Programming Assignment: Spelling Suggestions

In this assignment, we'll implement the highly practical feature of giving someone suggestions for how to correct a misspelled word.

This project will consist of two parts.

1. The first is creating a class which allows us to do single operation mutations of words. For example, it will let us find the word "his" by removing a single letter from "this".
2. The second part will enable us to look through these "nearby" words to find spelling suggestions. So if you mistype and write "swone" it will suggest to you words like "stone".

Getting Set Up

Before you begin this assignment, make sure you check Part 2 in the setup guide to make sure the starter code has not changed since you downloaded it. If you are an active learner, you will have also received an email about any starter code changes. If there have been any changes, follow the instructions in the setup guide for updating your code base before you begin.

1. Find the starter code

You should see a package called **spelling** in that starter code. You've worked in this package before.

The .java files we'll be focusing on in this assignment are NearbyWords and SpellingSuggest.

Assignment and Submission Details

Your submission of this assignment is divided into the three parts above. We recommend you finish part 1 (submit and have pass grading) before moving to part 2, and so forth. As always, we strongly recommend you write your tests as you write your implementation (and vice-versa).

Part 1: Implement letter mutation methods in Nearby Words.

1. A. Review the methods provided:

To help you get started, we've provided you with the following methods. Note that you will not need to deal with capital/lower-case letters, you can assume everything is lower-case.

public NearbyWords(Dictionary dict)

This constructor sets the instance variable dictionary. Our mutation methods need a dictionary to filter by only those mutations which result in real words (this is used heavily in the next assignment - Word Paths).

public List<String> distanceOne(String s, boolean wordsOnly)

This method constructs a list of String which are one mutation away from the parameter String s. It will limit the list of Strings to only real words if the boolean flag wordsOnly is true. It constructs this list by passing the list to the three methods "insertions", "substitutions", and "deletions".

public void substitutions(String s, List<String> currentList, boolean wordsOnly)

This method produces all substitutions of each letter in String s with any letter from the alphabet and adds them to the List currentList (if not already present and not the original word). It also uses the boolean wordsOnly flag along with access to the Dictionary object to select only real words when wordsOnly is true.

1. B. Author the following methods in NearbyWords:

public void insertions(String s, List<String> currentList, boolean wordsOnly)

This method behaves just like substitute above, except it tries to insert any letter in between existing letters (or to the start and end) to create new Strings (which it adds to currentList).

public void deletions(String s, List<String> currentList, boolean wordsOnly)

This method behaves just like substitute above, except it tries to remove any existing letter to create new Strings (which it adds to currentList).

Test your methods

If you didn't test your methods as you developed them, this is a good time to make sure that the methods produce all the String mutations you'd expect and that it properly selects just real words when the boolean wordsOnly is true. (Again, you'll use this flag in the WordPath assignment next.)

Part 2: Complete the method suggestions in NearbyWords.

public List<String> suggestions(String word, int numSuggestions)

This method allows us to implement the SpellingSuggest Interface and is very similar to the Breadth First Search you performed with AutoComplete last week. In this method, we'll be looking at providing a number of spelling suggestions for a misspelled word. To do this, we'll be using our methods which allow us to look at Strings one mutation (insert, delete, substitute) away.

The basic algorithm is below:

```
Input:  word for which to provide number of spelling suggestions

Input:  number of maximum suggestions to provide

Output: list of spelling suggestions



Create a queue to hold words to explore

Create a visited set to avoid looking at the same String repeatedly

Create list of real words to return when finished



Add the initial word to the queue and visited



while the queue has elements and we need more suggestions

  remove the word from the start of the queue and assign to curr

  get a list of neighbors (strings one mutation away from curr)

  for each n in the list of neighbors

     if n is not visited

       add n to the visited set

       add n to the back of the queue

       if n is a word in the dictionary
```

```
          add n to the list of words to return



return the list of real words
```

Hints

1. You will likely want feedback from your code as you are writing this method. To do this, you'll likely want to print the contents of the queue per while loop iteration or something similar. If you get stuck, just add more print statement to help debug.
2. We've provided a test case for suggestions in the NearbyWords main method. Feel free to use this to test other words. However, until you complete the Optimization below, don't try asking for a large number of suggestions from unique words. For example, asking for 10 suggestions for the mispelled word "kangaro" will likely cause your code to run for an extremely long time. See the optimization below for details.

Optional Optimization

If you've completed the code above, you should be able to ask for words like "tailo" and get a number of good suggestions. However, asking for 10 suggestions for a word like "kangaro" can cause serious trouble in terms of runtime. Why? Think about this for a minute before continuing.

Okay, hopefully you recognized that the search is going to get huge after finding kangaroo and kangaroos. Tons of nonsensical Strings will be explored in the path to trying to find other suggestions. Now, if you let the code run for a really long time (and you had a HUGE memory), it'd eventually find other words to suggest, but those words are so distant from kangaro that they are inappropriate suggestions.

To avoid this, you should use the variable THRESHOLD to stop the search after looking at a threshold number of Strings. We set the threshold at 1000, but you can try playing with this to see the results.

What and how to submit

Submission Part 1: Submit NearbyWords for Feedback on your String mutation methods

Upload the NearbyWords.java file for testing. Upload only the single NearbyWords.java file, NOT a zip file. The testing will test the behavior of your **insertion** and **deletion** methods. If you were diligent in self-testing, this should pass with flying colors. But don't worry if we catch a few cases you missed.

Submission Part 2: Submitting NearbyWords for Suggestions Feedback

upload the NearbyWords.java file for testing. Upload only the single NearbyWords.java file, NOT a zip file. This testing will examine the behavior of your **suggestions** method. The tests will not require you complete the optimization (so we won't test words like "kangaro"), but we still encourage you to do so.