Optional Programming Assignment: Word Paths

In this part of assignment, we'll use the same ideas from the last assignment (spelling suggestions) to have some fun. Specifically, we'll be implementing a game we call Word Paths.

Word Paths is a game where we'll try to find a path from one word to another with the restriction that we can only make one change at a time (letter permutation, letter insertion, letter deletion) and that whatever change we make, has to result in a real word.

For example, I can create a path from the word "time" to "theme" through 4 changes (or 5 total words including "time" and "theme"):

time -> tie -> tee -> thee -> theme

The changes below you'll recognize from spelling suggestions:

time -> tie results from a character deletion (m)

tie -> tee results from a character modification (i to e)

tee -> thee results from a character addition (e)

thee -> theme results from a character addition (m)

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 WPTree and WordPath.

2. Be sure you have finished the previous assignment on Spelling Suggestions

We'll be using the methods for word mutations in NearbyWords and the principles from spelling suggestions to complete this assignment.

Word Paths

1. Familiarize yourself with the starter code

1.a. First, review the public interface:

public interface WordPath

The abstract method in WordPath to override is:

public List<String> findPath(String word1, String word2)

This method will find a path from word1 to word2.

1.b. Second, review the provided code in WPTree

WPTree consists of the public class WPTree which implements WordPath. The methods provided include:

- a no argument constructor which will simply set the root of the tree to be null and will create a NearbyWords object (you need to add this).
- printQueue is a method which will help print a list of WPTreeNodes. This will be helpful when testing your find path method.

WPTreeNode has been entirely written for you. You should not need to change this class in anyway. A WPTreeNode contains a word, it's parent, and its children.

Review the methods in WPTreeNode as they will be highly useful to you. The methods provided include:

- a constructor which requires a String (the word) and a parent node (null for root). This allows you to create a node while connecting it back to its parent. (Note, it does not update the list of children for the parent, this needs to be done separately - see addChild method).
- getWord which returns the node's word

- addChild which creates a node (using the provided String) and makes it the child of the calling object node. It returns the new node if you need to use it.
- getChildren returns the list of children for a node
- buildPathToRoot is an essential method which will let you build a path from the root node to the calling object node.
- toString is useful for debugging/testing

1.c. Third, Author the findPath method

The core of this assignment is the creation of the findPath method. findPath will function very similarly to the suggestions method you authored in the prior NearbyWords assignment. The fundamental difference between these two methods is that suggestions simply searched for nearby words, whereas findPath must find, and return, the path to a specific word. To be able to return the path, we need to have a way of reconstructing how we got to the target word - and we'll do this by creating a tree as we search.

So the basic idea is to do a Breadth First Search while dynamically building a tree so you can reconstruct the path from word1 to word2.

If you want a significantly larger challenge, stop reading now and try to create your own algorithm on how to do this. Otherwise, please review the algorithm below carefully before starting to code:

```
Input:  word1 which is the start word

Input:  word2 which is the target word

Output: list of a path from word1 to word2 (or null)



Create a queue of WPTreeNodes to hold words to explore

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



Set the root to be a WPTreeNode containing word1

Add the initial word to visited

Add root to the queue
```

```
while the queue has elements and we have not yet found word2

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

  get a list of real word neighbors (one mutation from curr's word)

  for each n in the list of neighbors

     if n is not visited

       add n as a child of curr

       add n to the visited set

       add the node for n to the back of the queue

       if n is word2

          return the path from child to root



return null as no path exists
```

Hints

1. In the last assignment, we created the NearbyWords distanceOne methods to allow us to restrict the responding list to only dictionary (real) words. This is going to be very useful here (to avoid pruning non-dictionary words). Don't forget to create a NearbyWords object in the constructor.
2. The method addChild in WPTreeNode returns back the new child node. This can be useful when constructing the tree.
3. The methods printQueue and WPTreeNode toString will be essential in debugging and testing your code. Feel free to add a main method to WPTree to test WPTree outside of the GUI interface. For example, you might want to test findPath between varying words and print the queue contents at various points when the algorithm is running.
4. Think about corner cases. For example:

Should it run for a long time if you give it a basic word and ask it to try to reach an obscure word (like a path from "the" to "kangaroo")? Yes - it won't find a path, but it takes a while to realize that depending on the size of the dictionary.

What should happen if the user gives you a word2 which isn't a word in the dictionary? No path will exist, because you'll never find word2 as a dictionary (real) word when calling the distanceOne method.

What and how to submit

Submission Part 1: Submit WPTree for feedback on your findPath method

You will need to upload the WPTree.java file for testing. The testing will test the behavior of your **findPath** method. 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.