```
(hash('Murat')%3)+1
```

```
1
```

```
# PARAPHRASE THE PROBLEM IN YOUR OWN WORDS
# Inspect a binary tree starting from its root to understand if any duplicate values are present.
# If duplicates are found, return the value of the duplicate node closest to the root.
# If there are multiple duplicates, choose the one with the shortest distance to the root.
# If no duplicates are found, return -1.


#  IN THE QUESTION 1 CONTAINING YOUR PROBLEM, THERE ARE EXAMPLES THAT ILLUSTRATE HOW THE CODE SHOULD WORK. CREATE 2 NEW EXAMPLES THAT DEMONSTI
# Question 1 Example 1: Input: root = [1, 2, 2, 3, 5, 6, 7]
# Question 1 Example 2: Input: root = [1, 10, 2, 3, 10, 12, 12]
#
# Question 1 New Example 1:  [1, 2, 2, 3, 4, 4, 3]
#      The tree is symmetric, and there are duplicate values (2 and 4). Closest duplicate to the root is 2.
#       1
#      / \
#     2   2
#    / \ / \
#   3  4 4  3
#
# Question 1 New Example 2:  [1, 2, 3, 4, 4, 3]
#      The tree is symmetric and there are dublicate values (4 and 3). Closest duplicate to the root is 4.
#       1
#      / \
#     2   3
#    / \ /
#   4  4 3


# CODE THE SOLUTION TO YOUR ASSIGNED PROBLEM IN PYTHON (CODE CHUNK). TRY TO FIND THE BEST TIME AND SPACE COMPLEXITY SOLUTION!


# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def is_symmetric(root: TreeNode) -> int:
    if not root:
        return -1

    def dfs(node, seen):
        if not node:
            return -1

        if node.val in seen:
            return node.val
        seen.add(node.val)

        left_result = dfs(node.left, seen)
        right_result = dfs(node.right, seen)

        if left_result != -1:
            return left_result
        if right_result != -1:
            return right_result

        return -1

    return dfs(root, set())
```

```
# Example 1:
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

result = is_symmetric(root)
if result != -1:
    print("The closest duplicate value to the root is:", result)
else:
    print("No duplicate values found.")
```

```
 The closest duplicate value to the root is: 2
```

```
# Example 2:
root = TreeNode(1)
root.left = TreeNode(10)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(10)
root.right.left = TreeNode(12)
root.right.right = TreeNode(12)

result = is_symmetric(root)
if result != -1:
    print("The closest duplicate value to the root is:", result)
else:
    print("No duplicate values found.")
```

```
 The closest duplicate value to the root is: 10
```

```
# New Example 1:
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(4)
root.right.left = TreeNode(4)
root.right.right = TreeNode(3)

result = is_symmetric(root)
if result != -1:
    print("The closest duplicate value to the root is:", result)
else:
    print("No duplicate values found.")
```

```
 The closest duplicate value to the root is: 2
```

```
# New Example 2:
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(4)
root.right.left = TreeNode(3)

result = is_symmetric(root)
if result != -1:
    print("The closest duplicate value to the root is:", result)
else:
    print("No duplicate values found.")
```

```
 The closest duplicate value to the root is: 4
```

```
# EXPLAIN WHY YOUR SOLUTION WORKS
# The solution functions as recursively traversing the tree. It is controlled whether mode's value is already in the seen values, at each no
# If so, it returns that value, and indicates that a duplicate closest to the root. Otherwise, it proceeds with subtrees.
# If there is a duplicate, it returns that value. If there is no duplicate, it returns -1.
```

```
# EXPLAIN THE PROBLEM’S AND SPACE COMPLEXITY
# Time complexity: These notes are from our lesson slides: "The time complexity of recursion depends on the number of time the function call
# Factorial: the fact is called n times before reaching the base case so its O(1n) = O(n) If a recursive function called itself twice, then
# In this problem complexity; time complexity is O(n). As in the notes, n is the number of nodes. We traverse each node once.

# Space complexity: These are from our lesson slides: Space complexity of recursion
# Notice the call stack takes up space in memory. How much depends on the depth of the recursion. Think about the maximum amount of space th
# Factorial: O(n), when recursion reaches the base case Even when you have multiple branches, it’s possible only 1 branch at depth n is in m
# Here, the space complexity is also  O(n) due to the recursion stack and the set used to stoe values.


# EXPLAIN THE THINKING TO AN ALTERNATIVE SOLUTION (NO CODING REQUIRED, BUT A CLASSMATE READING THIS SHOULD BE ABLE TO CODE IT UP BASED OFF Y
# A breadth-first search (BFS) traversal of the tree can be used. It checks for duplicates level by level.
# A queue to perform BFS traversal can be used.
# At every level, we can keep track of seen values.
# Whenever a duplicate value comes, we return it as closest duplicate to the root.
```