UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Balaoro, Judge Wayne B.

Instructor:
Engr. Maria Rizette H. Sayo

November 9, 2025

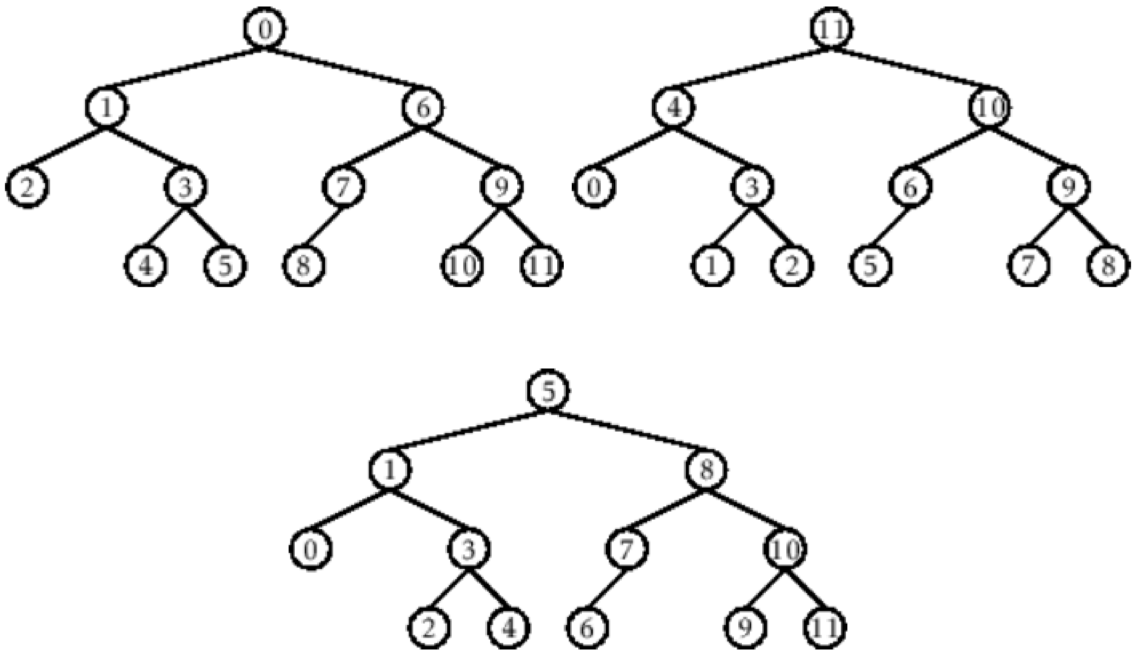
I. Objectives

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

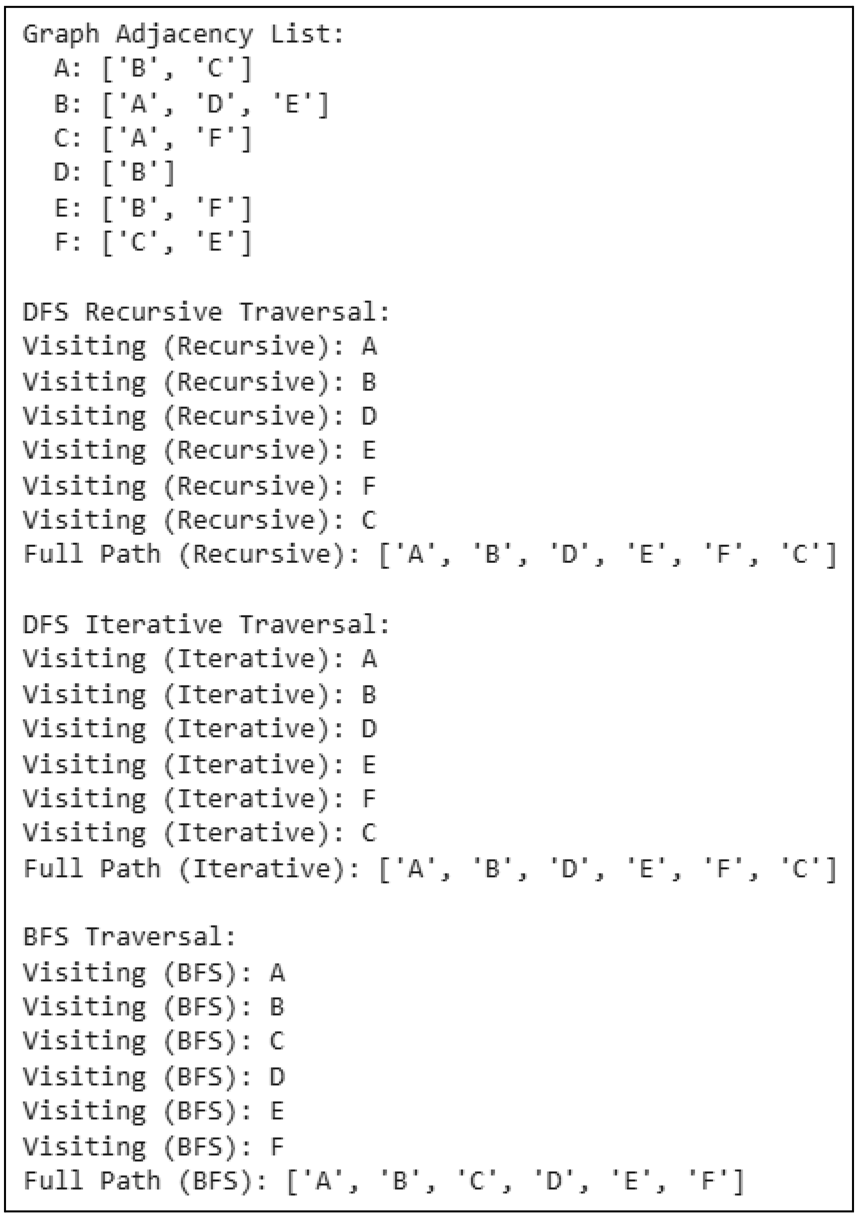
1. Tree Implementation

```
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



```
Graph Adjacency List:
  A: ['B', 'C']
  B: ['A', 'D', 'E']
  C: ['A', 'F']
  D: ['B']
  E: ['B', 'F']
  F: ['C', 'E']

DFS Recursive Traversal:
Visiting (Recursive): A
Visiting (Recursive): B
Visiting (Recursive): D
Visiting (Recursive): E
Visiting (Recursive): F
Visiting (Recursive): C
Full Path (Recursive): ['A', 'B', 'D', 'E', 'F', 'C']

DFS Iterative Traversal:
Visiting (Iterative): A
Visiting (Iterative): B
Visiting (Iterative): D
Visiting (Iterative): E
Visiting (Iterative): F
Visiting (Iterative): C
Full Path (Iterative): ['A', 'B', 'D', 'E', 'F', 'C']

BFS Traversal:
Visiting (BFS): A
Visiting (BFS): B
Visiting (BFS): C
Visiting (BFS): D
Visiting (BFS): E
Visiting (BFS): F
Full Path (BFS): ['A', 'B', 'C', 'D', 'E', 'F']
```

Figure 1: Program Output

1. I would prefer DFS over BFS, when I need to explore a single path as far as possible before trying another path just like how you solve a maze. I would use BFS when i want to find a shortest path just like how we see on our suggestions on friends that are the “friends of our friends” on social networking apps like Facebook, because it checks all nearby nodes first, layer by layer.

2. The difference is that BFS can use a lot of memory for wide graphs because its queue will have to hold many nodes at the same level. While the DFS can use a lot of memory for deep graphs because it uses a stack to store the entire long path it’s currently storing.

3. DFS uses a stack that uses the LIFO (Last-In, First-Out) to dive deeper which means it explores one branch completely before backtracking. On the other hand, BFS uses a

queue that implements FIFO (First-In, First-Out) to explore in layers visiting all neighbors of a node before moving to the next level.

4. The recursive DFS can fail with Stack Overflow Error if the graph is so deep. This can happen because each of those functions adds to the system's limited stack. To avoid this problem we use iterative DFS by using its own stack data structure, which is stored on a much larger system and won't overflow just because a path is too long.

IV. Conclusion

In this lab exercise, we examined tree data structures and carried out fundamental tree operations along with traversal techniques, focusing on pre-order, in-order, and post-order traversals. Through executing and examining the program, we discovered the hierarchical connections between nodes and how traversal algorithms handle each node uniquely based on their strategy. The exercise also enhanced our comprehension of DFS and BFS, their suitable applications, and the distinctions in space complexity and performance. By engaging in debugging and testing, we obtained practical experience in implementing theoretical concepts within a real program, highlighting the significance of selecting the appropriate traversal technique for effectively addressing real-world issues

References

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