

Complete Tree DSA — Deep Mastery Guide

Table of Contents

- Complete Tree DSA — Deep Mastery Guide
 - Table of Contents
 - Theoretical Foundation
 - What is a Tree?
 - Mathematical Analysis
 - Fundamental Properties
 - Important Theorems
 - Complexity Framework
 - Tree Classifications & Taxonomy
 - Tree Representations
 - Binary Tree Node
 - N-ary Node
 - Implicit / Array Representation
 - Core Traversal Patterns
 - DFS Traversals (Inorder / Preorder / Postorder)
 - BFS (Level Order)
 - Advanced Algorithms & Patterns
 - Height & Diameter (single pass)
 - Lowest Common Ancestor (LCA)
 - BST Patterns
 - Problem Categories
 - 60+ Curated Problems
 - Optimization Techniques
 - Study Notes & Mental Models
 - Practice Roadmap
 - Common Pitfalls & Solutions
 - Code Templates
 - Final Note

Theoretical Foundation

What is a Tree?

Definition: A tree is a connected, acyclic graph.

- Connected: a path exists between any two nodes
- Acyclic: no cycles
- For N nodes \rightarrow exactly $N - 1$ edges

Why Trees Matter

Trees model hierarchical data and fast search structures: file systems, DOM, B-trees, syntax trees, routing tables, tries, segment trees, and many more.

Historical Notes

- Euler (1736) \rightarrow foundations of graph theory
- Cayley (1857) \rightarrow counting trees
- Knuth (1968) \rightarrow algorithmic treatment of trees

Mathematical Analysis

Fundamental Properties

Property	Formula / Rule
Nodes	N
Edges	$N - 1$
Height	Longest root \rightarrow leaf path
Depth	Distance from root
Degree	Number of children
Leaves	Nodes with degree 0

Important Theorems

Unique path theorem: exactly one simple path between any two nodes.

Edge count theorem: connected graph with N nodes and $N-1$ edges is a tree.

Complexity Framework

Operation	Time
Traversal / Visit all nodes	$O(N)$
Search (unsorted)	$O(N)$
Insert / Delete (balanced)	$O(\log N)$
Insert / Delete (skewed)	$O(N)$

H = tree height. Balanced trees: $H = O(\log N)$. Skewed: $H = O(N)$.

Tree Classifications & Taxonomy

TREES

- |—— By Structure
 - |—— General Tree
 - |—— Binary Tree
 - |—— Full
 - |—— Complete
 - |—— Perfect
 - |—— Skewed
- |—— By Order
 - |—— Binary Search Tree (BST)
 - |—— AVL Tree
 - |—— Red-Black Tree
- |—— Special Trees
 - |—— Heap
 - |—— Trie
 - |—— Segment Tree

```
| └── Fenwick Tree  
| └── N-ary Tree
```

Tree Representations

Binary Tree Node

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

N-ary Node

```
struct Node {  
    int val;  
    vector<Node*> children;  
};
```

Implicit / Array Representation

Used for heaps and almost-complete binary trees.

- Parent(i) = $(i - 1) / 2$
- Left(i) = $2*i + 1$
- Right(i) = $2*i + 2$

Core Traversal Patterns

DFS Traversals (Inorder / Preorder / Postorder)

Inorder (LNR): useful for BSTs and sorted-order traversal.

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

Preorder (NLR): used for serialization & constructing trees.

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

Postorder (LRN): bottom-up DP and deletion.

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

BFS (Level Order)

```
vector<vector<int>> levelOrder(TreeNode* root) {  
    vector<vector<int>> res;  
    if (!root) return res;  
    queue<TreeNode*> q; q.push(root);  
    while (!q.empty()) {  
        int sz = q.size();  
        vector<int> level;  
        for (int i = 0; i < sz; ++i) {  
            auto node = q.front(); q.pop();  
            level.push_back(node->val);  
            if (node->left) q.push(node->left);  
            if (node->right) q.push(node->right);  
        }  
        res.push_back(level);  
    }  
    return res;  
}
```

Advanced Algorithms & Patterns

Height & Diameter (single pass)

```
int diameter = 0;  
int dfsHeight(TreeNode* root) {  
    if (!root) return 0;  
    int L = dfsHeight(root->left);  
    int R = dfsHeight(root->right);  
    diameter = max(diameter, L + R);  
    return 1 + max(L, R);  
}
```

Lowest Common Ancestor (LCA)

```
TreeNode* LCA(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (!root || root == p || root == q) return root;
    TreeNode* left = LCA(root->left, p, q);
    TreeNode* right = LCA(root->right, p, q);
    if (left && right) return root;
    return left ? left : right;
}
```

BST Patterns

Search

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

Validate BST (range check)

```
bool isValidBST(TreeNode* root, long lo = LONG_MIN, long hi = LONG_MAX) {
    if (!root) return true;
    if (root->val <= lo || root->val >= hi) return false;
    return isValidBST(root->left, lo, root->val) &&
           isValidBST(root->right, root->val, hi);
}
```

Problem Categories

- Traversal: Inorder / Preorder / Postorder / Level Order / Zigzag
- Structural: Height / Diameter / Balanced / Symmetric
- Paths: Root-to-leaf sums / Max path sum / All paths
- BST: Insert / Delete / Kth smallest / LCA in BST
- Advanced: Trie / Segment Tree / Fenwick Tree / Heap

60+ Curated Problems

Beginner

- [Inorder Traversal \(LC 94\)](#)
- [Preorder Traversal \(LC 144\)](#)
- [Postorder Traversal \(LC 145\)](#)
- [Maximum Depth \(LC 104\)](#)
- [Same Tree \(LC 100\)](#)

Intermediate

- [Diameter of Binary Tree \(LC 543\)](#)
- [Balanced Binary Tree \(LC 110\)](#)
- [Lowest Common Ancestor \(LC 236\)](#)
- [Zigzag Level Order \(LC 103\)](#)
- [Validate Binary Search Tree \(LC 98\)](#)

Advanced

- [Binary Tree Maximum Path Sum \(LC 124\)](#)
- [Serialize and Deserialize Binary Tree \(LC 297\)](#)
- [Vertical Order Traversal \(LC 987\)](#)
- [Recover Binary Search Tree \(LC 99\)](#)
- [Construct Binary Tree from Preorder and Inorder \(LC 105\)](#)

Optimization Techniques

1. Single-pass DFS: compute multiple values (height + diameter) in one traversal.
2. Bottom-up DP: use postorder to aggregate child results.
3. Bitmasking for state-space compression (e.g., keys collection problems).

Study Notes & Mental Models

- Tree problems often reduce to: $\text{Answer}(\text{node}) = f(\text{left}, \text{right})$
- Decide top-down vs bottom-up: top-down for path tracking, bottom-up for DP aggregations.

Practice Roadmap

Stage	Focus
Week 1	Traversals
Week 2	Height, Diameter, Balance
Week 3	BST operations
Week 4	Path & DP problems
Week 5	Advanced trees (Trie, SegTree)

Common Pitfalls & Solutions

Pitfall	Fix
Recomputing height repeatedly	Cache results (return heights)
Wrong base case / NULL handling	Always check for nullptr at start
Using wrong traversal order for DP	Use postorder for bottom-up aggregation

Code Templates

Universal DFS template (bottom-up)

```

int dfs(TreeNode* root) {
    if (!root) return 0;
    int L = dfs(root->left);
    int R = dfs(root->right);
    return process(L, R, root);
}

```

Iterative inorder (stack)

```

vector<int> inorderIter(TreeNode* root) {
    vector<int> res; stack<TreeNode*> st; TreeNode* cur = root;
    while (cur || !st.empty()) {
        while (cur) { st.push(cur); cur = cur->left; }
        cur = st.top(); st.pop(); res.push_back(cur->val);
        cur = cur->right;
    }
    return res;
}

```

Final Note

Trees are recursive by nature. Master recursion → master trees → master DSA.

File: [Tree/Readme.md](#)