Lab 09: Binary Search Tree - Solutions

Student Name: Muhammad Afzal Iqbal
Roll Number: CT-24263
Date: November 13, 2025

Exercise 1: Preorder and Postorder Traversals Using Stack

Code (lab9_ex1.cpp)
```cpp
#include <iostream>
#include <stack>  // For stack
using namespace std;

class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : val(value), left(NULL), right(NULL) {}
};

class BinarySearchTree {
public:
    TreeNode* root;
    BinarySearchTree() : root(NULL) {}

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

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

    // Preorder traversal using stack
    void preorder() {
        preorderIter(root);
        cout << endl;
    }

    // Postorder traversal using stack
    void postorder() {
        postorderIter(root);
        cout << endl;
    }

private:
    TreeNode* insertRec(TreeNode* node, int key) {
        if (node == NULL)
            return new TreeNode(key);
        if (key < node->val)
            node->left = insertRec(node->left, key);
        else if (key > node->val)
            node->right = insertRec(node->right, key);
        return node;
    }

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

```
    }

    void preorderIter(TreeNode* root) {
        if (root == NULL) return;
        stack<TreeNode*> s;
        s.push(root);
        while (!s.empty()) {
            TreeNode* node = s.top();
            s.pop();
            cout << node->val << " ";
            if (node->right) s.push(node->right);
            if (node->left) s.push(node->left);
        }
    }

    void postorderIter(TreeNode* root) {
        if (root == NULL) return;
        stack<pair<TreeNode*, bool>> s;
        s.push(make_pair(root, false));
        while (!s.empty()) {
            pair<TreeNode*, bool> top = s.top();
            s.pop();
            if (top.second) {
                cout << top.first->val << " ";
            } else {
                s.push(make_pair(top.first, true));
                if (top.first->right) s.push(make_pair(top.first->right, false));
                if (top.first->left) s.push(make_pair(top.first->left, false));
            }
        }
    }
};

int main() {
    BinarySearchTree bst;
    bst.insert(50);
    bst.insert(30);
    bst.insert(20);
    bst.insert(40);
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);

    cout << "Inorder: ";
    bst.inorder();

    cout << "Preorder: ";
    bst.preorder();

    cout << "Postorder: ";
    bst.postorder();

    return 0;
}
```

Output

```
Inorder: 20 30 40 50 60 70 80
Preorder: 50 30 20 40 70 60 80
Postorder: 20 40 30 60 80 70 50
```

Exercise 2: Employee Management System Using BST

Code (lab9_ex2.cpp)

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

```
class EmployeeNode {
public:
    int id;
    string name;
    string department;
    EmployeeNode* left;
    EmployeeNode* right;
    EmployeeNode(int i, string n, string d) : id(i), name(n), department(d), left(NULL), right(NULL) {}
};

class EmployeeBST {
public:
    EmployeeNode* root;
    EmployeeBST() : root(NULL) {}

    void insert(int id, string name, string department) {
        if (isIDPresent(root, id)) {
            cout << "Duplicate ID " << id << " skipped." << endl;
            return;
        }
        root = insertRec(root, id, name, department);
    }

    bool search(int id) {
        EmployeeNode* found = searchRec(root, id);
        if (found) {
            cout << "Found: ID=" << found->id << ", Name=" << found->name << ", Dept=" << found-
>department << endl;
            return true;
        } else {
            cout << "Employee with ID " << id << " not found." << endl;
            return false;
        }
    }

    void deleteNode(int id) {
        if (!isIDPresent(root, id)) {
            cout << "Employee with ID " << id << " not found. Deletion skipped." << endl;
            return;
        }
        root = deleteRec(root, id);
        cout << "Deleted employee with ID " << id << "." << endl;
    }

    void inOrderTraversal() {
        cout << "Employees (sorted by ID):" << endl;
        inOrderRec(root);
        cout << endl;
    }

    void findMin() {
        EmployeeNode* minNode = minValueNode(root);
        if (minNode) {
            cout << "Min ID Employee: ID=" << minNode->id << ", Name=" << minNode->name << ", Dept=" <<
minNode->department << endl;
        } else {
            cout << "Tree is empty." << endl;
        }
    }

    void findMax() {
        EmployeeNode* maxNode = maxValueNode(root);
        if (maxNode) {
            cout << "Max ID Employee: ID=" << maxNode->id << ", Name=" << maxNode->name << ", Dept=" <<
maxNode->department << endl;
        } else {
            cout << "Tree is empty." << endl;
        }
```

```
    }

private:
    bool isIDPresent(EmployeeNode* node, int id) {
        if (node == NULL) return false;
        if (id == node->id) return true;
        if (id < node->id) return isIDPresent(node->left, id);
        return isIDPresent(node->right, id);
    }

    EmployeeNode* insertRec(EmployeeNode* node, int id, string name, string department) {
        if (node == NULL) {
            return new EmployeeNode(id, name, department);
        }
        if (id < node->id) {
            node->left = insertRec(node->left, id, name, department);
        } else if (id > node->id) {
            node->right = insertRec(node->right, id, name, department);
        }
        return node;
    }

    EmployeeNode* searchRec(EmployeeNode* node, int id) {
        if (node == NULL || node->id == id) return node;
        if (id < node->id) return searchRec(node->left, id);
        return searchRec(node->right, id);
    }

    EmployeeNode* deleteRec(EmployeeNode* node, int id) {
        if (node == NULL) return node;
        if (id < node->id) {
            node->left = deleteRec(node->left, id);
        } else if (id > node->id) {
            node->right = deleteRec(node->right, id);
        } else {
            if (node->left == NULL) {
                EmployeeNode* temp = node->right;
                delete node;
                return temp;
            } else if (node->right == NULL) {
                EmployeeNode* temp = node->left;
                delete node;
                return temp;
            }
            EmployeeNode* temp = minValueNode(node->right);
            node->id = temp->id;
            node->name = temp->name;
            node->department = temp->department;
            node->right = deleteRec(node->right, temp->id);
        }
        return node;
    }

    EmployeeNode* minValueNode(EmployeeNode* node) {
        while (node && node->left != NULL) node = node->left;
        return node;
    }

    EmployeeNode* maxValueNode(EmployeeNode* node) {
        while (node && node->right != NULL) node = node->right;
        return node;
    }

    void inOrderRec(EmployeeNode* node) {
        if (node != NULL) {
            inOrderRec(node->left);
            cout << "ID=" << node->id << ", Name=" << node->name << ", Dept=" << node->department << endl;
            inOrderRec(node->right);
        }
```

```
    }
};

int main() {
    EmployeeBST empBST;
    empBST.insert(101, "Alice", "HR");
    empBST.insert(102, "Bob", "IT");
    empBST.insert(100, "Charlie", "Finance");
    empBST.insert(103, "David", "HR");

    empBST.insert(101, "Duplicate", "Test");

    empBST.inOrderTraversal();

    empBST.search(102);
    empBST.search(999);

    empBST.findMin();
    empBST.findMax();

    cout << "\n--- Deleting Employee 102 ---\n";
    empBST.deleteNode(102);

    cout << "\n--- After Deletion ---\n";
    empBST.inOrderTraversal();

    return 0;
}
```

Output

```
Duplicate ID 101 skipped.
Employees (sorted by ID):
ID=100, Name=Charlie, Dept=Finance
ID=101, Name=Alice, Dept=HR
ID=102, Name=Bob, Dept=IT
ID=103, Name=David, Dept=HR

Found: ID=102, Name=Bob, Dept=IT
Employee with ID 999 not found.
Min ID Employee: ID=100, Name=Charlie, Dept=Finance
Max ID Employee: ID=103, Name=David, Dept=HR

--- Deleting Employee 102 ---
Deleted employee with ID 102.

--- After Deletion ---
Employees (sorted by ID):
ID=100, Name=Charlie, Dept=Finance
ID=101, Name=Alice, Dept=HR
ID=103, Name=David, Dept=HR
```

Exercise 3: Least Common Ancestor of Two Nodes in BST

Code (lab9_ex3.cpp)

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

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (root == NULL || p == NULL || q == NULL) return NULL;

    // If both p and q are smaller than root, LCA must be in the left subtree
    if (p->val < root->val && q->val < root->val) {
        return lowestCommonAncestor(root->left, p, q);
    }

    // If both p and q are greater than root, LCA must be in the right subtree
    if (p->val > root->val && q->val > root->val) {
        return lowestCommonAncestor(root->right, p, q);
    }

    // Otherwise, the current root is the LCA (split point or one node is the root itself).
    return root;
}

int main() {
    TreeNode* root = new TreeNode(6);
    root->left = new TreeNode(2);
    root->right = new TreeNode(8);
    root->left->left = new TreeNode(0);
    root->left->right = new TreeNode(4);
    root->left->right->left = new TreeNode(3);
    root->left->right->right = new TreeNode(5);
    root->right->left = new TreeNode(7);
    root->right->right = new TreeNode(9);


    TreeNode* p_ex1 = root->left->left;  // 0
    TreeNode* q_ex1 = root->left->right->right;  // 5
    TreeNode* lca_ex1 = lowestCommonAncestor(root, p_ex1, q_ex1);
    cout << "LCA of " << p_ex1->val << " and " << q_ex1->val << " is " << lca_ex1->val << endl;

    TreeNode* p_ex2 = root->left->left;  // 0
    TreeNode* q_ex2 = root->right->right;  // 9
    TreeNode* lca_ex2 = lowestCommonAncestor(root, p_ex2, q_ex2);
    cout << "LCA of " << p_ex2->val << " and " << q_ex2->val << " is " << lca_ex2->val << endl;

    return 0;
}
```

Output

```
LCA of 0 and 5 is 2
LCA of 0 and 9 is 6
```

Exercise 4: Sum of All Nodes in BST

Code (lab9_ex4.cpp)

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

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

int sumOfBST(TreeNode* root) {
    if (root == NULL) return 0;
    return root->val + sumOfBST(root->left) + sumOfBST(root->right);
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);

    cout << "Sum of BST: " << sumOfBST(root) << endl;

    return 0;
}
```

Output

```
Sum of BST: 10
```

Exercise 5: Minimum Difference Between Any Two Nodes in BST

Code (lab9_ex5.cpp)

```cpp
#include <iostream>
#include <climits>
#include <stack>
#include <algorithm> // For min function
using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

int minDiffInBST(TreeNode* root) {
    if (root == NULL) return INT_MAX;

    stack<TreeNode*> s;
    TreeNode* curr = root;
    TreeNode* prev = NULL;
    int minDiff = INT_MAX;

    while (curr != NULL || !s.empty()) {
        // Traverse left
        while (curr != NULL) {
            s.push(curr);
            curr = curr->left;
        }

        // Visit the node
        curr = s.top();
        s.pop();

        // Calculate difference with the previous node (Inorder check)
        if (prev != NULL) {
            minDiff = min(minDiff, curr->val - prev->val);
        }

        // Move to the right subtree
        prev = curr;
        curr = curr->right;
    }
    return minDiff;
}

int main() {
    TreeNode* root = new TreeNode(4);
    root->left = new TreeNode(2);
    root->right = new TreeNode(6);
    root->left->left = new TreeNode(1);
    root->left->right = new TreeNode(3);

    cout << "Min difference: " << minDiffInBST(root) << endl;

    return 0;
}
```

Output

```
Min difference: 1
```