ACTIVITY NO. 9

TREES	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/04/25**
Section: CPE21S4	**Date Submitted: 10/04/25**
Name: Cruz, Axl Waltz E.	**Instructor:** Engr. Jimlord Quejado

1. Objective(s)

Create C++ code for the implementation of the tree data structure.

2. Intended Learning Outcomes (ILOs)

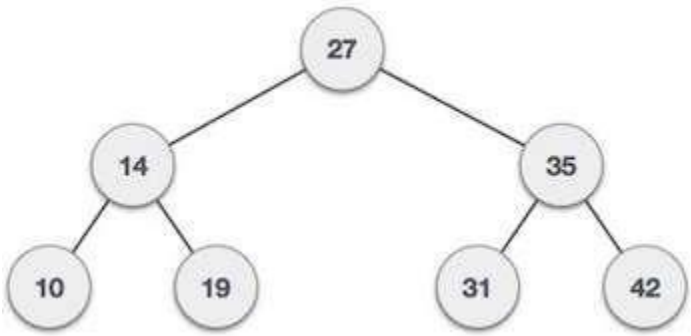
After this activity, the student should be able to:

- Create C++ code to implement a general tree, binary tree and binary search tree.
- Create C++ code for implementation of tree traversal methods such as pre-order, in-order and post- order traversal.
- Solve given problems using the tree data structure's implementation in C++

3. Discussion

Trees belong to the class of non-linear data structures. Unlike linear data structures, such as stacks or lists, where elements are stored in linear order, non-linear data structures store elements hierarchically. Thus, the traversal of a tree or graph cannot be completed in a single run.

A typical example of a tree is shown below.



Example of a binary tree

The elements inside the circles are called the nodes of the tree. A binary tree is a data structure where every node has at most two children. The topmost node is called the root node. A node with no children is called a leaf. Here the root node is at 27. The node at 14 is the left child of node 27, and the node at 35 is the right child. Similarly, the left and right children of the parent node 14 are 10 and 19 respectively. The leaves in the above example are 10, 19, 31, and 42. The height of the tree is defined as the longest path from the root to a leaf. In the above case, the height of the tree is 2.

The advantage of trees is that memory is utilized in an efficient way. In the case of a stack, deleting a middle element is difficult since we must shift the elements to the right of the stack. But in the case of a tree, we can delete the node x and modify its parent node to point to children of x.

Types of Trees

- Full binary tree: A binary tree is a full binary tree if every node has 0 or 2 children.
- Complete binary tree: A binary tree is a complete binary tree if all the levels are completely filled, except possibly the last level, and the last level has all keys as left as possible.
- Perfect binary tree: A binary tree is a perfect binary tree if all the internal nodes have two children and all leaf nodes are at the same level.
- Balanced binary tree: A binary tree is balanced if the height of the tree is $O(Log\ n)$ where n is the number of nodes.
- Binary search tree: In a binary search tree, a node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent's value.

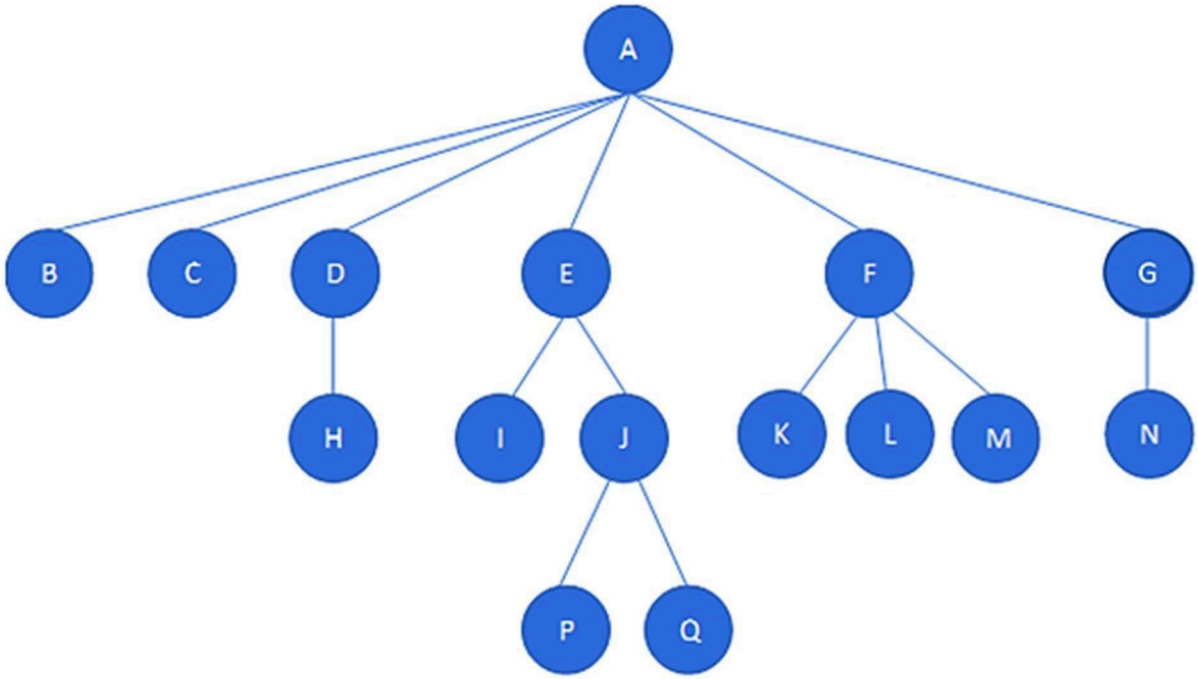
4. Materials and Equipment

Personal Computer with C++ IDE
Recommended IDE:

- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to C++17)

5. Procedure

ILO A: Create C++ code to implement a general tree, binary tree and binary search tree.



Task 1: Create code in C++ that will create a tree as shown in the figure above. Use linked lists as the internal representation of this tree. **Indicate your code screenshot and comments in table 9-1.**

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

using namespace std;

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

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

int computeHeights(BinaryTreeNode* node, vector<pair<char, int>>& heights) {
    if (!node) 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) {
    queue<pair<BinaryTreeNode*, int>> q;
    q.push({root, 0});

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

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

        if (node->left) q.push({node->left, depth + 1});
        if (node->right) q.push({node->right, depth + 1});
    }
}
```

```
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;
}
```

```
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
P       0       4
Q       0       4
```

- This code builds a binary tree with nodes labeled from A to Q and calculates the height and depth of each node. It uses recursion to compute the height of each node and a queue for level order traversal to compute depths. Finally, it prints each node along with its height and depth values.

Task 2: Complete the following table:

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	0	3
K	0	3
L	0	3
M	0	3
N	0	3
P	0	4
Q	0	4

Pre-order	A B C D H E I J P Q K L M O 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

Include this completed table as table 9-2 in section 6.

ILO B: Create C++ code for implementation of tree traversal methods such as pre-order, in-order and post- order traversal.

Task 3: After implementing the code for above, answer the following:

3.1 Given the tree diagram, find the result of the pre-order, post-order and in-order traversal strategies by hand.
Include this as table 9-3 in section 6.

3.2 Create a function for pre-order, post-order and in-order traversal. Make sure that each function displays an output into the console. Once you have the output, **create and fill table 9-4 in section 6** so that it contains the screenshot of the function, screenshot of the output and your observations. Your observations consist of a comparison between output in #1 and the output of your functions in #2.

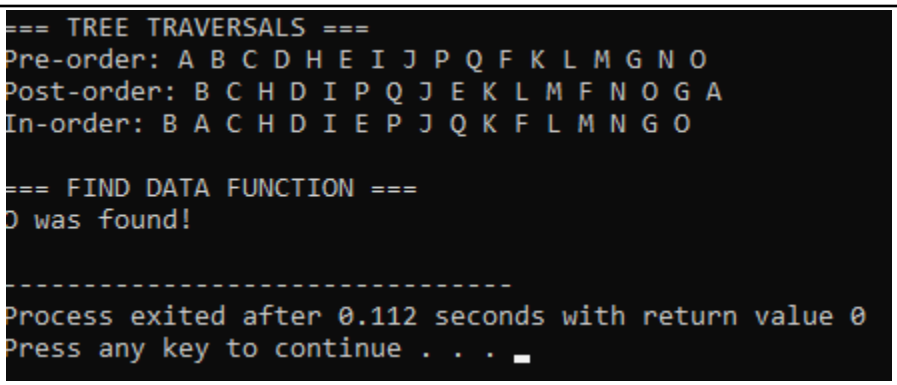
3.3. Create a function called `findData` that will take the following parameter: "`CHOICE, KEY`". `Choice` will determine which traversal method will be used to traverse the tree to find the

second parameter `KEY`. If the `KEY` is found, the value will be displayed such that: “{KEY} was found!”. Otherwise, no output. **Include your output as table 9-5 in section 6.**

3.4. Add a new leaf node called which will contain the data value of **"O"**. Make this node the child of nodeG. Perform the **findData** function you created in #3, what can you observe about the output? **Include your output and answers as table 9-6 in section 6.**

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

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

Node* newNode(char data) {
    Node* node = new Node();
    node->data = data;
    return node;
}

// === Build the tree ===
Node* buildTree() {
    Node* A = newNode('A');
    Node* B = newNode('B');
    Node* C = newNode('C');
    Node* D = newNode('D');
    Node* E = newNode('E');
    Node* F = newNode('F');
    Node* G = newNode('G');
    Node* H = newNode('H');
    Node* I = newNode('I');
    Node* J = newNode('J');
    Node* K = newNode('K');
    Node* L = newNode('L');
    Node* M = newNode('M');
    Node* N = newNode('N');
    Node* P = newNode('P');
    Node* Q = newNode('Q');
    Node* O = newNode('O'); // new node

    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}; // added O

    return A;
}
```

```cpp
void preOrder(Node* root) {
    if (!root) return;
    cout << root->data << " ";
    for (auto c : root->children)
        preOrder(c);
}

void postOrder(Node* root) {
    if (!root) return;
    for (auto c : root->children)
        postOrder(c);
    cout << root->data << " ";
}

void inOrder(Node* root) {
    if (!root) 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 findData(Node* root, char key) {
    if (!root) return false;
    if (root->data == key) {
        cout << key << " was found!" << endl;
        return true;
    }
    for (auto c : root->children)
        if (findData(c, key)) return true;
    return false;
}

int main() {
    Node* root = buildTree();

    cout << "=== TREE TRAVERSALS ===" << endl;
    cout << "Pre-order: ";
    preOrder(root);
    cout << "\nPost-order: ";
    postOrder(root);
    cout << "\nIn-order: ";
    inOrder(root);

    cout << "\n\n=== FIND DATA FUNCTION ===" << endl;
    findData(root, 'O');

    return 0;
}
```



- This code builds a general tree where each node can have multiple children and assigns character labels from A to O. It performs and prints pre order, post order, and in order traversals of the tree. It also includes a function to search for a specific character and prints a message if it is found.

6. Output

7. 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)

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?

(Screenshot output needed)

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) (Screenshot console output)

Post-order Traversal

(Screenshot created function) (Screenshot console output)

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

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

Node* newNode(int val) {
    Node* temp = new Node();
    temp->value = val;
    temp->left = temp->right = NULL;
    return temp;
}

Node* insert(Node* root, int val) {
    if (root == NULL)
        return newNode(val);

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

    return root;
}

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

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

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

int main() {
    Node* root = NULL;
```

```
    return root;
}

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

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

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

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

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

```
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
```

```
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
```

- This code creates a Binary Search Tree by inserting values from an array. It then performs and prints inorder, preorder, and postorder traversals of the tree. Each traversal visits the nodes in a different order like left root right or root left right.

8. Conclusion

Provide the following:

- Summary of lessons learned
  - I learned how to build different types of trees such as binary trees and general trees using nodes and pointers. I also learned how to perform different tree traversals like pre order, in order, and post order, and how to calculate height, depth, and search values in a tree.
- Analysis of the procedure
  - The procedures used in the code followed logical steps such as creating nodes, linking children, and applying recursion or loops for traversal and search. Each function had a clear purpose and helped demonstrate how tree structures work.
- Analysis of the supplementary activity
  - The extra activities like computing height, depth, and finding data added more depth to understanding tree properties. These helped apply knowledge beyond basic traversal and improved my thinking on how trees are used in real problems.
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?
  - I think I did well in this activity because I understood how each part of the code works and how to build and explore trees. My area for improvement is practicing more complex trees and understanding how to handle edge cases like empty trees or duplicate nodes.

9. Assessment Rubric