Experiment : 1a

Date : 24/01/2024

Title : Linear Search

Aim : To implement and analyze linear search algorithm.

Algorithm :

Step 1 : Start

Step 2 : Prompt user for array size, read input, and declare an array of that size.

Step 3 : Use a loop to input each array element from the user

Step 4 : Prompt user for the target element and read input.

Step 5 : Iterate through the array, compare each element with the target.

Step 6 : If target found, print the index; otherwise, print a not found message.

Step 7 : End

Program Implementation :

```
# include <stdio.h>
int linearSearch (int arr [ ], int n, int target ) {
    for (int i=0 ; i<n ; i++) {
        if (arr [i] = = target) {
            return i ; //Return the index if the target is found
        }
    }
    return -1; // Return -1 if the target is not found
}
int main(){
    int size ;
    printf ( "Enter the size of the array : ");
    Scan f ("%d", & size );
    int array [size] ; //Declare an array of the specified size
```

```
printf ("Enter %d elements for the array :\n", size);
for (int i=0; i< size ;i++){
    printf ("Element %d : ", i+1);
    scanf ("%d", & array [i]);
}
int target ;
printf ("Enter the target element to search :");
Scanf ("%d", & target) ;
int result = linearSearch (array , size , target);
if (result != -1){
    printf ("Element %d found at index %d \n", target, result);
} else {
    printf("Element %d not found in the array \n", target);
}
return 0;
}
```

Time Complexity Analysis :

The time complexity of the provided linear Search code is O(n), where "n" is the size of the array.
The loop for inputting array elements runs in O(n) time , where "n" is the size of the array .
The linear search function iterates through the array once (in the worst case) to find the target element .
In the worst case, it may have to check all "n" elements .
Therefore, the linear search has a time complexity of O(n).
The overall time complexity is dominated by the linear search , making the entire algorithm O(n). The efficiency of the linear search algorithm is linearly proportional to the size of the array .

Dry run with sample input output:

Input:

Enter the size of the array : 5
Enter the 5 elements for the array :
Element 1 : 8
Element 2 : 3
Element 3 : 12
Element 4 : 5
Element 5 : 7

Enter the target element to search : 12

Output:

Element 12 found at index 2

Result:

Therefore, linear search algorithm was implemented and anlayzed successfully.

Experiment : 1b Title : Binary Search

Aim : To implement and analyze bineary search algorithm.

Algorithm:

Step 1 : Start

Step 2 : Prompt user for the size of the array.

Step 3 : Read input, declare an array, and prompt user to input sorted elements.

Step 4 : Initialize low to 0 and high to size - 1.

Step 5 : While low < = high, calculate mid and compare arr[mid] with the target element, updating low and high accordingly.

Step 6 : If the target element is found, print its index; otherwise, print a not found message.

Step 7 : End

Program Implementation :

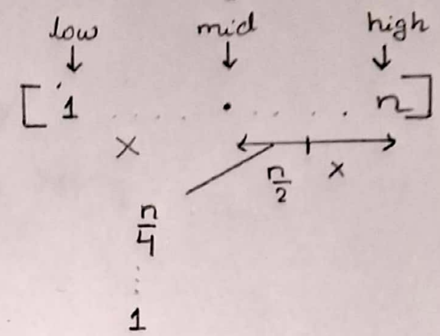
```
#include <stdio.h>

int binarySearch (int arr [], int size, int element) {
    int low, mid, high;
    low = 0;
    high = size - 1;
    while (low <= high){
        mid = (low + high) / 2;
        if (arr [mid] == element) {
            return mid;
        }
        if (arr [mid] < element) {
            low = mid + 1;
        } else {
            high = mid-1;
        }
```

```
    }
    return -1;
}
int main() {
    int size;
    printf("Enter the size of the array: ");
    scanf("%d", &size);
    int arr[size];
    printf("Enter %d sorted elements for the array:\n",
            size);
    for (int i=0; i< size; i++) {
        printf("Element %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    int element;
    printf("Enter the target element to search:");
    scanf("%d", &element);
    int searchIndex = binarySearch(arr, size, element);
    if (searchIndex != -1) {
        printf("The element %d was found at index %d \n",
                element, searchIndex);
    }else {
        printf("The element %d was not found in the
        array \n", element);
    }
    return 0;
}
```

Time Complexity Analysis :

$$mid = \frac{low + high}{2}$$

```
if (a[mid] > key)
    BS(a[n], key, low, mid-1)
else
    BS(a[n], key, mid+1, high)
return -1
```



Here, we get a reccurrence relation as :

$$T(n) = \begin{cases} T(n/2) + C & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

$$T(n) = T(n/2) + C \quad \text{--- (1)}$$

$$T(n/2) = T(n/4) + C \quad \text{--- (2)}$$

$$T(n/4) = T(n/8) + C \quad \text{--- (3)}$$

$$T(n) = T(n/4) + C + C = T(n/2^2) + 2C$$

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

$$= T(n/2^4) + 4C$$

$$= T(n/2^5) + 5C$$

$$\vdots$$

k times

$$= T(n/2^k) + kC$$

$$n = 2^k$$

$$\log n = \log 2^k$$

$$\log n = k$$

$$\therefore T(n) = T(n/2^k) + kC$$

$$= T(n/n) + kC$$

$$= T(1) + kC$$

$$= 1 + kC$$

$$= 1 + \log n \, C$$

$$O(\log_2 n)$$

So, the time complexity of the provided binary search code is $O(\log_2 n)$, where "n" is the size of the array.

Dry run with sample input and output :

Sample Input :

Enter the size of the array : 8
Enter 8 sorted elements for the array :

Element 1 : 10
Element 2 : 20
Element 3 : 30
Element 4 : 40
Element 5 : 50
Element 6 : 60
Element 7 : 70
Element 8 : 80

Enter the target element to search : 50

Sample Output :

The element 50 was found at index 4.

Result :

Therefore, binary search algorithm was implemented and analyzed successfully.