COMP 3160 – Fall 2014
Homework 4: Sets, Maps, and Hash Tables

Number of People: Individual. Feel free to ask me for help, or visit the Computer Science Learning Center (www.cs.memphis.edu/cslc).

Due: Thurs., Oct. 16 by 1:00 pm

Submission: Zip all of your Java source files (you can zip the entire project folder if using an IDE) into a single file and upload it to the proper folder in the eCourseware dropbox at https://elearn.memphis.edu. The non-coding portions can be submitted within your zip file, or as a hard copy at the beginning of class.

Coding Style: Use consistent indentation. Use standard Java naming conventions for **variableAndMethodNames**, **ClassNames**, **CONSTANT_NAMES**. Include a reasonable amount of comments.

Grader: TA, Kyle Cherry (kcherry2@memphis.edu). Questions about grading? Please contact him first!

1. **(10 pts)** Suppose you want to use a hash table to store movie information. The keys are the movie titles (as strings), while the values are the years in which the movies were made. The hash function is the length of the title string, modulo the number of indices in the table. The hash table has a capacity of 8, and it uses open addressing with linear probing. For this problem, assume that the table is never rehashed.

   a. *(5 pts)* Draw what the table looks like after adding these key-value pairs in this order:

      <"Forrest Gump", 1994>
      <"Inception", 2010>
      <"Batman", 1989>
      <"Superman", 1978>
      <"Star Wars", 1977>
      <"Jaws", 1975>

   b. *(2 pts)* Suppose you then want to add the pair <"Shrek", 2001> into the hash table. Describe what the linear probing procedure would do to find the next empty slot. What index would the new entry ultimately get placed at?

   c. *(3 pts)* An alternative to linear probing is *quadratic probing*. With linear probing, a collision at index i results in searching indices $i + 1$, $i + 2$, $i + 3$, $i + 4$, and so on. With quadratic probing, a collision at index i results in searching indices $i + 1$, $i + 4$, $i + 9$, $i + 16$, and so on. (Just as with linear probing, the indices should loop back around to the beginning of the table if necessary.) Repeat part (b), but this time using quadratic instead of linear probing. What happens as you try to find an empty spot?

2. **(15 pts)** Non-honors students: pick any 2 coding problems *(7.5 pts each)*. Honors students: do all 3 *(5 pts each)*.

   a. Within the **BSTSet** class that we wrote in lecture, implement the methods **union(BSTSet<E> s)** and **intersection(BSTSet<E> s)**. These methods should return the resulting **BSTSet** of those operations between the calling set and the argument. Do not modify the calling set. Feel free to call the existing class methods (**add**, **contains**, **remove**) if needed.

      Reminder about union: $A \cup B$ is the set of all elements that are in either A or B (or both). For example, if $A = \{1, 4, 5\}$ and $B = \{4, 2, 8, 9\}$, then $A \cup B = \{1, 2, 4, 5, 8, 9\}$.

      Reminder about intersection: $A \cap B$ is the set of all elements that are in both A and B. For example, if $A = \{1, 4, 5\}$ and $B = \{4, 2, 8, 9\}$, then $A \cap B = \{4\}$.

b. Here’s an example of a situation that’s well-suited for using the map data structure. Suppose you have a data file of students at a university, and you’re trying to get an idea of how many students are in each major. You can read the file and parse the information into a map, where the keys are the majors and the values are the counts of how many students are in that major.

   Write a method **countMajors(String filename)** that reads the data in **filename** and returns a **Map<String, Integer>** as described above. Feel free to use the built-in Java map classes (**java.util.HashMap**, **java.util.TreeMap**) in your solution. You can assume that each line of the data file is formatted like this:

   lastname,firstname,major

   I have provided a small-ish sample data file on eCourseware that you can use to help test your code. However, your method should work for <u>any</u> file formatted the same way!

c. Implement a **rehash()** method for the **HashMapChained** class that we wrote in lecture.