UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Ampong, J-kevin L.

Instructor:
Engr. Maria Rizette H. Sayo

November, 9, 2025

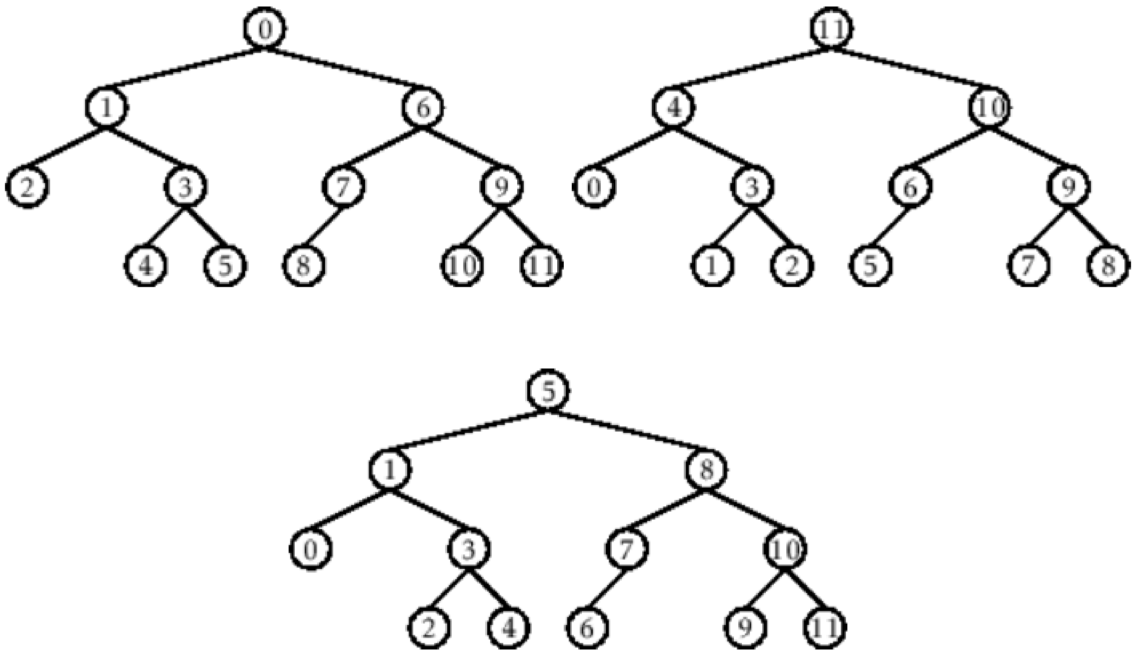
I. Objectives

Introduction

An abstract non-linear data type with a hierarchy-based structure is a tree. It is made up of links connecting nodes (where the data is kept). The root node of a tree data structure is where all other nodes and subtrees are connected to the root.

This laboratory activity aims to implement the principles and techniques in:

- To introduce Tree as Non-linear data structure
- To implement pre-order, in-order, and post-order of a binary tree



- Figure 1. Pre-order, In-order, and Post-order numberings of a binary tree

II. Methods

- Copy and run the Python source codes.
- If there is an algorithm error/s, debug the source codes.
- Save these source codes to your GitHub.
- Show the output

1. Tree Implementation

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

    def add_child(self, child_node):
        self.children.append(child_node)

    def remove_child(self, child_node):
        self.children = [child for child in self.children if child != child_node]
    def traverse(self):
        nodes = [self]
        while nodes:
            current_node = nodes.pop()
            print(current_node.value)
            nodes.extend(current_node.children)

    def __str__(self, level=0):
        ret = "  " * level + str(self.value) + "\n"
        for child in self.children:
            ret += child.__str__(level + 1)
        return ret

# Create a tree
root = TreeNode("Root")
child1 = TreeNode("Child 1")
child2 = TreeNode("Child 2")
grandchild1 = TreeNode("Grandchild 1")
grandchild2 = TreeNode("Grandchild 2")

root.add_child(child1)
root.add_child(child2)
child1.add_child(grandchild1)
child2.add_child(grandchild2)

print("Tree structure:")
print(root)

print("\nTraversal:")
root.traverse()
```

Questions:

1 When would you prefer DFS over BFS and vice versa?
The choice between DFS and BFS hinges on the problem you're trying to solve and the shape of your tree. You would prefer BFS (Breadth-First Search) whenever you need to find the shortest path between two nodes (in terms of levels or "hops"), as it explores level by level, guaranteeing the first time it finds a node, it has done so via the fewest steps. It's also the clear choice for problems like "find all nodes at level 4." You would prefer DFS (Depth-First Search) when the tree is exceptionally wide, as its memory usage depends on the tree's height, not its width, saving you from storing a massive number of nodes in a queue. DFS is also often simpler to implement (especially recursively) and is a good choice if you just need to find any path to a node or explore all parts of a tree, such as in a maze-solving or connectivity problem.

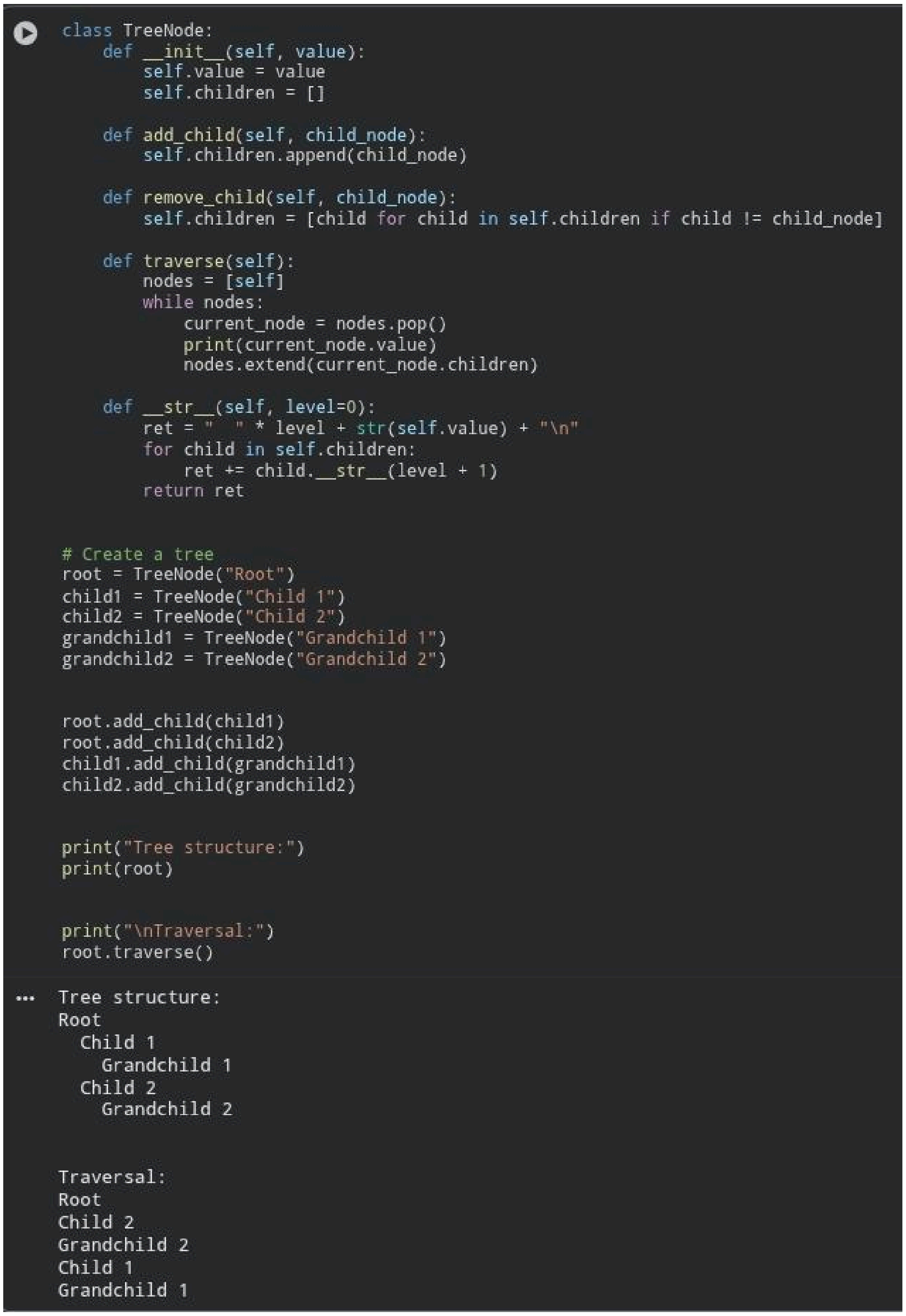
2 What is the space complexity difference between DFS and BFS?
The primary difference in space complexity relates to what each algorithm must store. DFS has a space complexity of O(h), where h is the maximum height (depth) of the tree. This is because it only needs to store the nodes on the single, current path from the root to the node it is visiting (either in the system's call stack for recursion or an explicit stack for iteration). In contrast, BFS has a space complexity of O(w), where w is the maximum width of the tree. This is because its queue must be able to store all the nodes at the tree's broadest level at the same time. Therefore, DFS is more memory-efficient for "bushy" or wide trees, while BFS is more memory-efficient for "stringy" or very deep, narrow trees.

3 How does the traversal order differ between DFS and BFS?

4 The traversal order is the most fundamental difference. **DFS** (Depth-First Search) explores as deeply as possible down one branch before **backtracking**. Using your tree, a pre-order DFS (which your traverse method implements) visits nodes by going as far down a path as it can: Root, then Child 1, then Grandchild 1. Only after hitting this dead end does it backtrack to visit Child 2 and then Grandchild 2.

5 When does DFS recursive fail compared to DFS iterative?
A recursive DFS fails when the tree is too deep, causing a stack overflow error. This happens because every recursive function call adds a new frame to the system's call stack, which has a fixed and relatively small memory limit (in Python, for example, this limit is often 1,000). If the tree's depth exceeds this limit, the stack runs out of space, and the program crashes. An iterative DFS (like the one you wrote) solves this by replacing the system's call stack with its own stack data structure (your nodes list). This data structure lives on the heap, which is limited only by your computer's total RAM—a far larger pool of memory. Therefore, an iterative DFS can handle trees of virtually any depth, failing only if the tree is so large it exhausts all available system memory, whereas the recursive version will fail on any tree deeper than the system's recursion limit.

III. Results



```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

    def add_child(self, child_node):
        self.children.append(child_node)

    def remove_child(self, child_node):
        self.children = [child for child in self.children if child != child_node]

    def traverse(self):
        nodes = [self]
        while nodes:
            current_node = nodes.pop()
            print(current_node.value)
            nodes.extend(current_node.children)

    def __str__(self, level=0):
        ret = "  " * level + str(self.value) + "\n"
        for child in self.children:
            ret += child.__str__(level + 1)
        return ret


# Create a tree
root = TreeNode("Root")
child1 = TreeNode("Child 1")
child2 = TreeNode("Child 2")
grandchild1 = TreeNode("Grandchild 1")
grandchild2 = TreeNode("Grandchild 2")


root.add_child(child1)
root.add_child(child2)
child1.add_child(grandchild1)
child2.add_child(grandchild2)


print("Tree structure:")
print(root)


print("\nTraversal:")
root.traverse()
```

```
Tree structure:
Root
  Child 1
    Grandchild 1
  Child 2
    Grandchild 2


Traversal:
Root
Child 2
Grandchild 2
Child 1
Grandchild 1
```

IV. Conclusion

In summary, the choice between DFS and BFS is a classic tradeoff.

BFS (Breadth-First Search) explores level-by-level using a queue. It is your go-to algorithm for finding the shortest path (in unweighted graphs), but it can consume significant memory (O(width)) if the tree is very wide.
DFS (Depth-First Search) explores branch-by-branch using a stack. It is memory-efficient (O(height)) for wide trees and is simple to implement. However, a recursive DFS can easily fail with a stack overflow on deep trees, making an iterative approach or BFS a safer alternative in that scenario.

References

[1] Co Arthur O.. "University of Caloocan City Computer Engineering Department Honor Code," UCC-CpE Departmental Policies, 2020.