Lecture 07: Balanced Search Trees (Red-Black BSTs)

C++ Code Samples — Sedgwick Algorithms Course — lecture-07-samples.cpp

```cpp
// ============================================================================
// Lecture 07: Balanced Search Trees (Red-Black BSTs)
// Sedgwick Algorithms Course
//
// Topics covered:
//   1. Left-Leaning Red-Black BST (LLRB) implementation
//   2. Insert with rotations and color flips
//   3. Search operation
//   4. In-order traversal showing node colors
//   5. Demo: insert sequence and show balanced tree height
//
// Key invariants of a Left-Leaning Red-Black BST:
//   - No node has two red links connected to it
//   - Every path from root to null has the same number of black links
//   - Red links lean left (no right-leaning red links)
//   - The root is always black
// ============================================================================

#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <queue>

using namespace std;

// === SECTION: LLRB Node and Color Constants ===

const bool RED   = true;
const bool BLACK = false;

struct RBNode {
    int key;
    RBNode* left;
    RBNode* right;
    bool color;    // Color of the link from parent to this node
    int size;      // Number of nodes in subtree (for rank queries)

    RBNode(int k, bool c) : key(k), left(nullptr), right(nullptr), color(c), size(1) {}
};

// === SECTION: Helper Functions ===

bool isRed(RBNode* node) {
    if (node == nullptr) return false;   // Null links are black
    return node->color == RED;
}

int nodeSize(RBNode* node) {
    return node == nullptr ? 0 : node->size;
}

void updateSize(RBNode* node) {
    if (node != nullptr)
        node->size = 1 + nodeSize(node->left) + nodeSize(node->right);
}

// === SECTION: Rotations and Color Flip ===
// These are the fundamental operations that maintain balance.

// Rotate left: fix a right-leaning red link
//     h            x
//    / \          / \
//   a   x  =>   h   c
//      / \     / \
//     b   c   a   b
RBNode* rotateLeft(RBNode* h) {
    RBNode* x = h->right;
    h->right = x->left;
    x->left = h;
    x->color = h->color;
    h->color = RED;
    updateSize(h);
    updateSize(x);
    return x;
}

// Rotate right: temporarily used during insert to fix two left reds
//       h        x
//      / \      / \
//     x   c => a   h
//    / \          / \
//   a   b        b   c
RBNode* rotateRight(RBNode* h) {
    RBNode* x = h->left;
    h->left = x->right;
    x->right = h;
    x->color = h->color;
    h->color = RED;
    updateSize(h);
    updateSize(x);
    return x;
}

// Flip colors: split a temporary 4-node
// Both children are red -> make them black and this node red
void flipColors(RBNode* h) {
    h->color = !h->color;
    if (h->left) h->left->color = !h->left->color;
    if (h->right) h->right->color = !h->right->color;
}

// === SECTION: Insert ===
// Insert follows standard BST insertion, then fixes up the tree bottom-up:
//   1. If right child is red and left is black -> rotate left
//   2. If left child and left-left grandchild are both red -> rotate right
//   3. If both children are red -> flip colors

RBNode* insertHelper(RBNode* h, int key) {
    // Standard BST insert at the bottom (new node is always red)
    if (h == nullptr) return new RBNode(key, RED);

    if (key < h->key)
        h->left = insertHelper(h->left, key);
    else if (key > h->key)
        h->right = insertHelper(h->right, key);
    else
        return h;   // Duplicate key, no insert

    // Fix-up: enforce LLRB invariants on the way back up
    if (isRed(h->right) && !isRed(h->left))       h = rotateLeft(h);
    if (isRed(h->left) && isRed(h->left->left))    h = rotateRight(h);
    if (isRed(h->left) && isRed(h->right))          flipColors(h);

    updateSize(h);
    return h;
}

// === SECTION: LLRB Tree Class ===

class LLRBTree {
    RBNode* root;

    void inOrderHelper(RBNode* node, vector<pair<int, string>>& result) {
        if (node == nullptr) return;
        inOrderHelper(node->left, result);
        result.push_back({node->key, isRed(node) ? "RED" : "BLK"});
        inOrderHelper(node->right, result);
    }

    int heightHelper(RBNode* node) {
        if (node == nullptr) return 0;
        return 1 + max(heightHelper(node->left), heightHelper(node->right));
    }

    int blackHeightHelper(RBNode* node) {
        // Count black links from root to any null (should be same for all paths)
        if (node == nullptr) return 0;
        int leftBH = blackHeightHelper(node->left);
        return leftBH + (isRed(node) ? 0 : 1);
    }

    void printTreeHelper(RBNode* node, const string& prefix, bool isLeft) {
        if (node == nullptr) return;
        cout << prefix;
        cout << (isLeft ? "|-- " : "\\-- ");
        cout << node->key << " (" << (isRed(node) ? "R" : "B") << ")" << endl;
        printTreeHelper(node->left, prefix + (isLeft ? "|   " : "    "), true);
        printTreeHelper(node->right, prefix + (isLeft ? "|   " : "    "), false);
    }

    void freeHelper(RBNode* node) {
        if (node == nullptr) return;
        freeHelper(node->left);
        freeHelper(node->right);
        delete node;
    }

public:
    LLRBTree() : root(nullptr) {}
    ~LLRBTree() { freeHelper(root); }

    void insert(int key) {
        root = insertHelper(root, key);
        root->color = BLACK;  // Root is always black
    }

    // Search: identical to standard BST search (colors don't affect it)
    bool search(int key) {
        RBNode* curr = root;
        while (curr != nullptr) {
            if (key < curr->key) curr = curr->left;
            else if (key > curr->key) curr = curr->right;
            else return true;
        }
        return false;
    }

    // In-order traversal with colors
```

```cpp
    vector<pair<int, string>> inOrderWithColors() {
        vector<pair<int, string>> result;
        inOrderHelper(root, result);
        return result;
    }

    int height() { return heightHelper(root); }
    int blackHeight() { return blackHeightHelper(root); }
    int size() { return nodeSize(root); }

    void printTree() {
        if (root == nullptr) { cout << "  (empty)" << endl; return; }
        printTreeHelper(root, "  ", false);
    }
};

// === MAIN ===

int main() {
    cout << "========================================" << endl;
    cout << " Lecture 07: Red-Black BSTs (LLRB)" << endl;
    cout << "========================================" << endl;

    // --- Demo 1: Build LLRB Tree incrementally ---
    cout << "\n--- Building LLRB Tree ---" << endl;
    cout << "Insert order: 10, 20, 30, 40, 50, 25, 35, 5, 15" << endl;
    cout << "(Worst case for a plain BST -- shows how RB stays balanced)" << endl;

    LLRBTree tree;
    int keys[] = {10, 20, 30, 40, 50, 25, 35, 5, 15};

    for (int k : keys) {
        tree.insert(k);
        cout << "\n  After insert " << k
             << " (height=" << tree.height()
             << ", black-height=" << tree.blackHeight()
             << ", size=" << tree.size() << "):" << endl;
        tree.printTree();
    }

    // --- Demo 2: Search ---
    cout << "\n--- Search Operations ---" << endl;
    for (int k : {25, 42, 50, 7}) {
        cout << "  Search " << k << ": " << (tree.search(k) ? "FOUND" : "NOT FOUND") << endl;
    }

    // --- Demo 3: In-order traversal with colors ---
    cout << "\n--- In-Order Traversal with Colors ---" << endl;
    auto ordered = tree.inOrderWithColors();
    cout << "  ";
    for (auto& [key, color] : ordered) {
        cout << key << "(" << color << ") ";
    }
    cout << endl;

    // --- Demo 4: Compare heights ---
    cout << "\n--- Balance Analysis ---" << endl;
    cout << "  Nodes inserted:  " << tree.size() << endl;
    cout << "  Actual height:   " << tree.height() << endl;
    cout << "  Black height:    " << tree.blackHeight() << endl;
    cout << "  Theoretical max: " << (int)ceil(2.0 * log2(tree.size() + 1)) << endl;
    cout << "  (LLRB height <= 2 * lg(N+1))" << endl;

    // --- Demo 5: Inserting sorted data (BST worst case) ---
    cout << "\n--- Sorted Insert Stress Test ---" << endl;
    LLRBTree tree2;
    cout << "  Inserting 1..31 in order (worst case for plain BST):" << endl;
    for (int i = 1; i <= 31; i++) {
        tree2.insert(i);
    }
    cout << "  31 nodes, plain BST height would be 31" << endl;
    cout << "  LLRB actual height:   " << tree2.height() << endl;
    cout << "  LLRB black height:    " << tree2.blackHeight() << endl;
    cout << "  Theoretical max:      " << (int)ceil(2.0 * log2(32)) << endl;

    return 0;
}
```