

Chapter 10: APPLICATIONS OF ARRAYS (SEARCHING AND SORTING)

In this chapter you will:

- ❖ Explore how to sort an array using the
 - ❖ bubble sort algorithm
 - ❖ selection sort algorithm
- ❖ Learn how to implement the binary search algorithm

List Processing

A **list** is a collection of values of the same type. Because all values are of the same type, a convenient place to store a list is in an array and, particularly, in a one-dimensional array. The size of a list is the number of elements in the list. Because the size of a list can increase and decrease, the array you use to store the list should be declared as the maximum size of the list.

Basic operations performed on a list include the following:

- Search the list for a given item.
- Sort the list.
- Insert an item in the list.
- Delete an item from the list.
- Print the list.

The following sections discuss searching and sorting algorithms.

Bubble Sort

There are many sorting algorithms. This section describes the sorting algorithm, called **bubble sort**, to sort a list.

Suppose `list[0]...list[n - 1]` is a list of n elements, indexed 0 to $n - 1$. We want to rearrange, that is, sort, the elements of `list` in increasing order. The bubble sort algorithm works as follows:

In a series of $n - 1$ iterations, the successive elements `list[index]` and `list[index + 1]` of `list` are compared. If `list[index]` is greater than `list[index + 1]`, then the elements `list[index]` and `list[index + 1]` are swapped, that is, interchanged.

It follows that the smaller elements move toward the top (beginning), and the larger elements move toward the bottom (end) of the list.

In the first iteration, we consider `list[0]...list[n - 1]`; in the second iteration, we consider `list[0]...list[n - 2]`; in the third iteration, we consider `list[0]...list[n - 3]`, and so on. For example, consider `list[0]...list[4]`, as shown in Figure 10-1.

In the first iteration, we consider `list[0]...list[n - 1]`; in the second iteration, we consider `list[0]...list[n - 2]`; in the third iteration, we consider `list[0]...list[n - 3]`, and so on. For example, consider `list[0]...list[4]`, as shown in Figure 10-1.

list	
<code>list[0]</code>	10
<code>list[1]</code>	7
<code>list[2]</code>	19
<code>list[3]</code>	5
<code>list[4]</code>	16

FIGURE 10-1 List of five elements

Iteration 1: Sort `list[0]...list[4]`. Figure 10-2 shows how the elements of `list` get rearranged in the first iteration.

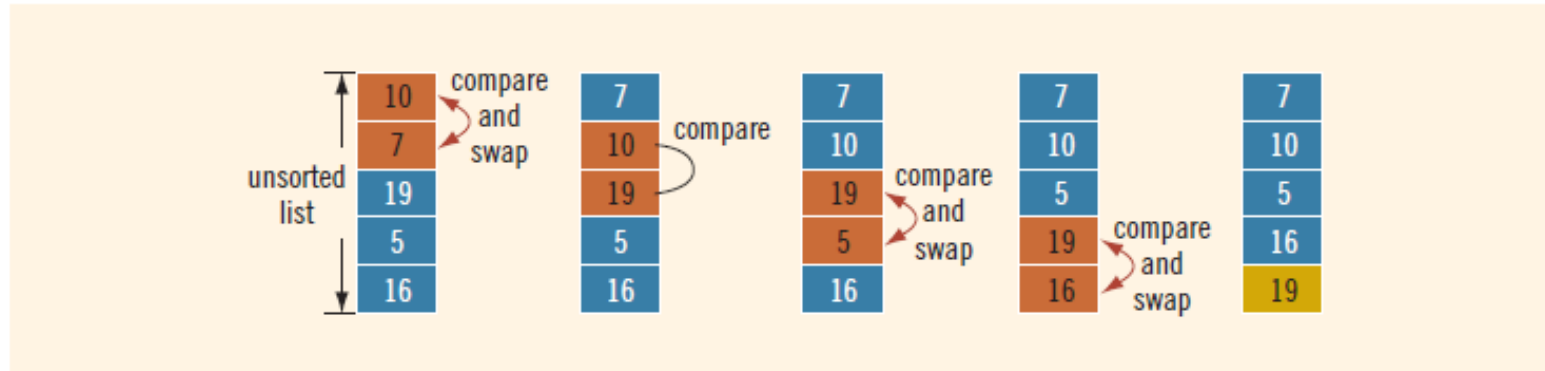


FIGURE 10-2 Elements of `list` during the first iteration

Notice that in the first diagram of Figure 10-2, `list[0] > list[1]`. Therefore, `list[0]` and `list[1]` are swapped. In the second diagram, `list[1]` and `list[2]` are compared. Because `list[1] < list[2]`, they do not get swapped. The third diagram of Figure 10-2 compares `list[2]` with `list[3]`; because `list[2] > list[3]`, `list[2]` is swapped with `list[3]`. Then, in the fourth diagram, we compare `list[3]` with `list[4]`. Because `list[3] > list[4]`, `list[3]` and `list[4]` are swapped.

After the first iteration, the largest element is at the last position. Therefore, in the next iteration, we consider `list[0...3]`.

After the first iteration, the largest element is at the last position. Therefore, in the next iteration, we consider `list[0...3]`.

Iteration 2: Sort `list[0...3]`. Figure 10-3 shows how the elements of `list` get rearranged in the second iteration.

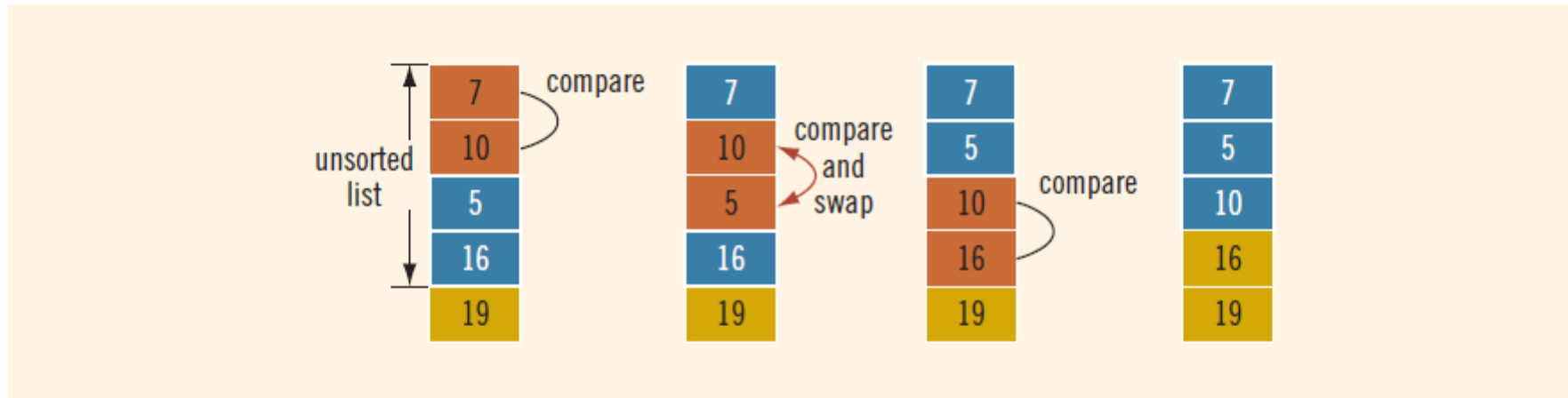


FIGURE 10-3 Elements of `list` during the second iteration

The elements are compared and swapped as in the first iteration. Here, only the list elements `list[0]` through `list[3]` are considered. After the second iteration, the last two elements are in the right place. Therefore, in the next iteration, we consider `list[0...2]`.

Iteration 3: Sort `list[0...2]`. Figure 10-4 shows how the elements of `list` get rearranged in the third iteration.

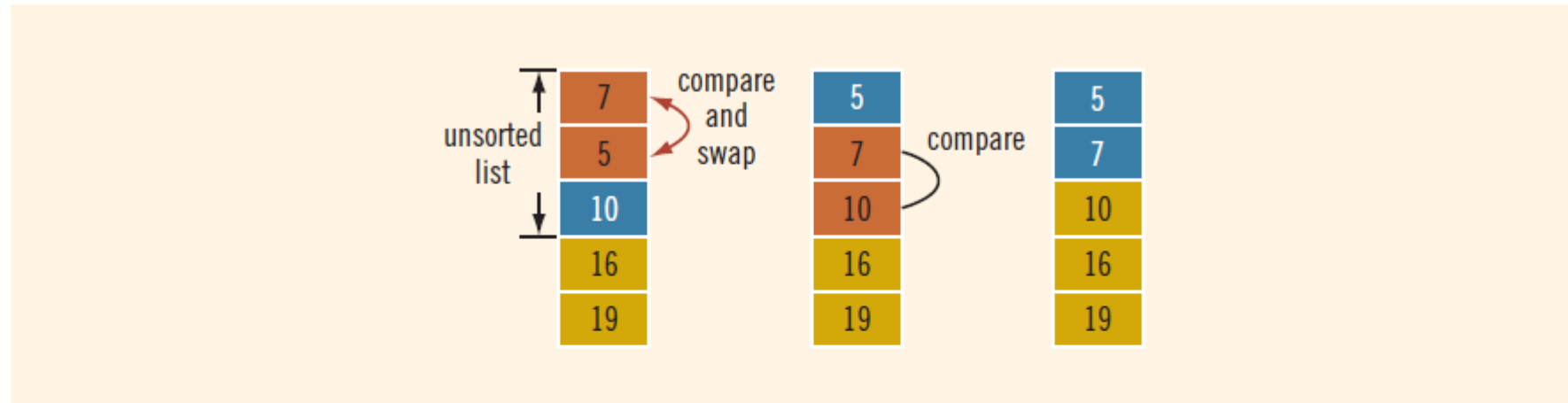


FIGURE 10-4 Elements of `list` during the third iteration

After the third iteration, the last three elements are in the right place. Therefore, in the next iteration, we consider `list[0...1]`.

Iteration 4: Sort `list[0...1]`. Figure 10-5 shows how the elements of `list` get rearranged in the fourth iteration.

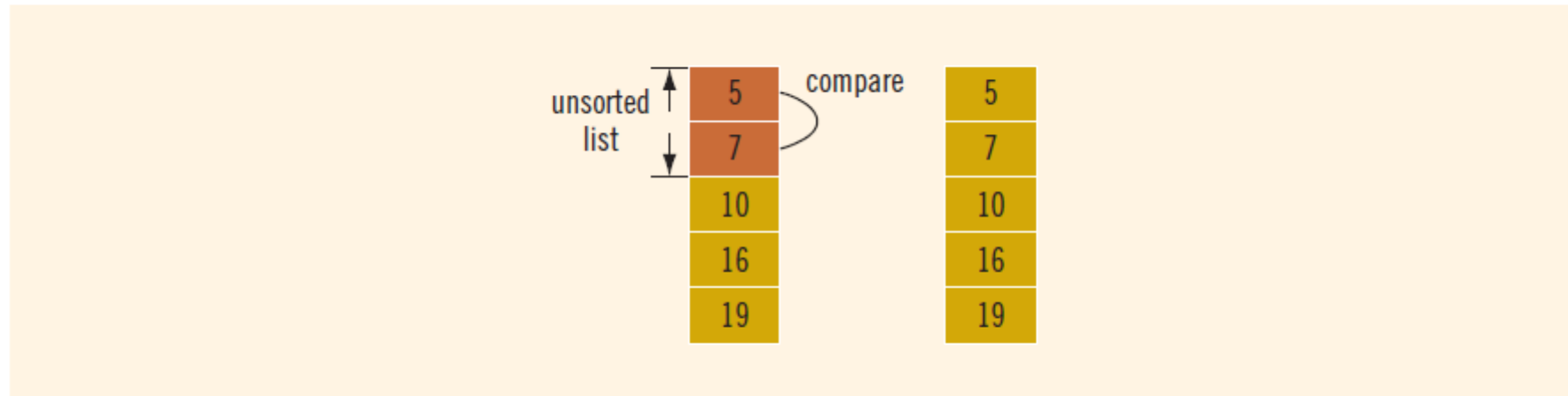


FIGURE 10-5 Elements of `list` during the fourth iteration

After the fourth iteration, `list` is sorted.

The following function implements the bubble sort algorithm:

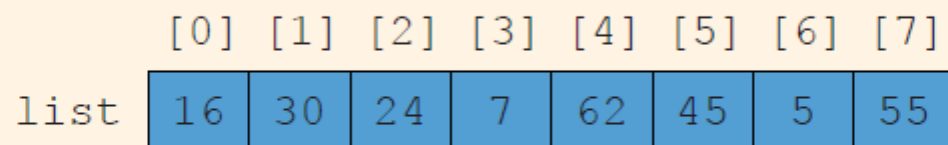
```
void bubbleSort(int list[], int length)
{
    int temp;
    int iteration;
    int index;
    for (iteration = 1; iteration < length; iteration++)
    {
        for (index = 0; index < length - iteration; index++)
            if (list[index] > list[index + 1])
            {
                temp = list[index];
                list[index] = list[index + 1];
                list[index + 1] = temp;
            }
    }
}
```

Selection Sort

This section describes another sorting algorithm called the **selection sort**.

As the name implies, in **selection sort** algorithm, we rearrange the list by selecting an element in the list and moving it to its proper position. 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 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.

Suppose you have the list shown in Figure 10-6.



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list	16	30	24	7	62	45	5	55

FIGURE 10-6 List of eight elements

Figure 10-7 shows the elements of `list` in the first iteration.

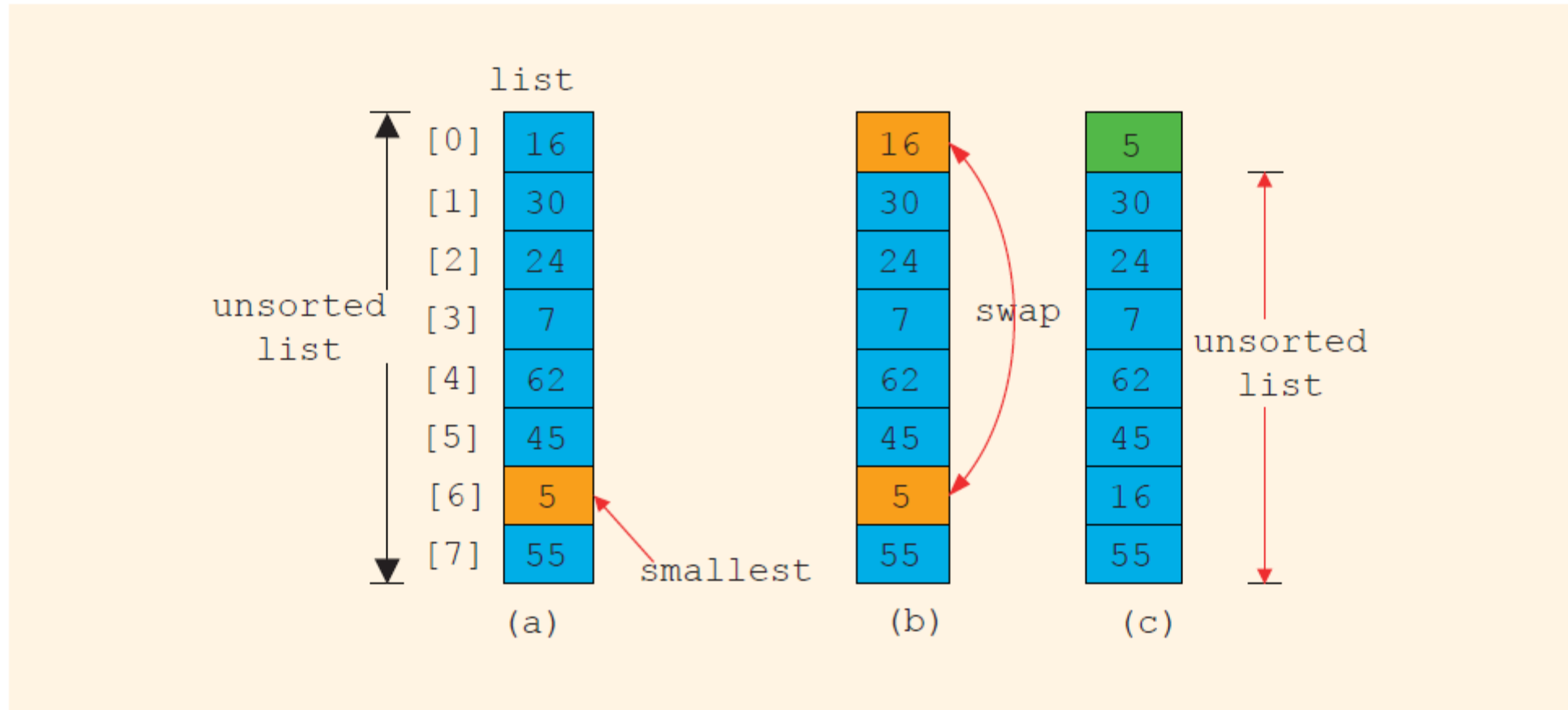


FIGURE 10-7 Elements of `list` during the first iteration

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 10-7(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 10-7(b). After swapping these elements, the resulting list is as shown in Figure 10-7(c).

Figure 10-8 shows the elements of `list` during the second iteration.

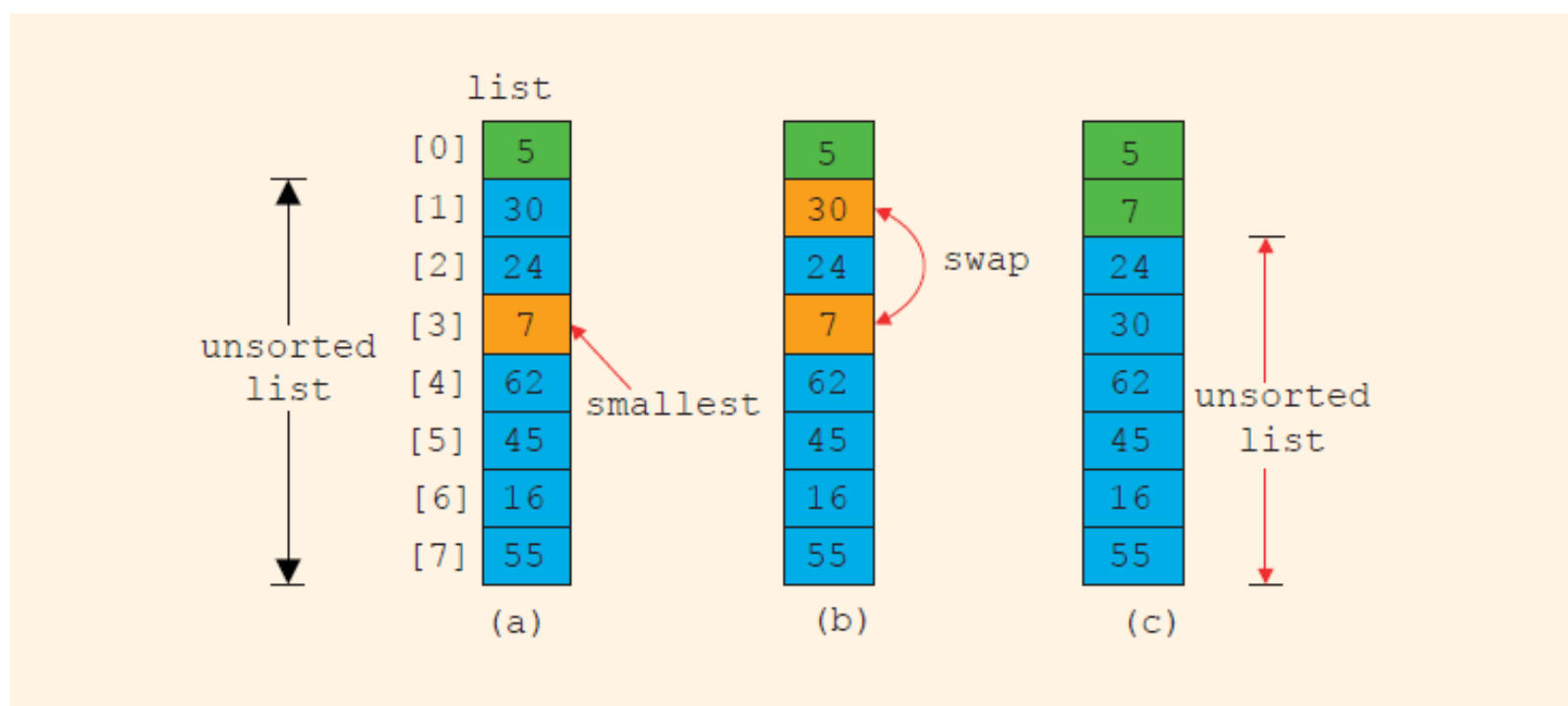


FIGURE 10-8 Elements of `list` during the second iteration

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 10-8(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 10-8(b). After swapping `list[1]` with `list[3]`, the resulting list is as shown in Figure 10-8(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:

- a. Find the location of the smallest element.
- b. Move the smallest element to the beginning of the unsorted list.

Initially, the entire list (that is, `list[0]...list[length - 1]`) is the unsorted list. After executing Steps a and b once, the unsorted list is `list[1]...list[length - 1]`. After executing Steps a and b a second time, the unsorted list is `list[2]...list[length - 1]`, and so on. In this way, 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 shown in the following pseudocode:

```
for (index = 0; index < length - 1; index++)
{
    a. Find the location, smallestIndex, of the smallest element in
       list[index]...list[length - 1].
    b. 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 the smallest element with `list[0]`. The second time through the loop, we locate the smallest element in `list[1]...list[length - 1]` and swap the smallest element with `list[1]`, and so on.

Step a is similar to the algorithm for finding the index of the largest item in the list, as discussed in Chapter 9. (Also see Chapter 9, Programming Exercise 2.) Here, we find the index of the smallest item in the list. The general form of Step a is:

[illegible]

Step b swaps the contents of `list[smallestIndex]` with `list[index]`. The following statements accomplish this task:

```
temp = list[smallestIndex];  
list[smallestIndex] = list[index];  
list[index] = temp;
```

It follows that to swap the values, three item assignments are needed. The following function, `selectionSort`, implements the selection sort algorithm:

```
void selectionSort(int list[], int length)
{
    int index;
    int smallestIndex;
    int location;
    int temp;

    for (index = 0; index < length - 1; index++)
    {
        //Step a
        smallestIndex = index;

        for (location = index + 1; location < length; location++)
            if (list[location] < list[smallestIndex])
                smallestIndex = location;

        //Step b
        temp = list[smallestIndex];
        list[smallestIndex] = list[index];
        list[index] = temp;
    }
}
```


Binary Search

A sequential search is not very efficient for large lists. It typically searches about half of the list. However, if the list is sorted, you can use another search algorithm called **binary search**. A binary search is much faster than a sequential search. In order to apply a binary search, *the list must be sorted*.

A binary search uses the “divide and conquer” technique to search the list. First, the search item is compared with the middle element of the list. If the search item is less than the middle element of the list, we restrict the search to the upper half of the list; otherwise, we search the lower half of the list.

Consider the following sorted list of `length = 12`, as shown in Figure 10-16.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

FIGURE 10-16 List of length 12

Suppose that we want to determine whether 75 is in the list. Initially, the entire list is the search list (see Figure 10-17).

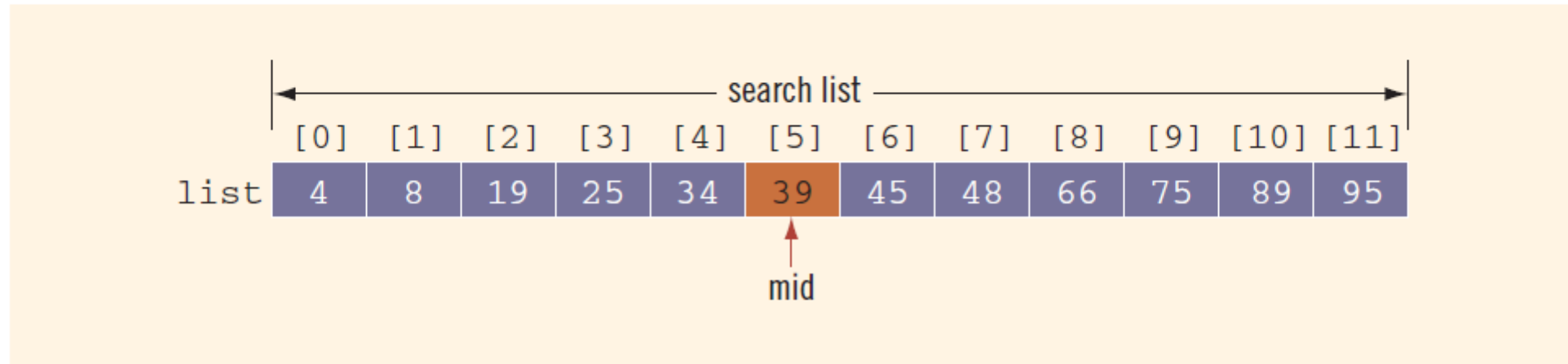


FIGURE 10-17 Search list, `list[0]...list[11]`

First, we compare 75 with the middle element, `list[5]` (which is 39), in the list. Since $75 \neq \text{list}[5]$ and $75 > \text{list}[5]$, next we restrict our search to `list[6]...list[11]`, as shown in Figure 10-18.

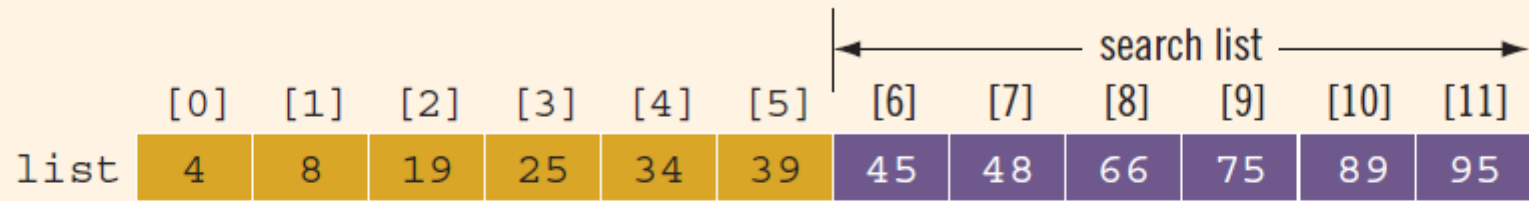


FIGURE 10-18 Search list, `list[6]...list[11]`

The above process is now repeated on `list[6]...list[11]`, which is a list of `length = 6`.

Because we frequently need to determine the middle element of the list, the binary search algorithm is usually implemented for array-based lists. To determine the middle element of the list, we add the starting index, `first`, and the ending index, `last`, of the search list and divide by 2 to calculate its index. That is, $mid = \frac{first + last}{2}$. Initially, `first = 0` and (since array index in C++ starts at 0 and `listLength` denotes the number of elements in the list) `last = listLength - 1`. (Note that the formula for calculating the middle element works regardless of whether the list has an even or odd number of elements.)

The following C++ function implements the binary search algorithm. If the item is found in the list, its location is returned. If the search item is not in the list, -1 is returned.

```
int binarySearch(const int list[], int listLength, int searchItem)
{
    int first = 0;
    int last = listLength - 1;
    int mid;

    bool found = false;

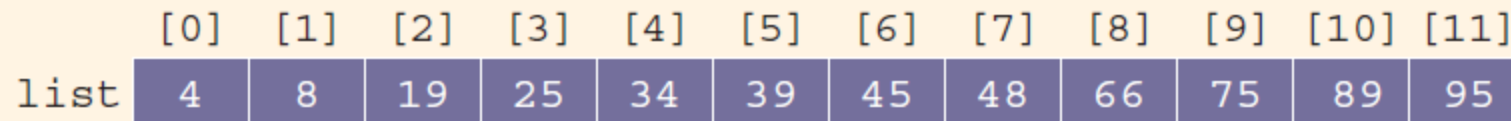
    while (first <= last && !found)
    {
        mid = (first + last) / 2;

        if (list[mid] == searchItem)
            found = true;
        else if (list[mid] > searchItem)
            last = mid - 1;
        else
            first = mid + 1;
    }

    if (found)
        return mid;
    else
        return -1;
} //end binarySearch
```

Note that in the binary search algorithm, two key (item) comparisons are made each time through the loop, except in the successful case—the last time through the loop—when only one key comparison is made.

Next, we walk through the binary search algorithm on the list shown in Figure 10-19.



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

FIGURE 10-19 Sorted list for a binary search

The size of the list in Figure 10-19 is 12, so `listLength = 12`. Suppose that the item for which we are searching is 89, so `searchItem = 89`. Before the **while** loop executes,

`first = 0`, `last = 11`, and `found = false`. In the following, we trace the execution of the `while` loop, showing the values of `first`, `last`, and `mid` and the number of key comparisons during each iteration.

Iteration	<code>first</code>	<code>last</code>	<code>mid</code>	<code>list[mid]</code>	No. of key comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1 (found is <code>true</code>)

The item is found at location 10, and the total number of key comparisons is 5.

References

1. Malik, D. S. (2010). *C++ programming: Program design including data structures*. Course Technology.