Experiment No.10
Implement Binary Search Algorithm.
Name: Swarup Satish Kakade
Roll No: 19
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 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:

1. 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.

2. 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.

3. 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.

4. 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:

1. 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.

2. 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).

3. 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.

4. 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:-

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

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

3. 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.

4. 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.

5. 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.

6. 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.

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

8. 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.

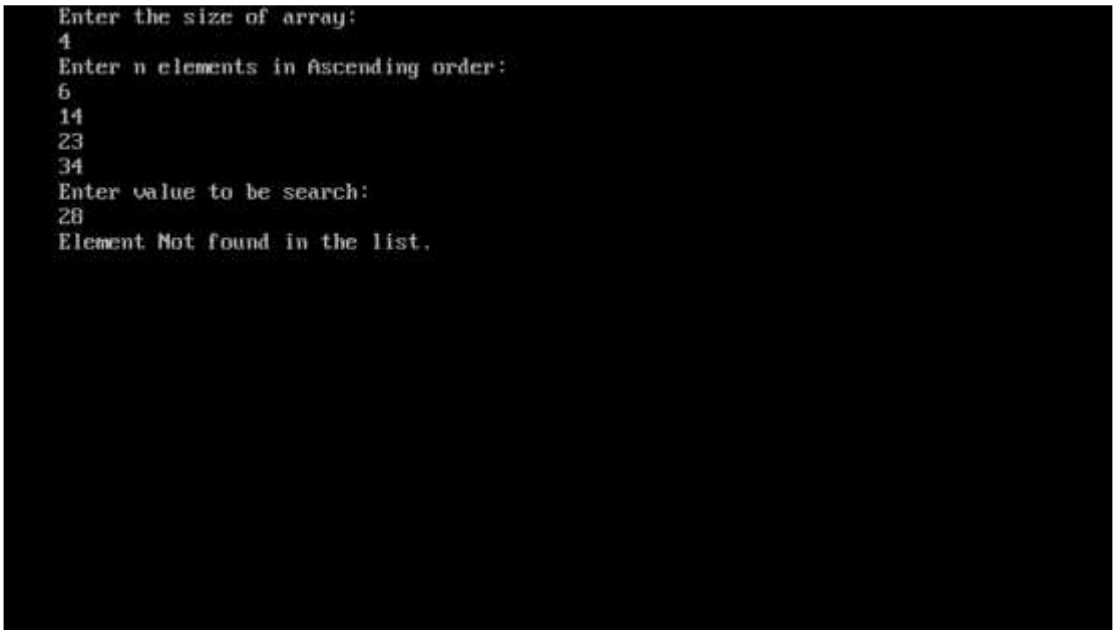
Code:

```
 #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:



```
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.
When you're dealing with substantial, sorted datasets, binary search takes the lead because of its remarkable efficiency. While linear search may suffice for compact, unsorted lists, its time complexity becomes increasingly burdensome as the dataset grows. Binary search, with its logarithmic time complexity, emerges as the superior option, surpassing linear search in terms of speed and practicality. This makes binary search an ideal and efficient approach for tasks such as locating entries in extensive references like 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 type of binary tree in which each node can have a maximum of two children: one on the left with smaller values and one on the right with greater values. Its connection to the binary search algorithm makes it a powerful tool for efficient searching, inserting, and deleting operations, all thanks to its inherent ordering property. BSTs find utility in a wide array of applications, including dictionaries, symbol tables, auto-complete functionalities, file systems, optimization challenges, and network routing. They are particularly handy for maintaining orderly data, and specialized self-balancing variants like AVL and Red-Black trees are harnessed to fine-tune and balance the structure.
In essence, binary search trees are versatile data structures that leverage the binary search method for a multitude of applications requiring organized and effective data handling.
Here are some notable applications of Binary Search Trees:

1. **Searching and Data Retrieval:** BSTs are remarkable for their efficiency in searching, inserting, and deleting elements, offering an average time complexity of O(log N) when the tree is balanced. This makes them indispensable for applications like dictionaries, symbol tables, and databases.
2. **Inorder Traversal:** BSTs have a unique feature of providing elements in a sorted order during an inorder traversal. This is incredibly valuable for tasks such as printing elements in a sorted fashion.
3. **Efficient Data Structures:** BSTs can serve as the underlying data structures for various abstract data types like sets, maps, and priority queues, contributing to their versatility.
4. **Auto-Complete and Suggestion Features:** In the realm of text editors and search engines, BSTs come into play for implementing auto-complete and

suggestion features. They play a crucial role in predicting and displaying potential search terms.

5. **File System Organization:** The efficiency of BSTs can be leveraged to organize and search for files in a file system. Many operating systems rely on binary search trees in their directory structures to ensure rapid directory lookups.
6. **Balancing Techniques:** Self-balancing BSTs, such as AVL trees and Red-Black trees, find applications across a range of scenarios, including maintaining sorted order and ensuring swift search operations. These balancing techniques enhance the overall performance and stability of BSTs.