Project Report on Binary Search Tree (BST) Implementation using C++

Table of Contents

1. Introduction

Binary Search Trees (BST) are a type of data structure that maintain elements in a sorted order, allowing for efficient insertion, deletion, and search operations. This project aims to implement a BST in C++ with various operations including insertion, searching, deletion, and different tree traversals (in-order, pre-order, and post-order).

2. Objectives

The primary objectives of this project are:

- To understand and implement the BST data structure in C++.
- To perform and test various operations such as insertion, search, and deletion.
- To implement and demonstrate inorder, preorder, and postorder traversals.
- To analyze the performance and correctness of the BST operations.

3. Binary Search Tree (BST) Overview

A Binary Search Tree is a tree data structure where each node has at most two children referred to as the left child and the right child. For each node:

- The left subtree contains only nodes with keys less than the node’s key.
- The right subtree contains only nodes with keys greater than the node’s key.

4. Methodology

Insertion Operation

The insertion operation adds a new node to the BST. Starting from the root, the new node is compared to the current node and placed in the left subtree if it is smaller or in the right subtree if it is larger. This process is repeated recursively until the appropriate position is found.

Search Operation

The search operation checks if a given key is present in the BST. Similar to the insertion process, the search begins at the root and proceeds left or right based on comparisons until the key is found or a leaf node is reached.

Deletion Operation

The deletion operation removes a node from the BST. There are three cases to consider:

- The node is a leaf: Simply remove the node.
- The node has one child: Remove the node and replace it with its child.
- The node has two children: Find the node's in-order successor (the smallest node in the right subtree), replace the node's key with the successor's key, and delete the successor.

Inorder Traversal

Inorder traversal visits nodes in a left-root-right order. It visits all nodes in ascending order of their keys.

Preorder Traversal

Preorder traversal visits nodes in a root-left-right order. It processes the root before its subtrees.

Postorder Traversal

Postorder traversal visits nodes in a left-right-root order. It processes the root after its subtrees.

4. Implementation

The implementation involves defining the structure of a BST node and the BST class that contains methods for various operations.

Code:

```
 #include <iostream>

using namespace std;

struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int k) : key(k), left(nullptr), right(nullptr) {}
};

class BinarySearchTree {
public:
    TreeNode* root;

    BinarySearchTree() : root(nullptr) {}

    void insert(int key) {
        root = insertRec(root, key);
    }

    void deleteNode(int key) {
        root = deleteRec(root, key);
    }

    TreeNode* search(int key) {
        return searchRec(root, key);
    }

    void inorder() {
        inorderRec(root);
        cout << endl;
    }

    void preorder() {
        preorderRec(root);
        cout << endl;
    }

    void postorder() {
```

```
        postorderRec(root);
        cout << endl;
    }

private:
    TreeNode* insertRec(TreeNode* node, int key) {
        if (node == nullptr) {
            return new TreeNode(key);
        }

        if (key < node->key) {
            node->left = insertRec(node->left, key);
        } else if (key > node->key) {
            node->right = insertRec(node->right, key);
        }

        return node;
    }

    TreeNode* deleteRec(TreeNode* root, int key) {
        if (root == nullptr) {
            return root;
        }

        if (key < root->key) {
            root->left = deleteRec(root->left, key);
        } else if (key > root->key) {
            root->right = deleteRec(root->right, key);
        } else {
            if (root->left == nullptr) {
                TreeNode* temp = root->right;
                delete root;
                return temp;
            } else if (root->right == nullptr) {
                TreeNode* temp = root->left;
                delete root;
                return temp;
            }

            TreeNode* temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteRec(root->right, temp->key);
        }
        return root;
    }

    TreeNode* minValueNode(TreeNode* node) {
        TreeNode* current = node;
```

```
        while (current && current->left != nullptr) {
            current = current->left;
        }
        return current;
    }

    TreeNode* searchRec(TreeNode* node, int key) {
        if (node == nullptr || node->key == key) {
            return node;
        }

        if (key < node->key) {
            return searchRec(node->left, key);
        }

        return searchRec(node->right, key);
    }

    void inorderRec(TreeNode* root) {
        if (root != nullptr) {
            inorderRec(root->left);
            cout << root->key << " ";
            inorderRec(root->right);
        }
    }

    void preorderRec(TreeNode* root) {
        if (root != nullptr) {
            cout << root->key << " ";
            preorderRec(root->left);
            preorderRec(root->right);
        }
    }

    void postorderRec(TreeNode* root) {
        if (root != nullptr) {
            postorderRec(root->left);
            postorderRec(root->right);
            cout << root->key << " ";
        }
    }
};

int main() {

    cout<<"Welcome to my BST program!!!"<<endl;

    BinarySearchTree bst;
```

```
    int choice, key;

    do {
        cout << "\nMenu:\n";
        cout << "1. Insert\n";
        cout << "2. Search\n";
        cout << "3. In-order Traversal\n";
        cout << "4. Pre-order Traversal\n";
        cout << "5. Post-order Traversal\n";
        cout << "6. Delete\n";
        cout << "7. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                int n;
                cout<<"How many nodes do you want to insert ?: ";
                cin>>n;
                for(int i=0;i<n;i++){
                    int j;
                    cout<<"Enter the node which you want to insert: ";
                    cin>>j;
                    bst.insert(j);
                }
                break;

            case 2:
                cout << "Enter key you want to search: ";
                cin >> key;
                if (bst.search(key) != nullptr) {
                    cout << "Found node with key: " << key << endl;
                } else {
                    cout << "Node with key " << key << " not found." << endl;
                }
                break;

            case 3:
                cout << "In-order traversal: ";
                bst.inorder();
                break;

            case 4:
                cout << "Pre-order traversal: ";
                bst.preorder();
                break;

            case 5:
```

```
                cout << "Post-order traversal: ";
                bst.postorder();
                break;

            case 6:
                cout << "Enter the node which you want to delete: ";
                cin >> key;
                bst.deleteNode(key);
                break;

            case 7:
                cout << "Exiting..." << endl;
                break;

            default:
                cout << "Invalid choice. Please try again." << endl;
        }
    } while (choice != 7);

    return 0;
}
```

6. Testing and Results

Testing involved inserting a series of numbers into the BST and verifying that they were correctly ordered using inorder traversal. The search function was tested with existing and non-existing keys, and deletion was tested to ensure nodes were properly removed and the tree structure remained correct.

7. Conclusion

The implemented Binary Search Tree (BST) in C++ successfully performs insertion, search, and deletion operations. The tree traversals (inorder, preorder, and postorder) correctly display the

elements in the desired order. This project demonstrates the effectiveness and efficiency of BSTs in managing sorted data.

8. Future Work

Future improvements could include:

- Balancing the BST to ensure O(log n) time complexity for operations.
- Implementing AVL or Red-Black Trees for self-balancing properties.
- Adding more functionality such as finding the height of the tree, counting the number of nodes, and displaying the tree visually.

9. References

- "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
- GeeksforGeeks articles on Binary Search Trees.
- Cplusplus.com for C++ syntax and library references.