```
    // Main method to test the operations
    public static void main(String[] args) {
        BinaryTree tree =
```

re write and also add complexities

Let's go through the tree concepts again with examples first, followed by the full Java implementation and time complexities for each method.

1. Binary Tree

A binary tree is a hierarchical data structure where each node has at most two children.

Example:

```markdown
      10
     /  \
    5    20
   / \   / \
  3   7 15  25
```

Here:

- Root: `10`
- Left child of `10`: `5`
- Right child of `10`: `20`
- Each child can have up to 2 more children.

2. Binary Search Tree (BST)

A Binary Search Tree (BST) is a binary tree with the following properties:

- Left subtree nodes have values smaller than the root.
- Right subtree nodes have values larger than the root.

Example:

Insert numbers into a BST: `20, 10, 30, 5, 15, 25, 35`.

```markdown
      20
     /  \
   10    30
```

```
      /  \   /  \
     5   15 25  35
```

- `10` is less than `20`, so it goes to the left.
- `30` is greater than `20`, so it goes to the right.
- Insert further values by following this rule.

3. Tree Traversals

a. In-order Traversal (Left, Root, Right)

Example using the tree:

```markdown
        10
       /  \
      5    20
     / \   / \
    3   7 15  25
```

Steps:

1. Start at `10`, go to the left subtree (`5`).
2. Go to the left subtree of `5` (`3`), print `3`.
3. Return to `5`, print `5`.
4. Visit right subtree of `5` (`7`), print `7`.
5. Return to `10`, print `10`.
6. Visit the right subtree of `10` (`20`).
7. Visit left subtree of `20` (`15`), print `15`.
8. Return to `20`, print `20`.
9. Visit right subtree of `20` (`25`), print `25`.

Output: `3 5 7 10 15 20 25`
Time Complexity: `O(n)` where `n` is the number of nodes.

b. Pre-order Traversal (Root, Left, Right)

Steps:

1. Start at `10`, print `10`.
2. Visit the left subtree (`5`), print `5`.
3. Visit the left subtree of `5` (`3`), print `3`.
4. Return to `5`, visit right subtree (`7`), print `7`.
5. Return to `10`, visit the right subtree (`20`), print `20`.
6. Visit the left subtree of `20` (`15`), print `15`.
7. Return to `20`, visit the right subtree (`25`), print `25`.

Output: `10 5 3 7 20 15 25`
Time Complexity: `O(n)`

c. Post-order Traversal (Left, Right, Root)

Steps:

1. Start at `10`, go to the left subtree (`5`).
2. Visit the left subtree of `5` (`3`), print `3`.
3. Return to `5`, visit right subtree (`7`), print `7`.
4. Print `5`.
5. Go to the right subtree of `10` (`20`).
6. Visit the left subtree of `20` (`15`), print `15`.
7. Visit the right subtree of `20` (`25`), print `25`.
8. Print `20`, then print root `10`.

Output: `3 7 5 15 25 20 10`
Time Complexity: `O(n)`

d. Level-order Traversal (Breadth-first)

Steps:

1. Print root `10`.
2. Print level 2: `5 20`.
3. Print level 3: `3 7 15 25`.

Output: `10 5 20 3 7 15 25`
Time Complexity: `O(n)`

4. Insertion and Deletion in BST

a. Insertion

Inserting a node into a BST is done by following the rule:

- If the value is smaller than the root, insert into the left subtree.
- If the value is larger, insert into the right subtree.

Example: Insert `12` into the BST.
Time Complexity: Average case `O(log n)`, worst case `O(n)` if the tree is unbalanced.

b. Deletion

Three cases arise during deletion:

1. **No child**: Remove the node directly.
2. **One child**: Replace the node with its child.
3. **Two children**: Replace the node with its in-order successor or predecessor.

Example: Delete `15` from the BST.
Time Complexity: Average case `O(log n)`, worst case `O(n)`.

5. Lowest Common Ancestor (LCA)

The LCA of two nodes is the lowest node that is an ancestor of both.

Example:

Find the LCA of `7` and `25`:

```markdown
        10
       /  \
      5   20
     / \   / \
    3  7  15  25
```

The LCA is `10` because it's the lowest node that is an ancestor of both `7` and `25`.
Time Complexity: `O(h)` where `h` is the height of the tree.

6. Tree Height / Depth

Recursive Approach:

The height of a binary tree is the longest path from the root to a leaf node.
Time Complexity: `O(n)`

Iterative Approach:

Using level-order traversal (BFS), we can calculate height by counting levels.
Time Complexity: `O(n)`

Java Implementation with Time Complexities

```java
class TreeNode {
    int value;
    TreeNode left, right;

    public TreeNode(int value) {
        this.value = value;
        left = right = null;
    }
}

public class BinaryTree {

    // Insert a node in Binary Search Tree (BST)
    public TreeNode insert(TreeNode root, int value) {
        if (root == null) {
            root = new TreeNode(value);
            return root;
        }
        if (value < root.value) {
            root.left = insert(root.left, value);
        } else if (value > root.value) {
            root.right = insert(root.right, value);
        }
        return root;
    }
```

```
    // In-order Traversal
    public void inOrderTraversal(TreeNode root) {
        if (root != null) {
            inOrderTraversal(root.left);
            System.out.print(root.value + " ");
            inOrderTraversal(root.right);
        }
    }

    // Pre-order Traversal
    public void preOrderTraversal(TreeNode root) {
        if (root != null) {
            System.out.print(root.value + " ");
            preOrderTraversal(root.left);
            preOrderTraversal(root.right);
        }
    }

    // Post-order Traversal
    public void postOrderTraversal(TreeNode root) {
        if (root != null) {
            postOrderTraversal(root.left);
            postOrderTraversal(root.right);
            System.out.print(root.value + " ");
        }
    }

    // Level-order Traversal
    public void levelOrderTraversal(TreeNode root) {
        if (root == null) return;

        java.util.Queue<TreeNode> queue = new java.util.LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            TreeNode current = queue.poll();
            System.out.print(current.value + " ");

            if (current.left != null) queue.add(current.left);
            if (current.right != null) queue.add(current.right);
        }
    }

    // Finding Lowest Common Ancestor (LCA)
    public TreeNode findLCA(TreeNode root, int n1, int n2) {
        if (root == null) return null;

        if (root.value > n1 && root.value > n2)
            return findLCA(root.left, n1, n2);

        if (root.value < n1 && root.value < n2)
            return findLCA(root.right, n1, n2);

        return root;
    }

    // Calculate the height (recursive)
    public int calculateHeight(TreeNode root) {
        if (root == null) return 0;
        int leftHeight = calculateHeight(root.left);
        int rightHeight = calculateHeight(root.right);
        return Math.max(leftHeight, rightHeight) + 1;
    }

    // Main method to test the operations
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        TreeNode root = null;

        // Insertion
        root = tree.insert(root, 10);
        root = tree.insert(root, 5);
        root = tree.insert(root, 20);
        root = tree.insert(root, 3);
```

```
        root = tree.insert(root, 7);
        root = tree.insert(root, 15);
        root = tree.insert(root, 25);

        // Traversals
        System.out.print("In-order Traversal: ");
        tree.inOrderTraversal(root);
        System.out.println();

        System.out.print("Pre-order Traversal: ");
        tree.preOrderTraversal(root);
        System.out.println();

        System.out.print("Post-order Traversal: ");
        tree.postOrderTraversal(root);
        System.out.println();

        System.out.print("Level-order Traversal: ");
        tree.levelOrderTraversal(root);
        System.out.println();

        // LCA
        TreeNode lca = tree.findLCA(root, 7, 25);
        System.out.println("LCA of 7 and 25: " + lca.value);

        // Height
        System.out.println("Height of the tree: " + tree.calculateHeight(root));
    }
}
```

Time Complexity Summary:

1. **Insert**: `O(log n)` on average, `O(n)` worst case.
2. **Traversals**: `O(n)` for all (in-order, pre-order, post-order, level-order).
3. **Lowest Common Ancestor (LCA)**: `O(h)` where `h` is the height of the tree.
4. **Tree Height**: `O(n)`.

This code demonstrates various operations on a binary search tree, including inserting nodes, performing different types of traversals, finding the lowest common ancestor, and calculating the tree's height.

Now give me 10 most asked coding question on trees and give code as well. First explain the problem statements using example then write code. Also explain about complexity if required. And give multiple solution if more ways can be important to know.

Memory updated

Here are 10 of the most commonly asked coding questions on trees. I'll explain each problem with an example, provide the code, and discuss time complexity. Where multiple solutions exist, I'll mention them as well.

1. Maximum Depth (or Height) of Binary Tree

Problem Statement:

Given a binary tree, find its maximum depth. The maximum depth is the number of nodes along the longest path from the root to the farthest leaf node.

Example:

```markdown
      3
     / \
    9   20
       /  \
      15   7
```

Output: `3` (as the longest path is `3 -> 20 -> 7`).

Solution 1: Recursive Approach

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

Time Complexity: `O(n)`, where `n` is the number of nodes in the tree.

Solution 2: Iterative Approach (Using BFS)

```java
public int maxDepth(TreeNode root) {
    if (root == null) return 0;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    int depth = 0;

    while (!queue.isEmpty()) {
        int levelSize = queue.size();
        for (int i = 0; i < levelSize; i++) {
            TreeNode current = queue.poll();
            if (current.left != null) queue.add(current.left);
            if (current.right != null) queue.add(current.right);
        }
        depth++;
    }
    return depth;
}
```

Time Complexity: `O(n)`

---

2. Check if Binary Tree is Symmetric

Problem Statement:

Given a binary tree, check if it's symmetric around its center.

Example:

```markdown
      1
     / \
    2   2
   / \ / \
  3  4 4  3
```

Output: `true` (The tree is a mirror of itself.)

Solution: Recursive Approach

```java
public boolean isSymmetric(TreeNode root) {
    return root == null || isMirror(root.left, root.right);
}

private boolean isMirror(TreeNode left, TreeNode right) {
    if (left == null || right == null) return left == right;
    if (left.val != right.val) return false;
    return isMirror(left.left, right.right) && isMirror(left.right, right.left);
}
```

Time Complexity: `O(n)`

---

3. Validate Binary Search Tree (BST)

Problem Statement:

Given a binary tree, determine if it's a valid BST.

Example:

```markdown
      2
     / \
    1   3
```

Output: `true` (This is a valid BST.)

Solution: Recursive Approach

```java
public boolean isValidBST(TreeNode root) {
    return validate(root, null, null);
}

private boolean validate(TreeNode node, Integer low, Integer high) {
    if (node == null) return true;
    if ((low != null && node.val <= low) || (high != null && node.val >= high)) return
false;
    return validate(node.left, low, node.val) && validate(node.right, node.val, high);
}
```

Time Complexity: `O(n)`

---

4. Lowest Common Ancestor (LCA) in a Binary Tree

Problem Statement:

Given a binary tree, find the lowest common ancestor of two given nodes.

Example:

```markdown
      3
     / \
    5   1
   / \ / \
  6  2 0  8
    / \
   7   4
```

For nodes `5` and `1`, the LCA is `3`.

Solution: Recursive Approach

```java
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || root == p || root == q) return root;

    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);

    if (left != null && right != null) return root;
    return left != null ? left : right;
}
```

Time Complexity: `O(n)`

---

5. In-order Successor in BST

Problem Statement:

Given a BST and a node `p`, find its in-order successor (the next larger node in in-order traversal).

Example:

```markdown
      20
     /  \
    10   30
      \
       15
```

For node `15`, the in-order successor is `20`.

Solution: Iterative Approach

```java
public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
    TreeNode successor = null;

    while (root != null) {
        if (p.val < root.val) {
            successor = root;
            root = root.left;
        } else {
            root = root.right;
        }
    }
    return successor;
}
```

Time Complexity: `O(h)` where `h` is the height of the tree.

---

6. Binary Tree Level Order Traversal

Problem Statement:

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

Example:

```markdown
      3
     / \
    9  20
      /  \
     15   7
```

Output: `[[3], [9, 20], [15, 7]]`

Solution: BFS Approach

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

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

    while (!queue.isEmpty()) {
        int levelSize = queue.size();
        List<Integer> currentLevel = new ArrayList<>();

        for (int i = 0; i < levelSize; i++) {
            TreeNode current = queue.poll();
            currentLevel.add(current.val);
            if (current.left != null) queue.add(current.left);
            if (current.right != null) queue.add(current.right);
        }
        result.add(currentLevel);
    }
    return result;
}
```

Time Complexity: `O(n)`

---

7. Construct Binary Tree from Preorder and Inorder Traversal

Problem Statement:

Given two arrays `preorder` and `inorder`, construct the binary tree.

Example:

```makefile
Preorder: [3, 9, 20, 15, 7]
Inorder:  [9, 3, 15, 20, 7]
```

Solution: Recursive Approach

```java
public TreeNode buildTree(int[] preorder, int[] inorder) {
    return helper(0, 0, inorder.length - 1, preorder, inorder);
}

private TreeNode helper(int preStart, int inStart, int inEnd, int[] preorder, int[]
inorder) {
    if (preStart > preorder.length - 1 || inStart > inEnd) return null;

    TreeNode root = new TreeNode(preorder[preStart]);
```

```
        int inIndex = 0;
        for (int i = inStart; i <= inEnd; i++) {
            if (inorder[i] == root.val) {
                inIndex = i;
                break;
            }
        }

        root.left = helper(preStart + 1, inStart, inIndex - 1, preorder, inorder);
        root.right = helper(preStart + inIndex - inStart + 1, inIndex + 1, inEnd, preorder,
    inorder);

        return root;
    }
```

Time Complexity: `O(n)`

---

8. Path Sum

Problem Statement:

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

Example:

```markdown
          5
         / \
        4   8
       /   / \
      11  13  4
     /  \      \
    7    2      1
```

For sum `22`, the path `5 -> 4 -> 11 -> 2` exists.

Solution: Recursive Approach

```java
public boolean hasPathSum(TreeNode root, int sum) {
    if (root == null) return false;
    if (root.left == null && root.right == null && sum == root.val) return true;
    return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum -
root.val);
}
```

Time Complexity: `O(n)`

---

9. Count Complete Tree Nodes

Problem Statement:

Given a complete binary tree, count the number of nodes.

Solution: Recursive Approach

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

Time Complexity: `O(n)`

---

10. **Binary Tree Zig

print code 10

10. Binary Tree Zigzag Level Order Traversal

Problem Statement:

Given a binary tree, return its zigzag level order traversal. The nodes' values should alternate between left-to-right and right-to-left at each level.

Example:

```markdown
    3
   / \
  9  20
    /  \
   15   7
```

Output: `[[3], [20, 9], [15, 7]]`

Solution: BFS Approach Using a Flag

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

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

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            List<Integer> currentLevel = new ArrayList<>(levelSize);

            for (int i = 0; i < levelSize; i++) {
                TreeNode current = queue.poll();
                if (leftToRight) {
                    currentLevel.add(current.val);
                } else {
                    currentLevel.add(0, current.val);
                }

                if (current.left != null) queue.add(current.left);
                if (current.right != null) queue.add(current.right);
            }

            result.add(currentLevel);
            leftToRight = !leftToRight;  // Toggle the direction
        }
        return result;
    }
```

Explanation:

1. A queue is used to perform level order traversal.
2. A flag `leftToRight` toggles after every level to switch between left-to-right and right-to-left.
3. When `leftToRight` is false, elements are added to the front of the list (using `add(0, element)`).

Time Complexity:

- **Time Complexity:** `O(n)` where `n` is the number of nodes.
- **Space Complexity:** `O(n)` due to the usage of the queue for storing nodes at each level.

This solution performs a breadth-first traversal while reversing the order at every level, as required for the zigzag pattern.

Check if if missed any one of the below and add these question as well with complexity if you missed.

Same Tree
Binary Tree Maximum Path Sum
Binary Tree Level Order Traversal
Lowest Common Ancestor of a Binary Tree
Binary Tree Right Side View
Subtree of Another Tree
Construct Binary Tree from Preorder and Inorder Traversal
Serialize and Deserialize Binary Tree

Validate Binary Search Tree
Kth Smallest Element in a BST

It looks like I missed a few questions from your list. Here's the additional information for those questions, including explanations, code, and complexities.

1. Same Tree

Problem Statement:

Given two binary trees, 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 value.

Example:

```markdown
Tree 1:
    1
   / \
  2   3

Tree 2:
    1
   / \
  2   3
```

Output: `` `true` `` (The trees are identical.)

Solution:

```java
public boolean isSameTree(TreeNode p, TreeNode q) {
    if (p == null && q == null) return true;
    if (p == null || q == null) return false;
    return (p.val == q.val) && isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}
```

Time Complexity: `` `O(n)` ``, where `` `n` `` is the number of nodes in the tree.

---

2. Binary Tree Maximum Path Sum

Problem Statement:

Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

Example:

```markdown
        10
       /  \
      2   10
     / \    \
    20  1   -25
            /  \
           3    4
```

Output: `42` (The path with maximum sum is `20 -> 2 -> 10 -> 10`.)

Solution:

```java
public int maxPathSum(TreeNode root) {
    int[] maxSum = new int[] { Integer.MIN_VALUE };
    maxPathSumHelper(root, maxSum);
    return maxSum[0];
}

private int maxPathSumHelper(TreeNode node, int[] maxSum) {
    if (node == null) return 0;

    int left = Math.max(0, maxPathSumHelper(node.left, maxSum));
    int right = Math.max(0, maxPathSumHelper(node.right, maxSum));

    maxSum[0] = Math.max(maxSum[0], left + right + node.val);

    return Math.max(left, right) + node.val;
}
```

Time Complexity: `O(n)`, where `n` is the number of nodes.

---

3. Binary Tree Right Side View

Problem Statement:

Given a binary tree, return the right side view of its nodes' values, i.e., the values of the nodes you can see when looking at the tree from the right side.

Example:

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

Output: `[1, 3, 5, 4]`

Solution:

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

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

    while (!queue.isEmpty()) {
        int levelSize = queue.size();
        for (int i = 0; i < levelSize; i++) {
            TreeNode current = queue.poll();
            if (i == levelSize - 1) result.add(current.val);
            if (current.left != null) queue.add(current.left);
            if (current.right != null) queue.add(current.right);
        }
    }
    return result;
}
```

Time Complexity: `O(n)`

---

4. Subtree of Another Tree

Problem Statement:

Given two binary trees, write a function to check if `tree2` is a subtree of `tree1`.

Example:

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

Tree 2:
      4
     / \
    1   2
```

Output: `true`

Solution:

```java
public boolean isSubtree(TreeNode s, TreeNode t) {
    if (s == null) return false;
    if (isSameTree(s, t)) return true;
```

```
        return isSubtree(s.left, t) || isSubtree(s.right, t);
    }
```

Time Complexity: `O(m * n)`, where `m` and `n` are the number of nodes in `s` and `t`, respectively.

---

5. Serialize and Deserialize Binary Tree

Problem Statement:

Design an algorithm to serialize and deserialize a binary tree. Serialization is the process of converting a tree into a string, and deserialization is the process of converting the string back into a tree.

Example:

- **Serialize:** `1,2,3,null,null,4,5`
- **Deserialize:** Reconstruct the tree from the serialized string.

Solution:

```java
    // Serialize
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        serializeHelper(root, sb);
        return sb.toString();
    }

    private void serializeHelper(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append("null,");
            return;
        }
        sb.append(node.val).append(",");
        serializeHelper(node.left, sb);
        serializeHelper(node.right, sb);
    }

    // Deserialize
    public TreeNode deserialize(String data) {
        String[] nodes = data.split(",");
        Queue<String> queue = new LinkedList<>(Arrays.asList(nodes));
        return deserializeHelper(queue);
    }

    private TreeNode deserializeHelper(Queue<String> queue) {
        String val = queue.poll();
        if (val.equals("null")) return null;
        TreeNode node = new TreeNode(Integer.parseInt(val));
        node.left = deserializeHelper(queue);
        node.right = deserializeHelper(queue);
        return node;
    }
```

Time Complexity: `O(n)`, where `n` is the number of nodes in the tree.

---

6. Validate Binary Search Tree (BST)

Problem Statement:

Validate if a binary tree is a BST.

Solution:

(Already covered in previous responses.)
Time Complexity: `O(n)`

---

7. Kth Smallest Element in a BST

Problem Statement:

Given a BST, find the k-th smallest element in the BST.

Example:

```markdown
      3
     / \
    1   4
     \
      2
```

For `k = 1`, the output is `1`.

Solution:

```java
public int kthSmallest(TreeNode root, int k) {
    Stack<TreeNode> stack = new Stack<>();
    TreeNode curr = root;

    while (curr != null || !stack.isEmpty()) {
        while (curr != null) {
            stack.push(curr);
            curr = curr.left;
        }

        curr = stack.pop();
        k--;
        if (k == 0) return curr.val;
        curr = curr.right;
    }
    return -1; // Should never reach here
}
```

Time Complexity: `O(h + k)`, where `h` is the height of the tree.

Each of these questions covers various tree-related concepts and operations, providing a comprehensive understanding of binary trees and binary search trees.

Now give list of most commonly asked question on trees in interviews. These have to be non coding questions.

Here's a list of some commonly asked non-coding questions on trees in interviews. These questions focus on concepts, definitions, and theoretical understanding rather than coding:

1. Explain the Different Types of Trees

- **Binary Tree:** A tree where each node has at most two children.
- **Binary Search Tree (BST):** A binary tree where the left child is less than the parent node, and the right child is greater.
- **AVL Tree:** A self-balancing binary search tree where the difference between heights of left and right subtrees cannot be more than one.
- **Red-Black Tree:** A self-balancing binary search tree where each node has an additional color attribute to ensure balancing.
- **B-Tree:** A balanced tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions.
- **Heap:** A specialized tree-based structure that satisfies the heap property (either max-heap or min-heap).

2. What is the Difference Between Binary Tree and Binary Search Tree?

- **Binary Tree:** A tree where each node can have up to two children but does not require any specific order.
- **Binary Search Tree:** A binary tree with the additional property that for any node, all nodes in the left subtree are less than the node, and all nodes in the right subtree are greater.

3. What is Tree Traversal and What Are the Different Types?

- **Tree Traversal:** The process of visiting each node in a tree data structure in a systematic manner.
  - **In-order Traversal:** Visit the left subtree, the node, and then the right subtree.
  - **Pre-order Traversal:** Visit the node, the left subtree, and then the right subtree.
  - **Post-order Traversal:** Visit the left subtree, the right subtree, and then the node.
  - **Level-order Traversal:** Visit nodes level by level from top to bottom.

4. What is the Difference Between Depth and Height of a Tree?

- **Depth (or Level):** The number of edges from the root node to a particular node.
- **Height:** The number of edges on the longest path from a node to a leaf node. The height of the tree is the height of the root node.

5. What is the Purpose of a Balanced Tree?

- **Balanced Tree:** A tree structure that maintains balance to ensure that operations such as insertion, deletion, and search can be performed in logarithmic time complexity. It prevents skewed structures that degrade performance.

6. What is the Height of a Perfectly Balanced Binary Tree?

- In a perfectly balanced binary tree, the height `h` is `O(log n)`, where `n` is the number of nodes. The tree is balanced such that every level is fully filled except possibly the last level.

7. Explain the Concept of Tree Rotation in AVL Trees.

- **Tree Rotation:** A technique used to maintain the balance of an AVL tree. Rotations are performed to correct imbalances caused by insertions or deletions.
  - **Left Rotation:** A rotation performed when a right-heavy subtree needs balancing.
  - **Right Rotation:** A rotation performed when a left-heavy subtree needs balancing.

8. What is the Difference Between Binary Tree and Binary Heap?

- **Binary Tree:** A general tree structure with at most two children per node, with no specific order requirement.
- **Binary Heap:** A complete binary tree that maintains a specific heap property (either max-heap or min-heap), where each parent node satisfies the heap property with respect to its children.

9. What Are the Advantages of Using a Binary Search Tree (BST)?

- **Advantages:**
  - Provides efficient in-order traversal.
  - Offers average-case time complexity of `O(log n)` for search, insertion, and deletion.
  - Allows dynamic set operations and fast lookup.

10. What Is a Threaded Binary Tree?

- **Threaded Binary Tree:** A binary tree where null pointers (pointers to non-existing children) are used to point to the next node in the in-order traversal. This helps in efficient in-order traversal without using additional stack or recursion.

These questions help evaluate a candidate's understanding of tree data structures and their properties, which are fundamental concepts in computer science.