

Activity No. 9.1	
Tree ADT	
<b>Course Code:</b> CPE010	<b>Program:</b> Computer Engineering
<b>Course Title:</b> Data Structures and Algorithms	<b>Date Performed:</b> 11/13/24
<b>Section:</b> CPE21S4	<b>Date Submitted:</b>
<b>Name(s):</b> Masangkay, Frederick D. Bona, Andrei Nycole So. Roallos, Jean Gabriel Vincent G. Santos, Andrei R. Zolina, Anna Marie L.	<b>Instructor:</b> Professor Maria Rizette Sayo
<b>A. Output(s) and Observation(s)</b>	
<b>Task 1: Create code in C++ that will create a tree as shown in the figure above. Use linked lists as the internal representation of this tree. Indicate your code screenshot and comments in table 9-1.</b>	
<b>SOURCE CODE:</b> <pre> #include &lt;iostream&gt; using namespace std;  // Definition of a node in the general tree struct TreeNode {     char data;     TreeNode* firstChild;     TreeNode* nextSibling;      TreeNode(char value) : data(value), firstChild(nullptr), nextSibling(nullptr) {} };  // Class to represent the general tree class GeneralTree { public:     GeneralTree() : root(nullptr) {}      // Function to set the root of the tree     void setRoot(char value) {         root = new TreeNode(value);     }      // Function to add a child to a given parent     void addChild(char parentValue, char childValue) {         TreeNode* parent = findNode(root, parentValue);         if (parent) {             TreeNode* child = new TreeNode(childValue);             if (!parent-&gt;firstChild) {                 parent-&gt;firstChild = child; // Add as first child             } else {                 TreeNode* sibling = parent-&gt;firstChild;                 while (sibling-&gt;nextSibling) { </pre>	

```

        sibling = sibling->nextSibling; // Traverse to the last sibling
    }
    sibling->nextSibling = child; // Add as next sibling
}
} else {
    cout << "Parent not found!" << endl;
}
}

// Function to perform pre-order traversal
void preOrder() {
    preOrderRec(root);
    cout << endl;
}

private:
    TreeNode* root;

    // Helper function to find a node with a specific value
    TreeNode* findNode(TreeNode* node, char value) {
        if (!node) return nullptr;
        if (node->data == value) return node;

        TreeNode* foundNode = findNode(node->firstChild, value);
        return foundNode ? foundNode : findNode(node->nextSibling, value);
    }

    // Helper function for pre-order traversal
    void preOrderRec(TreeNode* node) {
        if (node) {
            cout << node->data << " ";
            preOrderRec(node->firstChild);
            preOrderRec(node->nextSibling);
        }
    }
};

// Main function to demonstrate the general tree
int main() {
    GeneralTree tree;
    tree.setRoot('A'); // Setting 'A' as the root

    // Adding children
    char children[][2] = {
        {'A', 'B'}, {'A', 'C'}, {'A', 'D'},
        {'A', 'E'}, {'A', 'F'}, {'A', 'G'},
        {'D', 'H'}, {'E', 'I'}, {'E', 'J'},
        {'F', 'K'}, {'F', 'L'}, {'F', 'M'},
        {'G', 'N'}, {'J', 'P'}, {'J', 'Q'}
    };
};

```

```

for (const auto& pair : children) {
    tree.addChild(pair[0], pair[1]);
}

// Pre-order traversal of the tree
cout << "Pre-order Traversal: ";
tree.preOrder();
return 0;
}

```

**Table 9-1. General Tree**

**Task 2: Complete the following table:**

NODE	HEIGHT	DEPTH
A	4	0
B	3	1
C	2	1
D	2	2
E	1	2
F	1	2
G	1	2
H	1	3
I	0	3
J	0	3
K	0	3
L	0	3
M	0	3
N	0	3
P	0	4
Q	0	4

**Table 9-2. Completed Table**

**Task 3: After implementing the code for above, answer the following:**

**3.1 Given the tree diagram, find the result of the pre-order, post-order and in-order traversal strategies by hand. Include this as table 9-3 in section 6.**

<b>Pre-order</b>	A B C D H E I J P Q F K L M G N
<b>Post-order</b>	B C H D I P Q J E K L M F N G A
<b>In-order</b>	A B C D E F G H I J K L M N P Q

**Table 9-3. Traversal Strategies**

**Create a function for pre-order, post-order and in-order traversal. Make sure that each function displays an output into the console. Once you have the output, create and fill table 9-4 in section 6 so that it contains the screenshot of the function, screenshot of the output and your observations. Your observations consist of a comparison between output in #1 and the output of your functions in #2.**

```
#include <iostream>
using namespace std;

// Definition of a node in the general tree
struct TreeNode {
    char data;
    TreeNode* firstChild;
    TreeNode* nextSibling;

    TreeNode(char value) : data(value), firstChild(nullptr), nextSibling(nullptr) {}
};

// Class to represent the general tree
class GeneralTree {
public:
    GeneralTree() : root(nullptr) {}

    // Function to set the root of the tree
    void setRoot(char value) {
        root = new TreeNode(value);
    }

    // Function to add a child to a given parent
    void addChild(char parentValue, char childValue) {
        TreeNode* parent = findNode(root, parentValue);
        if (parent) {
            TreeNode* child = new TreeNode(childValue);
            if (!parent->firstChild) {
                parent->firstChild = child; // Add as first child
            } else {
                TreeNode* sibling = parent->firstChild;
                while (sibling->nextSibling) {
                    sibling = sibling->nextSibling; // Traverse to the last sibling
                }
                sibling->nextSibling = child; // Add as next sibling
            }
        } else {
            cout << "Parent not found!" << endl;
        }
    }
};
```

```

    }
}

// Function to perform pre-order traversal
void preOrder() {
    cout << "Pre-order Traversal: ";
    preOrderRec(root);
    cout << endl;
}

// Function to perform post-order traversal
void postOrder() {
    cout << "Post-order Traversal: ";
    postOrderRec(root);
    cout << endl;
}

// Function to perform in-order traversal
void inOrder() {
    cout << "In-order Traversal: ";
    inOrderRec(root);
    cout << endl;
}

private:
    TreeNode* root;

    // Helper function to find a node with a specific value
    TreeNode* findNode(TreeNode* node, char value) {
        if (!node) return nullptr;
        if (node->data == value) return node;

        TreeNode* foundNode = findNode(node->firstChild, value);
        return foundNode ? foundNode : findNode(node->nextSibling, value);
    }

    // Helper function for pre-order traversal
    void preOrderRec(TreeNode* node) {
        if (node) {
            cout << node->data << " "; // Visit the current node
            preOrderRec(node->firstChild); // Visit first child
            preOrderRec(node->nextSibling); // Visit next sibling
        }
    }

    // Helper function for post-order traversal
    void postOrderRec(TreeNode* node) {
        if (node) {
            postOrderRec(node->firstChild); // Visit first child
            postOrderRec(node->nextSibling); // Visit next sibling
            cout << node->data << " "; // Visit the current node
        }
    }
}

```

```

    }
}

// Helper function for in-order traversal
void inOrderRec(TreeNode* node) {
    if (node) {
        inOrderRec(node->firstChild); // Visit first child
        cout << node->data << " "; // Visit current node
        inOrderRec(node->nextSibling); // Visit next sibling
    }
}
};

// Main function to demonstrate the general tree
int main() {
    GeneralTree tree;
    tree.setRoot('A'); // Setting 'A' as the root

    // Adding children
    char children[][2] = {
        {'A', 'B'}, {'A', 'C'}, {'A', 'D'},
        {'A', 'E'}, {'A', 'F'}, {'A', 'G'},
        {'D', 'H'}, {'E', 'I'}, {'E', 'J'},
        {'F', 'K'}, {'F', 'L'}, {'F', 'M'},
        {'G', 'N'}, {'J', 'P'}, {'J', 'Q'}
    };

    for (const auto& pair : children) {
        tree.addChild(pair[0], pair[1]);
    }

    // Perform traversals
    tree.preOrder(); // Output pre-order traversal
    tree.postOrder(); // Output post-order traversal
    tree.inOrder(); // Output in-order traversal

    return 0;
}

```

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  // Definition of a node in the general tree
5  struct TreeNode {
6      char data;
7      TreeNode* firstChild;
8      TreeNode* nextSibling;
9
10     TreeNode(char value) : data(value), firstChild(nullptr), nextSibling(nullptr) {}
11 };
12
13 // Class to represent the general tree
14 class GeneralTree {
15 public:
16     GeneralTree() : root(nullptr) {}
17
18     // Function to set the root of the tree
19     void setRoot(char value) {
20         root = new TreeNode(value);
21     }
22
23     // Function to add a child to a given parent
24     void addChild(char parentValue, char childValue) {
25         TreeNode* parent = findNode(root, parentValue);
26         if (parent) {
27             TreeNode* child = new TreeNode(childValue);
28             if (!parent->firstChild) {
29                 parent->firstChild = child; // Add as first child
30             } else {
31                 TreeNode* sibling = parent->firstChild;
32                 while (sibling->nextSibling) {
33                     sibling = sibling->nextSibling; // Traverse to the last sibling
34                 }
35                 sibling->nextSibling = child; // Add as next sibling
36             }
37         } else {
38             cout << "Parent not found!" << endl;
39         }
40     }
41
42     // Function to perform pre-order traversal
43     void preOrder() {
44         cout << "Pre-order Traversal: ";
45         preOrderRec(root);
46         cout << endl;
47     }
```

```

49 // Function to perform post-order traversal
50 void postOrder() {
51     cout << "Post-order Traversal: ";
52     postOrderRec(root);
53     cout << endl;
54 }
55
56 // Function to perform in-order traversal
57 void inOrder() {
58     cout << "In-order Traversal: ";
59     inOrderRec(root);
60     cout << endl;
61 }
62
63 private:
64     TreeNode* root;
65
66 // Helper function to find a node with a specific value
67 TreeNode* findNode(TreeNode* node, char value) {
68     if (!node) return nullptr;
69     if (node->data == value) return node;
70
71     TreeNode* foundNode = findNode(node->firstChild, value);
72     return foundNode ? foundNode : findNode(node->nextSibling, value);
73 }
74
75 // Helper function for pre-order traversal
76 void preOrderRec(TreeNode* node) {
77     if (node) {
78         cout << node->data << " "; // Visit the current node
79         preOrderRec(node->firstChild); // Visit first child
80         preOrderRec(node->nextSibling); // Visit next sibling
81     }
82 }
83
84 // Helper function for post-order traversal
85 void postOrderRec(TreeNode* node) {
86     if (node) {
87         postOrderRec(node->firstChild); // Visit first child
88         postOrderRec(node->nextSibling); // Visit next sibling
89         cout << node->data << " "; // Visit the current node
90     }
91 }

```



```

93 // Helper function for in-order traversal
94 void inOrderRec(TreeNode* node) {
95     if (node) {
96         inOrderRec(node->firstChild); // Visit first child
97         cout << node->data << " "; // Visit current node
98         inOrderRec(node->nextSibling); // Visit next sibling
99     }
100 }
101 };
102
103 // Main function to demonstrate the general tree
104 int main() {
105     GeneralTree tree;
106     tree.setRoot('A'); // Setting 'A' as the root
107
108     // Adding children
109     char children[][2] = {
110         {'A', 'B'}, {'A', 'C'}, {'A', 'D'},
111         {'A', 'E'}, {'A', 'F'}, {'A', 'G'},
112         {'D', 'H'}, {'E', 'I'}, {'E', 'J'},
113         {'F', 'K'}, {'F', 'L'}, {'F', 'M'},
114         {'G', 'N'}, {'J', 'P'}, {'J', 'Q'}
115     };
116
117     for (const auto& pair : children) {
118         tree.addChild(pair[0], pair[1]);
119     }
120
121     // Perform traversals
122     tree.preOrder(); // Output pre-order traversal
123     tree.postOrder(); // Output post-order traversal
124     tree.inOrder(); // Output in-order traversal
125
126     return 0;
127 }

```

```

126
input
Pre-order Traversal: A B C D H E I J P Q F K L M G N
Post-order Traversal: H Q P J I M L K N G F E D C B A
In-order Traversal: B C H D I P Q J E K L M F N G A

...Program finished with exit code 0
Press ENTER to exit console.

```

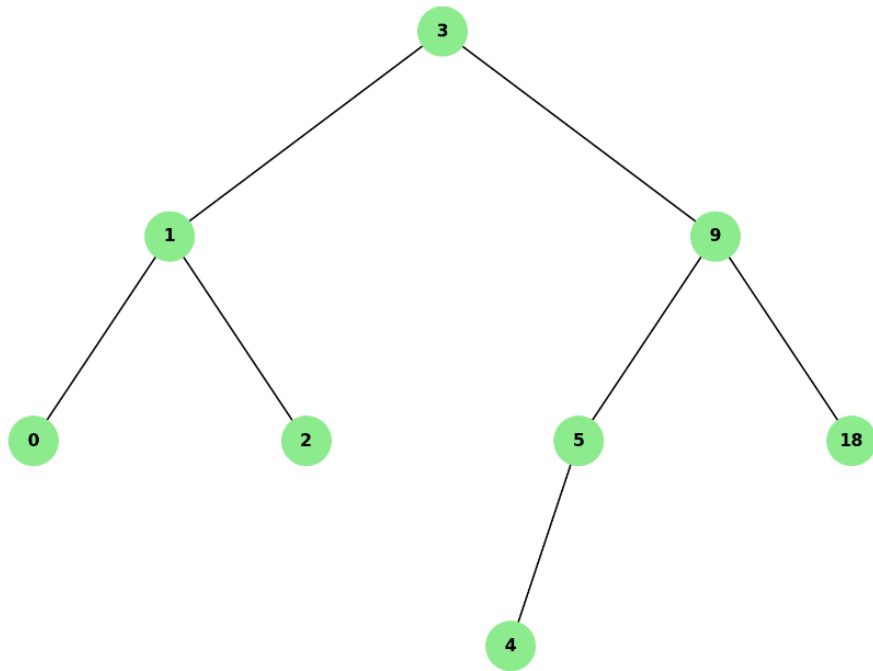
Observation	In this code, the pre-order traversal visits each node before its children, giving A B C D H E I J P Q F K L M G N. The post-order traversal visits all children before the node itself, producing H D I P Q J E K L M F N G C B A. The in-order function is unconventional for a general tree, as it visits each node's first child, the node, then siblings, resulting in the same output as post-order. This in-order approach fits this structure but isn't standard for non-binary trees.
-------------	--

## B. Answers to Supplementary Activity

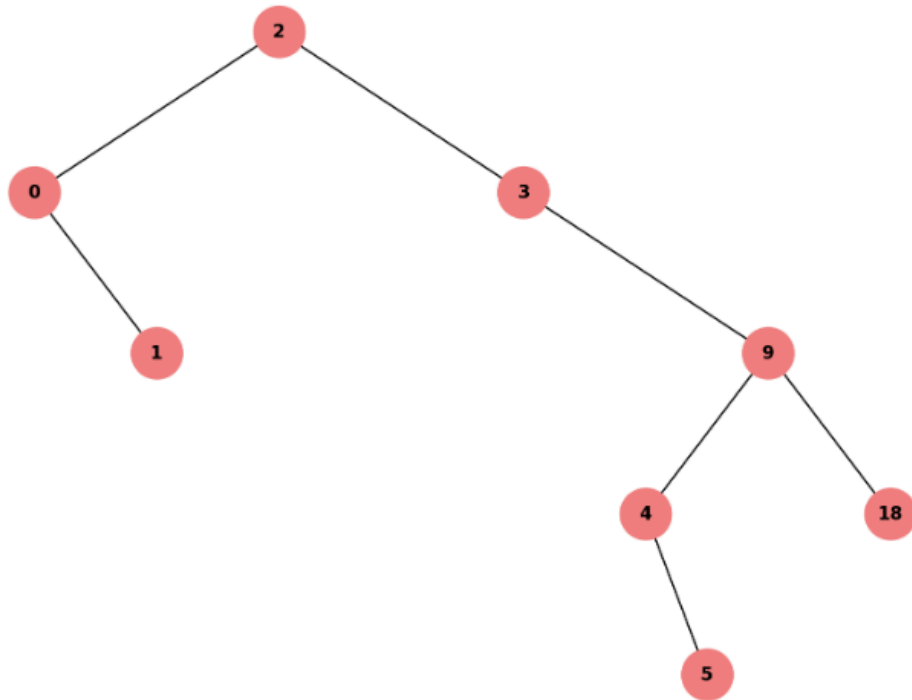
## Step 1:

```
1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     int value;
6     Node* left;
7     Node* right;
8     Node(int val) : value(val), left(nullptr), right(nullptr) {}
9 };
10
11 Node* insert(Node* node, int value) {
12     if (!node) return new Node(value);
13     if (value < node->value) node->left = insert(node->left, value);
14     else if (value > node->value) node->right = insert(node->right, value);
15     return node;
16 }
17
18 void inorder(Node* node) {
19     if (node) {
20         inorder(node->left);
21         cout << node->value << " ";
22         inorder(node->right);
23     }
24 }
25
26 int main() {
27     Node* root = nullptr;
28     int values[] = {2, 3, 9, 18, 0, 1, 4, 5};
29     for (int val : values) root = insert(root, val);
30
31     cout << "Inorder traversal: ";
32     inorder(root);
33     cout << endl;
34
35     return 0;
36 }
```

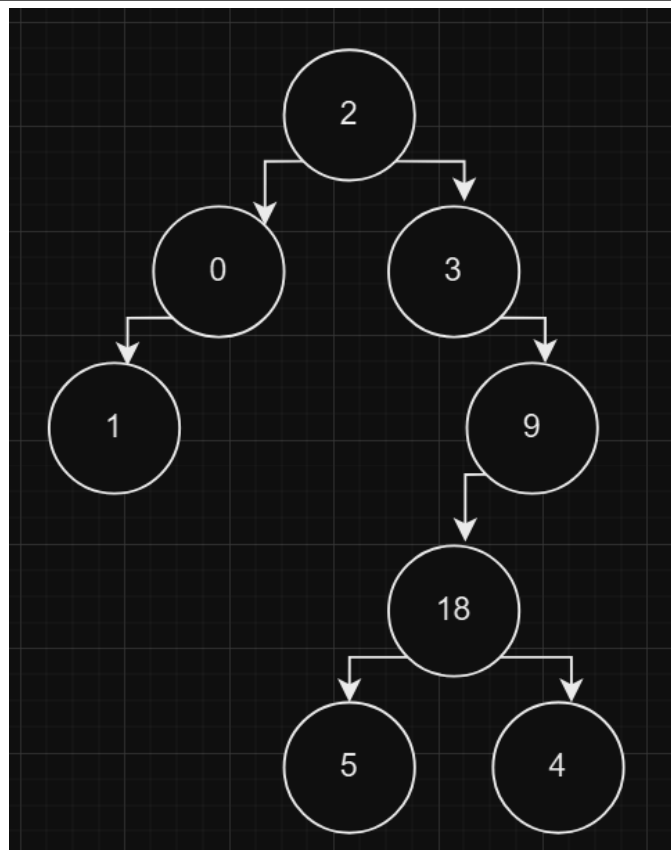
Step 2:



In-order Traversal



Pre-order Traversal



Post-order Traversal

Step 3:

#### In-order Traversal

```
void inorder(Node* node) {  
    if (node) {  
        inorder(node->left);  
        cout << node->value << " ";  
        inorder(node->right);  
    }  
}
```

```
Inorder traversal: 0 1 2 3 4 5 9 18
```

```
=== Code Execution Successful ===
```

In in-order traversal, the nodes are visited in the order of left subtree, root, and right subtree. This traversal is useful for printing nodes in sorted order in a binary search tree (BST).

### Pre-order Traversal

```
void preorder(Node* node) {  
    if (node) {  
        cout << node->value << " ";  
        preorder(node->left);  
        preorder(node->right);  
    }  
}
```

Preorder traversal: 2 0 1 3 9 4 5 18

In pre-order traversal, the root is visited first, followed by the left subtree and then the right subtree. It is often used for copying or serializing a tree structure.

### Post-order Traversal

```
void postorder(Node* node) {  
    if (node) {  
        postorder(node->left);  
        postorder(node->right);  
        cout << node->value << " ";  
    }  
}
```

Postorder traversal: 1 0 5 4 18 9 3 2

In post-order traversal, the left and right subtrees are visited before the root node. This traversal is useful for operations like tree deletion or postfix notation in expressions.

## C. Conclusion & Lessons Learned

In conclusion, this laboratory introduced us to tree data structures and their importance in programming. Trees comprise connected nodes, forming a hierarchy useful for organizing data. We learned about binary trees, where each node has up to two children, and explored different types like full, complete, and balanced binary trees.

Coding these structures and understanding tree traversals was challenging, but it helped us get a better grasp of how they work. Though some parts were tough, We see it as part of the learning process and be prepared to work with data structures in future projects.

## D. Assessment Rubric

## E. External References