UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 14

# Tree Structure Analysis

*Submitted by:*
Delinia, Filjohn B.

*Instructor:*
Engr. Maria Rizette H. Sayo

11/09/2025

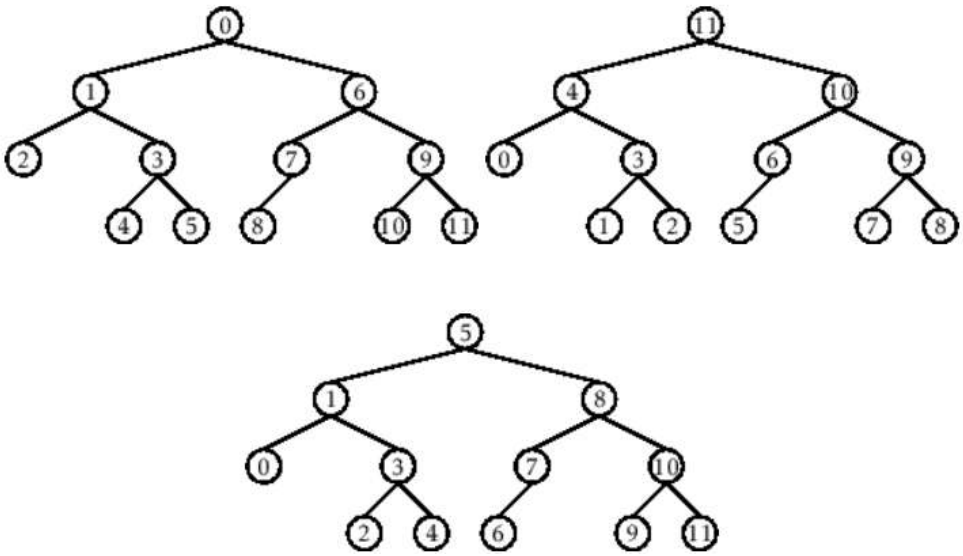
# I. Objectives

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

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


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

Figure 1. Output of the Program

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

The main difference between a binary tree and a general tree is that a binary tree allows each node to have at most two children—usually referred to as the left and right child, while a general tree can have any number of children per node.

**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 found at the leftmost node since smaller values are always stored on the left side, while the maximum value is found at the rightmost node since larger values are stored on the right side.

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

A complete binary tree differs from a full binary tree because in a complete binary tree, all levels are completely filled except possibly the last one, which is filled from left to right, while in a full binary tree, every node has either zero or two children—no node has only one child.

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

To delete a tree properly, a **post-order traversal** method should be used because it deletes the child nodes first before deleting the parent node, ensuring that all descendants are removed safely.

**Modified Code**

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 delete_tree(self):
    for child in self.children:
        child.delete_tree()
    print(f"Deleting node: {self.value}")
    self.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 and build the 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("\nDeleting tree:")
root.delete_tree()
```

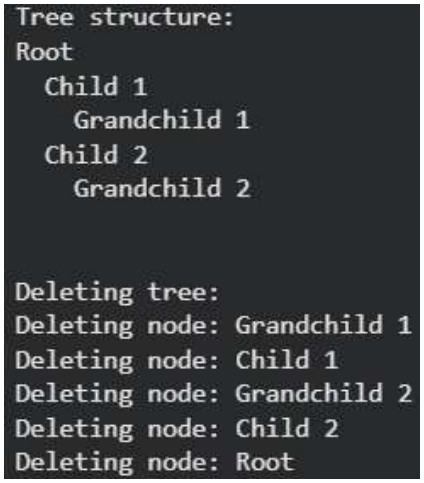


Figure 2. Modified output of the program

# IV. Conclusion

Overall, trees are flexible data structures with different types serving specific functions. A **binary tree** limits each node to two children, whereas a **general tree** can have any number of children. In a **Binary Search Tree**, the smallest value is found at the leftmost node and the largest at the rightmost node, following the BST ordering rule. **Complete** and **full binary trees** differ in structure: a complete tree fills every level except possibly the last from left to right, while a full tree requires each node to have either zero or two children. To **delete a tree correctly**, **post-order traversal** is used, removing child nodes before their parent to avoid leaving behind orphaned nodes. This method ensures that tree deletion and other operations are handled safely and efficiently.

# References

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

[2] GeeksforGeeks. (n.d.). *Tree data structure*. from https://www.geeksforgeeks.org/tree-data-structure/

[3] Programiz. (n.d.). *Binary search tree (BST) in Python*. from https://www.programiz.com/dsa/binary-search-tree

[4] TutorialsPoint. (n.d.). *Data structures – Tree*. from https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm