



UNIVERSITY OF CALOOCAN CITY  
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 13

---

# Tree Algorithm

---

*Submitted by:*  
Catahan, Joshua A.

*Instructor:*  
Engr. Maria Rizette H. Sayo

11, 10, 2025

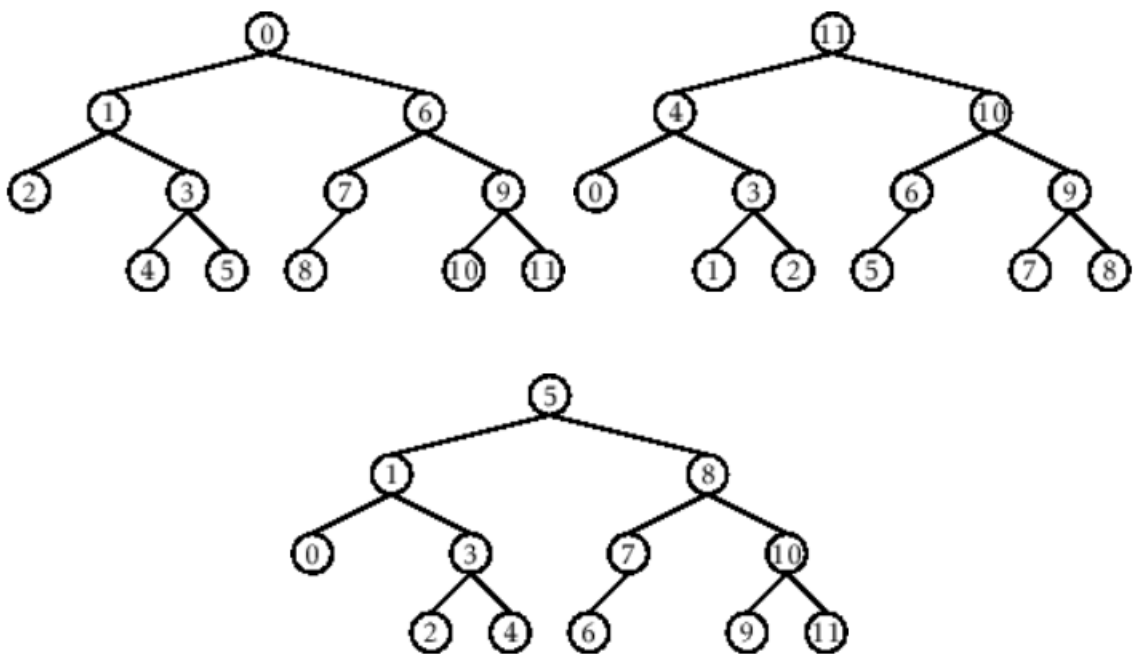
# I. Objectives

## 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

When I first ran the original program, the tree structure printed correctly, but the traversal output was right-to-left, which didn't match the order of the tree structure. I checked the code and saw that in the `traverse()` method, the children were added to the stack in the same order as they were stored, so the last child got visited first.

To fix this, I modified the traversal to reverse the children before adding them to the stack: `nodes.extend(reversed(current_node.children))`. This ensures that the first child is visited first,

giving a natural left-to-right DFS order. After this change, the program printed both the tree structure and the traversal in the correct, expected order.

```
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):
        # DFS traversal (left-to-right)
        nodes = [self]
        while nodes:
            current_node = nodes.pop()
            print(current_node.value)
            # reverse children to maintain left-to-right order
            nodes.extend(reversed(current_node.children))

    def __str__(self, level=0):
        ret = ""
        for child in self.children:
            ret += child.__str__(level + 1)
        return ret

# Create a sample 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)

# Display tree structure and traversal
print("Tree structure:")
print(root)

print("\nTraversal:")
root.traverse()
```

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

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

Figure 1: Screenshot of the Program

Answers to the questions above:

- 1. I would prefer DFS when I need to explore a tree deeply, like in maze solving or backtracking, because it uses less memory. I'd choose BFS when I need the shortest path or want to explore all nearby nodes first, like in routing or social networks.
- 2. DFS uses  $O(h)$  space for the current path, while BFS uses  $O(w)$  space for all nodes at a level, which can be much larger in wide trees.
- 3. DFS visits nodes deep into a branch before backtracking, while BFS visits all nodes level by level. DFS goes deep, BFS goes wide.
- 4. Recursive DFS can fail on very deep trees due to stack overflow, while iterative DFS avoids this by using an explicit stack.

## IV. Conclusion

In this laboratory activity, we successfully implemented a tree data structure and practiced traversing it using DFS. We encountered a minor issue with the traversal order, which we corrected by reversing the children in the stack to achieve a natural left-to-right DFS output. This lab reinforced our understanding of tree structures, traversal algorithms, and their differences, particularly between DFS and BFS. Notably, the questions in this Lab 13 were repeated from Lab

12, allowing us to review and deepen our comprehension of the concepts while applying them in a new context.

## References

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