Homework 10: Text Generation / Stage I

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. 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 (for training) 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 “learns” the style of the text that you feed it (assuming that the text is sufficiently long). The program then proceeds to happily generate nonsense texts written in the same style, and in any desirable length.

It turns out that this program is much more than a cute gimmick. The principles that underlie this program are heavily used in numerous applications, ranging from spell checkers to language translators to the technique that Google uses to complete your search phrases and email messages. Some researchers even use these algorithms to train computer programs to write computer programs.

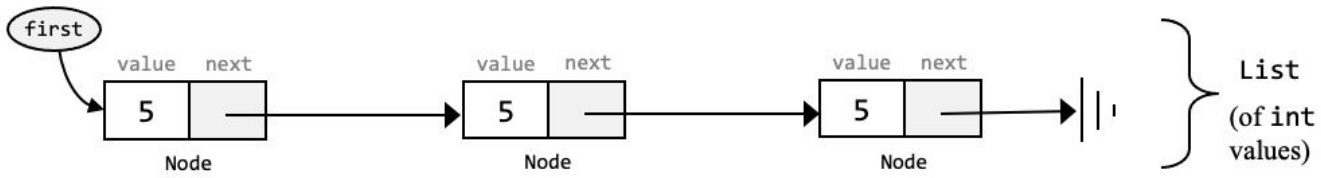
We hope that you’ll enjoy building this program, and playing with it.

The CharData class

In order to generate "Shakespeare-like" plays and "Darwin-like" texts we'll have to read and analyze plays and texts written by Shakespeare and Darwin. When reading this *training data*, we will keep track of every character that we read. In particular, each unique character will be characterized by four fields: the character, a counter that counts how many times it appears, and two probability fields names `p` and `pp`. The roles of the counter and probability fields in text generation will be explained in Part II of the project, so stay tuned. At this stage you are requested to review the code of the `CharData` class, which is given.

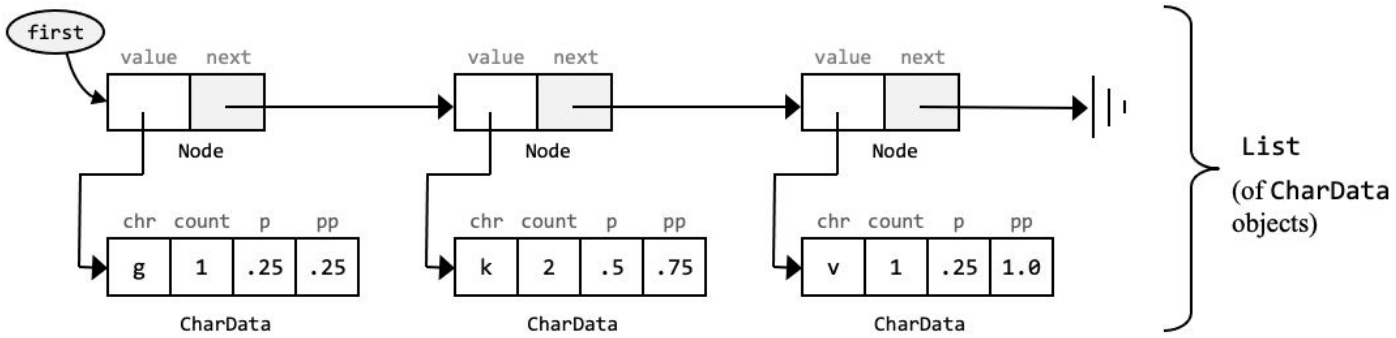
The List class

In lecture 9-2 we discussed linked lists of `int` values. These lists had the following structure:



The list above represents a sequence of `int` values (abstraction), and is built as a sequence of linked `Node` objects (implementation). Each node has a field of type `int` (`value`) and a field of type `Node` (`next`) that contains the address of the next `Node` in the list. Importantly, users of the list are not aware of the existence of the `Node` objects; they can add and remove `int` values using the `List` class API (slide 22 in lecture 9-2), in which the word "Node" is never mentioned.

In this project (Homework 10) we will build and use a customized version of this list that has a very similar structure. The basic difference is that each `Node` object contains not an `int` value, but rather a pointer to a `CharData` object. Here is the architecture:



At this stage you are requested to review the code of the `Node` class (which is given) and the code of the `List` class (which you have to complete). As we mentioned before, the `List` class API does not expose the `Node` objects. As far as the abstraction is concerned (namely, the `List` view seen by client code that builds and uses lists), a list is a sequence of `CharData` objects.

The Test1 class

This class performs a series of tests that test the various methods of the `List` class. Here is the output that this class generates:

```
% java Test1
Tests the addFirst method:
The list before adding any element: ()
The list after adding 'a' to the beginning: ((a 1 0.0 0.0))
The list after adding 'b' to the beginning: ((b 1 0.0 0.0) (a 1 0.0 0.0))
The list after adding 'c' to the beginning: ((c 1 0.0 0.0) (b 1 0.0 0.0) (a 1 0.0 0.0))

Tests the indexOf method:
Base list: ((c 1 0.0 0.0) (b 1 0.0 0.0) (a 1 0.0 0.0))
The index of 'a' is: 2
The index of 'b' is: 1
The index of 'c' is: 0
The index of 'd' is: -1

Tests the get method:
Base list: ((c 1 0.0 0.0) (b 1 0.0 0.0) (a 1 0.0 0.0))
The element at location 0 is: (c 1 0.0 0.0)
The element at location 1 is: (b 1 0.0 0.0)
The element at location 2 is: (a 1 0.0 0.0)

Tests the remove method:
Base list: ((c 1 0.0 0.0) (b 1 0.0 0.0) (a 1 0.0 0.0))
The list after removing 'b': ((c 1 0.0 0.0) (a 1 0.0 0.0))
The list after removing 'd': ((c 1 0.0 0.0) (a 1 0.0 0.0))
The list after removing 'a': ((c 1 0.0 0.0))
The list after removing 'c': ()
The list after removing 'e': ()

Tests the toArray method:
Base list: ((c 1 0.0 0.0) (b 1 0.0 0.0) (a 1 0.0 0.0))
The array elements:
(c 1 0.0 0.0)
(b 1 0.0 0.0)
(a 1 0.0 0.0)

Tests the iterator method:
Base list: ((f 1 0.0 0.0) (e 1 0.0 0.0) (d 1 0.0 0.0) (c 1 0.0 0.0) (b 1 0.0 0.0) (a 1 0.0 0.0))
The elements of the list from index 2 to the end:
(d 1 0.0 0.0)
(c 1 0.0 0.0)
(b 1 0.0 0.0)
(a 1 0.0 0.0)

Tests the update(chr) method, which works as follows:
If the character chr is not in the list, adds it to the list's beginning.
If the character chr is in the list, increments its counter.
The list before updating any element: ()
The list after updating 'a': ((a 1 0.0 0.0))
The list after updating 'b': ((b 1 0.0 0.0) (a 1 0.0 0.0))
The list after updating 'c': ((c 1 0.0 0.0) (b 1 0.0 0.0) (a 1 0.0 0.0))
The list after updating 'b': ((c 1 0.0 0.0) (b 2 0.0 0.0) (a 1 0.0 0.0))
The list after updating 'c': ((c 2 0.0 0.0) (b 2 0.0 0.0) (a 1 0.0 0.0))
The list after updating 'b': ((c 2 0.0 0.0) (b 3 0.0 0.0) (a 1 0.0 0.0))
```

Review the code of the `Test1.java` class.

The Test2 class

This class tests the construction of lists of `CharData` objects, with and without calculating probabilities. Here is the output that this class generates:

```
& java Test2
Testing the construction of a list of CharData objects from a given string input.
The probability fields of the CharData objects will be initialized to 0.
Input = "committee "
List = ((  1 0.0 0.0) (e 2 0.0 0.0) (t 2 0.0 0.0) (i 1 0.0 0.0) (m 2 0.0 0.0) (o 1 0.0 0.0) (c 1 0.0
0.0))

Testing the construction of a list of CharData objects from a given string input.
This time, the probability fields will be computed and set correctly.
Input = "committee "
List =  ((  1 0.1 0.1) (e 2 0.2 0.30000000000000004) (t 2 0.2 0.5) (i 1 0.1 0.6) (m 2 0.2 0.8) (o 1 0.1
0.9) (c 1 0.1 1.0))
```

Two comments: First, the logic for calculating the probabilities are explained in lecture 11-2. Second, don't worry about round-off errors like `0.30000000000000004`. You can treat it as `0.3`.

To Do

The objective of Stage I of this project is to develop the ability to build linked lists that represent `CharData` objects. There are six relevant classes:

- `CharData` (the code is given)
- `Node` (the code is given)
- `List`
- `ListIterator` (the code is given)
- `Test1`
- `Test2`

Your task is completing all the missing code in these classes. Try to do it until Sunday, January 10, 2021. On Sunday the rest of the project will be published.

Implementation Notes

CharData and Node:

If you reviewed the given code of these two classes, you may have noticed that their fields are neither `private`, nor `public`. This default visibility option is called in Java "package-private". Basically, it means that all the classes in the package can access (read and write) these fields directly. In this project we don't use packages, but you can assume that all the classes in the project's folder are in the same package. Therefore, all the classes can access the `CharData` and `Node` fields without having to go through getter and setter methods. This simplifies the code writing significantly.

List:

`indexOf:` Remember to skip the dummy node. Go through the list and count from 0 until you hit the correct `CharData` object. Then return the count. You can implement this logic with or without an iterator.

`get:` Similar to `IndexOf`. Instead of returning the index, return the object. Write a basic implementation without worrying about the exception. When the basic implementation is completed and tested, implement the exception.

`update:` This powerful method can be easily implemented by calling other `List` methods. Remember that you can access (read and write) the fields of `CharData` objects directly, without needing getters and setters.

`toArray:` Go through the list and put all the `CharData` objects in the array. Remember to return the array.

Test1:

Generally speaking, you have to write testing code that, when executed, generates the output shown in page 3 of this document.

`main:` Uncomment the test method that you want to run. You can run tests in isolation, or one after the other, as you please.

`testToArray:` Here's a nice opportunity to try implementing the printing process using a `for-each` statement.

`testIterator:` Start by building a list and adding `CharData` objects, as implied by the output shown in page 3. Then construct a `ListIterator` object (using a call to the `listIterator` method of `List`) and use the `hasNext()` and `next()` methods to go through the list, and print its values. See relevant examples in slides 23-27 of lecture 9-2.

`testUpdate:` Add `CharData` objects to the list, as implied by the output shown in page 3. The key thing to observe here is that when you add characters that already exist in the list, no `CharData` objects are added; Instead, the counters of these characters are incremented.

Test2:

Generally speaking, you have to write testing code that, when executed, generates the output shown in page 4.

`buildList:` This method can be easily implemented using the `List` method `update()`, which is the real workhorse here. Go through the string and update the list.

`calculateProbabilities:` Start by calculating the *total count* of all the characters in the list. For example, suppose that the list consists of the 3 characters `z`, `o`, and `l`. Suppose further the count of `z` is 1, the count of `o` is 4, and the count of `l` is 1. This implies that the total character count is 6.

(presumably, the input was `zooloo`). To calculate this total count, you have to go through the entire list. Then go through the list again, and calculate the probabilities `p` and `pp`. Calculating `p` is easy: `count` divided by total count. To calculate `pp`, set the `pp` of the first element in the list to the element's `p` value. For every other element in the list, set `pp` to the `p` value of this element plus the `pp` value of the previous element. We will have more to say about this calculation of *cumulative probabilities* in lecture 9-2.