CODING INTERVIEW PREPARATION

Day 5: Trees (Part 2)

Complete Study Guide with Multiple Solutions

TABLE OF CONTENTS

Page

- Day 5 Overview 3
- Key Concepts 4
- **Problems:**
  - Problem 1: Invert/Flip Binary Tree 5-7
  - Problem 2: Maximum Depth of Binary Tree 8-10
  - Problem 3: Same Tree 11-13
  - Problem 4: Subtree of Another Tree 14-16
  - Problem 5: Lowest Common Ancestor of a BST 17-19
  - Problem 6: Validate Binary Search Tree 20-22
  - Problem 7: Binary Tree Level Order Traversal 23-25
  - Problem 8: Kth Smallest Element in a BST 26-28
- Day 5 Summary 29

DAY 5: TREES (PART 2)

Key Concepts for Today:

- **Binary Trees** structure and properties
- **Binary Search Trees (BST)** and their invariants
- **Depth-First Search (DFS)** traversal techniques
- **Breadth-First Search (BFS)** and level-order processing
- **Tree manipulation** and structural modifications
- **Tree comparison** and validation algorithms

Problems Index:

1. **Invert/Flip Binary Tree** (Easy)
2. **Maximum Depth of Binary Tree** (Easy)
3. **Same Tree** (Easy)
4. **Subtree of Another Tree** (Easy)
5. **Lowest Common Ancestor of a BST** (Easy)
6. **Validate Binary Search Tree** (Medium)
7. **Binary Tree Level Order Traversal** (Medium)
8. **Kth Smallest Element in a BST** (Medium)

CORE CONCEPTS EXPLANATION

Binary Tree Fundamentals

Definition: A hierarchical data structure where each node has at most two children, referred to as left child and right child.

Tree Node Structure:

```
public class TreeNode
{
    public int val;
    public TreeNode left;
    public TreeNode right;
    public TreeNode(int val=0, TreeNode left=null, TreeNode right=null)
    {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

Key Properties:

- **Height**: Maximum distance from root to any leaf
- **Depth**: Distance from root to a specific node
- **Complete Tree**: All levels filled except possibly the last, filled left to right
- **Perfect Tree**: All internal nodes have two children, all leaves at same level
- **Balanced Tree**: Heights of left and right subtrees differ by at most 1

Binary Search Tree (BST)

Definition: A binary tree with the ordering property: for every node, all values in left subtree < node value < all values in right subtree.

BST Properties:

- **In-order traversal** yields sorted sequence
- **Search, insert, delete** operations in **O(log n)** average case
- **Balanced BST**: Guarantees **O(log n)** worst-case operations
- **Degenerate BST**: Can become linear chain with **O(n)** operations

BST Operations:

```
// Search in BST
TreeNode Search(TreeNode root, int target)
{
    if (root == null || root.val == target) return root;
    return target < root.val ? Search(root.left, target) : Search(root.right, target);
}
```

Tree Traversal Algorithms

Depth-First Search (DFS):

1. Pre-order (Root → Left → Right):

```
void PreOrder(TreeNode root)
{
    if (root != null)
    {
        Process(root.val);      // Process root
        PreOrder(root.left);    // Traverse left
        PreOrder(root.right);   // Traverse right
    }
}
```

2. In-order (Left → Root → Right):

```
void InOrder(TreeNode root)
{
    if (root != null)
    {
        InOrder(root.left);     // Traverse left
        Process(root.val);      // Process root
        InOrder(root.right);    // Traverse right
    }
}
```

3. Post-order (Left → Right → Root):

```
void PostOrder(TreeNode root)
{
    if (root != null)
    {
        PostOrder(root.left);   // Traverse left
        PostOrder(root.right);  // Traverse right
        Process(root.val);      // Process root
    }
}
```

Breadth-First Search (BFS):

```
void BFS(TreeNode root)
{
    if (root == null) return;

    var queue = new Queue<TreeNode>();
    queue.Enqueue(root);

    while (queue.Count > 0)
    {
        TreeNode node = queue.Dequeue();
        Process(node.val);

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

Common Tree Patterns

1. Tree Validation:

- Check structural properties (BST invariant, balance, etc.)
- Use bounds checking or in-order traversal validation

2. Tree Modification:

- Structural changes (inversion, pruning)
- Value modifications while maintaining properties

3. Tree Comparison:

- Structural equality, subtree relationships
- Often requires simultaneous traversal of multiple trees

4. Tree Search and Retrieval:

- Finding specific nodes, paths, or values
- Utilize tree properties for efficient search

Why These Concepts Matter in Interviews

Fundamental Data Structure:

- Trees appear in 30%+ of coding interviews
- Foundation for many advanced algorithms and data structures
- Tests understanding of recursion and tree properties

Problem-Solving Patterns:

- **Divide and conquer**: Break problem into subproblems on subtrees
- **State management**: Passing information up and down tree
- **Multiple traversal strategies**: Choose optimal approach for each problem

Real-world Applications:

- **File systems**: Directory structures as trees
- **Databases**: B-trees, indexing, query optimization
- **Compilers**: Abstract syntax trees, expression parsing
- **AI/ML**: Decision trees, game trees, neural networks
- **Graphics**: Scene graphs, spatial partitioning

PROBLEM 1: INVERT/FLIP BINARY TREE

Problem Statement:

Given the root of a binary tree, invert the tree, and return its root.

Test Cases:

```
Example 1:
Input: root = [4,2,7,1,3,6,9]
Output: [4,7,2,9,6,3,1]
Explanation:
Original:     Inverted:
    4             4
   / \           / \
  2   7         7   2
 / \ / \       / \ / \
1  3 6  9     9  6 3  1

Example 2:
Input: root = [2,1,3]
Output: [2,3,1]
Explanation:
Original:  Inverted:
   2          2
  / \        / \
 1   3      3   1
```

```
Example 3:
Input: root = []
Output: []
Explanation: Empty tree remains empty.
```

Interview Questions & Answers:

1. **Q:** "What does it mean to 'invert' a binary tree?"
   A: Swap the left and right children of every node in the tree. This creates a mirror image of the original tree.
2. **Q:** "Can you do this in-place without extra space?"
   A: Yes, by swapping left and right children directly using recursion or iterative approach with a stack/queue.
3. **Q:** "What's the difference between recursive and iterative approaches?"
   A: Recursive is more intuitive but uses O(h) space for call stack. Iterative uses explicit stack/queue, giving more control over space usage.
4. **Q:** "How do you handle edge cases like empty tree or single node?"
   A: Empty tree returns null, single node returns itself unchanged. Both are natural base cases.
5. **Q:** "Is there a specific traversal order that works best?"
   A: Any order works (pre-order, post-order, level-order) since we're swapping at each node regardless of children's state.

Approach 1: Recursive - Post-order

Idea: Recursively invert left and right subtrees, then swap them at current node.

Time Complexity: O(n) - visit each node exactly once
Space Complexity: O(h) - recursion call stack, where h is tree height

```
using System;

public class Solution
{
    public TreeNode InvertTree(TreeNode root)
    {
        // Base case: empty tree
        if (root == null) return null;

        // Recursively invert left and right subtrees
        TreeNode leftInverted = InvertTree(root.left);
        TreeNode rightInverted = InvertTree(root.right);

        // Swap the children
        root.left = rightInverted;
        root.right = leftInverted;

        return root;
```

```
    }
}
```

Approach 2: Iterative with Stack (DFS)

Explanation: Use explicit stack to perform depth-first traversal and swap children at each node.

Time Complexity: O(n) - visit each node exactly once
Space Complexity: O(h) - stack size in worst case

```
using System;
using System.Collections.Generic;

public class Solution
{
    public TreeNode InvertTree(TreeNode root)
    {
        if (root == null) return null;

        var stack = new Stack<TreeNode>();
        stack.Push(root);

        while (stack.Count > 0)
        {
            TreeNode node = stack.Pop();

            // Swap left and right children
            TreeNode temp = node.left;
            node.left = node.right;
            node.right = temp;

            // Add children to stack for processing
            if (node.left != null) stack.Push(node.left);
            if (node.right != null) stack.Push(node.right);
        }

        return root;
    }
}
```

Approach 3: Optimal - Iterative with Queue (BFS)

Detailed Reasoning: Use breadth-first search with queue for level-by-level inversion. This approach is optimal for trees that are wide rather than deep, and provides predictable memory usage patterns.

Time Complexity: O(n) - visit each node exactly once
Space Complexity: O(w) - where w is maximum width of tree

```
using System;
using System.Collections.Generic;
```

```
public class Solution
{
    /// <summary>
    /// Inverts binary tree using iterative BFS approach
    /// Time: O(n), Space: O(w) where w is max width
    /// </summary>
    public TreeNode InvertTree(TreeNode root)
    {
        // Handle empty tree
        if (root == null) return null;

        // Use queue for level-order traversal
        var queue = new Queue<TreeNode>();
        queue.Enqueue(root);

        while (queue.Count > 0)
        {
            TreeNode current = queue.Dequeue();

            // Swap left and right children
            TreeNode temp = current.left;
            current.left = current.right;
            current.right = temp;

            // Add non-null children to queue for next level processing
            if (current.left != null)
            {
                queue.Enqueue(current.left);
            }
            if (current.right != null)
            {
                queue.Enqueue(current.right);
            }
        }

        return root;
    }
}
```

Key Takeaways:

- **In-place modification**: Swap pointers without creating new nodes
- **Any traversal works**: Pre-order, post-order, or level-order all achieve same result
- **Space complexity varies**: Recursive uses call stack, iterative uses explicit data structure

PROBLEM 2: MAXIMUM DEPTH OF BINARY TREE

Problem Statement:

Given the root of a binary tree, return its maximum depth. A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Test Cases:

```
Example 1:
Input: root = [3,9,20,null,null,15,7]
Output: 3
Explanation:
    3
   / \
  9  20
    /  \
   15   7
Maximum path: 3 → 20 → 15 (or 7), depth = 3

Example 2:
Input: root = [1,null,2]
Output: 2
Explanation:
  1
   \
    2
Maximum path: 1 → 2, depth = 2

Example 3:
Input: root = []
Output: 0
Explanation: Empty tree has depth 0.
```

Interview Questions & Answers:

1. **Q:** "What's the difference between height and depth of a tree?"
   A: Conventionally, height is measured from leaves up (leaf height = 0), depth from root down (root depth = 0). This problem asks for depth but implements height + 1.
2. **Q:** "How do you handle unbalanced trees efficiently?"
   A: All approaches have same time complexity O(n) since we must visit every node. Space complexity varies based on approach.
3. **Q:** "Can you solve this iteratively instead of recursively?"
   A: Yes, using level-order traversal with queue and tracking current depth/level.
4. **Q:** "What if the tree is extremely deep (like a linked list)?"
   A: Recursive approach might cause stack overflow. Iterative approach with explicit stack is safer.
5. **Q:** "How would you find the minimum depth instead?"
   A: Similar approach, but return 1 + min(left_depth, right_depth), with special handling for nodes with only one child.

Approach 1: Recursive DFS

Idea: Use recursion to find maximum depth of left and right subtrees, then add 1 for current node.

Time Complexity: O(n) - visit each node once
Space Complexity: O(h) - recursion call stack, where h is tree height

```
using System;

public class Solution
{
    public int MaxDepth(TreeNode root)
    {
        // Base case: empty tree has depth 0
        if (root == null) return 0;

        // Recursively find depth of left and right subtrees
        int leftDepth = MaxDepth(root.left);
        int rightDepth = MaxDepth(root.right);

        // Return maximum depth plus 1 for current node
        return Math.Max(leftDepth, rightDepth) + 1;
    }
}
```

Approach 2: Iterative DFS with Stack

Explanation: Use explicit stack to simulate recursion, tracking both node and its current depth.

Time Complexity: O(n) - visit each node once
Space Complexity: O(h) - stack size in worst case

```
using System;
using System.Collections.Generic;

public class Solution
{
    public int MaxDepth(TreeNode root)
    {
        if (root == null) return 0;

        var stack = new Stack<(TreeNode node, int depth)>();
        stack.Push((root, 1));
        int maxDepth = 0;

        while (stack.Count > 0)
        {
            var (node, depth) = stack.Pop();

            // Update maximum depth seen so far
            maxDepth = Math.Max(maxDepth, depth);
```

```
            // Add children with incremented depth
            if (node.left != null)
            {
                stack.Push((node.left, depth + 1));
            }
            if (node.right != null)
            {
                stack.Push((node.right, depth + 1));
            }
        }

        return maxDepth;
    }
}
```

Approach 3: Optimal - Iterative BFS with Queue

Detailed Reasoning: Use level-order traversal to process tree level by level. This approach is optimal for wide trees and provides clear understanding of tree levels, making it easier to extend for level-specific operations.

Time Complexity: O(n) - visit each node once
Space Complexity: O(w) - where w is maximum width of tree

```
using System;
using System.Collections.Generic;

public class Solution
{
    /// <summary>
    /// Finds maximum depth using iterative BFS level-order traversal
    /// Time: O(n), Space: O(w) where w is max width
    /// </summary>
    public int MaxDepth(TreeNode root)
    {
        // Handle empty tree
        if (root == null) return 0;

        var queue = new Queue<TreeNode>();
        queue.Enqueue(root);
        int depth = 0;

        // Process each level completely
        while (queue.Count > 0)
        {
            int levelSize = queue.Count;
            depth++; // Increment depth for each level

            // Process all nodes at current level
            for (int i = 0; i < levelSize; i++)
            {
                TreeNode node = queue.Dequeue();

                // Add children for next level processing
```

```
                if (node.left != null)
                {
                    queue.Enqueue(node.left);
                }
                if (node.right != null)
                {
                    queue.Enqueue(node.right);
                }
            }
        }

        return depth;
    }
}
```

Key Takeaways:

- **Recursive simplicity**: Most intuitive approach for tree problems
- **Level-by-level processing**: BFS naturally handles level-specific logic
- **Space complexity trade-offs**: Call stack vs explicit data structure

PROBLEM 3: SAME TREE

Problem Statement:

Given the roots of two binary trees `p` and `q`, write a function to check if they are the same or not. Two binary trees are considered the same if they are structurally identical, and the nodes have the same values.

Test Cases:

```
Example 1:
Input: p = [1,2,3], q = [1,2,3]
Output: true
Explanation: Both trees have identical structure and values
    1       1
   / \     / \
  2   3   2   3

Example 2:
Input: p = [1,2], q = [1,null,2]
Output: false
Explanation: Different structure - left vs right child
  1     1
 /       \
2         2

Example 3:
Input: p = [1,2,1], q = [1,1,2]
Output: false
Explanation: Same structure but different values
```

```
    1       1
   / \     / \
  2   1   1   2
```

Interview Questions & Answers:

1. **Q:** "What makes two trees 'the same'?"
   A: They must have identical structure (same shape) and identical values at corresponding positions.
2. **Q:** "How do you handle null nodes in the comparison?"
   A: If both nodes are null, they're equal. If one is null and other isn't, they're not equal.
3. **Q:** "Can you optimize for trees with many duplicate subtrees?"
   A: For this problem, no optimization helps since we must check every node. But for repeated comparisons, subtree hashing could help.
4. **Q:** "What if trees have different structures but same values?"
   A: They're not the same. Structure must be identical, not just values.
5. **Q:** "How would you extend this to check if trees are mirrors of each other?"
   A: Modify comparison to check left of first tree against right of second tree, and vice versa.

Approach 1: Recursive Comparison

Idea: Recursively compare current nodes and their left and right subtrees.

Time Complexity: O(min(m,n)) - visit nodes until first difference found
Space Complexity: O(min(h1,h2)) - recursion call stack

```
using System;

public class Solution
{
    public bool IsSameTree(TreeNode p, TreeNode q)
    {
        // Base case: both nodes are null
        if (p == null && q == null) return true;

        // One is null, other is not
        if (p == null || q == null) return false;

        // Both non-null: check value and recursively check subtrees
        if (p.val != q.val) return false;

        return IsSameTree(p.left, q.left) && IsSameTree(p.right, q.right);
    }
}
```

Approach 2: Iterative with Stack

Explanation: Use stack to simulate recursion, pushing pairs of nodes to compare.

Time Complexity: O(min(m,n)) - visit nodes until difference found
Space Complexity: O(min(h1,h2)) - stack size

```
using System;
using System.Collections.Generic;

public class Solution
{
    public bool IsSameTree(TreeNode p, TreeNode q)
    {
        var stack = new Stack<(TreeNode, TreeNode)>();
        stack.Push((p, q));

        while (stack.Count > 0)
        {
            var (node1, node2) = stack.Pop();

            // Both null - continue
            if (node1 == null && node2 == null) continue;

            // One null, other not - trees are different
            if (node1 == null || node2 == null) return false;

            // Different values - trees are different
            if (node1.val != node2.val) return false;

            // Add children pairs to stack
            stack.Push((node1.left, node2.left));
            stack.Push((node1.right, node2.right));
        }

        return true;
    }
}
```

Approach 3: Optimal - Iterative BFS with Queue

Detailed Reasoning: Use level-order traversal to compare trees level by level. This approach provides early termination and clear level-by-level comparison logic, which can be helpful for debugging and understanding tree structure.

Time Complexity: O(min(m,n)) - stop at first difference
Space Complexity: O(min(w1,w2)) - where w is maximum width

```
using System;
using System.Collections.Generic;

public class Solution
{
```

```
    /// <summary>
    /// Compares two trees using iterative BFS approach
    /// Time: O(min(m,n)), Space: O(min(w1,w2))
    /// </summary>
    public bool IsSameTree(TreeNode p, TreeNode q)
    {
        // Use queue for level-order traversal of both trees
        var queue = new Queue<(TreeNode, TreeNode)>();
        queue.Enqueue((p, q));

        while (queue.Count > 0)
        {
            var (node1, node2) = queue.Dequeue();

            // Both nodes are null - this part matches
            if (node1 == null && node2 == null)
            {
                continue;
            }

            // One node is null, other is not - trees differ
            if (node1 == null || node2 == null)
            {
                return false;
            }

            // Both nodes exist but have different values - trees differ
            if (node1.val != node2.val)
            {
                return false;
            }

            // Add corresponding children pairs for next level comparison
            queue.Enqueue((node1.left, node2.left));
            queue.Enqueue((node1.right, node2.right));
        }

        return true; // All corresponding nodes matched
    }
}
```

Key Takeaways:

- **Early termination**: Return false as soon as difference is found
- **Null handling**: Both null = equal, one null = not equal
- **Simultaneous traversal**: Process both trees in lockstep

PROBLEM 4: SUBTREE OF ANOTHER TREE

Problem Statement:

Given the roots of two binary trees `root` and `subRoot`, return `true` if there is a subtree of `root` with the same structure and node values of `subRoot` and `false` otherwise.

A subtree of a binary tree `tree` is a tree that consists of a node in `tree` and all of this node's descendants. The tree `tree` could also be considered as a subtree of itself.

Test Cases:

```
Example 1:
Input: root = [3,4,5,1,2], subRoot = [4,1,2]
Output: true
Explanation: Subtree rooted at node 4 matches subRoot
    3
   / \
  4   5
 / \
1   2

Example 2:
Input: root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]
Output: false
Explanation: Subtree rooted at 4 has additional node 0
    3
   / \
  4   5
 / \
1   2
   /
  0

Example 3:
Input: root = [1,1], subRoot = [1]
Output: true
Explanation: Single node 1 appears as subtree
  1
 /
1
```

Interview Questions & Answers:

1. **Q:** "What's the difference between subtree and just finding a node with same value?"
   A: Subtree must match the entire structure and all descendant values, not just a single node value.
2. **Q:** "Can a tree be considered a subtree of itself?"
   A: Yes, the problem explicitly states this. The entire tree matches itself.

3. **Q:** "How do you optimize when trees have many duplicate values?"
   A: Use tree serialization to convert comparison to string matching, or use tree hashing for faster comparisons.
4. **Q:** "What if subRoot is null?"
   A: Null tree is typically considered a subtree of any tree, but problem constraints usually prevent this case.
5. **Q:** "How would you find all occurrences of a subtree?"
   A: Modify algorithm to continue searching after finding first match, collecting all matching root positions.

Approach 1: Brute Force - Check Every Node

Idea: For each node in the main tree, check if the subtree rooted at that node matches the target subtree.

Time Complexity: O(m × n) - check subtree equality at each of m nodes
Space Complexity: O(max(h1, h2)) - recursion call stack

```
using System;

public class Solution
{
    public bool IsSubtree(TreeNode root, TreeNode subRoot)
    {
        if (root == null) return false;

        // Check if current tree matches subRoot
        if (IsSameTree(root, subRoot)) return true;

        // Recursively check left and right subtrees
        return IsSubtree(root.left, subRoot) || IsSubtree(root.right, subRoot);
    }

    private bool IsSameTree(TreeNode p, TreeNode q)
    {
        if (p == null && q == null) return true;
        if (p == null || q == null) return false;
        if (p.val != q.val) return false;

        return IsSameTree(p.left, q.left) && IsSameTree(p.right, q.right);
    }
}
```

Approach 2: Tree Serialization

Explanation: Convert both trees to string representations, then use string matching algorithms to find subtree.

Time Complexity: O(m + n) - serialize both trees and string search
Space Complexity: O(m + n) - store serialized strings

```
using System;
using System.Text;

public class Solution
{
    public bool IsSubtree(TreeNode root, TreeNode subRoot)
    {
        string rootStr = Serialize(root);
        string subStr = Serialize(subRoot);

        return rootStr.Contains(subStr);
    }

    private string Serialize(TreeNode node)
    {
        if (node == null) return "null";

        var sb = new StringBuilder();
        sb.Append("#"); // Delimiter to avoid false matches
        sb.Append(node.val);
        sb.Append(",");
        sb.Append(Serialize(node.left));
        sb.Append(",");
        sb.Append(Serialize(node.right));

        return sb.ToString();
    }
}
```

Approach 3: Optimal - Tree Traversal with Optimization

Detailed Reasoning: Use DFS traversal but optimize by first checking if current node value matches subRoot's root value before doing expensive subtree comparison. This reduces unnecessary comparisons.

Time Complexity: O(m × n) worst case, but much better average case
Space Complexity: O(max(h1, h2)) - recursion call stack

```
using System;

public class Solution
{
    /// <summary>
    /// Checks if subRoot is subtree of root using optimized DFS
    /// Time: O(m*n) worst case, Space: O(max(h1,h2))
    /// </summary>
    public bool IsSubtree(TreeNode root, TreeNode subRoot)
    {
        // Empty main tree cannot contain non-empty subtree
        if (root == null) return subRoot == null;

        // Optimization: only check subtree equality if root values match
        if (root.val == subRoot.val && IsSameTree(root, subRoot))
```

```
        {
            return true;
        }

        // Recursively search in left and right subtrees
        return IsSubtree(root.left, subRoot) || IsSubtree(root.right, subRoot);
    }

    /// <summary>
    /// Helper function to check if two trees are identical
    /// </summary>
    private bool IsSameTree(TreeNode p, TreeNode q)
    {
        // Both null - trees match at this point
        if (p == null && q == null) return true;

        // One null, other not - trees don't match
        if (p == null || q == null) return false;

        // Both non-null - check value and recursively check subtrees
        return p.val == q.val &&
               IsSameTree(p.left, q.left) &&
               IsSameTree(p.right, q.right);
    }
}
```

Key Takeaways:

- **Two-phase checking**: First find candidate nodes, then verify subtree equality
- **Early termination**: Stop searching once subtree is found
- **Value-based optimization**: Only compare subtrees when root values match

PROBLEM 5: LOWEST COMMON ANCESTOR OF A BST

Problem Statement:

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST. According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself)."

Test Cases:

```
Example 1:
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: LCA of nodes 2 and 8 is 6
        6
       / \
      2   8
     / \ / \
```

```
    0  4 7  9
      / \
     3   5

Example 2:
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: LCA of nodes 2 and 4 is 2 (node can be ancestor of itself)

Example 3:
Input: root = [2,1], p = 2, q = 1
Output: 2
Explanation: LCA of 2 and 1 is 2
```

Interview Questions & Answers:

1. **Q:** "How does BST property help find LCA efficiently?"
   A: BST property allows us to determine which subtree contains each node without full traversal. If both nodes are on same side, LCA is in that subtree.
2. **Q:** "What if one node is ancestor of the other?"
   A: The ancestor node itself is the LCA. For example, if p is ancestor of q, then p is the LCA.
3. **Q:** "Can you solve this without recursion?"
   A: Yes, iterative approach works well since we're following a single path down the tree based on BST property.
4. **Q:** "What if the nodes don't exist in the BST?"
   A: Problem typically guarantees both nodes exist. In practice, you'd need to verify existence first.
5. **Q:** "How is this different from LCA in a general binary tree?"
   A: General binary tree requires full subtree exploration. BST property allows direct path navigation, reducing time complexity.

Approach 1: Recursive with Full Tree Search

Idea: Ignore BST property and search entire tree to find LCA (same as general binary tree approach).

Time Complexity: O(n) - potentially visit all nodes
Space Complexity: O(h) - recursion call stack

```
using System;

public class Solution
{
    public TreeNode LowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
    {
        // Base case
        if (root == null || 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 sides found nodes, current root is LCA
        if (left != null && right != null) return root;

        // Return whichever side found a node
        return left != null ? left : right;
    }
}
```

Approach 2: Recursive Using BST Property

Explanation: Utilize BST property to determine which subtree to search based on node values.

Time Complexity: O(h) - follow single path down tree
Space Complexity: O(h) - recursion call stack

```
using System;

public class Solution
{
    public TreeNode LowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
    {
        // Base case
        if (root == null) return null;

        // Both nodes are in left subtree
        if (p.val < root.val && q.val < root.val)
        {
            return LowestCommonAncestor(root.left, p, q);
        }

        // Both nodes are in right subtree
        if (p.val > root.val && q.val > root.val)
        {
            return LowestCommonAncestor(root.right, p, q);
        }

        // Nodes are on different sides or one equals root
        return root;
    }
}
```

Approach 3: Optimal - Iterative Using BST Property

Detailed Reasoning: Use BST property iteratively to navigate directly to LCA without recursion. This is optimal because it uses constant space and follows the shortest path to the answer.

Time Complexity: O(h) - follow single path, where h is tree height
Space Complexity: O(1) - only using pointer variables

```
using System;

public class Solution
{
    /// <summary>
    /// Finds LCA in BST using iterative approach with BST property
    /// Time: O(h), Space: O(1) where h is tree height
    /// </summary>
    public TreeNode LowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
    {
        TreeNode current = root;

        while (current != null)
        {
            // Both nodes are in left subtree
            if (p.val < current.val && q.val < current.val)
            {
                current = current.left;
            }
            // Both nodes are in right subtree
            else if (p.val > current.val && q.val > current.val)
            {
                current = current.right;
            }
            // Split point found: nodes are on different sides or one equals current
            else
            {
                return current;
            }
        }

        return null; // Should not reach here with valid input
    }
}
```

Key Takeaways:

- **BST property utilization**: Compare values to determine search direction
- **Split point identification**: LCA is where paths to p and q diverge
- **Iterative optimization**: Avoid recursion overhead with simple loop

PROBLEM 6: VALIDATE BINARY SEARCH TREE

Problem Statement:

Given the root of a binary tree, determine if it is a valid binary search tree (BST). A valid BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.

- Both the left and right subtrees must also be binary search trees.

Test Cases:

```
Example 1:
Input: root = [2,1,3]
Output: true
Explanation: Valid BST
    2
   / \
  1   3

Example 2:
Input: root = [5,1,4,null,null,3,6]
Output: false
Explanation: Invalid BST - node 3 violates BST property
    5
   / \
  1   4
     / \
    3   6
Node 3 is in right subtree of 5 but 3 < 5

Example 3:
Input: root = [2,2,2]
Output: false
Explanation: BST requires strict inequality (no duplicates in this definition)
```

Interview Questions & Answers:

1. **Q:** "What's wrong with just checking each node against its immediate children?"
   A: A node must satisfy constraints from all its ancestors, not just immediate parent. For example, a left child's right subtree must still be less than the original ancestor.
2. **Q:** "How do you handle duplicate values?"
   A: Problem definition typically requires strict inequality (< and >), so duplicates make BST invalid. Some definitions allow <= on one side.
3. **Q:** "Can you use in-order traversal to validate BST?"
   A: Yes! In-order traversal of valid BST produces sorted sequence. Check if traversal is strictly increasing.
4. **Q:** "What about integer overflow with min/max bounds?"
   A: Use nullable integers or long values for bounds, or use in-order traversal approach to avoid bound issues.
5. **Q:** "How would you handle trees with null nodes?"
   A: Null nodes don't violate BST property. Validation focuses on existing nodes and their relationships.

Approach 1: Incorrect - Local Validation Only

Idea: Only check each node against its immediate children (this approach is flawed but educational).

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

```
using System;

public class Solution
{
    // This approach is INCORRECT - shown for educational purposes
    public bool IsValidBST(TreeNode root)
    {
        if (root == null) return true;

        // This is wrong! Only checks immediate children
        bool leftValid = root.left == null || root.left.val < root.val;
        bool rightValid = root.right == null || root.right.val > root.val;

        // Recursively check subtrees
        return leftValid && rightValid &&
               IsValidBST(root.left) && IsValidBST(root.right);
    }
}
```

Approach 2: In-Order Traversal

Explanation: Use in-order traversal to generate sequence, then check if sequence is strictly increasing.

Time Complexity: O(n) - visit each node once
Space Complexity: O(h) - recursion call stack (or O(n) if storing values)

```
using System;

public class Solution
{
    private int? prev = null;

    public bool IsValidBST(TreeNode root)
    {
        prev = null; // Reset for each call
        return InOrderValidate(root);
    }

    private bool InOrderValidate(TreeNode root)
    {
        if (root == null) return true;

        // Check left subtree
        if (!InOrderValidate(root.left)) return false;
```

```
        // Check current node against previous in in-order sequence
        if (prev != null && root.val <= prev) return false;
        prev = root.val;

        // Check right subtree
        return InOrderValidate(root.right);
    }
}
```

Approach 3: Optimal - Bounds Checking

Detailed Reasoning: Pass down valid range bounds for each node. Each node must be within its allowed range, and we update bounds for recursive calls. This is optimal because it validates BST property correctly in single pass.

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

```
using System;

public class Solution
{
    /// <summary>
    /// Validates BST using bounds checking approach
    /// Time: O(n), Space: O(h)
    /// </summary>
    public bool IsValidBST(TreeNode root)
    {
        return ValidateWithBounds(root, null, null);
    }

    /// <summary>
    /// Validates node is within valid bounds and recursively validates subtrees
    /// </summary>
    /// <param name="node">Current node to validate</param>
    /// <param name="minBound">Minimum allowed value (exclusive)</param>
    /// <param name="maxBound">Maximum allowed value (exclusive)</param>
    private bool ValidateWithBounds(TreeNode node, int? minBound, int? maxBound)
    {
        // Null nodes are valid
        if (node == null) return true;

        // Check if current node violates bounds
        if ((minBound != null && node.val <= minBound) ||
            (maxBound != null && node.val >= maxBound))
        {
            return false;
        }

        // Recursively validate subtrees with updated bounds
        // Left subtree: all values must be < node.val
        // Right subtree: all values must be > node.val
        return ValidateWithBounds(node.left, minBound, node.val) &&
```

```
                ValidateWithBounds(node.right, node.val, maxBound);
        }
    }
```

Key Takeaways:

- **Global constraints**: Each node must satisfy constraints from all ancestors
- **Bounds propagation**: Pass valid range down the recursion tree
- **In-order property**: Valid BST produces sorted in-order traversal

PROBLEM 7: BINARY TREE LEVEL ORDER TRAVERSAL

Problem Statement:

Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

Test Cases:

```
Example 1:
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
Explanation:
    3      ← Level 0: [3]
   / \
  9  20    ← Level 1: [9,20]
    /  \
   15   7  ← Level 2: [15,7]

Example 2:
Input: root = [1]
Output: [[1]]
Explanation: Single node at level 0

Example 3:
Input: root = []
Output: []
Explanation: Empty tree has no levels
```

Interview Questions & Answers:

1. **Q:** "How do you keep track of which nodes belong to which level?"
   A: Process nodes level by level using queue. Process all nodes at current level before moving to next level.
2. **Q:** "Can you do this without using extra space for queue?"
   A: No efficient way without queue for BFS. You could use DFS with level parameter, but that changes the problem nature.

3. **Q:** "How would you modify this for right-to-left level order?"
   A: Process levels the same way, but reverse each level list before adding to result, or add nodes to front of each level list.
4. **Q:** "What if you need to return bottom-up level order?"
   A: Same algorithm, but reverse the final result list, or use stack to store levels.
5. **Q:** "How do you handle very wide trees efficiently?"
   A: Algorithm is already optimal for wide trees. Space complexity is O(w) where w is maximum width.

Approach 1: BFS with Node-Level Pairs

Idea: Use queue to store (node, level) pairs, then group nodes by level.

Time Complexity: O(n) - visit each node once
Space Complexity: O(n) - queue and result storage

```
using System;
using System.Collections.Generic;

public class Solution
{
    public IList<IList<int>> LevelOrder(TreeNode root)
    {
        var result = new List<IList<int>>();
        if (root == null) return result;

        var queue = new Queue<(TreeNode node, int level)>();
        queue.Enqueue((root, 0));

        while (queue.Count > 0)
        {
            var (node, level) = queue.Dequeue();

            // Ensure we have a list for this level
            if (level >= result.Count)
            {
                result.Add(new List<int>());
            }

            // Add current node value to its level
            result[level].Add(node.val);

            // Add children with next level
            if (node.left != null)
            {
                queue.Enqueue((node.left, level + 1));
            }
            if (node.right != null)
            {
                queue.Enqueue((node.right, level + 1));
            }
        }
```

```
        return result;
    }
}
```

Approach 2: DFS with Level Parameter

Explanation: Use DFS recursion but pass level parameter to build level-order result.

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

```
using System;
using System.Collections.Generic;

public class Solution
{
    public IList<IList<int>> LevelOrder(TreeNode root)
    {
        var result = new List<IList<int>>();
        DFS(root, 0, result);
        return result;
    }

    private void DFS(TreeNode node, int level, IList<IList<int>> result)
    {
        if (node == null) return;

        // Ensure we have a list for current level
        if (level >= result.Count)
        {
            result.Add(new List<int>());
        }

        // Add current node to its level
        result[level].Add(node.val);

        // Recursively process children at next level
        DFS(node.left, level + 1, result);
        DFS(node.right, level + 1, result);
    }
}
```

Approach 3: Optimal - BFS Level-by-Level Processing

Detailed Reasoning: Use BFS but process entire levels at once by tracking queue size. This is optimal because it clearly separates levels and processes nodes in true level-order fashion.

Time Complexity: O(n) - visit each node exactly once
Space Complexity: O(w) - where w is maximum width of tree

```
using System;
using System.Collections.Generic;

public class Solution
{
    /// <summary>
    /// Returns level order traversal using BFS with level-by-level processing
    /// Time: O(n), Space: O(w) where w is maximum width
    /// </summary>
    public IList<IList<int>> LevelOrder(TreeNode root)
    {
        var result = new List<IList<int>>();

        // Handle empty tree
        if (root == null) return result;

        var queue = new Queue<TreeNode>();
        queue.Enqueue(root);

        while (queue.Count > 0)
        {
            int levelSize = queue.Count; // Number of nodes at current level
            var currentLevel = new List<int>();

            // Process all nodes at current level
            for (int i = 0; i < levelSize; i++)
            {
                TreeNode node = queue.Dequeue();
                currentLevel.Add(node.val);

                // Add children for next level processing
                if (node.left != null)
                {
                    queue.Enqueue(node.left);
                }
                if (node.right != null)
                {
                    queue.Enqueue(node.right);
                }
            }

            // Add completed level to result
            result.Add(currentLevel);
        }

        return result;
    }
}
```

Key Takeaways:

- **Level-by-level processing**: Use queue size to process complete levels
- **BFS for level order**: Queue naturally handles left-to-right order
- **Space optimization**: Process and store one level at a time

PROBLEM 8: KTH SMALLEST ELEMENT IN A BST

Problem Statement:

Given the root of a binary search tree, and an integer `k`, return the `k`th smallest value (1-indexed) of all the values of the nodes in the tree.

Test Cases:

```
Example 1:
Input: root = [3,1,4,null,2], k = 1
Output: 1
Explanation: In-order: [1,2,3,4], 1st smallest = 1
    3
   / \
  1   4
   \
    2

Example 2:
Input: root = [5,3,6,2,4,null,null,1], k = 3
Output: 3
Explanation: In-order: [1,2,3,4,5,6], 3rd smallest = 3
      5
     / \
    3   6
   / \
  2   4
 /
1

Example 3:
Input: root = [1], k = 1
Output: 1
Explanation: Single node tree, 1st smallest = 1
```

Interview Questions & Answers:

1. **Q:** "Why is in-order traversal perfect for this problem?"
   A: In-order traversal of BST visits nodes in sorted ascending order, so kth element in traversal is kth smallest.
2. **Q:** "Can you optimize to avoid traversing the entire tree?"
   A: Yes, stop the in-order traversal as soon as we've found the kth element.

3. **Q:** "What if k is larger than the number of nodes?"
   A: Problem typically guarantees valid k, but in practice, you'd validate k <= tree size.
4. **Q:** "How would you handle frequent queries for different k values?"
   A: Augment BST nodes with subtree sizes to calculate rank directly, or cache in-order traversal result.
5. **Q:** "Can you do this iteratively instead of recursively?"
   A: Yes, use explicit stack to simulate in-order traversal iteratively.

Approach 1: In-Order Traversal with Full Array

Idea: Perform complete in-order traversal, store all values, then return kth element.

Time Complexity: O(n) - visit all nodes and create array
Space Complexity: O(n) - store all node values

```
using System;
using System.Collections.Generic;

public class Solution
{
    public int KthSmallest(TreeNode root, int k)
    {
        var values = new List<int>();
        InOrder(root, values);
        return values[k - 1]; // k is 1-indexed
    }

    private void InOrder(TreeNode root, List<int> values)
    {
        if (root == null) return;

        InOrder(root.left, values);
        values.Add(root.val);
        InOrder(root.right, values);
    }
}
```

Approach 2: Iterative In-Order with Stack

Explanation: Use explicit stack to perform in-order traversal iteratively, stopping when kth element is found.

Time Complexity: O(h + k) - where h is height, k elements processed
Space Complexity: O(h) - stack size

```
using System;
using System.Collections.Generic;

public class Solution
{
```

```
    public int KthSmallest(TreeNode root, int k)
    {
        var stack = new Stack<TreeNode>();
        TreeNode current = root;
        int count = 0;

        while (current != null || stack.Count > 0)
        {
            // Go to leftmost node
            while (current != null)
            {
                stack.Push(current);
                current = current.left;
            }

            // Process current node
            current = stack.Pop();
            count++;

            if (count == k)
            {
                return current.val;
            }

            // Move to right subtree
            current = current.right;
        }

        return -1; // Should not reach here with valid input
    }
}
```

Approach 3: Optimal - Early Termination Recursive In-Order

Detailed Reasoning: Use recursive in-order traversal but stop as soon as we find the kth element. This is optimal because it minimizes the number of nodes visited and uses minimal space.

Time Complexity: O(h + k) - height to reach leftmost + k elements
Space Complexity: O(h) - recursion call stack

```
using System;

public class Solution
{
    /// <summary>
    /// Finds kth smallest element using early-termination in-order traversal
    /// Time: O(h + k), Space: O(h)
    /// </summary>
    public int KthSmallest(TreeNode root, int k)
    {
        return InOrderKth(root, ref k);
    }
```

```
    /// <summary>
    /// Performs in-order traversal and returns kth smallest element
    /// Uses ref parameter to track remaining count across recursive calls
    /// </summary>
    private int InOrderKth(TreeNode root, ref int k)
    {
        if (root == null) return -1;

        // Search in left subtree first
        int leftResult = InOrderKth(root.left, ref k);
        if (leftResult != -1) return leftResult; // Found in left subtree

        // Process current node
        k--;
        if (k == 0) return root.val; // Found kth smallest

        // Search in right subtree if not found yet
        return InOrderKth(root.right, ref k);
    }
}
```

Key Takeaways:

- **BST in-order property**: In-order traversal gives sorted sequence
- **Early termination**: Stop traversal once kth element is found
- **Space-time trade-off**: Recursive vs iterative vs full array approaches

DAY 5 SUMMARY

Summary of Concepts Covered

Binary Tree Fundamentals:

- **Tree structure**: Hierarchical data with parent-child relationships
- **Tree properties**: Height, depth, balance, completeness
- **Node relationships**: Parent, child, ancestor, descendant, sibling

Tree Traversal Techniques:

- **DFS (Depth-First Search)**: Pre-order, in-order, post-order
- **BFS (Breadth-First Search)**: Level-order traversal
- **Recursive vs Iterative**: Trade-offs between simplicity and space usage

Binary Search Tree (BST):

- **BST property**: Left < root < right for all nodes
- **BST operations**: Search, insert, delete in O(log n) average case
- **In-order traversal**: Produces sorted sequence for BST

Tree Problem Patterns:

- **Tree modification**: Structural changes (inversion, pruning)
- **Tree validation**: Checking properties and invariants
- **Tree comparison**: Equality, subtree relationships
- **Tree search**: Finding specific nodes or values

Time & Space Complexity Summary

Problem Type	Brute Force	Optimized	Key Technique
Tree Inversion	**O(n)** time/space	**O(n)** time, **O(h)** space	In-place pointer swapping
Maximum Depth	**O(n)** recursive	**O(n)** iterative BFS	Level-by-level processing
Same Tree	**O(min(m,n))**	**O(min(m,n))**	Early termination
Subtree Check	**O(m×n)**	**O(m+n)** serialization	Tree serialization/hashing
BST LCA	**O(n)** general tree	**O(h)** BST property	BST property navigation
BST Validation	**O(n)** local check	**O(n)** bounds check	Global constraint propagation
Level Order	**O(n)** time/space	**O(n)** time, **O(w)** space	Queue-based BFS
Kth Smallest BST	**O(n)** full traversal	**O(h+k)** early stop	Early termination in-order

Pattern Recognition Guide

Use DFS When:

- Need to process entire subtrees before parent
- Working with tree depth or path-related problems
- Validating tree properties that depend on subtree results
- Memory usage should be proportional to height, not width

Use BFS When:

- Processing nodes level by level
- Finding shortest path in unweighted tree
- Tree width is manageable
- Need to process all nodes at same depth together

Use BST Property When:

- Tree is guaranteed to be BST
- Looking for specific values or ranges
- Need efficient search, insertion, or deletion
- Working with sorted data represented as tree

Common Tree Problem Patterns

Pattern 1: Tree Modification

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

    // Modify subtrees first
    root.left = ModifyTree(root.left);
    root.right = ModifyTree(root.right);

    // Apply modification at current node
    // (swap, remove, transform, etc.)

    return root;
}
```

Pattern 2: Tree Validation

```
bool ValidateTree(TreeNode root, constraints) {
    if (root == null) return true;

    // Check current node constraints
    if (!isValid(root, constraints)) return false;

    // Recursively validate subtrees with updated constraints
    return ValidateTree(root.left, updatedConstraints) &&
           ValidateTree(root.right, updatedConstraints);
}
```

Pattern 3: Tree Search

```
ResultType SearchTree(TreeNode root, target) {
    if (root == null) return defaultResult;

    if (found(root, target)) return result;

    // Search in appropriate subtree(s)
    ResultType leftResult = SearchTree(root.left, target);
    if (leftResult != defaultResult) return leftResult;

    return SearchTree(root.right, target);
}
```

Review Tips

- **Visualize trees**: Draw tree diagrams while solving problems
- **Master traversals**: Practice all DFS and BFS variations until automatic
- **Understand BST property**: Know when and how to exploit ordering
- **Handle null cases**: Always consider empty trees and null nodes

- **Space complexity awareness**: Understand call stack vs explicit data structure trade-offs

How to Practice Mock Interviews

1. **Problem Classification** (3 min): Identify tree type (general vs BST) and required operation
2. **Approach Discussion** (7 min): Choose DFS vs BFS, explain traversal strategy
3. **Implementation** (25 min): Code while explaining tree navigation logic
4. **Optimization** (5 min): Discuss early termination, space improvements
5. **Testing** (10 min): Walk through examples, test edge cases (empty, single node, unbalanced)

Common Mistakes to Avoid

- **Null pointer exceptions**: Always check `node != null` before accessing properties
- **Stack overflow**: Consider iterative solutions for very deep trees
- **Incorrect base cases**: Handle empty trees and leaf nodes properly
- **BST property violations**: Don't assume local validity implies global validity
- **Space complexity oversight**: Understand hidden space costs of recursion
- **Early termination missed**: Look for opportunities to stop traversal early

Advanced Topics for Further Study

- **Balanced BST**: AVL trees, Red-Black trees, tree rotations
- **Tree serialization**: Converting trees to strings and back
- **Lowest Common Ancestor**: General tree LCA algorithms
- **Tree diameter**: Finding longest path between any two nodes
- **Morris traversal**: In-order traversal with O(1) space
- **Threaded binary trees**: Adding threading for efficient traversal

Real-world Applications

- **File Systems**: Directory structures as trees
- **Databases**: B-trees for indexing, query optimization
- **Compilers**: Abstract syntax trees for parsing
- **Machine Learning**: Decision trees, random forests
- **Computer Graphics**: Scene graphs, BSP trees
- **Networking**: Spanning tree protocol, routing trees
- **AI**: Game trees for minimax algorithms
- **Data Compression**: Huffman coding trees

Tree Interview Preparation Checklist

- [ ] Master all traversal algorithms (recursive and iterative)
- [ ] Understand BST property and its applications
- [ ] Practice tree modification problems (inversion, pruning)
- [ ] Know tree validation techniques
- [ ] Understand space complexity of different approaches
- [ ] Practice drawing and visualizing tree problems
- [ ] Master edge case handling (null, single node, unbalanced)
- [ ] Know when to use DFS vs BFS
- [ ] Understand parent-child relationship navigation
- [ ] Practice level-order and reverse-level-order problems

This completes Day 5: Trees (Part 2) with comprehensive coverage of all 8 problems, following the same detailed structure and educational approach as previous days.