Experiment No.10
Implement Binary Search Algorithm.
Name: Aryan Gaikwad
Roll No: 09
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.

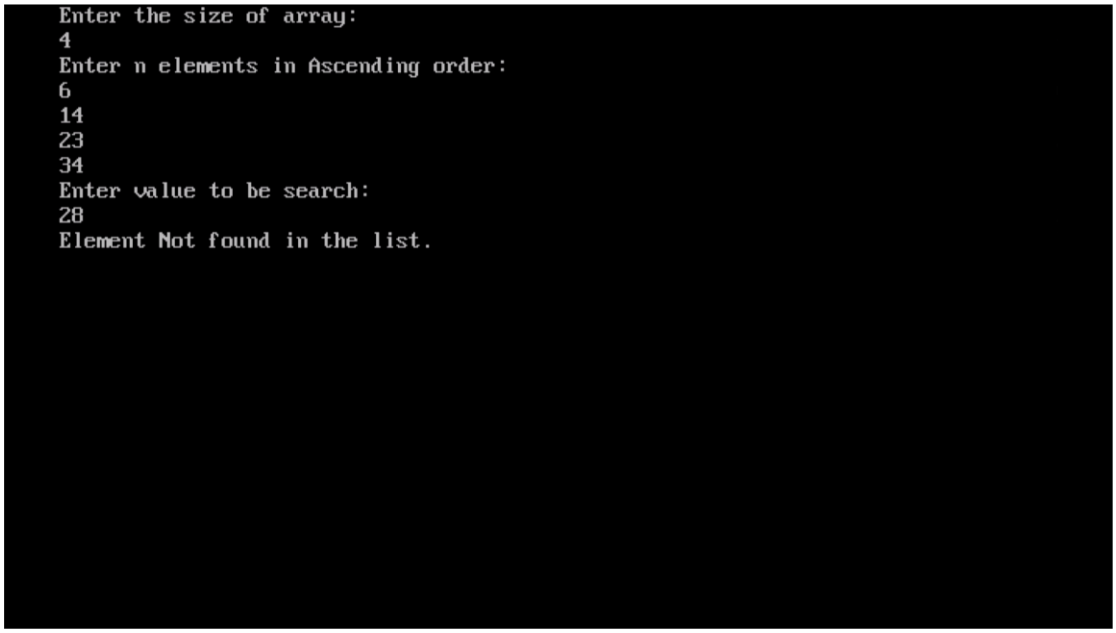
Code:

```
#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:



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 being searched is already sorted. Here's an example scenario to illustrate this:

Situation:

Suppose you have a phone book with names of individuals, and you want to find the phone number of a specific person. The phone book is organized in alphabetical order, and you know that the names are sorted.

Comparison:

1. Binary Search:
   - You open the phone book to the middle and compare the name you find with the target name.
   - If the name in the middle is the one you're looking for, you've found the number.
   - If the target name comes before the one in the middle alphabetically, you repeat the process in the first half of the phone book. Otherwise, you repeat it in the second half.
   - You keep dividing the search space in half with each step, eliminating half of the remaining names at each comparison.
   - In logarithmic time, you locate the target name.
2. Linear Search:
   - You start at the beginning of the phone book and go through each name one by one, comparing each name with the target name.
   - You continue this process until you find the target name or reach the end of the phone book.
   - In the worst case, you might need to examine every name in the phone book, which would take linear time (O(n), where n is the number of names in the phone book).

Efficiency:

In this scenario, binary search is significantly more efficient than linear search because it reduces the search space by half with each comparison. If you have a large phone book, binary search can find the target name in a much shorter time compared to the linear search, especially when the target name is towards the middle or end of the phone book. Linear search, on the other hand, would require examining each name one by one, which becomes increasingly inefficient as the dataset grows.

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.