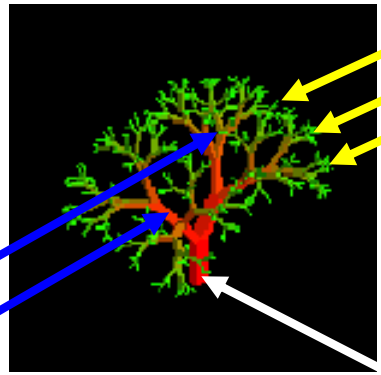
TREE

Nature lover's view of a tree

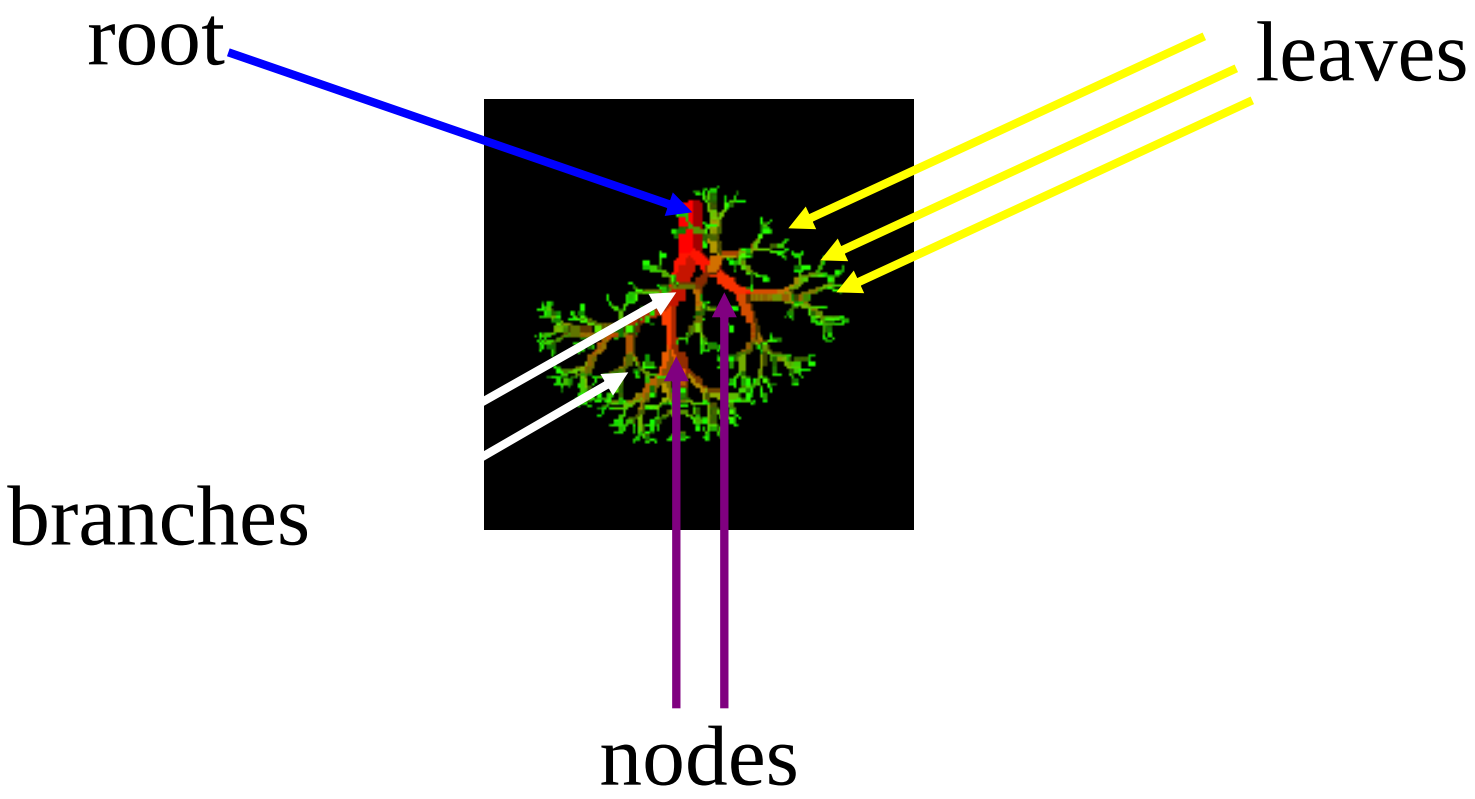


leaves

branches

root

Computer Scientist's View



Introduction to trees

- So far we have discussed mainly linear data structures – strings, arrays, lists, stacks and queues
- Now we will discuss a non-linear data structure called **tree**.
- Trees are mainly used to represent data containing a hierarchical relationship between elements, for example, records, family trees and table of contents.
- Consider a parent-child relationship

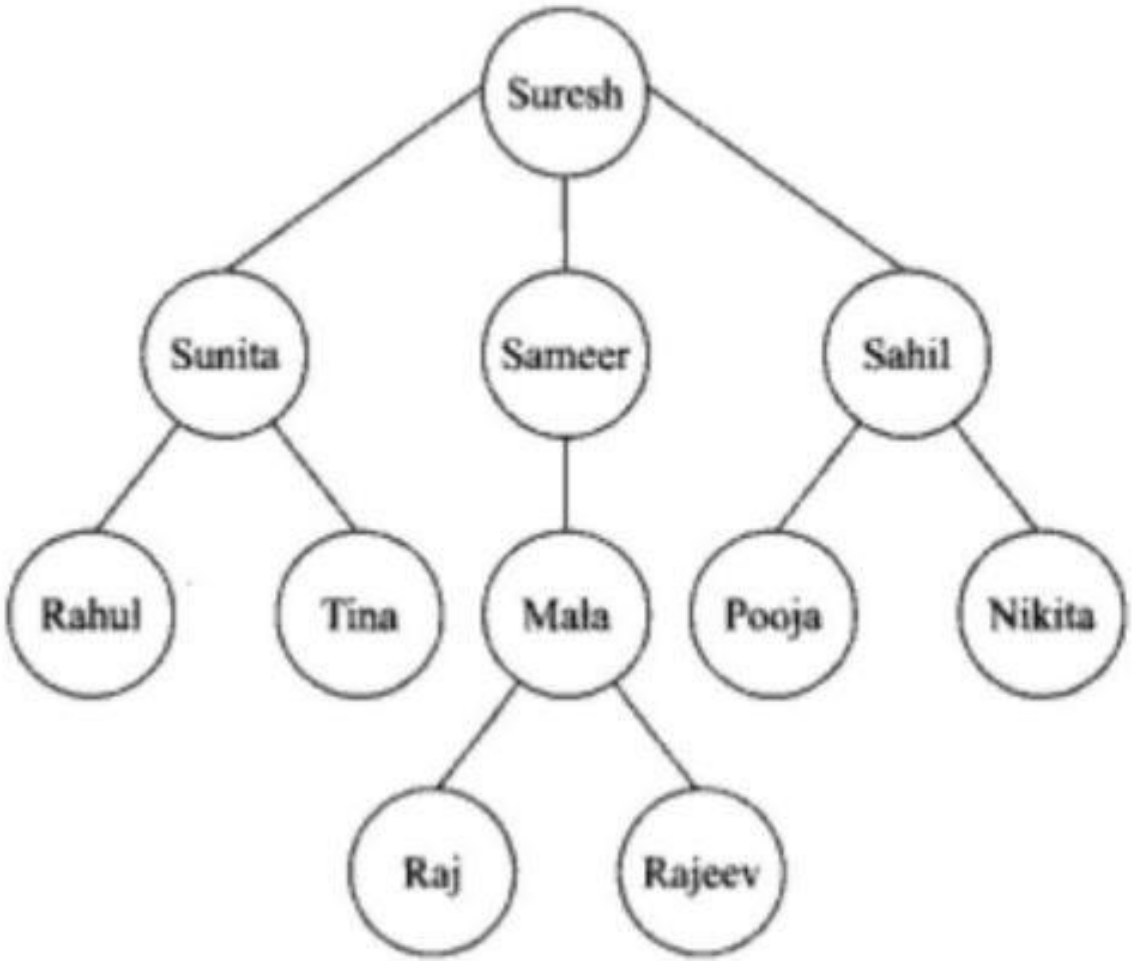
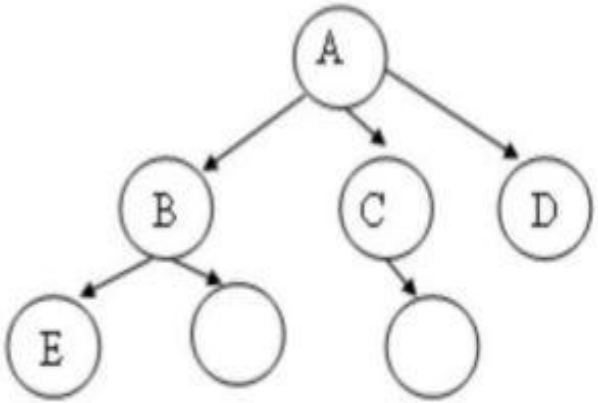


Fig. 8.1 *A Hypothetical Family Tree*

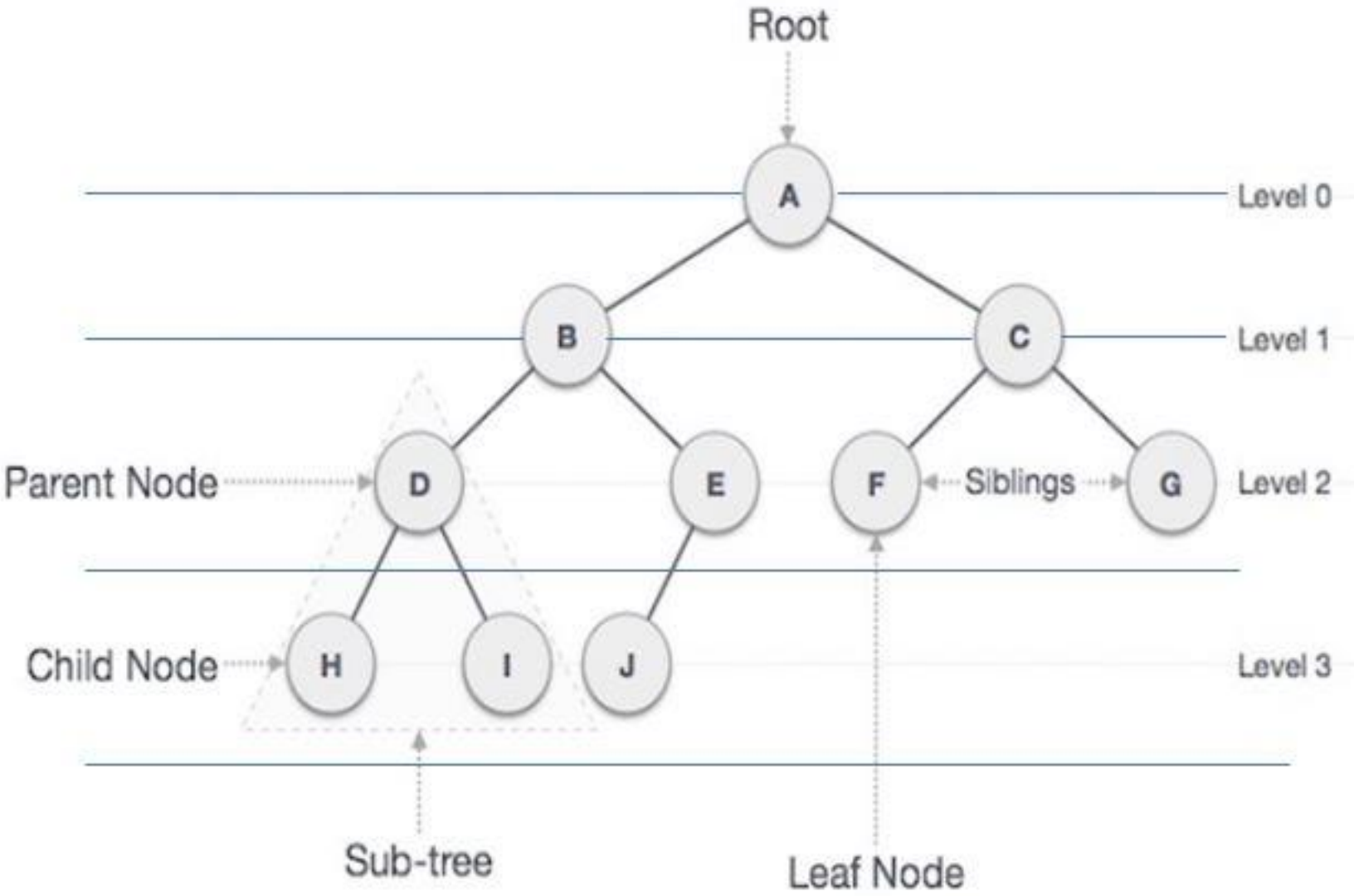
Tree

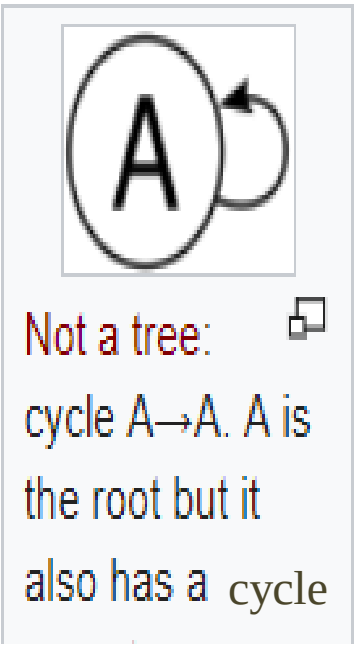
- A tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.
  - Tree is a sequence of **nodes**
  - There is a starting node known as a **root** node
  - Every node other than the root has a **parent** node.
  - Nodes may have any number of children

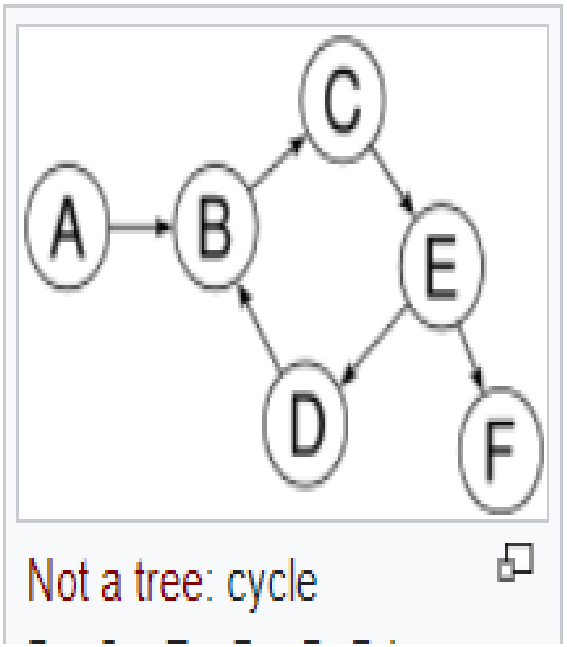


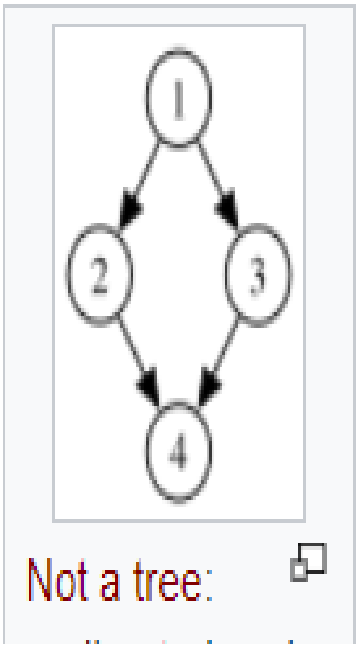
A has 3 children, B, C, D
A is parent of B

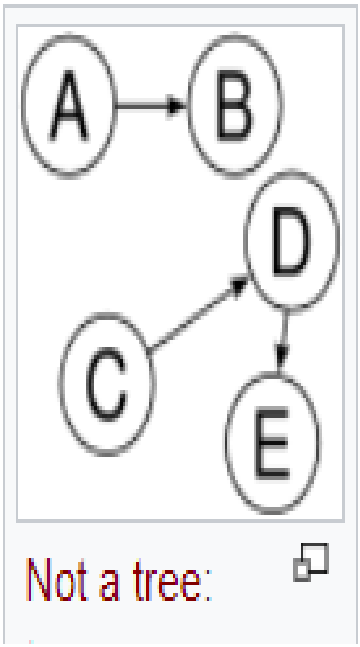
Fig:1 Example of tree











These figures are not a tree it is graph because there exist loop or cycle.

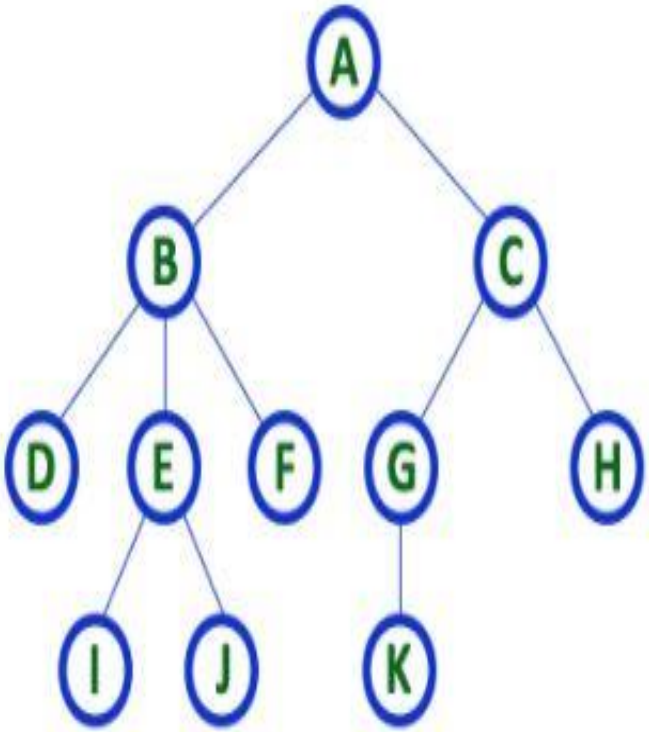
Characteristics of trees

- Non-linear data structure
- Combines advantages of an ordered array
- Searching as fast as in ordered array
- Insertion and deletion as fast as in linked list
- Simple and fast

Application

- Directory structure of a file store
- Structure of an arithmetic expressions
- Used in almost every 3D video game to determine what objects need to be rendered.
- Used in almost every high-bandwidth router for storing router-tables.
- used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.

Example



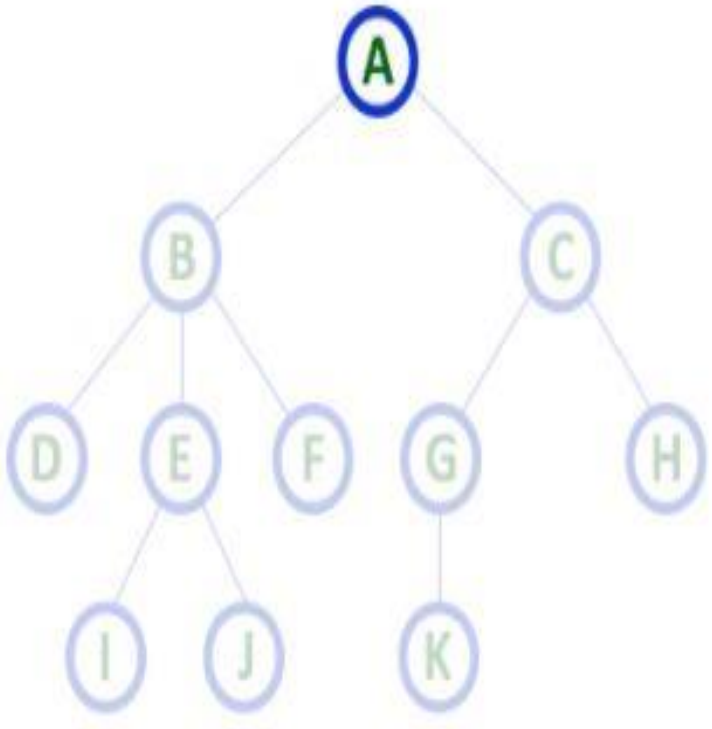
TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Tree Terminology:

- In a tree data structure we use following terminology…

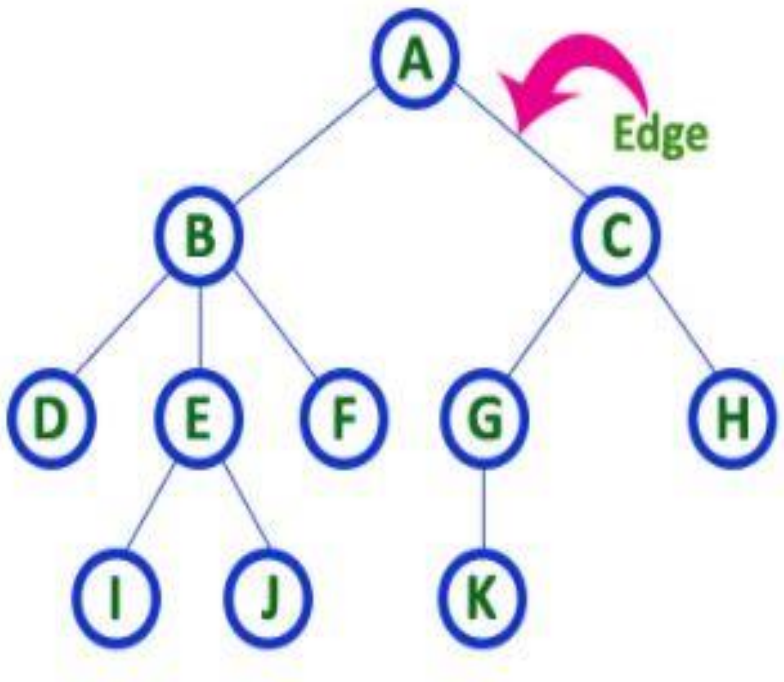
1. Root



Here 'A' is the 'root' node

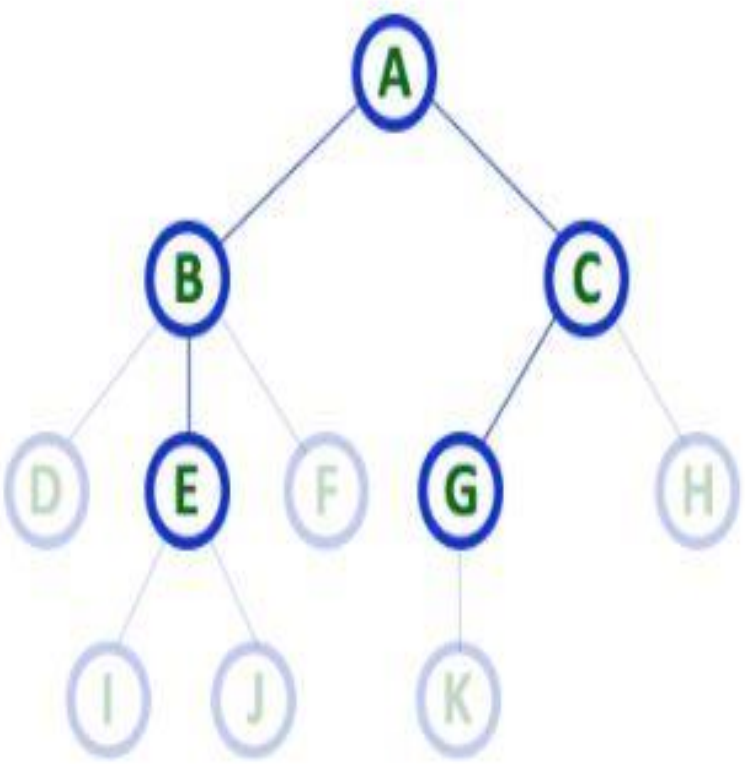
- In any tree the first node is called as ROOT node

2. Edge



- In any tree, 'Edge' is a connecting link between two nodes.

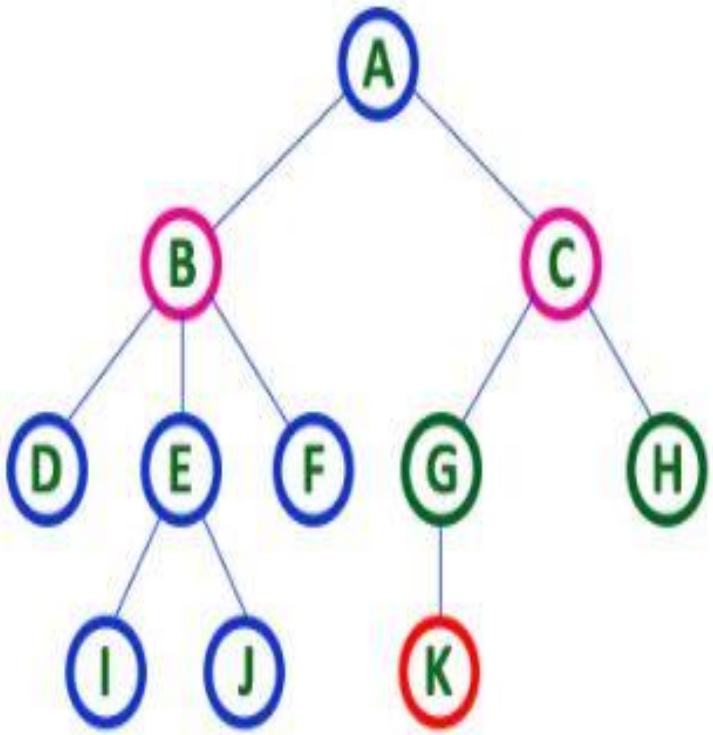
3. Parent



Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child



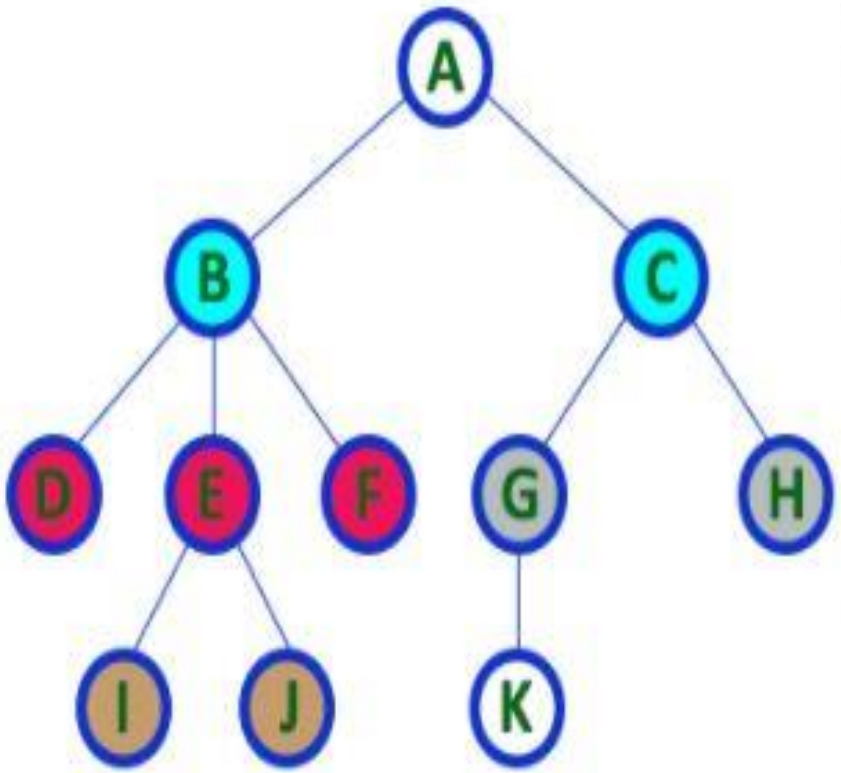
Here B & C are Children of A

Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

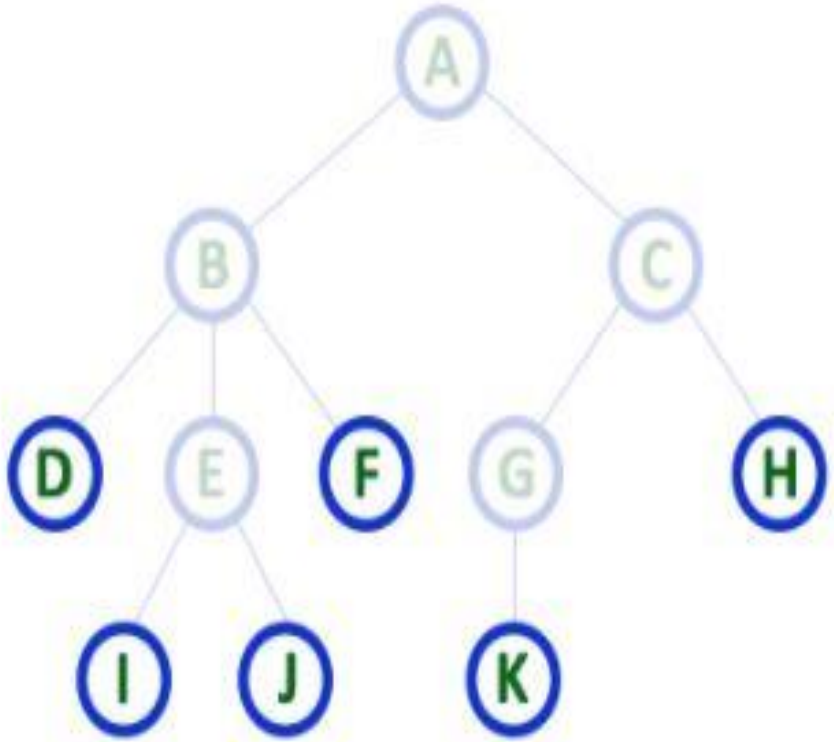
5. Sibling



Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

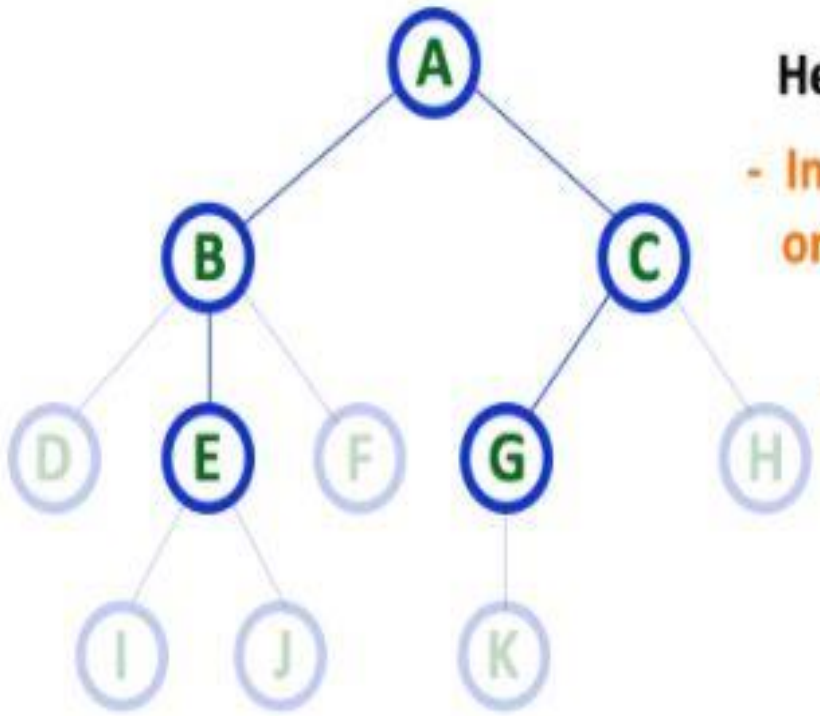
6. Leaf



Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal node

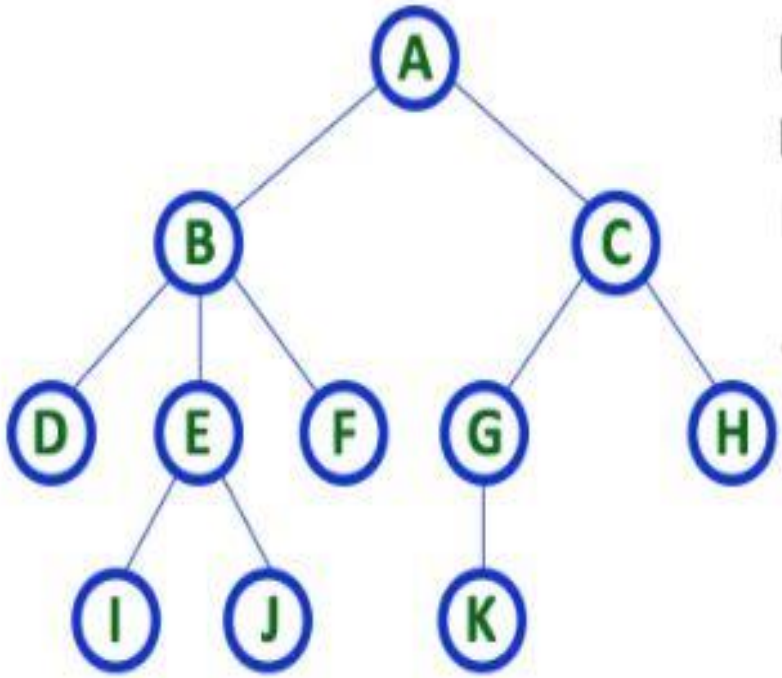


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node

- Every non-leaf node is called as 'Internal' node

8. Degree



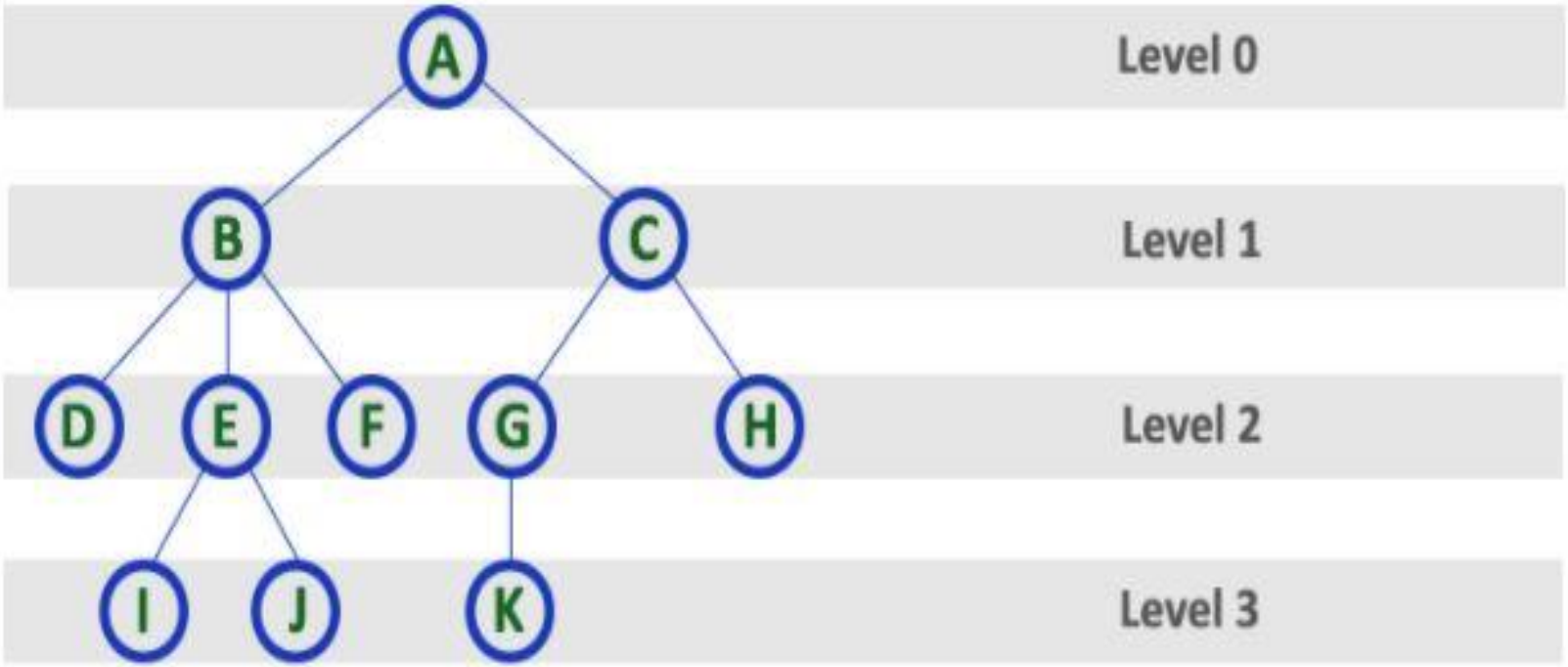
Here Degree of B is 3

Here Degree of A is 2

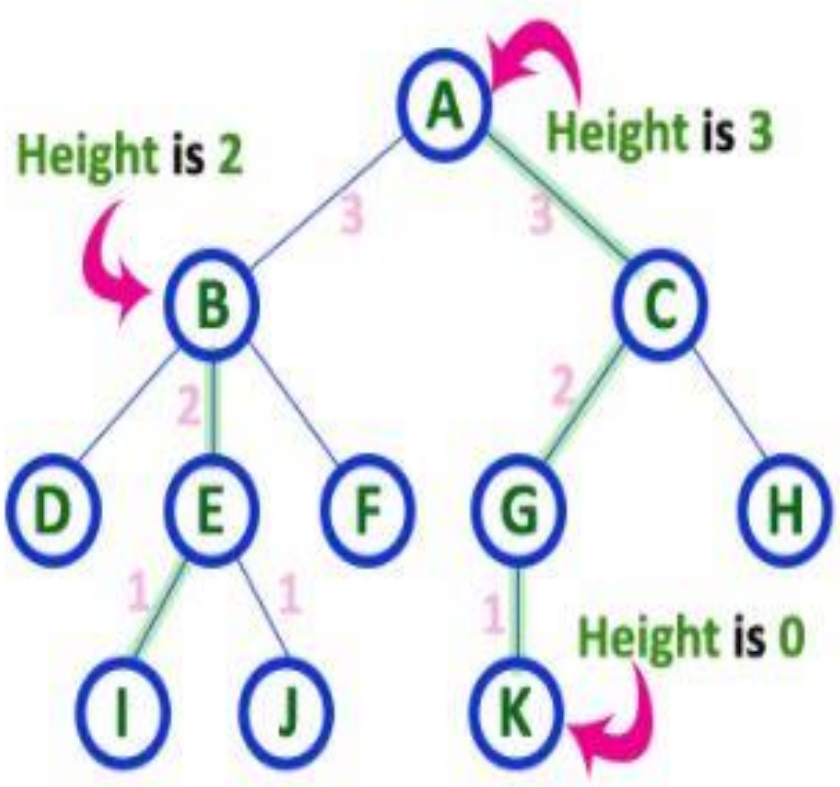
Here Degree of F is 0

- In any tree, 'Degree' a node is total number of children it has.

9. Level



10. Height

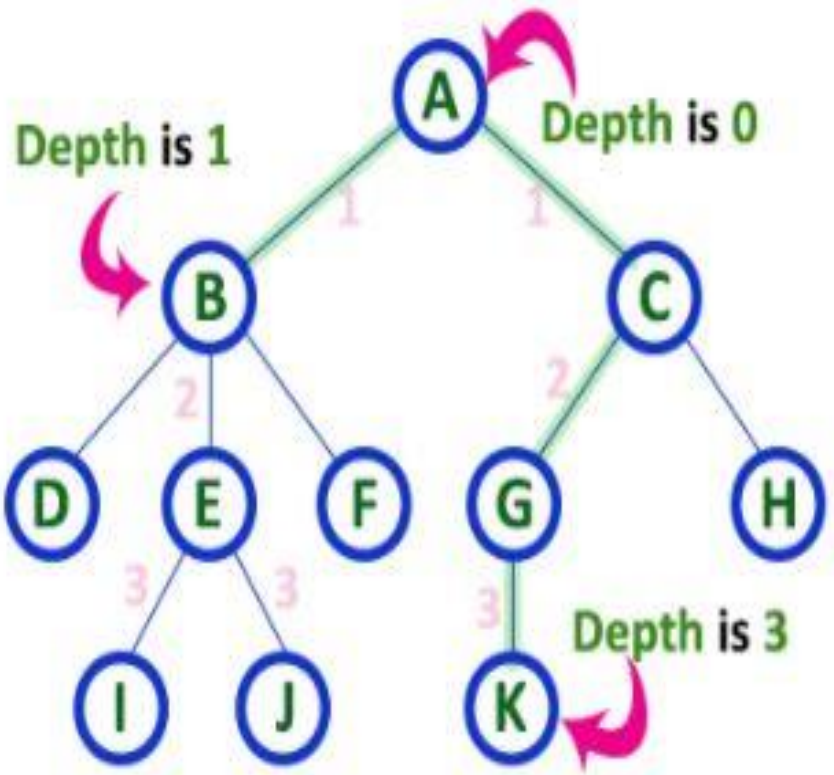


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.

- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

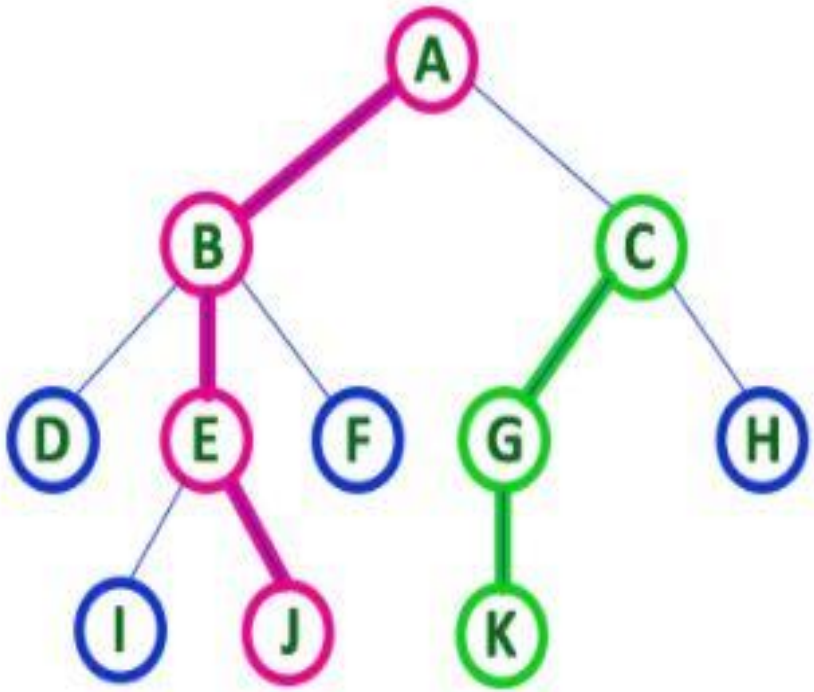


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.

- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

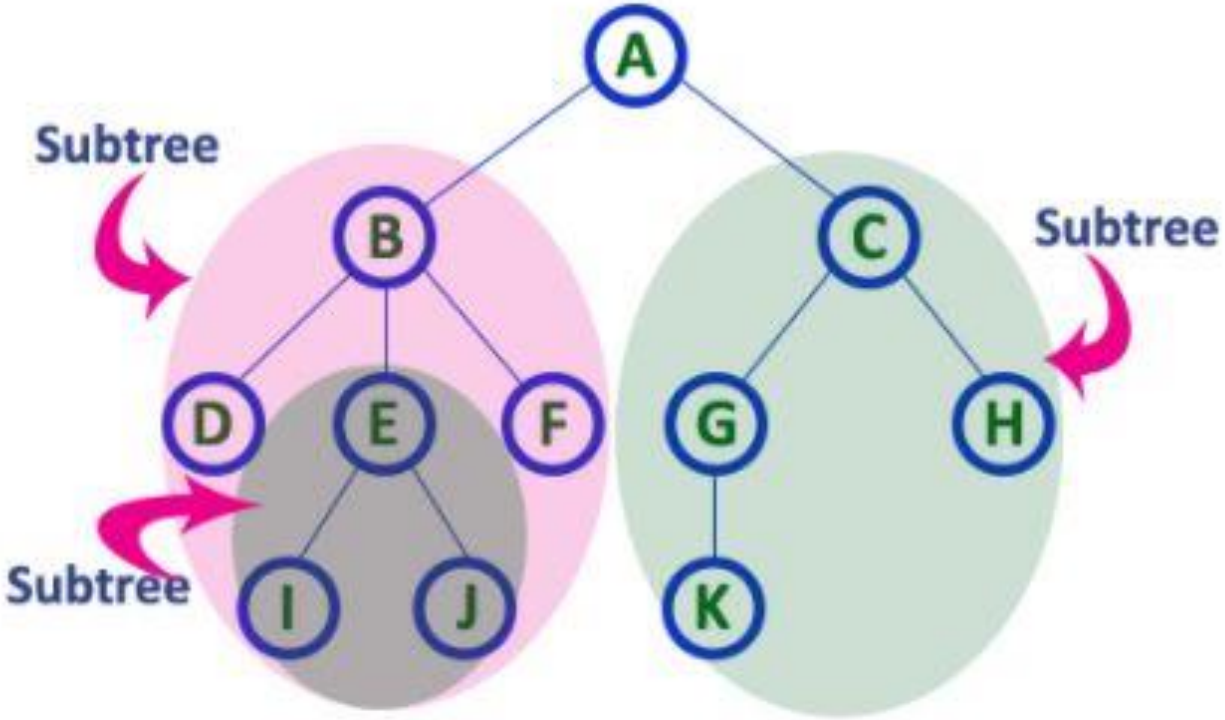
Here, 'Path' between A & J is

A - B - E - J

Here, 'Path' between C & K is

C - G - K

13. Sub Tree



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.

A tree in which every node can have a maximum of two children is called as Binary Tree.

- In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example

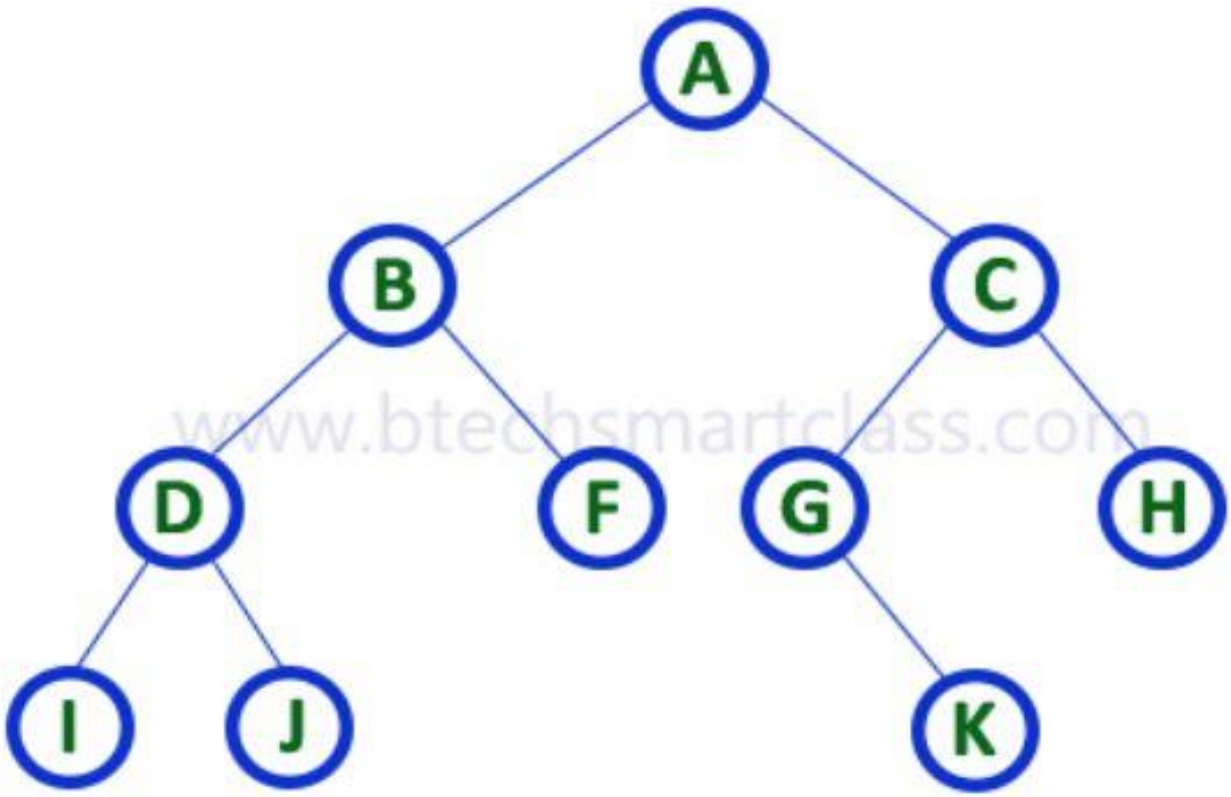


Fig: Binary Tree

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

1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-Tree**

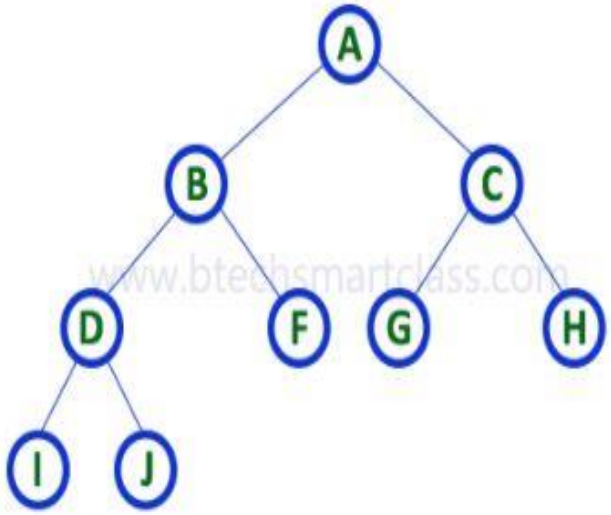
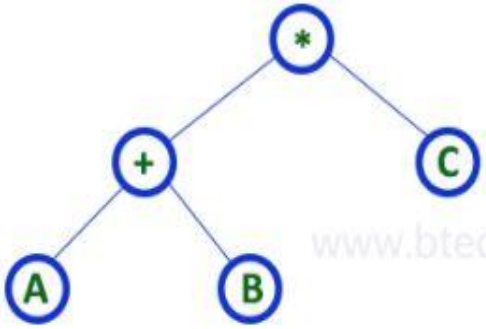


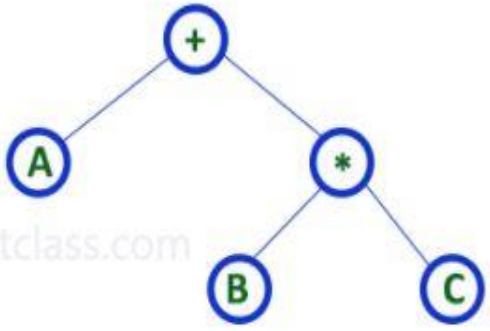
Fig: Strictly Binary Tree

Strictly binary tree data structure is used to represent mathematical expressions.

Example



(A + B) * C



A + B * C

2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

Complete binary tree is also called as **Perfect Binary Tree**

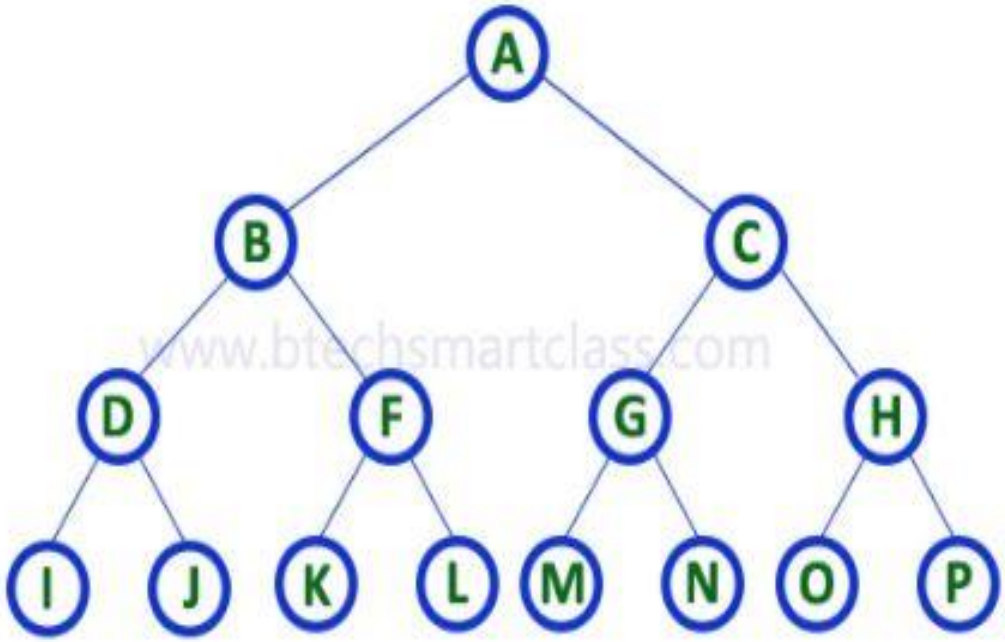


Fig: Complete Binary Tree

3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

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

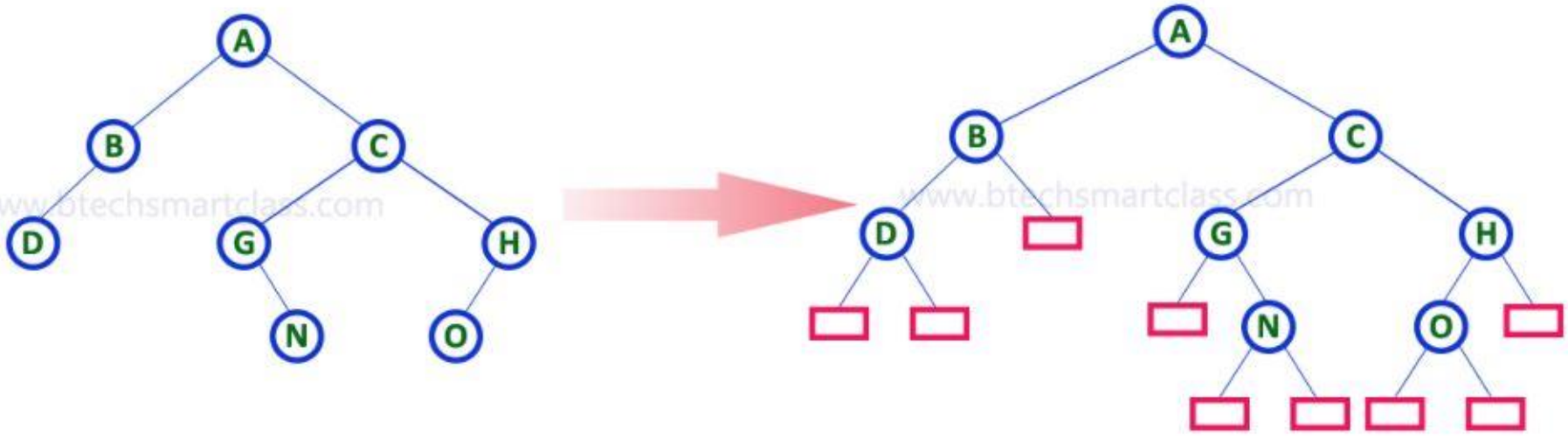


Fig: Extended Binary Tree

In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes.

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

Consider the following binary tree...

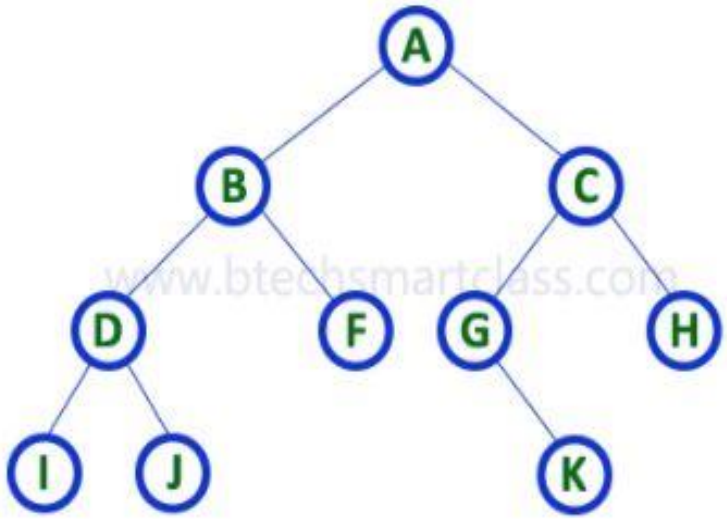
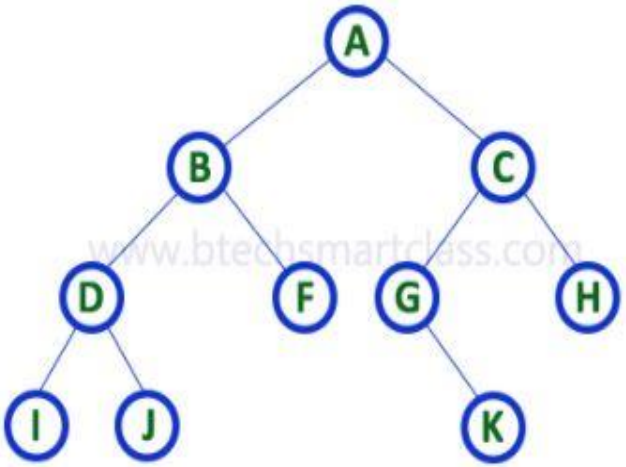


Fig: Binary Tree

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...

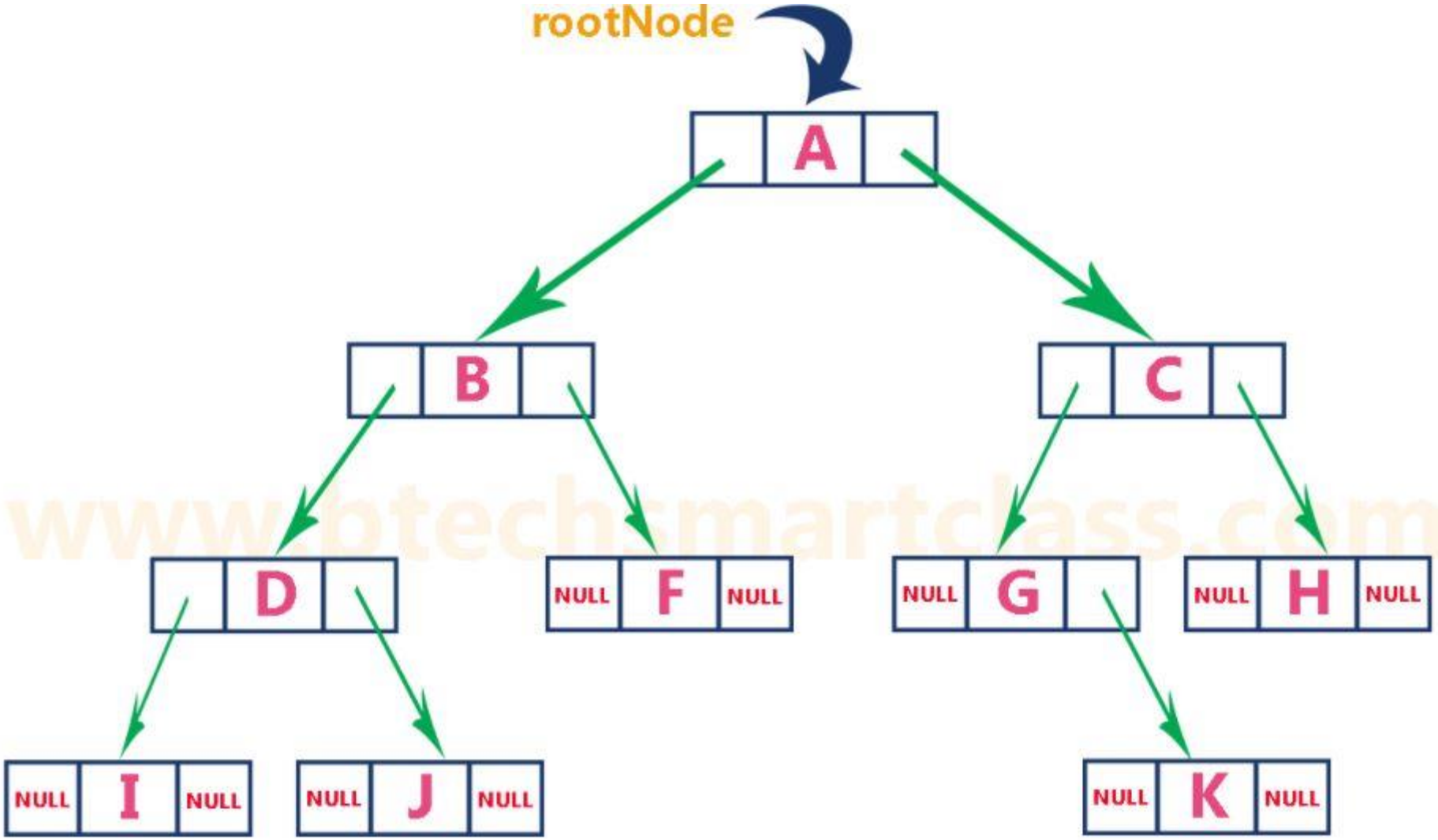
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

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 for storing left child address, second for storing 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...



Traversal

- The process of visiting each node in a tree exactly once, in a systematic way is called Traversal.

Binary Tree Traversals

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

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.

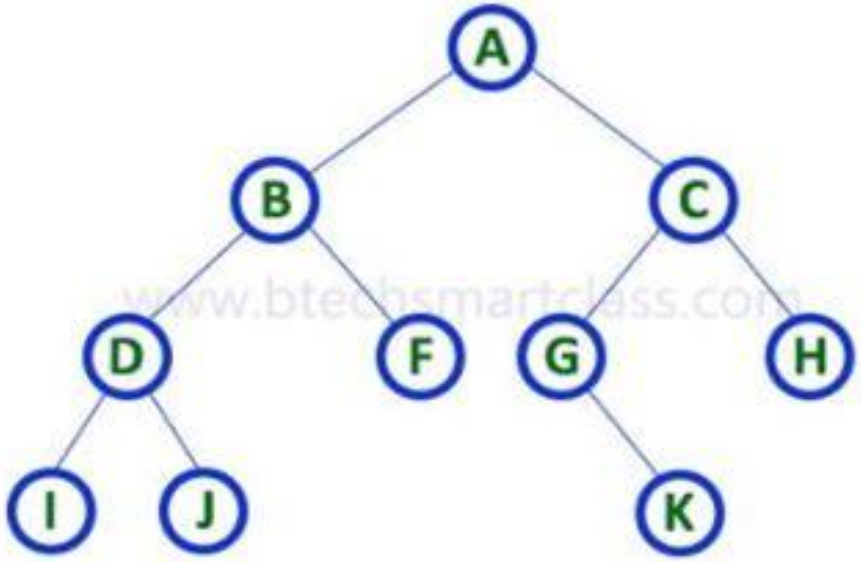
2. Pre - Order Traversal (root - leftChild - rightChild)

- 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.

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.

Q.1 Consider the following binary tree:



Find its:

i. **In-order traversal**
ii. **Pre-order traversal**
iii. **Post-order traversal**

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

   I - D - J - B - F - A - G - K - C - H

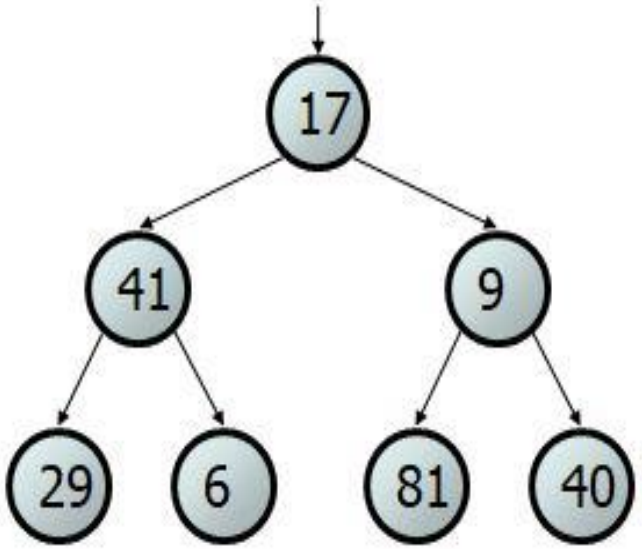
2. Pre - Order Traversal (root - leftChild - rightChild)

   A - B - D - I - J - F - C - G - K - H

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

   I - J - D - F - B - K - G - H - C - A

Q.2 Consider the following binary tree:

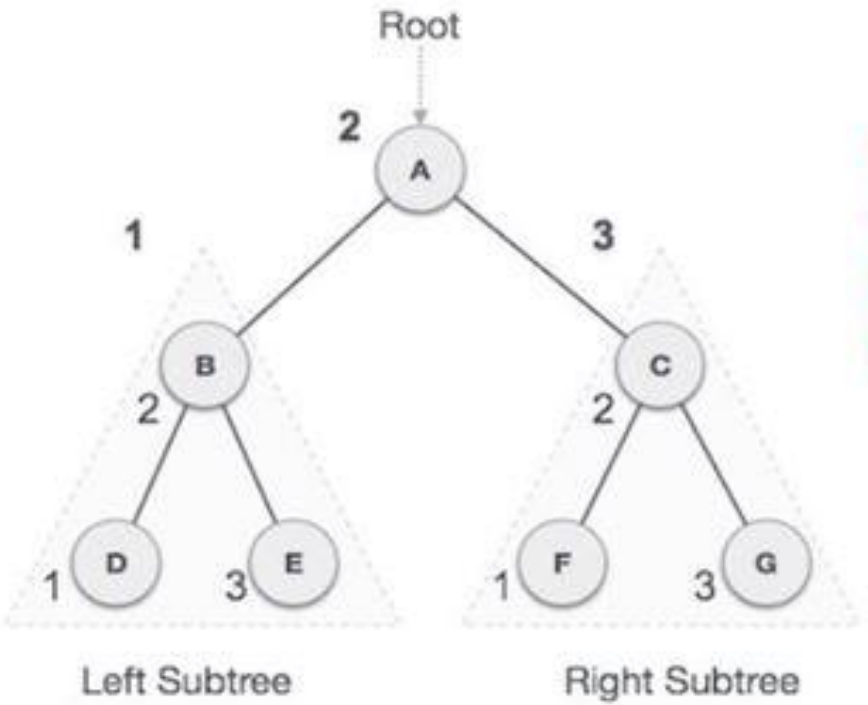


Find its:

i. **In-order traversal**
ii. **Pre-order traversal**
iii. **Post-order traversal**

- **In-order Traversal:** 29 41 6 17 81 9 40
- **Pre-order Traversal :** 17 41 29 6 9 81 40
- **Post-order Traversal :** 29 6 41 81 40 9 17

Q.3 Consider the following binary tree:

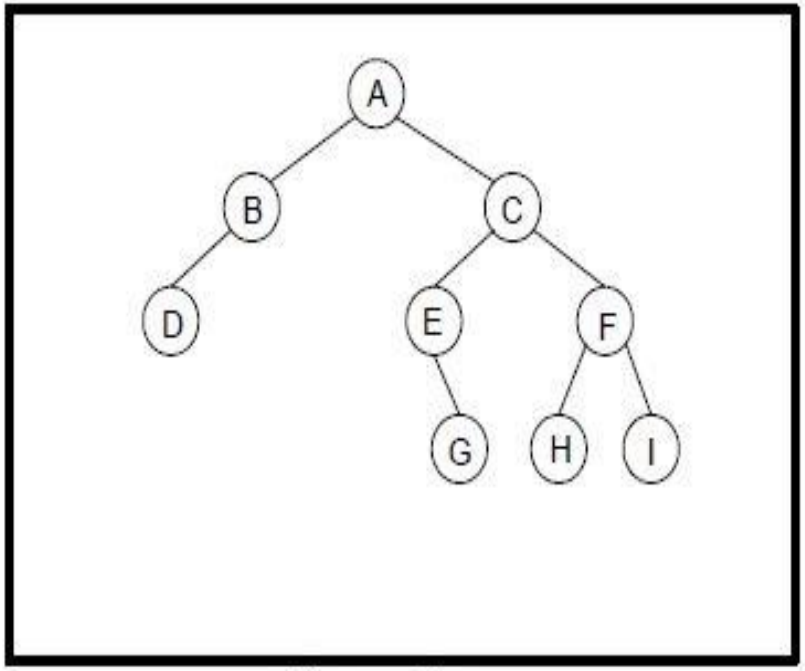


Find its:

i. **In-order traversal**
ii. **Pre-order traversal**
iii. **Post-order traversal**

- **In-order Traversal:** $D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$
- **Pre-order Traversal :** $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$
- **Post-order Traversal :** $D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Q.4 Traverse the following binary tree in pre, post, inorder



Binary Tree

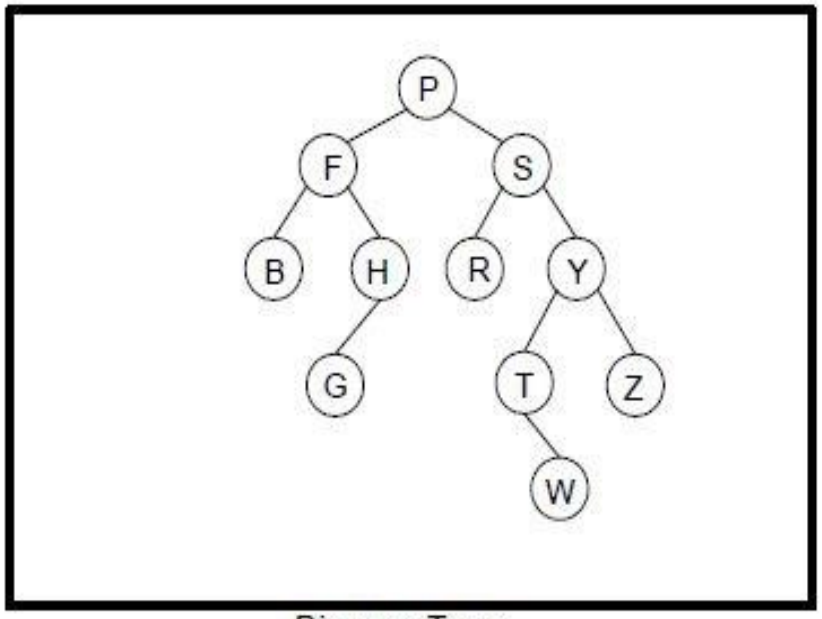
Preorder traversal
A, B, D, C, E, G, F, H, I

Postorder traversal
D, B, G, E, H, I, F, C, A

Inorder traversal yields:
D, B, A, E, G, C, H, F, I

Pre, Post, Inorder

Q.5 Traverse the following binary tree in pre, post, inorder



Binary Tree

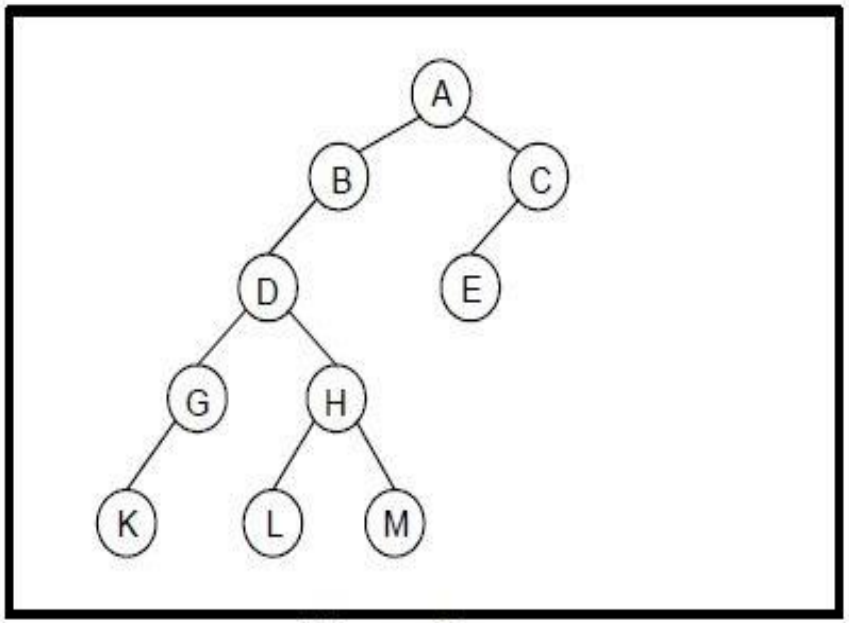
Preorder traversal
P, F, B, H, G, S, R, Y, T, W, Z

Postorder traversal
B, G, H, F, R, W, T, Z, Y, S, P

Inorder traversal
B, F, G, H, P, R, S, T, W, Y, Z

Pre, Post, Inorder

Q.6 Traverse the following binary tree in pre, post, inorder.



Binary Tree

Preorder traversal
A, B, D, G, K, H, L, M, C, E

Postorder travarsal
K, G, L, M, H, D, B, E, C, A

Inorder travarsal
K, G, D, L, H, M, B, A, E, C

Pre, Post, Inorder

Example 1:

Construct a binary tree from a given preorder and inorder sequence:

Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

Solution:

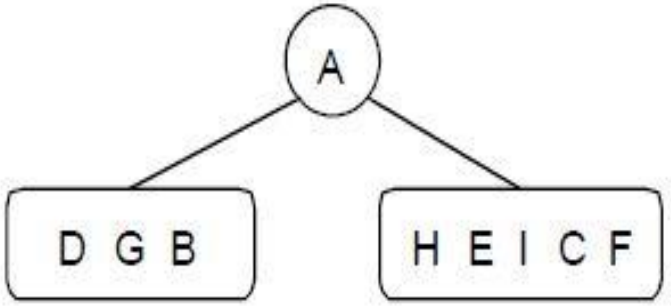
From Preorder sequence ***A*** *B D G C E H I F, the root is: A*

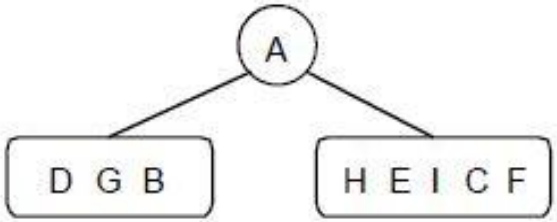
From Inorder sequence *D G B* ***A*** *H E I C F*, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree upto this point looks like:





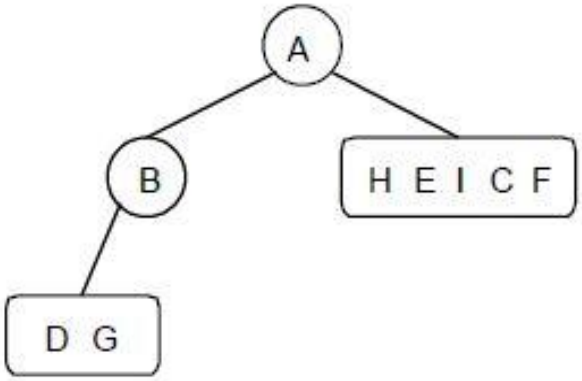
Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

To find the root, left and right sub trees for D G B:

From the preorder sequence B D G, the root of tree is: B

From the inorder sequence D G **B**, we can find that D and G are to the left of B.

The Binary tree upto this point looks like:

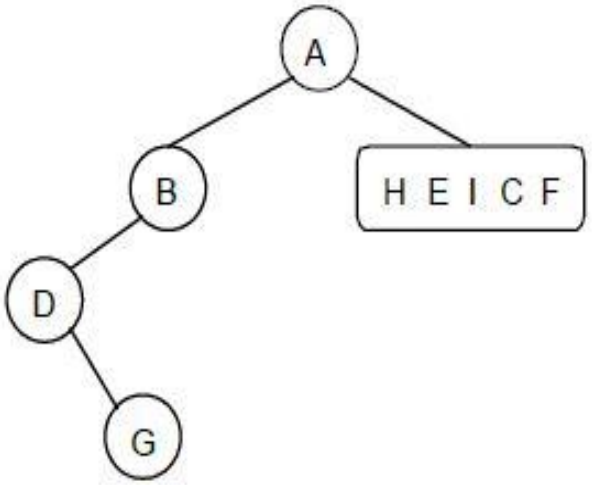


Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

To find the root, left and right sub trees for D G:

From the preorder sequence D ***G, the root of the tree is: D***

From the inorder sequence ***D*** G, we can find that there is no left node to D and G is at the right of D.
The Binary tree upto this point looks like:



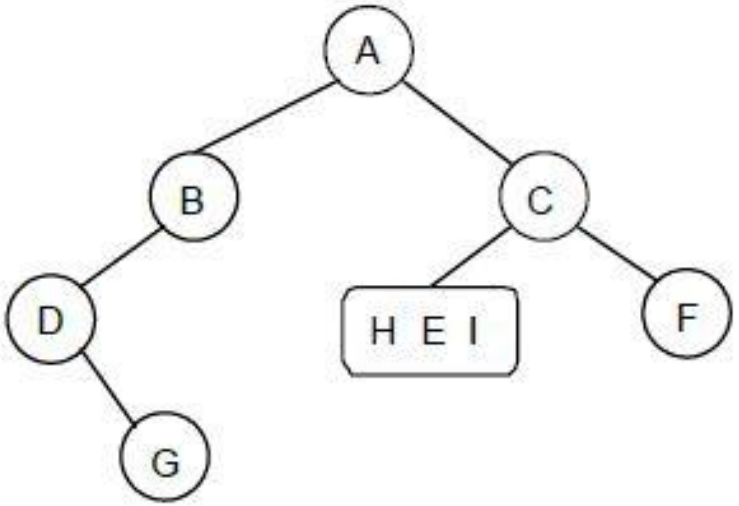
Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

To find the root, left and right sub trees for H E I C F:

From the preorder sequence C E H I F, the root of the left sub tree is: C

From the inorder sequence *H E I* ***C*** *F*, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:



Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

To find the root, left and right sub trees for H E I:

From the preorder sequence E H I, the root of the tree is: E

From the inorder sequence *H* ***E*** *I*, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:

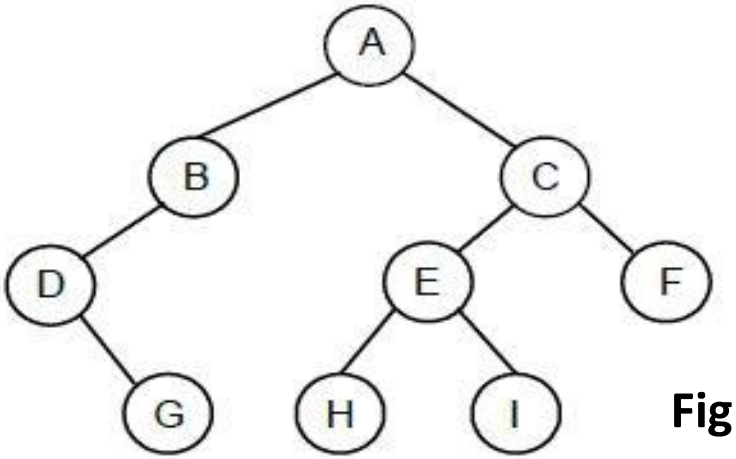


Fig: Binary Tree

Example 2:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

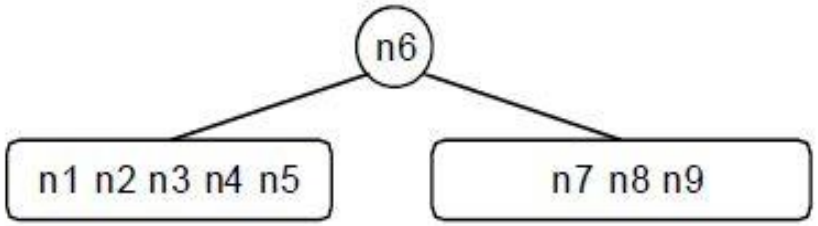
Solution:

From Postorder sequence n1 n3 n5 n4 n2 n8 n7 n9 **n6** *, the root is: n6*

From Inorder sequence *n1 n2 n3 n4 n5* **n6** *n7 n8 n9*, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5
Right sub tree is: n7 n8 n9

The Binary tree upto this point looks like:



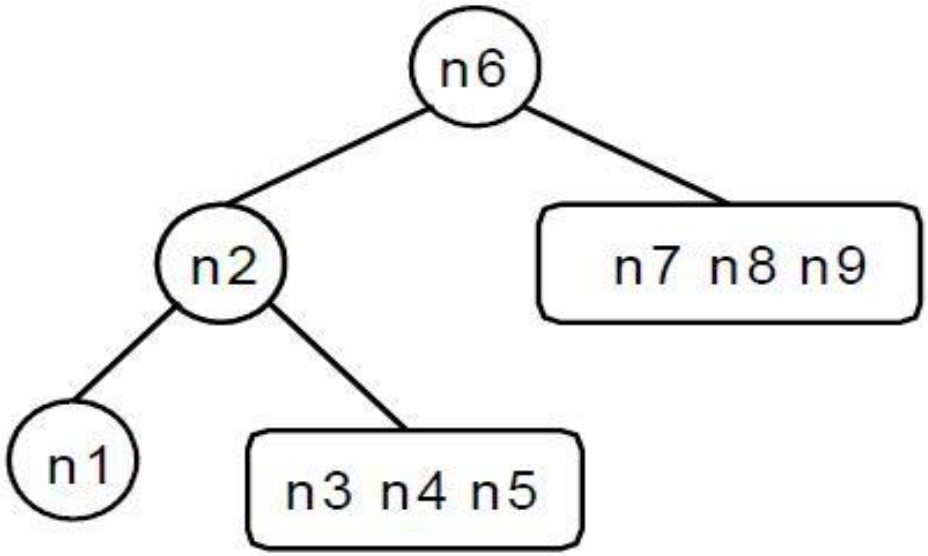
Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

To find the root, left and right sub trees for *n1 n2 n3 n4 n5*:

From the postorder sequence n1 n3 n5 n4 n2***, the root of tree is: n2***

From the inorder sequence *n1* **n2** *n3 n4 n5*, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.

The Binary tree upto this point looks like:



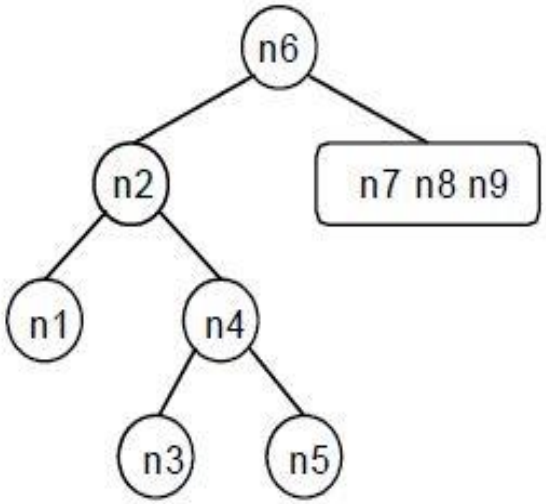
Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

To find the root, left and right sub trees for n3 n4 n5:

From the postorder sequence n3 n5 n4, ***the root of the tree is: n4***

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is to the right of n4.

The Binary tree upto this point looks like:



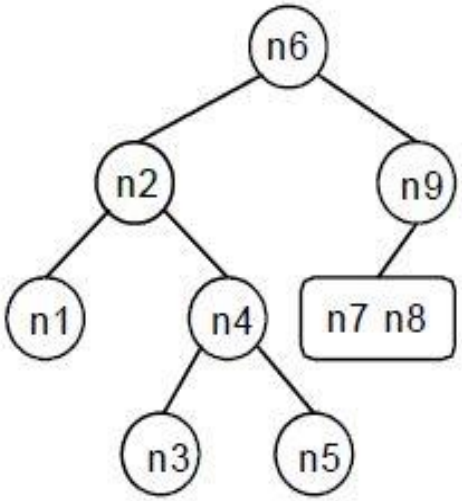
Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

To find the root, left and right sub trees for n7 n8 and n9:

From the postorder sequence n8 n7 n9, ***the root of the left sub tree is: n9***

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are to the left of n9 and no right subtree for n9.

The Binary tree upto this point looks like:



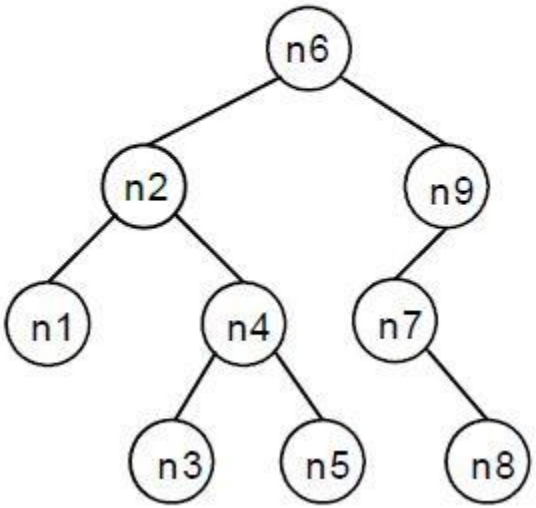
Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

To find the root, left and right sub trees for n7 and n8:

From the postorder sequence n8 n7***, the root of the tree is: n7***

From the inorder sequence **n7** *n8*, we can find that there is no left subtree for n7 and n8 is to the right of n7.

The Binary tree upto this point looks like:



Do this questions:

Construct a binary tree from a given preorder and inorder sequence:

Pre-order A - B - D - I - J - F - C - G - K - H

In-order I - D - J - B - F - A - G - K - C - H

Construct a binary tree from a given preorder and inorder sequence:

Pre-order Traversal : $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

In-order Traversal: $D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Construct a binary tree from a given preorder and inorder sequence:

Preorder traversal
A, B, D, C, E, G, F, H, I

Inorder traversal yields:
D, B, A, E, G, C, H, F, I

Construct a binary tree from a given postorder and inorder sequence:

In-order I - D - J - B - F - A - G - K - C - H

Post-order I - J - D - F - B - K - G - H - C - A

Construct a binary tree from a given postorder and inorder sequence:

In-order Traversal: $D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Post-order Traversal : $D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Construct a binary tree from a given postorder and inorder sequence:

Inorder traversal yields:
D, B, A, E, G, C, H, F, I

Postorder traversal
D, B, G, E, H, I, F, C, A

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.

A binary tree has the following time complexities...

1. Search Operation - O(n)
2. Insertion Operation - O(1)
3. Deletion Operation - O(n)

To enhance the performance of binary tree, we use special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focus on the search operation in binary tree.

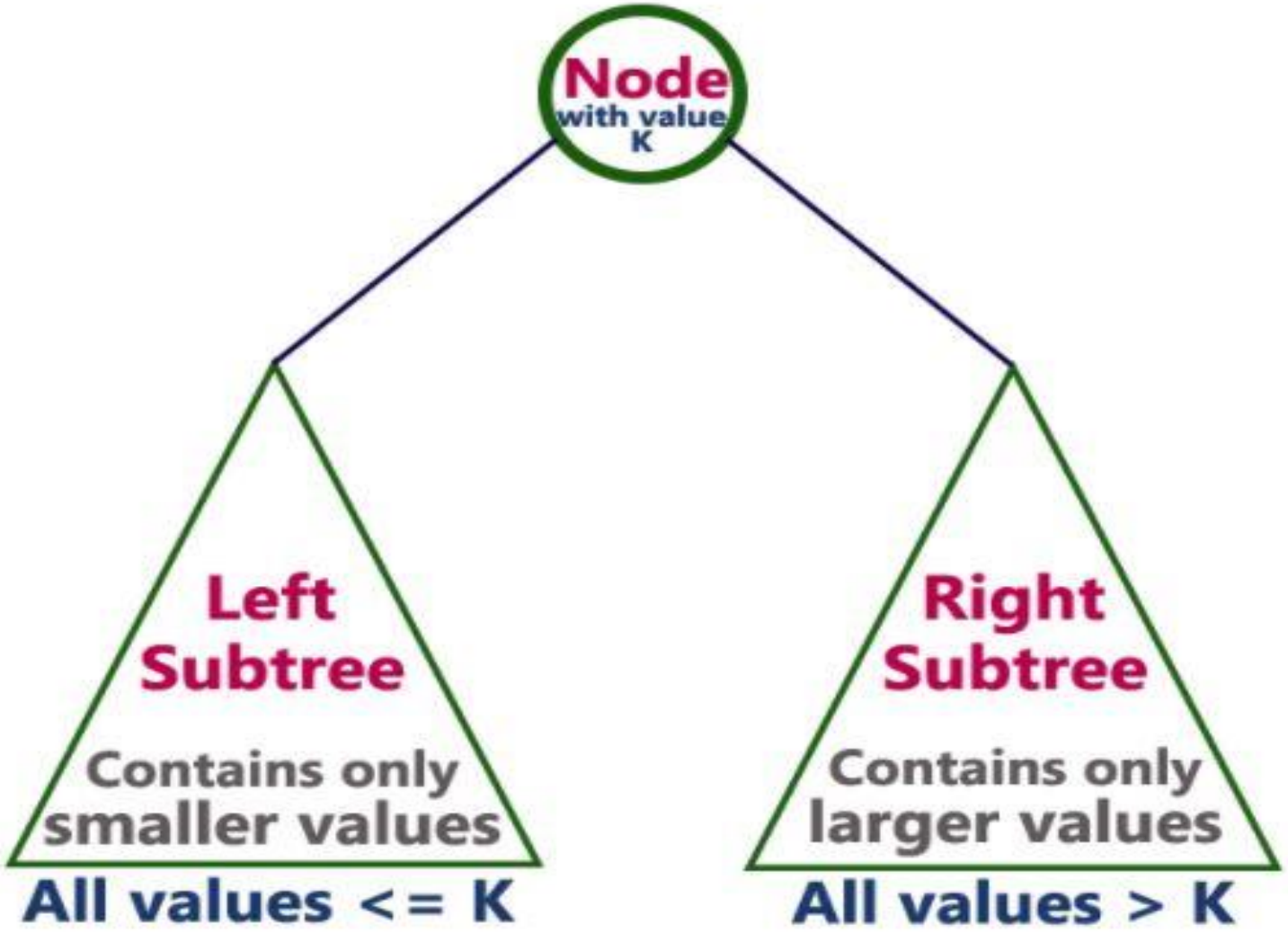
Binary search tree can be defined as follows...

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.

A binary search tree should satisfies the following condition:

- All the node in left sub tree of a node 'N' are less than the content of N.
- All the node in right sub tree of a node 'N' are greater than or equal to the content of N.
- Both the left and right sub tree of the node must be binary search tree.

Example:


Node
with value
K
Left
Subtree
Contains only
smaller values
All values <= K
Right
Subtree
Contains only
larger values
All values > K

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.

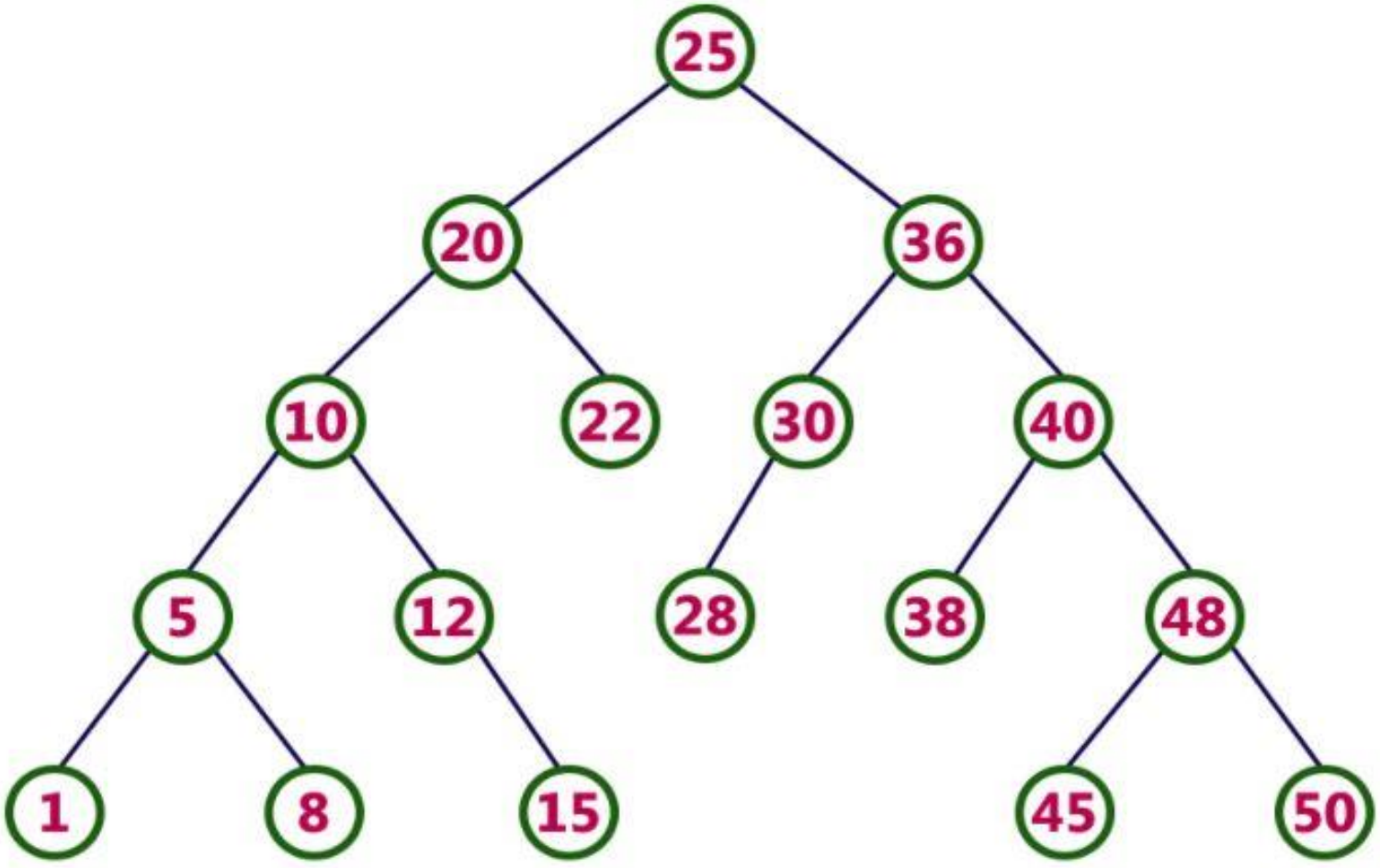


Fig: Binary Search Tree

Operations on a Binary Search Tree

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

1. Search
2. Insertion
3. Deletion

Search Operation in BST

In a binary search 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 the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6: If search element is larger, then continue the search process in right subtree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

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 set **root** to **newNode**.

Step 4: If the tree is **Not Empty**, then check whether 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. If newNode is **larger** than the node, then move to its **right** child.

Step 6: Repeat the above step until we reach to a **leaf** node (e.i., reach to NULL).

Step 7: After reaching a leaf node, then isert the newNode as **left child** if newNode is **smaller or equal** to that leaf else insert it as **right child**.

Question 1: Construct a Binary Search Tree by inserting the following sequence of numbers…

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

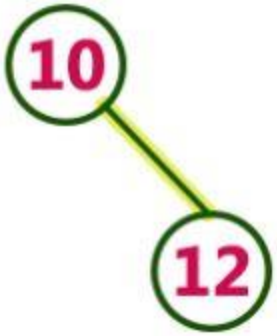
Above elements are inserted into a Binary Search Tree as follows…

Step:1

insert (10)

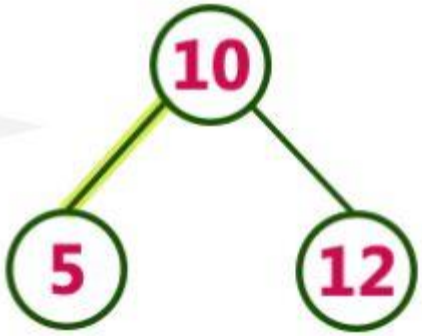
Step:2

insert (12)



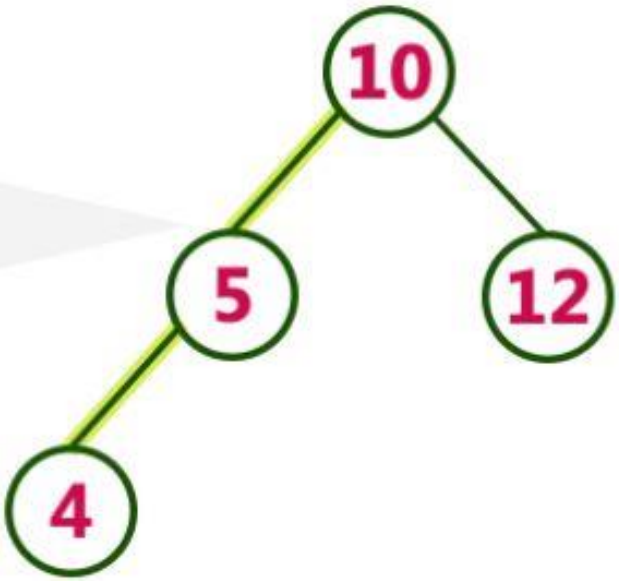
Step:3

insert (5)



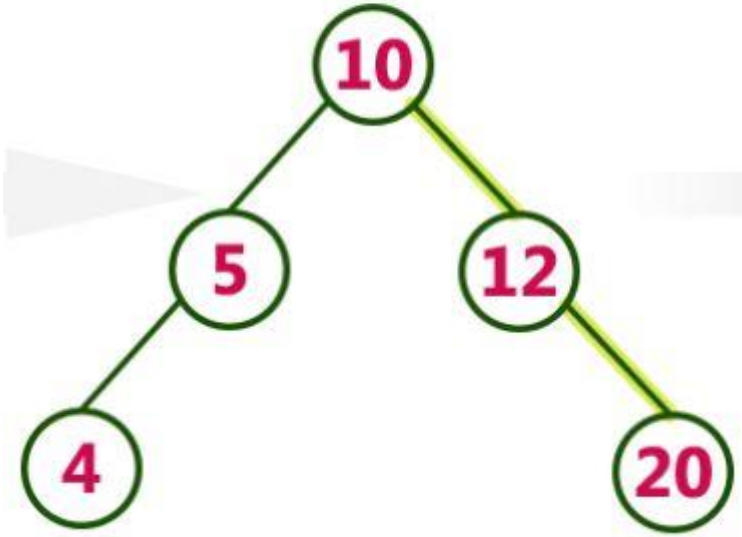
Step:4

insert (4)



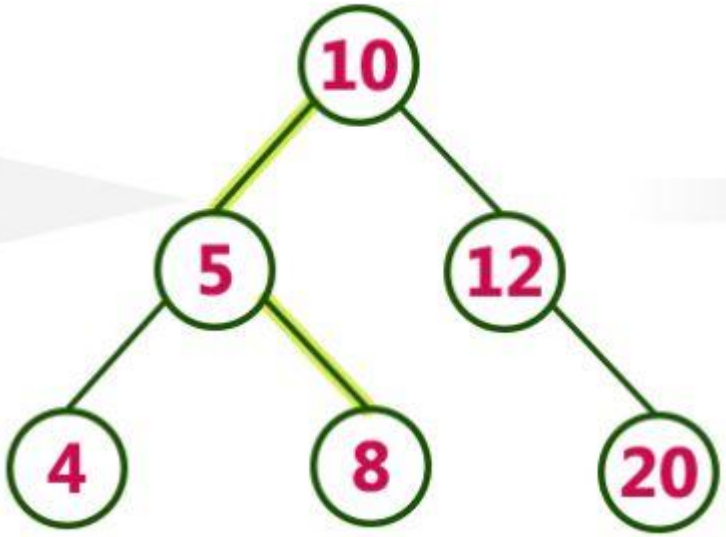
Step:5

insert (20)



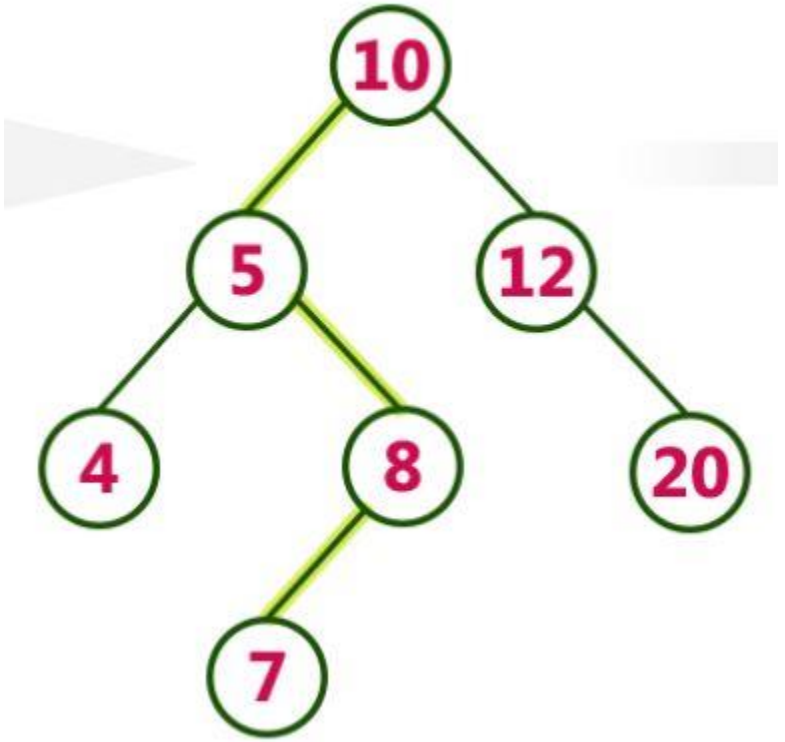
Step:6

insert (8)



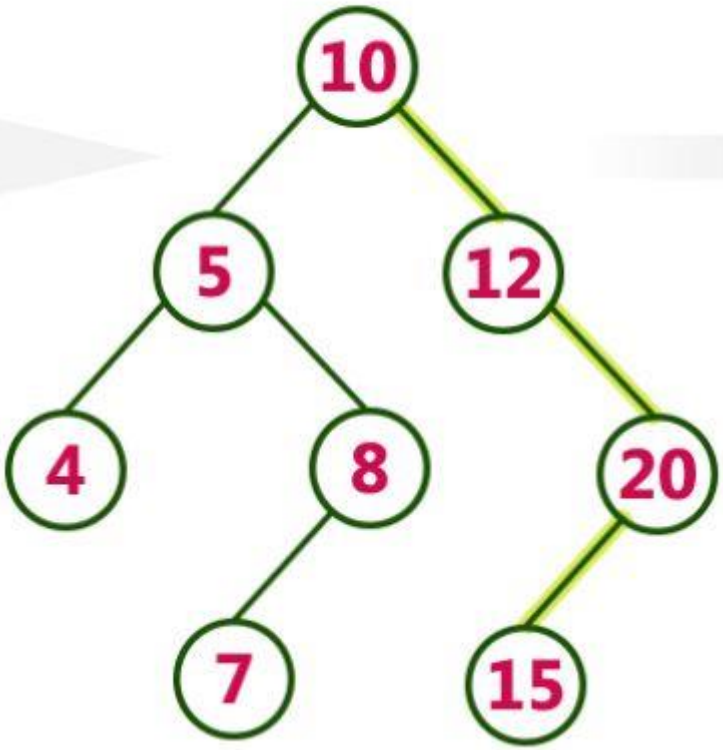
Step:7

insert (7)



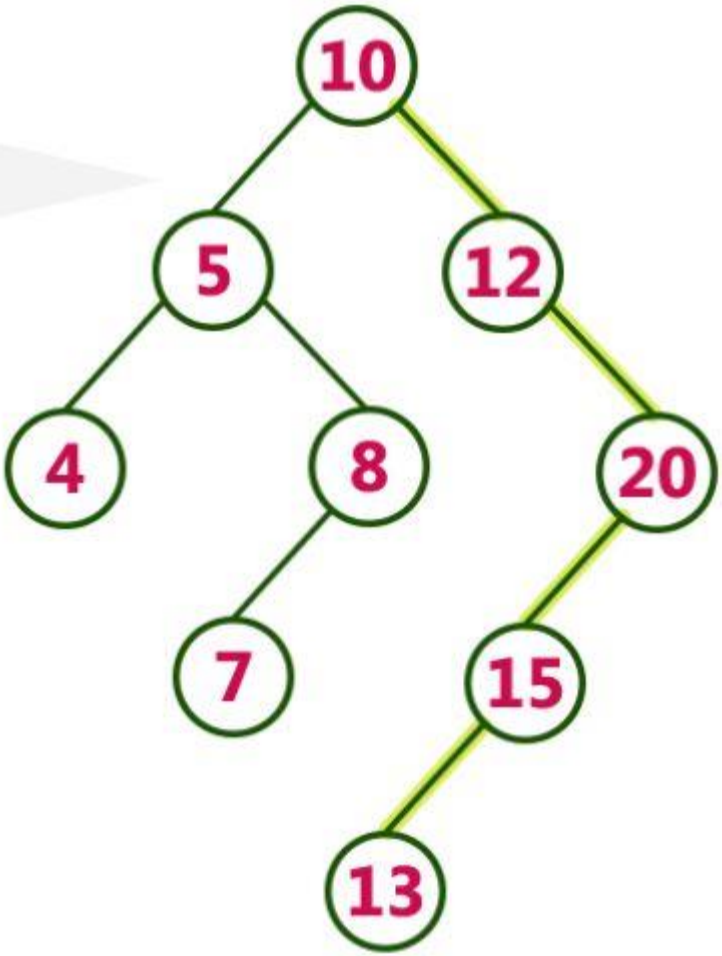
Step:8

insert (15)



Step:9

insert (13)



Question 2: Draw the binary search tree (BST) for the following sequence of data:

8,18,5,15,17,25,40,80

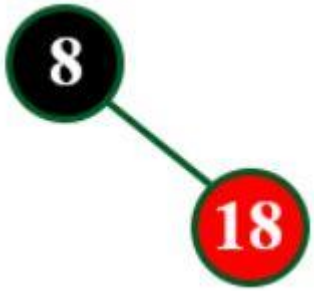
Above elements are inserted into a Binary Search Tree as follows...

Step:1

Insert (8)

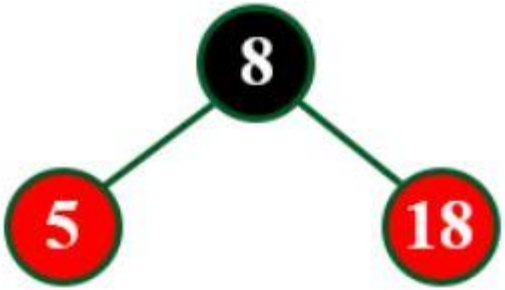
Step:2

Insert (18)



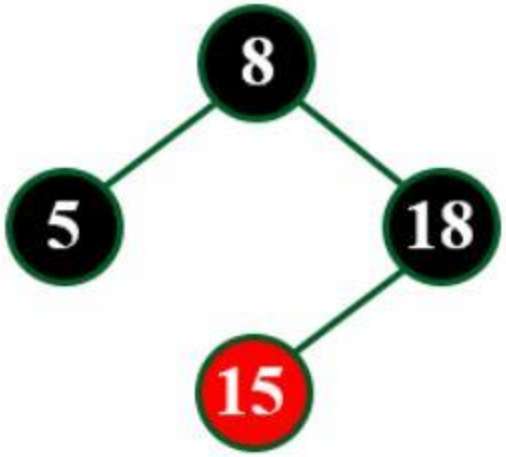
Step:3

Insert (5)



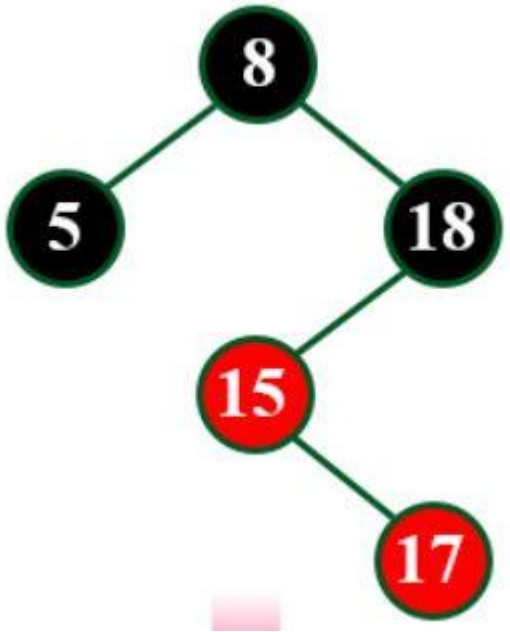
Step:4

Insert (15)



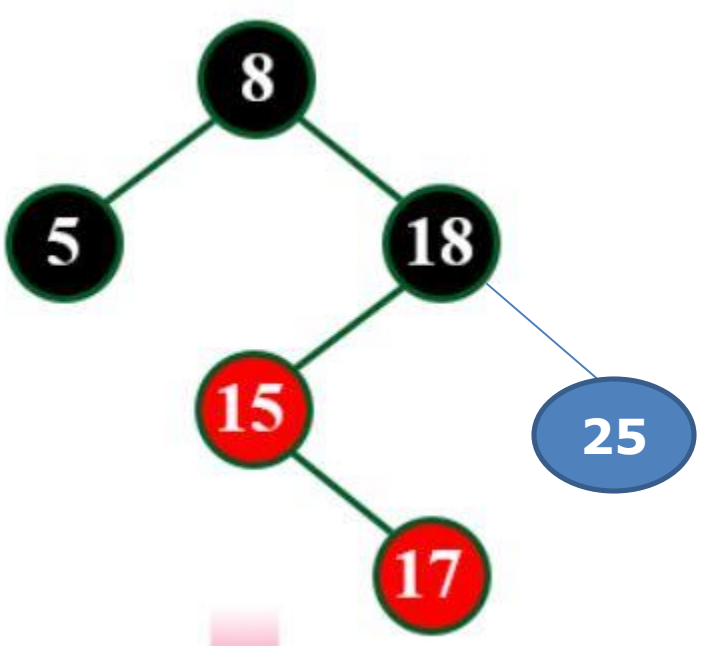
Step:5

Insert (17)



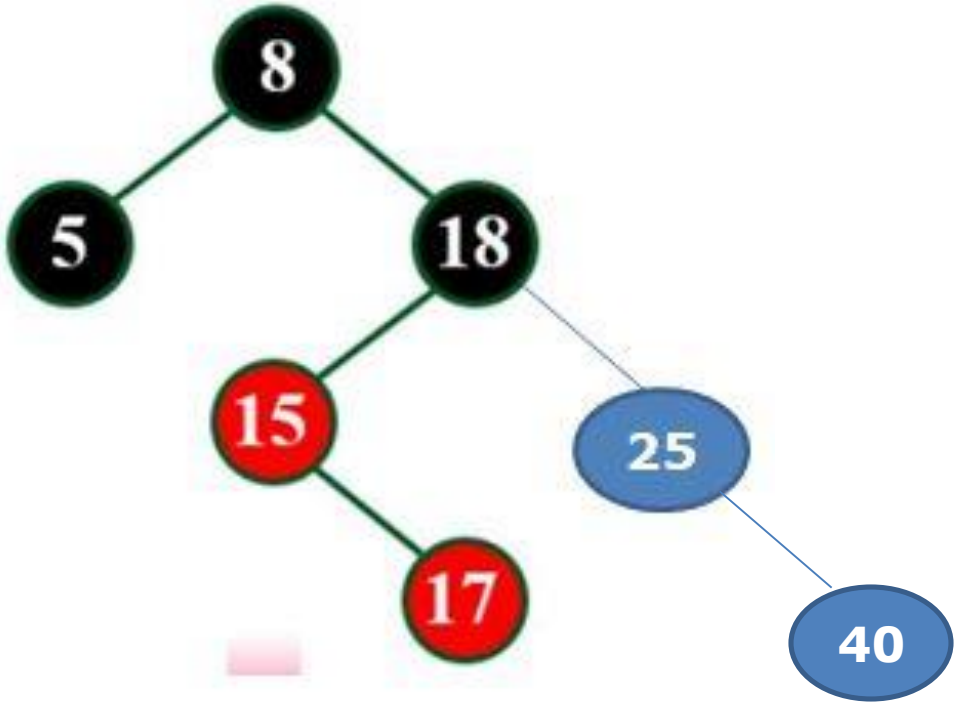
Step:6

Insert (25)



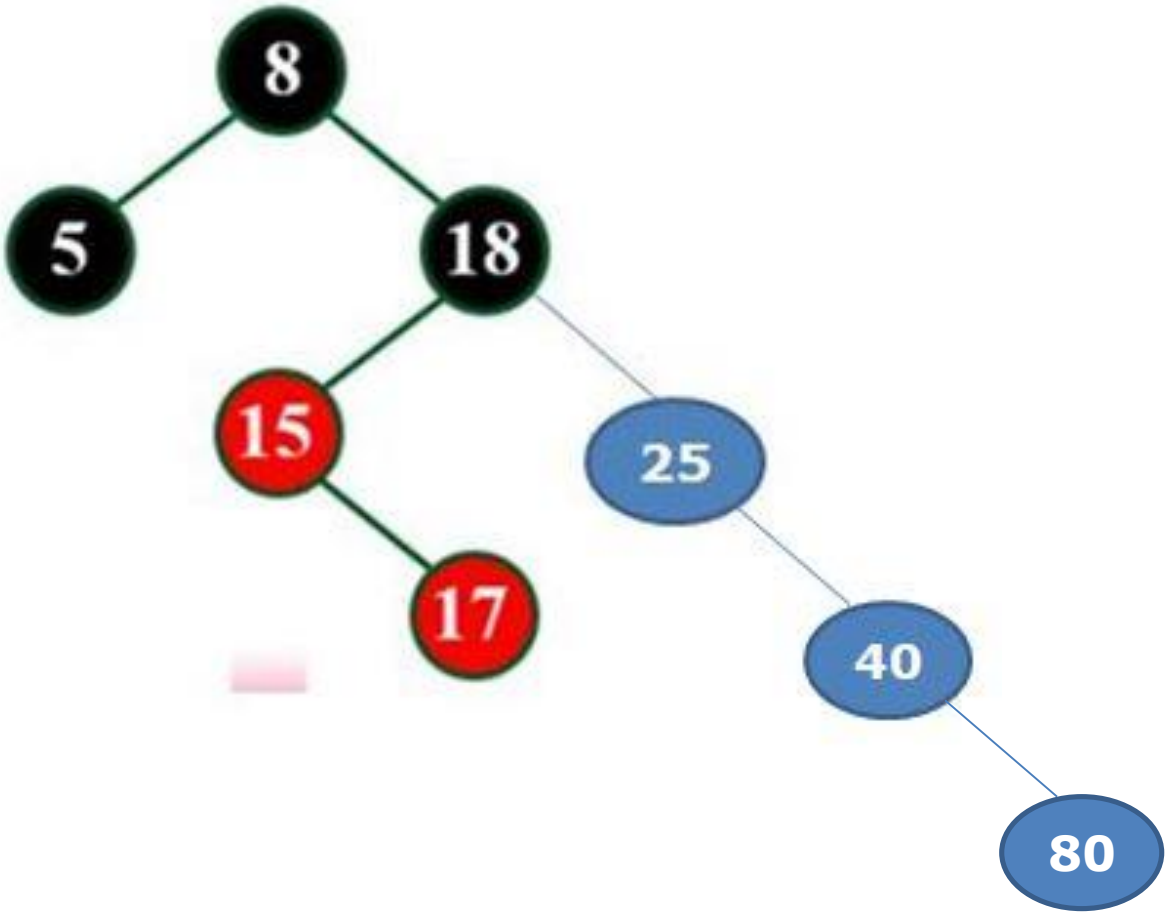
Step:7

Insert (40)



Step:8

Insert (80)



Question 3: Draw the binary search tree (BST) for the following sequence of data:

10, 30, 50, 40, 60, 100, 5, 2

Question 4: Construct a Binary Search Tree by inserting the following sequence of numbers…

14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

Question 5: Construct a Binary Search Tree by inserting the following sequence of numbers…

25, 20, 36, 10, 22, 30, 40, 28, 38, 48, 45, 50, 12, 15, 5

AVL Tree:

AVL tree is a self 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 hieghts of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as **balance factor**. The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

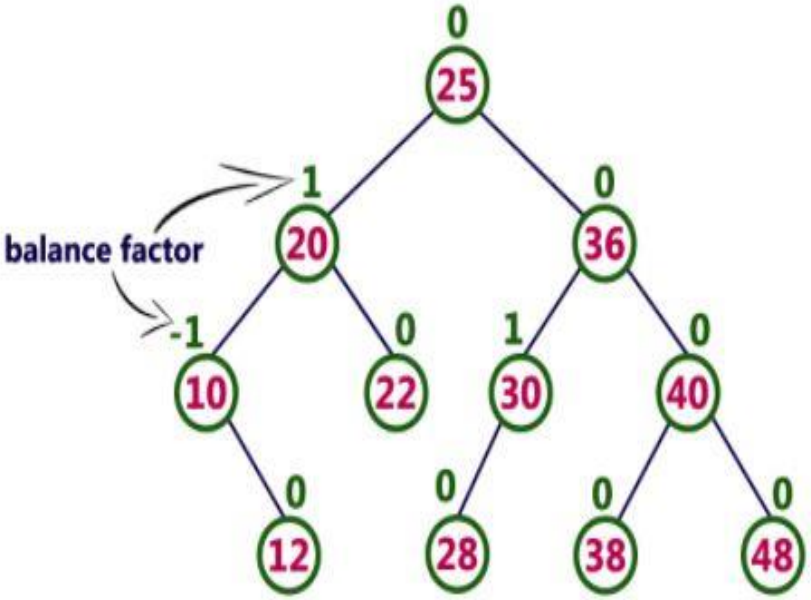
An AVL tree is defined as follows...

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

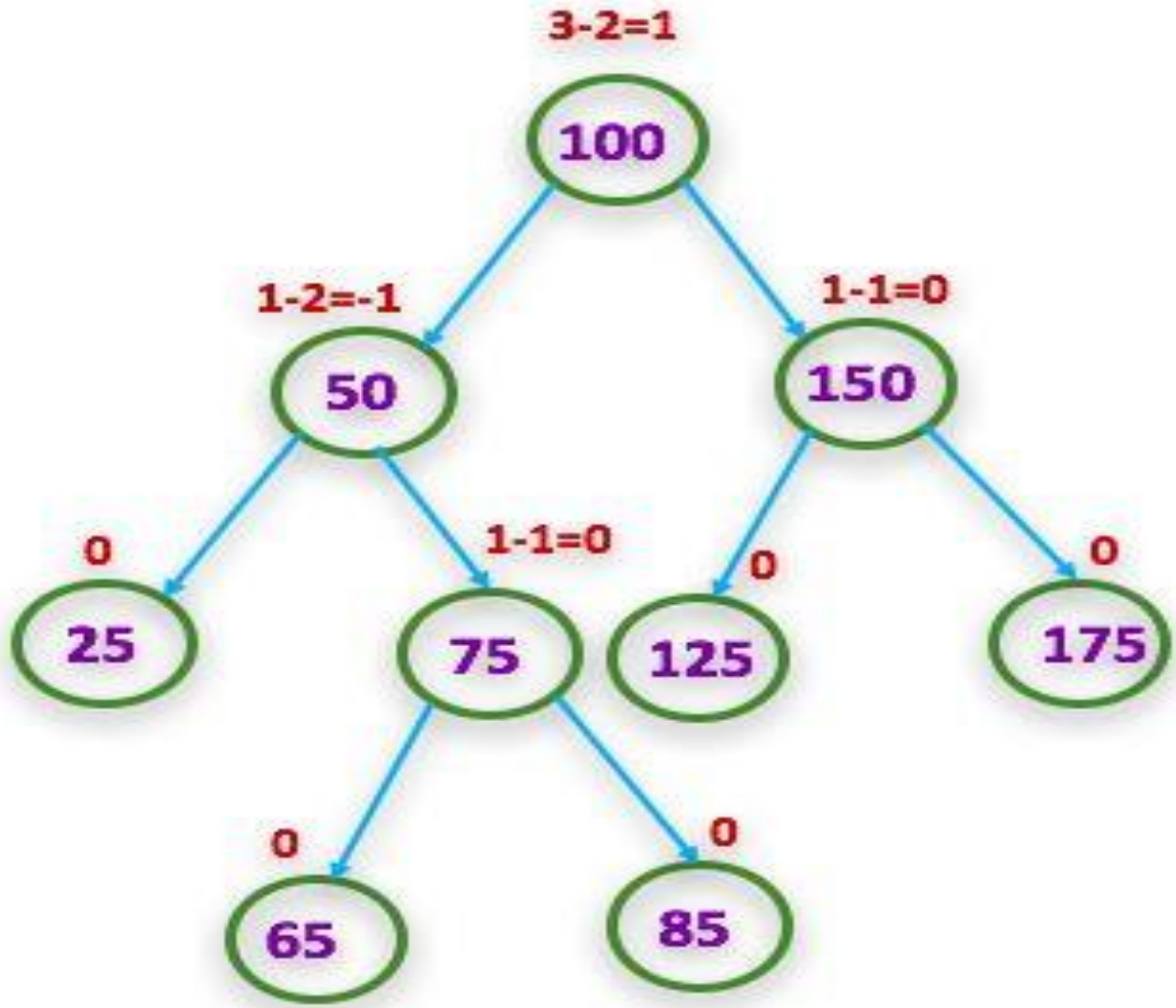
Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we are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

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example

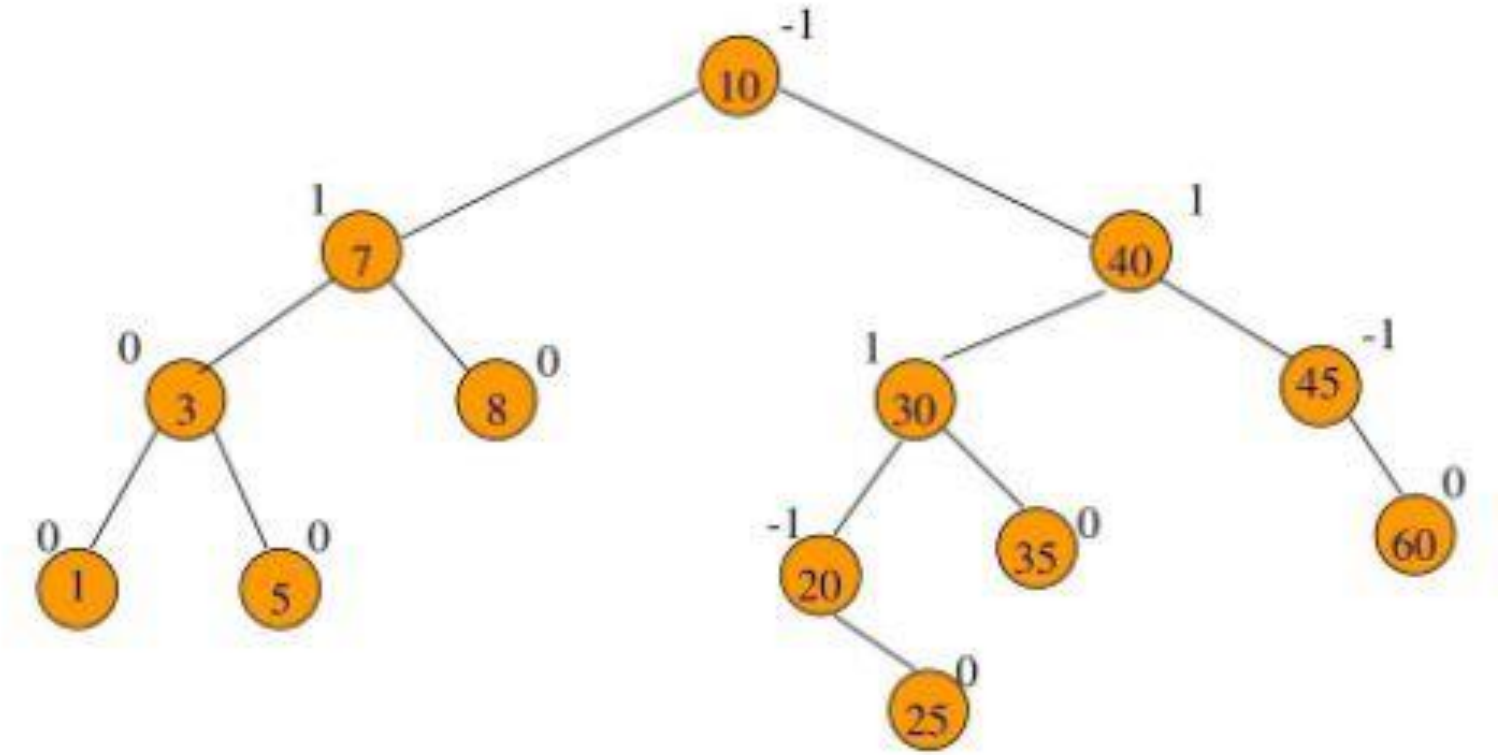


The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.



AVL Tree

Example AVL Tree



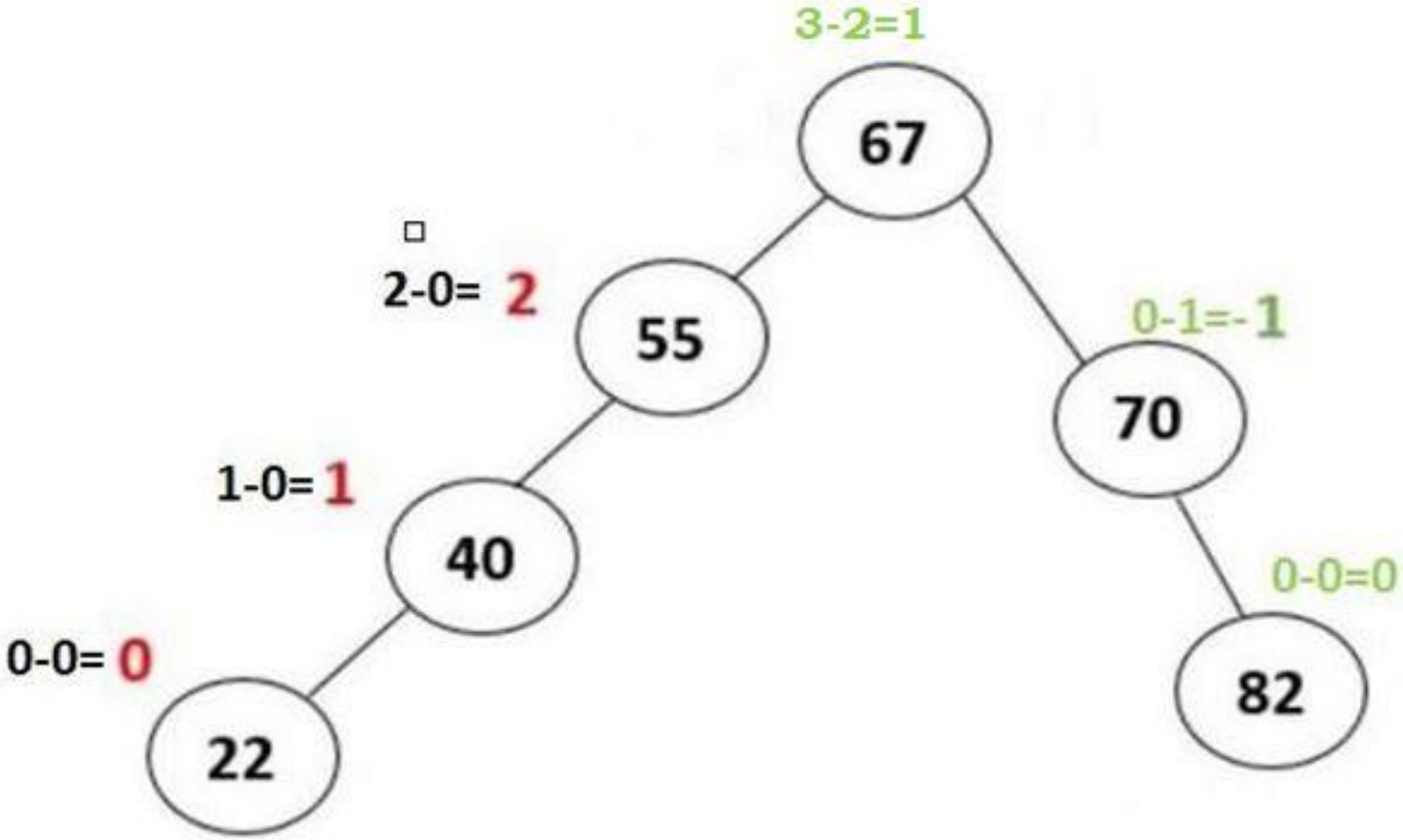


Fig: Not AVL Tree

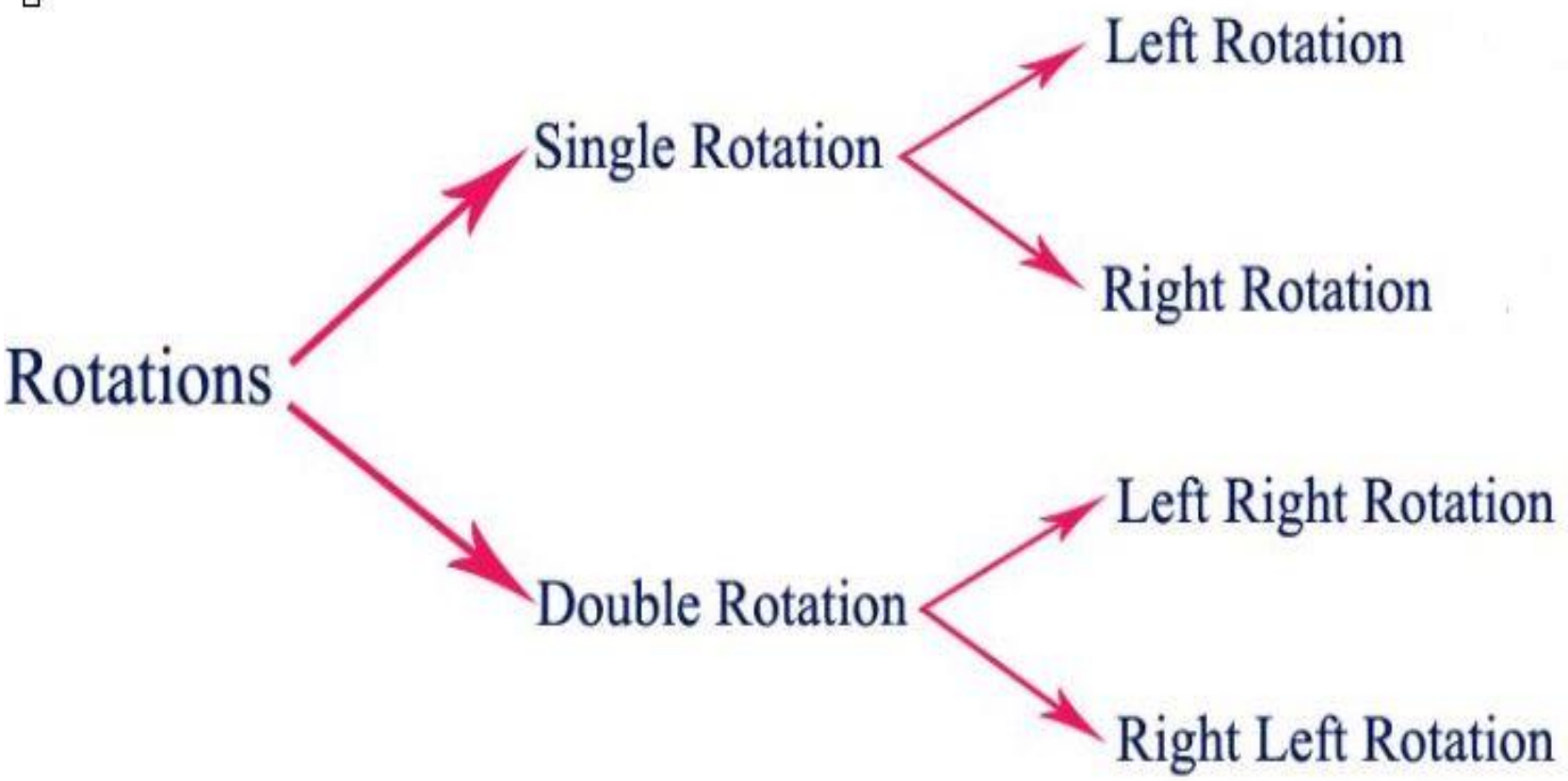
AVL Tree Rotations

In AVL tree, after performing every operation 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. We use **rotation** operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

Rotation operations are used to make a tree balanced.

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

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

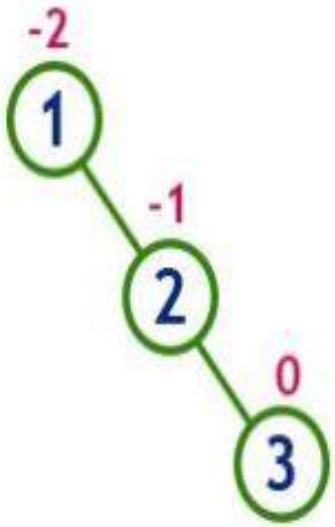


Rule:

1. Right of Right: (Rotate-Left or Left Rotation)

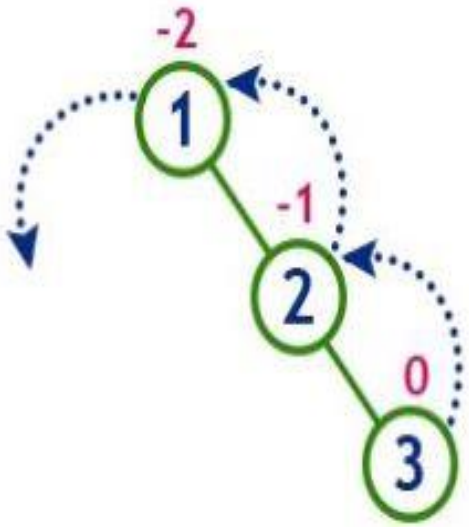
- To understand Left Rotation, Let us consider the following insertion operations into an AVL Tree…

insert 1, 2 and 3



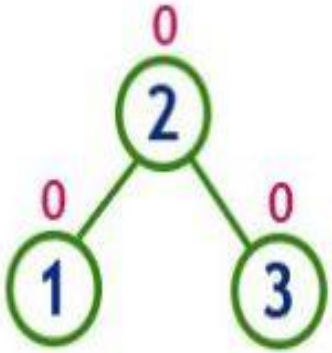
Tree is imbalanced

Right of Right



To make balanced

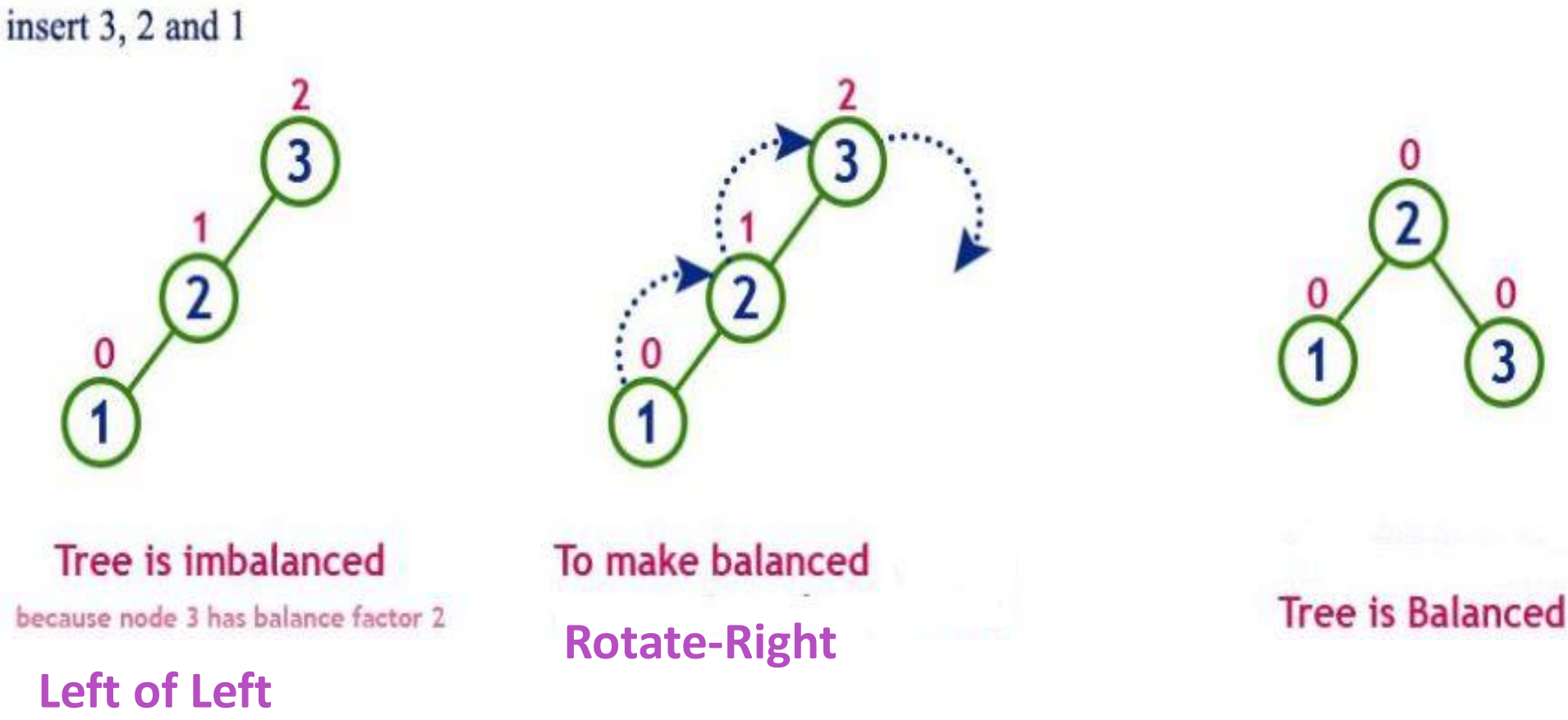
Rotate-left



Tree is Balanced

2. Left of Left: (Rotate-Right or Right Rotation)

- To understand Right Rotation, Let us consider the following insertion operations into an AVL Tree…

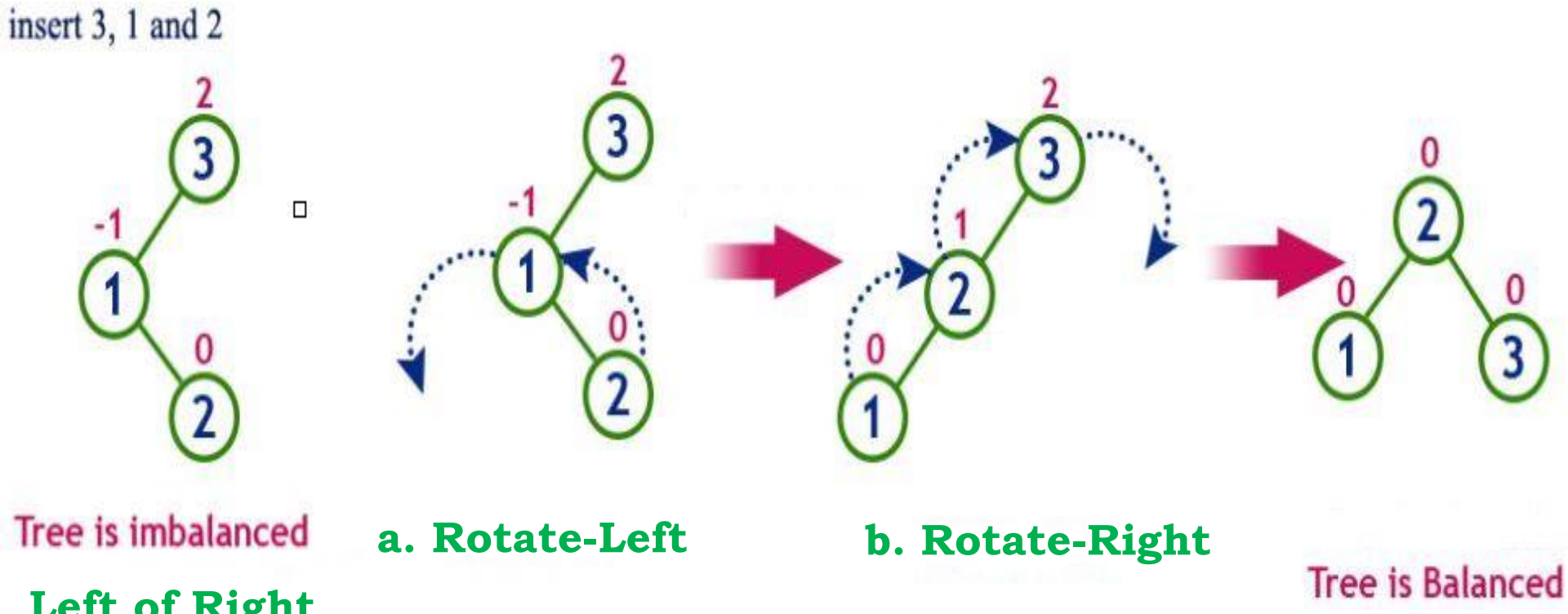


3. Left of Right (Left Right Rotation):

a. Rotate Left

b. Rotate Right

- To understand Left Right Rotation, Let us consider the following insertion operations into an AVL Tree…

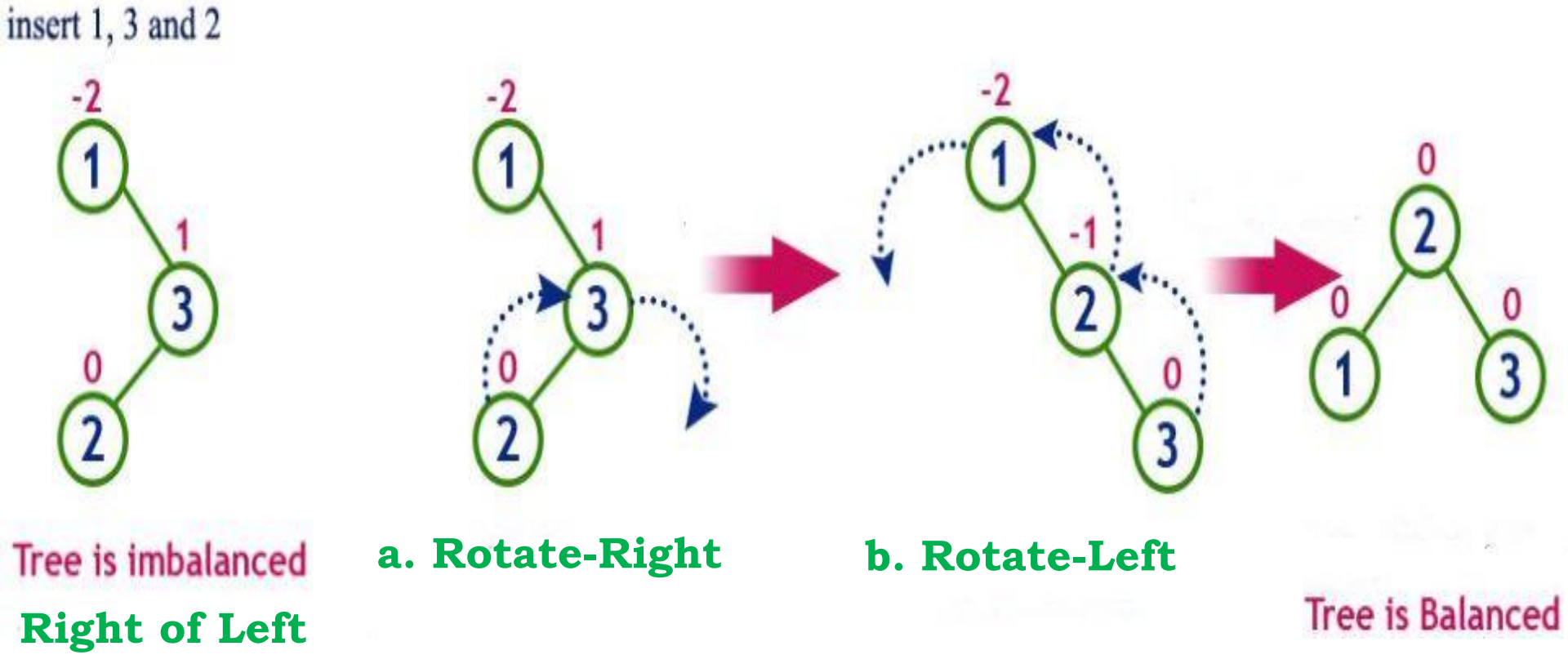


4. Right of Left (Right Left Rotation):

a. Rotate Right

b. Rotate Left

- To understand Right Left Rotation, Let us consider the following insertion operations into an AVL Tree…



Operations on an AVL Tree

The following operations are performed on an AVL tree...

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed similar to Binary search tree search operation. 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 matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6: If search element is larger, then continue the search process in right subtree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, 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 tree is said to be imbalanced. Then perform the suitable **Rotation** to make it balanced. And go for next operation.

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

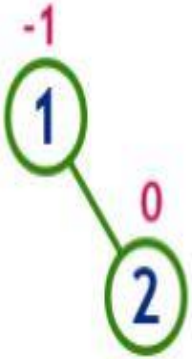
Step:1

insert 1

Tree is balanced

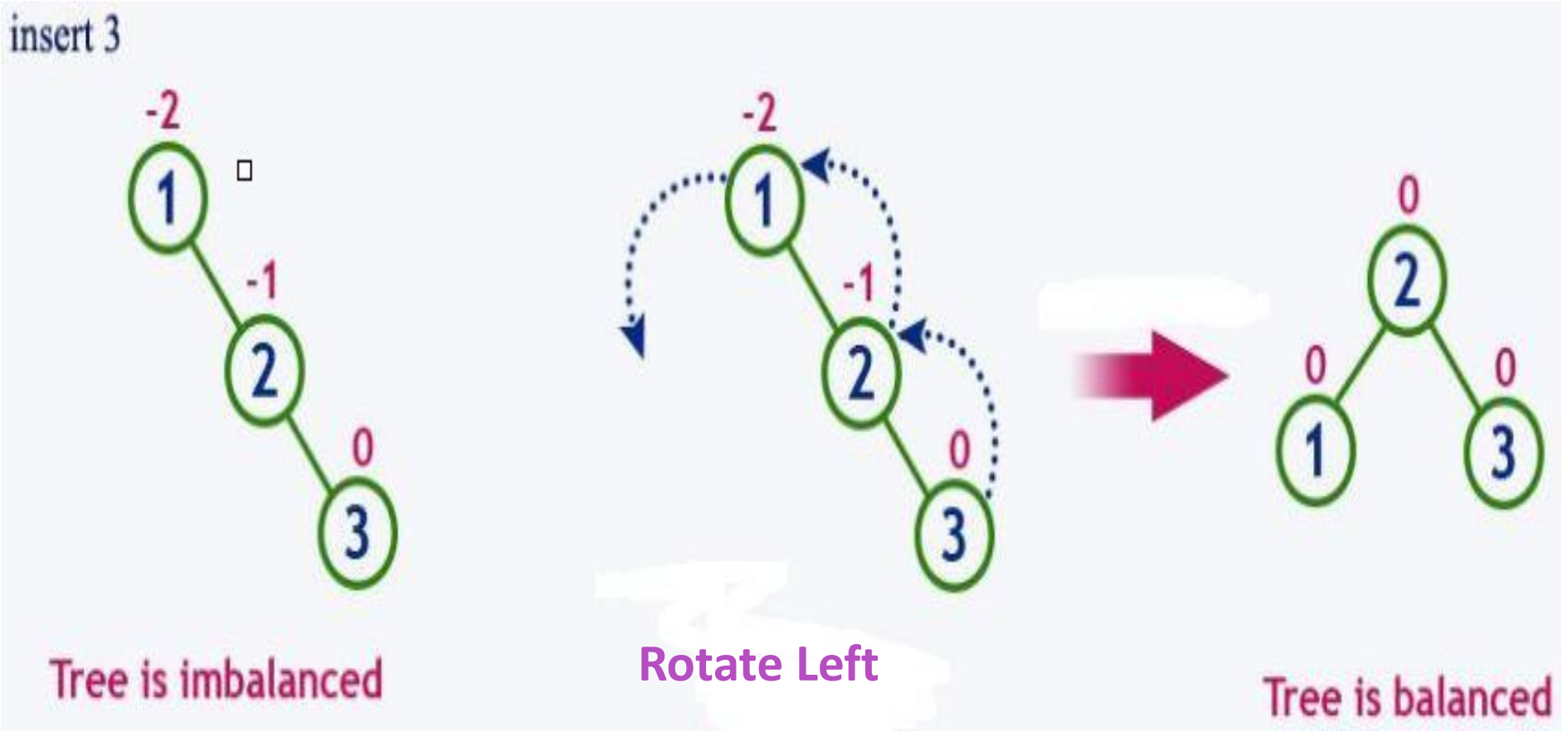
Step:2

insert 2



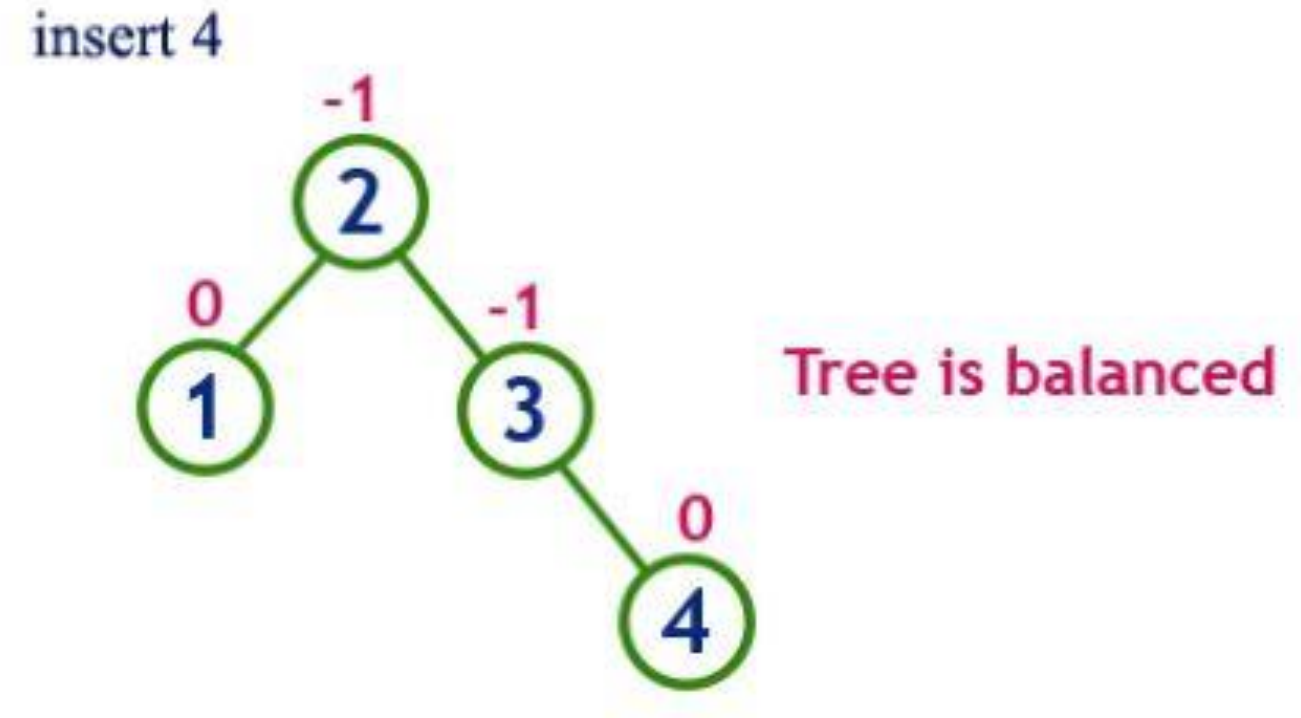
Tree is balanced

Step:3

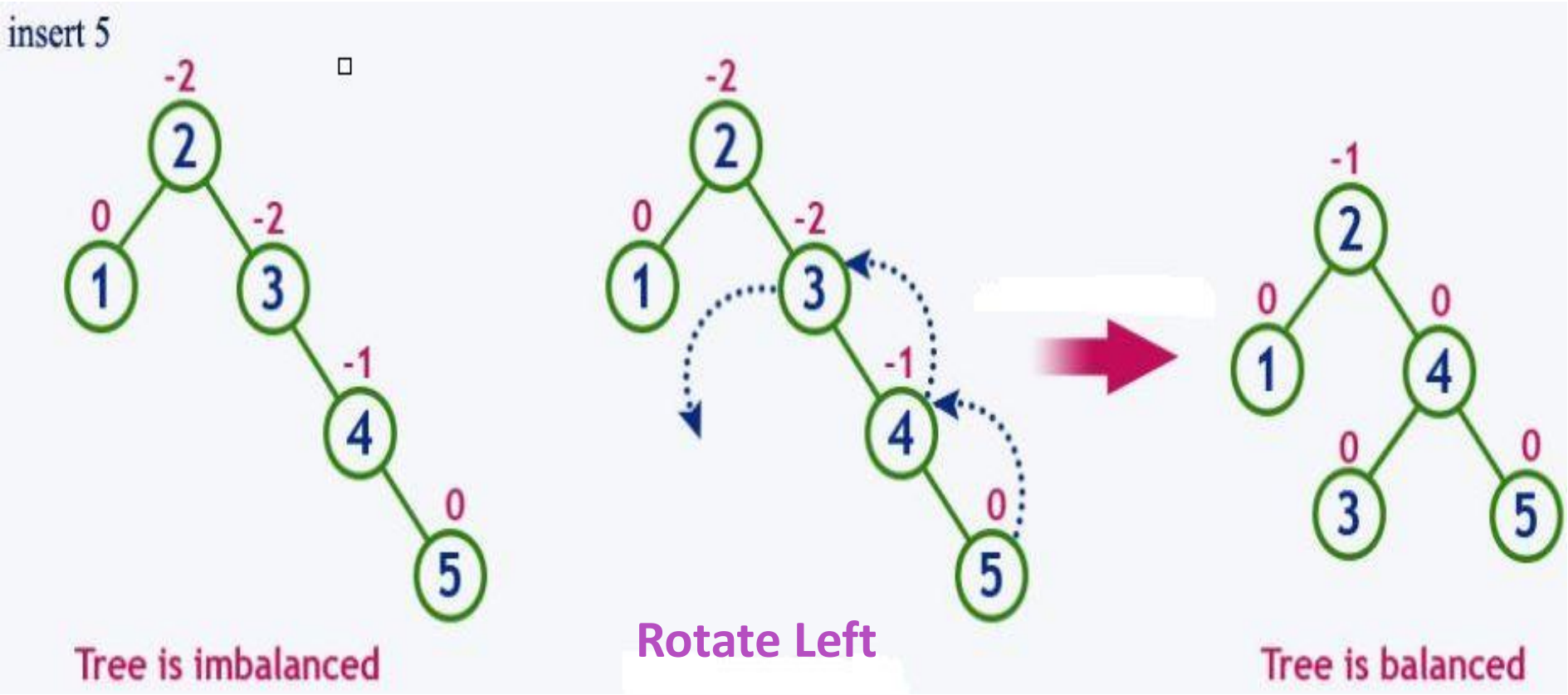


Right of Right

Step:4

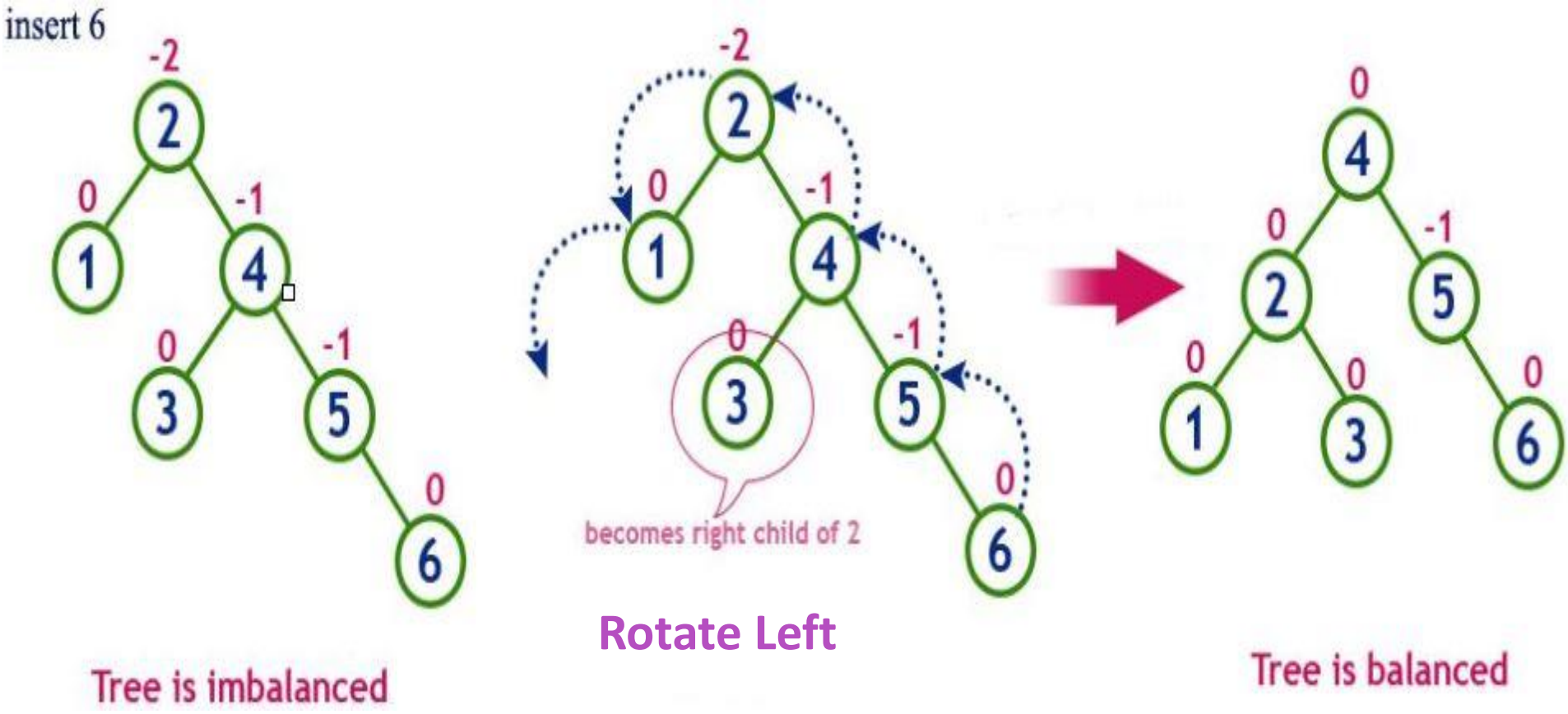


Step:5



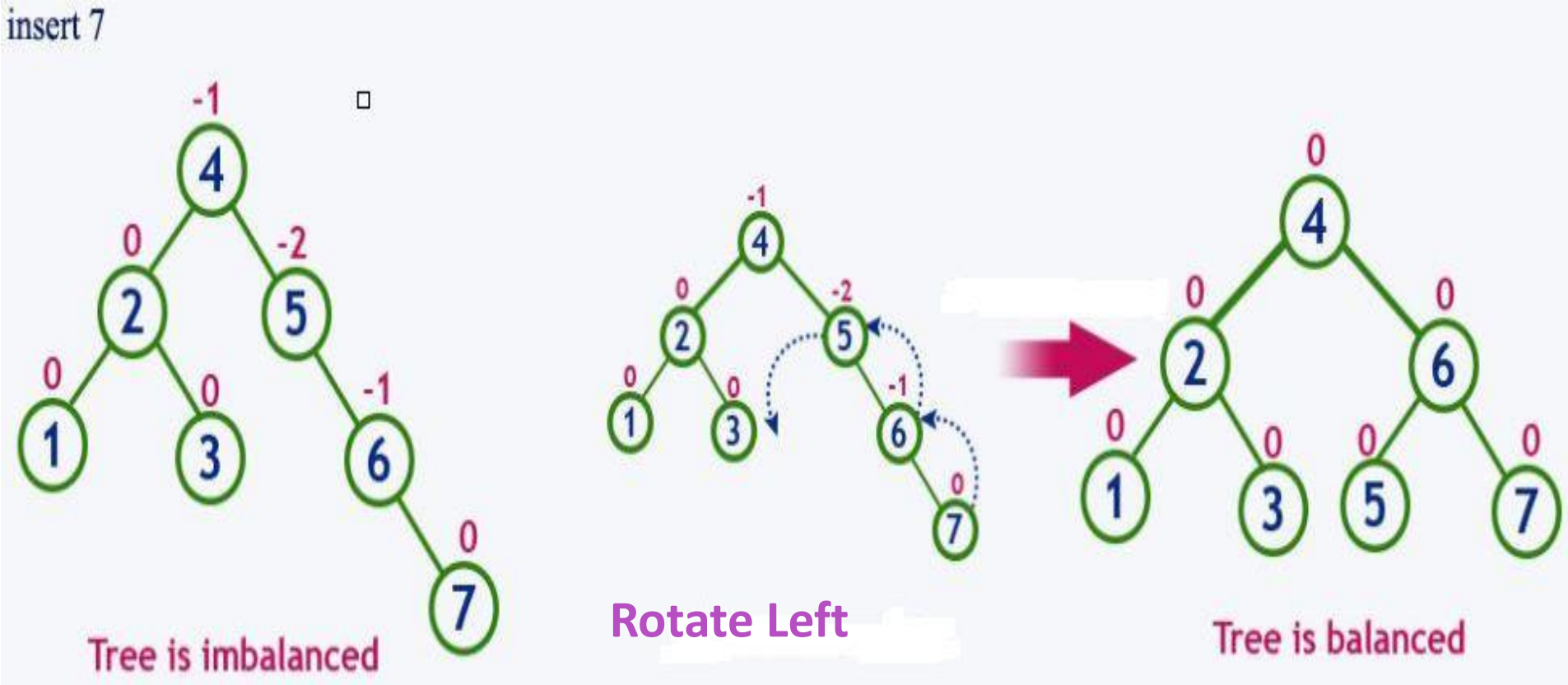
Right of Right
(Take 3,4,5)

Step:6



Right of Right
(Take 2,4,5 because 5 still have child)

Step:7



Right of Right
(Take 5,6,7)

Step:8

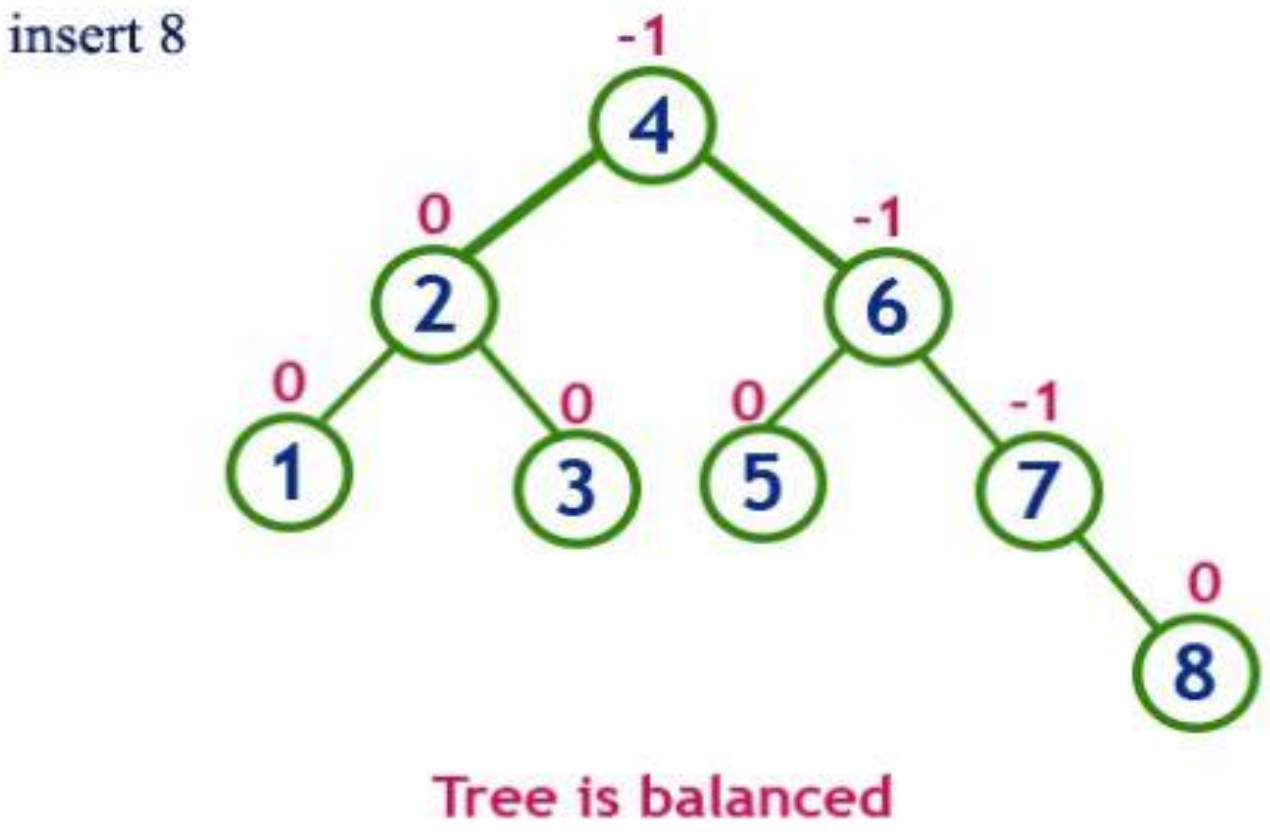


Fig: Final AVL Tree

Example:2

Construct AVL Tree for the following sequence of numbers-

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

Solution-

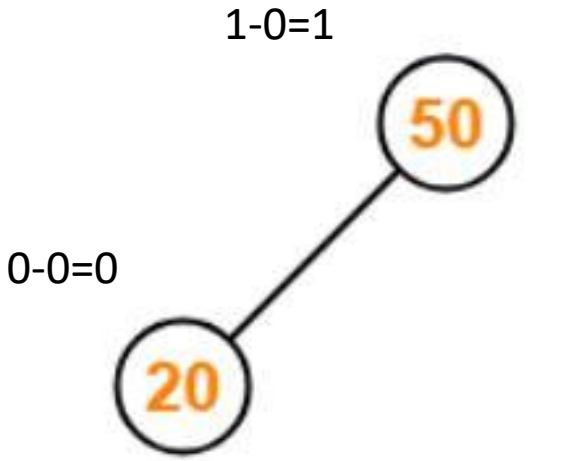
Step-01: Insert 50

0-0=0

Tree is Balanced

Step-02: Insert 20

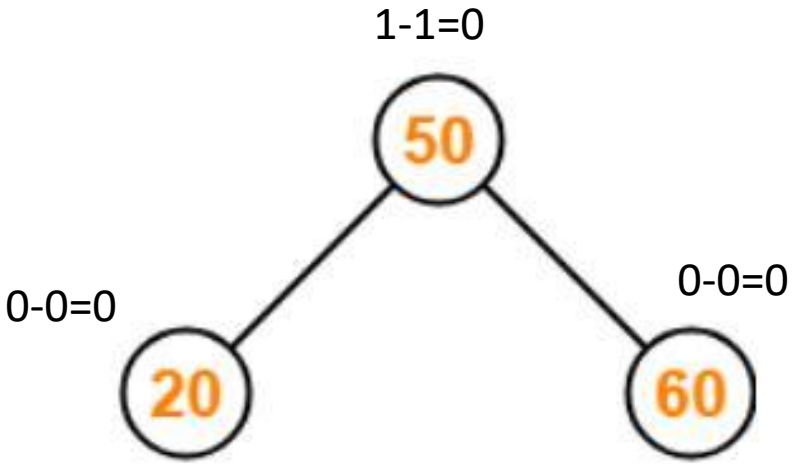
- As 20 < 50, so insert 20 in 50's left sub tree.



Tree is Balanced

Step-03: Insert 60

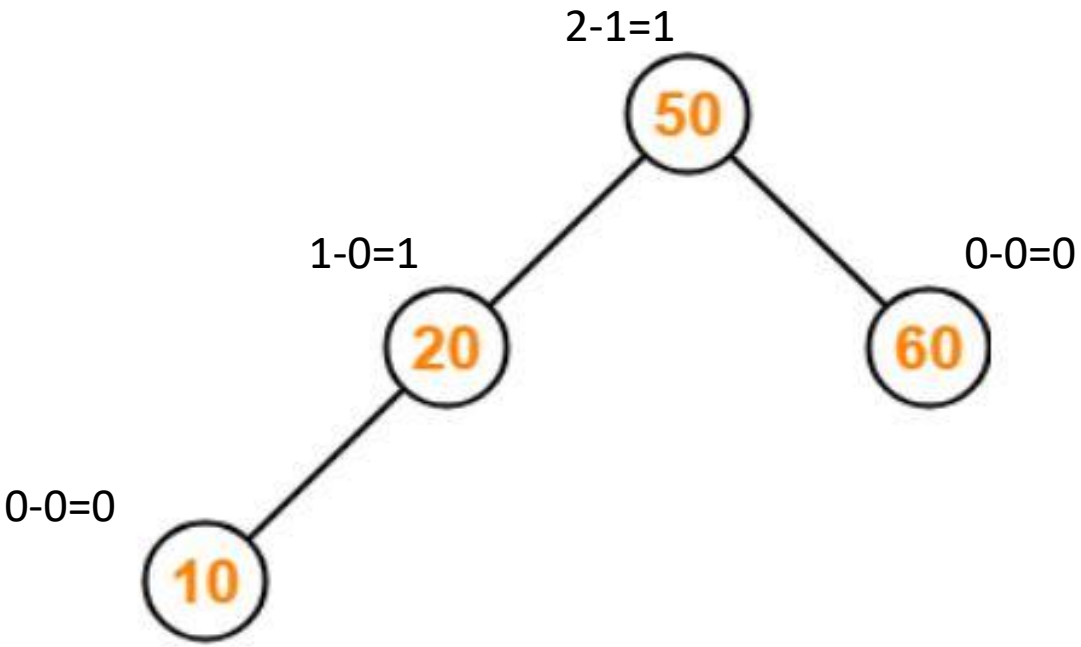
- As 60 > 50, so insert 60 in 50's right sub tree.



Tree is Balanced

Step-04: Insert 10

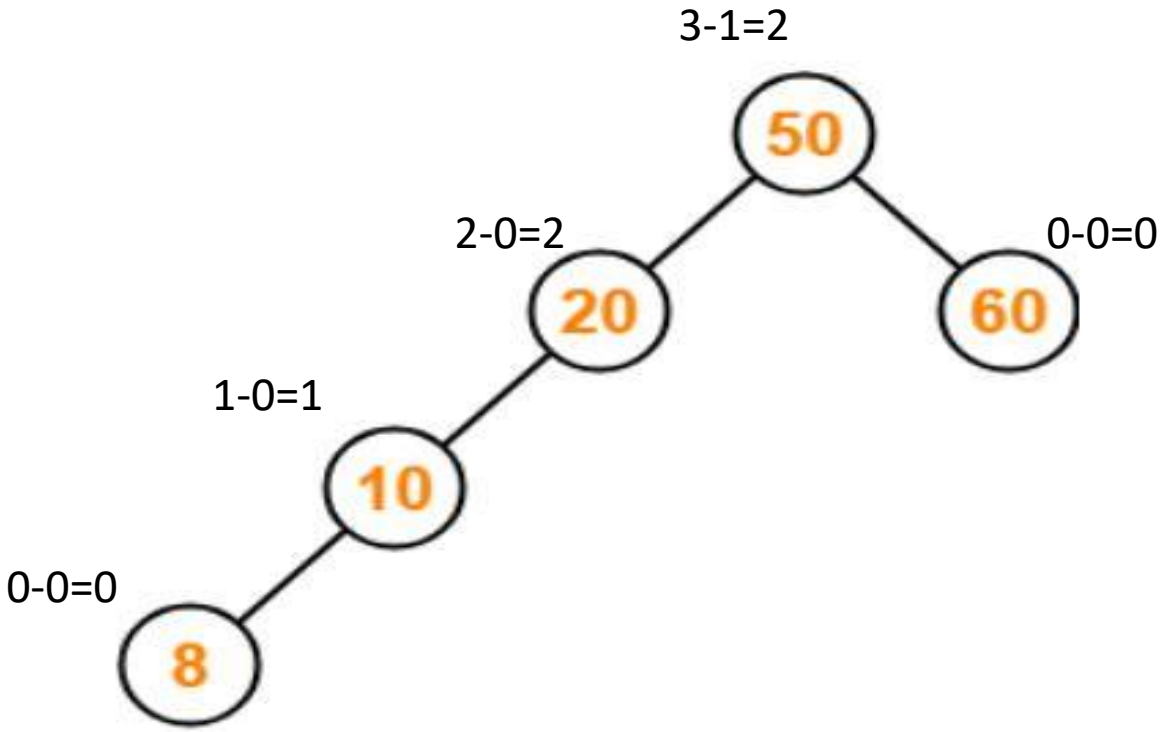
- As 10 < 50, so insert 10 in 50's left sub tree.
- As 10 < 20, so insert 10 in 20's left sub tree.



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.

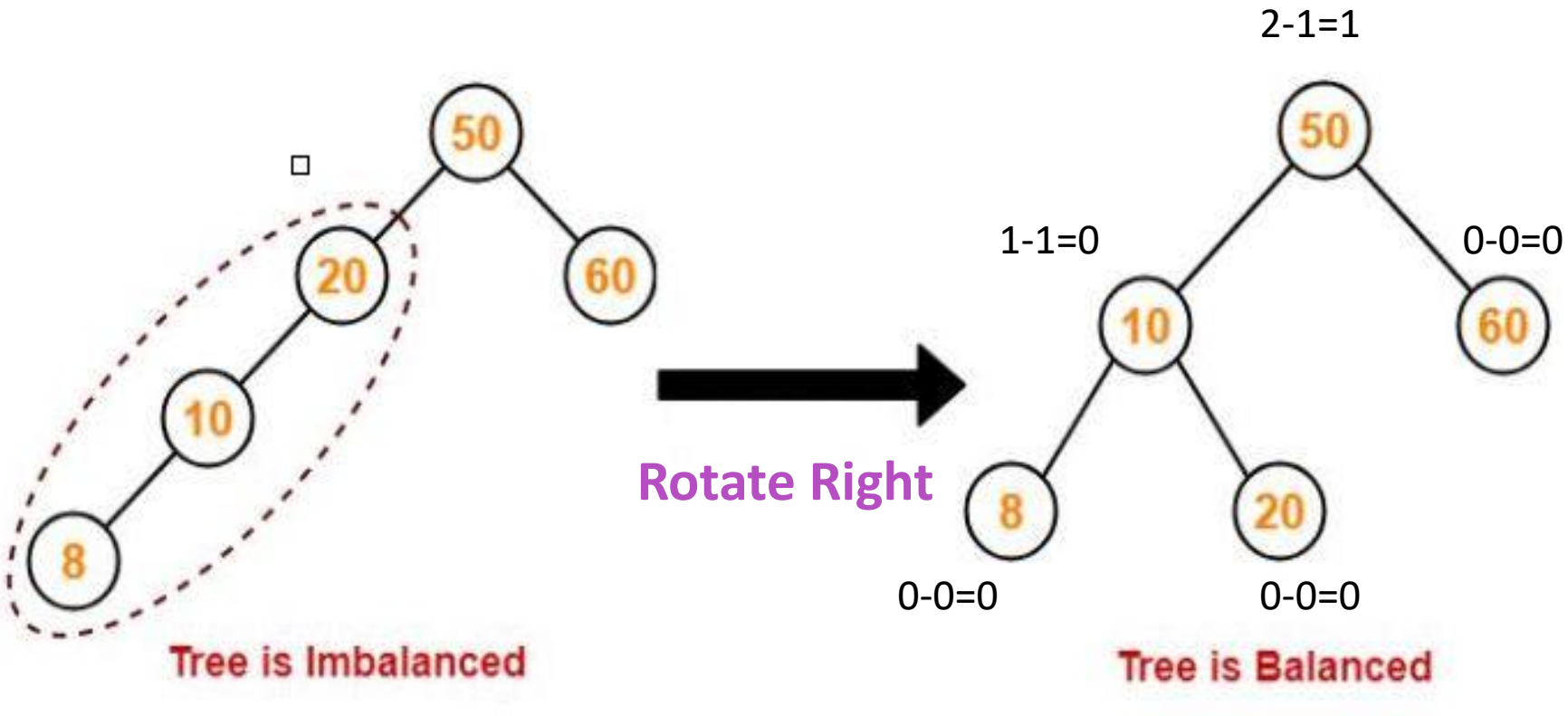


Tree is Imbalanced

To balance the tree,

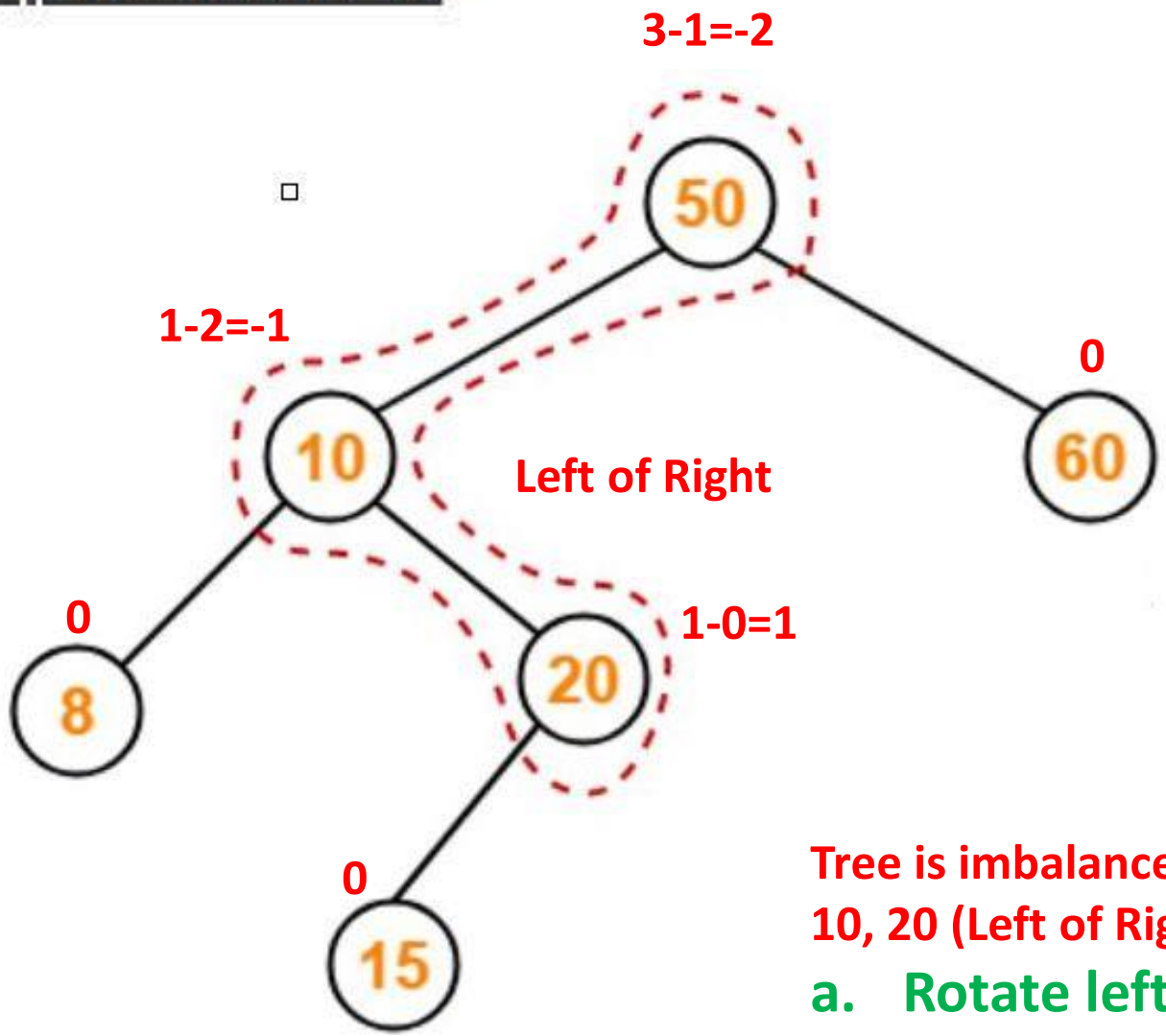
- Find the first imbalanced node on the path from the newly inserted node (node 8) to the root node.
- The first imbalanced node is node 20.
- Now, count three nodes from node 20 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-



Left of Left

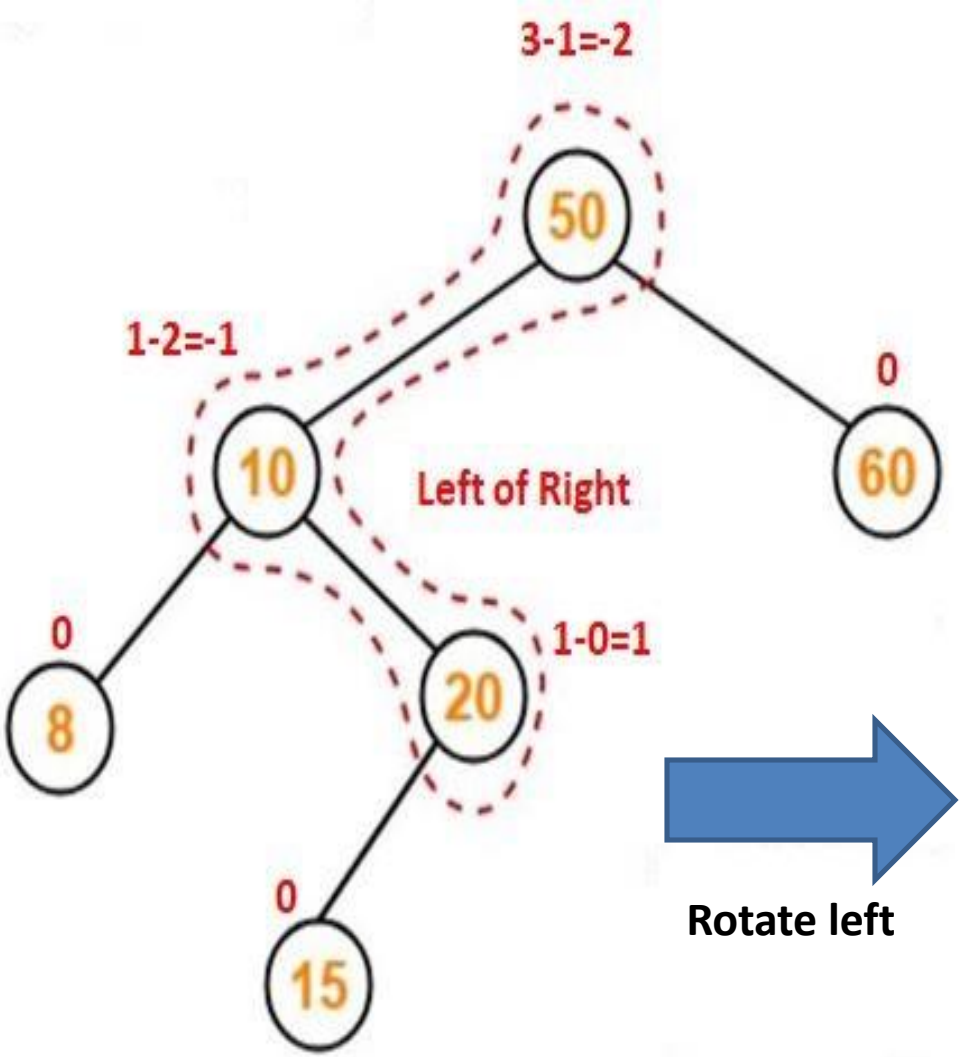
Step-06: Insert 15



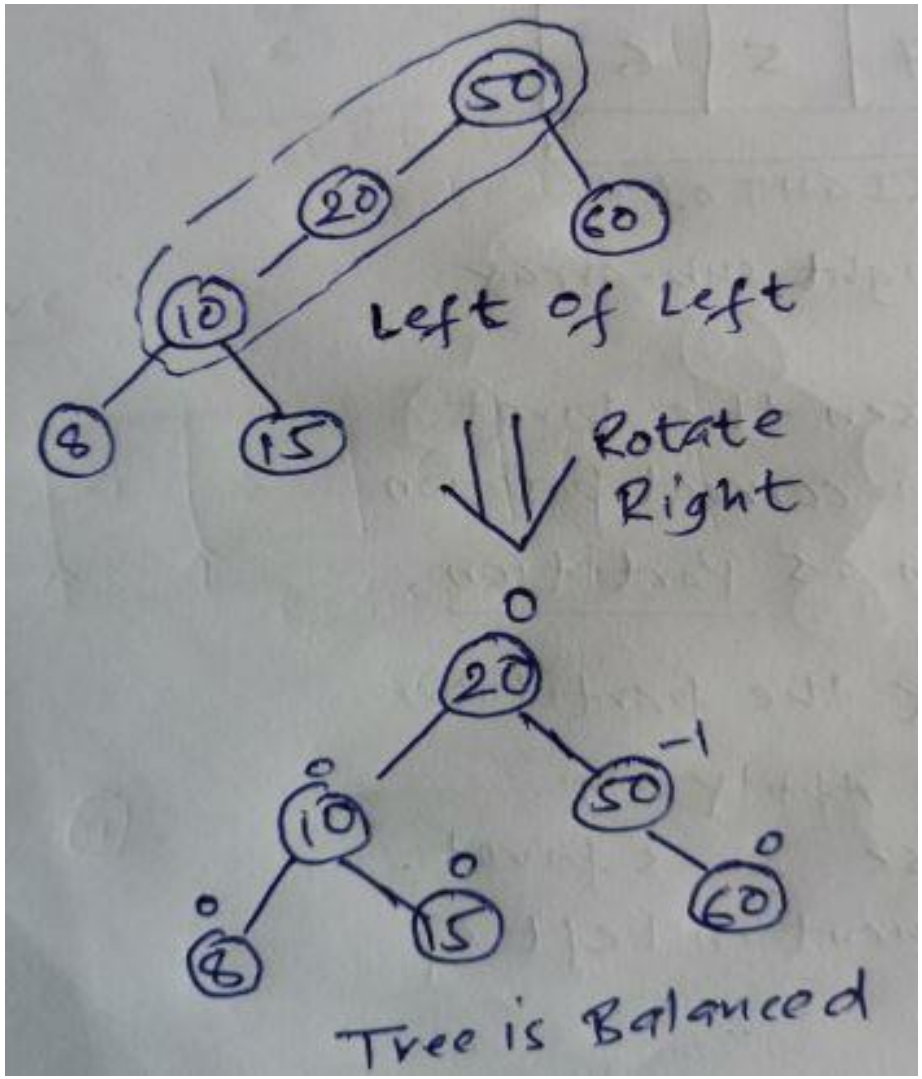
Tree is imbalanced. i.e in node 50, 10, 20 (Left of Right). So

a. Rotate left
b. Rotate right

Tree is Imbalanced

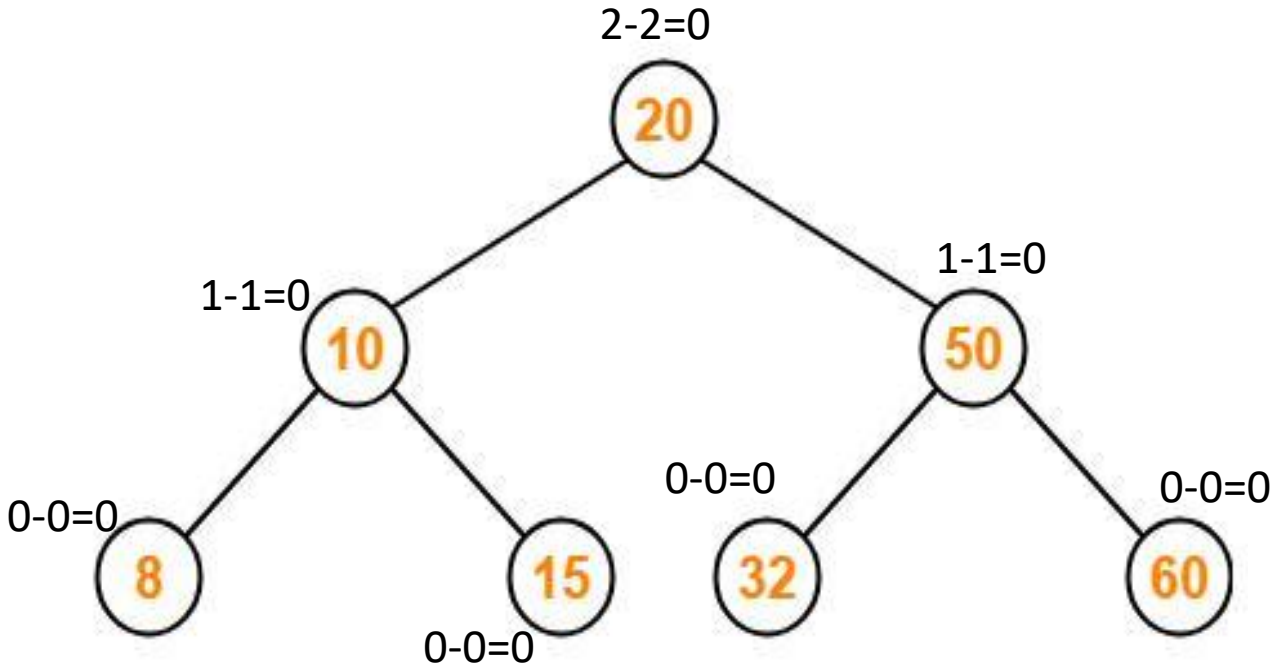

3-1=-2
50
1-2=-1
10
Left of Right
0
60
0
8
1-0=1
20
0
15

Rotate left


50
20
60
10
Left of Left
8
15
Rotate
Right
0
20
-1
10
50
0
8
15
60
Tree is Balanced

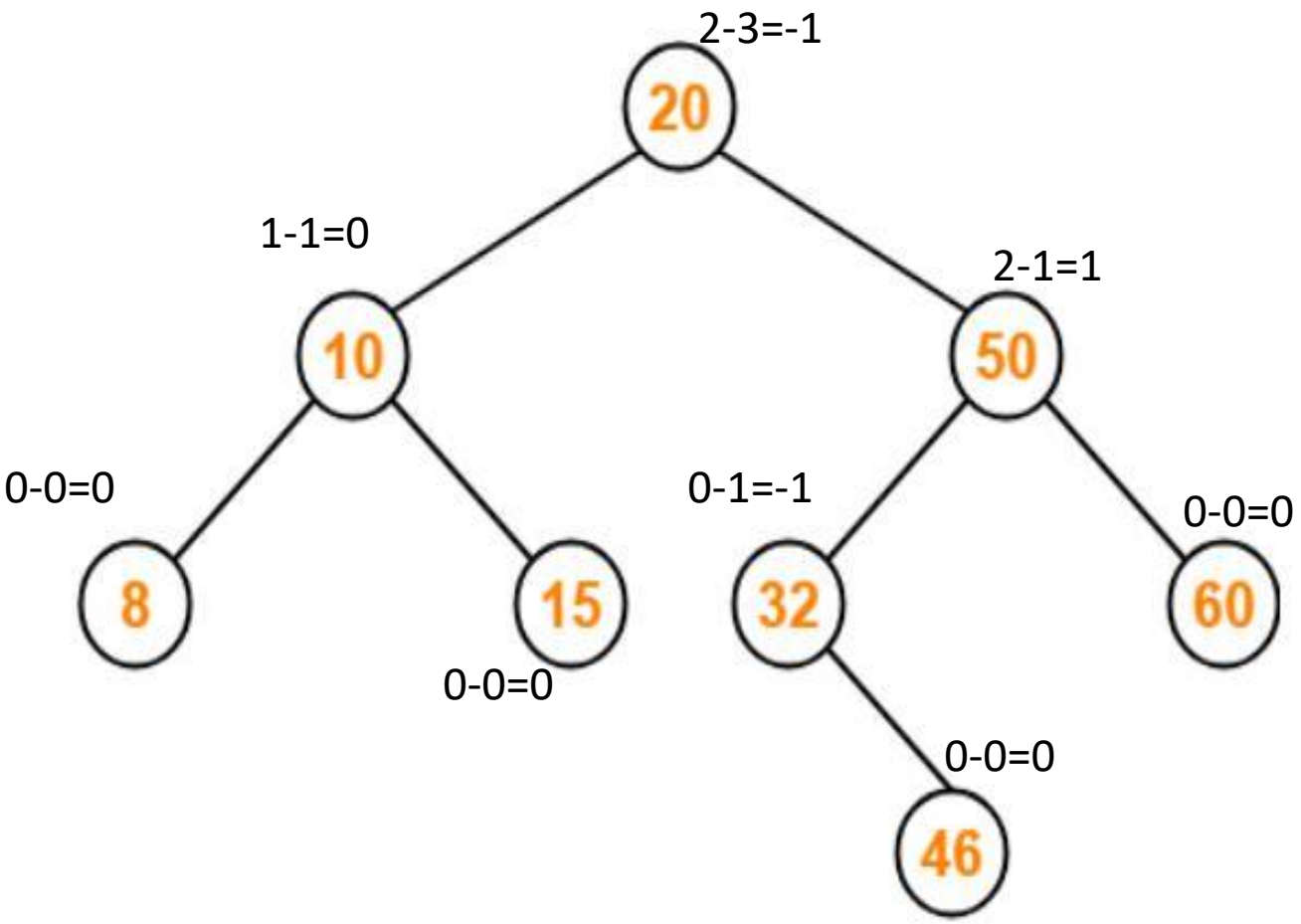
Step-07: Insert 32

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



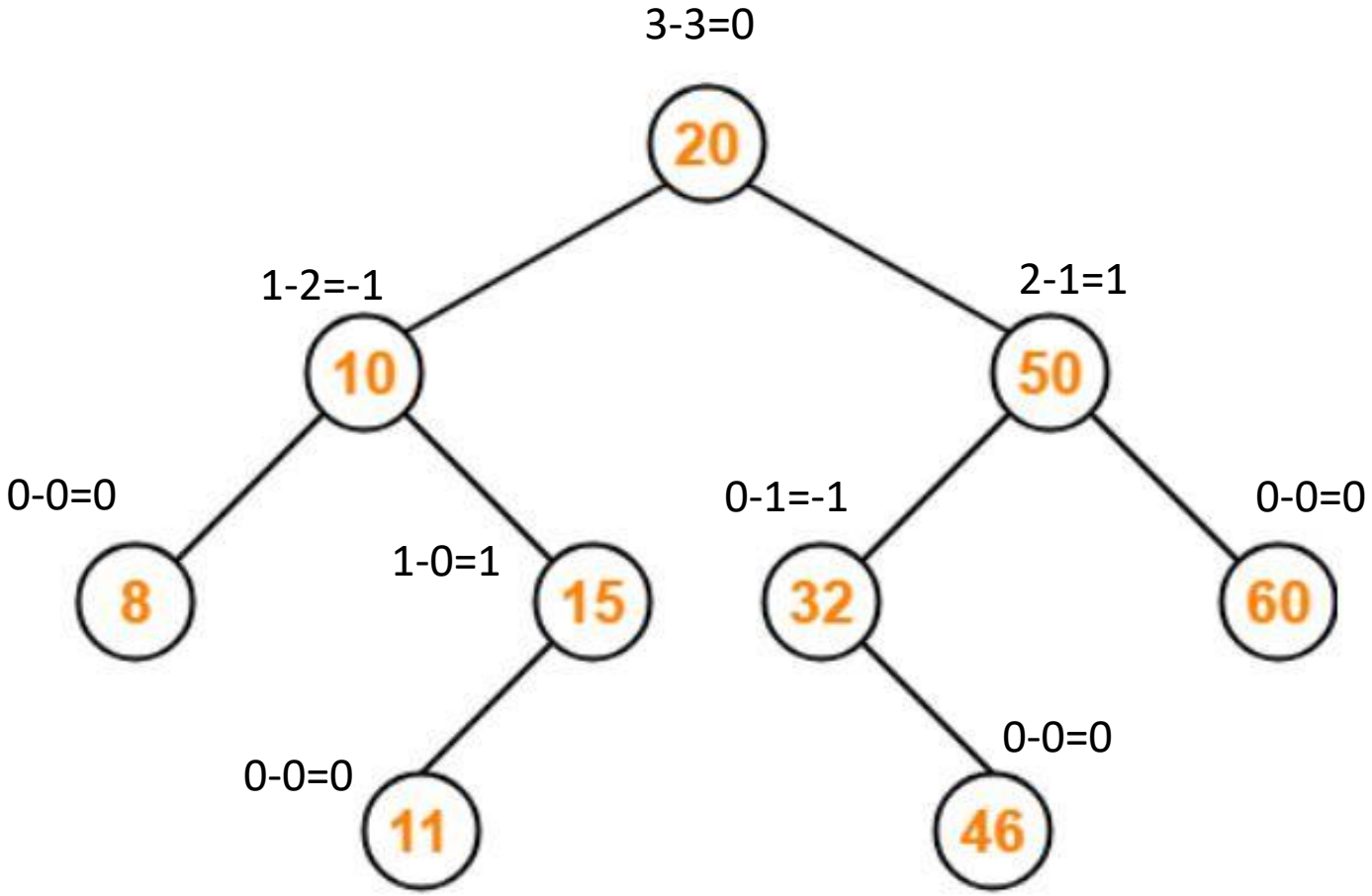
Tree is Balanced

Step-08: Insert 46



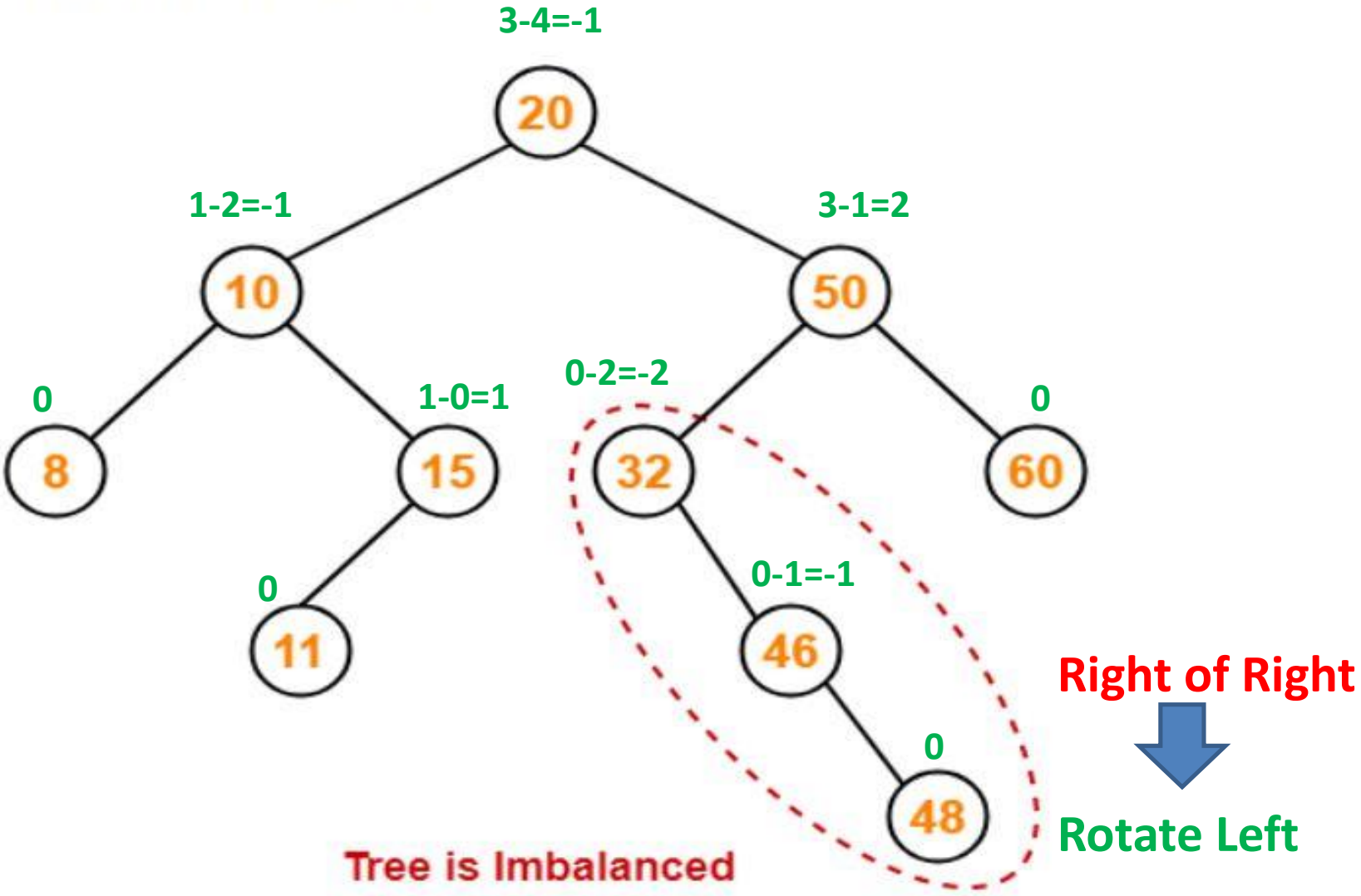
Tree is Balanced

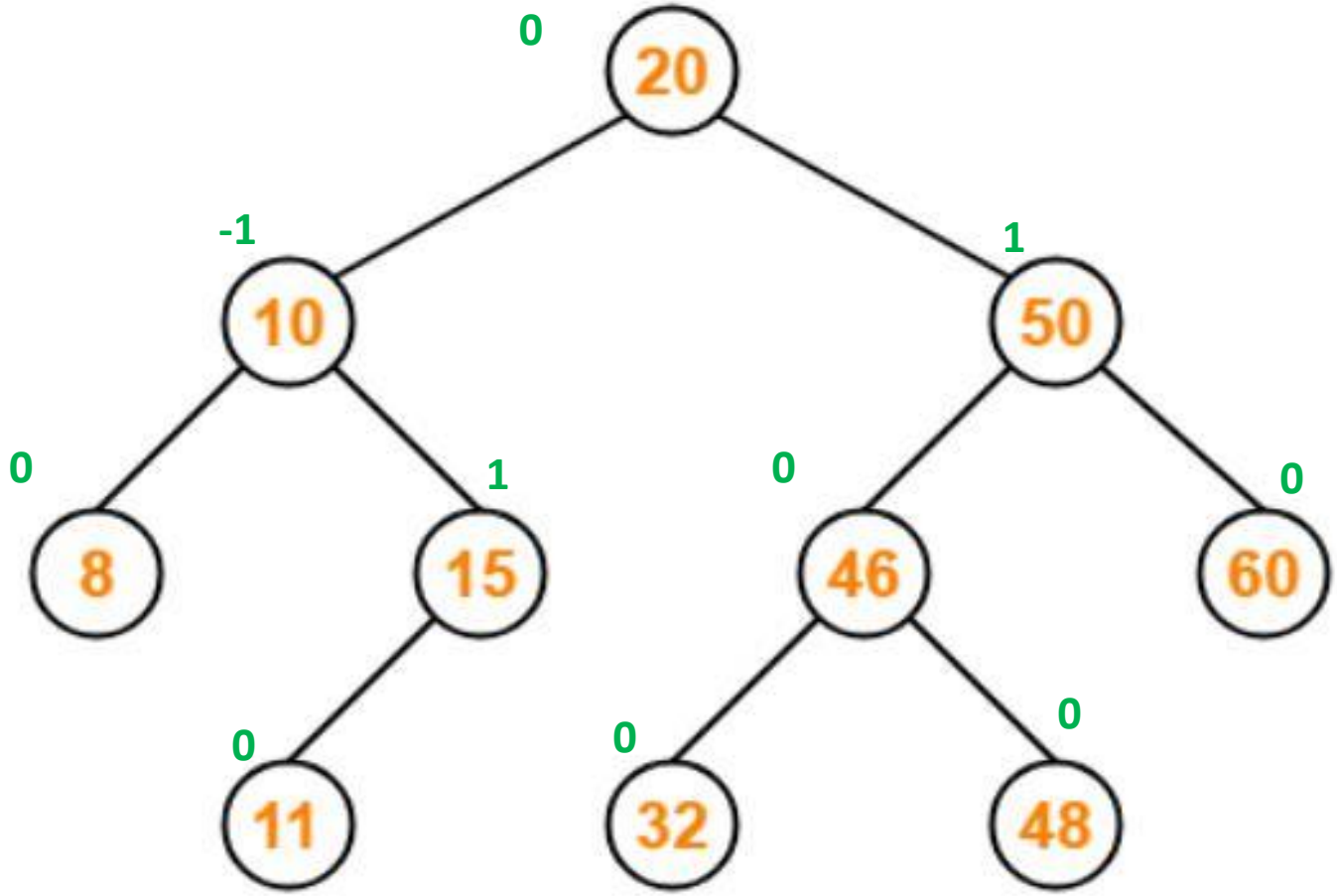
Step-09: Insert 11



Tree is Balanced

Step-10: Insert 48





Tree is Balanced

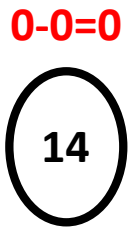
This is the final balanced AVL tree after inserting all the given elements.

Question 3: Construct a **AVL Tree** by inserting the following sequence of numbers…

14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

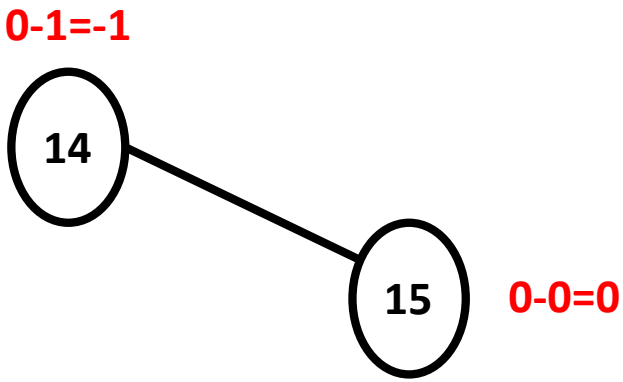
Solution:

Step-1: Insert 14



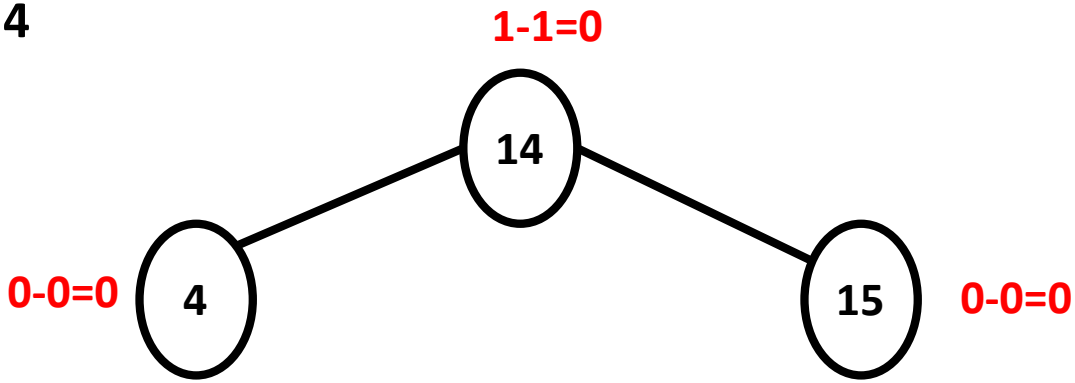
Tree is balanced

Step-2: **Insert 15**



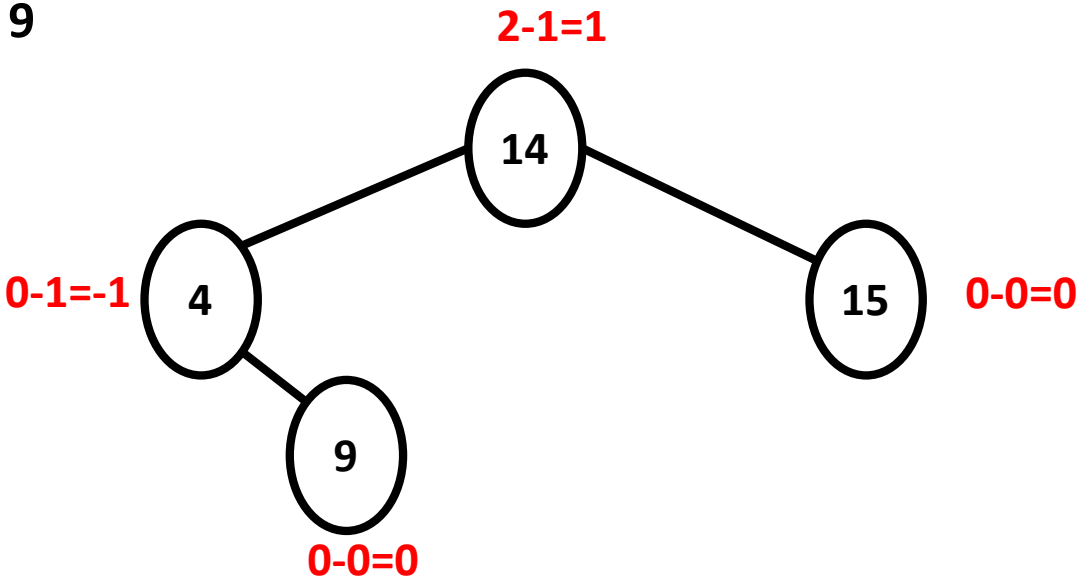
Tree is balanced

Step-3: **Insert 4**



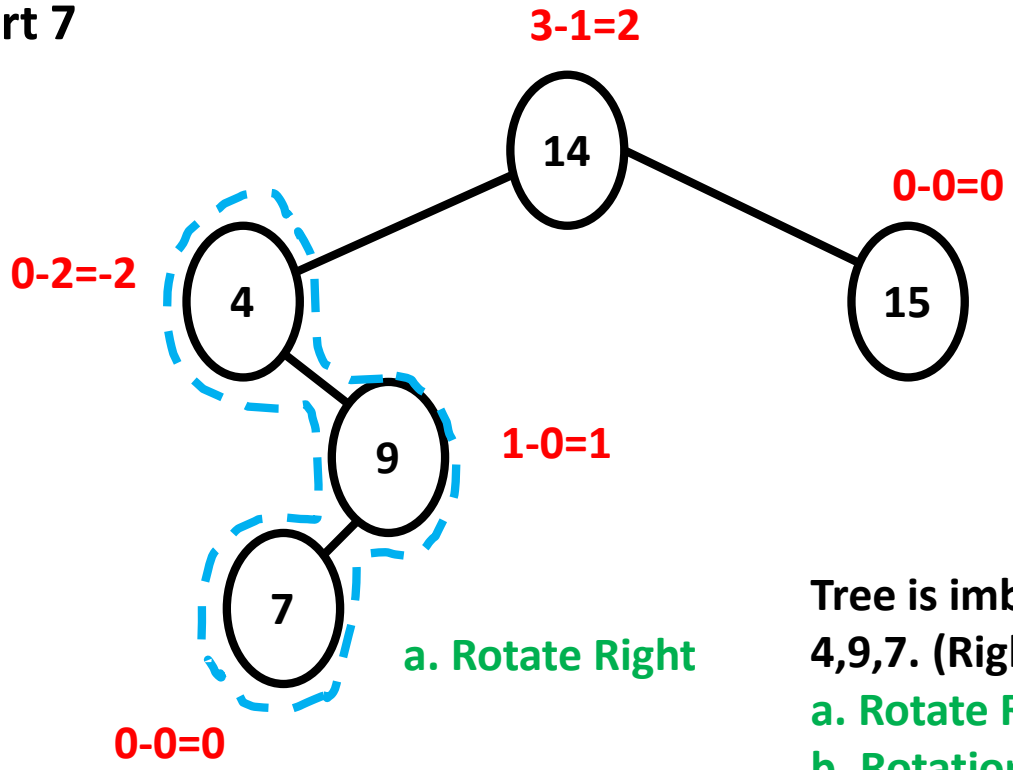
Tree is balanced

Step-4: Insert 9



Tree is balanced

Step-5: Insert 7

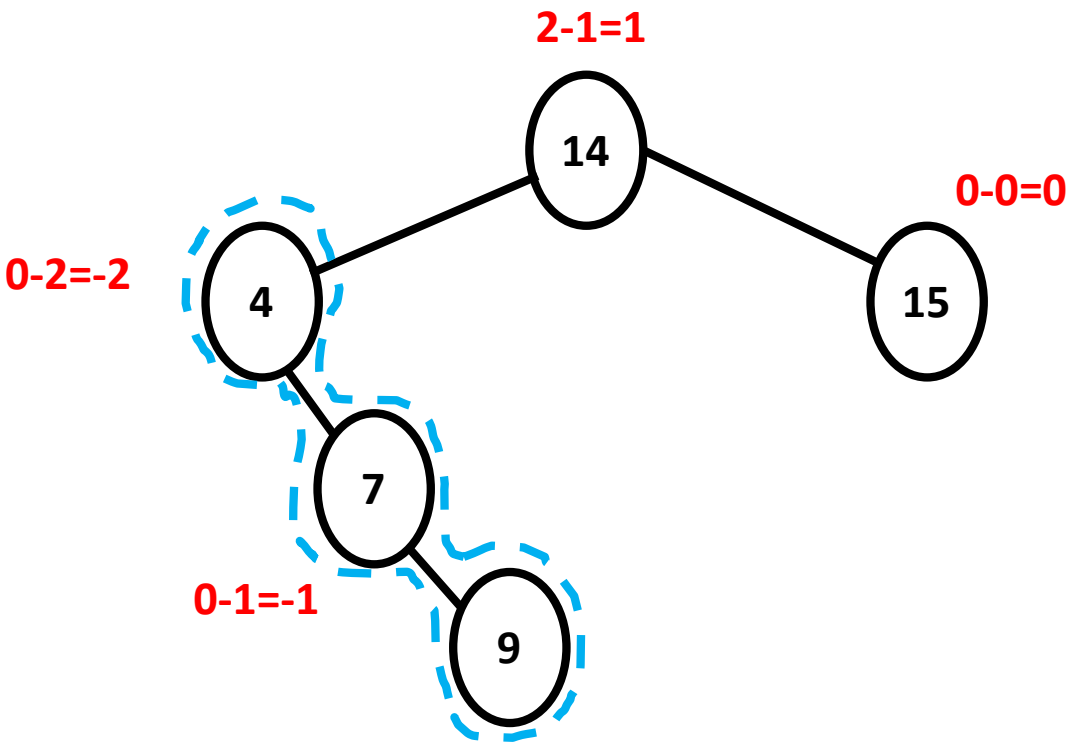


Tree is imbalance i.e in node 4,9,7. (Right of left)

a. Rotate Right

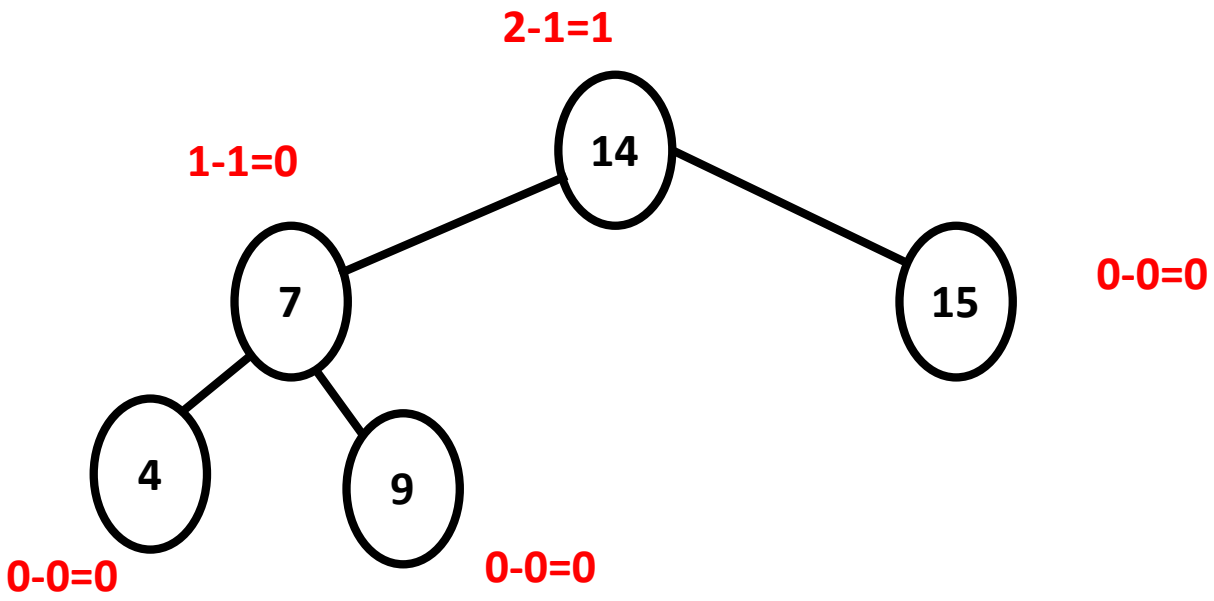
b. Rotation left

Tree is Imbalanced



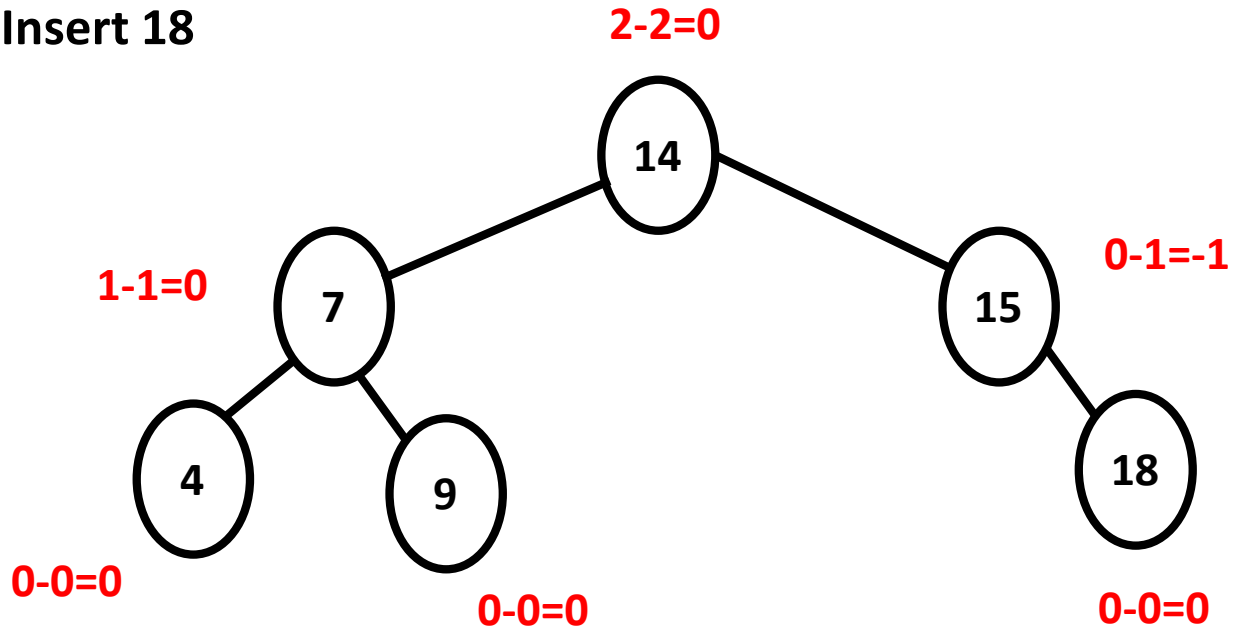
b. Rotation left

Tree is Imbalanced



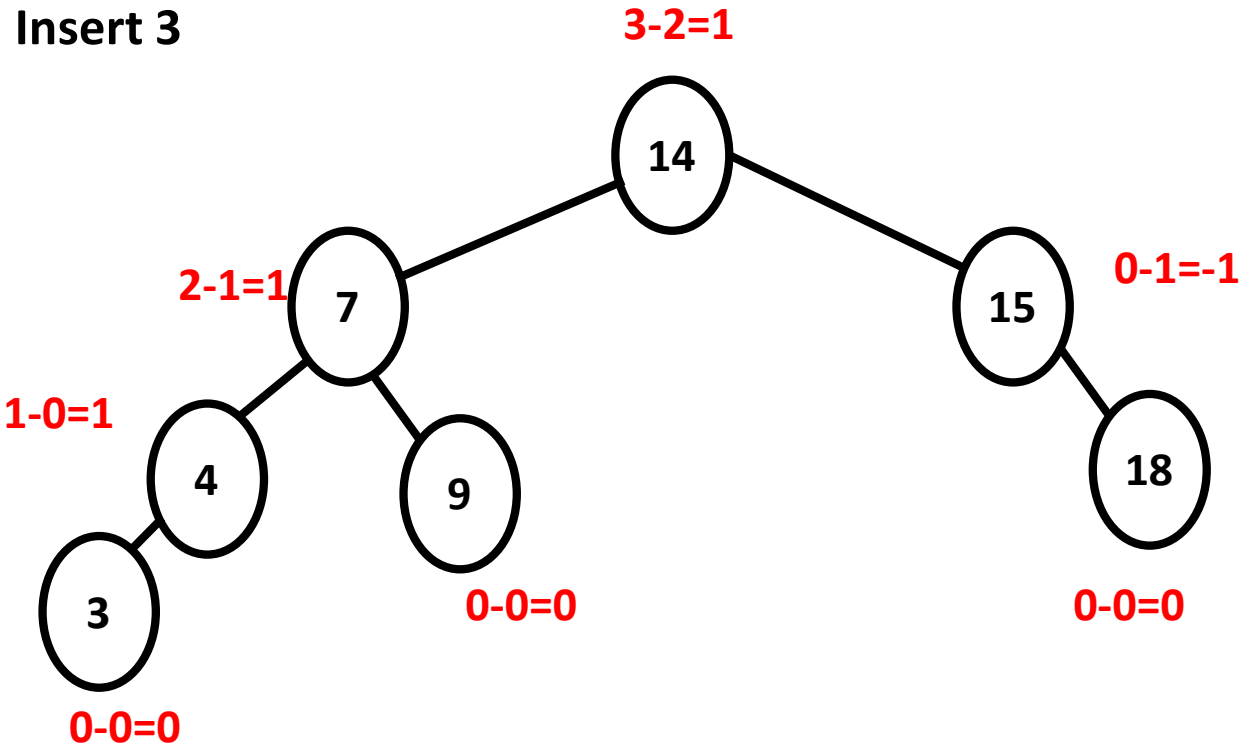
Tree is balanced

Step-6: Insert 18



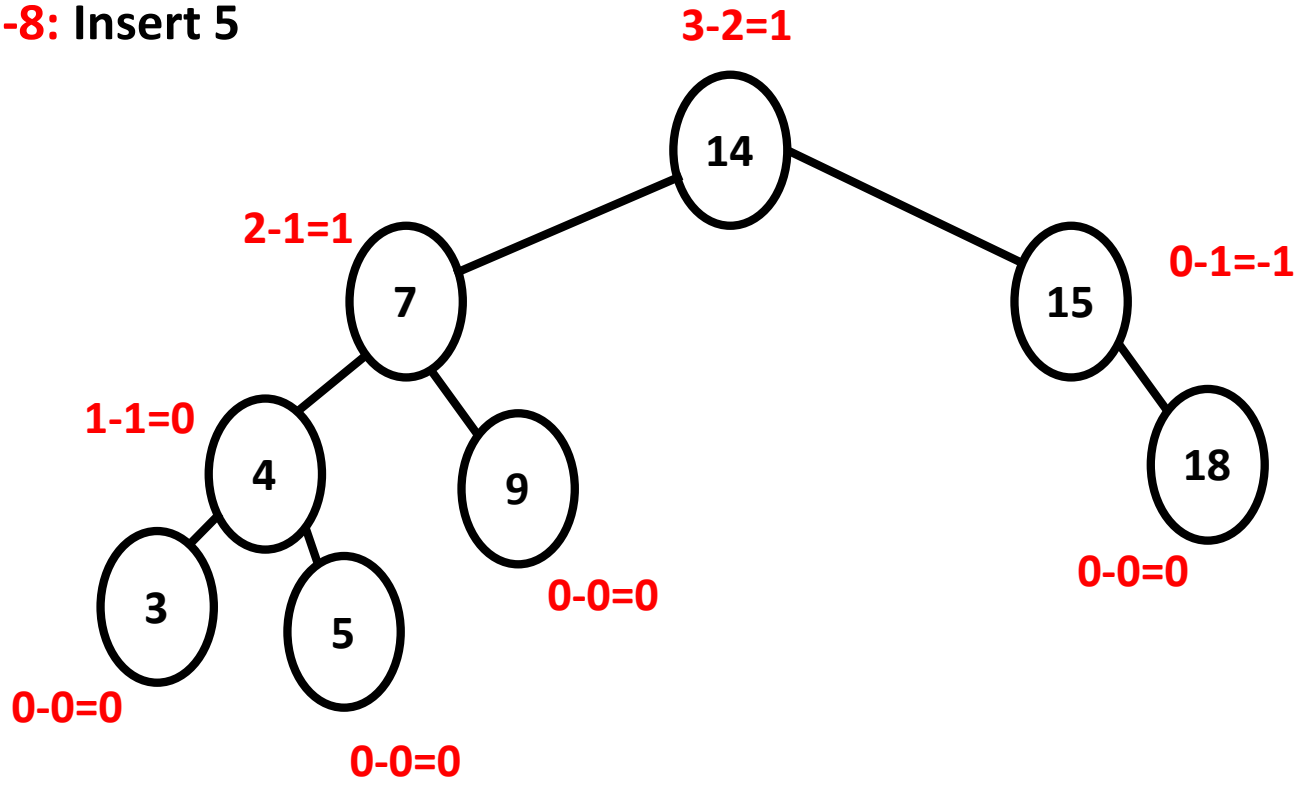
Tree is balanced

Step-7: Insert 3



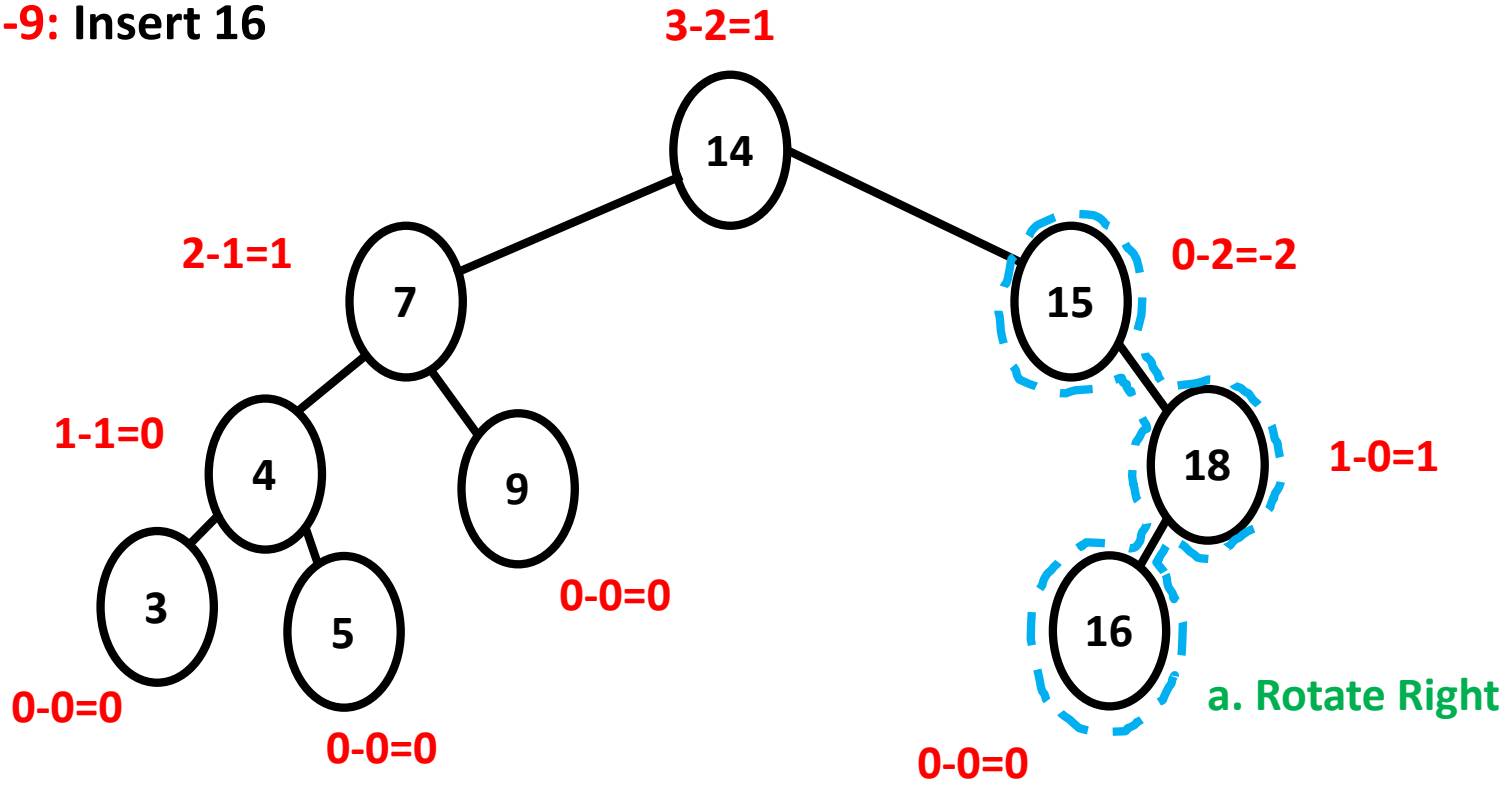
Tree is balanced

Step-8: Insert 5



Tree is balanced

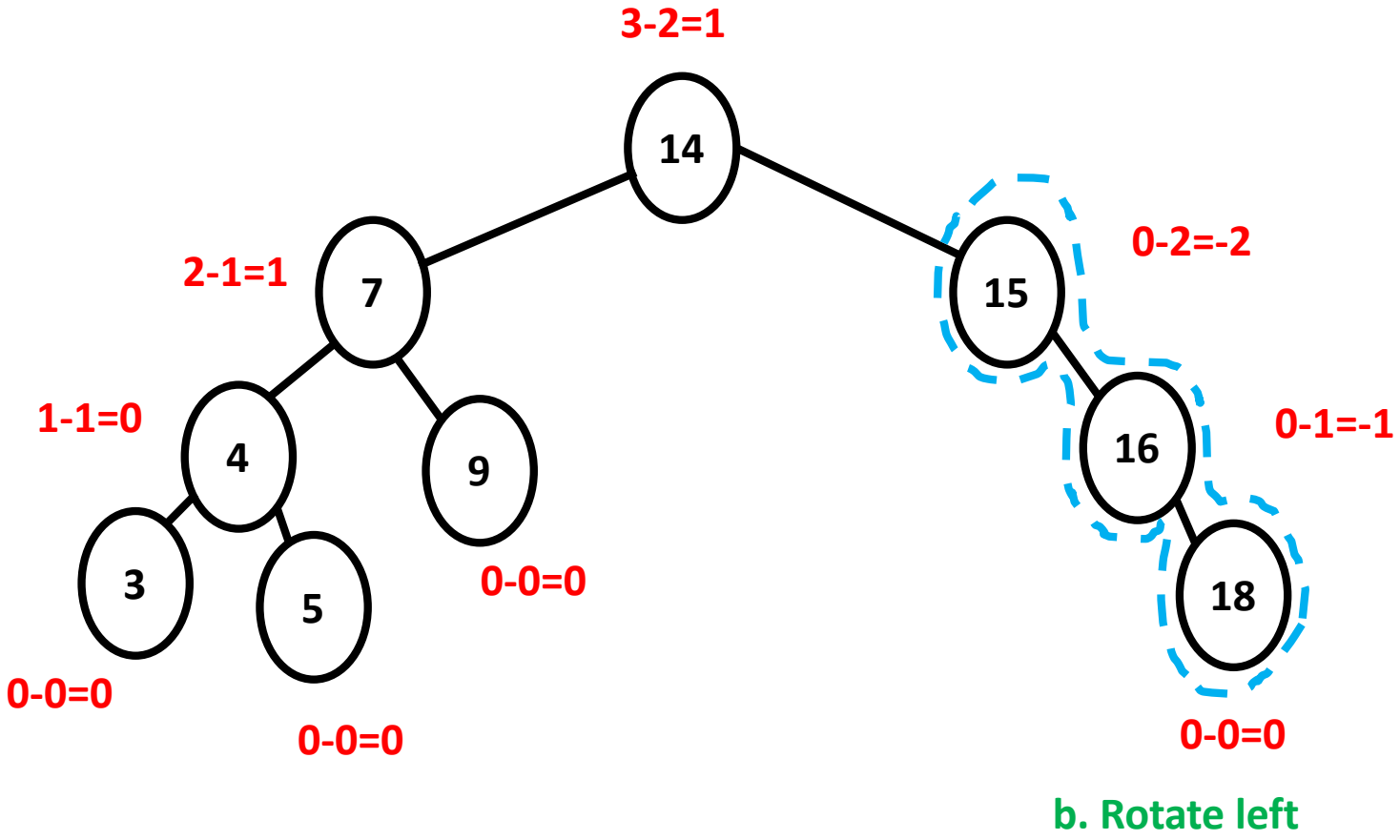
Step-9: Insert 16



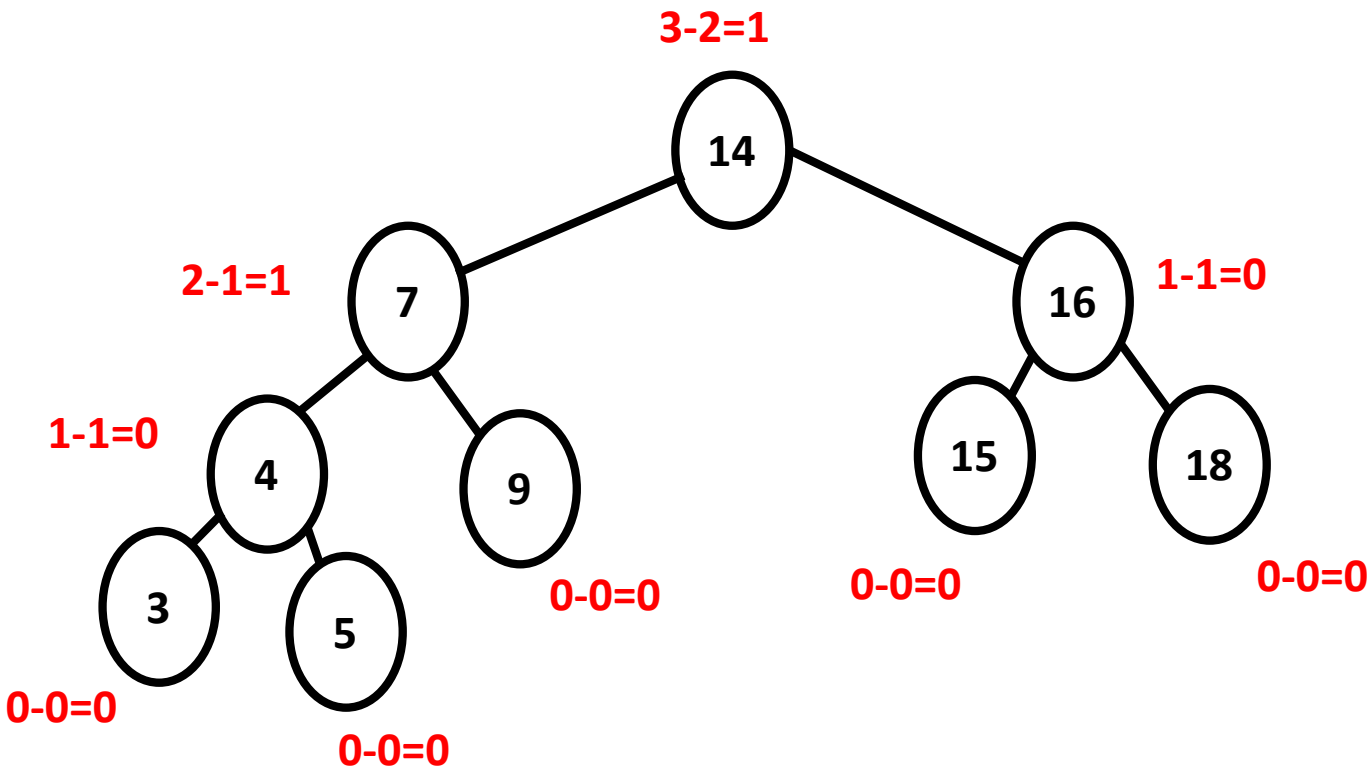
Tree is Imbalanced

Tree is imbalance i.e in node 15,18,16. (Right of Left)

a. Rotate Right
b. Rotate Left

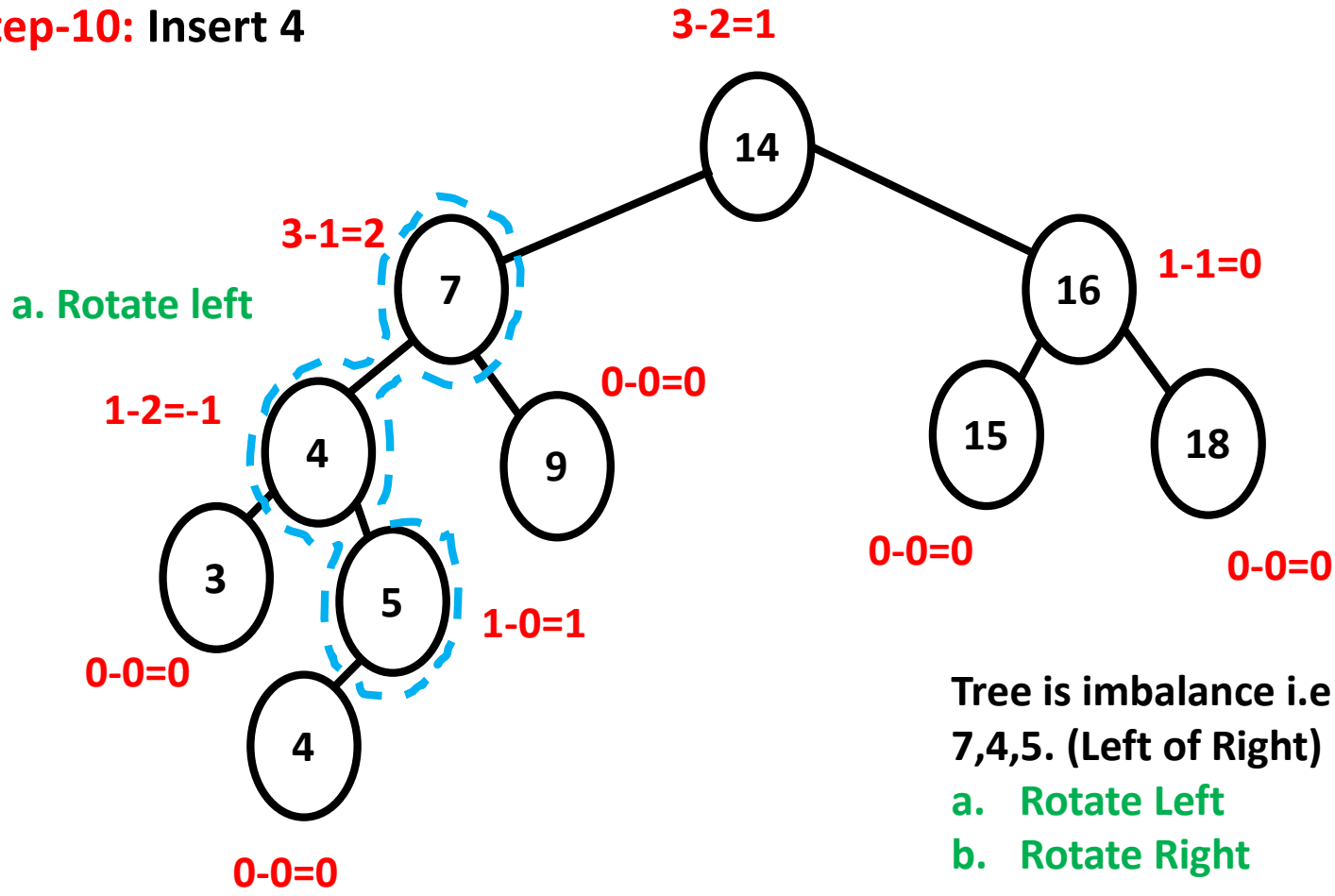


Tree is Imbalanced



Tree is balanced

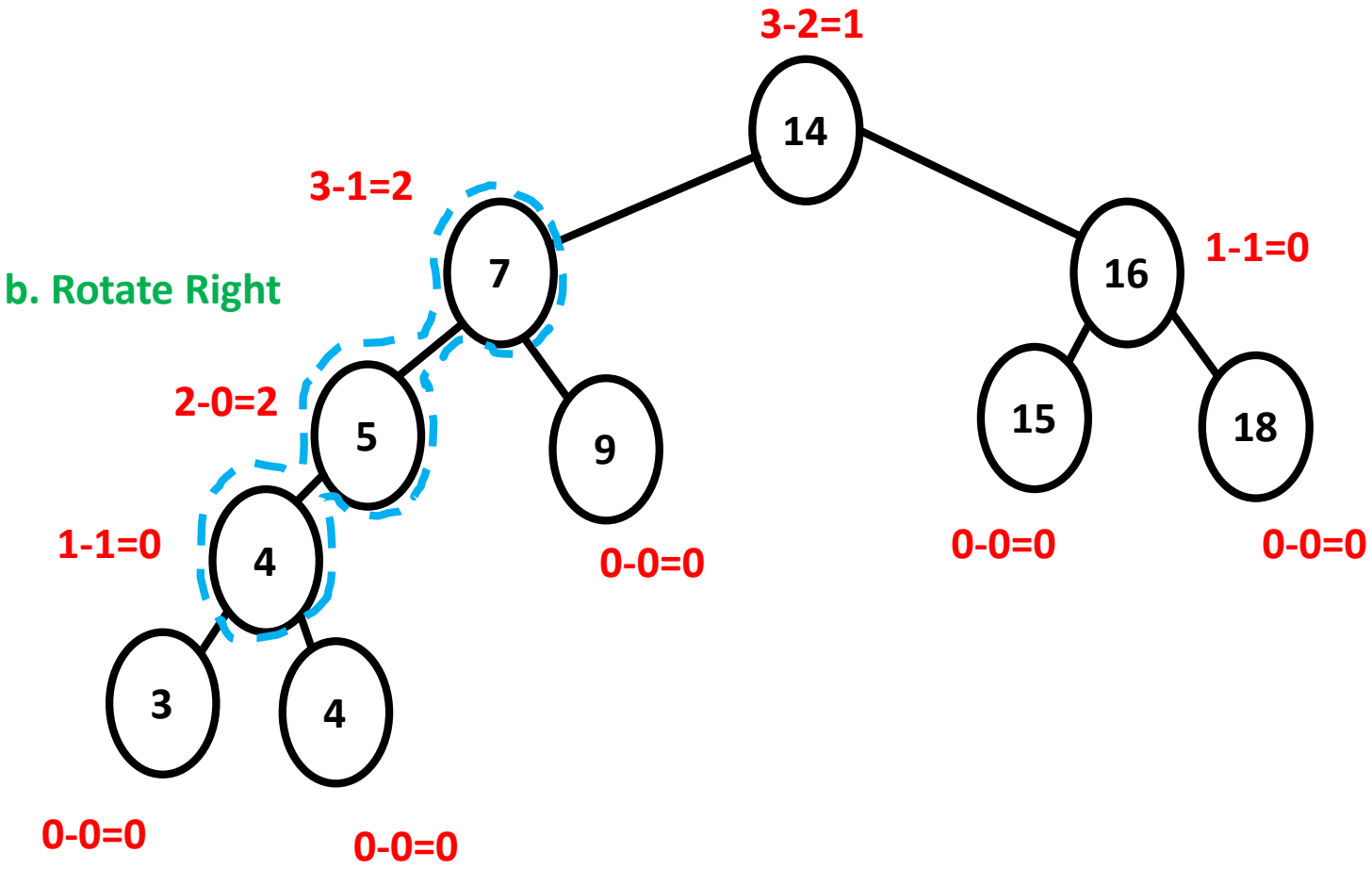
Step-10: Insert 4



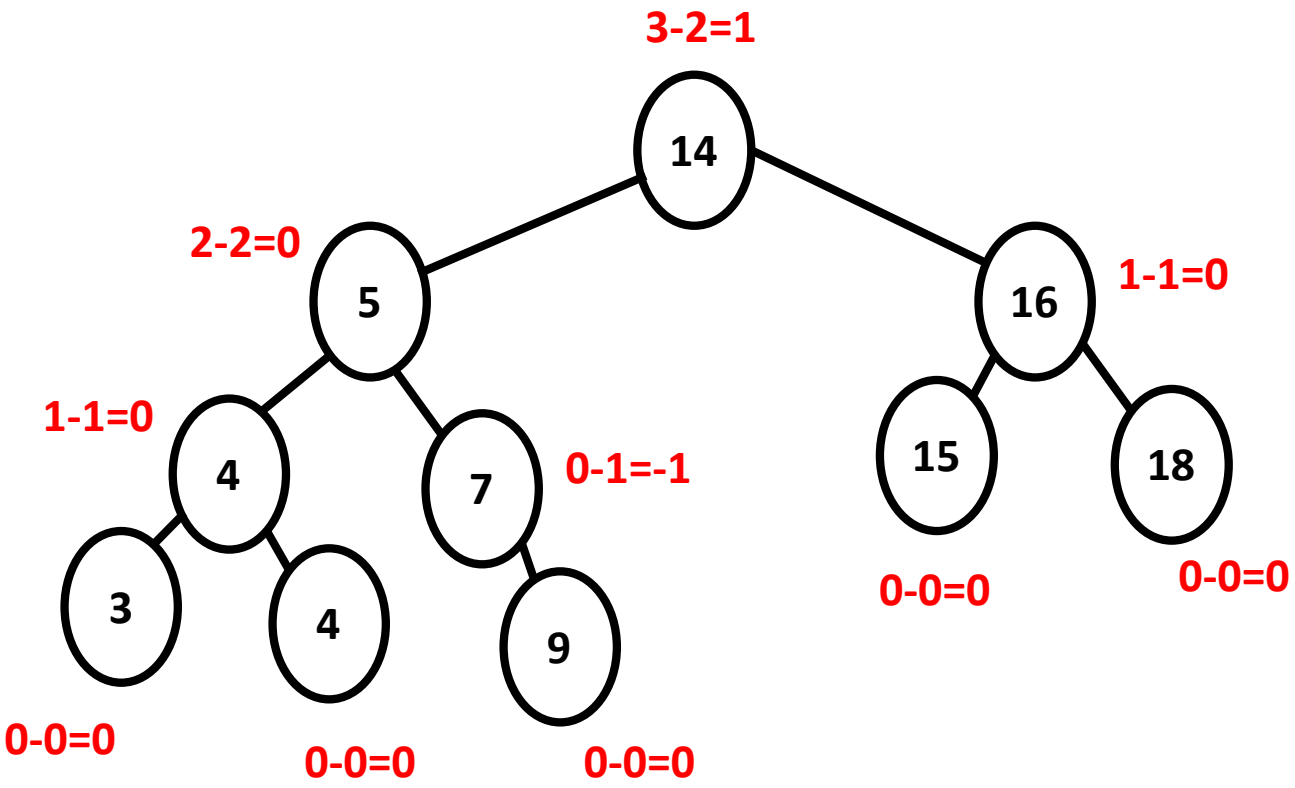
Tree is imbalance i.e in node 7,4,5. (Left of Right)

a. Rotate Left
b. Rotate Right

Tree is Imbalanced

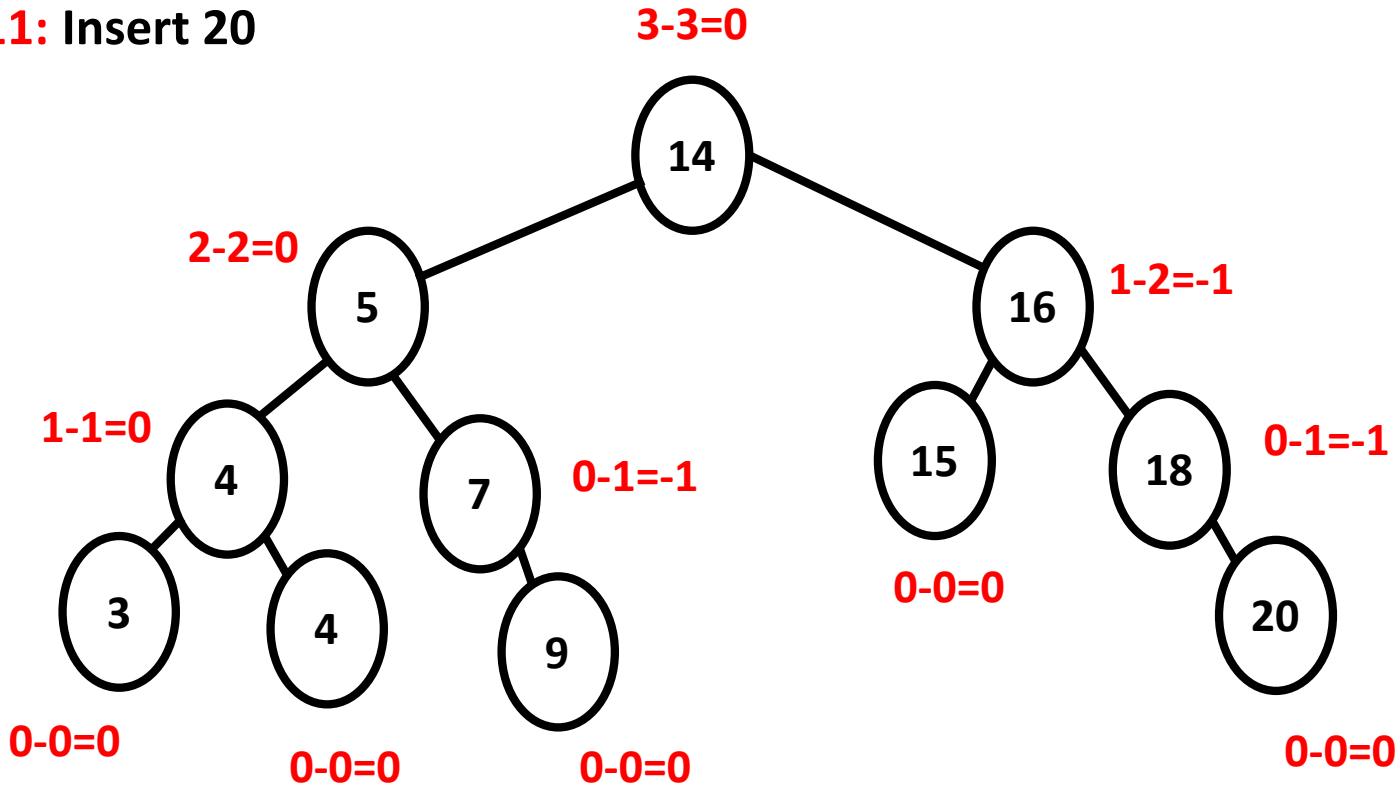


Tree is Imbalanced



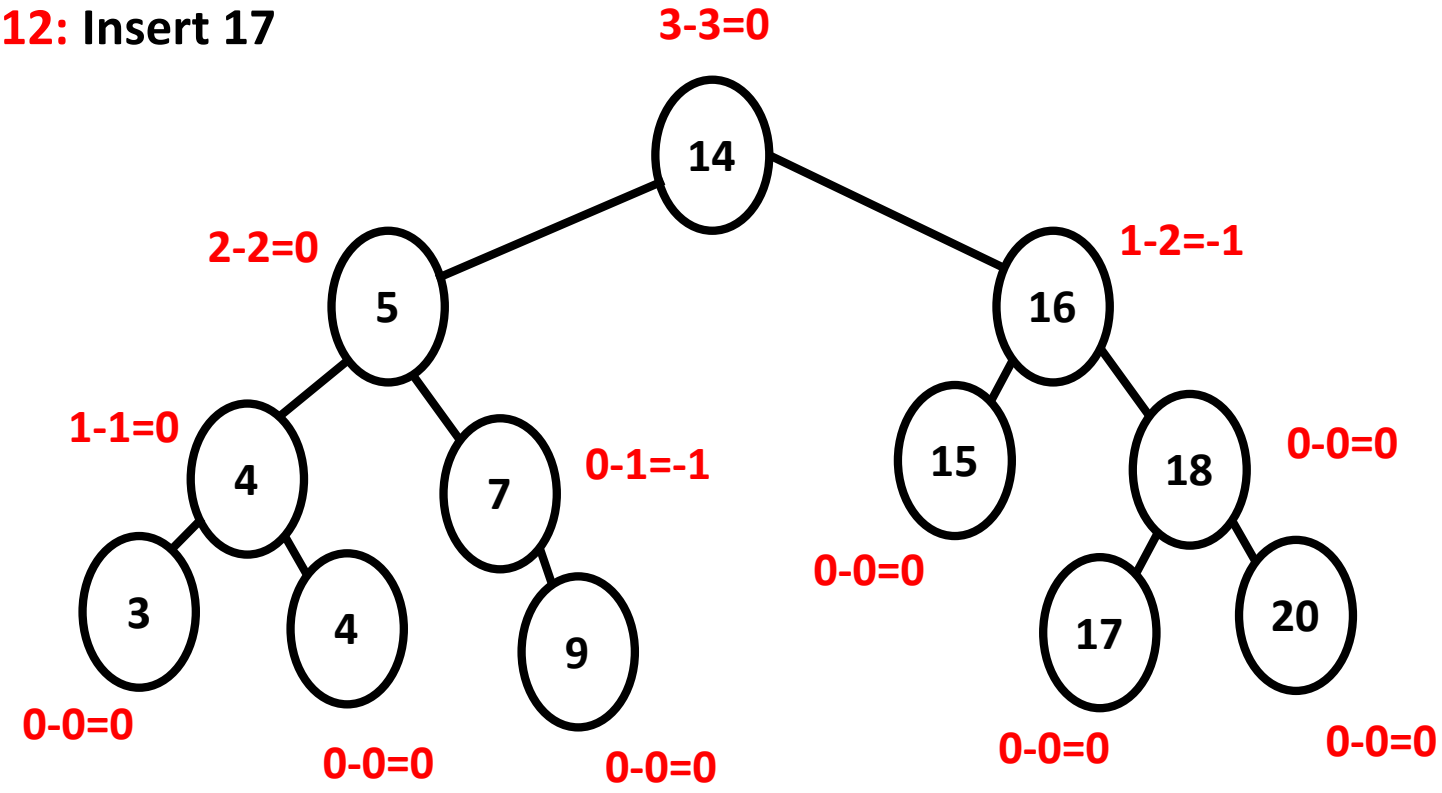
Tree is balanced

Step-11: Insert 20



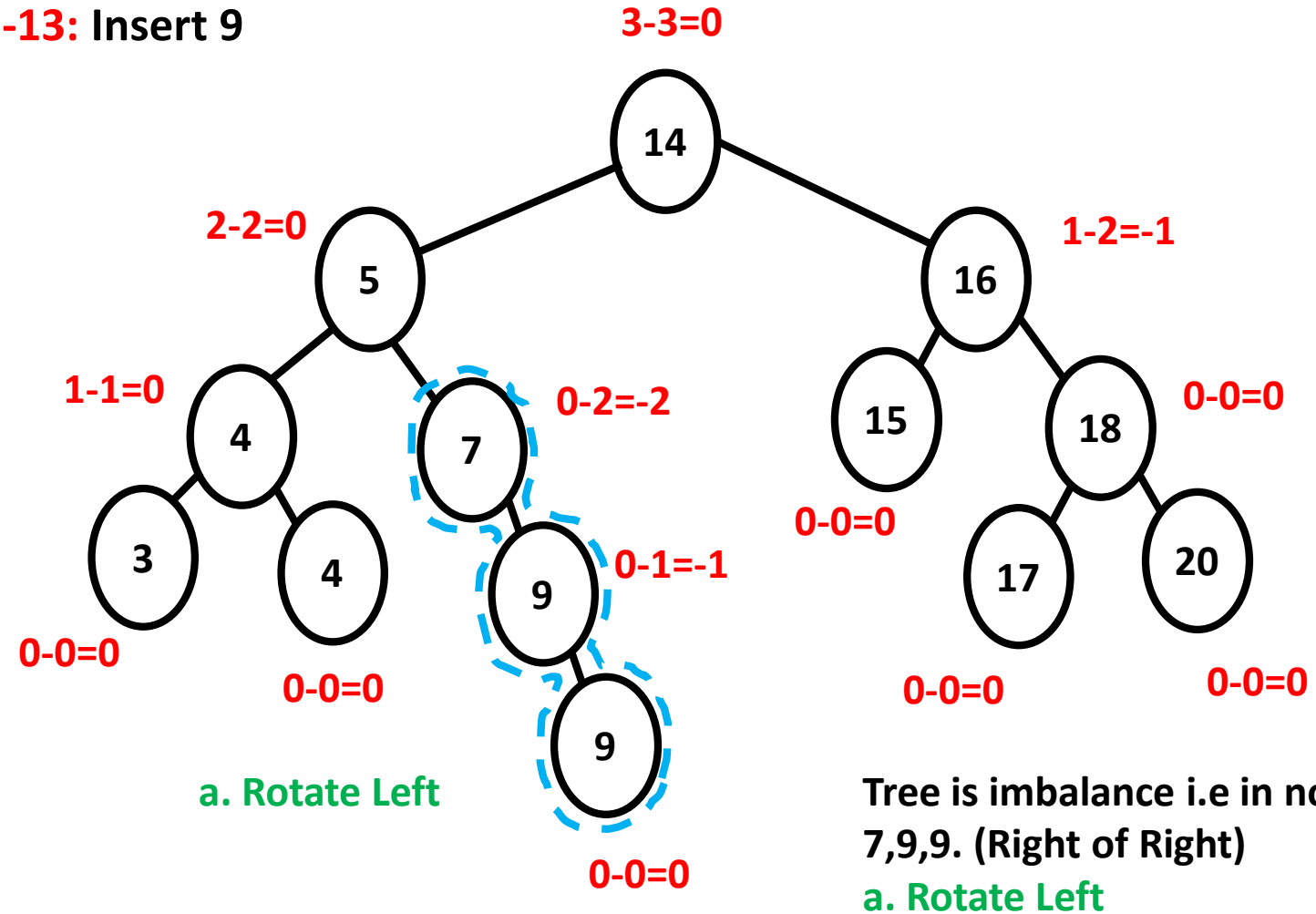
Tree is balanced

Step-12: Insert 17



Tree is balanced

Step-13: Insert 9

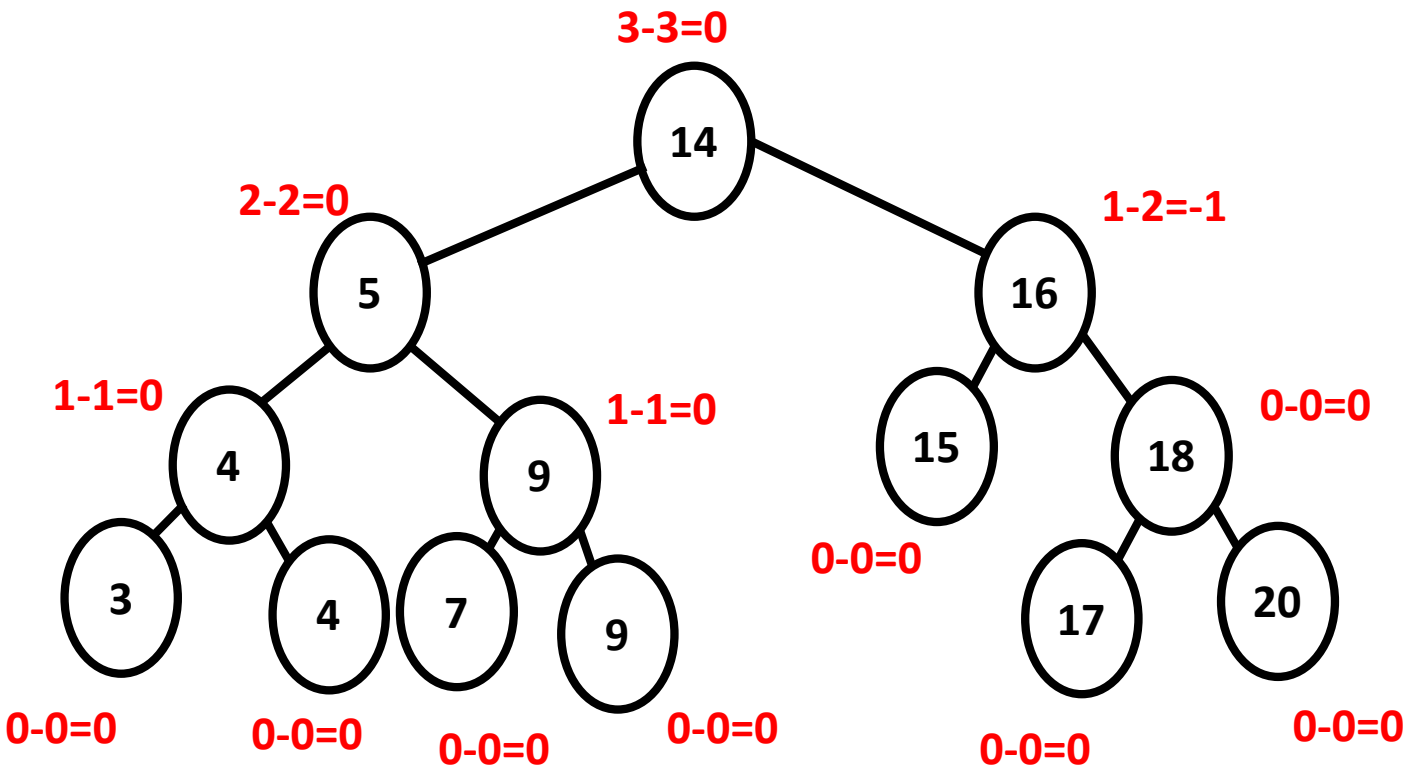


a. Rotate Left

Tree is Imbalanced

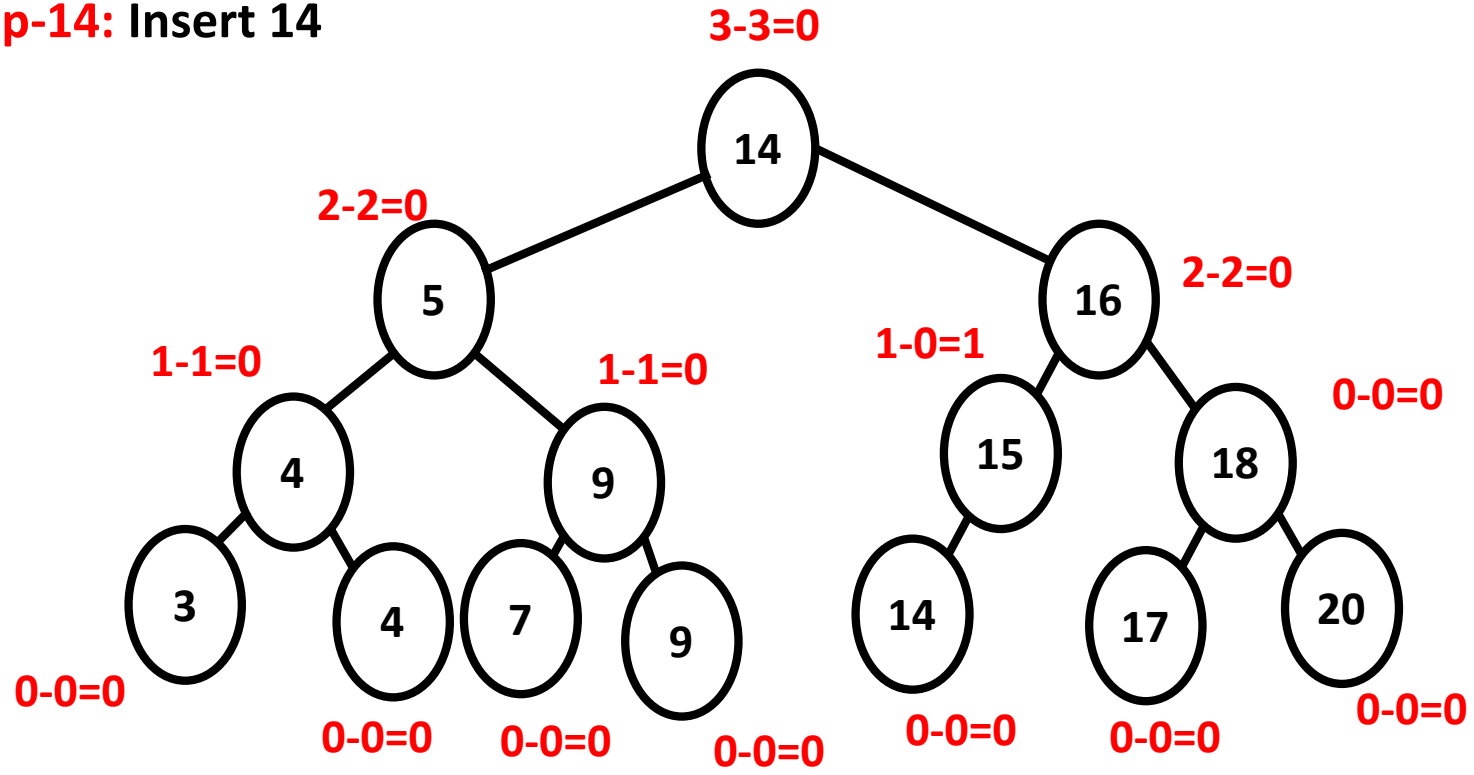
Tree is imbalance i.e in node 7,9,9. (Right of Right)

a. Rotate Left



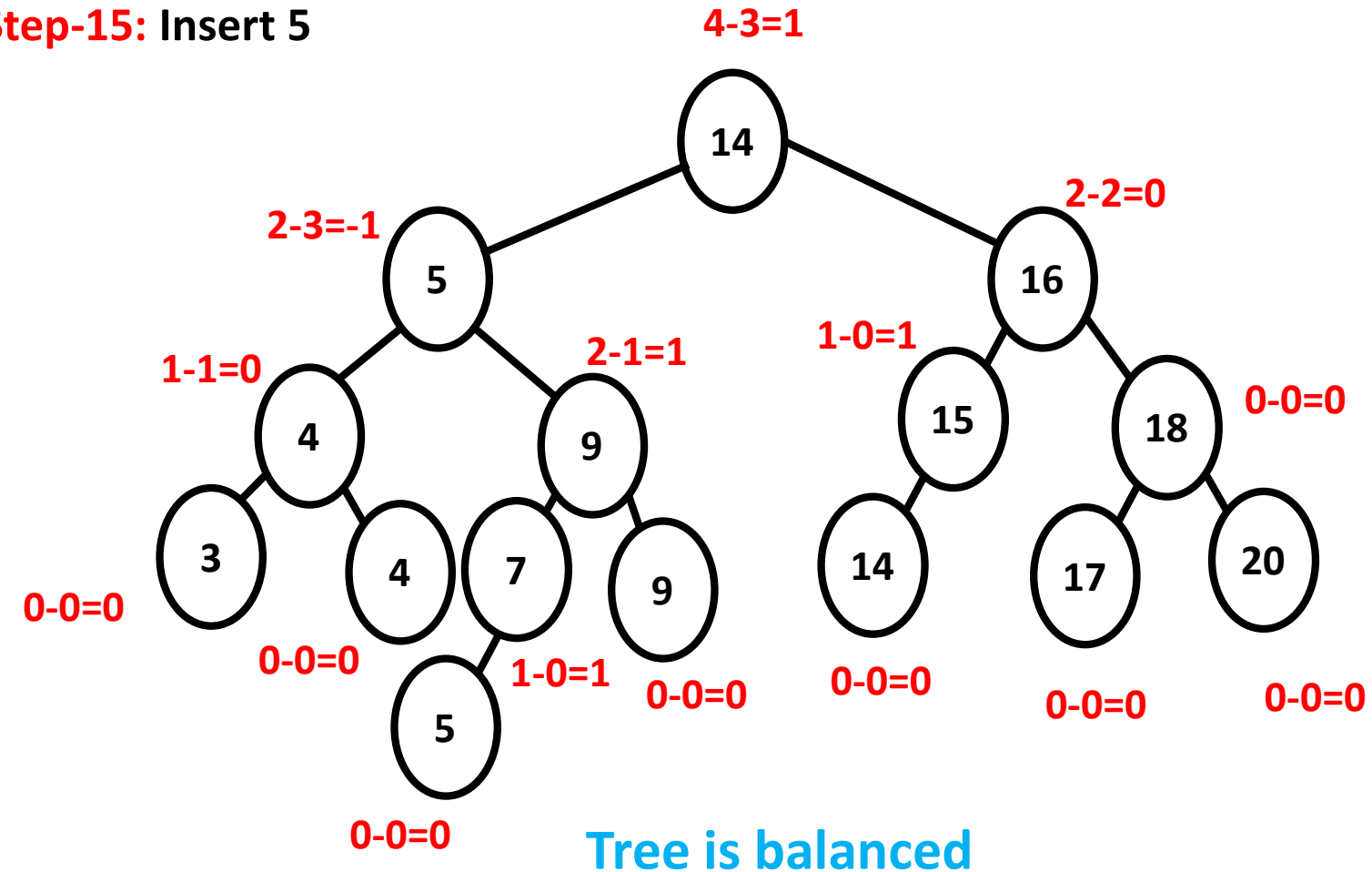
Tree is balanced

Step-14: Insert 14



Tree is balanced

Step-15: Insert 5



Tree is balanced

Final AVL Tree

Question 4: Draw the **AVL Tree** for the following sequence of data:

10, 30, 50, 40, 60, 100, 5, 2

Question 5: Construct a **AVL Tree** by inserting the following sequence of numbers…

25, 20, 36, 10, 22, 30, 40

Deletion Operation in AVL Tree

In an AVL Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Balance Factor condition. If the tree is balanced after deletion then go for next operation otherwise perform the suitable rotation to make the tree Balanced.

Application of Tree

- Searching
- Sorting
- Compiler (to validate the syntax, it uses syntax tree)
- Routers (store routing information)
- Database design
- File system
- Shortest path among the networks (minimum spanning tree)
- Organization chart
- Machine learning
- Encryption
- Block chain
- Gaming
- Coding