Margaret Nduta Njenga
MMU ID – 24846577

1. INTRODUCTION

With the ever-growing digital world, various fields seek to extract meaning and detect patterns in text. For instance, A marketing analyst wants to analyse reviews and feedback from customers to best meet their needs, or in sentiment analysis, where a therapist seeks to detect whether individuals are depressed or not based on online texts.

This application is a simple text analysis tool implemented in C# in Microsoft Visual Studio to address such cases. Given some text as input, it performs various tasks, including:

- Displaying the number of unique words.
- Displaying all the words in the text and their frequencies.
- Displaying all the words in the text file and their frequencies in ascending order.
- Displaying the longest word and its frequency.
- Displaying the most frequent word and its frequency.
- Displaying the frequency of a specific word.

The tool implements a linked list, particularly a singly linked list, a dynamic data structure, to efficiently store and manipulate words in memory.

2. DATA STRUCTURE

This tool uses the singly linked list, which comprises nodes containing two parts: the data and a reference to the next node.

Why a linked list?

Linked lists are dynamic data structures that can grow and shrink as the program runs. They are implemented by a class node that stores a word and a reference to the next word. A second class, Linked List, then manipulates the entire list of words.

In contrast, in static data structures such as arrays, a word would be stored in a fixed-size array. New words must be added at the end of the list, which is not scalable when working with a large dataset. To get their count, a parallel array needs to be created. With this approach, there could be duplicates and slower performance, particularly when performing tasks such as sorting.

Alternatively, in dictionaries, words would be stored as keys, and keys must be unique. Their corresponding counts would be stored as values. This would provide fast access to values based on their keys using hashing. For instance, the number of unique words can be determined by the number of keys, and displaying all the words with their counts would be as simple as printing the dictionary. However, they may consume more memory due to hashing, and sorting would be complex.

3. DESIGN AND IMPLEMENTATION

The application has three classes.

- The Node Class is the application's blueprint and defines each element's structure in the linked list.
- The Linked List Class that contains the word and all the methods to manipulate the linked list.
- The main program is where the file is read and refers to the linked list to output any task required in the console application.

Node Class
The class contains the Data(word), the Count, which tracks the frequency of words in every new node, and Next, which refers to the next node in the list. This node class is where the data lives and is used throughout the Linked List class to store and access words.

Linked List Class
A linked list class is created once a node class is defined, and methods will be declared to perform the various tasks. This class starts with an empty list, initialized with a head of the list that points to null, but this changes as new words are added. One of the methods, "AddUnique", is used to add new words. If a word being added already exists in the list, no new node is created; instead, the word count is incremented. If the word does not exist in the list, a new node is created and inserted at the front of the list, updating the head of the list to this new node. This ensures that all words being added are unique. As the unique words are being added, their count is also updated.

A temporary pointer is used to traverse the list to avoid overwriting the head. The nodes are counted to find the total number of unique words in the list, and the method is called in the main program to display the number of unique words.

Since the number of occurrences for each word is tracked and updated during insertion using Count, the frequency of each word can be displayed by accessing the Data and Count fields of each node.

The default way the linked list stores the words is not in any specific order. To order the list, say ascending alphabetical order, the concept of a collection of lists can be utilized, whereby a new list will be created that stores the words in alphabetical order. This involves traversing the linked list, extracting each word and its occurrence count, and adding them to the new list. The list is then sorted in ascending order based on the first letter of the words and displayed along with their frequencies.

In this tool, the longest word is found by traversing the entire list and comparing the length of each word at every step. The longest word found so far is then stored and updated as the temporary pointer moves through the entire list. Once the end of the list is reached, the longest word and its frequency are returned, which can then be displayed. This method only goes through the list once, hence the complexity of O(n).

To identify the most frequent word, a method is created that traverses the entire list. At each step, the method compares the current word's count with the highest count so far and updates the most frequent word. At the end of the list, the method returns the most frequently used word and its frequency. This method goes through the entire list once, hence its complexity of O(n).

To output the frequency of a given word, a method is defined with the given word as the argument. Since all words stored in the list are in lowercase format, the given word is also

converted to lowercase to match the words in the list. The method then traverses the list to search for a matching word. Once the word is found, its frequency is accessed and displayed. The method might have to go through the entire list to find a word that matches the given word, hence a complexity of O(n).

4. COMPLEXITY ANALYSIS

The most complex method used is sorting words in ascending alphabetical order. The method creates a new list and goes through the entire original list once as it adds the words and their frequencies to the new list. The complexity of going through the original list is O(n) since we are visiting each node once, and the complexity of adding each word and its count to the new list is also O(n). After adding each word, the method moves to the next node, with an O(n) complexity. Each node equals 1 operation, and n nodes equals n operations, resulting in a time complexity of O(1) each, and O(n) each in total, as the operation is linear. The new list is then sorted using OrderBy which sorts using the first letter in the words. Here, .Sort() could also be used as an alternative to OrderBy, and the complexity would still be O(n log n). OrderBy uses a comparison-based sorting algorithm like merge sort, which has a complexity of O(n log n). After sorting, the method then loops through the entire list to print out each word and its frequency. This complexity is O(n).

Therefore, the total complexity is O(n) + O(n) + O(n) + O(n log n) + O(n) = O(n log n) as it is the fastest.

SUMMARY

While(temp != null)	O(n)
sortedList.Add((temp.Data, temp.Count))	O(1) each, Total O(n)
Temp = temp.Next	O(n)
sortedList.OrderBy(w => w.word)	O(n log n)
Foreach (var item in sorted)	O(n)
TOTAL	**O(n log n)**

REAL WORLD USE CASES

Content needs to be clear and appropriate in fields like education or publishing. For instance, children's storybooks should not contain long words that are too difficult for them to pronounce. In this case, a text analysis tool can be handy when editing to identify long and complex words.

Another practical use of this text analysis tool is in academic writing feedback. For instance, a primary or secondary school teacher may want to identify overused words in a student's essay. Instead of manually skimming through the entire essay as they record repeated words such as "then" and "like," they can input the essay into the tool and request the most frequently used word.

Using the previous example, where the therapist seeks to detect whether individuals are depressed or not, they can input the online texts and also provide keywords they use to analyse depression, see the number of times an individual uses the words, and make decisions based on the output.

5. CONCLUSION

In conclusion, this report demonstrates the practical use of a singly linked list to solve real-world problems. By manually building a dynamic data structure to store and track word statistics, the application offers an efficient approach to analysing texts.