In this tutorial, you will learn what an avl tree is. Also, you will find working examples of various operations performed on an avl tree in C, C++, Java and Python.

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

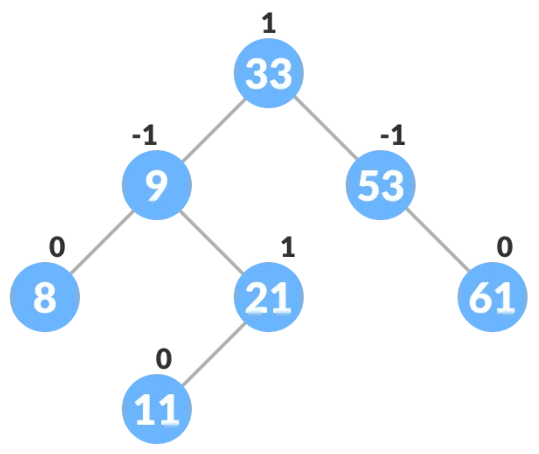
Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

An example of a balanced avl tree is:



Avl tree

Rotating the subtrees in an AVL Tree

In rotation operation, the positions of the nodes of a subtree are interchanged.

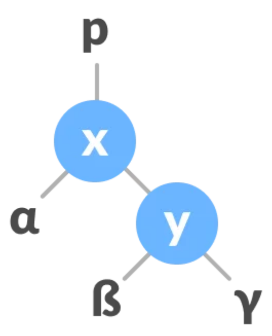
There are two types of rotations:

Left Rotate

In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

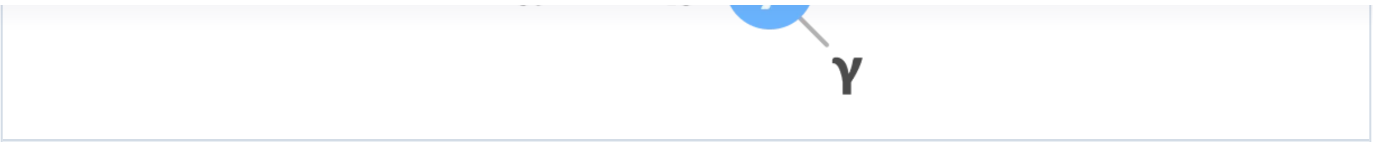
Algorithm

1. Let the initial tree be:



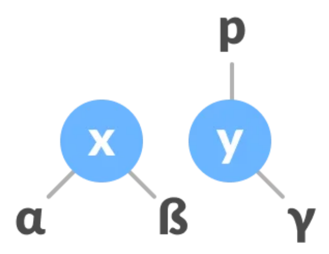
Left rotate

2. If `y` has a left subtree, assign `x` as the parent of the left subtree of `y`.



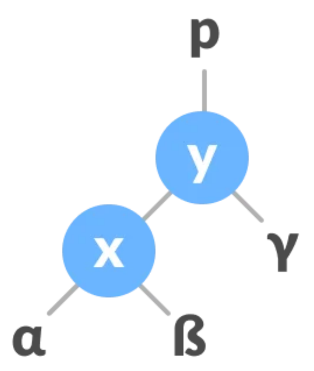
Assign x as the parent of the left subtree of y

3. If the parent of `x` is `NULL`, make `y` as the root of the tree.
4. Else if `x` is the left child of `p`, make `y` as the left child of `p`.
5. Else assign `y` as the right child of `p`.

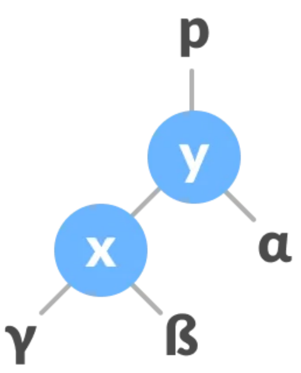


Change the parent of x to that of y

6. Make `y` as the parent of `x`.



Assign y as the parent of x.



Initial tree

2. If `x` has a right subtree, assign y as the parent of the right subtree of `x`.

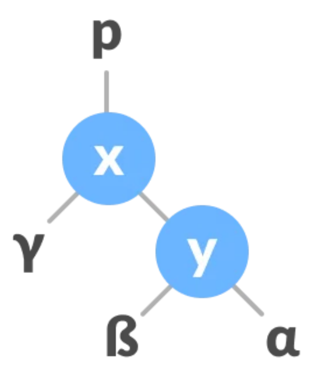
right-rotate

Assign y as the parent of the right subtree of x

3. If the parent of `y` is `NULL`, make `x` as the root of the tree.
4. Else if `y` is the right child of its parent `p`, make `x` as the right child of `p`.
5. Else assign `x` as the left child of `p`.

Assign the parent of y as the parent of x.

6. Make `x` as the parent of `y`.

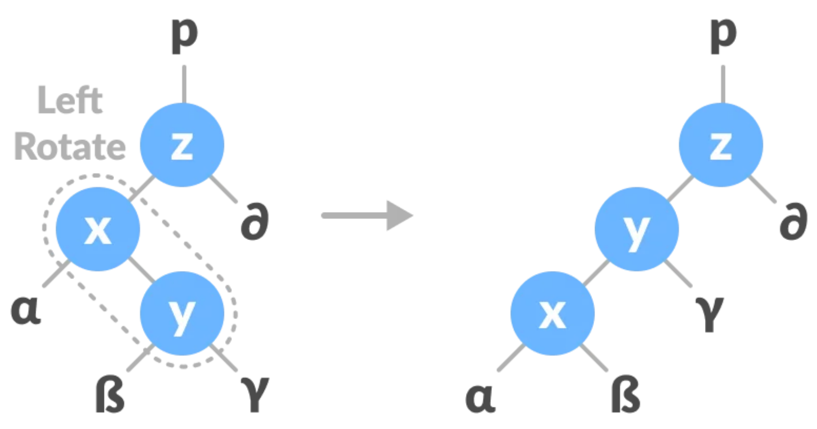


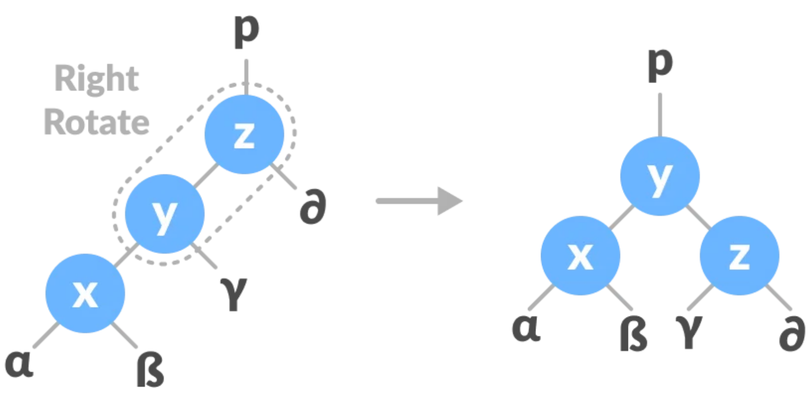
Assign x as the parent of y

Left-Right and Right-Left Rotate

In left-right rotation, the arrangements are first shifted to the left and then to the right.

1. Do left rotation on x-y.

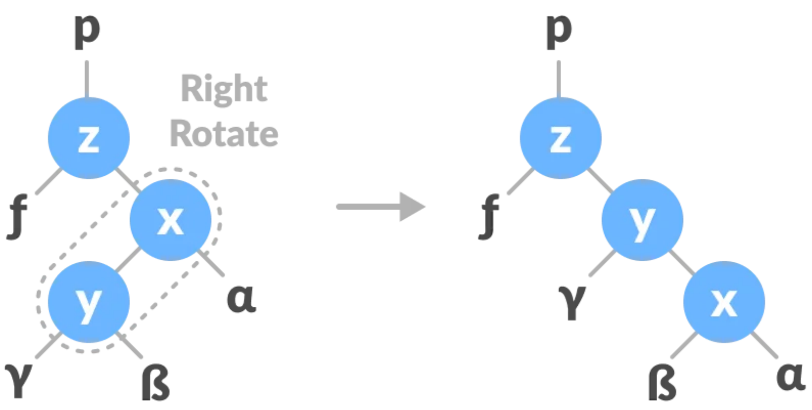




Right rotate z-y

In right-left rotation, the arrangements are first shifted to the right and then to the left.

1. Do right rotation on x-y.



Right rotate x-y

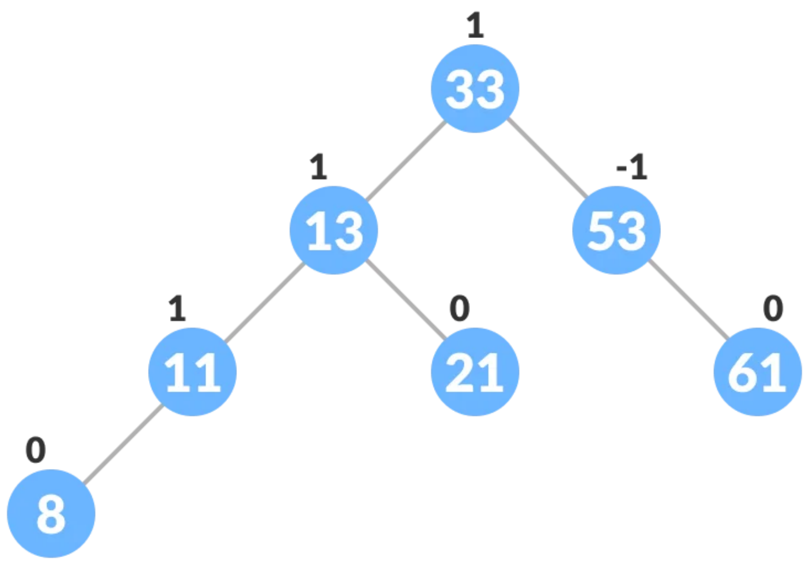
2. Do left rotation on z-y.

Left rotate z-y

Algorithm to insert a newNode

A `newNode` is always inserted as a leaf node with balance factor equal to 0.

1. Let the initial tree be:

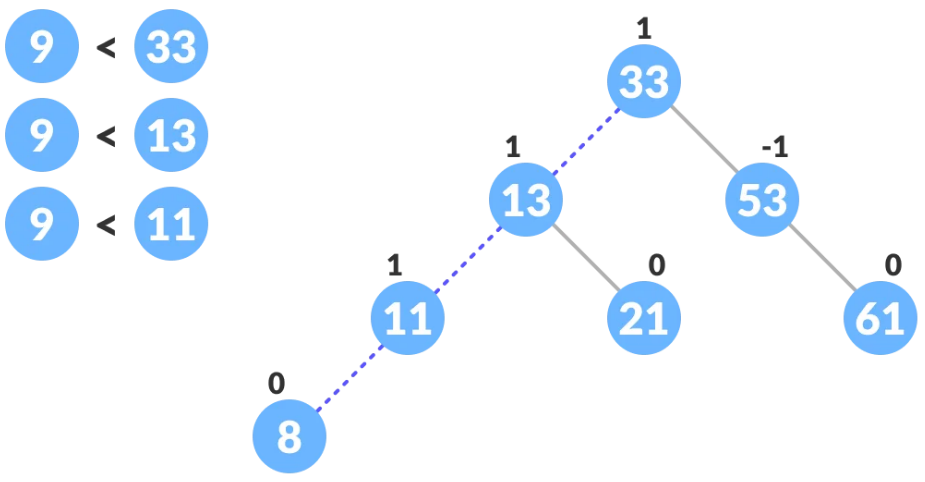


Initial tree for insertion

Let the node to be inserted be:

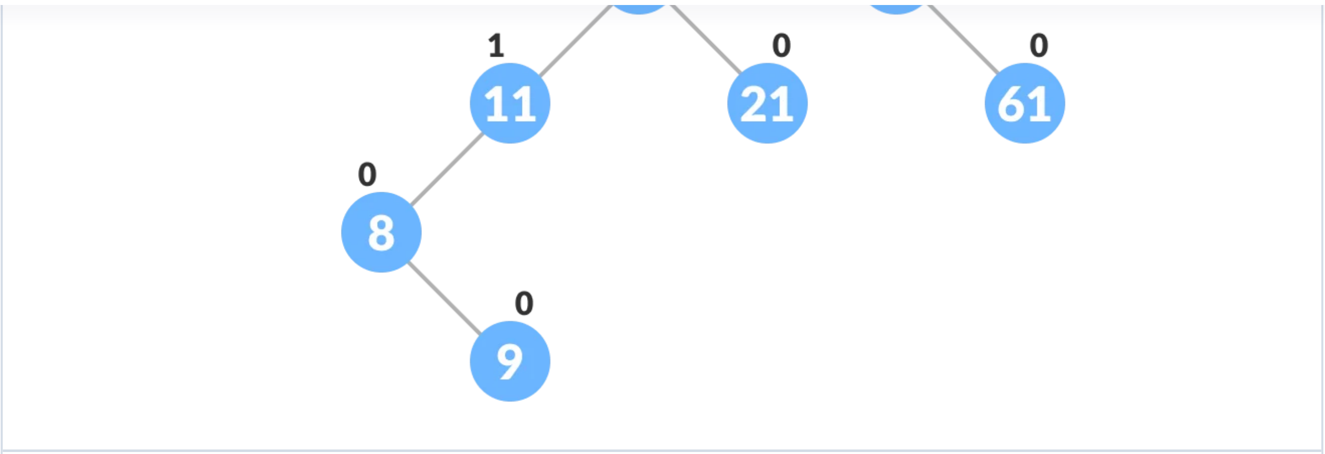
New node

2. Go to the appropriate leaf node to insert a `newNode` using the following recursive steps. Compare `newKey` with `rootKey` of the current tree.
   a. If `newKey` < `rootKey`, call insertion algorithm on the left subtree of the current node until the leaf node is reached.
   b. Else if `newKey` > `rootKey`, call insertion algorithm on the right subtree of current node until the leaf node is reached.
   c. Else, return `leafNode`.



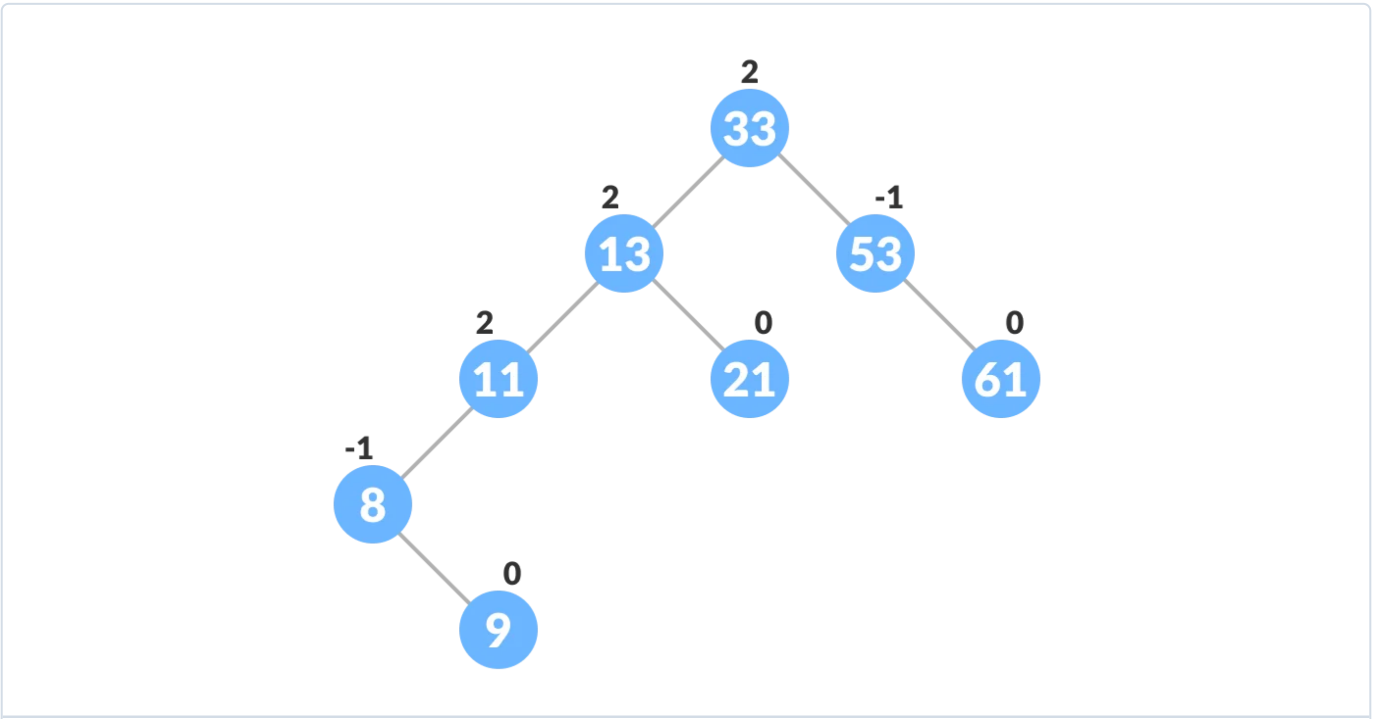
Finding the location to insert newNode

3. Compare `leafKey` obtained from the above steps with `newKey`:
   a. If `newKey` < `leafKey`, make newNode as the `leftChild` of `leafNode`.
   b. Else, make `newNode` as rightChild of `leafNode`.



Inserting the new node

4. Update `balanceFactor` of the nodes.

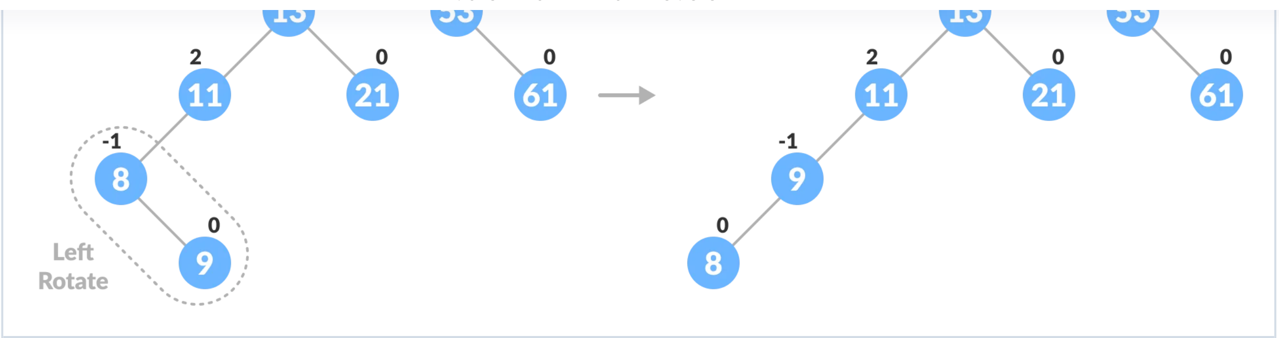


Updating the balance factor after insertion

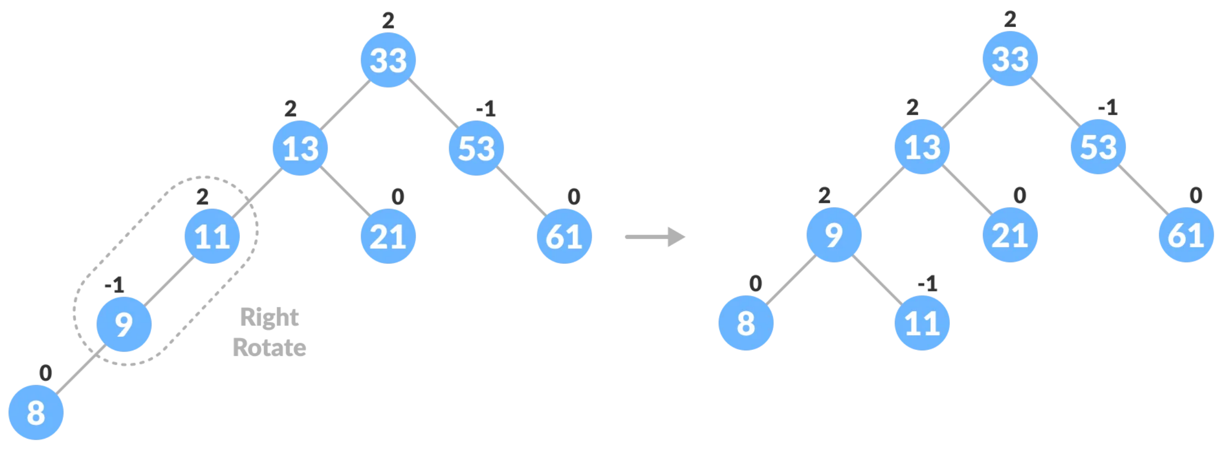
5. If the nodes are unbalanced, then rebalance the node.

   a. 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

      a. If `newNodeKey` < `leftChildKey` do right rotation.



Balancing the tree with rotation



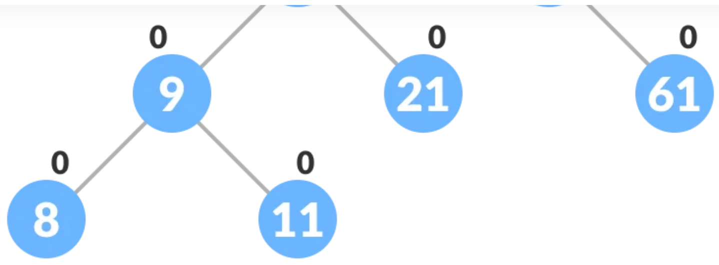
Balancing the tree with rotation

b. If `balanceFactor` < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation

  a. If `newNodeKey` > `rightChildKey` do left rotation.

  b. Else, do right-left rotation

6. The final tree is:

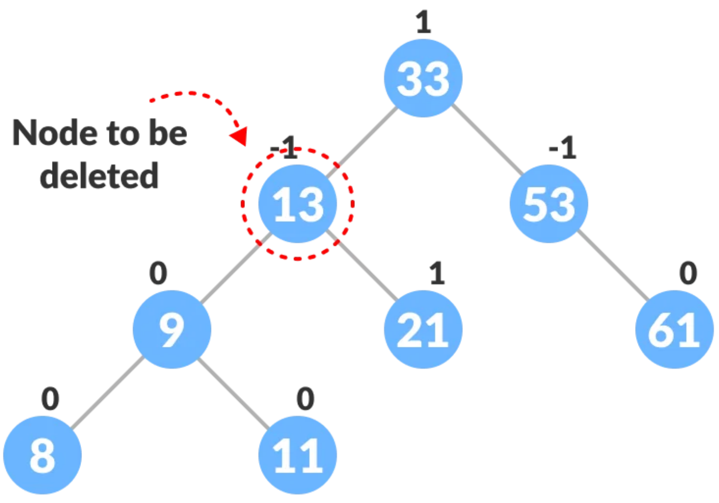


Final balanced tree

Algorithm to Delete a node

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

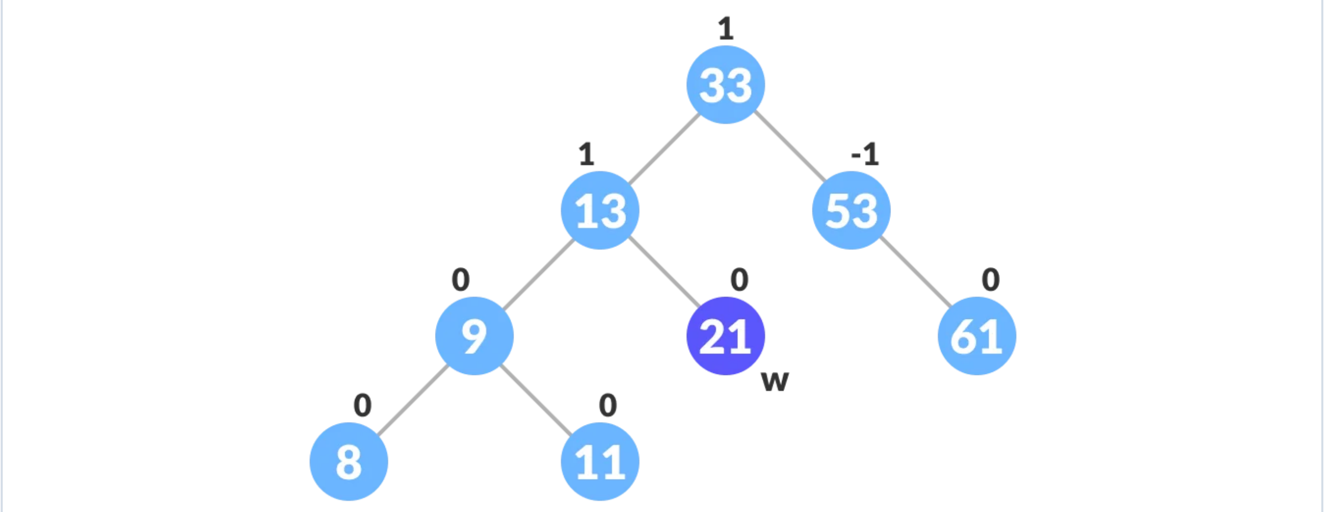
1. Locate `nodeToBeDeleted` (recursion is used to find `nodeToBeDeleted` in the code used below).



Locating the node to be deleted

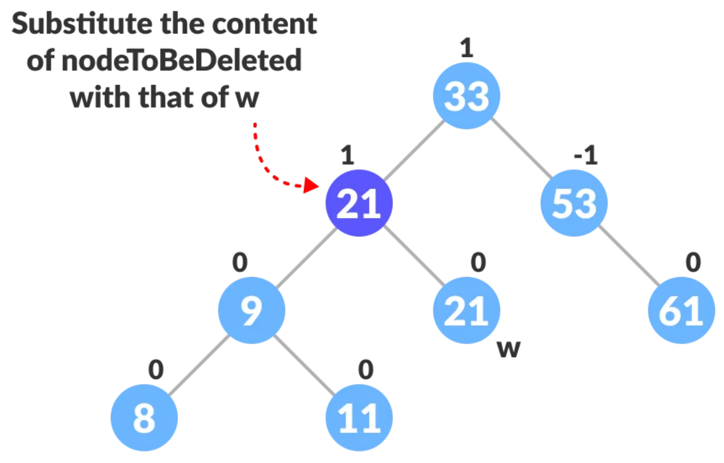
2. There are three cases for deleting a node:

c. If `nodeToBeDeleted` has two children, find the inorder successor `w` of `nodeToBeDeleted` (ie. node with a minimum value of key in the right subtree).



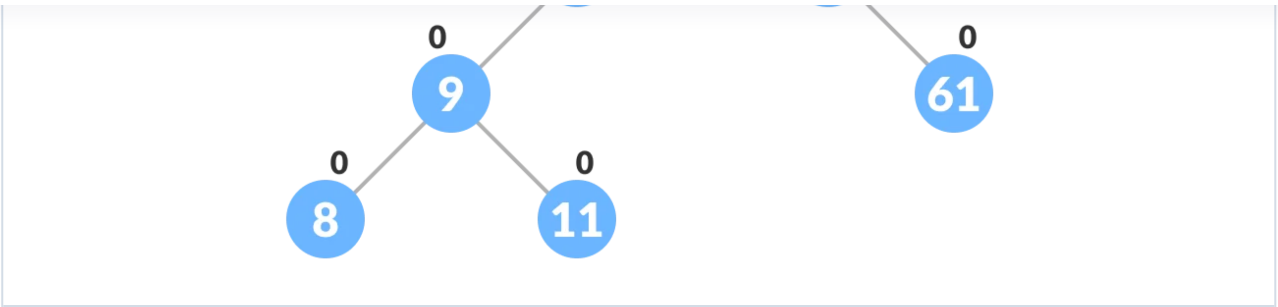
Finding the successor

a. Substitute the contents of `nodeToBeDeleted` with that of `w`.



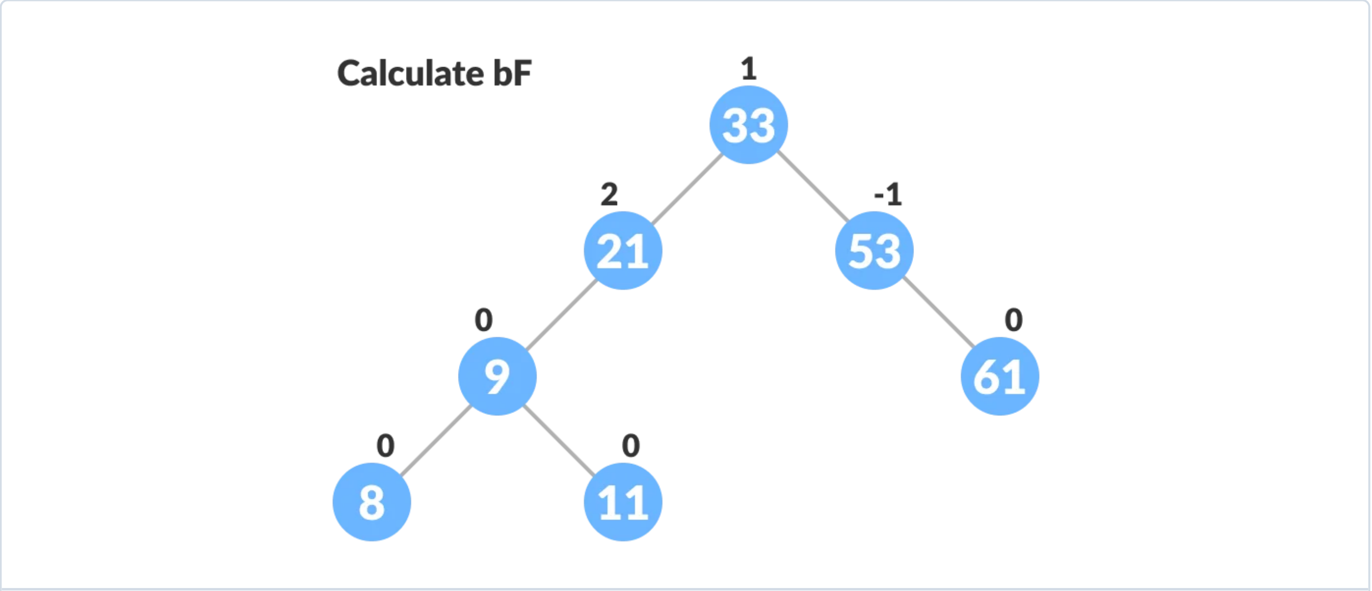
Substitute the node to be deleted

b. Remove the leaf node `w`.



Remove w

3. Update `balanceFactor` of the nodes.

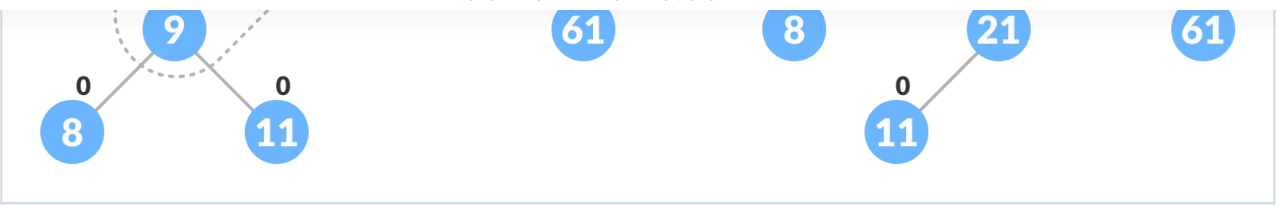


Update bf

4. Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.

   a. If `balanceFactor` of `currentNode` > 1,

      a. If `balanceFactor` of `leftChild` >= 0, do right rotation.



Right-rotate for balancing the tree

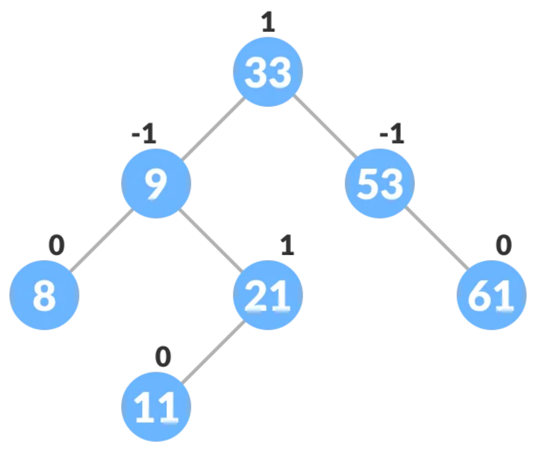
b. Else do left-right rotation.

b. If `balanceFactor` of `currentNode` < -1,

a. If `balanceFactor` of `rightChild` <= 0, do left rotation.

b. Else do right-left rotation.

5. The final tree is:



Avl tree final

Python, Java and C/C++ Examples

Python Java C C++

```
# Create a tree node
class TreeNode(object):
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1


class AVLTree(object):

    # Function to insert a node
    def insert_node(self, root, key):

        # Find the correct location and insert the node
        if not root:
            return TreeNode(key)
        elif key < root.key:
            root.left = self.insert_node(root.left, key)
        else:
            root.right = self.insert_node(root.right, key)

        root.height = 1 + max(self.getHeight(root.left),
```

Complexities of Different Operations on an AVL Tree

Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)

AVL Tree Applications

- For indexing large records in databases
- For searching in large databases

Binary Search Tree (/dsa/binary-search-tree)

Did you find this article helpful?

😃 | 🙁

Related Tutorials

DS & Algorithms

Perfect Binary Tree

(/dsa/perfect-binary-tree)

DS & Algorithms

Full Binary Tree

(/dsa/full-binary-tree)

DS & Algorithms

Balanced Binary Tree

(/dsa/balanced-binary-tree)

DS & Algorithms