```
                    maxChild = rightChild;
                }
                // If the parent is smaller than the larger child,
                if (table[parent].compareTo(table[maxChild]) < 0) {
                    // Swap the parent and child.
                    swap(table, parent, maxChild);
                    // Continue at the child level.
                    parent = maxChild;
                } else {   // Heap property is restored.
                    break; // Exit the loop.
                }
            }
        }
    }

    /** Swap the items in table[i] and table[j].
        @param table The array that contains the items
        @param i The index of one item
        @param j The index of the other item
     */
    private static <T extends Comparable<T>> void swap(T[] table,
                                                       int i, int j) {
        T temp = table[i];
        table[i] = table[j];
        table[j] = temp;
    }
}
```

EXERCISES FOR SECTION 8.8

SELF-CHECK

1. Build the heap from the numbers in the following list. How many exchanges were required? How many comparisons?

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

2. Shrink the heap from Question 1 to create the array in sorted order. How many exchanges were required? How many comparisons?

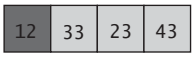
8.9 Quicksort

The next algorithm we will study is called *quicksort*. Developed by C. A. R. Hoare in 1962, it works in the following way: given an array with subscripts `first . . . last` to sort, quicksort rearranges this array into two parts so that all the elements in the left subarray are less than or equal to a specified value (called the *pivot*) and all the elements in the right subarray are greater than the pivot. The pivot is placed between the two parts. Thus, all of the elements on the left of the pivot value are smaller than all elements on the right of the pivot value, so the pivot value is in its correct position. By repeating this process on the two halves, the whole array becomes sorted.

As an example of this process, let's sort the following array:

We will assume that the first array element (44) is arbitrarily selected as the pivot value. A possible result of rearranging, or *partitioning,* the element values follows:

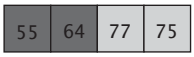
After the partitioning process, the pivot value, 44, is at its correct position. All values less than 44 are in the left subarray, and all values larger than 44 are in the right subarray, as desired. The next step would be to apply quicksort recursively to the two subarrays on either side of the pivot value, beginning with the left subarray (12, 33, 23, 43). Here is the result when 12 is the pivot value:



The pivot value is in the first position. Because the left subarray does not exist, the right subarray (33, 23, 43) is sorted next, resulting in the following situation:

The pivot value 33 is in its correct place, and the left subarray (23) and right subarray (43) have single elements, so they are sorted. At this point, we are finished sorting the left part of the original subarray, and quicksort is applied to the right subarray (55, 64, 77, 75). In the following array, all the elements that have been placed in their proper position are shaded dark.

If we use 55 for the pivot, its left subarray will be empty after the partitioning process and the right subarray 64, 77, 75 will be sorted next. If 64 is the pivot, the situation will be as follows, and we sort the right subarray (77, 75) next.



If 77 is the pivot and we move it where it belongs, we end up with the following array. Because the left subarray (75) has a single element, it is sorted and we are done.

Algorithm for Quicksort

The algorithm for quicksort follows. We will describe how to do the partitioning later. We assume that the indexes `first` and `last` are the endpoints of the array being sorted and that the index of the pivot after partitioning is `pivIndex`.

Algorithm for Quicksort

1. `if first < last` then
2. Partition the elements in the subarray `first . . . last` so that the pivot value is in its correct place (subscript `pivIndex`).
3. Recursively apply quicksort to the subarray `first . . . pivIndex - 1`.
4. Recursively apply quicksort to the subarray `pivIndex + 1 . . . last`.

Analysis of Quicksort

If the pivot value is a random value selected from the current subarray, then statistically it is expected that half of the items in the subarray will be less than the pivot and half will be greater than the pivot. If both subarrays always have the same number of elements (the best case), there will be log n levels of recursion. At each level, the partitioning process involves moving every element into its correct partition, so quicksort is $\mathbf{O}(n \log n)$, just like merge sort.

But what if the split is not 50–50? Let us consider the case where each split is 90–10. Instead of a 100-element array being split into two 50-element arrays, there will be one array with 90 elements and one with just 10. The 90-element array may be split 50–50, or it may also be split 90–10. In the latter case, there would be one array with 81 elements and one with just 9 elements. Generally, for random input, the splits will not be exactly 50–50, but neither will they all be 90–10. An exact analysis is difficult and beyond the scope of this book, but the running time will be bound by a constant $\times n \log n$.

There is one situation, however, where quicksort gives very poor behavior. If, each time we partition the array, we end up with a subarray that is empty, the other subarray will have one less element than the one just split (only the pivot value will be removed). Therefore, we will have n levels of recursive calls (instead of log n), and the algorithm will be $\mathbf{O}(n^2)$. Because of the overhead of recursive method calls (versus iteration), quicksort will take longer and require more extra storage on the run-time stack than any of the earlier quadratic algorithms. We will discuss a way to handle this situation later.

Code for Quicksort

Listing 8.8 shows the `QuickSort` class. The public method `sort` calls the recursive `quickSort` method, giving it the bounds of the `table` as the initial values of `first` and `last`. The two recursive calls in `quickSort` will cause the procedure to be applied to the subarrays that are separated by the value at `pivIndex`. If any subarray contains just one element (or zero elements), an immediate return will occur.

LISTING 8.8
QuickSort.java

```
/** Implements the quicksort algorithm. */
public class QuickSort {

    /** Sort the table using the quicksort algorithm.
        @pre table contains Comparable objects.
        @post table is sorted.
        @param table The array to be sorted
     */
    public static <T extends Comparable<T>> void sort(T[] table) {
        // Sort the whole table.
        quickSort(table, 0, table.length - 1);
    }

    /** Sort a part of the table using the quicksort algorithm.
        @post The part of table from first through last is sorted.
        @param table The array to be sorted
        @param first The index of the low bound
        @param last The index of the high bound
     */
```

```
    private static <T extends Comparable<T>> void quickSort(T[] table,
                                                            int first, int last) {
        if (first < last) {  // There is data to be sorted.
            // Partition the table.
            int pivIndex = partition(table, first, last);
            // Sort the left half.
            quickSort(table, first, pivIndex - 1);
            // Sort the right half.
            quickSort(table, pivIndex + 1, last);
        }
    }
    // Insert partition method. See Listing 8.9
    . . .
}
```

Algorithm for Partitioning

The `partition` method selects the pivot and performs the partitioning operation. When we are selecting the pivot, it does not really matter which element is the pivot value (if the arrays are randomly ordered to begin with). For simplicity we chose the element with subscript `first`. We then begin searching for the first value at the left end of the subarray that is greater than the pivot value. When we find it, we search for the first value at the right end of the subarray that is less than or equal to the pivot value. These two values are exchanged, and we repeat the search and exchange operations. This is illustrated in Figure 8.12, where `up` points to the first value greater than the pivot and `down` points to the first value less than or equal to the pivot value. The elements less than the pivot are shaded dark, and the elements greater than the pivot are in gray.

The value 75 is the first value at the left end of the array that is larger than 44, and 33 is the first value at the right end that is less than or equal to 44, so these two values are exchanged. The indexes `up` and `down` are advanced again, as shown in Figure 8.13.

The value 55 is the next value at the left end that is larger than 44, and 12 is the next value at the right end that is less than or equal to 44, so these two values are exchanged, and `up` and `down` are advanced again, as shown in Figure 8.14.

After the second exchange, the first five array elements contain the pivot value and all values less than or equal to the pivot; the last four elements contain all values larger than the pivot. The value 55 is selected once again by `up` as the next element larger than the pivot; 12 is selected by `down` as the next element less than or equal to the pivot. Since `up` has now "passed" `down`, these values are not exchanged. Instead, the pivot value (subscript `first`) and the value at position `down` are exchanged. This puts the pivot value in its proper position (the new subscript is `down`) as shown in Figure 8.15.

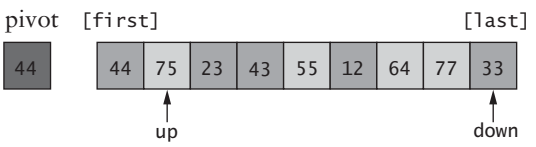


FIGURE 8.12
Locating First Values to Exchange

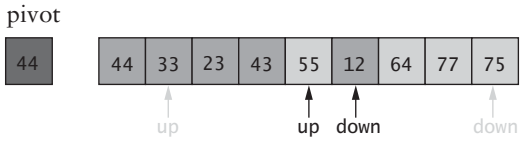


FIGURE 8.13
Array after the First Exchange

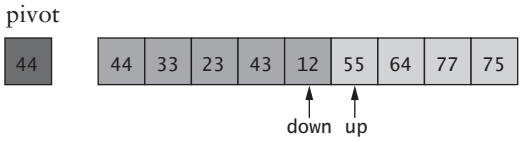


FIGURE 8.14
Array after the Second Exchange

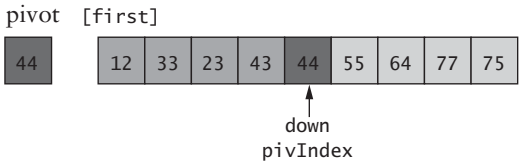


FIGURE 8.15
Array after the Pivot Is Inserted

The partition process is now complete, and the value of down is returned to the pivot index pivIndex. Method quickSort will be called recursively to sort the left subarray and the right subarray. The algorithm for partition follows:

Algorithm for `partition` Method

1. Define the pivot value as the contents of table[first].
2. Initialize up to first and down to last.
3. **do**
4. Increment up until up selects the first element greater than the pivot value or up has reached last.
5. Decrement down until down selects the first element less than or equal to the pivot value or down has reached first.
6. **if** up < down then
7. Exchange table[up] and table[down].
8. **while** up is to the left of down
9. Exchange table[first] and table[down].
10. Return the value of down to pivIndex.

Code for `partition`

The code for partition is shown in Listing 8.9. The **while** statement:

```
while ((up < last) && (pivot.compareTo(table[up]) >= 0)) {
    up++;
}
```

advances the index up until it is equal to last or until it references an item in table that is greater than the pivot value. Similarly, the **while** statement:

```
while (pivot.compareTo(table[down]) < 0)) {
    down--;
}
```

moves the index down until it references an item in table that is less than or equal to the pivot value. The **do**–**while** condition

```
(up < down)
```

ensures that the partitioning process will continue while up is to the left of down.

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

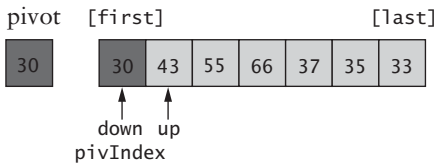
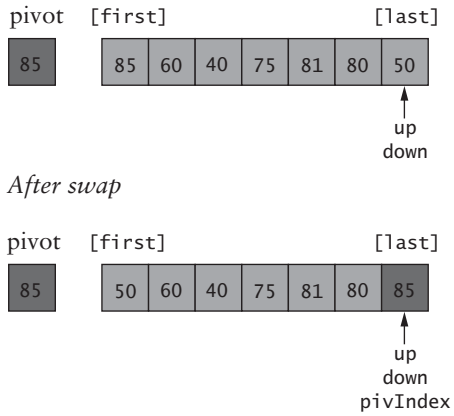


FIGURE 8.17
Values of **up**, **down**, and `pivIndex` If the Pivot Is the Largest Value



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 dark 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.

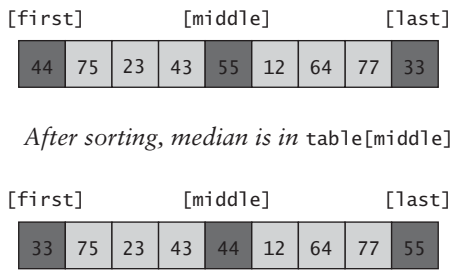


FIGURE 8.18 Sorting First, Middle, and Last Elements in Array

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.

Code for 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].
}
```

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.

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.

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