Activity No. 9	
TREES	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11/13/2024
Section: CPE21S1	**Date Submitted:** 11/15/2024
Name(s): Abarabar, John Nathan R. Aduana, Jake Norwin Bulambao, Adrian Justin M. Gaspar, Aaron Rowen Potestades, North Nygel G.	**Instructor:** Engr. Sayo

6. Output

Table 9.1

Code

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

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

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

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

void displayTree(TreeNode* node, int depth = 0) {
    for (int i = 0; i < depth; ++i) {
        cout << "  ";
    }
    cout << node->value << endl;

    for (auto child : node->children) {
        displayTree(child, depth + 1);
    }
}

int main() {
    TreeNode* root = new TreeNode('A');

    TreeNode* child1 = new TreeNode('B');
    TreeNode* child2 = new TreeNode('C');
    TreeNode* child3 = new TreeNode('D');
    TreeNode* child4 = new TreeNode('E');
    TreeNode* child5 = new TreeNode('F');
    TreeNode* child6 = new TreeNode('G');

    addChild(root, child1);
    addChild(root, child2);
    addChild(root, child3);
    addChild(root, child4);
    addChild(root, child5);
    addChild(root, child6);

    TreeNode* child3_1 = new TreeNode('H');
    TreeNode* child4_1 = new TreeNode('I');
    TreeNode* child4_2 = new TreeNode('J');
    TreeNode* child5_1 = new TreeNode('K');
    TreeNode* child5_2 = new TreeNode('L');
    TreeNode* child5_3 = new TreeNode('M');
    TreeNode* child6_1 = new TreeNode('N');

    addChild(child3, child3_1);
    addChild(child4, child4_1);
    addChild(child4, child4_2);
    addChild(child5, child5_1);
    addChild(child5, child5_2);
    addChild(child5, child5_3);
    addChild(child6, child6_1);

    TreeNode* child4_2_1 = new TreeNode('P');
    TreeNode* child4_2_2 = new TreeNode('Q');

    addChild(child4_2, child4_2_1);
    addChild(child4_2, child4_2_2);

    cout << "Tree structure:" << endl;
    displayTree(root);

    return 0;
}
```

Comments

In terms of length, our code could be shorter, but it gets the job done and creates a linked list-based tree that is structured like the graph given. We do this by manually creating each node and connecting them to the previous node, like in a tree.

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	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

Table 9.3

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

Table 9.4

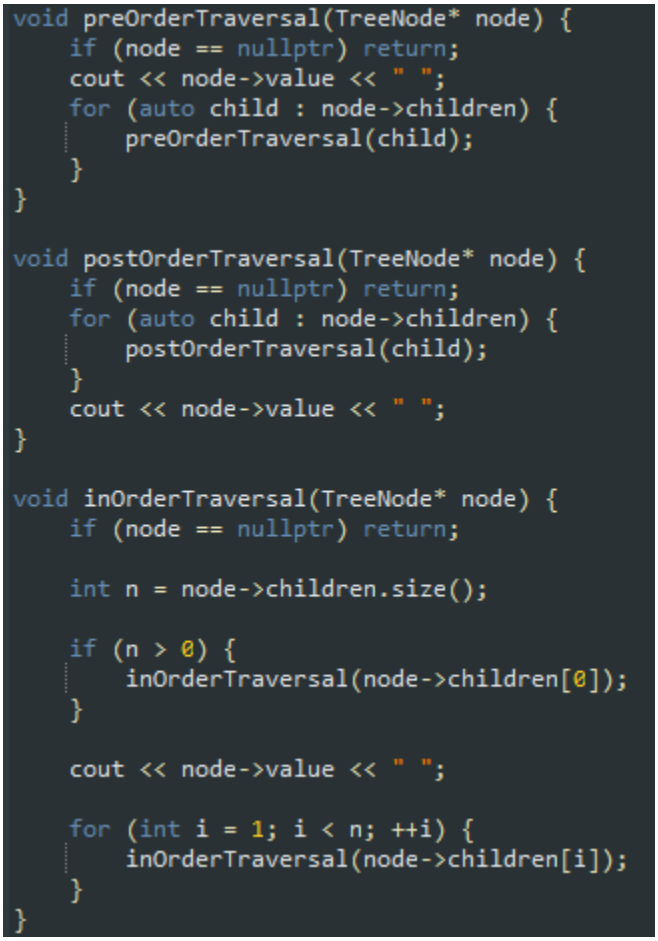
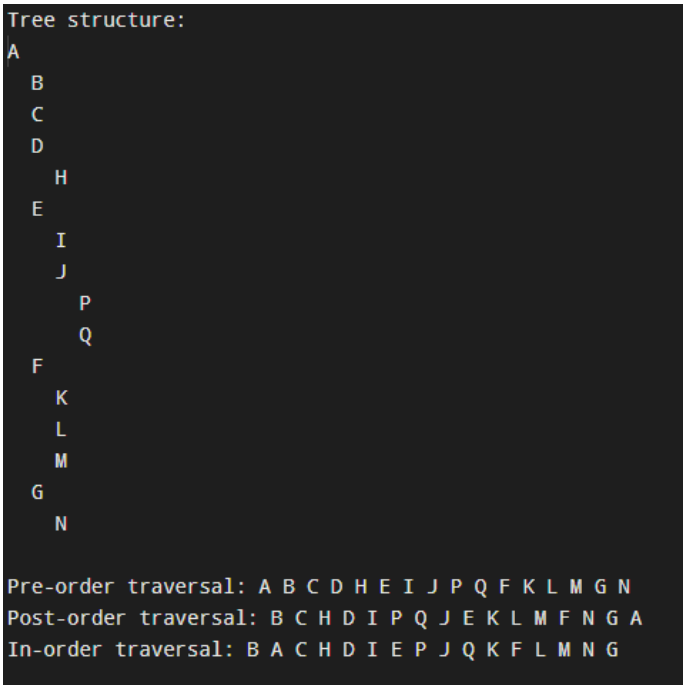
Screenshot	Observations
	We created 3 functions to traverse through the created tree, as specified by the lab manual, those being pre, post and in-order traversal. We did this through recursion in all of the traversal methods, though their implementations are different from each other to achieve the different results we need from them.
	We can see that the pre and post-order traversals were correct, the only different one being the in-order traversal.

Table 9.5

Code	Output
`` void findData(TreeNode* root, char choice, char key) { if (choice == 'P' || choice == 'p') { preOrderTraversal(root, key); } else if (choice == 'O' || choice == 'o') { postOrderTraversal(root, key); } else if (choice == 'I' || choice == 'i') { inOrderTraversal(root, key); } else { cout << "Invalid choice!" << endl; } } ``` + modification to traversals to return char once found.	``` Searching for 'P' with Pre-order traversal: P was found! Searching for 'Q' with Post-order traversal: Q was found! Searching for 'F' with In-order traversal: F was found! ```

Table 9.6

Output	Observations
`` Searching for 'O' with In-order traversal: O was found! ```	What we can observe from the output is that the code we created successfully added a new node in the position we wanted, and located it with in-order traversal, though the others would have worked as well.

7. Supplementary Activity

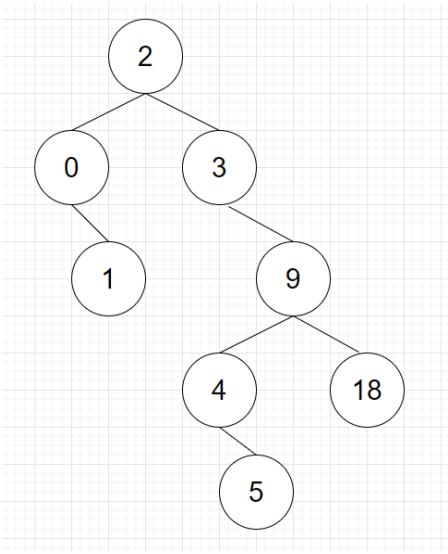
Step 1: Implement a binary search tree that will take the following values: 2, 3, 9, 18, 0, 1, 4, 5.

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

// Node structure for the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(NULL), right(NULL) {}
};

class BinarySearchTree {
public:
    // Function to insert a new node into the BST
    Node* insert(Node* root, int val) {
        if (root == NULL) {
            return new Node(val);  // Create a new node if tree/subtree is empty
        }

        if (val < root->data) {
            root->left = insert(root->left, val);  // Insert in the left subtree
        } else if (val > root->data) {
            root->right = insert(root->right, val);  // Insert in the right subtree
        }

        return root;  // Return the unchanged node pointer
    }

    // Inorder traversal of the BST (prints values in sorted order)
    void inorderTraversal(Node* root) {
        if (root == NULL) return;

        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
};

int main() {
    BinarySearchTree bst;
    Node* root = NULL;

    // Values to be inserted
    int values[] = {2, 3, 9, 18, 0, 1, 4, 5};
    int numValues = sizeof(values) / sizeof(values[0]);

    // Insert each value into the BST
    for (int i = 0; i < numValues; ++i) {
        root = bst.insert(root, values[i]);
    }

    // Display the BST using inorder traversal
    cout << "Inorder traversal of BST: ";
    bst.inorderTraversal(root);
    cout << endl;

    return 0;
}
```

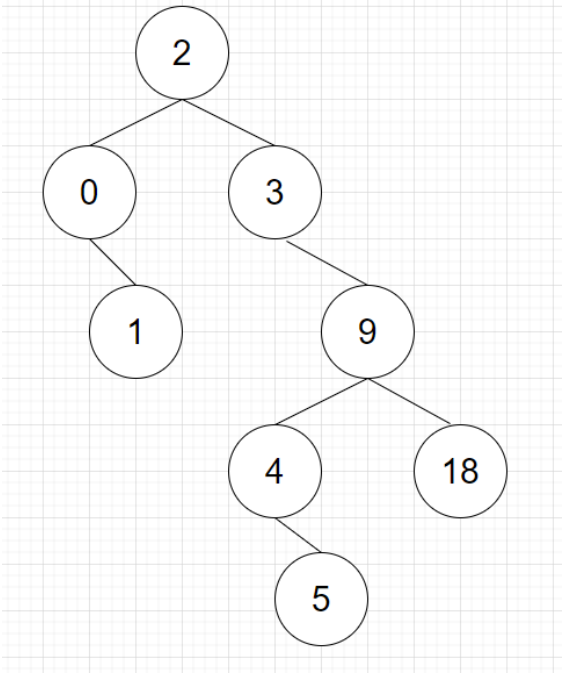
Step 2: Create a diagram to show the tree after all values have been inserted. Then, with the use of visual aids (like arrows and numbers) indicate the traversal order for in-order, pre-order and post-order traversal on the diagram.

(Screenshot of tree diagram)

```
Initial tree: 2-3-9-18-0-1-4-5-
```



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

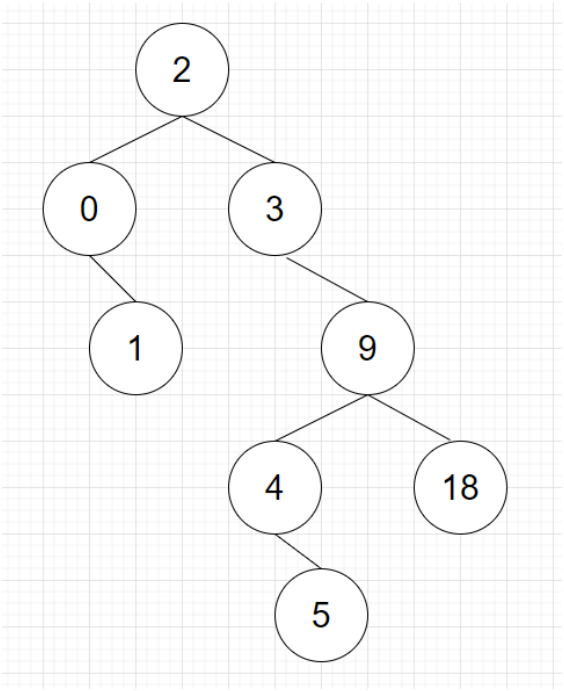



```
// In-order traversal function
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        cout << root->data << "-";
        inorder(root->right);
    }
}
```

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

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

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

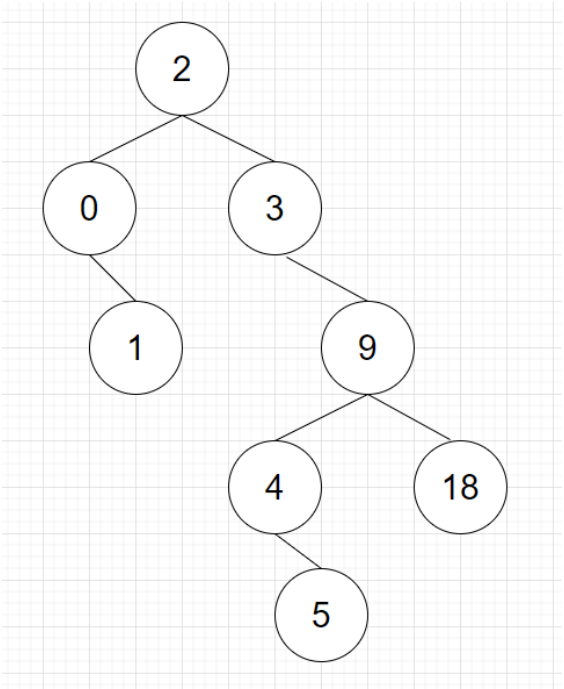



```
// Pre-order traversal function
void preorder(Node* root) {
    if (root != NULL) {
        cout << root->data << "-";
        preorder(root->left);
        preorder(root->right);
    }
}
```

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

Traversal Order: 2 - 0 - 1 - 3 - 9 - 4 - 5 - 18

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



```
// Post-order traversal function
void postorder(Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        cout << root->data << "-";
    }
}
```

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

Traversal Order: 1 - 0 - 5 - 4 - 18 - 9 - 3 - 2

Step 3: Compare the different traversal methods. In-order traversal was performed with what function? (Screenshot output needed)

```
// In-order traversal function
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        cout << root->data << "-";
        inorder(root->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)

```
// Pre-order traversal function
void preorder(Node* root) {
    if (root != NULL) {
        cout << root->data << "-";
        preorder(root->left);
        preorder(root->right);
    }
}
```

(Screenshot console output)

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

Observation: There is a built-in function of pre-order traversal in C++ and was able to make an order of the tree in a pre-order.

Post-order Traversal

(Screenshot created function)

```cpp
// Post-order traversal function
void postorder(Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        cout << root->data << "-";
    }
}
```

(Screenshot console output)

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

Observation: There is a built-in function of post-order traversal in C++ and was able to make an order of the tree in a post-order.

8. Conclusion

In conclusion, Trees are fundamental data structures, The Binary Search Tree (BST), is a type of tree that maintains order, ensuring that values to the left of a node are smaller and values to the right are larger. The activities demonstrate how trees have different types of order, in-order, pre-order, and post-order, allowing users to explore how data is accessed and processed in different sequences, each with its own use cases. The supplementary activity is not that hard, there are built in functions to be used which makes it easier, and we can say that we did well on it.

9. Assessment Rubric