NOTES

Constructing tree from given traversal order

One of the following combinations can be used to uniquely draw binary tree-

1. Inorder and Preorder
2. Inorder and Postorder

Constructing Tree from given inorder and preorder traversal

As we know that in a preorder sequence, leftmost element is the root of the tree and by searching the same in inorder sequence, we can find left and right subtree i.e. all elements to the left of root in inorder are left subtree and all elements to the right of root in inorder are right subtree.

Inorder Sequence: 3 5 7 10 17 20 30
Preorder Sequence: 10 5 3 7 20 17 30

10 is the root element so from inorder sequence 3 5 7 are part of left subtree and 17 20 30 are part of right subtree.

10

3 5 7 17 20 30

For squence 3 5 7, look at the preorder sequence, 5 is the root element and from property of BST we can say that 3 is left child of 5 and 7 is right child of 5.

Similarly for sequence 17 20 30, look at the preorder sequence, 20 is the root element and from property of BST we can say that 17 is left child of 5 and 30 is right child of 5.

10

5 20

3 7 17 30

```
class Node:
    def __init__(self, data):
        self.value = data
        self.left = None
        self.right = None

pre_index = 0

def find_Index(array, start, end, value):
    for i in range(start, end + 1):
        if array[i] == value:
            return i

def build_tree(inOrder, preOrder, start, end):
    global pre_index

    if (start > end):
        return None

    root = Node(preOrder[pre_index])
    pre_index += 1

    if start == end :
        return root

    in_index = find_Index(inOrder, start, end, root.value)
```

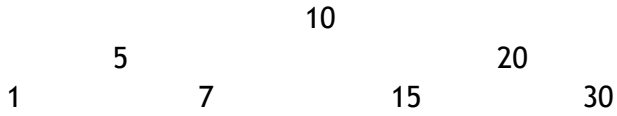
```
    root.left = build_tree(inOrder, preOrder, start, in_index-1)
    root.right = build_tree(inOrder, preOrder, in_index + 1, end)
    return root

def PostOrder(root):
    if root is None:
        return
    PostOrder(root.left)
    PostOrder(root.right)
    print(root.value,end = " ")

inOrder = [10 , 20 , 30 , 100 , 150 , 200 , 300]
preOrder = [100 , 20 , 10 , 30 , 200 , 150 , 300]
length = len(inOrder)
root = build_tree(inOrder, preOrder, 0, (length - 1))
print('Postorder Traversal of the constructed tree:')
PostOrder(root)
```

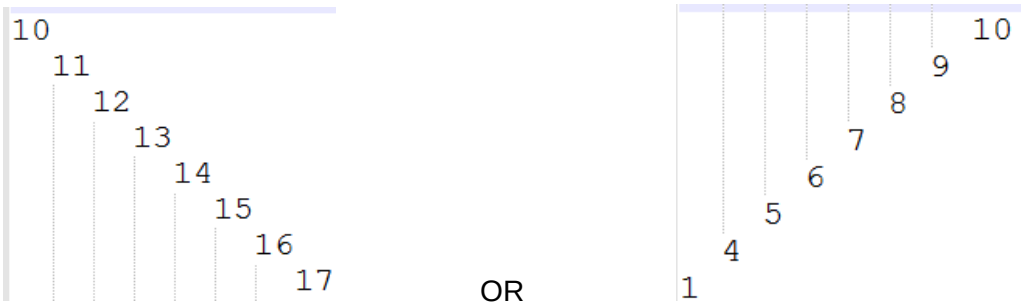
Using inorder & postorder

Inorder: 1 5 7 10 15 20 30
Postorder: 1 7 5 15 30 20 10 [Last element, at index n-1, Root element]



AVL Tree

Consider the following BSTs



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 is O(n) for above cases as they are 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. Here comes the **AVL tree** that is named after their inventor **Adelson, Velski & Landis**, AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

BalanceFactor = height(left-subtree) − height(right-subtree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

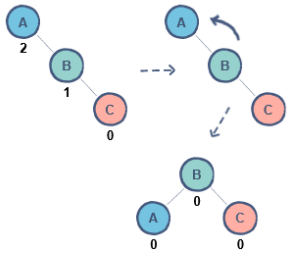
Insertion in AVL Tree

To balance itself, an AVL tree may perform the following four kinds of rotations −
1. Left rotation
2. Right rotation
3. Left-Right rotation
4. Right-Left rotation

The first two rotations are **single rotations** and the next two rotations are **double rotations**. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

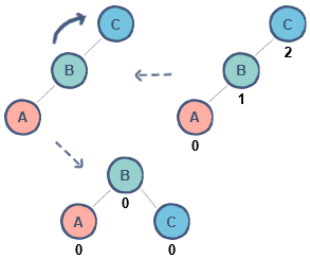
Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



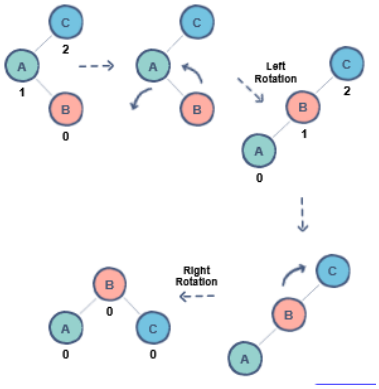
Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation. As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.



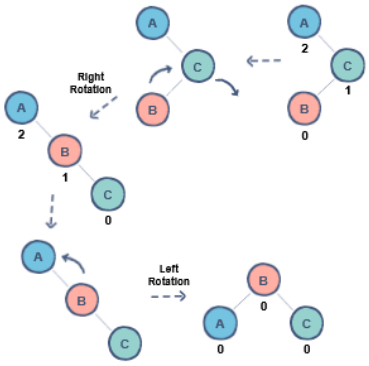
Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.



Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.



Overall summary

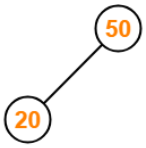
If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.
If BF(node) = -2 and BF(node -> right-child) = 1, perform RR rotation.
If BF(node) = -2 and BF(node -> right-child) = +1, perform RL rotation.
If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation.

::An Example::
Construct AVL Tree for the following sequence of numbers-
50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

1. `Start with 50`

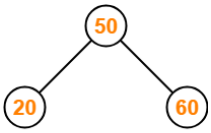
Tree is Balanced

2. As `20` < `50`, so insert `20` in `50`'s left sub tree.



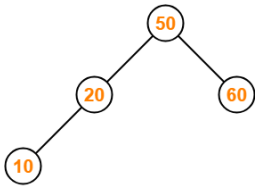
Tree is Balanced

3. As `60` > `50`, so insert `60` in `50`'s right sub tree.



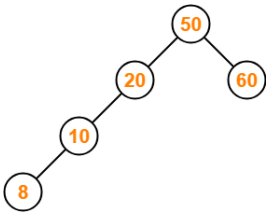
Tree is Balanced

4. As `10` < `50`, so insert `10` in `50`'s left sub tree `and 10` < `20`, so insert `10` in `20`'s left sub tree.



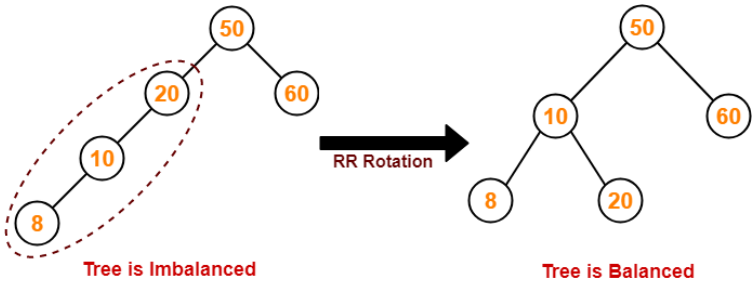
Tree is Balanced

5. As `8` < `50`, so insert `8` in `50`'s left sub tree `and as 8` < `20`, so insert `8` in `20`'s left sub tree `and as 8` < `10`, so insert `8` in `10`'s left sub tree.



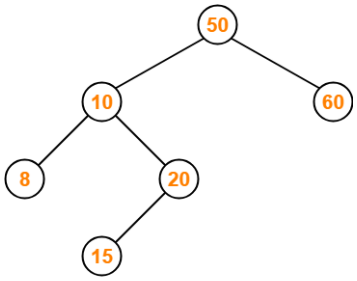
Tree is Imbalanced

`Apply rotation as balanace factor is more than 1 fro element 20 and 50 so apply RR rotation`



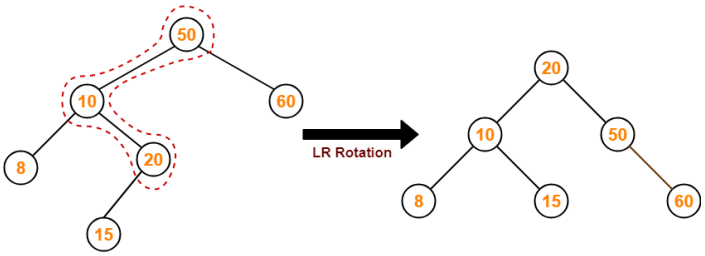
Tree is Imbalanced Tree is Balanced

6. As `15` < `50`, so insert `15` in `50`'s left sub tree `and as 15` > `10`, so insert `15` in `10`'s right sub tree `and as 15` < `20`, so insert `15` in `20`'s left sub tree.



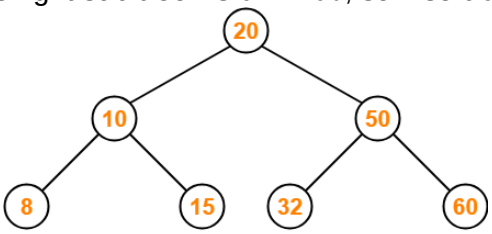
Tree is Imbalanced

Tree is imbalanced so apply LR Rotation because 10 has balance factor -1 and 50 has balance factor 2



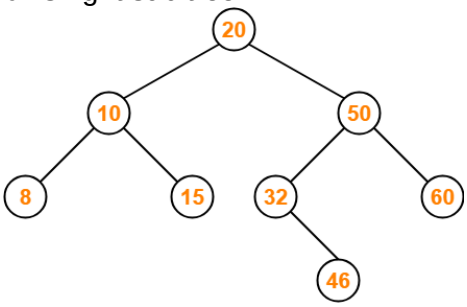
Tree is Imbalanced Tree is Balanced

7. As 32 > 20, so insert 32 in 20's right sub tree.As 32 < 50, so insert 32 in 50's left sub tree.



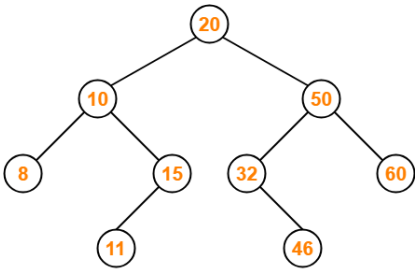
Tree is Balanced

8. As 46 > 20, so insert 46 in 20's right sub tree and as 46 < 50, so insert 46 in 50's left sub tree and as 46 > 32, so insert 46 in 32's right sub tree.



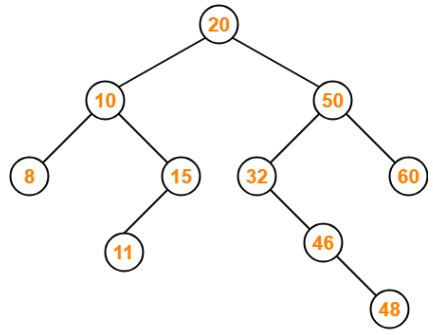
Tree is Balanced

9. As 11 < 20, so insert 11 in 20's left sub tree and as 11 > 10, so insert 11 in 10's right sub tree and as 11 < 15, so insert 11 in 15's left sub tree.



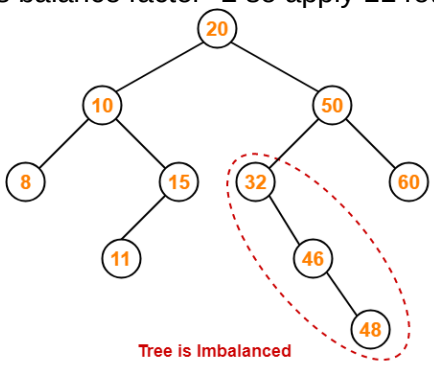
Tree is Balanced

10. As 48 > 20, so insert 48 in 20's right sub tree and 48 < 50, so insert 48 in 50's left sub tree and 48 > 32, so insert 48 in 32's right sub tree and 48 > 46, so insert 48 in 46's right sub tree.



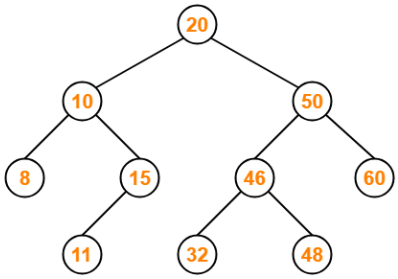
Tree is imbalanced

Tree is imbalanced because 32 has balance factor -2 so apply LL rotation



Tree is Imbalanced

LL Rotation



Tree is Balanced