5 UNIT V SEARCHING, SORTING AND HASHING

Searching – Linear Search – Binary Search. Sorting – Selection sort – Insertion sort – Quick sort – Merge Sort – Hashing – Hash Functions – Separate Chaining – Open Addressing. Case Study: Pacemaker data buffering, Data structures in firmware for wearable health devices, DSP filter coefficient handling using arrays and lists.

5.1 Searching

Searching is the process of finding the **position of a specific element** (key) in a list, array, or data structure. Searching is very important because in many applications, we need to locate data quickly.

Searching Techniques

To search an element in a given array, it can be done in following ways:

1. Sequential(Linear) Search
2. Binary Search

5.1.1 Linear Search

- Sequential search is also called as Linear Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- Elements are searched one by one from the beginning.
- Works on **unsorted arrays**.
- Simple and easy to implement.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements

given in the list. If the element is matched, it returns the value index, else it returns -1.

A simple approach is to do linear search, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[] If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

Example: Consider the following list of elements and the element to be searched is 12.

search element 12

Step 1:

search element (12) is compared with first element (65)

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

Program

```
#include <iostream.h>
#include <conio.h>
void main()
{
    int a[50], n, key, i;
    cout << "Enter number of elements: ";
    cin >> n;
    cout << "Enter elements of array:\n";
    for(i = 0; i < n; i++)
        cin >> a[i];
    cout << "Enter element to search: ";
    cin >> key;

    // Search for key
    for(i = 0; i < n; i++)
    {
        if(a[i] == key)
        {
            cout << "Element found at position " << i + 1;
            getch();
            return; // Exit program after finding element
        }
```

```
    }
    cout << "Element not found";
    getch();
}
```

Advantages:

- Simple to understand and implement.
- Works on unsorted data.
- No extra memory required.

Disadvantages:

- Slow for large arrays (O(n) time).

Applications:

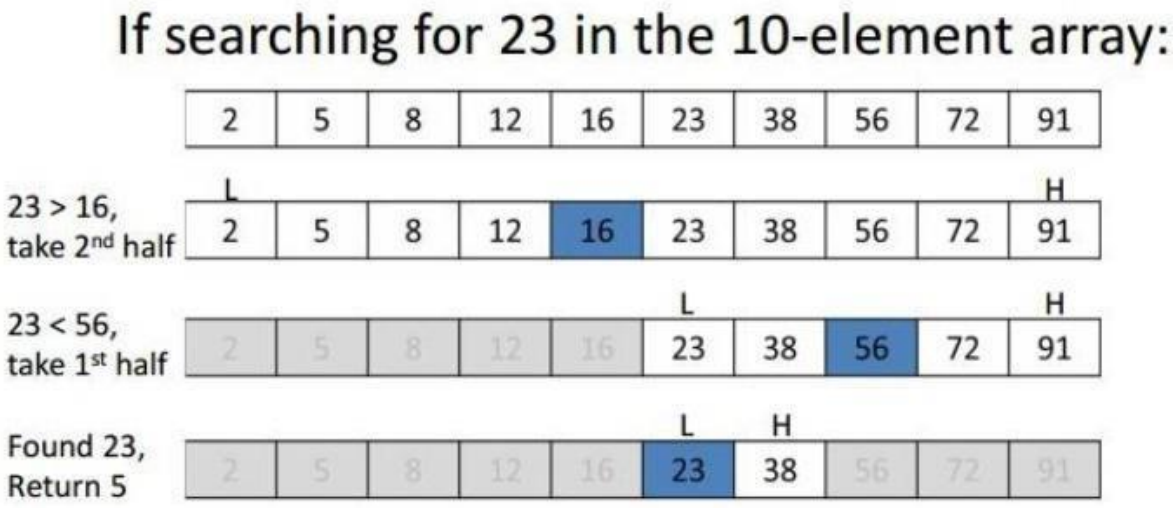
1. Searching elements in small data sets
2. Searching in unsorted lists or arrays
3. Finding a record in a sequential file
4. Searching elements in arrays and linked lists
5. Checking whether an element exists or not
6. Searching in dynamic data where frequent updates occur
7. Searching when no indexing technique is available

5.1.2 BINARY SEARCH

Definition:

Binary search works on sorted arrays**. It repeatedly divides the array in half to locate the element.**

- Search a sorted array by repeatedly dividing the search interval in half.
- Begin with an interval covering the whole array.
- If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise narrow it to the upper half.



- Repeatedly check until the value is found or the interval is empty.

Algorithm

- Compare x with the middle element
- If x matches with middle element, we return the mid index.
- Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
- Else (x is smaller) recur for the left half.

Program

int binarySearch(int arr[], int l, int r, int x)

```
{
if (r >= l)
{
        int mid = l + (r - l)/2; if (arr[mid] == x)
              return mid; if (x<arr[mid])
                 return binarySearch(arr, l, mid-1, x);
        }else


}
}
return binarySearch(arr, mid+1, r, x);
```

Advantages:

- Very fast (O(log n) time).
- Efficient for large data.

Disadvantages:

- Only works on sorted arrays.

Applications:

1. Searching elements in **sorted arrays or lists**
2. Finding data in **large databases** with sorted records
3. Searching in **dictionaries and phone directories**
4. Efficient searching in **static data sets**
5. Finding an element's **position or index** in a sorted list
6. Used in **library management systems** for book search
7. Searching in **computer memory** where data is stored in sorted order.