```cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "AdjacencyList.h"


void run_test_case(const vector<pair<string, string>>& edges,
int power_iterations, const map<string, double>&
expected_pagerank) {
    AdjacencyList graph;

    for (const auto& edge : edges) {
        graph.InsertEdge(edge.first, edge.second);
    }

    graph.PageRank(power_iterations);

    const auto& pagerank_map = graph.GetPageRankCon();
    for (const auto& expected : expected_pagerank) {
        const auto& vertex = expected.first;
        double expected_rank = expected.second;

        // Retrieve the actual rank from the pagerank_map
        double actual_rank = pagerank_map.at(vertex).first;

        REQUIRE(Approx(expected_rank) == actual_rank);
    }
}

TEST_CASE("TEST 1: Graph construction") {
    AdjacencyList graph;
    graph.InsertEdge("A", "B");
    graph.InsertEdge("B", "C");
    graph.InsertEdge("C", "D");

    REQUIRE(graph.Outdegree("A") == 1);
    REQUIRE(graph.Outdegree("B") == 1);
    REQUIRE(graph.Outdegree("C") == 1);
    REQUIRE(graph.Outdegree("D") == 0);
}

TEST_CASE("TEST 2: n = 100, p = 100, webpages = 100") {
    AdjacencyList graph;

        for (int i = 0; i < 100; i++) {
            string from = "A" + to_string(i);
```

```
            string to = "A" + to_string((i + 1) % 100);
            graph.InsertEdge(from, to);
        }
        graph.PageRank(100); // Run PageRank for 100 iterations

        // Test the PageRank values here
        const auto& page_rank_con = graph.GetPageRankCon();

        // Check if the values are uniform
        double uniform_value = 1.0 / 100;
        for (const auto& pr : page_rank_con) {
            REQUIRE(pr.second.first == Approx(uniform_value));
        }
}

TEST_CASE("TEST 3: Simple graph with 5 vertices") {
    AdjacencyList graph;
graph.InsertEdge("A", "B");
graph.InsertEdge("B", "C");
graph.InsertEdge("C", "D");
graph.InsertEdge("D", "E");
graph.InsertEdge("E", "A");
graph.PageRank(100); // Run PageRank for 100 iterations

const auto& page_rank_con = graph.GetPageRankCon();


double uniform_value = 1.0 / 5;
for (const auto& pr : page_rank_con) {
REQUIRE(pr.second.first ==
Approx(uniform_value).epsilon(0.001));
}
}


TEST_CASE("TEST 4: 10,000 power_iterations") {
    AdjacencyList graph;

    // Add vertices and edges to the graph
    graph.InsertEdge("A", "B");
    graph.InsertEdge("A", "C");
    graph.InsertEdge("B", "A");
    graph.InsertEdge("C", "A");
    graph.InsertEdge("C", "B");

    // Run the PageRank algorithm for 10,000 iterations
```

```
    graph.PageRank(10000);

    // Store the PageRank score for each vertex in a map
    std::map<std::string, double> pageRankMap;
    for (const auto& iter : graph.GetPageRankCon()) {
        pageRankMap[iter.first] = iter.second.first;
    }

    // Check the correctness of the algorithm output
    REQUIRE(pageRankMap["A"] == Approx(  0.4444444444 ));
    REQUIRE(pageRankMap["B"] == Approx(  0.3333333333 ));
    REQUIRE(pageRankMap["C"] == Approx(  0.2222222222 ));
}

TEST_CASE("TEST 5:PI = 1, V = 2") {
    AdjacencyList graph;

    // Add vertices and edges to the graph
    graph.InsertEdge("google.com", "gmail.com");
    graph.InsertEdge("gmail.com", "google.com" );

    // Run the PageRank algorithm for 10 iterations
    graph.PageRank(2);

    // Store the PageRank score for each vertex in a map
    std::map<std::string, double> pageRankMap;
    for (const auto& iter : graph.GetPageRankCon()) {
        pageRankMap[iter.first] = iter.second.first;
    }

    // Check the correctness of the algorithm output
    REQUIRE(pageRankMap["google.com"] ==
Approx(0.50).epsilon(0.01));
    REQUIRE(pageRankMap["gmail.com"] ==
Approx(0.50).epsilon(0.01));

}
```