CHAPTER 6

C++ Sorting

LESSON 6.1

Sorting Algorithm

Selection

There are several sorting algorithms in the literature. In this chapter, we discuss some of the most commonly used sorting algorithms. To compare the performance of these algorithms, we also provide the analysis of these algorithms. These sorting algorithms can be applied to either array-based lists or linked lists. We will specify whether the algorithm being developed is for array-based lists or linked lists.

The functions implementing these sorting algorithms are included as public members of the related class. (For example, for an array-based list, these are the members of the class arrayListType.) By doing so, the algorithms have direct access to the list elements.

Suppose that the sorting algorithm selection sort (described in the next section) is to be applied to array-based lists. The following statements show how to include selection sort as a member of the class arrayListType:

```
template <class elemType>
class arrayListType
{
public:
        void selectionSort();
...
};
```

Selection Sort Array Based Lists

In selection sort, a list is sorted by selecting elements in the list, one at a time, and moving them to their proper positions. This algorithm finds the location of the smallest element in the unsorted portion of the list and moves it to the top of the unsorted portion (that is, the whole list) of the list. The first time we locate the smallest item in the entire list, the second time we locate the smallest item in the list starting from the second element in the list, and so on. Selection sort described here is designed for array-based lists.

Suppose you have the list shown in Figure 6.1.1.

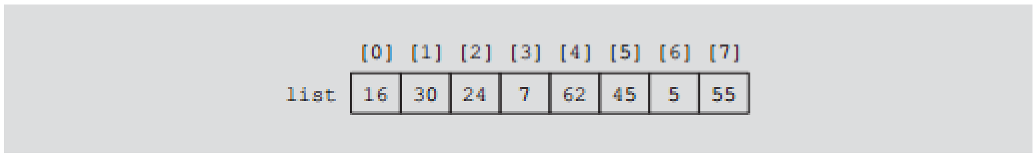

Figure 6.1.1 List of 8 elements

Figure 6.1.2 shows the elements of list in the first iteration.

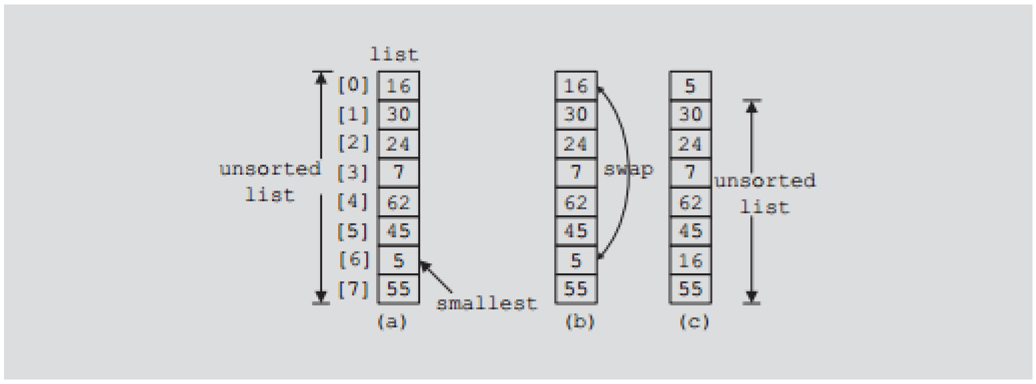

Figure 6.1.2 Elements of list during the first iteration

Initially, the entire list is unsorted. So we find the smallest item in the list. The smallest item is at position 6, as shown in Figure 6.1.2(a). Because this is the smallest item, it must be moved to position 0. So we swap 16 (that is, list[0]) with 5 (that is, list[6]), as shown in Figure 6.1.2 (b). After swapping these elements, the resulting list is as shown in Figure 6.1.2 (c).

Figure 6.1.3 shows the elements of list in the second iteration.

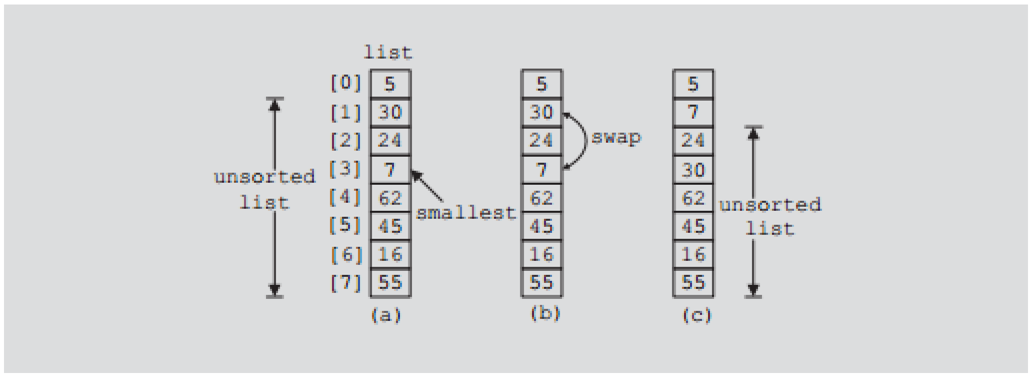

Figure 6.1.3 Elements of list during the second iteration

Now the unsorted list is list[1]...list[7]. So we find the smallest element in the unsorted list. The smallest element is at position 3, as shown in Figure 6.1.3 (a). Because the smallest element in the unsorted list is at position 3, it must be moved to position 1. So we swap 7 (that is, list[3]) with 30 (that is, list[1]), as shown in Figure 6.1.3(b). After swapping list[1] with list[3], the resulting list is as shown in Figure 6.1.3 (c).

Now the unsorted list is list[2]...list[7]. So we repeat the preceding process of finding the (position of the) smallest element in the unsorted portion of the list and moving it to the beginning of the unsorted portion of the list. Selection sort, thus, involves the following steps.

In the unsorted portion of the list:

1. Find the location of the smallest element.
2. Move the smallest element to the beginning of the unsorted list.

Initially, the entire list, list[0]...list[length-1], is the unsorted list. After executing Steps 1 and 2 once, the unsorted list is list[1]...list[length-1]. After executing Steps 1 and 2 a second time, the unsorted list is list[2]...list[length-1], and so on.

We can keep track of the unsorted portion of the list and repeat Steps a and b with the help of a for loop as follows:

```
for (index = 0; index < length - 1; index++)
{
    1. Find the location, smallestIndex, of the smallest element
       in list[index]...list[length - 1].
    2. Swap the smallest element with list[index]. That is, swap
       list[smallestIndex] with list[index].
}
```

The first time through the loop, we locate the smallest element in list[0]...list[length-1] and swap this smallest element with list[0]. The second time through the loop, we locate the smallest element in list[1]...list[length-1] and swap this smallest element with list[1], and so on. This process continues until the length of the unsorted list is 1. (Note that a list of length 1 is sorted.) It, therefore, follows that to implement selection sort, we need to implement Steps 1 and 2.

Given the starting index, first, and the ending index, last, of the list, the following C++ function returns the index of the smallest element in list[first]...list[last]:

```
template <class elemType>
int arrayListType<elemType>::minLocation(int first, int last)
{
        int minIndex;
        minIndex = first;
        for (int loc = first + 1; loc <= last; loc++)
                if( list[loc] < list[minIndex])
                        minIndex = loc;
        return minIndex;
} //end minLocation
```

Given the locations in the list of the elements to be swapped, the following C++ function, swap, swaps those elements:

```
template <class elemType>
void arrayListType<elemType>::swap(int first, int second)
{
        elemType temp;
        temp = list[first];
        list[first] = list[second];
        list[second] = temp;
}//end swap
```

We can now complete the definition of the function selectionSort:

```
template <class elemType>
void arrayListType<elemType>::selectionSort()
{
        int minIndex;
        for (int loc = 0; loc < length - 1; loc++)
        {
                minIndex = minLocation(loc, length - 1);
                swap(loc, minIndex);
        }
}
```

You can add the functions to implement selection sort in the definition of the class arrayListType as follows:

```
template<class elemType>
class arrayListType
{
        public:
                //Place the definitions of the function given earlier here.
        void selectionSort();
        ...
        private:
                //Place the definitions of the members given earlier
                here.
        void swap(int first, int second);
        int minLocation(int first, int last);
};
```

Example:

```
#include <iostream>                                      //Line 1
#include "arrayListType.h"                               //Line 2

using namespace std;                                     //Line 3

int main()                                               //Line 4
{                                                        //Line 5
    arrayListType<int> list;                             //Line 6
    int num;                                             //Line 7

    cout << "Line 8: Enter numbers ending with -999"
         << endl;                                        //Line 8

    cin >> num;                                          //Line 9

    while (num != -999)                                  //Line 10
    {                                                    //Line 11
        list.insert(num);                                //Line 12
        cin >> num;                                      //Line 13
    }                                                    //Line 14

    cout << "Line 15: The list before sorting:" << endl; //Line 15
    list.print();                                        //Line 16
    cout << endl;                                        //Line 17

    list.selectionSort();                                //Line 18

    cout << "Line 19: The list after sorting:" << endl;  //Line 19
    list.print();                                        //Line 20
    cout << endl;                                        //Line 21

    return 0;                                            //Line 22
}                                                        //Line 23
```

Sample Run: In this sample run, the user input is shaded.

```
Line 8: Enter numbers ending with -999
34 67 23 12 78 56 36 79 5 32 66 -999
Line 15: The list before sorting:
34 67 23 12 78 56 36 79 5 32 66
Line 19: The list after sorting:
5 12 23 32 34 36 56 66 67 78 79
```

For the most part, the preceding output is self-explanatory. Notice that the statement in Line 12 calls the function insert of the class arrayListType. Similarly, the statements in Lines 16 and 20 call the function print of the class arrayListType. The statement in Line 18 calls the function selectionSort to sort the list.

Note:

1. Selection sort can also be implemented by selecting the largest element in the (unsorted portion of the) list and moving it to the bottom of the list. You can easily implement this form of selection sort by altering the if statement in the function minLocation, and

passing the appropriate parameters to the corresponding function and the function swap, when these functions are called in the function selectionSort.

2. Selection sort can also be applied to linked lists. The general algorithm is the same, and the details are left as an exercise for you.

LESSON 6.2

Insertion Sort

The previous section described and analyzed the selection sort algorithm. It was shown that if n ¼ 1000, the number of key comparisons is approximately 500,000, which is quite high. This section describes the sorting algorithm called insertion sort, which tries to improve—that is, reduce—the number of key comparisons.

Insertion sort sorts the list by moving each element to its proper place. Consider the list given in Figure 6.2.1.

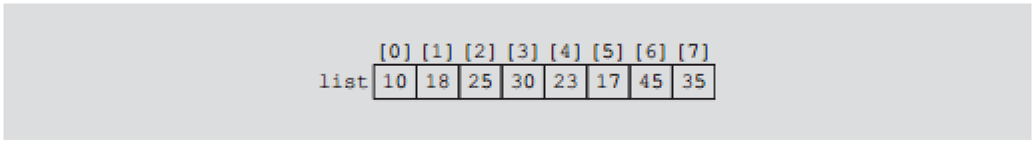

Figure 6.2.1 List

The length of the list is 8. In this list, the elements list[0], list[1], list[2], and list[3] are in order. That is, list[0]...list[3] is sorted, as shown in Figure 6.2.2(a).

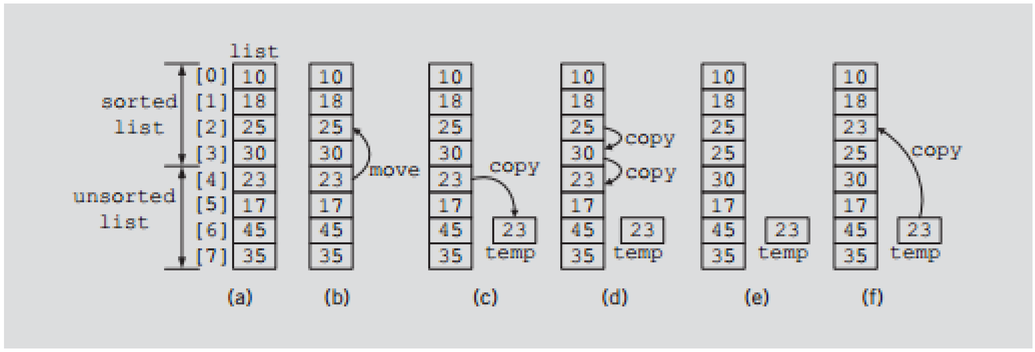

Figure 6.2.2 list elements while moving list[4] to its proper place

Next, we consider the element list[4], the first element of the unsorted list. Because list[4] < list[3], we need to move the element list[4] to its proper location. It follows that element list[4] should be moved to list[2], as shown in Figure 6.2.2(b). To move list[4] into list[2], first we copy list[4] into temp, a temporary memory space—see Figure 6.2.2 (c).

Next, we copy list[3] into list[4], and then list[2] into list[3], as shown in Figure 6.2.2 (d). After copying list[3] into list[4] and list[2] into list[3], the list is as shown in Figure 6.2.2 (e). Next we copy temp into list[2]. Figure 6.2.2 (f) shows the resulting list.

Now list[0]...list[4] is sorted and list[5]...list[7] is unsorted. We repeat this process on the resulting list by moving the first element of the unsorted list into the sorted list in the proper place.

From this discussion, we see that during the sorting phase the array containing the list is divided into two sublists, upper and lower. Elements in the upper sublist are sorted; elements in the lower sublist are to be moved to the upper sublist in their proper places one at a time. We use an index—say, firstOutOfOrder—to point to the first element in the lower sublist; that is, firstOutOfOrder gives the index of the first element in the unsorted portion of the array. Initially, firstOutOfOrder is initialized to 1.

This discussion translates into the following pseudoalgorithm:

```
for (firstOutOfOrder = 1; firstOutOfOrder < length; firstOutOfOrder++)
  if (list[firstOutOfOrder] is less than list[firstOutOfOrder - 1])
  {
       copy list[firstOutOfOrder] into temp

       initialize location to firstOutOfOrder

       do
       {
           a. move list[location - 1] one array slot down
           b. decrement location by 1 to consider the next element
              sorted of the portion of the array
       }
       while (location > 0 && the element in the upper list at
              location - 1 is greater than temp)
  }
copy temp into list[location]
```

The length of this list is 8; that is, length = 8. We initialize firstOutOfOrder to 1 (see Figure 6.2.3).

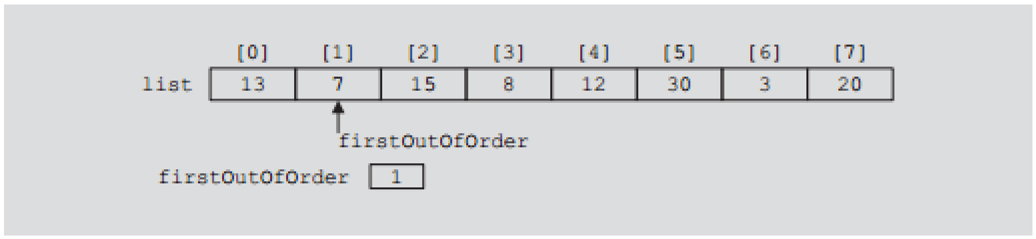

Figure 6.2.3 firstOutOfOrder = 1

Now list[firstOutOfOrder] = 7, list[firstOutOfOrder - 1] = 13 and 7 < 13, and the expression in the if statement evaluates to true, so we execute the body of the if statement.

temp = list[firstOutOfOrder] = 7
location = firstOutOfOrder = 1

Next, we execute the do...while loop.

list[1] = list[0] = 13   (copy list[0] into list[1]) location = 0 (decrement location)

The do...while loop terminates because location = 0. We copy temp into list[location]—that is, into list[0]. Figure 6.2.4 shows the resulting list.

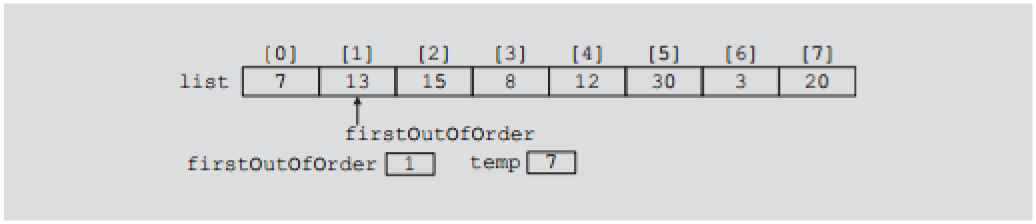

Figure 6.2.4 list after the first iteration of insertion sort

Now suppose that we have the list given in Figure 6.2.5(a).

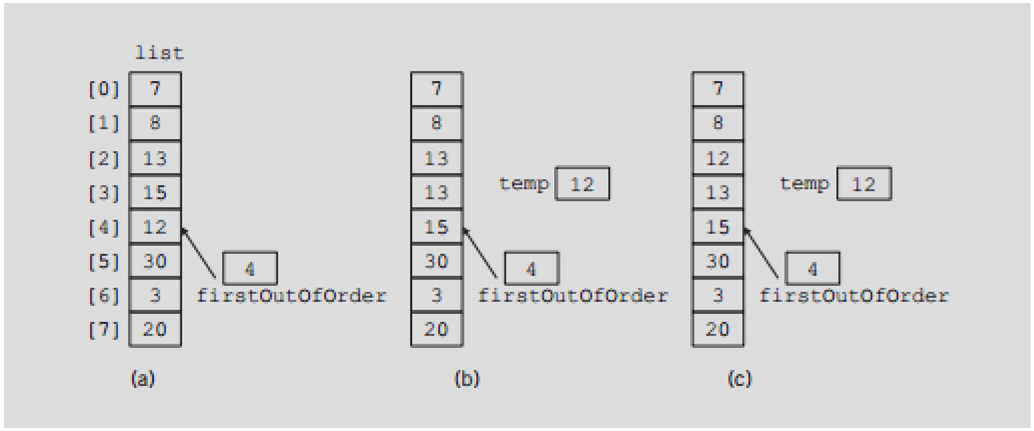

Figure 6.2.5 list elements while moving list[4] to its proper place

Here list[0]...list[3], or the elements list[0], list[1], list[2], and list[3], are in order. Now firstOutOfOrder = 4. Because list[4] < list[3], the element list[4], which is 12, needs to be moved to its proper location.

As before:

temp = list[firstOutOfOrder] = 12
location = firstOutOfOrder = 4

First, we copy list[3] into list[4] and decrement location by 1. Then we copy list[2] into list[3] and again decrement location by 1. Now the value of location is 2. At this point, the list is as shown in Figure 6.2.5 (b).

Next, because list[1] < temp, the do...while loop terminates. At this point, location is 2, so we copy temp into list[2]. That is, list[2] = temp = 12. Figure 6.2.5 (c) shows the resulting list.

Suppose that we have the list given in Figure 6.2.6.

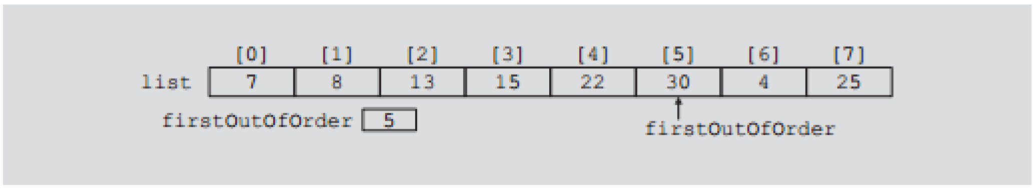

Figure 6.2.6 First out-of-order element is at position 5

Here list[0]...list[4], or the elements list[0], list[1], list[2], list[3], and list[4], are in order. Now firstOutOfOrder = 5. Because list[5] > list[4], the if statement evaluates to false. So the body of the if statement does not execute and the next iteration of the for loop, if any, takes place. Note that this is the case when the firstOutOfOrder element is already at the proper place. So we simply need to advance firstOutOfOrder to the next array element, if any.

We can repeat this process for the remaining elements of list to sort list.

The following C++ function implements the previous algorithm:

```
template <class elemType>
void arrayListType<elemType>::insertionSort()
{
    int firstOutOfOrder, location;
    elemType temp;

    for (firstOutOfOrder = 1; firstOutOfOrder < length;
                              firstOutOfOrder++)
        if (list[firstOutOfOrder] < list[firstOutOfOrder - 1])
        {
            temp = list[firstOutOfOrder];
            location = firstOutOfOrder;

            do
            {
                list[location] = list[location - 1];
                location--;
            }
            while (location > 0 && list[location - 1] > temp);

            list[location] = temp;
        }
} //end insertionSort
```

Insertion Sort Linked List Based lists

Insertion sort can also be applied to linked lists. Therefore, this section describes insertion sort for linked lists. Consider the linked list shown in Figure 6.2.7.

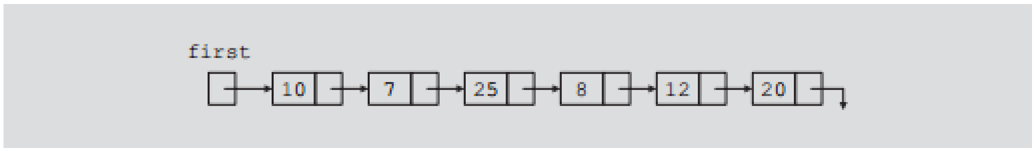

Figure 6.2.7 Linked list

In Figure 6.2.7, first is a pointer to the first node of the linked list.

If the list is stored in an array, we can traverse the list in either direction using an index variable. However, if the list is stored in a linked list, we can traverse the list in only one direction starting at the first node because the links are only in one direction, as shown in Figure 6.2.7. Therefore, in the case of a linked list, to find the location of the node to be inserted, we do the following. Suppose that firstOutOfOrder is a pointer to the node that is to be moved to its proper location, and lastInOrder is a pointer to the last node of the sorted portion of the list. For example, see the linked list in Figure 6.2.8.

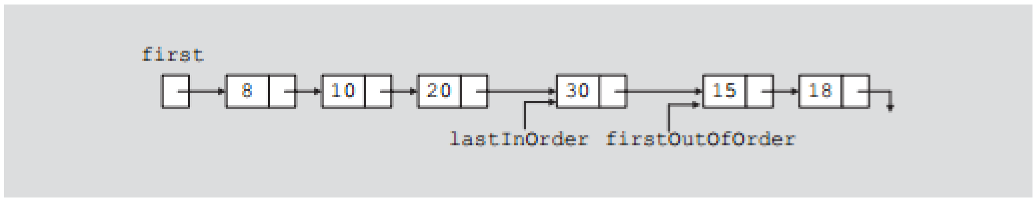

Figure 6.2.8 Linked list and pointers lastInOrder and firstOutOfOrder

First, we compare the info of firstOutOfOrder with the info of the first node. If the info of firstOutOfOrder is smaller than the info of first, then the node firstOutOfOrder is to be moved before the first node of the list; otherwise, we search the list starting at the second node to find the location where to move firstOutOfOrder. As usual, we search the list using two pointers, for example current and trailCurrent. The pointer trailCurrent points to the node just before current. In this case, the node firstOutOfOrder is to be moved between trailCurrent and current. Of course, we also handle any special cases such as an empty list, a list with only one node, or a list in which the node firstOutOfOrder is already in the proper place.

This discussion translates into the following algorithm:

```
if (firstOutOfOrder->info is less than first->info)
    move firstOutOfOrder before first
else
{
    set trailCurrent to first
    set current to the second node in the list first->link;

     //search the list
    while (current->info is less than firstOutOfOrder->info)
    {
        advance trailCurrent;
        advance current;
    }

    if (current is not equal to firstOutOfOrder)
    {    //insert firstOutOfOrder between current and trailCurrent
        lastInOrder->link = firstOutOfOrder->link;
        firstOutOfOrder->link = current;
        trailCurrent->link = firstOutOfOrder;
    }
    else   //firstOutOfOrder is already at the first place
        lastInOrder = lastInOrder->link;
}
```

Let us illustrate this algorithm on the list shown in Figure 6.2.9. We consider several cases.

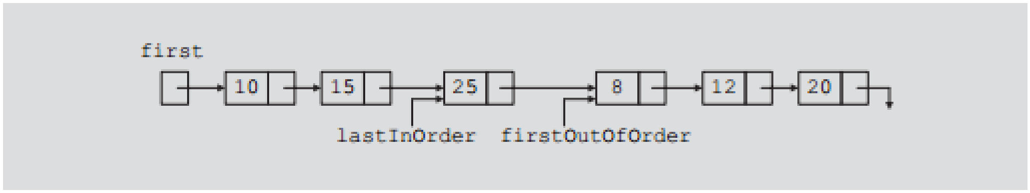

Figure 6.2.9 Linked list and pointers lastInOrder and firstOutOfOrder

Case 1: Because firstOutOfOrder->info is less than first->info, the node firstOutOfOrder is to be moved before first. So we adjust the necessary links, and Figure 6.2.10 shows the resulting list.

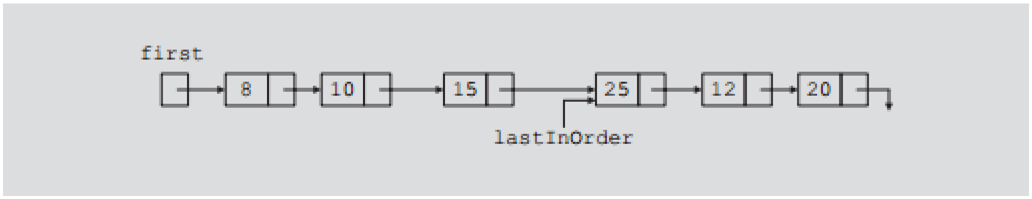

Figure 6.2.10 Linked list after moving the node with info 8 to the beginning

Case 2: Consider the list shown in Figure 6.2.11

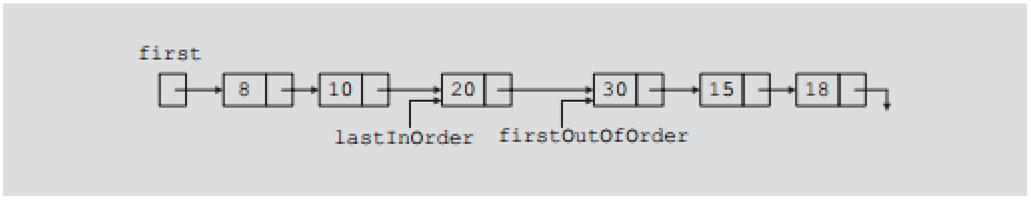

Figure 6.2.11 Linked list and pointers lastInOrder and firstOutOfOrder

Because firstOutOfOrder->info is greater than first->info, we search the list to find the place where firstOutOfOrder is to be moved. As explained previously, we use the pointers trailCurrent and current to traverse the list. For this list, these pointers end up at the nodes as shown in Figure 6.2.12.

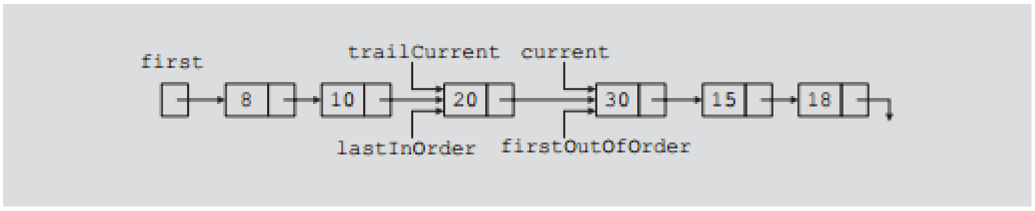

Figure 6.2.12 Linked list and pointers trailCurrent and current

Because current is the same as firstOutOfOrder, the node firstOutOfOrder is in the right place. So no adjustment of the links is necessary.

Case 3: Consider the list in Figure 6.2.13.

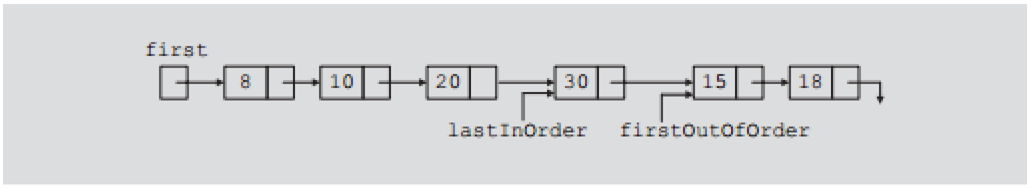

Figure 6.2.13 Linked list and pointers lastInOrder and firstOutOfOrder

Because firstOutOfOrder->info is greater than first->info, we search the list to find the place where firstOutOfOrder is to be moved. As in Case 2, we use the pointers trailCurrent and current to traverse the list. For this list, these pointers end up at the nodes as shown in Figure 6.2.14.

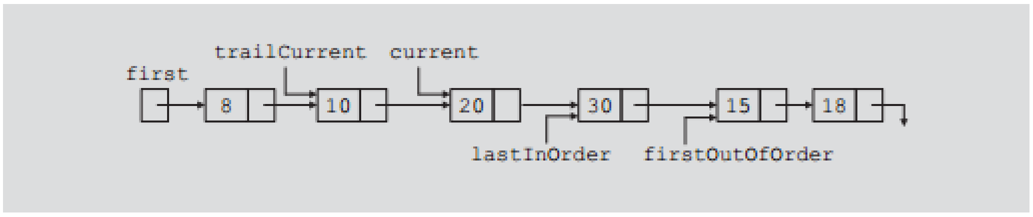

Figure 6.2.14 Linked list and pointers trailCurrent and current

Now, firstOutOfOrder is to be moved between trailCurrent and current. So we adjust the necessary links and obtain the list as shown in Figure 6.2.15.

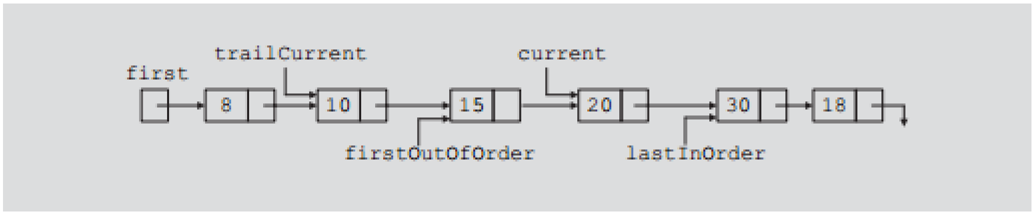

Figure 6.2.15 Linked list after moving firstOutOfOrder between trailCurrent and current

We now write the C++ function, linkedInsertionSort, to implement the previous algorithm:

```
template <class elemType>
void unorderedLinkedList<elemType>::linkedInsertionSort()
{
    nodeType<elemType> *lastInOrder;
    nodeType<elemType> *firstOutOfOrder;
    nodeType<elemType> *current;
    nodeType<elemType> *trailCurrent;

    lastInOrder = first;

    if (first == NULL)
       cerr << "Cannot sort an empty list." << endl;
    else if (first->link == NULL)
        cout << "The list is of length 1. "
             << "It is already in order." << endl;
    else
        while (lastInOrder->link != NULL)
        {
            firstOutOfOrder = lastInOrder->link;

            if (firstOutOfOrder->info < first->info)
            {
                lastInOrder->link = firstOutOfOrder->link;
                firstOutOfOrder->link = first;
                first = firstOutOfOrder;
            }

    else
    {
        trailCurrent = first;
        current = first->link;

        while (current->info < firstOutOfOrder->info)
        {
            trailCurrent = current;
            current = current->link;
        }

        if (current != firstOutOfOrder)
        {
            lastInOrder->link = firstOutOfOrder->link;
            firstOutOfOrder->link = current;
            trailCurrent->link = firstOutOfOrder;
        }
        else
            lastInOrder = lastInOrder->link;
    }
  } //end while
} //end linkedInsertionSort
```

LESSON 6.3

Quicksort

Quicksort described here is for array-based lists. The algorithm for linked lists can be developed in a similar manner and is left as an exercise for you.

In quicksort, the list is partitioned in such a way that combining the sorted lowerSublist and upperSublist is trivial. Therefore, in quicksort, all the sorting work is done in partitioning the list. Because all the sorting work occurs during the partition, we first describe the partition procedure in detail.

To partition the list into two sublists, first we choose an element of the list called the pivot. The pivot is used to divide the list into two sublists: the lowerSublist and the upperSublist. The elements in the lowerSublist are smaller than the pivot, and the elements in the upperSublist are greater than the pivot. For example, consider the list in Figure 6.3.1.

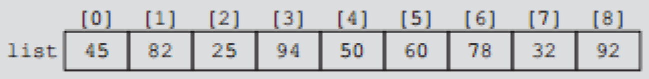

Figure 6.3.1 list before the partition

There are several ways to determine the pivot. However, the pivot is chosen so that, it is hoped, the lowerSublist and upperSublist are of nearly equal size. For illustration purposes, let us choose the middle element of the list as the pivot. The partition procedure that we describe partitions this list using the pivot as the middle element, in our case 50, as shown in Figure 6.3.2.

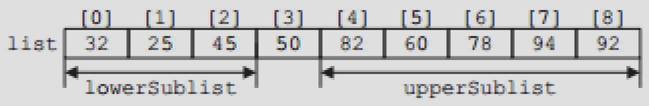

Figure 6.3.2 list after the partition

From Figure 6.3.2, it follows that after partitioning list into lowerSublist and upperSublist, the pivot is in the right place. Thus, after sorting lowerSublist and upperSublist, combining the two sorted sublists is trivial.

The partition algorithm is as follows: (We assume that pivot is chosen as the middle element of the list.)

1. Determine the pivot, and swap the pivot with the first element of the list. Suppose that the index smallIndex points to the last

element smaller than the pivot. The index smallIndex is initialized to the first element of the list.

2. For the remaining elements in the list (starting at the second element) If the current element is smaller than the pivot
    a. Increment smallIndex.
    b. Swap the current element with the array element pointed to by smallIndex.
3. Swap the first element, that is, the pivot, with the array element pointed to by smallIndex.

Step 2 can be implemented using a for loop, with the loop starting at the second element of the list.

Step 1 determines the pivot and moves the pivot in the first array position. During the execution of Step 2, the elements of the list get arranged as shown in Figure 6.3.3. (Suppose the name of the array containing the list elements is list.)

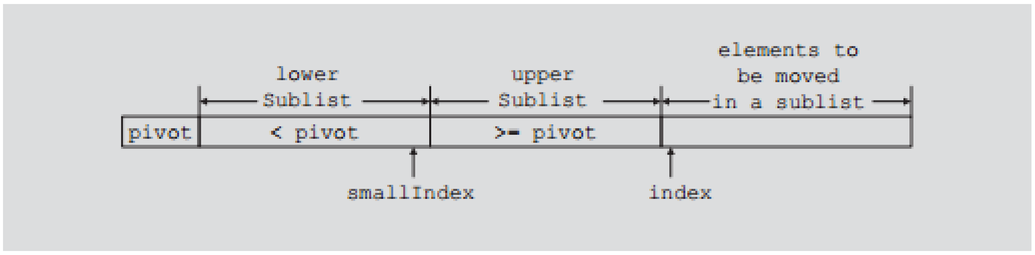

Figure 6.3.3 List during the execution of Step 2

As shown in Figure 6.3.3, the pivot is in the first array position, elements in lowerSublist are less than the pivot, and elements in the upperSublist are greater than or equal to the pivot. The variable smallIndex contains the index of the last element of lowerSublist and the variable index contains the index of the next element that needs to be moved either in lowerSublist or in upperSublist. As explained in Step 2, if the next element of the list (that is, list[index]) is less than the pivot, we advance smallIndex to the next array position and swap list[index] with list[smallIndex]. Next we illustrate Step 2.

Suppose that list is as given in Figure 6.3.4.

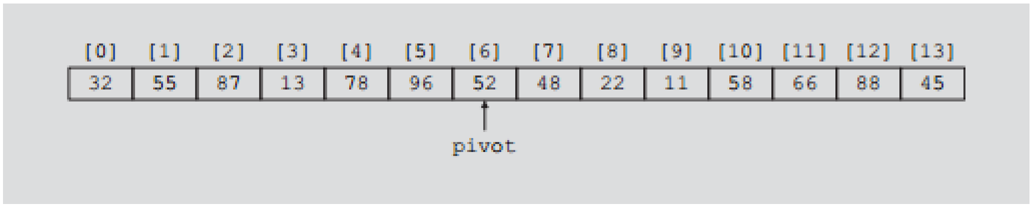

Figure 6.3.4 list before sorting

For the list in Figure 6.3.4, the pivot is at position 6. After moving the pivot at the first array position, the list is shown in Figure 6.3.5. (Notice that in Figure 6.3.5, 52 is swapped with 32.)

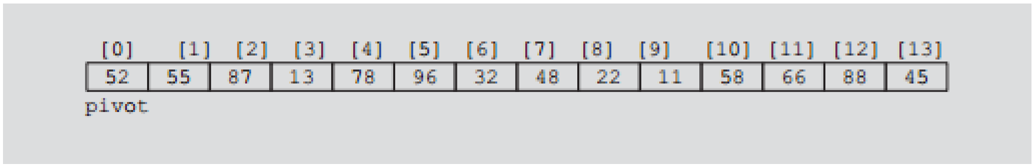

Figure 6.3.5 List after moving pivot at the first array position

Suppose that after executing Step 2 a few times, the list is as shown in Figure 6.3.6.

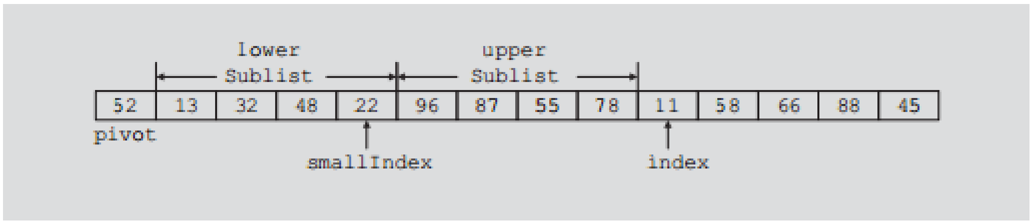

Figure 6.3.6 List after a few iterations of Step 2

As shown in Figure 6.3.6, the next element of the list that needs to be moved in a sublist is indicated by index. Because list[index] < pivot, we need to move the element list[index] in lowerSublist. To do so, we first advance smallIndex to the next array position and then swap list[smallIndex] with list[index]. The resulting list is shown in Figure 6.3.7. (Notice that 11 is swapped with 96.)

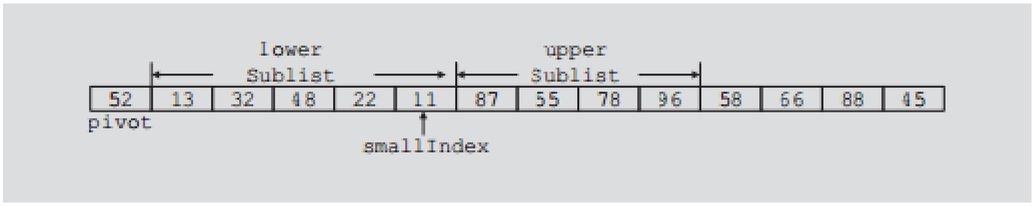

Figure 6.3.7 List after moving 11 into lowerSublist

Now consider the list in Figure 6.3.8.

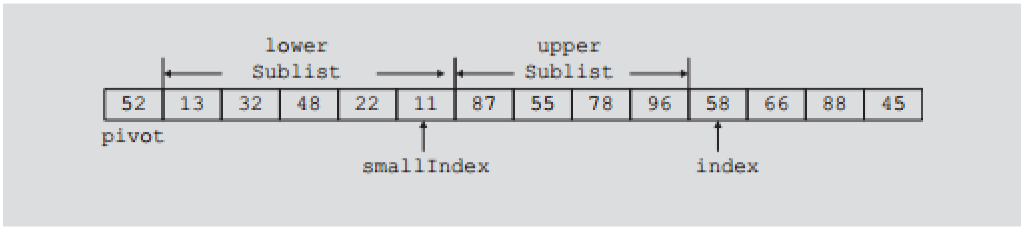

Figure 6.3.8 List before moving 58 into a sublist

For the list in Figure 6.3.8, list[index] is 58, which is greater than the pivot. Therefore, list[index] is to be moved in upperSublist. This is accomplished by leaving 58 at its position and increasing the size of upperSublist, by one, to the next array position. After moving 58 into upperSublist, the list is as shown in Figure 6.3.9.

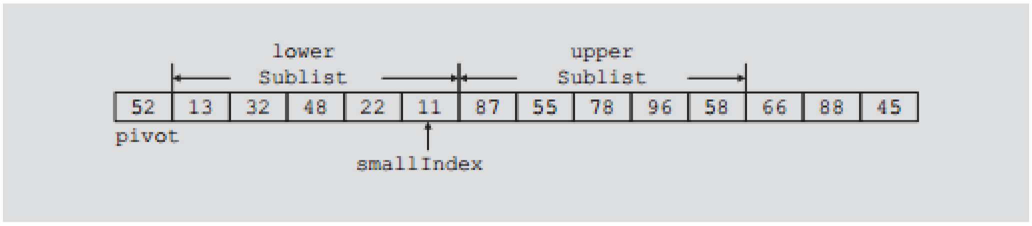

Figure 6.3.9 List after moving 58 into upperSublist

After moving the elements that are less than the pivot into lowerSublist and elements that are greater than the pivot into upperSublist (that is, after completely executing Step 2), the resulting list is as shown in Figure 6.3.10.

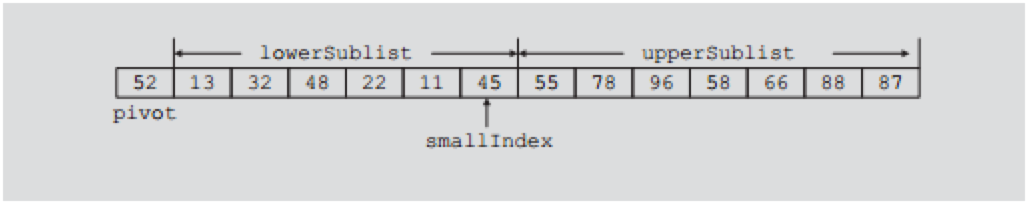

Figure 6.3.10 List elements after arranging into lowerSublist and upperSublist

Next, we execute Step 3 and move 52, the pivot, to the proper position in the list. This is accomplished by swapping 52 with 45. The resulting list is shown in Figure 6.3.11.

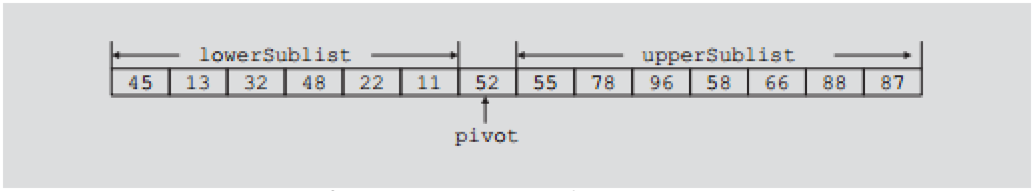

Figure 6.3.11 List after swapping 52 with 45

As shown in Figure 6.3.11, the preceding algorithm, Steps 1, 2, and 3, partitions the list into two sublists such that the elements less than the pivot are in lowerSublist and elements greater than or equal to the pivot are in upperSublist.

To partition the list into the lower and upper sublists, we only need to keep track of the last element of the lowerSublist and the next element of the list that needs to be moved either in lowerSublist or in upperSublist. In fact, upperSublist is between the two indices smallIndex and index.

We now write the function, partition, to implement the preceding partition algorithm. After rearranging the elements of the list, the function returns the location of the pivot so that we can determine the starting and ending locations of the sublists. Also, because the function partition will be a member of the class, it has direct access to the array containing the list. Thus, to partition a list, we need to pass only the starting and ending indices of the list.

```
template <class elemType>
int arrayListType<elemType>::partition(int first, int last)
{
    elemType pivot;

    int index, smallIndex;

    swap(first, (first + last) / 2);

    pivot = list[first];
    smallIndex = first;

    for (index = first + 1; index <= last; index++)
        if (list[index] < pivot)
        {
            smallIndex++;
            swap(smallIndex, index);
        }

    swap(first, smallIndex);

    return smallIndex;
}
```

As you can see from the definition of the function partition, certain elements of the list need to be swapped. The following function, swap, accomplishes this task:

```
template <class elemType>
void arrayListType<elemType>::swap(int first, int second)
{
    elemType temp;

    temp = list[first];
    list[first] = list[second];
    list[second] = temp;
}
```

Once the list is partitioned into lowerSublist and upperSublist, we again apply the quicksort method to sort the two sublists. Because both sublists are sorted using the same quicksort algorithm, the easiest way to implement this algorithm is to use recursion.

Therefore, this section gives the recursive version of quicksort. As explained previously, after rearranging the elements of the list, the function partition returns the index of the pivot so that the starting and ending indices of the sublists can be determined. Given the starting and ending indices of a list, the following function, recQuickSort, implements the recursive version of quicksort:

```
template <class elemType>
void arrayListType<elemType>::recQuickSort(int first, int last)
{
    int pivotLocation;

    if (first < last)
    {
        pivotLocation = partition(first, last);
        recQuickSort(first, pivotLocation - 1);
        recQuickSort(pivotLocation + 1, last);
    }
}
```

Finally, we write the quicksort function, quickSort, that calls the function recQuickSort of the original list:

```
template <class elemType>
void arrayListType<elemType>::quickSort()
{
    recQuickSort(0, length -1);
}
```

LESSON 6.4

Mergesort

Like quicksort, mergesort uses the divide-and-conquer technique to sort a list. Mergesort also partitions the list into two sublists, sorts the sublists, and then combines the sorted sublists into one sorted list. This section describes mergesort for linked list-based lists

Mergesort and quicksort differ in how they partition the list. As discussed earlier, quicksort first selects an element in the list, called pivot, and then partitions the list so that the elements in one sublist are less than pivot and the elements in the other sublist are greater than or equal to pivot. By contrast, mergesort divides the list into two sublists of nearly equal size. For example, consider the list whose elements are as follows:

list: 35 28 18 45 62 48 30 38

Mergesort partitions this list into two sublists as follows:

first sublist: 35 28 18 45
second sublist: 62 48 30 38

The two sublists are sorted using the same algorithm (that is, mergesort) used on the original list. Suppose that we have sorted the two sublists. That is, suppose that the lists are now as follows:

first sublist: 18 28 35 45
second sublist: 30 38 48 62

Next, mergesort combines, that is, merges, the two sorted sublists into one sorted list.

Figure 6.4.1 further illustrates the mergesort process.

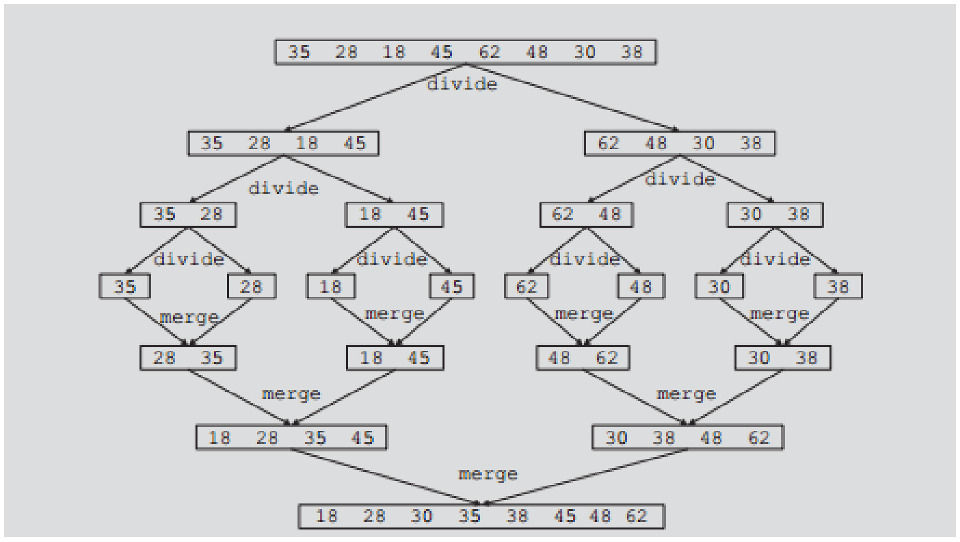

Figure 6.4.1
Mergesort algorithm

From Figure 6.4.1, it is clear that in mergesort, most of the sorting work is done in merging the sorted sublists.

The general algorithm for mergesort is as follows:

```
if the list is of a size greater than 1
{
        1. Divide the list into two sublists.
        2. Mergesort the first sublist.
        3. Mergesort the second sublist.
        4. Merge the first sublist and the second sublist.
}
```

As remarked previously, after dividing the list into two sublists—the first sublist and the second sublist—these two sublists are sorted using mergesort. In other words, we use recursion to implement mergesort.

We next describe the necessary algorithm to:

- Divide the list into two sublists of nearly equal size.
- Mergesort both sublists.
- Merge the sorted sublists.

Divide

Because data is stored in a linked list, we do not know the length of the list. Furthermore, a linked list is not a random access data structure. Therefore, to divide the list into two sublists, we need to find the middle node of the list.

Consider the list in Figure 6.4.2.

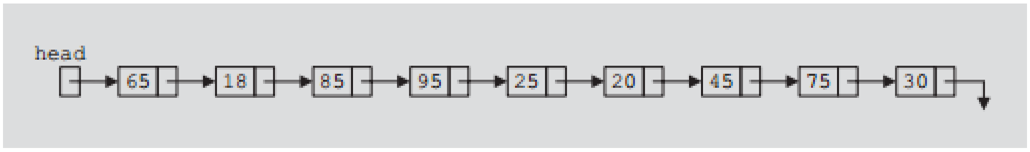

Figure 6.4.2 Unsorted linked list

To find the middle of the list, we traverse the list with two pointers—say, middle and current. The pointer middle is initialized to the first node of the list. Because this list has more than two nodes, we initialize current to the third node. (Recall that we sort the list only if it has more than one element because a list of size 1 is already sorted. Also, if the list has only two nodes, we set current to NULL.) Consider the list shown in Figure 6.4.3.

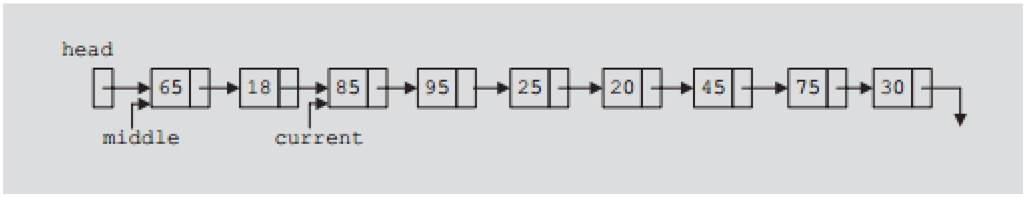

Figure 6.4.3 middle and current before traversing the list

Every time we advance middle by one node, we advance current by one node. After advancing current by one node, if current is not NULL, we again advance current by one node. That is, for the most part, every time middle advances by one node, current advances by two nodes. Eventually, current becomes NULL and middle points to the last node of the first sublist. For example, for the list in Figure 6.4.3, when current becomes NULL, middle points to the node with info 25 (see Figure 6.4.4).

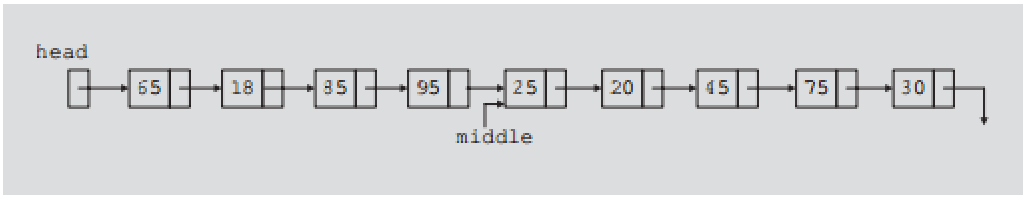

Figure 6.4.4 middle after traversing the list

It is now easy to divide the list into two sublists. First, using the link of middle, we assign a pointer to the node following middle. Then we set the link of middle to NULL. Figure 6.4.5 shows the resulting sublists.

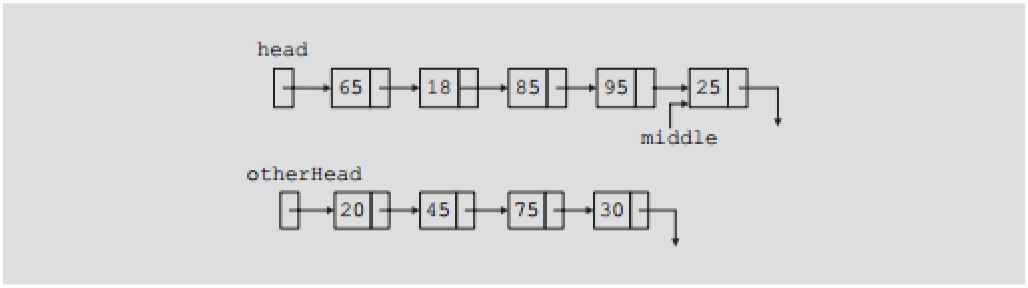

Figure 6.4.5 List after dividing it into two lists

This discussion translates into the following C++ function, divideList:

```
template <class Type>
void unorderedLinkedList<Type>::
                divideList(nodeType<Type>* first1,
                           nodeType<Type>* &first2)
```

```
{
    nodeType<Type>* middle;
    nodeType<Type>* current;

    if (first1 == NULL)   //list is empty
        first2 = NULL;
    else if (first1->link == NULL)  //list has only one node
        first2 = NULL;
    else
    {
        middle = first1;
        current = first1->link;

        if (current != NULL)    //list has more than two nodes
            current = current->link;
        while (current != NULL)
        {
            middle = middle->link;
            current = current->link;
            if (current != NULL)
                current = current->link;
        } //end while

        first2 = middle->link; //first2 points to the first
                               //node of the second sublist
        middle->link = NULL;   //set the link of the last node
                               //of the first sublist to NULL
    } //end else
} //end divideList
```

Now that we know how to divide a list into two sublists of nearly equal size, next we focus on merging the sorted sublists. Recall that, in mergesort, most of the sorting work is done in merging the sorted sublists.

Merge

Once the sublists are sorted, the next step in mergesort is to merge the sorted sublists. Sorted sublists are merged into a sorted list by comparing the elements of the sublists, and then adjusting the references of the nodes with the smaller info. Let us illustrate this procedure on the sublists shown in Figure 6.4.6. Suppose that first1 points to the first node of the first sublist, and first2 points to the first node of the second sublist.

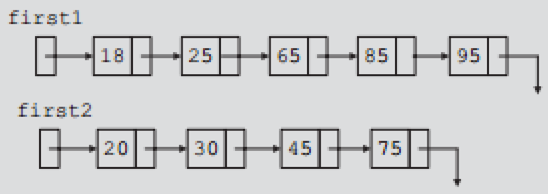

Figure 6.4.6 Sublists before merging

We first compare the info of the first node of each of the two sublists to determine the first node of the merged list. We set newHead to point to the first node of the merged list. We also use the pointer lastMerged to keep track of the last node of the merged list. The pointer of the first node of the sublist with the smaller node then advances to the next node of that sublist. Figure 10-38 shows the sublist of Figure 6.4.7 after setting newHead and lastMerged and advancing first1.

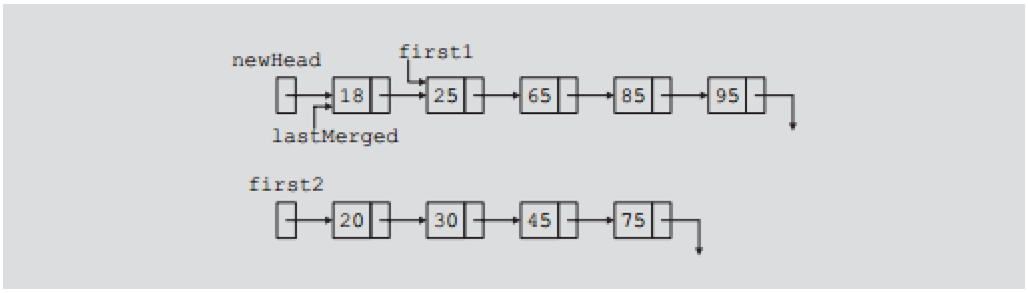

Figure 6.4.7 Sublists after setting newHead and lastMerged and advancing first1

In Figure 6.4.7, first1 points to the first node of the first sublist that is yet to be merged with the second sublist. So we again compare the nodes pointed to by first1 and first2, and adjust the link of the smaller node and the last node of the merged list so as to move the smaller node to the end of the merged list. For the sublists shown in Figure 6.4.7, after adjusting the necessary links, we have Figure 6.4.9.

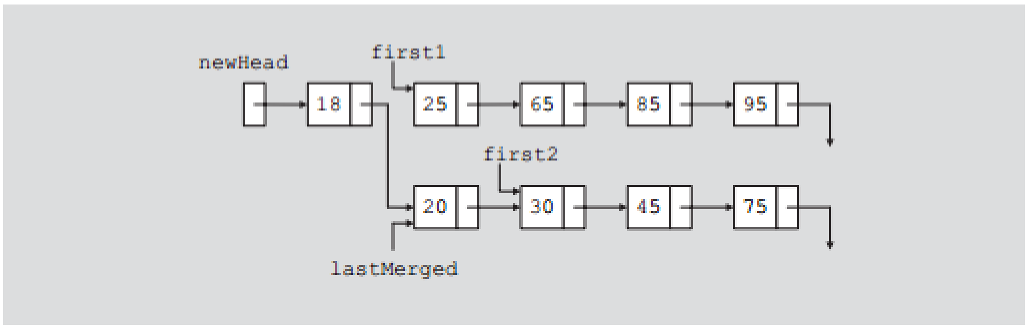

Figure 6.4.9 Merged list after putting the node with info 20 at the end of the merged list

We continue this process for the remaining elements of both sublists. Every time we move a node to the merged list, we advance either first1 or first2 to the next node. Eventually, either first1 or first2 becomes NULL. If first1 becomes NULL, the first sublist is exhausted first, so we attach the remaining nodes of the second sublist at the end of the partially merged list. If first2 becomes NULL, the second sublist is exhausted first, so we attach the remaining nodes of the first sublist at the end of the partially merged list.

Following this discussion, we can now write the C++ function, mergeList, to merge the two sorted sublists. The references (that is, addresses) of the first nodes of the sublists are passed as parameters to the function mergeList.

```
template <class Type>
nodeType<Type>* unorderedLinkedList<Type>::
                  mergeList(nodeType<Type>* first1,
                            nodeType<Type>* first2)
{
    nodeType<Type> *lastSmall; //pointer to the last node of
                               //the merged list
    nodeType<Type> *newHead;   //pointer to the merged list

    if (first1 == NULL)   //the first sublist is empty
        return first2;
    else if (first2 == NULL)   //the second sublist is empty
        return first1;
    else
    {
        if (first1->info < first2->info) //compare the first nodes
        {
            newHead = first1;
            first1 = first1->link;
            lastSmall = newHead;
        }
        else
        {
            newHead = first2;
            first2 = first2->link;
            lastSmall = newHead;
        }

    while (first1 != NULL && first2 != NULL)
    {
        if (first1->info < first2->info)
        {
            lastSmall->link = first1;
            lastSmall = lastSmall->link;
            first1 = first1->link;
        }
        else
        {
            lastSmall->link = first2;
            lastSmall = lastSmall->link;
            first2 = first2->link;
        }
    } //end while

    if (first1 == NULL) //first sublist is exhausted first
        lastSmall->link = first2;

        else               //second sublist is exhausted first
            lastSmall->link = first1;

        return newHead;
    }
}//end mergeList
```

Finally, we write the recursive mergesort function, recMergeSort, which uses the divideList and mergeList functions to sort a list. The reference of the first node of the list to be sorted is passed as a parameter to the function recMergeSort.

```
template <class Type>
void unorderedLinkedList<Type>::recMergeSort(nodeType<Type>* &head)
{
    nodeType<Type> *otherHead;

    if (head != NULL)  //if the list is not empty
        if (head->link != NULL) //if the list has more than one node
        {
            divideList(head, otherHead);
            recMergeSort(head);
            recMergeSort(otherHead);
            head = mergeList(head, otherHead);
        }
} //end recMergeSort
```

We can now give the definition of the function mergeSort, which should be included as a public member of the class unorderedLinkedList. (Note that the functions divideList, merge, and recMergeSort can be included as private members of the class unorderedLinkedList because these functions are used only to implement the function mergeSort.) The function mergeSort calls the function recMergeSort and passes first to this function. It also sets last to point to the last node of the list. The definition of the function mergeSort is as follows:

```
template<class Type>
void unorderedLinkedList<Type>::mergeSort()
{
    recMergeSort(first);

    if (first == NULL)
        last = NULL;
    else
    {
        last = first;
        while (last->link != NULL)
            last = last->link;
    }
} //end mergeSort
```