🌳 Trees - Complete Interview Guide

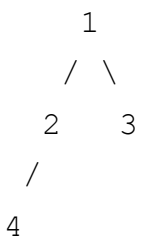
A comprehensive guide to tree data structures with **interview-focused insights** and **problem-solving tips**.

📌 Tree Types & Classifications

> [!IMPORTANT]
> **Interview Must-Know**: You will often be asked to differentiate between these tree types. Understanding their properties is essential!

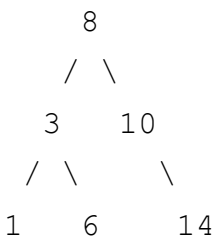
Binary Tree

A tree where each node has **at most 2 children** (left and right).



Binary Search Tree (BST) ⭐⭐⭐

A binary tree where for every node: **left subtree < node < right subtree**.



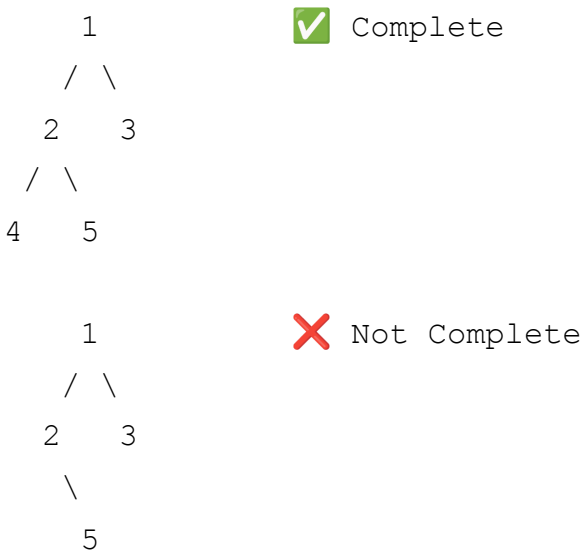
Operation	Average	Worst Case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

> [!CAUTION]
> **Interview Trap**: BST property must hold for the **entire subtree**, not just immediate children!

Complete Binary Tree ⭐⭐

A binary tree where:

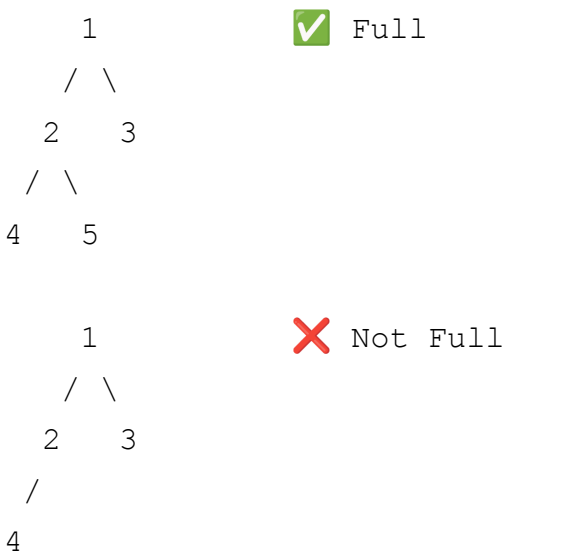
- All levels are **completely filled** except possibly the last
- The last level has nodes **as far left as possible**



> [!TIP]
> **Used in**: Binary Heaps, Priority Queues. Array representation works perfectly!

Full (Strict) Binary Tree ⭐

A binary tree where every node has **either 0 or 2 children** (no node has only 1 child).



Property	Formula
Leaf nodes	`(n + 1) / 2`
Internal nodes	`(n - 1) / 2`
Total nodes	`2L - 1` (where L = leaves)

Perfect Binary Tree ⭐

A binary tree where:

- All **internal nodes have 2 children**
- All **leaves are at the same level**

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

Property	Formula
Total nodes	`2^(h+1) - 1`
Leaf nodes	`2^h`
Height	`log₂(n+1) - 1`

> [!NOTE]
> A perfect binary tree is both **complete** AND **full**.

Balanced Binary Tree ⭐⭐⭐

A binary tree where the **height difference** between left and right subtrees of **every node** is at most 1.

```
// Check if tree is balanced - Common Interview Question!
boolean isBalanced(TreeNode root) {
    return checkHeight(root) != -1;
}

int checkHeight(TreeNode node) {
    if (node == null) return 0;
    int left = checkHeight(node.left);
```

```
        if (left == -1) return -1;
        int right = checkHeight(node.right);
        if (right == -1) return -1;
        if (Math.abs(left - right) > 1) return -1;
        return 1 + Math.max(left, right);
    }
```

> [!IMPORTANT]
> **Examples**: AVL Trees, Red-Black Trees. These guarantee O(log n) operations.

Degenerate (Skewed) Tree

A tree where each parent has **only one child**, essentially becoming a **linked list**.

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

Property	Value
Height	`n - 1` (worst case)
Search	O(n)

> [!WARNING]
> **Interview Insight**: This is the **worst case** for BST. This is why balanced trees exist!

📊 Tree Types Comparison

Type	All Levels Full?	Nodes per Parent	Special Property
Complete	Except last (left-filled)	0, 1, or 2	Heap-friendly
Full	No requirement	0 or 2 only	No single children
Perfect	Yes, all levels	2 (internal), 0 (leaf)	Complete + Full
Balanced	No requirement	Any	Height diff ≤ 1
BST	No requirement	0, 1, or 2	left < node < right

📌 Foundational Concepts

Tree

A **hierarchical data structure** consisting of nodes connected by edges, with one node designated as the **root** and no cycles present. Each node can have zero or more child nodes, forming a parent-child relationship.

> [!IMPORTANT]
> **Interview Hot Topic**: Trees are fundamental to many algorithm questions. Always clarify if the tree is **binary**, **BST**, **balanced**, or **general** before solving.

Node

A fundamental unit in a tree that contains **data** and **references** (pointers/links) to other nodes. Each node stores a value and maintains connections to its children.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
```

Root

The **topmost node** in a tree that has no parent. It serves as the starting point for traversing the tree and is the ancestor of all other nodes.

> [!TIP]
> **Problem Solving**: Most tree problems start from the root. If `root == null`, handle it as a base case returning `0`, `null`, or an empty result.

Edge

A connection or link between two nodes representing a parent-child relationship.

Property	Formula
Number of Edges	`n - 1` (where n = number of nodes)

> [!NOTE]
> **Interview Tip**: If asked about edges in a tree, remember: **edges = nodes - 1**. This is a common quick-check question.

Parent & Child

- **Parent**: A node with one or more children directly below it
- **Child**: A node directly connected to another node when moving away from the root

> [!IMPORTANT]
> In a tree, every node (except root) has **exactly one parent**.

Leaf (External Node) ⭐

A node that has **no children**. These are the terminal nodes located at the bottommost level of various branches.

```
// Check if node is a leaf
boolean isLeaf(TreeNode node) {
    return node != null && node.left == null && node.right == null;
}
```

> [!TIP]
> **Common Pattern**: Many problems require special handling at leaves (e.g., path sum, leaf-to-leaf paths).

Internal Node

Any node that has **at least one child**. The root is an internal node unless the tree contains only one node.

Siblings

Nodes that **share the same parent** node. They exist at the same level in the tree structure.

📐 Structural Concepts

Subtree ⭐

A tree structure formed by a **node and all its descendants**. Any node in a tree can be viewed as the root of its own subtree.

> [!IMPORTANT]
> **Interview Pattern**: Subtree problems often use **recursion** where each node is treated as a root of its subtree.

Ancestor

A node that lies on the **path from the root to a given node**. Every node is an ancestor of itself.

```
// Find Lowest Common Ancestor (LCA) - Classic Interview Problem
TreeNode LCA(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || root == p || root == q) return root;
    TreeNode left = LCA(root.left, p, q);
    TreeNode right = LCA(root.right, p, q);
    if (left != null && right != null) return root;
    return left != null ? left : right;
}
```

> [!CAUTION]
> **LCA (Lowest Common Ancestor)** is a very common interview question. Know this pattern by heart!

Descendant

A node that can be reached by following edges **downward** from a given node. Every node is a descendant of itself.

Diameter (Width) ⭐⭐⭐

The length of the **longest path between any two nodes** in the tree. This path may or may not pass through the root.

> [!IMPORTANT]
> **How to Calculate**:
>
> `Diameter = max(leftHeight + rightHeight) at any node`

```
int diameter = 0;

int findDiameter(TreeNode root) {
    height(root);
    return diameter;
}

int height(TreeNode node) {
    if (node == null) return 0;
    int left = height(node.left);
    int right = height(node.right);
    diameter = Math.max(diameter, left + right); // Update diameter
    return 1 + Math.max(left, right);
}
```

Key Points	Details
Time Complexity	O(n)
Space Complexity	O(h) where h = height
Common Mistake	Forgetting that diameter might not pass through root

> [!WARNING]
> **Interview Trap**: Don't assume the longest path always goes through the root!

Width of a Level

The **number of nodes** at a particular level of the tree.

Width of a Tree (Maximum Width) ⭐

The **maximum number of nodes** that exist at any single level.

> [!TIP]
> **How to Calculate**: Use **BFS (Level Order Traversal)** and track the size of each level.

```
int maxWidth(TreeNode root) {
    if (root == null) return 0;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int maxWidth = 0;
    while (!queue.isEmpty()) {
        int levelSize = queue.size();
        maxWidth = Math.max(maxWidth, levelSize);
        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();
            if (node.left != null) queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }
    }
    return maxWidth;
}
```

📏 Measurement Concepts

Path

A **sequence of nodes and edges** connecting one node to another. The path from node A to node B is **unique** in a tree structure.

> [!NOTE]
> **Interview Insight**: Path uniqueness in trees is what makes them simpler than graphs for many algorithms.

Path Length

The **number of edges** in a path.

Example	Nodes in Path	Path Length
Path with 4 nodes	A → B → C → D	**3 edges**
Path with 2 nodes	A → B	**1 edge**

> [!TIP]
> **Formula**: `Path Length = Number of Nodes - 1`

Depth (Level) of a Node ⭐⭐

The **number of edges from the root** to that specific node.

Node	Depth
Root	**0**
Root's children	**1**
Root's grandchildren	**2**

```
// Calculate depth of a node
int depth(TreeNode root, TreeNode target, int d) {
    if (root == null) return -1;
    if (root == target) return d;
    int left = depth(root.left, target, d + 1);
    if (left != -1) return left;
    return depth(root.right, target, d + 1);
}
```

> [!IMPORTANT]
> **Remember**: Depth is measured **from the root DOWN** to the node.

Height of a Node ⭐⭐⭐

The **number of edges** on the longest path from that node **down to a leaf**. Leaf nodes have a height of **0**.

```
// Calculate height of a node
int height(TreeNode node) {
    if (node == null) return -1;  // Empty tree convention
    return 1 + Math.max(height(node.left), height(node.right));
}
```

> [!IMPORTANT]
> **Remember**: Height is measured **from the node DOWN** to the deepest leaf.

Height of a Tree ⭐⭐⭐

The **height of the root node**, or equivalently, the number of edges on the longest path from the root to any leaf.

Tree Type	Height
Empty tree	**-1** (by convention)
Single node (root only)	**0**
Complete binary tree	$\lfloor \log_2(n) \rfloor$

> [!TIP]
> **Problem Solving Formula**:
>
> ```
> height(node) = 1 + max(height(left), height(right))
> Base case: height(null) = -1
> ```

📊 Height vs Depth - Quick Reference

Concept	Direction	Root Value	Leaf Value
Depth	Top → Down (from root)	0	Varies
Height	Bottom → Up (to leaves)	Height of tree	0

> [!CAUTION]
> **Common Interview Confusion**: Don't mix up height and depth!
>
> - Depth: How far from root
> - Height: How far to deepest leaf

Level ⭐⭐

Often used **interchangeably with depth**. Nodes at the same depth are said to be at the same level. The root is at level 0.

Level	Description	Max Nodes (Binary Tree)
0	Root only	$2^0 = \mathbf{1}$
1	Root's children	$2^1 = \mathbf{2}$
2	Grandchildren	$2^2 = \mathbf{4}$

Level	Description	Max Nodes (Binary Tree)
L	Level L	**2^L**

> [!IMPORTANT]
> **Key Formulas**:
>
> - Max nodes at level L: `2^L` (binary tree)
> - Total nodes in perfect tree of height H: `2^(H+1) - 1`
> - Level of node with index i (0-indexed array): $\lfloor \log_2(i+1) \rfloor$

```
// Print all nodes at a given level
void printLevel(TreeNode root, int level) {
    if (root == null) return;
    if (level == 0) {
        System.out.print(root.val + " ");
        return;
    }
    printLevel(root.left, level - 1);
    printLevel(root.right, level - 1);
}

// Count nodes at a specific level
int countNodesAtLevel(TreeNode root, int level) {
    if (root == null) return 0;
    if (level == 0) return 1;
    return countNodesAtLevel(root.left, level - 1)
         + countNodesAtLevel(root.right, level - 1);
}
```

> [!TIP]
> **Interview Pattern**: Level-based problems often use **BFS** (Queue) or **recursion with level parameter**.

Degree of a Node ⭐

The **number of children** that node has.

Node Type	Degree
Leaf	**0**
Node with one child	**1**

Node Type	Degree
Node with two children (binary)	**2**

```
// Calculate degree of a node
int degree(TreeNode node) {
    if (node == null) return 0;
    int count = 0;
    if (node.left != null) count++;
    if (node.right != null) count++;
    return count;
}

// Count leaf nodes (degree 0)
int countLeaves(TreeNode root) {
    if (root == null) return 0;
    if (root.left == null && root.right == null) return 1;
    return countLeaves(root.left) + countLeaves(root.right);
}

// Count internal nodes (degree > 0)
int countInternalNodes(TreeNode root) {
    if (root == null) return 0;
    if (root.left == null && root.right == null) return 0;
    return 1 + countInternalNodes(root.left) + countInternalNodes(root.ri
}
```

> [!TIP]
> **Interview Insight**: Leaf count and internal node count problems are common. Remember: `Leaves = Internal + 1` (for full binary tree).

Degree of a Tree ⭐

The **maximum degree** among all nodes in the tree.

Tree Type	Degree
Binary Tree	**2**
Ternary Tree	**3**
N-ary Tree	**N**

```
// Find maximum degree (for N-ary tree)
int maxDegree(TreeNode root) {
    if (root == null) return 0;
    int degree = root.children.size();  // For N-ary tree
    for (TreeNode child : root.children) {
        degree = Math.max(degree, maxDegree(child));
    }
    return degree;
}
```

> [!IMPORTANT]
> **Key Relationship**:
>
> - For a **full binary tree** with L leaves: `Total nodes = 2L - 1`
> - For a **full binary tree**: `Internal nodes = Leaves - 1`

Size ⭐

The **total number of nodes** in the tree.

```
int size(TreeNode root) {
    if (root == null) return 0;
    return 1 + size(root.left) + size(root.right);
}
```

> [!TIP]
> **Time Complexity**: O(n) - Must visit every node

🎯 Quick Interview Formulas Cheat Sheet

Concept	Formula / Calculation
Edges	`n - 1`
Height	`1 + max(left_height, right_height)`
Depth	Count edges from root to node
Diameter	`max(left_height + right_height)` at any node
Size	`1 + size(left) + size(right)`

Concept	Formula / Calculation
Path Length	`nodes_in_path - 1`
Max nodes at level L	`2^L` (binary tree)
Max nodes in tree of height H	`2^(H+1) - 1` (binary tree)
Min height for n nodes	$\lfloor \log_2(n) \rfloor$

🌳 Tree Traversals (Must-Know!) ⭐⭐⭐

> [!IMPORTANT]
> **These 4 traversals appear in 80%+ of tree interview questions!**

DFS Traversals

```
// 1. PREORDER: Root → Left → Right (Used for: copying tree, prefix expre
void preorder(TreeNode root) {
    if (root == null) return;
    System.out.print(root.val + " ");  // Process root FIRST
    preorder(root.left);
    preorder(root.right);
}

// 2. INORDER: Left → Root → Right (Used for: BST gives SORTED order!)
void inorder(TreeNode root) {
    if (root == null) return;
    inorder(root.left);
    System.out.print(root.val + " ");  // Process root in MIDDLE
    inorder(root.right);
}

// 3. POSTORDER: Left → Right → Root (Used for: deleting tree, postfix ex
void postorder(TreeNode root) {
    if (root == null) return;
    postorder(root.left);
    postorder(root.right);
    System.out.print(root.val + " ");  // Process root LAST
}
```

> [!TIP]
> **Memory Trick**:
>
> - Pre = root **PRE**cedes children
> - In = root is **IN** the middle
> - Post = root comes **POST** (after) children

BFS Traversal (Level Order) ⭐⭐

```
// 4. LEVEL ORDER: Level by level, left to right
void levelOrder(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        int levelSize = queue.size();  // KEY: capture level size!
        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();
            System.out.print(node.val + " ");
            if (node.left != null) queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }
        System.out.println();  // New line after each level
    }
}
```

> [!CAUTION]
> **Common Mistake**: Forgetting to capture `levelSize` before the loop causes bugs when you need level-by-level processing!

🎯 Classic Interview Problems

Path Sum (Root to Leaf) ⭐⭐

```
// Check if any root-to-leaf path equals target sum
boolean hasPathSum(TreeNode root, int targetSum) {
    if (root == null) return false;

    // Reached a leaf - check if remaining sum equals leaf value
```

```
    if (root.left == null && root.right == null) {
        return targetSum == root.val;
    }

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

Validate BST ⭐⭐⭐

```
// Check if tree is a valid BST
boolean isValidBST(TreeNode root) {
    return validate(root, Long.MIN_VALUE, Long.MAX_VALUE);
}

boolean validate(TreeNode node, long min, long max) {
    if (node == null) return true;
    if (node.val <= min || node.val >= max) return false;
    return validate(node.left, min, node.val) &&
           validate(node.right, node.val, max);
}
```

> [!WARNING]
> **Interview Trap**: Don't just compare with immediate children! Use min/max bounds for the entire subtree.

Invert Binary Tree ⭐ (Famous Google Question)

```
TreeNode invertTree(TreeNode root) {
    if (root == null) return null;

    // Swap children
    TreeNode temp = root.left;
    root.left = root.right;
    root.right = temp;

    // Recurse
    invertTree(root.left);
    invertTree(root.right);
```

```
        return root;
    }
```

🔥 Top Interview Patterns (With Solutions)

Pattern 1: Two Pointers - Same Tree ⭐⭐

Compare two trees node by node simultaneously.

```
    // Check if two trees are identical
    boolean isSameTree(TreeNode p, TreeNode q) {
        // Both null = same
        if (p == null && q == null) return true;
        // One null, one not = different
        if (p == null || q == null) return false;
        // Values must match AND both subtrees must match
        return p.val == q.val &&
               isSameTree(p.left, q.left) &&
               isSameTree(p.right, q.right);
    }
```

Pattern 2: Two Pointers - Symmetric Tree ⭐⭐

Check if a tree is a mirror of itself.

```
    boolean isSymmetric(TreeNode root) {
        if (root == null) return true;
        return isMirror(root.left, root.right);
    }

    boolean isMirror(TreeNode left, TreeNode right) {
        if (left == null && right == null) return true;
        if (left == null || right == null) return false;
        return left.val == right.val &&
               isMirror(left.left, right.right) &&   // outer
               isMirror(left.right, right.left);     // inner
    }
```

> [!TIP]
> **Key Insight**: Compare left.left with right.right (outer) and left.right with right.left (inner).

Pattern 3: BFS - Right Side View ⭐⭐

Return what you see from the right side of the tree.

```
List<Integer> rightSideView(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) return result;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        int levelSize = queue.size();
        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();
            // Last node in this level = visible from right
            if (i == levelSize - 1) {
                result.add(node.val);
            }
            if (node.left != null) queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }
    }
    return result;
}
```

> [!TIP]
> **Variation**: For **Left Side View**, check `if (i == 0)` instead.

Pattern 4: BST Property - Kth Smallest Element ⭐⭐⭐

Use inorder traversal (gives sorted order in BST).

```
int count = 0;
int result = 0;

int kthSmallest(TreeNode root, int k) {
```

```
        inorder(root, k);
        return result;
    }

    void inorder(TreeNode node, int k) {
        if (node == null) return;

        inorder(node.left, k);      // Go left first (smaller values)

        count++;
        if (count == k) {
            result = node.val;
            return;
        }

        inorder(node.right, k);     // Then right (larger values)
    }
```

> [!IMPORTANT]
> **BST Golden Rule**: Inorder traversal of BST = **sorted ascending order**!

Pattern 5: Bottom Up - Max Depth ⭐

Calculate height from bottom up.

```
    int maxDepth(TreeNode root) {
        if (root == null) return 0;
        int leftDepth = maxDepth(root.left);
        int rightDepth = maxDepth(root.right);
        return 1 + Math.max(leftDepth, rightDepth);
    }
```

Pattern 6: Bottom Up - Balanced Tree Check ⭐⭐

Check balance while computing height (single pass).

```
    boolean isBalanced(TreeNode root) {
        return getHeight(root) != -1;
    }
```

```
int getHeight(TreeNode node) {
    if (node == null) return 0;

    int leftHeight = getHeight(node.left);
    if (leftHeight == -1) return -1;  // Left subtree unbalanced

    int rightHeight = getHeight(node.right);
    if (rightHeight == -1) return -1; // Right subtree unbalanced

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

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

> [!TIP]
> **Why -1?** We use -1 as a signal for "unbalanced" - it propagates up and short-circuits the recursion.

Pattern 7: Zigzag Level Order ⭐⭐

BFS with alternating direction.

```
List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) return result;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    boolean leftToRight = true;

    while (!queue.isEmpty()) {
        int levelSize = queue.size();
        LinkedList<Integer> level = new LinkedList<>();

        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();

            // Add based on direction
            if (leftToRight) {
                level.addLast(node.val);
            } else {
```

```
                level.addFirst(node.val);  // Reverse order
            }

            if (node.left != null) queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }

        result.add(level);
        leftToRight = !leftToRight;  // Flip direction
    }
    return result;
}
```

📊 Pattern Summary Cheat Sheet

Pattern	When to Use	Template
DFS Recursion	Need info from children	`return f(left) OP f(right)`
BFS Level Order	Level-by-level processing	Queue + levelSize loop
Two Pointers	Compare two subtrees	Pass two nodes to helper
BST Inorder	Need sorted order	Left → Process → Right
Bottom Up	Compute while returning	Return -1 for invalid

> [!TIP]
> **Pro Tip**: Most tree problems can be solved with **recursion** by thinking:
>
> 1. What do I do at the current node?
> 2. What do I get from left subtree?
> 3. What do I get from right subtree?
> 4. How do I combine them?

> [!IMPORTANT]
> **Time Complexity** for most tree problems:
>
> - **O(n)** time - visit each node once
> - **O(h)** space - recursion stack (h = height, worst case O(n), best case O(log n))