Data Structures Using C++ 2E

Chapter 10

Sorting Algorithms

Objectives

- Learn the various sorting algorithms
- Explore how to implement selection sort, insertion sort, Shellsort, quicksort, mergesort, and heapsort
- Discover how the sorting algorithms discussed in this chapter perform
- Learn how priority queues are implemented

Sorting Algorithms

- Several types in the literature
  - Discussion includes most common algorithms
- Analysis
  - Provides a comparison of algorithm performance
- Functions implementing sorting algorithms
  - Included as `public` members of related class

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

Selection Sort: Array-Based Lists

- List sorted by selecting elements in the list
  - Select elements one at a time
  - Move elements to their proper positions
- Selection sort operation
  - Find location of the smallest element in unsorted list portion
    - Move it to top of unsorted portion of the list
  - First time: locate smallest item in the entire list
  - Second time: locate smallest item in the list starting from the second element in the list, and so on

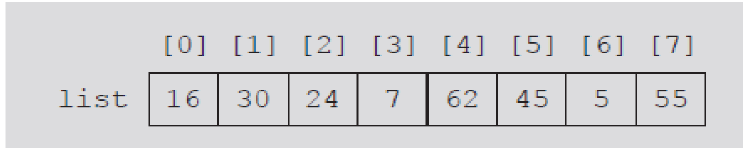

FIGURE 10-1 List of 8 elements

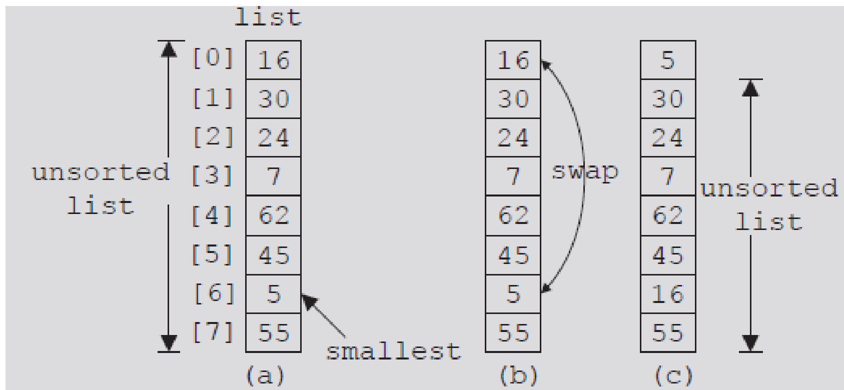

FIGURE 10-2 Elements of list during the first iteration

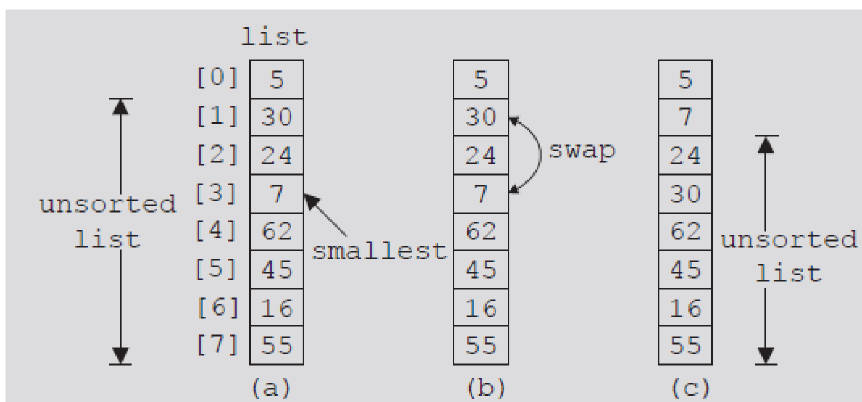

FIGURE 10-3 Elements of `list` during the second iteration

Selection Sort: Array-Based Lists (cont'd.)

- Selection sort steps
  - In the unsorted portion of the list
    - Find location of smallest element
    - Move smallest element to beginning of the unsorted list
- Keep track of unsorted list portion with a `for` loop

```
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].
}
```

Selection Sort: Array-Based Lists (cont'd.)

- Given: starting index, first, ending index, last
  - C++ function returns 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
```

Selection Sort: Array-Based Lists (cont'd.)

- Function `swap`
- Definition of function `selectionSort`

```
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

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);
    }
}
```

Selection Sort: Array-Based Lists (cont'd.)

- Add functions to implement selection sort in the definition of `class arrayListType`

```
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);
};
```

Analysis: Selection Sort

- Search algorithms
  - Concerned with number of key (item) comparisons
- Sorting algorithms
  - Concerned with number of key comparisons and number of data movements
- Analysis of selection sort
  - Function `swap`
    - Number of item assignments: $3(n-1)$
  - Function `minLocation`
    - Number of key comparisons of $O(n^2)$

Insertion Sort: Array-Based Lists

- Attempts to improve high selection sort key comparisons
- Sorts list by moving each element to its proper place
- Given list of length eight

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list	10	18	25	30	23	17	45	35

FIGURE 10-4 `list`

Insertion Sort: Array-Based Lists (cont'd.)

- Elements `list[0]`, `list[1]`, `list[2]`, `list[3]` in order
- Consider element `list[4]`
  - First element of unsorted list

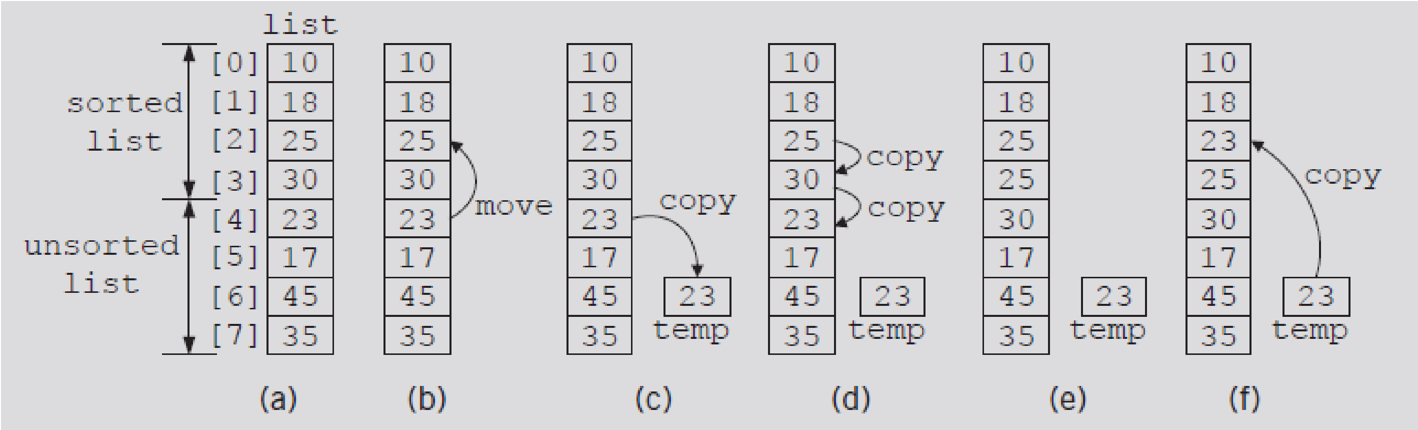

FIGURE 10-5 `list` elements while moving `list[4]` to its proper place

Insertion Sort: Array-Based Lists (cont'd.)

- Array containing list divided into two sublists
  - Upper and lower
- Index `firstOutOfOrder`
  - Points to first element in the lower sublist

```
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]
```

Insertion Sort: Array-Based Lists (cont'd.)

- `length` = 8
- Initialize `firstOutOfOrder` to one

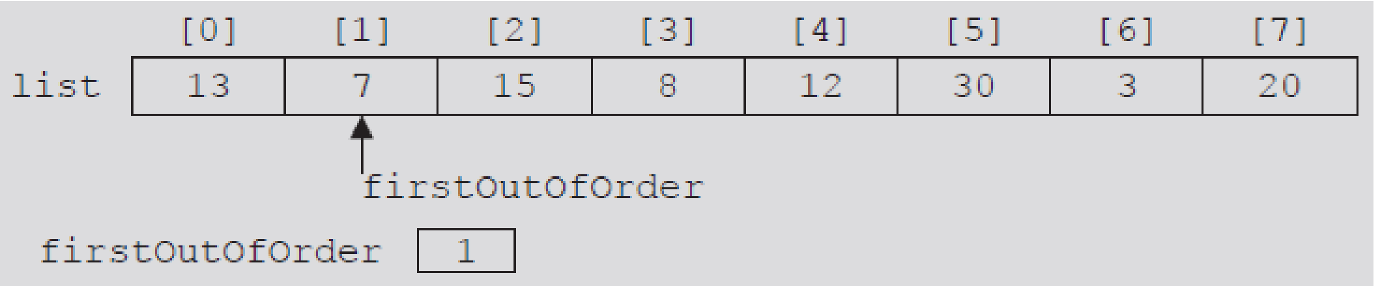

FIGURE 10-6 `firstOutOfOrder` = 1

Insertion Sort: Array-Based Lists (cont'd.)

- `list[firstOutOfOrder] = 7`
- `list[firstOutOfOrder - 1] = 13`
  - 7 < 13
- Expression in `if` statement evaluates to true
  - Execute body of `if` statement
    - `temp = list[firstOutOfOrder] = 7`
    - `location = firstOutOfOrder = 1`
  - Execute the `do...while` loop
    - `list[1] = list[0] = 13`
    - `location = 0`

Insertion Sort: Array-Based Lists (cont'd.)

- `do...while` loop terminates
  - Because `location` = 0
    - Copy `temp` into `list[location]` (`list[0]`)

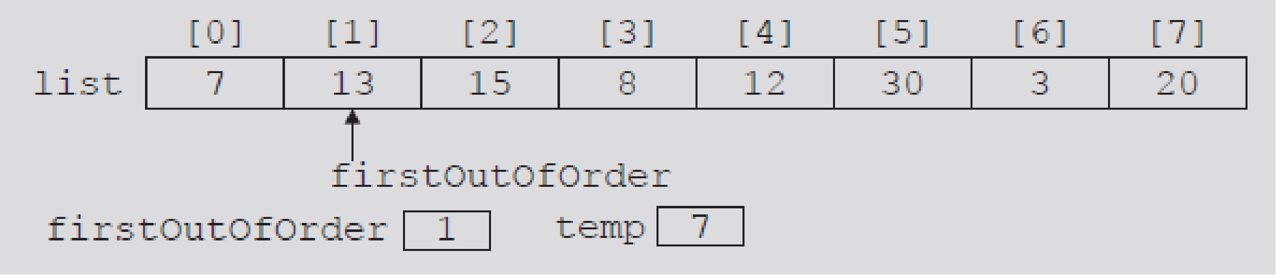

FIGURE 10-7 `list` after the first iteration of insertion sort

Insertion Sort: Array-Based Lists (cont'd.)

- Suppose `list` given in Figure 10-8(a)
  - Walk through code

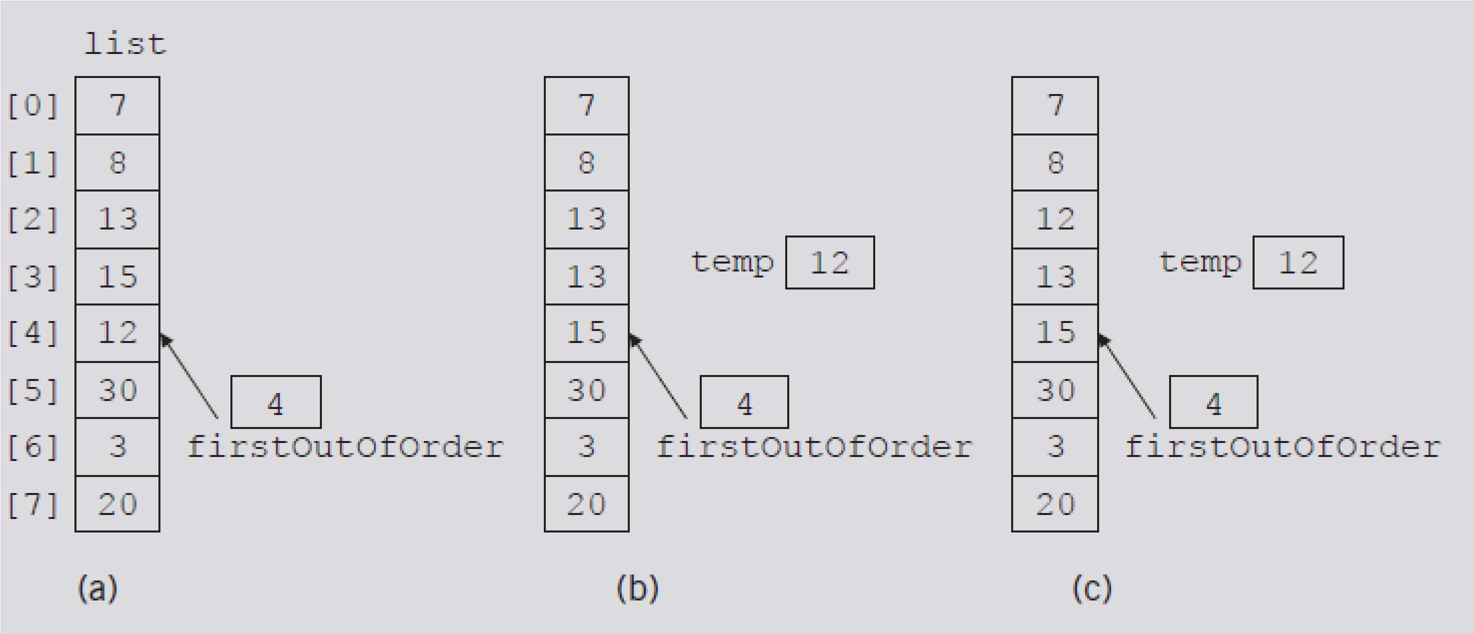

FIGURE 10-8 `list` elements while moving `list[4]` to its proper place

Insertion Sort: Array-Based Lists (cont'd.)

- Suppose `list` given in Figure 10-9
  - Walk through code

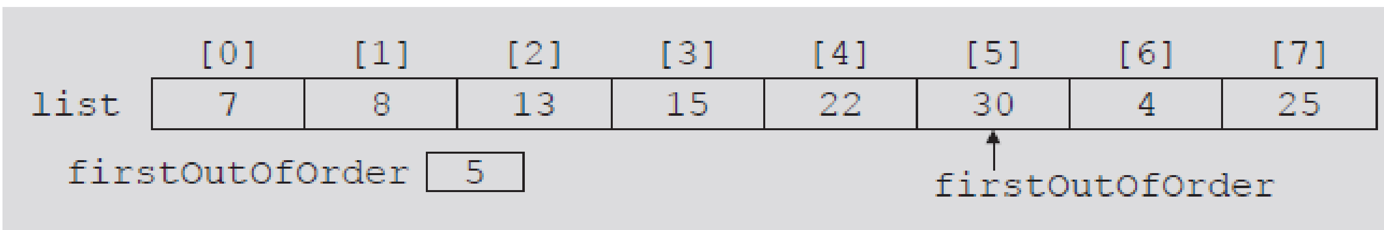

FIGURE 10-9 First out-of-order element is at position 5

Insertion Sort: Array-Based Lists (cont'd.)

- C++ function implementing 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

- If list stored in an array
  - Traverse list in either direction using index variable
- If list stored in a linked list
  - Traverse list in only one direction
    - Starting at first node: links only in one direction

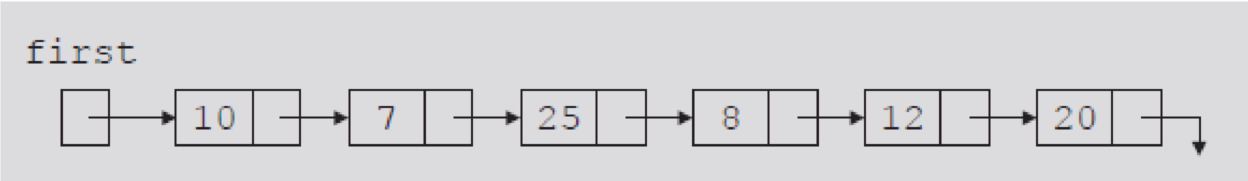

FIGURE 10-10 Linked list

Insertion Sort: Linked List-Based Lists (cont'd.)

- `firstOutOfOrder`
  - Pointer to node to be moved to its proper location
- `lastInOrder`
  - Pointer to last node of the sorted portion of the list

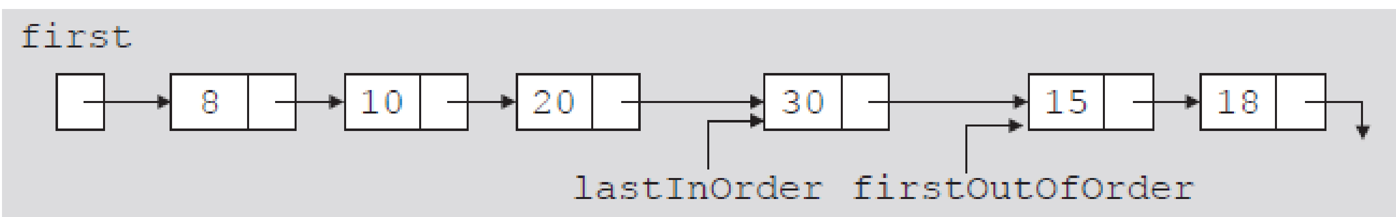

FIGURE 10-11 Linked list and pointers `lastInOrder` and `firstOutOfOrder`

Insertion Sort: Linked List-Based Lists (cont'd.)

- Compare `firstOutOfOrder` info with `first` node info
  - If `firstOutOfOrder` info smaller than `first` node info
    - `firstOutOfOrder` moved before `first` node
  - Otherwise, search list starting at second node to find location where to move `firstOutOfOrder`
- Search list using two pointers
  - `current`
  - `trailCurrent`: points to node just before `current`
- Handle any special cases

Insertion Sort: Linked List-Based Lists (cont'd.)

```
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;
}
```

Insertion Sort: Linked List-Based Lists (cont'd.)

- Case 1
  - `firstOutOfOrder`->info less than `first`->info
    - Node `firstOutOfOrder` moved before `first`
    - Adjust necessary links

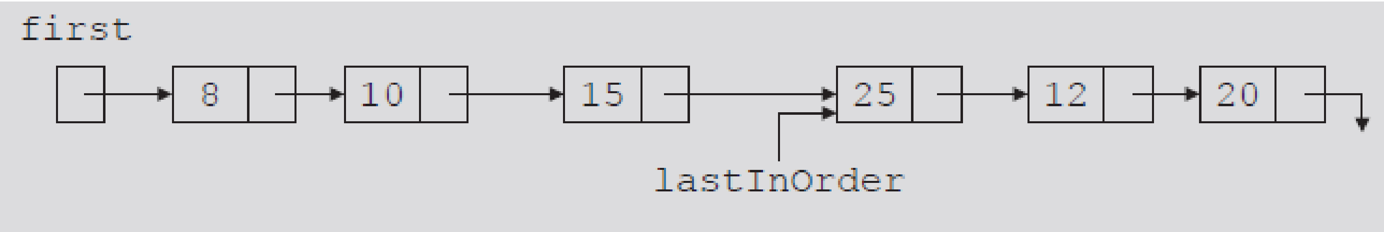

FIGURE 10-13 Linked list after moving the node with info 8 to the beginning

Insertion Sort: Linked List-Based Lists (cont'd.)

- Review Case 2 on page 546
- Review Case 3 on page 546
- Review function `linkedInsertionSort` on page 547
  - Implements previous algorithm

Analysis: Insertion Sort

TABLE 10-1 Average-case behavior of the selection sort and insertion sort for a list of length *n*

Algorithm	Number of comparisons	Number of swaps/item assignments
Selection sort	$(1/2)n(n-1) = O(n^2)$	$3(n-1) = O(n)$
Insertion sort	$(1/4)n^2 + O(n) = O(n^2)$	$(1/4)n^2 + O(n) = O(n^2)$

Shellsort

- Reduces number of item movements in insertion sort by modifying it
  - Introduced in 1959 by D.E. Shell
  - Also known as diminishing-increment sort
- List elements viewed as sublists at a particular distance
  - Each sublist sorted
    - Elements far apart move closer to their final position

Shellsort (cont'd.)

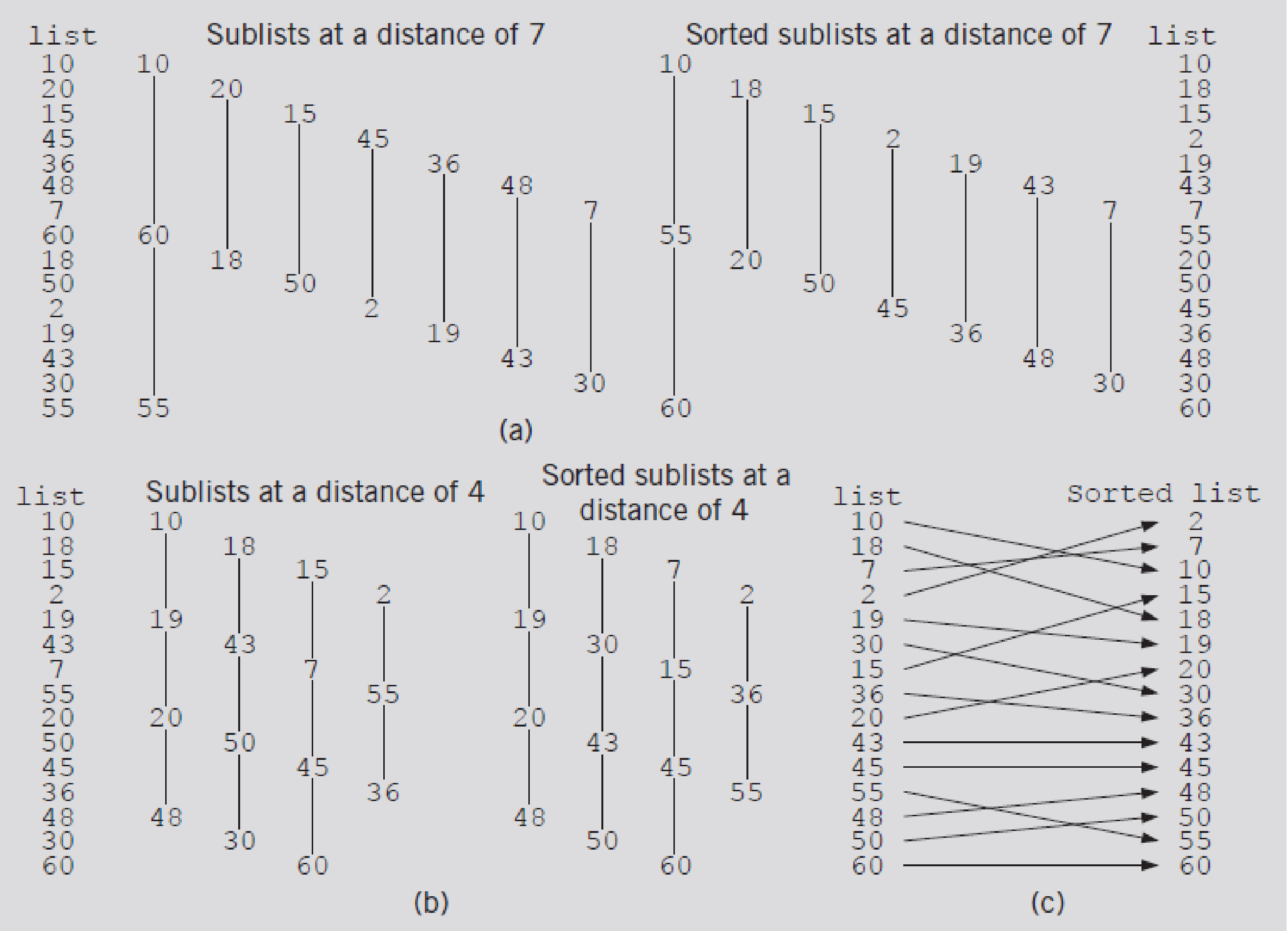

FIGURE 10-19 Lists during Shellsort

Shellsort (cont'd.)

- Figure 10-19
  - Sort elements at a distance of 7, 4, 1
    - Called increment sequence
- Desirable to use as few increments as possible
- D.E. Knuth recommended increment sequence
  - 1, 4, 13, 40, 121, 364, 1093, 3280. . . .
    - Ratio between successive increments: about one-third
    - i^{th} increment = 3 • (i – 1)*th* increment + 1
- Certain increment sequences must be avoided
  - 1, 2, 4, 8, 16, 32, 64, 128, 256. . . .
    - Bad performance

Shellsort (cont'd.)

- Function implementing Shellsort algorithm

```
template <class elemType>
void arrayListType<elemType>::shellSort()
{
    int inc;

    for (inc = 1; inc < (length - 1) / 9; inc = 3 * inc + 1);

    do
    {
        for (int begin = 0; begin < inc; begin++)
            intervalInsertionSort(begin, inc);

        inc = inc / 3;
    }
    while (inc > 0);
} //end shellSort
```

Shellsort (cont'd.)

- Function `shellSort`
  - Uses function `intervalInsertionSort`
    - Modified version of insertion sort for array-based lists
- `intervalInsertionSort`
  - Sublist starts at variable `begin`
  - Increment between successive elements given by variable `inc` instead of one
- Analysis of Shellsort
  - Difficult to obtain

Lower Bound on Comparison-Based Sort Algorithms

- Comparison tree
  - Graph tracing comparison-based algorithm execution
    - Node: comparison drawn as a circle
    - Leaf: rectangle representing final node ordering
    - Root node: top node in the figure
    - Branch: straight line connecting two nodes
    - Path: sequence of branches between nodes

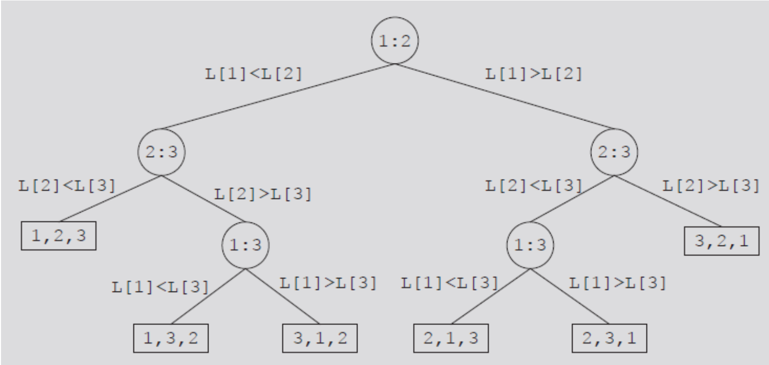

FIGURE 10-20 Comparison tree for sorting three items

Lower Bound on Comparison-Based Sort Algorithms (cont'd.)

- Unique permutation of the elements
  - Associated with each path from the root to a leaf
- Theorem
  - Let L be a list of n distinct elements. Any sorting algorithm that sorts L by comparison of the keys only, in its worst case, makes at least $O(n\log_2 n)$ key comparisons

Quicksort: Array-Based Lists

- Uses the divide-and-conquer technique to sort a list
  - List partitioned into two sublists
    - Two sublists sorted and combined into one list
    - Combined list then sorted using quicksort (recursion)
- Trivial to combine sorted `lowerSublist` and `upperSublist`
- All sorting work done in partitioning the list

```
if (the list size is greater than 1)
{
   a. Partition the list into two sublists, say lowerSublist and upperSublist.
   b. Quicksort lowerSublist.
   c. Quicksort upperSublist.
   d. Combine the sorted lowerSublist and sorted upperSublist.
}
```

Quicksort: Array-Based Lists (cont'd.)

- Pivot divides list into two sublists
  - `lowerSublist`: elements smaller than pivot
  - `upperSublist`: elements greater than pivot
- Choosing the pivot
  - `lowerSublist` and `upperSublist` nearly equal

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
list	45	82	25	94	50	60	78	32	92

FIGURE 10-21 List before the partition

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
list	32	25	45	50	82	60	78	94	92
	lowerSublist				upperSublist				

FIGURE 10-22 List after the partition

Quicksort: Array-Based Lists (cont'd.)

- Partition algorithm
  - Determine pivot; swap pivot with first list element
    - Suppose index `smallIndex` points to last element smaller than pivot. `smallIndex` initialized to first list element
  - For the remaining list elements (starting at second element): If current element smaller than pivot
    - Increment `smallIndex`
    - Swap current element with array element pointed to by `smallIndex`
  - Swap first element (pivot) with array element pointed to by `smallIndex`

Quicksort: Array-Based Lists (cont'd.)

- Function `partition`
  - Passes starting and ending list indices
  - Swaps certain elements 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;
}
```

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

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

Quicksort: Array-Based Lists (cont'd.)

- Given starting and ending list indices
  - Function `recQuickSort` implements the recursive version of quicksort
- Function `quickSort` calls `recQuickSort`

```
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);
    }
}
```

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

Analysis: Quicksort

	Number of comparisons	Number of swaps
Average case	$1.39n\log_2 n + O(n) = O(n\log_2 n)$	$0.69n\log_2 n + O(n) = O(n\log_2 n)$
Worst case	$(1/2)(n^2 - n) = O(n^2)$	$(1/2)n^2 + (3/2)n - 2 = O(n^2)$

TABLE 10-2 Analysis of quicksort for a list of length *n*

Mergesort: Linked List-Based Lists

- Quicksort
  - Average-case behavior: $O(n\log_2 n)$
  - Worst-case behavior: $O(n^2)$
- Mergesort behavior: always $O(n\log_2 n)$
  - Uses divide-and-conquer technique to sort a list
  - Partitions list into two sublists
    - Sorts sublists
    - Combines sorted sublists into one sorted list
- Difference between mergesort and quicksort
  - How list is partitioned

Mergesort: Linked List-Based Lists (cont'd.)

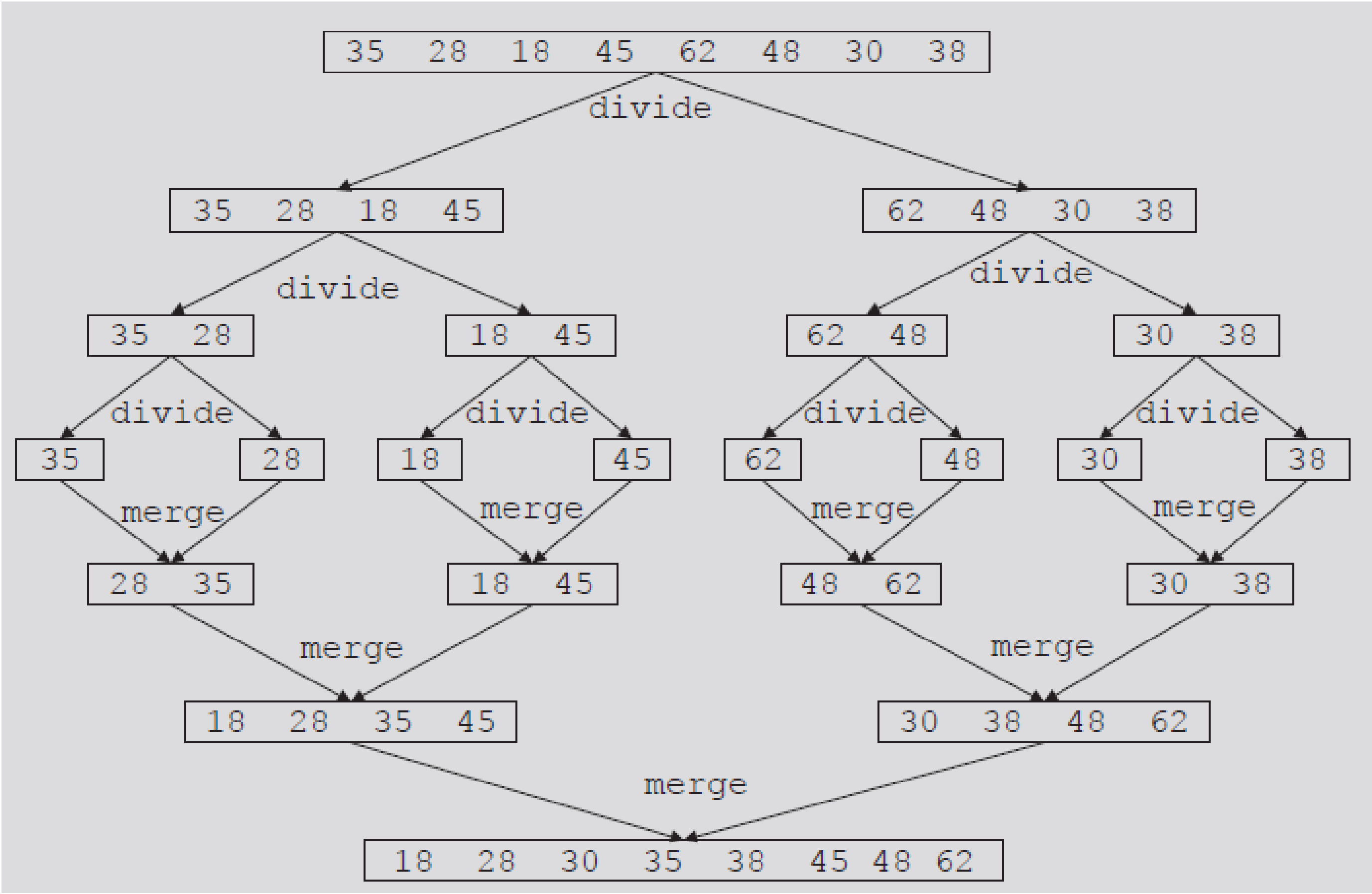

FIGURE 10-32 Mergesort algorithm

Mergesort: Linked List-Based Lists (cont'd.)

- Most sorting work done in merging sorted sublists
- General algorithm for mergesort

```
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.
}
```

Divide

- To divide list into two sublists
  - Need to find middle node
  - Use two pointers: `middle` and `current`
    - Advance middle by one node, advance `current` by one node
    - `current` becomes NULL; `middle` points to last node
  - Divide list into two sublists
    - Using the link of `middle`: assign pointer to node following `middle`
    - Set link of `middle` to `NULL`
- See function `divideList` on page 561

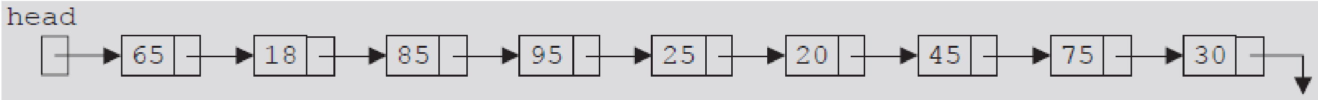

FIGURE 10-33 Unsorted linked list

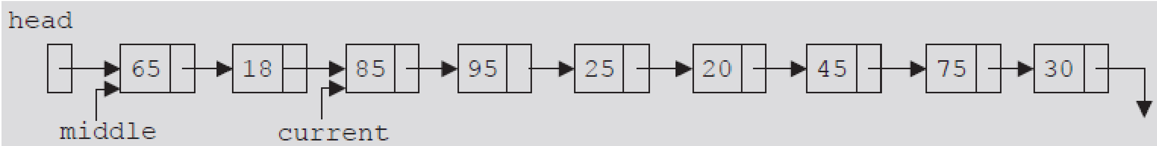

FIGURE 10-34 `middle` and `current` before traversing the list

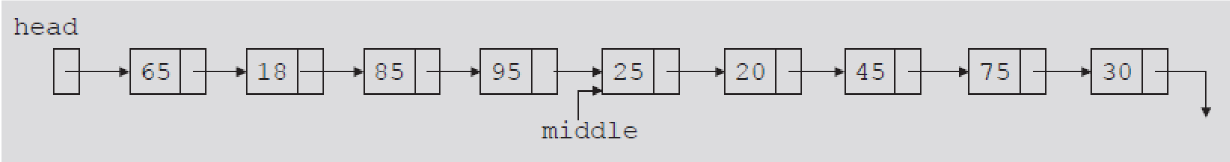

FIGURE 10-35 `middle` after traversing the list

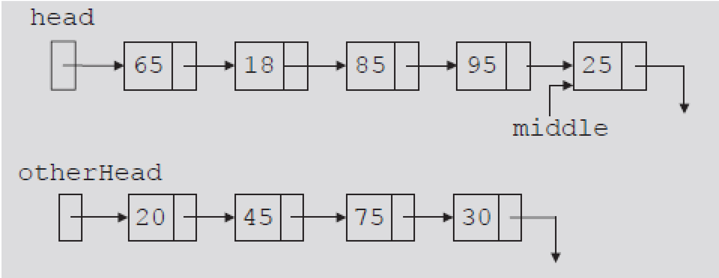

FIGURE 10-36 List after dividing it into two lists

Merge

- Once sublists sorted
  - Next step: merge the sorted sublists
- Merge process
  - Compare elements of the sublists
  - Adjust references of nodes with smaller info
- See code on page 564 and 565

Analysis: Mergesort

- Maximum number of comparisons made by mergesort: $O(n \log_2 n)$
- If $W(n)$ denotes number of key comparisons
  – Worst case to sort L: $W(n) = O(n \log_2 n)$
- Let $A(n)$ denote number of key comparisons in the average case
  – Average number of comparisons for mergesort
  – If n is a power of 2
    - $A(n) = n \log_2 n - 1.25n = O(n \log_2 n)$

Heapsort: Array-Based Lists

- Overcomes quicksort worst case
- Heap: list in which each element contains a key
  - Key in the element at position *k* in the list
    - At least as large as the key in the element at position $2k + 1$ (if it exists) and $2k + 2$ (if it exists)
- C++ array index starts at zero
  - Element at position *k*
    - $k + 1th$ element of the list

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
85	70	80	50	40	75	30	20	10	35	15	62	58

FIGURE 10-41 A heap

Heapsort: Array-Based Lists (cont'd.)

- Data given in Figure 10-41
  - Can be viewed in a complete binary tree
- Heapsort
  - First step: convert list into a heap
    - Called `buildHeap`
  - After converting the array into a heap
    - Sorting phase begins

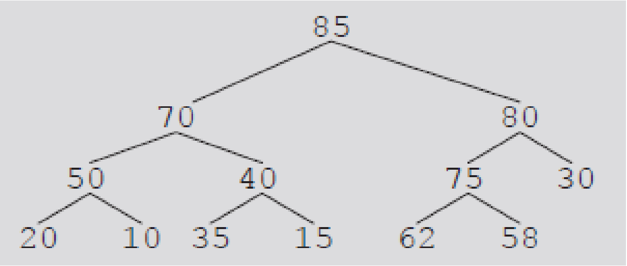

FIGURE 10-42 Complete binary tree corresponding to the list in Figure 10-41

Build Heap

```
int largeIndex = 2 * low + 1;   //index of the left child

while (largeIndex <= high)
{
   if ( largeIndex < high)
       if (list[largeIndex] < list[largeIndex + 1])
          largeIndex = largeIndex + 1;  //index of the larger child
   if (list[low] > list[largeIndex])   //the subtree is already in
                                       //a heap
       break;
   else
   {
       swap(list[low], list[largeIndex]);   //Line swap**
       low = largeIndex;  //go to the subtree to further
                          //restore the heap
       largeIndex = 2 * low + 1;
   } //end else
}//end while
```

Build Heap (cont'd.)

- Function `heapify`
  - Restores the heap in a subtree
  - Implements the `buildHeap` function
    - Converts list into a heap

```
template<class elemType>
void arrayListType<elemType>::heapify(int low, int high)
{
   int largeIndex;

   elemType temp = list[low]; //copy the root node of the subtree

   largeIndex = 2 * low + 1;  //index of the left child

   while (largeIndex <= high)
   {
       if (largeIndex < high)
          if (list[largeIndex] < list[largeIndex + 1])
             largeIndex = largeIndex + 1; //index of the largest
                                          //child

       if (temp > list[largeIndex]) //subtree is already in a heap
          break;
       else
       {
          list[low] = list[largeIndex]; //move the larger child
                                        //to the root
          low = largeIndex; //go to the subtree to restore the heap
          largeIndex = 2 * low + 1;
       }
   }//end while


   list[low] = temp; //insert temp into the tree, that is, list

} //end heapify
```

Build Heap (cont'd.)

```
template <class elemType>
void arrayListType<elemType>::buildHeap()
{
   for (int index = length / 2 - 1; index >= 0; index--)
       heapify(index, length - 1);
}
```

Build Heap (cont'd.)

- The heapsort algorithm

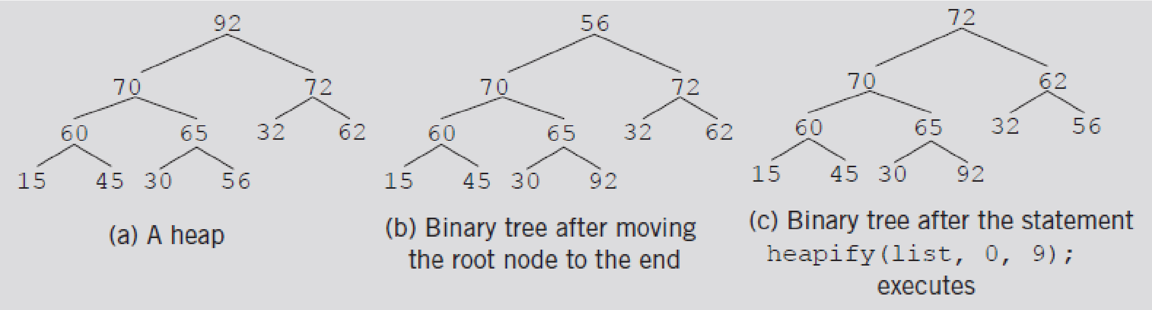

FIGURE 10-48 Heapsort

```
template <class elemType>
void arrayListType<elemType>::heapSort()
{
    elemType temp;

    buildHeap();

    for (int lastOutOfOrder = length - 1; lastOutOfOrder >= 0;
                                          lastOutOfOrder--)
    {
        temp = list[lastOutOfOrder];
        list[lastOutOfOrder] = list[0];
        list[0] = temp;
        heapify(0, lastOutOfOrder - 1);
    }//end for
}//end heapSort
```

Analysis: Heapsort

- *Given L a* list of *n* elements where $n > 0$
- Worst case
  - Number of key comparisons to sort *L*
    - $2n\log_2 n + O(n)$
  - Number of item assignments to sort *L*
    - $n\log_2 n + O(n)$
- Average number of comparisons to sort *L*
  - $O(n\log_2 n)$
- Heapsort takes twice as long as quicksort
  - Avoids the slight possibility of poor performance

Priority Queues (Revisited)

- Customers or jobs with higher priorities
  - Pushed to front of the queue
- Assume priority of the queue elements is assigned using the relational operators
  - In a heap, largest list element is always the first element of the list
  - After removing largest list element
    - Function `heapify` restores the heap in the list
  - Implement priority queues as heaps
    - To ensure largest element of the priority queue is always the first element of the queue

Priority Queues (Revisited) (cont'd.)

- Insert an element in the priority queue
  - Insert new element in first available list position
    - Ensures array holding the list is a complete binary tree
  - After inserting new element in the heap, the list might no longer be a heap
    - Restore the heap (might result in moving the new entry to the root node)

```
while (the parent of the new entry is smaller than the new entry)
    swap the parent with the new entry.
```

Priority Queues (Revisited) (cont'd.)

- Remove an element from the priority queue
  - Assume priority queue implemented as a heap
    - Copy last element of the list into first array position
    - Reduce list length by one
    - Restore heap in the list
  - Other operations for priority queues
    - Can be implemented in the same way as implemented for queues

Summary

- Search algorithms may require sorted data
- Several sorting algorithms available
  - Selection sort, insertion sort, Shellsort, quicksort, mergesort, and heapsort
    - Can be applied to either array-based lists or linked lists
- Compare algorithm performance through analysis
  - Number of key comparisons
  - Number of data movements
- Functions implementing sorting algorithms
  - Included as `public` members of the related class