# Spelling Corrector Project

## Introduction

You are familiar with spell checkers. For most spell checkers, a candidate word is considered to be spelled correctly if it is found in a long list of valid words called a dictionary. Google provides a more powerful spell corrector for validating the keywords we type into the input text box. It not only checks against a dictionary, but, if it doesn't find the keyword in the dictionary, it suggests a most likely replacement. To do this it associates with every word in the dictionary a frequency, the percent of the time that word is expected to appear in a large document. When a word is misspelled (i.e. it is not found in the dictionary) Google suggests a "similar" word ("similar" will be defined later) whose frequency is larger or equal to any other "similar" word.

In this project you will create such a spell corrector. There is one major difference. Our spell checker will only validate a single word rather than each word in a list of words.

For this program we need a dictionary similar to Google's. Our dictionary is generated using a large text file. The text file contains a large number of unsorted words. Every time your program runs it will create the dictionary from this text file. The dictionary will contain every word in the text file and a frequency for that word. Instead of storing a percent, your program need only store the number of times the word appears in the text file.

When storing or looking up a word in the dictionary we want the match to be case insensitive. That is, a new or candidate word matches a word in the dictionary independent of the case of the letters in the respective words. Thus, the word "move" matches "Move". If the strings "apple", "Apple", and "APPLE" each appear once in the input file, then your representation of the word (it could be any of the three words or some other variation) would appear once and would have a frequency of 3 associated with it.

## Trie (Data Structure)

You are required to implement your dictionary as a Trie (pronounced "try"). A Trie is a tree-based data structure designed to store items that are sequences of characters from an alphabet. Each Trie-Node stores a count and a sequence of Nodes, one for each element in the alphabet. Each Trie-Node has a single parent except for the root of the Trie which does not have a parent. A sequence of characters from the alphabet is stored in the Trie as a path in the Trie from the root to the last character of the sequence.

For our Trie we will be storing words (a word is a sequence of alphabetic characters) so the length of the sequence of Nodes in every Node will be 26, one for each letter of the alphabet. For any node ***a*** we will represent the count in ***a*** as ***a***.count. For the array of Nodes in ***a*** we will use ***a***.nodes. For the node in ***a's*** sequence of Nodes associated with the character ***c*** we will use

***a***.nodes[***c***]. For instance, ***a***.nodes['b'] represents the node in ***a***'s array of Nodes corresponding to the character 'b'.

Each node in the root's array of Nodes represents the first letter of a word stored in the Trie. Each of *those* Nodes has an array of Nodes for the second letter of the word, and so on. For example, the word "kick" would be stored as follows:

***root***.nodes['k'].nodes['i'].nodes['c'].nodes['k']

The count in a node represents the number of times a word represented by the path from the root to that node appeared in the text file from which the dictionary was created. Thus, if the word "kick" appeared twice, ***root***.nodes['k'].nodes['i'].nodes['c'].nodes['k'].count = 2.

If the word "kicks" appears at least once in the text file then it would be stored as

> ***root***.nodes['k'].nodes['i'].nodes['c'].nodes['k'].nodes['s']

and ***root***.nodes['k'].nodes['i'].nodes['c'].nodes['k'].nodes['s'].count would be greater than or equal to one.

If the the count value of any node, *n*, is zero then the word represented by the path from the root to *n* did not appear in the original text file. For example, if ***root***.nodes['k'].nodes['i'].count = 0 then the word "ki" does not appear in the original text file. A node may have descendant nodes even if its count is zero. Using the example above, some of the nodes representing "kick" and "kicks" would have counts of 0 (e.g **root**.nodes['k'], **root**.nodes['k'].nodes['i'], and **root**.nodes['k'].nodes['i'].nodes['c']) but **root**.nodes['k'].node['i'].nodes['c'].nodes['k'] and **root**.nodes['k'].node['i'].nodes['c'].nodes['k'].nodes['s'] would have counts greater than 0.

## Word/Node Count Example

Using our example above, with only "kick" and "kicks" in the Trie it would have 2 words and 6 nodes (root, k, i, c, k, s). If we were to add "kicker" the Trie would have 3 words and 8 nodes. If we were to add the word "apple" to the same Trie, it would have 4 words and 13 nodes. Adding the word "ape" would result in 5 words and 14 nodes. Adding "brick" would result in 6 words and 19 nodes.

## Additional Required Methods

Although not strictly required to make your Trie work as a dictionary to solve the spelling corrector problem, your Trie must implement the following three additional methods that are commonly implemented in general purpose Java classes: toString(), hashCode(), and equals(Object).

The toString specification is as follows:

For each word, in alphabetical order:
<word>\n
…

Your hashCode() method should produce hashCode values that satisfy the general contract of hashCode (as specified in the Javadoc for the hashCode method of Object) and are *reasonably* unique. You can achieve this by having a hashcode variable that adds together the hashcodes of each String included into the dictionary. The hashCode() function then returns this variable. **You should not use toString() or similar methods inside your hashcode() method.**

The equals() method has to be thorough! You need to traverse both Tries fully and make sure they are the same. **You should not use toString(), similar string methods, or hashcode() to create your equals() method.**

**Use of other data structures is not allowed in the trie or node classes**. For example, it is common to use a separate data structure, such as a Stack or Queue to traverse the nodes of a tree structure. This is not allowed in your Trie implementation. Equally you may not store chars within each individual node. Instead of relying on these other data structures, both the equals() and the toString() methods must be recursive. The hashcode method should run in constant time.

Your Trie class must implement the ITrie interface provided on the course website and your TrieNode must implement the INode interface.

# Spell Corrector Functionality

You will load all the words found in the provided file into your Trie using Trie.add(String). The user's input string will be compared against the Trie using the Trie.find(String). If the input string (independent of case) is found in the Trie, the program will indicate that it is spelled correctly by returning the final INode of the word. If the case independent version of the input string is not found, your program will return the final INode of the <u>most</u> "similar" word as defined below. If the input string is not found and there is no word "similar" to it, your program should return null.

The following rules define how a word is determined to be <u>most</u> similar:

1. It has the "closest" edit distance from the input string
2. If multiple words have the same edit distance, the most similar word is the one with the closest edit distance that is found the most times in the dictionary
3. If multiple words are the same edit distance and have the same count/frequency, the most similar word is the one that is first alphabetically

An edit distance of 1 will be defined below. If there is more than one word in the dictionary that is an edit distance of 1 from the input string then return the one that appears the greatest number of times in the original text file. If two or more words are an edit distance of 1 from the input string and they both appear the same number of times in the input file, return the word that is alphabetically first. If there is no word at an edit distance of 1 from the input string then the <u>most</u> "similar" word in the dictionary (if it exists) will have an edit distance of 2. A word *w* in the dictionary has an edit distance of 2 from the input string if there exists a string *s* (*s* need not be in the dictionary) such that *s* has an edit distance of 1 from the input string and *w* has an edit distance of 1 from *s*. As with an edit distance of 1, if more than one word has an edit distance of 2 from the input string choose the one that appears most often in the input file. If more than two appear equally often choose the one that is alphabetically first. If there is no word in the dictionary that has an edit distance of 1 or 2 then there is no word in the dictionary “similar” to the input string. In that case return null.

There are 4 measures of edit distance we will use: deletion distance, transposition distance, alteration distance, and insertion distance. A word in the dictionary has an edit distance of 1 from the input string if it has a deletion distance of 1, or a transposition distance of 1, or an alteration distance of one, or an insertion distance or 1.

*The Four Edit Distances:*

- Deletion Distance: A string s has a deletion distance 1 from another string *t* if and only if *t* is equal to *s* with one character removed. The only strings that are a deletion distance of 1 from “bird” are “ird”, “brd”, “bid”, and “bir”. Note that if a string *s* has a deletion distance of 1 from another string *t* then |s| = |t| -1. Also, there are exactly |*t*| strings that are a deletion distance of 1 from *t*. The dictionary may contain 0 to *n* of the strings one deletion distance from *t*.
- Transposition Distance: A string s has a transposition distance 1 from another string *t* if and only if *t* is equal to *s* with two adjacent characters transposed. The only strings that are a transposition Distance of 1 from “house” are “ohuse”, “huose”, “hosue” and “houes”. Note that if a string *s* has a transposition distance of 1 from another string *t* then |s| = |t|. Also, there are exactly |*t*| - 1 strings that are a transposition distance of 1 from *t*. The dictionary may contain 0 to *n* of the strings one transposition distance from *t*.
- Alteration Distance: A string s has an alteration distance 1 from another string *t* if and only if *t* is equal to *s* with exactly one character in s replaced by a lowercase letter that is not equal to the original letter. The only strings that are an alternation distance of 1 from “top” are “aop”, “bop”, …, “zop”, “tap”, “tbp”, …, “tzp”, “toa”, “tob”, …, and “toz”. Note that if a string *s* has an alteration distance of 1 from another string *t* then |s| = |t|. Also, there are exactly 25* |*t*| strings that are an alteration distance of 1 from *t*. The dictionary may contain 0 to *n* of the strings one alteration distance from *t*.
- Insertion Distance: A string s has an insertion distance 1 from another string *t* if and only if *t* has a deletion distance of 1 from *s*. The only strings that are an insertion distance of 1 from “ask” are “aask”, “bask”, “cask”, … “zask”, “aask”, “absk”, “acsk”, … “azsk”, “asak”, “asbk”, “asck”, … “aszk”, “aska”, “askb”, “askc”, … “askz”. Note that if a string *s* has an insertion distance of 1 from another string *t* then |s| = |t|+1. Also, there are exactly 26* (|*t*| + 1) strings that are an insertion distance of 1 from *t*. The dictionary may contain 0 to *n* of the strings one insertion distance from *t*.

# Deliverables

Create classes that correctly implement the `ITrie, INode,` and `ISpellCorrector` interfaces according to this specification. All three classes should have a constructor that takes no arguments. The `ITrie, INode,` and `ISpellCorrector` interfaces are provided on the course web site in the files associated with this project. Remember you must implement all methods defined in these Interfaces. A main class that runs your spelling corrector is also provided. You will need to create an instance of your `ISpellCorrector` implementing class inside the 'main' method of the Main class in the place indicated by the comment in the file. The provided code exists in a 'spell' package. Your code should also be placed in a 'spell' package.

## Example Programs

java spell.Main words.txt bbig
Suggestion is: big

java spell.Main words.txt hello
Suggestion is: hello

java spell.Main words.txt abcxyz
No similar word found