Data Structures and Algorithms

Assignment #5

Due: Program Assignment part Monday May 15th 11:00 am

This programming assignment will deal with hashing. It is very similar to, and shorter than, Assignment #4. Instead of using splay trees you are going to use *hash tables* with collisions resolved using *open addressing and linear probing.*

In addition to implementing methods for hash tables, the goal of the assignment is to see the influence of the load factor on the number of comparisons you have to make in order to find/insert entries in a hash table.

You are going to use the file **enroll.txt** of Assignment #4 that contains student numbers as input data to fill a *hash table of capacity 101 entries.* Each entry, like in Ass #4, is a pair (key, value) i.e., (student number, number of courses). Recall that each student number is a string of 7 digits (e.g., 9703456), one per line of the file. The same student number can appear several times in the file. The file terminates with a student number of 0000000 that should not be entered in the hash table.
You have to define your own hash function.

Besides the *hash(k)*, *size()* and *isEmpty()* methods, you should implement:

- *find(k)* which looks for the entry in the table with key k and if found returns the value x found in the entry. If *find(k)* fails, it should return the value 0.
- *Insert(k)* If there is no entry in the table with the key k, an entry with this key should be inserted and the *value* field should be set to 1. *Collisions should be revolved using open addressing and linear probing.* (no chaining!). If there is already an entry in the table with the key k, the *value* field of that node is incremented by 1.
- We give you a break and don't ask for a *delete(k)* method although it does not present any difficulty.
- *stats()*. This method is going to report statistics on the insertion of the keys from **enroll.txt**. Specifically, you should print out:
  1. The load factor of the hash table after you have entered all the student numbers.
  2. The total number of collisions that have occurred. This should be counted as follows. For each insert that enters a new key count:
     - If the entry given by the hash function is free, no collision
     - If the entry given by the hash function is already taken the number of collisions is the number of consecutive entries that are already taken. So if the entry following the one given by the hash function is free, the number of collisions for this insertion is 1. If the s entries following the one given by the hash function are taken, the number of collisions is $(1+s)$.

     For each insert that finds the key already there, count:
     - The number of comparisons that were made to find that the key was already there minus one.

Your first output should be the results of calling *stats()* when all keys from **enroll.txt** have been inserted.

You will then "find" the entries corresponding to the student numbers in **query.txt** and report the total number of collisions after all student numbers from *query.txt* have been processed. You should save the keys of the **query.txt** file in an (unsorted) array AQ[].

The number of collisions should be counted as for the case, i.e.:

- If the key is in the table, the number of collisions is the number of key comparisons minus one
- If the key is not in the table, it is the number of comparisons to detect that the key is not in the table minus one.

In order to see the influence of the load factor, you should repeat the process for a *hash table of capacity 199 entries*. For insertion in the new table *rehash* the contents of the first one. For the query part use the contents of AQ[]. Of course you will need a new hash function reflecting the new capacity of the table.

Print the new load factor, the number of collisions while rehashing, and the number of collisions while processing the entries stored in AQ[].

Do you think rehashing was necessary?

A template for this assignment will be posted soon.