| | |
|---|---|
| **Assignment no. 4** | 4 |
| **Aim** | **a) Write a Python program to store roll numbers of student in array who attended training program in random order. Write function for searching whether particular student attended training program or not, using Linear search and Sentinel search.**<br>**b) Write a Python program to store roll numbers of student array who attended training program in sorted order. Write function for searching whether particular student attended training program or not, using Binary search and Fibonacci search.** |
| **Objective** | To understand the concept of searching method<br>To find the performance of searching method<br>To apply searching method to check whether an element is present in the given array/list. |
| **Outcome** | To design and implement different searching methods<br>To calculate time and space complexity of different searching method<br>To apply functions of different searching method on given data for checking the presence of a particular element. |
| **OS/Programming tools used** | (64-Bit) 64-BIT Fedora 17 or latest 64-BIT Update of Equivalent Open-source OS or latest 64-BIT Version and update of Microsoft Windows 7 Operating System onwards Programming Tools (64-Bit)<br>Eclipse with Python plugin or Pycharm IDE |

**Theory related to assignment:**

In this assignment we will implement the different searching techniques.

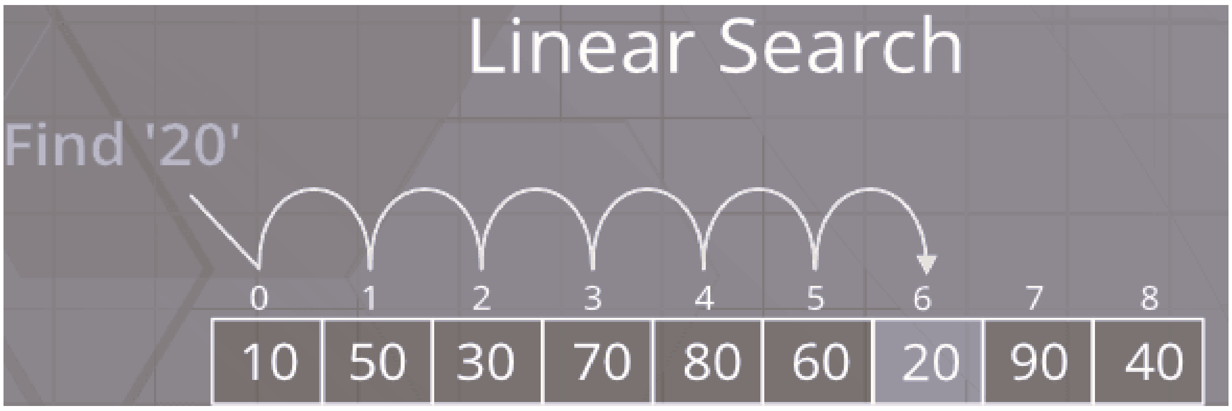
For this purpose, we need to accept the student's data in array or list. Thereafter, apply the following searching operations to search an element within an array or list.

The program should display the position of an element if it is found in input list or display not found message.

**Linear Search:**

A **linear search** or **sequential search** is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched. A linear search runs in at worst linear time and makes at most *n* comparisons, where *n* is the length of the list. If each element is equally likely to be searched, then linear search has an average case of (n+1)/2 comparisons, but the average case can be affected if the search probabilities for each element vary.

**Example of linear search:**



**Steps to perform linear search**

Linear Search (Array A, Value x)

```
Step 1: Set i to 0
Step 2: if i = n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

**Pseudocode for Linear search:**

```
int search(arr[], int count, int key)

1. found := false
2. position := –1, index := 0
3. while (index < count and !found)
4.         If (key == arr[index]) then
5.                 found := true
6.                 position := index
7.                 break;
8.         End If
9.          index:=index + 1
10. end while
11. return position

end search
```

**Time Complexity:**

Best Case – O(1)
Worst Case – O(n)
Average Case – O(n)

**Space Complexity:** O(1)

**Sentinel Search or Sentinel Linear Search:**

Sentinel Linear Search as the name suggests is a type of Linear Search where the number of comparisons is reduced as compared to a traditional linear search. When a linear search is performed on an array of size N then in the worst case a total of N comparisons are made when the element to be searched is compared to all the elements of the array and (N + 1) comparisons are made for the index of the element to be compared so that the index is not out of bounds of the array which can be reduced in a Sentinel Linear Search.

**Pseudocode Sentinel Search**

```
procedure Sentinel_Search(int arr[], int n, int x)

    1.  int last := arr[n - 1];     // Last element of the array
    2.  arr[n - 1] := x;       // Element to be searched is placed at last index
    3.  int i = 0;
    4.  while (arr[i] != x)
    5.          i++;
    6.  arr[n - 1] := last;       // Put the last element back
    7.  if ((i < n - 1) || (x == arr[n - 1]))
    8.          cout << x << " is present at index " << i;
    9.  else
    10.         cout << "Not found“

end Sentinel_search
```

**Example of sentinel search**

Consider the following array of elements

| | | | | |
|---|---|---|---|---|
| 5 | 3 | 6 | -12 | 20 |

We have to find the element 6. The first step would be to assign the last element to the int last and place the element to be found at last index position.

i.e. int last=20 and arr[4]=6. Our array would now look like this.

| | | | | |
|---|---|---|---|---|
| 5 | 3 | 6 | -12 | 6 |

Now traverse though the entire array from the beginning to search for the element 6 just like linear search.

| | | | | |
|---|---|---|---|---|
| 5 | 3 | 6 | -12 | 6 |

| | | | | |
|---|---|---|---|---|
| 5 | 3 | 6 | -12 | 6 |

You will find the element at arr[2]

| | | | | |
|---|---|---|---|---|
| 5 | 3 | 6 | -12 | 6 |

As your element was found before the last index position, it implies that it was initially present in the array at that particular position (Refer to step 7 of pseudocode).

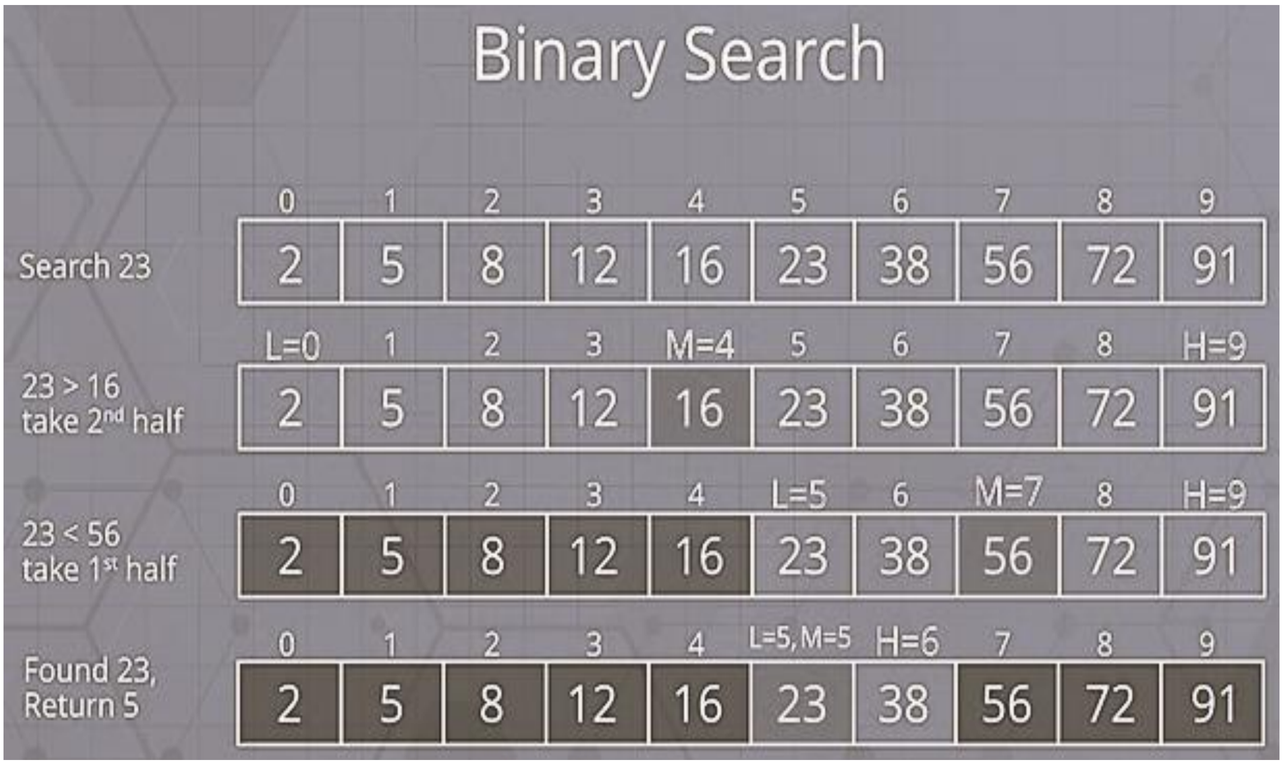
**Time complexity**

| | **Linear Search** | **Sentinel Search** |
|---|---|---|
| **Best Case** | **O(1) – two comparisons** | **O(1) – three comparisons** |
| **Worst Case** | **O(n) – 2N comparisons** | **O(n) – N+2 comparisons** |
| **Average Case** | **O(n) - 2N comparisons** | **O(n) – N+2 comparisons** |
| **Space complexity** | **O(1)** | **O(1)** |

**Binary Search**

- Search a sorted array by repeatedly dividing the search interval in half.
- It begins 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.
- If the value of the search key is greater than the item in the middle of the interval, narrow it to the upper half.
- Repeatedly check until the value is found or the interval is empty.
- We ignore half of the elements just after one comparison.
- Upper bound of Loop is decreasing by n/2 after each comparison.
- Time complexity is O(log n).

Consider the following example:



Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Binary Search Algorithm can be implemented in two ways:

1. Iterative Method
2. Recursive Method

The procedure for implementation of binary search using iterative method is as follows:

**Pseudocode for Binary Search**

```
procedure binarySearch(int arr[], int s, int e, int x)

        {

        // Precondition :- arr should be sorted array
        // Postcondition :- will return position of x if present otherwise -1
    1.  while (s<= e)
    2.  {
    3.  m :=floor((s+e) / 2);            //find mid element

    4.  if (arr[m] == x)                 // Check if x is present at mid
    5.          return m;
    6.  if (arr[m] < x)                  // If x greater, ignore left half
    7.          s := m + 1;
    8.  else
    9.          e := m - 1;              // If x is smaller, ignore right half
    10. }
    11.  return -1;              // if we reach here, then element was not present
        }
end binarySearch
```

The following is the procedure for implementation of binary search using recursive method:

```
Procedure binarySearch(int arr[], int s, int e, int x)

{

Precondition :- arr should be sorted array

Postcondition :- will return position of x if present otherwise -1

    1.  if(s<=e)
    2.  {
    3.  int m :=floor((s+e) / 2);
    4.  if (arr[m] == x)                 // Check if x is present at mid
    5.  return m;
    6.  if (arr[m] < x)                  // If x greater, ignore left half
    7.  s := m + 1;
    8.  return(binarySearch(arr,s,e,x));
    9.  else
    10. e := m - 1;              // If x is smaller, ignore right half
    11. return(binarySearch(arr,s,e,x));
    12. }
    13. else
    14. return -1;                    // if we reach here, then element was not present
    15. }
```

end binarySearch

**Time complexity**

- Best case – O(1)
- Average, worst case - O(log n)

**Space complexity**

- Best case – O(1)
- Average, worst case - O(log n)

**Fibonacci Search**

- Fibonacci Search is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.
- It is very much similar to binary search
- Works on sorted arrays
- A Divide and Conquer Algorithm
- Time complexity is O(log n)
- Binary search divides given array at mid but Fibonacci search divides in unequal parts
- Binary Search uses division operator to divide range. Fibonacci Search doesn't use /, but uses + and -. The division operator may be costly on some CPUs.
- Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.
- It is applied on nonlinear unimodal function

**Pseudocode for Fibonacci Search**

Procedure Fibonacci_Search(arr[ ], x, N):

```
1.   m = 0
2.   while Fibo(m) < N              // repeat till mth Fibonacci no. is less than N
3.       m = m + 1
4.   offset = -1
5.   while (Fibo(m) > 1)
6.       mid = min( offset + Fibo(m - 2) , N - 1)
7.       if (x > arr[mid])
8.           m = m - 1
9.           offset = mid
10.      elif (x < arr[mid])
11.          m = m - 2
12.      else
13.          return mid
14.  end while
15.  if(!Fibo(m - 1) and arr[offset + 1] == x)
16.      return offset + 1
17.  return -1
     End Fibonacci_Search
```

**Example of Fibonacci search**

Search 70 in the following array.

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

Size=10 Fm=13  Fm-1=8

70<90. So, take first half.

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

70>60. So, take second half.

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

Size=8 Fm=8  Fm-1=5

Arr[0+5]=Arr[5]

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

70<80. Take first half.

Size=2 Fm=2  Fm-1=1

Arr[6+1]=Arr[7]

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

70 found.

Size=1 Fm=1  Fm-1=0

Arr[6+0]=Arr[6]

**Time complexity**

Best case – O(1)
Average, worst case - O(log n)

**Space complexity**

Best case – O(1)
Average, worst case - O(log n)

**Conclusion:**

The functions for different searching methods are implemented successfully on student data with efficient time and space complexity.

**Review Questions:**

1. What are the various applications of linear search?
2. When to use linear search?
3. Can linear search be done on Linked Lists?
4. What is the best-case time complexity for Linear Search?

5. Which search algorithm is best if data keeps changing frequently?
6. Can linear search be made parallel for execution on multiple CPU cores?
7. What is the time complexity of Linear Search for string data?
8. What is the time complexity of Linear Search for Integer data?
9. Why is the worst-case time complexity same as average case for Linear Search?
10. Compare Binary Search vs Linear Search
11. Explain how does the Sentinel Search work?
12. Is Sentinel Linear Search better than normal Linear Search?
13. Explain why complexity of Binary Search is O(log n)?
14. What is the time complexity of Fibonacci Search?