Assessed Individual 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 `SpellCheck` 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 map data structure. Then your program should read words from the second file and check if they are in the map. For words that do exist in the map nothing needs to be done. For words which are not in the map, your program should suggest possible correct spellings by printing to the standard output. You should perform modifications of a misspelled word given in Section 2. In this assignment you will implement and compare (experimentally) a linked list based and a hash table based map.

Provisional and Indicative Timing Guidelines

Step 1: from Week 4 (after lecture on Maps)

develop your linked list based map, implement and test some misspelled word modifications

Step 2: from Week 5 (after lecture on Hash Tables)

develop your hash table based map, implement and test remaining misspelled word modifications

Step 3: from Week 6

run tests, run comparison of list-based and map-based implementations, finalise your report

2 Modifications of Misspelled Word

You should perform the 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.
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.

3 Implementation

You are to implement the program based on a map, let us call it M. You will use M for storing the words in the dictionary input file specified by the first command line argument. After storing all the words in M, you should open the second file (the one to be spell-checked) for reading and look up the words in the second file in map M. If any word w of the second file is not in M, you have to try all possible modifications of w suggested above, in Section 2. Notice that different modifications may result in the same word. The implementation of the map, see sections below, insures that each map entry contains a unique word. Therefore, to make sure that there are no duplicates on the list of possible spellings, create a second map S (for each misspelled word), and insert all modifications of the misspelled words that are in map M. After you went over all modifications, print out all the words stored in map S.

4 Classes and Interfaces

`SpellCheck` **(class to complete)**

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

`IMap` **(interface provided)**

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

`MapException` **(class to implement)**

This exception should be thrown by your map in case of unexpected conditions, see classes `HashTableMap` and `LinkedListMap` for cases in which to throw this exception.

`LinkedListMap` **(class to implement)**

This class implements a map based on linked list. You can use Java's build in *LinkedList* class by using `java.util.LinkedList`. This class must implement the provided `IMap` 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 LinkedListMap()`
: Constructor for the class

`public void insert(String key)`
: This method inserts a new entry in the map.

`public boolean find(String key)`
: Returns true if the map has the specified key and false otherwise.

`public void remove(String key) throws MapException`
: Removes entry with specified key. Throws exception if no such entry exists.

`public int numberOfElements()`
: Returns the number of elements stored in the map.

`public Iterator elements()`
: Returns an *Iterator* over all map 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 `LinkedListMap.java` file).

`IHashCode` **(interface provided)**

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

`StringHashCode` **(class to implement)**

This class implements the `IHashCode` 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 us say its name is `hCode`) of class `IHashCode`, say this private object has name `hCode`. The hash table uses the 2 object whenever it needs a hash code: `hCode.giveCode(key)`.

`IHashTableMonitor` **(interface provided)**

This is an interface your hash table based map should implement.

`TestHashTableMap` **(class provided)**

This is a program which we will use to test your hash table implementation. Compile and run it once you have implemented your `HashTableMap` class. It will run some tests on your hash table and will let you know which tests are passed/failed by your hash table. Read the source code of this class to understand what each test if doing and to fix your implementation in case of failed tests. To get the full score on the assignment, you must pass all the tests.

`HashTableMap` **(class to implement)**

This class implements a map based on hash table, and should implement the provided `IMap` interface. It should also implement the `IHashTableMonitor` interface implement the methods needed by `TestHashTableMap`. 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 should implement the following constructors.

`public HashTableMap() throws MapException`

We do not want the user to use default constructor since the user has to specify the `IHashCode`, and the maximum load factor at construction time. Thus this default constructor must simply throw `MapException` if it is ever called.

`public HashTableMap(IHashCode inputCode, float maxLoadFactor)`

This is the constructor for the hash table which takes an `IHashCode` object and float `maxLoadFactor` which specifies the maximum allowed load factor for the hash table. If the load factor becomes larger than `maxLoadFactor`, the (private) `rehash()` method must be invoked.

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 void insert(String key)`

This method inserts a new entry in the map.

`public boolean find(String key)`

Returns true if map has the specified key and false otherwise.

`public void remove(String key) throws MapException`

Removes entry with specified key. Throws exception if no such entry exists.

`public int numberOfElements()`

Returns the number of elements stored in the map.

`public Iterator elements()`

Returns an *Iterator* over all map 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 `HashTableMap.java` file).

`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, do not 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 `(float) numberOfProbes/numberOfOperations`. As you decrease the maximum allowed load factor, the average number of probes should go down. When you run the `TestHashTableMap` 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.

5 Hash Table vs. Linked List Map 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 map, we also have to check if that word is already in the map.
For a hash table, checking and inserting is expected to take a constant amount of time, and therefore inserting all elements in the map 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 map should take quadratic time. Thus hash table based implementation running time plot should resemble a linear function, linked 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 `getKey()`).

7 Submission

Submit a `.zip` or `.tar.gz` archive file electronically on **Vision**. The archive should contain:

- The `.java` source files of your program (do **not** include the compiled `.class` files).

  Submit also the Java files provided (whether you have altered them or not). You should not need to alter the `TestHashTableMap` class or the interfaces provided. If you do so, explain the reason for doing it in your report (see below).

- (Optional but recommended) The tests you have been conducting (other than with the `d*.txt` files provided or with `TestHashTableMap`) to verify your implementation of the Spell-Checker. This could be `.txt` dictionary files and `.txt` input files to check together with the expected output.

- A short report in `.pdf`, `.rtf`, `.odt`, `.doc` or `.docx` format.

  Your report should:

  1. Start with a cover page,
  2. Include a simple specification of the program,
  3. Indicate which IDE environment you have used, and to compile and run the program,
  4. Explain briefly your design choices, if your program meets the specification fully, if your program has known limitations,
  5. Provide and discuss the chart comparison between Linked List and Hash Table implementations.

You will be required to **demonstrate** your program during the lab slot in week 7.

8 Marking Scheme

Your **overall mark** will be computed as follows.

- Program compiles, produces a meaningful output 15 marks
- Coding style 10 marks
- `HashTableMap` implementation 15 marks
- `TestHashTableMap` pass 25 marks
- `SpellCheck` program implementation 15 marks
- Comparison chart of Linked List vs. Hash Table running times 5 marks
- Demonstration of your program during the lab slot in week 7. 15 marks

Your coursework is due to be submitted by 5:30PM on Tuesday 25th of October, 2016. If you hand in work late, without extenuating circumstances, 10% of the maximum available mark will be deducted from the mark awarded for each day late.