COSC-211: Data Structures
HW6: Hash tables

Due Friday, November 4, 11:59 pm

1 Implementing a simple hash set

Our goal for this assignment is to implement a *hash table* that stores and retrieves unique keys. (Unlike last week's binary search tree implementation, we will *only* store keys this time).

The interface: We will be implementing the `AmhHashSet<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. A collision occurs any time the key that you're adding hashes to a non-empty position in the table.[1]

2 Getting started

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

`https://bit.ly/cosc-211-f22-hw6`

Unzip the code, and you will see the following files:

- `AmhHashSet.java`
  The interface that defines the interface given above.

[1]This is not a standard part of a hash table interface. It's something we're using for our assignment.

- `WrapperList.java`
  A wrapper around the standard Java `HashSet` class, making it conform to the `AmhHashSet` interface. This class serves as our reference standard that we assume to function correctly.

- `ChainedHashSet.java`
  A skeleton class that **you will complete**. This class should implement a *hash table with chaining* to store its values, thus implementing the `AmhHashSet` interface.

- `HTGenerator.java`
  The program that creates a sequence of pseudo-randomly chosen *hash set* operations, along with random (but reasonable!) values. This generator also emits what the correct result for each operation should be.

- `HTTester.java`
  The program that reads in a sequence of *hash set* operations and then applies them to a `ChainedHashSet`, comparing the results from that object to the correct results read in with each record.

2.1 Your hash set

Your assignment, in short, is to implement `ChainedHashSet`. Add whatever data members and supporting methods you need to store the requested values in your hash table.[2] You are encouraged to use the standard Java `LinkedList` class to implement the chains at every entry. You do not need to implement any array resizing, but you are welcome to if you wish.

About 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 trivial 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, but don't spent too much time on it.

2.2 The tester

`HTTester.java` is used to test `AmhHashSet`. Specifically, it reads 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.

[2]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.

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
```

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

```
$ java HTTester
USAGE: java HTTester <storage size> <input pathname> <output pathname>

$ java HTTester 10 simple-test.txt simple-test.log
```

If you then open `simple-test.log`, you will see something like this:

```
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. `HTGenerator` will create a random test sequence like the one above:

```
$ java HTGenerator
USAGE: java HTGenerator <range> <test length> <output pathname>

$ java HTGenerator 20 15 simple-test.txt
```

The result, in `simple-test.txt`, looks like this:

```
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 once your code handles the smaller one well:

```
$ java HTGenerator 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 `HTTester`…

```
$ java HTTester 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. You can search for the string `MISMATCH` to find any errors. You can also examine the last line, which reports the *number of collisions* that your hash table incurred during the run.

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

Go to GradeScope for our course, where you can submit your work. It will be auto-tested, and you will see whether it *compiles* and *runs* successfully. Again, if the run fails, it won't tell you why; you need to go back and do more testing yourself. You may submit early and often!

You only need to submit your `ChainedHashSet.java`.

This assignment is due on Friday, November 4, 11:59 pm.