UNIT – 5

Searching

There are basically two aspects of computer programming. One is data organization also commonly called as data structures. Till now we have seen about data structures and the techniques and algorithms used to access them. The other part of computer programming involves choosing the appropriate algorithm to solve the problem. Data structures and algorithms are linked each other. After developing programming techniques to represent information, it is logical to proceed to manipulate it. This chapter introduces this important aspect of problem solving.

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

7.1. Linear Search:

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons.

The time complexity of linear search is O(n).

Algorithm:

Let array a[n] stores n elements. Determine whether element 'x' is present or not.

```
linsrch(a[n], x)
{
        index = 0;
        flag = 0;
        while (index < n) do
        {
                if (x == a[index])
                {
                        flag = 1;
                        break;
                }
```

```
            index ++;
        }
        if(flag == 1)
            printf("Data found  at %d position", index);
        else
            printf("data not found");

}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for: 45, we'll look at 1 element before success
39, we'll look at 2 elements before success
8, we'll look at 3 elements before success
54, we'll look at 4 elements before success
77, we'll look at 5 elements before success
38 we'll look at 6 elements before success
24, we'll look at 7 elements before success
16, we'll look at 8 elements before success
4, we'll look at 9 elements before success
7, we'll look at 10 elements before success
9, we'll look at 11 elements before success
20, we'll look at 12 elements before success

For any element not in the list, we'll look at 12 elements before failure.

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	0	1	2	3	4	5	6	7	8
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

1. Searching for x = 7 Search successful, data found at 3rd position.

2. Searching for x = 82 Search successful, data found at 7th position.

3. Searching for x = 42 Search un-successful, data not found.

7.1.1. A non-recursive program for Linear Search:

```
# include <stdio.h>
# include <conio.h>

main()
{
	int number[25], n, data, i, flag = 0;
	clrscr();
	printf("\n Enter the number of elements: ");
	scanf("%d", &n);
	printf("\n Enter the elements: ");
	for(i = 0; i < n; i++)
		scanf("%d", &number[i]);
	printf("\n Enter the element to be Searched: ");
	scanf("%d", &data);
	for( i = 0; i < n; i++)
	{
		if(number[i] == data)
		{
			flag = 1;
			break;
		}
	}
	if(flag == 1)
		printf("\n Data found at location: %d", i+1);
	else
		printf("\n Data not found ");
}
```

7.1.2. A Recursive program for linear search:

```
# include <stdio.h>
# include <conio.h>
void linear_search(int a[], int data, int position, int n)
{
	if(position < n)
	{
		if(a[position] == data)
			printf("\n Data Found at %d ", position);
		else
			linear_search(a, data, position + 1, n);
	}
	else
		printf("\n Data not found");
}
void main()
{
	int a[25], i, n, data;
	clrscr();
	printf("\n Enter the number of elements: ");
	scanf("%d", &n);
	printf("\n Enter the elements: ");
	for(i = 0; i < n; i++)
		scanf("%d", &a[i]);
	printf("\n Enter the element to be seached: ");
	scanf("%d", &data);
	linear_search(a, data, 0, n);
	getch();
}
```

7.2. BINARY SEARCH

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid]. Similarly, if a[mid] > x, then further search is only necessary in that part of the file which follows a[mid].

If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array a[n] of elements in increasing order, $n \geq 0$, determine whether 'x' is present, and if so, set j such that x = a[j] else return 0.

```
binsrch(a[], n, x)
{
        low = 1; high = n;
        while (low ≤ high) do
        {
                mid = ⌊(low + high)/2⌋
                if (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid + 1;
                else return mid;
        }
        return 0;
 }
```

low and high are integer variables such that each time through the loop either 'x' is found or low is increased by at least one or high is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually low will become greater than high causing termination in a finite number of steps if 'x' is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	1	2	3	4	5	6	7	8	9	10	11	12
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, found

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, found

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8, found

If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, found

If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, found

If we are searching for x = 20: (This needs 1 comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20, found

If we are searching for x = 24: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24, found

If we are searching for x = 38: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24
low = 8, high = 8, mid = 16/2 = 8, check 38, found

If we are searching for x = 39: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39, found

If we are searching for x = 45: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54

low = 10, high = 10, mid = 20/2 = 10, check 45, found

If we are searching for x = 54: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54, found

If we are searching for x = 77: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 12, high = 12, mid = 24/2 = 12, check 77, found

The number of comparisons necessary by search element:

20 – requires 1 comparison;
8 and 39 – requires 2 comparisons;
4, 9, 24, 54 – requires 3 comparisons and
7, 16, 38, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding 37/12 or approximately 3.08 comparisons per successful search on the average.

Example 2:

Let us illustrate binary search on the following 9 elements:

Index	0	1	2	3	4	5	6	7	8
Elements	-15	-6	0	7	9	23	54	82	101

Solution:

The number of comparisons required for searching different elements is as follows:

1. If we are searching for x = 101: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

2. Searching for x = 82: (Number of comparisons = 3)

low	high	mid
1	9	5
6	9	7
8	9	8
		found

3. Searching for x = 42: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

4. Searching for x = -14: (Number of comparisons = 3)

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	3	2	3	4	1	3	2	3	4

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If $x < a(1)$, $a(1) < x < a(2)$, $a(2) < x < a(3)$, $a(5) < x < a(6)$, $a(6) < x < a(7)$ or $a(7) < x < a(8)$ the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons.

Thus the average number of element comparisons for an unsuccessful search is:

$$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$$

Time Complexity:

The time complexity of binary search in a successful search is O(log n) and for an unsuccessful search is O(log n).

7.2.1. A non-recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

main()
{
	int number[25], n, data, i, flag = 0, low, high, mid;
	clrscr();
	printf("\n Enter the number of elements: ");
	scanf("%d", &n);
	printf("\n Enter the elements in ascending order: ");
	for(i = 0; i < n; i++)
		scanf("%d", &number[i]);
	printf("\n Enter the element to be searched: ");
	scanf("%d", &data);
	low = 0; high =  n-1;
	while(low <= high)
	{
		mid = (low + high)/2;
		if(number[mid] == data)
		{
			flag = 1;
			break;
		}
		else
		{
			if(data < number[mid])
				high = mid - 1;
			else
				low = mid + 1;
		}
	}
	if(flag == 1)
		printf("\n Data found at location: %d", mid + 1);
	else
		printf("\n Data Not Found ");
}
```

7.2.2. A recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

void bin_search(int a[], int data, int low, int high)
{
	int mid ;
	if( low <= high)
	{
		mid = (low + high)/2;
		if(a[mid] == data)
			printf("\n Element found at location: %d ", mid + 1);
		else
		{
			if(data < a[mid])
				bin_search(a, data, low, mid-1);
			else
```

```
                        bin_search(a, data, mid+1, high);
                }
        }
        else
                printf("\n Element not found");
}
void main()
{

                int a[25], i, n, data; clrscr();
                printf("\n Enter the number of elements: ");
                scanf("%d", &n);
                printf("\n Enter the elements in ascending order: ");
                for(i = 0; i < n; i++)
                        scanf("%d", &a[i]);
                printf("\n Enter the element to be searched: ");
                scanf("%d", &data);
                bin_search(a, data, 0, n-1);
                getch();
                }
```

Exercises

1. Write a recursive “C” function to implement binary search and compute its time complexity.

2. An array contains “n” elements of numbers. The several elements of this array may contain the same number “x”. Write an algorithm to find the total number of elements which are equal to “x” and also indicate the position of the first such element in the array.

Multiple Choice Questions

1. What is the worst-case time for serial search finding a single item in an array? [D]
   A. Constant time
   B. Quadratic time
   C. Logarithmic time
   D. Linear time

2. What is the worst-case time for binary search finding a single item in an array? [B]
   A. Constant time
   B. Quadratic time
   C. Logarithmic time
   D. Linear time

3. What additional requirement is placed on an array, so that binary search may be used to locate an entry? [C]
   A. The array elements must form a heap.
   B. The array must have at least 2 entries
   C. The array must be sorted.
   D. The array’s size must be a power of two.

4. Which searching can be performed recursively ? [B]
   A. linear search
   B. both
   C. Binary search
   D. none

5. Which searching can be performed iteratively ? [B]
   A. linear search
   B. both
   C. Binary search
   D. none

6. In which searching technique elements are eliminated by half in each pass . [C]
   A. Linear search
   B. both
   C. Binary search
   D. none

7. Binary search algorithm performs efficiently on a [C]
   E. linked list
   F. both
   G. array
   H. none