University of Puerto Rico – Mayagüez
Department of Computer Science and Engineering

CIIC 4020 / ICOM 4035 – Data Structures
Prof. Juan O. López Gerena
Spring 2019-2020
Laboratory #7 – Hash Tables

In this lab activity, we'll add functionality to our Hash Table implementation. The two factors that affect a hash table's performance are: *initial capacity* and *load factor*. We've already incorporated initial capacity, and now we'll incorporate the load factor.

The load factor of a hash table, defined as $\frac{num.elements}{capacity}$, is a measure of how full it is, where by capacity we are referring to the number of *slots* or *buckets*. Recall that as the hash table's size increases, collisions become more likely, thereby decreasing the efficiency of the hash table. The load factor is used to try to strike a balance between having a capacity that is too large, which might waste space, and a capacity that is too small, which might result in frequent collisions.

The hash table implementation in java.util.HashMap uses a load factor of 0.75, so we'll use that value as well. It's important to note that this value is chosen because that implementation uses separate chaining (*which we'll also be using*). However, open addressing schemes typically aim for a load factor of 0.5. The algorithmic analysis used to calculate the ideal load factors are beyond the scope of this course.

Exercise 1: Add rehash() method

Copy your existing HashTableSC implementation and add the following field:

```
private final static double loadFactor = 0.75;
```

Add a private method named **`rehash`** that doubles the amount of buckets whenever the load factor of our hash table exceeds the `loadFactor` constant. This new method should be invoked by the **`put`** method when appropriate.

The **`rehash`** method must not only create a new array with a different capacity, but as the name implies, it needs to re-hash all of the entries in the hash table. Why? Because when we initially calculated the appropriate bucket for each entry, we used the number of buckets as part of the calculation. Now, that number of buckets will have changed, so we need to perform the calculation once again for each one. This somewhat similar to what happens with the `reAllocate` method for dynamic arrays, where we need to copy all of the values. In this case, we need to add all of the values to the new buckets by again performing the calculation for the bucket number, but this time using the new capacity.

Test your implementation by copying the MapTest class from the *original* HashTable implementation, and specify 2 as the initial capacity. You can skip using the HashTableFactory class if you wish, and just specify directly that you want to create an object of type HashTableSC.

Note: Ideally, we would always set the amount of buckets to be a prime number, since this would reduce the amount of collisions.

Exercise 2: HashSet

Java includes a Set implementation called HashSet that is backed by a hash table. In this exercise, you'll create your own version of HashSet. Copy the Set interface from the beginning of the course, and create an implementation called HashSet that uses an embedded hash table (using object composition).

Note that even if our hash table is not iterable, our HashSet class can be iterable by using the existing methods of our hash table (think about it).

Test your implementation, including when rehash gets called at least twice!