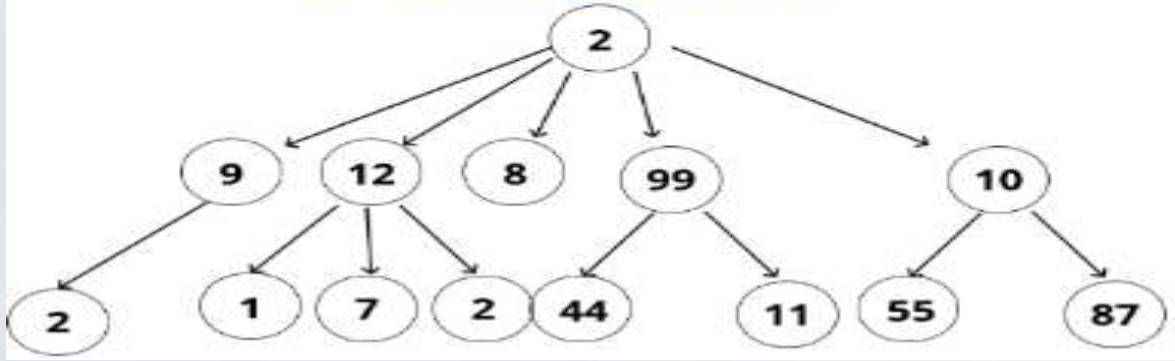
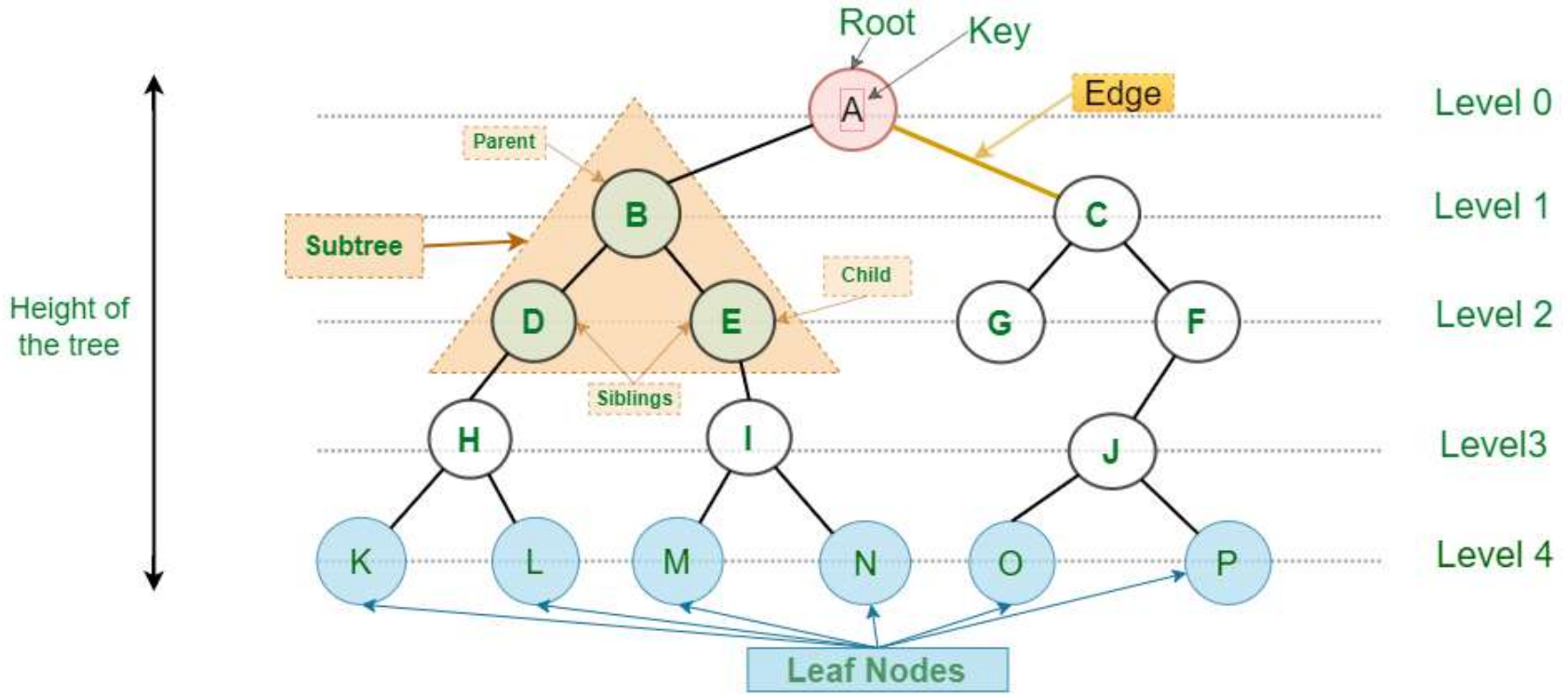
TREES



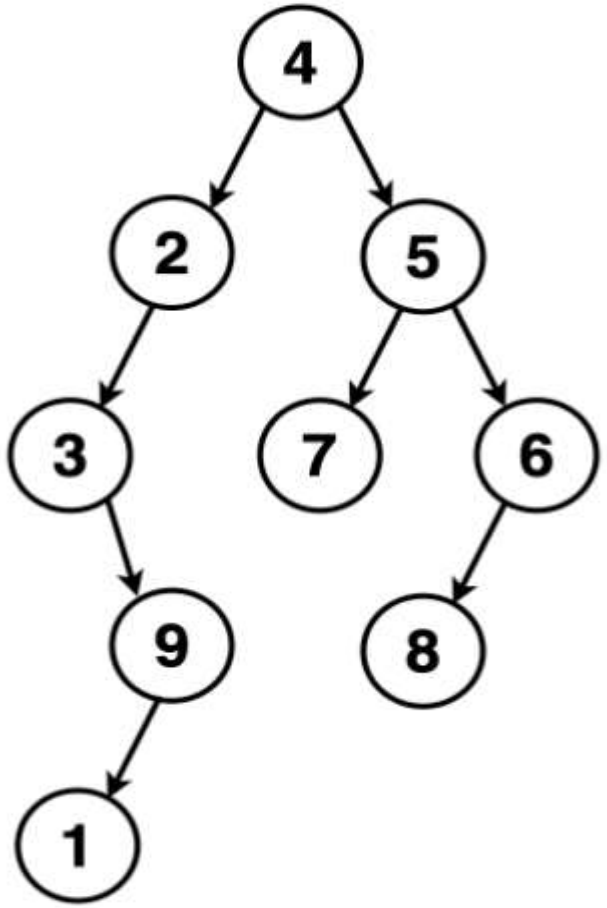
DATA STRUCTURES

Tree – Level Order Traversal



A B C D E G F H I J K L M N O P

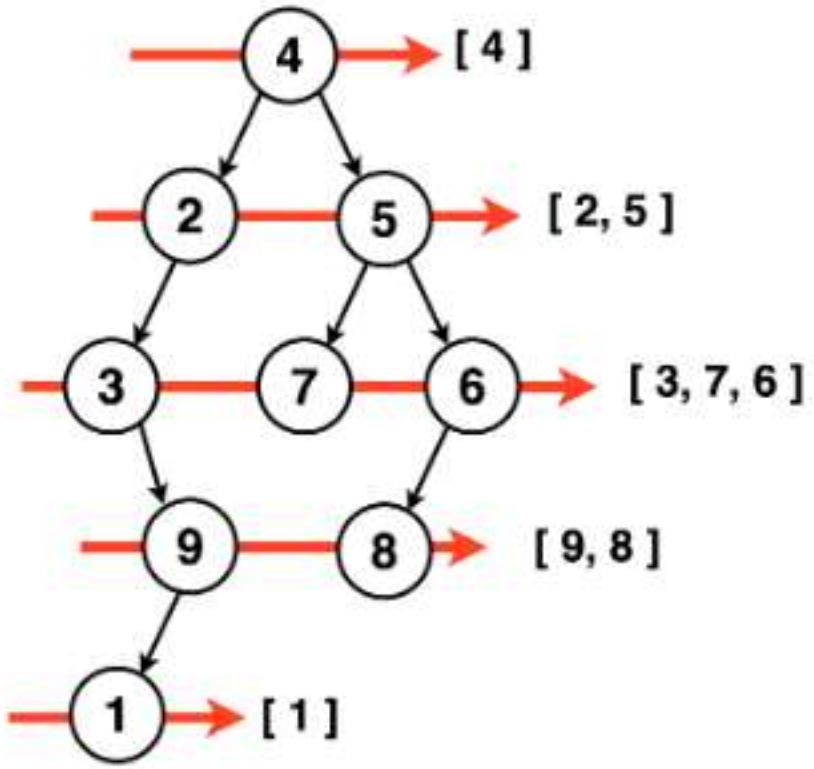
Tree – Level Order Traversal



1. Find **height** of tree.
2. for each level, run a recursive function by maintaining current height.
3. if the level of a node matches, print that node.

Output:[[4],[2, 5],[3, 7, 6],[9, 8],[1]]

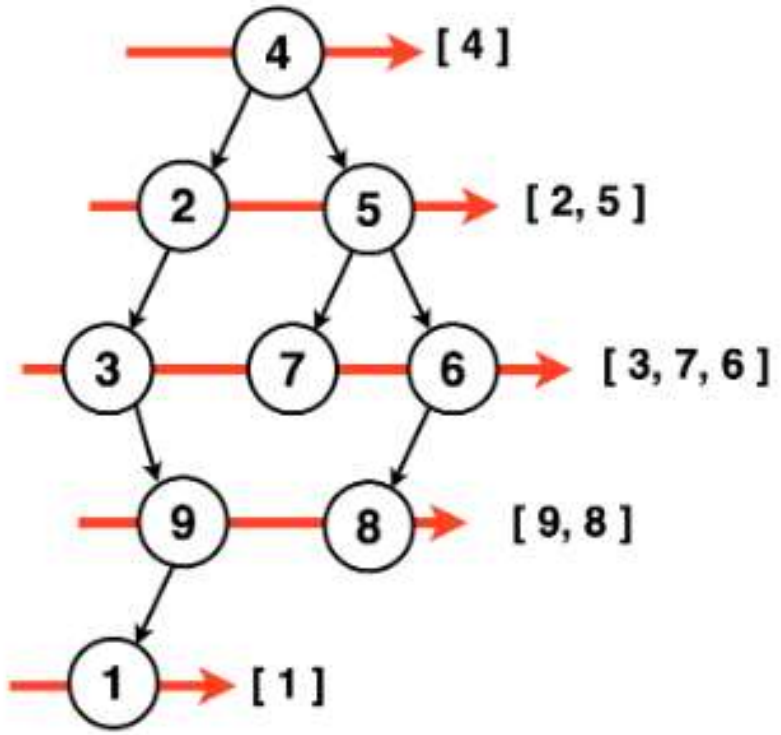
Tree – Level Order Traversal



level order traversal : [[4], [2, 5], [3, 7, 6], [9, 8], [1]]

```
int height(struct node* node)
    {
     if (node == NULL) return 0;
  else {
int lheight = height(node->left);
int rheight = height(node->right);
if (lheight > rheight)
     return (lheight + 1);
 else
return (rheight + 1);
        }
   }
```

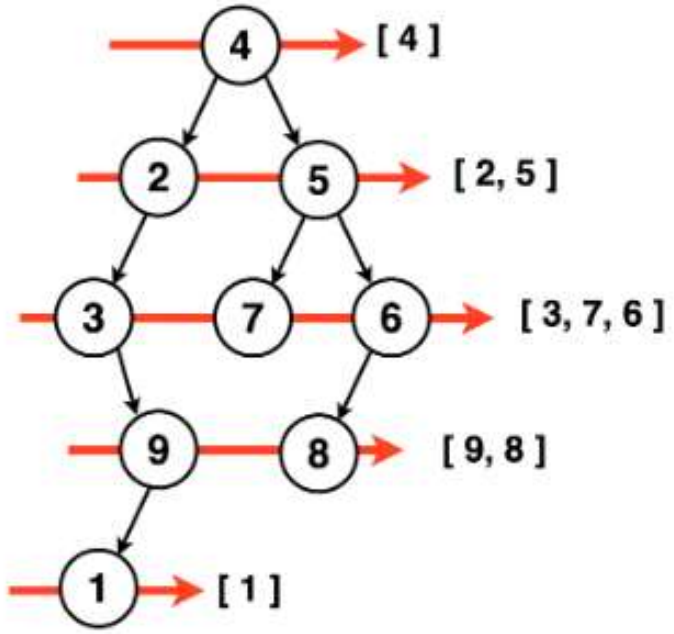
Tree – Level Order Traversal



level order traversal : [[4], [2, 5], [3, 7, 6], [9, 8], [1]]

```
void printLevelOrder(struct node* root)
  {
      int h = height(root);
      int i;
        for (i = 1; i <= h; i++)
            printCurrentLevel(root, i);
  }
```

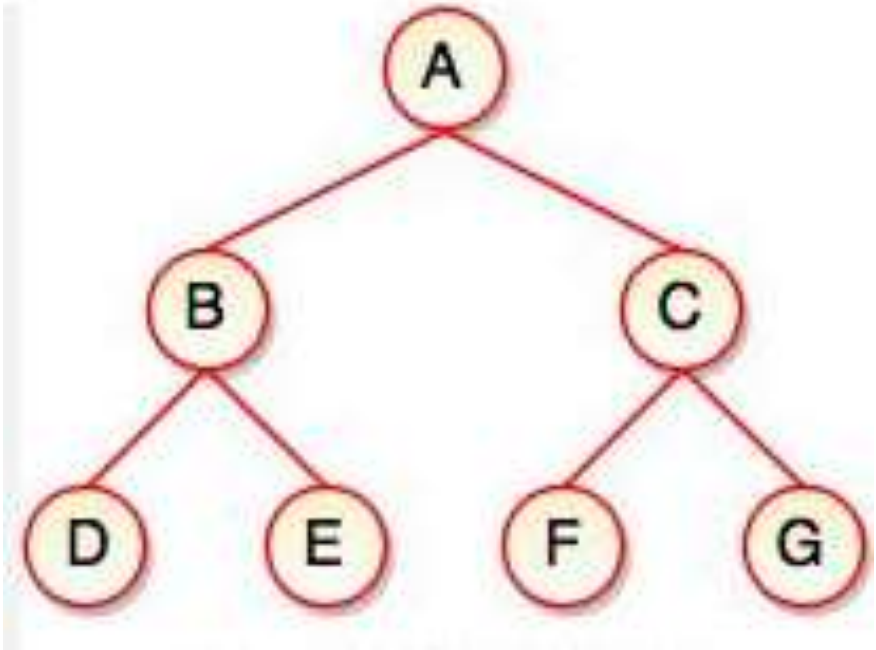
Tree – Level Order Traversal

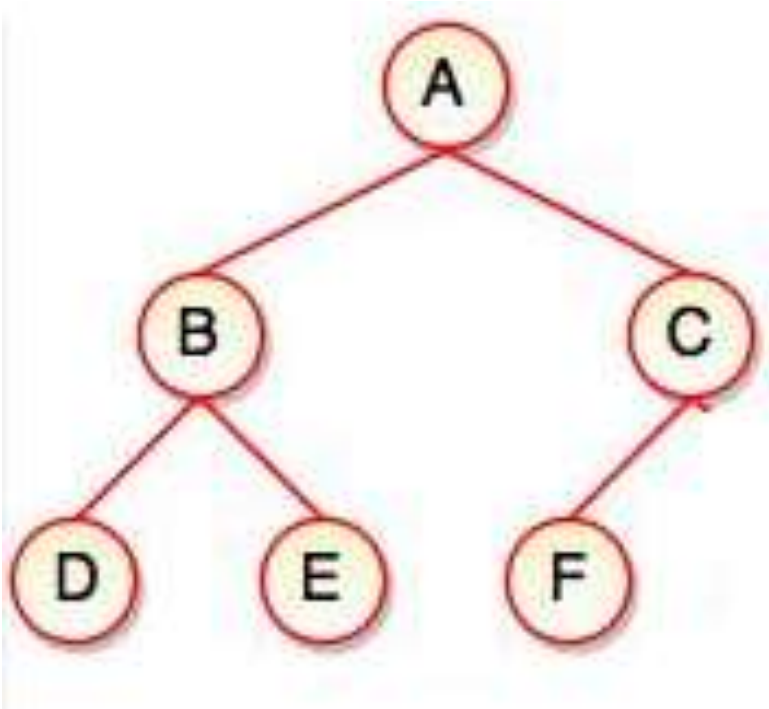


level order traversal : [[4], [2, 5], [3, 7, 6], [9, 8], [1]]

```
void printCurrentLevel(struct node* root, int level)
    {
       if (root == NULL)
              return;
       if (level == 1)
            printf("%d ", root->data);
       else if (level > 1)
         { printCurrentLevel(root->left, level - 1);
            printCurrentLevel(root->right, level - 1);
         }
    }
```

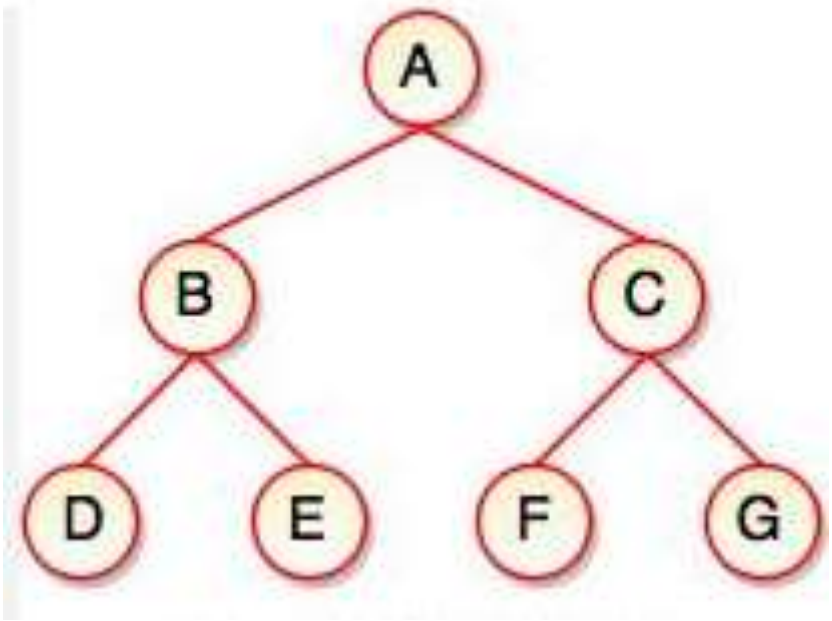
Tree – Full Binary Tree





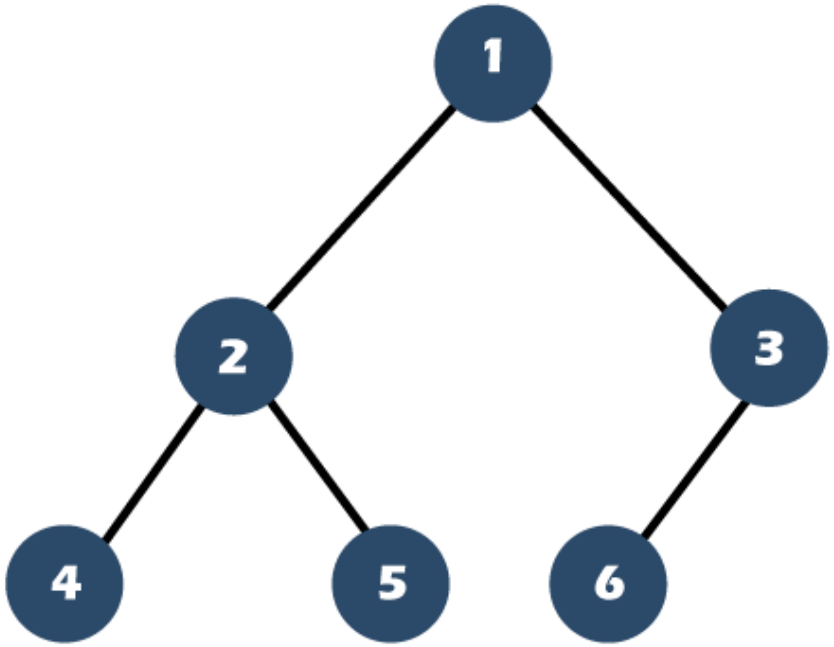
A full binary tree is a binary tree where every node has either zero or two children. It's also known as a proper, plane, or strict binary tree.

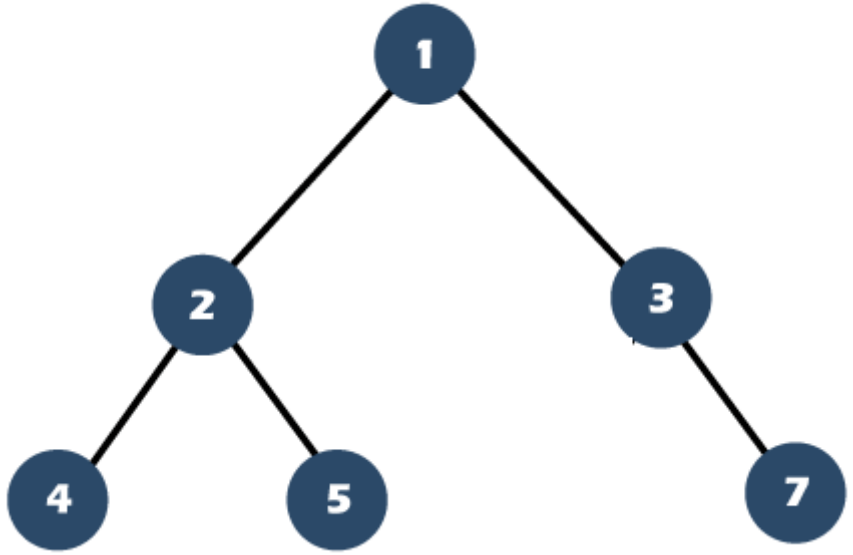
Tree – Full Binary Tree




```
bool isFullBinaryTree(Node* root) {
// Empty tree is considered full
        if (root == nullptr) {
            return true;
        }
// If a node has only one child, it's not full
  if ((root->left == nullptr && root->right != nullptr) ||
    (root->left != nullptr && root->right == nullptr)) {
            return false;
        }
// Recursively check left and right subtrees
    return isFullBinaryTree(root->left) &&
           isFullBinaryTree(root->right);
    }
```

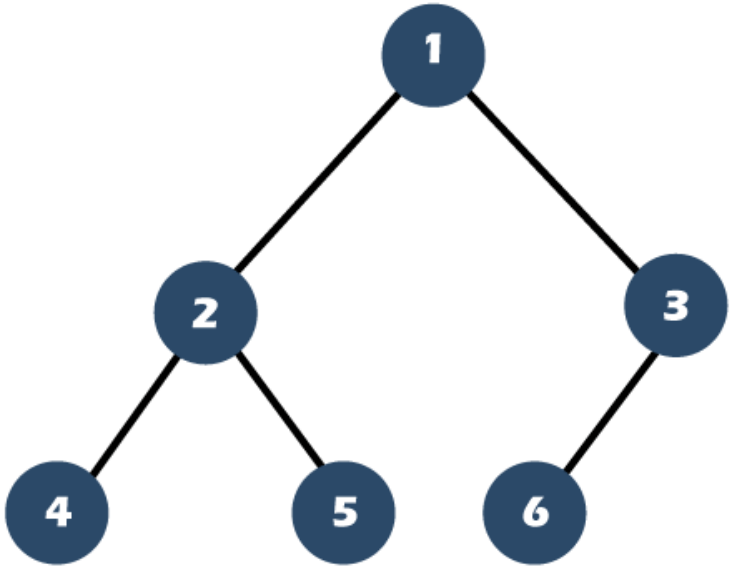
Tree – Complete Tree





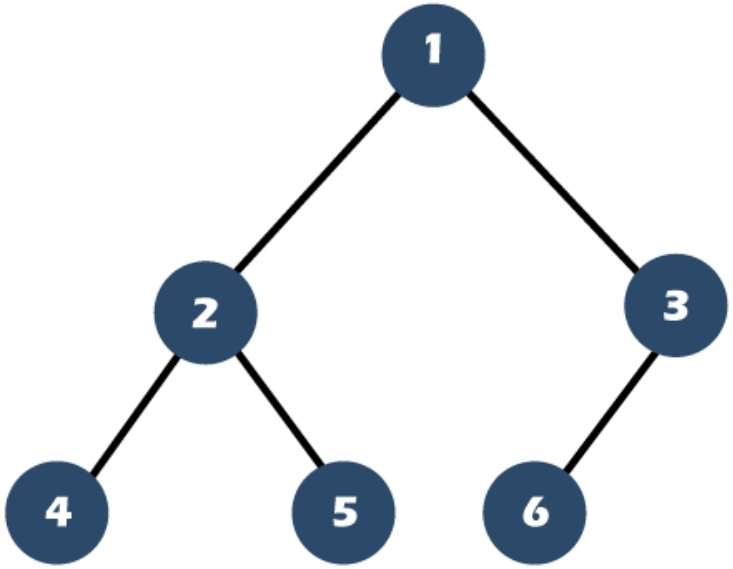
A binary tree is said to be a complete binary tree when all the levels are completely filled except the last level, which is filled from the left.

Tree – Complete Tree



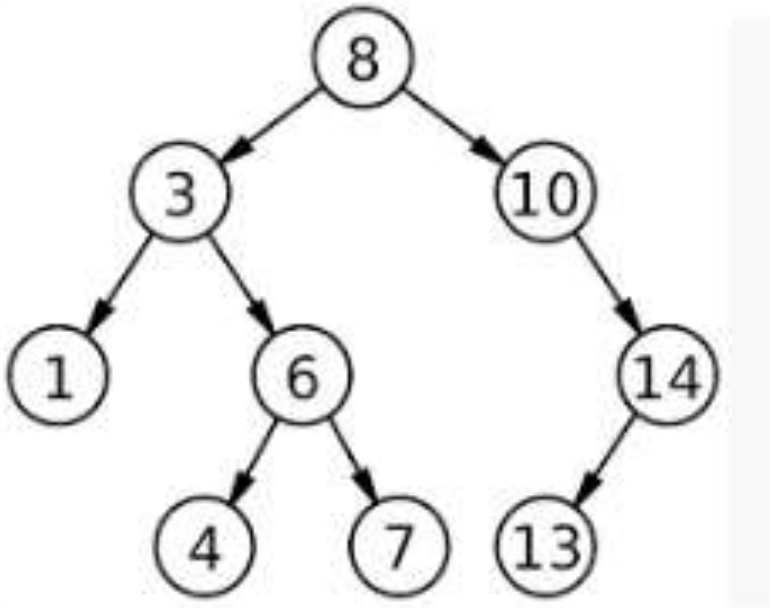

```
bool isCompleteBT(Node *root) {
    if (!root) return true;

        queue<Node *> q;
        q.push(root);
         bool flag = false; // Indicates if a non-full node

    while (!q.empty( )) {
     Node *curr = q.front( );
     q.pop( );
```

Tree – Complete Tree



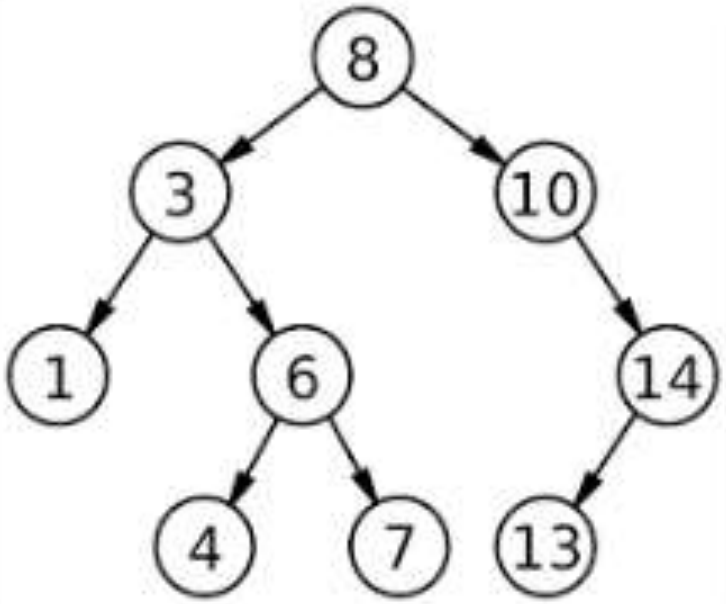

```
if (curr->left) {
// If we've already encountered a non-full node, it's not complete
      if (flag) return false;
       q.push(curr->left);
          }
 else
        flag = true; // Mark that we've seen a non-full node
if (curr->right) {
// If we've already encountered a non-full node, it's not complete
        if (flag) return false;
         q.push(curr->right);
         }
 else
       flag = true; // Mark that we've seen a non-full node
     }
   return true;
 }
```

Tree – BST Tree



A Binary Search Tree is a data structure storing data in a sorted manner. Each node in a Binary Search Tree has at most two children, a left child and a right child, with the left child containing values less than the parent node and the right child containing values greater than the parent node. This hierarchical structure allows for efficient searching, insertion, and deletion operations on the data stored in the tree.

Tree – BST Tree



isBSTUtil(root, minVal, maxVal):

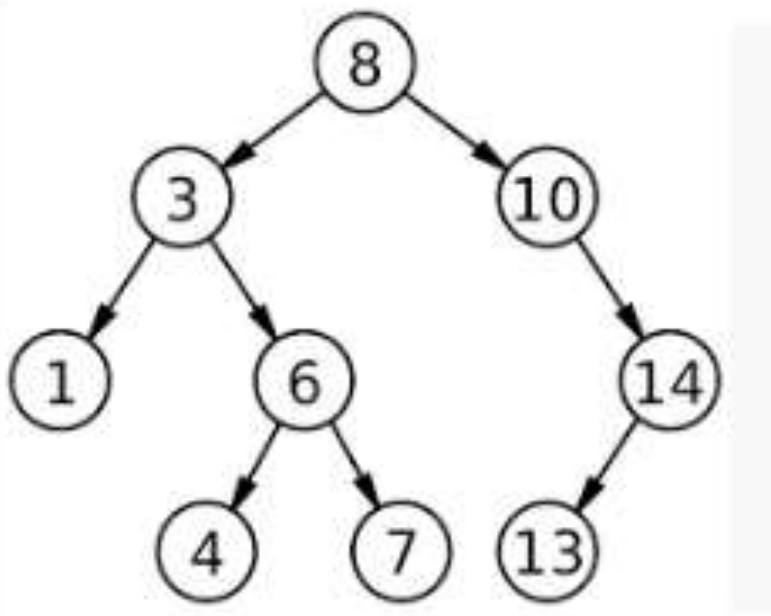
Recursive function checks if the current node is a BST, given the allowed minimum and maximum values for its data.

If the current node is null, it's considered a BST (base case).

If the node's data is not within the allowed range, it's not a BST.

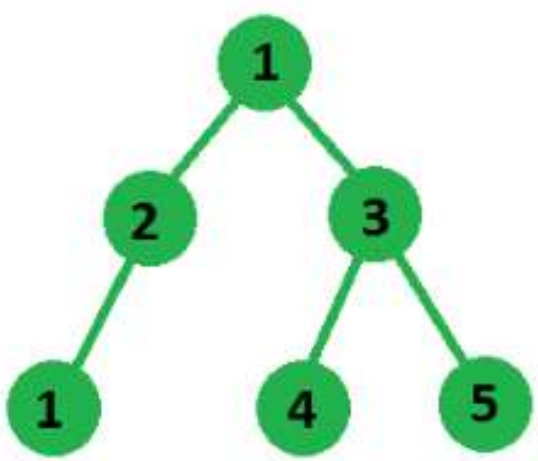
Recursively check the left subtree with an updated maximum value (node's data - 1) and the right subtree with an updated minimum value (node's data + 1).

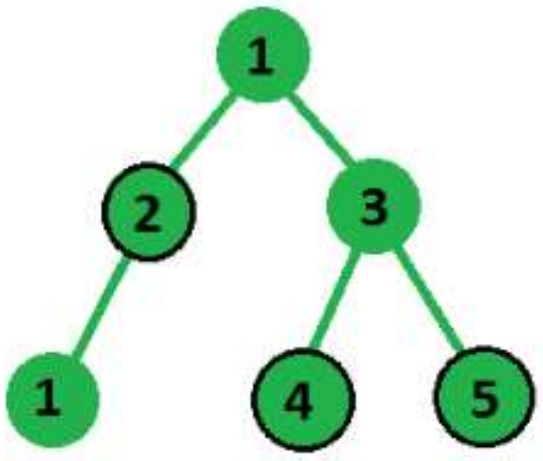
Tree – BST Tree




```
bool isBSTUtil(Node *root, int minVal, int maxVal) {
        if (root == nullptr)
                return true;

    if (root->data < minVal || root->data > maxVal)
                return false;

return isBSTUtil(root->left, minVal, root->data - 1) &&
            isBSTUtil(root->right, root->data + 1, maxVal);
        }
```

Tree – Maximum Sum of Alternate Nodes





Max Sum = 2 + 4 + 5 = 11

```
int maxSumAlternateNodes(Node* root) {
        if (!root) return 0;
            return maxSum(root);
}
```

Tree – Maximum Sum of Alternate Nodes

```
int maxSum(Node* root) {
        if (!root) return 0;
// Calculate sum including current node and its grandchildren
int include = root->data;
            if (root->left) {
                include += maxSum(root->left->left);
                include += maxSum(root->left->right);
              }
        if (root->right) {
                include += maxSum(root->right->left);
                include += maxSum(root->right->right);
              }
// Calculate sum excluding current node
        int exclude = maxSum(root->left) + maxSum(root->right);
    return max(include, exclude);
  }
```

THANK YOU