CSCI 340
Data Structures and Algorithms
Spring 2019

Project 3 - Spellchecker

Points: 40
Due date: Thursday, March 14

This project is similar to one used at UC Irvine.

In this project you will develop a data structure known as a Trie. A Trie fills the same data structures niche as a HashSet<String>. You are required to use a Trie. Using a HashSet<String> or other data structure instead of the Trie will result in a grade of 0.

Background
Many applications, such as word processors, text editors, and email clients, include a spell-checking feature. A spellchecker's job is relatively simple. Given a list of correctly-spelled words, which we'll call a *lexicon*, and the input text, the spellchecker reports all of the *misspellings* (i.e. words that are not found in the lexicon) in the input. When a misspelling is found in the input, the spellchecker will also suggest words appearing in the lexicon that are somewhat like each of the misspelled words.

As an example, suppose that the lexicon contains the following words:

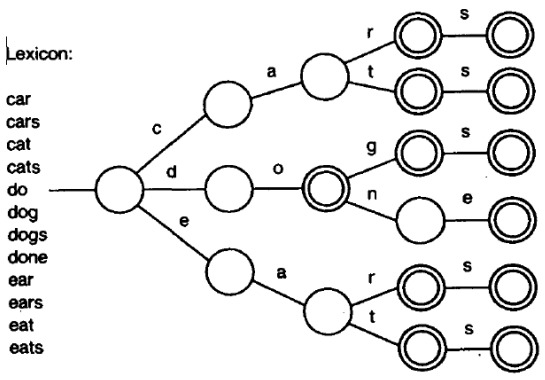
```
bake cake main rain vase
```

If the input file contains the word **vake**, a good spellchecker will state that **vake** is misspelled, since it does not appear in the lexicon, and will suggest that perhaps **bake**, **cake**, or **vase** was the word that was actually intended.

Of course, a spellchecker's task is centered around searching for words in a potentially large lexicon. An efficient search is critical to the performance of the spellchecker, since it will be searching the lexicon not only for the words in the input, but possibly hundreds of suggestions that it might like to make for each misspelling. We will implement our lexicon using a Trie.

Your Assignment – SpellChecker.java
You are to write a program that implements a rudimentary spellchecker. It should use a inner class named Lexicon that is implemented as a Trie.

The Lexicon Class
A Trie is a tree where every node can have up to 26 children one for each letter of the alphabet. Below is a figure of a Trie containing just a few short words beginning with c, d, and e. The nodes that represent the end of words have concentric circles. In your Trie, each node should have an array of 26 children, as well as a `boolean` value, named something like `isWord`. `isWord` is set to true only if this node represents a word.



The Lexicon class has a constructor and one public method. You are, of course, free to use private helper methods. These will not be tested.

The constructor should read all the words from `enable1augmented.txt` and build a Trie containing them. `enable1augmented.txt` is a list of thousands of English words, that has been augmented with some contractions at the end, without the apostrophes.

`public boolean containsWord(String word)` which returns true if the word is in the lexicon, false otherwise. Once the Trie is constructed, searching it is straightforward. Say we are searching for the word "cat". We start at the root, go to the 'c' subtree, go to its 'a' subtree and go to its 't' subtree. At this point we look and see that the `isWord` is true, so we return true. That is "cat" is a word.

There are two ways that false may be returned.

- If we come to the 't' node and `isWord` is false, then we return false.
- We might also come to the point where there is no child node to move to. In the example above, if we search for "does" we will visit the 'd' subtree and its 'o' subtree, but there is no 'e' subtree from there. In this case, we also return false.

Looking up words in a Trie is a O(k) operation where k is the number of letters in the word.

The Spellchecker Program

Your program should create the lexicon, then prompt the user for an input filename. You should process the input file line at a time. After reading in the line of text, you should make it into all lowercase and remove all punctuation. Then, each word in the line is looked up in the lexicon. If it is not found, the program writes the line number (start counting at 1), the word that was misspelled, and a sorted list of suggestions. For example, given this input file:

```
This is a lne of text that has a missspelling in it.
```

generates the following output using the default wordlist file, `enable1augmented.txt`:

```
Line 1: "lne" is not spelled correctly!
Suggestions:
[ane, lane, lee, lie, line, lone, lune, lye, ne, one]

Line 1: "missspelling" is not spelled correctly!
Suggestions:
[miss spelling, misspelling]
```

Generating the Suggestions

There are two popular text-mode spell checkers on Unix/Linux systems. One is called *ispell*; the other is a GNU "free software" program called *aspell*. Both use similar techniques for generating suggestions for misspelled words. Here are five typical techniques:

- Swapping each adjacent pair of characters in the word.
  Example: "hte" should generate the suggestion "the"
- In between each adjacent pair of characters in the word (also before the first character and after the last character), each letter from 'a' through 'z' is inserted.
  Example: "lne" should generate the suggestions "lane" and "line"
- Deleting each character from the word.
  Example: "missspelling" should generate the suggestion "misspelling"
- Replacing each character in the word with each letter from 'a' through 'z'.
  Example: "lqne" should generate the suggestions "lane" and "line"
- Splitting the word into a pair of words by adding a space in between each adjacent pair of characters in the word. It should be noted that this will only generate a suggestion if *both* words in the pair are found in the lexicon.
  Example: "missspelling" should generate the suggestion "miss spelling"

Your SpellChecker class should generate suggestions using all of these techniques.

It should be noted that there are other ways to generate suggestions, including using algorithms that pay attention not only to the letters, but what the letters actually sound like. One such well-known algorithm is called the Soundex algorithm. You are **not** required to implement such algorithms for this project.

Notes

1. You may hardcode the word list file name. Before you turn in the program, make sure the filename refers to `enable1augmented.txt`. **Do NOT** make a copy of this file into your lab account. It is 1.7M is size. You may, of course, make a copy onto your own computer, but if you do so, make certain that your submitted project refers either to the one in the course directory or a local copy. E.g. Do not include a path starting with C:\\ or the equivalent.
2. Your program should ignore capitalization.
3. Your program should ignore all punctuation. For example "Holy cow!" should be treated as two words, "holy" and "cow". Leading and trailing punctuation should be eliminated. Dashes and hyphens occur inside of words and should be replaced by spaces. Apostrophes may appear inside of words. These should just be eliminated.
4. Your instructor has included the following files for you test your code:
    a. `enable1augmented.txt` which is described above.
    b. The following files to test your code when using enable1augmented.txt
        i. `OCaptainMyCaptain.txt` which contains Walt Whitman's famous eulogy to Abraham Lincoln.
        ii. `WhereTheSideWalkEnds.txt` which contains Shel Silverstein's famous poem.
    c. `tinyLexicon.txt` which contains the words in the example above.
    d. The following file to test your code when using tinyLexicon.txt
        i. `nonsense.txt`
    e. `scrabble.pdf` which is a research paper from the 1980s that uses a Trie and a related optimized data structure called a Dawg to play Scrabble.

What to turn in:

Submit a single java source file named `SpellChecker.java`. Do not submit any data files. These are large files and there is no need to use up disk space submitting them.

Do not use package statements in your code.