UNIT-IV

AVL TREE

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as balance factor. The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.

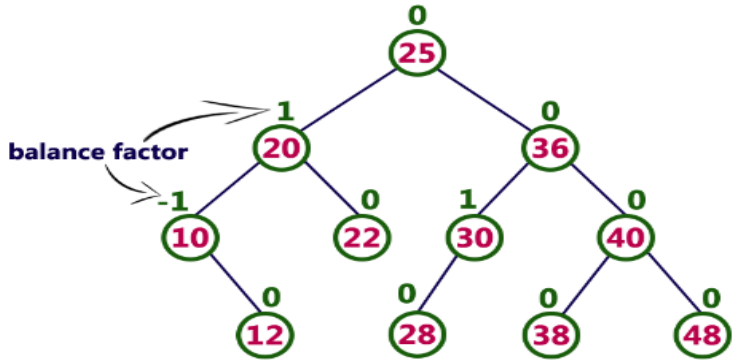
An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor:

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we calculate as follows...

Balance factor = heightOfLeftSubtree – heightOfRightSubtree

Example of AVL Tree



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

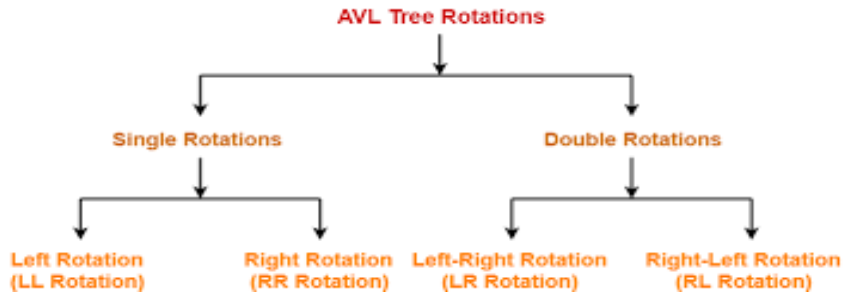
AVL Tree Rotations

In AVL tree, after performing operations like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

Rotation is the process of moving nodes either to left or to right to make the tree balanced.

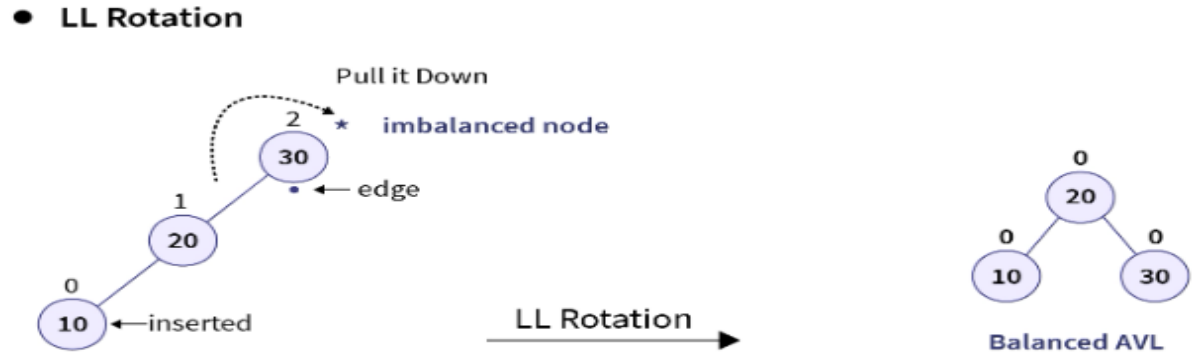
There are four rotations and they are classified into two types.



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:

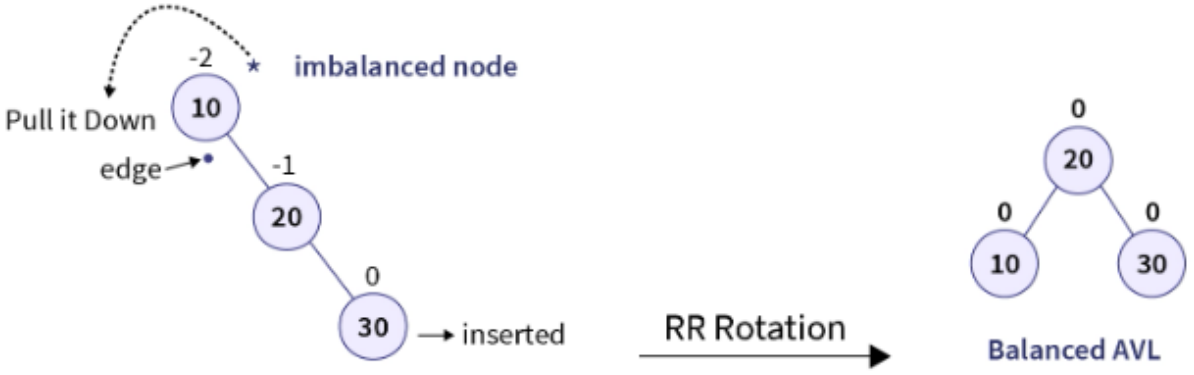


After calculating the balance factor of all nodes in the tree, it is evident that the insertion of element 10 has caused an imbalance in the root node, resulting in a left-skewed tree. This scenario represents a case of L-L insertion. To counteract the left skewness, a right rotation (clockwise rotation) is performed on the imbalanced node. This rotation shifts the heavier left subtree to the right side, restoring balance to the tree structure.

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:



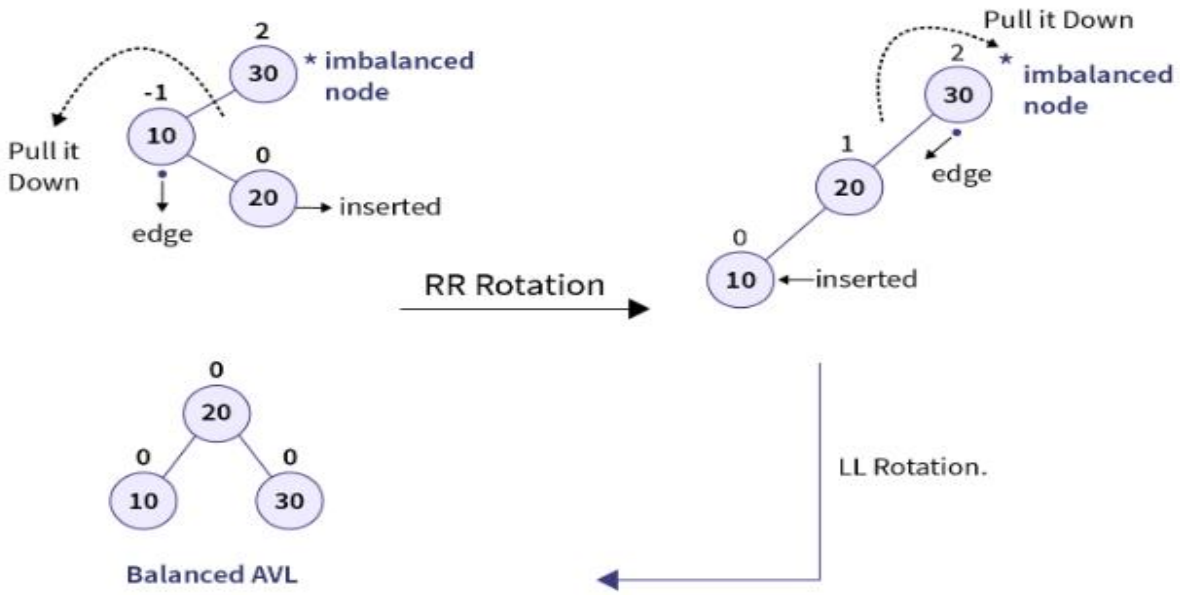
Upon calculating the balance factor of all the nodes, we can confirm that the root node of the tree is imbalanced (balance factor = 2) when the element 30 is inserted using RR-insertion.

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

In some cases, a single tree rotation is not sufficient to balance the tree after a height-affecting operation. One such case is the Left-Right (LR) insertion, where an imbalance occurs when a node is inserted into the right subtree of the left child of the imbalanced node. This situation can be illustrated with the example of creating a BST using elements 30, 10, and 20. Upon insertion of element 20 as the right child of the node with value 10, the tree becomes imbalanced with the root node having a balance factor of 2. Positive balance factors indicate left-heavy nodes, while negative balance factors indicate right-heavy nodes.

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:



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:

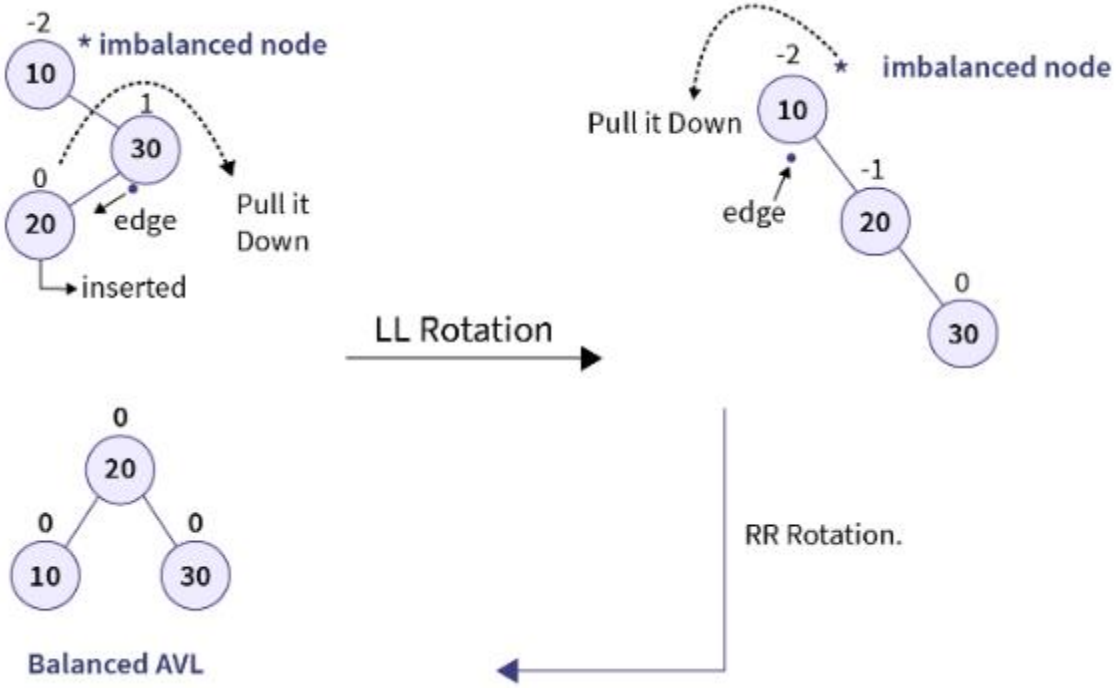
1. Apply RR Rotation (anti-clockwise rotation) on the left subtree of the imbalanced node as the left child of the imbalanced node is right-heavy. This process flips the tree and converts it into a left-skewed tree.
2. Perform LL Rotation (clock-wise rotation) on the imbalanced node to balance the left-skewed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



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.

Operations on an AVL Tree

The following operations are performed on AVL tree

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree.

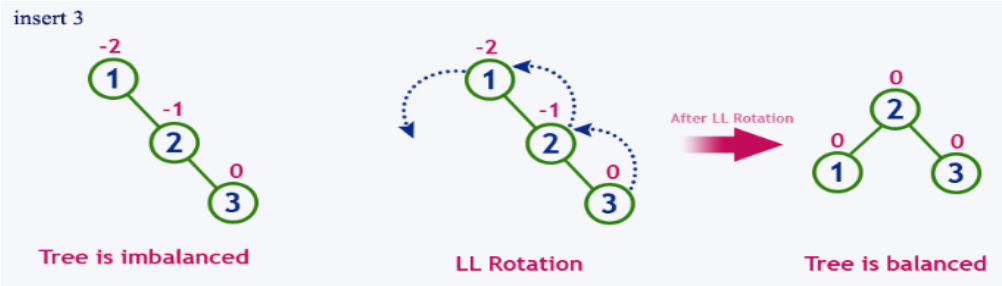
Step 1 - Read the search element from the user.
Step 2 - Compare the search element with the value of root node in the tree.
Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
Step 5 - If search element is smaller, then continue the search process in left subtree.
Step 6 - If search element is larger, then continue the search process in right subtree.
Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

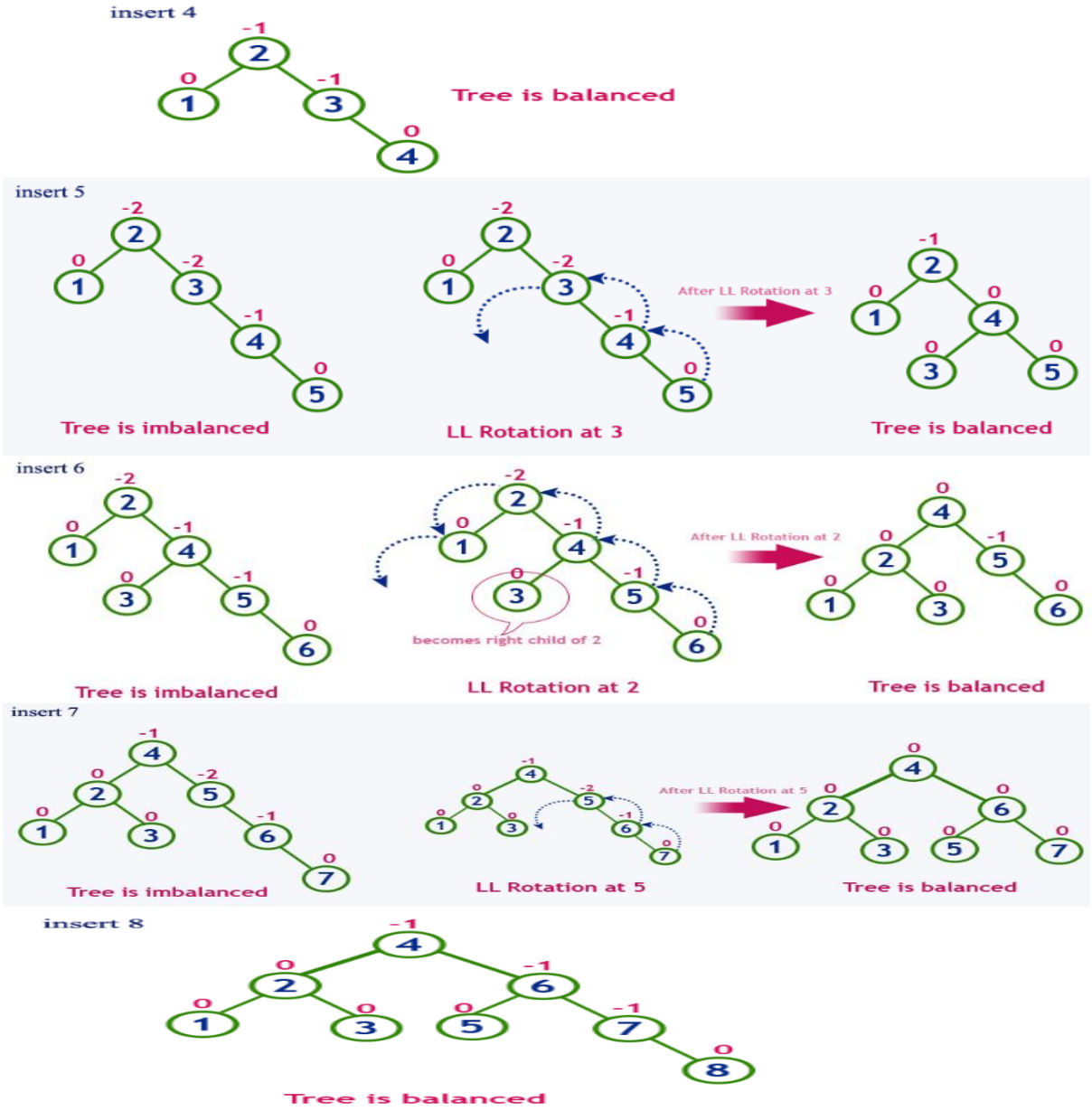
Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
Step 2 - After insertion, check the Balance Factor of every node.
Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.





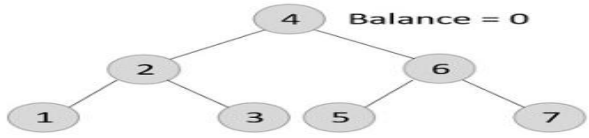
Deletion

An AVL tree is a self-balancing binary search tree where the height difference between its left and right subtrees (known as the balance factor) is limited to be at most 1. When elements are inserted or deleted from the AVL tree, it may become unbalanced, and rotations are performed to maintain the balance factor and keep the tree height in check.

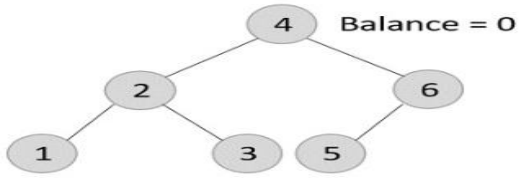
AVL tree deletion can be a bit more complex than insertion because we need to ensure that the tree remains balanced after the deletion operation. There are three main cases to consider when deleting a node from an AVL tree:

- **Case 1 (Deletion of a leaf node)** − If the node to be deleted is a leaf node, then it is deleted without any replacement as it does not disturb the binary search tree property. However, the balance factor may get disturbed, so rotations are applied to restore it.
- **Case 2 (Deletion of a node with one child)** − If the node to be deleted has one child, replace the value in that node with the value in its child node. Then delete the child node. If the balance factor is disturbed, rotations are applied.
- **Case 3 (Deletion of a node with two child nodes)** − In this case, we find the node's in-order predecessor (the maximum node in the left subtree) or the in-order successor (the minimum node in the right subtree), copy its value to the node to be deleted, and then recursively delete the predecessor/successor. If the balance factor exceeds 1 after deletion, apply balance algorithms.

Example:

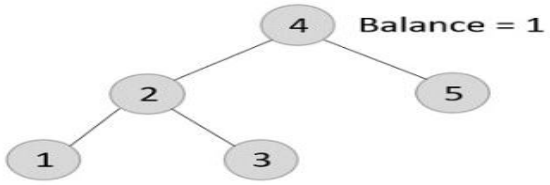


Deleting element 7 from the tree above: the element 7 is a leaf, we normally remove the element without disturbing any other node in the tree



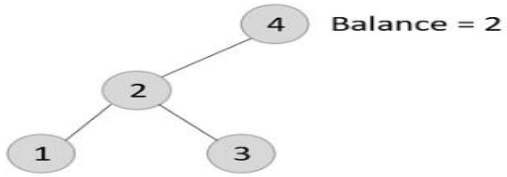
Delete element 6:

Element 6 is not a leaf node and has one child node attached to it. In this case, we replace node 6 with its child node: node 5.

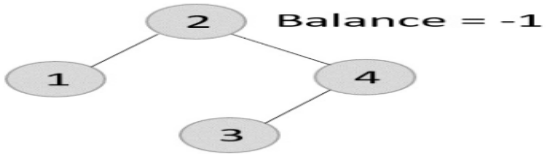


The balance of the tree becomes 1, and since it does not exceed 1 the tree is left as it is.

If we delete the element 5 further, we would have to apply the left rotations; either LL or LR since the imbalance occurs at both 1-2-4 and 3-2-4.

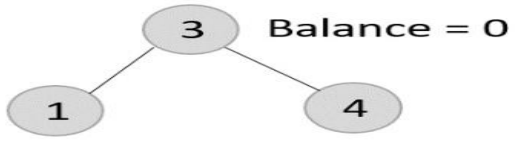


The balance factor is disturbed after deleting the element 5, therefore we apply LL rotation (we can also apply the LR rotation here). Once the LL rotation is applied on path 1-2-4, the node 3 remains as it was supposed to be the right child of node 2 (which is now occupied by node 4). Hence, the node is added to the right subtree of the node 2 and as the left child of the node 4.



Delete element 2

As mentioned in case 3, this node has two children. Therefore, we find its inorder successor that is a leaf node (say, 3) and replace its value with the inorder successor.



The balance of the tree still remains 0, therefore we leave the tree as it is without performing any rotations.

AVL Tree implementation using Java's Collection framework.

```
import java.util.Scanner;
import java.util.TreeSet;

public class AVLTreeEx {
        TreeSet<Integer> avlTree = new TreeSet<>();
        public void insertElement(int element) {
        avlTree.add(element);
        }

        public void deleteElement(int element) {
        avlTree.remove(element);
        }

        public void printTree() {
        System.out.println("AVL Tree: " + avlTree);
        }

        public static void main(String[] args) {
                AVLTreeEx a=new AVLTreeEx();
        Scanner scanner = new Scanner(System.in);

        while (true) {
        System.out.print("1. Insert element \n2. Delete element\n3. Display\n4. Exit\nEnter your choice:");
        int choice = scanner.nextInt();
        switch (choice) {
                case 1:
                        System.out.print("Enter element to insert: ");
                        int insEle = scanner.nextInt();
                                a.insertElement(insEle);
                 break;
                case 2:
                        System.out.print("Enter element to delete: ");
                        int delEle = scanner.nextInt();
                        a.deleteElement(delEle);
                 break;
                case 3:
                        a.printTree();
                break;
                case 4:
                      System.exit(0);
                break;
                default:
                        System.out.println("Invalid choice! Try again.");
        }
        System.out.println();
      }
   }
}
```

Red Black Tree

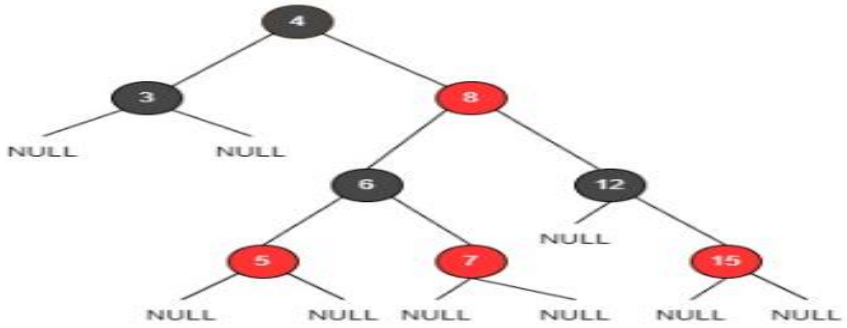
Red-Black Trees are another type of the Balanced Binary Search Trees with two coloured nodes: Red and Black. It is a self-balancing binary search tree that makes use of these colours to maintain the balance factor during the insertion and deletion operations. Hence, during the Red-Black Tree operations, the memory uses 1 bit of storage to accommodate the colour information of each node.

Def: Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.

In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

Properties of Red Black Tree

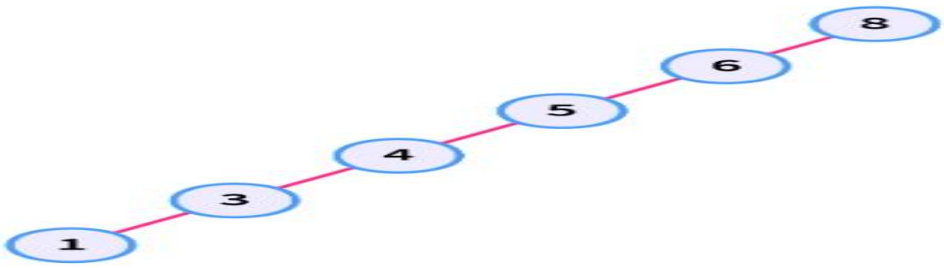
Property 1: Red - Black Tree must be a Binary Search Tree.
Property 2: The ROOT node must be colored BLACK.
Property 3: The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
Property 4: In all the paths of the tree, there should be same number of BLACK colored nodes.
Property 5: Every new node must be inserted with RED color.
Property 6: Every leaf (e.i. NULL node) must be colored BLACK.



Note: Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.

Why Red Black Trees?

All the operations in Binary Search Trees cost O(h) time to perform, where *h* h is the height of the tree which in the worst case (Skewed Tree) can go up to n as shown below -



Hence, if somehow we can place an upper bound on the maximum height of the tree we can achieve good time complexity to perform operations in a red-black tree.

Since we can guarantee that at any instance of time the height of the red-black tree will be O(log(n)) therefore we can place an upper bound of O(log(n)) on the time complexity of the search, insert, and delete operations.

But the same upper bound can also be achieved with AVL trees, so why do we even need red-black trees? The answer is, AVL trees will cause more number of rotations during inserting and deleting. So it is advisable to use red-black trees when we have a large number of insert/delete operations to be performed.

Comparison with AVL Tree

While both Red-Black trees and AVL trees are self-balancing binary search trees, they vary in their approach towards maintaining balance, thereby leading to different performance characteristics. AVL trees provide faster lookup times due to their stricter adherence to balance, ensuring that the difference in the height of the left and right sub-trees of any node does not exceed 1. On the other hand, Red-Black trees allow for faster insertions and deletions since they're less rigid about balance, permitting a greater height difference. Consequently, AVL trees are preferred when the task involves more frequent lookups, while Red-Black trees are better suited for situations with many insertions and deletions.

Black Height of Red-black trees

Black height of a node is defined as the number of black nodes on any path from the node (not inclusive) to its leaf nodes (null nodes, which are also black). This includes any black nodes encountered along that path but does not count the red nodes. By definition, every path from a node to its descendant leaves has the same number of black nodes, helping to maintain the tree's balanced structure. This property is critical in maintaining approximately log2(n) height in Red-Black trees, ensuring efficient searching, insertion, and deletion operations.

Insertion into RED BLACK Tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

1. Recolor
2. Rotation
3. Rotation followed by Recolor

The insertion operation in Red Black tree is performed using the following steps...

Step 1 - Check whether tree is Empty.
Step 2 - If tree is Empty then insert the newNode as Root node with color Black and exit from the operation.
Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.
Step 4 - If the parent of newNode is Black then exit from the operation.
Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.
Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.
Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

Example

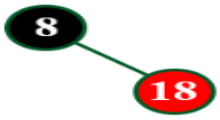
Create a RED BLACK Tree by inserting following sequence of number
8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

Tree is Empty. So insert newNode as Root node with black color.

insert (18)

Tree is not Empty. So insert newNode with red color.

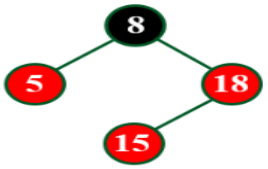


insert (5)

Tree is not Empty. So insert newNode with red color.

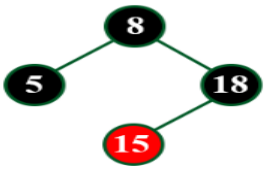
insert (15)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

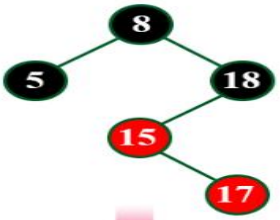
After RECOLOR



After Recolor operation, the tree is satisfying all Red Black Tree properties.

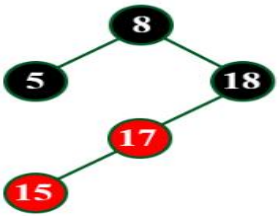
insert (17)

Tree is not Empty. So insert newNode with red color.

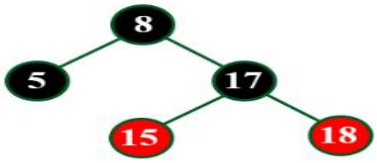


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

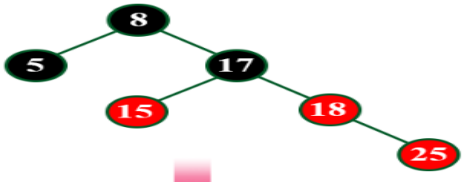


After Right Rotation & Recolor



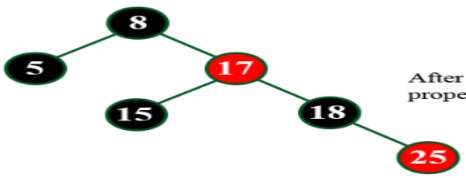
insert (25)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

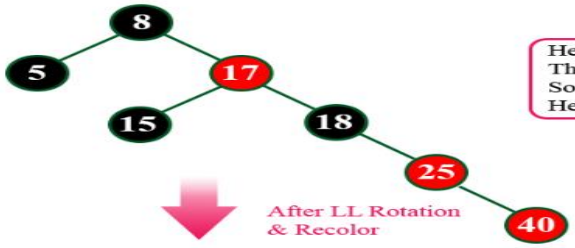
After Recolor



After Recolor operation, the tree is satisfying all Red Black Tree properties.

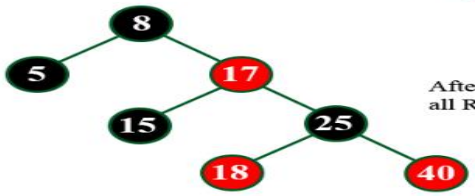
insert (40)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

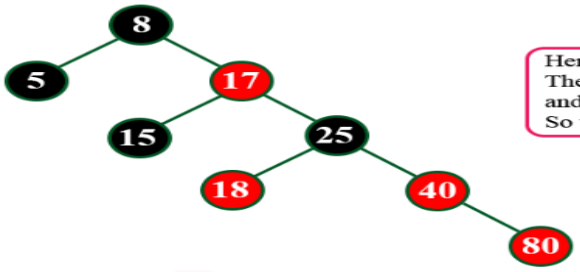
After LL Rotation & Recolor



After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

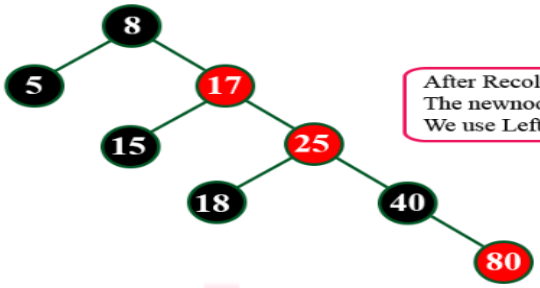
insert (80)

Tree is not Empty. So insert newNode with red color.



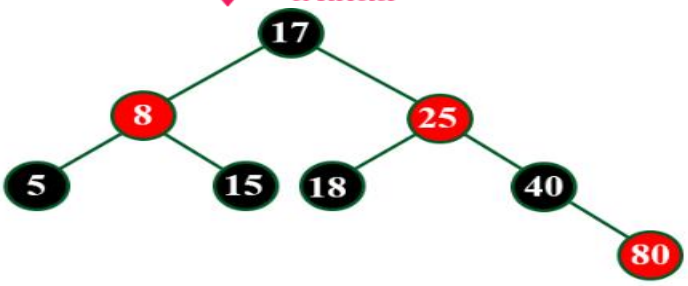
Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



After Recolor again there are two consecutive Red nodes (17 & 25).
The newnode's parent sibling color is Black. So we need Rotation.
We use Left Rotation & Recolor.

After Left Rotation & Recolor



Finally above tree is satisfying all the properties of Red Black Tree and it is a perfect Red Black tree.

B-Tree

In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children. But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children. B-Tree was developed in the year 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.

B-Tree can be defined as follows

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

B-Tree of Order m has the following properties

Property 1 - All leaf nodes must be at same level.

Property 2 - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.

Property 3 - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.

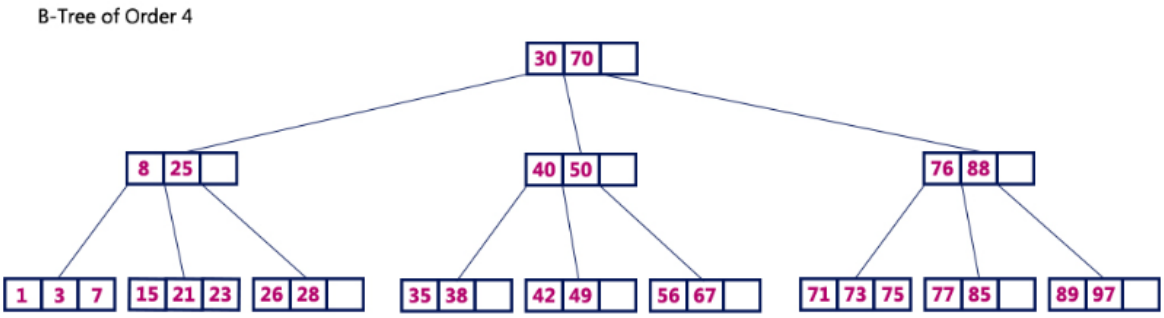
Property 4 - If the root node is a non leaf node, then it must have atleast 2 children.

Property 5 - A non leaf node with n-1 keys must have n number of children.

Property 6 - All the key values in a node must be in Ascending Order.

For example, B-Tree of Order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.

Example:



Why Do You Need B Tree Data Structure?

The B tree data structure is needed due to the following reasons:

- B Tree is an extension of M-way tree. While B trees are self-balanced, M way trees can be balanced, skewed or any way. In case of external storage, there is a need for faster access. An M-way tree can help ease searches for external storage more efficiently than a normal Binary Search Tree. However, due to the self-balancing nature of a B tree, it had fewer levels and thus the access-time is cut short to a huge extent by a B Tree.
- A B tree facilitates ordered sequential access and simplifies insertion and deletion of records when there are millions of records. This is possible due to the reduced height and balanced nature of the B tree.

Operations on a B-Tree

The following operations are performed on a B-Tree

- Search
- Insertion
- Deletion

Search Operation in B-Tree

The search operation in B-Tree is similar to the search operation in Binary Search Tree. In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but here we make an n-way decision every time. Where 'n' is the total number of children the node has. In a B-Tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with first key value of root node in the tree.
Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.
Step 5 - If search element is smaller, then continue the search process in left subtree.
Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only. The insertion operation is performed as follows.

Step 1 - Check whether tree is Empty.
Step 2 - If tree is Empty, then create a new node with new key value and insert it into the tree as a root node.
Step 3 - If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
Step 5 - If that leaf node is already full, split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
Step 6 - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

Construct a B-Tree of Order 3 by inserting numbers from 1 to 10.

Max keys =m-1=3-1=2 Min Keys=m\2-1=1
Max Children's =m=3 Min Children's=m/2=2

insert(1)

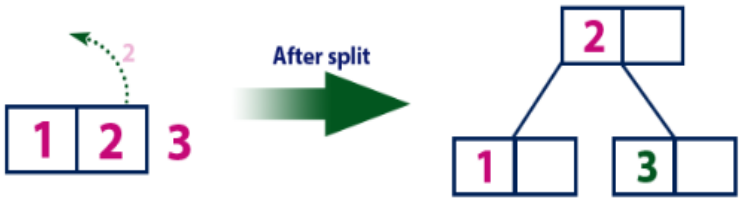
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

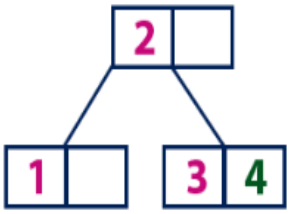
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.

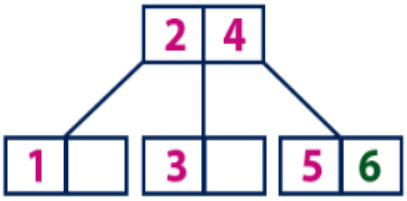


insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

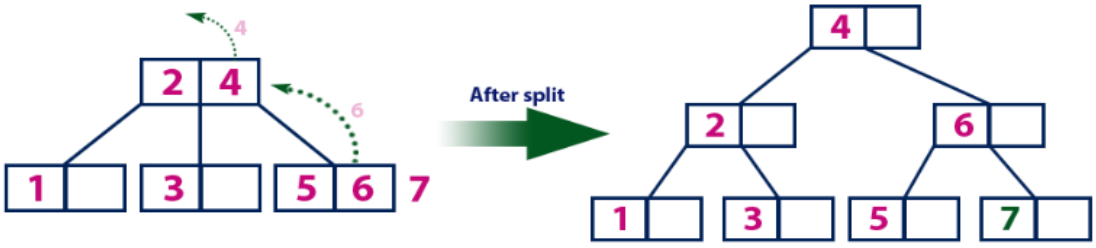
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



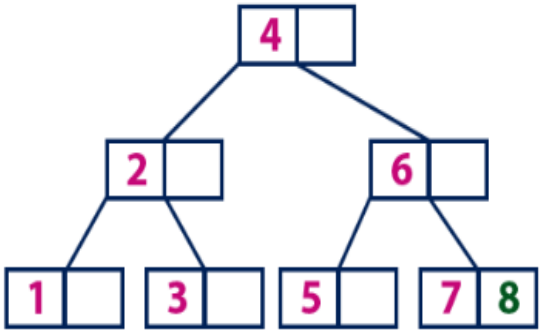
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



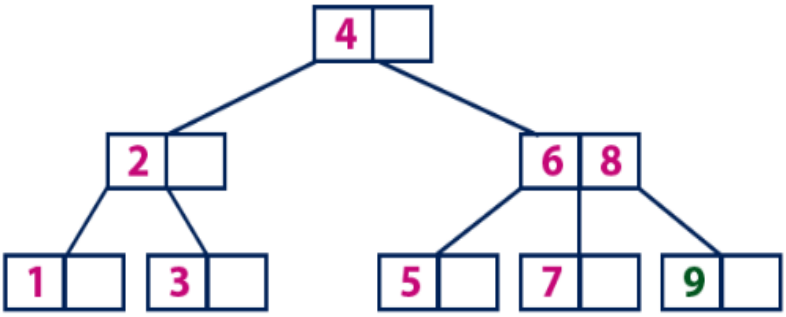
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



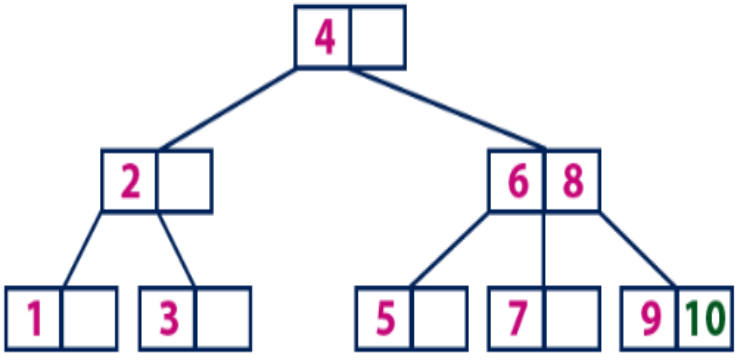
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



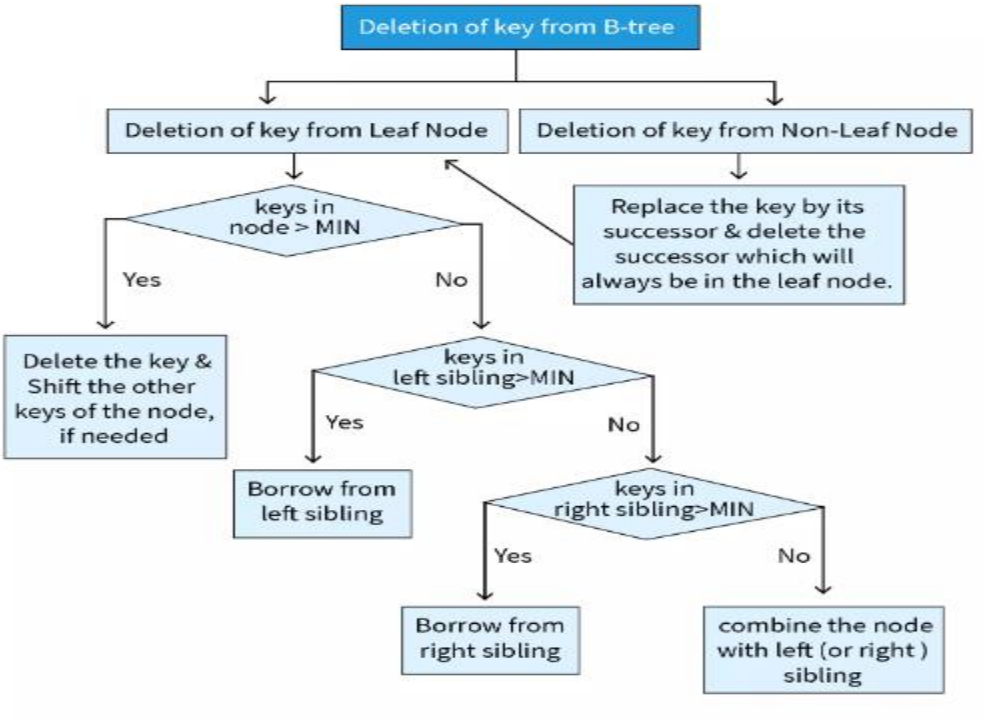
Deletion Operation on B Tree

During insertion, we had to ensure that the number of keys in the node doesn't cross a maximum. Similarly, during deletion, we need to ensure that the number of keys in the node after deletion doesn't go below the minimum number of keys that a node can hold. Thus, in case of deletion, a combination process takes place instead of a split.

Deletion can be studied using 2 cases:

1. Deletion from a leaf node.
2. Deletion from a non-leaf node.

The 2 cases can be represented by the following flowchart:



Case 1 - Deletion from Leaf Node

1.1. If the node has more than MIN keys - Deletion of key does not violate any property and thus the key can be deleted easily by shifting other keys of the node, if required.

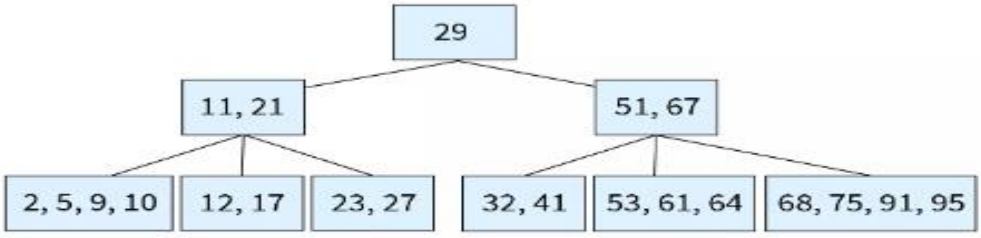
1.2. If the node has less than MIN keys - This kind of deletion violates a property of B tree. In case the keys in the left sibling are greater than MIN, keys are borrowed from there. If the keys in right sibling are greater than MIN, then keys are borrowed from there. If either of these do not hold true, then a combination of node takes place with either of the siblings.

Case 2 - Deletion from Non-Leaf Node

In this case, the successor key (smallest key in the right subtree) is copied at the place of the key to be deleted and then the successor is deleted. This case further reduces to Case 1, i.e. deletion from a leaf node.

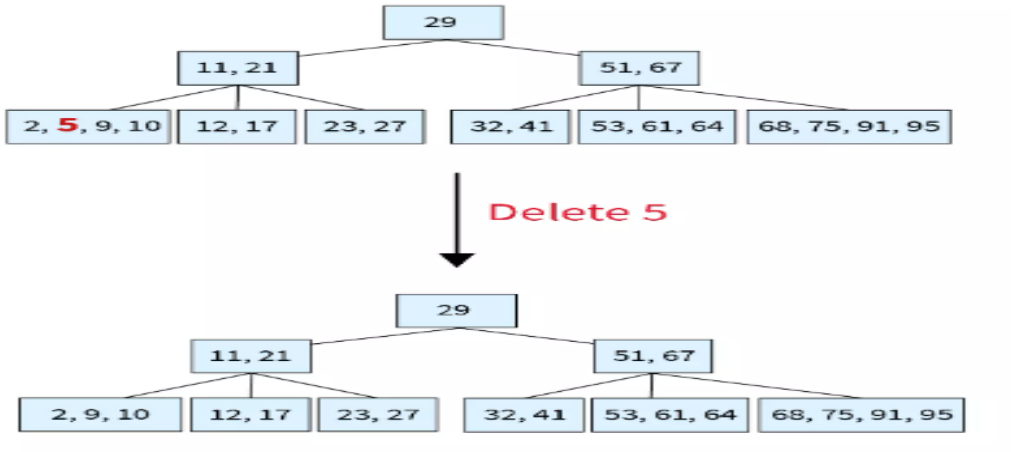
Now, let's understand deletion using an example. Consider the following B tree of order 4 with the nodes 5, 12, 32 and 53 to be deleted in the given order.

Since the order of the B tree is 4, the minimum and maximum number of keys in a node are 2 and 4 respectively.



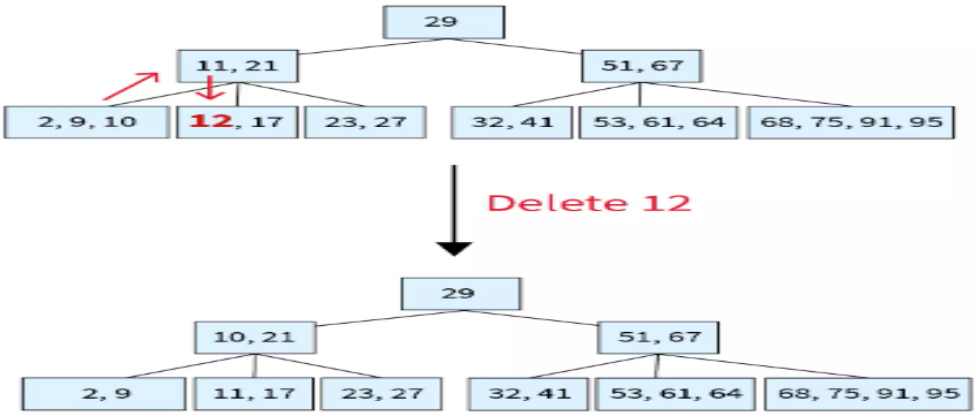
Step 1 - Deleting 5

Since 5 is a key in a leaf node with keys>MIN, this would be Case 1.1. A simple deletion with key shift would be done. Keys 9 and 10 are shifted left to fill the gap created by the deletion of 5.



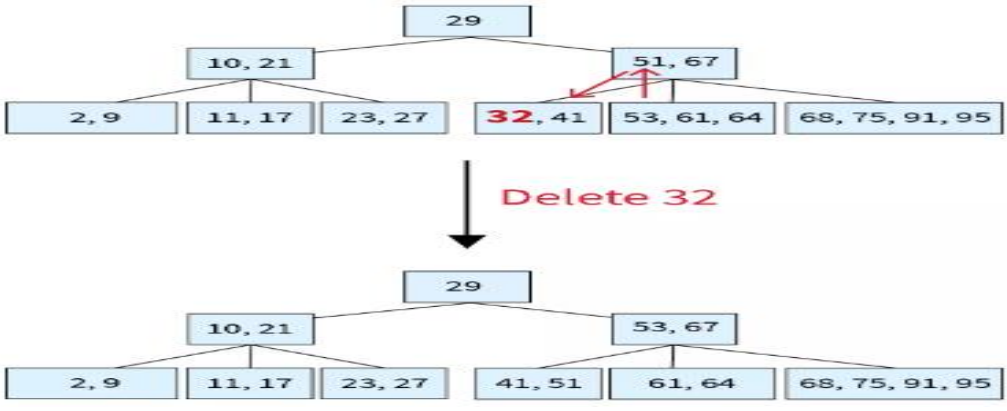
Step 2 - Deleting 12

Here, key 12 is to be deleted from node [12,17]. Since this node has only MIN keys, we will try to borrow from its left sibling [2,9,10] which has more than MIN keys. The parent of these nodes [11,21] contains the separator key 11. So, the last key of left sibling (10) is moved to the place of the separator key and the separator key is moved to the underflow node (the node where deletion took place). The resulting tree after deletion can be found as follows:



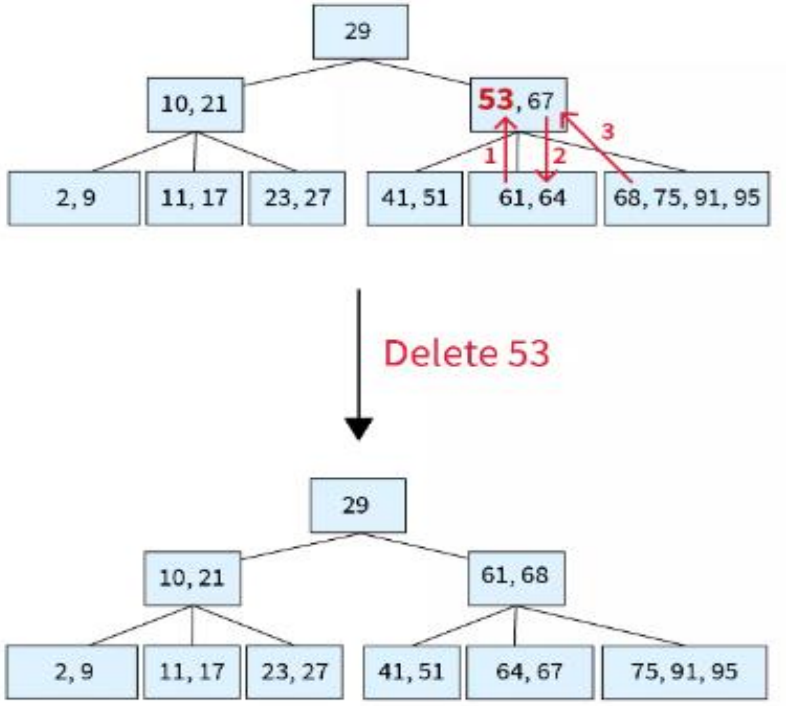
Step 3 - Deleting 32

Here, key 32 is to be deleted from node [32, 41]. Since this node has only MIN keys and does not have a left sibling, we will try to borrow from its right sibling [53, 61, 64] which has more than MIN keys. The parent of these nodes [51, 67] contains the separator key 51. So, the first key of right sibling (53) is moved to the place of the separator key and the separator key is moved to the underflow node (the node where deletion took place). The resulting tree after deletion can be found as follows:



Step 4 - Deleting 53

Here key 53 is to be deleted from node [53, 67] which is a non-leaf node. In such a case, the successor key (61) will be copied in place of 53 and now the task reduces to deletion of 61 from the leaf node. Since this node would have less than MIN keys, we check for the left sibling. Since the left sibling has only MIN keys, we move to the right sibling. The leftmost key of the right sibling (68) moves to the parent node and replaces the separator (67) while the separator shifts to the underflow node making it [64,67]. The resulting tree after deletion can be found as follows



Application of B Tree

- **Database or File System** - Consider having to search details of a person in Yellow Pages (directory containing details of professionals). Such a system where you need to search a record among millions can be done using B Tree.
- **Search Engines and Multilevel Indexing** - B tree Searching is used in search engines and also querying in MySql Server.
- **To store blocks of data** - B Tree helps in faster storing blocks of data in secondary storage due to the balanced structure of B Tree.