Activity No. 9.1	
Trees	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: Nov 13, 2024**
Section: CPE21S4	**Date Submitted: Nov 13, 2924**
Name(s): Esteban, Fernandez, Sanchez, Tandayu, Valleser	**Instructor:** Engr. Ma. Rizette Sayo

A. Output(s) and Observation(s)

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

// TreeNode represents each node in the general tree.
struct TreeNode {
    char data; // Character data for each node (e.g., 'A', 'B', etc.)
    vector<TreeNode*> children; // List of child nodes

    // Constructor to initialize a node with a given character.
    TreeNode(char val) : data(val) {}
};

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

// Recursive function to print the tree in a structured way.
void printTree(TreeNode* root, int depth = 0) {
    if (!root) return;
    // Print indentation for each level
    for (int i = 0; i < depth; ++i) cout << "  ";
    cout << root->data << endl;
    for (TreeNode* child : root->children) {
        printTree(child, depth + 1);
    }
}

int main() {
    // Creating nodes for each character in the tree
    TreeNode* A = new TreeNode('A');
    TreeNode* B = new TreeNode('B');
    TreeNode* C = new TreeNode('C');
    TreeNode* D = new TreeNode('D');
    TreeNode* E = new TreeNode('E');
    TreeNode* F = new TreeNode('F');
    TreeNode* G = new TreeNode('G');
    TreeNode* H = new TreeNode('H');
```

```
    TreeNode* I = new TreeNode('I');
    TreeNode* J = new TreeNode('J');
    TreeNode* K = new TreeNode('K');
    TreeNode* L = new TreeNode('L');
    TreeNode* M = new TreeNode('M');
    TreeNode* N = new TreeNode('N');
    TreeNode* P = new TreeNode('P');
    TreeNode* Q = new TreeNode('Q');

    // Constructing the tree structure
    addChild(A, B);
    addChild(A, C);
    addChild(A, D);
    addChild(A, E);
    addChild(A, F);
    addChild(A, G);

    addChild(D, H);

    addChild(E, I);
    addChild(E, J);
    addChild(E, P);
    addChild(E, Q);

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

    addChild(G, N);

    // Print the tree starting from the root node
    printTree(A);

        return 0;
}
```

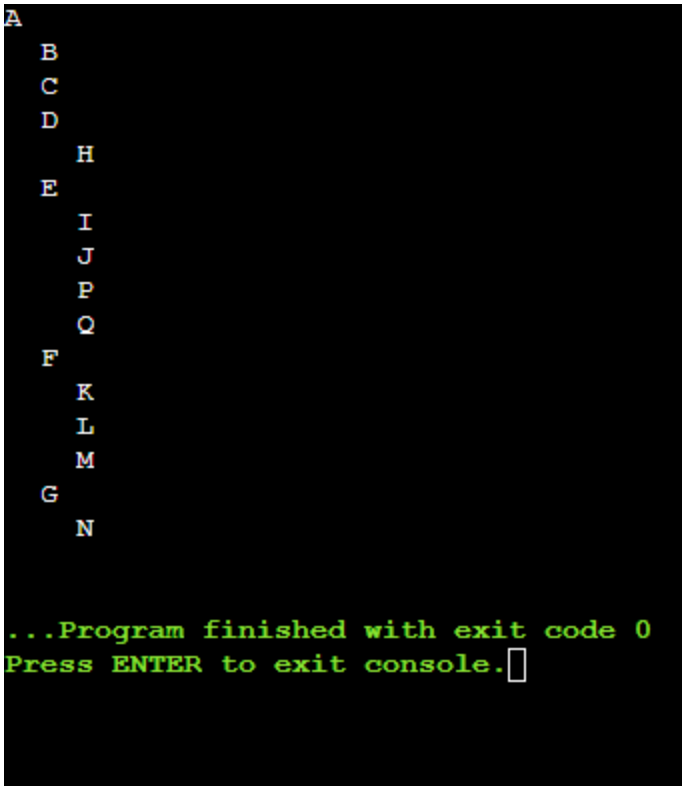


Table 9-1

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

P	0	3
Q	0	3

Table 9-2

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

Table 9-3

Code

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

class TreeNode {
public:
    char data;
    vector<TreeNode*> children;
    TreeNode(char val) : data(val) {}
};

class GeneralTree {
public:
    TreeNode* root;

    GeneralTree(char rootData) {
        root = new TreeNode(rootData);
    }

    TreeNode* addChild(TreeNode* parent, char childData) {
        TreeNode* child = new TreeNode(childData);
        parent->children.push_back(child);
        return child;
    }

    void preOrder(TreeNode* node) {
        if (!node) return;
        cout << node->data << " ";
        for (TreeNode* child : node->children) {
            preOrder(child);
        }
    }

    void postOrder(TreeNode* node) {
        if (!node) return;
        for (TreeNode* child : node->children) {
```

```
            postOrder(child);
        }
        cout << node->data << " ";
    }

    void inOrder(TreeNode* node) {
        if (!node) return;
        if (!node->children.empty()) {
            inOrder(node->children[0]);  // Leftmost child
        }
        cout << node->data << " ";
        for (size_t i = 1; i < node->children.size(); i++) {
            inOrder(node->children[i]);
        }
    }
};

int main() {
    GeneralTree tree('A');
    TreeNode* B = tree.addChild(tree.root, 'B');
    TreeNode* C = tree.addChild(tree.root, 'C');
    TreeNode* D = tree.addChild(tree.root, 'D');
    TreeNode* E = tree.addChild(tree.root, 'E');
    TreeNode* F = tree.addChild(tree.root, 'F');
    TreeNode* G = tree.addChild(tree.root, 'G');

    tree.addChild(D, 'H');

    tree.addChild(E, 'I');
    tree.addChild(E, 'J');
    TreeNode* P = tree.addChild(E, 'P');
    TreeNode* Q = tree.addChild(E, 'Q');

    tree.addChild(F, 'K');
    tree.addChild(F, 'L');
    tree.addChild(F, 'M');

    tree.addChild(G, 'N');

    cout << "Pre-order traversal: ";
    tree.preOrder(tree.root);
    cout << endl;

    cout << "Post-order traversal: ";
    tree.postOrder(tree.root);
    cout << endl;

    cout << "In-order traversal: ";
    tree.inOrder(tree.root);
    cout << endl;
```

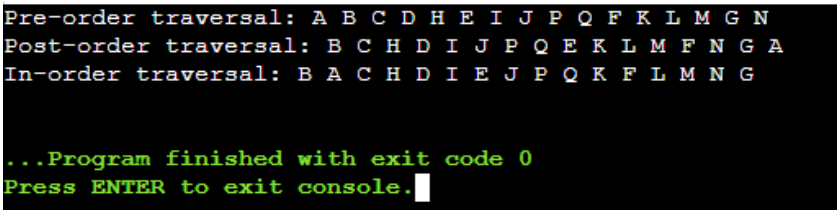
	return 0; }
Output	
Observation	The difference between our manual results in Task 3.1 and the code output in Task 3.2 is probably because in-order traversal doesn't have a standard way to handle trees with more than two children, so we might have interpreted it differently by hand. The code visits children in a specific order, which may not match the order we used in my manual calculations. Also, any small differences in how we ordered nodes in each parent could cause differences in the traversal results.

Table 9-4

Code	#include <iostream>
	#include <vector>
	using namespace std;
	
	class TreeNode {
	public:
	char data;
	vector<TreeNode*> children;
	TreeNode(char val) : data(val) {}
	};
	
	class GeneralTree {
	public:
	TreeNode* root;
	
	GeneralTree(char rootData) {
	root = new TreeNode(rootData);
	}
	
	TreeNode* addChild(TreeNode* parent, char childData) {
	TreeNode* child = new TreeNode(childData);
	parent->children.push_back(child);
	return child;
	}

```
    void preOrder(TreeNode* node) {
        if (!node) return;
        cout << node->data << " ";
        for (TreeNode* child : node->children) {
            preOrder(child);
        }
    }

    void postOrder(TreeNode* node) {
        if (!node) return;
        for (TreeNode* child : node->children) {
            postOrder(child);
        }
        cout << node->data << " ";
    }

    void inOrder(TreeNode* node) {
        if (!node) return;
        if (!node->children.empty()) {
            inOrder(node->children[0]);  // Leftmost child
        }
        cout << node->data << " ";
        for (size_t i = 1; i < node->children.size(); i++) {
            inOrder(node->children[i]);
        }
    }
};

void findData(TreeNode* node, const string& choice,
char key) {
    if (choice == "PRE") {
        if (node->data == key) {
            cout << key << " was found!" << endl;
            return;
        }
        for (TreeNode* child : node->children) {
            findData(child, choice, key);
        }
    } else if (choice == "POST") {
        for (TreeNode* child : node->children) {
            findData(child, choice, key);
        }
        if (node->data == key) {
            cout << key << " was found!" << endl;
            return;
        }
    } else if (choice == "IN") {
        if (!node->children.empty()) {
            findData(node->children[0], choice, key);
        }
        if (node->data == key) {
```

```
            cout << key << " was found!" << endl;
            return;
        }
        for (size_t i = 1; i < node->children.size(); i++) {
            findData(node->children[i], choice, key);
        }
    }
};

int main() {
    GeneralTree tree('A');
    TreeNode* B = tree.addChild(tree.root, 'B');
    TreeNode* C = tree.addChild(tree.root, 'C');
    TreeNode* D = tree.addChild(tree.root, 'D');
    TreeNode* E = tree.addChild(tree.root, 'E');
    TreeNode* F = tree.addChild(tree.root, 'F');
    TreeNode* G = tree.addChild(tree.root, 'G');

    tree.addChild(D, 'H');

    tree.addChild(E, 'I');
    tree.addChild(E, 'J');
    TreeNode* P = tree.addChild(E, 'P');
    TreeNode* Q = tree.addChild(E, 'Q');

    tree.addChild(F, 'K');
    tree.addChild(F, 'L');
    tree.addChild(F, 'M');

    tree.addChild(G, 'N');

        cout << "Finding 'M' with pre-order traversal:" <<
endl;
    findData(tree.root, "PRE", 'M');
    cout << "Finding 'L' with post-order traversal:" <<
endl;
    findData(tree.root, "POST", 'L');
    cout << "Finding 'K' with in-order traversal:" <<
endl;
    findData(tree.root, "IN", 'K');
    return 0;
}
```

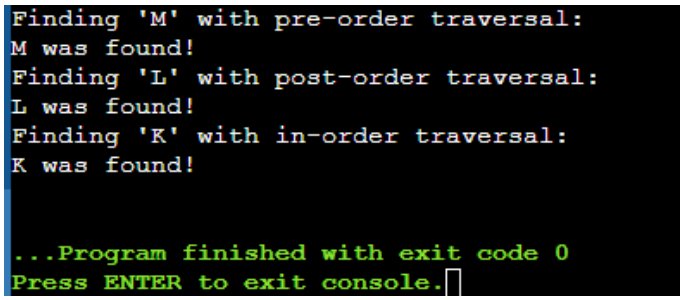
Output	

Table 9-5

Code	#include <iostream>
	#include <vector>
	using namespace std;
	
	class TreeNode {
	public:
	  char data;
	  vector<TreeNode*> children;
	  TreeNode(char val) : data(val) {}
	};
	
	class GeneralTree {
	public:
	  TreeNode* root;
	
	  GeneralTree(char rootData) {
	  root = new TreeNode(rootData);
	  }
	
	  TreeNode* addChild(TreeNode* parent, char childData) {
	  TreeNode* child = new TreeNode(childData);
	  parent->children.push_back(child);
	  return child;
	  }
	
	  void preOrder(TreeNode* node) {
	  if (!node) return;
	  cout << node->data << " ";
	  for (TreeNode* child : node->children) {
	  preOrder(child);
	  }
	  }
	
	  void postOrder(TreeNode* node) {
	  if (!node) return;
	  for (TreeNode* child : node->children) {
	  postOrder(child);
	  }
	  cout << node->data << " ";

```
    }

    void inOrder(TreeNode* node) {
        if (!node) return;
        if (!node->children.empty()) {
            inOrder(node->children[0]);  // Leftmost child
        }
        cout << node->data << " ";
        for (size_t i = 1; i < node->children.size(); i++) {
            inOrder(node->children[i]);
        }
    }
};

void findData(TreeNode* node, const string& choice,
char key) {
    if (choice == "PRE") {
        if (node->data == key) {
            cout << key << " was found!" << endl;
            return;
        }
        for (TreeNode* child : node->children) {
            findData(child, choice, key);
        }
    } else if (choice == "POST") {
        for (TreeNode* child : node->children) {
            findData(child, choice, key);
        }
        if (node->data == key) {
            cout << key << " was found!" << endl;
            return;
        }
    } else if (choice == "IN") {
        if (!node->children.empty()) {
            findData(node->children[0], choice, key);
        }
        if (node->data == key) {
            cout << key << " was found!" << endl;
            return;
        }
        for (size_t i = 1; i < node->children.size(); i++) {
            findData(node->children[i], choice, key);
        }
    }
};

int main() {
    GeneralTree tree('A');
    TreeNode* B = tree.addChild(tree.root, 'B');
    TreeNode* C = tree.addChild(tree.root, 'C');
    TreeNode* D = tree.addChild(tree.root, 'D');
```

	TreeNode* E = tree.addChild(tree.root, 'E');
	TreeNode* F = tree.addChild(tree.root, 'F');
	TreeNode* G = tree.addChild(tree.root, 'G');
	
	tree.addChild(D, 'H');
	
	tree.addChild(E, 'I');
	tree.addChild(E, 'J');
	TreeNode* P = tree.addChild(E, 'P');
	TreeNode* Q = tree.addChild(E, 'Q');
	TreeNode* O = tree.addChild(G, 'O');
	
	tree.addChild(F, 'K');
	tree.addChild(F, 'L');
	tree.addChild(F, 'M');
	
	tree.addChild(G, 'N');
	
	cout << "Finding 'O' with pre-order traversal:" << endl;
	findData(tree.root, "PRE", 'O');
	cout << "Finding 'O' with post-order traversal:" << endl;
	findData(tree.root, "POST", 'O');
	cout << "Finding 'O' with in-order traversal:" << endl;
	findData(tree.root, "IN", 'O');
	return 0;
	
	return 0;
	}

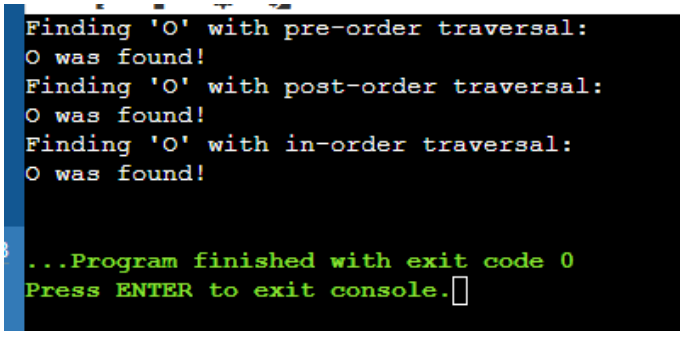
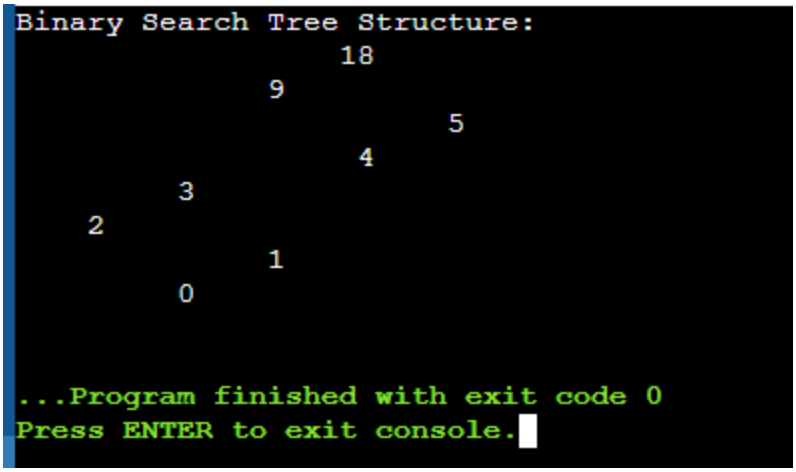
Output	
Answer	Once we add node "O," the findData function should be able to find it no matter which traversal method we choose, as long as we set the CHOICE parameter to a valid traversal option.

Table 9-6

B. Answers to Supplementary Activity

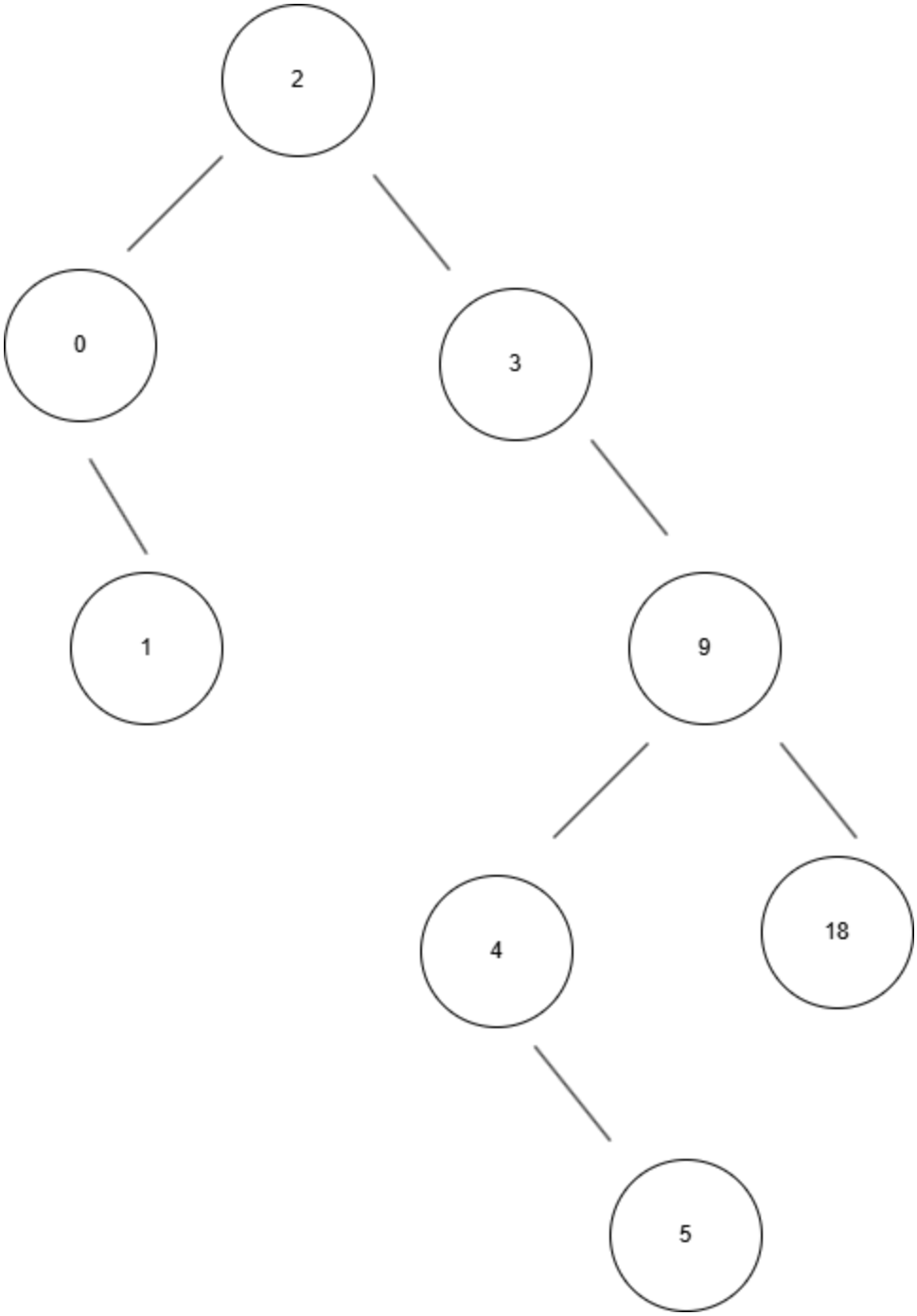
```
#include <iostream>
#include <iomanip>
using namespace std;

class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
};

class BinarySearchTree {
public:
    TreeNode* root;

    BinarySearchTree() : root(nullptr) {}

    // Insert function
    TreeNode* insert(TreeNode* node, int val) {
        if (node == nullptr) {
            return new TreeNode(val);
        }
        if (val < node->data) {
            node->left = insert(node->left, val);
        } else if (val > node->data) {
            node->right = insert(node->right, val);
        }
        return node;
    }

    void insert(int val) {
        root = insert(root, val);
    }

    // Display function (in-order visual representation)
    void display(TreeNode* node, int space = 0, int indent = 5) {
        if (node == nullptr) return;
        space += indent;
        display(node->right, space);
        cout << setw(space) << node->data << endl;
        display(node->left, space);
    }

    // Traversal functions
    void inOrder(TreeNode* node) {
        if (node != nullptr) {
```

```
            inOrder(node->left);
            cout << node->data << " ";
            inOrder(node->right);
        }
    }

    void preOrder(TreeNode* node) {
        if (node != nullptr) {
            cout << node->data << " ";
            preOrder(node->left);
            preOrder(node->right);
        }
    }

    void postOrder(TreeNode* node) {
        if (node != nullptr) {
            postOrder(node->left);
            postOrder(node->right);
            cout << node->data << " ";
        }
    }
};

int main() {
    BinarySearchTree bst;
    int values[] = {2, 3, 9, 18, 0, 1, 4, 5};

    for (int val : values) {
        bst.insert(val);
    }

    cout << "Binary Search Tree Structure:\n";
    bst.display(bst.root);

    cout << "\nIn-order Traversal: ";
    bst.inOrder(bst.root);
    cout << endl;

    cout << "Pre-order Traversal: ";
    bst.preOrder(bst.root);
    cout << endl;

    cout << "Post-order Traversal: ";
    bst.postOrder(bst.root);
    cout << endl;

    return 0;
}
```

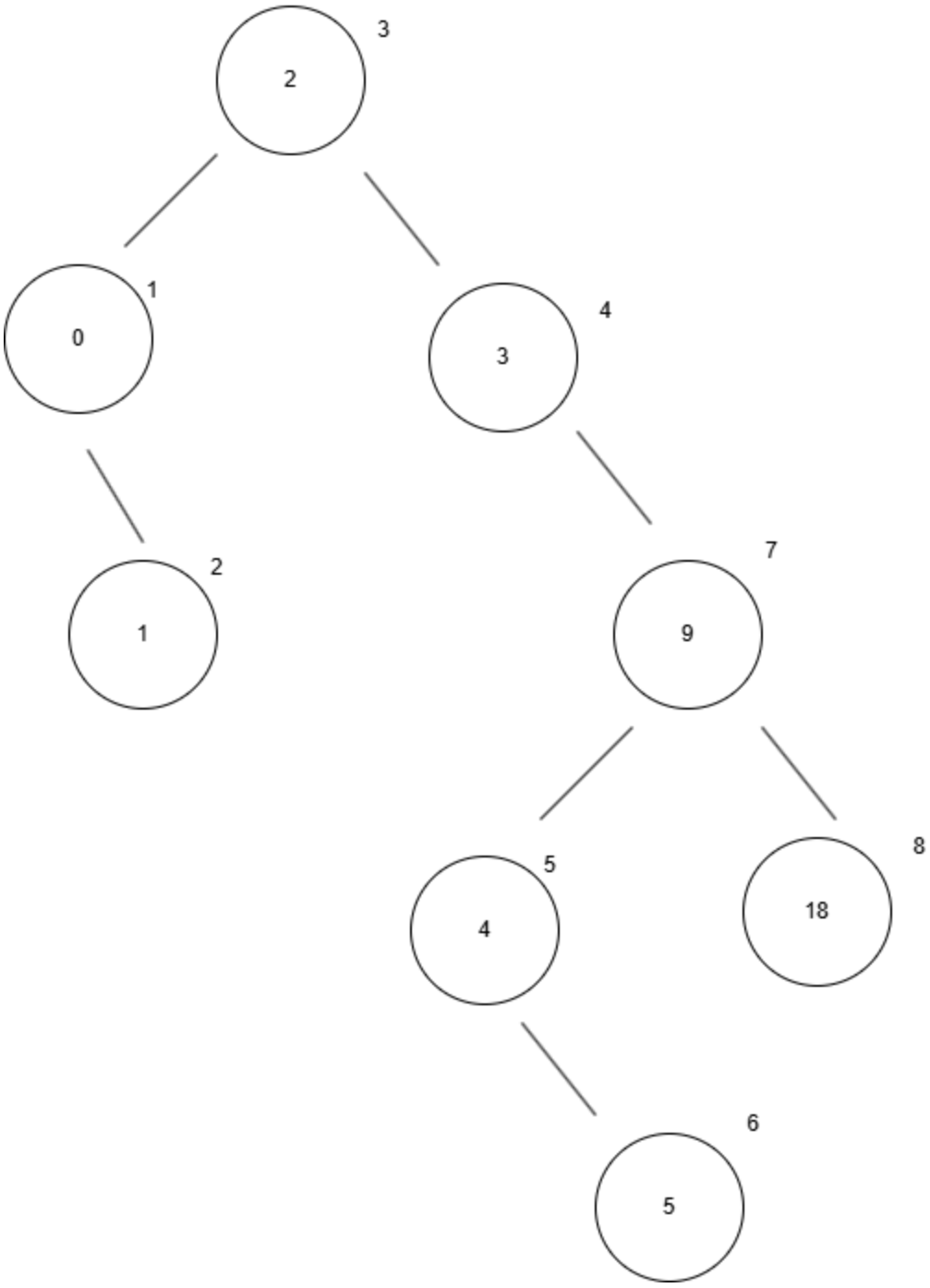


(Screenshot of code)

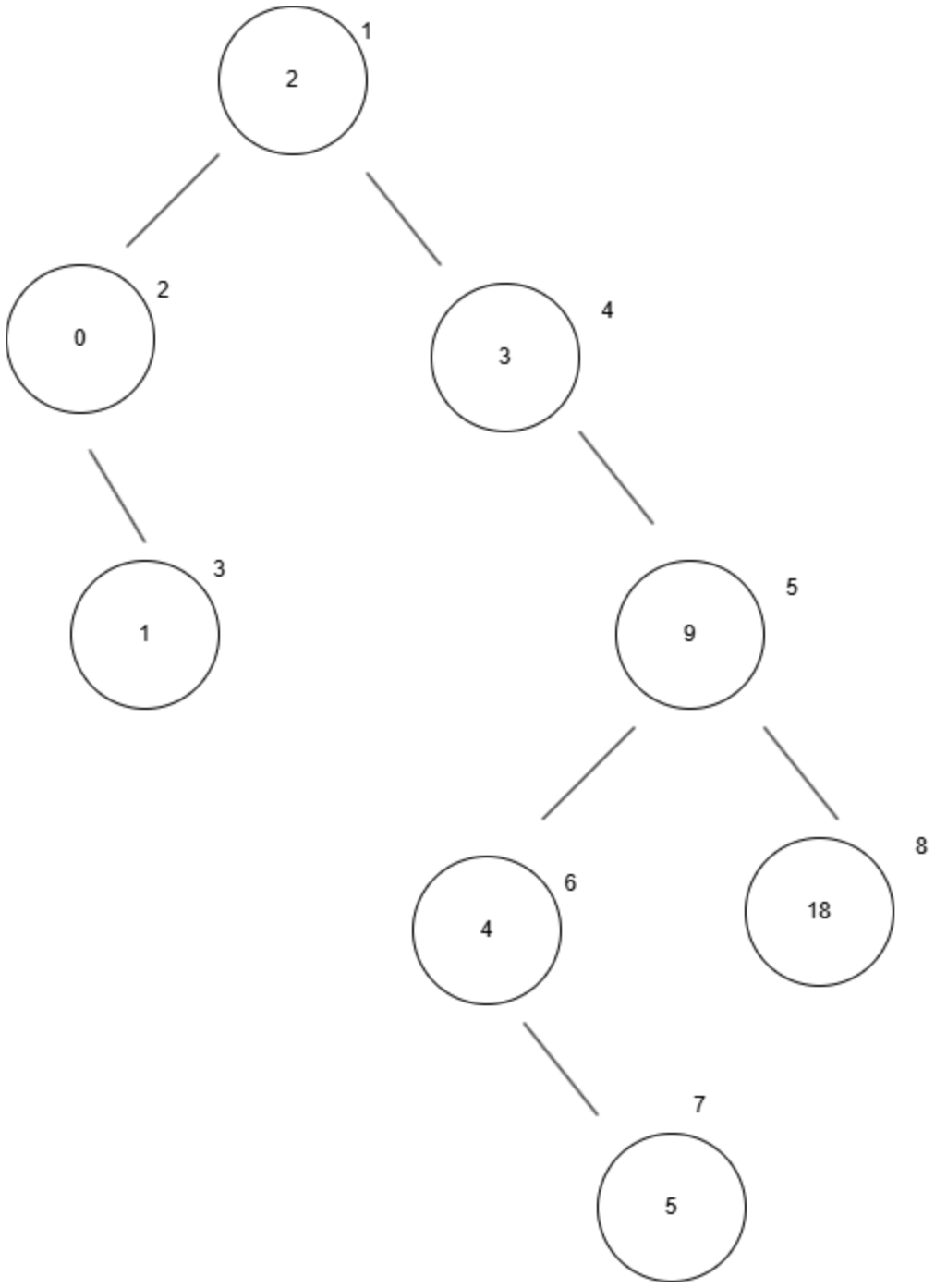
Step 1



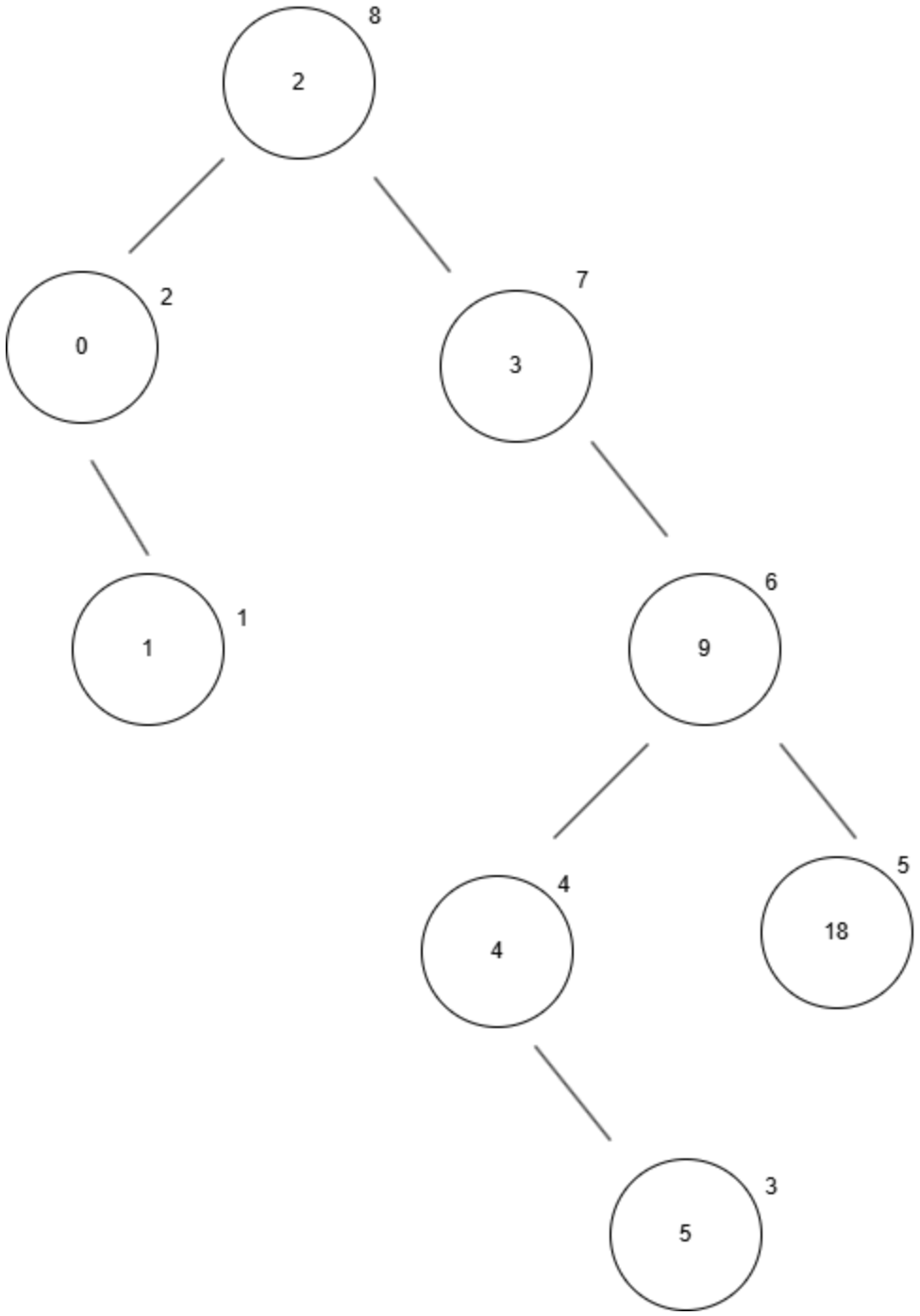
(Screenshot of tree diagram)



(Screenshot of tree diagram with indicated in-order traversal)



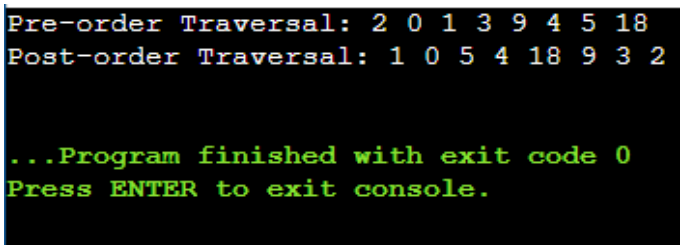
(Screenshot of tree diagram with indicated pre-order traversal)



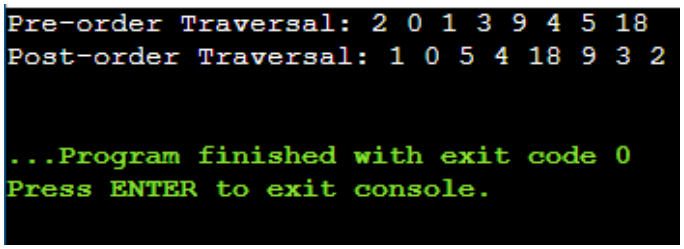
(Screenshot of tree diagram with indicated post-order traversal)

Step 2

Pre-order Traversal **(Screenshot created function)**	**void preOrder(TreeNode* node) {** **if (node != nullptr) {** **cout << node->data << " ";** **preOrder(node->left);** **preOrder(node->right);** **}**

	} }
(Screenshot console output)	

Post-order Traversal
(Screenshot created function)
	void postOrder(TreeNode* node) {
	if (node != nullptr) {
	postOrder(node->left);
	postOrder(node->right);
	cout << node->data << " ";
	}
	}

(Screenshot console output)	
Answer	**In Step 3, the code's output matches the pre-order, in-order, and post-order traversal orders shown in the Step 2 diagrams. The pre-order traversal correctly starts at the root and follows the left to right sequence as expected. The post-order traversal also follows the right order, visiting all nodes in each subtree before going back to the root. So, there's no difference between the manual diagram and the code's traversal output.**

C. Conclusion & Lessons Learned

In this activity, We learned about tree structures, including general and binary trees, and how different traversal methods (pre-order, in-order, and post-order) affect node visitation. Implementing tree nodes and building hierarchical relationships helped us understand tree depth and height better, though aligning manual results with code outputs for traversals was challenging. The supplementary task with the binary search tree further solidified my grasp on insertion and traversal logic. Overall, We believe we did well but need to focus on verifying manual and code-based outputs for consistency. Moving forward, We aim to improve my debugging skills and explore more cases more thoroughly in tree data structures.

D. Assessment Rubric

E. External References