Programming Skill P1

Due: Friday, December 6

CSC 215: Algorithm Design and Analysis

Objectives

- Demonstrate mastery of skill P1: *Empirically evaluate the efficiency of an algorithm (including comparing multiple algorithms)*
- Implement an algorithm to solve a problem
- Perform Empirical Analysis of the running time of various implementations
- Generate a report describing the results

Background

Given a map containing city names and their populations, how can we find the largest cities that begin with a certain prefix (the largest cities beginning with "New" or with "Hamd")? Depending on the structure of the map (tree vs. hash table), there are different algorithms that we can use. In this assignment, we will empirically analyze the performance of these two data structures at identifying the largest cities that begin with a particular prefix. We have already written the code using a tree-based map (a variation on an AVL tree), while you will implement the version using a hash table (or HashMap).

Ultimately, we are asking the following question:

What is the most efficient data structure to determine the largest cities that begin with the same prefix?

Task

Your task is to do the following:

- Complete the final unimplemented algorithm (MapExperiment.tableSearch) described below to report all cities that start with the same string, sorted by population from lowest to greatest, with ties broken using alphabetical ordering.
    - The provided code includes a verification function to test the accuracy of your implementation by comparing it with the results of the existing MapExperiment.treeSearch method.
- Modify the parameters of the experiment code provided to test the effectiveness of each data structure and algorithm. In your experiment, be sure to:
    - Test both data structures and their corresponding algorithms.

    - For each algorithm, test a varying size of the map (sizes should range from 100 to 100,000; our data set has slightly fewer than 100,000 cities). The starter code uses a smaller range.
        - The starter code increases the size by 1 each iteration, which is impractical and unnecessary for larger sizes
        - Change the for-loop statement so it increases the size by a constant factor – for example, size = size + size/2 (increases by 50% each iteration).
    - For each size, perform multiple repetitions. This is important to smooth off the timing, so use at least 100 repetitions. The starter code only performs 2 repetitions.
    - For each repetition, create a query and find the matching cities while measuring the time to compute this set of cities.
        - To keep it consistent across the methods, the experimental code precomputes a collection of queries to use.
    - Report the average time taken as the size of the map varies. Ideally, save it as a CSV file (for easy importing).
        - Note: The supplied version does this.
- After collecting that data, import it into a spreadsheet and plot the growth of the various functions. Use a log-log scale plot to illustrate.
- You should then generate a short report summarizing the experiment, explaining your findings, and showing the performance graph.
- You should also submit the raw data in a CSV or other spreadsheet format, which allows the instructor to verify the accuracy of the results.

Group Work and Submissions

- You can work on this in groups of your choosing (up to 5 people per team). The members can be from different sections ***as long as they have the same instructor.***
- Only one member of each group should submit.
- The report should indicate **all** of the members of the group.
- Make sure that your submission includes:
    - All of the code and instructions needed to run the experiment.
    - The report as a PDF, including the graph(s) of the running time performances in the report
    - The raw data from the experiment (CSV or similar spreadsheet format)

Report

Although the report is short, it should be presented in a professional manner. Imagine submitting it to the project manager for your company. You should include in the report:

1. A title with the name of the team members,
2. A brief description (one paragraph) of the problem,
3. A brief description of the experiment that was run, which should include the parameters such as range of sizes tested, number of repetitions, the methods tested, and the type of machine used (OS, CPU speed, memory size, bus-speed if available),
4. A graph (or graphs) depicting the results (showing the growth rate of each method), and

5. Your conclusions about each method. Which method performed the best for time? Which would be the best option considering time-space trade-offs. Be sure to justify the conclusions from the evidence of the experiment and using logic.

Scoring

As with the "exams", this assignment is assessing your *mastery* of skill P1. So, it is an all-or-nothing grade. See the attached rubric for details about how grading for this assignment will be done, but to achieve mastery (and hence receive credit) you will need to score a 90% or higher on the rubric. If you score lower, you will be given feedback to revise and resubmit.

Prefix Matching Algorithms

Here are the two data structures and algorithms being tested. The first one is already implemented.

Red-Black Tree. Like an AVL tree, a red-black tree is a self-balancing binary search tree. The tree is organized by city name. We find the first city in alphabetical order that matches our prefix query and carry out an inorder traversal of the tree until we reach a city that does not match the prefix. We then sort all of the matching cities by their population. This gives us the matching cities in order from lowest population to greatest.

Hash Table. Iterate through all of the keys in the hash table, adding any that match our prefix query to a list. Sort that list of matching cities by their population. This gives us the matching cities in order from lowest population to greatest.

Exemplar Report

Take a look at the exemplar report attached with this assignment. This report relates to a different experiment (testing the effectiveness of various growth functions when dynamically resizing an ArrayStack). However, the approach taken is nicely done and illustrates a very thorough professional report.