King Saud University

College of Computer and Information Sciences

Data Structures (CSC 212)

Fall Trimester 2022

Course Programming Assignment

Phase 1 (07 October 2022 11:59 pm).

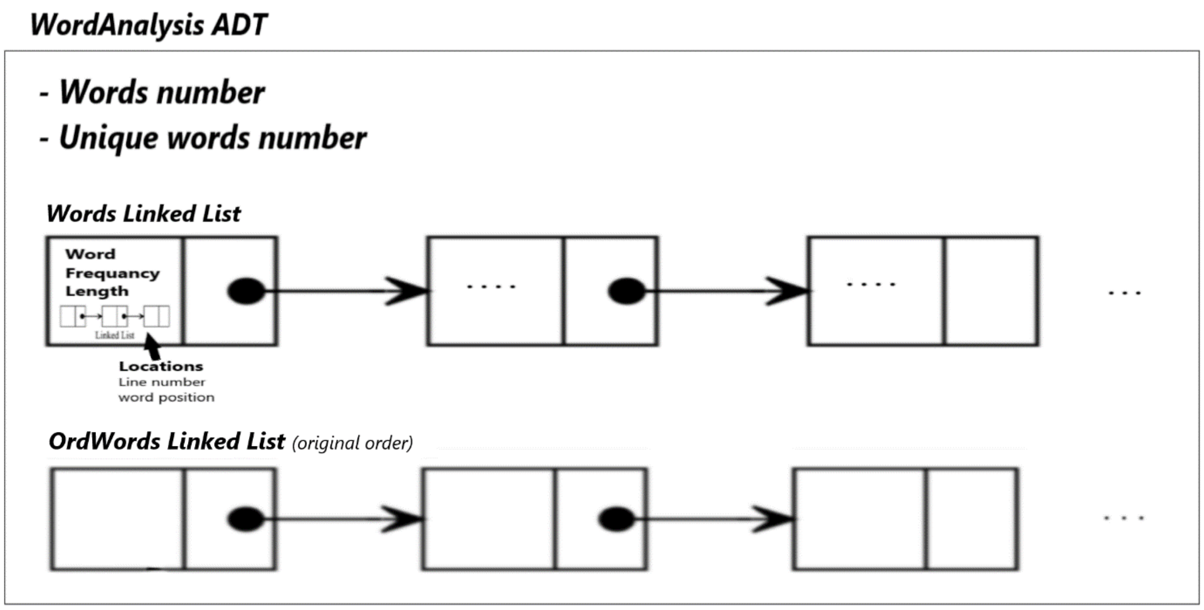
Team#9		**Phase#1**
Name	***ID***	**Section**
Sarah Alajlan	***442202314***	55027
Norah alguraishi	***442201375***	
Majd Alqahtani	***442202282***	

Instructor: Lubna Alhinti

Phase#1

In the first phase of the project, you are asked to describe your suggested design of the ADT for the problem described above and perform the following tasks:

(a) Give a graphical representation of the ADT to show its structure. Make sure to label the diagram clearly.



(b) Write at least one paragraph describing your diagram from part (a). Make sure to clearly explain each component in your design. Also, discuss and justify the choices and the assumptions you make.

- We will use a Words Linked-List which will store words, its frequencies and lengths, and also it will contain another Linked-List to store the word's Location as a Line number and the word position in each node.
- A Linked List OrdWords stores the words in their original order in the file as it will be useful for operation (7).
- after trying out different ADTS we concluded that the linked list is most suitable for the following reasons:
  1/the linked list is more flexible compared to an array. if we would want to update the file with more words in the future, we simply could add more nodes on the other hand, the array will be restricted on one size which will be hard adding new words in the same array.
  2/ the time complexity is better with most methods compared with other ADTS we tried in the process.

Assumption: - duplication isn't allowed within the linked list because we have to keep track of the occurrences of the words therefore duplicated words will be stored as a one word in the list while updating the frequency

(c) Give a specification of the operations (1), (2), (3), (4), (5), (6), and (7) as well as any other supporting operations you may need to read the text from a text file and store the results in the ADT (e.g., insert).

Operations:

1. **Method** Length(Len: int)

Requires: Linked list Words is not empty. **Input:** none.

Result: Displays the total number of words in a text file. **Output:** Len.

2. **Method** UniqueLength(Len: int)

Requires: Linked list Words is not empty and len has a valid value. **Input:** none.

Result : Displays the total number of unique words in a text file. **Output** : Len.

3. **Method** WordOccurrences(Word : String , Count : int)

Requires: Linked list Words is not empty and Word already exists . **Input:** Word.

Result: Search for a particular word in Words linked list ,then returns the total number of occurrences of that word. **Output:** Count.

4. **Method** Word_SpecficLength(Len: int, Count : int)

Requires: Linked list Words is not empty and len has a valid value. **Input**: Len.

Result: Displays the total number of words with a particular length that is given. **Output**: Count.

5. **Method** SortedOccurrences()

Requires: Linked list Words is not empty. **Input**: none.

Result: Displays the unique words and their occurrences Sorted by total occurrences of each word (form most frequent to the least). **Output**: none.

6. **Method** Occurre_locWord(word: String)

Requires: linked List Words is not empty and no null values. **Input:** word

Results: Displays every words position in the following format (line number , position within one line). **Output:** none.

7. **Method** Adjacent(word : String , word1: String , isAdj : Boolean)

Requires: Linked list OrdWords is not empty and no null values for the inputs. **Input:** String word and String word1.

Results: checks if the two words occur adjacent to each other in the file one occurrence of both words is enough to satisfy it. Doing so, by checking the occurrence and the positions of the two words whether they are adjacent or not. **Output:** isAdj.

8. **Method** Sort_Pq()

Requires: Linked list Word is not empty. **Input:** none

Results: the words are sorted according of their frequency/occurrence. By making the frequency equals the priority in the priority queue then the sorted list will be returned to the Words linked list.

Output: List.

9. **Method** Read(fname: String)

Requires: File is not empty. **Input:** fname.

Results: it will read from the file fname and stores every word in a separate node with its frequency, locations and length. Single letters, hyphenated and apostrophized words will be treated as a one word and punctuation will be ignored.

Output: none

(d) Provide the time complexity (worst case analysis) for all the operations discussed above using Big O notation. For operations (3) and (4), consider two cases:

the first case, when the words in the text file have lengths that are evenly distributed among different lengths (i.e., the words should have different lengths starting from 1 to the longest with k characters),

the second case, when the lengths of words are not evenly distributed. For all operations, assume that the length of the text file is n, the number of unique words is m, and the longest word in the file has a length of k characters.

1. Length() **O(1)**

2. UniqueLength() **O(1)**

3. WordOccurrences(String Word)

i. Evenly distributed **O(m)**

ii. Not evenly distributed **O(m)**

4. Word_SpecficLength(int len)

i. Evenly distributed **O(1)**

ii. Not evenly distributed **O(m)**

5. SortedOccurrences() **O(m)**

6. Occurre_locWord(String word) **O(m)**

7. Adjacent(String word , String word1) **O(n)**