Experiment No.10
Implement Binary Search Algorithm.
Name: Gautam D. Chaudhari
Roll No: 04
Date of Performance:
Date of Submission:
Marks:
Sign:

Experiment No. 10: Binary Search Implementation.

Aim : Implementation of Binary Search Tree ADT using Linked List.

Objective:

1) Understand how to implement a BST using a predefined BST ADT.

2) Understand the method of counting the number of nodes of a binary tree.

Theory:

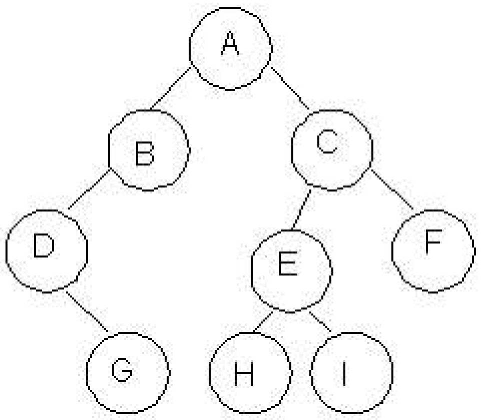
A binary tree is a finite set of elements that is either empty or partitioned into disjoint subsets. In other words nodes in a binary tree have at most two children and each child node is referred to as left or right child.

Traversals in trees can be in one of the three ways: preorder, postorder, inorder.

Preorder Traversal

Here the following strategy is followed in sequence

1. Visit the root node R
2. Traverse the left subtree of R
3. Traverse the right subtree of R



Description	Output
Visit Root	A
Traverse left sub tree – step to B then D	ABD
Traverse right subtree – step to G	ABDG
As left subtree is over. Visit root , which is already visited so go for right subtree	ABDGC
Traverse the left subtree	ABDGCEH
Traverse the right sub tree	ABDGCEHIF

Inorder Traversal

Here the following strategy is followed in sequence

1. Traverse the left subtree of R
2. Visit the root node R
3. Traverse the right sub tree of R

Description	Output
Start with root and traverse left sub tree from A-B-D	D
As D doesn't have left child visit D and go for right subtree of D which is G so visit this.	DG
Backtrack to D and then to B and visit it.	DGB
Backtrack to A and visit it	DGBA
Start with right sub tree from C-E-H and visit H	DGBAH
Now traverse through parent of H which is E and then I	DGBAHEI
Backtrack to C and visit it and then right subtree of E which is F	DGBAHEICF

Postorder Traversal

Here the following strategy is followed in sequence

1. Traverse the left subtree of R
2. Traverse the right sub tree of R
3. Visit the root node R

Description	Output
Start with left sub tree from A-B-D and then traverse right sub tree to get G	G
Now Backtrack to D and visit it then to B and visit it.	GD
Now as the left sub tree is over go for right sub tree	GDB
In right sub tree start with leftmost child to visit H followed by I	GDBHI
Visit its root as E and then go for right sibling of C as F	GDBHIEF
Traverse its root as C	GDBHIEFC
Finally a root of tree as A	GDBHIEFCA

Algorithm

Algorithm: PREORDER(ROOT)

Algorithm :

```
Function Pre-order( root )
    - Start
    - If root is not null then
Display the data in root
Call pre order with left pointer of root(root -> left)
Call pre order with right pointer of root(root -> right)
    - Stop
```

Algorithm: INORDER(ROOT)

```
Algorithm :
Function in-order( root )
    - Start
    - If root is not null then
Call in order with left pointer of root (root -> left )
Display the data in root
Call in order with right pointer of root(root -> right )
- Stop
```

Algorithm: POSTORDER(ROOT)

```
Algorithm :
Function post-order ( root )
    - Start
    - If root is not null then
Call post order with left pointer of root (root -> left)
Call post order with right pointer of root (root -> right)
Display the data in root
    - Stop
```

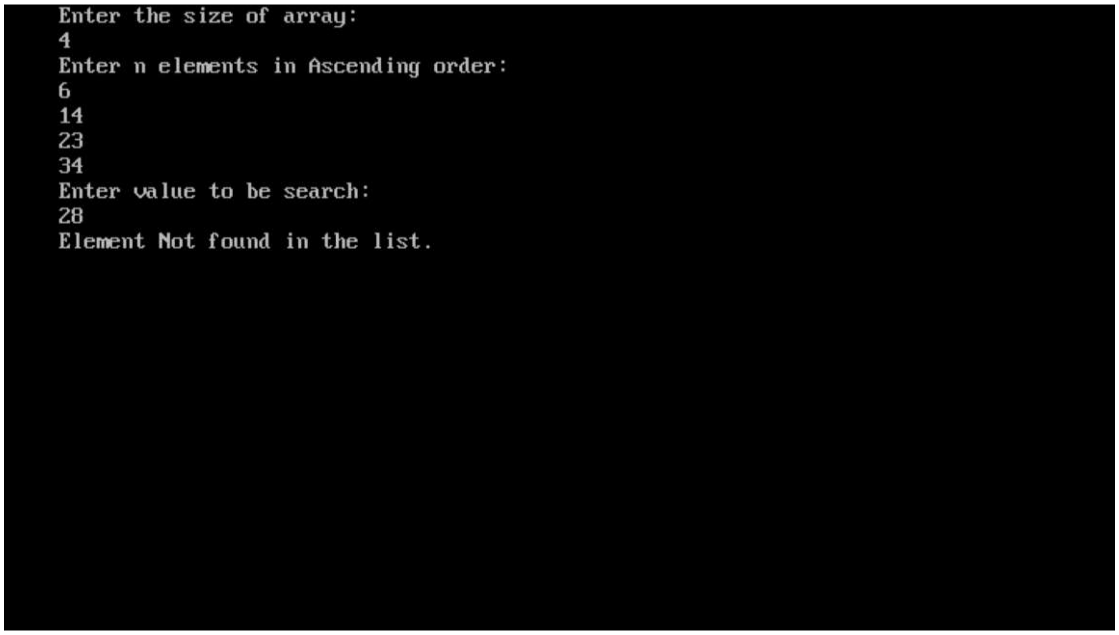
Code:

```
 #include <stdio.h>

#include <conio.h>



int main()

{

int first, last, middle, n, i, find, a[100]; setbuf(stdout, NULL);

clrscr();

printf("Enter the size of array: \n");

scanf("%d",&n);
```

```
printf("Enter n elements in Ascending order: \n");

for (i=0; i < n; i++)

scanf("%d",&a[1]);


printf("Enter value to be search: \n");

 scanf("%d", &find);

first=0;

last=n - 1;

middle=(first+last)/2;

while (first <= last)

{

if (a[middle]<find)

{

 first=middle+1;

}

else if (a[middle]==find)

{

printf("Element found at index %d.\n",middle);

 break;

}

else

{

last=middle-1;

middle=(first+last)/2;

}

}

if (first > last)
```

```
printf("Element Not found in the list.");
 getch();
 return 0;
}
```

Output:



```
Enter the size of array:
4
Enter n elements in Ascending order:
6
14
23
34
Enter value to be search:
28
Element Not found in the list.
```

Conclusion:

1.Describe a situation where binary search is significantly more efficient than linear search.

Binary search is significantly more efficient than linear search in situations where you have a large sorted dataset and need to find a specific element. Here's why:
Scenario: Imagine you have a phone book with thousands of names sorted in alphabetical order by last name. You need to find the phone number of a person with a specific last name, "Smith."

Linear Search: If you were to use a linear search, you would start from the first entry and go through the phone book one page at a time, checking each name until you find "Smith." This can take a considerable amount of time if "Smith" is near the end of the phone book.
Binary Search: In contrast, with binary search, you can start in the middle of the phone book. Since it's sorted, you can quickly determine if "Smith" would be in the first or second half. You repeat this process, effectively halving the search space with each step. This method allows you to find "Smith" much faster, particularly in large datasets.

2.Explain the concept of "binary search tree." How is it related to binary search, and what are its applications?
A Binary Search Tree (BST) is a hierarchical data structure used for organizing and storing a set of elements, such as numbers, in a way that supports efficient searching, insertion, and deletion operations. It is related to binary search in the sense that it leverages the binary search algorithm to perform these operations.
Binary Search Trees (BSTs) share a fundamental relationship with the Binary Search algorithm, which is evident in their design and purpose. Both are rooted in the efficient retrieval of data. Binary Search operates on sorted arrays, dividing the search space in half with each comparison. Similarly, BSTs are tree structures where each node is organized according to the binary search property, ensuring that left subtrees contain smaller elements, and right subtrees hold larger ones. This structural similarity enables BSTs to provide fast searching, insertion, and deletion operations with a time complexity of O(log n) for balanced trees. Therefore, the relationship between BSTs and Binary Search lies in their common objective of optimizing data retrieval through divide-and-conquer techniques.