Experiment No.10
Implement Binary Search Algorithm.
Name: SHLOK.R.YADAV
Roll No: 64
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:

- Understand how to implement a BST using a predefined BST ADT.
- Understand the method of counting the number of nodes of a binary tree.

Theory:

A Binary Search Tree (BST) is a hierarchical data structure that organizes data in a way that allows for efficient searching, insertion, and deletion operations. It is characterized by the following properties:

- Tree Structure:
  - A BST is composed of nodes, where each node contains a key (value or data) and has at most two children.
  - The top node of the tree is called the root, and it is the starting point for all operations.

- Nodes in a BST are organized in a hierarchical manner, with child nodes below parent nodes.

- Binary Search Property:

- The defining property of a BST is the binary search property, which ensures that for any given node:
  - All nodes in its left subtree have keys (values) less than or equal to the node's key.
  - All nodes in its right subtree have keys greater than the node's key.

- In-order Traversal:
  - In-order traversal of a BST visits the nodes in ascending order of their keys.
  - This property makes BSTs useful for applications that require data to be stored and retrieved in sorted order.

- Unique Keys:
  - In a typical BST, all keys are unique. Duplicate keys may be handled differently in variations like the AVL tree.

Operations on Binary Search Trees:

- Insertion:
  - To insert a new element into a BST, start at the root and compare the value to be inserted with the key of the current node.
  - If the value is less, move to the left child; if it's greater, move to the right child.
  - Repeat this process until an empty spot (NULL) is found, and then create a new node with the value to be inserted.

- Deletion:
  - To delete a node with a specific key:
    - Locate the node, which may involve searching for the node to be deleted.
    - If the node has no children, remove it from the tree.
    - If the node has one child, replace it with its child.
    - If the node has two children, find either the node with the next highest key (successor) or the node with the next lowest key (predecessor).
    - Replace the node to be deleted with the successor (or predecessor) and then recursively delete the successor (or predecessor).

- Search:
  - To search for a key in the BST, start at the root and compare the key with the key of the current node.
    - If they match, the key is found.
    - If the key is less, move to the left child; if it's greater, move to the right child.
    - Repeat this process until the key is found or an empty spot is reached.

- Traversal:
  - In-order, pre-order, and post-order traversals can be implemented to visit all nodes in the tree.
    - In-order traversal visits nodes in ascending order, pre-order traversal visits the root before its children, and post-order traversal visits the root after its children.

Complexity Analysis:

The time complexity of basic BST operations depends on the height of the

tree. In a well-balanced BST (e.g., AVL tree), the height is logarithmic, resulting in efficient O(log n) operations. However, in the worst case, where the tree degenerates into a linked list, the time complexity becomes O(n). This highlights the importance of maintaining balanced BSTs for optimal performance. Various self-balancing BSTs, such as AVL trees and Red-Black trees, ensure logarithmic height and efficient operations in all cases.

Algorithm:-

- Node Structure:
  - Define a structure for the BST node containing data, left child, and right child pointers.

- Initialization:
  - Initialize the root pointer as NULL to represent an empty tree.

- Insertion:
  - To insert a new element with key `k`:
    - If the tree is empty, create a new node with data `k` and set it as the root.
    - Otherwise, start at the root and compare `k` with the current node's data.
    - If `k` is less, move to the left child; if greater, move to the right child.
    - Repeat this process until an empty spot is found, and insert the new node.

- Deletion:
  - To delete a node with key `k`:
    - If the tree is empty, do nothing.
    - Otherwise, search for the node with key `k`.
    - If the node has no children, remove it from the tree.

- If it has one child, replace it with its child.
- If it has two children, find the successor or predecessor, replace the node with it, and recursively delete the successor or predecessor.

- Search:
  - To search for a key `k`:
    - Start at the root and compare `k` with the current node's data.
    - If they match, return the node.
    - If `k` is less, move to the left child; if greater, move to the right child.
    - Repeat until `k` is found or an empty spot is reached.

- Traversal:
  - Implement in-order, pre-order, and post-order traversals to visit nodes.
  - In-order visits nodes in ascending order, pre-order starts at the root, and post-order visits the root after its children.

- Balancing (Optional):
  - Ensure the tree remains balanced for efficient operations, or use self-balancing BST structures like AVL or Red-Black trees.

- Complexity:
  - Basic BST operations have O(log n) time complexity on average if the tree is balanced, where 'n' is the number of nodes.
  - In the worst case (unbalanced tree), they can have O(n) time complexity.

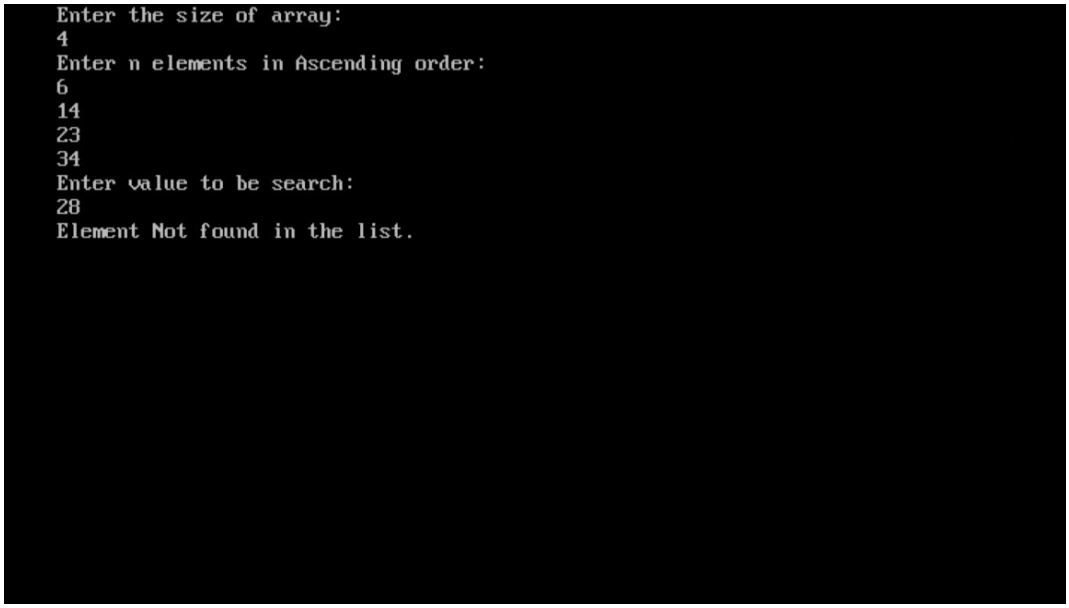
Code:

```
#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:



Conclusion:

- Describe a situation where binary search is significantly more efficient than linear search.
- Binary search is significantly more efficient than linear search when you have a sorted list of data. In a sorted list, binary search can quickly narrow down the search space by repeatedly dividing it in half, making it a logarithmic time complexity algorithm, O(log n). This is much faster than linear search, which has a time complexity of O(n), where it needs to scan through the entire list one element at a time. So, binary search is highly advantageous when you have a large dataset and you need to find a specific value quickly.

- 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 data structure that is related to binary search

in the sense that it organizes data in a tree-like structure, where each node has at most two children, a left child and a right child. The key property of a BST is that the values in the left subtree are less than or equal to the value at the current node, and the values in the right subtree are greater. This property allows for efficient searching, insertion, and deletion of elements in a sorted manner.

Applications of binary search trees include:

- Searching: Binary search trees provide efficient searching for a specific value, similar to binary search in an array, but they can be dynamically updated as new elements are added or removed.

- Sorting: In-order traversal of a binary search tree can produce a sorted list of elements.

- Auto-Complete and Suggestion Systems: Binary search trees are used to implement auto-complete and suggestion features in text editors and search engines.

- Database Indexing: Binary search trees are used in database indexing to speed up data retrieval.

- Balanced Binary Search Trees (e.g., AVL and Red-Black Trees): These specialized binary search trees ensure that the tree remains balanced, which guarantees logarithmic time complexity for search, insertion, and deletion operations.

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.