Assignment-2:

Prepared by: Zekiye Erdem

Paraphrase the problem in your own words.

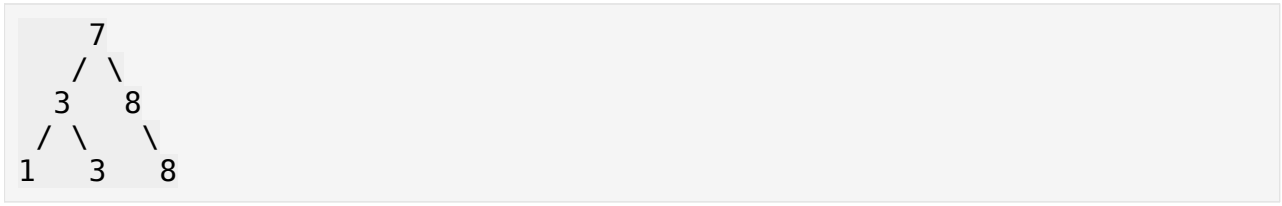
Original Question: Given the root of a binary tree, check whether it is contains a duplicate value. If a duplicate exists, return the duplicate value. If there are multiple duplicates, return the one with the closest distance to the root. If no duplicate exists, return -1.

Paraphrase Quesiton: We will examine a binary tree to check if it contains duplicate values. If a duplicate exists, return the duplicate value. If there are multiple duplicates, return the one with the closest distance to the root. If no duplicates are found, return -1.

Create 1 new example.

Example

- Input = [7, 3, 8, 1, 3, 10, 8]
- Output =[3]



Explanation:

- The root node is 7.
- The left child of 7 is 3, and the right child of 7 is 8.
- The left subtree of 3 has children 1 and 3.
- The right subtree of 8 has a child 8.
- Duplicate values:
- 3 occurs twice (closer to the root at level 2).
- 8 also occurs twice but is further from the root (level 3).

So Output should be : 3

Trace/walkthrough 1 example that your partner made and explain it.

My partner Example:

1- Here, the value 3 appears twice, with the second occurrence as the left child of the node with value 3. The expected output should be 3 since it's the first duplicate encountered closest to the root.

- Input: root = TreeNode(5, TreeNode(3, TreeNode(2), TreeNode(3)), TreeNode(7))
- Expected Output: 3

My Walkthrough:

Correspondance tree structure from my partner's example:

```
    5
   / \
  3   7
 / \
2   3
```

Steps to Check for Duplicates:

1. Start at the root node with value 5.
   - Add 5 to a set of seen values: seen = {5}.
2. Move to the left child of 5, which is 3.
   - Add 3 to the set: seen = {5, 3}.
3. Move to the left child of 3, which is 2.
   - Add 2 to the set: seen = {5, 3, 2}.
4. Move to the right child of 3, which is another 3.
   - Check if 3 is already in the set: Yes, it is. This is the first duplicate encountered closest to the root.

Output: The duplicate value closest to the root is 3.

Copy the solution your partner wrote.

```
from collections import deque

# 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_duplicate(root: TreeNode) -> int:
     if not root:
        return -1
```

```
    # Use a set to keep track of seen values
     seen = set()
    # Use a queue for BFS
     queue = deque([root])

     while queue:
        node = queue.popleft()

        # Check if the value has already been seen
        if node.val in seen:
            return node.val
        seen.add(node.val)

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

    # If no duplicates are found
     return -1
```

Explain why their solution works in your own words.

My partner's provided solution is designed to check for duplicates in a binary tree using *Breadth-First Search* (BFS).

1. Breadth-First Search Approach:

   The BFS algorithm is used here to traverse the binary tree level by level. This ensures that the nodes closer to the root are checked for duplicates before the nodes at deeper levels. Since the problem requires returning the duplicate closest to the root, BFS is the ideal choice.

2. Use of a Set to Track Seen Values:

   The **seen** set is used to keep track of all values encountered during the traversal:

   - When a node's value is encountered, it is checked against the seen set.
   - If the value is already in the set, it means a duplicate has been found.
   - If not, the value is added to the seen set.
   - This ensures efficient checking for duplicates in O(1) time for each value.

3. Queue for Level-Order Traversal:

   The deque from the collections module is used to implement the queue for BFS:

   - Nodes are added to the queue as they are discovered (i.e., their left and right children).

- Nodes are processed one at a time in the order they were added, ensuring level-order traversal.

4. Edge Case Handling:
   - If the tree is empty (root is None), the function immediately returns -1, as no duplicates can exist.
   - If no duplicates are found after traversing all nodes, the function returns -1.

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

Complexity Analysis:

Time Complexity: Each node is visited exactly once. Checking for duplicates in a set is $O(1)$ per node. Overall time complexity is **O(n)**, where n is the number of nodes.

Space Complexity: The space required for the seen set is proportional to the number of unique nodes: O(n). The queue can hold at most the number of nodes at the widest level of the tree: O(w), where w is the maximum width of the tree. Thus, the total space complexity is **O(n)**.

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

The solution efficiently detects duplicates in a binary tree using Breadth-First Search (BFS), ensuring duplicates closest to the root are found first.

A set tracks seen values, enabling constant time checks for duplicates, while a deque handles level-order traversal. This approach is time-efficient $O(n)$ and space-efficient $O(n)$ but could be further optimized for large trees.

It correctly handles edge cases like an empty tree or no duplicates but assumes all node values are hashable.

Improvements include adding comments for clarity, handling non-hashable values, and explicitly tracking node depth for future extensions.

Reflection

Please write a 200 word reflection documenting your process from assignment 1, and your presentation and review experience with your partner at the bottom of the Jupyter Notebook under a new heading "Reflection." Again, export this Notebook as pdf.

In **Assignment 1**, I developed an algorithm to identify missing values in an integer list using negative indexing. The process began with understanding the problem's requirements: to detect missing values efficiently within the constraints of an array. My approach leverages each number in the array to mark its corresponding index as negative. This method works because any index that remains positive indicates a missing value.

The choice of negative indexing was both innovative and practical. It eliminated the need for additional data structures, reducing space complexity. The algorithm's simplicity also ensures it operates in linear time, making it highly efficient for large arrays. A significant realization during this process was how well the method handles duplicates. Even if a number appears multiple times, it re-marks the same index, leaving the results unaffected and ensuring missing values are still correctly identified.

One challenge I faced was ensuring the algorithm correctly handles edge cases, such as empty arrays or numbers outside the expected range. By iterating through potential issues, I fine-tuned the implementation to handle all cases robustly. This assignment strengthened my understanding of in-place array manipulation and algorithm design, showcasing the power of leveraging array indices for efficient problem-solving.