Activity No. <9>	
TREES	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/4/25**
Section: CPE21S4	**Date Submitted: 20/4/25**
Name(s): Angelo Quioyo	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s):

ILO A:

Main CPP:

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

using namespace std;

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

    TreeNode(char val) {
        data = val;
    }
};

TreeNode* createNode(char val) {
    return new TreeNode(val);
}

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

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

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

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

        for (TreeNode* child : current->children) {
            q.push({child, depth + 1});
        }
    }
}

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

    int maxChildHeight = -1;

    for (TreeNode* child : node->children) {
        int childHeight = computeHeights(child, heights);
        if (childHeight > maxChildHeight) {
            maxChildHeight = childHeight;
        }
    }

    int height = maxChildHeight + 1;
    heights.push_back({node->data, height});
    return height;
}
```

```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');

    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};

    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       3       0
B       0       1
C       0       1
D       1       1
E       2       1
F       1       1
G       1       1
H       0       2
I       0       2
J       1       2
K       0       2
L       0       2
M       0       2
N       0       2
P       0       3
Q       0       3
```

Explanation:
This C++ program defines a generic tree structure where each node can have multiple children, using a TreeNode struct. It calculates both the depth (distance from the root) and height (longest path to a leaf) for each node in the tree. The computeDepths function performs a level-order traversal using a queue, while computeHeights uses a recursive post-order approach. After constructing the tree and computing values, the results are sorted and printed in a table format for nodes A through Q (excluding O). Overall, the code effectively demonstrates tree traversal techniques and how to compute key node metrics.

Table 9-2:

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

ILO B:

Pre Order	A B C D H E I J P Q F K L M G N O
Post Order	B C H D I P Q J E K L M F N O G A
In Order	B A C D H E I J P Q F K L M G N O

Table 9-4 to Table 9-6:

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

struct Node {
    char val;
    vector<Node*> child;
};

Node* makeNode(char v) {
    Node* n = new Node();
    n->val = v;
    return n;
}

void pre(Node* root) {
    if (!root) return;
    cout << root->val << " ";
    for (auto c : root->child)
        pre(c);
}

void post(Node* root) {
    if (!root) return;
    for (auto c : root->child)
        post(c);
    cout << root->val << " ";
}

void in(Node* root) {
    if (!root) return;
    if (!root->child.empty())
        in(root->child[0]);
    cout << root->val << " ";
    for (int i = 1; i < root->child.size(); i++)
        in(root->child[i]);
}

bool find(Node* root, char key) {
    if (!root) return false;
    if (root->val == key) {
        cout << key << " was found!" << endl;
        return true;
    }
    for (auto c : root->child)
        if (find(c, key)) return true;
    return false;
}

int main() {
    Node* A = makeNode('A');
    Node* B = makeNode('B');
    Node* C = makeNode('C');
    Node* D = makeNode('D');
    Node* E = makeNode('E');
    Node* F = makeNode('F');
    Node* G = makeNode('G');
    Node* H = makeNode('H');
    Node* I = makeNode('I');
    Node* J = makeNode('J');
    Node* K = makeNode('K');
    Node* L = makeNode('L');
    Node* M = makeNode('M');
    Node* N = makeNode('N');
    Node* P = makeNode('P');
    Node* Q = makeNode('Q');
    Node* O = makeNode('O');

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

    cout << "Pre-order: ";
    pre(A);
    cout << "\nPost-order: ";
    post(A);
    cout << "\nIn-order: ";
    in(A);

    cout << "\n\nFinding node O...\n";
    find(A, 'O');

    return 0;
}
```

Output:

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

Finding node O...
O was found!
```

Explanation

This C++ program create a generic tree using a Node structure where each node can have multiple children, and it implements three tree traversal methods: pre-order, post-order, and a customized in-order. The pre, post, and in functions perform depth-first traversals, printing nodes in different sequences depending on the traversal type. It also includes a find function that recursively searches for a specific node by value and confirms when it's found. In main(), a tree rooted at node 'A' is built, and node 'O' is included as a child of 'G' to demonstrate the search functionality. This code is a solid example of recursive tree traversal and node searching in n-ary trees.

B. Answers to Supplementary Activity:

Main CPP:

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

struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
};

TreeNode* create(int value) {
    TreeNode* node = new TreeNode();
    node->data = value;
    node->left = node->right = NULL;
    return node;
}

TreeNode* insert(TreeNode* root, int value) {
    if (root == NULL)
        return create(value);

    if (value < root->data)
        root->left = insert(root->left, value);
    else
        root->right = insert(root->right, value);

    return root;
}

void inorder(TreeNode* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

void preorder(TreeNode* root) {
    if (root == NULL) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

void postorder(TreeNode* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

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

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

different traversal methods:

```
Tree Traversals:
In-order Traversal (L, Root, R): 0 1 2 3 4 5 9 18
Pre-order Traversal (Root, L, R): 2 0 1 3 9 4 5 18
Post-order Traversal (L, R, Root): 1 0 5 4 18 9 3 2
```

Inorder traversal:

```
In-order Traversal (L, Root, R): 0 1 2 3 4 5 9 18
```

Pre Order:

```
Pre-order Traversal (Root, L, R): 2 0 1 3 9 4 5 18
```

Post order:

```
Post-order Traversal (L, R, Root): 1 0 5 4 18 9 3 2
```

C. **Conclusion & Lessons Learned:**
Implementing the tree data structure in C++ was challenging but successful. I now know how to build different kinds of trees, including a general tree, a binary tree, and a binary search tree (BST). These structures are crucial for efficiently sorting and organizing data. I also learned about the various tree traversal methods: pre-order, in-order, and post-order. The in-order traversal is particularly important as it allows for the retrieval of elements from a Binary Search Tree in ascending order, which is a key property of the structure.

D. Assessment Rubric

E. External References