Lab 4: Hashing

In this lab, you will implement several different collision resolution techniques for Hashing.
You will also use Hashing in an application for word counting on a large text.

- **Part 0: Overview of startup code structure:** your TA will walk you through this part.
  - Download the following file: Lab7.zip
  - Hash functions:
    `StringHash`: Simple hashing function for strings
    `ReverseStringHash`: Uses the same function as before, but on the reversed sting
  - Hash tables:
    `ArrayHashTable`: An implementation for a hash table handling overflow by finding another place in the table via probing table positions. This must be general enough such that any form of probing can be done. So, the constructor requires 2 hash functions, in the case of linear and quadratic probing, the second one will be null, in the case of double hashing or quadratic double hashing the second hash function will be provided.
    `ChainedHashTable`: An implementation for a hash table handling overflow by using separate chaining (each table position can hold an array list of elements hashing to that position)
  - Stepping Functions: Functions to take the i value of the step and return the actual step value, the second and third require the long value of the second hash to calculate the step:
    `LinearProbeStep`, `QuadraticProbeStep`, `DoubleHashStep`, `QuadraticDoubleHashStep`.
  - Testing
    `ToyTest`: A simple test to check if all values are being added, and overwriting and get work properley, for debugging
    `HashingTest`: A test to check the statistics of the various methods of collision resolution using book chapters to test with.

- **Part 1: Implementing Hashing with different collision resolution techniques**
  Note: Be aware that `remove` and `contains` are implemented in such a way that methods `put` and `get` must be aware of special values to indicate that a table position has never been occupied (NULL at the table) and that the position is AVAILABLE but it has been occupied before (Entry<K,V> present at the table with key=null).
  1. **Implementing Linear Hashing**
     In this part, you will complete the implementation of methods `get` and `put` in class ArrayHashTable which implements interface HashTable. You will implement linear hashing, which attempts to place a key in the following successive positions h(k) mod N, (h(k)+1) mod N, (h(k)+2) mod N, etc. We can simply say the i-th attempt uses the function (h(k)+i) mod N.
     The hash function h1 has been initialized in the constructor and the stepping function as well. In addition LinearProbeStep method has been passed with the correct value for the step (check its method step).
     Use class ToyTest to test your implementation (you can comment out the tests for the classes you have not implemented yet).
  2. **Implementing quadratic hashing, (linear) double hashing, and quadratic double hashing.**
     Modify your implementation to allow for different in-table collision resolution techniques: linear hashing, quadratic hashing, linear double hashing, quadratic double hashing. This can be done by simply changing the "step" that you move away from the home address h(k). In linear hashing this step is step(i)=i, while for quadratic hashing step(i)=i*i. Linear double hashing and quadratic double hashing uses another hash function h2(k) to calculate the step:

step(i,k)=i*h2(k) and step(i,k)=i*i*h2(k), respectively.
Within ArrayHashTable the constructor requires a SteppingFunction object, this object should be used to calculate the step for each of the 4 cases, the object takes an int for the i value, a long (may or may not be used, depending on double hashing), and a size for taking the modulus (mod).
The stepping function has been initialized in the constructor, but their methods step must be updated to work correctly as they currently return 1 (see classes QuadraticProbeStep, DoubleHashStep, QuadraticDoubleHashStep). Test your code with ToyTest.

3. **Implementing Hashing with separate Chaining**
   Here the elements of the hash table are Linked Lists of entries (K,V) where K is a key and V is its associated value. Implement methods `get` and `put` for class ChainedHash.
   If you run out of time during the lab, feel free to skip this part and use this solution (ChainedHashTable.java) so that you can move on to Part 2 and perform the tests there.

- **Part 2: Word count problem.**
  You are given two large texts TheAeneidBook1.txt and TheOdysseyBook1.txt.
  Your task is to compute the list of different words in the text and the number of times each word occurs.
  The class `HashingTest` already implements for you the word parsing from these files, so that you can concentrate on word count.
  We propose an efficient way of doing this using a Hash Table. Store in the hash table each word found (key K) and its frequency (value V). Given a list of words (with repetitions), go through this list one by one. A new word is added to the hash table with frequency one, while a word already present in the hash table should get its frequency incremented.
  This should be done by simply using one of the Hashing classes that implement interface HashTable and only calling methods provided by this interface. This code is going to be a loop with a small body to be added inside method `HashingTest.HashAndStats(ArrayList words, HashTable table)`.
  The main program of class `HashingTest` takes care of testing it using different collision resolution techniques so that we can compare stats.
  Run this program and analyze its output.
  Feel free to play with different table sizes later and compare stats.

Created by Lucia Moura and Lachlan Plant.