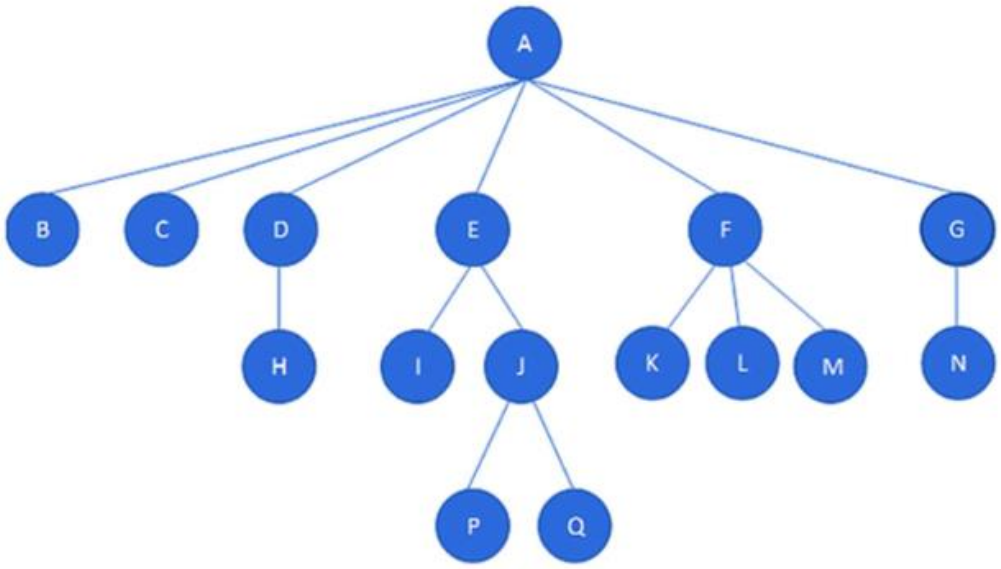
Activity No. 9.1	
Tree ADT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/4/2025**
Section: CPE21S4	**Date Submitted: 10/4/2025**
Name(s): Anastacio, Lester Arvid P.	**Instructor: Engr. Jimlord M. Quejado**

6. Output

Table 9-1
Visual Representation:



Code:

Tree.cpp

```cpp
#include <iostream>
#include <vector>

// Define the Node structure
struct Node {
    char data;
    std::vector<Node*> children;  // List of children nodes

    // Constructor to create a new node with a given value
    Node(char value) {
        data = value;
    }
};

// Function to add a child node to a parent node
void addNodeChild(Node* parent, Node* child) {
    parent->children.push_back(child);
}

// Function to print the tree for debugging
void Tree(Node* root, int level = 0) {
    if (root == nullptr)
        return;

    // Print the current node's data
    for (int i = 0; i < level; i++) {
        std::cout << "  ";  // Indentation for child nodes
    }
    std::cout << root->data << std::endl;

    // Recursively print the children nodes
    for (Node* child : root->children) {
        Tree(child, level + 1);
    }
}
```

Observation:

This is basically the core functions that will make the tree that is equivalent to the visual representation, the struct Node is where we basically list the children nodes and where we create a new node with a given value, in simpler terms this creates new nodes for nodes that will conceive child nodes, and void addNodeChild is where the new node will be added in below the parent node as you can see it pushes the child node at the back of the parent node and the void Tree function is where the Tree is basically printed to follow the expected outcome basically it prints out in a specific order which is pre-order traversal.

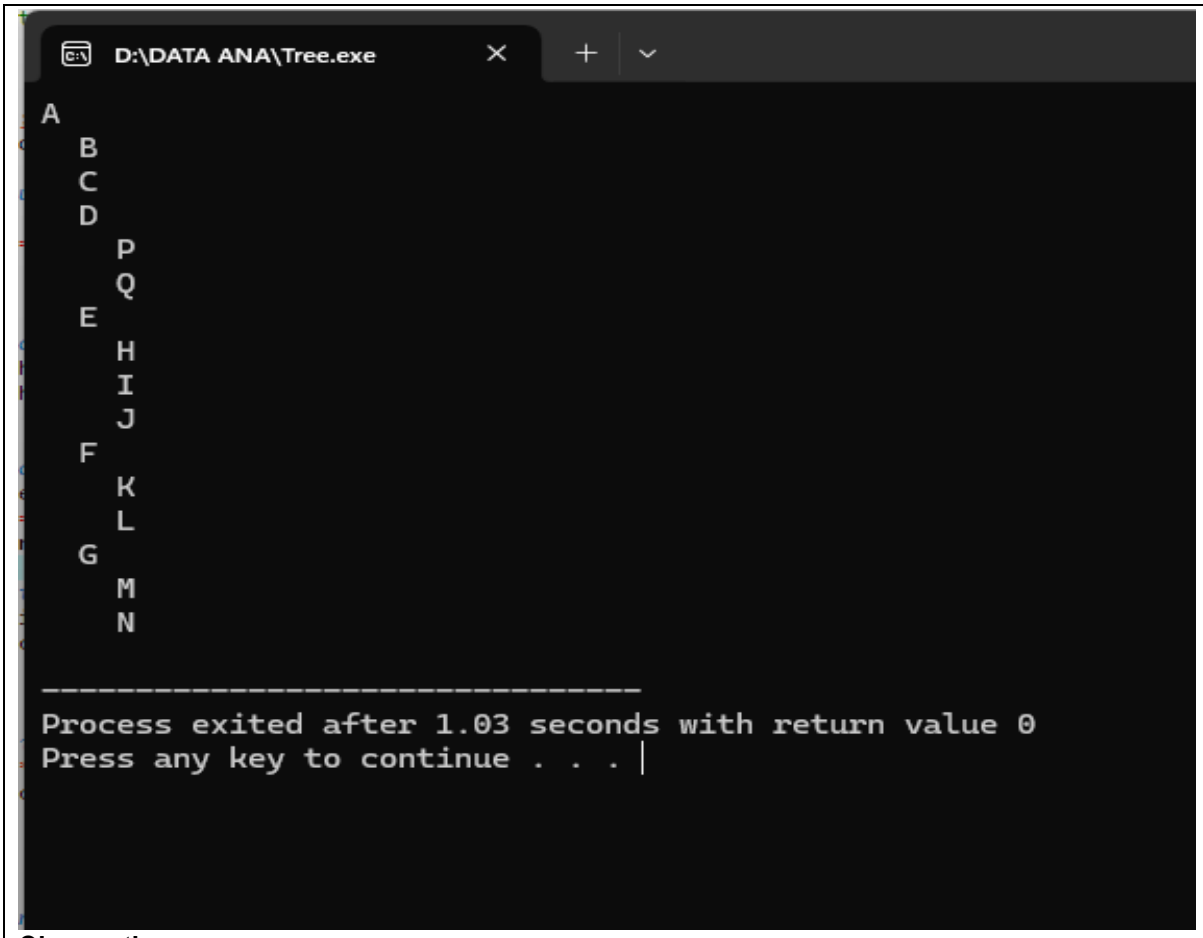
Tree.cpp

```cpp
int main() {
    // Creating nodes for the tree
    Node* A = new Node('A');
    Node* B = new Node('B');
    Node* C = new Node('C');
    Node* D = new Node('D');
    Node* E = new Node('E');
    Node* F = new Node('F');
    Node* G = new Node('G');
    Node* H = new Node('H');
    Node* I = new Node('I');
    Node* J = new Node('J');
    Node* K = new Node('K');
    Node* L = new Node('L');
    Node* M = new Node('M');
    Node* N = new Node('N');
    Node* P = new Node('P');
    Node* Q = new Node('Q');

    // Building the tree structure
    addNodeChild(A, B);
    addNodeChild(A, C);
    addNodeChild(A, D);
    addNodeChild(A, E);
    addNodeChild(A, F);
    addNodeChild(A, G);

    addNodeChild(E, H);
    addNodeChild(E, I);
    addNodeChild(E, J);

    addNodeChild(F, K);
    addNodeChild(F, L);

    addNodeChild(G, M);
    addNodeChild(G, N);

    addNodeChild(D, P);
    addNodeChild(D, Q);

    // Print the tree (for verification)
    Tree(A);

```

Observation:

This is where I now input the values of each nodes to follow in the visual representation.

Output:



Observation:

The output here shows the hierarchy of each node, which follows the visual representation of it, as you can see the A is the root of the tree where everything basically comes from it has three child nodes that are B, C, D, E, F, G and among those only D, E, F, G has a node child which makes Them a parent node while B and C are what you call a Leaf node.

Table 9-2

Node	Height	Depth
A	3	0
B	0	1
C	0	1
D	1	1
E	2	1
F	2	1
G	2	1
H	0	2
I	0	2
J	0	2
K	0	2
L	0	2
M	0	2
N	0	2
O	0	2
P	0	2
Q	0	2

Table 9-3:

Answer:

Pre-order	**A, B, C, D, H, E, I, J, P, Q, F, K, L, M, G, N**
Post-order	**B, C, H, D, I, P, Q, J, E, K, L, M, F, N, G, A**
In-order	**B, A, C, H, D, I, E, P, J, Q, K, F, L, M, N, G**

Table 9-4:

Added Functions:

```
38
39  // Pre-order Traversal: Node -> Child1 -> Child2 ...
40  void preOrderTraversal(Node* root) {
41      if (root == nullptr)
42          return;
43
44      std::cout << root->data << " "; // 1. Visit the node (Root)
45
46      // 2. Recursively visit all children from left-to-right
47      for (Node* child : root->children) {
48          preOrderTraversal(child);
49      }
50  }
51
52  // Post-order Traversal: Child1 -> Child2 ... -> Node
53  void postOrderTraversal(Node* root) {
54      if (root == nullptr)
55          return;
56
57      // 1. Recursively visit all children from left-to-right
58      for (Node* child : root->children) {
59          postOrderTraversal(child);
60      }
61
62      std::cout << root->data << " "; // 2. Visit the node (Root)
63  }
64
65  // In-order Traversal (Generalized for N-ary Tree)
66  // Child1 -> Node -> Child2 -> Child3 ...
67  void inOrderTraversal(Node* root) {
68      if (root == nullptr)
69          return;
70
71      if (!root->children.empty()) {
72          // 1. Visit the first child
73          inOrderTraversal(root->children[0]);
74      }
75
76      std::cout << root->data << " "; // 2. Visit the node (Root)
77
78      // 3. Visit the remaining children
79      for (size_t i = 1; i < root->children.size(); ++i) {
80          inOrderTraversal(root->children[i]);
81      }
82  }
83
```

Observation:

The three new added functions, preOrderTraversal, postOrderTraversal, and inOrderTraversal, are there to successfully implement the respective generic tree traversals by strategically positioning the node-printing statement relative to the recursive calls to its children.

Output:

Table 9-5:
Added Functions:

Tree.cpp

```cpp
#include <iostream>
#include <vector>
#include <string> // Required for the CHOICE parameter

// Define the Node structure (omitted for brevity, assume it's the same)
struct Node {
    char data;
    std::vector<Node*> children;
    Node(char value) { data = value; }
};

// Function to add a child node (omitted for brevity)
void addNodeChild(Node* parent, Node* child) {
    parent->children.push_back(child);
}

// Traversal Functions (omitted for brevity, assume they are the same)
void preOrderTraversal(Node* root) { /* ... */ }
void postOrderTraversal(Node* root) { /* ... */ }
void inOrderTraversal(Node* root) { /* ... */ }
void Tree(Node* root, int level = 0) { /* ... */ }


// Helper function that performs the recursive search.
// It returns true if the key is found, false otherwise.
bool searchTree(Node* root, char key) {
    if (root == nullptr) {
        return false;
    }

    // 1. Check the current node
    if (root->data == key) {
        return true;
    }

    // 2. Recursively check all children
    for (Node* child : root->children) {
        if (searchTree(child, key)) {
            return true; // Key found in a child branch
        }
    }
```

Tree.cpp

```cpp
    return false; // Key not found in this branch
}

// The required public function signature using the CHOICE parameter
void findData(std::string choice, char key, Node* root) {
    // The CHOICE parameter is acknowledged but the search logic remains the same
    // as the goal is simply to find the key efficiently, regardless of the specific traversal order.

    std::cout << "Searching for '" << key << "' using " << choice << ": ";

    if (searchTree(root, key)) {
        std::cout << "{" << key << "} was found!" << std::endl;
    } else {
        std::cout << "Key not found." << std::endl;
    }
}

int main() {
    // ... Node initialization (A-Q) ...
    Node* A = new Node('A');
    Node* B = new Node('B');
    // ... (rest of nodes H-Q) ...
    Node* C = new Node('C');
    Node* D = new Node('D');
    Node* E = new Node('E');
    Node* F = new Node('F');
    Node* G = new Node('G');
    Node* H = new Node('H');
    Node* I = new Node('I');
    Node* J = new Node('J');
    Node* K = new Node('K');
    Node* L = new Node('L');
    Node* M = new Node('M');
    Node* N = new Node('N');
    Node* P = new Node('P');
    Node* Q = new Node('Q');

    // ... Building the tree structure (A to G, D->H, E->I/J, J->P/Q, F->K/L/M, G->N) ...
    addNodeChild(A, B); addNodeChild(A, C); addNodeChild(A, D); addNodeChild(A, E); addNodeChild(A, F); addNodeChild(A, G);
    addNodeChild(D, H);
    addNodeChild(E, I); addNodeChild(E, J);
    addNodeChild(J, P); addNodeChild(J, Q);
    addNodeChild(F, K); addNodeChild(F, L); addNodeChild(F, M);

    // ... Traversal testing calls (omitted) ...

    std::cout << "\n========================================" << std::endl;
    std::cout << "      findData Function Calls       " << std::endl;
    std::cout << "========================================" << std::endl;

    // Test Case 1: Find a node that exists ('J')
    findData("Pre-order", 'J', A);

    // Test Case 2: Find a node that exists ('M')
    findData("Post-order", 'M', A);

    // Test Case 3: Find a node that does not exist ('Z')
    findData("In-order", 'Z', A);

    return 0;
}
```

Output:

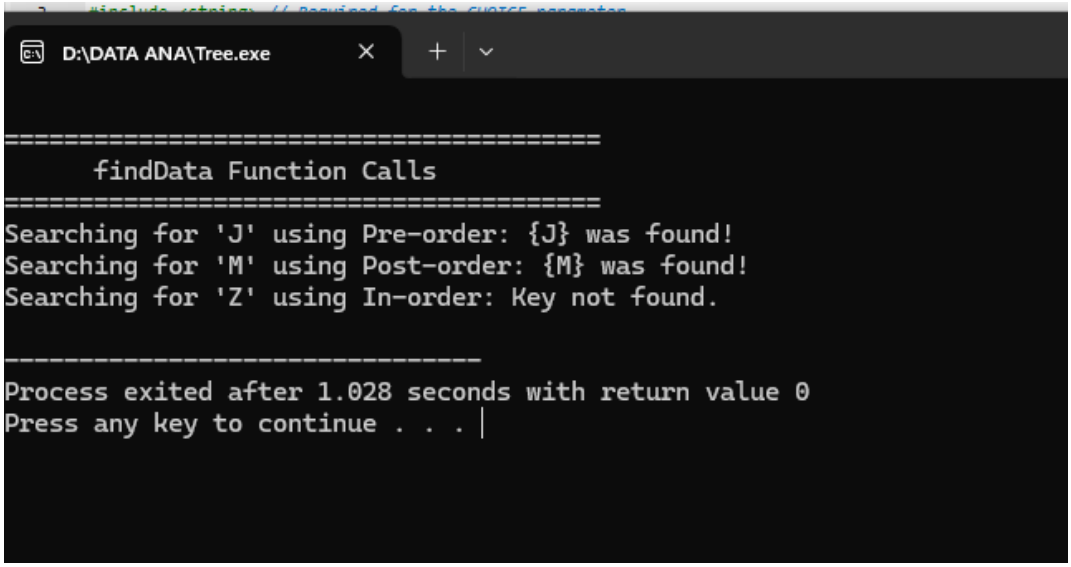


Table 9-6:

Added Functions:

Tree.cpp

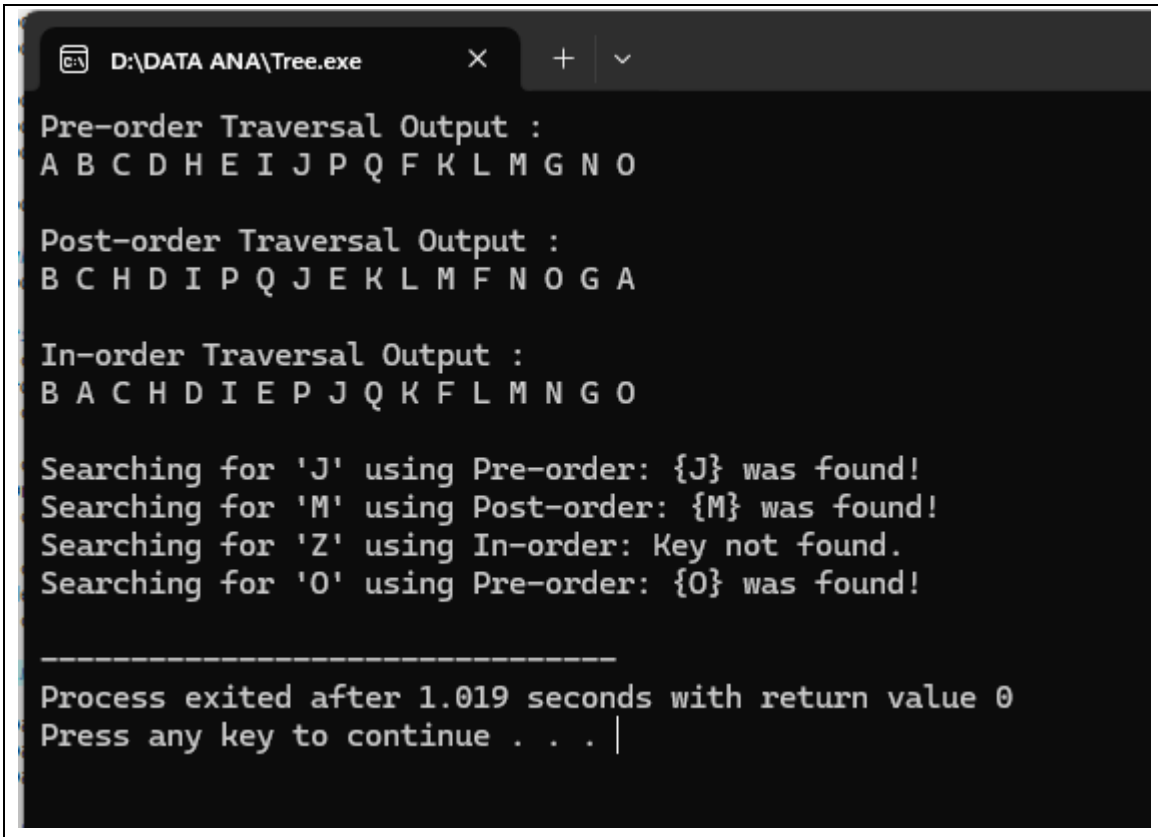
```cpp
#include <iostream>
#include <vector>
#include <string>

// Define the Node structure
struct Node {
    char data;
    std::vector<Node*> children;  // List of children nodes

    // Constructor to create a new node with a given value
    Node(char value) {
        data = value;
    }

    // Simple destructor (for cleanup, though not strictly required by the prompt)
    ~Node() {
        for (Node* child : children) {
            delete child;
        }
    }
};

// Function to add a child node to a parent node
void addNodeChild(Node* parent, Node* child) {
    parent->children.push_back(child);
}

// Function to print the tree for debugging
void Tree(Node* root, int level = 0) {
    if (root == nullptr)
        return;

    // Print the current node's data
    for (int i = 0; i < level; i++) {
        std::cout << "  ";  // Indentation for child nodes
    }
    std::cout << root->data << std::endl;

    // Recursively print the children nodes
    for (Node* child : root->children) {
        Tree(child, level + 1);
    }
}
```

```cpp

// Pre-order Traversal: Node -> Child1 -> Child2 ...
void preOrderTraversal(Node* root) {
    if (root == nullptr)
        return;

    std::cout << root->data << " "; // 1. Visit the node (Root)

    // 2. Recursively visit all children from left-to-right
    for (Node* child : root->children) {
        preOrderTraversal(child);
    }
}

// Post-order Traversal: Child1 -> Child2 ... -> Node
void postOrderTraversal(Node* root) {
    if (root == nullptr)
        return;

    // 1. Recursively visit all children from left-to-right
    for (Node* child : root->children) {
        postOrderTraversal(child);
    }

    std::cout << root->data << " "; // 2. Visit the node (Root)
}

// In-order Traversal (Generalized for N-ary Tree): Child1 -> Node -> Child2 ...
void inOrderTraversal(Node* root) {
    if (root == nullptr)
        return;

    if (!root->children.empty()) {
        // 1. Visit the first child
        inOrderTraversal(root->children[0]);
    }

    std::cout << root->data << " "; // 2. Visit the node (Root)

    // 3. Visit the remaining children
    for (size_t i = 1; i < root->children.size(); ++i) {
        inOrderTraversal(root->children[i]);
```

```
    }
}

// Helper function that performs the recursive search.
bool searchTree(Node* root, char key) {
    if (root == nullptr) {
        return false;
    }

    if (root->data == key) {
        return true;
    }

    // Recursively check all children
    for (Node* child : root->children) {
        if (searchTree(child, key)) {
            return true;
        }
    }

    return false;
}

// The required findData function
void findData(std::string choice, char key, Node* root) {
    std::cout << "Searching for '" << key << "' using " << choice << ": ";

    if (searchTree(root, key)) {
        std::cout << "{" << key << "} was found!" << std::endl;
    } else {
        std::cout << "Key not found." << std::endl;
    }
}

int main() {
    // Creating nodes for the tree
    Node* A = new Node('A');
    Node* B = new Node('B');
    Node* C = new Node('C');
    Node* D = new Node('D');
    Node* E = new Node('E');
    Node* F = new Node('F');
```

```
    Node* G = new Node('G');
    Node* H = new Node('H');
    Node* I = new Node('I');
    Node* J = new Node('J');
    Node* K = new Node('K');
    Node* L = new Node('L');
    Node* M = new Node('M');
    Node* N = new Node('N');
    Node* P = new Node('P');
    Node* Q = new Node('Q');
    Node* O = new Node('O');

    // Building the tree structure (based on the diagram)
    addNodeChild(A, B);
    addNodeChild(A, C);
    addNodeChild(A, D);
    addNodeChild(A, E);
    addNodeChild(A, F);
    addNodeChild(A, G);

    addNodeChild(D, H);

    addNodeChild(E, I);
    addNodeChild(E, J);

    addNodeChild(J, P);
    addNodeChild(J, Q);

    addNodeChild(F, K);
    addNodeChild(F, L);
    addNodeChild(F, M);

    addNodeChild(G, N);

    // Make 'O' the child of 'G'
    addNodeChild(G, C);

    // Testing the new traversal functions
    std::cout << "Pre-order Traversal Output :\n";
    preOrderTraversal(A);
    std::cout << "\n\n";

    std::cout << "Post-order Traversal Output :\n";
    postOrderTraversal(A);
    std::cout << "\n\n";

    std::cout << "In-order Traversal Output :\n";
    inOrderTraversal(A);
    std::cout << "\n\n";

    // findData Function Calls

    findData("Pre-order", 'J', A);
    findData("Post-order", 'M', A);
    findData("In-order", 'Z', A);

    // findData Function Call with new node 'O'
    findData("Pre-order", 'O', A);

    // Clean up memory
    delete A;

    return 0;
}
```

Output:



7. Supplementary Activity

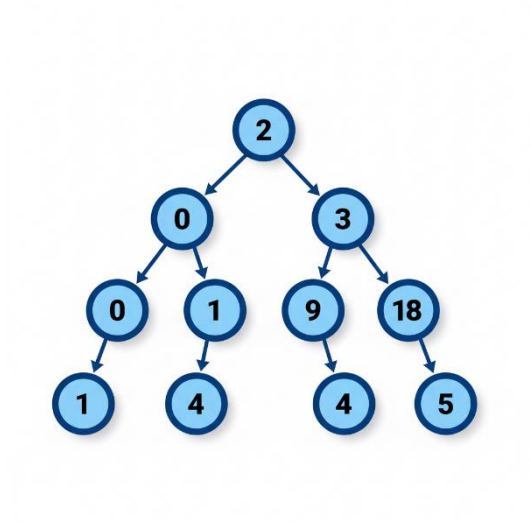
Code:

Tree.cpp

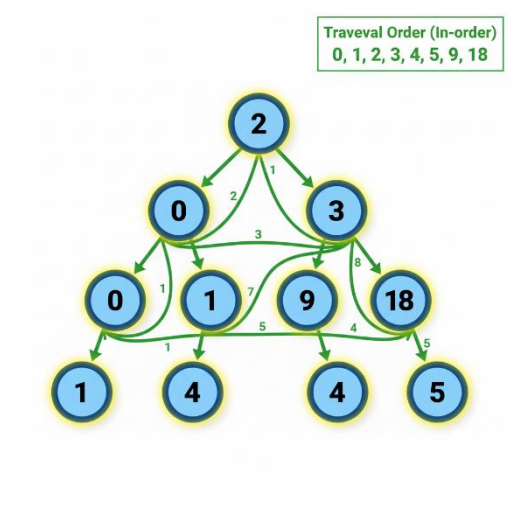
```cpp
#include <iostream>
#include <vector>
#include <algorithm> // For std::find, though not strictly needed here

// New Node structure for a Binary Search Tree
struct BSTNode {
    int data;
    BSTNode* left;
    BSTNode* right;

    // Constructor
    BSTNode(int value) : data(value), left(nullptr), right(nullptr) {}

    // Destructor for proper memory cleanup
    ~BSTNode() {
        // Recursive deletion for the tree
        delete left;
        delete right;
    }
};

// Function to insert a value into the BST
BSTNode* insert(BSTNode* root, int value) {
    // If the tree is empty, create a new node
    if (root == nullptr) {
        return new BSTNode(value);
    }

    // Traverse to the correct position based on BST rule
    if (value < root->data) {
        root->left = insert(root->left, value); // Insert left
    } else if (value > root->data) {
        root->right = insert(root->right, value); // Insert right
    }
    // If value equals root->data, do nothing (no duplicates)

    return root;
}
```

```cpp

// In-order Traversal: Left -> Node -> Right (Prints sorted order)
void inOrderTraversal(BSTNode* root) {
    if (root == nullptr) return;
    inOrderTraversal(root->left);
    std::cout << root->data << " ";
    inOrderTraversal(root->right);
}

// Pre-order Traversal: Node -> Left -> Right
void preOrderTraversal(BSTNode* root) {
    if (root == nullptr) return;
    std::cout << root->data << " ";
    preOrderTraversal(root->left);
    preOrderTraversal(root->right);
}

// Post-order Traversal: Left -> Right -> Node
void postOrderTraversal(BSTNode* root) {
    if (root == nullptr) return;
    postOrderTraversal(root->left);
    postOrderTraversal(root->right);
    std::cout << root->data << " ";
}


int main() {
    BSTNode* root = nullptr;
    std::vector<int> values = {2, 3, 9, 18, 0, 1, 4, 5};

    // Insert all values into the BST
    for (int val : values) {
        root = insert(root, val);
    }

    // (Comparison)
    std::cout << "In-order Traversal:  ";
    inOrderTraversal(root);
    std::cout << "\n";

    // (Pre-order)
    std::cout << "Pre-order Traversal: ";
    preOrderTraversal(root);
    std::cout << "\n";

    // (Post-order)
    std::cout << "Post-order Traversal: ";
    postOrderTraversal(root);
    std::cout << "\n";

    // Clean up memory
    delete root;

    return 0;
}
```

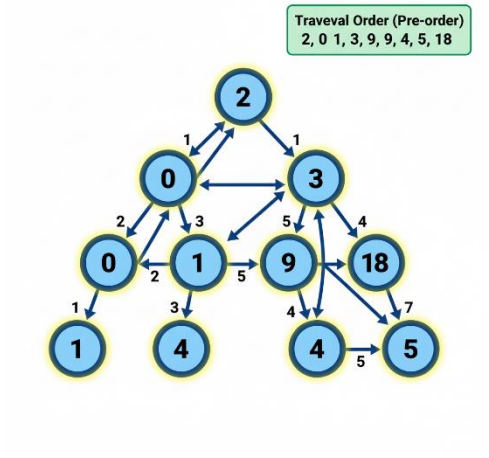
Tree Diagram:
Original:



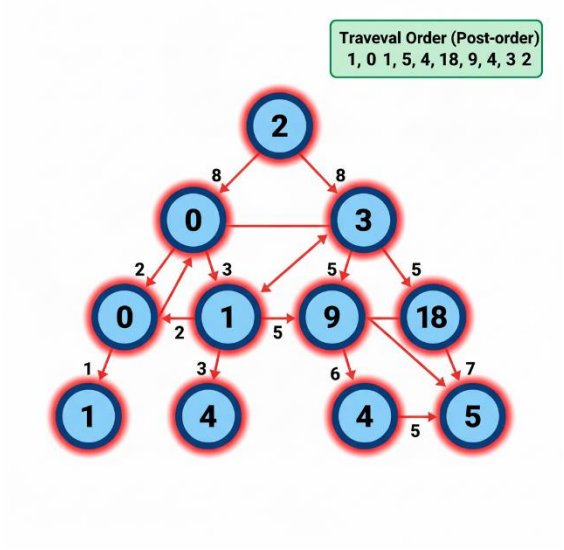
In-Order Traversal



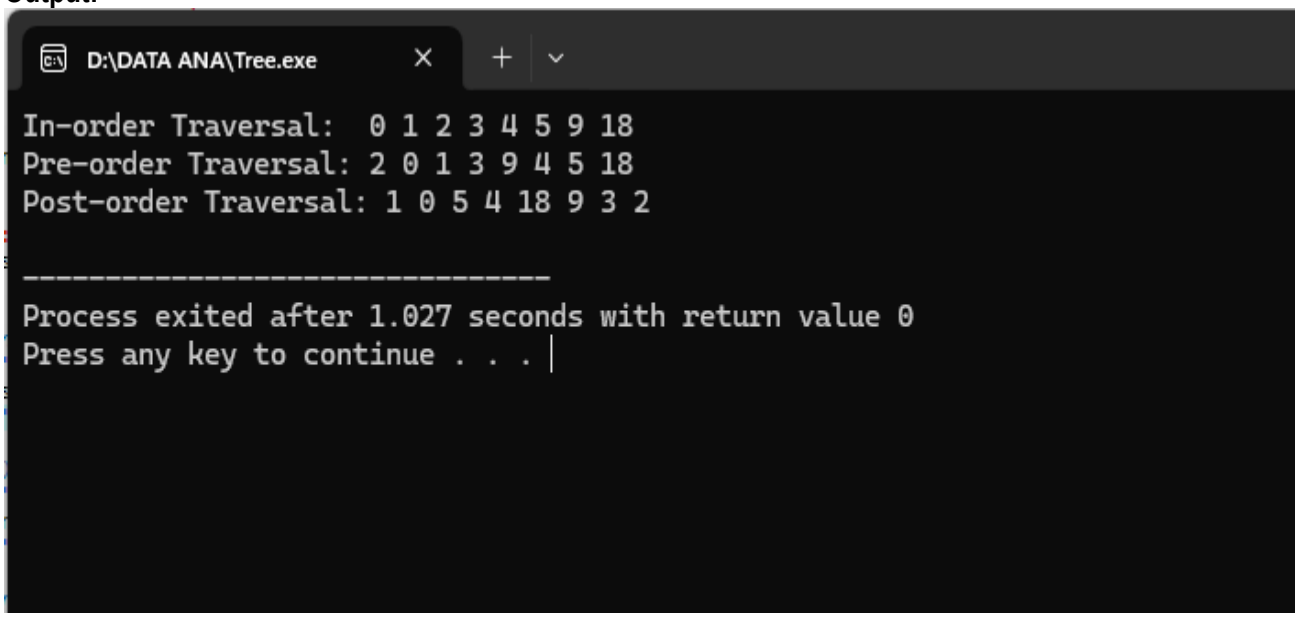
Pre-order Traversal



Post-order Traversal



Output:



Observations:

I've noticed that in in-order traversal the output is quite identical to the manual result indicated in the diagram, the function correctly visits the nodes in the order of Left subtree first next it moves to the nodes and then moves to the Right Subtree while in Pre-order Traversal visits the nodes in a over where it first goes to the Node and them moves to the left subtree and then lastly to the right subtree and in Post-order Traversal it traverse in a order in which it starts with the left subtree and then right subtree and then the node is the last, and also the node here is basically the parent node.

8. Conclusion

- In this activity, I have successfully gone through the differences between the Generic (N-ary) Tree and the Binary Search Tree (BST), starting with the correct implementation and functional testing of Pre-order, Post-order, and generalized In-order traversals for the Generic Tree . I verified that the function outputs matched the theoretical manual calculations, and I also integrated a simple, yet powerful depth-first search (DFS) function (findData) to efficiently locate data and confirm that the search can operate independently of the traversal order, i.e., the performance of the search was tested after the change of the generic tree structure. Next, I implemented a separate Binary Search Tree (BST) with the insertion logic for the sequence {2,3,9,18,0,1,4,5} allowing which visuals and functionalities have successfully proven the fundamental concept, i.e., the In-order traversal of a BST uniquely yields a perfectly sorted output, thus the critical lesson that structure dictates specialized function was retained. The whole process and from theoretical derivation

to it's execution was very successful, with future improvements mainly targeted towards complex operations like BST deletion and balancing.
9. Assessment Rubric