Hands-on Activity 9.2	
Implementing and Traversing Binary Trees	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 11/27/24**
Section: CPE 21S4	**Date Submitted: 11/27/24**
Group Members: Virtucio, Dominic Joseph Bonifacio, Nyko Adrein Magistrado, Aira Pauleen Planta, Calvin Earl Solis, Paul Vincent	**Instructor:** Engr. Maria Rizette Sayo

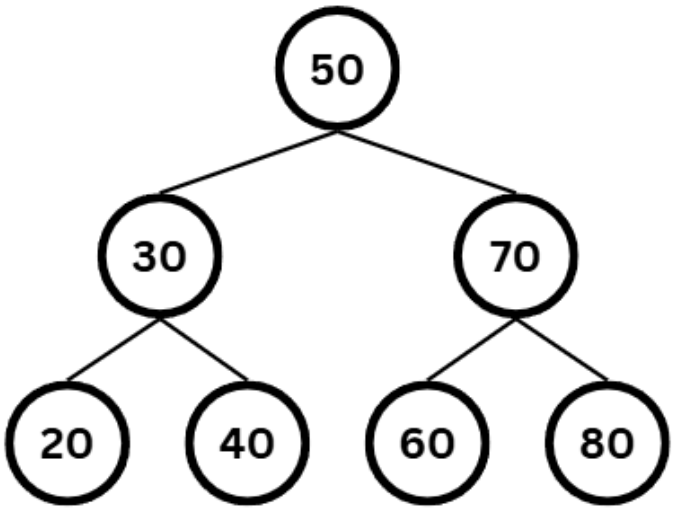
5. Procedure

C/C++

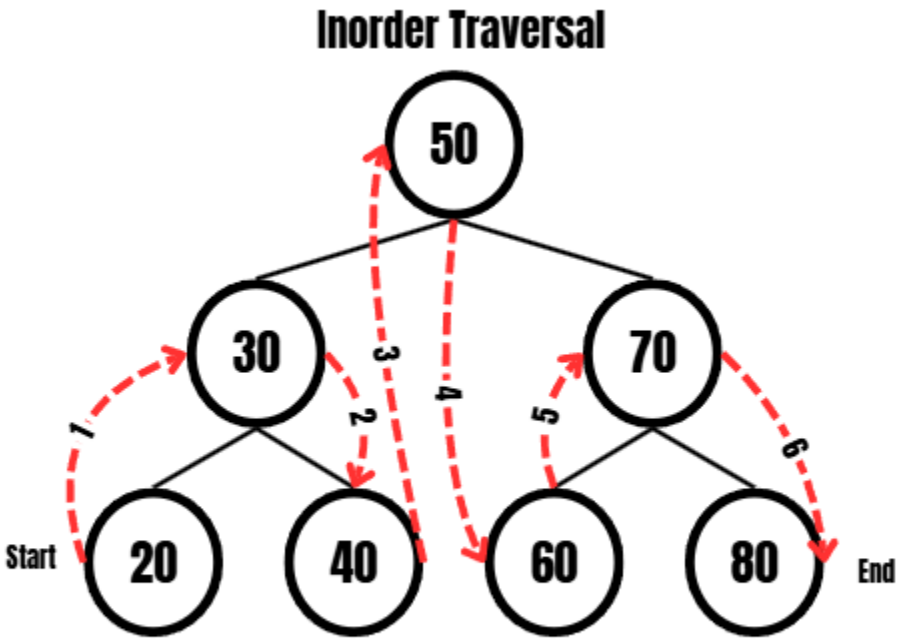
```
#include <iostream>
using namespace std;

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

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

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

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

    return root;
}

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

void preorder(Node* root) {
```

```
    if (root == nullptr) {
        return;
    }
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

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

int main() {
    Node* root = nullptr;

    // Insert values into the tree
    int values[] = {50, 30, 20, 40, 70, 60, 80};
    for (int val : values) {
        root = insert(root, val);
    }

    cout << "Inorder Traversal: ";
    inorder(root);
    cout << endl;

    cout << "Preorder Traversal: ";
    preorder(root);
    cout << endl;

    cout << "Postorder Traversal: ";
    postorder(root);
    cout << endl;

    return 0;
}
```

(Screenshot of tree diagram)



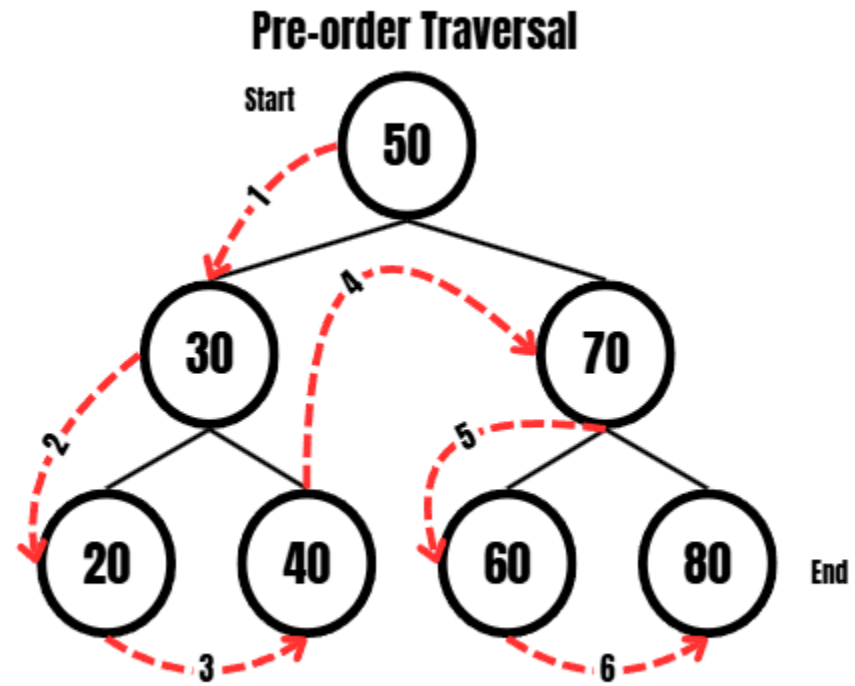
(Screenshot of tree diagram with indicated in-order traversal)



output:

```
Inorder Traversal: 20 30 40 50 60 70 80
```

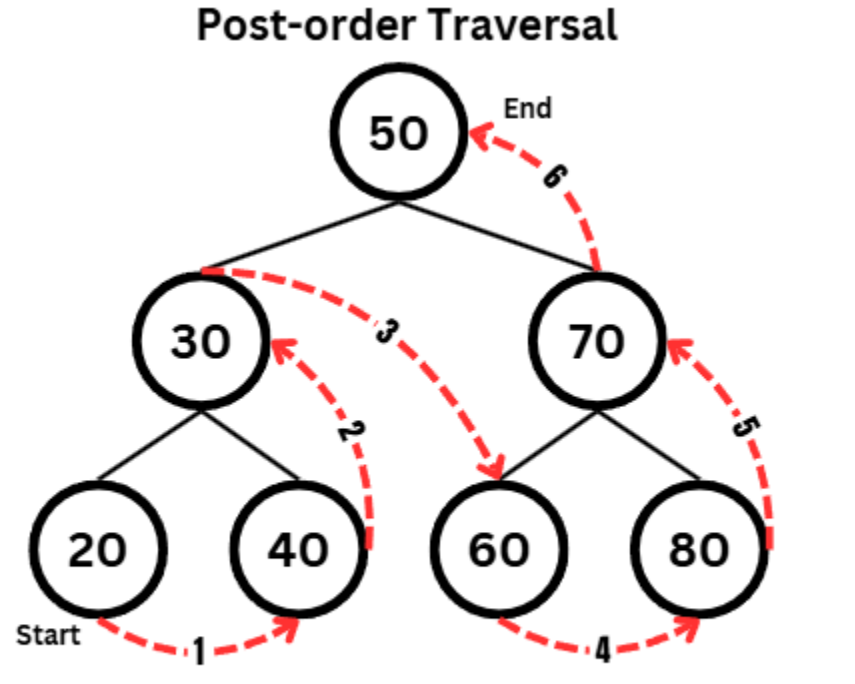
(Screenshot of tree diagram with indicated pre-order traversal)



output:

```
Preorder Traversal: 50 30 20 40 70 60 80
```

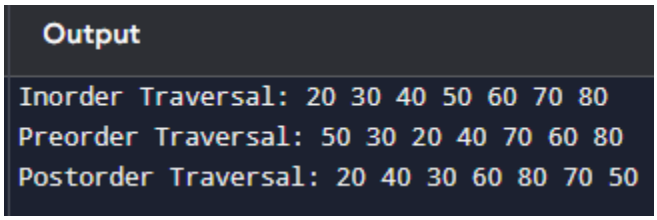
(Screenshot of tree diagram with indicated post-order traversal)



output:

```
Postorder Traversal: 20 40 30 60 80 70 50
```

Output



Conclusion

In conclusion, this interactive exercise on implementing and navigating binary trees has greatly deepened our understanding of fundamental data structures and algorithms, an important part of computer science and software development.

Throughout the exercise, we learned how to construct a binary tree using a structured Node format. Each Node consists of a data field and two pointers, representing the left and right children. This structure is essential for organizing data hierarchically, allowing for efficient searching, insertion, and deletion operations. Understanding how to define and manipulate these Nodes has provided us with a solid foundation in tree data structures.

The insertion function was really helpful as it explained how to systematically add values to the tree in a way that is still a BST. Here, we learned that with every value, the function compares it to the Node's data in the current Node, then determines whether to travel left (if smaller values) or right (for larger values). This recursive approach further simplifies the code while simultaneously showing why tree structures are indeed elegant solutions for ordered data handling.

Moreover, we made practical implementations of different traversal strategies on a tree: inorder, preorder, and postorder. While each one has its own application as well as a different viewpoint when it comes to viewing stored data, they each remain unique from one another. For example, the inorder traversal permits access to the values in ascending order; it is of great value to applications requiring sorted data. Preorder traversal has a lot of advantage in case a copy of the tree needs to be constructed or in evaluating a prefix expression, while postorder traversal is frequently utilized in the situations where parent Node needs to be processed after its children. Such scenarios arise while deleting.