```
#ifndef AVLTREE_H
#define AVLTREE_H

#include "Node.h"
#include <string>
#include <queue>
using std::string;
using std::to_string;
using std::queue;
using std::max;

/// Object class for a self-balancing (Adelson-Velsky and Landis) tree
class AVLTree {

private:
    Node* _root = nullptr;
    static Node* _rotateLeft(Node* root);
    static Node* _rotateRight(Node* root);
    static Node* _rotateRightLeft(Node* root);
    static Node* _rotateLeftRight(Node* root);
    static Node* _getLeftmost(Node* root);
    static int _getHeight(Node* root);
    static void _updateHeight(Node* root);
    static int _getBalance(Node* root);
    static Node* _updateBalance(Node* root);
    static Node* _deleteNode(Node* root, Node* parent);
    static string _idToString(int id);
    static void _copyInorder(Node* root, string& names);
    static void _copyPreorder(Node* root, string& names);
    static void _copyPostorder(Node* root, string& names);
    static void _copyLevelorder(Node* root, string& names);
    static Node* _insert(Node* root, int id, const string& name);
    static bool _remove(Node* root, Node* parent, int id);
    static bool _removeInorder(Node* root, Node* parent, int* count);
    static void _search(Node* root, int id, string& match);
    static void _search(Node* root, const string& name, vector<string>& matches);

public:
    enum Traversal { INORDER, PREORDER, POSTORDER, LEVELORDER };
    bool insert(int id, const string& name);
    bool remove(int id);
    bool removeInorder(int count);
    string search(int id);
    vector<string> search(const string& name);
    string traversalToString(Traversal type);
    int levelCount();
};

/**
 *  @brief  Rotate a left-left tree branch clockwise
 *  @param  root pointer to the root @c Node of the tree branch
 *  @return pointer to the root @c Node of the rotated tree branch
 */
Node* AVLTree::_rotateRight(Node* root)
{
    Node* left = root->left;
    Node* leftRight = left->right;
    left->right = root;
    root->left = leftRight;
    return left;
```

```
}

/**
 *  @brief  Rotate a right-right tree branch counterclockwise
 *  @param  root pointer to the root @c Node of the tree branch
 *  @return pointer to the root @c Node of the rotated tree branch
 */
Node* AVLTree::_rotateLeft(Node* root)
{
    Node* right = root->right;
    Node* rightLeft = right->left;
    right->left = root;
    root->right = rightLeft;
    return right;
}

/**
 *  @brief  Rotate a right-left tree branch; first, clockwise, then, counterclockwise
 *  @param  root pointer to the root @c Node of the tree branch
 *  @return pointer to the root @c Node of the rotated tree branch
 */
Node* AVLTree::_rotateRightLeft(Node* root)
{
    root->right = _rotateRight(root->right);
    return _rotateLeft(root);
}

/**
 *  @brief  Rotate a left-right tree branch; first, counterclockwise, then, clockwise
 *  @param  root pointer to the root @c Node of the tree branch
 *  @return pointer to the root @c Node of the rotated tree branch
 */
Node* AVLTree::_rotateLeftRight(Node* root)
{
    root->left = _rotateLeft(root->left);
    return _rotateRight(root);
}

/**
 *  @brief  Get the leftmost @c Node of a tree branch
 *  @param  root pointer to the root @c Node of the tree branch
 *  @return pointer to the leftmost @c Node of the tree branch
 */
Node* AVLTree::_getLeftmost(Node* root)
{
    if (!root) return nullptr;
    if (!root->left->left)
    {
        Node* leftmost = root->left;
        root->left = leftmost->right;
        return leftmost;
    }
    return _getLeftmost(root->left);
}

/**
 *  @brief  Get the height of a tree
 *  @param  root pointer to the root @c Node of the tree
 *  @return height of the tree
 */
```

```
int AVLTree::_getHeight(Node* root)
{
    if (!root) return -1;
    int left_height = _getHeight(root->left);
    int right_height = _getHeight(root->right);
    return 1 + max(left_height, right_height);
}

/**
 *  @brief Update the height of a tree
 *  @param root pointer to the root @c Node of the tree
 */
void AVLTree::_updateHeight(Node* root)
{
    Node* left = root->left;
    Node* right = root->right;
    if (left) left->height = _getHeight(root->left);
    root->height = _getHeight(root);
    if (right) right->height = _getHeight(root->right);
}

/**
 *  @brief  Get the balance factor of a tree
 *  @param  root pointer to the root @c Node of the tree
 *  @return balance factor of the tree
 */
int AVLTree::_getBalance(Node* root)
{
    return _getHeight(root->left) - _getHeight(root->right);
}

/**
 *  @brief  Update the balance factor of a tree
 *  @param  root  pointer to the root @c Node of the tree
 *  @return pointer to the root @c Node of the updated tree
 */
Node* AVLTree::_updateBalance(Node* root)
{
    const int balance = _getBalance(root);
    if (balance < -1)
    {
        if (_getBalance(root->right) == 1) root = _rotateRightLeft(root);
        else root = _rotateLeft(root);
    }
    else if (balance > 1)
    {
        if (_getBalance(root->left) == -1) root = _rotateLeftRight(root);
        else root = _rotateRight(root);
    }
    return root;
}

/**
 *  @brief  Delete a @c Node from a tree
 *  @param  root  pointer to the root @c Node of the tree
 *  @param  parent  pointer to the parent of the root @c Node
 *  @return pointer to the root @c Node of the updated tree
 */
Node* AVLTree::_deleteNode(Node* root, Node* parent)
{
```

```
    Node* left = root->left;
    Node* right = root->right;

    if (!left && !right)
    {
        Node* removal = root;
        if (parent)
        {
            if (parent->left == root) parent->left = nullptr;
            else if (parent->right == root) parent->right = nullptr;
        }
        delete removal;
        root = parent;
    }

    else
    if (left && !right)
    {
        root->id = left->id;
        root->name = left->name;
        root->left = left->left;
        root->right = left->right;
        delete left;
    }

    else
    if (!left)
    {
        root->id = right->id;
        root->name = right->name;
        root->left = right->left;
        root->right = right->right;
        delete right;
    }

    else
    if (!right->left)
    {
        root->id = right->id;
        root->name = right->name;
        root->right = right->right;
        delete right;
    }

    else
    {
        Node* leftmost = _getLeftmost(right);
        root->id = leftmost->id;
        root->name = leftmost->name;
        delete leftmost;
    }

    return root;
}

/**
 *  @brief  Convert an integer ID to an 8-character string
 *  @param  id integer ID
 *  @return 8-character string representing the integer ID
 */
```

```
string AVLTree::_idToString(const int id)
{
    string prefix;
    string id_string = to_string(id);
    size_t prefix_len = (8 - id_string.length());

    while (prefix_len--) prefix += '0';
    id_string = (prefix + id_string);
    return id_string;
}

/**
 *  @brief Copies a comma-separated inorder traversal to a string
 *  @param root pointer to the root @c Node of a tree
 *  @param names string to which names list is copied
 */
void AVLTree::_copyInorder(Node* root, string& names)
{
    if (!root) return;

    _copyInorder(root->left, names);
    names += (root->name + ", ");
    _copyInorder(root->right, names);
}

/**
 *  @brief Copies a comma-separated preorder traversal to a string
 *  @param root  pointer to the root @c Node of a tree
 *  @param names  string to which names list is copied
 */
void AVLTree::_copyPreorder(Node* root, string& names)
{
    if (!root) return;

    names += (root->name + ", ");
    _copyPreorder(root->left, names);
    _copyPreorder(root->right, names);
}

/**
 *  @brief Copies a comma-separated postorder traversal to a string
 *  @param root  pointer to the root @c Node of a tree
 *  @param names  string to which names list is copied
 */
void AVLTree::_copyPostorder(Node* root, string& names)
{
    if (!root) return;

    _copyPostorder(root->left, names);
    _copyPostorder(root->right, names);
    names += (root->name + ", ");
}

/**
 *  @brief Copies a comma-separated levelorder traversal to a string
 *  @param root  pointer to the root @c Node of a tree
 *  @param names  string to which names list is copied
 */
void AVLTree::_copyLevelorder(Node* root, string& names)
{
```

```
    if (!root) return;

    queue<Node*> nodes_queue;
    nodes_queue.push(root);

    int nodes_counted = 0;
    int nodes_expected = 1;

    while (!nodes_queue.empty())
    {
        Node* current = nodes_queue.front();
        nodes_queue.pop();

        names += (current->name + ", ");

        Node* left = current->left;
        Node* right = current-> right;

        if (left) { nodes_queue.push(left); nodes_counted++; }
        if (right) { nodes_queue.push(right); nodes_counted++; }

        if (!(--nodes_expected))
        {
            nodes_expected = nodes_counted;
            nodes_counted = 0;
        }
    }
}

/**
 *  @brief  Create, with ID and name, and insert a @c Node to a tree
 *  @param  root  pointer to the root @c Node of the tree
 *  @param  id  integer ID
 *  @param  name full name
 *  @return pointer to the root @c Node of the updated tree
 *  @complexity O(log n) (worst-case)
 */
Node* AVLTree::_insert(Node* root, const int id, const string& name)
{
    if (root == nullptr) return new Node(id, name);

    const int root_id = root->id;

    if (id < root_id) root->left = _insert(root->left, id, name);
    else root->right = _insert(root->right, id, name);

    root = _updateBalance(root);

    return root;
}

/**
 *  @brief  Identify, by ID, and remove a @c Node from a tree
 *  @param  root  pointer to the root @c Node of the tree
 *  @param  parent  pointer to the parent of the root @c Node
 *  @param  id integer ID
 *  @return boolean indicating whether a node was removed
 *  @complexity O(log n) (worst-case)
 */
bool AVLTree::_remove(Node* root, Node* parent, const int id)
```

```
{
    if (!root) return false;
    int root_id = root->id;
    bool left_is_removed = false, current_is_removed = false, right_is_removed = false;
    if (id < root_id) left_is_removed = _remove(root->left, root, id); else
    if (id > root_id) right_is_removed = _remove(root->right, root, id); else
    {
        root = _deleteNode(root, parent);
        current_is_removed = true;
        if (root) _updateHeight(root);
    }
    return (left_is_removed || current_is_removed || right_is_removed);
}

/**
 *  @brief  Identify, by inorder count, and remove a @c Node from a tree
 *  @param  root  pointer to the root @c Node of the tree
 *  @param  parent  pointer to the parent of the root @c Node
 *  @param  count pointer to a counter that preserves its state through recursive calls
 *  @return boolean indicating whether a node was removed
 *  @complexity O(log n) (worst-case)
 */
bool AVLTree::_removeInorder(Node* root, Node* parent, int* count)
{
    if (!root) return false;
    bool left_is_removed = _removeInorder(root->left, root, count);
    bool current_is_removed = false;
    if (!((*count)--))
    {
        root = _deleteNode(root, parent);
        if (root) _updateHeight(root);
        current_is_removed = true;
    }
    bool right_is_removed = false;
    if (root) right_is_removed = _removeInorder(root->right, root, count);
    return (left_is_removed || current_is_removed || right_is_removed);
}

/**
 *  @brief Search, by ID, and copy, to a string, the name of a @c Node from a tree
 *  @param root  pointer to the root @c Node of the tree
 *  @param id integer ID
 *  @param match string to which name of found ID is copied
 *  @complexity O(log n) (worst-case)
 */
void AVLTree::_search(Node* root, const int id, string& match)
{
    if (!root) return;

    const int root_id = root->id;
    if (id == root_id) match = root->name;
    if (id < root_id) _search(root->left, id, match);
    if (id > root_id) _search(root->right, id, match);
}

/**
 *  @brief Search, by name, and copy a list of IDs of @c Nodes from a tree
 *  @param root  pointer to the root @c Node of the tree
 *  @param name  full name
 *  @param matches vector to which list of IDs is to be copied
```

```
 *  @complexity O(n) (worst-case)
 */
void AVLTree::_search(Node* root, const string& name, vector<string>& matches)
{
    if (!root) return;

    const string root_name = root->name;
    if (name == root_name) matches.push_back(_idToString(root->id));
    _search(root->left, name, matches);
    _search(root->right, name, matches);
}

/**
 *  @brief  Get a comma-separated list of names from nodes in @c this tree
 *  @param  type type of tree traversal to generate the list from
 *  @return comma-separated list of names from a tree as a string
 *  @complexity O(n) (worst-case)
 */
string AVLTree::traversalToString(const Traversal type)
{
    if (!_root) return "";
    string names;

    switch (type)
    {
        case INORDER: _copyInorder(_root, names); break;
        case PREORDER: _copyPreorder(_root, names); break;
        case POSTORDER: _copyPostorder(_root, names); break;
        case LEVELORDER: _copyLevelorder(_root, names); break;
        default: return "";
    }

    names.pop_back(); // removes the last space
    names.pop_back(); // removes the last comma
    return names;
}

/**
 *  @brief  Get the number of levels from root to most distant leaf of @c this tree
 *  @return highest level of @c this tree
 *  @complexity O(n) (worst-case)
 */
int AVLTree::levelCount()
{
    if (!_root) return 0;

    queue<Node*> nodes_queue;
    nodes_queue.push(_root);

    int nodes_counted = 0;
    int nodes_expected = 1;

    int level_count = 0;

    while (!nodes_queue.empty())
    {
        Node* current = nodes_queue.front();
        nodes_queue.pop();

        Node* left = current->left;
```

```
        Node* right = current-> right;

        if (left) { nodes_queue.push(left); nodes_counted++; }
        if (right) { nodes_queue.push(right); nodes_counted++; }

        if (!(--nodes_expected))
        {
            nodes_expected = nodes_counted;
            nodes_counted = 0;

            level_count++;
        }
    }
    return level_count;
}

/**
 *  @brief  Create, with ID and name, and insert a @c Node to @c this tree
 *  @param  id  integer ID
 *  @param  name full name
 *  @return boolean indicating whether the node was inserted successfully
 */
bool AVLTree::insert(const int id, const string& name)
{
    if (!search(id).empty()) return false;
    Node* root = _insert(_root, id, name);
    if (root != nullptr) _root = root;
    return root != nullptr;
}

/**
 *  @brief  Identify, by ID, and remove a @c Node from @c this tree
 *  @param  id integer ID
 *  @return boolean indicating whether a node was removed
 */
bool AVLTree::remove(const int id)
{
    bool is_leaf = (!_root->left && !_root->right);
    bool is_removed = _remove(_root, nullptr, id);
    if (is_leaf && is_removed) { _root = nullptr; }
    return is_removed;
}

/**
 *  @brief  Identify, by inorder count, and remove a @c Node from @c this tree
 *  @param  id inorder count
 *  @return boolean indicating whether a node was removed
 */
bool AVLTree::removeInorder(int count)
{
    bool is_leaf = (!_root->left && !_root->right);
    bool is_removed = _removeInorder(_root, nullptr, &count);
    if (is_leaf && is_removed) { _root = nullptr; }
    return is_removed;
}

/**
 *  @brief  Search for an ID from the @c Nodes of @c this tree
 *  @param  id integer ID
 *  @return name corresponding to the found ID
```

```
 */
string AVLTree::search(const int id)
{
    string match;
    _search(_root, id, match);
    return match;
}

/**
 *  @brief  Search for a name from the @c Nodes of @c this tree
 *  @param  root  pointer to the root @c Node of the tree
 *  @param  name  full name
 *  @param  matches vector to which list of IDs is to be copied
 */
vector<string> AVLTree::search(const string& name)
{
    vector<string> matches;
    _search(_root, name, matches);
    return matches;
}

#endif //AVLTREE_H
```