Data Structure and Algorithm

Laboratory Activity No. 14

Tree Structure Analysis

Submitted by:
Mamanao Kurt Marwin C.

Instructor:
Engr. Maria Rizette H. Sayo

Nov 7 2025

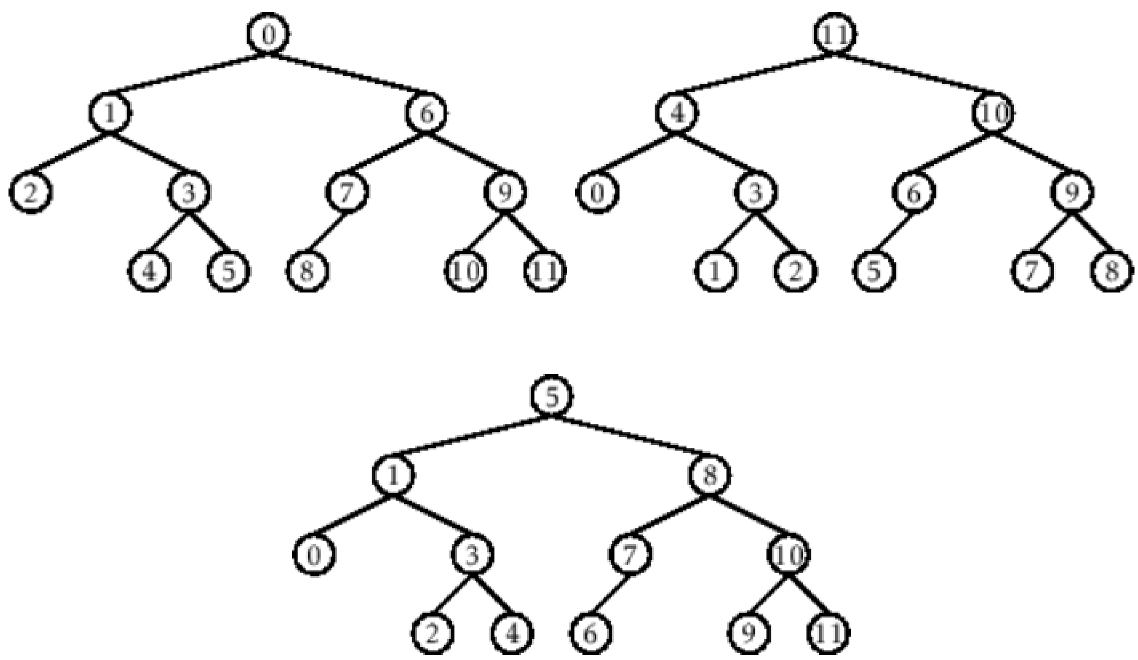
I. Objectives

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. What is the main difference between a binary tree and a general tree?
2. In a Binary Search Tree, where would you find the minimum value? Where would you find the maximum value?
3. How does a complete binary tree differ from a full binary tree?
4. What tree traversal method would you use to delete a tree properly? Modify the source codes.

III. Results

Questions:

What is the main difference between a binary tree and a general tree?

INPUT

```python
class GeneralTreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []  # Can have multiple children

class BinaryTreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None     # Only two possible children
        self.right = None

# Example
general_root = GeneralTreeNode("A")
general_root.children.extend(["B", "C", "D"])

binary_root = BinaryTreeNode("A")
binary_root.left = BinaryTreeNode("B")
binary_root.right = BinaryTreeNode("C")

print("General Tree: A -> B, C, D")
print("Binary Tree: A -> B (left), C (right)")
```

OUTPUT

```
General Tree: A -> B, C, D
Binary Tree: A -> B (left), C (right)
```

Figure 1: Shows that a general tree can have many children per node, while a binary tree can only have two

In my view, a **general tree** can have as many child nodes as needed, while a **binary tree** is limited to only two , left and right. Binary trees are used for faster searching and sorting, while general trees are for flexible hierarchies.

In a Binary Search Tree, where would you find the minimum value? Where would you find the maximum value?

INPUT

```python
class BSTNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def insert(root, value):
    if root is None:
        return BSTNode(value)
    if value < root.value:
        root.left = insert(root.left, value)
    else:
        root.right = insert(root.right, value)
    return root

def find_min(root):
    while root.left:
        root = root.left
    return root.value

def find_max(root):
    while root.right:
        root = root.right
    return root.value
```

```python
# Create BST
root = BSTNode(50)
insert(root, 30)
insert(root, 70)
insert(root, 20)
insert(root, 40)
insert(root, 80)

print("Minimum Value:", find_min(root))
print("Maximum Value:", find_max(root))
```

OUTPUT

```
Minimum Value: 20
Maximum Value: 80
```

Figure 2: The smallest value is found at the far left node, and the largest at the far right node.

I learned that in a **Binary Search Tree**, the **minimum value** is always at the **leftmost node**, and the **maximum value** is at the **rightmost node**. This is because smaller numbers go left and larger numbers go right.

How does a complete binary tree differ from a full binary tree?

INPUT

```python
# Question 3: Complete vs Full Binary Tree

# Full Binary Tree: every node has 0 or 2 children
# Complete Binary Tree: all levels are filled except maybe the last

print("Full Binary Tree Example:")
print("        A")
print("      /   \\")
print("     B     C")
print("    / \\   / \\")
print("   D   E F   G")

print("\nComplete Binary Tree Example:")
print("        A")
print("      /   \\")
print("     B     C")
print("    / \\   /")
print("   D   E F")
```

OUTPUT

```
Full Binary Tree Example:
        A
      /   \
     B     C
    / \   / \
   D   E F   G

Complete Binary Tree Example:
        A
      /   \
     B     C
    / \   /
   D   E F
```

Figure 3: Full binary tree has all nodes with 0 or 2 children.
Complete binary tree fills levels left to right but the last level may not be full**.**

To me, a full binary tree is when every node has exactly two or zero children.
A complete binary tree fills all levels except possibly the last, which fills from left to right.

What tree traversal method would you use to delete a tree properly? Modify the source codes.

INPUT

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def delete_tree(node):
    if node is None:
        return
    delete_tree(node.left)
    delete_tree(node.right)
    print(f"Deleting node: {node.value}")
    del node

# Create example tree
root = Node("A")
root.left = Node("B")
root.right = Node("C")
root.left.left = Node("D")
root.left.right = Node("E")

print("Deleting tree in post-order:")
delete_tree(root)
```

OUTPUT

```
Deleting tree in post-order:
Deleting node: D
Deleting node: E
Deleting node: B
Deleting node: C
Deleting node: A
```

Figure 4: Nodes are deleted using post-order traversal — left, right, then root — ensuring all children are deleted before their parent.

I use **post-order traversal** for deleting a tree because it deletes child nodes before the parent node. This prevents losing access to child nodes and avoids memory errors.

IV. Conclusion

In this lab, I learned that there are many types of trees, general trees, binary trees, and binary search trees, each with different rules and uses.
I also learned where to find the minimum and maximum values in a binary search tree, how to tell the difference between **full** and **complete** binary trees, and why **post-order** traversal is best for deleting a tree.

Overall, this activity helped me understand how trees are structured and how they are used to organize data in computer systems.

References

[1] Programiz, “Binary Tree and Its Types,” *Programiz*, 2024. [Online]. Available: https://www.programiz.com/dsa/binary-tree