1) **Maximum Depth of Binary Tree**:
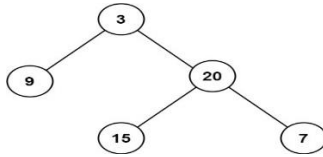
Given the root of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

Input: root = [3,9,20,null,null,15,7]

Input: root = [3,9,20,null,null,15,7]

Output: 3

Example 2:

Input: root = [1,null,2]

Output: 2

```
 int maxDepth(TreeNode* root)

  {

     //Base case:

     if(root==NULL)

         return 0; //return 0 height if root is null

     int ltree = maxDepth(root->left);

     int rtree = maxDepth(root->right);

     return 1+ max(ltree, rtree);

  }
```

- Time complexity: *O(n)*

- Space complexity: *O(n)*

----

2) **Binary Tree Preorder Traversal**: Given the root of a binary tree, return *the preorder traversal of its nodes' values*.

```
      1
     / \
    2   3
   / \
  4   5
```
Output: 1 2 4 5 3

Recursive traversal:
```cpp
vector<int> Preorder(TreeNode* root)
{
   vector<int>ans;
   if (root == NULL)
     return ans;

   // Deal with the node
   ans.push_back(root->val);

   // Recur on left subtree
   Preorder(root->left);

   // Recur on right subtree
   Preorder(root->right);
}
```

//Iterative way of traversal
```cpp
//Time complexity:O(n)
//Space complexity:O(n)
vector<int> preorderTraversal(TreeNode* root)
   {
     vector<int>ans;
     // Base Case
   if (root == NULL)
     return ans;

   // Create an empty stack and push root to it
   stack<TreeNode*> nodeStack;
   nodeStack.push(root);

   /* Pop all items one by one. Do following for every popped item
      a) print it
      b) push its right child
      c) push its left child
   Note that right child is pushed first so that left is processed first */
   while (!nodeStack.empty())
```

```
  {
     // Pop the top item from stack and print it
     root = nodeStack.top();
     ans.push_back(root->val);
     nodeStack.pop();

     // Push right and left children of the popped node to stack
     if (root->right)
        nodeStack.push(root->right);
     if (root->left)
        nodeStack.push(root->left);
  }
  return ans;
 }
```
Time complexity: O(n)
Space complexity: O(n)

----

3) **Binary Tree Inorder Traversal**: Given the root of a binary tree, return *the inorder traversal of its nodes' values*.

```
        1
       / \
      2   3
     / \
    4   5
```

O/P: 4 2 5 1 3

Recursive Approach:

```
vector<int> inOrder(TreeNode* root)
{
   vector<int>ans;
   if (root == NULL)
      return ans;

  inOrderTrav(curr -> left);
  inOrder.push_back(curr -> data);
  inOrderTrav(curr -> right);
}
```
Time complexity: O(n)
Space complexity: O(n)

```cpp
//Iterative function for inorder tree traversal
 vector<int> inorderTraversal(TreeNode* root)
 {
   stack<Node*> s;
    Node* curr = root;

   while (curr != NULL || !s.empty())
{

     // Reach the left most Node of the
     // curr Node
     while (curr != NULL) {

       // Place pointer to a tree node on
       // the stack before traversing
       // the node's left subtree
       s.push(curr);
       curr = curr->left;
     }

     // Current must be NULL at this point
     curr = s.top();
     s.pop();

    ans.push_back(curr->val);

     // we have visited the node and its
     // left subtree.  Now, it's right
     // subtree's turn
     curr = curr->right;

   }
return ans;
}
 Time complexity: O(n)
Space complexity: O(1)
```
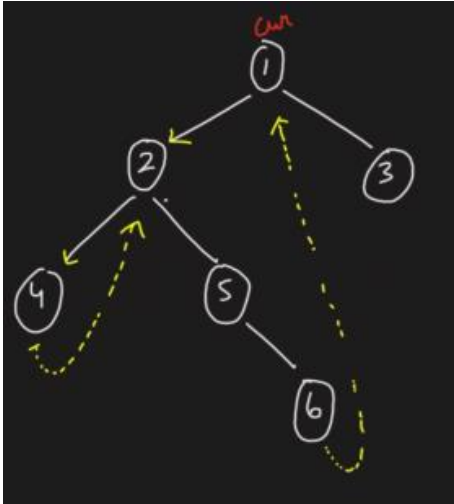

 Morris inorder traversal:

Approach:

- Initialize current as root
- While current is not NULL
- If the current does not have left child
    - Print current's data
    - Go to the right, i.e., current = current->right
- Else, find rightmost node in current left subtree OR node whose right child == current.
- If we found right child == current
    - Update the right child as NULL of that node whose right child is current
    - Print current's data
    - Go to the right, i.e. current = current->right
- Else
    - Make current as the right child of that rightmost node we found; and
    - Go to this left child, i.e., current = current->left

```cpp
vector<int> MorrisTraversal(TreeNode* root)
{
    vector<int>ans;
    if (root == NULL)
        return;

    TreeNode* current = root;
    while (current != NULL)
    {

        if (current->left == NULL)
        {
```

```
        ans.push_back(root->val);
        current = current->right;
     }
     else
    {

        /* Find the inorder predecessor of current */
        pre = current->left;
        while (pre->right != NULL && pre->right != current)
           pre = pre->right;

        /* Make current as the right child of its
           inorder predecessor */
        if (pre->right == NULL) {
           pre->right = current;
           current = current->left;
        }

        /* Revert the changes made in the 'if' part to
           restore the original tree i.e., fix the right
           child of predecessor */
        else {
           pre->right = NULL;
           ans.push_back(root->val);
           current = current->right;
        } /* End of if condition pre->right == NULL */
     } /* End of if condition current->left == NULL*/
  } /* End of while */
        return ans;
}
```
----

4) **Binary Tree Postorder Traversal**: Given the root of a binary tree, return *the postorder traversal of its nodes' values*.

```
      1
     / \
    2   3
   / \
  4   5
```

Using Recursion:
```
void printPostorder(struct Node* node)
{
   if (node == NULL)
```

```cpp
        return;

    // First recur on left subtree
    printPostorder(node->left);

    // Then recur on right subtree
    printPostorder(node->right);

    // Now deal with the node
    cout << node->data << " ";
}
```
Using Iterative method:
```cpp
vector<int> postorderTraversal(TreeNode* root) {
    vector<int> result;
    if (!root) return result;

    stack<TreeNode*> s;
    TreeNode* lastVisited = nullptr;
    TreeNode* current = root;

    while (current || !s.empty()) {
        // Go to the leftmost node
        while (current) {
            s.push(current);
            current = current->left;
        }

        // Peek the node on top of the stack
        TreeNode* node = s.top();

        // If the right child exists and hasn't been visited, process the right subtree
        if (node->right && lastVisited != node->right) {
            current = node->right;
        } else {
            // Process the node
            result.push_back(node->val);
            lastVisited = node;
            s.pop();
        }
    }

    return result;
}
```
----

5) **Balanced Binary Tree**: Given a binary tree, determine if it is
height-balanced
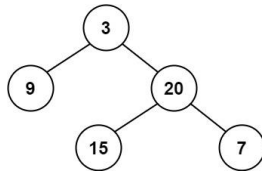
Binary tree would be balanced, when abs(ltree-rtree))<=1
Approach:
1) Base case: if the current node is NULL, return 0 (height of an empty tree)
2) Recursively calculate the height of the left subtree, If the left subtree is unbalanced, propagate the unbalance status(-1)
3) Recursively calculate the height of the right subtree, If the right subtree is unbalanced, propagate the unbalance status(-1)
4) Check if the difference in height between left and right subtrees is greater than 1, If it's greater, the tree is unbalanced,
   return -1 to propagate the unbalance status
5) Return the maximum height of left and right subtrees, adding 1 for the current node

Time Complexity: O(N)
Space Complexity: O(1) Extra Space + O(H) Recursion Stack space (Where "H" is the height of binary tree)

*/

```cpp
bool isBalanced(Node* root) {
    // Check if the tree's height difference
    // between subtrees is less than 2
    // If not, return false; otherwise, return true
    return dfsHeight(root) != -1;
}

// Recursive function to calculate
// the height of the tree
int dfsHeight(Node* root) {
    // Base case: if the current node is NULL,
    // return 0 (height of an empty tree)
    if (root == NULL) return 0;

    // Recursively calculate the
    // height of the left subtree
    int leftHeight = dfsHeight(root->left);

    // If the left subtree is unbalanced,
    // propagate the unbalance status
    if (leftHeight == -1)
```

```
        return -1;

    // Recursively calculate the
    // height of the right subtree
    int rightHeight = dfsHeight(root->right);

    // If the right subtree is unbalanced,
    // propagate the unbalance status
    if (rightHeight == -1)
        return -1;

    // Check if the difference in height between
    // left and right subtrees is greater than 1
    // If it's greater, the tree is unbalanced,
    // return -1 to propagate the unbalance status
    if (abs(leftHeight - rightHeight) > 1)
        return -1;

    // Return the maximum height of left and
    // right subtrees, adding 1 for the current node
    return max(leftHeight, rightHeight) + 1;
  }
```

----

6) **Diameter of Binary Tree**: Given the root of a binary tree, return *the length of the **diameter** of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The **length** of a path between two nodes is represented by the number of edges between them.
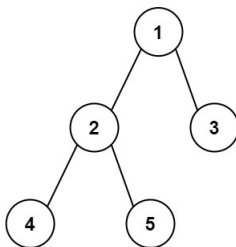
/*Diameter: The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The length of a path between two nodes is represented by the number of edges between them.

Approach:

1) Since Daimeter is max path between two nodes => (ltree + rtree) at any node => ltree/rthree of nth node would be height of n-1th node
   we should calculate and return height of every node, that would become ltree or rtee

2) Diameter = max(diameter, ltree + rtree)

```
int diameterOfBinaryTree(TreeNode* root) {
    int diameter = 0;
    height(root, diameter);
    return diameter;
}

int height(TreeNode* root, int& diameter) {

    if (root==NULL)
    {
        return 0;
    }

    int lh = height(root->left, diameter);
    int rh = height(root->right, diameter);

    diameter = max(diameter, lh + rh);

    return 1 + max(lh, rh);
}
```
----

7) **Print Left View of a Binary Tree**
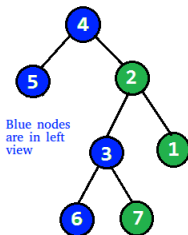
Time Complexity: O(N)
Space Complexity: O(H)   (H -> Height of the Tree)

Approach:
We need to follow approach: Rt L R(preorder-recursive), can not follow level order traversal as it will increase time complexity
Used vector to store node value
When vector::size() == level means first node of that level:

```
void checkLeftView(Node* root, int level, vector<int>&ans)
{
        if(root == NULL)
                return;
```

```
        if(ans.size()==level)
                ans.push_back(root->data);

  checkLeftView(root->left, level+1, ans);
        checkLeftView(root->right, level+1, ans);

}

vector<int> leftView(Node *root)
{
  vector<int>ans;
  checkLeftView(root, 0, ans);
  return ans;
}
```

----

8) **Binary Tree Right Side View**: Given the root of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom*.
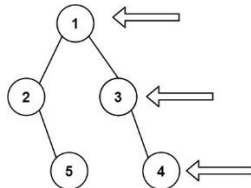
Time Complexity: O(N)
Space Complexity: O(H)   (H -> Height of the Tree)

Approach:
We need to follow approach: Rt R L(preorder-recursive), can not follow level order traversal as it will increase time complexity
Used vector to store node value
When vector::size() == level means first node of that level:

```
void checkRightView(Node* root, int level, vector<int>&ans)
{
        if(root == NULL)
                return;
        if(ans.size()==level)
                ans.push_back(root->data);

  checkRightView(root->right, level+1, ans);
  checkRightView(root->left, level+1, ans);

}
```
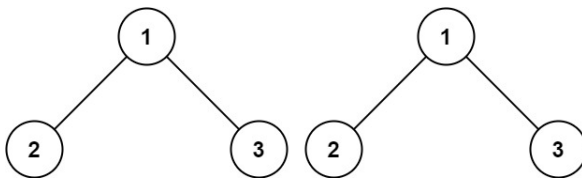
```
vector<int> rightView(Node *root)
{
  vector<int>ans;
  checkRightView(root, 0, ans);
  return ans;
}
```
----

9) **Same Tree**/Identical Tree:
Given the roots of two binary trees p and q, write a function to check if they are the same or not.
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.
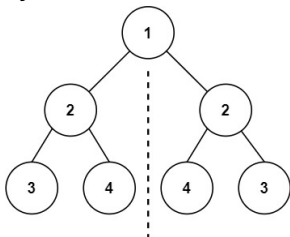
Approach:
    1)   We will call method isSametree with two args p and q
Will check if p->val == q->val && then will call isSameTree(p->left, q->left) && isSameTree(p->right, q->right) to ensure all returns true.
    2)
3) Check if left or right null, if anyone is null then return lt==rt

```
bool isSameTree(TreeNode* p, TreeNode* q)
  {
    if(p==NULL || q==NULL )
    {
      return p==q;
    }

    return (p->val == q->val) && isSameTree(p->left, q->left) && isSameTree(p->right, q->right);

  }
```
----

10) **Symmetric Tree**: Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

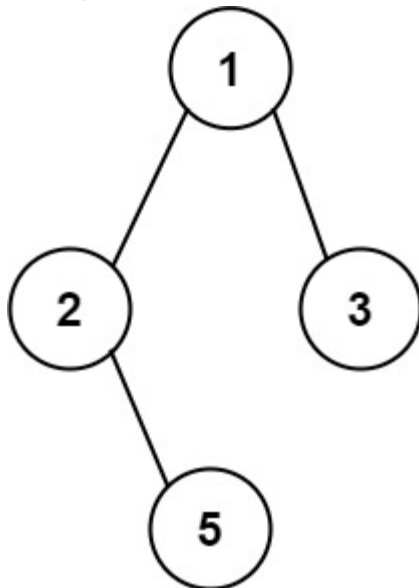**Approach:**
1) We will call method isSymmetricBT with two args root->left and root->right
2) Check if left or right null, if anyone is null then return lt==rt
3) Check lt->val==rt->val && recursively check below condition:
   (lt->left, rt->right) &&(lt->right, rt->left)

```
bool isSymmetric(TreeNode* root)
  {
    if(root==NULL)
      return true;
    return isSymmetricBT(root->left, root->right);
  }
private:
  bool isSymmetricBT(TreeNode* lt, TreeNode* rt)
  {
    if(lt==NULL || rt==NULL)
      return lt==rt;
    return (lt->val==rt->val) && isSymmetricBT(lt->right, rt->left) && isSymmetricBT(lt->left,
rt->right);
  }
```
----

11) **Binary Tree Paths**: Given the root of a binary tree, return *all root-to-leaf paths in **any
order**.*A **leaf** is a node with no children.

Example 1:

Input: root = [1,2,3,null,5]
Output: ["1->2->5","1->3"]
Example 2:
Input: root = [1]

Output: ["1"]

Approach:
- will call binaryTreePaths() and inside it will call paths(TreeNode* root, string str, vector<string> &ans) => will pass default string "" to identify empty/first time called to append only value to empty string.
- If str is not empty then we should append -> and root->val
- Check if root is leaf: if(root->left == NULL && root->right ==NULL) , means we found first path which will be pushed back to ans.
- Otherwise recursive call on left subtree and right subtree until we get root == null.

```
void paths(TreeNode* root, string str, vector<string> &ans) {
    if(root == NULL) return;

    //It means, it's first time and we should append root->val to str
    if(str == "")
        str+=  to_string(root->val);

     //It means, it's not first time and we should append -> to str
    else
         str+= "->"+to_string(root->val);

    //We found leaf node and now we got result string and can be added to ans.
    if(root->left == NULL && root->right ==NULL)
      ans.push_back(str);
    else {
      paths(root->left, str, ans);
      paths(root->right, str, ans);
    }
  }
  vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> ans;
    paths(root, "",ans);
    return ans;
  }
```
----
12) **Subtree of Another Tree**: Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.
A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

Approach:

- First, check if subroot(root of smaller tree) is null, if null then return true, because tree with null root will be subroot of any tree.
- Now check if root of main tree is null, if so, then return false
- Now check if both tree are identical, if yes then return true.
- Finally, recursively call issubtree function for left subtree and subroot and right subtree with subroot, either should be true to return true.
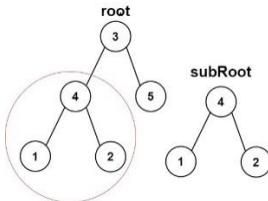


```cpp
bool isSubtree(TreeNode* root, TreeNode* subRoot) {
    if(subRoot==NULL)
        return true;
    if(root==NULL)
        return false;
    if(isidentical(root,subRoot))
        return true;
    return isSubtree(root->left,subRoot) || isSubtree(root->right,subRoot);
}

bool isidentical(TreeNode* a, TreeNode* b){
    if(a==NULL || b==NULL)
        return a==b;
    return (a->val == b->val) && isidentical(a->left,b->left) && isidentical(a->right,b->right);
}
```
----

12) Search in a Binary Search Tree:
You are given the root of a binary search tree (BST) and an integer val.
Find the node in the BST that the node's value equals val and return the subtree rooted with that node. If such a node does not exist, return null.
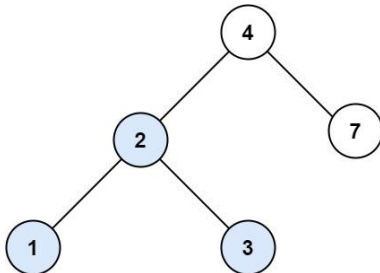
**Input:** root = [4,2,7,1,3], val = 2
Output: [2,1,3]

```cpp
TreeNode* searchBST(TreeNode* root, int val) {
    if(root == NULL || root->val == val) return root;
    if(val < root->val) return searchBST(root->left, val);
```

```
        else return searchBST(root->right, val);
    }
```

----

13) **Range Sum of BST**: Given the root node of a binary search tree and two integers low and high, return *the sum of values of all nodes with a value in the **inclusive** range [low, high].*
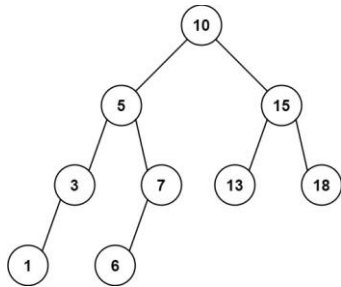
Input: root = [10,5,15,3,7,null,18], low = 7, high = 15
Output: 32
Explanation: Nodes 7, 10, and 15 are in the range [7, 15]. 7 + 10 + 15 = 32.
Approach:
The function first checks if the root is NULL. If it is, then it returns 0. If the root is not NULL, the function recursively calls itself for the left and right subtrees, and stores the sum of values in 'left' and 'right' variables respectively.
Then, the function checks if the value of the root node falls within the given range. If it does, then it returns the sum of the root node's value, 'left' and 'right'. If it doesn't, then it returns 'left' and 'right' sum.

```cpp
int rangeSumBST(TreeNode* root, int low, int high)
{
      //If root is null, return 0
      if(root == nullptr) return 0;

      //If root->val <low: then no need to traverse to left of root, traverse
//to right of it.
      if(root->val < low) return rangeSumBST(root->right, low, high);

      //If root->val >high: then no need to traverse to right of root,
//traverse to left of it.
      if(root->val > high) return rangeSumBST(root->left, low, high);

//If values are in range then: add up values of root, left subtree and right
subtree
      return root->val + rangeSumBST(root->left, low, high) +
rangeSumBST(root->right, low, high);
    }
```
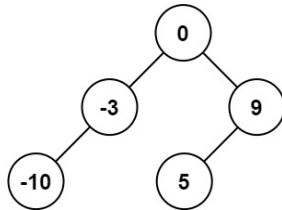 Time complexity: *O(n)*
Space complexity: *O(h)* where h is the height of the binary search tree
----

14) **Convert Sorted Array to Binary Search Tree**: Given an integer array nums where the elements are sorted in **ascending order**, convert *it to a*
height-balanced *binary search tree*.

Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
**Explanation:** [0,-10,5,null,-3,null,9] is also accepted:

```
TreeNode*helper(vector<int>&arr,int low,int high) {
    if(low > high) return NULL;
    int mid = low+(high-low)/2;
    TreeNode*root=new TreeNode(arr[mid]);
    root->left=helper(arr,low,mid-1);
    root->right=helper(arr,mid+1,high);
    return root;

  }
  TreeNode* sortedArrayToBST(vector<int>& arr) {
    int n=arr.size();
    return helper(arr,0,n-1);
  }
```
----
15) **Merge Two Binary Trees**: You are given two binary trees root1 and root2.
Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.
Return *the merged tree*.
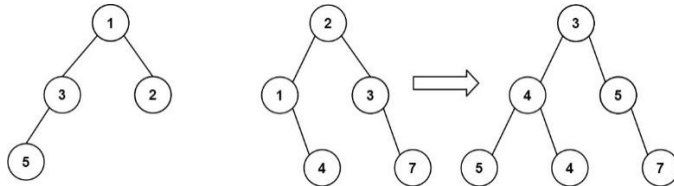
Input: root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]
Output: [3,4,5,5,4,null,7]
```
TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
    if (root1==NULL) return root2; // If root1 is null, return root2
    if (root2==NULL) return root1; // If root2 is null, return root1
    // Merge the values of root1 and root2 and create a new node
```

```
    TreeNode* merged = new TreeNode(root1->val + root2->val);
    // Recursively merge the left children
    merged->left = mergeTrees(root1->left, root2->left);
    // Recursively merge the right children
    merged->right = mergeTrees(root1->right, root2->right);

    return merged;
}
```
----

16) **Maximum Depth of Binary Tree**: Given the root of a binary tree, return *its maximum depth*. A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.
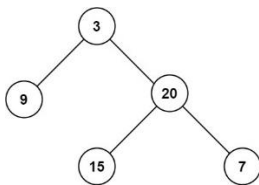

```
int maxDepth(TreeNode* root) {
    if (root == NULL)
        return 0;
    int l = maxDepth(root->left);
    int r = maxDepth(root->right);
    return max(l, r) + 1;
```
----

17) **Lowest Common Ancestor of a Binary Search Tree**: Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.
According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**)."
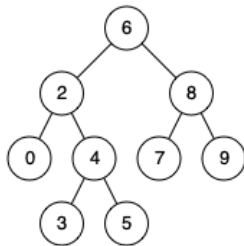
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.
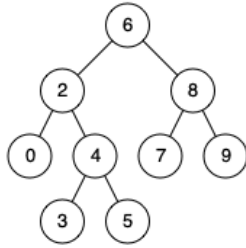
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    //base case
    if (root == NULL || root == p || root == q) {
      return root;
    }
    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);

    //result
    if(left == NULL) {
      return right;
    }
    else if(right == NULL) {
      return left;
    }
    else { //both left and right are not null, we found our result
      return root;
    }
  }
```
----
18) **Path Sum**: Given the root of a binary tree and an integer targetSum, return true if the tree has a **root-to-leaf** path such that adding up all the values along the path equals targetSum. A **leaf** is a node with no children.
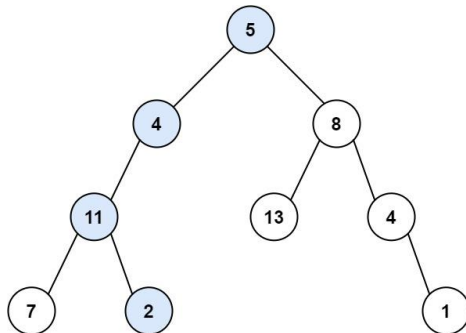
Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22
Output: true
Explanation: The root-to-leaf path with the target sum is shown.
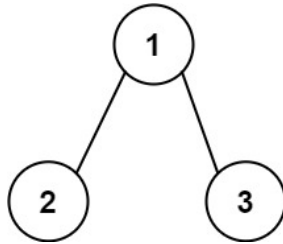
Input: root = [1,2,3], targetSum = 5
Output: false
Explanation: There two root-to-leaf paths in the tree:
(1 --> 2): The sum is 3.
(1 --> 3): The sum is 4.
There is no root-to-leaf path with sum = 5.

```cpp
bool solve(TreeNode* &root, int &targetSum, int sum) {
    if (root == NULL)
      return false;
    sum = sum + root->val;

    //Check if leaf node is found and sum is equal to target sum or not:
    //If it is equal to targetSum, return true else return false.
    if (root->left == NULL && root->right == NULL) {
      if (sum == targetSum)
        return true;
      else
        return false;
    }
    //Recursively traverse left and right subtree and return (leftAns || rightAns) =>if path was
found anyone will be returning true back to actual call/root.
    bool leftAns = solve(root->left, targetSum, sum);
    bool rightAns = solve(root->right, targetSum, sum);
    return leftAns || rightAns;
  }
  bool hasPathSum(TreeNode* root, int targetSum) {
    if (root == NULL)
      return false;
    int sum = 0;
    bool ans = solve(root, targetSum, sum);
    return ans;
  }
```
----

19) **Minimum Absolute Difference in BST**: Given the root of a Binary Search Tree (BST), return *the minimum absolute difference between the values of any two different nodes in the tree*.
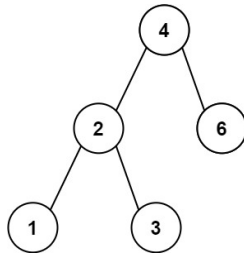
Input: root = [4,2,6,1,3]
Output: 1
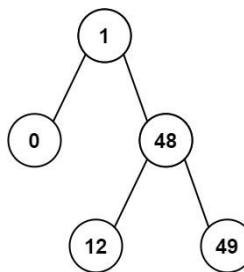
Input: root = [1,0,48,null,null,12,49]
Output: 1

Approach:
1. Perform inorder traversal of BST and store it in a array.
2. The array is sorted, as it is BST.
3. Now just do : Start loop 1 to n-1.
   - Arr[i] – Arr[i-1]
   - Minimize the above value. i.e. Find the min most difference
   - No need to use abs() as array is sorted. i+1 element is always >= i element
4. Return the min value.

```
void inorder(TreeNode* root){
    if(root){
      inorder(root->left);
      a.push_back(root->val);
      inorder(root->right);
    }
  }

  int getMinimumDifference(TreeNode* root) {
    int m=INT_MAX;
    inorder(root);
```

```
        for(int i=1;i<a.size();i++){
           m=min(m,a[i]-a[i-1]);

        }
        return m;

   }
```

----

20) **Sum of Left Leaves**: Given the root of a binary tree, return *the sum of all left leaves*.
A **leaf** is a node with no children. A **left leaf** is a leaf that is the left child of another node.
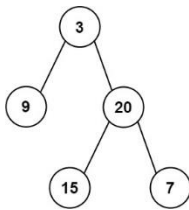
Input: root = [3,9,20,null,null,15,7]
Output: 24
Explanation: There are two left leaves in the binary tree, with values 9 and 15 respectively.

```
void solve(TreeNode* &root,int &ans){
   if(root==NULL){
      return;
   }
      if(root->left!=NULL)
{

         if(root->left->left==NULL && root->left->right==NULL){
            ans=ans+root->left->val;
         }

      }
      solve(root->left,ans);
      solve(root->right,ans);

}
   int sumOfLeftLeaves(TreeNode* root) {
      int ans=0;
         solve(root,ans);
         return ans;
   }
```
----

21) **Binary Tree Level Order Traversal**: Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).
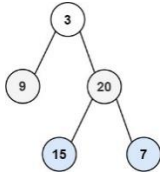
Example 1:
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
Example 2:
Input: root = [1]
Output: [[1]]
Example 3:
Input: root = []
Output: []

```
vector<vector<int>> levelOrder(TreeNode* root)
{
    vector<vector<int>>ans;
    //Base case
    if(root==NULL)
        return ans;
    queue<TreeNode*>q;
    q.push(root);

    while(!q.empty()){
      int s=q.size();
      vector<int>v;
      for(int i=0;i<s;i++){
        TreeNode *node=q.front();
        q.pop();
        if(node->left!=NULL)q.push(node->left);
        if(node->right!=NULL)q.push(node->right);
        v.push_back(node->val);
      }
      ans.push_back(v);
    }
    return ans;
}
```
Time complexity: o(n)
Space complexity: O(n)

----

22) **Binary Tree Zigzag Level Order Traversal**:
Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*.
(i.e., from left to right, then right to left for the next level and alternate between).
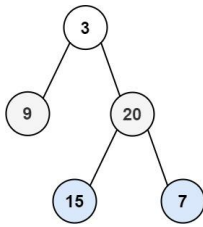
```cpp
vector < vector < int >> zigzagLevelOrder(TreeNode * root)
{
  vector < vector < int >> result;
  //Base case
  if (root == NULL)
  {
    return result;
  }

  queue < TreeNode * > nodesQueue;
  nodesQueue.push(root);
  bool leftToRight = true;

  while (!nodesQueue.empty())
  {
    int size = nodesQueue.size();
    //Take vector of size, will store element by index to print in left to right or    ///reverse.
    vector < int > row(size);
    for (int i = 0; i < size; i++)
    {
      TreeNode * node = nodesQueue.front();
      nodesQueue.pop();

      // find position to fill node's value
      int index = (leftToRight) ? i : (size - 1 - i);

      row[index] = node -> val;
      if (node -> left)
      {
        nodesQueue.push(node -> left);
      }
      if (node -> right)
      {
        nodesQueue.push(node -> right);
      }
    }
    // after this level
    leftToRight = !leftToRight;
```

```
      result.push_back(row);
   }
   return result;
 }
```
----

23) **Top View of Binary Tree:** Given below is a binary tree. The task is to print the top view of binary tree. Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. For the given below tree

```
     1
   /   \
  2     3
 / \   / \
4   5 6   7
```
Top view will be: 4 2 1 3 7

Approach:
we will store Node and line(level number) to queue.
we will travere in level order traversal
we will store node->data into map[line]
*/

```cpp
vector<int> topView(Node *root)
  {
     vector<int> ans;
     if(root == NULL) return ans;
     map<int,int> mpp;
     queue<pair<Node*, int>> q;
     q.push({root, 0});
     while(!q.empty()) {
       auto it = q.front();
       q.pop();
       Node* node = it.first;
       int line = it.second;

                   //Check if level/line is not already entered then only add it in map
                   //otherwise top view data will be overwritten by bottom view nodes.
                   if(mpp.find(line) == mpp.end())
                   {
                           mpp[line] = node->data;
                   }

       if(node->left != NULL) {
         q.push({node->left, line-1});
       }
       if(node->right != NULL) {
```

```
        q.push({node->right, line + 1});
      }

    }

    for(auto it : mpp) {
      ans.push_back(it.second);
    }
    return ans;

  }
```
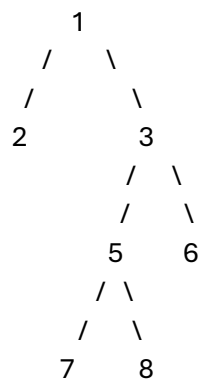Time Complexity: O(N)
Space Complexity: O(N)

--

24) **Bottom View of Binary Tree:**
Given a binary tree, return an array where elements represent the bottom view of the binary tree from left to right.
Note: If there are multiple bottom-most nodes for a horizontal distance from the root, then the latter one in the level traversal is considered. For example, in the below diagram, 3 and 4 are both the bottommost nodes at a horizontal distance of 0, here 4 will be considered.

```
              1
            /   \
           /     \
          2       3
                 / \
                /   \
               5     6
              / \
             /   \
            7     8
```

Output: [7, 5, 8, 6]

Approach:
we will store Node and line(level number) to queue.
we will travere in level order traversal

we will store node->data into map[line]
*/

```cpp
vector <int> bottomView(Node *root)
{
    vector<int> ans;
    if(root == NULL) return ans;
    map<int,int> mpp;
    queue<pair<Node*, int>> q;
    q.push({root, 0});
    while(!q.empty()) {
      auto it = q.front();
      q.pop();
      Node* node = it.first;
      int line = it.second;

      //Add data into map, no need to check if exist in map or not, anyway it's bottom view
and will be overwritten for bottom view nodes
      mpp[line] = node->data;

      if(node->left != NULL) {
        q.push({node->left, line-1});
      }
      if(node->right != NULL) {
        q.push({node->right, line + 1});
      }

    }

    for(auto it : mpp) {
      ans.push_back(it.second);
    }
    return ans;
  }
```

Time Complexity: O(N)
Space Complexity: O(N)

----

25) **Validate Binary Search Tree**: **Problem Statement:** You are given the root of a binary tree. The task is to determine if the given binary tree qualifies as a binary search tree. Conditions for a Binary Tree to qualify as Binary Search Tree (BST):

- The left child's key is **less** than the key of its parent.
- The right child's key is **more** than the key of its parent.
- The left and right subtrees also count as BST individually.

Example 1:
Input: Address of root node given
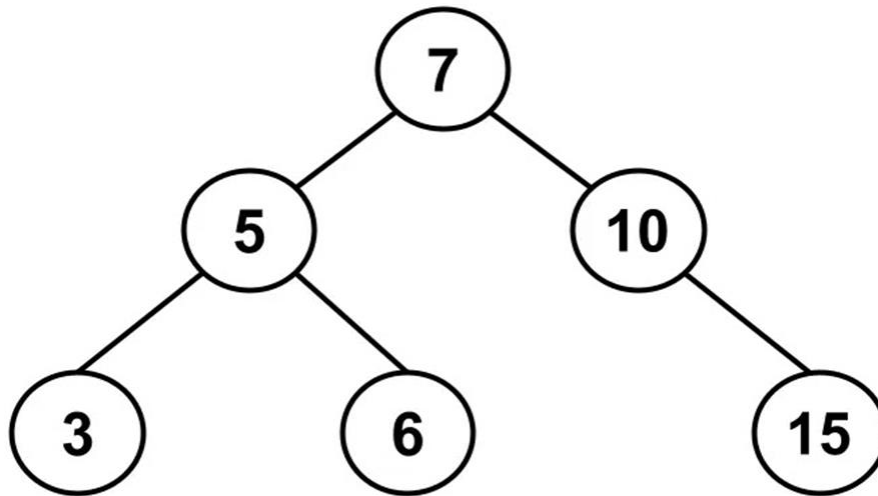
Output: True
Explanation: Since the above Binary tree obeys all the conditions of BST return true.
Example 2:
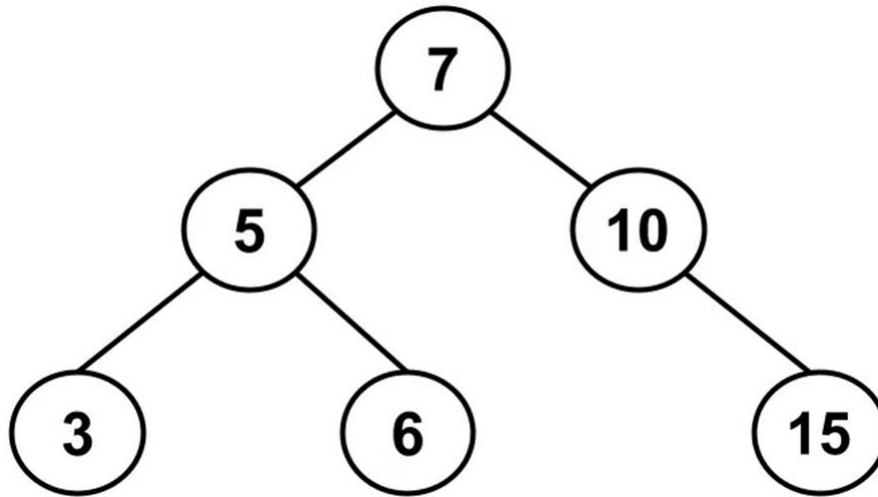**Input: Address of root node given.**
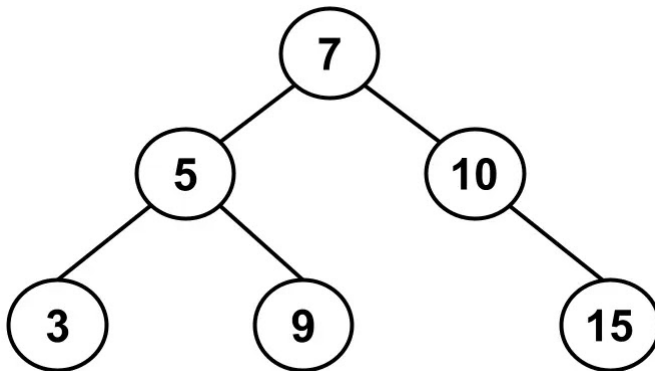
Output: False
Explanation: The root contains key 7, hence its left subtree should have nodes containing a key less than 7 but there is a node having its key as 9. This violates the principle of BST.
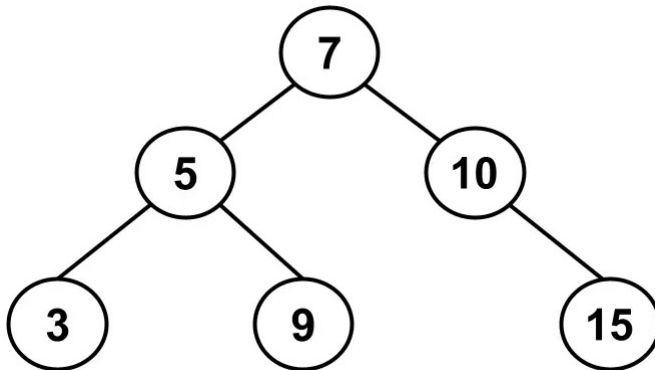
Example 1:
Input: Address of root node given

Output: True
Explanation: Since the above Binary tree obeys all the conditions of BST return true.
Example 2:
Input: Address of root node given.

Output: False
Explanation: The root contains key 7, hence its left subtree should have nodes containing a key less than 7 but there is a node having its key as 9. This violates the principle of BST.
Solution
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*
Pre-requisites: Recursion, *Tree traversal techniques*
Solution 1:
Initial thoughts: Let us assume we have a node X. To check for BST, X's key should not only be compared with its immediate left and right children but also its entire left and right subtree. So checking the entire left and right subtree for every node in a tree will be very inefficient.
Approach: The efficient method to solve this problem in a single traversal is discussed below:-
- **Before visiting every node, a range is assigned to it. The lower bound and upper bound of the range is passed as parameters to the function.**

- **While visiting the node observe if the node's key lies between the upper and lower bound.**
  - **If not, return a false without any further traversal.**
 - **If yes, make recursion calls once for the left subtree and right subtree of the node.**
- **Once we reach null, return true.**
  NOTE: The lower bound of the starting range is a minimum integer, say INT_MIN, and the upper bound is a maximum integer, say INT_MAX because the root's key can have any value.
  Have a look at the dry run for a better understanding.
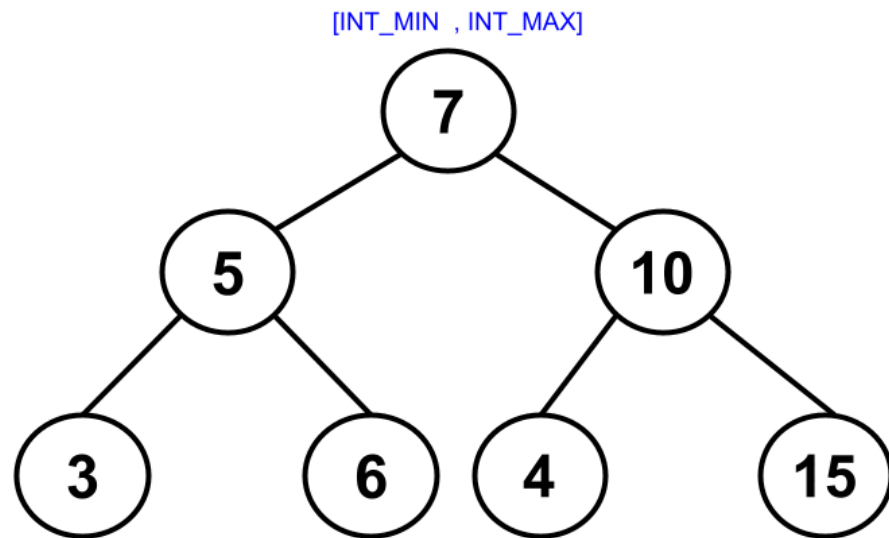
[INT_MIN , INT_MAX]

- **Start with root(key = 7) and the said initial range. INT_MIN < 7 < INT_MAX. Continue traversal.**
- **Arrive at node with key 5 with [INT_MIN,7] as range.INT_MIN < 5 < 7.Continue traversal.**
- **Arrive at a node with key 3 with [INT_MIN, 5] as a range. INT_MIN < 3 < 5. Continue traversal.**
- **Reaches NULL. Return true.**
- **Arrive at the node with key 6 with [5, 7] as a range. 5 < 6 < 7. Continue traversal.**
- **Reaches NULL. Return true.**
- **Arrive at a node with key 10 with [7, INT_MAX] as the range. 7 < 10 < INT_MAX. Continue traversal.**
- **Arrive at a node with key 4 with [7, 10] as the range. 4 does not lie in the range. Do not continue traversal, return false.**
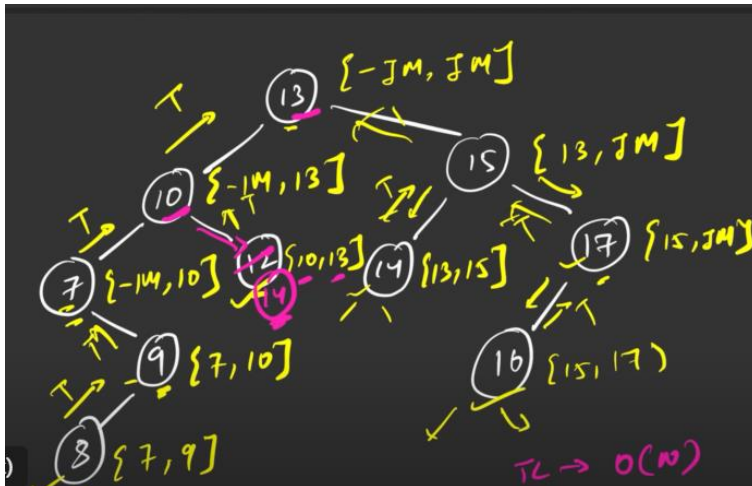
```cpp
bool isValidBST(Node * root)
{
        return solve(root, INT_MIN, INT_MAX);
}

   bool solve(Node * root, int minVal, int maxVal) {
      if (root == NULL) return true;
      if (root -> data >= maxVal || root -> data <= minVal) return false;
      return solve(root -> left, minVal, root -> data) && solve(root -> right,
      root -> data, maxVal);
   }
int main() {
   Node * root = new Node(7);
   root -> left = new Node(5);
   root -> left -> left = new Node(3);
   root -> left -> right = new Node(6);
   root -> right = new Node(10);
   root -> right -> left = new Node(4);
   root -> right -> right = new Node(15);
   Solution ob;
   bool ans = ob.isValidBST(root);
   if (ans == true) {
      cout << "Valid BST";
   } else {
      cout << "Invalid BST";
   }
   return 0;
}
```

Output: Invalid BST
Time Complexity: O(N) for traversing N nodes.
Space Complexity: O(1) if we ignore the auxiliary stack space.

----

26) **Flatten Binary Tree to Linked List**: Given the root of a binary tree, flatten the tree into a "linked list":

- The "linked list" should use the same TreeNode class where the right child pointer points to the next node in the list and the left child pointer is always null.
- The "linked list" should be in the same order as a **pre-order traversal** of the binary tree.
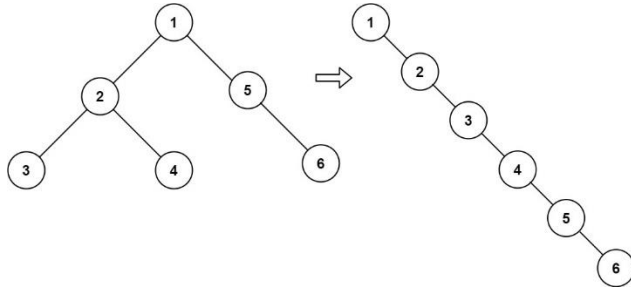
Example 1:
Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]
Example 2:
Input: root = []
Output: []
Example 3:
Input: root = [0]
Output: [0]

Morris traversal approach:
1) set current to root
2) Traverse until current is not null
3) check if current->left is not null then set prev to current->left
4) Check right of prev until it is not null, when it's null set prev->right = current->right
5) set current->right = current->left and remove current->left
6) Now set current=current->right(else part of if(current->left == NULL)

```
void flatten(TreeNode* root)
  {
    TreeNode* current = root;
    while(current != NULL)
    {
      If(current->left == NULL)
              current = current->right;
      else
       {
         TreeNode* prev= current->left;
         while(prev->right != NULL)
         {
```

```
            prev = prev->right;
        }
        prev->right = current->right;
        current->right= current->left;
        current->left = NULL;
      }
    }
  }
```
Time complecity:O(n)
Space complexity: O(1)

 ----

27) **Add node to leaf in binary tree:**

```
void addNodeToLeafNode(treeNode* root)
{
   // if node is null, return
   if (!root)
      return;

   // if node is leaf node, print its data
   if (!root->left && !root->right)
   {
      root->left = createNode(root->data);
      root->right = createNode(root->data);
      return;
   }

   // if left child exists, check for leaf
   // recursively
   if (root->left)
      addNodeToLeafNode(root->left);

   // if right child exists, check for leaf
   // recursively
   if (root->right)
      addNodeToLeafNode(root->right);
}

void traversePreOrder(treeNode* temp)
{
   if (temp != NULL)
   {
      cout << " " << temp->data;
      traversePreOrder(temp->left);
      traversePreOrder(temp->right);
```

```
    }
}

int main() {
    treeNode* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->right->left = createNode(5);

    //addNodeToLeafNode(root);
    traversePreOrder(root);

}
```
----

28) **Predecessor and Successor: Problem Statement:** Given a Binary Search Tree and a 'key' value which represents the data data of a node in this tree. Return the inorder predecessor and successor of the given node in the BST.

• The predecessor of a node in BST is that node that will be visited just before the given node in the inorder traversal of the tree. Return nullptr if the given node is the one that is visited first in the inorder traversal.

• The successor of a node in BST is that node that will be visited immediately after the given node in the inorder traversal of the tree. Return nullptr if the given node is visited last in the inorder traversal.

Example 1:Input:Binary Search Tree: 5 3 7 2 4 6 9 1 -1 -1 -1 -1 -1 8 10, Key = 8
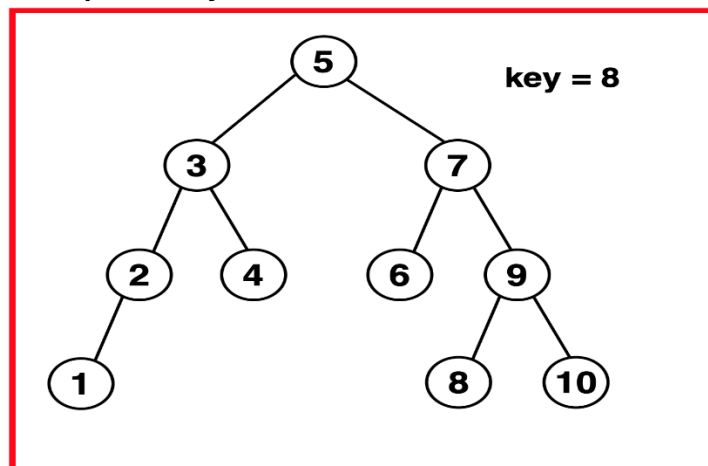
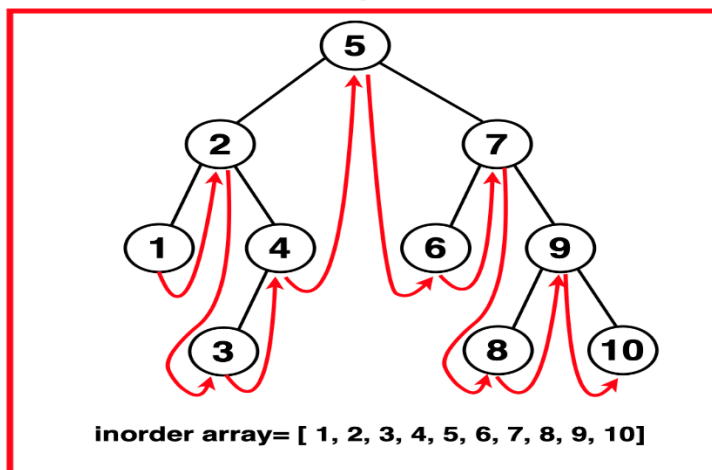**Output:Inorder Predecessor : 7, Inorder Successor : 9Explanation:**

inorder array= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
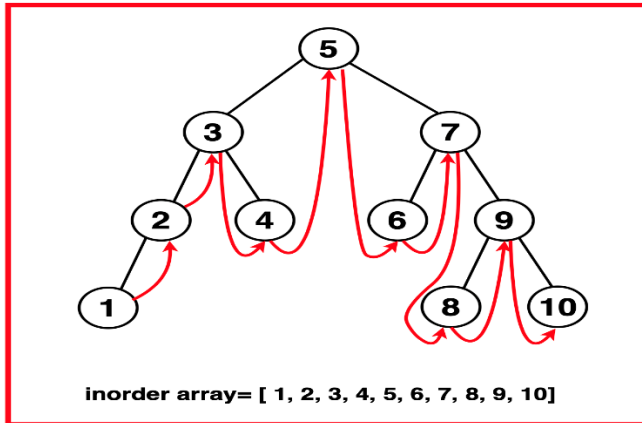
Example 2:Input:Binary Search Tree:

5 2 7 1 4 6 9 -1 -1 3 -1 -1 -1 8 10, Key = 3

key = 3
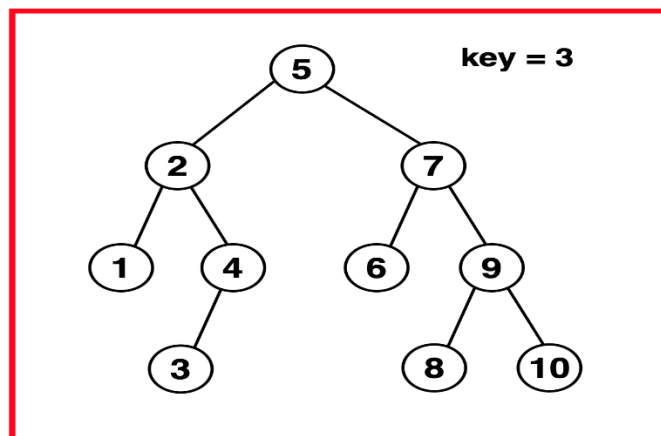
Output: Inorder Predecessor

: 2, Inorder Successor : 5Explanation:

inorder array= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Algorithm:

Step 1:Initialise a variable 'successor' as int max.

Step 2: Traverse the Binary Search Tree (BST) based on the key value.

- **If the current node value is smaller than the key, move to its right child.**

- **If the current node value is greater than the key, update 'successor' as the minimum between the current 'successor' value and the current node value then move to its left subtree.**
Step 3: Continue this process till reaching null.
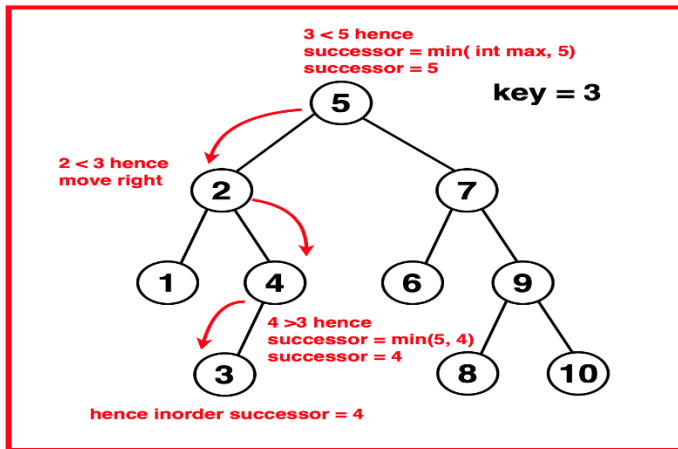

```cpp
// Function to find the inorder successor of a node in a BST
TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
    TreeNode* successor = NULL;

    // Traverse until root is not null
    while (root != NULL) {
        // If the value of p is greater or
        // equal to the current root's value
        if (p->val >= root->val) {
            // Move to the right subtree
            root = root->right;
        } else {
            // If the value of p is smaller, move to the left subtree
            // Update the successor to the current root and traverse left
            successor = root;
            root = root->left;
        }
    }

    // Return the inorder successor node
    return successor;
}
int main() {

    // Constructing the BST
    TreeNode* root = new TreeNode(5);
    root->left = new TreeNode(3);
    root->right = new TreeNode(6);
```

```cpp
    root->left->left = new TreeNode(2);
    root->left->right = new TreeNode(4);
    root->right->right = new TreeNode(7);

    cout << "BST: ";
    printInOrder(root);
    cout << endl;

    // Node for which we want to
    // find the inorder successor
    TreeNode* p = root->left->right;

    // Find the inorder successor
    TreeNode* successor = inorderSuccessor(root, p);

    // Print the inorder successor's value
    if (successor != nullptr) {
        cout << "Inorder Successor of " << p->val << " is: " << successor->val << endl;
    } else {
        cout << "Inorder Successor of " << p->val << " does not exist." << endl;
    }

    return 0;
}
```

----

29) Check whether BST contains Dead End

Given a <u>Binary Search Tree</u> that contains unique positive integer values greater than 0. The task is to complete the function isDeadEnd which returns true if the BST contains a dead end else returns false. Here Dead End means a leaf node, at which no other node can be inserted.

Input :

```
      8
     / \
    5   9
   / \
  2   7
 /
1
```

Output :
Yes
Explanation :
Node 1 is a Dead End in the given BST.
Example 2:
Input :
```
      8
     / \
    7   10
   /   / \
  2   9  13
```
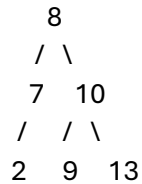
Output :
Yes
Explanation :
Node 9 is a Dead End in the given BST.

APPROACH:
1. *Initialize the minimum value to 1 and the maximum value to INT_MAX.*
   2. Recursively traverse the binary tree, starting from the root node.
   3. At each node, find its range [min,max].
- *If **min==max**, return true immediately.*
- *Otherwise, recursively traverse the left subtree with the new minimum value as 'min' and the new maximum value as 'root->data -1'.*
- *Also, recursively traverse the right subtree with the new minimum value as 'root->data+1' and the new maximum value as 'max'.*
   4. If either of the recursive calls returns true, then return true. Otherwise, return fasle.
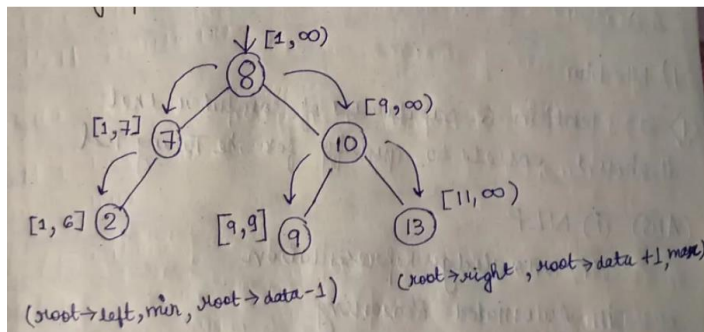

```
// Returns true if there is a dead end in tree,
// else false.
bool isDeadEndUtil(Node *root, int low, int high) {
    // Base case
    if (root == NULL)
        return false;
    // Check if current node falls within the range
```

```cpp
    if (root->data >= low && root->data <= high)
        return true;
    // Recur for left and right subtrees
    return isDeadEndUtil(root->left, low, root->data - 1) || isDeadEndUtil(root->right, root->data + 1, high);
}
bool isDeadEnd(Node *root) {
    return isDeadEndUtil(root, 1, INT_MAX);
}
// Driver program
int main()
{
    Node *root = NULL;
    root = insert(root, 8);
    root = insert(root, 5);
    root = insert(root, 2);
    root = insert(root, 3);
    root = insert(root, 7);
    root = insert(root, 11);
    root = insert(root, 4);
    if (isDeadEnd(root) == true)
        cout << "Yes " << endl;
    else
        cout << "No " << endl;
    return 0;
}
```

----

31) **Maximum Width of Binary Tree**: Given the root of a binary tree, return *the **maximum width** of the given tree*.

The **maximum width** of a tree is the maximum **width** among all levels.

The **width** of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.
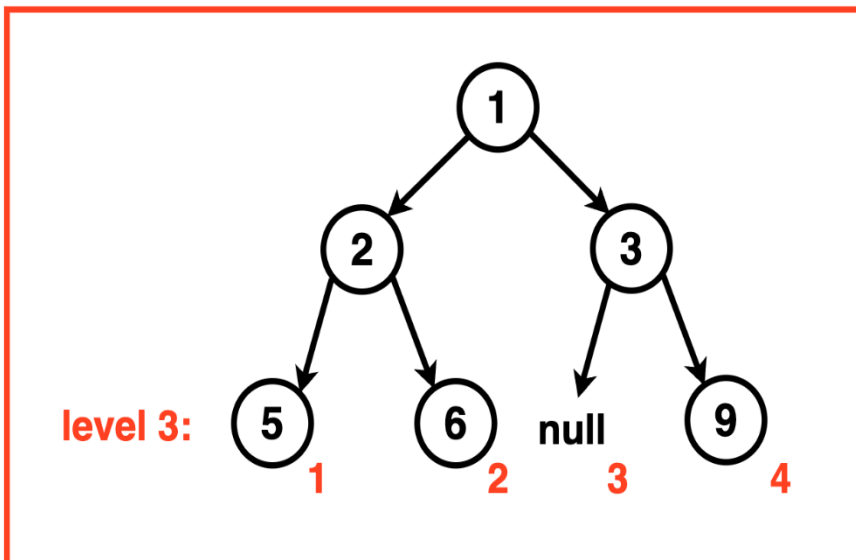
It is **guaranteed** that the answer will in the range of a **32-bit** signed integer.

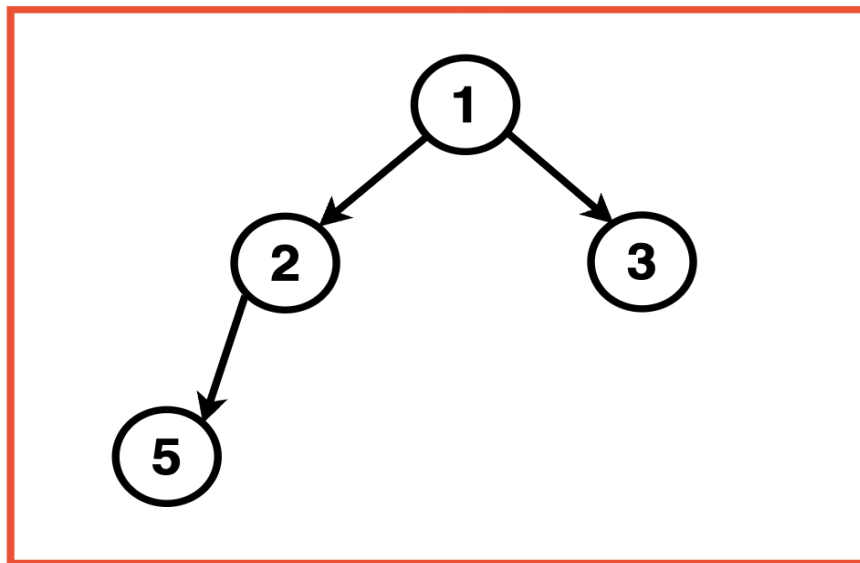**Example 1:Input:**Binary Tree: 1 2 3 5 6 -1 9
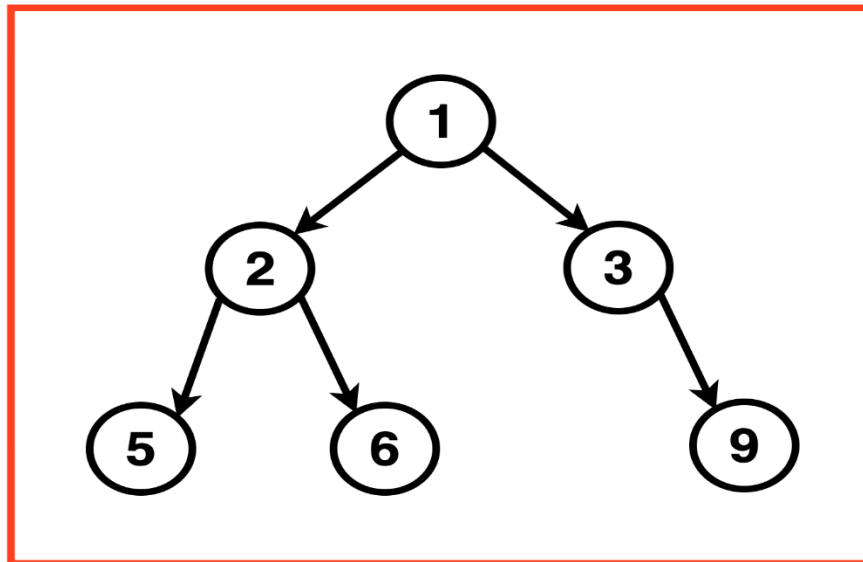
Output:

Maximum Width: 4**Explanation:** Level 3 is the widest level of the Binary Tree and whose end-to-end width is 4 comprising of nodes: {5, 6, null, 9}.
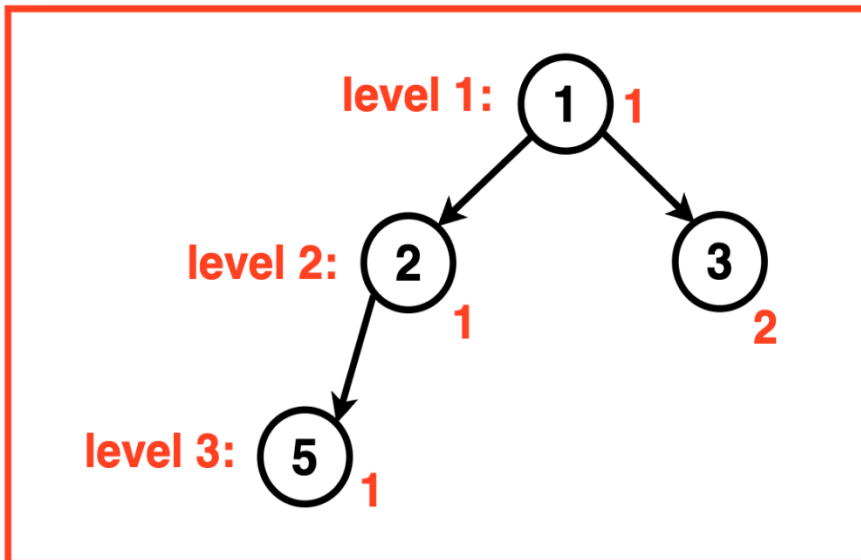

**Example**

2:Input: Binary Tree: 1 2 3 5

Output :

Maximum Width: 2**Explanation:** Level 2 is the widest level of the Binary Tree and whose end-to-end width is comprised of nodes: {2, 3}.


Practice:

Solve Problem

To determine the maximum width of a tree, an effective strategy would be to assign and identify indexes for the leftmost and rightmost nodes at each level. Using these indexes, we can calculate the width for each level by subtracting the index of the leftmost node from that of the rightmost node.
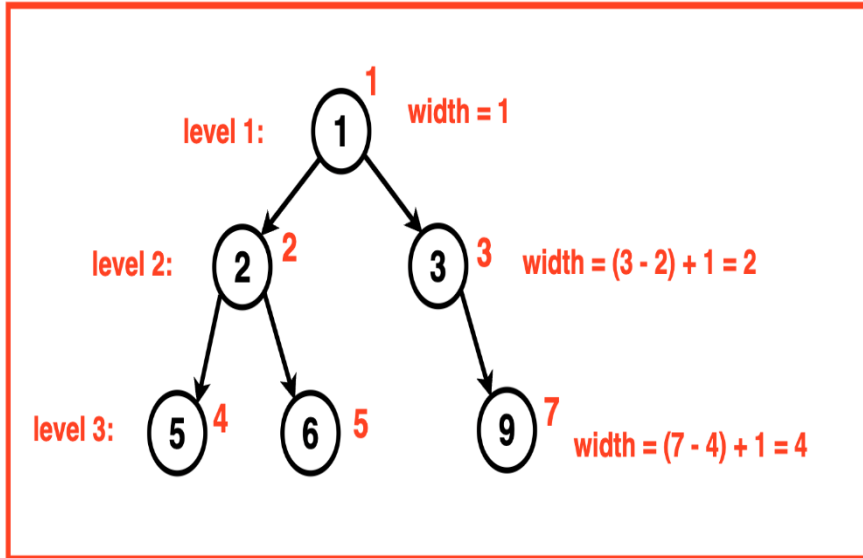
Start by assigning an index to the root node as 0. For each level, the left child gets an index equal to 2 * parent index, and the right child gets an index equal to 2 * parent index + 1. Using a level order traversal, we use the leftmost and rightmost nodes at each level and using their indices, get the width at that level. Keep track of the maximum width encountered during the traversal. Whenever a wider level is found, update the maximum width.
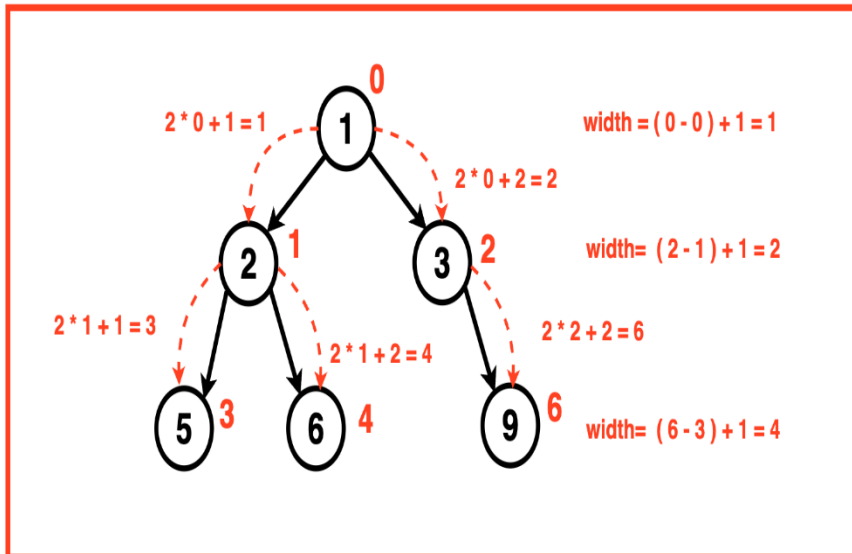
Algorithm:

Step 1: Initialize a variable `ans` to store the maximum width. If the root is null, return 0 as the width of an empty tree is zero.

Step 2: Create a queue to perform level-order traversal and each element of this queue would be a pair containing a node and its vertical index. Push the root node and its position (initially 0) into the queue.

Step 3: While the queue is not empty, perform the following steps:

- Get the number of nodes at the current level (size).

- Get the position of the front node in the current level which is the leftmost minimum index at that level.
- Initialize variables first and last to store the first and last positions of nodes in the current level.

  Step 4: Backtracking: For each node in the current level:
- Calculate the current position relative to the minimum position in the level.
- Get the current node (node) from the front of the queue.
- If this is the first node in the level, update the first variable.
- If this is the last node in the level, update the last variable.
- Enqueue the left child of the current node with index: 2 x current index - 1.
- Enqueue the right child of the current node with index: 2 x current index + 1.
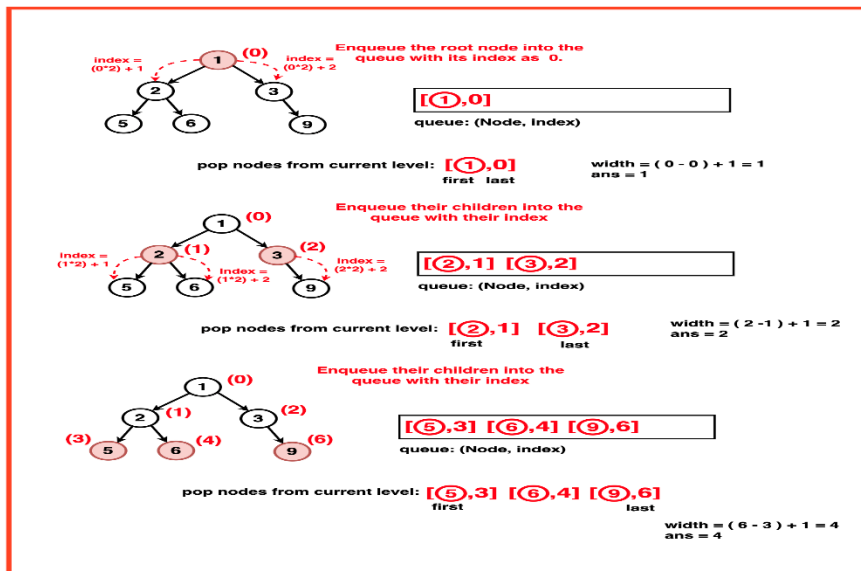
Step 5: Update the maximum width (ans) by calculating the difference between the first and last positions, and adding 1.

Step 6: Repeat the level-order traversal until all levels are processed. The final value of `ans` represents the maximum width of the binary tree, return it.

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <queue>

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
```

```cpp
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};




class Solution {
public:
  // Function widthOfBinaryTree to find the
  // maximum width of the Binary Tree
  int widthOfBinaryTree(TreeNode* root) {
    // If the root is null,
    // the width is zero
    if (!root) {
      return 0;
    }

    // Initialize a variable 'ans'
    // to store the maximum width
    int ans = 0;

    // Create a queue to perform level-order
    // traversal, where each element is a pair
    // of TreeNode* and its position in the level
    queue<pair<TreeNode*, int>> q;
    // Push the root node and its
    // position (0) into the queue
    q.push({root, 0});

    // Perform level-order traversal
    while (!q.empty()) {
      // Get the number of
      // nodes at the current level
      int size = q.size();
      // Get the position of the front
      // node in the current level
      int mmin = q.front().second;

      // Store the first and last positions
      // of nodes in the current level
      int first, last;

      // Process each node
      // in the current level
      for (int i = 0; i < size; i++) {
```

```cpp
            // Calculate current position relative
            // to the minimum position in the level
            int cur_id = q.front().second - mmin;
            // Get the current node
            TreeNode* node = q.front().first;
            // Pop the front node from the queue
            q.pop();

            // If this is the first node in the level,
            // update the 'first' variable
            if (i == 0) {
                first = cur_id;
            }

            // If this is the last node in the level,
            // update the 'last' variable
            if (i == size - 1) {
                last = cur_id;
            }

            // Enqueue the left child of the
            // current node with its position
            if (node->left) {
                q.push({node->left, cur_id * 2 + 1});
            }

            // Enqueue the right child of the
            // current node with its position
            if (node->right) {
                q.push({node->right, cur_id * 2 + 2});
            }
        }

        // Update the maximum width by calculating
        // the difference between the first and last
        // positions, and adding 1
        ans = max(ans, last - first + 1);
    }

    // Return the maximum
    // width of the binary tree
    return ans;
    }
};
```

```
int main() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(5);
    root->right = new TreeNode(1);
    root->left->left = new TreeNode(6);
    root->left->right = new TreeNode(2);
    root->right->left = new TreeNode(0);
    root->right->right = new TreeNode(8);
    root->left->right->left = new TreeNode(7);
    root->left->right->right = new TreeNode(4);

    Solution sol;

    int maxWidth = sol.widthOfBinaryTree(root);

    cout << "Maximum width of the binary tree is: "
            << maxWidth << endl;

    return 0;
}
```

Time Complexity: O(N) where N is the number of nodes in the binary tree. Each node of the binary tree is enqueued and dequeued exactly once, hence all nodes need to be processed and visited. Processing each node takes constant time operations which contributes to the overall linear time complexity.

Space Complexity: O(N) where N is the number of nodes in the binary tree. In the worst case, the queue has to hold all the nodes of the last level of the binary tree, the last level could at most hold N/2 nodes hence the space complexity of the queue is proportional to O(N).

----

32) **Construct Binary Tree from Preorder and Inorder Traversal**:
Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return *the binary tree*.
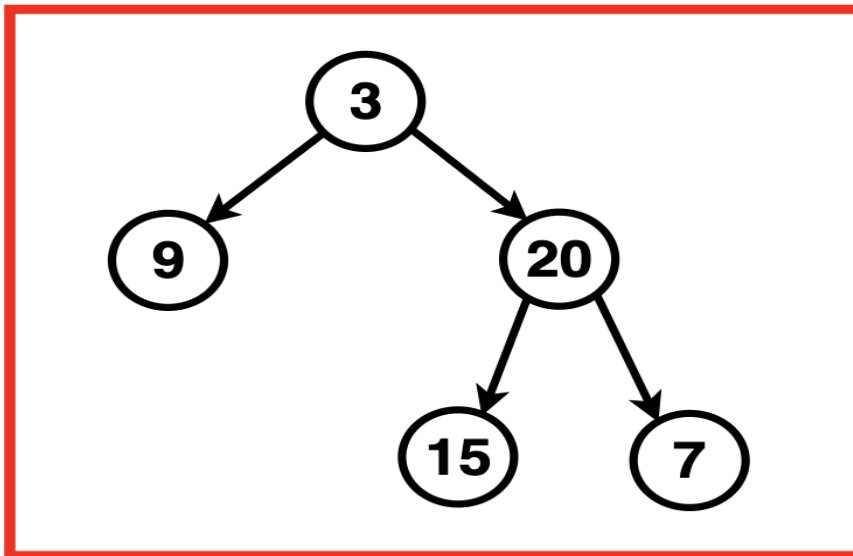
Example 1:Input: Inorder: [40, 20, 50, 10, 60, 30], Preorder: [10, 20, 40, 50, 30, 60]

Output:
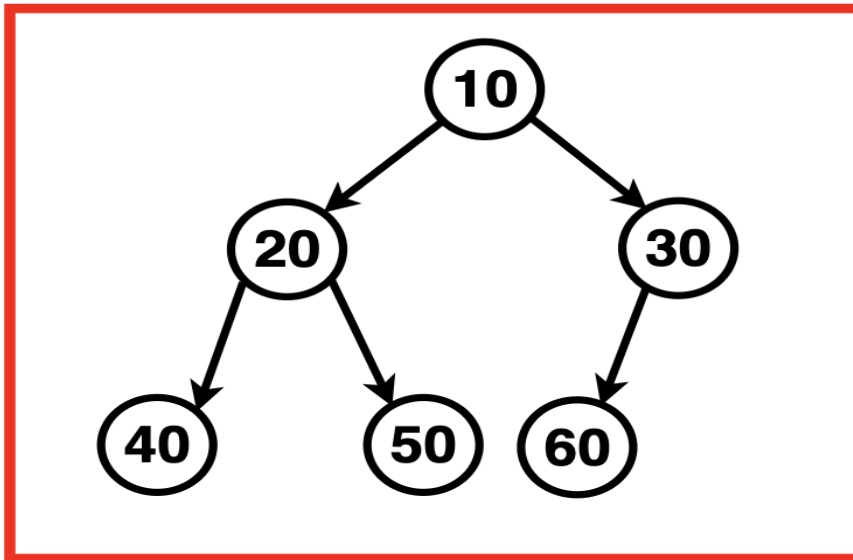
Explanation: The unique Binary Tree hence created has inorder traversal: [40, 20, 50, 10, 60, 30] and preorder traversal: [10, 20, 40, 50, 30, 60].**Example 2:Input:**Inorder: [9, 3, 15, 20, 7], Preorder: [3, 9, 20, 15, 7]

Output :

Explanation: The unique Binary Tree hence created has inorder traversal:[9, 3, 15, 20, 7]and preorder traversal: [3, 9, 20, 15, 7].

Before we dive into the algorithm, it's essential to grasp the significance of inorder and preorder traversals. Inorder traversal allows us to identify a node and its left and right subtrees, while preorder traversal ensures we always encounter the root node first. Leveraging these properties, we can uniquely construct a binary tree. The core of our approach lies in a recursive algorithm that creates one node at a time. We locate this root node in the inorder traversal, which splits the array into the left and right subtrees.
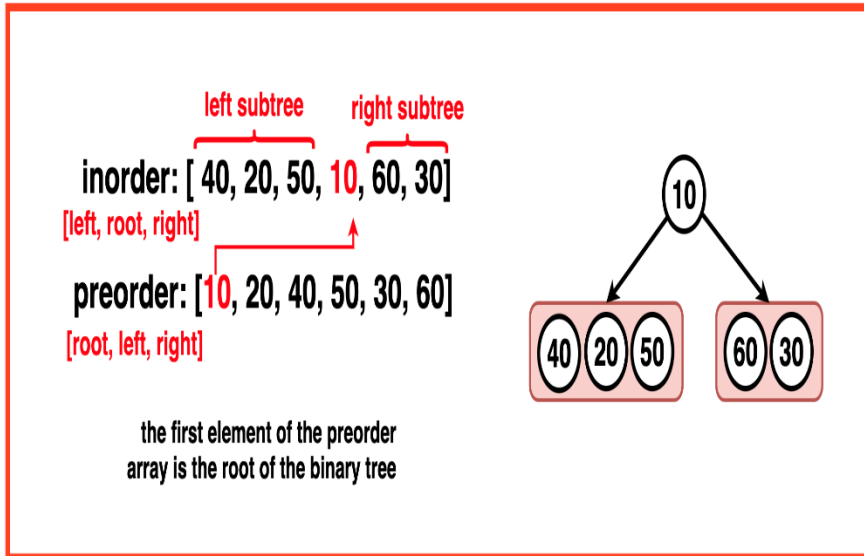
The inorder array keeps getting divided into left and subtrees hence to avoid unnecessary array duplication, we use variables (inStart, inEnd) and (preStart, preEnd) on the inorder and preorder array respectively. These variables effectively define the boundaries of the current subtree within the original inorder and preorder traversals. Everytime we encounter the root of a subtree via preorder traversal, we locate its position in the inorder array to get the left and right subtrees. So to save complexity on the linear look up, we employ a hashmap to store the index of each element in the inorder traversal. This transforms the search operation into a constant-time lookup.

Algorithm:

Step 1: Create an empty map to store the indices of elements in the inorder traversal. Iterate through each element in the inorder traversal and store its index in the map (inMap) using the element as the key and its index as the value.
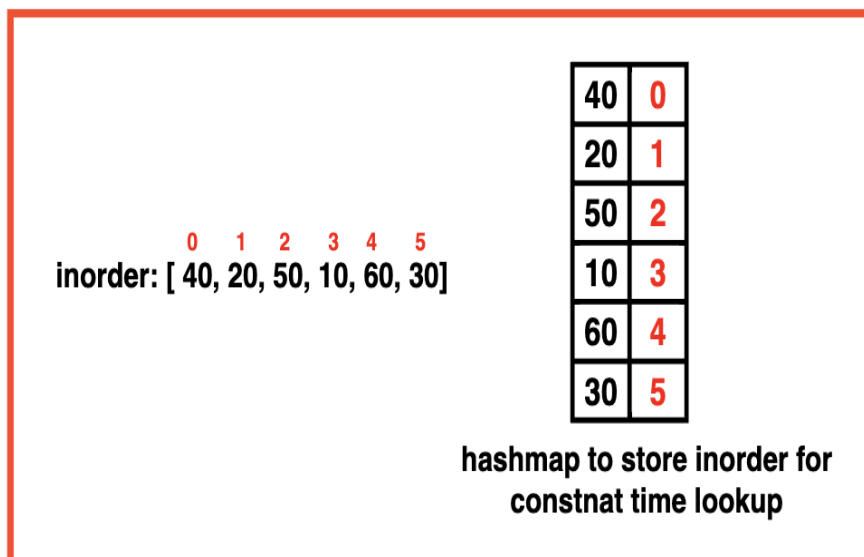

Step 2: Create a recursive helper function `buildTree` with the following parameters:

- Preorder vector
- Start index of preorder (preStart), initially set to 0

- End index of preorder (preEnd), initially set to preorder.size() - 1.
- Inorder vector
- Start index of inorder (inStart), initially set to 0.
- End index of inorder (inEnd), initially set to inorder.size() - 1.
- Map for efficient root index lookup in the inorder traversal.

Step 3: Base Case: Check if preStart is greater than preEnd or inStart is greater than inEnd. If true, return NULL, indicating an empty subtree/node.
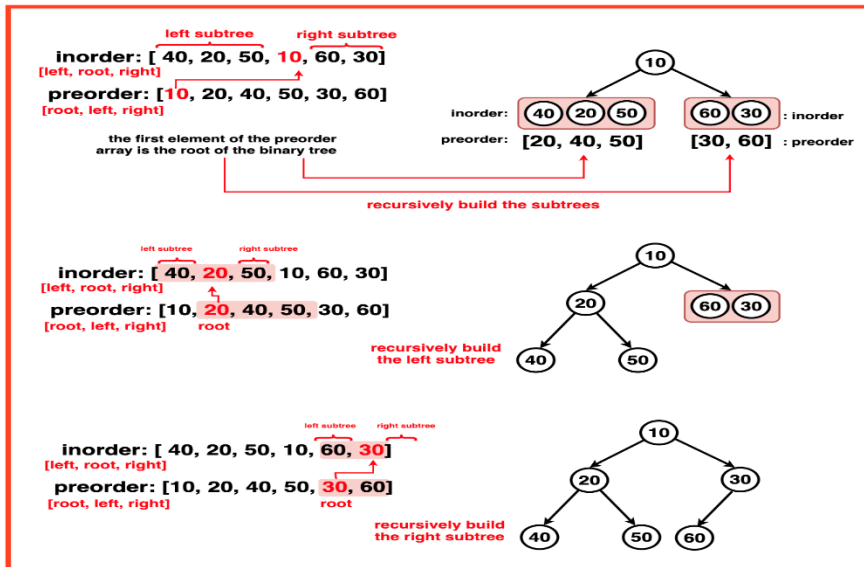
Step 4: The root node for the current subtree is the first element in the preorder traversal (preorder[preStart]). Find the index of this root node in the inorder traversal using the map (inMap[rootValue]). This is the rootIndex.

Step 5: Hence, the left subtree ranges from inStart to rootIndex. Subtracting these indexes gives us the leftSubtreeSize.

Step 6: TMake two recursive calls to buildTree to build the left and right subtrees: For the left subtree:

- Update preStart to preStart + 1 (moving to the next element in preorder)
- Update preEnd to preStart + leftSubtreeSize (end of left subtree in preorder)
- Update inEnd to rootIndex - 1 (end of left subtree in inorder)

For the right subtree:

- Update preStart to preStart + leftSubtreeSize + 1 (moving to the next element after the left subtree)
- Update preEnd to the original preEnd (end of right subtree in preorder)
- Update inStart to rootIndex + 1 (start of right subtree in inorder)

Step 7: Return the root node constructed for the current subtree. The function returns the root of the entire binary tree constructed from the preorder and inorder traversals.

```
#include <iostream>
#include <unordered_map>
#include <vector>
```

```cpp
#include <queue>
#include <map>

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to build a binary tree
    // from preorder and inorder traversals
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder){
        // Create a map to store indices
        // of elements in the inorder traversal
        map<int, int> inMap;

        // Populate the map with indices
        // of elements in the inorder traversal
        for(int i = 0; i < inorder.size(); i++){
            inMap[inorder[i]] = i;
        }

        // Call the private helper function
        // to recursively build the tree
        TreeNode* root = buildTree(preorder, 0, preorder.size()-1, inorder, 0, inorder.size()-1, inMap);

        return root;
    }

private:
    // Recursive helper function to build the tree
    TreeNode* buildTree(vector<int>& preorder, int preStart, int preEnd,
            vector<int>& inorder, int inStart, int inEnd, map<int, int>& inMap){
        // Base case: If the start indices
        // exceed the end indices, return NULL
        if(preStart > preEnd || inStart > inEnd){
            return NULL;
```

```cpp
        }

        // Create a new TreeNode with value
        // at the current preorder index
        TreeNode* root = new TreeNode(preorder[preStart]);

        // Find the index of the current root
        // value in the inorder traversal
        int inRoot = inMap[root->val];

        // Calculate the number of
        // elements in the left subtree
        int numsLeft = inRoot - inStart;

        // Recursively build the left subtree
        root->left = buildTree(preorder, preStart + 1, preStart + numsLeft,
                inorder, inStart, inRoot - 1, inMap);

        // Recursively build the right subtree
        root->right = buildTree(preorder, preStart + numsLeft + 1, preEnd,
                inorder, inRoot + 1, inEnd, inMap);

        // Return the current root node
        return root;
    }
};

// Function to print the
// inorder traversal of a tree
void printInorder(TreeNode* root){
    if(!root){
        return;
    }
    printInorder(root->left);
    cout << root->val << " ";
    printInorder(root->right);
}

// Function to print the
// given vector
void printVector(vector<int>&vec){
    for(int i = 0; i < vec.size(); i++){
        cout << vec[i] << " ";
    }
```

```cpp
        cout << endl;
}


int main() {
    vector<int> inorder = {9, 3, 15, 20, 7};
    vector<int> preorder = {3, 9, 20, 15, 7};

    cout << "Inorder Vector: ";
    printVector(inorder);

    cout << "Preorder Vector: ";
    printVector(preorder);

    Solution sol;

    TreeNode* root = sol.buildTree(preorder, inorder);

    cout << "Inorder of Unique Binary Tree Created: "<< endl;
    printInorder(root);
    cout << endl;


    return 0;
}
```

Time Complexity: O(N) where N is the number of nodes in the Binary Tree. This is because each node of the Binary Tree is visited once.

Space Complexity: O(N) where N is the number of nodes in the Binary Tree. The inorder hashmap to store the inorder array for fast lookup takes up space proportional to the input nodes. An auxiliary stack space ~ O(H) where H is the height of the Binary Tree is used. This is the stack space used to build the tree recursively. In the case of a skewed tree, the height of the tree will be H ~ N hence the worst case auxiliary space is O(N).

----

33) **Populating Next Right Pointers in Each Node**: You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:
```cpp
struct Node {
 int val;
 Node *left;
 Node *right;
 Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.
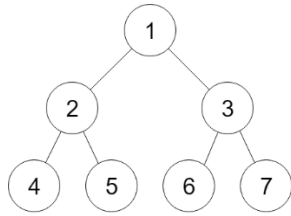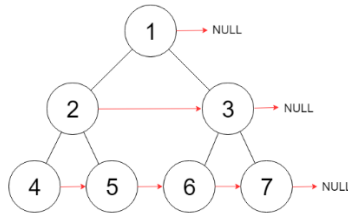Initially, all next pointers are set to NULL.

Figure A   Figure B

Input: root = [1,2,3,4,5,6,7]
Output: [1,#,2,3,#,4,5,6,7,#]
Explanation: Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.
Example 2:
Input: root = []
Output: []

```
  Node* connect(Node* root) {
    queue<Node*> q;
    if (root) q.push(root);
    while (q.size()) {
      int len = q.size();
      Node* curr;
      while (len--) {
        curr = q.front(), q.pop();
        curr->next = len ? q.front():NULL;
        if (curr->left) q.push(curr->left);
        if (curr->right) q.push(curr->right);
      }
    }
    return root;
  }
```

----

34) **Kth Smallest/largest Element in a BST**: Given a Binary Search Tree and an integer 'K'. Find and return the 'K-th' smallest and 'K-th' largest element in the given Binary Search Tree.

Example 1:Input: Binary Search Tree: 5 3 7 1 4 6 8 -1 2, K =3
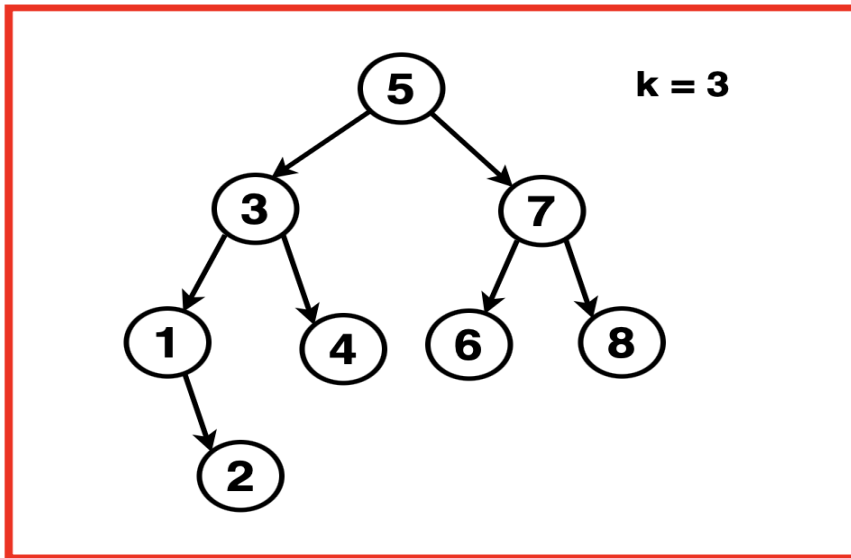

Output: 3rd smallest: 3, 3rd largest: 6**Explanation:** All the elements of the BST in the sorted order would be: [1, 2, 3, 4, 5, 6, 7, 8]. From this array is it evident that the index of the Kth smallest element would be K-1 and the index of the Kth largest element would be 1-K or (size of elements) - K. Hence 3rd smallest = 3 and 3rd largest = 6.**Example 2:Input:**Binary Search Tree: 8 5 12 4 7 10 14 -1 -1 6 -1 -1 -1 13, K = 4

Algorithm for Kth Smallest Element:
Step 1: Perform inorder traversal from the root node. At every visited node, increment a counter variable to keep track of visited nodes. Inorder Traversal: Traverse the left subtree, then current node then right subtree.
Step 2: When the counter reaches K, store the value of the current node as the Kth smallest.
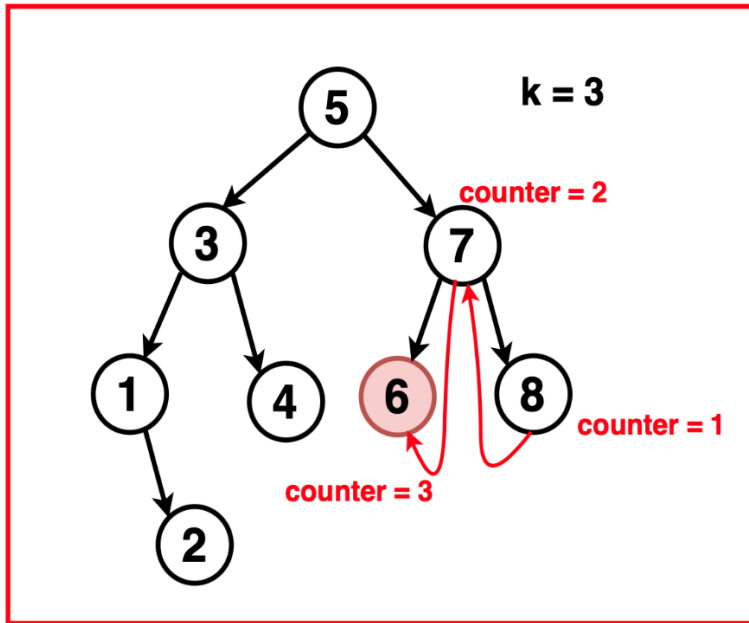**Step 3:** Return this value as the Kth smallest.
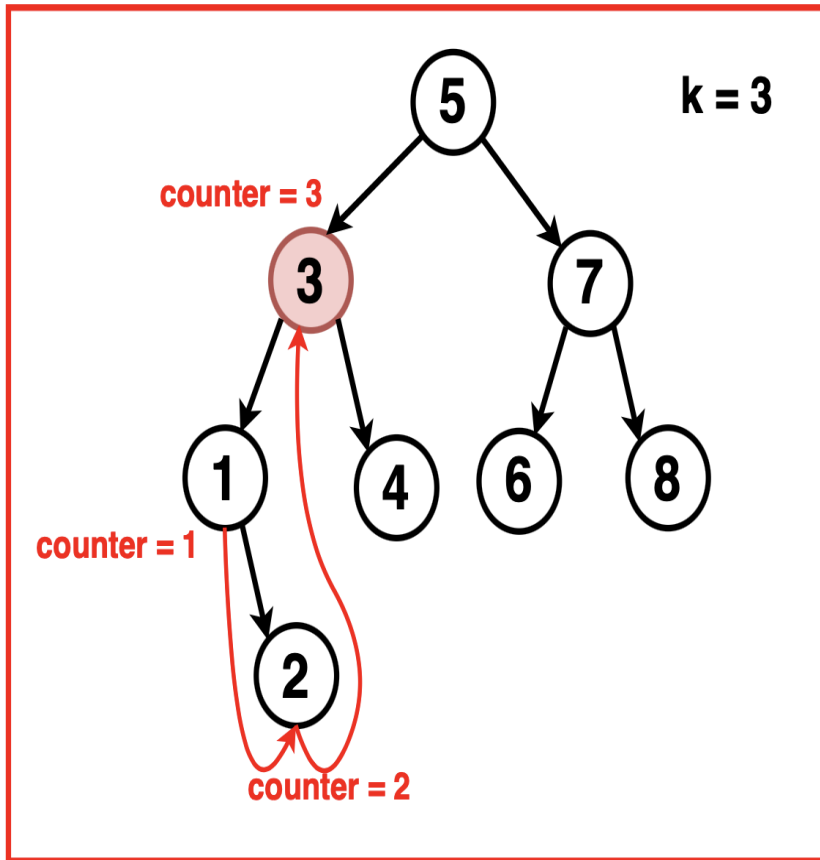
Algorithm for Kth Largest Element:

Step 1: Perform reverse inorder traversal from the root node. At every visited node, increment a counter variable to keep track of visited nodes. Traverse the right subtree, then current node then left subtree.

Step 2: When the counter reaches K, store the value of the current node as the Kth smallest.

Step 3: Return this value as the Kth largest.

```cpp
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

  // Helper function to perform reverse inorder
  // traversal to find Kth largest element
  void reverseInorder(TreeNode* node, int& counter, int k, int& kLargest) {
    if (!node || counter >= k) return;

    // Traverse right subtree
    reverseInorder(node->right, counter, k, kLargest);

    // Increment counter after visiting right subtree
    counter++;

    // Check if current node is the Kth largest
    if (counter == k)
  {
      kLargest = node->val;
      return;
  }

    // Traverse left subtree if Kth largest is not found yet
    reverseInorder(node->left, counter, k, kLargest);
```

```cpp
    }

    // Helper function to perform inorder
    // traversal to find Kth smallest element
    void inorder(TreeNode* node, int& counter, int k, int& kSmallest) {
        if (!node || counter >= k) return;

        // Traverse left subtree
        inorder(node->left, counter, k, kSmallest);

        // Increment counter after visiting left subtree
        counter++;

        // Check if current node is the Kth smallest
        if (counter == k) {
            kSmallest = node->val;
            return;
        }

        // Traverse right subtree if Kth smallest is not found yet
        inorder(node->right, counter, k, kSmallest);
    }

    pair<int, int> findKth(TreeNode* root, int k)
{
        int kSmallest = INT_MIN, kLargest = INT_MIN;
        // Counter to track visited nodes
        int counter = 0;

        // Find Kth smallest element
        // (perform inorder traversal)
        inorder(root, counter, k, kSmallest);

        // Reset counter for Kth largest element
        counter = 0;
        // Find Kth largest element
        // (perform reverse inorder traversal)
        reverseInorder(root, counter, k, kLargest);

        return make_pair(kSmallest, kLargest);
    }
```

```
int main() {

        // Creating a BST
        TreeNode* root = new TreeNode(10);
        root->left = new TreeNode(5);
        root->right = new TreeNode(13);
        root->left->left = new TreeNode(3);
        root->left->left->left = new TreeNode(2);
        root->left->left->right = new TreeNode(4);
        root->left->right = new TreeNode(6);
        root->left->right->right = new TreeNode(9);
        root->right->left = new TreeNode(11);
        root->right->right = new TreeNode(14);

        cout << "Binary Search Tree: "<< endl;
        printInOrder(root);
        cout << endl;

        // Find the Kth smallest and largest elements
        int k = 3;
        cout << "k: "<< k <<endl;
        pair<int, int> kthElements = findKth(root, k);

        cout << "Kth smallest element: " << kthElements.first << endl;
        cout << "Kth largest element: " << kthElements.second << endl;

        return 0;
}
```

Time Complexity: O(N) where N is the number of nodes in the Binary Search Tree as we traverse in inorder and reverse inorder fashion to get to the required nodes. We visit each node once resulting in time complexity proportional to the number of nodes in the BST.
Space Complexity : O(1) as no additional space is allocated or data structures used to store any values.

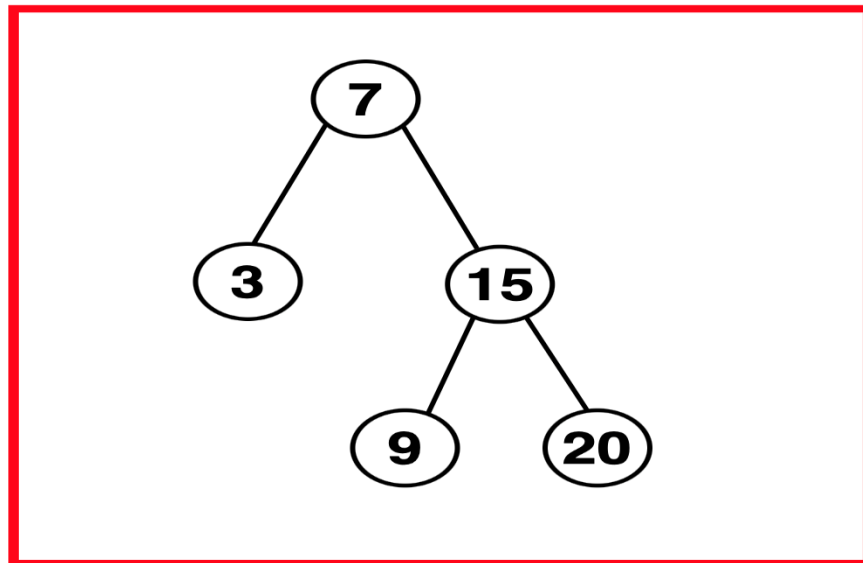
30) **Binary Search Tree Iterator**
Problem Statement: Implement the BSTIterator class that represents an iterator over the inorder traversal of a Binary Search Tree:
- **BSTIterator(TreeNode root):** Initialises an object of the BSTIterator class. The root of the BST is given as part of the constructor. The pointer should be initialised to a non-existent number smaller than any element in the BST.
- **boolean hasNext ( ):** Returns true if there exists a number in the traversal to the right of the pointer, otherwise returns false.
- **int next ( ):** Moves the pointer to the right, then returns the number at the pointer.

The first call to next ( ) will return the smallest element in the BST. Next ( ) calls are always valid, ie. there will be at least a next number in the in order traversal when next ( ) is called.

Example 1:Input: Binary Search Tree: 7 3 15 -1 -1 9 20**Functions Called:**
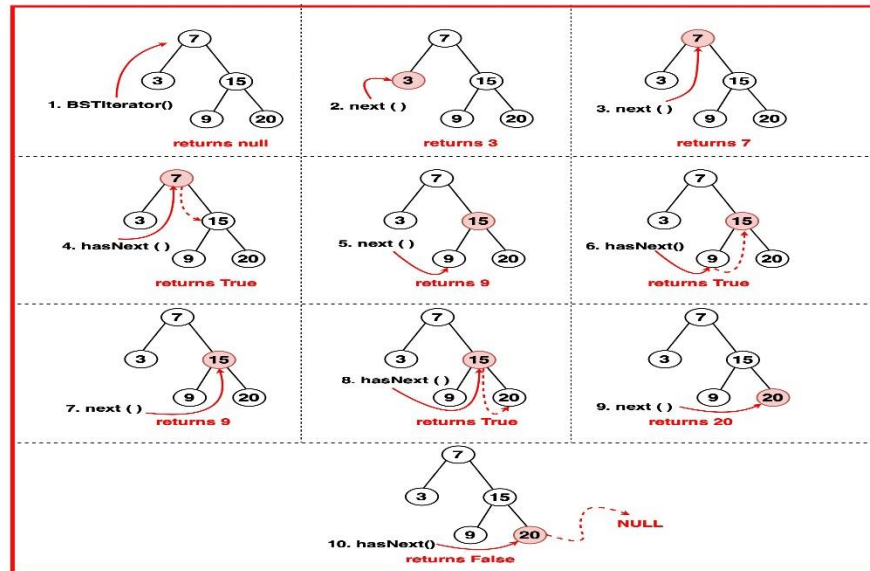
- BSTIterator()
- next()
- next()
- hasNext()
- next()
- hasNext()
- next()
- hasNext()
- next()
- hasNext()

Output:

- Null
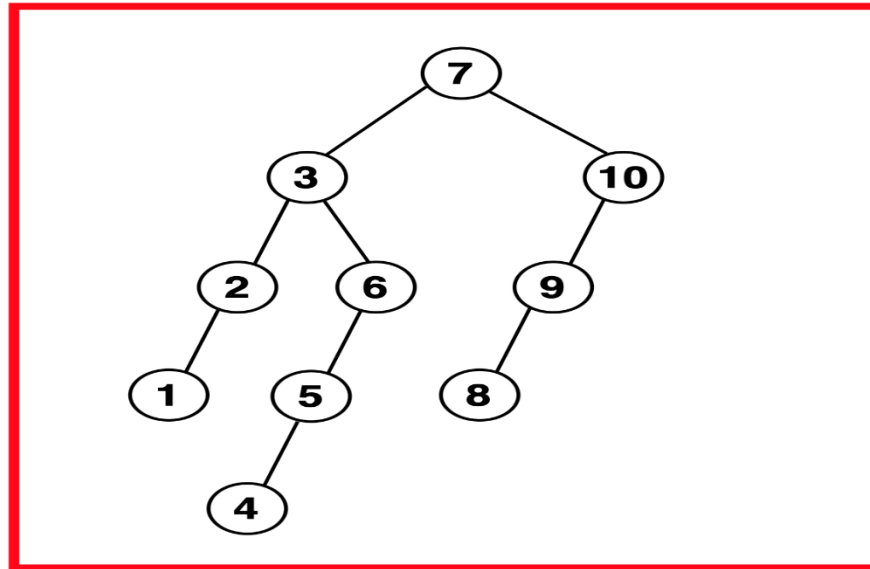- 3
- 7
- True
- 9
- True
- 15
- True
- 20
- False

Explanation:

Example 2:
Input: Binary Search Tree: 7 3 10 2 6 9 -1 1 -1 5 -1 8 -1 -1 -1 4 -1 -1 -1

Functions Called:

- BSTIterator()
- next()
- next()
- next()
- hasNext()
-
-         next()
- next()
- next()
- next()
- next()
- next()

Output :

- null
- 1
- 2
- 3
- True
- 4
- 5
- 6
- 7
- 8
- 9
- 10

Explanation:
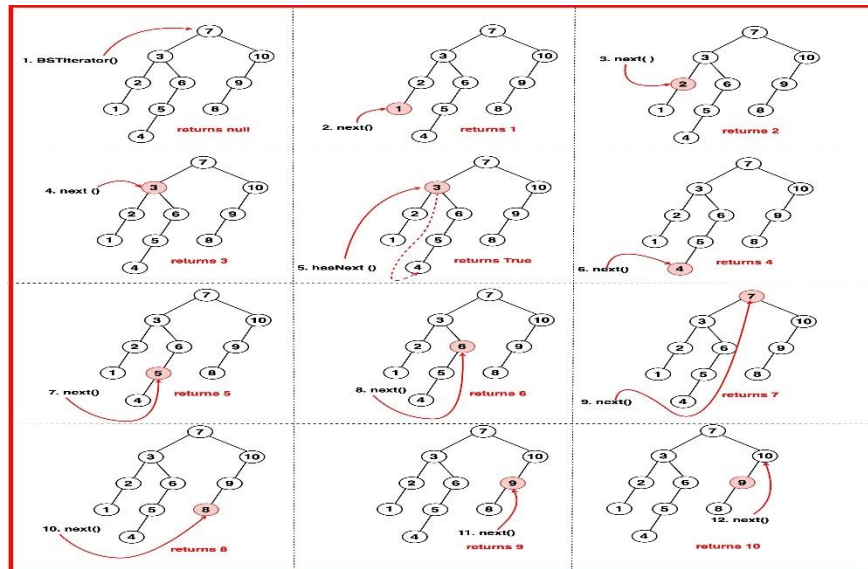
The previous approach uses O(N) space complexity that grows linearly with the number of nodes in the BST. This can be optimised to a space complexity of O(H) where H is the height of the tree and O(1) time complexity for the next() and hasNext() operations by leveraging the properties of a Binary Search Tree.

The optimised approach leverages Binary Search Tree (BST) properties to create an iterator with O(H) space complexity (where H is the tree height) and O(1) time complexity for next() and hasNext() operations.

By using a stack and performing an iterative traversal, the constructor initialises by navigating to the leftmost nodes and storing them in a stack. The next() function retrieves the top element, explores its right subtree, and adds left descendants to the stack.

hasNext() checks the stack for remaining elements to iterate over, signalling true if elements exist and false if the stack is empty.
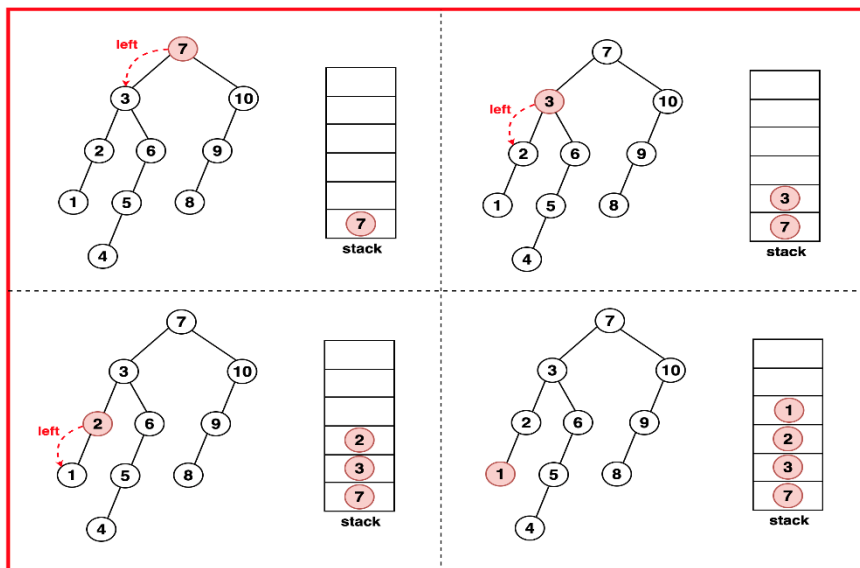
Algorithm

Step 1: Constructor BSTIterator(TreeNode root) Implementation:

- Utilise a stack data structure (Last In First Out) within the constructor.
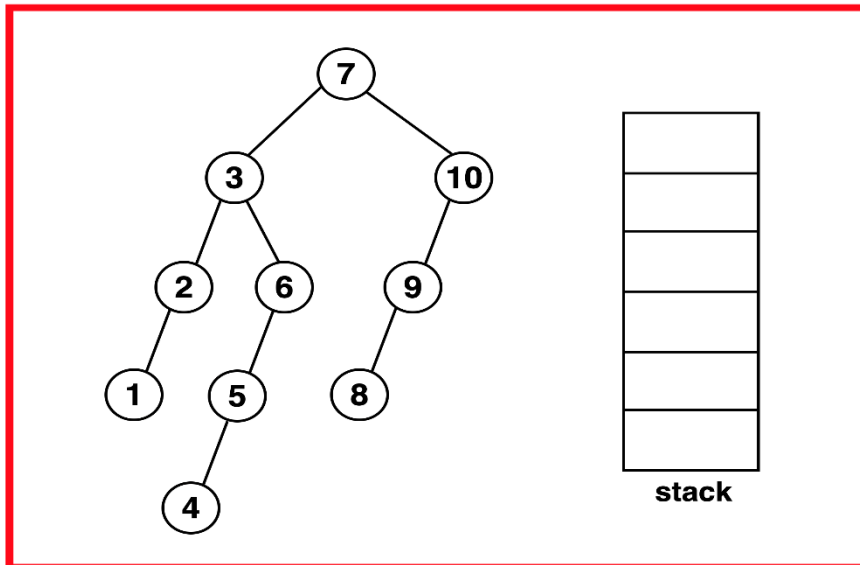
- Initially, traverse to the extreme left of the BST from the given root and push each node encountered onto the stack.


Step 2: next () function

- Upon calling next () pop the top element from the stack.
- Move to the right of the popped node and traverse down the left subtree of this right node, pushing encountered nodes onto the stack.
   Step 3: hasNext() function:
- Check if the stack is not empty, if the stack contains elements, it implies there are nodes that can be iterated over by next(), so return true.
- If the stack is empty, there are no more nodes to iterate over, hence return false.
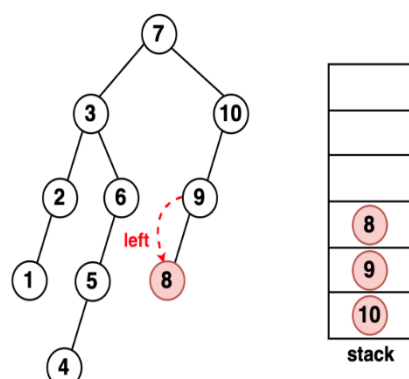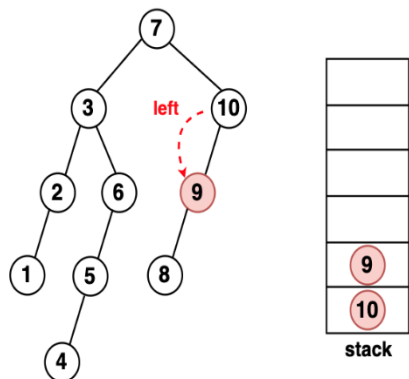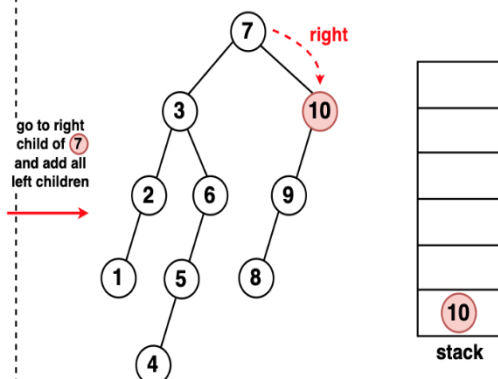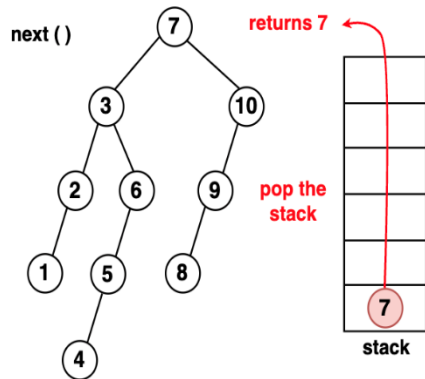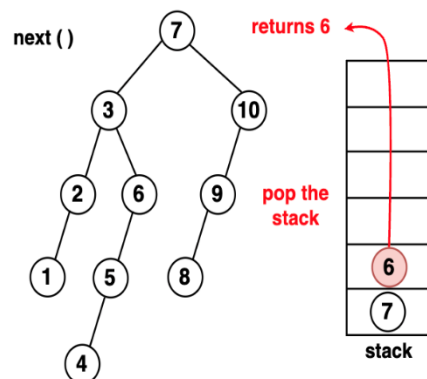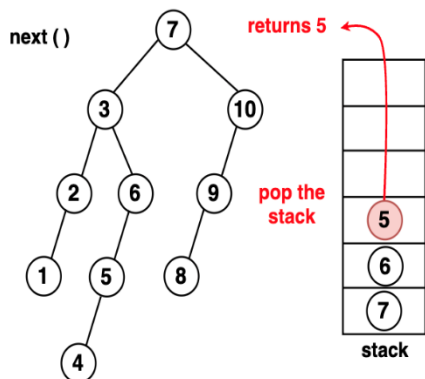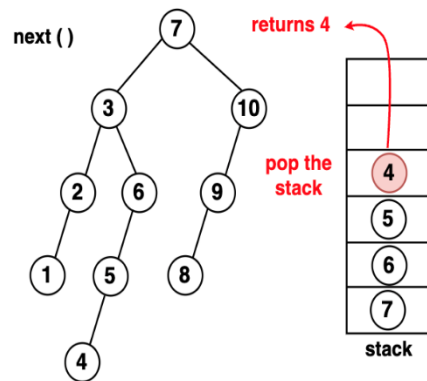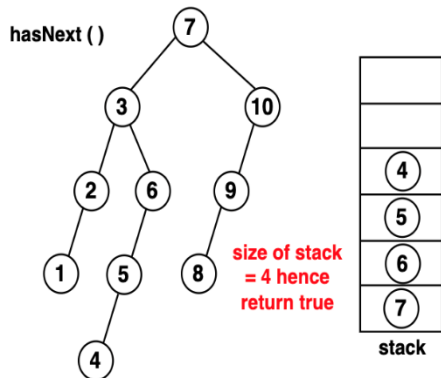
```cpp
#include <iostream>
#include <vector>
#include <climits>
#include <stack>
using namespace std;

// Definition of TreeNode structure
// for a binary tree node
struct TreeNode {
    // Value of the node
    int val;

    // Pointer to the left child node
    TreeNode* left;

    // Pointer to the right child node
    TreeNode* right;

    // Constructor to initialize the node with a
    // value and set left and right pointers to null
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class BSTIterator {
    // Private stack to store tree nodes
    private:
        stack<TreeNode *> myStack;

    public:
        // Constructor initializing the
        // BSTIterator with the root of the BST
        BSTIterator(TreeNode *root){
            // Initialize the stack with leftmost nodes
            pushAll(root);
        }

        // Checks if there is a next
        // smallest number in the BST
        bool hasNext(){
            // Returns true if stack is not empty
            return !myStack.empty();
        }
```

```cpp
    // Returns the next smallest number in the BST
    int next() {
       // Get the top node from stack
       TreeNode* tmpNode = myStack.top();
       // Remove the top node from stack
       myStack.pop();
       // Add leftmost nodes of the right subtree
       pushAll(tmpNode->right);
        // Return the value of the removed node
       return tmpNode->val;
    }

  private:
    // Pushes all the leftmost nodes starting
    // from the given node onto the stack
    void pushAll(TreeNode* node){
       // Iterate through left nodes,
       // pushing each onto the stack
       for(; node != NULL; myStack.push(node), node = node->left);
    }
};


// Function to perform an in-order traversal
// of a binary tree and print its nodes
void printInOrder(TreeNode* root) {
  // Check if the current node
  // is null (base case for recursion)
  if (root == nullptr) {
    // If null, return and
    // terminate the function
    return;
  }

  // Recursively call printInOrder
  // for the left subtree
  printInOrder(root->left);

  // Print the value of the current node
  cout << root->val << " ";

  // Recursively call printInOrder
  // for the right subtree
  printInOrder(root->right);
```

```cpp
}

int main() {
    // Create a sample binary search tree:
    // 7 3 15 -1 -1 9 20
    TreeNode* root = new TreeNode(7);
    root->left = new TreeNode(3);
    root->right = new TreeNode(15);
    root->right->left = new TreeNode(9);
    root->right->right = new TreeNode(20);

    cout << "Tree Initialised: ";
    printInOrder(root);
    cout << endl;

    // Create a BSTIterator object
    // initialized with the root of the BST
    BSTIterator bstIterator(root);


    // Function calls and their outputs
    cout << "Functions Called:" << endl;
    cout << "BSTIterator()" << endl;
    cout << "next(): " << bstIterator.next() << endl;
    cout << "next(): " << bstIterator.next() << endl;
    cout << "hasNext(): " << (bstIterator.hasNext() ? "true" : "false") << endl;
    cout << "next(): " << bstIterator.next() << endl;
    cout << "hasNext(): " << (bstIterator.hasNext() ? "true" : "false") << endl;
    cout << "next(): " << bstIterator.next() << endl;
    cout << "hasNext(): " << (bstIterator.hasNext() ? "true" : "false") << endl;
    cout << "next(): " << bstIterator.next() << endl;
    cout << "hasNext(): " << (bstIterator.hasNext() ? "true" : "false") << endl;

    return 0;
}
```

Time Complexity: O(1) as next() and hasNext() occur is constant time, the element pushed onto the stack during traversal to the leftmost node and during subsequent traversals will take O(H) time for each traversal.

Space Complexity : O(h) where H is the height of the tree as the additional space required to store the nodes will be the height of the tree at maximum.

-32) **Boundary Traversal of a Binary Tree:**

Given a Binary Tree, find its Boundary Traversal. The traversal should be in the following order:

1. **Left boundary nodes:** defined as the path from the root to the left-most node ie- the leaf node you could reach when you always travel preferring the left subtree over the right subtree.
2. **Leaf nodes:** All the leaf nodes except for the ones that are part of left or right boundary.
3. **Reverse right boundary nodes:** defined as the path from the right-most node to the root. The right-most node is the leaf node you could reach when you always travel preferring the right subtree over the left subtree. Exclude the root from this as it was already included in the traversal of left boundary nodes.

Note: If the root doesn't have a left subtree or right subtree, then the root itself is the left or right boundary.

Input:

```
    1
   / \
  2   3
 /\  /\
4  56  7
  /\
 8  9
```

Output: 1 2 4 8 9 6 7 3

Vertical Order Traversal of a Binary Tree:

Count BST nodes that lie in a given range

Given a Binary Search Tree (BST) and a range **l-h(inclusive),** count the number of nodes in the BST that lie in the given range.

- The values smaller than root go to the left side
- The values greater and equal to the root go to the right side

- **Input:**
-     10

-    / \
-   5 50
- / / \
- 1 40 100
- l = 5, h = 45
- **Output:** 3
- **Explanation:** 5 10 40 are the node in the
- range
- **Example 2:**
- **Input:**
-    5
-   / \
-   4 6
- / \
- 3   7
- l = 2, h = 8
- **Output:** 5
- **Explanation:** All the nodes are in the
- given range.