```cpp
#include "../src/AVL_Interface.h"
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <set>

using namespace std;


/*
    To check output (At the Project1 directory):
        g++ -std=c++14 -Werror -Wuninitialized -o build/test test-unit/test.cpp src/AVL_Interface.cpp src/AVL_Tree.cpp
        ./build/test
*/


/*
!!!!!Notes!!!!!
Change private to public on line 7 in AVL_Interface.h to test interface specific functions (please change back when complete)
Add -DDBUG=1 to compile command to supress successful/unsuccessful printouts

*/


std::string IDToName(const std::string& ID) {
    // Testing function to convert ID's to names to validate correctness of traversals
    std::string name;
    for (const auto& num: ID) {
        name += 17 + (int)num;  //ASCII 'A' = 65, numbers start at 48
    }
    return name;
}

std::string NameToID(const std::string& name) {
    // Testing function to convert names ID's to validate correctness of removals
    std::string ID;
    for (const auto& num: name) {
        ID += (int)num - 17;    //ASCII 'A' = 65, numbers start at 48
    }
    return ID;
}

std::string getRandomID(const std::string& seed) {
    // Generate random IDs for large tests
    srand(stoi(seed));
    std::string ID;
    for (size_t i{}; i < 8; ++i) {
        ID += std::to_string(rand() % 10);
    }
    return ID;
}


TEST_CASE("BST Basic Insert", "[tree]"){
    AVL_Tree tree;
    vector<std::string> names;
    std::string ID = "12345678";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);

    sort(names.begin(), names.end());
    REQUIRE(names == tree.InOrderTraversal());
}
```

```
TEST_CASE("BST Small Insert", "[tree]"){
    // Test case #5
    AVL_Tree tree;
    vector<std::string> names;
    std::string ID = "50005000";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "35354334";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "76543210";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "56567342";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "83711221";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "17449900";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);

    sort(names.begin(), names.end());
    REQUIRE(names == tree.InOrderTraversal());
}

TEST_CASE("BST Small Inorder Insert", "[tree]"){
    // Test case #5
    AVL_Tree tree;
    vector<std::string> names;
    std::string ID = "17449900";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "35354334";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "50005000";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "56567342";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "76543210";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "83711221";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);

    sort(names.begin(), names.end());
    REQUIRE(names == tree.InOrderTraversal());
}
```

```
TEST_CASE("BST Small Reverse Insert", "[tree]"){
    // Test case #5
    AVL_Tree tree;
    vector<std::string> names;
    std::string ID = "83711221";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "76543210";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "56567342";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "50005000";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "35354334";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "17449900";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);

    sort(names.begin(), names.end());
    REQUIRE(names == tree.InOrderTraversal());
}

TEST_CASE("BST Pre and PostOrder Traversal", "[tree]"){
    // Test Pre and post order traversals
    AVL_Tree tree;
    std::string ID = "00000000";
    tree.insert(IDToName(ID), ID);
    ID = "11111111";
    tree.insert(IDToName(ID), ID);
    ID = "22222222";
    tree.insert(IDToName(ID), ID);

    std::vector<string> names = {"BBBBBBBB", "AAAAAAAA", "CCCCCCCC"};
    REQUIRE(names == tree.PreOrderTraversal());
    names = {"AAAAAAAA", "CCCCCCCC", "BBBBBBBB"};
    REQUIRE(names == tree.PostOrderTraversal());
}
```

```
TEST_CASE("BST Removal", "[tree]"){
    // Test case #4
    AVL_Tree tree;
    vector<std::string> names;
    std::string ID = "45674567";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "35455565";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "87878787";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);
    ID = "95462138";
    names.push_back(IDToName(ID));
    tree.insert(IDToName(ID), ID);

    sort(names.begin(), names.end());
    REQUIRE(names == tree.InOrderTraversal());

    tree.remove("45674567");
    names.erase(names.begin() + 1);
    REQUIRE(names == tree.InOrderTraversal());

    tree.removeInOrder(2);
    names.erase(names.begin() + 2);
    REQUIRE(names == tree.InOrderTraversal());
}

TEST_CASE("BST Large Insert", "[tree]"){
    // The penultimate insertion test
    AVL_Tree tree;
    set<std::string> names;
    vector<std::string> names_vec;
    std::string ID = "12345678";

    for (size_t i{1}; i <= 1'000; ++i) {
        ID = getRandomID(ID);
        names.insert(IDToName(ID));
        tree.insert(IDToName(ID), ID);

        if (i % 5 == 0) {
            names_vec.clear();
            names_vec.reserve(names.size());
            names_vec.assign(names.begin(), names.end());
            REQUIRE(names_vec == tree.InOrderTraversal());
        }
    }
}
```

```
TEST_CASE("BST Large Insert with Removals", "[tree]"){
    // The penultimate test
    AVL_Tree tree;
    set<std::string> names;
    auto names_it = names.begin();
    vector<std::string> names_vec;
    std::string ID = "12345678";
    int index_to_remove{};

    for (size_t i{1}; i <= 1'000; ++i) {
        ID = getRandomID(ID);
        names.insert(IDToName(ID));
        tree.insert(IDToName(ID), ID);

        if (i % 10 == 0) {
            for (size_t j{}; j < 2; ++j) {
                index_to_remove = rand() % names.size();
                names_it = names.begin();
                std::advance(names_it, index_to_remove);
                tree.remove(NameToID(*names_it));
                names.erase(names_it);
            }
            for (size_t j{}; j < 2; ++j) {
                index_to_remove = rand() % names.size();
                names_it = names.begin();
                std::advance(names_it, index_to_remove);
                tree.removeInOrder(index_to_remove);
                names.erase(names_it);
            }
            names_vec.clear();
            names_vec.reserve(names.size());
            names_vec.assign(names.begin(), names.end());
            REQUIRE(names_vec == tree.InOrderTraversal());
        }
    }
}
```

```
TEST_CASE("BST Largest Insert with Removals", "[tree]"){
    // The ultimate test
    AVL_Tree tree;
    set<std::string> names;
    auto names_it = names.begin();
    vector<std::string> names_vec;
    std::string ID = "12345678";
    int index_to_remove{};

    for (size_t i{1}; i <= 10'000; ++i) {
        ID = getRandomID(ID);
        names.insert(IDToName(ID));
        tree.insert(IDToName(ID), ID);

        if (i % 20 == 0) {
            for (size_t j{}; j < 4; ++j) {
                index_to_remove = rand() % names.size();
                names_it = names.begin();
                std::advance(names_it, index_to_remove);
                tree.remove(NameToID(*names_it));
                names.erase(names_it);
            }
            for (size_t j{}; j < 4; ++j) {
                index_to_remove = rand() % names.size();
                names_it = names.begin();
                std::advance(names_it, index_to_remove);
                tree.removeInOrder(index_to_remove);
                names.erase(names_it);
            }
            names_vec.clear();
            names_vec.reserve(names.size());
            names_vec.assign(names.begin(), names.end());
            REQUIRE(names_vec == tree.InOrderTraversal());
        }
    }
}
```

```
TEST_CASE("BST Largest Insert with Extra Removals", "[tree]"){
    // The ultimate test with extra removals
    AVL_Tree tree;
    set<std::string> names;
    auto names_it = names.begin();
    vector<std::string> names_vec;
    std::string ID = "12345678";
    int index_to_remove{};

    for (size_t i{1}; i <= 10'000; ++i) {
        ID = getRandomID(ID);
        names.insert(IDToName(ID));
        tree.insert(IDToName(ID), ID);

        if (i % 20 == 0) {
            for (size_t j{}; j < 8; ++j) {
                index_to_remove = rand() % names.size();
                names_it = names.begin();
                std::advance(names_it, index_to_remove);
                tree.remove(NameToID(*names_it));
                names.erase(names_it);
            }
            for (size_t j{}; j < 8; ++j) {
                index_to_remove = rand() % names.size();
                names_it = names.begin();
                std::advance(names_it, index_to_remove);
                tree.removeInOrder(index_to_remove);
                names.erase(names_it);
            }
            names_vec.clear();
            names_vec.reserve(names.size());
            names_vec.assign(names.begin(), names.end());
            REQUIRE(names_vec == tree.InOrderTraversal());
        }
    }
}
```

```
TEST_CASE("BST Possibly Excessive Insert and Removal", "[tree]"){
    // Excessively large test, just because
    AVL_Tree tree;
    set<std::string> names;
    auto names_it = names.begin();
    vector<std::string> names_vec;
    std::string ID = "12345678";
    int index_to_remove{};

    for (size_t i{1}; i <= 1'000'000; ++i) {
        ID = getRandomID(ID);
        names.insert(IDToName(ID));
        tree.insert(IDToName(ID), ID);

        if (i % 50 == 0) {
            for (size_t j{}; j < 15; ++j) {
                index_to_remove = rand() % names.size();
                names_it = names.begin();
                std::advance(names_it, index_to_remove);
                tree.remove(NameToID(*names_it));
                names.erase(names_it);
            }
            for (size_t j{}; j < 15; ++j) {
                index_to_remove = rand() % names.size();
                names_it = names.begin();
                std::advance(names_it, index_to_remove);
                tree.removeInOrder(index_to_remove);
                names.erase(names_it);
            }
            names_vec.clear();
            names_vec.reserve(names.size());
            names_vec.assign(names.begin(), names.end());
            REQUIRE(names_vec == tree.InOrderTraversal());
        }
    }
}
```

```
TEST_CASE("Interface Valid Name", "[interface]"){
    AVL_Interface interface;
    // Name is required to be enclosed by "". Only alphabetic characters and spaces allowed. Empty strings are invalid
    REQUIRE(interface.isValidName("\"Bob the Builder\"") == true);
    REQUIRE(interface.isValidName("\"Geralt of Rivia\"") == true);
    REQUIRE(interface.isValidName("\"Dave\"") == true);
    REQUIRE(interface.isValidName("") == false);                        //Empty
    REQUIRE(interface.isValidName("\"Bob the Builder1\"") == false);    // Non-alphabetic char
    REQUIRE(interface.isValidName("\"Bill_Nye\"") == false);            // _ instead of space
    REQUIRE(interface.isValidName("\"\"") == false);                    // Empty
    REQUIRE(interface.isValidName("Bob the Builder") == false);         // Not enclosed by ""
    REQUIRE(interface.isValidName("Geralt of Rivia") == false);         // Not enclosed by ""
}

TEST_CASE("Interface Valid ID", "[interface]"){
    AVL_Interface interface;
    // ID must be 8 digits longs, numbers only
    REQUIRE(interface.isValidID("12345678") == true);
    REQUIRE(interface.isValidID("88888888") == true);
    REQUIRE(interface.isValidID("") == false);              // Empty
    REQUIRE(interface.isValidID("1234") == false);          // Too short
    REQUIRE(interface.isValidID("Bill_Nye") == false);      // Not numbers
    REQUIRE(interface.isValidID(" ") == false);             // Just a space
}

TEST_CASE("Interface Valid Insert Command", "[interface]"){
    AVL_Interface interface;
    // Checks tokenized input. First entry must be "insert", second entry must be valid name, third entry must be valid ID
    vector<string> commands = {"insert", "\"Dave\"", "12345678"};
    REQUIRE(interface.isValidInsert(commands) == true);
    commands[1] = "\"Geralt of Rivia\"";
    commands[2] = "88888888";
    REQUIRE(interface.isValidInsert(commands) == true);
    commands[1] = "Dave";
    REQUIRE(interface.isValidInsert(commands) == false);    // Invalid name
    commands[1] = "\"Geralt of Rivia\"";
    commands[2] = "8888";
    REQUIRE(interface.isValidInsert(commands) == false);    // Invlaid ID
    commands[2] = "88888888";
    commands[0] = "somethingThatsNotInsert";
    REQUIRE(interface.isValidInsert(commands) == false);    // Invlaid command
    commands[0] = "";
    REQUIRE(interface.isValidInsert(commands) == false);    // Invalid command
    commands[0] = "insert";
    commands.push_back("12345678");
    REQUIRE(interface.isValidInsert(commands) == false);    // Invalid vector length
    commands.pop_back();
    commands.pop_back();
    REQUIRE(interface.isValidInsert(commands) == false);    // Invalid vector length
    commands.clear();
    REQUIRE(interface.isValidInsert(commands) == false);    // Empty vector
}
```

```
TEST_CASE("Interface Valid Remove Command", "[interface]"){
    AVL_Interface interface;
    // Checks tokenized input. First entry must be "remove", second entry must be valid ID
    vector<string> commands = {"remove", "12345678"};
    REQUIRE(interface.isValidRemove(commands) == true);
    commands[1] = "88888888";
    REQUIRE(interface.isValidRemove(commands) == true);
    commands[1] = "1234";
    REQUIRE(interface.isValidRemove(commands) == false);    // Invalid ID
    commands[1] = "";
    REQUIRE(interface.isValidRemove(commands) == false);    // Empty ID
    commands[1] = "88888888";
    commands[0] = "somethingThatsNotRemove";
    REQUIRE(interface.isValidRemove(commands) == false);    // Invalid command
    commands[0] = "";
    REQUIRE(interface.isValidRemove(commands) == false);    // Invalid command
    commands[0] = "remove";
    commands.push_back("12345678");
    REQUIRE(interface.isValidRemove(commands) == false);    // Invalid vector length
    commands.pop_back();
    commands.pop_back();
    REQUIRE(interface.isValidRemove(commands) == false);    // Invalid vector length
    commands.clear();
    REQUIRE(interface.isValidRemove(commands) == false);    // Empty vector
}

TEST_CASE("Interface Valid Search Command", "[interface]"){
    AVL_Interface interface;
    // Checks tokenized input. First entry must be "search", second entry must be valid ID or name
    vector<string> commands = {"search", "12345678"};
    REQUIRE(interface.isValidSearch(commands) == true);
    commands[1] = "88888888";
    REQUIRE(interface.isValidSearch(commands) == true);
    commands[1] = "\"Dave\"";
    REQUIRE(interface.isValidSearch(commands) == true);
    commands[1] = "\"Geralt of Rivea\"";
    REQUIRE(interface.isValidSearch(commands) == true);
    commands[1] = "1234";
    REQUIRE(interface.isValidSearch(commands) == false);    // Invalid ID
    commands[1] = "Bob";
    REQUIRE(interface.isValidSearch(commands) == false);    // Invalid Name
    commands[1] = "";
    REQUIRE(interface.isValidSearch(commands) == false);    // Empty ID
    commands[1] = "88888888";
    commands[0] = "somethingThatsNotSearch";
    REQUIRE(interface.isValidSearch(commands) == false);    // Invalid command
    commands[0] = "";
    REQUIRE(interface.isValidSearch(commands) == false);    // Invalid command
    commands[0] = "search";
    commands.push_back("12345678");
    REQUIRE(interface.isValidSearch(commands) == false);    // Invalid vector length
    commands.pop_back();
    commands.pop_back();
    REQUIRE(interface.isValidSearch(commands) == false);    // Invalid vector length
    commands.clear();
    REQUIRE(interface.isValidSearch(commands) == false);    // Empty vector
}
```

```
TEST_CASE("Interface Valid Remove Nth Command", "[interface]"){
    AVL_Interface interface;
    // Checks tokenized input. First entry must be "removeInorder", second entry must be valid integer
    vector<string> commands = {"removeInorder", "1"};
    REQUIRE(interface.isValidRemoveNth(commands) == true);
    commands[1] = "88888888";
    REQUIRE(interface.isValidRemoveNth(commands) == true);
    commands[1] = "0";
    REQUIRE(interface.isValidRemoveNth(commands) == true);
    commands[1] = "";
    REQUIRE(interface.isValidRemoveNth(commands) == false); // Empty int
    commands[1] = "88888888";
    commands[0] = "somethingThatsNotRemoveNth";
    REQUIRE(interface.isValidRemoveNth(commands) == false); // Invalid command
    commands[0] = "";
    REQUIRE(interface.isValidRemoveNth(commands) == false); // Invalid command
    commands[0] = "removeInorder";
    commands.push_back("5");
    REQUIRE(interface.isValidRemoveNth(commands) == false); // Invalid vector length
    commands.pop_back();
    commands.pop_back();
    REQUIRE(interface.isValidRemoveNth(commands) == false); // Invalid vector length
    commands.clear();
    REQUIRE(interface.isValidRemoveNth(commands) == false); // Empty vector
}
```