Lab 9: Hash Tables[1]
Due: 30 November 2016

A hash table can be very useful to speed up the MINIMAX technique for finding optimal game strategies. Once the value of a game board has been computed, the board, along with the computed value, is stored in a hash table. Whenever the MINIMAX algorithm needs to evaluate a board, it checks to see if that board is in the hash table. If it is, it can use the stored value, potentially saving a significant amount of time by not having to recompute the value. In this lab, you will implement and test a chained hash table for **GameBoard**s, a *very* simple game board class.

The **GameBoard**s are 4 x 4 arrays of randomly generated integers from 0 to 2, inclusive. We will say that this is a two-person game and that the 1s are the pieces of Player 1 and the 2s are the pieces of Player 2. We will assume that we are evaluating the board from the perspective of Player 1 and use a very simple evaluation function: the value of a board to Player 1 is the number of pieces they have on the board minus the number of pieces Player 2 has on the board. This number is calculated by the **boardValue** function in the supplied **GameBoard** class. So a <key, value> pair that is being stored in the hash table will be a **<board, board.boardValue()>** pair, where **board** is a **GameBoard**. I have provided an **Entry** class that you can use to create these pairs. **Entry** objects contain a **GameBoard** and an integer, the latter of which is the value of the game board.

Note that the value provided by the boardValue function is *not* the hash code of the board. The function that computes the hash code is the hashCode function, also in the GameBoard class.

The Assignment

I have given you a **TestHashTables** class that provides a framework to test your **HashTable** class. It's a "framework" in the sense that you need to add some code to it. **The places where you need to add code, along with instructions on what you need to add, are indicated by comments that start with // ***.** We will be using this to test your code.

The **GameBoard** and **Entry** classes are both complete. You should not change any of this code.

You need to write a **HashTable** class that implements a chained hash table for **Entry**s. You should implement your hash table as an array of **ArrayList**s.

The following methods, in addition to your constructor(s), are *required*. You may not need them all, but we want the class to have all the funcitonality of a hash table. You can write more methods as long as they do not change the functionality of your class.

---

[1] Adapted from a lab by Jonathan Shewchuk at University of California, Berkeley.

insert: This method has two parameters, a **GameBoard** and an integer that is the value of the **GameBoard**. The method needs to create an **Entry** object that contains these two items and insert that **Entry** object into the hash table; this method should also return the **Entry**.

find: This method has one parameter, a **GameBoard**. The method needs to search for that **GameBoard** (key) in the hash table. If it finds that key, i.e. if there is an **Entry** in the relevant chain whose **GameBoard** matches the parameter **GameBoard**, it returns the **Entry**, but does not remove it. If it does not find such an **Entry**, it should return **null**.

remove: This method has one parameter, a **GameBoard**. The method needs to search for that **GameBoard** (key) in the hash table. If it finds that key, i.e. if there is an **Entry** in the relevant chain whose **GameBoard** matches the parameter **GameBoard**, it removes that **Entry** from the hash table. It also returns that **Entry**. If it does not find such an **Entry**, it should return **null**.

isEmpty: Returns true if the has table is empty; otherwise, it returns false.

getLoadFactor: Returns the load factor of the hash table

getNumBuckets: Returns the number of buckets (chains) in the hash table.

compress: This method compresses the hash code sent to it so that it fits into the hash table range of indices. You may use use the following:

```
public int compress(int hashCode) {
    return (((13 * hashCode) + 37) % 16908799) % numBuckets;
}
```

calcVariance: Calculate the variance of the chain length in the tests of your hash table with different load factors (see below).

Note that the actual hash function is not in the **HashTable** class since it is *problem specific* and so is in the **GameBoard** class.
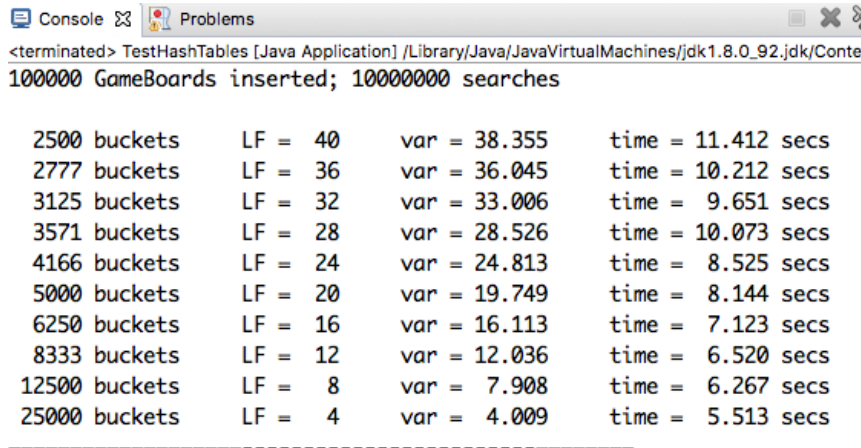

Required Output

I have provided a file (sample-output.rtf) that shows you a sample run from my code. Here are some more details:

Part 1

For this part of the lab, write code at the place indicated by "OUTPUT: PART 1" in the provided **TestHashTables.java** file to test your hash table at different load factors (from 40.0 down to 4.0, by decrements of 4.0) after you have inserted 100,000 random boards. In order for the differences in time to be discernible, measure the time needed to do 10,000,000 lookups, each one using a randomly generated board (but these do not need to be the same ones you inserted, since we're just timing *attempts* at finding a key). You will probably want to do fewer

searches while you are debugging your code. Your output should show, for each load factor, the number of buckets, the load factor, the variance in the length of the chains, and the total time taken for the 10,000,000 searches. Your program should produce something like the following output for this part of the lab:

```
Console 🞨  🧑 Problems                                          ◻ ✖ 🧑
<terminated> TestHashTables [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Conte
100000 GameBoards inserted; 10000000 searches

     2500 buckets      LF =  40      var = 38.355      time = 11.412 secs
     2777 buckets      LF =  36      var = 36.045      time = 10.212 secs
     3125 buckets      LF =  32      var = 33.006      time =  9.651 secs
     3571 buckets      LF =  28      var = 28.526      time = 10.073 secs
     4166 buckets      LF =  24      var = 24.813      time =  8.525 secs
     5000 buckets      LF =  20      var = 19.749      time =  8.144 secs
     6250 buckets      LF =  16      var = 16.113      time =  7.123 secs
     8333 buckets      LF =  12      var = 12.036      time =  6.520 secs
    12500 buckets      LF =   8      var =  7.908      time =  6.267 secs
    25000 buckets      LF =   4      var =  4.009      time =  5.513 secs
    _____
```

To create this nice formatting, you will probably want to use **System.out.printf**. I will leave it up to you look online and figure out how to use it. Your number of buckets and load factors should match mine, but it is not necessary, nor is it possible, to have the same variances and timings that I have here. You know the relationship between the number of items in the hash table, the number of buckets in the hash table, and the load factor. So, to get a desired load factor, compute the number of buckets that would give you that load factor, given that you have 10,000 items to put in the hash table.

The item labeled "var" in the output above is the variance of the list lengths. Variance is a statistical measure of how much variation there is in a set of numbers, i.e. how much they are spread apart. If all the number are the same, the variance is zero. You can find the formula for variance at http://mathworld.wolfram.com/Variance.html. Start at the paragraph starting with: "If the underlying distribution is not known, then the sample variance may be computed as…" But be careful to read the next paragraph starting: "Note that the sample variance 🖾 defined above is *not* an unbiased estimator**…"** This changes the formula you need to use.

Part 2

For this part of the lab, you need to complete the code in the section "OUTPUT: PART 2" that I have written to test your **HashTable** functions. Again, the places where you need to write code are marked with a comment starting with // ***.

Submitting Your Program

Please submit your program in the usual way on BlackBoard and turn in a hardcopy.