

---

## UNIT 9    SEARCHING

---

Structure	Page Nos.
9.0    Introduction	40
9.1    Objectives	40
9.2    Linear Search	41
9.3    Binary Search	44
9.4    Applications	47
9.5    Summary	48
9.6    Solutions / Answers	48
9.7    Further Readings	48

---

### 9.0    INTRODUCTION

---

Searching is the process of looking for something: Finding one piece of data that has been stored within a whole group of data. It is often the most time-consuming part of many computer programs. There are a variety of methods, or algorithms, used to search for a data item, depending on how much data there is to look through, what kind of data it is, what type of structure the data is stored in, and even where the data is stored - inside computer memory or on some external medium.

Till now, we have studied a variety of data structures, their types, their use and so on. In this unit, we will concentrate on some techniques to *search* a particular data or piece of information from a large amount of data. There are basically two types of searching techniques, ***Linear or Sequential Search and Binary Search***.

Searching is very common task in day-to-day life, where we are involved some or other time, in searching either for some needful at home or office or market, or searching a word in dictionary. In this unit, we see that if the things are organised in some manner, then search becomes efficient and fast.

All the above facts apply to our computer programs also. Suppose we have a telephone directory stored in the memory in an array which contains Name and Numbers. Now, what happens if we have to find a number? The answer is search that number in the array according to name (given). If the names were organised in some order, searching would have been fast.

*So, basically a search algorithm is an algorithm which accepts an argument 'a' and tries to find the corresponding data where the match of 'a' occurs in a file or in a table.*

---

### 9.1    OBJECTIVES

---

After going through this unit, you should be able to:

- know the basic concepts of searching;
- know the process of performing the Linear Search;
- know the process of performing the Binary Search and
- know the applications of searching.

## 9.2 LINEAR SEARCH

Linear search is not the most efficient way to search for an item in a collection of items. However, it is very simple to implement. Moreover, if the array elements are arranged in random order, it is the only reasonable way to search. In addition, efficiency becomes important only in large arrays; if the array is small, there aren't many elements to search and the amount of time it takes is not even noticed by the user. Thus, for many situations, linear search is a perfectly valid approach.

Before studying Linear Search, let us define some terms related to search.

A **file** is a collection of records and a record is in turn a collection of fields. A field, which is used to differentiate among various records, is known as a '**key**'.

For example, the telephone directory that we discussed in previous section can be considered as a file, where each record contains two fields: name of the person and phone number of the person.

Now, it depends on the application whose field will be the 'key'. It can be the name of person (usual case) and it can also be phone number. We will locate any particular record by matching the input argument 'a' with the key value.

The simplest of all the searching techniques is *Linear or Sequential Search*. As the name suggests, all the records in a file are searched sequentially, one by one, for the matching of key value, until a match occurs.

The Linear Search is applicable to a table which it should be organised in an array. Let us assume that a file contains 'n' records and a record has 'a' fields but only one key. The values of key are organised in an array say 'm'. As the file has 'n' records, the size of array will be 'n' and value at position R(i) will be the key of record at position i. Also, let us assume that 'el' is the value for which search has to be made or it is the search argument.

Now, let us write a simple algorithm for Linear Search.

### Algorithm

Here, m represents the unordered array of elements  
 n represents number of elements in the array and  
 el represents the value to be searched in the list

Sep 1: [Initialize]

k=0

flag=1

Step 2: Repeat step 3 for k=0,1,2,...n-1

Step 3: if (m[k]=el )

then

flag=0

print "Search is successful" and element is found at location (k+1)

stop

endif

Step 4: if (flag=1) then

print "Search is unsuccessful"

endif

Step 5: stop

Program 9.1 gives the program for Linear Search.

```
/*Program for Linear Search*/
/*Header Files*/
#include<stdio.h>
#include<conio.h>
/*Global Variables*/
int search;
int flag;
/*Function Declarations*/
int input (int *, int, int);
void linear_search (int *, int, int);
void display (int *, int);
/*Functions */
void linear_search(int m[ ], int n, int el)
{
    int k;
    flag = 1;
    for(k=0; k<n; k++)
    {
        if(m[k]==el
        {
            printf("\n Search is Successful\n");
            printf("\n Element : %i Found at location : %i", element, k+1);
            flag = 0;
        }
    }
    if(flag==1)
        printf("\n Search is unsuccessful");
}
void display(int m[ ], int n)
{
    int i;
    for(i=0; i< 20; i++)
    {
        printf("%d", m[i];
    }
}
int input(int m[ ], int n, int el)
{
    int i;
    n = 20;
    el = 30;
    printf("Number of elements in the list : %d", n);
    for(i=0; i<20; i++)
    {
        m[i]=rand( )%100;
    }
    printf("\n Element to be searched :%d", el);
    search = el;
    return n;
}
/* Main Function*/
```

```

void main( )
{
    int n, el, m[200];
    number = input(m, n, el);
    el = search;
    printf("\n Entered list as follows: \n");
    display(m, n);
    linear_search(m, n, el);
    printf("\n In the following list\n");
    display(m, n);
}

```

### Program 9.1: Linear Search

Program 9.1 examines each of the key values in the array ‘m’, one by one and stops when a match occurs or the total array is searched.

#### Example:

A *telephone directory* with  $n = 10$  records and Name field as key. Let us assume that the names are stored in array ‘m’ i.e.  $m(0)$  to  $m(9)$  and the search has to be made for name “Radha Sharma”, i.e. element = “Radha Sharma”.

#### Telephone Directory

<i>Name</i>	<i>Phone No.</i>
Nitin Kumar	25161234
Preeti Jain	22752345
Sandeep Singh	23405678
Sapna Chowdhary	22361111
Hitesh Somal	24782202
R.S.Singh	26254444
Radha Sharma	26150880
S.N.Singh	25513653
Arvind Chittora	26252794
Anil Rawat	26257149

The above algorithm will search for element = “Radha Sharma” and will stop at 6th index of array and the required phone number is “26150880”, which is stored at position 7 i.e.  $6+1$ .

#### Efficiency of Linear Search

How many number of comparisons are there in this search in searching for a given element?

The number of comparisons depends upon where the record with the argument key appears in the array. If record is at the first place, number of comparisons is ‘1’, if record is at last position ‘n’ comparisons are made.

If it is equally likely for that the record can appear at any position in the array, then, a successful search will take  $(n+1)/2$  comparisons and an unsuccessful search will take ‘n’ comparisons.

In any case, the order of the above algorithm is  $O(n)$ .

### ☞ Check Your Progress 1

- 1) Linear search uses an exhaustive method of checking each element in the array against a key value. When a match is found, the search halts. Will sorting the array before using the linear search have any effect on its order of efficiency?  
.....
- 2) In a best case situation, the element was found with the fewest number of comparisons. Where, in the list, would the key element be located?  
.....

---

## 9.3 BINARY SEARCH

---

An unsorted array is searched by *linear search* that scans the array elements one by one until the desired element is found.

The reason for sorting an array is that we search the array “quickly”. Now, if the array is sorted, we can employ *binary search*, which brilliantly halves the size of the search space each time it examines one array element.

An array-based binary search selects the middle element in the array and compares its value to that of the key value. Because, the array is sorted, if the key value is less than the middle value then the key must be in the first half of the array. Likewise, if the value of the key item is greater than that of the middle value in the array, then it is known that the key lies in the second half of the array. In either case, we can, in effect, “throw out” one half of the search space or array with only one comparison.

Now, knowing that the key must be in one half of the array or the other, the binary search examines the mid value of the half in which the key must reside. The algorithm thus narrows the search area by half at each step until it has either found the key data or the search fails.

As the name suggests, binary means two, so it divides an array into *two* halves for searching. This search is applicable only to an **ordered table** (in either ascending or in descending order).

Let us write an algorithm for Binary Search and then we will discuss it. The array consists of elements stored in ascending order.

### Algorithm

Step 1: Declare an array ‘k’ of size ‘n’ i.e. k(n) is an array which stores all the keys of a file containing ‘n’ records

Step 2:  $i \leftarrow 0$

Step 3:  $low \leftarrow 0$ ,  $high \leftarrow n-1$

Step 4: while ( $low \leq high$ )do  
     $mid = (low + high)/2$   
    if ( $key = k[mid]$ ) then  
        write “record is at position”,  $mid+1$  //as the array  
        starts from the 0<sup>th</sup> position  
    else  
        if ( $key < k[mid]$ ) then  
             $high = mid - 1$

```

        else
            low = mid + 1
        endif
    endif
endwhile

```

Step 5: Write “Sorry, key value not found”

Step 6: Stop

Program 9.2 gives the program for Binary Search.

```

/*Header Files*/
#include<stdio.h>
#include<conio.h>
/*Functions*/
void binary_search(int array[ ], int value, int size)
{
    int found=0;
    int high=size-1, low=0, mid;
    mid = (high+low)/2;
    printf("\n\n Looking for %d\n", value);
    while((!found)&&(high>=low))
    {
        printf("Low %d Mid%d High%d\n", low, mid, high);
        if(value==array[mid] )
        {printf("Key value found at position %d",mid+1);
          found=1;
        }
        else
        {if (value<array[mid])
            high = mid-1;
          else
            low = mid+1;
          mid = (high+low)/2;
        }
    }
    if (found==1
    printf("Search successful");
    else
    printf("Key value not found");
}
/*Main Function*/
void main(void)
{
    int array[100], i;
    /*Inputting Values to Array*/
    for(i=0;i<100;i++)
    { printf("Enter the name:");
      scanf("%d", array[i]);
    }
    printf("Result of search %d\n", binary_searchy(array,33,100));
    printf("Result of search %d\n", binary_searchy(array, 75,100));
    printf("Result of search %d\n", binary_searchy(array,1,100));
}

```

**Program 9.2 : Binary Search**

### Example:

Let us consider a file of 5 records, i.e.,  $n = 5$   
And  $k$  is a sorted array of the keys of those 5 records.

$k$	
11	0
22	1
33	2
44	3
55	4

Let key = 55, low = 0, high = 4

Iteration 1:  $\text{mid} = (0+4)/2 = 2$

$k(\text{mid}) = k(2) = 33$

Now  $\text{key} > k(\text{mid})$

So  $\text{low} = \text{mid} + 1 = 3$

Iteration 2: low = 3, high = 4 (low  $\leq$  high)

$\text{Mid} = 3+4 / 2 = 3.5 \sim 3$  (integer value)

Here  $\text{key} > k(\text{mid})$

So  $\text{low} = 3+1 = 4$

Iteration 3: low = 4, high = 4 (low  $\leq$  high)

$\text{Mid} = (4+4)/2 = 4$

Here  $\text{key} = k(\text{mid})$

So, the record is at  $\text{mid}+1$  position, i.e., 5

### Efficiency of Binary Search

Each comparison in the binary search reduces the number of possible candidates where the key value can be found by a factor of 2 as the array is divided in two halves in each iteration. Thus, the maximum number of key comparisons are approximately  $\log n$ . So, the order of binary search is  **$O(\log n)$** .

### **Comparative Study of Linear and Binary Search**

Binary search is lots faster than linear search. Here are some comparisons:

#### **NUMBER OF ARRAY ELEMENTS EXAMINED**

array size	linear search (avg. case)	binary search (worst case)
8	4	4
128	64	8
256	128	9
1000	500	11
100,000	50,000	18

A **binary search** on an array is  $O(\log_2 n)$  because at each test, you can “throw out” one half of the search space or array whereas a **linear search** on an array is  $O(n)$ .

It is noteworthy that, for very small arrays a **linear search** can prove faster than a **binary search**. However, as the size of the array to be searched increases, the binary

search is the clear winner in terms of number of comparisons and therefore overall speed.

Still, the binary search has some drawbacks. First, it requires that the data to be searched be in sorted order. If there is even one element out of order in the data being searched, it can throw off the entire process. When presented with a set of unsorted data, the efficient programmer must decide whether to sort the data and apply a binary search or simply apply the less-efficient linear search. Is the cost of sorting the data is worth the increase in search speed gained with the binary search? If you are searching only once, then it is probably to better do a linear search in most cases.

## Check Your Progress 2

- 1) State True or False
  - a. The order of linear search in worst case is  $O(n^2)$  True/False
  - b. Linear search is more efficient than Binary search. True/False
  - c. For Binary search, the array has to be sorted in ascending order only. True/False
- 2) Write the Binary search algorithm where the array is sorted in descending order.

## 9.4 APPLICATIONS

The searching techniques are applicable to a number of places in today's world, may it be Internet, search engines, on line enquiry, text pattern matching, finding a record from database, etc.

The most important application of searching is to track a particular record from a large file, efficiently and faster.

Let us discuss some of the *applications of Searching* in the world of computers.

### 1. Spell Checker

This application is generally used in **Word Processors**. It is based on a program for checking spelling, which it checks and searches sequentially. That is, it uses the concept of *Linear Search*. The program looks up a word in a list of words from a dictionary. Any word that is found in the list is assumed to be spelled correctly. Any word that isn't found is assumed to be spelled wrong.

### 2. Search Engines

Search engines use software robots to survey the Web and build their databases. Web documents are retrieved and indexed using keywords. When you enter a query at a search engine website, your input is checked against the search engine's keyword indices. The best matches are then returned to you as hits. For checking, it uses any of the Search algorithms.

Search Engines use software programs known as robots, spiders or crawlers. A robot is a piece of software that automatically follows hyperlinks from one document to the next around the Web. When a robot discovers a new site, it sends information back to its main site to be indexed. Because Web documents are one of the least static forms of publishing (i.e., they change a lot), robots also update previously catalogued sites. How quickly and comprehensively they carry out these tasks vary from one search engine to the next.



### 3. String Pattern matching

Document processing is rapidly becoming one of the dominant functions of computers. Computers are used to edit, search and transport documents over the Internet, and to display documents on printers and computer screens. Web ‘surfing’ and Web searching are becoming significant and important computer applications, and many of the key computations in all of this document processing involves character strings and string pattern matching. For example, the Internet document formats HTML and XML are primarily text formats, with added tags for multimedia content. Making sense of the many terabytes of information on the Internet requires a considerable amount of text processing. This is accomplished using *trie* data structure, which is a tree-based structure that allows for faster searching in a collection of strings.

---

## 9.5 SUMMARY

---

Searching is the process of looking for something. Searching a list consisting of 100000 elements is not the same as searching a list consisting of 10 elements. We discussed two searching techniques in this unit namely Linear Search and Binary Search. Linear Search will directly search for the key value in the given list. Binary search will directly search for the key value in the given sorted list. So, the major difference is the way the given list is presented. Binary search is efficient in most of the cases. Though, it had the overhead that the list should be sorted before search can start, it is very well compensated through the time (which is very less when compared to linear search) it takes to search. There are a large number of applications of Searching out of whom a few were discussed in this unit.

---

## 9.6 SOLUTIONS / ANSWERS

---

### Check Your Progress 1

- 1) No
- 2) It will be located at the beginning of the list

### Check Your Progress 2

- 1) (a) F  
(b) F  
(c) F

---

## 9.7 FURTHER READINGS

---

### Reference Books

1. *Fundamentals of Data Structures in C++* by E. Horowitz, Sahai and D. Mehta, Galgotia Publications.
2. *Data Structures using C and C ++* by Yedidyah Hangsam, Moshe J. Augenstein and Aaron M. Tanenbaum, PHI Publications.
3. *Fundamentals of Data Structures in C* by R.B. Patel, PHI Publications.

### Reference Websites

[http:// www.cs.umbc.edu](http://www.cs.umbc.edu)  
<http://www.fredosaurus.com>

