DATA STRUCTURES
HOMEWORK 3

1 Implementing a simple hash set

Our goal for this assignment is to implement a *hash table* that stores and retrieves unique keys. In keeping with Java nomenclature, we will refer to this implemented structure as a *hash set*, distiguishing it from the *key-value storage* of a *hash map*.

The interface: We will be implementing the `SimpleHashSet<E>` interface, which must implement the following methods:

- `public boolean insert (E key)`
  If it is not already present, add `key` to the set. Return whether `key` was added.
- `public boolean lookup (E key)`
  Return whether `key` is present in the set.
- `public boolean remove (E key)`
  If present, remove `key` from the set. Return whether `key` was removed.
- `public int size ()`
  Return the number of keys in the set.
- `public int getNumberCollisions ()`
  Return the number of collisions that have occurred so far while inserting values into the set.

2 The code

Get started by creating yourself a directory for this project and grab some source code:

`https://bit.ly/AMHCS-2019F-211-hw3`

Unzip the code, and you will see that `SimpleHashSet.java` defines the interface described above. Here are the other files:

2.1 Your hash set

The file `AmhHashSet.java` is the skeleton of a class that implements the `SimpleHashSet` interface. **Your assignment**, in short, is to implement this class. Add whatever data members and supporting methods you need to store the requested values in your hash table.[1] You may use either *open addressing* or *chaining*.

Note that the constructor accepts as `capacity` parameter. You should use this value to determine the size of the array that will store your values. Do not worry about dynamically re-sizing the array under load. If you implement chaining, the number of collisions may get high if the array is relatively small; if you implement open addressing and fill the table, your code may need to throw an exception if n exceeds m.

Notes on arrays and generics: Java's arrays do not like generics (that is the unspecified `<E>` of many container classes). You cannot create an array of pointers to `E`; you would, instead, need to create an array of `Object`, and then write code to cast `Object` pointers to `E` pointers, which Java will allow.

Notes on hash functions: The `Object` class, at the very top of the class hierarchy, defines the `hashCode()` method. Thus, every Java object, regardless of type, has this method. However, this method doesn't generate a good hash for every type; indeed, for the `Integer` class (which we will use in this assignment), the `hashCode()` method returns the `int` value itself. That is, the hash function is: $h(x) = x$. This is a lousy hash function, but combined with a modulus operation, it will still allow you to implement a hash table with a mediocre spread of values. Feel free to implement just about any kind of better hash function. Use the Google, and see what you find, but don't get bogged down in it.

2.2 The tester

`HTTest.java` is used to test `AmhHashSet`. Specifically, it reads, from a file provided on standard input, a list of *insert*, *lookup*, and *remove* operations, along with their expected outcomes. It then compares the expected outcome with the one produced by an `AmhHashSet`, and shows the result. In the end, it shows the number of collisions that the `AmhHashSet` reported.

The list of operations and expected outcomes looks like this:

```
lookup 3 false
insert 5 true
insert 5 false
lookup 5 true
remove 17 false
remove 5 true
lookup 5 false
```

[1] It should go without saying, and yet I will say it: You must implement your own hash table within an array that your code manages. Using pre-existing hash table/set/map classes is not allowed.

If you put a list of operations like this into a file (e.g., `simple-test.txt`), then you can run `HTTest` like so:

```
$ java HTTest
USAGE: java HTTest <storage size>

$ java HTTest 10 < simple-test.txt
lookup          3 -> false : match
insert          5 ->  true : match
insert          5 -> false : match
lookup          5 ->  true : match
remove         17 -> false : match
remove          5 ->  true : match
lookup          5 -> false : match
Total collisions = 0
```

2.3 The test generator

You can write and modify such test files by hand, putting your code through its debugging cases to find simple problems. Ultimately, though, you should test your code on a larger, automated input. `HTTestGenerator` will create a random test sequence like the one above:

```
$ java HTTestGenerator
USAGE: java HTTestGenerator <range> <test length>

$ java HTTestGenerator 20 15
remove 8 false
insert 11 true
insert 6 true
remove 11 true
insert 9 true
lookup 5 false
remove 6 true
insert 18 true
insert 8 true
remove 18 true
remove 8 true
insert 19 true
remove 19 true
insert 13 true
remove 9 true
```

Of course, a test case of values from 1 to 20, with a total of 15 operations, is not large. Generating a larger one is desirable, but the output should be redirected into a file:

```
$ java HTTestGenerator 1000000 25000 > big-test.txt
```

The file `big-test.txt` (which you can open your text editor) will be 25,000 lines long, and use keys from 1 to 1,000,000. If you then run this through `HTTest` (again redirecting the output so that it doesn't all appear on the console)...

```
$ java HTTest 10000 < big-test.txt > big-test.log
```

...the `big-test.txt` file will be processed using a 10,000 entry array, and its results dumped into `big-test.log`. You can open this latter file in a text editor to examine it, but some command line tools can help search it for interesting things first. Specifically, we are likely want to know whether any operations returned incorrect results, so we can do this:

```
$ grep MISMATCH big-test.log
```

If any mismatches between the expected result and the actual result occurred, lines with the `MISMATCH` string will be shown. If nothing is printed, no lines contained that string, and all of the operations did the right thing. We may also want to know how many collisions occurred:

```
$ tail -n 1 big-test.log
Total collisions = 311
```

Test your code until you're convinced that it's working correctly. Run it with different array sizes and see how the collision numbers change.

3 How to submit your work

Submit your `AmhHashSet.java` source code using the CS submissions web page:

`https://www.cs.amherst.edu/submit`

This assignment is due on Friday, Nov-01, 11:59 pm.