CSC220 Lab11
Hash Tables

The goal of this week's lab is:

1. Practice using hash tables
2. Learn about the importance of debugging

Part 0

- Create a new project. Create a new Java project and call it **Lab11**.
- Create a package inside your project and call it **lab11**.
- Download the Lab11-Assignment11 ZIP folder from Blackboard and copy in the following files:
    - **HashEntry.java**. This class provides the implementation of a single element inside a hash table for you. This class has two fields:
        - **element**: the actual value stored in the hash table
        - **isActive**: a flag that will be used to keep track of removed elements (deleted flag in the lecture)
    - **QuadraticProbingHashTable.java.** You will be working on developing methods for this class. This class represents a hash table that will hold hash entries inside an array. This class has been partially implemented for you. The current implementation provides the fields, the signature of the methods you are about to implement, and a toString method to print the hash table. The toString method prints the value if present followed by "T" or "F" depending on isActive flag. If there is no element stored in a specific position in the table, the toString method will print "e" (i.e. empty).

Part 1 – The problem

For this lab, you are asked to write methods to be added to

QuadraticProbingHashTable.java. As the name of the class suggests, you are about to implement methods for a hash table that uses quadratic probing when collision happens. The hash table is implemented as an array called **HashTable** that holds objects of type **HashEntry**. A **HashEntry** is a class that has two fields: one is the **element** which holds the value stored in the hash table and the other is called **isActive** which is a flag to keep track of elements removed from the hash table. The implementation of the **HashEntry class has been provided for you.** Take a couple of minutes before you continue to get familiar with the implementation of this class.

The HashTable class has been started for you. The current implementation provides the field, the signature of the methods you are about to implement, and a toString method to print the hash table. **Remember that you ARE NOT allowed to change the signature of the methods.** However, you are allowed to use helper functions if you need to.

For debugging purposes, you might want to check the structure of the hash table. We provided a simple toString method for you that simply prints the hash table along with the isActive field (i.e., deleted flag). Before you move on to the next part, take a couple of minutes and look at the definition of both classes. Make sure you know what fields they have, how you can instantiate from each of them, and how to call various methods.

For this lab you are required for implement the following methods:

1. **public QuadraticProbingHashTable(int size)**
2. **public int hash(int value, int tableSize)**
3. **public void insert(int x)**
4. **public void rehash()**

Use the following online tool that nicely implements quadratic hashing and make sure you understand how quadratic probing works:
https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html

Part 2 – QuadraticProbingHashTable constructor

The signature of this constructor should be:

public QuadraticProbingHashTable(int size)

As the signature of this method suggests, this is a constructor for this class. This constructor will initialize the status of an object from this class based on the input parameter (i.e., size). So, this function will **create the array HashTable** with the specified size and initialize all of its elements to be null. You need to be careful about whether any other field needs to be initialized at this stage (hint: think about what the value of **currentSize** should be).

Part 3 – hash method

The signature of this method should be:

public int hash(int value, int tableSize)

This method accepts an integer value and returns the hash value based on the table size. Note that you are supposed to use the modulo (mod) operator as we saw in class for your hash function. That means the hash value you are returning is going to be a number **between 0 and tableSize-1** inclusive. The input value can be positive or negative. (-x and x should hash to the same value) **This function is not supposed to handle the probing**. Collision handling and probing should be part of your insert function (or another helper function).

Part 4 – insert method

The signature of this method should be:

public void insert(int x)

This function accepts a value (called x) and inserts it into the hash table. If the item is already in the table, this method does nothing and returns. **You need to resolve collisions using quadratic probing.** It is highly recommended that you write a helper function to do this (but this is not mandatory). Note that other fields of this class have to be updated accordingly after the insert. **If the load factor (percentage full) of the table is 0.75 or higher after the insert**, this function should perform rehashing as instructed below.

For this lab, check the load factor first (assuming that the value will be added to the table), if rehash is required, rehash first and then insert the value.

Part 5 – rehash method

The signature of this method should be:

public void rehash()

This function will increase the size of the hash table by a factor of two and rehash the elements into the new hash table (your insert method should call the rehash function as soon as the load factor (percentage full) of the hash table is equal or greater than **0.75**).

Part 6 – Test your code

As usual you need to test the functionality of the methods you have implemented. A set of test cases has been provided for you as part of **QuadraticProbingHashTable.java**. Uncomment the lab portion of the tests and run the main function. If you see any red text that says “TEST FAILED”, you need to debug your code.

How to debug your code?

1. Use the Eclipse debugger you learned about during the first lab.
2. If you see JavaStackOverflow, that means that you have an infinite recursive call and your recursive call is filling up the “call stack” (we talked about this concept in class). Go back and debug the method that is causing the problem.
3. Infinite loops! How would you know you have an infinite loop? As you should know from CSC120, if you have an infinite loop in your code, your code will not stop running. An easy way to inspect that in Eclipse is to look at your console window. If there is a red square (instead of grey) stop button, then your code is still running and although sometimes this could mean that your code just takes a long time to run, it probably indicates and infinite loop in this class.

Don’t forget: lab is due Thursday @ 11:59pm!