UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Dispo, Lei Andrew T.

Instructor:
Engr. Maria Rizette H. Sayo

11, 09, 2025

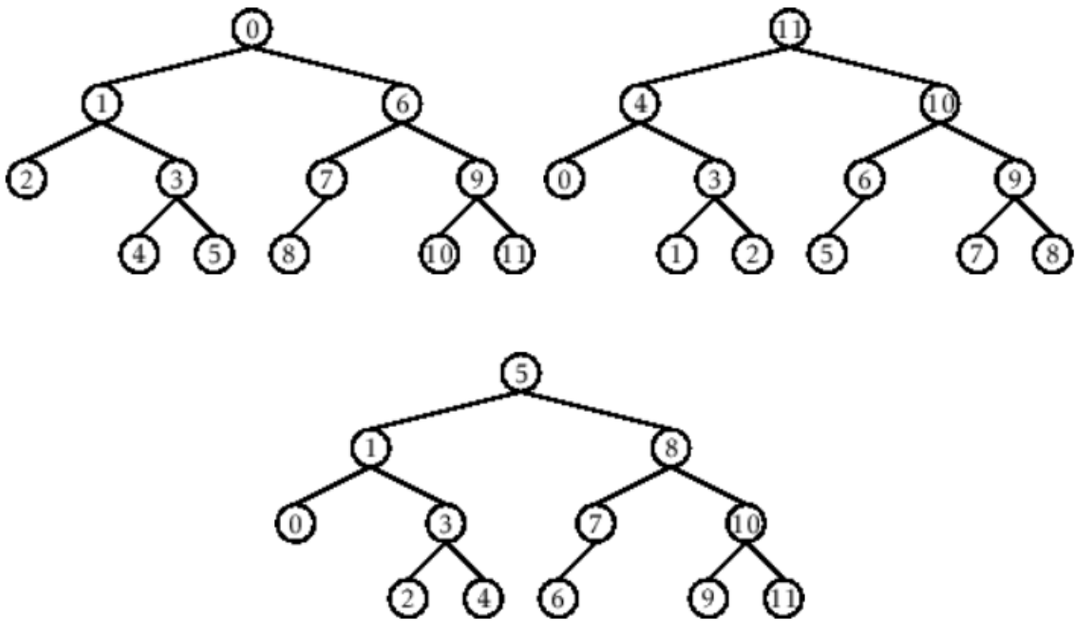
Objectives

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

I. 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?

II. Results

1. DFS is preferred when exploring all possible paths or checking connectivity in deep or large graphs, such as solving mazes or topological sorting. BFS is better when the goal is to find the shortest path or explore nodes level by level, such as in navigation or network routing.
2. DFS uses less memory because it goes deep one path at a time. BFS uses more since it stores many nodes in a queue.
3. DFS goes deep into one branch before moving to another, while BFS explores all nodes level by level. In short, DFS goes deep first; BFS goes wide first.
4. Recursive DFS can fail on very large or deep graphs because it uses too many recursive calls, causing a stack overflow. Iterative DFS avoids this problem by using a manual stack instead of recursion.

Figure 1

III. Conclusion

This program creates a simple tree structure using classes. It allows adding, removing, and displaying child nodes, and shows how to traverse and print all nodes in the tree.

DFS and BFS are both graph traversal algorithms. DFS goes deep first and uses less memory, while BFS explores level by level and is better for finding the shortest path. Recursive DFS can cause stack overflow on large graphs, but iterative DFS avoids this issue.

References

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