Experiment No.10
Implement Binary Search Algorithm.
Name: AARYAN CHANDRAKANT GOLE
Roll No: 12
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.

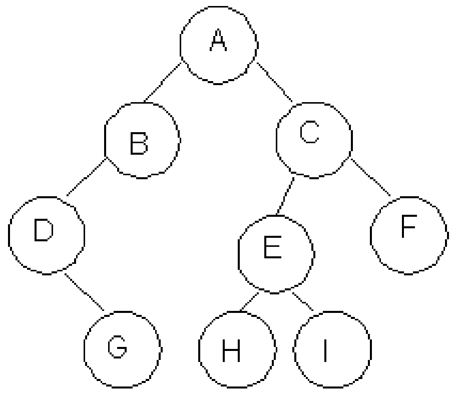
Theory:

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	ABDGCEHI F

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)

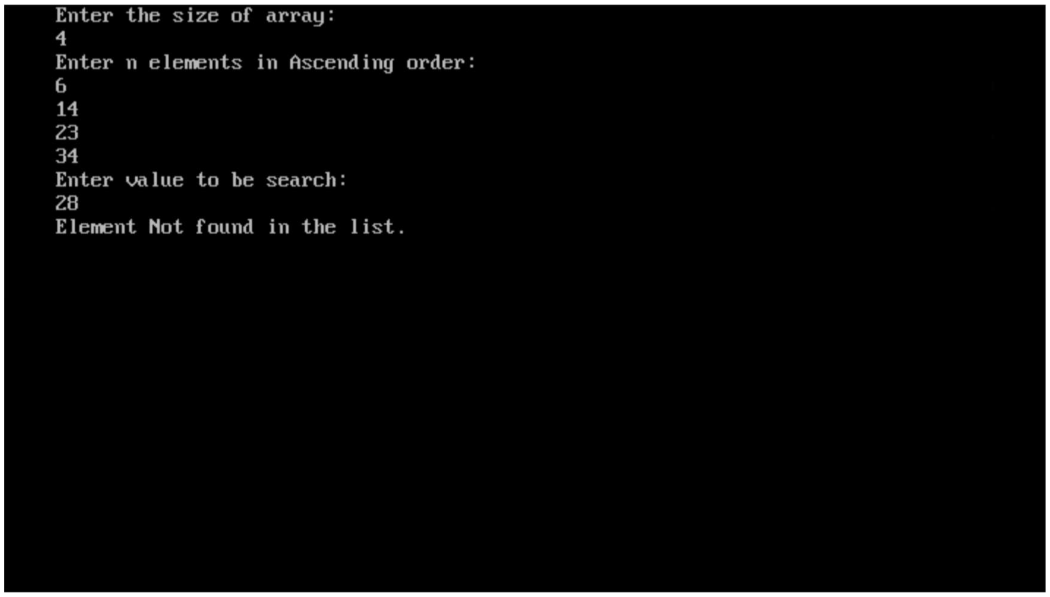
Display the data in root

- 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.

In situations where you have a large, sorted dataset, binary search is the preferred choice due to its efficiency. Linear search may work well for small, unsorted lists, but as the dataset size increases, the time complexity of linear search becomes a significant drawback. Binary search's logarithmic time complexity allows it to outperform linear search, making it a practical and efficient choice for tasks like finding entries in phone books, dictionaries, and sorted databases.

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 binary tree where each node has at most two children: a left child with smaller values and a right child with greater values. It's related to binary search and supports efficient searching, insertion, and deletion operations due to its ordering property. Applications include dictionaries, symbol tables, auto-complete features, file systems, optimization problems, and network routing. BSTs can be used for maintaining sorted data, and self-balancing variants (e.g., AVL, Red-Black trees) are employed for optimization and balancing purposes.
Overall, binary search trees are versatile data structures that leverage the binary search algorithm for various applications where data needs to be organized and searched efficiently.
Applications of Binary Search Trees:

1. **Searching and Data Retrieval:** BSTs excel in searching, inserting, and deleting elements with an average time complexity of O(log N) in a balanced tree. This is commonly used in dictionaries, symbol tables, and databases.
2. **Inorder Traversal:** BSTs provide elements in sorted order when performing an inorder traversal. This is helpful for tasks like printing elements in sorted order.
3. **Efficient Data Structures:** BSTs can be used as underlying data structures for various abstract data types, such as sets, maps, and priority queues.

4. **Auto-Complete and Suggestion Features:** BSTs can be used in text editors and search engines for implementing auto-complete and suggestion features. They help predict and display potential search terms.
5. **File System Organization:** BSTs can be used to efficiently organize and search for files in a file system. Directory structures in operating systems often use binary search trees for quick directory lookups.
6. **Balancing Techniques:** Self-balancing BSTs, like AVL trees and Red-Black trees, are used in a variety of applications, including maintaining sorted order and ensuring fast search operations.