CS 0445 Spring 2022
Recitation Exercise 11

Note: This exercise will very likely require more than a single recitation session to complete. I recommend starting it prior to your recitation session. You may also need to complete it after your recitation has ended. I will post my solution early next week so that you will see it prior to Exam 2.

Introduction:

Recently in lecture we discussed hashing and specifically the linear probing collision resolution scheme. This scheme is implemented in the author's HashedDictionary<K, V> class, as listed in file HashedDictionary.java. (Note: You will also need the interface file, DictionaryInterface.java, for this exercise).

An issue that we discussed with regard to open addressing in general, and linear probing in particular, is that, as the table fills the performance of the hash table begins to degrade. We said that a "rule of thumb" for linear probing is to keep a hash table of size M no more than ½ full. to maintain this density, the table will be "up-sized" when its alpha value (N/M) gets above ½. To see how this is done, look carefully at the HashedDictionary.java file. Note in particular the isTableTooFull() method, which is called after each add() to the table.

Details:

In this exercise we will empirically examine the performance of the linear probing scheme as implemented in the HashedDictionary<K, V> class. I have modified this class in the following simple ways:

1) I have added an instance variable called `probes`. In each call of the linearProbe() method (which is the method to actually access the table when doing an add(), remove() or getValue() call) I initialize `probes` to 0 and then increment it for each access of the table array.
2) I have added an instance method called getProbes() which will return the current value of `probes`. The idea is that after a call to add(), remove(), or getValue(), we can use the method getProbes() so see how many probes of the table were necessary for that operation.
3) I have changed the final variable MAX_LOAD_FACTOR into a regular variable so that it can be modified. Note that MAX_LOAD_FACTOR is utilized in the isTableTooFull() method in determining when the table should be resized.
4) I have added an instance method called setLoadFactor(double factor) which will mutate the MAX_LOAD_FACTOR variable to a new value. This will allow a user of the HashedDictionary<K,V> to change how full the table will get before it is resized.

Note: Be sure to use the HashedDictionary.java file from this recitation exercise, and not the version from the course Handouts. The version in the course Handouts does not have the modifications specified above.

Write a main program called Rec11.java, which takes 2 command line arguments, and which will do the following:

1) Parse the first command line argument into an int, which will represent the initial size of the HashedDictionary<K,V>. [Note: If the argument is prime, then it will be used for the table size. If the argument is not prime, then the next prime number larger than the argument will be selected for the table size – this is specified in the author's constructor method, so you don't have to figure this yourselves]. Call this value `init_M`.

2) Parse the second command line argument into a double, which will represent the maximum alpha value for the table before resizing. Call this value `alpha`.
3) Create a new HashedDictionary<String,String> using the `init_M` value from the first command line argument for the size. Then call the setLoadFactor() method with the `alpha` value from the second command line argument.
4) Assign `N = alpha * init_M` = number of keys to add to the table.
5) Add N/2 random "words" (up to 10 letters in length) to the table, keeping track of the total number of probes required for these N/2 adds. To generate random "words" of up to 10 letters in length, use the method you wrote for Recitation Exercise 9. If you did not complete Recitation Exercise 9, you can use my solution (posted in the course Canvas site in file StringHelp.java). Keep track of the maximum number of probes needed for any add and the total number of probes needed for all of the adds, and output the max and average values for these N/2 adds.
6) Repeat this process again for the next N/2 adds, outputting the same information. Note that for these N/2 adds the table will be more full, so you may get different results. **Note:** If N is odd then by the procedures above you will do only N-1 rather than N total adds. This is fine as long as you use the correct divisors for your averages.
7) At this point the table should be at its maximum alpha level. Now do 1000 getValue() calls using random "words", also keeping track of the maximum and total number of probes necessary for the method calls. After all of the calls have completed, output the maximum and average number of probes for the 1000 calls.

To see how your output might look, see my example output in file Rec11-out.txt. This output was generated via the invocation:

```
> java Rec11 199 0.5
```

Once your main program is working correctly, run it with different command line arguments to see how the results change. Specifically, use the following 4 values for the first argument: 199, 401, 797, 1601. Note that we are approximately doubling the table size with each new value, and all of these values are prime numbers.

For each of the table sizes above, use the following values for the second argument: 0.5, 0.7, 0.9, 0.95.

Note the trends in your output with the runs. In particular, note the effect of both the `alpha` value and the `init_M` value on the probe counts.

Consider your results and consider whether or not the "rule of thumb" of 0.5 for the maximum alpha for linear probing is a good choice. Discuss your programs and results with your classmates.