LESSON SET

8 Searching and Sorting Arrays

PURPOSE

1. To introduce the concept of a search routine
2. To introduce the linear and binary searches
3. To introduce the concept of a sorting algorithm
4. To introduce the bubble and selection sorts

PROCEDURE

1. Students should read the Pre-lab Reading Assignment before coming to lab.
2. Students should complete the Pre-lab Writing Assignment before coming to lab.
3. In the lab, students should complete labs assigned to them by the instructor.

PRE-LAB READING ASSIGNMENT

Search Algorithms

A search algorithm is a procedure for locating a specific datum from a collection of data.

For example, suppose you want to find the phone number for Wilson Electric in the phonebook. You open the phonebook to the business section under W and then look for all the entries that begin with the word Wilson. There are numerous such entries, so you look for the one(s) that end with Electric. This is an example of a **search algorithm**. Since each section in the phonebook is alphabetized, this is a particularly easy search. Of course, there are numerous types of "collections of data" that one could search. In this section we will focus on searching arrays. Two algorithms, the linear and binary searches, will be studied. We will see that each algorithm has its advantages and disadvantages.

Linear Search

The easiest array search to understand is probably the **linear search**. This algorithm starts at the beginning of the array and then steps through the elements sequentially until either the desired value is found or the end of the array is reached. For example, suppose we want to find the first occurrence of the letter "o" in the word "Harpoon." We can visualize the corresponding character array as follows:

0	1	2	3	4	5	6	7
H	a	r	p	o	o	n	\0

In C++ we can initialize the character array with the desired string:

```
char word[8] = "Harpoon";
```

So `word[0]='H'`, `word[3]= 'p'`, and `word[7] = '\0 '`. The `'\0'` marks the end of the string and is called the null character. It is discussed further in Lesson Set 10. If we perform a linear search looking for `'o'`, then we first check `word[0]` which is not equal to `'o'`. So we then move to `word[1]` which is also not equal to `'o'`. We continue until we get to `word[4]='o'`. At this point the subscript 4 is returned so we know the position in the array that contains the first occurrence of the letter `'o'`. What would happen if we searched for `'z'`? Certainly we would step through the array until we reached the end and not find any occurrence of `'z'`. What should the search function return in this case? It is customary to return –1 since this is not a valid array subscript. Here is the complete program that performs the linear search:

Sample Program 8.1:

```
// This program performs a linear search on a character array

#include <iostream>
using namespace std;

int searchList(char[], int, char); // function prototype
const int SIZE = 8;
```

```
int main()
{
      char word[SIZE] = "Harpoon";
      int found;
      char ch;

      cout << "Enter a letter to search for:" << endl;
      cin >> ch;

      found = searchList(word, SIZE, ch);
      if (found == -1)
            cout << "The letter " << ch
                 << " was not found in the list" << endl;
      else
            cout << "The letter " << ch <<" is in the " << found + 1
                 << " position of the list" << endl;

      return 0;

}


//*****************************************************************
//                      searchList
//
// task:          This searches an array for a particular value
// data in:       List of values in an array, the number of
//                elements in the array, and the value searched for
//                in the array
// data returned: Position in the array of the value or -1 if value
//                not found
//
//*****************************************************************

int searchList(char list[], int numElems, char value)
{
      for (int count = 0;count < numElems; count++)
      {
           if (list[count] == value)
                      // each array entry is checked to see if it contains
                      // the desired value.
           return count;
                      // if the desired value is found, the array subscript
                      // count is returned to indicate the location in the array
      }
      return -1;      // if the value is not found, -1 is returned
}
```

For example, suppose we wish to search the word "Harpoon" for the letter `'o'`. The function SearchList does the linear search and returns the index 4 of the array where `'o'` is found. However, the program outputs 5 for the position since we want to output the character's position within the string rather than its storage location in the `word` array. You have certainly noticed that there is a second occurrence of `'o'` in the word "Harpoon." However, the linear search does not find it since it quits after finding the first occurrence.

One advantage of the linear search is its simplicity. It is easy to step sequentially through an array and check each element for a designated value. Another advantage is that the elements of the array do not need to be in any order to implement the algorithm. For example, to search the integer arrays

First Array	23	45	12	456	99

Second Array	12	29	45	23	456

for the integer 99, the linear search will work. It will return 4 for the first array and –1 for the second. The main disadvantage of the linear search is that it is time-consuming for large arrays. If the desired piece of data is not in the array, then the search has to check every element of the array before it returns –1. Even if the desired piece of data is in the array, there is a very good chance that a significant portion of the array will need to be checked to find it. So we need a more efficient search algorithm for large arrays.

The Binary Search

A more efficient algorithm for searching an array is the **binary search** which eliminates half of the array every time it does a check. The drawback is that the data in the array must be ordered to use a binary search. If we are searching an array of integers, then the values stored in the array must be arranged in order from largest to smallest or smallest to largest.

Examples: Consider the following three integer arrays:

1)	19	15	13	13	11	6	-1	-3

2)	19	15	16	13	13	11	-1	-3

3)	-3	0	1	1	12	14	18	25

The arrays in 1) and 3) could be searched using a binary search. In 1) the values are arranged largest to smallest and in 3) the values are arranged smallest to largest. However, the array in 2) could not be searched using a binary search due to the first three elements of the array: the values of the elements decrease from 19 to 15 but then increase from 15 to 16.

Now that we know which types of arrays are allowed, let us next describe what the binary search actually does. For the sake of argument, let us assume the values of an integer array are arranged from smallest to largest and the integer we are searching for is stored in the variable `wanted`. We first pick an element in the middle of the array—let us call it `middle`. Think about how the number,

whether it be even or odd, of elements in the array affects this choice. If `middle` = `wanted`, then we are done. Otherwise, `wanted` must be either greater than or less than `middle`. If `wanted < middle`, then since the array is in ascending order we know that `wanted` must be before `middle` in the array so we can ignore the second half of the array and search the first half. Likewise, if `wanted > middle`, we can ignore the first half of the array and search just the second half. In both cases we can immediately eliminate half of the array. Once we have done this, we will choose the middle element of the half that is left over and then repeat the same process until either `wanted` is found or it is determined that `wanted` is not in the array.

The following program performs a binary search on an array of integers that is ordered from largest to smallest. Students should think about the logic of this search and how it differs from the argument given above for data ordered smallest to largest.

Sample Program 8.2:

```
// This program demonstrates a Binary Search


#include <iostream>
using namespace std;

int binarySearch(int [], int, int);  // function prototype

const int SIZE = 16;

int main()
{
      int found, value;
      int array[] = {34,19,19,18,17,13,12,12,12,11,9,5,3,2,2,0};
                    // array to be searched

      cout << "Enter an integer to search for:" << endl;
      cin >> value;

      found = binarySearch(array, SIZE, value);
                    // function call to perform the binary search
                    // on array looking for an occurrence of value
      if (found == -1)
            cout << "The value " << value << " is not in the list" << endl;
      else
      {
            cout << "The value " << value << " is in position number "
                 << found + 1 << " of the list" << endl;
      }
      return 0;
}
```

continues

```
//*****************************************************************
//                     binarySearch
//
// task:          This searches an array for a particular value
// data in:       List of values in an orderd array, the number of
//                elements in the array, and the value searched for
//                in the array
// data returned: Position in the array of the value or -1 if value
//                not found
//
//*****************************************************************
int binarySearch(int array[],int numElems,int value) //function heading
{
      int first = 0;               // First element of list
      int last = numElems - 1;     // last element of the list
      int middle;                  // variable containing the current
                                   // middle value of the list

      while (first <= last)
      {
            middle = first + (last - first) / 2;

      if (array[middle] == value)
            return middle;         // if value is in the middle, we are done

      else if (array[middle]<value)
            last = middle - 1;     // toss out the second remaining half of
                                   // the array and search the first
      else
            first = middle + 1;    // toss out the first remaining half of
                                   // the array and search the second
      }

      return -1;                   // indicates that value is not in the array
}
```

If you run this program and search for 2, the output indicates that 2 is in the 14th position of the array. Since 2 is in the 14th and 15th position, we see that the binary search found the first occurrence of 2 in this particular data set; however, in Lab 8.2 you will search for values other than 2 and see that there are other possibilities for which occurrence of a sought value is found.

Sorting Algorithms

We have just seen how to search an array for a specific piece of data; however, what if we do not like the order in which the data is stored in the array? For example, if a collection of numerical values is not in order, we might like them to be so we can use a binary search to find a particular value. Or, if we have a list of names, we may want them put in alphabetical order. To sort data stored in an array, one uses a **sorting algorithm**. In this section we will consider two such algorithms—the bubble sort and the selection sort.

The Bubble Sort

The bubble sort is a simple algorithm used to arrange data in either **ascending** (lowest to highest) or **descending** (highest to lowest) order. To see how this sort works, let us arrange the array below in ascending order.

9	2	0	11	5
Element 0	Element 1	Element 2	Element 3	Element 4

The bubble sort begins by comparing the first two array elements. If `Element 0 > Element 1`, which is true in this case, then these two pieces of data are exchanged. The array is now the following:

2	9	0	11	5
Element 0	Element 1	Element 2	Element 3	Element 4

Next elements 1 and 2 are compared. Since `Element 1 > Element 2`, another exchange occurs:

2	0	9	11	5
Element 0	Element 1	Element 2	Element 3	Element 4

Now elements 2 and 3 are compared. Since 9 < 11, there is no exchange at this step. Next elements 3 and 4 are compared and exchanged:

2	0	9	5	11
Element 0	Element 1	Element 2	Element 3	Element 4

At this point we are at the end of the array. Note that the largest value is now in the last position of the array. Now we go back to the beginning of the array and repeat the entire process over again. Elements 0 and 1 are compared. Since 2 > 0, an exchange occurs:

0	2	9	5	11
Element 0	Element 1	Element 2	Element 3	Element 4

Next elements 1 and 2 are compared. Since 2 < 9, no swap occurs. However, when we compare elements 2 and 3 we find that 9 > 5 and so they are exchanged. Since `Element 4` contains the largest value (from the previous pass), we do not need to make any more comparisons in this pass.

The final result is:

0	2	5	9	11
Element 0	Element 1	Element 2	Element 3	Element 4

The data is now arranged in ascending order and the algorithm terminates. Note that the larger values seem to rise "like bubbles" to the larger positions of the array as the sort progresses.

We just saw in the previous example how the first pass through the array positioned the largest value at the end of the array. This is always the case. Likewise, the second pass will always position the second to largest value in the second position from the end of the array. The pattern continues for the third pass, fourth pass, and so on until the array is fully sorted. Subsequent passes have one less array element to check than their immediate predecessor.

Sample Program 8.3:

```
// This program uses a bubble sort to arrange an array of integers in
// ascending order

#include <iostream>
using namespace std;

// function prototypes

void bubbleSortArray(int [], int);
void displayArray(int[], int);

const int SIZE = 5;

int main()
{
      int values[SIZE] = {9,2,0,11,5};

      cout << "The values before the bubble sort is performed are:" << endl;
      displayArray(values,SIZE);

      bubbleSortArray(values,SIZE);

      cout << "The values after the bubble sort is performed are:" << endl;
      displayArray(values,SIZE);

      return 0;
}
//*****************************************************************
//                      displayArray
//
// task:          to print the array
// data in:       the array to be printed, the array size
// data out:      none
//
//*****************************************************************

void displayArray(int array[], int elems)    // function heading
{                                            // displays the array
      for (int count = 0; count < elems; count++)
            cout << array[count] << "  " << endl;
}

//*****************************************************************
//                      bubbleSortArray
//
// task:          to sort values of an array in ascending order
// data in:       the array, the array size
// data out:      the sorted array
//
//*****************************************************************
```

```
void bubbleSortArray(int array[], int elems)
{
      bool swap;
      int temp;
      int bottom = elems - 1;     // bottom indicates the end part of the
                                  // array where the largest values have
                                  // settled in order
do
      {
      swap = false;
      for (int count = 0; count < bottom; count++)
      {
            if (array[count] > array[count+1])
            {              // the next three lines do a swap
               temp = array[count];
               array[count] = array[count+1];
               array[count+1] = temp;
               swap = true; // indicates that a swap occurred
            }
      }
      bottom--;     // bottom is decremented by 1 since each pass through
                    // the array adds one more value that is set in order

} while(swap != false);
         // loop repeats until a pass through the array with
         // no swaps occurs
}
```

While the bubble sort algorithm is fairly simple, it is inefficient for large arrays since data values only move one at a time.

The Selection Sort

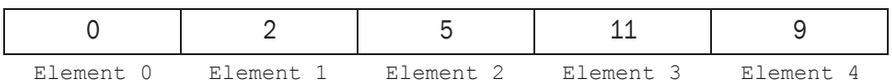
A generally more efficient algorithm for large arrays is the **selection sort**. As before, let us assume that we want to arrange numerical data in ascending order. The idea of the selection sort algorithm is to first locate the smallest value in the array and move that value to the beginning of the array (i.e., position 0). Then the next smallest element is located and put in the second position (i.e., position 1). This process continues until all the data is ordered. An advantage of the selection sort is that for n data elements at most n-1 moves are required. The disadvantage is that $n(n-1)/2$ comparisons are always required. To see how this sort works, let us consider the array we arranged using the bubble sort:

9	2	0	11	5
Element 0	Element 1	Element 2	Element 3	Element 4

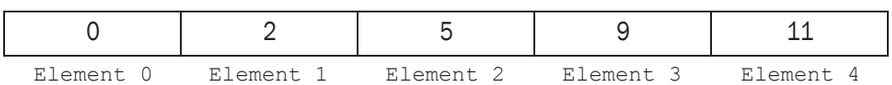
First the smallest value is located. It is 0, so the contents of `Element 0` and `Element 2` are swapped:

0	2	9	11	5
Element 0	Element 1	Element 2	Element 3	Element 4

Next we look for the second smallest value. The important point to note here is that we do not need to check `Element 0` again since we know it already contains the smallest data value. So the sort starts looking at `Element 1`. We see that the second smallest value is 2, which is already in `Element 1`. Starting at `Element 2` we see that 5 is the smallest of the remaining values. Thus the contents of `Element 2` and `Element 4` are swapped:



Finally, the contents of `Element 3` and `Element 4` are compared. Since 11 > 9, the contents are swapped leaving the array ordered as desired:



Sample Program 8.4:

```
// This program uses a selection sort to arrange an array of integers in
// ascending order

#include <iostream>
using namespace std;

// function prototypes

void selectionSortArray(int [], int);
void displayArray(int[], int);
const int SIZE = 5;

int main()
{
      int values[SIZE] = {9,2,0,11,5};

      cout << "The values before the selection sort is performed are:" << endl;
      displayArray(values,SIZE);

      selectionSortArray(values,SIZE);
      cout << "The values after the selection sort is performed are:" << endl;
      displayArray(values,SIZE);

      return 0;
}

//****************************************************************
//                      displayArray
//
// task:          to print the array
// data in:       the array to be printed, the array size
// data out:      none
//
//****************************************************************
```

```
void displayArray(int array[], int elems)    // function heading
{                                            // Displays array
      for (int count = 0; count < elems; count++)
            cout << array[count] << "  ";
      cout << endl;
}

//****************************************************************
//                      selectionSortArray
//
// task:          to sort values of an array in ascending order
// data in:       the array, the array size
// data out:      the sorted array
//
//****************************************************************


void selectionSortArray(int array[], int elems)
{
      int seek;      // array position currently being put in order
      int minCount;  // location of smallest value found
      int minValue;  // holds the smallest value found

      for (seek = 0; seek < (elems-1); seek++)  // outer loop performs the swap
                                                // and then increments seek
      {
           minCount = seek;
           minValue = array[seek];
           for(int index = seek + 1; index < elems; index++)
           {
                   // inner loop searches through array
                   // starting at array[seek] searching
                   // for the smallest value. When the
                   // value is found, the subscript is
                   // stored in minCount. The value is
                   // stored in minValue.

               if(array[index] < minValue)
               {
                   minValue = array[index];
                   minCount = index;
               }
           }

               // the following two statements exchange the value of the
               // element currently needing the smallest value found in the
               // pass(indicated by seek) with the smallest value found
               // (located in minValue)
```

continues

```
                array[minCount] = array[seek];
                array[seek] = minValue;

        }
    }
```

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. The advantage of a linear search is that it is _____.
2. The disadvantage of a linear search is that it is _____.
3. The advantage of a binary search over a linear search is that a binary search is _____.
4. An advantage of a linear search over a binary search is that the data must be _____for a binary search.
5. After 3 passes of a binary search, approximately what fraction of the original array still needs to be searched (assuming the desired data has not been found)? _____
6. While the _____sort algorithm is conceptually simple, it can be inefficient for large arrays because data values only move one at a time.
7. An advantage of the _____sort is that, for an array of size n, at most $n - 1$ moves are required.
8. Use the bubble sort on the array below and construct the first 3 steps that actually make changes. (Assume the sort is from smallest to largest).

19	-4	91	0	-17
Element 0	Element 1	Element 2	Element 3	Element 4
Element 0	Element 1	Element 2	Element 3	Element 4
Element 0	Element 1	Element 2	Element 3	Element 4
Element 0	Element 1	Element 2	Element 3	Element 4

9. Use the selection sort on the array below and construct the first 3 steps that actually make changes. (Assume the sort if from smallest to largest).

19	-4	91	0	-17
Element 0	Element 1	Element 2	Element 3	Element 4
Element 0	Element 1	Element 2	Element 3	Element 4
Element 0	Element 1	Element 2	Element 3	Element 4
Element 0	Element 1	Element 2	Element 3	Element 4