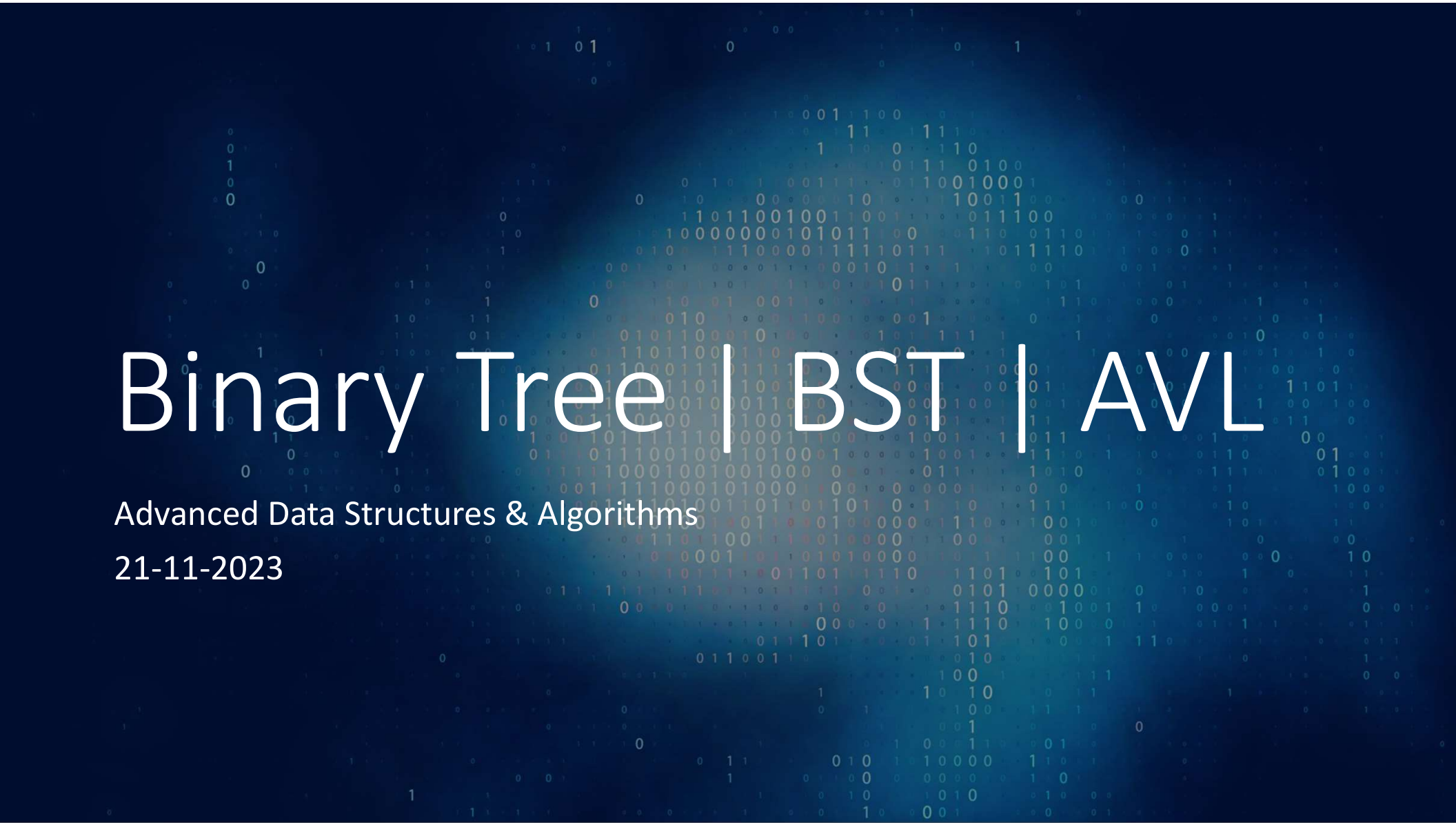


Binary Tree | BST | AVL

Advanced Data Structures & Algorithms

21-11-2023

Important Properties of Binary Tree

Property 1: Maximum number of nodes in a Binary Tree of height H is

$$2^{H+1} - 1$$

Property 2: Minimum number of nodes in a Binary Tree of height H is

$$H + 1$$

Property 3: Maximum number of nodes at any nodes at any level L in a binary tree

$$2^{L}$$

Important Properties of Binary Tree

Property 4: Total number of leaf nodes N_{leaf} in a binary tree is equal to the total number of nodes with 2 children or degree 2 N_{degree2} + 1.

$$N_{leaf} = N_{degree2} + 1$$

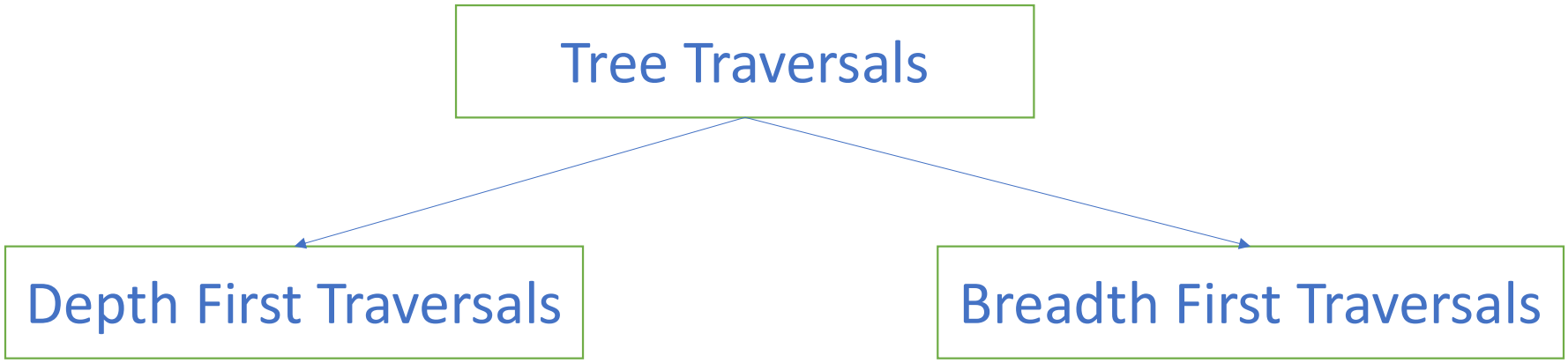
Problems

1. A binary tree T has n leaf nodes. The number nodes of degree 2 in T is _____?
2. In a binary tree, the difference between the number of nodes in the left and right subtrees is at most 2. If the height of the tree is h > 0, then the minimum number of nodes in the tree is _____?
   1. 2^{h-1}
   2. $2^{h-1}+1$
   3. $2^{h}-1$
   4. 2^{h}

Problems

1. The height of binary tree is the maximum number of edges in any root to leaf path. The maximum number of nodes in a binary tree of height h is _____?
   1. 2^h
   2. $2^{h-1} - 1$
   3. $2^{h+1} - 1$
   4. 2^{h+1}
2. A binary tree T has 20 leaves. The number of nodes in T having 2 children is _____?

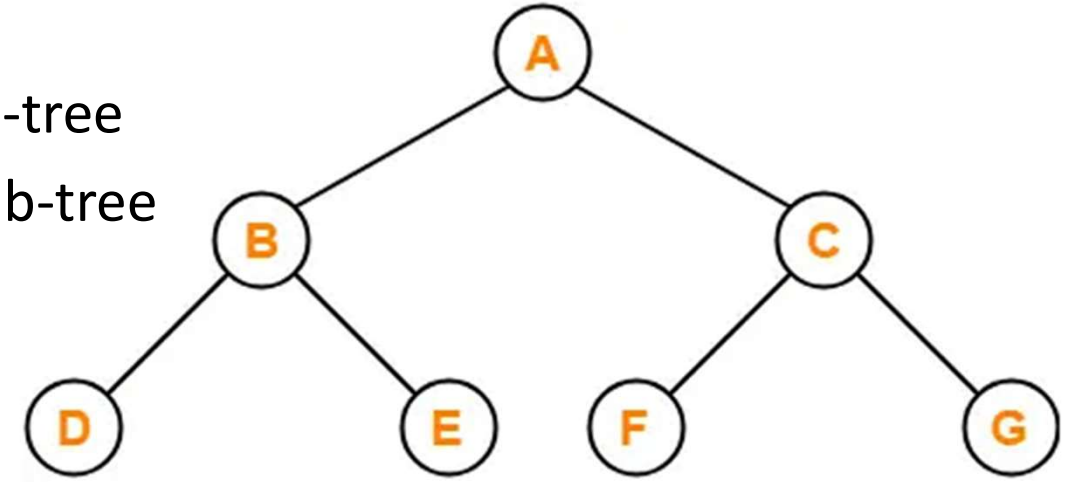
Tree Traversals



1. Preorder (root -> left -> right)
2. Inorder (left -> root -> right)
3. Postorder (left -> right -> root)

Preorder Traversal

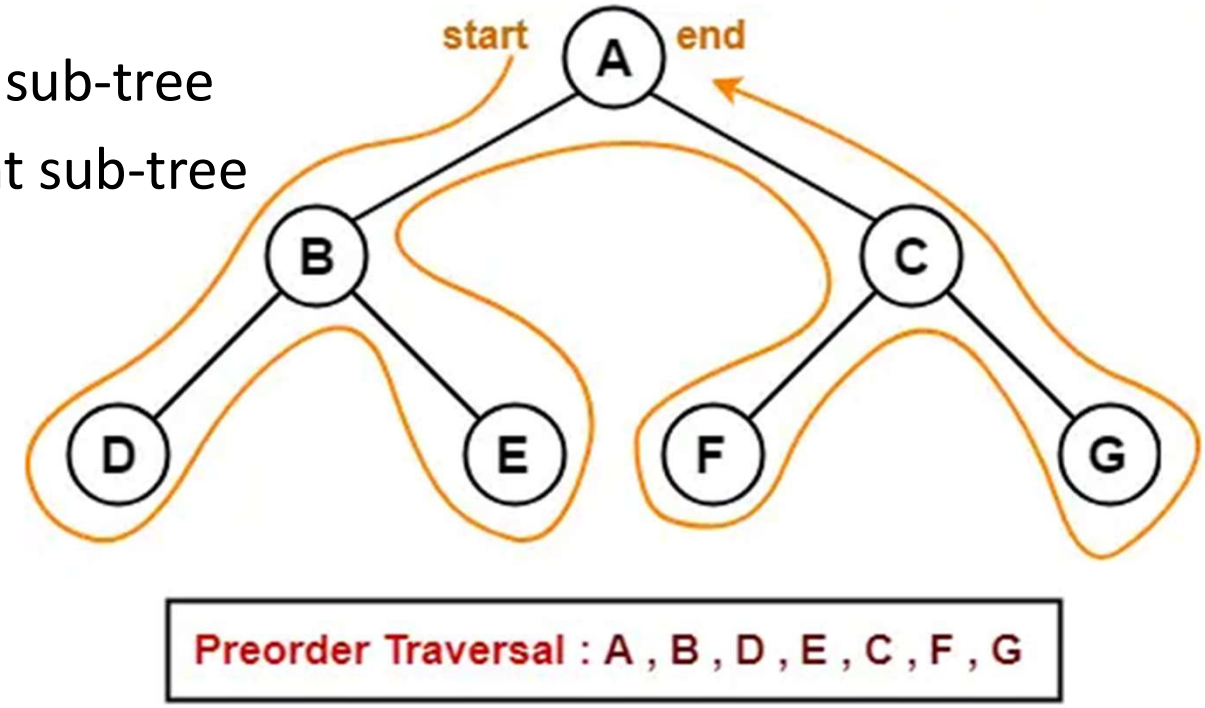
- Visit the root
- Traverse the left sub-tree
- Traverse the right sub-tree



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

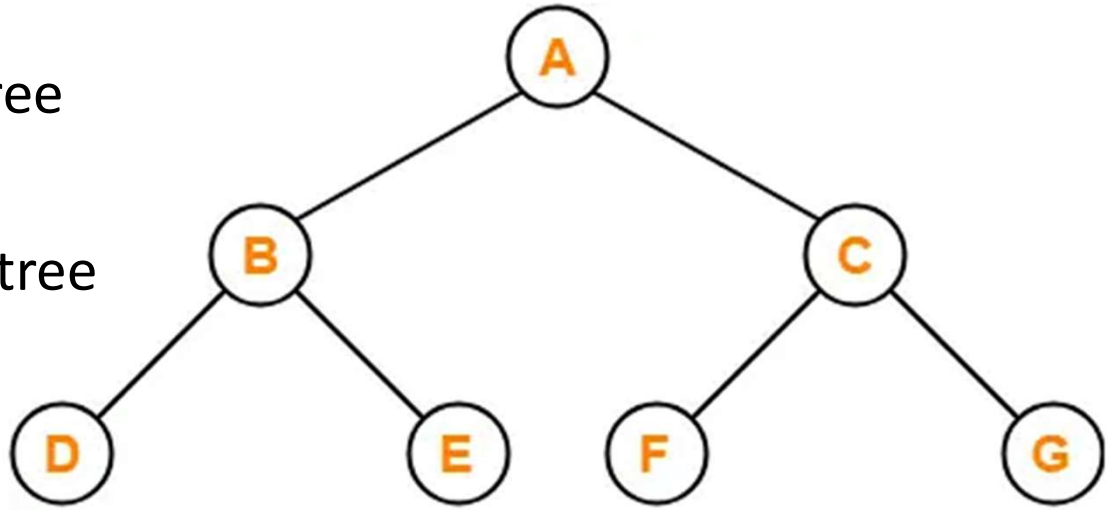
Preorder Traversal Shortcut

- Visit the root
- Traverse the left sub-tree
- Traverse the right sub-tree



Inorder Traversal

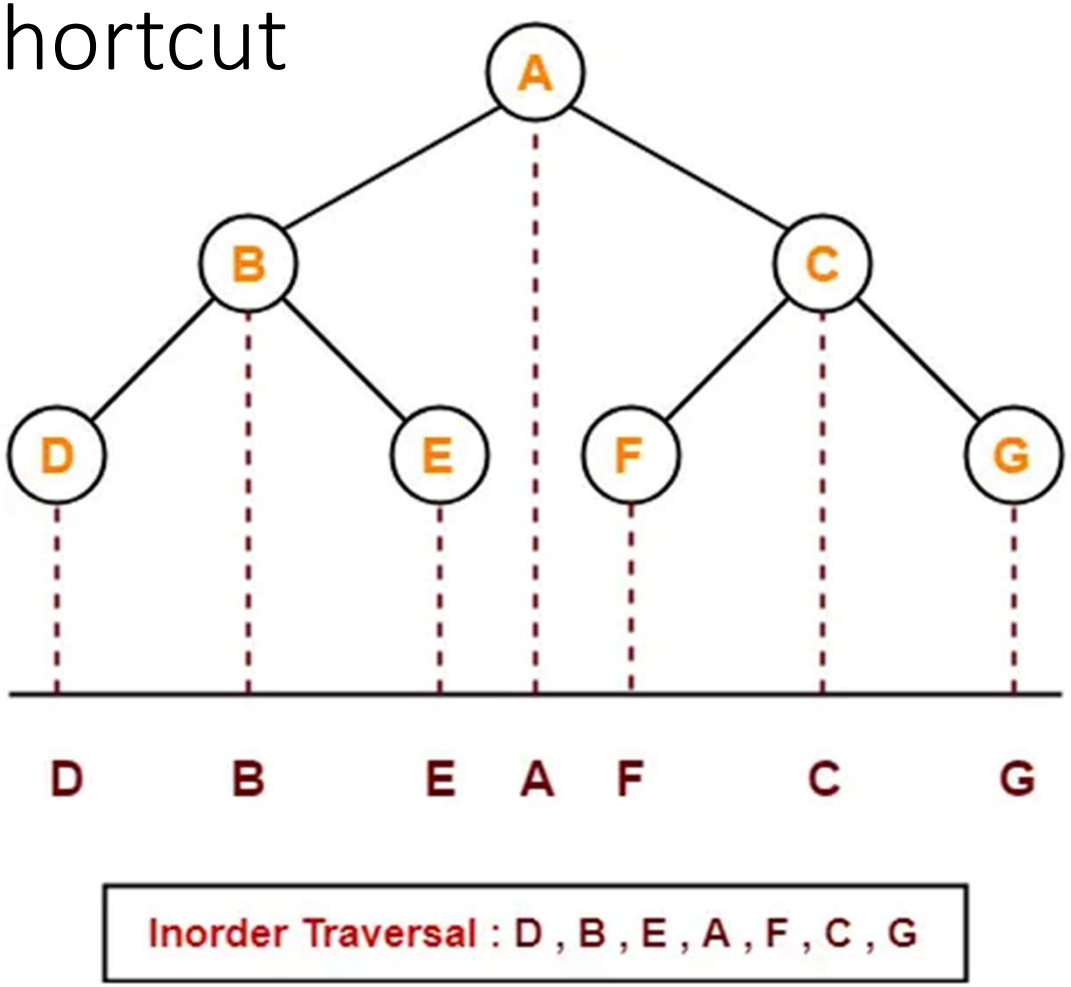
- Traverse the left sub-tree
- Visit the root
- Traverse the right sub-tree



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

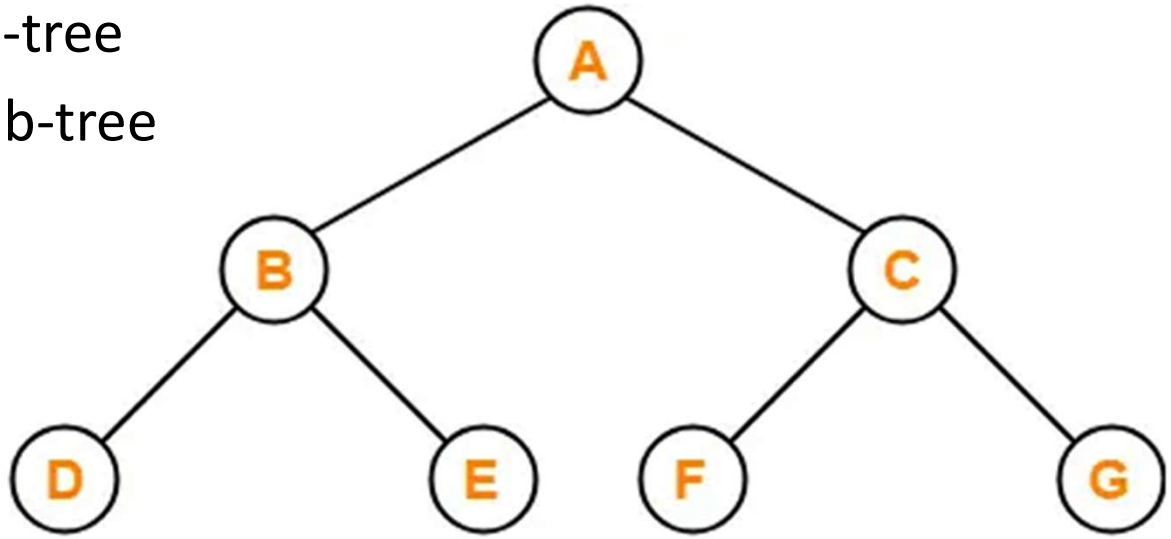
Inorder Traversal Shortcut

- Traverse the left sub-tree
- Visit the root
- Traverse the right sub-tree



Postorder Traversal Shortcut

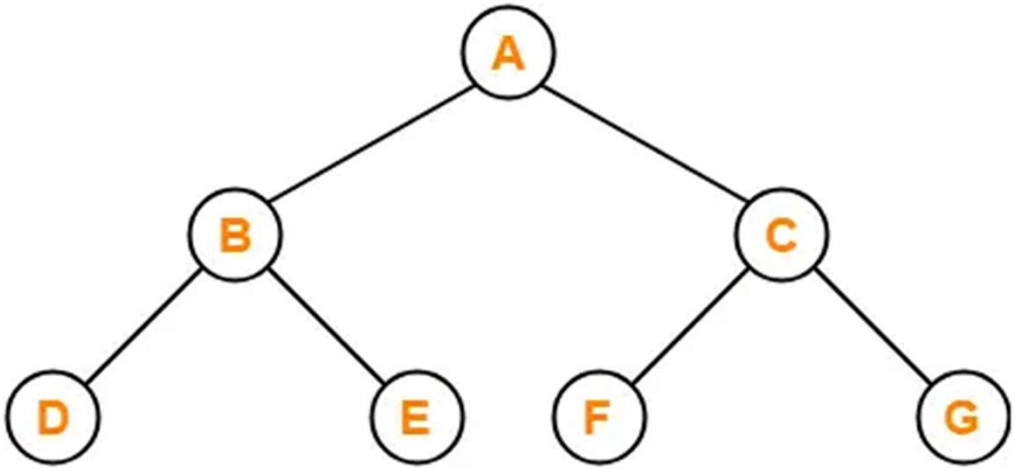
- Traverse the left sub-tree
- Traverse the right sub-tree
- Visit the root



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

Breadth First Traversal

- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as Level Order Traversal.

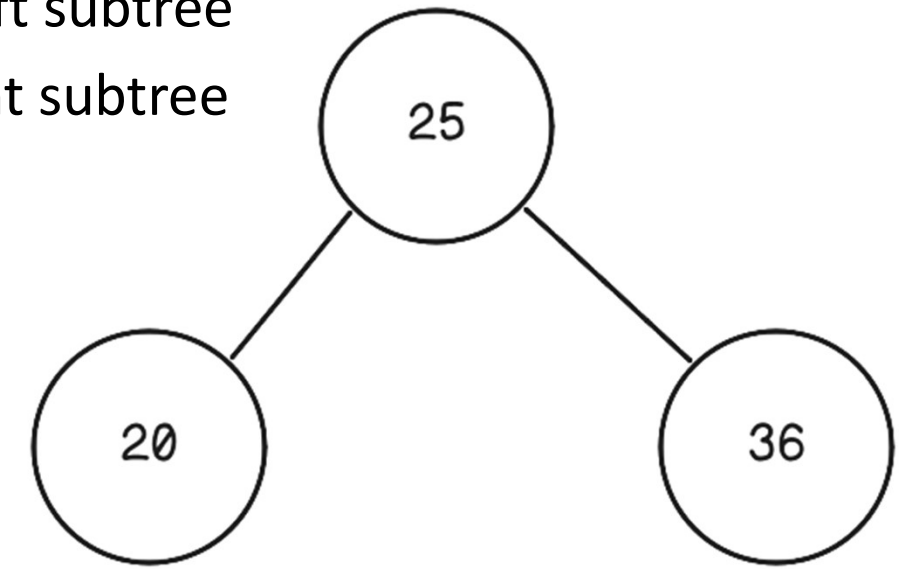


Level Order Traversal : A , B , C , D , E , F , G

Binary Search Tree (BST)

In BST, each node contains -

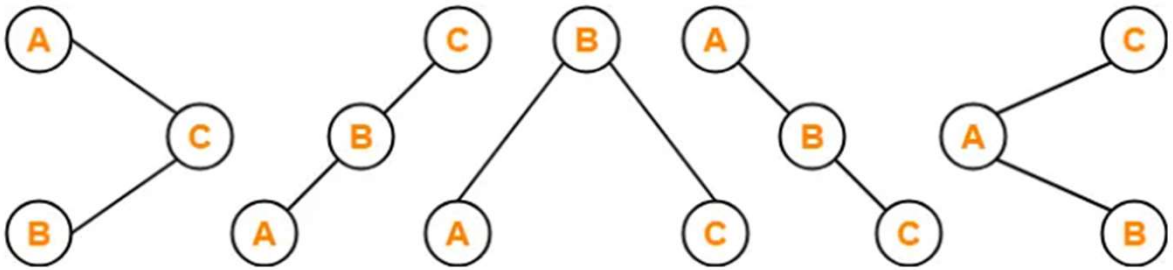
- Only smaller values in its left subtree
- Only larger values in its right subtree



Number of Binary Search Trees

Number of distinct BST possible with n distinct keys is

$$\frac{{}^{2n}C_n}{n+1}$$



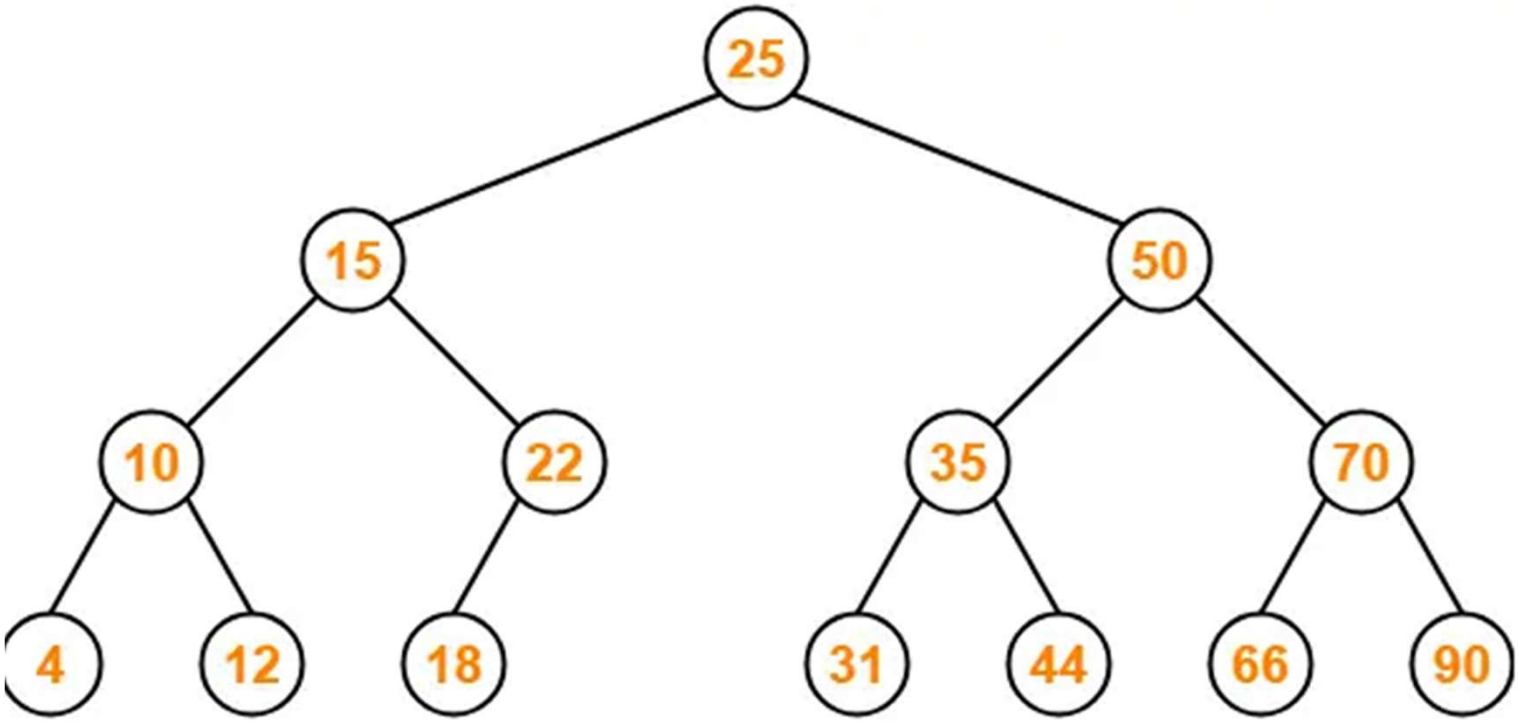
BST Operations

- Search
- Insertion
- Deletion

Search Operations

1. Compare the key with the value of root node.
2. If the key is present at the root node, then return the root node.
3. If the key is greater than the root node value, then recur for the root node's **right subtree**.
4. If the key is smaller than the root node value, then recur for the root node's **left subtree**.

Search Operations

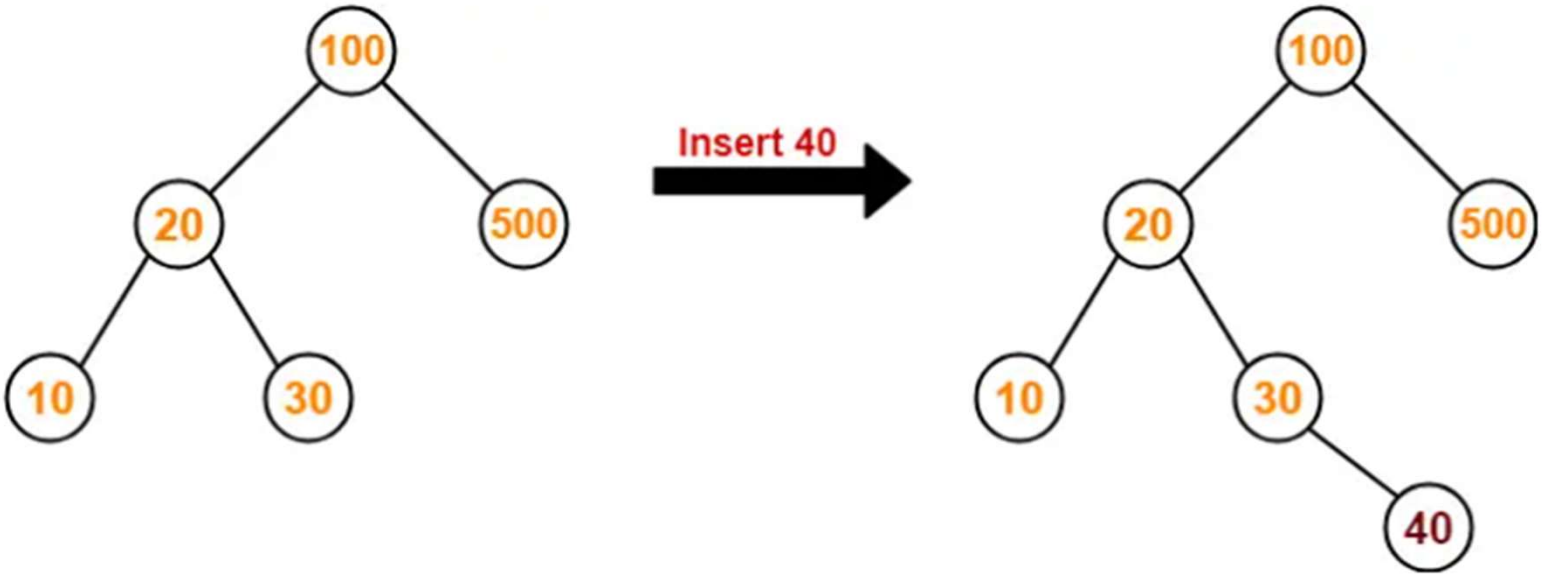


Search Operation

```
def _search(self, root, key):
    if root is None or root.key == key:
        return root
    if key < root.key:
        return self._search(root.left, key)
    return self._search(root.right, key)
```

Insertion Operation

- Search the key to be inserted from the root node till some leaf node is reached.
- Once a leaf node is reached, insert the key as child of that leaf node.



Insertion Operation

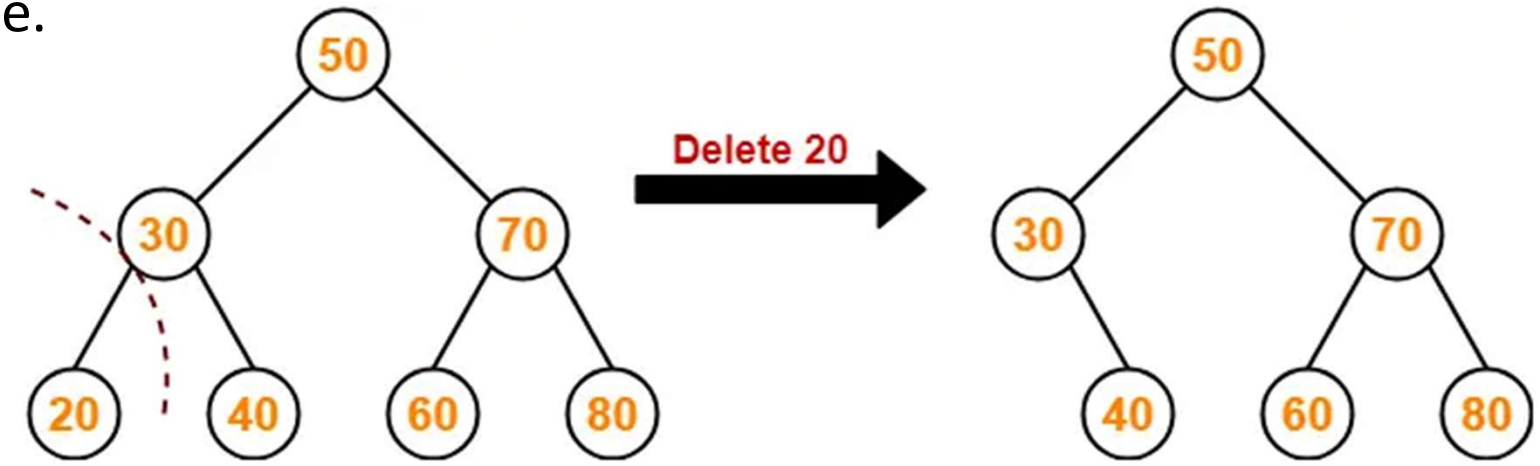
```
def _insert(self, root, key):
    if root is None:
        return TreeNode(key)
    if key < root.key:
        root.left = self._insert(root.left, key)
    else:
        root.right = self._insert(root.right, key)
    return root
```

Deletion Operation

There can be three cases when deleting a node in BST

Case-01: Deletion Of A Node Having No Child (Leaf Node)

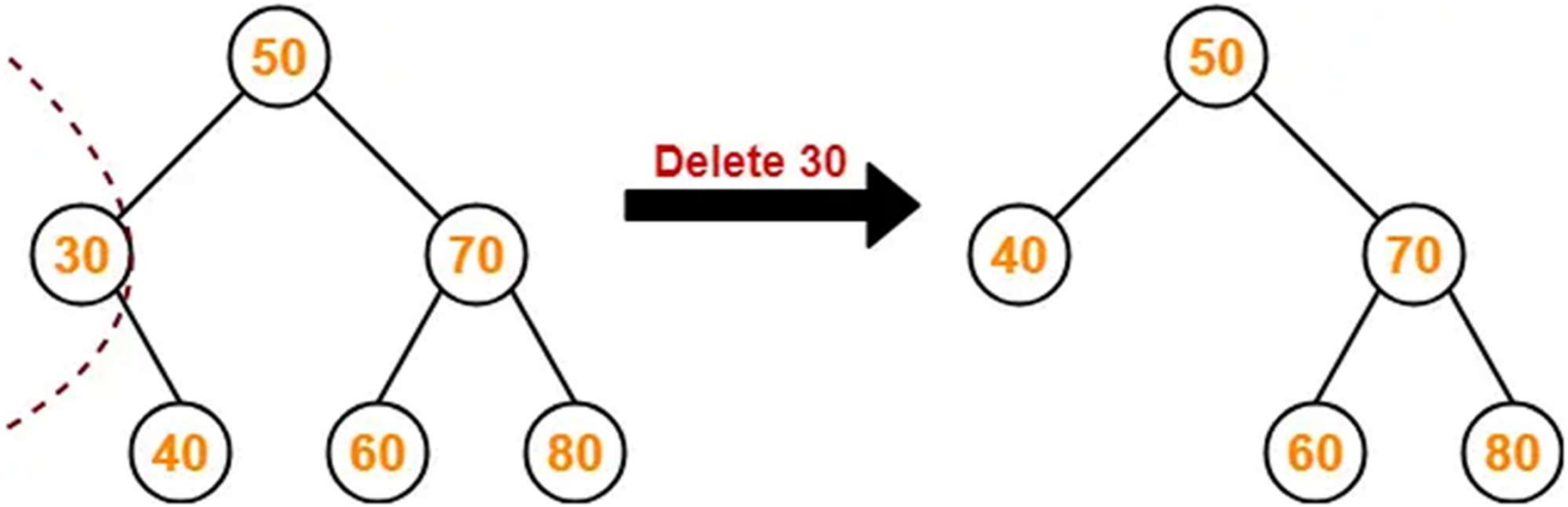
- Just remove / disconnect the leaf node that is to be deleted from the tree.



Deletion Operation

Case-02: Deletion of a Node having only One Child

- Just make the child of the deleting node, the child of its grandparent.



Deletion Operation

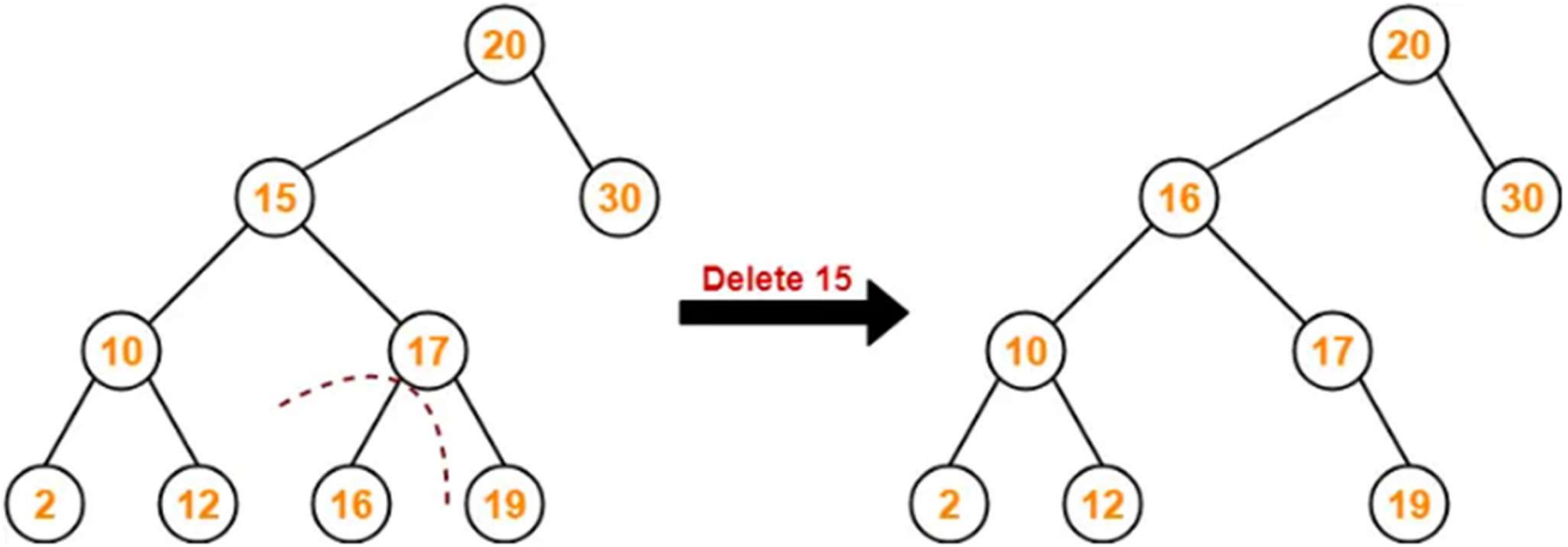
Case-03: Deletion of a Node having two children

A node with two children may be deleted from the BST in two ways-

Method-01:

- Visit to the right subtree of the deleting node.
- Pluck the least value element called as inorder successor.
- Replace the deleting element with its inorder successor.

Deletion Operation



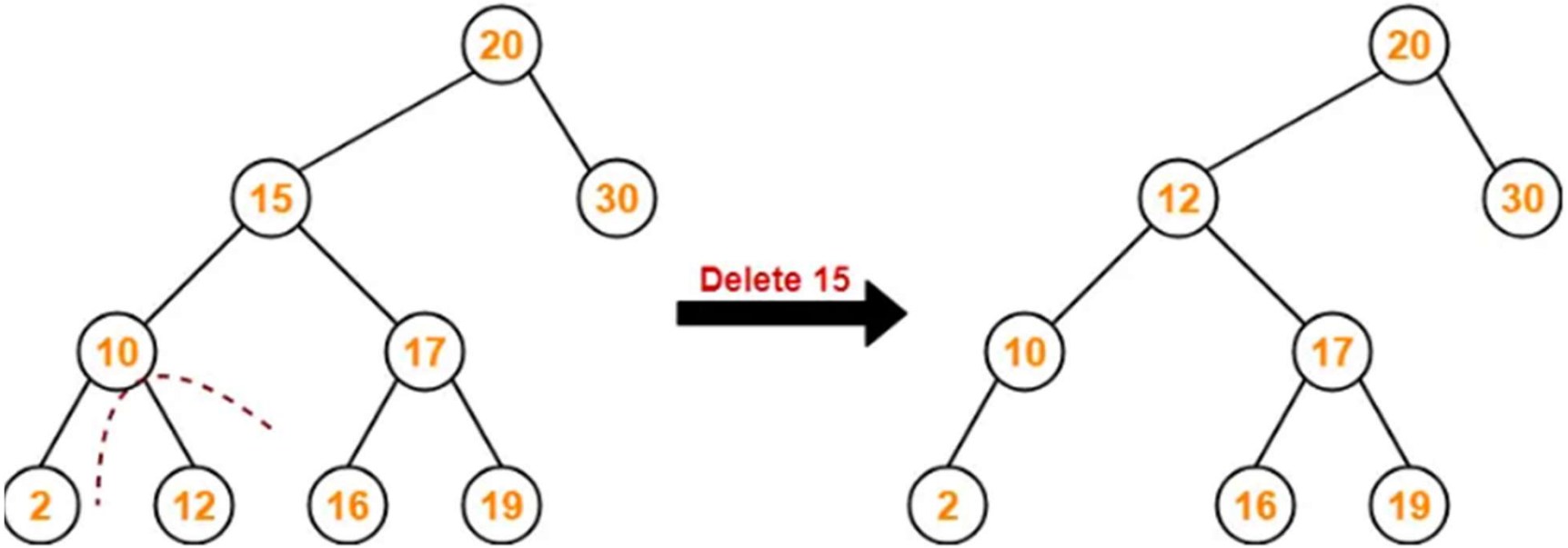
```python
def _delete(self, root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = self._delete(root.left, key)
    elif key > root.key:
        root.right = self._delete(root.right, key)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        root.key = self._min_value_node(root.right).key
        root.right = self._delete(root.right, root.key)
    return root

def _min_value_node(self, node):
    current = node
    while current.left is not None:
        current = current.left
    return current
```

Deletion Operation

Method-02:

- Visit to the left subtree of the deleting node.
- Pluck the greatest value element called as inorder predecessor.
- Replace the deleting element with its inorder predecessor.

Deletion Operation



Time Complexity in BST

- Time Complexity of all BST operations is $O(h)$, where h is the height of the tree.

Worst Case:

- When the height becomes n

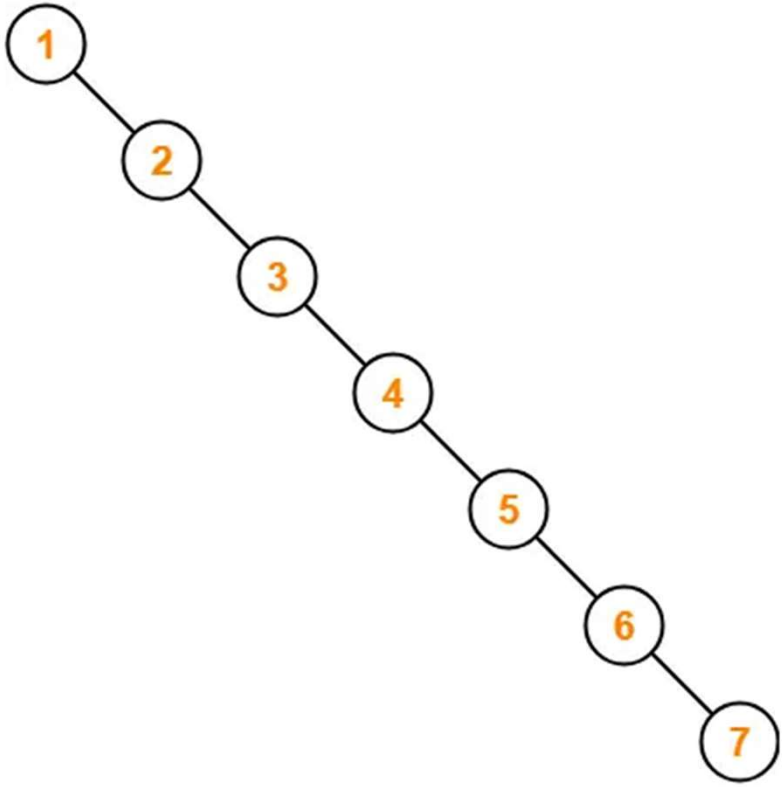
$$O(n)$$

Best Case:

- When the height becomes $\log n$. This occurs only when the tree is balanced.

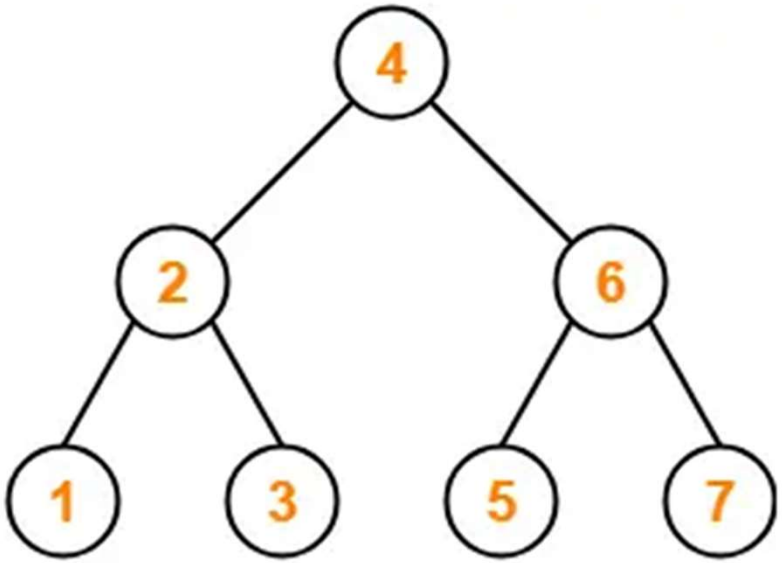
$$O(\log n)$$

Time Complexity in BST



Skewed Binary Search Tree

Time Complexity in BST



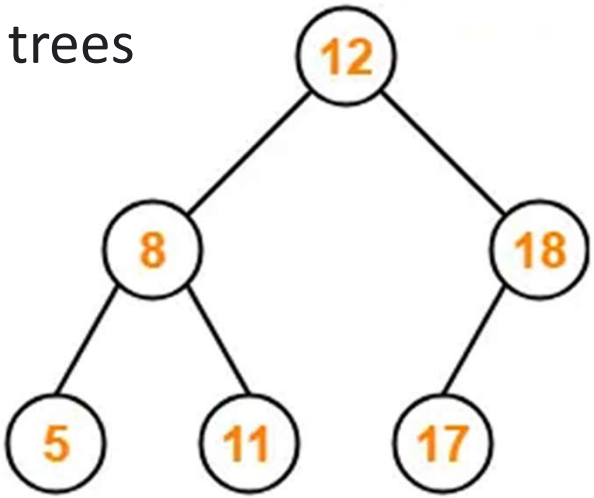
Balanced Binary Search Tree

Topics to be covered

- Constructing Binary Tree from Tree Traversals
- AVL Trees
- B-Tree
- Quad Tree
- Oct Tree

AVL Trees

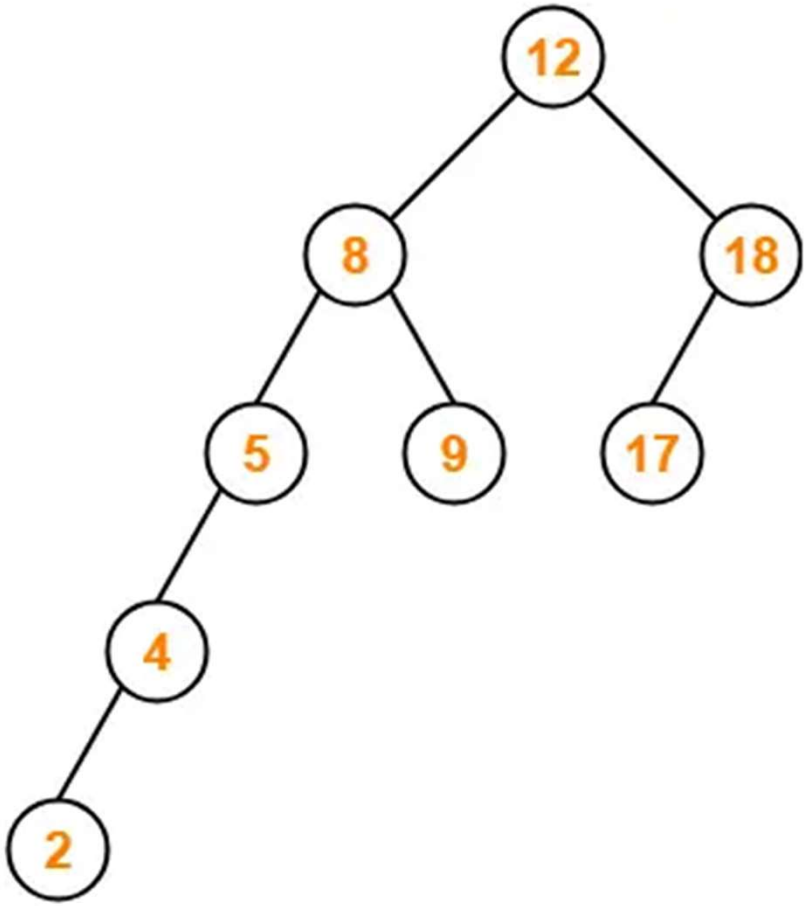
- Adelson, Velskii, and Landis (AVL)
- Special kind of Binary Search Tree (BST)
- Also called as self-balancing binary search trees



AVL Trees

- A Tree is an AVL tree because -
  - It is a binary search tree
  - The difference between the height of left subtree and right subtree of every node is **at most one**.

AVL Trees



Height of left subtree = 4
Height of right subtree = 2

Difference = 4 - 2 = 2 is greater than 1

AVL Trees

Balance Factor

- In AVL tree,
  - Balance factor is defined for every node.
  - Balance factor of a node = Height of its left subtree – Height of its right subtree
  - Balance factor of every node is either 0 or 1 or -1.

AVL Trees Operations

- **Search**
- **Insertion**
- **Deletion**

After performing any operation on AVL tree, the balance factor of each node is checked.

AVL Trees Operations

Case 1

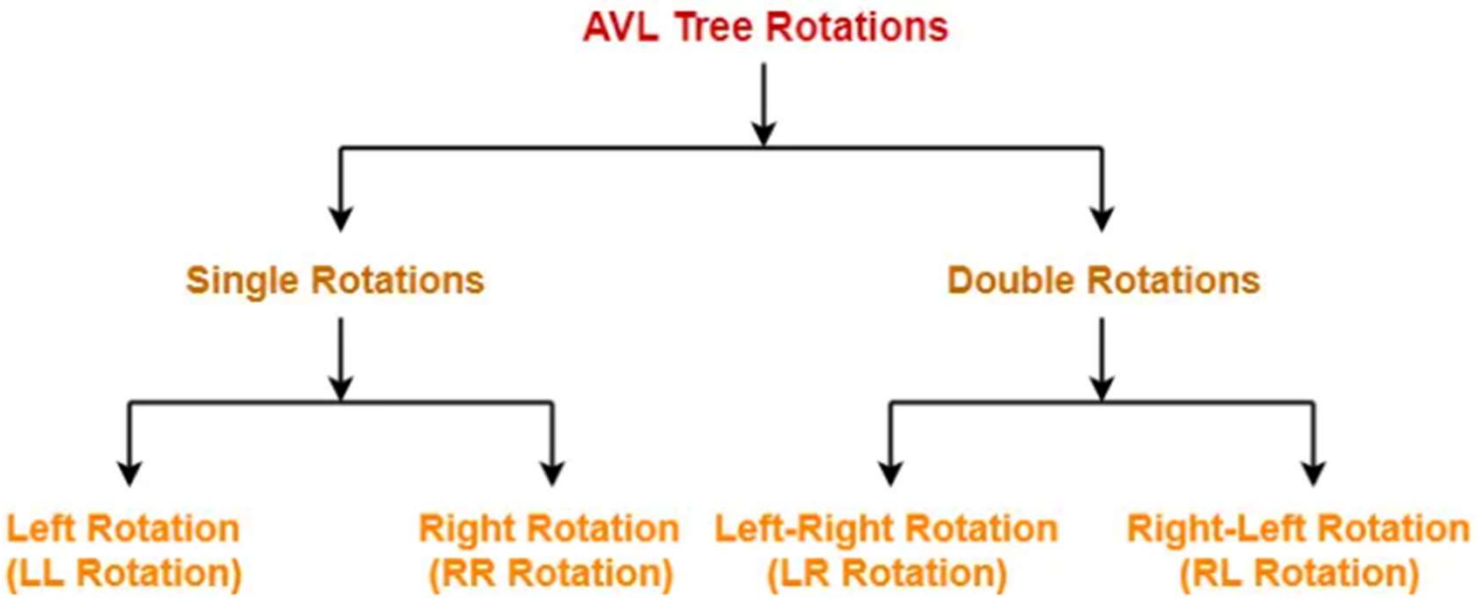
- After the operation, the balance factor of each node is either 0 or 1 or -1.
- In this case, the AVL tree is considered to be balanced.
- The operation is concluded.

Case 2

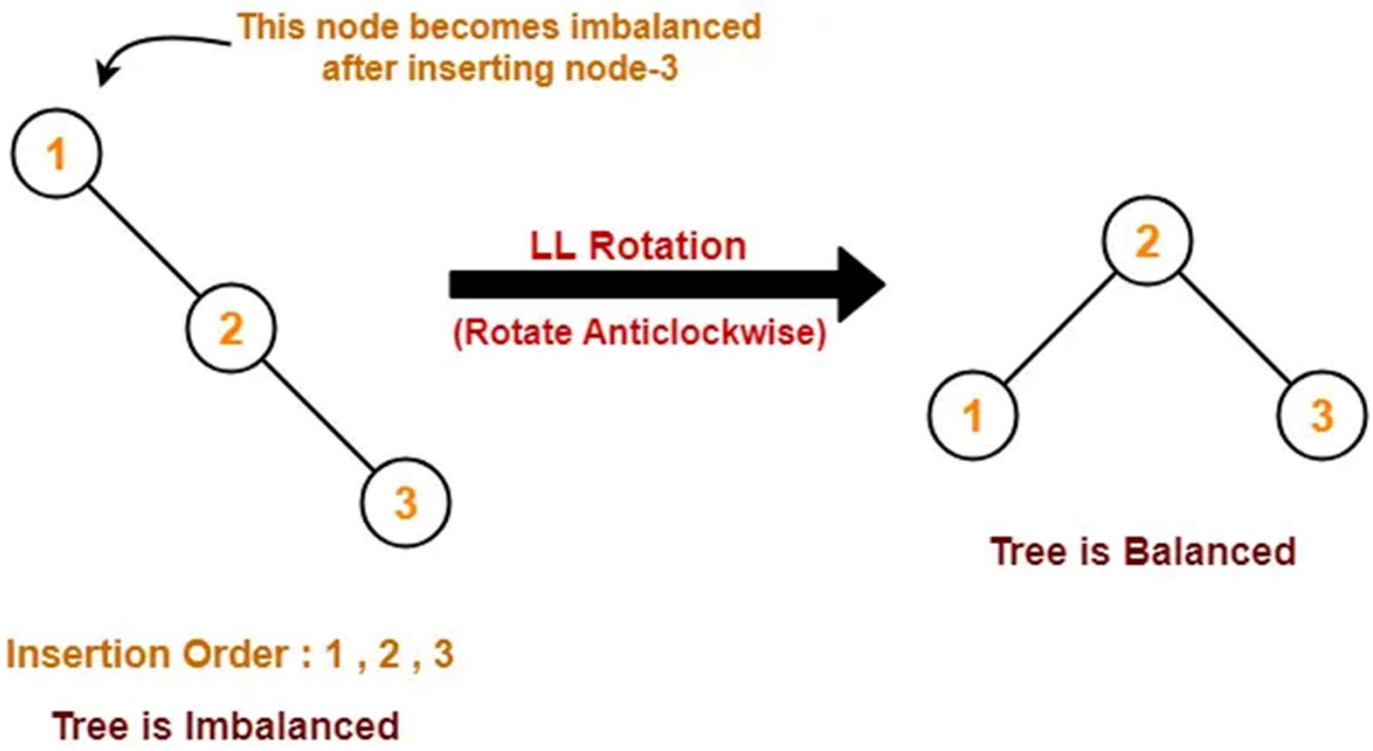
- After the operation, the balance factor of at least one node is not 0 or 1 or -1.
- In this case, the AVL tree is considered to be imbalanced.
- Rotations are then performed to balance the tree.

AVL Tree Rotations

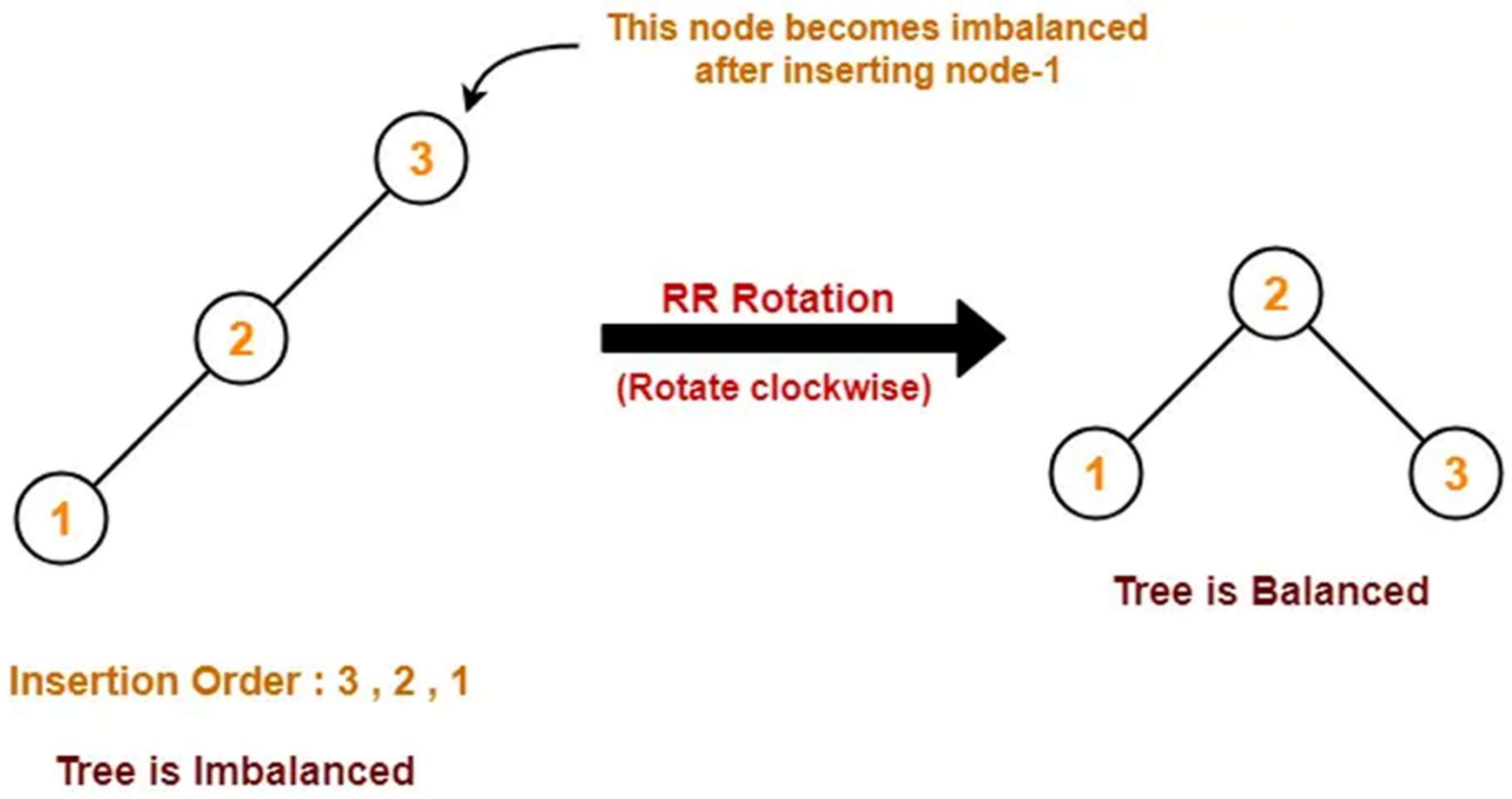
- Rotation is the process of moving the nodes to make tree balanced.
- There are 4 kinds of rotations possible in AVL Trees.



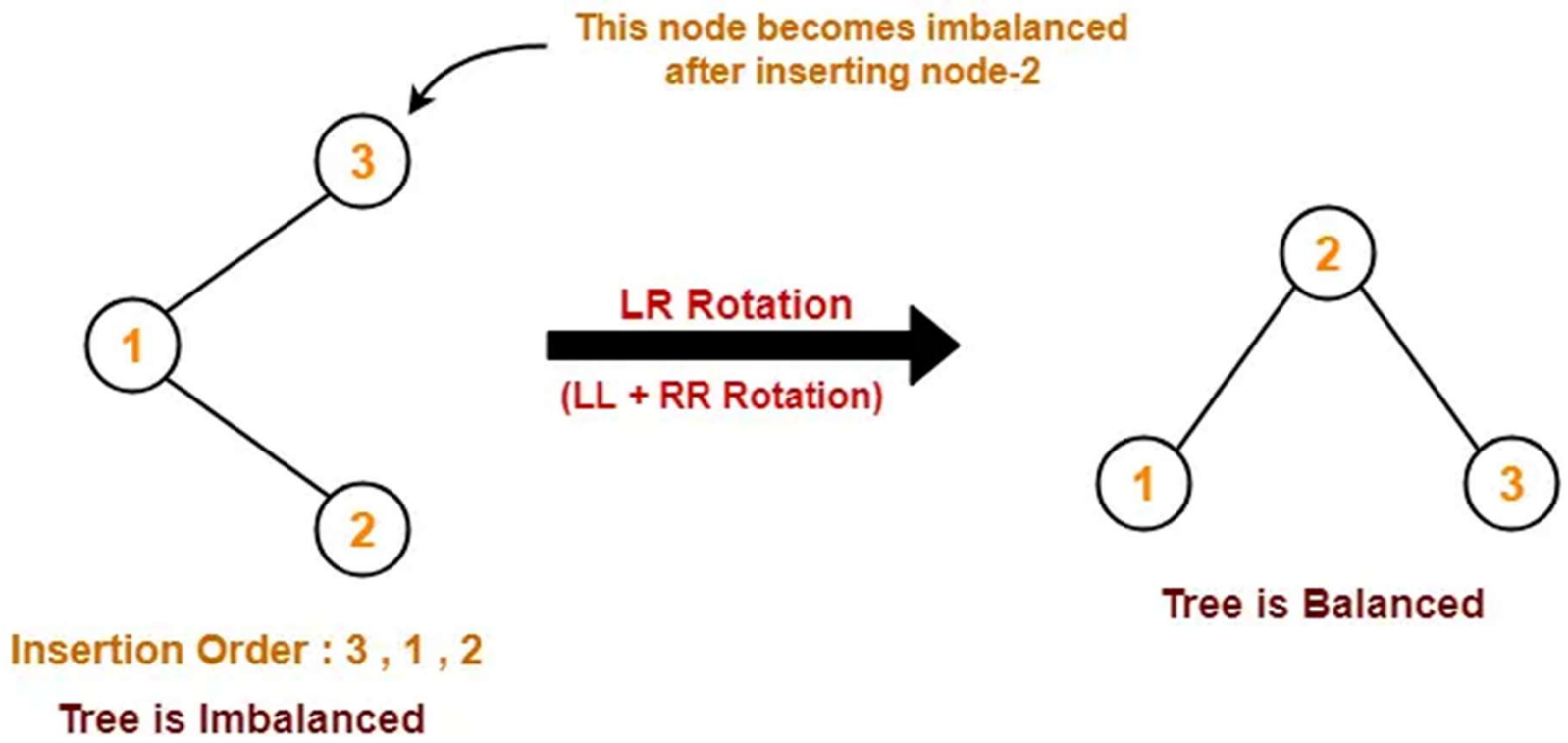
LL Rotation


This node becomes imbalanced
after inserting node-3
1
2
3
LL Rotation
(Rotate Anticlockwise)
2
1
3
Tree is Balanced
Insertion Order : 1 , 2 , 3
Tree is Imbalanced

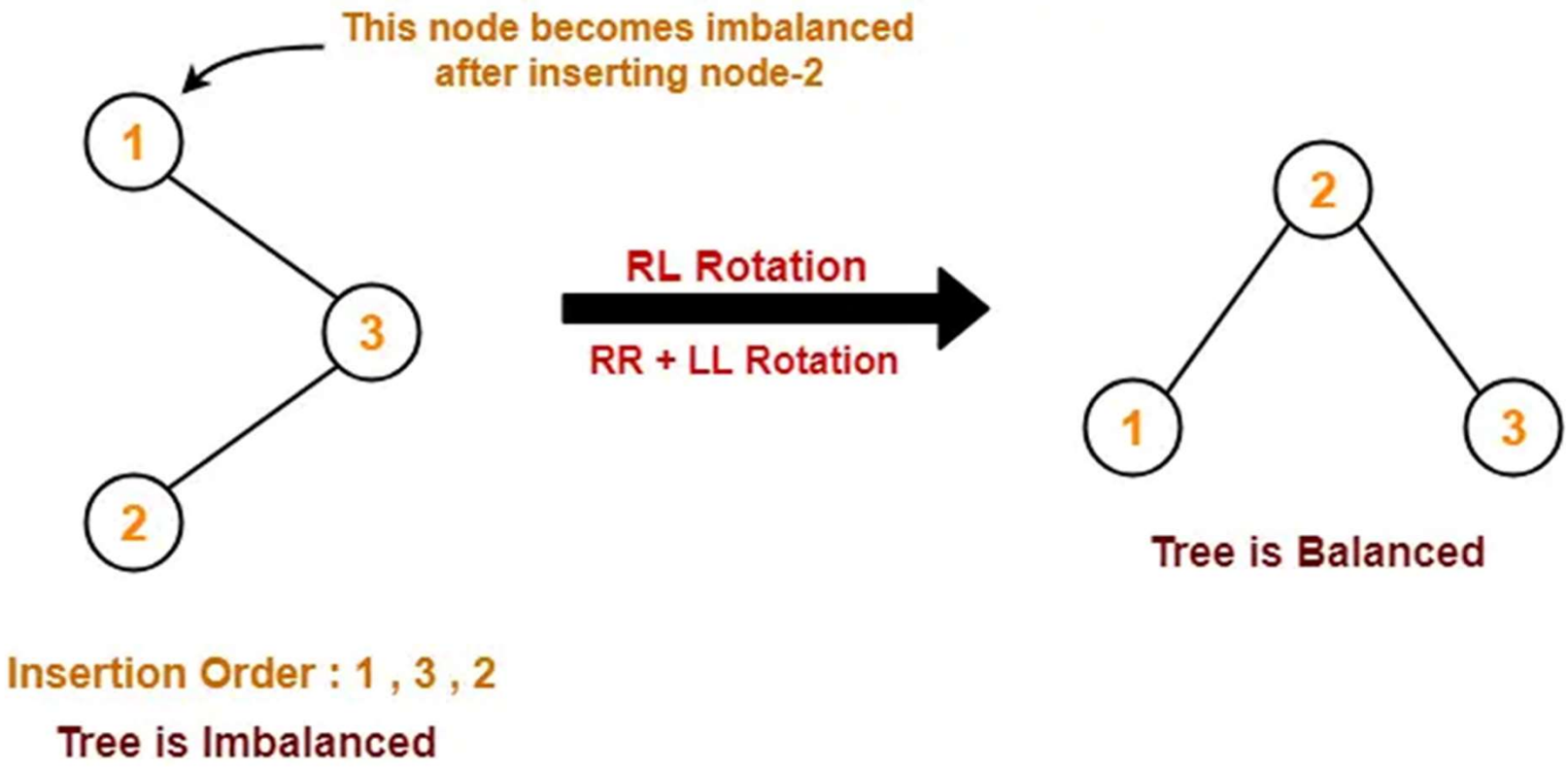
RR Rotation


This node becomes imbalanced
after inserting node-1
3
2
1
RR Rotation
(Rotate clockwise)
2
1
3
Tree is Balanced
Insertion Order : 3 , 2 , 1
Tree is Imbalanced

LR Rotation


This node becomes imbalanced
after inserting node-2
3
1
2
LR Rotation
(LL + RR Rotation)
2
1
3
Tree is Balanced
Insertion Order : 3 , 1 , 2
Tree is Imbalanced

RL Rotation


This node becomes imbalanced
after inserting node-2
1
3
2
RL Rotation
RR + LL Rotation
2
1
3
Tree is Balanced
Insertion Order : 1 , 3 , 2
Tree is Imbalanced

AVL Tree Properties

Property 1: Maximum possible number of nodes in AVL tree of height H is

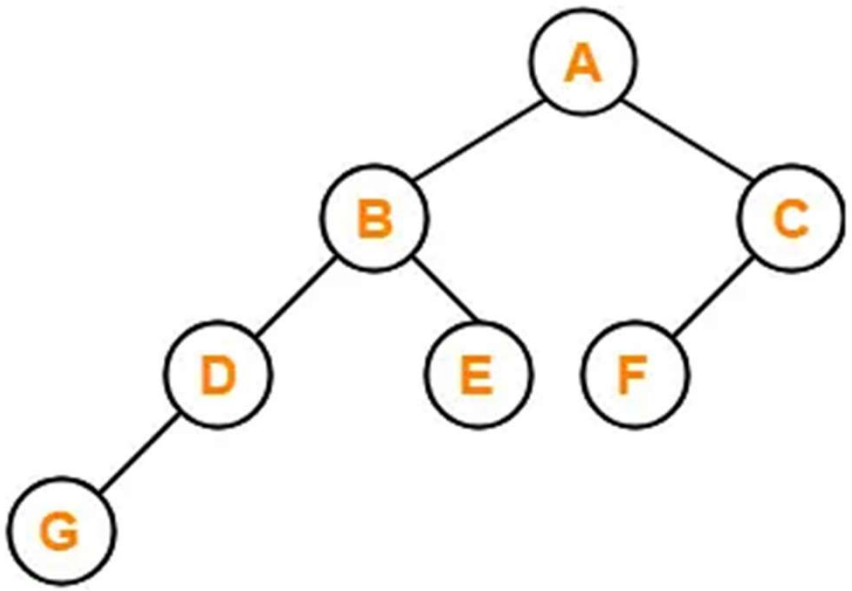
$$2^{H+1} - 1$$

Property 2: Minimum number of nodes in AVL Tree of height H is given by a recursive relation:

$$\mathrm{N(H) = N(H-1) + N(H-2) + 1}$$

where $N(0) = 1, N(1) = 2$

AVL Tree Properties



AVL Tree

(Height = 3)

- Minimum possible number of nodes in AVL tree of height 3 = 7

AVL Tree Properties

Property 3: Minimum possible height of AVL Tree using N nodes = $\lceil \log N \rceil$

Property 4: Maximum height of AVL Tree using N nodes is calculated using recursive relation-

$N(H) = N(H-1) + N(H-2) + 1$

$N(0) = 1$

$N(1) = 2$

Problems

1. Find the minimum number of nodes required to construct AVL Tree of height = 3.
2. Find the minimum number of nodes required to construct AVL Tree of height = 4.
3. What is the maximum height of any AVL tree with 10 nodes?
4. What is the maximum height of any AVL tree with 77 nodes?

Insertion in AVL Tree

To insert an element in the AVL tree, follow the following steps-

- Insert the element in the AVL tree in the same way the insertion is performed in BST.
- After insertion, **check the balance factor** of each node of the resulting tree.

Insertion in AVL Tree

Case 1

- After the insertion, the balance factor of each node is either 0 or 1 or -1.
- In this case, the AVL tree is considered to be balanced.
- Conclude the operation.
- Insert the next element if any.

Case 2

- After the insertion, the balance factor of at least one node is not 0 or 1 or -1.
- In this case, the AVL tree is considered to be imbalanced.
- Perform the suitable rotation to balance the tree.
- After the tree is balanced, insert the next element if any.

Insertion in AVL Tree

Rules to remember before performing Insertion Operation.

Rule 1: After inserting an element in the existing AVL tree,

- Balance factor of only those nodes will be affected that lies on the path from the newly inserted node to the root node.

Rule 2: To check whether the AVL tree is still balanced or not after the insertion,

- There is no need to check the balance factor of every node.
- Check the balance factor of only those nodes that **lies on the path from the newly inserted node** to the root node.

Insertion in AVL Tree

Rule 3: After inserting an element in the AVL tree,

- If tree becomes imbalanced, then there exists one particular node in the tree by balancing which the entire tree becomes balanced automatically.
- To rebalance the tree, balance that particular node.

To find that particular node,

- Traverse the path from the newly inserted node to the root node.
- Check the balance factor of each node that is encountered while traversing the path.
- The first encountered imbalanced node will be the node that needs to be balanced.

Insertion in AVL Tree

To balance that node,

- Count three nodes in the direction of leaf node.
- Then, use the concept of AVL tree rotations to re balance the tree.

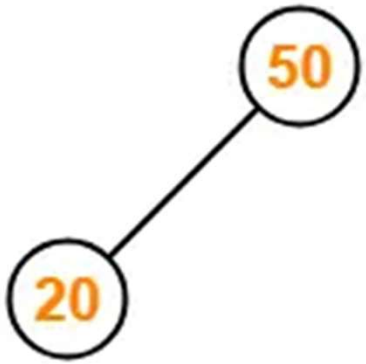
Example

- Construct AVL Tree for the following sequence of numbers-

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

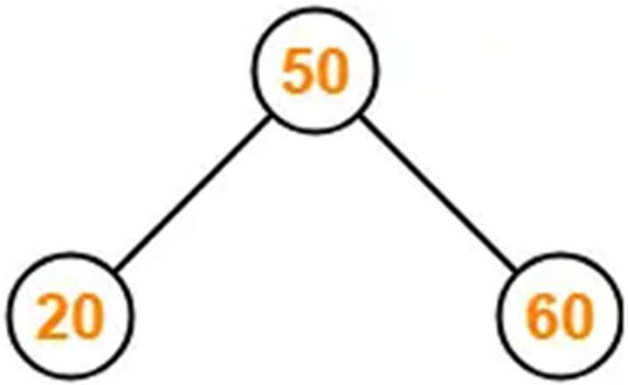
Insert 50

Insert 20



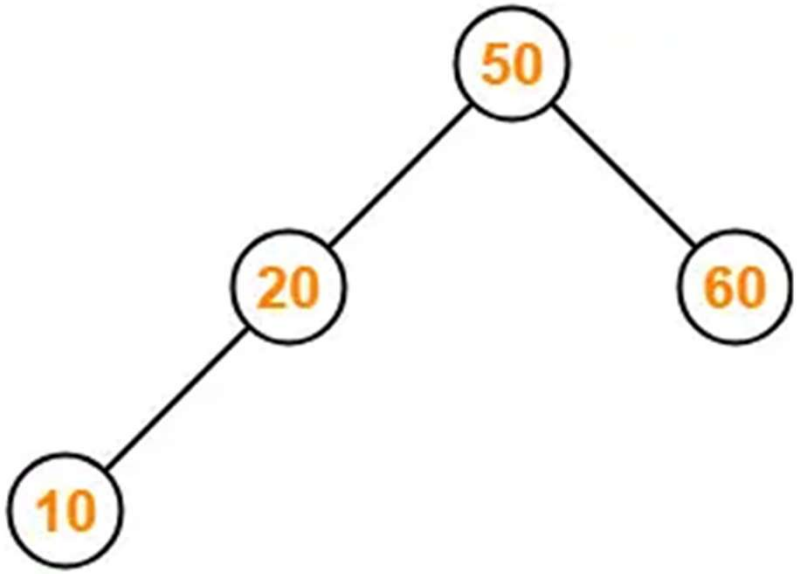
Tree is Balanced

Insert 60



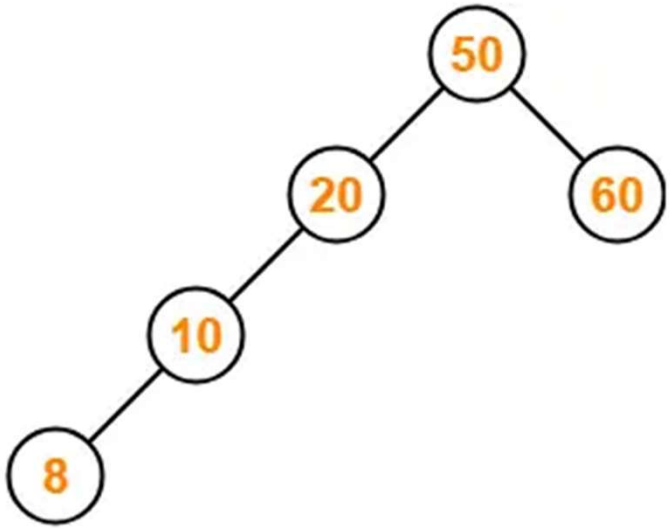
Tree is Balanced

Insert 10



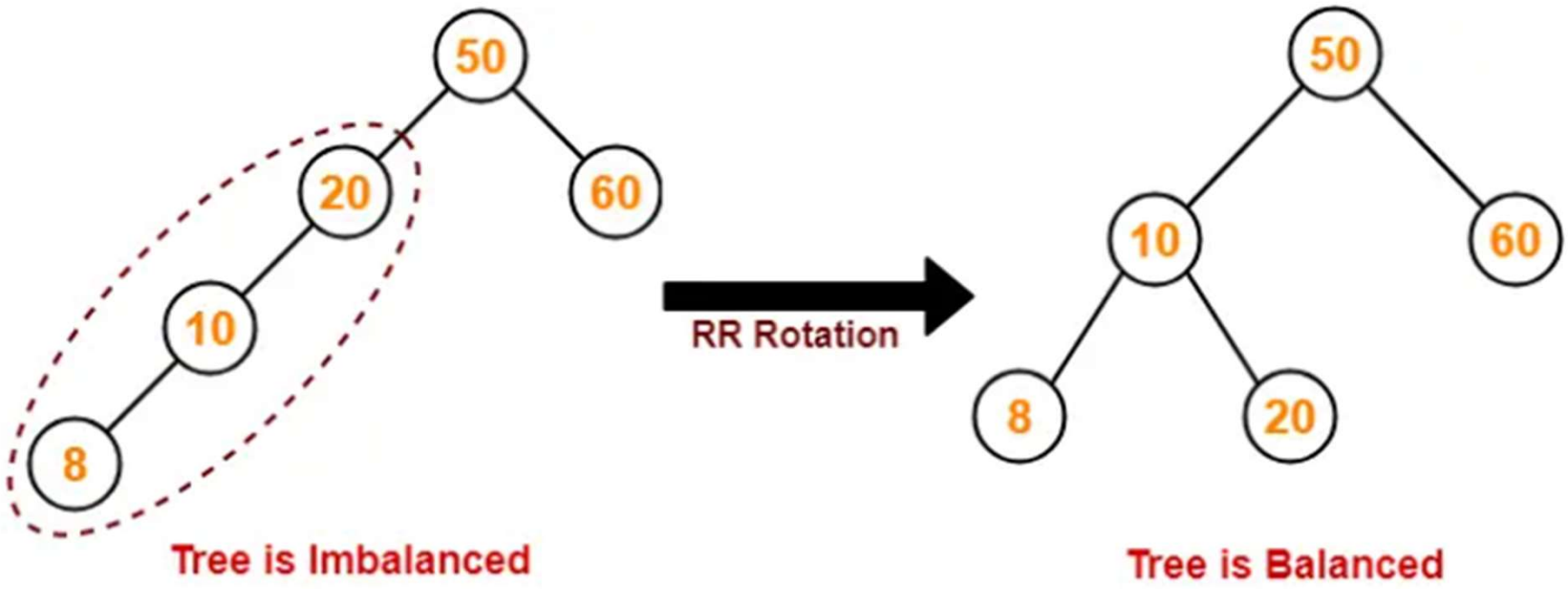
Tree is Balanced

Insert 8

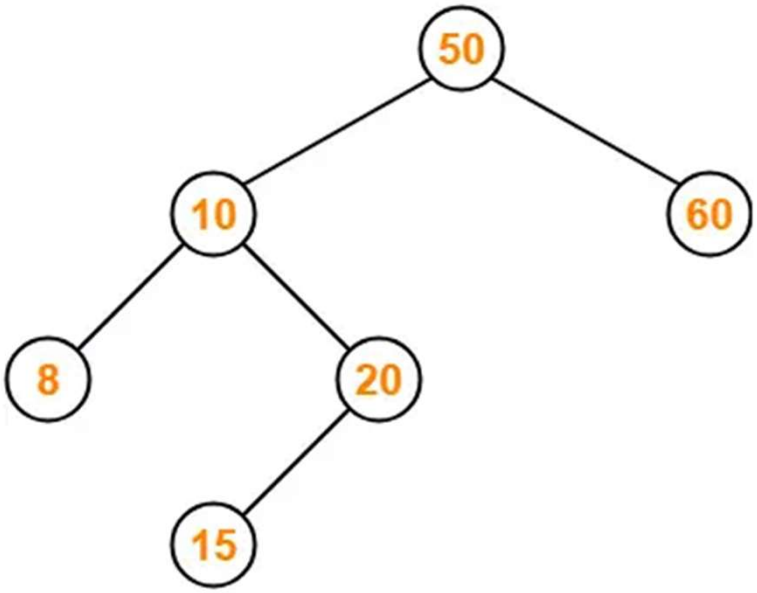


Tree is Imbalanced

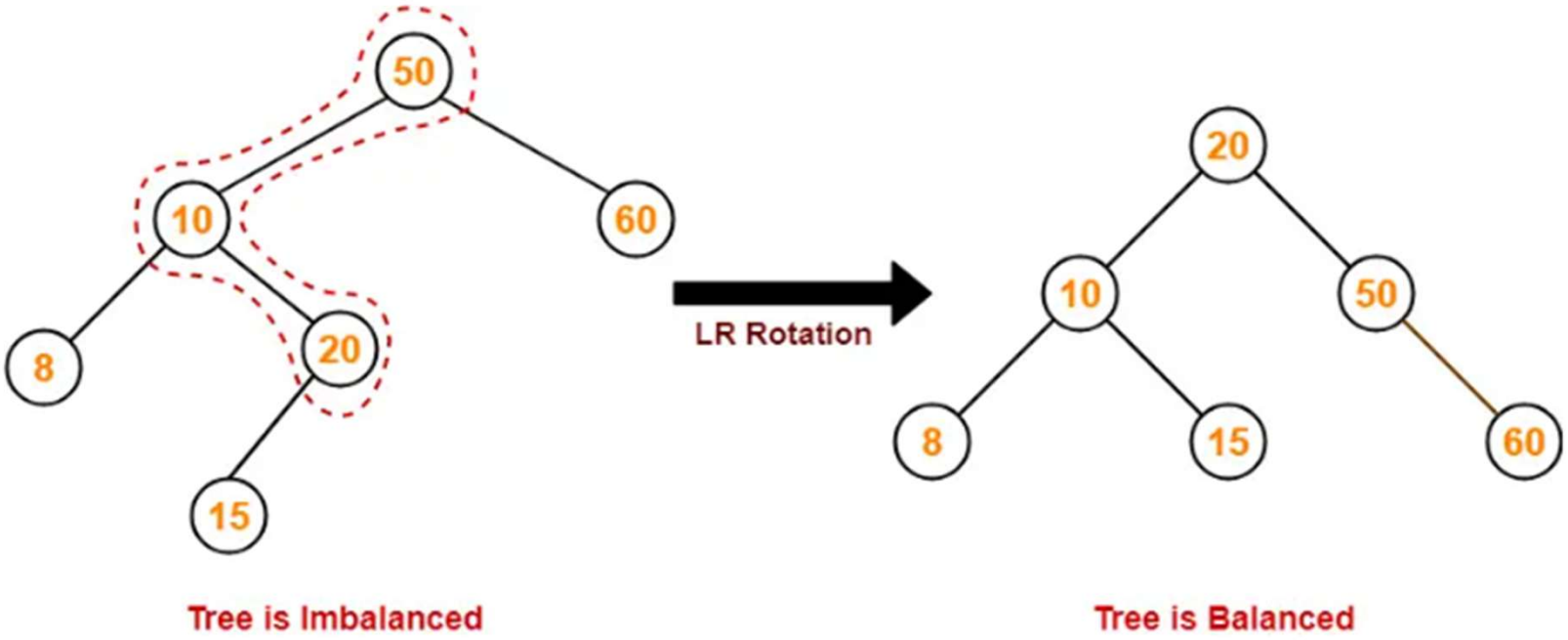
Balanced Factor at Node 20 is 2



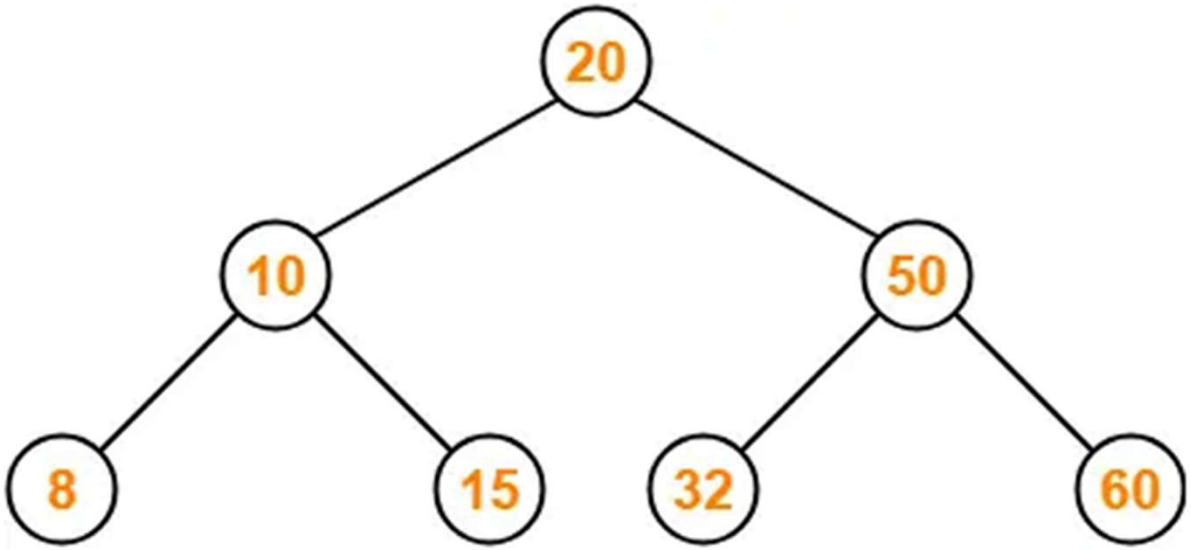
Insert 15



Tree is Imbalanced

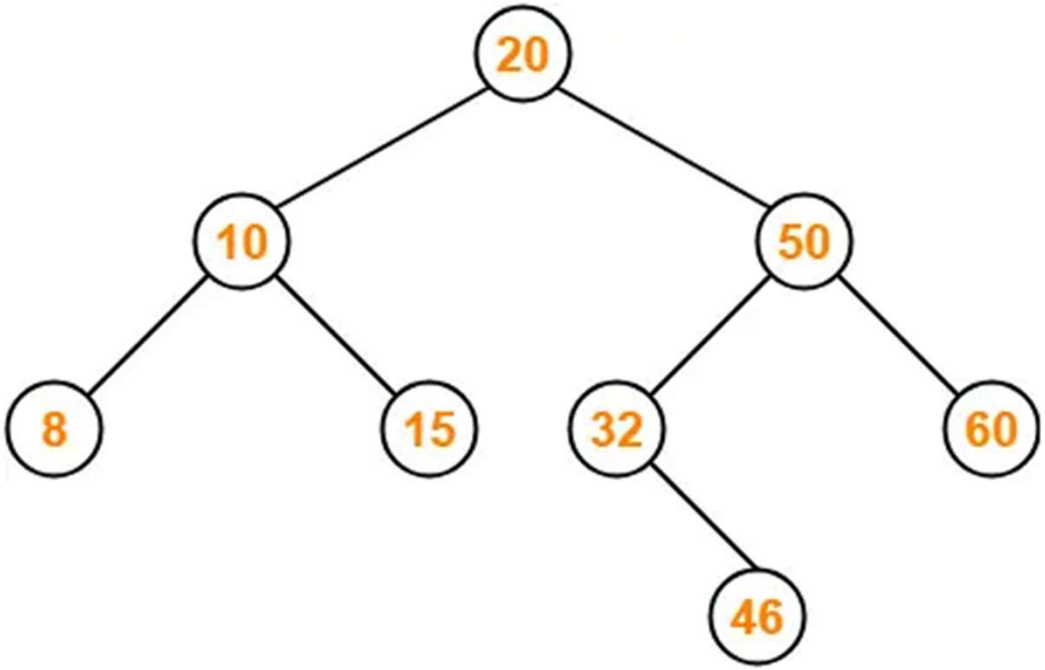

50
10
60
8
20
15
LR Rotation
20
10
50
8
15
60
Tree is Imbalanced
Tree is Balanced

Insert 32



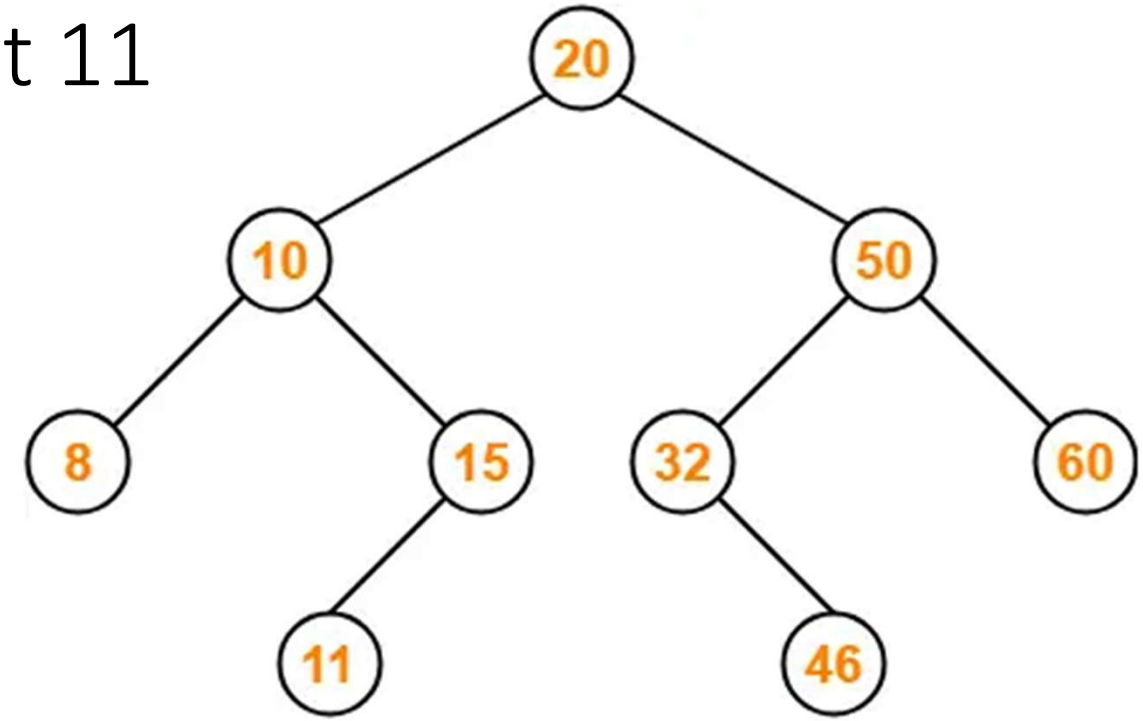
Tree is Balanced

Insert 46



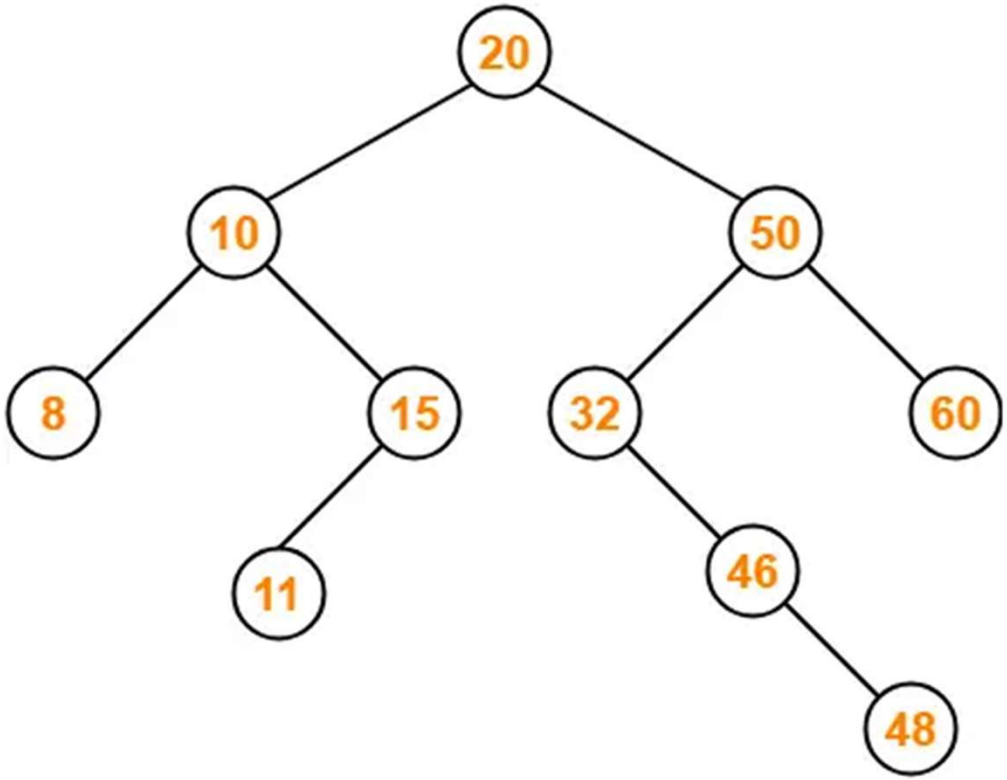
Tree is Balanced

Insert 11

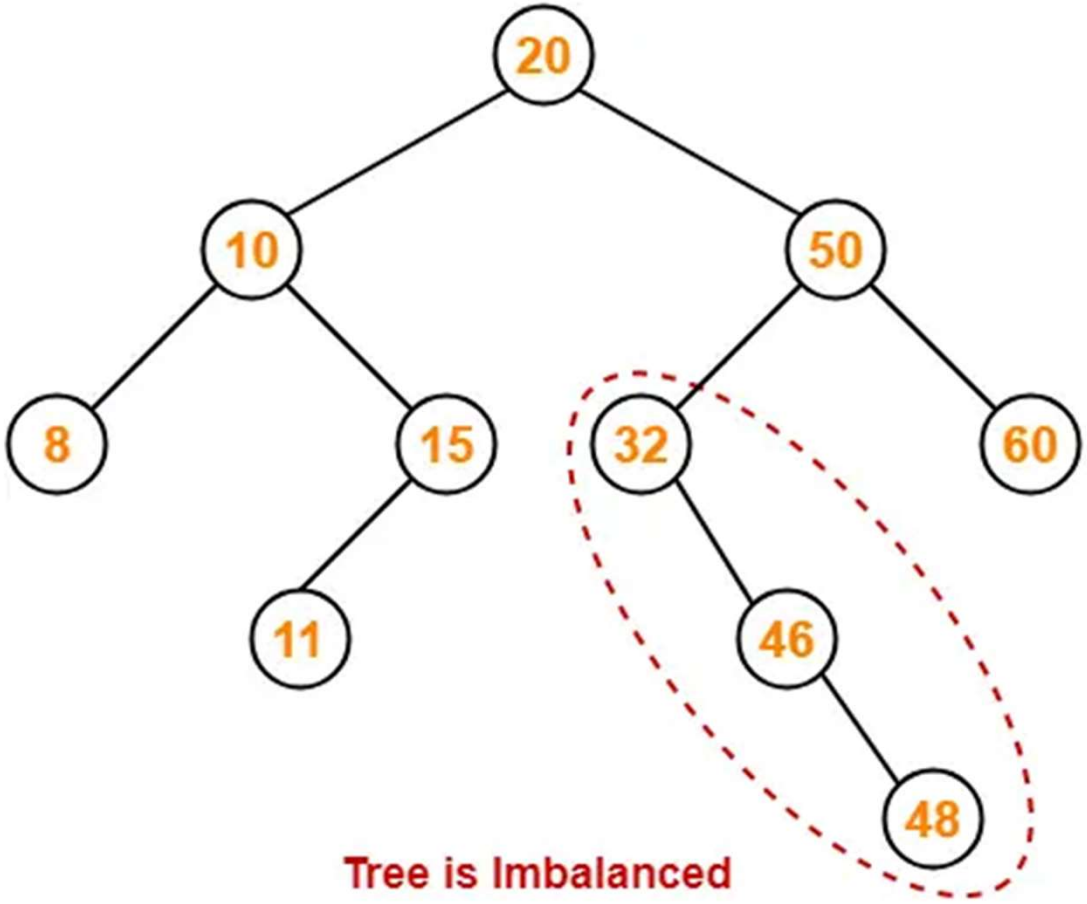


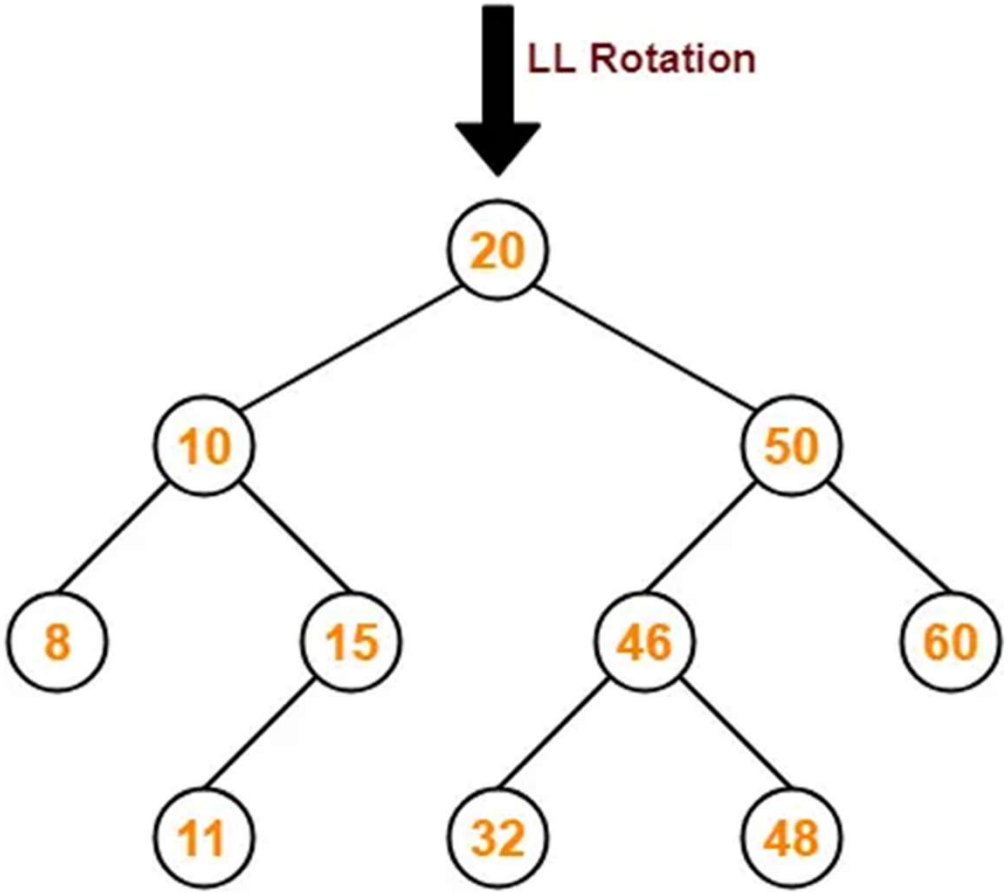
Tree is Balanced

Insert 48



Tree is imbalanced


20
10
50
8
15
32
60
11
46
48
Tree is Imbalanced



Tree is Balanced