



Data Structure and Algorithm
Laboratory Activity No. 14

Tree Structure Analysis

Submitted by:
Ramos, Jan Lawrence M.

Instructor:
Engr. Maria Rizette H. Sayo

November 09, 2025

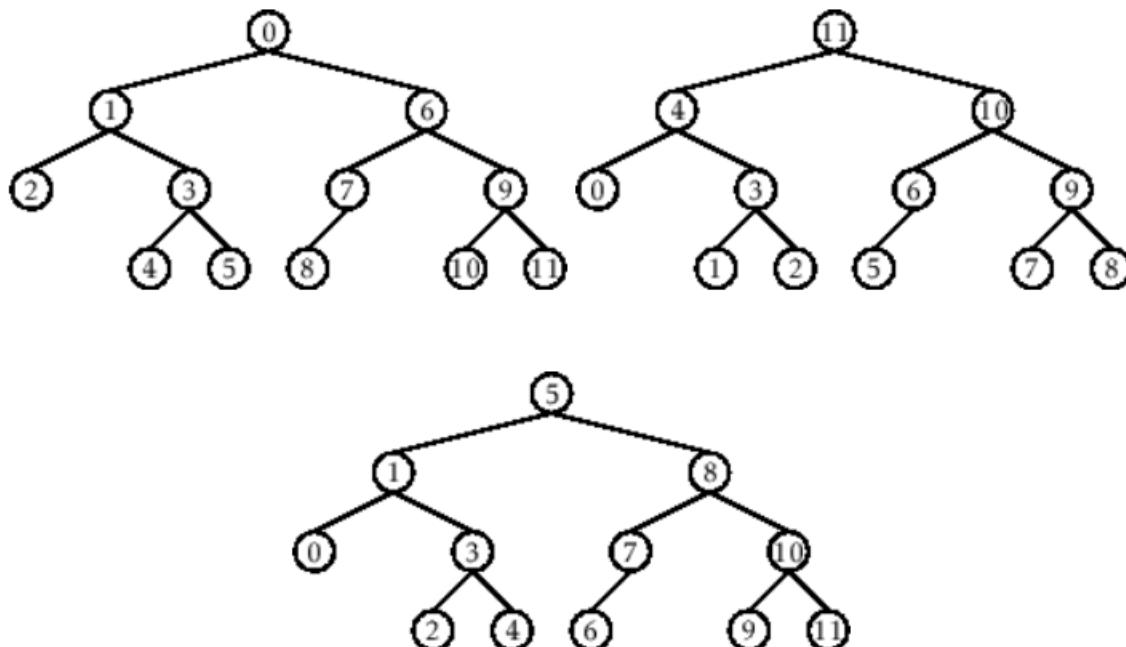
I. Objectives

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 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

```
[2] ✓ Os
  class BinaryTreeNode:
      def __init__(self, value):
          self.value = value
          self.left = None
          self.right = None

      def insert(self, value):
          if value < self.value:
              if self.left is None:
                  self.left = BinaryTreeNode(value)
              else:
                  self.left.insert(value)
          else:
              if self.right is None:
                  self.right = BinaryTreeNode(value)
              else:
                  self.right.insert(value)

      def find_min(self):
          current = self
          while current.left is not None:
              current = current.left
          return current.value

      def find_max(self):
          current = self
          while current.right is not None:
              current = current.right
          return current.value

      def delete_tree(self):
          # Post-order traversal for proper deletion
          if self.left:
              self.left.delete_tree()
          if self.right:
              self.right.delete_tree()
          print(f"Deleting node: {self.value}")
          self.left = None
          self.right = None

  # Enhanced general tree with proper deletion
  class EnhancedTreeNode(TreeNode):
      def delete_tree(self):
          # Post-order traversal for proper deletion
          for child in self.children:
              child.delete_tree()
          print(f"Deleting node: {self.value}")
          self.children = []

  # Test the enhanced implementation
  print("Enhanced Tree Implementation:")
  enhanced_root = EnhancedTreeNode("Root")
  enhanced_child1 = EnhancedTreeNode("Child 1")
  enhanced_child2 = EnhancedTreeNode("Child 2")

  enhanced_root.add_child(enhanced_child1)
  enhanced_root.add_child(enhanced_child2)

  print("Tree structure before deletion:")
  print(enhanced_root)
```

```

print("\nDeleting tree:")
enhanced_root.delete_tree()

## Binary Search Tree demonstration
print("\nBinary Search Tree Demonstration:")
bst_root = BinaryTreeNode(50)
bst_root.insert(30)
bst_root.insert(70)
bst_root.insert(20)
bst_root.insert(40)
bst_root.insert(60)
bst_root.insert(80)

print(f"Minimum value in BST: {bst_root.find_min()}")
print(f"Maximum value in BST: {bst_root.find_max()}")

print("\nDeleting Binary Search Tree:")
bst_root.delete_tree()

*** Enhanced Tree Implementation:
Tree structure before deletion:
Root
    Child 1
    Child 2

Deleting tree:
Deleting node: Child 1
Deleting node: Child 2
Deleting node: Root

Binary Search Tree Demonstration:
Minimum value in BST: 20
Maximum value in BST: 80

Deleting Binary Search Tree:
Deleting node: 20
Deleting node: 40
Deleting node: 30
Deleting node: 60
Deleting node: 80
Deleting node: 70
Deleting node: 50

```

Figure 1 and 2: This screenshots is the debugged code and output program.

Questions and Answers

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

The primary difference lies in the number of children each node can have [2]:

- **Binary Tree:** Each node can have at most two children (left and right child)
- **General Tree:** Nodes can have any number of children (zero or more)

This structural difference affects algorithms, storage requirements, and traversal methods. Binary trees are more constrained but enable efficient searching and sorting algorithms.

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

In a Binary Search Tree (BST) [3]:

- **Minimum Value:** Always located in the leftmost node of the tree
- **Maximum Value:** Always located in the rightmost node of the tree

This property stems from the BST invariant where all left descendants contain values less than the parent node, and all right descendants contain values greater than the parent node.

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

According to tree theory [2]:

- **Full Binary Tree:** Every node has either 0 or 2 children (no nodes with only one child)
- **Complete Binary Tree:** All levels are completely filled except possibly the last level, which is filled from left to right

A full binary tree focuses on the number of children per node, while a complete binary tree emphasizes the filling order of levels.

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

For proper tree deletion, **post-order traversal** is recommended because it processes children before their parent [3]. This ensures that child nodes are properly deallocated before their parent nodes, preventing dangling references or memory leaks.

The modified source code implements this approach:

python

```
def delete_tree(self):
    # Post-order traversal for proper deletion
    for child in self.children:
        child.delete_tree()
    print(f"Deleting node: {self.value}")
```

IV. Conclusion

This laboratory activity successfully demonstrated the implementation and manipulation of tree data structures, highlighting the differences between general trees and binary trees. The implementation revealed that binary trees impose stricter constraints on node children but enable more efficient search operations through Binary Search Trees. The post-order traversal method proved essential for proper tree deletion, ensuring child nodes are processed before their parents. Understanding these fundamental tree concepts and their practical implementations provides a solid foundation for solving complex hierarchical data problems in computer science applications.

References

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