`String` objects are the only ones that can be created by assignment operations such as this one. We describe how to create other kinds of objects in the next section.

Two reference variables can reference the same object. The statement

```
String welcome = greeting;
```

copies the address in greeting to welcome, so `String` variable welcome also references the object shown in Figure A.2.

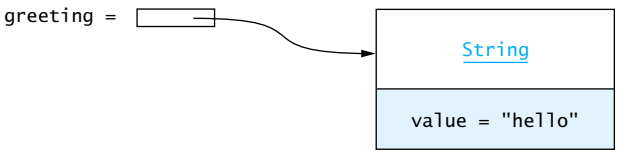


FIGURE A.2
Variable Greeting References a String Object

Creating Objects

The Java `new` *operator* can be used to create an instance of a class. The expression

```
new String("qwerty")
```

creates a new `String` instance (an object) that stores the character sequence consisting of the first six characters of the top row of letters on the standard keyboard (called a "`qwerty`" keyboard). The expression `new String("qwerty")` invokes a special method for the `String` class called a *constructor*. A constructor executes whenever a new object of any type is created; in this case, it initializes the contents of a `String` object to the character sequence `"qwerty"`.

The object created by the expression new `String("qwerty")` is an *anonymous* or unnamed object. Normally we want to be able to refer to objects that we create. We can declare a reference variable of type `String` and assign this object to the reference variable:

```
String keyboard = new String("qwerty");
```

PROGRAM STYLE

Use of `var` to declare local variables

The `main` method of the HelloWorld class declared two local variables: scan and name. Variables are often declared using the form:

type name = *expression*;

In the latter case, the compiler verifies that the type of the *expression* is the same (or can be converted to) the type specified.

Java 10 introduces a special type called **var** to reduce redundancy in local variable declarations. Instead of the original declaration of `scan`

```
Scanner scan = new Scanner(System.in);
```

the statement

```
var scan = new Scanner(System.in);
```

uses **var** to represent the data type `Scanner` because the compiler can infer from the *expression* that `scan` is an object of type `Scanner`. This is simpler than the original version in which type `Scanner` appears twice in the statement.

Similarly, instead of the statement

```
String name = scan.nextLine();
```

we could use

```
var name = scan.nextLine();
```

because the compiler knows that method `nextLine` returns a `String` and can therefore infer that the type of name is String. However this does not reduce redundancy and is not as obvious to the program reader, so we don't recommend it.

EXERCISES FOR SECTION A.2

SELF-CHECK

1. For the following assignment statement, assume that `x`, `y` are type **`double`** and `m`, `n` are type **`int`**. List the order in which the operations would be performed. Include any widening and narrowing conversions that would occur.

   ```
   m = (int) (x * y + m / n / y * (m + x));
   ```

2. What is the value assigned to `m` in Exercise 1 when `m` is 5, `n` is 3, `x` is 2.5, and `y` is 2.0?

3. What is the difference between a reference variable and a primitive-type variable?

4. Draw a diagram similar to Figure A.2 that shows the effect of the following statements.

   ```
   String y = new String("abc");
   String z = "def";
   String w = z;
   ```

A.3 Java Control Statements

The control statements of a programming language determine the flow of execution through a program. They fall into three categories: sequence, selection, and repetition.

Sequence and Compound Statements

A group of statements that is executed in sequence is written as a *compound statement* delimited (enclosed) by braces. The statements execute in the order in which they are listed.

EXAMPLE A.2 The following statements constitute a compound statement:

```
{
    double x = 3.45;
    double y = 2 * x;
    int i = (int) y;
    i++;
}
```

Selection and Repetition Control

Table A.4 shows the Java control statements for selection and repetition. (Java uses the same syntax for control structures as do C and C++.) We assume that you are familiar with basic programming control structures from your first course, so we won't dwell on them here. We will discuss the enhanced **`for`** statement in Chapter 2.

TABLE A.4
Java Control Statements

Control Structure	Purpose	Syntax

`if ... else`	Used to write a decision with *conditions* that select the alternative to be executed. Executes the first (second) alternative if the *condition* is true (false)	`if (condition) {`
		`  ...`
		`} else {`
		`  ...`
		`}`

`switch`	Used to write a decision with scalar values (integers, characters, enumerators) or strings that select the alternative to be executed. Executes the *statements* following the *label* that is the *selector* value. Execution falls through to the next **`case`** if there is no **`return`** or **`break`**. Executes the statements following **`default`** if the *selector* value does not match any *label*	`switch (selector) {`
		`  case label : statements; break;`
		`  case label : statements; break;`
		`  ...`
		`  default : statements;`
		`}`

`while`	Used to write a loop that specifies the repetition *condition* in the loop header. The *condition* is tested before each iteration of the loop, and, if it is true, the loop body executes; otherwise, the loop is exited	`while (condition) {`
		`  ...`
		`}`

`do . . . while`	Used to write a loop that executes at least once. The repetition *condition* is at the end of the loop. The *condition* is tested after each iteration of the loop, and, if it is true, the loop body executes again; otherwise, the loop is exited	`do {`
		`  ...`
		`} while (condition);`

TABLE A.4
Continued

Control Structure	Purpose	Syntax

for	Used to write a loop that specifies the *initialization*, repetition *condition*, and *update* steps in the loop header. The *initialization* statements execute before loop repetition begins; the *condition* is tested before each iteration of the loop and, if it is true, the loop body executes; otherwise, the loop is exited. The *update* statements execute after each iteration	`for (`*initialization*`;` *condition*`;`
		  update`) {`
		  `...`
		`}`

for	Used to write a loop that operates on each item in an *array* (see Section A.8). Each time through the loop body, the *variable* is assigned to the next item in the *array*. This is known as the enhanced `for` statement	`for (`*type variable* `:` *array*`) {`
		  `...`
		`}`

In Table A.3, each *condition* is a **`boolean`** expression in parentheses. Type **`boolean`** expressions often involve comparisons written using equality (`==`, `!=`) and relational operators (`<`, `<=`, `>`, `>=`). For example, the condition `(x + y > x - y)` is true if the sum of the two variables shown is larger than their difference. The logical operators `!` (not or complement), `&&` (and), and `||` (or) are used to combine **`boolean`** expressions. For example, the condition `(n >= 0 && n <= 10)` is true if `n` has a value between 0 and 10, inclusive.

Java uses short-circuit evaluation, which means that evaluation of a **`boolean`** expression terminates when its value can be determined. For example, if in the expression *bool1* `||` *bool2*, *bool1* is true, the expression must be true, so *bool2* is not evaluated. Similarly, in the expression *bool3* `&&` *bool4*, if *bool3* is false, the expression must be false, so *bool4* is not evaluated.

EXAMPLE A.3 In the condition

```
(num != 0 && sum / num)
```

if `num` is 0, the expression following `&&` is not evaluated. This prevents a division by zero error.

EXAMPLE A.4 The operator % in the condition `(nextInt % 2 == 0)` gives the remainder after an integer division, so the condition is true if `nextInt` is an even number. If `maxVal` has been defined, the following loops (**`for`** loop on top, **`while`** loop at the bottom) store the `sum` of the even integers from 1 to `maxVal` in variable `sum` (initial value 0), and they store the product of the odd integers in variable `prod` (initial value 1).

```
int sum = 0;   int prod = 1;
for (int nextInt = 1; nextInt <= maxVal; nextInt++) {
  if (nextInt % 2 == 0) {
    sum += nextInt;
  } else {
    prod *= nextInt;
  }
}
```

```
int sum = 0;  int prod = 1;
int nextInt = 1;
while (nextInt <= maxVal) {
  if (nextInt % 2 == 0) {
    sum += nextInt;
  } else {
    prod *= nextInt;
  }
  nextInt++
}
```

Nested `if` Statements

You can write **`if`** statements that select among more than two alternatives by nesting one **`if`** statement inside another. Often each inner **`if`** statement will follow the keyword **`else`** of its corresponding outer **`if`** statement.

EXAMPLE A.5 The following nested **`if`** statement has four alternatives. The conditions are evaluated in sequence until one evaluates to **`true`**. The compound statement following the first true condition then executes.

```
if (operator == '+') {
    result = x + y;
    addOp++;
}
else
    if (operator == '-') {
        result = x - y;
        subtractOp++;
    }
    else
        if (operator == '*') {
            result = x * y;
            multiplyOp++;
        }
        else
            if (operator == '/') {
                result = x / y;
                divideOp++;
            }
```

PROGRAM STYLE

Braces and Indentation in Control Statements

Java programmers often place the opening brace { on the same line as the control statement header. The closing brace } aligns with the first word in the control statement header. We will always indent the statements inside a control structure to clarify the meaning of the control statement.

Although we write the symbols } **`else`** { on one line, another popular style convention is to place the word **`else`** under the symbol } and aligned with **`if`** as shown on the left below. Also some programmers prefer to place the symbol { on its own line (shown on the right).

```
if (nextInt % 2 == 0) {
    sum += nextInt;
}
else {
    prod *= nextInt;
}
```

```
if (nextInt % 2 == |0|)
{
    sum += nextInt;
}
else
{
    prod *= nextInt;
}
```

Some programmers omit the braces when a true task or false task or a loop body consists of a single statement. Others prefer to include them always, both for clarity and because having the braces will permit them to insert additional statements later if needed.

PITFALL

Omitting Braces around a Compound Statement

The braces in the preceding example delimit compound statements. Each compound statement consists of two statements. If you omit a brace, you will get the syntax error '`else` without '`if`'.

PROGRAM STYLE

Writing `if` Statements with Multiple Alternatives

Java programmers often write nested **`if`** statements like those in the preceding example without indenting each nested **`if`**. The following multiple-alternative decision has the same meaning but is easier to write and read.

```
if (operator == '+') {
    result = x + y;
    addOp++;
} else if (operator == '-') {
    result = x - y;
    subtractOp++;
} else if (operator == '*') {
    result = x * y;
    multiplyOp++;
} else if (operator == '/') {
    result = x / y;
    divideOp++;
}
```

The `switch` Statement

The **`if`** statement in Example A.5 could also be written as the following **`switch`** statement. Each **`case`** label (e.g., '+') indicates a possible value of the selector expression operator. The statements that follow a particular label execute if the selector has that value. The type of the selector can be `char`, `byte`, `short`, `int`, `String` or an enumeration (see A.9).

The **`break`** statements cause an exit from the **`switch`** statement. Without them, execution would continue on to the statements in the next case. The last case, with label **`default`**, executes if the selector value doesn't match any **`case`** label. (Note that the compound statements for each case are not surrounded by braces.)

```
switch (operator) {
    case '+':
        result = x + y;
        addOp++;
        break;
    case '-':
        result = x - y;
        subtractOp++;
        break;
    case '*':
        result = x * y;
```

```
            multiplyOp++;
            break;
        case '/':
            result = x / y;
            divideOp++;
            break;
        default:
            // Do nothing
    }
```

EXERCISES FOR SECTION A.3

SELF-CHECK

1. What is the purpose of the **break** statement in the preceding **switch** statement? List the statements that would execute when operator is '–' with the **break** statements in place and if they were removed.
2. What is the difference between a **while** loop and a **do ... while** loop? What is the minimum number of repetitions of the loop body with each kind of loop?

PROGRAMMING

1. Rewrite the `for` statement in Example A.4 using a **do ... while** loop.

A.4 Methods and Class `Math`

Java programmers can use methods to define a group of statements that perform a particular operation. Methods are very similar to functions in other programming languages such as Python and C++. The Java method `minChar` that follows returns the character with the smaller Unicode value. The statements beginning with keyword **return** cause an exit from the method; the expression following **return** is the method result.

```
static char minChar(char ch1, char ch2) {
   if (ch1 <= ch2)
       return ch1;
   else
       return ch2;
}
```

The modifier **static** indicates that `minChar` is a *static method* or *class method*. A static method must be called by listing the name of the class in which it is defined, followed by a dot, then by the method name and any arguments. This is called *dot notation*. For example, the statement

```
char ch = ClassName.minChar('a', 'A');
```

would store the letter `A` in `ch` because uppercase letters have smaller codes than lowercase letters. (If method `minChar` is called within the class that defines it, the prefix *ClassName*. is not needed.) If the modifier `static` does not appear in a method header, the method is an *instance method*. We describe how to invoke instance methods next and show how to define them afterward.

The Instance Methods println and print

Methods that are not preceded by the modifier **static** are instance methods. To call or invoke an instance method, you need to apply it to an object using dot notation:

```
object.method(arguments)
```

One instance method that is useful for output operations is the method println (defined in class PrintStream). It can be applied to the PrintStream object System.out (the console window), which is defined in the System class. It has a single argument of any data type. If x is a type **double** variable, the statement

```
System.out.println(x);
```

displays the value of x in the console window. The statement

```
System.out.println("Value of x is " + x);
```

has a String expression as its argument (+ means concatenate, or join, strings). The string consists of the character sequence: Value of x is followed by the characters that represent the value of variable x. If x is 123.45, the output line will be

```
Value of x is 123.45
```

You would get the same effect using the statement pair

```
System.out.print("Value of x is ");
System.out.println(x);
```

The method print also displays its argument in the console window. However, it does not follow this information with the *newline* character, so the next execution of print or println will display information on the same output line.

PITFALL

Static Methods Can't Call Instance Methods

A static method can call other static methods directly. Also, an instance method can call a static method. However, a static method, including method main, can't call an instance method without first creating an object and applying the instance method to that object.

Call-by-Value Arguments

In Java, all method arguments are call-by-value. This means that if the argument is a primitive type, its value (not its address) is passed to the method, so the method can't modify the argument value and have the modification remain after return from the method. Some other programming languages provide a call-by-reference or call-by-address mechanism so that a method can modify a primitive-type argument.

If the argument is of a class type, the value that is passed to the method is the value of the reference variable, not the value of the object itself (see Section A.2). The reference variable value points to the object, allowing the method to access the object itself using the methods of the object's own class. Any modification to the object will remain after the return from the method. This will be discussed in Section A.7.

The Class Math

Class Math is part of the Java language, and it provides a collection of methods that are useful for performing common mathematical operations. These are all **static** methods, so the prefix Math. is required in order to invoke a method of this class.

Table A.5 shows some of these methods. The first column shows the result type for each method followed by its *signature* (the method name and the argument types). For example, for method `ceil`, the first column shows that the method returns a type **`double`** result and has a type **`double`** argument. The data type *numeric* means that any of the numeric types can be used.

Escape Sequences

The main method in the following `SquareRoots` class contains a loop that displays the first 10 integers and their square roots in two columns (see Figure A.3).

```
public class SquareRoots {
    public static void main(String[] args) {
        System.out.println("n \tsquare root");
        for (int n = 1; n <= 10; n++) {
            System.out.println(n + "\t" + Math.sqrt(n));
        }
    }
}
```

TABLE A.5
Class `Math` Methods

Method	Behavior
`static` *numeric* `abs(`*numeric*`)`	Returns the absolute value of its *numeric* argument (the result type is the same as the argument type)
`static double ceil(double)`	Returns the smallest whole number that is not less than its argument
`static double cos(double)`	Returns the trigonometric cosine of its argument (an angle in radians)
`static double exp(double)`	Returns the exponential number *e* (i.e., 2.718 . . .) raised to the power of its argument
`static double floor(double)`	Returns the largest whole number that is not greater than its argument
`static double log(double)`	Returns the natural logarithm of its argument
`static` *numeric* `max(`*numeric*`,` *numeric*`)`	Returns the larger of its *numeric* arguments (the result type is the same as the argument types)
`static` *numeric* `min(`*numeric*`,` *numeric*`)`	Returns the smaller of its *numeric* arguments (the result type is the same as the argument type)
`static double pow(double, double)`	Returns the value of the first argument raised to the power of the second argument
`static double random()`	Returns a random number greater than or equal to 0.0 and less than 1.0
`static double rint(double)`	Returns the closest whole number to its argument
`static long round(double)`	Returns the closest **`long`** to its argument
`static int round(float)`	Returns the closest **`int`** to its argument
`static double sin(double)`	Returns the trigonometric sine of its argument (an angle in radians)
`static double sqrt(double)`	Returns the square root of its argument
`static double tan(double)`	Returns the trigonometric tangent of its argument (an angle in radians)
`static double toDegrees(double)`	Converts its argument (in radians) to degrees
`static double toRadians(double)`	Converts its argument (in degrees) to radians

FIGURE A.3
Sample Run of Class SquareRoots

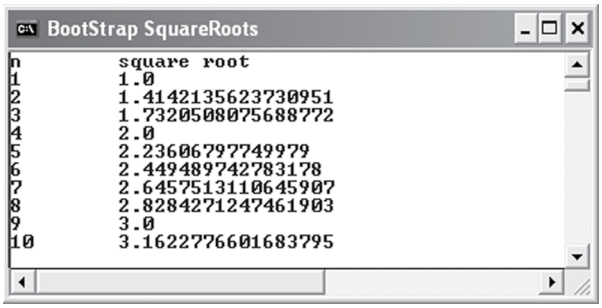


TABLE A.6
Escape Sequences

Sequence	Meaning
`\n`	Start a new output line
`\t`	Tab character
`\\`	Backslash character
`\"`	Double quote
`\'`	Single quote or apostrophe
`\u`*dddd*	The Unicode character whose code is *dddd* where each digit *d* is a hexadecimal digit in the range 0 to F (0–9, A–F)

The `println` statements use the *escape sequence* `\t`, the tab character, to align the column label `square root` with the numbers in the second output column (see Figure A.3). An escape sequence is a sequence of two characters beginning with the character `\`. Some escape sequences are used for special output control characters. Others are used to represent characters or symbols that have a special meaning in Java. For example, a double quote character by itself is a string delimiter, so we need to use the sequence `\"` to represent the double quote character in a string. Table A.6 lists some common escape sequences and their meaning.

The escape sequence that starts with `\u` represents a Unicode character. The character code uses four hexadecimal digits, where a hexadecimal digit is formed using four binary bits and ranges from 0 (all bits 0) to F (all four bits 1). The hexadecimal digit A corresponds to a decimal value of 10, and the hexadecimal digit F corresponds to a decimal value of 15.

EXERCISES FOR SECTION A.4

SELF-CHECK

1. Identify the escape sequences in the following string. Show how this line would be displayed. Which of the escape sequences could be replaced by the second character of the pair without changing the effect?

```
System.out.println("Jane\'s motto is \n\"semper fi\"\n, according to Jim");
```

PROGRAMMING

1. Write a Java program that displays all odd powers of 2 between 1 and 29. Display the power that 2 is being raised to, as well as the result, on each line. Use tab characters between numbers.
2. Write a Java program that displays n and the natural log of n for values of n of 1000, 2000, 4000, 8000, and so on. Display the first 20 lines for this sequence. Use tab characters between numbers.

A.5 The `String`, `StringBuilder`, `StringBuffer`, and StringJoiner Classes

In this section, we discuss four Java classes that are used to process sequences of characters. We begin with the `String` class.

The `String` Class

The `String` class defines a data type that is used to store a sequence of characters. Table A.7 describes some `String` class methods. The first column shows the result type for each method followed by its signature. For example, for method `charAt`, the first column shows that the method returns a type `char` result and has a type `int` argument. The second column describes what the method does. The phrase "this string" means the string to which the method is applied by the dot notation. If type `Object` is listed as an argument type in column 1, any kind of object can be an argument. (We discuss type `Object` in Chapter 1.)

TABLE A.7
Selected Methods in `java.lang.String`

Method	Behavior
`char charAt(int pos)`	Returns the character at position `pos`
`int compareTo(String)`	Returns a negative integer if this string's contents precede the argument string's contents in the dictionary; returns 0 if this string and the argument string have the same contents; returns a positive integer if this string's contents follow those of the argument string. This comparison is case-sensitive
`int compareToIgnoreCase(String)`	Returns a negative, zero, or positive integer according to whether this string's contents precede, match, or follow the argument string's contents in the dictionary, ignoring case
`boolean equals(Object)`	Returns **`true`** if this string's contents are the same as its argument string's contents
`boolean equalsIgnoreCase(String)`	Returns **`true`** if this string's contents are the same as the argument string's contents, ignoring case
`int indexOf(char)` `int indexOf(String)`	Returns the index within this string of the first occurrence of its character or string argument, or –1 if the argument is not found
`int indexOf(char, int index)` `int indexOf(String, int index)`	Returns the index within this string of the first occurrence of its first character or string argument, starting at the specified `index`

(Continued)

TABLE A.7
Continued

Method	Behavior
`int lastIndexOf(char)` `int lastIndexOf(String)`	Returns the index within this string of the rightmost occurrence of its character or string argument
`int lastIndexOf(char, int index)` `int lastIndexOf(String, int index)`	Returns the index within this string of the last occurrence of its first character or string argument, searching backward and stopping at the specified `index`
`int length()`	Returns the length of this string
`String replace(char oldChar, char newChar)`	Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`
`String substring(int start)`	Returns a new string that is a substring of this string, starting at position `start` and going to the end of the string
`String substring(int start, int end)`	Returns a new string that is a substring of this string, starting with the character at position `start` and ending with the character at position `end - 1`
`String toLowerCase()`	Returns a new string in which all of the letters in this string are converted to lowercase
`String toUpperCase()`	Returns a new string in which all of the letters in this string are converted to uppercase
`String trim()`	Returns a new string in which all the white space is removed from both ends of this string
`static String format(String format, Object... args)`	Returns a new string with the arguments `args` formatted as prescribed by the string `format`
`String[] split(String pattern)`	Separates the string into an array of tokens, where each token is delimited by a string that matches the regular expression `pattern`

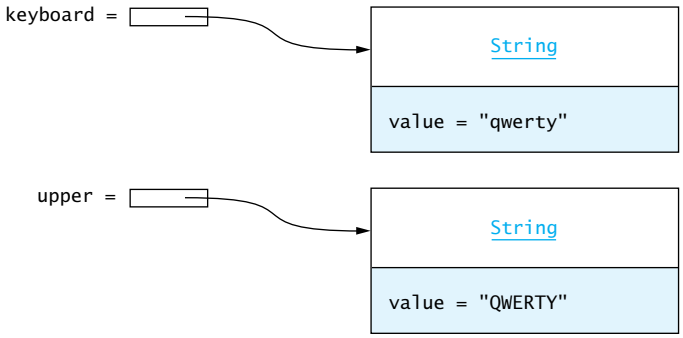
EXAMPLE A.6 Assume that (`keyboard type String`) contains `"qwerty"`. We evaluate several expressions:

- `keyboard.charAt(0) is 'q'.`
- `keyboard.length() is 6.`
- `keyboard.indexOf('o') is -1.`
- `keyboard.indexOf('y') is 5.`

The statement

```
String upper = keyboard.toUpperCase();
```

creates a new `String` object, referenced by the variable `upper`, that stores the character sequence `"QWERTY"`, but the `String` object referenced by `keyboard` is unchanged, as shown here.



Finally, the expression

```
keyboard.charAt(keyboard.length() - 1)
```

applies two instance methods to `keyboard`. The inner call, to method `length`, returns the value 6; the outer call, to method `charAt`, returns `y`, the last character in the string (at position 5).

The method `substring` returns a new string containing a portion of the `String` object to which it is applied. If it is called with just one argument, the contents of the string returned will be all characters from its argument position to the end of the string. If it is called with two arguments, the contents of the string returned will be all characters from its first argument position up to, but excluding, the character at its second argument position. However, the string to which method `substring` is applied is not changed.

EXAMPLE A.7 The expression

```
keyboard.substring(0, keyboard.length() - 1)
```

returns a new string `"qwert"` consisting of all characters except for the last character in the string referenced by `keyboard`. The contents of `keyboard` are unchanged.

Strings Are Immutable

Strings are different from most other Java objects in that they are *immutable*. What this means is that you cannot modify a `String` object. If you attempt to do so, Java will create a new object that contains the modified character sequence. The following statements create a new `String` object storing the character sequence `"Koffman, Elliot"` that is referenced by `myName` (indicated by the blue arrow in Figure A.4). The original `String` object still exists (at least temporarily) and contains the character sequence `"Elliot Koffman"`, but it is no longer referenced by `myName` (indicated by the dashed arrow in Figure A.4).

```
String myName = "Elliot Koffman";
myName = myName.substring(7) + ", " +
myName.substring(0, 6);
```

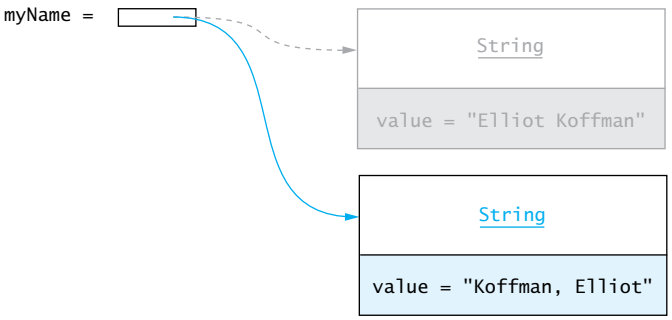


FIGURE A.4 Old and New Strings Referenced by myName

PITFALL

Attempting to Change a Character in a String

You might try to change the first character in `myName` using either of the following statements:

```
myName.charAt(0) = 'X'; // Invalid attempt to change character at
                        // position 0
myName[0] = 'X';        // Invalid attempt to treat string as array
```

Both statements cause syntax errors. The first statement will not work because method `charAt` returns a value, but a variable must be on the left side of the assignment operator. The second statement attempts to change the first character in a string by treating it as an array of characters. You cannot do this in Java.

The Garbage Collector

Storage space for objects that are no longer referenced is automatically reclaimed by the Java *garbage collector* so that the storage space can be reallocated and reused. The storage space occupied by the first `String` object in Figure A.4 will be reclaimed by the garbage collector. In other programming languages, the programmer is responsible for reclaiming any storage space that is no longer needed.

Comparing Objects

You can't use the relational (<, <=, >, >=) or equality operators (==, !=) to compare the values stored in strings or other objects. After the assignment

```
String anyName = new String(myName);
```

the condition (`anyName == myName`) would be **`false`**, even though these variables have the same contents. The reason is that the == operator compares the *addresses* stored in `anyName` and `myName`, and the String objects that are referenced by these variables have different addresses (see Figure A.5).

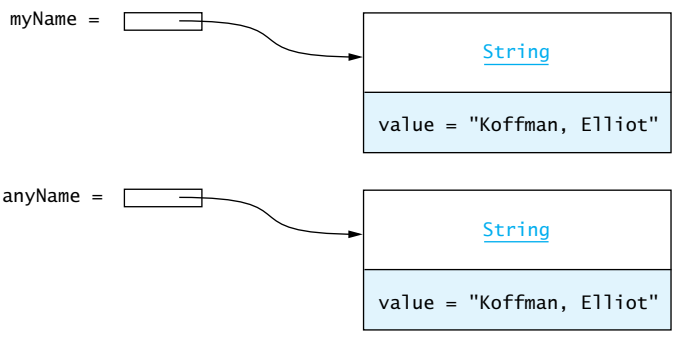


FIGURE A.5 Two String Objects at Different Addresses with the Same Contents

To compare the character sequences stored in two `String` objects, you need to use one of the Java `String` comparison methods: `equals`, `equalsIgnoreCase`, `compareTo`, or `compareToIgnoreCase`. In general, if you want to compare instances of classes that you write, you will need to write at least an `equals` and a `compareTo` method for that class.