Homework # 11
due Monday, November 25, 10:00 PM

In this homework, you will a create a variant of the `Map` ADT that represents friendships between profiles. We will use the technique of double hashing to resolve collisions.

1 The Profile Map

Each profile on a social media platform may have a friend relationship with many other profiles. We will store these relationships in a map. The `ProfileMap` ADT stores associations between `Profile` objects (the keys) and `List`s of `Profile` objects that are friends (the values). Friendships here are not symmetrical, as `a` can have `b` as a friend, while `b` does not have `a`.

This new ADT is a variant of the `Map` ADT. It does not have a put method. We do not directly support replacing a value for a given key. Instead, we allow the user to add new profiles to the list of friends. This can be achieved by using the `get` operation to obtain the `List` and then `add` to it. We also support an `add` operation that adds a new profile to the map, and another that adds a new association.

Whenever a `Profile` object is passed to a `get` or `add` operation, it will be added to the map as a key. In this way, the map is trying to keep track of all the profiles it knows exist. The `find` operation does not assert the existence of the profile, and thus does not add it the the map if it does not exist already. However, profiles can be added to the value lists directly without being added as keys in the map.

2 Concerning Hash Tables

For this assignment, you will implement a special-purpose hash table that maps `Profile`s to lists of related `Profile`s. The hash table will use the idea of double hashing. Please read the textbook, especially sections 11.2–11.4, pages 581–602 (3rd ed., pp. 569–590). Alternatively, Wikipedia (https://en.wikipedia.org/wiki/Hash_table) has a nice introduction as well, pay special attention to the "Collision resolution" section. Unlike the textbook, our hash table will increase in size when the table has too many elements in it.

In Java, every object has its own hash function (inherited from the `Object` class, and often overridden). If two different `Profile`s are equal (using the normal `equals` method) their hash code will be the same. Conversely, if they are not considered equal, they should *ideally* hash to different values. There is, however, a problem with generating a unique index for all unique objects: it's *very* difficult (often impossible) to do! Indeed there are usually more possible objects than spaces in any practical array and so we must resolve collisions somehow.

We will use "double hashing" which is explained in the textbook on pp. 596 (pp. 584 in the 3rd. ed.). The second hash function is as Knuth recommends and as explained on the top of page 598 (586 in the 3rd. ed.). Please also do "self-test" exercise 9 on page 598 (586) to make sure you understand double-hashing. The answers are given in the textbook on p. 611(p. 596), but (1) don't look until you have completed the *entire* exercise, and (2) don't just accept them; make sure you understand how the answers are computed!

3 Implementing the ProfileMap ADT

This ADT is a special purpose Table that has a special set of methods:

size() Return the number of entries.

find(Profile) Return a list of profiles that are friends with the given key (profile). Returns null if the key is not in the table.

get(Profile) Return a list of profiles that are friends with the given key (profile). If there was no association, create a new one with an empty list. This method never returns null.

add(Profile) Create a new empty association for the given profile. If an association already exists, return false.

add(Profile, Profile) Add the second profile to the first profile's friend list. If either profile is not yet a key, create an association for that key first. If nothing in the map changes, return false.

remove(Profile) Remove any association for this given key (profile). Returns the former association (possibly null) of the key.

getAll() Return a list of all the keys in the map.

Since profiles are mutable, the ADT will clone every profile from the client stored as a key in the table. It will also clone whenever returning a key. The goal is that no reference to an object serving as a key will be exposed to the client. This way, the entries will stay in the correct place in the table even if the client attempts to mutate a `Profile` object. So make sure that the `Profile` objects in your value lists are not the same as the ones that are keys. The value lists will be exposed!

An interesting aspect of the implementation is that the hash table entries inherit from `ArrayList` which makes it somewhat simpler to return the list associated with a key; the entry itself is that list. In essence we have something like an endogeneous table: the entry is the value returned. The key is stored in an extra field added to the list object.

3.1 Concerning place holders in the table

When an entry is removed, we need to leave a place holder in its place so that elements that previously were pushed off because of collisions will know to continue looking. On the other hand, if you are adding a new entry to the table, then you can overwrite a place holder.

We use a special unique place holder object with a null key; no other entries are allowed to have null keys.

3.2 Concerning rehashing the table

The table should initially have an array of size 7. When the number of used spaces (place holders and real entries) in the map becomes more than `CROWDED` times the capacity of the array, a new array should be allocated and all entries should be "rehashed". The new array should have size at least $4n$, where n is the number of (true) entries, but in order to ensure that double hashing works, the size of the table must be the larger of twin primes, and never less than the initial capacity (7). We provide a `Primes` class to help you locate twin primes.

Once the new array is allocated, all entries (other than place holders) need to be placed in their correct places in the new table, which are probably different than before because the array size has changed.

4 What you need to do

You need to complete the implementation of `ProfileMap`. We give the required data structure to use (don't add or remove from the fields already declared for you). You need to implement the invariant checker and use it in assertions for all public methods.

5 Files

The repository for this homework assignment includes the following files:

src/TestInternals.java Tests for the invariant checkers.

src/TestProfileMap.java Tests for the `ProfileMap` class.

src/TestEfficiency.java Efficiency tests for the ProfileMap class. (Each test should take a few seconds)

src/UnlockTest.java Class to unlock the locked tests.

src/edu/uwm/cs351/ProfileMap.java Skeleton file for the main ADT.

src/edu/uwm/cs351/Profile.java The profile data type.

src/edu/uwm/cs351/SocialApp.java An example application.

src/edu/uwm/cs351/util.Primes.java Utility class for prime numbers.

logs/cs351Log Your log file.

ProfilePictures/*.jpg Image files for use in the application.