CS145 PROGRAMMING ASSIGNMENT

LEVENSHTEIN DISTANCE

OVERVIEW

This program focuses on programming with Java Collections classes. You will implement a module that finds a simplified Levenshtein distance between two words represented by strings.

Your program will need support files `LevenDistanceFinder.Java`, `dictionary.txt`, while you will be implementing your code in the `LevenDistanceFinder.java` file.

INSTRUCTIONS

The Levenshtein distance, named for it's creator Vladimir Levenshtein, is a measure of the distance between two words. The edit distance between two strings is the minimum number of operations that are needed to transform one sting into the other. For a full implementation of the distance we would need to account for three different types of operations, removing a letter, adding a letter, or changing a letter.

For THIS program however, we are going to focus solely on the ability to change one letter to another. So, we will be thinking of changing one letter at a time, while maintaining the fact that the word is a still a valid word at all times.

So, for an example, the edit distance from "dog" to "cat" is 3. One possible path for this is

```
"dog" to "dot" to "cot" to cat"
```

Note, that there might be other paths, but they all involve at least 3 changes. Another longer example is from "witch" to "coven".

```
witch->watch->match->march->marcs->mares->mores->moves->coves->coven
```

You will be implementing a secondary class called `LevenshteinFinder` that will be used by the main program to solve the problem.

SAMPLE EXECUTIONS

This is what the program should like when you run it.

```
* Welcome to the Levenshtein Edit distance solver                *
* Please type in two words of the same size that will be used   *
* To find a path between them.                                   *
*****************************************************************

What word would you like to start with?
--->Witch
What word would you like to end with?
--->Coven
The distance between your words is 9
The path between your words is : witch->watch->match->march->marcs-
>mares->mores->moves->coves->coven
*****************************************************************
Total execution time: 14407ms.
```

```
* Welcome to the Levenshtein Edit distance solver                *
* Please type in two words of the same size that will be used   *
* To find a path between them.                                   *
*****************************************************************

What word would you like to start with?
--->dog
What word would you like to end with?
--->cat
The distance between your words is 3
The path between your words is : dog->dot->cot->cat
*****************************************************************
Total execution time: 1262ms
```

```
* Welcome to the Levenshtein Edit distance solver                *
* Please type in two words of the same size that will be used   *
* To find a path between them.                                   *
*****************************************************************

What word would you like to start with?
--->bookkeeper
What word would you like to end with?
--->sunglasses
The distance between your words is -1
The path between your words is : There is no path
*****************************************************************
Total execution time: 23378ms.
```

GETTING STARTED

In order to get started with this assignment, start by downloading the Programming Assignment - CS145 - Leven Distance ZIP file from Canvas. In it you will find the following two files.

File	Actions
`dictionary.txt`	The primary list of words for the game to use.
`LevenDistanceFinder`.java	This is the file that you will run. No major modifications can be made to this file.

THE GENERAL ALGORITHM

From your initial list of words, compute a map from every word to its immediate neighbor words. A neighbor word is a word that is exactly the same except for one character is changed.

So DOG and DOT are neighbor words, as well as DOT and COT. But DOG and COT are not as they require two changes.

Once you have this map built, you can use it to find the distance between two words.

Then assuming the distance between two words is at least zero, you can then find the link between two words.

YOUR INSTRUCTIONS

You are given a client program `LevenDistanceFinder.java` that does the file processing and user interaction. `LevenDistanceFinder.java` reads a dictionary text file as input and passes its entire contents to you as a list of strings. You are to write a class called `LevenshteinFinder` that will manage the actual work of finding the distance and the path.

The class will probably have the following necessary private fields:

- Starting and Ending Strings
- A map of words to a set of neighbors.
- An integer distance that starts at -1.
- A List of strings that is the path itself.

NECESSARY METHODS

PUBLIC LEVENSHTEINFINDER(STRING, STRING, SET<STRING>)

Your constructor is passed a string which is the starting word of the path. A string that is the ending word of the path, and a collection of words as a set. It should use these values to initialize the class. The set of words will initially be all words from the dictionary.

You should throw an `IllegalArgumentException` if length of the two words is not the same.

You will want to store the starting and stopping words for later use, but you should not need to save the words list.

Using the word list, first you will want to pull out only the words that are of the correct size. Then from this smaller list, what you will want to do is to create a map of words to their "neighbor" words. Start this by going through every word and add it and an empty set, to a map.

You will then go through the set a second time and check every word to every other word. If they are "neighbors" you will add each word to the set that goes with the other word. So if stringA and stringB are neigbors, then you add stringA to stringB's set, and vice versa.

Once the neighbor map is created, and you are still in the constructor, you will run the `findDistacnce` method and the `findPath` method to save those values for future delivery.

Note that this is all done in the constructor, so that all the work is done when it is “newed”/created.

PRIVATE INT DIFFERENTLETTERS(STRING A, STRING B)

This method should take two strings and find the number of letters that are different from each other and return that number.

PUBLIC INT GETDISTANCE()

This method is called by the main program to get the distance between the two words. Note that you found this in a different method. So this should just return a saved value. If it is longer than 2 lines, you are doing it wrong.

PUBLIC STRING GETPATH()

This method returns a string that is the path from the first word to the second word, assuming it exists. If the path distance is negative one, then return back an error message `“There is no path”`, if the path distance is zero or higher, then take the pathArray and convert it to a string with arrows in between.

Example: `love->lave->late->hate`

PRIVATE INT FINDDISTANCE(STRING A, STRING B)

This method finds the distance between two strings. *(Note, only the distance, not the path itself).* Start by creating two empty sets. Add the staring word to the second set. Set a counter to zero. Then while the sets are different sizes and the 2nd set does not contain the ending word, add everything from set 2 into set one, clear the 2nd set, and then take every word from set1 AND the neighbor of every word from set1 into set 2. Increment the loop counter.

When the loop finishes, if the 2nd set contains the final word return the counter because it was found, if the 2nd set doesn't contain the word, return -1 because there is no path.

PRIVATE VOID FINDPATH(STRING A, STRING B)

This method will find the path between two words. It should probably be the last method you work on. In fact I would create it, and have it do nothing until you are ready for it.

When running this method should only be run after findDistance() so that the distance has already been found and stored in a private int value.

When you are ready for it, this method should do the following.

Initialize the class path List to a new empty List.

Check the distance, if it is negative, store an error message in the path List and then exit the method.

If the distance is zero or positive add the first element to the list.

Now in a loop that starts at the distance minus 1, and goes down until 1, look at the set of neighbors of the word in the last box of the list. Find one that has a distance to the ending word that matches the current loop counter. There may be multiples, but there has to be at least one. Find this word, and add it to the list.

Now repeat the loop until the for loop quits. Then add the ending word to the list.

You are done.

Here is an example for the path from `love -> hate`.

> The distance from love to hate is 3, so that is bigger than -1. So add love to the list. The list is now size one. Now start your loop from distance -1 (2) to 1. So i is currently 2. Find any word in the neighbor of love, that has a distance to hate of size 2. *Use your findDistance method here!*. One of those words is "lave". So add that to the array.
>
> Next round, i is one. The list is now [love, lave]. So find a neighbor of "lave" that has a distance to "hate" of one. One possible word is "late". So add "late" to the list which now looks like [love, lave, late]
>
> That should finish the loop, add "hate" to the list. The final list looks like [love, lave, late, hate]. Store that list in the class field. And you are done.

OTHER INSTRUCTIONS

Your program should exactly reproduce the format and general behavior demonstrated on the previous pages.

You may assume that the list of words passed to your constructor will be a nonempty list of nonempty strings composed entirely of lowercase letters.

Use the `TreeMaps, TreeSets` and `ArrayLists` implementations for all of your ADT's.

HINTS

If you have not had to deal with exceptions before the proper way to throw and exception is the following: `throw new IlleagalStateException()` or `IllegalArgumentEXception()` as appropriate. This will end your method and return back to the primary program.

STYLE GUIDELINES AND GRADING:

You should introduce private methods to avoid redundancy and to break up large methods into smaller methods. In particular, **you should not have any methods that have more than 50 lines of code in their body** (not counting blank lines and lines that have just comments or curly braces). If you have a method that requires more than 50 lines of code, then you should break it up into smaller methods.

Make sure your output looks exactly like the sample output up above, you don't want to lose points because you decided to look a little different.

You may only use techniques and ideas that you have learned in this class. You are not allowed to use more advanced techniques that you might find on the internet. I want you to practice using the collection tools available as well as the idea of following a set plan.

You should follow general good style guidelines such as: making fields `private` and avoiding unnecessary fields; declaring collection variables using interface types; appropriately using control structures like loops and `if/else`; properly using indentation, good variable names and types; and not having any lines of code longer than 100 characters. Comment your code descriptively in your own words at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, pre/post-conditions, and exceptions.

For reference, one solution is around 150 lines long including comments and blank lines (100 "substantive" lines). However this is not a requirement, your code may be as long/short as you need it to be. But if you find yourself going excessively long, you may want to take a step back and reconsider your approach.

Note that this program may take some time to run. If you look at the examples above, “witch” and “coven” took 15 seconds to finish, and “bookkeeper”/“sunglasses” took even longer.