In this homework, you will implement a generic hash table. We will use the technique of chaining to resolve collisions.

1 Concerning Hash Tables

For this assignment, you will re-implement the Map ADT using a hash table. 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). It is the goal of this method to generate a unique integer (used as an array index) for every unique instantiation of its class. If two different objects are considered “equal” (o1.equals(o2)) they should hash to the same value. 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 to do! This problem coupled with the fact that there are often more possible objects than spaces in the array necessitates some sort of collision resolution.

For our “chaining” strategy of collision resolution, our table will consist of an array of “buckets”. These buckets can be implemented in a number of ways, but for this assignment each bucket will exist as a collection (instantiated as an ArrayList) of map entries. These entries will find their bucket based on the hash value of their key. When two different keys hash to the same bucket, we can resolve the collision easily by allowing them both into the collection. The fact that ArrayLists preserve order is an added bonus that means our buckets will preserve the order that elements were hashed to them (until a rehashing occurs).

You need to handle both “nice” hash functions which are evenly spread out and “nasty” ones in which hash values frequently collide. We provide tests for both situations.

Our class will be generic and will use the fact that all objects in Java implement a hashCode method. We provide a method to create arrays of generic type; as explained in class, these arrays are not actually of the generic type (we are lying to the compiler), and thus you must ensure that the array never escapes the scope of the generic type parameter.

2 Implementing the MyHashTable ADT

As you saw in previous assignments, Java provides a Map interface and a helper abstract class AbstractMap. You will extend the latter class again. As before, we provide a helper method to cast an Object passed as a key to type K.

Our MyHashTable<K,V> class will need to implement the following methods:

- int hash(Object) Return hash of the object, ensuring it can be used as an index in our table. This method should not crash if the table is null or has zero length.
- V get(Object) Get the value corresponding to a given key or null otherwise.
- V put(K,V) Put an entry into the table, returning the previous value (if any).
boolean containsKey(Object) Return whether the argument is a key in the table. (For efficiency)

V remove(Object) If the argument is a key in the table, remove the corresponding entry and return the value. Otherwise return null.

Set entrySet() Return the set of entries backed by the table.

The entry set should override the following methods.

int size() Return the number of entries.

void clear() Remove everything from the table and return it to its original capacity.

boolean contains(Object) Returns whether the parameter is an entry in the table. (Needed for efficiency.)

boolean remove(Object) If the table contains the parameter entry, remove the it and return true. Otherwise, return false. (Needed for efficiency.)

Iterator iterator() Return an iterator over the set.

3 Concerning Rehashing the Table

The table should initially have an array of size 10. When the number of elements in the set becomes more than MyHashTable.LOAD_FACTOR times the capacity of the array, a new array should be allocated (use the makeArray helper method) and all items should be “rehashed”. The new array should have size $2n + 1$, where $n$ is the current capacity.

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

4 Concerning the Iterator

The required design for the hash table iterator has three fields:

_bucketIt Nested iterator over the current bucket.

_nextBucket Index of the next bucket to iterate over.

_myVersion The iterator’s version.

For this iterator we take advantage of the fact that our buckets are Collections that can be responsible for iterating over their own contents. We will utilize a nested iterator (_bucketIt) to iterate over a bucket’s contents.

When the iterator is started, the nested iterator should be retrieved from the first non-null/non-empty bucket. The _nextBucket field should be set to the next non-null/non-empty bucket after the current one. If there are no more buckets to iterate over, the next bucket is set to the length of the array.

You may utilize the nested iterator’s remove() method to perform removes, but be sure to also update any relevant versions or counts!
5 What you need to do

You need to complete the implementation of `MyHashTable`, its entry set and its iterator, together with the two invariant checkers. As before, the iterator invariant should not be checked unless the versions match, and any version mismatch inside an iterator method should throw an exception. (What kind?) As usual, the invariant should be checked at the start and end of any method that changes the table, or just at the start if it merely accesses the table.

6 Files

The repository for this homework assignment includes the following files:

- `src/TestInternals.java` Tests for the invariant checkers.
- `src/TestHashTable.java` Tests for the `MyHashTable` class.
- `src/TestEfficiency.java` Efficiency tests for the `MyHashTable` class. (Should take under 20s)
- `src/TestStats.java` Gives statistics for various-sized instantiations of your hash table.
- `src/edu/uwm/cs351/PlateAssist.java` An example driver using your hash table to map license plates to registrations.
- `src/edu/uwm/cs351/AbstractEntry.java` Class for the entry objects in the table.
- `src/edu/uwm/cs351/MyHashTable.java` Skeleton file for the main class.