Assignment 2

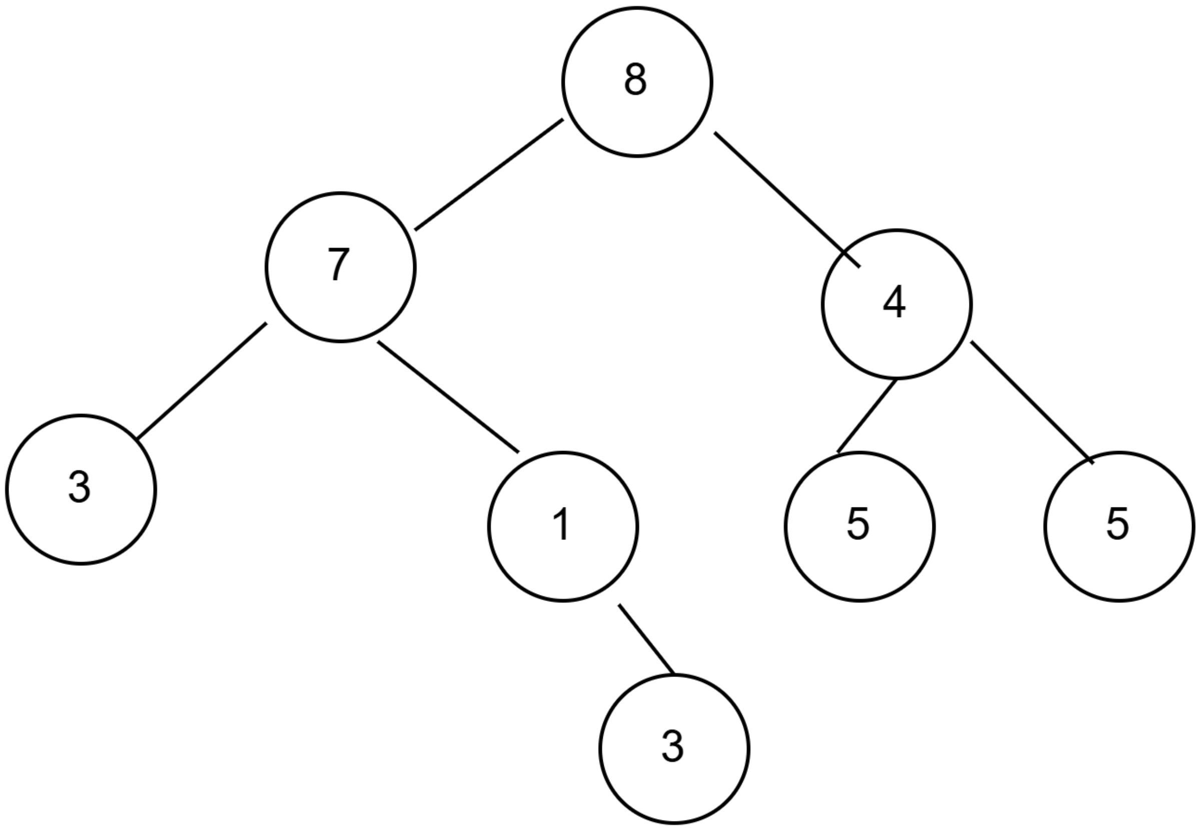
Paraphrase the problem in your own words.

Given a binary tree as input, create a function that will determine if the tree has any duplicate values. If it does, the code will return the value of the duplicate pair that has the smallest distance from the root, where distance is measure by the shortest route by BFS (level order traversal). If no matching pairs are found, the function returns -1.

New Example

Input tree, root1: [8, 7, 4, 3, 5, 1, 5, 3]

Output: 5



```
In [3]: root1 = TreeNode(8)
        root1.left = TreeNode(7)
        root1.right = TreeNode(4)
        root1.left.left = TreeNode(3)
        root1.right.left = TreeNode(5)
        root1.left.right = TreeNode(1)
        root1.right.right = TreeNode(5)
        root1.left.right.right = TreeNode(3)

        is_duplicate(root1)
```

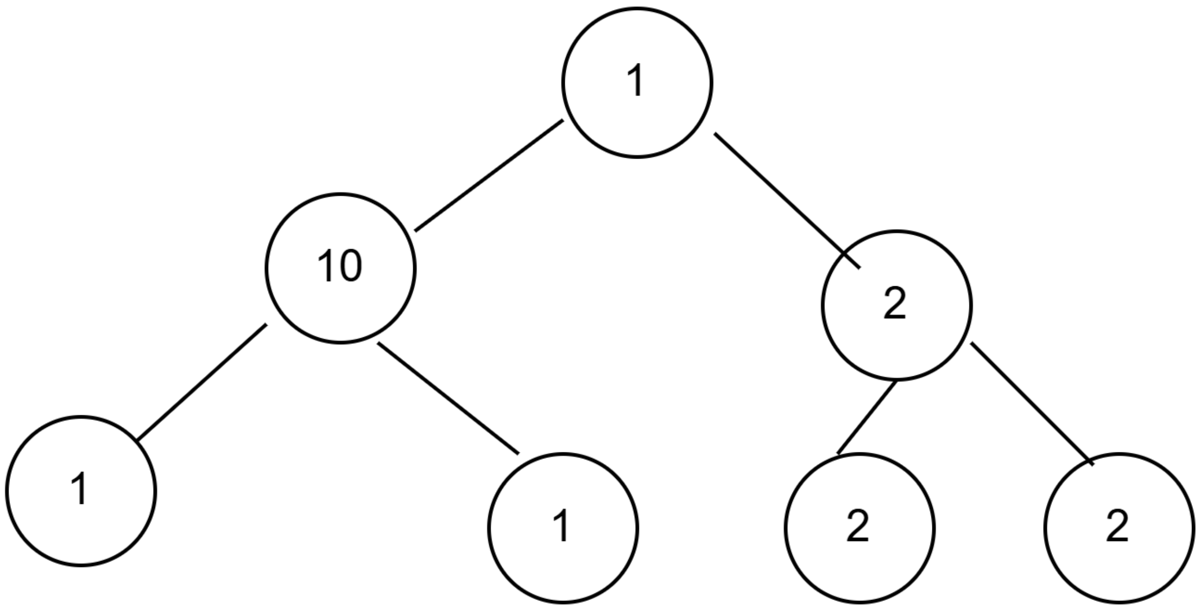
```
Out[3]: 5
```

Partner's Example Walkthrough

```
In [2]: root = TreeNode(1)
        root.left = TreeNode(10)
        root.right = TreeNode(2)
        root.left.left = TreeNode(1)
        root.left.right = TreeNode(1)
        root.right.left = TreeNode(2)
        root.right.right = TreeNode(2)

        # Input tree, root: [1, 10, 2, 1, 2, 2]
        # Output: 1
        is_duplicate(root)
```

```
Out[2]:  1
```



In this example the values 1 and 2 both appear in duplicate nodes. The first pair to be visited using BFS (Level Order Traversal), is the pair counted as closest to the root.

In the case of this tree we visit the nodes in the order: 1, 10, 2, 1, 2, 2. So the first duplicate value is 1.

Partner's Code

```
In [1]:
from collections import deque

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 root:
        queue = deque([root])                       # O(1)
        visited = set()                             # O(1)

        while queue:
            node = queue.popleft()                  # O(1)
            if node.val in visited:                 # O(n)
                return node.val
            visited.add(node.val)                   # O(1)

            if node.left:
                queue.append(node.left)             # O(1)
            if node.right:
                queue.append(node.right)            # O(1)


    return -1
```

Explain why their solution works in your own words

The function takes a binary tree as input. We initialize a 'deque' object containing the root of the tree called 'queue'. Deque allows us to pop items from the left or right end of queue. We also initialize an empty set called 'visited'. This set will keep track of the values of all of the nodes we have visited.

We begin a loop, while queue is nonempty.

First Loop Execution: The first step is to remove the leftmost item of queue (in the first instatiation of the loop this will be the root of the tree). Then we add the current node (the root) to 'visited'. Now we add the left and right children of our current node to the queue, if they exist (i.e., root.left.val and root.right.val).

Subsuquent loops: We remove the leftmost item from the queue, this is our current node. If the current node's value' is in 'visited', we have found a duplicate and return the value of the current node. If not, we add the current node's value to "visited". Now we add the left and right children of our current node to 'queue' if they exist.

Here we are applying the order first in the queue, first out of the queue. This way we are prioritizing the nodes with shortest distance from the root in terms of level order traversal.

Explain the 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 tree. Creating the queue from root, has time complexity O(1). Initializing the set 'visited' has O(1).

Popping an element from the end of 'queue' has time complexity O(1). Checking if an element is in 'visited' is O(len(visited)). The number of elements in 'visited' is at most the number of nodes in the tree. So the time complexity is O(n).

Appending elements to queue has time complexity O(1).

The space complexity is O(n). The code uses space for 'queue' and 'visited'. The object 'queue' has at most 3 elements at a time (the set begins with 1 element, and in each execution of the loop we add 2 elements, and subsequently remove 1 element). So it has a space complexity of O(3).

The space used for the set 'visited' is O(n). Since 'visited' has at most n elements.

Critique of partner's solution

The code is clean and efficient, and only has time and space complexity of O(n). It could also be done using recursion, but the time and space complexity would not improve as we still need to keep track of the nodes visited which will still give us a space and time complexity of O(n). The code runs as expected and gives the desired ouptut.

Reflection

The assignments in this module helped me to focus on the space and time complexity of code, and how it is processed by the computer. In particular, I compared the time and space complexity of some of the standard built in Python methods and functions. I considered different sorting algorithms and the costs and benefits of using them. Although it is often easier to work with a sorted list, using a sorting function is not necessarily the most efficient way to program an algorithm.

In the first assignment I thought through how to implement my for loop as a recursive function, intending to reduce the time complexity of my code. However, in the end the more streamlined code which did not include recursion was faster. I realized I should also consider eliminating any unnecessary operations to quicken processing.

Additionally, I considered which type of objects I was using to store information in the programs. Sets are generally faster than lists, and using deque objects can be even faster as you can remove elements from either end. Looking at my partners’ code I reflected on the whether a hash or deque object could have been applied in the first assignment to increase efficiency. Overall, it seems that the most effective code is often the option that is the simplest and most elegant.