Activity No. 9.1	
Tree ADT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 11/13/2024**
Section: CPE21S1	**Date Submitted: 11/13/2024**
Name(s): Kurt Gabriel Anduque **Jhon Hendricks Bautista** **Matt Clemence Magboo** **Christian Dale Pateña** **Redj Guillian Bonifacio**	**Instructor: Mrs. Maria Rizette Sayo**
A. Output(s) and Observation(s)	
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.	

```cpp
#include <iostream>
#include <vector>
#include <string>

class TreeNode {
public:
    char data;
    std::vector<TreeNode*> children;

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

    void addChild(TreeNode* child) {
        children.push_back(child);
    }
};

TreeNode* buildTree() {
    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');

    A->addChild(B);
    A->addChild(C);
    A->addChild(D);
    A->addChild(E);
    A->addChild(F);
    A->addChild(G);

```

```

    D->addChild(H);
    D->addChild(I);
    D->addChild(J);

    F->addChild(K);
    F->addChild(L);
    F->addChild(M);

    G->addChild(N);

    J->addChild(P);
    J->addChild(Q);

    return A;
}

int main() {
    TreeNode* root = buildTree();
    std::cout << "Tree built with root node: " << root->data << std::endl;
    return 0;
}

```

Task 2: Complete the following table:

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

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

order traversal.

Task 3

3.1.

Pre-order	1, 2, 4, 5, 3, 6, 7
Post-order	4, 2, 5, 1, 6, 3, 7
In-order	4, 5, 2, 6, 7, 3, 1

Table 9-3

3.2

Function	Screenshot of Function and Output	Observations

Pre-order Traversal	`void preOrder(Node* node) {`
	`    if (node == nullptr) {`
	`        return;`
	`    }`
	`    cout << node->data << " ";`
	`    preOrder(node->left);`
	`    preOrder(node->right);`
	`}`
	
	`Pre-order traversal: 1 2 4 5 3 6 7`
		Matches expected pre-order output from Task 3.1.

In-order Traversal	`void inOrder(Node* node) {`
	`    if (node == nullptr) {`
	`        return;`
	`    }`
	`    inOrder(node->left);`
	`    cout << node->data << " ";`
	`    inOrder(node->right);`
	`}`
	
	`In-order traversal: 4 2 5 1 6 3 7`
		Matches expected in-order output from Task 3.1.

Post-order Traversal	`void postOrder(Node* node) {`
	`    if (node == nullptr) {`
	`        return;`
	`    }`
	`    postOrder(node->left);`
	`    postOrder(node->right);`
	`    cout << node->data << " ";`
	`}`
	
	`Post-order traversal: 4 5 2 6 7 3 1`
		Matches expected post-order output from Task 3.1.

Table 9-4

3.3

Source Code

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

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

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

void preOrder(Node* node, int key, bool &found) {
    if (node == nullptr || found) {
        return;
    }
    if (node->data == key) {
        cout << key << " was found!" << endl;
        found = true;
        return;
    }
    preOrder(node->left, key, found);
    preOrder(node->right, key, found);
}

void inOrder(Node* node, int key, bool &found) {
    if (node == nullptr || found) {
        return;
    }
    inOrder(node->left, key, found);
    if (node->data == key) {
        cout << key << " was found!" << endl;
        found = true;
        return;
    }
    inOrder(node->right, key, found);
}
```

```
}

void postOrder(Node* node, int key, bool &found) {
    if (node == nullptr || found) {
        return;
    }
    postOrder(node->left, key, found);
    postOrder(node->right, key, found);
    if (node->data == key) {
        cout << key << " was found!" << endl;
        found = true;
        return;
    }
}

void findData(Node* root, int choice, int key) {
    bool found = false;
    switch (choice) {
        case 1:
            preOrder(root, key, found);
            break;
        case 2:
            inOrder(root, key, found);
            break;
        case 3:
            postOrder(root, key, found);
            break;
        default:
            cout << "Invalid choice!" << endl;
    }
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    root->right->right = new Node(7);

```

```
    cout << "Searching for 5 in Pre-order traversal: ";
    findData(root, 1, 5);

    cout << "Searching for 6 in In-order traversal: ";
    findData(root, 2, 6);

    cout << "Searching for 8 in Post-order traversal: ";
    findData(root, 3, 8);

    delete root->left->left;
    delete root->left->right;
    delete root->right->left;
    delete root->right->right;
    delete root->left;
    delete root->right;
    delete root;

    return 0;
}
```

Output

```
Searching for 5 in Pre-order traversal: 5 was found!
Searching for 6 in In-order traversal: 6 was found!
Searching for 8 in Post-order traversal:

=== Code Execution Successful ===
```

3.4

Source Code

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

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

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

void preOrder(Node* node, char key, bool &found) {
    if (node == nullptr || found) {
        return;
    }
    if (node->data == key) {
        cout << key << " was found!" << endl;
        found = true;
        return;
    }
    preOrder(node->left, key, found);
    preOrder(node->right, key, found);
}

void inOrder(Node* node, char key, bool &found) {
    if (node == nullptr || found) {
        return;
    }
    inOrder(node->left, key, found);
    if (node->data == key) {
        cout << key << " was found!" << endl;
        found = true;
        return;
    }
    inOrder(node->right, key, found);
}

```

```cpp

void postOrder(Node* node, char key, bool &found) {
    if (node == nullptr || found) {
        return;
    }
    postOrder(node->left, key, found);
    postOrder(node->right, key, found);
    if (node->data == key) {
        cout << key << " was found!" << endl;
        found = true;
        return;
    }
}

void findData(Node* root, int choice, char key) {
    bool found = false;
    switch (choice) {
        case 1:
            preOrder(root, key, found);
            break;
        case 2:
            inOrder(root, key, found);
            break;
        case 3:
            postOrder(root, key, found);
            break;
        default:
            cout << "Invalid choice!" << endl;
    }
}

int main() {
    Node* root = new Node('1');
    root->left = new Node('2');
    root->right = new Node('3');
    root->left->left = new Node('4');
    root->left->right = new Node('5');
    root->right->left = new Node('6');
    root->right->right = new Node('7');

```

```
    root->right->left->right = new Node('O');

    cout << "Searching for 'O' in Pre-order traversal: ";
    findData(root, 1, 'O');

    cout << "Searching for 'O' in In-order traversal: ";
    findData(root, 2, 'O');

    cout << "Searching for 'O' in Post-order traversal: ";
    findData(root, 3, 'O');

    delete root->left->left;
    delete root->left->right;
    delete root->right->left->right;
    delete root->right->left;
    delete root->right->right;
    delete root->left;
    delete root->right;
    delete root;

    return 0;
}
```

Output

```
Searching for 'O' in Pre-order traversal: O was found!
Searching for 'O' in In-order traversal: O was found!
Searching for 'O' in Post-order traversal: O was found!


=== Code Execution Successful ===
```

Answers

The successful search for the new node "O" using pre-order, in-order, and post-order methods shows that the findData function works properly and that the tree structure is strong. Each method can find the same node, meaning they are set up correctly. This also shows that the tree is more efficient than a line of items. The findData function lets users pick their preferred method, making sure all nodes can be found no matter which order is used.

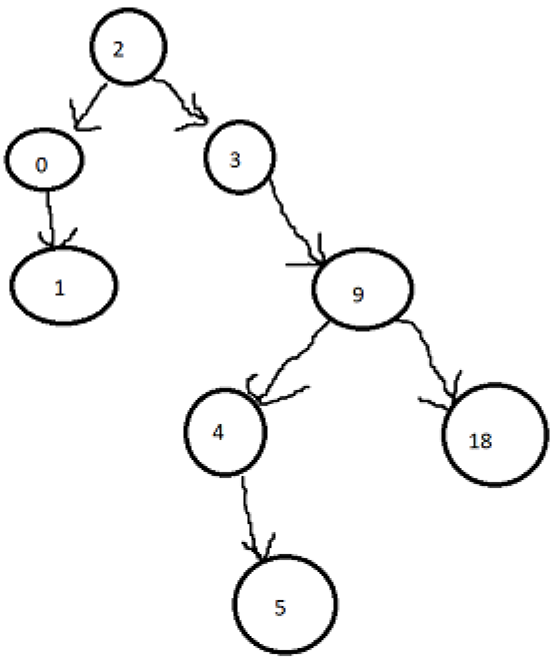
B. Answers to Supplementary Activity

Step1: code

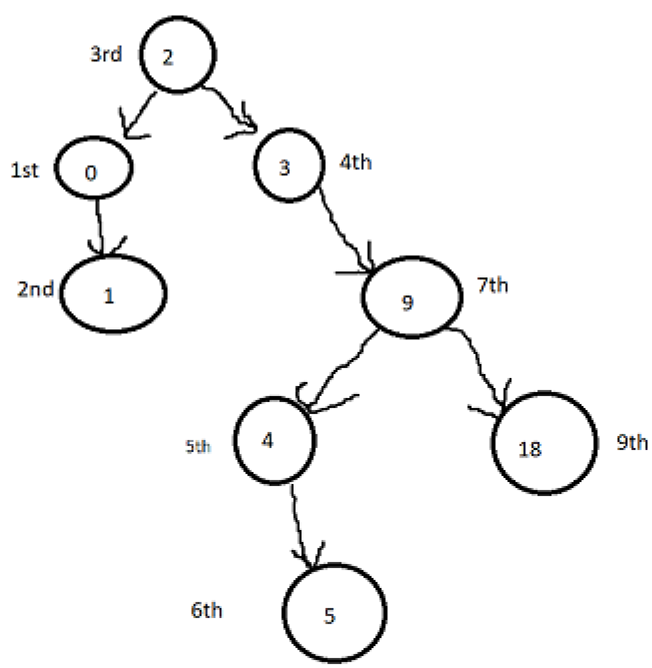
```cpp
class TreeNode {
public:
    int value;
    TreeNode* left;
    TreeNode* right;

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

class BinarySearchTree {
public:
    BinarySearchTree() : root(nullptr) {}

    void insert(int value) {
        root = insertNode(root, value);
    }

    void inOrderTraversal() {
        inOrder(root);
        cout << endl;
    }

private:
    TreeNode* root;

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

    void inOrder(TreeNode* node) {
        if (node == nullptr) {
            return;
        }
        inOrder(node->left);
        cout << node->value << " ";
        inOrder(node->right);
    }
};

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

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

    cout << "In-order traversal of the BST: ";
    bst.inOrderTraversal();
```

Step2:

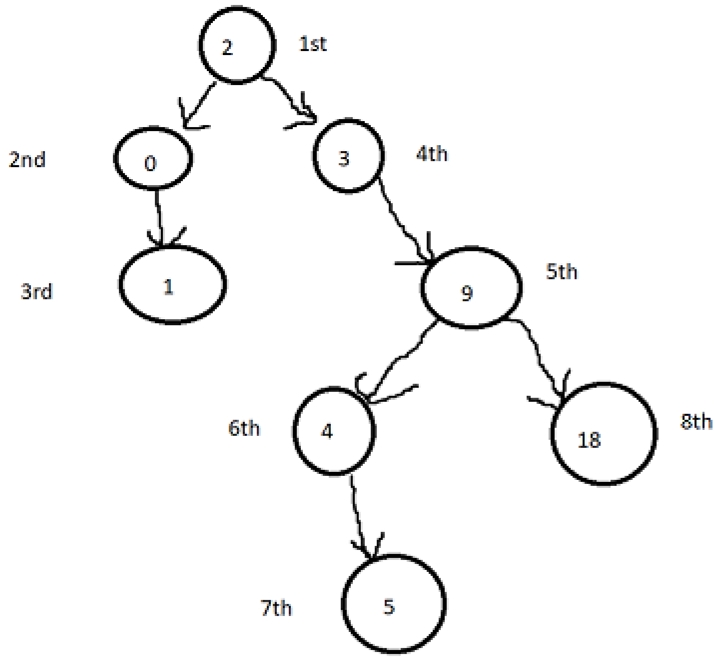
Tree



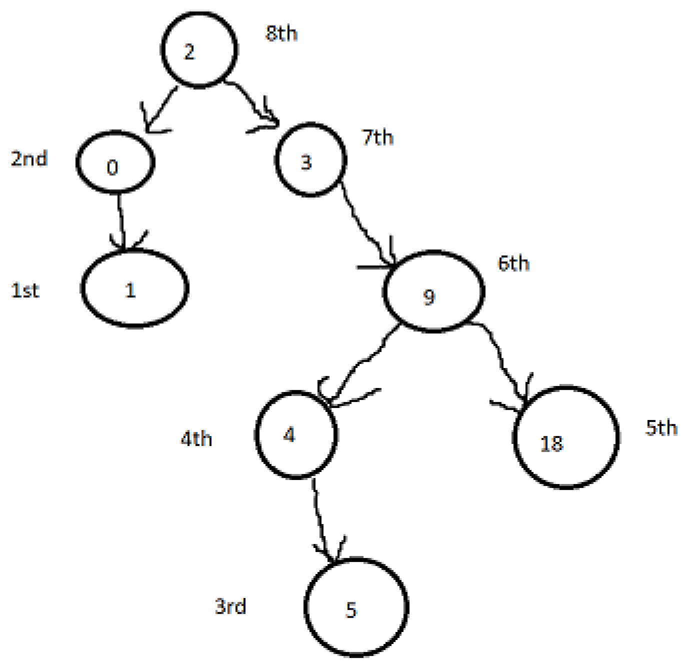
In Order Traversal



Pre-Order Traversal



Post-Order Traversal



Step3:

Pre Order Traversal
Code:

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

```
cout << "Pre-order traversal of the BST: ";
bst.preOrderTraversal();
```

Console Output:

```
Pre-order traversal of the BST: 2 0 1 3 9 4 5 18
```

Post Order Traversal
Code:

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

```
cout << "Post-order traversal of the BST: ";
bst.postOrderTraversal();
```

Console Output:

```
Post-order traversal of the BST: 1 0 5 4 18 9 3 2
```

The output of step 2 and 3 were the same.

C. Conclusion & Lessons Learned

After finishing the assigned task, we learned how to implement a binary search tree, basically the given values that are not in order will be in order due to the binary search tree method. This search method that we use arranges the given values within the array to the correct sequence in ascending order of each value. After implementing a binary search tree we needed to implement those values inside the array in-order, pre-order, post-order traversal while creating a tree diagram visualizing those values in a tree graph. To implement this type of order we needed to implement a traversal that caters to each order like the in-order, pre-order, and post-order traversal. What we learned about each of those orders is that in-order it first visits the left child, then the root or main node, lastly the right child. For the pre-order it starts with the root node, then it visits the left and right children. Lastly is the post order traversal, this traversal basically visits the left and right children first then the root node at last.
D. Assessment Rubric
E. External References