Lab Cycle: 7 **CSL 201 Data structures Lab**
Date of Submission: 5-12-2022
Faculty In charge: Dr Binu V P
Objective: Learn Binary Search Trees

A Binary Search Tree (BST) is an acyclic connected graph with one node designated as root node which has the following properties:

- Each node (item in the tree) has a distinct value.
- Each node has at most two children
- Both the left and right subtrees must also be binary search trees.
- The left subtree of a node contains only values less than the node's value.
- The right subtree of a node contains only values greater than the node's value.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

The major advantage of binary search trees over other data structures is that the related Search and sorting algorithms can be implemented very efficiently.

Operations

Searching

Searching a binary tree for a specific value can be a recursive or iterative process. This explanation covers a recursive method.

We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals the root, the search is successful. If the value is less than the root, search the left subtree. Similarly, if it is greater than the root, search the right subtree. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item must not be present in the tree.

This operation requires O(log n) time in the average case, but needs O(n) time in the worst-case, when the unbalanced tree resembles a linked list (degenerate tree).

Insertion

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

This operation requires time proportional to the height of the tree in the worst case, which is O(log n) time in the average case over all trees, but Ω(n) time in the worst case.

Deletion

There are three cases to be considered:

- **Deleting a leaf:** Deleting a node with no children is easy, as we can simply remove it from the tree.
- **Deleting a node with one child:** Delete it and replace it with its child.
- **Deleting a node with two children:** Suppose the node to be deleted is called N. We replace the value of N with either its in-order successor (the left-most child of the right subtree) or the in-order predecessor (the right-most child of the left subtree).

Tree Traversal

Tree-traversal refers to the process of visiting (examining and/or updating) each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well.

Compared to linear data structures like linked lists and one dimensional arrays, which have only one logical means of traversal, tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed define the traversal type. These steps (in no particular order) are: performing an action on the current node (referred to as "visiting" the node),

traversing to the left child node, and traversing to the right child node. Thus the process is most easily described through recursion.

Preorder

To traverse a non-empty binary tree in preorder, perform the following operations recursively at each node, starting with the root node:

1. **Visit the root.**
2. **Traverse the left subtree.**
3. **Traverse the right subtree.**

(This is also called Depth-first traversal.)

Preorder traversal is used to create a copy of the tree.

Preorder traversal is also used to get prefix expression of an expression tree.

Inorder

To traverse a non-empty binary tree in inorder, perform the following operations recursively at each node:

1. **Traverse the left subtree.**
2. **Visit the root.**
3. **Traverse the right subtree.**

(This is also called Symmetric traversal.)

In-order traversal is used to retrives data of binary search tree in sorted order.

Postorder

To traverse a non-empty binary tree in postorder, perform the following operations recursively at each node:

1. **Traverse the left subtree.**
2. **Traverse the right subtree.**
3. **Visit the root.**

Postorder traversal is used to delete the tree.

Postorder traversal is also used to get the postfix expression of an expression tree.

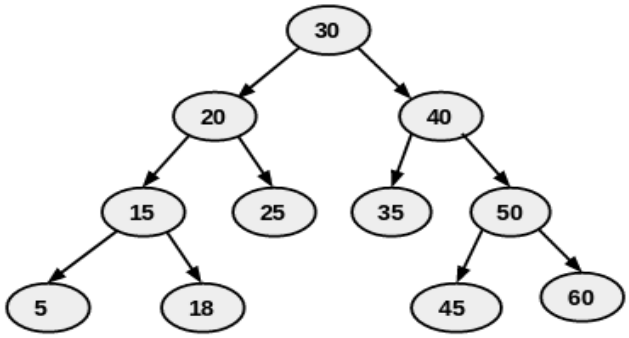
Level Order

Finally, trees can also be traversed in level-order, where we visit every node on a level before going to a lower level. This is also called **Breadth-first** traversal.

Sort

A binary search tree can be used to implement a simple but efficient sorting algorithm. Insert all the values we wish to sort into a binary search tree—and then do an inorder traversal

"Build a Binary Search Tree and Implement all the operations and Traversals"



1.Create a binary search tree (use the sample tree above)

2.Do the traversals

- **Inorder -output is {5 , 15 , 18 , 20 , 25 , 30 , 35 , 40 , 45 , 50 , 60}**
- **Preorder-output is {30 , 20 , 15 , 5 , 18 , 25 , 40 , 35 , 50 , 45 , 60}**
- **Post Order-output is {5 , 18 , 15 , 25 , 20 , 35 , 45 , 60 , 50 , 40 , 30}**
- **Level Order-output is {30 , 20 , 40 , 15 , 25 , 35 , 50 , 5 , 18 , 45 , 60}**

3.Delete a specified node

4.Search for a given key element (print found/not found also print the number of comparisons made)