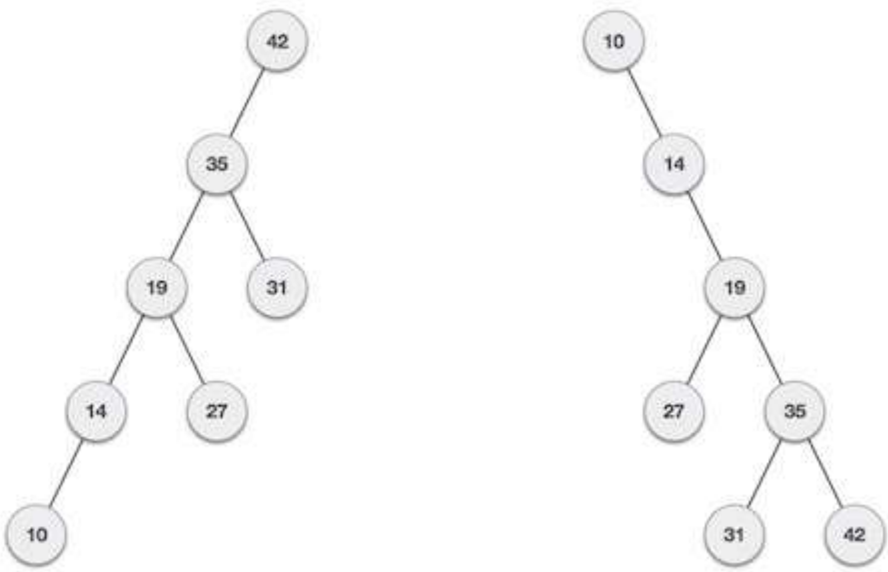
What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −

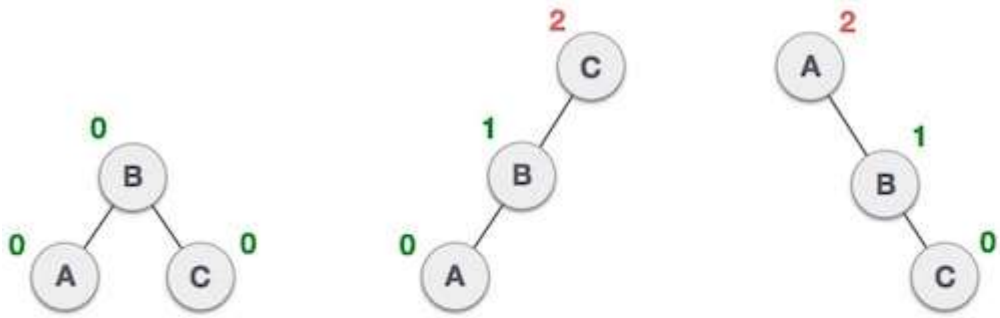


If input 'appears' non-increasing manner   If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

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**.

Here we see that the first tree is balanced and the next two trees are not balanced −



Balanced   Not balanced   Not balanced

In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

```
BalanceFactor = height(left-sutree) − height(right-sutree)
```

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

AVL Rotations

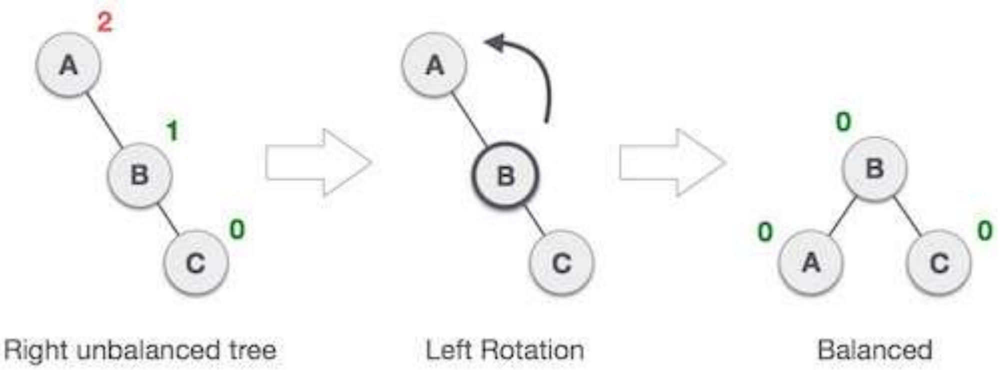
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- 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

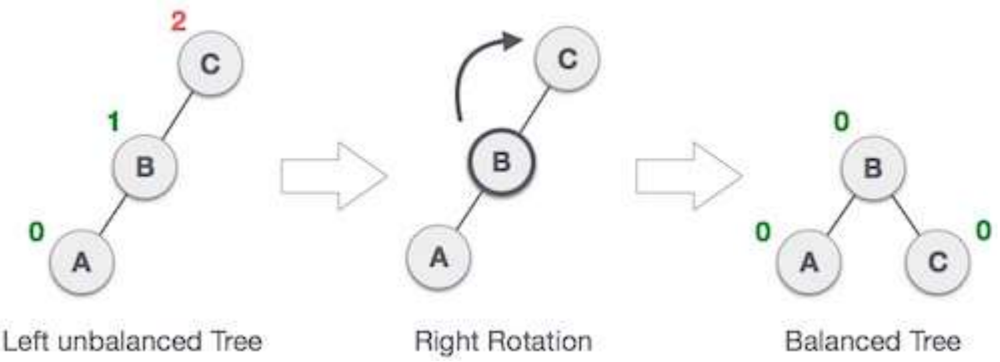
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

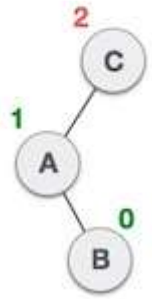
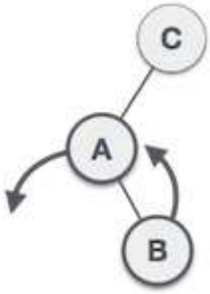
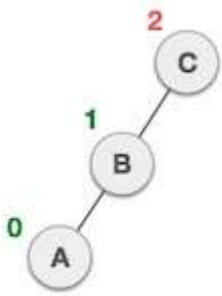
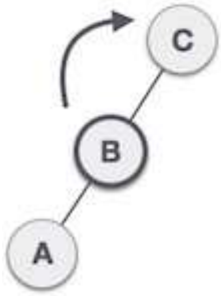
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

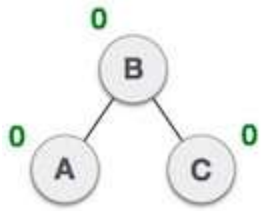


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

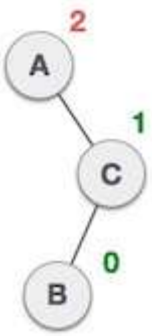
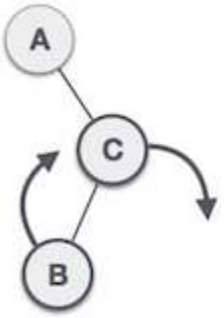
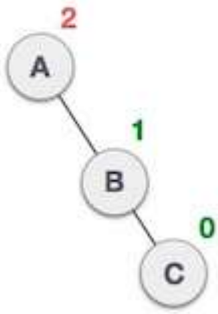
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right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

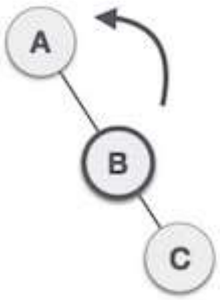
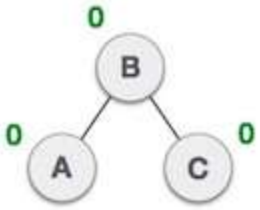
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.

	The tree is now balanced.

Right-Left Rotation

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

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	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.

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.

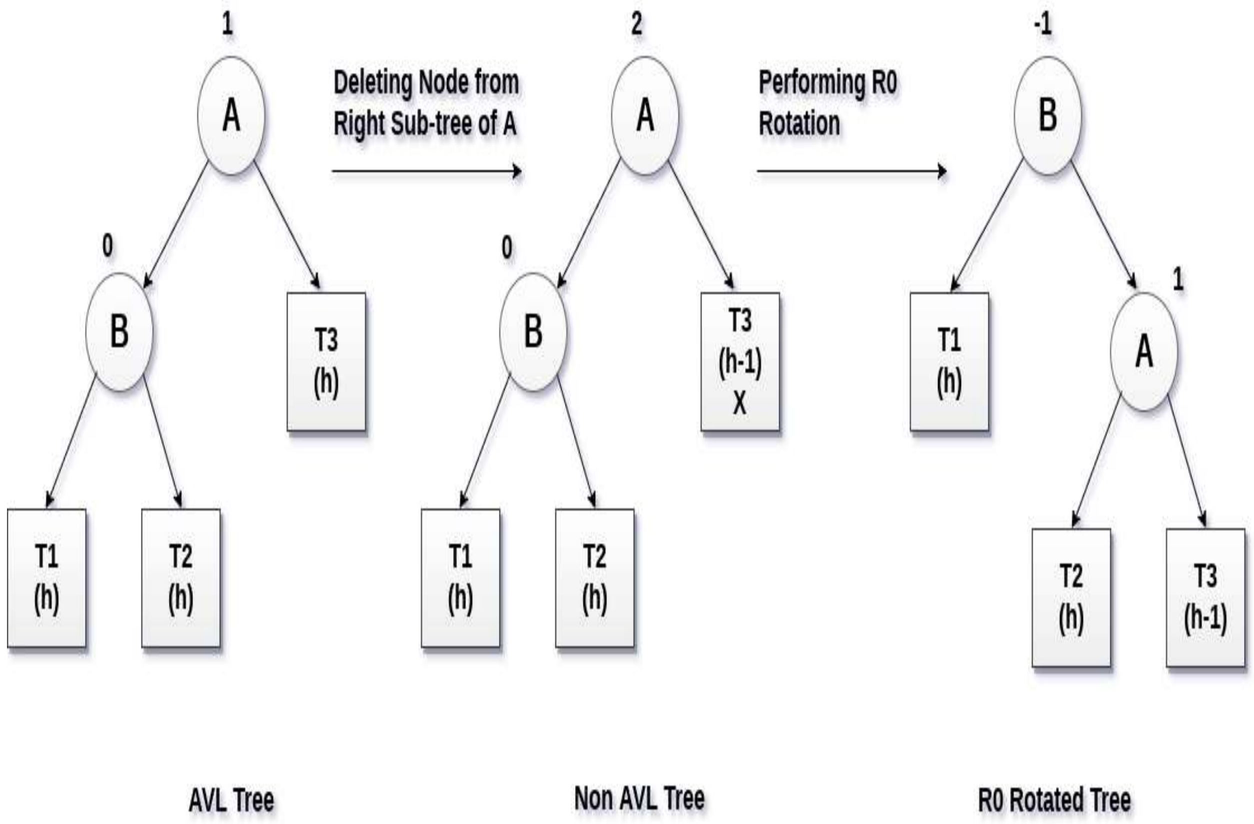
If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

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:

R0 rotation (Node B has balance factor 0)

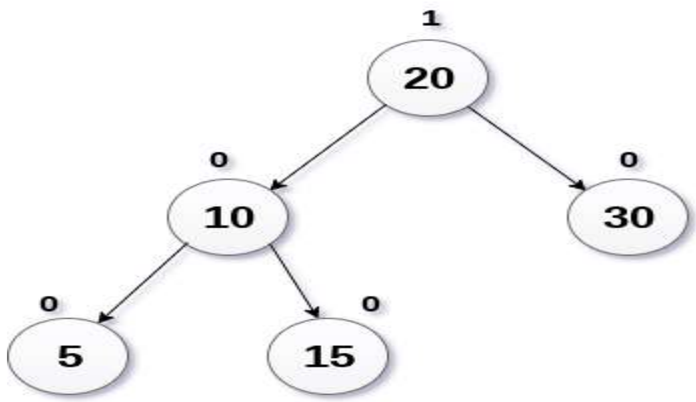
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

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. The sub-trees T2 and T3 becomes the left and right sub-tree of the node A. the process involved in R0 rotation is shown in the following image.



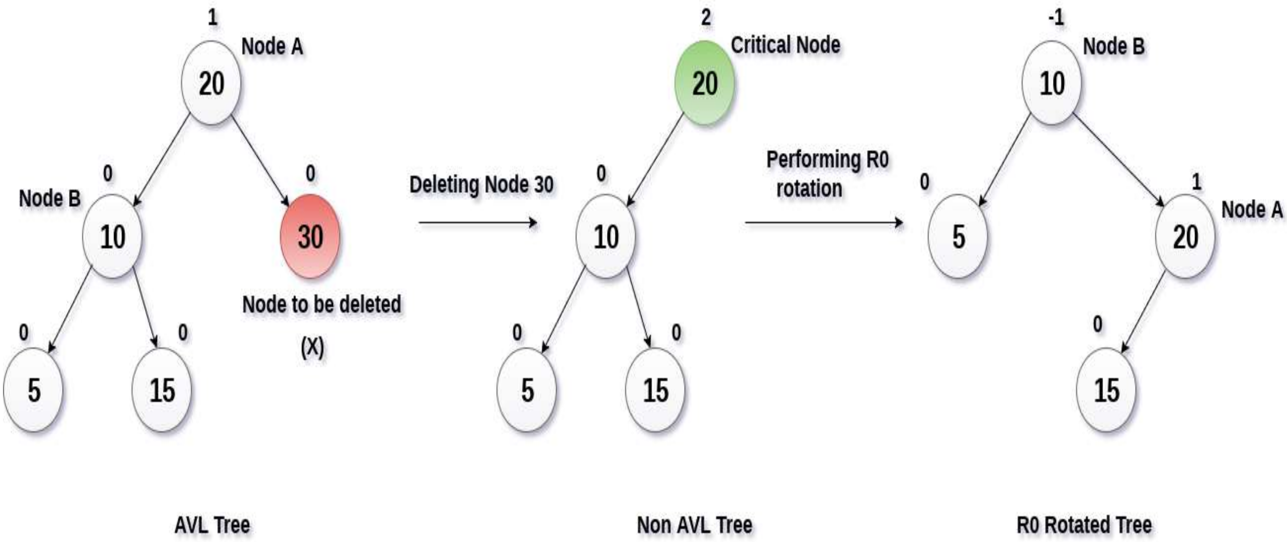
Example:

Delete the node 30 from the AVL tree shown in the following image.



Solution

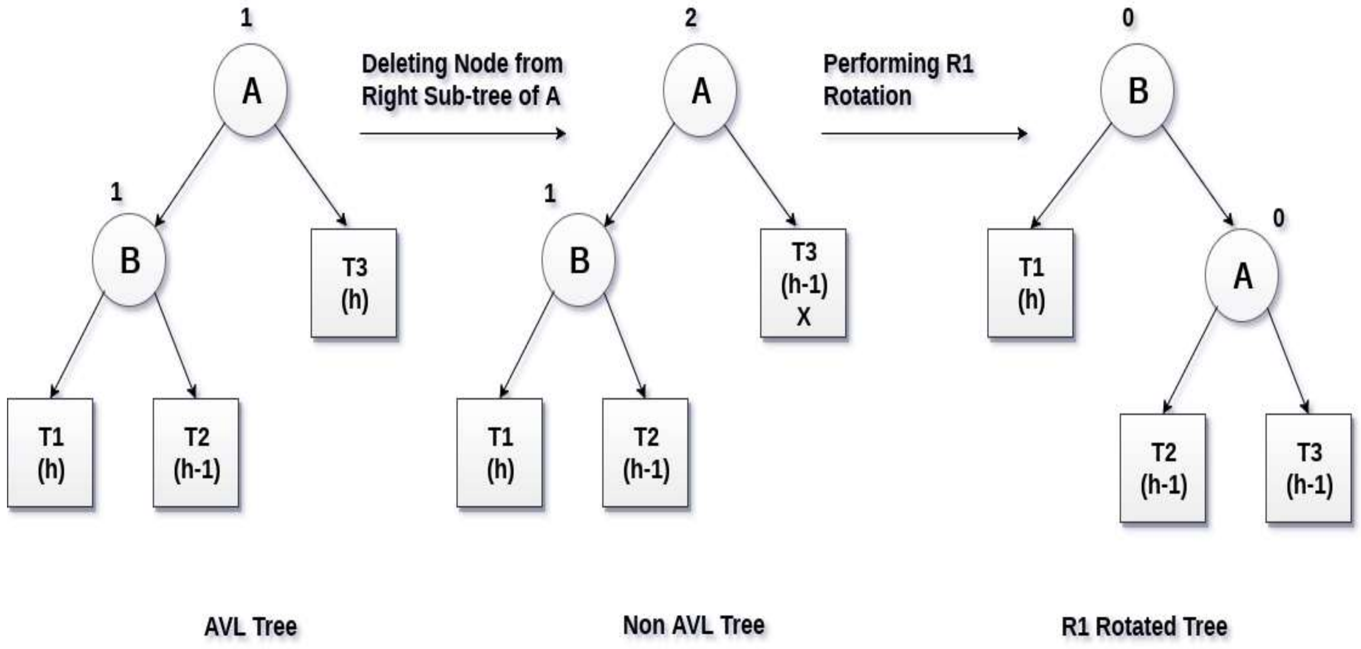
In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



R1 Rotation (Node B has balance factor 1)

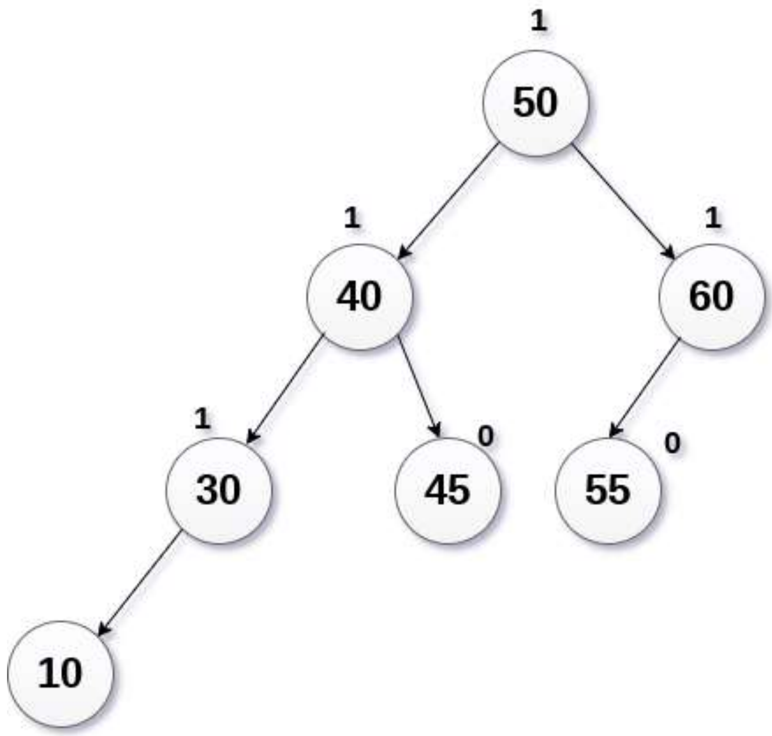
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.

The process involved in R1 rotation is shown in the following image.



Example

Delete Node 55 from the AVL tree shown in the following image.

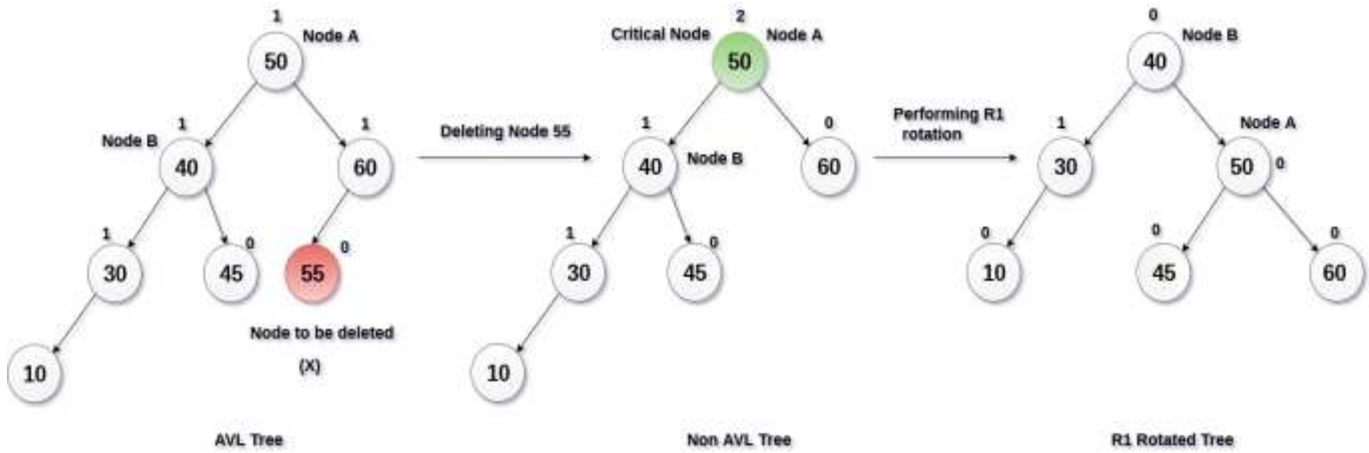


AVL Tree

Solution :

Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. node A which becomes the critical node. This is the condition of R1 rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A (i.e. 45).

The process involved in the solution is shown in the following image.

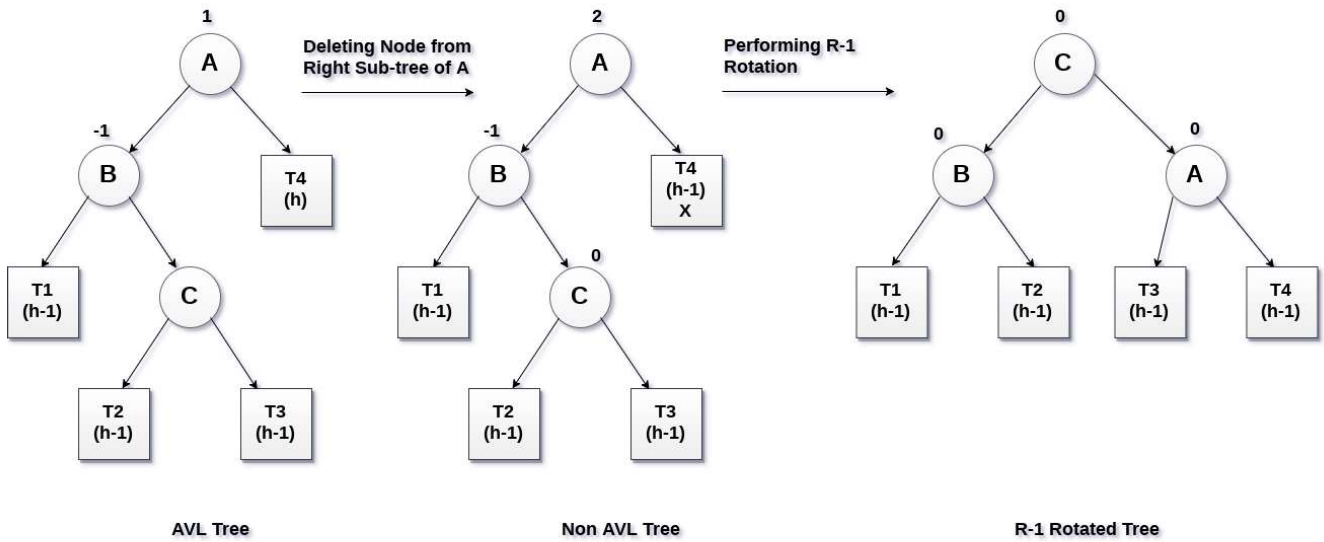


R-1 Rotation (Node B has balance factor -1)

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.

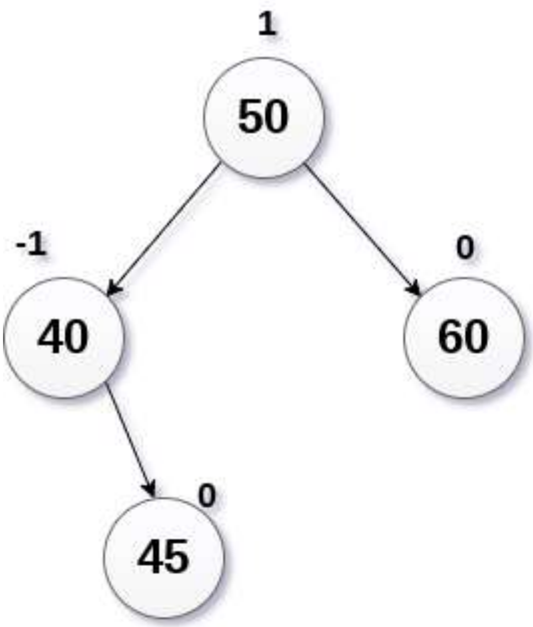
The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A.

The process involved in R-1 rotation is shown in the following image.



Example

Delete the node 60 from the AVL tree shown in the following image.



Solution:

in this case, node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e. 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child.