Complete Tree DSA – Deep Mastery Guide

Table of Contents

---

1. Theoretical Foundation of Trees

What is a Tree?

Formal Definition: A Tree T = (V, E) is a connected, acyclic, undirected graph where:

- V is a non-empty set of vertices (nodes)
- E is a set of edges connecting vertices
- There exists exactly one path between any two vertices

Alternative Definitions (All Equivalent):

1. Connected graph with V - 1 edges
2. Acyclic graph with V - 1 edges
3. Connected graph where removing any edge disconnects the graph
4. Acyclic graph where adding any edge creates a cycle
5. Graph with exactly one simple path between any two vertices

Theorem: All five definitions above are equivalent.

Proof Sketch:

- (1) → (2): Connected + V-1 edges → must be acyclic (otherwise cycle would allow removing edge while staying connected)
- (2) → (3): Acyclic + V-1 edges → must be connected (otherwise could add edge without cycle)
- Continue circular proof to show equivalence ∎

Relationship to Graph Theory

Trees are fundamental special cases of graphs:

- **Every tree is a graph**, but not every graph is a tree
- Trees are the minimal connected graphs (removing any edge disconnects)
- Trees are the maximal acyclic graphs (adding any edge creates cycle)

Key Distinction:

```
Graph                          Tree
├── May have cycles            ├── Acyclic (no cycles)
├── |E| can vary               ├── |E| = |V| - 1 (exact)
├── Multiple paths possible    ├── Unique path between nodes
└── May be disconnected        └── Always connected
```

Historical Context

Leonhard Euler (1736): Laid foundations of graph theory, leading to tree concepts

Arthur Cayley (1857):

- Studied enumeration of trees in chemical structure analysis
- **Cayley's Formula**: Number of labeled trees on n vertices = n^(n-2)
- Example: For n=3, there are 3^(3-2) = 3 distinct labeled trees

Gustav Kirchhoff (1847):

- Used trees to analyze electrical circuits
- **Matrix-Tree Theorem**: Relates spanning trees to determinant of Laplacian matrix

Dénes Kőnig (1936): Formalized tree theory in first graph theory textbook

Modern Applications:

- File systems (directory trees)
- Organization hierarchies
- XML/HTML DOM structures
- Database indexing (B-trees, B+ trees)
- Compiler parse trees
- Decision trees in ML
- Network routing protocols

Core Principles

Hierarchy: Trees naturally model hierarchical relationships

- Root at top level
- Parent-child relationships
- Ancestor-descendant relationships

Acyclicity: No circular dependencies

- Enables recursive reasoning
- Guarantees termination
- Simplifies traversal algorithms

Connectivity: Single path property

- Unique parent for each node (except root)
- Path from root to any node is unique
- Enables well-defined depth/level concepts

---

2. Mathematical Properties & Formal Definitions

Fundamental Tree Theorem

Theorem 1 (Edge Count): A tree with n vertices has exactly n - 1 edges.

Proof by Induction:

Base Case: n = 1

- Single vertex, zero edges
- 1 - 1 = 0 ✓

Inductive Hypothesis: Assume true for all trees with k vertices (k ≥ 1)

Inductive Step: Consider tree T with k + 1 vertices

- T is connected, so removing any leaf vertex v (with degree 1) leaves a connected subtree T'
- T' has k vertices
- By IH, T' has k - 1 edges
- Original tree T = T' + vertex v + 1 edge
- Total edges in T = (k - 1) + 1 = k = (k + 1) - 1 ✓

Therefore, by induction, any tree with n vertices has n - 1 edges. ∎

Corollary: If a graph has n vertices and n edges, it contains at least one cycle.

Path Uniqueness Property

Theorem 2: In a tree, there exists exactly one simple path between any two vertices.

Proof by Contradiction:

Assume there exist two distinct simple paths P_1 and P_2 between vertices u and v.

Let w be the first vertex where P_1 and P_2 diverge, and let z be the next vertex where they reconverge.

Then:

- P_1 contains path from w to z: w → ... → z

- P_2 contains path from w to z: w → ... → z (different from P_1)

These two paths form a cycle in the tree, contradicting the definition of a tree (acyclic).

Therefore, only one simple path can exist between any two vertices. ∎

Formal Terminology

Root: Designated node at the top of a rooted tree

- Arbitrary choice in general tree
- Well-defined in specific structures (BST)

Parent: Node directly above current node in tree hierarchy

- Every node except root has exactly one parent
- Notation: parent(v)

Child: Node directly below current node

- A node can have 0 or more children
- Notation: children(v) = $\{c_1, c_2, ..., c_k\}$

Ancestor: Node on path from root to current node

- Includes node itself in reflexive definition
- ancestor(v) = {v, parent(v), parent(parent(v)), ..., root}

Descendant: Node reachable by following child pointers

- Includes node itself in reflexive definition
- Inverse of ancestor relationship

Leaf (External Node): Node with no children

- degree(v) = 1 in unrooted tree
- Terminal node in computations

Internal Node: Non-leaf node

- Has at least one child
- Contains computation logic in expression trees

Sibling: Nodes sharing the same parent

- siblings(v) = children(parent(v)) \ {v}

Subtree: Tree formed by node and all its descendants

- subtree(v) = {v} ∪ descendants(v)
- Critical for recursive algorithms

Height, Depth, and Level

Depth of Node v (depth(v)):

- Number of edges on path from root to v
- depth(root) = 0
- depth(v) = depth(parent(v)) + 1

Height of Node v (height(v)):

- Length of longest path from v to any leaf in its subtree
- height(leaf) = 0
- height(v) = max(height(c) for c in children(v)) + 1

Height of Tree:

- Height of root
- h = max(depth(v) for all vertices v)

Level: Set of all nodes at same depth

- Level k = {v : depth(v) = k}
- Level 0 = {root}

Relationships:

```
For node v in tree of height h:
- 0 ≤ depth(v) ≤ h
- 0 ≤ height(v) ≤ h
- depth(v) + height(v) ≤ h (equality for nodes on longest path)
```

Theorem 3: In a binary tree with n nodes, minimum height = $\lceil \log_2(n+1) \rceil - 1$

Proof:

- Perfect binary tree of height h has 2^(h+1) - 1 nodes
- For n nodes: n ≤ 2^(h+1) - 1
- n + 1 ≤ 2^(h+1)
- $\log_2(n+1) \leq h + 1$
- $h \geq \log_2(n+1) - 1$
- Minimum h = $\lceil \log_2(n+1) \rceil - 1$ ∎

Degree Properties

Degree of Node: Number of edges incident to the node

Theorem 4 (Leaf Count in Binary Tree): In a binary tree, if n_0 = number of leaves and n_2 = number of nodes with 2 children, then:

$n_0 = n_2 + 1$

Proof: Let n_1 = number of nodes with 1 child, n = total nodes, e = edges

- $n = n_0 + n_1 + n_2$
- $e = n - 1$ (tree property)
- $e = 0 \cdot n_0 + 1 \cdot n_1 + 2 \cdot n_2$ (counting edges by parent degree)

Therefore:

- $n - 1 = n_1 + 2n_2$
- $(n_0 + n_1 + n_2) - 1 = n_1 + 2n_2$
- $n_0 + n_1 + n_2 - 1 = n_1 + 2n_2$
- $n_0 = n_2 + 1$ ∎

Applications: Used in Huffman coding analysis, expression tree properties

Recursive Nature of Trees

Fundamental Recursive Property: Every subtree of a tree is itself a tree.

This enables:

1. **Structural recursion**: Break problem into subproblems on subtrees
2. **Inductive proofs**: Prove property for tree by proving for subtrees
3. **Divide-and-conquer algorithms**: Process subtrees independently

Recursive Decomposition:

```
Tree T with root r:
T = {r} ∪ subtree(c₁) ∪ subtree(c₂) ∪ ... ∪ subtree(cₖ)

where c₁, c₂, ..., cₖ are children of r
```

Induction Principle for Trees:

To prove property P holds for all trees:

Base Case: Prove P for trees with 1 node (single vertex)

Inductive Step: For tree T with root r and children c_1, ..., c_k:

- Assume P holds for all subtrees subtree(c_i)
- Prove P holds for T using the inductive hypothesis

This is the foundation for recursive tree algorithms.

---

3. Tree Taxonomy & Classification

Complete Taxonomy

```
TREES
├── General Trees (n-ary trees)
```

```
    ├── Unordered
    └── Ordered (children have specific order)
    │
    ├── Binary Trees
    │   ├── Full Binary Tree
    │   ├── Complete Binary Tree
    │   ├── Perfect Binary Tree
    │   ├── Degenerate/Skewed Binary Tree
    │   └── Balanced Binary Tree
    │
    ├── Binary Search Trees (BST)
    │   ├── Standard BST
    │   ├── AVL Tree (height-balanced)
    │   ├── Red-Black Tree
    │   ├── Splay Tree
    │   └── Treap
    │
    ├── Heap Trees
    │   ├── Min Heap
    │   ├── Max Heap
    │   └── Fibonacci Heap
    │
    ├── Tries (Prefix Trees)
    │   ├── Standard Trie
    │   ├── Compressed Trie (Radix Tree)
    │   └── Suffix Trie
    │
    ├── Segment Trees
    │   ├── Standard Segment Tree
    │   ├── Lazy Propagation Segment Tree
    │   └── Persistent Segment Tree
    │
    ├── Fenwick Trees (Binary Indexed Trees)
    │
    ├── B-Trees & Variants
    │   ├── B-Tree
    │   ├── B+ Tree
    │   └── B* Tree
    │
    └── Specialized Trees
        ├── Huffman Tree
        ├── Expression Tree
        ├── Syntax Tree
        ├── Decision Tree
        └── Quadtree/Octree
```

Binary Tree Classification

1. Full Binary Tree

Definition: Every node has either 0 or 2 children (no nodes with exactly 1 child)

Properties:

- If n = total nodes, then n is always odd
- If n_0 = leaves, then $n = 2n_0 - 1$
- Number of internal nodes = $n_0 - 1$

Example:

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

Use Cases: Expression trees, decision trees where every decision is binary

2. Complete Binary Tree

Definition: All levels completely filled except possibly the last, which is filled from left to right

Properties:

- If height = h, then 2^h ≤ n ≤ 2^(h+1) - 1
- Can be efficiently stored in array representation
- Parent of node i is at index ⌊(i-1)/2⌋
- Left child at 2i + 1, right child at 2i + 2

Example:

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

Use Cases: Heaps, priority queues

Why Array Representation Works: No wasted space except at last level

3. Perfect Binary Tree

Definition: All internal nodes have exactly 2 children, and all leaves are at same level

Properties:

- Height h → exactly 2^(h+1) - 1 nodes
- Number of leaves = 2^h
- Number of internal nodes = 2^h - 1

- Most space-efficient binary tree structure

Example:

```
      1
     / \
    2   3
   / \ / \
  4  5 6  7
```

Use Cases: Theoretical analysis, optimal binary tree structure

Theorem: Perfect binary tree of height h has exactly 2^(h+1) - 1 nodes

Proof:

- Level k has 2^k nodes
- Total nodes = Σ(k=0 to h) 2^k = (2^(h+1) - 1) / (2 - 1) = 2^(h+1) - 1 ∎

4. Degenerate (Skewed) Binary Tree

Definition: Each parent node has only one child, essentially a linked list

Types:

- **Left-skewed**: All nodes have only left children
- **Right-skewed**: All nodes have only right children

Properties:

- Height = n - 1 (worst case)
- Time complexity of operations degrades to O(n)
- Space-inefficient

Example (Right-skewed):

```
1
 \
  2
   \
    3
     \
      4
```

Use Cases: Worst-case analysis, demonstrating need for balancing

5. Balanced Binary Tree

Definition: For every node, height difference between left and right subtrees ≤ k (typically k=1)

Height-Balanced (AVL Definition):

- |height(left) - height(right)| ≤ 1 for all nodes

Properties:

- Height = O(log n)
- Operations remain O(log n)
- Requires rebalancing during modifications

Example:

```
      4
     / \
    2   6
   / \ / \
  1  3 5  7
```

Use Cases: Self-balancing BSTs, database indexing

Binary Search Tree (BST) Property

Definition: For every node v:

- All nodes in left subtree have values < v.val
- All nodes in right subtree have values > v.val
- Both subtrees are also BSTs

Ordering Invariant:

```
For node with value k:
left.val < k < right.val
```

Properties:

- Inorder traversal yields sorted sequence
- Search, insert, delete in O(h) where h = height
- h ranges from O(log n) (balanced) to O(n) (skewed)

Comparison Table

Property	Full	Complete	Perfect	Balanced
Node children	0 or 2	Any	0 or 2	Any
Level filling	Variable	All but last	All levels	Variable
Height (n nodes)	O(log n) to O(n)	O(log n)	O(log n)	O(log n)

Property	Full	Complete	Perfect	Balanced
Array storage	Inefficient	Efficient	Efficient	Inefficient
Node count	Odd only	Any	2^(h+1)-1	Any
Primary use	Structure	Heaps	Theory	BST variants

---

4. Tree Representations & Memory Models

Representation 1: Pointer-Based (Most Common)

Standard Node Structure (C++)

```
// Binary Tree Node
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

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

// N-ary Tree Node
struct Node {
    int val;
    vector<Node*> children;

    Node(int x) : val(x) {}
};

// BST Node with Parent Pointer
struct BSTNode {
    int val;
    BSTNode* left;
    BSTNode* right;
    BSTNode* parent; // Useful for certain operations

    BSTNode(int x) : val(x), left(nullptr), right(nullptr), parent(nullptr) {}
};
```

Memory Layout

```
TreeNode object (64-bit system):
├── val:    4 bytes (int)
├── left:   8 bytes (pointer)
├── right:  8 bytes (pointer)
└── padding: 4 bytes (alignment)
Total: 24 bytes per node
```

```
For n nodes: ~24n bytes + overhead
```

Advantages:

- Natural representation of tree structure
- Easy insertion/deletion
- Flexible for any tree shape
- Direct pointer navigation

Disadvantages:

- Memory overhead (pointers + padding)
- Poor cache locality (nodes scattered in memory)
- Potential for memory leaks if not managed properly

Representation 2: Array-Based (Implicit Tree)

Complete Binary Tree in Array

Index Mapping:

```
// For node at index i (0-indexed):
parent(i)      = (i - 1) / 2
leftChild(i)   = 2 * i + 1
rightChild(i)  = 2 * i + 2

// Check validity:
bool hasLeft(i, n)  { return 2*i + 1 < n; }
bool hasRight(i, n) { return 2*i + 2 < n; }
```

Example:

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

Array: [1, 2, 3, 4, 5, 6]
Index:  0  1  2  3  4  5
```

Advantages:

- No pointer overhead (just values)
- Excellent cache locality
- Simple index arithmetic

- Efficient for complete/perfect trees

Disadvantages:

- Wastes space for sparse trees
- Fixed size (need to resize)
- Only efficient for complete binary trees

When to Use: Heaps, complete binary trees, priority queues

Representation 3: Array with Explicit Parent-Child

```
struct ArrayTree {
    vector<int> values;          // Node values
    vector<int> parent;          // parent[i] = parent of node i
    vector<vector<int>> children; // children[i] = children of node i
    int root;                    // Index of root node

    ArrayTree(int n) : values(n), parent(n, -1), children(n), root(0) {}
};
```

Advantages:

- Efficient for graph-like tree operations
- Easy parent access
- Good for dynamic trees

Disadvantages:

- Higher memory usage
- More complex index management

Representation 4: Left-Child Right-Sibling (LCRS)

Converts n-ary tree to binary tree representation:

```
struct LCRSNode {
    int val;
    LCRSNode* leftChild;   // First child
    LCRSNode* rightSibling; // Next sibling

    LCRSNode(int x) : val(x), leftChild(nullptr), rightSibling(nullptr) {}
};
```

Example Transformation:

```
N-ary Tree:        1
                 / | \
```

```
                2  3  4
               /|
              5 6

LCRS Binary:    1
               /
              2
             / \
            5   3
             \   \
              6   4
```

Use Cases: File systems, DOM representation, converting n-ary to binary

Memory Model Comparison

Representation	Space per Node	Cache Locality	Flexibility	Best For
Pointer	24-32 bytes	Poor	High	General trees
Array (implicit)	4-8 bytes	Excellent	Low	Heaps
Parent-child array	16-24 bytes	Good	Medium	Graph operations
LCRS	16-24 bytes	Medium	High	N-ary trees

Cache Performance Analysis

Cache Line: Typically 64 bytes on modern CPUs

Pointer-Based Trees:

- Nodes scattered in memory
- Each access = potential cache miss
- Tree traversal: O(n) cache misses in worst case

Array-Based Trees:

- Contiguous memory
- Sequential access = cache-friendly
- Tree traversal: O(n/64) cache misses (64-byte lines)

Performance Impact:

```
Experiment (10^6 nodes, level-order traversal):
- Array representation: ~50ms
- Pointer representation: ~200ms
- 4x slowdown due to cache misses
```

---

5. Core Traversal Patterns (DFS/BFS)

Traversal Overview

Tree Traversal: Process of visiting each node exactly once in a systematic way

Classification:

```
Tree Traversals
├── Depth-First Search (DFS)
│   ├── Preorder  (Root → Left → Right)
│   ├── Inorder   (Left → Root → Right)
│   └── Postorder (Left → Right → Root)
│
└── Breadth-First Search (BFS)
    └── Level-order (Level by level)
```

Pattern 1: Preorder Traversal (NLR)

Order: Root → Left → Right

Mnemonic: "Process before descending"

Applications:

- Create copy of tree
- Prefix expression (Polish notation)
- Serialize tree structure
- Directory listing (parent before children)

Recursive Implementation

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

    // Process current node
    cout << root->val << " ";

    // Traverse left subtree
    preorder(root->left);

    // Traverse right subtree
    preorder(root->right);
}
```

Time Complexity: O(n) - each node visited once
Space Complexity: O(h) - recursion stack depth, h = height

Iterative Implementation (Stack)

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

    stack<TreeNode*> 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;
}
```

Key Insight: Stack simulates recursion call stack. Right child pushed before left so left is popped first.

Example

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

Preorder: 1, 2, 4, 5, 3
          ↑  (process root first)
```

Trace:

```
1. Visit 1, go left
2. Visit 2, go left
3. Visit 4 (leaf), backtrack
4. Visit 5 (leaf), backtrack
5. Visit 3 (leaf)
```

Pattern 2: Inorder Traversal (LNR)

Order: Left → Root → Right

Mnemonic: "Process in between"

Applications:

- **BST traversal → sorted order** (most important)
- Infix expression evaluation
- Validate BST
- Find kth smallest in BST

Recursive Implementation

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

    // Traverse left subtree
    inorder(root->left);

    // Process current node
    cout << root->val << " ";

    // Traverse right subtree
    inorder(root->right);
}
```

Iterative Implementation (Stack)

```
vector<int> inorderIterative(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> st;
    TreeNode* curr = root;

    while (curr != nullptr || !st.empty()) {
        // Go to leftmost node
        while (curr != nullptr) {
            st.push(curr);
            curr = curr->left;
        }

        // Process current node
        curr = st.top();
        st.pop();
        result.push_back(curr->val);

        // Move to right subtree
        curr = curr->right;
    }

    return result;
}
```

Pattern: Always go left first, process, then go right

Example

```
BST:        4
           / \
          2   6
         / \ / \
        1  3 5  7

Inorder: 1, 2, 3, 4, 5, 6, 7
         ↑  Sorted order!
```

Theorem: Inorder traversal of BST yields nodes in ascending order

Proof:

- By BST property: left < root < right
- Inorder processes: left → root → right
- By induction on subtrees, result is sorted ∎

Pattern 3: Postorder Traversal (LRN)

Order: Left → Right → Root

Mnemonic: "Process after descending"

Applications:

- Delete tree (delete children before parent)
- Postfix expression (Reverse Polish notation)
- Tree DP (compute from leaves up)
- Calculate directory sizes

Recursive Implementation

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

    // Traverse left subtree
    postorder(root->left);

    // Traverse right subtree
    postorder(root->right);

    // Process current node
    cout << root->val << " ";
}
```

Iterative Implementation (Two Stacks)

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

    stack<TreeNode*> st1, st2;
    st1.push(root);

    // Fill st2 with reverse postorder
    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);
    }

    // Pop from st2 to get postorder
    while (!st2.empty()) {
        result.push_back(st2.top()->val);
        st2.pop();
    }

    return result;
}
```

Insight: Postorder = reverse of modified preorder (Root → Right → Left)

Iterative Implementation (One Stack)

```
vector<int> postorderOneStack(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> st;
    TreeNode* curr = root;
    TreeNode* lastVisited = nullptr;

    while (curr != nullptr || !st.empty()) {
        // Go to leftmost
        while (curr != nullptr) {
            st.push(curr);
            curr = curr->left;
        }

        TreeNode* peekNode = st.top();

        // If right child exists and not processed
        if (peekNode->right != nullptr && lastVisited != peekNode->right) {
            curr = peekNode->right;
```

```
        } else {
            result.push_back(peekNode->val);
            lastVisited = st.top();
            st.pop();
        }
    }

    return result;
}
```

Key: Track last visited node to avoid reprocessing

Example

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

Postorder: 4, 5, 2, 3, 1
                      ↑ (root last)
```

Use Case - Delete Tree:

```
void deleteTree(TreeNode* root) {
    if (root == nullptr) return;

    deleteTree(root->left);   // Delete left subtree
    deleteTree(root->right);  // Delete right subtree
    delete root;              // Delete root (postorder)
}
```

Pattern 4: Level-order Traversal (BFS)

Order: Level by level, left to right

Applications:

- Shortest path in unweighted tree
- Serialize tree level by level
- Level-order construction
- Minimum depth problems

Standard Implementation (Queue)

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

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

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

        // Process all nodes at current level
        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;
}
```

Time Complexity: O(n)
Space Complexity: O(w) where w = maximum width of tree

Maximum Width:

- Best case (skewed): O(1)
- Worst case (complete): O(n/2) = O(n)

Example

```
Tree:       1
           / \
          2   3
         / \ / \
        4  5 6  7

Level-order: [[1], [2, 3], [4, 5, 6, 7]]
```

Variant: Right-side View

```cpp
vector<int> rightSideView(TreeNode* root) {
    vector<int> result;
    if (root == nullptr) return result;

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

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

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

            // Last node of each level
            if (i == levelSize - 1) {
                result.push_back(node->val);
            }

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

    return result;
}
```

Traversal Comparison Table

Traversal	Order	Key Use	Recursion Pattern	Iterative Tool
Preorder	NLR	Copy/Serialize	Process first	Stack (R then L)
Inorder	LNR	BST sorting	Process middle	Stack + curr pointer
Postorder	LRN	Delete/DP	Process last	Two stacks or tracking
Level-order	By level	Shortest path	Not natural	Queue

Traversal Invariants

Preorder Invariant: Parent appears before all descendants in output

Inorder Invariant: For BST, output is sorted

Postorder Invariant: Parent appears after all descendants in output

Level-order Invariant: Nodes at depth d appear before nodes at depth d+1

---

6. Binary Tree Core Patterns

Pattern 6.1: Height and Depth

Problem: Find Height of Binary Tree

Height Definition: Longest path from node to leaf (in number of edges)

Recursive Formulation:

```
height(null) = -1  (or 0 depending on convention)
height(node) = 1 + max(height(left), height(right))
```

Implementation

```
int height(TreeNode* root) {
    // Base case: empty tree
    if (root == nullptr) return -1; // or 0

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

    return 1 + max(leftHeight, rightHeight);
}
```

Time Complexity: O(n) - visit each node once
Space Complexity: O(h) - recursion stack

Proof of Correctness:

Base Case: Empty tree has height -1 ✓

Inductive Step: Assume correct for all subtrees

- Height of tree = 1 + max(height of subtrees)
- By IH, subtree heights are correct
- Therefore, tree height is correct ∎

Common Mistake

```
// WRONG: Count nodes instead of edges
int heightWrong(TreeNode* root) {
    if (root == nullptr) return 0;
    return 1 + max(heightWrong(root->left), heightWrong(root->right));
}
```

This counts nodes on path, not edges. For single node, returns 1 instead of 0.

Pattern 6.2: Diameter of Binary Tree

Problem: Find length of longest path between any two nodes

Key Insight: Longest path may or may not pass through root

Approach 1: Naive O(n^2)

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

    // Case 1: Path passes through root
    int throughRoot = height(root->left) + height(root->right) + 2;

    // Case 2: Path in left subtree
    int leftDiameter = diameter(root->left);

    // Case 3: Path in right subtree
    int rightDiameter = diameter(root->right);

    return max({throughRoot, leftDiameter, rightDiameter});
}
```

Problem: Recalculates height repeatedly → O(n^2)

Approach 2: Optimized O(n)

```
class Solution {
private:
    int maxDiameter = 0;

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

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

        // Update diameter: path through this node
        int currentDiameter = leftHeight + rightHeight;
        maxDiameter = max(maxDiameter, currentDiameter);

        // Return height for parent calculation
        return 1 + max(leftHeight, rightHeight);
    }

public:
    int diameterOfBinaryTree(TreeNode* root) {
        heightAndDiameter(root);
        return maxDiameter;
```

```
    }
};
```

Pattern: Calculate height + update diameter in single pass

Time Complexity: O(n)
Space Complexity: O(h)

Pattern 6.3: Balanced Binary Tree Check

Problem: Determine if tree is height-balanced

Definition: For every node, |height(left) - height(right)| ≤ 1

Approach 1: Top-Down O(n^2)

```
bool isBalanced(TreeNode* root) {
    if (root == nullptr) return true;

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

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

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

Problem: Recalculates height → O(n^2)

Approach 2: Bottom-Up O(n)

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

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

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

        // Check current node balance
        if (abs(leftHeight - rightHeight) > 1) return -1;

        return 1 + max(leftHeight, rightHeight);
    }

public:
```

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

Pattern: Use special value (-1) to propagate "unbalanced" status

Time Complexity: O(n)
Space Complexity: O(h)

Pattern 6.4: Symmetric Tree

Problem: Check if tree is mirror of itself

Recursive Approach

```
bool isMirror(TreeNode* left, TreeNode* right) {
    // Both null → symmetric
    if (left == nullptr && right == nullptr) return true;

    // One null → not symmetric
    if (left == nullptr || right == nullptr) return false;

    // Check: values equal + subtrees mirror
    return (left->val == right->val) &&
           isMirror(left->left, right->right) &&
           isMirror(left->right, right->left);
}

bool isSymmetric(TreeNode* root) {
    if (root == nullptr) return true;
    return isMirror(root->left, root->right);
}
```

Time Complexity: O(n)
Space Complexity: O(h)

Iterative Approach (Queue)

```
bool isSymmetric(TreeNode* root) {
    if (root == nullptr) return true;

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

    while (!q.empty()) {
        TreeNode* left = q.front(); q.pop();
```

```
        TreeNode* right = q.front(); q.pop();

        if (left == nullptr && right == nullptr) continue;
        if (left == nullptr || right == nullptr) return false;
        if (left->val != right->val) return false;

        // Enqueue in mirror order
        q.push(left->left);
        q.push(right->right);
        q.push(left->right);
        q.push(right->left);
    }

    return true;
}
```

Pattern 6.5: Path Sum Problems

Problem: Check if Path Sum Exists

```
bool hasPathSum(TreeNode* root, int targetSum) {
    // Base case: empty tree
    if (root == nullptr) return false;

    // Leaf node: check if sum matches
    if (root->left == nullptr && root->right == nullptr) {
        return root->val == targetSum;
    }

    // Recurse with reduced target
    int remainingSum = targetSum - root->val;
    return hasPathSum(root->left, remainingSum) ||
           hasPathSum(root->right, remainingSum);
}
```

Time Complexity: O(n)
Space Complexity: O(h)

Problem: Find All Root-to-Leaf Paths with Sum

```
class Solution {
private:
    void findPaths(TreeNode* root, int targetSum,
                   vector<int>& path, vector<vector<int>>& result) {
        if (root == nullptr) return;

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

```
        // Check if leaf with target sum
        if (root->left == nullptr && root->right == nullptr) {
            if (root->val == targetSum) {
                result.push_back(path);
            }
        } else {
            // Recurse to children
            int remaining = targetSum - root->val;
            findPaths(root->left, remaining, path, result);
            findPaths(root->right, remaining, path, result);
        }

        // Backtrack: remove current node
        path.pop_back();
    }

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

Pattern: Backtracking on trees

Pattern 6.6: Lowest Common Ancestor (LCA)

Problem: Find lowest (deepest) node that is ancestor of both p and q

Approach: Recursive

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    // Base case
    if (root == nullptr || 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 both found in different subtrees → current is LCA
    if (left != nullptr && right != nullptr) {
        return root;
    }

    // Otherwise, return non-null child (or null if both null)
```

```
    return left != nullptr ? left : right;
}
```

Time Complexity: O(n)
Space Complexity: O(h)

Proof of Correctness:

Case 1: p and q in different subtrees

- left != nullptr and right != nullptr
- Current node is LCA ✓

Case 2: Both in left subtree

- right == nullptr
- LCA is in left subtree, return left ✓

Case 3: Both in right subtree

- left == nullptr
- LCA is in right subtree, return right ✓

LCA in BST (Optimized)

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    // Leverage BST property
    while (root != nullptr) {
        // Both in left subtree
        if (p->val < root->val && q->val < root->val) {
            root = root->left;
        }
        // Both in right subtree
        else if (p->val > root->val && q->val > root->val) {
            root = root->right;
        }
        // Split point found → this is LCA
        else {
            return root;
        }
    }
    return nullptr;
}
```

Time Complexity: O(h)
Space Complexity: O(1)

---

7. Binary Search Tree (BST) Deep Analysis

BST Ordering Invariant

Definition: For every node v in BST:

```
∀ node x in left subtree:  x.val < v.val
∀ node y in right subtree: y.val > v.val
```

Key Property: This must hold recursively for all nodes

Why Inorder Traversal Works

Theorem: Inorder traversal of BST produces values in sorted order

Proof by Induction:

Base Case: Single node tree

- Inorder: [node.val]
- Trivially sorted ✓

Inductive Hypothesis: Assume true for all BSTs with n ≤ k nodes

Inductive Step: Consider BST T with k+1 nodes, root r

- Let L = left subtree, R = right subtree
- By BST property: all values in L < r < all values in R
- By IH: inorder(L) is sorted, inorder(R) is sorted
- Inorder(T) = inorder(L) + [r] + inorder(R)
- Since max(L) < r < min(R), result is sorted ✓

BST Operations

1. Search

```
TreeNode* search(TreeNode* root, int target) {
    // Base case: not found or found
    if (root == nullptr || root->val == target) {
        return root;
    }

    // Search left or right based on BST property
    if (target < root->val) {
        return search(root->left, target);
    } else {
        return search(root->right, target);
    }
}

// Iterative version (more efficient)
TreeNode* searchIterative(TreeNode* root, int target) {
```

```
    while (root != nullptr && root->val != target) {
        root = (target < root->val) ? root->left : root->right;
    }
    return root;
}
```

Time Complexity: O(h) where h = height

- Best case: O(log n) (balanced)
- Worst case: O(n) (skewed)

Space Complexity:

- Recursive: O(h)
- Iterative: O(1)

2. Insert

```
TreeNode* insert(TreeNode* root, int val) {
    // Base case: found insertion point
    if (root == nullptr) {
        return new TreeNode(val);
    }

    // Recursive case: go left or right
    if (val < root->val) {
        root->left = insert(root->left, val);
    } else if (val > root->val) {
        root->right = insert(root->right, val);
    }
    // If val == root->val, don't insert (no duplicates)

    return root;
}
```

Time Complexity: O(h)
Space Complexity: O(h)

Invariant Preservation: After insertion, BST property maintained

3. Delete (Most Complex)

Three Cases:

1. **Node is leaf**: Simply remove
2. **Node has one child**: Replace with child
3. **Node has two children**: Replace with inorder successor/predecessor

```
TreeNode* deleteNode(TreeNode* root, int key) {
    if (root == nullptr) return nullptr;

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

        // Case 1: Leaf or one child
        if (root->left == nullptr) {
            TreeNode* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            TreeNode* temp = root->left;
            delete root;
            return temp;
        }

        // Case 2: Two children
        // Find inorder successor (leftmost in right subtree)
        TreeNode* successor = findMin(root->right);

        // Copy successor's value to current node
        root->val = successor->val;

        // Delete successor
        root->right = deleteNode(root->right, successor->val);
    }

    return root;
}

TreeNode* findMin(TreeNode* root) {
    while (root->left != nullptr) {
        root = root->left;
    }
    return root;
}
```

Time Complexity: O(h)
Space Complexity: O(h)

Why Inorder Successor?

- Smallest value greater than current
- Maintains BST property when substituted
- Located in right subtree

BST Validation

Problem: Verify if tree is valid BST

Incorrect Approach

```
// WRONG: Only checks immediate children
bool isValidBST(TreeNode* root) {
    if (root == nullptr) return true;

    if (root->left && root->left->val >= root->val) return false;
    if (root->right && root->right->val <= root->val) return false;

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

Problem: Doesn't check entire subtree constraints

Counterexample:

```
    5
   / \
  1   6
     / \
    4   7  ← Invalid! 4 < 5 but in right subtree
```

Correct Approach: Range Constraints

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

bool validate(TreeNode* root, long minVal, long maxVal) {
    // Empty tree is valid
    if (root == nullptr) return true;

    // Check current node against range
    if (root->val <= minVal || root->val >= maxVal) {
        return false;
    }

    // Recursively validate subtrees with updated ranges
    return validate(root->left, minVal, root->val) &&
           validate(root->right, root->val, maxVal);
}
```

Time Complexity: O(n)
Space Complexity: O(h)

Key Insight: Pass down valid range for each subtree

Alternative: Inorder Traversal Check

```
class Solution {
private:
    TreeNode* prev = nullptr;

public:
    bool isValidBST(TreeNode* root) {
        if (root == nullptr) return true;

        // Check left subtree
        if (!isValidBST(root->left)) return false;

        // Check current node (inorder position)
        if (prev != nullptr && prev->val >= root->val) {
            return false;
        }
        prev = root;

        // Check right subtree
        return isValidBST(root->right);
    }
};
```

Property Used: Inorder traversal should be strictly increasing

Kth Smallest Element in BST

Problem: Find kth smallest value (1-indexed)

Approach 1: Inorder Traversal

```
class Solution {
private:
    int count = 0;
    int result = -1;

    void inorder(TreeNode* root, int k) {
        if (root == nullptr) return;

        inorder(root->left, k);

        count++;
        if (count == k) {
```

```
            result = root->val;
            return;
        }

        inorder(root->right, k);
    }

public:
    int kthSmallest(TreeNode* root, int k) {
        inorder(root, k);
        return result;
    }
};
```

Time Complexity: O(n) worst case, O(k) average
Space Complexity: O(h)

Optimization: Augmented BST

Idea: Store subtree size in each node

```
struct AugmentedNode {
    int val;
    int leftSize;  // Number of nodes in left subtree
    AugmentedNode* left;
    AugmentedNode* right;

    AugmentedNode(int x) : val(x), leftSize(0), left(nullptr), right(nullptr) {}
};

int kthSmallest(AugmentedNode* root, int k) {
    if (root == nullptr) return -1;

    int leftCount = root->leftSize + 1;  // +1 for root

    if (k == leftCount) {
        return root->val;
    } else if (k < leftCount) {
        return kthSmallest(root->left, k);
    } else {
        return kthSmallest(root->right, k - leftCount);
    }
}
```

Time Complexity: O(h)
Space Complexity: O(h)

Tradeoff: Faster queries, but need to maintain leftSize during insertions/deletions

Range Sum Query in BST

Problem: Sum of all values in range [L, R]

```
int rangeSumBST(TreeNode* root, int L, int R) {
    if (root == nullptr) return 0;

    // Current node outside range
    if (root->val < L) {
        // Only explore right subtree
        return rangeSumBST(root->right, L, R);
    }
    if (root->val > R) {
        // Only explore left subtree
        return rangeSumBST(root->left, L, R);
    }

    // Current node in range
    return root->val +
           rangeSumBST(root->left, L, R) +
           rangeSumBST(root->right, L, R);
}
```

Time Complexity: O(n) worst case, O(h + k) where k = nodes in range
Space Complexity: O(h)

Optimization: Pruning based on BST property

---

8. Balanced Trees & Height Optimization

Why Balancing Matters

Unbalanced BST:

```
Worst case (skewed):    1
                         \
                          2
                           \
                            3
                             \
                              4

Height: O(n)
Search/Insert/Delete: O(n)
```

Balanced BST:

```
        2
       / \
```

```
      1   3
           \
            4

Height: O(log n)
Search/Insert/Delete: O(log n)
```

Goal: Maintain height = O(log n) for n nodes

AVL Trees

Definition: Self-balancing BST where height difference between left and right subtrees ≤ 1 for all nodes

Balance Factor:

```
BF(node) = height(left) - height(right)
```

AVL Property: |BF(node)| ≤ 1 for all nodes

AVL Height Guarantee

Theorem: AVL tree with n nodes has height $h \leq 1.44 \log_2(n+2) - 0.328$

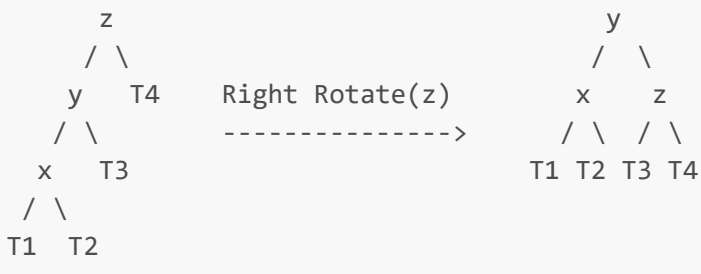
Proof Sketch: Using Fibonacci sequence analysis

- Minimum nodes in AVL tree of height h follows Fibonacci recurrence
- N(h) ≥ F(h+3) - 1 where F(k) is kth Fibonacci number
- Solving gives h = O(log n) ∎

AVL Rotations

Four Cases:

1. Left-Left (LL) Case:




```
TreeNode* rightRotate(TreeNode* z) {
    TreeNode* y = z->left;
    TreeNode* T3 = y->right;
```

```
    // Perform rotation
    y->right = z;
    z->left = T3;

    // Update heights
    z->height = 1 + max(height(z->left), height(z->right));
    y->height = 1 + max(height(y->left), height(y->right));

    return y;  // New root
}
```

2. Right-Right (RR) Case:

```
    z                                y
   / \                              / \
  T1  y      Left Rotate(z)        z   x
     / \     ------------->       / \ / \
    T2  x                       T1 T2 T3 T4
       / \
      T3 T4
```

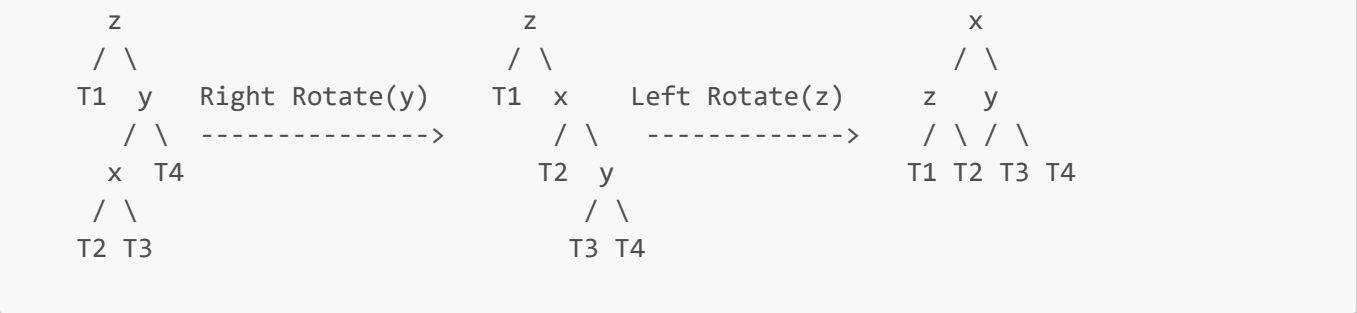
```
TreeNode* leftRotate(TreeNode* z) {
    TreeNode* y = z->right;
    TreeNode* T2 = y->left;

    // Perform rotation
    y->left = z;
    z->right = T2;

    // Update heights
    z->height = 1 + max(height(z->left), height(z->right));
    y->height = 1 + max(height(y->left), height(y->right));

    return y;  // New root
}
```

3. Left-Right (LR) Case:

```
       z                               z                           x
      / \                             / \                         / \
     y   T4  Left Rotate(y)          x   T4  Right Rotate(z)     y   z
    / \      -------------->        / \       ------------->    / \ / \
   T1  x                           y  T3                       T1 T2 T3 T4
      / \                         / \
     T2 T3                       T1 T2
```

```
// LR Case: First left rotate on left child, then right rotate on root
root->left = leftRotate(root->left);
return rightRotate(root);
```

4. Right-Left (RL) Case:




```
// RL Case: First right rotate on right child, then left rotate on root
root->right = rightRotate(root->right);
return leftRotate(root);
```

Complete AVL Insert

```
struct AVLNode {
    int val;
    AVLNode* left;
    AVLNode* right;
    int height;

    AVLNode(int x) : val(x), left(nullptr), right(nullptr), height(1) {}
};

int height(AVLNode* node) {
    return node == nullptr ? 0 : node->height;
}

int getBalance(AVLNode* node) {
    return node == nullptr ? 0 : height(node->left) - height(node->right);
}

AVLNode* insert(AVLNode* root, int val) {
    // 1. Perform normal BST insert
    if (root == nullptr) {
        return new AVLNode(val);
    }

    if (val < root->val) {
        root->left = insert(root->left, val);
    } else if (val > root->val) {
```

```
            root->right = insert(root->right, val);
        } else {
            return root;  // Duplicate values not allowed
        }

        // 2. Update height
        root->height = 1 + max(height(root->left), height(root->right));

        // 3. Get balance factor
        int balance = getBalance(root);

        // 4. If unbalanced, perform rotations

        // Left-Left Case
        if (balance > 1 && val < root->left->val) {
            return rightRotate(root);
        }

        // Right-Right Case
        if (balance < -1 && val > root->right->val) {
            return leftRotate(root);
        }

        // Left-Right Case
        if (balance > 1 && val > root->left->val) {
            root->left = leftRotate(root->left);
            return rightRotate(root);
        }

        // Right-Left Case
        if (balance < -1 && val < root->right->val) {
            root->right = rightRotate(root->right);
            return leftRotate(root);
        }

        return root;
    }
```

Time Complexity: O(log n) guaranteed
Space Complexity: O(log n)

Red-Black Trees

Properties:

1. Every node is either red or black
2. Root is black
3. All leaves (nil) are black
4. Red node has only black children (no two red nodes in a row)
5. All paths from node to leaves contain same number of black nodes

Height Guarantee: $h \leq 2 \log_2(n+1)$

Why?

- Black height (number of black nodes on path) = bh
- Minimum nodes in tree with black height bh = 2^bh - 1
- n ≥ 2^bh - 1
- bh ≤ $\log_2(n+1)$
- Total height h ≤ 2·bh ≤ 2 $\log_2(n+1)$ ∎

Comparison: AVL vs Red-Black:

Property	AVL	Red-Black
Height bound	1.44 log n	2 log n
Balance strictness	Strict	Relaxed
Rotations (insert)	≤ 2	≤ 2
Rotations (delete)	O(log n)	≤ 3
Search	Faster	Slightly slower
Insert/Delete	Slower	Faster
Use case	Read-heavy	Write-heavy

Red-Black in Practice: Used in C++ `std::map`, Java `TreeMap`, Linux kernel

---

9. Recursive Thinking & Tree DP

Tree DP Fundamentals

Core Idea: Bottom-up dynamic programming on tree structure

Pattern:

1. Process children first (postorder-like)
2. Combine children's results
3. Return result for parent to use

Template:

```
ResultType treeDP(TreeNode* root) {
    // Base case
    if (root == nullptr) return baseValue;

    // Recurse on children
    ResultType leftResult = treeDP(root->left);
    ResultType rightResult = treeDP(root->right);

    // Combine and return
```

```
    return combine(root, leftResult, rightResult);
}
```

Pattern 9.1: Maximum Path Sum

Problem: Find maximum path sum in binary tree (LC 124)

Path: Sequence of nodes where each node appears at most once

Key Insight: At each node, decide whether to:

1. Include in path from parent
2. Start new path here

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

    // Returns: max path sum that can be extended to parent
    int maxPathDown(TreeNode* root) {
        if (root == nullptr) return 0;

        // Max path sum from left and right children
        // (only positive contributions help)
        int left = max(0, maxPathDown(root->left));
        int right = max(0, maxPathDown(root->right));

        // Update global max: path through current node
        int pathThroughNode = root->val + left + right;
        maxSum = max(maxSum, pathThroughNode);

        // Return max path that can be extended to parent
        // Can only include one child
        return root->val + max(left, right);
    }

public:
    int maxPathSum(TreeNode* root) {
        maxPathDown(root);
        return maxSum;
    }
};
```

Time Complexity: O(n)
Space Complexity: O(h)

Why It Works:

- Each node computes two things:
    1. Best path through itself (for global max)

2. Best path to extend upward (for parent)

- Greedy choice: Take positive contributions only

Pattern 9.2: House Robber III

Problem: Rob houses in binary tree; can't rob adjacent nodes

State Definition:

```
rob[node] = {
    with: max money if rob this node,
    without: max money if don't rob this node
}
```

```
class Solution {
private:
    pair<int, int> robDFS(TreeNode* root) {
        if (root == nullptr) return {0, 0};

        auto [leftWith, leftWithout] = robDFS(root->left);
        auto [rightWith, rightWithout] = robDFS(root->right);

        // If rob current node, can't rob children
        int withCurrent = root->val + leftWithout + rightWithout;

        // If don't rob current, take max from children
        int withoutCurrent = max(leftWith, leftWithout) +
                             max(rightWith, rightWithout);

        return {withCurrent, withoutCurrent};
    }

public:
    int rob(TreeNode* root) {
        auto [withRoot, withoutRoot] = robDFS(root);
        return max(withRoot, withoutRoot);
    }
};
```

Time Complexity: O(n)
Space Complexity: O(h)

DP Recurrence:

```
with[node] = node.val + without[left] + without[right]
without[node] = max(with[left], without[left]) +
                max(with[right], without[right])
```

Pattern 9.3: Tree Diameter Revisited (DP Perspective)

State: At each node, track maximum path length in its subtree

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

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

        int L = depth(root->left);
        int R = depth(root->right);

        // DP update: path through current node
        diameter = max(diameter, L + R);

        // Return depth for parent
        return 1 + max(L, R);
    }

public:
    int diameterOfBinaryTree(TreeNode* root) {
        depth(root);
        return diameter;
    }
};
```

Pattern 9.4: Binary Tree Cameras

Problem: Minimum cameras to monitor all nodes; camera at node covers itself, parent, and children

States:

```
0: Node not covered, no camera
1: Node covered, no camera
2: Node covered, has camera
```

```
class Solution {
private:
    int cameras = 0;

    // Returns state of node
    int dfs(TreeNode* root) {
        if (root == nullptr) return 1;  // Null = covered

        int left = dfs(root->left);
        int right = dfs(root->right);
```

```
        // If any child not covered, must place camera here
        if (left == 0 || right == 0) {
            cameras++;
            return 2;  // Has camera
        }

        // If any child has camera, current is covered
        if (left == 2 || right == 2) {
            return 1;  // Covered, no camera
        }

        // Both children covered but no camera nearby
        // Leave current uncovered (parent will handle)
        return 0;  // Not covered
    }

public:
    int minCameraCover(TreeNode* root) {
        // If root not covered after DFS, add camera at root
        if (dfs(root) == 0) cameras++;
        return cameras;
    }
};
```

Greedy Strategy: Place cameras as low as possible (leaf parents)

Pattern 9.5: Subtree DP Pattern

General Form: Compute property of each subtree

```
struct SubtreeInfo {
    int size;      // Number of nodes
    int sum;       // Sum of values
    bool isBST;    // Is valid BST
    int minVal;    // Minimum value
    int maxVal;    // Maximum value

    // Can have multiple properties
};

SubtreeInfo computeSubtree(TreeNode* root) {
    if (root == nullptr) {
        return {0, 0, true, INT_MAX, INT_MIN};
    }

    SubtreeInfo left = computeSubtree(root->left);
    SubtreeInfo right = computeSubtree(root->right);

    SubtreeInfo current;
    current.size = 1 + left.size + right.size;
```

```
    current.sum = root->val + left.sum + right.sum;

    // Check BST property
    current.isBST = left.isBST && right.isBST &&
                    root->val > left.maxVal &&
                    root->val < right.minVal;

    current.minVal = min({root->val, left.minVal, right.minVal});
    current.maxVal = max({root->val, left.maxVal, right.maxVal});

    return current;
}
```

Applications:

- Largest BST subtree
- Count nodes with sum in range
- Subtree with maximum average

---

10. Advanced Tree Algorithms

Morris Traversal (Threaded Binary Tree)

Goal: Inorder traversal with O(1) space (no stack/recursion)

Idea: Use null right pointers to create temporary links back to ancestors

Algorithm:

1. If left child exists, find inorder predecessor
2. Create temporary link from predecessor to current
3. When revisit via link, remove link and move right
4. If no left child, process current and move right

```
vector<int> morrisInorder(TreeNode* root) {
    vector<int> result;
    TreeNode* curr = root;

    while (curr != nullptr) {
        if (curr->left == nullptr) {
            // No left subtree, process current and go right
            result.push_back(curr->val);
            curr = curr->right;
        } else {
            // Find inorder predecessor
            TreeNode* pred = curr->left;
            while (pred->right != nullptr && pred->right != curr) {
                pred = pred->right;
            }
```

```
            if (pred->right == nullptr) {
                // Create thread
                pred->right = curr;
                curr = curr->left;
            } else {
                // Thread exists, remove it and process current
                pred->right = nullptr;
                result.push_back(curr->val);
                curr = curr->right;
            }
        }
    }

    return result;
}
```

Time Complexity: O(n) - each edge traversed at most twice
Space Complexity: O(1) - only pointers, no stack

Proof of O(n) Time:

- Each edge examined at most twice:
    1. Once when creating thread
    2. Once when removing thread
- Total edge examinations = O(n) ∎

Applications:

- Memory-constrained environments
- Tree traversal without recursion
- Educational value (understanding tree structure)

Serialize and Deserialize Binary Tree

Problem: Convert tree to string and reconstruct from string

Preorder Serialization

```
class Codec {
public:
    // Serialize tree to string
    string serialize(TreeNode* root) {
        if (root == nullptr) return "#";

        return to_string(root->val) + "," +
               serialize(root->left) + "," +
               serialize(root->right);
    }

    // Deserialize string to tree
    TreeNode* deserialize(string data) {
```

```
        queue<string> nodes;
        stringstream ss(data);
        string item;

        while (getline(ss, item, ',')) {
            nodes.push(item);
        }

        return deserializeHelper(nodes);
    }

private:
    TreeNode* deserializeHelper(queue<string>& nodes) {
        string val = nodes.front();
        nodes.pop();

        if (val == "#") return nullptr;

        TreeNode* root = new TreeNode(stoi(val));
        root->left = deserializeHelper(nodes);
        root->right = deserializeHelper(nodes);

        return root;
    }
};
```

Time Complexity: O(n)
Space Complexity: O(n)

Why Preorder?: Allows reconstruction without ambiguity

Flatten Binary Tree to Linked List

Problem: Flatten tree to right-skewed "linked list" using right pointers

Requirement: Use preorder traversal order

```
void flatten(TreeNode* root) {
    if (root == nullptr) return;

    // Flatten left and right subtrees
    flatten(root->left);
    flatten(root->right);

    // Save right subtree
    TreeNode* tempRight = root->right;

    // Move left subtree to right
    root->right = root->left;
    root->left = nullptr;

    // Find end of new right subtree
```

```
    TreeNode* curr = root;
    while (curr->right != nullptr) {
        curr = curr->right;
    }

    // Attach original right subtree
    curr->right = tempRight;
}
```

Time Complexity: O(n^2) due to finding end each time

Optimized O(n) Version

```
class Solution {
private:
    TreeNode* prev = nullptr;

public:
    void flatten(TreeNode* root) {
        if (root == nullptr) return;

        // Reverse postorder: right, left, root
        flatten(root->right);
        flatten(root->left);

        // Connect current to previous
        root->right = prev;
        root->left = nullptr;
        prev = root;
    }
};
```

Time Complexity: O(n)
Space Complexity: O(h)

Vertical Order Traversal

Problem: Return nodes by vertical column from left to right

Definition: Column index = horizontal distance from root

- Root at column 0
- Left child at column - 1
- Right child at column + 1

```
vector<vector<int>> verticalTraversal(TreeNode* root) {
    map<int, map<int, multiset<int>>> nodes;  // {col -> {row -> values}}
    queue<tuple<TreeNode*, int, int>> 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<int>> result;
    for (auto& [col, rows] : nodes) {
        vector<int> column;
        for (auto& [row, vals] : rows) {
            column.insert(column.end(), vals.begin(), vals.end());
        }
        result.push_back(column);
    }

    return result;
}
```

Time Complexity: O(n log n) due to multiset insertions
Space Complexity: O(n)

Boundary Traversal

Problem: Print boundary nodes (left boundary + leaves + right boundary)

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

    if (!isLeaf(root)) result.push_back(root->val);

    // Add left boundary (top to bottom, excluding leaves)
    addLeftBoundary(root->left, result);

    // Add all leaves (left to right)
    addLeaves(root, result);

    // Add right boundary (bottom to top, excluding leaves)
    addRightBoundary(root->right, result);

    return result;
}

bool isLeaf(TreeNode* node) {
    return node && !node->left && !node->right;
```

```
}

void addLeftBoundary(TreeNode* node, vector<int>& result) {
    while (node) {
        if (!isLeaf(node)) result.push_back(node->val);
        node = node->left ? node->left : node->right;
    }
}

void addLeaves(TreeNode* node, vector<int>& result) {
    if (node == nullptr) return;
    if (isLeaf(node)) {
        result.push_back(node->val);
        return;
    }
    addLeaves(node->left, result);
    addLeaves(node->right, result);
}

void addRightBoundary(TreeNode* node, vector<int>& result) {
    vector<int> temp;
    while (node) {
        if (!isLeaf(node)) temp.push_back(node->val);
        node = node->right ? node->right : node->left;
    }
    result.insert(result.end(), temp.rbegin(), temp.rend());
}
```

---

11. Tree Construction & Transformation Patterns

Pattern 11.1: Construct from Preorder and Inorder

Given: Preorder and inorder traversals
Goal: Reconstruct unique binary tree

Key Insights:

1. Preorder first element = root
2. Find root in inorder → splits left/right subtrees
3. Recursively construct subtrees

```
class Solution {
private:
    unordered_map<int, int> inorderMap;  // value -> index
    int preIndex = 0;

    TreeNode* build(vector<int>& preorder, int inStart, int inEnd) {
        if (inStart > inEnd) return nullptr;

        // Root is current element in preorder
```

```
        int rootVal = preorder[preIndex++];
        TreeNode* root = new TreeNode(rootVal);

        // Find root position in inorder
        int inIndex = inorderMap[rootVal];

        // Build left and right subtrees
        root->left = build(preorder, inStart, inIndex - 1);
        root->right = build(preorder, inIndex + 1, inEnd);

        return root;
    }

public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        // Build map for O(1) lookup
        for (int i = 0; i < inorder.size(); i++) {
            inorderMap[inorder[i]] = i;
        }

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

Time Complexity: O(n)
Space Complexity: O(n) for map + O(h) recursion

Why Unique?:

- Preorder determines root sequence
- Inorder determines left/right split
- Together, they uniquely identify structure

Pattern 11.2: Construct from Inorder and Postorder

```
class Solution {
private:
    unordered_map<int, int> inorderMap;
    int postIndex;

    TreeNode* build(vector<int>& postorder, int inStart, int inEnd) {
        if (inStart > inEnd) return nullptr;

        // Root is current element from end of postorder
        int rootVal = postorder[postIndex--];
        TreeNode* root = new TreeNode(rootVal);

        int inIndex = inorderMap[rootVal];

        // Build RIGHT first (postorder processes right before left)
        root->right = build(postorder, inIndex + 1, inEnd);
```

```
        root->left = build(postorder, inStart, inIndex - 1);

        return root;
    }

public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        postIndex = postorder.size() - 1;

        for (int i = 0; i < inorder.size(); i++) {
            inorderMap[inorder[i]] = i;
        }

        return build(postorder, 0, inorder.size() - 1);
    }
};
```

Critical Difference: Process right subtree before left (postorder property)

Pattern 11.3: Construct BST from Preorder

Key Insight: For BST, only preorder needed (BST property determines structure)

```
TreeNode* bstFromPreorder(vector<int>& preorder) {
    int index = 0;
    return build(preorder, index, INT_MIN, INT_MAX);
}

TreeNode* build(vector<int>& preorder, int& index, int minVal, int maxVal) {
    if (index >= preorder.size()) return nullptr;

    int val = preorder[index];

    // Check if current value fits in allowed range
    if (val < minVal || val > maxVal) return nullptr;

    index++;
    TreeNode* root = new TreeNode(val);

    // Left subtree: values < root
    root->left = build(preorder, index, minVal, val);

    // Right subtree: values > root
    root->right = build(preorder, index, val, maxVal);

    return root;
}
```

Time Complexity: O(n)
Space Complexity: O(h)

Pattern 11.4: Construct from Level Order

```
TreeNode* buildFromLevelOrder(vector<int>& levelorder) {
    if (levelorder.empty()) return nullptr;

    TreeNode* root = new TreeNode(levelorder[0]);
    queue<TreeNode*> q;
    q.push(root);

    int i = 1;
    while (!q.empty() && i < levelorder.size()) {
        TreeNode* node = q.front();
        q.pop();

        // Add left child
        if (i < levelorder.size()) {
            node->left = new TreeNode(levelorder[i++]);
            q.push(node->left);
        }

        // Add right child
        if (i < levelorder.size()) {
            node->right = new TreeNode(levelorder[i++]);
            q.push(node->right);
        }
    }

    return root;
}
```

Impossibility Results

Theorem: Cannot uniquely reconstruct binary tree from:

1. Preorder + postorder only
2. Single traversal only

Counterexample (Preorder + Postorder):

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

Preorder: [1, 2]           [1, 2]
Postorder: [2, 1]          [2, 1]
Same traversals, different trees!
```

Why Inorder Helps: Distinguishes left from right subtree

---

12. Problem Pattern Classification

Category 1: Structure Verification

Pattern: Check if tree satisfies specific structural property

Problems:

12.1 Same Tree (LC 100)

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

12.2 Subtree of Another Tree (LC 572)

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

12.3 Complete Binary Tree Check

```
bool isCompleteTree(TreeNode* root) {
    queue<TreeNode*> q;
    q.push(root);
    bool seenNull = false;

    while (!q.empty()) {
        TreeNode* node = q.front();
        q.pop();

        if (node == nullptr) {
            seenNull = true;
        } else {
            if (seenNull) return false;  // Node after null
            q.push(node->left);
            q.push(node->right);
        }
    }
```

```
        return true;
    }
```

Category 2: Path Problems

Pattern: Find/count paths satisfying conditions

Problems:

12.4 Binary Tree Maximum Path Sum (LC 124) - Covered in DP section

12.5 Path Sum III (LC 437) - Count paths with sum

```
class Solution {
private:
    int count = 0;
    unordered_map<long long, int> prefixSum;

    void dfs(TreeNode* root, long long currSum, int targetSum) {
        if (root == nullptr) return;

        currSum += root->val;

        // Count paths ending at current node
        count += prefixSum[currSum - targetSum];

        // Add current sum to map
        prefixSum[currSum]++;

        // Recurse
        dfs(root->left, currSum, targetSum);
        dfs(root->right, currSum, targetSum);

        // Backtrack
        prefixSum[currSum]--;
    }

public:
    int pathSum(TreeNode* root, int targetSum) {
        prefixSum[0] = 1;  // Base case
        dfs(root, 0, targetSum);
        return count;
    }
};
```

12.6 Longest Zigzag Path (LC 1372)

```
class Solution {
private:
    int maxLength = 0;
```

```
    pair<int, int> dfs(TreeNode* root) {
        // Returns {leftZigzag, rightZigzag}
        if (root == nullptr) return {-1, -1};

        auto [leftL, leftR] = dfs(root->left);
        auto [rightL, rightR] = dfs(root->right);

        int currLeft = leftR + 1;   // Zigzag from right child of left
        int currRight = rightL + 1;  // Zigzag from left child of right

        maxLength = max({maxLength, currLeft, currRight});

        return {currLeft, currRight};
    }

public:
    int longestZigZag(TreeNode* root) {
        dfs(root);
        return maxLength;
    }
};
```

Category 3: Traversal Variants

12.7 Zigzag Level Order (LC 103)

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

    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();

            // Find position based on direction
            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);
        }
```

```
            leftToRight = !leftToRight;
            result.push_back(level);
        }

        return result;
    }
```

Category 4: Modification Problems

12.8 Invert Binary Tree (LC 226)

```
TreeNode* invertTree(TreeNode* root) {
    if (root == nullptr) return nullptr;

    // Swap children
    TreeNode* temp = root->left;
    root->left = invertTree(root->right);
    root->right = invertTree(temp);

    return root;
}
```

12.9 Trim BST (LC 669)

```
TreeNode* trimBST(TreeNode* root, int low, int high) {
    if (root == nullptr) return nullptr;

    if (root->val < low) {
        return trimBST(root->right, low, high);
    }
    if (root->val > high) {
        return trimBST(root->left, low, high);
    }

    root->left = trimBST(root->left, low, high);
    root->right = trimBST(root->right, low, high);

    return root;
}
```

Category 5: BST-Specific

12.10 Convert Sorted Array to BST (LC 108)

```
TreeNode* sortedArrayToBST(vector<int>& nums) {
    return build(nums, 0, nums.size() - 1);
}
```

```
TreeNode* build(vector<int>& nums, int left, int right) {
    if (left > right) return nullptr;

    int mid = left + (right - left) / 2;
    TreeNode* root = new TreeNode(nums[mid]);

    root->left = build(nums, left, mid - 1);
    root->right = build(nums, mid + 1, right);

    return root;
}
```

12.11 Recover BST (LC 99) - Two nodes swapped

```
class Solution {
private:
    TreeNode *first = nullptr, *second = nullptr, *prev = nullptr;

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

        inorder(root->left);

        if (prev && root->val < prev->val) {
            if (first == nullptr) first = prev;
            second = root;
        }
        prev = root;

        inorder(root->right);
    }

public:
    void recoverTree(TreeNode* root) {
        inorder(root);
        swap(first->val, second->val);
    }
};
```

---

13. 70+ Curated Tree Problems (Structured Roadmap)

Beginner Level (Foundation Building)

Binary Tree Basics (10 problems):

1. Maximum Depth of Binary Tree (LC 104) ⭐ Start here
2. Minimum Depth of Binary Tree (LC 111)
3. Invert Binary Tree (LC 226) ⭐ Classic

4. Same Tree (LC 100)
5. Symmetric Tree (LC 101) ⭐ Important pattern
6. Merge Two Binary Trees (LC 617)
7. Sum of Left Leaves (LC 404)
8. Path Sum (LC 112) ⭐ Foundation for path problems
9. Average of Levels (LC 637)
10. Binary Tree Paths (LC 257)

BST Introduction (5 problems): 11. Search in BST (LC 700) ⭐ Understand BST property 12. Insert into BST (LC 701) 13. Delete Node in BST (LC 450) ⭐ Complete operation set 14. Minimum Absolute Difference in BST (LC 530) 15. Two Sum IV - BST (LC 653)

Intermediate Level

Tree Traversal Mastery (8 problems): 16. Binary Tree Preorder Traversal (LC 144) ⭐ Iterative version 17. Binary Tree Inorder Traversal (LC 94) ⭐ Iterative version 18. Binary Tree Postorder Traversal (LC 145) ⭐ Iterative version 19. Binary Tree Level Order Traversal (LC 102) ⭐ BFS foundation 20. Binary Tree Zigzag Level Order (LC 103) 21. Binary Tree Right Side View (LC 199) ⭐ Important variant 22. Vertical Order Traversal (LC 987) 23. Boundary of Binary Tree (LC 545)

Height and Balance (6 problems): 24. Balanced Binary Tree (LC 110) ⭐ Important check 25. Diameter of Binary Tree (LC 543) ⭐ Classic DP problem 26. Maximum Binary Tree (LC 654) 27. Construct String from Binary Tree (LC 606) 28. Find Bottom Left Tree Value (LC 513) 29. Count Good Nodes in Binary Tree (LC 1448)

Path Problems (7 problems): 30. Path Sum II (LC 113) ⭐ All paths 31. Path Sum III (LC 437) ⭐ Prefix sum technique 32. Binary Tree Maximum Path Sum (LC 124) ⭐ Hard but fundamental 33. Sum Root to Leaf Numbers (LC 129) 34. Longest Univalue Path (LC 687) 35. Maximum Difference Between Node and Ancestor (LC 1026) 36. Pseudo-Palindromic Paths (LC 1457)

BST Operations (7 problems): 37. Validate Binary Search Tree (LC 98) ⭐ Must know 38. Kth Smallest Element in BST (LC 230) ⭐ Inorder application 39. Lowest Common Ancestor of BST (LC 235) ⭐ BST optimization 40. Convert Sorted Array to BST (LC 108) ⭐ Construction 41. Convert Sorted List to BST (LC 109) 42. Trim a BST (LC 669) 43. Balance a BST (LC 1382)

Advanced Level

Tree Construction (6 problems): 44. Construct Binary Tree from Preorder and Inorder (LC 105) ⭐ Essential 45. Construct Binary Tree from Inorder and Postorder (LC 106) ⭐ Essential 46. Construct BST from Preorder (LC 1008) 47. Maximum Binary Tree II (LC 998) 48. All Possible Full Binary Trees (LC 894) 49. Recover Binary Search Tree (LC 99)

Tree DP (7 problems): 50. House Robber III (LC 337) ⭐ Classic DP on tree 51. Binary Tree Cameras (LC 968) ⭐ Greedy + DP 52. Longest Zigzag Path (LC 1372) 53. Count Nodes With Highest Score (LC 2049) 54. Maximum Product of Splitted Binary Tree (LC 1339) 55. Number of Nodes in Sub-Tree With Same Label (LC 1519) 56. Create Binary Tree from Descriptions (LC 2196)

Complex Operations (6 problems): 57. Serialize and Deserialize Binary Tree (LC 297) ⭐ Essential skill 58. Flatten Binary Tree to Linked List (LC 114) ⭐ In-place modification 59. Populating Next Right Pointers (LC 116)

★ Level linking 60. Populating Next Right Pointers II (LC 117) 61. Binary Tree to DLL (Custom) 62. Clone Binary Tree with Random Pointer (Custom)

LCA and Relationships (5 problems): 63. Lowest Common Ancestor (LC 236) ★ Fundamental algorithm 64. LCA of Deepest Leaves (LC 1123) 65. Distance Between Nodes (Custom) 66. Kth Ancestor of Tree Node (LC 1483) 67. Find Distance in Binary Tree (LC 1740)

Hard Challenges

Advanced BST (5 problems): 68. Count of Smaller Numbers After Self (LC 315) - BST application 69. Contains Duplicate III (LC 220) - BST window 70. Closest BST Value II (LC 272) - k closest 71. Largest BST Subtree (LC 333) 72. Maximum Sum BST in Binary Tree (LC 1373)

Hard Patterns (8 problems): 73. Binary Tree Coloring Game (LC 1145) 74. Distribute Coins in Binary Tree (LC 979) 75. Sum of Distances in Tree (LC 834) ★ Advanced DP 76. All Nodes Distance K (LC 863) ★ Graph conversion 77. Step-By-Step Directions (LC 2096) 78. Minimum Number of Days to Disconnect Island (LC 1568) 79. Vertical Order Traversal II (Custom) 80. Morris Traversal Implementation (Custom)

---

14. Optimization Techniques & Invariants

Technique 1: Early Termination

Principle: Stop recursion as soon as answer is known

```
// Bad: Always explores entire tree
int countNodes(TreeNode* root) {
    if (root == nullptr) return 0;
    return 1 + countNodes(root->left) + countNodes(root->right);
}

// Good: For complete binary tree, use mathematical property
int countNodes(TreeNode* root) {
    if (root == nullptr) return 0;

    int leftHeight = getLeftHeight(root);
    int rightHeight = getRightHeight(root);

    if (leftHeight == rightHeight) {
        // Perfect tree
        return (1 << leftHeight) - 1;  // 2^h - 1
    }

    return 1 + countNodes(root->left) + countNodes(root->right);
}
```

Speedup: $O(\log^2 n)$ instead of $O(n)$ for complete binary trees

Technique 2: Reference vs Value Passing

```
// Inefficient: Copies path every time
void findPaths(TreeNode* root, vector<vector<int>>& result,
               vector<int> path, int sum) {
    // path is copied here
    path.push_back(root->val);  // O(n) copy
    // ...
}

// Efficient: Pass by reference + backtrack
void findPaths(TreeNode* root, vector<vector<int>>& result,
               vector<int>& path, int sum) {
    path.push_back(root->val);  // O(1)
    // ... process
    path.pop_back();  // Backtrack
}
```

Speedup: $O(n^2)$ → O(n) space, faster overall

Technique 3: Tail Recursion Optimization

```
// Not tail-recursive
int sum(TreeNode* root) {
    if (root == nullptr) return 0;
    return root->val + sum(root->left) + sum(root->right);
}

// Tail-recursive (convertible to iteration)
int sumHelper(TreeNode* root, int acc) {
    if (root == nullptr) return acc;
    // Not truly tail-recursive for trees (multiple recursive calls)
    // But can use accumulator pattern for linear structures
}

// Iterative version (stack depth control)
int sumIterative(TreeNode* root) {
    if (root == nullptr) return 0;

    int total = 0;
    stack<TreeNode*> st;
    st.push(root);

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

        total += node->val;

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

```
    return total;
}
```

Benefit: Explicit stack control, no stack overflow risk

Technique 4: Memoization for Repeated Subproblems

```
// Problem: Count universal subtrees (all nodes have same value)
class Solution {
private:
    unordered_map<TreeNode*, int> memo;
    int count = 0;

public:
    int countUnivalSubtrees(TreeNode* root) {
        isUnival(root);
        return count;
    }

    pair<bool, int> isUnival(TreeNode* root) {
        if (root == nullptr) return {true, 0};

        auto [leftUnival, leftVal] = isUnival(root->left);
        auto [rightUnival, rightVal] = isUnival(root->right);

        bool isUnivalTree = leftUnival && rightUnival &&
                            (root->left == nullptr || root->val == leftVal) &&
                            (root->right == nullptr || root->val == rightVal);

        if (isUnivalTree) count++;

        return {isUnivalTree, root->val};
    }
};
```

Technique 5: Space Optimization - Morris Traversal

Already covered in Advanced Algorithms section. Key benefit: O(n) time, O(1) space.

Technique 6: Pruning with Global Variables

```
// Find mode in BST - use global state to avoid unnecessary work
class Solution {
private:
    int maxCount = 0;
    int currentCount = 0;
    int currentVal;
    TreeNode* prev = nullptr;
    vector<int> modes;
```

```
public:
    vector<int> findMode(TreeNode* root) {
        inorder(root);
        return modes;
    }

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

        inorder(root->left);

        // Process current node
        if (prev == nullptr || root->val != prev->val) {
            currentCount = 1;
        } else {
            currentCount++;
        }

        if (currentCount > maxCount) {
            maxCount = currentCount;
            modes.clear();
            modes.push_back(root->val);
        } else if (currentCount == maxCount) {
            modes.push_back(root->val);
        }

        prev = root;

        inorder(root->right);
    }
};
```

Pattern: Single pass with global state tracking

Technique 7: Level-wise Processing Pattern

```
// Calculate average at each level efficiently
vector<double> averageOfLevels(TreeNode* root) {
    vector<double> result;
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        double levelSum = 0;

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

```
            levelSum += node->val;

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

        result.push_back(levelSum / levelSize);
    }

    return result;
}
```

Key: Capture level size before loop to process level atomically

Invariants to Remember

Invariant 1: Tree Size

```
size(tree) = 1 + size(left) + size(right)
```

Invariant 2: Tree Height

```
height(tree) = 1 + max(height(left), height(right))
```

Invariant 3: BST Property

```
∀ node: left.val < node.val < right.val (strict ordering)
```

Invariant 4: AVL Balance Factor

```
|height(left) - height(right)| ≤ 1
```

Invariant 5: Heap Property

```
Min-Heap: parent.val ≤ children.val
Max-Heap: parent.val ≥ children.val
```

Invariant 6: Complete Tree Node Count

```
For complete tree of height h:
2^h ≤ nodes ≤ 2^(h+1) - 1
```

Invariant 7: Path Count

```
Number of root-to-leaf paths = number of leaves
```

Invariant 8: Degree Sum (Handshaking Lemma for Trees)

```
Σ degree(v) = 2 × edges = 2(n - 1)
```

---

15. Common Pitfalls & Debugging Strategies

Pitfall 1: Null Pointer Handling

Problem: Forgetting null checks

```
// WRONG: Null pointer dereference
int height(TreeNode* root) {
    return 1 + max(height(root->left), height(root->right));
    // Crashes when root is null!
}

// CORRECT: Always check null first
int height(TreeNode* root) {
    if (root == nullptr) return -1;  // or 0, depending on convention
    return 1 + max(height(root->left), height(root->right));
}
```

Debugging Strategy:

- Add null check at start of every function
- Use assertions: `assert(root != nullptr);` where applicable

Pitfall 2: Global Variables Without Reset

Problem: Using global variables across multiple test cases

```
// WRONG: maxSum persists between test cases
class Solution {
    int maxSum = INT_MIN;

public:
    int maxPathSum(TreeNode* root) {
        helper(root);
        return maxSum;  // Bug: maxSum not reset!
```

```
    }
};

// CORRECT: Reset in public function
class Solution {
    int maxSum;

public:
    int maxPathSum(TreeNode* root) {
        maxSum = INT_MIN;  // Reset here
        helper(root);
        return maxSum;
    }
};
```

Debugging Strategy: Always initialize/reset class members in public methods

Pitfall 3: Incorrect Base Cases

Problem: Wrong base case definition

```
// WRONG: Doesn't handle single-node tree correctly
int diameter(TreeNode* root) {
    if (root == nullptr) return 0;

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

    // For single node: left=0, right=0, should return 0
    // But this returns 2!
    return 2 + left + right;  // WRONG
}

// CORRECT: Count edges, not nodes
int diameter(TreeNode* root) {
    if (root == nullptr) return 0;

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

    return left + right;  // Count edges
}
```

Test Case: Always test with single-node tree: `root = TreeNode(1)`

Pitfall 4: Height vs Depth Confusion

Height: Edges from node to deepest leaf (bottom-up)
Depth: Edges from root to node (top-down)

```
// For node at depth d in tree of height h:
// depth(node) + height(node) ≤ h

// Don't confuse:
int depth(TreeNode* root) {
    // Measures from root DOWN
}

int height(TreeNode* root) {
    // Measures from leaves UP
}
```

Pitfall 5: Modifying Tree Structure During Traversal

Problem: Changing pointers while iterating

```
// DANGEROUS: Modifying tree during recursion
void dangerousModify(TreeNode* root) {
    if (root == nullptr) return;

    dangerousModify(root->left);

    // This invalidates root->right for next call!
    root->right = root->left;

    dangerousModify(root->right);  // Might revisit same subtree
}

// SAFE: Save pointers first
void safeModify(TreeNode* root) {
    if (root == nullptr) return;

    TreeNode* leftSubtree = root->left;
    TreeNode* rightSubtree = root->right;

    safeModify(leftSubtree);

    // Now safe to modify
    root->right = root->left;

    safeModify(rightSubtree);
}
```

Pitfall 6: Integer Overflow in Sums

```
// WRONG: Can overflow with large values
int sumTree(TreeNode* root) {
    if (root == nullptr) return 0;
```

```
    return root->val + sumTree(root->left) + sumTree(root->right);
}

// BETTER: Use long long
long long sumTree(TreeNode* root) {
    if (root == nullptr) return 0LL;
    return (long long)root->val + sumTree(root->left) + sumTree(root->right);
}
```

Pitfall 7: Not Considering Edge Cases

Critical Edge Cases for Trees:

1. Empty tree (`root == nullptr`)
2. Single node tree
3. Skewed tree (all left or all right)
4. Perfect binary tree
5. Tree with negative values
6. Tree with duplicate values

```
// Always test these cases:
TreeNode* empty = nullptr;
TreeNode* single = new TreeNode(1);
TreeNode* leftSkewed = buildLeftSkewed(5);
TreeNode* perfect = buildPerfect(3);
```

Pitfall 8: BST Validation - Only Checking Immediate Children

Already covered in BST section. Always use range-based validation!

Pitfall 9: Memory Leaks in Tree Construction

```
// WRONG: Memory leak if exception thrown
TreeNode* buildTree() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);

    // If this throws, root and root->left are leaked!
    root->right = riskyOperation();

    return root;
}

// BETTER: Use smart pointers or RAII
unique_ptr<TreeNode> buildTreeSafe() {
    auto root = make_unique<TreeNode>(1);
    root->left = make_unique<TreeNode>(2);
    root->right = riskyOperation();
```

```
    return root;
}
```

Pitfall 10: Incorrect Comparison in BST

```
// WRONG: Uses >= instead of >
bool isValidBST(TreeNode* root, long min, long max) {
    if (root == nullptr) return true;

    // Should be > min and < max (strict inequalities)
    if (root->val >= max || root->val <= min) return false;
    //              ^^                 ^^
    // This allows duplicates, violating BST property

    return isValidBST(root->left, min, root->val) &&
           isValidBST(root->right, root->val, max);
}
```

Debugging Strategies

Strategy 1: Visualize the Tree

```
void printTree(TreeNode* root, int space = 0) {
    if (root == nullptr) return;

    space += 5;
    printTree(root->right, space);

    cout << endl;
    for (int i = 5; i < space; i++) cout << " ";
    cout << root->val << endl;

    printTree(root->left, space);
}
```

Strategy 2: Add Trace Statements

```
int helper(TreeNode* root, int depth = 0) {
    string indent(depth * 2, ' ');
    cout << indent << "Visiting: " << (root ? to_string(root->val) : "null") <<
endl;

    if (root == nullptr) return 0;

    int result = 1 + helper(root->left, depth + 1) + helper(root->right, depth +
1);
```

```
    cout << indent << "Returning: " << result << endl;
    return result;
}
```

Strategy 3: Verify Invariants

```
bool verifyBST(TreeNode* root, long min = LONG_MIN, long max = LONG_MAX) {
    if (root == nullptr) return true;

    // Check invariant
    assert(root->val > min && root->val < max);

    return verifyBST(root->left, min, root->val) &&
           verifyBST(root->right, root->val, max);
}
```

Strategy 4: Unit Test Small Cases

```
void testHeight() {
    // Test 1: Empty tree
    assert(height(nullptr) == -1);

    // Test 2: Single node
    TreeNode* single = new TreeNode(1);
    assert(height(single) == 0);

    // Test 3: Two levels
    single->left = new TreeNode(2);
    assert(height(single) == 1);

    // Test 4: Unbalanced
    single->left->left = new TreeNode(3);
    assert(height(single) == 2);
}
```

---

16. Universal Tree Code Templates

Template 1: Binary Tree Node Definition

```
// Standard definition
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode() : val(0), left(nullptr), right(nullptr) {}
```

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

// With parent pointer
struct TreeNodeWithParent {
    int val;
    TreeNodeWithParent* left;
    TreeNodeWithParent* right;
    TreeNodeWithParent* parent;

    TreeNodeWithParent(int x)
        : val(x), left(nullptr), right(nullptr), parent(nullptr) {}
};

// N-ary tree node
struct Node {
    int val;
    vector<Node*> children;

    Node() {}
    Node(int _val) : val(_val) {}
    Node(int _val, vector<Node*> _children) : val(_val), children(_children) {}
};
```

Template 2: DFS Traversal (All Types)

```
// Preorder (NLR)
void preorder(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->val << " ";      // Process
    preorder(root->left);          // Left
    preorder(root->right);         // Right
}

// Inorder (LNR)
void inorder(TreeNode* root) {
    if (root == nullptr) return;
    inorder(root->left);           // Left
    cout << root->val << " ";      // Process
    inorder(root->right);          // Right
}

// Postorder (LRN)
void postorder(TreeNode* root) {
    if (root == nullptr) return;
    postorder(root->left);         // Left
    postorder(root->right);        // Right
    cout << root->val << " ";      // Process
}
```

```
// Generic DFS template with callback
void dfs(TreeNode* root, function<void(TreeNode*)> visit) {
    if (root == nullptr) return;
    visit(root);
    dfs(root->left, visit);
    dfs(root->right, visit);
}
```

Template 3: BFS Level-Order Traversal

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

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

    while (!q.empty()) {
        int levelSize = q.size();  // Capture size for current level
        vector<int> 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;
}
```

Template 4: Tree DP (Bottom-Up Recursion)

```
// Generic tree DP template
struct DPState {
    // Define state variables
    int value1;
    int value2;
    bool flag;
    // ... additional state
};
```

```
DPState treeDFS(TreeNode* root) {
    // Base case
    if (root == nullptr) {
        return DPState{/* base values */};
    }

    // Recurse on children
    DPState leftState = treeDFS(root->left);
    DPState rightState = treeDFS(root->right);

    // Combine results
    DPState currentState;
    // ... combine logic using leftState, rightState, and root

    return currentState;
}

// Example: Max path sum with state
pair<int, int> maxPathSumDP(TreeNode* root) {
    // Returns: {maxPathEndingHere, globalMax}
    if (root == nullptr) return {0, INT_MIN};

    auto [leftPath, leftMax] = maxPathSumDP(root->left);
    auto [rightPath, rightMax] = maxPathSumDP(root->right);

    int pathEndingHere = root->val + max({0, leftPath, rightPath});
    int pathThroughHere = root->val + max(0, leftPath) + max(0, rightPath);
    int globalMax = max({leftMax, rightMax, pathThroughHere});

    return {pathEndingHere, globalMax};
}
```

Template 5: Backtracking on Trees

```
void backtrack(TreeNode* root,
               vector<int>& path,
               vector<vector<int>>& result) {
    if (root == nullptr) return;

    // Make choice
    path.push_back(root->val);

    // Check if goal state
    if (isGoal(root)) {
        result.push_back(path);
    }

    // Recurse
    backtrack(root->left, path, result);
    backtrack(root->right, path, result);
```

```
    // Undo choice (backtrack)
    path.pop_back();
}
```

Template 6: LCA (Lowest Common Ancestor)

```
// Binary tree LCA
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (root == nullptr || 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;  // Split
    return left ? left : right;       // Both in one subtree
}

// BST LCA (optimized)
TreeNode* lowestCommonAncestorBST(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;  // Split point
        }
    }
    return nullptr;
}
```

Template 7: BST Operations

```
// Search
TreeNode* searchBST(TreeNode* root, int val) {
    while (root && root->val != val) {
        root = (val < root->val) ? root->left : root->right;
    }
    return root;
}

// Insert
TreeNode* insertBST(TreeNode* root, int val) {
    if (root == nullptr) return new TreeNode(val);

    if (val < root->val) {
        root->left = insertBST(root->left, val);
```

```
    } else {
        root->right = insertBST(root->right, val);
    }

    return root;
}

// Delete
TreeNode* deleteBST(TreeNode* root, int key) {
    if (root == nullptr) return nullptr;

    if (key < root->val) {
        root->left = deleteBST(root->left, key);
    } else if (key > root->val) {
        root->right = deleteBST(root->right, key);
    } else {
        // Node found
        if (root->left == nullptr) return root->right;
        if (root->right == nullptr) return root->left;

        // Two children: find inorder successor
        TreeNode* minRight = root->right;
        while (minRight->left) minRight = minRight->left;

        root->val = minRight->val;
        root->right = deleteBST(root->right, minRight->val);
    }

    return root;
}

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

bool validate(TreeNode* root, long minVal, long maxVal) {
    if (root == nullptr) return true;
    if (root->val <= minVal || root->val >= maxVal) return false;

    return validate(root->left, minVal, root->val) &&
           validate(root->right, root->val, maxVal);
}
```

Template 8: Tree Construction

```
// From preorder and inorder
class Solution {
private:
    unordered_map<int, int> inorderMap;
    int preIndex = 0;
```

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

    TreeNode* build(vector<int>& preorder, int inStart, int inEnd) {
        if (inStart > inEnd) return nullptr;

        int rootVal = preorder[preIndex++];
        TreeNode* root = new TreeNode(rootVal);

        int inIndex = inorderMap[rootVal];

        root->left = build(preorder, inStart, inIndex - 1);
        root->right = build(preorder, inIndex + 1, inEnd);

        return root;
    }
};
```

Template 9: Serialization

```
// Serialize
string serialize(TreeNode* root) {
    if (root == nullptr) return "#";
    return to_string(root->val) + "," +
           serialize(root->left) + "," +
           serialize(root->right);
}

// Deserialize
TreeNode* deserialize(string data) {
    queue<string> nodes;
    stringstream ss(data);
    string item;

    while (getline(ss, item, ',')) {
        nodes.push(item);
    }

    return deserializeHelper(nodes);
}

TreeNode* deserializeHelper(queue<string>& nodes) {
    string val = nodes.front();
    nodes.pop();
```

```
    if (val == "#") return nullptr;

    TreeNode* root = new TreeNode(stoi(val));
    root->left = deserializeHelper(nodes);
    root->right = deserializeHelper(nodes);

    return root;
}
```

Template 10: Iterative DFS with Stack

```
// Preorder iterative
vector<int> preorderIterative(TreeNode* root) {
    vector<int> result;
    if (root == nullptr) return result;

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

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

        result.push_back(node->val);

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

    return result;
}

// Inorder iterative
vector<int> inorderIterative(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> st;
    TreeNode* curr = root;

    while (curr || !st.empty()) {
        while (curr) {
            st.push(curr);
            curr = curr->left;
        }

        curr = st.top();
        st.pop();
        result.push_back(curr->val);

        curr = curr->right;
    }
```

```
        return result;
    }
```

Conclusion

This guide provides comprehensive coverage of tree data structures, from foundational theory to advanced algorithms. Key takeaways:

Fundamental Principles:

- Trees are acyclic connected graphs with n-1 edges
- Recursive structure enables divide-and-conquer algorithms
- Understanding traversals is crucial for all tree problems

Critical Skills:

- Master all four traversals (preorder, inorder, postorder, level-order)
- Understand BST property and its implications
- Practice tree DP for complex optimization problems
- Know when to use recursion vs iteration

Problem-Solving Framework:

1. Identify the pattern (traversal, DP, construction, etc.)
2. Choose appropriate traversal or algorithm
3. Handle base cases carefully
4. Optimize with early termination and pruning
5. Test edge cases thoroughly

Interview Preparation:

- Start with beginner problems to build intuition
- Progress through intermediate patterns (paths, LCA, validation)
- Master advanced topics (Morris, serialization, tree DP)
- Practice hard problems for confidence

Time Complexity Summary:

- Traversals: O(n)
- BST operations: O(h) where h = height
- Balanced tree operations: O(log n) guaranteed
- Tree DP: O(n) typically
- Construction from traversals: O(n)

Space Complexity Considerations:

- Recursive: O(h) for call stack
- Iterative: O(h) for explicit stack (DFS) or O(w) for queue (BFS)
- Morris: O(1) space traversal

This guide serves as a comprehensive reference for tree DSA mastery, suitable for interviews, competitive programming, and academic study. Regular practice with the curated 70+ problems will solidify understanding and build problem-solving speed.

End of Complete Tree DSA – Deep Mastery Guide