Contents :-

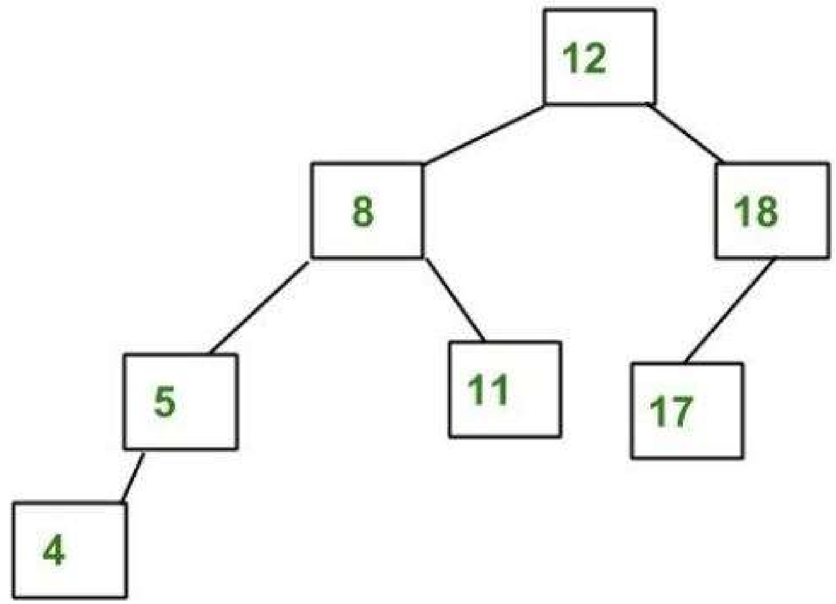
- AVL Tree
- Why AVL Tree
- Height of AVL Tree
- Insertion in AVL Tree
- Rotations
- Deletion in AVL Tree

AVL Tree:-

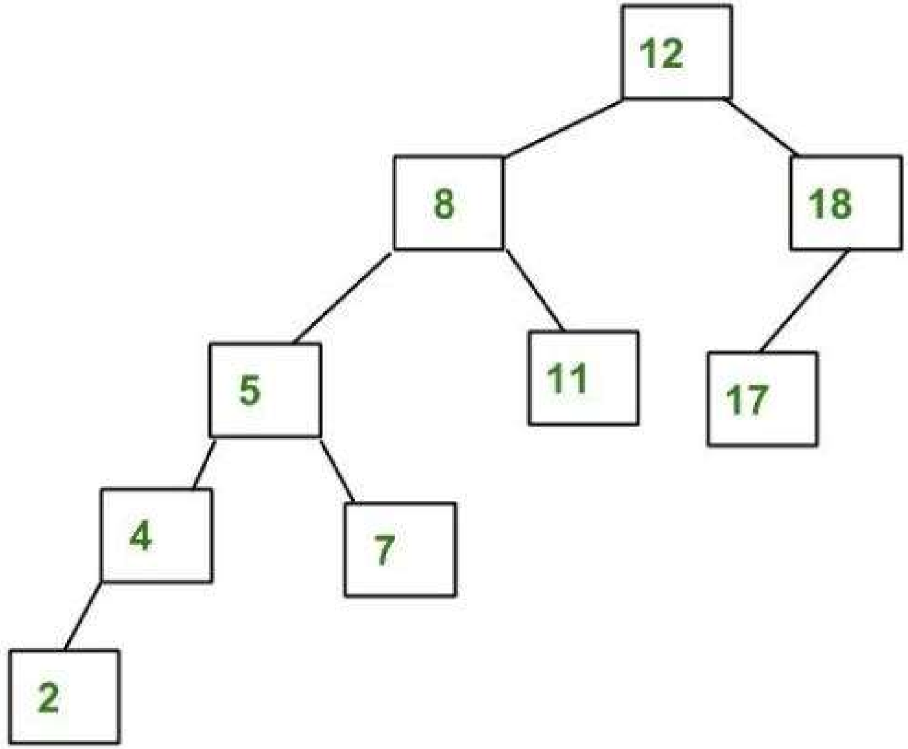
- AVL tree is a self-balancing Binary Search Tree (**BST**).
- where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.
- Balancing Factor in AVL Tree =height(Left) – height(Right)
- The Balancing factor will always be -1,0,+1.

Example of AVL Tree :-

- Balancing Factor =height(Left) – height(Right)
- 4 -> 0-0 =0
- 5 -> 1-0 =1
- 11-> 0-0 =0
- 8 -> 2-1 =1
- 17 -> 0-0 =0
- 18 -> 1-0 =1
- 12 -> 3-2 =1



Example of a Tree that is NOT AVL Tree :-



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 the height of the tree remains **O(log(n))** after every insertion and deletion.
- Then we can guarantee an upper bound of **O(log(n))** for all these operations.
- The height of an AVL tree is always **O(log(n))** where **n** is the number of nodes in the tree.

Height of AVL Tree:-

- The height of a AVL Tree T Storing n keys is O(logn). Why ?
- The minimum number of nodes in AVL tree of height h is n(h).
- We see that n(1)=1 and n(2)=2
- For h>= 3, an AVL tree of height h contains the root node, one subtree of height h-1 and other subtree of height h-1 or h-2.
- i.e. n(h)=1+n(h-1)+n(h-2)

Height of an AVL Tree:-

- Knowing $n(h-1) >= n(h-2)$, we get
- As the height of tree increases the number of nodes can not reduce.
- $n(h)=1+n(h-1)+n(h-2)$
- $n(h)=1+n(h-2)+n(h-2) > 2n(h-1)$
- $n(h) > 2n(h-2)$

  $> 4n(h-4)$

  $>6n(h-8)$……

  $>2^{i}(h-2i)$
- When $i=h/2 -1$
- We get $n(h) > 2^{h/2 -1} n(2) = 2^{h/2}$
- Taking $\log h < 2 \log n(h)$

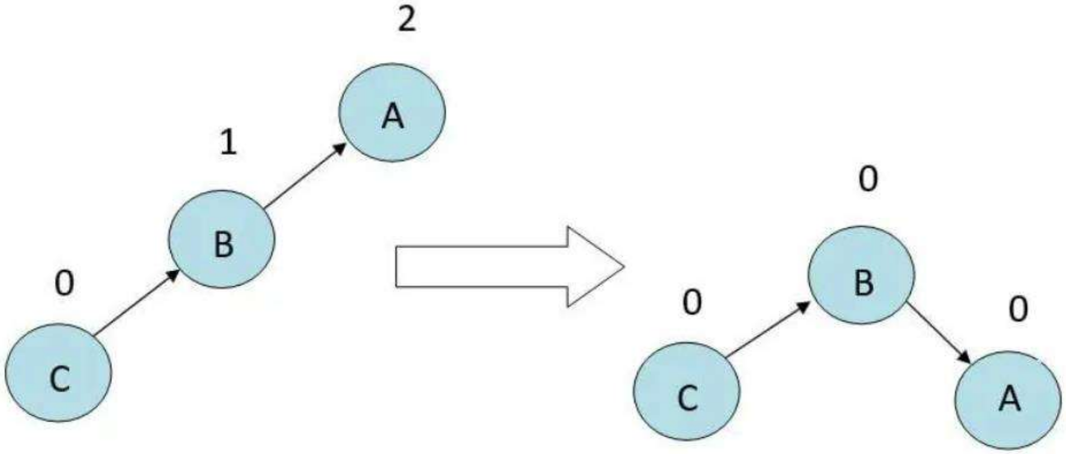
Insertion in AVL Tree:-

- To make sure that the given tree remains AVL after every insertion.
- We perform the following LL rotation, RR rotation, LR rotation, and RL rotation.

a. Left – Left Rotation
b. Right – Right Rotation
c. Right – Left Rotation
d. Left – Right Rotation

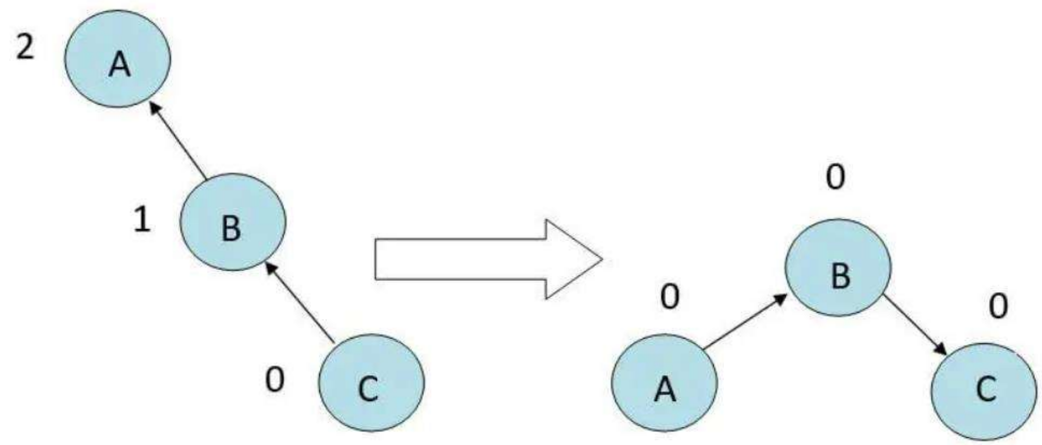
Left –Left Rotation

- This rotation is performed when a new node is inserted at the left child of the left subtree.
- LL rotation is also known as clockwise rotation.



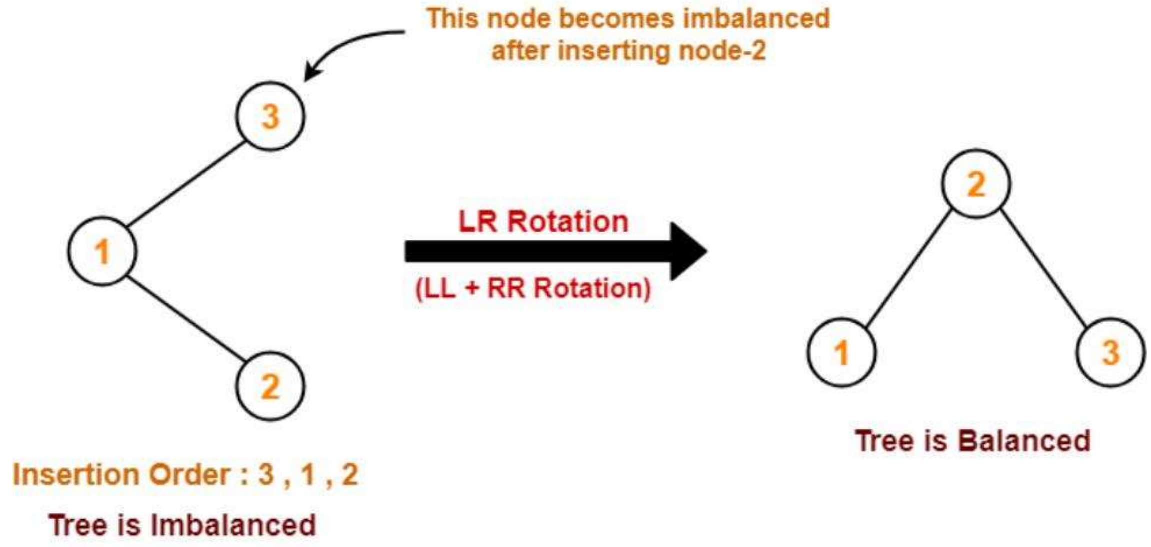
Right- Right Rotation

- This rotation is performed when a new node is inserted at the right child of the right subtree.
- LL rotation is also known as anti-clockwise rotation.



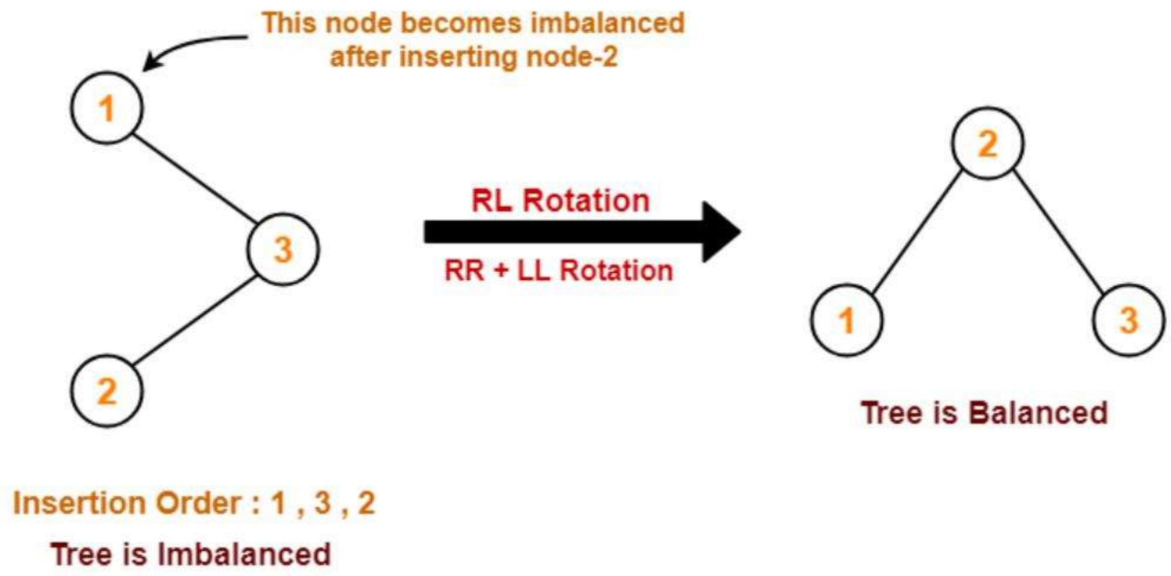
Left-Right Rotation:-

- This rotation is performed when a new node is inserted at the left child of the right subtree.
- Left-Right Rotation is the combination of RR rotation and LL rotation
- At first, RR rotation is performed on the subtree then, LL rotation is performed on the part of the full tree from inserted node to the first node



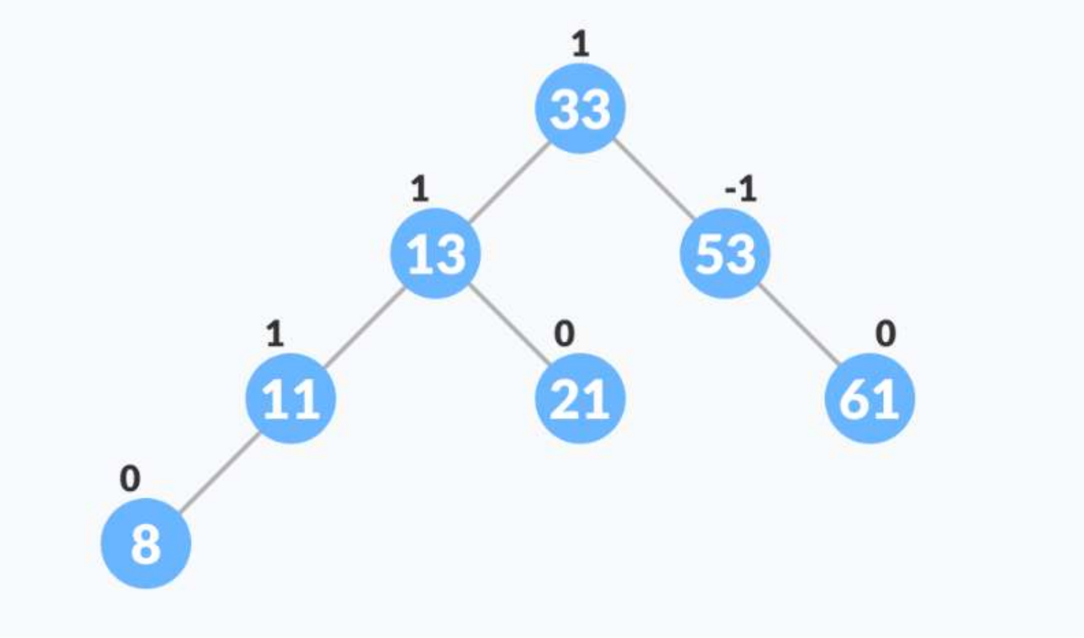
Right-Left Rotation:-

- This rotation is performed when a new node is inserted at the right child of the left subtree
- In this case, the first LL rotation is performed on the subtree where the change has been made
- the RR rotation is performed on the pa of the full tree from the inserted node to the top of the tree, that is, the first node



Example of Insertion:-

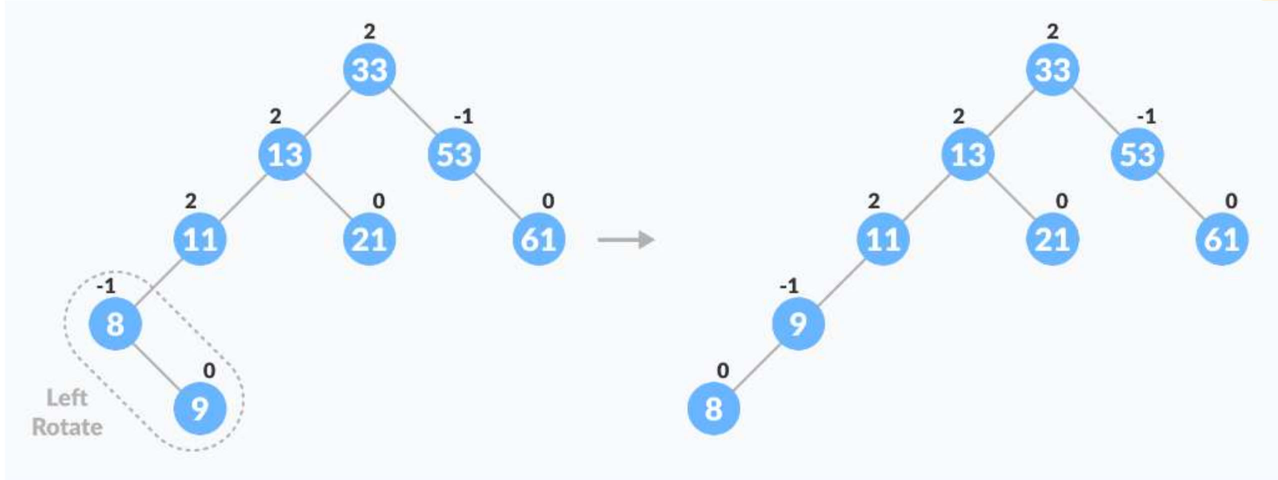
- We need to insert 9



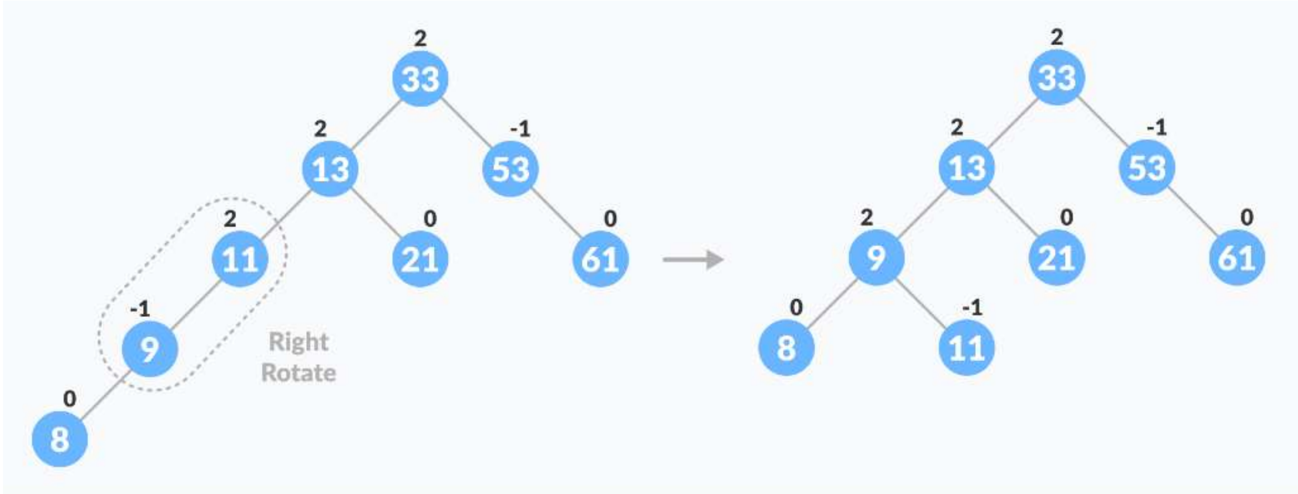
Initial tree for insertion

Example of insertion:-

- If balanceFactor > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation.
- If newNodeKey < leftChildKey do right rotation.
- Else, do left-right rotation.

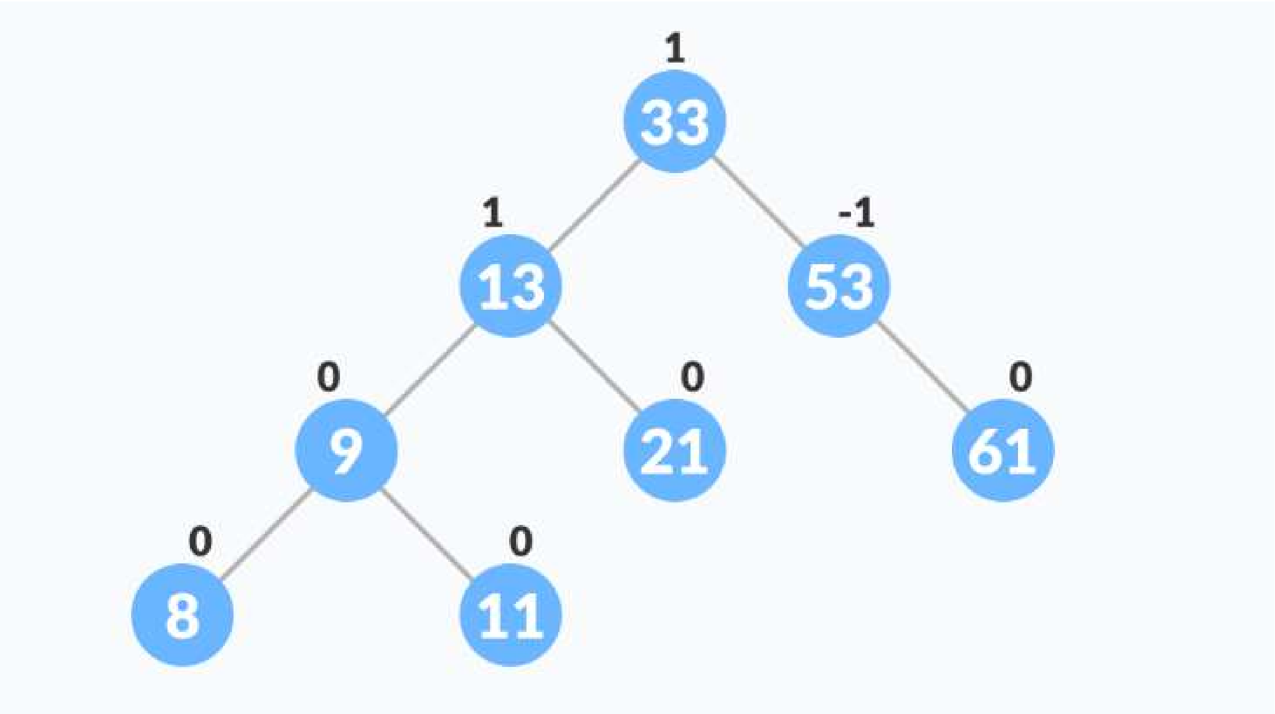


Balancing the tree with rotation



Balancing the tree with rotation

Example of insertion:-

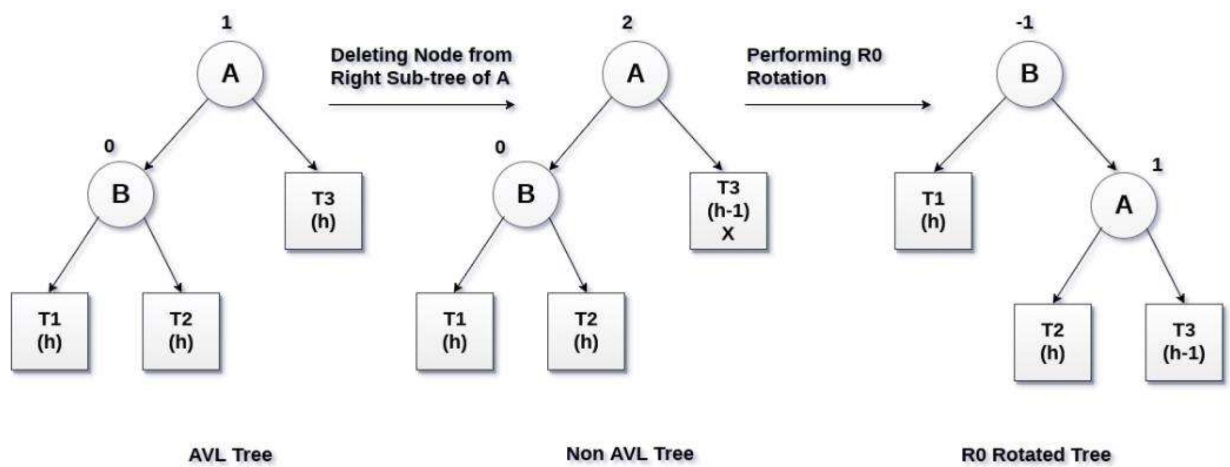


Deletion in AVL Tree :-

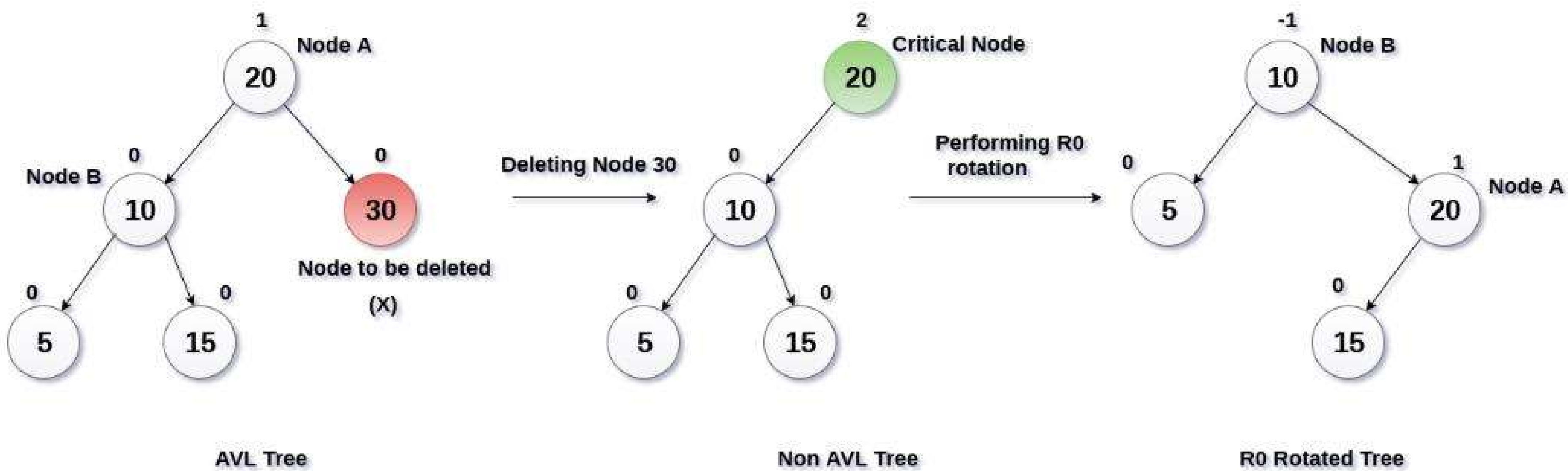
- Deleting a node from an AVL tree is similar to that in a binary search tree
- Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness.
- For this purpose, we need to perform rotations. The two types of rotations are L rotation and R rotation.
- Here, we will discuss R rotations. L rotations are the mirror images of them.
- If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied.
- if the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.
- Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three different situations:

R 0 rotation (Node B has BF 0)

- If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.
- The critical node A is moved to its right and the node B becomes the root of the tree with T1 as its left sub-tree.

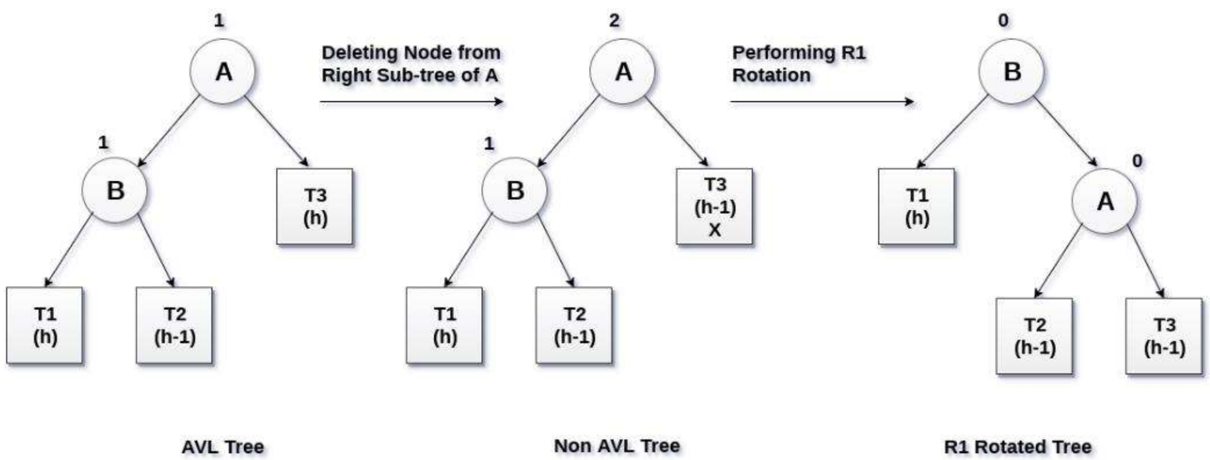


R0 rotation (Node B has BF 0)

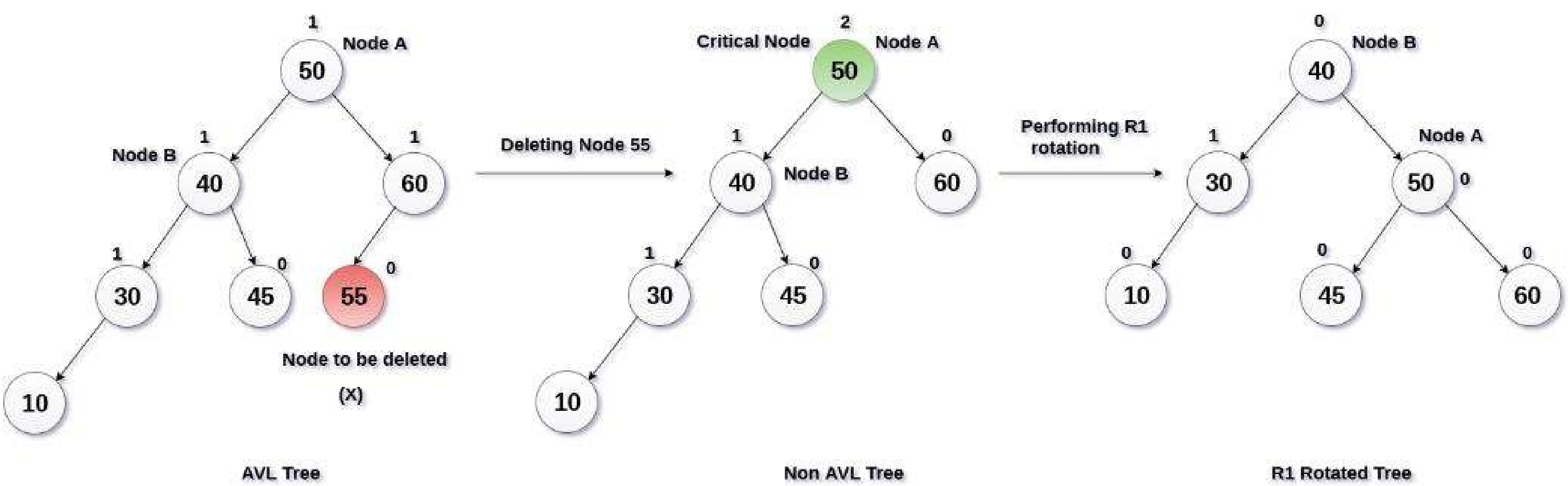


R1 Rotation (Node B has BF 1)

- R1 Rotation is to be performed if the balance factor of Node B is 1.
- In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively.
- T1 is to be placed as the left sub-tree of the node B.

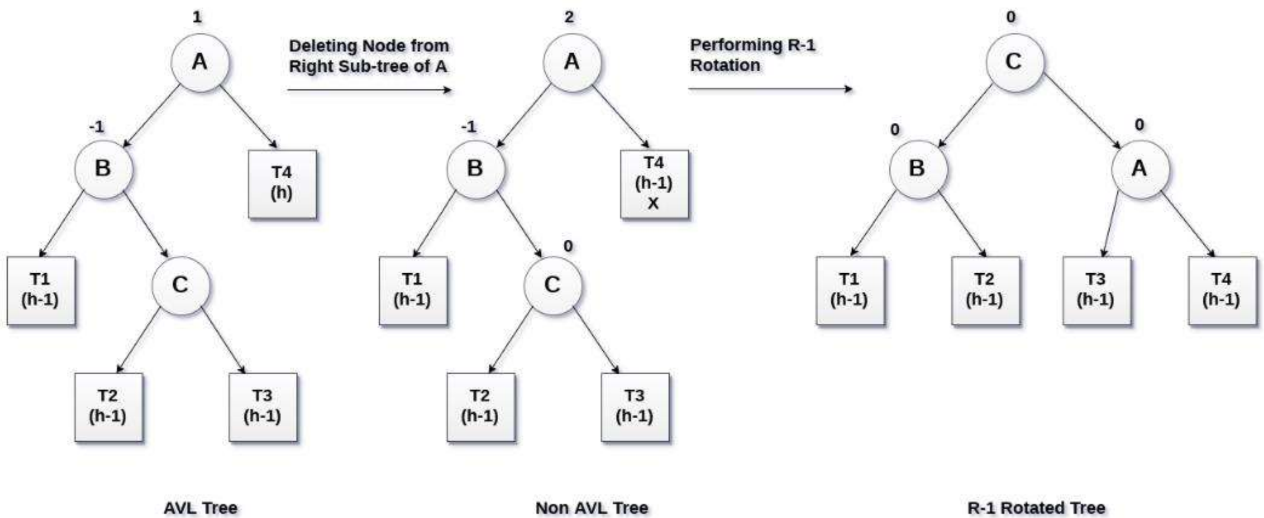


R1 Rotation (Node B has BF 1)

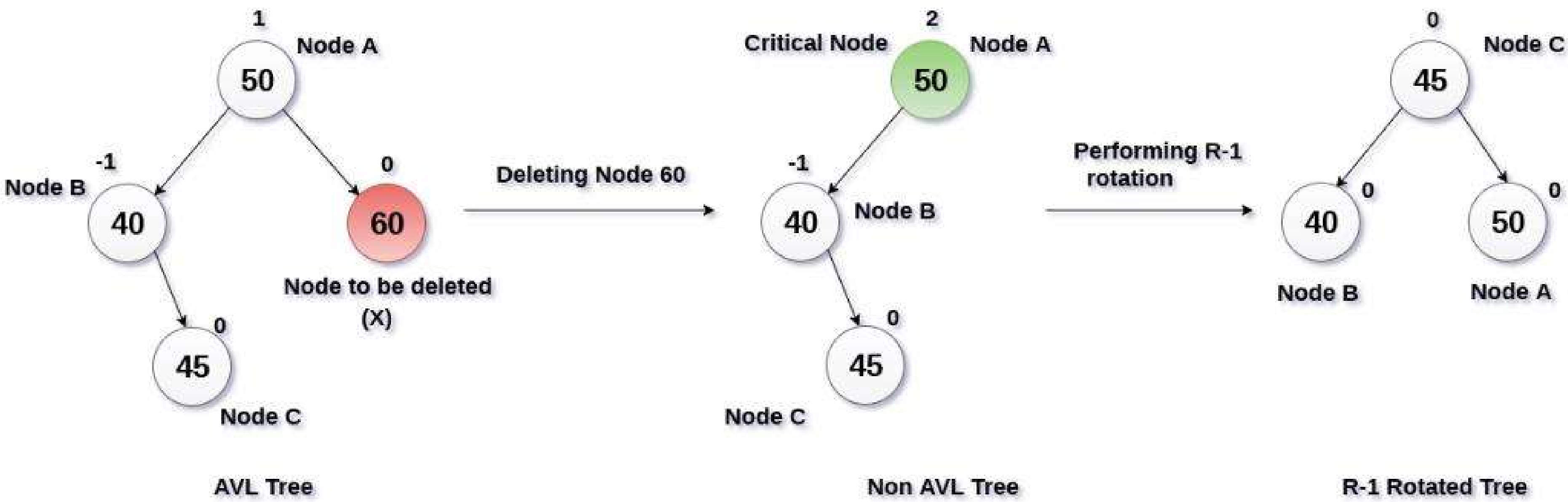


R-1 Rotation (Node B has BF -1)

- R-1 rotation is to be performed if the node B has balance factor -1.
- This case is treated in the same way as LR rotation.
- In this case, the node C, which is the right child of node B.
- becomes the root node of the tree with B and A as its left and right children respectively.



R-1 Rotation (Node B has BF -1)



Thankyou !