RandomWriter.java hamlet.txt

You will write a program that reads an input file and uses it to build a large data structure of word groups called "N- grams" as a basis for randomly generating new text that sounds like it came from the same author as that file. You will use the Map and List collections. Below is an example log of interaction between your program and the user. But what, you may ask, is an N-gram?

This program makes random text based on a document. Give me an input file and an 'N' value for groups of words, and I'll create random text for you.

Input file name? hamlet.txt
Value of N? 3

# of random words to generate (0 to quit)? 40
... chapel. Ham. Do not believe his tenders, as you
go to this fellow. Whose grave's this, sirrah?
Clown. Mine, sir. [Sings] O, a pit of clay for to
the King that's dead. Mar. Thou art a scholar; speak
to it. ...

# of random words to generate (0 to quit)? 20
... a foul disease, To keep itself from noyance; but
much more handsome than fine. One speech in't I
chiefly lov'd. ...

# of random words to generate (0 to quit)? 0
Exiting

The "Infinite Monkey Theorem" states that an infinite number of monkeys typing random keys forever would eventually produce the works of William Shakespeare. That's silly, but could a monkey randomly produce a new work that "sounded like" Shakespeare's works, with similar vocabulary, wording, punctuation, etc.? What if we chose words at random, instead of individual letters? Suppose that rather than each word having an equal probability of being chosen, we weighted the probability based on how often that word appeared in Shakespeare's works?

Picking random words would likely produce gibberish, but let's look at chains of two words in a row. For example, perhaps Shakespeare uses the word "to" occurs 10 times total, and 7 of those occurrences are followed by "be", 1 time by "go", and 2 times by "eat". We can use those ratios when choosing the next word. If the last word we chose is "to", randomly choose "be" next 7/10 of the time, "go" 1/10 of the time, and "eat" 2/10. We never choose any other word to follow "to". We call a chain of two words like this, such as "to be", a 2-gram.

Go, get you have seen, and now he makes as itself? (2-gram)

A sentence of 2-grams isn't great, but look at chains of 3 words (3-grams). If we chose the words "to be", what word should follow? If we had a collection of all sequences of 3 words-in-a-row with probabilities, we could make a weighted random choice. If Shakespeare uses "to be" 22 times and follows them with "or" 5 times, "in" 3 times, "with" 10 times, and "alone" 4 times, we could use these weights to randomly choose the next word. So now the algorithm would pick the third word based on the first two, and the fourth based on the (second+third), and so on.

One woe doth tread upon another's heel, so fast they follow. (3-gram)

You can generalize the idea from 2-grams to N-grams for any integer N. If you make a collection of all groups of N-1 words along with their possible following words, you can use this to select an Nth word given the preceding N-1 words. The higher N level you use, the more similar the new random text will be to the original data source. Here is a random sentence generated from 5-grams of Hamlet, which is starting to sound a lot like the original:

I cannot live to hear the news from England, But I do prophesy th'election lights on Fortinbras. (5-gram)

Each particular piece of text randomly generated in this way is also called a Markov chain. Markov chains are very useful in computer science and elsewhere, such as artificial intelligence, machine learning, economics, and statistics.

Step 1: Building Map of N-Grams

In this program, you will read the input file one word at a time and build a particular compound collection, a map from prefixes to su□ixes. If you are building 3-grams, that is, N-grams for N=3, then your code should examine sequences of 2 words and look at what third word follows those two. For later lookup, your map should be built so that it connects a collection of N-1 words with another collection of all possible su□ixes; that is, all possible N'th words that follow the previous N-1 words in the original text. For example if you are computing N-grams for N=3 and the pair of words "to be" is

followed by "or" twice and "just" once, your collection should map the key {to, be} to the value {or, just, or}. The following figure illustrates the map you should build from the file:

When reading the input file, the idea is to keep a window of N-1 words at all times, and as you read each word from the file,discard the first word from your window and append the new word. The following

figure shows the file being read and the map being built over time as each of the first few words is read to make 3-grams:

File Location	Data	
to be | or not to be just	map	= {}
	window	= {to, be}
to be or | not to be just	map	= { {to, be} : {or} }
	window	= {be, or}
to be or not | to be just	map	= { {to, be} : {or},
{be, or} : {not} }		
	window	= {or not}
to be or not to | be just	map	= { {to, be} : {or},
		{be, or} : {not} ,
{or, not} :{to}}		
	window	= {not, to}
to be or not to be | just	map	= { {to, be} : {or},
		{be, or} : {not} ,
		{or, not} :{to},
{not, to} : {be}}		
	window	= {to, be}
to be or not to be just | ...	map	= { {to, be} : {or, just},
		{be, or} : {not} ,
		{or, not} :{to},
		{not, to} : {be}}
	window	= {be, just}
to be or not to be just be who you want to be or not okay you want okay	map	= {{to, be} : {or, just, or}, {be, or} : {not, not}, {or, not} : {to, okay}, {not, to} : {be}, {be, just} : {be}, {just, be} : {who}, {be, who} : {you}, {who, you} : {want}, {you, want} : {to, okay}, {want, to} : {be},

{not, okay} : {you},
{okay, you} : {want},
{want, okay} : {to},
{okay, to} : {be} }

Note that the order matters: For example, the prefix {you, are} is di□erent from the prefix {are, you}. Note that the same word can occur multiple times as a su□ix, such as "or" occurring twice after the prefix {to, be}.

Also notice that the map wraps around. For example, if you are computing 3-grams like the above example, perform 2 more iterations to connect the last 2 prefixes in the end of the file to the first 2 words at the start of the file. In our example above, this leads to {want, okay} connecting to "to" and {okay, to} connecting to "be". If we were doing 5-grams, we would perform 4 more iterations and connect the last 4 prefixes to the first 4 words in the file, and so on. This turns out to be very useful to help your algorithm later on in the program.

You should not change case or strip punctuation of words as you read them. The casing and punctuation turns out to help the sentences start and end in a more authentic way. Just store the words in the map as you read them.

Step 2: Generating Random Text

To generate random text from your map of N-grams, first choose a random starting point for the document. To do this, pick a randomly chosen key from your map. Each key is a collection of N-1 words. Those N-1 words will form the start of your random text. This collection of N-1 words will be your sliding "window" as you create your text.

For all subsequent words, use your map to look up all possible next words that can follow the current N-1 words, and randomly choose one with appropriate weighted probability. If you have built your map the way we described, as a map from {prefix} → {suffixes}, this simply amounts to choosing one of the possible suffix words at random. Once you have chosen your random suffix word, slide your current "window" of N-1 words by discarding the first word in the window and appending the new suffix.

Since your random text likely won't happen to start and end at the beginning/end of a sentence, just prefix and suffix your random text with "..." to indicate this. Here is another partial log of execution:

Input file? tiny.txt

Value of N? 3

# of random words to generate (0 to quit)? 16

... who you want okay to be who you want to be or not to be or ...

Development Strategy and Hints

This program can be tricky if you don't develop and debug it step-by-step. Don't try to write everything all at once. Make sure to test each part of the algorithm before you move on.

1. Think about exactly what types of collections to use for each part. Are duplicates allowed? Does order matter? Do you need random access? Where will you add/remove elements? Etc. Note that some parts of each program require you to make compound collections, that is, a collection of collections.

2. Test each function with a very small input first. For example, use input file tiny.txt with a small number of words because you can print your entire map and examine its contents.

3. Recall that you can print the contents of any collection and examine its contents for debugging.

4. To choose a random prefix from a map, consider using the map's keys member function, which returns a List containing all of the keys in the map.

Sample Run:

```
Enter the seed value:
5
Enter the size of N-Gram:
2
Enter the number of words to generate(0 to quit)?
100
... him: be ground, And my you tutor: suit the are HAMLET Your wisdom should
break Purpose is should with these words are HAMLET Make you behavior hath
made men fear grow proverb is should with these are HAMLET Or like it in these
ventages with these are HAMLET It you griefs to him. HAMLET It you mouth,
and that. Will you pardon and that. Will you own spoke my you mother, in these
words soever she To KING CLAUDIUS O, confound the are HAMLET I OPHELIA,
ROSENCRANTZ, and that. Will he are HAMLET Bid the are HAMLET Sir, a but,
...
```