Trees – V

Height Balanced Trees
(AVL Trees)

If a BST is balanced, one can do insertion, search and deletion in O(log n) time. But if it is skewed, then these operations could take O(n) time. Thus when n is large, it is always of interest to keep the tree balanced as far as possible. One of the first tree balancing algorithms was suggested by two Russian scientists Adel'son-Vel'skii and Landis. Such balanced trees are known by their initials as AVL trees.

An AVL tree is an almost balanced tree. The heights of the left sub tree and the right subtree determine whether the tree is balanced or not.

The height of a node is the longest path length, including itself.

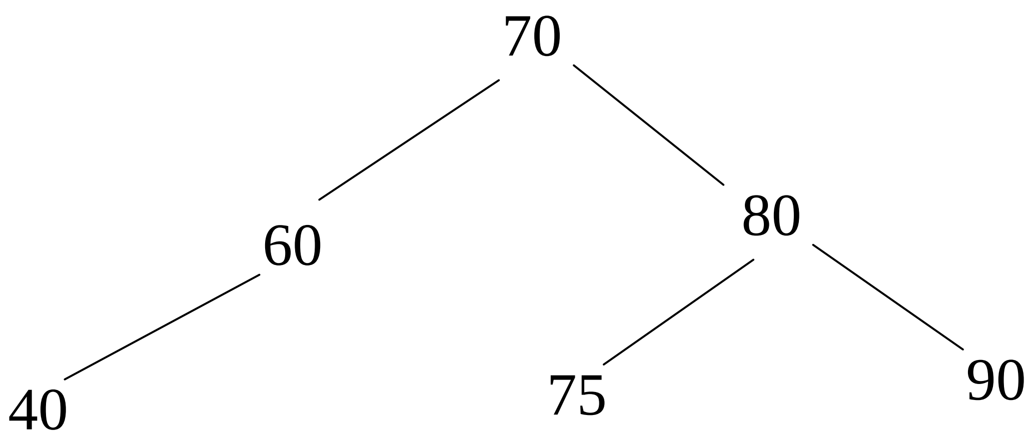
In the following tree the heights of the nodes are indicated below:

Nodes 40, 75, 90 : Height 1
Nodes 60, 80 Height 2
Node 70 : Height 3.

A new term is introduced now, called the Balance factor.

The Balance Factor (B.F.) of any node of a Binary search tree, is the difference between the height of right child node (HR) and the height of the left child node (HL).

B.F. = HR – HL



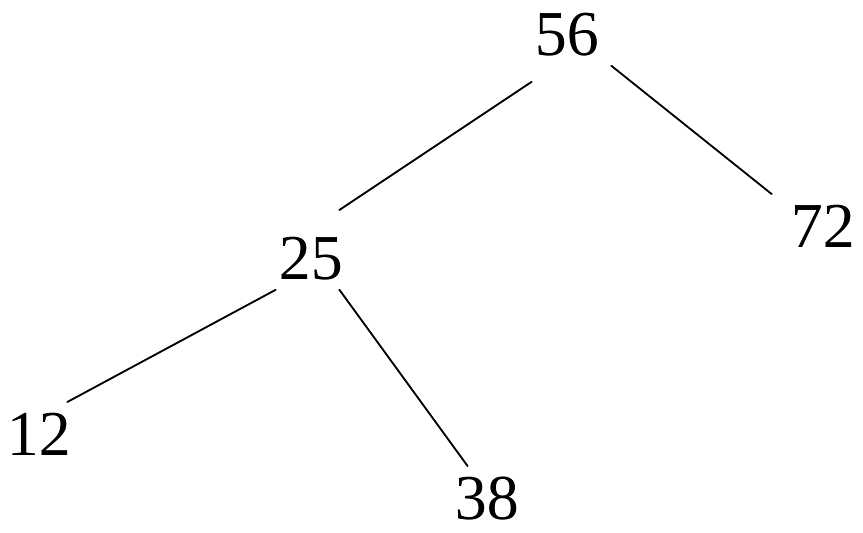
A Height balanced tree (an AVL tree), is one which is
either empty,
or
each node of the tree has a balance factor of 0, -1 or 1.

If BF for any node is 2 or -2 , we say that the tree is unbalanced. The tree can be balanced by appropriate rotation of sub trees around a particular node. If the BF of all nodes remains 0, -1 or1, after a new node is inserted in the tree, nothing needs to be done. However, if the tree gets unbalanced, we have to carry out rotation of subtrees. The rotation will depend on where the new node was inserted in the tree. After a node is inserted, we have to consider 4 separate cases of unbalancing .

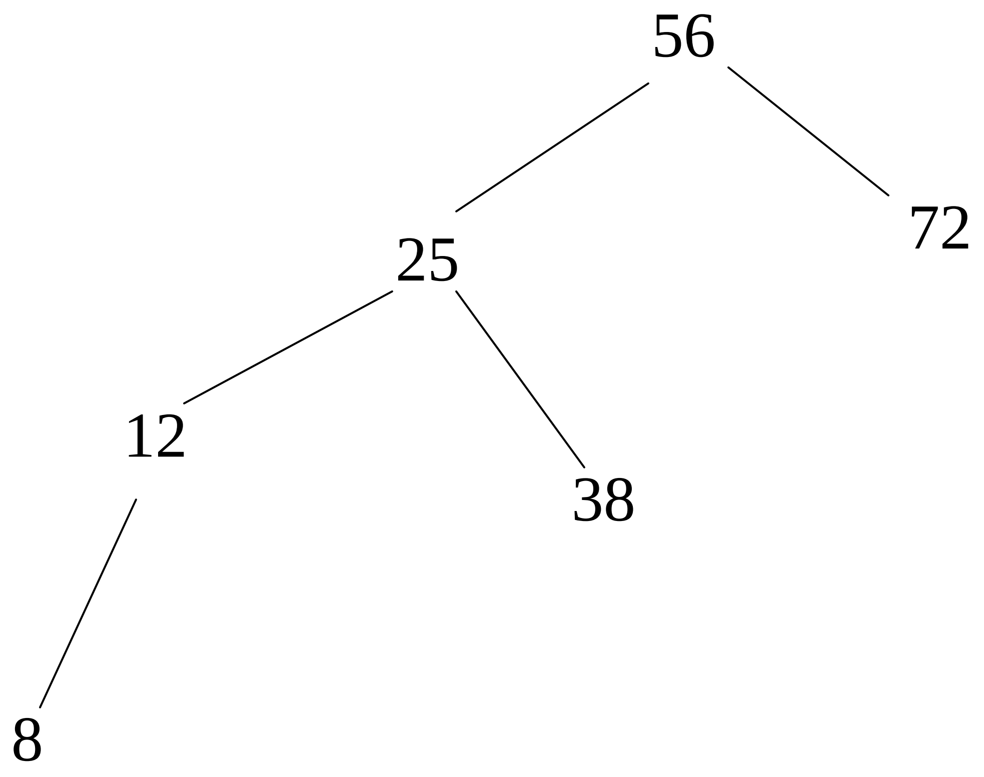
Case LL:

Here we consider the case where the tree gets unbalanced after a node is added to the **left** subtree of the **left** child. The node with BF 2 or -2 must be brought down to a lower level.

Consider the following tree. It is an AVL tree, as each node as BF of 0, -1, or 1.

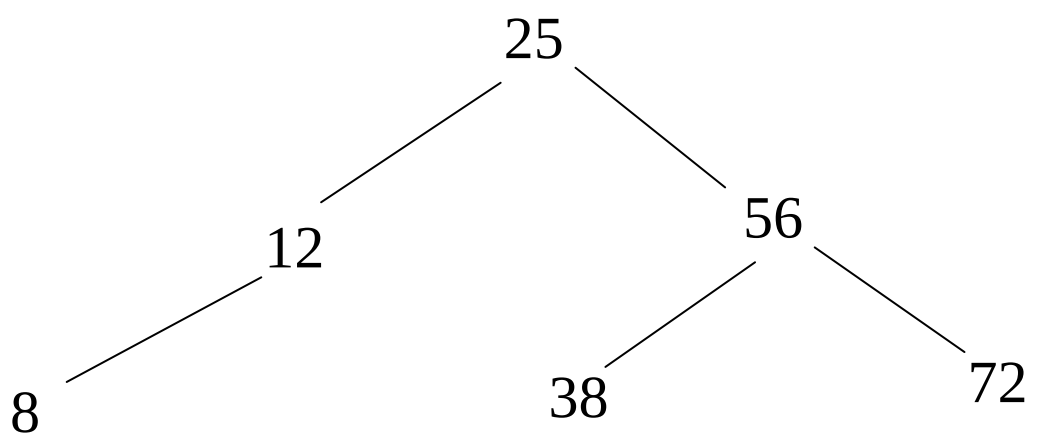


Let us now insert a new node 8 in this tree, which will be in the left subtree of left child (25) of the root (56).

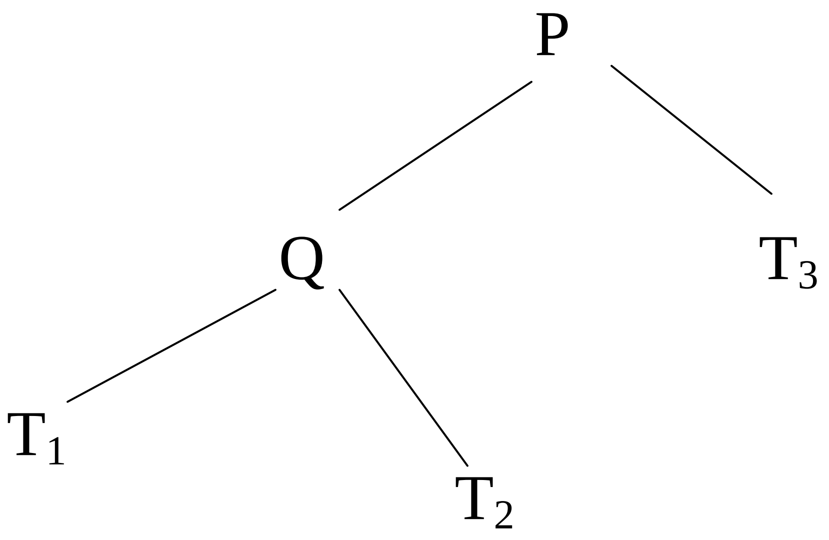


Balance factor BF for 12 and 25 is -1, but for 56 the BF is -2. So the tree is unbalanced. The balance has been disturbed by insertion of a node in the left subtree of the left child 25.

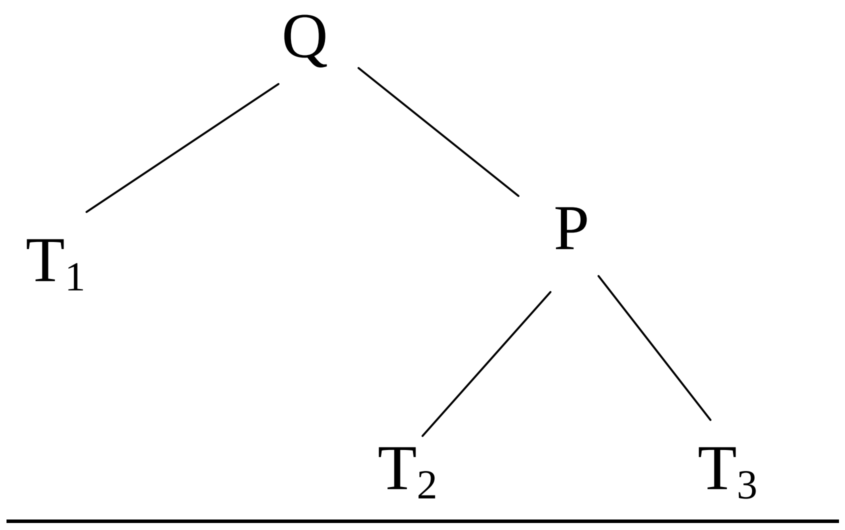
A right rotation of the tree around the node 56, causes node 25 to move up and 56 to move down the tree. To maintain the Binary search tree property, the right subtree of 25 needs to be attached to node 56. The tree is now balanced and is shown below. Node 56 also takes care of node 38.



Now consider the general LL case, where P and Q are nodes, and T_1 , T_2 , T_3 are sub trees. Let us say insertion of a node in T_1 causes the tree to be unbalanced.

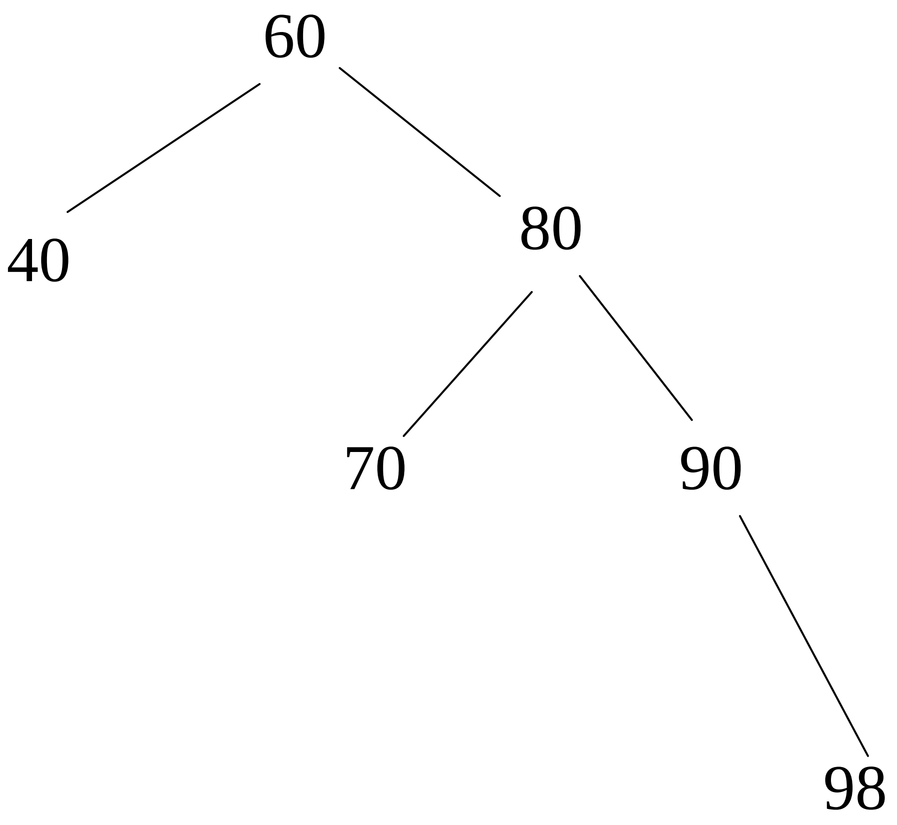


Note that the BF of the nodes in the unbalanced tree are either monotonically increasing or decreasing (NO change in sign). The tree can be balanced by (right rotation) moving P down, and moving Q up. The subtrees are taken care in the following way:

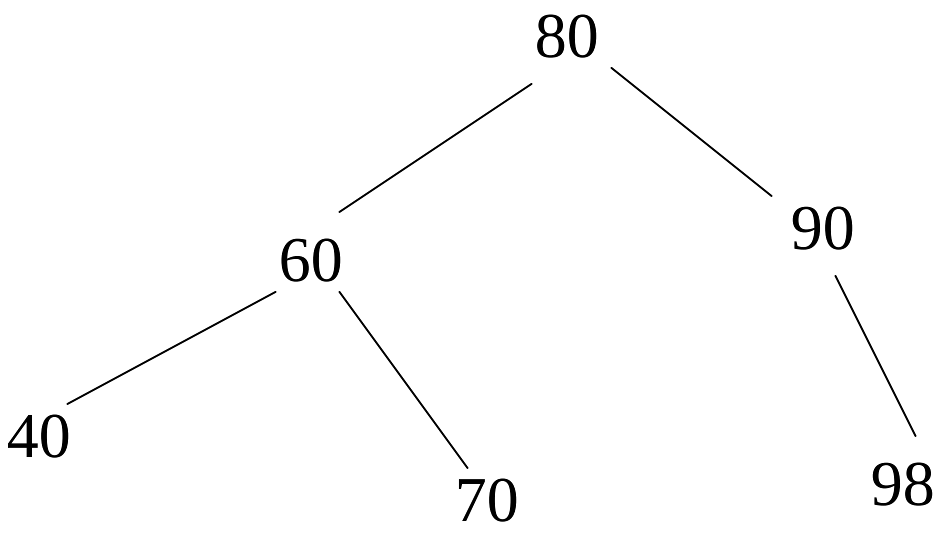


Case RR:

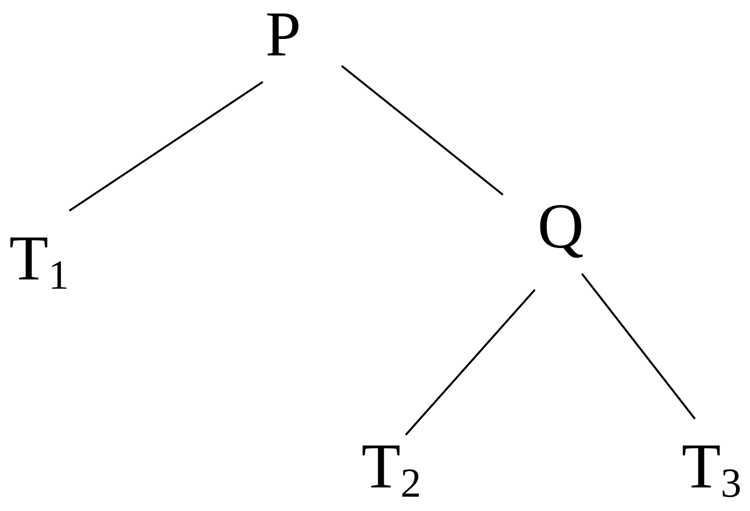
This case is similar to LL case, except that the tree gets unbalanced when a node is added to the **Right** subtree of the **Right** child of the root. Consider the following tree which became unbalanced after insertion of node 98. Node 98 was inserted in the right subtree of the right child 80.



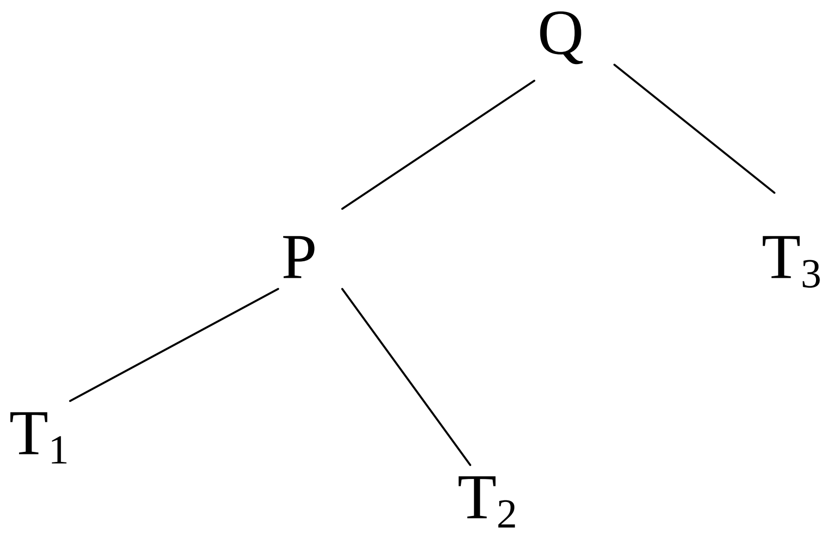
The tree can be balanced by a **left rotation** which causes node 60 to move down and node 80 to move up, as shown



General case: Consider a general case where the tree becomes unbalanced by inserting a node in T_3.



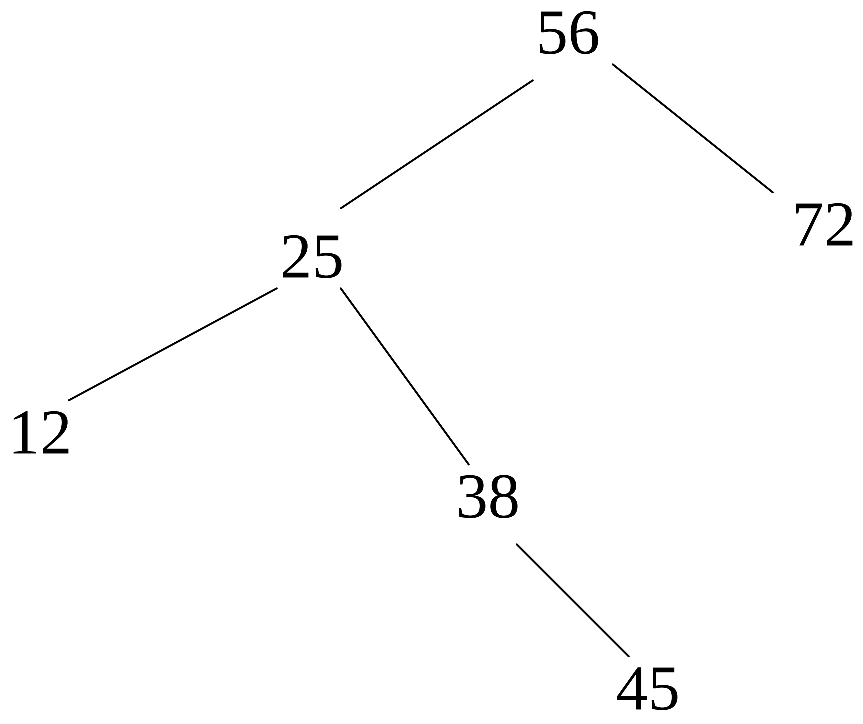
Such a tree is balanced by moving up Q and moving down P (Left rotation). The subtrees are taken care of as follows:



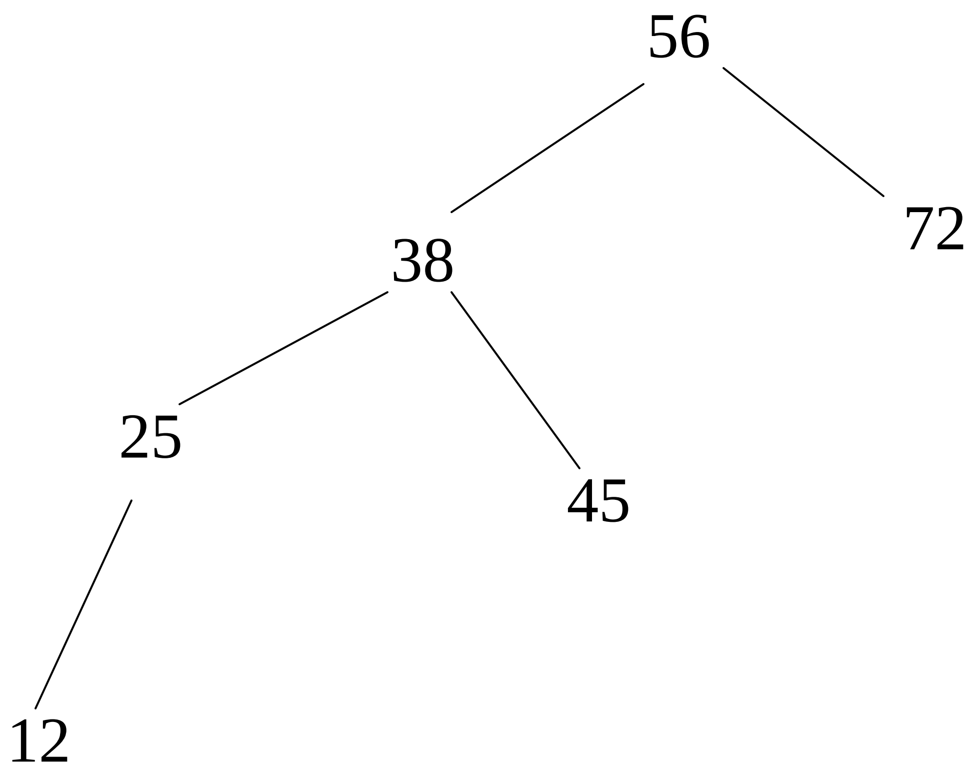
Case RL:

This is a case where the tree gets unbalanced after a node is added to the **right subtree** of the **left child**. Let us check the balance factor in the following tree after node 45 is attached to right subtree of the left node25, . Here node 12 has zero BF, node 25 has BF

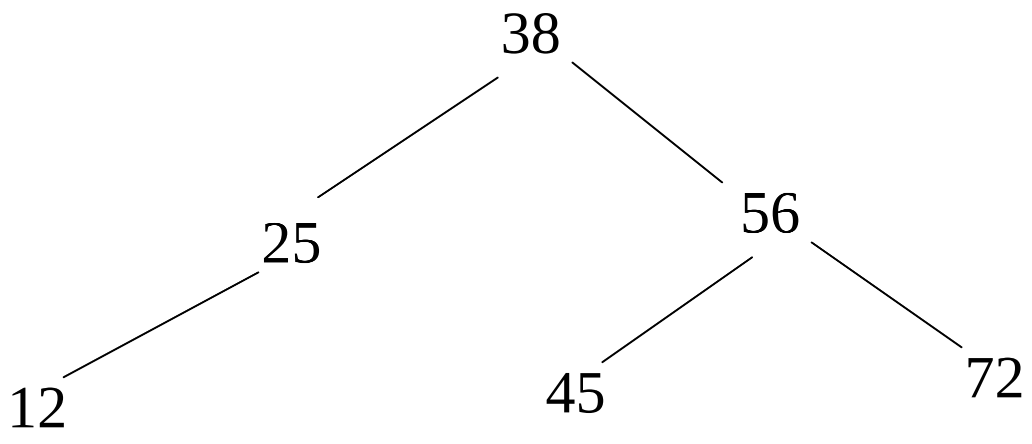
1, and node 56 has BF of -2. Note that along that path, the Balance Factors are going as 0, -1, 2, that is, first decreasing and then increasing.



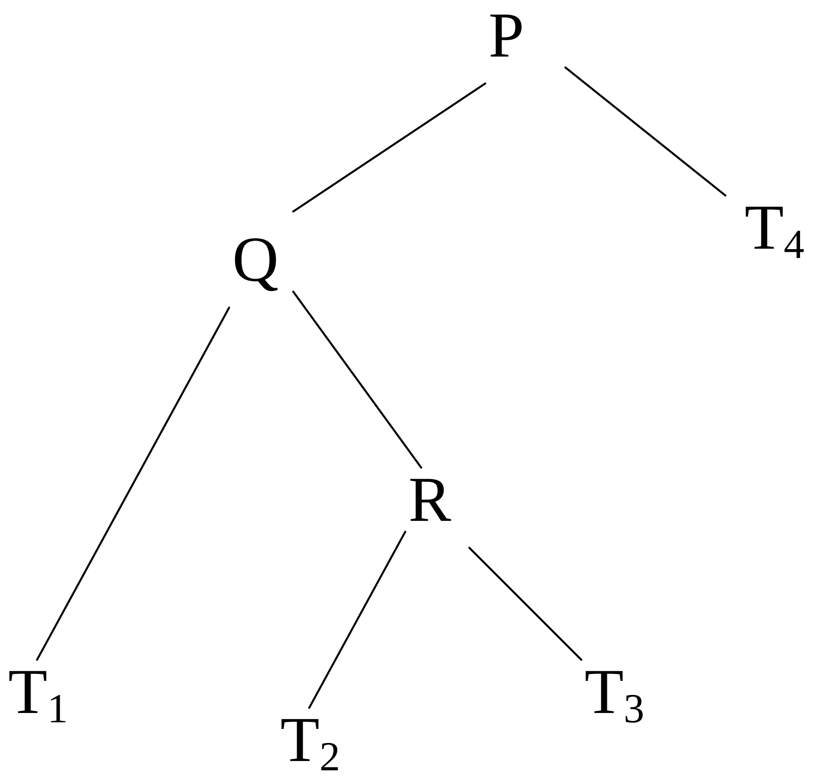
The RL case is more difficult to handle than the LL and RR cases. It needs a double rotation to balance the tree. A left rotation brings up the node 38 as shown below:



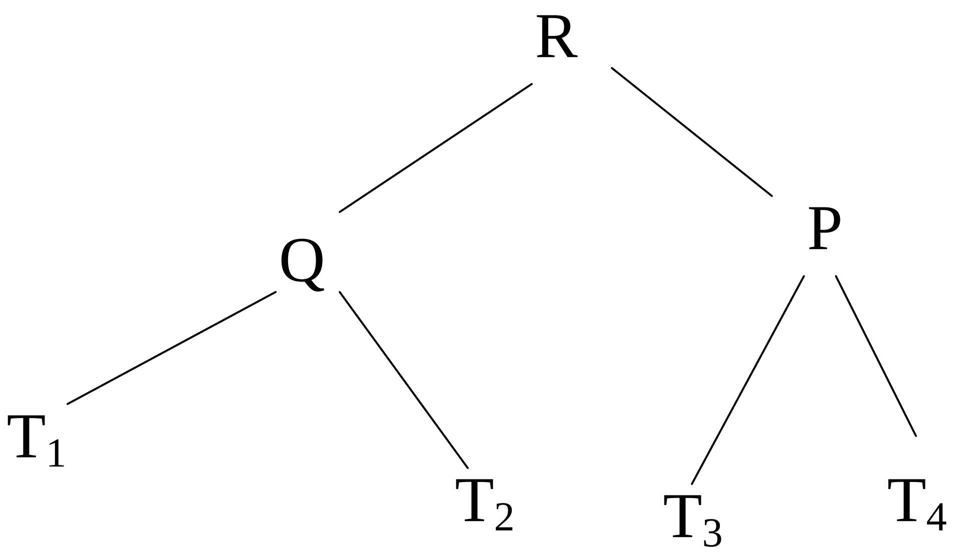
The tree is still unbalanced, but the nodes along the path to the root have the Balance Factors as -1, -1, and -2 (the BF have the same sign). This means a right rotation would be able to balance the tree, as it is similar to the LL case. The right rotation results in the following balanced tree:



Let us now consider the general case, of a tree with nodes P, Q and R with 4 subtrees. The tree gets unbalanced after insertion of a node in T_2 or T_3 .

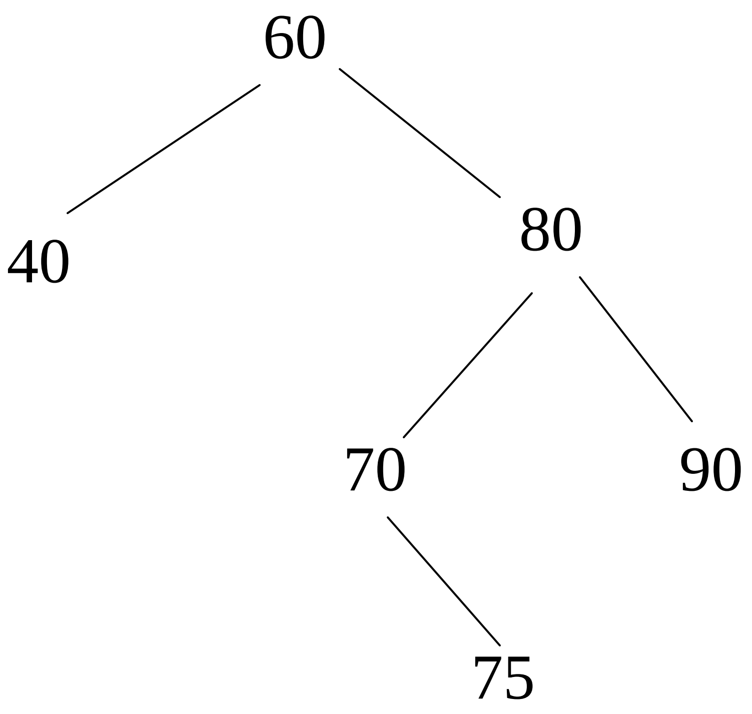


The tree can be balanced by a double rotation. The final effect of the double rotation is shown below. Note that in this case the node R moves up, with nodes P and Q as its children and the subtrees are taken care of in the following way. Also note that the left to right ordering of subtrees T_1 T_2 T_3 T_4 is not disturbed by the balancing process.

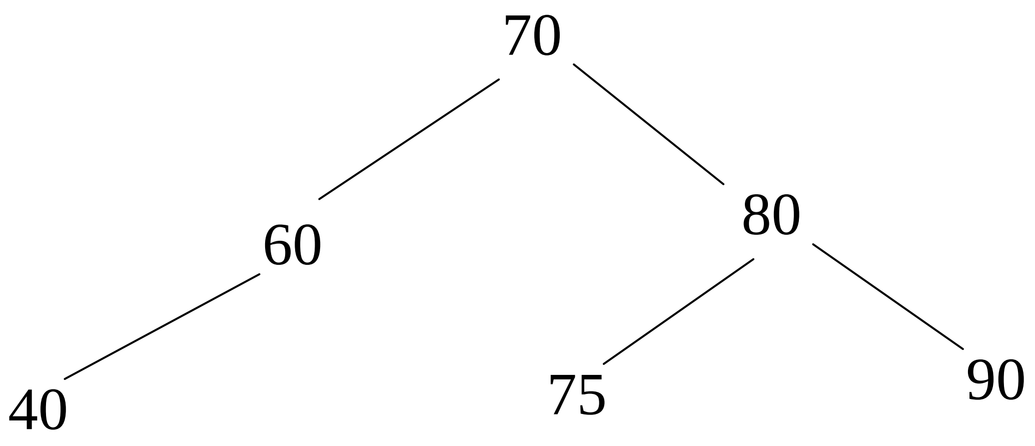


Case LR:

This case is similar to the RL case. The tree gets unbalanced when a node is added to the Left subtree of the Right child. It also needs a double rotation to balance the tree. Consider the following tree which became unbalanced after inserting the node 75 in the left subtree of right child 80. The balance factor of 60 and 80 are of different signs.



The tree is balanced by moving 70 up .Node 60 retains its left subtree, and node 80 retains its right subtree. The left and right subtrees of the root node 70 are taken care of by nodes 60 and 80.


70
60
80
40
75
90