

CHAPTER 19

Sorting

Objectives

- To study and analyze time complexity of various sorting algorithms (§§19.2–19.7).
- To design, implement, and analyze insertion sort (§19.2).
- To design, implement, and analyze bubble sort (§19.3).
- To design, implement, and analyze merge sort (§19.4).
- To design, implement, and analyze quick sort (§19.5).
- To design and implement a binary heap (§19.6).
- To design, implement, and analyze heap sort (§19.6).
- To design, implement, and analyze bucket sort and radix sort (§19.7).
- To design, implement, and analyze external sort for files that have a large amount of data (§19.8).

19.1 Introduction

Key Point: Sorting algorithms are good examples to study for understanding algorithm design and analysis.

When presidential candidate Barack Obama visited Google in 2007, 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.

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, functions, and arrays.
- Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

The data to be sorted might be integers, doubles, strings, or other items. Data may be sorted in increasing order or decreasing order. 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.

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

19.2 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.

Figure 19.1 shows how to sort the list { 2, 9, 5, 4, 8, 1, 6 } using insertion sort.

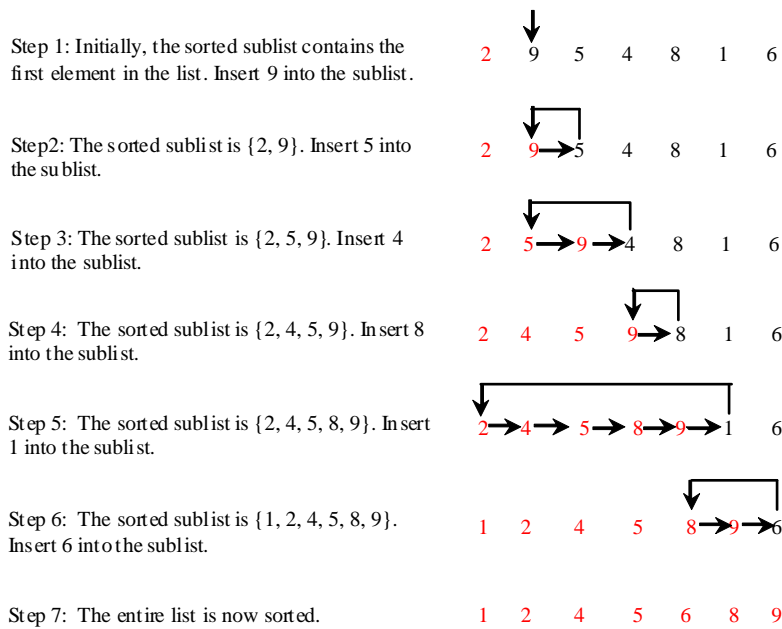


Figure 19.1

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

The algorithm can be described as follows:

```
for (int i = 1; i < listSize; 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 19.2](#), 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]`.

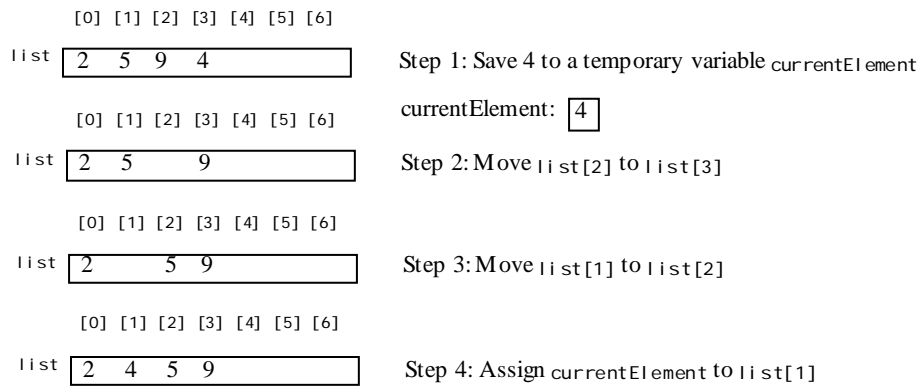


Figure 19.2

A new element is inserted into a sorted sublist.

The algorithm can be expanded and implemented as in [Listing 19.1](#).

Listing 19.1 InsertionSort.cpp

```

1  #include <iostream>
2  using namespace std;
3
4  void insertionSort(int list[], int listSize)
5  {
6      for (int i = 1; i < listSize; i++)
7      {
8          // Insert list[i] into a sorted sublist list[0..i-1] so that
9          // list[0..i] is sorted.
10         int currentElement = list[i];
11         int k;
12         for (k = i - 1; k >= 0 && list[k] > currentElement; k--)
13         {
14             list[k + 1] = list[k];
15         }
16
17         // Insert the current element into list[k+1]
18         list[k + 1] = currentElement;
19     }
20 }
21
22 int main()
23 {
24     const int SIZE = 9;
25     int list[] = {1, 7, 3, 4, 9, 3, 3, 1, 2};
26     insertionSort(list, SIZE);
27     for (int i = 0; i < SIZE; i++)
28         cout << list[i] << " ";
29
30     return 0;
31 }
```

Sample output

1 1 2 3 3 3 4 7 9

The `insertionSort` function sorts any array of `int` elements. The function is implemented with a nested `for` loop. The outer loop (with the loop control variable `i`) (line 6) 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 function, trace it with the following statements:

```
int list[] = {1, 9, 4.5, 6.6, 5.7, -4.5};
insertionSort(list, 6);
```

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,

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

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

Check point

- 19.1 Describe how an insertion sort works. What is the time complexity for an insertion sort?
- 19.2 Use Figure 19.1 as an example to show how to apply a bubble sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.
- 19.3 If a list is already sorted, how many comparisons will the `insertionSort` function perform?

19.3 Bubble Sort

Key Point: A bubble sort sorts the array in multiple phases. Each pass successively swaps the neighboring elements if the elements are not in order.

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.

Figure 19.3a 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 19.3.

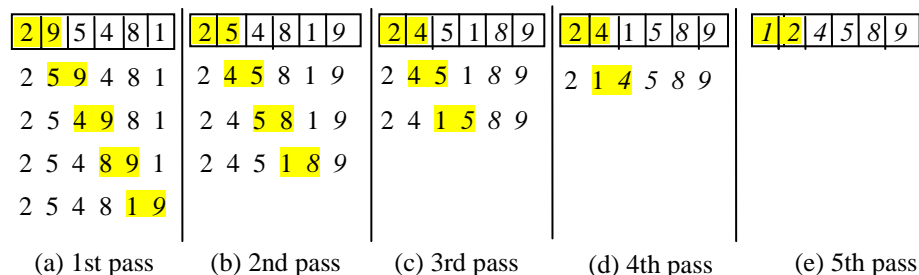


Figure 19.3

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 19.3b, 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 19.3c, you compare and order pairs of elements sequentially except the last two elements, because they are already in order. So 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 19.1](#).

Listing 19.2 Bubble Sort Algorithm

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

Note that 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 19.2 as in [Listing 19.3](#).

Listing 19.3 Improved Bubble Sort Algorithm

```
1  bool needNextPass = true;
2  for (int k = 1; k < arraySize && needNextPass; k++)
3  {
4      // Array may be sorted and next pass not needed
5      needNextPass = false;
6      // Perform the kth pass
7      for (int i = 0; i < arraySize - k; i++)
8      {
9          if (list[i] > list[i + 1])
10             {
11                 swap list[i] with list[i + 1];
12                 needNextPass = true; // Next pass still needed
13             }
14     }
15 }
```

The algorithm can be implemented in [Listing 19.4](#).

Listing 19.4 BubbleSort.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  // The function for sorting the numbers
5  void bubbleSort(int list[], int arraySize)
6  {
7      bool needNextPass = true;
8
9      for (int k = 1; k < arraySize && needNextPass; k++)
10     {
11         // Array may be sorted and next pass not needed
12         needNextPass = false;
13         for (int i = 0; i < arraySize - k; i++)
14         {
15             if (list[i] > list[i + 1])
```

```

16     {
17         // Swap list[i] with list[i + 1]
18         int temp = list[i];
19         list[i] = list[i + 1];
20         list[i + 1] = temp;
21
22         needNextPass = true; // Next pass still needed
23     }
24 }
25 }
26 }
27
28 int main()
29 {
30     const int SIZE = 9;
31     int list[] = {1, 7, 3, 4, 9, 3, 3, 1, 2};
32     bubbleSort(list, SIZE);
33     for (int i = 0; i < SIZE; i++)
34         cout << list[i] << " ";
35
36     return 0;
37 }

```

Sample output

```
1 1 2 3 3 3 4 7 9
```

In the best-case analysis, 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 analysis, 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:

$$\begin{aligned}
 & (n - 1) + (n - 2) + \dots + 2 + 1 \\
 &= \frac{(n - 1)n}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)
 \end{aligned}$$

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

Check point

- 19.4 Describe how a bubble sort works. What is the time complexity for a bubble sort?
- 19.5 Use Figure 19.3 as an example to show how to apply a bubble sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.
- 19.6 If a list is already sorted, how many comparisons will the `bubbleSort` function perform?

19.4 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, merge them.

The algorithm for a merge sort is given in **Listing 19.5**.

Listing 19.5 Merge Sort Algorithm

```
void mergeSort(int list[], int arraySize)
{
    if (arraySize > 1)
    {
        mergeSort on list[0 ... arraySize / 2];
        mergeSort on list[arraySize / 2 + 1 ... arraySize];
        merge list[0 ... arraySize / 2] with
            list[arraySize / 2 + 1 ... arraySize];
    }
}
```

Figure 19.4 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); merge (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).

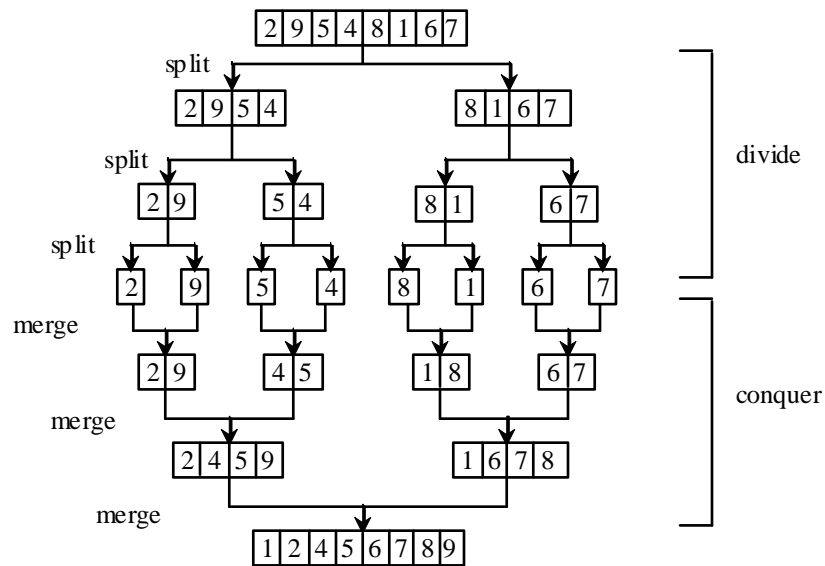


Figure 19.4

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 19.6](#).

Listing 19.6 MergeSort.cpp

```

1  #include <iostream>
2  using namespace std;
3
4  // Function prototype
5  void arraycopy(int source[], int sourceStartIndex,
6               int target[], int targetStartIndex, int length);
7
8  void merge(int list1[], int list1Size,
9            int list2[], int list2Size, int temp[]);
10
11 // The function for sorting the numbers
12 void mergeSort(int list[], int arraySize)
13 {
14     if (arraySize > 1)
15     {
16         // Merge sort the first half
17         int* firstHalf = new int[arraySize / 2];
18         arraycopy(list, 0, firstHalf, 0, arraySize / 2);

```

```

19     mergeSort(firstHalf, arraySize / 2);
20
21     // Merge sort the second half
22     int secondHalfLength = arraySize - arraySize / 2;
23     int* secondHalf = new int[secondHalfLength];
24     arraycopy(list, arraySize / 2, secondHalf, 0, secondHalfLength);
25     mergeSort(secondHalf, secondHalfLength);
26
27     // Merge firstHalf with secondHalf
28     merge(firstHalf, arraySize / 2, secondHalf, secondHalfLength,
29           list);
30
31     delete [] firstHalf;
32     delete [] secondHalf;
33 }
34 }
35
36 void merge(int list1[], int list1Size,
37           int list2[], int list2Size, int temp[])
38 {
39     int current1 = 0; // Current index in list1
40     int current2 = 0; // Current index in list2
41     int current3 = 0; // Current index in temp
42
43     while (current1 < list1Size && current2 < list2Size)
44     {
45         if (list1[current1] < list2[current2])
46             temp[current3++] = list1[current1++];
47         else
48             temp[current3++] = list2[current2++];
49     }
50
51     while (current1 < list1Size)
52         temp[current3++] = list1[current1++];
53
54     while (current2 < list2Size)
55         temp[current3++] = list2[current2++];
56 }
57
58 void arraycopy(int source[], int sourceStartIndex,
59               int target[], int targetStartIndex, int length)
60 {
61     for (int i = 0; i < length; i++)
62     {
63         target[i + targetStartIndex] = source[i + sourceStartIndex];
64     }
65 }
66
67 int main()
68 {
69     const int SIZE = 9;
70     int list[] = {1, 7, 3, 4, 9, 3, 3, 1, 2};
71     mergeSort(list, SIZE);
72     for (int i = 0; i < SIZE; i++)
73         cout << list[i] << " ";
74
75     return 0;
76 }

```

The `mergeSort` function (lines 12-34) creates a new array `firstHalf`, which is a copy of the first half of `list` (line 18). The algorithm invokes `mergeSort` recursively on `firstHalf` (line 19). The length of the `firstHalf` is `arraySize / 2` and that of the `secondHalf` is `arraySize - arraySize / 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 25). After `firstHalf` and `secondHalf` are sorted, they are merged to `list` (lines 28-29). So, array `list` is now sorted.

The `merge` function (lines 36-56) merges two sorted arrays. This function merges arrays `list1` and `list2` into a new array `temp`. So, the size of `temp` should be `list1Size + list2Size`. `current1` and `current2` point to the current element to be considered in `list1` and `list2` (lines 39-41). The function repeatedly compares the current elements from `list1` and `list2` and moves the smaller one to `temp`. If the smaller one is in `list1`, `current1` is increased by 1 (line 46). If the smaller one is in `list2`, `current2` is increased by 1 (line 48). 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 51-52). If there are still unmoved elements in `list2`, copy them to `temp` (lines 54-55).

Figure 19.5 illustrates how to merge 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 them and move the smaller element 1 to `temp`, as shown in Figure 19.5a. `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 19.5b, 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 19.5c.

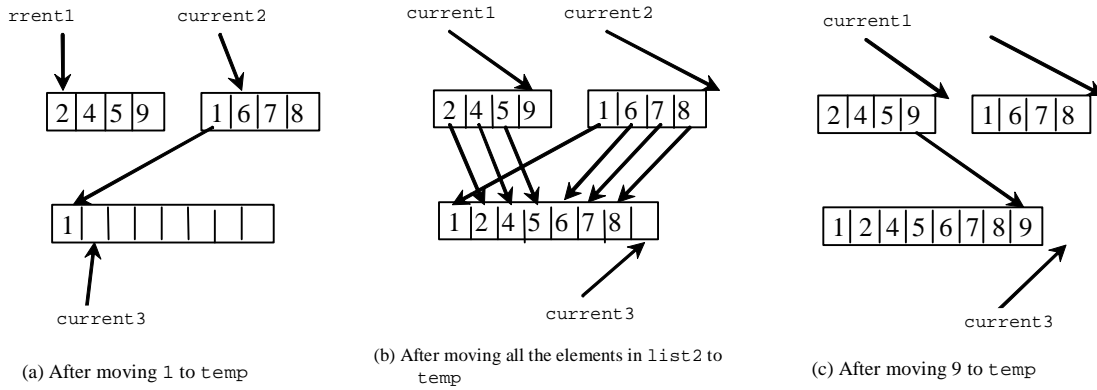


Figure 19.5

Two sorted arrays are merged into one sorted array.

Let $T(n)$ denote the time required for sorting an array of n elements using merge sort. Without loss of generality, assume n is a power of 2. The merge sort algorithm copies the array into two subarrays, sorts them using the same algorithm recursively, and then merges the subarrays. The time to copy `list` to `firstHalf` and `secondHalf` is n and the time to merge two sorted subarrays is $n - 1$. So,

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

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. It can be shown that

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

The complexity of merge sort is $O(n \log n)$. This algorithm is better than selection sort, insertion sort, and bubble sort.

Check point

- 19.7 Describe how a merge sort works.
- 19.8 Use Figure 19.4 as an example to show how to apply a merge sort on $\{45, 11, 50, 59, 60, 2, 4, 7, 10\}$.
- 19.9 What is the time complexity for a merge sort?

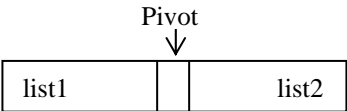
19.5 Quick Sort

Key Point: 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 that 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 19.7**.

Listing 19.7 Quick Sort Algorithm

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



The diagram illustrates the partitioning step of the quick sort algorithm. It shows a horizontal array divided into three sections. The left section is labeled 'list1', the middle section is labeled 'Pivot' with a downward arrow pointing to it, and the right section is labeled 'list2'. This represents the state where elements less than or equal to the pivot are in list1, the pivot is in its final position, and elements greater than the pivot are in list2.

Each **partition** places the pivot in the right place. 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 the first element in the array is chosen as the pivot. (Programming Exercise 19.4 proposes an alternative strategy for selecting the pivot.)

Figure 19.6 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 19.6b. The highlighted pivot is placed in the right place in the array. Apply quick sort on two partial arrays (4 2 1 3 0) and then (8 9 6 7). The pivot 4 partitions (4 2 1 3 0) into just one partial array (0 2 1 3), as shown in Figure 19.6c. Apply quick sort on (0 2 1 3). The pivot 0 partitions it into just one partial array (2 1 3), as shown in Figure 19.6d. Apply quick sort on (2 1 3). The pivot 2 partitions it into (1) and (3), as shown in Figure 19.6e. Apply quick sort on (1). Since the array contains just one element, no further partition is needed.

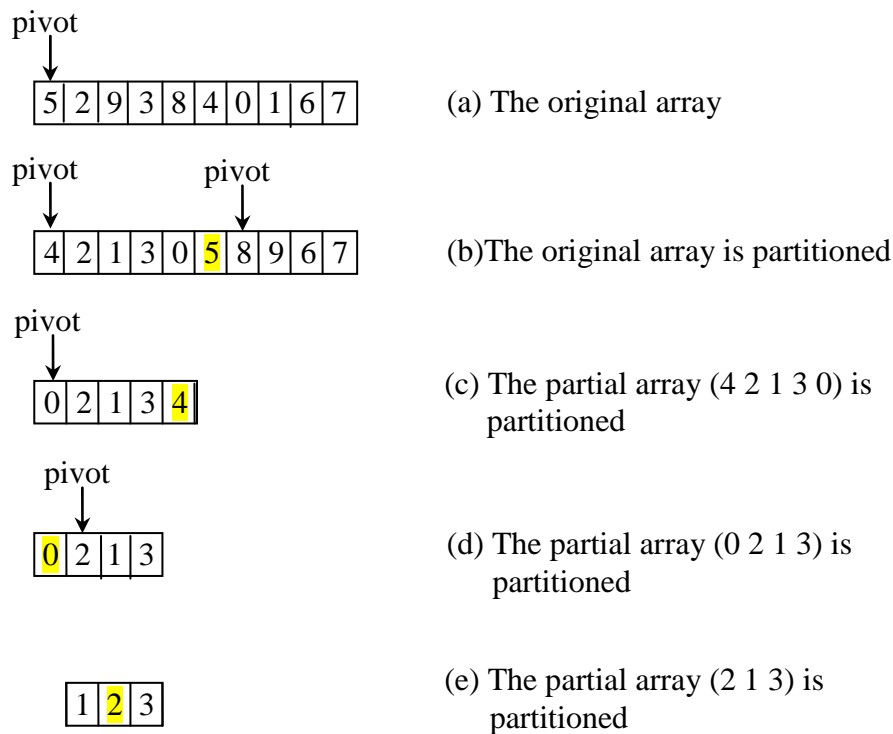


Figure 19.6

The quick sort algorithm is recursively applied to partial arrays.

The quick sort algorithm is implemented in [Listing 19.8](#). There are two overloaded `quickSort` functions in the class. The first function (line 2) is used to sort an array. The second is a helper function (line 6) that sorts a partial array with a specified range.

Listing 19.8 QuickSort.cpp

```

1  #include <iostream>
2  using namespace std;
3
4  // Function prototypes
5  void quickSort(int list[], int arraySize);
6  void quickSort(int list[], int first, int last);
7  int partition(int list[], int first, int last);
8
9  void quickSort(int list[], int arraySize)
10 {
11     quickSort(list, 0, arraySize - 1);

```

```

12 }
13
14 void quickSort(int list[], int first, int last)
15 {
16     if (last > first)
17     {
18         int pivotIndex = partition(list, first, last);
19         quickSort(list, first, pivotIndex - 1);
20         quickSort(list, pivotIndex + 1, last);
21     }
22 }
23
24 // Partition the array list[first..last]
25 int partition(int list[], int first, int last)
26 {
27     int pivot = list[first]; // Choose the first element as the pivot
28     int low = first + 1; // Index for forward search
29     int high = last; // Index for backward search
30
31     while (high > low)
32     {
33         // Search forward from left
34         while (low <= high && list[low] <= pivot)
35             low++;
36
37         // Search backward from right
38         while (low <= high && list[high] > pivot)
39             high--;
40
41         // Swap two elements in the list
42         if (high > low)
43         {
44             int temp = list[high];
45             list[high] = list[low];
46             list[low] = temp;
47         }
48     }
49
50     while (high > first && list[high] >= pivot)
51         high--;
52
53     // Swap pivot with list[high]
54     if (pivot > list[high])
55     {
56         list[first] = list[high];
57         list[high] = pivot;
58         return high;
59     }
60     else
61     {
62         return first;
63     }
64 }
65
66 int main()
67 {
68     const int SIZE = 9;
69     int list[] = {1, 7, 3, 4, 9, 3, 3, 1, 2};
70     quickSort(list, SIZE);
71     for (int i = 0; i < SIZE; i++)

```



```

72     cout << list[i] << " ";
73
74     return 0;
75 }

```

The `partition` function (lines 25-64) partitions array `list[first..last]` using the pivot. The first element in the partial array is chosen as the pivot (line 27). Initially, `low` points to the second element in the partial array (line 28), and `high` points to the last element in the partial array (line 29).

The function searches for the first element from left forward in the array that is greater than the pivot (lines 34-35), then search for the first element from right backward in the array that is less than or equal to the pivot (lines 38-39). Swap these two elements (line 42-47). Repeat the same search and swap operations until all the elements are searched in the while loop (lines 31-48).

The function returns the new index for the pivot that divides the partial array into two parts if the pivot has been moved (line 58). Otherwise, return the original index for the pivot (line 62).

Figure 19.7 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 19.7a.

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 19.7b. Swap 9 with 1, as shown in Figure 19.7c. Continue the search, and move `low` to point to element 8 and `high` to point to element 0, as shown in Figure 19.7d. Swap element 8 with 0, as shown in Figure 19.7e. Continue to move `low` until it passes `high`, as shown in Figure 19.7f. Now all the elements are examined. Swap the pivot with element 4 at index `high`. The final partition is shown in Figure 19.7g. The index of the pivot is returned when the function is finished.

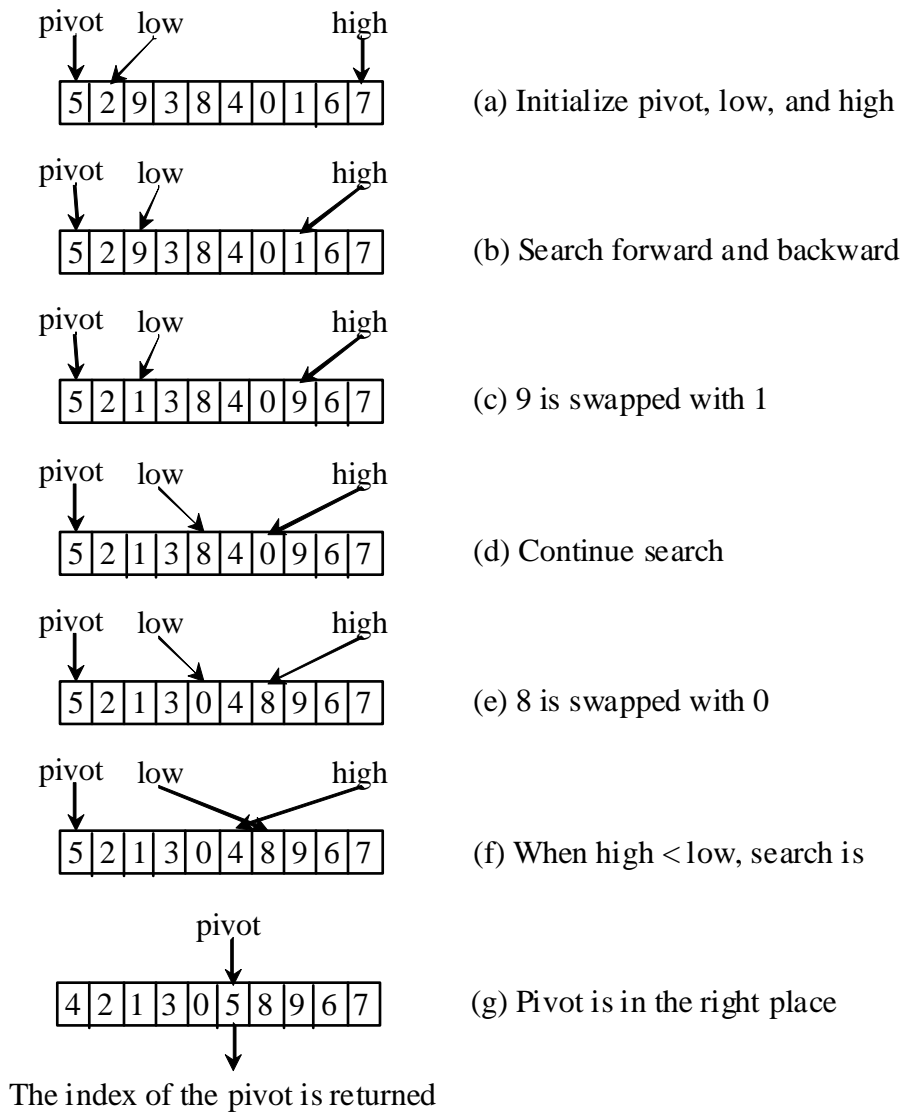


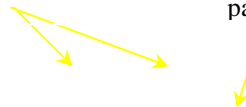
Figure 19.7

The **partition** function 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 analysis. Thus, the time required for partition is $O(n)$.

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 $(n-1) + (n-2) + \dots + 2 + 1 = O(n^2)$ 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 time


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

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

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 sort and quick sort** 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.

Check point

- 19.10 Describe how quick sort works. What is the time complexity for a quick sort?
- 19.11 Why is quick sort more space efficient than merge sort?
- 19.12 Use Figure 19.6 as an example to show how to apply a quick sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

19.6 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 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 *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.

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 19.8**, 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).

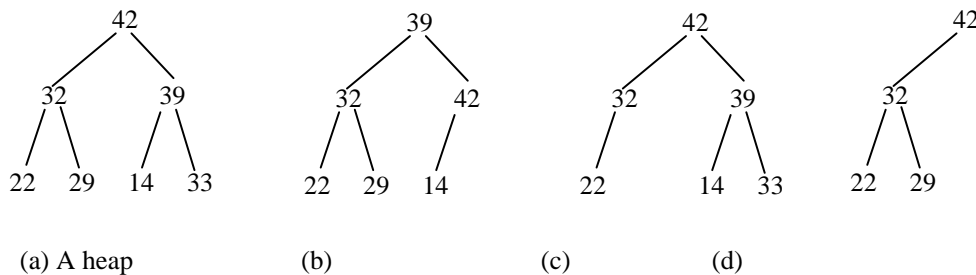


Figure 19.8

A binary heap is a special complete binary tree.

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 for deleting the root. For an interactive demo on how a heap works, go to www.cs.armstrong.edu/liang/animation/HeapAnimation.html, as shown in

Figure 19.9.

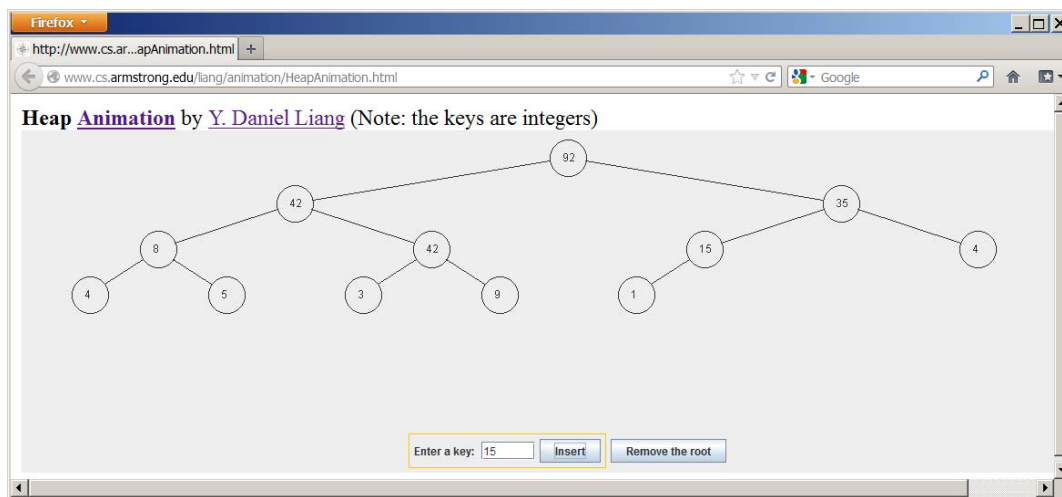


Figure 19.9

The heap animation tool enables you to insert a key and delete the root visually.

19.6.1 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 19.10a** can be stored using the array in Figure 19.10b. 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 ($(4 - 1) / 2$).

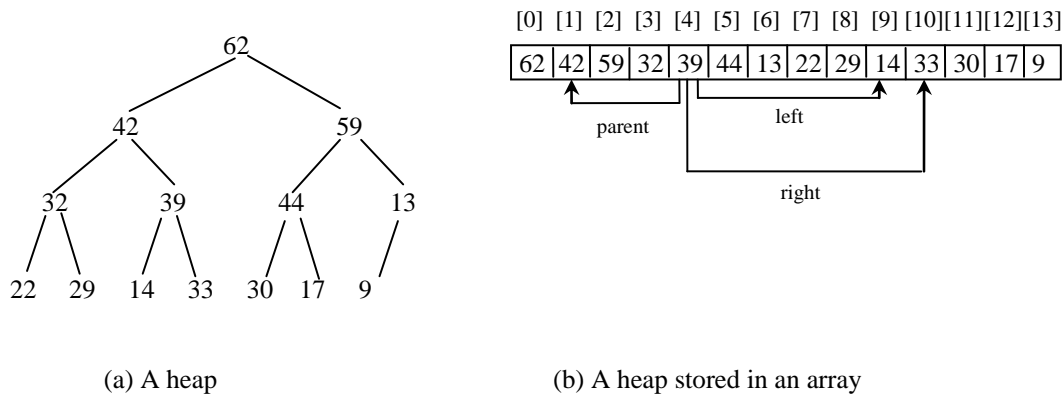


Figure 19.10

A binary heap can be implemented using an array.

19.6.2 Adding a New Node

To add a new node to the heap, first add it to the end of the heap and 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 19.11**, after adding numbers 3, 5, 1, 19, 11, and 22 in this order.

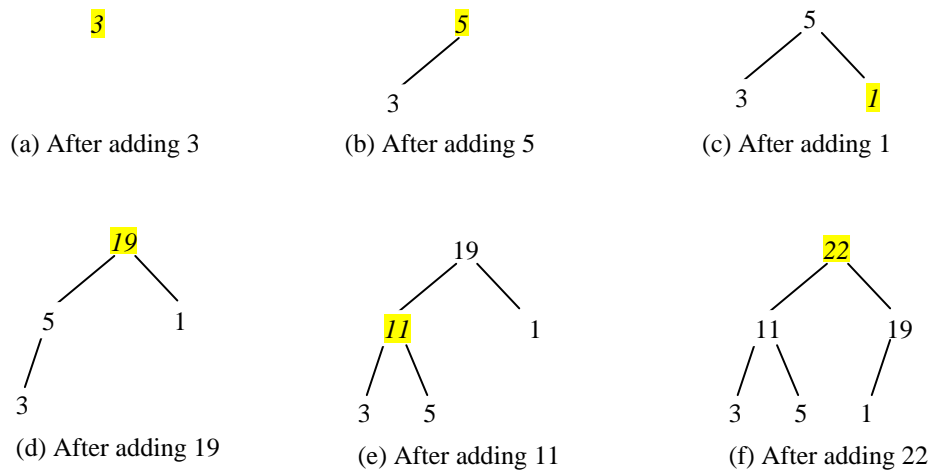


Figure 19.11

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 19.12a**.

Swap 88 with 19, as shown in Figure 19.19b. Swap 88 with 22, as shown in Figure 19.12c.

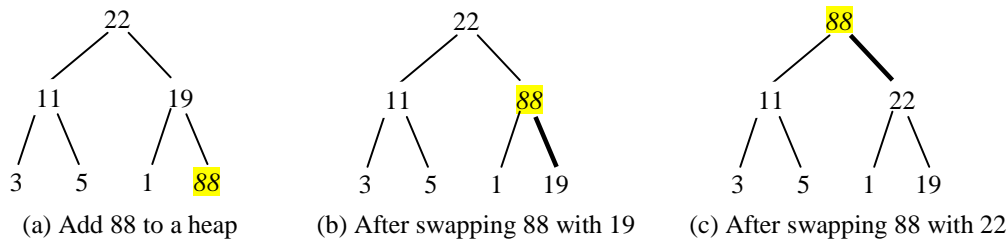


Figure 19.12

Rebuild the heap after adding a new node.

19.6.3 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 19.13 shows the process of rebuilding a heap after the root 62 is removed from Figure 19.10a. Move the last node, 9, to the root, as shown in Figure 19.13a. Swap 9 with 59, as shown in Figure 19.13b; swap 9 with 44, as shown in Figure 19.13c; and swap 9 with 30, as shown in Figure 19.13d.

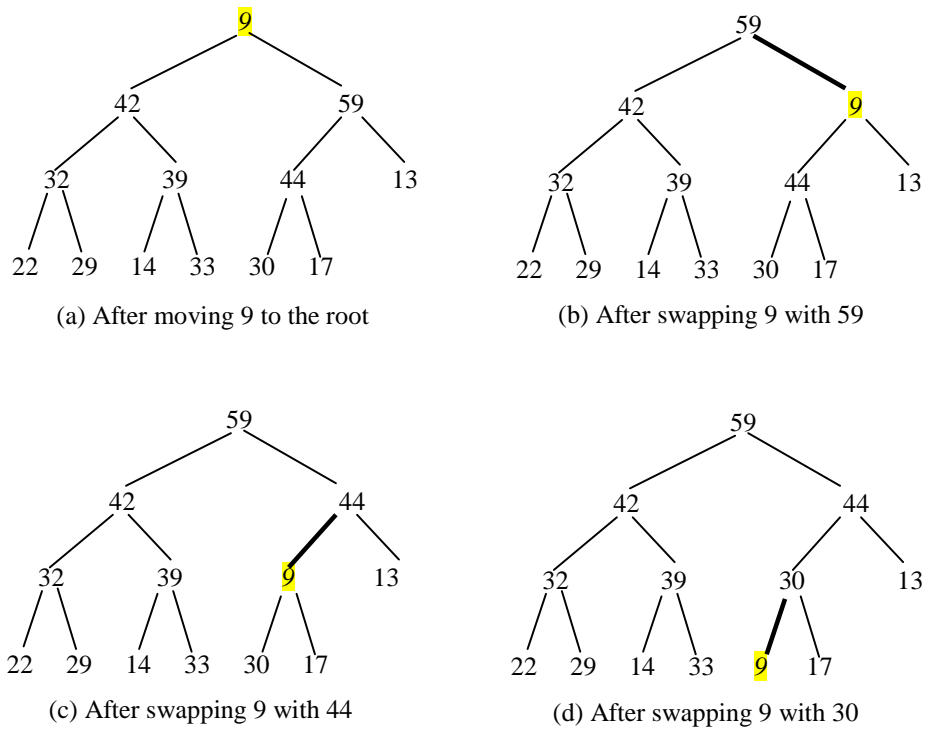
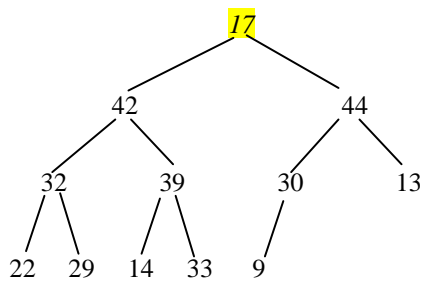


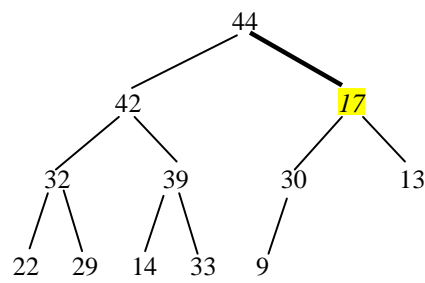
Figure 19.13

Rebuild the heap after the root 62 is removed.

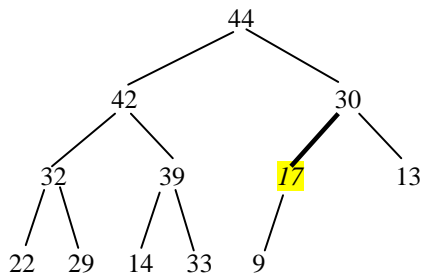
Figure 19.14 shows the process of rebuilding a heap after the root, 59, is removed from Figure 19.13d. Move the last node, 17, to the root, as shown in Figure 19.14a. Swap 17 with 44, as shown in Figure 19.14b, and then swap 17 with 30, as shown in Figure 19.14c.



(a) After moving 17 to the root



(b) After swapping 17 with 44



(c) After swapping 17 with 30

Figure 19.14

Rebuild the heap after the root, 59, is removed.

19.6.4 The **Heap** Class

Now you are ready to design and implement the **Heap** class. The class diagram is shown in **Figure 19.15**. Its implementation is given in **Listing 19.9**.

Heap<T>	
-vector: vector<T>	Stores elements in the vector.
+Heap()	Creates a default heap.
+Heap(elements[: T, arraySize: int)	Creates a heap with the specified objects.
+remove(): T	Removes the root from the heap and returns it.
+add(element: T): void	Adds a new object to the heap.
+getSize(): int const	Returns the size of the heap.

Figure 19.15

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

Listing 19.9 Heap.h

```

1  #ifndef HEAP_H
2  #define HEAP_H
3  #include <vector>
4  #include <stdexcept>
5  using namespace std;
6
7  template<typename T>
8  class Heap
9  {
10 public:
11     Heap();
12     Heap(T elements[], int arraySize);
13     T remove() throw (runtime_error);
14     void add(T element);
15     int getSize() const;
16
17 private:
18     vector<T> v;
19 };
20
21 template<typename T>
22 Heap<T>::Heap()
23 {
24 }
25
26 template<typename T>
27 Heap<T>::Heap(T elements[], int arraySize)
28 {
29     for (int i = 0; i < arraySize; i++)
30     {
31         add(elements[i]);
32     }
33 }
34
35 // Remove the root from the heap
36 template<typename T>
37 T Heap<T>::remove() throw (runtime_error)
38 {
39     if (v.size() == 0)
40         throw runtime_error("Heap is empty");
41
42     T removedElement = v[0];
43     v[0] = v[v.size() - 1]; // Copy the last to root
44     v.pop_back(); // Remove the last element
45
46     // Maintain the heap property
47     int currentIndex = 0;
48     while (currentIndex < v.size())
49     {
50         int leftChildIndex = 2 * currentIndex + 1;
51         int rightChildIndex = 2 * currentIndex + 2;
52
53         // Find the maximum between two children
54         if (leftChildIndex >= v.size()) break; // The tree is a heap
55         int maxIndex = leftChildIndex;
56         if (rightChildIndex < v.size())
57         {
58             if (v[maxIndex] < v[rightChildIndex])
59                 maxIndex = rightChildIndex;
60         }
61         if (maxIndex != currentIndex)
62         {
63             swap(v[currentIndex], v[maxIndex]);
64             currentIndex = maxIndex;
65         }
66     }
67 }

```

```

60         maxIndex = rightChildIndex;
61     }
62 }
63
64 // Swap if the current node is less than the maximum
65 if (v[currentIndex] < v[maxIndex])
66 {
67     T temp = v[maxIndex];
68     v[maxIndex] = v[currentIndex];
69     v[currentIndex] = temp;
70     currentIndex = maxIndex;
71 }
72 else
73     break; // The tree is a heap
74 }
75
76 return removedElement;
77 }
78
79 // Insert element into the heap and maintain the heap property
80 template<typename T>
81 void Heap<T>::add(T element)
82 {
83     v.push_back(element); // Append element to the heap
84     int currentIndex = v.size() - 1; // The index of the last node
85
86     // Maintain the heap property
87     while (currentIndex > 0)
88     {
89         int parentIndex = (currentIndex - 1) / 2;
90         // Swap if the current element is greater than its parent
91         if (v[currentIndex] > v[parentIndex])
92         {
93             T temp = v[currentIndex];
94             v[currentIndex] = v[parentIndex];
95             v[parentIndex] = temp;
96         }
97         else
98             break; // the tree is a heap now
99
100         currentIndex = parentIndex;
101     }
102 }
103
104 // Get the number of element in the heap
105 template<typename T>
106 int Heap<T>::getSize() const
107 {
108     return v.size();
109 }
110
111 #endif

```

A heap is represented using a vector internally (line 18). You may change it to other data structures, but the [Heap](#) class contract will remain unchanged

The `remove()` function (lines 36–77) removes and returns the root. To maintain the heap property, the function moves the last element 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 element becomes a leaf or is not less than its children.

The `add(T element)` function (lines 80–102) appends the element to the tree and then swaps it with its parent if it is greater than its parent. This process continues until the new element becomes the root or is not greater than its parent.

19.6.5 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` function, and remove all the elements from the heap using the `remove` function. The elements are removed in descending order. Listing 19.10 gives a program for sorting an array using a heap.

Listing 19.10 `HeapSort.cpp`

```
1  #include <iostream>
2  #include "Heap.h"
3  using namespace std;
4
5  template<typename T>
6  void heapSort(T list[], int arraySize)
7  {
8      Heap<T> heap;
9
10     for (int i = 0; i < arraySize; i++)
11         heap.add(list[i]);
12
13     for (int i = 0; i < arraySize; i++)
14         list[arraySize - i - 1] = heap.remove();
15 }
16
17 int main()
18 {
19     const int SIZE = 9;
20     int list[] = {1, 7, 3, 4, 9, 3, 3, 1, 2};
21     heapSort(list, SIZE);
22     for (int i = 0; i < SIZE; i++)
23         cout << list[i] << " ";
24
25     return 0;
26 }
```

Note that the largest element in the heap is removed first. So, the removed elements from the heap are placed in the array in the reversed order (lines 13-14).

19.6.6 Heap Sort Time Complexity

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 heap** is the number of nodes in the longest path from the root to a leaf node. Since a heap is a complete binary tree, the first level has 1 node, the second level has 2 nodes, the k th level has 2^{k-1} nodes, the $(h-1)$ level has 2^{h-2} nodes, and the h th level has at least 1 and at most 2^{h-1} nodes. Therefore,

$$1 + 2 + \dots + 2^{h-2} < n \leq 1 + 2 + \dots + 2^{h-2} + 2^{h-1}$$

That is,

$$2^{h-1} - 1 < n \leq 2^h - 1$$

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

$$h - 1 < \log(n + 1) \leq h$$

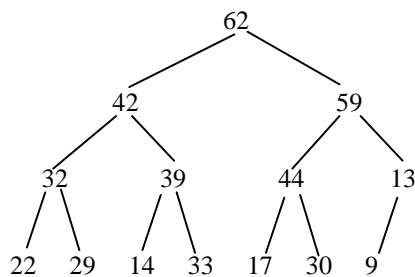
Thus, $h < \log(n + 1) + 1$ and $\log(n + 1) \leq h$. Therefore, $\log(n + 1) \leq h < \log(n + 1) + 1$. Hence, the height of the heap is $O(\log n)$.

Since the **add** function 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** function 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** function is invoked n times, the total time for producing a sorted array from a heap is $O(n \log n)$.

Both **merge sorts 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

- 19.13 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.
- 19.14 What is the return value from invoking the `remove` function if the heap is empty?
- 19.15 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.
- 19.16 Show the heap after the root in the heap in Figure 19.14c is removed.
- 19.17 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?
- 19.18 Show the steps of creating a heap using {45, 11, 50, 59, 60, 2, 4, 7, 10}.
- 19.19 Given the following heap, show the steps of removing all nodes from the heap.



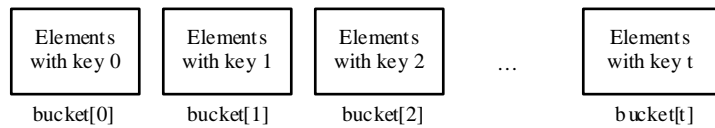
- 19.20 What is the time complexity of a heap sort?

19.7 Bucket Sort and Radix Sort

Key Point: *Bucket sorts and radix sorts are efficient for sorting integers.*

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. The lower bound for general sorting algorithms is $O(n \log n)$, so no sorting algorithms based on comparisons can perform better than $O(n \log n)$. However, if the keys are small integers, you can use a bucket sort without having to compare the keys.

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

Note that 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 ten buckets.

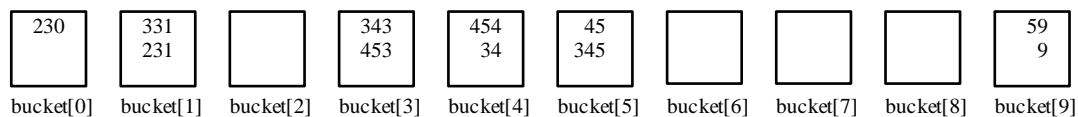
It is worthwhile to note that a bucket sort is **stable**, meaning that if two elements in the original list have the same key value, their order is not changed in the sorted list. That is, if element e_1 and element e_2 have the same key and e_1 precedes e_2 in the original list, e_1 still precedes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

Assume that 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.

Consider sorting the elements with the following keys:

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

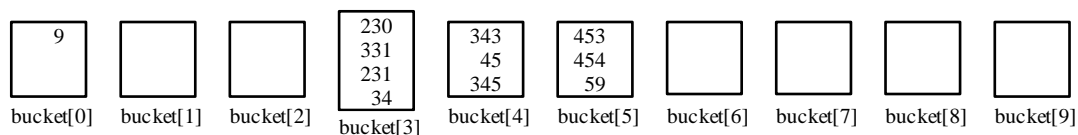
Apply the bucket sort on the last radix position, and the **elements are put** into the buckets as follows:



After being removed 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:

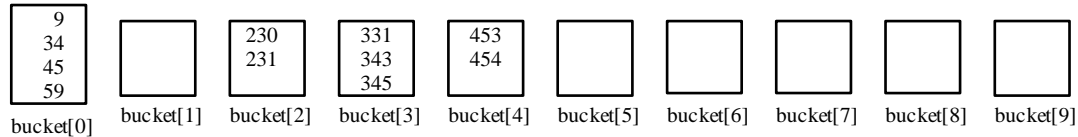


After being removed from the buckets, the elements are in the following order:

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

(Note that 9 is 009.)

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



After being removed 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.

In general, 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.

Check point

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

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

19.8 External Sort

Key Point: *You can sort a large amount data using an external sort.*

All the sort algorithms discussed in the preceding sections assume that 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 and 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*.

For simplicity, assume that two million `int` values are stored in a binary file named `largedata.dat`. This file was created using the program in Listing 19.11.

Listing 19.11 CreateLargeFile.cpp

```
1 #include <iostream>
2 #include <fstream>
3 #include <cstdlib>
4 using namespace std;
5
6 int main()
7 {
```



```

8     fstream output;
9     output.open("largedata.dat", ios::out | ios::binary);
10
11     for (int i = 0; i < 2000000; i++)
12     {
13         int value = rand();
14         output.write(reinterpret_cast<char*>(&value), sizeof(value));
15     }
16
17     output.close();
18     cout << "File created" << endl;
19
20     fstream input;
21     input.open("largedata.dat", ios::in | ios::binary);
22     int value;
23
24     cout << "The first 10 numbers in the file are " << endl;
25     for (int i = 0; i < 10; i++)
26     {
27         input.read(reinterpret_cast<char*>(&value), sizeof(value));
28         cout << value << " ";
29     }
30
31     input.close();
32
33     return 0;
34 }
35

```

Sample output

```

File created
The first 10 numbers in the file are
130 10982 1090 11656 7117 17595 6415 22948 31126 9004

```

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 19.16](#). 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 that 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, \dots , and S_k , where the last segment, S_k , may contain less than 100,000 values.

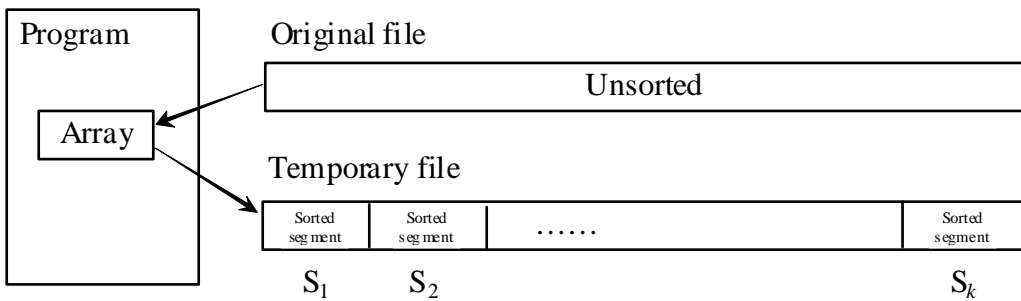


Figure 19.16

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 19.17** shows how to merge eight segments.

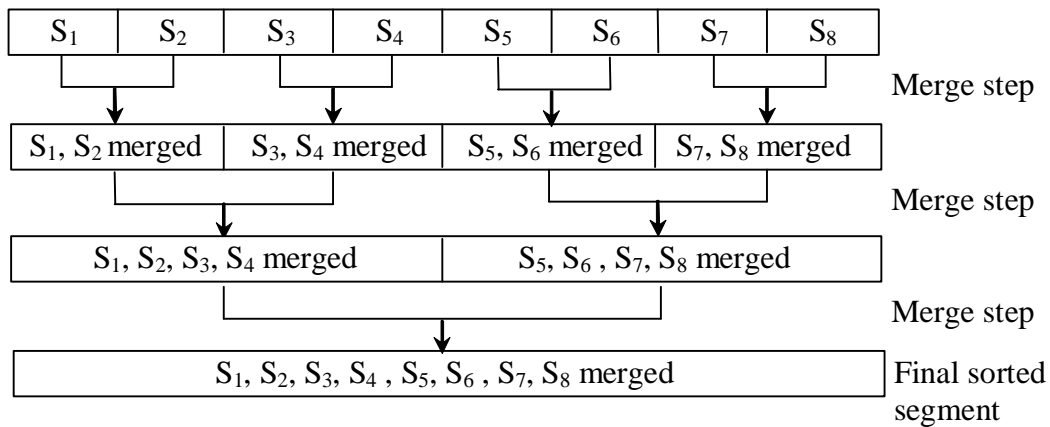


Figure 19.17

Sorted segments are merged iteratively.

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.

19.8.1 Implementing Phase I

Listing 19.12 gives the function that reads each segment of data from a file, sorts the segment, and stores the sorted segments into a new file. The function returns the number of segments.

Listing 19.12 Creating Initial Sorted Segments

```
1  // Sort original file into sorted segments
2  int initializeSegments(int segmentSize, string sourceFile, string fl)
3  {
4      int* list = new int[segmentSize];
5
6      fstream input;
7      input.open(sourceFile.c_str(), ios::in | ios::binary);
8      fstream output;
9      output.open(fl.c_str(), ios::out | ios::binary);
10
11     int numberOfSegments = 0;
12     while (!input.eof())
13     {
14         int i = 0;
15         for ( ; !input.eof() && i < segmentSize; i++)
16         {
17             input.read(reinterpret_cast<char*>
18                 (&list[i]), sizeof(list[i]));
19         }
20
21         if (input.eof()) i--;
22         if (i <= 0)
23             break;
24         else
25             numberOfSegments++;
26
27         // Sort an array list[0..i-1]
28         quickSort(list, i);
29
30         // Write the array to fl.dat
31         for (int j = 0; j < i; j++)
32         {
33             output.write(reinterpret_cast<char*>
34                 (&list[j]), sizeof(list[j]));
35         }
36     }
37
38     input.close();
39     output.close();
40     delete [] list;
41 }
```

```

42     return numberOfSegments;
43 }

```

The function declares an array with the specified segment size in line 4, declares a data input stream for the original file in line 7, and declares a data output stream for a temporary file in line 9.

Lines 15–19 read a segment of data from the file into the array. Line 28 sorts the array. Lines 31–35 write the data in the array to the temporary file.

The number of the segments is returned in line 42. Note that every segment has `segmentSize` number of elements except the last segment, which may have a smaller number of elements.

19.8.2 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 19.18**.

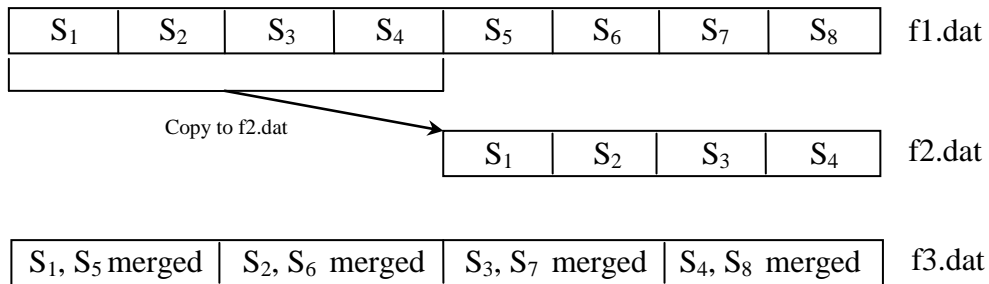


Figure 19.18

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 19.13 gives a function that copies the first half of the segments in f1.dat to f2.dat. **Listing 19.14** gives a function that merges a pair of segments in f1.dat and f2.dat. **Listing 19.15** gives a function that merges two segments.

Listing 19.13 Copying First Half Segments

```
1 // Copy the first half the number of segments from f1.dat to f2.dat
2 void copyHalfToF2(int numberOfSegments, int segmentSize,
3   fstream& f1, fstream& f2)
4 {
5   for (int i = 0; i < (numberOfSegments / 2) * segmentSize; i++)
6   {
7     int value;
8     f1.read(reinterpret_cast<char*>(& value), sizeof(value));
9     f2.write(reinterpret_cast<char*>(& value), sizeof(value));
10  }
11 }
```

Listing 19.14 Merging All Segments

```
1 // Merge all segments
2 void mergeSegments(int numberOfSegments, int segmentSize,
3   fstream& f1, fstream& f2, fstream& f3)
4 {
5   for (int i = 0; i < numberOfSegments; i++)
6   {
7     mergeTwoSegments(segmentSize, f1, f2, f3);
8   }
9
10 // f1 may have one extra segment; copy it to f3
11 while (!f1.eof())
12 {
13   int value;
14   f1.read(reinterpret_cast<char*>(&value), sizeof(value));
15   if (f1.eof()) break;
16   f3.write(reinterpret_cast<char*>(&value), sizeof(value));
17 }
18 }
```

Listing 19.15 Merging Two Segments

```
1 // Merge two segments
2 void mergeTwoSegments(int segmentSize, fstream& f1, fstream& f2,
3   fstream& f3)
4 {
5   int intFromF1;
6   f1.read(reinterpret_cast<char*>(&intFromF1), sizeof(intFromF1));
7   int intFromF2;
8   f2.read(reinterpret_cast<char*>(&intFromF2), sizeof(intFromF2));
```

```

9     int f1Count = 1;
10    int f2Count = 1;
11
12    while (true)
13    {
14        if (intFromF1 < intFromF2)
15        {
16            f3.write(reinterpret_cast<char*>
17                (&intFromF1), sizeof(intFromF1));
18            if (f1.eof() || f1Count++ >= segmentSize)
19            {
20                if (f1.eof()) break;
21                f3.write(reinterpret_cast<char*>
22                    (&intFromF2), sizeof(intFromF2));
23                break;
24            }
25            else
26            {
27                f1.read(reinterpret_cast<char*>
28                    (&intFromF1), sizeof(intFromF1));
29            }
30        }
31        else
32        {
33            f3.write(reinterpret_cast<char*>
34                (&intFromF2), sizeof(intFromF2));
35            if (f2.eof() || f2Count++ >= segmentSize)
36            {
37                if (f2.eof()) break;
38                f3.write(reinterpret_cast<char*>
39                    (&intFromF1), sizeof(intFromF1));
40                break;
41            }
42            else
43            {
44                f2.read(reinterpret_cast<char*>
45                    (&intFromF2), sizeof(intFromF2));
46            }
47        }
48    }
49
50    while (!f1.eof() && f1Count++ < segmentSize)
51    {
52        int value;
53        f1.read(reinterpret_cast<char*>
54            (&value), sizeof(value));
55        if (f1.eof()) break;
56        f3.write(reinterpret_cast<char*>
57            (&value), sizeof(value));
58    }
59
60    while (!f2.eof() && f2Count++ < segmentSize)
61    {
62        int value;
63        f2.read(reinterpret_cast<char*>(&value), sizeof(value));
64        if (f2.eof()) break;
65        f3.write(reinterpret_cast<char*>(&value), sizeof(value));
66    }
67 }

```

19.8.3 Combining Two Phases

Listing 19.16 gives the complete program for sorting `int` values in `largedata.dat` and storing the sorted data in `sortedfile.dat`.

Listing 19.16 SortLargeFile.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include "QuickSort.h"
4  #include <string>
5  using namespace std;
6
7  // Function prototype
8  void sort(string sourcefile, string targetfile);
9  int initializeSegments(int segmentSize,
10     string sourcefile, string f1);
11 void mergeTwoSegments(int segmentSize, fstream& f1, fstream& f2,
12     fstream& f3);
13 void merge(int numberOfSegments, int segmentSize,
14     string f1, string f2, string f3, string targetfile) ;
15 void copyHalfToF2(int numberOfSegments, int segmentSize,
16     fstream& f1, fstream& f2);
17 void mergeOneStep(int numberOfSegments, int segmentSize,
18     string f1, string f2, string f3);
19 void mergeSegments(int numberOfSegments, int segmentSize,
20     fstream& f1, fstream& f2, fstream& f3);
21 void copyFile(string f1, string targetfile);
22 void displayFile(string filename);
23
24 int main()
25 {
26     // Sort largedata.dat into sortedfile.dat
27     sort("largedata.dat", "sortedfile.dat");
28
29     // Display the first 100 numbers in sortedfile.dat
30     displayFile("sortedfile.dat");
31 }
32
33 // Sort sourcefile into targetfile
34 void sort(string sourcefile, string targetfile)
35 {
36     const int MAX_ARRAY_SIZE = 10000;
37
38     // Implement Phase 1: Create initial segments
39     int numberOfSegments =
40         initializeSegments(MAX_ARRAY_SIZE, sourcefile, "f1.dat");
41
42     // Implement Phase 2: Merge segments recursively
43     merge(numberOfSegments, MAX_ARRAY_SIZE,
44         "f1.dat", "f2.dat", "f3.dat", targetfile);
45 }
46
47 // Sort original file into sorted segments
```

```

48 int initializeSegments(int segmentSize, string sourceFile, string f1)
49 {
50     // Same as Listing 19.12, so omitted
51 }
52
53 void merge(int numberOfSegments, int segmentSize,
54     string f1, string f2, string f3, string targetfile)
55 {
56     if (numberOfSegments > 1)
57     {
58         mergeOneStep(numberOfSegments, segmentSize, f1, f2, f3);
59         merge((numberOfSegments + 1) / 2, segmentSize * 2,
60             f3, f1, f2, targetfile);
61     }
62     else
63     { // rename f1 as the final sorted file
64         copyFile(f1, targetfile);
65         cout << "\nSorted into the file " << targetfile << endl;
66     }
67 }
68
69 void copyFile(string f1, string targetfile)
70 {
71     fstream input;
72     input.open(f1.c_str(), ios::in | ios::binary);
73
74     fstream output;
75     output.open(targetfile.c_str(), ios::out | ios::binary);
76     int i = 0;
77     while (!input.eof()) // Continue if not end of file
78     {
79         int value;
80         input.read(reinterpret_cast<char*> (& value), sizeof(value));
81         if (input.eof()) break;
82         output.write(reinterpret_cast<char*> (& value), sizeof(value));
83     }
84
85     input.close();
86     output.close();
87 }
88
89 void mergeOneStep(int numberOfSegments, int segmentSize, string f1,
90     string f2, string f3)
91 {
92     fstream f1Input;
93     f1Input.open(f1.c_str(), ios::in | ios::binary);
94
95     fstream f2Output;
96     f2Output.open(f2.c_str(), ios::out | ios::binary);
97
98     // Copy half number of segments from f1.dat to f2.dat
99     copyHalfToF2(numberOfSegments, segmentSize, f1Input, f2Output);
100     f2Output.close();
101
102     // Merge remaining segments in f1 with segments in f2 into f3
103     fstream f2Input;
104     f2Input.open(f2.c_str(), ios::in | ios::binary);
105     fstream f3Output;
106     f3Output.open(f3.c_str(), ios::out | ios::binary);
107

```



```

108     mergeSegments(numberOfSegments / 2, segmentSize, f1Input, f2Input,
f3Output);
109
110     f1Input.close();
111     f2Input.close();
112     f3Output.close();
113 }
114
115 // Copy first half number of segments from f1.dat to f2.dat
116 void copyHalfToF2(int numberOfSegments, int segmentSize, fstream& f1,
fstream& f2)
117 {
118     // Same as Listing 19.13, so omitted
119 }
120
121 // Merge all segments
122 void mergeSegments(int numberOfSegments, int segmentSize, fstream& f1,
fstream& f2, fstream& f3)
123 {
124     // Same as Listing 19.14, so omitted
125 }
126
127 // Merge two segments
128 void mergeTwoSegments(int segmentSize, fstream& f1, fstream& f2,
fstream& f3)
129 {
130     // Same as Listing 19.15, so omitted
131 }
132
133 // Display the first 10 numbers in the specified file
134 void displayFile(string filename)
135 {
136     fstream input(filename.c_str(), ios::in | ios::binary);
137     int value;
138     for (int i = 0; i < 100; i++)
139     {
140         input.read(reinterpret_cast<char*>(& value), sizeof(int));
141         cout << value << " ";
142     }
143     input.close();
144 }

```

Sample output

```
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 19.11, CreateLargeFile.cpp, to create largedata.dat. Invoking

`sort("largedata.dat", "sortedfile.dat")` (line 27) reads data from `largedata.dat` and writes

sorted data `sortedfile.dat`. Invoking `displayFile("sortedfile.dat")` (line 30) displays the first

100 numbers in the specified file. Note that the files are created using binary I/O. You cannot view them using a

text editor such as Notepad.

The `sort` function first creates initial segments from the original array and stores the sorted segments in a new file `f1.dat` (lines 39-40), and then produces a sorted file in `targetfile` (lines 43-44).

The `merge` function

```
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` function is invoked recursively with many merge steps. Each merge step reduces the `numberOfSegments` by half and doubles the sorted segment size. After completing one merge step, the next merge step merges the new segments in `f3` to `f2`, using `f1` to assist the merge. So the statement to invoke the new merge function is (line 97-98)

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

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

The recursive `merge` function ends when `numberOfSegments` is 1. In this case, `f1` contains sorted data. File `f1` is copied to `targetfile` in line 102.

19.8.4 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 $\frac{n}{2^3 c}$. After $\log(\frac{n}{c})$ merge steps, the number of segments has been reduced to 1.

Therefore, the total number of merge steps is $\log(\frac{n}{c})$.

In each merge step, half the number of segments are read from file **f1** and 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 $\log(\frac{n}{c})$, the total number of I/Os is

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

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

Check point

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

19.23 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
- bucket sort
- external sort
- complete binary tree
- heap
- heap sort
- height of a heap
- merge sort
- quick sort
- radix sort

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-case 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 sorts* 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 www.cs.armstrong.edu/liang/cpp3e/quiz.html.

Programming Exercises

- 19.1 (*Generic bubble sort*) Write a generic function for bubble sort.
- 19.2 (*Generic merge sort*) Write a generic function for merge sort.
- 19.3 (*Generic quick sort*) Write a generic function for quick sort.

19.4 (*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 last elements in the list.

19.5 (*Generic heap sort*) Write a test program that invokes the generic sort function to sort an array of `int` values, an array of `double` values, and an array of strings.

19.6 (*Checking order*) Write the following overloaded functions that check whether an array is ordered in ascending order or descending order. By default, the function checks ascending order. To check descending order, pass `false` to the ascending argument in the function.

```
// T is a generic type
bool ordered(T list[], int size)

// T is a generic type
bool ordered(T list[], int size, bool ascending)
```

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

19.8 (*Execution time for sorting*) Write a program that obtains the execution time of selection sort, insertion sort, bubble sort, merge sort, quick sort, and heap sort for input size 500,000, 1,000,000, 1,500,000, 2,000,000, 2,500,000, and 3,000,000. Your program should print a table like this:

Array size	Selection Sort	Insertion Sort	Bubble Sort	Merge Sort	Quick Sort	Heap Sort
500000						
1000000						
1500000						
2000000						
2500000						
3000000						

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

```
long startTime = time(0);  
perform the task;  
long endTime = time(0);  
long executionTime = endTime - startTime;
```

- 19.9 (Execution time for external sorting) Write a program that obtains the execution time of external sort 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	5000000	10000000	15000000	20000000	25000000	30000000
Time						