



UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Ruperto, April Anne A.

Instructor:
Engr. Maria Rizette H. Sayo

November 7, 2025

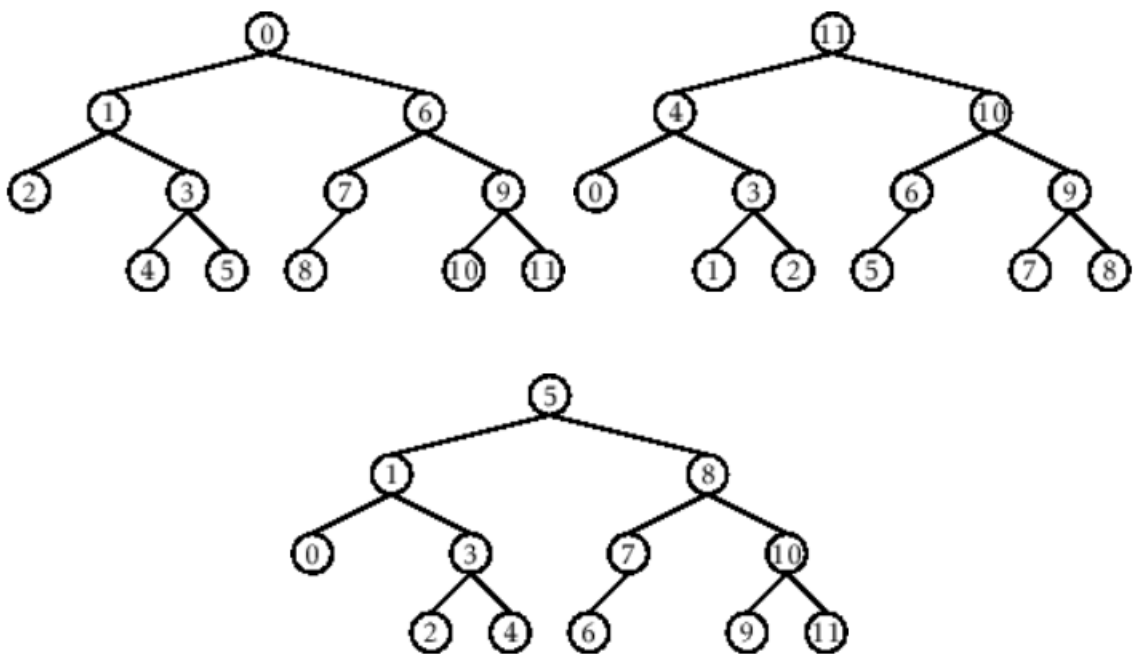
I. Objectives

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

Please follow this link: https://github.com/Ruperto-April-Anne/CPE-201L-DSA-2-A/DSA_Lab13.ipynb

Answer:

1. Depth-First Search (DFS) would be your choice when you want to delve deep into a single path before coming back, which is the case in maze solving, puzzles, or cycle detection in graphs. DFS is also the choice of method when the answer is probably at a far node, or you have to check all possible ways (like backtracking problems).
2. The space complexity of DFS is $O(h)$ where h is the maximum depth of the search tree (or $O(V)$ in the worst case), whereas BFS has a space complexity of $O(V)$ as it stores all nodes at the current level in a queue. Therefore, BFS generally consumes more memory than DFS, especially in wide or large graphs.

3. DFS tries to find a path by going deeper, hence it follows one branch until it cannot go any further and then it backtracks. On the other hand, BFS goes breadth-first, it visits all neighbors of a node before moving to the next level. That is why BFS is used to find the shortest path in unweighted graphs.
4. The recursive DFS function cannot work properly if the graph or tree is very deep as it will result in a stack overflow due to limited recursion depth. The iterative DFS, however, uses an explicit stack data structure, so it can handle larger or deeper graphs without running out of recursion limits.

IV. Conclusion

In conclusion, Depth-First Search (DFS) and Breadth-First Search (BFS) are both fundamental graph traversal methods, thus different situations require the use of one or the other. DFS is great for delving deep into paths and is commonly used in backtracking and pathfinding problems, whereas BFS is the best way to find the shortest path and nodes at the same level. Moreover, the differences between the two methods that are outlined in traversing order, memory usage and node management.

Overall, the decision whether to use DFS or BFS is determined by the problem at hand. DFS is preferable when saving memory and exploring deeply are important, while BFS should be used when the goal is to find the shortest path and the search must be complete. Knowing their advantages and disadvantages, one can decide which algorithm is the most suitable for a given computational or real-world application.

References

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