CHAPTER

23

Sorting

Objectives

- To study and analyze time complexity of various sorting algorithms (§§23.2–23.7).
- To design, implement, and analyze insertion sort (§23.2).
- To design, implement, and analyze bubble sort (§23.3).
- To design, implement, and analyze merge sort (§23.4).
- To design, implement, and analyze quick sort (§23.5).
- To design and implement a binary heap (§23.6).
- To design, implement, and analyze heap sort (§23.6).
- To design, implement, and analyze bucket and radix sorts (§23.7).
- To design, implement, and analyze external sort for files that have a large amount of data (§23.8).

23.1 Introduction

Key Point

Sorting algorithms are good examples for studying algorithm design and analysis.

When president Barack Obama visited Google in 2007, the Google CEO Eric Schmidt asked Obama the most efficient way to sort a million 32-bit integers (www.youtube.com/watch?v=k4RRi_ntQc8). Obama answered that the bubble sort would be the wrong way to go. Was he right? We will examine different sorting algorithms in this chapter and see if he was correct.

why study sorting?

Sorting is a classic subject in computer science. There are three reasons to study sorting algorithms.

- First, sorting algorithms illustrate many creative approaches to problem solving, and these approaches can be applied to solve other problems.
- Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, methods, and arrays.
- Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

what data to sort?

The data to be sorted might be integers, doubles, characters, or objects. Section 7.11, Sorting Arrays, presented selection sort. The selection-sort algorithm was extended to sort an array of objects in Section 19.5, Case Study: Sorting an Array of Objects. The Java API contains several overloaded sort methods for sorting primitive-type values and objects in the `java.util.Arrays` and `java.util.Collections` classes. For simplicity, this chapter assumes

1. data to be sorted are integers,
2. data are stored in an array, and
3. data are sorted in ascending order.

The programs can be easily modified to sort other types of data, to sort in descending order, or to sort data in an `ArrayList` or a `LinkedList`.

There are many algorithms for sorting. You have already learned selection sort. This chapter introduces insertion sort, bubble sort, merge sort, quick sort, bucket sort, radix sort, and external sort.

23.2 Insertion Sort

Key Point

The insertion-sort algorithm sorts a list of values by repeatedly inserting a new element into a sorted sublist until the whole list is sorted.

insertion-sort animation on Companion Website

Figure 23.1 shows how to sort a list {`2`, `9`, `5`, `4`, `8`, `1`, `6`} using insertion sort. For an interactive demo on how insertion sort works, go to liveexample.pearsoncmg.com/dsanimation/InsertionSortNeweBook.html.

The algorithm can be described as follows:

```
for (int i = 1; i < list.length; i++) {
  insert list[i] into a sorted sublist list[0..i-1] so that
  list[0..i] is sorted.
}
```

To insert `list[i]` into `list[0..i-1]`, save `list[i]` into a temporary variable, say `currentElement`. Move `list[i-1]` to `list[i]` if `list[i-1] > currentElement`, move `list[i-2]` to `list[i-1]` if `list[i-2] > currentElement`, and so on, until `list[i-k] <= currentElement` or `k > i` (we pass the first element of the sorted list). Assign `currentElement` to `list[i-k+1]`. For example, to insert `4` into {`2`, `5`, `9`} in Step 4 in Figure 23.2, move `list[2]` (`9`) to `list[3]` since `9 > 4` and move `list[1]` (`5`) to `list[2]` since `5 > 4`. Finally, move `currentElement` (`4`) to `list[1]`.

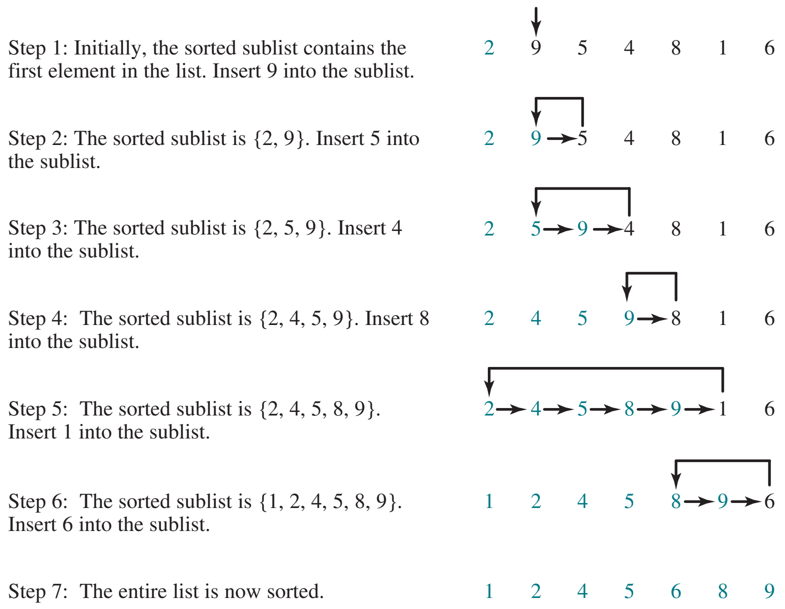

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.
2 9 5 4 8 1 6
Step 2: The sorted sublist is {2, 9}. Insert 5 into the sublist.
2 9 5 4 8 1 6
Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.
2 5 9 4 8 1 6
Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 into the sublist.
2 4 5 9 8 1 6
Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 into the sublist.
2 4 5 8 9 1 6
Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.
1 2 4 5 8 9 6
Step 7: The entire list is now sorted.
1 2 4 5 6 8 9

FIGURE 23.1 Insertion sort repeatedly inserts a new element into a sorted sublist.

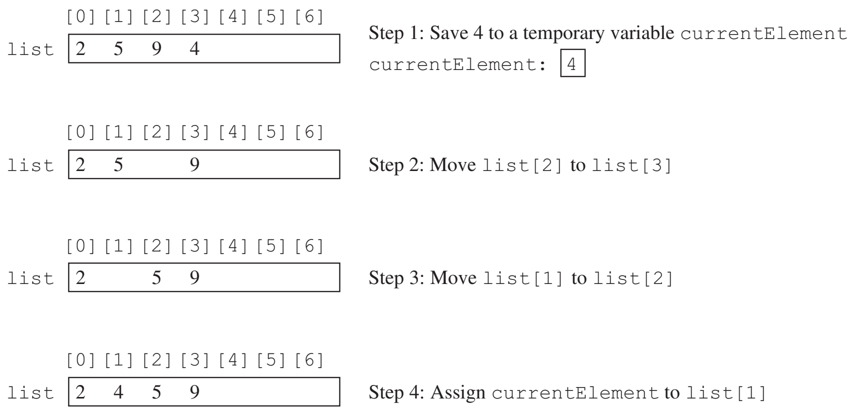


FIGURE 23.2 A new element is inserted into a sorted sublist.

The algorithm can be expanded and implemented as in Listing 23.1.

LISTING 23.1 InsertionSort.java

```java
public class InsertionSort {
  /** The method for sorting the numbers */
  public static void insertionSort(int[] list) {
    for (int i = 1; i < list.length; i++) {
      /** Insert list[i] into a sorted sublist list[0..i-1] so that
          list[0..i] is sorted. */
      int currentElement = list[i];
      int k;
      for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
        list[k + 1] = list[k];
      }

      // Insert the current element into list[k + 1]
      list[k + 1] = currentElement;
    }
  }
}
```

shift

insert

The `insertionSort(int[] list)` method sorts an array of `int` elements. The method is implemented with a nested `for` loop. The outer loop (with the loop control variable `i`) (line 4) is iterated in order to obtain a sorted sublist, which ranges from `list[0]` to `list[i]`. The inner loop (with the loop control variable `k`) inserts `list[i]` into the sublist from `list[0]` to `list[i-1]`.

To better understand this method, trace it with the following statements:

```
int[] list = {1, 9, 4, 6, 5, -4};
InsertionSort.insertionSort(list);
```

insertion-sort time complexity

The insertion-sort algorithm presented here sorts a list of elements by repeatedly inserting a new element into a sorted partial array until the whole array is sorted. At the *k*th iteration, to insert an element into an array of size *k*, it may take *k* comparisons to find the insertion position and *k* moves to insert the element. Let $T(n)$ denote the complexity for insertion sort, and c denote the total number of other operations such as assignments and additional comparisons in each iteration. Thus,

$$\begin{aligned} T(n) &= (2 + c) + (2 \times 2 + c) + \cdots + (2 \times (n - 1) + c) \\ &= 2(1 + 2 + \cdots + n - 1) + c(n - 1) \\ &= 2\frac{(n-1)n}{2} + cn - c = n^2 - n + cn - c \\ &= O(n^2) \end{aligned}$$

Therefore, the complexity of the insertion-sort algorithm is $O(n^2)$. Hence, the selection and insertion sorts are of the same time complexity.

Check Point

23.2.1 Describe how an insertion sort works. What is the time complexity for an insertion sort?

23.2.2 Use Figure 23.1 as an example to show how to apply an insertion sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.2.3 If a list is already sorted, how many comparisons will the `insertionSort` method perform?

23.3 Bubble Sort

Key Point

A bubble sort sorts the array in multiple passes. Each pass successively swaps the neighboring elements if the elements are not in order.

bubble sort

The bubble-sort algorithm makes several passes through the array. On each pass, successive neighboring pairs are compared. If a pair is in decreasing order, its values are swapped; otherwise, the values remain unchanged. The technique is called a *bubble sort* or *sinking sort* because the smaller values gradually "bubble" their way to the top and the larger values sink to the bottom. After the first pass, the last element becomes the largest in the array. After the second pass, the second-to-last element becomes the second largest in the array. This process is continued until all elements are sorted.

bubble-sort illustration

Figure 23.3a shows the first pass of a bubble sort on an array of six elements (2 9 5 4 8 1). Compare the elements in the first pair (2 and 9) and no swap is needed because they are already in order. Compare the elements in the second pair (9 and 5) and swap 9 with 5 because 9 is greater than 5. Compare the elements in the third pair (9 and 4) and swap 9 with 4. Compare the elements in the fourth pair (9 and 8) and swap 9 with 8. Compare the elements in the fifth pair (9 and 1) and swap 9 with 1. The pairs being compared are highlighted and the numbers already sorted are italicized in Figure 23.3. For an interactive demo on how bubble sort works, go to liveexample.pearsoncmg.com/dsanimation/BubbleSortNeweBook.html.

bubble sort animation on the Companion Website

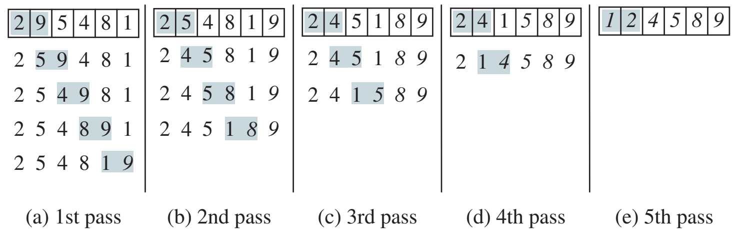


FIGURE 23.3 Each pass compares and orders the pairs of elements sequentially.

The first pass places the largest number (9) as the last in the array. In the second pass, as shown in Figure 23.3b, you compare and order pairs of elements sequentially. There is no need to consider the last pair because the last element in the array is already the largest. In the third pass, as shown in Figure 23.3c, you compare and order pairs of elements sequentially except the last two elements because they are already in order. Thus, in the *k*th pass, you don't need to consider the last $k - 1$ elements because they are already ordered.

The algorithm for a bubble sort is described in Listing 23.2. algorithm

LISTING 23.2 Bubble-Sort Algorithm

```
for (int k = 1; k < list.length; k++) {
  // Perform the kth pass
  for (int i = 0; i < list.length - k; i++) {
    if (list[i] > list[i + 1])
      swap list[i] with list[i + 1];
  }
}
```

Note if no swap takes place in a pass, there is no need to perform the next pass because all the elements are already sorted. You can use this property to improve the algorithm in Listing 23.2, as in Listing 23.3.

LISTING 23.3 Improved Bubble-Sort Algorithm

```
boolean needNextPass = true;
for (int k = 1; k < list.length && needNextPass; k++) {
  // Array may be sorted and next pass not needed
  needNextPass = false;
  // Perform the kth pass
  for (int i = 0; i < list.length - k; i++) {
    if (list[i] > list[i + 1]) {
      swap list[i] with list[i + 1];
      needNextPass = true; // Next pass still needed
    }
  }
}
```

The algorithm can be implemented in Listing 23.4.

LISTING 23.4 BubbleSort.java

```
public class BubbleSort {
  /** Bubble sort method */
  public static void bubbleSort(int[] list) {
    boolean needNextPass = true;

    for (int k = 1; k < list.length && needNextPass; k++) {
      // Array may be sorted and next pass not needed
      needNextPass = false;
      for (int i = 0; i < list.length - k; i++) {
```

perform one pass

```
        if (list[i] > list[i + 1]) {
          // Swap list[i] with list[i + 1]
          int temp = list[i];
          list[i] = list[i + 1];
          list[i + 1] = temp;

          needNextPass = true; // Next pass still needed
        }
      }
    }
  }

  /** A test method */
  public static void main(String[] args) {
    int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
    bubbleSort(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }

}
```

```
-2 1 2 2 3 3 5 6 12 14
```

bubble-sort time complexity

In the best case, the bubble-sort algorithm needs just the first pass to find that the array is already sorted—no next pass is needed. Since the number of comparisons is $n - 1$ in the first pass, the best-case time for a bubble sort is $O(n)$.

In the worst case, the bubble-sort algorithm requires $n - 1$ passes. The first pass makes $n - 1$ comparisons, the second pass makes $n - 2$ comparisons, and so on; the last pass makes 1 comparison. Thus, the total number of comparisons is as follows:

$$(n - 1) + (n - 2) + \cdots + 2 + 1$$

$$= \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

Therefore, the worst-case time for a bubble sort is $O(n^2)$.

Check Point

23.3.1 Describe how a bubble sort works. What is the time complexity for a bubble sort?

23.3.2 Use Figure 23.3 as an example to show how to apply a bubble sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.3.3 If a list is already sorted, how many comparisons will the **bubbleSort** method perform?

23.4 Merge Sort

Key Point

The merge-sort algorithm can be described recursively as follows: The algorithm divides the array into two halves and applies a merge sort on each half recursively. After the two halves are sorted, the algorithm then merges them.

merge sort

The algorithm for a *merge sort* is given in Listing 23.5.

LISTING 23.5 Merge-Sort Algorithm

base condition
sort first half
sort second half
merge two halves

```
public static void mergeSort(int[] list) {
  if (list.length > 1) {
    mergeSort(list[0 ... list.length / 2]);
    mergeSort(list[list.length / 2 + 1 ... list.length]);
    merge list[0 ... list.length / 2] with
```

```
        list[list.length / 2 + 1 ... list.length];
    }
  }
```

Figure 23.4 illustrates a merge sort of an array of eight elements (2 9 5 4 8 1 6 7). The original array is split into (2 9 5 4) and (8 1 6 7). Apply a merge sort on these two subarrays recursively to split (2 9 5 4) into (2 9) and (5 4) and (8 1 6 7) into (8 1) and (6 7). This process continues until the subarray contains only one element. For example, array (2 9) is split into the subarrays (2) and (9). Since array (2) contains a single element, it cannot be further split. Now merge (2) with (9) into a new sorted array (2 9) and (5) with (4) into a new sorted array (4 5). Merge (2 9) with (4 5) into a new sorted array (2 4 5 9) and finally merge (2 4 5 9) with (1 6 7 8) into a new sorted array (1 2 4 5 6 7 8 9).

merge-sort illustration

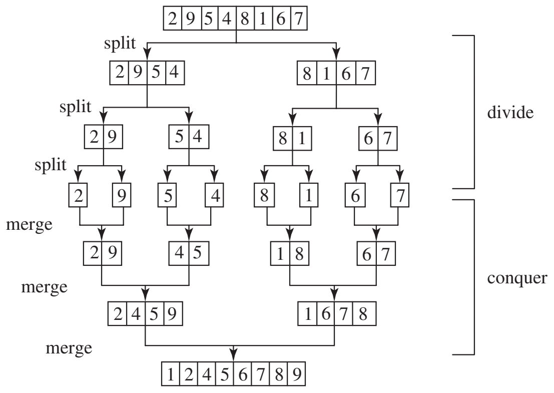


FIGURE 23.4 Merge sort employs a divide-and-conquer approach to sort the array.

The recursive call continues dividing the array into subarrays until each subarray contains only one element. The algorithm then merges these small subarrays into larger sorted subarrays until one sorted array results.

The merge-sort algorithm is implemented in Listing 23.6.

LISTING 23.6 MergeSort.java

```
public class MergeSort {
  /** The method for sorting the numbers */
  public static void mergeSort(int[] list) {
    if (list.length > 1) {                                    // base case
      // Merge sort the first half
      int[] firstHalf = new int[list.length / 2];
      System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
      mergeSort(firstHalf);                                   // sort first half

      // Merge sort the second half
      int secondHalfLength = list.length - list.length / 2;
      int[] secondHalf = new int[secondHalfLength];
      System.arraycopy(list, list.length / 2,
        secondHalf, 0, secondHalfLength);
      mergeSort(secondHalf);                                  // sort second half

      // Merge firstHalf with secondHalf into list
      merge(firstHalf, secondHalf, list);                     // merge two halves
    }
  }

  /** Merge two sorted lists */
  public static void merge(int[] list1, int[] list2, int[] temp) {
    int current1 = 0; // Current index in list1
```

list1 to temp

list2 to temp

rest of list1 to temp

rest of list2 to temp

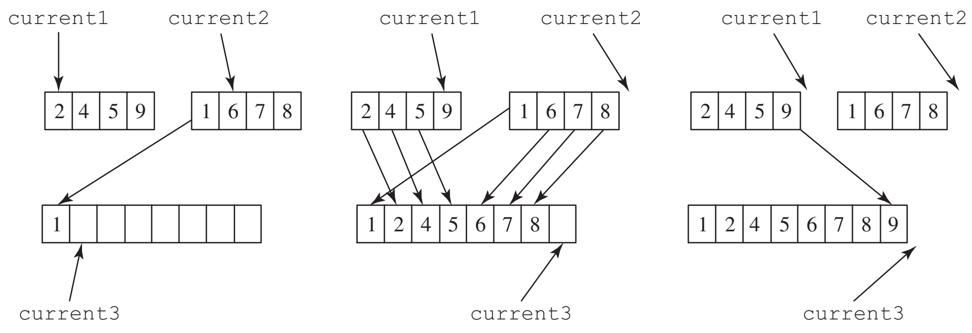
```
25       int current2 = 0; // Current index in list2
26       int current3 = 0; // Current index in temp
27
28       while (current1 < list1.length && current2 < list2.length) {
29         if (list1[current1] < list2[current2])
30           temp[current3++] = list1[current1++];
31         else
32           temp[current3++] = list2[current2++];
33       }
34
35       while (current1 < list1.length)
36         temp[current3++] = list1[current1++];
37
38       while (current2 < list2.length)
39         temp[current3++] = list2[current2++];
40     }
41
42     /** A test method */
43     public static void main(String[] args) {
44       int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
45       mergeSort(list);
46       for (int i = 0; i < list.length; i++)
47         System.out.print(list[i] + " ");
48     }
49   }
```

The **mergeSort** method (lines 3–20) creates a new array **firstHalf**, which is a copy of the first half of **list** (line 7). The algorithm invokes **mergeSort** recursively on **firstHalf** (line 8). The length of the **firstHalf** is **list.length / 2** and the length of the **secondHalf** is **list.length – list.length / 2**. The new array **secondHalf** was created to contain the second part of the original array **list**. The algorithm invokes **mergeSort** recursively on **secondHalf** (line 15). After **firstHalf** and **secondHalf** are sorted, they are merged to **list** (line 18). Thus, array **list** is now sorted.

The **merge** method (lines 23–40) merges two sorted arrays **list1** and **list2** into array **temp**. **current1** and **current2** point to the current element to be considered in **list1** and **list2** (lines 24–26). The method repeatedly compares the current elements from **list1** and **list2** and moves the smaller one to **temp**. **current1** is increased by **1** (line 30) if the smaller one is in **list1**, and **current2** is increased by **1** (line 32) if the smaller one is in **list2**. Finally, all the elements in one of the lists are moved to **temp**. If there are still unmoved elements in **list1**, copy them to **temp** (lines 35–36). If there are still unmoved elements in **list2**, copy them to **temp** (lines 38–39).

merge animation on Companion Website

Figure 23.5 illustrates how to merge the two arrays **list1** (2 4 5 9) and **list2** (1 6 7 8). Initially, the current elements to be considered in the arrays are 2 and 1. Compare



(a) After moving 1 to temp

(b) After moving all the elements in list2 to temp

(c) After moving 9 to temp

FIGURE 23.5 Two sorted arrays are merged into one sorted array.

them and move the smaller element 1 to **temp**, as shown in Figure 23.5a. **current2** and **current3** are increased by 1. Continue to compare the current elements in the two arrays and move the smaller one to **temp** until one of the arrays is completely moved. As shown in Figure 23.5b, all the elements in **list2** are moved to **temp** and **current1** points to element 9 in **list1**. Copy 9 to **temp**, as shown in Figure 23.5c. For an interactive demo on how merge works, go to liveexample.pearsoncmg.com/dsanimation/MergeSortNeweBook.html.

The **mergeSort** method creates two temporary arrays (lines 6 and 12) during the dividing process, copies the first half and the second half of the array into the temporary arrays (lines 7 and 13), sorts the temporary arrays (lines 8 and 15), then merges them into the original array (line 18), as shown in Figure 23.6a. You can rewrite the code to recursively sort the first half of the array and the second half of the array without creating new temporary arrays, then merge the two arrays into a temporary array and copy its contents to the original array, as shown in Figure 23.6b. This is left for you to do in Programming Exercise 23.20.

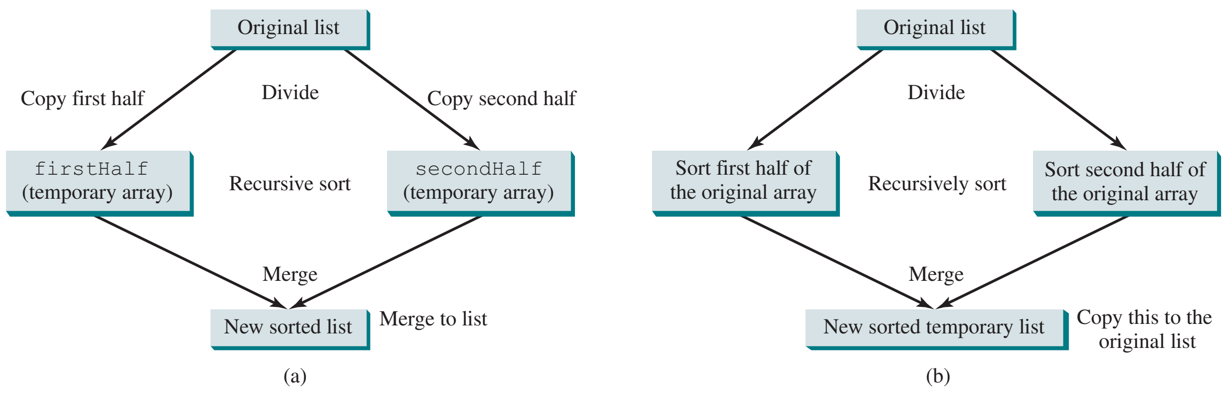

Original list
Divide
Copy first half
Copy second half
firstHalf (temporary array)
Recursive sort
secondHalf (temporary array)
Merge
New sorted list
Merge to list
(a)
Original list
Divide
Sort first half of the original array
Recursively sort
Sort second half of the original array
Merge
New sorted temporary list
Copy this to the original list
(b)

FIGURE 23.6 Temporary arrays are created to support a merge sort.

Note

A merge sort can be implemented efficiently using parallel processing. See Section 32.16, Parallel Programming, for a parallel implementation of a merge sort.

merge-sort time complexity

Let us analyze the running time of merge sort now. Let $T(n)$ denote the time required for sorting an array of n elements using a merge sort. Without loss of generality, assume n is a power of 2. The merge-sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, then merges the subarrays. Therefore,

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + mergetime$$

The first $T\left(\frac{n}{2}\right)$ is the time for sorting the first half of the array and the second $T\left(\frac{n}{2}\right)$ is the time for sorting the second half. To merge two subarrays, it takes at most $n - 1$ comparisons to compare the elements from the two subarrays, and n moves to move elements to the temporary array. Thus, the total time is $2n - 1$. Therefore,

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2n - 1 = O(n \log n)$$

$O(n \text{ log} n)$ merge sort

The complexity of a merge sort is $O(n \log n)$. This algorithm is better than selection sort, insertion sort, and bubble sort because the time complexity of these algorithms is $O(n^2)$. The **sort** method in the `java.util.Arrays` class is implemented using a variation of the merge-sort algorithm.

Check Point

23.4.1 Describe how a merge sort works. What is the time complexity for a merge sort?

23.4.2 Use Figure 23.4 as an example to show how to apply a merge sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.4.3 What is wrong if lines 6–15 in Listing 23.6, MergeSort.java, are replaced by the following code?

```
// Merge sort the first half
int[] firstHalf = new int[list.length /  2 + 1];
System.arraycopy(list, 0, firstHalf, 0, list.length / 2 + 1);
mergeSort(firstHalf);

// Merge sort the second half
int secondHalfLength = list.length - list.length / 2 - 1;
int[] secondHalf = new int[secondHalfLength];
System.arraycopy(list, list.length / 2 + 1,
  secondHalf, 0, secondHalfLength);
mergeSort(secondHalf);
```

23.5 Quick Sort

Key Point

quick sort

A **`quick sort`** *works as follows: The algorithm selects an element, called the pivot, in the array. It divides the array into two parts so all the elements in the first part are less than or equal to the pivot, and all the elements in the second part are greater than the pivot. The quick-sort algorithm is then recursively applied to the first part and then the second part.*

The quick-sort algorithm, developed by C.A.R. Hoare in 1962, is described in Listing 23.7.

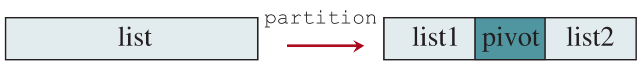
Listing 23.7 Quick-Sort Algorithm

base condition
select the pivot
partition the list
sort first part
sort second part

```
public static void quickSort(int[] list) {
  if (list.length > 1) {
    select a pivot;
    partition list into list1 and list2 such that
      all elements in list1 <= pivot and
      all elements in list2 > pivot;
    quickSort(list1);
    quickSort(list2);
  }
}
```

how to partition

Each partition places the pivot in the right place. It divides the list into two sublists as shown in the following figure.



The selection of the pivot affects the performance of the algorithm. Ideally, the algorithm should choose the pivot that divides the two parts evenly. For simplicity, assume that the first element in the array is chosen as the pivot. (Programming Exercise 23.4 proposes an alternative strategy for selecting the pivot.)

quick-sort illustration

Figure 23.7 illustrates how to sort an array (5 2 9 3 8 4 0 1 6 7) using quick sort. Choose the first element, 5, as the pivot. The array is partitioned into two parts, as shown in

Figure 23.7b. The highlighted pivot is placed in the right place in the array. Apply quick sort on two subarrays (4 2 1 3 0) then (8 9 6 7). The pivot 4 partitions (4 2 1 3 0) into just one subarrays (0 2 1 3), as shown in Figure 23.7c. Apply quick sort on (0 2 1 3). The pivot 0 partitions it into just one subarrays (2 1 3), as shown in Figure 23.7d. Apply quick sort on (2 1 3). The pivot 2 partitions it into (1) and (3), as shown in Figure 23.7e. Apply quick sort on (1). Since the array contains just one element, no further partition is needed.

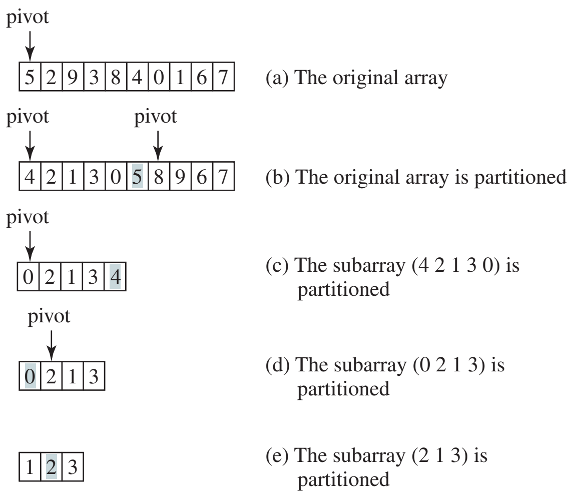

pivot
5 2 9 3 8 4 0 1 6 7
(a) The original array
pivot
pivot
4 2 1 3 0 5 8 9 6 7
(b) The original array is partitioned
pivot
0 2 1 3 4
(c) The subarray (4 2 1 3 0) is partitioned
pivot
0 2 1 3
(d) The subarray (0 2 1 3) is partitioned
1 2 3
(e) The subarray (2 1 3) is partitioned

FIGURE 23.7 The quick-sort algorithm is recursively applied to subarrays.

The quick-sort algorithm is implemented in Listing 23.8. There are two overloaded `quickSort` methods in the class. The first method (line 2) is used to sort an array. The second is a helper method (line 6) that sorts a subarray with a specified range.

LISTING 23.8 QuickSort.java

```
 1 public class QuickSort {
 2   public static void quickSort(int[] list) {
 3     quickSort(list, 0, list.length - 1);
 4   }
 5
 6   public static void quickSort(int[] list, int first, int last) {
 7     if (last > first) {
 8       int pivotIndex = partition(list, first, last);
 9       quickSort(list, first, pivotIndex -  1);
10       quickSort(list, pivotIndex + 1, last);
11     }
12   }
13
14   /** Partition the array list[first..last] */
15   public static int partition(int[] list, int first, int last) {
16     int pivot = list[first]; // Choose the first element as the pivot
17     int low = first + 1; // Index for forward search
18     int high = last; // Index for backward search
19
20     while (high > low) {
21       // Search forward from left
22       while (low <= high && list[low] <= pivot)
23         low++;
24
25       // Search backward from right
```

sort method

helper method

recursive call

forward

backward

swap

place pivot

pivot's new index

pivot's original index

```
26       while (low <= high && list[high] > pivot)
27         high--;
28 
29       // Swap two elements in the list
30       if (high > low) {
31         int temp = list[high];
32         list[high] = list[low];
33         list[low] = temp;
34       }
35     }
36 
37     while (high > first && list[high] >= pivot)
38       high--;
39 
40     // Swap pivot with list[high]
41     if (pivot > list[high]) {
42       list[first] = list[high];
43       list[high] = pivot;
44       return high;
45     }
46     else {
47       return first;
48     }
49   }
50 
51   /** A test method */
52   public static void main(String[] args) {
53     int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
54     quickSort(list);
55     for (int i = 0; i < list.length; i++)
56       System.out.print(list[i] + " ");
57   }
58 }
```

```
-2 1 2 2 3 3 5 6 12 14
```

The **partition** method (lines 15–49) partitions the array **list[first..last]** using the pivot. The first element in the partial array is chosen as the pivot (line 16). Initially, **low** points to the second element in the subarrays (line 17) and **high** points to the last element in the subarrays (line 18).

Starting from the left, the method searches forward in the array for the first element that is greater than the pivot (lines 22–23), then searches from the right backward for the first element in the array that is less than or equal to the pivot (lines 26–27). It then swaps these two elements and repeats the same search and swap operations until all the elements are searched in a **while** loop (lines 20–35).

The method returns the new index for the pivot that divides the subarrays into two parts if the pivot has been moved (line 44). Otherwise, it returns the original index for the pivot (line 47).

partition illustration

partition animation on Companion Website

Figure 23.8 illustrates how to partition an array (5 2 9 3 8 4 0 1 6 7). Choose the first element, 5, as the pivot. Initially, **low** is the index that points to element 2 and **high** points to element 7, as shown in Figure 23.8a. Advance index **low** forward to search for the first element (9) that is greater than the pivot, and move index **high** backward to search for the first element (1) that is less than or equal to the pivot, as shown in Figure 23.8b. Swap 9 with 1, as shown in Figure 23.8c. Continue the search and move **low** to point to element 8 and **high** to point to element 0, as shown in Figure 23.8d. Swap element 8 with 0, as shown in Figure 23.8e. Continue to move **low** until it passes **high**, as shown in Figure 23.8f. Now all the elements are examined. Swap the pivot with element 4 at index `high`. The final partition is shown in Figure 23.8g. The index of the pivot is returned when the method is finished. For an interactive demo on how partition works, go to liveexample.pearsoncmg.com/dsanimation/QuickSortNeweBook.html.

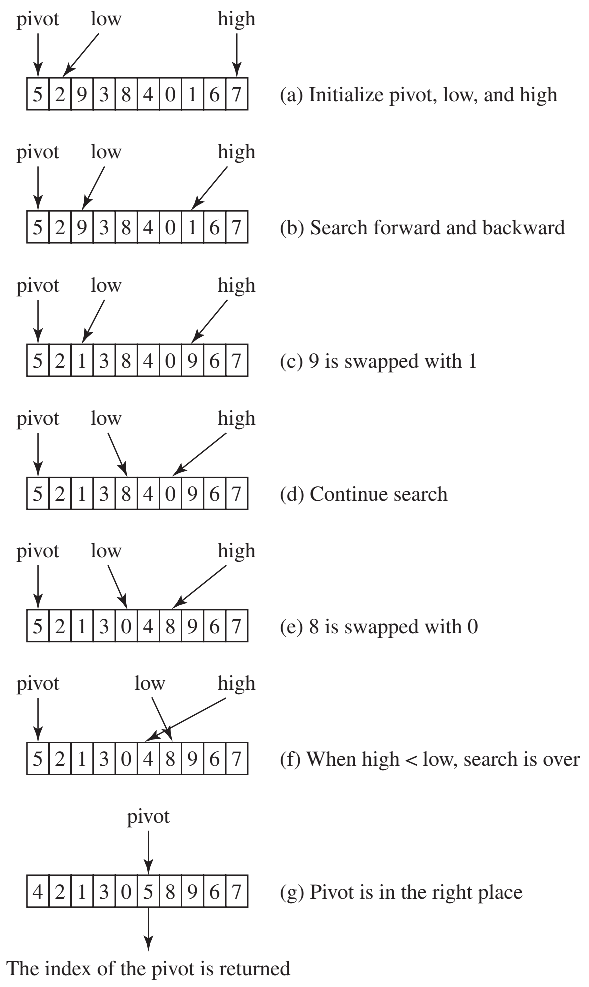


FIGURE 23.8 The `partition` method returns the index of the pivot after it is put in the correct place.

To partition an array of n elements, it takes n comparisons and n moves in the worst case. Thus, the time required for partition is $O(n)$.

$O(n)$ partition time

$O(n^2)$ worst-case time

In the worst case, the pivot divides the array each time into one big subarray with the other array empty. The size of the big subarray is one less than the one before divided. The algorithm requires $(n - 1) + (n - 2) + \cdots + 2 + 1 = O(n^2)$ time.

$O(n \log n)$ best-case time

In the best case, the pivot divides the array each time into two parts of about the same size. Let $T(n)$ denote the time required for sorting an array of n elements using quick sort. Thus,

recursive quick sort on two subarrays

partition time

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n.$$

Similar to the merge-sort analysis, $T(n) = O(n \log n)$.

$O(n \log n)$ average-case time

On the average, the pivot will not divide the array into two parts of the same size or one empty part each time. Statistically, the sizes of the two parts are very close. Therefore, the average time is $O(n \log n)$. The exact average-case analysis is beyond the scope of this book.

Both merge and quick sorts employ the divide-and-conquer approach. For merge sort, the bulk of the work is to merge two sublists, which takes place *after* the sublists are sorted. For quick sort, the bulk of the work is to partition the list into two sublists, which takes place *before* the sublists are sorted. Merge sort is more efficient than quick sort in the worst case, but the two are equally efficient in the average case. Merge sort requires a temporary array for sorting two subarrays. Quick sort does not need additional array space. Thus, quick sort is more space efficient than merge sort.

quick sort vs. merge sort

Check Point

23.5.1 Describe how quick sort works. What is the time complexity for a quick sort?

23.5.2 Why is quick sort more space efficient than merge sort?

23.5.3 Use Figure 23.7 as an example to show how to apply a quick sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

23.5.4 If lines 37–38 in the QuickSort program is removed, will it still work? Give a counter example to show that it will not work.

23.6 Heap Sort

Key Point

A heap sort uses a binary heap. It first adds all the elements to a heap and then removes the largest elements successively to obtain a sorted list.

heap sort
root
left subtree
right subtree
length
depth
leaf

Heap sorts use a binary heap, which is a complete binary tree. A binary tree is a hierarchical structure. It either is empty or it consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*. The *length* of a path is the number of the edges in the path. The *depth* of a node is the length of the path from the root to the node. A node is called a *leaf* if it does not have subtrees.

A *binary heap* is a binary tree with the following properties:

- Shape property: It is a complete binary tree.
- Heap property: Each node is greater than or equal to any of its children.

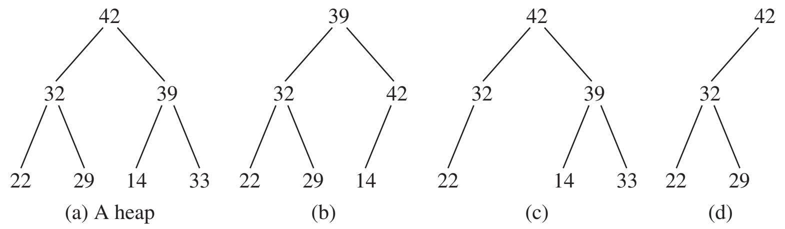


FIGURE 23.9 A binary heap is a special complete binary tree.

complete binary tree

A binary tree is *complete* if each of its levels is full, except that the last level may not be full and all the leaves on the last level are placed leftmost. For example, in Figure 23.9, the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete. Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap because the root (39) is less than its right child (42).

heap

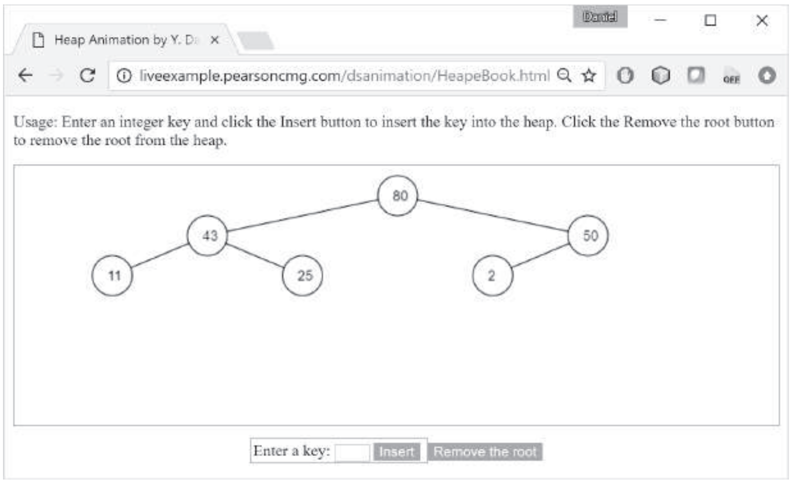
Note

Heap is a term with many meanings in computer science. In this chapter, heap means a binary heap.

Pedagogical Note

A heap can be implemented efficiently for inserting keys and deleting the root. For an interactive demo on how a heap works, go to liveexample.pearsoncmg.com/dsanimation/HeapeBook.html, as shown in Figure 23.10.

heap animation on Companion Website



Usage: Enter an integer key and click the Insert button to insert the key into the heap. Click the Remove the root button to remove the root from the heap.

FIGURE 23.10 The heap animation tool enables you to insert a key and delete the root visually. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

23.6.1 Storing a Heap

A heap can be stored in an `ArrayList` or an array if the heap size is known in advance. The heap in Figure 23.11a can be stored using the array in Figure 23.11b. The root is at position 0, and its two children are at positions 1 and 2. For a node at position i, its left child is at position $2i + 1$, its right child is at position $2i + 2$, and its parent is $(i - 1)/2$. For example, the node for element 39 is at position 4, so its left child (element 14) is at 9 $(2 \times 4 + 1)$, its right child (element 33) is at 10 $(2 \times 4 + 2)$, and its parent (element 42) is at 1 $((4 - 1)/2)$.

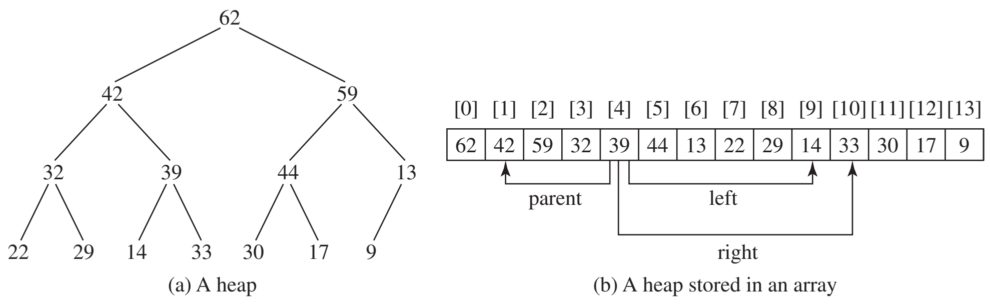


FIGURE 23.11 A binary heap can be implemented using an array.

23.6.2 Adding a New Node

To add a new node to the heap, first add it to the end of the heap then rebuild the tree as follows:

```
Let the last node be the current node;
while (the current node is greater than its parent) {
  Swap the current node with its parent;
  Now the current node is one level up;
}
```

Suppose a heap is initially empty. That heap is shown in Figure 23.12, after adding numbers 3, 5, 1, 19, 11, and 22 in this order.

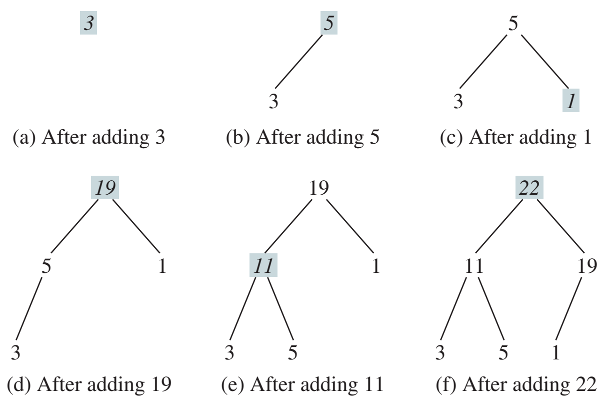


FIGURE 23.12 Elements 3, 5, 1, 19, 11, and 22 are inserted into the heap.

Now consider adding 88 into the heap. Place the new node 88 at the end of the tree, as shown in Figure 23.13a. Swap 88 with 19, as shown in Figure 23.13b. Swap 88 with 22, as shown in Figure 23.13c.

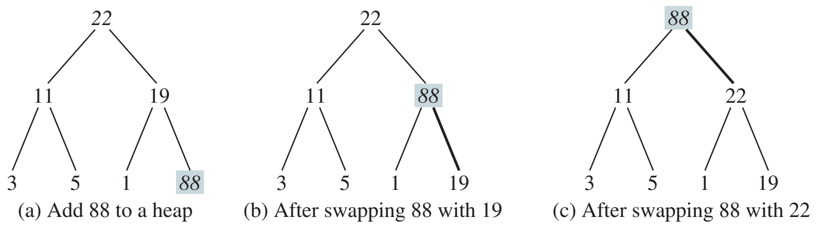


FIGURE 23.13 Rebuild the heap after adding a new node.

23.6.3 Removing the Root

Often you need to remove the maximum element, which is the root in a heap. After the root is removed, the tree must be rebuilt to maintain the heap property. The algorithm for rebuilding the tree can be described as follows:

```
Move the last node to replace the root;
Let the root be the current node;
while (the current node has children and the current node is
       smaller than one of its children) {
  Swap the current node with the larger of its children;
  Now the current node is one level down;
}
```

Figure 23.14 shows the process of rebuilding a heap after the root 62 is removed from Figure 23.11a. Move the last node 9 to the root, as shown in Figure 23.14a. Swap 9 with 59, as shown in Figure 23.14b; swap 9 with 44, as shown in Figure 23.14c; and swap 9 with 30, as shown in Figure 23.14d.

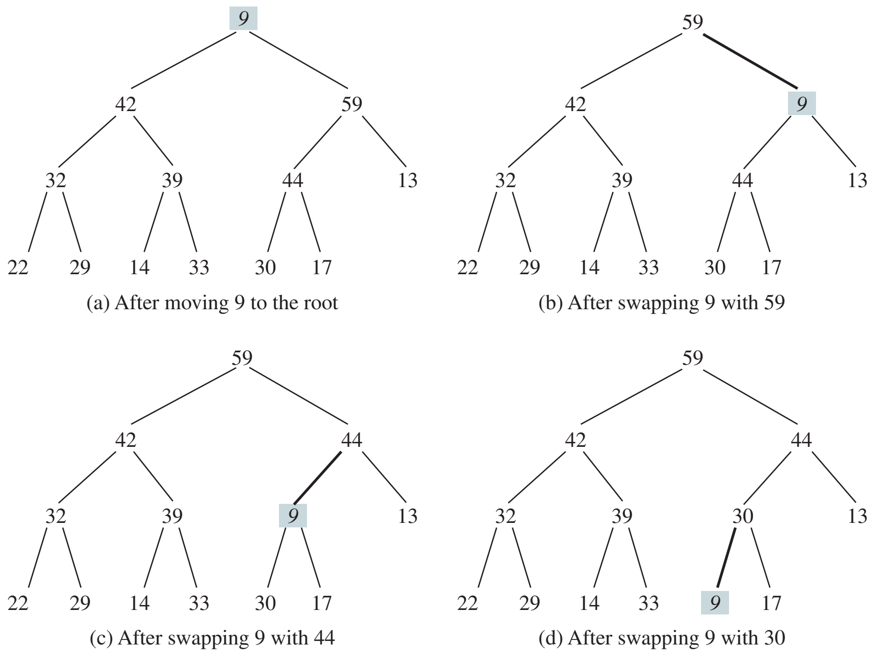

(a) After moving 9 to the root
(b) After swapping 9 with 59
(c) After swapping 9 with 44
(d) After swapping 9 with 30

FIGURE 23.14 Rebuild the heap after the root 62 is removed.

Figure 23.15 shows the process of rebuilding a heap after the root 59 is removed from Figure 23.14d. Move the last node 17 to the root, as shown in Figure 23.15a. Swap 17 with 44, as shown in Figure 23.15b, then swap 17 with 30, as shown in Figure 23.15c.

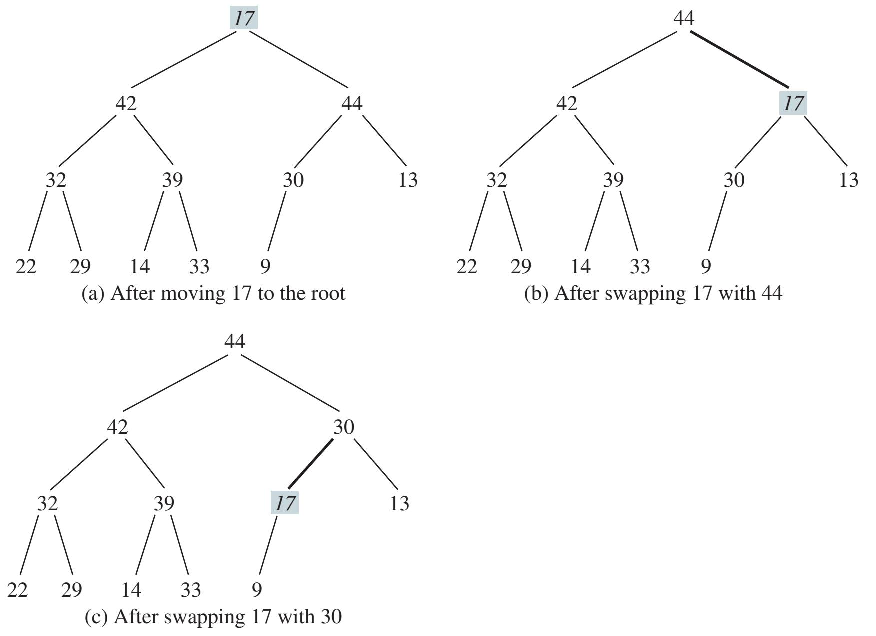
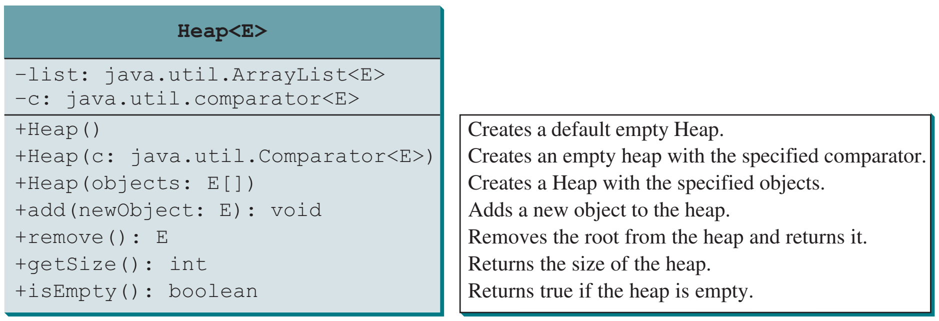

(a) After moving 17 to the root
(b) After swapping 17 with 44
(c) After swapping 17 with 30

FIGURE 23.15 Rebuild the heap after the root 59 is removed.

23.6.4 The **Heap** Class

Now you are ready to design and implement the **Heap** class. The class diagram is shown in Figure 23.16. Its implementation is given in Listing 23.9.



```
Heap<E>
-list: java.util.ArrayList<E>
-c: java.util.comparator<E>
+Heap()                                   Creates a default empty Heap.
+Heap(c: java.util.Comparator<E>)         Creates an empty heap with the specified comparator.
+Heap(objects: E[])                       Creates a Heap with the specified objects.
+add(newObject: E): void                  Adds a new object to the heap.
+remove(): E                              Removes the root from the heap and returns it.
+getSize(): int                           Returns the size of the heap.
+isEmpty(): boolean                       Returns true if the heap is empty.
```

FIGURE 23.16 The **Heap** class provides operations for manipulating a heap.

LISTING 23.9 Heap.java

internal heap representation
comparator

```
public class Heap<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<>();
  private java.util.Comparator<? super E> c;

  /** Create a default heap */
  public Heap() {
    this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
  }

  /** Create a heap with a specified comparator */
  public Heap(java.util.Comparator<E> c) {
    this.c = c;
  }

  /** Create a heap from an array of objects */
  public Heap(E[] objects) {
    this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
    for (int i = 0; i < objects.length; i++)
      add(objects[i]);
  }

  /** Add a new object into the heap */
  public void add(E newObject) {
    list.add(newObject); // Append to the heap
    int currentIndex = list.size() - 1; // The index of the last node

    while (currentIndex > 0) {
      int parentIndex = (currentIndex - 1) / 2;
      // Swap if the current object is greater than its parent
      if (c.compare(list.get(currentIndex),
          list.get(parentIndex)) > 0) {
        E temp = list.get(currentIndex);
        list.set(currentIndex, list.get(parentIndex));
        list.set(parentIndex, temp);
      }
      else
        break; // The tree is a heap now

      currentIndex = parentIndex;
    }
  }

  /** Remove the root from the heap */
  public E remove() {
    if (list.size() == 0) return null;
```

no-arg constructor
create a comparator
internal heap representation
constructor
add a new object
append the object
swap with parent
heap now
remove the root
empty heap

```
46
47      E removedObject = list.get(0);                                        root
48      list.set(0, list.get(list.size() - 1));                               new root
49      list.remove(list.size() -  1);                                        remove the last
50
51      int currentIndex = 0;
52      while (currentIndex < list.size()) {                                  adjust the tree
53        int leftChildIndex = 2 * currentIndex + 1;
54        int rightChildIndex = 2 * currentIndex + 2;
55
56        // Find the maximum between two children
57        if (leftChildIndex >= list.size()) break; // The tree is a heap
58        int maxIndex = leftChildIndex;
59        if (rightChildIndex < list.size()) {
60          if (c.compare(list.get(maxIndex),                                 compare two children
61              list.get(rightChildIndex)) < 0) {
62            maxIndex = rightChildIndex;
63          }
64        }
65
66        // Swap if the current node is less than the maximum
67        if (c.compare(list.get(currentIndex),
68                    list.get(maxIndex)) < 0) {
69          E temp = list.get(maxIndex);                                      swap with the larger child
70          list.set(maxIndex, list.get(currentIndex));
71          list.set(currentIndex, temp);
72          currentIndex = maxIndex;
73        }
74        else
75          break; // The tree is a heap
76      }
77
78      return removedObject;
79    }
80
81    /** Get the number of nodes in the tree */
82    public int getSize() {
83      return list.size();
84    }
85
86    /** Return true if heap is empty */
87    public boolean isEmpty() {
88      return list.size() == 0;
89    }
90  }
```

A heap is represented using an array list internally (line 2). You can change the array list to other data structures, but the **Heap** class contract will remain unchanged.

The elements in the heap can be compared using a comparator or using a natural order if no comparator is specified.

The no-arg constructor (line 6) creates an empty heap using a natural order for the elements as the comparator. The comparator is created using a lambda expression (line 7).

The **add(E newObject)** method (lines 23–41) appends the object to the tree then swaps the object with its parent if the object is greater than its parent. This process continues until the new object becomes the root or is not greater than its parent.

The **remove()** method (lines 44–79) removes and returns the root. To maintain the heap property, the method moves the last object to the root position and swaps it with its larger child if it is less than the larger child. This process continues until the last object becomes a leaf or is not less than its children.

23.6.5 Sorting Using the **Heap** Class

To sort an array using a heap, first create an object using the **Heap** class, add all the elements to the heap using the **add** method, and remove all the elements from the heap using the **remove** method. The elements are removed in descending order. Listing 23.10 gives a program for sorting an array using a heap.

Listing 23.10 HeapSort.java

heapSort method default

heapSort method using comparator

create a Heap

add element

remove element

invoke sort method

```
 1 import java.util.Comparator;
 2
 3 public class HeapSort {
 4   /** Heap sort method */
 5   public static <E> void heapSort(E[] list) {
 6     // Create a Heap of integers
 7     heapSort(list, (e1, e2) -> ((Comparable<E>)e1).compareTo(e2));
 8   }
 9
10   /** Heap sort method */
11   public static <E> void heapSort(E[] list, Comparator<E> c) {
12     // Create a Heap of integers
13     Heap<E> heap = new Heap<>(c);
14
15     // Add elements to the heap
16     for (int i = 0; i < list.length; i++)
17       heap.add(list[i]);
18
19     // Remove elements from the heap
20     for (int i = list.length - 1; i >= 0; i--)
21       list[i] = heap.remove();
22   }
23
24   /** A test method */
25   public static void main(String[] args) {
26     Integer[] list = {-44, -5, -3, 3, 3, 1, -4, 0, 1, 2, 4, 5, 53};
27     heapSort(list);
28     for (int i = 0; i < list.length; i++)
29       System.out.print(list[i] + " ");
30   }
31 }
```

```
-44 -5 -4 -3 0 1 1 2 3 3 4 5 53
```

Two **heapSort** methods are provided here. The **heapSort(E[] list)** method (line 5) sorts a list in a natural order using the **Comparable** interface. The **heapSort(E[] list, Comparator<E> c)** method (line 11) sorts a list using a specified comparator.

23.6.6 Heap Sort Time Complexity

height of a heap

Let us turn our attention to analyzing the time complexity for the heap sort. Let h denote the *height* for a heap of n elements. The height of a nonempty tree is the length of the path from the root node to its furthest leaf. The *height* of a tree that contains a single node is **0**. Conventionally, the height of an empty tree is **-1**. Since a heap is a complete binary tree, the first level has 1 (2^0) node, the second level has 2 (2^1) nodes, the kth level has 2^{k-1} nodes, the h level has 2^{h-1} nodes, and the last $(h + 1)$th level has at least 1 and at most 2^h nodes. Therefore,

$$1 + 2 + \cdots + 2^{h-1} < n \le 1 + 2 + \cdots + 2^{h-1} + 2^h$$

That is,

$$\begin{aligned} 2^h - 1 &< n \le 2^{h+1} - 1 \\ 2^h &< n + 1 \le 2^{h+1} \\ h &< \log(n + 1) \le h + 1 \end{aligned}$$

heap sort vs. merge sort

Thus, $h < \log(n + 1)$ and $h \ge \log(n + 1) - 1$. Therefore, $\log(n + 1) - 1 \le h < \log(n + 1)$. Hence, the height of a heap is $O(\log n)$. More precisely, you can prove that $h = \lfloor \log n \rfloor$ for a non-empty tree.

$O(n \log n)$ worst-case time

Since the `add` method traces a path from a leaf to a root, it takes at most h steps to add a new element to the heap. Thus, the total time for constructing an initial heap is $O(n \log n)$ for an array of n elements. Since the `remove` method traces a path from a root to a leaf, it takes at most h steps to rebuild a heap after removing the root from the heap. Since the `remove` method is invoked n times, the total time for producing a sorted array from a heap is $O(n \log n)$.

Both merge and heap sorts require $O(n \log n)$ time. A merge sort requires a temporary array for merging two subarrays; a heap sort does not need additional array space. Therefore, a heap sort is more space efficient than a merge sort.

Check Point

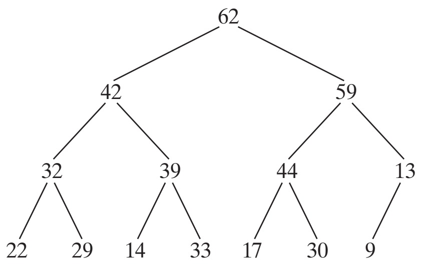
23.6.1 What is a complete binary tree? What is a heap? Describe how to remove the root from a heap and how to add a new object to a heap.

23.6.2 Add the elements **4**, **5**, **1**, **2**, **9**, and **3** into a heap in this order. Draw the diagrams to show the heap after each element is added.

23.6.3 Show the steps of creating a heap using {**45**, **11**, **50**, **59**, **60**, **2**, **4**, **7**, **10**}.

23.6.4 Show the heap after the root in the heap in Figure 23.15c is removed.

23.6.5 Given the following heap, show the steps of removing all nodes from the heap.



23.6.6 Which of the following statements are wrong?

```
Heap<Object> heap1 = new Heap<>();
Heap<Number> heap2 = new Heap<>();
Heap<BigInteger> heap3 = new Heap<>();
Heap<Calendar> heap4 = new Heap<>();
Heap<String> heap5 = new Heap<>();
```

23.6.7 What is the return value from invoking the `remove` method if the heap is empty?

23.6.8 What is the time complexity of inserting a new element into a heap, and what is the time complexity of deleting an element from a heap?

23.6.9 What is the height of a nonempty heap? What is the height of a heap with 16, 17, and 512 elements? If the height of a heap is 5, what is the maximum number of nodes in the heap?

23.7 Bucket and Radix Sorts

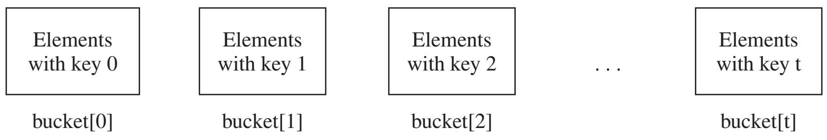
Bucket and radix sorts are efficient for sorting integers.

Key Point

All sort algorithms discussed so far are general sorting algorithms that work for any types of keys (e.g., integers, strings, and any comparable objects). These algorithms sort the elements by comparing their keys. It has been proven that no sorting algorithms based on comparisons can perform better than $O(n \log n)$. However, if the keys are integers, you can use a bucket sort without having to compare the keys.

bucket sort

The *bucket sort* algorithm works as follows. Assume the keys are in the range from `0` to `t`. We need `t + 1` buckets labeled `0`, `1`, . . . , and `t`. If an element's key is `i`, the element is put into the bucket `i`. Each bucket holds the elements with the same key value.



You can use an `ArrayList` to implement a bucket. The bucket-sort algorithm for sorting a list of elements can be described as follows:

```
public static void bucketSort(E[] list) {
  E[] bucket = (E[])new java.util.ArrayList[t+1];

  // Distribute the elements from list to buckets
  for (int i = 0; i < list.length; i++) {
    int key = list[i].getKey(); // Assume element has the getKey() method

    if (bucket[key] == null)
      bucket[key] = new java.util.ArrayList<>();

    bucket[key].add(list[i]);
  }

  // Now move the elements from the buckets back to list
  int k = 0; // k is an index for list
  for (int i = 0; i < bucket.length; i++) {
    if (bucket[i] != null) {
      for (int j = 0; j < bucket[i].size(); j++)
        list[k++] = bucket[i].get(j);
    }
  }
}
```

Clearly, it takes $O(n + t)$ time to sort the list and uses $O(n + t)$ space, where n is the list size.

stable

Note if t is too large, using the bucket sort is not desirable. Instead, you can use a radix sort. The radix sort is based on the bucket sort, but a radix sort uses only 10 buckets.

It is worthwhile to note a bucket sort is *stable*, meaning that if two elements in the original list have the same key value, their order is not changed in the sorted list. That is, if element e_1 and element e_2 have the same key and e_1 precedes e_2 in the original list, e_1 still precedes e_2 in the sorted list.

radix sort

Assume the keys are positive integers. The idea for the *radix sort* is to divide the keys into subgroups based on their radix positions. It applies a bucket sort repeatedly for the key values on radix positions, starting from the least-significant position.

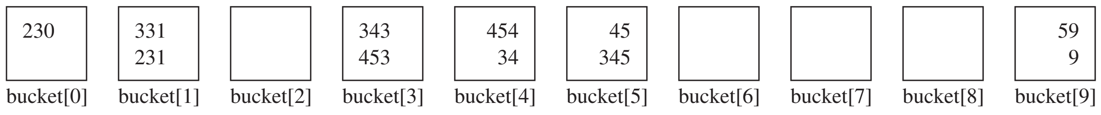
radix sort on Companion Website

Consider sorting the elements with the following keys:

```
331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9
```

queue

Apply the bucket sort on the last radix position, and the elements are put into the buckets as follows:

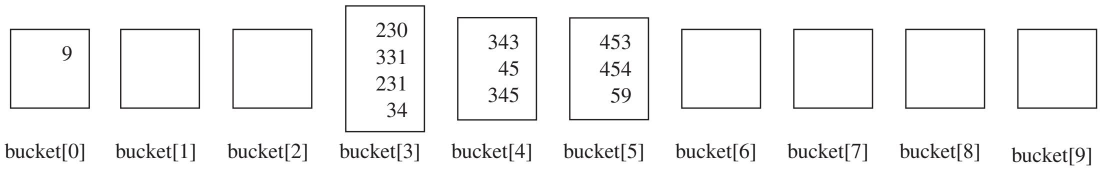


After collecting the elements from the buckets, the elements are in the following order:

```
230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9
```

Apply the bucket sort on the second-to-last radix position, and the elements are put into the buckets as follows:

queue



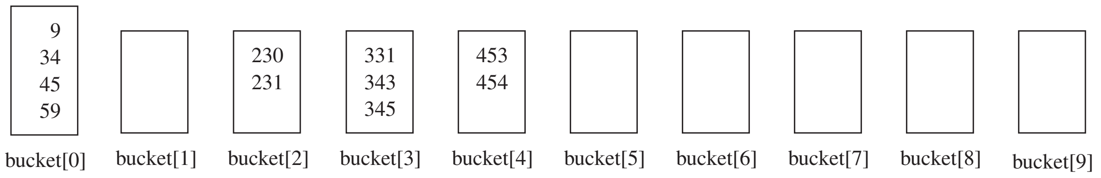
After collecting the elements from the buckets, the elements are in the following order:

```
9, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59
```

(Note **9** is **009**.)

Apply the bucket sort on the third-to-last radix position, and the elements are put into the buckets as follows:

queue



After collecting the elements from the buckets, the elements are in the following order:

```
9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454
```

The elements are now sorted.

Radix sort takes $O(dn)$ time to sort n elements with integer keys, where d is the maximum number of the radix positions among all keys.

23.7.1 Can you sort a list of strings using a bucket sort?

23.7.2 Show how the radix sort works using the numbers **454**, **34**, **23**, **43**, **74**, **86**, and **76**.

Check Point

23.8 External Sort

You can sort a large amount of data using an external sort.

Key Point

All the sort algorithms discussed in the preceding sections assume all the data to be sorted are available at one time in internal memory, such as in an array. To sort data stored in an external file, you must first bring the data to the memory then sort it internally. However, if the file is too large, all the data in the file cannot be brought to memory at one time. This section discusses how to sort data in a large external file. This is called an *external sort*.

external sort

For simplicity, assume two million `int` values are stored in a binary file named **largedata.dat**. This file was created using the program in Listing 23.11.

LISTING 23.11 `CreateLargeFile.java`

```java
import java.io.*;

public class CreateLargeFile {
  public static void main(String[] args)  throws Exception {
```

a binary output stream

output an int value

close output file

read an int value

close input file

```
      DataOutputStream output = new DataOutputStream(
        new BufferedOutputStream(
        new FileOutputStream("largedata.dat")));

      for (int i = 0; i < 2_000_000; i++)
        output.writeInt((int)(Math.random() * 1000000));

      output.close();

      // Display first 100 numbers
      DataInputStream input = new DataInputStream(
        new BufferedInputStream(new FileInputStream("largedata.dat")));
      for (int i = 0; i < 100; i++)
        System.out.print(input.readInt() + " ");

      input.close();
    }
  }
```

```
569193 131317 608695 776266 767910 624915 458599 5010 ... (omitted)
```

A variation of merge sort can be used to sort this file in two phases:

Phase I: Repeatedly bring data from the file to an array, sort the array using an internal sorting algorithm, and output the data from the array to a temporary file. This process is shown in Figure 23.17. Ideally, you want to create a large array, but its maximum size depends on how much memory is allocated to the JVM by the operating system. Assume the maximum array size is 100,000 `int` values. In the temporary file, every 100,000 `int` values are sorted. They are denoted as S_1, S_2, . . . , and S_k, where the last segment, S_k, may contain less than 100,000 values.

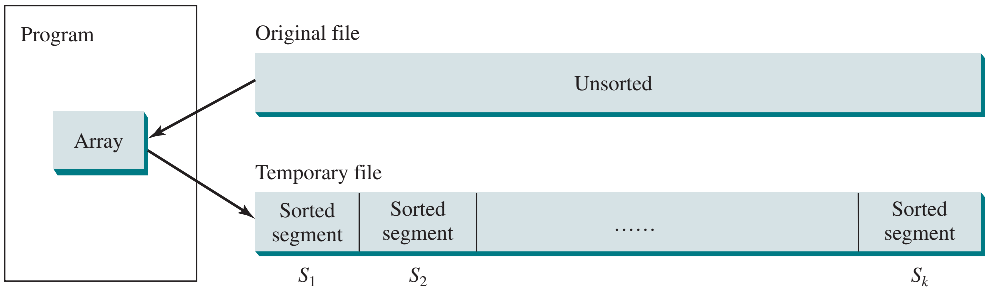


FIGURE 23.17 The original file is sorted in segments.

Phase II: Merge a pair of sorted segments (e.g., S_1 with S_2, S_3 with S_4, . . . , and so on) into a larger sorted segment and save the new segment into a new temporary file. Continue the same process until only one sorted segment results. Figure 23.18 shows how to merge eight segments.

Note

It is not necessary to merge two successive segments. For example, you can merge S_1 with S_5, S_2 with S_6, S_3 with S_7, and S_4 with S_8, in the first merge step. This observation is useful in implementing Phase II efficiently.

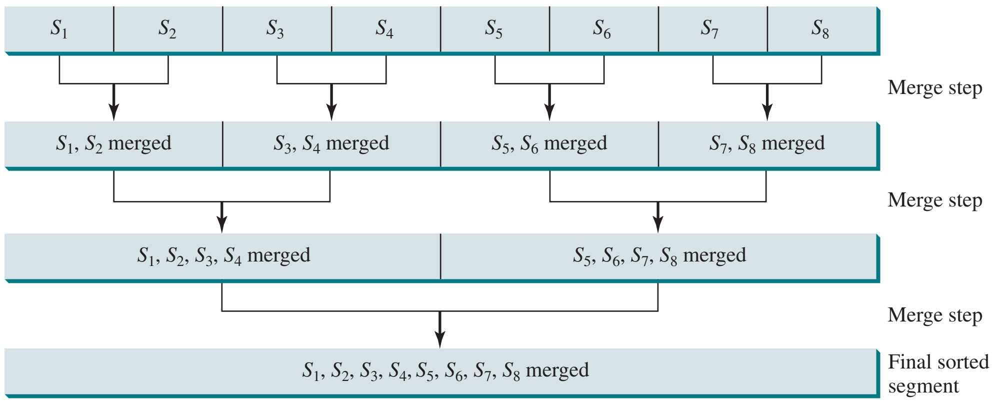


FIGURE 23.18 Sorted segments are merged iteratively.

23.8.1 Implementing Phase I

Listing 23.12 gives the method that reads each segment of data from a file, sorts the segment, and stores the sorted segments into a new file. The method returns the number of segments.

LISTING 23.12 Creating Initial Sorted Segments

```
 1  /** Sort original file into sorted segments */
 2  private static int initializeSegments
 3      (int segmentSize, String originalFile, String f1)
 4      throws Exception {
 5    int[] list = new int[segmentSize];
 6    DataInputStream input = new DataInputStream(
 7      new BufferedInputStream(new FileInputStream(originalFile)));
 8    DataOutputStream output = new DataOutputStream(
 9      new BufferedOutputStream(new FileOutputStream(f1)));
10
11    int numberOfSegments = 0;
12    while (input.available() > 0) {
13      numberOfSegments++;
14      int i = 0;
15      for ( ; input.available() > 0 && i < segmentSize; i++) {
16        list[i] = input.readInt();
17      }
18
19      // Sort an array list[0..i-1]
20      java.util.Arrays.sort(list, 0, i);
21
22      // Write the array to f1.dat
23      for (int j = 0; j < i; j++) {
24        output.writeInt(list[j]);
25      }
26    }
27
28    input.close();
29    output.close();
30
31    return numberOfSegments;
32  }
```

original file

file with sorted segments

sort a segment

output to file

close file

return # of segments

The method creates an array with the maximum size in line 5, a data input stream for the original file in line 6, and a data output stream for a temporary file in line 8. Buffered streams are used to improve performance.

Lines 14–17 read a segment of data from the file into the array. Line 20 sorts the array. Lines 23–25 write the data in the array to the temporary file.

The number of segments is returned in line 31. Note every segment has `MAX_ARRAY_SIZE` number of elements except the last segment, which may have fewer elements.

23.8.2 Implementing Phase II

In each merge step, two sorted segments are merged to form a new segment. The size of the new segment is doubled. The number of segments is reduced by half after each merge step. A segment is too large to be brought to an array in memory. To implement a merge step, copy half the number of segments from the file **f1.dat** to a temporary file **f2.dat**. Then, merge the first remaining segment in **f1.dat** with the first segment in **f2.dat** into a temporary file named **f3.dat**, as shown in Figure 23.19.

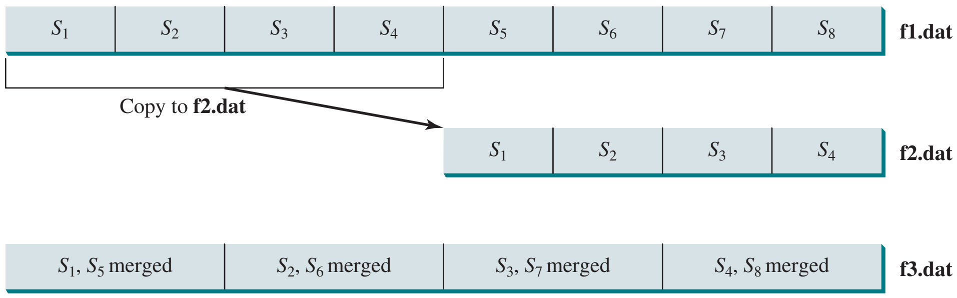


FIGURE 23.19 Sorted segments are merged iteratively.

Note

f1.dat may have one segment more than **f2.dat**. If so, move the last segment into **f3.dat** after the merge.

Listing 23.13 gives a method that copies the first half of the segments in **f1.dat** to **f2.dat**. Listing 23.14 gives a method that merges a pair of segments in **f1.dat** and **f2.dat**. Listing 23.15 gives a method that merges two segments.

LISTING 23.13 Copying First Half Segments

```
private static void copyHalfToF2(int numberOfSegments,
    int segmentSize, DataInputStream f1, DataOutputStream f2)
    throws Exception {
  for (int i = 0; i < (numberOfSegments / 2) * segmentSize; i++) {
    f2.writeInt(f1.readInt());
  }
}
```

input stream f1
output stream f2
segments copied

LISTING 23.14 Merging All Segments

```
private static void mergeSegments(int numberOfSegments,
    int segmentSize, DataInputStream f1, DataInputStream f2,
    DataOutputStream f3)  throws Exception {
  for (int i = 0; i < numberOfSegments; i++) {
    mergeTwoSegments(segmentSize, f1, f2, f3);
  }

  // If f1 has one extra segment, copy it to f3
  while (f1.available() > 0) {
    f3.writeInt(f1.readInt());
  }
}
```

input stream f1 and f2
output stream f3
merge two segments
extra segment in f1?

LISTING 23.15 Merging Two Segments

```
private static void mergeTwoSegments(int segmentSize,
    DataInputStream f1, DataInputStream f2,
    DataOutputStream f3) throws Exception {
  int intFromF1 = f1.readInt();
  int intFromF2 = f2.readInt();
  int f1Count = 1;
  int f2Count = 1;

  while (true) {
    if (intFromF1 < intFromF2) {
      f3.writeInt(intFromF1);
      if (f1.available() == 0 || f1Count++ >= segmentSize) {
        f3.writeInt(intFromF2);
        break;
      }
      else {
        intFromF1 = f1.readInt();
      }
    }
    else {
      f3.writeInt(intFromF2);
      if (f2.available() == 0 || f2Count++ >= segmentSize) {
        f3.writeInt(intFromF1);
        break;
      }
      else {
        intFromF2 = f2.readInt();
      }
    }
  }

  while (f1.available() > 0 && f1Count++ < segmentSize) {
    f3.writeInt(f1.readInt());
  }

  while (f2.available() > 0 && f2Count++ < segmentSize) {
    f3.writeInt(f2.readInt());
  }
}
```

input stream f1 and f2
output stream f3
read from f1
read from f2
write to f3
segment in f1 finished
write to f3
segment in f2 finished
remaining f1 segment
remaining f2 segment

23.8.3 Combining Two Phases

Listing 23.16 gives the complete program for sorting `int` values in **largedata.dat** and storing the sorted data in **sortedfile.dat**.

LISTING 23.16 SortLargeFile.java

```
import java.io.*;

public class SortLargeFile {
  public static final int MAX_ARRAY_SIZE = 100000;
  public static final int BUFFER_SIZE = 100000;

  public static void main(String[] args)  throws Exception {
    // Sort largedata.dat to sortedfile.dat
    sort("largedata.dat", "sortedfile.dat");

    // Display the first 100 numbers in the sorted file
```

max array size
I/O stream buffer size

create initial segments
merge recursively
merge one step
merge recursively
final sorted file
input stream `f1Input`
output stream `f2Output`
copy half segments to `f2`
close `f2Output`
input stream `f2Input`
output stream `f3Output`
merge two segments
close streams

```java
    displayFile("sortedfile.dat");
  }

  /** Sort data in source file and into target file */
  public static void sort(String sourcefile, String targetfile)
      throws Exception {
    // Implement Phase 1: Create initial segments
    int numberOfSegments =
      initializeSegments(MAX_ARRAY_SIZE, sourcefile, "f1.dat");

    // Implement Phase 2: Merge segments recursively
    merge(numberOfSegments, MAX_ARRAY_SIZE,
      "f1.dat", "f2.dat", "f3.dat", targetfile);
  }

  /** Sort original file into sorted segments */
  private static int initializeSegments
      (int segmentSize, String originalFile, String f1)
      throws Exception {
    // Same as Listing 23.12, so omitted
  }

  private static void merge(int numberOfSegments, int segmentSize,
      String f1, String f2, String f3, String targetfile)
      throws Exception {
    if (numberOfSegments > 1) {
      mergeOneStep(numberOfSegments, segmentSize, f1, f2, f3);
      merge((numberOfSegments + 1) / 2, segmentSize * 2,
        f3, f1, f2, targetfile);
    }
    else { // Rename f1 as the final sorted file
      File sortedFile = new File(targetfile);
      if (sortedFile.exists()) sortedFile.delete();
      new File(f1).renameTo(sortedFile);
    }
  }

  private static void mergeOneStep(int numberOfSegments,
      int segmentSize, String f1, String f2, String f3)
      throws Exception {
    DataInputStream f1Input = new DataInputStream(
      new BufferedInputStream(new FileInputStream(f1), BUFFER_SIZE));
    DataOutputStream f2Output = new DataOutputStream(
      new BufferedOutputStream(new FileOutputStream(f2), BUFFER_SIZE));

    // Copy half number of segments from f1.dat to f2.dat
    copyHalfToF2(numberOfSegments, segmentSize, f1Input, f2Output);
    f2Output.close();

    // Merge remaining segments in f1 with segments in f2 into f3
    DataInputStream f2Input = new DataInputStream(
      new BufferedInputStream(new FileInputStream(f2), BUFFER_SIZE));
    DataOutputStream f3Output = new DataOutputStream(
      new BufferedOutputStream(new FileOutputStream(f3), BUFFER_SIZE));

    mergeSegments(numberOfSegments / 2,
      segmentSize, f1Input, f2Input, f3Output);

    f1Input.close();
    f2Input.close();
```

```
 72     f3Output.close();
 73   }
 74
 75   /** Copy first half number of segments from f1.dat to f2.dat */
 76   private static void copyHalfToF2(int numberOfSegments,
 77       int segmentSize, DataInputStream f1, DataOutputStream f2)
 78       throws Exception {
 79     // Same as Listing 23.13, so omitted
 80   }
 81
 82   /** Merge all segments */
 83   private static void mergeSegments(int numberOfSegments,
 84       int segmentSize, DataInputStream f1, DataInputStream f2,
 85       DataOutputStream f3)  throws Exception {
 86     // Same as Listing 23.14, so omitted
 87   }
 88
 89   /** Merges two segments */
 90   private static void mergeTwoSegments(int segmentSize,
 91     DataInputStream f1, DataInputStream f2,
 92     DataOutputStream f3)  throws Exception {
 93     // Same as Listing 23.15, so omitted
 94   }
 95
 96   /** Display the first 100 numbers in the specified file */
 97   public static void displayFile(String filename) {
 98     try {
 99       DataInputStream input =
100         new DataInputStream(new FileInputStream(filename));
101       for (int i = 0; i < 100; i++)
102         System.out.print(input.readInt() + " ");
103       input.close();
104     }
105     catch (IOException ex) {
106       ex.printStackTrace();
107     }
108   }
109 }
```

display file

```
0 1 1 1 2 2 2 3 3 4 5 6 8 8 9 9 9 10 10 11...(omitted)
```

Before you run this program, first run Listing 23.11, CreateLargeFile.java, to create the file **largedata.dat**. Invoking `sort("largedata.dat", "sortedfile.dat")` (line 9) reads data from `largedata.dat` and writes sorted data to `sortedfile.dat`. Invoking `display-File("sortedfile.dat")` (line 12) displays the first `100` numbers in the specified file. Note the files are created using binary I/O. You cannot view them using a text editor such as Notepad.

The `sort` method first creates initial segments from the original array and stores the sorted segments in a new file, **f1.dat** (lines 19–20), then produces a sorted file in `targetfile` (lines 23–24).

The `merge` method

```
merge(int numberOfSegments, int segmentSize,
  String f1, String f2, String f3, String targetfile)
```

merges the segments in `f1` into `f3` using `f2` to assist the merge. The `merge` method is invoked recursively with many merge steps. Each merge step reduces the `numberOfSegments` by half and doubles the sorted segment size. After the completion of one merge step, the next merge step merges the new segments in f3 to f2 using f1 to assist the merge. The statement to invoke the new merge method is

```
merge((numberOfSegments + 1) / 2, segmentSize * 2,
  f3, f1, f2, targetfile);
```

The numberOfSegments for the next merge step is (numberOfSegments + 1) / 2. For example, if numberOfSegments is 5, numberOfSegments is 3 for the next merge step because every two segments are merged but one is left unmerged.

The recursive merge method ends when numberOfSegments is 1. In this case, f1 contains sorted data. File f1 is renamed to targetfile (line 45).

23.8.4 External Sort Complexity

In the external sort, the dominating cost is that of I/O. Assume n is the number of elements to be sorted in the file. In Phase I, n number of elements are read from the original file and output to a temporary file. Therefore, the I/O for Phase I is $O(n)$.

In Phase II, before the first merge step, the number of sorted segments is $\frac{n}{c}$, where c is MAX_ARRAY_SIZE. Each merge step reduces the number of segments by half. Thus, after the first merge step, the number of segments is $\frac{n}{2c}$. After the second merge step, the number of segments is $\frac{n}{2^2c}$, and after the third merge step the number of segments is $\frac{n}{2^3c}$. After $\log\left(\frac{n}{c}\right)$ merge steps, the number of segments is reduced to 1. Therefore, the total number of merge steps is $\log\left(\frac{n}{c}\right)$.

In each merge step, half the number of segments are read from file f1 then written into a temporary file f2. The remaining segments in f1 are merged with the segments in f2. The number of I/Os in each merge step is $O(n)$. Since the total number of merge steps is $\log\left(\frac{n}{c}\right)$, the total number of I/Os is

$$O(n) \times \log\left(\frac{n}{c}\right) = O(n \log n)$$

Therefore, the complexity of the external sort is $O(n \log n)$.

Check Point

23.8.1 Describe how external sort works. What is the complexity of the external sort algorithm?

23.8.2 Ten numbers {2, 3, 4, 0, 5, 6, 7, 9, 8, 1} are stored in the external file **largedata.dat**. Trace the SortLargeFile program by hand with MAX_ARRAY_SIZE 2.

Key Terms

bubble sort	890	heap sort	900
bucket sort	907	height of a heap	906
complete binary tree	900	merge sort	892
external sort	909	quick sort	896
heap	900	radix sort	908

Chapter Summary

1. The worst-case complexity for a *selection sort*, *insertion sort*, *bubble sort*, and *quick sort* is $O(n^2)$.
2. The average- and worst-case complexity for a *merge sort* is $O(n \log n)$. The average time for a quick sort is also $O(n \log n)$.
3. *Heaps* are a useful data structure for designing efficient algorithms such as sorting. You learned how to define and implement a heap class, and how to insert and delete elements to/from a heap.
4. The time complexity for a *heap sort* is $O(n \log n)$.
5. *Bucket* and *radix sorts* are specialized sorting algorithms for integer keys. These algorithms sort keys using buckets rather than by comparing keys. They are more efficient than general sorting algorithms.
6. A variation of the merge sort—called an *external sort*—can be applied to sort large amounts of data from external files.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

Sections 23.3–23.5

23.1 (*Generic bubble sort*) Write the following two generic methods using bubble sort. The first method sorts the elements using the `Comparable` interface, and the second uses the `Comparator` interface.

```
public static <E extends Comparable<E>>
  void bubbleSort(E[] list)
public static <E> void bubbleSort(E[] list,
  Comparator<? super E> comparator)
```

23.2 (*Generic merge sort*) Write the following two generic methods using merge sort. The first method sorts the elements using the `Comparable` interface and the second uses the `Comparator` interface.

```
public static <E extends Comparable<E>>
  void mergeSort(E[] list)
public static <E> void mergeSort(E[] list,
  Comparator<? super E> comparator)
```

23.3 (*Generic quick sort*) Write the following two generic methods using quick sort. The first method sorts the elements using the `Comparable` interface, and the second uses the `Comparator` interface.

```
public static <E extends Comparable<E>>
  void quickSort(E[] list)
public static <E> void quickSort(E[] list,
  Comparator<? super E> comparator)
```

23.4 (*Improve quick sort*) The quick-sort algorithm presented in the book selects the first element in the list as the pivot. Revise it by selecting the median among the first, middle, and the last elements in the list.

***23.5** (*Modify merge sort*) Rewrite the `mergeSort` method to recursively sort the first half of the array and the second half of the array without creating new temporary arrays, then merge the two into a temporary array and copy its contents to the original array, as shown in Figure 23.6b.

23.6 (*Check order*) Write the following overloaded methods that check whether an array is ordered in ascending order or descending order. By default, the method checks ascending order. To check descending order, pass `false` to the ascending argument in the method.

```
public static boolean ordered(int[] list)
public static boolean ordered(int[] list, boolean ascending)
public static boolean ordered(double[] list)
public static boolean ordered
  (double[] list, boolean ascending)
public static <E extends Comparable<E>>
  boolean ordered(E[] list)
public static <E extends Comparable<E>> boolean ordered
  (E[] list, boolean ascending)
public static <E> boolean ordered(E[] list,
  Comparator<? super E> comparator)
public static <E> boolean ordered(E[] list,
  Comparator<? super E> comparator, boolean ascending)
```

Section 23.6

max-heap
min-heap

23.7 (*Min-heap*) The heap presented in the text is also known as a *max-heap*, in which each node is greater than or equal to any of its children. A *min-heap* is a heap in which each node is less than or equal to any of its children. Min-heaps are often used to implement priority queues. Revise the `Heap` class in Listing 23.9 to implement a min-heap.

23.8 (*Generic insertion sort*) Write the following two generic methods using insertion sort. The first method sorts the elements using the `Comparable` interface, and the second uses the `Comparator` interface.

```
public static <E extends Comparable<E>>
  void insertionSort(E[] list)
public static <E> void insertionSort(E[] list,
  Comparator<? super E> comparator)
```

***23.9** (*Generic heap sort*) Write the following two generic methods using heap sort. The first method sorts the elements using the `Comparable` interface, and the second uses the `Comparator` interface. (*Hint*: Use the `Heap` class in Programming Exercise 23.5.)

```
public static <E extends Comparable<E>>
  void heapSort(E[] list)
public static <E> void heapSort(E[] list,
  Comparator<? super E> comparator)
```

****23.10** (*Heap visualization*) Write a program that displays a heap graphically, as shown in Figure 23.10. The program lets you insert and delete an element from the heap.

23.11 (*Heap `clone` and `equals`*) Implement the `clone` and `equals` method in the `Heap` class.
Use the template at `liveexample.pearsoncmg.com/test/Exercise23_11.txt` for your code.

Section 23.7

***23.12** (*Radix sort*) Write a program that randomly generates 1,000,000 integers and sorts them using radix sort.

***23.13** (*Execution time for sorting*) Write a program that obtains the execution time of selection sort, bubble sort, merge sort, quick sort, heap sort, and radix sort for input size 50,000, 100,000, 150,000, 200,000, 250,000, and 300,000. Your program should create data randomly and print a table like this:

Array size	Selection Sort	Bubble Sort	Merge Sort	Quick Sort	Heap Sort	Radix Sort
50,000						
100,000						
150,000						
200,000						
250,000						
300,000						

(*Hint*: You can use the following code template to obtain the execution time.)

```
long startTime = System.nanoTime();
perform the task;
long endTime = System.nanoTime();
long executionTime = endTime - startTime;
```

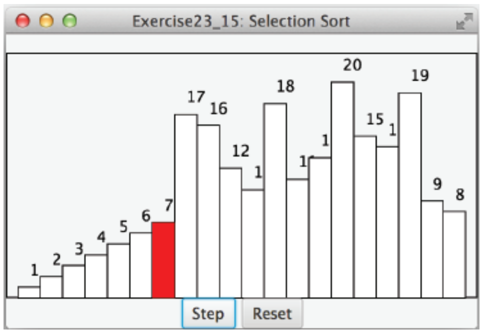
Section 23.8

***23.14** (*Execution time for external sorting*) Write a program that obtains the execution time of external sorts for integers of size 5,000,000, 10,000,000, 15,000,000, 20,000,000, 25,000,000, and 30,000,000. Your program should print a table like this:

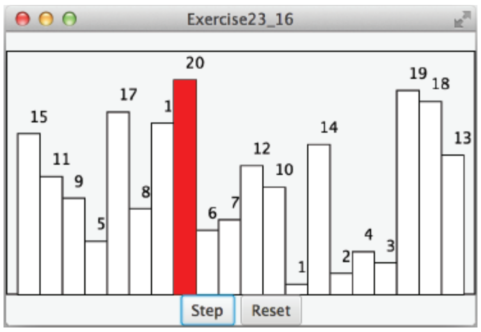
File size	5,000,000	10,000,000	15,000,000	20,000,000	25,000,000	30,000,000
Time						

Comprehensive

***23.15** (*Selection-sort animation*) Write a program that animates the selection-sort algorithm. Create an array that consists of 20 distinct numbers from 1 to 20 in a random order. The array elements are displayed in a histogram, as shown in Figure 23.20a. Clicking the *Step* button causes the program to perform an iteration of the outer loop in the algorithm and repaints the histogram for the new array. Color the last bar in the sorted subarray. When the algorithm is finished, display a message to inform the user. Clicking the *Reset* button creates a new random array for a new start. (You can easily modify the program to animate the insertion algorithm.)



(a)



(b)

FIGURE 23.20 (a) The program animates selection sort. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) The program animates bubble sort.

***23.16** (*Bubble-sort animation*) Write a program that animates the bubble-sort algorithm. Create an array that consists of 20 distinct numbers from 1 to 20 in a random order. The array elements are displayed in a histogram, as shown in Figure 23.20b. Clicking the *Step* button causes the program to perform one comparison in the algorithm and repaints the histogram for the new array. Color the bar that represents the number being considered in the swap. When the algorithm is finished, display a message to inform the user. Clicking the *Reset* button creates a new random array for a new start.

***23.17** (*Radix-sort animation*) Write a program that animates the radix-sort algorithm. Create an array that consists of 20 random numbers from 0 to 1,000. The array elements are displayed, as shown in Figure 23.21. Clicking the *Step* button causes the program to place a number in a bucket. The number that has just been placed is displayed in red. Once all the numbers are placed in the buckets, clicking the *Step* button collects all the numbers from the buckets and moves them back to the array. When the algorithm is finished, clicking the *Step* button displays a message to inform the user. Clicking the *Reset* button creates a new random array for a new start.

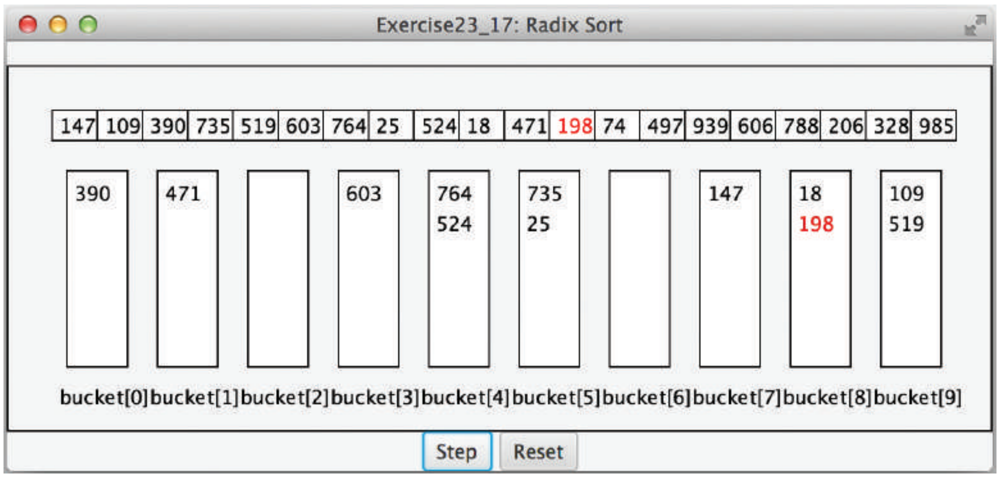
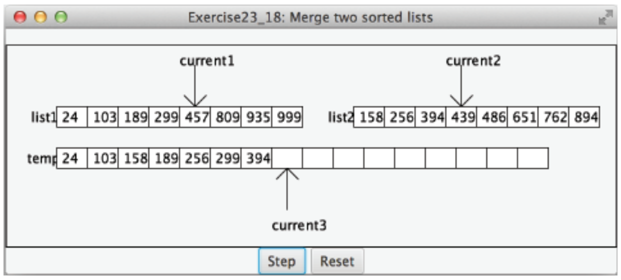
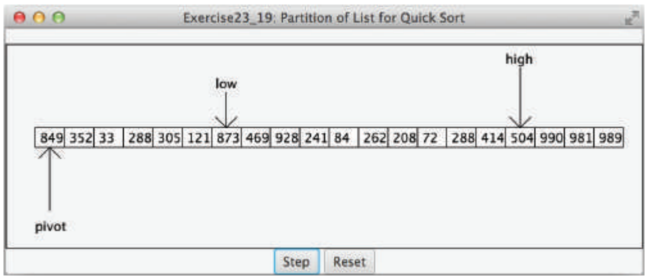

Exercise23_17: Radix Sort
147 109 390 735 519 603 764 25 524 18 471 198 74 497 939 606 788 206 328 985
390
471
603
764
524
735
25
147
18
198
109
519
bucket[0] bucket[1] bucket[2] bucket[3] bucket[4] bucket[5] bucket[6] bucket[7] bucket[8] bucket[9]
Step
Reset

FIGURE 23.21 The program animates radix sort. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***23.18** (*Merge animation*) Write a program that animates the merge of two sorted lists. Create two arrays, `list1` and `list2`, each of which consists of 8 random numbers from 1 to 999. The array elements are displayed, as shown in Figure 23.22a. Clicking the *Step* button causes the program to move an element from `list1` or `list2` to `temp`. Clicking the *Reset* button creates two new random arrays for a new start. When the algorithm is finished, clicking the *Step* button displays a message to inform the user.


Exercise23_18: Merge two sorted lists
current1
current2
list1 24 103 189 299 457 809 935 999
list2 158 256 394 439 486 651 762 894
temp 24 103 158 189 256 299 394
current3
Step
Reset

(a)



(b)

FIGURE 23.22 The program animates a merge of two sorted lists. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) The program animates a partition for quick sort.

***23.19** (*Quick-sort partition animation*) Write a program that animates the partition for a quick sort. The program creates a list that consists of 20 random numbers from 1 to 999. The list is displayed, as shown in Figure 23.22b. Clicking the *Step* button causes the program to move `low` to the right or `high` to the left, or swap the elements at `low` and `high`. Clicking the *Reset* button creates a new list of random numbers for a new start. When the algorithm is finished, clicking the *Step* button displays a message to inform the user.