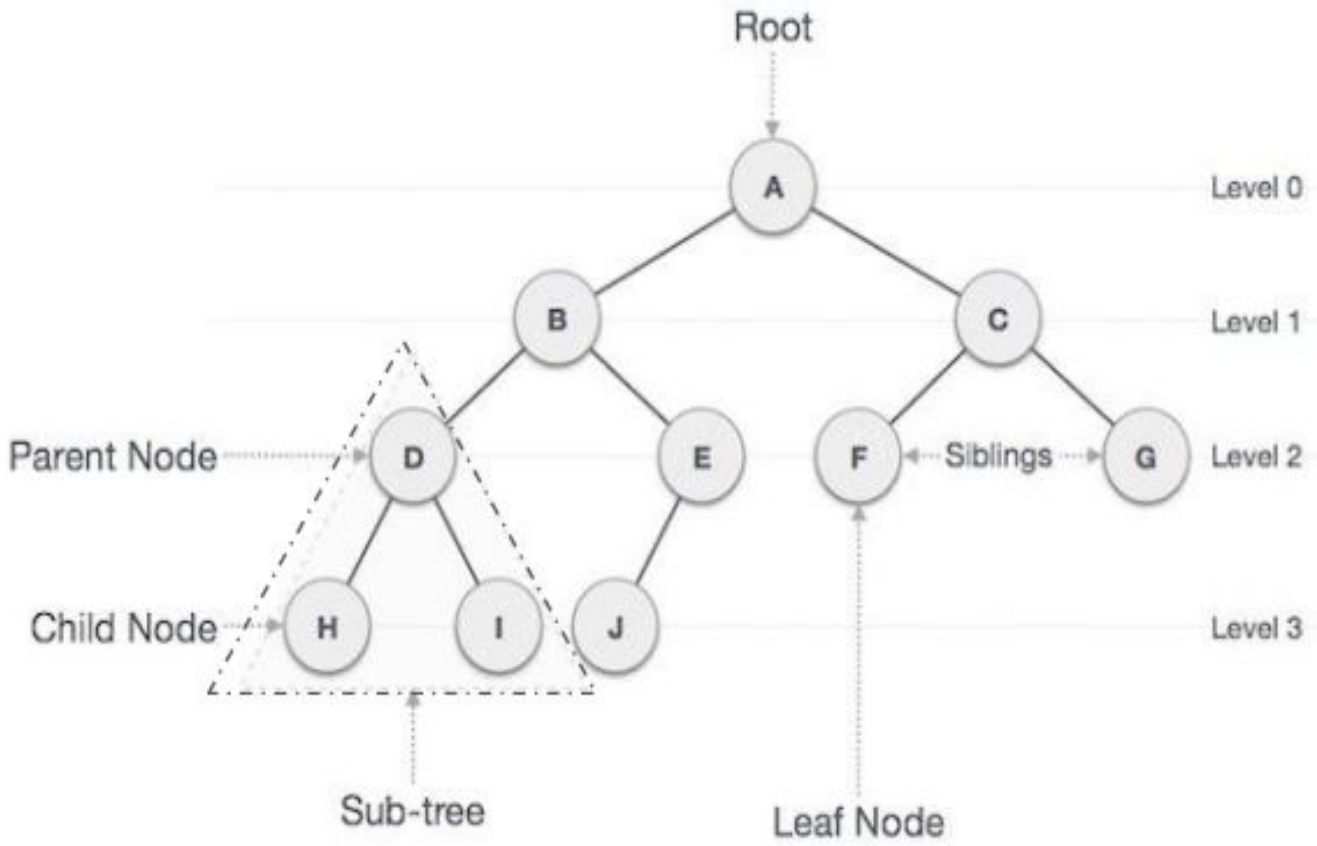
# Trees

# Basic Terminology

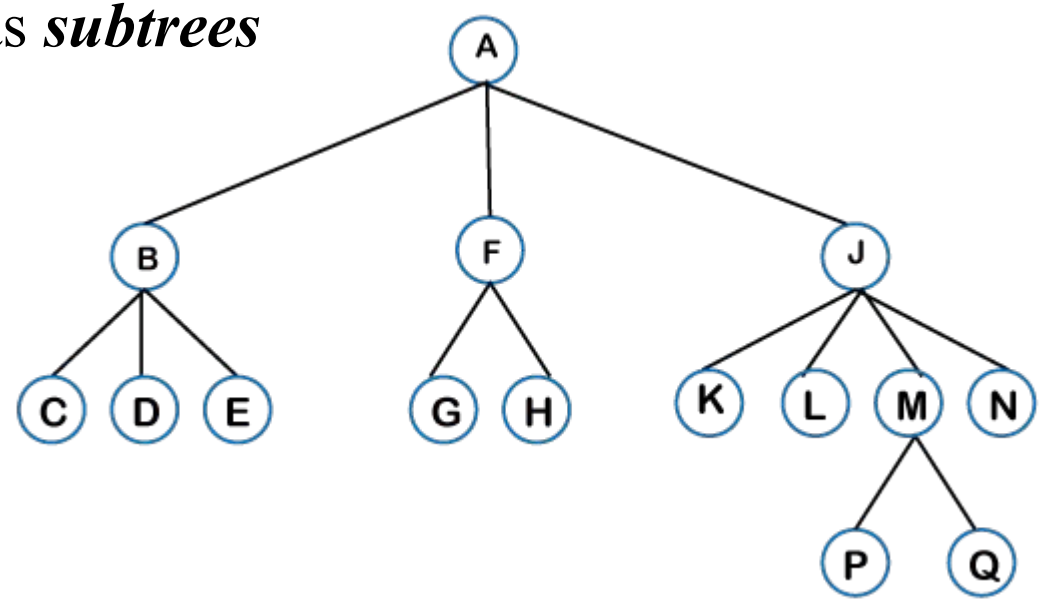


- Introduction to Tree
- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. Each node contains some data and the link or reference of other nodes that can be called children.

- Root is a special node in a tree. The entire tree originates from it. It does not have a parent. E.g. A
- Parent node is an immediate predecessor of a node. E.g. B is parent of D and E
- All immediate successors of a node are its children. D and E are children of B.
- Node which does not have any child is called as leaf. E.g. H, I, J, F and G are leaf nodes
- Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf. E.g. Line between A and B is edge
- Nodes with the same parent are called siblings. E.g. D and E are siblings
- Path is a number of successive edges from source node to destination node. A-B-E-J is path from node A to E
- The Depth The number of edges from the tree's node to the root is.
- The Height of a tree is the number of edges from the node to the deepest leaf. The tree height is also considered the root height E.g. Height of A is edges between A and H, as that is the longest path, which is 3.
- Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on. E. g. Level of H, I & J is 3. Level of D, E, F & G is 2
- Degree of a node represents the number of children of a node. E.g. Degree of D is 2 and of E is 1
- Descendants of a node represent subtree. E.g. Nodes D, H, I represent one subtree.
- Ancestor node:- An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors.
- Descendant: The immediate successor of the given node is known as a descendant of a node.

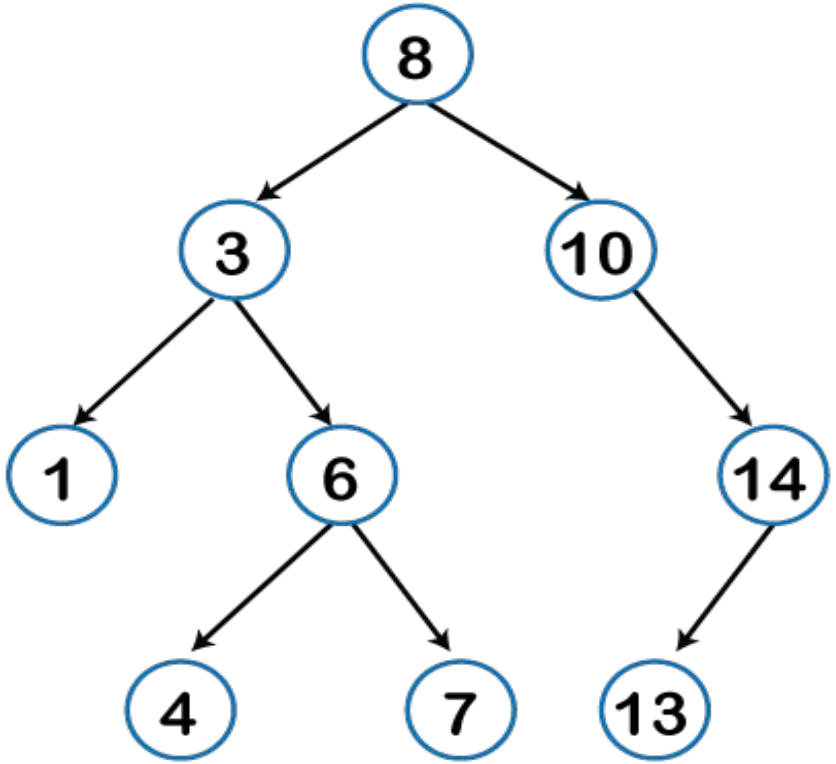
- **General tree:**

The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***

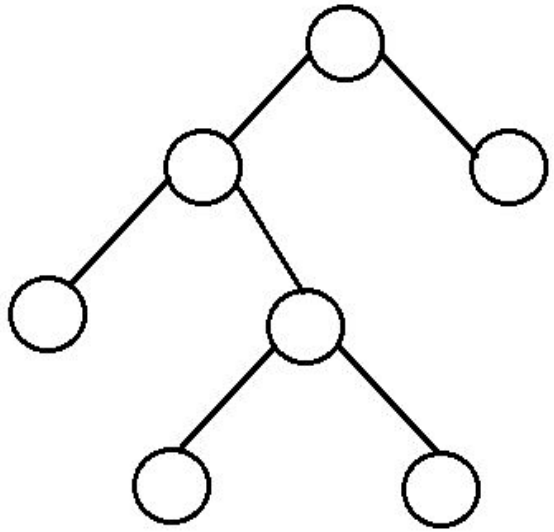


- Binary Tree:

In a Binary tree, every node can have at most 2 children, left and right. In diagram below, node 3 forms left subtree and node 10 forms right subtree.

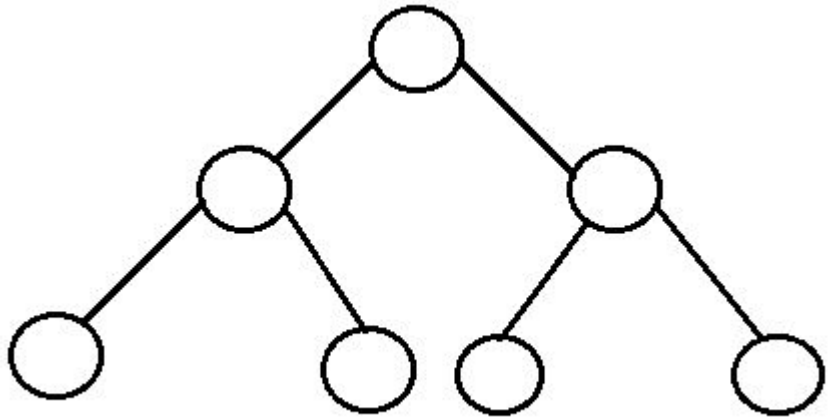


- Full Binary Tree: It is a tree in which every node in the tree has either 0 or 2 children



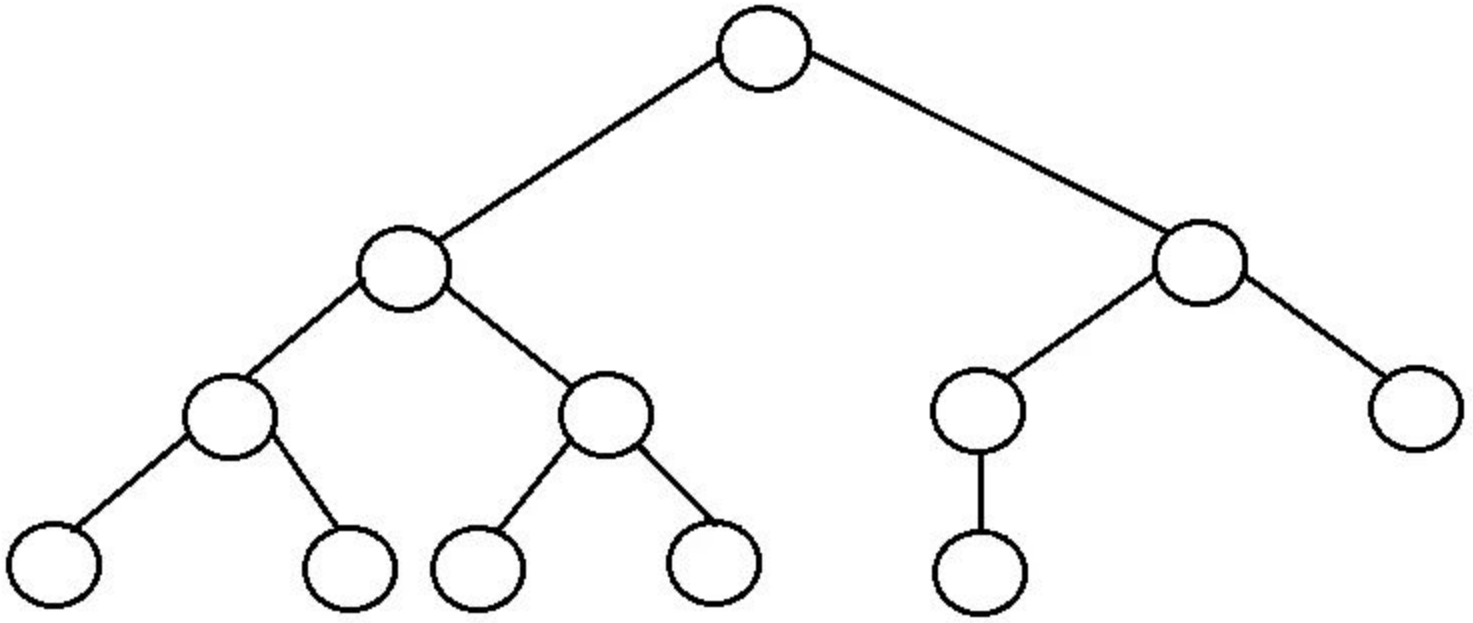
- The number of nodes, $n$, in a full binary tree is at least n = 2^h – 1, and atmost $n = 2^{h+1} - 1$, where $h$ is the height of the tree.
- The number of leaf nodes $l$, in a full binary tree is number, $L$ of internal nodes + 1, i.e, $l = L+1$.

**Perfect binary tree:** It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



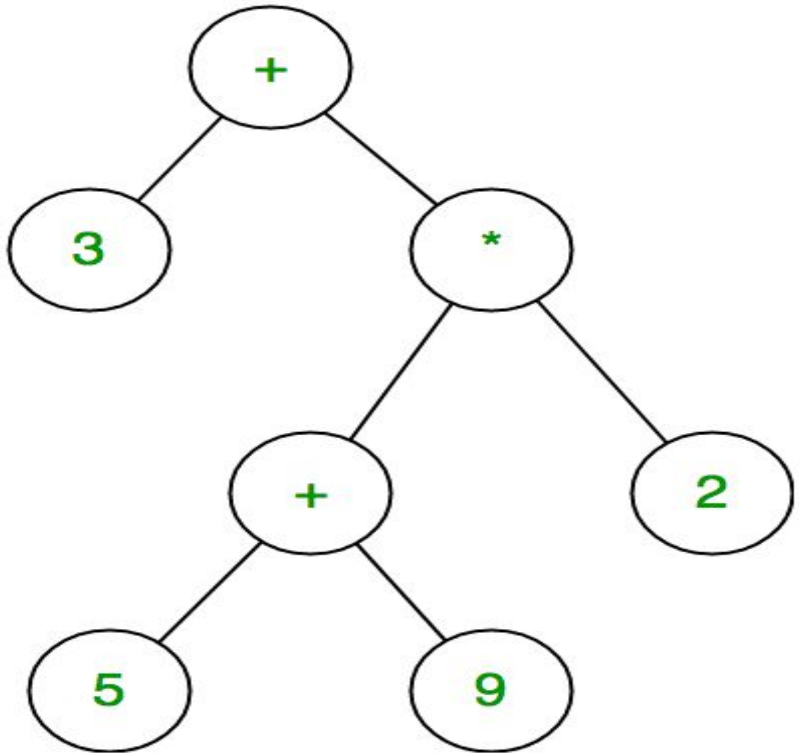
- A perfect binary tree with *l* leaves has *n = 2*l-1* nodes.
- In perfect full binary tree, *l = 2^h* and *n = 2^(h+1) - 1* where, *n* is number of nodes, *h* is height of tree and *l* is number of leaf nodes

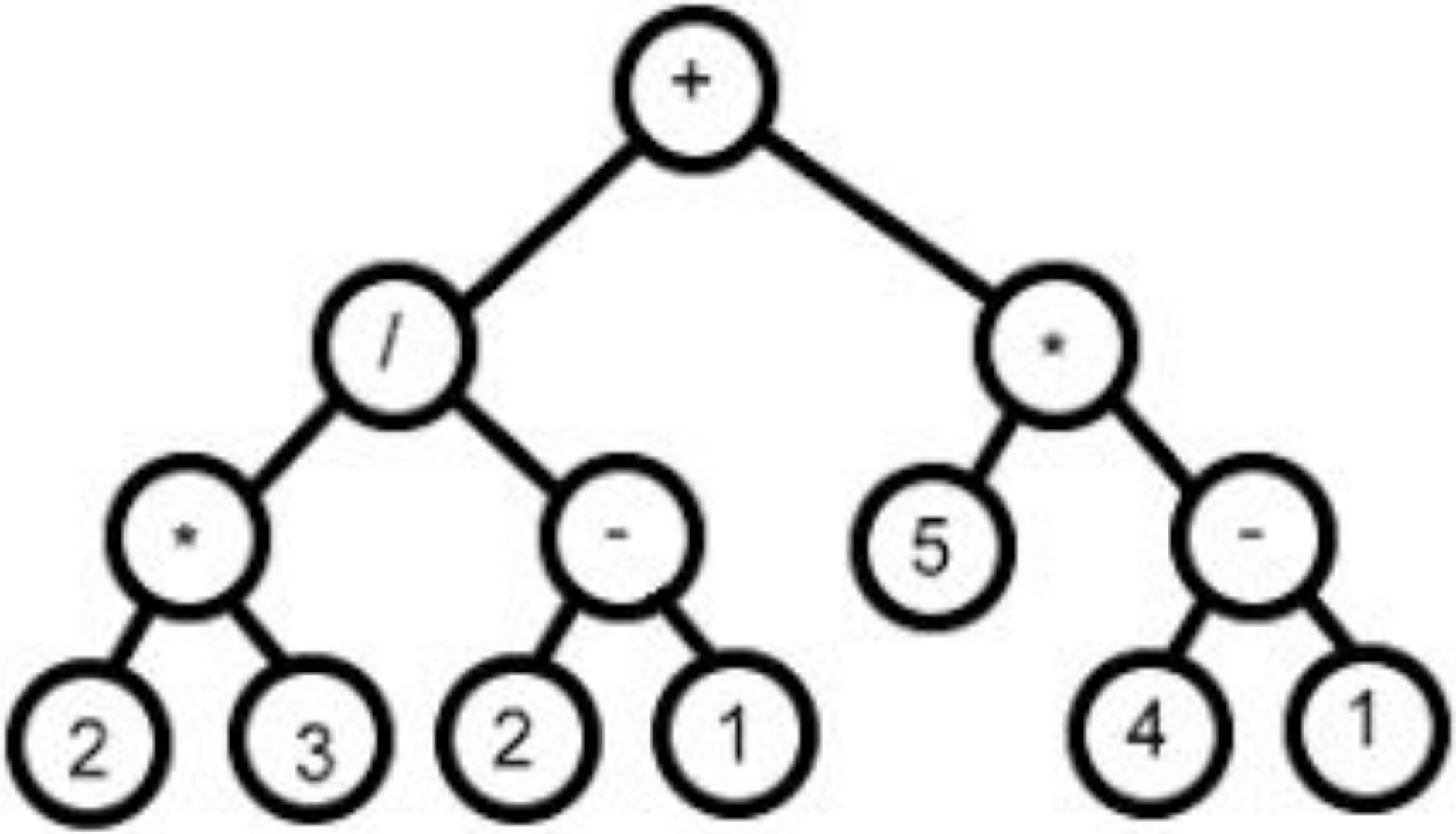
- **Complete Binary Tree:** A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.



In complete binary tree, the left and right children of node K is 2*K and 2*K+1 respectively. The parent of node K is [K/2].

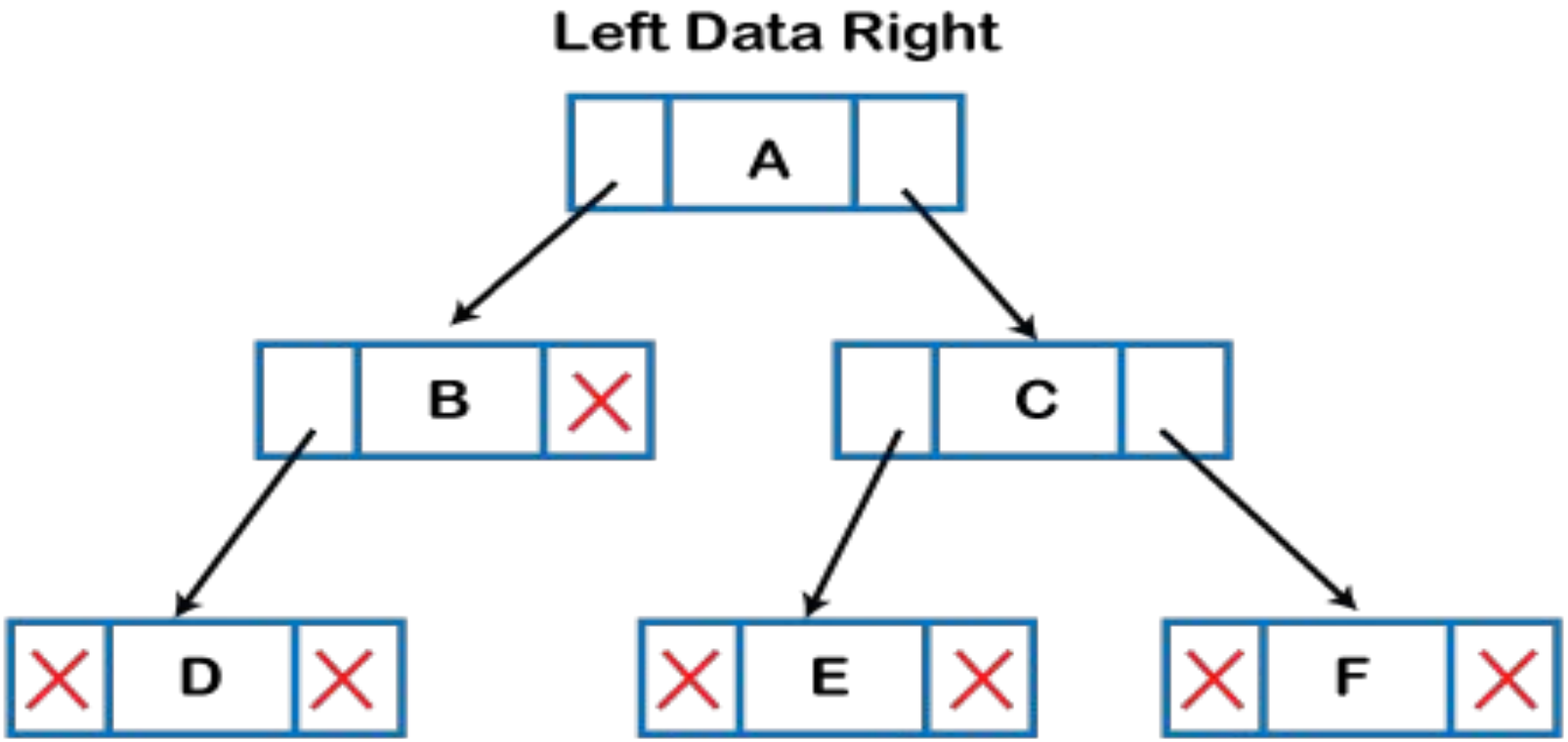
- Expression Tree: The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand.
- Expression tree for 3 + ((5+9)*2) would be:






Expression tree for 2*3/(2-1)+5*(4-1)

# Representation of Binary Tree in memory



- The above figure shows the representation of the tree data structure in the memory. In linked and dynamic representation, the linked list data structure is used. Each node constitutes of a data part and two link parts. The two link parts store address of left and right child nodes. Data part is used to store the information about the binary tree element. This is a better representation as nodes can be added or deleted at any location. Memory utilization is better in this binary tree representation.

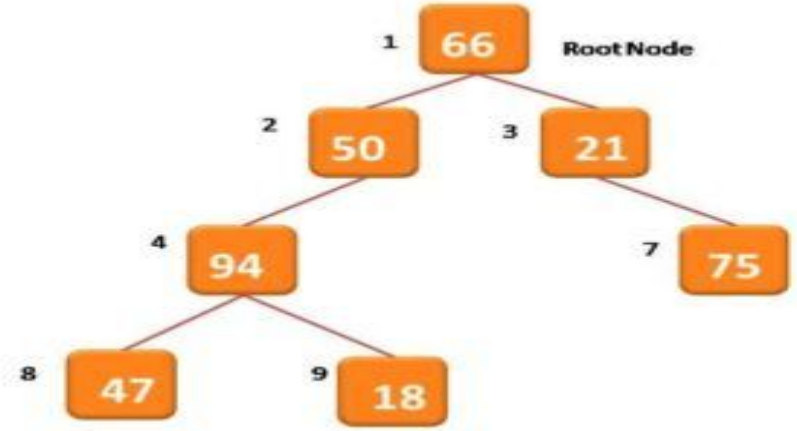
```
struct node
{
    int data;
    struct node *left;
    struct node *right;
}
```

Sequential Representation of binary Trees:

Suppose T is a binary tree that is complete or nearly complete. Then there is an efficient way of maintaining T in memory called sequential representation of T. This representation uses only a single linear array Tree as follows:

- The root R of T is stored in Tree[1].
- If node N occupies Tree[k], then its left child is stored in Tree[2*k] and its right child is stored in Tree[2*k + 1].

Again Null is used to indicate an empty subtree. If Tree[1] = Null then the tree is empty.



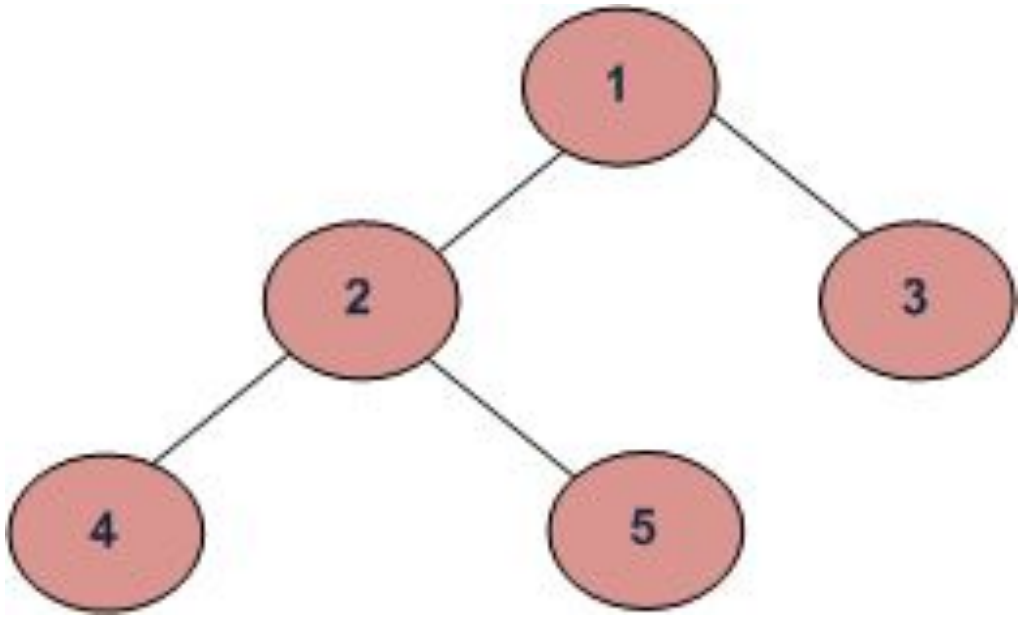

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 66 | 50 | 21 | 94 | | | 75 | 47 | 18 |

# Traversing Binary Tree

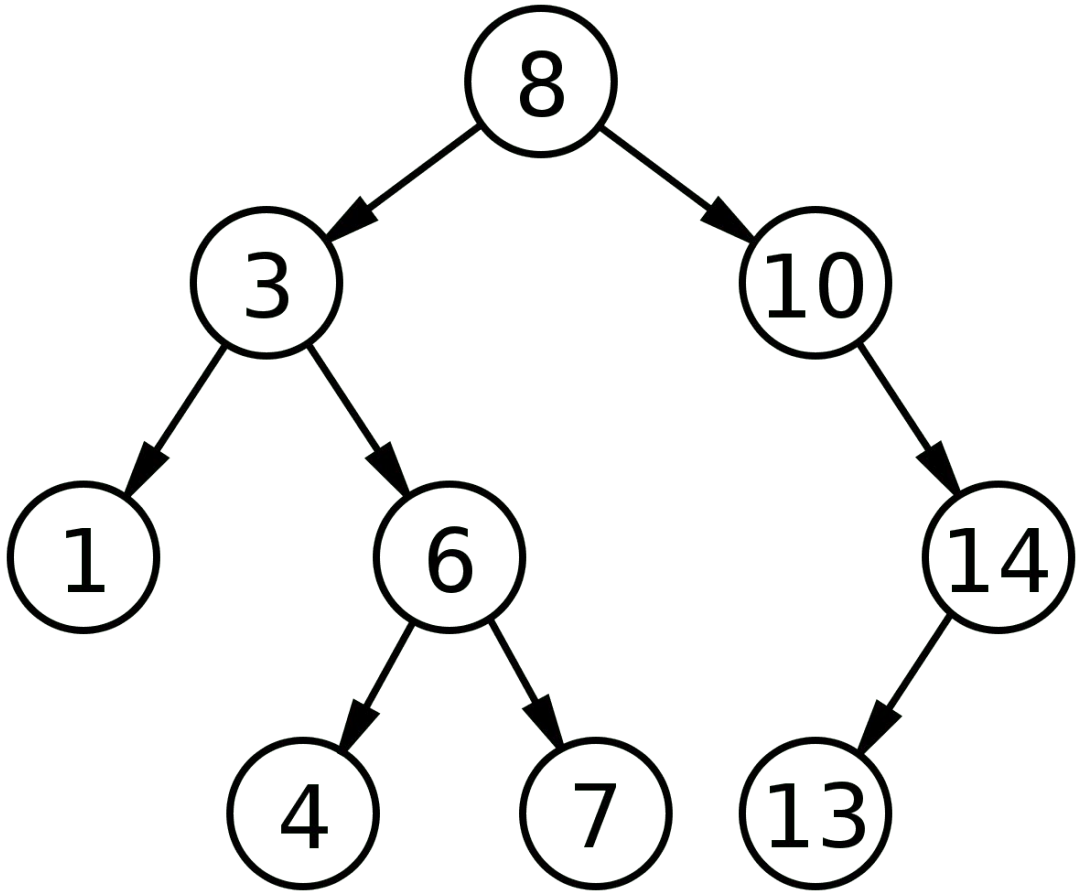
- There are three standard ways of traversing a binary tree T with root R. These three algorithms are called preorder, inorder, postorder, are as follows:
- Preorder
    1. Process the root R.
    2. Traverse the left subtree of R in preorder.
    3. Traverse the right subtree of R in preorder.
- Inorder
    1. Traverse the left subtree of R in inorder.
    2. Process the root R.
    3. Traverse the right subtree of R in inorder.
- Postorder
    1. Traverse the left subtree of R in postorder.
    2. Traverse the right subtree of R in postorder.
    3. Process the root R.

The three algorithms are sometimes called , respectively, the root-left-right(**R**LR)traversal, the left-root-right(L**R**R) traversal and the left-right-root(LR**R**) traversal.

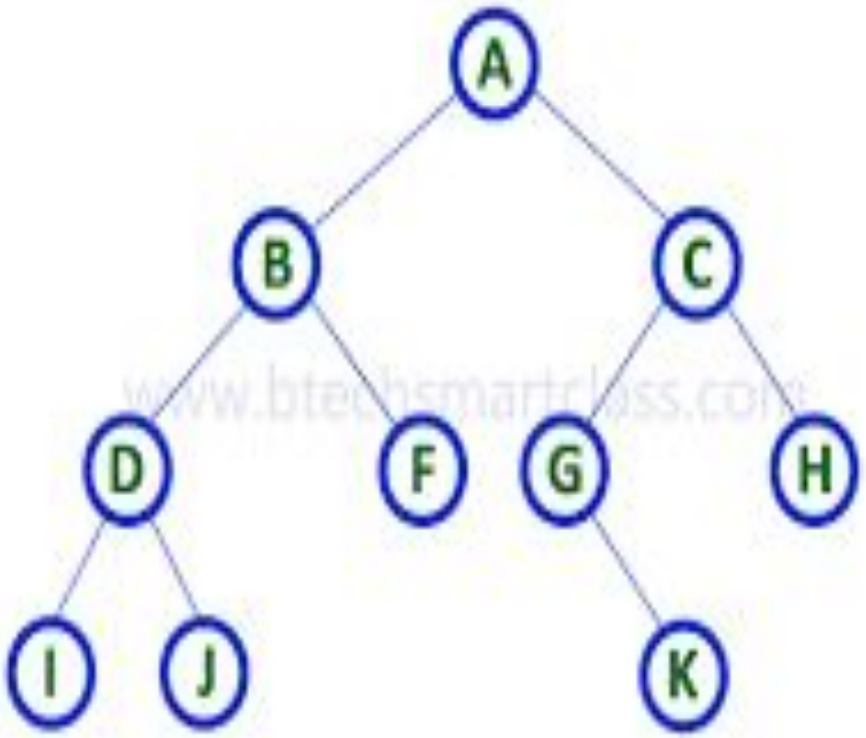
Preorder: 1 2 4 5 3
Inorder: 4 2 5 1 3
Postorder: 4 5 2 3 1



Preorder: 8 3 1 6 4 7 10 14 13
Inorder: 1 3 4 6 7 8 10 13 14
postorder: 1 4 7 6 3 13 14 10 8



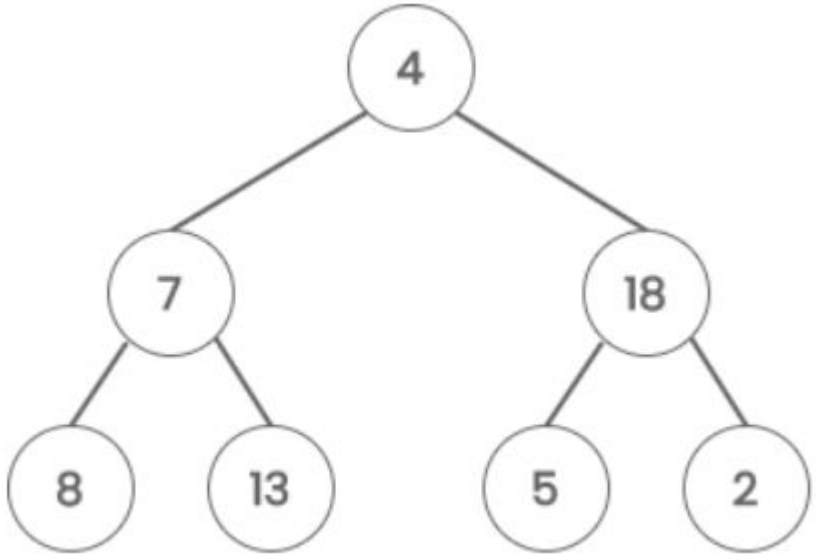
Preorder: A B D I J F C G K H
Inorder: I D J B F A G K C H
Postorder: I J D F B K G H C A



Pre-order Traversal Without Recursion

The following operations are performed to traverse a binary tree in pre-order using a stack:

1. Start with root node and push onto stack.
2. Repeat while the stack is not empty
    a. POP the top element (PTR) from the stack and process the node.
    b. PUSH the right child of PTR onto to stack.
    c. PUSH the left child of PTR onto to stack.

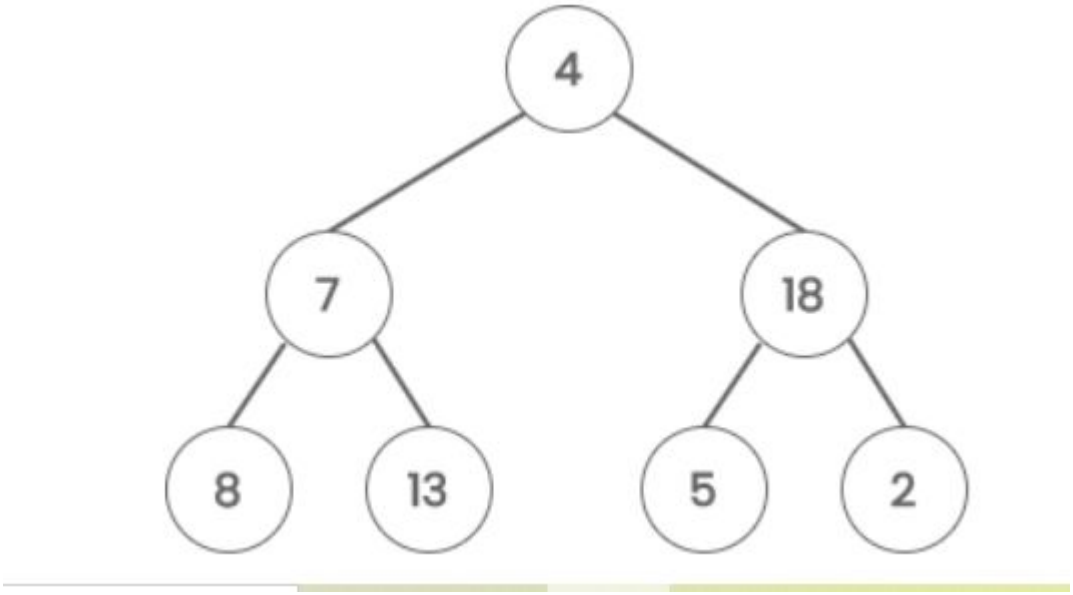


1. Set TOP = 1. STACK[1] = NULL and PTR = ROOT.
2. Repeat Steps 3 to 5 while PTR ≠ NULL:
3. Apply PROCESS to PTR->DATA.
4. If PTR->RIGHT ≠ NULL, then:
   Set TOP = TOP + 1, and STACK[TOP] = PTR->RIGHT.
   [End of If]
5. If PTR->LEFT ≠ NULL, then:
   Set PTR = PTR -> LEFT.
   Else:
   Set PTR = STACK[TOP] and TOP = TOP - 1.
   [End of If]
   [End of Step 2 loop.]
6. Exit.

In-order Traversal Without Recursion

The following operations are performed to traverse a binary tree in in-order using a stack:

1. Start from the root, call it PTR.
2. Push PTR onto stack if PTR is not NULL.
3. Move to left of PTR and repeat step 2.
4. If PTR is NULL and stack is not empty, then Pop element from stack and set as PTR.
5. Process PTR and move to right of PTR , go to step 2.

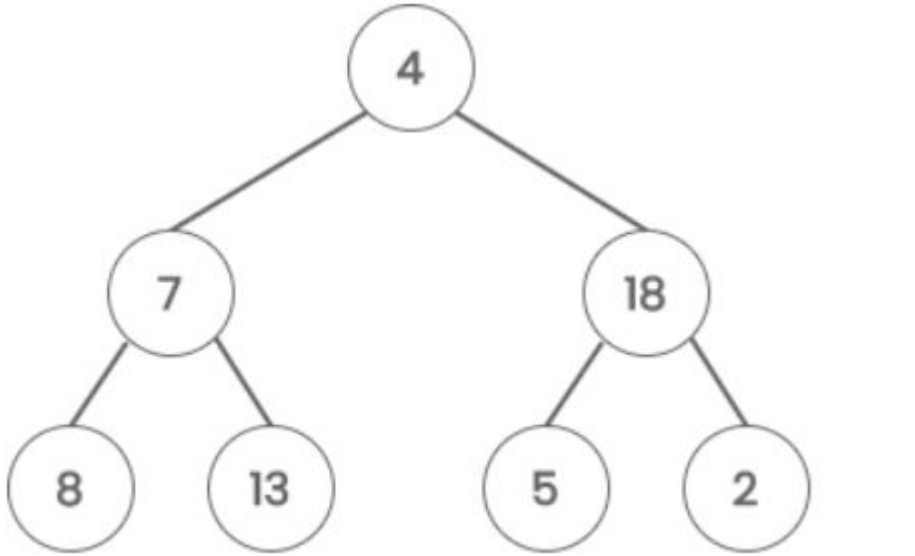


1. Set TOP = 1, STACK[1] = NULL and PTR = ROOT.
2. Repeat while PTR ≠ NULL:
    (a) Set TOP = TOP + 1 and STACK[TOP] = PTR.
    (b) Set PTR = PTR->LEFT.
  [End of loop.]
3. Set PTR = STACK[TOP] and TOP = TOP - 1.
4. Repeat Steps 5 to 7 while PTR ≠ NULL:
5.  Apply PROCESS to PTR->INFO.
6.  If PTR->RIGHT ≠ NULL, then:
      (a) Set PTR = PTR->RIGHT.
      (b) Go to Step 3.
    [End of If]
7. Set PTR = STACK[TOP] and TOP = TOP -1.
  [End of Step 4 loop.]
8. Exit.

Post-order Traversal Without Recursion

The following operations are performed to traverse a binary tree in post-order using a stack:

1. Start from the root, call it PTR.
2. Push PTR onto stack if PTR is not NULL.
3. Move to left of PTR and repeat step 2.
4. If PTR has a right child R, then PUSH -R onto the stack.
5. Pop and process positive element from stack and set as PTR.
6. If a negative node is popped, (PTR = -N), then set PTR = N and go to step 2.



1. Set TOP = 1. STACK[1] = NULL and PTR = ROOT.
2. Repeat Steps 3 to 5 while PTR ≠ NULL:
3. Set TOP = TOP + 1 and STACK[TOP] = PTR.
4. If PTR->RIGHT ≠ NULL. then: [Push on STACK. ]
   Set TOP = TOP + 1 and STACK[TOP] = PTR->-RIGHT.
   [End of If structure.]
5. Set PTR = PTR->LEFT. [Updates pointer PTR.]
   [End of Step 2 loop.]
6. Set PTR = STACK[TOP] and TOP = TOP - 1.
7. Repeat while PTR > 0:
   (a) Apply PROCESS to PTR->INFO.
   (b) Set PTR = STACK[TOP] and TOP = TOP - 1.
   [End of loop.]
8. If PTR < 0, then:
   (a) Set PTR = -PTR.
   (b) Go to Step 2.
   [End of If structure]
9. Exit.

Two trees are equal or identical when they have same data and arrangement of data is also same.
To identify if two trees are identical, we need to traverse both trees simultaneously, and while traversing we need to compare data and children of the trees.

sameTree(tree1, tree2)

1. If both trees are empty then return 1.
2. Else If both trees are non - empty
   (a) Check data of the root nodes (tree1->data == tree2->data)
   (b) Check left subtrees recursively i.e.,
   call sameTree( tree1->left_subtree, tree2->left_subtree)
   (c) Check right subtrees recursively i.e.,
   call sameTree( tree1->right_subtree, tree2->right_subtree)
   (d) If a,b and c are true then return 1.
3 Else return 0 (one is empty and other is not)

Given a binary tree, we have to create a clone of given binary tree. We will traverse the binary tree using pre order traversal and create a clone of every node. The problem of creating a duplicate tree can be broken down to sub problems of creating duplicate sub trees.

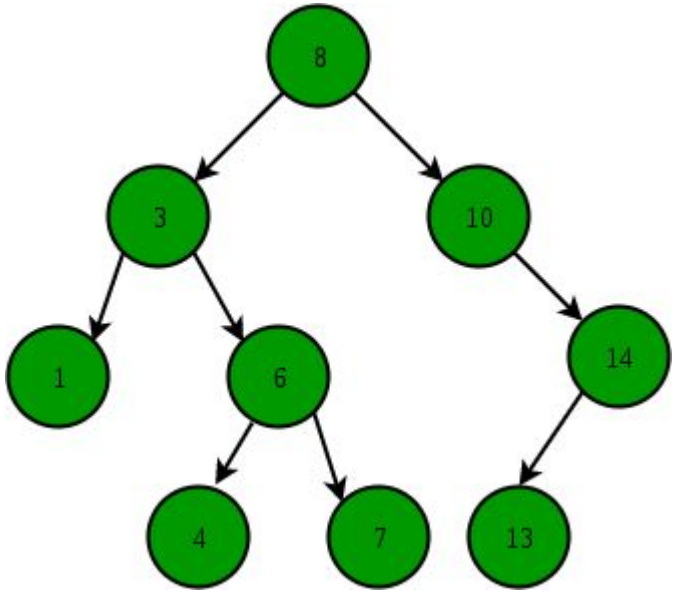
Algorithm to create a duplicate binary tree

Let "root" be the root node of binary tree.

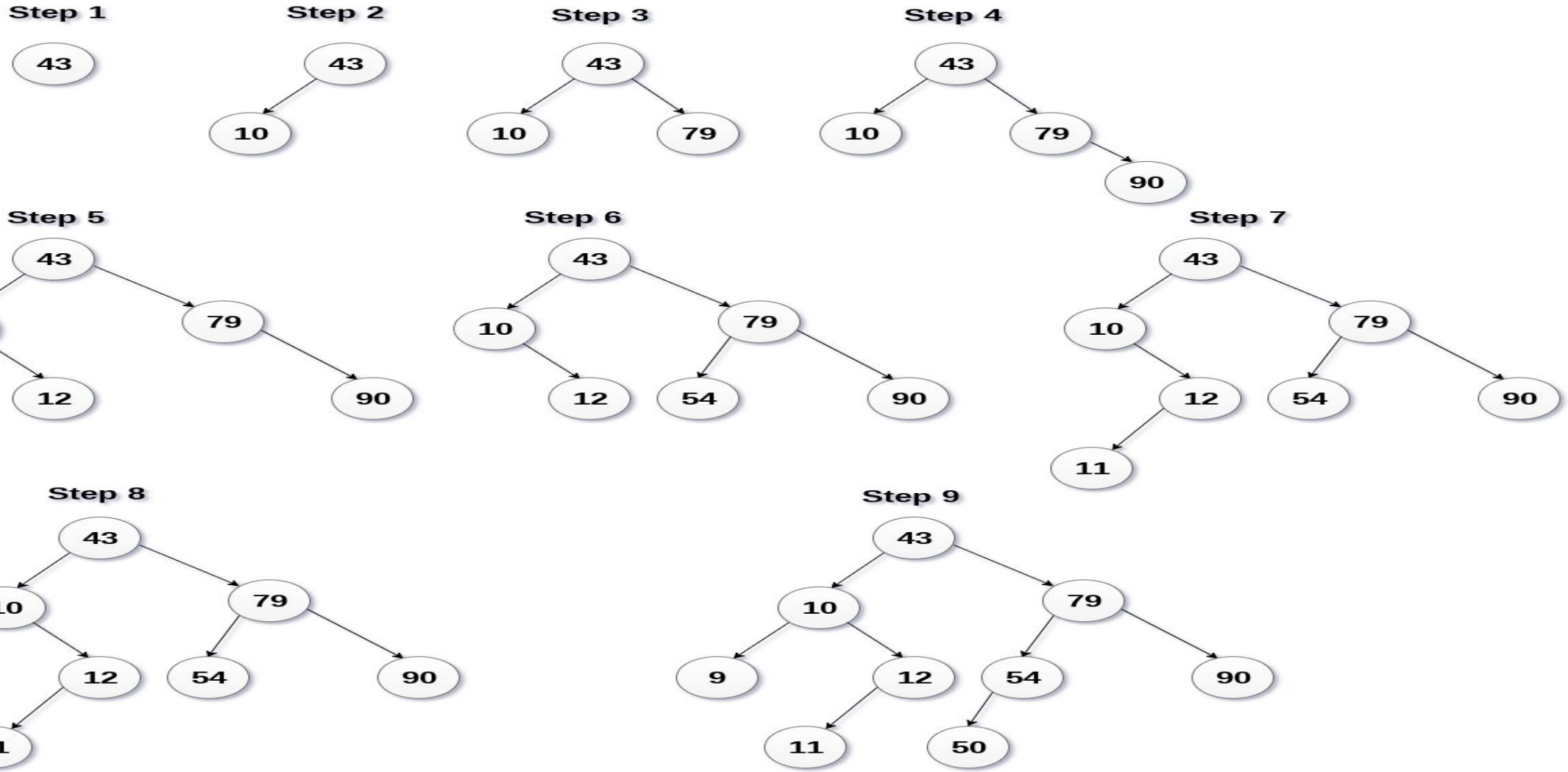
1. If root is equal to NULL, then return NULL.
2. Create a new node and copy data of root node into new node.
3. Recursively, create clone of left sub tree and make it left sub tree of new node.
4. Recursively, create clone of right sub tree and make it right sub tree of new node.
5. Return new node.

**Binary Search Tree** is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



## Create BST -43, 10, 79, 90, 12, 54, 11, 9, 50




Binary search Tree Creation

## Search Operation in BST

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

**Step 1 -** Read the search element from the user.

**Step 2 -** Compare the search element with the value of root node in the tree.

**Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function

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

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

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

**Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node

**Step 8 -** If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

**Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

# Search Operation in BST

Search (ROOT, ITEM)

**Step 1:** IF ROOT -> DATA = ITEM OR ROOT = NULL

Return ROOT

ELSE

IF ROOT < ROOT -> DATA

Return search(ROOT -> LEFT, ITEM)

ELSE

Return search(ROOT -> RIGHT,ITEM)

[END OF IF]

[END OF IF]

**Step 2:** END

## Insert Operation in BST

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must node violate the property of binary search tree at each value.

Step 1 - Allocate the memory for tree.

Step 2 - Set the data part to the value and set the left and right pointer of tree, point to NULL.

Step 3 - If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.

Step 4 - Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.

Step 5 - If this is false, then perform this operation recursively with the right sub-tree of the root.

## Insert Operation in BST

Insert (TREE, ITEM)

Step 1: IF TREE = NULL

Allocate memory for TREE

SET TREE -> DATA = ITEM

SET TREE -> LEFT = TREE -> RIGHT = NULL

ELSE

IF ITEM < TREE -> DATA

Insert(TREE -> LEFT, ITEM)

ELSE

Insert(TREE -> RIGHT, ITEM)

[END OF IF]

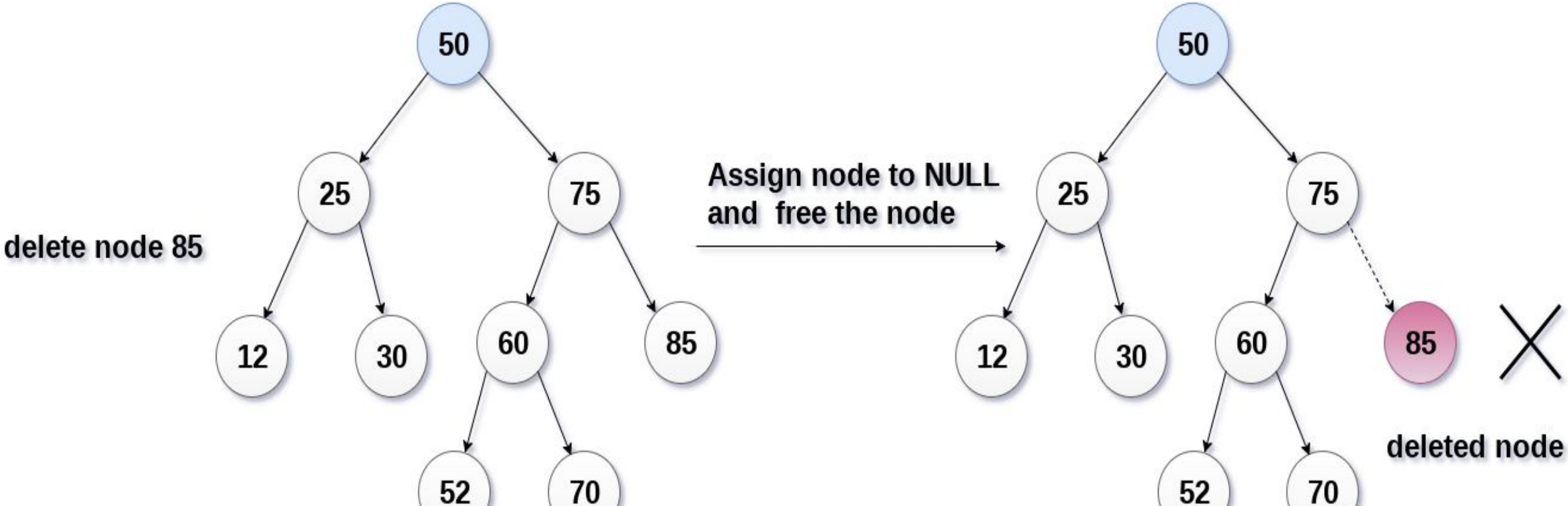
[END OF IF]

Step 2: END

- **Delete operation in BST**

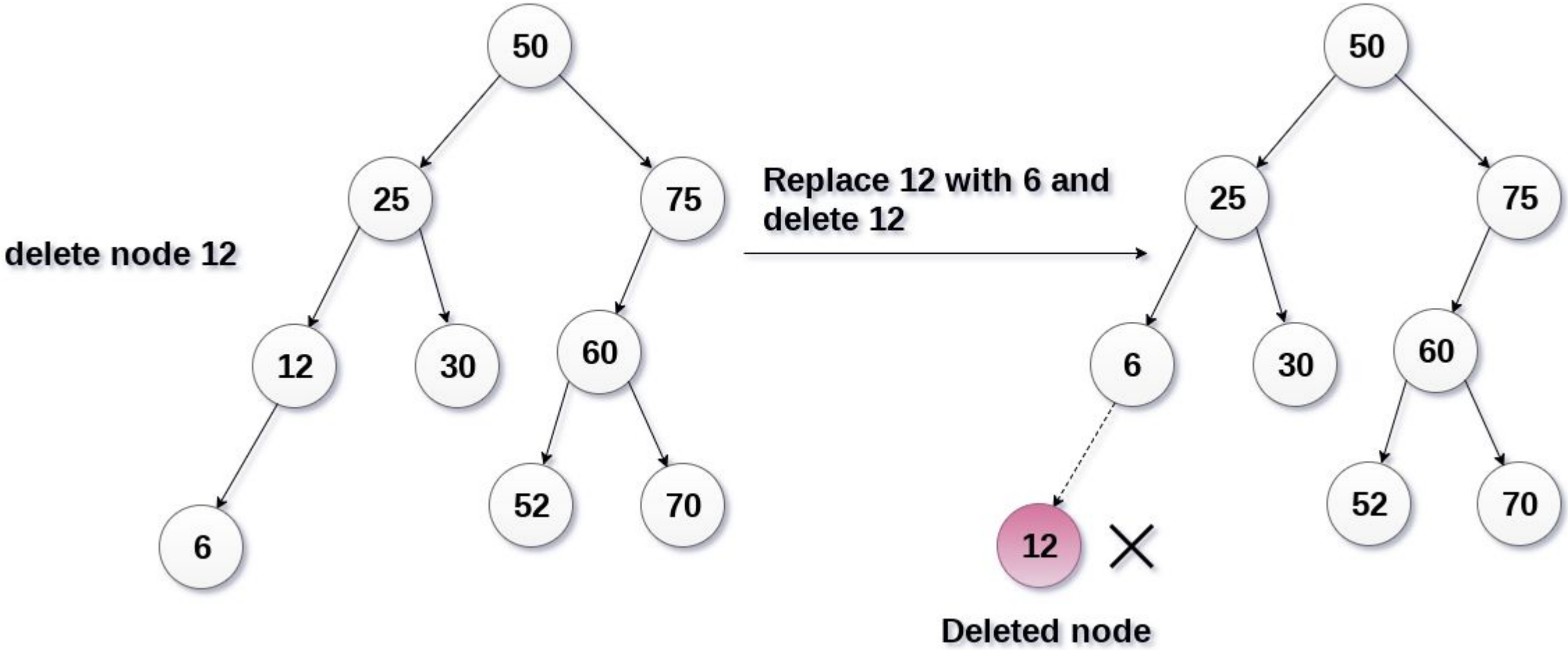
Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

- The node to be deleted is a leaf node: It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.



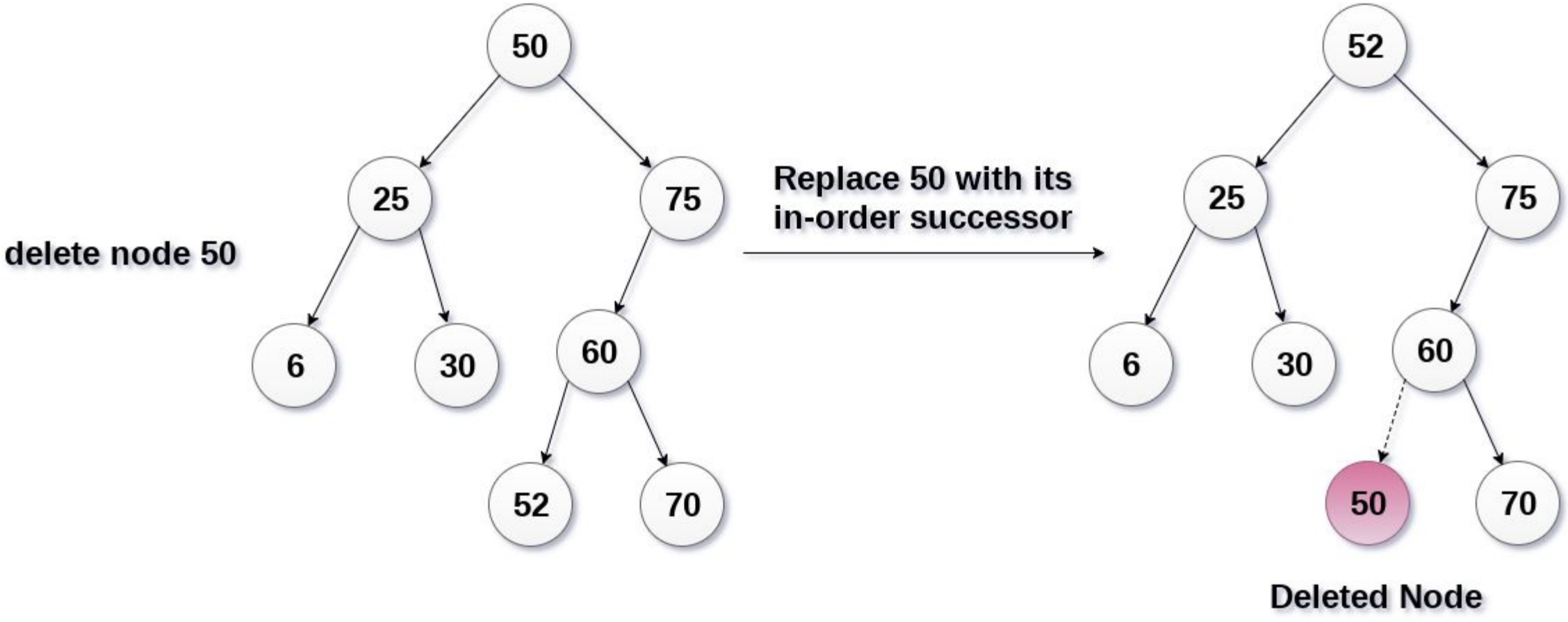
- **Delete operation in BST**

The node to be deleted has only one child: In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.



- **Delete operation in BST**

The node to be deleted has two children: It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.



- **Delete operation in BST**

You need to delete the node with value item and then return the root of the modified tree. First, we need to find the node to be deleted and then replace it by the appropriate node if needed.

Check if the root is NULL, if it is, just return the root itself. It's an empty tree!

If root.val < item, recurse the right subtree.

If root.val > item, recurse the left subtree.

If both above conditions above false, this means root.val == item.

Now we first need to check how many child did root have.

CASE 1: No Child → Just delete root or deallocate space occupied by it

CASE 2: One Child →Replace root by its child

CASE 3: Two Children

Find the inorder successor of the root (Its the smallest element of its right subtree). Let's call it new_root.

Replace root by its inorder successor

Now recurse to the right subtree and delete new_root.

9. Return the root.

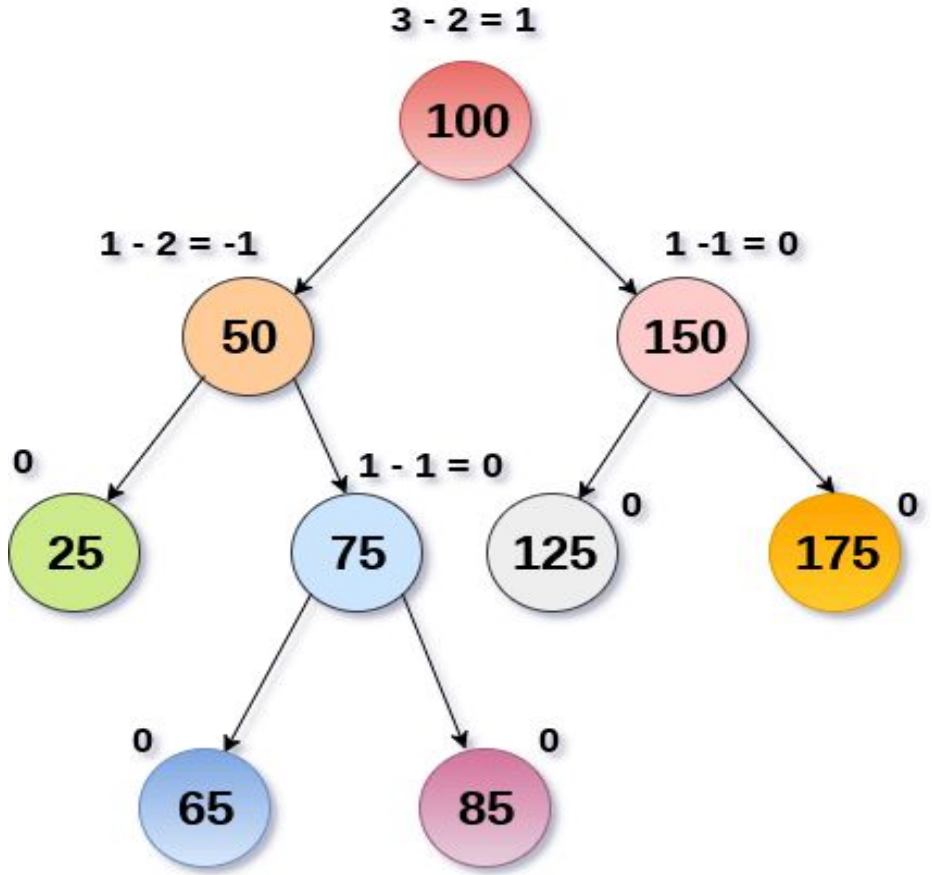
```
TreeNode delete_element(TreeNode root, int item)
 {
  if ( root == NULL )
     return root
     if ( root.val < item)
     root.right = delete_element(root.right, item)
    else if ( root.val > item )
     root.left = delete_element(root.left, item)
    else if ( root.val == item )
    { // No child
     if ( root.left == NULL and root.right == NULL )
        delete root
     // Only one child
     else if ( root.left == NULL or root.right == NULL )
     {
        if(root.left != NULL)
        {
           TreeNode new_root = root.left
           delete root
           return new_root
        }
        else
        {
           TreeNode new_root = root.right
           delete root
           return new_root
        }
     }
// Both left and right child exist
        else
        {
           TreeNode new_root = find_inorder_successor(root.right)
           root.val = new_root.val
           root.right = delete_element(root.right, new_rot.val)
        }
    }
    return root
}

TreeNode find_inorder_successor(TreeNode root)
{
    TreeNode curr = root
    while ( curr.left != NULL )
        curr = curr.left

    return curr
}
```

# AVL Tree

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honor of its inventors.
- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.
- Balance Factor (k) = height (left(k)) - height (right(k))
- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that 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.

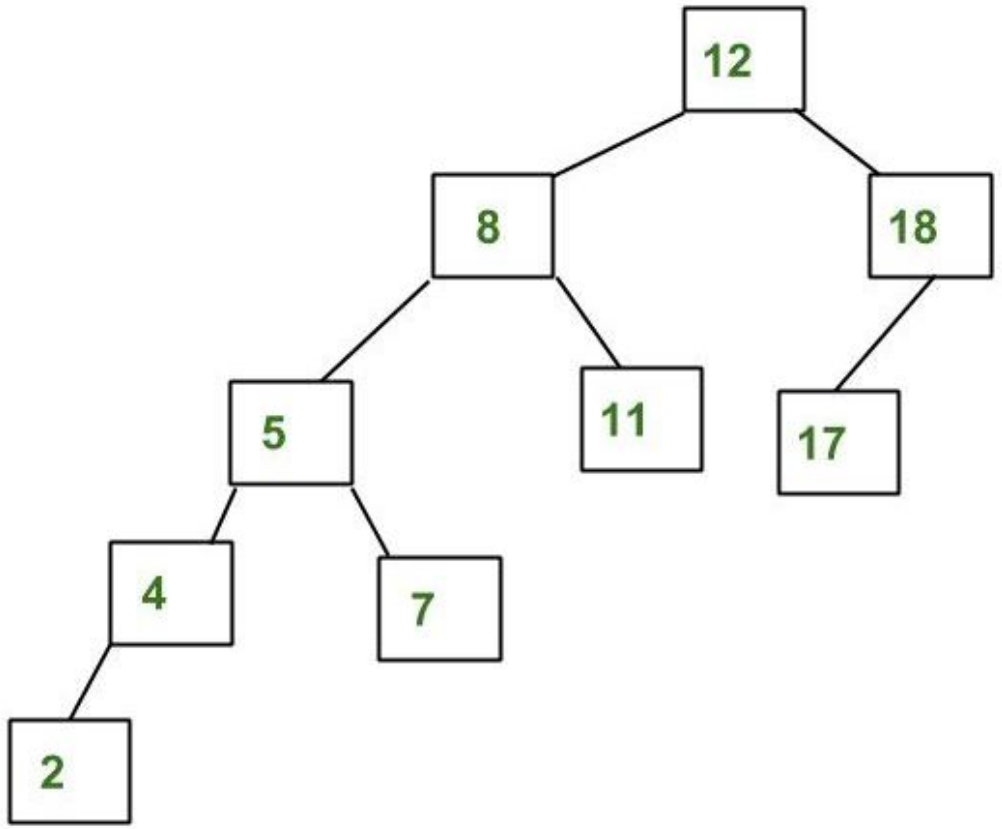
# AVL Tree

- **Definition**: An empty binary tree is an AVL tree. A non empty binary tree T is an AVL tree iff given T(L) and T(R) to be the left and right subtrees of T and h(T(L)) and h(T(R)) to be the heights of subtrees T(L) and T(R) respectively, T(L) and T(R) are AVL trees and | h(T(L)) – h(T(R)) | <= 1.
- h(T(L)) – h(T(R)) is known as the balance factor (BF) and for an AVL tree the balance factor of a node can be either 0, 1, or -1.




AVL Tree

# Not an 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)**. However, 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.

Operations on AVL tree

- 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 in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
- Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

# AVL Rotations

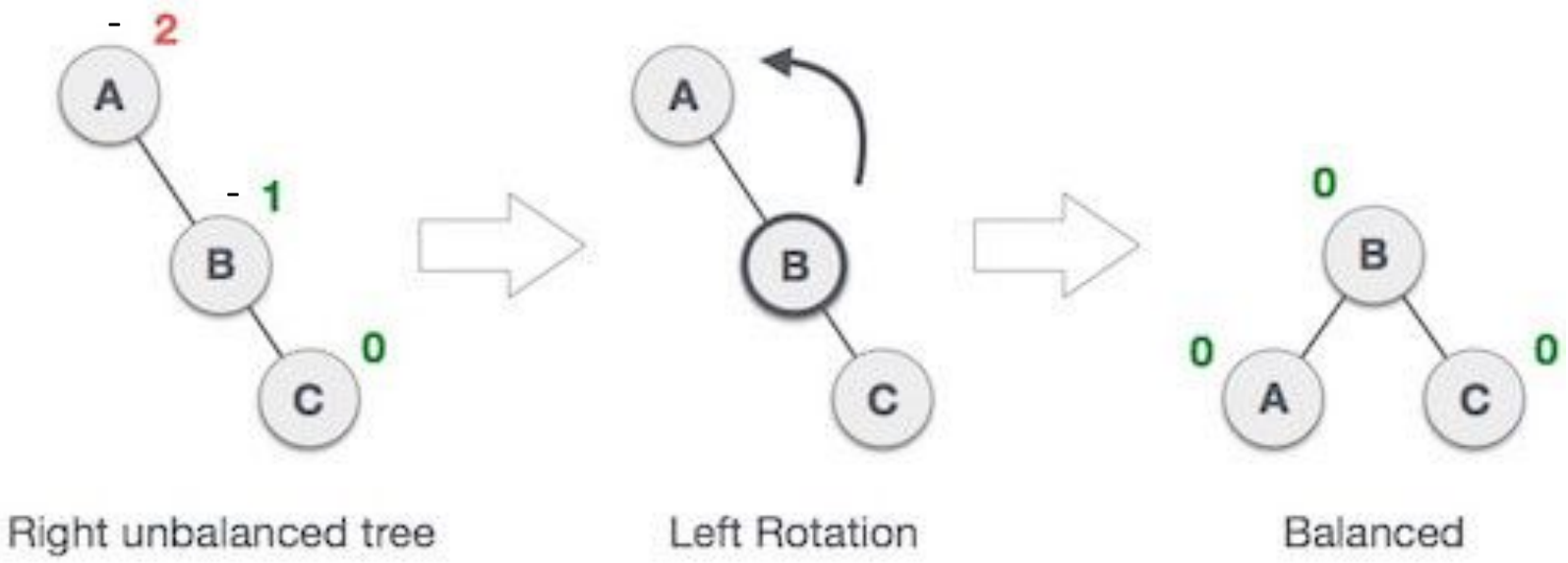
- We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

1. LL rotation: Inserted node is in the left subtree of left subtree of A
2. RR rotation : Inserted node is in the right subtree of right subtree of A
3. LR rotation : Inserted node is in the right subtree of left subtree of A
4. RL rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

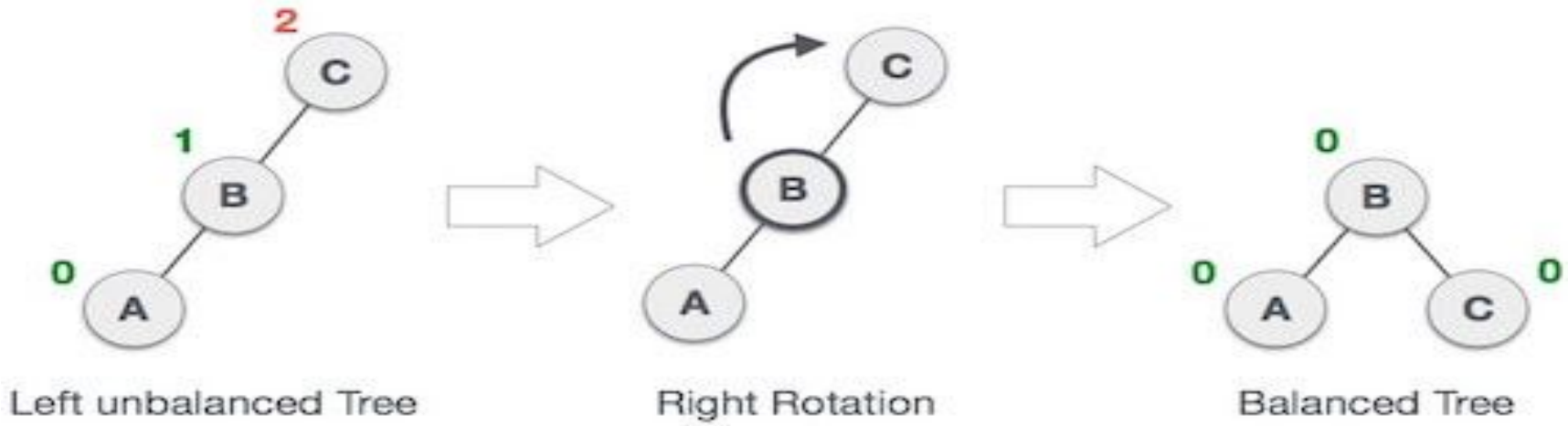
The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2,

- RR Rotation
- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2




- In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

- LL Rotation
- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



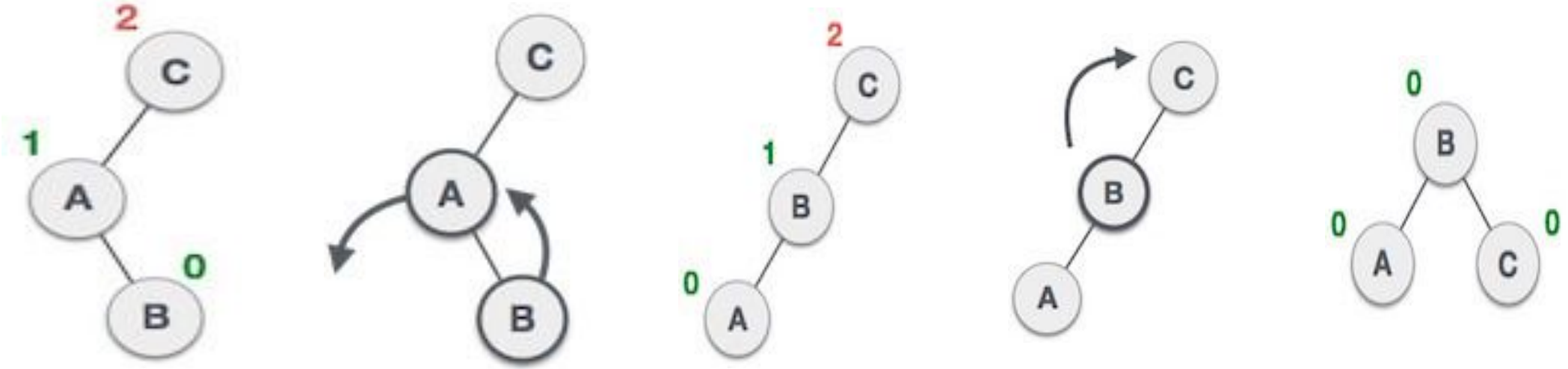

- In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

- LR Rotation
- Double rotations are bit tougher than single rotation. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C.

As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.

Now we perform LL clockwise rotation on full tree, i.e. on node C.
node **C** has now become the right subtree of node B, A is left subtree of B



- R L rotation
- R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.