Perfect Hashing

Objectives

The objective of this programming assignment is to increase your understanding of not just Java, but also hashing by implementing a perfect hashing algorithm and by collecting statistics on its performance that you might not have understood in your Data Structures course. **This project is to be done individually**.

Introduction

In this project, you will implement the perfect hashing scheme described here. We summarize the main ideas here. You should consult your textbook or other resources for details and proofs.

First, perfect hashing is hashing without collisions. The wiki page shows that if you hash n items into a table of size n^2 with a randomly chosen hash function (separate chaining, quadratic probing, cuckoo, among others), then the probability of not having any collisions is greater than 1/2. What happens if you are unlucky and you get a collision? Then, pick another hash function and try again. With independent trials, the expected number of attempts you have to make in order to achieve zero collisions is less than 2.

Now, a table of size n^2 is really big and a huge waste of memory. So, in our perfect hashing scheme, we don't directly hash to a table of size n^2. First, we hash into a *primary hash table* of size n. There will be some collisions. To resolve the collisions at a slot of the primary hash table, we create a *secondary hash table*. If t items collide at a certain slot of the primary hash table, then we create a secondary hash table of size t^2 and use perfect hashing to store the t items. The resources given shows that the expected number of slots used by all of the secondary hash tables is less than $2n$. If you are thinking that this hashing scheme is just separate chaining with the linked lists replaced by hash tables, that is pretty close — just remember that the linked lists are replaced by *collision-free* hash tables.

How do you search for an item in this hash table? You have to hash twice. First, you hash the item to find its slot in the primary hash table. If that slot is not empty, then you find its slot in the secondary hash table. If the slot in the secondary hash table is also non-empty, then you compare the item against the item stored in the secondary hash table. If there's a match, you found the item. Otherwise, the item is not in the hash table.

Note that each secondary hash table has its own hash function, since it might have been necessary to try a few hash functions before you found one that did not result in any collisions. So, the hash function would have to be stored in the secondary hash table.

The perfect hashing scheme described above requires the ability to "randomly pick a hash function." In particular, we have to be able to randomly pick a *different* hash function if the one we just tried doesn't work because it resulted in a collision. How do we do that? This is accomplished by "universal hashing". (Never mind the word "universal". It is a bit of a misnomer. It should really be called "randomized hashing", but most people think hashing is already random, so "randomized random" doesn't make much sense either.)

The universal hash functions presented in Section 5.8 of our textbook provide a method for generating random hash functions. First, we need a prime number p that is larger than any key that will be hashed. Then, we select two random integers a and b, such that $1 \le a \le p - 1$ and $0 \le b \le p - 1$. Then, we can define a hash function $h_{a,b}($ $)$ using these two random integers:

$$h_{a,b}(x) = ((a x + b) \bmod p) \bmod m$$

where m is the table size (which does not need to be prime in this scheme). Thus, for every pair of a and b, we get a hash function. The proof in the resources shows that random hash functions chosen this way satisfies the definition of "universal" and has provably good performance.
To hash strings we first convert the string into a number, then we use the hash function above to guarantee "universality". In one scheme, we pick a random constant c such that $1 \le a \le p - 1$. Then, we interpret each character of the string as a number (think ASCII), so a string str becomes a sequence of numbers d[0] d[1] d[2] d[3] ... d[t]. Now we can convert the string into a number:

$$g_c(\text{str}) = (\text{d}[0]\, c^{t} + \text{d}[1]\, c^{t-1} + \text{d}[2]\, c^{t-2} \ldots + \text{d}[t]) \bmod p$$

In a program, we should calculate this value using Horner's rule. Also, we would have to make sure that the arithmetic does not result in any overflows. (This can be accomplished by "modding out" by p at every step.)

The last remaining thing to point out is that this perfect hashing scheme only works if we know all the keys in advance. (Otherwise, we cannot tell how many items hash into the same slot of the primary hash table.) There are several applications where we would know the keys in advance. One example is when we burn files onto a CD or DVD. Once the disc is finalized, no additional files can be added to the disc. We can construct a perfect hash table of these filenames (perhaps to tell us the location of the file on the disc) and burn the hash table along with the files onto the disc. Another example is place names for a GPS device. Names of cities and towns will not change very often. We can build a perfect hash table for place names. When the GPS device is updated, a new hash table will have to be constructed, but updates are not frequent events. This last example is the basis of your programming project.

Assignment

Your assignment is to apply the perfect hashing scheme described above to a file containing approximately 16,000 city names in the United States. The data comes from geonames.org. In addition to the names of all the cities with population above 1000, the file also has the latitude and longitude of each of these cities. Here are a few lines from the file:

```
Abington, MA

42.10482 -70.94532

Abita Springs, LA

30.47853 -90.03758

Abram, TX

26.1998 -98.41113

Absarokee, MT

45.5205 -109.44294
```

The data for each city is stored in two lines of text. The first line is the name of the city followed by the state. The second line of text has the latitude and the longitude of the city (in that order). You should treat the city name and state as a single entity to be hashed — i.e., hash the string "Abington, MA" (comma included) rather than "Abington". You may assume that the city names with the state designation included is a unique string, even though this is not entirely true in real life. (For example, there are 3 separate cities in Indiana named "Georgetown". Two of these have been removed from our file.)

The complete file is here: US_Cities_LL.txt.

Since implementing universal hash functions can be a little bit tricky, a Java class that implements universal hash functions is provided: Hash24.java. The 24 in Hash24 indicates that the methods in the class work with values as large as 2^{24} which is approximately 16 million. This is more than large enough for the purposes of this project. To use the Hash24 class, simply create a Hash24 object and then use it to invoke the universal hash function:

```
Hash24 h1 = new Hash24() ; ... index = h1.hash("Abington, MA") ;
```

The Hash24 object h1 "remembers" the randomly chosen *a*, *b* and *c* used in the universal hash function. So, you can store h1 and retrieve it later and it can be used to compute the same hash function.

You will implement two separate programs. The first program takes the file US_Cities_LL.txt, creates a hash table using perfect hashing and stores the hash table in a file. While creating the hash table, the first program must also print out some statistics about the hash table (see below). The second program reads in the hash table from a file and lets the user query the hash table. If the city is found, the second program prints out the latitude and longitude as well as a URL that can be pasted into a web browser to locate the city in Google Maps.

Details

CityTable

Your hashtable class must be called CityTable and live in a package called hash341 (so that it will be compatible with the Hash24 class). In addition, your CityTable class must implement these methods:

- A constructor that takes the name of a file and a primary hash table size:
  `public CityTable (String fname, int tsize) ;`
  This constructor should use the information in the file to construct the hash table using the

perfect hashing scheme described above. The size of the primary hash table should be tsize. While constructing the hash table, this constructor should print out the following

  - a dump of the hash function used.
  - number of cities read in.
  - primary hash table size.
  - maximum number of collisions in a slot of the primary hash table.
  - for each *i* between 0 and 24 (inclusive), the number of primary hash table slots that have *i* collisions.
  - all the cities in the primary hash table slot that has the largest number of collisions. If there is more than one such slot, pick one arbitrarily.
  - for each *j* between 1 and 20 (inclusive), the number of secondary hash tables that tried *j* hash functions in order to find a hash function that did not result in any collisions for the secondary hash table. Include only the cases where at least 2 cities hashed to the same primary hash table slot. (I.e., we exclude the primary hash table slots that did not have any collisions from the calculations.)
  - The average number of hash functions tried per slot of the primary hash table that had at least two items. (As before, we exclude the primary hash table slots that did not have any collisions from the calculations.)

- Here is some sample output for you to compare.
  Note that this constructor cannot begin constructing secondary hash tables until all of the data have been read in. So, construction of the hash table takes two passes. The first pass reads in each city from the file and figures out where it belongs in the primary hash table. The second pass looks at each slot in the primary hash table and creates a secondary hash table for each slot where this is needed.
- A find() method:
  ```
  public City find(String cName) ;
  ```
  The find() method should return a reference to a City object found in the hash table. You should implement the City class in the hash341 package and make sure that it includes the following public data members. The rest of the City class is up to you.

```
public String name ;
public float latitude ;
public float longitude ;
```

- A writeToFile() method that stores the hash table in a file with the given filename using Java's writeObject() method.
  ```
  public void writeToFile(String fName) ;
  ```
  You can find an example of using writeObject() here. Note that writeObject() will write the entire hash table to a binary file in one step. The writeObject() method will also recursively follow all references in an object and write the objects that are referenced as well. All objects written out by writeObject must belong to classes that have declared implements Serializable. For example, the Hash24 class has such a declaration. The Serializable interface is an example of a *marker interface*. You do not need to implement any methods to declare that the class implements Serializable.

- A readFromFile() method that will read an entire CityTable back from the file with the given filename using Java's readObject()method:
  ```
  public static CityTable readFromFile(String fName) ;
  ```
  You can find an example of using readObject() here. Note that readFromFile() must be a static method because we do not yet have a CityTable object until we have created one from the file. Thus, readFromFile() must be invoked using the CityTable class name.
  Your readFromFile() method must be compatible with the test program, ProjTest.java, which should produce output that looks like: ProjTest.txt.

Other Classes

You will probably find it necessary to implement other classes (hint, hint). You should make sure that your collection of programs plus ProjTest.java compile on our Linux server.

First Program

The first program that you implement should read in the text file US_Cities_LL.txt, construct a hash table using perfect hashing, print out the statistics described above and write out the hash table to a file named US_Cities_LL.ser.

Second Program

The second program that you implement should read in the hash table from the file named US_Cities_LL.ser and prompt the user to type in a city name (including the state). If the city is in the hash table, your program should print out the city's name and coordinates and also a URL that can be cut-and-pasted into a web browser.

Running your second program should look somewhat like this: ProjTypescript.txt.

If you wish you can optionally include code that asks the system to open the Google Maps URL in the default browser. (See sample code below.) Make sure that the program still compiles and runs on GL using a terminal emulator and prints out the Google Maps URL to standard output in all situations. Otherwise the graders would not be able to check your program.

```
import java.awt.Desktop ;
import java.net.URI ;
```

```
...

static void openURL (String url) {
// cribbed from:
// http://java-hamster.blogspot.com/2007/06/troubles-with-javaawtdesktop-browse.html

try {
if (Desktop.isDesktopSupported()) {
      Desktop desktop = Desktop.getDesktop();

      if (desktop.isSupported(Desktop.Action.BROWSE)) {
            desktop.browse(new URI(url));
      } else {
            System.out.println("No browser. URL = " + url) ;
      }
} else {
      System.out.println("No desktop. URL = " + url) ; }
} catch (Exception e) { e.printStackTrace(); }
}
```

What to Submit

You should copy over all of your Java source code and have your .java files in their own directories which are in turn under the src directory. Then zip the entire directory structure, naming it Project1.zip. Then submit it through the course website.

Warning

We will be making some requirement changes as we get closer to the deadline. The project will not change or have additions. You will be required to use some Java components