EENG212 – Algorithms & Data Structures
Fall 08/09 – Lecture Notes # 2

**OUTLINE**

♦ Review of Arrays in C
♦ Declaration and Initialization of Arrays
♦ Sorting: Bubble Sort
♦ Searching: Linear and Binary Search

**ARRAYS**

♦ Arrays are defined to be a sequence/set of data elements of the same type. Having an array, each array element can be accessed by its position in the sequence of the array.

♦ **Declaration** of the Arrays: Any array declaration contains: the *array name*, the *element type* and the *array size*.

Ex: 
```
int a[20], b[3], c[7];
float f[5], c[2];
char m[4], n[20];
```

♦ **Initialisation** of an array is the process of assigning initial values. Typically declaration and initialisation are combined.

Ex: 
```
int a[4]={1,3,5,2};
float b[3]={2.0, 5.5, 3.14};
char name[4]= {'E','m','r','e'};
```

Ex: Write a program to calculate and print the average of the following array of integers.
(4, 3, 7, -1, 7, 2, 0, 4, 2, 13)

```
#include<stdio.h>
#define size 10
int main()
{
    int x[10]={4,3,7,-1,7,2,0,4,2,13}, i, sum=0;
    float av;
    for(i=0,i<=size-1;i++)
        sum = sum + x[i];
    av = (float)sum/size;
    printf("The average of the numbers=%.2f\n", av);
    return 0;
}
```

**SORTING**

♦ **Sorting** an array is the ordering the array elements in ascending (increasing -from min to max) or descending (decreasing – from max to min) order.

Ex: {2 1 5 3 2} ------ → {1, 2, 2, 3, 5}

**Bubble Sort:** Smaller values in the list gradually “bubble” their way upward to the top of the array. The technique makes several passes through the array. On each pass successive pairs of elements are compared. In the case of Ascending Order Sorting, If the pair is in increasing order (or equal) the pair is unchanged. If a pair is in descending order, their values are swapped in the array:
/* This program sorts the array elements in the ascending order*/
#include <stdio.h>
#define  SIZE 5
void BubbleSort(int []);
int main()
{
    int a[SIZE] = {2,1,5,3,2};
    int i;
    printf("The elements of the array before sorting\n");
    for (i=0; i<= SIZE-1; i++)
        printf("%4d", a[i]);
    BubbleSort(a);
    printf("The elements of the array after sorting\n");
    for (i=0; i<= SIZE-1; i++)
        printf("%4d", a[i]);
    return 0;
}

void BubbleSort(int A[ ])
{
    int i, pass, hold;
    for(pass=1; pass<= SIZE-1; pass++){
        for(i=0; i<= SIZE-2; i++) {
            if(A[i] >A[i+1]){
                hold =A[i];
                A[i]=A[i+1];
                A[i+1]=hold;
            }
        }
    }
}
Ex: Write a program to determine the median of the array given below:

(9, 4, 5, 1, 7, 78, 22, 15, 96, 45)

Note that the median of an array is the middle element of a sorted array.

/* This program determines the median of an array*/
#include <stdio.h>
#define SIZE 10
void BubbleSort(int []);
int main()
{
    int a[SIZE]= {9, 4, 5, 1, 7, 78, 22, 15, 96, 45}, median;
    int i;
    printf(“The elements of the input array is\n”);
    for (i=0; i<= SIZE-1; i++)
        printf(“%4d”, a[i]);
    BubbleSort(a);
    printf(“The elements of the sorted array is\n”);
    for (i=0; i<= SIZE-1; i++)
        printf(“%4d”, a[i]);
    median = a[SIZE/2];
    printf(“The median of the array is=%d”, median);
    return 0;
}

void BubbleSort(int A[ ])
{
    int i, pass, hold;
    for (pass=1; pass<= SIZE-1; pass++){
        for (i=0; i<= SIZE-2; i++)
            if(A[i] >A[i+1]){
                hold =A[i];
                A[i]=A[i+1];
                A[i+1]=hold;
            }
    }
}

SEARCHING

- The process of finding a particular element of an array is called searching. There two popular searching techniques: Linear search and binary search. The linear search compares each array element with the search key.
- If the search key is a member of the array, typically the location of the search key is reported to indicate the presence of the search key in the array. Otherwise, a sentinel value (i.e. -1) is reported to indicate the absence of the search key in the array.

Linear Search: Each member of the array is visited until the search key is found.

Ex: Write a program to search for the search key entered by the user in the following array:

(9, 4, 5, 1, 7, 78, 22, 15, 96, 45)

You can use the linear search in this example.
/* This program is an example of the Linear Search*/
#include <stdio.h>
#define SIZE 10
int LinearSearch(int [], int);
int main()
{
    int a[SIZE]= {9, 4, 5, 1, 7, 78, 22, 15, 96, 45}, key, pos;
    printf("Enter the Search Key\n");
    scanf("%d", &key);
    pos = LinearSearch(a, key);
    if(pos == -1)
        printf("The search key is not in the array\n");
    else
        printf("The search key %d is at location %d\n", key, pos);
    return 0;
}

int LinearSearch (int b[], int skey)
{
    int i;
    for (i=0; i<= SIZE-1; i++)
        if(b[i] == skey)
            return i;
    return -1;
}

Binary Search: Given a sorted array Binary Search algorithm can be used to perform fast searching of a search key on the sorted array.

The following program uses pointer notation to implement the binary search algorithm for the search key entered by the user in the following array:

(3, 5, 9, 11, 15, 17, 22, 25, 37, 68)

#include <stdio.h>
#define SIZE 10
int BinarySearch(int [], int);
int main()
{
    int a[SIZE]= {3, 5, 9, 11, 15, 17, 22, 25, 37, 68}, key, pos;
    printf("Enter the Search Key\n");
    scanf("%d", &key);
    pos = BinarySearch(a, key);
    if(pos == -1)
        printf("The search key is not in the array\n");
    else
        printf("The search key %d is at location %d\n", key, pos);
    return 0;
}
int BinarySearch (int A[], int skey)
{
    int low=0, high=SIZE-1, middle;
    while (low <= high)
    {
        middle = (low+high)/2;
        if (skey == A[middle])
            return middle;
        else if (skey < A[middle])
            high = middle - 1;
        else
            low = middle + 1;
    }
    return -1;
}

The Computational Complexity (Cost) of the Binary Search algorithm is measured by the maximum (worst case) number of Comparisons it performs for searching operations.

The searched array is divided by 2 for each comparison/iteration. Therefore, the maximum number of comparisons is measured by:

$log_2(n)$  where $n$ is the size of the array

Ex: If a given sorted array 1024 elements, then the maximum number of comparisons required is:

$log_2(1024) = 10$  (only 10 comparisons is enough)

Note that the Computational Complexity of the Linear Search is the maximum number of comparisons you need to search the array. As you are visiting all the array elements in the worst case, then, the number of comparisons required is:

$n$  (n is the size of the array)

Ex: If a given an array of 1024 elements, then the maximum number of comparisons required is:

$n = 1024$  (As many as 1024 comparisons may be required)