UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

# Tree Algorithm

*Submitted by:*
Talagtag Mark Angel T.

*Instructor:*
Engr. Maria Rizette H. Sayo

November 9, 2025

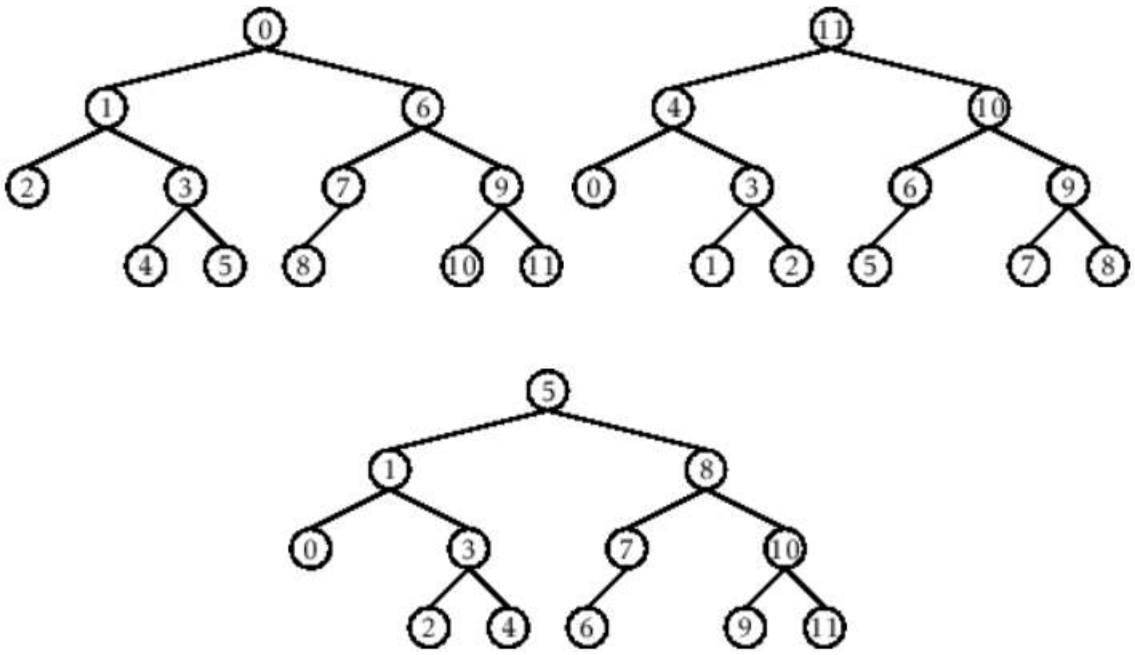
## I. Objectives

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

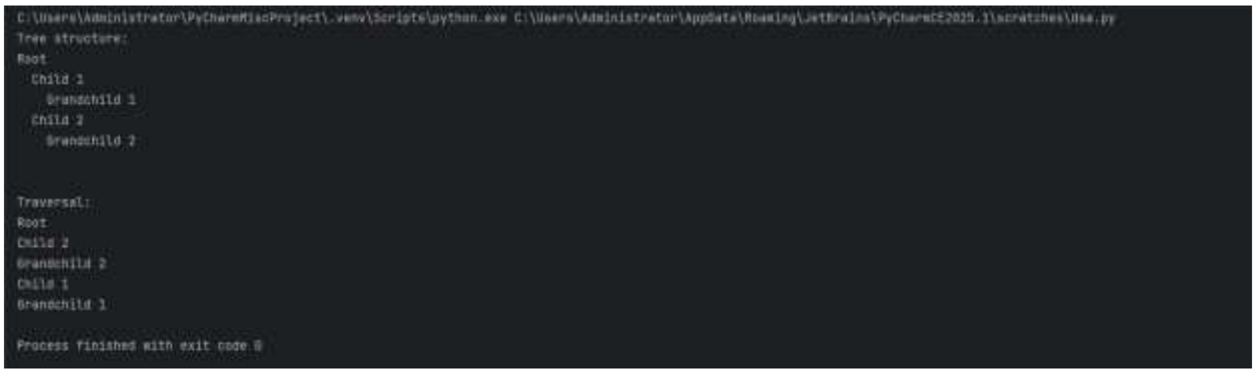
1. Tree Implementation

```
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 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



```
C:\Users\Administrator\PyCharmMiscProject\.venv\Scripts\python.exe C:\Users\Administrator\AppData\Roaming\JetBrains\PyCharmCE2025.1\scratches\dsa.py
Tree structure:
Root
  Child 1
    Grandchild 1
  Child 2
    Grandchild 2


Traversal:
Root
Child 2
Grandchild 2
Child 1
Grandchild 1

Process finished with exit code 0
```

1. You would prefer Depth-First Search (DFS) over Breadth-First Search (BFS) when you need to explore deep paths, such as in maze solving or when memory is limited, whereas BFS is better for finding the shortest path in unweighted graphs or when performing level-order traversals.

2. The space complexity difference is that DFS typically uses memory proportional to the tree's height, making it efficient for deep trees, while BFS uses memory proportional to the tree's maximum width, which can be substantial for wide trees.

3. The traversal order differs in that DFS explores as far as possible along each branch before backtracking, resulting in a deep exploration pattern, while BFS explores all nodes at the present depth level before moving to nodes at the next level, resulting in a level-by-level exploration.

4. DFS recursive can fail due to stack overflow errors when dealing with very deep trees, as it is limited by the system's recursion depth, whereas DFS iterative uses an explicit stack and is only limited by available memory, making it more suitable for large or deeply nested structures.

## IV. Conclusion

In conclusion tree traversal strategy selection depends on the specific problem requirements and constraints. Depth-First Search (DFS) excels in scenarios requiring deep path exploration and memory efficiency, particularly when solutions lie far from the root or when dealing with deeply nested structures. Its space complexity scales with tree height, making it ideal for deep but narrow trees. Conversely, Breadth-First Search (BFS) proves superior for shortest-path finding in unweighted graphs and level-based processing, though it demands more memory as it scales with tree width.

The implementation approach also significantly impacts performance and reliability. While recursive DFS offers elegant code structure, it risks stack overflow with deep trees due to recursion limits. Iterative DFS, though more complex to implement, provides greater stability and memory efficiency for large-scale applications. Ultimately, the choice between DFS and BFS—and between their recursive or iterative implementations—should be guided by the specific tree characteristics, memory constraints, and the nature of the problem being solved, with DFS favoring depth-oriented tasks and BFS excelling in breadth-focused scenarios.

## References

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

[2] Cormen Thomas H., et al. "Introduction to Algorithms," MIT Press, 2022. https://www.cs.mcgill.ca/~akroit/math/compsci/Cormen%20Introduction%20to%20Algorithms.pdf

[3] Knuth Donald E.. "The Art of Computer Programming, Volume 1: Fundamental Algorithms," Addison-Wesley, 2018. https://www.goodreads.com/book/show/1414240

[4] Sedgewick Robert, Wayne Kevin. "Algorithms," Addison-Wesley, 2020. https://algs4.cs.princeton.edu/home/

[5] Lafore Robert. "Data Structures and Algorithms in Java," Pearson Education, 2019. https://dl.ebooksworld.ir/motoman/Wiley.Data.Structures.and.Algorithms.in.Java.6th.Edition.www.EBooksWorld.ir.pdf