Lab 8: - Making a Hash of Things

Objective: Investigate properties of hash tables, and conduct empirical experiments to verify some theoretical results about those hash tables.

Background: In the context of hash tables, a **search miss**, as its name suggests, occurs when a search ends in failure -- the sought after key is not found. The **search miss cost** is the number of probes required to prove that a key (not in the table) is indeed not in the table.

For instance, consider the table OOEEOOOE (O = Occupied cell, E = Empty cell, aka null)

If a key hashed to index 0, it would take **3** probes to **prove** that it was indeed not there: we would probe elements 0, 1 and 2. The search miss cost would therefore be 3.

if a key instead hashed to index 2, it would take just 1 probe to prove that it was indeed not present, since element 2 is empty. The search miss cost would be 1.

We define the **average search miss cost** as the sum of the search miss costs at each index divided by the table size N.

For example, with OOEEOOOE, the cost would be (3 + 2 + 1 + 1 + 4 + 3 + 2 + 1) / 8 = 2.125

As usual, beware edge cases, such as OOEEEOEO. Also, beware OREOs, they are addictive!

These results are **empirical**, based on examining hash table contents. But theoreticians have been hard at work, and found that the theoretical average search miss cost, using linear probing, is:

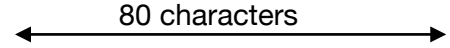
$$\sim \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

where as usual λ is the load factor n/N. In this lab, you will empirically measure average search miss cost and then print out the results.

Note: To allow you to concentrate on the algorithms, rather than get lost in the code, for this project we are going to subclass LinearProbingHashTable, a modification of a class by Sedgewick and Wayne of Princeton University. Read over that class -- it is quite straightforward -- before you continue.

Classes: This lab requires that you write 3 classes
1. TitanProbeHashMap: This subclass of LinearProbingHashTable will define at least 3 methods (you may define others as you deem appropriate):

toString(): Returns a String consisting of O's and E's (for Occupied and Empty (null) cells). If the String is more than 80 characters, insert a line break after j*80 characters, where j = 1, 2, 3, ... e.g.,




```
OOOEEEEOOOOEOEOEOEO ... O
EEOEOEOEEEOEOEOEEEO ... E
...
```

empiricalAverageSearchMissCost(): This method will return, as a double, the empirical average search miss cost, by processing the table as described above.

empiricalAverageSearchMissCost(String str): This overloaded method will return as a double, the empirical average search miss cost, but instead of working with the actual table, it will merely take a String of O's and E's. For example:

empiricalAverageSearchMissCost("OOEEOOOE") would return 2.125.

Note that once you have the first version written, this version should be a minor tweak.

Question: Should any of these be static?

2. HashMapPlayground: This class will put averageSearchMissCost() through its paces, and contain at *least* one method:

printExperimentalResultsTable(): Inserts λ*N random int keys into an empty hash table of size N, and compute the average search miss cost for that table. If you did this just once, you might get an errant result: so do this 1000 times, and *average* the 1000 average search miss costs.

You will repeat this for λ = 0.10 to 0.90 by 0.10. (see the Sample Output). Use N = 8192 as the table size.

Finally, print a nicely labeled table, where each row corresponds to a particular value of λ, and the columns represent λ, theoretical average search miss cost and the empirical average search miss cost.

For grading purposes, use nextInt() to generate any random number between -2^{31} and $2^{31}-1$, based on a Random object with a seed of 42.

Question: are you satisfied with the *behavior* of this method? is it doing too little? too much? Are you satisfied with the *name* of the method?

3. Lab8:
1. main(): instantiate a HashMapPlayground and invoke printExperimentalResultsTable()
2. testASMC(): use JUnit testing to test empiricalAverageSearchMissCost() for the cases "EEEEEEEE", "OOEEOOOE", "EEOE", "OOEEEOEO", "EOEEEOEO"

Sample Output:

L	Empirical ASMC	Theoretical ASMC
0.1	1.117	1.117
0.2	1.281	1.282
0.3	1.520	1.520
0.4	1.889	1.885
0.5	2.500	2.502
0.6	3.625	3.590
0.7	6.056	5.953
0.8	13.000	13.137
0.9	50.500	49.048

Requirements: Your output should match that shown in the sample output. Code must adhere to best practices and standards described in class and specified on the website. Your code should be **rigorously** tested. Your code may be tested with any values, not just those alluded to in the output section.

Git Requirements: Get (or should we say Git 😀) into the habit of using Git to make commits (and push them to GitHub) when you have achieved a significant milestone. For instance, after creating the project; and after finishing each method.

Deliverables: Your entire project, in a folder named LastNameFirstName-Lab8, zipped, and a screenshot (.png or .jpg) showing the sample output, uploaded separately (not in the zipped project). Also, include a screenshot from GitHub showing that you have made ≥ 4 commits.