# Tree Data Structure

|  |  |
|---|---|
| Created by | j jubink |
| Created time | @February 26, 2025 10:08 PM |
| Tags |  |

## Binary Tree Practice Questions with Solutions

### 1. Find the Depth of a Binary Tree

#### Algorithm:

1. If the tree is empty, return 0.
2. Compute the depth of the left and right subtrees recursively.
3. The depth of the tree is the maximum of the left and right subtree depths plus 1.

#### Python Code:

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def find_depth(root):
    if not root:
        return 0
    left_depth = find_depth(root.left)
    right_depth = find_depth(root.right)
    return max(left_depth, right_depth) + 1
```

```
# Example Usage
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print("Depth of the tree:", find_depth(root))
```

## 2. Count the Number of Nodes in a Binary Tree

### Algorithm:

1. If the tree is empty, return 0.
2. Recursively count the nodes in the left and right subtrees.
3. Return the sum of left and right subtree counts plus 1 for the root node.

### Python Code:

```
def count_nodes(root):
    if not root:
        return 0
    return 1 + count_nodes(root.left) + count_nodes(root.right)

# Example Usage
print("Number of nodes in the tree:", count_nodes(root))
```

## 3. Find the Maximum Value in a Binary Tree

### Algorithm:

1. If the tree is empty, return negative infinity.
2. Find the maximum value in the left and right subtrees recursively.
3. Return the maximum of root, left max, and right max.

### Python Code:

```
def find_max(root):
    if not root:
        return float('-inf')
    left_max = find_max(root.left)
    right_max = find_max(root.right)
    return max(root.key, left_max, right_max)

print("Maximum value in the tree:", find_max(root))
```

## 4. Perform Level Order Traversal (BFS)

### Algorithm:

1. Use a queue to traverse the tree level by level.
2. Print each node and enqueue its left and right children.

### Python Code:

```
from collections import deque

def level_order_traversal(root):
    if not root:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.key, end=" ")
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

```
print("Level order traversal:")
level_order_traversal(root)
```

## 5. Perform Inorder, Preorder, and Postorder Traversals

### Python Code:

```
def inorder(root):
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

def preorder(root):
    if root:
        print(root.key, end=" ")
        preorder(root.left)
        preorder(root.right)

def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.key, end=" ")

print("Inorder traversal:")
inorder(root)
print("\nPreorder traversal:")
preorder(root)
print("\nPostorder traversal:")
postorder(root)
```

---

This section covers **Binary Tree** operations. Next, I will add **Binary Search Tree (BST) questions with solutions.** 🚀

## 🌳 Basic Level Questions

### 1. Find the Height of a Binary Tree

**Algorithm:**

1. If the tree is empty, return 0.
2. Recursively find the height of the left and right subtrees.
3. Return `1 + max(left_height, right_height)`.

```
python
CopyEdit
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def find_height(root):
    if root is None:
        return 0
    return 1 + max(find_height(root.left), find_height(root.right))

# Example Usage
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
print(find_height(root))  # Output: 3
```

---

### 2. Find the Depth of a Specific Node

**Algorithm:**

1. Start from the root with depth `0`.

2. Traverse left or right recursively.
3. If the node is found, return the depth.

```
python
CopyEdit
def find_depth(root, key, depth=0):
    if root is None:
        return -1
    if root.key == key:
        return depth
    left_depth = find_depth(root.left, key, depth + 1)
    if left_depth != -1:
        return left_depth
    return find_depth(root.right, key, depth + 1)

print(find_depth(root, 4))  # Output: 2
```

---

## 3. Find the Total Number of Nodes in a Binary Tree

**Algorithm:**

1. If tree is empty, return 0.
2. Recursively count left and right subtree nodes.
3. Return `1 + left_count + right_count`.

```
python
CopyEdit
def count_nodes(root):
    if root is None:
        return 0
    return 1 + count_nodes(root.left) + count_nodes(root.right)
```

```
print(count_nodes(root))  # Output: 4
```

---

## 4. Find the Total Number of Leaf Nodes

**Algorithm:**

1. If a node has no children, it is a leaf.
2. Recursively count leaf nodes in left and right subtrees.

```
python
CopyEdit
def count_leaf_nodes(root):
    if root is None:
        return 0
    if root.left is None and root.right is None:
        return 1
    return count_leaf_nodes(root.left) + count_leaf_nodes(root.right)

print(count_leaf_nodes(root))  # Output: 2
```

---

## 5. Perform Preorder, Inorder, and Postorder Traversals

**Algorithm:**

- **Preorder:** Visit root → Left → Right
- **Inorder:** Left → Root → Right
- **Postorder:** Left → Right → Root

```
python
CopyEdit
def preorder(root):
    if root:
        print(root.key, end=" ")
        preorder(root.left)
        preorder(root.right)

def inorder(root):
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.key, end=" ")

preorder(root)   # Output: 1 2 4 3
inorder(root)    # Output: 4 2 1 3
postorder(root)  # Output: 4 2 3 1
```

---

## 6. Find the Maximum and Minimum Values in a Binary Tree

**Algorithm:**

1. If tree is empty, return `None`.
2. Recursively find the max and min in left and right subtrees.

```
python
CopyEdit
def find_max(root):
    if root is None:
        return float('-inf')
    return max(root.key, find_max(root.left), find_max(root.right))

def find_min(root):
    if root is None:
        return float('inf')
    return min(root.key, find_min(root.left), find_min(root.right))

print(find_max(root))  # Output: 4
print(find_min(root))  # Output: 1
```

## 7. Find the Sum of All Nodes

**Algorithm:**

1. If tree is empty, return 0.
2. Recursively sum up left and right subtrees.

```
python
CopyEdit
def sum_nodes(root):
    if root is None:
        return 0
    return root.key + sum_nodes(root.left) + sum_nodes(root.right)

print(sum_nodes(root))  # Output: 10
```

## 8. Count the Number of Full Nodes

**Algorithm:**

- A full node has **both left and right children**.

```
python
CopyEdit
def count_full_nodes(root):
    if root is None:
        return 0
    count = count_full_nodes(root.left) + count_full_nodes(root.right)
    if root.left and root.right:
        count += 1
    return count

print(count_full_nodes(root))  # Output: 1
```

## 9. Count the Number of Half Nodes

**Algorithm:**

- A half node has **only one child**.

```
python
CopyEdit
def count_half_nodes(root):
    if root is None:
        return 0
    count = count_half_nodes(root.left) + count_half_nodes(root.right)
    if (root.left and not root.right) or (root.right and not root.left):
        count += 1
    return count

print(count_half_nodes(root))  # Output: 2
```

## 10. Check if Two Binary Trees are Identical

**Algorithm:**

1. If both are `None`, return `True`.
2. If one is `None`, return `False`.
3. Compare root values and recursively check left and right subtrees.

```
python
CopyEdit
def are_identical(root1, root2):
    if root1 is None and root2 is None:
        return True
    if root1 is None or root2 is None:
        return False
    return (root1.key == root2.key and
            are_identical(root1.left, root2.left) and
            are_identical(root1.right, root2.right))

root2 = Node(1)
root2.left = Node(2)
root2.right = Node(3)
root2.left.left = Node(4)

print(are_identical(root, root2))  # Output: True
```

## 🌳 Intermediate Level Binary Tree Questions with Solutions

These questions focus on **tree properties, modifications, and advanced operations.** Each problem includes an **algorithm** and a **Python implementation.**

---

### 1️⃣ Find the Diameter of a Binary Tree

> Diameter is the longest path between any two nodes in the tree. It can go through or not through the root.

#### Algorithm:

1. Recursively calculate the height of left and right subtrees.
2. Compute the diameter at each node: `left_height + right_height + 1`.

3. Keep track of the maximum diameter while traversing the tree.

```
python
CopyEdit
class TreeInfo:
    def __init__(self, height=0, diameter=0):
        self.height = height
        self.diameter = diameter

def find_diameter(root):
    def helper(node):
        if node is None:
            return TreeInfo(0, 0)

        left = helper(node.left)
        right = helper(node.right)

        height = 1 + max(left.height, right.height)
        diameter = max(left.diameter, right.diameter, left.height + right.height + 1)

        return TreeInfo(height, diameter)

    return helper(root).diameter

# Example Usage
print(find_diameter(root))  # Output: 4
```

---

## 2️⃣ Find the Lowest Common Ancestor (LCA) of Two Nodes

> The LCA of two nodes p and q is the lowest node in the tree that has both p and q as descendants.

### Algorithm:

1. If the root is `None` or matches `p` or `q`, return root.
2. Recursively find LCA in the left and right subtrees.
3. If **both subtrees return non-null values**, the current node is LCA.

```
python
CopyEdit
def find_lca(root, p, q):
    if root is None or root.key == p or root.key == q:
        return root

    left = find_lca(root.left, p, q)
    right = find_lca(root.right, p, q)

    if left and right:
        return root
    return left if left else right

print(find_lca(root, 2, 4).key)  # Output: 2
```

---

## 3️⃣ Check if a Binary Tree is a Full Binary Tree

> A full binary tree is where every node has 0 or 2 children.

### Algorithm:

1. If the node is `None`, return `True`.
2. If the node has **only one child**, return `False`.
3. Recursively check left and right subtrees.

```
python
CopyEdit
```

```
def is_full_binary_tree(root):
    if root is None:
        return True
    if (root.left is None and root.right is None):
        return True
    if (root.left and root.right):
        return is_full_binary_tree(root.left) and is_full_binary_tree(root.right)
    return False

print(is_full_binary_tree(root))  # Output: False
```

---

## 4️⃣ Check if a Binary Tree is a Complete Binary Tree

> A complete binary tree is where all levels are filled except the last, which must be filled from left to right.

### Algorithm:

1. Use **level-order traversal (BFS)** to check if all non-null nodes come before null nodes.
2. If a null node appears before all nodes are processed, return `False`.

```
python
CopyEdit
from collections import deque

def is_complete_binary_tree(root):
    if root is None:
        return True

    queue = deque([root])
    encountered_null = False
    while queue:
        node = queue.popleft()
        if node:
            if encountered_null:  # If we already found a null, it's not complete
                return False
            queue.append(node.left)
            queue.append(node.right)
        else:
            encountered_null = True  # Mark that we found a null node

    return True

print(is_complete_binary_tree(root))  # Output: True
```

## 5️⃣ Check if a Binary Tree is a Perfect Binary Tree

> A perfect binary tree is where all leaf nodes are at the same level, and every internal node has two children.

### Algorithm:

1. Compute the **depth of the leftmost leaf** (this is the expected depth).
2. Recursively check if all **leaf nodes** are at the expected depth.
3. Ensure every **internal node has exactly 2 children**.

```
python
CopyEdit
def find_depth(node):
    d = 0
    while node:
        d += 1
        node = node.left
    return d
```

```
def is_perfect_tree(root, depth, level=0):
    if root is None:
        return True
    if root.left is None and root.right is None:
        return depth == level + 1
    if root.left is None or root.right is None:
        return False
    return is_perfect_tree(root.left, depth, level + 1) and is_perfect_tree(root.righ
t, depth, level + 1)

def check_perfect_binary_tree(root):
    return is_perfect_tree(root, find_depth(root))

print(check_perfect_binary_tree(root))  # Output: False
```

---

## 6️⃣ Find the Level of a Given Node

> Level means distance from root (Root is level 0).

### Algorithm:

1. Start at the root with **level 0**.
2. Use recursion to check left and right subtrees.
3. Return the first level where the node is found.

```
python
CopyEdit
def find_level(root, key, level=0):
    if root is None:
        return -1
    if root.key == key:
        return level
    left_level = find_level(root.left, key, level + 1)
    if left_level != -1:
        return left_level
    return find_level(root.right, key, level + 1)

print(find_level(root, 4))  # Output: 2
```

---

## 7️⃣ Mirror (Invert) a Binary Tree

> Swap left and right subtrees of every node.

### Algorithm:

1. If tree is empty, return.
2. Recursively swap left and right subtrees.

```
python
CopyEdit
def mirror_tree(root):
    if root is None:
        return
    root.left, root.right = root.right, root.left
    mirror_tree(root.left)
    mirror_tree(root.right)

mirror_tree(root)  # Tree is now mirrored
```

---

## 8️⃣ Find the Sum of All Nodes at a Given Depth

> Sum all nodes at a given level k.

### Algorithm:

1. Traverse tree with **level-order traversal**.
2. Sum nodes at the given level.

```
python
CopyEdit
def sum_at_depth(root, k):
    if root is None:
        return 0
    if k == 0:
        return root.key
    return sum_at_depth(root.left, k-1) + sum_at_depth(root.right, k-1)

print(sum_at_depth(root, 2))  # Output: Sum of nodes at depth 2
```

---

## 9️⃣ Convert a Binary Tree to a Doubly Linked List

> Convert tree nodes in-order into a doubly linked list.

### Algorithm:

1. Use **in-order traversal** to link nodes like a **DLL**.
2. Maintain **prev** and **head pointers**.

```
python
CopyEdit
class TreeToDLL:
    def __init__(self):
        self.prev = None
        self.head = None

    def convert(self, root):
        if root is None:
            return
        self.convert(root.left)
        if self.prev:
            self.prev.right = root
            root.left = self.prev
        else:
            self.head = root
        self.prev = root
        self.convert(root.right)

tree_to_dll = TreeToDLL()
tree_to_dll.convert(root)
dll_head = tree_to_dll.head  # Head of Doubly Linked List
```

## 🌳 Advanced Level Binary Tree Questions with Solutions

These questions focus on **complex tree applications** like serialization, width calculation, and subtree validation. Each problem includes an **algorithm** and a **Python implementation**.

---

### 1️⃣ Convert a Binary Tree into its Mirror Tree

> A mirror tree is obtained by swapping left and right subtrees recursively.

#### Algorithm:

1. If tree is empty, return `None`.
2. Recursively swap the left and right children.

```
python
CopyEdit
def mirror_tree(root):
    if root is None:
        return None
    root.left, root.right = root.right, root.left
    mirror_tree(root.left)
    mirror_tree(root.right)
    return root

# Usage
mirror_tree(root)  # Converts tree into its mirror
```

---

## 2️⃣ Print All Root-to-Leaf Paths in a Binary Tree

> Print all paths from root to leaves as a list of values.

### Algorithm:

1. Use **DFS (Depth First Search)** traversal.
2. Maintain a **path list** while traversing.
3. When a leaf node is reached, print the path.

```
python
CopyEdit
def print_root_to_leaf_paths(root, path=[]):
    if root is None:
        return
    path.append(root.key)
    if root.left is None and root.right is None:
        print(" → ".join(map(str, path)))  # Print path
    else:
        print_root_to_leaf_paths(root.left, path.copy())
        print_root_to_leaf_paths(root.right, path.copy())

# Usage
```

```
print_root_to_leaf_paths(root)
```

---

## 3️⃣ Find the Width of a Binary Tree

> Width is the maximum number of nodes at any level in the tree.

### Algorithm:

1. Use **level-order traversal (BFS)** to count nodes at each level.
2. Track the **max width** while traversing.

```
python
CopyEdit
from collections import deque

def tree_width(root):
    if root is None:
        return 0

    queue = deque([root])
    max_width = 0

    while queue:
        level_size = len(queue)  # Nodes at this level
        max_width = max(max_width, level_size)

        for _ in range(level_size):
            node = queue.popleft()
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
    return max_width

# Usage
print(tree_width(root))  # Output: Maximum width of the tree
```

---

## 4️⃣ Serialize and Deserialize a Binary Tree

> Convert a binary tree to a string (serialization) and back to a tree (deserialization).

### Algorithm:

1. Use **preorder traversal** to serialize the tree into a string.
2. Use the same order to reconstruct the tree.

```
python
CopyEdit
def serialize(root):
    if root is None:
        return "X,"
    return str(root.key) + "," + serialize(root.left) + serialize(root.right)

def deserialize(data):
    def helper(nodes):
        val = nodes.pop(0)
        if val == "X":
            return None
        node = TreeNode(int(val))
        node.left = helper(nodes)
        node.right = helper(nodes)
        return node

    return helper(data.split(","))
```

```
# Usage
tree_string = serialize(root)
print(tree_string)  # Output: Serialized tree
new_root = deserialize(tree_string)
```

---

## 5️⃣ Check if a Given Tree is a Subtree of Another Tree

> A tree T2 is a subtree of tree T1 if there exists a node in T1 whose subtree matches T2 exactly.

### Algorithm:

1. If `T2` is `None`, return `True`.
2. If `T1` is `None`, return `False`.
3. Check if trees are **identical** at the current node.
4. Recursively check in the **left and right subtrees**.

```
python
CopyEdit
def is_identical(root1, root2):
    if not root1 and not root2:
        return True
    if not root1 or not root2 or root1.key != root2.key:
        return False
    return is_identical(root1.left, root2.left) and is_identical(root1.right, root2.rig
ht)

def is_subtree(T1, T2):
    if T2 is None:
        return True
    if T1 is None:
        return False
    if is_identical(T1, T2):
        return True
    return is_subtree(T1.left, T2) or is_subtree(T1.right, T2)

# Usage
print(is_subtree(root1, root2))  # Output: True or False
```

## 6️⃣ Find the Maximum Path Sum in a Binary Tree

> The maximum path sum is the highest sum from any node to any node (not necessarily root to leaf).

### Algorithm:

1. Use **DFS traversal** to calculate the max sum at each node.
2. Consider the max of **left & right subtrees** plus current node value.
3. Update a **global max sum variable** while traversing.

```
python
CopyEdit
class MaxPathSum:
    def __init__(self):
        self.max_sum = float('-inf')

    def find_max_path_sum(self, root):
        def helper(node):
            if not node:
                return 0
            left = max(0, helper(node.left))
            right = max(0, helper(node.right))
            self.max_sum = max(self.max_sum, left + right + node.key)
            return node.key + max(left, right)
        helper(root)
        return self.max_sum

max_path = MaxPathSum()
print(max_path.find_max_path_sum(root))  # Output: Maximum path sum
```

## 7️⃣ Find the Distance Between Two Nodes in a Binary Tree

> The distance is the number of edges between two nodes.

### Algorithm:

1. Find **LCA** of both nodes.
2. Compute **distance from LCA** to each node.
3. Add both distances.

```
python
CopyEdit
def distance_from_root(root, key, distance=0):
    if root is None:
        return -1
    if root.key == key:
        return distance
    left = distance_from_root(root.left, key, distance + 1)
    return left if left != -1 else distance_from_root(root.right, key, distance + 1)

def find_distance(root, a, b):
    lca = find_lca(root, a, b)
    d1 = distance_from_root(lca, a)
    d2 = distance_from_root(lca, b)
    return d1 + d2
```

```
# Usage
print(find_distance(root, 4, 7))  # Output: Distance between nodes
```