UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Directo, Hannah Thea B.

Instructor:
Engr. Maria Rizette H. Sayo

November 7, 2025

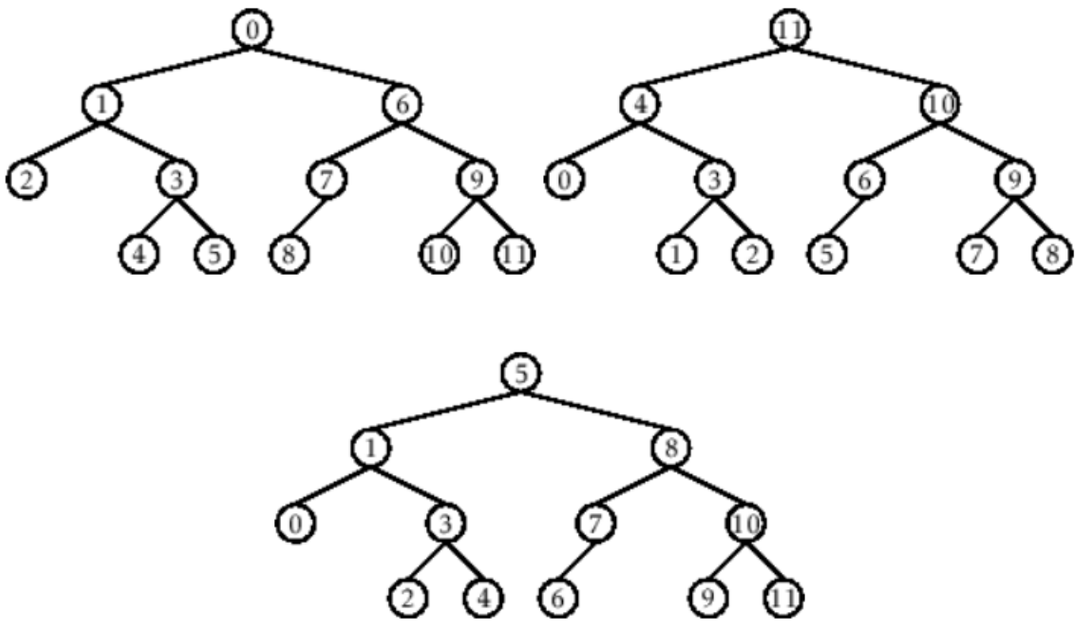
I. Objectives

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

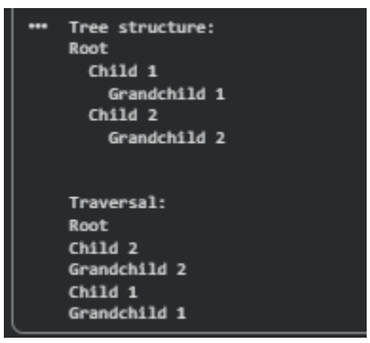
- Copy and run the Python source codes.
- If there is an algorithm error/s, debug the source codes.
- Save these source codes to your GitHub.
- Show the output

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()
```

Output:



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. Depth-First Search (DFS) is preferred when you want to explore deep parts of a tree or graph first, or when memory is limited because it doesn't need to store many nodes at once. It is also useful when any valid solution is acceptable rather than the shortest one. On the other hand, Breadth-First Search (BFS) is better when you need to find the shortest path or explore nodes level by level, since it visits all nearby nodes before going deeper.

2. DFS has a space complexity of O(h), where *h* is the height or depth of the tree. This means it only stores nodes along the current path. BFS, however, has a space complexity of O(w), where *w* is the width of the tree, because it needs to keep all nodes at the current level in memory. As a result, BFS usually uses more memory than DFS.

3. The traversal order in DFS goes **deep first**, exploring one branch fully before moving to another. It follows a path until it reaches the end before backtracking. In contrast, BFS goes **wide first**, visiting all nodes at one level before moving down to the next level. This difference affects how quickly each method reaches certain nodes.

4. Recursive DFS can fail when dealing with **very deep trees or graphs** because each recursive call adds to the system's call stack. If the depth is too large, it can cause a **stack overflow** error. Iterative DFS avoids this issue because it uses an explicit stack in memory rather than relying on the system's call stack, making it safer for large or deep structures.

IV. Conclusion

This laboratory exercise helped enhance my understanding of how tree structures work and how they organize data in a hierarchical manner. Through the implementation and traversal of the tree, I learned the differences between depth-first and breadth-first searches, including their practical uses and limitations. The activity also showed how trees can effectively handle connected data, which is essential in various applications such as databases, file management, and decision-making systems. Overall, this experiment strengthened my grasp of non-linear data structures and highlighted their importance in solving real-world computational problems.