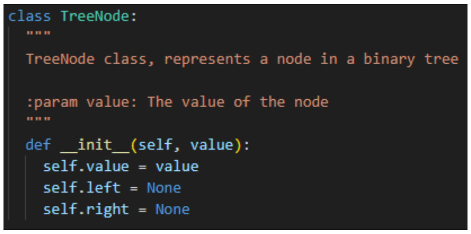
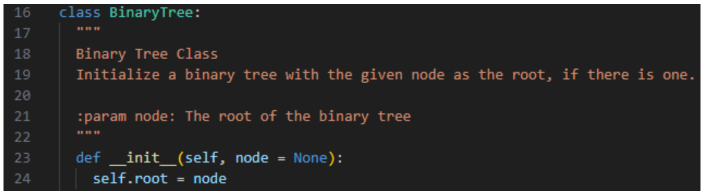
Keiffer Tan

Binary Tree Report

** I have to shoutout the Codeium, I first started this assignment without it and after I installed it, I was able to accomplish so much in a shorter amount of time that it would've taken me! I especially love the way it helps me with comments and docstrings! It can predict what I would've written and helps with the format especially since I don't remember how to format it sometimes! **

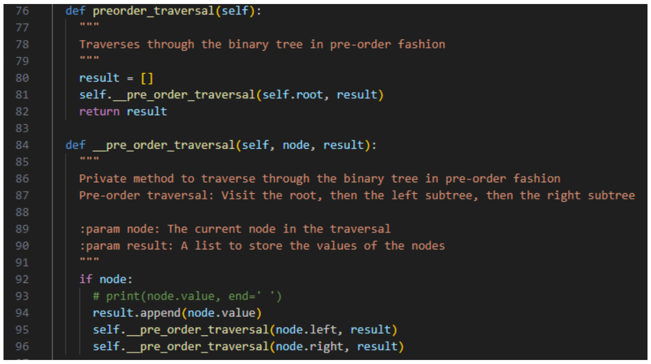


```
class TreeNode:
    """
    TreeNode class, represents a node in a binary tree

    :param value: The value of the node
    """
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```



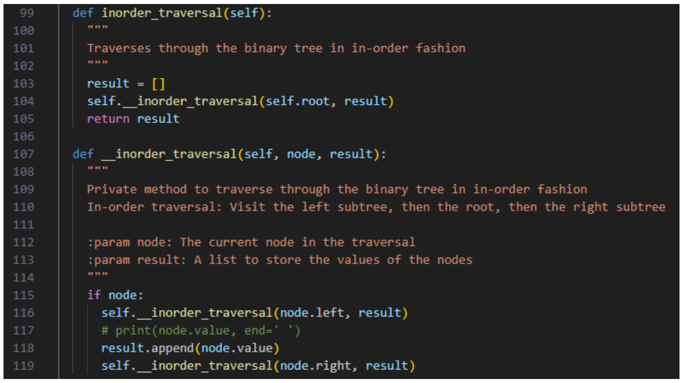
```
class BinaryTree:
    """
    Binary Tree Class
    Initialize a binary tree with the given node as the root, if there is one.

    :param node: The root of the binary tree
    """
    def __init__(self, node = None):
        self.root = node
```

- TreeNode class is created to define the structure of a node within a tree data structure.
- The BinaryTree class is defined with internal method such as adding nodes, dynamically retrieving the depth/height of the tree, and different traversal methods.



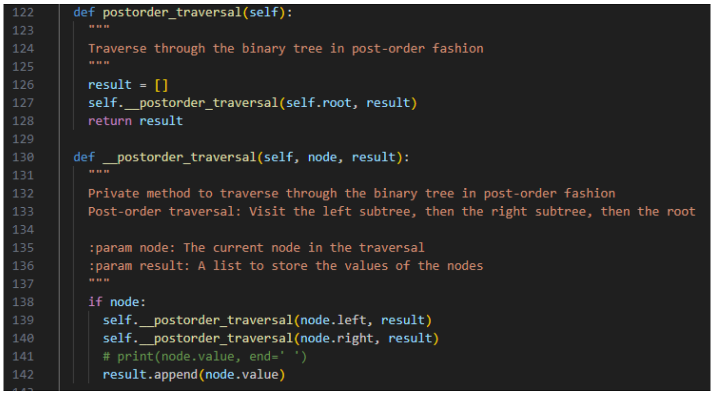
```
    def preorder_traversal(self):
        """
        Traverses through the binary tree in pre-order fashion
        """
        result = []
        self.__pre_order_traversal(self.root, result)
        return result

    def __pre_order_traversal(self, node, result):
        """
        Private method to traverse through the binary tree in pre-order fashion
        Pre-order traversal: Visit the root, then the left subtree, then the right subtree

        :param node: The current node in the traversal
        :param result: A list to store the values of the nodes
        """
        if node:
            # print(node.value, end=' ')
            result.append(node.value)
            self.__pre_order_traversal(node.left, result)
            self.__pre_order_traversal(node.right, result)
```

- My implementation of the order transversals requires a public and private method since I need to be able to pass in the root node address. Generally, the root node should be private and inaccessible outside of the class methods. (I should probably make it as __root so that it is a private variable)
- I recursively traverse through the tree, appending the root, then travel the left node, and then the right node



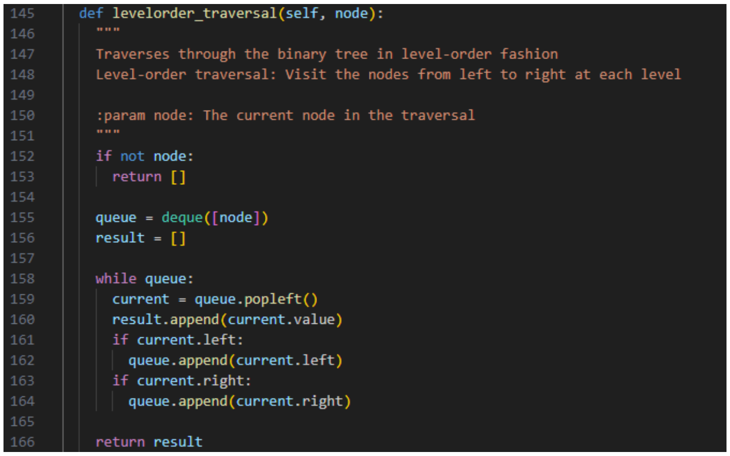
```
    def inorder_traversal(self):
        """
        Traverses through the binary tree in in-order fashion
        """
        result = []
        self.__inorder_traversal(self.root, result)
        return result

    def __inorder_traversal(self, node, result):
        """
        Private method to traverse through the binary tree in in-order fashion
        In-order traversal: Visit the left subtree, then the root, then the right subtree

        :param node: The current node in the traversal
        :param result: A list to store the values of the nodes
        """
        if node:
            self.__inorder_traversal(node.left, result)
            # print(node.value, end=' ')
            result.append(node.value)
            self.__inorder_traversal(node.right, result)
```

- Similar to before, I just needed to switch where I am appending the node, so I moved to append to the middle since in-order traversal is L-N-R



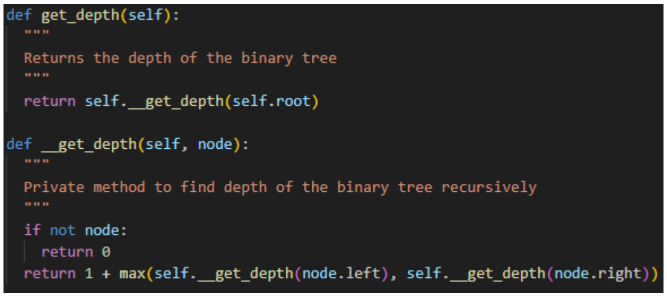
```
def postorder_traversal(self):
    """
    Traverse through the binary tree in post-order fashion
    """
    result = []
    self.__postorder_traversal(self.root, result)
    return result

def __postorder_traversal(self, node, result):
    """
    Private method to traverse through the binary tree in post-order fashion
    Post-order traversal: Visit the left subtree, then the right subtree, then the root

    :param node: The current node in the traversal
    :param result: A list to store the values of the nodes
    """
    if node:
        self.__postorder_traversal(node.left, result)
        self.__postorder_traversal(node.right, result)
        # print(node.value, end=' ')
        result.append(node.value)
```

- As usual, I just moved the append to the end since post-order traversal is L-R-N

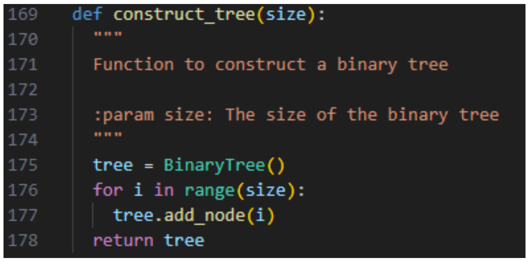


```
def levelorder_traversal(self, node):
    """
    Traverses through the binary tree in level-order fashion
    Level-order traversal: Visit the nodes from left to right at each level

    :param node: The current node in the traversal
    """
    if not node:
        return []

    queue = deque([node])
    result = []

    while queue:
        current = queue.popleft()
        result.append(current.value)
        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)

    return result
```

- I had trouble thinking about the iterative way to approach level-order traversal, and I think I couldn't wrap my head around it at first. Eventually it worked, and I was able to use this same concept in adding a node to the tree!



```
def get_depth(self):
    """
    Returns the depth of the binary tree
    """
    return self.__get_depth(self.root)

def __get_depth(self, node):
    """
    Private method to find depth of the binary tree recursively
    """
    if not node:
        return 0
    return 1 + max(self.__get_depth(node.left), self.__get_depth(node.right))
```



```
def construct_tree(size):
    """
    Function to construct a binary tree

    :param size: The size of the binary tree
    """
    tree = BinaryTree()
    for i in range(size):
        tree.add_node(i)
    return tree
```

- For testing, I created a method to dynamically retrieve the depth of the tree to test for different sizes of trees.

Keiffer Tan

- Construct_tree method is just a simple way to create a tree of a certain size (amount of elements)