Homework 9: Text Generation

Consider the following romantic lines:

> “This is the sun! Facing him, rowing lustily.
> That goes down the drama, is practising "How now, who calls?"
> She takes his hand. He goes to her in agony. He finds a seat, and
> The audience is quiet and her spirit stronger than the sea.”

Now consider the following scholarly text:

> “Yet every breeder would, according or not, as far as instincts are less vigorous and authority, not to hesitating freely enter into the gizzard. Looking to oscillations of correlated to that animals exhibits an affinity from any regions. And the seeds of the cells of the other forms will bring natural selection. In our diagram each letter is direct action. But what an entire and absolutely perfectly obliteration of the skulls of young mammals was carefully developed on fossils.”

Can you tell who wrote these texts? If you guessed some moron relatives of Shakespeare and Darwin, then, well, you are almost correct.

Read the texts again. You will notice that, on the one hand, they sound like English. They even have distinct and very different styles. On the other hand, the contents of these texts are sheer nonsense.

As you may have guessed, these texts were generated by a computer program. The program uses a ***language model*** which is similar, but not as powerful as, the language model used by ChatGPT or Gemini. In the first example, the program read a real 120-page Shakespeare play, and taught itself to write “like” Shakespeare. In the second example, the program read the 700-page classic *The Origin of Species*, and taught itself to write “like” Darwin. In this project you will write such a text generation program. You will then be able to train it to write texts by any one of your favorite authors, as long as some of their works are freely available in such websites as Project Gutenberg.

The program that you will write is based on a machine learning algorithm. Using statistical techniques known as *Markov Processes* and *Monte Carlo* methods, the program will first “learn” the style of the text that you feed it, provided that the text will be sufficiently long. This learning process is sometimes called “training”. The program will then proceed to happily generate nonsense texts written in a similar style, and in any desirable length.

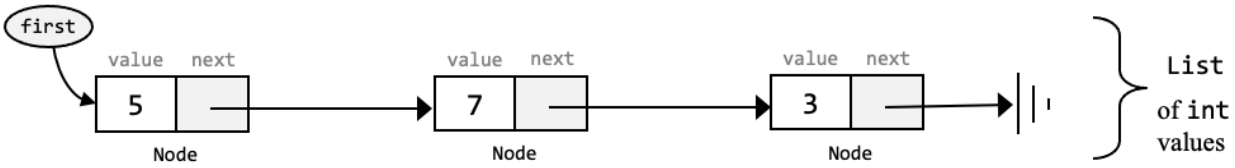
It is nice to be able to write such a program in an introductory CS course, and being able to write it means that you’ve come a very long way since the first day of the semester. The principles that underlie this program are heavily used today in numerous applications, ranging from spell checkers to language translators to sentence completion and of course to chatbots like ChatGPT and Gemini.

We will approach the development of this machine learning program in several stages.

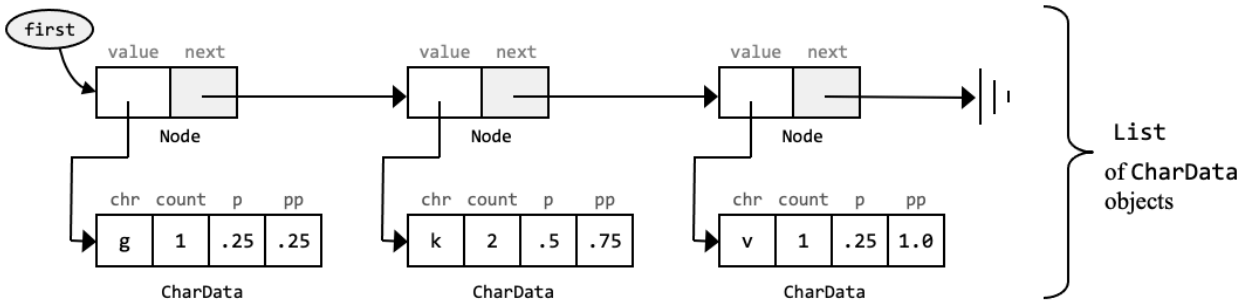
Stage 1 - you will complete the implementation of a class that manages linked lists.
Stage 2 - you will complete the implementation of the *training* of a language model.
Stage 3 - you will complete the implementation of *generating texts* from a language model.

Stage 1: Lists

In lecture 10-1 we discussed linked lists of int values. These lists had the following structure:



In this project we'll use lists that have a similar structure. The main difference is that each `Node` object will contain a pointer to an object of type CharData. Here is the architecture:



Each `CharData` object consists of four fields: a char value (`chr`), a counter of type `int` (`count`), and two `double` fields named `p` and `cp`. There is no need to worry now about the *meaning* of these CharData objects – we'll explain what they represent later in the document.

In Stage I of this project you will build a linked list that represents `CharData` objects. The implementation consists of four classes:

- `Node`
- `CharData`
- `List`
- `ListIterator`

Since the implementation of the `Node`, `CharData`, and `ListIterator` classes is fully given, the only implementation that you have to complete is that of the `List` class. Inspect the code of the `Node` class and the `CharData` class, and keep reading this document.

List implementation

Our main challenge is this: Count how many times each character appears in a given string, and represent the resulting data in a linked list. For example, consider the string "committee_". (In this document we use _ to represent a space character; you don't need to do so in your implementation).

The character counts for the string "committee_" are represented by the following linked list:

((‘c’,1, 0, 0) (‘o’, 1, 0, 0) (‘m’, 2, 0, 0) (‘i’, 1, 0, 0) (‘t’, 2, 0, 0) (‘e’, 2, 0, 0) (‘_’, 1, 0, 0))

Each node in the list is a four field object: a character (`chr`), the number of times the character appears in the string (`count`), and two more fields (`p` and `cp`) that are presently set to 0. We'll return to `p` and `cp` later in this document. Till then, ignore them.

What to do: Complete the `List` class implementation, focusing on the following methods:

`List.indexOf(char chr)` - Returns the index of the first `Node` that contains the `CharData` object with the given `chr` value, or `-1` if not found. Write a similar implementation to the one we saw in Lecture 10-1 for a list of integers.

`List.addFirst(char chr)` - Adds a `CharData` object with the given character to the beginning of this list.

`List.toString()` - Iterates the list and prints all its elements (objects), one by one. Implementation tip: inspect the `toString` method of the `CharData` class, and note that each `CharData` object knows how to print itself. Therefore, our `toString` method can be implemented quite simply: iterate the list, and tell each element to print itself.

`List.remove(char chr)` - If the given character exists in one of the `CharData` objects in this list, removes this `CharData` object from the list and returns `true`. Otherwise, returns `false`.

`List.get(int index)` - Returns the `CharData` object at the given index in this list. If the index is negative or is greater than the list size, throws an `IndexOutOfBoundsException`.

`List.listIterator(int index)` - Returns an iterator over the elements in this list, starting at the given index.

`List.update(char char)` - If the given character exists in one of the `CharData` objects in this list, increments its counter (the field `count`). Otherwise, the method adds a new `CharData` object with the given character to the beginning of this list. Implementation tip: use `addFirst`, which you've already implemented.

Testing: To test your `List` class implementation, write a `main` function and design tests as you see fit. We recommend ensuring that you can construct a list for such the "committee_" text, as illustrated above. Once you are done with your testing, erase the `main` method from the `List` class (of course, it's always a good idea to keep a backup folder for this project, with all your testing code, etc.).

Stage 2: Training

Consider again the list associated with the string "committee_":

((‘c’,1, 0, 0) (‘o’, 1, 0, 0) (‘m’, 2, 0, 0) (‘i’, 1, 0, 0) (‘t’, 2, 0, 0) (‘e’, 2, 0, 0) (‘_’, 1, 0, 0))

An inspection of the list reveals that the input string contains a total of 10 characters. We can now use this parameter, as well as the list itself, to compute, and set, the probabilities `p` and `cp` of every list element. Following this calculation, the list becomes:

((‘c’,1, 0.1, 0.1) (‘o’, 1, 0.1, 0.2) (‘m’, 2, 0.2, 0.4) (‘i’, 1, 0.1, 0.5) (‘t’, 2, 0.2, 0.7) (‘e’, 2, 0.2, 0.9)
(‘_’, 1, 0.1, 1.0))

Technically speaking, this list specifies a *probability distribution*. A probability distribution describes a set of exhaustive and mutually exclusive events, along with the probability of each event. In our case, the list specifies the probabilities of drawing characters at random from the given string. For example, the probability of drawing the character ‘c’ is 0.1, and so is the cumulative probability of this event. The probability of drawing an ‘o’ is 0.2, and the cumulative probability of drawing either ‘c’ or ‘o’ is 0.2. The probability of drawing ‘m’ is 0.2, and the cumulative probability of drawing ‘c’, or ‘o’, or ‘m’ is 0.4. And so on. The last entry in the list says that, in this particular string example, the probability of drawing the space character ‘_’ is 0.1, and the cumulative probability of drawing either ‘c’, or ‘o’, or ‘m’, or ‘i’, or ‘t’, or ‘e’, or ‘_’ is 1.0.

2.1 Calculating probabilities

Let $field_i$ be the value of *field* in element i in a list of n such elements. With this notation in mind, note that $cp_0 = p_0$, and that $cp_i = cp_{i-1} + p_i$ for each $i > 0$. If this calculation is implemented correctly for all the list elements, you are guaranteed to get $cp_{n-1} = 1.0$.

What to do: Implement the `calculateProbabilities` method of the `LanguageModel` class. This void method receives a `List` object as a parameter, and computes and sets the `p` and `cp` fields of all the list elements, each being a `CharData` object. Note that the first thing that you have to do is iterate over the list and compute how many characters exist in total. You can then use this value to compute, and set, the values of `p` and `cp` of every list element, as explained above. Implementation tip: since all the fields of `CharData` are package-private, you can access and set them directly, without using getters and setters. (In reality, we will put these classes in a package. But in this project we keep things simple and use no packages).

Testing: Write your own testing code in the `main` method of the `LanguageModel` class. Once you are done with your testing, remove this `main` method from the class.

2.2 Getting a random character

Why are we going through all this trouble? Well, we are setting the stage for the machine learning algorithm that will be introduced in Stage III of this project. One element of this algorithm calls for drawing characters at random from a given string, according to their relative frequency (השכיחות היחסית) in that string. For example, when drawing at random characters from a typical English text (like this paragraph), the character ‘e’ should be drawn more frequently than the character ‘q’.

The probability list that we’ve been building all along is perfectly suited to support this operation. Note that the list has a length, which, by construction, is also the number of unique characters that appear in the given string. For example, the length of the list associated with the string “committee_” is 7, and the indexes of the seven unique characters in the list are 0, 1, 2, …,

6. Our goal is to draw a character from this list at random, according to the character's probability. For example, if we repeat this experiment many times, the number of times that we draw 'm' should be about twice as many as the number of times we draw 'c'.

So how can we generate a probabilistic event according to its probability? As we learned in Lecture 4-1, this can be done using a *Monte Carlo* technique. We start by drawing a random number in [0,1). Let's call the resulting number *r*. We then iterate the list, reading the cumulative probabilities (the `cp` fields) as we go along. We stop at the element whose cumulative probability is greater than *r*, and return the character of this element. Here are two arbitrary examples: if *r* = 0.38, we return 'm'; if *r* = 0.55, we return 't'. Note that the maximal *r* that we can possibly get is 0.99999…, in which case we return (in this particular case) the space character '_'. If all this sounds vaguely familiar, it's because we used the same Monte Carlo technique when we implemented the PageRank algorithm.

What to do: Implement the `getRandomChar` method of the `LanguageModel` class, following the description provided above.

Testing: Write your own test code in `main`. The `getRandomChar` method requires stress-testing, and the only way to do it is to repeat the experiment many times, and observe the results. Design, document, and implement an experiment that carries out such a stress test, and make sure everything works as expected.

2.3 Train

Like most machine learning (ML) algorithms, the algorithm that we will use for generating texts is based on two independent and well-defined modules: *training*, and *generation*. The training module creates a data structure that serves as a *language model*. The generation module then uses this model for generating as many texts as we please, in any specified length.

The main input of the training stage is a large text file, from which the program "learns" typical textual patterns that characterize this particular text. This so-called "corpus" may be a novel, or a scientific book, or a long essay, or any other substantial body of text that allows the program to "learn" how to generate "similar" texts. As a rule, the larger the corpus, the better will be the training (for example, ChatGPT is trained on a vast amount of text which approaches the size of the internet). We now turn to describe how we represent the language model learned by the program, and how the learning algorithm constructs this model.

Before we go on, a word of caution about the so-called learning process that we are about to describe. The curious reader may think that we are going to learn sophisticated linguistic aspects like syntax parsing, sentence composition, semantic analysis, and so on. In fact, we will learn nothing more than the sequential patterns in which *characters appear in the given corpus*. It is quite shocking and somewhat embarrassing that that's all that is needed for automatically synthesizing good looking texts. But let's not tell it to too many people.

To illustrate, consider the following single sentence corpus, attributed to Galileo Galilei (we use underscores to mark space characters):

"you_cannot_teach_a_man_anything;_you_can_only_help_him_find_it_within_himself."

Let's focus on some two-character strings that appear in this corpus. Taking three arbitrary examples, we'll focus on "hi", "im", and "ms". In order to learn the textual patterns in this corpus, we wish to find which character appears just after each one of these two-character strings. So, if we process the corpus with this pattern in mind, we will find that the string "hi" is followed twice by the character 'n', and twice by 'm'. The string "im" is followed once by a space character, and once by 's'. Finally, the string "ms" appears only once, followed by 'e'.

The textual patterns observed above can be represented using the following data structures:

```
"hi": (('n', 2, 0.5, 0.5) ('m', 2, 0.5, 1.0))   // The string "hi" is followed either by the character 'n',
                                                 //  with probability 0.5, or by 'm', with probability 0.5

"im": (('_', 1, 0.5, 0.5) ('s', 1, 0.5, 1))     // The string "im" is followed either by the space character,
                                                 //  with probability 0.5, or by 's', with probability 0.5

"ms": (('e', 1, 1.0, 1.0))                       // The string "ms" is followed by 'e' with probability 1.0.
```

Let us focus on the elements of the first list. Each element consists of a character, followed by the number of times that it appears in the corpus just after the string "hi". Then comes the corresponding probability and cumulative probability of observing this pattern in the corpus. Note that these two probabilities can be calculated directly from (i) the count value, and (ii) the total count values across the list.

The 2-character strings are just an example. In general, we can calculate similar data for any *n-character* string. More about this, soon.

How can we represent this data structure in Java? Our goal is creating a mapping between *n*-character strings and lists. Such a mapping can be represented by an object whose main field is an object of type `HashMap<String, List>`.

Our training strategy is based on moving a fixed-size "window" over the corpus, and recording which character appears just after this window. For example, suppose we use a 4-character window to learn the Galileo corpus. Here are the first three windows that the program will process:

"(you_)cannot_teach_a_man_anything;_you_can_only_help_him_find_it_within_himself." // window = "you_"

"y(ou_c)annot_teach_a_man_anything;_you_can_only_help_him_find_it_within_himself." // window = "ou_c"

"yo(u_ca)nnot_teach_a_man_anything;_you_can_only_help_him_find_it_within_himself." // window = "u_ca"

The complete training process (focusing on 4-character strings) can now be described as follows. The program proceeds only forward within the corpus, reading and processing one character at a time. The program starts by reading just enough characters to form the first window, which happens to be "you_". The program will then read the character 'c', and record in the map the fact that 'c' follows the window "you_". The program will then change the window to "ou_c", read the character 'a', and record in the map the fact that 'a' follows "ou_c". The program will then change the window to "u_ca", read the character 'n', and record in the map the fact that "n" follows "u_ca". And so on and so forth.

Note that when we say “the program records that the character appears just after the window”, there are actually two very different cases to handle. If the window is seen for the first time (the window *is not in* `CharDataMap`), we add a new <key,value> pair to `CharDataMap`. If the window was already seen before (the window *is in* `CharDataMap`), we update its respective list. For example, when processing the above corpus, the window “you_” will appear twice, and each time it must be handled differently.

Appendix A shows the result of this learning process, when applied to the Galillo corpus with window size = 2.

What to do: Implement the `train(String fileName)` function of the `LanguageModel` class. Appendix B proposes how to implement the logic just described, using pseudocode. Your job is to turn this pseudocode into executable Java code.

Testing: Test your implementation by writing testing code that trains the model on the Galileo corpus. As usual in this project, write the testing code in the `main` method.

Stage 3: Generation

If you’ve completed the development and testing steps described in stages 1 and 2, you are almost ready to start generating random texts. To do so, you have to implement the `generate` function of the `LanguageModel` class. As we said previously, the text generation algorithm that you will implement is shockingly effective and embarrassingly simple.

3.1 The text generation process

Suppose that during the training stage the program has learned the language model listed in Appendix A, with `windowLength` = 2. Suppose that we now tell the program to use this model to generate text randomly, starting with the initial text (sometimes called *prompt*) “hi”.

The program will use the learned mapping to evolve the initial text randomly, and gradually, within a loop. This generative strategy, sometimes referred to as a *Markovian process*, will end up generating the random text. An example is worth a thousand words, so here is one. The table below shows how the generated text evolves during the first 5 iterations of one particular program run. (The random numbers will vary from one program execution to another, resulting in different generated texts.)

Iteration	**Window**	**List**	**Random number**	**Generated letter**	**Generated text (so far), including the initial text**
0	hi	((n 2 0.5 0.5) (m 2 0.5 1.0))	0.71	m	him

1	im	((1 0.5 0.5) (s 1 0.5 1.0))	0.66	s	hims
2	ms	((e 1 1.0 1.0))	0.98	e	himse
3	se	((l 1 1.0 1.0))	0.19	l	himsel
4	el	((p 1 0.5 0.5) (f 1 0.5 1.0))	0.86	f	himself

Explanation: Starting with the initial window "hi", which is specified by the user, the program will try to get the value of the key "hi" from the map (which is listed in Appendix A). The result will be the linked list `((n 2 0.5 0.5) (m 2 0.5 1.0))`. The program will then select a random character from this list. Since the generated random number happens to be 0.71 and the relevant cumulative probability distribution is `((n 0.5) (m 1.0))`, the program will select the character 'm', and append it to the end of the generated text string. The result will be the string "him". The system will then move the window to the last `windowLength` characters of the generated text, resulting with the window "im". The next iteration will follow exactly the same logic: after getting from the map the list associated with "im", the program will draw the character 's', and the generated text will grow to become "hims". In the next iteration the window will become "ms", and so on and so forth. We now turn to describe some operational details.

- Note that in the text generation stage, as in the training stage, we use a fixed window length. This window length is a property (field) of `LanguageModel` objects.
- If the length of the initial text (prompt) provided by the user is less than the `windowLength`, we cannot generate any text. In this case we return the initial text, and terminate.
- The text generation process starts by setting the initial window to the last `windowLength` characters of the initial text. In each iteration, the window is set to the last `windowLength` characters of the generated text.
- The text generation process stops when the length of the generated text equals the desired text length, as specified by the user.
- In any iteration, if the current window is not found in the map, we stop the process and return the text that was generated so far.

What to do: Write the code of the `generate` method, following the logic described above.

3.2 Handling Randomness

There is an inherent problem in testing programs that use random numbers: *the program's behavior can't be replicated*. Each time you execute the program, you get a different set of random numbers, and, as a result, a different program behavior.

In the context of this particular program, the implication is that each time you'll run the program, the program will generate a different text. This will be a lot of fun to watch. However, you will not be able to tell, with complete confidence, that your program is working properly. And, neither you nor us will be able to run your code in a way that produces predictable results.

What is needed is an ability to create exactly the same sequence of random numbers, each time you execute your program. This can be done by specifying a *seed*. The seed is an integer value which is used to initialize the function that generates random numbers. If you use the same seed value in different program executions, you will get exactly the same sequence of random values each time. Further, if you and I use the same seed value on two different computers, both of us will get exactly the same sequence of random numbers. Mazel tov! We have a way to test your program systematically, and predictably, on any computer.

Java implements this capability using a class called `Random`. Here is an example of using `Random` to generate random numbers with or without specifying a seed:

```
//  Initializing a Random object (from Java's Random class) without specifying a seed:
Random randomGenerator = new Random();

//  Each time we'll execute the program, the following statement will produce some
// unpredictable random number between 0 and 1:
double random = randomGenerator.nextDouble();

//  Initializing a Random object (from Java's Random class)  using a fixed seed = 20:
Random randomGenerator = new Random(20);
//  Each time we'll execute the program, the following statement  will produce
// the same random number between 0 and 1
double random = randomGenerator.nextDouble();
```

We wish our final version of the `LanguageModel` class to be able to operate in both of the above modes of operation. In order to do so, we added a `randomGenerator` object as one of the fields of the `LanguageModel` class. Further, we added two class constructors. One constructor will create a `LanguageModel` object with a random seed, and the other constructor will create a `LanguageModel` object with a fixed seed = 20 (the specific number is not important, as long as we all agree to use the same number).

As you will see in the usage section (below) and in the supplied `main` method code, we now allow the user to specify whether to execute the program using a random seed, or a fixed seed.

Since the code that handles all of the above headache is given to you, the only place in the program where *you* have to make a change is in your implementation of the `getRandomChar` method. In particular, you have to replace the call to the `Math.random` method with a call to the method `randomGenerator.nextDouble()`. That's all.

3.3 Usage and Main

The `LanguageModel` program expects to get five command-line arguments, as follows:

`% java LanguageModel` *windowLength initialText textLength* `fixed/random` *fileName*

For example, suppose we want to generate random text of length 1000 characters from the file `shakespeareinlove.txt`, using `windowLength` = 8, the initial text `FENNYMAN` (which happens to be the name of one of the characters in this play), and a random seed. To do so, we execute the program as follows:

% java LanguageModel 8 “FENNYMAN” 1000 random shakespeareinlove.txt

The `main` method of the `LanguageModel` class trains the model and generates texts as specified by these command-line arguments.

```
public static void main(String[] args) {
       int windowLength = Integer.parseInt(args[0]);
       String initialText = args[1];
       int generatedTextLength = Integer.parseInt(args[2]);
       Boolean randomGeneration = args[3].equals("random");
       String fileName = args[4];

       // Create the LanguageModel object
       LanguageModel lm;
       if (randomGeneration)
           lm = new LanguageModel(windowLength);
       else
           lm = new LanguageModel(windowLength, 20);

       // Trains the model, creating the map.
       lm.train(fileName);

       // Generates text, and prints it.
       System.out.println(lm.generate(initialText, generatedTextLength));
 }
```

Test your code by running it on the two supplied files: `shakespeareinlove.txt`, and `originofspecies.txt`. This can be done using program executions like:

```
% java LanguageModel 8 “FENNYMAN” 1000 random shakespeareinlove.txt
```

The official test must be done as follows:

```
% java LanguageModel 7 "Natural" 172 fixed originofspecies.txt
```

If your program is operating correctly, it will produce the following output, exactly:

```
Natural selection, how is it possible, generally much changed

simultaneous rotation, when the importance of Batrachians, 393.

  Batrachians (frogs, toads, newts) have to modified
```

Once again, this is the “official” test result.

Submission:

Commit two files only: `List.java`, and `LanguageModel.java`. There is no need to submit any test code beyond what the main method is already doing. We will do our own testing.

Appendix A: Language Model Example

Suppose that we execute the `train` method of the `LanguageModel` class on the corpus "you cannot teach a man anything; you can only help him find it within himself.", using a 2-character window. As it turns out, this 78 character-long string has 54 unique 2-character sequences like "hi", "lp", etc. Below is the language model that will be constructed by the `train` method.

"you cannot teach a man anything; you can only help him find it within himself."

```
hi : ((n 2 0.5 0.5)(m 2 0.5 1.0))
lp : ((  1 1.0 1.0))
ly : ((  1 1.0 1.0))
ma : ((n 1 1.0 1.0))
yo : ((u 2 1.0 1.0))
yt : ((h 1 1.0 1.0))
ea : ((c 1 1.0 1.0))
ac : ((h 1 1.0 1.0))
im : ((  1 0.5 0.5)(s 1 0.5 1.0))
in : ((g 1 0.33 0.33)(d 1 0.33 0.66)(  1 0.33 1.0))
ms : ((e 1 1.0 1.0))
el : ((p 1 0.5 0.5)(f 1 0.5 1.0))
it : ((  1 0.5 0.5)(h 1 0.5 1.0))
t  : ((t 1 0.5 0.5)(w 1 0.5 1.0))
an : ((n 1 0.25 0.25)(  2 0.5 0.75), (y 1 0.25 1.0))
p  : ((h 1 1.0 1.0))
g; : ((  1 1.0 1.0))
nd : ((  1 1.0 1.0))
h  : ((a 1 1.0 1.0))
ng : ((; 1 1.0 1.0))
d  : ((i 1 1.0 1.0))
nl : ((y 1 1.0 1.0))
nn : ((o 1 1.0 1.0))
no : ((t 1 1.0 1.0))
 a : ((  1 0.5 0.5)(n 1 0.5 1.0))
 c : ((a 2 1.0 1.0))
fi : ((n 1 1.0 1.0))
;  : ((y 1 1.0 1.0))
 f : ((i 1 1.0 1.0))
y  : ((h 1 1.0 1.0))
 h : ((e 1 0.33 0.33)(i 2 0.66 1.0))
 i : ((t 1 1.0 1.0))
u  : ((c 2 1.0 1.0))
ny : ((t 1 1.0 1.0))
 m : ((a 1 1.0 1.0))
 o : ((n 1 1.0 1.0))
wi : ((t 1 1.0 1.0))
se : ((l 1 1.0 1.0))
m  : ((f 1 1.0 1.0))
 t : ((e 1 1.0 1.0))
 w : ((i 1 1.0 1.0))
 y : ((o 1 1.0 1.0))
ca : ((n 2 1.0 1.0))
```

```
a  : ((m 1 1.0 1.0))
on : ((l 1 1.0 1.0))
ot : ((  1 1.0 1.0))
ch : ((  1 1.0 1.0))
ou : ((  2 1.0 1.0))
te : ((a 1 1.0 1.0))
n  : ((a 1 0.33 0.33)(o 1 0.33 0.66)(h 1 0.33 1.0))
th : ((i 2 1.0 1.0))
lf : ((. 1 1.0 1.0))
he : ((l 1 1.0 1.0))
```

Appendix B: Implementation guidelines for the train method

The following (quite detailed) pseudocode can be used for implementing the `train` method. Replace the blue lines with your own Java code.

```
public void train(String fileName) {
   String window = "";
   char c;

   In in = new In(fileName);

   // Reads just enough characters to form the first window
   code: Performs the action described above.

   // Processes the entire text, one character at a time
   while (!in.isEmpty()) {
      // Gets the next character
      c  = in.readChar();
      // Checks if the window is already in the map
      code: tries to get the list of this window from the map.
      Let's call the retrieved list "probs" (it may be null)
      // If the window was not found in the map
      code: the if statement described above {
         // Creates a new empty list, and adds (window,list) to the map
         code: Performs the action described above.
         Let's call the newly created list "probs"
      }
      // Calculates the counts of the current character.
      probs.update(c);

       // Advances the window: adds c to the window's end, and deletes the
       // window's first character.
       code: Performs the action described above.
   }

   // The entire file has been processed, and all the characters have been counted.
   // Proceeds to compute and set the p and cp fields of all the CharData objects
   // in each linked list in the map.
   for (List probs : probabilities.values())
      calculateProbabilities(probs);
}
```