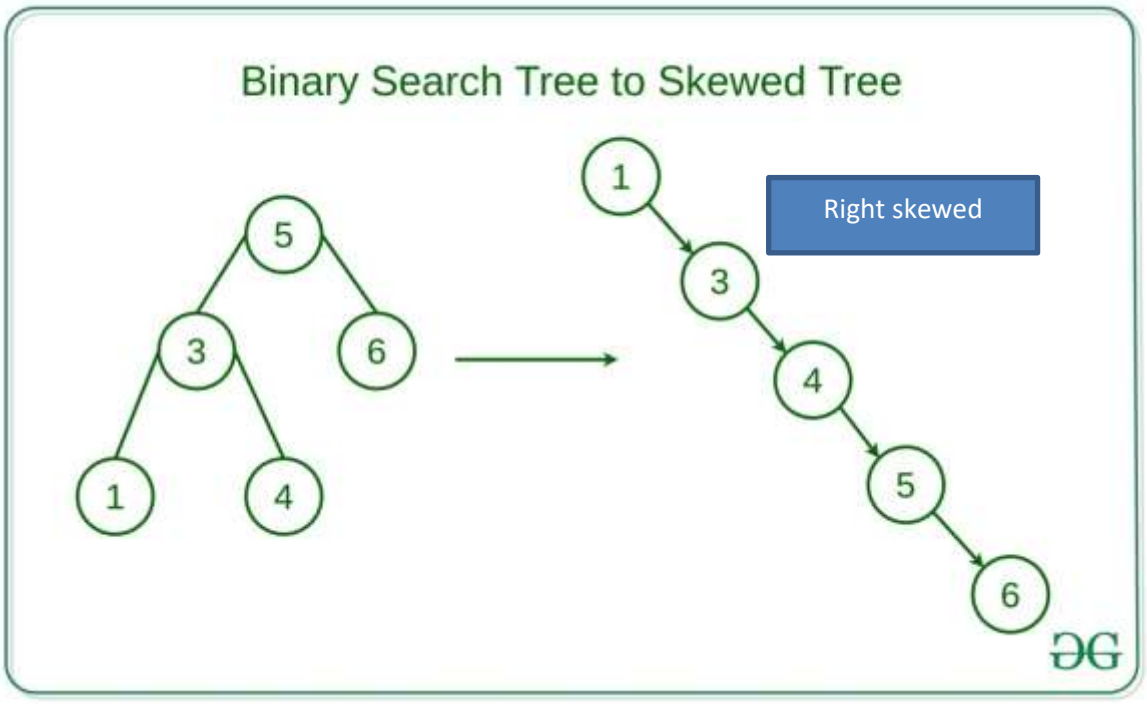
AVL Tree

AVL tree is introduced to overcome a major problem of Binary Search Tree. The problem is related to height of the BST. The problem is Explain below

The height of a binary tree is the path from root to any leaf node. If there are n nodes in the tree then height of the tree is:--



Explanation: -- if the node of the BST is come into sorted order then the BST will called either left skewed or right skewed. Which is the worst – case performance in case of BST? It will take linear time complexity O(n).

Skewed BST

Height of the tree: h=N-1, called Max height of the tree (in case consider root node is 0)

Time complexity: linear time complexity O(n)

No.of nodes in the tree: h+1 (in case of take root node as 0)

Complete BST

Height of the tree: h=log (2n), called minimum height of the tree

Time Complexity: O log(2n)

No. of nodes in the tree: 2N-1

So, we have to balance the BST using balance factor. So AVL Tree is a self-balancing binary search tree.

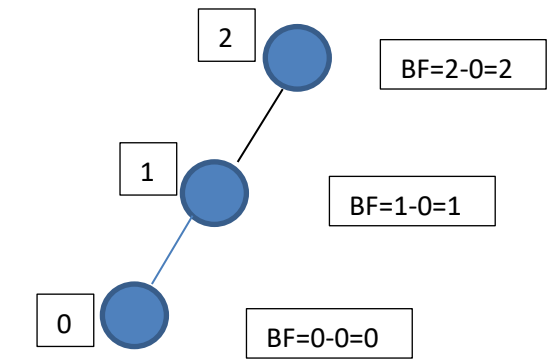
Balance Factor

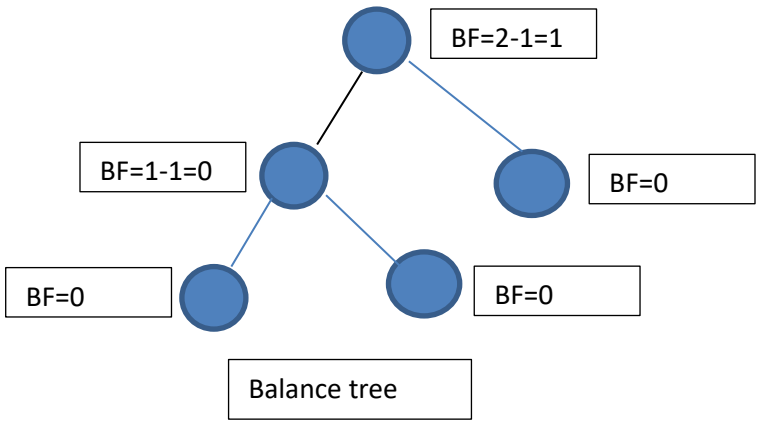
BF=height(left-subtree)-height(right-subtree)

In AVL Tree each node is associated with a balance factor. Which is calculated by subtracting the height of its right sub tree from the height of its left sub tree and this balance factor for each node cannot be greater than 1 (which means BF can be 0, -1, and 1)

→ In case of calculating height always count the edges not the nodes

→Rotation can do always using 3 nodes. No matter how long the tree rotation is done with first 3 nodes.

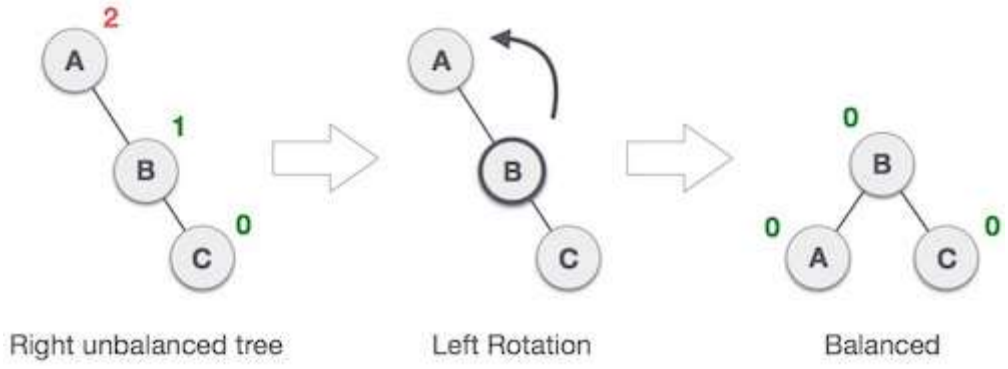




There are 4 types of rotation can be perform on a binary tree to make it a balance BST

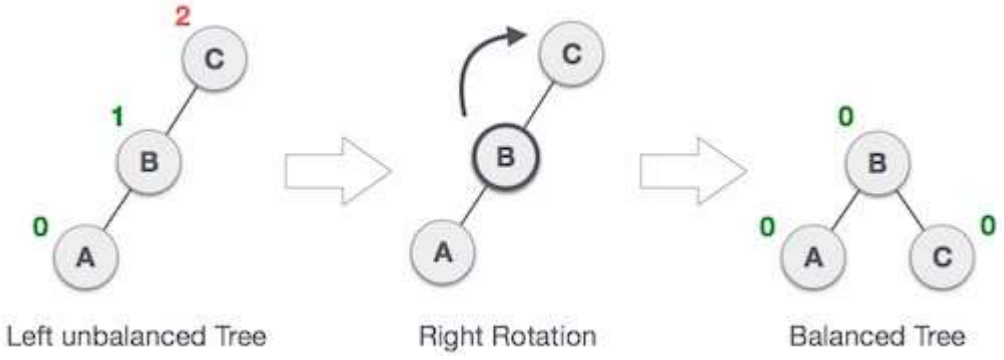
1. Left Rotation
2. Right Rotation
3. Left-right rotation
4. Right-left rotation

Left Rotation: -- if a tree is unbalance when a node is inserted into the right subtree of a right subtree. Then we need to perform single left rotation.



When node C is inserted into right sub tree node A became unbalanced. So we have to perform left rotation on node A.(since node insert only right sub tree so it is single left rotation)

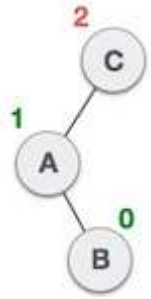
Right- rotation: -- if a tree is unbalanced when a node is inserted into left subtree of a left sub- tree then we needs to perform Single right rotation to make the node balanced.



When node A is inserted into the left of a left sub tree then node C is unbalanced. To make it balanced we have to perform a single right rotation.

Left-Right Rotation: -- if a tree is unbalanced when a node is inserted into the right sub-tree of left subtree then we has to perform LR rotation.

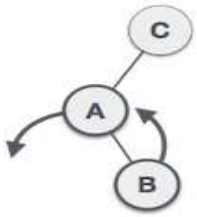
First perform left rotation then again perform right rotation



Node B is inserted on right subtree of node A which is left sub tree of C. that makes node C unbalance.

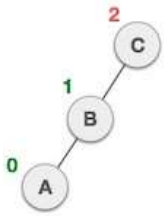
First insert A left = tree is balance

Insert B right of A= tree unbalance so perform LR rotation.

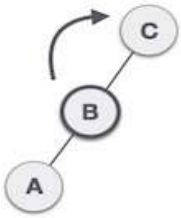


1st perform left rotation on C

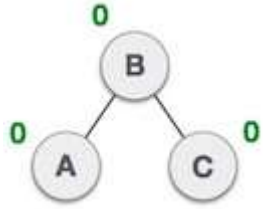
This makes **A**, the left subtree of **B**.



Node **C** is still unbalanced. However now, it is because of the left-subtree of the left-subtree.



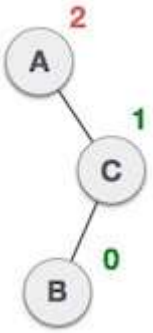
Perform right rotation



The tree is now balanced.

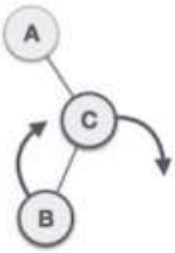
Right left Rotation: -- when node insert left sub tree pf right sub tree which makes tree unbalance

First perform right rotation then perform left rotation.

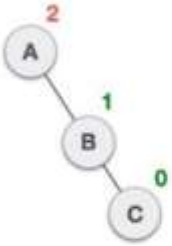


A node has been inserted into left sub tree of A to the right sub tree of C

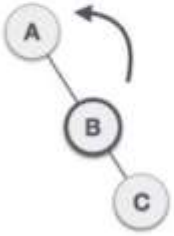
A node inserted left sub tree of right sub tree



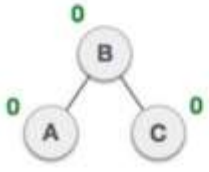
First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.



Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.



A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.



The tree is now balanced.