UNIT-III

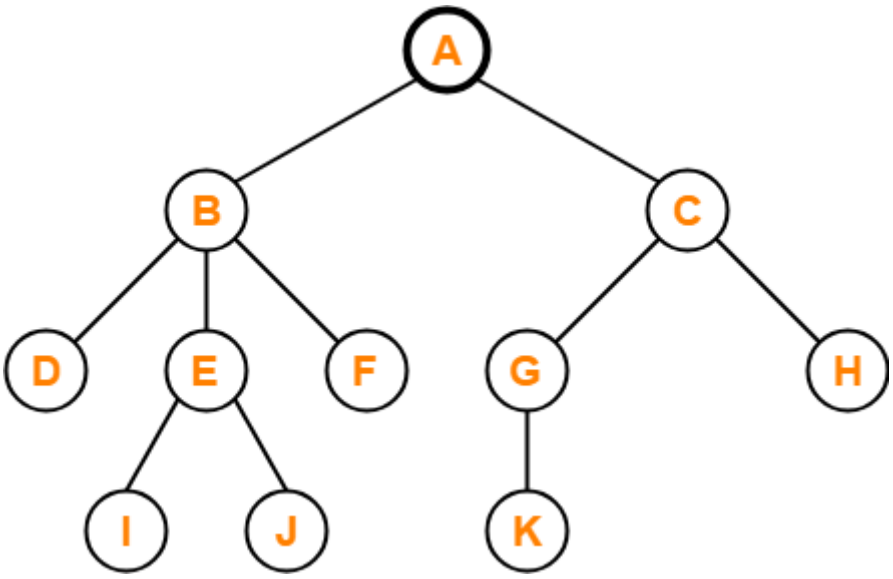
TREES

Tree Data Structure : Tree is a non-linear data structure which organizes data in a hierarchical structure. In a tree data structure, a node can have any number of child nodes.

(OR)

If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.

Example :



Properties-

The important properties of tree data structure are-

- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has exactly (n-1) edges.
- A graph is a tree if and only if it is minimally connected.
- Any connected graph with n vertices and (n-1) edges is a tree.

Tree Terminology-

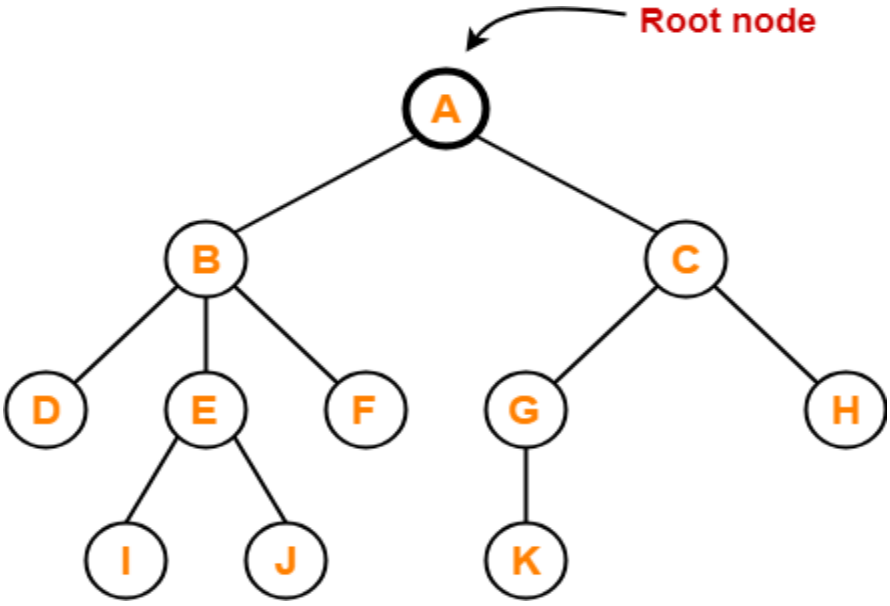
The important terms related to tree data structure are-

1. Root
2. Edge
3. Parent
4. Child Node
5. Siblings
6. Internal Node
7. Degree of a Node
8. Height of a tree
9. Depth of a tree
10. Level of a tree
11. Leaf node
12. Subtree

1. Root-

1. The first node from where the tree originates is called as a root node.
2. In any tree, there must be only one root node.
3. We can never have multiple root nodes in a tree data structure.

Example :

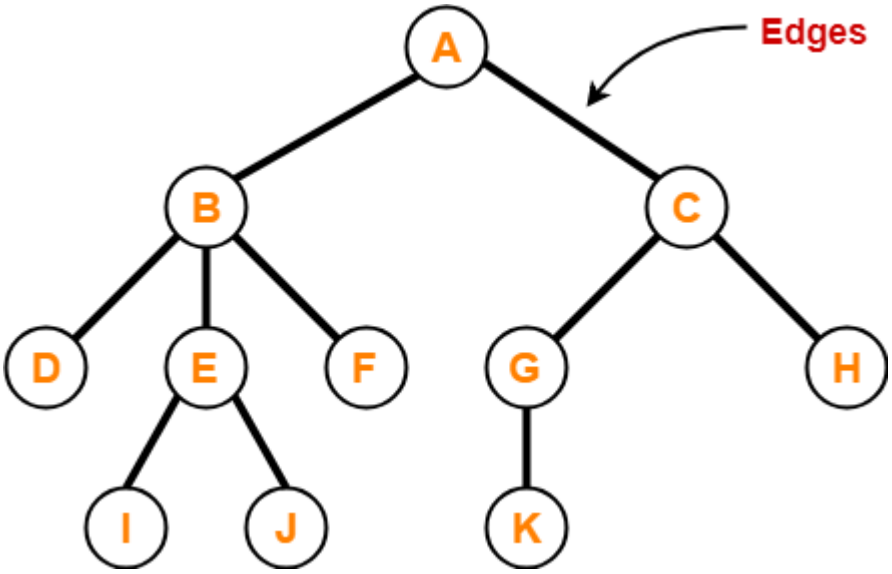


Here, node A is the only root node.

2. Edge :

- The connecting link between any two nodes is called as an edge.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.

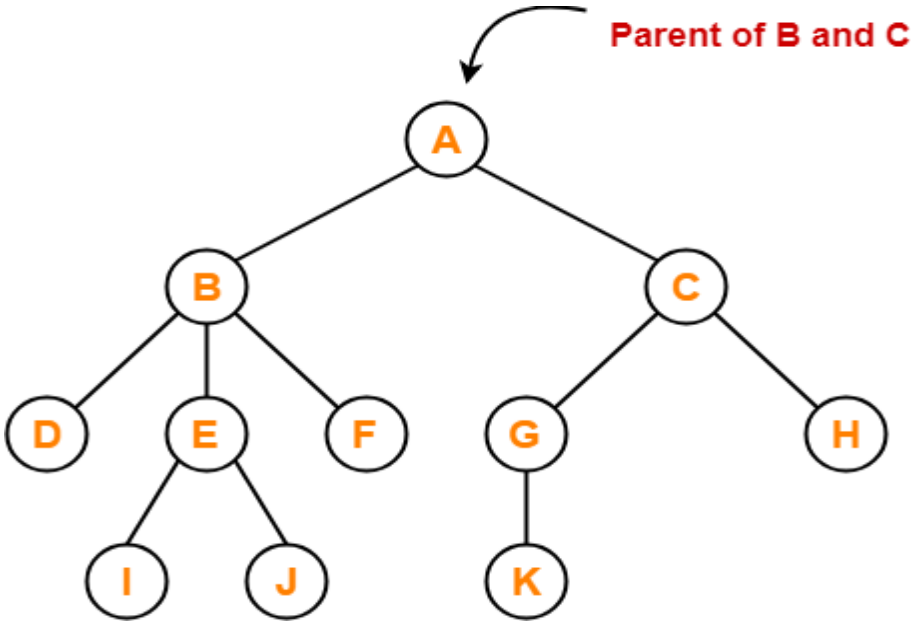
Example :



3. Parent

- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.

Example :



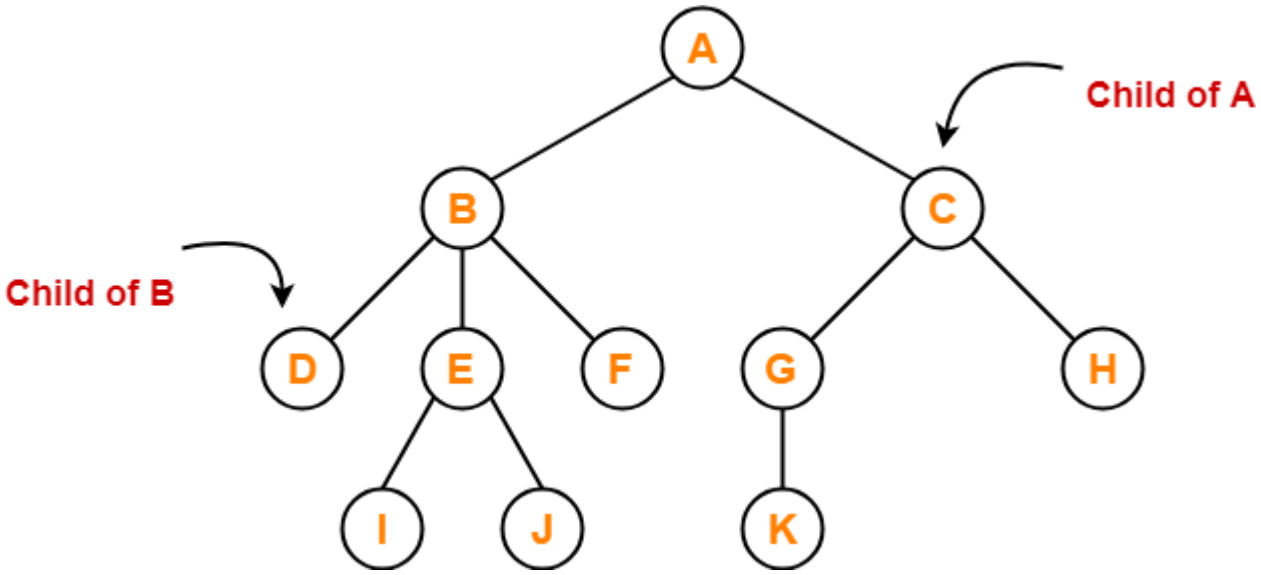
Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

4. Child Node :

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.

Example :



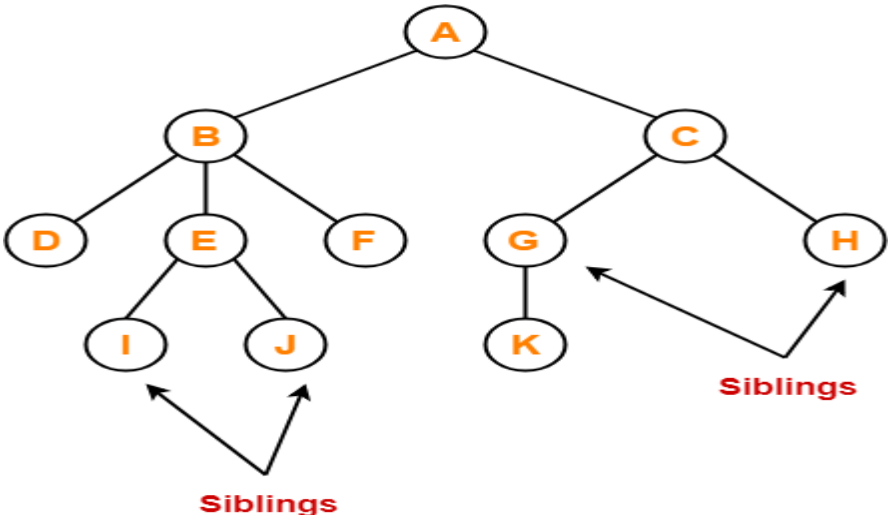
Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

5. Siblings :

- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.

Example :



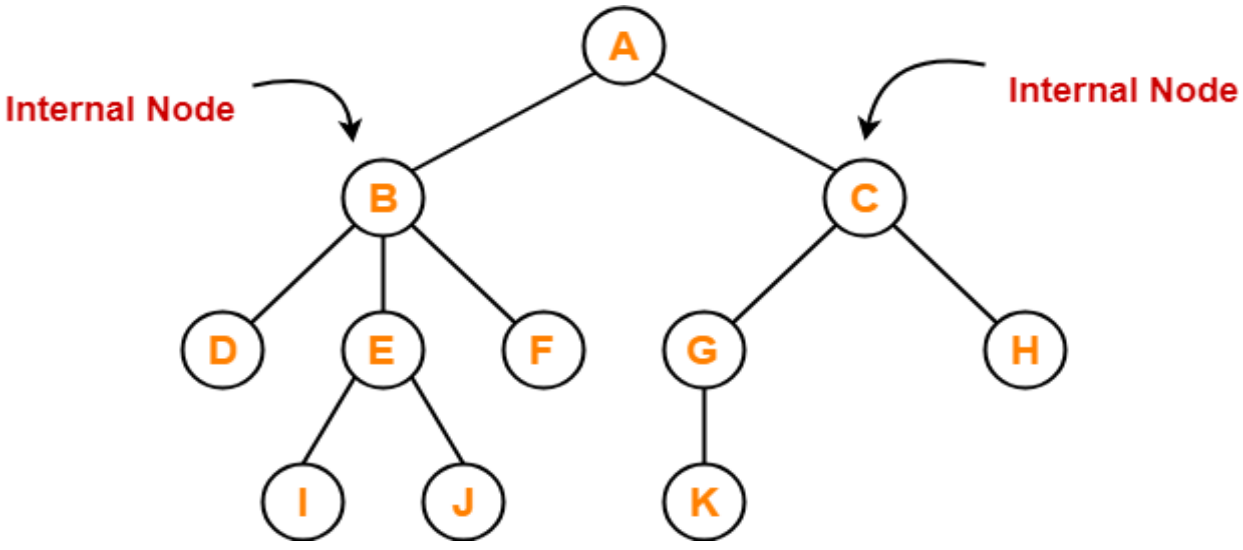
Here,

- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

6. Internal Node

- The node which has at least one child is called as an internal node.
- Internal nodes are also called as non-terminal nodes.
- Every non-leaf node is an internal node.

Example :

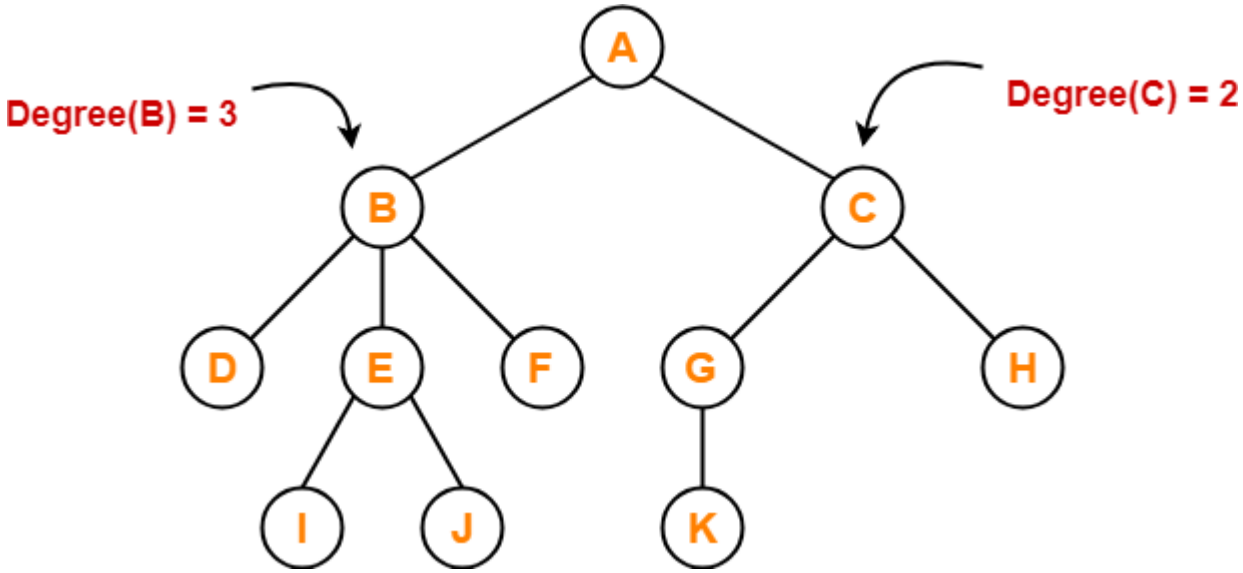


Here, nodes A, B, C, E and G are internal nodes.

7. Degree of a Node :

- Degree of a node is the total number of children of that node.
- Degree of a tree is the highest degree of a node among all the nodes in the tree.

Example :



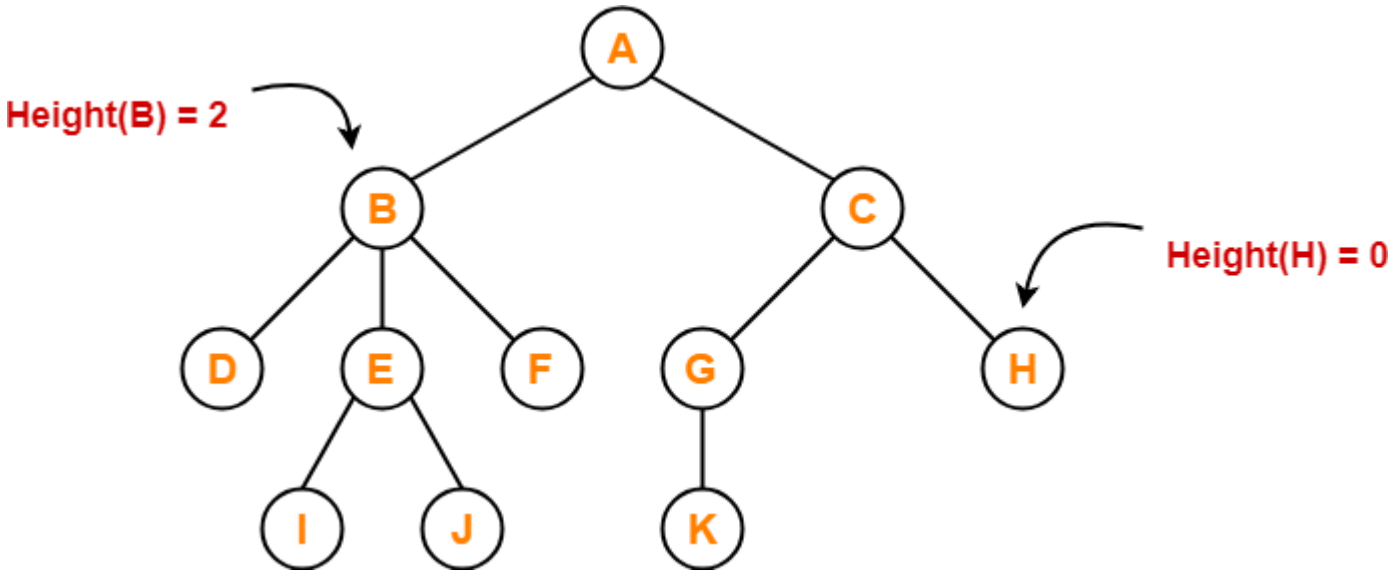
Here,

- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

8. Height of a tree :

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as height of that node.
- Height of a tree is the height of root node.
- Height of all leaf nodes = 0

Example :



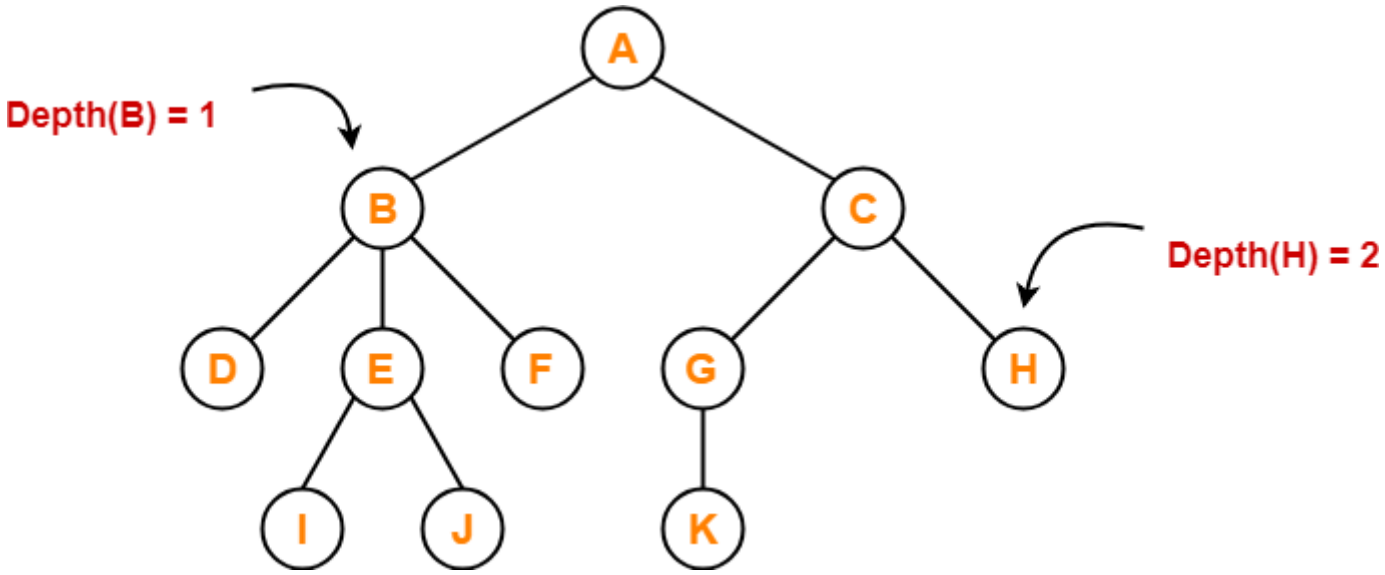
Here,

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

9. Depth of a tree :

- Total number of edges from root node to a particular node is called as depth of that node.
- Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.

Example :



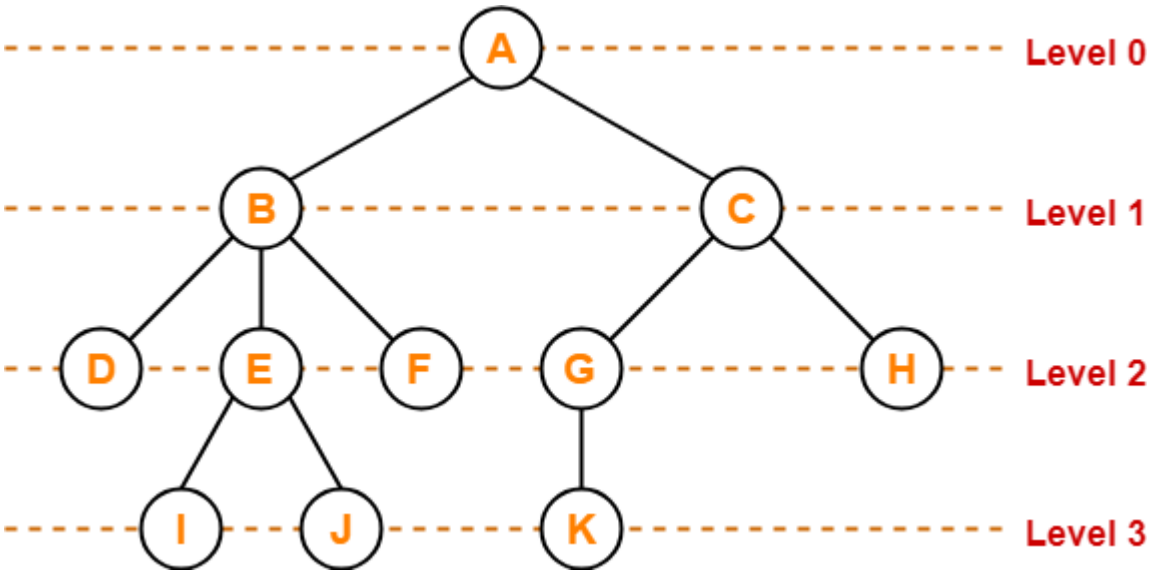
Here,

- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3

10. Level of a tree :

- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.

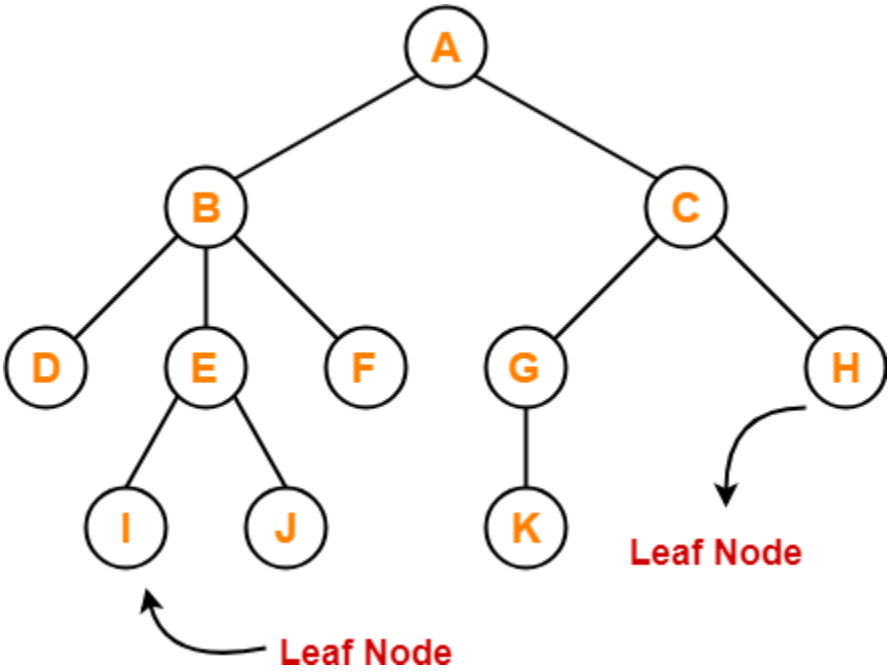
Example :



11. Leaf Node :

- The node which does not have any child is called as a leaf node.
- Leaf nodes are also called as external nodes or terminal nodes.

Example :

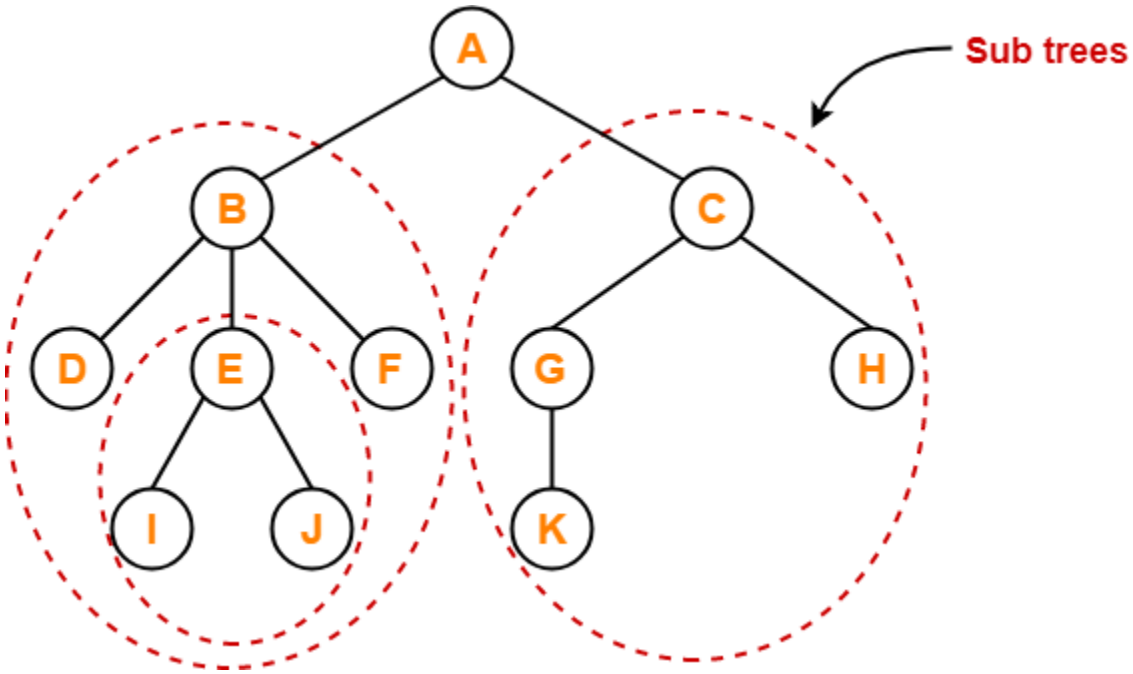


Here, nodes D, I, J, F, K and H are leaf nodes.

12. Subtree :

- In a tree, each child from a node forms a subtree recursively.
- Every child node forms a subtree on its parent node.

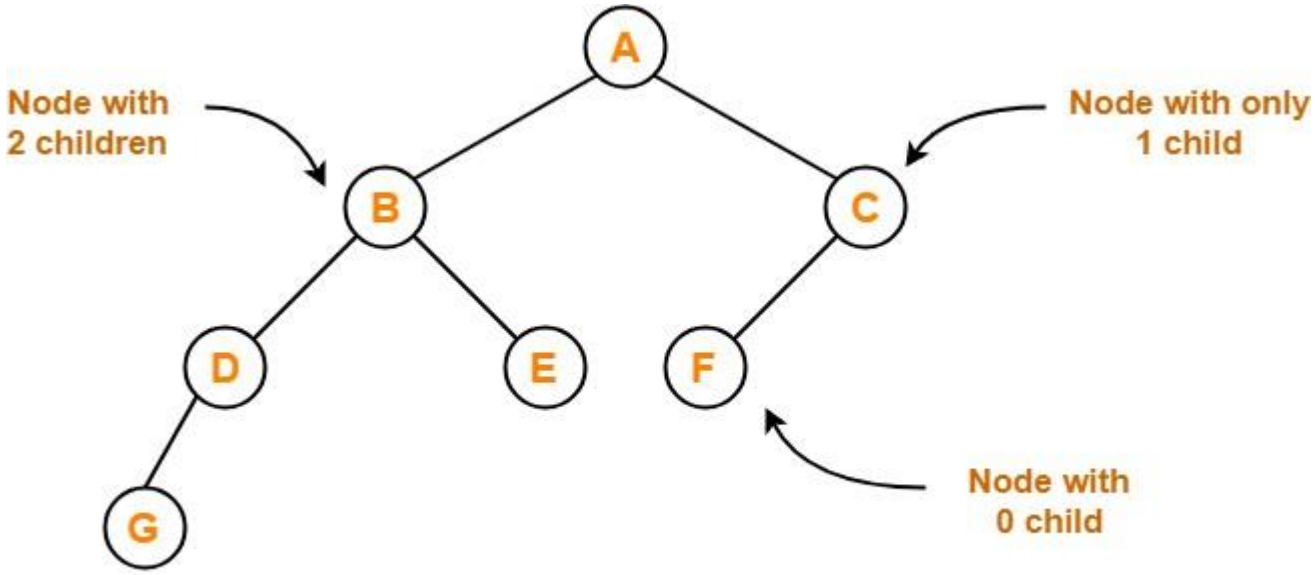
Example :



Binary Tree :

- Binary Tree is a special data structure in which each node can have at most 2 children.
- In Binary tree, each node has either 0 child or 1 child or 2 children.

Example :



Binary Tree Example

Binary Tree Properties

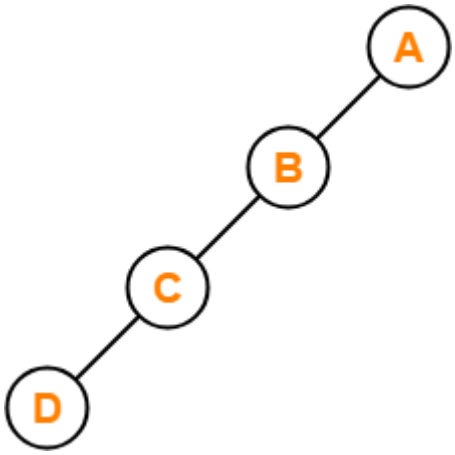
- Minimum number of nodes in a binary tree of height H = H + 1.
- Maximum number of nodes in a binary tree of height H = $2^{H+1} - 1$
- Maximum number of nodes at any level 'L' in a binary tree = 2^{L}

Types of Binary Tree

1. Left Skewed Binary Tree
2. Right Skewed Binary Tree
3. Full Binary Tree (or) Strictly Binary Tree.
4. Complete Binary Tree (or) Perfect Binary Tree

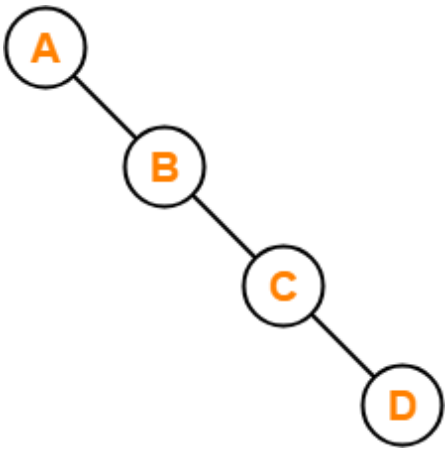
1. Left Skewed Binary Tree : If the right subtree is missing in every node of a tree is called Left Skewed Binary Tree.

Example :



2. Right Skewed Binary Tree : If the left subtree is missing in every node of a tree is called Right Skewed Binary Tree.

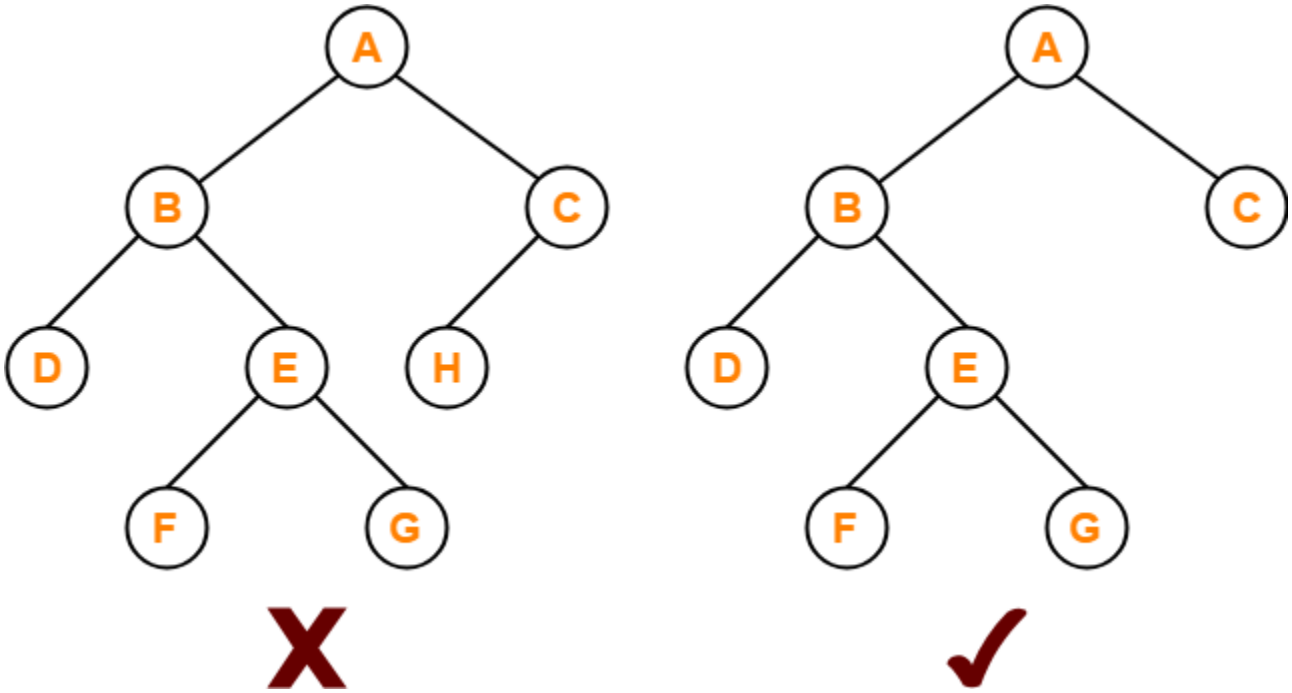
Example :



3. Full Binary Tree :

- A binary tree in which every node has either 0 or 2 children is called as a Full binary tree.
- Full Binary tree is also called as Strictly Binary tree.

Example :



Here,

- First binary tree is not a full binary tree.
- This is because node C has only 1 child.

4. Complete Binary Tree

A complete binary tree is a binary tree that satisfies the following 2 properties-

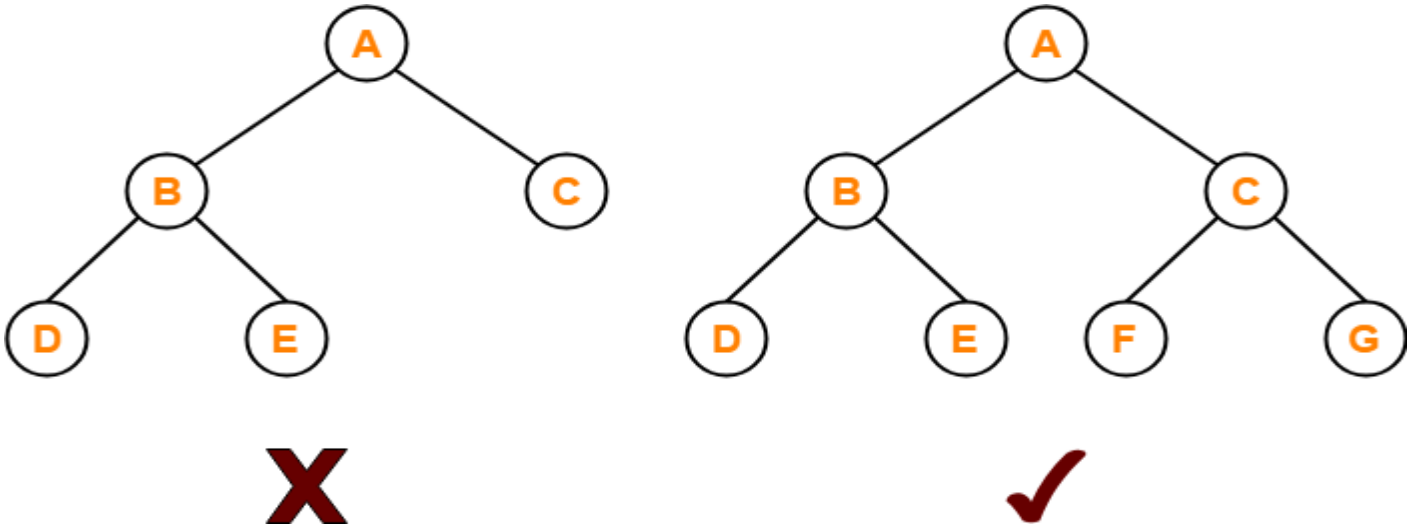
- Every internal node has exactly 2 children.
- All the leaf nodes are at the same level.
- Complete binary tree is also called as Perfect binary tree.
- In Complete binary tree there must be 2^{level} number of nodes.

For example :

At level 2 there must be $2^2 = 4$ nodes

At level 3 there must be $2^3 = 8$ nodes.

Example :



Here,

- First binary tree is not a complete binary tree.
- This is because all the leaf nodes are not at the same level.

Binary Tree Representations

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

1. Array Representation
2. Linked List Representation

1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

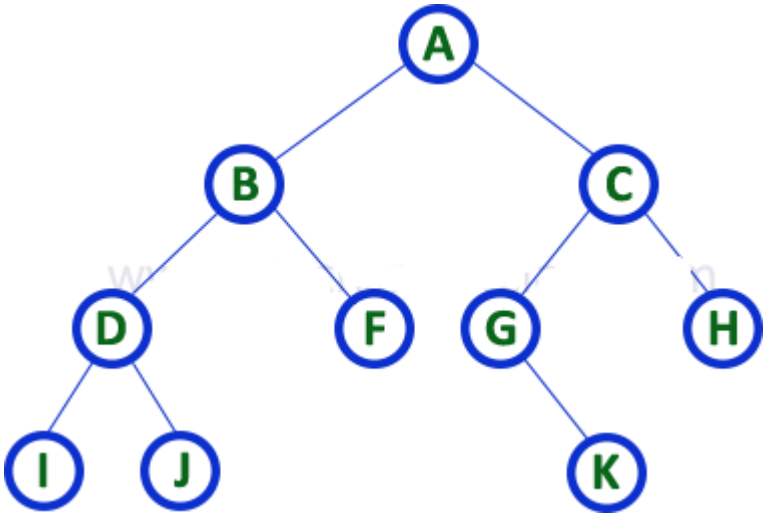
Array Representation of Binary tree following 3 rules

1 : Consider the **Root node** at Index = 0

2 : **Left child** = 2i + 1 , Where " **i** " is the position of the parent

3 : **Right child** = 2i + 2 , Where " **i** " is the position of the parent

Example :



Solution :

The above Binary tree of Height = 3

Maximum number of nodes in a binary tree of height H = $2^{3+1} - 1 = 2^4 - 1 = 15$

Position of **A** = 0

Position of **B** = 2i + 1 = 2.0 + 1 = 1

Position of **C** = 2i + 2 = 2.0 + 2 = 2

Position of **D** = 2i + 1 = 2.1 + 1 = 3

Position of **F** = 2i + 2 = 2.1 + 2 = 4

Position of **G** = 2i + 1 = 2.2 + 1 = 5

Position of **H** = 2i + 2 = 2.2 + 2 = 6

Position of **I** = 2i + 1 = 2.3 + 1 = 7

Position of **J** = 2i + 2= 2.3 + 2 = 8

Position of **K** = 2i + 2 = 2.5 + 2 = 12

Binary tree represented using Array representation is

Index	Nodes
0	A
1	B
2	C
3	D
4	F
5	G
6	H
7	I
8	J
9	-
10	-
11	-
12	K
13	-
14	-

Advantages : The direct access to any node can be possible in finding the root node or right and left child of any particular node is fast because of the random access.

Disadvantages : This type of representation is wastage of memory

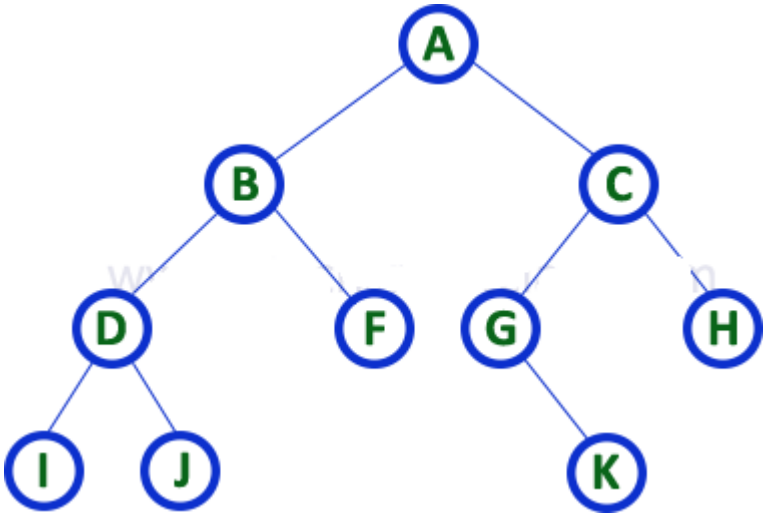
2. Linked List Representation of Binary Tree

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

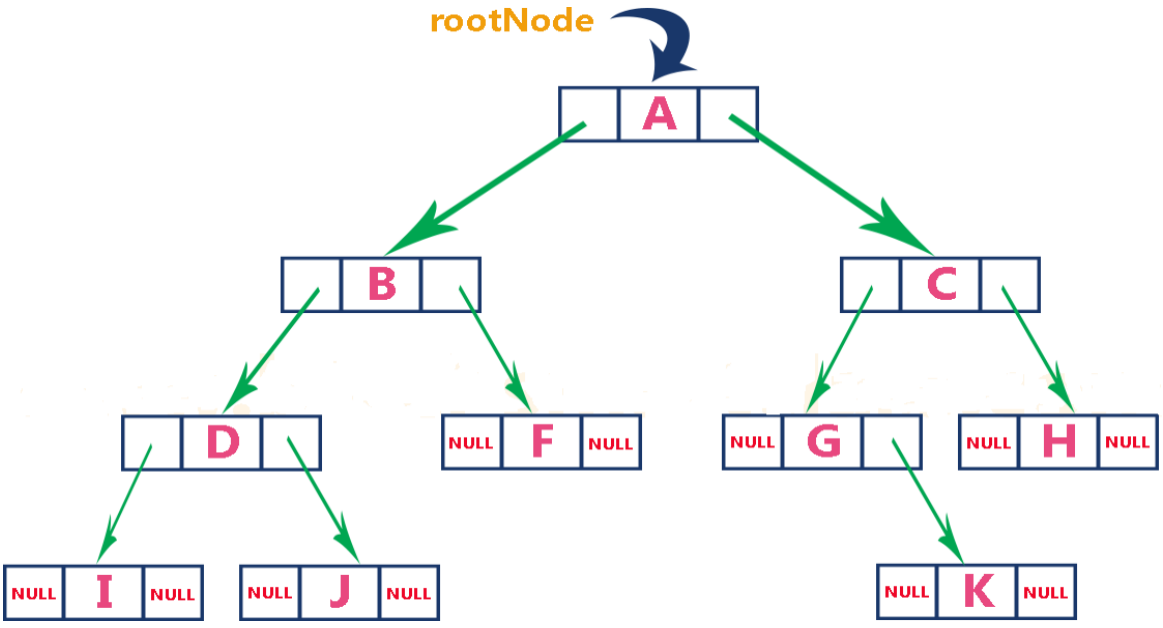
In this linked list representation, a node has the following structure...

Left Child Address	Data	Right Child Address

Example :



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



Binary Tree Traversal : Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.

Various Binary Tree traversal techniques are-

- Preorder Traversal
- Inorder Traversal
- Postorder Traversal

1. Preorder Traversal:

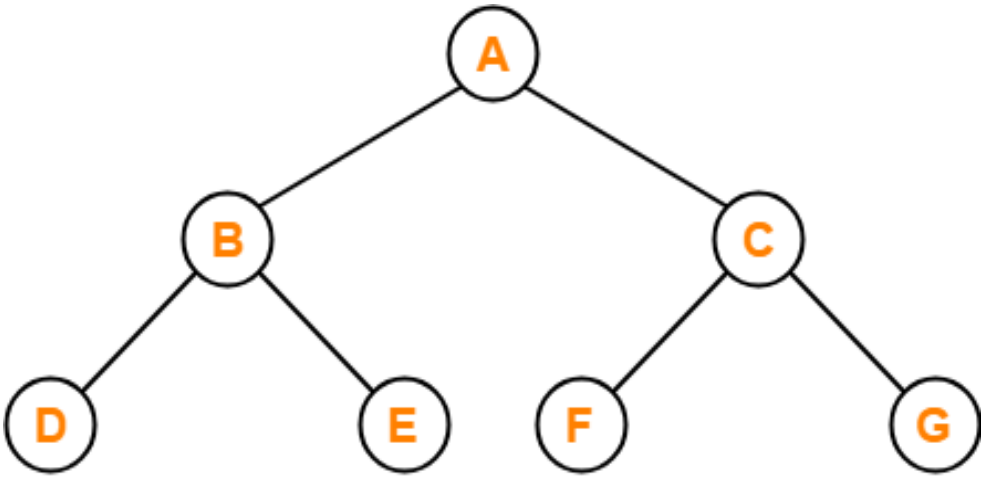
Algorithm-

1. Visit the root
2. Traverse the left sub tree i.e. call Preorder (left sub tree)
3. Traverse the right sub tree i.e. call Preorder (right sub tree)

Root → Left → Right

Example-

Consider the following example-



Preorder Traversal : A , B , D , E , C , F , G

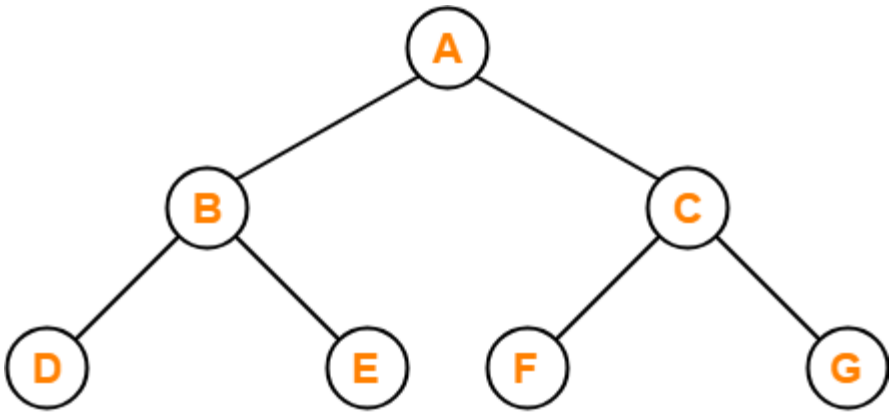
2. Inorder Traversal:

Algorithm-

1. Traverse the left sub tree i.e. call Inorder (left sub tree)
2. Visit the root
3. Traverse the right sub tree i.e. call Inorder (right sub tree)

Left → Root → Right

Example: Consider the following example-



Inorder Traversal : D , B , E , A , F , C , G

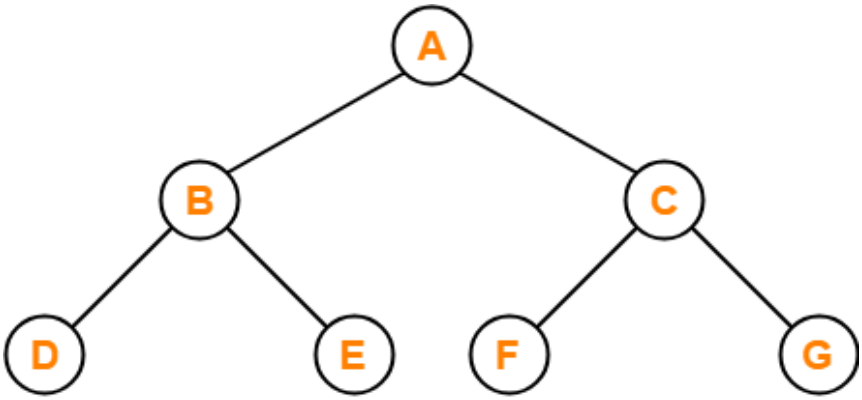
3. Postorder Traversal:

Algorithm-

1. Traverse the left sub tree i.e. call Postorder (left sub tree)
2. Traverse the right sub tree i.e. call Postorder (right sub tree)
3. Visit the root

Left → Right → Root

Example: Consider the following example-



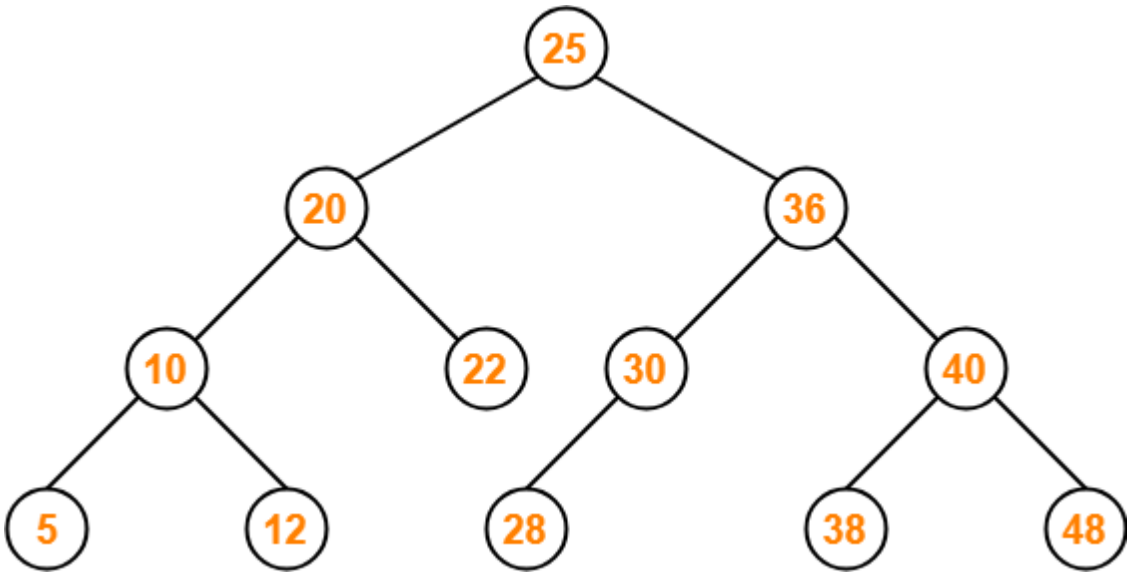
Postorder Traversal : D , E , B , F , G , C , A

Binary Search Tree (BST)

Definition of Binary Search Tree (BST)

- Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
- In a binary search tree, the value of all the nodes in the left sub-tree is less than the value ofthe root.
- Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
- This rule will be recursively applied to all the left and right sub-trees of the root.

Example :



Note: Every binary search tree is a binary tree but every binary tree need not to be binary search tree.

Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

- Insertion
- Deletion
- Searching

1.Insertion Operation in BST

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node.

The insertion operation is performed as follows...

Step 1 - Create a newNode 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** to **newNode**.

Step 4 - If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).

Step 5 - If newNode is **smaller** than **or equal** to the node then move to its **left** child.

Step 6 - If newNode is **larger** than the node then move to its **right** child.

Step 7- Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).

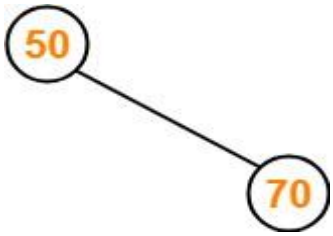
Step 8 - After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller or equal** to that leaf node or else insert it as **right child**.

Example: Construct a Binary Search Tree (BST) for the following sequence of numbers-

50, 70, 60, 20, 90, 10, 40, 100

Step1: Insert 50

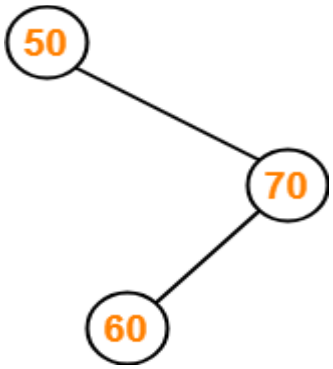
Step2: Insert 70, As 70 > 50, so insert 70 to the right of 50.



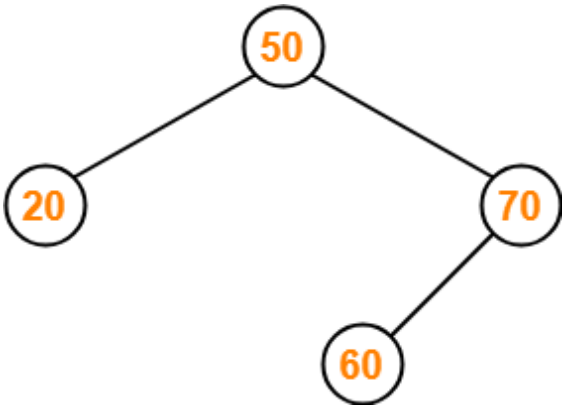
Step3: Insert 60,

As 60 > 50, so insert 60 to the right of 50.

As 60 < 70, so insert 60 to the left of 70.

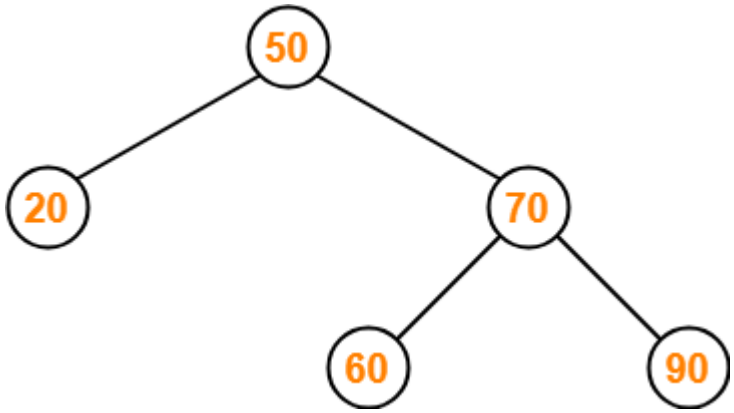


Step4: Insert 20, As 20 < 50, so insert 20 to the left of 50.



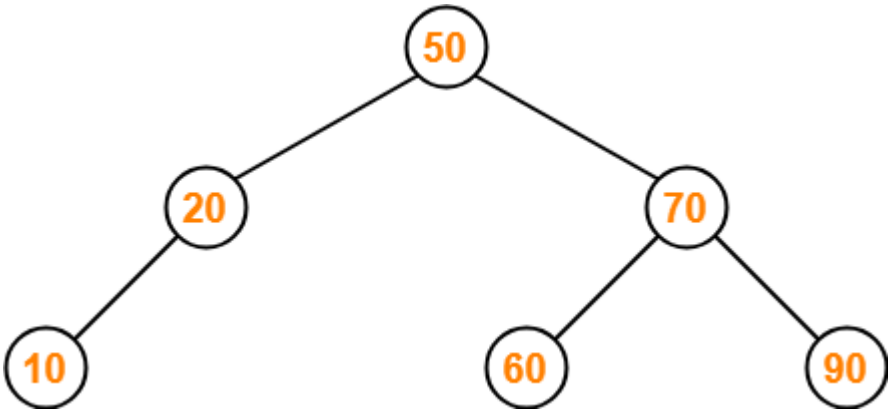
Step5: Insert 90,

- As 90 > 50, so insert 90 to the right of 50.
- As 90 > 70, so insert 90 to the right of 70.



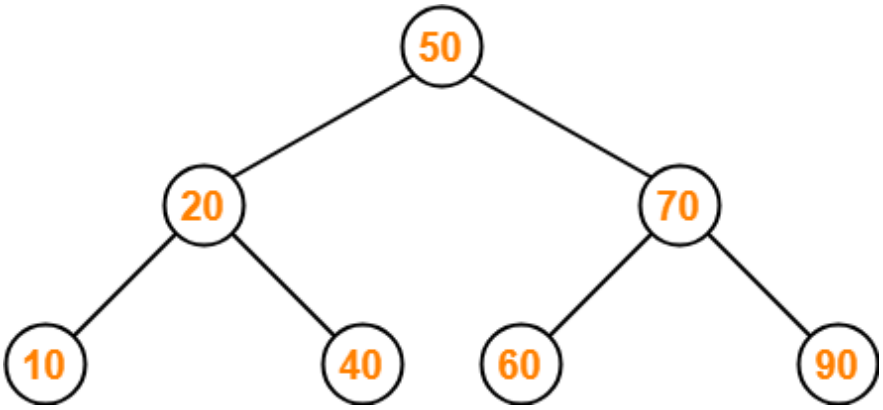
Step6: Insert 10,

- As 10 < 50, so insert 10 to the left of 50.
- As 10 < 20, so insert 10 to the left of 20.



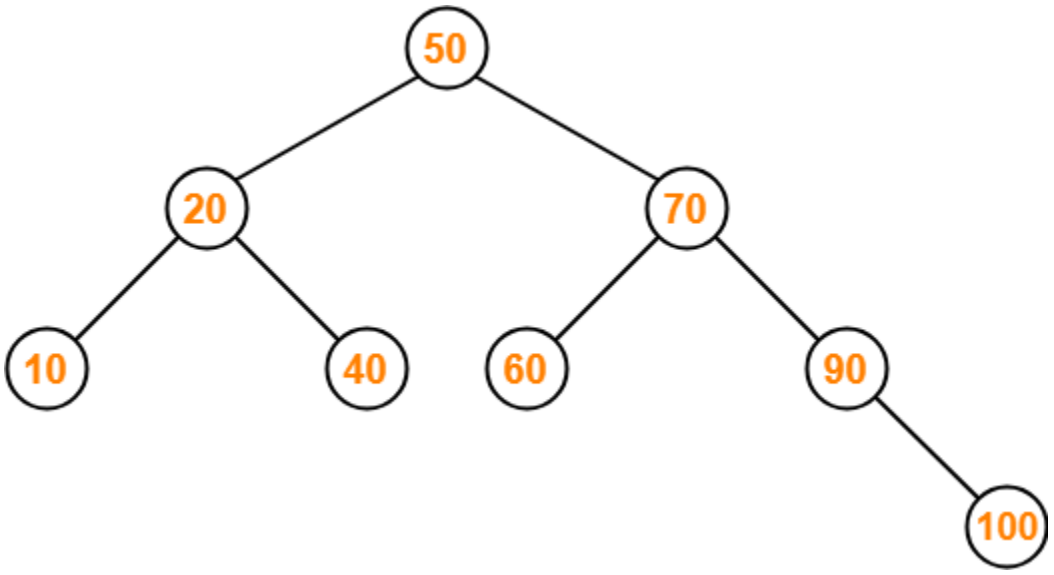
Step7: Insert 40,

- As 40 < 50, so insert 40 to the left of 50.
- As 40 > 20, so insert 40 to the right of 20.

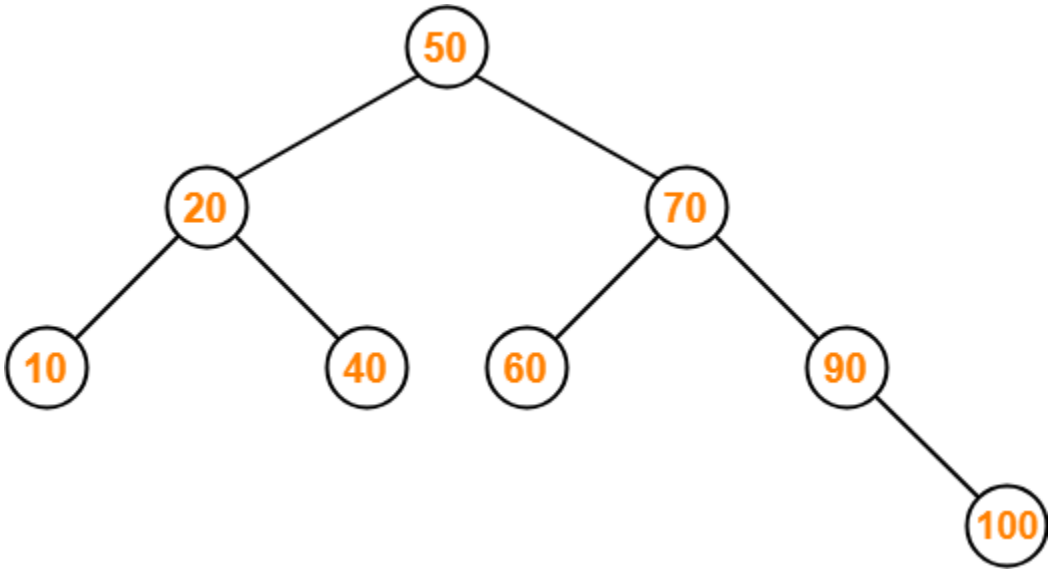


Step8: Insert 100,

- As 100 > 50, so insert 100 to the right of 50.
- As 100 > 70, so insert 100 to the right of 70.
- As 100 > 90, so insert 100 to the right of 90.



Therefore, Finally the Binary Search Tree is



2. Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O (log n)** time complexity.

Deleting a node from Binary search tree includes following three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

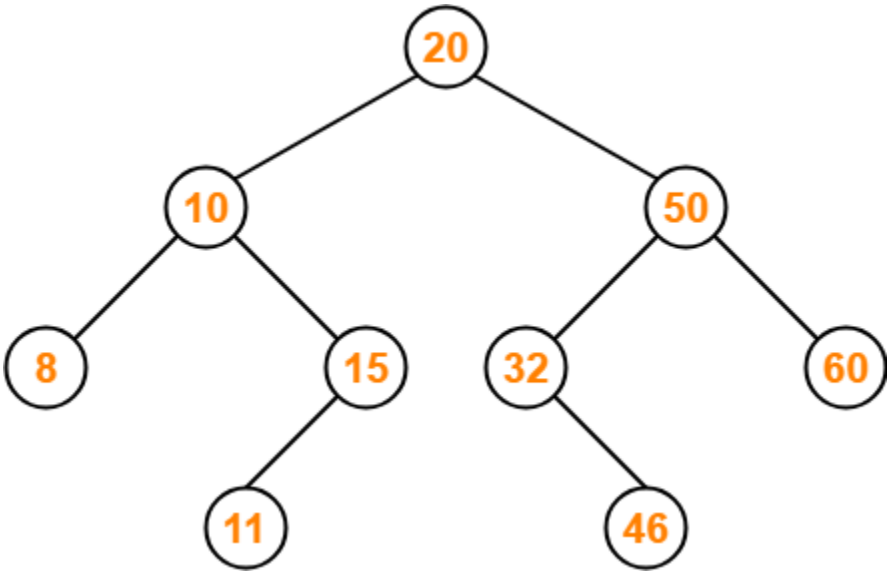
Case 1: Deleting a Leaf node (A node with no children)

We use the following steps to delete a leaf node from BST...

Step 1 - **Find** the node to be deleted using **search operation**

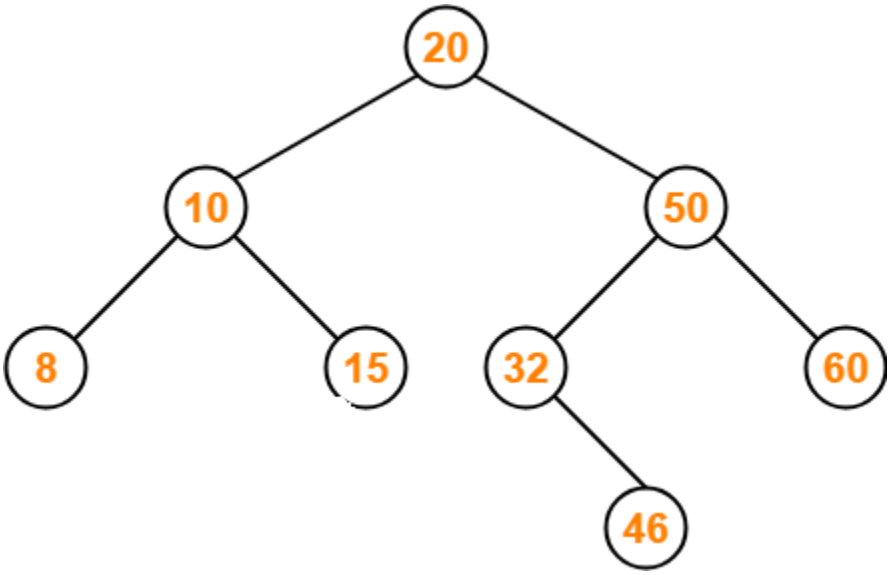
Step 2 - It is the simplest case, in this case, replace the leaf node with the NULL and Delete the node using **free** function (If it is a leaf) and terminate the function.

Example:



If we want to delete node 11,

It is the simplest case, in this case, replace the leaf node with the NULL and Delete the node 11



Case 2: Deleting a node with one child

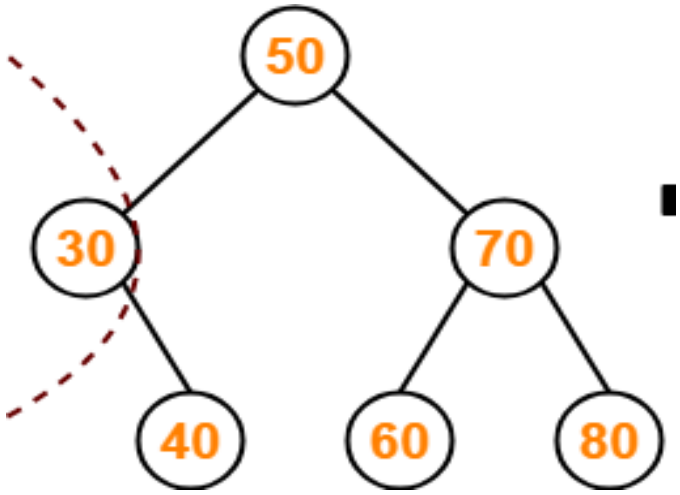
We use the following steps to delete a node with one child from BST...

Step 1 - **Find** the node to be deleted using **search operation**

Step 2 - If it has only one child then create a link between its parent node and child node.

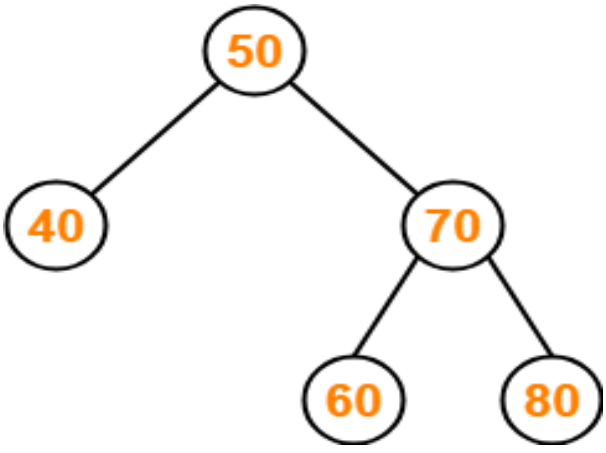
Step 3 - The node will be replaced with its child node. Delete the node using **free** function and terminate the function.

Example:



If we want to delete node 30,

It has only one child. The node will be replaced with its child node. Delete the node using **free** function and terminate the function.



Case 3: Deleting a node with two children

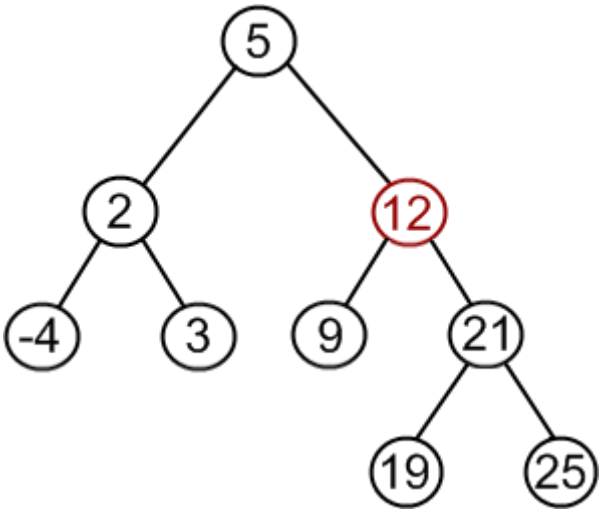
Step 1 - Find the node to be deleted using search operation

Step 2 - if the node has two children, find the inorder successor of the node.

Step 3 -Copy the contents of the inorder successor to the node to be deleted. Note, that only values are replaced, not nodes. Now we have two nodes with the same value.

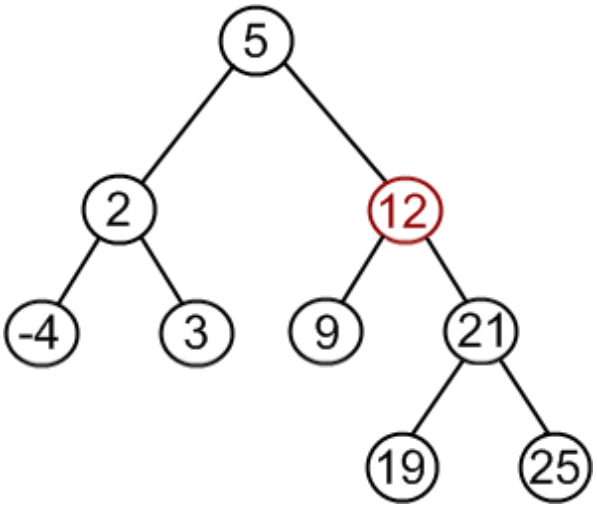
Step 4 – Finally delete the inorder successor.

Example 1 :

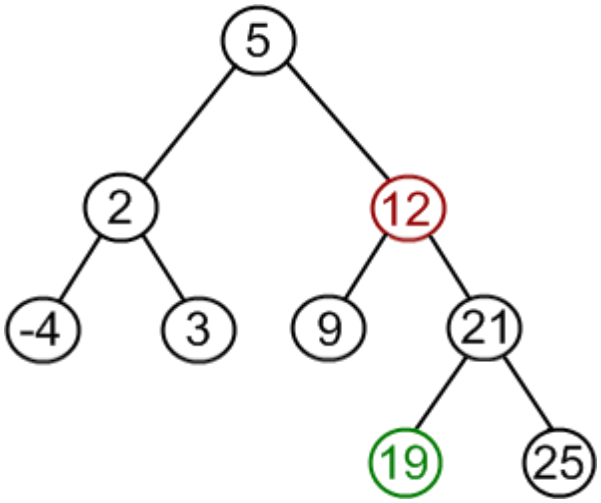


Delete **node 12** in the following **BST**

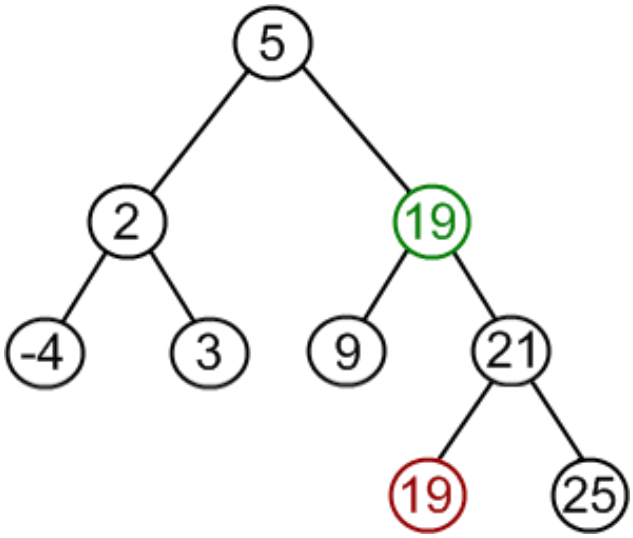
Step1 : First, we must find the node with the value 12



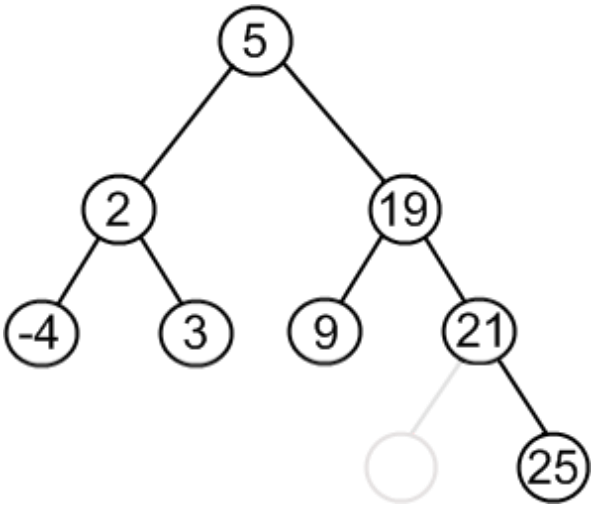
Step2 : ***Next***, we **find** the ***successor*** **node** of the node **12**. The ***successor*** **node** is the **node** with the ***minimum*** **value** in the ***right*** **subtree**, i.e, node 19.



Step 3 : Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.



Step 4 : The **last step** is **deleting** the node 19 from the left sub tree.



3.Search Operation in Binary Search Tree (BST)

In an Binary Search tree (BST), the search operation is performed with **O (log n)** time complexity.

We use the following steps to search an element in Binary Search Tree (BST).

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.

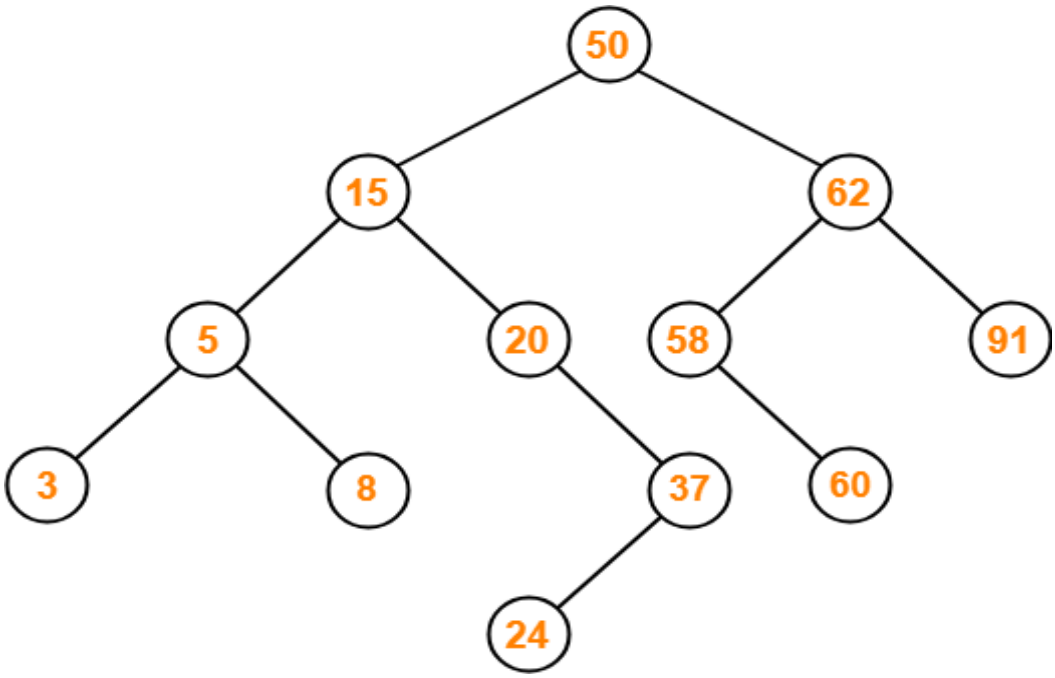
Step 6 - If search element is larger, then continue the search process in right sub tree.

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.

Example:



If we want search the element 24.

The following steps to search an element 24 in AVL tree...

1. Read the search element is 24
2. Compare the search element (24) with the value of root node (50) in the tree. Since 24 is Less than 50, then the search process in left sub tree.
3. Compare the search element (24) with the value (15) in the left sub tree. Since 24 is greater than 15, then the search process in right sub tree.
4. Compare the search element (24) with the value (20) in the right sub tree. Since 24 is greater than 20, then the search process in right sub tree.
5. Compare the search element (24) with the value (37) in the right sub tree. Since 24 is less than 37, then the search process in left sub tree.
6. Compare the search element (24) with the value (24) in the left sub tree. Since 24 is equal to 24, Then display "Element is Found" and terminate the function.

AVL TREE

Definition of AVL Tree:

- The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.
- 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.
- 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.

Definition of Balance factor :

Balance factor (BF) = Height of Left Sub tree – Height of Right Sub tree

In AVL tree, Balance factor (BF) of every node is either 0 or 1 or -1.

Where,

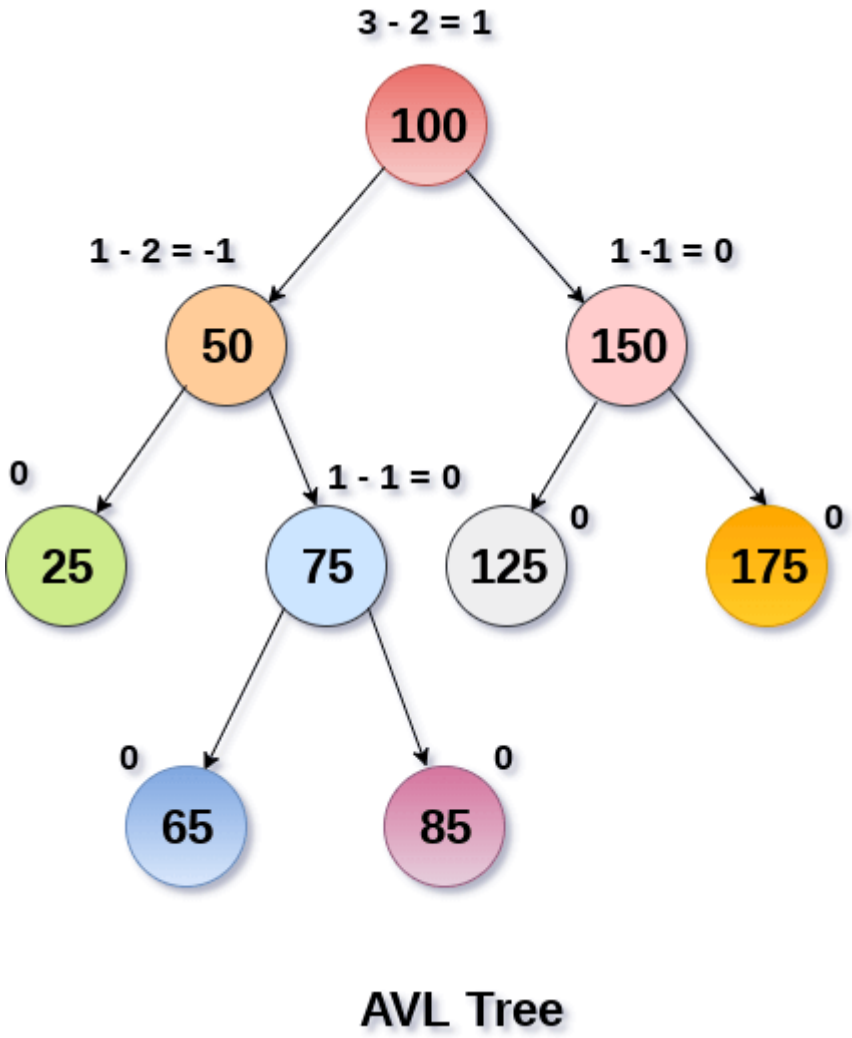
1: BF = 0, if the tree is perfectly balanced, that is, both the left sub-tree and right Sub-tree of the same Height.

2: BF = +1, If the height of the left sub-tree is one level higher than the height of the right sub- tree.

3: BF = -1, If the height of the left sub-tree is one level lower than the height of the right sub-tree.

4: BF <= -2 and BF >= +2, indicates the tree is said to be unbalanced tree as there is an Unacceptable difference between the heights of its left sub-tree and right sub-tree.

Example of AVL Tree:



AVL Tree

The above tree is a binary search tree and every node is satisfying balance factor condition.

So this tree is said to be an AVL tree.

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

AVL Tree Rotations:

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

Rotation: Rotation means to convert an unbalanced tree into a balanced tree is known as Rotation.

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

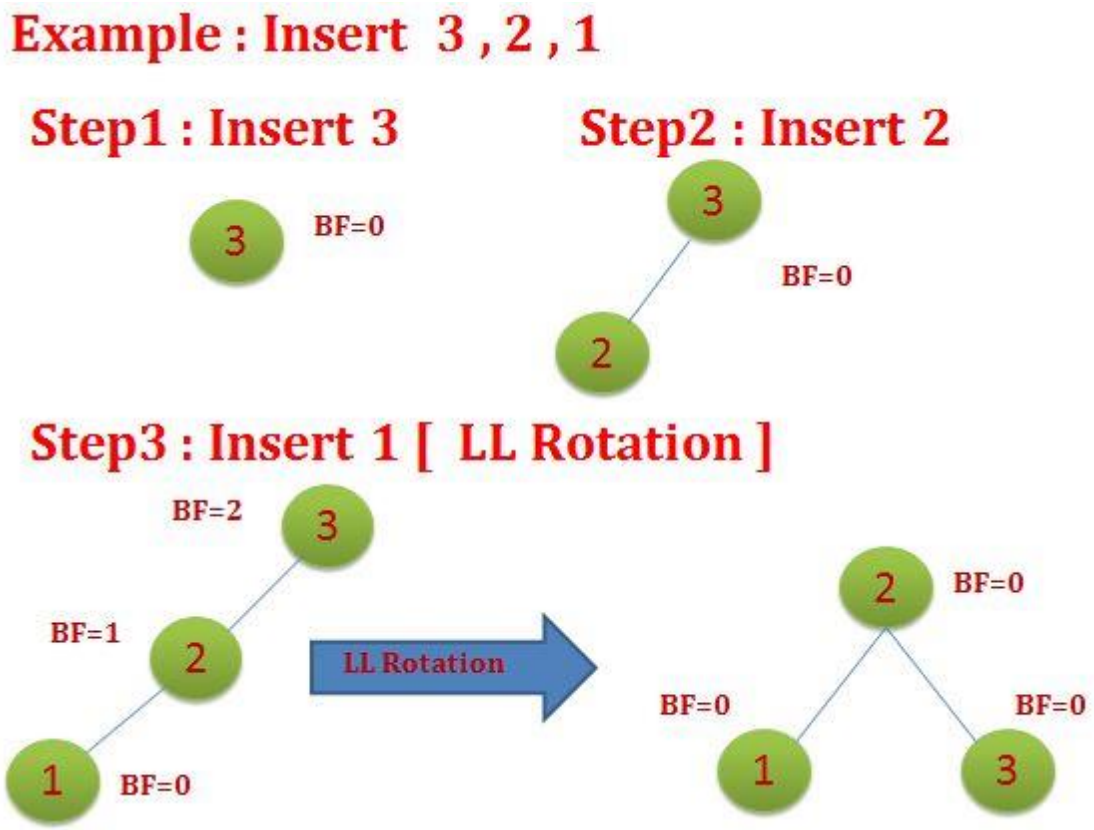
1. Single Rotations

- Left Rotation (LL Rotation)
- Right Rotation (RR Rotation)

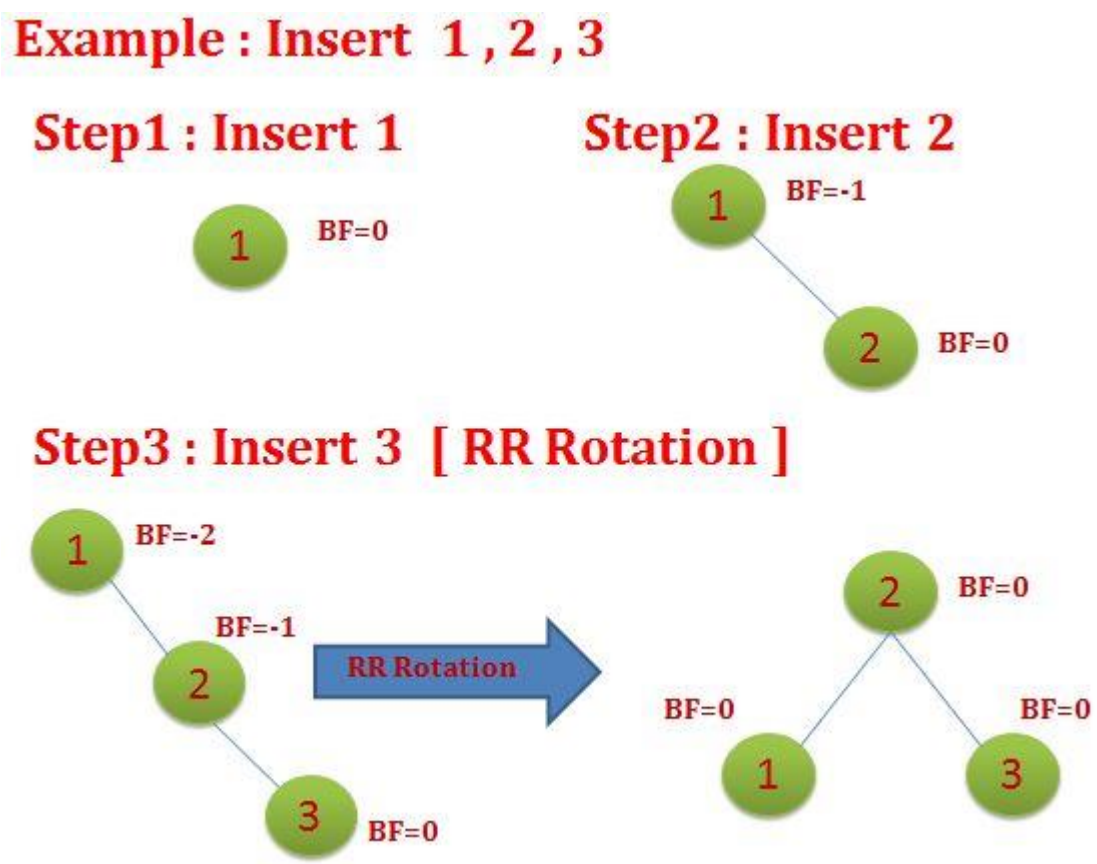
2. Double Rotations

- Left-Right Rotation (LR Rotation)
- Right-Left Rotation (RL Rotation)

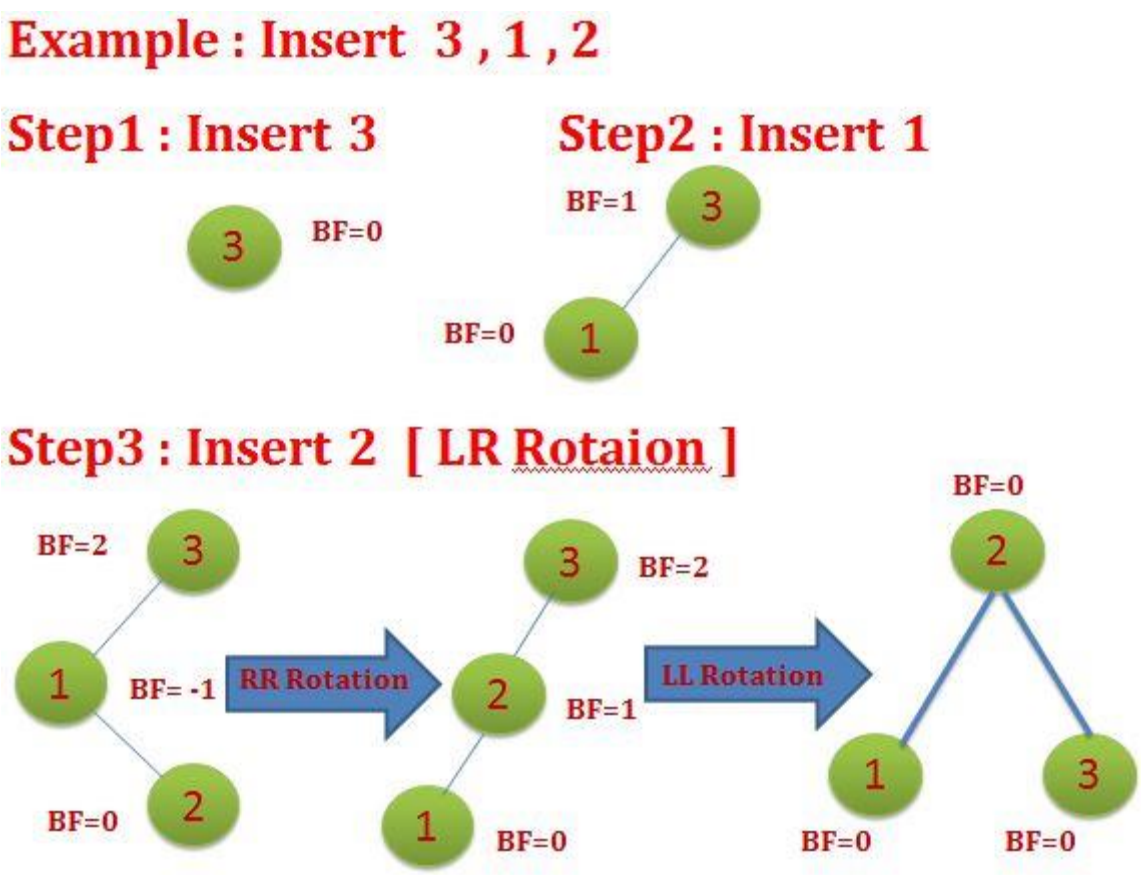
1. Left Rotation (LL Rotation): Left Rotation (LL Rotation) is a single rotation. If a tree becomes unbalanced, when a node is inserted into the left sub tree of the left sub tree, then we perform LL Rotation, LL Rotation is clockwise direction, which is applied on the edge below a node having balance factor 2.



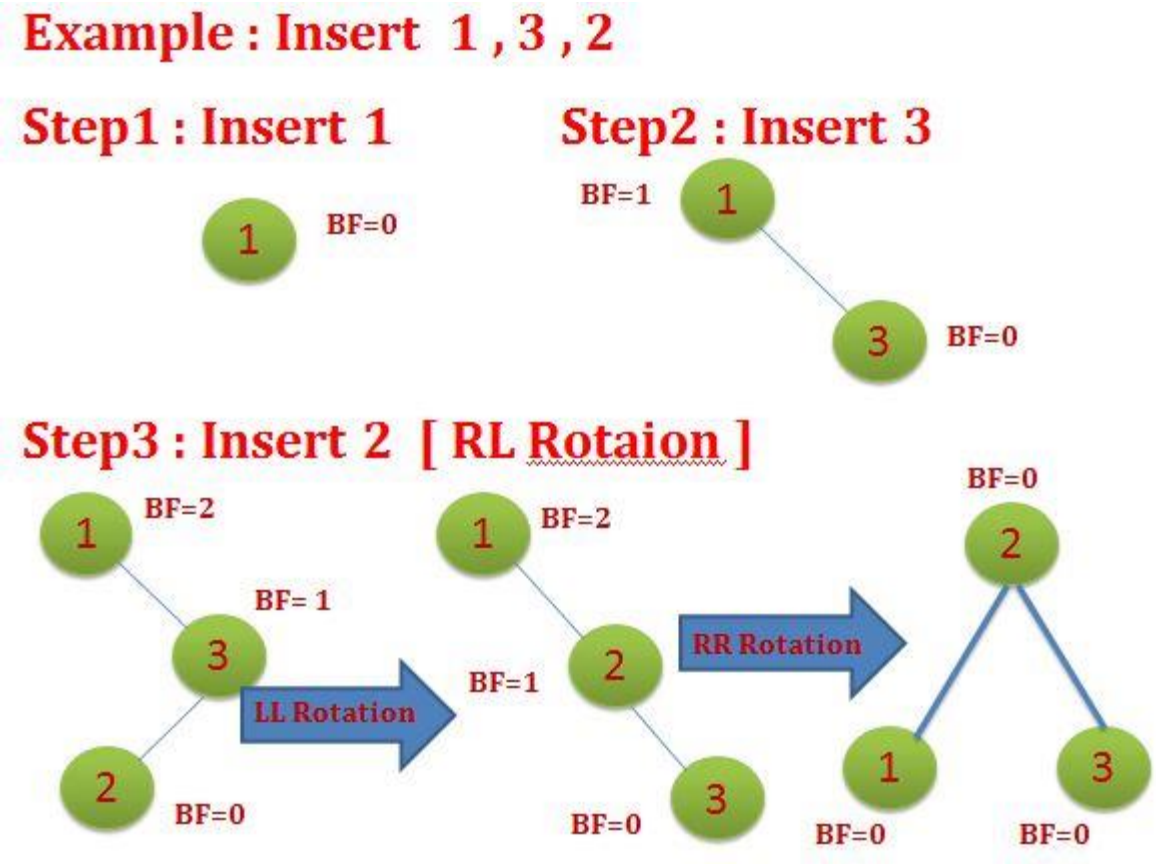
2. Right Rotation (RR Rotation): Right Rotation (RR Rotation) is a single rotation. If a tree becomes Unbalanced, when a node is inserted into the right sub tree of the right sub tree, then we perform RR Rotation, RR Rotation is an anti-clockwise direction, which is applied on the edge below a node having balance factor -2.



3. Left-Right Rotation (LR Rotation): Left-Right Rotation (LR Rotation) is a double rotation. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on sub tree and then LL rotation is performed on full tree.



4. Right-Left Rotation (RL Rotation): Right-Left Rotation (RL Rotation) is a double rotation. RL rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on sub tree and then RR rotation is performed on full tree.



Operations of AVL Tree :

1 : Insertion

2 : Deletion

3 : Searching

Insertion Operation in AVL Tree

In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2 - After insertion, check the Balance Factor of every node.

Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

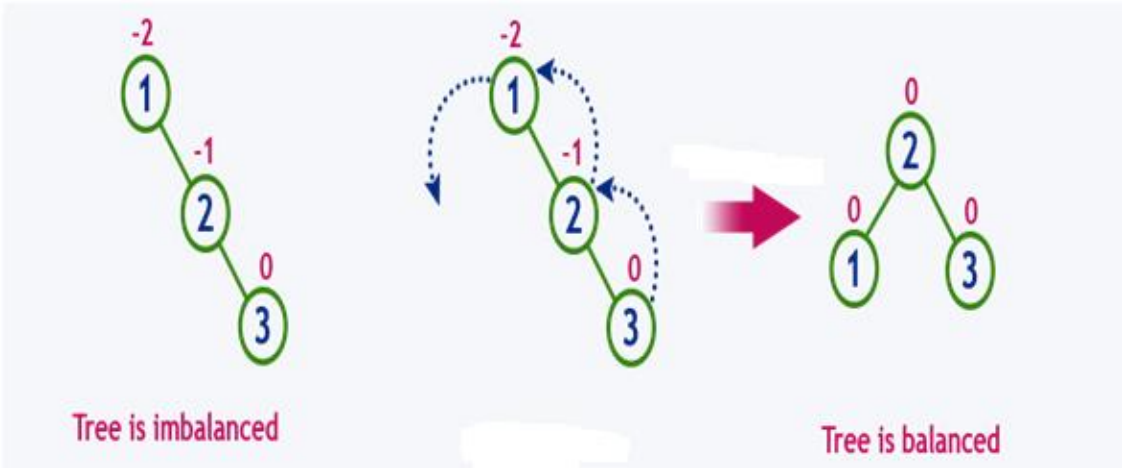
Example: Construct AVL Tree for the following sequence of numbers-

1, 2, 3, 4, 5, 6, 7, 8

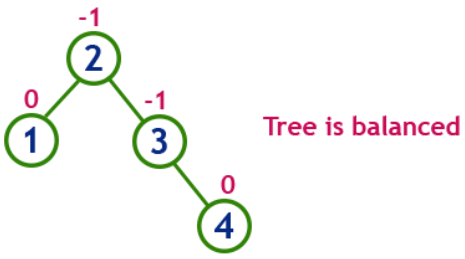
Step 1: Insert 1

Step 2: Insert 2

Step 3: Insert 3

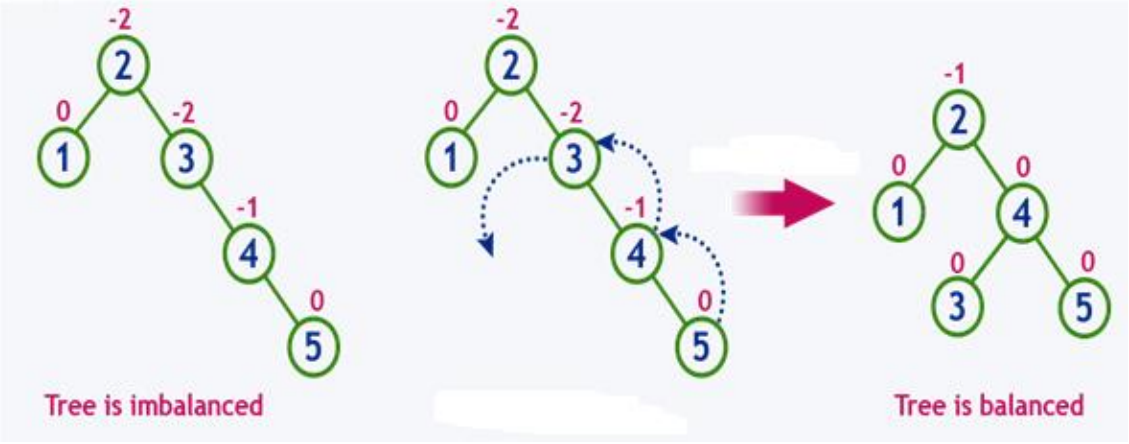


Step 4: Insert 4

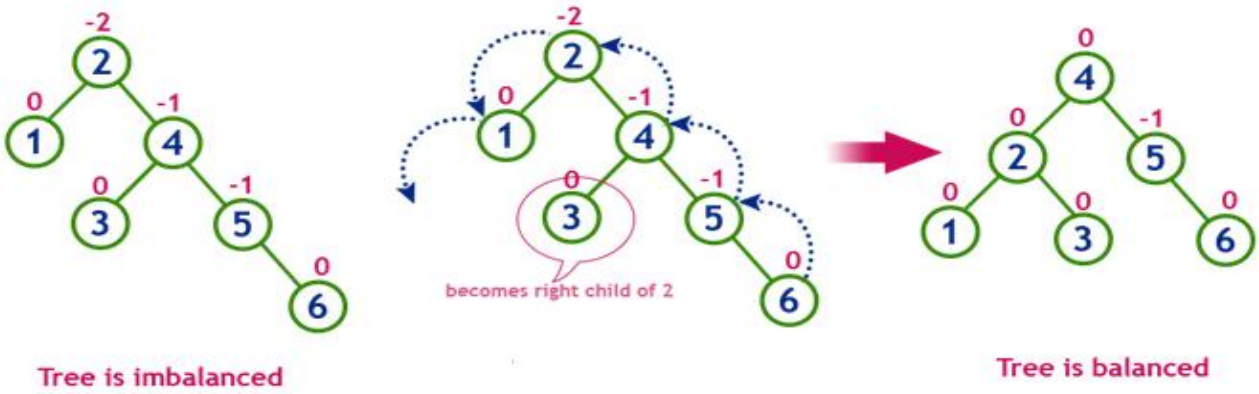


Tree is balanced

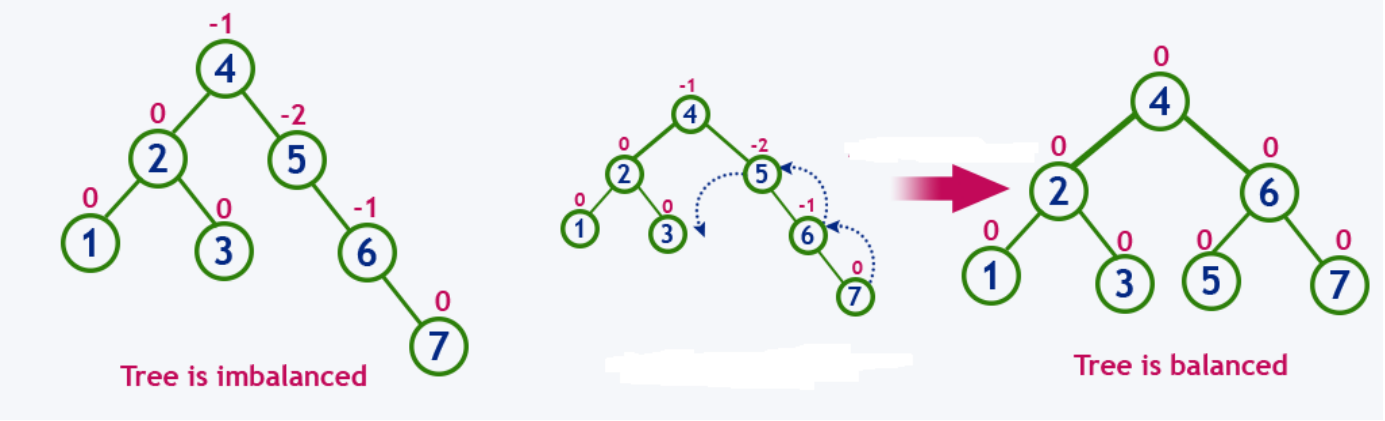
Step 5: Insert 5



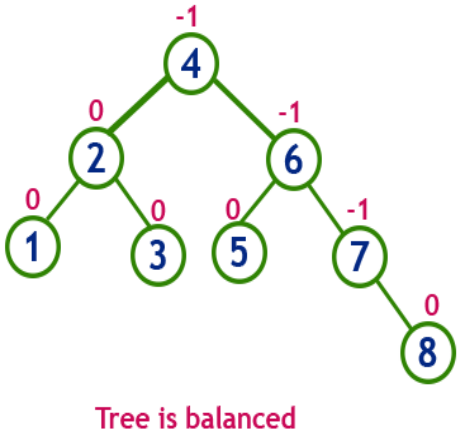
Step 6: Insert 6



Step 7: Insert 7



Step 8: Insert 8

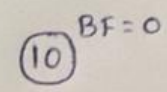


Example 1 Construct AVL Tree for the following list.

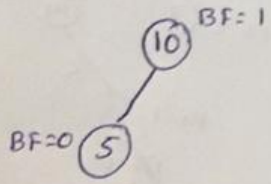
10, 5, 3, 2, 7, 14, 13, 11, 18, 1, 25, 28, 12

Solution

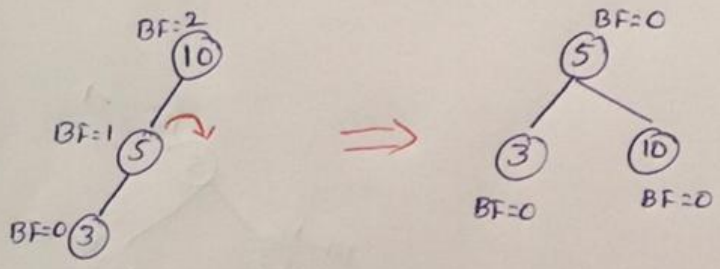
Step1 insert 10



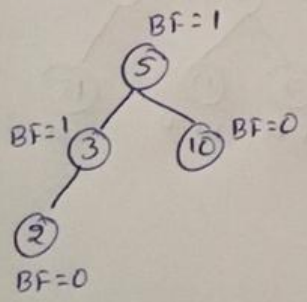
Step2 insert 5



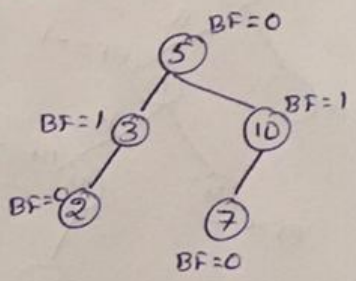
Step3 insert 3



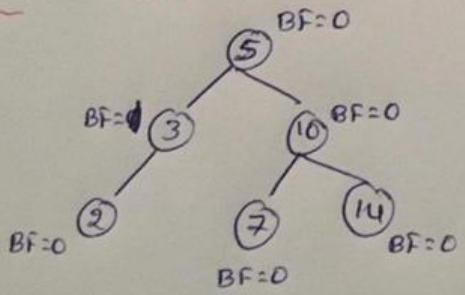
Step4 insert 2



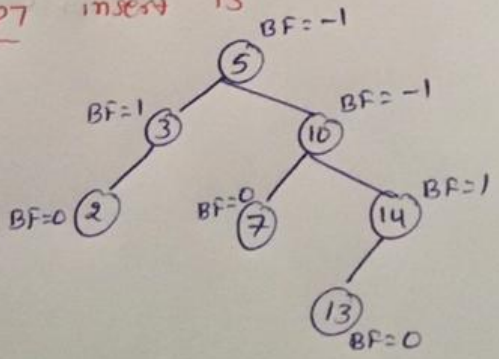
Step5 insert 7



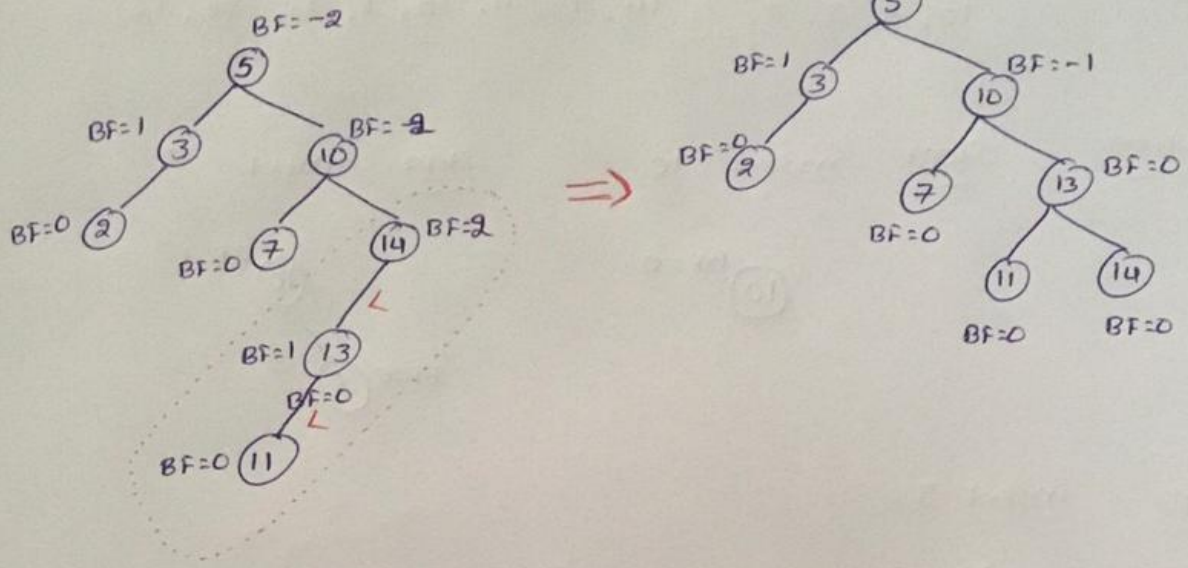
Step6 insert 14



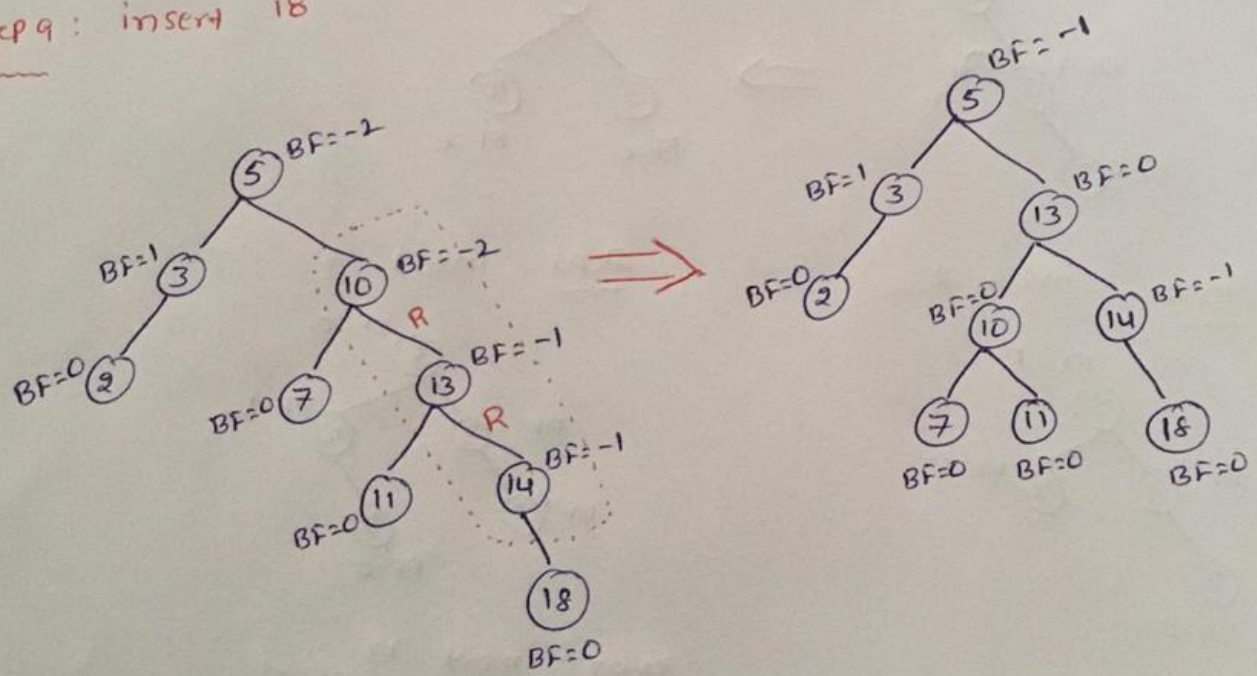
Step7 insert 13



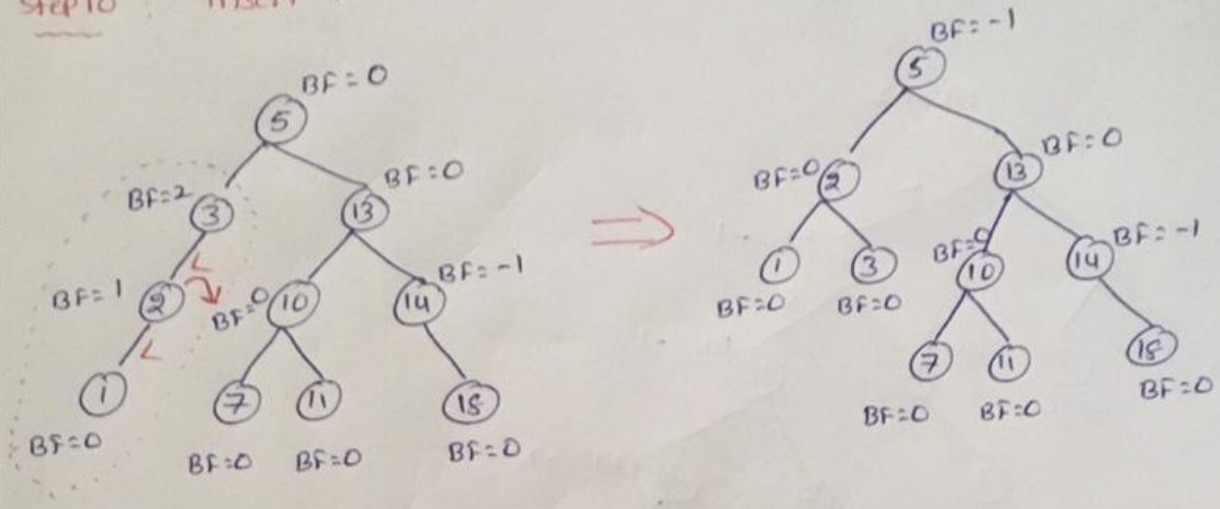
Step 8 : insert 11



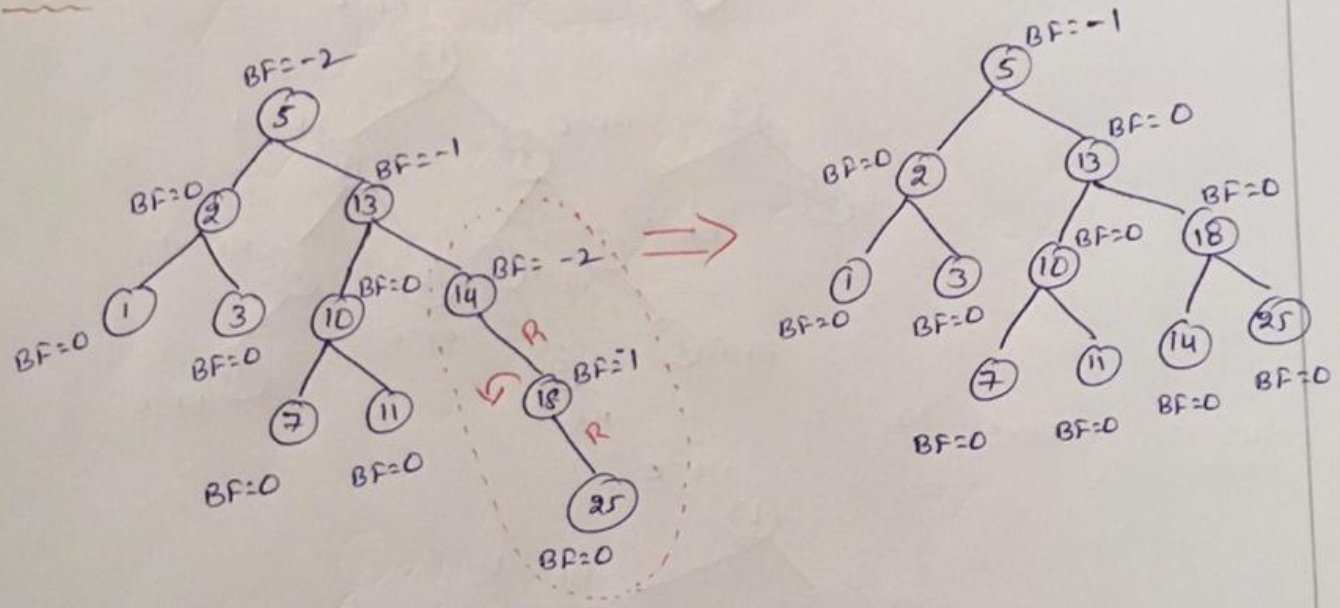
Step 9 : insert 18



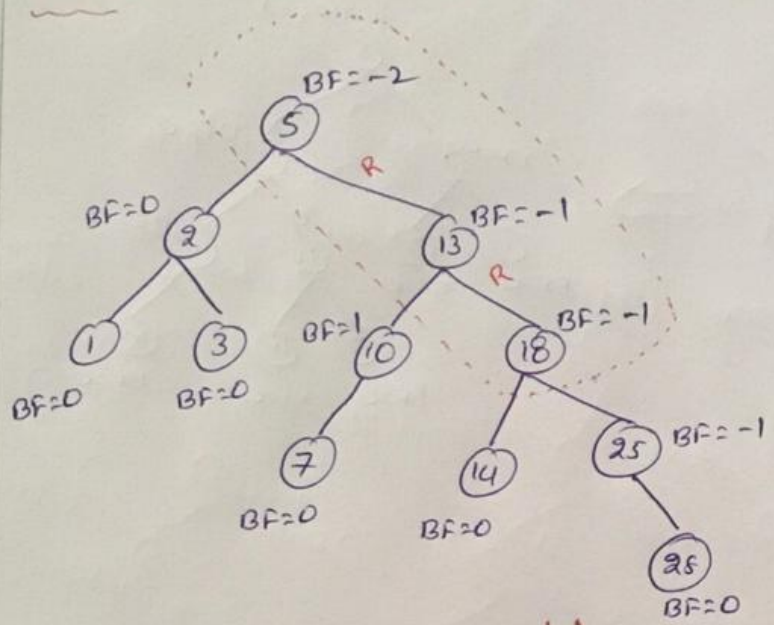
Step 10 : insert 1

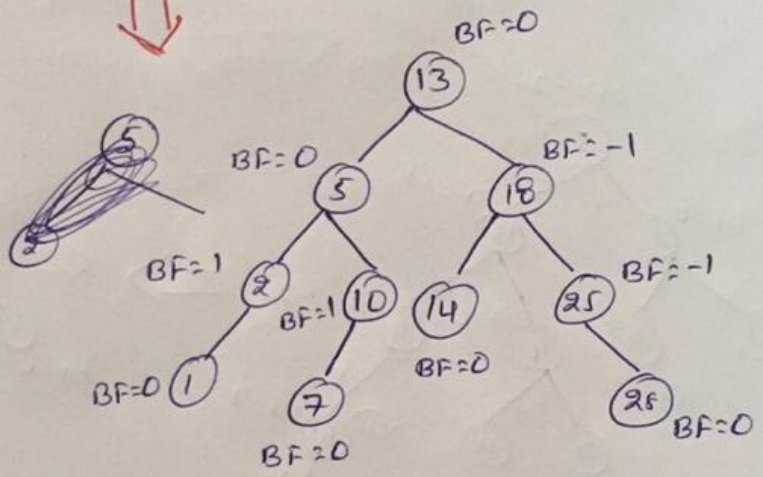


Step 11 : insert 25

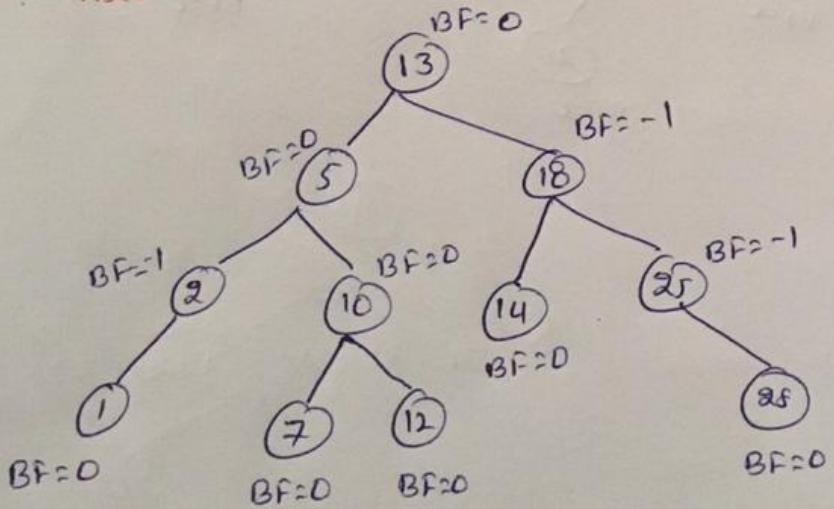


Step 12 : insert 28





Step 13 : insert 12

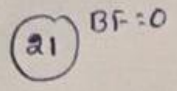


Example 2 : Construct AVL Tree for the following List.

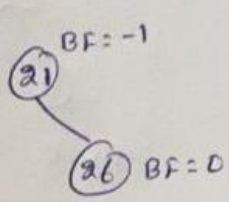
21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3.

Solution :

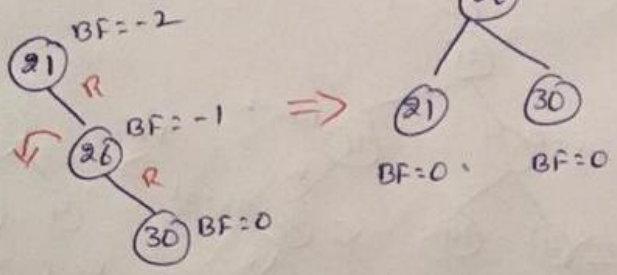
Step 1 : insert 21



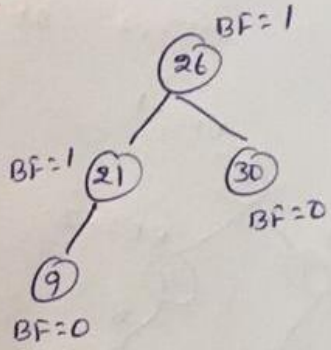
Step 2 : insert 26



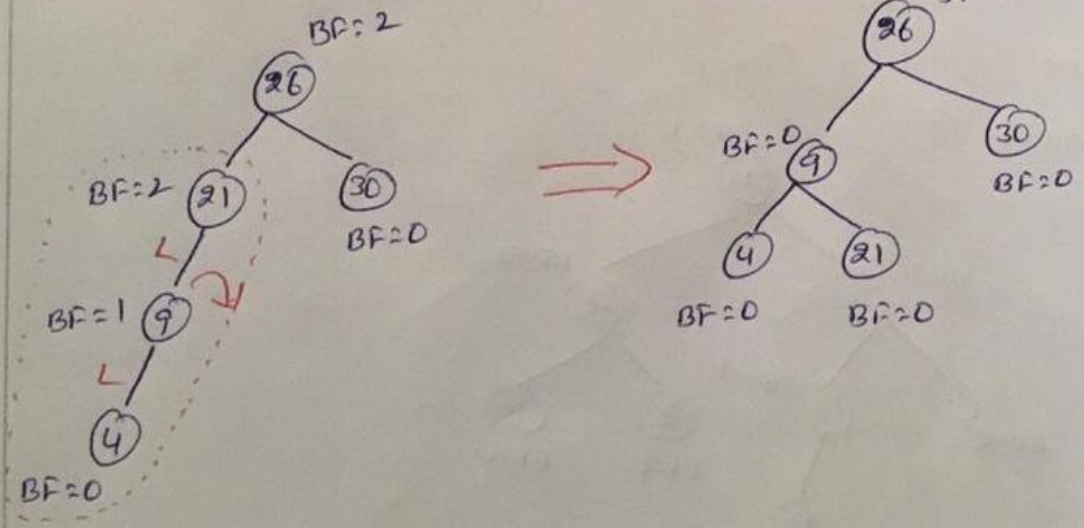
Step 3 : insert 30

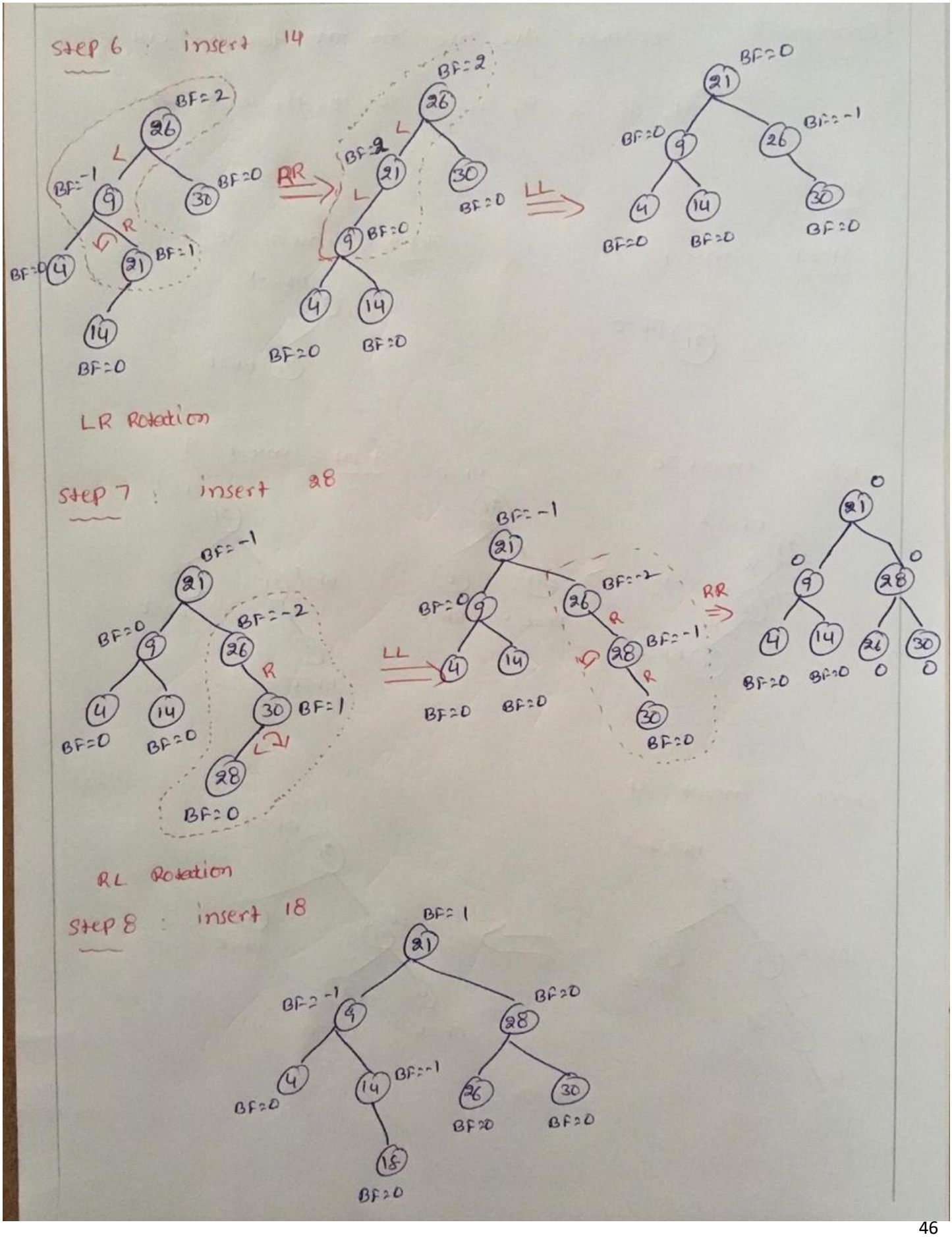


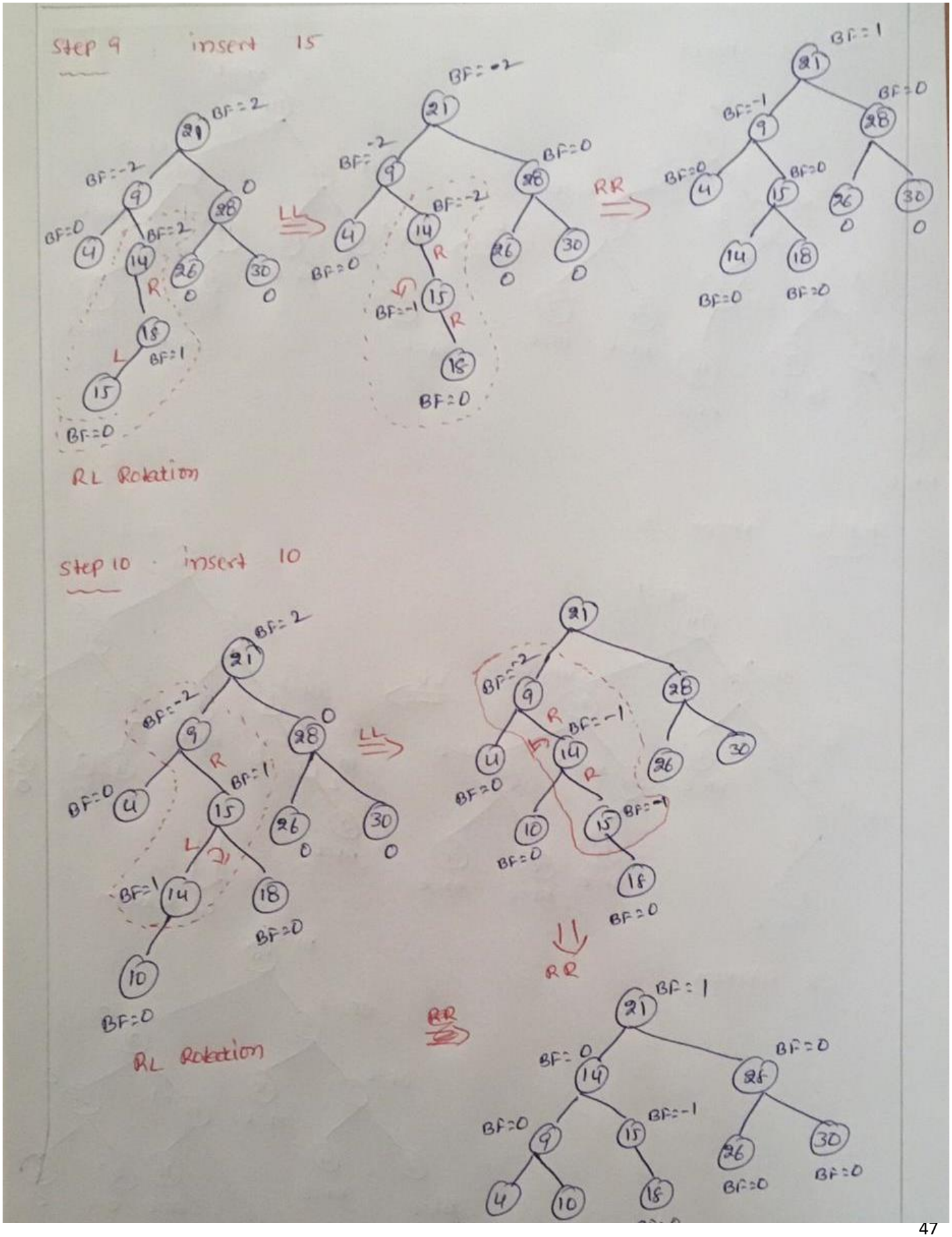
Step 4 : insert 9

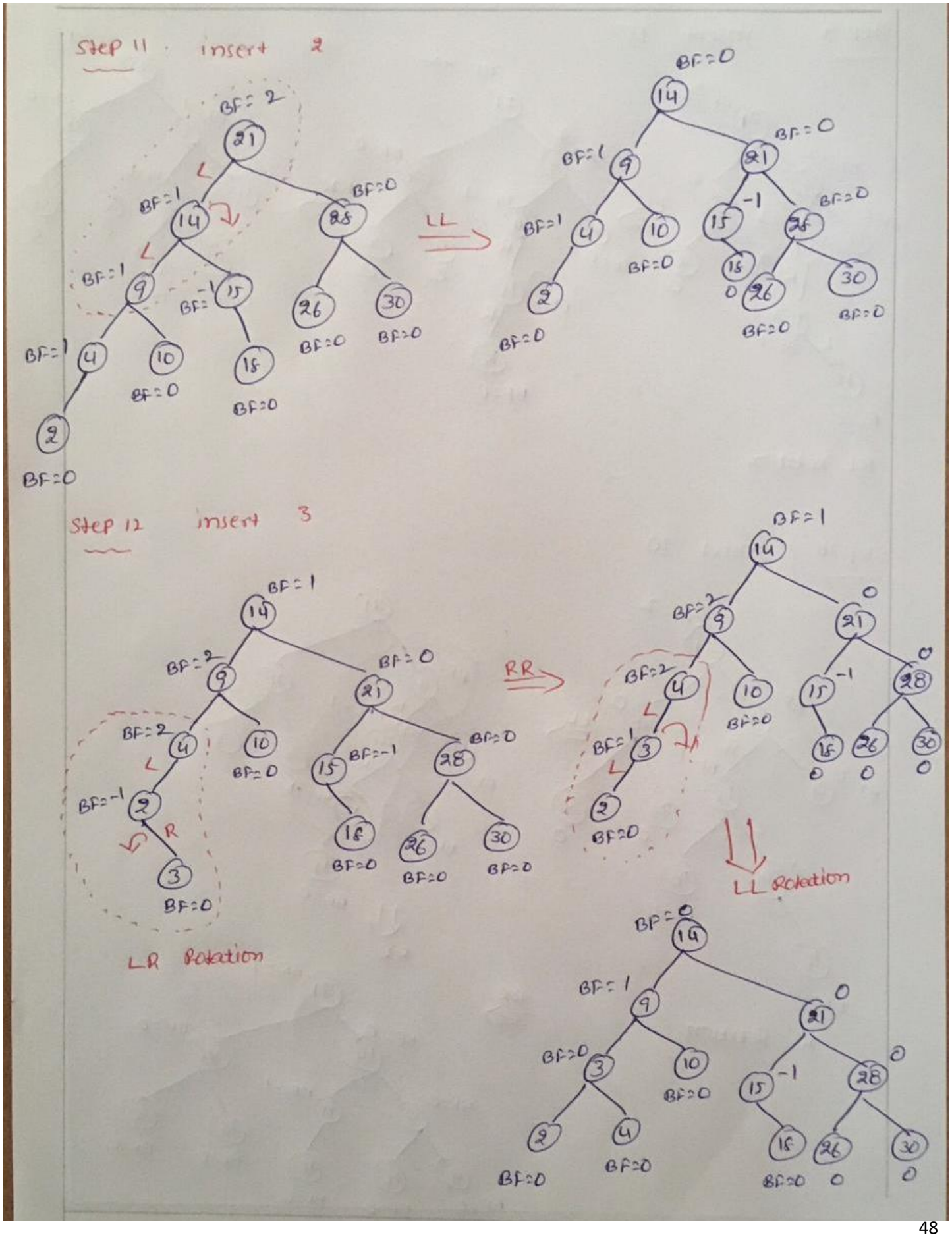


Step 5 : insert 4




Step 6 : insert 14
BF=2
26
L
BF=-1
9
30
BF=0
R
4
21
BF=1
14
BF=0
BF=0
RR
BF=2
26
L
BF=2
21
30
BF=0
L
9
BF=0
4
14
BF=0
BF=0
LL
BF=0
21
BF=0
9
BF=-1
26
4
14
30
BF=0
BF=0
BF=0
LR Rotation
Step 7 : insert 28
BF=-1
21
BF=0
9
BF=-2
26
4
14
30
BF=1
BF=0
BF=0
28
BF=0
BF=-1
21
BF=0
9
BF=-2
26
LL
4
14
R
28
BF=-1
BF=0
BF=0
R
30
BF=0
RR
0
21
0
0
9
28
4
14
24
30
BF=0
BF=0
0
0
RL Rotation
Step 8 : insert 18
BF=1
21
BF=-1
4
BF=0
28
4
14
BF=-1
26
30
BF=0
BF=0
BF=0
18
BF=0


Step 9 insert 15
LL
RR
RL Rotation
Step 10 insert 10
LL
RR
RL Rotation


Step 11 insert 2
BF=2
21
14
85
9
15
4
10
18
2
26
30
LL
14
9
21
4
10
15
28
2
18
26
30
Step 12 insert 3
RR
LR Rotation
LL Rotation
3

Deletion in an AVL Tree

- Deletion in an AVL tree is similar to that in a BST.
- Deletion of a node tends to disturb the *balance factor.* Thus to balance the tree, we again use the Rotation mechanism.

Deletion in AVL tree consists of two steps:

- **Removal of the node:** The given node is removed from the tree structure. The node to be removed can either be a leaf or an internal node.
- **Re-balancing of the tree:** The elimination of a node from the tree can cause disturbance to the balance factor of certain nodes. Thus it is important to re- balance *the nodes;* since the balance factor is the primary aspect that ensures the tree is an AVL Tree.

Deleting a node from Binary search tree includes following three cases...

Case 1: Deleting a Leaf node (A node with no children)

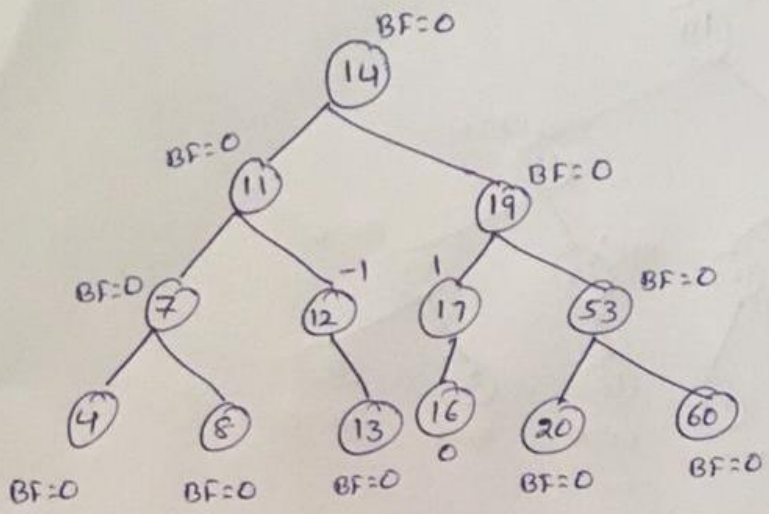
Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Note: There are certain points that must be kept in mind during a deletion process.

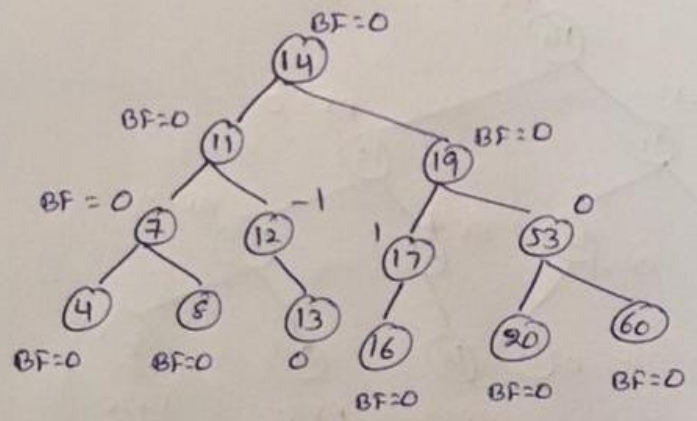
- If the node to be deleted is a leaf node, it is simply removed from the tree.
- If the node to be deleted has one child node, the child node is replaced with the node to be deleted simply.
- If the node to be deleted has two child nodes then,
  - Either replace the node with it's **inorder predecessor** , i.e, **the largest element of the left sub tree.**
  - Or replace the node with it's **inorder successor** , i.e, **the smallest element of the right sub tree.**

Example : Consider the following AVL Tree

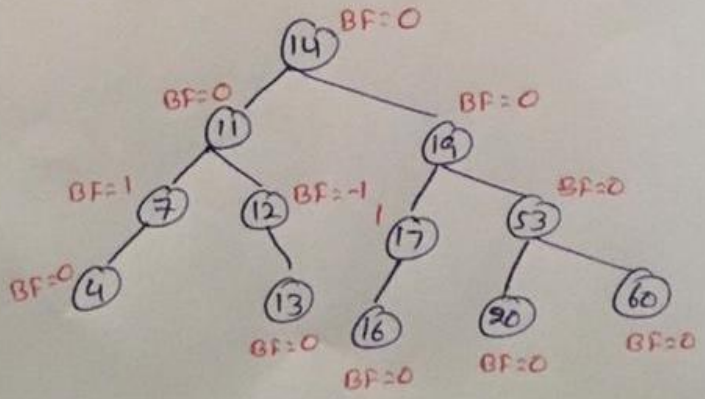


Delete the elements 8, 7, 11, 14, 17 from the AVL Tree.

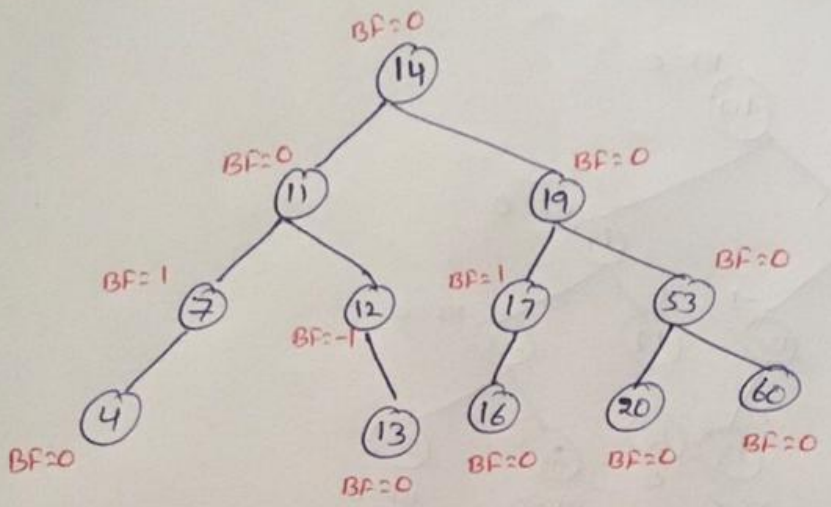
Step1 : Delete element 8. (case-1)



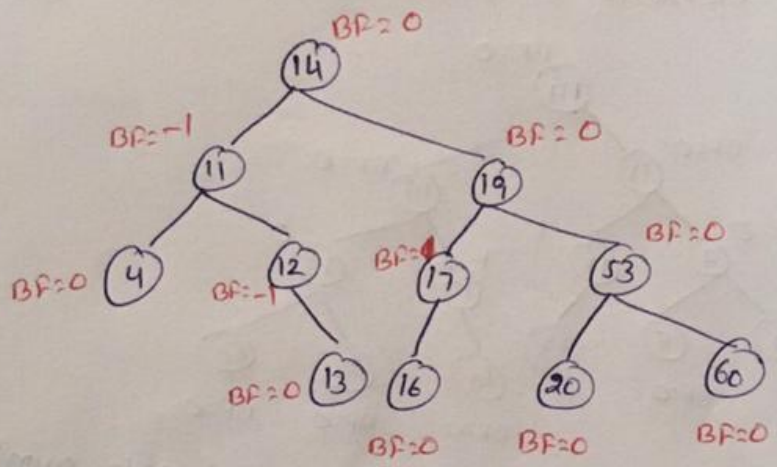
it is the simplest case, replace the leaf node with the NULL and delete the node and terminate the function.



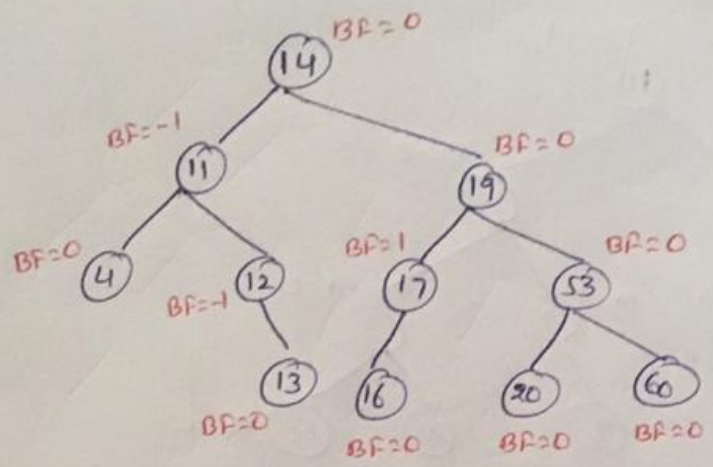
Step2 : Delete element 7, [case-II]



Deletion of a node 7 having one child node, Delete it and Replace with its child.

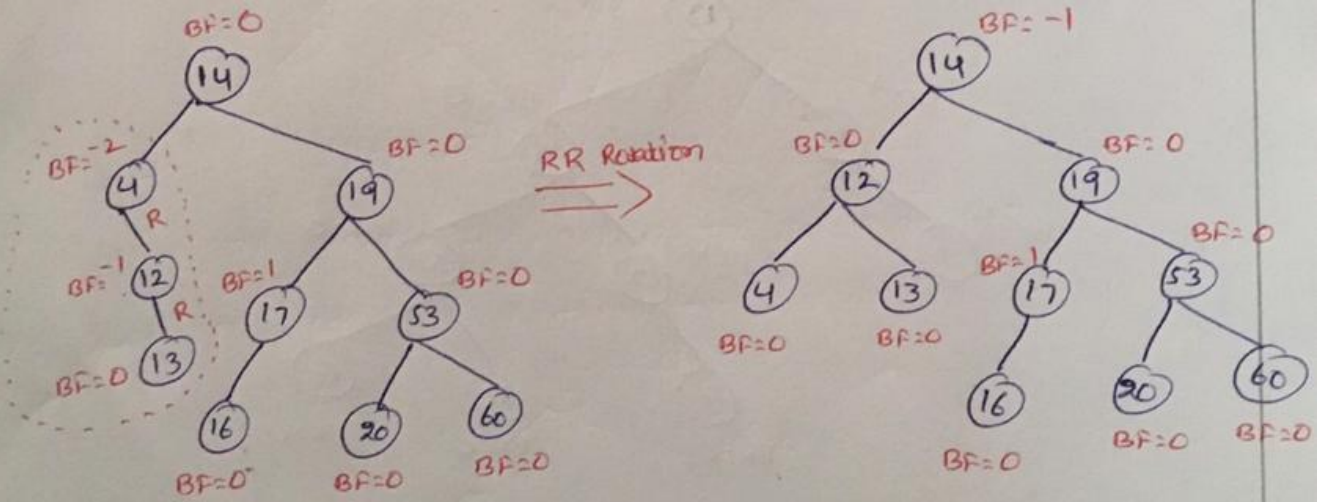


Step3 : Delete element 11

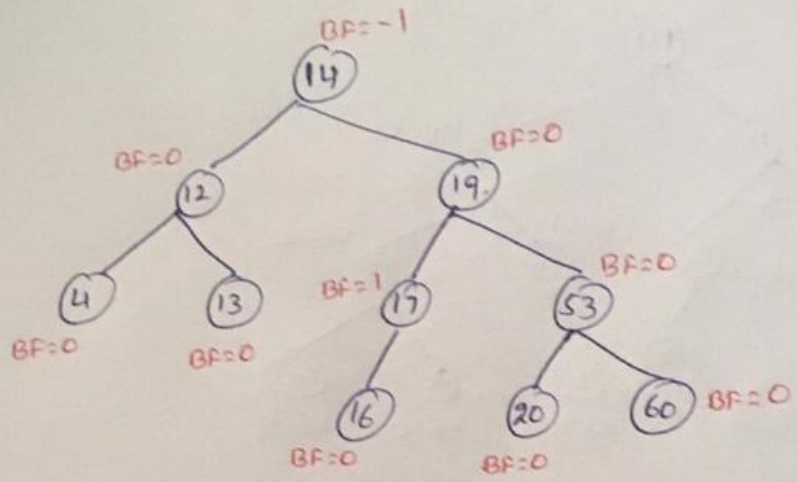


Deletion of a node 11 having two child nods, Now we have to find the inorder Predessor or inorder Successor of the node To find the inorder Predessor, copy the content of the Predessor node of the node into the deletion node.

[inorder = 4, 11, 12, 13, 14, 16, 17, 19, 20, 53, 60]

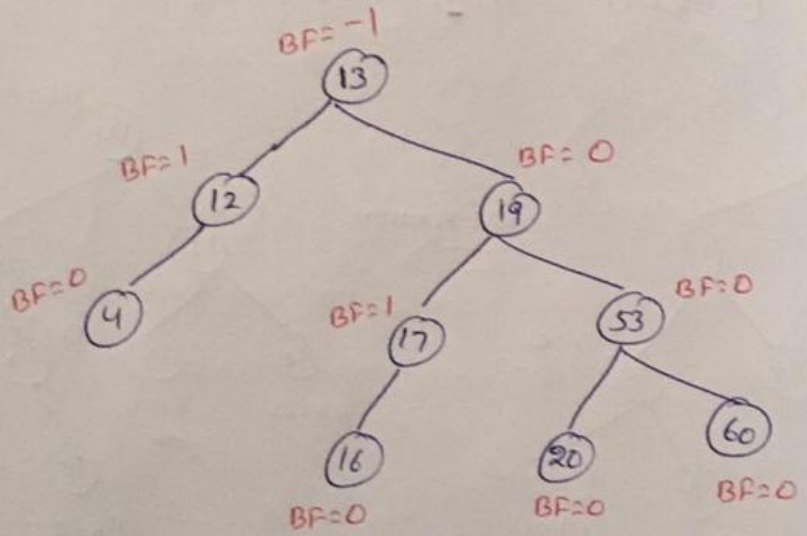


Step4 : Delete element 14



Deletion of a node 14 having two children node. Now we have to find inorder Predessor (or) inorder Successor.

-> copy the contents of the Predessor node of the node into the deletion node. [inorder : 4, 12, 13, 14, 16, 17, 19, 20, 53, 60]



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.

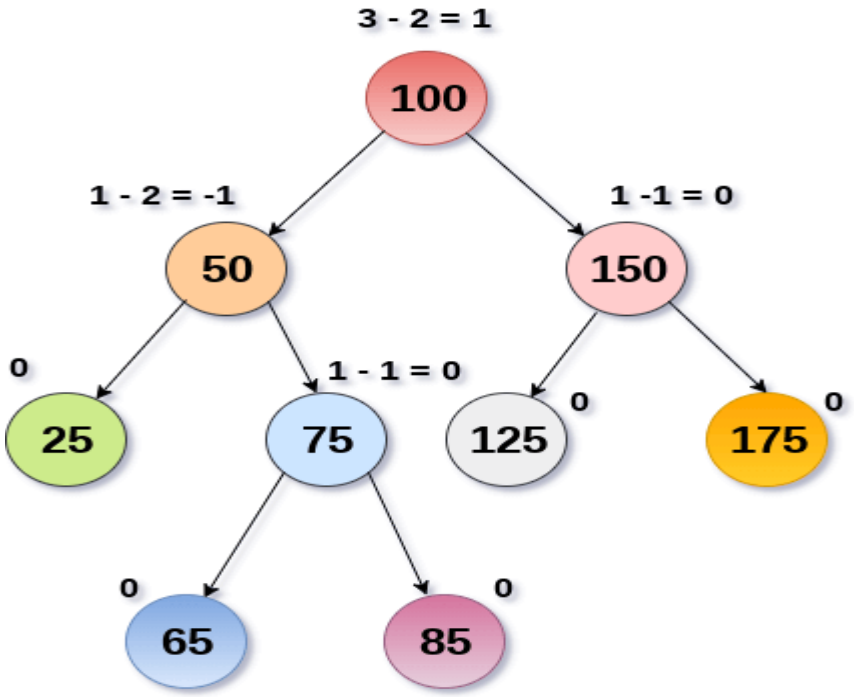
Step 6 - If search element is larger, then continue the search process in right sub tree.

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.

Example:



AVL Tree

If we want search the element 85.

The following steps to search an element 85 in AVL tree...

1. Read the search element is 85
2. Compare the search element (85) with the value of root node (100) in the tree. Since 85 is Less than 100, then the search process in left sub tree.
3. Compare the search element (85) with the value (50) in the left sub tree. Since 85 is greater than 50, then the search process in right sub tree.
4. Compare the search element (85) with the value (75) in the right sub tree. Since 85 is greater than 75, then the search process in right sub tree.
5. Compare the search element (85) with the value (85) in the right sub tree. Since 85 is equal to 85, Then display "Element is Found" and terminate the function.

Red - Black Tree

Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK. In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

Properties of Red Black Tree

Property #1: Red - Black Tree must be a Binary Search Tree.

Property #2: The ROOT node must be colored BLACK.

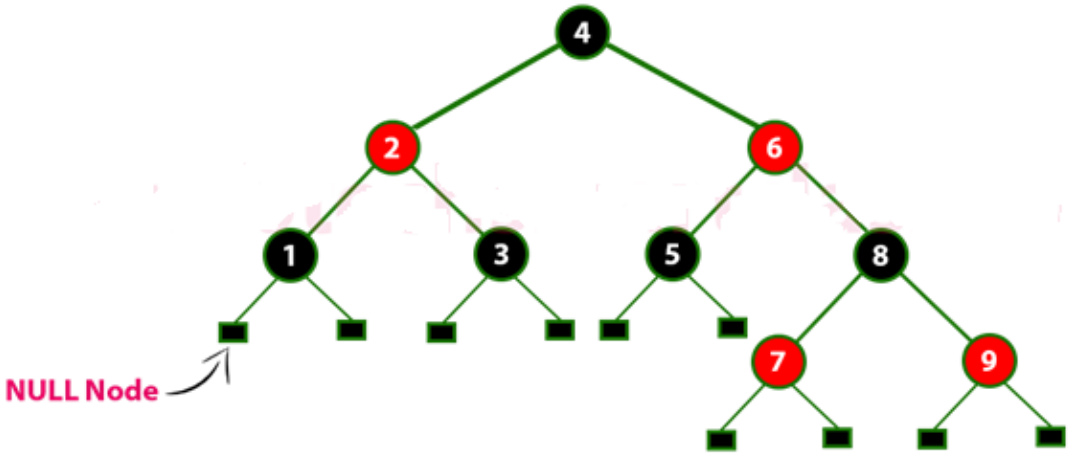
Property #3: The children of Red colored node must be colored BLACK. (**There should not be two consecutive RED nodes**).

Property #4: In all the paths of the tree, there should be same number of BLACK colored nodes.

Property #5: Every new node must be inserted with RED color.

Property #6: Every leaf (i.e, NULL node) must be colored BLACK.

Example



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

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

Insertion into RED BLACK Tree

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

1.Recolor

2. Rotation

3. Rotation followed by Recolor

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

Step 1 - If tree is Empty , Create newNode as Root node with color Black .

Step 2 - If tree is not Empty , Create newNode as leaf node with color Red.

Step 3 - If the parent of newNode is Black then exit from the operation.

Step 4 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

a) If it is colored Black or NULL then do suitable Rotation and Recolor it.

b) If it is colored Red then perform Recolor and also check if parent's parent of newnode is not root node then recolor it & recheck.

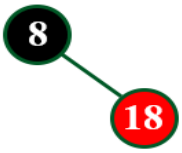
Create a RED BLACK Tree by inserting following sequence of number
8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

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

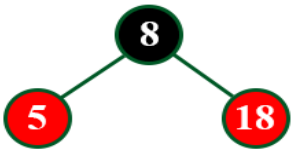
insert (18)

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



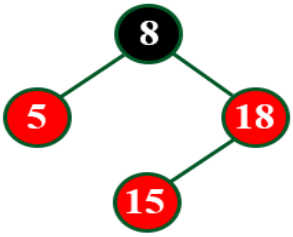
insert (5)

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



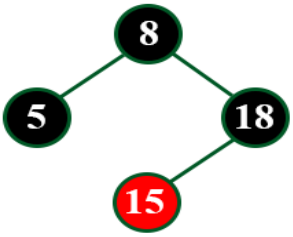
insert (15)

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.

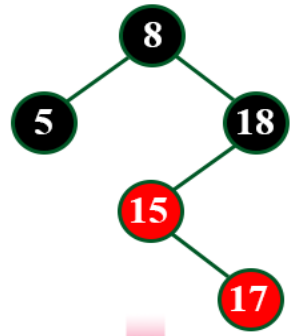
After RECOLOR



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

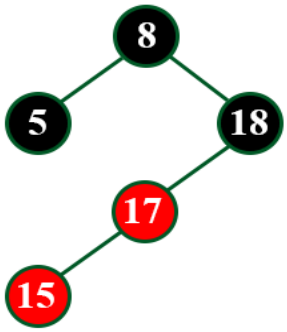
insert (17)

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

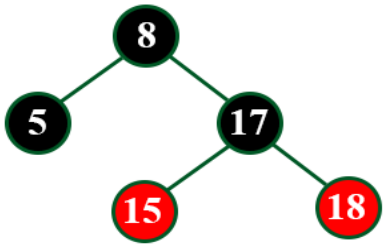


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Rotation

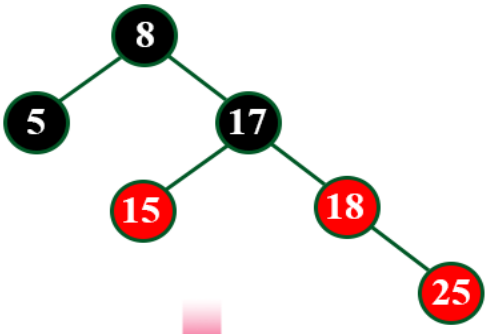


After Rotation & Recolor



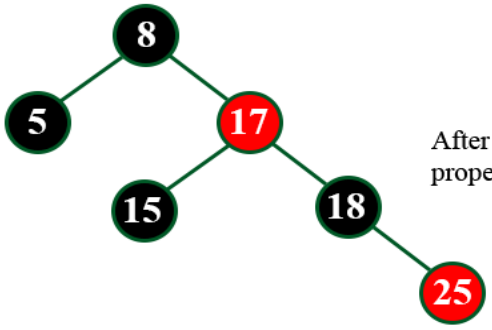
insert (25)

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.

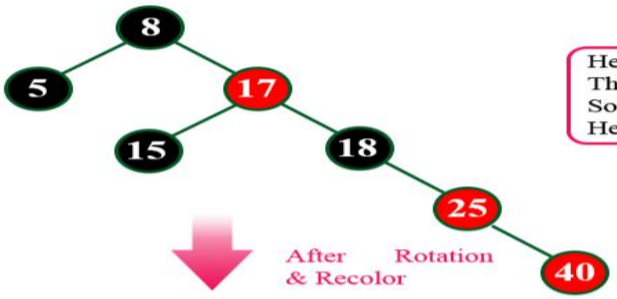
After Recolor



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

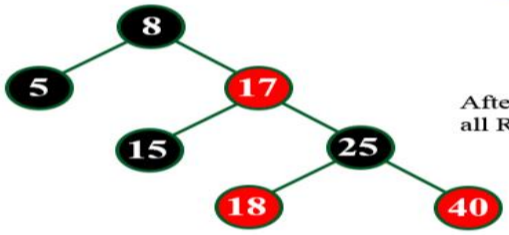
insert (40)

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 Rotation and Recheck.

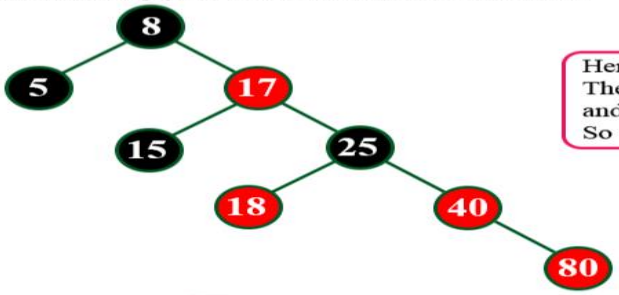
After Rotation & Recolor



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

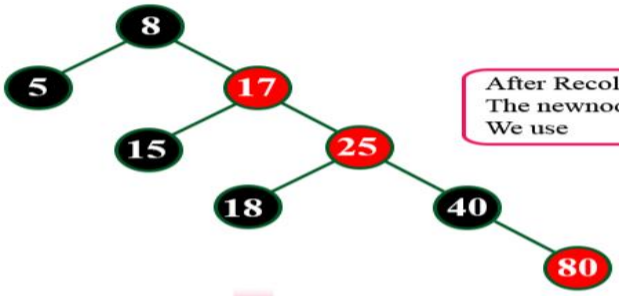
insert (80)

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



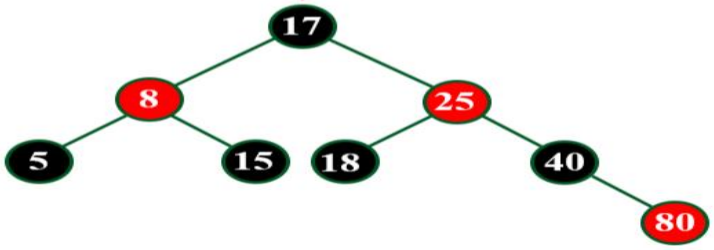
Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

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 Rotation & Recolor.

After Rotation & Recolor



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