Comparing Native Java Data Structure Operation Runtimes

A brief experiment by Shah Zafrani

Find the source code at https://github.com/ShahZafrani/6045-assignment-2

Original Assignment Instructions (avaiable here)

- Find two data structure implementations in the Java standard library, which according to the documentation have different running times (for example, Linked list vs Array implementation of lists, or hash table vs tree based implementation of sets or maps)
- Compare at least two operations (insert, search, delete etc), for different n values, and plot the actual time they take. You may want to run the same thing several times and use averages.
- Analyze the results. Some possible interesting questions
    - Do you get different results when you run the same thing several times ? How different ? Why do you think it happens ?
    - Can you derive the constants from the data ? How accurately ?
    - Do the results go with the theory ? How much do they deviate ?
    - Does the actual data matter ? The data types ? (String vs Integer vs ...)
    - What implementation would you use and why

Submit the source code, and a 2-page-ish 'paper' with the plots and analysis. Paper does not have to be formal, I want to see your thinking more than how good you are at word/latex/English

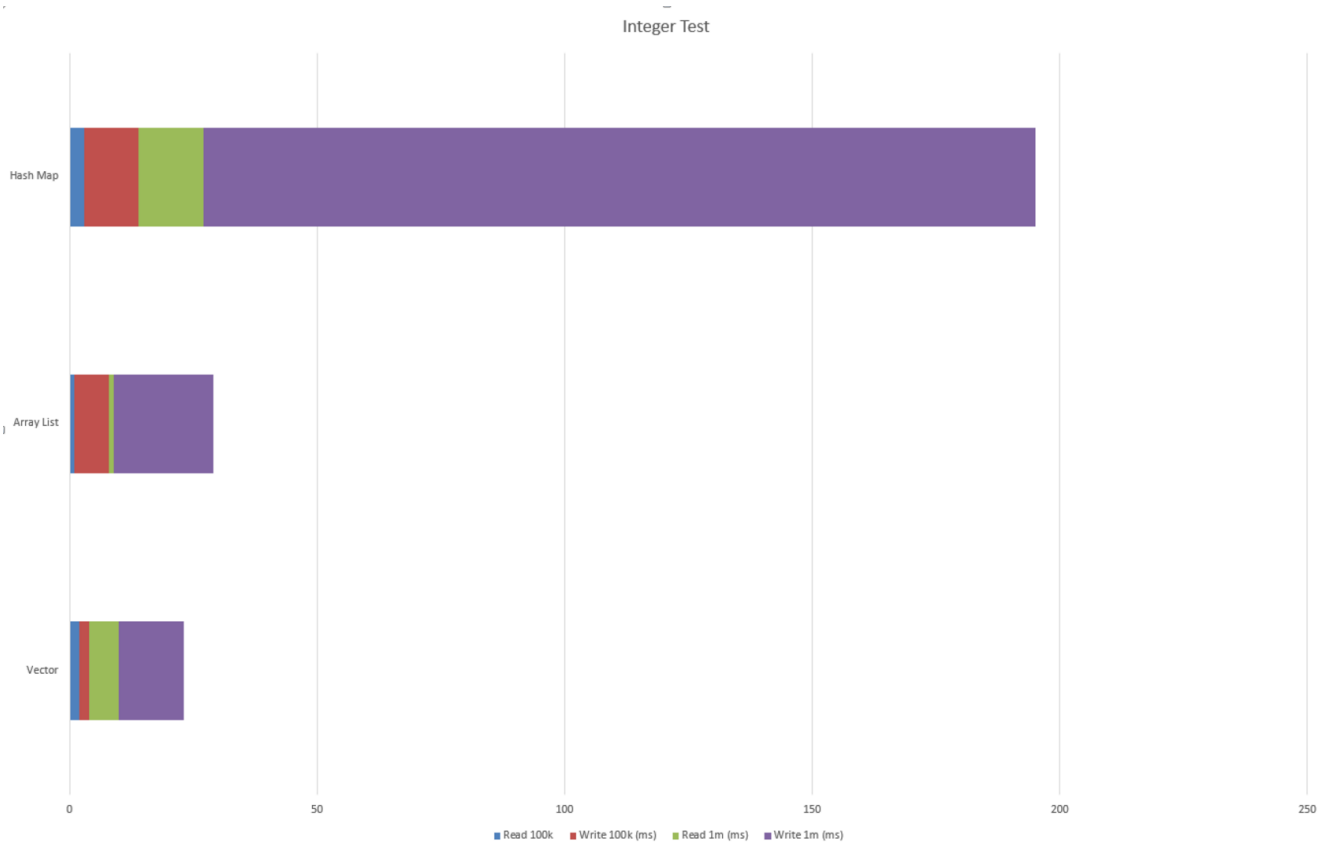
First Impressions and Runtime Assumptions

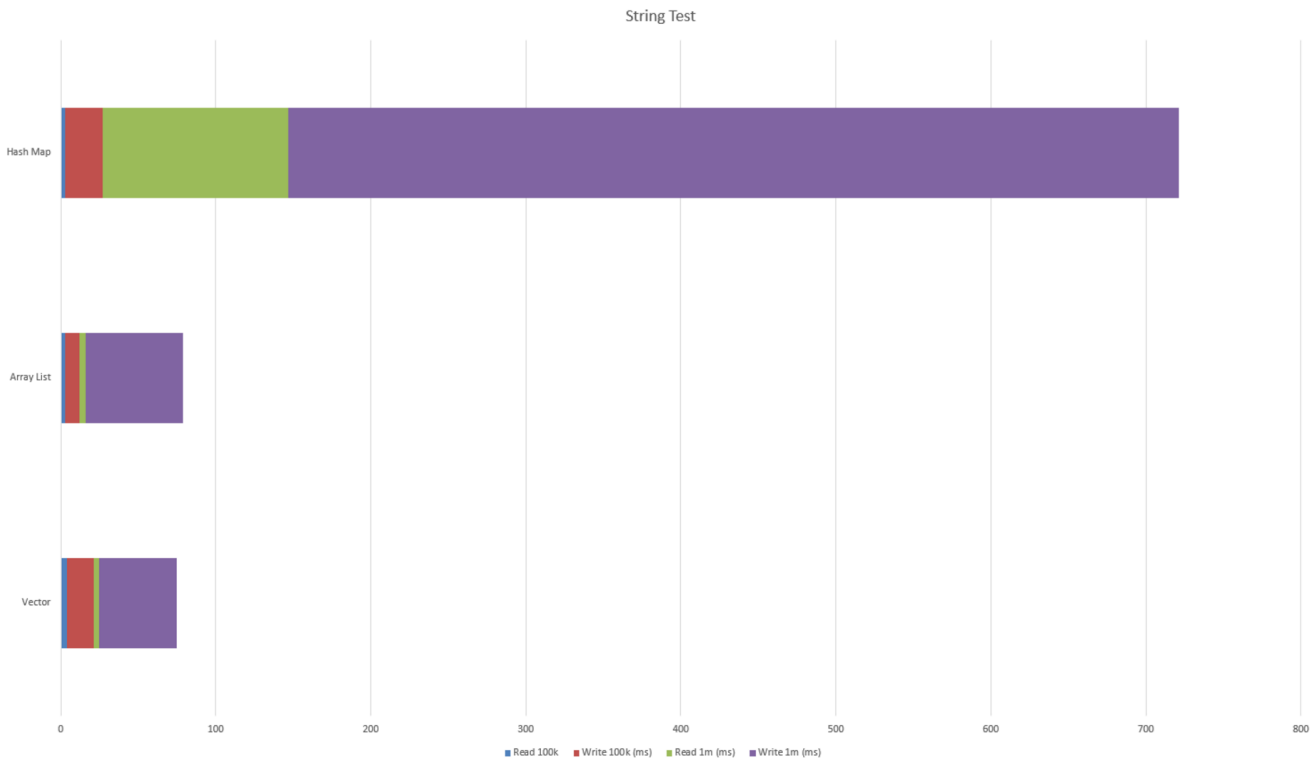
I chose to compare read and write operations using Integers and Strings on 4 different data structures: LinkedList, HashMap, Vector, and ArrayList. When choosing these I assumed that the HashMap would be the fastest for reads and writes, followed by Vector then ArrayList, with LinkedList lagging behind. I didn't have a very scientific reason for expecting HashMap to be the fastest, but have always heard people say it's the fastest. Vectors and ArrayLists should be blazingly fast because they both function as arrays under the hood. LinkedList reads are expected to be slow because of their node based implementation.

Test Implementation

I chose to insert integers and strings in an ordered manner for each data structure. This can lead to data structures that benefit from being ordered performing better than ones that don't. The tests were designed to insert objects (integers and then strings) and then read each one of them. This is done with input sizes of a hundred thousand elements and one million elements. These tests are averaged over the course of ten runs. The code to implement these tests was deceptively easy to write, so I wrote the test once for LinkedList and then copy-pasted it to make the other four test cases while only changing two or three lines. In many scenarios having large amounts of identical code is problematic, here it makes the test code easier to read and work with. I also decided to implement the program as a command line interface with inputs reminiscent of some older DOS programs for aesthetic purposes.

Results





What I Learned

Next time I should output to csv. I should use a more powerful machine. I should test inserting and reading larger objects.