THE UNIVERSITY OF AZAD JAMMU AND KASHMIR, MUZAFFARABAD

Department of Software Engineering

Submitted to:	Engr. Asma Javed
Course Title:	Data Structure and Algorithm
Course Code:	CS-2101
Session:	2024-2028
Lab No:	09
Roll No:	2024-SE-15
Submitted By:	Shahzad Ahmed Awan
Submission date:	January 9, 2026

Table of Contents

Lab 09: Implementation of Tree Data Structure in C++ 3
Objective 3
Tools Used 3
Theory (Minimal) 3
Implementation Details 3
1. Node Creation 3
3. Inorder Traversal 4
4. Preorder Traversal 4
5. Postorder Traversal 4
6. Search Operation 5
7. Counting Nodes 5
8. Height of Tree 5
9. Memory Deallocation 6
10. Main Function Stimulation: 6
Output of the Above Stimulation & Implementation: 7
Result 7
Conclusion 7

Table of Figure

Figure 1:Blueprint for Node Creation 3
Figure 2: Function for creating Node 3
Figure 3: Function for Inorder traversal 4
Figure 4: Function for Preorder Traversal 4
Figure 5: Function for Postorder Traversal 4
Figure 6: Function for Search Operation 5
Figure 7: Function for Counting Nodes 5
Figure 8: Function for Height Calculation 5
Figure 10: Function for Deleting a Tree 6
Figure 9: All Stimulation Code in the main Function 6
Figure 11: Output of the Above Stimulation & Implementation 7

Lab 09: Implementation of Tree Data Structure in C++

Objective

To implement a **Binary Tree** in C++ and perform basic tree operations.

Tools Used

- Dev C++

Theory (Minimal)

A Tree is a non-linear data structure used to store data in hierarchical form.
A Binary Tree allows at most two children for each node.

Implementation Details

1. Node Creation

Used a structure to store data and pointers to left and right child.

```
// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
};
```

Figure 1:Blueprint for Node Creation

2. Create Node Function

A function is used to dynamically create and initialize a new node

```
// Create a new node
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

Figure 2: Function for creating Node

3. Inorder Traversal

Traversal order: **Left → Root → Right**

```
// Inorder Traversal (Left -> Root -> Right)
void inorder(Node* root) {
    if (root == NULL)
        return;

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

Figure 3: Function for Inorder traversal

4. Preorder Traversal

Traversal order: **Root → Left → Right**

```
// Preorder Traversal (Root -> Left -> Right)
void preorder(Node* root) {
    if (root == NULL)
        return;

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

Figure 4: Function for Preorder Traversal

5. Postorder Traversal

Traversal order: **Left → Right → Root**

```
// Postorder Traversal (Left -> Right -> Root)
void postorder(Node* root) {
    if (root == NULL)
        return;

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

Figure 5: Function for Postorder Traversal

6. Search Operation

Used recursive approach to search an element in the tree.

```
// Function for  Searching an element in tree
bool search(Node* root, int key) {
    if (root == NULL)
        return false;

    if (root->data == key)
        return true;

    return search(root->left, key) || search(root->right, key);
}
```

Figure 6: Function for Search Operation

7. Counting Nodes

Function used to count total nodes in the tree.

```
// Count total nodes
int countNodes(Node* root) {
    if (root == NULL)
        return 0;

    return 1 + countNodes(root->left) + countNodes(root->right);
}
```

Figure 7: Function for Counting Nodes

8. Height of Tree

Height is calculated using recursive comparison of left and right subtrees.

```
// Find height of tree
int height(Node* root) {
    if (root == NULL)
        return 0;

    int leftHeight = height(root->left);
    int rightHeight = height(root->right);

    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
}
```

Figure 8: Function for Height Calculation

9. Memory Deallocation

All nodes are deleted using postorder traversal to avoid memory leakage.

```
// Delete entire tree (free memory)
void deleteTree(Node* root) {
    if (root == NULL)
        return;

    deleteTree(root->left);
    deleteTree(root->right);
    delete root;
}
```

Figure 9: Function for Deleting a Tree

10. Main Function Stimulation:

```
int main() {
    // Creating tree
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

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

    cout << "\nPreorder Traversal: ";
    preorder(root);

    cout << "\nPostorder Traversal: ";
    postorder(root);

    cout << "\n\nTotal Nodes: " << countNodes(root);
    cout << "\nLeaf Nodes: " << countLeafNodes(root);
    cout << "\nHeight of Tree: " << height(root);

    int key = 5;
    if (search(root, key))
        cout << "\nElement " << key << " found in tree.";
    else
        cout << "\nElement " << key << " not found in tree.";

    // Freeing memory
    deleteTree(root);

    return 0;
}
```

Figure 10: All Stimulation Code in the main Function

Output of the Above Stimulation & Implementation:

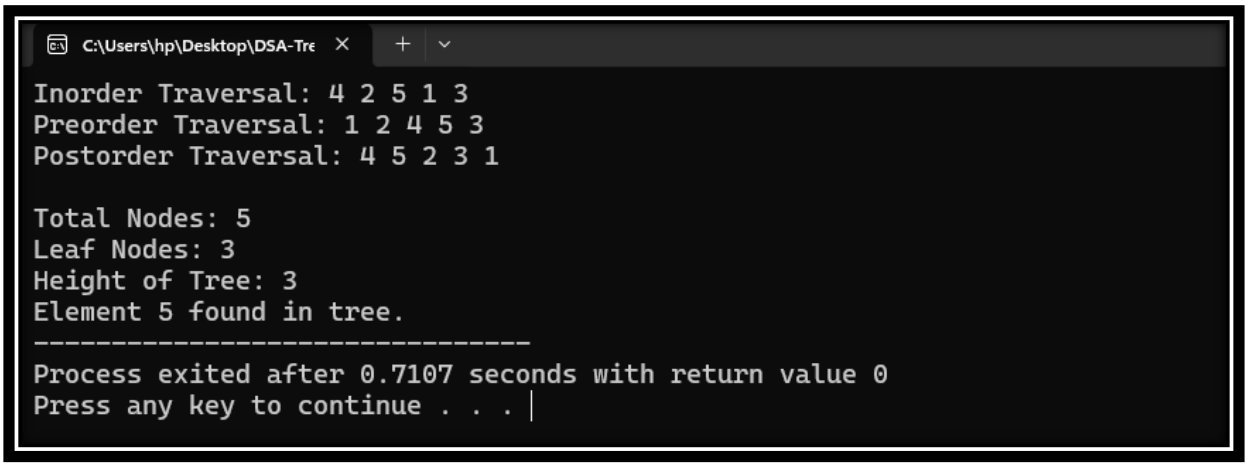

Figure 11: Output of the Above Stimulation & Implementation

Result

The Binary Tree was successfully implemented and all operations worked correctly.

Conclusion

Tree operations including traversal, searching, counting, and height calculation were implemented successfully using C++.