UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 14

Tree Structure Analysis

Submitted by:
Barbas, Steven Jade P.

Instructor:
Engr. Maria Rizette H. Sayo

November, 9, 2025

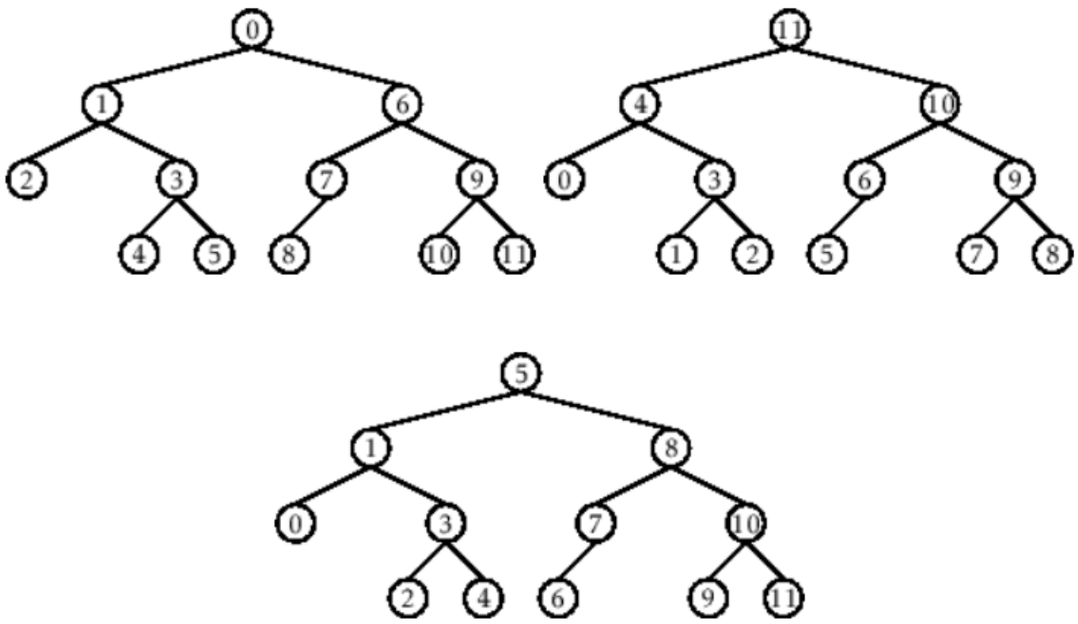
I. Objectives

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?

A binary tree can have at most two children per node, while a general tree can have any number of children. This makes binary trees more restricted in structure. General trees are more flexible for representing hierarchical data.

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, the minimum value is found by going to the leftmost node. The maximum value is found by going to the rightmost node. This is because smaller values are always on the left and larger values on the right.

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

A full binary tree has every node with either 0 or 2 children, while a complete binary tree is filled from left to right at each level. In a complete tree, all levels are fully filled except possibly the last. The last level in a complete tree must have nodes as far left as possible.

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

To delete a tree properly, you can use level-order traversal with a queue. This method processes nodes level by level from top to bottom. It ensures all nodes are visited and cleaned up systematically by adding children to the queue before deleting the current node.

```python
from collections import deque

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

    def delete_tree(self):
        # Level-order traversal using queue
        queue = deque([self])
        while queue:
            current_node = queue.popleft()
            # Add all children to the queue before deleting current node
            queue.extend(current_node.children)
            # Clear children and mark for deletion
            current_node.children = []
            print(f"Deleted node: {current_node.value}")

# 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()
```

Figure 1 Screenshot of program

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


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

Figure 2 Screenshot of program output

IV. Conclusion

In conclusion, this laboratory activity I successfully answer the questions about the code that execute tree structures as non-linear data types and implemented various traversal methods. I learned that binary trees are limited to two children per node while general trees can have unlimited children. The activity also showed that in Binary Search Trees, minimum values are found at the left nodes and values at the right nodes. Finally, I implemented level-order traversal using a queue to properly delete trees by processing nodes systematically from top to bottom.

References

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