Assignment Justification

Data Structures

Since the data of this assignment has a fixed size in term of the number of words. Moreover, the Application should have to store all the data to perform search tasks, so that I decided to use ArrayList and HashTable as the core Data Structures to maximize the performance of the application by making use of the fact that they have the constant time complexity for most of the frequently used operations such as Access Item (into an ArrayList), Add Item (at the end of an ArrayList), Insert a Key (into a HashTable) and Search for a Key (in a HashTable) [1].

On the other hand, the main purpose of this assignment tasks are to find the position of words and phrase, so I decided to use a Standard Trie [1, pp. 586] to store all the position of every word. Moreover, the entire document consists of multiple sessions so I created a data structure to store all the content of each section so that I can easily search for words or phrases in a particular section using this data structure.

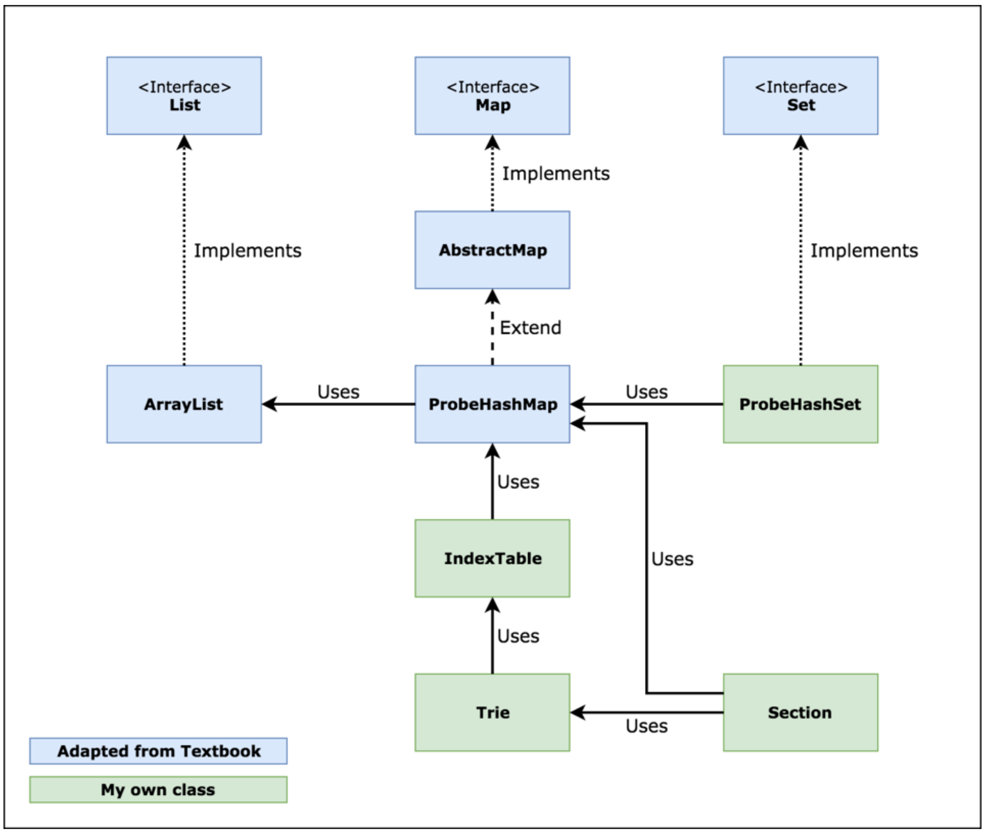

Figure 1. Data Structure Diagram

Each section contains a Map that stores all the lines, which the key is the line number and the value is the line text. Moreover, In order to enhance the search speed, I created an IndexTable for each word in a Trie which can be considered as an occurrence list of that word. The IndexTable is basically a MultiMap of line numbers and a List of columns that represents the word's position. This idea is based on the Inverted Index File concept which is mentioned in the textbook [1, pp. 594]

In order to enhance the performance, we could use the Compressed Trie instead of the Standard one because the compressed tried can reduce the redundant nodes in the Trie. One other potential improvement method is using the text-compress technique (e.g Huffman's Algorithms) to reduce the size of the dataset before storing them in the concrete data structure.

Justifications

By using those Data Structures, I can get the all the positions of a word with the time complexity of ***O(bm)*** with **m** is the length of the word and **b** is the average branch factor of the Trie (in the worse case, **b** is equal to the size of the alphabet)

Firstly, the Section Index is loaded to a List, I used ArrayList because it reserves the ordering of the Sections while reading from the file, so I don't have to sort it again. Then I added all the text of line into a Map which maps line numbers to the text of that line, I also add the position of each word into an IndexTable of the associated section. So that I can search on it later.

For the WordCount method, I only have to return the size of the index tables which is very efficient with the time complexity of **O(1)**

For the PhraseOccurrence method, since I store all the text in my data structure, so I only have to search for the first word of the phrase and then loop through the phrase characters to compare texts. by using this method, it runs with the complexity of **O(mn**) with m is the number of characters in phrase and n is the number of first word occurrences.

For the PrefixOccurence method, I use the recursive method to find all the descendants of a node in the Trie, so that I can get all the Index Table of the words that match the prefix with the time complexity of **O(nd)** with **n** is the number of characters in the word and **d** is the size of alphabet, which can be both considered as constants. So the time complexity can be considered as constant.

For the Search Methods that include basic logic operations (AND, OR, NOT) I decided to use the Set data structure incorporated with set operations such as addAll, removeAll and retainAll [1, pp. 445] to manipulate the compound result of multiple words and logic oprations. By using Sets, I can get the result with the complexity of ***O(swn)*** with **s** is the number of sections that we are searching on, **w** is the number of words in the query**,** and **n** is the number of occurrences. Since the number of sections and number of words in the query can be considered as constants, so the time complexity can be considered as linear.

References

1. M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Data structures and algorithms in Java. John Wiley & Sons, 2014