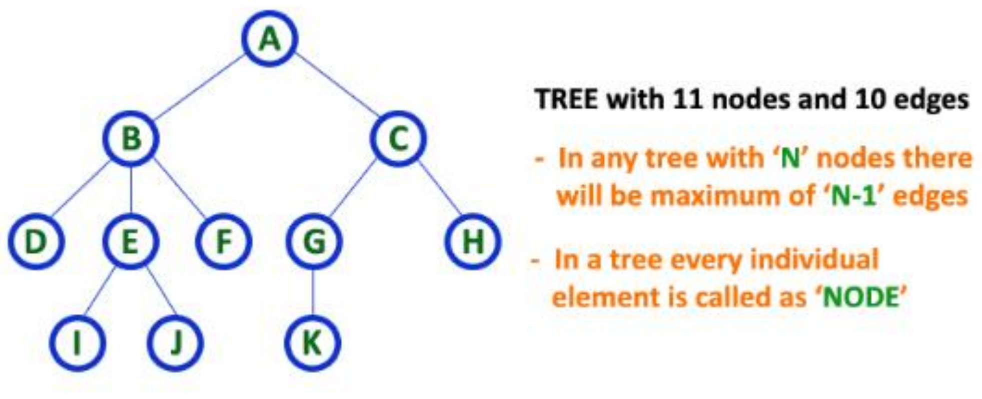
MODULE IV

TREES

Definition of Trees: Tree is a non-linear data structure which organizes data in hierarchical structure. Tree represents the nodes connected by edges. The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent.


A
B
C
D
E
F
G
H
I
J
K
TREE with 11 nodes and 10 edges
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

3.1 Basic Terminologies:

In a tree data structure, we use the following terminology...

1. Root

In a tree data structure, the first node is called as **Root Node**. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. In the above diagram, A is root node.

2. Edge

In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. Parent node can also be defined as "**The node which has child / children**". Here, A,B,C,E ang G are parent nodes.

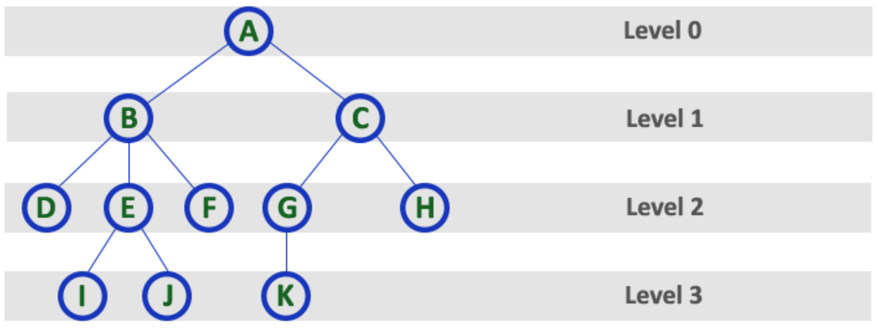
4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node.
B and C are children of A.
G And H are children of C

K is child of G

5. Siblings
In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**.
Here, B and C are siblings
D,E and F are siblings
G and H are siblings
I and J are siblings

6. Leaf
In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as External Nodes or Terminal node.
D, I, J, F, K and H are leaf nodes
7. Internal Nodes
In a tree data structure, the node which has at least one child is called as **INTERNAL Node**. Internal nodes are also called as '**Non-Terminal**' nodes.
A,B,C,E and G are internal nodes.
8. Degree
In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'
Degree of B is 3
Degree of A is 2
Degree of F is 0
9. Level
In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



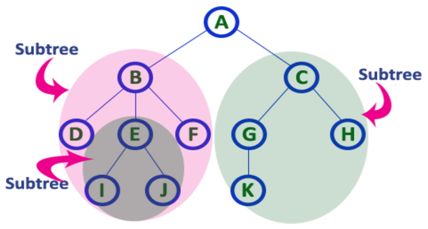
10. Height
In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. Here height of tree is 3.
11. Depth
In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. Here depth of tree is 3.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between those two Nodes. Path between A and j is A-B-E-J.

13. Sub Tree

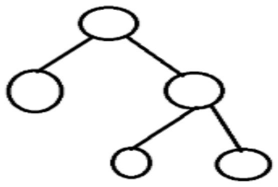
In a tree data structure, each child from a node forms a sub tree recursively. Every child node will form a sub tree on its parent node.



3.2 Binary Tree

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as left child and the other is known as right child. In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



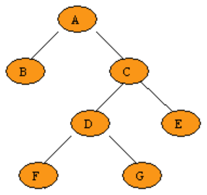
Binary Tree

Types of binary trees:

There are different types of binary trees and they are...

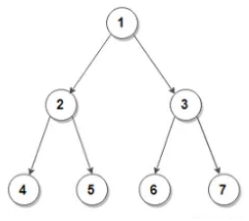
a) **Strictly Binary Tree**

- In strictly binary tree, every node should have exactly two children or none.
- Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree.



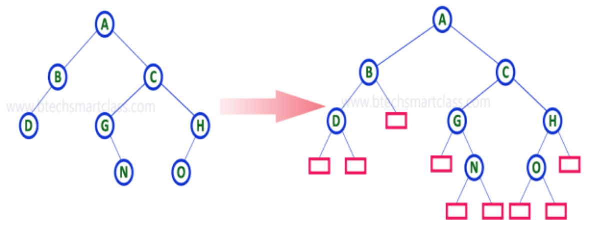
b) **Complete binary tree:**

- A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled and all nodes are left as possible.
- Complete binary tree is also called as Perfect Binary Tree.



c) Extended Binary Tree

- The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



3.3 Representation of binary tree:

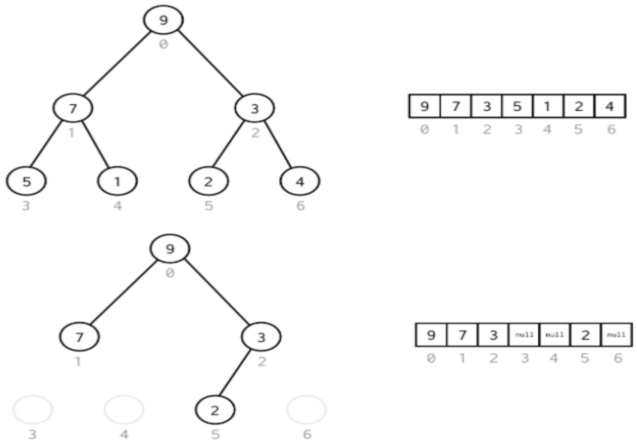
A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...

1. Array Representation

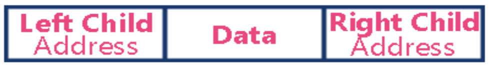
In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. Consider the above example of binary tree and it is represented as follows...



2. Linked List Representation

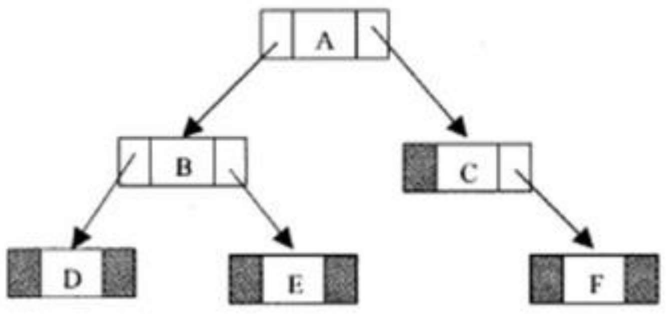
We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field stores left child address, second stores actual data and third for storing right child address.

In this linked list representation, a node has the following structure...



The above example of binary tree represented using Linked list representation is shown as follows...

The linked list representation of this binary tree is:



3.4 Binary tree traversals

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node.

```
Algorithm Inorder(tree)
  1. Traverse the left subtree, i.e., call Inorder(left-subtree)
  2. Visit the root.
  3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child.

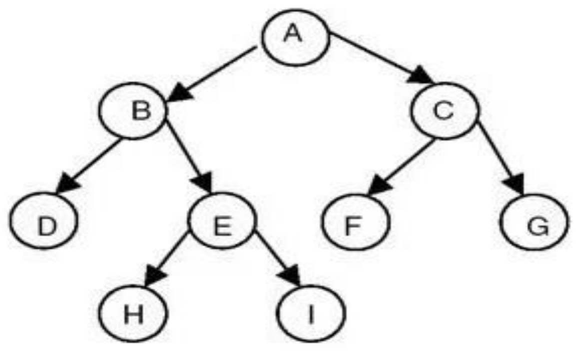
```
Algorithm Preorder(tree)
  1. Visit the root.
  2. Traverse the left subtree, i.e., call Preorder(left-subtree)
  3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

3. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node.

```
Algorithm Postorder(tree)
  1. Traverse the left subtree, i.e., call Postorder(left-subtree)
  2. Traverse the right subtree, i.e., call Postorder(right-subtree)
  3. Visit the root.
```

Example:



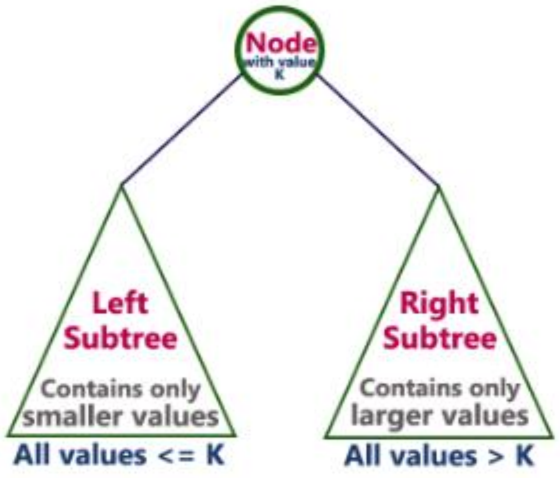
Inorder : DBHEIAFCG

Preorder : ABDEHICFG

Postorder : DHIEBFGCA

3.5.1 Binary search tree:

- In a binary tree, every node can have maximum of two children but there is no order of nodes based on their values.
- In binary tree, the elements are arranged as they arrive to the tree, from top to bottom and left to right.
- To enhance the performance of binary tree, we use special type of binary tree known as **Binary Search Tree**.
- Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.



Note: Every binary search tree is a binary tree but all the binary trees need not to be binary search trees.

Operations of binary search tree:

1. Insertion
2. Deletion

Insertion: In this operation, a new node can be inserted into any position in a binary tree.

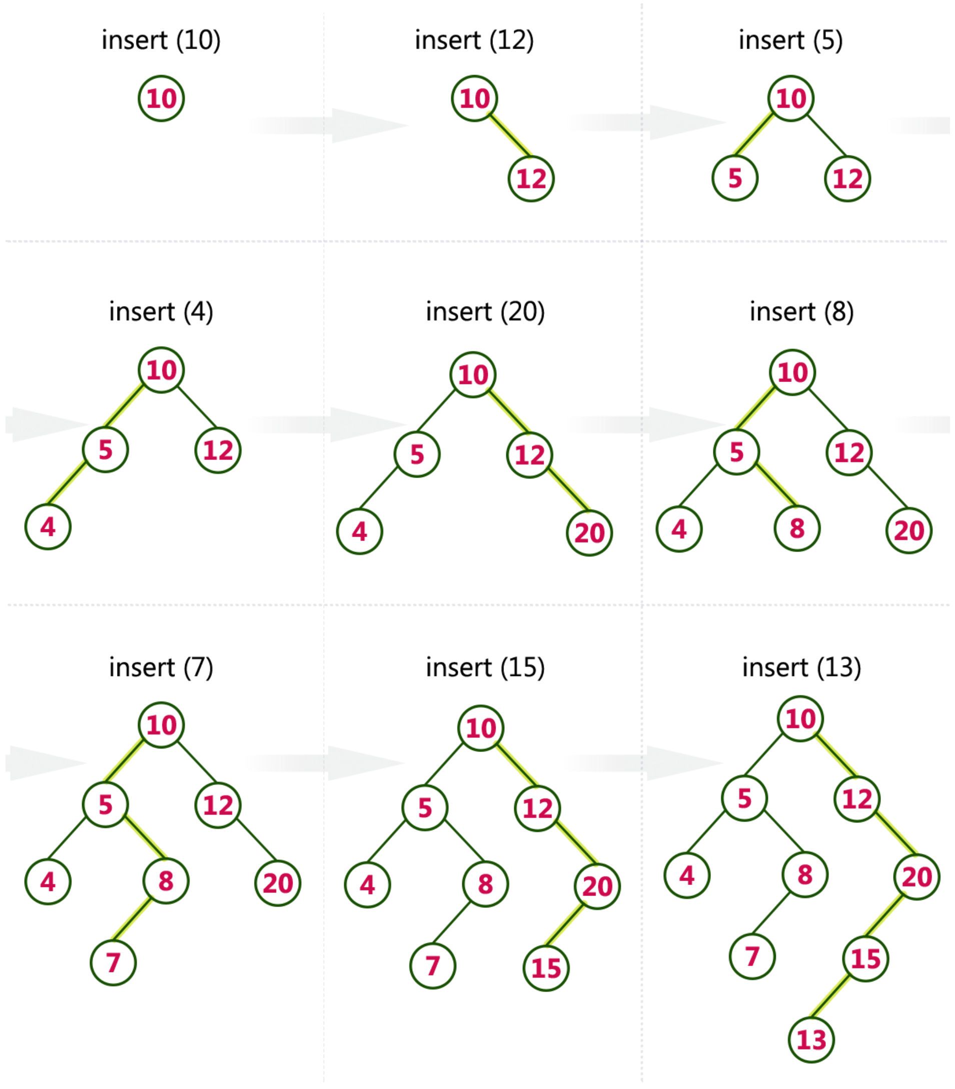
- **Step 1:** Create a new node with given value and set its **left** and **right** to **NULL**.
- **Step 2:** Check whether tree is Empty.
- **Step 3:** If the tree is **Empty**, then set **root** = **new node**.
- **Step 4:** If the tree is **Not Empty**, then check whether value of new node is **smaller** or **larger** than the root node.
- **Step 5:** If newNode is **smaller** than the **root node**, then insert in the left sub-tree.
- **Step 6:** If newNode is **larger** than the root node, then insert in the right sub-tree.
- **Step 7:** Repeat the above step until we reach to a **leaf** node.

Example:

Construct a Binary Search Tree by inserting the following sequence of numbers...

10, 12, 5, 4, 20, 8, 7, 15 and 13

Above elements are inserted into a Binary Search Tree as follows...



Deletion: When we delete a node, three possibilities arise. The procedure for deleting a given node **'z'** from a binary search tree needs to consider the following three cases:

- If the node 'Z' has no children simply delete it.
- If the node 'Z' has only a single child, then delete 'Z' by creating a link between its parent and child node.
- If the node 'Z' has two children then find the **largest** node in its **left sub tree** (OR) the **smallest** node in its **right sub tree and swap both deleting item and the node found above.**

Example:

1) ***Node to be deleted is leaf***

```
          50                            50
       /     \         delete(20)      /   \
      30      70       --------->    30     70
     /  \    /  \                     \    /  \
   20   40  60   80                   40  60   80
```

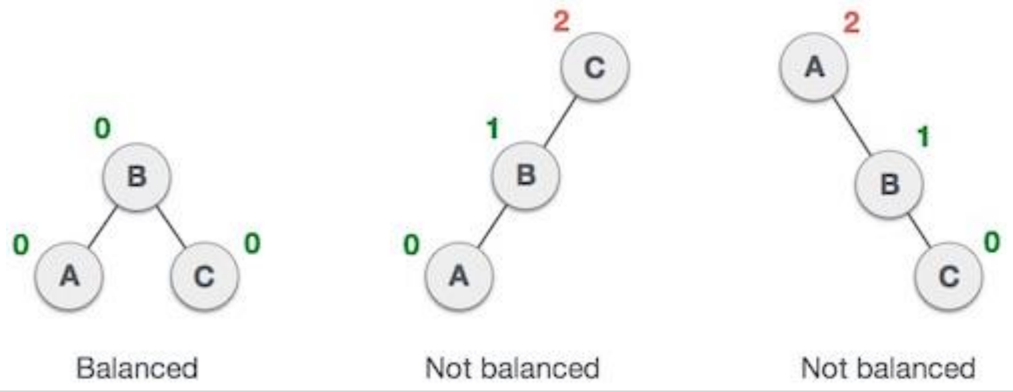
2) ***Node to be deleted has only one child***

```
          50                            50
       /     \         delete(30)      /   \
      30      70       --------->    40     70
        \    /  \                          /  \
        40  60   80                       60   80
```

3) ***Node to be deleted has two children***

```
          50                            60
       /     \         delete(50)      /   \
      40      70       --------->    40    70
             /  \                            \
            60   80                           80
```

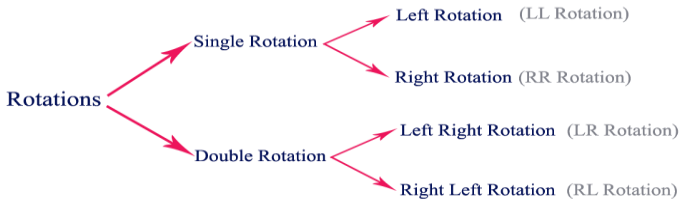
Height balanced Binary Trees (AVL Trees):
An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.
AVL was founded by Adelson, Velski & Landis.
Here we see that the first tree is balanced and the next two trees are not balanced −



BalanceFactor = height(left-sutree) − height(right-sutree)

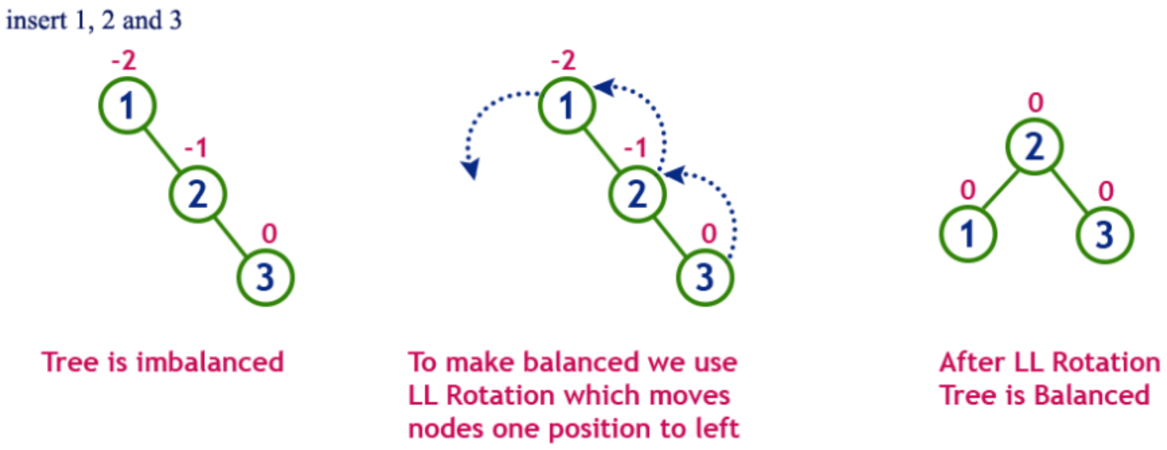
AVL Rotations
An AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation
  There are **four** rotations and they are classified into **two** types.


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

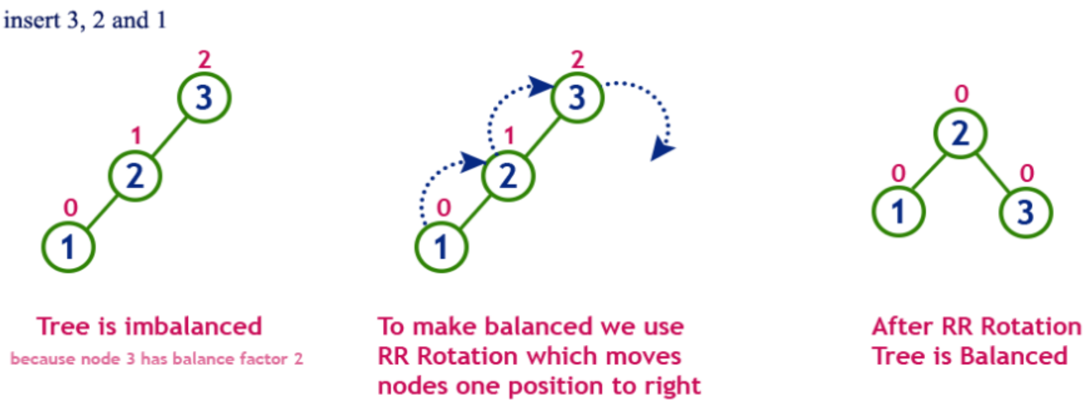
Single Left Rotation (LL Rotation)

- In LL Rotation every node moves one position to left from the current position.


insert 1, 2 and 3
-2
1
-1
2
0
3
Tree is imbalanced
To make balanced we use LL Rotation which moves nodes one position to left
After LL Rotation Tree is Balanced

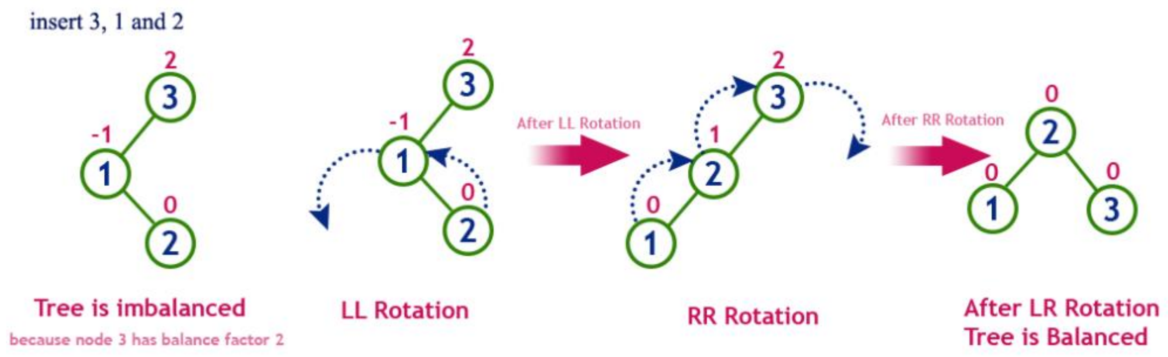
Single Right Rotation (RR Rotation)

In RR Rotation every node moves one position to right from the current position.


insert 3, 2 and 1
2
3
1
2
0
1
Tree is imbalanced
because node 3 has balance factor 2
To make balanced we use RR Rotation which moves nodes one position to right
After RR Rotation Tree is Balanced

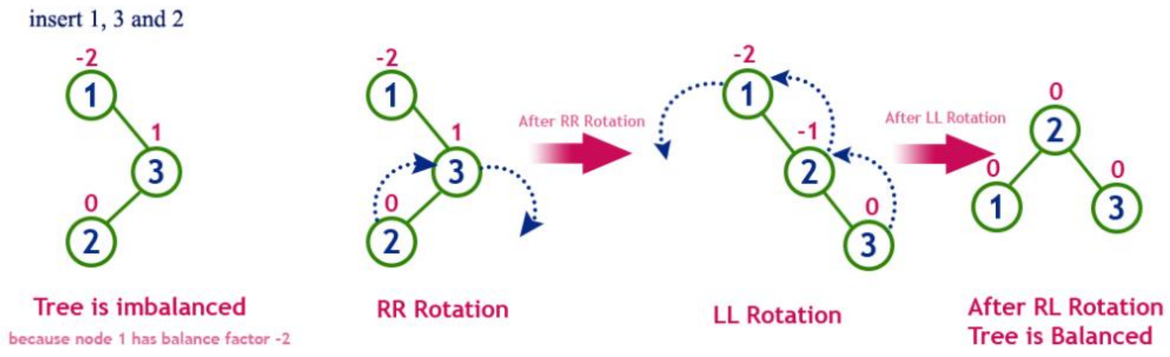
Left Right Rotation (LR Rotation)

The LR Rotation is combination of single left rotation followed by single right rotation.

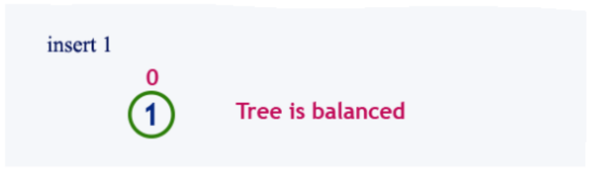


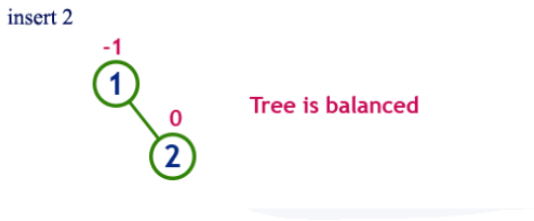
Right Left Rotation (RL Rotation)

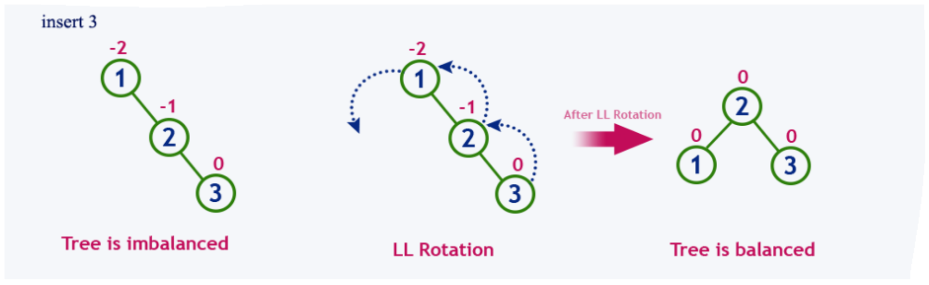
The RL Rotation is combination of single right rotation followed by single left rotation.

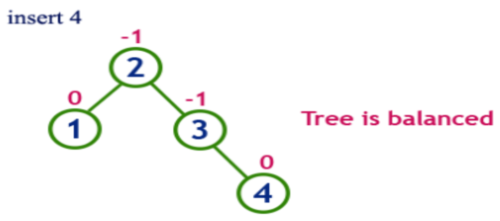


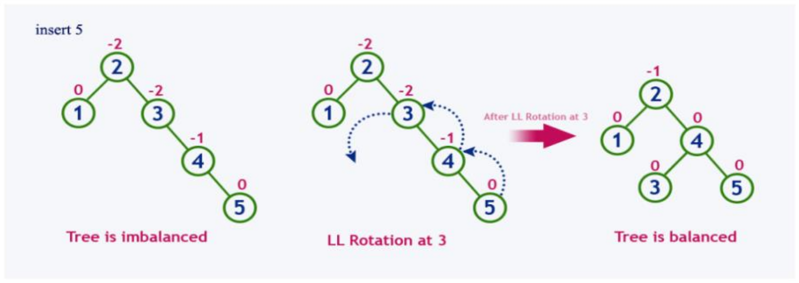
Example: Construct an AVL Tree by inserting numbers from 1 to 8.

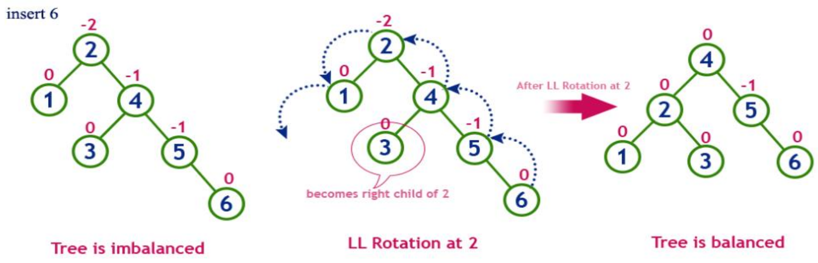


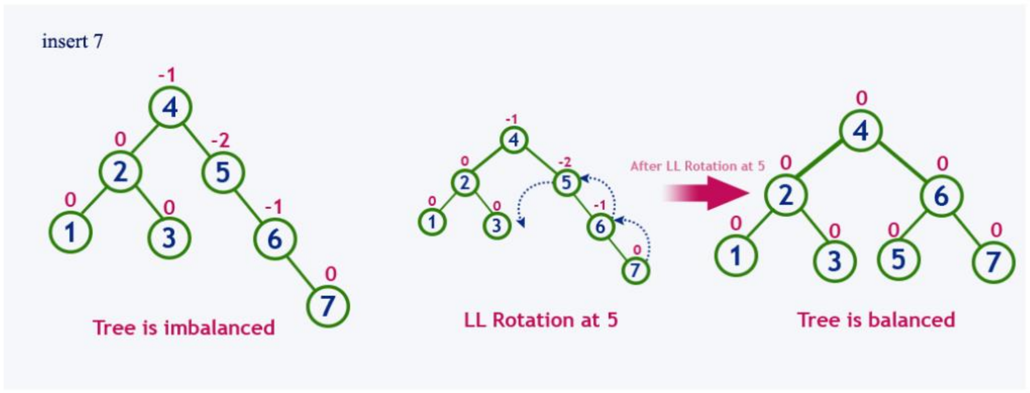


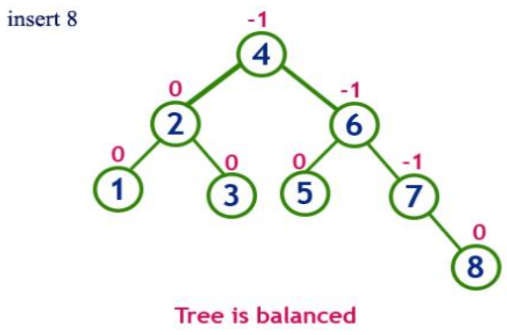



insert 4
Tree is balanced


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



Operations on AVL tree:

- Insertion
- Deletion

Insertion algorithm:

- 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 tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

Deletion algorithm:

- Step 1: Delete element from tree using Binary Search Tree insertion logic.
- Step 2: After Deletion, 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 tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

3.8 B-Trees:

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

B-Tree of Order m has the following properties...

- **Property #1** - All the 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 at least 2 children.
- **Property #5** - A non leaf node with n-1 keys must have n number of children.

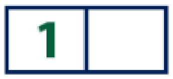
- **Property #6** - All the key values within a node must be in Ascending Order.

Operations on B-tree:
1. Insertion
2. Deletion

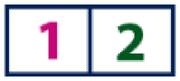
Insertion algorithm:
In a B-Tree, the new element must be added only at leaf node. That means, always the new key Value is attached to leaf node only. The insertion operation is performed as follows...

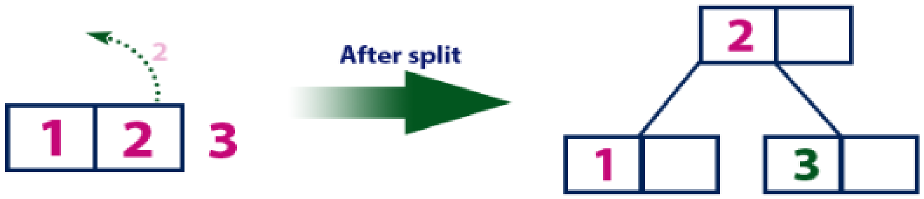
- **Step 1:** Check whether tree is Empty.
- **Step 2:** If tree is **Empty**, then create a new node with new key value and insert into the tree as a root node.
- **Step 3:** If tree is **Not Empty**, insert every element from list to the leaf node until it is full.
- **Step 4:** Once it is full, split the node into two nodes by sending middle value to its parent node.
- **Step 5:** If the splitting is occurring to the 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.
Insert 1



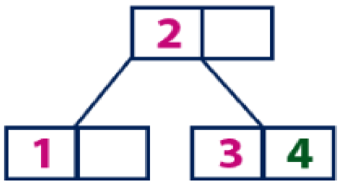
Insert 2



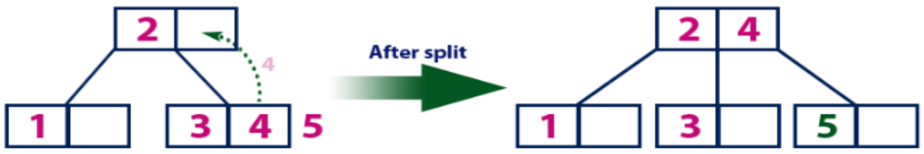


Insert 3

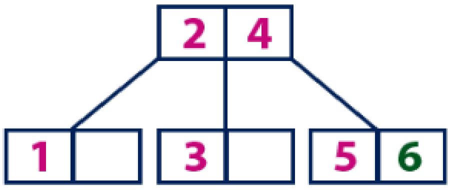
Insert 4



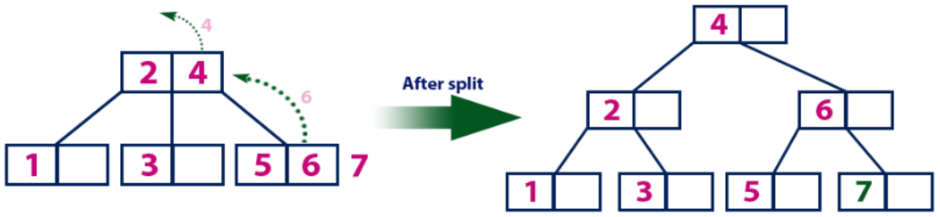
Insert 5



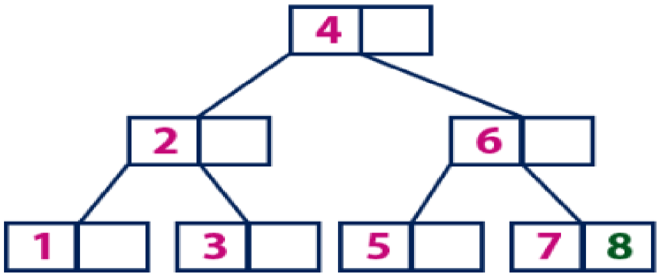
Insert 6



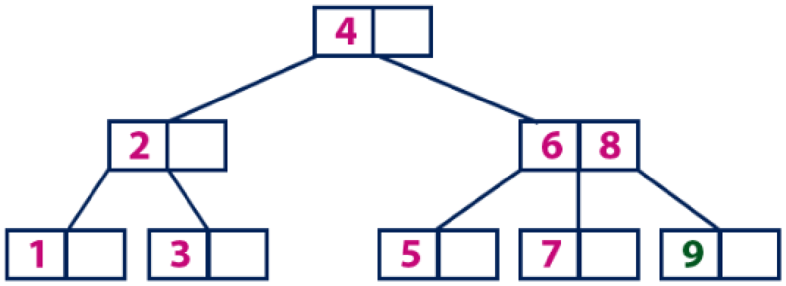
Insert 7



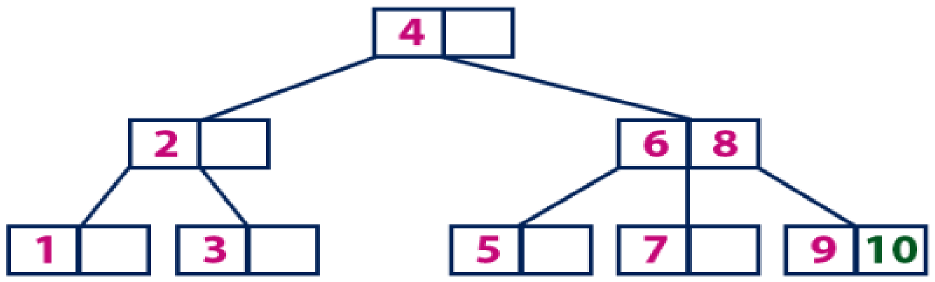
Insert 8



Insert 9



Insert 10


4
2
6 8
1
3
5
7
9 10