UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Sumel, Hendrix Nathan L.

Instructor:
Engr. Maria Rizette H. Sayo

November 6, 2025

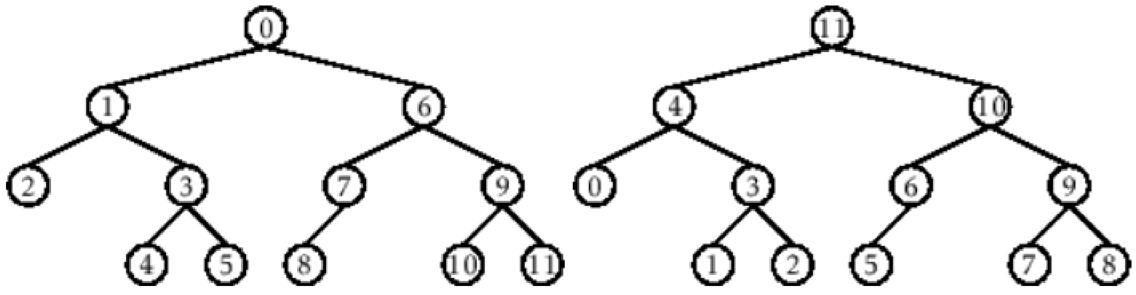
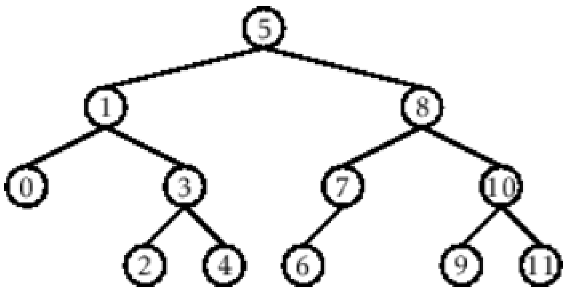
I. Objectives

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?

1. When would you prefer DFS over BFS and vice versa?
DFS is preferred when:

- Finding a path between two nodes (DFS often finds paths faster)
- Solving puzzles with one solution (like mazes)
- Topological sorting in trees
- Memory constraints (DFS uses less memory for deep trees)
- When you need to explore all possibilities (backtracking problems)

BFS is preferred when:

- Finding the shortest path in unweighted trees
- Finding the minimum number of edges between nodes
- Level-order traversal requirements
- When the solution is likely close to the root
- Web crawling or social network analysis

2. What is the space complexity difference between DFS and BFS?
DFS Space Complexity: O(h) where h is the height of the tree

- Stores one complete path from root to leaf at a time
- More memory efficient for deep, narrow trees

BFS Space Complexity: O(w) where w is the maximum width of the tree

- Stores all nodes at the current level
- Can require more memory for wide, shallow trees

Key Difference: DFS memory usage depends on tree depth, while BFS memory usage depends on tree width.

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

DFS (Depth-First Search):

- Explores as deep as possible along each branch before backtracking
- **Pre-order:** Root → Left subtree → Right subtree
- **Post-order:** Left subtree → Right subtree → Root
- Goes to the deepest level first

BFS (Breadth-First Search):

- Explores all nodes at current depth before moving deeper
- Goes level by level
- Always visits nodes in increasing order of their distance from root

Visual Example (Tree: A(B(D,E), C(F,G))):

DFS Pre-order: A → B → D → E → C → F → G
BFS: A → B → C → D → E → F → G

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

DFS Recursive fails when:

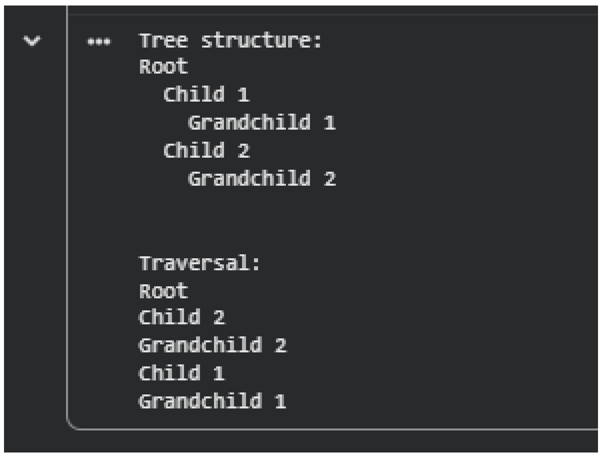
- **Stack overflow** for very deep trees (Python recursion limit ~1000)
- **Large trees** that exceed recursion depth
- **Performance-critical applications** where function call overhead matters

When explicit stack control is needed for specific ordering

DFS Iterative is better when:

- Dealing with very deep trees
- Need to avoid recursion limits
- Want explicit control over the stack
- Performance is critical
- Memory management needs to be optimized

III. Results



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


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

Image 1. Googlecolab output lab report 13 (SUMEL). Untitled10.ipynb - Colab

IV. Conclusion

This laboratory activity successfully implemented and analyzed tree data structures along with fundamental traversal algorithms, particularly Depth-First Search (DFS) and Breadth-First Search (BFS), providing comprehensive insights into their operational characteristics, performance trade-offs, and practical applications. Through hands-on implementation, we observed that DFS employs a depth-oriented approach, exploring entire branches to their fullest extent before backtracking, while BFS systematically examines all nodes at each level before progressing deeper, making it ideal for level-order processing and shortest-path problems in trees. The comparative analysis revealed that while both algorithms serve distinct purposes, their selection depends heavily on specific problem constraints: DFS excels in memory-constrained environments and when exploring deep hierarchical relationships, whereas BFS proves indispensable for scenarios requiring level-by-level analysis or finding minimum-depth solutions. The implementation also highlighted critical practical considerations, particularly the limitations of recursive DFS for deep trees due to stack overflow concerns, emphasizing the importance of iterative approaches for production-level applications. This exercise not only reinforced fundamental computer science concepts including tree structures, recursion, stack and queue operations, and complexity analysis but also provided valuable experience in algorithm selection based on real-world constraints. The knowledge gained serves as a crucial foundation for understanding more advanced tree algorithms and their applications across various domains including file systems, database indexing, network routing, and artificial intelligence, while demonstrating the importance of choosing appropriate traversal strategies based on specific problem requirements and system constraints.

References

[1] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. "Introduction to Algorithms," MIT Press, 2009.

[2] Goodrich, Michael T., Tamassia, Roberto. "Data Structures and Algorithms in Python," John Wiley & Sons, 2013.

[3] Python Software Foundation. "Python Documentation: Data Structures," 2023.

[4] Sedgewick, Robert, Wayne, Kevin. "Algorithms," Addison-Wesley Professional, 2011.

[5] Kleinberg, Jon, Tardos, Éva. "Algorithm Design," Pearson Education, 2006.

[6] Skiena, Steven S. "The Algorithm Design Manual," Springer, 2008.