Spell Checking and Autocomplete

In this assignment you'll be adding the ability to flag misspelled words as well as perform auto-complete suggestions for the user's text as they type. This last part is particularly exciting when you get it working--your text editor will start to behave just like the editor on your phone!

There are two parts to this assignment, and as usual you'll submit a separate file for each part.

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.

Also, you'll need to have Eclipse open with the starter code, as usual. We'll be working with an entirely new package (the spelling package) so it's not important that your code from Modules 1-3 work perfectly for you to get this part working.

Part 1: Spell checking and more benchmarking

In this module you learned about binary search trees, and how they can be used to store keys to facilitate more efficient insertion and retrieval than a linked list. In this section you will implement a dictionary of words, and compare the performance of using a LinkedList vs a Binary Search Tree for this implementation. In doing this implementation and comparison, you will also be adding the ability to highlight misspelled words in the text editor.

1. **Implement the DictionaryLL class** which implements the Dictionary interface using a LinkedList data structure. For this assignment, we are ignoring the case of words. Your dictionary should store all words as lower case words, i.e. they should be converted to lower case before you put them in. Similarly, your dictionary should convert a word to lower case before it checks to see whether it is in the Dictionary. You will have the option to enable case sensitivity (e.g. so "Christine" is considered a word, but "christine" or "CHristine" is flagged as misspelled) in the optional extension to this project, described below.

Test your class by running the JUnit test suite we provide. Note: there is very little to do to implement this class so if you think it's too easy, you're probably doing it right.

2. **Implement the DictionaryBST class** which implements the Dictionary interface using a TreeSet (balanced Binary Search Tree) data structure. For this assignment, we are ignoring the case of

words. Your dictionary should store all words as lower case words, i.e. they should be converted to lower case before you put them in. Similarly, your dictionary should convert a word to lower case before it checks to see whether it is in the Dictionary. You will have the option to enable case sensitivity (e.g. so "Christine" is considered a word, but "christine" or "CHristine" is flagged as misspelled) in the optional extension to this project, described below.

Test your class by running the JUnit test suite we provide. Note: there is very little to do to implement this class so if you think it's too easy, you're probably doing it right.

3. **Predict the running time to find words in each Dictionary implementation** as a function of n, the number of words in the dictionary. (Technically because you are converting words to lowercase before you check whether they are in the dictionary, the running time also depends on the length of the word, but we will ignore that here by always looking for a word of the same length).

4. **Run the DictionaryBenchmarking class** to time your two dictionary implementations. We have provided the full implementation of this class, but you might find it helpful to play around with the settings by changing the values of the variables at the top of the file to get better results. Note that you are testing worst case running time here by always looking for a word that is not present in the dictionary.

5. Copy and paste your output to Google Sheets as you did for the assignment in Module 2 so that you can **graph the running times of both data implementations as the size of the dictionary grows**. You will probably find that the data is very noisy, and in particular you might NOT see a trend you expect with the DictionaryBST class. Try to think about why that might be. We'll ask you about it on the quiz.

6. **Run the text editor application and see the results of your dictionary implementation**. Now, when you run the application and check the box for Spelling Suggestions, you will see misspelled words highlighted. You won't be able to see spelling suggestions for the word yet. That will come in the next Week's assignment.

What to upload for Part 1

Create a zip file containing the files DictionaryLL.java and DictionaryBST.java and upload this zip file for grading. You shouldn't submit until you've uploaded all parts (or you will lose your submission attempt).

You can see the exact tests that we will run in the file DictionaryGrader.java. Please refer to that file when interpreting your grading results. You can even run this file to see exactly what output your code produces on our grading tests, and then use the grading feedback to figure out which tests are producing incorrect output to help you fix your errors and submit again.

Part 2: Autocomplete

Next you will implement a Trie data structure to enable the autocomplete functionality of your text editor. That is, when the user types partial words, the editor will give suggested completions to choose from in a drop down menu under the word.

1. **Examine the files AutoComplete.java and Dictionary.java**. These are the two interfaces that your AutoCompleteDictionaryTrie class will implement. You do not need to change these files. Just read the comments and understand what each method is supposed to do.

2. **Implement the methods addWørd, size, isWord and predictCompletions in AutoCompleteDictionaryTrie.java, using a trie data structure to store the words in a dictionary.** Each method is described in more detail in the comments in the provided code.

As in part 1, your implementation should convert each word to lowercase before inserting it into the trie. When you look for a word in the trie, you should look for the all lowercase version of the word. The methods Character.toLowerCase and/or String.toLowerCase will come in handy here.

Your implementation should make use of the TrieNode class we have provided for you. You should not need to modify this class at all, but if you want to you may add methods to this class, but you should not remove or modify any of these methods.

Hint for predictCompletions: To write the predictCompletions method, use the breadth first search algorithm Leo described this week. You'll need a queue, which you can produce by creating a LinkedList and always adding nodes to the back (addLast) and removing them from the front (removeFirst). In fact, this is the default behavior of the add and remove methods in a LinkedList! There is some starter code and comments in the provided file that will give you more hints about how to complete this part.

We have also provided for you a helper method to print the trie using a pre-order traversal. You may find this method useful in debugging your code.

You can test your methods using the AutoCompleteDictionaryTrie tester class we provide. You might want to add additional tests to this class. You can also find the method calls we will use to grade your implementation in the file TrieGrader.java. See more about this file in the "What to upload for Part 2" section below.

What to upload for Part 2

For this part, create a zip file containing AutoCompleteDictionaryTrie.java, *along with any other files you have modified that AutoCompleteDictionaryTrie depends on.* Upload this file as your submission. We will run grading scripts on the Coursera platform against your code and provide you feedback with which tests passed, and if any failed. You can see the exact tests that we will run in the file TrieGrader.java. Please refer to that file when interpreting your grading results. You can even run this file to see exactly what output your code produces on our grading tests, and then use the grading feedback to figure out which tests are producing incorrect output to help you fix your errors and submit again.

Part 3 (Purely optional, no submission):

This part is completely optional. You can work on it if you're ready for a challenge, and to make your spell checker that much more realistic! We will ask you if you have completed this extension in the end of module quiz. That question will not be graded. We're just curious how many of you are completing this part, so be honest.

1. **Make a copy of your AutoCompleteDictionaryTrie.java file and name it AutoCompleteMatchCase.java.** Change the name of the class to AutoCompleteMatchCase, as well. If you do this using the copy and paste interface in the Eclipse package explorer, the class name will be renamed for you. You simply right-click on the AutoCompleteDictionaryTrie class in the package explorer and select copy. Then right click on the spelling package and select paste, and type in the correct name of the new class (AutoCompleteMatchCase). If you make the copy via your file explorer, you will need to manually change the name of the class.

2. Similarly, **make a copy of your DictionaryHashSet class** and name if DictionaryHashSetMatchCase.

3. In each of these new classes, change the implementation so that they account for misspellings due to case. Note that our dictionary is case-sensitive, but it's not as simple as just matching the case in the dictionary exactly. E.g. you must account for capital letters at the start of words. Here are some hints and ideas:

- You probably want to allow the first letter in a non-capitalized word to be allowed to be capitalized, but you should not allow the first letter in a capitalized word to be non-capitalized (e.g. "Hello" should always be OK, but "christine" should not).
- Words in all caps are also OK. E.g. HELLO or CHRISTINE

- The simplest approach might be to add all legal capitalization permutations as separate words in the dictionary, but of course, this makes the dictionary pretty large, so you might want to play around with other ideas. This is up to you!

4. Test your implementation by (a) writing tester classes and (b) modifying the application to use your new classes. You will need to change the LaunchClass to use your new classes where appropriate (getAutoComplete and getDictionary) and you will also need to set the variable matchCase in the AutoSpellingTextArea class to "false".

We're not providing a lot of guidance here because it's just for fun, but feel free to discuss and get ideas with others in the discussion forum.