CS 130A: Winter 2019

Programming Assignment 1

Assigned: Jan 17 **Due: Jan 31, Midnight 11:59 PM**

Your assignment is to implement a hash table, using Modulo Primes method, using Separate Chaining to resolve collisions. *You are expected to implement it using C++.* In a small number of cases, we will make an exception and let you use Python, but talk to the TAs about logistics of using non-C language.

1 Basic Assignment (80 pts)

The goal is to construct a universal hash function to map 32 bit IP addresses to a table of size roughly 256, using the fact that 257 is a prime. The input to your program is a file, containing a sequence of Hash Table commands: 'insert x,' 'delete x' and 'lookup x'.

The first line of the file will contain the desired size of the hash table (256 for the basic assignment). The last line of the input file will be a command 'stat', which asks you to print the data structure statistics. The lines inbetween will contain an intermixed sequence of insert, delete, and lookup commands, one per line.

The command 'insert x' adds x to the hash table *if x is not already in the table*; otherwise it flags an error message. Similarly, 'delete x' will remove x from the table if it is present; otherwise flag an error message. The 'lookup x' either returns 'x: found' or 'x: not found' depending on whether or not x is currently in the hash table.

At the command 'stat', you should print a summary as follows:

- The (space separated list of) coefficients of your hash function: a_1 a_2 a_3 a_4.
  Use the convention that a_1 is the coefficient for the *least significant byte* and a_4 the coefficient for the *most significant type* in the IP address.
- Number of items successfully inserted:
- Number of items successfully deleted:
- The number of empty slots in the hash table:
- The number of hash table slots with exactly 1 item:
- The maximum number of collisions in the hash table: (count, location)

The last statistic asks for maximum length of the chain at a hash table slot *at any time during the sequence of operations.* (This requires you to keep track of the collision chains and remember its maximum length during the course of the algorithm.) You should report both the length (the number of items that mapped to the same location at any time) as well as the location index in the table. (If there are multiple maxima, return the largest index location with the maximum length.)

You are expected to get the input file from the standard input, and you should **name your basic program "prog1A"**. Therefore, the following command should execute your program:
make
./prog1A <sample.txt

2 Advanced (20 pts)

Generalize the hash table construction to (roughly) any power of 2. The first line of the input specifies the (approximate) size of the hash table, namely, 2^b. For the target size 2^b, you should use the *smallest prime number* p that is larger than 2^b. (The list of primes can be found at the website:
https://www2.cs.arizona.edu/icon/oddsends/primes.htm)

Each input key x is then divided into groups of b bits, namely, $x_1, x_2, \ldots, x_k$, starting from the least significant bit. Using the Modulo Primes method, your hash function will have the form

$$\sum_{i=1}^{k} a_i x_i \pmod{p}$$

where a_i are the random integers in the range $[0, p-1]$.

For instance, if $b = 11$, then the 32-bit IP addresses are divided into three groups of $10, 11, 11$ bits, where the most significant chunk x_3 has 10 bits. The rest of the hash table construction remains the same as in the basic assignment. On the command stat, your program will again list the 8 statistics of the hash table.

Name your advanced program "prog1B" so that your code should work with the following command.

make
./prog1B <sample.txt

3 Input Format and Test Data

The input file will be structured like this:

- The first line will indicate the target size of the hash table. This value is **256** for the basic part, and 2^b for some integer b for the advanced part.
- The last line will always be the **stat** command for both parts of the assignment.
- The lines in between will include numerous hash table operations such as **insert**, **delete**, or **lookup** with the IP addresses.

The following is a sample input file :

```
256
insert 121.180.184.61
lookup 92.98.71.256
insert 92.98.71.256
insert 79.175.14.252
lookup 92.98.71.256
delete 92.98.71.256
lookup 92.98.71.256
stat
```

And the corresponding output for this input file is :

```
92.98.71.256: not found.    // the IP has not been inserted yet.
92.98.71.256: found.        // it is found, because it had been inserted before the lo
92.98.71.256: not found.    // it has been deleted, therefore not found.
10 89 210 162               // a1 a2 a3 a4
3                           // inserted 3 IP addresses.
1                           // one of them was successfully deleted.
255                         // the number of empty slots.
0                           // the number of slots that has exactly 1 element.
2 115                       // the max number of collisions is 2, and it is
                            // in the 115th bucket.
```

The output structure is as shown above, and it is the same for both the basic and the advanced parts, except that in the advanced part the number of coefficients are not necessarily 4. In that case you should print all the coefficients you have.

4 Testing and Grading

1. Your programs will be graded as follows. We should be able to run your program as `./prog1A < input.txt` of `./prog1B < input.txt`.
2. We will provide you a set of test input, along with corresponding correct output.
3. If your program handles the provided test cases correctly, you will receive 50% of the points automatically.
4. To receive the remaining 50% of the points, your program must handle new (unseen) test cases created by us.

5 Submission Instructions

1. Include a **README** file. This *must* contain the following information in the given format.

```
CSIL login :  [Your username here], UCSB Email :  [Your ID]@umail.ucsb.edu
```

This is needed to map your CSIL usernames to your records.

2. Please state in the README file if you have done the advanced part of the assignment.
3. Include a ‘`makefile`’, so that running ‘`make`’ compiles your source files and generates the executable(s) ‘`prog1A`’ and ‘`prog1B`’.
4. In order to electronically turnin your project files, use the following command:

   ```
   turnin prog1@cs130a LIST
   ```

   where LIST is the list of all your files, i.e., your makefile, all your source and header files. (The turnin program is located in `/usr/bin/turnin` on `csil.cs.ucsb.edu` only.) If you turnin more than once then the version that will be graded is the last version you turned in before the deadline.
5. After you finish the homework, please make sure that it works on CSIL computers with the given inputs and gives the same outputs.