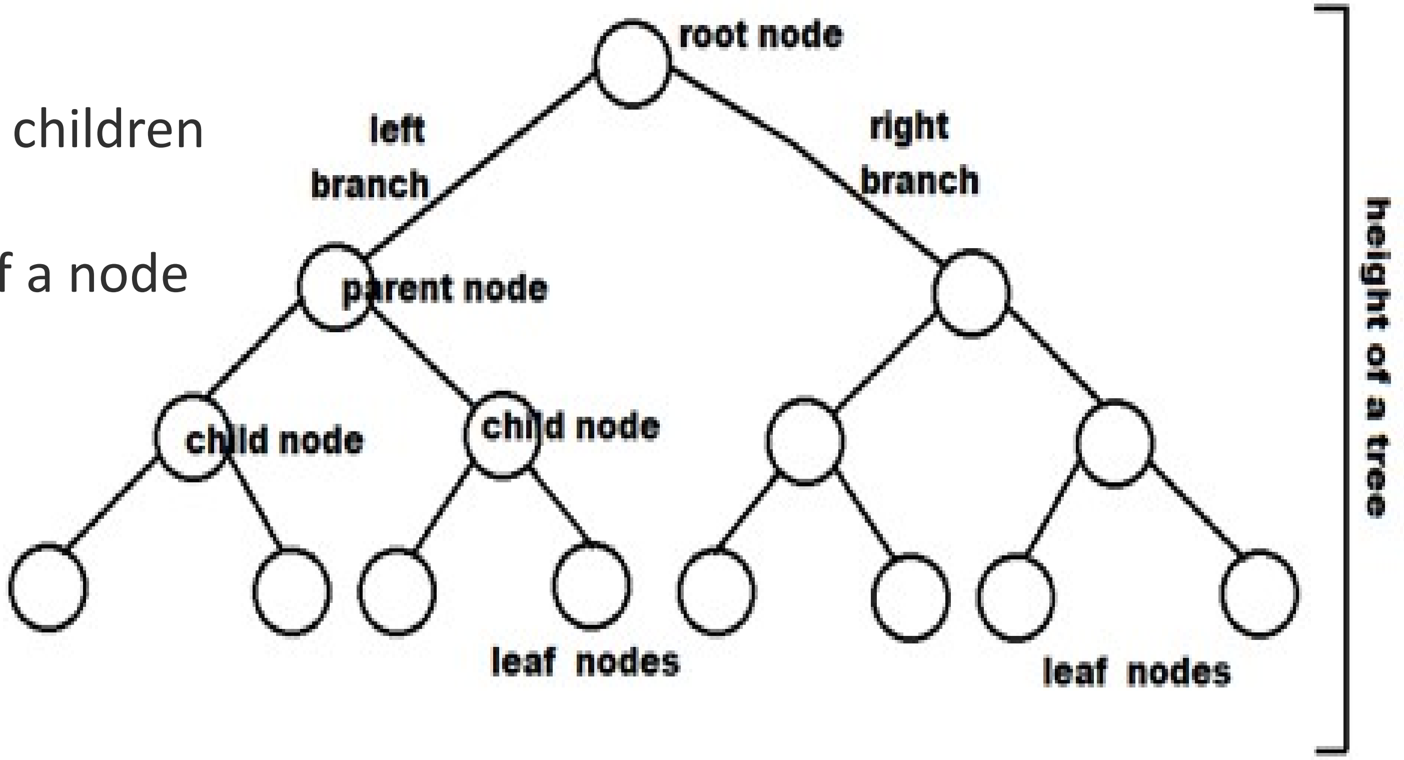
Binary Tree and Binary Search Tree and AVL and Red Black Tree

Tree

- ❑ Trees are Non-Linear and Abstract Data Structures that simulate a hierarchical structure.
- ❑ There is one and only one path between every pair of vertices in a tree.
- ❑ There is (n-1) edges in a tree with n vertices.
  - ❑ Any connected graph with n vertices and (n-1) edges is a tree.

- ❑ **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.
- ❑ Height of a node
- ❑ Height of the Tree
- ❑ Depth of a node
- ❑ Forest

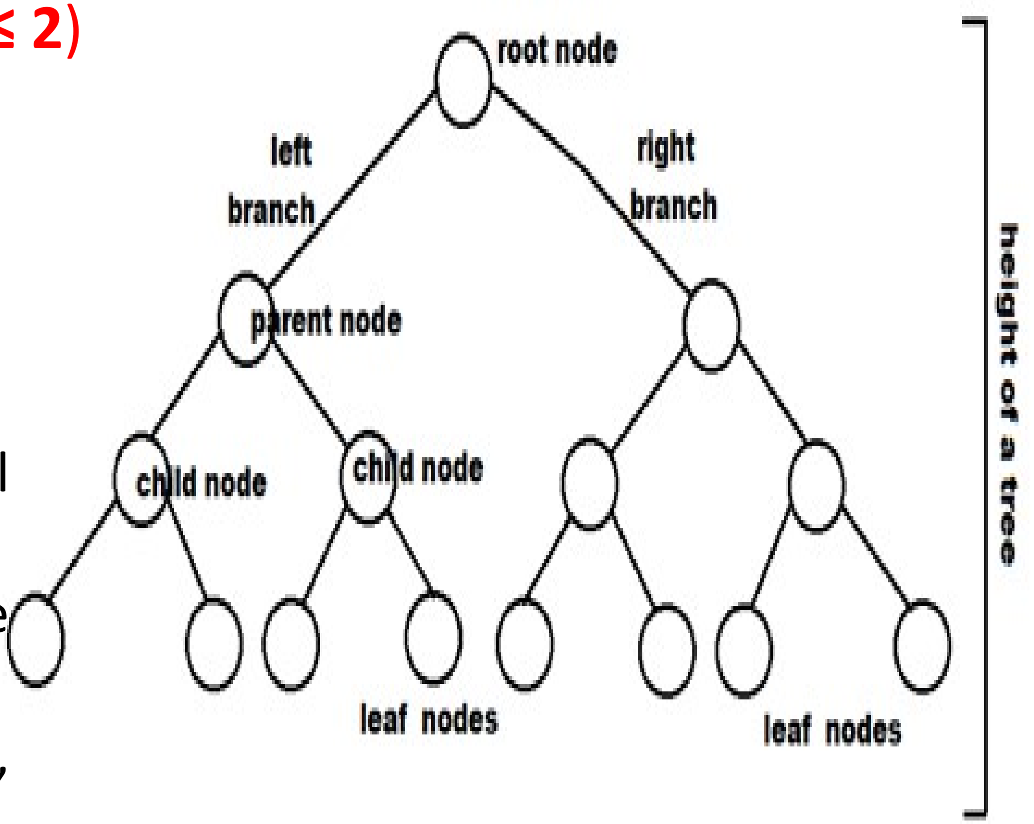


Binary Tree

- ❑ A binary tree is a data structure composed of nodes, where each node contains a value and two references to other nodes, called the left child and the right child.
- ❑ In a binary tree each node can have n children **($0 \le n \le 2$)**

- At each level of *i*, the maximum number of nodes is 2^i.
- If height = 4, the maximum number of nodes (1+2+4+8+16) = 31
  - at height h is $2^0 + 2^1 + 2^2 + \dots 2^h = 2^{h+1} - 1$.

- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum.
- On the other hand, if the number of nodes is maximum, then the height of the tree would be minimum.



Binary Tree

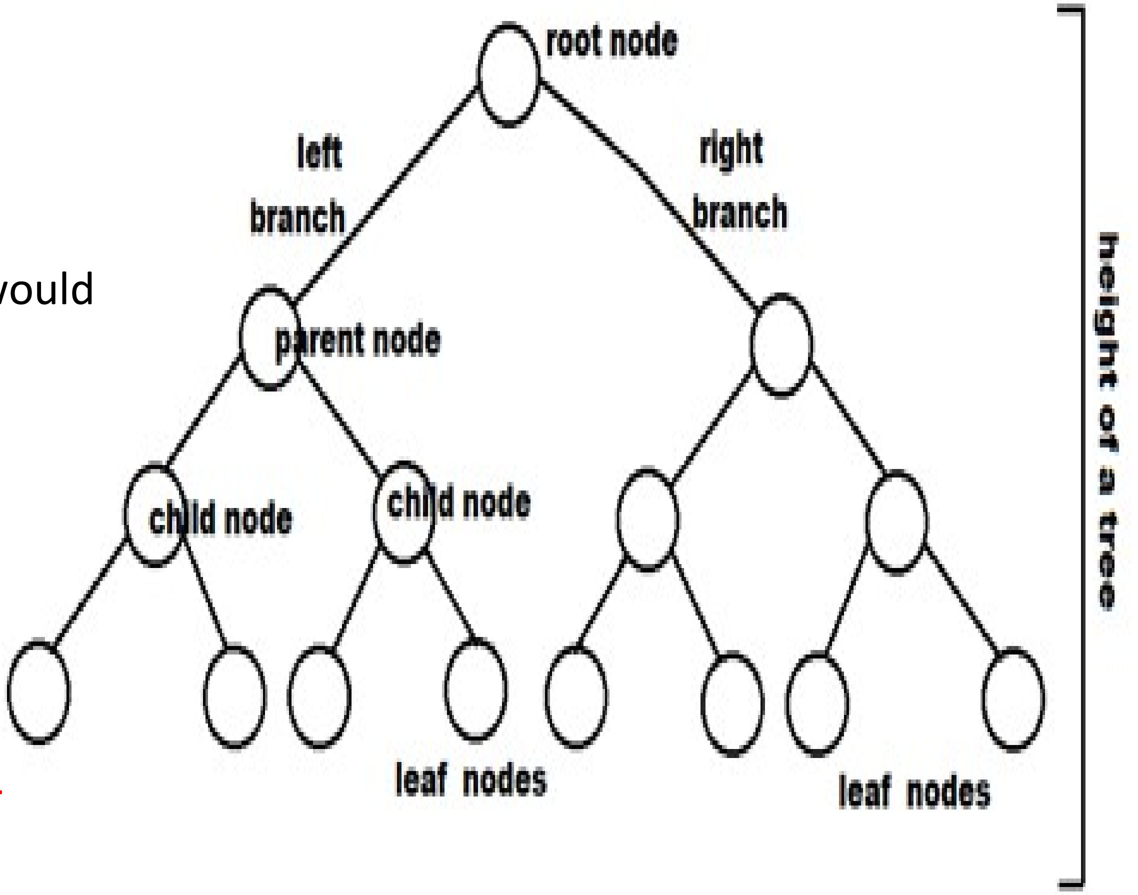
- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum.
  - Let's n is number of nodes in the binary tree.
  - Then, n = h+1; meaning h= n-1

- At each level of *i*, the maximum number of nodes is 2^i.
- If height = 4, the maximum number of nodes (1+2+4+8+16) = 31
  - at height h is $2^0 + 2^1 + 2^2 + \ldots 2^h = 2^{h+1} - 1$.

- if the number of nodes is maximum, then the height of the tree would be minimum.

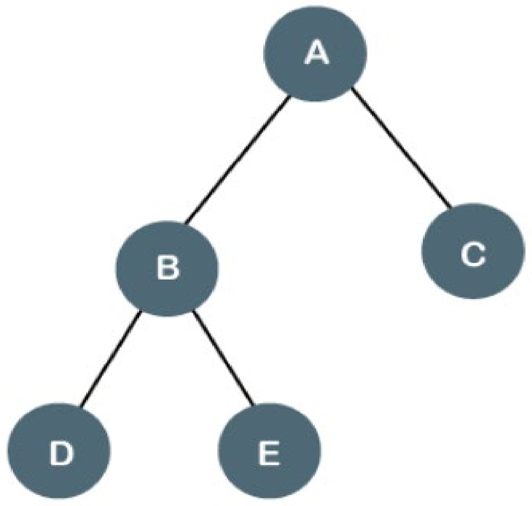
Let's n is number of nodes in the binary tree.

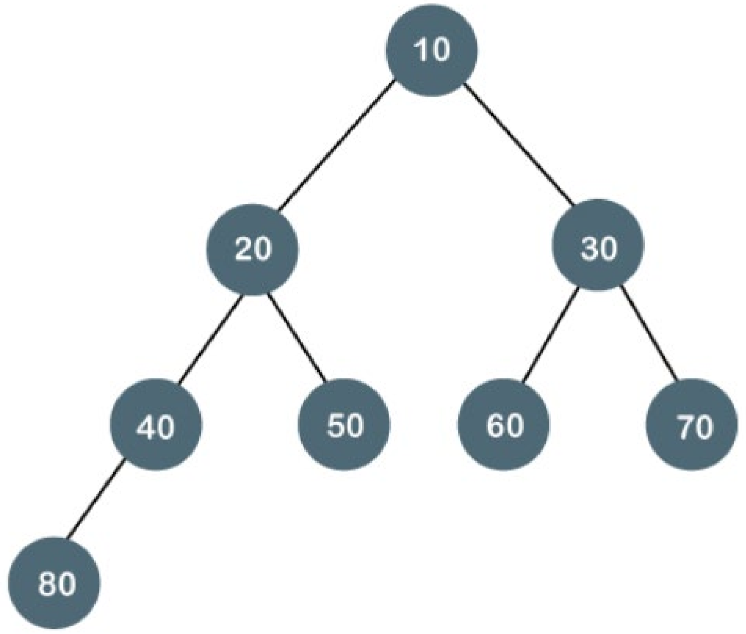
i.e. $n = 2^{h+1} - 1$;
hence, minimum height, $\mathbf{h = \log_2(n+1) - 1}$



Types of Binary tree

- ❑ Full binary tree: A full binary tree is a binary tree in which every node has either two children or no children.
    - ❑ each node must contain 2 children except the leaf nodes.

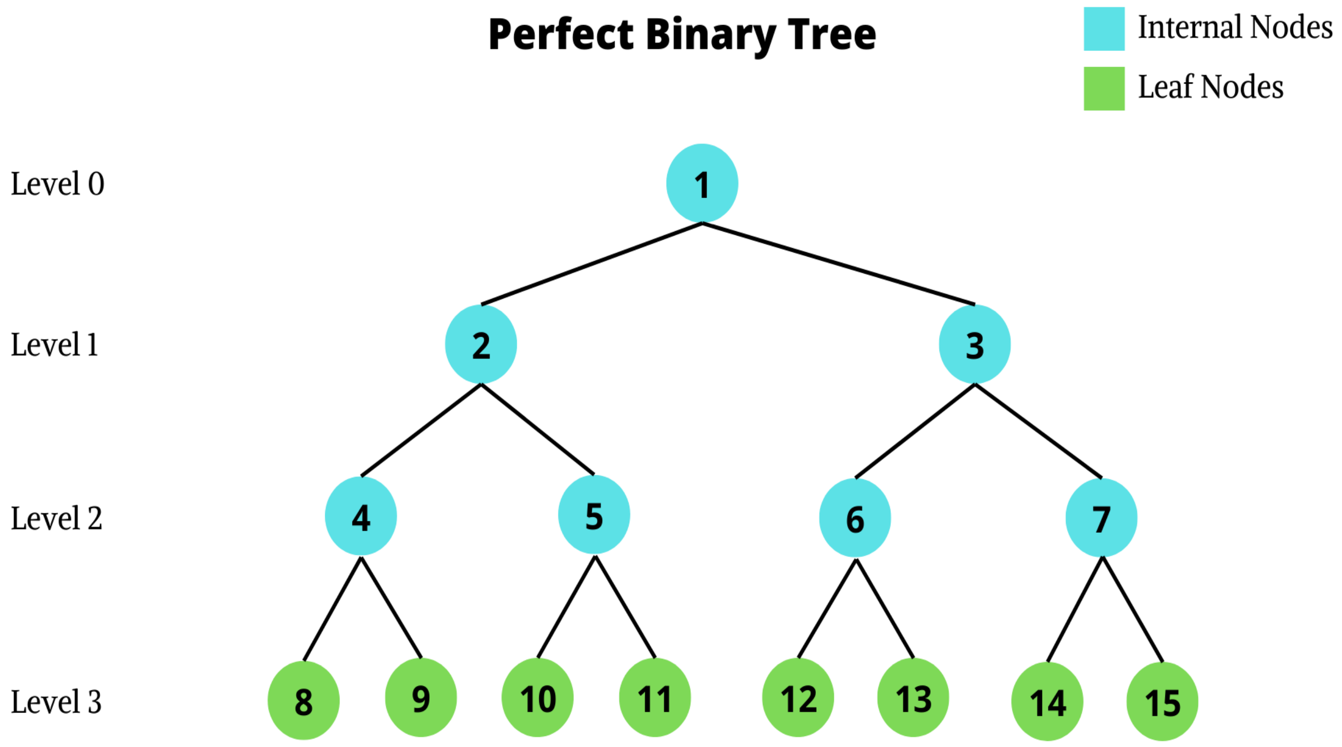




- ❑ Complete binary tree: A complete binary tree is a binary tree in which all levels except the last are completely filled
    - ❑ All of the nodes must be as far to the left as feasible in the last level. The nodes should be added from the left

Types of Binary tree

Perfect binary tree: A perfect binary tree is a binary tree in which all internal nodes have two children and all leaf nodes are at the same level.

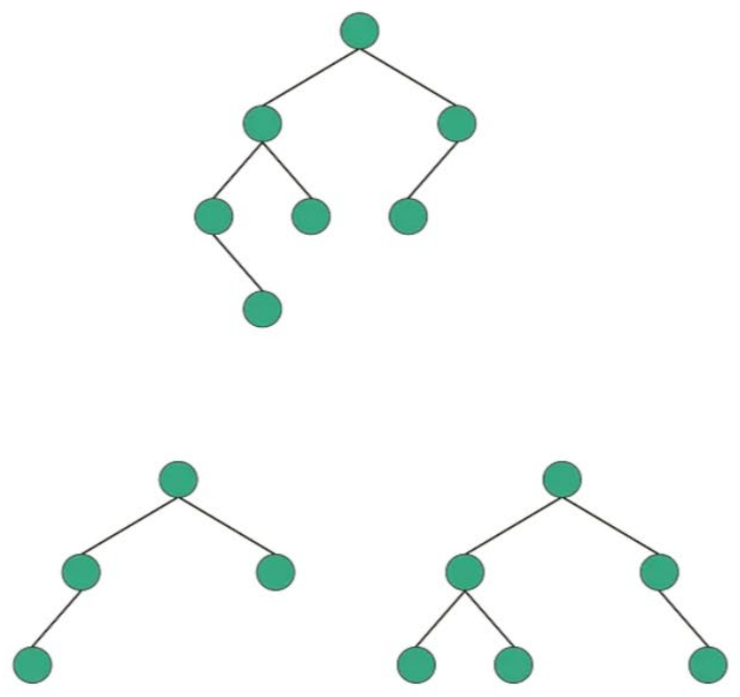


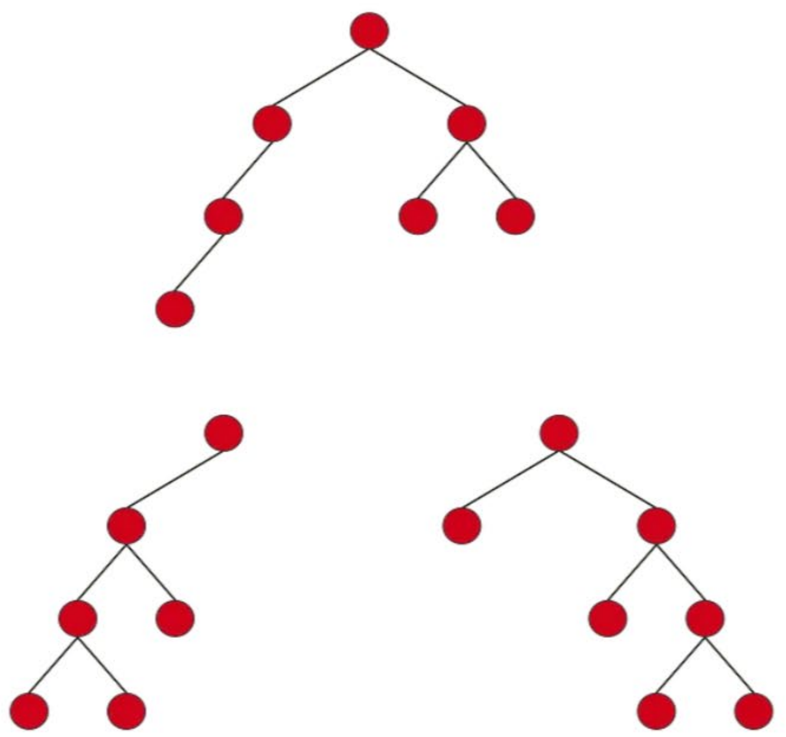
Types of Binary tree

Balanced binary tree: A balanced binary tree is a binary tree in which the heights of the left and right subtrees of every node differ by at most one.

Some operations in a an unbalanced tree may take longer than O(log n) time

AVL Tree and Red-Black Tree are well-known data structure to generate/maintain Balanced Binary Search Tree.

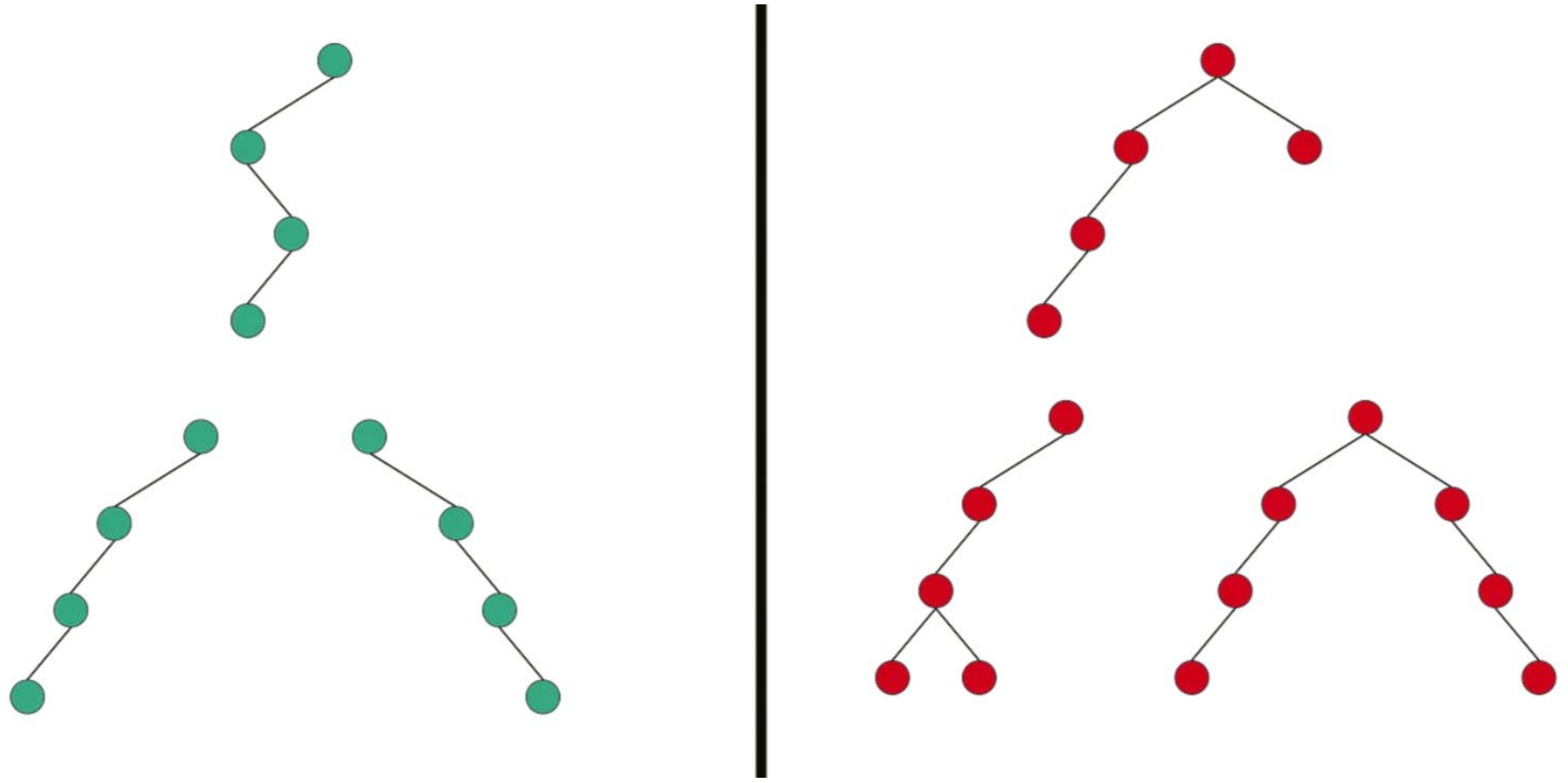




Types of Binary tree

Degenerate (or pathological) tree: A degenerate (or pathological) tree is a tree in which each parent node has only one associated child node.

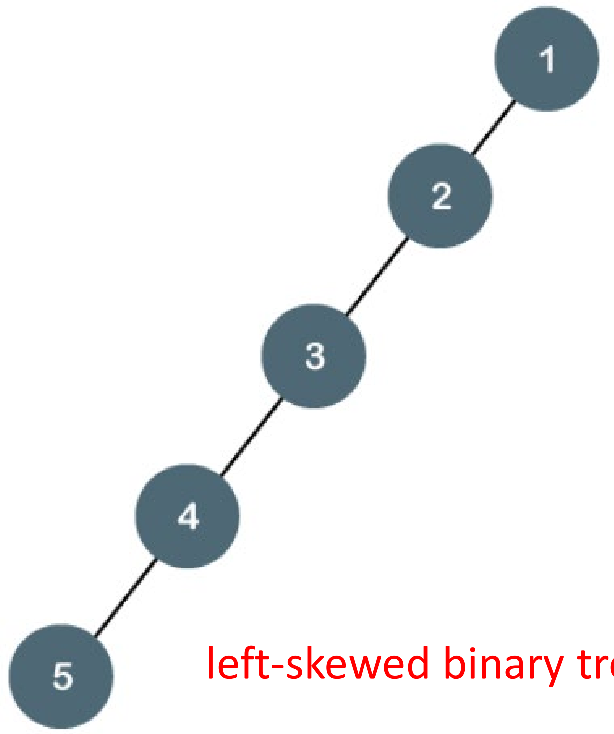
Height of a Degenerate Binary Tree is equal to Total number of nodes in that tree.



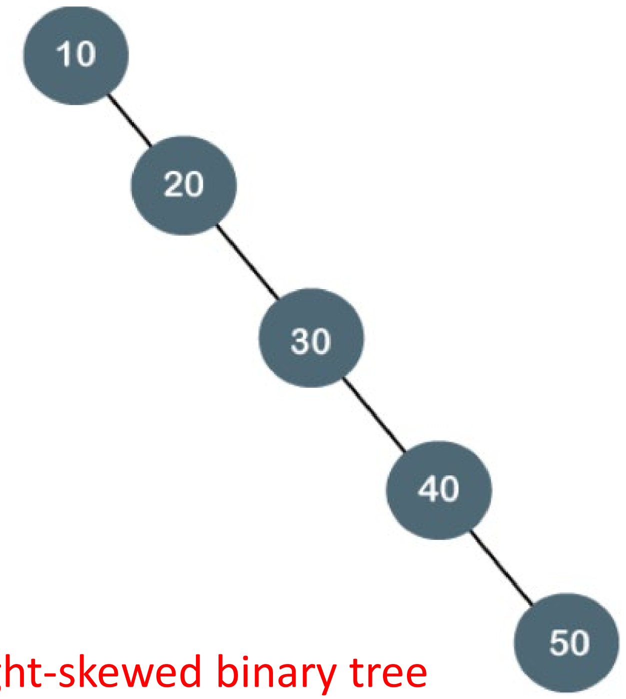
Types of Binary tree

Skewed binary tree

- In a skewed binary tree is a special type of Degenerate (or pathological) tree,
- Here, all the nodes are either left-skewed or right-skewed.
- A left-skewed binary tree is a tree in which each node has at most one right child.
- A right-skewed binary tree is a tree in which each node has at most one left child.



left-skewed binary tree



right-skewed binary tree

Binary Search Tree Traversal

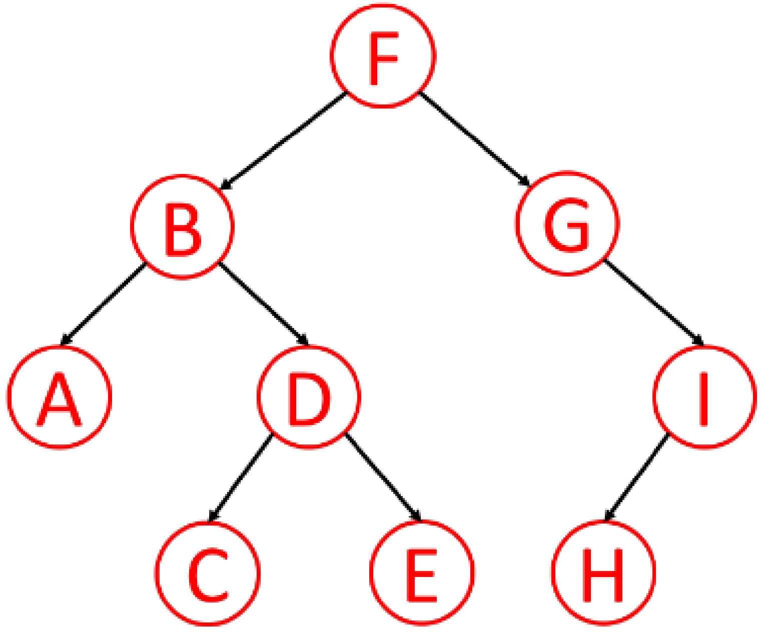
- ❑ Traverse the BST to visit all the nodes in the tree in a specific order. There are three main types of traversal: in-order, pre-order, and post-order.

- ❑ In-order traversal visits the nodes in ascending order of their keys
- ❑ pre-order traversal visits the current node before its children
- ❑ post-order traversal visits the current node after its children

- ❑ Traversal can be implemented using recursion or an iterative algorithm, and the time complexity is O(n), where n is the number of nodes in the tree.

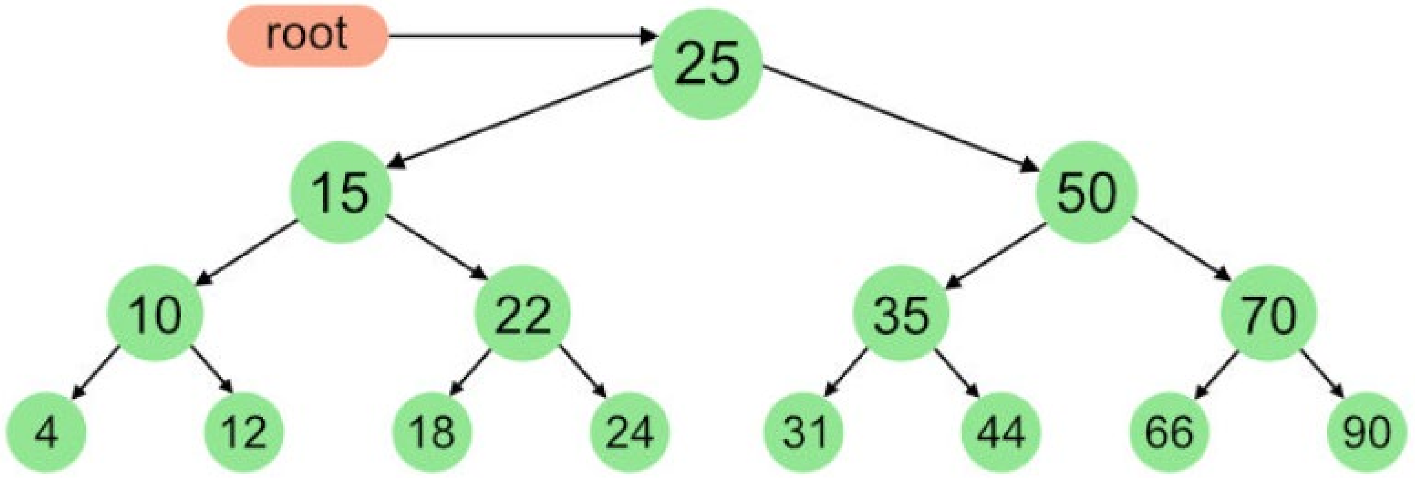
Binary Tree: Pre-order Traversal

- Pre-order traversal is to visit the root first. Then traverse the left subtree. Finally, traverse the right subtree.



Preorder:

F	B	A	D	C	E	G	I	H

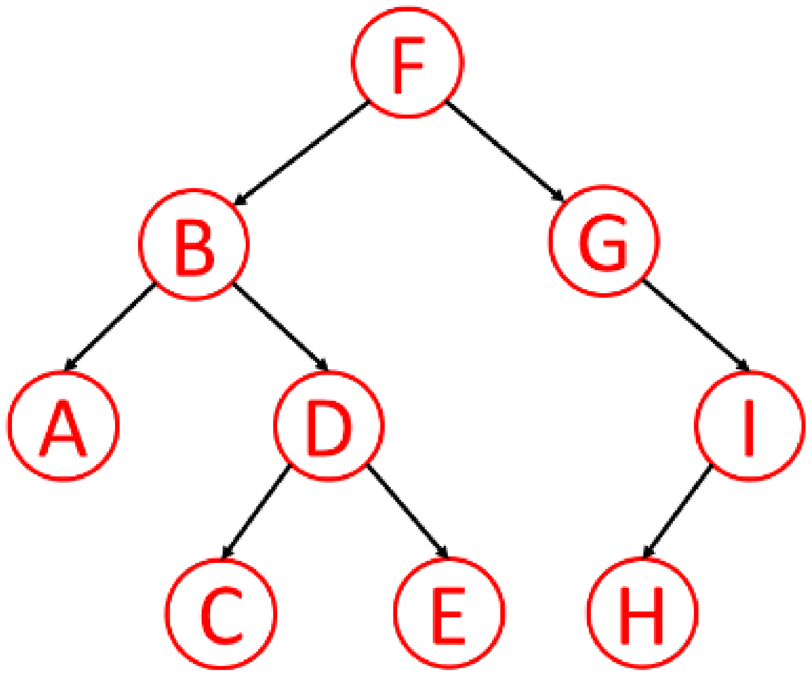


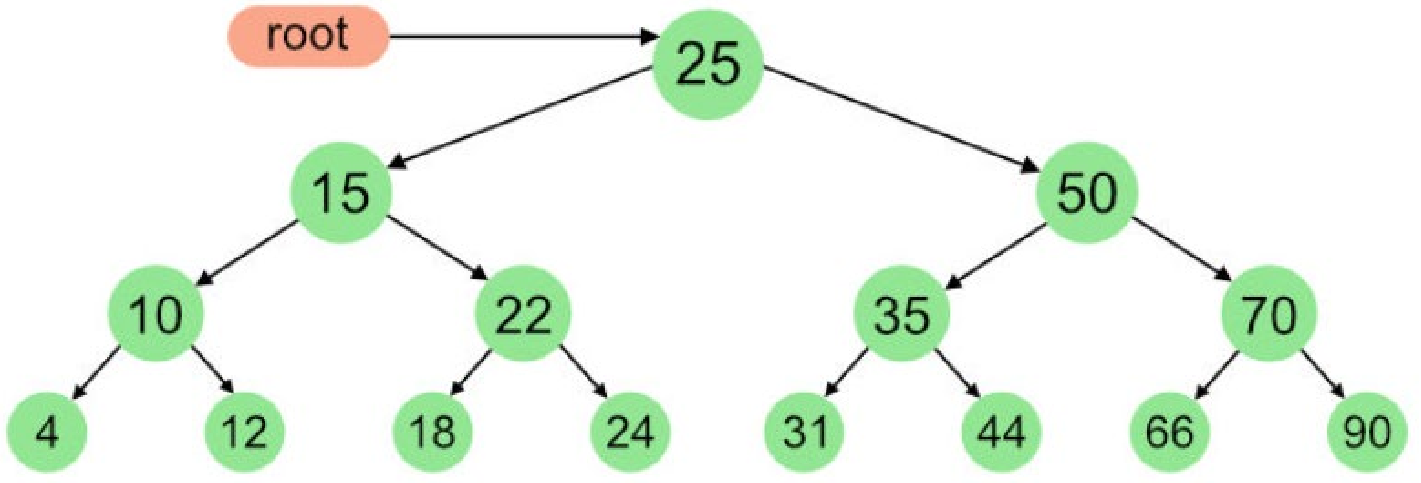
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

```python
def preorder(self):
    print(self.val)
    if self.left is not None:
        self.left.preorder()
    if self.right is not None:
        self.right.preorder()
```

Binary Tree: In-order Traversal

- In-order traversal is to traverse the left subtree first. Then visit the root. Finally, traverse the right subtree.





4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

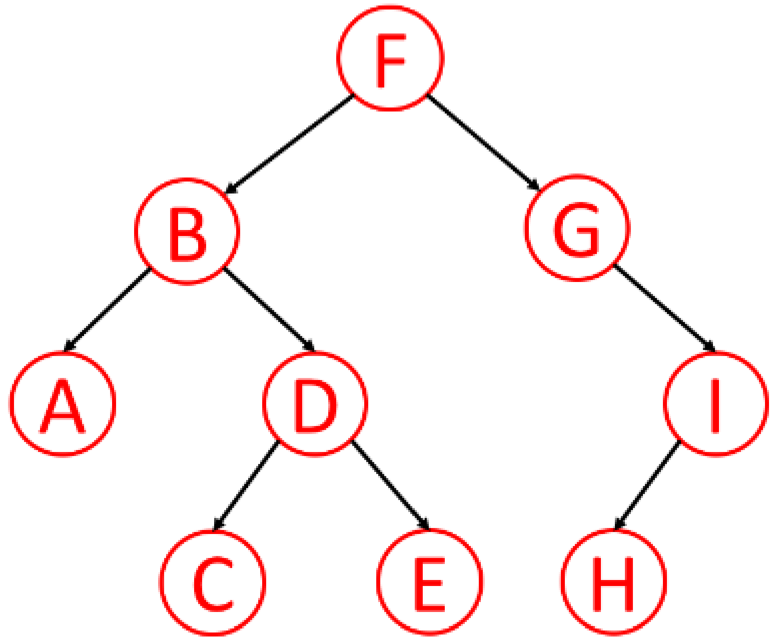
Inorder:

A	B	C	D	E	F	G	H	I

```
def inorder(self):
    if self.left is not None:
        self.left.inorder()
    print(self.val)
    if self.right is not None:
        self.right.inorder()
```

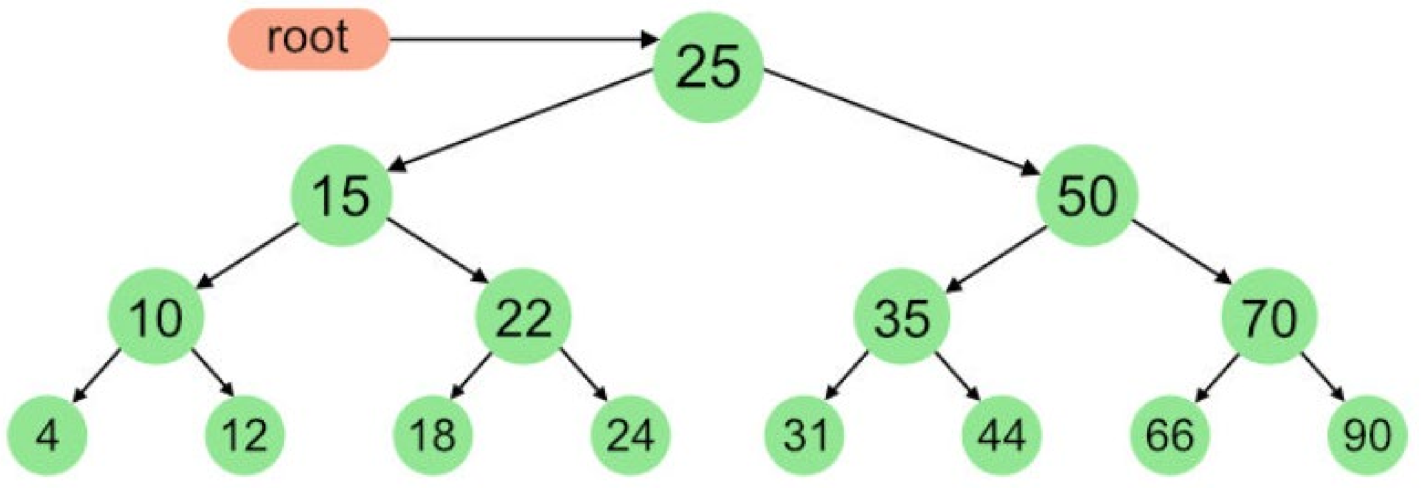
Binary Tree: Post-order Traversal

- Post-order traversal is to traverse the left subtree first. Then traverse the right subtree. Finally, visit the root.

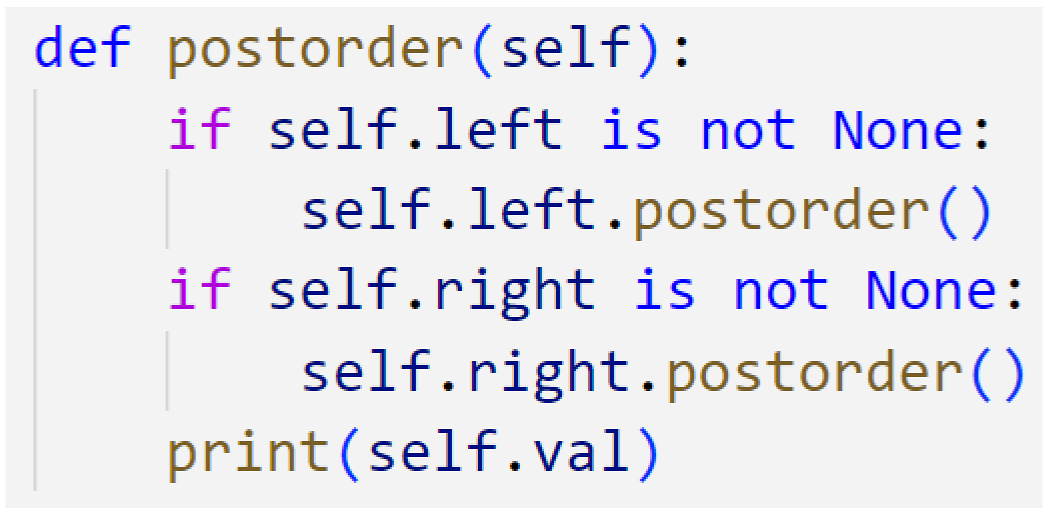


Postorder:

A	C	E	D	B	H	I	G	F

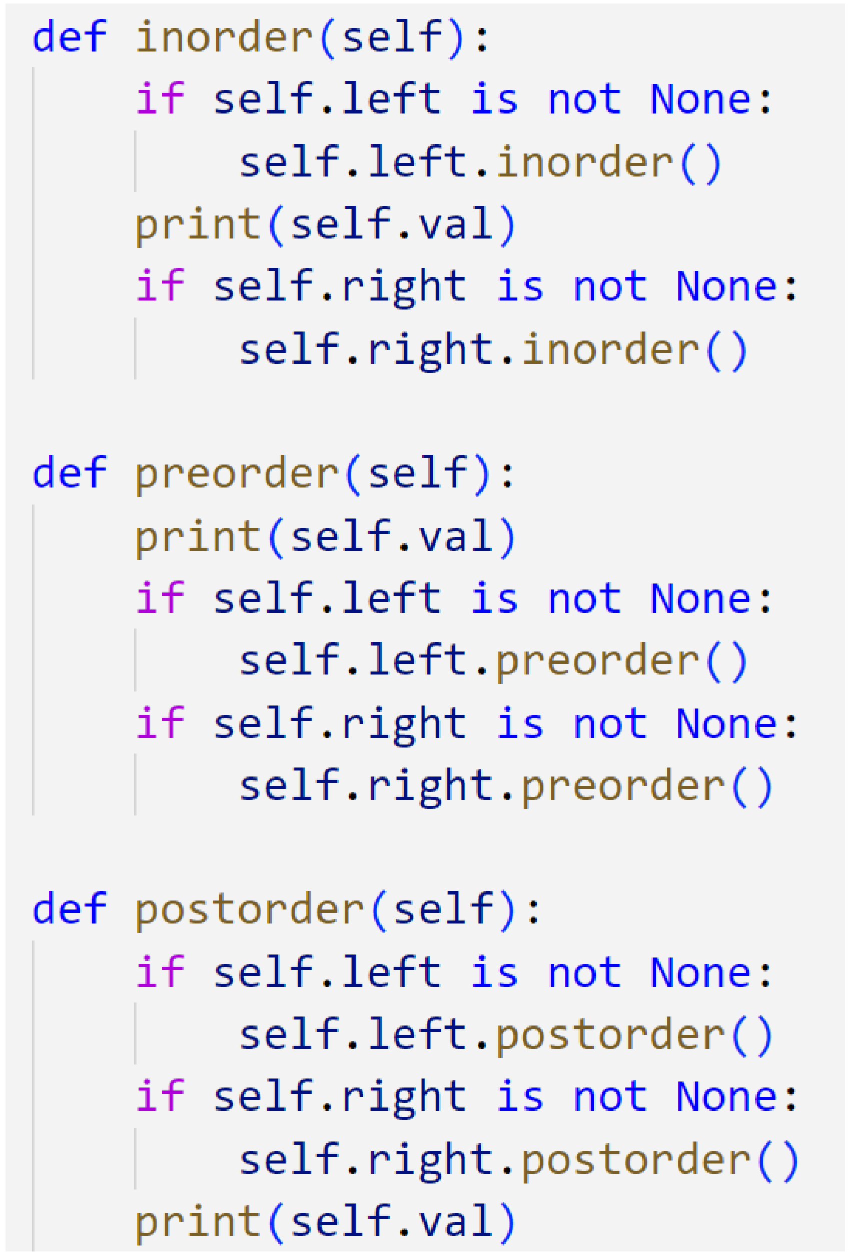


4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



```python
def postorder(self):
    if self.left is not None:
        self.left.postorder()
    if self.right is not None:
        self.right.postorder()
    print(self.val)
```

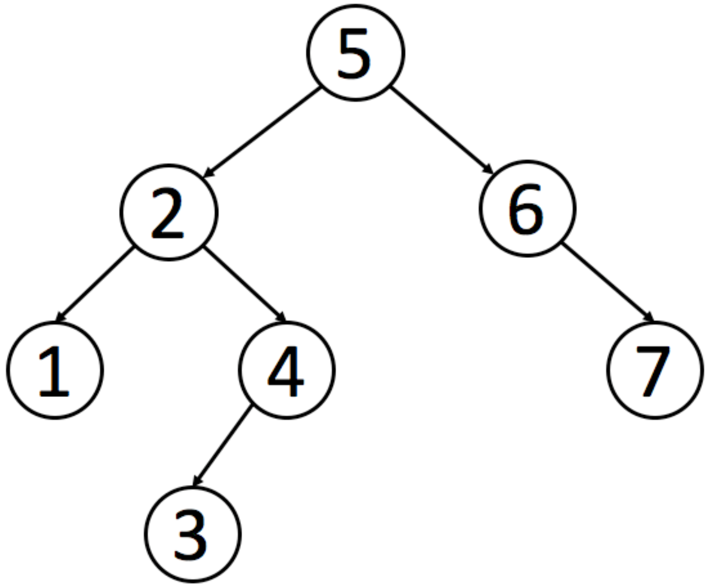
Traversal code



```
def inorder(self):
    if self.left is not None:
        self.left.inorder()
    print(self.val)
    if self.right is not None:
        self.right.inorder()

def preorder(self):
    print(self.val)
    if self.left is not None:
        self.left.preorder()
    if self.right is not None:
        self.right.preorder()

def postorder(self):
    if self.left is not None:
        self.left.postorder()
    if self.right is not None:
        self.right.postorder()
    print(self.val)
```

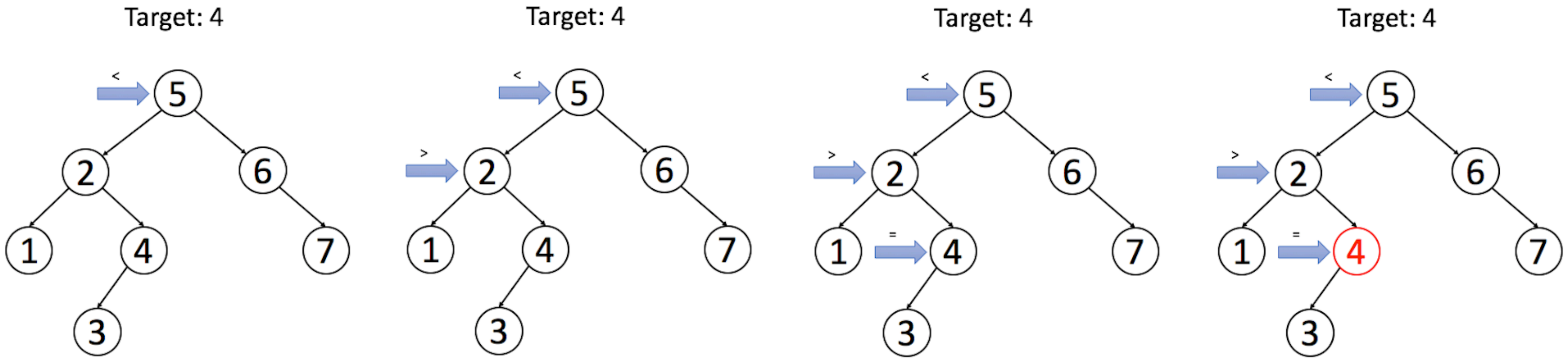
Binary Search Tree

- ❑ A Binary Search Tree is a special form of a binary tree.
- ❑ The value in each node must be greater than (or equal to) any values in its left subtree, but less than (or equal to) any values in its right subtree.

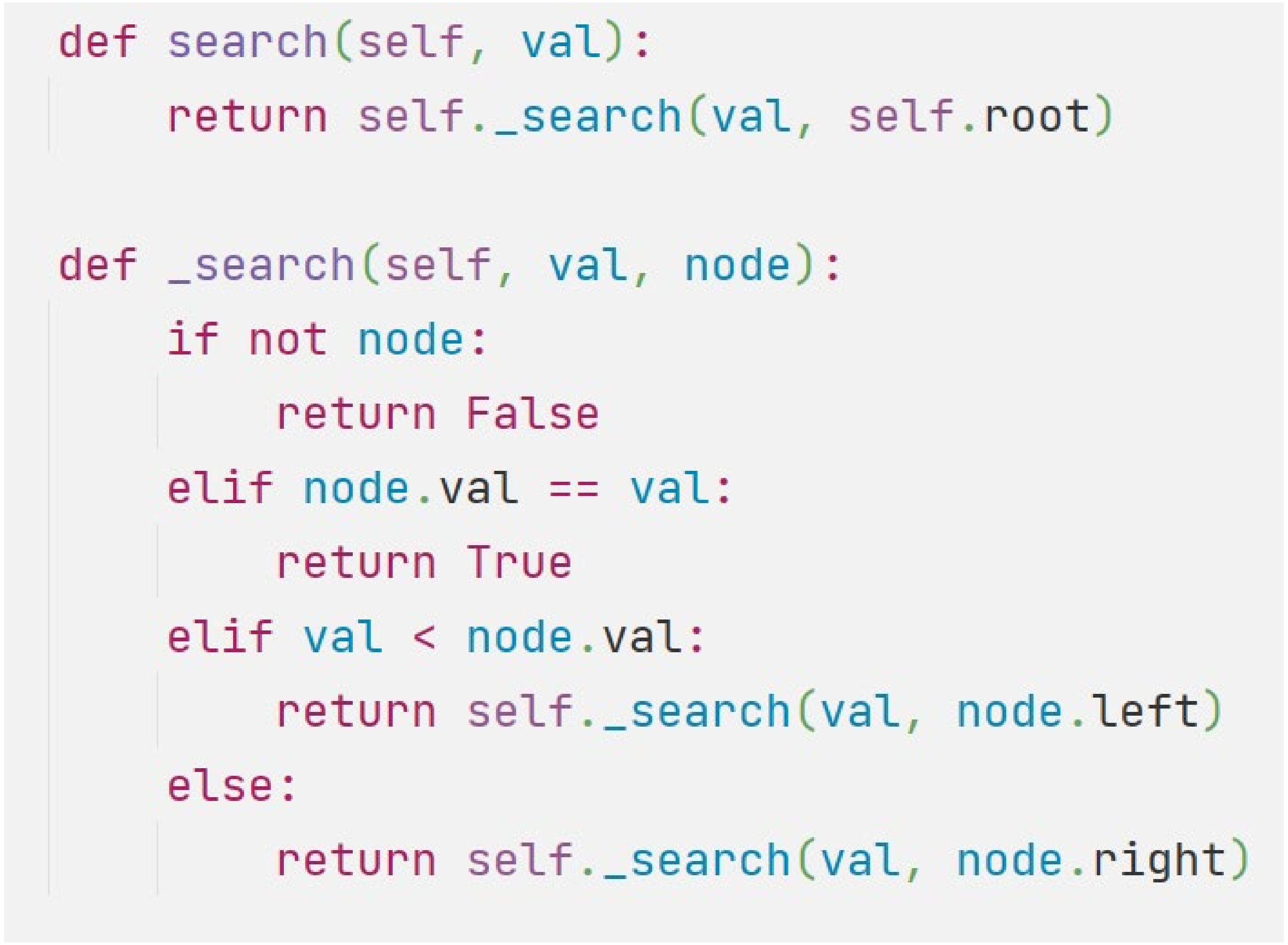


Binary Search Tree – Search Operation

- ❑ return the node if the target value is equal to the value of the node;
- ❑ continue searching in the *left subtree* if the target value is less than the value of the node;
- ❑ continue searching in the right subtree if the target value is larger than the value of the node.
- ❑ The time complexity of the search operation in a balanced BST is O(log n), where n is the number of nodes in the tree.



Binary Search Tree – Search Operation

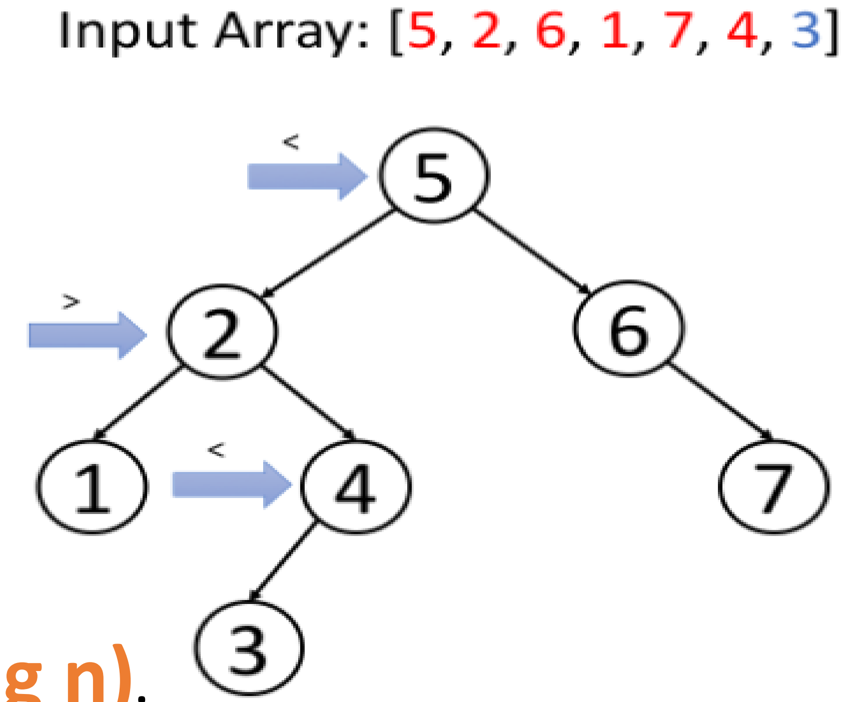


```
def search(self, val):
    return self._search(val, self.root)

def _search(self, val, node):
    if not node:
        return False
    elif node.val == val:
        return True
    elif val < node.val:
        return self._search(val, node.left)
    else:
        return self._search(val, node.right)
```

Binary Search Tree – Insertion Operation

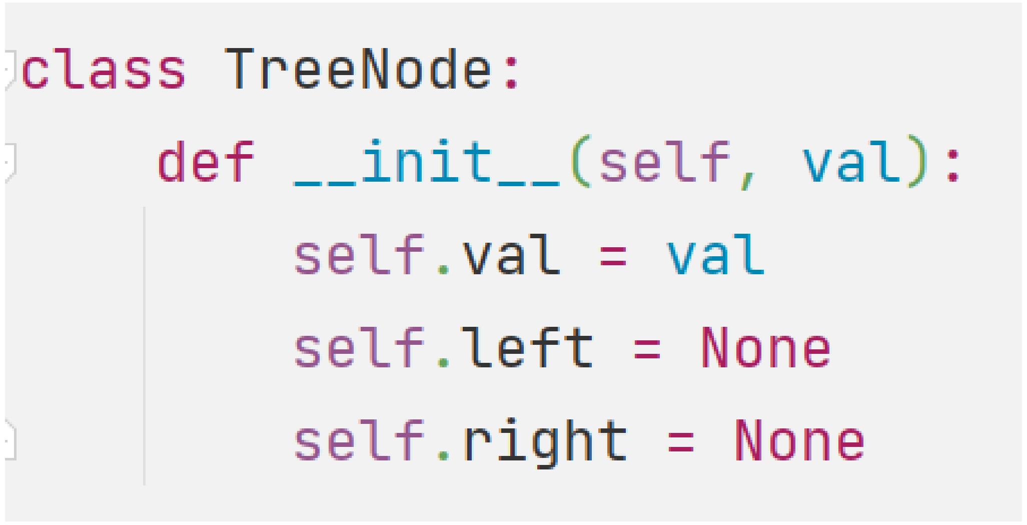
❑ Similar to the search strategy, for each node, we will:

1. search the left or right subtrees according to the relation of the value of the node and the value of our target node;
2. repeat STEP 1 until reaching an external node;
3. add the new node as its left or right child depending on the relation of the value of the node and the value of our target node.

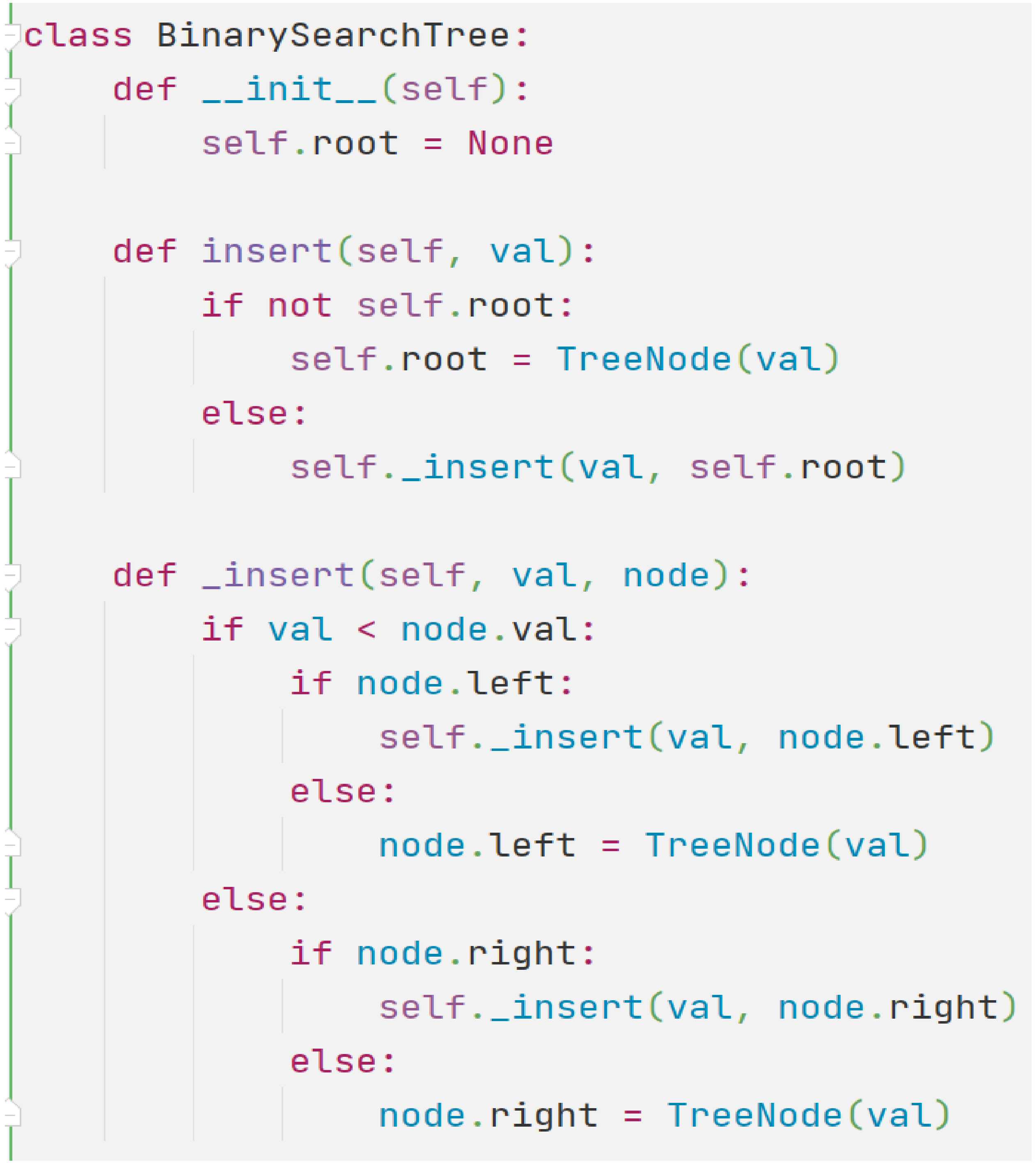


Time complexity **of the insertion operation in a balanced BST is also** **O(log n)**.

Binary Search Tree – Insertion Operation



```python
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```



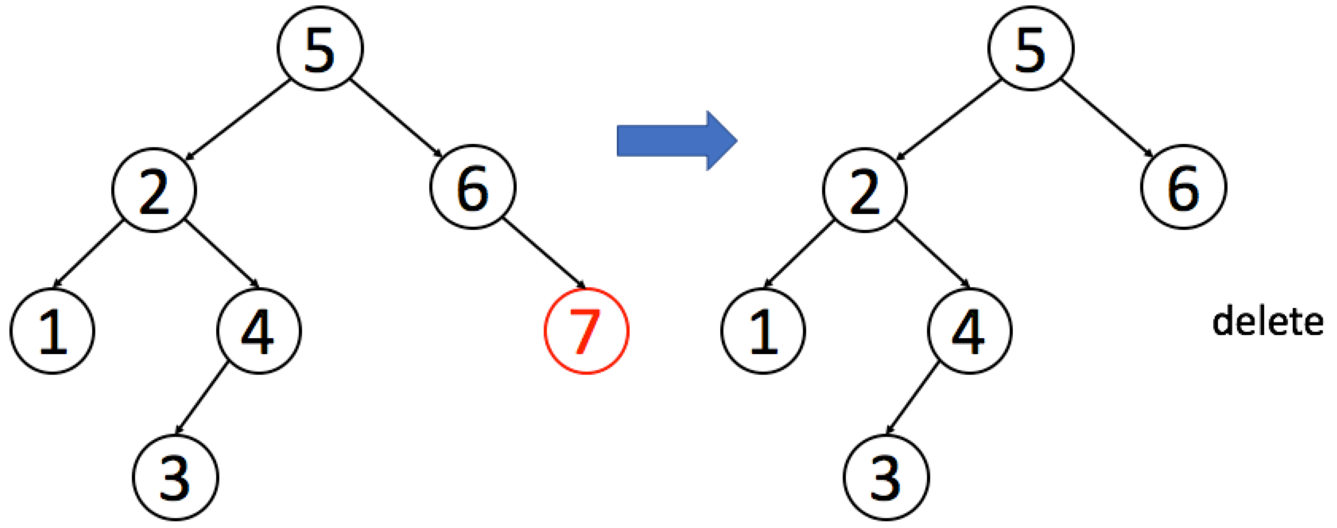
```python
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, val):
        if not self.root:
            self.root = TreeNode(val)
        else:
            self._insert(val, self.root)

    def _insert(self, val, node):
        if val < node.val:
            if node.left:
                self._insert(val, node.left)
            else:
                node.left = TreeNode(val)
        else:
            if node.right:
                self._insert(val, node.right)
            else:
                node.right = TreeNode(val)
```

Binary Search Tree – Deletion Operation

- ❑ Deletion is more complicated than the two operations we mentioned before.
- ❑ There are also many different strategies for deletion.

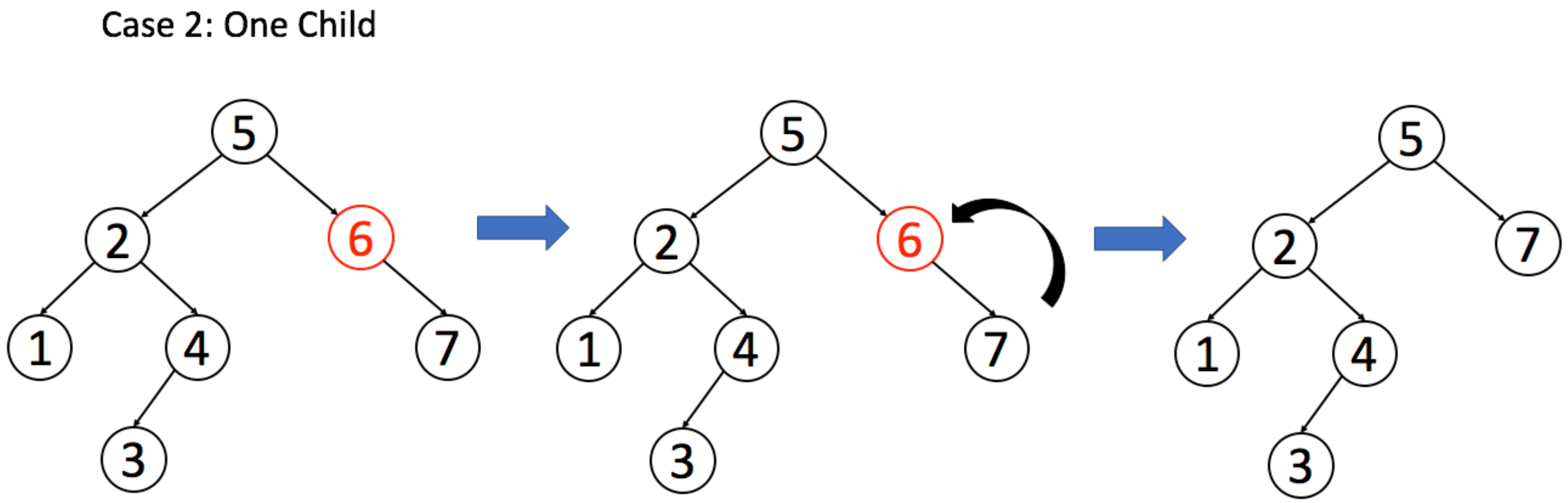
- If the target node has ***no child***, we can simply remove the node

Case 1: No Child



Binary Search Tree – Deletion Operation

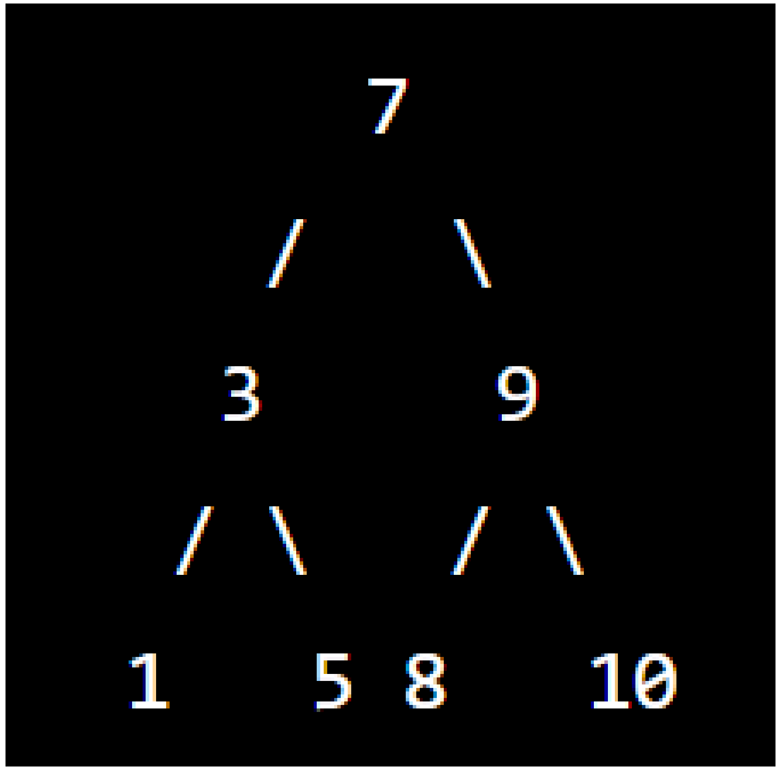
- If the target node has ***no child***, we can simply remove the node
- If the target node has ***one child***, we can use its child to replace itself



Binary Search Tree – Deletion Operation

- If the target node has ***two children***, replace the node with its in-order successor or predecessor node and delete that node.

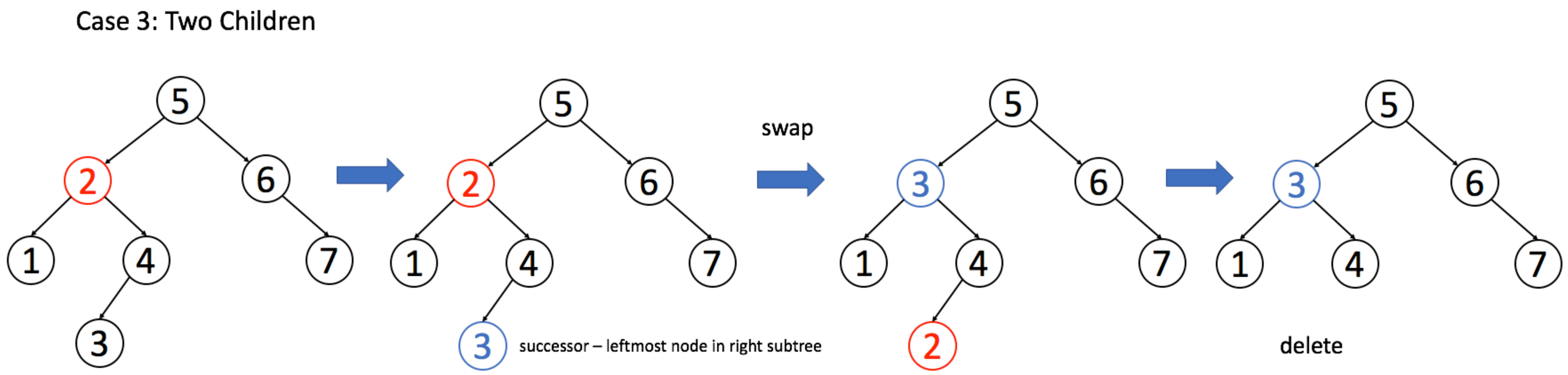
Inorder: 1, 3, 5, 7, 8, 9, 10



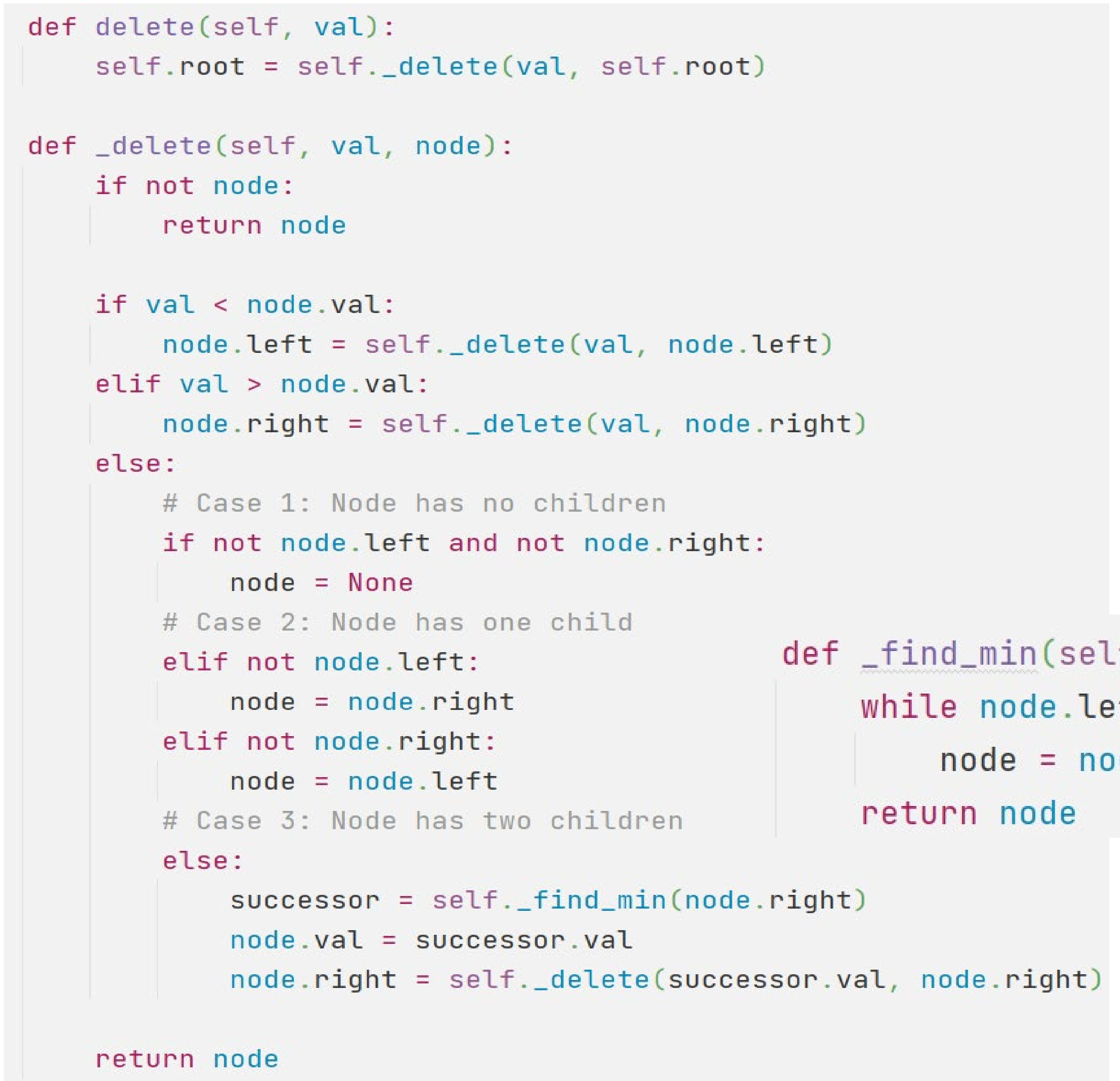
```
       7
     /   \
    3     9
   / \   / \
  1   5 8   10
```

Binary Search Tree – Deletion Operation

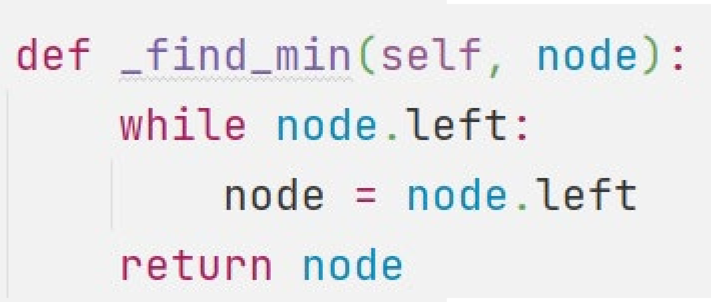
- If the target node has ***two children***, replace the node with its in-order successor or predecessor node and delete that node.



❑ Delete Operation



```
def delete(self, val):
    self.root = self._delete(val, self.root)

def _delete(self, val, node):
    if not node:
        return node

    if val < node.val:
        node.left = self._delete(val, node.left)
    elif val > node.val:
        node.right = self._delete(val, node.right)
    else:
        # Case 1: Node has no children
        if not node.left and not node.right:
            node = None
        # Case 2: Node has one child
        elif not node.left:
            node = node.right
        elif not node.right:
            node = node.left
        # Case 3: Node has two children
        else:
            successor = self._find_min(node.right)
            node.val = successor.val
            node.right = self._delete(successor.val, node.right)

    return node
```



```
def _find_min(self, node):
    while node.left:
        node = node.left
    return node
```

AVL Tree

- ❑ An AVL tree is a type of self-balancing binary search tree named after its inventors Adelson-Velsky and Landis.

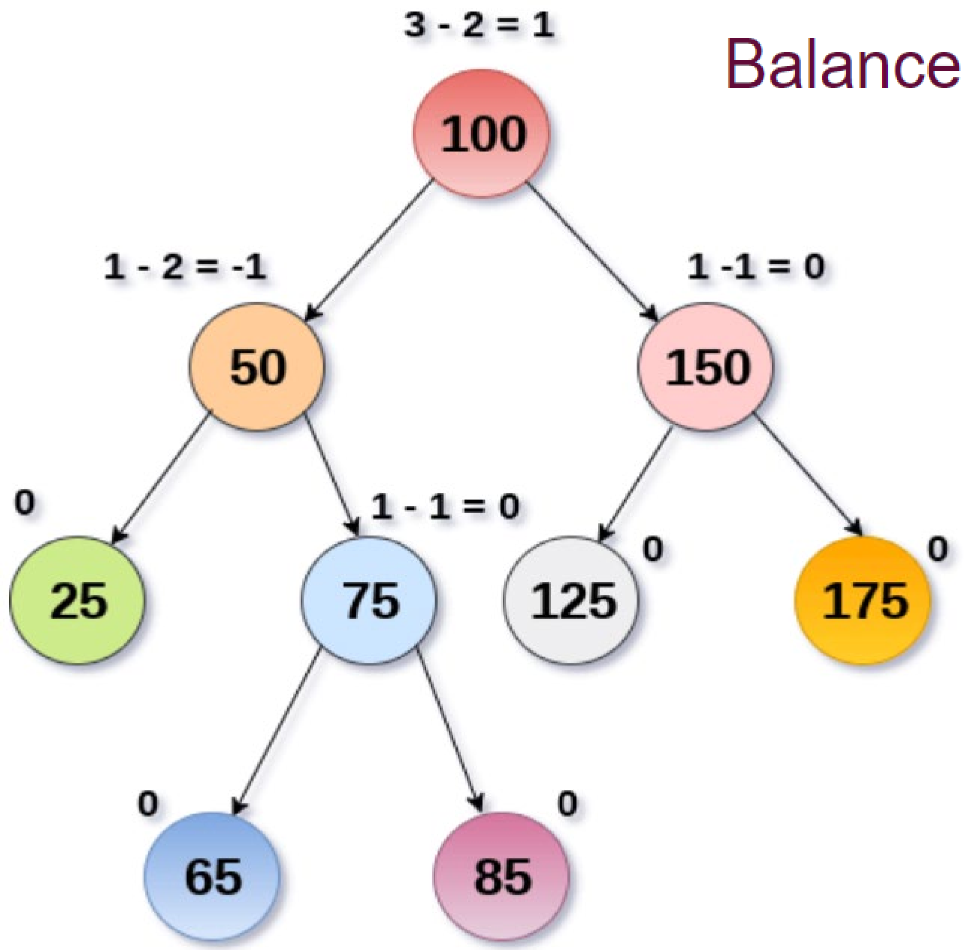
- ❑ It is designed to maintain a balanced tree structure by automatically reorganizing itself as nodes are added or removed from the tree.
  - ❑ The AVL tree is balanced by ensuring that the height difference between the left and right subtrees of any node in the tree is no more than one.

  - ❑ This ensures that the worst-case time complexity for common operations such as searching, insertion, and deletion is O(log n).

- ❑ It is used in many applications, including database indexing, compiler design, and computer graphics.

AVL Tree

The AVL tree is balanced by ensuring that the height difference between the left and right subtrees of any node in the tree is no more than one.

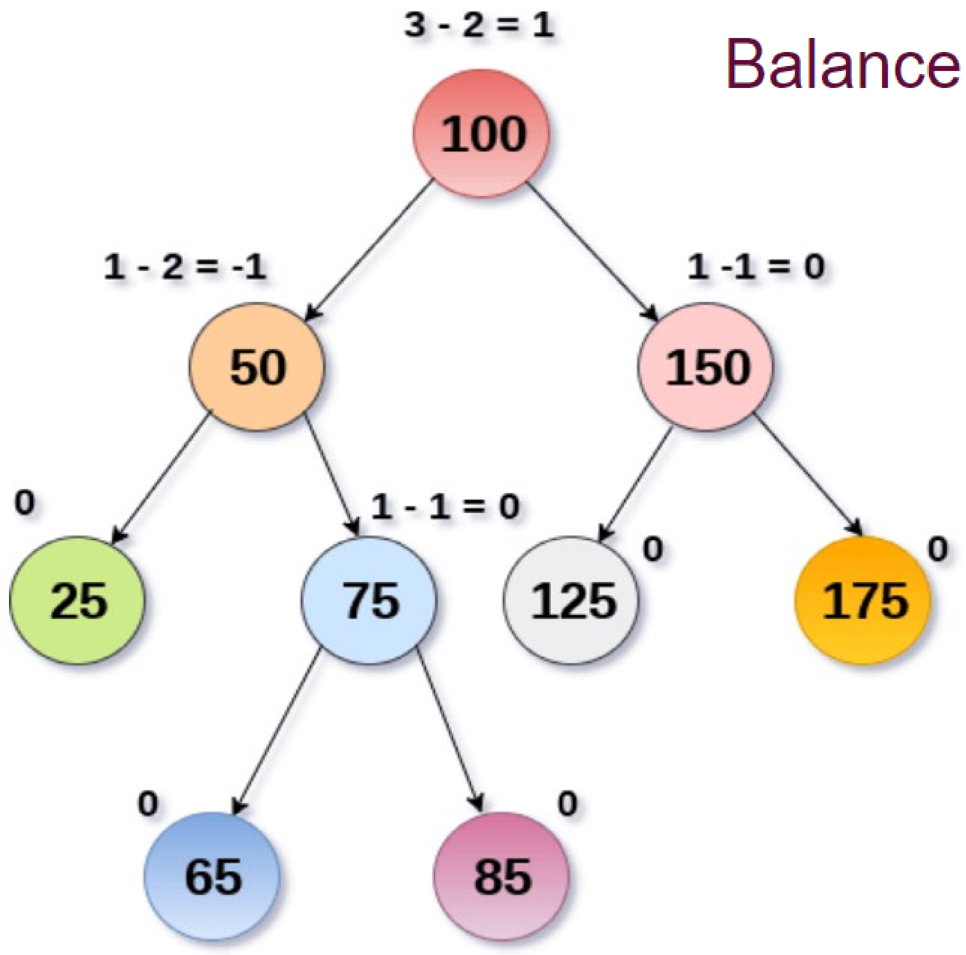


Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1:
  - Then the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0
  - Then the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1,
  - it means that the left sub-tree is one level lower than the right sub-tree.

AVL Tree (cont.)

The AVL tree is balanced by ensuring that the height difference between the left and right subtrees of any node in the tree is no more than one.



Balance Factor (k) = height (left(k)) - height (right(k))

- AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree.
- Searching and traversing do not lead to the violation in property of AVL tree.
- Insertion and deletion are the operations which can violate this property and therefore, they need to be revisited

Why AVL Tree?

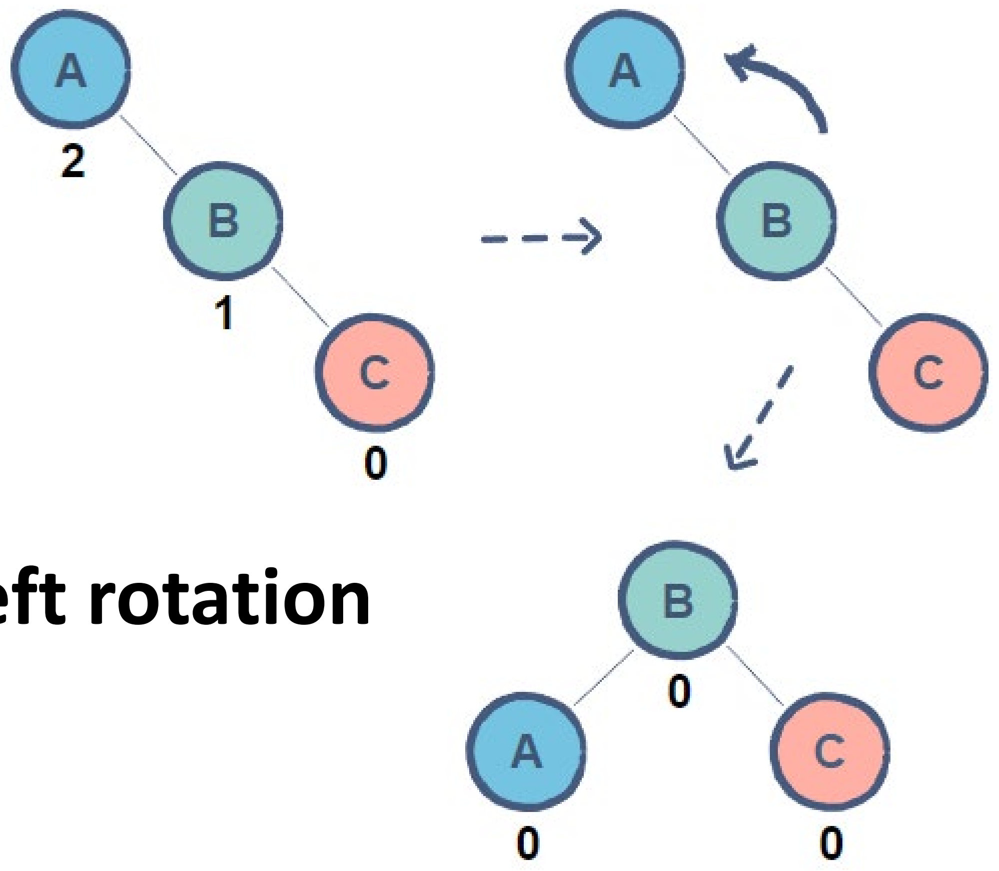
- ❑AVL tree controls the height of the binary search tree by not letting it to be skewed.
- ❑The time taken for all operations in a binary search tree of height h is **O(h)**.
- ❑It can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n
- ❑AVL tree imposes an upper bound on each operation to be

  O(log n) where n is the number of nodes.

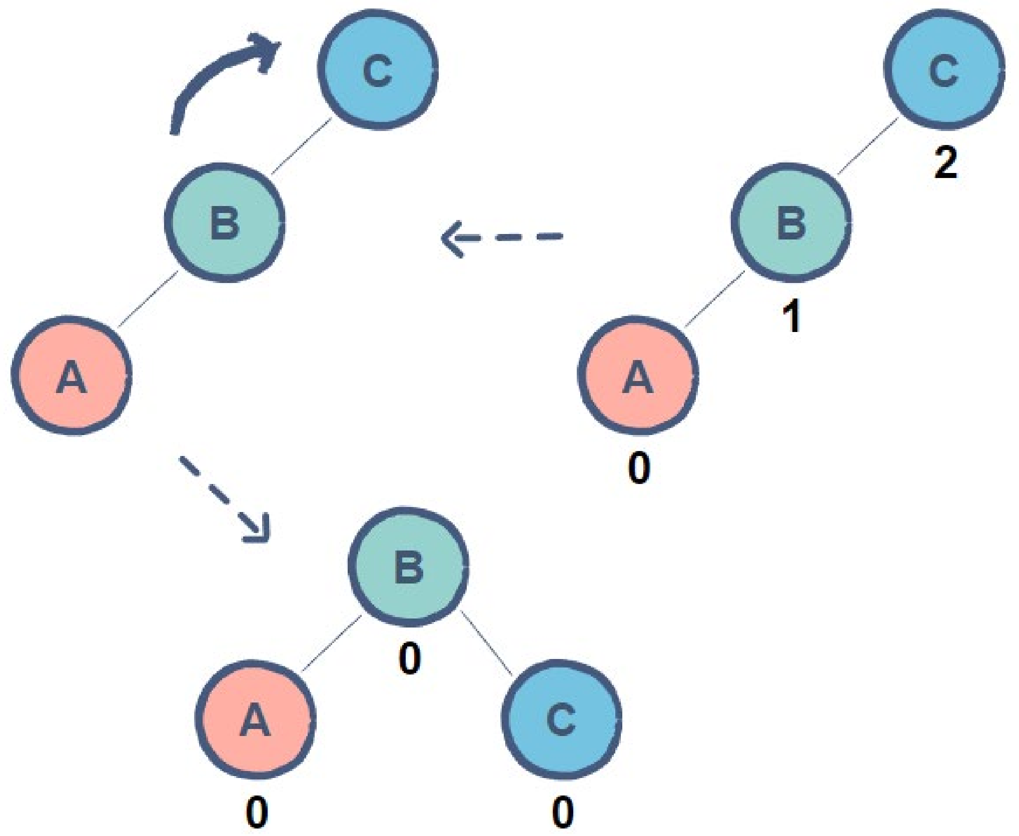
AVL Rotations

- ❑Left rotation
- ❑Right rotation
- ❑Left-Right rotation
- ❑Right-Left rotation

AVL Rotations



Left rotation

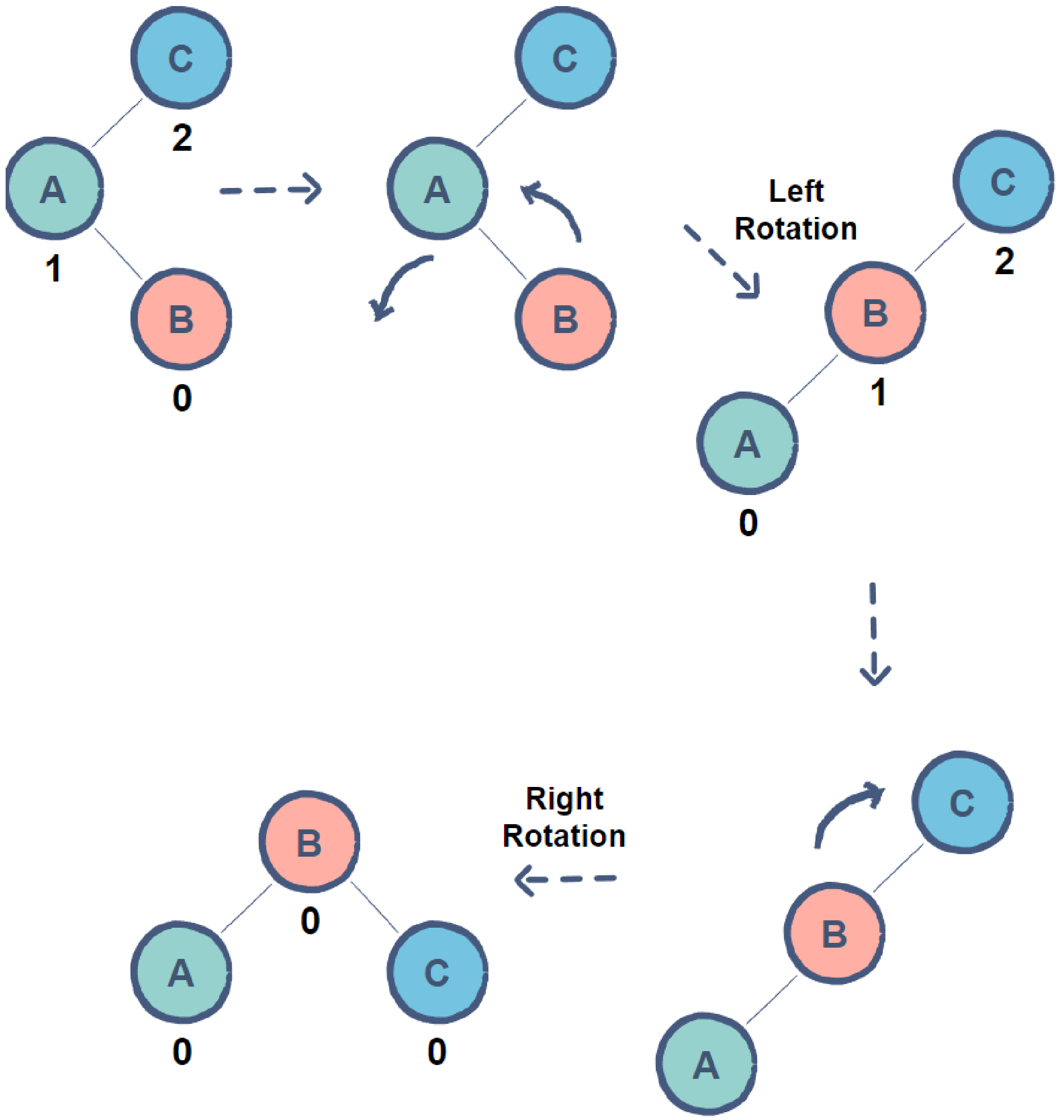


Right rotation

- A single rotation applied when a node is inserted in the right subtree of a right subtree.
- **node A** has a balance factor of 2 after the insertion of **node C**.
- By rotating the tree left, **node B** becomes the root resulting in a balanced tree.

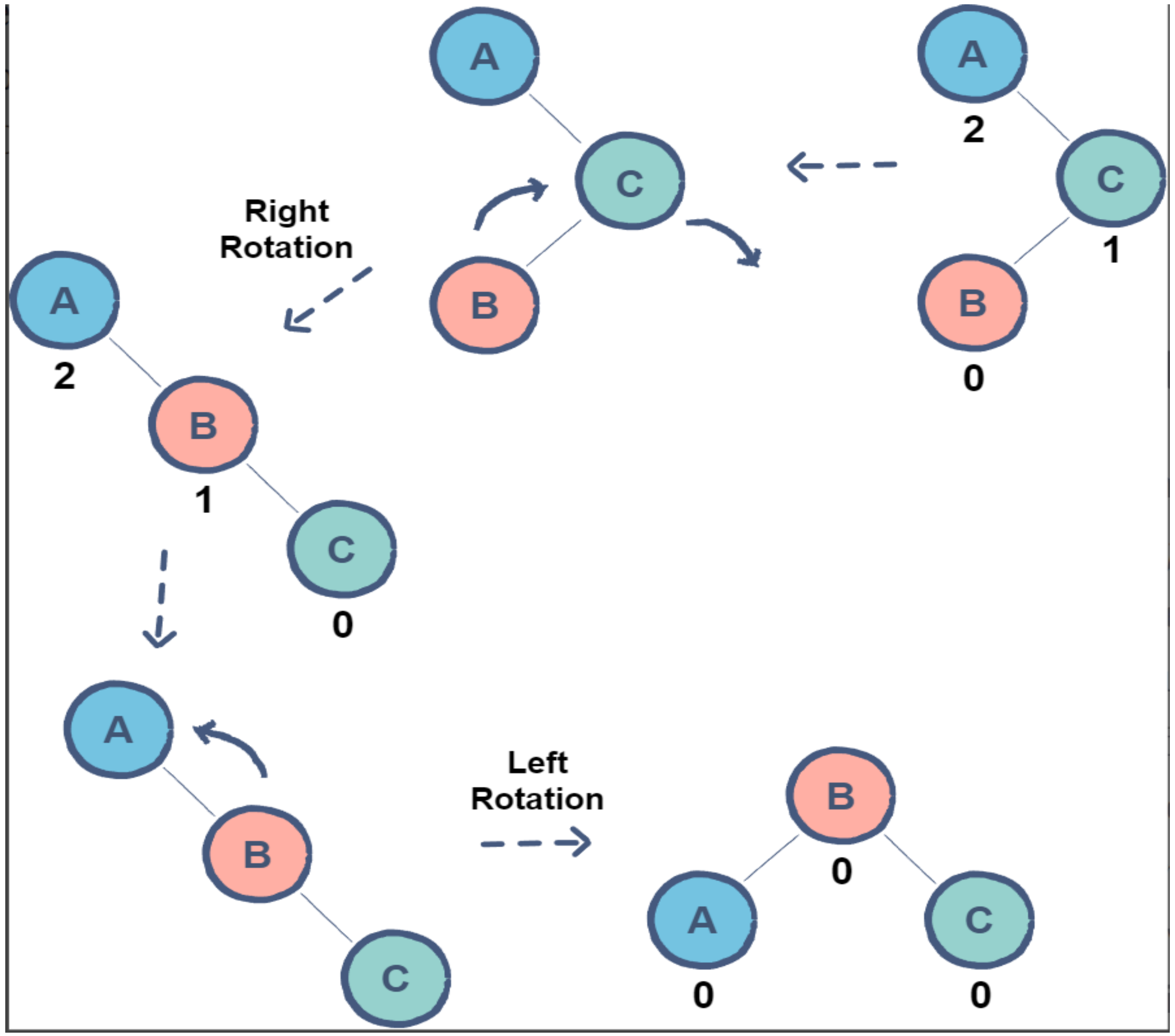
Left-Right Rotation

- A double rotation in which a *left rotation* is followed by a *right rotation*.
- In the given example, **node B** is causing an imbalance resulting in **node C** to have a balance factor of 2.
- As **node B** is inserted in the right subtree of **node A**, a *left rotation* needs to be applied.
- However, a single rotation will not give us the required results. Now, all we have to do is apply the *right rotation* as shown before to achieve a balanced tree.



Right-Left Rotation

- A double rotation in which a *right rotation* is followed by a *left rotation*.
- In the given example, **node B** is causing an imbalance resulting in **node A** to have a balance factor of 2.
- As **node B** is inserted in the left subtree of **node C**, a *right rotation* needs to be applied.
- a single rotation will not give us the required results.
- Now, by applying the *left rotation* as shown before, we can achieve a balanced tree.



Left-Left Imbalance – Solution Right Rotation

- This occurs when the balance factor of a node is +2
- and its left child's balance factor is +1.
- To fix this imbalance, perform a right rotation on the unbalanced node.
- This moves the left child up to become the new root, and the unbalanced node becomes the right child of the new root.

Right-Right Imbalance – Solution Left Rotation

- This occurs when the balance factor of a node is -2
- and its right child's balance factor is -1.
- To fix this imbalance, perform a left rotation on the unbalanced node.
- This moves the right child up to become the new root, and the unbalanced node becomes the left child of the new root.

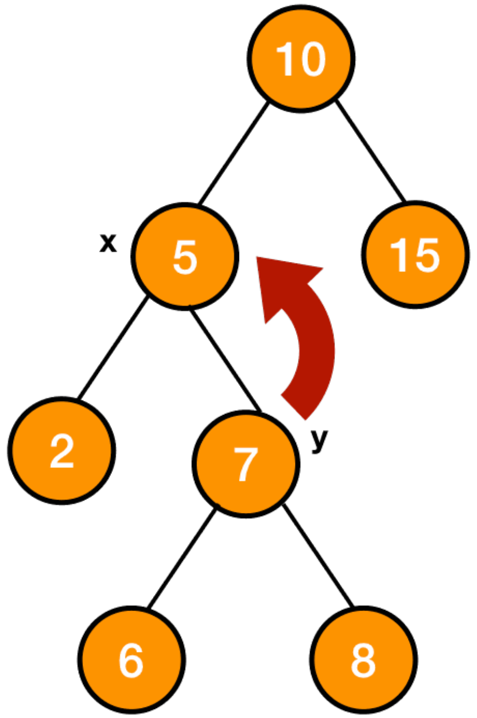
Left-Right Imbalance – Solution Left-Right Double Rotation

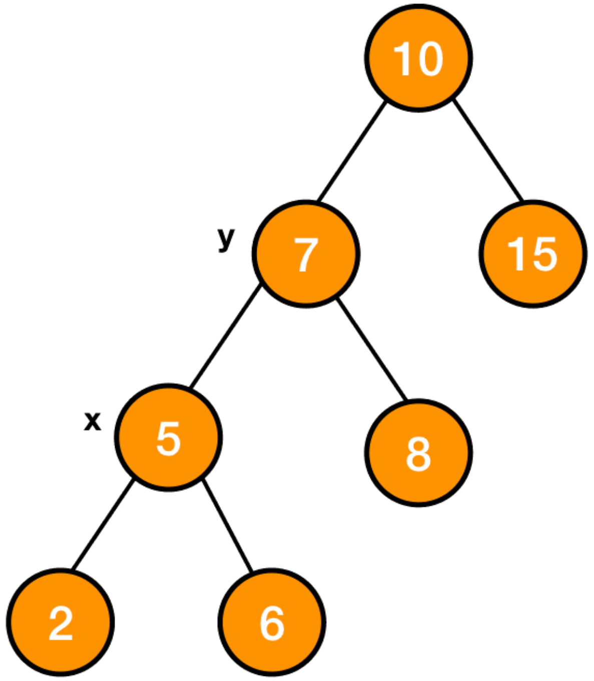
- This occurs when the balance factor of a node is +2
- and its left child's balance factor is -1.
- To fix this imbalance, perform a left-right double rotation.
- This involves performing a left rotation on the left child, followed by a right rotation on the unbalanced node.

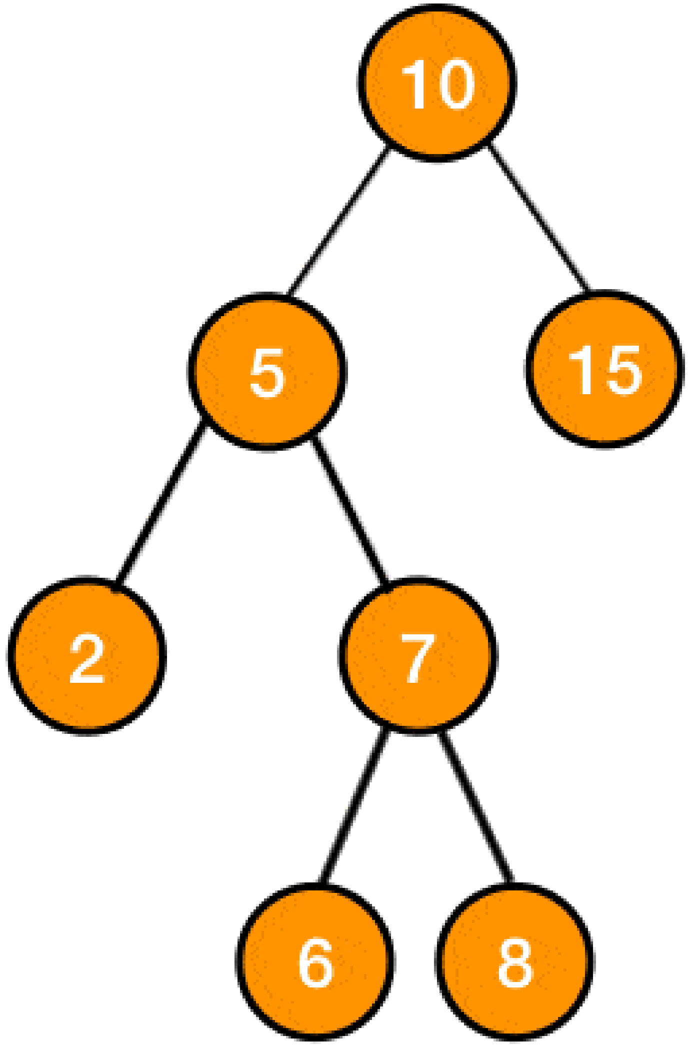
Right-Left Imbalance – Solution Left Rotation

- occurs when the balance factor of a node is -2
- and its right child's balance factor is +1.
- To fix this imbalance, perform a right-left double rotation.
- This involves performing a right rotation on the right child, followed by a left rotation on the unbalanced node.

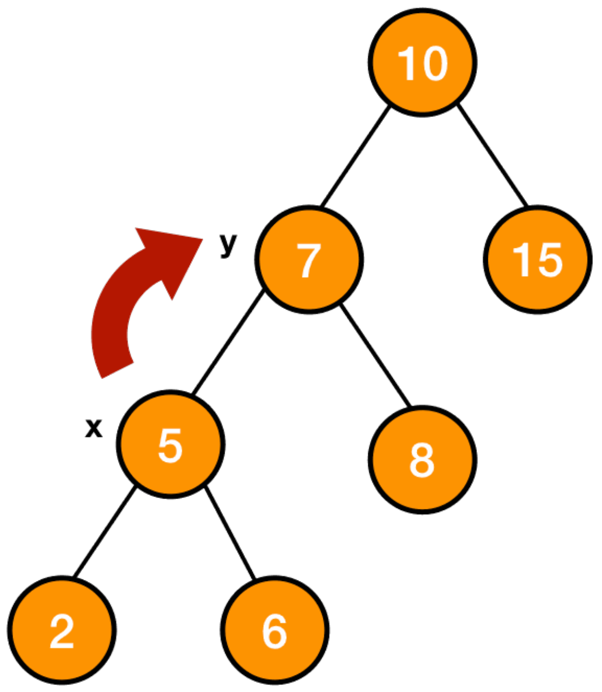
Left Rotation

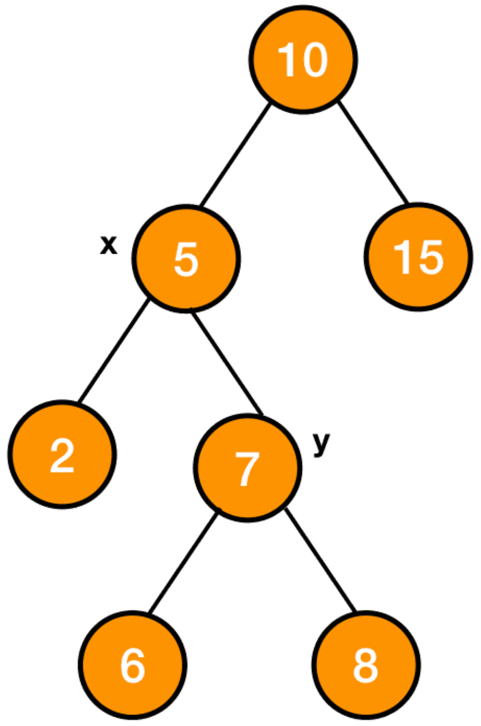


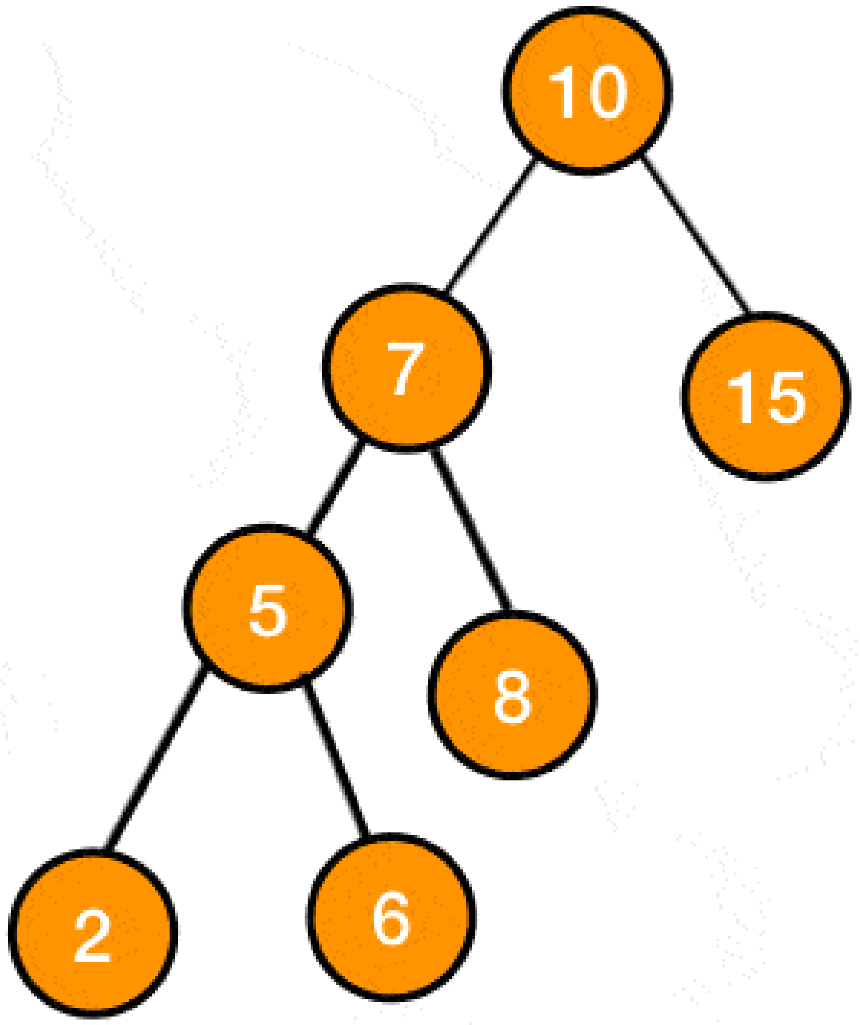




Right Rotation





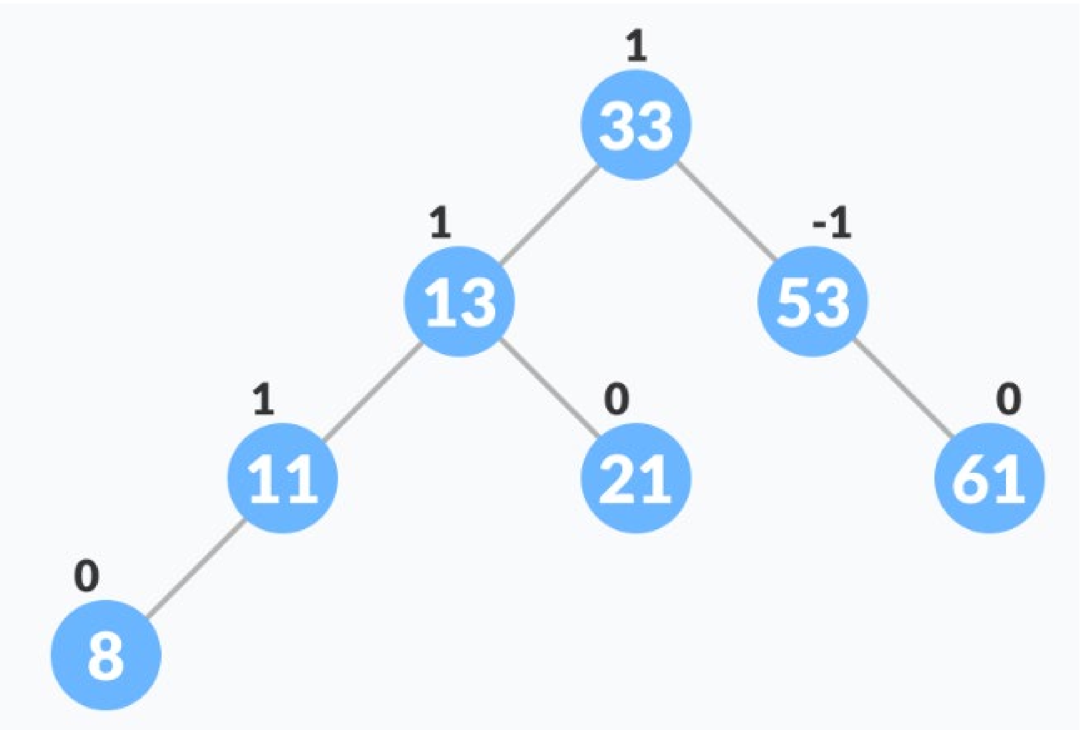


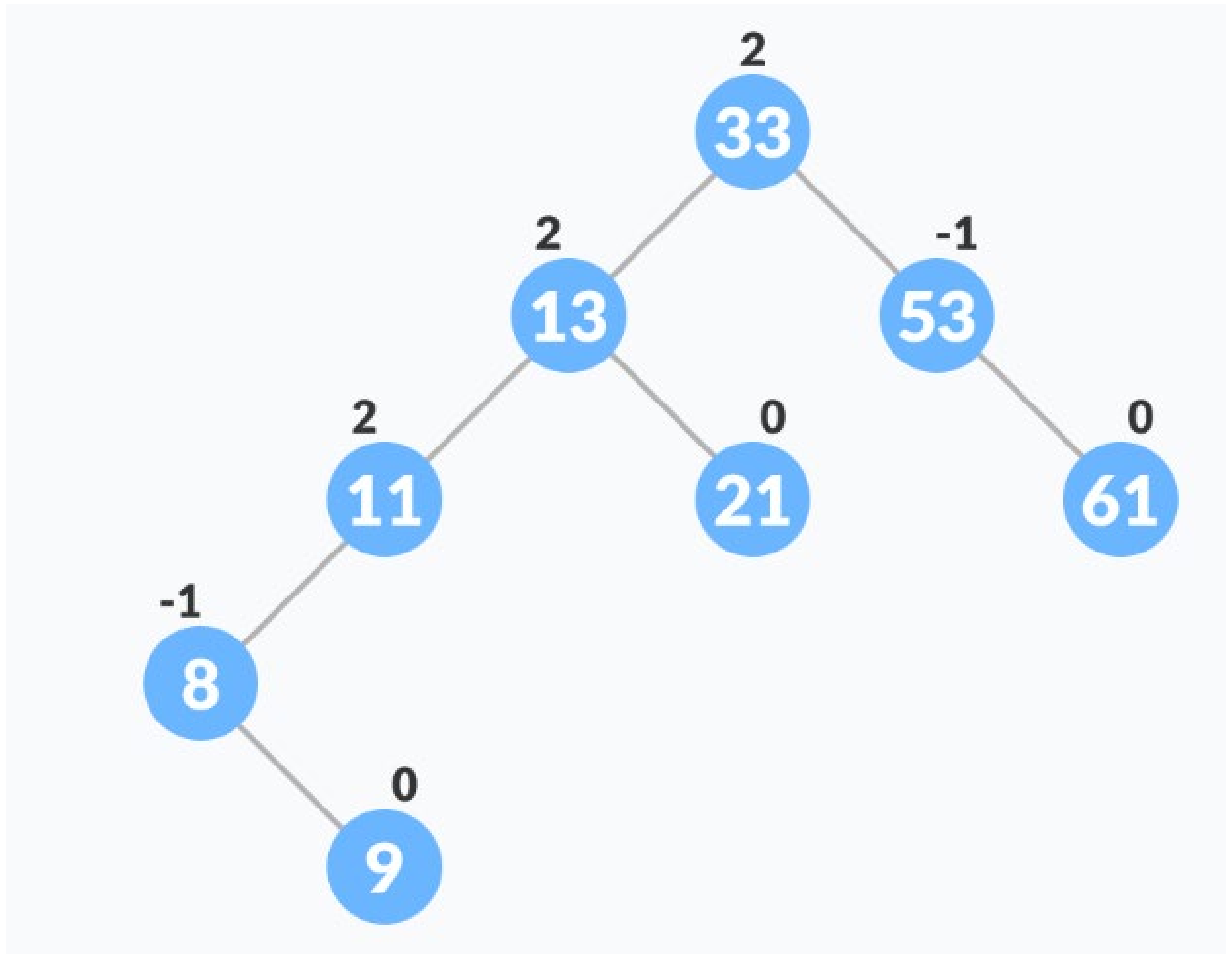
Insert on AVL Tree

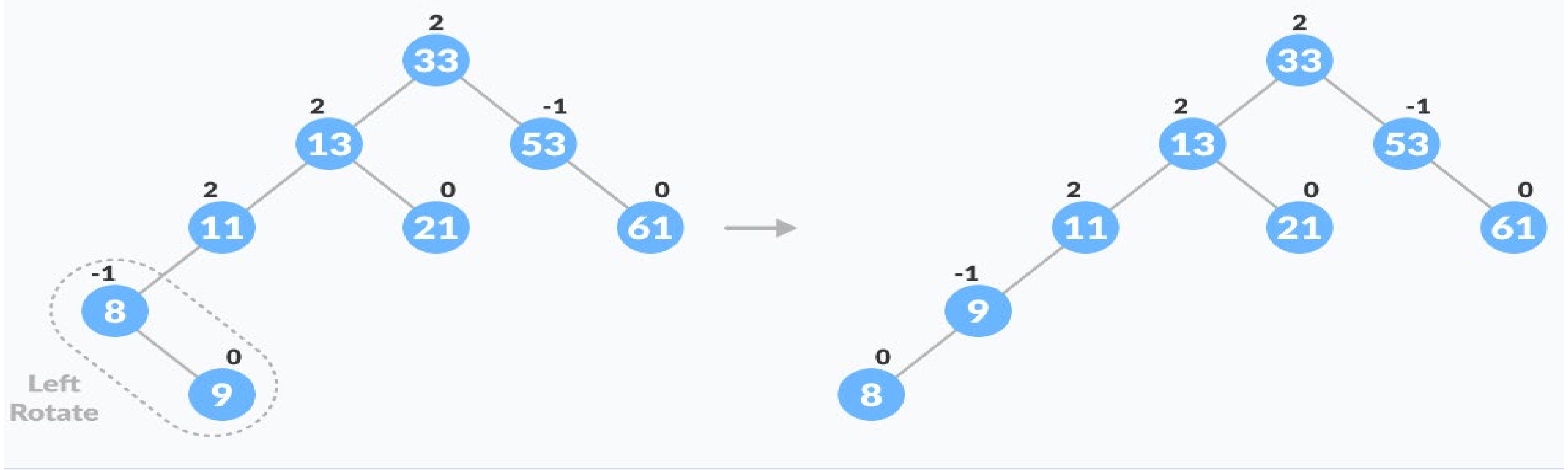
- Insert the new node in the tree as a leaf node.
- Starting from the new node, traverse up the tree to the root node, updating the balance factor of each node along the way.
- Depending on the type of imbalance (left-left, left-right, right-right, or right-left), perform a rotation operation to balance the subtree rooted at the unbalanced node.
- After the rotation operation is performed, update the balance factors of the affected nodes.
- Continue the traversal up the tree until the root node is reached, updating the balance factors of each node along the way.
- If the root node's balance factor becomes greater than 1 or less than -1, then perform a rotation operation to balance the entire tree.
- The AVL tree now has the new node inserted, and is still balanced.

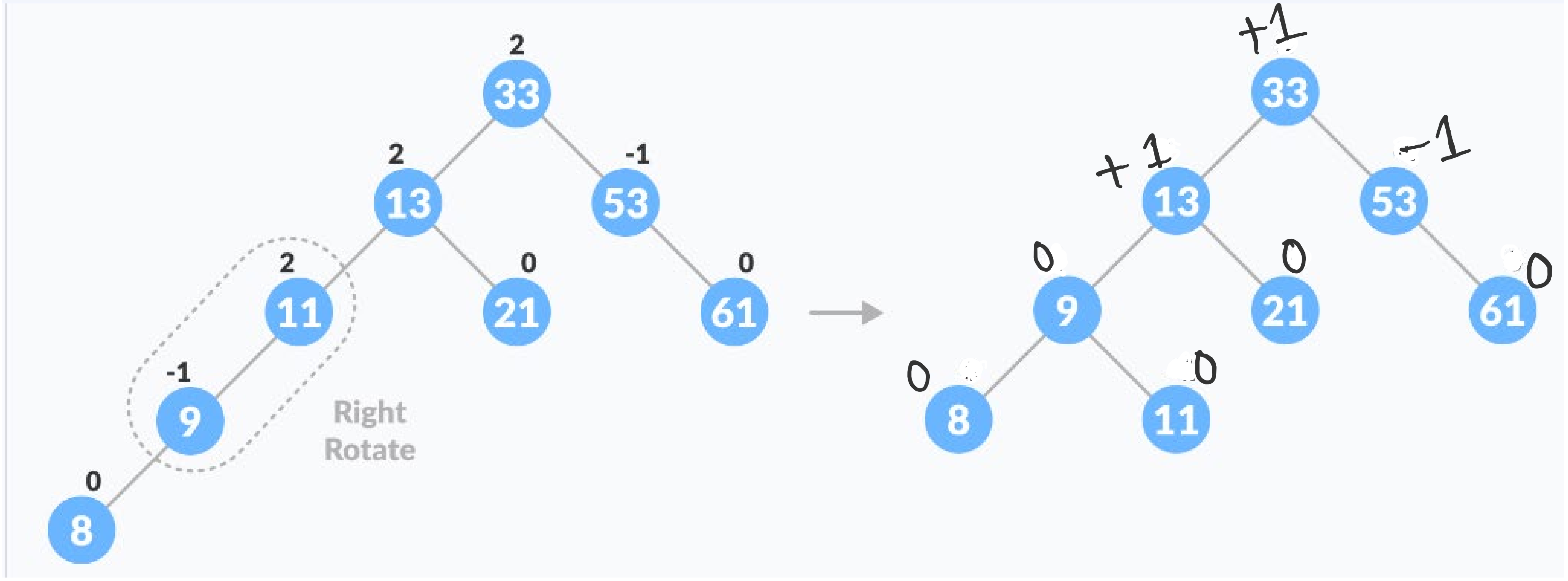
Insert

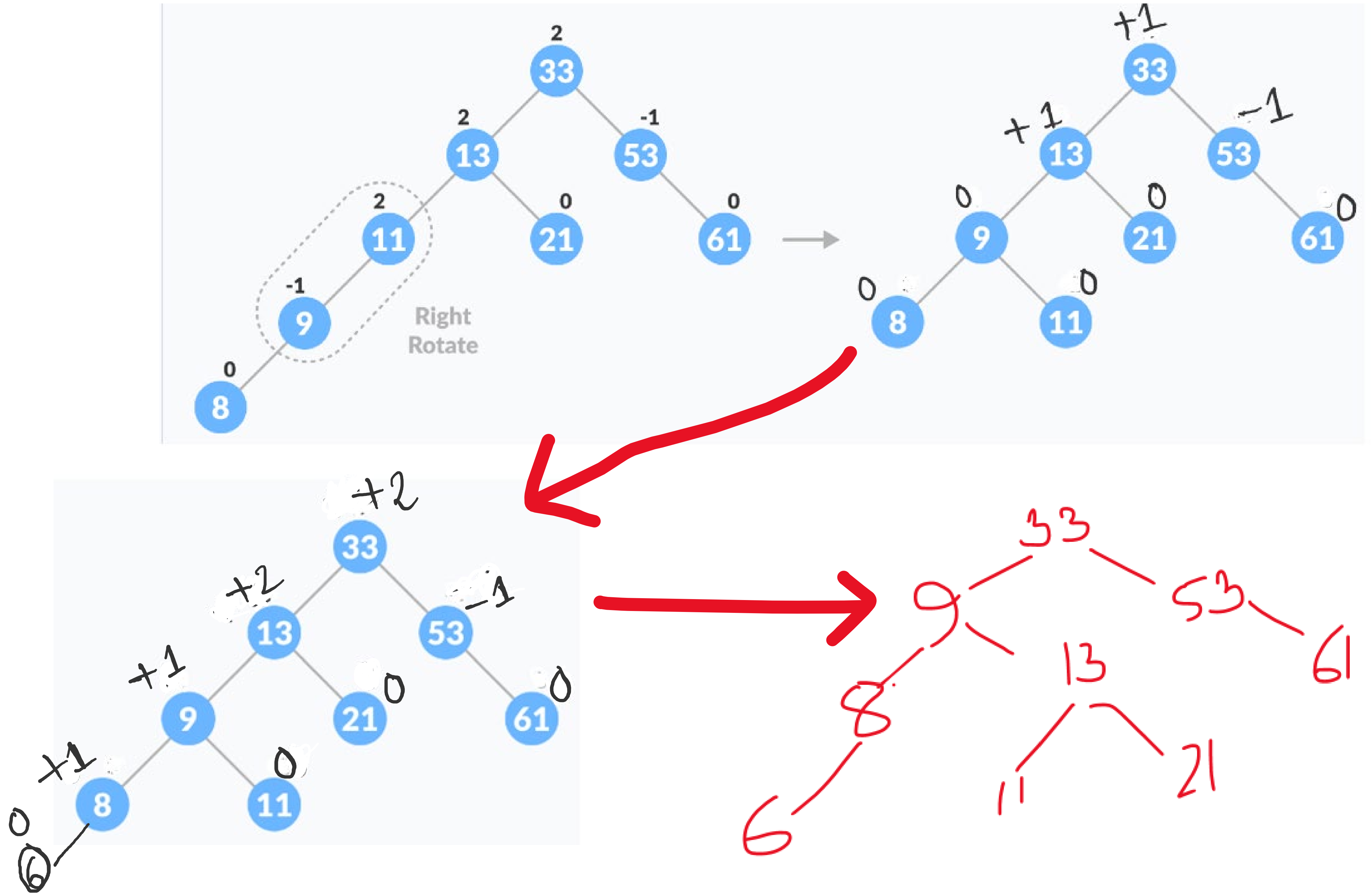
A new node is always inserted as a leaf node with balance factor equal to 0.






Left
Rotate

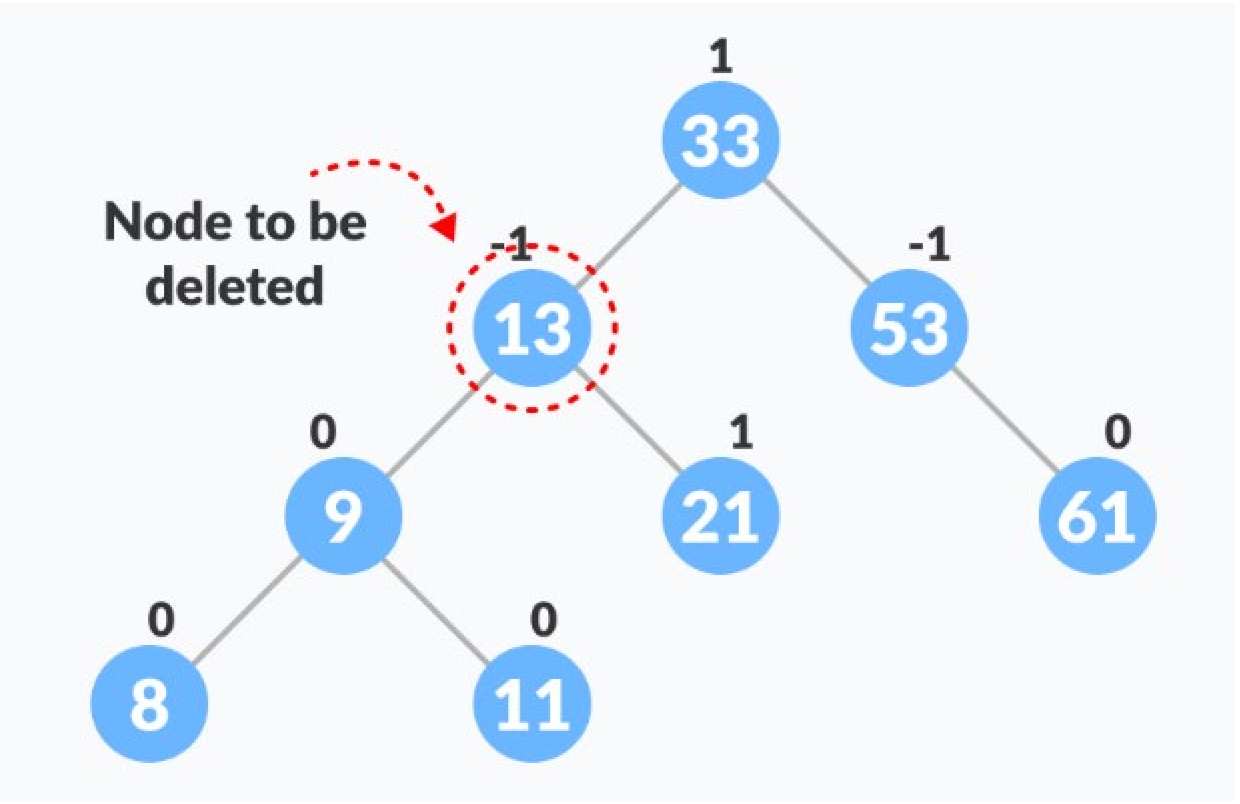

Right
Rotate


Right Rotate
2
33
2
13
-1
53
2
11
0
21
0
61
-1
9
0
8
+1
33
+1
13
-1
53
0
9
0
21
0
61
0
8
0
11
+2
33
+2
13
-1
53
+1
9
0
21
0
61
+1
8
0
11
0
6
33
9
53
8
13
61
6
11
21

Delete on AVL Tree

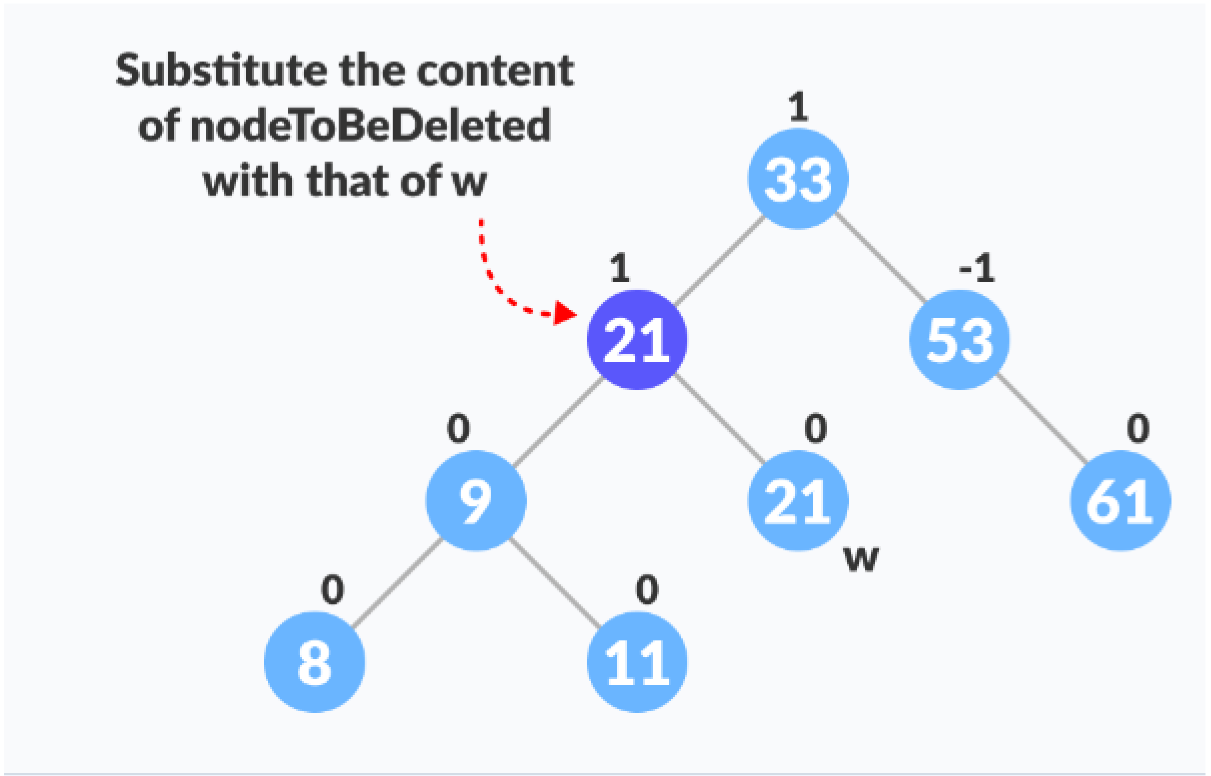
- Delete the node based on the BST’s delete procedure
- Starting from the parent of the deleted node, traverse up the tree to the root node, updating the balance factor of each node along the way.
- If a node's balance factor becomes greater than 1 or less than -1 during the traversal, then that node is unbalanced and a rebalancing operation must be performed.
- Depending on the type of imbalance (left-left, left-right, right-right, or right-left), perform a rotation operation to balance the subtree rooted at the unbalanced node.
- After the rotation operation is performed, update the balance factors of the affected nodes.
- Continue the traversal up the tree until the root node is reached, updating the balance factors of each node along the way.
- If the root node's balance factor becomes greater than 1 or less than -1, then perform a rotation operation to balance the entire tree.
- The AVL tree now has the new node inserted, and is still balanced.

Delete

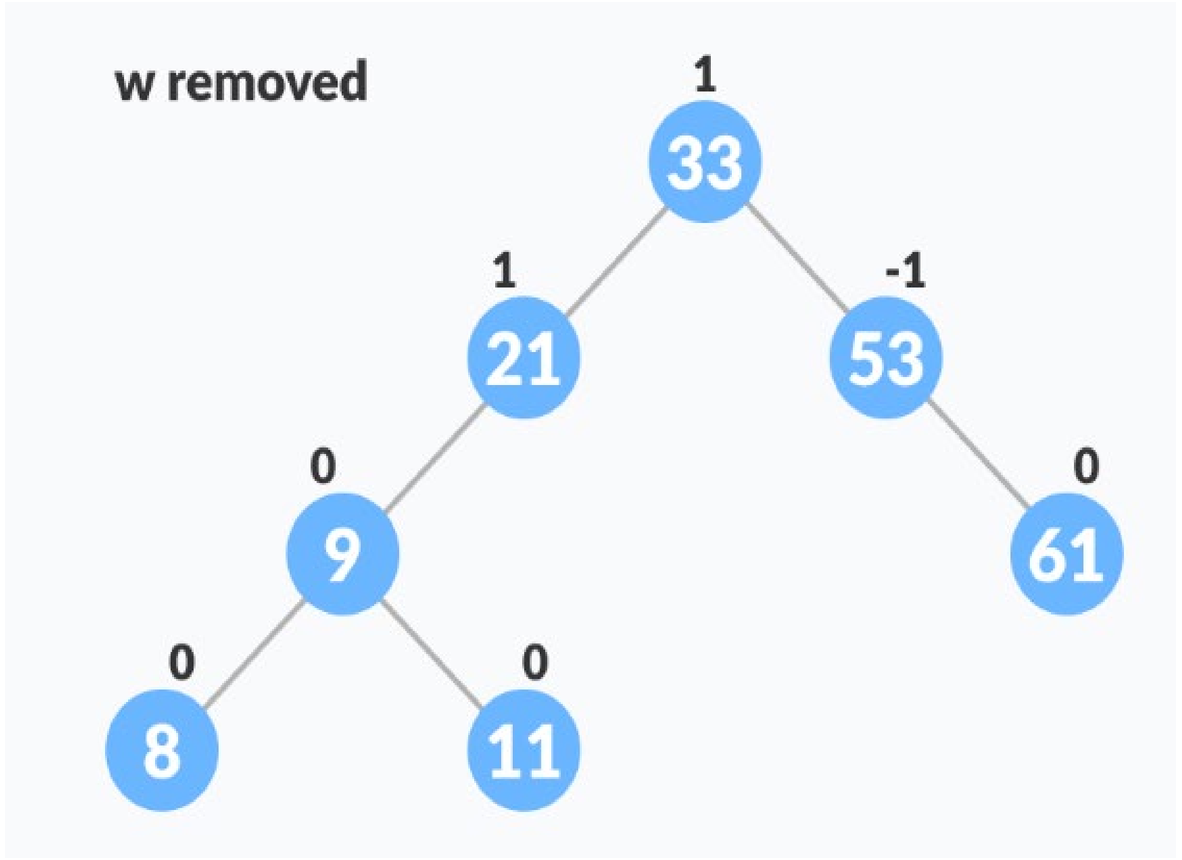


As we have seen in the BST, there are three possible cases for deleting a node:

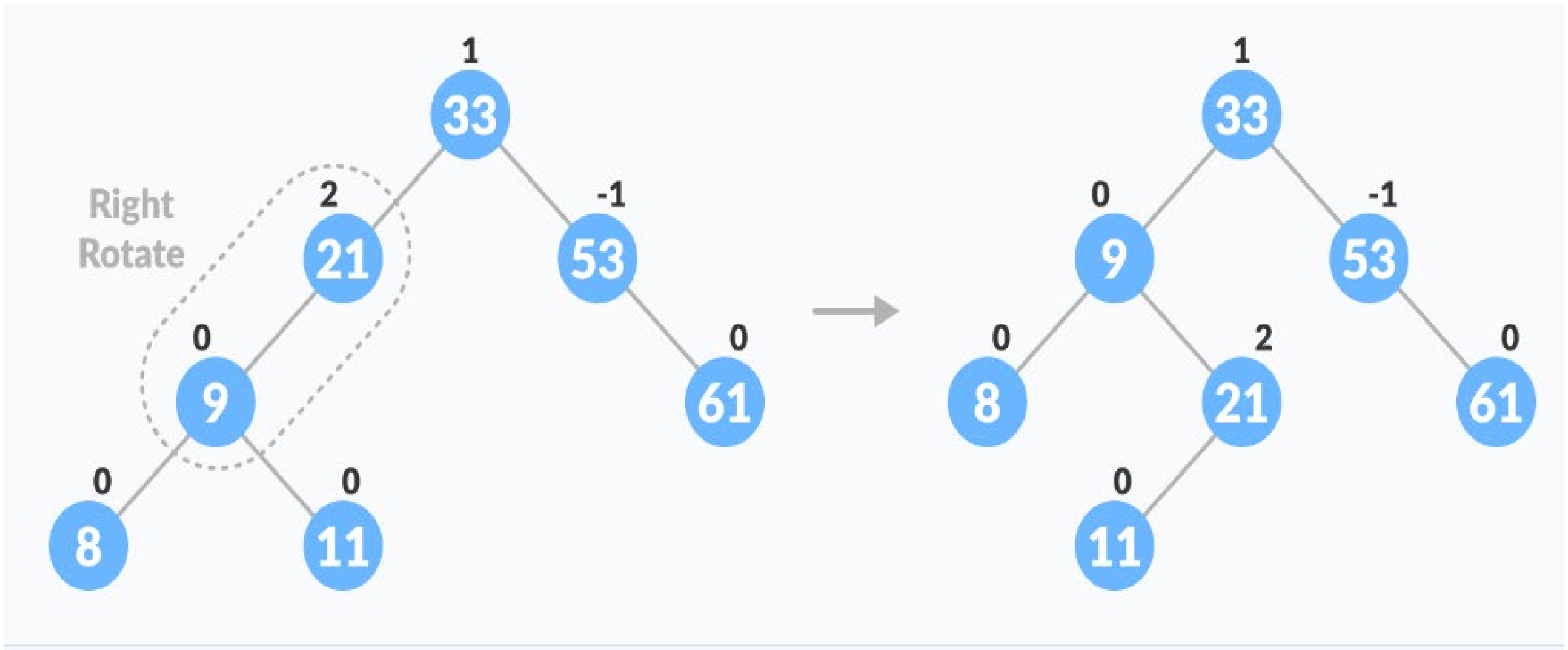
1. Leaf node
2. A node with one child
3. If the node has two children



Substitute the node to be deleted



Delete



Right-rotate for balancing the tree

Rebalance: (First condition)

- If balance factor (current node) > 1
  - If balance factor (left child) > = 0
    - Do right rotation
  - Otherwise do left-right rotation

Rebalance: (Second Condition)

- If balance factor (current node) < -1
  - If balance factor (right child) <= 0
    - Do left rotation
  - Otherwise, do right-left rotation

Red Black Tree

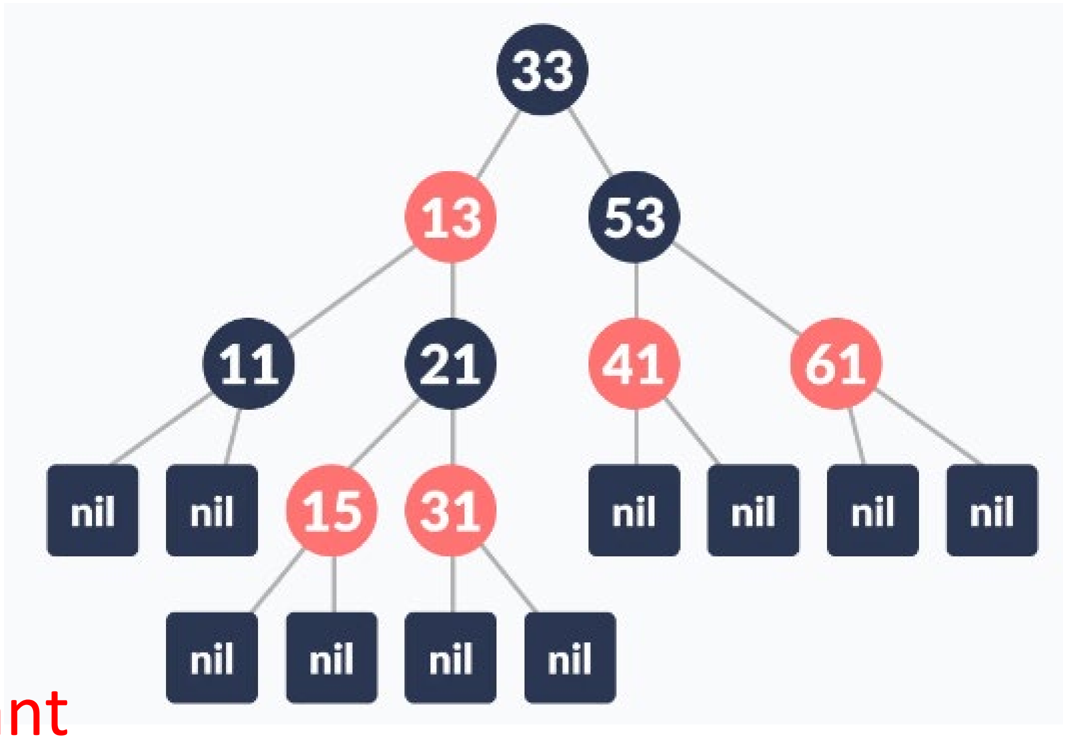
1.Every node is either red or black.

2.Every leaf (NULL) is black.

3.The root is black

4.If a node is red, then both its children are black.

5.Every simple path from a node to a descendant leaf contains the same number of black nodes.



The limitations put on the node colors ensure that any simple path from the root to a leaf is not more than twice as long as any other such path. It helps in maintaining the self-balancing property of the red-black tree.

Red Black Tree – Insertion

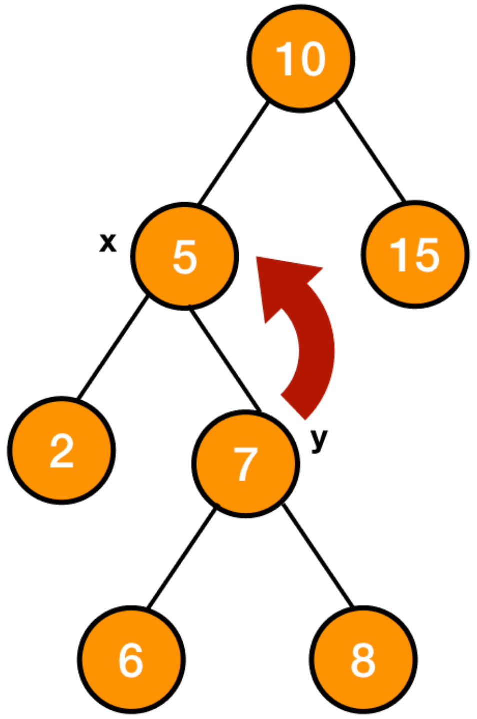
- A Red-Black Tree, every new node must be inserted with the color RED.
-
- The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property.

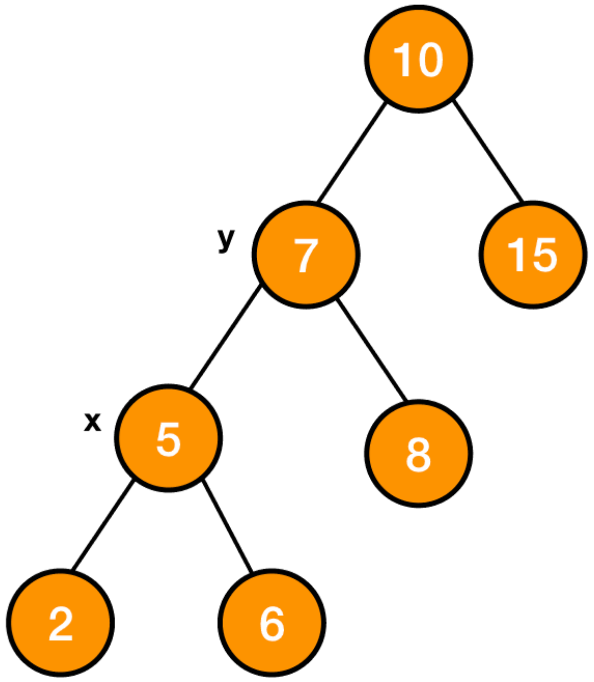
- After every insertion operation, we need to check all the properties of Red-Black Tree.

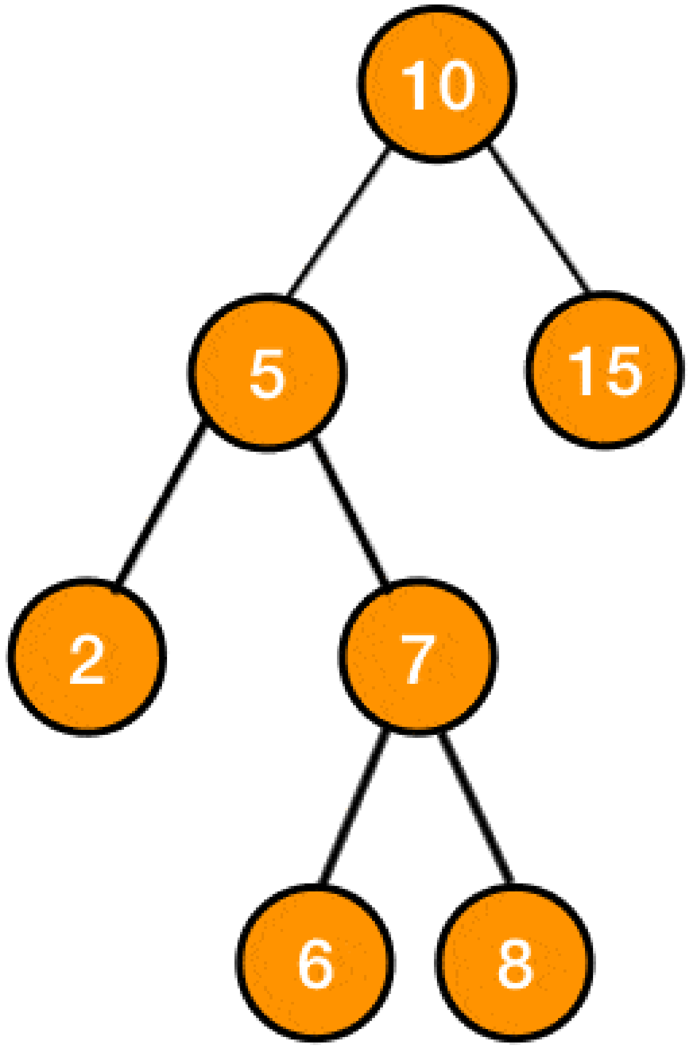
- If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

    - Recolor
    - Rotation
    - Rotation followed by Recolor

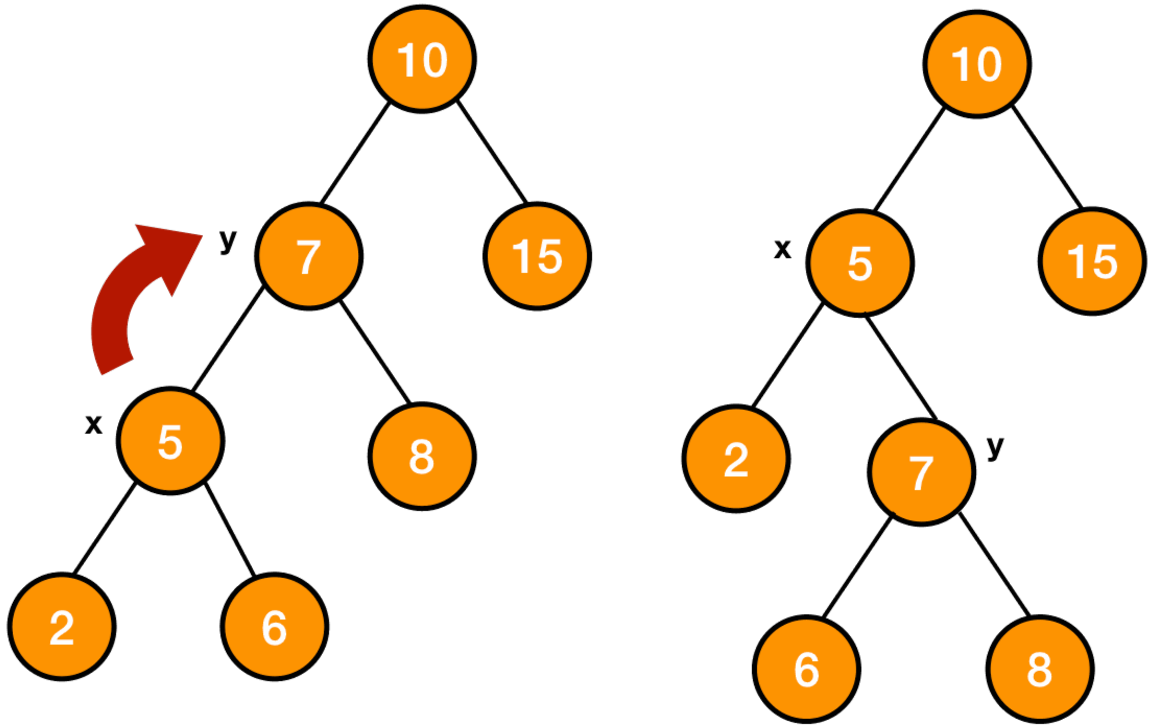
Left Rotation

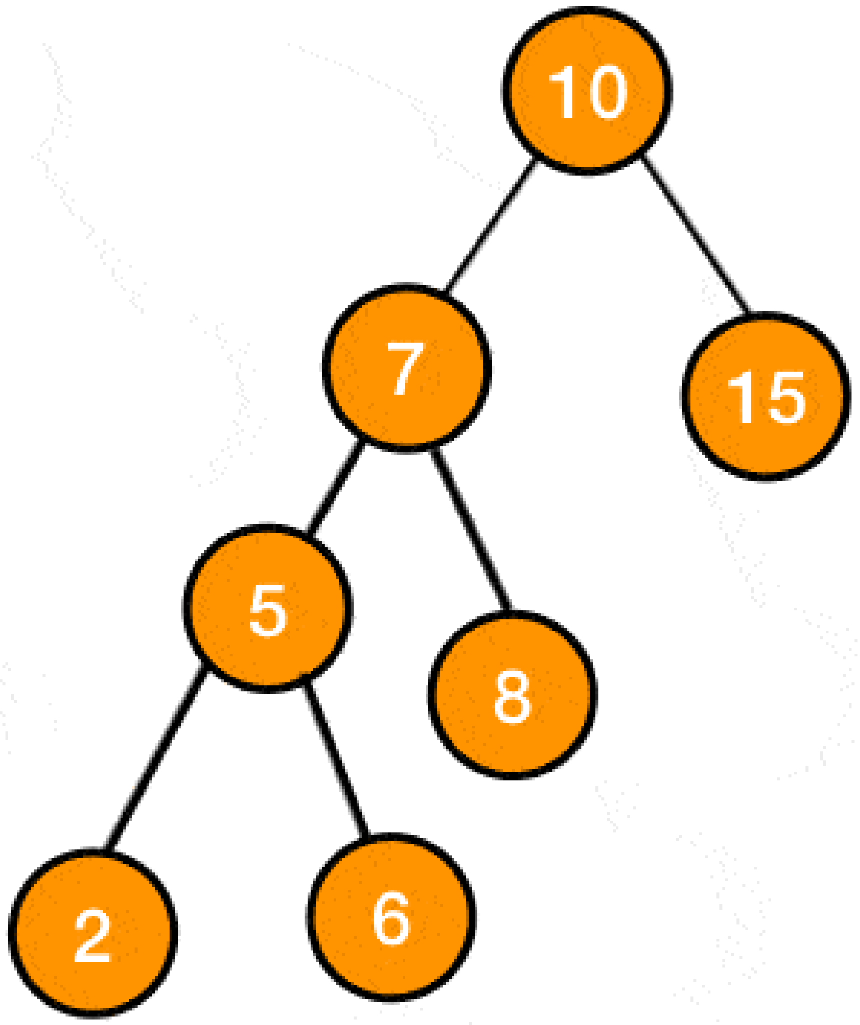






Right Rotation





Red Black Tree – Insertion

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

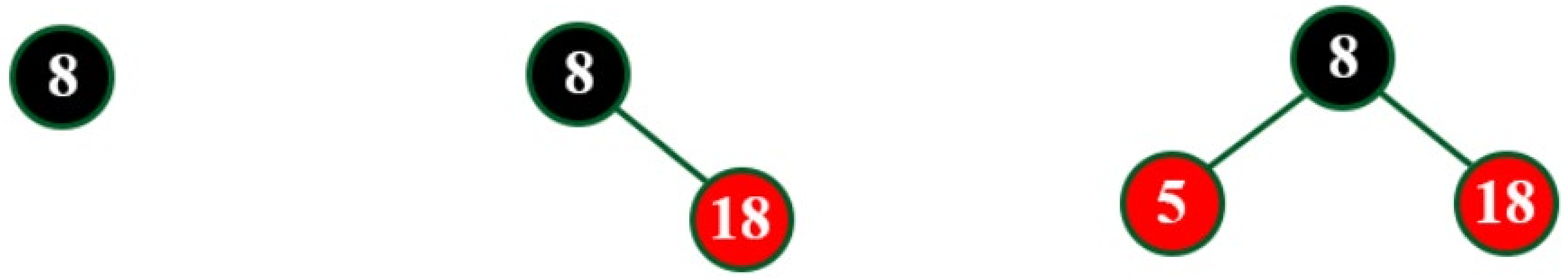
Step 4 - If the parent of newNode is Black then exit from the operation.

Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

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



If the newly inserted node is black. This would affect the black height condition and fixing that would be difficult.

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

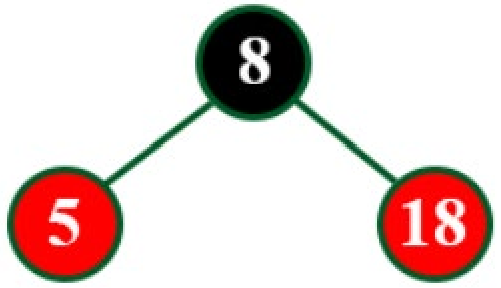
Step 4 - If the parent of newNode is Black then exit from the operation.

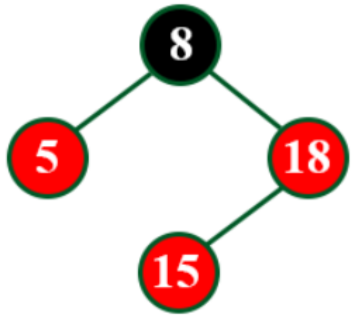
Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

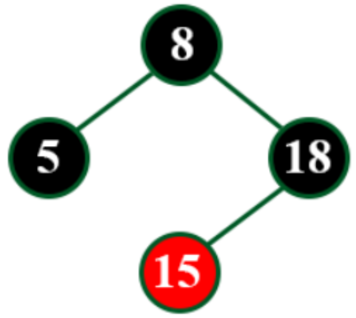
Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

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





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

Red Black Tree – Insertion

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

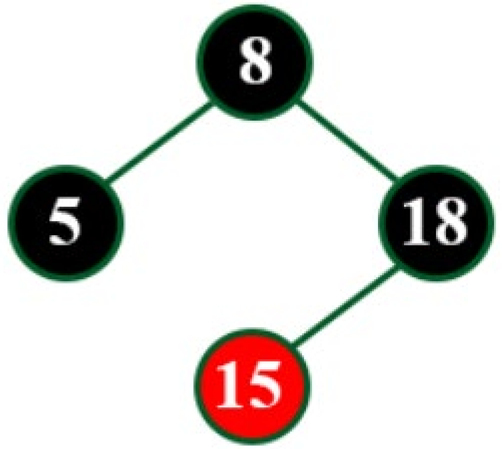
Step 4 - If the parent of newNode is Black then exit from the operation.

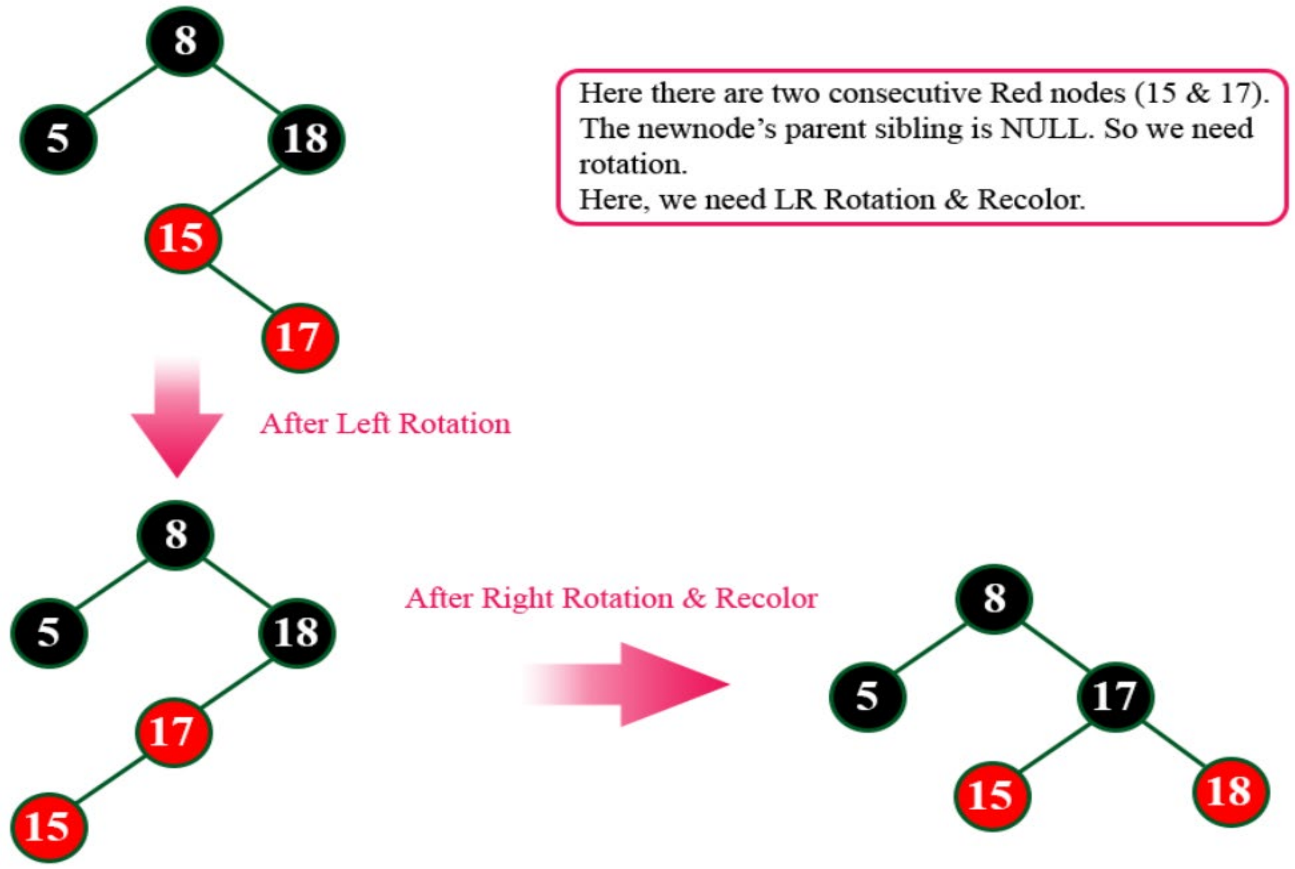
Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

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





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.

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

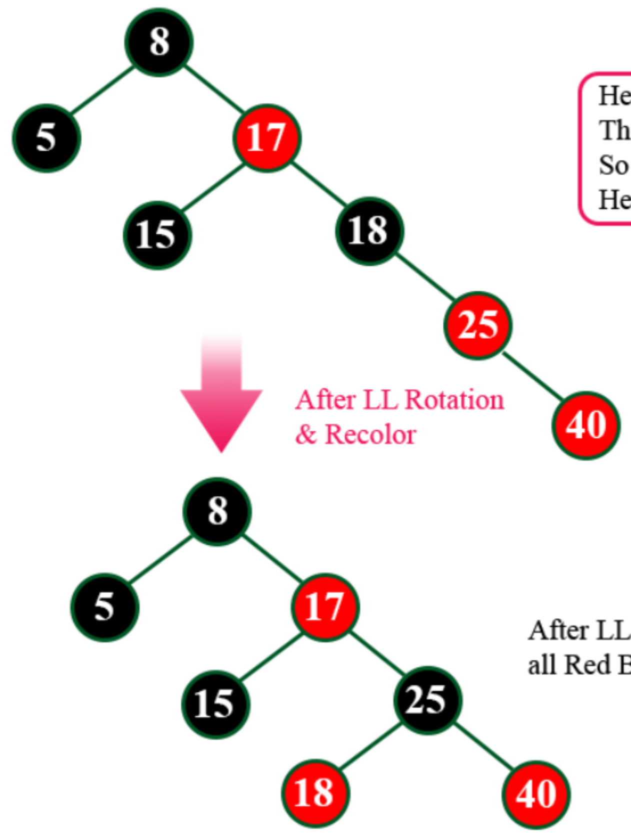
Step 4 - If the parent of newNode is Black then exit from the operation.

Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

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



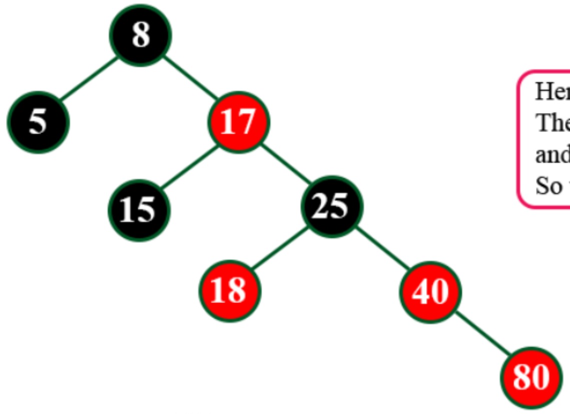
Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

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

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

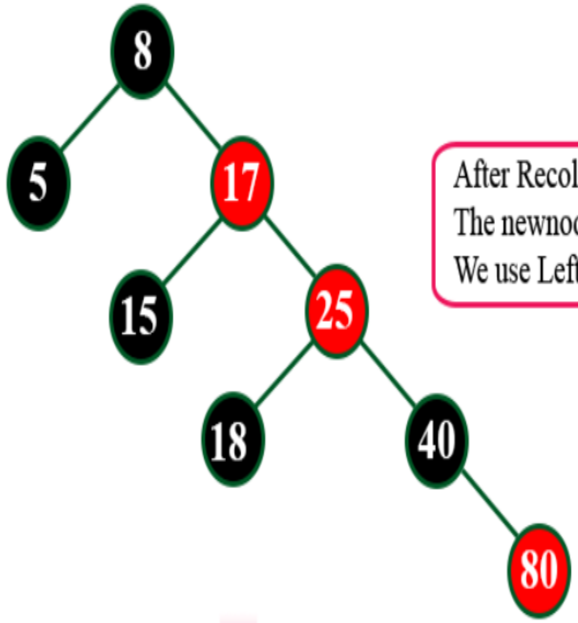
insert (80)

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



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

After Recolor



After Recolor again there are two consecutive Red nodes (17 & 25).
The newnode's parent sibling color is Black. So we need Rotation.
We use Left Rotation & Recolor.

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

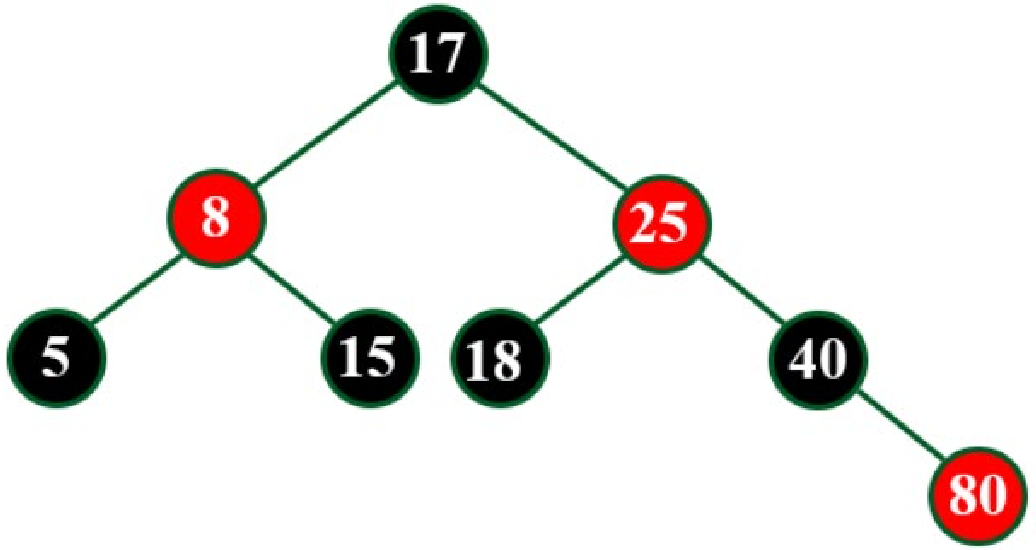
Step 4 - If the parent of newNode is Black then exit from the operation.

Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

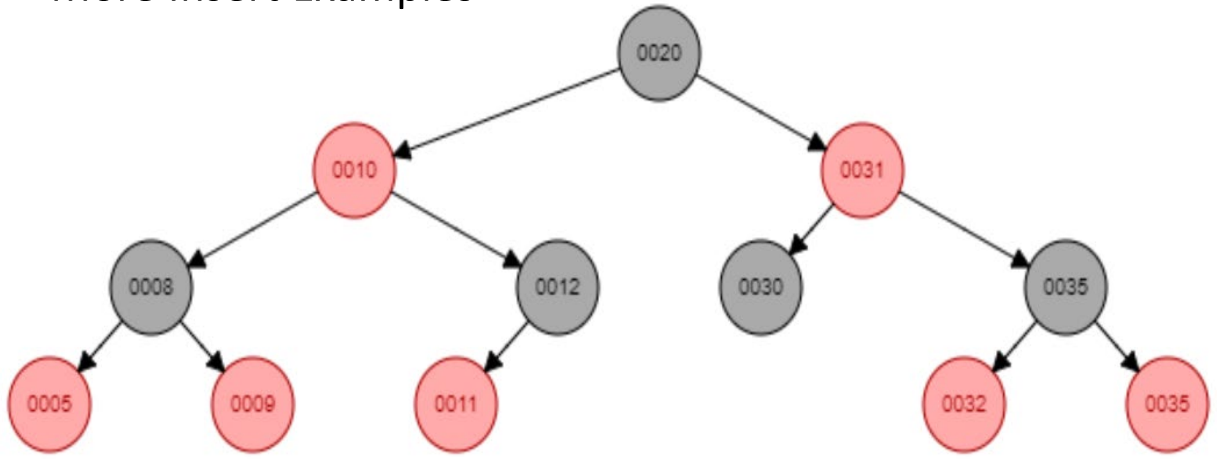
Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

After Left Rotation & Recolor



More Insert Examples



Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

Step 4 - If the parent of newNode is Black then exit from the operation.

Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

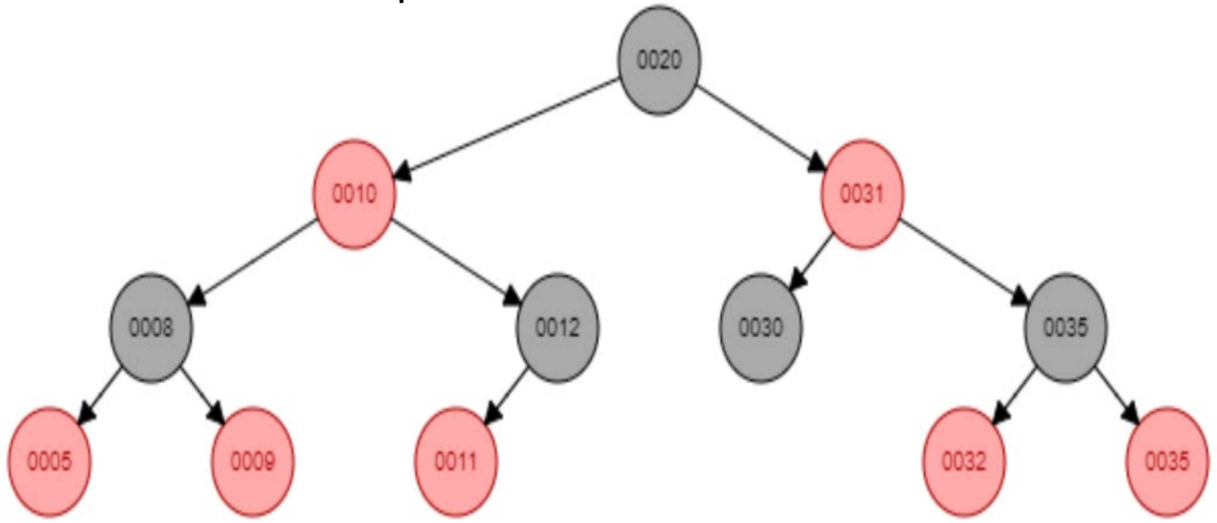
Step 4 - If the parent of newNode is Black then exit from the operation.

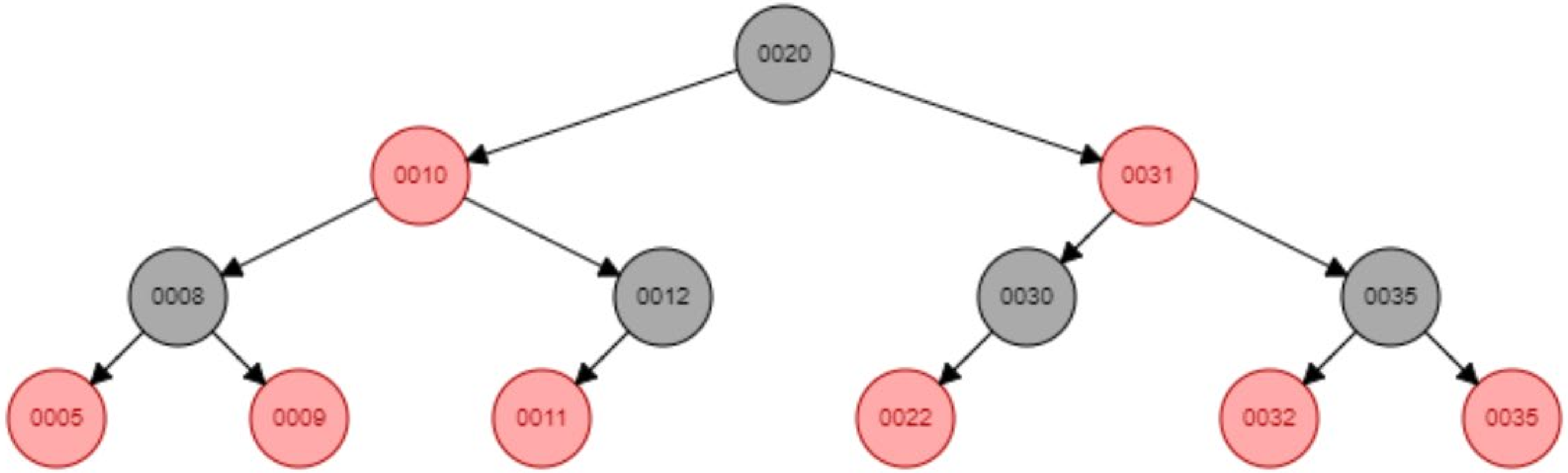
Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

More Insert Examples





Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

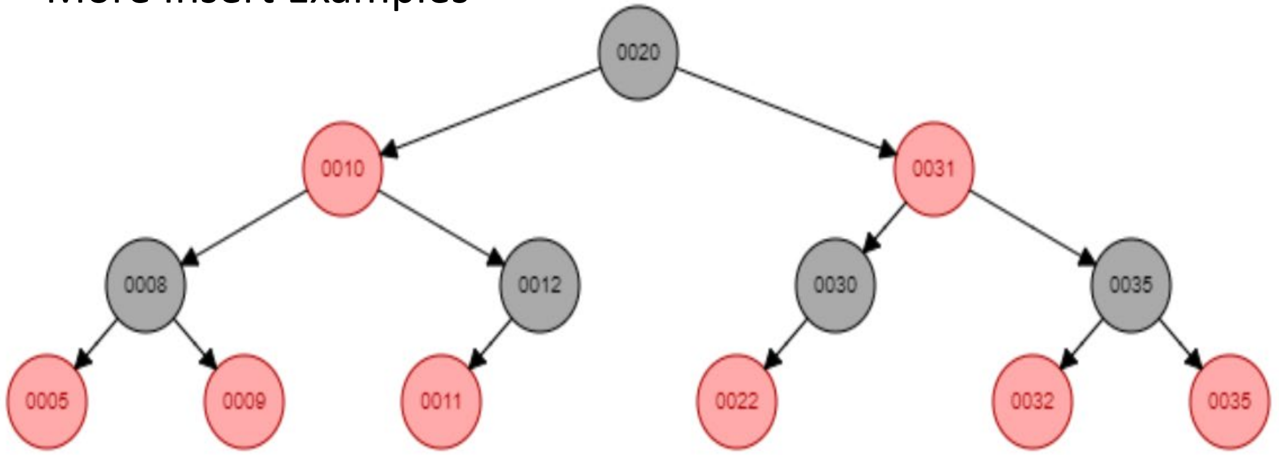
Step 4 - If the parent of newNode is Black then exit from the operation.

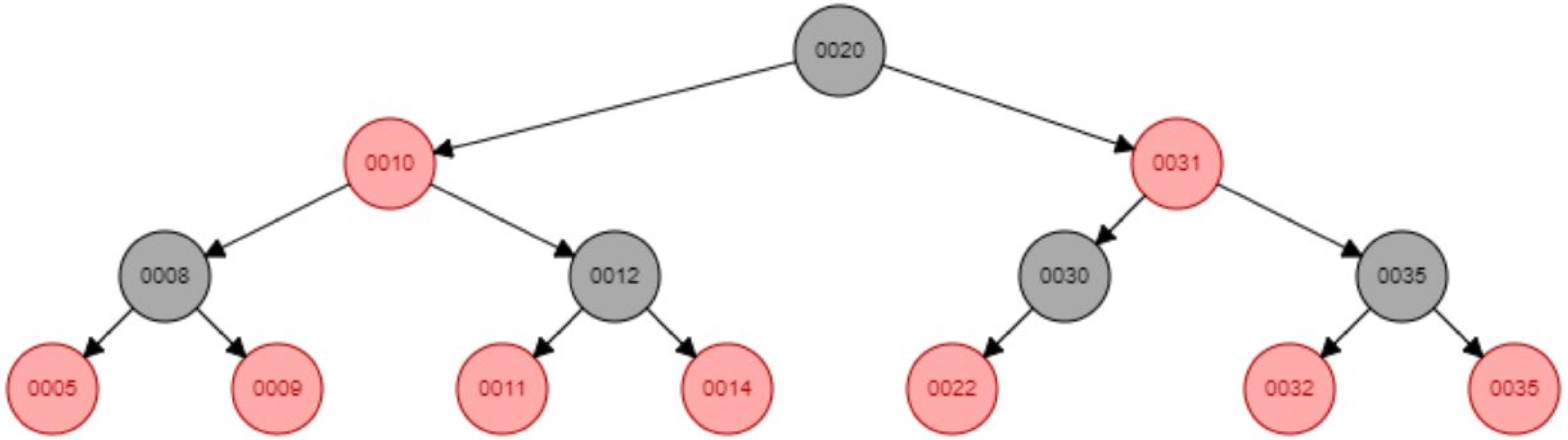
Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

More Insert Examples





Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

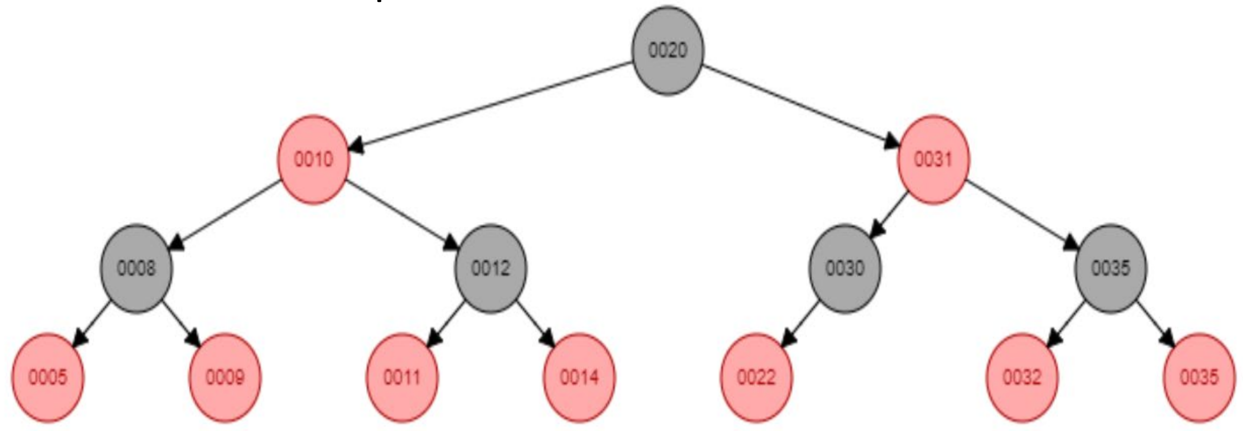
Step 4 - If the parent of newNode is Black then exit from the operation.

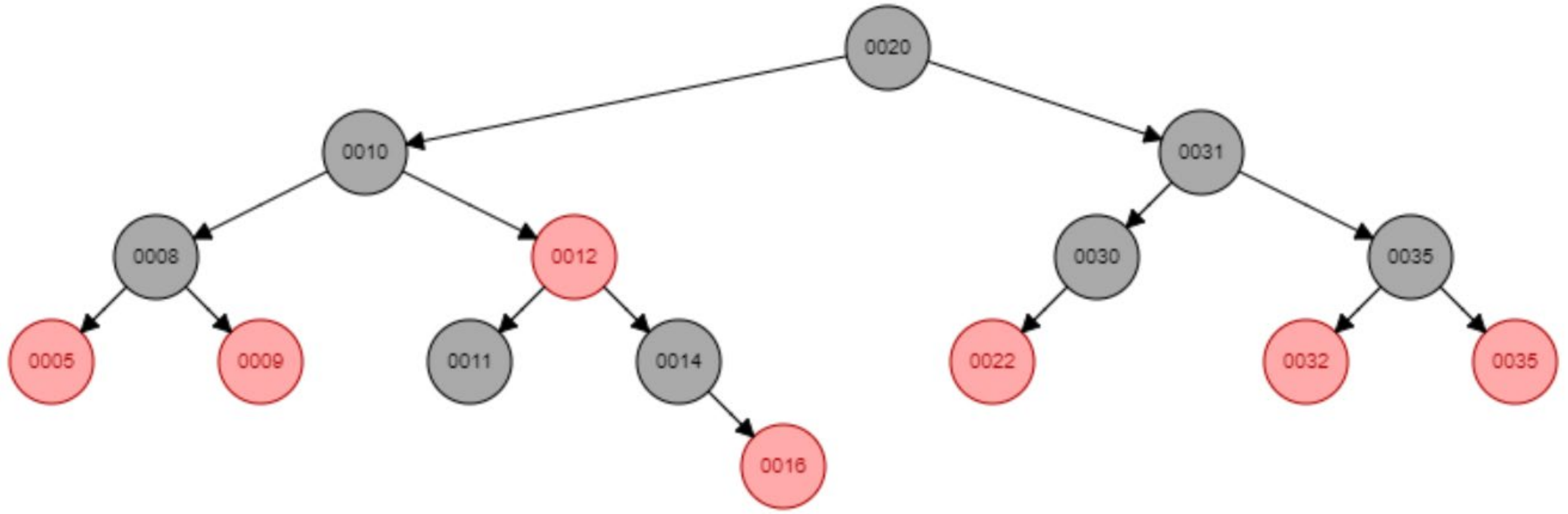
Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

More Insert Examples





Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the ope

Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.

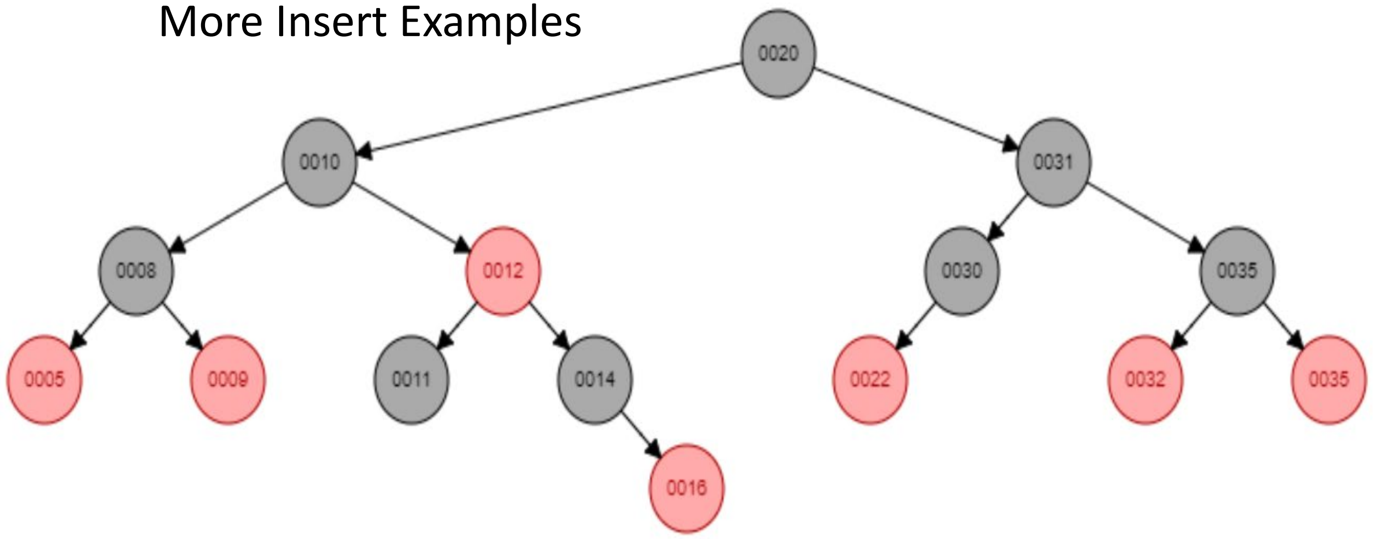
Step 4 - If the parent of newNode is Black then exit from the operation.

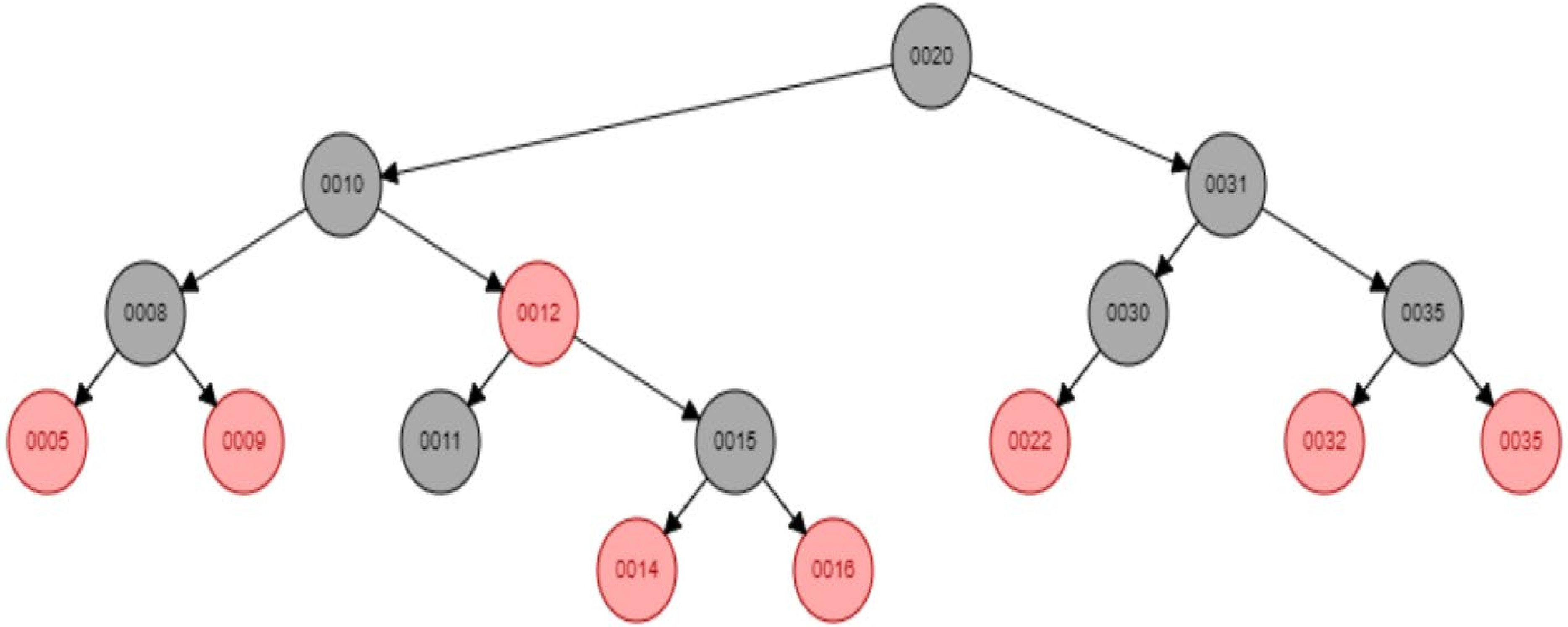
Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.

Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.

Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

More Insert Examples





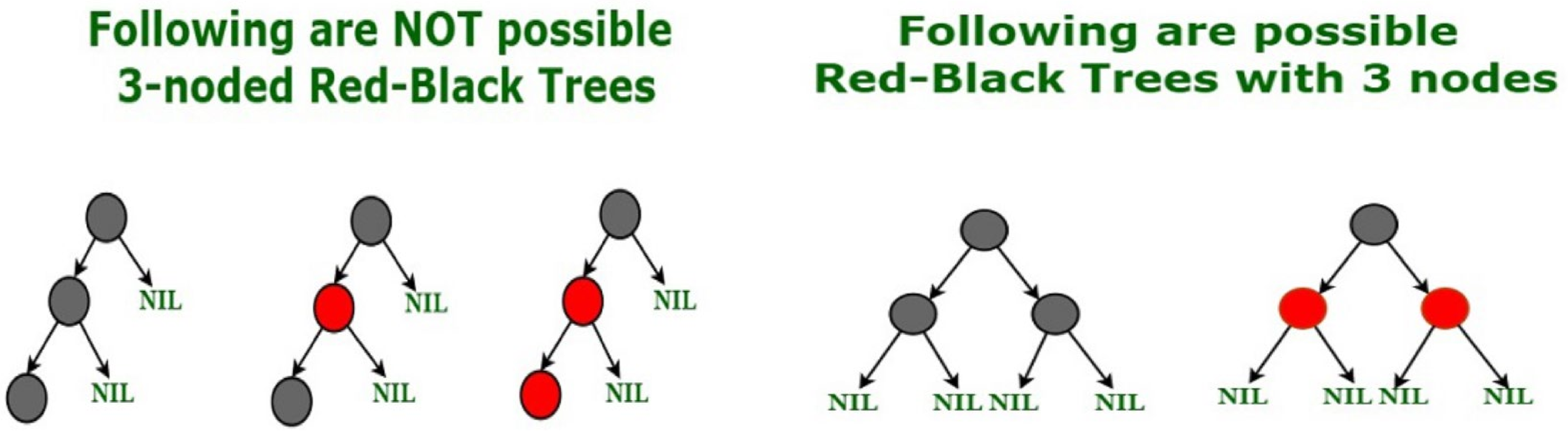
Deletion Operation in Red Black Tree

- The deletion operation in Red-Black Tree is similar to deletion operation in BST.
- But after every deletion operation, we need to check with the Red-Black Tree properties.
- If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Re-Color to make it Red-Black Tree.

Red Black Tree vs AVL Tree

- ❑ The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion.
- ❑ So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred.
- ❑ if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over the Red-Black Tree.

Ensuring Balance?


Following are NOT possible 3-noded Red-Black Trees
Following are possible Red-Black Trees with 3 nodes
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL