CS310: Advanced Data Structures and Algorithmns

Spring 2014 Programming Assignment 2

Due: Thursday, March 13 2014 at midnight

Goals

This assignment aims to help you:

- Learn about hashing, in two important variations.
- Do some refactoring of class relationships.
- See what AbstractCollection and AbstractMap are good for.
- Do actual timings to check the claim of O(1) lookup by hashing.

Reading

Read Weiss Chap. 20, except the HashMap code.

Questions

1. Download Weiss' HashSet<E>, and delete the imports with "weiss" in them. Use package cs310.util instead of weiss.util. The sources should therefore be in src/cs310/util. See PA1 and the Java documentation for the use of packages.

   Delete the non-standard method getMatch(). Add back imports to JDK interfaces and the JDK classes AbstractCollection<E> and exception classes, until it compiles. Be sure to *not* import the JDK HashSet (no * in JDK imports!). Test it with a little TestSet program like Fig. 6.32, pg. 265.

2. Download MapDemo from Chap. 6, minimally modify it to use HashMap, again replacing all "weiss" imports. Use the package cs310.client for this client code, so put the source in src/cs310/client.

   Implement the following changes:

   - Rename the MapDemo and call it TestMap.
   - Make sure the test function handles exceptions and prints out what happens, but goes on testing anyway.
   - Give TestMap a constructor with a Map argument, make printMap an object method, and put the rest of the testing code in method test(), which calls printMap at its end. Then main just calls the constructor, then test() on that object, and requires no command arguments.
   - Have it construct and test TestMaps for java.util.HashMap and java.util.TreeMap.

3. Weiss implements HashSet (with getMatch) and then uses that implementation to implement HashMap. This is backwards to the way the JDK does it. Maps are more powerful than Sets, so we should implement Map, and then use it to implement Set, just as the JDK does. Then we don't need an "extra" method (getMatch of Weiss' Set) to make it work.

   Implement your own HashMap class HashMap1, in package cs310.util. Start by copying HashSet.java and rename it HashMap1.java, since that's where the real implementation code (involving the array of

HashEntry) goes now. HashMap1 should not extend Weiss' MapImpl, because that assumes Set has getMatch, a non-standard method. **Instead, it should directly extend JDK's AbstractMap<K,V>**. Be sure to read the JDK documentation for AbstractMap to see what methods you need to implement, including ones already implemented but not using the power of hashing. Make sure you end up with fast $O(1)$ get, put, and remove. You can get useful code out of MapImpl (pp. 749–755).

For this problem, have keySet(), values(), and entrySet() throw UnSupportedOperationException (that means some of the TestMap code will fail and that's ok, as long as it continues testing). Be sure to fix the Javadoc header comments in HashMap1 as well as the code! You can copy header comments from Weiss's MapImpl, or make up your own. Test it with TestMap from problem 2, edited to additionally construct and test a HashMap1.

4. Write HashSet1 (in package cs310.util) based on HashMap1 as simply as possible without giving up the power of hashing for $O(1)$ actions. HashSet1 ISA AbstractCollection, and HASA HashMap1, with keys but unused (null) values. Test it with TestSet, edited to construct a HashSet1 as well as a HashSet.

5. Do one of the following and understand the solution for the other:

   (a) Implement separate chaining. Call your class HashMap2, starting from HashMap1 of problem 3. Test it with TestMap, edited to additionally construct a HashMap2.

   (b) Finish HashMap1 by implementing keySet(), values(), and entrySet(). First implement keySet() by following the general setup in MapImpl, pg. 753, except that keySetIterators implementation needs to look like iterator in HashSet of problem 1, that is, work directly from the hashtable array. Then you can use that iterator to scan the underlying data and implement iterators for values and Map.Entry's. For entrySet, you'll need another private class... extends ViewClass..., not shown on pg. 753 because in Weiss's setup, the whole thing is built on such a Set.

6. **Performance testing:** Write a cs310.client.TestMapPerf class that (in a method called timeTest) times n random gets from 3 different sized Maps, using maps from line number to String word loaded from "words", the dictionary file we used for pa1 (Use implementation ideas from pa1, but a different data structure, of course). So you have 3 Map¡Integer,String¿ representing 3 cases:

   Case 0 loads line numbers 0, 4, 8, 12, . . . , case 1 loads lines 0, 2, 4, 6,. . . , and case 2 loads all the lines. In other words – if case 0 has m words then case 1 has $2m$ words and case 2 has $4m$ words.

   Generate $n = 10,000$ random numbers. For each number get the line number and word from each of the 3 maps. Print out the number of milliseconds (ms) for the 10,000 gets for each case. Different computers perform differently, so if the times are not (mostly) between 200 ms and 2000 ms (2 seconds) for each test case, go to 100,000 gets or 1000 gets to bring the times into this range, for sufficient accuracy and not-too-long runs.

   Report the class (HashMap or whatever), the case number (0, 1, or 2), the time (ms), the number of gets, and the microseconds/get for each test, a total of 12 lines of output. Note that the Date class can do these timings and java.util.Random can generate a sequence of random numbers in the desired range. The TestMapPerf constructor takes an empty Map and its timeTest() method takes a filename. The main creates Maps using HashMap1 and HashMap2, java.util.HashMap, and java.util.TreeMap and tests all of them with the argued file. Usage: java TestMapPerf 10000 words (for $n = 10,000$, the starting case)

Implementation notes

- In the JDK, Collection does not implement Serializable, so you can drop this implements clause and avoid the annoying warnings about ids for the classes.

- HashEntry is not generic in Weiss's code, which makes it easier to create an array of HashEntry, since arrays of generic types can only be created by a trick shown on pg. 156, lines 37–38. With a non-generic HashEntry, you end up having to cast now and then, for example, on line 32 of pg 795. But as long as you realize this cast is OK and expected, its no real problem.

- For help on generics, see "Notes on Generics" under Resources on the class web page.

- In the file memo.txt, answer (briefly) the following questions, in one to three pages (60-180 lines) of text:

1. Does your HashSet1 prove that any Map implementation can be turned into a Set application (with the help of AbstractCollection)? Why or why not?
2. Report on experiments with TestMap. Does your data agree with the O(1) claim of hashing?

Delivery

Before the due date, assemble files in the pa2 subdirectory of your provided cs310 directory on our UNIX site. The pa2 directory should contain the src and classes subdirectories (similar to pa1). To compile use the following command, while in the src directory: javac -d ../classes cs310/*/*.java

The src subdirectory should contain two subdirectories client and util, hence the slightly different compilation command. The source files are as follows:

- memo.txt (plain txt file)
- cs310.util.HashSet.java
- cs310.client.TestSet.java Usage: java cs310.client.TestSet
- cs310.client.TestMap.java Usage: java cs310.client.TestMap
- cs310.util.HashMap1.java (with working keySet, etc. if doing 5b)
- cs310.util.HashSet1.java
- cs310.util.HashMap2.java, if doing 5a
- cs310.client.TestMapPerf.java Usage: java cs310.client.TestMapPerf 10000 words

words is the Unix dictionary.