UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Palmes, Lewis Clark L.

Instructor:
Engr. Maria Rizette H. Sayo

November 9, 2025

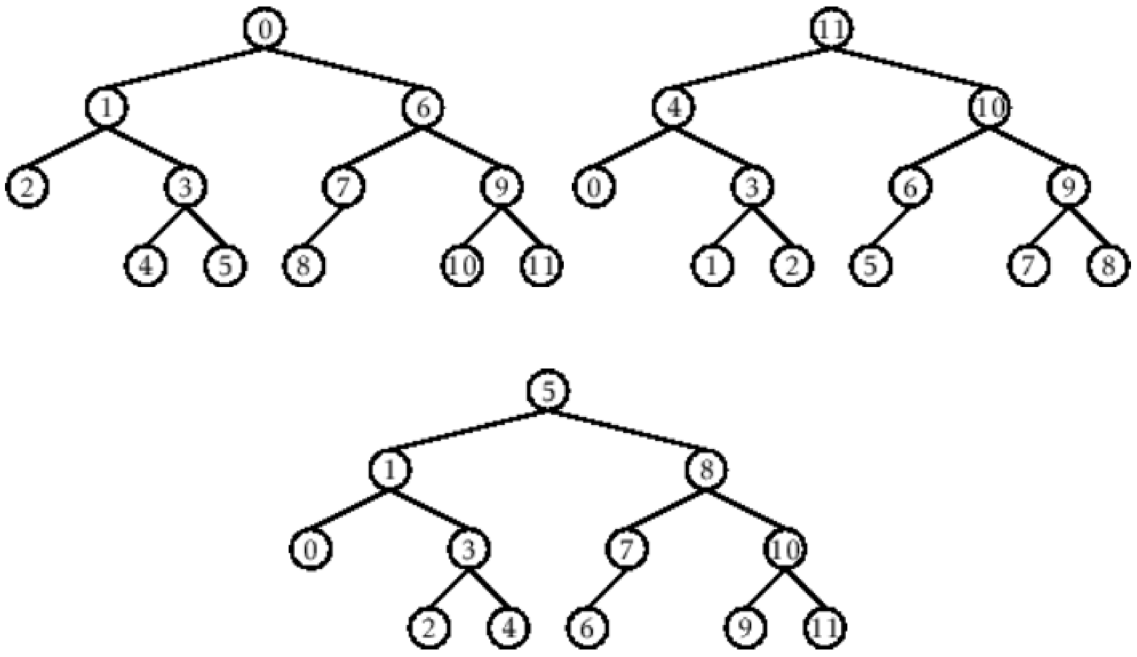
I. Objectives

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?
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?

   Use DFS when exploring deep paths, solving backtracking problems, or when memory is limited. Use BFS when you need the shortest path in an unweighted graph or level-order traversal since it explores nodes layer by layer.

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

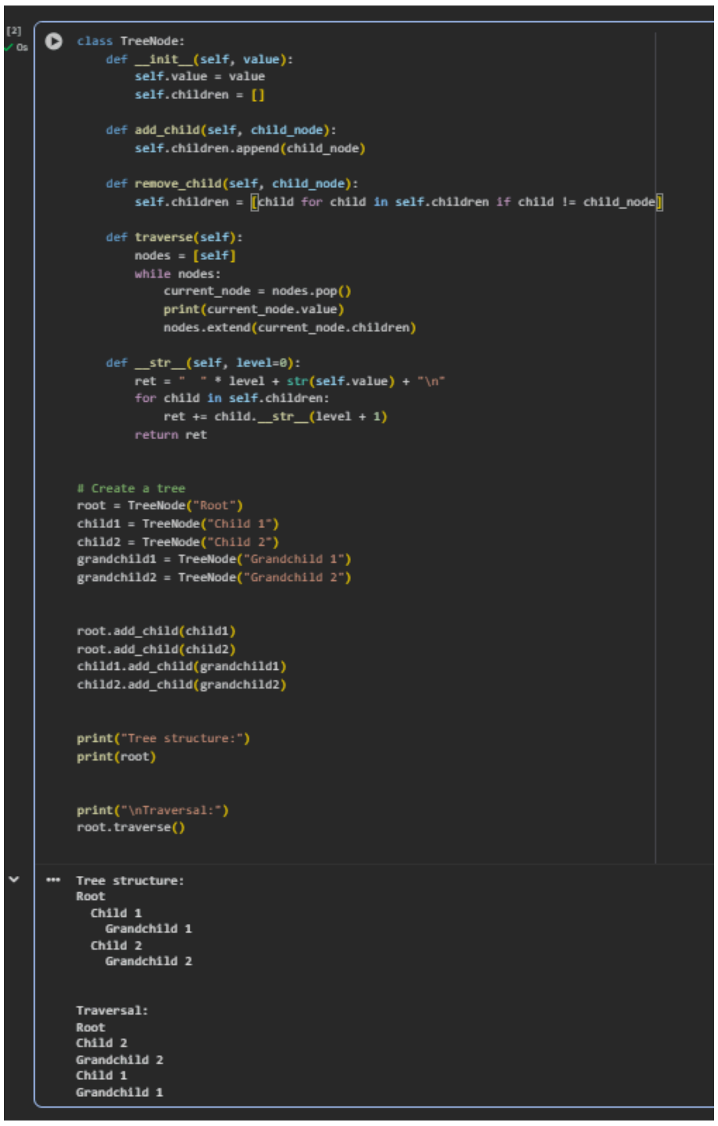
   Depth-First Search has a space complexity that grows with the maximum depth of the search, because it only needs to store the nodes along the current path being explored. Breadth-First Search, on the other hand, requires memory that grows with the number of nodes at each level of the graph, since it keeps all nodes at the current depth in memory before moving deeper. This makes Breadth-First Search much more memory-intensive than Depth-First Search, especially in wide or highly connected graphs.

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

   DFS explores one branch as far as possible before backtracking, using a stack or recursion, resulting in a depth-oriented order. BFS explores all nodes at the current depth before moving deeper, using a queue, resulting in a level-by-level traversal that ensures shortest path discovery in unweighted graphs.

4. When does DFS recursive fail compared to DFS iterative?

   Recursive DFS can fail on very deep graphs because it relies on the call stack, leading to stack overflow when the recursion depth limit is exceeded. Iterative DFS, on the other hand, uses an explicit stack stored in the heap, allowing it to handle much deeper or larger graphs without crashing.



```
[2]
✓ 0s
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
```

Figure 1 Screenshot of program

This Python code defines a TreeNode class for creating and managing a tree structure, allowing nodes to add or remove children, display the tree, and perform a depth-first traversal. The example builds a tree with a root, two children, and two grandchildren, then prints the tree structure followed by the traversal order of the nodes.

IV. Conclusion

This laboratory activity focused on understanding and implementing tree data structures and traversal algorithms using Python. It introduced trees as hierarchical, non-linear data structures composed of nodes connected by links, with emphasis on constructing trees and performing traversals such as depth-first search. The provided code demonstrated creating a tree with multiple levels, adding and removing child nodes, printing the tree structure, and traversing nodes in a depth-first manner. Additionally, the activity explored theoretical concepts by comparing DFS and BFS in terms of use cases, space complexity, traversal order, and limitations of recursive versus iterative DFS. Overall, this exercise reinforced fundamental tree operations and deepened understanding of search strategies in data structures.

References

[1] A. Ahmed, "Simple Explanation on BFS and DFS Graph Traversal Methods," *DevCript*, 9 Aug. 2022. [Online]. Available: https://www.devcript.com/simple-explanation-on-bfs-and-dfs-graph-traversal-methods/

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

[3] "Parallel BFS graph traversal on images using structured grid," IEEE Xplore. Available: https://ieeexplore.ieee.org/document/5652307/

[4] R. Tarjan, "Depth-first search and linear graph algorithms," in *Proc. 12th Annual Symposium on Switching and Automata Theory*, East Lansing, MI, USA, Oct. 13-15 1971, pp. 114-121. Available: https://collaborate.princeton.edu/en/publications/depth-first-search-and-linear-graph-algorithms

[5] "Tree Traversal Techniques in Python," *GeeksforGeeks*, 23 Jul. 2025. Available: https://www.geeksforgeeks.org/tree-traversal-techniques-in-python/