CL-218 Data Structures LAB

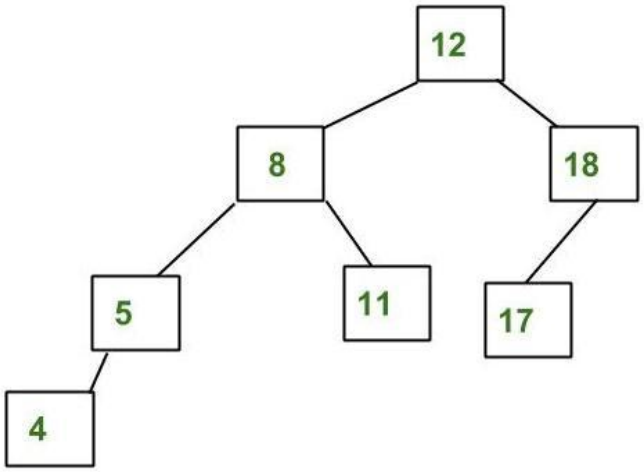
AVL Tree Insertion

INSTRUCTOR: MUHAMMAD HAMZA

AVL Tree (Insertion)

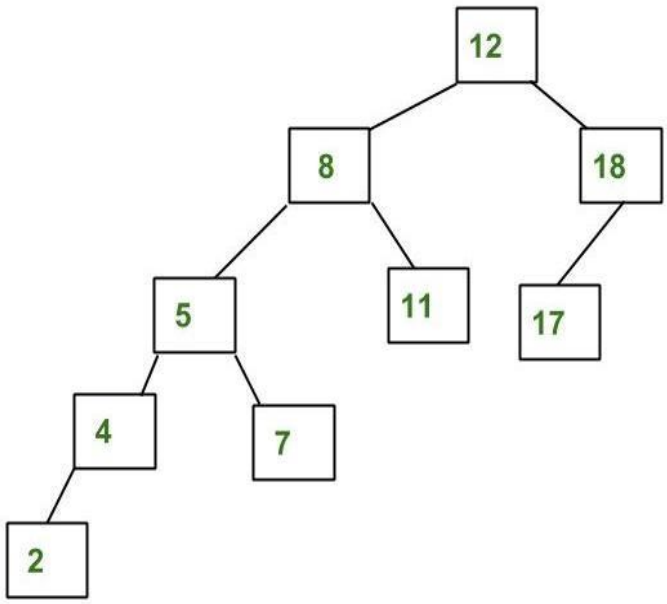
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree

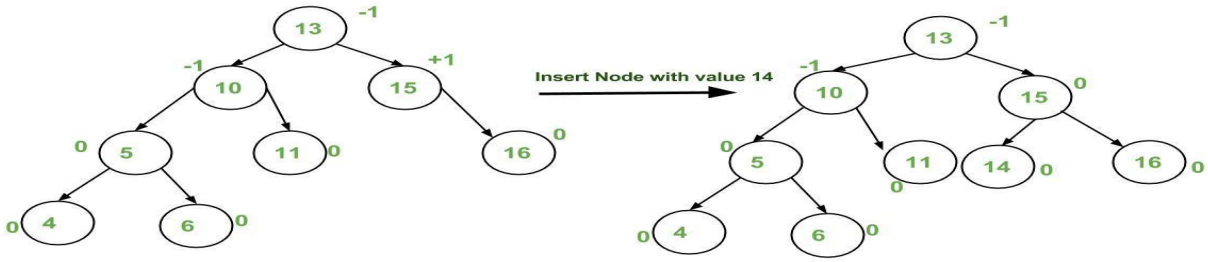


The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

Why AVL Trees?

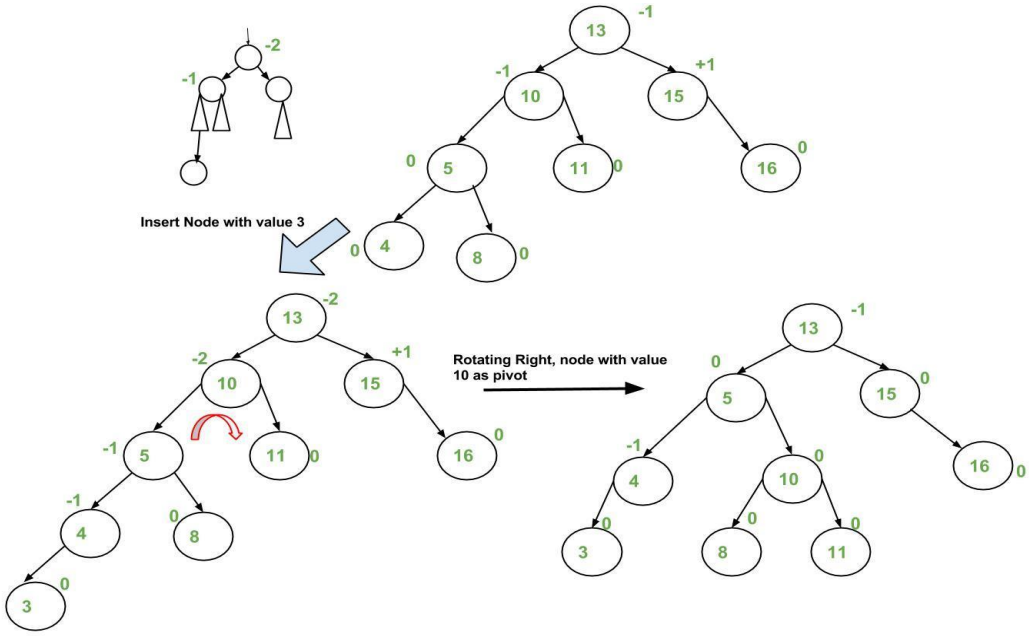
Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

Use case where no rotation is required:



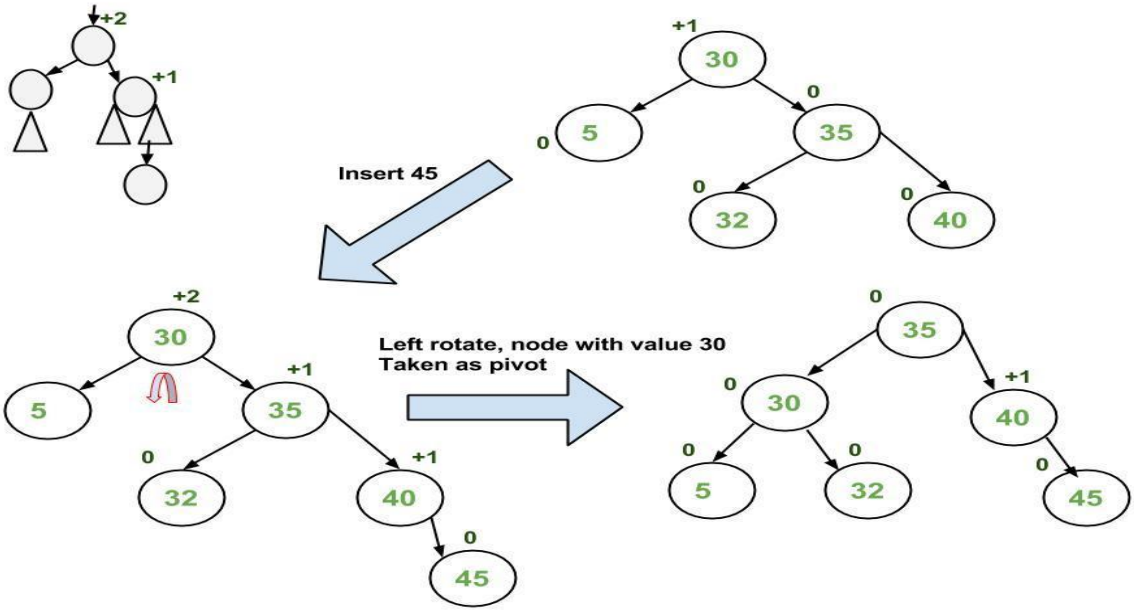
Case 1- Insertion in the left subtree of left child:

Solution: Apply simple right rotation.



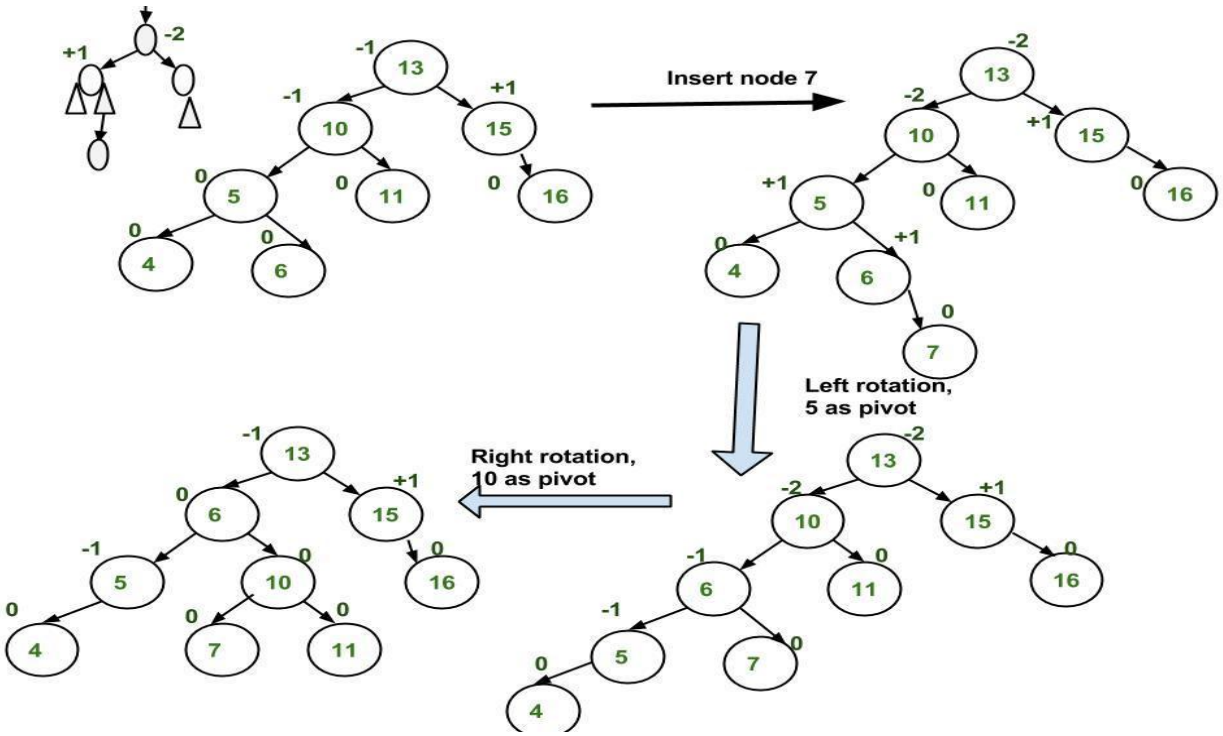
Case 2- Insertion in the right subtree of right child:

Solution: Apply simple left rotation.



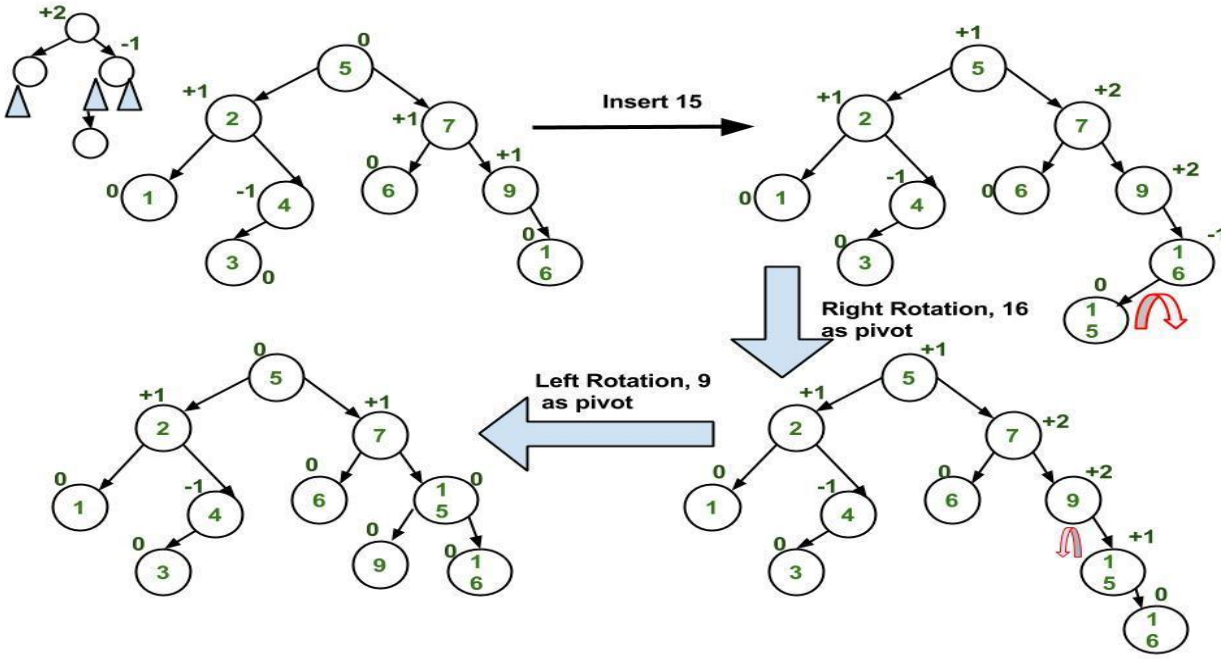
Case 3- Insertion in the right subtree of left child:

Solution: Apply left rotation on the left child first, then right rotation on the parent node.



Case 4- Insertion in the right subtree of left child:

Solution: Apply right rotation on the right child first, then left rotation on the parent node.



Implementation

Following is the implementation for AVL Tree Insertion. The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.
1) Perform the normal BST insertion.
2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.
3) Get the balance factor (left subtree height – right subtree height) of the current node.
4) If the balance factor is greater than 1, then the current node is unbalanced and we are either in the Left Left case or left Right case. To check whether it is the left left case or not, compare the newly inserted key with the key in the left subtree root.
5) If the balance factor is less than -1, then the current node is unbalanced and we are either in the Right-Right case or Right-Left case. To check whether it is the Right Right

case or not, compare the newly inserted key with the key in the right subtree root.