UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Villanueva, Bryan O.

Instructor:
Engr. Maria Rizette H. Sayo

November 3, 2025

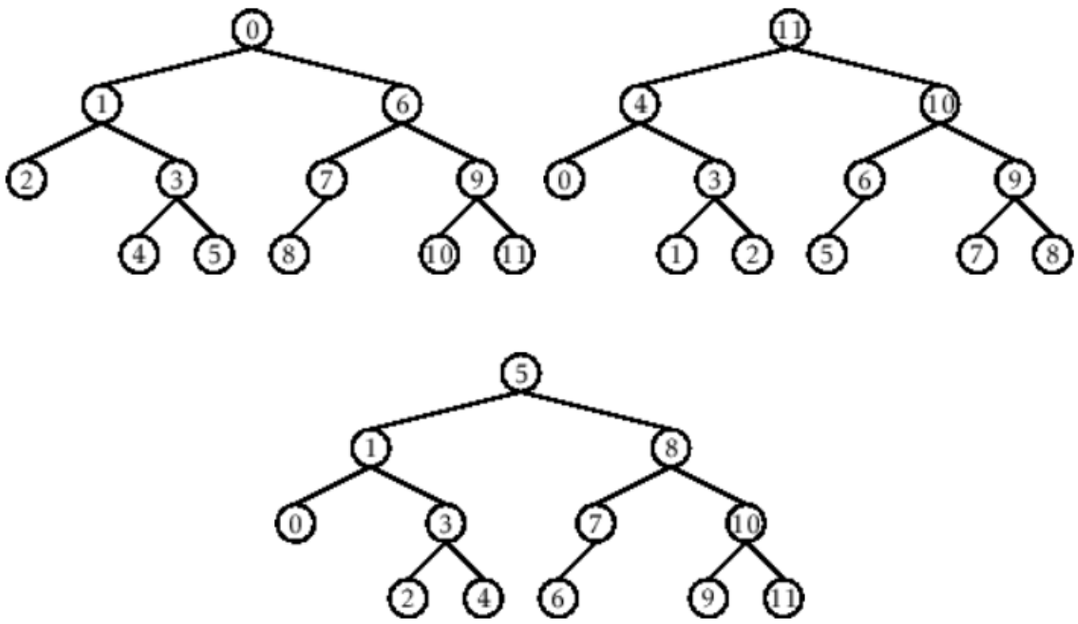
I. Objectives

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()
```

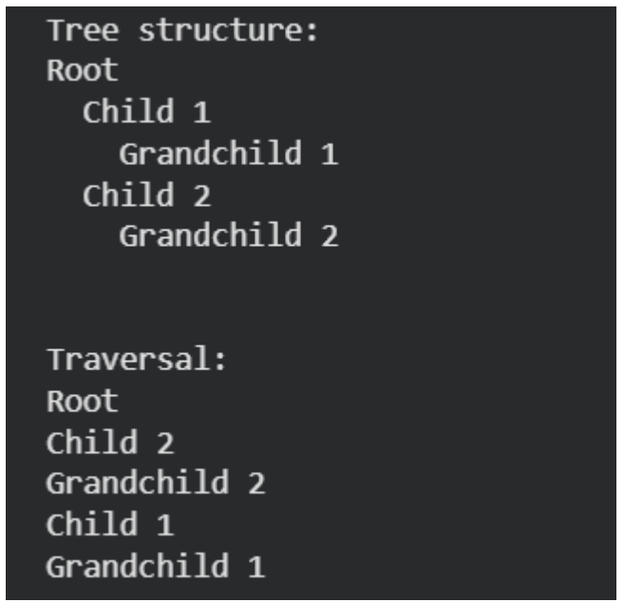
Questions:

1. When would you prefer DFS over BFS and vice versa?
2. What is the space complexity difference between DFS and BFS?
3. How does the traversal order differ between DFS and BFS?
4. When does DFS recursive fail compared to DFS iterative?

III. Results

```
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()
```

Figure 1 Input



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


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

Figure 2 Output

Questions:

1. When would you prefer DFS over BFS and vice versa?

   - In my code, the traverse() function works like DFS because it uses a stack behavior (pop()). I would prefer DFS when I want to go deep into one branch first before exploring others. This is useful when the tree is very wide or when I want to reach the deeper nodes faster. I would prefer BFS if I needed to visit the nodes level by level, but my current code does not do BFS.

2. What is the space complexity difference between DFS and BFS?

   - DFS uses less memory because it only needs to store the nodes along one path plus their children. In my code, the list works like a stack, so it only grows based on how deep the tree is. BFS uses more memory because it needs to store all nodes in a level before going to the next level. So if the tree is wide, BFS takes a lot more space.

3. How does the traversal order differ between DFS and BFS?

   - In my code, the traversal follows DFS order. It prints the node, then goes deeper into its children before moving to other branches. So it follows a “go deep first” style. BFS would print nodes level by level, starting from the root, then its children, then the grandchildren, and so on.

4. When does DFS recursive fail compared to DFS iterative?

   - DFS recursive can fail when the tree is very deep because Python has a recursion limit. If the depth is too big, it can cause a stack overflow. DFS iterative, like the one in my code, does not have this problem because it uses its own stack (a list), so it can handle deeper trees without crashing.

IV. Conclusion

In this activity, I was able to understand how a tree works as a non-linear data structure. I learned how nodes are connected and how the root becomes the main starting point of all the other parts of the tree. I also practiced the different tree traversals like pre order, in order, and post order, which helped me see how the order of visiting nodes can change the output. Overall, this lab gave me a clearer idea of how trees are used and why they are important in storing and organizing data.