UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Sorellano,John Kenneth T

Instructor:
Engr. Maria Rizette H. Sayo

November 9, 2025

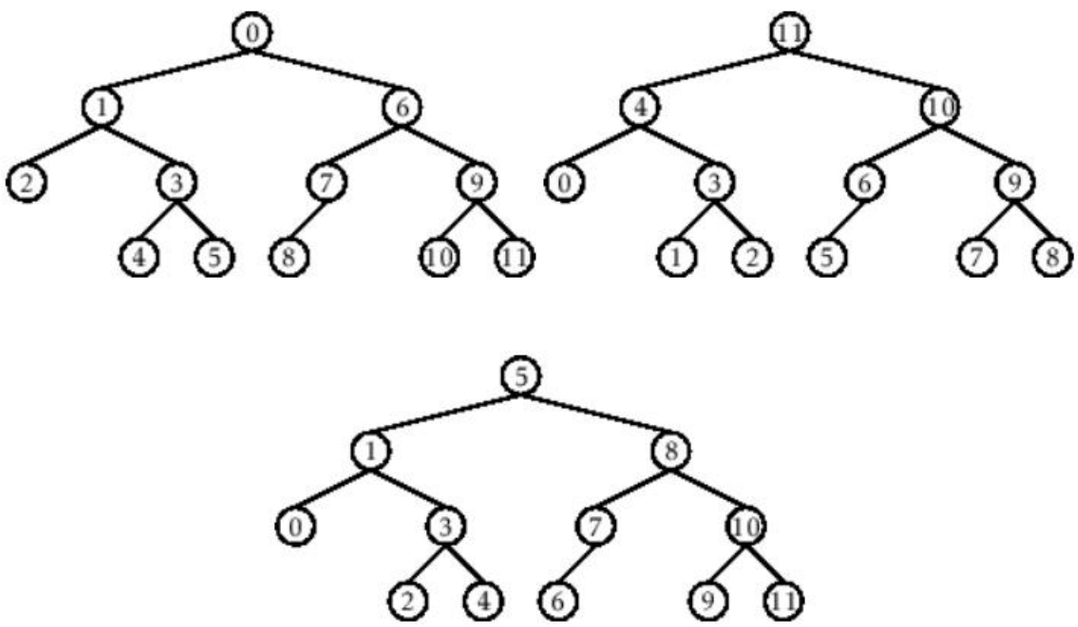
I. Objectives

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

Questions:

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

- DFS (depth-first search) is effective for addressing certain puzzles such as sorting problems and is beneficial for investigating and discovering intricate routes of information. In reverse BFS (breadth-first search) examines the information tier by tier or layer by layer beginning with the parent node; it's also preferable if you desire the quickest route in unweighted graphs.

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

- Depth first Search only stores nodes along the current path (O(h), where h is the depth), this usually utilizes and use less memory. Storing every node at the current level (O(w), where w is the maximum width), Breadth first search needs a lot more memory.

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

- Depth first search examines a branch of data to its fullest extent before returning to explore other options. each one and exploring thoroughly within the graph. The breadth-first search investigates every exploring a node's neighbors prior to progressing to the subsequent level, traversing nodes level by level.

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

- The depth first search encounters a stack overflow due to excessive recursive calls, it can result in DFS recursion to be unsuccessful. DFS iterative is more secure for extensive graphs or deep graphs because it an explicit stack can be utilized to prevent this problem.

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


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

IV. Conclusion

In summary, the graph traversal techniques DFS and BFS offer distinct advantages; BFS excels at finding the shortest path in unweighted graphs, while DFS is more effective for deep exploration that necessitates backtracking. While stack overflow can lead to DFS recursion issues in deep graphs, the space complexity of DFS is generally less than that of BFS. The issue will demonstrate which method, DFS or BFS, is superior, since both techniques excel in various circumstances. Selecting the most suitable algorithm for the task involves comprehension of its traversal sequences, space requirements, and relevant application areas.

References

- GeeksforGeeks. (n.d.). Breadth First Search (BFS) for a graph. GeeksforGeeks. Retrieved October 25, 2025, from https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
- Wikipedia contributors. (2021, May 19). Comparison of depth-first and breadth-first search. Wikipedia. Retrieved October 25, 2025, from https://en.wikipedia.org/wiki/Comparison_of_depthfirst_and_breadth-first_search