UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Bron, Jhustine A.

Instructor:
Engr. Maria Rizette H. Sayo

November 3, 2025

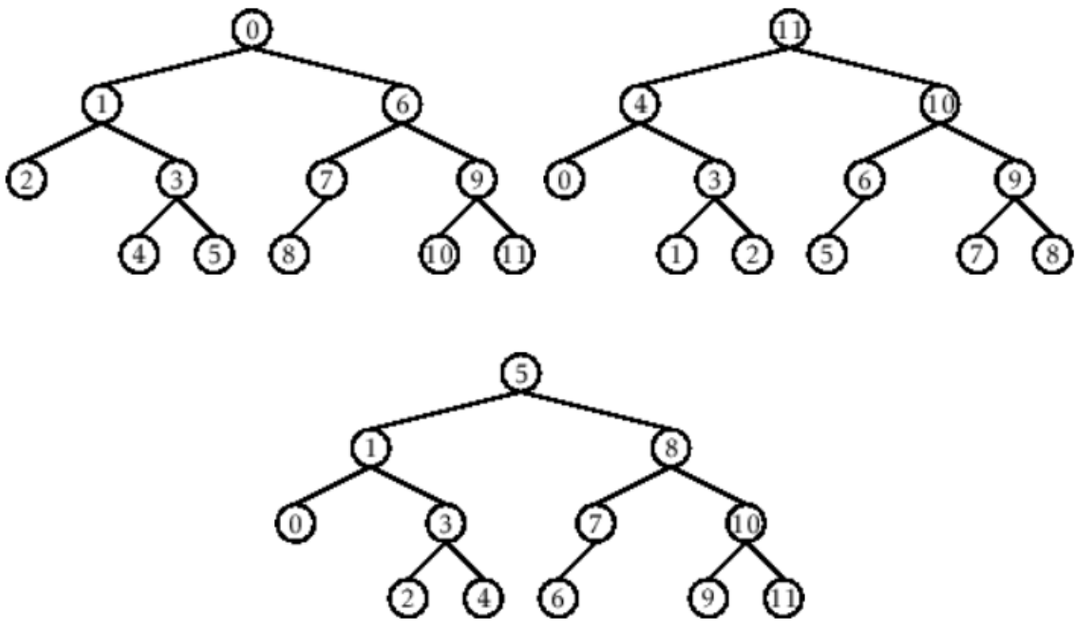
I. Objectives

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]
```

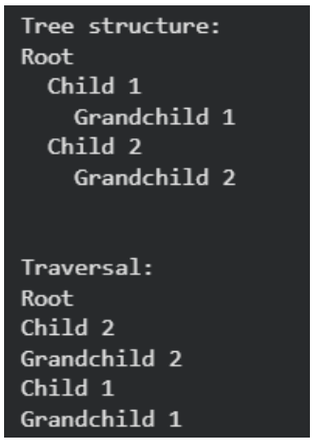
```
    def traverse(self):
        nodes = [self]
        while nodes:
            current_node = nodes.pop()
            print(current_node.value)
            nodes.extend(current_node.children)

    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

Output of the Program:



Answers:

1. I would prefer using Depth-First Search (DFS) when I need to explore a tree or graph as deeply as possible before backtracking. DFS is ideal for problems like pathfinding, puzzle solving, and topological sorting, where the goal is to reach the end of a branch or check all possible paths. On the other hand, I would use Breadth-First Search (BFS) when the shortest path or minimum number of steps is important, such as in finding the shortest route in an unweighted graph or navigating levels in a tree. In short, DFS goes "deep" first, while BFS goes "wide" first, depending on what the problem needs (GeeksforGeeks, 2024).

2. The space complexity of DFS and BFS differs mainly because of how they store nodes. DFS typically has a space complexity of O(h), where *h* is the height of the tree, since it only needs to remember nodes on the current path. BFS, however, has a space complexity of O(w), where *w* is the maximum width of the tree, because it stores all nodes at the current level before moving to the next. This means BFS generally uses more memory than DFS, especially in wide trees or graphs

3. The traversal order between DFS and BFS is also different. In DFS, nodes are visited by following one branch as deep as possible before moving to another branch. For example, it might go from root → left child → left grandchild, and only after finishing that path will it move to the next. BFS, in contrast, visits nodes level by level. It starts at the root, then moves to all immediate children, then to all grandchildren, and so on. This makes BFS more suitable for level-order processing, while DFS is better for depth-based exploration,

4. Recursive DFS can fail when the recursion goes too deep, leading to a stack overflow error, especially in very large or deep trees. This happens because each recursive call takes up space in the call stack, and once the limit is reached, the program crashes. Iterative DFS, which uses an explicit stack data structure, avoids this issue because it does not rely on the system's call stack. Therefore, in cases of deep recursion or large datasets, iterative DFS is more reliable and memory-efficient (Sedgewick & Wayne, 2022).

IV. Conclusion

In conclusion, both DFS and BFS are essential traversal techniques, each suited for different types of problems. DFS is great when I need to go deep into a structure to explore all possible paths, while BFS works best when I need to find the shortest route or handle data level by level. Their main difference lies in how they store and visit nodes, which also affects their space usage. Overall, knowing when to use each one makes it easier to design efficient algorithms and avoid issues like stack overflow or excessive memory use.

References

GeeksforGeeks, “Difference between BFS and DFS,” *GeeksforGeeks*, Nov. 01, 2025. https://www.geeksforgeeks.org/dsa/difference-between-bfs-and-dfs/

“When to Use Depth First Search vs Breadth First Search,” Jul. 23, 2024. https://hypermode.com/blog/depth-first-search-vs-breadth-first-search

GeeksforGeeks, “Time and space complexity of DFS and BFS algorithm,” *GeeksforGeeks*, Jul. 23, 2025. https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-dfs-and-bfs-algorithm/