

**Gebze Technical University
Computer Engineering**

CSE 222 - 2018 Spring

HOMEWORK 06 REPORT

**AHMED SEMİH ÖZMEKİK
171044039**

Course Assistant:

1 INTRODUCTION

1.1 Problem Definition

Our main goal is to perform basic Natural Language Processing operations which are retrieving bi-grams and calculating TFIDF values from a given dataset. Let's try to define those operations and the problem in detail. For a given text dataset consisting of multiple files which have meaningful sentences for a particular language, we are expected to retrieve bi-grams. Now, what is a bi-gram? Bi-gram is a piece of text containing of two sequential words which occurs in a given text at least once, and they are very informative tools to reveal the semantic relation between words. But for our scope we only concern about how to find and extract them from this raw dataset. Another operation we are expected to perform is that to calculate TFIDF values. TFIDF is some kind of a score which reflects the importance of a word for a single document.

Our main problem in general, is defined above; now with considering the arsenal we are given for to achieve this goal, let's expand our problem definition a bit more. We have two different maps defined for us. One of them is mapping the words to the other map which is mapping the file names to the position lists. Let's call our first map as "Word Map" and the other map as "File Map". Again, "Word Map" maps the words to "File Map"s. Each word has one "File Map" and each "File Map" has file names mapping to the position lists of that specific word which was mapping to it's "File Map" in the first place. And those two maps and their relations in our arsenal defines our structure which we must use to achieve our main goal. We will talk about this structure and it's relations more in the System Requirements section.

1.2 System Requirements

Our core requirement for to achieve our main goal is the structure we defined above. Let's try to understand this structure in detail. We have two different maps in this structure. Let's start with the first one which we called as "Word Map". The key is being "words" for this particular map and the value is our second map, but will come to that soon. "Word Map" is a hash map and uses linear probing in it's hash table. But in "Word Map" we are defined with another structure among the keys (words) of this map. This another structure is very similar to linked-list, which actually links the keys to each other: Each entry in our hash table keeps a pointer to the next inserted entry, which improves some particular method's efficiency in our implementation while making the "Word Map" is iterable again with

efficiency. Those mentioned operations becomes efficient because the traversal accomplished without the need for visiting empty cells in table.

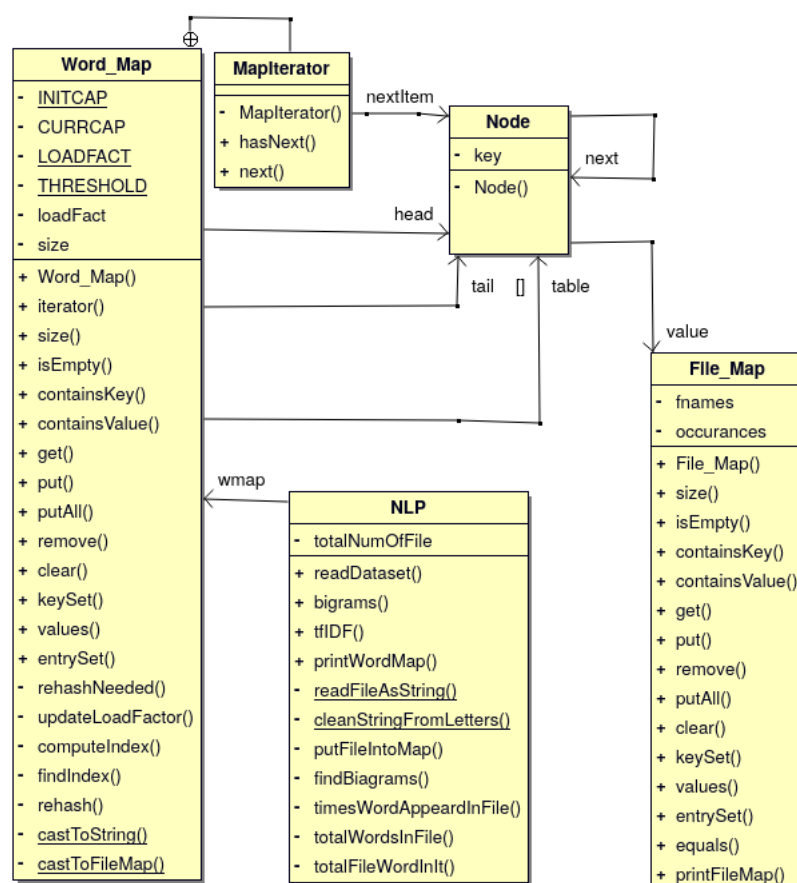
Our second map is the value for “Word Map”, which we called as “File Map”. As the name suggest, it is a file map with keys are file names and the values are position lists. Each “File Map” belongs to a specific word, meaning that those file names are the name of the files where that specific word appears in. And the position list is the list of positions of where this words appeared exactly.

After we satisfy all those requirement of this structure, we can consider on problems such as how to find bi-grams and so on. Armed with this structure, this tasks becoming easier. For instance, since we know the position list and the file for a given word, we can easily determine bi-grams from those maps, with checking the next positions of each position in that file.

After achieving this goals, we are only left with a requirement of simple input-output interaction with user. We will have two different query types: One for retrieving bi-grams of a word and the other one is for calculating the TFIDF value.

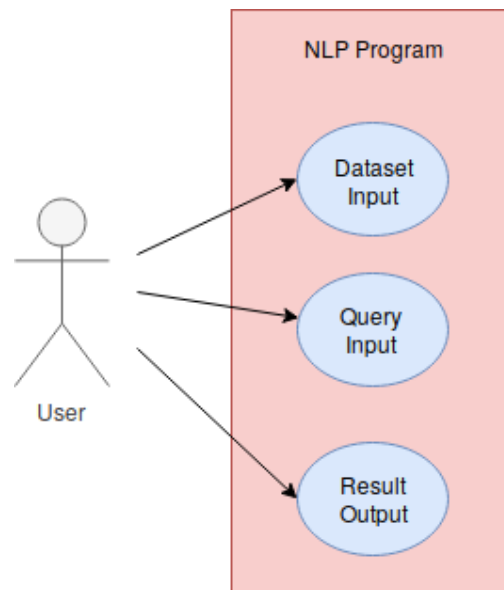
2 METHOD

2.1 Class Diagrams



2.2 Use Case Diagrams

User will give an input file in which consists of multiple queries and will provide a dataset. NLP program will print the result to standard output:



2.3 Problem Solution Approach

I have connected each keys in “Word Map” so that we can traverse the map with more efficiently. In the hash table algorithm for “Word Map” I have kept table length as odd number so that will make collisions less frequent. Even though, choosing list implementation in the “File Map” was costly, inserting all the positions into the list in the occurrences list made the insertion operation simple while I filling the structure. If I were to find bi-grams and TFIDF from just the dataset without using any controller structure in between, that would be very costly compared to now. To be able to iterate over the map efficiently in any case, (which I did a lot in NLP implementation) was a good approach, so to say. Beside the design solution approaches of two maps, let’s examine the algorithms I followed in “NLP” class.

The “readDataSet()” method was quite straight forward. For each file from dataset, I got the text inside the file as whole string, and thanks to the regular expressions I was being able to turn the raw text string to a clean and useful string which purged from any unwanted letters. Then, after I got the words array from text string, I put each word to the “Word Map” gracefully.

In the “biagrams()” for a given word’s filemap, I compared the each position in the position list with the words in the word map, to find which one of them occurs next to the given word.

In the “tfidf()” method, I found every argument in the formula separately. Hence it would be much cleaner; and it was quite simple task to get necessary arguments once we had the structure defined.

For the last, in “printWordMap()” I have used the iterator for “Word Map”, and got benefit from it.

You can find the time complexities and implementation details for each method in the following pages.

3 RESULT

3.1 Test Cases

For the test cases, let’s test each unit:

Case 1: (Word_Map)

I have used the same “File_Map” for to test this unit:

```
public static void main(String[] args) throws IOException {
    File_Map file_maps = new File_Map();
    Word_Map word_map = new Word_Map();

    file_maps.put("a", 13);
    file_maps.put("b", 15);

    word_map.put("word1", file_maps);
    word_map.put("word2", file_maps);
    word_map.put("word3", file_maps);
    word_map.put("word4", file_maps);
    word_map.put("word5", file_maps);

    for(Object object: word_map){
        String word = (String) object;
        System.out.println("Word: " + word);
        File_Map fmap = (File_Map) word_map.get(word);
        fmap.printFileMap();
    }
}
```

Case 2: (File_Map)

```
public static void main(String[] args) throws IOException {
    File_Map file_maps = new File_Map();
    Word_Map word_map = new Word_Map();

    file_maps.put("filename01", 13);
    file_maps.put("filename01", 20);
    file_maps.put("filename01", 33);
    file_maps.put("filename02", 5);
    file_maps.put("filename02", 57);
    file_maps.put("filename02", 90);

    file_maps.printFileMap();
}
```

Case 3:

Now, let's test those both units together, with the real data:

```
NLP nlp = new NLP();
nlp.readDataset( dir: "Test/");
nlp.printWordMap();
```

```
Word0 word1 word2 word3 word1 word3 word2
```

Case 4:

Checking the “dataset” file for “grow” with recursively:

```
NLP nlp = new NLP();
nlp.readDataset( dir: "dataset/");
System.out.println(nlp.bigrams( word: "grow"));
```

```
drh0use@wife:~/Downloads/HW6$ grep -rnw "dataset" -e 'grow'
dataset/0000167:2:Indonesia's agriculture sector will grow
dataset/0006429:3:and grow more high-value export crops, the communist party
dataset/0001184:54:grow only one pct this year compared to five pct in 1986,
drh0use@wife:~/Downloads/HW6$
```

Case 5:

Testing the concrete program:

```
NLP nlp = new NLP();
nlp.readDataset( dir: "dataset/");
String filename = "input.txt";

String queryFile = new String(Files.readAllBytes(Paths.get(filename)));
String[] lines = queryFile.split( regex: "\\r?\\n");
String[][] queries = new String[lines.length][];

for (int i=0; i<lines.length; ++i)
    queries[i] = lines[i].split( regex: "\\s+");

for (String[] query: queries){
    if (query.length == 2) // bi-grams query.
        System.out.println(nlp.bigrams(query[1]));
    else // TFIDF query.
        System.out.printf("%.7f\\n", nlp.tfIDF(query[1], query[2]));
    System.out.print("\\n");
}
```

```
bigram very
tfidf coffee 0001978
bigram world
bigram costs
bigram is
tfidf Brazil 0000178|
```

3.2 Running Results

Result 1:

```
Word: word1
      a->[13]
      b->[15]
Word: word2
      a->[13]
      b->[15]
Word: word3
      a->[13]
      b->[15]
Word: word4
      a->[13]
      b->[15]
Word: word5
      a->[13]
      b->[15]
```

Result 2:

```
/usr/lib/jvm/java-1.11.0-openjdk-amd64/
      filename01->[13, 20, 33]
      filename02->[5, 57, 90]

Process finished with exit code 0
```

Result 3:

```
[Word0]--->
          0000026->[0]
[word1]--->
          0000026->[1, 4]
[word2]--->
          0000026->[2, 6]
[word3]--->
          0000026->[3, 5]
```

Result 4:

```
Enter
/usr/lib/jvm/java-1.11.0-openjdk-amd64/bin/
File>0001184 biagram>grow only
File>0006429 biagram>grow more
File>0000167 biagram>grow by
[grow only, grow more, grow by]

Process finished with exit code 0
```

Result 5:

```
/usr/lib/jvm/java-1.11.0-openjdk-amd64/bin/java -javaagent:/home/drh0use/Downloads/Test/idea-IC-183
[very vulnerable, very rapid, very promising, very aggressive, very soon, very difficult, very attr

0.0048782

[world markets, world prices, world market, world cocoa, world coffee, world for, world bank, world

[costs of, costs Transport, costs have, costs and]

[is likely, is committed, is a, is now, is very, is passed, is difficult, is unlikely, is planned,

0.0073839

Process finished with exit code 0
```

Note: I wanted to write my calculations of time complexities in here, but trying to write equations of the calculations was almost impossible in Libre Office. Instead, I wrote them in Latex and added to this file. So, for the full report there might a small inconsistency with margins and font etc. But I had no other choice. I have given all the details about the calculations of time complexities of methods, starting from next page and continues until the end of the report.

1. WordMap.findIndex(key): Linear probing.

```
private int findIndex(String key){
    int idx = computeIndex(key);

    while(table[idx] != null && !key.equals(table[idx].key)){
        ++idx;
        if (idx >= table.length) // back to top.
            idx = 0;
    }

    return idx;
}
```

Figure 1: WordMap.findIndex()

$$\begin{aligned} T(n) &= T_{\text{WordMap.computeIndex}()} + 4 \times n \\ T_{\text{best}}(n) &= \mathcal{O}(1) \\ T_{\text{worst}}(n) &= \mathcal{O}(n) \end{aligned} \tag{1}$$

2. WordMap.clear():

```
public void clear() {
    size = 0;
    table = new Node[CURRCAP];
    updateLoadFactor();
}
```

Figure 2: WordMap.clear()

$$\begin{aligned} T(n) &= 2 + T_{\text{WordMap.updateLoadFactor}()}(n) \\ T(n) &= \mathcal{O}(1) \end{aligned} \tag{2}$$

3. WordMap.containsKey():

```
public boolean containsKey(Object key) {
    int idx = findIndex((String) key);
    if (table[idx] == null)
        return false;

    return true;
}
```

Figure 3: WordMap.containsKey()

$$\begin{aligned} T(n) &= T_{\text{WordMap.findIndex}()}(n) + 2 \\ T(n) &= \mathcal{O}(1) \end{aligned} \tag{3}$$

4. WordMap.containsValue():

```
public boolean containsValue(Object value) {
    for (Object theKey: this){
        File_Map theValue = (File_Map) get(theKey);
        if (theValue.equals(value))
            return true;
    }
    return false;
}
```

Figure 4: WordMap.containsValue()

$$\begin{aligned} T(n) &= 3 \times n \\ T(n) &= \mathcal{O}(n) \end{aligned} \tag{4}$$

5. WordMap.get():

```
public Object get(Object key) {
    String theKey = castToString(key);

    int idx = findIndex(theKey);
    if (table[idx] == null)
        throw new NoSuchElementException("Key does not exist!");
    return table[idx].value;
}
```

Figure 5: WordMap.get()

$$\begin{aligned} T(n) &= T_{\text{WordMap.findIndex}}(n) + 3 \\ T_{\text{best}}(n) &= \mathcal{O}(1) \\ T_{\text{worst}}(n) &= \mathcal{O}(n) \end{aligned} \tag{5}$$

6. WordMap.rehashNeeded(), WordMap.updateLoadFactor(), WordMap.computeIndex():

```
private boolean rehashNeeded() { return loadFact > THRESHOLD; }

private void updateLoadFactor() { loadFact = (float) size / table.length; }

private int computeIndex(String key)
{
    int index = key.hashCode() % CURRCAP;
    if (index < 0)
        index += table.length;
    return index;
}
```

Figure 6: 3 methods.

$$\begin{aligned} T_{\text{WordMap.rehashNeeded}}(n) &= \mathcal{O}(1) \\ T_{\text{WordMap.updateLoadFactor}}(n) &= \mathcal{O}(1) \\ T_{\text{WordMap.computeIndex}}(n) &= \mathcal{O}(1) \end{aligned} \tag{6}$$

7. WordMap.keySet():

```
public Set keySet() {
    Set<String> set = new HashSet<>();

    for (Object key:this)
        set.add((String) key);

    return set;
}
```

Figure 7: WordMap.keySet()

$$\begin{aligned} T(n) &= T_{Set.add()}(n) \times n + 2 \\ T(n) &= \mathcal{O}(n) \end{aligned} \tag{7}$$

8. WordMap.put():

```
public Object put(Object key, Object value) {

    String theKey = castToString(key);
    File_Map theValue = castToFileMap(value);

    // findIndex() uses linear probing.
    int idx = findIndex(theKey);

    if (table[idx] == null) { // Key does not exist, so insert the entry.
        table[idx] = new Node(theKey, theValue);
        ++size;
        updateLoadFactor();

        if(head == null){
            head = table[idx];
            tail = head;
        }
        else{
            tail.next = table[idx];
            tail = tail.next;
        }

        if(rehashNeeded())
            rehash();
    }
}
```

Figure 8: WordMap.put()

$$\begin{aligned} T(n) &= 8 + T_{WordMap.findIndex()}(n) + T_{WordMap.updateLoadFactor()}(n) + T_{rehashNeeded()}(n) + T_{rehash}(n) \\ T_{best}(n) &= \mathcal{O}(1) \\ T_{worst}(n) &= \mathcal{O}(n) \end{aligned} \tag{8}$$

9. WordMap.putAll():

```
public void putAll(Map m) {
    Map<String, File_Map> map = (Map<String, File_Map>) m;
    for (Map.Entry<String, File_Map> entry: map.entrySet())
        put(entry.getKey(), entry.getValue());
}
```

Figure 9: WordMap.putAll()

$$\begin{aligned} n &= m.size() \\ T(n) &= T_{WordMap.put}() \times n \\ T(n) &= \mathcal{O}(n) \end{aligned} \tag{9}$$

10. WordMap.rehash():

```
private void rehash(){
    CURRCAP = 2*CURRCAP + 1; // less collusion probability in odd numbers.
    table = new Node[CURRCAP];

    Node temp = head;
    head = null;
    tail = null;
    size = 0;
    while(temp != null){
        put(temp.key, temp.value);
        temp = temp.next;
    }

    updateLoadFactor();
}
```

Figure 10: WordMap.rehash()

$$\begin{aligned} T(n) &= 6 + T_{WordMap.put}() \times 2 + T_{WordMap.updateLoadFactor}() \times 1 \\ T(n) &= \mathcal{O}(n) \end{aligned} \tag{10}$$

11. WordMap.values():

Figure 11: WordMap.values()

```
public Collection values() {
    Collection set = new HashSet();
    for (Object key: this)
        set.add(get(key));
    return set;
}
```

$$\begin{aligned} T(n) &= T_{Set.add}() \times n \times T_{get}() \times 1 \\ T(n) &= \mathcal{O}(n) \end{aligned} \tag{11}$$

12. FileMap.containsKey():

```
public boolean containsKey(Object key) {  
    String theKey = (String) key;  
    return fnames.contains(theKey);  
}
```

Figure 12: FileMap.containsKey()

$$\begin{aligned} T(n) &= T_{ArrayList.contains(n)}(n) + 2 \\ T(n) &= \mathcal{O}(n) \end{aligned} \tag{12}$$

13. FileMap.containsValue():

```
public boolean containsValue(Object value) {  
    Integer theValue = (Integer) value;  
    for (List list: occurrences){  
        if (list.contains(theValue))  
            return true;  
    }  
    return false;  
}
```

Figure 13: FileMap.containsValue()

$$\begin{aligned} T(n) &= T_{ArrayList.contains(n1)}(n) \times T_{ArrayList.contains(n2)}(n) \\ T_{best}(n) &= \mathcal{O}(n) \end{aligned} \tag{13}$$

14. FileMap.size(), FileMap.isEmpty():

Figure 14: FileMap.size(), FileMap.isEmpty()

```
@Override  
public int size() { return fnames.size(); }  
  
@Override  
public boolean isEmpty() { return fnames.isEmpty(); }  
  
/
```

$$\begin{aligned} T_{FileMap.size()}(n) &= T_{ArrayList.size(n)}(n) \\ T_{FileMap.size()}(n) &= \mathcal{O}(1) \\ T_{FileMap.isEmpty()}(n) &= T_{ArrayList.isEmpty(n)}(n) \\ T_{FileMap.isEmpty()}(n) &= \mathcal{O}(1) \end{aligned} \tag{14}$$

15. FileMap.entrySet():

```
public Set<Entry> entrySet() {
    Set<Entry> set = new HashSet<>();

    for(int idx = 0; idx<fnames.size(); ++idx){
        set.add(new AbstractMap.
            SimpleEntry<>(fnames.get(idx), occurances.get(idx)));
    }

    return set;
}
```

Figure 15: FileMap.entrySet()

$$\begin{aligned} T(n) &= 1 + T_{Set.add()}(n) \times n \\ T(n) &= \mathcal{O}(n) \end{aligned} \tag{15}$$

16. FileMap.keySet(), FileMap.values():

Figure 16: FileMap.keySet(), FileMap.values()

```
public Set keySet() { return new HashSet<>(fnames); }

/**
 * Returns the Set of Positions.
 * @return the Set of Positions.
 */
@Override
public Collection values() { return new HashSet<>(occurances); }

/**
```

$$\begin{aligned} T_{FileMap.keySet()}(n) &= \mathcal{O}(n) \\ T_{FileMap.values()}(n) &= \mathcal{O}(n) \end{aligned} \tag{16}$$

17. FileMap.put():

```
public Object put(Object key, Object value) {
    String theKey = (String) key;
    Integer theValue = (Integer) value;

    if (!containsKey(theKey)){ // insert the new pairs, if key does not exist
        fnames.add(theKey);
        occurances.add(new ArrayList<>());
    }

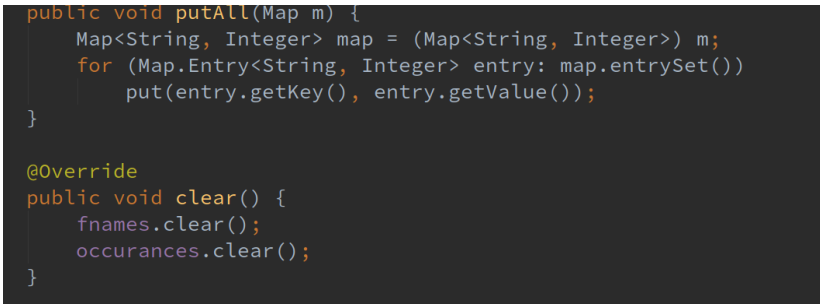
    int index = fnames.indexOf(theKey);
    if (!occurances.get(index).contains(theValue)) // key exists but value does not
        occurances.get(index).add(theValue);

    return null;
}
```

Figure 17: FileMap.put()

$$\begin{aligned}
T(n) &= 3 + T_{ArrayList.add()}(n) \times 3 + T_{containsKey()}(n) + T_{ArrayList.get()}(n) \times 2 + T_{ArrayList.contains()}(n) \\
T(n) &= \mathcal{O}(n)
\end{aligned}
\tag{17}$$

18. FileMap.putAll(), FileMap.clear():



```

public void putAll(Map m) {
    Map<String, Integer> map = (Map<String, Integer>) m;
    for (Map.Entry<String, Integer> entry: map.entrySet())
        put(entry.getKey(), entry.getValue());
}

@Override
public void clear() {
    fnames.clear();
    occurances.clear();
}

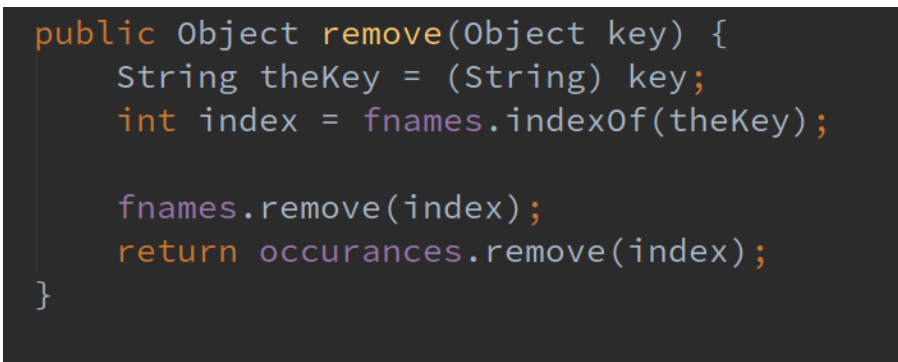
```

clear.png

Figure 18: FileMap.putAll(), FileMap.clear()

$$\begin{aligned}
T(n) &= m.size() \times n + 1 \\
T(n) &= \mathcal{O}(m.size() \times n)
\end{aligned}
\tag{18}$$

19. FileMap.remove():



```

public Object remove(Object key) {
    String theKey = (String) key;
    int index = fnames.indexOf(theKey);

    fnames.remove(index);
    return occurances.remove(index);
}

```

Figure 19: FileMap.remove()

$$\begin{aligned}
T(n) &= 1 + T_{ArrayList.indexOf}(n) + T_{ArrayList.remove()}(n) \times 2 \\
T(n) &= \mathcal{O}(n)
\end{aligned}
\tag{19}$$

20. NLP.cleanStringFromLetters():

```
private static String cleanStringFromLetters(String file){
    return file.
        replace(target: "\n", replacement: " "). // clean new line.
        replace(target: "\r", replacement: ""). // clean carriage return.
        replace(target: " ", replacement: ""). // clean heads.
        trim(). // clean redundant spaces.
        replaceAll(regex: "\\p{Punct}", replacement: ""); // clean punctuation.
}
```

Figure 20: NLP.cleanStringFromLetters()

$$\begin{aligned} n &= \text{String.length}()T(n) = T_{\text{String.replace}}(n) \times 4 + T_{\text{String.trim}}(n) \\ T(n) &= \mathcal{O}(n) \end{aligned} \quad (20)$$

21. NLP.readFileAsString():

```
/* Returns the given file as a whole string */
private static String readFileAsString(String file){
    try {
        return new String(Files.readAllBytes(Paths.get(file)));
    } catch (IOException e) {

        System.out.println("In reading file:" + e.getMessage());
        e.printStackTrace();
    }
    return file; // Exception thrown case, return the input file.
}
```

Figure 21: NLP.readFileAsString()

$$\begin{aligned} n &= \text{words} \\ T(n) &= \mathcal{O}(n) \end{aligned} \quad (21)$$

22. NLP.readDataset():

```
/*Reads the dataset from the given dir and created a word map */
public void readDataset(String dir){

    File[] datasetFileList = Objects.requireNonNull(new File(dir).listFiles());
    totalNumOfFile = datasetFileList.length;

    // Put each file to the map.
    for (File datasetFile : datasetFileList) {
        String text = readFileAsString(datasetFile.getAbsolutePath());
        text = cleanStringFromLetters(text);
        putFileIntoMap(text, datasetFile.getName());
    }
}
```

Figure 22: NLP.readDataset()

$$\begin{aligned}
n_1 &= totalNumOfFile \\
n_2 &= wordInEachFile \\
n &= n_1 \times n_2 (totalNumOfWordInDataset) \\
T(n) &= T_{NLP.readFileAsString()}(n) + T_{NLP.cleanStringFromLetters()}(n) + T_{NLP.putFileIntoMap()}(n) \\
T(n) &= \mathcal{O}(n)
\end{aligned} \tag{22}$$

23. NLP.putFileIntoMap():

```
private void putFileIntoMap(String text, String filename){
    // Split the text into words with regular expression.
    String[] words = text.split(regex: "\\s+");

    for (int position=0;position<words.length;++position){
        File_Map file_map;
        String word = words[position];

        if (wmap.containsKey(word)){ // map has the word, just add a new position.
            file_map = (File_Map)wmap.get(word);
            file_map.put(filename, position);
        }
        else{ // map doesn't have the word, insert the word and a new file map.
            file_map = new File_Map();
            file_map.put(filename, position);
            wmap.put(word, file_map);
        }
    }
}
```

Figure 23: NLP.putFileIntoMap()

$$\begin{aligned}
n &= words.length() \\
T(n) &= T_{String.split()}(n) + n \times (6) \\
T(n) &= 7 \times n \\
T(n) &= \mathcal{O}(n)
\end{aligned} \tag{23}$$

24. NLP.printWordMap():

```
/*Print the WordMap by using its iterator*/
public void printWordMap()
{
    for (Object object: wmap){
        String key = (String) object;
        File_Map value = (File_Map) wmap.get(key);
        System.out.printf("[%s]--->\n", key);
        value.printFileMap();
    }
}
```

Figure 24: NLP.printWordMap()

$$\begin{aligned}
 n &= \text{totalNumOfWordInDataset} \\
 T(n) &= \mathcal{O}(n)
 \end{aligned}
 \tag{24}$$

25. NLP.biagrams():

```

/*Finds all the bigrams starting with the given word*/
public List<String> bigrams(String word){

    List<String> biagramList = new ArrayList<>();

    if (wmap.containsKey(word))
        findBiagrams(word, biagramList);

    return biagramList;
}

```

Figure 25: NLP.biagrams()

$$\begin{aligned}
 T(n) &= T_{NLP.findBiagrams()}(n) \\
 T(n) &= \mathcal{O}(n)
 \end{aligned}
 \tag{25}$$

26. NLP.findBiagrams():

```

// Traverse the entries in the word's file map.
for (Map.Entry entryInFileMap : wordFileMap.entrySet()) {

    // Get both key and value from the entry.
    String filename = (String) entryInFileMap.getKey();
    List<Integer> sourcePositionList = (List<Integer>) entryInFileMap.getValue();

    // Traverse all the words in the map.
    for (Map.Entry entryInWordMap : wmap.entrySet()) {
        String targetWord = (String) entryInWordMap.getKey();
        File_Map targetFileMap = (File_Map) entryInWordMap.getValue();

        // Whether target map has a filename the source word is in.
        if (targetFileMap.containsKey(filename)) {
            List<Integer> targetPositionList = (List<Integer>)targetFileMap.get(filename);

            // Check the all positions of source word in the target file.
            for (Integer pos : sourcePositionList) {
                if (targetPositionList.contains(pos+1)){
                    String biagram = sourceWord + " " + targetWord;
                    if (!biagramList.contains(biagram))
                        biagramList.add(biagram);
                }
            }
        }
    }
}

```

Figure 26: NLP.findBiagrams()

$$\begin{aligned}
n_1 &= \text{word.fileMap.length}() \\
n_2 &= \text{totalNumOfWordInDataset} \\
n_3 &= \text{totalPositionsOfWordInFile} \\
n &= n_1 \times n_2 \times n_3 \\
T(n) &= \mathcal{O}(n)
\end{aligned} \tag{26}$$