



UNIVERSITY OF CALOOCAN CITY
COMPUTER ENGINEERING DEPARTMENT



Data Structure and Algorithm

Laboratory Activity No. 13

Tree Algorithm

Submitted by:
Uy, Junichiro H.

Instructor:
Engr. Maria Rizette H. Sayo

November 09, 2025

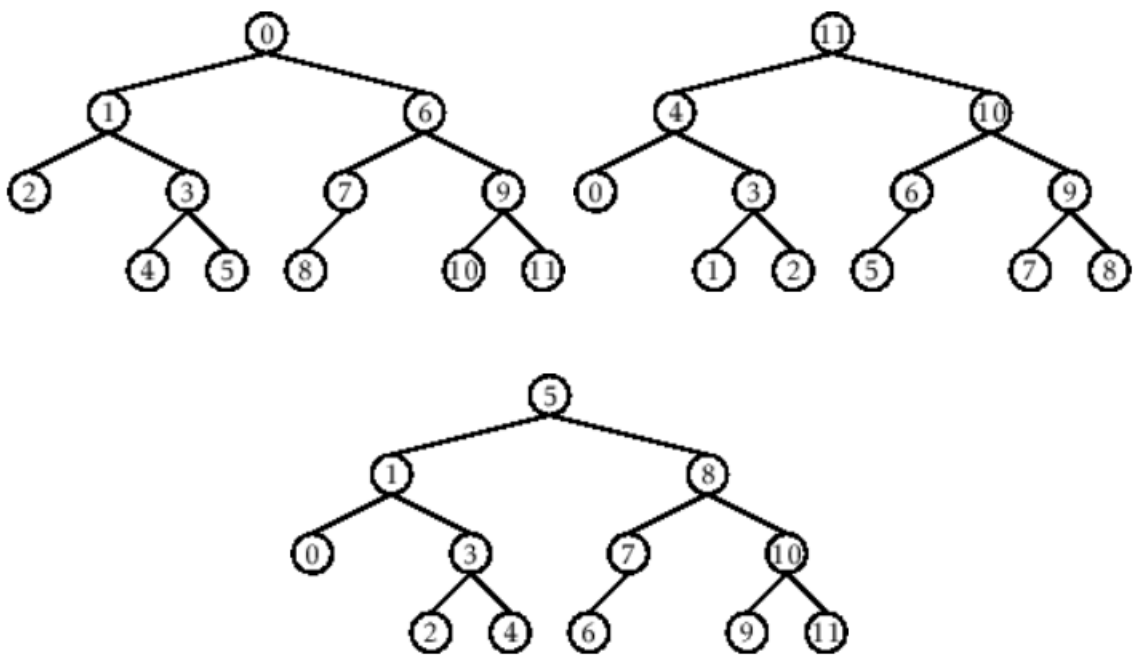
I. Objectives

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

Present the visualized procedures done. Also present the results with corresponding data

Pre-Order vs In-Order vs Post-Order:

Python Code

```

class BinaryNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def print_pre_order(node):
    if node:
        #Visit the Root
        print(node.val, end=' ')

```

```

        #Go Left
        print_pre_order(node.left)
        #Go Right
        print_pre_order(node.right)

def print_in_order(node):
    if node:
        #Go Left
        print_in_order(node.left)
        #Visit the Root
        print(node.val, end=' ')
        #Go Right
        print_in_order(node.right)

def print_post_order(node):
    if node:
        #Go Left
        print_post_order(node.left)
        #Go Right
        print_post_order(node.right)
        #Visit the Root
        print(node.val, end=' ')

root = BinaryNode(0)
root.left = BinaryNode(1)
root.right = BinaryNode(6)

root.left.left = BinaryNode(2)
root.left.right = BinaryNode(3)
root.left.right.left = BinaryNode(4)
root.left.right.right = BinaryNode(5)

root.right.left = BinaryNode(7)
root.right.right = BinaryNode(9)
root.right.left.left = BinaryNode(8)
root.right.right.left = BinaryNode(10)
root.right.right.right = BinaryNode(11)

print("Pre-order Traversal:")
print_pre_order(root)
print("\n")

print("In-order Traversal:")
print_in_order(root)
print("\n")

print("Post-order Traversal:")
print_post_order(root)
print("\n")

```

Output:

Pre-order Traversal:

0 1 2 3 4 5 6 7 8 9 10 11

In-order Traversal:

2 1 4 3 5 0 8 7 6 10 9 11

Post-order Traversal:

2 4 5 3 1 8 7 10 11 9 6 0

Questions:

1. When would you prefer DFS over BFS and vice versa?
 - Actually, it depends on your objective. We choose DFS if we want to find an existing path between nodes and explore one path until we reach a conclusion before backtracking, just like solving a maze, and if the memory is very limited, for example, if the tree is very wide but not very deep, we can use DFS to save memory usage compared to using BFS. Now, let's talk about BFS. We use it if we want to find the shortest path between nodes, we want to find nodes by level, we want to find nodes that are closest to each other, and if the tree is deep and then narrow, then we go for BFS if we have limited memory.
2. What is the space complexity difference between DFS and BFS?
 - The way I see it, the space complexity of DFS is just the tree's maximum height or $O(h)$. As for BFS, it is just the tree's maximum width or $O(w)$.
3. How does the traversal order differ between DFS and BFS?
 - I have my own take on these. Before, I like to play maze games, and there is this method we called "Wall method" and I think this is very close to DFS because in this method, I choose one wall, and stick to it until I reach the exit, just like in DFS wherein you choose one path and go as deep as possible.
 - As for BFS, I see this just like the RPG games I've played before. Usually, we need to hit all the requirements to move to the next level, just like in BFS wherein it would check all nodes in each level first before going to the next.
4. When does DFS recursive fail compared to DFS iterative?
 - After searching online, I found that recursive DFS fails if the tree is very deep. Since recursive DFS uses call stacks, it was mentioned that there is a function call limit in Python, which is approximately 1000 calls, so if your tree's height is greater than this, you will get a stack overflow. While iterative DFS uses an explicit stack (like a list) that is stored in the system's heap. The heap is limited by the total RAM, so this is larger than the call stack.

IV. Conclusion

The conclusion expresses the summary of the whole laboratory report as perceived by the authors of the report.

References

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