Lab 6: Making a Hash of Things

Pre-Lab Notes

Lab submissions will **not be accepted late** – they must be submitted by **midnight the day after they're assigned** (to account for time zone issues). Some latitude might be allowed (I won't disqualify a submission that's mere minutes late!) but that's purely discretionary.

Labs are treated differently from assignments – you're encouraged to work with fellow students, ask questions of the TA during your lab section, and generally collaborate to complete your work. You should still submit your own work, **including citing with whom you collaborated in your submission**, but using answers developed by others will not be treated as plagiarism.

Nevertheless, you are still encouraged to make a good faith effort to complete your labwork, in order to exercise your understanding of the topics it covers. Lab submissions are marked purely 1 or 0 by whether the TA believes you at least made an honest attempt, even if you didn't succeed, so the value is in what the attempt teaches you rather than simply having the right answer.

1. Maps and Hash Tables

Now that we have some experience with key-based data structures from Priority Queues, it's time to start exploring some other options keys give us. Significantly, Priority Queues are still governed by some kind of order – hence the “queue” part of the name – which means that even if we have the key for an entry we want, we can't fish it back out of the Priority Queue until it's turn comes up. Adaptable Priority Queues at least let us remove an arbitrary entry if we already had a reference to it, but not to find one based on its key.

Maps are a way to store and access entries using their keys. It's a fairly broad structure with many implementations and sub-types, but in general, Maps function similar to an array that uses whatever data you want as the index instead of the entry's position. The only condition is that, as every position in an array is unique, every key in the Map must also be unique.

Hash Tables, then, implement Maps by going into detail about how to turn any arbitrary key into a way to retrieve data. This makes the similarity to arrays explicit by creating a “bucket array” for storing the entries, each with a unique position. It's the job of the hash function and the compression function to turn whatever arbitrary key you've chosen into a specific assignment to a particular bucket, both for storing the entry and retrieving it later. All these steps are easier said than done, however, and riddled with design decisions that can have a huge impact on the Hash Table's performance.

The goal of this lab will be to learn when and how to apply a Hash Table to a problem. They're tricky structures, even if you're using the pre-built class in Java, but they can be extremely useful – as you'll hopefully see today!

2. Outline

A classic example of a problem you can solve with a Hash Table is one that actually runs counter to what most Hash Tables are trying to accomplish. We spent a lot of time in class talking about what makes a good Hash Function, i.e. one that reduces collisions between entries trying to fit in the same bucket. Most of the benefits of going from an arbitary key to a particular space in the Hash Table in constant time are lost if multiple entries start piling up in the same place.

Conversely, Hash Tables can also be a good way to add up those collisions – since anything

with the same key will get dropped into the same bucket, each bucket can instead be used as a counter. It's particularly helpful if you're not sure what you'll be counting before you start doing it, since aside from configuring what the keys look like in general, you don't need to specify what keys to look out for or even how many there might be.

A classic example of this sort of problem is counting up how many times each word shows up in a document. It's the sort of problem that can be solved with other data structures, true, but can be a little clunky to implement – there's a lot of words in a language, after all. Knowing in advance how many different words (and which ones) will be in a document is pretty tricky, but it's no problem for a Hash Table configured to accept Strings as keys. So that's what we'll do.

2.1 Them's Countin' Words

Write a program that accepts lines of text written into the terminal. You might want to submit your input one word at a time, or as one long line, or as a bunch of separate lines, but there must be a trigger that finishes the input process and moves on to the next step.

That next step is counting each instance of each word in the text you've submitted. It should be able to recognize upper-case and lower-case words as the same word, not to mention ignore any neighbouring punctuation.

Once it's done counting the number of instances for each word, print the results to the terminal. The results needn't be printed in any particular order, neither the alphabetical order of the words nor by the frequency of their appearances, but each word from the text should be printed along with how many times it appeared.

You can use the standard Hashtable class built into Java for this (you might also need to look at the functions provided by Entry, don't forget those). In fact, you should, even if you can think of some clever solution using a different data structure!

3. Deliverables

All lab submissions will be done through CourSys at https://courses.cs.sfu.ca/. For this lab, that should include:

1. A .zip archive of your code, organized into a single Java project. To ensure compatibility, it's recommended to use IntelliJ while developing your code, then using the export function under File -> Export -> Project To .zip File. **Don't forget to include a test file**, a Main.java with a main method that prints the results of a few runs of your functions to the terminal would be best.

Be sure you tidy up your code before submitting! Check the Java code style guide posted on the course website, and do your best to provide helpful comments.