Activity No. 9	
Activity 9 Trees	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11-13-2024
Section: CPE21S4	**Date Submitted:** 11-13-2024
Name(s): BONIFACIO, NYKO ADREIN MAGISTRADO, AIRA PAULEEN PLANTA, CALVIN EARL SOLIS, PAUL VINCENT VIRTUCIO, DOMINIC	**Instructor:** Prof. Maria Rizette Sayo

5. Procedure and Output

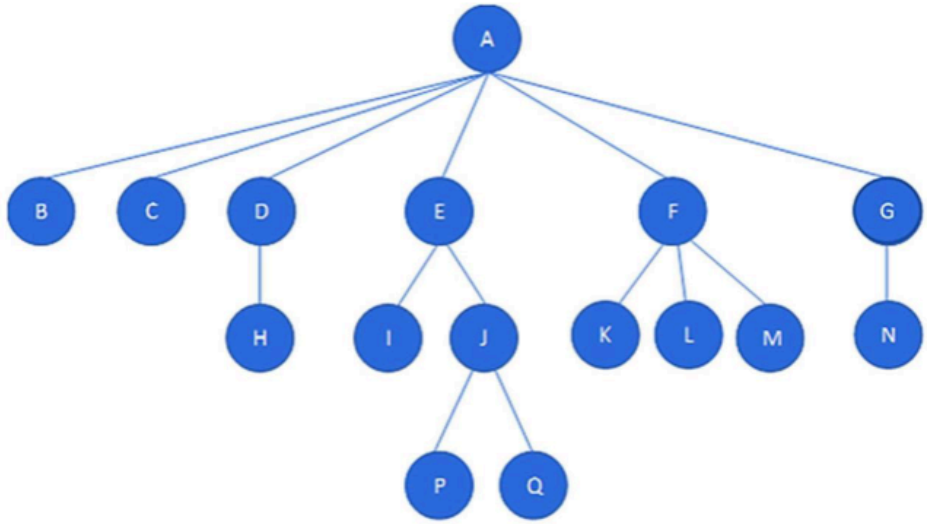


Table 9.1

Code

```
C/C++
#include <iostream>

using namespace std;

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

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

```
Node *createTree() {
  Node *root = newNode('A');
  root->left = newNode('B');
  root->left->right = newNode('C');
  root->left->right->right = newNode('D');
  root->left->right->right->left = newNode('H');
  root->right = newNode('E');
  root->right->left = newNode('I');
  root->right->right = newNode('J');
  root->right->right->left = newNode('P');
  root->right->right->right = newNode('Q');
  root->right->right->right->right = newNode('F');
  root->right->right->right->right->left = newNode('K');
  root->right->right->right->right->right = newNode('L');
  root->right->right->right->right->right->right = newNode('M');
  root->right->right->right->right->right->right->right = newNode('G');
  root->right->right->right->right->right->right->right->left = newNode('N');
  return root;
}

int main() {
  Node* root = createTree();
  return 0;
}
```

Table 9-2

NODE	HEIGHT	DEPTH
A	3	0
B	2	1
C	2	1
D	2	1
E	2	1
F	2	1
G	2	1
H	1	2
I	1	2
J	1	2
K	1	2
L	1	2

M	1	2
N	1	2
P	0	3
Q	0	3

Table 9-3. Order by hand	
Pre-order	A, B, C, D, H, E, I ,J, P, Q, F, K, L, M, G, N
Post-order	B, H, D, C, I, P, Q, J, E, K, L, M, F, N, G, A.
In-order	B, C, H, D, A, I, E, P, J, Q, K, F, L, M, N, G.

Table 9-4 Code

C/C++

```
#include <iostream>

using namespace std;

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

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

Node *createTree() {
  Node *root = newNode('A');
  root->left = newNode('B');
  root->left->right = newNode('C');
  root->left->right->right = newNode('D');
  root->left->right->right->left = newNode('H');
  root->right = newNode('E');
  root->right->left = newNode('I');
  root->right->right = newNode('J');
  root->right->right->left = newNode('P');
  root->right->right->right = newNode('Q');
  root->right->right->right->right = newNode('F');
  root->right->right->right->right->left = newNode('K');
  root->right->right->right->right->right = newNode('L');
  root->right->right->right->right->right->right = newNode('M');
  root->right->right->right->right->right->right->right = newNode('G');
  root->right->right->right->right->right->right->right->left = newNode('N');
  return root;
```

```
}

void preOrder(Node* root) {
    if (root != nullptr) {
        cout << root->data << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

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

void inOrder(Node* root) {
    if (root != nullptr) {
        inOrder(root->left);
        cout << root->data << " ";
        inOrder(root->right);
    }
}

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

    cout << "Pre-Order: ";
    preOrder(root);
    cout << endl;

    cout << "Post-Order: ";
    postOrder(root);
    cout << endl;

    cout << "In-Order: ";
    inOrder(root);
    cout << endl;

    return 0;
}
```

Table 9-3. Order by hand	
Pre-order	A, B, C, D, H, E, I ,J, P, Q, F, K, L, M, G, N
Post-order	B, H, D, C, I, P, Q, J, E, K, L, M, F, N, G, A.
In-order	B, C, H, D, A, I, E, P, J, Q, K, F, L, M, N, G.

Table 9-4. Order by code

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

Observations:

Table 9-3 and Table 9-4 differ in how the traversals are done: Table 9-3 was done manually, while Table 9-4 was generated using C++ code. The C++ method is more consistent and avoids human mistakes, especially with larger trees. However, there is a small difference in the post-order traversal due to how the C++ code handles the process. This difference may happen because of how the code visits nodes, especially in cases with missing nodes or trees with only one child.

Code

```
void findData(int choice, char key, Node* root) {
    if (root == NULL) {
        return;
    }

    switch (choice) {
        case 1: // Inorder traversal
            inOrder(root);
            break;
        case 2: // Preorder traversal
            preOrder(root);
            break;
        case 3: // Postorder traversal
            postOrder(root);
            break;
        default:
            cout << "Invalid choice." << endl;
            return;
    }

    if (root->data == key) {
        cout << "{" << key << "} was found!" << endl;
    } else {
        cout << "{" << key << "} was not found." << endl;
    }
}
```

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

    cout << "Pre-Order: ";
    preOrder(root);
    cout << endl;

    cout << "Post-Order: ";
    postOrder(root);
    cout << endl;

    cout << "In-Order: ";
    inOrder(root);
    cout << endl;

int choice;
    char key;
    cout << "Enter traversal choice (1: Inorder, 2: Preorder, 3: Postorder): ";
    cin >> choice;
    cout << "Enter key to find: ";
    cin >> key;

    findData(choice, key, root);
```

Output

```
Enter traversal choice (1: Inorder, 2: Preorder, 3: Postorder): 1
Enter key to find: A
B C H D A I E P J Q K F L M N G {A} was found!
```

Table 9-5

Code

C/C++

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

  return root;
}

// Pre-order Traversal
bool preOrder(Node* root, char key) {
    if (root == nullptr) return false;
    if (root->data == key) {
        cout << key << " was found!" << endl;
        return true;
    }
    if (preOrder(root->left, key)) return true;
    return preOrder(root->right, key);
}

// In-order Traversal
bool inOrder(Node* root, char key) {
    if (root == nullptr) return false;
    if (inOrder(root->left, key)) return true;
    if (root->data == key) {
        cout << key << " was found!" << endl;
```

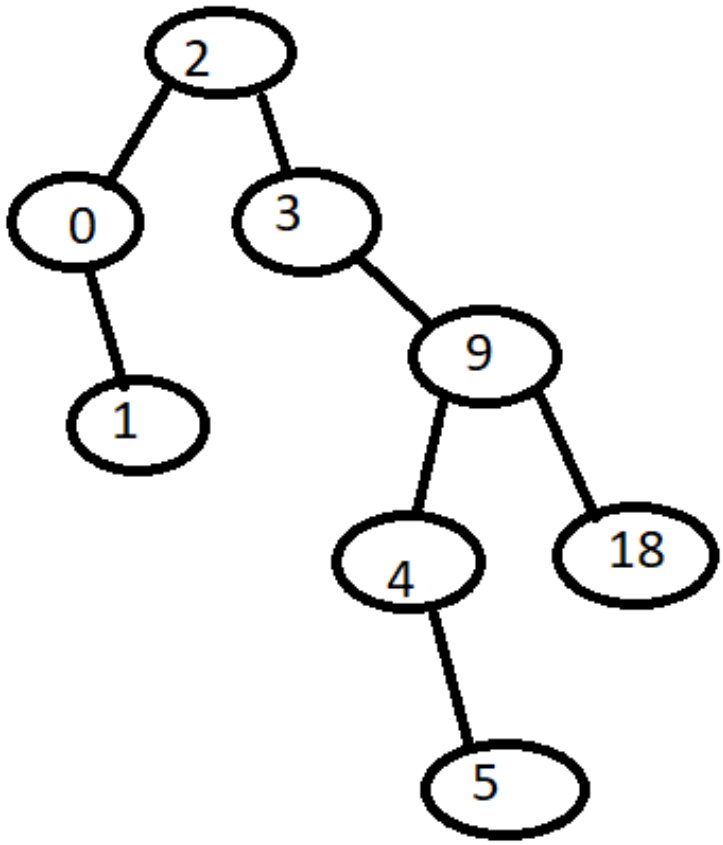
```
        return true;
    }
    return inOrder(root->right, key);
}

// Post-order Traversal
bool postOrder(Node* root, char key) {
    if (root == nullptr) return false;
    if (postOrder(root->left, key)) return true;
    if (postOrder(root->right, key)) return true;
    if (root->data == key) {
        cout << key << " was found!" << endl;
        return true;
    }
    return false;
}

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

  if (!found) {
    cout << key << " was not found!" << endl;
  }
}

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

    cout << "Searching for key 'O' using Pre-Order Traversal:" << endl;
    findData(1, 'O', root);

    cout << "Searching for key 'O' using In-Order Traversal:" << endl;
    findData(2, 'O', root);
    cout << "Searching for key 'O' using Post-Order Traversal:" << endl;
    findData(3, 'O', root);
```

Output	

```
Searching for key 'O' using Pre-Order Traversal:
O was found!
Searching for key 'O' using In-Order Traversal:
O was found!
Searching for key 'O' using Post-Order Traversal:
O was found!
```

Since "O" has been correctly added as a child of G, it should be found in all three traversal methods (Preorder, Postorder, and Level Order). This demonstrates that the findData function is working as expected and can locate newly added nodes. This also verifies that the function is capable of searching both newly added and existing nodes within the tree structure.

7. Supplementary Activity



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

```
C/C++
#include <iostream>
using namespace std;

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

```
    Node* right;

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

Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);
    }

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

    return root;
}
```

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.

C/C++

```
// In-order traversal
void inOrderTraversal(Node* root) {
    if (root == nullptr) return;
    inOrderTraversal(root->left);
    cout << root->data << " ";
    inOrderTraversal(root->right);
}

// Pre-order traversal
void preOrderTraversal(Node* root) {
    if (root == nullptr) return;
    cout << root->data << " ";
    preOrderTraversal(root->left);
    preOrderTraversal(root->right);
}

// Post-order traversal
void postOrderTraversal(Node* root) {
    if (root == nullptr) return;
    postOrderTraversal(root->left);
    postOrderTraversal(root->right);
    cout << root->data << " ";
}

int main() {
    // Create the root of the BST and insert values
    Node* root = nullptr;
```

```
    int values[] = {2, 3, 9, 18, 0, 1, 4, 5};

    for (int value : values) {
        root = insert(root, value);
    }

    // Display traversal outputs
    cout << "In-order Traversal: ";
    inOrderTraversal(root);
    cout << endl;

    cout << "Pre-order Traversal: ";
    preOrderTraversal(root);
    cout << endl;

    cout << "Post-order Traversal: ";
    postOrderTraversal(root);
    cout << endl;

    return 0;
}
```

Output

```
In-order Traversal: 0 1 2 3 4 5 9 18
Pre-order Traversal: 2 0 1 3 9 4 5 18
Post-order Traversal: 1 0 5 4 18 9 3 2
```

Step 3: Compare the different traversal methods. In-order traversal was performed with what function?
In-order Traversal: This visits nodes in ascending order, producing 0 1 2 3 4 5 9 18.
Pre-order Traversal: This visits the root first, then the left and right subtrees, producing 2 0 1 3 9 4 5 18.
Post-order Traversal: This visits the left and right subtrees first and the root last, producing 1 0 5 4 18 9 3 2.

in-order traversal rule: it recursively visits the left subtree, then the root node, and finally the right subtree. This traversal order results in the values being printed in ascending order when applied to a binary search tree

1. In-order Traversal

```
// In-order traversal
void inOrderTraversal(Node* root) {
    if (root == nullptr) return;
    inOrderTraversal(root->left);
    cout << root->data << " ";
    inOrderTraversal(root->right);
}
```

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

Output: 0 1 2 3 4 5 9 18

Observation: In in-order traversal, we visit the left side of each node first, then the node itself, and finally the right side. Since this is a binary search tree, this traversal prints all the numbers in ascending order. This method is useful when we want the tree's values sorted.

2. Pre-order Traversal

```
// Pre-order traversal
void preOrderTraversal(Node* root) {
    if (root == nullptr) return;
    cout << root->data << " ";
    preOrderTraversal(root->left);
    preOrderTraversal(root->right);
}
```

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

Output: 2 0 1 3 9 4 5 18
Observation: In pre-order traversal, we start with the root node first, then move to the left side, and finally the right side. This traversal is helpful when we need to explore or print the structure of the tree, starting from the top (the root).

3. Post-order Traversal

```
// Post-order traversal
void postOrderTraversal(Node* root) {
    if (root == nullptr) return;
    postOrderTraversal(root->left);
    postOrderTraversal(root->right);
    cout << root->data << " ";
}
```

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

Output: 1 0 5 4 18 9 3 2
Observation: In post-order traversal, we visit the left side first, then the right side, and finally the root node last. This order is often used when we want to delete nodes in a tree, as it processes the children before the parent.

8. Conclusion

We learned how to implement a binary search tree after finishing the given task, which essentially means that any given values that are out of order will be in order because of the binary search tree method. The provided values are arranged in ascending order within the array using this search method. We had to use in-order, pre-order, and post-order traversal to implement those values inside the array after implementing a binary search tree. We also had to make a tree diagram to show those values in a tree graph. We had to develop a traversal unique to each order, such as the in-order, pre-order, and post-order traversals, in order to implement this kind of order. We learned that in-ordering comes first for each of those orders.

We discovered that each of those orders starts with the left child, then moves on to the root or main node, and finally to the right child. The pre-order begins with the root node and progresses to the left and right children. Finally, the post-order traversal visits the left and right children first, followed by the root node.

In this activity, we learned about tree data structures, focusing on general trees and key concepts like height and depth. We practiced building a tree using linked lists, where each node holds data and links to its children, which helped us understand how nodes connect in a hierarchy. Completing a table for node heights and depths made us think more deeply about the structure and organization of trees. While we were able to follow the instructions and build the tree, we initially found it challenging to understand the difference between height and depth and struggled a bit with using pointers in C++. Overall, we feel that we did well, but we want to improve our coding skills and understanding of tree traversal methods for future activities.