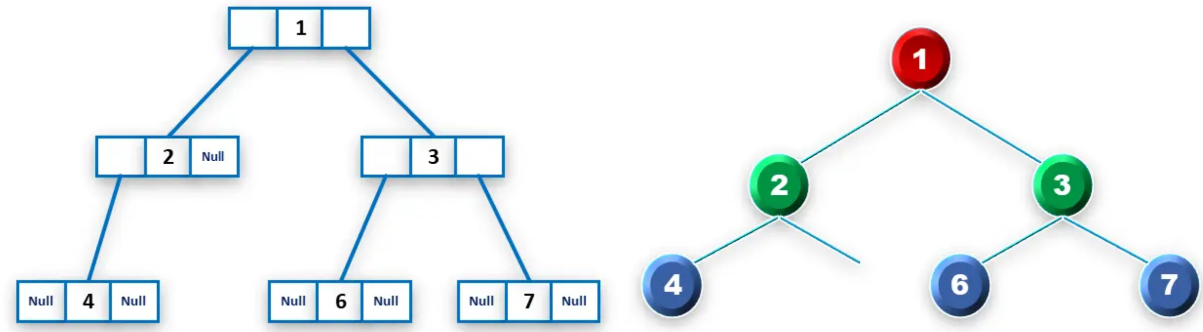
TREES CONCEPTS

Types of Binary Trees in DSA

1. **Full Binary Tree**: Each node has either 0 or 2 children.
2. **Complete Binary Tree**: All levels are completely filled except possibly for the last level, which is filled from left to right.
3. **Perfect Binary Tree**: All internal nodes have exactly two children and all leaf nodes are at the same level.
4. **Balanced Binary Tree**: The height of the left and right subtrees of any node differ by at most one.
5. **Degenerate Tree (Skewed Tree)**: Every parent node has only one child. It can be either left-skewed or right-skewed.
6. **Binary Search Tree (BST)**: A binary tree where the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree only nodes with keys greater than the node's key.

Representation of a Binary tree



Linked Representation of Binary Tree

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};
int main() {
    // Creating a simple binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
```

```
    root->left->right = new Node(5);
    return 0;
}
```

Tree Traversals(BFS/DFS)

Breadth-First Search (BFS)

Tc → O(N) Sc→ O(N)

BFS traverses the tree level by level, starting from the root node. It uses a queue to keep track of nodes at the current level before moving to the next level. Here is an example implementation in C++:

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

Initialize:

- `result = []`
- `queue q = [1]`

Iterations:

1. **Level 1**:
    - **Level Size**: `1`
    - **Current Level**: `[]`
    - Dequeue `1`, visit it, add its children `2` and `3` to the queue.
        - `currentLevel = [1]`

    - `queue q = [2, 3]`
  - Add `currentLevel` to `result`.
    - `result = [[1]]`
2. **Level 2**:
  - **Level Size**: `2`
  - **Current Level**: `[]`
  - Dequeue `2`, visit it, add its children `4` and `5` to the queue.
    - `currentLevel = [2]`
    - `queue q = [3, 4, 5]`
  - Dequeue `3`, visit it (no children to add).
    - `currentLevel = [2, 3]`
    - `queue q = [4, 5]`
  - Add `currentLevel` to `result`.
    - `result = [[1], [2, 3]]`
3. **Level 3**:
  - **Level Size**: `2`
  - **Current Level**: `[]`
  - Dequeue `4`, visit it (no children to add).
    - `currentLevel = [4]`
    - `queue q = [5]`
  - Dequeue `5`, visit it (no children to add).
    - `currentLevel = [4, 5]`
    - `queue q = []`
  - Add `currentLevel` to `result`.
    - `result = [[1], [2, 3], [4, 5]]`

Final Result:

- `result = [[1], [2, 3], [4, 5]]`

```
vector<vector<int>> bfsLevelOrder(Node* root) {
    vector<vector<int>> result;
    if (root == nullptr) return result;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        vector<int> currentLevel;
        for (int i = 0; i < levelSize; ++i) {
            Node* current = q.front();
            q.pop();
            currentLevel.push_back(current->data);

            if (current->left != nullptr) q.push(current->lef
t);
            if (current->right != nullptr) q.push(current->ri
ght);
        }
        result.push_back(currentLevel);
    }

    return result;
}
```

Depth-First Search (DFS)

Tc → O(N) Sc→ O(N)

DFS traverses the tree by exploring as far as possible along each branch before backtracking. DFS can be further divided into three types: Preorder, Inorder, and

Postorder.

Preorder Traversal(Recursion) → Root-Left-Right

Visit the root node, then the left subtree, and finally the right subtree.

```
void preorder(Node* root) {
    if (root == nullptr) return;

    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}
```

Inorder Traversal(Recursion)→ Left-Root-Right

Visit the left subtree, the root node, and finally the right subtree.

```
void inorder(Node* root) {
    if (root == nullptr) return;

    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}
```

Postorder Traversal(Recursion) → Left-Right-Root

Visit the left subtree, the right subtree, and finally the root node.

```
void postorder(Node* root) {
    if (root == nullptr) return;

    postorder(root->left);
    postorder(root->right);
```

```
    cout << root->data << " ";
}
```

Preorder Traversal(Iterative) → Root-Left-Right

Let's walk through a dry run of the iterative preorder traversal function `preorderIterative` using a simple binary tree:

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

1. Initialize:
    - `result` = []
    - `stack` = [1]
2. Iterations:
    - Pop 1, push 3 and 2. `result` = [1], `stack` = [3, 2]
    - Pop 2, push 5 and 4. `result` = [1, 2], `stack` = [3, 5, 4]
    - Pop 4. `result` = [1, 2, 4], `stack` = [3, 5]
    - Pop 5. `result` = [1, 2, 4, 5], `stack` = [3]
    - Pop 3. `result` = [1, 2, 4, 5, 3], `stack` = []

Final result: `[1, 2, 4, 5, 3]`.

```
vector<int> preorderIterative(Node* root) {
    vector<int> result;
    if (root == nullptr) return result;

    stack<Node*> s;
    s.push(root);
```

```
    while (!s.empty()) {
        Node* current = s.top();
        s.pop();
        result.push_back(current->data);

        if (current->right != nullptr) s.push(current->righ
t);
        if (current->left != nullptr) s.push(current->left);
    }

    return result;
}
```

Inorder Traversal(Iterative) → Left-Root-Right

Let's walk through a dry run of the iterative inorder traversal function `inorderIterative` using a simple binary tree:

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

1. Initialize:
    - `result` = []
    - `stack` = []
    - `current` = 1
2. Iterations:
    - Push 1, move to 2. `stack` = [1]
    - Push 2, move to 4. `stack` = [1, 2]
    - Push 4, move to null. `stack` = [1, 2, 4]

- Pop 4, move to null. `result` = [4], `stack` = [1, 2]
- Pop 2, move to 5. `result` = [4, 2], `stack` = [1]
- Push 5, move to null. `stack` = [1, 5]
- Pop 5, move to null. `result` = [4, 2, 5], `stack` = [1]
- Pop 1, move to 3. `result` = [4, 2, 5, 1], `stack` = []
- Push 3, move to null. `stack` = [3]
- Pop 3, move to null. `result` = [4, 2, 5, 1, 3], `stack` = []

Final result: `[4, 2, 5, 1, 3]`.

```
vector<int> inorderIterative(Node* root) {
    vector<int> result;
    stack<Node*> s;
    Node* current = root;

    while (true) {
        // Traverse to the leftmost node
        if (current != nullptr) {
            s.push(current);
            current = current->left;
        }
                else{
                if(s.empty() == true) break;
        // Process the node and move to the right subtree
        current = s.top();
        s.pop();
        result.push_back(current->data);
        current = current->right;
        }
    }
    return result;
}
```

PostOrder Traversal (Iterative 2 Stacks) → Left-Right-Root

> Observe properly and revise it

Tc → O(N) Sc→ O(2N) — because of 2stack

Let's walk through a dry run of the iterative postorder traversal function `postorderIterative` using two stacks on a simple binary tree:

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

1. Initialize:
    - `result` = []
    - `stack1` = [1]
    - `stack2` = []
2. Iterations:
    - Pop 1 from `stack1`, push to `stack2`, push 2 and 3 to `stack1`. `stack1` = [2, 3], `stack2` = [1]
    - Pop 3 from `stack1`, push to `stack2`. `stack1` = [2], `stack2` = [1, 3]
    - Pop 2 from `stack1`, push to `stack2`, push 4 and 5 to `stack1`. `stack1` = [4, 5], `stack2` = [1, 3, 2]
    - Pop 5 from `stack1`, push to `stack2`. `stack1` = [4], `stack2` = [1, 3, 2, 5]
    - Pop 4 from `stack1`, push to `stack2`. `stack1` = [], `stack2` = [1, 3, 2, 5, 4]
3. Final result:
    - Pop all elements from `stack2` to get the postorder traversal: `[4, 5, 2, 3, 1]`.

```
vector<int> postorderIterative(Node* root) {
    vector<int> result;
```

```
    if (root == nullptr) return result;

    stack<Node*> stack1, stack2;
    stack1.push(root);

    while (!stack1.empty()) {
        root = stack1.top();
        stack1.pop();
        stack2.push(root);

        // Push left and right children to stack1
        if (root->left != nullptr) stack1.push(root->left);
        if (root->right != nullptr) stack1.push(root->right);
    }

    // Collect nodes from stack2 for postorder result
    while (!stack2.empty()) {
        result.push_back(stack2.top()->data);
        stack2.pop();
    }

    return result;
}
```

PostOrder Traversal (Iterative 1 Stack) → Left-Right-Root

> Do revise Again →striver L12

Tc → O(2N) Sc→ O(N)

Let's walk through a dry run of the iterative postorder traversal function `postorderIterativeSingleStack` using one stack on a simple binary tree:

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

1. Initialize:
    - `result` = []
    - `stack` = []
    - `current` = 1
2. Iterations:
    - Push 1, move to 2. `stack` = [1]
    - Push 2, move to 4. `stack` = [1, 2]
    - Push 4, move to null. `stack` = [1, 2, 4]
    - Pop 4, visit it, move to null. `result` = [4], `stack` = [1, 2]
    - Pop 2, move to 5. `result` = [4], `stack` = [1]
    - Push 5, move to null. `stack` = [1, 5]
    - Pop 5, visit it, move to null. `result` = [4, 5], `stack` = [1]
    - Pop 2, visit it, move to null. `result` = [4, 5, 2], `stack` = [1]
    - Pop 1, move to 3. `result` = [4, 5, 2], `stack` = []
    - Push 3, move to null. `stack` = [3]
    - Pop 3, visit it, move to null. `result` = [4, 5, 2, 3], `stack` = []
    - Pop 1, visit it, move to null. `result` = [4, 5, 2, 3, 1], `stack` = []

Final result: `[4, 5, 2, 3, 1]`.

```
vector<int> postorderIterativeSingleStack(Node* root) {
    vector<int> result;
    if (root == nullptr) return result;
```

```
    stack<Node*> s;
    Node* temp = nullptr;
    Node* current = root;

    while (!s.empty() || current != nullptr) {
        if (current != nullptr) {
            // Push nodes to the stack until reaching the lef
tmost node
            s.push(current);
            current = current->left;
        } else {
            Node* topNode = s.top();
            temp = topNode->right;
            if(temp==null){
                temp = topNode;
                s.pop();
                result.push_back(temp);
                while(!s.empty() && temp == topNode->right){
                    temp = topNode;
                    s.pop();
                    result.push_back(temp->data);
                }
            }
            else{
                cur = temp;
            }

        }
    }
    return result;
}
```

Preorder, Inorder, and Postorder Traversal (Iterative with One Stack)

striver L13

Tc → O(N) Sc→ O(N)

```
     1
    / \
   2   3
```

1. Initialize:
   - `pre` = []
   - `in` = []
   - `post` = []
   - `stack` = [(1, 1)]
2. Iterations:
   - Pop (1, 1), visit 1 (preorder), push 2, push back (1, 2). `pre` = [1], `stack` = [(1, 2), (2, 1)]
   - Pop (2, 1), visit 2 (preorder), push null, push back (2, 2). `pre` = [1, 2], `stack` = [(1, 2)]
   - Pop (2, 2), visit 2 (inorder), push null, push back (2, 3). `in` = [2], `stack` = [(1, 2)]
   - Pop (2, 3), visit 2 (postorder). `post` = [2], `stack` = [(1, 2)]
   - Pop (1, 2), visit 1 (inorder), push 3, push back (1, 3). `in` = [2, 1], `stack` = [(1, 3), (3, 1)]
   - Pop (3, 1), visit 3 (preorder), push null, push back (3, 2). `pre` = [1, 2, 3], `stack` = [(1, 3)]
   - Pop (3, 2), visit 3 (inorder), push null, push back (3, 3). `in` = [2, 1, 3], `stack` = [(1, 3)]
   - Pop (3, 3), visit 3 (postorder). `post` = [2, 3], `stack` = [(1, 3)]
   - Pop (1, 3), visit 1 (postorder). `post` = [2, 3, 1], `stack` = []

Final results:

- `pre` = [1, 2, 3]
- `in` = [2, 1, 3]
- `post` = [2, 3, 1]

```
vector<int> preInPostTraversal(TreeNode* root){
    stack<pair<TreeNode*,int>> s;
    st.push({root,1});
    if(root == nullptr) return;
    vector<int> pre,in,post;

    while(!s.empty()){
        auto it = s.top();
        s.pop();

        if(it.second == 1){
            pre.push_back(it.first ->val);
            it.second++;
            if(it.first->left!=NULL){
            s.push({it.first->left,1});
            }
        }

        else if(it.second == 2){
            in.push_back(it.first ->val);
            it.second++;
            if(it.first->right != NULL){
                s.push({it.first->right,1});
            }
        }
        else{
            post.push_back(it.first->val);
        }
    }
```

```
    //return pre/in/post
}
```

Maximum Depth/Height of Binary tree

[Maximum Depth of Binary Tree - LeetCode]

☐ Done

Tc → O(N) Sc→ O(N)

recurrence → 1 + max (left, right) // main logic

```
int maxDepth(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }
    int leftDepth = maxDepth(root->left);
    int rightDepth = maxDepth(root->right);
    return 1+max(leftDepth, rightDepth);
}
```

Check For Balanced Binary Tree

[Balanced Binary Tree - LeetCode]

☐ Done

Tc → O(N) Sc→ O(N)

```
bool checkBalance(TreeNode* root){
  return maxDepth(root) != -1;
}

int maxDepth(TreeNode* root) {
    if (root == nullptr) {
        return 0;
```

```
    }
    int leftDepth = maxDepth(root->left);
    if(leftDepth==-1) return -1;   //addition
    int rightDepth = maxDepth(root->right);
    if(rightDepth==-1) return -1;  //addition
    return (abs(leftDepth-rightDepth) > 1) return -1;  //additi
    return 1+max(leftDepth, rightDepth);
}
```

Diameter For Balanced Binary Tree

[Diameter of Binary Tree - LeetCode]()

- [ ] Done

Tc → O(N) Sc→ O(N)

```
bool diameterOfbinaryTree(TreeNode* root){
     int maxi = 0;
     maxDepth(root,maxi);
     return maxi;
}

int maxDepth(TreeNode* root,int &maxi) {
    if (root == nullptr) {
        return 0;
    }
    int leftDepth = maxDepth(root->left,maxi);
    int rightDepth = maxDepth(root->right,maxi);
     maxi = max(maxi,leftDepth+rightDepth); //addition
    return 1+max(leftDepth, rightDepth);
}
```

Maximum path For Binary Tree

[Binary Tree Maximum Path Sum - LeetCode]()

> please revise again its hard

- [ ] Done

Tc → O(N) Sc→ O(N)

```
int maxPathSum(TreeNode* root){
     int maxi = INT_MIN;
     maxDepth(root,maxi);
     return maxi;
}

int maxDepth(TreeNode* root,int &maxi) {
    if (root == nullptr) {
        return 0;
    }
    int leftDepth = max(0,maxDepth(root->left,maxi));
    int rightDepth = max(0,maxDepth(root->right,maxi));
     maxi = max(maxi,leftDepth+rightDepth+root->val); //additio
    return root->val+max(leftDepth, rightDepth);
}
```

Check if two trees are Identical

[Same Tree - LeetCode]()

- [x] ~~Done~~

Tc → O(N) Sc→ O(N)

```
   bool isSameTree(TreeNode* p, TreeNode* q) {
        if(p==nullptr|| q==nullptr){
         return (p==q);
        }
        return isSameTree(p->left,q->left)  && (p->val==q->val) &
    }
```

Zig Zag in binary tree

[Binary Tree Zigzag Level Order Traversal - LeetCode]()

- [ ] Done

Tc → O(N) Sc→ O(N)

```
vector<vector<int>> zigzag(Node* root){
    vector<vector<int>> result;
     if (root==nullptr)
    {
        return result;
    }
    queue<Node*> que;
    que.push(root);
    bool left2Right = true;

    while (!que.empty())
    {
        int size = que.size();
        vector<int> row(size);

        for (int i = 0; i < size; i++)
        {
            Node* node = que.front();
            que.pop();
```

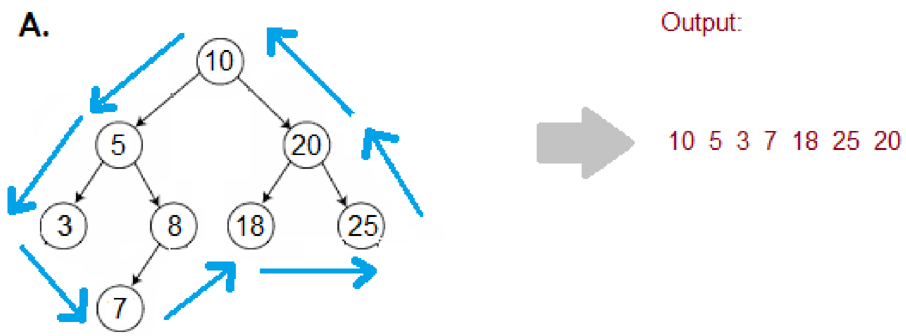
```
            int index = (left2Right) ? i : size-1-i;

            row[index] = node->data;

            if(node->left!=nullptr){
                que.push(node->left);
            }
            if(node->right!=nullptr){
                que.push(node->right);
            }
        }
        left2Right = !left2Right;
        result.push_back(row);

    }

    return result;

}
int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    root->right->right = new Node(7);

    vector<vector<int>> result = zigzag(root);

    // Output the zigzag level order traversal
    for (const auto& level : result) {
        for (int val : level) {
            cout << val << " ";
        }
```

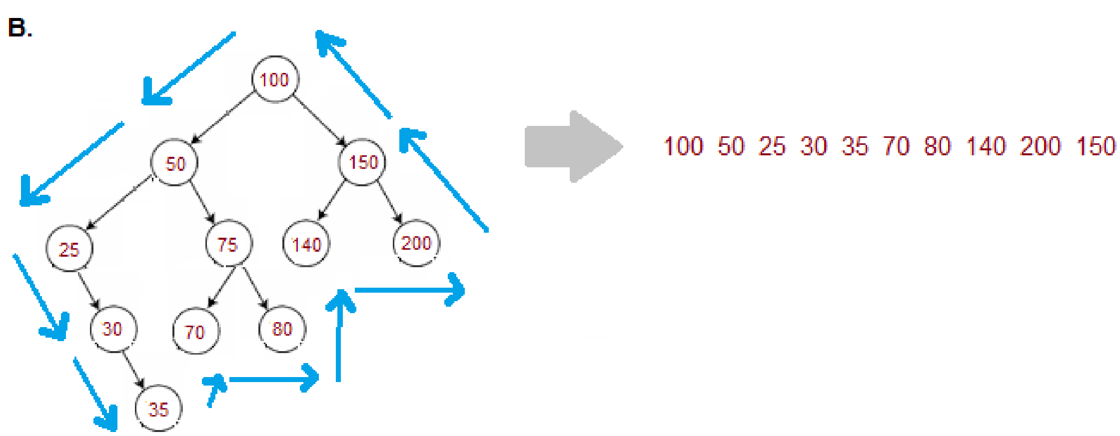
```
    }
    return 0;
}
```

Boundary Traversal in binary tree

[Boundary of Binary Tree - LeetCode]()

> It's easy just observe and revise again





- [ ] Done

Tc → O(N) Sc→ O(N)

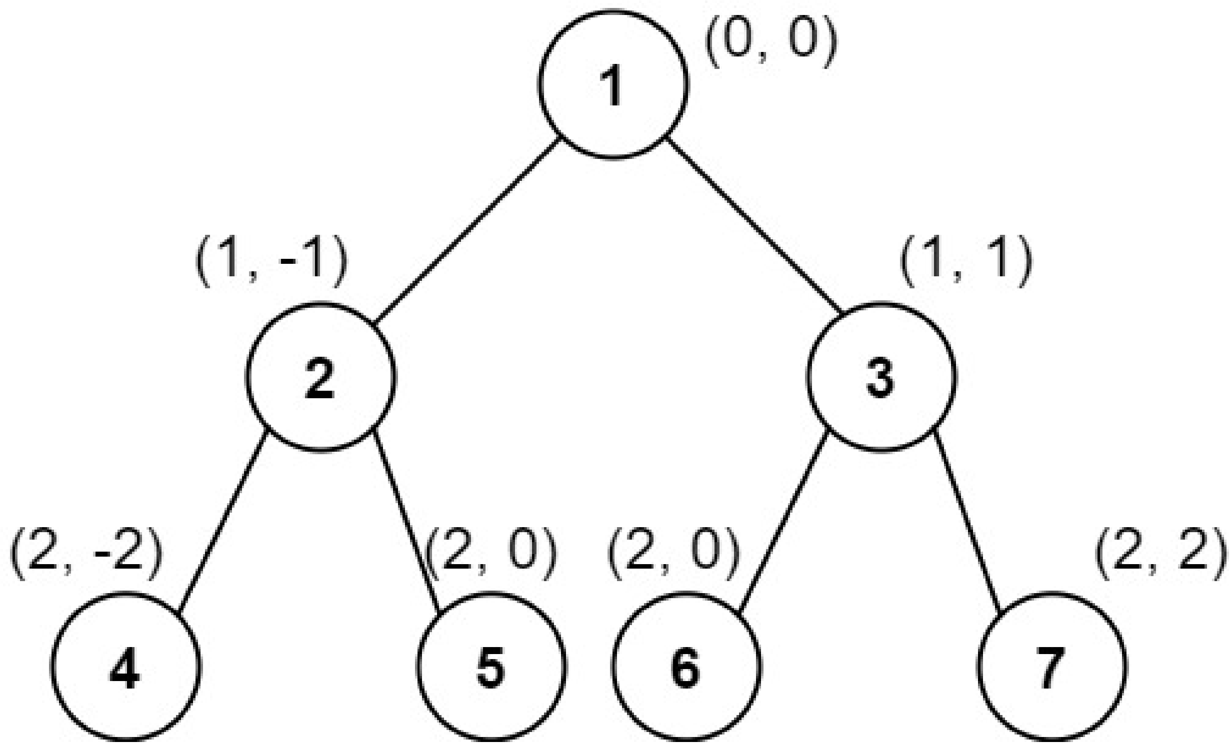
```
bool isLeaf(Node* root) {
        return root->left == nullptr && root->right == nullptr;
    }

    void addLeftNodes(Node* root,vector<int> &res){
        Node* cur = root->left;
        if(!isLeaf(cur)){
            res.push_back(cur->data);
        }
        if(cur->left){
            cur = cur->left;
        }else{
            cur = cur->right;
        }
    }
    void addRightNodes(Node* root,vector<int> &res){
        Node* cur = root->right;
        vector<int> temp;
        if(!isLeaf(cur)){
            temp.push_back(cur->data);
        }
        reverse(temp.begin(),temp.end());
        if(cur->right){
            cur = cur->right;
        }else{
            cur = cur->left;
        }

        for (int i = 0; i < temp.size(); i++)
        {
            res.push_back(temp[i]);
        }

    }
    void addLeaves(Node* root,vector<int> &res){
```

```
        if(isLeaf(root)){
            res.push_back(root->data);
            return;
        }
        addLeaves(root->left,res);
        addLeaves(root->right,res);

    }
    vector<int> printBoundary(Node* root){
        vector<int> res;
        if(!root){
            return res;
        }
        if(!isLeaf(root)) {
            res.push_back(root->data);
        }

        addLeftNodes(root,res);
        addLeaves(root,res);
        addRightNodes(root,res);
        return res;
    }
```

Vertical Traversal in binary tree

[Vertical Order Traversal of a Binary Tree - LeetCode]

> It's easy just observe and revise again



- [ ] Done

Tc → O(N) Sc→ O(N)

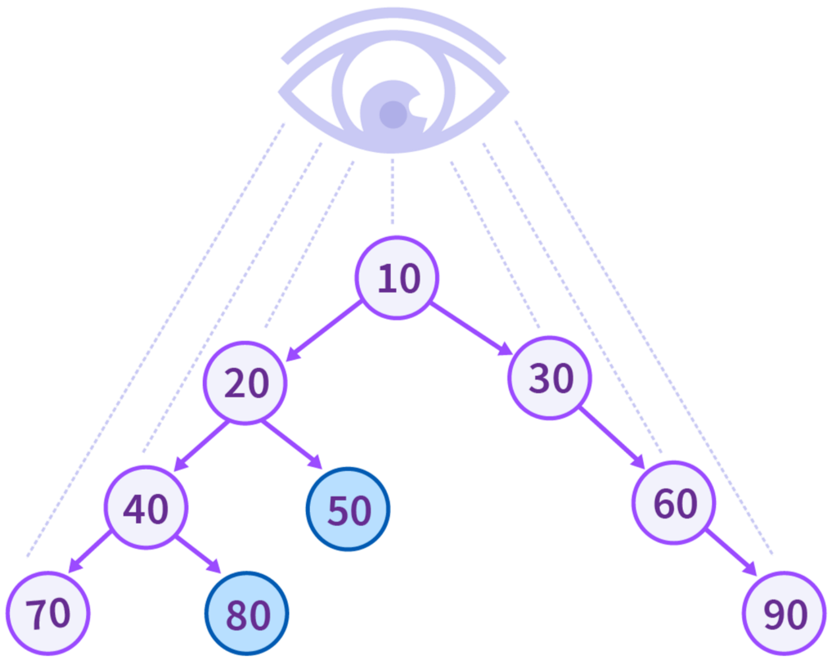
```
class Solution {
public:
    vector<vector<int>> verticalTraversal(TreeNode* root) {
        map<int,map<int,multiset<int>>> nodes;
        queue<pair<TreeNode*,pair<int,int>>> que;
        que.push({root,{0,0}});

        while(!que.empty()){
            auto p = que.front();
            que.pop();
            TreeNode* node = p.first;

            int x = p.second.first;  //vertical
            int y = p.second.second; // level
            nodes[x][y].insert(node->val);// ver,level then val
```

```
            if(node->left){ // traverse to left
                que.push({node->left,{x-1,y+1}});
            }
            if(node->right){ // traverse to left
                que.push({node->right,{x+1,y+1}});
            }
        }

        vector<vector<int>> res;
        for(auto p : nodes){
            vector<int> temp;
            for(auto q : p.second){
                temp.insert(temp.end(),q.second.begin(),q.secon
            }

            res.push_back(temp);
        }

        return res;
    }
};
```

Top View in binary tree

[Top View of Binary Tree | Practice | GeeksforGeeks]

> It's easy just observe and revise again



Top view- 70, 40, 10, 30, 60, 90

- [ ] Done

Tc → O(N) Sc→ O(N)

```
vector<int> topView(Node *root)
    {
        vector<int> ans;
        if(root==nullptr){
            return ans;
        }
        map<int,int> nodes;
        queue<pair<Node*,int>> que;
        que.push({root,0});
        while(!que.empty()){
            auto p = que.front();
            que.pop();
```

```
            Node* node = p.first;
            int x = p.second;
            if(nodes.find(x)==nodes.end()){
                nodes[x] = node->data;
            }
            if(node->left){
                que.push({node->left,x-1});
            }

            if(node->right){
                que.push({node->right,x+1});
            }
        }
        for(auto it : nodes){
            ans.push_back(it.second);
        }
        return ans;
    }
```

Bottom View in binary tree

Bottom View Of Binary Tree - Naukri Code 360

Bottom View of Binary Tree | Practice | GeeksforGeeks

> It's easy just observe and revise again

- [ ] Done

Tc → O(N) Sc→ O(N)

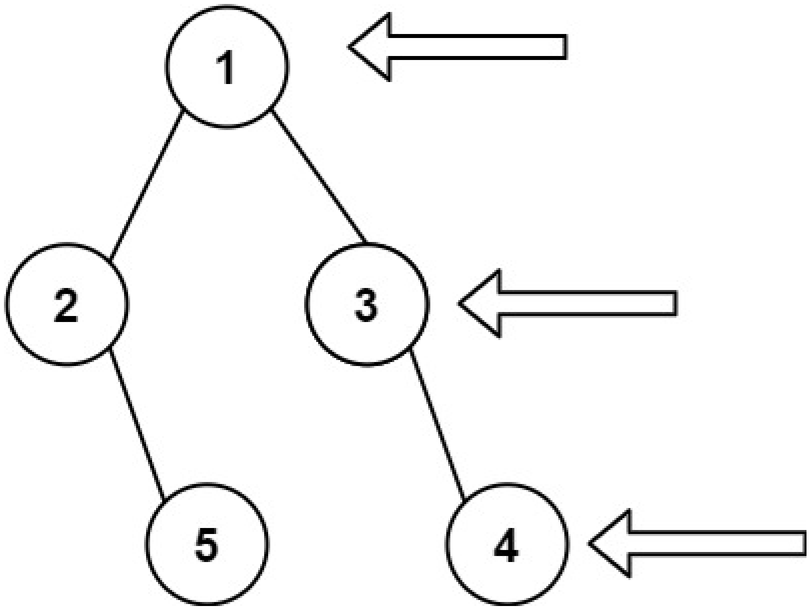
```

vector<int> bottomView(TreeNode<int> * root){
    vector<int> res;
    if(root==nullptr){
        return res;
```

```
    }
    map<int,int> m;
    queue<pair<TreeNode<int>*,int>> que;
    que.push({root,0});

    while(!que.empty()){
        auto p = que.front();
        que.pop();

        TreeNode<int>* node = p.first;
        int x = p.second;

        m[x] = node->data;

        if (node->left) {
            que.push({node->left, x - 1});
        }

        if (node->right) {
            que.push({node->right, x + 1});
        }
    }

    for(auto it :m){
            res.push_back(it.second);
        }
    return res;
}
```

Right View in binary tree

[Binary Tree Right Side View - LeetCode]

> It's easy just observe and revise again



- [ ] Done

Tc → O(N) Sc→ O(N)

```
void recursion(TreeNode* root,int level,vector<int> &res){
       if(root==nullptr){
           return;
       }
       if(level==res.size()){
           res.push_back(root->val);
       }
       recursion(root->right,level+1,res);
       recursion(root->left,level+1,res);
   }
   vector<int> rightSideView(TreeNode* root) {
       vector<int> res;
       recursion(root,0,res);
       return res;
   }
```

Check for Symmetric binary tree

Symmetric Tree - LeetCode

Let's consider the binary tree below for the dry run:

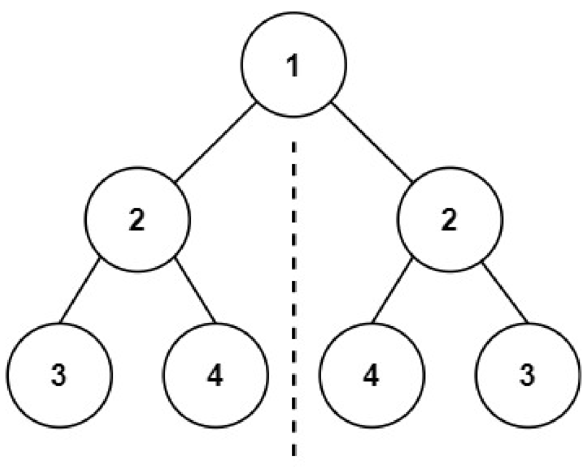
```
     1
    / \
   2   2
  /     \
 3       3
```

Step-by-Step Dry Run

1. **Initial Call:**
    - **Function**: `isSymmetric(root)`
    - **Input**: `root` points to node `1`.
    - **Action**: Calls `helper(root->left, root->right)` with `left` pointing to node `2` (left subtree) and `right` pointing to node `2` (right subtree).
2. **First Call to `helper(left, right)`:**
    - **Input**: `left` points to node `2` (left child of root), `right` points to node `2` (right child of root).
    - **Check**: Both nodes are not `nullptr`, and `left->val` (2) equals `right->val` (2).
    - **Action**: Calls `helper(left->left, right->right)` with `left->left` pointing to node `3` (left child of left subtree) and `right->right` pointing to node `3` (right child of right subtree).
3. **Second Call to `helper(left->left, right->right)`:**
    - **Input**: `left->left` points to node `3`, `right->right` points to node `3`.
    - **Check**: Both nodes are not `nullptr`, and `left->left->val` (3) equals `right->right->val` (3).
    - **Action**: Calls `helper(left->left->left, right->right->right)` (both `nullptr`) and `helper(left->left->right, right->right->left)` (both `nullptr`).

4. **Base Case Calls**:
    - **Input**: Both `left->left->left` and `right->right->right` are `nullptr`.
    - **Output**: Returns `true` since both are `nullptr`.
    - **Input**: Both `left->left->right` and `right->right->left` are `nullptr`.
    - **Output**: Returns `true` since both are `nullptr`.
5. **Back to First Call to `helper(left, right)`**:
    - **Action**: Now, it checks `helper(left->right, right->left)` with `left->right` as `nullptr` and `right->left` as `nullptr`.
6. **Base Case Call**:
    - **Input**: Both `left->right` and `right->left` are `nullptr`.
    - **Output**: Returns `true` since both are `nullptr`.
7. **Final Return**:
    - **Output**: All recursive calls returned `true`, so the tree is symmetric, and `isSymmetric` returns `true`

> It's easy just observe and revise again



- [ ] Done

Tc → O(N) Sc→ O(N)

```
bool helper(TreeNode* left,TreeNode* right){
    if(left==nullptr || right == nullptr){
        return left==right;
    }
    if(left->val!=right->val){
        return false;
    }
    return helper(left->left,right->right) && helper(left->r
}
bool isSymmetric(TreeNode* root) {
    if(root==nullptr) return false;
    return helper(root->left,root->right);
}
```

Print root to Node path in binary tree

Path In A Tree - Naukri Code 360

Let's consider the binary tree below for the dry run:

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

- **Tree Root**: Node `1`
- **Target** `x`: `5`

1. **Initial Call**:
    - Function: `pathInATree(root, 5)`
    - `ans` is initialized as an empty vector: `ans = []`

- Calls `helper(root, ans, 5)`

2. **First Call to `helper(root, ans, 5)`**:
   - **Input**: `root` points to node `1`, `ans = []`
   - Action: `root->data` (`1`) is not `5`, so push `1` to `ans`: `ans = [1]`
   - Calls `helper(root->left, ans, 5)` with `root->left` pointing to node `2`
3. **Second Call to `helper(root->left, ans, 5)`**:
   - **Input**: `root` points to node `2`, `ans = [1]`
   - Action: `root->data` (`2`) is not `5`, so push `2` to `ans`: `ans = [1, 2]`
   - Calls `helper(root->left, ans, 5)` with `root->left` pointing to node `4`
4. **Third Call to `helper(root->left->left, ans, 5)`**:
   - **Input**: `root` points to node `4`, `ans = [1, 2]`
   - Action: `root->data` (`4`) is not `5`, so push `4` to `ans`: `ans = [1, 2, 4]`
   - Calls `helper(root->left, ans, 5)` with `root->left` as `nullptr`
5. **Base Case Call to `helper(nullptr, ans, 5)`**:
   - **Input**: `root` is `nullptr`
   - Returns `false` because the node is `nullptr`.
6. **Back to Third Call**:
   - Calls `helper(root->right, ans, 5)` with `root->right` as `nullptr`
   - **Input**: `root` is `nullptr`
   - Returns `false`
   - Pops `4` from `ans`: `ans = [1, 2]`
7. **Back to Second Call**:
   - Calls `helper(root->right, ans, 5)` with `root->right` pointing to node `5`
8. **Fourth Call to `helper(root->right, ans, 5)`**:
   - **Input**: `root` points to node `5`, `ans = [1, 2]`
   - Action: `root->data` (`5`) equals `x` (`5`), so push `5` to `ans`: `ans = [1, 2, 5]`

- Returns `true`

9. **Final Return**:
   - All recursive calls return `true`, so the path is `[1, 2, 5]`
   - `pathInATree` returns `ans = [1, 2, 5]`

> It's easy just observe and revise again

- [ ] Done

Tc → O(N) Sc→ O(N)

```
bool helper(TreeNode<int>* root,vector<int> &arr,int x){
    if(root==nullptr){
        return false;
    }
    arr.push_back(root->data);
    if(root->data == x){
        return true;
    }

    if (helper(root->left, arr, x) || helper(root->right, arr, x
        return true;
    }
    arr.pop_back();
    return false;
}
vector<int> pathInATree(TreeNode<int> *root, int x)
{
    vector<int> ans;

    if(root==nullptr){
        return ans;
    }
    helper(root,ans,x);
```

```
    return ans;
}
```

Maximum Width in binary tree

[Maximum Width of Binary Tree - LeetCode]

Let's consider the binary tree below for the dry run:

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

1. **Initial Setup**:
    - Queue: `[(1, 0)]` (root node with index 0)
    - `ans = 0`
2. **Level 1**:
    - Process node `1` at index `0`.
    - Add children: `2` (index 1), `3` (index 2).
    - `first = 0`, `last = 0`, `ans = max(0, 0-0+1) = 1`.
    - Queue: `[(2, 1), (3, 2)]`
3. **Level 2**:
    - Process nodes `2` and `3`.
    - Add children: `4` (index 1), `5` (index 4).
    - `first = 0`, `last = 1`, `ans = max(1, 1-0+1) = 2`.
    - Queue: `[(4, 1), (5, 4)]`
4. **Level 3**:

- Process nodes `4` and `5`.
- No more children to add.
- `first = 0`, `last = 3`, `ans = max(2, 3-0+1) = 4`.

Final `ans = 4`

> It's tricky just observe properly and do dry run once

- [ ] Done

Tc → O(N) Sc→ O(N)

```
int widthOfBinaryTree(TreeNode* root) {
   if(root == nullptr) return 0;

   queue<pair<TreeNode*, unsigned long long int>> q;
   q.push({root, 0}); // root node
   int ans = 0;
   while(!q.empty()) {
       unsigned long long int mini = q.front().second; // index
       int size = q.size();
       unsigned long long int first, last;

       for(int i = 0; i < size; i++) {
           unsigned long long int cur = q.front().second - mini
           TreeNode* node = q.front().first;
           q.pop();
           if(i == 0) first = cur;
           if(i == size - 1) last = cur;
           if(node->left) {
               q.push({node->left, cur * 2 + 1}); // indexing
           }
           if(node->right) {
               q.push({node->right, cur * 2 + 2}); // indexing
           }
```

```
        }
        ans = max(ans, static_cast<int>(last - first + 1));
    }
    return ans;
}
```