UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 14

# Tree Structure Analysis

*Submitted by:*
Bron, Jhustine A.

*Instructor:*
Engr. Maria Rizette H. Sayo

November 4, 2025

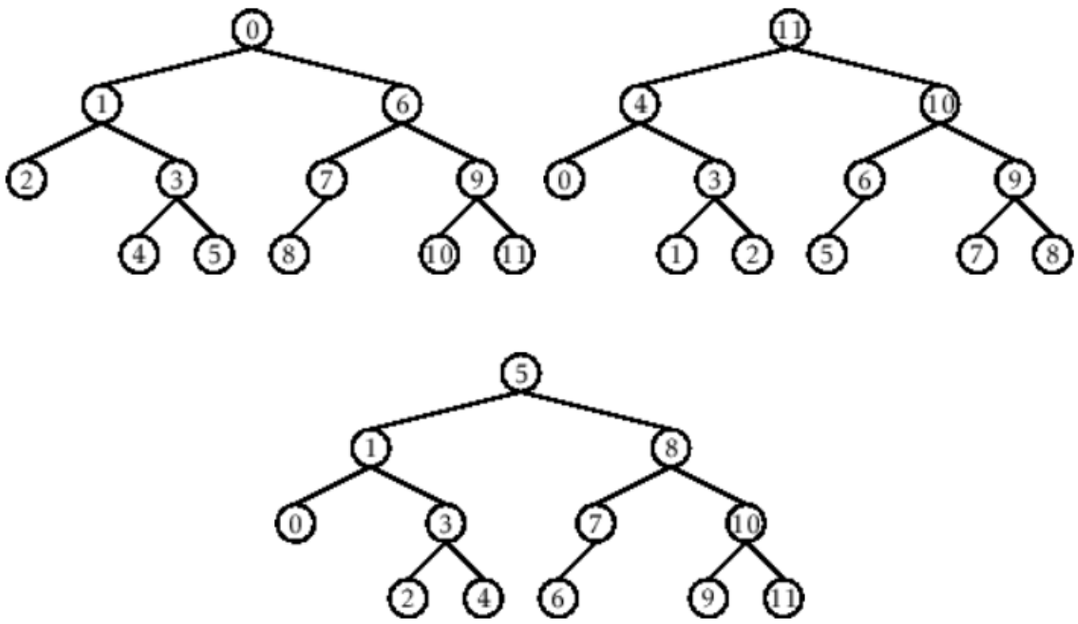
# I. Objectives

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

Output of the program:

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


Traversal:
Root
Child 2
Grandchild 2
Child 1
Grandchild 1
```

1. The main difference between a binary tree and a general tree lies in the number of children each node can have. In a binary tree, every node can have at most two children, usually referred to as the left and right child. This restriction makes binary trees more structured and suitable for applications like binary search trees (BSTs) or heaps. On the other hand, a general tree allows each node to have any number of children, which makes it more flexible but also more complex to manage and traverse. General trees are commonly used for representing hierarchical data such as organizational charts or file systems (GeeksforGeeks, 2024).

2. In a Binary Search Tree (BST), the minimum value is always found at the leftmost node, because each left child is smaller than its parent. To find it, you start from the root and keep going left until you reach a node with no left child. Conversely, the maximum value is located at the rightmost node, since each right child is greater than its parent. You find it by continuously moving to the right until there are no more right children

3. A complete binary tree and a full binary tree differ in structure and completeness. A full binary tree is one where every node has either zero or two children, meaning no node has only one child. A complete binary tree, on the other hand, is filled level by level from left to right, except possibly for the last level, which may not be completely filled. In short, a full tree focuses on the number of children per node, while a complete tree focuses on the arrangement of nodes by levels

4. To delete a tree properly, the best traversal method to use is post-order traversal, because it deletes the children before the parent node. This ensures that no node is deleted before its descendants, preventing dangling references. The modified version of the source code for deleting a tree can be written like this:

```
def delete_tree(node):
    if node is None:
        return
    for child in node.children:
        delete_tree(child)
    print(f"Deleting node: {node.value}")
    node.children = []
```

Output of the modified program:

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


Traversal:
Root
Child 2
Grandchild 2
Child 1
Grandchild 1

Deleting the tree:
Deleting node: Grandchild 1
Deleting node: Child 1
Deleting node: Grandchild 2
Deleting node: Child 2
Deleting node: Root
```

# IV. Conclusion

In conclusion, this activity helped me understand how trees work in Python, including how to build, traverse, and properly delete them. I learned that different types of trees, like binary and general trees, have their own structures and purposes. The post-order traversal method is the most effective for deleting a tree since it removes the child nodes before the parent, preventing any reference issues. Overall, this exercise showed how tree structures are essential in organizing data efficiently and how traversal methods can be applied depending on the task I need to perform.

# References

GeeksforGeeks, “Difference between General tree and Binary tree,” *GeeksforGeeks*, Jul. 12, 2025. https://www.geeksforgeeks.org/dsa/difference-between-general-tree-and-binary-tree/

“Tree traversal.” https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm