🌳 Tree Interview Quick-Check Pattern

A structured mental model and pattern-driven guide to confidently solve any tree problem in interviews.

---

1. 🧠 When to Recognize a Tree Problem

Ask yourself:

- ❓ Is the input a **tree node** (with `.left` and `.right`)?
- ❓ Do I need to **traverse** the entire structure?
- ❓ Am I solving something **per level**, **per depth**, or **per subtree**?
- ❓ Does it mention **BST**, **balance**, or **lowest common ancestor**?
- ❓ Am I working with **recursion** or **layer-by-layer logic**?

> 💡 Most tree problems boil down to either **traversal**, **divide & conquer**, or **value aggregation**.

---

2. 🧰 Common Tree Traversal Techniques

Traversal Type	Use Case Example
Preorder (Node → Left → Right)	Clone tree, serialize tree
Inorder (Left → Node → Right)	BST-based problems
Postorder (Left → Right → Node)	Delete tree, evaluate expression tree
Level Order (BFS)	Per level values, zigzag, max per level
DFS (Recursive)	Depth tracking, path sums, balance
DFS (Iterative)	When recursion stack must be avoided
BFS (Queue-based)	Breadth traversal, shortest path

---

3. 🧪 Core Templates

✅ Recursive Inorder Traversal

```
function inorderTraversal(root) {
  const result = [];
  function dfs(node) {
    if (!node) return;
    dfs(node.left);
    result.push(node.val);
    dfs(node.right);
  }
  dfs(root);
  return result;
}
```

✅ Level Order Traversal (BFS)

```
function levelOrder(root) {
  if (!root) return [];
  const queue = [root], result = [];

  while (queue.length) {
    const levelSize = queue.length;
    const level = [];

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();
      level.push(node.val);
      if (node.left) queue.push(node.left);
      if (node.right) queue.push(node.right);
    }

    result.push(level);
  }
  return result;
}
```

✅ Max Depth of Binary Tree

```
function maxDepth(root) {
  if (!root) return 0;
  return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
}
```

✅ Path Sum (DFS)

```
function hasPathSum(root, target) {
  if (!root) return false;
  if (!root.left && !root.right) return root.val === target;
  return hasPathSum(root.left, target - root.val) ||
         hasPathSum(root.right, target - root.val);
}
```

✅ Validate Binary Search Tree

```
function isValidBST(root) {
  function helper(node, min, max) {
    if (!node) return true;
    if (node.val <= min || node.val >= max) return false;
    return helper(node.left, min, node.val) &&
           helper(node.right, node.val, max);
  }
  return helper(root, -Infinity, Infinity);
}
```

✅ Lowest Common Ancestor

```
function lowestCommonAncestor(root, p, q) {
  if (!root || root === p || root === q) return root;
  const left = lowestCommonAncestor(root.left, p, q);
  const right = lowestCommonAncestor(root.right, p, q);
  return left && right ? root : left || right;
}
```

---

4. 🧱 Edge Cases to Always Watch For

- Empty tree (`null`)
- One-node tree
- Tree is skewed (left-only or right-only)
- Duplicates (allowed in BST?)
- Target node not present
- Same node as both inputs (LCA, path)
- Recursion depth / stack overflow (deep trees)

> 🧠 Always test with a **single node**, and a **deep unbalanced** tree.

---

5. 🧠 Mental Model for Tree Problems

All tree problems fall into these 5 categories:

Category	Trigger Words	Key Techniques
Traversal	"Print", "List", "Visit"	DFS, BFS, Recursion
Path-based	"Sum", "Depth", "Diameter", "Path"	Recursion + return values
Search/Check	"Contains", "Find", "Validate BST"	DFS with bounds, comparisons

Divide & Conquer	"Merge", "Rebuild", "LCA"	Recursively combine results
Transform	"Serialize", "Flatten", "Mirror"	Pre/Post-order transformations

🔁 Problem Solving Loop

1. 📌 What’s the **return value**: number, list, boolean, node?
2. 🧭 Do I need to **traverse all nodes**?
3. 📤 Do I need to **return values or compute** during traversal?
4. 📈 What order matters: preorder, inorder, postorder, level order?
5. 🧱 Did I test edge cases (empty, single node, skewed)?

✅ Final Interview Checklist

- Is this DFS, BFS, or something custom?
- Do I need recursion or can it be iterative?
- Should I use a queue (for BFS) or stack (DFS)?
- Do I need to store **intermediate results** or just final value?
- Did I check **leaf nodes**, **depth**, **null nodes**, and **cycles**?