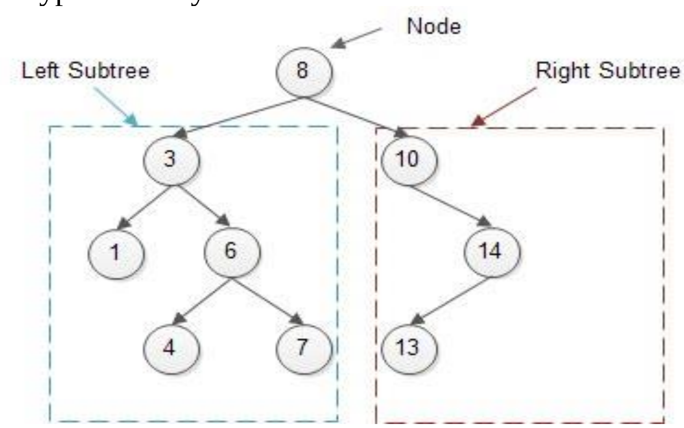
Experiment 6- Binary Search Tree

Learning Objective: Student should be able to construct a binary search tree and traverse it.

Tools: C/C++/Java/Python under Windows or Linux environment.

Theory: BST is a non-linear data structure where the elements are not accessed and stored linearly. For a binary tree to be a binary search tree, the data of all the nodes in the left sub-tree of the root node should be < the data of the root, and the data of all the nodes in the right sub-tree of the root node should be > the data of the root. Following figure shows the diagrammatic view of typical binary search tree.



There are some common operations on the binary search tree:

- Insert – inserts a new node into the tree
- Delete – removes an existing node from the tree
- Traverse – traverse the tree in pre-order, in-order and post-order. For the binary search tree, only in-order traversal makes sense
- Search – search for a given node's key in the tree

All binary search tree operations are O(H), where H is the depth of the tree. The minimum height of a binary search tree is $H = \log_2 N$, where N is the number of the tree's nodes. Therefore the complexity of a binary search tree operation in the best case is O(logN); and in the worst case, its complexity is O(N).

Algorithm-Insert

Step 1 - Create a newNode with given value and set its left and right to NULL.
Step 2 - Check whether tree is Empty.
Step 3 - If the tree is Empty, then set root to newNode.
Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).
Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.
Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

Algorithm- Delete

Case 1: Deleting a leaf node

Step 1 - Find the node to be deleted using search operation

Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has only one child then create a link between its parent node and child node.

Step 3 - Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3 - Swap both deleting node and node which is found in the above step.

Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2

Step 5 - If it comes to case 1, then delete using case 1 logic.

Step 6- If it comes to case 2, then delete using case 2 logic.

Step 7 - Repeat the same process until the node is deleted from the tree.

Algorithm-Search

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6- If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Advantages:

- The cost of insert(), delete(), search() can be kept to O(logN) where N is the number of nodes in the tree.
- The tree can be traversed in the increasing (or decreasing) order of keys, we just need to do the in-order.
- With BST - Nth smallest, Nth largest element can be found out easily as it is possible to look at the BST as a sorted array.

Disadvantage:

- To keep the time complexity O(log N) the BST should always be kept as a Balanced BST.

Applications:

1. It is used to implement dictionary.
2. It is used to implement multilevel indexing in DATABASE.
3. Tu implement Huffman Coding Algorithm.
4. It is used to implement searching Algorithm.

Learning Outcomes: The student should have the ability to implement BST with all the operations.

Course Outcomes: Upon completion of the course students will be able to construct a binary search tree with all the operations on it.

Conclusion:

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				