

# Complete Tree Pattern - Deep Study Guide

## Table of Contents

- 1. Theoretical Foundation
- 2. Mathematical Analysis
- 3. Tree Types - Deep Dive
- 4. Core Traversal Patterns
- 5. Advanced Pattern Recognition
- 6. Problem Categories with Analysis
- 7. 60+ Curated Problems
- 8. Optimization Techniques
- 9. Study Notes & Mental Models
- 10. Practice Roadmap
- 11. Common Pitfalls & Solutions

## Theoretical Foundation

What is a Tree?

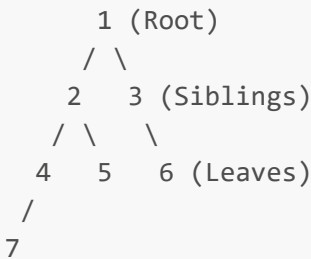
**Definition:** A tree is a hierarchical data structure consisting of nodes connected by edges, with one node designated as the root and all other nodes organized in a parent-child relationship.

**Historical Context:**

- Rooted in graph theory (Euler, 1736)
- Binary search trees formalized by P.F. Windley (1960)
- AVL trees by Adelson-Velsky and Landis (1962)
- Red-Black trees by Rudolf Bayer (1972)
- B-trees by Rudolf Bayer and Edward M. McCreight (1971)

**Core Principle:** Trees provide  $O(\log n)$  operations for search, insert, and delete in balanced scenarios, making them fundamental for efficient data organization and retrieval.

## Tree Terminology



**Key Terms:**

- **Root:** Top node with no parent
- **Parent:** Node with children
- **Child:** Node with a parent
- **Leaf:** Node with no children
- **Internal Node:** Node with at least one child
- **Siblings:** Nodes sharing the same parent
- **Ancestor:** Any node on path from root to current node
- **Descendant:** Any node reachable by going downward
- **Subtree:** Tree formed by a node and all its descendants
- **Depth:** Distance from root to node
- **Height:** Distance from node to deepest leaf
- **Level:** Set of nodes at same depth

## Tree Properties

### Height of Tree:

```
Height = max(depth of all leaves)
Empty tree: height = -1
Single node: height = 0
```

### Number of Nodes:

```
Minimum nodes at height h = h + 1
Maximum nodes at height h = 2^(h+1) - 1
```

### Relationship:

```
For n nodes:
Minimum height = ⌈log2(n+1)⌉ - 1
Maximum height = n - 1
```

## Mathematical Analysis

### Complexity Analysis

#### Perfect Binary Tree

```
Nodes at level i = 2^i
Total nodes = 2^(h+1) - 1
Leaves = 2^h
Internal nodes = 2^h - 1
Height = log2(n+1) - 1
```

## Complete Binary Tree

```
Height =  $\lfloor \log_2(n) \rfloor$   
Last level: nodes =  $n - (2^h - 1)$   
Array representation:  
- Left child of  $i = 2i + 1$   
- Right child of  $i = 2i + 2$   
- Parent of  $i = \lfloor (i-1)/2 \rfloor$ 
```

## Binary Search Tree (Balanced)

```
Search:  $O(\log n)$   
Insert:  $O(\log n)$   
Delete:  $O(\log n)$   
Space:  $O(n)$ 
```

## Binary Search Tree (Skewed)

```
Search:  $O(n)$   
Insert:  $O(n)$   
Delete:  $O(n)$   
Degenerates to linked list
```

## Recursion Analysis

### Time Complexity of Tree Traversal:

```
 $T(n) = 2T(n/2) + O(1)$  [visiting each node once]  
Using Master Theorem:  
 $T(n) = O(n)$ 
```

### Space Complexity:

```
Recursion stack depth:  
Best case (balanced):  $O(\log n)$   
Worst case (skewed):  $O(n)$ 
```

---

## Tree Types - Deep Dive

## Type 1: Binary Tree

**Definition:** Each node has at most 2 children (left and right).

**Node Structure:**

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

**Variants:**

### A. Full Binary Tree

- Every node has 0 or 2 children
- No node has exactly 1 child
- Nodes =  $2 * \text{leaves} - 1$

### B. Complete Binary Tree

- All levels filled except possibly last
- Last level filled from left to right
- Used in heaps
- Height =  $\lfloor \log_2(n) \rfloor$

### C. Perfect Binary Tree

- All internal nodes have 2 children
- All leaves at same level
- Nodes =  $2^{(h+1)} - 1$

### D. Balanced Binary Tree

- Height difference of left and right subtrees  $\leq 1$
- For all nodes recursively
- Ensures  $O(\log n)$  operations

### E. Degenerate/Skewed Tree

- Each parent has only one child
- Essentially a linked list
- Height =  $n - 1$

## Type 2: Binary Search Tree (BST)

**Definition:** Binary tree with ordering property:

For each node:

- All values in left subtree < node.val
- All values in right subtree > node.val

### Properties:

1. Inorder traversal gives sorted sequence
2. Search operation follows binary search
3. No duplicate values (typically)

### Operations Complexity:

Operation	Balanced	Unbalanced
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Min/Max	$O(\log n)$	$O(n)$

### Key Operations:

#### Search:

```
TreeNode* search(TreeNode* root, int val) {
    if (!root || root->val == val) return root;
    if (val < root->val) return search(root->left, val);
    return search(root->right, val);
}
```

#### Insert:

```
TreeNode* insert(TreeNode* root, int val) {
    if (!root) return new TreeNode(val);
    if (val < root->val) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}
```

#### Delete (3 cases):

1. **Leaf node:** Simply remove
2. **One child:** Replace with child
3. **Two children:** Replace with inorder successor/predecessor

### Type 3: AVL Tree

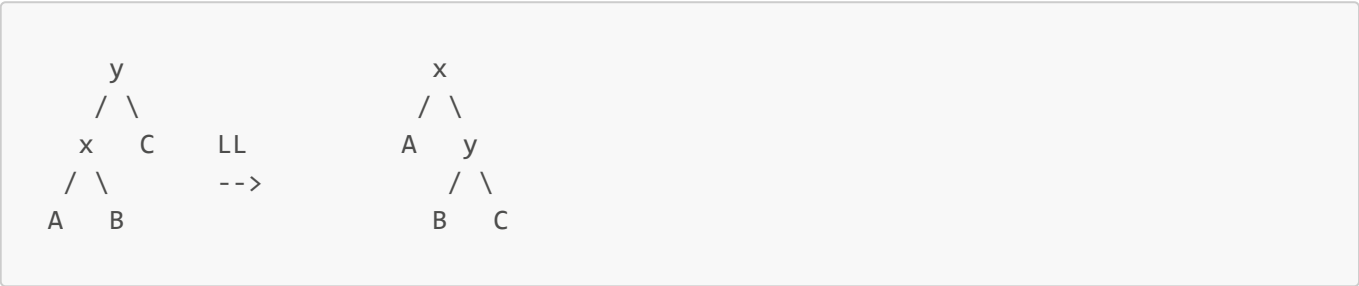
**Definition:** Self-balancing BST where height difference of left and right subtrees  $\leq 1$ .

**Balance Factor:**

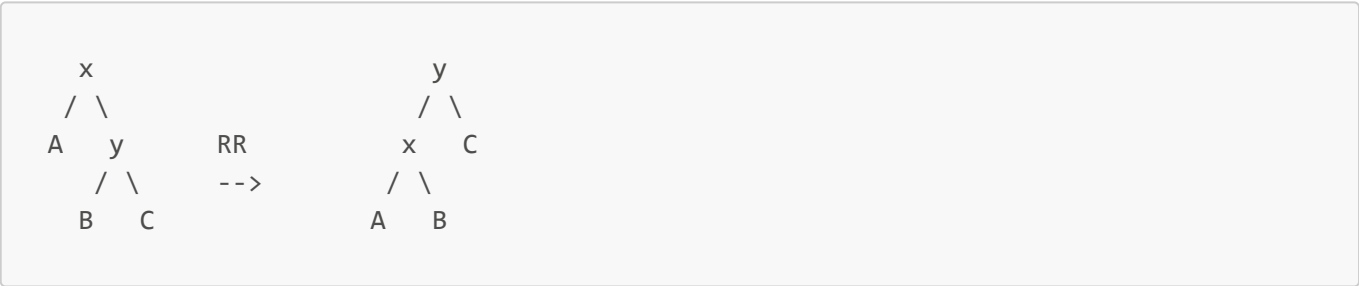
```
BF(node) = height(left) - height(right)
Valid: BF ∈ {-1, 0, 1}
```

**Rotations:**

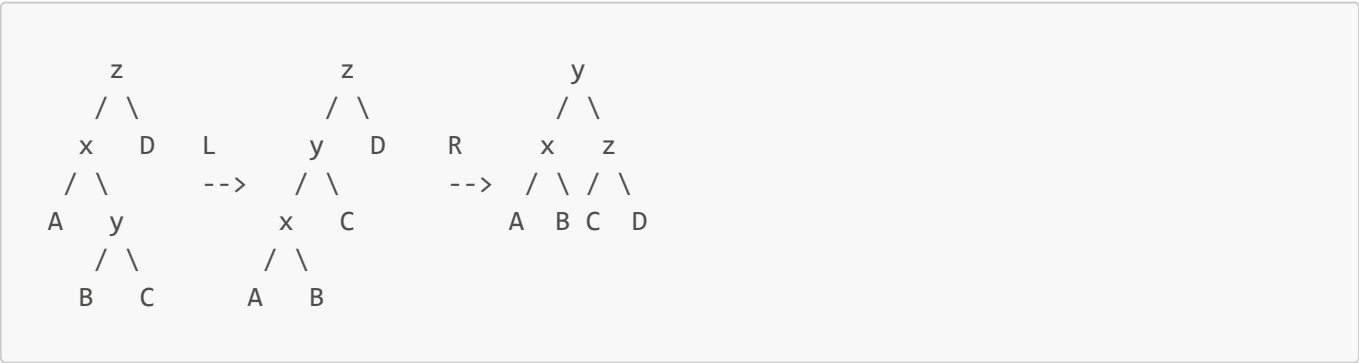
**1. Left Rotation (LL):**



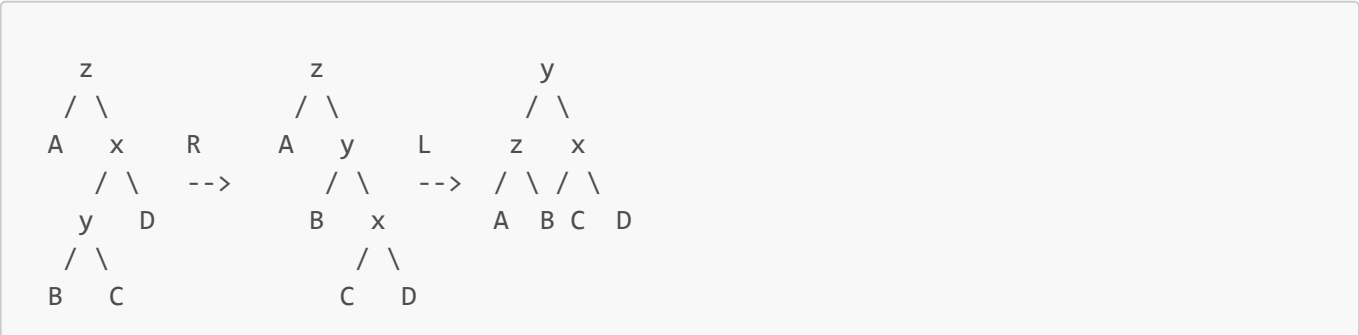
**2. Right Rotation (RR):**



**3. Left-Right (LR):**



**4. Right-Left (RL):**



**Time Complexity:**

```
Search:  $O(\log n)$   
Insert:  $O(\log n)$   
Delete:  $O(\log n)$   
Rotation:  $O(1)$ 
```

**Type 4: Red-Black Tree****Properties:**

1. Every node is RED or BLACK
2. Root is always BLACK
3. All leaves (NIL) are BLACK
4. RED node's children must be BLACK
5. All paths from node to leaves have same number of BLACK nodes

**Advantages over AVL:**

- Less strict balancing (faster insertion/deletion)
- Maximum height =  $2 * \log_2(n+1)$
- Used in: C++ STL map/set, Java TreeMap/TreeSet

**Rotation similar to AVL but with color changes****Type 5: Heap (Binary Heap)**

**Definition:** Complete binary tree satisfying heap property.

**Types:****A. Max Heap:**

```
Parent  $\geq$  Children  
Root = Maximum element
```

**B. Min Heap:**

```
Parent  $\leq$  Children  
Root = Minimum element
```

**Array Representation:**

For index  $i$ :  
 Parent:  $\lfloor (i-1)/2 \rfloor$   
 Left child:  $2i + 1$   
 Right child:  $2i + 2$

### Operations:

Operation	Time	Description
insert	$O(\log n)$	Add and bubble up
extractMax/Min	$O(\log n)$	Remove root and heapify
getMax/Min	$O(1)$	Return root
heapify	$O(\log n)$	Maintain heap property
buildHeap	$O(n)$	Create heap from array

### Heapify Down (for Max Heap):

```
void heapifyDown(vector& heap, int i, int size) {
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;

    if (left < size && heap[left] > heap[largest])
        largest = left;
    if (right < size && heap[right] > heap[largest])
        largest = right;

    if (largest != i) {
        swap(heap[i], heap[largest]);
        heapifyDown(heap, largest, size);
    }
}
```

### Type 6: Trie (Prefix Tree)

**Definition:** Tree for storing strings where each path represents a prefix.

#### Node Structure:

```
struct TrieNode {
    unordered_map children;
    bool isEndOfWord;
    TrieNode() : isEndOfWord(false) {}
};
```

#### Properties:

- Each edge labeled with a character
- Root represents empty string
- Path from root to node = prefix
- Marked nodes = complete words

**Operations:**

Operation	Time	Space
Insert	$O(m)$	$O(m)$ [m = word length]
Search	$O(m)$	$O(1)$
StartsWith	$O(m)$	$O(1)$
Delete	$O(m)$	$O(1)$

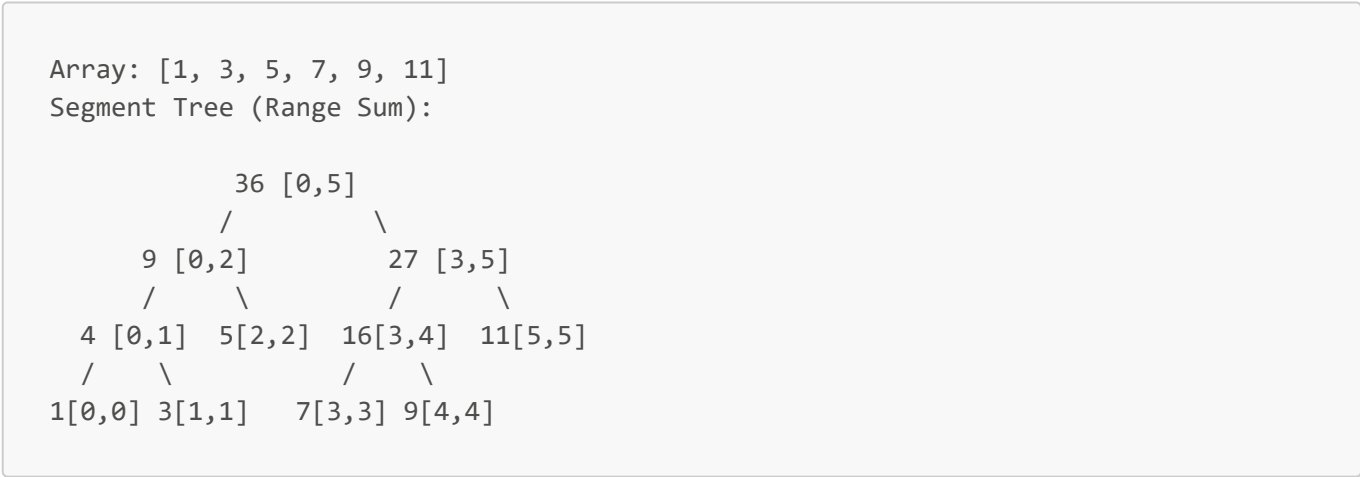
**Use Cases:**

- Autocomplete
- Spell checker
- IP routing
- Dictionary implementation

Type 7: Segment Tree

**Definition:** Binary tree for storing intervals/segments, allows efficient range queries.

**Structure:**



**Operations:**

Build	$O(n)$
Query	$O(\log n)$
Update	$O(\log n)$
Space	$O(4n) = O(n)$

**Applications:**

- Range sum queries
- Range minimum/maximum queries
- Range GCD/LCM queries

## Type 8: Fenwick Tree (Binary Indexed Tree)

**Definition:** Array-based structure for efficient prefix sum queries and updates.

### Properties:

- Uses binary representation of indices
- Parent of  $i = i - (i \& -i)$
- More space-efficient than Segment Tree

### Operations:

Update	$O(\log n)$
Query	$O(\log n)$
Space	$O(n)$

---

## Core Traversal Patterns

### Pattern 1: Depth-First Search (DFS)

#### A. Inorder Traversal (Left → Root → Right)

##### Recursive:

```
void inorder(TreeNode* root) {  
    if (!root) return;  
    inorder(root->left);  
    process(root->val);  
    inorder(root->right);  
}
```

##### Iterative:

```
vector inorder(TreeNode* root) {  
    vector result;  
    stack st;  
    TreeNode* curr = root;  
  
    while (curr || !st.empty()) {  
        // Go to leftmost node  
        while (curr) {  
            st.push(curr);  
            curr = curr->left;  
        }
```

```

    }
    // Process node
    curr = st.top(); st.pop();
    result.push_back(curr->val);
    // Move to right
    curr = curr->right;
}
return result;
}

```

### Use Cases:

- BST: gives sorted order
- Expression trees: infix notation

## B. Preorder Traversal (Root → Left → Right)

### Recursive:

```

void preorder(TreeNode* root) {
    if (!root) return;
    process(root->val);
    preorder(root->left);
    preorder(root->right);
}

```

### Iterative:

```

vector preorder(TreeNode* root) {
    vector result;
    if (!root) return result;
    stack st;
    st.push(root);

    while (!st.empty()) {
        TreeNode* node = st.top(); st.pop();
        result.push_back(node->val);
        // Push right first (so left is processed first)
        if (node->right) st.push(node->right);
        if (node->left) st.push(node->left);
    }
    return result;
}

```

### Use Cases:

- Tree copying
- Expression trees: prefix notation

- Tree serialization

### C. Postorder Traversal (Left → Right → Root)

#### Recursive:

```
void postorder(TreeNode* root) {  
    if (!root) return;  
    postorder(root->left);  
    postorder(root->right);  
    process(root->val);  
}
```

#### Iterative (2 Stack Method):

```
vector postorder(TreeNode* root) {  
    vector result;  
    if (!root) return result;  
    stack st1, st2;  
    st1.push(root);  
  
    while (!st1.empty()) {  
        TreeNode* node = st1.top(); st1.pop();  
        st2.push(node);  
        if (node->left) st1.push(node->left);  
        if (node->right) st1.push(node->right);  
    }  
  
    while (!st2.empty()) {  
        result.push_back(st2.top()->val);  
        st2.pop();  
    }  
    return result;  
}
```

#### Use Cases:

- Delete tree (free memory)
- Expression trees: postfix notation
- Dependency resolution

### Pattern 2: Breadth-First Search (Level Order)

#### Basic Level Order:

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

```

queue q;
q.push(root);

while (!q.empty()) {
    int levelSize = q.size();
    vector currentLevel;

    for (int i = 0; i < levelSize; i++) {
        TreeNode* node = q.front(); q.pop();
        currentLevel.push_back(node->val);

        if (node->left) q.push(node->left);
        if (node->right) q.push(node->right);
    }
    result.push_back(currentLevel);
}
return result;
}

```

### Variations:

#### A. Zigzag Level Order:

```

// Alternate direction each level
bool leftToRight = true;
if (!leftToRight)
    reverse(currentLevel.begin(), currentLevel.end());
leftToRight = !leftToRight;

```

#### B. Right Side View:

```

// Take last element of each level
result.push_back(currentLevel.back());

```

#### C. Vertical Order:

```

// Use map>
// Track column number for each node

```

**Time Complexity:**  $O(n)$  - visit each node once **Space Complexity:**  $O(w)$  -  $w$  is maximum width

## Advanced Pattern Recognition

### Decision Framework

START: Analyze Problem

↓

What type of tree?

→ Binary Tree (general)

↓

What's the goal?

→ Traverse all nodes?

↳ Choose DFS or BFS

→ Find path/validate property?

↳ DFS with recursion

→ Level-based operations?

↳ BFS (queue)

→ Construct/modify tree?

↳ Recursive construction

→ Binary Search Tree

↓

→ Search/Insert/Delete?

↳ BST property exploitation

→ Find k-th element?

↳ Inorder traversal

→ Validate BST?

↳ Inorder (check sorted) or Range checking

→ Need ordering/priority?

↳ Heap (Priority Queue)

→ String prefix operations?

↳ Trie

→ Range queries?

↳ Segment Tree / Fenwick Tree

## Pattern Recognition Keywords

### DFS Indicators:

- "path from root to leaf"
- "sum of paths"
- "validate property"
- "serialize/deserialize"
- "construct tree"
- "depth" related

### BFS Indicators:

- "level order"
- "minimum depth"
- "right/left side view"
- "level by level"
- "shortest path" (in unweighted tree)
- "width" related

**BST Indicators:**

- "sorted"
- "k-th smallest/largest"
- "range sum"
- "validate BST"
- "inorder successor/predecessor"

**Recursion Indicators:**

- "tree property" (height, diameter, balanced)
- "subtree"
- "same tree"
- "symmetric tree"

---

## Problem Categories with Analysis

### Category 1: Tree Traversal Fundamentals

#### 1.1 Inorder Traversal (LC 94)

**Difficulty:** Easy

**Pattern:** DFS Inorder

**Key Concepts:**

- Recursive vs Iterative
- Stack simulation of recursion
- Morris Traversal ( $O(1)$  space)

**Morris Traversal (Advanced):**

```
vector inorderMorris(TreeNode* root) {
    vector result;
    TreeNode* curr = root;

    while (curr) {
        if (!curr->left) {
            result.push_back(curr->val);
            curr = curr->right;
        } else {
            // Find predecessor
```

```

        TreeNode* pred = curr->left;
        while (pred->right && pred->right != curr)
            pred = pred->right;

        if (!pred->right) {
            // Create thread
            pred->right = curr;
            curr = curr->left;
        } else {
            // Remove thread
            pred->right = nullptr;
            result.push_back(curr->val);
            curr = curr->right;
        }
    }
}
return result;
}

```

**Time:**  $O(n)$

**Space:**  $O(1)$  for Morris,  $O(h)$  for recursive/iterative

## 1.2 Level Order Traversal (LC 102)

**Difficulty:** Medium

**Pattern:** BFS

**Approach:**

```

vector<vector> levelOrder(TreeNode* root) {
    vector<vector> result;
    if (!root) return result;

    queue q;
    q.push(root);

    while (!q.empty()) {
        int size = q.size();
        vector 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);
        }
        result.push_back(level);
    }
}

```

```
    return result;
}
```

**Key Insight:** Track level size before processing

## Category 2: Tree Property Validation

### 2.1 Maximum Depth (LC 104)

**Difficulty:** Easy

**Pattern:** DFS Recursion

**Recursive:**

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

**Iterative (BFS):**

```
int maxDepth(TreeNode* root) {
    if (!root) return 0;
    queue q;
    q.push(root);
    int depth = 0;

    while (!q.empty()) {
        int size = q.size();
        depth++;
        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front(); q.pop();
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }
    return depth;
}
```

**Time:**  $O(n)$

**Space:**  $O(h)$  recursive,  $O(w)$  iterative

### 2.2 Balanced Binary Tree (LC 110)

**Difficulty:** Easy

**Pattern:** Bottom-up DFS

**Approach:**

```
int checkHeight(TreeNode* root) {
    if (!root) return 0;

    int leftHeight = checkHeight(root->left);
    if (leftHeight == -1) return -1;

    int rightHeight = checkHeight(root->right);
    if (rightHeight == -1) return -1;

    if (abs(leftHeight - rightHeight) > 1) return -1;

    return 1 + max(leftHeight, rightHeight);
}

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

**Key Insight:** Return -1 to signal imbalance early

**2.3 Diameter of Binary Tree (LC 543)**

**Difficulty:** Easy

**Pattern:** DFS with global variable

**Problem:** Longest path between any two nodes

**Approach:**

```
int diameter = 0;

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

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

    // Update diameter (path through current node)
    diameter = max(diameter, leftHeight + rightHeight);

    return 1 + max(leftHeight, rightHeight);
}

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

**Key Insight:** Diameter = left height + right height at each node

## Category 3: Path Sum Problems

### 3.1 Path Sum (LC 112)

**Difficulty:** Easy

**Pattern:** DFS Recursion

**Problem:** Root-to-leaf path with target sum

**Approach:**

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

    // Leaf node check
    if (!root->left && !root->right)
        return targetSum == root->val;

    int remaining = targetSum - root->val;
    return hasPathSum(root->left, remaining) ||
        hasPathSum(root->right, remaining);
}
```

### 3.2 Path Sum II (LC 113)

**Difficulty:** Medium

**Pattern:** DFS with backtracking

**Problem:** Return all root-to-leaf paths with target sum

**Approach:**

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

    path.push_back(root->val);

    if (!root->left && !root->right && target == root->val) {
        result.push_back(path);
    }

    dfs(root->left, target - root->val, path, result);
    dfs(root->right, target - root->val, path, result);

    path.pop_back(); // Backtrack
}
```

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

**Key Technique:** Backtracking to explore all paths

### 3.3 Path Sum III (LC 437)

**Difficulty:** Medium

**Pattern:** DFS with prefix sum

**Problem:** Count paths (not necessarily root-to-leaf) with target sum

#### Approach 1: Brute Force $O(n^2)$

```
int countPaths(TreeNode* root, long sum) {
    if (!root) return 0;
    return (root->val == sum) +
        countPaths(root->left, sum - root->val) +
        countPaths(root->right, sum - root->val);
}

int pathSum(TreeNode* root, int targetSum) {
    if (!root) return 0;
    return countPaths(root, targetSum) +
        pathSum(root->left, targetSum) +
        pathSum(root->right, targetSum);
}
```

#### Approach 2: Prefix Sum $O(n)$

```
int pathSum(TreeNode* root, int targetSum) {
    unordered_map prefixSum;
    prefixSum[0] = 1; // Important: path from root
    return dfs(root, 0, targetSum, prefixSum);
}

int dfs(TreeNode* root, long currSum, int target,
        unordered_map& prefixSum) {
    if (!root) return 0;

    currSum += root->val;
    int count = prefixSum[currSum - target];

    prefixSum[currSum]++;
    count += dfs(root->left, currSum, target, prefixSum);
}
```

```
count += dfs(root->right, currSum, target, prefixSum);
prefixSum[currSum]--;

return count;
}
```

**Key Insight:** Similar to subarray sum = k

Category 4: Lowest Common Ancestor (LCA)

#### 4.1 LCA of Binary Tree (LC 236)

**Difficulty:** Medium

**Pattern:** DFS Recursion

**Approach:**

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (!root || root == p || root == q) return root;

    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);

    if (left && right) return root; // Both found
    return left ? left : right;    // Return non-null
}
```

**Time:**  $O(n)$

**Space:**  $O(h)$

**Key Cases:**

1. Both in left subtree → return left result
2. Both in right subtree → return right result
3. Split between subtrees → current node is LCA

#### 4.2 LCA of BST (LC 235)

**Difficulty:** Easy

**Pattern:** BST property exploitation

**Approach:**

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

    // Both in left subtree
    if (p->val < root->val && q->val < root->val)
```

```

        return lowestCommonAncestor(root->left, p, q);

    // Both in right subtree
    if (p->val > root->val && q->val > root->val)
        return lowestCommonAncestor(root->right, p, q);

    // Split point or one is ancestor
    return root;
}

```

### Iterative:

```

TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    while (root) {
        if (p->val < root->val && q->val < root->val)
            root = root->left;
        else if (p->val > root->val && q->val > root->val)
            root = root->right;
        else
            return root;
    }
    return nullptr;
}

```

**Time:**  $O(h)$

**Space:**  $O(1)$  iterative

## Category 5: BST Operations

### 5.1 Validate BST (LC 98)

**Difficulty:** Medium

**Pattern:** DFS with range checking

#### Approach 1: Range Checking

```

bool isValidBST(TreeNode* root, long min, long max) {
    if (!root) return true;
    if (root->val <= min || root->val >= max) return false;
    return isValidBST(root->left, min, root->val) &&
           isValidBST(root->right, root->val, max);
}

bool isValidBST(TreeNode* root) {
    return isValidBST(root, LONG_MIN, LONG_MAX);
}

```

#### Approach 2: Inorder Traversal

```

bool isValidBST(TreeNode* root) {
    TreeNode* prev = nullptr;
    return inorder(root, prev);
}

bool inorder(TreeNode* root, TreeNode*& prev) {
    if (!root) return true;
    if (!inorder(root->left, prev)) return false;
    if (prev && prev->val >= root->val) return false;
    prev = root;
    return inorder(root->right, prev);
}

```

**Time:**  $O(n)$

**Space:**  $O(h)$

## 5.2 Kth Smallest in BST (LC 230)

**Difficulty:** Medium

**Pattern:** Inorder traversal

### Approach 1: Full Inorder

```

void inorder(TreeNode* root, vector& nums) {
    if (!root) return;
    inorder(root->left, nums);
    nums.push_back(root->val);
    inorder(root->right, nums);
}

int kthSmallest(TreeNode* root, int k) {
    vector nums;
    inorder(root, nums);
    return nums[k-1];
}

```

### Approach 2: Early Stopping

```

int kthSmallest(TreeNode* root, int k) {
    int count = 0;
    int result = -1;
    inorder(root, k, count, result);
    return result;
}

void inorder(TreeNode* root, int k, int& count, int& result) {
    if (!root || count >= k) return;

```

```

    inorder(root->left, k, count, result);
    count++;
    if (count == k) {
        result = root->val;
        return;
    }
    inorder(root->right, k, count, result);
}

```

**Time:**  $O(h + k)$

**Space:**  $O(h)$

Category 6: Tree Construction

### 6.1 Construct from Preorder & Inorder (LC 105)

**Difficulty:** Medium

**Pattern:** Recursive construction

**Key Insight:**

- Preorder: [Root | Left subtree | Right subtree]
- Inorder: [Left subtree | Root | Right subtree]

**Approach:**

```

TreeNode* buildTree(vector& preorder, vector& inorder) {
    unordered_map inMap;
    for (int i = 0; i < inorder.size(); i++)
        inMap[inorder[i]] = i;
    return build(preorder, 0, preorder.size()-1,
                inorder, 0, inorder.size()-1, inMap);
}

TreeNode* build(vector& preorder, int preStart, int preEnd,
                vector& inorder, int inStart, int inEnd,
                unordered_map& inMap) {
    if (preStart > preEnd || inStart > inEnd) return nullptr;

    TreeNode* root = new TreeNode(preorder[preStart]);
    int inRoot = inMap[root->val];
    int numsLeft = inRoot - inStart;

    root->left = build(preorder, preStart+1, preStart+numsLeft,
                    inorder, inStart, inRoot-1, inMap);
    root->right = build(preorder, preStart+numsLeft+1, preEnd,
                    inorder, inRoot+1, inEnd, inMap);

    return root;
}

```

**Time:**  $O(n)$

**Space:**  $O(n)$  for map +  $O(h)$  recursion

## 6.2 Serialize and Deserialize (LC 297)

**Difficulty:** Hard

**Pattern:** Preorder DFS

**Approach:**

```
class Codec {
public:
    string serialize(TreeNode* root) {
        if (!root) return "#";
        return to_string(root->val) + "," +
            serialize(root->left) + "," +
            serialize(root->right);
    }

    TreeNode* deserialize(string data) {
        queue nodes;
        stringstream ss(data);
        string item;
        while (getline(ss, item, ','))
            nodes.push(item);
        return buildTree(nodes);
    }

    TreeNode* buildTree(queue& nodes) {
        string val = nodes.front();
        nodes.pop();
        if (val == "#") return nullptr;

        TreeNode* root = new TreeNode(stoi(val));
        root->left = buildTree(nodes);
        root->right = buildTree(nodes);
        return root;
    }
};
```

## Category 7: View Problems

### 7.1 Right Side View (LC 199)

**Difficulty:** Medium

**Pattern:** BFS or DFS

**BFS Approach:**

```
vector rightSideView(TreeNode* root) {
    vector result;
    if (!root) return result;

    queue q;
    q.push(root);

    while (!q.empty()) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front(); q.pop();
            if (i == size - 1) // Last node in level
                result.push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }
    return result;
}
```

### DFS Approach:

```
void dfs(TreeNode* root, int level, vector& result) {
    if (!root) return;
    if (level == result.size())
        result.push_back(root->val);
    dfs(root->right, level+1, result); // Right first!
    dfs(root->left, level+1, result);
}

vector rightSideView(TreeNode* root) {
    vector result;
    dfs(root, 0, result);
    return result;
}
```

## 7.2 Vertical Order Traversal (LC 987)

**Difficulty:** Hard

**Pattern:** BFS with coordinates

**Approach:**

```
vector<vector> verticalTraversal(TreeNode* root) {
    map>> nodes; // col -> row -> values
    queue<tuple> q; // node, row, col
    q.push({root, 0, 0});
```

```

while (!q.empty()) {
    auto [node, row, col] = q.front();
    q.pop();
    nodes[col][row].insert(node->val);
    if (node->left) q.push({node->left, row+1, col-1});
    if (node->right) q.push({node->right, row+1, col+1});
}

vector<vector> result;
for (auto& [col, rows] : nodes) {
    vector column;
    for (auto& [row, vals] : rows)
        column.insert(column.end(), vals.begin(), vals.end());
    result.push_back(column);
}
return result;
}

```

## Category 8: Subtree Problems

### 8.1 Subtree of Another Tree (LC 572)

**Difficulty:** Easy

**Pattern:** DFS with helper function

**Approach:**

```

bool isSubtree(TreeNode* root, TreeNode* subRoot) {
    if (!root) return false;
    if (isSameTree(root, subRoot)) return true;
    return isSubtree(root->left, subRoot) ||
           isSubtree(root->right, subRoot);
}

bool isSameTree(TreeNode* p, TreeNode* q) {
    if (!p && !q) return true;
    if (!p || !q) return false;
    return p->val == q->val &&
           isSameTree(p->left, q->left) &&
           isSameTree(p->right, q->right);
}

```

**Time:**  $O(m * n)$  where  $m$  = nodes in root,  $n$  = nodes in subRoot

## 60+ Curated Problems

### Foundation Problems (Easy - 15)

#### 1. Maximum Depth of Binary Tree (LC 104) ★

2. **Invert Binary Tree** (LC 226) ★
3. **Same Tree** (LC 100) ★
4. **Symmetric Tree** (LC 101) ★
5. **Path Sum** (LC 112) ★
6. **Minimum Depth** (LC 111)
7. **Balanced Binary Tree** (LC 110)
8. **Diameter of Binary Tree** (LC 543)
9. **Merge Two Binary Trees** (LC 617)
10. **Binary Tree Paths** (LC 257)
11. **Sum of Left Leaves** (LC 404)
12. **Range Sum of BST** (LC 938)
13. **Search in BST** (LC 700)
14. **Insert into BST** (LC 701)
15. **Delete Node in BST** (LC 450)

#### Core Traversal (Easy-Medium - 10)

16. **Binary Tree Inorder Traversal** (LC 94) ★
17. **Binary Tree Preorder Traversal** (LC 144) ★
18. **Binary Tree Postorder Traversal** (LC 145) ★
19. **Binary Tree Level Order** (LC 102) ★
20. **Binary Tree Zigzag Level Order** (LC 103)
21. **Binary Tree Right Side View** (LC 199)
22. **Average of Levels** (LC 637)
23. **N-ary Tree Level Order** (LC 429)
24. **Find Bottom Left Tree Value** (LC 513)
25. **Cousins in Binary Tree** (LC 993)

#### BST Problems (Medium - 12)

26. **Validate Binary Search Tree** (LC 98) ★
27. **Kth Smallest Element in BST** (LC 230) ★
28. **Convert Sorted Array to BST** (LC 108) ★
29. **BST Iterator** (LC 173)
30. **Lowest Common Ancestor of BST** (LC 235)
31. **Inorder Successor in BST** (LC 285)
32. **Closest BST Value** (LC 270)
33. **BST to Greater Sum Tree** (LC 1038)
34. **Recover Binary Search Tree** (LC 99)
35. **Trim a BST** (LC 669)
36. **Two Sum IV - BST** (LC 653)
37. **Unique BSTs** (LC 96)

#### Advanced Tree Problems (Medium-Hard - 15)

38. **Lowest Common Ancestor** (LC 236) ★
39. **Path Sum II** (LC 113) ★
40. **Path Sum III** (LC 437) ★

- 41. **Binary Tree Maximum Path Sum** (LC 124) ★
- 42. **Serialize and Deserialize Binary Tree** (LC 297) ★
- 43. **Construct Binary Tree from Preorder and Inorder** (LC 105) ★
- 44. **Construct Binary Tree from Inorder and Postorder** (LC 106)
- 45. **Flatten Binary Tree to Linked List** (LC 114)
- 46. **Populating Next Right Pointers** (LC 116)
- 47. **Count Complete Tree Nodes** (LC 222)
- 48. **Sum Root to Leaf Numbers** (LC 129)
- 49. **Binary Tree Vertical Order Traversal** (LC 314)
- 50. **All Nodes Distance K** (LC 863)
- 51. **Distribute Coins in Binary Tree** (LC 979)
- 52. **House Robber III** (LC 337)

### Trie Problems (Medium - 6)

- 53. **Implement Trie** (LC 208) ★
- 54. **Add and Search Word** (LC 211)
- 55. **Word Search II** (LC 212)
- 56. **Design Search Autocomplete System** (LC 642)
- 57. **Replace Words** (LC 648)
- 58. **Longest Word in Dictionary** (LC 720)

### Advanced Data Structures (Hard - 7)

- 59. **Binary Tree Cameras** (LC 968)
- 60. **Vertical Order Traversal** (LC 987)
- 61. **Maximum Sum BST in Binary Tree** (LC 1373)
- 62. **Range Module** (LC 715) - Segment Tree
- 63. **Count of Smaller Numbers After Self** (LC 315)
- 64. **The Skyline Problem** (LC 218)
- 65. **Number of Ways to Reorder Array** (LC 1569)

---

## Optimization Techniques

### Technique 1: Morris Traversal

**Purpose:** O(1) space traversal using threading

**Concept:** Use rightmost node in left subtree to create temporary link back to current node

**Inorder Morris:**

```
void morrisInorder(TreeNode* root) {
    TreeNode* curr = root;
    while (curr) {
        if (!curr->left) {
            process(curr);
            curr = curr->right;
        }
    }
}
```

```

    } else {
        TreeNode* pred = curr->left;
        while (pred->right && pred->right != curr)
            pred = pred->right;

        if (!pred->right) {
            pred->right = curr; // Create thread
            curr = curr->left;
        } else {
            pred->right = nullptr; // Remove thread
            process(curr);
            curr = curr->right;
        }
    }
}
}
}

```

### Advantages:

- $O(1)$  space (no stack/queue)
- Modifies and restores tree
- Each edge traversed at most 3 times

### Technique 2: Parent Pointer Tracking

**Problem:** Find parent without recursion

**Solution:** Use map to track parents during traversal

```

unordered_map parent;
queue q;
q.push(root);
parent[root] = nullptr;

while (!q.empty()) {
    TreeNode* node = q.front(); q.pop();
    if (node->left) {
        parent[node->left] = node;
        q.push(node->left);
    }
    if (node->right) {
        parent[node->right] = node;
        q.push(node->right);
    }
}
}

```

### Use Cases:

- All Nodes Distance K
- LCA problems

- Path finding

### Technique 3: Path Sum with Prefix Sum

**Pattern:** Two-sum technique applied to trees


```
int pathSum(TreeNode* root, int targetSum) {
    unordered_map prefixSum;
    prefixSum[0] = 1;
    return dfs(root, 0, targetSum, prefixSum);
}

int dfs(TreeNode* root, long currSum, int target,
        unordered_map& prefixSum) {
    if (!root) return 0;

    currSum += root->val;
    int count = prefixSum[currSum - target];

    prefixSum[currSum]++;
    count += dfs(root->left, currSum, target, prefixSum);
    count += dfs(root->right, currSum, target, prefixSum);
    prefixSum[currSum]--; // Backtrack

    return count;
}
```

**Key Insight:**  = number of paths ending here with sum x

### Technique 4: Iterative DFS with Stack State

**For problems requiring state tracking:**

```
struct State {
    TreeNode* node;
    int phase; // 0: first visit, 1: after left, 2: after right
    State(TreeNode* n, int p) : node(n), phase(p) {}
};

vector postorderIterative(TreeNode* root) {
    vector result;
    if (!root) return result;
    stack st;
    st.push(State(root, 0));

    while (!st.empty()) {
        State& curr = st.top();

        if (curr.phase == 0) {
            curr.phase = 1;

```

```

        if (curr.node->left)
            st.push(State(curr.node->left, 0));
    } else if (curr.phase == 1) {
        curr.phase = 2;
        if (curr.node->right)
            st.push(State(curr.node->right, 0));
    } else {
        result.push_back(curr.node->val);
        st.pop();
    }
}
return result;
}

```

## Technique 5: Memoization for Tree Problems

**Problem:** Avoid recomputing subtree results

```

class Solution {
    unordered_map memo;
public:
    int rob(TreeNode* root) {
        if (!root) return 0;
        if (memo.count(root)) return memo[root];

        // Option 1: Rob this house
        int robThis = root->val;
        if (root->left)
            robThis += rob(root->left->left) + rob(root->left->right);
        if (root->right)
            robThis += rob(root->right->left) + rob(root->right->right);

        // Option 2: Skip this house
        int skipThis = rob(root->left) + rob(root->right);

        return memo[root] = max(robThis, skipThis);
    }
};

```

---

## Study Notes & Mental Models

### Mental Model 1: The Recursion Tree

**Concept:** Every recursive call creates a subtree of computation

```

      maxDepth(1)
     /      \
maxDepth(2)  maxDepth(3)

```

```

      /      \      \
maxDepth(4)  maxDepth(5)  maxDepth(6)

Returns: 1 + max(left_result, right_result)

```

**Key Insight:** Base case = leaves, build result bottom-up

## Mental Model 2: The Level Sweep

**Think of BFS as sweeping levels:**

```

Level 0: [1]           → Process all nodes at level 0
Level 1: [2, 3]        → Process all nodes at level 1
Level 2: [4, 5, 6]     → Process all nodes at level 2

```

**Pattern:**

1. Know current level size
2. Process exactly that many nodes
3. Add next level nodes to queue

## Mental Model 3: The Two Questions

**For every node, ask:**

1. What do I need from my children?
2. What do I return to my parent?

**Example (Balanced Tree):**

- Need: Height of left and right subtrees
- Return: My height to parent
- Decision: Check if  $|\text{left\_height} - \text{right\_height}| \leq 1$

## Mental Model 4: The Path Collector

**For path problems:**

```

Path = [decisions made from root to current]
At each node:
1. Add current to path
2. Make decision (leaf? target sum?)
3. Recurse to children
4. Remove current from path (backtrack)

```

**Analogy:** Like walking through a maze, marking your trail, and erasing as you backtrack

## Mental Model 5: The BST Navigation

**Think of BST as a decision tree:**

At each node, ask: "Am I looking for something smaller or larger?"  
Smaller → Go left  
Larger → Go right  
Equal → Found it!

**Property:** Every decision eliminates half the search space (if balanced)

---

## Practice Roadmap

### Week 1: Foundation (Easy)

**Day 1-2: Basic Traversal**

1. Maximum Depth (LC 104)
2. Minimum Depth (LC 111)
3. Invert Binary Tree (LC 226)
4. Same Tree (LC 100)

**Day 3-4: Property Validation** 5. Balanced Binary Tree (LC 110) 6. Symmetric Tree (LC 101) 7. Diameter of Binary Tree (LC 543) 8. Binary Tree Paths (LC 257)

**Day 5-6: Basic Path Problems** 9. Path Sum (LC 112) 10. Sum of Left Leaves (LC 404) 11. Merge Two Binary Trees (LC 617) 12. Count Complete Tree Nodes (LC 222)

**Day 7: Review**

- Solve all 12 problems again without hints
- Compare recursive vs iterative approaches
- Document time/space complexities

### Week 2: Intermediate (Medium)

**Day 8-9: Advanced Traversal** 13. Binary Tree Level Order (LC 102) 14. Binary Tree Zigzag Level Order (LC 103) 15. Binary Tree Right Side View (LC 199) 16. Vertical Order Traversal (LC 987)

**Day 10-11: BST Operations** 17. Validate BST (LC 98) 18. Kth Smallest in BST (LC 230) 19. Convert Sorted Array to BST (LC 108) 20. BST Iterator (LC 173)

**Day 12-13: Path Problems Advanced** 21. Path Sum II (LC 113) 22. Path Sum III (LC 437) 23. Sum Root to Leaf Numbers (LC 129) 24. Binary Tree Maximum Path Sum (LC 124)

**Day 14: Review & Analysis**

- Identify common patterns
- Practice pattern recognition
- Time yourself on random problems

## Week 3: Advanced (Medium-Hard)

**Day 15-16: LCA & Construction** 25. Lowest Common Ancestor (LC 236) 26. LCA of BST (LC 235) 27. Construct from Preorder & Inorder (LC 105) 28. Construct from Inorder & Postorder (LC 106)

**Day 17-18: Complex Problems** 29. Serialize and Deserialize (LC 297) 30. Flatten Binary Tree (LC 114) 31. Populating Next Right Pointers (LC 116) 32. All Nodes Distance K (LC 863)

**Day 19-20: Advanced Applications** 33. House Robber III (LC 337) 34. Binary Tree Cameras (LC 968) 35. Distribute Coins (LC 979) 36. Maximum Sum BST in Binary Tree (LC 1373)

### Day 21: Comprehensive Review

- Mixed problem practice
- Focus on optimization techniques
- Analyze multiple approaches per problem

## Week 4: Mastery & Special Structures

**Day 22-23: Trie Problems** 37. Implement Trie (LC 208) 38. Add and Search Word (LC 211) 39. Word Search II (LC 212) 40. Replace Words (LC 648)

**Day 24-25: Advanced Structures** 41. Range Sum Query - Segment Tree 42. Count of Smaller Numbers After Self (LC 315) 43. The Skyline Problem (LC 218)

### Day 26-27: Mock Interviews

- Random medium/hard problems
- Time limit: 45 minutes each
- Explain approach before coding

### Day 28: Final Review

- Review all patterns
- Quick solve foundation problems
- Document learnings

---

## Common Pitfalls & Solutions

### Pitfall 1: Null Pointer Access

**Problem:** Not checking for nullptr before accessing node

```
// WRONG
int maxDepth(TreeNode* root) {
    return 1 + max(maxDepth(root->left), maxDepth(root->right));
}

// CORRECT
int maxDepth(TreeNode* root) {
    if (!root) return 0; // Base case!
```

```
    return 1 + max(maxDepth(root->left), maxDepth(root->right));  
}
```

**Solution:** Always check for nullptr at function start

## Pitfall 2: Wrong Base Case

**Problem:** Incorrect termination condition

```
// WRONG (for leaf nodes)  
if (!root->left && !root->right) return 0;  
  
// CORRECT  
if (!root) return 0;  
if (!root->left && !root->right) return 1; // Leaf has height 1
```

### Common Errors:

- Returning 0 for leaf instead of 1
- Not handling nullptr
- Confusing depth with height

## Pitfall 3: Forgetting to Backtrack

**Problem:** Path not restored after recursion

```
// WRONG  
void findPaths(TreeNode* root, vector& path) {  
    path.push_back(root->val);  
    if (isLeaf(root)) savePath(path);  
    findPaths(root->left, path);  
    findPaths(root->right, path);  
    // Missing path.pop_back()!  
}  
  
// CORRECT  
void findPaths(TreeNode* root, vector& path) {  
    path.push_back(root->val);  
    if (isLeaf(root)) savePath(path);  
    if (root->left) findPaths(root->left, path);  
    if (root->right) findPaths(root->right, path);  
    path.pop_back(); // Backtrack!  
}
```

## Pitfall 4: BST Range Confusion

**Problem:** Using INT\_MIN/MAX instead of LONG\_MIN/MAX

```
// WRONG (fails for INT_MIN/MAX values)
bool isValidBST(TreeNode* root) {
    return validate(root, INT_MIN, INT_MAX);
}

// CORRECT
bool isValidBST(TreeNode* root) {
    return validate(root, LONG_MIN, LONG_MAX);
}
```

## Pitfall 5: Level Order Size Management

**Problem:** Not tracking level size correctly

```
// WRONG
queue q;
q.push(root);
while (!q.empty()) {
    TreeNode* node = q.front(); q.pop();
    // Where does one level end?
}

// CORRECT
queue q;
q.push(root);
while (!q.empty()) {
    int levelSize = q.size(); // Capture size
    for (int i = 0; i < levelSize; i++) {
        TreeNode* node = q.front(); q.pop();
        // Process level
    }
}
```

## Pitfall 6: Return Type Confusion

**Problem:** Returning wrong type or value

```
// Problem asks for node count
int countNodes(TreeNode* root) {
    if (!root) return 0; // ✓
    return 1 + countNodes(root->left) + countNodes(root->right); // ✓
}

// Problem asks for tree depth
int maxDepth(TreeNode* root) {
    if (!root) return 0; // ✓ (not -1!)
    return 1 + max(maxDepth(root->left), maxDepth(root->right)); // ✓
}
```

## Pitfall 7: Modifying Tree During Traversal

**Problem:** Changing structure while iterating

```
// WRONG
void deleteNodes(TreeNode* root, int val) {
    if (!root) return;
    if (root->val == val) {
        delete root; // Dangerous!
        return;
    }
    deleteNodes(root->left, val);
    deleteNodes(root->right, val);
}

// CORRECT
TreeNode* deleteNodes(TreeNode* root, int val) {
    if (!root) return nullptr;
    root->left = deleteNodes(root->left, val);
    root->right = deleteNodes(root->right, val);
    if (root->val == val) {
        // Handle deletion properly
        TreeNode* temp = root->right;
        delete root;
        return temp;
    }
    return root;
}
```

---

## Complexity Analysis Deep Dive

### Time Complexity Patterns

#### 1. Single Traversal: $O(n)$

Visit each node exactly once  
Examples:

- Preorder, Inorder, Postorder
- Level Order
- Count nodes

#### 2. Each Node Processed Once: $O(n)$

Even with complex processing per node  
Examples:

- Serialize/Deserialize

- Path Sum
- Validate BST

### 3. Nested Recursion: $O(n^2)$

For each node, process entire subtree

Examples:

- Subtree of Another Tree (worst case)
- Naive Path Sum III

### 4. BST Operations: $O(\log n)$ to $O(n)$

Balanced BST:  $O(\log n)$

Skewed BST:  $O(n)$

Examples:

- Search, Insert, Delete
- Kth Smallest

## Space Complexity Patterns

### 1. Recursion Stack: $O(h)$

$h$  = height of tree

Balanced:  $O(\log n)$

Skewed:  $O(n)$

### 2. BFS Queue: $O(w)$

$w$  = maximum width

Perfect binary tree:  $O(n/2) = O(n)$

Example: Level Order

### 3. Additional Storage: $O(n)$

Storing all nodes

Examples:

- Inorder traversal result
- Serialization string
- Level-wise storage

### 4. In-place: $O(1)$

Only using pointers  
Examples:

- Morris Traversal
- Tree Modification

Comparison Table

Operation	Balanced BST	Skewed BST	Heap	Trie (m = word length)
Search	O(log n)	O(n)	O(n)	O(m)
Insert	O(log n)	O(n)	O(log n)	O(m)
Delete	O(log n)	O(n)	O(log n)	O(m)
Min/Max	O(log n)	O(n)	O(1)	N/A
Space	O(n)	O(n)	O(n)	O(ALPHABET_SIZE * n * m)

Advanced Techniques (Continued)

Technique 6: Bottom-Up vs Top-Down

Top-Down (Preorder style):

- Pass information from parent to children
- Good for: path tracking, range validation

```
void topDown(TreeNode* root, int pathSum) {
    if (!root) return;
    pathSum += root->val;
    // Use pathSum for current node
    topDown(root->left, pathSum);
    topDown(root->right, pathSum);
}
```

Bottom-Up (Postorder style):

- Compute from children and return to parent
- Good for: subtree properties, aggregation

```
int bottomUp(TreeNode* root) {
    if (!root) return 0;
    int left = bottomUp(root->left);
    int right = bottomUp(root->right);
    // Compute using left and right results
    return computeResult(left, right, root->val);
}
```

## Technique 7: Global Variable vs Return Value

### Global Variable Pattern:

```
class Solution {
    int maxSum = INT_MIN;
public:
    int maxPathSum(TreeNode* root) {
        helper(root);
        return maxSum;
    }

    int helper(TreeNode* root) {
        if (!root) return 0;
        int left = max(0, helper(root->left));
        int right = max(0, helper(root->right));
        maxSum = max(maxSum, left + right + root->val);
        return max(left, right) + root->val;
    }
};
```

### Return Value Pattern:

```
pair helper(TreeNode* root) {
    // Return both: {path_through_node, max_ending_at_node}
    if (!root) return {INT_MIN, 0};
    auto left = helper(root->left);
    auto right = helper(root->right);
    int throughNode = left.second + right.second + root->val;
    int endingHere = max(left.second, right.second) + root->val;
    int maxPath = max({throughNode, left.first, right.first});
    return {maxPath, endingHere};
}
```

## Technique 8: Iterative with Two Stacks

### For Postorder without recursion:

```
vector postorderTwoStacks(TreeNode* root) {
    vector result;
    if (!root) return result;

    stack s1, s2;
    s1.push(root);

    // Push to s2 in reverse postorder
    while (!s1.empty()) {
```

```

        TreeNode* node = s1.top(); s1.pop();
        s2.push(node);
        if (node->left) s1.push(node->left);
        if (node->right) s1.push(node->right);
    }

    // Pop from s2 gives postorder
    while (!s2.empty()) {
        result.push_back(s2.top()->val);
        s2.pop();
    }
    return result;
}

```

## Technique 9: Binary Lifting for LCA

### For multiple LCA queries:

```

class BinaryLifting {
    vector<vector> up; // up[node][i] = 2^i-th ancestor
    vector depth;
    int LOG;

public:
    BinaryLifting(TreeNode* root, int n) {
        LOG = ceil(log2(n)) + 1;
        up.assign(n, vector(LOG, -1));
        depth.assign(n, 0);
        dfs(root, -1, 0);

        // Precompute ancestors
        for (int j = 1; j < LOG; j++) {
            for (int i = 0; i < n; i++) {
                if (up[i][j-1] != -1)
                    up[i][j] = up[up[i][j-1]][j-1];
            }
        }
    }

    void dfs(TreeNode* node, int parent, int d) {
        if (!node) return;
        up[node->val][0] = parent;
        depth[node->val] = d;
        dfs(node->left, node->val, d+1);
        dfs(node->right, node->val, d+1);
    }

    int lca(int u, int v) {
        if (depth[u] < depth[v]) swap(u, v);

        // Bring u to same level as v

```

```

    int diff = depth[u] - depth[v];
    for (int i = 0; i < LOG; i++) {
        if ((diff >> i) & 1)
            u = up[u][i];
    }

    if (u == v) return u;

    // Binary lift both until different
    for (int i = LOG-1; i >= 0; i--) {
        if (up[u][i] != up[v][i]) {
            u = up[u][i];
            v = up[v][i];
        }
    }
    return up[u][0];
}
};

```

**Time:**  $O(n \log n)$  preprocessing,  $O(\log n)$  per query

---

## Problem-Solving Templates

### Template 1: DFS Traversal (Recursive)

```

void dfs(TreeNode* root) {
    // Base case
    if (!root) return;

    // Preorder: Process here
    process(root);

    // Recurse
    dfs(root->left);

    // Inorder: Process here
    // process(root);

    dfs(root->right);

    // Postorder: Process here
    // process(root);
}

```

### Template 2: BFS Traversal

```

void bfs(TreeNode* root) {
    if (!root) return;
}

```

```

queue q;
q.push(root);

while (!q.empty()) {
    int levelSize = q.size();

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

        // Process node
        process(node);

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

    // After level processing
}
}

```

### Template 3: Path Tracking with Backtracking

```

void findPaths(TreeNode* root, vector& path,
               vector<vector>& allPaths) {
    if (!root) return;

    // Add to path
    path.push_back(root->val);

    // Check if target reached
    if (!root->left && !root->right) {
        if (isValidPath(path))
            allPaths.push_back(path);
    }

    // Recurse
    findPaths(root->left, path, allPaths);
    findPaths(root->right, path, allPaths);

    // Backtrack
    path.pop_back();
}

```

### Template 4: BST Search/Insert/Delete

```

// Search
TreeNode* search(TreeNode* root, int val) {
    if (!root || root->val == val) return root;
    return val < root->val ?
        search(root->left, val) :
        search(root->right, val);
}

// Insert
TreeNode* insert(TreeNode* root, int val) {
    if (!root) return new TreeNode(val);
    if (val < root->val)
        root->left = insert(root->left, val);
    else
        root->right = insert(root->right, val);
    return root;
}

// Delete
TreeNode* deleteNode(TreeNode* root, int val) {
    if (!root) return nullptr;

    if (val < root->val) {
        root->left = deleteNode(root->left, val);
    } else if (val > root->val) {
        root->right = deleteNode(root->right, val);
    } else {
        // Found node to delete
        if (!root->left) return root->right;
        if (!root->right) return root->left;

        // Two children: find inorder successor
        TreeNode* minRight = findMin(root->right);
        root->val = minRight->val;
        root->right = deleteNode(root->right, minRight->val);
    }
    return root;
}

TreeNode* findMin(TreeNode* node) {
    while (node->left) node = node->left;
    return node;
}

```

## Template 5: Tree Construction

```

TreeNode* buildTree(vector& preorder, vector& inorder) {
    // Create inorder index map
    unordered_map inMap;
    for (int i = 0; i < inorder.size(); i++)

```

```
        inMap[inorder[i]] = i;

        return build(preorder, 0, preorder.size()-1,
                     inorder, 0, inorder.size()-1, inMap);
    }

    TreeNode* build(vector& pre, int preStart, int preEnd,
                   vector& in, int inStart, int inEnd,
                   unordered_map& inMap) {
        if (preStart > preEnd) return nullptr;

        TreeNode* root = new TreeNode(pre[preStart]);
        int inRoot = inMap[root->val];
        int leftSize = inRoot - inStart;

        root->left = build(pre, preStart+1, preStart+leftSize,
                          in, inStart, inRoot-1, inMap);
        root->right = build(pre, preStart+leftSize+1, preEnd,
                           in, inRoot+1, inEnd, inMap);

        return root;
    }
```

---

## Interview Strategies

### Strategy 1: Clarify Requirements

#### Always ask:

1. Can the tree be empty? (nullptr)
2. Can nodes have duplicate values?
3. Is it a binary tree or BST?
4. What should I return if input is invalid?
5. Are there memory/space constraints?
6. Will there be multiple queries?

### Strategy 2: Start Simple

#### Progression:

1. Explain brute force approach
2. Identify optimization opportunity
3. Implement optimal solution
4. Discuss edge cases
5. Analyze complexity

### Strategy 3: Choose Right Traversal

#### Decision Tree:

Need parent info before children? → Preorder  
Need children info before parent? → Postorder  
Need sorted order (BST)? → Inorder  
Need level-wise processing? → Level Order

## Strategy 4: Test Cases

### Always test:

1. Empty tree (nullptr)
2. Single node
3. Left-skewed tree
4. Right-skewed tree
5. Perfect binary tree
6. Tree with negative values
7. Large tree (performance)

## Strategy 5: Explain as You Code

### Talk through:

1. Base case first
2. Recursive hypothesis
3. Why this approach works
4. Edge cases handled
5. Time/Space complexity

---

## Quick Reference Card

### Tree Properties

Height: Longest path to leaf  
Depth: Distance from root  
Balanced:  $|h_{\text{left}} - h_{\text{right}}| \leq 1$   
Complete: All levels filled left-to-right  
Perfect: All leaves at same level

### Traversal Order

Preorder: Root → Left → Right  
Inorder: Left → Root → Right  
Postorder: Left → Right → Root  
LevelOrder: Level by level (BFS)

## BST Property

Left subtree < Root < Right subtree  
Inorder traversal → Sorted sequence

## Common Patterns

Path Sum → DFS with accumulator  
LCA → DFS returning found nodes  
View → BFS with level tracking  
Validate → DFS with range/inorder  
Construction → Recursion with indices

## Complexity Cheat Sheet

Operation	Avg Case	Worst Case
Traversal	$O(n)$	$O(n)$
BST Search	$O(\log n)$	$O(n)$
Heap Insert	$O(\log n)$	$O(\log n)$
Build Heap	$O(n)$	$O(n)$
Trie Search	$O(m)$	$O(m)$

---

## Resources for Further Study

### Books

1. "Introduction to Algorithms" (CLRS) - Chapter 12-13
2. "Algorithm Design Manual" - Steven Skiena
3. "Elements of Programming Interviews" - Trees chapter

### Online Resources

1. LeetCode Tree Problems Tag
2. GeeksforGeeks Tree Data Structure
3. Visualgo - Tree Visualization
4. Binary Tree Bootcamp - AlgoExpert

### Practice Platforms

1. LeetCode (150+ tree problems)
2. HackerRank (Tree challenges)
3. CodeForces (Tree problems in contests)
4. InterviewBit (Tree practice section)

# Conclusion

## Key Takeaways:

1. **Master the fundamentals:** Understanding basic traversals is crucial
2. **Recognize patterns:** Most problems fall into established patterns
3. **Practice recursion:** Trees are naturally recursive structures
4. **Think bottom-up:** Often easier than top-down for complex problems
5. **Optimize space:** Consider iterative solutions when needed
6. **Use BST properties:** Exploit ordering when available
7. **Draw it out:** Visualize tree structure and recursion
8. **Handle nullptr:** Always check before accessing
9. **Choose right traversal:** Match problem requirements
10. **Test thoroughly:** Cover all edge cases

## Next Steps:

1. Complete Week 1 foundation problems
2. Implement all traversals (recursive + iterative)
3. Solve 5 problems daily
4. Review patterns weekly
5. Time yourself on medium problems
6. Practice explaining solutions out loud
7. Implement without IDE help
8. Compare multiple approaches
9. Join study group or find accountability partner
10. Track progress and weak areas

**Remember:** Tree mastery comes from consistent practice and pattern recognition. Start with easy problems, build intuition, then tackle harder ones. Don't memorize solutions—understand the underlying principles.

Good luck with your tree journey! 🌲