Assessed Individual Coursework 1 — Word Index

Learning Outcomes:

- *Ability to analyse and hence choose suitable algorithms and data structures for a given problem*
- *To design and implement medium sized programs based on a range of standard algorithms*
- *Understanding the distinction between Abstract Data Type (ADT) properties and concrete ADT realisations*
- *Appreciation of need for integration of multiple ADTs in substantial programs*
- *Appreciation of efficiencies/reassurances from ADT reuse*
- *To develop practical problem-solving skills in the context of programming*
- *To be able to critically analyse and hence choose suitable algorithms and data structures for a given problem*

1 Overview

In this assignment, you are requested to write a simple word index program. The program will record the position of words in text files in a directory and then offer to retrieve the positions of a given word. The program should be named `WordIndex`.

Your program will read a set of commands from a given file. The program should execute these commands sequentially. Such commands would first request to read all the words of the available text files and insert them into a map data structure. Then, search for a given word, add the words of a new file, remove a file from the list of words, or print a summary the word index. The command file name is given as argument on the command line. The directory containing the text files is called `TextFiles`.

In this assignment you will implement and compare (experimentally) a linked list based and a hash table based map for storing these word positions.

Provisional and Indicative Timing Guidelines

Step 1: Week 4
develop your linked list based map, implement and test the indexing and searching for a word

Step 2: Week 5
develop your hash table based map (you can first implement linear probing collision handling before later changing to a double hashing), implement and test the removing of files from the map, the sorting of the output, and the summary

Step 3: Week 6
finalise your double hashing collision handling, run tests, run comparison of list-based and hash-table-based implementations, finalise your report

2 Implementation

You are to implement the program based on a map, let us call it M. You will use map M for storing the words that appear in the text files with their positions (file names and line numbers).

After storing all the words in M , the program should process the commands for searching for a word, adding a new file, removing an indexed file, display summary of the index. Notice that a word could occur several times in a file, the search result should display first the file where the word occurs the most, and all the lines in that file where it occurs by ascending order.

3 Input and output

Your program will read its commands from one input file given as argument to the program. The input is composed of a sequence of lines where each line is a Word Index command.

A class `WordTxtReader` which will read words one by one from a file and indicate its positions is provided. The class should be used both to read the commands and the words in the text files. See comments in the file `WordTxtReader.java` for the usage. Note that the class will discard any character which is neither a letter of the English alphabet (`a-z` or `A-Z`) nor a number (`0-9`). The program will return a lowercase version of each word.

The commands, their syntax and suggested outputs are as follows:

Adding to the map the words of all the files

```
addall
```

Adds to the map the words and their positions in each file for all the `.txt` files in the `TextFiles` folder. An example of a `TextFiles` directory is given containing the text version of the lectures of the course (`lect*.txt`).

Input: `addall`

Suggested output: `27073 entries have been indexed from 16 files`

Searching for a word

```
search n word
```

Search for a word in the text files that have been indexed. The output should be by order of most occurrence in a file, and should be limited to the number of file given as argument (`n`).

Example input: `search 3 algorithms`

Suggested output:

```
The word "algorithms" occurs 76 times in 7 files:  7 times in lect15.txt
    (lines 21, 86, 94, 95, 233, 278, 402)
  6 times in lect00.txt
    (lines 16, 54, 121, 139, 147, 173)
  5 times in lect02.txt
    (lines 57, 165, 212, 869, 1094)
```

Adding to the map the words of a given file

```
add filename
```

Adds to the map the words and their positions in the given file. Note that the text file needs to be placed in `TextFiles` as the program will only read files from that directory. For example, the command `add example.txt` will read and insert to the map all the words and their positions in the file `TextFiles/example.txt`.

Example input: `add example.txt`

Suggested output: `15 entries have been indexed from file "example.txt"`

Removing in the map the word positions from a given file

`remove filename`

Removes from the map the positions of words from the given file. If a word has no more positions associated with it, it is removed from the map. Note that the file itself should not be removed from the `TextFiles` directory.

Example input: `remove example.txt`

Suggested output: `15 entries have been removed from file "example.txt"`

Overview of the Word Index

`overview`

Prints a summary of the number of indexed words, indexed positions and indexed files.

Suggested output:

```
Overview:
  number of words: 1304
  number of positions: 954
  number of files: 20
```

4 Classes and Interfaces

`WordIndex` **(class to complete)**

This is the class which contains the main program. You have to write most of the main program, the code in WordIndex.java currently reads the command line arguments (file names). The name must stay the same, WordIndex. In the version that you submit, 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.

`IWordMap` **(interface provided)**

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

`WordException` **(class to implement)**

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

`WordTxtReader` **(class provided)**

This class provides a facility for reading words from a file. See section 3 for information.

`IPosition` and `WordPosition` **(interface and class provided)**

These interface and class represent a word position in a file.

`ListWordMap` **(class to implement)**

This class implements a map based on linked list. You can use Java's built-in `LinkedList` class by using `java.util.LinkedList`. This class must implement the provided `IWordMap` 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 ListWordMap()`

Constructor for the class

`public void addPos(String word, IPosition pos)`
This method adds a new position to an entry of the map. It creates the entry if word is not already present in the map.

`public void removeWord(String word) throws WordException`
This method removes from the map the entry for word. Throws exception if word is not present in the map.

`public void removePos(String word, IPosition pos) throws WordException`
This method removes from the map position for word. Throws exception if word is not present in the map or if word is not associated to the given position.

`public Iterator<String> words()`
This method returns an `Iterator`[1]over all words in the map. The iteration is over objects of class `String`.

`public Iterator<IPosition> positions(String word) throws WordException`
This method returns an `Iterator` over all positions of word. The iteration is over objects of class `IPosition`. Throws exception if word is not present in the map.

`public int numberOfEntries()`
This method returns the number of entries stored in the map.

`IHashMonitor` **(interface provided)**
This is an interface your hash table based map should implement.

`HashWordMapProvidedTest` **(class provided)**
This contains unit test cases which we will use to test your hash table implementation. Compile and run it once you have implemented your `HashWordMap` 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.

`HashWordMapProvidedExp` **(class provided)**
This program will run some experiments to display the number of probes depending on the max load factor of your hash table based map implementation.

`HashWordMap` **(class to implement)**
This class implements a map based on hash table, and should implement the provided IWordMap interface. It should also implement the `IHashMonitor` interface and implement the methods needed by `HashWordMapProvidedTest`.

You should use open addressing with double hashing strategy. Start with an initial hash table of size 13. Increase its size to the next prime number at least twice larger than the current array size 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 HashWordMap()`
A constructor for the class which sets the maximum load factor to 0.5.

`public HashWordMap(float maxLoadFactor)`
A constructor for the class which allows to set the maximum load factor at construction time.

[1] You can use `java.util.Iterator` which gives the `Iterator` interface (to use this interface, say `import java.util.Iterator` in the beginning of the file).

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 addPos(String word, IPosition pos)`
: This method adds a new position to an entry of the map. It creates the entry if word is not already present in the map.

`public void removeWord(String word) throws WordException`
: This method removes from the map the entry for word. Throws exception if word is not present in the map.

`public void removePos(String word, IPosition pos) throws WordException`
: This method removes from the map position for word. Throws exception if word is not present in the map or if word is not associated to the given position.

`public Iterator<String> words()`
: This method returns an Iterator over all words in the map. The iteration1 is over objects of class `String`.

`public Iterator<IPosition> positions(String word) throws WordException`
: This method returns an `Iterator` over all positions of word. The iteration is over objects of class `IPosition`. Throws exception if word is not present in the map.

`public int numberOfEntries()`
: This method returns the number of entries stored in the map.

`public float getMaxLoadFactor()`
: This method returns the maximum authorised load factor.

`public float getLoadFactor()`
: This method returns the current load factor.

`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 `HashWordMapProvidedTest` 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.

`public int hashCode(String s)`
: This method returns the hash code as an integer of a given string. You have to use the polynomial accumulation hash code for strings we talked about in class. Note that you can implement any auxiliary methods that you need to help you implement this method, but they must be declared as private methods. You should not use Java's `.hashCode` method.

`ListWordMapTest` and `HashWordMapTest` **(classes to implement)**

These classes should contain jUnit 4 test cases[2] that you used to test your `ListWordMap` class and `HashWordMap` class implementations. Tests that are focusing on the map methods can appear in both testing classes. Cases in `HashWordMapTest` could also test the behaviour of your hash table map with regards to the maximum and current load factors. You could combine the following strategies to write your test cases:

- Write one test per branch of your program (testing that each if-branches or each loop-condition behaves as you expect), this is to ensure your test cases to cover most cases.
- Write tests for borderline cases like when trying to remove an entry from an empty map, or adding an empty word.
- When you notice that your program had bug, write a test case to test that you have properly fixed the problem. This will help you not repeat the bug later (this is called a *non-regression* test).

5 Hash Table vs. Linked List Map Implementation

Text files containing Shakespeare's work (from Project Gutenberg[3]) of different sizes (`01-MND.txt`, ..., `06-KR3-MND.txt`) are provided in a second `TextFiles Shakespeare` directory. Run your program with this directory using a `command.txt` file argument containing the commands `addall` and `search 3 Hermia` to check your Word Index programme.

Get the running time using after each file's words/positions are added to to the map with 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 text file, we should plot the running time vs. the size of the text file, that is the number of words in the text file. Count the number of words in each text file and plot, on the same chart, the number of words versus the running time for the list and for the hash based map implementations.

The running time is essentially the time it takes to insert all the dictionary words, since the second command is only for searching the word Hermia. 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 can be made 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, and linked list based implementation should resemble a quadratic function.

[2] See `HashWordMapProvidedTest` class for examples of jUnit 4 test cases.

[3] http://www.gutenberg.org/

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 maximise information hiding. Any access to the variables should be done with accessor methods (like `getVar()` and `setVar(...)` for a private variable `var`).
- Use appropriate stream when printing output: normal output should be on the standard output (using `System.out`). Error or warning notifications should be on the standard error output (using `System.err`).

7 Note on plagiarism and collusion

- The coursework is an **individual** coursework.

- You are permitted to **discuss** the coursework with your with your classmates. You can **get help** from lecturer and lab helpers in lab sessions. You can get help and **ask questions** to lecturer, via GitLab-Student or by email or at the beginning or end of lecture sessions, or during the office hour of the lecturer.

- Coursework reports must be written in your **own words** and any code in their coursework must be your **own code**. If some text or code in the coursework has been taken from other sources, these sources must be **properly referenced**. Failure to reference work that has been obtained from other sources or to copy the words and/or code of another student is **plagiarism** and if detected, this will be reported to the School's Discipline Committee. If a student is found guilty of plagiarism, the penalty could involve voiding the course.

- Students must **never** give hard or soft copies of their coursework reports or code to another student. Students must always **refuse** any request from another student for a copy of their report and/or code.

- Sharing a coursework report and/or code with another student is **collusion**, and if detected, this will be reported to the School's Discipline Committee. If found guilty of collusion, the penalty could involve voiding the course.

- **Special note for re-using available code:** If you are re-using code that you have not yourself written, then this must clearly be indicated. At all time, you have to make clear what part is not yours, and what in fact is your contribution. Re-using existing code and amending it is perfectly fine, as long as you are not trying to pass it on as your own work, so you must clearly state where it is taken from. A brief additional explanation of why you chose this code would be an added benefit and even adds value to your work. If your code is found elsewhere by the person marking your work, and you have not mentioned this, you may find yourself having to go before a disciplinary committee and face grave consequences.

8 Submission

Submit an archive file (e.g., a .zip or .tar.gz) electronically on Canvas.

- Follow the subdirectory of the template files provided on GitLab-Student. This should include all[4] the .java source files of your program as well as your jUnit 4 test cases *Test.java. Do not include the compiled .class files.

  You should fork the F28DA-2021-2022-CW1 on GitLab-Student and regularly commit your work. You should **not** invite any other students to your project nor share your code with other students. Remember that the final submission is through Canvas.

- A short report in .pdf, .rtf, .odt, .doc or .docx format only. Your report should:
  1. Explain briefly your design choices, if your program meets the specification fully, if your program has known limitations,
  2. Showcase examples of outputs your program produces for given inputs (create your own examples similar to the ones given in Section 3),
  3. Provide and discuss the chart comparison between Linked List and Hash Table implementations.

Your coursework is due to be submitted by Monday, 28 February 2022, at 15:30.
The course applies the University's coursework policy:

- No individual extension for coursework submissions.
- Deduction of 30% from the mark awarded for up to 5 working days late submission.
- Submission more than 5 working days late will not get a mark.
- If you have mitigating circumstances for an extension, talk to your Personal Tutor and submit a Mitigating Circumstances (MC) form with supporting documentation to the School Office.

You will be required to take part in **peer-testing** after submission. You will be using your implementation and test cases. At the end of the peer-testing period you will be asked to submit a short **reflective summary** on Canvas[5].

9 Marking Scheme

- Coding style, program compiles, produces a meaningful output 20 marks
- WordIndex, ListWordMap, HashWordMap implementations 20 marks
- HashWordMapProvidedTest pass 20 marks
- Report including comparison chart of Linked List vs. Hash Table running times 20 marks
- Test cases submitted, peer-testing involvement and reflective summary 20 marks

[4] Submit also the Java files provided (whether you have altered them or not). You should not modify the interfaces provided. If you consider you need to add or modify some method signatures in the interfaces, please speak first to the lecturer of the course.

[5] In some circumstances, a demonstration of your program could be organised, this needs to be approved or requested by the lecturer of the course.