Hands-on Activity 9.1	
Tree ADT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/3/25
Section: CPE21S4	**Date Submitted:** 10/4/25
Name(s): Avila, Vince Gabriel V.	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

ILO A:

Syntax:

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

using namespace std;

struct BinaryTreeNode {
    char data;
    BinaryTreeNode* left;
    BinaryTreeNode* right;

    BinaryTreeNode(char val) {
        data = val;
        left = nullptr;
        right = nullptr;
    }
};

int computeHeights(BinaryTreeNode* node, vector<pair<char, int>>& heights) {
    if (node == nullptr) return -1;

    int leftHeight = computeHeights(node->left, heights);
    int rightHeight = computeHeights(node->right, heights);
    int height = max(leftHeight, rightHeight) + 1;

    heights.push_back({node->data, height});
    return height;
}

void computeDepths(BinaryTreeNode* root, vector<pair<char, int>>& depths) {
    if (!root) return;

    queue<pair<BinaryTreeNode*, int>> q;
    q.push({root, 0});

    while (!q.empty()) {
        BinaryTreeNode* curr = q.front().first;
        int depth = q.front().second;
        q.pop();

        depths.push_back({curr->data, depth});

        if (curr->left != nullptr) q.push({curr->left, depth + 1});
        if (curr->right != nullptr) q.push({curr->right, depth + 1});
    }
}
```

```cpp
int main() {
    BinaryTreeNode* A = new BinaryTreeNode('A');

    A->left = new BinaryTreeNode('B');
    A->right = new BinaryTreeNode('C');

    A->left->left = new BinaryTreeNode('D');
    A->left->right = new BinaryTreeNode('E');

    A->right->left = new BinaryTreeNode('F');
    A->right->right = new BinaryTreeNode('G');

    A->left->left->left = new BinaryTreeNode('H');
    A->left->right->left = new BinaryTreeNode('I');
    A->left->right->right = new BinaryTreeNode('J');

    A->left->right->right->left = new BinaryTreeNode('P');
    A->left->right->right->right = new BinaryTreeNode('Q');

    A->right->left->left = new BinaryTreeNode('K');
    A->right->left->right = new BinaryTreeNode('L');

    A->right->right->left = new BinaryTreeNode('M');
    A->right->right->right = new BinaryTreeNode('N');

    vector<pair<char, int>> depths;
    vector<pair<char, int>> heights;

    computeDepths(A, depths);
    computeHeights(A, heights);

    sort(depths.begin(), depths.end());
    sort(heights.begin(), heights.end());

    cout << "Node\tHeight\tDepth\n";
    for (char ch = 'A'; ch <= 'Q'; ++ch) {
        if (ch == 'O') continue;

        int depth = -1, height = -1;

        for (auto& d : depths) {
            if (d.first == ch) depth = d.second;
        }

        for (auto& h : heights) {
            if (h.first == ch) height = h.second;
        }

        cout << ch << "\t" << height << "\t" << depth << "\n";
    }

    return 0;
}
```

Output:

Node	Height	Depth
A	4	0
B	3	1
C	2	1
D	1	2

E	**2**	**2**
F	**1**	**2**
G	**1**	**2**
H	**0**	**3**
I	**0**	**3**
J	**1**	**3**
K	**0**	**3**
L	**0**	**3**
M	**0**	**3**
N	**0**	**3**
O	**0**	**3**
P	**0**	**4**
Q	**0**	**4**

Analysis:

This code builds a binary tree and computes the height and depth of each node. It takes a recursive approach for computing the height of each node, and a queue-based approach for computing the depth of each node level by level. Once this information has been collected, it organizes it and displays the information for the height and depth of each node, that is node A through node Q (excluding O), so you can easily see how high or deep each node is in the tree.

ILO B:

Synatax:

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

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

TreeNode* createNode(char val) {
    TreeNode* node = new TreeNode();
    node->data = val;
    return node;
}

void preorder(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->data << " ";
    for (auto child : root->children) {
        preorder(child);
    }
}

void postorder(TreeNode* root) {
    if (root == nullptr) return;
    for (auto child : root->children) {
        postorder(child);
    }
    cout << root->data << " ";
}

void inorder(TreeNode* root) {
    if (root == nullptr) return;
    if (!root->children.empty()) inorder(root->children[0]);
    cout << root->data << " ";
    for (int i = 1; i < root->children.size(); i++) {
        inorder(root->children[i]);
    }
}

bool searchData(TreeNode* root, char key) {
    if (root == nullptr) return false;

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

    for (auto child : root->children) {
        if (searchData(child, key))
            return true;
    }
    return false;
}
```

```cpp
int main() {
    TreeNode* A = createNode('A');
    TreeNode* B = createNode('B');
    TreeNode* C = createNode('C');
    TreeNode* D = createNode('D');
    TreeNode* E = createNode('E');
    TreeNode* F = createNode('F');
    TreeNode* G = createNode('G');
    TreeNode* H = createNode('H');
    TreeNode* I = createNode('I');
    TreeNode* J = createNode('J');
    TreeNode* K = createNode('K');
    TreeNode* L = createNode('L');
    TreeNode* M = createNode('M');
    TreeNode* N = createNode('N');
    TreeNode* P = createNode('P');
    TreeNode* Q = createNode('Q');
    TreeNode* O = createNode('O'); // new leaf

    A->children = {B, C, D, E, F, G};
    D->children = {H};
    E->children = {I, J};
    J->children = {P, Q};
    F->children = {K, L, M};
    G->children = {N, O}; // new child added to G

    cout << "Pre-order Traversal: ";
    preorder(A);

    cout << "\nPost-order Traversal: ";
    postorder(A);

    cout << "\nIn-order Traversal: ";
    inorder(A);

    // Search for node O
    cout << "\n\nSearching for node O..." << endl;
    searchData(A, 'O');

    return 0;
}
```

Output:

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

Searching for node O...
O was found!
```

Analysis:

This code builds a tree in which each node may have multiple children. It provides a way to traverse the tree using pre-order, post-order, and in-order strategies, as well as a search function to search for a particular node like 'O'. This will help me practice working with trees, and searching for data.

B. Answers to Supplementary Activity

ILO C:

Syntax:

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

struct Node {
    int num;
    Node* left;
    Node* right;
};

Node* makeNode(int x) {
    Node* p = new Node();
    p->num = x;
    p->left = p->right = NULL;
    return p;
}

Node* insert(Node* root, int x) {
    if (!root) return makeNode(x);
    if (x < root->num)
        root->left = insert(root->left, x);
    else
        root->right = insert(root->right, x);
    return root;
}

void inorder(Node* root) {
    if (!root) return;
    inorder(root->left);
    cout << root->num << " ";
    inorder(root->right);
}

void preorder(Node* root) {
    if (!root) return;
    cout << root->num << " ";
    preorder(root->left);
    preorder(root->right);
}

void postorder(Node* root) {
    if (!root) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->num << " ";
}

int main() {
    Node* root = NULL;
    int data[] = {2, 3, 9, 18, 0, 1, 4, 5};
    int n = sizeof(data) / sizeof(data[0]);

    cout << "Inserting values into the Binary Search Tree:\n";
    for (int i = 0; i < n; i++) {
        cout << data[i] << " ";
        root = insert(root, data[i]);
    }

    cout << "\n\nTree Traversals:\n";
    cout << "In-order Traversal (L, Root, R): ";
    inorder(root);
    cout << "\nPre-order Traversal (Root, L, R): ";
    preorder(root);
    cout << "\nPost-order Traversal (L, R, Root): ";
    postorder(root);
    cout << "\n";

    return 0;
}
```

Output:

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

Searching for node O...
O was found!
```

Analysis:

This program builds a binary search tree by adding numbers sequentially. It then visits the numbers and prints them in three different ways: in-order, pre-order, and post-order. This helps me see how to structure and examine data that is organized into a tree and demonstrates how the order in which I visit those nodes vary based on which method I used to traverse the tree.

C. Conclusion & Lessons Learned

In conclusion, it helped me learn how to manipulate various types of trees in C++. The first one taught me how to calculate the height and depth of nodes in a binary tree. The second taught me how to traverse nodes in a tree with many children. The last one allowed me to create and traverse a binary search tree. With all of the activities complete, I now have a better understanding of how trees are structured and how to traverse a tree in different ways.

D. Assessment Rubric

E. External References

GeeksforGeeks. (2025d, August 2). *Binary Tree Data structure*. GeeksforGeeks.

https://www.geeksforgeeks.org/dsa/binary-tree-data-structure/

GeeksforGeeks. (2025b, July 23). *Inorder traversal of binary tree*. GeeksforGeeks.

https://www.geeksforgeeks.org/dsa/inorder-traversal-of-binary-tree/

GeeksforGeeks. (2025b, July 23). *Check if two nodes are cousins in a Binary Tree*. GeeksforGeeks.

https://www.geeksforgeeks.org/dsa/check-two-nodes-cousins-binary-tree/