CPS 112: Computational Thinking with
Algorithms and Data Structures
Homework 8 (20 points)

Handed out: April 4, 2018 (Wednesday)
Due: 11:59 pm April 11, 201 (Wednesday)
Late submissions accepted with penalty until 11:59 pm Apr 13, 2018 (Friday)

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 debug log, and your cover sheet.

- Make sure to compile and thoroughly test all of your programs before you submit your code. **Any file that does not compile will receive a 0!**

- 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 contents of the file **(no identifying information please!)**. 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 logical 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. (10 pts) Chaining Hash Map

You'll be using hash maps to create the text generator. In *ChainingHashMap.java* implement `ChainingHashMap`, which is a subclass of the `AbstractHashMap` class. This hash map should use chaining, rather than re-hashing, to resolve collisions. In particular, rather than maintaining an array of `Entries`, it should have an array of `ArrayLists` of entries. Each position of the array should either contain `null` (if no keys have hashed to this position) or contain an `ArrayList` of `Entries` whose keys map to this position.

Important note: Java will not allow you to have a variable `ArrayList<Entry>[] entries`; it's just a limitation of the language. You can, however, have a variable of type `ArrayList[]` entries. Note that in this case `entries[i].get(j)` is of type `Object`. Doing this will cause the "unchecked or unsafe operations" warning, but that's okay; you know what you're doing. You know that the lists are filled with `Entry` objects. If you want to treat the result as an `Entry`, you need to typecast like this: `Entry e = (Entry)entries[i].get(j)`. If you forget to do this you will get incompatible type errors.

Implement the following methods.

`Constructor` – Calls the superclass' constructor, setting the maximum load to 5 and initializes the array to the given capacity.

`put(key, value)` – If `key` is already in the map, replaces its associated value with the given `value`. Otherwise it adds an entry with `key` and `value` to the table.

`printEntries()` – (Will not be graded, but would sure be useful for testing and debugging!)

Stop and test. You should now be able to thoroughly test `put`. Make sure it works before moving on! You can write empty stubs for the other methods so it will compile.

`resize()` – Create a larger array (twice as big!) and rehash everything. (Also make `put` call `resize` if the load reaches the maximum).

Stop and test. Make sure resizing works the way it should!

`get(key)` – If `key` is in the map, it should return the associated value. Otherwise, return `null`.

Stop and test. Test your `ChainingHashMap` thoroughly before moving on. Make sure you can store and search for items that have the same hash value. Add items to your map and print it out, to make sure it looks like what you would expect.

Now, in *ChainingHashMapIterator.java* implement the `ChainingHashMapIterator` class, which should be a subclass of `AbstractHashMapIterator`. In many ways it will be similar to your `ProbingHashMapIterator`, but you must now take the chains into account as well.

Implement the `iterator` method in `ChainingHashMap` and **test your iterator thoroughly.**

2. (5 pts) 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 count 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 values (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 values. 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
  `ChainingHashMap m = (ChainingHashMap)countTable.get("the")`
  is a hash map containing the counts of the words following "the" and
  `Integer c = (Integer)m.get("dog")`
  should be the number of times "dog" followed "the" in the training text.

Note: If you have not been able to get your `ChainingHashMap` working the way you'd like, you may alter `TextGenerator` to work with standard Java `HashMap`s instead.

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`.
    - 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, temporarily 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 get something like:

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

(Obviously your formatting may vary). Note: Once you are convinced that your method works, don't forget to remove the print statements!!!

3. (5 pts) 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 `ChainingHashMap` (`counts`) and a number (`totalCount`). The map is assumed to map words to counts. The number `totalCount` should be the total count of all the words in the map. The `sampleWord` method randomly selects a word based on the relative frequency of the words in `counts` and returns it. This method makes use of your iterator, so it had better work right!

Your `generateText` method should:

- Generate the first word using the frequencies in the `totalTable` (pass `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 map in `countTable` associated with 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 a filename as a command line parameter. The file should contain the source text. The program creates a `TextGenerator`, opens the given file, trains the `TextGenerator` using the contents of the file, uses the `TextGenerator` to generate 100 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.

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.