```cpp
#include <iostream>
#include <chrono>
#include <math.h>
#include <random>
#include <algorithm>

#include <vector>
#include <unordered_map>

//  Lets look at searching lists vs vectors vs maps

/*  Each function needs to be modified to return the time it took to
  complete
*   the neccessary searches. Do not include setup in the testing time.
    */
class Node {
private:
    Node* _next;
    int _data;

public:
    Node(int data) : _next(nullptr), _data(data) {}

    void setNext(Node* next) { _next = next; }
    void setData(int data) { _data = data; }
    Node* getNext() { return _next; }
    int getData() { return _data; }
};

class LinkedList {
private:
    int _size;
    Node* _head;

public:
    LinkedList() : _size(1), _head(new Node(_size)) {};
    ~LinkedList() {
        reset();
        delete _head;
    }

    void addNode() {
        _size++;
        Node* traversal_ptr = _head;
        Node* new_node = new Node(_size);

        while (traversal_ptr->getNext() != nullptr) {
            traversal_ptr = traversal_ptr->getNext();
        }
```

```cpp

            traversal_ptr->setNext(new_node);
        }

        // This is the function which needs to be timed.
        std::chrono::duration<double> searchNodeByValue(int value) {
            int idx = 0;
            int search_value = value + 1;

            auto start = std::chrono::steady_clock::now();
            Node* traversal_ptr = _head;

            while (traversal_ptr->getNext() != nullptr) {
                if (traversal_ptr->getData() == search_value) {
                    auto end = std::chrono::steady_clock::now();
                    return end - start;
                }

                idx++;
                traversal_ptr = traversal_ptr->getNext();
            }

            if (traversal_ptr->getNext() == nullptr &&
                traversal_ptr->getData() == search_value) {
                auto end = std::chrono::steady_clock::now();
                return end - start;
            }

            auto bad_end = std::chrono::steady_clock::now();
            return start - bad_end; // Neg value for err
        }

        void reset() {
            Node* traversal_ptr = _head;
            Node* current_ptr = nullptr;

            while (traversal_ptr->getNext() != nullptr) {
                current_ptr = traversal_ptr;
                traversal_ptr = traversal_ptr->getNext();
                delete current_ptr;
            }

            _head = traversal_ptr;
            _size = 1;
            _head->setData(_size);
        }
    };

    // Takes the position starting from 0
```

```cpp
std::chrono::duration<double> testVectorSearch(int value, int value_pos, int vec_size) {
    int search_value = value;
    int search_value_pos = value_pos;
    int vector_size = vec_size;

    std::vector<int> integer_vector;
    integer_vector.reserve(vector_size);

    for (int idx = 0; idx < vector_size; idx++) {
        integer_vector.push_back(0);
        if (idx == search_value_pos) { integer_vector.push_back(value); }
    }

    //Time below here
    auto start = std::chrono::steady_clock::now();
    auto integer_it = std::find(integer_vector.begin(), integer_vector.end(), search_value);
    auto end = std::chrono::steady_clock::now();

    if (integer_it != integer_vector.end()) {
        return end - start;
    }

    return start - end; // negative numbers means i know there's an err
}

std::chrono::duration<double> testMapSearch(std::string key, int value, int value_pos, int map_size) {
    // Really quick and dirty map setup
    std::string search_key = key;
    int search_value = value;
    int search_value_pos = value_pos;
    int umap_size = map_size;

    std::unordered_map<std::string, int> str_int_umap;

    for (int idx = 0; idx < umap_size; idx++) {
        str_int_umap.insert({ "Default", 0 });
        if (idx == value) { str_int_umap.insert({search_key, search_value}); }
    }

    //Time below here
    auto start = std::chrono::steady_clock::now();
    int found_value = str_int_umap[search_key];
    auto end = std::chrono::steady_clock::now();

    return end - start;
```

```cpp
}

std::vector<std::chrono::duration<double>> singleSearchStructureTest(LinkedList* list, int search_value_pos, int search_range, bool show_results) {
    LinkedList* linked_list = list;
    int value_pos = search_value_pos;
    int range = search_range;
    bool display_results = show_results;

    std::vector<std::chrono::duration<double>> test_time_data;
    double map_ns, vec_ns, list_ns;

    // Start at 1 cuz linked linked constructor says so
    for (int idx = 1; idx <= range; idx++) { linked_list->addNode(); }

    std::chrono::duration<double> list_search_time = linked_list->searchNodeByValue(value_pos);
    std::chrono::duration<double> vec_search_time = testVectorSearch(1, value_pos, range);
    std::chrono::duration<double> map_search_time = testMapSearch("Target", 1, value_pos, range);

    map_ns = map_search_time.count() * pow(10, 9);
    vec_ns = vec_search_time.count() * pow(10, 9);
    list_ns = list_search_time.count() * pow(10, 9);

    test_time_data.push_back(map_search_time);
    test_time_data.push_back(vec_search_time);
    test_time_data.push_back(list_search_time);

    if (display_results) {
        std::cout << std::endl;
        std::cout << "-------------------------------------------------------------" << std::endl;
        std::cout << "          Single Test Results                     " << std::endl;
        std::cout << "-------------------------------------------------------------" << std::endl;
        std::cout << "This test iterated " << value_pos + 1 << " times through structures containing " << range << " elements." << std::endl;
        std::cout << ">> Unordered Map search time:\t" << map_search_time.count()
            << " == " << map_ns << " ns" << std::endl;
        std::cout << ">>        Vector search time:\t" << vec_search_time.count()
```

```cpp
            << " == " << vec_ns << " ns" << std::endl;
        std::cout << ">>          List search time:\t" << list_search_time.count()
            << " == " << list_ns << " ns" << std::endl;
    }

    return test_time_data;
}

void manyRandomValueTest(LinkedList* list, int search_reps, int max_range, bool show_results) {
    LinkedList* linked_list = list;
    std::vector<std::chrono::duration<double>> single_test_data;
    std::vector<std::vector<std::chrono::duration<double>>> multiple_test_data;
    int repitions = search_reps, maximum_range = max_range;
    bool display_results = show_results;

    int range, value;
    double sum_map = 0.0, sum_vec = 0.0, sum_list = 0.0;

    // Run the test
    for (int search_rep = 0; search_rep < repitions; search_rep++) {
        srand((unsigned)time(NULL) + std::rand()); // Reseed the rando generator
        range = (std::rand() % maximum_range);
        value = std::rand() % range;

        single_test_data = singleSearchStructureTest(linked_list, value, range, false);
        multiple_test_data.push_back(single_test_data);
    }

    // Calculate the results
    for (int idx = 0; idx < repitions; idx++) {
        sum_map += multiple_test_data[idx][0].count();
        sum_vec += multiple_test_data[idx][1].count();
        sum_list += multiple_test_data[idx][2].count();
    }

    double avg_map = sum_map / repitions;
    double avg_vec = sum_vec / repitions;
    double avg_list = sum_list / repitions;
    double avg_map_ns = avg_map * pow(10, 9);
    double avg_vec_ns = avg_vec * pow(10, 9);
    double avg_list_ns = avg_list * pow(10, 9);

    // Display the lads
    if (display_results) {
        std::cout << std::endl;
        std::cout << "-------------------------------------------------------------" << std::endl;
        std::cout << "          Multiple Test Results                              " << std::endl;
        std::cout << "-------------------------------------------------------------" << std::endl;
        std::cout << "Over " << repitions << " repitions, the average time recorded is: " << std::endl;
        std::cout << ">> Unordered Map avg search time:\t" << avg_map
            << " == " << avg_map_ns << " ns" << std::endl;
        std::cout << ">>        Vector avg search time:\t" << avg_vec
            << " == " << avg_vec_ns << " ns" << std::endl;
        std::cout << ">>          List avg search time:\t" << avg_list
            << " == " << avg_list_ns << " ns" << std::endl << std::endl;
    }
}

// Will search through more and more data
void singleRampUpTests(LinkedList* list, int max_ramps, bool show_results) {
    std::vector<std::vector<std::chrono::duration<double>>> ramp_test_data;
    LinkedList* linked_list = list;
    int maximum_ramps = max_ramps;
    bool display_results = show_results;

    for (int current_ramp = 1; current_ramp <= maximum_ramps; current_ramp++) {
        srand((unsigned)time(NULL) % std::rand());  // Reseed

        int container_size = pow(10, current_ramp);
        int search_value = container_size - 1;      // Final value

        std::vector<std::chrono::duration<double>> single_ramp_data =
            singleSearchStructureTest(linked_list, search_value, container_size, false);

        ramp_test_data.push_back(single_ramp_data);
    }

    if (display_results) {
        std::cout << std::endl;
        std::cout << "-------------------------------------------------------------" << std::endl;
        std::cout << "      Ramp Test Results                       " <<
        std::endl;
        std::cout << "-------------------------------------------------------------" << std::endl;
        std::cout << "The test ran " << maximum_ramps << " ramps... " << std::endl;

        for (int idx = 1; idx <= maximum_ramps; idx++) {
            double sci_map = ramp_test_data[idx - 1][0].count(), sci_vec = ramp_test_data[idx - 1][1].count(),
                sci_list = ramp_test_data[idx - 1][2].count(),
                ns_map = sci_map * pow(10, 9), ns_vec = sci_vec * pow(10, 9), ns_list = sci_list * pow(10, 9);

            std::cout << std::endl;
            std::cout << "Ramp " << idx << ": Searched " << pow(10, idx) << " elements." << std::endl;
            std::cout << ">> Unordered Map avg search time:\t" << sci_map << "\t== " << ns_map << " ns." << std::endl;
            std::cout << ">>        Vector avg search time:\t" << sci_vec << "\t== " << ns_vec << " ns." << std::endl;
            std::cout << ">>          List avg search time:\t" << sci_list << "\t== " << ns_list << " ns." << std::endl;
        }
    }
}


int main(){
    LinkedList* linked_list = new LinkedList();
    manyRandomValueTest(linked_list, 4, 1000, true);
    linked_list->reset();
    singleRampUpTests(linked_list, 4, true);
    linked_list->reset();
    return 0;
}
```