Assignment 4: Hashtables

In this assignment we'll be revisiting the rhyming dictionary from assignment 2. But this time we'll be loading it into a hashtable and using the hashtable ADT to implement a bad poetry generator.

Point breakdown

TO DO #1: Implement a hashtable - 60 points
TO DO #2: Loading the dictionary - 20 points
TO DO #3: Removing unrhymable words - 20 points

To Do 1: Implementing a Hashtable

You'll be implementing a hashtable class called `MyHashtable`. It implements the interface `DictionaryInterface`. Dictionary operations were discussed in class. There's a description of them on pages 643-651 of the book as well, though the book calls these `Tables` instead of `Dictionaries` (as discussed in class, we're using the term *dictionary* as well as the standard names for dictionary operations instead of the book's non-standard names).

The hashtable you'll be making will use `Strings` as the keys and `Object` as the values. Similar to linked lists, by storing `Object` as values, you can store any kind of object in the hashtable.

To implement a hashtable:

- You'll need to define a protected inner class inside `MyHashtable` called Entry (similar to how you defined an inner class for `Node` in Assignment 2). This inner class stores Key/Value pairs. So it has two fields:
    - `String key`
    - `Object value`

  It also should have a constructor for initializing the key and value.
- Your hashtable will define three protected fields (remember that protected means that the field can only be accessed from within the class or by *any child class of the class*).
    - `int tableSize` - the size of the array being used by the hashtable
    - `int size` - the number of key/value entries stored in the hashtable
    - `MyLinkedList[] table` - an array of MyLinkedList. The reason that each element of the array is a linked list is to store multiple entries which collide, that is, for which the hash for the different keys is the same index in the table.
- You'll be implementing the following methods on MyHashtable
    - `public boolean isEmpty()`
      Returns true if the hashtable is empty, false otherwise. You can use the `size` field to determine this easily.
    - `public int size()`
      Returns the size (number of key/value pairs stored in the hashtable).

- `public Object put(String key, Object value)`
  Adds a new key/value pair to the hashtable. If the key has been previously added, it replaces the value stored with this key with the new value, and returns the old value. Otherwise it returns null. There's more info on how to implement this method below.
- `public Object get(String key)`
  Returns the value stored with the key. If the key has not previously been stored in the hashtable, returns `null`. There's more info about how to implement this method below.
- `public void remove(String key)`
  Removes the key/value pair associated with the key from the hashtable. There's more info about how to implement this method below.
- `public void clear()`
  Empties the hashtable. The easiest way to do this is to just set `table` equal to a new fresh array - the old one will be garbage collected (memory reclaimed) by java. Remember to set `size` to 0 as well.
- `public String[] getKeys()`
  Returns an array of all the keys stored in the table. This function is necessary because having all the keys is the only way to iterate through the values in a hashtable. There's more info about how to implement this method below.
- `public MyHashtable(int tableSize)`
  The constructor for the hashtable. Takes an argument that is used to set the size of the array used to store the hashtable. Initialize `tableSize`, `table`, and `size`.

Hash Codes

In a hashtable, to compute an index into the array given a key you compute a hashcode for the key. Since our keys are all `Strings`, we'll be using the method `hashCode()` which is already provided on `Strings`. As an example:

```
String key = "hello";
int hashCode = key.hashCode();
```

The integer returned by `hashCode()` ranges over the full range of negative and positive integers. So the number could be way out of range for indexing our table (depending on our array size) or could be negative, which we definitely can't use for indexing our array. So we'll use the same trick we talked about with array-based Queues of using the modulo operator to get the number within range:

```
int arrayIndex = Math.abs(hashCode) % tableSize;
```

Math.abs() gets the absolute value (to get rid of negative numbers) and `% tableSize` puts the number into the range `0..tableSize-1` by returning the remainder after dividing by `tableSize`.

`get()`, `put()` and `remove()` all take a key as one of the arguments. So these functions will all need to compute an array index from the key to look in the table.

Remember that our table is an array of type `MyLinkedList`, where each item in the linked list is an `Entry` (storing a key and value). Why can't we just store the values directly in the table? The reason is that hash functions can result in *collisions*, where two different keys get mapped to the same array index (because they have the same hash code). So we have to story our entries (key/value pairs) in lists. In a hashtable, this list is called a bucket (or sometimes a slot). Each list in the table stores entries whose keys result in hash collisions. But if our hash function is good, it will spread the data out well so that no bucket ever gets too long.

Implementing `Object get(String key)`

To implement `Object get(String key)` you need to:

1. Compute an array index given the key (see above).
2. If that location in the table is `null`, that means nothing has been stored using a key with this hash code. So we can return null.
3. If the location isn't `null`, then it contains a `MyLinkedList` which is the bucket for all keys that collide using the hash function.
4. Linearly search through the bucket (the list), comparing the key for each entry with the key passed into `get()`. If you find a match, return the value. If you get to the end of the list without finding a match, return `null` (nothing stored for this key).

Implementing `Object put(String key, Object value)`

To implement `Object put(String key, Object value)` you need to:

1. Compute an array index given the key.
2. If that location in the table is `null`, that means nothing has been previously stored using a key with this hash code.
    a. Create a new `MyLinkedList` to be the bucket.
    b. Add the new `Entry` for the key/value pair to the list.
    c. Set this location in the array equal to the new bucket (list).
    d. Increment the `size` (the number of unique keys you have stored).
3. If the location in the table isn't `null`, that means keys with this colliding hash code have been previously stored. So our new key/value pair might be a key that's already been added (in which case we replace the value), or a brand new key (in which case we add a new `Entry` to the bucket).
    a. Linearly search through the bucket (the list) stored at this array location comparing the key for each entry with the key passed into `put()`. If you get a match, this means this key as been previously stored. Save the old value in the `Entry` (so you can return it) and replace it with the new value. You don't need to increment the size since you're not adding a new key.
    b. If you don't find the key in the bucket, then just add a new `Entry` (with the key and value) to the beginning of the list. Increment the `size`.

4. Return the old value if storing using an existing key (step 3.a above), otherwise return `null` if you're adding a new key (step 2 or step 3.b).

Implementing `void remove(String key)`

To implement `void remove(String key)` you need to:

1. Compute an array index given the key.
2. If that location in the table is null, then this key has definitely not been used to store a value. No need to do anything.
3. If the location in the table has a bucket, we need to linearly search it to see if it contains an `Entry` with the key. If you find an `Entry` in the bucket (linked list) with the key:
    a. Remove this `Entry` from the bucket.
    b. Decrement `size` (the number of unique keys stored in the hashtable).

Implementing `String[] getKeys()`

To implement `String[] getKeys()` you need to:

1. Create a `String[]` with a size equal to the number of unique keys in the hashtable (hint: one of our hashtable fields is keeping track of this).
2. Iterate through the hashtable array. For each table location that isn't `null`:
    a. Iterate through the bucket (linked list), getting the key out of each `Entry` and storing it in the array of strings you created in step 1. You'll need some kind of counter to keep track of where in the array of `Strings` you're adding the key.
3. Return the `String[]`

Extra Functions for Experimentation

Two extra functions that are not part of the `DictionaryInterface` have been provided on `MyHashtable` to let you experiment with how collisions change as you change the table size of `MyHashtable`. There's no To Do item associated with these functions; they're just for your own experimentation.

`public int biggestBucket()` returns the size of the largest bucket (the most collisions) in the hashtable.

`public float averageBucket()` returns the average bucket size.

Together, these two functions give you a sense of how frequently collisions are happening in the hashtable. As you make the table size smaller, the number of collisions will go up. In the limit of creating a hashtable with 1 table entry, then every key/value pair is stored in one big list.

On `MyHashtable` there's also an implementation of `public String toString()`. This allows you to print out the key/value pairs in your hashtable. There's also a method in `RhymingDict.java` called `public void testDictionary(DictionaryInterface dict)`. You can use this method to test your hashtable once you've implemented it. It does some simple adding, removing and replacing of key/value pairs and prints out the hashtable so you can confirm your table is working correctly.

Rhyming Dict

After you've made your hashtable, the remaining two To Do items are in `RhymingDict.java`. `RhymingDict.java` already does the following:

- Creates a `MyHashTable` with size 20,000.
    - The *keys* we'll use in this hashtable are rhyming groups (like `"AA1 V AH0"`).
    - The *values* we'll use in this hashtable are `MySortedLinkedList`. Each `MySortedLinkedList` will store individual words sharing a rhyme group.
    - We're providing you with a working version of `MySortedLinkedList`.
- Does the file management to read each line from the CMU Pronunciation dictionary
    - The CMU Pronunciation dictionary is a free data source of how each word in English is pronounced, useful for text-to-speech or rhyming applications.
- Writes poems
    - Picks two rhyming groups at random from an array of keys.
    - Gets the `MySortedLinkedList` of words for each group.
    - Picks two random indices for each list (based on the length of the list), and uses these two get four words, two words from each list.
    - Uses those words to make a poem, e.g.

```
"Roses are tapers,
violets are calmest.
I am vapors
and you are promised."
```

Note: *we removed most of the bad words from the dictionary, but the poems might still sometimes make bad or offensive juxtapositions*

You need to implement the following:

- **TO DO # 2**: Store each line from the CMU dictionary in the hashtable. This involves implementing the method `storeRhyme()`.
    - Use `getWord()` and `getRhymeGroup()` to get the word and rhyme group for the line.
    - Lookup (get) the key (the rhyme group) in the `Dictionary` (hashtable). If the result is null, then this rhyme group has not been added before.
        - Create a new `MySortedLinkedList`.
        - Add the word to the list.
        - Put the key (rhyme group) and value (list) in the `Dictionary`.
    - If the result of the lookup (get) isn't null, then we've already started a word list for this rhyme group.
        - Add the word to the list returned by `get()`. Nothing needs to be added to the `Dictionary` since the list is already in the `Dictionary`.

- **TO DO #3**: Remove the unrhymable words from the dictionary. Some words are in a rhyme group by themselves. That means that nothing rhymes with them. We want to get rid of those before trying to make poems. You'll do this by implementing `removeUnrhymables()`.
    - Use `getKeys()` to get an array of all the keys.
    - Iterate through all the keys, retrieving the value (linked list) associated with each key.
        - If the length of the list is 1, that means there's only one word in the list: nothing rhymes with it. Use `Dictionary.remove()` to remove this entry.
    - If you're curious to see what words don't have rhymes (at least according to the CMU pronunciation dictionary), you could add a println to print out the words as you remove their corresponding entries. If you do this, don't forget to comment it out before you turn it in.

Example Input and Output

`RhymingDict` can take 0, 1 or 2 command line arguments.

- The first argument is a seed for the random number generator. If you provide 0 arguments this defaults to the current system time.
- The second argument is the number of poems to generate. If 0 or 1 arguments are provided, this defaults to 3.

For this command line:

```
java RhymingDict 20 4
```

the output should look like:

```
If I were attuned
then you'd be the muggy,
And we'd both be marooned
and never be buggy

If I were tiber
then you'd be the jonas,
And we'd both be fiber
and never be bonus

Roses are flourish,
violets are deeply.
I am nourish
and you are steeply.

Roses are learners,
violets are overturn.
I am burners
and you are sunburn.
```

Turning the code in

- Create a directory with the following name: <student ID>_assignment4 where you replace <student ID> with your actual student ID. For example, if your student ID is 1234567, then the directory name is 1234567_assignment4
- Put a copy of your edited files in the directory (`RhymingDict.java`, `MyHashtable.java`). Note: your `Entry` helper class should be implemented as an inner class *inside* of `MyHashtable`.
- Compress the folder using zip. Zip is a compression utility available on mac, linux and windows that can compress a directory into a single file. This should result in a file named <student ID>_assignment4.zip (with <student ID> replaced with your real ID of course).
- Double-check that your code compiles and that your files can unzip properly. You are responsible for turning in working code.
- Upload the zip file through the page for Assignment 4 in canvas.