CPS 112: Computational Thinking with
Algorithms and Data Structures
Homework 7 (20 points)

Handed out: November 3 2016 (Thursday)
Due: 11:59pm November 10, 2016 (Thursday)
Late submissions accepted with penalty until 11:59pm November 12 (Saturday)

- **If you run into trouble or have questions, arrange to meet with me or a tutor!**

- **Before you submit:**
- **Please review the homework guidelines available on Canvas**

- Read the assignment very carefully to ensure that you have followed all instructions and satisfied all requirements.

- Create a zip file named like this: *yourusernameHWX.zip* (with your username and replacing the *X* with the assignment number) that contains a directory named *yourusernameHWX*, which contains all of your .java files, your Makefile, your debug log, and your cover sheet.

- Make sure to thoroughly test all of your programs before you submit your code.

- Ensure that your code conforms to the style expectations set out in the homework guidelines.

    - Make sure your variable names are descriptive and follow standard capitalization conventions.

    - Put comments wherever necessary. Comments at the head of each file should include a description of the code. Comments at the beginning of methods describe what the method does, what the parameters are, and what the return value is. Use comments elsewhere to help your reader follow the flow of your code.

    - *Program readability and elegance are as important as correctness.* After you've written your function, read and re-read it to eliminate any redundant/unused lines of code, and to make sure variable and function names are intuitive and relevant.

In this assignment you will use hash tables to create a program that can take a body of text, and generate random text in (roughly) the same style. For example, here is some randomly generated text in the style of A Midsummer Night's Dream by William Shakespeare:

"...the Palace of study. QUINCE That's all forgot? All with all the tinker! Starveling! God's my gentle day! [Lies down.] OBERON What do wish the chase; The more devils than despise. HERMIA I think it doth kill cankers in the glimmering light, By the night you in silence yet so That I entice you? FAIRY Ready. BOTTOM Masters, the dank and hair to the time I will keep his heart to make it else commit'st thy hours! Shine comforts from light, And she, with haste, For now the face; and me: Why are these sleepers be. THESEUS Here comes one...."
As you can see, it doesn't make a whole lot of sense, but it has the feeling of Shakespeare's writing. Here is some text randomly generated using a collection of fairy tales by The Brothers Grimm:

"...the queen in the finest flowers growing old woman has been so overcome with a hurry that he came to the carpenter, he had a great toe, and thought: 'One is not knowing what they fell before them, and down and round hundred cages. When he answered, 'With all ran away. Then he likewise wept and in the father and soon contrive to ashes.' Then Roland into the two daughters; one of fur put a little room and you in the worse for a joke that she ran to set out through every morning; but she came down to do..."

Again, it doesn't make much sense, but it certainly has a fairy tale style. You can find other fun examples of this idea on-line. For instance,

Automatic Computer Science Paper Generator: http://pdos.csail.mit.edu/scigen/
Postmodern Essay Generator: http://www.elsewhere.org/pomo/

1. **Making HashMap More Useful**

You'll be using HashMaps to create the text generator, but first you'll have to make a few improvements over the basic `HashMap` implementation given in class (and lab). In *HashMap.java* add a new class `ChainingHashMap` that is a subclass of the `HashMap` class. First and foremost, your new class should use chaining, rather than re-hashing, to resolve collisions:

• Override the `put` method so it uses chaining:
  - First hash the key.
  - If the slot corresponding to the hash value contains `null`, then put an `ArrayList` containing the key in the slot in the keys table and an `ArrayList` containing the data in slot in the data table.
  - Otherwise, the slot contains a list. Search the list to see if the key is already there.
    - If it is, change the data item in the same location of the list in the same slot in the data table to the new data item.
    - If it isn't, append the key to the list stored in this slot and similarly append the data to the list stored in this slot in the data table.
  - You no longer have to worry about re-sizing or keeping track of the number of keys, so don't. Chaining provides the advantage that your hash map can now store more items than it has slots!

- Override the `get` method so it takes the chaining into account.
  - First hash the key.
  - If the slot corresponding to the hash value contains `null`, return `null`.
  - Otherwise, the slot contains a list. You should search the list to see if the key is in the list.
    - If it is, return the data item in the same position of the list in the same slot in the data.
    - It if isn't, return `null`.
- Override the `containsKey` method in a similar manner to how you overrode the `get` method.
- Override the `getKeyValuePairs` method to take the chaining into account.
  - Go through all the lists in all the slots to add each key/value pair to the returned list.
- We could continue to use the built in `hashCode` method that all objects have, but because int data types can be negative, the result could be a negative value (test for yourself by printing out the hash generated by the String "family") In Java, the modulus operator does not gauratee a positive value and negative values are bad for indexing. So override hash as follows
  - Convert the key to a String
  - Sum up the ASCII values for each character in the String
  - Modulus the sum with the hash table size

Test your `ChainingHashMap` thoroughly before moving on. Make sure you can store and search for items that have the same hash value (for instance, 23 and 43 should hash to the same place. Add items to your map and print it out, to make sure it looks like what you would expect. You may need to add extra prints to your put method to print out the underlying data structures to make sure everything is okay.

2. **Training the TextGenerator**

You have been provided with a template for the `TextGenerator` class in *TextGenerator.java*, with some method implementations missing. You will have to fill in the implementations for the `train` and `generateText` methods. First, let's get an idea of how the `TextGenerator` will do its job.

You'll be creating what is called a Markov model to generate random text. The idea is to take each word and determine how often every other word follows that word. For instance the word "the" is far more likely to be followed by the word "dog" than "ran." On the other hand, "dog" is far more likely to be followed by "ran" than "dog." Once we have these frequencies, we can generate text by generating a word, and then generating a word to follow that word, and then a word to follow that word, and so on.

First, you will fill in the `train` method. This method takes a body of text (as a string) and uses it to increase the frequency counts of words following other words (stored in hash maps). Note the three instance variables in the constructor of `TextGenerator`:

- `totalWords` – the total number of words that the `TextGenerator` has been trained on (not the number of distinct words, just words total)

- `totalTable` – this hash map uses strings for keys (words) and stores numbers as data (counts). It stores the total number of times each word was encountered in the training text.
- `countTable` – this hash map uses strings for keys (words). It will store hash maps as data. Each of these inner hash maps will use strings for keys (words) and store numbers (counts). This hash map of hash maps is for storing the number of times each word follows each other word. So `countTable.get("the")` should return a ChainingHashMap and doing a get on that as in `myHash.get("dog")` should be the number of times "dog" followed "the" in the training text.

Your train method should:

- Split the training text into a list of words.
- For each word in the list (except the last one).
  - Increase `totalWords` by 1.
  - Increase the `totalTable` entry for that word by 1.
    - Note: if the word is not already in the table, you must add it, and initialize its count.
  - Increase the `countTable` entry for that word and the next word in the list by 1.
    - Note: if the word is not already in `countTable`, you must add it, setting its associated data entry to a new `ChainingHashMap` (with 1000 slots).
    - Note: if the word is in the `countTable`, but the next word is not in the inner hash map associated with the word, you must add the next word, and initialize its count.

To test your train method, add some print statements at the end to print out `totalWords`, `totalTable`, and `countTable` and try training on simple examples to make sure they look the way they should. For instance, if you train with the string 'a b a b a' you should end up with:

```
totalWords:4
totalTable:HashMap{"a":2, "b":2}
countTable:HashMap{"a":HashMap{"b":2}, "b":HashMap{"a":2}}
```

Note: Once you are convinced that your method works, don't forget to remove the print statements!!!

3. **Generating Text**
   Next, fill in the `generateText` method, which uses the frequency counts gathered in the train method to randomly generate text. The method takes a parameter: the number of words to generate.

   You will make use of another method that has been provided: `sampleWord`. This method takes a list of Pair objects (`countList`) and a number (`totalCount`). The first element of each pair should be a string, and the second element a count (such as might be returned by the `getKeyValuePairs` method of a hash map that uses strings for keys and counts for data). The number `totalCount` should be the total count of all the words in the list. The `sampleWord` method randomly selects a word based on the relative frequency of the words in the list and returns it.

Your `generateText` method should:

- Generate the first word using the frequencies in the `totalTable` (pass the key/value pairs of `totalTable` and `totalWords` to `sampleWord`) and add it to the text.
- In a loop, until you've generated the specified number of words:
  - Generate and add the next word using the frequencies in `countTable` associated with the last word (pass the key/value pairs of the hash map contained in `countTable`, indexed by the last word as well as the count in `totalTable` associated with the last word).
- Return the text with "..." added to the beginning and end.

For instance, if you trained your TextGenerator on 'a b a b a' then `generateText(4)` should return '...a b a b...' around half the time and '...b a b a...' the other half.

You have been provided a main function that takes as command line parameters a filename (containing the source text) and the number of words to generate. It creates a `TextGenerator`, opens the given file, trains the `TextGenerator` using the contents of the file, use the `TextGenerator` to generate the specified number of words, and prints the generated text.

4. Generate Some Random Text!

Project Gutenberg (www.gutenberg.org) is a repository of public domain ebooks and a great place to find source texts for your program. If you go to their "Book Search" section, you can find particularly popular authors and books. When downloading a book, there are several formats to choose from. You should select "Plain Text UTF-8" as this is just a plain text file, which your program can easily process.

In your write-up, include your favorite randomly generated 100-word passages (make sure to indicate the source text). You must choose at least 2 source texts by different authors and include at least one randomly generated passage from each. Note: Many Project Gutenberg files have a substantial preamble, which can sometimes show up in your random text too. Amusing though it may be to see the word "download" crop up in otherwise Shakespearian text, you may wish to delete this preamble first to avoid that kind of thing.