UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Acebedo , Sebastian C.

Instructor:
Engr. Maria Rizette H. Sayo

November 9, 2025

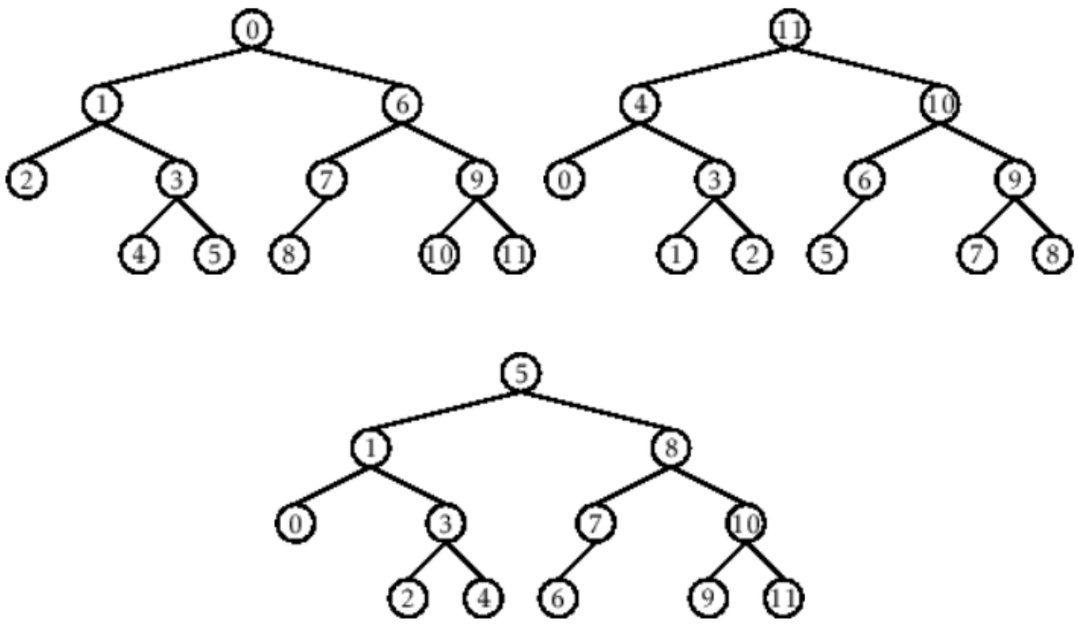
I. Objectives

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?
2. What is the space complexity difference between DFS and BFS?
3. How does the traversal order differ between DFS and BFS?
4. When does DFS recursive fail compared to DFS iterative?

III. Results

1. When would you prefer DFS over BFS and vice versa?

- I'd prefer DFS (Depth First Search) when I want to explore a path deeply before moving to another, like in puzzles, mazes, or when I just need to check if a path exists. It's also good when the solution is deep in the tree. On the other hand, I'd use BFS (Breadth First Search) when I want the shortest path or need to explore all nodes level by level, like in finding the shortest route in a graph.

2. What is the space complexity difference between DFS and BFS?

- DFS usually has a lower space complexity, around O(h) where h is the height of the tree (or O(V) in the worst case for graphs). BFS, however, needs to store all nodes at

the current level, so its space complexity is O(V) (number of vertices). This means BFS uses way more memory, especially for wide graphs.

3. How does the traversal order differ between DFS and BFS?
   - In DFS, we go deep first, visiting nodes along one path before backtracking. The order looks like going down one branch all the way, then moving to the next.
   - In BFS, we go level by level, visiting all nodes at one depth before moving deeper. It's more like spreading out evenly before going further.

4. When does DFS recursive fail compared to DFS iterative?
   - DFS recursive might fail when the tree or graph is too deep, because it can cause a stack overflow error due to too many recursive calls. DFS iterative, which uses an explicit stack, doesn't have this problem since it controls the stack manually and can handle deeper graphs without crashing.

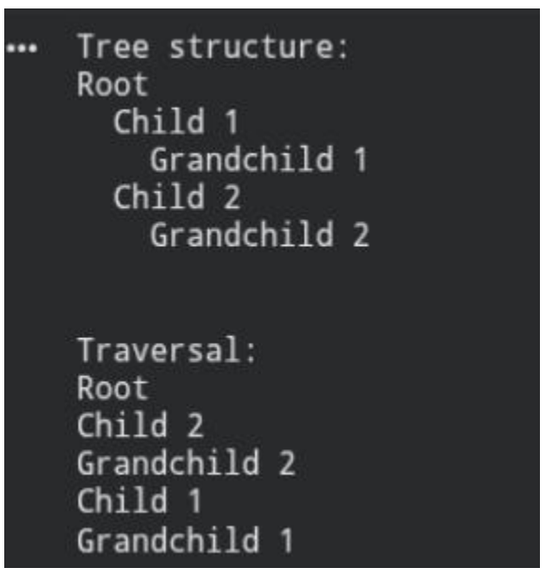


Figure 1 Screenshot of program

IV. Conclusion

In conclusion, this laboratory activity helped me understand how trees work as a non-linear data structure. I learned how nodes are connected in a hierarchical manner, starting from the root node down to its subtrees. By implementing the pre-order, in-order, and post-order traversals, I was able to see how different traversal methods affect the order in which data is

accessed. Overall, this activity strengthened my understanding of tree structures and their importance in organizing and managing data efficiently.

References

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