```
/*
Alexis Aguilar
CSCE 3110.001
Assignment 3
Question 2
Fall 2024
*/

#include <stdio.h>
#include <iostream>
#include "avl.hpp"
using namespace std;

#define IS_ROOT 0
#define IS_LEFT 1
#define IS_RIGHT 2

/**
 * Returns the height of a given node
 * @param node The node whose height we want to get
 * @return -1 if node is NULL, otherwise the height stored in the node
 */
int height(AvlNode* node) {
    return node == NULL ? -1 : node->height;
}

/**
 * Updates the height of a node based on its children's heights
 * Height is calculated as 1 + max(left child height, right child height)
 * @param node The node whose height needs to be updated
 */
void updateHeight(AvlNode* node) {
    if (node != NULL) {
        node->height = 1 + max(height(node->left), height(node->right));
    }
}

/**
 * Calculates the balance factor of a node
 * Balance factor = height(left subtree) - height(right subtree)
 * @param node The node whose balance factor we want to calculate
 * @return Balance factor of the node. A value > 1 means left-heavy, < -1 means
right-heavy
 */
int getBalance(AvlNode* node) {
    return node == NULL ? 0 : height(node->left) - height(node->right);
}

/**
 * Performs a right rotation on a given node
 * Used to balance the tree when a node becomes left-heavy
 *
 *      y                  x
 *     / \                / \
 *    x   T3     =>     T1  y
 *   / \                   / \
 *  T1  T2                T2  T3
 *
 * @param y The root of the subtree to be rotated
```

```
 * @return The new root after rotation (x)
 */
AvlNode* rotateRight(AvlNode* y) {
    AvlNode* x = y->left;
    AvlNode* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    updateHeight(y);  // Update y first as it becomes lower level
    updateHeight(x);  // Update x after as it becomes the new root

    return x;  // New root
}

/**
 * Performs a left rotation on a given node
 * Used to balance the tree when a node becomes right-heavy
 *
 *    x                    y
 *   / \                  /   \
 * T1   y       =>      x     T3
 *     / \             /  \
 *    T2  T3         T1   T2
 *
 * @param x The root of the subtree to be rotated
 * @return The new root after rotation (y)
 */
AvlNode* rotateLeft(AvlNode* x) {
    AvlNode* y = x->right;
    AvlNode* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    updateHeight(x);  // Update x first as it becomes lower level
    updateHeight(y);  // Update y after as it becomes the new root

    return y;  // New root
}

/**
 * Finds the node with minimum value in a given subtree
 * Used in deletion when removing a node with two children
 * @param node Root of the subtree to search
 * @return Pointer to the node with minimum value
 */
AvlNode* findMin(AvlNode* node) {
    if (node == NULL) return NULL;
    while (node->left != NULL)
        node = node->left;
    return node;
}

/**
```

```
 * Balances a node if its balance factor indicates it's unbalanced
 * Handles all four cases of imbalance:
 * 1. Left-Left (single right rotation)
 * 2. Left-Right (left rotation then right rotation)
 * 3. Right-Right (single left rotation)
 * 4. Right-Left (right rotation then left rotation)
 *
 * @param node The node to check and balance if necessary
 * @return The new root of the balanced subtree
 */
AvlNode* balance(AvlNode* node) {
    if (node == NULL) return NULL;

    // Update height of current node
    updateHeight(node);

    // Get balance factor to check if node is unbalanced
    int balance = getBalance(node);

    // Left Heavy (balance > 1)
    if (balance > 1) {
        // Check if it's Left-Left or Left-Right case
        if (getBalance(node->left) >= 0) {
            // Left-Left Case: single right rotation
            return rotateRight(node);
        } else {
            // Left-Right Case: left rotation on left child, then right rotation
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }
    }

    // Right Heavy (balance < -1)
    if (balance < -1) {
        // Check if it's Right-Right or Right-Left case
        if (getBalance(node->right) <= 0) {
            // Right-Right Case: single left rotation
            return rotateLeft(node);
        } else {
            // Right-Left Case: right rotation on right child, then left rotation
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }
    }

    // Node is balanced
    return node;
}

/**
 * Helper function for insertion
 * Recursively finds the correct position to insert the new value
 * and balances the tree after insertion
 *
 * @param node Current node in recursion
 * @param info Value to insert
 * @return New root of the subtree after insertion and balancing
 */
AvlNode* insertHelper(AvlNode* node, const int& info) {
```

```
    // Base case: empty spot found, create new node
    if (node == NULL)
        return new AvlNode(info, NULL, NULL);

    // Recursive insertion based on BST property
    if (info < node->element)
        node->left = insertHelper(node->left, info);
    else if (info > node->element)
        node->right = insertHelper(node->right, info);
    else
        return node;  // Duplicate value, ignore

    // Balance the current node after insertion
    return balance(node);
}

/**
 * Main insertion function that maintains AVL tree properties
 * @param info Value to insert
 * @param root Reference to the root of the tree
 */
void insert(const int& info, AvlNode*& root) {
    root = insertHelper(root, info);
}

/**
 * Helper function for deletion
 * Recursively finds the node to delete and handles three cases:
 * 1. Node has no children
 * 2. Node has one child
 * 3. Node has two children
 *
 * @param node Current node in recursion
 * @param info Value to delete
 * @return New root of the subtree after deletion and balancing
 */
AvlNode* removeHelper(AvlNode* node, const int& info) {
    if (node == NULL)
        return NULL;

    // Recursively find the node to delete
    if (info < node->element)
        node->left = removeHelper(node->left, info);
    else if (info > node->element)
        node->right = removeHelper(node->right, info);
    else {
        // Node to delete found - handle the three cases
        if (node->left == NULL || node->right == NULL) {
            // Case 1 & 2: No child or one child
            AvlNode* temp = node->left ? node->left : node->right;
            if (temp == NULL) {
                // No child case
                temp = node;
                node = NULL;
            } else {
                // One child case: copy contents of child to current node
                *node = *temp;
            }
            delete temp;
```

```
        } else {
            // Case 3: Two children
            // Find successor (minimum value in right subtree)
            AvlNode* temp = findMin(node->right);
            node->element = temp->element;
            // Delete the successor
            node->right = removeHelper(node->right, temp->element);
        }
    }

    if (node == NULL)
        return NULL;

    // Balance the current node after deletion
    return balance(node);
}

/**
 * Main deletion function that maintains AVL tree properties
 * @param info Value to delete
 * @param root Reference to the root of the tree
 */
void remove(const int& info, AvlNode*& root) {
    root = removeHelper(root, info);
}

// Print methods - provided in original code, unchanged
void print(AvlNode *root, int level, int type) {
    if (root == NULL) {
        return;
    }
    if (type == IS_ROOT) {
        std::cout << root->element << "\n";
    } else {
        for (int i = 1; i < level; i++) {
            std::cout << "| ";
        }
        if (type == IS_LEFT) {
            std::cout << "|l_" << root->element << "\n";
        } else {
            std::cout << "|r_" << root->element << "\n";
        }
    }
    if (root->left != NULL) {
        print(root->left, level + 1, IS_LEFT);
    }
    if (root->right != NULL) {
        print(root->right, level + 1, IS_RIGHT);
    }
}

void print(AvlNode *root) {
    print(root, 0, IS_ROOT);
}

/**
 * Main function - reads and processes commands from input file
 * Supports three operations:
 * - insert <value>: Inserts a value into the AVL tree
```

```
 * - delete <value>: Removes a value from the AVL tree
 * - print: Displays the current state of the tree
 */
int main(int argc, const char * argv[]) {
    AvlNode *root = NULL;
    std::string filename = argv[1];
    freopen(filename.c_str(), "r", stdin);
    std::string input;
    int data;
    while(std::cin >> input){
        if (input == "insert"){
            std::cin>>data;
            insert(data, root);
        } else if(input == "delete"){
            std::cin>>data;
            remove(data, root);
        } else if(input == "print"){
            print(root);
            std::cout << "\n";
        } else{
            std::cout<<"Data not consistent in file";
            break;
        }
    }
    return 0;
}
```