Data Structures (15B11CI311)

Odd Semester 2020

3^{rd} Semester , Computer Science and Engineering

Jaypee Institute Of Information Technology (JIIT), Noida

Outline

Different Searching Techniques:

- Linear Search
- Analysis of time and space complexity of linear search
- Binary Search
- Analysis of time and space complexity of binary search

Linear Search

- Searching is useful to find any record stored in the file.
- Searching books in library

For that we have two searching algorithms:

- Linear Search/Sequential Search
- Binary Search

Input: Array **A** of **n** elements

Output: Return index/position of the element **X** in the array **A**

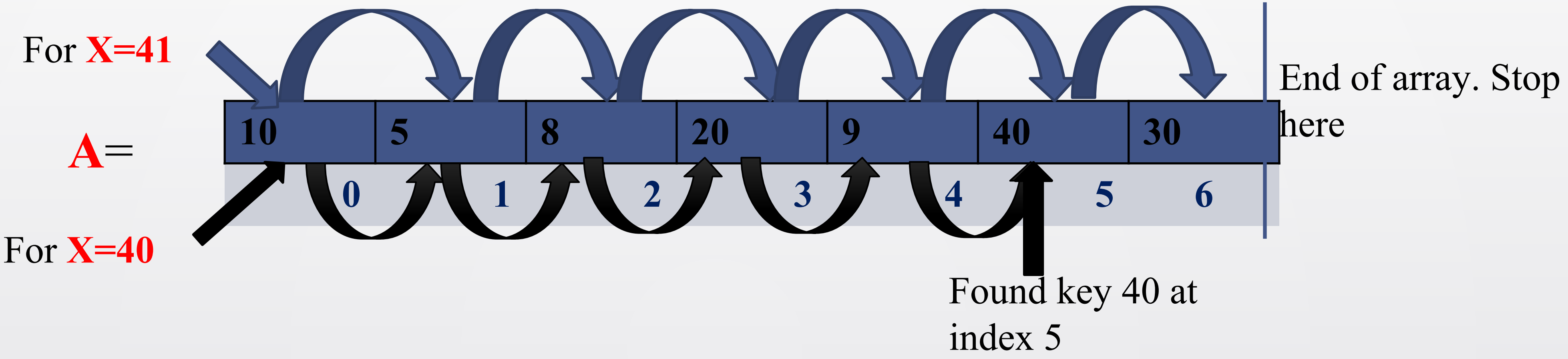
Two Cases:

Either element is present : Linear Search Gives the Index

Or elements not present: Elements not found

ILLUSTRATION OF LINEAR SEARCH

Find (Search-key) X=40



There are a possibility of two cases: Suppose search-key is X

Case 1: X is present in Array A. X=40

Linear Search gives the location index of element 40

Location/Index= 5+1= 6

Case II: X not present in Array A. X=41

Linear Search gives the message that **Element not Found in the given Array**.

Iterative Implementation of Linear Search

```
#include <iostream>
using namespace std;
 int Linear_search(int A[], int n, int x)
{
   int i;
   for (i = 0; i < n; i++)
       if (A[i] == x)
           return i;
   return -1;
}
```

```
int main(void)
{
   int A[] = { 2, 3, 4, 10, 40 };
   int x = 10;
   int n = sizeof(arr) / sizeof(arr[0]);
   int result = Linear_search(A, n, x);
  if (result == -1)
      cout<<"Element is not present in array"
else
     cout<<"Element is present at index " <<result;
  return 0;
}
```

Recursive Linear Search

Recursive implementation of linear search

```
int RLinear_search(int A[], int l, int r, int x)
{
    if (r < l)
      return -1;
    if (A[l] == x)
      return l;
    return RLinear_search(A, l+1, r, x);
}
```

l=lower (left) index of array that is 0.
r=higher(right) index of array A that is r=n-1 (number of element)
x= element to be search
RLinear_search will be called from main function. Code for main function same as in non-iterative code. Just replace the **Linear_search**(A, n, x) by **RLinear_search** (A, l, r, x)

Time and Space Complexity of Linear Search

- **Best case** what is the minimum number of comparisons that can be done for n items = comparisons: 1
  - Occurs when x is the first element examined

- **Worst case** what is the maximum number of comparisons for n items= comparisons: n
  - Case 1: x is the last element examine
  - Case 2: x is not in the list

- **Average case** on average, how many comparisons do you expect the algorithm to do
  - This is bit tougher because it depends on
    - The order of the elements list
    - The probability that x is in the list at all

Linear Search

- **Worst Case :**
  - x is the last element examine or x is not in the list

  In either case we have $T(n)=n$

- **Average Case :**

  We assume that x appears in an array and it is equally likely to occur at any position in the array with probability 1/n

  $T(n)=1.1/n+2.1/n + 3.1/n + \ldots\ldots + n.1/n$

  $=(1+2+3+\ldots..+ n).1/n$

  $= n(n+1)/2 * 1/n=(n+1)/2$

Binary Search

- Binary search works on the sorted array only.
- If you have sorted array then go for binary search than linear search.
- In the binary search, searching start from mid of the array and check for the element if it at mid or not.
- If element not present at the mid then array is divided into two part from mid called left sub-array and right sub-array
- Now check element is smaller or greater than mid element. If it is smaller than mid element then check in left sub-array and if greater than mid element then check in right-subarray.
- Repeat the process until element found or process end.

Binary Search: Pseudo Code

- The method is recursive:
- Compare X with the middle value A[mid].
- If X =A[mid], return mid
- If X < A[mid], then X can only be in the left half of A[ ], because A[ ] is sorted. So call the function recursively on the left half.
- If X > A[mid], then X can only be in the right half of A[ ], because A[ ] is sorted. So call the function recursively on the right half.

Illustration of Binary search

A[12]:

3	5	8	13	15	18	25	29	35	45	47	70
0	1	2	3	4	5	6	7	8	9	10	11

Search for X=13

mid = (0+11)/2 = 5. Compare X with A[5]: 13<18.

So search in left half X[0..4]

3	5	8	13	15	18	25	29	35	45	47	70
0	1	2	3	4	5	6	7	8	9	10	11

mid = (0+4)/2 = 2. Compare X with A[2]: 13> 8.

.

Illustration of Binary search

So search right half A[3..4]

3	5	8	13	15	18	25	29	35	45	47	70
0	1	2	3	4	5	6	7	8	9	10	11

mid = (3+4)/2 = 3.Compare X with A[3]: X=X[3]=13.

- Return 3.

3	5	8	13	15	18	25	29	35	45	47	70
0	1	2	3	4	5	6	7	8	9	10	11

Iterative Code of Binary Search

```
// program to implement recursive Binary Search
#include <iostream>
using namespace std;
int binary_search(int A[], int l, int r, int x)
{
        while (l <= r) {
                int mid = l + (r - l) / 2;
                if (A[mid] == x)
                        return mid;
                if (A[mid] < x)     // If x greater, ignore left half
                        l = mid + 1;
                else                        // If x is smaller, ignore right half
                    r = mid - 1;
        }
return -1; // if we reach here, then element was   not present
}
```

```
int main(void)
{
     int A[10];
     int x = 35;
     int n =10;
    printf("Enter the sorted array");
    for(int i=0;i<n;i++)
    scanf("%d",&A[i]);
     int result = binarySearch(A, 0, n - 1, x);
     if(result == -1)
    cout << "Element found in the array"
    else
     cout << "Element not found" << result;
     return 0;
}
```

The Recursive Code of Binary Search

```
int binary_search(int X, int A[], int left, int right)
{
    if (left == right)
        if (X==A[left]) return left;
        else return -1;
    int mid = (left+right)/2;
    if (X==A[mid]) return mid;
    if (X < A[mid]) return binarySearch (X, A, left, mid-1);
    if (X >A[mid]) return binarySearch(X, A, mid+1, right);
}
```

Time Complexity of Binary Search

- Count the number of visits to search an sorted array of size *n*
  - We visit one element (the middle element) then search either the left or right subarray. Always array is divided into two part. The recurrence relation for Binary Search :

$$T(n) = T(n/2) + 1$$

- If n is $n/2$, then $T(n/2) = T(n/4) + 1$

Substituting into the original equation:

$T(n) =(T(n/4)+1) + 1= T(n/4)+2$

now if n=n/4 then

T(n)=T(n/8)+3=T(n/2^3)+3

- This generalizes to: $T(n) = T(n/2^k) + k$

  when there is only one element the T(1)=1

Assume

$n/2^k = 1$

$n = 2^k$

Taking log both side and solving it:

$k = \log n$

- Then: $T(n) = T(1) + \log_2(n) = O(\log(n))$

Binary search is an $O(\log(n))$ algorithm

Apply binary search on the following array. Search the key 67 and 114 in the array.

6	**8**	**11**	**15**	**67**	**80**	**101**	**115**	**118**	**120**

- *https://www.geeksforgeeks.org/linear-search/*
- *https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm*