assignment_2

November 18, 2024

0.0.1 Paraphrase the problem in your own words.

Given a binary tree, how do we find if it contains any duplicate values? If it has one duplicate value, return it. If it has multiple duplicate values, return the closest one to the root. And if there is no duplicate value, return -1"

0.0.2 Create 1 new example that demonstrates you understand the problem. Trace/walkthrough 1 example that your partner made and explain it.

root = [1, 2, 4, 2, 3, 4, 8]

output should be 2

0.0.3 Copy the solution your partner wrote.

[ ]:
```python
# Your answer here
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def is_symmetric(root: TreeNode) -> int:
    """
    Checks if a binary tree contains duplicates. Returns the duplicate value
    closest to the root. If no duplicates exist, returns -1.

    :param root: TreeNode, the root of the binary tree
    :return: int, the duplicate value closest to the root or -1 if no␣
 ↪duplicates exist
    """
    if not root:  # Handle an empty tree
        return -1

    # Initialize queue for BFS and a set for tracking seen values
    queue = [root]
    seen = set()
```

```
    while queue:
        # Dequeue the front node
        node = queue.pop(0)

        # Check for duplicate
        if node.val in seen:
            return node.val
        seen.add(node.val)

        # Enqueue the left and right children if they exist
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    # If no duplicates were found, return -1
    return -1

# Helper function to construct binary tree from list
def create_binary_tree(values):
    if not values:
        return None
    nodes = [TreeNode(val) if val is not None else None for val in values]
    kids = nodes[::-1]
    root = kids.pop()
    for node in nodes:
        if node:
            if kids:
                node.left = kids.pop()
            if kids:
                node.right = kids.pop()
    return root
```

0.0.4 Explain why their solution works in your own words.

This code works because it finds duplicates in a binary tree by searching level by level, starting from the root. It uses a Breadth-First Search (BFS) approach, which ensures that we check nodes closest to the root first before moving deeper into the tree. This is important because the problem asks us to return the closest duplicate to the root. To keep track of values we've already seen, the code uses a set called seen. As we visit each node, we check if its value is already in seen. If it is, we know we've found a duplicate, and since BFS checks nodes in order of their distance from the root, this will be the closest duplicate. At that point, the function immediately returns the duplicate value. If the value is not in seen, we add it to the set and continue checking the node's left and right children by adding them to a queue. This queue ensures that we process nodes in the correct order. Finally, if we finish traversing the entire tree and don't find any duplicates, the function returns -1 to indicate there are no duplicates.

0.0.5 Explain the problem's time and space complexity in your own words.

The time complexity of this code is O(n), where n is the number of nodes in the binary tree. This is because we visit each node exactly once to check its value and add its children to the queue. Since we don't revisit any nodes, the total time spent is directly proportional to the number of nodes. The space complexity comes from two things: 1. Queue: The queue holds all the nodes at a particular level before moving to the next level. In the worst case, this could be half the nodes in the tree (if the tree is a complete binary tree). So, the queue takes up space proportional to O(n) in the worst case. 2. Set: The seen set stores the values of all the nodes we've visited. In the worst case, all the nodes have unique values, so the set size will also grow to O(n).

Together, the queue and set give a space complexity of O(n).

0.0.6 Critique your partner's solution, including explanation, and if there is anything that should be adjusted.

The code works well, and it's easy to understand. However, more comments on the code would be better. Also, the function's name which is is_symmetric, can be changed because it's unrelated to what the function does. A more descriptive name like find_closest_duplicate would make the purpose of the code clearer.

0.0.7 Reflection

Great coding and problem solving experience. Focusing on binary trees, I initially struggled with managing the paths during recursion, but breaking the problem into smaller steps helped. I focused on understanding depth-first search (DFS) and how it naturally handles branching structures like trees. Testing the code and validating its output gave me a better understanding of how my solution works in practice. I could see how different tree structures affected the output and whether it was anticipated. This hands-on testing solidified my understanding. Reviewing my partner's code was also a great learning experience. Their code works well, and it's well structured, making it easy to understand. However, as suggested above, there is room to improve. As a whole, a great experience. I learnt about algorithms and data structure, coding, and testing, and I tried to understand someone's thoughts.