

Lab 3 : Binary Search Trees

Course: CS20009.04 Data Structure

Name: Yijia Chen

Student Number: 24300240127

Date: October 21, 2025

Introduction

This lab aims to implement the following operations on binary search trees:

- Searching a node with a given key.
- Finding the successor and predecessor of a given node.
- Inorder, preorder, and postorder tree walk.

Implementation

Search

The `searchRecursive` or `searchIterative` procedure follows a simple path from the root `x` downwards. The BST property is used to determine the next step, to go left if `key < x` or go right otherwise. The function terminates when either `x == key` or the path reaches a leaf node.

Successor and Predecessor

`successor` returns the node with the smallest key greater than `x`'s key if it exists. There are two cases:

- If the right subtree of `x` is not empty, the successor `y` is the smallest element, which is also the leftmost node, in the right subtree.
- If the right subtree of `x` is empty, the successor `y` is "the lowest ancestor of `x` whose left child is also an ancestor of `x`".

`predecessor` returns the node with the largest key less than `x`'s key if it exists. It is symmetric to `successor`.

Tree Walk

`inorderTreeWalk` prints out all the keys in a BST in inorder, because of the BST property that, the left subtree of a node contains all keys less than its key, and the right subtree contains all keys greater than its key.

The implementation of `inorderTreeWalk`, `preorderTreeWalk` and `postorderTreeWalk` are similar. The only change is the order of visiting the current node and its two subtrees.

```

#pragma once
#include <iostream>

template <class T>
struct Node {
    T key;
    Node<T> *parent, *left, *right;
};

template <class T>
struct BST {
    Node<T> * root;

    // (1)

    T searchRecursive(Node<T>* x, T k) {           // recursion version
        if (x == nullptr) return -1;
        else if (x->key == k) return x->key;
        else if (x->key > k) return searchRecursive(x->left, k);
        else return searchRecursive(x->right, k);
    }

    T searchIterative(Node<T>* x, T k) {          // iterative version
        while (x != nullptr && k != x->key) {
            if (k < x->key) x = x->left;
            else x = x->right;
        }
        return (x == nullptr) ? -1 : x->key;
    }

    // (2)

    T minTree(Node<T>* x) {
        while (x->left != nullptr) x = x->left;
        return x->key;
    }

    T successor(Node<T>* x) {
        if (x->right != nullptr) {
            return minTree(x->right);
        } else {
            Node<T>* y = x->parent;
            while (y != nullptr && x == y->right) {
                x = y;
                y = y->parent;
            }
            return (y == nullptr) ? -1 : y->key;
        }
    }

    T maxTree(Node<T>* x) {
        while (x->right != nullptr) x = x->right;
    }
};

```

```

    return x->key;
}

T predecessor(Node<T>* x) {
    if (x->left != nullptr) {
        return maxTree(x->left);
    } else {
        Node<T>* y = x->parent;
        while (y != nullptr && x == y->left) {
            x = y;
            y = y->parent;
        }
        return (y == nullptr) ? -1 : y->key;
    }
}

// (3)

void inorderTreeWalk(Node<T>* x) {
    if (x != nullptr) {
        inorderTreeWalk(x->left);
        std::cout << x->key << " ";
        inorderTreeWalk(x->right);
    }
}

void preorderTreeWalk(Node<T>* x) {
    if (x != nullptr) {
        std::cout << x->key << " ";
        preorderTreeWalk(x->left);
        preorderTreeWalk(x->right);
    }
}

void postorderTreeWalk(Node<T>* x) {
    if (x != nullptr) {
        postorderTreeWalk(x->left);
        postorderTreeWalk(x->right);
        std::cout << x->key << " ";
    }
}
};


```

The following code tests the correctness of the above functions.

```

#include <iostream>
#include "BST.h"
using namespace std;

// Helper function: insert a new node into the BST
template <class T>
Node<T>* insertNode(BST<T>& tree, T key) {
    Node<T>* z = new Node<T>{key, nullptr, nullptr, nullptr};
    Node<T>* y = nullptr;
    Node<T>* x = tree.root;

    while (x != nullptr) {
        y = x;
        if (key < x->key)
            x = x->left;
        else
            x = x->right;
    }

    z->parent = y;
    if (y == nullptr)
        tree.root = z;
    else if (key < y->key)
        y->left = z;
    else
        y->right = z;

    return z;
}

// Helper to print section headers
void printHeader(const string& title) {
    cout << "\n===== " << title << " =====\n";
}

int main() {
    BST<int> tree{};
    tree.root = nullptr;

    // Build a tree with various shapes
    //          15
    //         /   \
    //        6    18
    //       / \   / \
    //      3  7  17  20
    //     / \   \
    //    2  4   13
    //           /
    //          9

    Node<int>* n15 = insertNode(tree, 15);
    Node<int>* n6  = insertNode(tree, 6);
}

```

```

Node<int>* n18 = insertNode(tree, 18);
Node<int>* n3  = insertNode(tree, 3);
Node<int>* n7  = insertNode(tree, 7);
Node<int>* n17 = insertNode(tree, 17);
Node<int>* n20 = insertNode(tree, 20);
Node<int>* n2  = insertNode(tree, 2);
Node<int>* n4  = insertNode(tree, 4);
Node<int>* n13 = insertNode(tree, 13);
Node<int>* n9  = insertNode(tree, 9);

// (3) Traversals
printHeader("Tree Traversals");
cout << "Inorder:   ";
tree.inorderTreeWalk(tree.root);
cout << "\nPreorder:  ";
tree.preorderTreeWalk(tree.root);
cout << "\nPostorder: ";
tree.postorderTreeWalk(tree.root);
cout << endl;

// (1) Search tests
printHeader("Search Tests");
int keys_to_search[] = {13, 9, 15, 1, 22};
for (int k : keys_to_search) {
    int r1 = tree.searchRecursive(tree.root, k);
    int r2 = tree.searchIterative(tree.root, k);
    cout << "Key " << k << ":" ;
    if (r1 != -1)
        cout << "found (recursive=" << r1 << ", iterative=" << r2 << ")\n";
    else
        cout << "not found\n";
}

// (2) Successor / Predecessor tests
printHeader("Successor / Predecessor Tests");
auto testSuccPred = [&](Node<int>* node) {
    cout << "Node " << node->key << ":" ;
    int succ = tree.successor(node);
    int pred = tree.predecessor(node);
    cout << "successor=" << ((succ == -1) ? string("null") : to_string(succ))
        << ", predecessor=" << ((pred == -1) ? string("null") : to_string(pred)) << "\n";
};

testSuccPred(n15); // root
testSuccPred(n6); // internal node
testSuccPred(n2); // smallest
testSuccPred(n20); // largest
testSuccPred(n13); // internal with left child
testSuccPred(n9); // leaf

// Additional trees

```

```

printHeader("Single-node Tree");
BST<int> single{};
Node<int>* s1 = insertNode(single, 10);
single.inorderTreeWalk(single.root);
cout << "\nSuccessor=" << single.successor(s1)
    << ", Predecessor=" << single.predecessor(s1) << endl;

printHeader("Left-skewed Tree");
BST<int> left{};
Node<int>* a = insertNode(left, 5);
Node<int>* b = insertNode(left, 4);
Node<int>* c = insertNode(left, 3);
Node<int>* d = insertNode(left, 2);
Node<int>* e = insertNode(left, 1);
left.inorderTreeWalk(left.root);
cout << "\nSuccessor(3)=" << left.successor(c)
    << ", Predecessor(3)=" << left.predecessor(c) << endl;

printHeader("Right-skewed Tree");
BST<int> right{};
Node<int>* r1 = insertNode(right, 1);
Node<int>* r2 = insertNode(right, 2);
Node<int>* r3 = insertNode(right, 3);
Node<int>* r4 = insertNode(right, 4);
Node<int>* r5 = insertNode(right, 5);
right.inorderTreeWalk(right.root);
cout << "\nSuccessor(3)=" << right.successor(r3)
    << ", Predecessor(3)=" << right.predecessor(r3) << endl;

cout << "\nAll tests completed.\n";
return 0;
}

```

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022).
Introduction to Algorithms (4th ed.). MIT Press.

Acknowledgement

ChatGPT-5 provided valuable feedback and suggestions on [the test code implementation, wording and sentence structure](#).

Appendix: Lab Specification

Write code for the following methods of binary search trees.

(1) SEARCH(x, k) (recursion and iterative version); (20%)

(2) SUCCESSOR(x) and PREDECESSOR(x); (30%)

(3) INORDER-TREE-WALK(T), PREORDER-TREE-WALK(T), POSTORDER-TREE-WALK(T); (50%)

Note. T points the root of a binary search tree, x points any node in a binary search tree, and k denotes a key.