

Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Delinia, Filjohn B.

Instructor:
Engr. Maria Rizette H. Sayo

11/09/2025

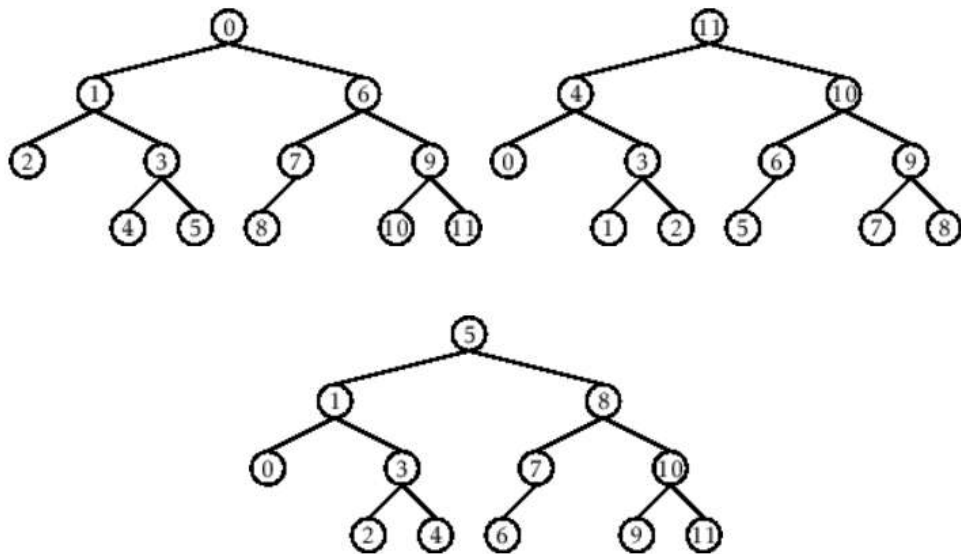
I. Objectives

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

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

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

Figure 1. Output of the program

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

I tend to use **DFS** when I need to go deep into a branch before checking others, especially if I'm searching for something far down or want to minimize memory usage. On the other hand, I choose **BFS** when I need to find the shortest route in an unweighted graph or explore nodes in order of their levels, such as in network or pathfinding problems.

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

From what I understand, **DFS** is more memory-efficient since it only keeps track of nodes along the current path, which takes about $O(d)$ space, where d is the depth. Meanwhile, **BFS** consumes more memory because it stores every node at the current level in a queue, using around $O(b^d)$ space, where b is the branching factor.

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

In my observation, **DFS** explores by diving deeply into one branch before moving to another, while **BFS** progresses gradually, visiting all nodes on one level before going to the next.

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

I realized that **recursive DFS** might fail when dealing with very deep trees or graphs because it can trigger a **stack overflow** due to recursion limits. To avoid this issue, I would use **iterative DFS** with a manual stack, which can handle deeper or larger data structures without running out of stack space.

Conclusion

To sum up, **Depth-First Search (DFS)** and **Breadth-First Search (BFS)** are both fundamental techniques for exploring graphs, each with its own strengths. **DFS** is best used when the goal is to search deeply through one path at a time and when memory resources are limited, while **BFS** works well for finding the shortest path or examining nodes level by level. Their main distinctions come from how they handle node storage, traversal order, and memory consumption. Knowing when to use each approach and being aware of the possible issues with recursive DFS, such as stack overflow in very deep structures, helps ensure efficient and effective algorithm use.

References

- [1] Co Arthur O.. “University of Caloocan City Computer Engineering Department Honor Code,” UCC-CpE Departmental Policies, 2020.
- [2] GeeksforGeeks. (2025, July 23). *Time and space complexity of DFS and BFS algorithm*.
<https://www.geeksforgeeks.org/time-and-space-complexity-of-dfs-and-bfs-algorithm/>
- [3] Codecademy. (n.d.). *Depth-First Search (DFS) algorithm explained*.
<https://www.codecademy.com/article/depth-first-search-conceptual>
- [4] Zhang, Y. (n.d.). *DFS vs BFS: Understanding the differences*.
<https://www.yingjie-zhang.com/algorithm/dfs-bfs>