UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT

Data Structure and Algorithm

Laboratory Activity No. 13

# Tree Algorithm

*Submitted by:*
Mamanao Kurt Marwin CI.

*Instructor:*
Engr. Maria Rizette H. Sayo

NOV 7 2025

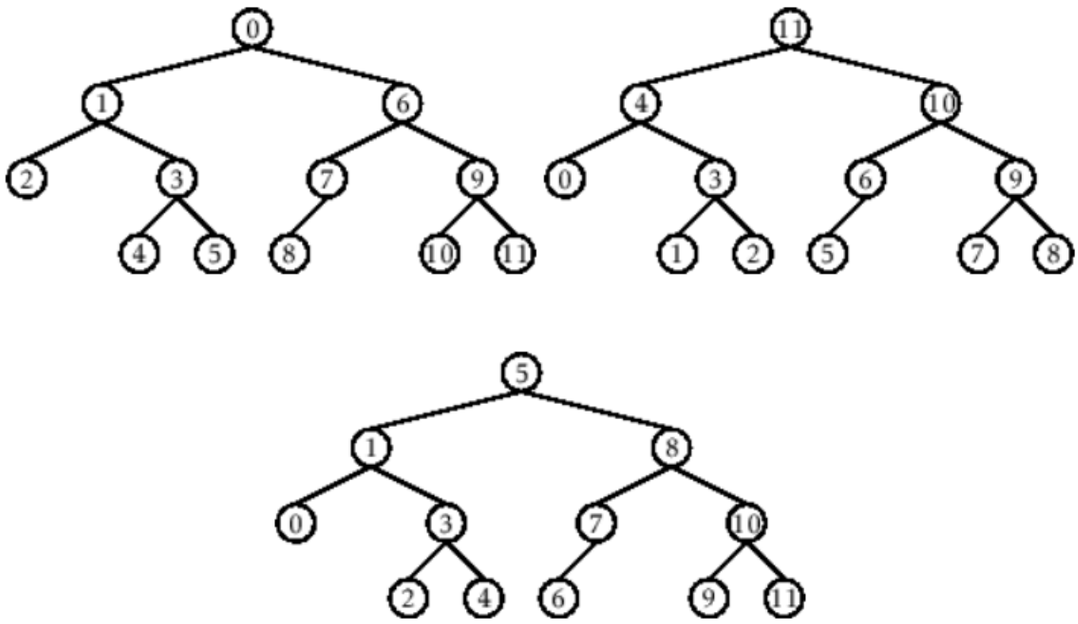
# I. Objectives

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

Questions:

**When would you prefer DFS over BFS and vice versa?**

**INPUT**

```python
# Depth First Search (DFS)
def dfs(root):
    print("DFS Traversal:")
    stack = [root]
    while stack:
        node = stack.pop()
        print(node.value)
        stack.extend(reversed(node.children))

# Breadth First Search (BFS)
def bfs(root):
    print("\nBFS Traversal:")
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.value)
        queue.extend(node.children)
```

```python
# Example Tree
root = Node("A")
b = Node("B")
c = Node("C")
d = Node("D")
e = Node("E")
f = Node("F")

root.add_child(b)
root.add_child(c)
b.add_child(d)
b.add_child(e)
c.add_child(f)

dfs(root)
bfs(root)
```

**OUTPUT**

```
DFS Traversal:
A
B
D
E
C
F

BFS Traversal:
A
B
C
D
E
F
```

**Figure 1:** Output shows DFS explores depth first (A-B-D-E-C-F) while BFS explores level by level (A-B-C-D-E-F).

I prefer **DFS** when I want to go deep into a branch before checking others, like solving puzzles or pathfinding. I use **BFS** when I want to find the shortest path or check all nodes

level by level. DFS goes deep fast, while BFS spreads wide.

**What is the space complexity difference between DFS and BFS?**
**INPUT**

```
def dfs_space(root):
    stack = [root]
    max_space = 0
    while stack:
        node = stack.pop()
        max_space = max(max_space, len(stack))
        stack.extend(node.children)
    return max_space

def bfs_space(root):
    from collections import deque
    queue = deque([root])
    max_space = 0
    while queue:
        node = queue.popleft()
        max_space = max(max_space, len(queue))
        queue.extend(node.children)
    return max_space

# Test using same tree as before
print("DFS Max Space:", dfs_space(root))
print("BFS Max Space:", bfs_space(root))
```

**OUTPUT**

```
DFS Max Space: 1
BFS Max Space: 2
```

**Figure 2:** Shows that DFS usually uses less space, while BFS uses more since it stores all nodes in the current level.

DFS saves memory because it only keeps track of the current path, while BFS uses more space because it stores all neighbors in memory. So DFS is lighter in memory but may go deeper.

**How does the traversal order differ between DFS and BFS?**
**INPUT**

```
def dfs_order(root):
    order = []
    stack = [root]
    while stack:
        node = stack.pop()
        order.append(node.value)
        stack.extend(reversed(node.children))
    return order

def bfs_order(root):
    from collections import deque
    order = []
    queue = deque([root])
    while queue:
        node = queue.popleft()
        order.append(node.value)
        queue.extend(node.children)
    return order

print("DFS Order:", dfs_order(root))
print("BFS Order:", bfs_order(root))
```

**OUTPUT**

```
DFS Order: ['A', 'B', 'D', 'E', 'C', 'F']
BFS Order: ['A', 'B', 'C', 'D', 'E', 'F']
```

DFS goes down each path until it can't, while BFS moves level by level. DFS is like exploring one tunnel fully, BFS is like exploring all tunnels at the same depth before moving deeper

**When does DFS recursive fail compared to DFS iterative:**

**INPUT**

```
# Question 4: DFS Recursive vs Iterative

# Recursive DFS
def dfs_recursive(node):
    print(node.value)
    for child in node.children:
        dfs_recursive(child)

# Iterative DFS
def dfs_iterative(root):
    stack = [root]
    while stack:
        node = stack.pop()
        print(node.value)
        stack.extend(reversed(node.children))

print("\nRecursive DFS:")
dfs_recursive(root)

print("\nIterative DFS:")
dfs_iterative(root)
```

OUTPUT

```
Recursive DFS:
A
B
D
E
C
F

Iterative DFS:
A
B
D
E
C
F
```

**Figure 4:** Recursive DFS may fail when the tree is too deep, causing a recursion limit error. Iterative DFS avoids this by using a stack manually.

Recursive DFS can fail in very large trees because Python has a recursion limit. The iterative one is safer since it uses a loop, not function calls. So I use iterative when I expect a large tree.

.

# IV. Conclusion

In this activity, I learned how tree structures work and how to use **DFS** and **BFS** to explore them. DFS goes deep first while BFS goes wide first. DFS uses less memory but can crash if the tree is too deep when using recursion. BFS is better for shortest path but uses more space.

I understood that both have their own use — DFS is faster for depth problems, BFS is better for finding shortest routes. This activity helped me see how trees are not just theory but useful for searching and organizing data.

# References

[1]GeeksforGeeks, “Difference Between BFS and DFS,” *GeeksforGeeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/