UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Adoracion, Jerick Dave D.

Instructor:
Engr. Maria Rizette H. Sayo

November, 11, 2025

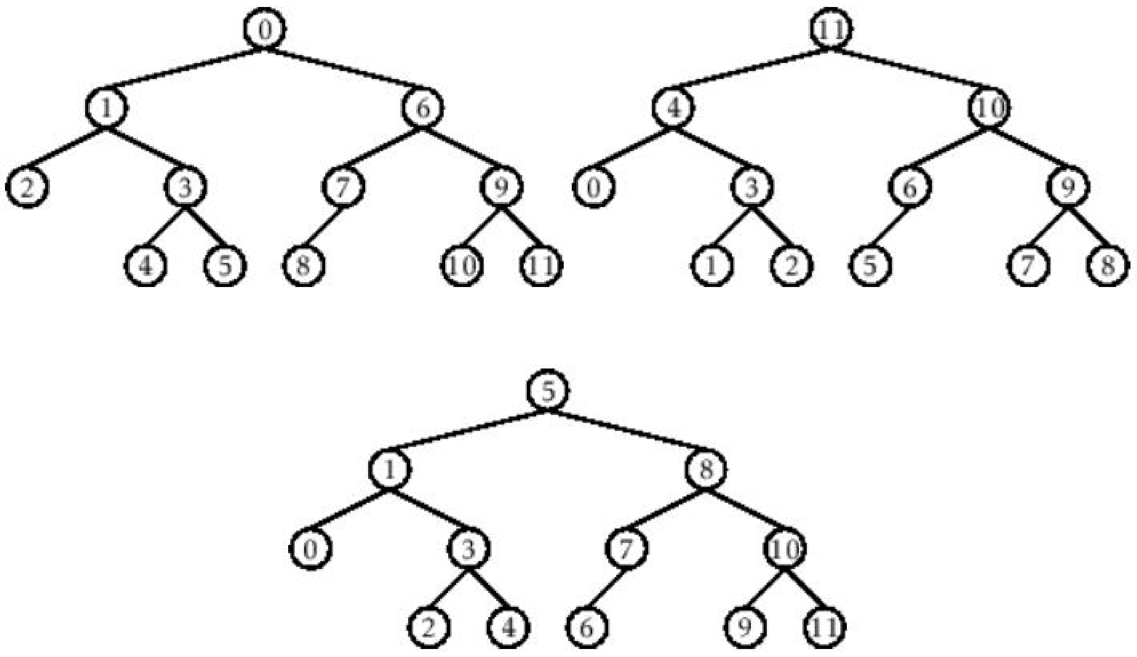
I. Objectives

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

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

DFS (Depth First Search): Prefer when the solution is deep, memory is limited, or you need to explore all possible paths (e.g., topological sort, path existence, backtracking problems).

BFS (Breadth First Search): Prefer when the solution is shallow or you need the shortest path in an unweighted graph.

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

Depth-First Search (DFS)

Space complexity: O(h)

h = the maximum depth of the search tree (or the longest path from root to leaf).

Reason:

- DFS explores one branch at a time.
- At any point, it only needs to store the nodes on the **current path** (plus some bookkeeping like visited nodes).
- So the memory used grows **linearly** with the depth of the tree/graph.

Example:
If the maximum depth of a tree is 10, DFS will at most hold 10 nodes on the call stack (in recursive form) or in its explicit stack (iterative form).

Breadth-First Search (BFS)

Space complexity: O(b^d)

b = branching factor (average number of children per node)

d = depth of the shallowest goal node

Reason:

- BFS explores **level by level**, storing **all nodes at the current frontier** before moving to the next level.
- The number of nodes in the last level (before the goal) can grow **exponentially** with depth.
- Hence, BFS usually consumes **much more memory** than DFS.

Example:
If each node has 3 children (*b = 3*) and you go to depth 5,
BFS may need to store up to 35=2433^5 = 24335=243 nodes at once.

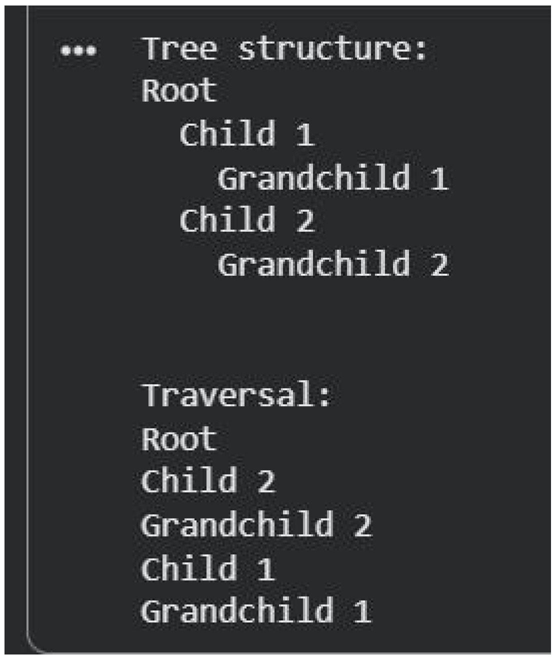
3. How does the traversal order differ between DFS and BFS?

DFS: Goes as deep as possible along a branch before backtracking (root → left → left → … → backtrack).

BFS: Explores all neighbors at the current depth before moving to the next level (level by level).

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

- **Recursive DFS** may fail due to **stack overflow** when the recursion depth exceeds the system's call stack limit (especially in very deep or large graphs).
- **Iterative DFS** using an explicit stack avoids this issue and can handle deeper graphs safely.



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


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

Figure 1: Output

IV. Conclusion

In conclusion, learning about Depth-First Search (DFS) and Breadth-First Search (BFS) has been an important part of my growth as a computer engineering student. These algorithms have taught me how to think critically about problem-solving and efficiency in computer systems. By understanding how each algorithm works, when to use them, and their differences in space and time complexity, I have gained a deeper appreciation for how computers handle data and perform searches. This knowledge has strengthened my analytical and programming skills, which are essential in designing effective and optimized solutions.

Furthermore, studying DFS and BFS has shown me the importance of choosing the right algorithm for a given problem, a skill that applies to many areas of computer engineering such as networking, artificial intelligence, and data processing. It has also helped me develop a mindset focused on logical reasoning, resource management, and adaptability. Overall, learning these algorithms has not only improved my technical understanding but also enhanced my ability to approach complex challenges systematically

References

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

[2] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA: Addison-Wesley, 2011.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA: Pearson, 2007.

[4] E. Horowitz, S. Sahni, and S. Rajasekaran, *Fundamentals of Computer Algorithms*, 2nd ed. Rockville, MD: Computer Science Press, 2008.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: MIT Press, 2022.