UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Luminario, Venice Lou Gabrielle M.

Instructor:
Engr. Maria Rizette H. Sayo

Nov 8, 2025

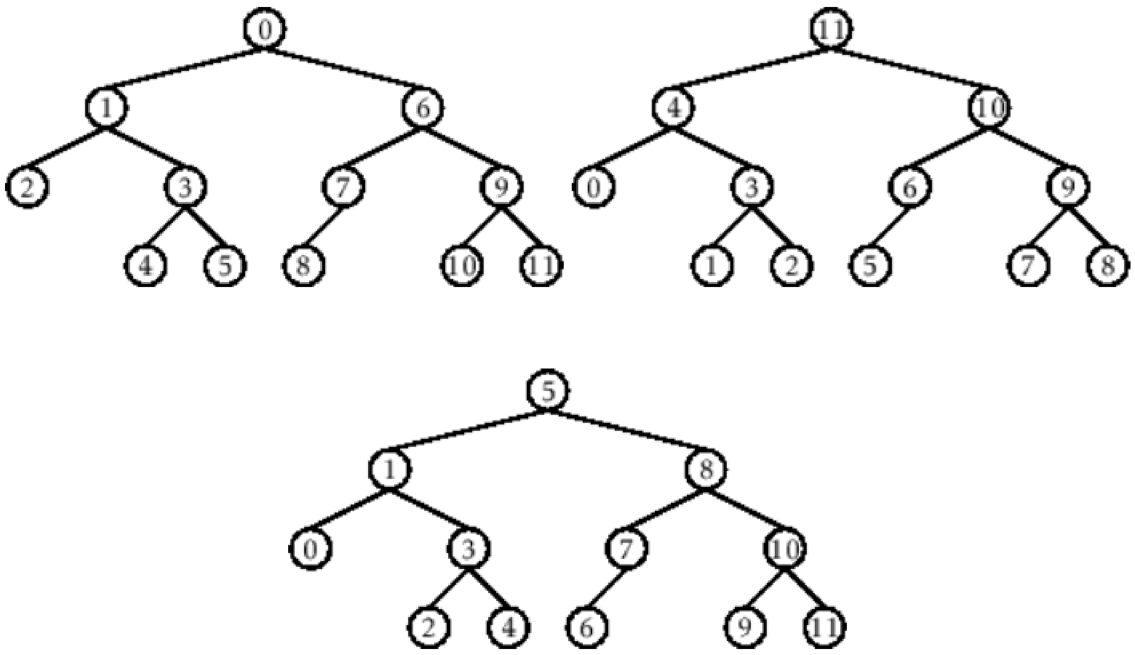
I. Objectives

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()
```

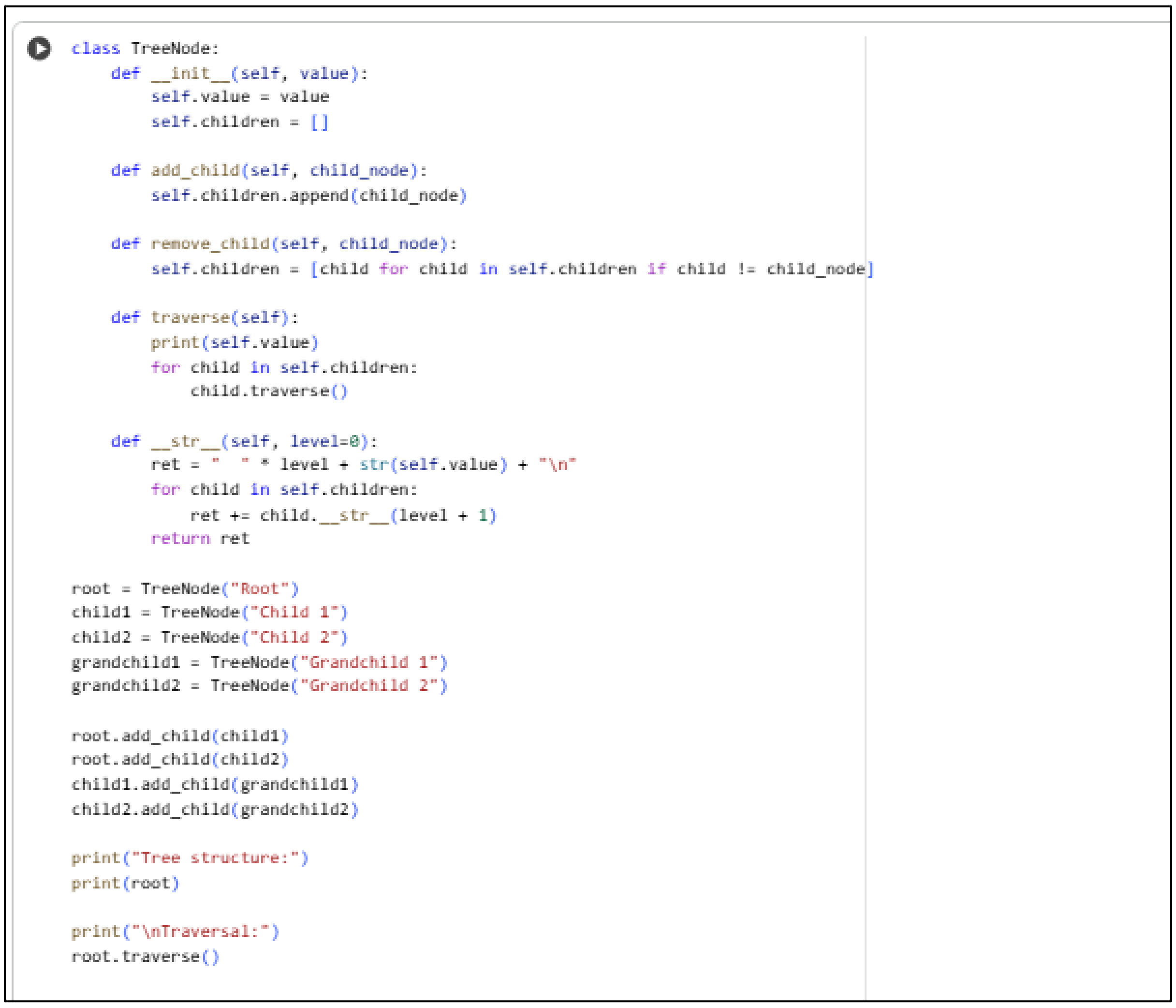
Questions:

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?

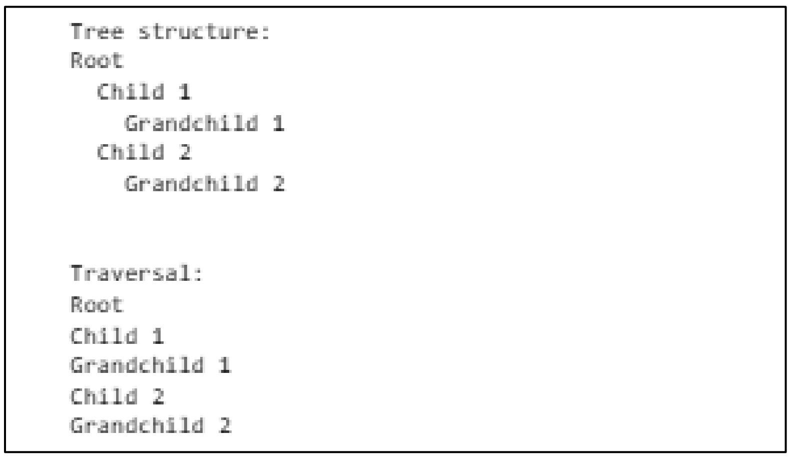
III. Results

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:



```python
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):
        print(self.value)
        for child in self.children:
            child.traverse()

    def __str__(self, level=0):
        ret = "  " * level + str(self.value) + "\n"
        for child in self.children:
            ret += child.__str__(level + 1)
        return ret

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()
```



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


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

Figure 1 Screenshot of program

Questions:

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

I prefer DFS for diving deep into one branch before others, ideal for puzzles, backtracking, or pathfinding where depth trumps breadth. BFS suits me when seeking the shortest path in unweighted graphs or processing nodes level by level, like exploring social networks. Basically, DFS plunges straight down fast, while BFS spreads wide pick based on the problem's needs.

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

DFS uses O(h) space (height of tree, up to O(n) if skewed), storing just the path in a stack. BFS needs O(w) space (widest level, up to O(n) if balanced), using a queue for all nodes at a level. In our code, iterative DFS avoids recursion limits but caps at O(h), while BFS eats more space for wide trees.

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

DFS dives deep down one family line, fully exploring from grandpa to a grandchild before backtracking, while BFS visits the family level by level first grandpa, then all his kids, then all the grandkids like a systematic reunion. DFS fully explores one path to the end before backtracking, while BFS explores all neighboring paths at the present depth before moving to the next level.

4 When does DFS recursive fail compared to DFS iterative?

Recursive DFS can crash on a super deep family tree because it's like asking too many "and then what happened?" questions in a row, overloading your memory. Iterative DFS avoids this by keeping a written list, and it's also better at tracking where you've been to avoid going in circles.

IV. Conclusion

The conclusion expresses the summary of the whole laboratory report as perceived by the authors of the report.

Choose DFS to explore deep paths and save memory, perfect for puzzles and mazes. Use BFS to find the shortest path or process by levels, ideal for social networks. DFS is a focused deep dive; BFS is a careful wide search.

References

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