# Assignment 2

## Henry Arzumanian

***Paraphrase the problem in your own words***

Return the closest to the root duplicate you find in a binary tree. If there are no duplicates, return -1.

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

My example:

```
        (8)
    (5)     (1)
  (3) (3)  (0) (1)
```

Input: root = [8, 5, 1, 3, 3, 0, 1] Level Order Traversel

Output: 1

My partner's example:

Input: root = [1, 3, 7, 5, 6, 10, 9]

Output: -1

In his example, the output is -1 because the tree contains no duplicates

***Copy the solution your partner wrote***

In [ ]:
```python
# importing necessary library
from collections import deque
# creating a template for each node which will have two child (left and right)
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# defining a new function which can convert a list of input to a binary tree struct
def list_to_tree(nodes):

    # creating and storing first node as root
    root = TreeNode(nodes[0])
    # initiates level order traversal
    queue = deque([root])
```

```
    i = 1

    # creating a while loop which will keep stroing left and right child
    while queue and i < len(nodes):
        current = queue.popleft() # Pop the leftmost node from the queue

        # creating left child if the value is not None
        if nodes[i] is not None:
            current.left = TreeNode(nodes[i])
            queue.append(current.left)

        i += 1
    # Creating right child if the value is not None
        if i < len(nodes) and nodes[i] is not None:
            current.right = TreeNode(nodes[i])
            queue.append(current.right)

        i += 1
    return root

# finding duplicates and storing them in dictionary
def find_closest_duplicate(root):
    queue = deque([(root, 0)])
    duplicates = {}

    while queue:

        # Pop the leftmost node from the queue
        node, distance = queue.popleft()

        # check for duplicate
        if node.val in duplicates:
            return node.val

        # update dictionary with current node's value and distance
        duplicates[node.val] = distance

        # add left child to the queue with an increased distance
        if node.left:
            queue.append((node.left, distance + 1))

        # add right child to the queue with an increased distance
        if node.right:
            queue.append((node.right, distance + 1))

    # returning -1 if no duplicates found
    return -1

def is_symmetric(root: TreeNode) -> int:
    return find_closest_duplicate(root)
```

In [ ]:

```
# Demo usage for example 4:
# Convert the list to a binary tree and storing a root4
root4 = list_to_tree([1,3,7,6,6,10,7])
```

```
# expected outcome is -1
print(is_symmetric(root4))
```

6

***Explain why their solution works in your own wordss***

His solution works, but it doesn't always produce the required outcome. Instead of returning the duplicate that is closest to the root in terms of depth, it returns the first duplicate it encounters while performing a BFS traversal. So in the example I provided above, my partner's solution would return 3 instead of 1 (the correct answer).

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

The time complexity is O(N) because in the function "list_to_tree," we have to iterate over all elements in a list, and in the function "find_closest_duplicate", in the worst-case scenario, we have to traverse all nodes.

The space complexity is also O(N) because we have to store all nodes in a tree.

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

I think my partner is on the right track with his approach, but he forgot to keep track of depth. To solve this issue, the function should keep track of depth and store all the duplicates in a dictionary or (maybe) a priority queue. And after it traverses the entire tree, it should return the value with the lowest depth. He can also improve the average time complexity by adding some exit requirements (i.e., exit if depth equals 1).

***Reflection***

To successfully solve Assignment 1, I first had to understand the stated requirements. Once I understood what the algorithm was supposed to do, I created an outline and then a rough implementation of it. That first attempt produced the desired outcome but was not very efficient and was difficult to read. I realized I could improve it by using a set instead of a for loop and a list to identify and keep only unique values. And then I eliminated the anohter for loop by using list comprehension instead, making the code shorter and easier to read. More than anything, Assignment 1 helped me improve my comprehension of time and space complexity. I reviewed the appropriate chapters and practiced more using online sources.

Reviewing my partner's assignment helped me understand trees and tree traversal methods better. It forced me to think about the trade-off between DFS and BFS and consider multiple approaches to this problem. In the process, I also learned how to implement a priority queue in Python. I realized that reviewing someone's code is the best exercise for improving the readability of your own code, and I will try to do it every week.