Assessed Coursework 1 — Spell-Checker

Based on Goodrich & Tamassia

1 Overview

In this assignment, you are requested to write a simple spell-checker program. Your program should be named *Spell* and it will take as command line arguments two file names. The first name is the dictionary file which contains correctly spelled words (On Vision the file `dictionary.txt` has been provided). The second file contains the text to be spell-checked. Your program should first read all words from the dictionary file and insert them into a dictionary data structure. Then your program should read words from the second file and check if they are in the dictionary. For words that do exist in the dictionary nothing needs to be done. For words which are not in the dictionary, your program should suggest possible correct spellings by printing to the standard output. You should perform the following modifications of a misspelled word to handle commonly made mistakes:

- *Letter substitution*: go over all the characters in the misspelled word, and try to replace a character by any other character. In this case, if there are k characters in a word, the number of modifications to try is $26k$. For example, in a misspelled word 'lat', substituting 'c' instead of 'l' will produce a word 'cat', which is in the dictionary.
- *Letter omission*: try to omit (in turn, one by one) a single character in the misspelled word and see if the word with omitted character is in the dictionary. In this case, there are k modifications to try where k is the number of characters in the word. For example, if the misspelled word is 'catt', omitting the last character 't' will produce a word 'cat' which is in the dictionary.
- *Letter insertion*: try to insert a letter in the misspelled word. In this case, if the word is k characters long, there are $26 * (k + 1)$ modifications to try, since there are 26 characters to try to insert and $k + 1$ places (including the beginning and the end of the word) to insert a character. For example, for word 'plce', inserting letter 'a' in the middle will produce a correctly spelled word 'place'.
- *Letter reversal*: Try swapping every 2 adjacent characters. For a word of length k, there are $k - 1$ pairs to try to swap. For example in a misspelled word 'paernt', swapping letters 'e' and 'r' will produce a correctly spelled word 'parent'.

For each word which was misspelled, on a separate line, print out the misspelled word and all possible correct spellings that you found in the dictionary. For example, if your dictionary file contains the words 'cats like on of to play', and the file to spell check contains 4 words: 'Catts lik o play', the output should be:

```
catts => cats
lik => like
o => on, to, of
```

Notice that the list of possible correct spelling must contain only unique words. In the example above, for the misspelled word 'catts', removing the first 't' or the second 't' leads to the same word 'cats', but this word appears in the output only once. For the modifications of a word above, the Java class *StringBuffer* and its built-in methods are very useful.

Deadline: 3:30PM on Wednesday 29th of October, 2014

A file `FileWordRead.java` which will read the next word from an opened file is provided. See comments in the file `FileWordRead.java` for the usage. In this implementation, all words are converted to lower case so that the words 'Cat' and 'cat' are treated as one word. Thus all the words you read from the file using the program will be lowercase words.

In this assignment you will implement and compare (experimentally) a linked list based and a hash table based dictionary.

2 Implementation

You are to implement the program based on a dictionary, let's call is D. You will use D for storing the words in the dictionary input file specified by the first command line argument. After storing all the words in D, you should open the second file (the one to be spell-checked) for reading and look up the words in the second file in dictionary D. If any word w of the second file is not in D, you have to try all possible modifications of w suggested above, in Section 1. Notice that different modifications may result in the same word. The implementation of the dictionary, see sections below, insures that each dictionary entry contains a unique word. Therefore, to make sure that there are no duplicates on the list of possible spellings, create a second dictionary S (for each misspelled word), and insert all modifications of the misspelled words that are in dictionary D. After you went over all modifications, print out all the words stored in dictionary S.

3 Classes Provided

3.1 Spell

This is the class which contains the main program, which you mostly have to write yourself. Currently the code reads from the file with names specified by command line arguments. You can uncomment some lines, as explained in file `Spell.java` to read from a file with a specific name (which is more convenient for debugging).

3.2 Dictionary

The interface your dictionary should implement (both the linked list based dictionary and the hash table based dictionary).

3.3 HashCode

This is an interface for the class used to create a hash code for an object.

3.4 TestHashDictionary

This is a program which we will use to test your hash table implementation. Compile and run it once you have implemented your *HashTable* class. It will run some tests on your hash table and will let you know which tests are passed/failed by your hash table. To get the full score on the assignment, you must pass all the tests.

4 Classes to Implement

4.1 DictionaryException

This exception should be thrown by your dictionary in case of unexpected conditions, see sections 4.3 and 4.4 for cases in which to throw this exception.

4.2 StringHashCode

This class implements the *HashCode* interface and should be used to get a hash code for strings. You have to use the polynomial accumulation hash code for strings we talked about in class. It must only have one public method: `public int giveCode(Object key)`. You can implement any other methods that you want, but they must be declared as private methods. You pass an object of this type to the constructor of the hash table, which then assigns it to a private object (let's say its name is `hCode`) of class *HashCode*, say this private object has name `hCode`. The hash table uses the 2 object whenever it needs a hash code: `hCode.giveCode(key)`.

4.3 HashDictionary

This class implements a dictionary based on hash table, and should implement the provided *Dictionary* interface. You should use open addressing with double hashing strategy. Start with an initial hash table of size 7. Increase its size to the next prime number at least twice larger than the current array size (which is N) when the load factor gets larger than the maximum allowed load factor (maximum allowed load factor is to be given to the constructor to the hash table). You must design your hash function so that it produces few collisions.

You must implement the following public methods, and all other methods which you might implement for this class must be private. Any member variables must also be private.

- `public HashDictionary() throws DictionaryException` We don't want the user to use default constructor since the user has to specify the `HashCode`, and the maximum load factor at construction time. Thus this default constructor must simply throw `DictionaryException` if it is ever called.
- `public HashDictionary(HashCode inputCode, float inputLoadFactor)` This is the constructor for the hash table which takes an `HashCode` object and float `inputLoadFactor` which specifies the maximum allowed load factor for the hash table. If the load factor becomes larger than `inputLoadFactor`, the (private) `rehash()` method must be invoked.
- `public void insert(String key)` This method inserts a new entry in the Dictionary.
- `public boolean find(String key)` Returns true if dictionary has the specified key and false otherwise.
- `public Iterator elements()` Returns an *Iterator* over all dictionary entries. The iteration is over objects of class String. You can use `java.util.Iterator` which gives the *Iterator* interface (to use this interface, say `import java.util.Iterator` in the beginning of `HashDictionary.java` file).
- `public void remove(String key) throws DictionaryException` Removes entry with specified key. Throws exception if no such entry exists.

- `public float averNumProbes()` This method returns an average number of probes performed by your hash table so far. You should count the total number of operations performed by the hash table (each of find, insert, remove count as one operation, don't count any other operations) and also the total number of probes performed so far by the hash table. When `averNumProbes()` is called, it should return $\frac{\text{(float) num probes so far}}{\text{num operations so far}}$. As you decrease the maximum allowed load factor, the average number of probes should go down. When you run the `TestHashDictionary` program, it will run your hash table at different load factors and will print out the average probe numbers versus the running time. If you see that the average probe number goes up as the max load factor goes up, you are probably computing probes/implementing hash table correctly. You can implement any other methods that you want, but they must be declared as private methods.

4.4 ListDictionary

This class implements a dictionary based on linked list. You can use Java's build in *LinkedList* class by using `java.util.LinkedList`. This class must implement the provided *Dictionary* interface. You must implement the following public methods, and all the other methods which you might implement for this class must be private. Also any member variables must be private.

- `public ListDictionary()` Constructor for the class
- `public void insert(String key)` This method inserts a new entry in the Dictionary.
- `public boolean find(String key)` Returns true if the dictionary has the specified key and false otherwise.
- `public Iterator elements()` Returns an *Iterator* over all dictionary entries. The iteration is over objects of class *String*. You can use `java.util.Iterator` which gives the *Iterator* interface (to use this interface, say `import java.util.Iterator` in the beginning of `ListDictionary.java` file).
- `public void remove(String key) throws DictionaryException` Removes entry with specified key. Throws exception if no such entry exists.

4.5 Spell

This is the class which contains the main program. You have to write most of the main program, the code in `Spell.java` currently reads the command line arguments (file names). You can rewrite everything in this file, but the name must stay the same, `Spell`. In the version that you hand in, you should be using the hash table dictionary implementation. The linked list based implementation should be used only for comparison with the hash table implementation.

5 Hash Table vs. Linked List Dictionary Implementation

Dictionary files of different sizes (`d1.txt`, ..., `d6.txt`) are provided. Run your program with these different dictionaries and the same `checkText.txt` file to check the spelling of words. That is the second command line argument stays the same, while the first one goes through `d1.txt`, ..., `d6.txt`. Get the running time using the Java method: `System.currentTimeMillis()`. Note that this method will return the current time, **NOT** the running time from the start of the program. Therefore, to get the total time (in milliseconds) your program took to complete, you should measure the current time at the very

start of the program, then at the very end, and subtract the two. Since what changes between the different runs is the size of the dictionary file, we should plot the running time vs. the size of the dictionary file, that is the number of words in the dictionary file. Count the number of words in each dictionary file and plot, on the same chart, the number of words versus the running time for the list and hash based dictionary implementations.

The running time is essentially the time it takes to insert all the dictionary words, since the second file for spell checking has only 2 words to check. When we insert a word into a dictionary, we also have to check if that word is already in the dictionary.

For a hash table, checking and inserting is expected to take a constant amount of time, and therefore inserting all elements in the dictionary should take a linear time. For a linked list, inserting is constant amount of time, but checking if the element is already in the list is linear amount of time, and therefore inserting all elements in the dictionary should take quadratic time. Thus hash based implementation running time plot should resemble a linear function, list based implementation should resemble a quadratic function.

6 Coding Style

Your mark will be based partly on your coding style. Here are some recommendations:

- Variable and method names should be chosen to reflect their purpose in the program.
- Comments, indenting, and whitespaces should be used to improve readability.
- No variable declarations should appear outside methods ("instance variables") unless they contain data which is to be maintained in the object from call to call. In other words, variables which are needed only inside methods, whose value does not have to be remembered until the next method call, should be declared inside those methods.
- All variables declared outside methods ("instance variables") should be declared private (not protected) to maximize information hiding. Any access to the variables should be done with accessor methods (like `key()` and `value()`).

7 Marking and Submission

Your overall mark will be computed as follows.

- Comparison chart of Linked List vs. Hash table running times: 5 marks
- Program compiles, produces a meaningful output: 15 marks
- `HashDictionaryTest` pass: 25 marks (5 tests, each test is worth 5 marks)
- Coding Style: 10 marks
- `HashDictionary` implementation: 15 marks
- Spell program implementation: 15 marks

Submit both a hardcopy of your coursework in the coursework box and an electronic version on Vision. You will be required to demonstrate your program during the lab slot in week 7 (Friday October, 31st). Your coursework is due to be submitted by Wednesday October, 29th, 3.30PM.