CS 310 Data Structures
Programming Assignment 2
Spring, 2021

Hashing

Due Sunday, Apr. 4 by midnight in your cs310/pa2 directory

Purpose

This assignment aims to help you:

- Learn about hashing and hash tables.
- Gain experience in fairly sophisticated generic class programming, and inner classes.
- See what JDK AbstractMap is good for, and inheritance in general.
- Understand “view” semantics for keySet(), entrySet(), and values() of Map
- Do actual timings to check the claim of O(1) lookup by hashing.

Reading

Read S&W Sec. 3.1 on STs (Maps), and Sec. 3.4 on hash tables. Also read the JDK API docs on HashMap, including the discussion of keySet, values, and entrySet as returning "set views". If you have Weiss, read Section 6.10 “Views in the collections API”. For generics, read S&W pp. 122-123, pp. 134-135, and pg. 158. Then read Java tutorial on Generics at least through the page on "Raw Types".

See Suggested Steps for more information.

Description

1. **Setting up.** Transfer the pa2setup directory system (zip file provided) to your development area. This is linked to the class web page along with this assignment. This provides `HashMap.java.txt`, the original Java 8 JDK code and several other files as well as the needed directories for your project. Here is the directory structure (by Linux du) for the source files. As usual, ignore the numbers at the starts of lines. A similar directory system would be used under bin for IDE use.

```
10      ./src/cs310/client
61      ./src/cs310/util
71      ./src/cs310
```

Note how there are two levels below src, accommodating packages cs310.util, for HashMap and related files, and cs310.client, for the two test clients, TestMap.java and TestMapPerf.java.

To compile, we need two javac commands, one for each source directory, however we handle the .class files (in separate bin area or not):

- Case of separate binaries: cd src, then `javac -d ../bin cs310/util/*.java` and `javac -d ../bin cs310/client/*.java`
- Case of commingled binaries: cd src, then `javac cs310/util/*.java` and `javac cs310/client/*.java`

To run TestMap: It doesn't need an input file.

- Case of separate binaries: cd bin and then `java cs310.client.TestMap`.
- Case of commingled binaries: cd src and then `java cs310.client.TestMap`

`HashMap.java.txt` is the actual OpenJDK Java 8 HashMap source, in a .txt file so as not to cause error markings in an IDE. Note how it (and all the other HashMap versions) extends AbstractMap.java, our first encounter with inheritance in a project. That means that if HashMap does not implement a Map method and AbstractMap does implement it, the one in AbstractMap will be used when the client calls that method. AbstractMap is also there, but you don't need to change it. HashMap1.java is HashMap.java.txt with all the TreeNode code removed, and all the methods that AbstractMap supplies removed, except for the all-important O(1) methods we want to test. Also unneeded Map constructors have been removed. This greatly simplified source file, still fully operational, should be readable. Run the provided cs310.client.TestMap to check your setup. There are no deliverables for this part.

2. **Encapsulating the HashMap.** Modify HashMap1.java into HashMap2.java, where the second version makes all the non-Map-API methods and instance variables private and all inner classes private as well. This has the effect of "sealing up" the HashMap class, or encapsulating it. The JDK HashMap class is the way it is because it is used as the basis for several other JDK classes such as LinkedHashMap and HashSet, but we want to focus on HashMap. Run TestMap, slightly edited to use HashMap2, to check that you haven't broken it by over-privitizing or something else. Do not remove the comments in the source file: removing comments is a bad programming practice.

Important note on HashMap's implementation When you read HashMap's get and put, you will see the following mysterious code: `tab[i=(n-1)&hash]`, where `n` is the size of the hash table, `tab` is the hash table itself, and `hash` is a 32-bit hashed value computed from the key. It is important to realize that `n` here is always an exact power of 2, so for example, `n` = 2^11 = 2048 = binary 100000000000 (1 followed by 11 binary 0s). Then `n-1` is the binary number just below 100000000000, so is 11111111111 (11 binary 1s). This forms a "mask for 11 bits", so when the bitwise AND is computed by `i=(n-1)&hash`, the 11 lowest bits of `hash` are extracted from `hash` to `i`, with 0s in all higher bits of `i`. So `i` <= 2047, and is a good bucket number for the hash table of 2048 buckets. Since i is a good bucket number, and tab is the hash table here, tab[i] is the reference to the collision list for the bucket.This reference is to a Node in the original HashMap code (the lead Node of the collision list), but can equally well be to a ArrayList<Node> in the new code.

3. **Using ArrayList<Node> for collision lists.** Modify HashMap2.java into HashMap3.java by (eventually) replacing each collision list, a Node list in HashMap2.java, with a JDK `ArrayList<Node>`. We can use the Node as defined in HashMap2 for the ArrayList<Node>, ignoring its `next` reference, since the ArrayList will handle the relationships between elements. The idea is to make the code much simpler and more beautiful than what you see in the original HashMap2.java file, by hiding all the low-level list-scanning details, i.e., following the next reference in HashMap code. The switch from the current collision list code needs to be done carefully so as not to disturb the crucial `entrySet` method (depended upon by AbstractMap) while working on `get` and `put`. Instead of yanking out the array of Nodes, let the array of Nodes `table` coexist with your new array of ArrayList<Node> named `table1`, and let the entrySet code (and other code) continue to use the old Node array. The entryset will then appear empty if you display it, but it at least it works. Create the new array of ArrayList<Node> for new HashMap3 instance variable `table1`, and size it as a power of 2, 16 to start with. Put the array creation in the no-args constructor, the only one we'll use for testing here. Implement get and put using the array of ArrayList<Node> in `table1`. Now the get and put code should look simpler, although not as neat as the code in `SeparateChainingHashST`, pg. 465, since we still have to iterate down a collision list in our HashMap code (whereas this iteration is cleverly hidden inside the little maps in the textbook code). Test get and put by running TestMap after editing it to use a HashMap3. It only does a few gets and puts, so `resize` should not be called at all in this testing, but it's a good idea to leave the calls to it in `put` for later use (at least commented-out ones).

4. **Getting resize working.** To handle large data, we need `resize` working. Modify HashMap3.java into HashMap4.java by reimplementing resize to use the ArrayLists. There are two ways to go on this. You can build a new HT twice as big as before using code like that in the constructor, and then move all the key-value pairs from old to new HT, or you can create a whole new HashMap4 object, load it by calling its put using all the key-value pairs of the old HT, and then copy its table1 to the real table1. The second option is like the book's approach on pg. 474. However, note that we don't have use of keySet() to find all the keys here, because keySet

is still implemented using table, not table1. We have iterate through the buckets and for each, iterate through its ArrayList. To test resize, run TestMapPerf, which reads and puts all the words (99170 of them) from the UNIX spell dictionary "words" into the HashMap, thus calling resize many times. The words file is supplied in the project's base directory. First be sure to edit TestMapPerf to use HashMap4.

5. **Performance testing**. Study the cs310.client.TestMapPerf supplied program that (in its timeTest method) times N random gets from 3 different sized maps, using maps from line number to String word loaded from "words", the largest case being all those words (99170 in all). The program uses only the constructor, get, and put (and from that, resize) of each Map. For a particular Map, case 0 uses line numbers up to #words/4, case 1 uses line numbers up to #words/2 and case 2 uses all the words, so #words=99170. As provided, it times JDK HashMap and JDK TreeMap and the provided HashMap1. You just need to change "HashMap1" to "HashMap4" to switch it over to test HashMap4. Also make sure there is no output (System.out.println(...)) during timings. Run it (see usage at end of this paragraph) to print out timing results, and find a value of N to make the times (mostly) between 100 ms and 2000 ms (2 seconds). Start with N=1,000,000 gets or 10,000,000 gets to bring the times into this range, for sufficient accuracy and not-too-long runs. The program reports the class (HashMap or whatever), the case number (0, 1, or 2), the time (ms), the number of gets, and the microseconds/get (us) for each test, a total of 12 lines of output. Usage (first try): `java cs310.client.TestMapPerf 1000000 <input file>` for N=1000000 gets, the starting case. The input file is words, provided (same as used for SpellChecker).

6. **Optional**. Implement remove and containsKey in HashMap4 to use the ArrayList collision lists. Add tests for containsKey to TestMap. Use TestMap, and TestMapPerf to check it. FrequencyCounter still can't be used as a client because it needs to iterate through the keys of the map. If you have time and excellent debugging skills, especially optionally try finishing the conversion, but be warned that the HashIterator is particularly tricky

memo.txt In the file memo.txt, answer the following questions, in one to three pages (60-120 lines) of text. Use complete sentences, as you would for an English class

1. Discuss your experiences in writing these programs. What was the hardest part for you? Are you now a hashing expert? Have you thought of any applications of hashing in problems you have seen outside this class?
2. Explain how JDK documentation for Map (or what else) guided you to decide which methods you needed to make private.
3. Report on experiments with TestMapPerf, showing one final run's output. Does your data agree with the O(1) claim of hashing? How do your HashMap4 compare to JDK's? Did switching to ArrayList collision lists hurt performance? How does JDK HashMap compare to JDK TreeMap?
4. Explain in your own words what it means that the keySet is a view on the JDK HashMap.

Delivery

Before the due date, assemble all the files in the pa2 subdirectory of your provided cs310 directory on our UNIX site. The files we will look at for grading are:

memo.txt (plain text file) in cs310/pa2/memo.txt

cs310.util.HashMap2.java (with "private" additions) in cs310/pa2/src/cs310/util/HashMap2.java·
cs310.util.HashMap3.java (with array of ArrayList<Node>) in cs310/pa2/src/cs310/util/HashMap3.java
cs310.client.TestMap.java (testing HashMap3) in cs310/pa2/src/cs310/client/TestMap.java
cs310.util.HashMap4.java (with array of ArrayList<Node>, and working resize) in cs310/pa2/src/cs310/util/HashMap4.java cs310.client.TestMapPerf.java (testing HashMap4, if submitted) in cs310/pa2/src/cs310/client/TestMapPerf.java