AVL TREES

SELF BALANCING BINARY SEARCH TREES

RAHUL PATIL
MAHESH MOKALE
LAKSHMI SOWJANYA KALLEPALLI

AGENDA

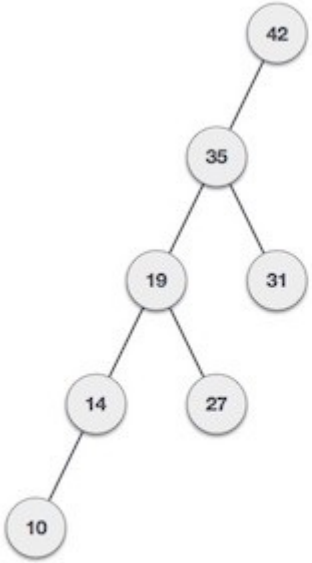
1. INTRODUCTION TO AVL TREES
2. HOW IS TREE BALANCED?
3. INSERTION OPERATION
4. DELETION OPERATION
5. ADVANTAGES OF USING AVL TREES
6. APPLICATIONS OF AVL TREES
7. SUMMARY

INTRODUCTION

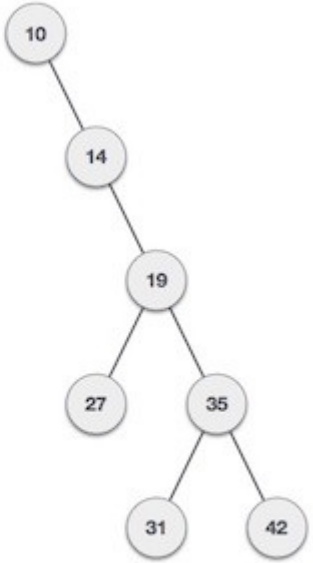
In certain cases BST can perform as bad as linear search algorithms ie. O(n). AVL trees were created just to avoid that. So, AVL trees are height balancing Binary Search Trees. AVL trees check the height of both the subtrees of each node and make sure that the difference in both heights is either 1, 0 or -1. The difference is called Balance Factor.

Balance Factor = height(left-subtree) - height(right-subtree)

Worst performance of BST

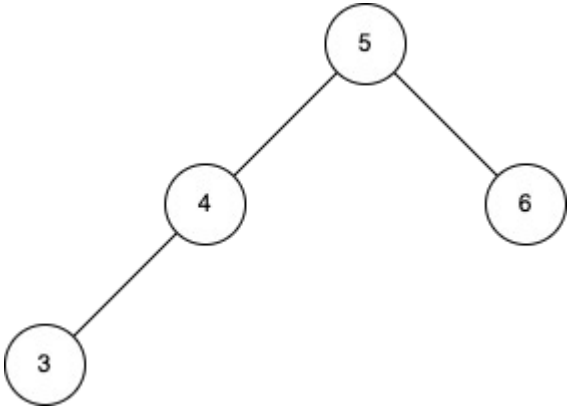


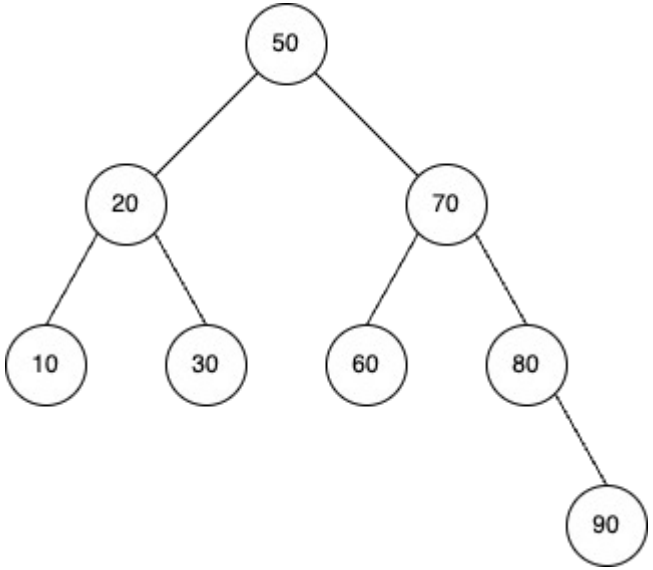
If input ‘appears’ non-increasing manner



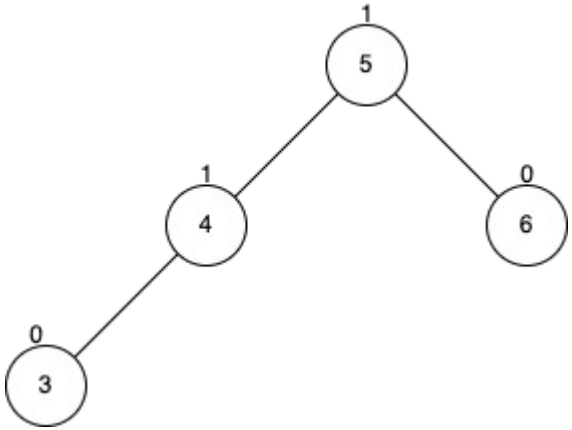
If input ‘appears’ in non-decreasing manner

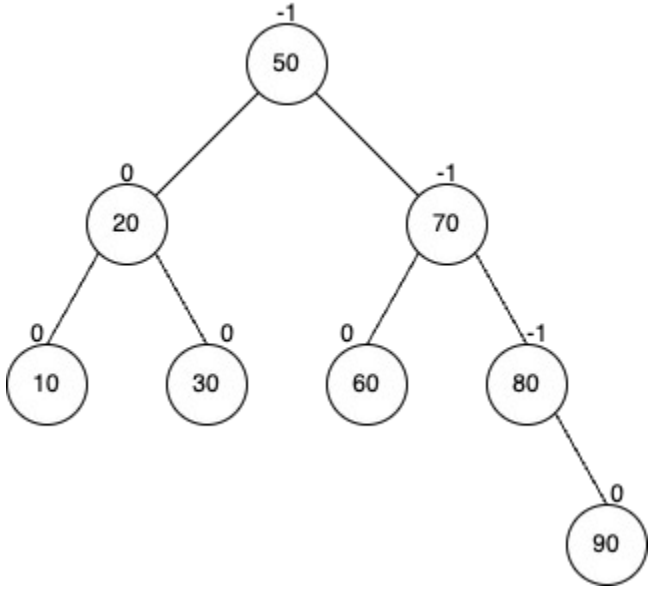
Can you calculate the balance factor?



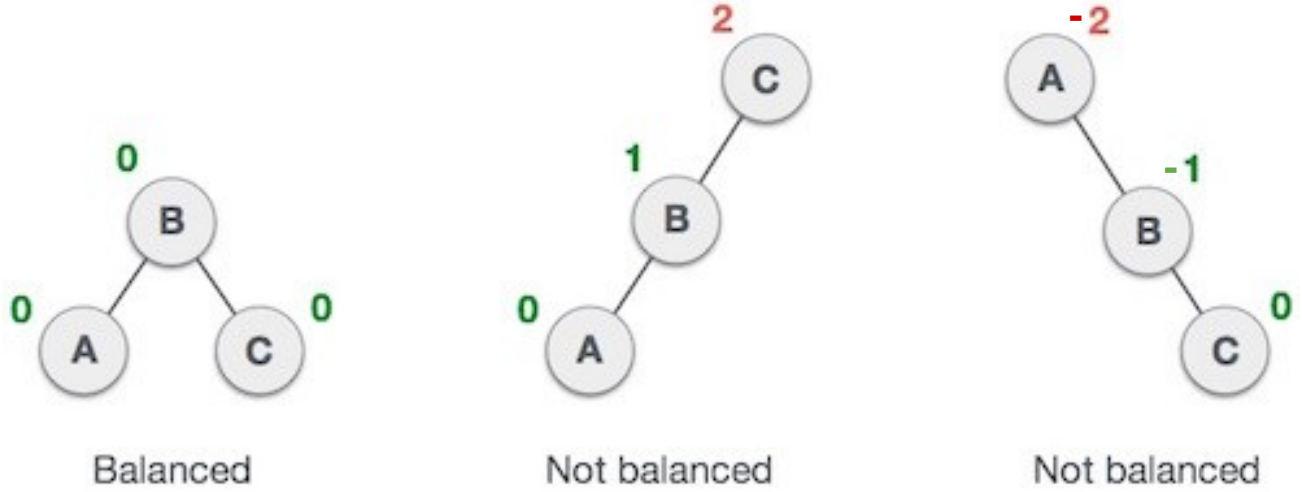


Solution



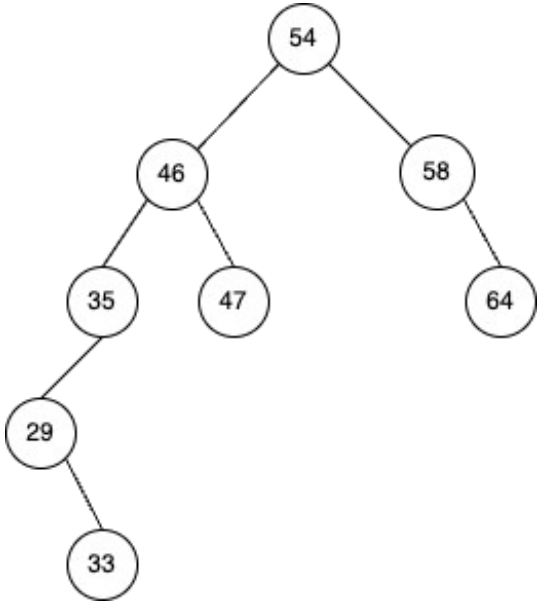


EXAMPLES OF BALANCED AND UNBALANCED TREES

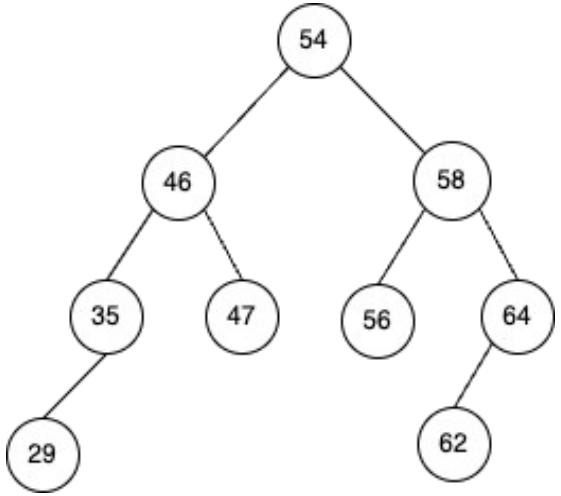


QUIZ

STATE WHETHER FOLLOWING TREES ARE BALANCED OR UNBALANCED.

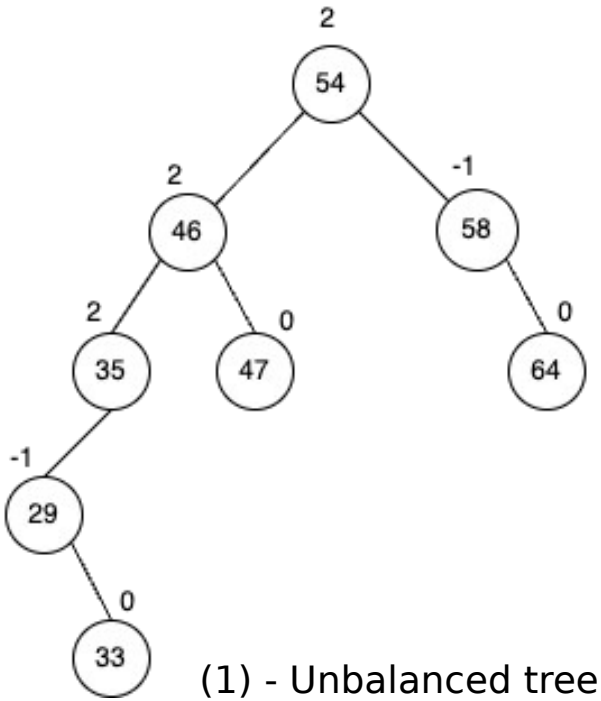


(1)

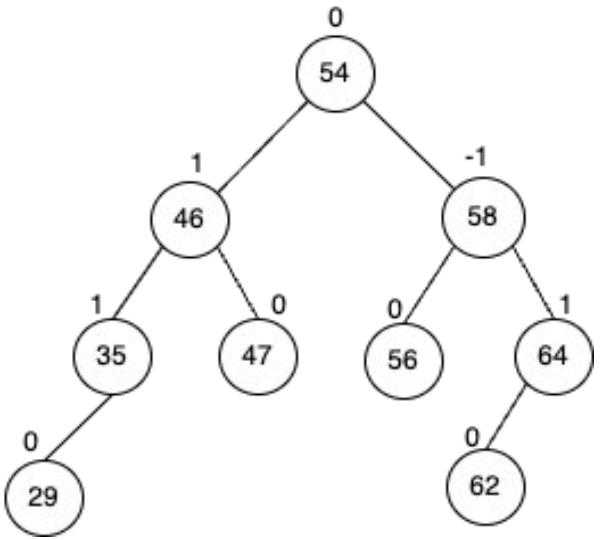


(2)

SOLUTION



(1) - Unbalanced tree



(2) - Balanced tree

HOW IS TREE BALANCED?

There are 4 types of rotations that are performed when the insertion takes place in AVL Trees.

1] Left Rotation

2] Right Rotation

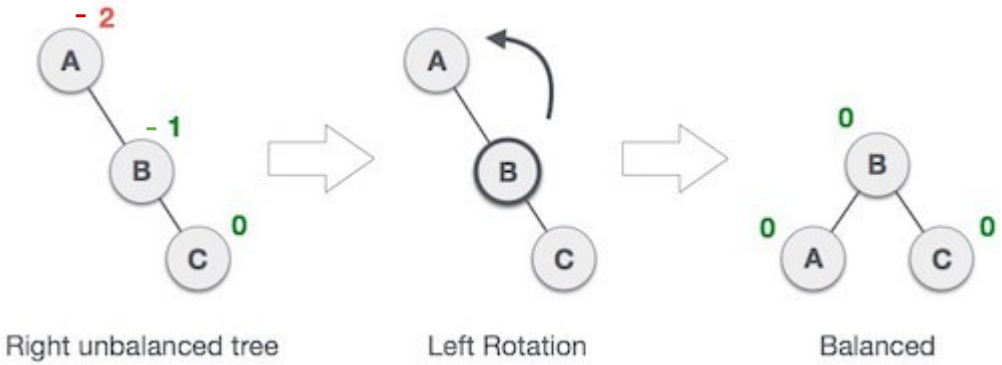
3] LR Rotation (Left Right Rotation)

4] RL Rotation (Right Left Rotation)

Left Rotation

When Right Rotation takes place while insertion?

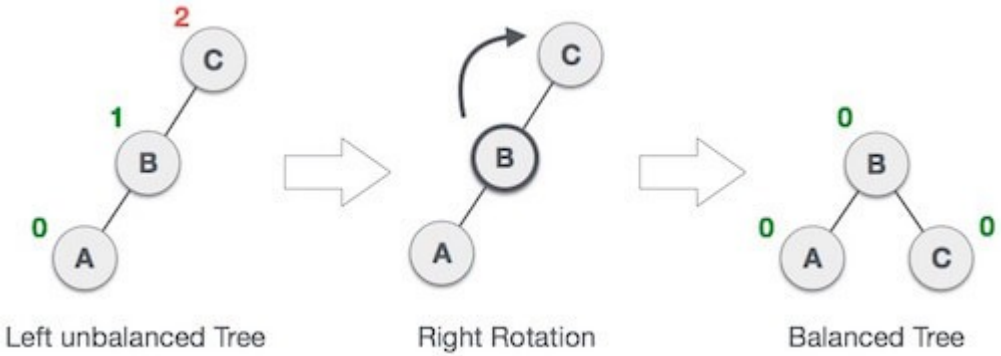
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform Left rotation, Left Rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



Right Rotation

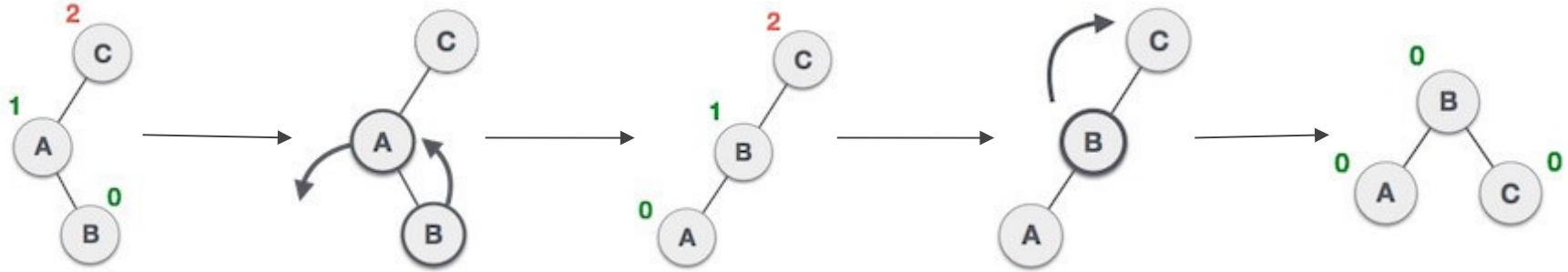
When Right Rotation takes place while insertion?

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform right rotation, Right Rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



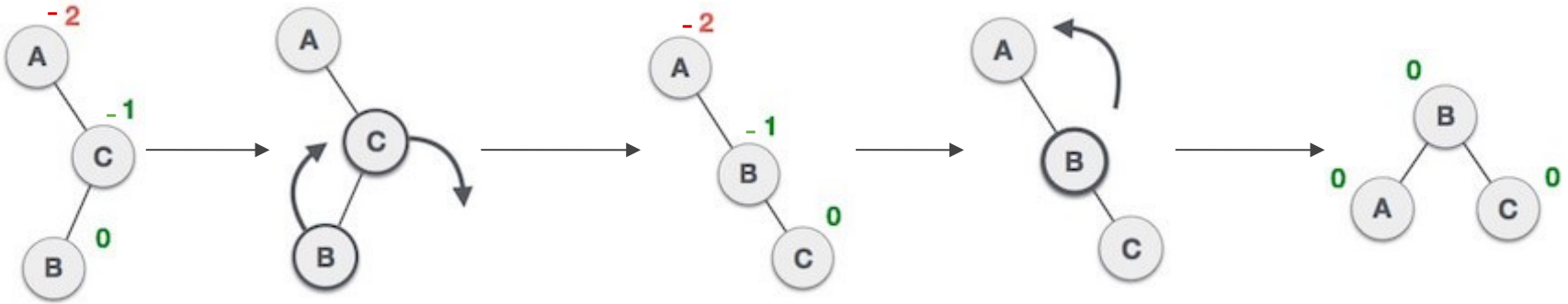
LR Rotation

A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C



RL Rotation

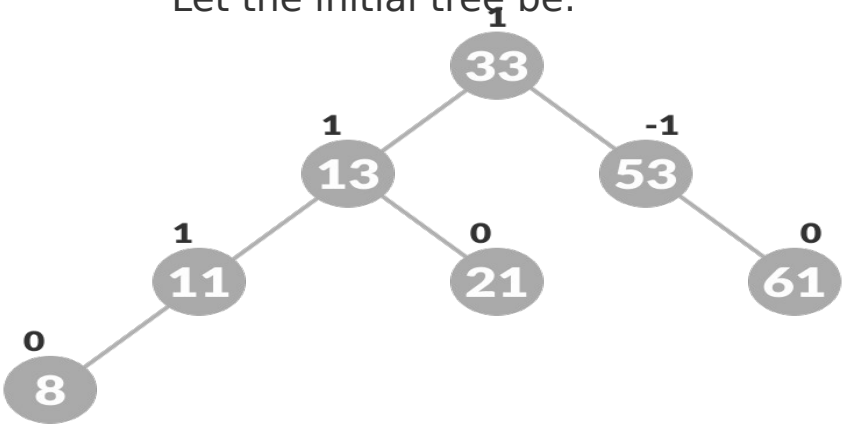
A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A



Algorithm to insert a new Node

A new Node is always inserted as a leaf node with balance factor equal to 0.

Let the initial tree be:



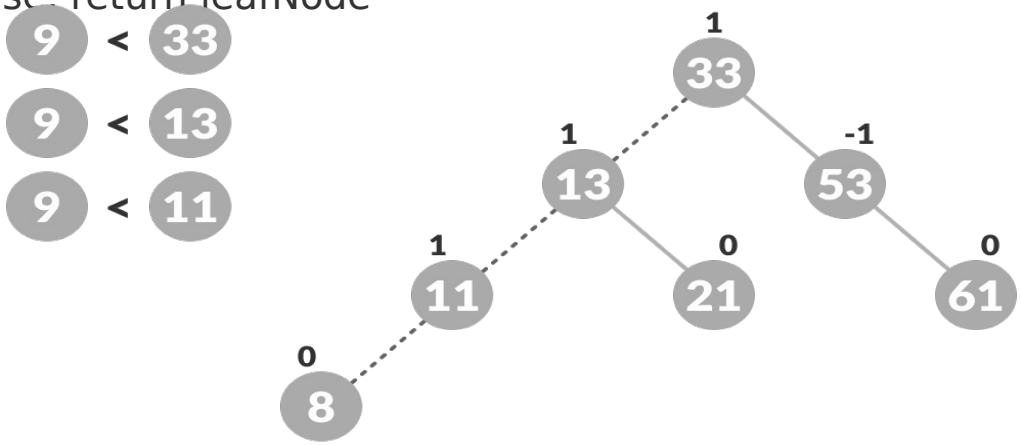
Initial tree for insertion

Let the node to be inserted be:

New node

Go to the appropriate leaf node to insert a **newNode** using the following recursive steps. Compare **newKey** with **rootKey** of the current tree.

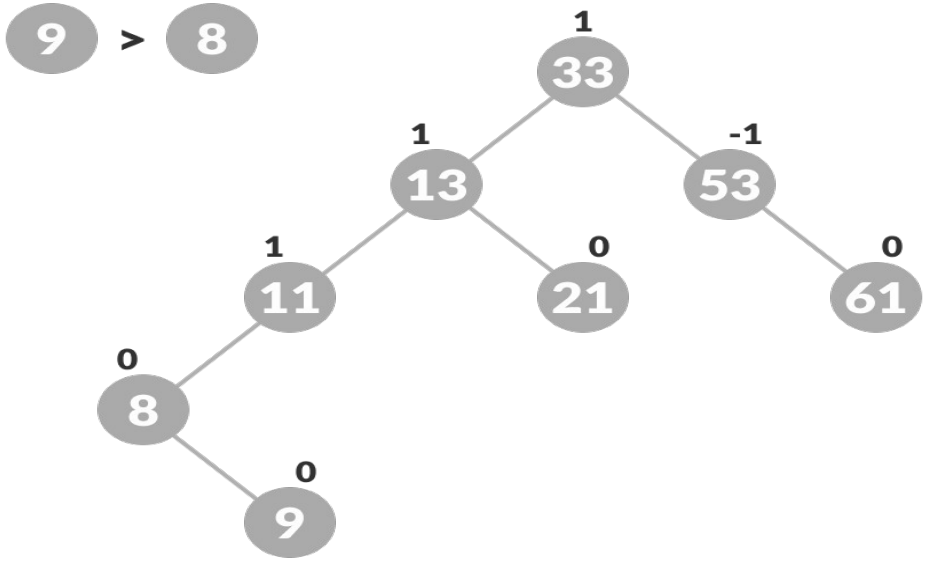
1. If **newKey** < **rootKey**, call insertion algorithm on the left subtree of the current node until the leaf node is reached.
2. Else if **newKey** > **rootKey**, call insertion algorithm on the right subtree of current node until the leaf node is reached.
3. Else, return leafNode



Finding the location to insert newNode

Compare **leafKey** obtained from the above steps with newKey:

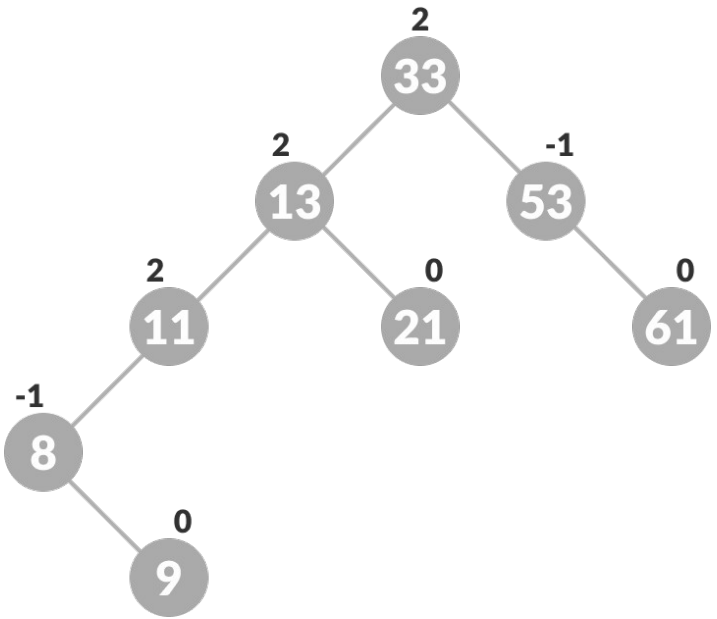
1. If **newKey** < **leafKey**, make **newNode** as the **leftChild** of leafNode.
2. Else, make **newNode** as **rightChild** of leafNode.



Update the balance factor and check if its balanced or imbalanced tree

Inserting the new node

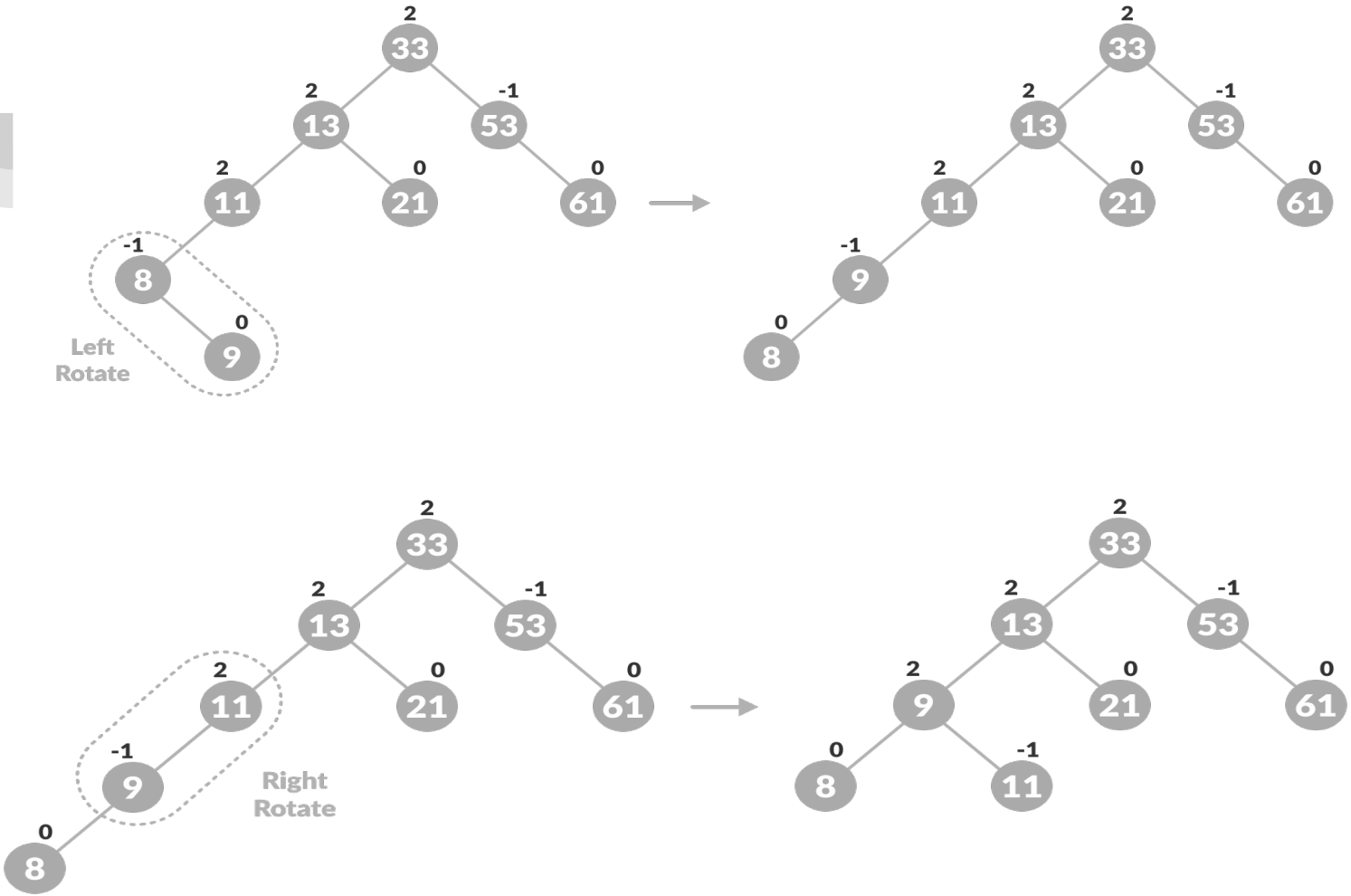
Updated **balanceFactor** of the nodes.



Updating the balance factor after insertion

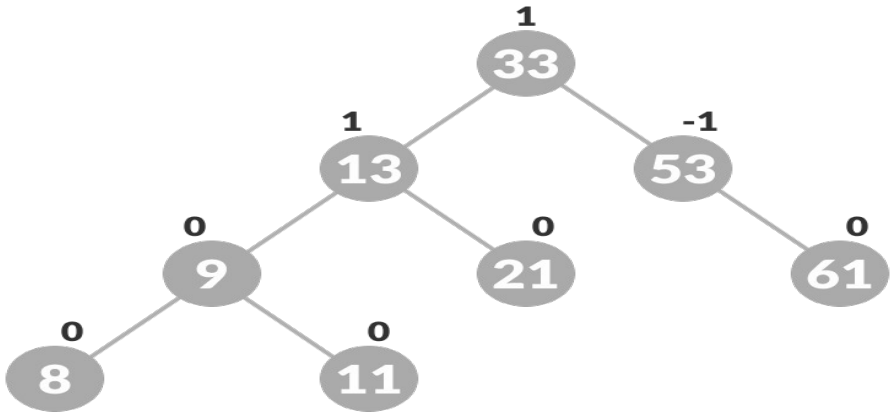
Let’s Revise how the tree can be balanced

If the nodes are unbalanced, then rebalance the node.
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.
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
If **newNodeKey > rightChildKey** do left rotation.
Else, do right-left rotation



Balancing the tree with rotation

Final balanced tree



Time complexity Of Insertion

For insertion operation, the running time complexity of the AVL tree is **O(log n)** for searching the position of insertion and getting back to the root.

The time complexity of the AVL tree remains **O(log n)** at all the times because the AVL tree is always balanced.

DELETION

There are three cases for deleting a node:

•If nodeToBeDeleted is the **leaf node** (i.e.., does not have any child), then remove nodeToBeDeleted.

•If nodeToBeDeleted has **one child**, then substitute the contents of nodeToBeDeleted with that of the child. Remove the child.

•If nodeToBeDeleted has **two children**, find the inorder successor or inorder predecessor of nodeToBeDeleted.

Let’s Revise how the tree can be balanced

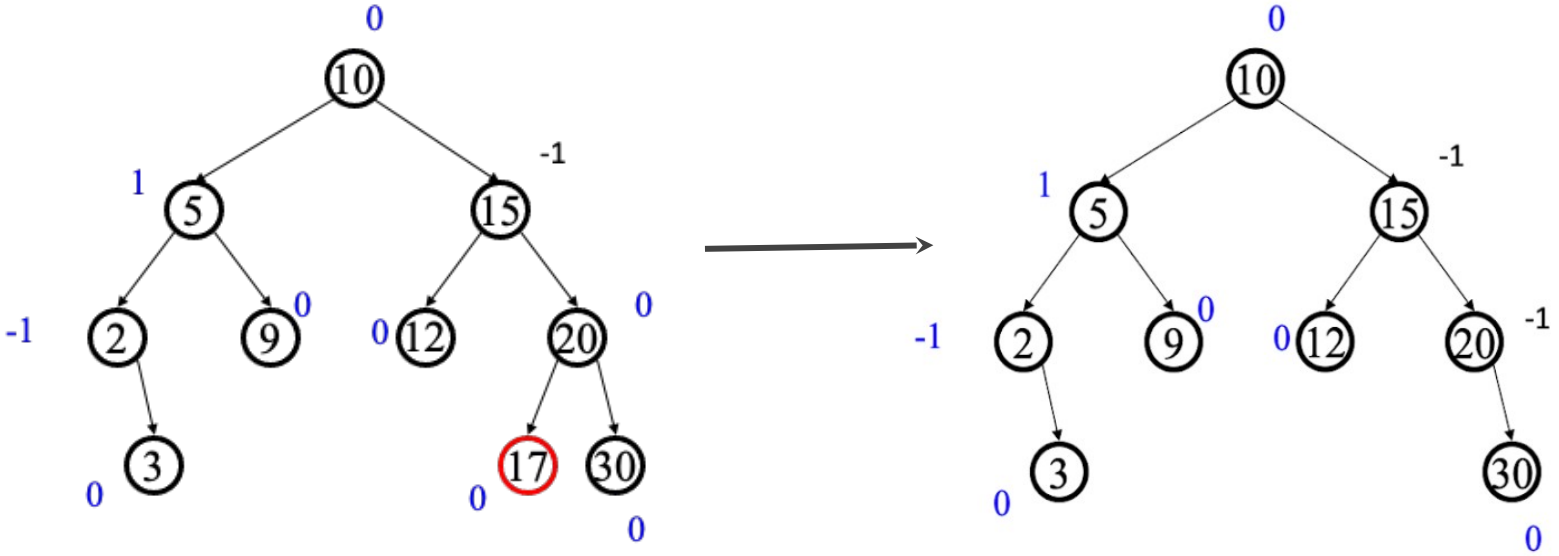
From the action position (parent node of the deleted node) , find the first imbalanced node and apply rotations on it.

If the nodes are unbalanced, then rebalance the node.

1. 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
    1. If **b.f of root_left >=0** do right rotation.
    2. Else, do left-right rotation.
2. 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
    1. If **b.f of root_right <=0** do left rotation.
    2. Else, do right-left rotation

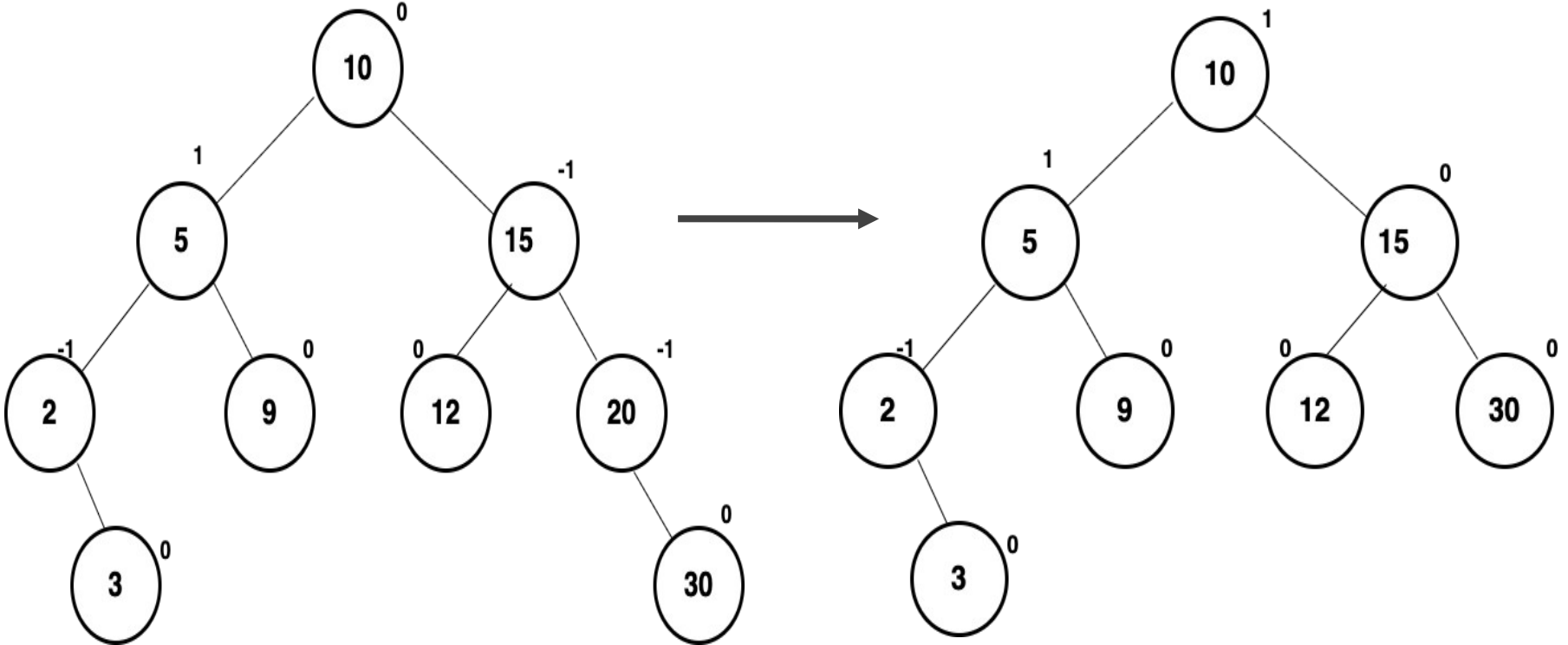
CASE 1: A NODE WITH NO CHILD

DELETE 17 FROM THE TREE.



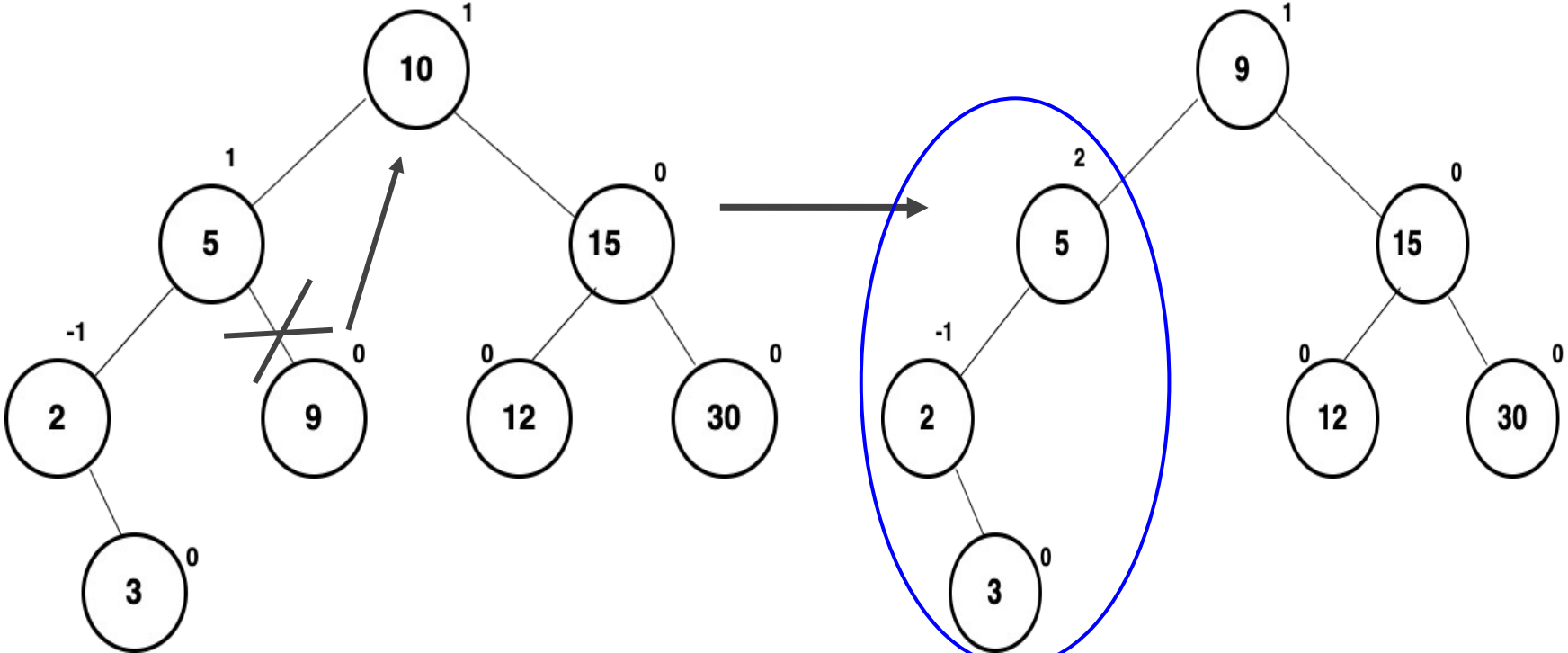
CASE 2: A NODE WITH ONLY ONE CHILD

DELETE 20 FROM THE TREE.



CASE 3:A NODE WITH TWO CHILDREN

DELETE THE ROOT NODE(10)



Balanced or Unbalanced ??

>> Replace the root node with inorder predecessor or inorder successor.(Here we chose inorder predecessor).

>> Update the balance tree after deletion.

>> Check unbalanced or balanced ? Yes, It is unbalanced

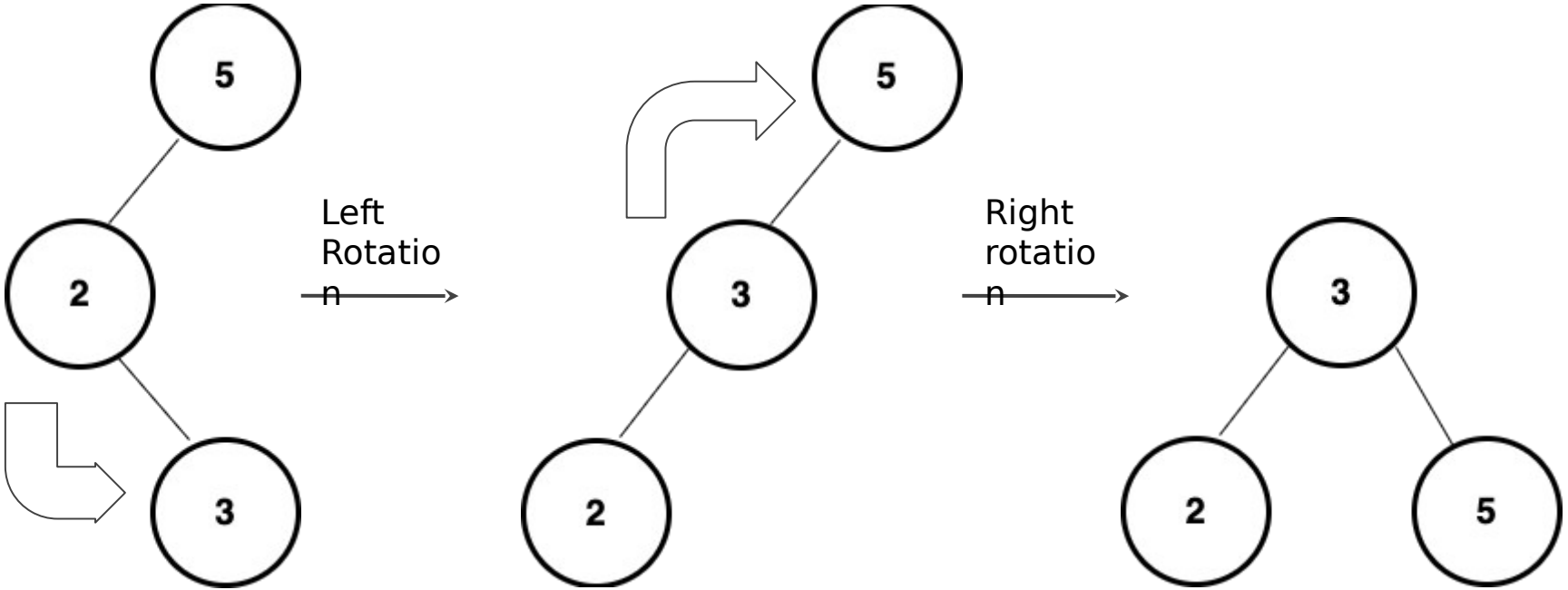
>> From the action position (parent node of the deleted node) , find the first imbalanced node and apply rotations on it.

>> Explanation: First imbalanced node : Node 5.

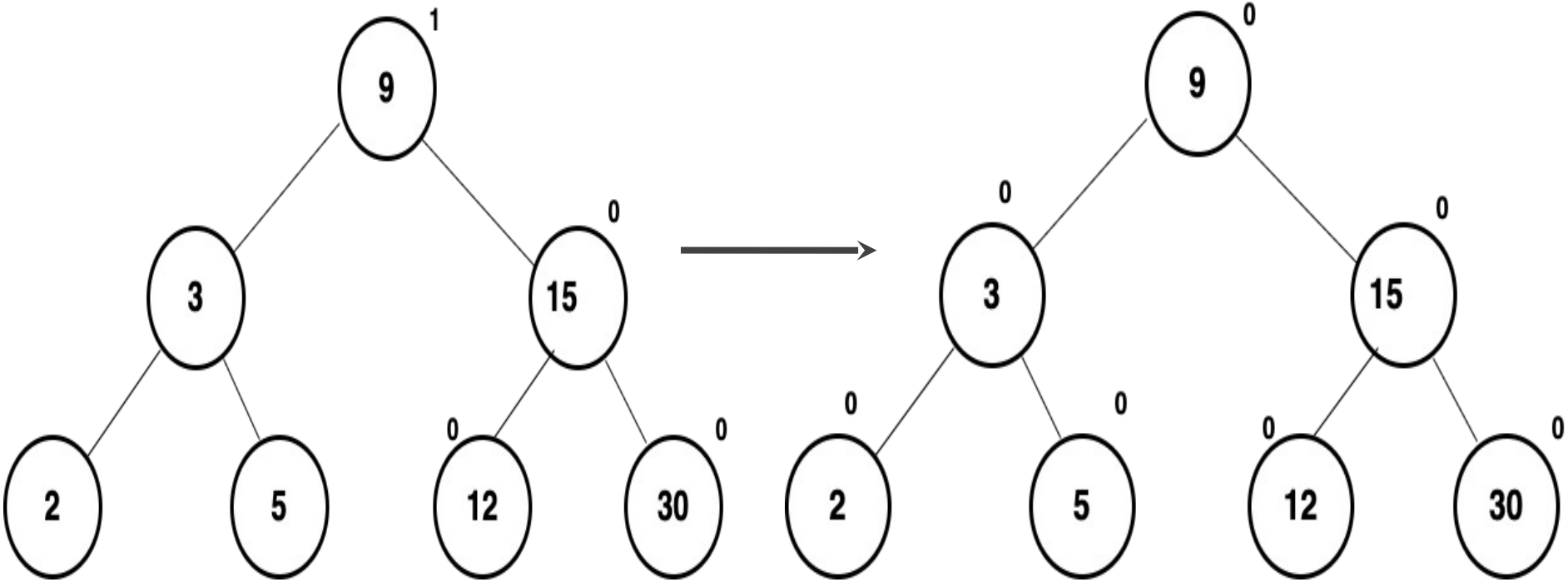
Balanced factor of node 5 is 2 which is greater than 1.

`balanceFactor` > 1, it means the height of the left subtree is greater than that of the right subtree.

If balance factor of root_leftkey >= 0 apply right rotation. Else apply left-right rotation.


5
2
3
Left
Rotatio
n
5
3
2
Right
rotatio
n
3
2
5

Update the balance factor



Time complexity of deletion operations :

Deletion	O(h)
Rotation	C1
Updating the height	C2
Updating balance factor	C3

Total time complexity = O(h) + C1+C2+C3
= O(h) where h is the height of the tree.
= O(log n)

Time complexity of AVL Trees

Operation	Best Case	Average Case	Worst Case
Insert	O(log n)	O(log n)	O(log n)
Delete	O(log n)	O(log n)	O(log n)

ADVANTAGES OF USING AVL TREES

- AVL trees have efficient search time complexity
- AVL tree has capabilities of self-balancing
- AVL tree is faster than the red-black tree as the AVL tree is more strictly balanced

APPLICATIONS OF AVL TREES

1. AVL tree is used when there are few operations of insertion and deletion are performed
2. It is used when a short search time is needed
3. AVL tree is used when the data is almost sorted and only a few operations are required to sort it completely.
4. AVL tree is effective to search the data fast and efficiently in large databases
5. It is widely used for sorting sets and dictionary data
6. AVL tree is used in applications where the searching should be effective rather than database application

SUMMARY

AVL tree is a self-balancing binary search tree where the balance of the tree is checked by the balance factor and modified whenever required by performing a rotation process. Insertion and Deletion time complexity of AVL tree is O(log n) and the searching time complexity of the AVL tree is O(n) which makes it better than binary search tree and red-black tree. AVL tree is used for faster and efficient searching of data in a large dataset and therefore, it is always recommended to learn and understand the AVL tree in detail.