**CS2010 Algorithms and Data Structures Assignment 3 Report**

**Overview of Program**

The AnagramFinder class is tasked with creating a dictionary of words from a text document where words are listed with their anagrams. In order to find a word in the dictionary, its base form must be known. The base form of a word is a sorted version of its letters, such that words made up of the same letters (i.e. anagrams) will have the same base forms. Each word is stored in the dictionary under its base form.

The aim of this report is to compare implementations of this dictionary using two different data structures – a binary search tree and a hashtable. In order to compare these data structures, the average overall runtime and the average method runtimes will be compared. In order to fully understand which data structure is most appropriate in different situations, the runtimes will be compared for different sizes of input dictionaries – one of approximately 110 words, one of 10,000 words and one of 25,000 words.

**AnagramFinder Class**

The AnagramFinder class contains the main methods of the program. The *createDictionary()* function simply adds each word from a passed String array to the data structure using the related class’s *add()* method. The *getBaseFormOf()* method takes in a String and converts it to its “base form”. The passed String is converted to lowercase letters then sorted into alphabetical order by converting it to a char array and calling *Arrays.sort(char[] a). Arrays.sort(char[] a)* uses a “Dual-Pivot Quicksort”[[1]](#footnote-1) to sort the passed array in ascending numerical order – in this case, alphabetically. The sorted array is then converted back to a String and returned by the method.

**BSTAnagram Class**

The BSTAnagram class is an extension of the AnagramFinder class with its own *add()* and *search()* methods which override those methods in the superclass. The class utilises the TreeMap data structure found in *java.util* to implement a binary search tree.

The *add()* method takes a String and adds it to the binary search tree. To do this, the passed String is converted to its base form using the superclass’s *getBaseFormOf()* method. If that base form already exists as a key of the tree, the word is added to the String array of values at that key. If the base form does not already exist, the word is added to an array and then added to the table at that key.

The *search()* method takes a String and searches for its associated values in the binary search tree. The String is again converted to its base form using *getBaseFormOf().* If the tree contains that base form as a key, its related value array is returned. If not, the passed word is not in the dictionary and the function returns an empty array.

**HTAnagram Class**

The HTAnagram class is an extension of the AnagramFinder class with its own *add()* and *search()* methods which override those methods in the superclass. The class utilises the Hashtable data structure found in *java.util* to implement a hashtable.

The HTAnagram *add()* and *search()* methods are identical to those of the BSTAnagram class.

**Optimisation**

Several steps were taken to optimise the solution:

1. In the *getBaseFormOf()* method, all words are converted to lowercase before being added to the dictionary. This ensures that comparison between base forms are correct and avoids creating extra entries for words that contain the same letters but may have some in uppercase.

1. The data structures use arrays rather than arraylists to store the values of each key. Although each array must be copied and resized when adding a new word to the value whereas an arraylist can be dynamically resized without copying, when the implementations were compared the methods using arrays were found to be faster.

**Efficiency**

To measure runtime, the *System.nanoTime()* function was used to record the start and end time of each test in the AnagramFinderTest class. Each test was run twenty-five times and an average runtime calculated from the results of those tests.

First I compared the average overall runtimes of all program functions on different sizes of dictionaries:

|  |  |  |
| --- | --- | --- |
|  | BSTAnagram *(nanoseconds)* | HTAnagram *(nanoseconds)* |
| small\_dictionary.txt | 349,792 | 440,042 |
| medium\_dictionary.txt | 52,480,888 | 21,356,794 |
| Dictionary\_25K.txt | 82,601,795 | 54,145,681 |

Table 1: Average program runtimes for different sized dictionaries

These runtimes show that the HTAnagram class is a much more efficient implementation overall than the BSTAnagram class. While the two implementations are close for the small dictionary, the larger the dictionary gets the more efficient the hashtable gets in comparison to the binary search tree.

The runtime of the individual functions *add()* and *search()* functions for each data structure were then investigated. Again, twenty-five iterations of each function were used to generate an average:

|  |  |  |
| --- | --- | --- |
|  | BSTAnagram *(nanoseconds)* | HTAnagram *(nanoseconds)* |
| add | 1,786 | 2,002 |
| search | 8,007 | 2,651 |

Table 2: Average method runtimes

This showed that while the BSTAnagram *add()* method was slightly faster than that of the HTAnagram class, the HTAnagram class’s *search()* method was, on average, four times faster than that of BSTAnagram.

**Conclusion**

This report found that, for large dictionaries of word, a hashtable implementation was considerably faster than a binary search tree for structuring and searching through an anagram dictionary.

1. https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#sort(char[]) [↑](#footnote-ref-1)