Phone Book

The ***closed hashing**** approach seen previously resolves hashing collisions within the table (array) itself with a variety of collision resolution techniques (e.g. linear probing). An ***open hashing*** approach uses an array of linked lists of table entries, rather than an array of table entries. Collisions will simply be appended to the end of the list, and collision resolution is as simple as traversing the list.

> *Confusingly, you will see *closed hashing* referred to as *open addressing* (or see them used interchangeably). The reason for this is that, with closed hashing / open addressing, the location of an object in the hash table is not completely determined by its hash code. In other words, the potential address is *open*, while the *location* of the object is *closed* inside the table itself. *Closed hashing* and *open addressing* are therefore synonyms. *Open addressing* and *open hashing* are antonyms.

Open hashing trades some extra storage overhead (all the lists / objects required) for better performance (compared to closed hashing).

Phone Book

In the olden days, there existed clandestine *books* that conveniently stored a database of people and their phone numbers, in case you needed to call someone but didn't have their phone number memorized. Every so often, the books were updated and re-distributed. It was great at the time, but we have more sophisticated technology now. They're still useful though! Not to actually use, but as a teaching tool, because they're essentially a physical manifestation of a hash table. Google it if you don't know what a phone book is.

Create a class **PhoneBook.java** that will implement a hash table of <Person, PhoneNumber> pairs (you will need to write these classes). The PhoneBook class should again have an array of Entry objects; this time, an Entry should maintain a reference to the next (possible) Entry at a specific location, thus making it a linked list. You could of course use an array of java.util.LinkedList objects, but writing your own linked list code will be good review (thankfully it won't require a complete linked list class, just a few linked-list-type algorithms).

Your PhoneBook class should implement the IMap interface (supplied), which has the following methods:

- `PhoneNumber put(Person key, PhoneNumber value)`
  - Returns the previous value mapped to this key, should it exist, or `null` if it is a new key
- `PhoneNumber get(Person key) //perform a lookup`
- `PhoneNumber remove(Person key)`
  - Remove supplied key mapping, returns value previously mapped to the deleted key (or `null`)
- `int size() //return current number of key/value pairs in the table`

The supplied types will now be responsible for providing hash functions. Thankfully, Java ensures that *every* class has an `int hashCode()` method, inherited from Object. However, Object's implementation (in general) gives unique objects unique hash codes - probably not a good idea if you want a custom type to be usable in a hash table.

The Person and PhoneNumber classes should override the `hashCode` method (in addition to the `equals` method). These classes are responsible for hashing themselves. Answer these questions before you continue:

1. True or false: All "equivalent" objects (their `equals` method call returns true, however that might be implemented) should have the same hash code.

2. True or false: All objects with the same hash code should be "equivalent" (a call to their `equals` method should return true).

The hash table's job is to ensure everything fits inside the array, resolve collisions, and do all the bookkeeping.

Thoroughly test your hash table before proceeding (this might be a good time to learn how to use the JUnit test framework on your own; you used JUnit testing in the TwentyQuestions lab). Remember that the goal of a good hash function is a uniform distribution, with as few collisions as possible, while maximizing table usage. There is no right or wrong way to write a hash function, you'll have to be creative. Don't forget that hash functions should be quick and easy to compute.

Once you have tested your hash function you will use the data in the **"White Pages.txt"** file that contains names and IDs. Populate your hash table with the data then test your `get(Person key)`, `remove(Person key)`, and `size()` methods int a separate tester class.

Finally, make a copy of your PhoneBook class called MyHashTable that will store <key, value> pairs in a ***generic*** fashion. In other words, rather than *only* storing <Person, PhoneNumber> pairs, this class will use Java generics to store <K, V> pairs. Make the necessary changes to finish your own generic hash table class!
Note: You will need to make a copy of the IMap interface and modify it to work with your new generic class.
Note: Java won't let you instantiate a generic array, google "Java generic array" for workarounds.

(Advanced) Tree Map

Make a new class called PhoneBookTree that uses a binary search tree (BST) as the backbone of the hash table, rather than a table / linked list.

This is how Java's TreeMap class works* – while the TreeMap implementation gives up some of the performance of Java's HashMap (a hash table), it does have the benefit that elements are maintained in sorted order (sometimes very useful).

**Java 8 actually uses a self-balancing "Red-Black" tree, but the concept is the same*

Rather than re-writing all that code, now might be a good time to convert your MyBST class to a generic version, i.e. MyBST is now MyBST<T>.

Note that you can't instantiate a generic array (e.g. `T[] t = new T[]`). Google "how to instantiate a generic array java" for an explanation.

(Advanced) BST for collision resolution

Additionally, you could also make a hash table that stores entries in a regular array but uses a BST rather than a linked list for collision resolution. For very long collision chains, this would give a significant performance boost (and is how Java handles open hashing in the HashMap class at the time of writing).