UNIT-III

AVL Trees: Representation and its advantages, Operations in AVL Trees-insertion, deletion and rotation.

B-Trees: Definition and advantages, B-Tree of Order M, Insertion and Searching in B-trees.

Introduction to Red-Black Trees and Splay Trees.

****

AVL Tree

AVL Tree is invented by GM **A**delson - **V**elsky and EM **L**andis in 1962. The tree is named AVL in honour of its inventors.

- 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 sub trees 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 extra information known as balance factor.

  An AVL tree is defined as follows...

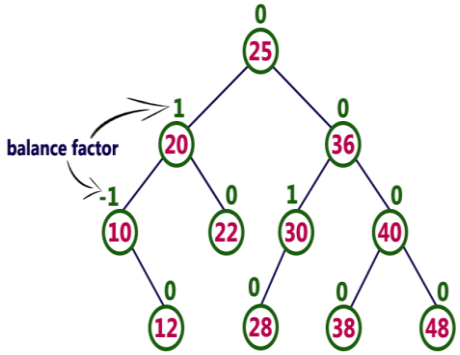
An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

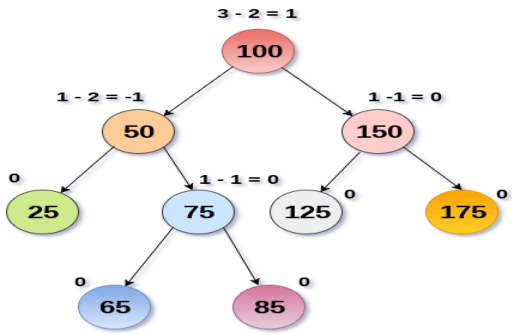
Balance factor: Balance factor of a node is the difference between the heights of the left and right sub trees of that node. The balance factor of a node is calculated either **height of left sub tree - height of right sub tree** (OR) **height of right sub tree - height of left sub tree**.

In the following explanation, we calculate as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

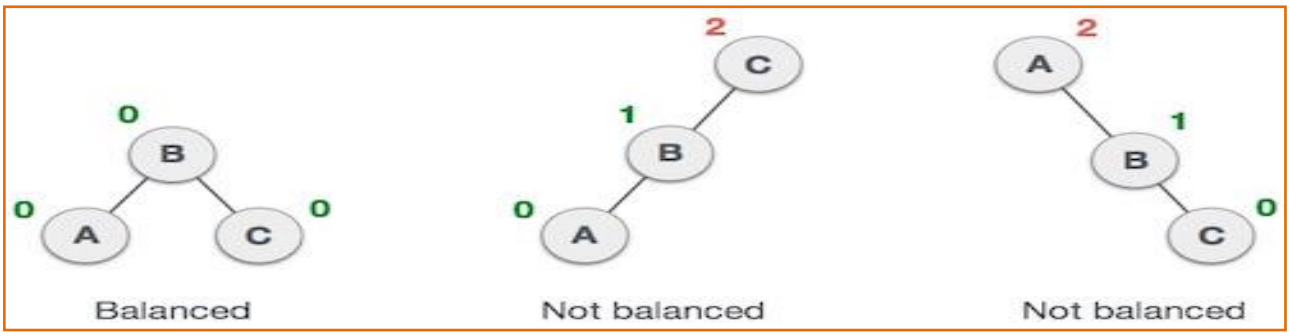
Example of AVL Tree:





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

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



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

We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1

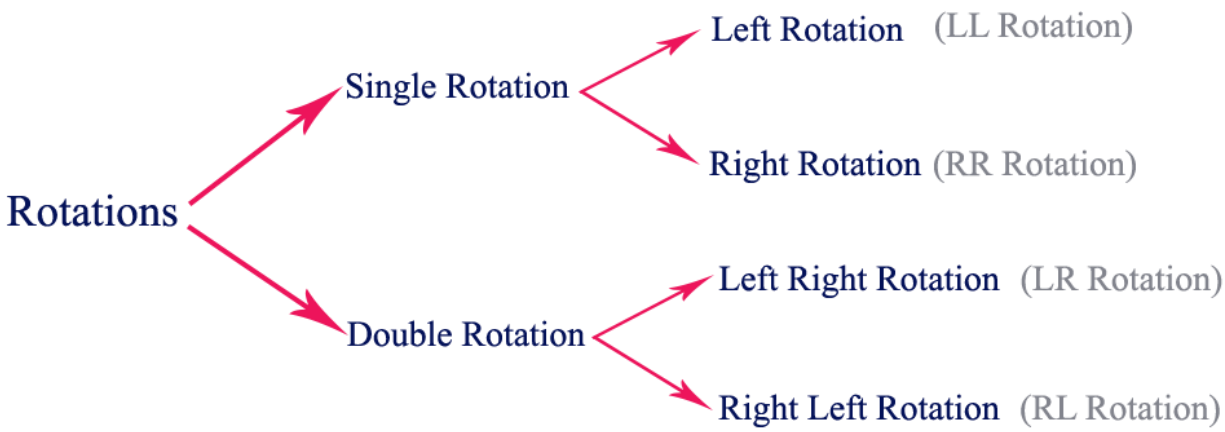
AVL Tree Rotations

- 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.

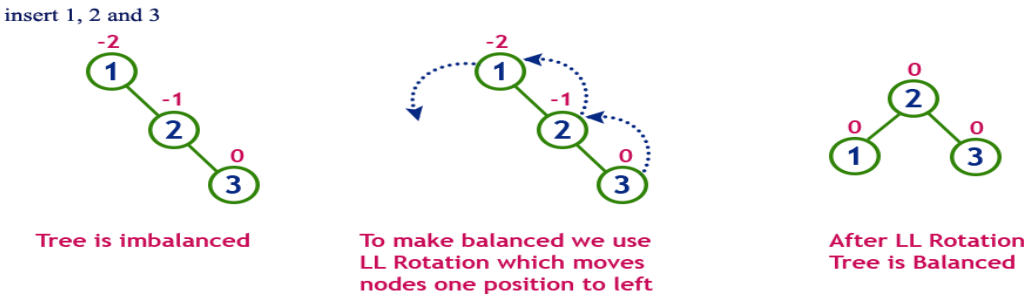
There are four rotations and they are classified into two types.



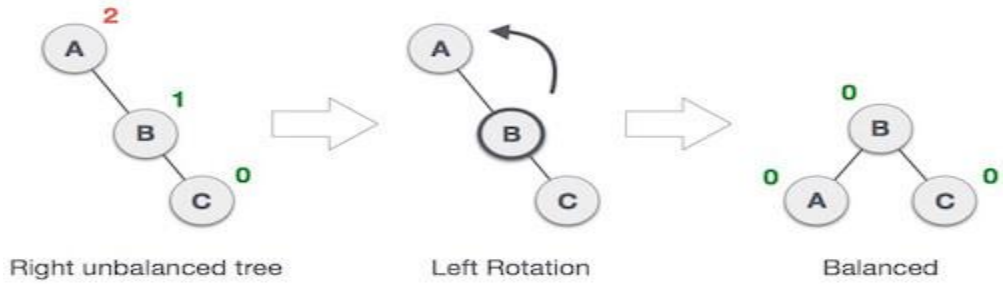
1. Single Left Rotation (LL Rotation):

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...

Ex-1:



Ex-2:

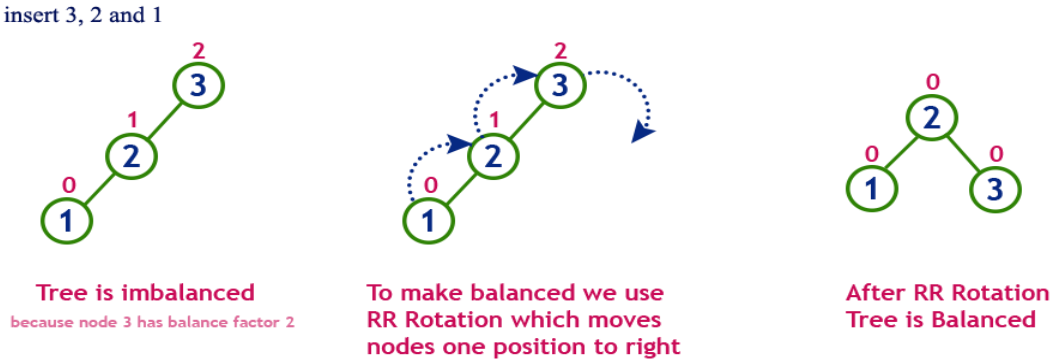


In above example, node A has balance factor -2 because a node C is inserted in the right sub tree of A right sub tree. We perform the RR rotation on the edge below A.

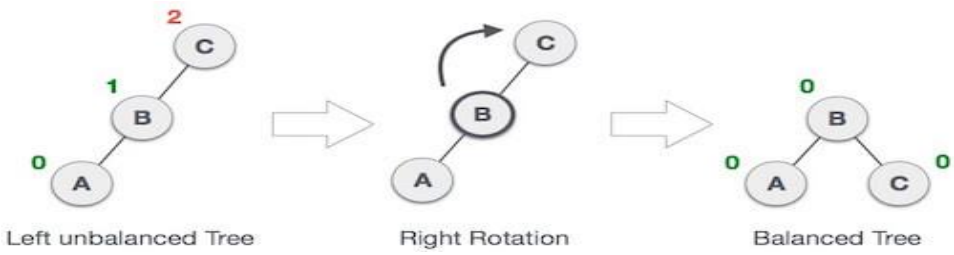
2. Single Right Rotation (RR Rotation):

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

Ex-1:



Ex-2:

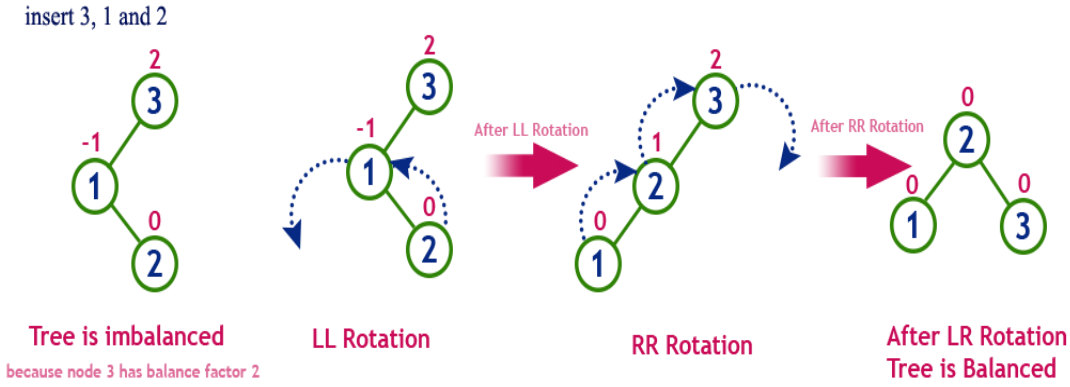


In above example, node C has balance factor 2 because a node A is inserted in the left sub tree of C left sub tree. We perform the LL rotation on the edge below A.

3. Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...

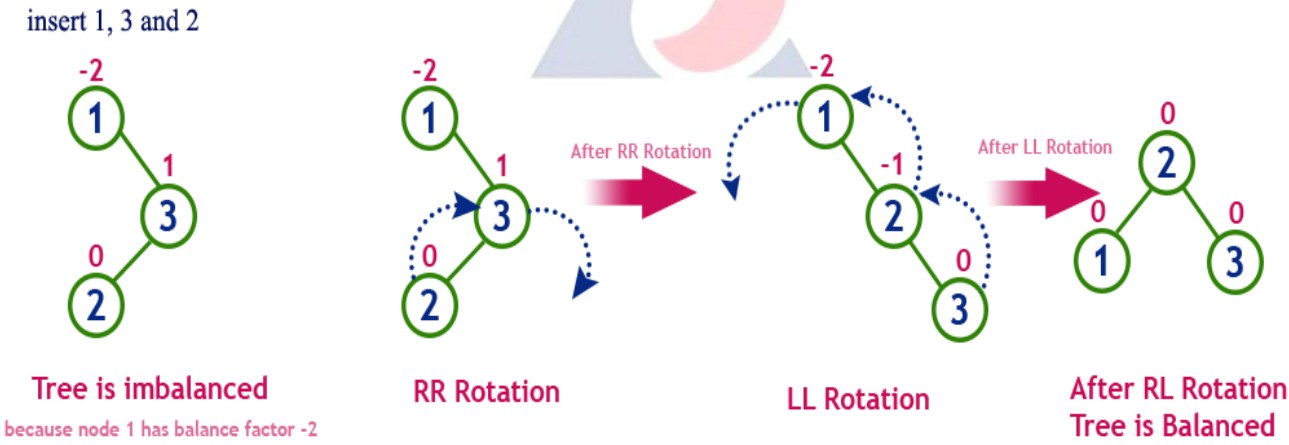
Ex-1:



4. Right Left Rotation (RL Rotation):

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...

Ex-1:



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. **Insertion**
2. **Deletion**
3. **Search**

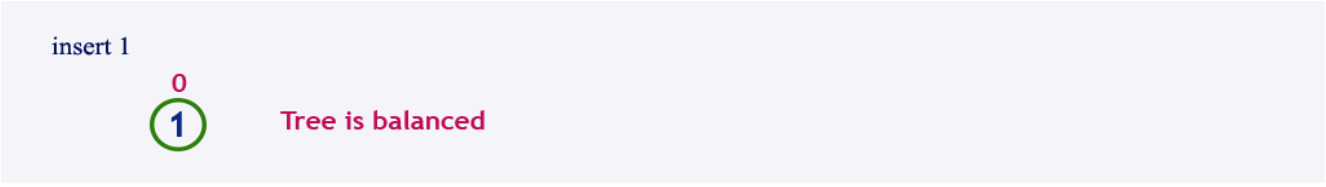
1. Insertion Operation in AVL Tree:

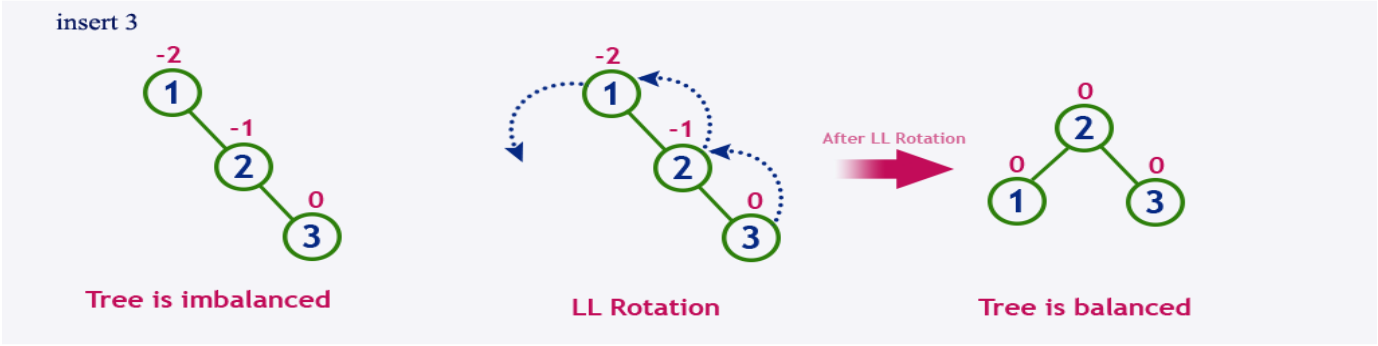
In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

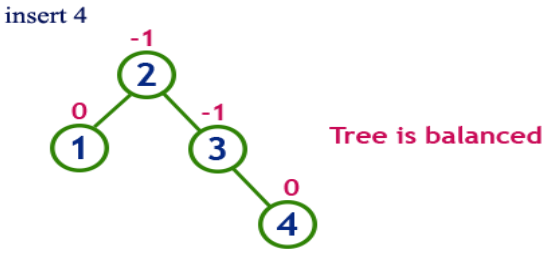
- **Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2 -** After insertion, check the **Balance Factor** of every node.
- **Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4 -** If the **Balance 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.

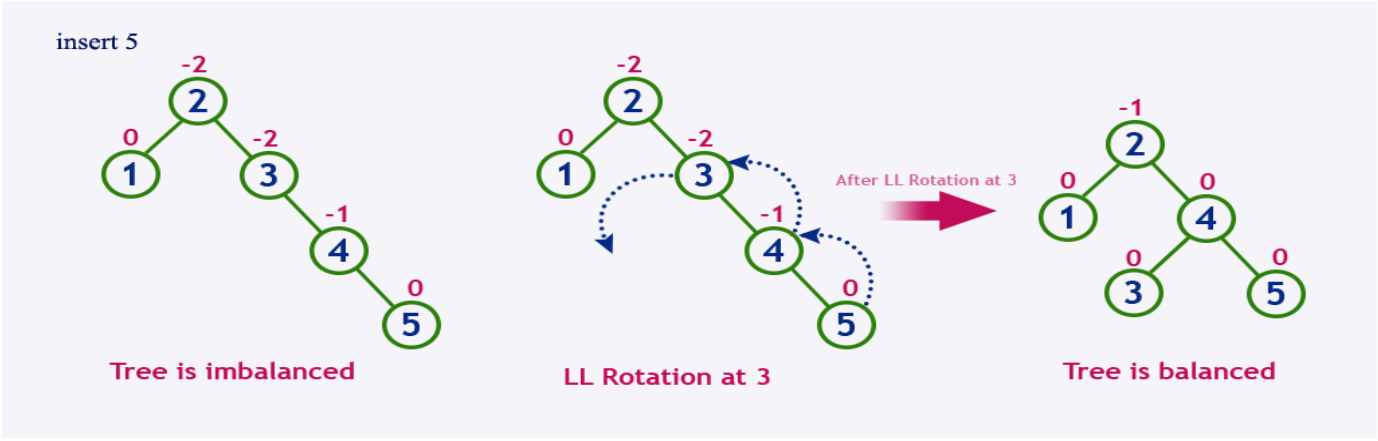
SN	Rotation	Description
1	LL Rotation	The new node is inserted to the right sub-tree of right sub-tree of critical node.
2	RR Rotation	The new node is inserted to the left sub-tree of the left sub-tree of the critical node.
3	LR Rotation	The new node is inserted to the right sub-tree of the left sub-tree of the critical node.
4	RL Rotation	The new node is inserted to the left sub-tree of the right sub-tree of the critical node.

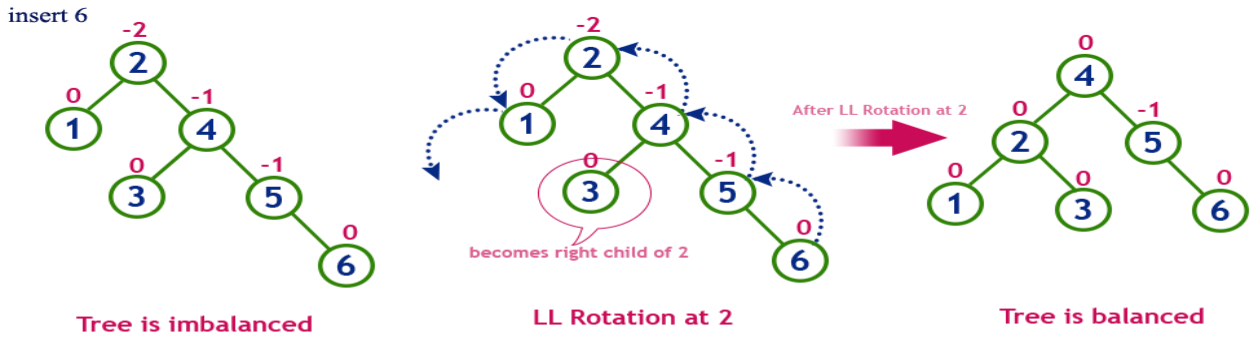
Example:1- Construct an AVL Tree by inserting numbers from 1 to 8.

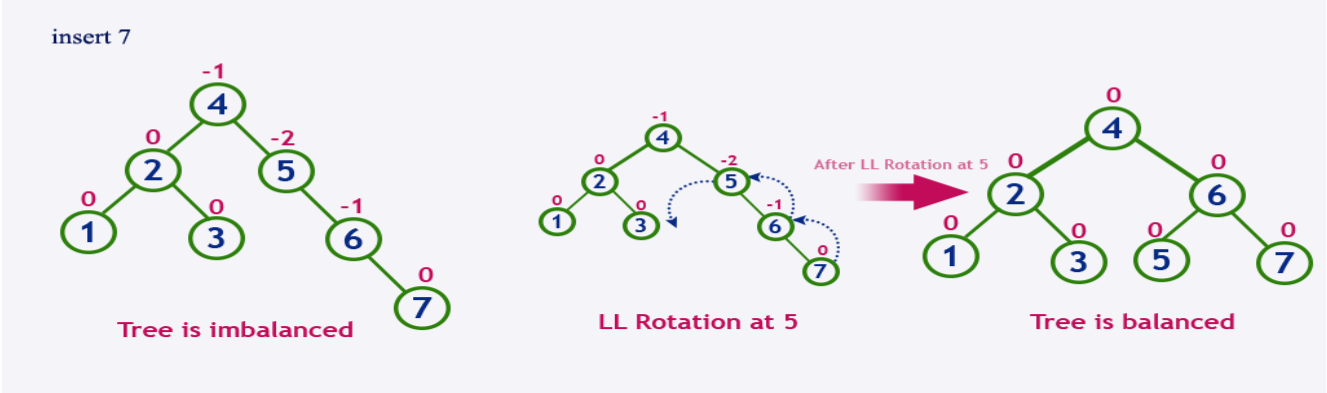


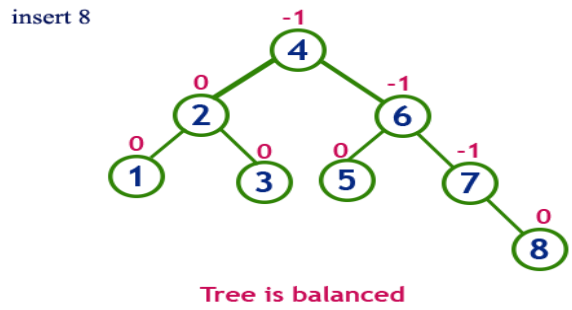

insert 3
Tree is imbalanced
LL Rotation
After LL Rotation
Tree is balanced


insert 4
Tree is balanced


insert 5
Tree is imbalanced
LL Rotation at 3
After LL Rotation at 3
Tree is balanced


insert 6
Tree is imbalanced
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
Tree is balanced

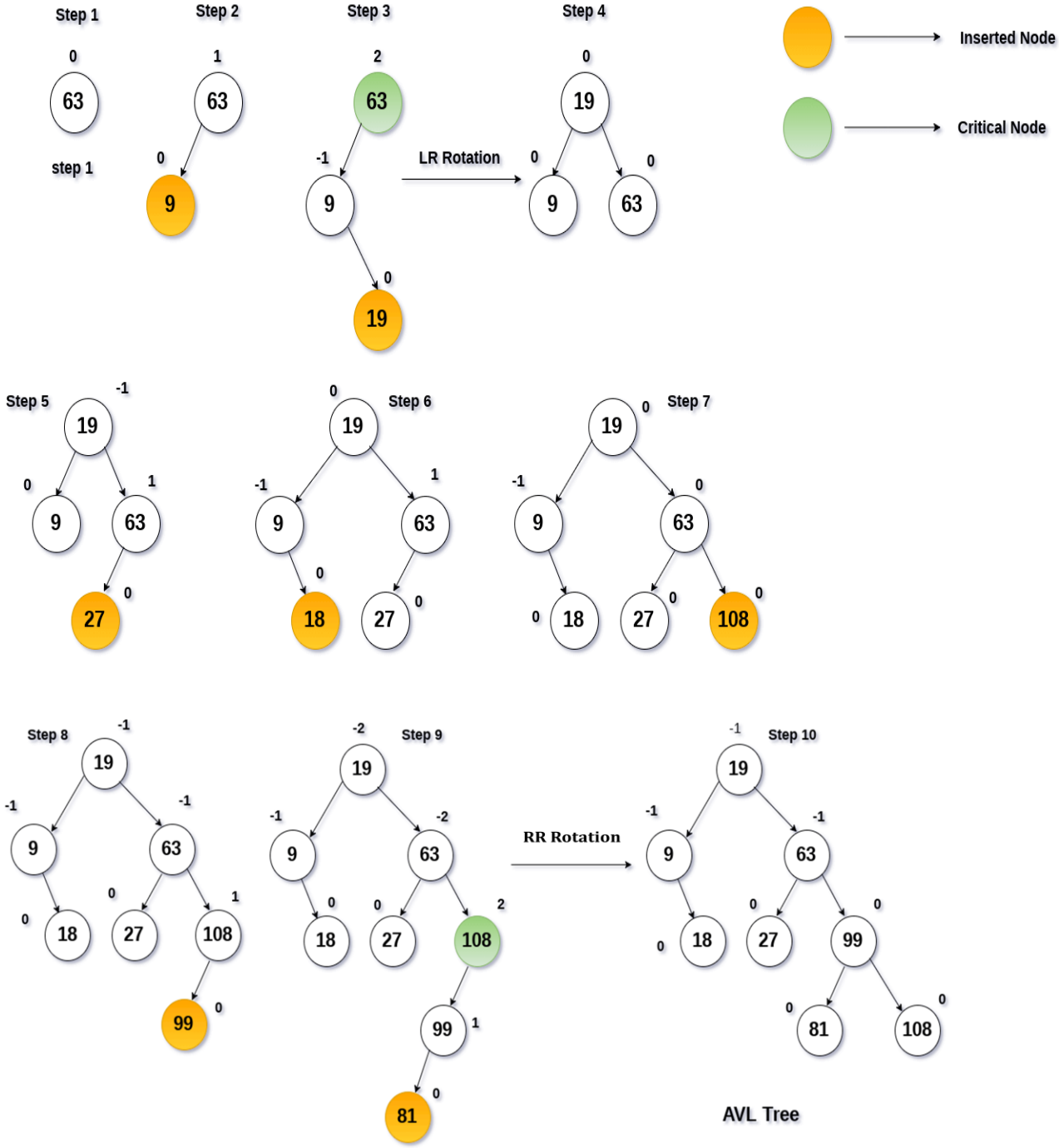

insert 7
Tree is imbalanced
LL Rotation at 5
After LL Rotation at 5
Tree is balanced



Example-2: Construct an AVL tree by inserting the following elements in the given order.

63, 9, 19, 27, 18, 108, 99, 81

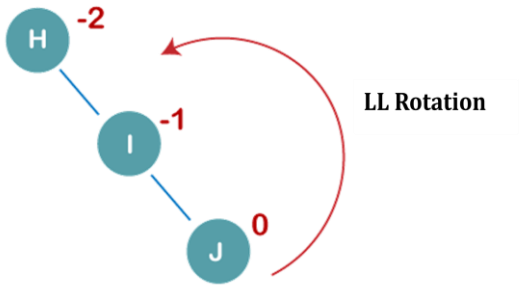
All the elements are inserted in order to maintain the order of binary search tree.



Example-3: Construct an AVL tree having the following elements:

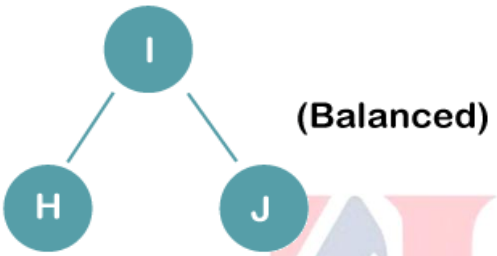
H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J

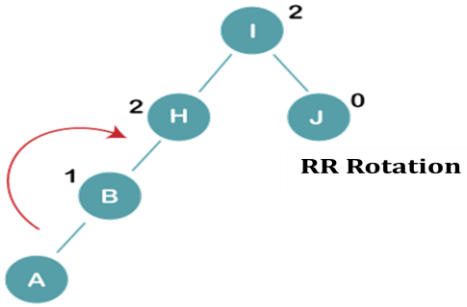


- On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2.
- Since the BST is right-skewed, we will perform LL Rotation on node H.

The resultant balance tree is:

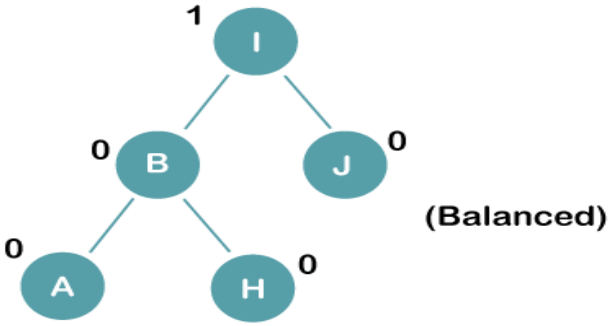


2. Insert B, A

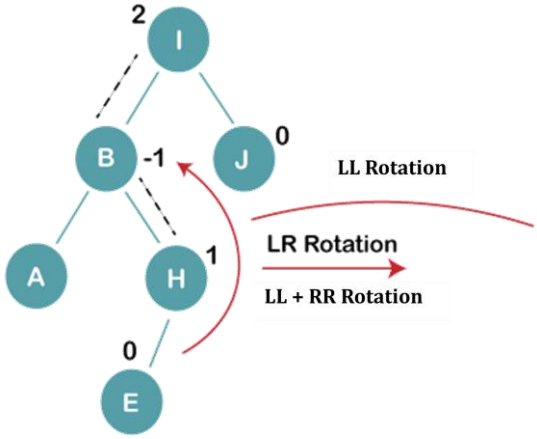


- On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H.
- Since the BST from H is left-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:



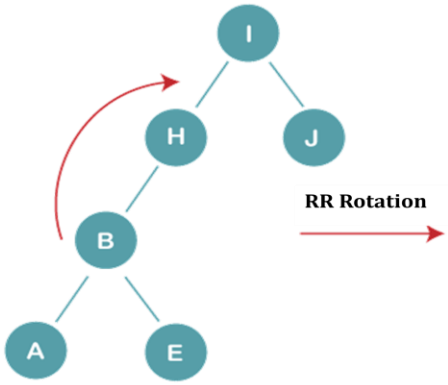
3. Insert E



- On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I.
- we find that it is inserted in the left sub tree of right sub tree of I, we will perform LR Rotation on node I. LR = LL+ RR rotation

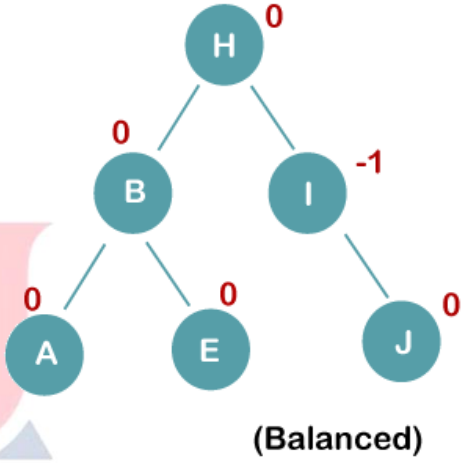
3 a) We first perform LL rotation on node B

The resultant tree after LL rotation is:

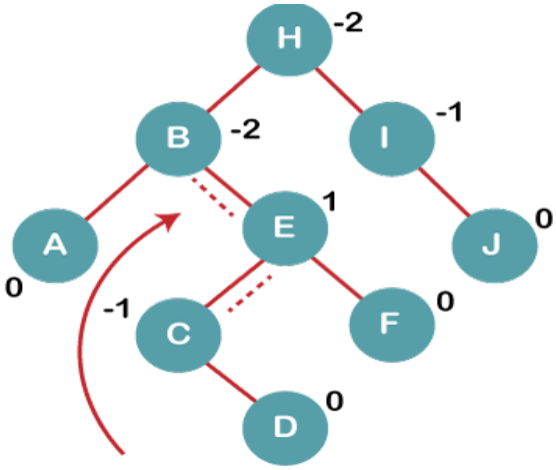


3b) We first perform RR rotation on the node I

The resultant balanced tree after RR rotation is



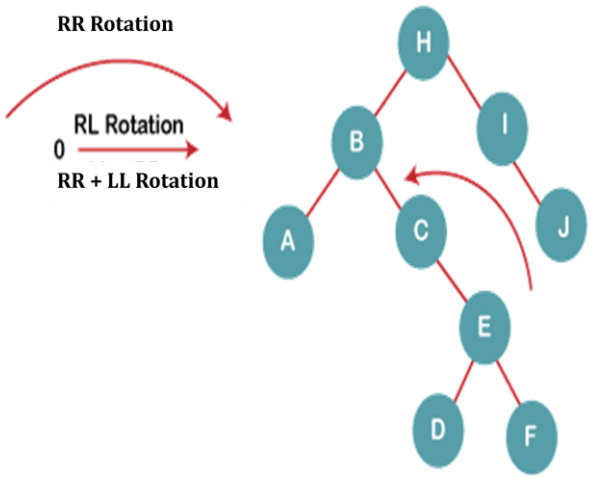
4. Insert C, F, D



- On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B.
- We find that it is inserted in the right sub tree of left sub tree of B, we will perform RL Rotation on node I. RL = RR + LL rotation.

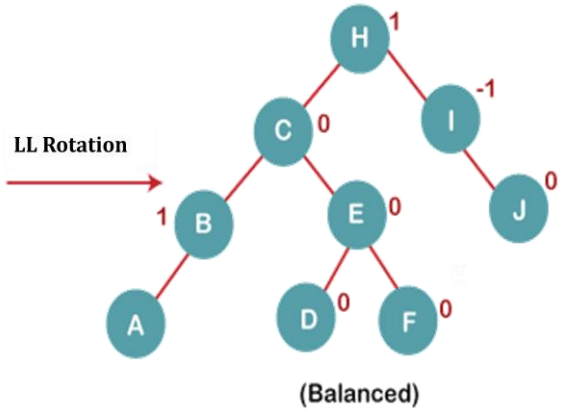
4a) we first perform RR rotation on node E

The resultant tree after RR rotation is:

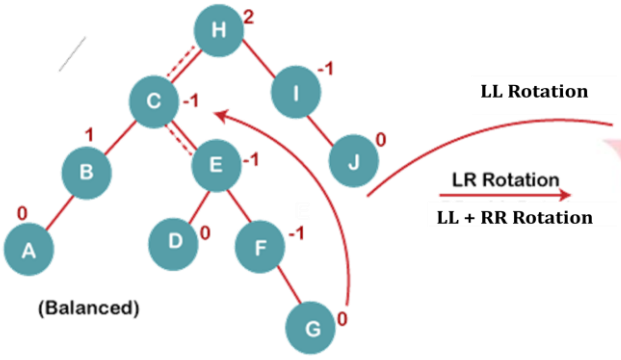


4b) We then perform LL rotation on node B

The resultant balanced tree after LL rotation is:



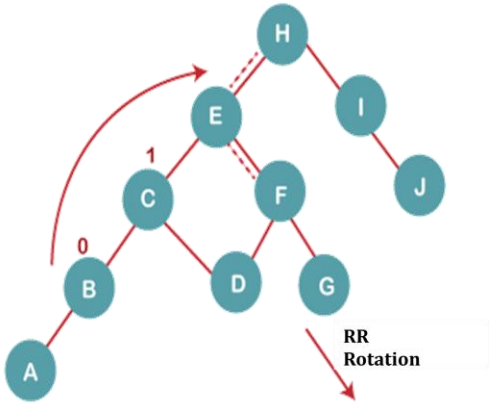
5. Insert G



- On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H.
- We find that it is inserted in the left sub tree of right sub tree of H, we will perform LR Rotation on node I. LR = LL + RR rotation.

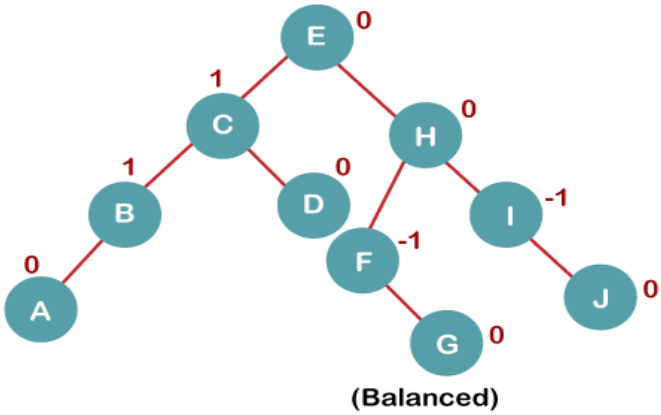
5 a) We first perform LL rotation on node C

The resultant tree after LL rotation is:

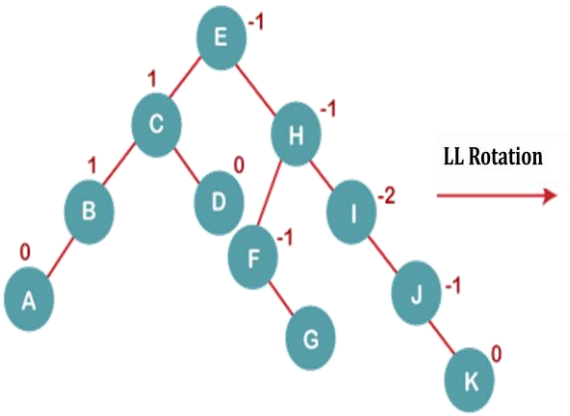


5 b) We then perform RR rotation on node H

The resultant balanced tree after RR rotation is:

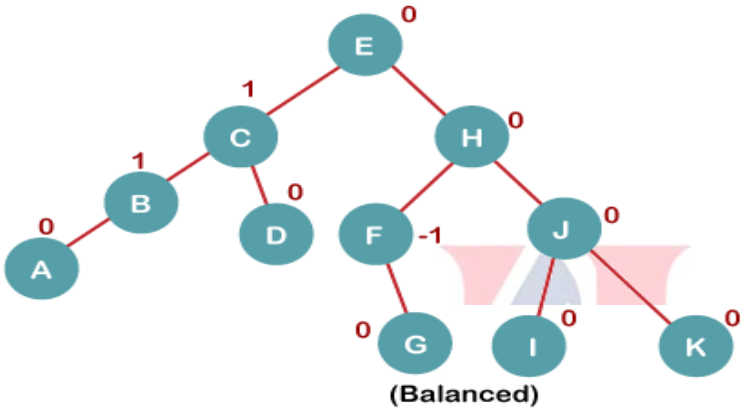


6. Insert K



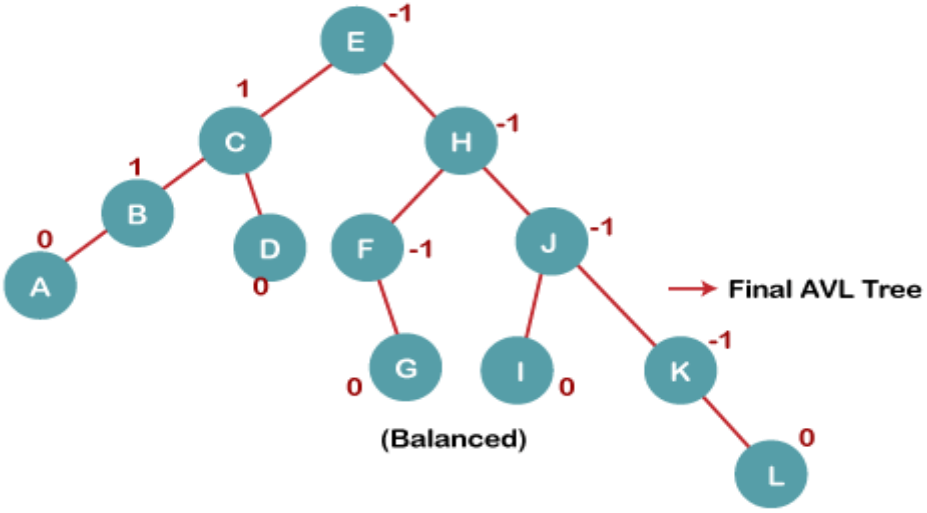
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



Example-4: Construct AVL Tree for the following sequence of numbers-

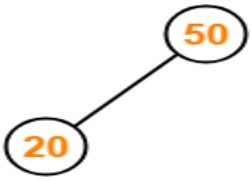
50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

Step-01: Insert 50

Tree is Balanced

Step-02: Insert 20

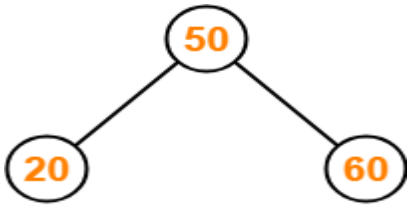
- As 20 < 50, so insert 20 in 50's left sub tree.



Tree is Balanced

Step-03: Insert 60

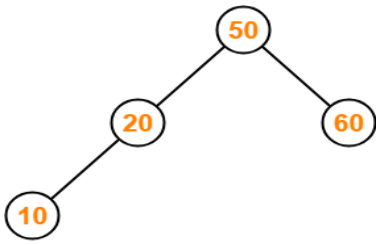
- As 60 > 50, so insert 60 in 50's right sub tree.



Tree is Balanced

Step-04: Insert 10

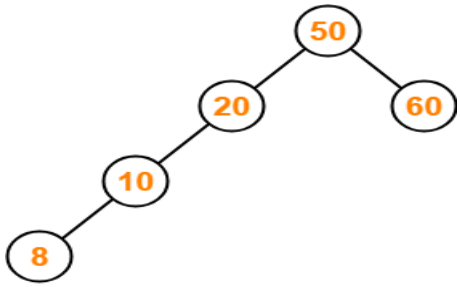
- As 10 < 50, so insert 10 in 50's left sub tree.
- As 10 < 20, so insert 10 in 20's left sub tree.



Tree is Balanced

Step-05: Insert 8

- As 8 < 50, so insert 8 in 50's left sub tree.
- As 8 < 20, so insert 8 in 20's left sub tree.
- As 8 < 10, so insert 8 in 10's left sub tree.

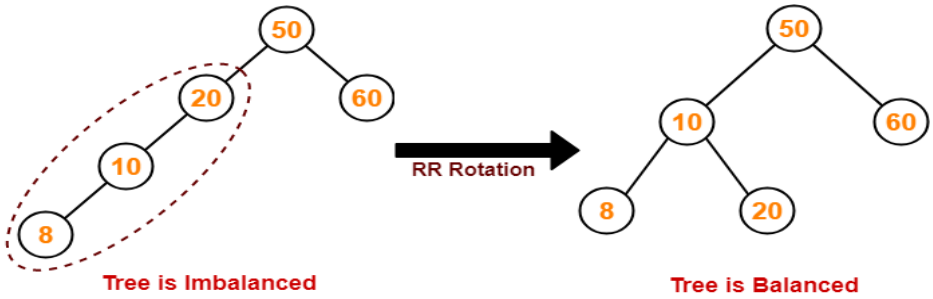


Tree is Imbalanced

To balance the tree,

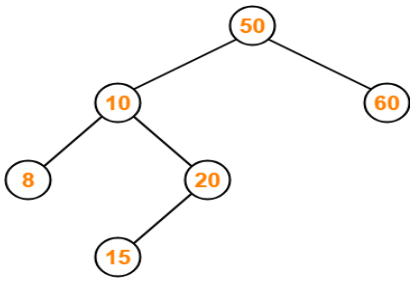
- Find the first imbalanced node on the path from the newly inserted node (node 8) to the root node.
- The first imbalanced node is node 20.
- Now, count three nodes from node 20 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-



Step-06: Insert 15

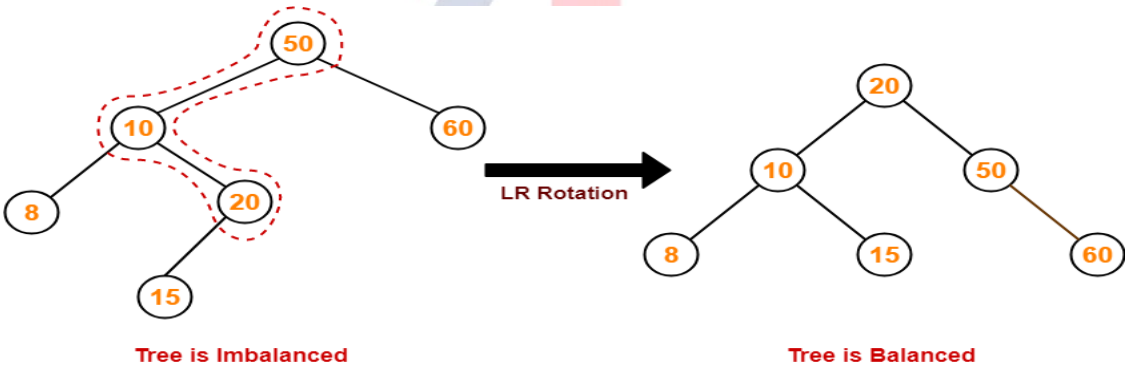
- As 15 < 50, so insert 15 in 50's left sub tree.
- As 15 > 10, so insert 15 in 10's right sub tree.
- As 15 < 20, so insert 15 in 20's left sub tree.



To balance the tree,

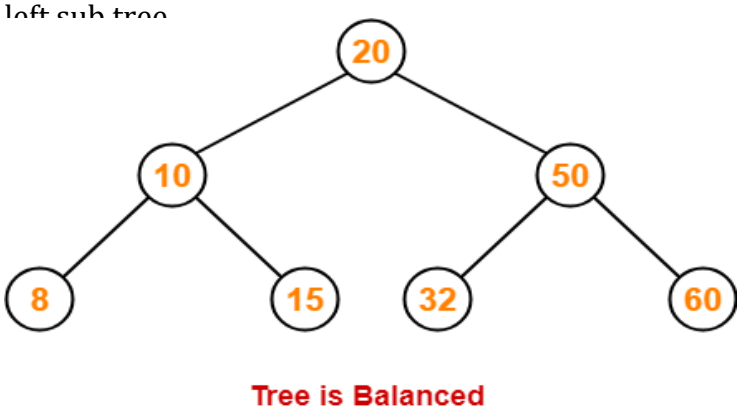
- Find the first imbalanced node on the path from the newly inserted node (node 15) to the root node.
- The first imbalanced node is node 50.
- Now, count three nodes from node 50 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-



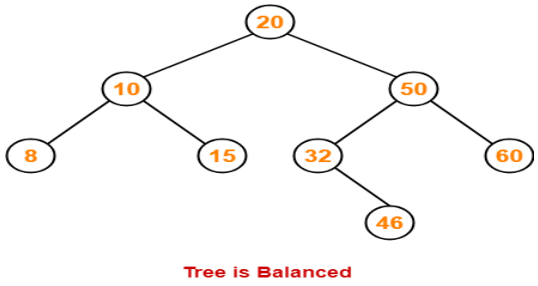
Step-07: Insert 32

- As 32 > 20, so insert 32 in 20's right sub tree.
- As 32 < 50, so insert 32 in 50's left sub tree.



Step-08: Insert 46

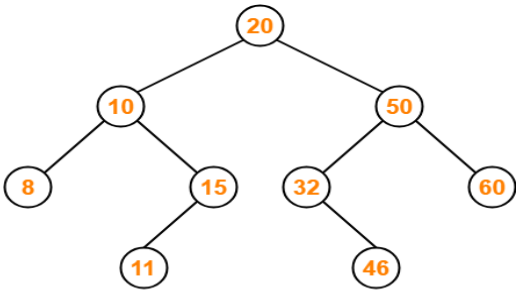
- As 46 > 20, so insert 46 in 20's right sub tree.
- As 46 < 50, so insert 46 in 50's left sub tree.
- As 46 > 32, so insert 46 in 32's right sub tree.



Tree is Balanced

Step-09: Insert 11

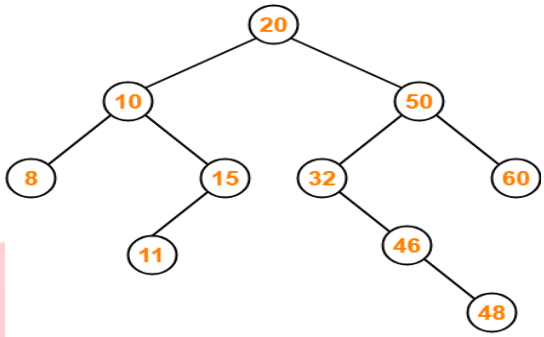
- As 11 < 20, so insert 11 in 20's left sub tree.
- As 11 > 10, so insert 11 in 10's right sub tree.
- As 11 < 15, so insert 11 in 15's left sub tree.



Tree is Balanced

Step-10: Insert 48

- As 48 > 20, so insert 48 in 20's right sub tree.
- As 48 < 50, so insert 48 in 50's left sub tree.
- As 48 > 32, so insert 48 in 32's right sub tree.
- As 48 > 46, so insert 48 in 46's right sub tree.

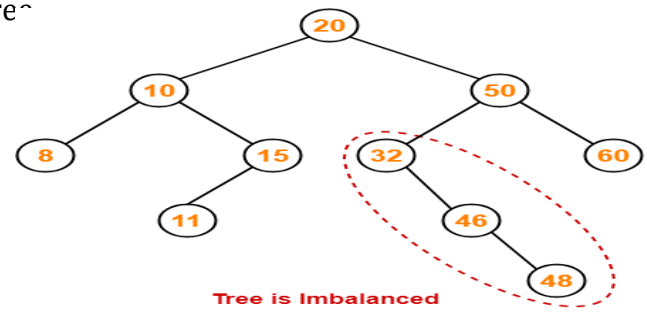


Tree is imbalanced

To balance the tree,

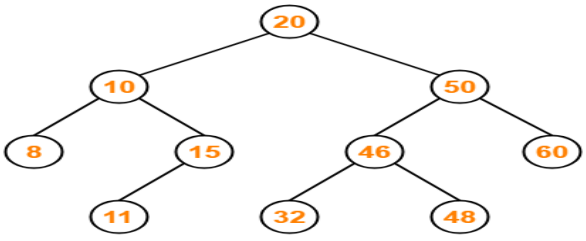
- Find the first imbalanced node on the path from the newly inserted node (node 48) to the root node.
- The first imbalanced node is node 32.
- Now, count three nodes from node 32 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-



Tree is Imbalanced

LL Rotation



Tree is Balanced

This is the final balanced AVL tree after inserting all the given elements.

2. Deletion operation in AVL Tree:

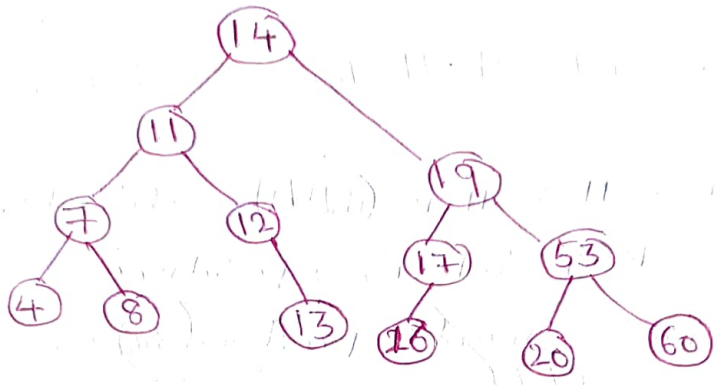
The deletion operation in the AVL tree is the same as the deletion operation in BST. In the AVL tree, the node is always deleted as a leaf node and after the deletion of the node, the balance factor of each node is modified accordingly. **Rotation operations are used to modify the balance factor of each node**. The algorithm steps of deletion operation in an AVL tree are:

1. Locate the node to be deleted
2. If the node does not have any child, then remove the node
3. If the node has one child node, replace the content of the deletion node with the child node and remove the node
4. If the node has two children nodes, find the **in order successor** node 'k' which has no child node and replace the contents of the deletion node with the 'k' followed by removing the node.
5. Update the balance factor of the AVL tree (**Balance factor should be either +1,0,-1**).

AVL-Tree (Deletion operation)

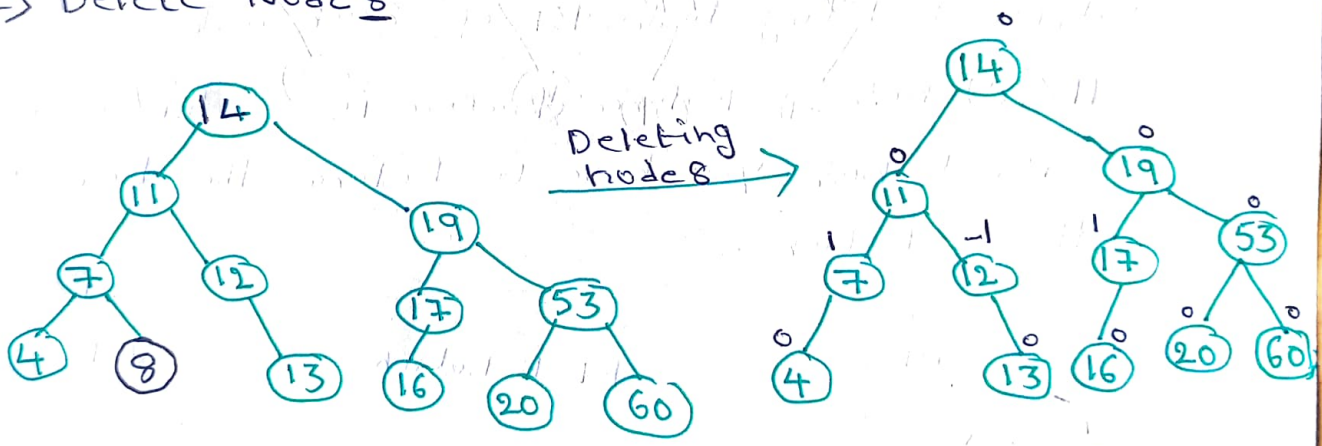
Ex1) Consider the AVL Tree

Delete 8, 7, 11, 14, 17.



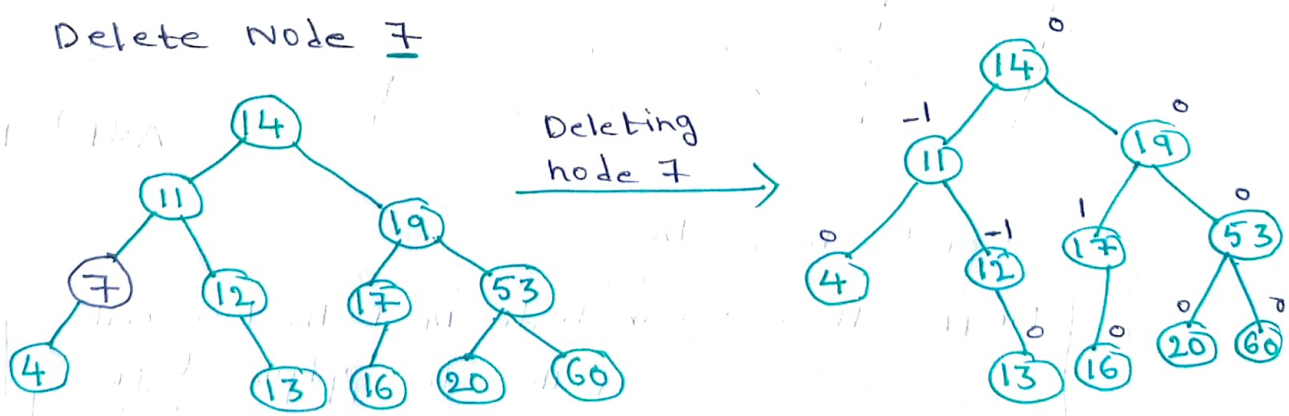
Sol:

1) Delete Node 8



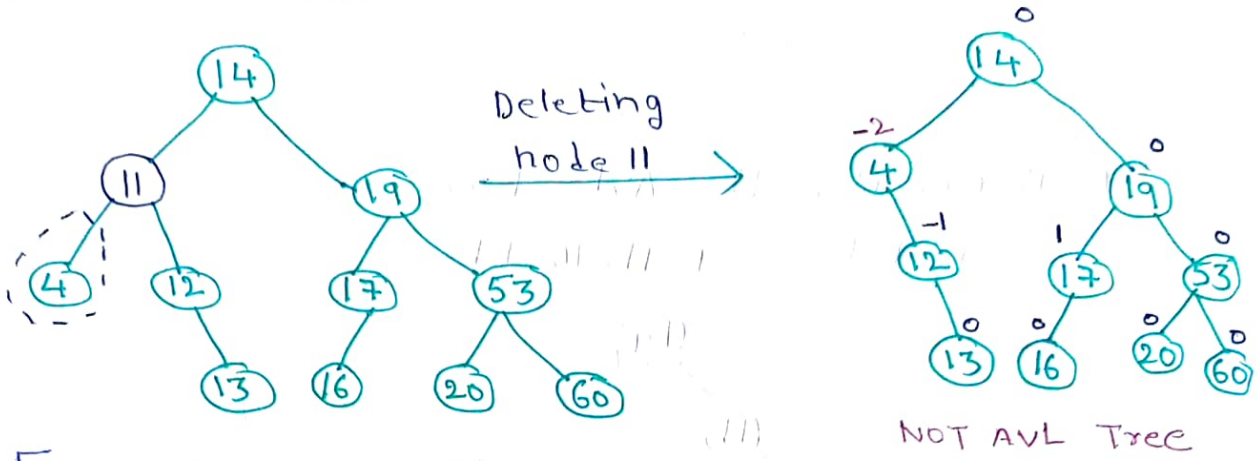
* After Deleting node check the balance factor of each node.

2) Delete Node 7



* After Deleting node check the BF of each node

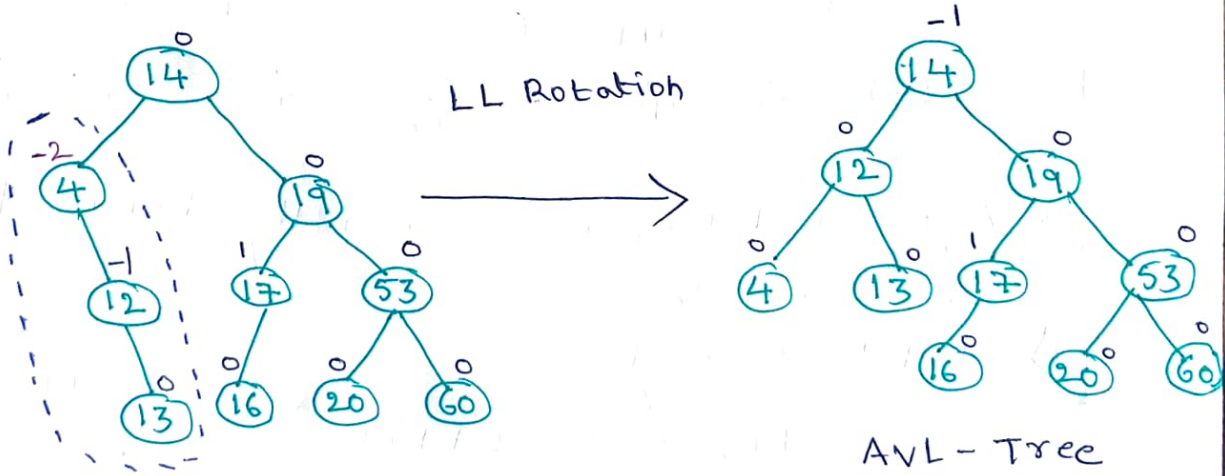
3) Delete node 11



NOTE: Here Node '11' is the two child node, so check the BST deletion operations.
i.e find Inorder predecessor (or) Inorder Successor
Here I choose the predecessor.

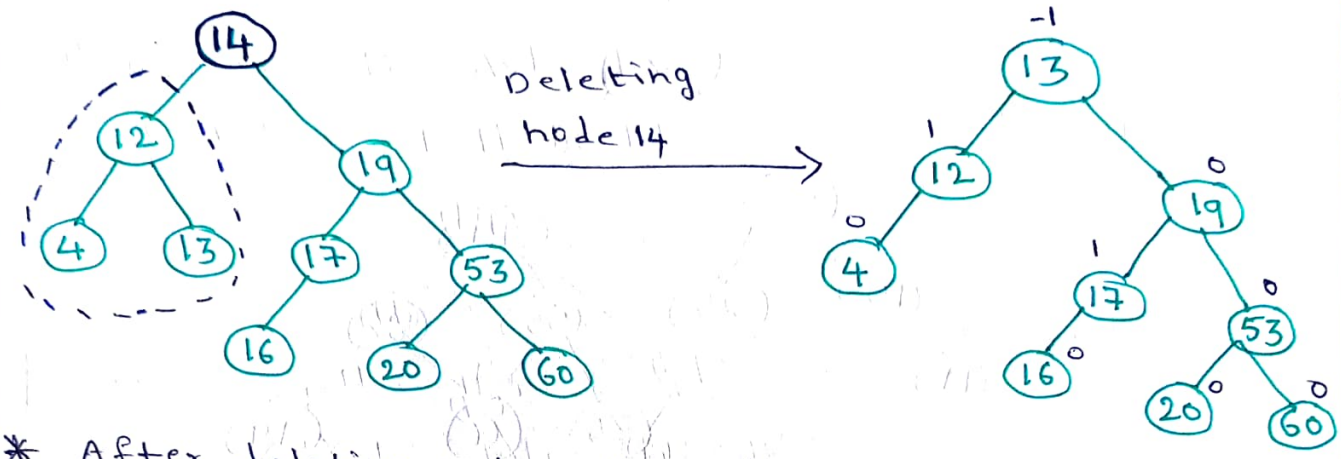
* After Deleting node BF of Each node.

Here node '4' Balance factore is -2. This node is critical node you have to balance the tree as per the rotations.



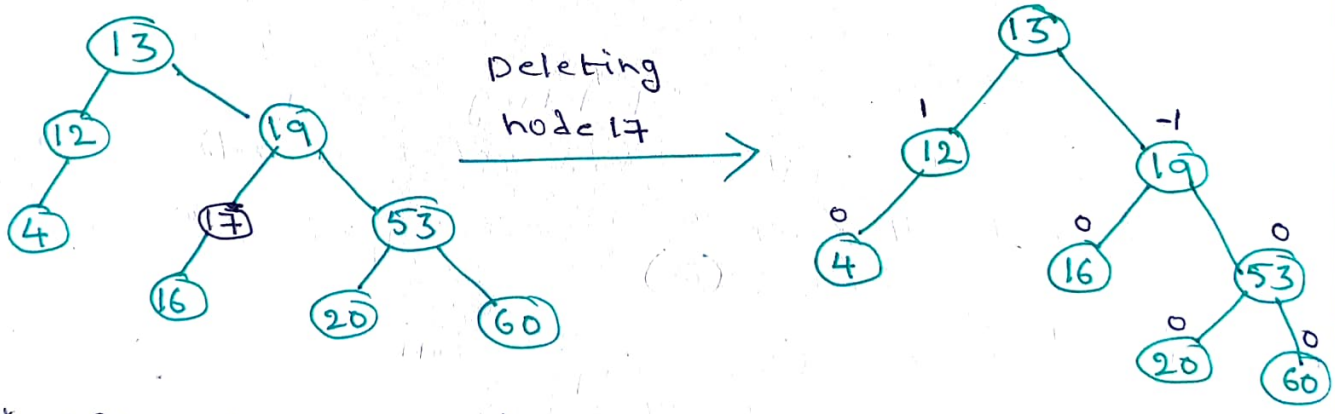
4) Delete node 14.

NOTE: Here again node '14' is the two child node. So follow the above logic
i.e find Inorder predecessor.



* After deleting node check balance factor.

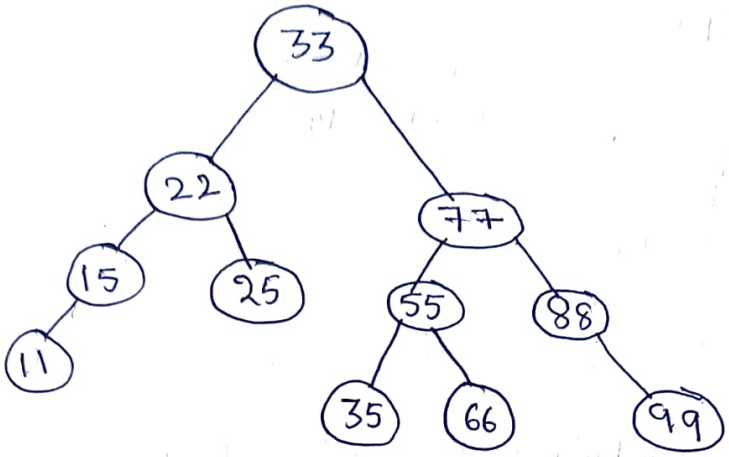
5) Delete node 17



* After deleting node check the B.F of each node

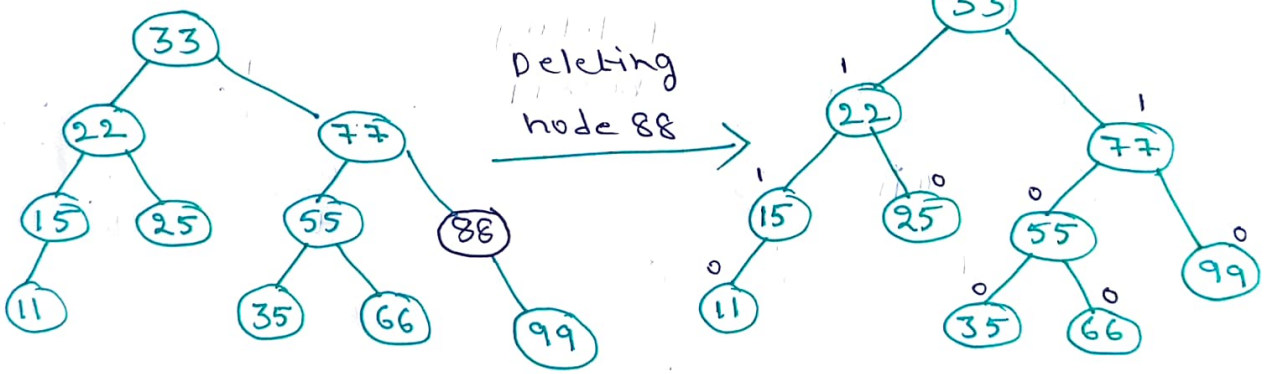
Example 2:) Consider the AVL Tree

Delete 88, 99, 22, 33, 11, 15, 35, 66, 99, 77, 55.



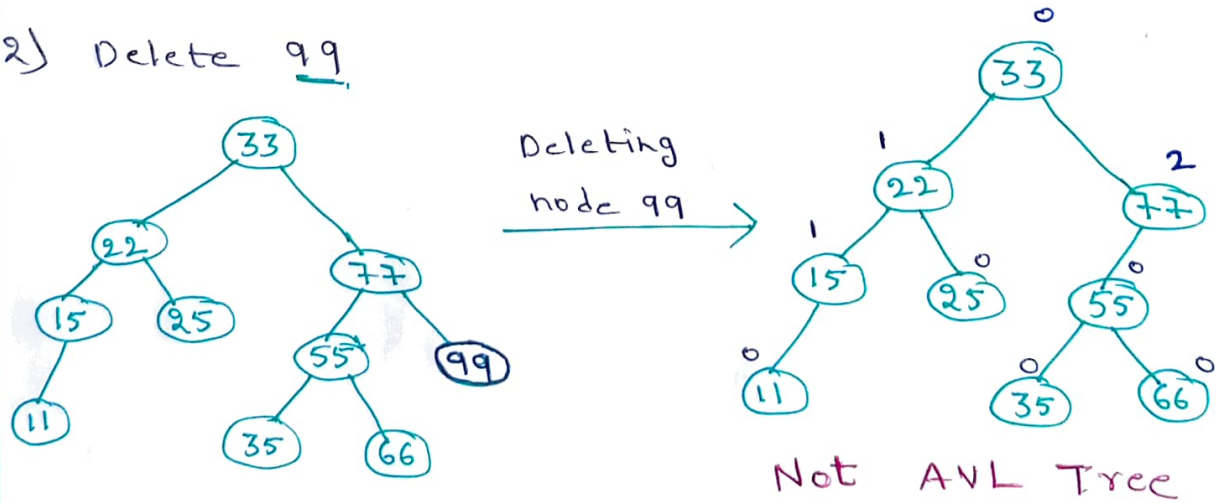
Sol:

1) Delete 88



* After Deleting a node check the BF of Each node.

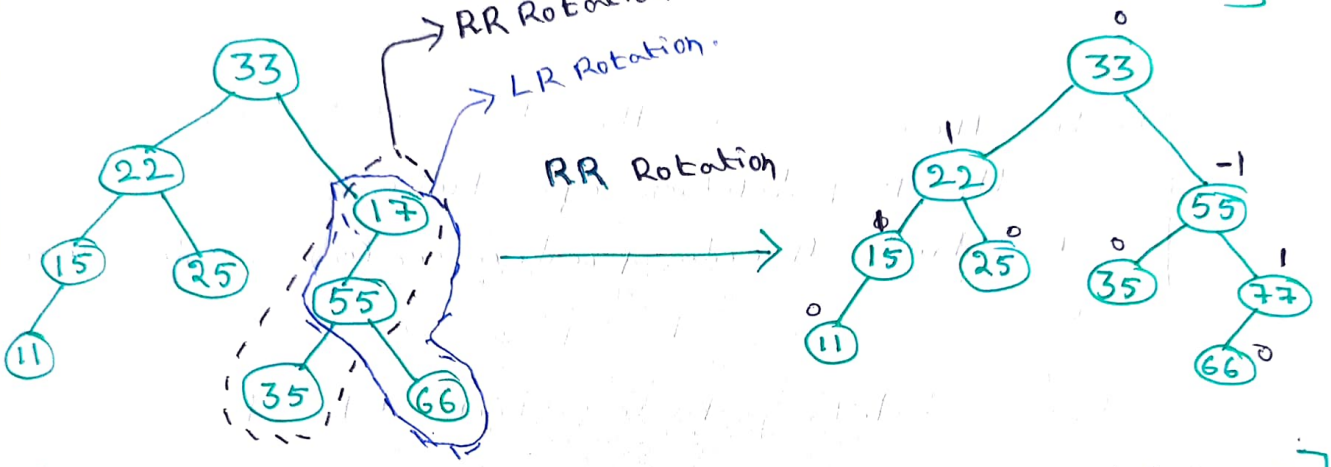
2) Delete 99



Not AVL Tree

* After Deleting a node check BF of Each node

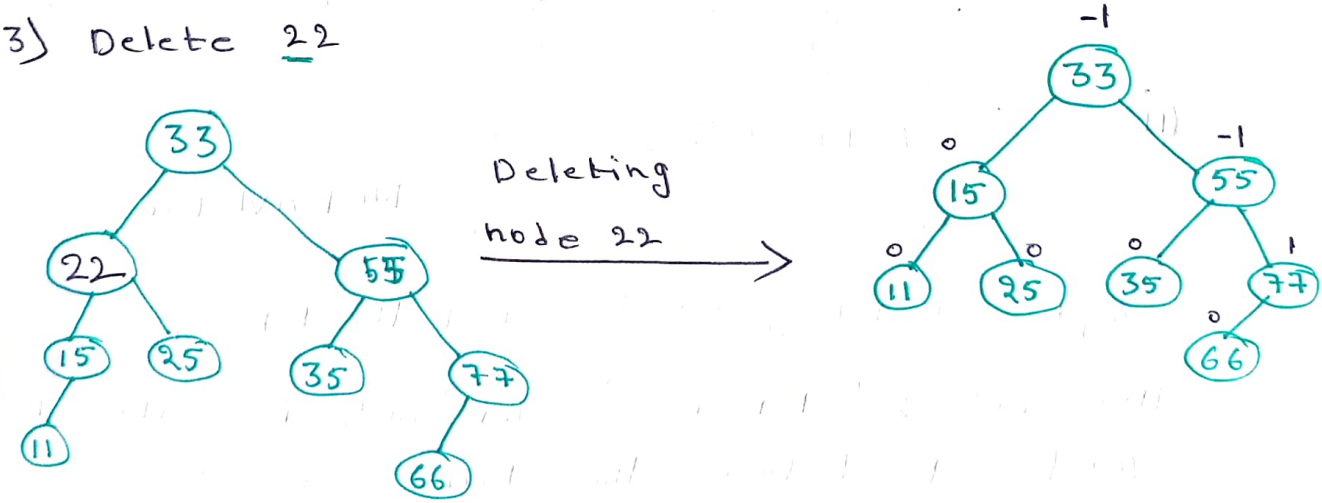
NOTE: Here node 77 Balance factor is '2'. This node is critical node, you have to balance the tree as per rotations



Here - You can follow either RR Rotation (or) LR Rotation

* Here I am following RR Rotation. After that check the Balance Factor.

3) Delete 22

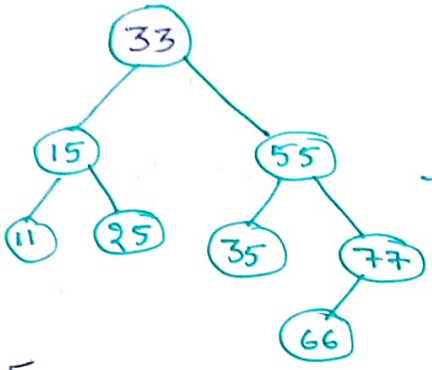


NOTE: Here Node '22' is the two child node, so check the BST Deletion operations.

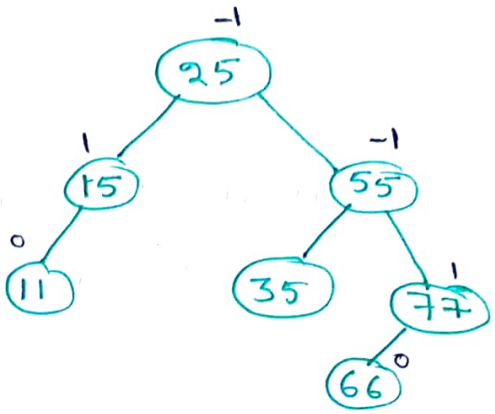
i.e find In order predecessor (or) In order Successor

Here I choose In order predecessor. After that check the BF of Each node.

4) Delete 33



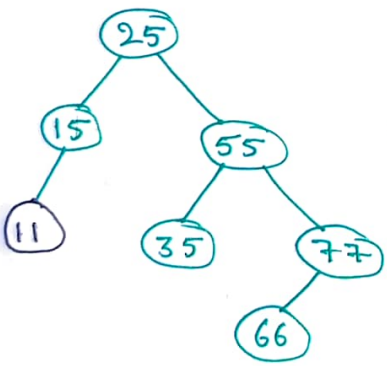
Deleting node 33 →



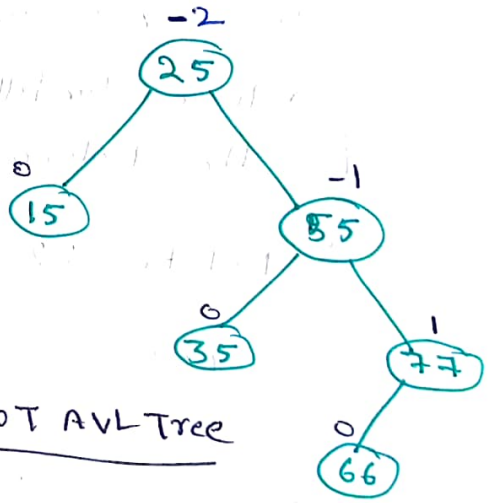
[NOTE: Here 33 is the two child node. So check the BST deletion operations.
i.e find Inorder predecessor (or) Inorder Successor
Here I choose the predecessor.]

* After deleting node, check BF of Each node.

5) Delete 11



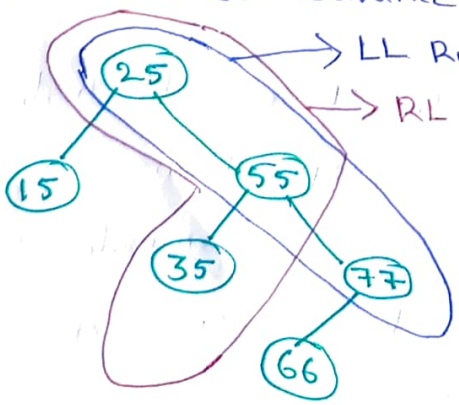
Deleting node 11 →



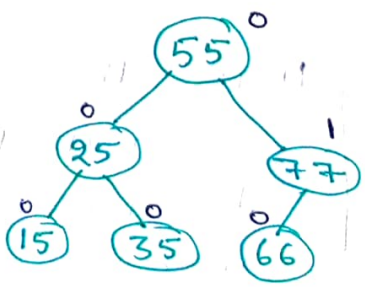
NOT AVL Tree

* After Deleting a node, check the BF of Each node

[Here node '25' BF is '2'. This node is critical node.
You have to balance the tree as per the rotations.]

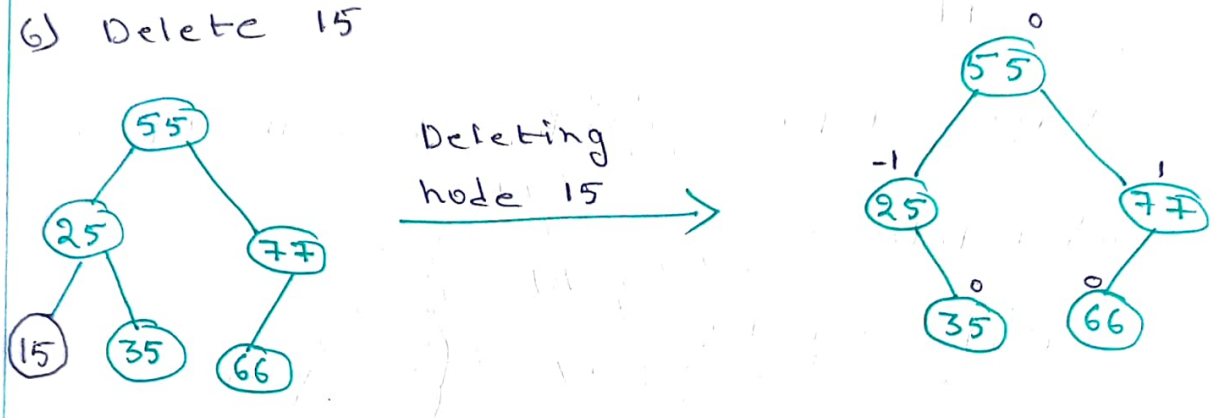


LL Rotation →



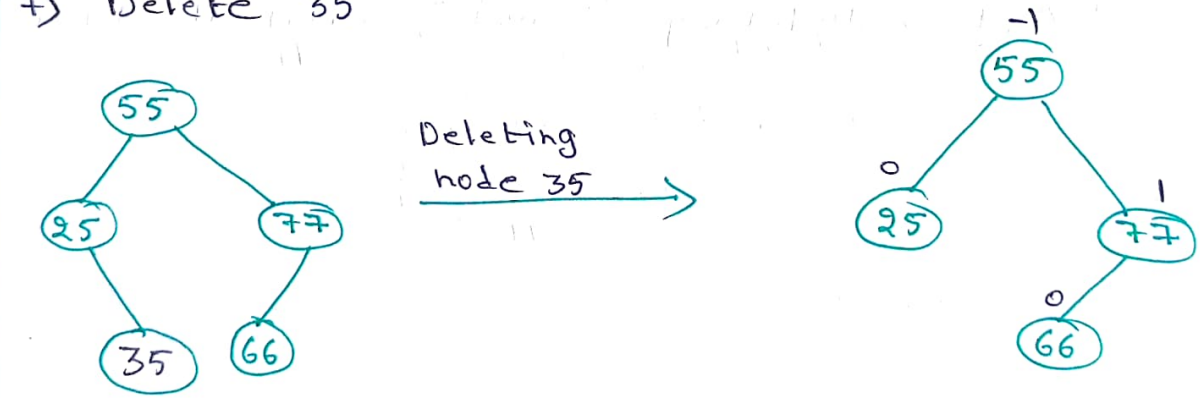
* After Rotation check the BF of Each node.

6) Delete 15



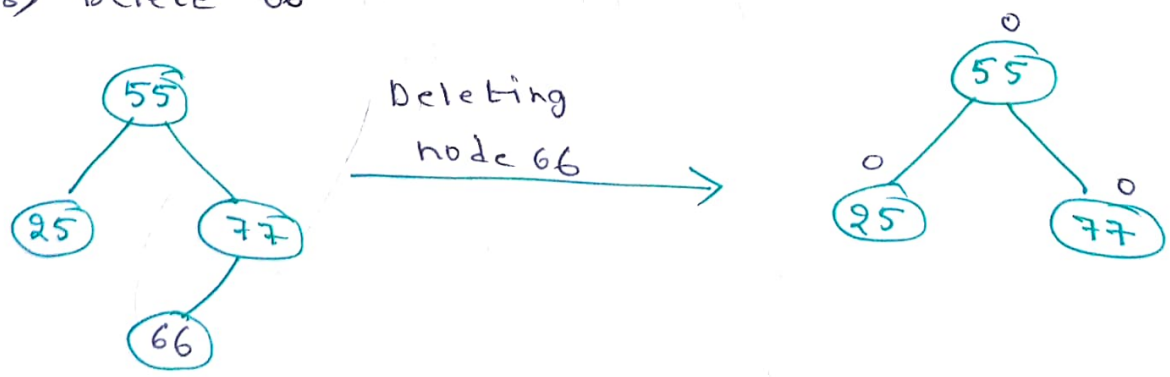
* After Deleting a node ~~BF of~~, check BF of node.

7) Delete 35



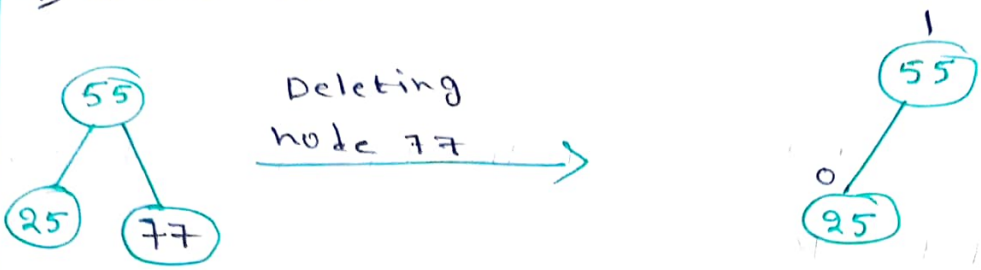
* After Deleting a node, check BF of Each node

8) Delete 66



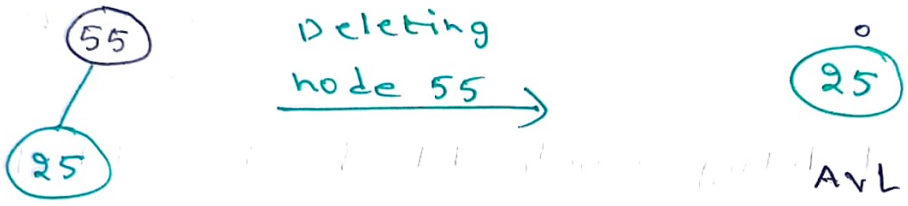
* After Deleting a node, check BF of Each node.

9) Delete 77



* After Deleting a node, check the BF of Each node.

10) Delete 55



AVL Tree.

* After Deleting a node, check the BF of Eachnode

3. 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 sub tree.
- **Step 6 -** If search element is larger, then continue the search process in right sub tree.
- **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.

B -Tree Data structure

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.

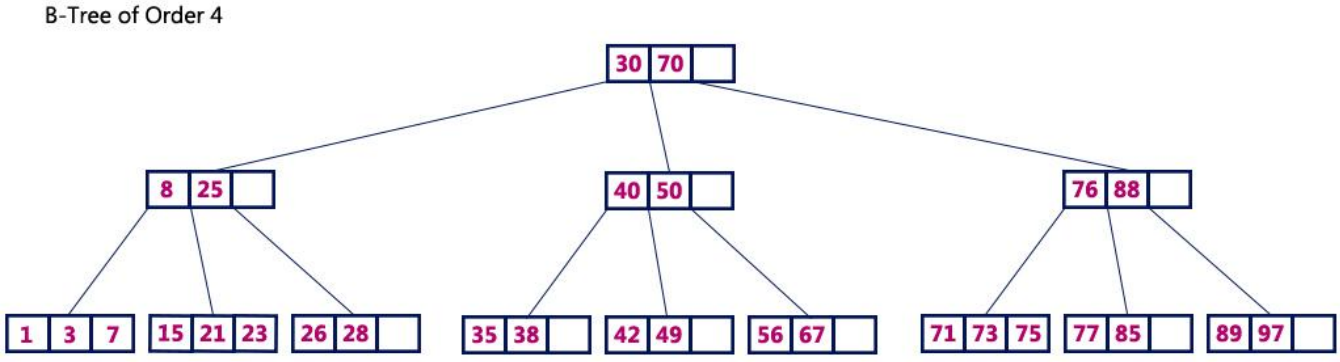
Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

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:



Operations on a B-Tree:

The following operations are performed on a B-Tree...

1. **Search**
2. **Insertion**
3. **Deletion**

1. 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.

2. 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: 01- Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

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

insert(1)

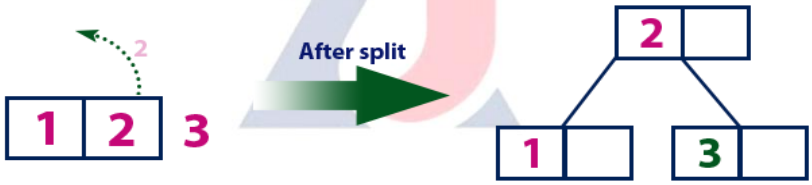
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

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.

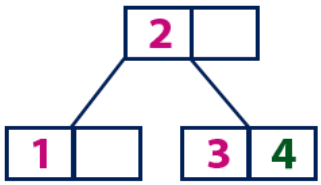
insert(3)

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.



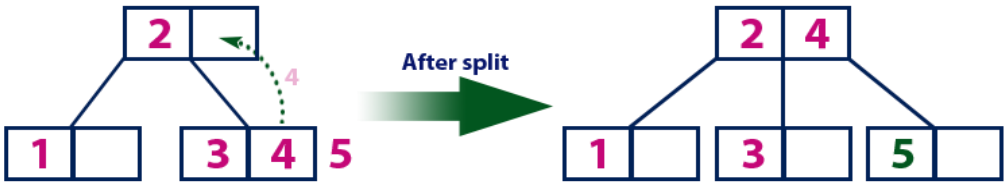
insert(4)

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.



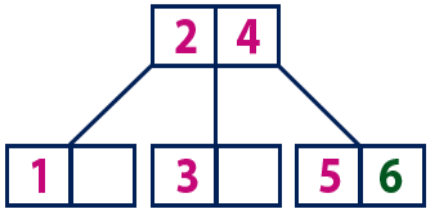
insert(5)

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.



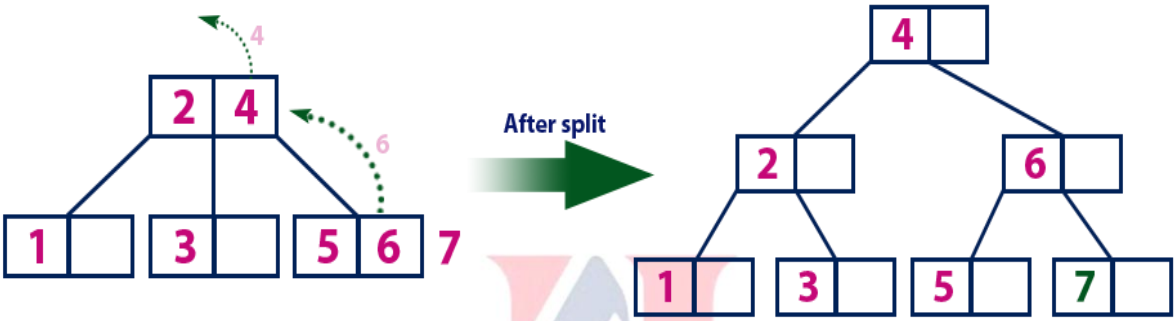
insert(6)

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.



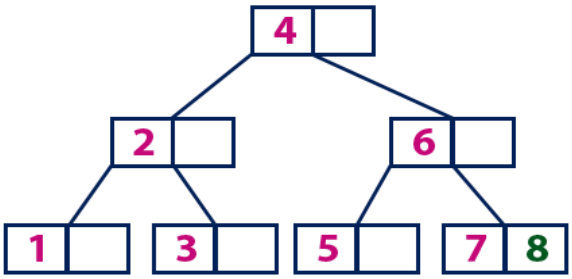
insert(7)

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.



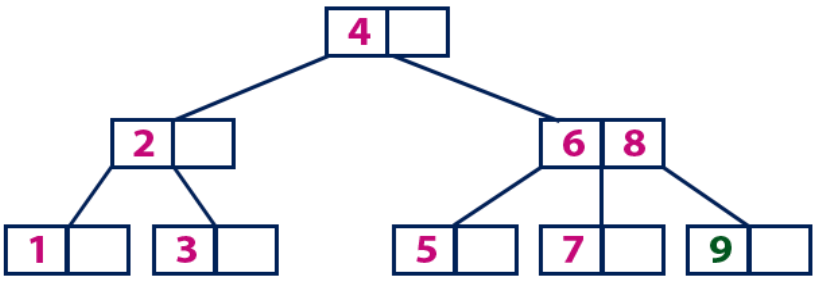
insert(8)

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.



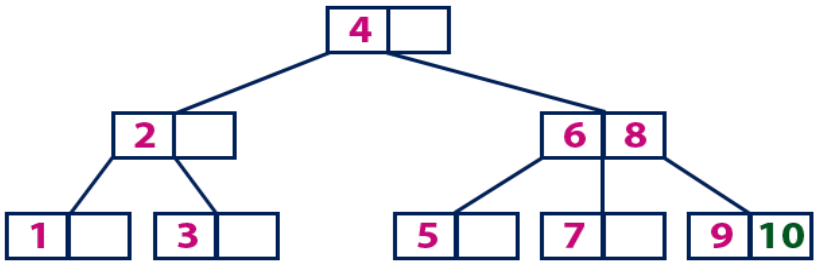
insert(9)

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.



Example-2:

The insertion operation for a B Tree is done similar to the Binary Search Tree but the elements are inserted into the same node until the maximum keys are reached. The insertion is done using the following procedure –

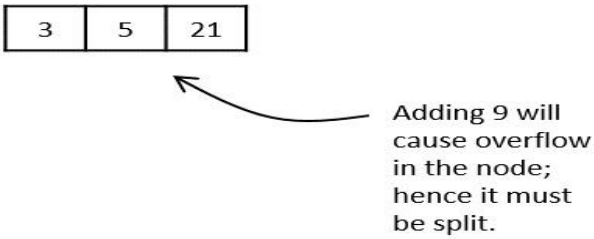
Step 1 – Calculate the maximum (m−1) and, minimum ([m/2[−1) number of keys a node can hold, where m is denoted by the order of the B Tree.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

- Order (m) = 4
- Maximum Keys (m − 1) = 3
- Minimum Keys ($\lceil \frac{m}{2} \rceil$) − 1 = 1
- Maximum Children = 4
- Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2

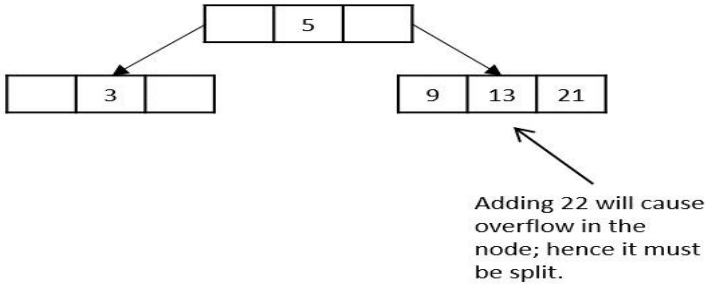
Step 2 – The data is inserted into the tree using the binary search insertion and once the keys reach the maximum number, the node is split into half and the median key becomes the internal node while the left and right keys become its children

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

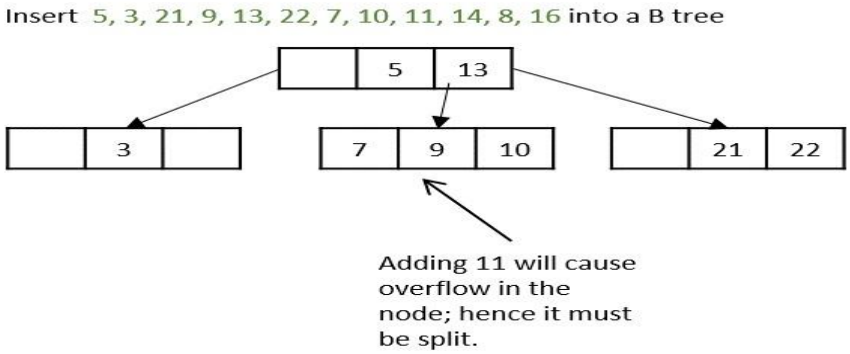


Step 3 – All the leaf nodes must be on the same level.

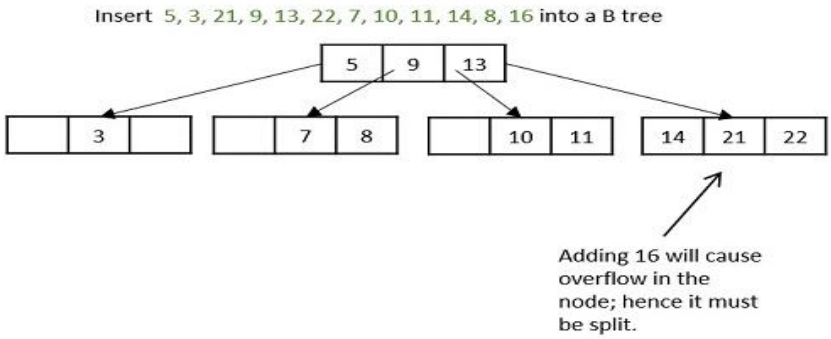
Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree



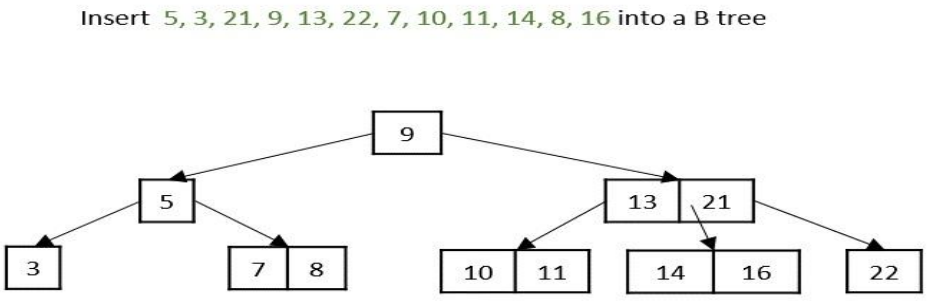
The keys, 5, 3, 21, 9, 13 are all added into the node according to the binary search property but if we add the key 22, it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued.



Another hiccup occurs during the insertion of 11, so the node is split and median is shifted to the parent.



- While inserting 16, even if the node is split in two parts, the parent node also overflows as it reached the maximum keys. Hence, the parent node is split first and the median key becomes the root. Then, the leaf node is split in half the median of leaf node is shifted to its parent.



The final B tree after inserting all the elements is achieved

Applications of B-Trees:

- It is used in large databases to access data stored on the disk
- Searching for data in a data set can be achieved in significantly less time using the B-Tree
- With the indexing feature, multilevel indexing can be achieved.
- Most of the servers also use the B-tree approach.
- B-Trees are used in CAD systems to organize and search geometric data.

Advantages of B-Trees:

- B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- B-Trees are self-balancing.
- High-concurrency and high-throughput.
- Efficient storage utilization.

Disadvantages of B-Trees:

- B-Trees are based on disk-based data structures and can have a high disk usage.
- Not the best for all cases.
- Slow in comparison to other data structures

Introduction to Red-Black Trees

- A red-black tree is a kind of **self-balancing binary search tree** where each node has an **extra bit,** and that bit is often interpreted as the colour (red or black).
- These colours are used to ensure that the tree remains balanced during insertions and deletions.
- Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree.
- This tree was invented in 1972 by **Rudolf Bayer.**

Red - Black Tree is another variant of Binary Search Tree in which every node is coloured either RED or BLACK. We can define a Red Black Tree as follows...

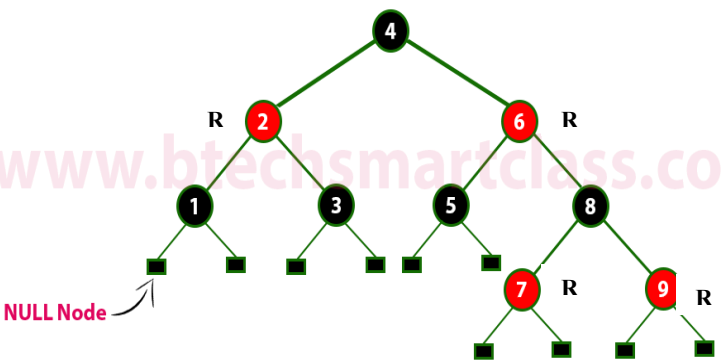
Red Black Tree is a Binary Search Tree in which every node is coloured either RED or BLACK.

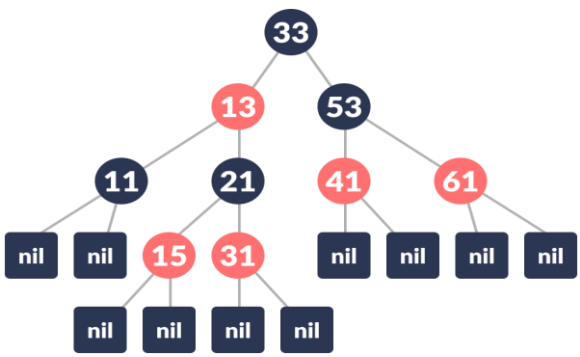
In Red Black Tree, the colour of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

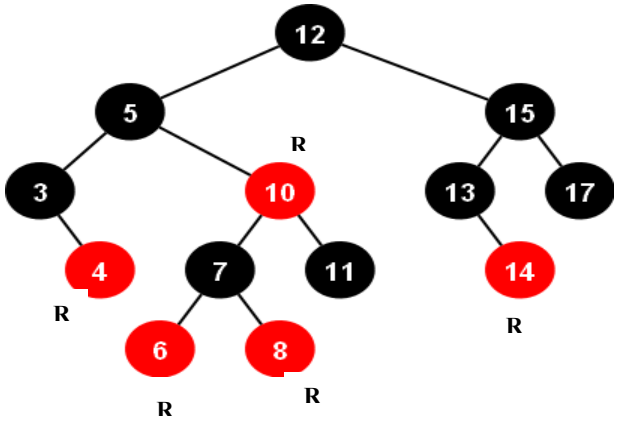
Properties of Red Black Tree:

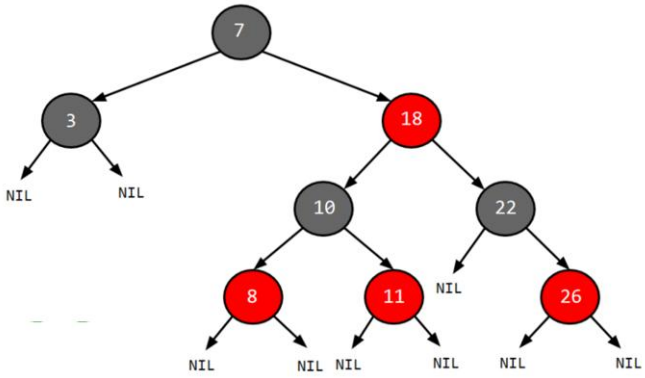
- **Property -1:** Red - Black Tree must be a Binary Search Tree.
- **Property -2:** The ROOT node must be coloured BLACK.
- **Property -3:** The children of Red coloured node must be coloured BLACK. (**There should not be two consecutive RED nodes)**.
- **Property -4:** In all the paths of the tree,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

Examples:









The above tree is a Red-Black tree where every node is satisfying all the properties of Red-Black Tree.

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

Red Black Tree – Insertion:

In a Red-Black Tree, every new node must be inserted with the colour RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a colour property. After every insertion operation, we need to check all the properties of Red-Black Tree.

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

- **Step -1 -** Check whether tree is Empty. If tree is Empty then create / insert the **newNode** as Root node with colour **Black** and exit from the operation.
- **Step -2 -** If tree is not empty then insert the newNode as leaf node with colour **Red.**
- **Step- 3 -** If the parent of newNode is **Black then exit** from the operation.
- **Step- 5 -** If the parent of newNode is **Red then check** the colour of parent node's **sibling.**
    a) If it is coloured **Black** or **NULL** then make suitable Rotation (**newnode, parent, and Grand parent**) and Recolor it.
    b) If it is coloured **Red** then performs Recolor (parent and sibling) **and** also checks if parent's parent of newnode is not **ROOT** then recolor it and recheck**.** Repeat the same until tree becomes Red Black Tree.

Example-01:

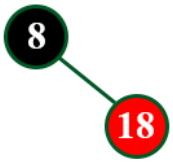
Create a RED BLACK Tree by inserting following sequence of number
8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

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

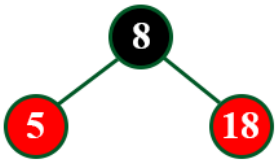
insert (18)

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



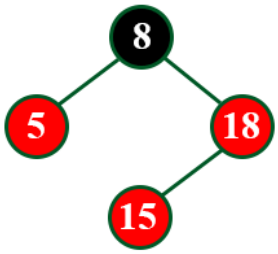
insert (5)

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



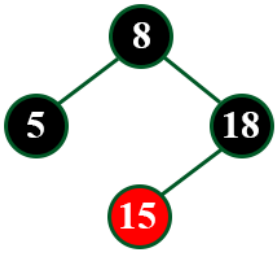
insert (15)

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.

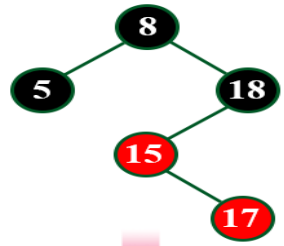
After RECOLOR



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

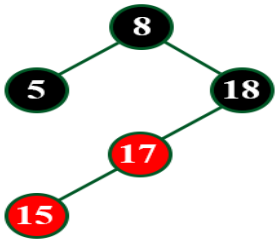
insert (17)

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

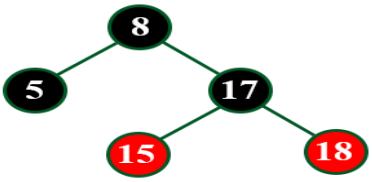


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

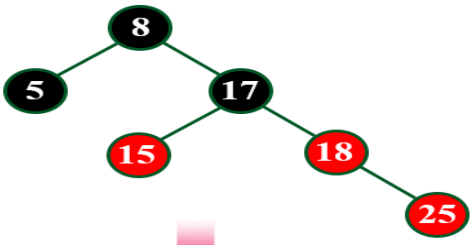


After Right Rotation & Recolor



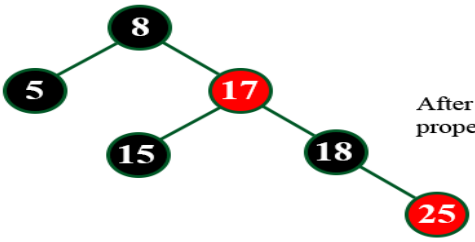
insert (25)

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.

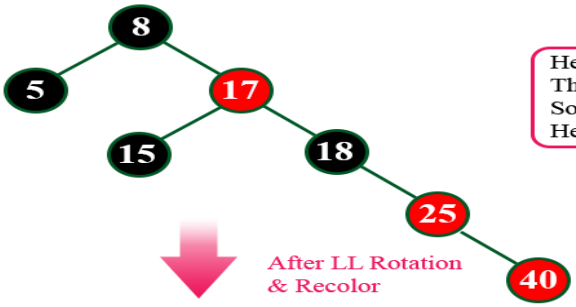
After Recolor



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

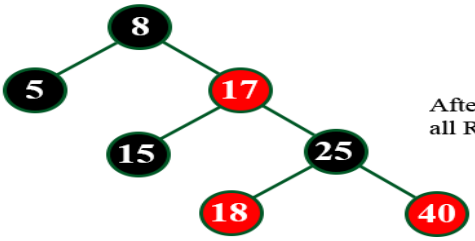
insert (40)

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.

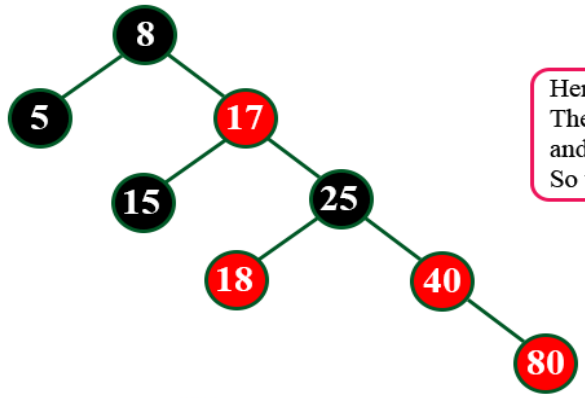
After LL Rotation
& Recolor



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

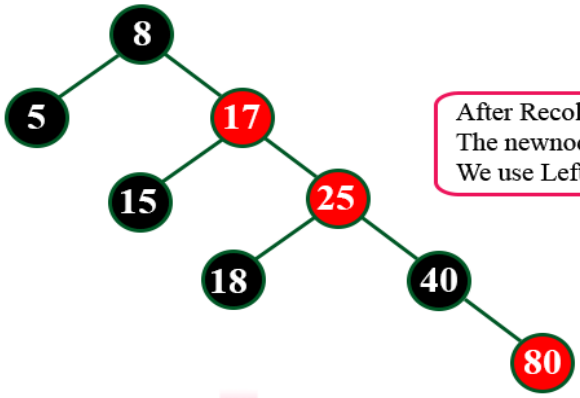
insert (80)

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



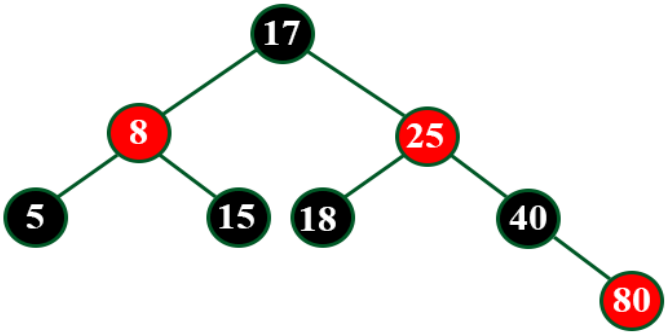
Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

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.

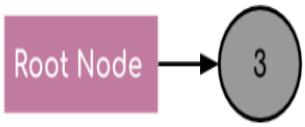
After Left Rotation
& Recolor



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

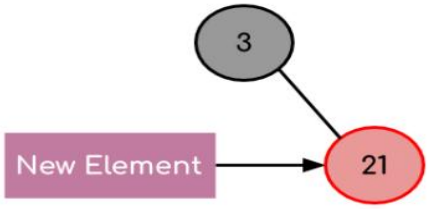
Example-02: Creating a red-black tree with elements 3, 21, 32 and 15 in an empty tree.

Step 1: Inserting element 3 inside the tree.



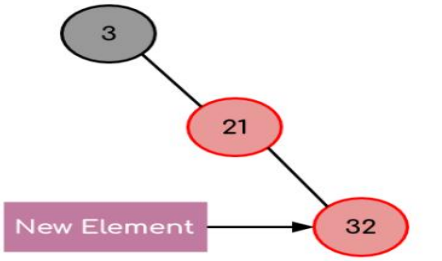
When the first element is inserted it is inserted as a root node and as root node has black colour so it acquires the colour black.

Step 2: Inserting element 21 inside the tree.

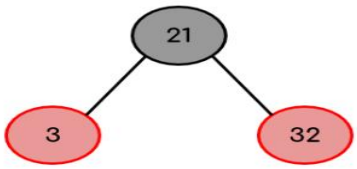


The new element is always inserted with a **red colour** and as 21 > 3 so it becomes the part of the right sub tree of the root node.

Step 3: Inserting element 32 inside the tree.

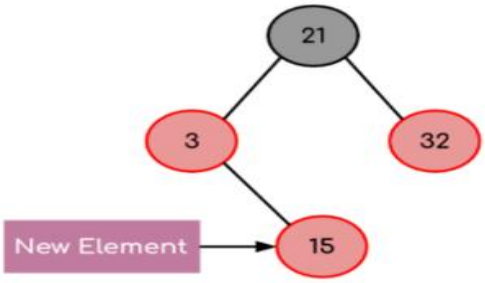


Here we see that as two red node are not possible and also we can see the conditions of RR rotation so it will follow RR rotation and recolouring to balance the tree.

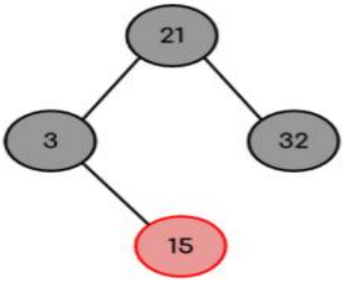


Here there are two consecutive Red nodes (21 & 32). The newnode's parent sibling is NULL. so we need a rotation & Recolor. Here we Use RR Rotation and Recheck

Step 4: Inserting element 15 inside the tree.

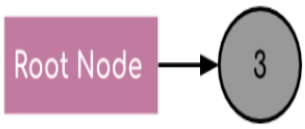


Here we see that as two red node are not possible and also we perform recolouring in the tree which result in red coloured root node. So we simply colour it to black.



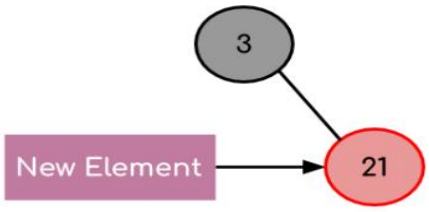
Example-02: Creating a red-black tree with elements 3, 21, 32 and 15 in an empty tree.

Step 1: Inserting element 3 inside the tree.



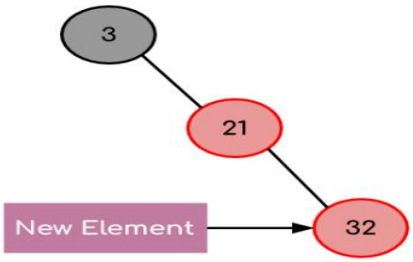
When the first element is inserted it is inserted as a root node and as root node has black colour so it acquires the colour black.

Step 2: Inserting element 21 inside the tree.

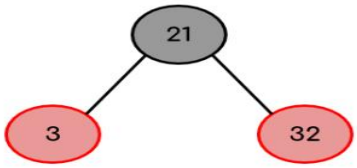


The new element is always inserted with a **red colour** and as 21 > 3 so it becomes the part of the right sub tree of the root node.

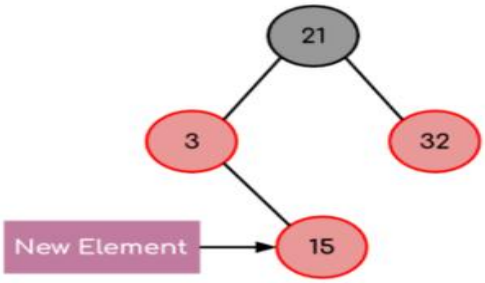
Step 3: Inserting element 32 inside the tree.



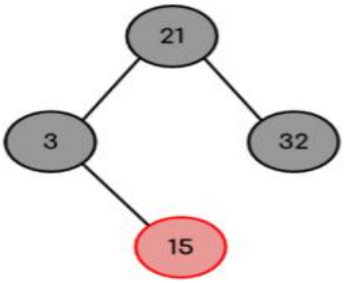
Here we see that as two red node are not possible and also we can see the conditions of RR rotation so it will follow RR rotation and recolouring to balance the tree.



Step 4: Inserting element 15 inside the tree.

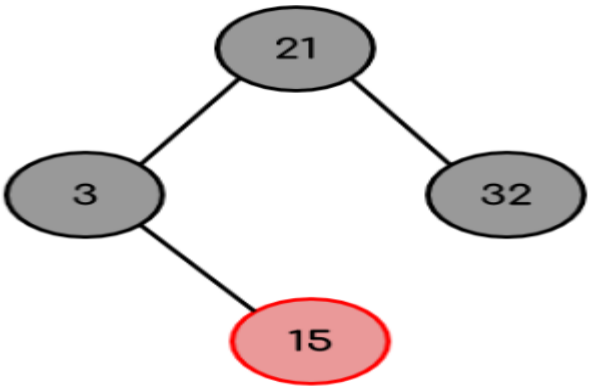


Here we see that as two red node are not possible and also we perform recolouring in the tree which result in red coloured root node. So we simply colour it to black.



Here there are two consecutive Red nodes (21 & 32). The newnode's parent sibling is NULL. so we need a rotation & Recolor. Here we Use RR Rotation and Recheck

Final Tree Structure:



The final tree will look like this

Red - Black Tree (insertion)

Ex3) Create a Red-Black Tree by inserting following sequence of numbers.

10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70

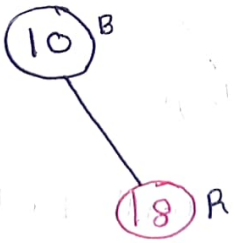
Sol:

insert(10);

Tree is Empty, so insert newnode as root node with Black color.

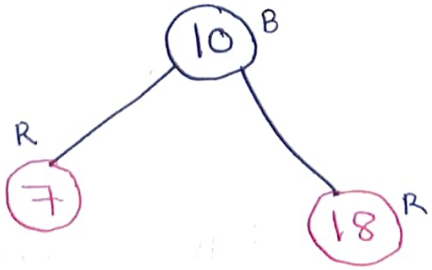
insert(18):

Tree is not Empty, so insert newnode with Red color.



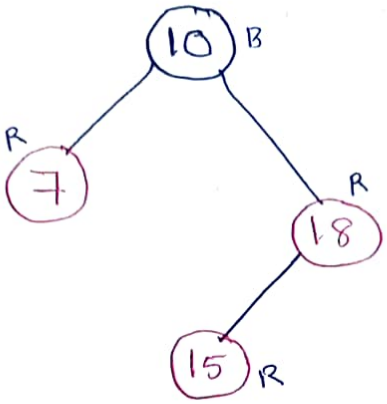
insert(7):

Tree is not Empty, so insert newnode with Red color.



insert (16) :

Tree is not Empty, so insert newnode with Red color.

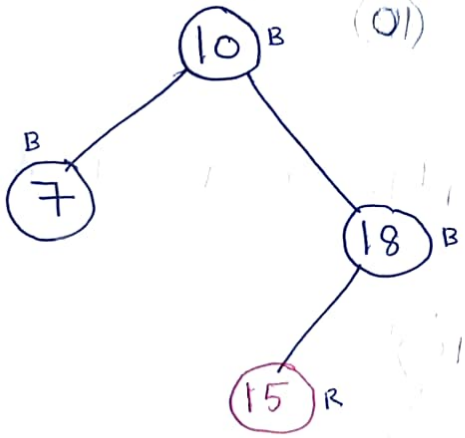


Here, There are two Consecutive Red nodes (15 & 18).

The newnode's parent sibling color is Red

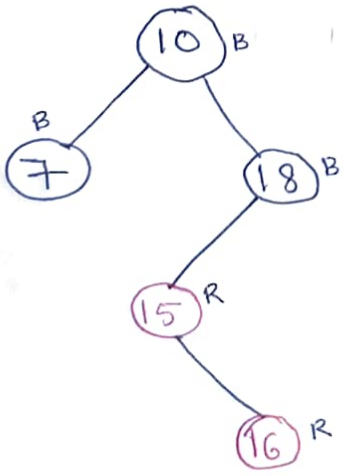
So. use REcolor.

Here Parent's Parent is root.

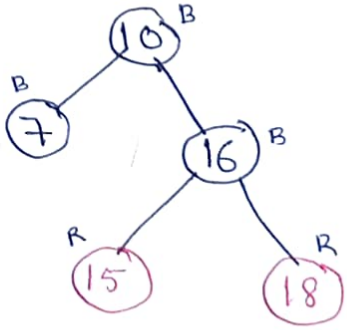


(or)

insert (16) : new node with Red color



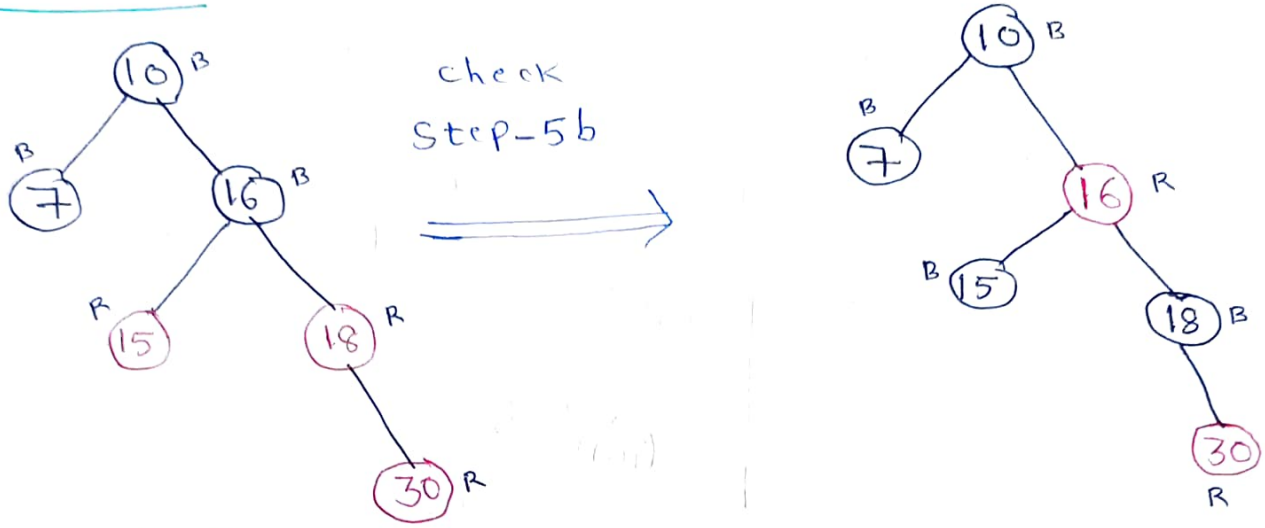
LR Rotation.



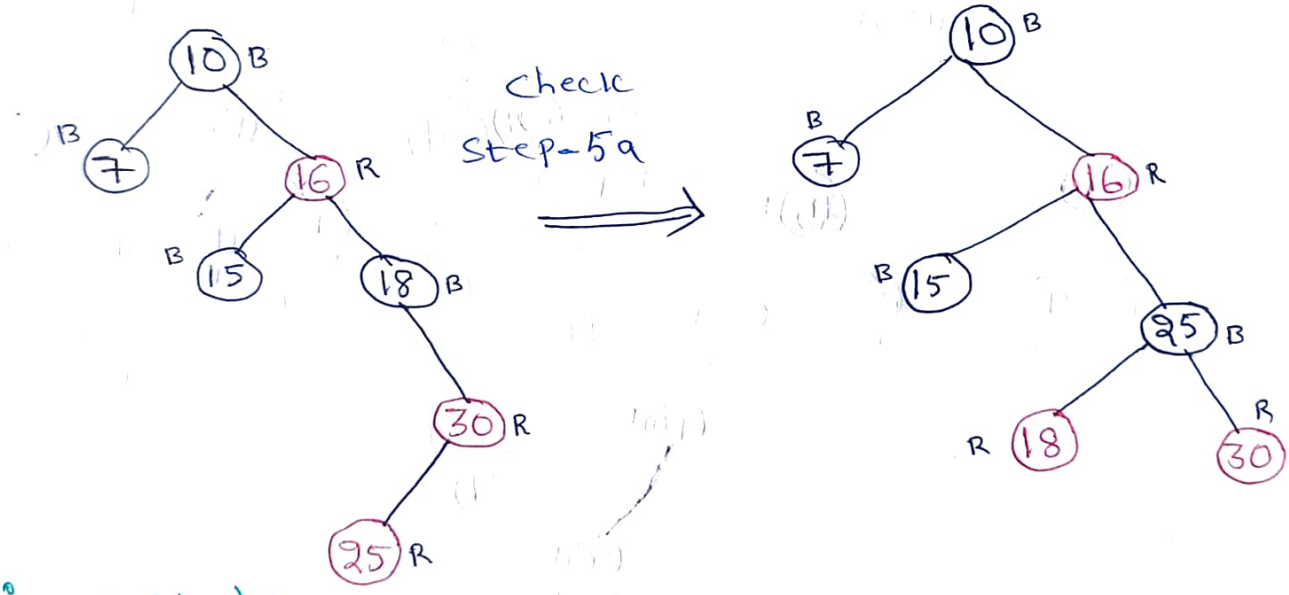
Here two consecutive Red Nodes

Here newnode parent's sibling is NULL, so Perform rotation & REColor

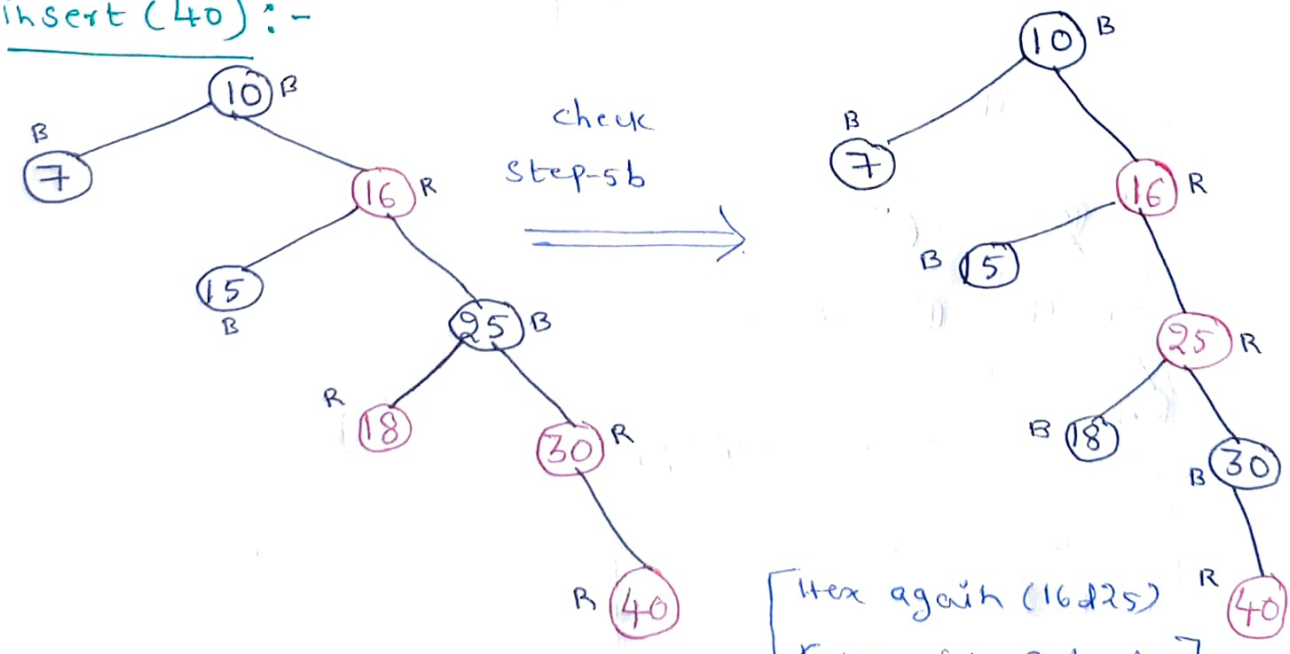
insert (30):

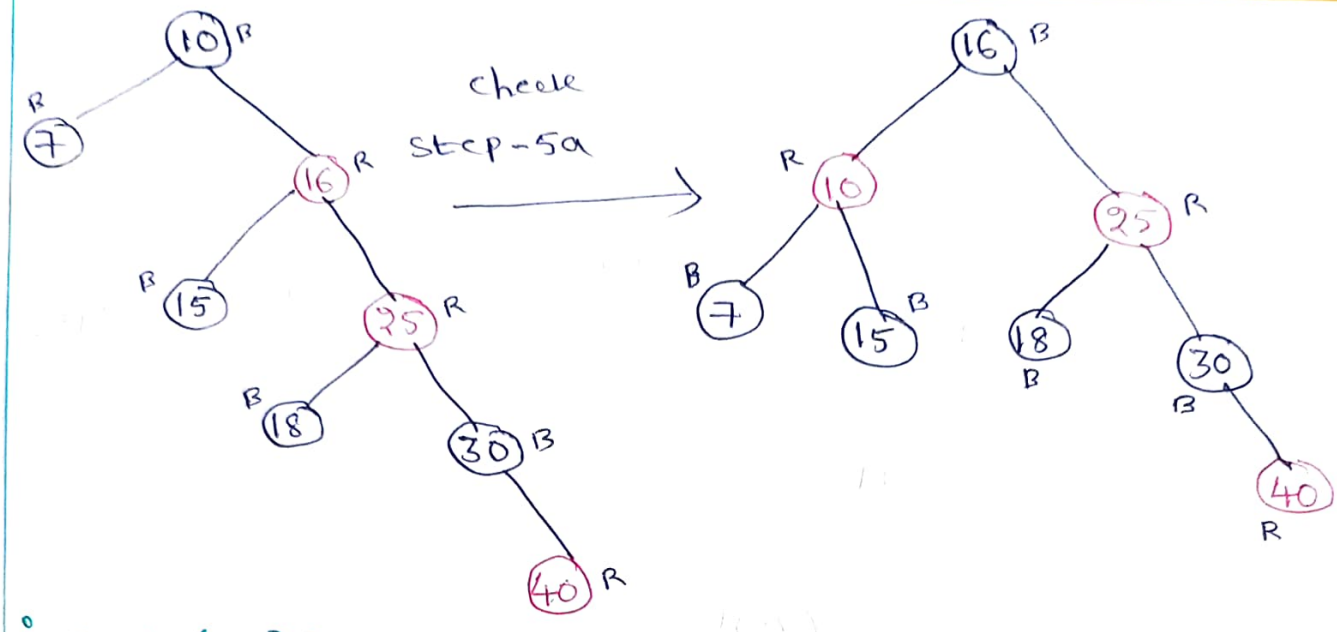


insert (25):

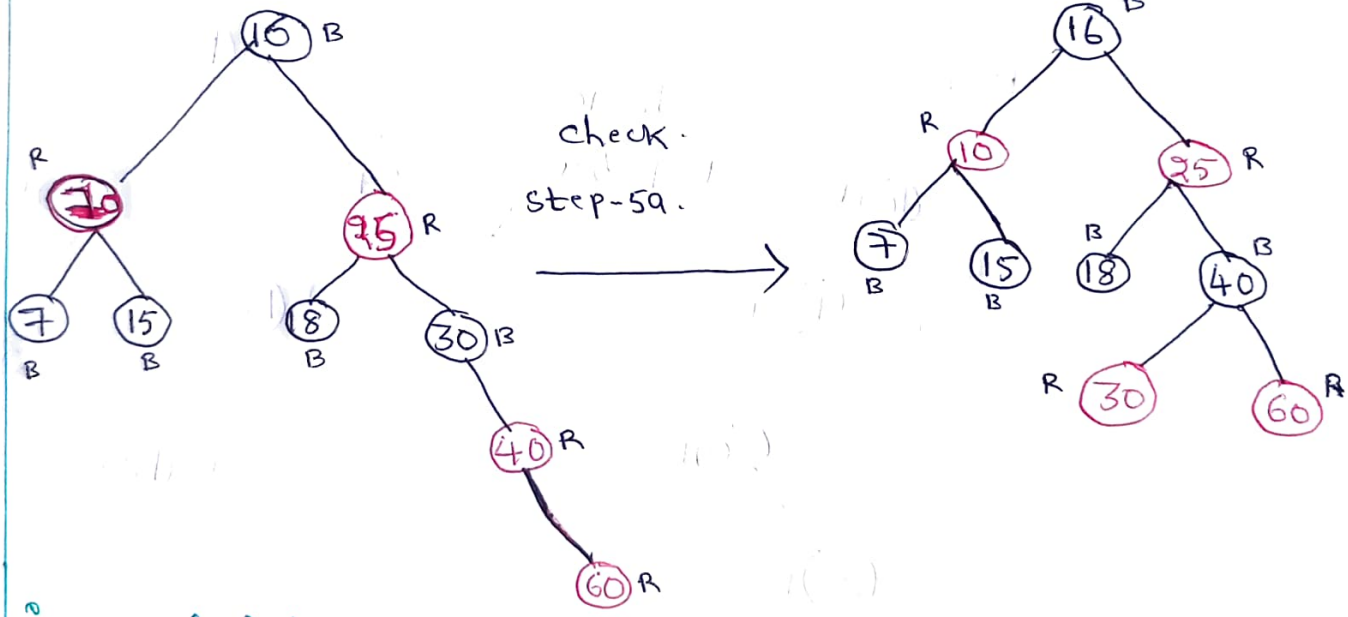


insert (40):-

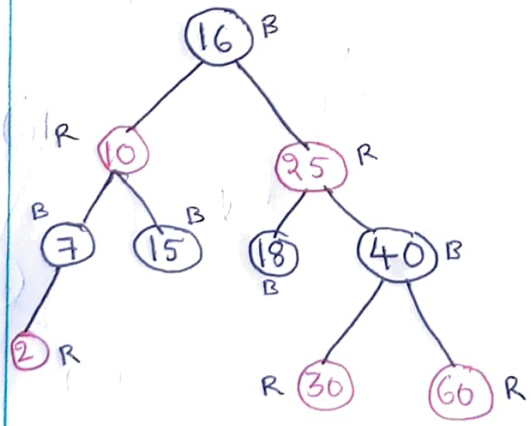




insert (60) :-

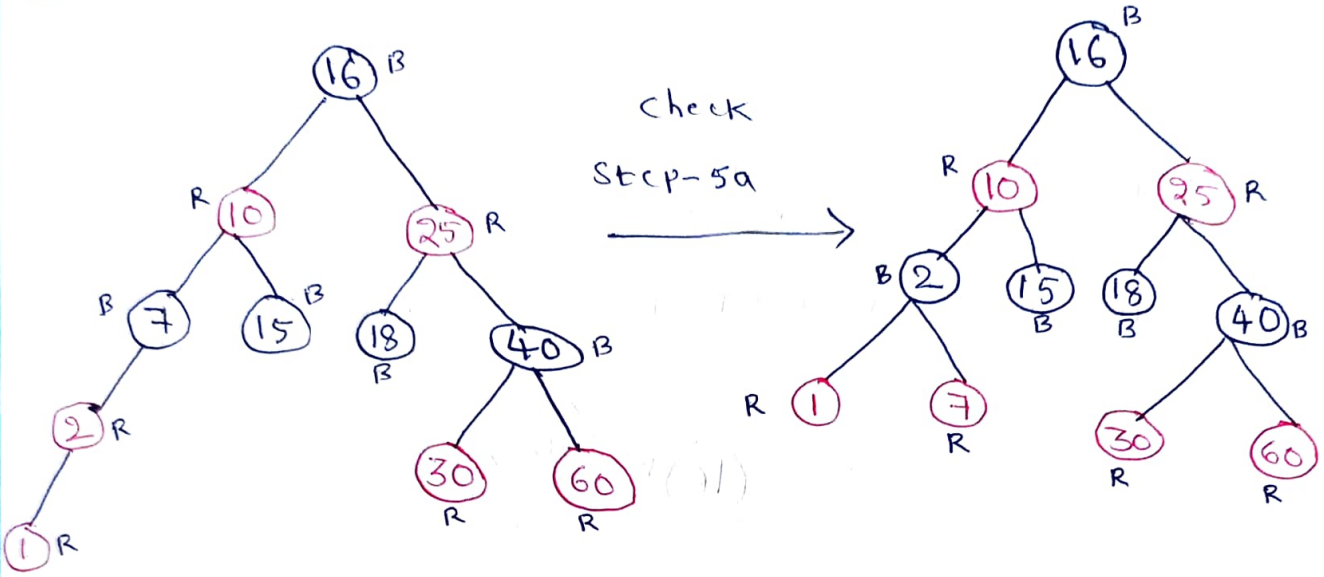


insert (2) :-

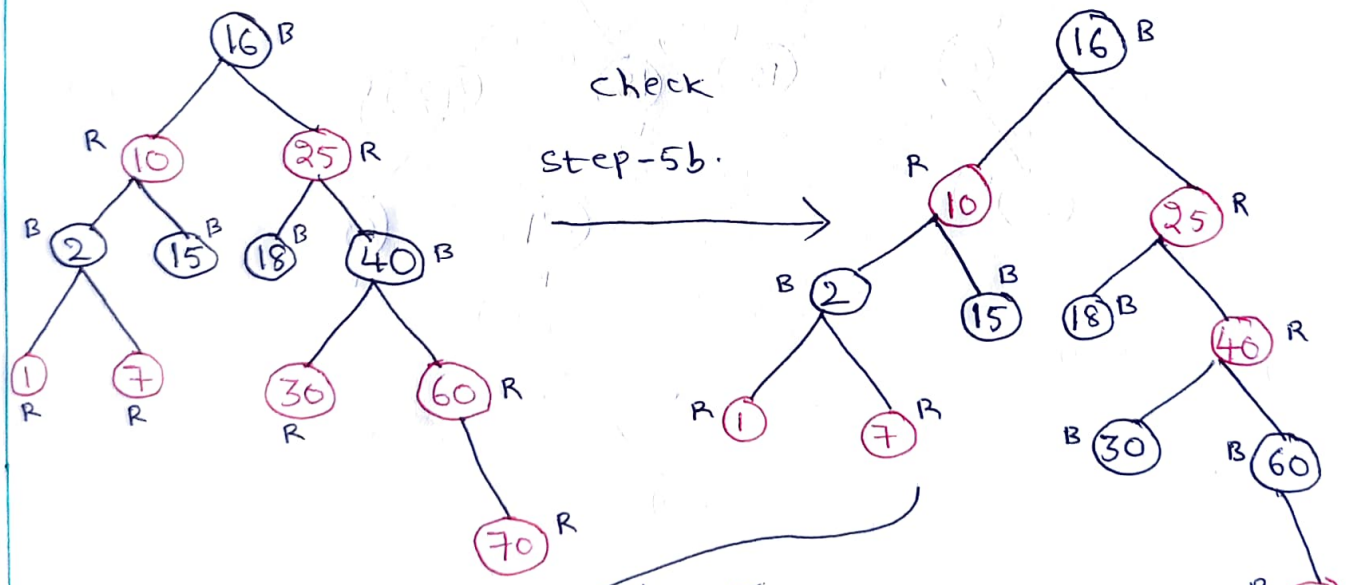


[There is no two Red node consecutives]

insert (1) :-

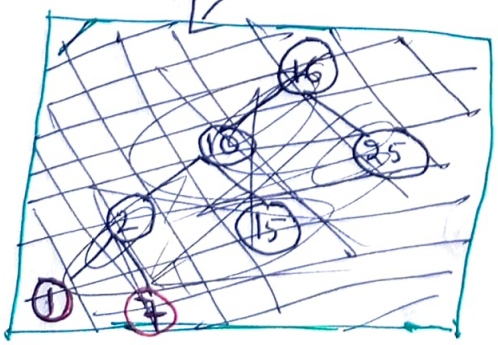


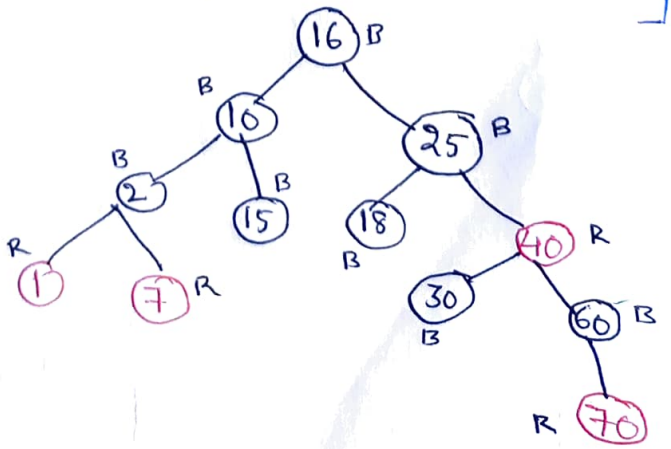
insert (70) :-



check step-5b

Here again (25 & 40) nodes are two Red consecutive nodes.



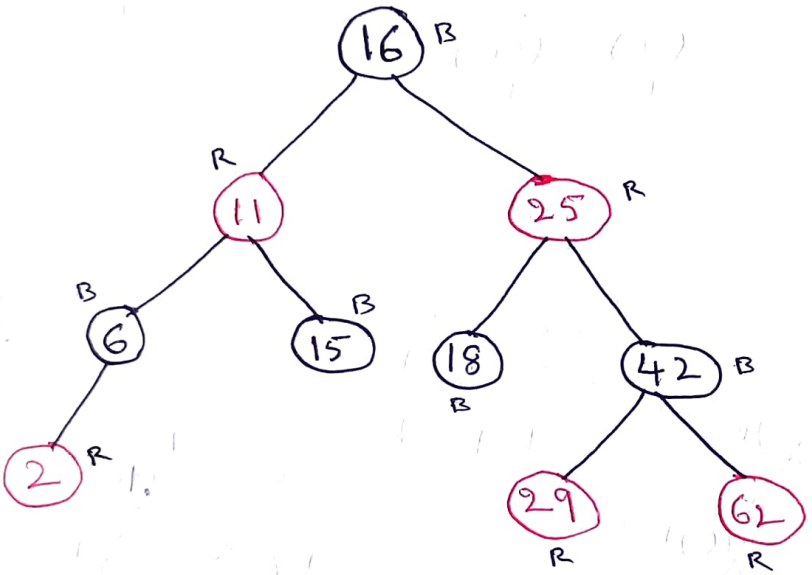


Ex4) Create a Red-Black Tree by inserting following sequence of numbers.

11, 18, 6, 15, 16, 29, 25, 42, 62, 2

Sol: After inserting elements—

The final Red-Black Tree is



Splay Tree

Splay trees are the self-balancing or **self-adjusted binary search trees**. In other words, we can say that the splay trees are the variants of the binary search trees.

The prerequisite for the splay trees that we should know about the binary search trees.

- As we already know, the time complexity of a binary search tree in every case. The time complexity of a binary search tree in the average case is **O(logn)** and the time complexity in the worst case is O(n).
- In a binary search tree, the value of the left sub tree is smaller than the root node, and the value of the right sub tree is greater than the root node; in such case, the time complexity would be **O(logn)**.
- If the binary tree is left-skewed or right-skewed, then the time complexity would be O(n).
- To limit the skewness, the AVL and Red-Black tree came into the picture, having **O(logn)** time complexity for all the operations in all the cases.
- We can also improve this time complexity by doing more practical implementations, so the new Tree data structure was designed, known as a Splay tree.

What is a Splay Tree?

A splay tree is a self-balancing tree, but AVL and Red-Black trees are also self-balancing trees then. What makes the splay tree unique two trees? It has **one extra property that makes it unique is *splaying.***

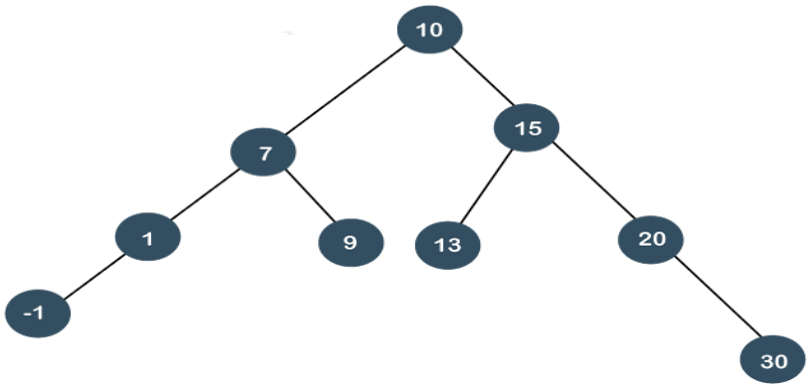
Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

- A splay tree contains the same operations as a Binary search tree, i.e., Insertion, deletion and searching, but it also contains **one more operation, i.e., splaying**. So. All the operations in the splay tree are followed by splaying.

Splaying an element is the process of bringing it to the root position by performing suitable rotation operations.

- Splay trees are not strictly balanced trees, but they are roughly balanced trees
- In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.
- By splaying elements we bring more **frequently used elements closer to the root** of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Suppose we want to search 7 element in the tree, which is shown below:



- To search any element in the splay tree, first, we will perform the standard binary search tree operation. As 7 is less than 10 so we will come to the left of the root node. After performing the search operation, we need to **perform splaying**.
- Here splaying means that the operation that we are performing on any element **should become the root node** after performing some rearrangements. The rearrangement of the tree will be done through the rotations.

Rotations in Splay Tree:

There are six types of rotations used for splaying:

1. Zig rotation (Right rotation)
2. Zag rotation (Left rotation)
3. Zig zag (Zig followed by zag)
4. Zag zig (Zag followed by zig)
5. Zig zig (two right rotations)
6. Zag zag (two left rotations)

Factors required for selecting a type of rotation

The following are the factors used for selecting a type of rotation:

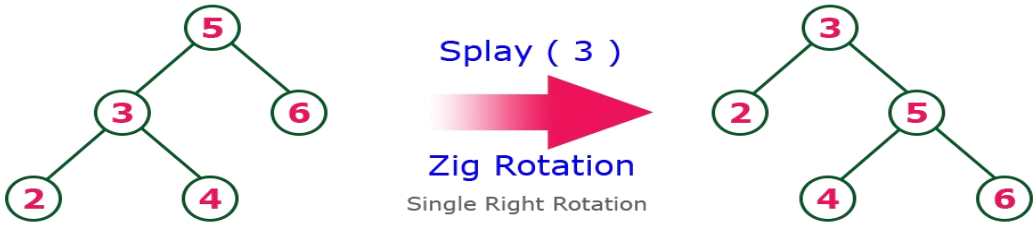
- ✓ Does the node which we are trying to rotate have a grandparent?
- ✓ Is the node left or right child of the parent?
- ✓ Is the node left or right child of the grandparent?

Examples:

Zig Rotation:

- The **Zig Rotation** in splay tree is similar to the single **right rotation** in AVL Tree rotations.
- In zig rotation, every node moves one position to the right from its current position.

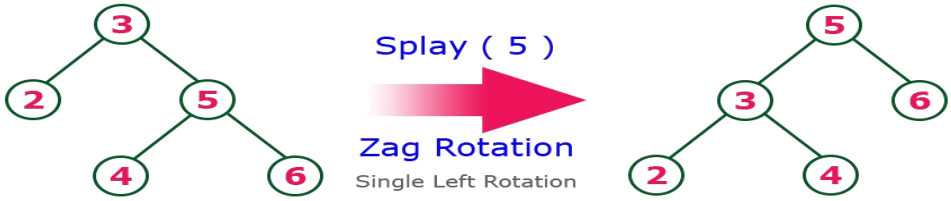
Consider the following example...



Zag Rotation:

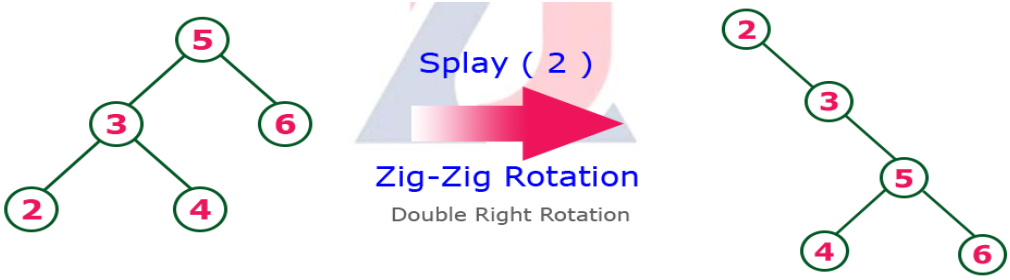
- The **Zag Rotation** in splay tree is similar to the single **left rotation** in AVL Tree rotations.
- In zag rotation, every node moves one position to the left from its current position.

Consider the following example...



Zig-Zig Rotation:

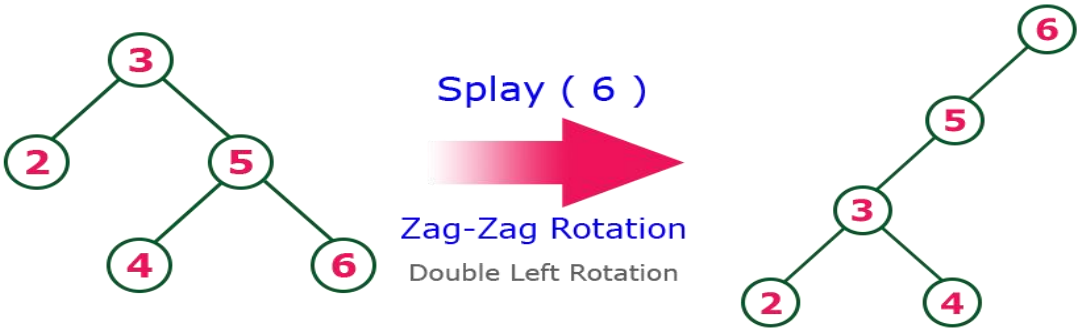
- The **Zig-Zig Rotation** in splay tree is a **double zig rotation**.
- In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



Zag-Zag Rotation:

- The **Zag-Zag Rotation** in splay tree is a **double zag** rotation.
- In zag-zag rotation, every node moves two positions to the left from its current position.

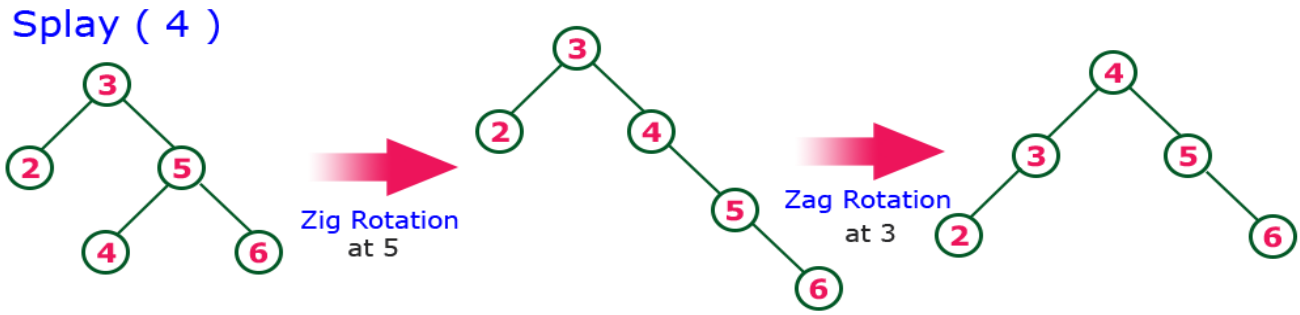
Consider the following example...



Zig-Zag Rotation:

- The **Zig-Zag Rotation** in splay tree is a sequence of **zig rotation followed by zag rotation**.
- In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position.

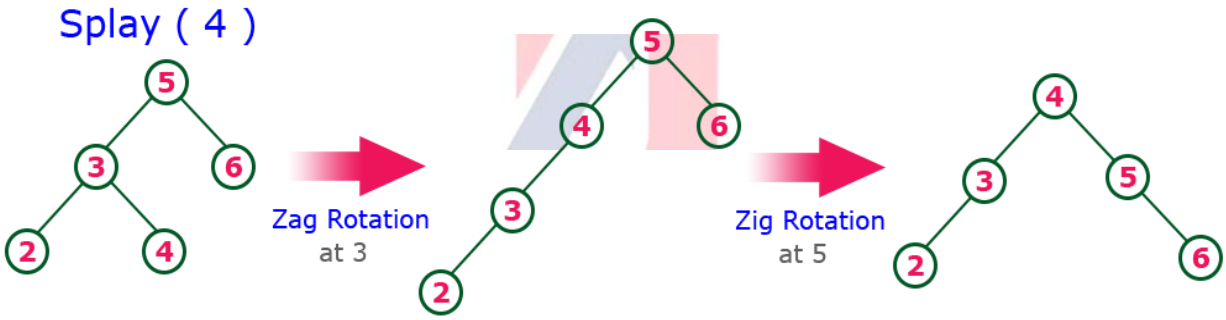
Consider the following example...



Zag-Zig Rotation:

- The **Zag-Zig Rotation** in splay tree is a sequence of **zag rotation followed by zig rotation.**
- In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position.

Consider the following example...



Every Splay tree must be a binary search tree but it is need not to be balanced tree.