

CS3310, WMU-CS, Spring 2019

HW3: Hashing Based Search Engine

- **Given:** February 21, 2019
- **Due :** March 15, 2019
- **Collaboration: Pairs!**
- **Grading:** Report 10%, Style 10%, Testing 10%, Performance 20%, Functionality 50%
- **Weight: 200 points** (double the weightage of individual weights in HW1 and HW2)

Objectives

1. Understand long, at times somewhat vague, requirements to develop a computing application
2. Design and develop different solutions to a computing problem
3. Experience working with Hash Tables and Hash Maps
4. Understand hashing techniques
5. Experience pair programming
6. Continue to get in-depth knowledge of manipulating arrays and linked lists

In essence, Analyze a complex computing problem and to apply principles of computing and other relevant disciplines to identify high-performing solutions.

Overview

This assignment is a little more open-ended than what you may be used to. We no longer break things down into separate problems below; the entire assignment *is* the problem now. This also means that the “rubric” for the assignment *is* the grading breakdown above, and it *all* applies.

Your task for this assignment is to write a simplified search engine that we call `WMUsearchEngine.java`, subject to the specifications and constraints detailed below. Deciding what data structures and implementations to use, and implementing them as efficiently as possible, will be critical in earning maximum performance points. You are expected to do this by implementing the `Map` interface using various **hash table** techniques

(your choice, see below).

Important: The 20% for Performance will be awarded by evaluating your submissions according to both **time** and **space** required by your search engine. Less time is better, less space is better, and both are considered **separately** for 10% each. Performance points are only awarded for programs that produce correct results!

Partners

You are **allowed (strongly recommended, but not strictly required)** to work in pairs for this assignment. Both partners will receive the same grade. Only one student should submit the actual code zip on Elearning, but **all** students must submit a text note indicating their partner's name and WMU ID. Students who fail to do this risk getting a zero. Make sure that both students names are in the code that is submitted as well. Also, late days apply to both students if the pair submits late.

Paired programming is an excellent technique for working with a partner: you write the code together, taking turns acting as driver (typing) and navigator (reading, watching, guiding). This is way more effective than trying to split up the work and coding separately. Also, you will want to spend some time discussing the assignment and coming up with a game plan before you begin. **START EARLY. Your discussion, analysis and approach decisions must be documented in the README file you submit. You must also document how the code was developed and who contributed to which parts.**

WMU Search Engine

The “good news” is that you get to follow in some very famous footsteps and write an entire search engine! (Well, at least a basic one...) This will require applying the knowledge and techniques you've been learning in CS1110, CS1120 and in this course all semester! In the following, instructions mention Java but if you are using other high-level language, make appropriate changes to file extensions etc.

Your `WMUsearchEngine.java` program must have a single command-line argument which is the name of a file containing a list of websites and the keywords that are contained on each site. You'll want to use this file to construct an index that maps each keyword to a collection of URLs in which it appears. Since reading in this file and constructing the index will take significant time, your program should output “Index Created” followed by a newline once that operation completes.

The file will be structured to have the URL for each site on one line, followed by a single line containing words found on that site, separated by whitespace. Note, this could be more than one space character, or a tab! There are plenty of methods in Java's Scanner that will help you parse these lines. There will be multiple URLs in the file, alternating on each line between the URL and the word list. These files could be many *many* lines long, but a very short example named `urls.txt` might look as follows:

```
http://www.cars.com/
car red truck wheel blue fast brown silver gray
https://en.wikipedia.org/wiki/Cat
cat feline small fur red brown fast gray
http://www.foobar.com/baz
foo bar baz lorem ipsum red blue yellow silver
```

The end-goal of this program is to provide a way for the user to ask for the set of URLs that contain some set of words. The sets of words will be specified as a logical expression. In the example above, if I were to ask for the pages containing "red AND fast", WMUsearchEngine should return the first two URLs, but not the third one, as it does not contain "fast" even though it does contain "red". Similarly, if I were to ask for "car OR baz", the second URL would be omitted, as it contains neither of these words. Note that queries may contain many operands (words) and operators (AND, OR).

To make the input simpler, we will be specifying these queries in *postfix* notation (i.e., operand1 operand2 operator). This will allow you to use a stack to evaluate expressions (rather than expression trees). We will also use *symbols* for the operations rather than words so that the search engine can differentiate words in queries from the operations. The queries above would be given as "red fast &&" and "car baz ||" respectively. The main WMUsearchEngine program will read queries as one word or logical operation at a time from standard input, in response to a > prompt.

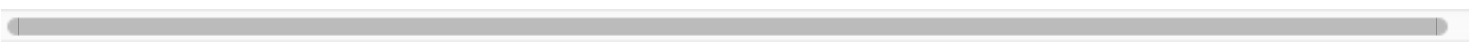
There are two more operations your WMUsearchEngine implementation will need to handle. The first is ?, which requires your program to print the URLs corresponding to the expression at the top of the query stack, **one per line with no additional formatting**. Your program will loop until the quit operation is given, an !. When the user quits, simply exit the program without producing any additional output.

Here is simple sample run that demonstrates all aspects of the program operation, with results based on the short sample input file above.

```
$ java WMUsearchEngine urls.txt
Index Created
> ?
> baz
> red
> ?
http://www.cars.com/
https://en.wikipedia.org/wiki/Cat
http://www.foobar.com/baz
> &&
> ?
http://www.foobar.com/baz
> !
$
```

You will be incorporating time measurements in your `main()` function, alternatively you can also write a script to measure time to see how quickly your `WMUsearchEngine` program works. To do this you'll want to make a file that contains words and commands in response to the prompts that can be used to run the program with standard input redirection. For example, if you put the above input sequence (`? baz red ? && ? !`) in a plain text file called `ops.txt`, you can time and run the program by executing your script called `myTimingScript` as:

```
$ ./myTimingScript java WMUsearchEngine urls.txt <ops.txt >output.txt
```



We will make some small and large URL input files available on Elearning for you to use in testing your application, however, you should test your program with your own test files to cover various test cases. You might want to create a first iteration of this program using a very simple straightforward hash-map implementation.

The Hash Table

You are expected to achieve excellent performance in `WMUsearchEngine.java` by developing and comparing various hashing techniques. Name your best version that is used in the `WMUsearchEngine` program `MyBestHashMap.java`. Obviously your `MyBestHashMap.java` must implement the `Map` interface, but beyond that you have *quite* a few options for how to

proceed:

- Your hash table can use any of the following techniques (but no others!): **separate chaining** or **open addressing**.
- For open addressing you can resolve collisions by **linear probing**, **quadratic probing**, or **double hashing**. (Make sure you understand how the various strategies relate to the size of the bucket array.)

Make sure that you include extensive comments at the start of your `MyBestHashMap` implementation to clarify what type of collision resolution strategy it implements. This should go without saying, but you'll need "all the usual pieces" for your `MyBestHashMap` implementation. In particular:

- Make sure that they are complete and cover **basic** operations (including `toString()` and `CompareTo()` methods) and exception conditions. (Also, if you're still not using base classes for testing against an interface, make **sure** you finally start doing so.)

And that's it. Yes, you're **really** on your own for figuring out what kind of hash tables you should implement, and which one to use in the end and how to effectively do so in order to implement the search engine.

All critical map operations, except `insert`, must run in $O(1)$ *expected* time (or better); `insert` can run in $O(1)$ *amortized* time in case you have to grow the bucket array table to keep the load factor down.

Got Extra Hash Tables?

Depending on just how serious you are about those Performance points, you may well end up writing **multiple** different hash tables over the course of this assignment. However, you have to pick **one** of those to use in `WMUsearchEngine.java`, named `MyBestHashMap.java`. You are welcome to submit other implementations as well, each named accordingly to indicate what type of hash table technique it uses. Include all the benchmarking data, results and analysis that contributed to your final decision on which implementation to use for the search engine.

Iterative Development

You cannot know how fast your `MyBestHashMap` is until it's actually written. You cannot

improve your `MyBestHashMap` until you can tell how fast it is. So the worst mistake you can make is to “think about it” for days without writing any code. (Thinking ahead is good in principle, thinking ahead for too long is the problem here.)

We recommend you start **right now** by writing the simplest `HashMap` you can think of and making that work. For example you could write one based on separate chaining but with a fixed array size.

You want your test cases and benchmarks in place before you keep going. Make sure that your test cases are complete and that your benchmarks tell you how well the various `Map` operations work for that first version of `HashMap`. You should probably save a backup (or even submit early!) as soon as you get done with the first round.

From then on, it’s “try to improve things” followed by “see if the tests still pass” followed by “benchmark to see if things *actually* got better” followed by either “Woops, that was a bad idea, let’s undo that” or “Yay, I made progress, let’s save a backup of the new version” and so on and so forth. We predict that there will be a correlation between how well you do and how often you “went around” this iterative development cycle.

What Classes Are Allowed?

You may **not** use `java.util.HashMap` or `java.util.LinkedHashMap` to implement your `HashMap`s and you may **not** use those classes or `java.util.HashSet` in your `WMUsearchEngine` solution either. You also may not use other map implementations (yours or Java’s) as part of your own `MyBestHashMap` classes. However, in order to write the `WMUsearchEngine.java` application you may (and are expected to) reuse other interfaces and classes that have been provided, that you developed for other assignments, or from the Java library.

If in doubt about what is permitted, better to ask the graders and the instructor first! You don’t want to find out minutes before the deadline that you used something that’s not okay :(

What about the README?

You should use your `report` file to explain how you approached this assignment, what you did in what order, how it worked out, how your plans changed, etc. Try to summarize all the different ways you developed, evaluated, and improved your `JHUG1e` application and various `Hashmaps` over time. If you don’t have a story to tell here, you probably didn’t do enough... Include `README` file to explain how to run your program.

Random Hints

- We'll try to give some guidelines on performance expectations for particular `WMUsearchEngine` inputs, but no promises.
- You may want to look into using a profiler if you cannot think of any other ways to improve the performance of your code anymore.
- You should probably keep track of how the performance changed over various versions of your implementations. If you don't do that, you may end up with changes that made things worse but you never noticed since you didn't have the data to check. A source-code-control-system / or a revision-control-system will be very handy to use (even private use of github would be handy).
- Hash tables are implemented with arrays. You may be tempted to use the `ArrayList` class from the Java API, but will likely find it easier (and faster?) in the long run to work directly with standard arrays, resizing manually when necessary. It is okay to have a compiler warning about casting to a generic array type as a result; no other warnings are allowed however.

Deliverables

As usual, follow file naming conventions as you turn in a [zipped](#) (.zip only) archive containing *all* source code, your README file, your analysis report, and any other deliverables required by the assignment. The zip should contain **no derived files** whatsoever (i.e. no .class files, no .html files other than the ones created by Javadoc, etc.), but should allow building **all** derived files. Include a [plain text](#) README file (not README.txt or README.docx or whatnot) that briefly explains what your programs do and contains any other notes you want us to check out before grading.

Obviously it should contain Signed [Plagiarism Declaration](#) and

A brief report (in a pdf file) on your observations of comparing theoretical vs empirically observed time complexities. Note this report will include (a) a brief description of problem statement(s), (b) algorithms descriptions (if these are standard, commonly known algorithms, then just mention their names along with customization to your specific solution(s), otherwise give the pseudo-code of your algorithms, (c) theoretically derived complexities of the algorithms used in your code, (d) table(s) of the observed time complexities, and (e) plots comparing theoretical vs. empirical along with your observations (e.g. do theoretical agree with your implementation, why? Why not?).

Finally, make sure to include your name and email address in every file you turn in

(well, in every file for which it makes sense to do so anyway)!

Grading

For reference, here is a short explanation of the grading criteria; some of the criteria don't apply to all problems, and not all of the criteria are used on all assignments.

Packaging refers to the proper organization of the stuff you hand in, following both the guidelines for Deliverables above as well as the general submission instructions for assignments.

Style refers to Java programming style, including things like consistent indentation, appropriate identifier names, useful comments, suitable javadoc documentation, etc. Style also includes proper modularization of your code (into interfaces, classes, methods, using `public`, `protected`, and `private` appropriately, etc.). Simple, clean, readable code is what you should be aiming for.

Testing refers to proper unit tests for all of the data structure classes you developed for this assignment. Make sure you test **all** (implied) axioms that you can think of and **all** exception conditions that are relevant.

Performance refers to how fast/with how little memory your program can produce the required results compared to other submissions.

Functionality refers to your programs being able to do what they should according to the specification given above; if the specification is ambiguous and you had to make a certain choice, defend that choice in your README file and the report.

If your programs cannot be built you will get no points whatsoever. If your programs cannot be built without warnings using `javac -Xlint:all` we will take off 10% (except if you document a very good reason; no, you cannot use the `@SuppressWarnings` annotation either). If your programs fail miserably even once, i.e. terminate with an exception of any kind, we will take off 10% (however we'll also take those 10% off if you're trying to be "excessively smart" by wrapping your whole program into a universal try-catch).

Thanks to Joanne Selinski and Peter Fröhlich - basic ideas of this assignment comes from them.