Experiment No.10
Implement Binary Search Algorithm.
Name:Ayush Gupta
Roll No:14
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.

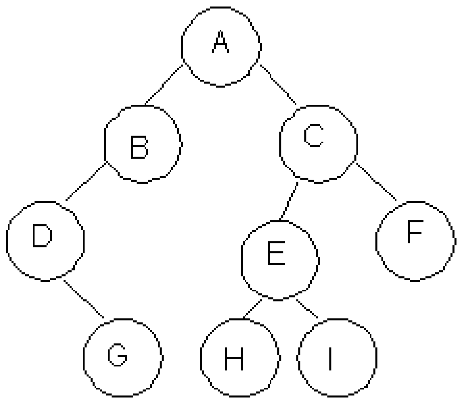
Theory:

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

- Visit the root node R
- Traverse the left subtree of R
- 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

- Traverse the left subtree of R
- Visit the root node R
- 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

- Traverse the left subtree of R
- Traverse the right sub tree of R
- 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

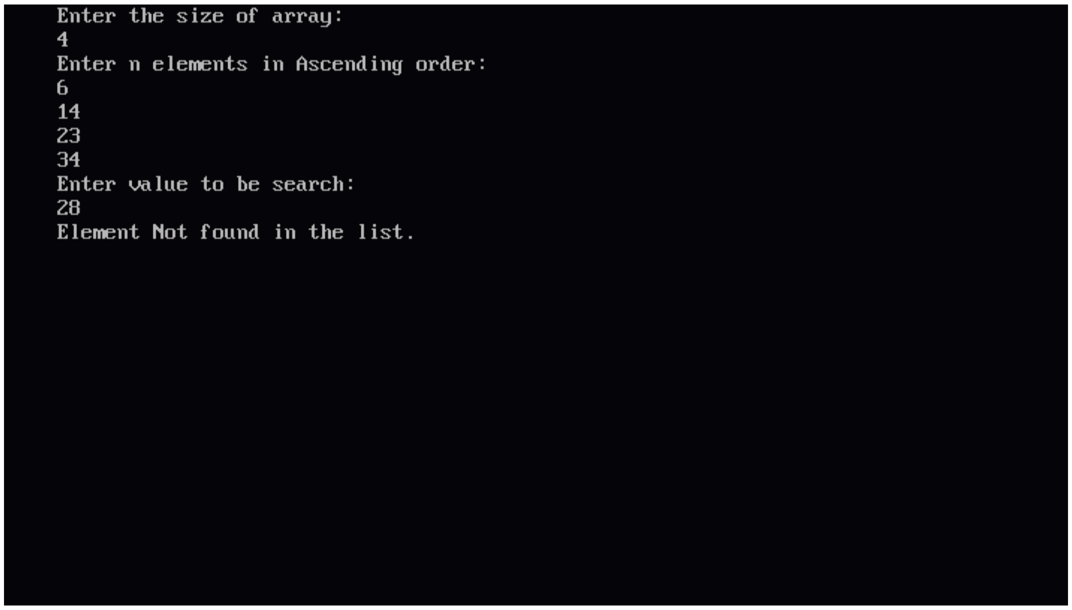
- Stop

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 the data is sorted or organized in a way that allows for the application of binary search. The primary advantage of binary search is its ability to quickly locate a target element in a sorted dataset by repeatedly dividing the search space in half. Here's a scenario illustrating when binary search is advantageous:

Scenario: Finding a Word in a Dictionary

Imagine you have a physical printed dictionary, and you want to look up a specific word. The words in the dictionary are arranged in alphabetical order, which is a sorted order. In this situation:

- Binary Search:You open the dictionary roughly in the middle (e.g., around the letter 'M'). You check the word at that position and determine whether the word you're looking for comes before or after it in the alphabet. Based on this comparison, you then proceed to a specific section of the dictionary (either the first half or the second half). You repeat this process, halving the search space with each step, until you locate the word. Binary search allows you to find the word efficiently in just a few iterations, even in a large dictionary.
- Linear Search:With a linear search, you'd start at the beginning of the dictionary and flip through the pages one by one, checking each word until you find the word you're looking for. In the worst case, you might need to go through the entire dictionary, page by page, until you locate the word. This can be a time-consuming and inefficient process, especially in a large dictionary.

In this scenario, binary search is significantly more efficient because it leverages the sorted order of the words to quickly narrow down the search space, reducing the number of comparisons needed to find the target word. It's a practical demonstration of how binary search excels when dealing with sorted data.

2) Explain the concept of "binary search tree". How is it related to binary search, and what are its applications

Binary Search Tree (BST) is a data structure that builds upon the concept of binary search and adds a hierarchical structure. It is a type of binary tree where each node has at most two children, and the nodes are organized in such a way that it allows for efficient searching, insertion, and deletion of elements.

Key Characteristics of Binary Search Tree (BST):

1. Binary Tree Structure: A BST is a binary tree, meaning each node can have at most two children, typically referred to as the left child and the right child.
2. Binary Search Property: The elements in a BST are organized in a way that satisfies the binary search property: for each node:
   - All nodes in its left subtree have values less than the node's value.
   - All nodes in its right subtree have values greater than the node's value.
3. Hierarchical Order: The hierarchical ordering of elements based on their values ensures that searching, insertion, and deletion operations can be performed more efficiently than in an unstructured list.

Relation to Binary Search:

The concept of a Binary Search Tree is an extension of the binary search algorithm to efficiently store and manipulate data. Binary search is typically used for searching in sorted arrays, while BSTs allow for efficient search and other operations in a dynamic data structure.

In binary search, the focus is on finding an element by repeatedly dividing a sorted array in half. In a Binary Search Tree, the focus is on maintaining a tree structure with a hierarchical organization of elements that preserves the binary search property.

Applications of Binary Search Trees:

Binary Search Trees are used in various applications and scenarios, including:

1. Search Operations: BSTs allow for efficient searching. The binary search property ensures that search operations can quickly locate elements within the tree, with a time complexity of O(log n) on average for balanced trees.
2. Insertion and Deletion: Insertion and deletion operations can also be performed efficiently in a BST, maintaining the binary search property after these operations. The time complexity is O(log

n) on average.

3. Ordered Data Storage: BSTs are used to maintain data in sorted order. In-order traversal of the tree yields elements in ascending order, making them suitable for applications that require ordered data.
4. Data Retrieval: Binary Search Trees are used in databases and file systems to retrieve data efficiently based on keys or values.
5. Balanced BSTs: Balanced Binary Search Trees, such as AVL trees and Red-Black trees, are used in applications like maintaining sorted data in databases, providing efficient search operations, and balancing the tree automatically to ensure logarithmic height.
6. Dictionary Implementations: Many programming languages use BSTs or balanced BSTs to implement dictionaries and associative arrays. Keys are stored in the tree, and values can be associated with them.
7. Auto-Completion and Suggestion Engines: Binary Search Trees can be used in auto-completion and suggestion features in text editors and search engines, providing quick suggestions based on partial input.
8. Symbol Tables: In computer science and software engineering, symbol tables for compilers and interpreters often use Binary Search Trees to store and retrieve identifiers.

Binary Search Trees are a fundamental data structure that combines the efficient searching properties of binary search with a hierarchical tree structure. Their applications are diverse, ranging from data retrieval to maintaining sorted data efficiently.