UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 14

Tree Structure Analysis

Submitted by:
Directo, Hannah Thea B.

Instructor:
Engr. Maria Rizette H. Sayo

November 8, 2025

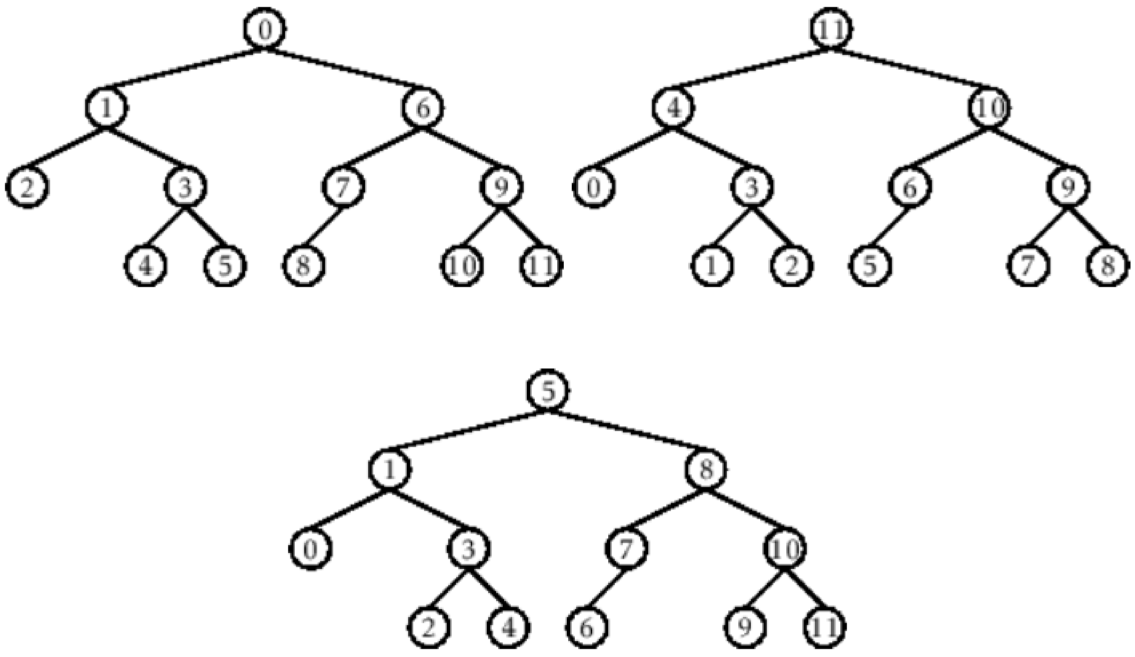
I. Objectives

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

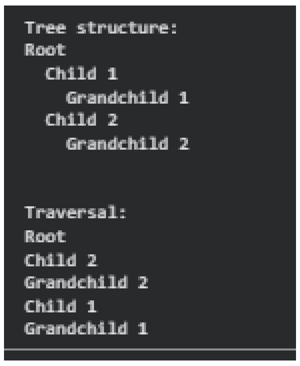
II. Methods

- 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:



```
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 the output

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.

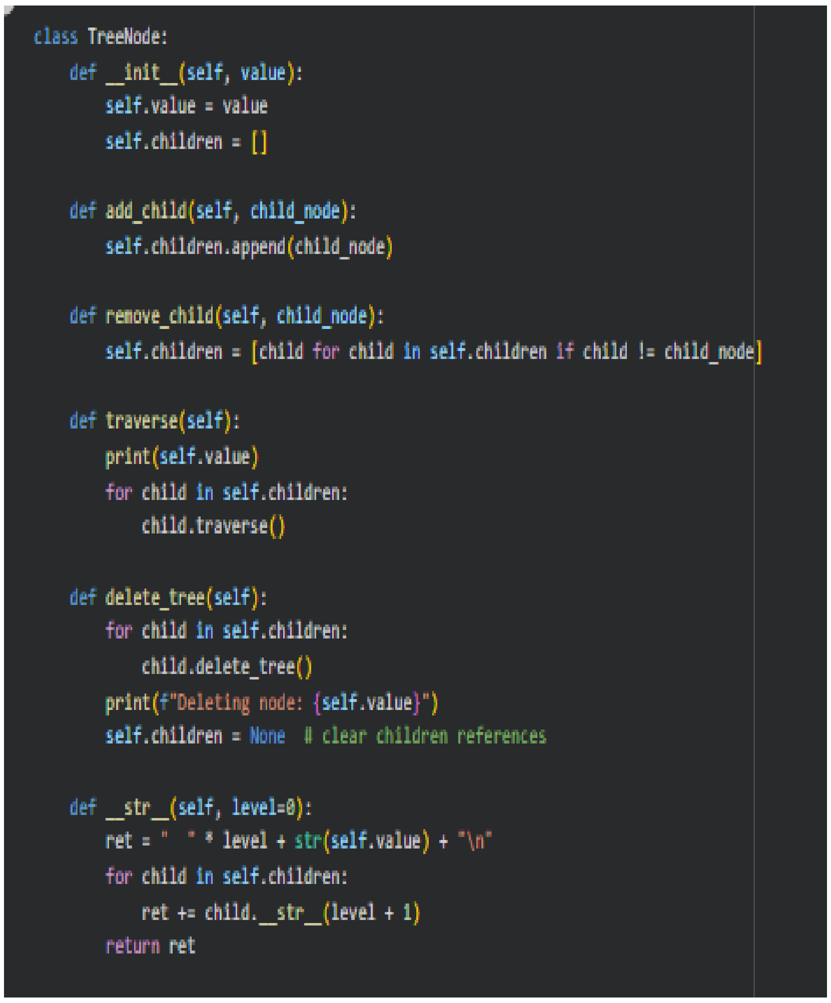
III. Results

1. The main difference between a binary tree and a general tree is the number of children that each node can have. In a binary tree, every node can only have up to two children, usually called the left and right child. Meanwhile, in a general tree, a node can have any number of children without restriction. This makes a general tree more flexible in representing hierarchical data, such as organization charts or file directories, while a binary tree is more structured and often used in searching and sorting algorithms.

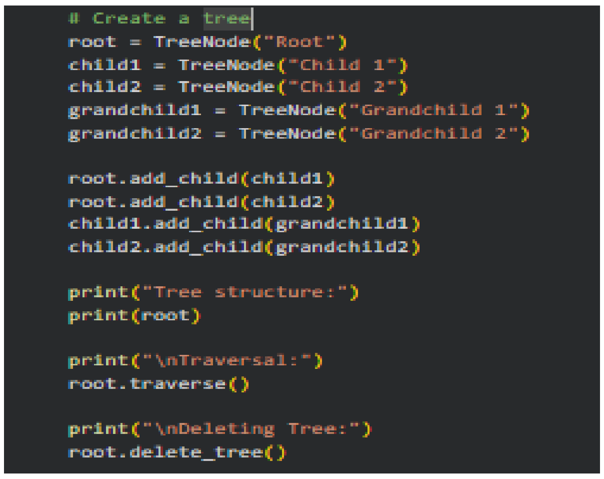
2. In a Binary Search Tree (BST), the smallest value can always be found in the leftmost node, because smaller values are consistently stored in the left subtree. On the other hand, the largest value can be found in the rightmost node, since greater values are stored in the right subtree. By continuously following the left child, you reach the minimum value; by following the right child, you reach the maximum value. This structure allows efficient searching of both smallest and largest elements in a BST.

3. A complete binary tree and a full binary tree may sound similar, but they are different in structure. A full binary tree means that every node has either zero or two children, and no node has only one child. A complete binary tree, on the other hand, means that all levels are completely filled except possibly the last one, and in the last level, all nodes must be filled from left to right. Therefore, a full binary tree focuses on having exact pairs of children, while a complete binary tree focuses on how the tree levels are filled.

4. When deleting a tree, the proper traversal method to use is **post-order traversal**. This is because in post-order traversal, the algorithm visits and deletes all the child nodes before deleting the parent node. This order prevents the program from losing access to child nodes that still need to be deleted. To apply this in the given Python code, a delete_tree() method can be added that first recursively deletes all children and then the parent node. This ensures that the entire tree is properly removed from memory in the correct sequence.

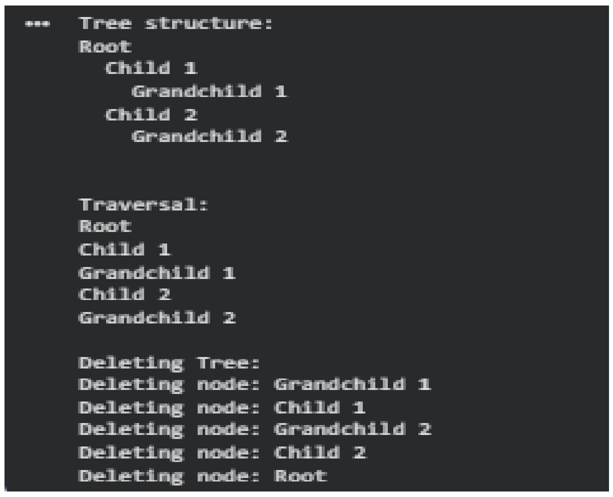


```python
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

    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):
        print(self.value)
        for child in self.children:
            child.traverse()

    def delete_tree(self):
        for child in self.children:
            child.delete_tree()
        print(f"Deleting node: {self.value}")
        self.children = None  # clear children references

    def __str__(self, level=0):
        ret = "  " * level + str(self.value) + "\n"
        for child in self.children:
            ret += child.__str__(level + 1)
        return ret
```



```python
# 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()
```

Figure 3 Screenshot of code



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


Traversal:
Root
Child 1
Grandchild 1
Child 2
Grandchild 2

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

Figure 2 Screenshot of the code

Figure 4 Screenshot of the modified source code

IV. Conclusion

This laboratory activity helped in understanding how tree structures work and how they represent hierarchical data. By implementing and modifying the Python code, the difference between various tree types and traversal methods became clearer. It also showed that post-order traversal is the proper method for deleting a tree. Overall, the activity enhanced the understanding of tree data structures and their importance in organizing and processing data efficiently.