```
class Node
{
	public:
	int data;
	Node *left;
	Node *right;
};
```

//Function to find minimum in a tree.

```
Node* FindMin(Node* root)
{
	while(root->left != NULL)
	{
		root = root->left;
	}
	return root;
}
```

// Function to search a delete a value from tree.

```
Node* Delete(Node *root, int data)
{
	if(root == NULL)
	{
		return root;
	}
	else if(data < root->data)
	{
		root->left = Delete(root->left,data);
	}
	else if (data > root->data)
	{
		root->right = Delete(root->right,data);
	}
	else
	{
		// Case 1:  No child
		if(root->left == NULL && root->right == NULL)
		{
			delete root;
			root = NULL;
		}
		//Case 2: One child
		else if(root->left == NULL)
		{
			Node *temp = root;
			root = root->right;
			delete temp;
		}
```

```
		else if(root->right == NULL)
		{
			Node *temp = root;
			root = root->left;
			delete temp;
		}
		// case 3: 2 children
		else
		{
			Node *temp = FindMin(root->right);
			root->data = temp->data;
			root->right = Delete(root->right,temp->data);
		}
	}
	return root;
}
```

//Function to visit nodes in Inorder

```
void Inorder(Node *root)
{
	if(root == NULL)
	{
		return;
	}

	Inorder(root->left);       //Visit left subtree
	printf("%d ",root->data);  //Print data
	Inorder(root->right);      // Visit right subtree
}
```

//Function to visit nodes in Preorder

```
void Preorder(Node *root)
{
	if(root == NULL)
	{
		return;
	}

	//Visit left subtree
	printf("%d ",root->data);  //Print data
	Preorder(root->left);
	Preorder(root->right);     // Visit right subtree
}
```

//Function to visit nodes in Postorder

```
void Postorder(Node *root)
{
	if(root == NULL)
```

```
	{
		return;
	}

	Postorder(root->left);      //Visit left subtree
	Postorder(root->right);     // Visit right subtree
	printf("%d ",root->data);   //Print data
}
```

// Function to Insert Node in a Binary Search Tree

```
Node* Insert(Node *root,char data)
{
	if(root == NULL)
	{
		root = new Node();
		root->data = data;
		root->left = root->right = NULL;
	}
	else if(data <= root->data)
	{
		root->left = Insert(root->left,data);
	}
	else
	{
		root->right = Insert(root->right,data);
	}
	return root;
}
```

//To search an element in BST, returns true if element is found

```
bool Search(Node* root,int data)
{
	if(root == NULL)
	{
		return false;
	}
	else if(root->data == data)
	{
		return true;
	}
	else if(data <= root->data)
	{
		return Search(root->left,data);
	}
	else
	{
		return Search(root->right,data);
	}
```

```
}

int main()
{
	/*Code To Test the logic
	  Creating an example tree
	                    5
	                  /   \
	                 3     10
	                / \      \
	               1   4     11

   */
	Node* root = NULL;
	root = Insert(root,5); root = Insert(root,10);
	root = Insert(root,3); root = Insert(root,4);
	root = Insert(root,1); root = Insert(root,11);
	//level order traversal
	//bfs(root);
	// Deleting node with value 5, change this value to test other cases
	//root = Delete(root,5);

	// Ask user to enter a number.
	int number;
	cout<<"Enter number be searched\n";
	cin>>number;
	//If number is found, print "FOUND"
	if(Search(root,number) == true)
	{
		cout<<"Found\n";
	}
	else
	{
		cout<<"Not Found\n";
	}
	int min_val = findMinimum(root);
	cout<<"minimum value in tree "<<min_val<<endl;
	int max_val = findMaximum(root);
	cout<<"maximum value in tree "<<max_val<<endl;
	//Print Nodes in Inorder
	cout<<"Inorder: ";
	Inorder(root);
	int height = treeHeight(root);
	cout<<"tree height"<<height<<endl;
	cout<<"Preorder: ";
```

```
	Preorder(root);
	cout<<"Postorder: ";
	Postorder(root);
	cout<<"\n";
}
```

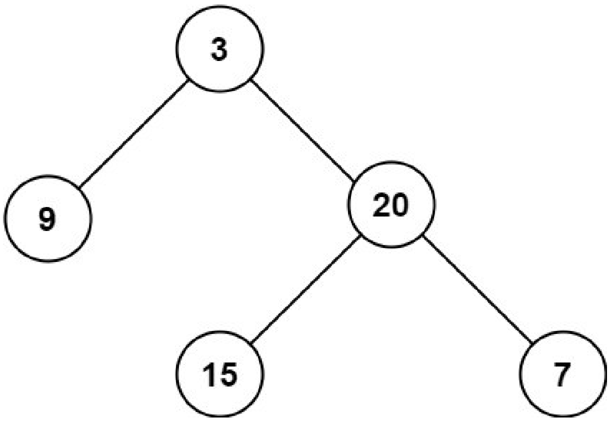
******************Tree questions****************************

1) Maximum Depth/Height of Binary Tree

Given the `root` of a binary tree, returnits 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.

Example 1:




```
int treeHeight(Node* root)
{
	if(root == NULL)
```

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

Output: 3

```
	{
		return 0;
	}
	else
	{
		int ltreeHeight = treeHeight(root->left);
		int rtreeHeight = treeHeight(root->right);
		return max(ltreeHeight,rtreeHeight)+1;
	}

}
```

----

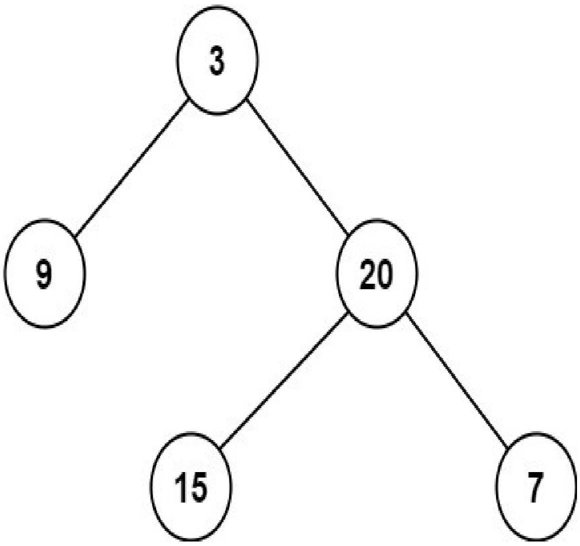
2) Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Note:A leaf is a node with no children.

Example 1:



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

Output: 2

```
int minDepth(TreeNode* root)
{
	//Base case:
	if(root == nullptr)
	{
		return 0;
	}
	//find min from both left subtree and right subtree
	int minLeft = minDepth(root->left);
	int minRight = minDepth(root->right);

	//handle skewed tree: either left subtree is null or right subtree is null
	if(minLeft ==0 || minRight == 0)
	{
		return 1+ max(minLeft, minRight);
	}
	return 1+ min(minLeft, minRight);

}
```

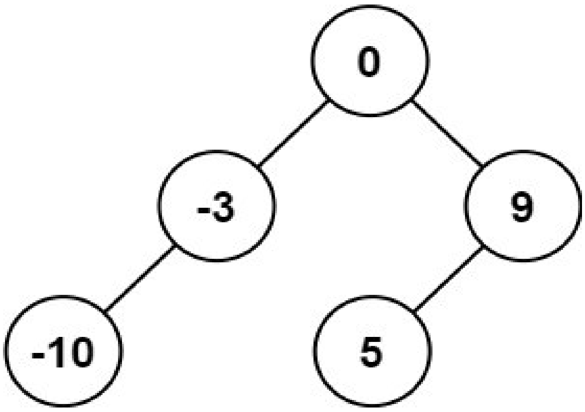
----

3) 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.

A **height-balanced** binary tree is a binary tree in which the depth of the two subtrees of every node never differs by more than one.

Example 1:



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:

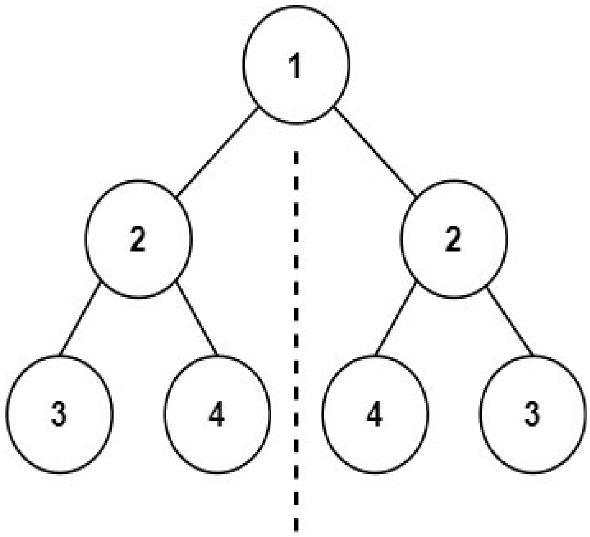
```
TreeNode* sortedArrayToBST(vector<int>& nums)
{
        int n = nums.size();

        return TreeCreator(nums, 0, n-1);

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

----

4) Symmetric Tree

Given the `root` of a binary tree, check whether it is a mirror of itself(i.e., symmetric around its center).

Example 1:



Input: root = [1,2,2,3,4,4,3]

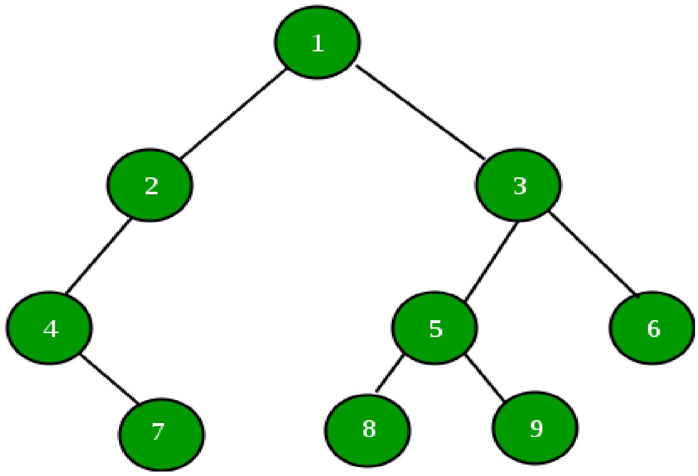
Output: true

```
bool solve(Node * r1, Node * r2)
{
        if(r1 == NULL && r2 == NULL)
        {
                return true;
        }

        else if(r1 == NULL || r2 == NULL || r1->val != r2->val)
        {
                return false;
        }

        return solve(r1->left, r2->right) && solve(r1->right, r2->left);
}

bool isSymmetric(Node* root)
{
        return solve(root->left, root->right);
}
```

----

5) Find maximum in BST:

Given a Binary Tree, find the maximum(or minimum) element in it. For example, maximum in the following Binary Tree is 9.




```
int findMaximum(Node* root)
{
	if(root == NULL)
	{
		return -1;
	}
	else if((root->right)== NULL)
	{
		return root->data;
	}
	return findMaximum(root->right);
}
```

----

6) Find Minimum of BST:

```
int findMinimum(Node* root)
{
	if(root == NULL)
	{
		return -1;
	}
	else if((root->left)== NULL)
	{
		return root->data;
	}
	return findMinimum(root->left);
}
```

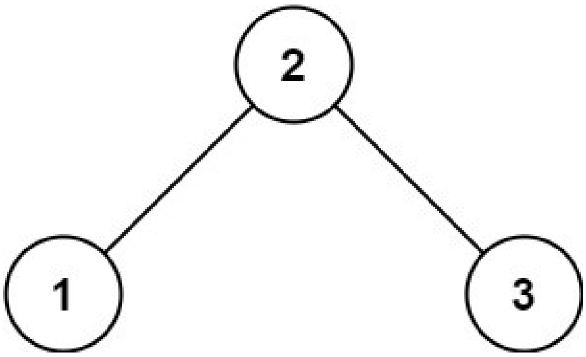
----

7) Validate Binary Search Tree

A **valid BST** is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:



Input: root = [2,1,3]

Output: true

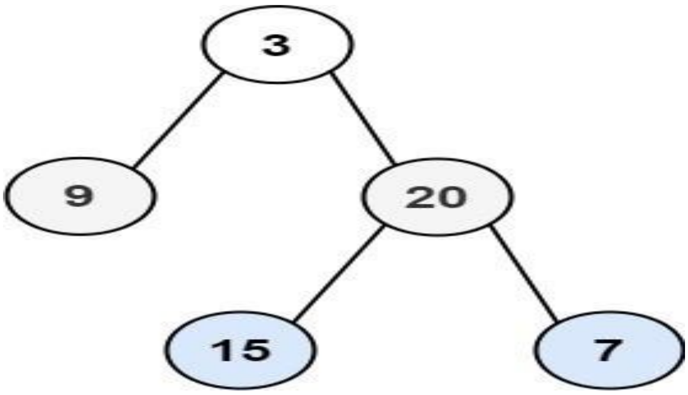
```
bool isValidBST(TreeNode* root, long min = LONG_MIN, long max = LONG_MAX)
{
	//Base case:
	if (root == NULL)
	{
		return true;
	}
	//return false, if root->val is less than infinity or greater than infinity.
	if (root->val <= min || root->val >= max)
	{
		return false;
	}
	//leftsubtree value is b/w -∞ and root. right sub tree value is b/w root and ∞.
	return isValidBST(root->left, min, root->val) && isValidBST(root->right, root->val,
max);
}
```

----

8) 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:



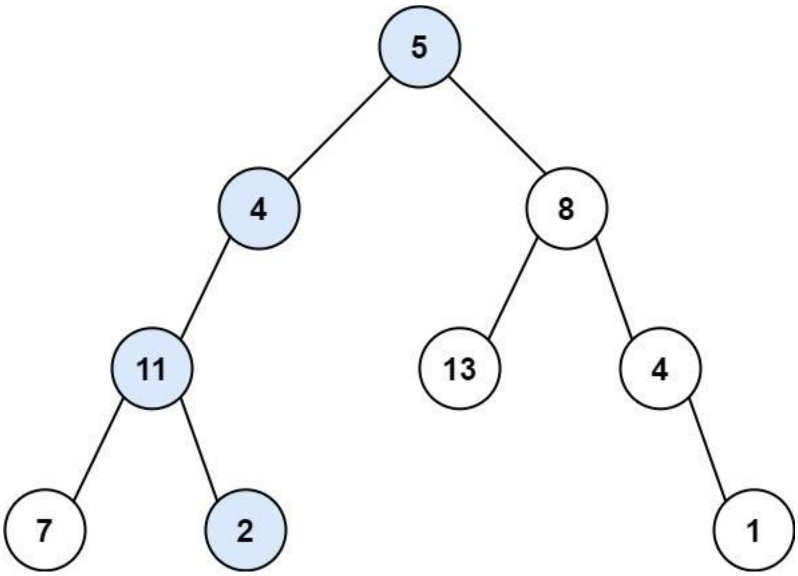
Input: root = [3,9,20,null,null,15,7]

Output: [[3],[9,20],[15,7]]

```
void bfs(Node* root)
{
	if(root == NULL)
	{
		return;
	}
	else
	{
		queue<Node*>Q;
		Q.push(root);
		while(!Q.empty())
		{
			Node* current = Q.front();
			cout<<current->data<<endl;
			if(current->left != NULL)
			{
				Q.push(current->left);
			}
			if(current->right != NULL)
			{
				Q.push(current->right);
			}
			Q.pop();
		}
	}
}
```

----

9) Path Sum:



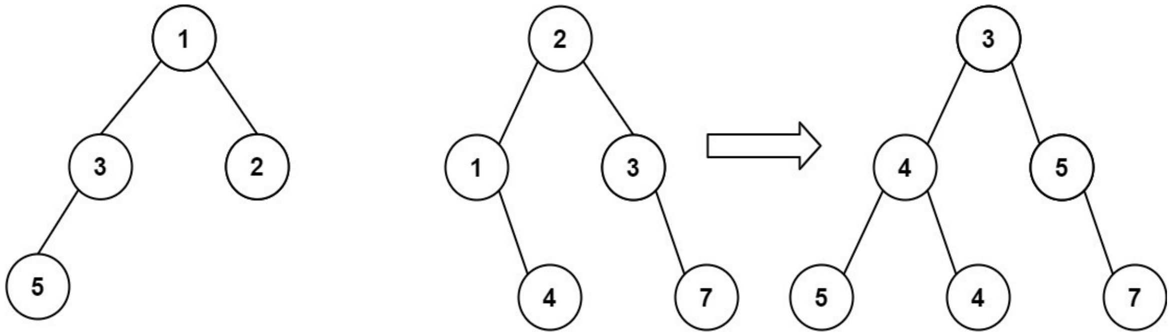

```
bool hasPathSum(TreeNode* root, int targetSum)
{
	if(root==NULL) return false;
```

```
        if(root->left==NULL && root->right==NULL) return targetSum==root->val;
        return hasPathSum(root->left,targetSum-root->val) || hasPathSum(root-
>right,targetSum-root->val);
}
```

---

10)Merge Two Binary Trees

Example 1:



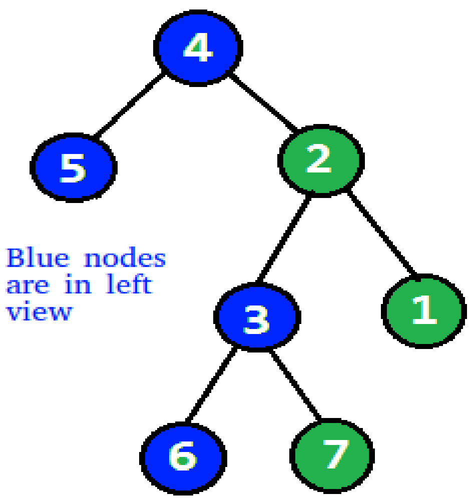
Input: root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]

Output: [3,4,5,5,4,null,7]

```
TreeNode* mergeTrees(TreeNode* r1, TreeNode* r2)
{
        //Base case:
        if(r1==NULL)return r2;
        if(r2==NULL)return  r1;

        //add value of first list and second list
        r1->val = r1->val + r2->val;
        r1->left = mergeTrees(r1->left,r2->left);
        r1->right = mergeTrees(r1->right,r2->right);
        return r1;
}
```

---

11)Print Left View of a Binary Tree



```
vector<int> leftView(Node *root)
{
	if(!root)
	{
		return {};
	}
	vector<int> v; //store values of nodes in the rightmost
	queue<Node*> Q; //store node type values in queue
	Q.push(root); //push root
	while(!Q.empty())
	{ //repeat steps until queue is not empty

		int size = Q.size();  // current size of queue
		for(int i = 0; i < size; i++)
		{
			Node* t = Q.front(); //declare a temp node and put front node of queue
			Q.pop();
			if(i==0)
			{   //if node is rightmost
				v.push_back(t->data); //push the value of rightmost node into
vector
			}
			if(t->left)
			{ //if temp->left != NULL then push into queue
				Q.push(t->left);
			}
			if(t->right)
			{
				//if temp->right != NULL then push into queue
				Q.push(t->right);
			}
		}
	}
}
return v; //finally we have all values

}
```