UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Enverzo, Kyle Andrey D.

Instructor:
Engr. Maria Rizette H. Sayo

November 8, 2025

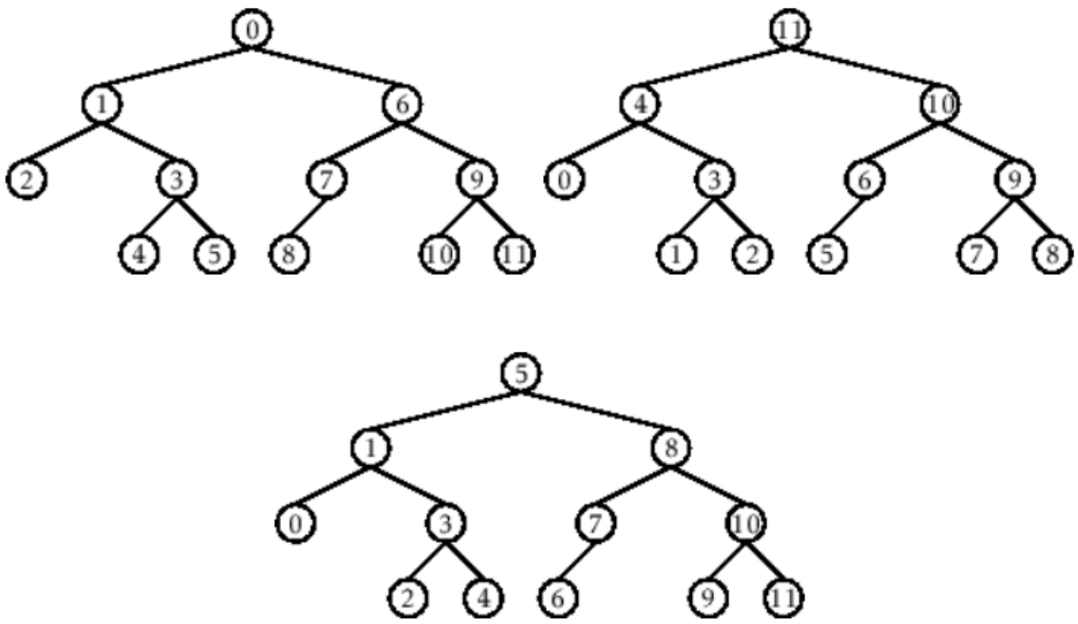
I. Objectives

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

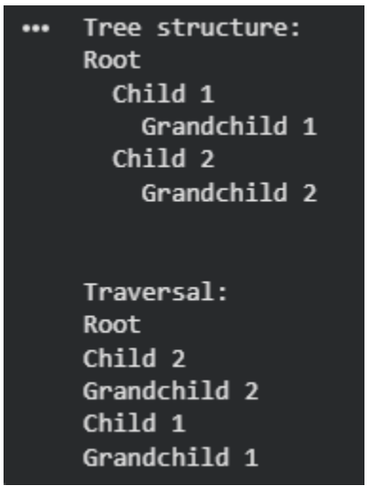


Figure 1 Screenshot of program

ANSWER

1. DFS is the method of choice in cases where you want to go as deep as possible before retracting your steps, especially if the shortest route is not your requirement, or if the space you have is limited. It is helpful for problems such as solving puzzles or deeply exploring paths. On the contrary, BFS is the preferred method when there is a need to establish the shortest path in a graph with no weights, or when the level-wise exploration of nodes is desired, like in the cases of shortest path problems or social network analysis.

2. DFS is said to generally involve a space complexity of O(h), with h being the height of the tree or the depth of the recursion stack, whereas BFS is characterized by a space complexity of O(w), wherein w refers to the maximum width of the tree or graph. In cases of wide trees or graphs, BFS could consume a lot more space, the amount of which would depend on how broad the tree is at any level, while DFS would be less space-consuming but could still suffer from stack overflow in case of very deep recursion.

3. In DFS, nodes are taking the deepest possible path and then backtracking to explore other branches, thus a depth-first traversal is achieved in which nodes are visited prior to their siblings. In BFS, nodes are explored by levels, thus all nodes at the current depth are visited before the next level is entered, hence a breadth-first traversal that treats siblings before children is obtained.

4. The use of DFS recursive function can be problematic in cases of deep trees or graphs due to its dependence on the system's call stack which can potentially lead to a stack overflow in case of

deep trees. On the other hand, DFS iterative employs an explicit stack to manage the nodes which does not expose it to the risk of stack overflow thus making it more robust for deep trees whereas it still may face memory problems in the case of extremely large graphs with a high number of nodes.

IV. Conclusion

To conclude, Depth-First Search (DFS) and Breadth-First Search (BFS) are two of the most important algorithms for tree and graph traversal, each one appropriate for a different case. When space is concerned, DFS is the winner because it occupies only a small memory area when exploring a deep but narrow structure. Consequently, it is the best choice for exhaustive searching or path-finding where depth exploration is a must. Nonetheless, if the depth of the searched area is significantly deep and the recursion is used, then there's no doubt that the overflow of the stack is a potential risk factor. BFS, however, is of great assistance in finding the shortest path and to explore the nodes level by level but this comes at the price of more memory usage because of the retention of all the nodes at a given depth. In conclusion, the decision to use either DFS or BFS rests on the specific problem features, the available memory, and the type of data being traversed.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: MIT Press, 2022.

[2] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA: Addison-Wesley, 2011
.

[3] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, *Algorithms*, New York, NY: McGraw-Hill, 2008.

[4] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*, 2nd ed. Rockville, MD: Computer Science Press, 1998.