Assignment 2

30 Marks

Due: Monday, 15 October 2018 at 6:00pm

Introduction

Searching is a common process in many applications, with textual searches being a very common type of search. Search engines, query engines and many data mining applications are examples of textual search applications. These applications make use of sophisticated algorithms and data structures to provide efficient search behaviour.

In this assignment you will implement a general textual search application. The application is to be optimised to perform many complex searches on a single document or collection of documents.

Details

You are provided with three files representing the document to be searched, an index of sections in that document and a set of stop words. The document to be searched (`shakespeare.txt`) is a large text file containing the complete works of William Shakespeare. The index of sections (`shakespeare-index.txt`) contains the titles of the works included in the document and the line number on which they start. The set of stop words (`stop-words.txt`) contains words that are to be ignored in a logical search. All three files are encoded in UTF-8 format.

When searches are performed on the document, the results need to indicate the position at which the search term(s) appear in the document. The position is indicated by the line number on which the term appears and the column number of the first character of the term. Searches will only be for English words and will ignore the case of the words in either the document or search term. (e.g. If the document contained the text “Well, come again tomorrow.”, a search for the phrase “well come again tomorrow” would match it.)

When determining the text that makes up a single word in the document, most punctuation marks can be ignored. The exception is for an apostrophe in the middle of a word, such as for a possessive noun or a contraction. (e.g. “I’ll” would be stored as the word “i’ll”, and “honour’s” as “honour’s”.) Apostrophes, or other punctuation, at the start or end of a word can be ignored. (e.g. “'tis” would be stored as the word “tis”, and “years'” as “years”.)

The file `shakespeare-index.txt` contains the titles and line numbers of each of the works in the shakespeare.txt file. The titles and line numbers are separated by a comma. Your application needs to use this index file to determine if a search term appears in a particular work of Shakespeare. For example, searching for “to be or not to be” in “the tragedy of hamlet”, should return that it is on line 25,779 and column 1. Whereas, searching for “to be or not to be” in “the tragedy of romeo and juliet”, should return that no match was found. The titles should **not** have punctuation removed and the full title, **exactly** as it is in the index file, is needed when identifying a section in a search.

The `stop-words.txt` file contains a list of words that are used too commonly in English text to be useful as parts of search terms. This file contains one stop word per line in the file. Your application needs to store all of the stop words found in the document, as there may be situations when a search needs to include them. But, most searches you implement should ignore any of the stop words that appear in the search term. The only search that will not ignore the stop words is the search for a `phraseOccurrence`. For example, the famous phrase “to be or not to be” in *Hamlet* would not be able to be found if you ignored the stop words, as almost the entire phrase consists of stop words.

Your application needs to store the data in appropriate data structures allowing efficient searching. Part of your mark will be based on the data structures you use and your justification of your choices. Your application should be designed with the assumption that the document will be searched many times and that searching needs to return a result as fast as possible. You should process the document's text to optimise search performance. Searching is expected to occur in main memory.

Your application should be designed so that it will work with any text document that has an associated index in the format described above. The application should also work with any list of stop words. Testing of your application may use a different text document and index file, or a different set of stop words.

Your application needs to implement the following searches:

1. Count the number of occurrences of a word.
2. Search for a single word.
3. Search for a prefix of a word. (e.g. Searching for "obscure" would find "obscure", "obscured", "obscures" and "obscurely".
4. Search for a phrase, where a phrase is a sequence of words that are contiguous (e.g. "to be or not to be").
5. Implement boolean search logic for the operators AND, OR and NOT.

There are two parts to the fifth search. One part is to find all lines which match the search criteria. The second part is to find all occurrences of the terms in the search criteria within one or more sections of the document.

So that you do not need to implement parsing of search strings, you are provided with an interface called `Search` and a stub class `AutoTester` that implements it. These will be used to automate the testing of your application's functionality. Your `AutoTester` class will need to implement the constructor stub and override the interface's methods. These methods should call the related functionality in your assignment and return the results defined in the `Search` interface's JavaDoc. If you do not implement some searches, the methods related to these should not be implemented in your `AutoTester` class. You are provided with two simple entity classes called `Pair` and `Triple` that are used by `Search` to return results from various methods.

You may implement a user interface for the assignment if you would like to do so. This will not be executed by markers and will not be considered in assessing your mark for the assignment.

Hints

The intention of the assignment is to determine not just your ability to implement sophisticated data structures and their related algorithms. It is to determine your ability to select data structures and algorithms that are suitable for complex tasks. This may involve using a combination of data structures and algorithms to provide a good solution.

Section 13.3.4 in the textbook describes an approach to implementing a search engine. You may find it helpful in providing ideas for this assignment.

Constraints

You may not move any provided classes or interfaces to different packages. You may not add, remove or change anything in the `Search` interface. You may not add, remove or change anything in `Pair` or `Triple`. You may not move the text files out of the `files` directory and you may not change the names of these files.

You may **not** use anything from the Java Collections Framework in the JDK, or any other Java data structures library, to implement the Abstract Data Types (ADTs) or Concrete Data Types (CDTs) in your assignment. You may implement the Comparable, Comparator, Iterator or Iterable interfaces for use in your ADTs or CDTs. Your implementation of the data structures in the assignment must be based on basic Java language constructs and not libraries. See:

- https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

The restriction of not using anything from the Java Collections Framework (JCF) does not apply to your implementation of `AutoTester`. Some of its methods return results that are types from the JCF. Your implementation of these methods will need to put the results found from your data structure searches into an appropriate JCF concrete type.

Analysis

In the JavaDoc comments for your CDT classes indicate the run-time efficiency of each public method. Use big-O notation to indicate efficiency (e.g. O(n log n)). This should be on a separate line of the method's JavaDoc comment, as the last line of the description and before any `@param` or `@return` tags. When determining run-time efficiency of a method, take into account any methods that it may call.

In the JavaDoc comments for your CDT classes indicate the memory usage efficiency of the data structure. This should be in the JavaDoc comment for the class and should be on a separate line, which is the last line of the description and before any `@author`, `@see` or any other tags.

You must provide an explanation as to how you determined the run-time and memory usage efficiencies.

In a file called **`readme.pdf`**, provide a justification for the algorithms and data structures that you used in implementing the application. The justification should explain why the algorithms and data structures that you implemented were appropriate in the context of efficiently performing many different searches on the same document in memory.

COMP7505 Extension

Students who are studying COMP7505 must complete a further research component to this assignment. This extension is worth an additional 10 marks. Your final mark displayed on BlackBoard's Grade Centre will be scaled to be out of 30. If a student studying COMP3506 completes this extension it will not be marked.

Pre-processing the text in the document to optimise searching is an expensive process. You are to research options of how to minimise this cost. For example, can the results of the pre-processing be saved to a file that can then be loaded directly into the data structures used for searching? You are to research possible approaches and write a report describing one of these approaches. The description should briefly summarise why the approach was an appropriate selection in comparison to other approaches. Your report should analyse the run-time efficiency of the selected approach and compare this to the run-time efficiency of pre-processing the document. You should draw a conclusion as to whether the selected approach is a better alternative than pre-processing the document or not.

You may attempt to implement the approach in your application. If you do so, the `AutoTester` constructor should check the `documentFileName` to determine if it has been previously pre-processed. If it has been pre-processed then the constructor should apply the optimisation strategy to create the application's data structures. If the file has not been previously pre-processed, then it

should be pre-processed as it is loaded and then the optimisation strategy should be able to be applied to that file in future uses of `AutoTester`.

You may use Java's serialisation functionality for the implementation. Just making your data structures implement the `Serializable` interface will gain minimal marks for this functionality. If you use Java' serialisation functionality, your report must provide a detailed description of its implementation, not just say that the objects' state is saved to the file system.

The report must be less than 1000 words, not including any references. Your report should follow the IEEE standard for in-text citations and bibliographic referencing. Failure to do this may lead to allegations of plagiarism.

This report is to be in a file called **`report.pdf`**. Failure to submit this report will cap your final mark for this assignment to be a **maximum** of 13 out of 30. Failure to implement the approach described in your report, will result in **only** losing the marks specified in the criteria for this functionality.

Submission

You must submit your completed assignment electronically through Blackboard. You should submit your assignment as a single zip file called **`assign2.zip`** (use this name – all lower case). This zip file **must** contain the code that was provided for the assignment in the same directory structure as was provided in the **`provided_code.zip`**. You may put your own implementation code in whatever packages you believe are appropriate, as long as `AutoTester` can be executed in its existing package. The **`readme.pdf`** file, and **`report.pdf`** file if you are a COMP7505 student, must be in the root directory of **`assign2.zip`**. Submissions that **do not** conform to these requirements (e.g. different directory structures, packages or classes) **may not** be marked.

You may submit your assignment multiple times before the deadline – only the **last** submission will be marked.

Late submission of the assignment will **not** be marked. In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension.

See the course profile for details of how to apply for an extension:
http://www.courses.uq.edu.au/student_section_loader.php?section=5&profileId=94208

Requests for extensions **must** be made no later than 48 hours prior to the submission deadline. The application and supporting documentation (e.g. medical certificate) **must** be submitted via my.UQ. You **must** retain the original documentation for a minimum period of six months to provide as verification should you be requested to do so.

Please read the section in the course profile about plagiarism. Submitted assignments will be electronically checked for potential plagiarism.

Completeness

The marking criteria below are based on the assumption that the entire assignment is completed. A partially complete assignment will be accepted and marked but the final mark for the assignment will be scaled based on the amount of work completed. For example, completing only search 1 would result in your final mark being reduced by 50%. Completing only searches 1 and 2 would result in your final mark being reduced by 40%.

Marking Criteria

Functionality Marks

Each of the functional points below is worth ⅓ of a mark. There is a bonus of 1 mark for passing at least 26 of these tests, and another bonus of 1 mark for passing at least 28 of these tests.

- `wordCount` correctly counts the occurrences of a single word that has no edge cases (e.g. no punctuation adjacent to the word or word starting or ending a line).
- `wordCount` correctly counts the occurrences of a single word.
- `phraseOccurrence` correctly locates all occurrences of a single word that has no edge cases (e.g. no punctuation adjacent to the word or word starting or ending a line).
- `phraseOccurrence` correctly locates all occurrences of a single word.
- `phraseOccurrence` correctly locates all occurrences of a phrase that has no edge cases.
- `phraseOccurrence` correctly locates all occurrences of a phrase that occurs on a single line.
- `phraseOccurrence` correctly locates all occurrences of a phrase.
- `prefixOccurrence` correctly locates all prefixes of the word, except the word itself.
- `prefixOccurrence` correctly locates all prefixes of the word.
- `wordsOnLine` correctly finds lines with more than one of the words that have no edge cases (e.g. no punctuation adjacent to or within the phrase).
- `wordsOnLine` correctly finds lines with all the words that have no edge cases.
- `wordsOnLine` correctly finds lines with all the words.
- `someWordsOnLine` correctly finds lines with at least one of the words that have no edge cases.
- `someWordsOnLine` correctly finds lines with some of the words that have no edge cases.
- `someWordsOnLine` correctly implements “or” logic.
- `wordsNotOnLine` correctly finds lines with at least one of the required words and at least not one of the excluded words that have no edge cases.
- `wordsNotOnLine` correctly finds lines with at least all the required words and none of the excluded words that have no edge cases.
- `wordsNotOnLine` correctly implements “and/not” logic.
- `simpleAndSearch` correctly finds position of all words, in one section, that have no edge cases.
- `simpleAndSearch` correctly finds position of all words, in multiple sections, that have no edge cases.
- `simpleAndSearch` correctly finds position of all words, in multiple sections.
- `simpleOrSearch` correctly finds position of words, in one section, that have no edge cases.
- `simpleOrSearch` correctly finds position of words, in multiple sections, that have no edge cases.
- `simpleOrSearch` correctly finds position of words, in multiple sections.
- `simpleNotSearch` correctly finds position of all required words, in one section that does not have any of the excluded words; where the words have no edge cases.
- `simpleNotSearch` correctly finds position of all required words, in multiple sections that do not have any of the excluded words; where the words have no edge cases.
- `simpleNotSearch` correctly finds position of all required words, in multiple sections that do not have any of the excluded words.
- `compoundAndOrSearch` correctly finds position of all required words, in one section that has at least one of the `orWords`; where the words have no edge cases.
- `compoundAndOrSearch` correctly finds position of all required words, in multiple sections that have at least one of the `orWords`; where the words have no edge cases.
- `compoundAndOrSearch` correctly finds position of all required words, in multiple sections that have at least one of the `orWords`.

Criteria	Excellent	Good	Satisfactory	Poor
Code Quality	▪ Code is well structured with excellent readability and clearly conforms to a coding standard. ▪ Comments are clear, concise and comprehensive, enhancing comprehension of method usage and implementation details.	▪ Code is well structured with good readability and conforms to a coding standard. ▪ Comments are clear and comprehensive, mostly enhancing comprehension of method usage and implementation details.	▪ Code is well structured with good readability and mostly conforms to a coding standard. ▪ Comments are fairly clear and comprehensive, providing some useful information about method usage or implementation details.	▪ Parts of the code are not well structured or are difficult to read, not conforming to a coding standard. ▪ Comments are often not clear or comprehensive; or provide little useful information about method usage or implementation details.
	4	3.5 3	2.5 2	1.5 1 0.5 0
Design	▪ Data structures and algorithms are very good choices in terms of efficiency and are implemented in an exemplary manner. ▪ Justification of design is valid, clear, concise, articulate and well informed.	▪ Data structures and algorithms are good choices in terms of efficiency and are implemented with very few inefficiencies or minor errors. ▪ Justification of design is valid, clear and well informed.	▪ Data structures and algorithms are appropriate choices in terms of efficiency and are implemented with few inefficiencies or minor errors. ▪ Justification of design is valid and clear.	▪ Some data structures and algorithms are inappropriate choices in terms of efficiency or are implemented with several inefficiencies or some errors. ▪ Justification of design is invalid or unclear.
	10 9	8	7 6 5	4 3 2 1 0
Analysis	▪ Correctly determined and clearly explained run-time and space efficiency of every method and CDT.	▪ Correctly determined and clearly explained run-time and space efficiency of almost all methods and all CDTs.	▪ Correctly determined and fairly clearly explained run-time and space efficiency of most methods and CDTs.	▪ Correctly determined run-time and space efficiency of some methods and CDTs; or poorly explained how efficiencies were determined.
	4	3.5 3	2.5 2	1.5 1 0.5 0
COMP7505 Report	▪ Correct, clear and concise description of approach. ▪ Valid, clear, concise and well-reasoned summary of alternatives. ▪ Correct run-time analysis of approach with an insightful comparison to pre-processing leading to an informed conclusion.	▪ Correct and clear description of approach. ▪ Valid and well-reasoned summary of alternatives. ▪ Correct run-time analysis of approach with an insightful comparison to pre-processing.	▪ Correct and reasonably clear description of approach. ▪ Valid and moderately well-reasoned summary of alternatives. ▪ Correct run-time analysis of approach with a fairly clear and valid comparison to pre-processing.	▪ Incorrect or unclear description of approach. ▪ Invalid or poorly reasoned summary of alternatives. ▪ Incorrect run-time analysis of approach, or an unclear or invalid comparison to pre-processing.
	8 7	6	5 4	3 2 1 0
COMP7505 Implemented	▪ Implementation of approach works and is well designed.	▪ Implementation of approach works.	▪ Implementation of approach works but is awkward or inefficient.	▪ Implementation relies solely on Serializable interface, or does not work, or was not attempted.
	2	1.5	1	0.5 0
Total:				