System Design and Programming II

CSCI – 1943

David L. Sylvester, Sr., Professor

Chapter 8

Searching and Sorting

Arrays

Introduction to Search Algorithms

A search algorithm is a method of locating a specific item in a larger collection of data.

It is very common for programs not only to store and process data stored in arrays, but to search arrays for specific items.

We will discuss two search algorithms

- Linear Search
- Binary Search

The Linear Search

A very simple algorithm, sometimes called a sequential search.

- Uses a loop to sequentially step through an array
- Starts the search with the first element

Search process

It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered. If the value being searched for is not found in the array, the algorithm will unsuccessfully search to the end of the array.

Linear Search pseudocode

```
set found to false.
set  position to -1.
Set  index to  0.
while found is false and index < number of elements
        if list[index] is equal to search value
                found = true.
                position = index.
        end if
        add 1 to index.
        return position.
```

found set to true when value found; will cause the program to exit the loop

element location of found value assigned to position

increment index by 1 in order to check the next element of the array

returns the value of position to the calling function (i.e. **main**)

Linear Search Sample Code

```
int  searchList (int list[ ] , int numElems, int value)
{
    int index = 0;              // used as a subscript to search the array
    int position = -1;          // used to record position of search value
    bool found = false;         // Flag to indicate if the value was found

    while (index < numElems && !found)
    {
        if (list[index] == value) // If the value is found
        {
            found = true;       // Set the flag
            position = index;   // Record the value's subscript
        }
        index++;
    }
    return position;
}
```

increment index by 1 in order to check the next element of the array

element location of found value assigned to position

returns the value of position to the calling function (i.e. **main**)

Linear Search	
Advantages	Disadvantages
Simple algorithm	Inefficient
Easy to understand	
No special requirement on stored data	

If the array being searched contains 20,000 elements, and the value is in the last element the algorithm will search the entire array to get to the searched value. (Reading the array 20,000 times.)

On average, an item is just as likely to be found at the beginning of the array as near the end. Typically, for an array of N items, the linear search will locate an item in N/2 attempts. If an array has 50,000 elements, the linear search will make a comparison with 25,000 items of them in a typical case. This is assuming, of course, that the search item is consistently found in the array. **(If not found in the array, you will be searching each element; from start to finish.)**

N/2 is the average number of comparisons, where N is the maximum number of comparisons.

The Binary Search

It’s a clever algorithm that is much more efficient than the linear search. ***Its only requirement is that the values in the array be sorted in order.***

Instead of testing the array’s first element, this algorithm starts with element in the middle. If that element happens to contain the desired value, then the search is over.

Otherwise, the value on the middle element is either greater than or less than value being searched for.

If greater than, (and in the list), the value will be found somewhere in the last half of the array.

If less than, (and again, in the list), the value will be found somewhere in the first half of the array.

(*In either case, half of the array has been eliminated.*)

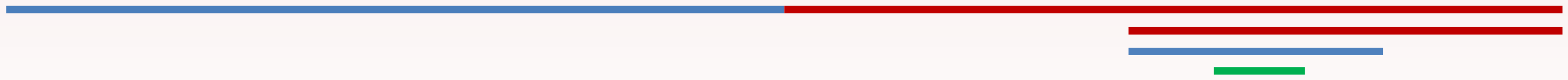
If the desired value is not found in the middle element, the procedure is repeated for the half of the array that potentially contains the value.

For instance, if the last half of the array is to be searched, the algorithm immediately tests its middle element. If the desired value isn’t found there, the search is narrowed to the quarter of the array that resides before or after that element.

This procedure continues the value searched until found or there are no more elements to test.

Ex: **Search for 83 using binary search**

Sorted List																	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
10	18	22	38	45	49	50	58	60	63	77	79	80	81	**83**	90	93	95



Binary Search pseudocode

```
set first index to 0.
set last index to the last subscript in the array.
set found to false.
set position to -1.
while found is not true and first is less than or equal to last
    set middle to the subscript halfway between array(first)
            and array(last)
    if array[middle] equals the desired value
            set found to true.  ←  found set to true when value found;
                                   will cause the program to exit the loop
            set position to middle.  ←  element location of found
                                        value assigned to position
    else if array[middle] is greater than the desired value
            set last to middle - 1.  ←  Set last to check lower
                                        half of array elements
    else
            set first to middle + 1.  ←  Set first to check greater
                                         half of array elements
    end if.
end while.
return position.  ←  returns the value of position to
                     the calling function (i.e. main)
```

Binary Search Sample Code

```
int binarySearch(int array[ ], int numElems, int value)
{
        int first = 0,                                     // First array element
        last = numElems - 1,                      // Last array element
        middle,                                            // Midpoint of search
        position = -1;                                     // Position of search value
   bool found = false;                                     // Flag
   while (!found && first <= last)
    {
        middle = (first + last) /2;                        // Calculate midpoint
        if (array[middle] == value)                        // If value is found at middle
        {
             found = true;  ←       found set to true when value found;
                                   will cause the program to exit the loop
             position = middle;  ←———————————————————— element location of found
                                                         value assigned to position
        }
        else if (array[middle] > value)       // if value is in lower half
             last = middle – 1;  ←    Set last to check lower
                                      half of array elements
        else
             first = middle + 1;  ←   Set first to check greater
                                      half of array elements
        }
        return position;  ←    returns the value of position to
                               the calling function (i.e. main)
}
```

Efficiency of Binary Search

The binary search obviously is much more efficient than the linear search. Every time it make a comparison and fails to find the desired item, it eliminates half of the remaining portion of the array that must be searched.

To search an array of 1,000 elements, would take no more than 10 comparisons.   Compared to the linear search algorithm, which would make an average of 500 comparisons.

Powers of 2 are used to calculate the maximum number of comparisons the binary search will make on an array or any size. Simply find the smallest power of 2 that is greater than or equal to the number of elements in an array.

Ex: A maximum of 16 comparisons will be made on an array of 50,000 elements (2^{16} = 65,536) and a maximum of 20 comparisons will be made on an array of 1,000,000 elements (2^{20} = 1,048,576).

Introduction to Sorting Algorithms

Sorting algorithms are used to arrange data into some order.

Oftentimes the data in an array must be sorted in some order.

Char data	sorted alphabetically, or grouped
Int data	highest to lowest

To sort data in an array, programmers must use an appropriate sorting algorithm. A sorting algorithm is a technique for scanning through an array and rearranging its contents in some specific order.

Two simple sorting algorithms are the:

- Bubble sort
- Selections sort

Bubble Sort

The bubble sort is an easy way to arrange data in ascending or descending order.

- Ascending (lowest to highest)
- Descending (Highest to lowest)

Given the following array,

7	2	3	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

- Bubble sort starts by comparing the first two elements in the array. If element 0 is greater than element 1, they are exchanged (swapped).

2	7	3	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

- This method is repeated with element 1 and 2. If element 1 is greater than element 2, they are exchanged. This is done

2	3	7	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

- Next elements 2 and 3 are compared. These elements are already in proper order (element 2 is less than 3), so no exchange takes place.
- As the cycle continues, elements 3 and 4 are compared. Once again, no exchange is necessary.
- When elements 4 and 5 are compared, however, an exchange must take place because element 4 is greater than element 5.

2	**3**	**7**	**8**	**1**	**9**
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

- At this point, the entire array has been scanned, but its contents aren't quite in the right order yet. So, the sort starts over again with elements 0 and 1, and so on.

2	**3**	**7**	**1**	**8**	**9**
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

- The sort repeatedly passes through the array until no exchanges are made. Ultimately, the array appears as follows:

1	**2**	**3**	**7**	**8**	**9**
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Bubble Sort pseudocode

```
do
    set swap flag to false.
    for count is set to each subscript in array from 0 through the
        next-to-last subscript            ← Loop will performed from the 0
                                            element to the max size of the array -1
      if array[count] is greater than array[count+1] ← check to see if  one element is
                                                       greater that the following one
        swap the contents of array[count] and array[count+1].
        set swap flag to true.            ← flag set to true if swap was done
      end if.
    end for.
while any elements have be swapped.
```

do while loop

Nested for loop

Bubble Sort Program

Array in Ascending Order

```
// This program asks for the number of hours worked
// by six employees.  It stores the values in an array.
#include <iostream>
using namespace std;

void sortArray(int[ ], int);
void showArray(int[ ], int);

int main()
{
    // Array of unsorted values
    int values[6] = {10, 2, 30, 40, 22, 6};

    // Display the values
    cout << "The unsorted values are:\n";
    showArray(values, 6);

    // Sort the values.
    sortArray(values, 6);

    // Display them again.
    cout << "The sorted values are:\n";
    showArray(values, 6);
    return 0;
}
```

```
void sortArray(int arr[ ], int size)
{
    bool swap;
    int temp;

    do
   {
        swap = false;
        for (int count = 0;count < size-1; count++)
        {
            If (arr[count] > arr[count+1])
           {
                        temp = arr[count];
                        arr[count] = arr[count+1];
                        arr[count+1] = temp;
                        swap = true;
           }
         }
     } while (swap);
}

void showArray(int arr[ ], int size)
{
    for (int count = 0; count < size -1; count++)
    cout << arr[count] << " ";
    cout << endl;
}
```

Inside the sortArray function is a for loop nested inside a *do-loop*. The for loop sequences through the entire array, comparing each element with its neighbor, and swapping them if necessary. Anytime two elements are exchanged, the flag variable is set to true.

The *for-loop* must be executed repeatedly until it can sequence through the entire array without making any exchanges. This is why it is nested inside a do-while loop. The do-while loop sets swap to false, and then executes the for loop. If swap is set to true after the for loop has finished the do-while loop repeats.

```
for (int count = 0; count < (size-1); count++)
    if (array[count] > array[count+1];
```

The variable count holds the array subscript values. It starts at zero and is incremented as long as it is less than size-1. the value of size is the number of elements in the array, and count stops just short of reaching this value because the following line compares each element with the one after it.

When array[count] is the next-to-last element, it will be compared to the last element. If the for loop were allowed to increment count past size-1, the last element in the array would be compared to a value outside the array.

The Selection Sort

The bubble sort is inefficient for large arrays because items only move by one element at a time. The selection sort, however usually performs fewer exchanges because it moves items immediately to their final position in the array.

Selection Sort Procedures:

1. Smallest value in array is located and moved to element 0,
2. The next smallest value is located and moved to element 1,
3. The process continues until all elements have been placed in their proper order.

Selection Sort

The selection sort scans the array starting with element 0, and locates the element with the smallest value. The contents of this element are then swapped with the content of element 0.

Given the following array,

5	7	2	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The first pass would yield,

1	7	2	8	9	5
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The second pass would yield,

1	2	7	8	9	5
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The third pass would yield,

1	2	5	8	9	7
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The process is repeated until all elements have been place in their proper order.

Selection Sort pseudocode

```
for startScan is set to each subscript in array from 0 through the next-to-
last subscript.
    set index variable to startScan.
    set minIndex to startScan.
    set minValue to  array[startScan].
    for index is set to each subscript in array from (startScan +1) through
            the last subscript
        if array[index] is less than minValue
            set minValue to array[index].
            set minIndex to index.
        endif.
    end for.
    set array[minIndex] to array[startScan].
    set array[startScan] to minValue.
end for.
```

for loop

Nested for loop

Loop will performed from the 0 element to the max size of the array -1

check to see if one element is greater that the following one

flag set to true if swap was done

Inner loop sequences through the array, starting at startScan +1, searching for the smallest value.

Outer loop then exchanges the contents of this element with array[startScan] and increments startScan

Selection Sort Program

```
// This program uses the selection sort algorithm to sort an
// array in ascending order.
#include <iostream>
using namespace std;

// Function prototypes
void selectionSort(int[ ], int);
void showArray(int[ ], int);

int main()
{
    // Define an array with unsorted values
    const int SIZE = 6;
    int values[SIZE] = {5, 7, 2, 8, 9, 1};

    // Display the values.
    cout << "The unsorted values are\n“;
    showArray(values, SIZE);

    // Sort the values.
    selectionSort(values, SIZE);

    // Display the values again.
    cout << "The sorted values are\n";
    showArray(values, SIZE);
    return 0;
}
```

```
// Definition of function selectSort.

void selectionSort(int array[ ], int size)
{
    int startScan, minIndex, minValue;

    for (startScan = 0; startScan < (size - 1); startScan++)
    {
        minIndex = startScan;
        minValue = array[startScan];
        for(int index = startScan + 1; index < size; index++)
        {
            if (array[index] < minValue)
            {
                minValue = array[index];
                minIndex = index;
            }
        }
        array[minIndex] = array[startScan];
        array[startScan] = minValue;
    }
}

// Definition of function showArray

void showArray(int array[], int size)
{
    for (int count = 0; count < size; count++)
        cout << array[count] << " ";
        cout << endl;
}
```

Vectors - Initialization

When writing a c++ program, you must remember that vectors do not accept initialization lists. In order to input values into the vector must use the push_back member function.

Ex:

```
for (int value = 912; value <= 922; value++)
id.push_back(value);

//initialize the units vector with data.
units.push_back(842);
units.push_back(416);
units.push_back(127);
units.push_back(514);
units.push_back(437);
units.push_back(269);
units.push_back(97);
units.push_back(492);
units.push_back(212);
```

Notice that each time a value is added to the vector, the push_back function is called because the [ ] operators cannot be used to add values to the vector

Vectors - Initialization

The code on the previous page appears repetitious because the push_back member function is written each time a value is placed in the vector.

This code can be simplified by storing valued needing to stored into the vector into an array, then using loops to call the push_back member function.

Ex:

```
const int NUM_PRODS = 9;
int unitsSold[NUM_PRODS] = {843, 416, 127, 514, 437
                            269, 97, 492, 212};

// Loop initializing units vector
for (count  = 0; count < NUM_PRODS; count++)
        units.push_back(unitsSold[count]);
```

Vectors – Function Headers

Ex:
```
void initVector (vector<int> &id, vector<int> &units
        vector<double> &prices)
```

Vector parameters are references (**as indicated by the & that precedes the parameter name**). This is not needed when using arrays, because by default, vectors are passed by value, whereas arrays are only passed by reference. When a value in a vector argument will be changed, the vector must be passed into a reference parameter.

Function prototype

Ex:
```
void initVector(vector<int> &, vector<int> &,
        vector<double> &)
```

Function call

Ex:
```
void initVector(id, units, prices);
```