Lecture 06: Binary Search Trees

C++ Code Samples — Sedgwick Algorithms Course — lecture-06-samples.cpp

```cpp
// ============================================================================
// Lecture 06: Binary Search Trees
// Sedgwick Algorithms Course
//
// Topics covered:
//   1. BST node struct with insert, search, delete
//   2. In-order, pre-order, post-order traversal
//   3. Find min/max in BST
//   4. BST successor and predecessor
//   5. Demo building a BST and showing all operations
// ============================================================================

#include <iostream>
#include <vector>
#include <string>
#include <functional>

using namespace std;

// === SECTION: BST Node and Core Operations ===
// A BST maintains the invariant: left < node < right for all nodes.
// Average case: O(log N) for insert/search/delete.
// Worst case (degenerate tree): O(N).

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

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

// Insert a key into the BST. Returns the (possibly new) root.
BSTNode *insert(BSTNode *node, int key) {
    if (node == nullptr) return new BSTNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    // Duplicate keys are ignored in this implementation
    return node;
}

// Search for a key. Returns the node if found, nullptr otherwise.
BSTNode *search(BSTNode *node, int key) {
    if (node == nullptr || node->key == key) return node;
    if (key < node->key) return search(node->left, key);
    return search(node->right, key);
}

// Find the node with the minimum key (leftmost node).
BSTNode *findMin(BSTNode *node) {
    if (node == nullptr) return nullptr;
    while (node->left != nullptr) node = node->left;
    return node;
}

// Find the node with the maximum key (rightmost node).
BSTNode *findMax(BSTNode *node) {
    if (node == nullptr) return nullptr;
    while (node->right != nullptr) node = node->right;
```

```
    return root;
}

// Delete a key from the BST. Returns the (possibly new) root.
// Three cases:
//   1. Leaf node: simply remove it
//   2. One child: replace node with its child
//   3. Two children: replace with in-order successor, then delete successor
TreeNode * deleteNode(TreeNode * root, int key) {
    if (root == nullptr) return nullptr;

    if (key < root->key) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->key) {
        root->right = deleteNode(root->right, key);
    } else {
        // Found the node to delete
        if (root->left == nullptr) {
            // Case 1 or 2: no left child
            TreeNode * temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            // Case 2: no right child
            TreeNode * temp = root->left;
            delete root;
            return temp;
        } else {
            // Case 3: two children -- replace with in-order successor
            TreeNode * successor = findMin(root->right);
            root->key = successor->key;
            root->right = deleteNode(root->right, successor->key);
        }
    }
    return root;
}

// === SECTION: Traversals ===
// In-order (Left, Root, Right) -- produces sorted output for a BST
// Pre-order (Root, Left, Right) -- useful for copying/serializing
// Post-order (Left, Right, Root) -- useful for deletion/cleanup

void inOrder(TreeNode * root, vector<int>& result) {
    if (root == nullptr) return;
    inOrder(root->left, result);
    result.push_back(root->key);
    inOrder(root->right, result);
}

void preOrder(TreeNode * root, vector<int>& result) {
    if (root == nullptr) return;
    result.push_back(root->key);
    preOrder(root->left, result);
    preOrder(root->right, result);
}

void postOrder(TreeNode * root, vector<int>& result) {
    if (root == nullptr) return;
    postOrder(root->left, result);
    postOrder(root->right, result);
    result.push_back(root->key);
}

// === SECTION: Successor and Predecessor ===
```

```
// Successor: the node with the smallest key greater than the given key.
// Predecessor: the node with the largest key smaller than the given key.

// In-order successor of a given key.
// Strategy: if node has right subtree, successor is min of right subtree.
// Otherwise, walk from root keeping track of the last left-turn ancestor.
BSTNode * successor(BSTNode * root, int key) {
    BSTNode * succ = nullptr;
    BSTNode * curr = root;
    while (curr != nullptr) {
        if (key < curr->key) {
            succ = curr;            // This node could be successor
            curr = curr->left;
        } else if (key > curr->key) {
            curr = curr->right;
        } else {
            // Found the node
            if (curr->right != nullptr) {
                return minValueNode(curr->right);
            }
            break;
        }
    }
    return succ;
}

// In-order predecessor of a given key.
BSTNode * predecessor(BSTNode * root, int key) {
    BSTNode * pred = nullptr;
    BSTNode * curr = root;
    while (curr != nullptr) {
        if (key > curr->key) {
            pred = curr;            // This node could be predecessor
            curr = curr->right;
        } else if (key < curr->key) {
            curr = curr->left;
        } else {
            // Found the node
            if (curr->left != nullptr) {
                return maxValueNode(curr->left);
            }
            break;
        }
    }
    return pred;
}

// === SECTION: Utility ===

void printVector(const vector<int>& v, const string &name) {
    cout << name << ": [";
    for (int i = 0; i < (int)v.size(); ++i) {
        if (i > 0) cout << ", ";
        cout << v[i];
    }
    cout << "]" << endl;
}

// Print the tree structure with indentation (rotated 90 degrees)
void printTree(BSTNode *node, const string &prefix, bool isLeft) {
    if (node == nullptr) return;
    cout << prefix;
    cout << (isLeft ? "|-- " : "\\-- ");
    cout << node->key << endl;
```

```
    printTree(root->left, prefix + (isLeft ? "|   " : "    "), true);
    printTree(root->right, prefix + (isLeft ? "|   " : "    "), false);
}

// Free all nodes
void freeTree(BstNode* root) {
    if (root == nullptr) return;
    freeTree(root->left);
    freeTree(root->right);
    delete root;
}

// === MAIN ===

int main() {
    cout << "========================================" << endl;
    cout << " Lecture 06: Binary Search Trees" << endl;
    cout << "========================================" << endl;

    // --- Demo 1: Build a BST ---
    cout << "\n--- Building BST ---" << endl;
    cout << "Insert order: 50, 30, 70, 20, 40, 60, 80, 10, 35" << endl;

    BstNode* root = nullptr;
    int keys[] = {50, 30, 70, 20, 40, 60, 80, 10, 35};
    for (int k : keys) {
        root = insert(root, k);
    }

    cout << "\nTree structure:" << endl;
    printTree(root, "", false);

    // --- Demo 2: Traversals ---
    cout << "\n--- Traversals ---" << endl;
    vector<int> result;

    inOrder(root, result);
    printVector(result, "In-order  ");
    result.clear();

    preOrder(root, result);
    printVector(result, "Pre-order ");
    result.clear();

    postOrder(root, result);
    printVector(result, "Post-order");
    result.clear();

    // --- Demo 3: Search ---
    cout << "\n--- Search ---" << endl;
    for (int k : {40, 55, 80}) {
        BstNode* found = search(root, k);
        cout << "Search " << k << ": " << (found ? "FOUND" : "NOT FOUND") << endl;
    }

    // --- Demo 4: Min and Max ---
    cout << "\n--- Min / Max ---" << endl;
    cout << "Min: " << findMin(root)->key << endl;
    cout << "Max: " << findMax(root)->key << endl;

    // --- Demo 5: Successor and Predecessor ---
    cout << "\n--- Successor / Predecessor ---" << endl;
    for (int k : {20, 35, 50, 70, 80}) {
        BstNode* s = successor(root, k);
```

```
    // --- Demo 6: Delete ---
    cout << "\n--- Delete Operations ---" << endl;

    cout << "  Deleting 20 (leaf):" << endl;
    root = deleteNode(root, 20);
    printTree(root, "", false);

    cout << "  Deleting 30 (one child):" << endl;
    root = deleteNode(root, 30);
    printTree(root, "", false);

    cout << "  Deleting 50 (two children, root):" << endl;
    root = deleteNode(root, 50);
    printTree(root, "", false);

    inOrder(root, result);
    printVector(result, "  Final in-order");

    freeTree(root);
    return 0
}
```