WordAnalysis ADT

CSC212

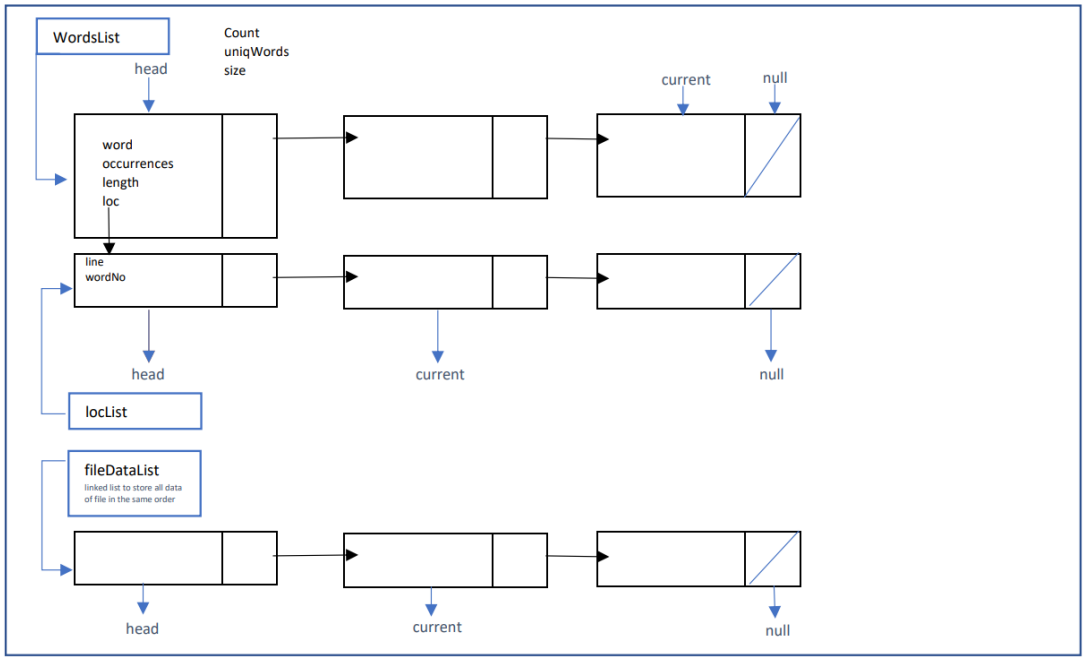
Phase One

Dr. Rehab AlAhmadi

Reema AlMansour- 442200477 (Leader)
Reema AlAngari- 442200134
Group 10
Section 41196

Graphical Representation

WordAnalysis_ADT



Overview

In this project, we designed a Java text analysis -WordAnalysis ADT tool- that will read an input, a text file in (.txt) format, store it in the main memory, then perform several word analytics tasks such as determining the number of occurrences and the locations of different words. This tool has two linked lists wordsList and fileDataList. The wordsList is used to store each word in the text file in it, with the number of its occurrences. The tool will check if the word has a previous occurrence, if it does, the number of occurrences will increase, if not, it will add the word to the list and set the occurrences to 1; no word is stored twice. The list then uses a priority queue to sort itself based on the number of occurrences (most to least). On the other hand, the fileDataList will be used to save the original order of the words in the text file.

Specifications

Elements:

The elements are of generic type <Type> (The elements are placed in nodes for linked list implementation).

Structure:

The elements are linearly arranged. The first element is called head, there is an element called current.

Domain:

The number of elements in the list is bounded therefore the domain is finite. Type name of elements in the domain: List.

Operations:

We assume all operations operate on a list wordsList.

1. **Method** WordsList(String file, Linked list wordsList)

   Requires: Text file not empty. Input: file.

   Results: Reads data from the text file and stores it in wordsList. Each node will hold a single word from the text file, the same word can't be stored twice. The number of occurrences increases if the word appears twice or more and has a new location in the locList. Each node will have a word, occurrences, length and location. After adding all words to the list, we sort them using a priority queue taking the occurrences value as the priority value. Output: wordsList.

   - All words are arranged in the same order as in the original text file.
   - If the same word occurs again in the text file, it is saved in its own node with a different frequency and location.
   - All words are separated by at least one space.
   - Single letter words (e.g., a, I) are counted as words.
   - Punctuation (e.g., commas, periods, etc.) is to be ignored.
   - Hyphenated words (e.g., decision-makers) or apostrophized words (e.g., customer's) are to be read as single words.

2. **Method** WordsCount (Integer size)

   Requires: non-empty linked list wordsList. Input: none.

   Results: Displays the total number of all words in a text file. Output: size.

3. **Method** uniqWords(Integer uniq_size)

   Requires: non-empty linked list wordsList. Input: none.

   Results: Displays the total number of all unique words in a text file. Output: uniq_size

4. **Method:** wordOccur(String word, Integer count)

   Requires: non-empty linked list wordsList. Input: word (word is not null)
   Results: Displays the total number of occurrences of a particular given word in the text file. Output: count

5. **Method** WordLenght(Integer length, Integer count)

   Requires: non-empty linked list wordsList. Input: length. (length>=1)

   Results: Displays the total number of words of a specific length in a text file. Output: count

6. **Method** displayUniqueOccur()

   Requires: non-empty linked list wordsList. Input: none
   Results: Uses priority queue to arrange words and then displays the unique words and their occurrences sorted by the total occurrences of each word (from the most frequent to the least). Output: none.

7. **Method** displayOccurLoc(String word)

   Requires: non-empty linked list wordsList. Input: word. (word is not null)

   Results: Displays the locations of the occurrences of a word starting from the top of the text file (i.e., as a list of line and word positions). Every new-line character '\n ' indicates the end of a line. Output: none.

8. **Method** addjWords(String word1, String word2, Boolean flag)

   Requires: non-empty linked list wordsList. Input: word1, word2 (word1 and word2 are not null)
   Results: Searches for the first word (word1) and checks if the second word (word2) occurs adjacent to it. If it does once or more in the text file linked list, then the flag is set to true (at least one occurrence of both words is needed to satisfy this operation). If one or both words are not found in the file, the flag is set to false. Output: flag.

9. **Method** priorityQsort()

   Requires: non-empty linked list wordsList. Input: none

   Results: Sorts all of the list words in a priority queue according to their occurrences in the text file, number of occurrences is used as the value of priority, queue is then moved back to the list in order of occurrences (most to least). Output: Sorted list.

Time Complexity

1. **Method** public int WordsCount() **O(1)**

2. **Method** public int uniqWords() **O(1)**

3. **Method** public int wordOccur(String word) **O(m)**

If all words occur once, it returns 1. O(1)

4. **Method** public int WordLenght(int length) **O(m)**

If each word has a different length, it returns 1. O(1)

5. **Method** public void displayUniqueOccur() **O(m)**

6. **Method** public void displayOccurLoc(String word) **O(m)**

7. **Method** public boolean addjWords(String word1, String word2) **O(n)**