```
            return(sum)
```

Problem-75 Given a BST of size n, in which each node r has an additional field $r \rightarrow size$, the number of the keys in the sub-tree rooted at r (including the root node r). Give an $O(h)$ algorithm $GreaterthanConstant(r, k)$ to find the number of keys that are strictly greater than k (h is the height of the binary search tree).

Solution:

```
def GreaterthanConstant (r, k):
    keysCount = 0
    while (r):
        if (k < r.data):
            keysCount = keysCount + r.right.size + 1
            r = r.left
        else if (k > r.data):
            r = r.right
        else:
            keysCount = keysCount + r.right.size
            break
    return keysCount
```

The suggested algorithm works well if the key is a unique value for each node. Otherwise when reaching $k=r.data$, we should start a process of moving to the right until reaching a node y with a key that is bigger then k, and then we should return $keysCount + y.size$. Time Complexity: $O(h)$ where h=$O(n)$ in the worst case and $O(logn)$ in the average case.

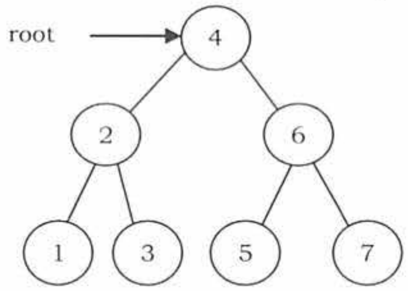
6.12 Balanced Binary Search Trees

In earlier sections we have seen different trees whose worst case complexity is $O(n)$, where n is the number of nodes in the tree. This happens when the trees are skew trees. In this section we will try to reduce this worst case complexity to $O(logn)$ by imposing restrictions on the heights.

In general, the height balanced trees are represented with $HB(k)$, where k is the difference between left subtree height and right subtree height. Sometimes k is called *balance factor*.

Full Balanced Binary Search Trees

In $HB(k)$, if $k = 0$ (if balance factor is zero), then we call such binary search trees as *full* balanced binary search trees. That means, in $HB(0)$ binary search tree, the difference between left subtree height and right subtree height should be at most zero. This ensures that the tree is a full binary tree. For example,



Note: For constructing $HB(0)$ tree refer to *Problems* section.

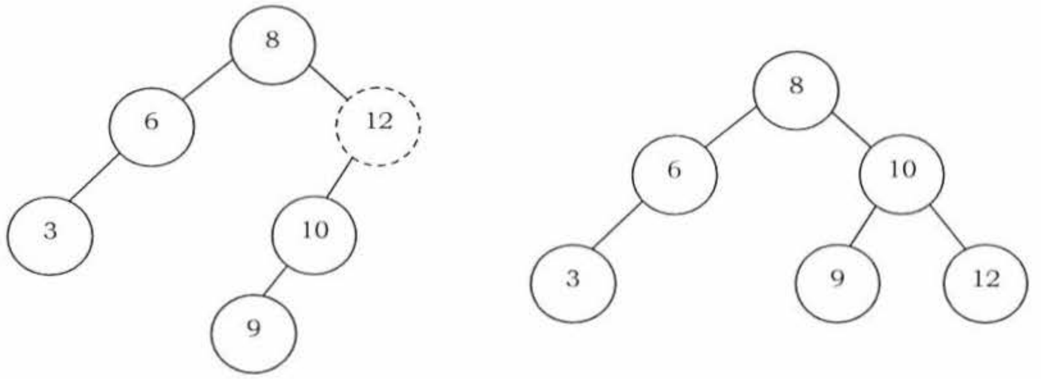
6.13 AVL (Adelson-Velskii and Landis) Trees

In $HB(k)$, if $k = 1$ (if balance factor is one), such a binary search tree is called an *AVL tree*. That means an AVL tree is a binary search tree with a *balance* condition: the difference between left subtree height and right subtree height is at most 1.

Properties of AVL Trees

A binary tree is said to be an AVL tree, if:

- It is a binary search tree, and
- For any node X, the height of left subtree of X and height of right subtree of X differ by at most 1.



As an example, among the above binary search trees, the left one is not an AVL tree, whereas the right binary search tree is an AVL tree.

Minimum/Maximum Number of Nodes in AVL Tree

For simplicity let us assume that the height of an AVL tree is h and $N(h)$ indicates the number of nodes in AVL tree with height h. To get the minimum number of nodes with height h, we should fill the tree with the minimum number of nodes possible. That means if we fill the left subtree with height $h-1$ then we should fill the right subtree with height $h-2$. As a result, the minimum number of nodes with height h is:

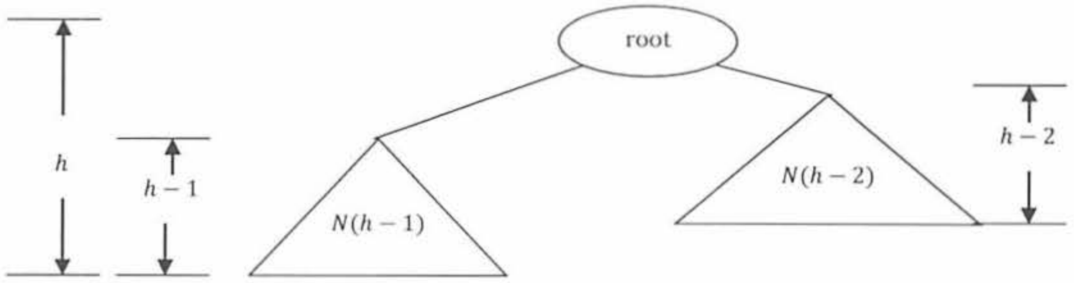
$$N(h) = N(h-1) + N(h-2) + 1$$

In the above equation:

- $N(h-1)$ indicates the minimum number of nodes with height $h-1$.
- $N(h-2)$ indicates the minimum number of nodes with height $h-2$.
- In the above expression, "1" indicates the current node.

We can give $N(h-1)$ either for left subtree or right subtree. Solving the above recurrence gives:

$$N(h) = O(1.618^h) \Rightarrow h = 1.44logn \approx O(logn)$$



Where n is the number of nodes in AVL tree. Also, the above derivation says that the maximum height in AVL trees is O($logn$). Similarly, to get maximum number of nodes, we need to fill both left and right subtrees with height $h-1$. As a result, we get:

$$N(h) = N(h-1) + N(h-1) + 1 = 2N(h-1) + 1$$

The above expression defines the case of full binary tree. Solving the recurrence we get:

$$N(h) = O(2^h) \Rightarrow h = logn \approx O(logn)$$

∴ In both the cases, AVL tree property is ensuring that the height of an AVL tree with n nodes is O($logn$).

AVL Tree Declaration

Since AVL tree is a BST, the declaration of AVL is similar to that of BST. But just to simplify the operations, we also include the height as part of the declaration.

```
class AVLNode:
    def __init__(self,data,balanceFactor,left,right):
        self.data = data
        self.balanceFactor = 0
        self.left = left
        self.right = right
```

Finding the Height of an AVL tree

```
def height(self):
    return self.recHeight(self.root)
def recHeight(self,root):
    if root == None:
            return 0
    else:
            leftH = self.recHeight(r.left)
            rightH = self.recHeight(r.right)
            if leftH>rightH:
              return 1+leftH
            else:
              return 1+rightH
```

Time Complexity: O(1).

Rotations

When the tree structure changes (e.g., with insertion or deletion), we need to modify the tree to restore the AVL tree property. This can be done using single rotations or double rotations. Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of a subtree by 1.

So, if the AVL tree property is violated at a node *X*, it means that the heights of left(*X*) and right(*X*) differ by exactly 2. This is because, if we balance the AVL tree every time, then at any point, the difference in heights of left(*X*) and right(*X*) differ by exactly 2. Rotations is the technique used for restoring the AVL tree property. This means, we need to apply the rotations for the node *X*.

Observation: One important observation is that, after an insertion, only nodes that are on the path from the insertion point to the root might have their balances altered, because only those nodes have their subtrees altered. To restore the AVL tree property, we start at the insertion point and keep going to the root of the tree.

While moving to the root, we need to consider the first node that is not satisfying the AVL property. From that node onwards, every node on the path to the root will have the issue.

Also, if we fix the issue for that first node, then all other nodes on the path to the root will automatically satisfy the AVL tree property. That means we always need to care for the first node that is not satisfying the AVL property on the path from the insertion point to the root and fix it.

Types of Violations

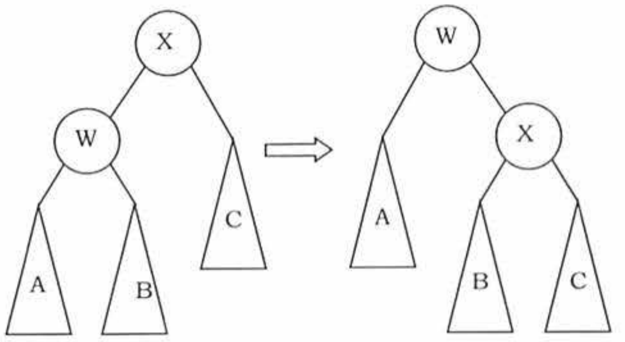
Let us assume the node that must be rebalanced is *X*. Since any node has at most two children, and a height imbalance requires that *X's* two subtree heights differ by two, we can observe that a violation might occur in four cases:

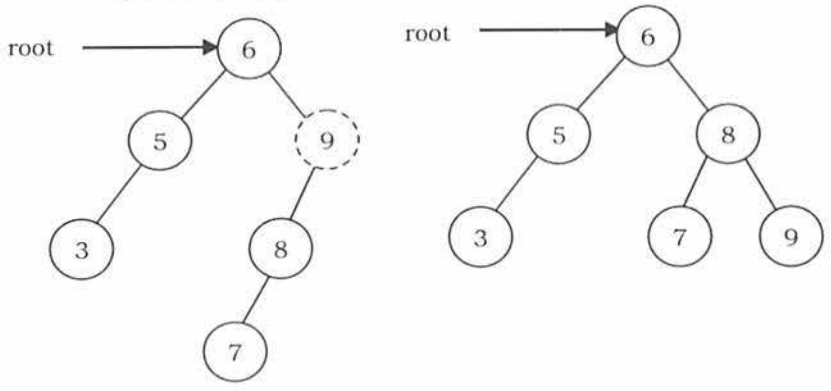
1. An insertion into the left subtree of the left child of *X*.
2. An insertion into the right subtree of the left child of *X*.
3. An insertion into the left subtree of the right child of *X*.
4. An insertion into the right subtree of the right child of *X*.

Cases 1 and 4 are symmetric and easily solved with single rotations. Similarly, cases 2 and 3 are also symmetric and can be solved with double rotations (needs two single rotations).

Single Rotations

Left Left Rotation (LL Rotation) [Case-1]: In the case below, node *X* is not satisfying the AVL tree property. As discussed earlier, the rotation does not have to be done at the root of a tree. In general, we start at the node inserted and travel up the tree, updating the balance information at every node on the path.



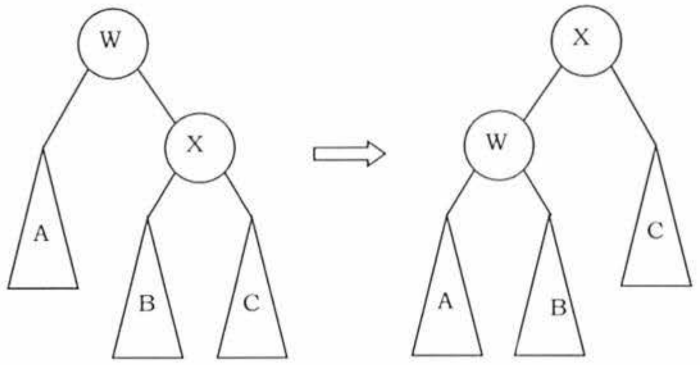


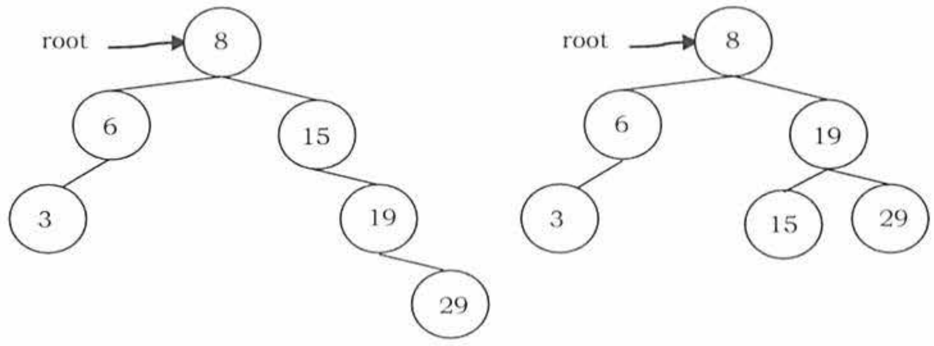
For example, in the figure above, after the insertion of 7 in the original AVL tree on the left, node 9 becomes unbalanced. So, we do a single left-left rotation at 9. As a result we get the tree on the right.

```
    def singleLeftRotate(self,root):
        W = root.left
        root.left = W.right
        W.right = root
        return W
```

Time Complexity: O(1). Space Complexity: O(1).

Right Right Rotation (RR Rotation) [Case-4]: In this case, node *X* is not satisfying the AVL tree property.





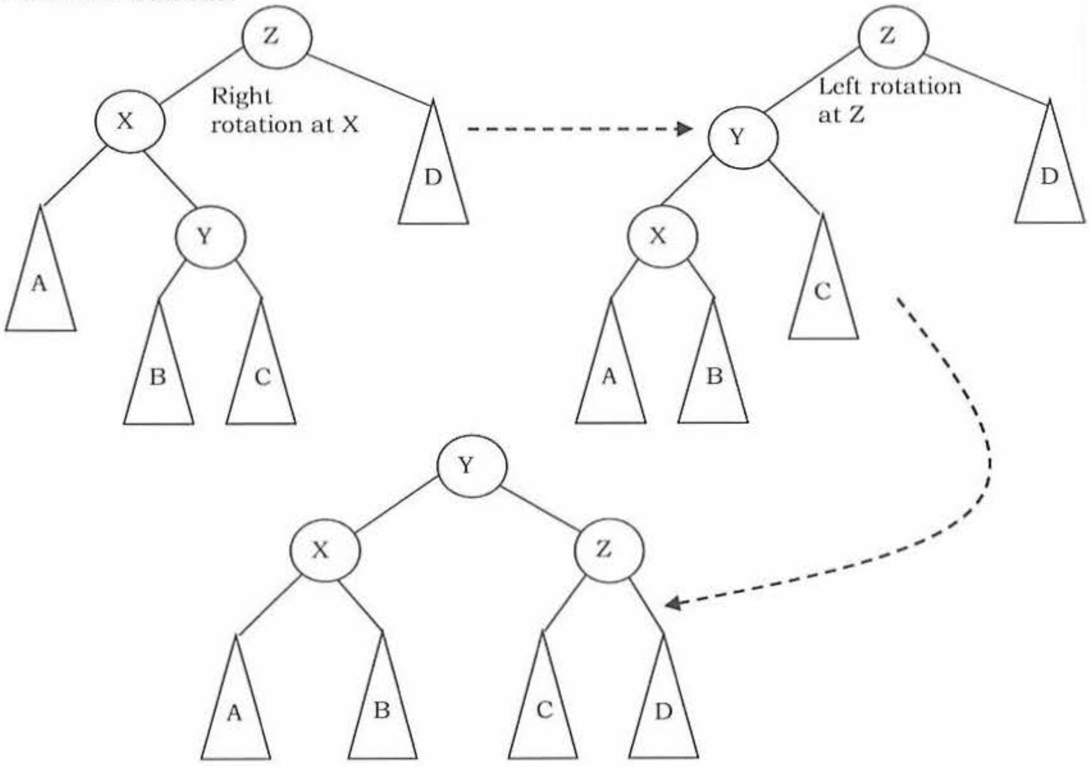
For example, in the above figure, after the insertion of 29 in the original AVL tree on the left, node 15 becomes unbalanced. So, we do a single right-right rotation at 15. As a result we get the tree on the right.

```
def singleRightRotate(self,root):
    X = root.right
    root.right = X.left
    X.left = root
    return X
```

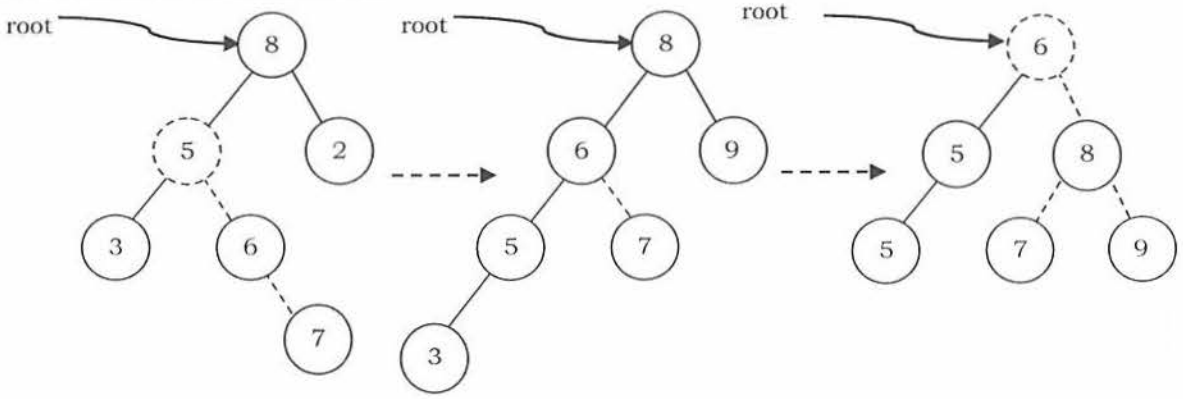
Time Complexity: O(1). Space Complexity: O(1).

Double Rotations

Left Right Rotation (LR Rotation) [Case-2]: For case-2 and case-3 single rotation does not fix the problem. We need to perform two rotations.



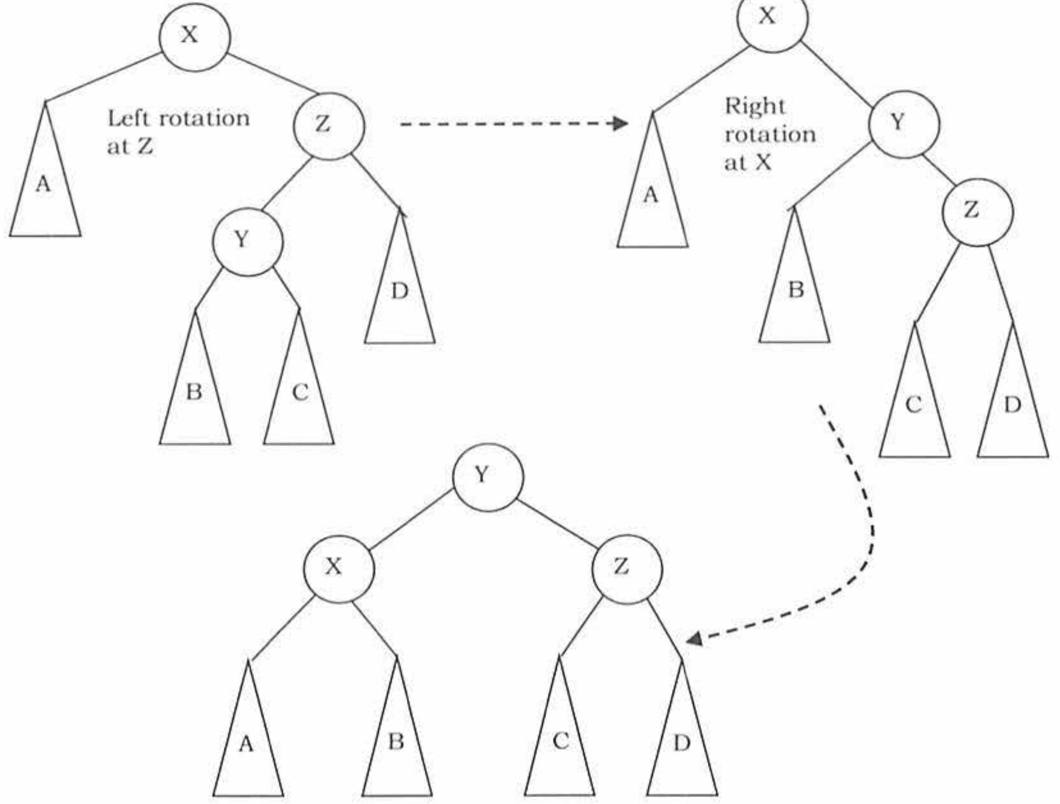
As an example, let us consider the following tree: Insertion of 7 is creating the case-2 scenario and right side tree is the one after double rotation.

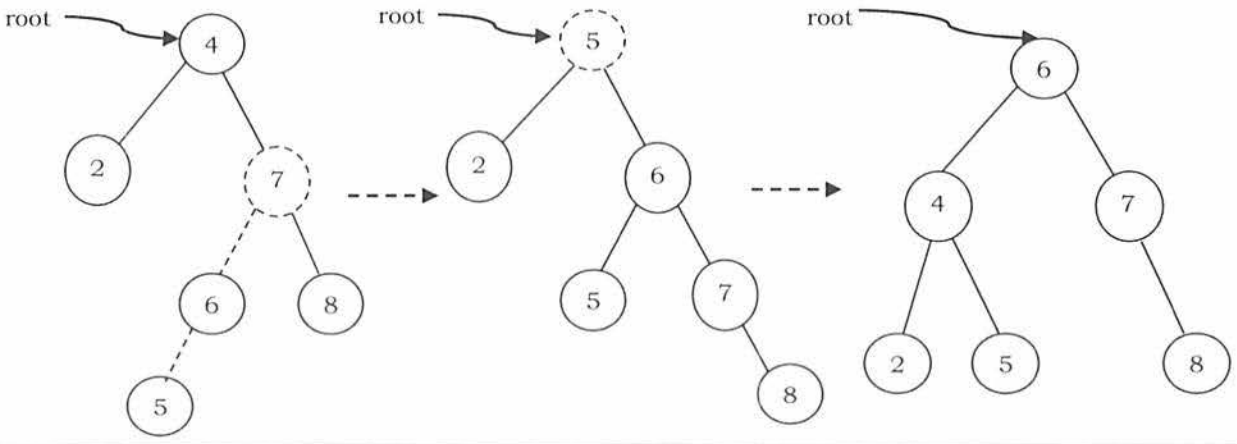


Code for left-right double rotation can be given as:

```
def rightLeftRotate(self,root):
    X = root.left
    if X.balanceFactor == -1:
        root.balanceFactor = 0
        X.balanceFactor = 0
        root = self.singleLeftRotate(root)
    else:
        Y = X.right
        if Y.balanceFactor == -1:
            root.balanceFactor = 1
            X.balanceFactor = 0
        elif Y.balanceFactor == 0:
            root.balanceFactor = 0
            X.balanceFactor = 0
        else:
            root.balanceFactor = 0
            X.balanceFactor = -1
        Y.balanceFactor = 0
        root.left = self.singleRightRoate(X)
        root = self.singleLeftRotate(root)
    return root
```

Right Left Rotation (RL Rotation) [Case-3]: Similar to case-2, we need to perform two rotations to fix this scenario.





As an example, let us consider the following tree: The insertion of 6 is creating the case-3 scenario and the right side tree is the one after the double rotation.

```
def rightLeftRotate(self, root):
    X = root.right
    if X.balanceFactor == 1:
        root.balanceFactor = 0
        X.balanceFactor = 0
        root = self.singleRightRoate(r)
    else:
        Y = X.left
        if Y.balanceFactor == -1:
            root.balanceFactor = 0
            X.balanceFactor = 1
        elif Y.balanceFactor == 0:
            root.balanceFactor = 0
            X.balanceFactor = 0
        else:
            root.balanceFactor = -1
            X.balanceFactor = 0
        Y.balanceFactor = 0
        root.right = self.singleLeftRotate(X)
        root = self.singleRightRoate(root)
    return root
```

Insertion into an AVL tree

Insertion into an AVL tree is similar to a BST insertion. After inserting the element, we just need to check whether there is any height imbalance. If there is an imbalance, call the appropriate rotation functions.

```
def insert(self,data):
    newNode = AVLNode(data,0,None,None)
    [self.root,taller] = self.recInsertAVL(self.root,newNode)
def recInsertAVL(self, root, newNode):
    if root == None:
        root = newNode
        root.balanceFactor = 0
        taller = True
    elif newNode.data< root.data:
        [root.left,taller] = self.recInsertAVL(root.left, newNode)
        if taller:
            if root.balanceFactor == 0 :
                root.balanceFactor = -1
            elif root.balanceFactor == 1:
                root.balanceFactor= 0
                taller = False
            else:
                root = self.rightLeftRotate(root)
                taller = False
    else :
        [root.right,taller] = self.recInsertAVL(root.right, newNode)
        if taller:
            if root.balanceFactor == -1:
                root.balanceFactor = 0
                taller = False
            elif root.balanceFactor == 0 :
                root.balanceFactor = 1
            else:
                root = self.rightLeftRotate(root)
                taller = False
    return [root,taller]
```

Time Complexity: O(*logn*). Space Complexity: O(*logn*).