UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Disomnong, Jalilah M.

Instructor:
Engr. Maria Rizette H. Sayo

November 6, 2025

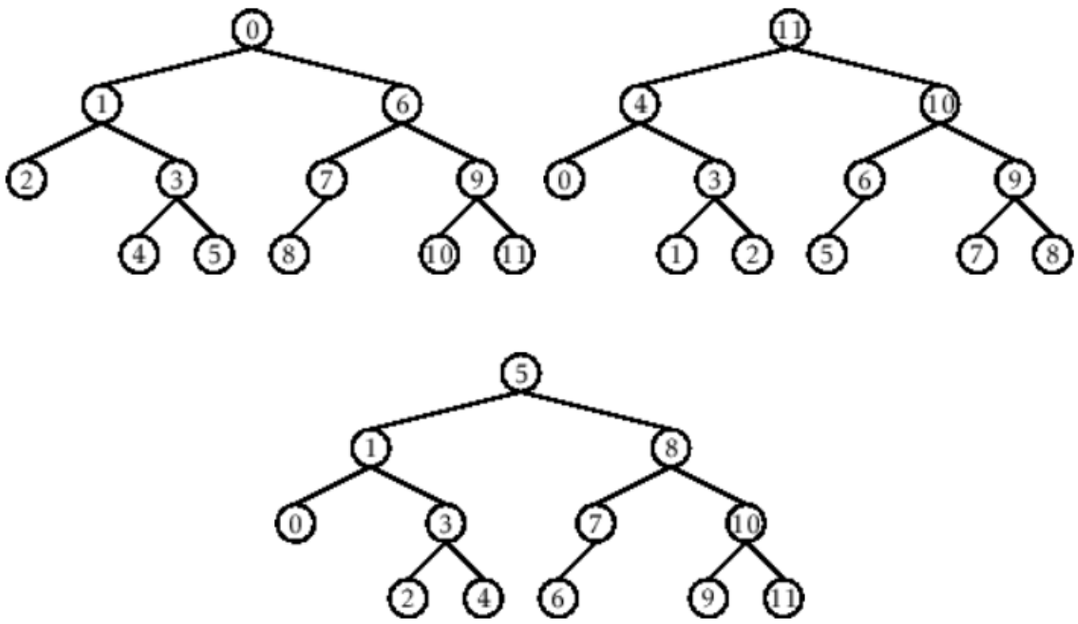
I. Objectives

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()
```

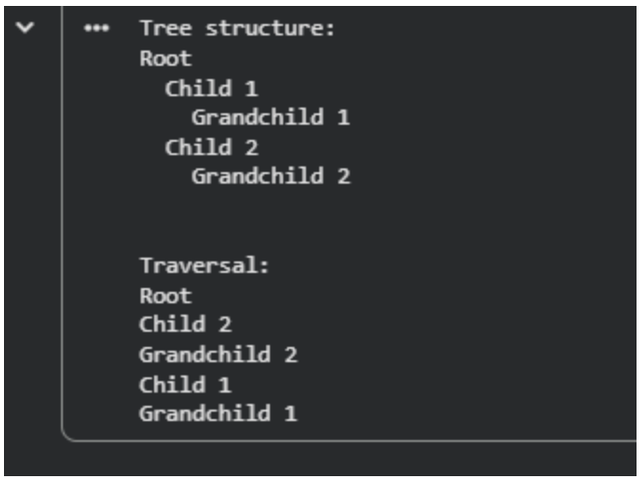
Questions:

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]
✓ 0s
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:")
```

Figure 1 Screenshot of Source code link: LAB 13 - Colab



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


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

Figure 2 Screenshot of output link: LAB 13 - Colab

In question 1, BFS and DFS are compared based on when they should be used. BFS is preferred when you want to explore nodes level by level, meaning you check everything that is close to the starting point first before going deeper. This is especially useful when you are trying to find the shortest path or the nearest result, because BFS always checks the closest connections first. On the other hand, DFS is preferred when exploring requires going as deep as possible into a branch before checking other branches. DFS is helpful for problems that involve going through

a complete path, such as solving mazes, exploring tree structures, checking if a path exists, or when backtracking is needed. According to AlgoCademy's article *"When to Use BFS vs DFS in Graph Problems: A Comprehensive Guide,"* BFS is better for shortest path problems, while DFS is better for exploring depth and structure (AlgoCademy, n.d.).

In question 2, the space complexity difference between BFS and DFS is discussed. BFS requires more memory because it stores all nodes at the current level before moving on to the next. If there are many nodes at a certain level, BFS needs to hold all of them at once, which increases memory usage. AlgoCademy explains that BFS has a space complexity of $O(b^d)$, where b is the branching factor (how many children a node can have) and d is the depth of the search (AlgoCademy, n.d.). DFS, however, only stores the nodes on the path currently being explored, making it more memory-efficient. DFS has a space complexity of $O(h)$, where h is the height or depth of the tree or graph. This means DFS usually uses less memory than BFS, which is why DFS is better for very deep graphs or trees when memory is limited.

In question 3, the difference in traversal order between DFS and BFS is explained. DFS explores nodes by going as deep as possible along one branch before backtracking to explore other branches. This means DFS follows a vertical approach, finishing one path first before trying another. In contrast, BFS explores nodes by visiting all nodes at the current level before going deeper. This makes BFS a horizontal approach where all neighbors are visited first before moving to nodes farther away. As stated by AlgoCademy, BFS expands outward from the starting point level by level, while DFS dives down until it reaches the end of a path (AlgoCademy, n.d.). Because of this, BFS is useful for level-order searches, while DFS is useful for deep exploration.

In question 4, the difference between recursive DFS and iterative DFS is explained, especially regarding when recursive DFS may fail. Recursive DFS uses the system's call stack to keep track of the function calls. If the graph or tree is too deep, the recursion can exceed the call stack limit, causing a stack overflow and making the program crash. However, iterative DFS avoids this problem by using a manual stack (a data structure in the program), which does not depend on the system's recursion limit. According to AlgoCademy, DFS can struggle in very deep or infinite graphs when recursion is used, because it depends on limited stack memory (AlgoCademy, n.d.). Therefore, recursive DFS may fail when the graph is extremely deep, but iterative DFS can handle it safely.

IV. Conclusion

In this laboratory activity, I was able to apply and understand the concept of trees as a non-linear data structure and explore how traversal techniques such as Depth-First Search (DFS)

and Breadth-First Search (BFS) work in a tree implementation. Through creating a Python program that builds a tree and performs traversal, I was able to visualize how nodes are connected and accessed based on their hierarchical relationships. By answering the guide questions, I learned when DFS and BFS should be used, how they differ in memory usage, and how their traversal orders affect problem-solving. I also gained insight into the limitations of recursive DFS, particularly in deep tree structures, and how the iterative version helps avoid recursion depth errors. Overall, this activity enhanced my understanding of tree operations, traversal techniques, and their importance in solving various computational problems, especially in Data Structures and Algorithms.

References

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

[2] AlgoCademy. (n.d.). *When to Use BFS vs DFS in Graph Problems: A Comprehensive Guide.* Retrieved from https://algocademy.com/blog/when-to-use-bfs-vs-dfs-in-graph-problems-a-comprehensive-guide/

[3] Google. (2024). Google Colaboratory. Retrieved April 18, 2024, from https://colab.research.google.com/