AVL (Adelson-Velsky and Landis) Tree is a self-balancing binary search tree that can perform certain operations in logarithmic time. It exhibits height-balancing property by associating each node of the tree with a balance factor and making sure that it stays between -1 and 1 by performing certain tree rotations.

This property prevents the Binary Search Tree from getting skewed, thereby achieving a minimal height tree that provides logarithmic time complexity for some major operations such as searching.

The height of the AVL tree is always balanced. The height never grows beyond log N, where N is the total number of nodes in the tree.

Introduction

In our fast-paced daily lives, we make use of a lot of different data structures and algorithms without even noticing them. For example, consider a scenario that you wish to call someone from your contact list that contains a ton of data.

For this, you need to find the phone number of that individual by using a searching process. This is internally implemented using specific data structures and uses particular algorithms to provide you with the best results in an efficient manner.

This is required as the faster the search, the more convenience you get, and the faster you can connect with other people.

With time, this searching process was gradually improved by implementing and developing new data structures that eradicate or reduce the limitations of the previously used methods. One such data structure is AVL Trees. It was developed to reduce the limitations of the searching process implemented using a non-linear data structure known as Binary Search Trees.

Now, you may ask what exactly was the limitation and how AVL Trees overcome it, providing us with a better searching process in terms of efficiency. For this, let's take a look at the Binary Search Trees (BSTs) search process, what are the limitations in this process and how the AVL Trees overcome them. Refer this article to revise searching using Binary Search Trees

What is an AVL Tree?

AVL Tree, named after its inventors Adelson-Velsky and Landis is a special variation of Binary Search Tree which exhibits self-balancing property i.e., AVL Trees automatically attain the minimal possible height of the tree after the execution of any operation. The AVL Trees implement the self-balancing property by attaching extra information known as the balance factor to each node of the tree, then verifying that the balance factor for all the nodes of the tree follows certain criteria **(Balancing Criteria)** upon the execution of any operation that affects the height of the tree, and finally applying certain **Tree Rotations** to maintain this criterion of height-balancing.

The Criterion of height balancing is a principle that determines whether a Binary Search Tree is unbalanced (skewed) or not. It states that:

Tip: A Binary Search Tree is considered to be balanced if any two sibling subtrees present in the tree don't differ in height by more than one level i.e., the difference between the height of the left subtree and the height of the right subtree for all the nodes of the tree should not exceed unity. If it exceeds unity, then the tree is known as an unbalanced tree.

Since skewed or unbalanced BSTs provide inefficient search operations, AVL Trees prevent unbalancing by defining what we call a balance factor for each node. Let's look at what exactly is this balance factor.

Highlights:

1. AVL Trees were developed to achieve logarithmic time complexity in BSTs irrespective of the order in which the elements were inserted.
2. AVL Tree implemented a Balancing Criteria (For all nodes, the subtrees' height difference should be at most 1) to overcome the limitations of BST.
3. It maintains its height by performing rotations whenever the balance factor of a node violates the Balancing Criteria. As a result, it has self-balancing properties.
4. It exists as a balanced BST at all times, providing logarithmic time complexity for operations such as searching.

Balance Factor

The balance factor in AVL Trees is an additional value associated with each node of the tree that represents the height difference between the left and the right sub-trees of a given node. The balance factor of a given node can be represented as:

balance_factor = (Height of Left sub-tree) - (Height of right sub-tree)

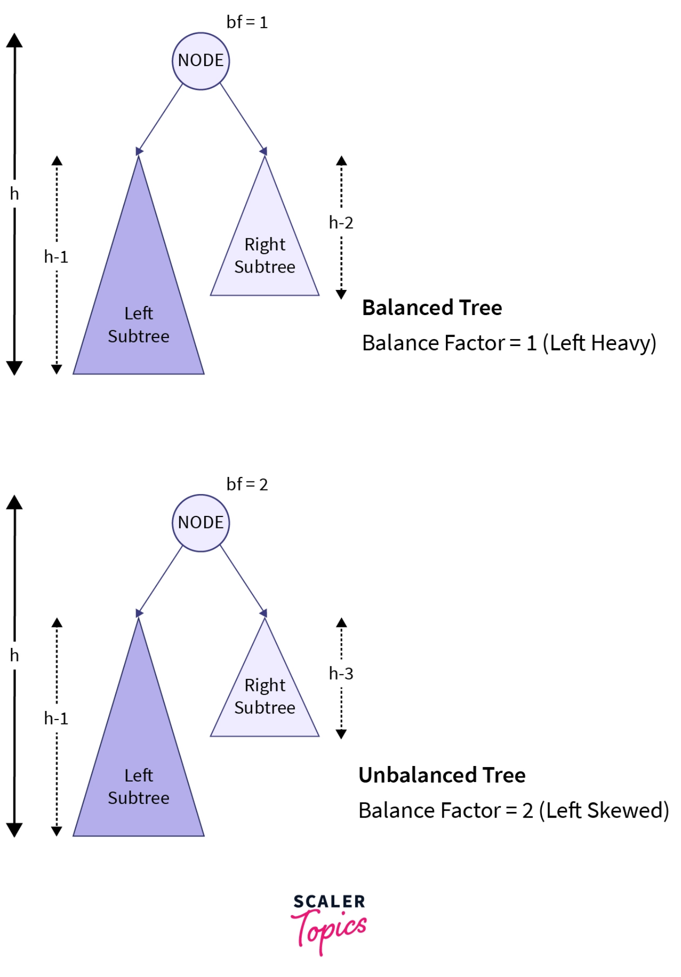
Or mathematically speaking,

$$bf = h_l - h_r$$

where bf is the balance factor of a given node in the tree, hl represents the height of the left subtree, and hr represents the height of the right subtree.

In the balanced tree example of the above illustration, we can observe that the height of the left subtree (h-1) is one greater than the height of the right subtree (h-2) of the highlighted node i.e., the given node is left-heavy having the balance factor of positive unity.

Since the balance factor of the node follows the Balancing Criteria (height difference should be at most unity), the given tree example is considered as a balanced tree.


bf = 1
NODE
h
h-1
Left Subtree
Right Subtree
h-2
Balanced Tree
Balance Factor = 1 (Left Heavy)
bf = 2
NODE
h
h-1
Left Subtree
Right Subtree
h-3
Unbalanced Tree
Balance Factor = 2 (Left Skewed)
SCALER Topics

Now, in the unbalanced tree example, we can observe that the tree is left-skewed i.e., the height of the left subtree is much greater than that on the right subtree. This is clearly an unbalanced tree as it is highly skewed. This is also indicated by the balance factor of the node as it doesn't follow the Balancing Criteria.

Hence, AVL Trees make use of the balance factor to check whether a given node is left-heavy (height of left sub-tree is one greater than that of right sub-tree), balanced, or right-heavy (height of right sub-tree is one greater than that of left sub-tree).

Hence, using the balance factor, we can find an unbalanced node in the tree and can locate where the height-affecting operation was performed that caused the imbalance of the tree.

NOTE: Since the leaf nodes don't contain any subtrees, the balance factor for all the leaf nodes present in the Binary Search Tree is equal to 0.

Upon the execution of any height-affecting operation on the tree, if the magnitude of the balance factor of a given node exceeds unity, the specified node is said to be unbalanced as per the Balancing Criteria. This condition can be mathematically represented with the help of the given equation:

$$bf=(h_l-h_r), s.t. bf\in[-1,0,1]$$

Or

$$|bf|=|h_l-h_r|\leq 1$$

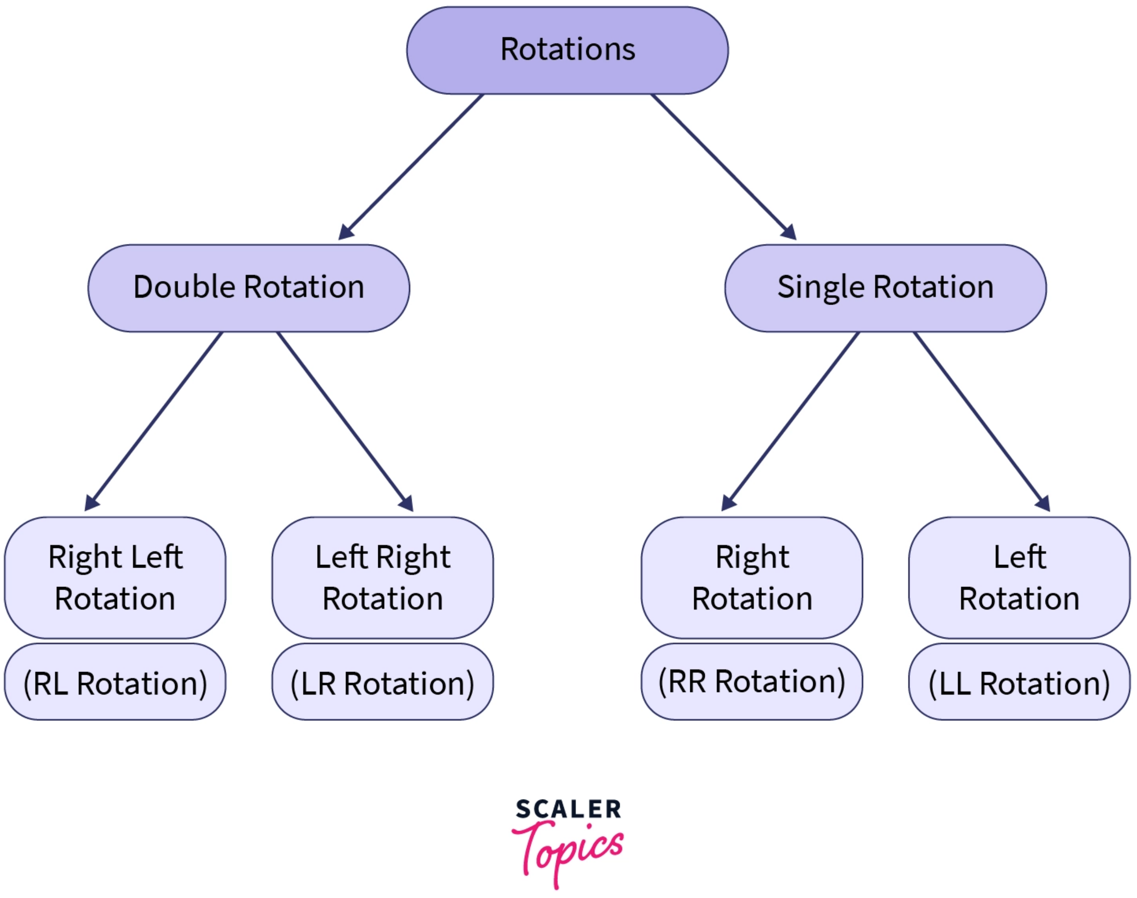
Here, the above equation indicates that the balance factor of any given node can only take the value of -1, 0, and 1 for a height-balanced Binary Search Tree. To maintain this criterion for all the nodes, AVL Trees take use of certain **Tree Rotations** that are discussed later in this article.

Highlights:

1. Balance Factor represents the height difference between the left and the right sub-trees for a given node.
2. For leaf nodes, the balance factor is 0.
3. AVL balance criteria: $|bf| \leq 1$ for all nodes.
4. Balance factor indicates whether a node is left heavy, right heavy, or balanced.

AVL Tree Rotation

As discussed earlier, the AVL Trees make use of the balance factor to check whether a given node is left-heavy (height of left sub-tree is one greater than that of right sub-tree), balanced, or right-heavy (height of right sub-tree is one greater than that of left sub-tree). If any node is unbalanced, it performs certain **Tree Rotations** to re-balance the tree.


Rotations
Double Rotation
Single Rotation
Right Left Rotation
(RL Rotation)
Left Right Rotation
(LR Rotation)
Right Rotation
(RR Rotation)
Left Rotation
(LL Rotation)
SCALER Topics

Tree Rotations: It is the process of changing the structure of the tree by moving smaller subtrees down and larger subtrees up, without interfering with the order of the elements.

If the balance factor of any node doesn't follow the AVL Balancing criterion, the AVL Trees make use of 4 different types of Tree rotations to re-balance themselves.

These rotations are classified based on the node imbalance cured by them i.e., a specific rotation is applied to counter the change that occurred in the balance factor of a node making it unbalanced.

These rotations include:

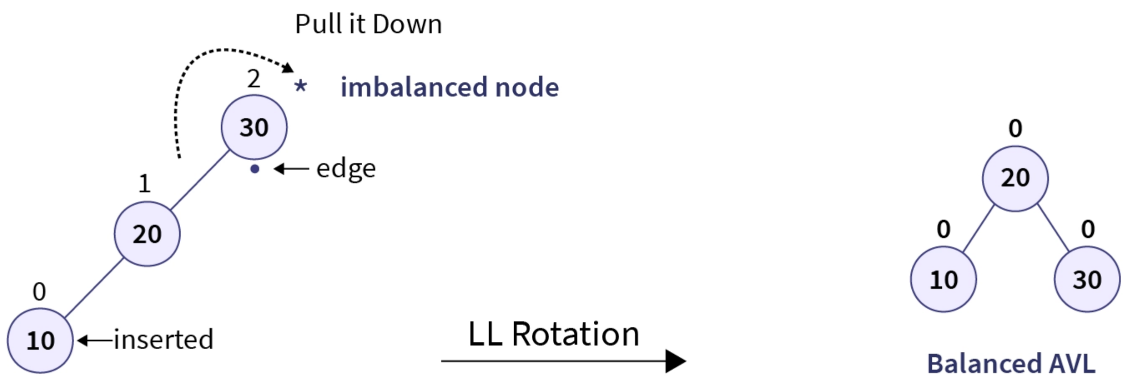
Now, let's look at all the tree rotations and understand how they can be used to balance the tree and make it follow the AVL balance criterion.

1. LL Rotation

It is a type of single rotation that is performed when the tree gets unbalanced, upon insertion of a node into the left subtree of the left child of the imbalance node i.e., upon **Left-Left (LL) insertion**. This imbalance indicates that the tree is heavy on the left side. Hence, a **right rotation (or clockwise rotation)** is applied such that this left heaviness imbalance is countered and the tree becomes a balanced tree. Let’s understand this process using an example:

Consider, a case when we wish to create a BST using elements 30, 20, and 10. Now, since these elements are given in a sorted order, the BST so formed is a left-skewed tree as shown below:

- **LL Rotation**



SCALER Topics

This is confirmed after calculating the balance factor of all the nodes present in the tree. As you can observe, when we insert element 10 in the tree, the root node becomes imbalanced (balance factor = 2) because the tree becomes left-skewed upon this operation. Also, notice that element 10 is inserted as a left child in the left subtree of the imbalanced node (here, the root node of the tree).

Hence, this is the case of **L-L insertion** and we will have to perform a certain operation to counter-act this left skewness.

Now, imagine a weighing scale in which we only have 5 kg on the left plate and nothing on the right plate. This is the case of left heavy since there is nothing on the right plate to balance the weight present in the left plate. Now, to balance this scale we can just add some weight to the right plate.

Hence, to balance the weight on one side we try to increase the weight on the other side. In the case of trees, instead of adding a new node (weight) on the lighter side, we try to rotate the structure of the tree around a pivot point thereby shifting the nodes from the heavier side to the lighter side.

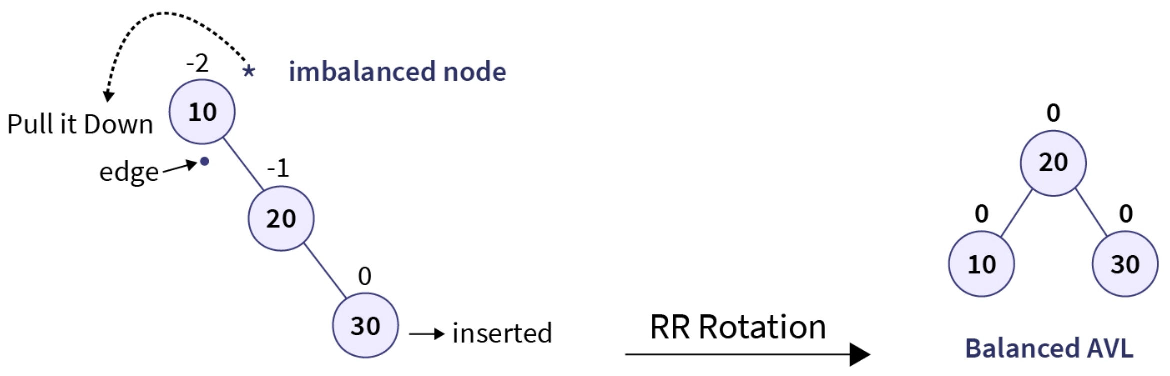
In our example, we have extra weight on the left subtree (LL insertion) therefore we will perform right rotation or clockwise rotation on the imbalanced node to transfer this node on the right side to retrieve a balanced tree i.e., we will pull the imbalanced node down by rotating the tree in a clockwise direction along the edge of the imbalanced or in this case, the root node.

2. RR Rotation

It is similar to that of LL Rotation but in this case, the tree gets unbalanced, upon insertion of a node into the right subtree of the right child of the imbalance node i.e., upon **Right-Right (RR)** insertion instead of the LL insertion. In this case, the tree becomes right heavy and a **left rotation (or anti-clockwise rotation)** is performed along the edge of the imbalanced node to counter this right skewness caused by the insertion operation. Let’s understand this process with an example:

Consider a case where we wish to create a BST using the elements 10, 20, and 30. Now, since the elements are given in sorted order, the BST so created becomes right-skewed as shown below:

- **RR Rotation**



SCALER Topics

Upon calculating the balance factor of all the nodes, we can confirm that the root node of the tree is imbalanced (balance factor = 2) when the element 30 is inserted using RR-insertion.

Hence, the tree is heavier on the right side and we can balance it by transferring the imbalanced node on the left side by applying an anti-clockwise rotation around the edge (pivot point) of the imbalanced node or in this case, the root node.

3. LR Rotation

So far we have discussed that when the tree is heavy on one side, just perform a single rotation in the opposite direction to counter the effect of tree skewness. But, there also exist some cases where a single tree rotation isn't enough to balance the tree i.e., we may need to perform one more rotation to finally counter the effects of the height-affecting operation.

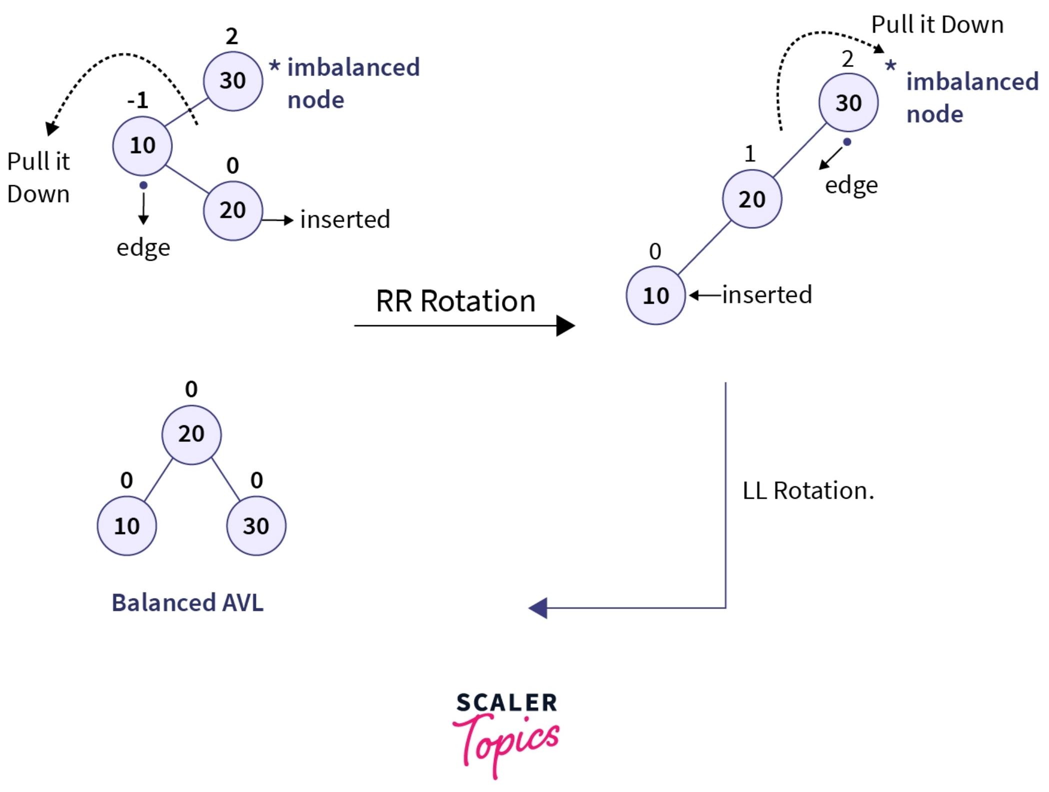
One such case is of **Left-Right (LR) insertion** i.e., the tree gets unbalanced, upon insertion of a node into the right subtree of the left child of the imbalance node. Let's understand this case using an example:

Consider a situation where you create a BST using elements 30, 10, and 20. Now, when the elements are inserted, the element 30 becomes the root, 10 becomes its left child, and when the element 20 is inserted, it is inserted as the right child of the node having the value 10. This causes an imbalance in the tree as the root node's balance factor becomes equal to 2.

Now, as per the previous discussions, you may have noticed that a positive balance factor indicates that the given node is **left-heavy** while a negative one indicates that the node is **right-heavy**.

Now, if we notice the immediate parent of the inserted node, we notice that its balance factor is negative i.e., its right-heavy. Hence, you may say that we should perform a left rotation (RR rotation) on the immediate parent of the inserted node to counter this effect. Let's perform this rotation and notice the change:

- **LR Rotation**


2
30
* imbalanced node
-1
10
Pull it Down
0
20
inserted
edge
RR Rotation
Pull it Down
2
30
* imbalanced node
1
20
edge
0
10
inserted
LL Rotation.
0
20
0
10
0
30
Balanced AVL
SCALER Topics

As you can observe, upon applying the RR rotation the BST becomes left-skewed and is still unbalanced. This is now the case of LL rotation and by rotating the tree along the edge of the imbalanced node in the clockwise direction, we can retrieve a balanced BST.

Hence, a simple rotation won't fully balance the tree but it may flip the tree in such a manner that it gets converted into a single rotation scenario, after which we can balance the tree by performing one more tree rotation.

This process of applying two rotations sequentially one after another is known as **double rotation** and since in our example the insertion was **Left-Right (LR) insertion**, this combination of RR and LL rotation is known as LR rotation. Hence, to summarize:

The LR rotation consists of 2 steps:

1. Apply RR Rotation (anti-clockwise rotation) on the left subtree of the imbalanced node as the left child of the imbalanced node is right-heavy. This process flips the tree and converts it into a left-skewed tree.
2. Perform LL Rotation (clock-wise rotation) on the imbalanced node to balance the left-skewed tree.

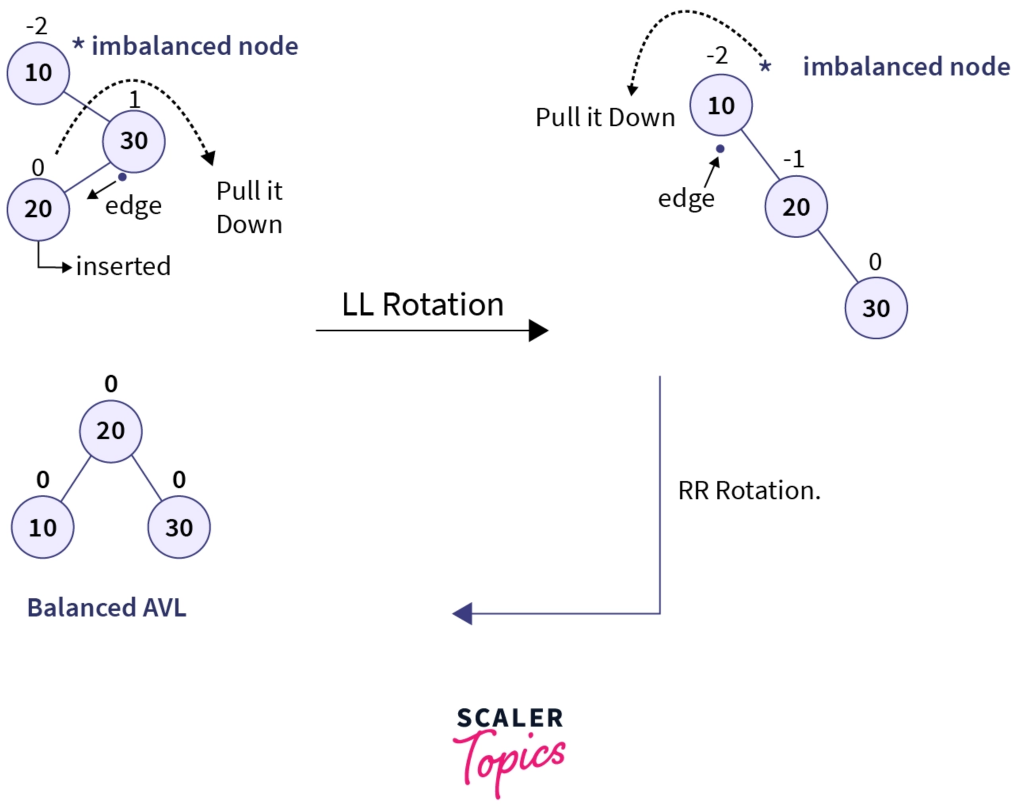
Hence, LR rotation is essentially a combination of RR and LL Rotation.

4. RL Rotation

It is similar to LR rotation but it is performed when the tree gets unbalanced, upon insertion of a node into the left subtree of the right child of the imbalance node i.e., upon **Right-Left (RL) insertion** instead of LR insertion. In this case, the immediate parent of the inserted node becomes left-heavy i.e., the LL rotation (right rotation or clockwise rotation) is performed that converts the tree into a right-skewed tree. After which, RR rotation (left rotation or anti-clockwise rotation) is applied around the edge of the imbalanced node to convert this right-skewed tree into a balanced BST.

Let's now observe an example of the RL rotation:

- **RL Rotation**


-2
10
* imbalanced node
1
30
0
20
edge
Pull it Down
inserted
LL Rotation
-2
10
imbalanced node
Pull it Down
edge
-1
20
0
30
RR Rotation.
0
20
0
10
0
30
Balanced AVL
SCALER Topics

In the above example, we can observe that the root node of the tree becomes imbalanced upon insertion of the node having the value 20. Since this is a type of RL insertion, we will perform LL rotation on the immediate parent of the inserted node thereby retrieving a right-skewed tree. Finally, we will perform RR Rotation around the edge of the imbalanced node (in this case the root node) to get the balanced AVL tree.

Hence, RL rotation consists of two steps:

1. Apply LL Rotation (clockwise rotation) on the right subtree of the imbalanced node as the right child of the imbalanced node is left-heavy. This process flips the tree and converts it into a right-skewed tree.
2. Perform RR Rotation (anti-clockwise rotation) on the imbalanced node to balance the right-skewed tree.

NOTE:

- Rotations are done only on three nodes (including the imbalanced node) irrespective of the size of the Binary Search Tree. Hence, in the case of a large tree always focus on the two nodes around the imbalanced node and perform the tree rotations.
- Upon insertion of a new node, if multiple nodes get imbalanced then traverse the ancestors of the inserted node in the tree and perform rotations on the first occurred imbalanced node. Continue this process until the whole tree is balanced. This process is knowns as **retracing** which is discussed later in the article.

Highlights:

1. Rotations are performed to maintain the AVL Balance criteria.
2. Rotation is a process of changing the structure without affecting the elements' order.
3. Rotations are done on an unbalanced node based on the location of the newly inserted node.
4. Single rotations include LL (clockwise) and RR (anti-clockwise) rotations.
5. Double rotations include LR (RR + LL) and RL (LL + RR) rotations.
6. Rotations are done only on 3 nodes, including the unbalanced node.

Operations on AVL Trees

Since AVL Trees are self-balancing Binary Search Trees, all the operations carried out using AVL Trees are similar to that of Binary Search Trees. Also, since searching an element and traversing the tree doesn't lead to any change in the structure of the tree, these operations can't violate the height balancing property of AVL Trees. Hence, searching and traversing operations are the same as that of Binary Search Trees.

However, upon the execution of each insertion or deletion operation, we perform a check on the balance factor of all the nodes and perform rotations to balance the AVL Tree if needed. Let's look at these operations in detail:

1. Insertion

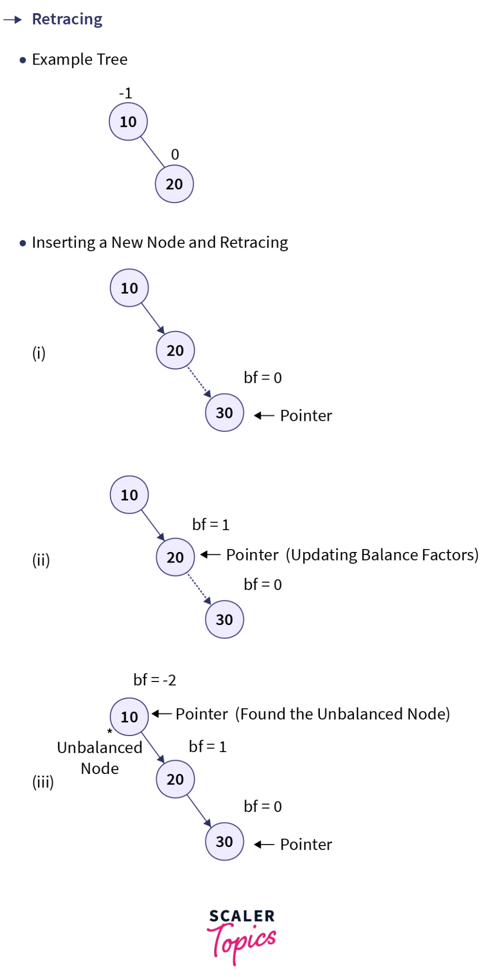
In Binary Search Trees, the new node (let say N) was inserted in the tree by traversing the tree using BST logic to locate a node having NULL as its child that can be replaced to insert the new node N. Hence, in BSTs a new node is always inserted as a leaf node by replacing the NULL value of a node’s child.

Just like the insertion in BSTs, the new node is always inserted as a leaf node in AVL Trees i.e., the balance factor of the newly inserted node is always equal to 0. However, after each insertion in the tree, the balance factor for the ancestors of the newly inserted node is checked to verify that the tree is balanced or not.

Here, only the ancestors of the inserted node are checked for imbalance because when a new node is inserted, it only alters the height of its ancestors thereby inducing an imbalance in the tree. This process of finding the unbalanced node by traversing the ancestors of the newly inserted node is known as **retracing**.

If the tree becomes unbalanced after inserting a new node, retracing helps us in finding the location of a node in the tree at which we need to perform the tree rotations to balance the tree.

The below gif demonstrates the retracing process upon inserting a new element in the AVL Tree:

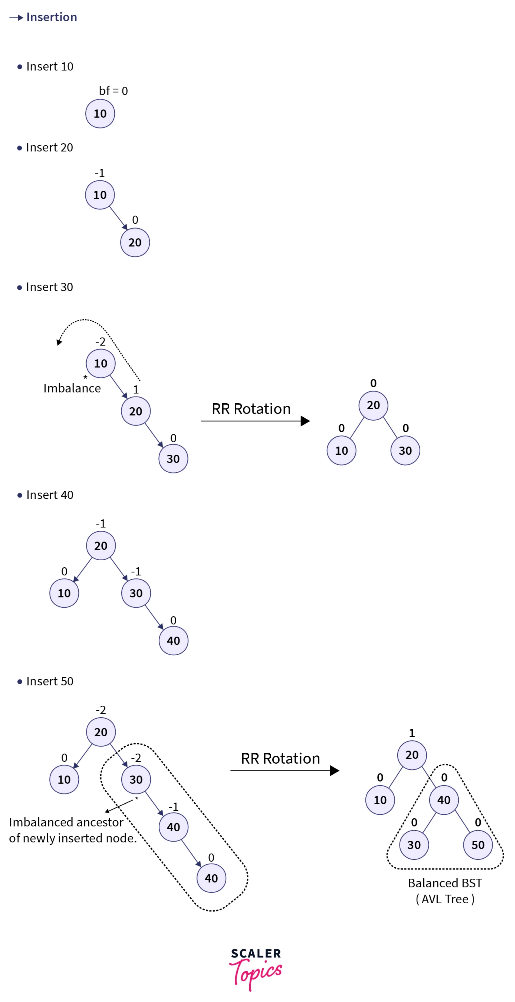

→ Retracing
• Example Tree
-1
10
0
20
• Inserting a New Node and Retracing
(i)
10
20
bf = 0
30
← Pointer
(ii)
10
bf = 1
20
← Pointer (Updating Balance Factors)
bf = 0
30
(iii)
bf = -2
10
← Pointer (Found the Unbalanced Node)
Unbalanced Node
bf = 1
20
bf = 0
30
← Pointer
SCALER Topics

Let’s look at the algorithm of the insertion operation in AVL Trees:

Insertion in AVL Trees:

1. START
2. Insert the node using BST insertion logic.
3. Calculate and check the balance factor of each node.
4. If the balance factor follows the AVL criterion, go to step 6
5. Else, perform tree rotations according to the insertion done. Once the tree is balanced go to step 6.
6. END

For better understanding let’s consider an example where we wish to create an AVL Tree by inserting the elements: 10, 20, 30, 40, and 50. The below gif demonstrates how the given elements are inserted one by one in the AVL Tree:


Insertion
Insert 10
bf = 0
10
Insert 20
-1
10
0
20
Insert 30
-2
10
Imbalance
1
20
0
30
RR Rotation
0
20
0
10
0
30
Insert 40
-1
20
0
10
-1
30
0
40
Insert 50
-2
20
0
10
-2
30
Imbalanced ancestor
of newly inserted node.
-1
40
0
40
RR Rotation
1
20
0
10
0
40
0
30
0
50
Balanced BST
(AVL Tree)
SCALER
Topics

2. Deletion

When an element is to be deleted from a Binary Search Tree, the tree is searched using various comparisons via the BST rule till the currently traversed node has the same value as that of the specified element. If the element is found in the tree, there are three different cases in which the deletion operation occurs depending upon whether the node to be deleted has any children or not:

Case 1: When the node to be deleted is a leaf node

- In this case, the node to be deleted contains no subtrees i.e., it's a leaf node. Hence, it can be directly removed from the tree.

Case 2: When the node to be deleted has one subtree

- In this case, the node to be deleted is replaced by its only child thereby removing the specified node from the BST.

Case 3: When the node to be deleted has both the subtrees.

- In this case, the node to be deleted can be replaced by one of the two available nodes:
    - It can be replaced by the node having the largest value in the left subtree **(Longest left node or Predecessor)**.
    - Or, it can be replaced by the node having the smallest value in the right subtree **(Smallest right node or Successor)**.

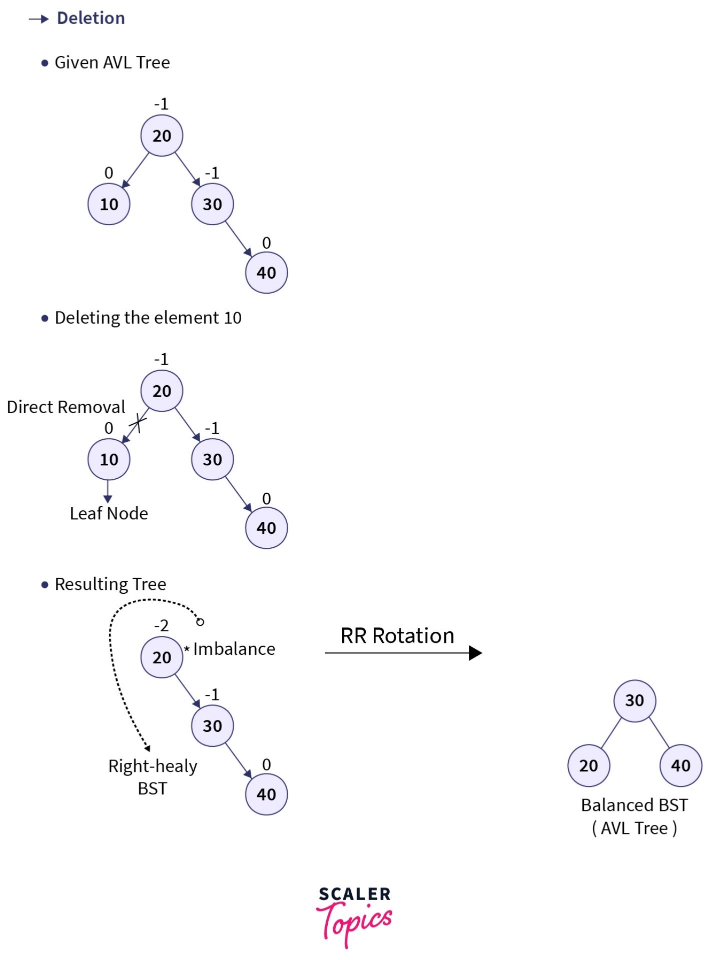
Just like the deletion operation in Binary Search Trees, the elements are deleted from AVL Trees depending upon whether the node has any children or not. However, upon every deletion in AVL Trees, the balance factor is checked to verify that the tree is balanced or not. If the tree becomes unbalanced after deletion, certain rotations are performed to balance the Tree.

Let's look at the algorithm of the deletion operation in AVL Trees:

Deletion in AVL Trees

1. START
2. Find the node in the tree. If the element is not found, go to step 7.
3. Delete the node using BST deletion logic.
4. Calculate and check the balance factor of each node.
5. If the balance factor follows the AVL criterion, go to step 7.
6. Else, perform tree rotations to balance the unbalanced nodes. Once the tree is balanced go to step 7.
7. END

For better understanding let’s consider an example where we wish to delete the element having value 10 from the AVL Tree created using the elements 10, 20, 30, and 40. The below gif demonstrates how we can delete an element from an AVL Tree:


→ Deletion
• Given AVL Tree
-1
20
0
10
-1
30
0
40
• Deleting the element 10
-1
20
Direct Removal
0
10
-1
30
Leaf Node
0
40
• Resulting Tree
-2
20
Imbalance
-1
30
Right-healy
BST
0
40
RR Rotation
30
20
40
Balanced BST
(AVL Tree)
SCALER
Topics

Highlights:

1. All AVL Tree operations are similar to that of BST except insertion and deletion.
2. Upon every insertion and deletion, we need to check whether the tree is balanced or not.
3. If required, the rotations are performed after each operation to balance the tree.