CS130A Project 1 : Hash Table
Due: Friday January 29, 2016 at midnight

Introduction

Key-value Stores have recently become a very popular alternative to cumbersome databases for cloud based applications. Commonly, each data item has a uniquely identifying key, and a value. Values can vary in size and type based on the application. In this programming project, we want to build a data structure which allows us to search and retrieve efficiently (both in time and in space) particular values given keys, not necessary in an ordered manner. For such applications, hash tables come in very handy.

We will take the application of students' information retrieval for our hash table. The key-value pairs that will be used in this assignment are of the type (int, Student), where the int will be the 7 or less digits as the perm# which is **UNIQUE** for each student, and Student is a class defining each student in the university with name (string) and GPA (double).

```
1  #include <string>
2
3  class Student
4  {
5      private:
6      string name;
7      double gpa;
8  };
```

Project Statement

Your task in this project is to do the following:

- Implement a hash table that takes in a 7-digit positive integer or less as a key and stores both the key and the associated Student object.
- Set the initial TABLE_SIZE to be 5
- Collisions should be resolved through 2 following methods,
    1. Linear probing: When searching for key k, in linear probing algorithm we examine the cell with index $h = hash(k)$; and increment h until we find a value of h for which the cell is free or contains the given key.
    2. Double hashing: When searching for key k, in double hashing algorithm we first examine the cell with the same index as linear probing $h = hash1(k)$; nevertheless, another hash function is incrementally be added as follows:
    $h = \Big(hash1(k) + i * hash2(k)\Big)\%TABLE\_SIZE$;

** **Note:** i is the iteration time. First $i = 0$ and if there was a collision, we increment it while there is a vacant or deleted place.
** **Note:** Your program should input data from stdin (for instance with cin command in C++).
** **Note:** There is a possibility to change collision solving method by entering string "linearprobing" or "doublehashing" in the input. You can find it in example. In default, if there was no input regarding collision method, your hashtable should use linear probing policy. We consider in input, we merely modify the collision policy once in the beginning and do not change it in the middle of test case.

- Use the following hash functions:
  $hash1(k) = (k \ \% \ 492113) \ \% \ TABLE\_SIZE$
  $hash2(k) = (k \ \% \ 392113) \ \% \ TABLE\_SIZE$
  Think about why this hash function might be suitable for this application?

- Deletions should be supported through the **lazy deletion** method. You can change key with something that never is going to be placed there as a perm number (for instance -10).

- Dynamic Resizing should be supported such that the table size should be doubled if the load factor crosses over 0.7. Check if resizing is needed after inserting an element. When you resize, you should rehash the elements of the table in the order they are stored in the table (going from index 0 to index TABLE_SIZE-1)

Required Functionality

Your hash table is supposed to support the following operations:

- insert(key,value): insert an entry into the hashtable. Your program should return either "item successfully inserted" or "item already present"

- lookup(key): Use the hash table to determine if key is in the data structure and print its associated value and also the position separated by a single space (see the example below). Your program should return either "item found; [value] [position]" or "item not found" See sample output below. Positions should be array indexed (beginning with 0)

- remove(key): Use the hash table to determine where key is, delete it from the hash table. Your program should return either "item successfully deleted" or "item not present in the table"

- print(): Print out the hash table (in the array format). For empty spots in the hash table, don't print anything.

For consistency, you can assume that the sequence of operations is provided in ASCII format, with each command on a separate line. In addition, your program should print out (one line per command) information after each operation, as follows:

- insert(key,value): "item successfully inserted" or "item already present"
- lookup(key): "item found; [value] [position]" or "item not found" (for instance after finding Kevin in index 5th, the string would be "item found; Kevin 5")
- remove(key): "item successfully deleted" or "item not present in the table"
- print(): prints out the whole hash table on the same line as two tuples of the format without space (int,string,double) as follows:
  (1234,name1,gpa1)(2345,name2,gpa2)(3456,name3,gpa3)

  Note: Since GPA is a double variable, in print function, you should always write it with 1 floating point. For instance, we can have 3.5, 4.0 and not 4.
  In order to do this in C++, you can first include iomanip header in the beginning of your code as follows:

```
#include <iomanip>
```

  Then use the following code to print gpa:

```
cout << std::fixed << std::setprecision(1) << gpa;
```

  Needless to say, you should replace gpa with your variable name; however, the rest should not be changed.

Additionally, whenever the table is doubled there should be a "table doubled" string printed to stdout. As this event happens after inserts, the statement should be printed in the line after the output from the corresponding insert. For example in the sample input the table doubles after the third insertion and the corresponding insert statement is present on the next line after the output of the insert statement.

***Note:** When you make the size of table doubled, you have to find **the smallest prime number larger than the doubled size**. Namely, if the size of table is 5, after doubling, it should be 11 (the smallest prime number greater than 2 * 5). Also, before doubling, you have to check whether the new item already exists or not. If it already exists, you should not double the size of table. You merely double size of table when are certain that new item will be placed in the hashtable. Also, incontrovertibly after doubling, you have to rehash all items inside of hashtable.

Example

Although you should implement both, in the following, we describe an example merely for linear probing.
Sample Input

linearprobing
insert 8670959 asad 3.9
insert 7670931 victor 3.6
insert 7636338 omid 4.0
lookup 7636338
insert 5712195 jin 2.5
print
delete 4444444
delete 5712195
print
delete 7636338
lookup 8670959
delete 8670959
print

Sample Output

item successfully inserted
item successfully inserted
item successfully inserted
item found; omid 4
table doubled
item successfully inserted
(7670931,victor,3.6)(7636338,omid,4)(5712195,jin,2.5)(8670959,asad,3.9)
item not present in the table
item successfully deleted
(7670931,victor,3.6)(7636338,omid,4)(8670959,asad,3.9)
item successfully deleted
item found; asad 8
item successfully deleted
(7670931,victor,3.6)

The TA/Readers should be able to run your program as "prog1" with input from stdin. Be sure to include a Makefile, and name the executable prog1. The TA/Readers should be able to terminate your program by Ctrl+D (EOF for linux) or Ctrl+Z (EOF for windows).

The programs are tested automatically so make sure that you provide the output exactly as specified and shown above with no extra space or extra lines to the output. The program should output to stdout. The input provided above is the subset of the input your program would be tested against. So be sure to check your program exhaustively for different cases.

A suggestion while testing the program is to take the sample output in a file and then save your output (you can either redirect it to a file when outputting to stdout or copy from output screen) in a separate file and run a d iff command against the two outputs to make

sure they match exactly.

Note: Last but not least, this project should be implemented individually. Also, please do not share your codes with each other. Submission process will be announced on Piazza website shortly.

*****Note:** You should use "Main.cpp" file along with this project description and implement its functions. You may change it with your own function names or so. In fact, as long as it generates the expected output for the input file, is fine. We suppose that input file has the valid format. For instance we do not have "insert Alex". However, it is good to program defensively all the time.

Submission

In order to submit your codes, you should login to Submit System using your umail email and CS password. In project "CS130A Project 1 Hashing", press Make Submission button and then you can send 5 files with the following names:
"Main.cpp", "Hashtable.cpp", "Hashtable.h", "Student.cpp", "Student.h"
System has the ability that show your output and its difference with the desired one. Hence, you can debug your code and also get a sense of how your project is going to be graded.

Extra Points

Do a empirical test to compare the aforementioned collision solving policies with respect to time complexity. Plot a figure with increasing number of samples in x axis and time of insert and lookup all items in y axis. To compare them easily, draw both methods' trend in one figure. You can use MATLAB or any other software to plot figure. Give a succinct explanation regarding which method is better. The report should be submitted via Gauchospace containing a figure and a terse explanation.