```cpp
#include "main.cpp"

#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <sstream>
#include <string>

TEST_CASE("Test PageRank with a small graph") {
    std::vector<std::vector<int>> graph = {
            {1, 2},
            {2},
            {},
            {0, 2}
    };
    std::vector<double> expected_scores = {0.25, 0.25, 0.25, 0.25};

    std::vector<double> scores = calculate_pagerank(graph, 10);
    REQUIRE(scores.size() == expected_scores.size());

    for (size_t i = 0; i < scores.size(); ++i) {
        REQUIRE(scores[i] == Approx(expected_scores[i]).epsilon(0.01));
    }
}

TEST_CASE("Test PageRank with a larger graph") {
    std::stringstream input;
    input << "A B\nB C\nC D\nD E\nE F\nB E\nF B\nF G\nG F\nG A\n";
    std::vector<std::vector<int>> graph;
    std::map<std::string, int> node_map;
    std::string from, to;
    int num_iterations = 20;

    // Build graph
    while (input >> from >> to) {
        // Assign unique integer id to each node
        if (node_map.count(from) == 0) {
```

```
37              // Add empty vector for new node
38              node_map[from] = node_map.size();
39              graph.emplace_back();
40          }
41          if (node_map.count(to) == 0) {
42              // Add empty vector for new node
43              node_map[to] = node_map.size();
44              graph.emplace_back();
45          }
46
47          // Add edge to graph
48          int from_id = node_map[from], to_id =
    node_map[to];
49          graph[from_id].push_back(to_id);
50      }
51
52      // Calculate PageRank scores
53      std::vector<double> pagerank_scores =
    calculate_pagerank(graph, num_iterations);
54
55      // Create an ordered map of website names and
    PageRank scores
56      std::map<std::string, double> pagerank_map;
57      for (auto mapping : node_map) {
58          std::string website_name = mapping.first;
59          int node_id = mapping.second;
60          pagerank_map[website_name] =
    pagerank_scores[node_id];
61      }
62
63      REQUIRE(pagerank_map.size() == node_map.size
    ());
64      REQUIRE(pagerank_map["A"] == Approx(0.05).
    epsilon(0.01));
65      REQUIRE(pagerank_map["B"] == Approx(0.19).
    epsilon(0.01));
66      REQUIRE(pagerank_map["C"] == Approx(0.13).
    epsilon(0.01));
67      REQUIRE(pagerank_map["D"] == Approx(0.08).
    epsilon(0.01));
68      REQUIRE(pagerank_map["E"] == Approx(0.13).
```

```
68 epsilon(0.01));
69     REQUIRE(pagerank_map["F"] == Approx(0.32).
   epsilon(0.01));
70     REQUIRE(pagerank_map["G"] == Approx(0.11).
   epsilon(0.01));
71 }
72
73 TEST_CASE("pagerank 4-node graph with 3 iterations
   ") {
74     vector<vector<int>> graph = {{1, 2}, {2, 3}, {
   3}, {0}};
75     vector<double> expected_scores = {0.375, 0.25
   , 0.25, 0.125};
76     vector<double> actual_scores =
   calculate_pagerank(graph, 3);
77     for (int i = 0; i < expected_scores.size(); i
   ++) {
78         REQUIRE(actual_scores[i] == Approx(
   expected_scores[i]).epsilon(0.01));
79     }
80 }
81
82 TEST_CASE("Pagerank non-trivial graph") {
83     vector<vector<int>> graph = {{1, 2}, {0, 2}, {
   0, 1}, {2}};
84     int num_iterations = 10;
85     vector<double> pagerank_scores =
   calculate_pagerank(graph, num_iterations);
86     REQUIRE(pagerank_scores.size() == 4);
87     CHECK(pagerank_scores[0] == Approx(0.3229).
   epsilon(0.01));
88     CHECK(pagerank_scores[1] == Approx(0.3229).
   epsilon(0.01));
89     CHECK(pagerank_scores[2] == Approx(0.3542).
   epsilon(0.01));
90     CHECK(pagerank_scores[3] == Approx(0.0000).
   epsilon(0.01));
91 }
92
93 TEST_CASE("Two node graph") {
94     vector<vector<int>> graph {{1}, {}};
```

```
 95     vector<double> expected_pagerank {0.5, 0.5};
 96     vector<double> pagerank = calculate_pagerank(
    graph, 10);
 97     REQUIRE(pagerank.size() == 2);
 98     for (int i = 0; i < pagerank.size(); i++) {
 99         REQUIRE(pagerank[i] == Approx(
    expected_pagerank[i]).epsilon(0.01));
100     }
101 }
102
103
104
105
```