Trees

06 October 2025 15:12

🧠 Recursive DFS Traversals

Assume we have this structure:

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

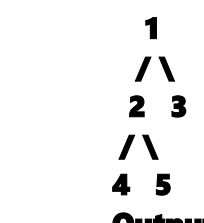
1️⃣ Preorder Traversal (Root → Left → Right)

👉 **Visit current node first, then go left, then right.**

```
void preorder(TreeNode* root) {
    if (root == NULL) return;
cout << root->val << " ";     // 1. Visit Root
    preorder(root->left);        // 2. Visit Left
    preorder(root->right);       // 3. Visit Right
}
```

🎯 **Example:**

Tree:



Output: 1 2 4 5 3

2️⃣ Inorder Traversal (Left → Root → Right)

👉 **Visit left first, then root, then right.**

```
void inorder(TreeNode* root) {
    if (root == NULL) return;
        inorder(root->left);         // 1. Visit Left
        cout << root->val << " ";     // 2. Visit Root
        inorder(root->right);        // 3. Visit Right
        }
```

🎯 **Example Output: 4 2 5 1 3**

3️⃣ Postorder Traversal (Left → Right → Root)

👉 **Visit both children before root.**

```
void postorder(TreeNode* root) {
    if (root == NULL) return;
```

```
    postorder(root->left);       // 1. Visit Left
    postorder(root->right);      // 2. Visit Right
    cout << root->val << " ";    // 3. Visit Root
}
```

🧩 Example Output: 4 5 2 3 1

🪜 4️⃣ Level Order (Breadth First Search)

Using queue — not recursion.

```
void levelOrder(TreeNode* root) {
    if (root == NULL) return;
queue<TreeNode*> q;
    q.push(root);
while (!q.empty()) {
        TreeNode* node = q.front();
        q.pop();
        cout << node->val << " ";
if (node->left) q.push(node->left);
        if (node->right) q.push(node->right);
    }
}
```

🧩 Output: 1 2 3 4 5

OR

```
queue<TreeNode*> q;
q.push(root);
while (!q.empty()) {
    int size = q.size();
    vector<int> level;
    for (int i = 0; i < size; i++) {
        TreeNode* node = q.front();
        q.pop();
        level.push_back(node->val);

        if (node->left) q.push(node->left);
        if (node->right) q.push(node->right);
    }
    ans.push_back(level);
}
```

🧩 Output:

```
[
  [1],
  [2, 3],
  [4, 5, 6]
]
```

☑️ 5. Iterative Preorder Traversal Using Stack

```
#include <iostream>
```

```
#include <stack>
using namespace std;
struct Node {
   int data;
   Node* left;
   Node* right;
Node(int val) {
       data = val;
       left = right = nullptr;
   }
};
void preorderIterative(Node* root) {
   if (root == nullptr) return;
   stack<Node*> st;
   st.push(root);
while (!st.empty()) {
       Node* curr = st.top();
       st.pop();
      cout << curr->data << " ";
// Push right first, then left (so left is processed first)
       if (curr->right)
          st.push(curr->right);
       if (curr->left)
          st.push(curr->left);
   }
}
int main() {
   // Example Tree
   //       1
   //      / \
   //     2   3
   //    / \   \
   //   4   5   6
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->right = new Node(6);
cout << "Preorder Traversal (Iterative): ";
   preorderIterative(root);
```

```
return 0;
}
```

🧠 **Output**

```
Preorder Traversal (Iterative): 1 2 4 5 3 6
```

☑6. Iterative Inorder Traversal Using Stack

```
#include <iostream>
#include <stack>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
Node(int val) {
        data = val;
        left = right = nullptr;
    }
};
void inorderIterative(Node* root) {
    stack<Node*> st;
    Node* curr = root;
while (curr != nullptr || !st.empty()) {
        // [1] Go to the leftmost node
        while (curr != nullptr) {
            st.push(curr);
            curr = curr->left;
        }
// [2] Process the top of the stack
        curr = st.top();
        st.pop();
        cout << curr->data << " ";
// [3] Move to the right subtree
        curr = curr->right;
    }
}
int main() {
    // Example Tree
    //       1
    //      / \
```

```
    //     2   3
    //    / \  \
    //   4   5   6
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->right = new Node(6);
cout << "Inorder Traversal (Iterative): ";
    inorderIterative(root);
return 0;
}
```

🧠 **Output**

```
Inorder Traversal (Iterative): 4 2 5 1 3 6
```

🌳 All 3 Iterative Traversals

```
#include <bits/stdc++.h>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = right = nullptr;
    }
};
// ---------------- Preorder Traversal (Iterative) ----------------
void preorder(Node* root) {
    if (root == nullptr) return;
    stack<Node*> st;
    st.push(root);
cout << "Preorder: ";
    while (!st.empty()) {
        Node* curr = st.top();
        st.pop();
        cout << curr->data << " ";
```

```
// Push right first so that left is processed first
        if (curr->right) st.push(curr->right);
        if (curr->left) st.push(curr->left);
    }
    cout << endl;
}
// ---------------- Inorder Traversal (Iterative) ----------------
void inorder(Node* root) {
    stack<Node*> st;
    Node* curr = root;
cout << "Inorder: ";
    while (curr != nullptr || !st.empty()) {
        // Reach leftmost node
        while (curr != nullptr) {
            st.push(curr);
            curr = curr->left;
        }
curr = st.top();
        st.pop();
        cout << curr->data << " ";
// Move to right subtree
        curr = curr->right;
    }
    cout << endl;
}
// ---------------- Postorder Traversal (Iterative using 2
stacks) ----------------
void postorder(Node* root) {
    if (root == nullptr) return;
stack<Node*> st1, st2;
    st1.push(root);
while (!st1.empty()) {
        Node* curr = st1.top();
        st1.pop();
        st2.push(curr);
// Push left and right children to st1
        if (curr->left) st1.push(curr->left);
        if (curr->right) st1.push(curr->right);
    }
cout << "Postorder: ";
    while (!st2.empty()) {
```

```
        cout << st2.top()->data << " ";
        st2.pop();
    }
    cout << endl;
}
// ---------------- Main ----------------
int main() {
    /*
           1
          / \
         2   3
        / \   \
       4   5   6
    */
    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->right = new Node(6);
preorder(root);
    inorder(root);
    postorder(root);
return 0;
}
```

🧠 **Output:**

```
Preorder: 1 2 4 5 3 6
Inorder: 4 2 5 1 3 6
Postorder: 4 5 2 6 3 1
```

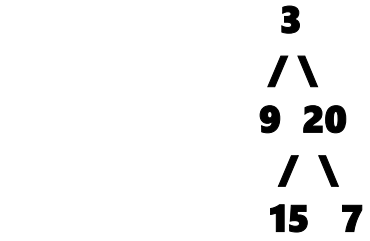
☑7. Maximum depth of binarry tree(DFS)

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == nullptr)
            return 0;

        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
```

```
            return 1 + max(leftDepth, rightDepth);
        }
    };
```

☑ Example:

For input:



Output:

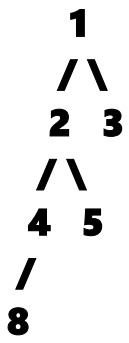
3

☑8. Balanced binarry tree(DFS)

```
class Solution {
public:
    int height(TreeNode* root) {
        if (root == nullptr)
            return 0;

        int left = height(root->left);
        if (left == -1) return -1; // left subtree not balanced

        int right = height(root->right);
        if (right == -1) return -1; // right subtree not balanced

        if (abs(left - right) > 1)
            return -1; // current node not balanced

        return 1 + max(left, right);
    }

    bool isBalanced(TreeNode* root) {
        return height(root) != -1;
    }
};
```

☑ Example:

Input:



Output:

False

☑9. Maximum Diameter of binarry tree(DFS)

```
class Solution {
public:
   int diameter = 0;

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

      int left = height(root->left);
      int right = height(root->right);

      diameter = max(diameter, left + right);  // update global
diameter

      return 1 + max(left, right);  // return height(to
determine left  &right)
   }

   int diameterOfBinaryTree(TreeNode* root) {
      height(root);
      return diameter;(excluding root node)
   }
};
```

☑ Example Trace:

For the tree:

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

- left height (2) = 2 (through 4-2-5)
- right height (3) = 1
- Diameter = 2 + 1 = 3

Output → 3

☑ 10. Maximum Path Sum of binarry tree(DFS)

```
class Solution {
public:
    int maxSum = INT_MIN;

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

        // compute max gain from left & right (ignore
negatives)
        int left = max(0, dfs(root->left));
        int right = max(0, dfs(root->right));

        // update maxSum considering path through current
node
        maxSum = max(maxSum, root->val + left + right);

        // return max gain to parent
        return root->val + max(left, right); (to determine left &
right)
    }

    int maxPathSum(TreeNode* root) {
        dfs(root);
        return maxSum;
    }
};
```

📄 Example 2

```
      -10
       / \
      9   20
```

/ \
15 7

- Left gain = 9
- Right gain = 20 + max(15,7)
- Best path = 15 + 20 + 7 = 42

☑ Output → 42

10.1 check whether target sum exists in node to leaf path

```
class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if (!root) return false;
        if (!root->left && !root->right)
            return targetSum == root->val;
        return hasPathSum(root->left, targetSum - root->val) ||
               hasPathSum(root->right, targetSum - root->val);
    }
};
```

10.2 Return node leaf path of tagert sum

```
class Solution {
public:
    vector<vector<int>> res;

    void dfs(TreeNode* root, int targetSum, vector<int>&
path) {
        if (!root) return;

        path.push_back(root->val);

        // If it's a leaf node and sum matches
        if (!root->left && !root->right && targetSum == root->
val)
            res.push_back(path);

        // Pass the reduced sum in the recursive calls
        dfs(root->left, targetSum - root->val, path);
        dfs(root->right, targetSum - root->val, path);

        // Backtrack
```

```
        path.pop_back();
    }

    vector<vector<int>> pathSum(TreeNode* root, int
targetSum) {
        vector<int> path;
        dfs(root, targetSum, path);
        return res;
    }
};
```

☑ 11. Check two trees are same(DFS)

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        // Both empty → same
        if (!p && !q) return true;

        // One empty, one not → different
        if (!p || !q) return false;

        // Values differ → different
        if (p->val != q->val) return false;

        // Recursively check left and right subtrees
        return isSameTree(p->left, q->left) && isSameTree(p->
right, q->right);
    }
};
```

🌳 Example:

```
Tree 1:
    1
   / \
  2   1
Tree 2:
    1
   / \
```

1 2

❌ Output:

false

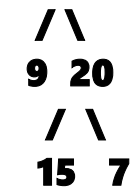
☑11. Zig Zag Level order Traversal(BFS)

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    vector<vector<int>> ans;
    if (!root) return ans;

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

    while (!q.empty()) {
        int size = q.size();
        vector<int> level(size);

        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front();
            q.pop();

            int index = leftToRight ? i : (size - 1 - i);
            level[index] = node->val;

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        ans.push_back(level);
        leftToRight = !leftToRight;
    }

    return ans;
}
```

✨ Example:

Input:

3



Output:

```
[
  [3],
  [20, 9],
  [15, 7]
]
```

Explanation:

- Level 1 → left to right → [3]
- Level 2 → right to left → [20, 9]
- Level 3 → left to right → [15, 7]

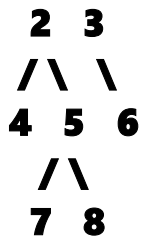
☑12. Boundary Traversal of Binary Tree

```
class Solution {
public:
    // Add left boundary (excluding leaves)
    void addLeftBoundary(TreeNode* root, vector<int>& res) {
        TreeNode* curr = root->left;
        while (curr) {
            if (curr->left || curr->right)
                res.push_back(curr->val);
            if (curr->left)
                curr = curr->left;
            else
                curr = curr->right;
        }
    }

    // Add leaf nodes
    void addLeaves(TreeNode* root, vector<int>& res) {
        if (!root) return;

        if (!root->left && !root->right) {
            res.push_back(root->val);
            return;
        }
```

```
        addLeaves(root->left, res);
        addLeaves(root->right, res);
    }

    // Add right boundary (excluding leaves, reversed)
    void addRightBoundary(TreeNode* root, vector<int>& res) {
        TreeNode* curr = root->right;
        vector<int> temp;

        while (curr) {
            if (curr->left || curr->right)
                temp.push_back(curr->val);
            if (curr->right)
                curr = curr->right;
            else
                curr = curr->left;
        }

        reverse(temp.begin(), temp.end());
        for (int val : temp) res.push_back(val);
    }

    // Main function
    vector<int> boundaryOfBinaryTree(TreeNode* root) {
        vector<int> res;
        if (!root) return res;

        res.push_back(root->val); // root always part of boundary

        addLeftBoundary(root, res);
        addLeaves(root, res);
        addRightBoundary(root, res);

        return res;
    }
};
```

☑ Example

```
        1
       / \
```



☑ **Output**

[1, 2, 4, 7, 8, 6, 3]

☑13. Vertical Traversal of Binary Tree

```
vector<vector<int>> verticalOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;

    map<int, vector<int>> mp; // col -> list of nodes
    queue<pair<TreeNode*, int>> q; // node + col
    q.push({root, 0});

    while (!q.empty()) {
        auto p = q.front(); q.pop();
        TreeNode* node = p.first;
        int col = p.second;

        mp[col].push_back(node->val);

        if (node->left) q.push({node->left, col - 1});
        if (node->right) q.push({node->right, col + 1});
    }

    for (auto& it : mp)
        result.push_back(it.second);

    return result;
}
```

OR

```
//for getting elements of a row in sorted manner
class Solution {
public:
    vector<vector<int>> verticalTraversal(TreeNode* root)
{
```

```
        // map: col → map<row, multiset<values>>
        map<int, map<int, multiset<int>>> nodes;
        queue<pair<TreeNode*, pair<int, int>>> q; // node,
{col, row}
        q.push({root, {0, 0}});
        while (!q.empty()) {
            auto p = q.front(); q.pop();
            TreeNode* node = p.first;
            int x = p.second.first;  // column
            int y = p.second.second; // row
            nodes[x][y].insert(node->val);
            if (node->left) q.push({node->left, {x - 1, y
+ 1}});
            if (node->right) q.push({node->right, {x + 1,
y + 1}});
        }
        vector<vector<int>> ans;
        for (auto& col : nodes) {
            vector<int> colNodes;
            for (auto& row : col.second)
                colNodes.insert(colNodes.end(),
row.second.begin(), row.second.end());
            ans.push_back(colNodes);
        }
        return ans;
    }
};
```

🌳 Example

Input:

```
    3
   / \
  9  20
    /  \
   15   7
```

Output:

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

Explanation:

- **Column -1 → [9]**
- **Column 0 → [3, 15]**
- **Column 1 → [20]**
- **Column 2 → [7]**

☑13. Top View of Binary Tree

```
class Solution {
public:
    vector<int> topView(TreeNode* root) {
        vector<int> result;
        if (!root) return result;

        map<int, int> topNode; // column -> node value
        queue<pair<TreeNode*, int>> q; // node + column
        q.push({root, 0});

        while (!q.empty()) {
            auto p = q.front(); q.pop();
            TreeNode* node = p.first;
            int col = p.second;

            // Only store the first node at each column
            if (topNode.find(col) == topNode.end()) {
                topNode[col] = node->val;
            }

            if (node->left) q.push({node->left, col - 1});
            if (node->right) q.push({node->right, col + 1});
        }

        for (auto it : topNode)
            result.push_back(it.second);

        return result;
    }
};
```

📓 Example

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

☑ Final map:

-1 → 2

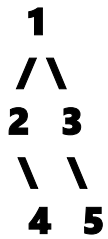
0 → 1
+1 → 3
+2 → 5

Answer: [2, 1, 3, 5]

☑13. Bottom View of Binary Tree

```
vector<int> bottomView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;

    map<int, int> bottomNode; // col -> node value
    queue<pair<TreeNode*, int>> q; // node + col
    q.push({root, 0});

    while (!q.empty()) {
        auto p = q.front(); q.pop();
        TreeNode* node = p.first;
        int col = p.second;

        // Here we overwrite the column every time (so last stays)
        bottomNode[col] = node->val;

        if (node->left) q.push({node->left, col - 1});
        if (node->right) q.push({node->right, col + 1});
    }

    for (auto it : bottomNode)
        result.push_back(it.second);

    return result;
}
```

📄 Example



☑ **Final map:**

col -1 → 2 (only one)

col 0 → 4   (1 is above it → 4 replaces it)
col +1 → 3
col +2 → 5

☑ Output: [2, 4, 3, 5]

☑14. Left View of Binary Tree

```
class Solution {
public:
   vector<int> leftSideView(TreeNode* root) {
      vector<int> ans;
      if (!root) return ans;

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

      while (!q.empty()) {
         int n = q.size();
         for (int i = 0; i < n; i++) {
            TreeNode* node = q.front(); q.pop();

            // first node of this level
            if (i == 0) ans.push_back(node->val);

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
         }
      }
      return ans;
   }
};
```

🧩 Example

```
        1
       / \
      2   3
     / \   \
    4   5   6
```

Left View: 1, 2, 4

☑15. Right View of Binary Tree

```
class Solution {
public:
   vector<int> rightSideView(TreeNode* root) {
      vector<int> ans;
      if (!root) return ans;

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

      while (!q.empty()) {
         int n = q.size();
         for (int i = 0; i < n; i++) {
            TreeNode* node = q.front(); q.pop();

            if (i == n - 1) ans.push_back(node->val); // last node of
level

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
         }
      }
      return ans;
   }
};
```

🧩 Example

```
        1
       / \
      2   3
     / \   \
    4   5   6
```

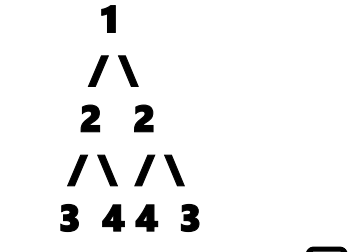
Right View: 1, 3, 6

☑16. Check Symmetry of Binary Tree

```
class Solution {
public:
   bool isSymmetric(TreeNode* root) {
      if (!root) return true;
      return isMirror(root->left, root->right);
```

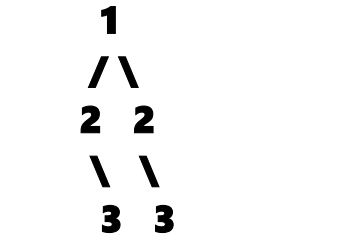
```
    }

    bool isMirror(TreeNode* left, TreeNode* right) {
        if (!left && !right) return true;
        if (!left || !right) return false;

        return (left->val == right->val) &&
               isMirror(left->left, right->right) &&
               isMirror(left->right, right->left);
    }
};
```

🧩 Example 1



Compare (2, 2) ☑
Compare (3, 3) ☑
Compare (4, 4) ☑
All pairs match → return true

🧩 Example 2



❌ Output: false

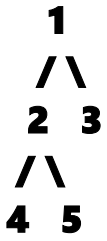
☑17. Root to node(target) path of Binary Tree

```
class Solution {
public:
    vector<int> rootToNodePath(TreeNode* root, int target) {
        vector<int> path;
        getPath(root, target, path);
        return path;
    }

private:
    bool getPath(TreeNode* root, int target, vector<int>& path) {
```

```
        if (!root) return false;

        path.push_back(root->val);

        if (root->val == target) return true;

        if (getPath(root->left, target, path) || getPath(root->right,
target, path))
            return true;

        path.pop_back(); // backtrack
        return false;
    }
};
```

Example:

For the tree:



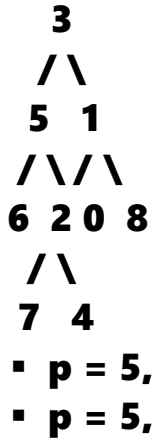
- **Target = 5**
- **Output: 1 2 5** ☑

☑18. Lowest common ancestor in Binary Tree

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root,
TreeNode* p, TreeNode* q) {
        if (!root) return nullptr;

        // If root is one of p or q, return root
        if (root == p || root == q) return root;

        // Search in left and right subtrees
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p,
q);
```

```
        // If p and q are found in both subtrees, root is LCA
        if (left && right) return root;

        // Otherwise, return non-null child
        return left ? left : right;
    }
};
```

Example tree:



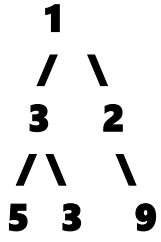
- p = 5, q = 1 → LCA = 3
- p = 5, q = 4 → LCA = 5

☑19. Max width of Binary Tree

```
class Solution {
public:
    int widthOfBinaryTree(TreeNode* root) {
        if (!root) return 0;

        queue<pair<TreeNode*, unsigned long long>> q;
        q.push({root, 0});
        unsigned long long maxWidth = 0;

        while (!q.empty()) {
            int size = q.size();
            unsigned long long left = q.front().second;   // first index
in level
            unsigned long long right = q.back().second;   // last index
in level
            maxWidth = max(maxWidth, right - left + 1);

            for (int i = 0; i < size; i++) {
```

```
            auto [node, idx] = q.front();
            q.pop();

            // Normalize index to prevent overflow
            idx = idx - left;

            if (node->left)
                q.push({node->left, 2 * idx + 1});
            if (node->right)
                q.push({node->right, 2 * idx + 2});
        }
    }

    return maxWidth;
  }
};
```

🌳 Example tree



🌼 Step-by-Step Dry Run

🟢 Initialization

`q = [(1, 0)]`
`maxWidth = 0`

🟩 Level 1:

`Queue → [(1, 0)]`

Step	Action	Details
1	size = 1	One node at this level
2	left = 0, right = 0	indices of first and last nodes
3	maxWidth = max(0, 0-0+1) = 1	width = 1
4	Pop (1, 0)	node=1, idx=0
5	Normalize idx → idx = 0 - 0 = 0	keep small numbers
6	Push left child (3, 2*0+1=1)	index = 1
7	Push right child (2, 2*0+2=2)	index = 2
☑	Queue after loop	[(3,1), (2,2)]

▨ **Level 2:**
Queue → [(3,1), (2,2)]

Step	Action	Details
1	size = 2	Two nodes at this level
2	left = 1, right = 2	
3	maxWidth = max(1, 2-1+1) = 2	width = 2
4	Pop (3,1)	node=3, idx=1
5	Normalize idx → idx = 1-1 = 0	
6	Push (5, *20+1=1), (3, 20+2=2)*	left and right children
7	Pop (2,2)	node=2, idx=2
8	Normalize idx → idx = 2-1=1	
9	Push (9, 2*1+2=4)	only right child
☑	Queue after loop	[(5,1), (3,2), (9,4)]

▨ **Level 3:**
Queue → [(5,1), (3,2), (9,4)]

Step	Action	Details
1	size = 3	Three nodes
2	left = 1, right = 4	
3	maxWidth = max(2, 4-1+1) = 4 ☑	width = 4
4	Pop (5,1) → Normalize idx=0	No children
5	Pop (3,2) → Normalize idx=1	No children
6	Pop (9,4) → Normalize idx=3	No children
☑	Queue empty → exit loop	

▩ **Final:**

maxWidth = 4
☑ **Answer = 4**

"It first calculates the width using front and back indices of the current level,
then processes each node through left and right ptr, normalizing indices to start from 0 for the next level, and assigns new indices (2*i+1, 2* i+2) for children to maintain
proper horizontal spacing."

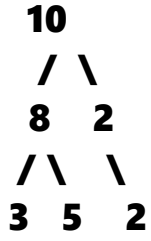
☑20. Children sum property Binary Tree

```
bool isSumTree(TreeNode* root) {
    // Base case: null node or leaf node
    if (!root || (!root->left && !root->right))
        return true;

    // Get values of left and right children
    int leftData = root->left ? root->left->val : 0;  //to check if one
of left/right          is zero
    int rightData = root->right ? root->right->val : 0;

    // Check current node and recurse for subtrees
    return (root->val == leftData + rightData)
           && isSumTree(root->left)
           && isSumTree(root->right);
}
```

OR

```
bool isSumTree(TreeNode* root) {
    if (!root || (!root->left && !root->right))
        return true; // null or leaf node

    int leftData = root->left ? root->left->val : 0;
    int rightData = root->right ? root->right->val : 0;

    if (root->val != leftData + rightData)
        return false;

    return isSumTree(root->left) && isSumTree(root->right);
}
```

✳ Example 1



- Check each node:
  - Node 10 → 8 + 2 = 10 ☑
  - Node 8 → 3 + 5 = 8 ☑
  - Node 2 → only right child 2 → 2 = 2 ☑

☑21. LCA of deepest leaves in Binary Tree

```
class Solution {
public:
    pair<TreeNode*, int> dfs(TreeNode* root) {
        if (!root) return {nullptr, 0};
        auto left = dfs(root->left);
        auto right = dfs(root->right);
        if (left.second == right.second)
            return {root, left.second + 1};
        else if (left.second > right.second)
            return {left.first, left.second + 1};
        else
            return {right.first, right.second + 1};
    }
    TreeNode* lcaDeepestLeaves(TreeNode* root) {
        return dfs(root).first;
    }
};
```

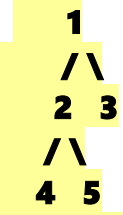
OR

```
class Solution {
public:
    TreeNode* lcaDeepestLeaves(TreeNode* root) {
        return dfs(root).second;
    }
private:
    // Returns {depth, lca} for each subtree
    pair<int, TreeNode*> dfs(TreeNode* root) {
        if (!root) return {0, nullptr};
        auto [ld, lnode] = dfs(root->left);
        auto [rd, rnode] = dfs(root->right);
        if (ld == rd)
            return {ld + 1, root};
```

```
        else if (ld > rd)
            return {ld + 1, lnode};
        else
            return {rd + 1, rnode};
    }
};
```

*****returning parents only if deepest leaves are at same level or returns the deepest leaf itself******

✳ Example A — Tree



Goal: show full recursion.

Call stack trace (depth-first, left-first)

a. dfs(1) — entering node 1
   → calls dfs(2)
b. dfs(2) — entering node 2
   → calls dfs(4)
c. dfs(4) — entering node 4
   - root != nullptr
   - calls dfs(4->left) → dfs(nullptr)
     - returns {nullptr, 0} (base case)
   - calls dfs(4->right) → dfs(nullptr)
     - returns {nullptr, 0}
   - Now inside dfs(4):
     - left = {nullptr, 0}
     - right = {nullptr, 0}
     - Compare depths: left.second == right.second (0 == 0) → true
     - Return {root, left.second + 1} → {4, 1}
       (Interpretation: deepest leaf in subtree rooted at 4 is 4 itself; height = 1)
       → dfs(4) returns {4, 1} to caller dfs(2).
d. Back in dfs(2) — after left:
   - left = {4, 1}
   - Now call dfs(5)
e. dfs(5) — symmetric to 4:
   - both children nullptr → both return {nullptr,0}
   - equal depths → return {5, 1}

→ dfs(5) returns {5, 1} to dfs(2).

f. Now inside dfs(2):
   - left = {4, 1}
   - right = {5, 1}
   - Compare depths: left.second == right.second (1 == 1) → true
   - Return {root, left.second + 1} → {2, 2}
     (Interpretation: deepest leaves in subtree of 2 are at depth 2 from node 2; node 2 is their LCA)
     → dfs(2) returns {2, 2} to dfs(1).

g. Back in dfs(1) — after left:
   - left = {2, 2}
   - Now call dfs(3)

h. dfs(3):
   - children are nullptr → both {nullptr,0}
   - equal → return {3, 1}
     → dfs(3) returns {3, 1} to dfs(1).

i. Now inside dfs(1):
   - left = {2, 2}
   - right = {3, 1}
   - Compare depths: left.second == right.second? (2 == 1) → false
   - left.second > right.second? (2 > 1) → true
   - Return {left.first, left.second + 1} → {2, 3}
     → dfs(1) returns {2, 3}. The .first is 2 → final answer LCA is node 2.

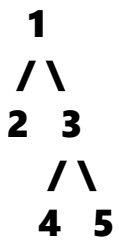
☑22. Serialize and deserialize a Binary Tree

```
class Codec {
public:
    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        if (!root) return "null";
        return to_string(root->val) + "," + serialize(root->left) + ","
+ serialize(root->right);
    }
    // Helper to rebuild tree from queue
    TreeNode* deserializeHelper(queue<string>& q) {
        string val = q.front(); q.pop();
        if (val == "null" || val.empty()) return nullptr;
        TreeNode* node = new TreeNode(stoi(val));
        node->left = deserializeHelper(q);
        node->right = deserializeHelper(q);
        return node;
    }
```

```
    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        stringstream s(data);
        string str;
        queue<string> q;
        while (getline(s, str, ',')) {
            if (!str.empty()) q.push(str); // ☑ skip empty tokens
        }
        return deserializeHelper(q);
    }
};
```

🧩 Example

Input Tree:



Serialized Output:

"1,2,null,null,3,4,null,null,5,null,null,"

Deserialized Tree:

☑ Reconstructs the *exact same structure* as input.

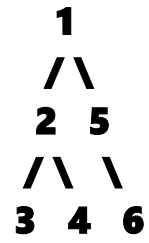
☑22. Serialize and deserialize a Binary Tree

```
class Solution {
public:
    void flatten(TreeNode* root) {
        if (!root) return;

        stack<TreeNode*> st;
        st.push(root);

        while (!st.empty()) {
            TreeNode* curr = st.top();
            st.pop();

            // Push right first, then left (so left is processed first)
```

```
        if (curr->right) st.push(curr->right);
        if (curr->left) st.push(curr->left);

        if (!st.empty())
            curr->right = st.top();  // Next node in preorder
        curr->left = nullptr;
      }
    }
};
```

Example

Input:



Flattened tree: