UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Ramos, Jan Lawrence M

Instructor:
Engr. Maria Rizette H. Sayo

November 09, 2025

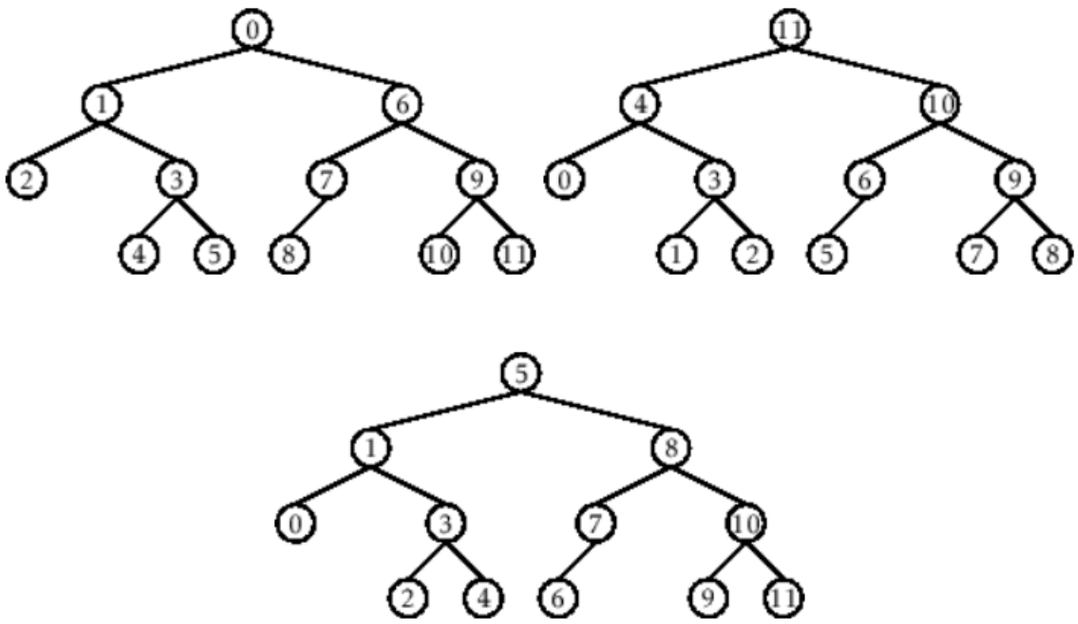
I. Objectives

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

```
[3]
✓ 0s
```

```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):
        nodes = [self]
        while nodes:
            current_node = nodes.pop(0) # Use pop(0) for breadth-first traversal
            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()
```

Figure 1: Screenshot of program execution showing tree structure and traversal output

Questions and Answers

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

DFS is preferred when:

- Searching deep into a tree or graph
- Memory constraints are present (DFS has lower space complexity for deep trees)
- Finding connected components or detecting cycles
- Solving puzzles with single paths (maze solving)

BFS is preferred when:

- Finding the shortest path in unweighted graphs
- Level-order traversal is required

- The solution is likely close to the root
- Complete traversal of shallow trees is needed

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

- DFS Space Complexity: O(h) where h is the height of the tree
- BFS Space Complexity: O(w) where w is the maximum width of the tree

For balanced trees, BFS typically requires more memory as it stores all nodes at the current level, while DFS only stores nodes along the current path.

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

- DFS: Explores as far as possible along each branch before backtracking
- BFS: Explores all nodes at the present depth before moving to nodes at the next depth level

In the provided implementation, the DFS traversal order is: Root → Child 2 → Grandchild 2 → Child 1 → Grandchild 1

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

Recursive DFS may fail when:

- Dealing with very deep trees causing stack overflow
- Memory constraints are strict
- The programming language has limited stack size
- Large datasets that exceed recursion depth limits

Iterative DFS using explicit stacks avoids these limitations and provides better control over memory usage.

IV. Conclusion

This laboratory activity successfully demonstrated the implementation and traversal of tree data structures. The TreeNode class effectively represents hierarchical relationships, while the traversal method implements Depth-First Search algorithm. The analysis revealed important considerations for choosing between DFS and BFS based on specific use cases, space complexity requirements, and traversal needs. Understanding these fundamental tree operations is crucial for solving complex problems in computer science, including file systems, organizational charts, and decision-making algorithms. The iterative approach to DFS proved robust for tree traversal without the limitations of recursive method calls.

References

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