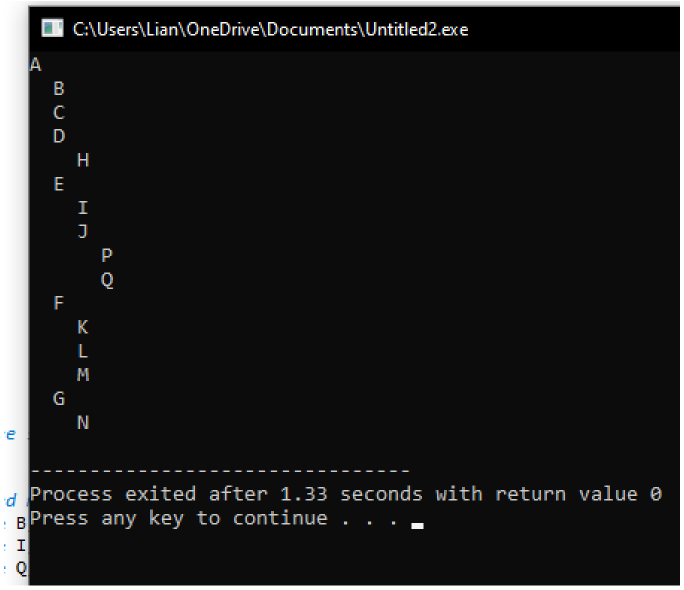
Activity No. 9	
TREE ADT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 13/11/2024
Section: CPE21S4	**Date Submitted:** 13/11/2024
Name(s): Mamaril, Justin Kenneth I, San Juan, Edson San Jose, Alexander Reyes, Alexzander Titong, Lee Ivan	**Instructor:** Prof. Maria Rizette Sayo

A. Output(s) and Observation(s)

C/C++

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

// Define the structure of a tree node
struct TreeNode {
    char data;
    vector<TreeNode*> children; // A
node can have multiple children

    // Constructor to initialize the
node
    TreeNode(char value) : data(value)
{}
};

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

// Function to print the tree in a
simple manner
void printTree(TreeNode* root, int level
= 0) {
    if (root == nullptr) return;

    // Indentation for each level
    for (int i = 0; i < level; ++i) cout
<< "  ";
    cout << root->data << endl;

    // Print each child of the node
```



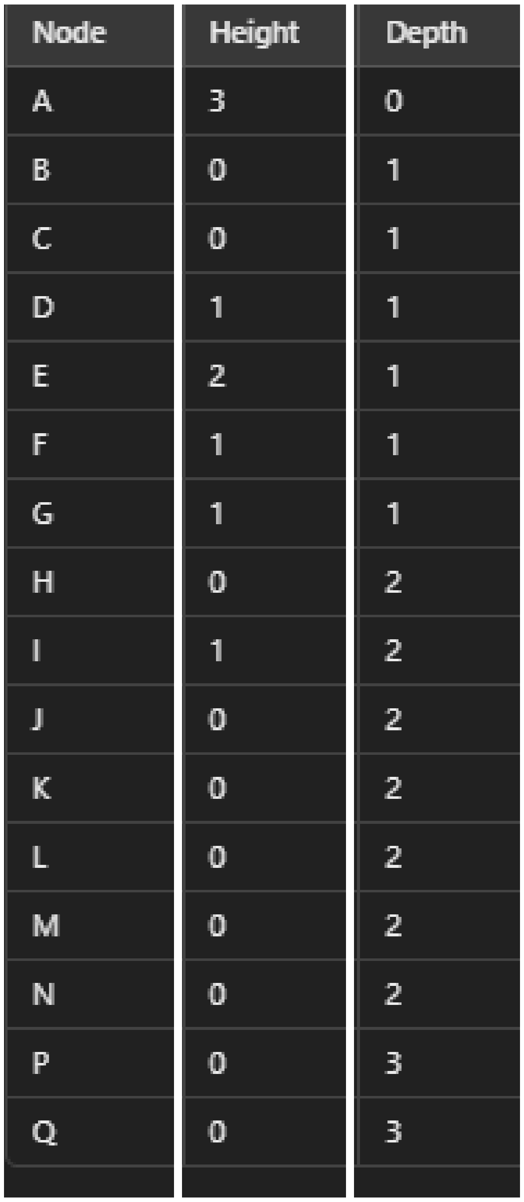
C:\Users\Lian\OneDrive\Documents\Untitled2.exe

```
A
  B
  C
  D
    H
  E
    I
    J
      P
      Q
  F
    K
    L
    M
  G
    N

--------------------------------
Process exited after 1.33 seconds with return value 0
Press any key to continue . . .
```

```
    for (TreeNode* child :
root->children) {
        printTree(child, level + 1);
    }
}

int main() {
    // Create nodes based on the tree
structure
    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');

    // Build the tree by adding children
to each node
    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(F, K);
    addChild(F, L);
    addChild(F, M);

    addChild(G, N);

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

    // Print the tree structure
    printTree(A);

    // Free allocated memory (to avoid
memory leaks)
    delete A; delete B; delete C; delete
D; delete E; delete F; delete G;
```

```
    delete H; delete I; delete J; delete
K; delete L; delete M; delete N;
    delete P; delete Q;

    return 0;
}
```

Table 9-1



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

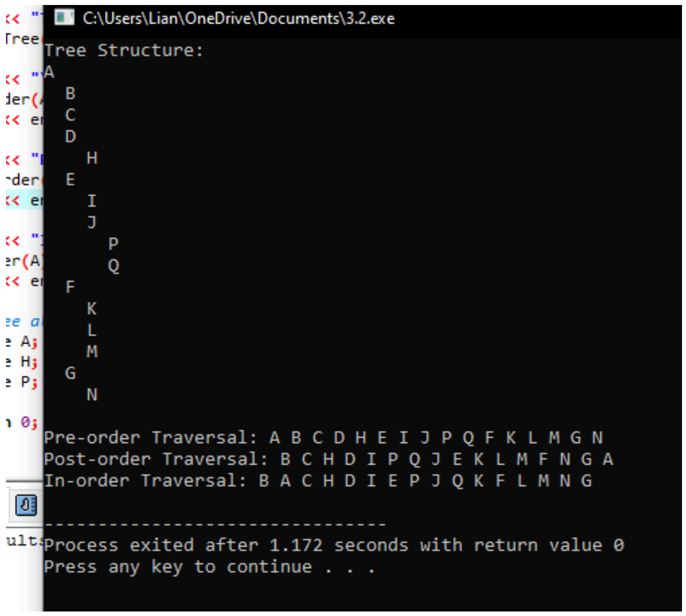
Table 9-2

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 **#Note**: In-order traversal is not typically applicable to general trees because it's traditionally defined for binary trees, where each node has a clear left and right subtree. The in-order sequence provided here is a custom traversal for this general tree structure, following an arbitrary left-to-right order.

Table 9-3

C/C++

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

struct TreeNode {
    char data;
    vector<TreeNode*> children;

    TreeNode(char value) : data(value)
{}
};

void addChild(TreeNode* parent,
TreeNode* child) {
    parent->children.push_back(child);
}

// Pre-order traversal: Visit root, then
each child recursively
```



C:\Users\Lian\OneDrive\Documents\3.2.exe

```
Tree Structure:
A
  B
  C
  D
    H
  E
    I
    J
      P
      Q
  F
    K
    L
    M
  G
    N

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

--------------------------------
Process exited after 1.172 seconds with return value 0
Press any key to continue . . .
```

```
void preOrder(TreeNode* root) {
    if (root == nullptr) return;

    cout << root->data << " ";  // Visit
root
    for (TreeNode* child :
root->children) {
        preOrder(child);         //
Visit each child
    }
}

// Post-order traversal: Visit each
child recursively, then root
void postOrder(TreeNode* root) {
    if (root == nullptr) return;

    for (TreeNode* child :
root->children) {
        postOrder(child);        //
Visit each child
    }
    cout << root->data << " ";   //
Visit root
}

// In-order traversal for N-ary tree:
Visit first child, root, then rest of
the children
void inOrder(TreeNode* root) {
    if (root == nullptr) return;

    if (!root->children.empty()) {
        inOrder(root->children[0]);  //
Visit first child (if exists)
    }
    cout << root->data << " ";       //
Visit root

    for (size_t i = 1; i <
root->children.size(); ++i) {
        inOrder(root->children[i]);  //
Visit remaining children
    }
}

void printTree(TreeNode* root, int level
= 0) {
    if (root == nullptr) return;

    for (int i = 0; i < level; ++i) cout
<< "  ";
    cout << root->data << endl;
```

OBSERVATION

Pre-order Traversal: The output order is A B C D H E I J P Q F K L M G N. This traversal visits the root first, then each child node, moving down each level from left to right. The pre-order traversal effectively reflects the hierarchical layout of the tree, beginning from the root and visiting each subtree sequentially.

Post-order Traversal: The output order is B C H D I P Q J E K L M F N G A. This traversal visits each subtree fully before the root, resulting in a bottom-up traversal order. Post-order is helpful for scenarios where processing each subtree needs to be completed before handling the parent node.

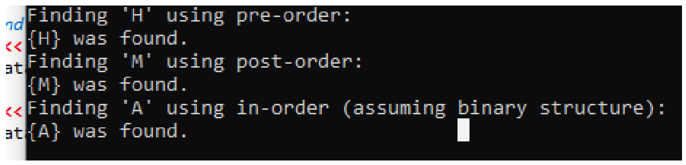
In-order Traversal (custom for N-ary trees): The output order is B A C H D I E P J Q K F L M N G. This in-order approach starts with the first child, then the root, then any remaining children. Unlike binary trees, there's no strict middle-root rule due to multiple children, so this traversal gives a root-centered, child-root-child pattern.

```
    for (TreeNode* child :
root->children) {
        printTree(child, level + 1);
    }
}

int main() {
    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');

    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(F, K);
    addChild(F, L);
    addChild(F, M);
    addChild(G, N);
    addChild(J, P);
    addChild(J, Q);

    cout << "Tree Structure:" << endl;
    printTree(A);

    cout << "\nPre-order Traversal: ";
    preOrder(A);
    cout << endl;

    cout << "Post-order Traversal: ";
    postOrder(A);
    cout << endl;

    cout << "In-order Traversal: ";
    inOrder(A);
    cout << endl;
    // Free allocated memory
    delete A; delete B; delete C; delete
D; delete E; delete F; delete G;
    delete H; delete I; delete J; delete
K; delete L; delete M; delete N;
    delete P; delete Q;

    return 0;
}
```

Table 9-4

C/C++

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

// Define the structure of a tree node
struct TreeNode {
    char data;
    vector<TreeNode*> children;

    TreeNode(char value) : data(value)
{}
};

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

// Pre-order traversal search
void findPreOrder(TreeNode* root, char
key, bool &found) {
    if (!root || found) return;

    if (root->data == key) {
        cout << "{" << key << "} was
found." << endl;
        found = true;
        return;
    }
    for (TreeNode* child :
root->children) {
```



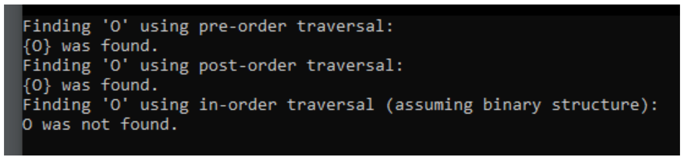
```
Finding 'H' using pre-order:
{H} was found.
Finding 'M' using post-order:
{M} was found.
Finding 'A' using in-order (assuming binary structure):
{A} was found.
```

```
        findPreOrder(child, key, found);
    }
}

// Post-order traversal search
void findPostOrder(TreeNode* root, char
key, bool &found) {
    if (!root || found) return;

    for (TreeNode* child :
root->children) {
        findPostOrder(child, key,
found);
    }
    if (root->data == key) {
        cout << "{" << key << "} was
found." << endl;
        found = true;
    }
}

// In-order traversal search (assuming
binary structure where in-order makes
sense)
void findInOrder(TreeNode* root, char
key, bool &found) {
    if (!root || found ||
root->children.size() == 0) return;

    if (root->children.size() > 0)
findInOrder(root->children[0], key,
found);
    if (root->data == key) {
        cout << "{" << key << "} was
found." << endl;
        found = true;
        return;
    }
    if (root->children.size() > 1)
findInOrder(root->children[1], key,
found);
}

// findData function that chooses the
traversal method based on CHOICE
void findData(TreeNode* root, const
string &choice, char key) {
    bool found = false;
    if (choice == "pre") {
        findPreOrder(root, key, found);
    } else if (choice == "post") {
        findPostOrder(root, key, found);
    } else if (choice == "in") {
        findInOrder(root, key, found);
    }
    if (!found) {
        cout << key << " was not found."
<< endl;
    }
}

int main() {
    // Create the tree structure
    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');

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

    addChild(G, N);

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

    // Find and display results
    cout << "Finding 'H' using
pre-order:" << endl;
    findData(A, "pre", 'H');

    cout << "Finding 'M' using
post-order:" << endl;
    findData(A, "post", 'M');
    cout << "Finding 'A' using in-order
(assuming binary structure):" << endl;
    findData(A, "in", 'A');

    // Clean up memory
    delete A; delete B; delete C; delete
D; delete E; delete F; delete G;
    delete H; delete I; delete J; delete
K; delete L; delete M; delete N;
    delete P; delete Q;

    return 0;
}
```

Table 9-5

C/C++

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

// Define the structure of a tree node
struct TreeNode {
    char data;
    vector<TreeNode*> children;

    TreeNode(char value) : data(value)
{}
};

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

// Pre-order traversal search
void findPreOrder(TreeNode* root, char
key, bool &found) {
    if (!root || found) return;

    if (root->data == key) {
        cout << "{" << key << "} was
found." << endl;
        found = true;
```



```
Finding 'O' using pre-order traversal:
{O} was found.
Finding 'O' using post-order traversal:
{O} was found.
Finding 'O' using in-order traversal (assuming binary structure):
O was not found.
```

OBSERVATION:

The output of Task 3.4 demonstrates that all traversal methods (pre-order, post-order, and adapted in-order) are capable of locating the new node "O" after it's added as a child of "G". This confirms that, in a general tree, any traversal method can be used to find a node as long as it fully explores the tree.

```
        return;
    }
    for (TreeNode* child :
root->children) {
        findPreOrder(child, key, found);
    }
}

// Post-order traversal search
void findPostOrder(TreeNode* root, char
key, bool &found) {
    if (!root || found) return;

    for (TreeNode* child :
root->children) {
        findPostOrder(child, key,
found);
    }
    if (root->data == key) {
        cout << "{" << key << "} was
found." << endl;
        found = true;
    }
}

// In-order traversal search (assuming
binary structure where in-order makes
sense)
void findInOrder(TreeNode* root, char
key, bool &found) {
    if (!root || found ||
root->children.size() == 0) return;

    if (root->children.size() > 0)
findInOrder(root->children[0], key,
found);
    if (root->data == key) {
        cout << "{" << key << "} was
found." << endl;
        found = true;
        return;
    }
    if (root->children.size() > 1)
findInOrder(root->children[1], key,
found);
}

// findData function that chooses the
traversal method based on CHOICE
void findData(TreeNode* root, const
string &choice, char key) {
    bool found = false;
    if (choice == "pre") {
        findPreOrder(root, key, found);
    } else if (choice == "post") {
        findPostOrder(root, key, found);
    } else if (choice == "in") {
        findInOrder(root, key, found);
    }
    if (!found) {
        cout << key << " was not found."
<< endl;
    }
}

int main() {
    // Create the tree structure
    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');

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

    addChild(G, N);

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

    // Task 3.4: Add new node 'O' as a
child of node 'G'
    TreeNode* O = new TreeNode('O');
    addChild(G, O);
    // Perform the findData function to
search for "O" using different
traversals
    cout << "Finding 'O' using pre-order
traversal:" << endl;
    findData(A, "pre", 'O');

    cout << "Finding 'O' using
post-order traversal:" << endl;
    findData(A, "post", 'O');

    cout << "Finding 'O' using in-order
traversal (assuming binary structure):"
<< endl;
    findData(A, "in", 'O');

    // Clean up memory
    delete A; delete B; delete C; delete
D; delete E; delete F; delete G;
    delete H; delete I; delete J; delete
K; delete L; delete M; delete N;
    delete P; delete Q; delete O;

    return 0;
}
```

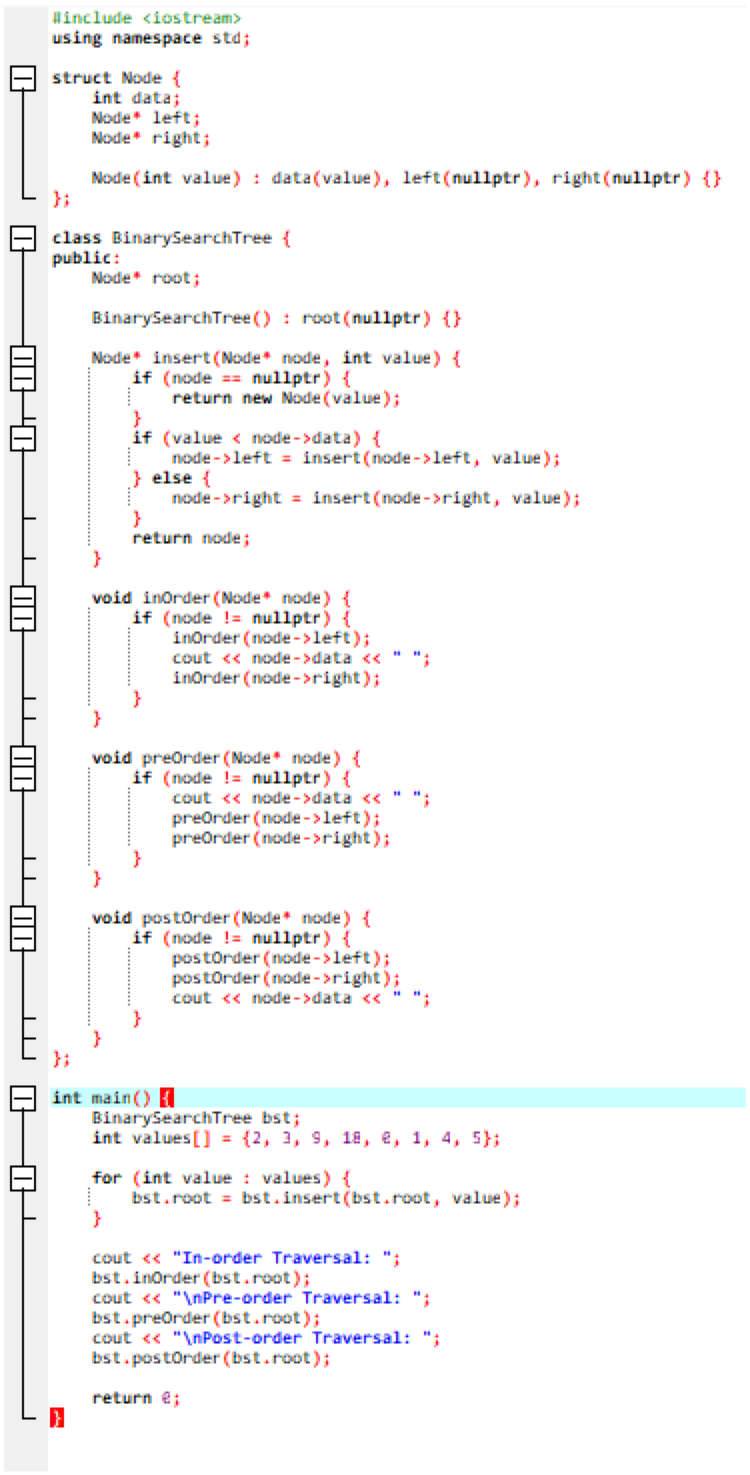
Table 9-6

B. Answers to Supplementary Activity

ILO C: Solve given problems using the tree data structure's implementation in C++

Step 1: Implement a binary search tree that will take the following values: 2, 3, 9, 18, 0, 1, 4, 5.

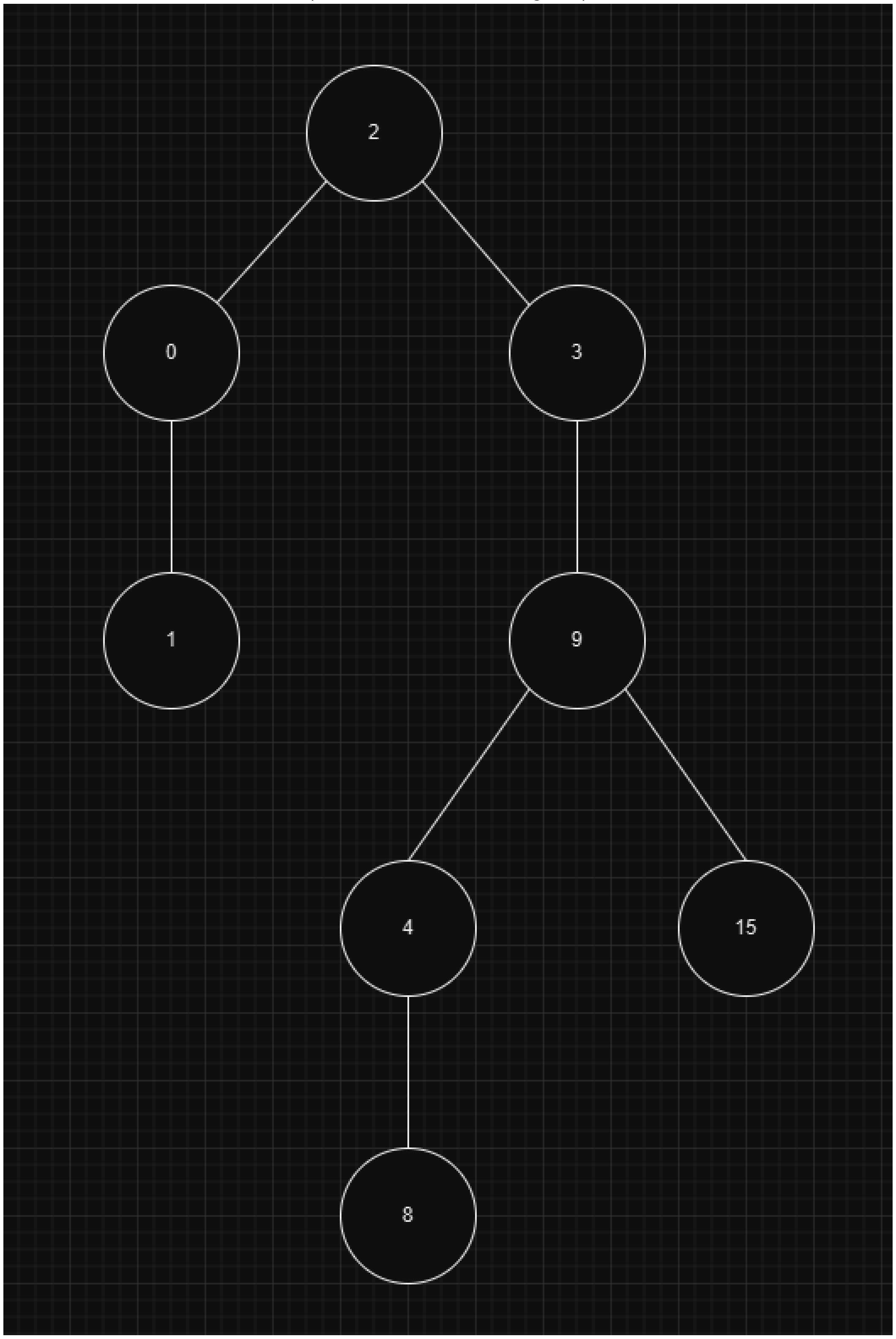
(Screenshot of code)



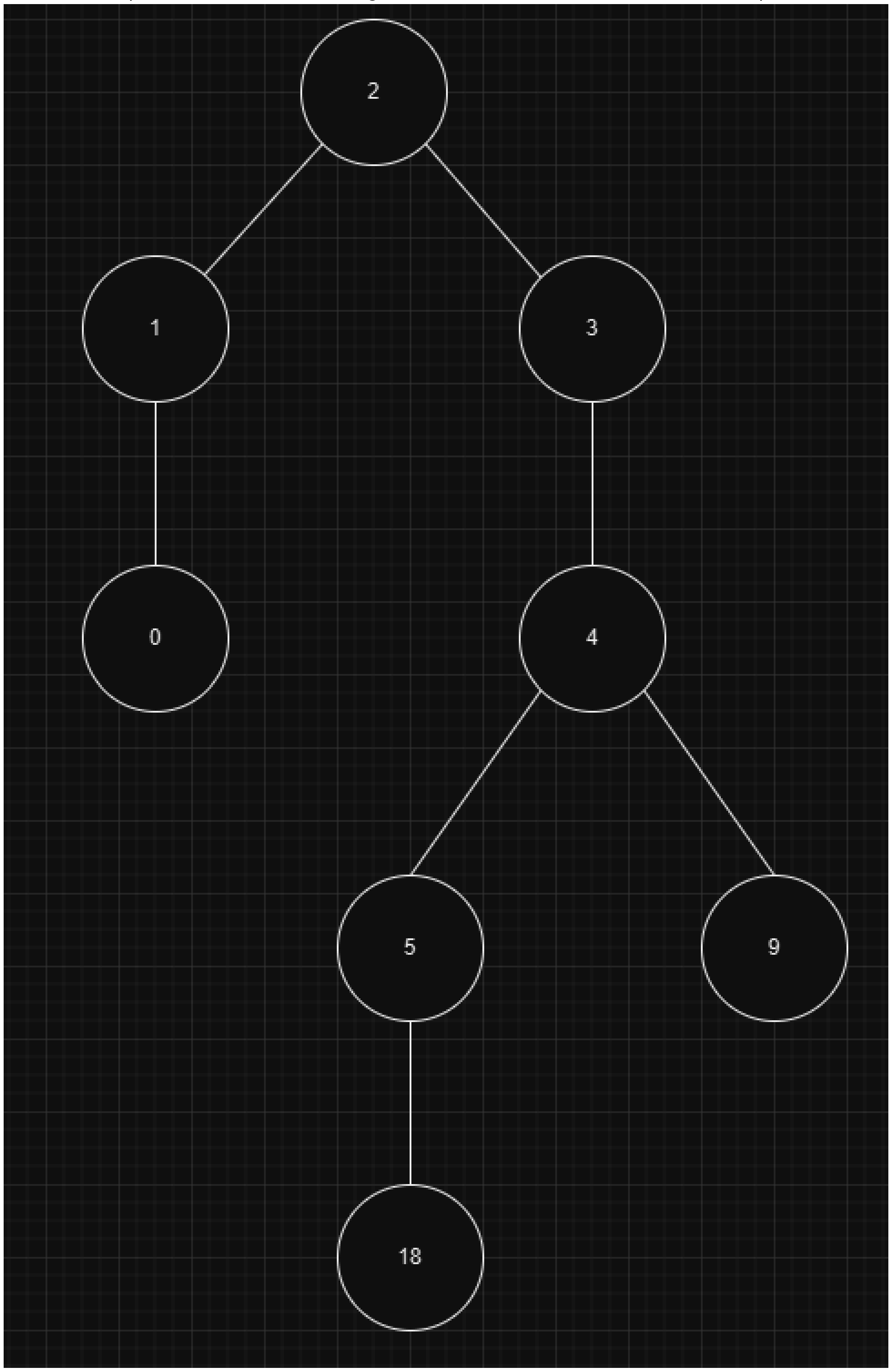
```cpp
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};

class BinarySearchTree {
public:
    Node* root;

    BinarySearchTree() : root(nullptr) {}

    Node* insert(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value);
        }
        if (value < node->data) {
            node->left = insert(node->left, value);
        } else {
            node->right = insert(node->right, value);
        }
        return node;
    }

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

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

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

int main() {
    BinarySearchTree bst;
    int values[] = {2, 3, 9, 10, 0, 1, 4, 5};

    for (int value : values) {
        bst.root = bst.insert(bst.root, value);
    }

    cout << "In-order Traversal: ";
    bst.inOrder(bst.root);
    cout << "\nPre-order Traversal: ";
    bst.preOrder(bst.root);
    cout << "\nPost-order Traversal: ";
    bst.postOrder(bst.root);

    return 0;
}
```

Step 2: Create a diagram to show the tree after all values have been inserted. Then, with the use of visual aids (like arrows and numbers) indicate the traversal order for in-order, pre-order and post-order traversal on the diagram.

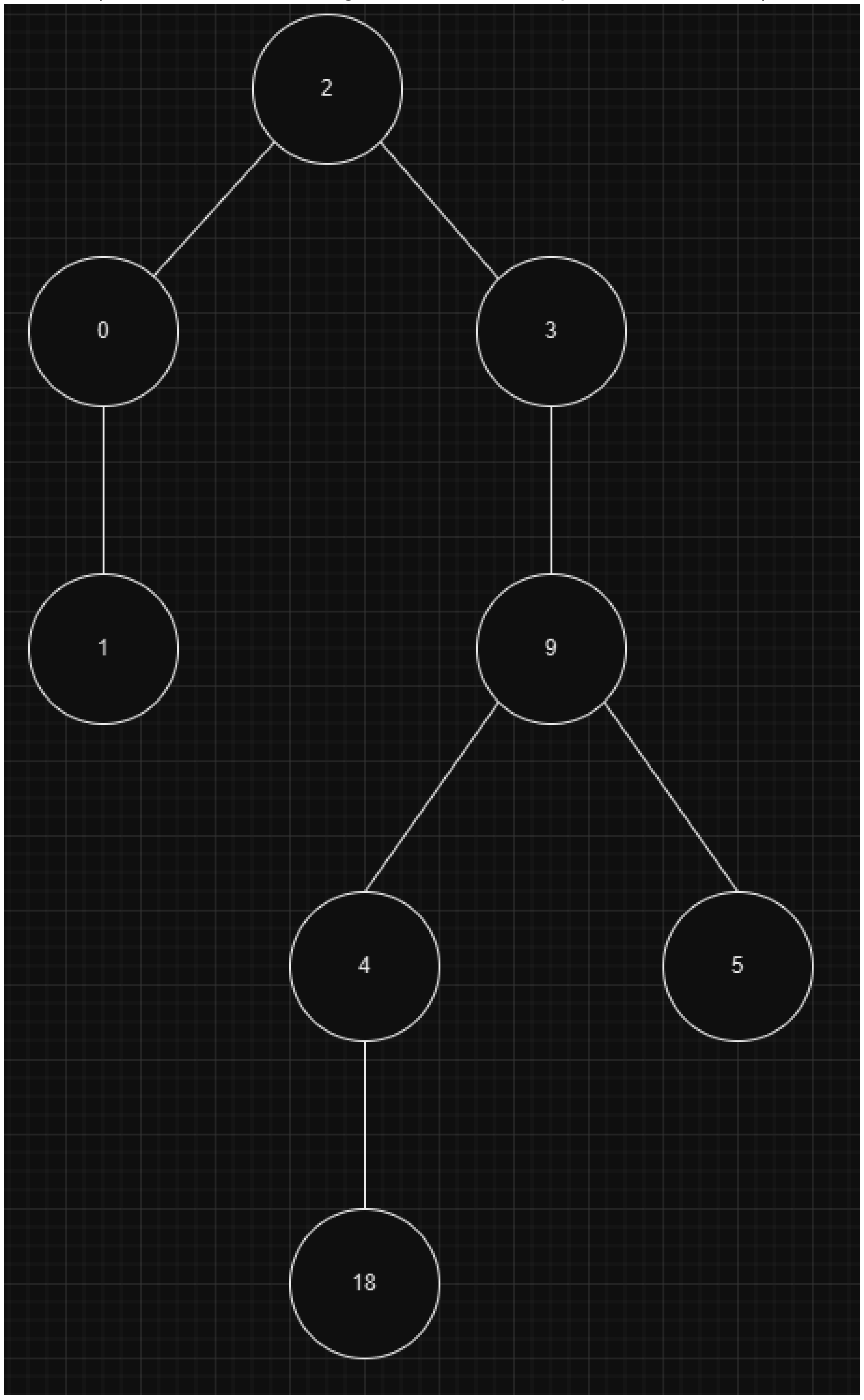
(Screenshot of tree diagram)



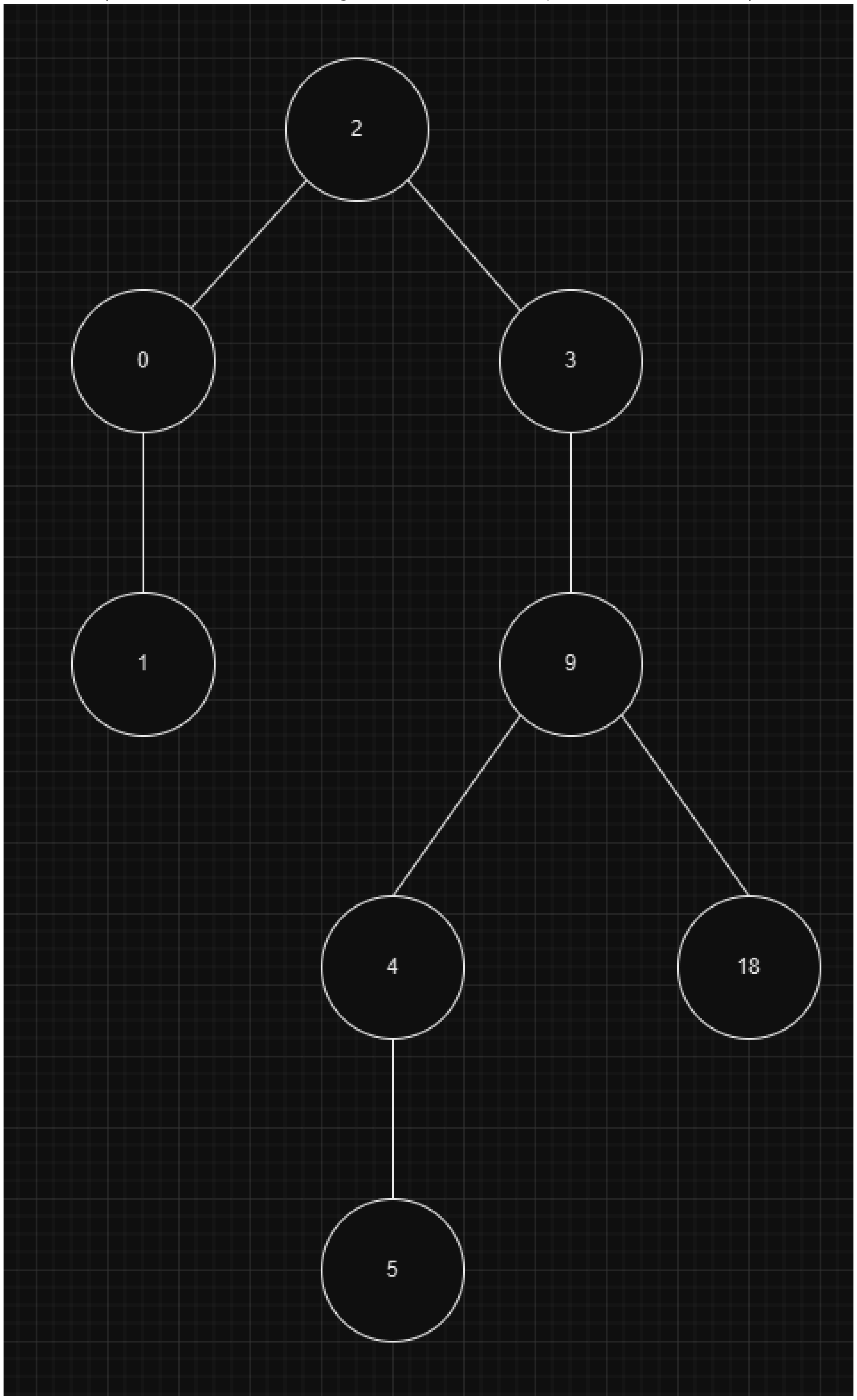
(Screenshot of tree diagram with indicated in-order traversal)



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

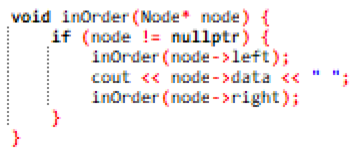


(Screenshot of tree diagram with indicated post-order traversal)

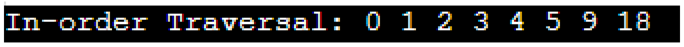


Step 3: Compare the different traversal methods. In-order traversal was performed with what function?
Function for In-Order Traversal: In the code above, inOrder is the function responsible for in-order traversal.

(Screenshot output needed)



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

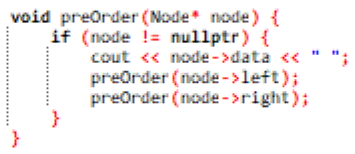


```
In-order Traversal: 0 1 2 3 4 5 9 18
```

Given the same input values above, what is the output with different traversal methods? For each output below, indicate your observation: is the output different from the pre-order and post-order traversal that you indicated in the diagrams shown in item #2.

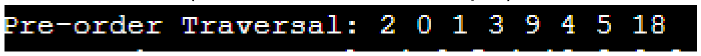
Pre-order Traversal

(Screenshot created function)



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

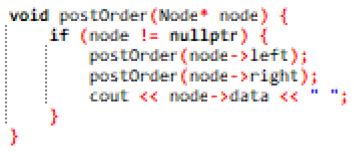
(Screenshot console output)



```
Pre-order Traversal: 2 0 1 3 9 4 5 18
```

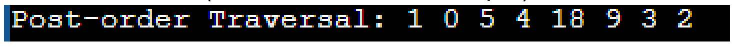
Post-order Traversal

(Screenshot created function)



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

(Screenshot console output)



```
Post-order Traversal: 1 0 5 4 18 9 3 2
```

C. Conclusion & Lessons Learned

In this activity, we learned how to create and work with tree data structures in C++. By building a general tree, binary tree, and binary search tree, we saw how trees organize data in a non-linear way. We also practiced different traversal methods (pre-order, in-order, and post-order) and learned how each one gives different results.

The supplementary activity helped us understand how binary search trees work and the benefits of each traversal method. We found that trees can store data more efficiently than lists or arrays, but they require careful handling of pointers.

Overall, we did well as a group, and each member helped us understand the concepts better. We aim to improve our skills by practicing more with complex trees and pointer management in the future.
D. Assessment Rubric
E. External References