UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 14

Tree Structure Analysis

Submitted by:
Palmes, Lewis Clark L.

Instructor:
Engr. Maria Rizette H. Sayo

November 9, 2025

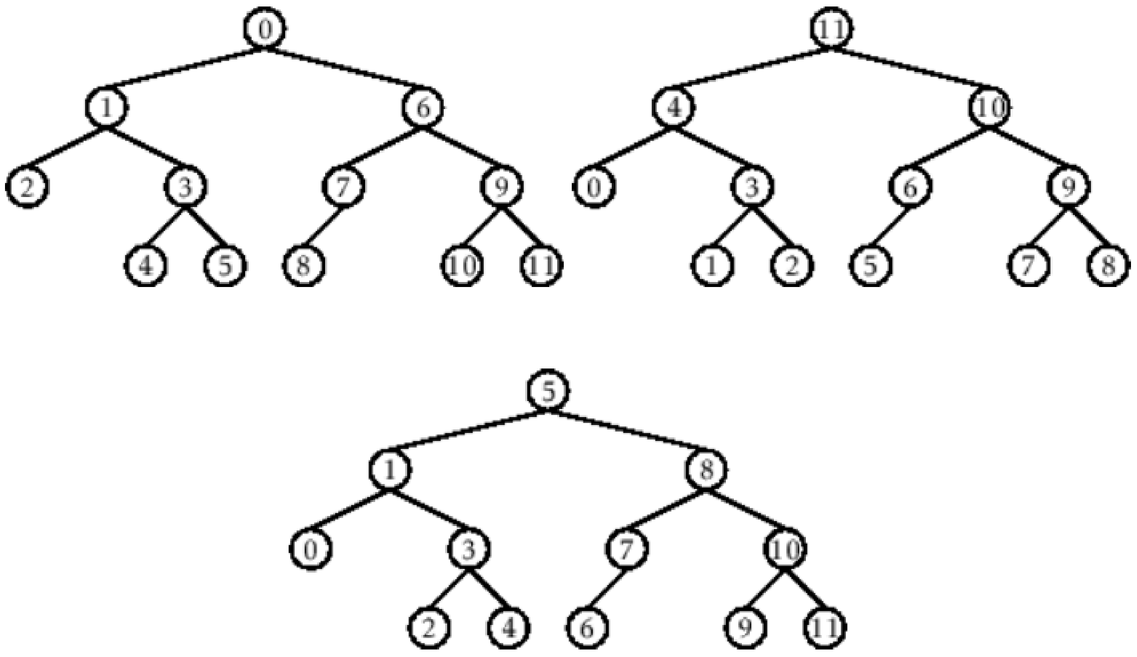
I. Objectives

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 What is the main difference between a binary tree and a general tree?
2 In a Binary Search Tree, where would you find the minimum value? Where would you find the maximum value?
3 How does a complete binary tree differ from a full binary tree?
4 What tree traversal method would you use to delete a tree properly? Modify the source codes.

III. Results

1. What is the main difference between a binary tree and a general tree?

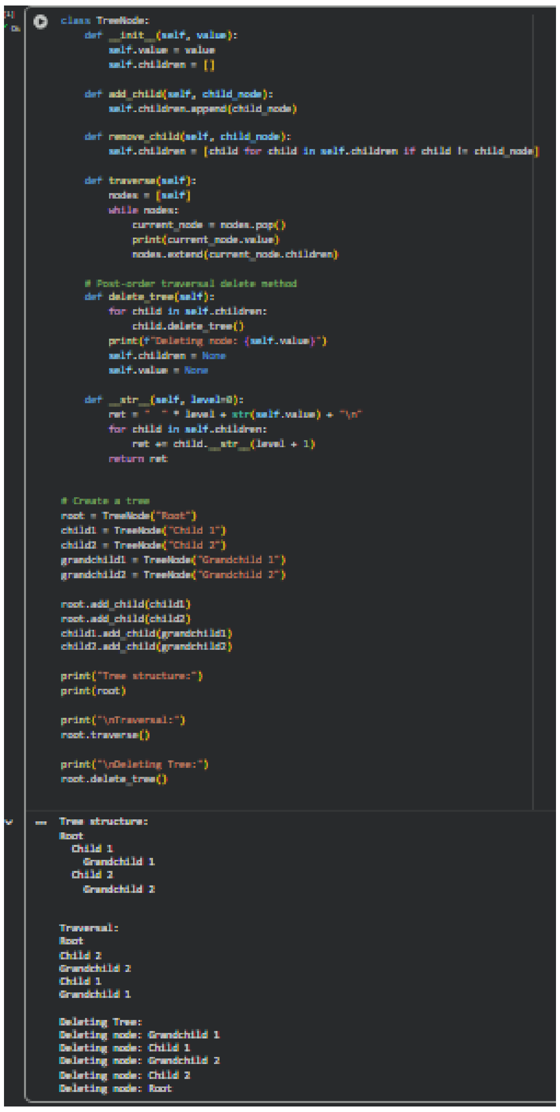
   The main difference is that a binary tree restricts each node to at most two children, commonly referred to as the left and right child, whereas a general tree allows any number of children for each node. This constraint in binary trees enables easier implementation of algorithms like search, insertion, and traversal, while general trees provide more flexible hierarchical structures suitable for representing organizational charts or file systems.

2. In a Binary Search Tree, where would you find the minimum value? Where would you find the maximum value?

    In a Binary Search Tree (BST), the minimum value is always located at the leftmost node, following the left child pointers from the root until no more left children exist. The maximum value is found at the rightmost node, following the right child pointers from the root until no more right children exist. This property arises because BSTs store smaller values on the left and larger values on the right of each node.

3. How does a complete binary tree differ from a full binary tree?

    A full binary tree is a binary tree where every node has either zero or two children, meaning all non-leaf nodes are fully populated. A complete binary tree, on the other hand, is completely filled on all levels except possibly the last, which is filled from left to right. Thus, all full binary trees are complete, but not all complete binary trees are full.



```python
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)

    # Post-order traversal delete method
    def delete_tree(self):
        for child in self.children:
            child.delete_tree()
        print(f"Deleting node: {self.value}")
        self.children = None
        self.value = None

    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()

print("\nDeleting Tree:")
root.delete_tree()
```

```
Tree structure:
Root
  Child 1
    Grandchild 1
  Child 2
    Grandchild 2


Traversal:
Root
Child 2
Grandchild 2
Child 1
Grandchild 1

Deleting Tree:
Deleting node: Grandchild 1
Deleting node: Child 1
Deleting node: Grandchild 2
Deleting node: Child 2
Deleting node: Root
```

Figure 1: Screenshot of the Program

4. What tree traversal method would you use to delete a tree properly? Modify the source codes.

    To properly delete a tree, post-order traversal should be used because it ensures that each node's children are deleted before the parent node, preventing dangling references. In code, this means visiting the left subtree, then the right subtree, and finally deleting the current node.

IV. Conclusion

In this laboratory activity, I learned how tree structures function as non-linear data types that represent hierarchical relationships between data elements. By implementing and analyzing traversal methods such as pre-order, in-order, and post-order, I gained a clearer understanding of how data can be accessed and managed efficiently in trees. The exercise also demonstrated how the post-order traversal method is essential for safely deleting a tree, as it removes child nodes before their parent nodes. Overall, this activity enhanced my understanding of tree operations, their practical applications in computer science, and their importance in organizing and processing complex data structures.

References

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

[2] “Postorder Traversal of Binary Tree,” *GeeksforGeeks*, 2025. Available: https://www.geeksforgeeks.org/dsa/postorder-traversal-of-binary-tree/

[3] “Tree Traversals — Inorder, Preorder and Postorder,” *GeeksforGeeks*, 2025. Available: https://www.geeksforgeeks.org/dsa/tree-traversals-inorder-preorder-and-postorder/

[4] “Write a Program to Delete a Tree,” *GeeksforGeeks*, 2023. Available: https://www.geeksforgeeks.org/dsa/write-a-c-program-to-delete-a-tree/