UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Villacin, Justine R.

Instructor:
Engr. Maria Rizette H. Sayo

November 03, 2025

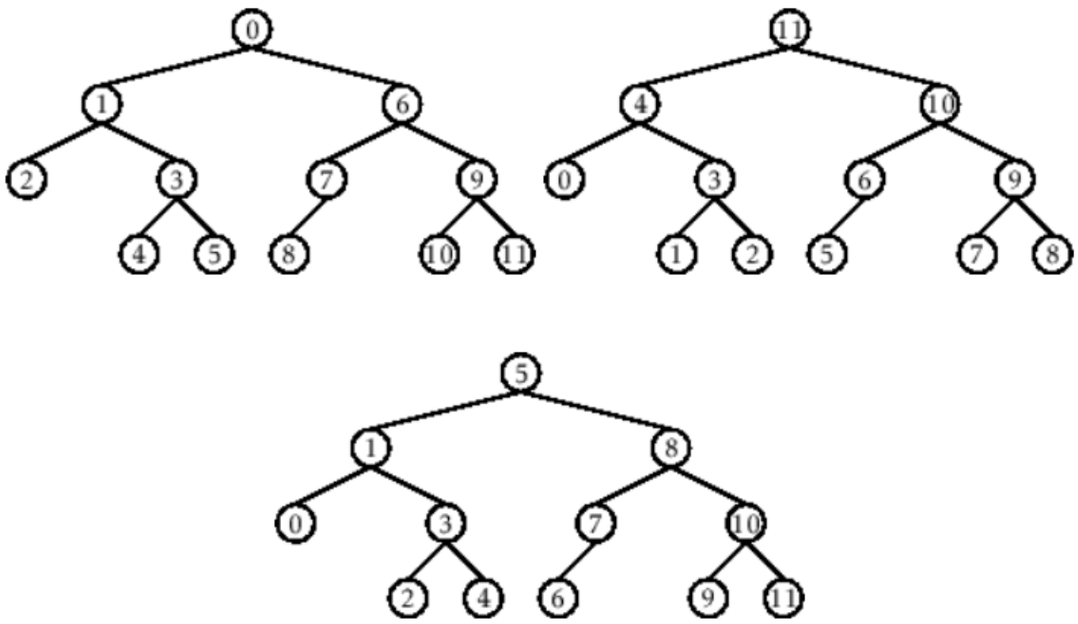
I. Objectives

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

Present the visualized procedures done. Also present the results with corresponding data visualizations such as graphs, charts, tables, or image . Please provide insights, commentaries, or explanations regarding the data. If an explanation requires the support of literature such as academic journals, books, magazines, reports, or web articles please cite and reference them using the IEEE format.

Please take note of the styles on the style ribbon as these would serve as the style format of this laboratory report. The body style is Times New Roman size 12, line spacing: 1.5. Body text should be in Justified alignment, while captions should be center-aligned. Images should be readable and include captions. Please refer to the sample below:

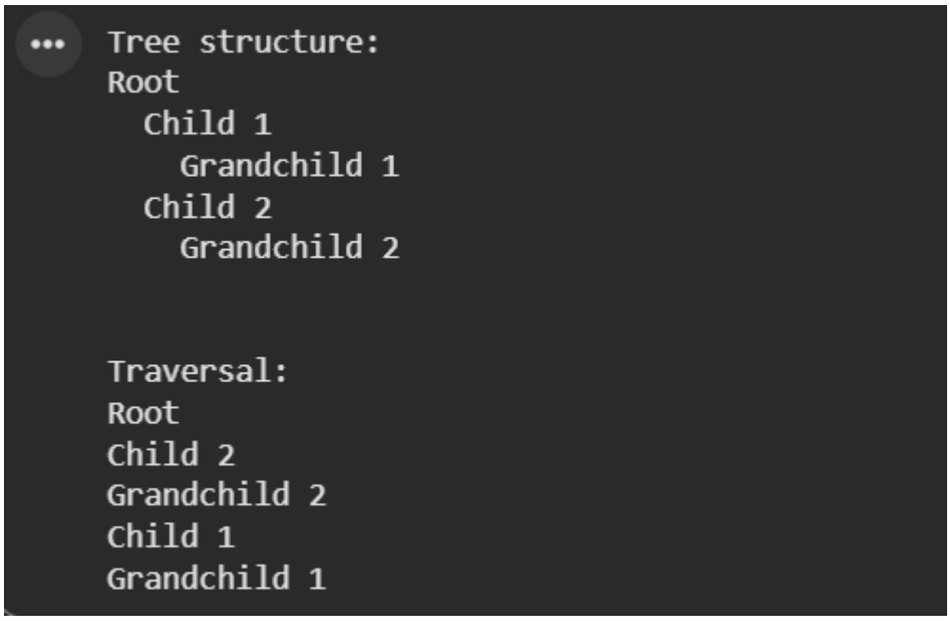


Figure 1 Output

If an image is taken from another literature or intellectual property, please cite them accordingly in the caption. Always keep in mind the Honor Code [1] of our course to prevent failure due to academic dishonesty.

ANSWERS:

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

- **Prefer DFS** when you need to explore all paths to their deepest point before backtracking. This is useful for tasks like checking if a path exists between two points, solving puzzles with multiple moves (like a maze), or traversing a file system. It is often simpler to implement recursively.
- **Prefer BFS** when you need to find the shortest path on an unweighted graph or when you need to explore nodes in order of their distance from the start. It is ideal for finding the closest related entity, like the nearest friend in a social network or the fewest moves to solve a puzzle.

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

The space complexity differs significantly in worst-case scenarios.

- **DFS:** The space complexity is **O(h)**, where h is the height of the tree. It only needs to store a single path from the root to a leaf node at any given time.
- **BFS:** The space complexity is **O(w)**, where w is the maximum width (or breadth) of the tree. In the worst case, the bottom level of a complete tree can have about n/2 nodes, which BFS must store in its queue before processing them.

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

The traversal order is the key difference.

- **DFS** goes deep down one branch of the tree before backtracking. It explores a path fully from the root to a leaf before moving to the next path. In your example tree, a DFS would print: Root, Child 1, Grandchild 1, Child 2, Grandchild 2.
- **BFS** explores the tree level by level. It visits all nodes at the present depth before moving on to nodes at the next depth level. In your example tree, a BFS would print: Root, Child 1, Child 2, Grandchild 1, Grandchild 2.

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

DFS recursive can fail due to a **stack overflow**.

- **Recursive DFS** uses the program's call stack. If the tree is very deep (has a very large height h), the recursion may become too deep and exceed the language's maximum call stack size, causing the program to crash.
- **Iterative DFS** (like the one in your code) uses a stack data structure in memory (the heap), which is typically much larger than the call stack. Therefore, it can handle much deeper trees without crashing.

IV. Conclusion

In summary, you choose DFS to go deep into a structure and BFS to explore it level by level. DFS uses less memory for deep graphs, while BFS uses more memory for wide graphs. Finally, for very deep trees, an iterative DFS is safer than a recursive one to avoid stack overflow errors.

References

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

[2] MDN Web Docs. (2023, September 6). *RangeError: Maximum call stack size exceeded*. Retrieved October 26, 2023, from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors/Too_much_recursion

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press.