CMP-5014Y Coursework 2 - Word Auto Completion with Tries

Student number: 100263247. Blackboard ID: uxk18qau

Tuesday 28th April, 2020 15:16

Contents

1 Part 1: Form a Dictionary and Word Frequency Count

This is my form dictionary method.
This takes in an arraylist of every word including repeats. It then uses javas inbuilt arraylist.sort() which uses merge sort. After this it goes through the entire arraylist each element at a time, keeping track of the previous word. If the previous word is the same as the current word it simply adds one to the frequency count for that word. When the word is different it adds the previous and the frequency count to the map, then sets the frequency to 1.

Psudocode of my formDictionary algorithm

Algorithm 1 FormDictionary(**in**) **return** *map*

Require: A document, *in*, containing words
Ensure: *map*, is the output containing the frequencies and the word

```
1:  sort(in)                                   ▷ Uses merge sort for arraylist in java
2:  f ← 0
3:  prev ← ""
4:  for word in in do
5:      if map equals(word) then
6:          f ← f + 1
7:      else
8:          mapput(prev, f)
9:      f ← 1
10: prev ← word
return map
```

FormDictionary Time complexity:
There are n words in the array

Worst Case:
Worst case would be when all the words are really long and all the same.
The fundamental operation for my algorithm is line 3
if map contains(word) then
The while Loop then goes through each element an performs the fundamental operation meaning this operation is executed n times giving it a time complexity of O(n)
Java uses Mergesort in ArrayList.sort() which has a time complexity of worst case: O(nLogn)
So it would be O(nLogn)+O(n)
O(n log(n)) is the dominant term so this is what it simplifies to.

Best Case:
Best case would be if all the words are really short and differ from each other as quick as possible making the sort time take less
The fundamental operation for my algorithm is line 3
if map contains(word) then
The while Loop then goes through each element an performs the fundamental operation meaning this operation is executed n times giving it a time complexity of O(n)
Java uses Mergesort in ArrayList.sort() which has a time complexity of best case: O(nLogn)
So it would be O(nLogn)+O(n)
O(n log(n)) is the dominant term so this is what it simplifies to.

Being as the best and worst case time complexity are the same the average case is also the same making it: O(n log(n))

2 Part 2: Implement a Trie Data Structure

I designed algorithms to complete the following
1. boolean add(String key): adds a key to the trie, creating any nodes required and returns true if add was successful (i.e. returns false if key is already in the trie, true otherwise).
2. boolean contains(String key): returns true if the word passed is in the trie as a whole word, not just as a prefix.
3. String outputBreadthFirstSearch(): returns a string representing a breadth first traversal.
4. String outputDepthFirstSearch(): returns a string representing a pre order depth first traversal.
5. Trie getSubTrie(String prefix): returns a new trie rooted at the prefix, or null if the prefix is not present in this trie.
6. List getAllWords(): returns a list containing all words in the trie. The order the words are returned in is unimportant, but you should make your algorithm getAllWords as efficient as possible.

Algorithm 2 add(**wordToAdd**) **return** *wasAlreadThere*

Require: Trie to have been constructed
1: **if** 3(*wordToAdd*)== *false* **then**
2: *current* ← *root*
3: **for** *i* ← 1 to *wordToAdd.length* **do**
4: **if** 15(*nextChar*) **then**
5: *currentNode*.14(*next*)
6: *currentNode* ← *next*
7: *currentNode*.13(*true*)
return *true*

Algorithm 3 contains(**wordToSearchFor**) **return** *wordWasThere*

Require: Trie to have been constructed, *current* ← *root*
1: **for all** charaters in *wordToSearchFor* **do**
2: *node* ← *current*.16(*c*)
3: **if** *node* == *null* **then return** *false*
4: *current* ← *node*
return *current*12

Algorithm 4 outputBreathFirstSearch() **return** *Traverasal*

Require: Trie to have been constructed, *current* ← *root*
1: *q* ←5(*root*, *q*)
2: **while** *q.size*() > 0 **do**
3: *current* ← *q.peek*()
4: *q.remove*
5: *currentChildren* ← *current*.10
6: *q* ←5(*current*, *q*)
7: *Traversal.appened*(*current*.11)
8: **return** *Traversal*

Algorithm 5 addChildrenToQueue(TrieNode node, Queue q) **return** *q*

1: *currentChildren* ← *current*.10
2: **for all** *currentChildren* **do**
3: *q.add*(*currentChild*)
4: **return** *q*

Algorithm 6 outputDepthFirstSearch() **return** $Traverasal$

Require: Trie to have been constructed, $current \leftarrow root$

```
1: rootChildren ← root.10
2: for all rootChildren do
3:     Traversal.append(7(rootChild)
4: return Traversal
```

Algorithm 7 depthRecursive(**node**) **return** $Traverasal$

Require: Trie to have been constructed

```
1: Traversal.append(11)
2: nodeChildren ← node.10
3: for all nodeChildren do
4:     Traversal.append(7(nodeChild)
5: return Traversal
```

Algorithm 8 getSubTrie(**prefix**) **return** *subTrie*

Require: Trie to have been constructed

```
1: current ← root
2: for all char in prefix do node ← current.16(currentChar)
3:     if node == null then
4:         return null
5:     current ← node
6: return new Trie(current)
```

Algorithm 9 getAllWords() **return** *words*

Require: Trie to have been constructed

```
1: if root.12 then
2:     words.add("")
3: rootChildren ← root.10
4: for all rootChildren do
5:     todo.push(rootChild)
6: while todo.size > 0 do
7:     current ← todo.peek
8:     todo.remove
9:     children ← current.10
10:    for all currentChildren do
11:        todo.push(currentChild)
12:    while currentWord.size > 0 do
13:        top ← currentWord.peek
14:        if top.17== false then
15:            done.add(top)
16:            currentWord.pop
17:        else
18:            topChildren ← top.10
19:            for all topChildren do
20:                if not done.3(topChild) then
21:                    doneWith ← false
22:            if doneWith == true then
23:                done.add(top)
24:                currentWord.pop
25:            else
26:                Exitwhileloop
27:    currentWord.push(current)
28:    if current.12 then
29:        for all currentWord do
30:            newWord.append(currentWordNode.11)
31:        words.add(newWord)
32: return words
```

TrieNode class

Algorithm 10 getChildren() **return** *Children*

```
1: return Children
```

Algorithm 11 getLetter() **return** *Letter*

```
1: return letter
```

Algorithm 12 getIsWordEnd() **return** $IsWordEnd$

```
1: return IsWordEnd
```

Algorithm 13 setIsWordEnd(**newIsWordEnd**)

```
1: IsWordEnd ← newIsWordEnd
```

Algorithm 14 addChild(**letter**)

```
1: i ← letter −′ a′
2: children[i] ← newTrieNode(letter)
```

Algorithm 15 isChild(**letter**) **return** *boolean*

```
1: for all children do
2:     if child.11== letter then
3:         return true
4: return false
```

Algorithm 16 getChildNode(**letter**) **return** *letterNode*

```
1: for all children do
2:     if child.11== letter then
3:         return child
4: return null
```

Algorithm 17 hasChildren **return** *letterNode*

```
1: for i = 1to26 do
2:     if 15(i +′ a′) then
3:         return true
4: return false
```

3 Part 3: Word Auto Completion Application

I added parents to the trienodes. This was to get the word when doing a breadth first search. I used breadth first to get the words in get all words then used the parents to track back up the trie to get the word that needed to be returned.
From getting all the words and frequencies I got all the probabilities with the getAllProbabilities function, this made a subtrie and then used that to get all the words and frequencies
This get all probabilities is used to get the top 3 (or less) probabilities for a prefix.

Algorithm 18 add(**key**, **quantity**) **return** *wasAlreadThere*

Require: Trie to have been constructed
1: **if** 19(*wordToAdd*)== *false* **then**
2: *current* ← *root*
3: **for** *i* ← 1 to *wordToAdd.length* **do**
4: **if** 28(*nextChar*) **then**
5: *currentNode*.27(*next*)
6: *currentNode* ← *next*
7: *currentNode*.SetQuantity(*quantity*)
return *true*

Algorithm 19 contains(**wordToSearchFor**) **return** *wordWasThere*

Require: Trie to have been constructed, *current* ← *root*
1: **for all** charaters in *wordToSearchFor* **do**
2: *node* ← *current*.29(*c*)
3: **if** *node* == *null* **then return** *false*
4: *current* ← *node*
return *current.getQuantity* > 0

Algorithm 20 outputBreathFirstSearch() **return** *Traverasal*

Require: Trie to have been constructed, *current* ← *root*
1: *q* ←21(*root*, *q*)
2: **while** *q.size*() > 0 **do**
3: *current* ← *q.peek*()
4: *q.remove*
5: *currentChildren* ← *current.getChildren*
6: *q* ←21(*current*, *q*)
7: *Traversal.appened*(*current.getLetter*)
8: **return** *Traversal*

Algorithm 21 addChildrenToQueue(TrieNode node, Queue q) **return** *q*

1: *currentChildren* ← *current.getChildren*()
2: **for all** *currentChildren* **do**
3: *q.add*(*currentChild*)
4: **return** *q*

Algorithm 22 outputDepthFirstSearch() **return** *Traverasal*

Require: Trie to have been constructed, *current* ← *root*
1: *rootChildren* ← *root.getChildren*
2: **for all** *rootChildren* **do**
3: *Traversal.append*(23(*rootChild*)
4: **return** *Traversal*

Algorithm 23 depthRecursive(**node**) **return** $Traverasal$

Require: Trie to have been constructed
1: $Traversal.append(node.getLetter)$
2: $nodeChildren \leftarrow node.getChildren$
3: **for all** $nodeChildren$ **do**
4: $\quad Traversal.append(23(nodeChild)$
5: **return** $Traversal$

Algorithm 24 getSubTrie(**prefix**) **return** $subTrie$

Require: Trie to have been constructed
1: $current \leftarrow root$
2: **for all** char in $prefix$ **do** $node \leftarrow current.29(currentChar)$
3: $\quad$ **if** $node == null$ **then**
4: $\quad\quad$ **return** $null$
5: $\quad current \leftarrow node$
6: **return** new Trie($current$)

Algorithm 25 getAllWords() **return** $words$

Require: Trie to have been constructed
1: **if** $root.getQuantity() > 0$ **then**
2: $\quad words.put(\text{""}, root.getQuantity())$
3: $q \leftarrow 21(root, q)$
4: **while** $q.size() > 0$ **do**
5: $\quad current \leftarrow q.peek()$
6: $\quad q.remove$
7: $\quad currentChildren \leftarrow current.getChildren()$
8: $\quad q \leftarrow 21(current, q)$
9: $\quad$ **if** $current.getQuantity() > 0$ **then**
10: $\quad\quad word.put((26(current).reverse, current.getQuantity)$
11: **return** $words$

Algorithm 26 getWordFromNode(**node**)

1: **if** $node$!=root **then**
2: $\quad str \leftarrow node.getLetter + 26(node.getParent)$
return str

Algorithm 27 addChild(**letter**)

1: $i \leftarrow letter -' a'$
2: $children[i] \leftarrow newTrieNode(letter, this)$

Algorithm 28 isChild(**letter**) **return** $boolean$

1: **for all** $children$ **do**
2: $\quad$ **if** $child.getLetter == letter$ **then**
3: $\quad\quad$ **return** $true$
4: **return** $false$

Algorithm 29 getChildNode(**letter**) **return** $letterNode$

1: **for all** $children$ **do**
2: $\quad$ **if** $child.getLetter() == letter$ **then**
3: $\quad\quad$ **return** $child$
4: **return** $null$

Algorithm 30 getAllProbabilities(**prefix**) **return** $LinkedHashMap < String, Double >$

```
1: subTrie = getSubTrie(prefix)
2: if subTrie!=null then
3:     total ← subTrie.root.getQuantity()
4:     21(subTrie.root,q)
5:     while q.size() > 0 do
6:         current ← q.peek()
7:         q.remove
8:         currentChildren ← current.getChildren
9:         q ←21(current, q)
10:        total ← current.getQuantity()
11:    allWords ← subTrie.25
12:    for entry in allWords do
13:        if entry.key == "" then
14:            r.put(prefix, entry.value/total)
15:        else
16:            r.put(entry.key, entry.value/total)
return r
```

Algorithm 31 getTopThreeProbabilities(**prefix**) **return** $LinkedHashMap < String, Double >$

```
1: all ← 30(prefix)
2: i ← 0
3: while all.size > 0 and i++ < 3 do
4:     for all key in all do
5:         if all.get(key) > highestProbability then
6:             highestProbability ← all.get(key)
7:             highestKey ← key
8:     r.put(highestKey, highestProbability)
9:     print(highestKey + "=" + highestProbability)
10:    all.remove(highestKey)
11: return r
```

4 Code Listing

Listing 1: DictionaryFinder.java

```
1  package DSA2;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.io.FileWriter;
6  import java.io.IOException;
7  import java.io.PrintWriter;
8  import java.util.*;
9
10 public class DictionaryFinder {
11     public DictionaryFinder(){
12     }
13     /**
14      * 1. read text document into a list of strings;
15      *
16      * Given
17      * Reads all the words in a comma separated text document into an Array
18      *
19      * @param file
20      */
21     public static ArrayList<String> readWordsFromCSV(String file) throws
           ↪ FileNotFoundException {
22         Scanner sc=new Scanner(new File(file));
23         sc.useDelimiter(" |,");
24         ArrayList<String> words=new ArrayList<>();
25         String str;
26         while(sc.hasNext()){
27             str=sc.next();
28             str=str.trim();
29             str=str.toLowerCase();
30             words.add(str);
31         }
32         return words;
33     }
34     /**
35      * Given
36      *
37      * @param c
38      * @param file
39      * @throws IOException
40      */
41     public static void saveCollectionToFile(Collection<?> c,String file) throws
           ↪ IOException {
42         FileWriter fileWriter = new FileWriter(file);
43         PrintWriter printWriter = new PrintWriter(fileWriter);
44         for(Object w: c){
45             printWriter.println(w.toString());
46         }
47         printWriter.close();
48     }
49
50     /**
51      * 2. form a set of words that exist in the document and count the number of
           ↪ times each word
52      * occurs in a method called FormDictionary;
53      * 3. sort the words alphabetically;
```

```
54       *
55       * Sorts all the words into alphabetical order
56       * Keeps track of the previous word
57       * When the current word is different to the previous word put the previous word
             ↪ into the map with the frequency
58       *
59       * @param in all the words in
60       * @return each word, frequencies of each word in alphabetical order
61       */
62      public LinkedHashMap<String,Number> formDictionary(ArrayList<String> in){
63          in.sort(String::compareTo);//sorts all the words into alphabetical order
64          LinkedHashMap<String, Number> map = new LinkedHashMap<>();
65          String prev = "";//keeps track of the previous word
66          int frequency = 0;//sets the frequency to 0 to start
67          for (String word: in){//for each word in the array
68              if (word.equals(prev)){//if it is the same as the last, and therefore
                     ↪ will already be added to the dictionary
69                  frequency+=1;//add one to the frequency that will be added with it
70              } else {//the word has changed and being as the words are sorted
                     ↪ alphabetically the previous word wont appear again in the list
71                  if (frequency!=0){//this excludes the first time when the frequency
                         ↪ is 0 and the prev is null
72                      map.put(prev,frequency);//puts the previous word in the map
                             ↪ because it wont appear again
73                  }
74                  frequency = 1;//sets the frequency back to 1 being as it is a new word
75              }
76              prev = word;//set the previous word to the current word for the next word
77          }
78          map.put(prev,frequency);//after all the words are done the last word still
                 ↪ isnt added as it only adds when the word changes therefore this line
                 ↪ adds the last word alphabetically to the map
79          return map;
80      }
81      /**
82       * 4. write the words and associated frequency to file.
83       *
84       * String builds the string to write to the file with a new line for each word
85       * Writes the string to the file specified
86       * @param map the dictionary with frequencies
87       * @param fileToWriteTo the file to save to
88       */
89      public static void saveToFile(HashMap<String, Number> map, String fileToWriteTo){
90          StringBuilder str = new StringBuilder();
91          for (String key : map.keySet()) {//goes through the map
92              str.append(key).append(",").append(map.get(key)).append("\n");//builds
                     ↪ the string
93          }
94          try {//tries to write the string to the file
95              FileWriter fileWriter = new FileWriter(fileToWriteTo);
96              fileWriter.write(str.toString());
97              fileWriter.close();
98          } catch (IOException e) {
99              e.printStackTrace();
100         }
101     }
102
103     /**
104      * Test Harness
105      *
```

```
106       * @param args
107       * @throws Exception
108       */
109      public static void main(String[] args) throws Exception {
110          System.out.println("Testing Part 1");
111          DictionaryFinder df=new DictionaryFinder();
112  //      1. read text document into a list of strings;
113          ArrayList<String> in= df.readWordsFromCSV("TextFiles\\testDocument.csv");
114  //      2. form a set of words that exist in the document and count the number of
     ↪ times each word
115  //      occurs in a method called FormDictionary;
116  //      3. sort the words alphabetically;
117          LinkedHashMap<String,Number> map = df.formDictionary(in);
118          for (Map.Entry<String,Number> entry : map.entrySet()) {//through the map
119              System.out.println(entry.getKey()+ "=" + entry.getValue());
120          }
121  //      4. write the words and associated frequency to file.
122          df.saveToFile(map,"TextFiles\\Results\\mytestDictionary.csv");
123      }
124  }
```

Listing 2: Trie.java

```java
package DSA2;

import java.util.*;

public class Trie {
    private TrieNode root;//root of the trie

    /**
     * Test harness
     * @param args
     */
    public static void main(String[] args) {
        System.out.println("Testing Part 2");
//        1. Define a TrieNode data structure and class that contains a list of offspring and a flag
//        to indicate whether the node represents a complete word or not. Offspring should be
//        stored in an array of fixed size 26 and the char values of the characters in the trie used as
//        the index. So, for example, the letter 'a' is represented by the position 0 in the offspring
//        array. Hence, the root node for the trie shown in Figure 1 would contain a TrieNode array
//        of size 26 with all null values except in positions 1 ('b') and 2 ('c').
//        2. Define a Trie data structure and class with a TrieNode as a root.
        Trie t = new Trie();
        t.add("bat");
        t.add("cheese");
        t.add("cheers");
        t.add("chat");
        t.add("cat");
        System.out.println("Added: bat,cheese,cheers,chat,cat");
        System.out.println("Going to try and add \"Cat\"");
        System.out.println(t.add("Cat"));
        System.out.println("Breath First Search: " + t.outputBreadthFirstSearch());
        System.out.println("Depth First Search: " + t.outputDepthFirstSearch());
        System.out.println("All words: " + t.getAllWords().toString());
        System.out.println("Creating sub trie of prefix \"ch\"");
        Trie sub = t.getSubTrie("ch");
        System.out.println("All words from sub trie: " +
            sub.getAllWords().toString());
    }

    /**
     * Constructor
     * For a trie that doesnt have any nodes on it already
     * Creates a TrieNode that is the root of the trie
     */
    public Trie(){
        root = new TrieNode(' ');
    }
    /**
     * Constructor
     * For a sub trie
     * @param root the node that needs to be the root of the trie
     */
    public Trie(TrieNode root){
        this.root = root;
    }
```

```
54
55      /**
56       * 1. boolean add(String key): adds a key to the trie, creating any nodes
             ↪ required and
57       * returns true if add was successful (i.e. returns false if key is already in
             ↪ the trie, true otherwise).
58       *
59       * Checks if the word is already in the trie
60       * For each character in the word
61       * Create child character if needed
62       * Then look at the child node
63       *
64       * @param key the word to add to the trie
65       * @return true if it was already in the trie
66       */
67      public boolean add(String key) {
68          if (!contains(key)){
69              TrieNode current = root;
70              key = key.toLowerCase();
71              for (int i = 0; i < key.length(); i++) {
72                  if (!current.isChild(key.charAt(i))){
73                      current.addChild(key.charAt(i));
74                  }
75                  current = current.getChildNode(key.charAt(i));
76              }
77              current.setWordEnd(true);
78              return true;
79          }
80          return false;
81      }
82
83      /**
84       * 2. boolean contains(String key): returns true if the word passed is in the
             ↪ trie as a
85       * whole word, not just as a prefix
86       *
87       * For each characters in the key
88       * Try to get the child of the next character
89       * If it is null return false right away
90       * If all the charaters are there return if the last charater is an end of the
             ↪ word
91       *
92       * @param key The word being looked for
93       * @return true if the word is in the trie, false if not
94       */
95      public boolean contains(String key){
96          TrieNode current = root;//start at the root
97          key=key.toLowerCase();
98          for (char c:key.toCharArray()) {//goes through each character
99              current = current.getChildNode(c);//attempt to ge the child node
100             if (current == null) {//is the child null
101                 return false;
102             }
103         }
104         return current.getIsWordEnd();//if all the characters are there
105     }
106
107     /**
108      * 3. String outputBreadthFirstSearch(): returns a string representing a breadth
             ↪ first
```

```
109      * traversal.
110      *
111      * Get the children nodes of the root
112      * Add the children to the queue
113      * Take off the queue then look at the children of that node and add them to the
             ↪ queue
114      * Keep going until the queue is empty
115      *
116      * @return returns the letters in breath first order
117      */
118     public String outputBreadthFirstSearch(){
119         StringBuilder str = new StringBuilder();//make a string builder to return
120         Queue q = new LinkedList<TrieNode>();//make a queue with a linked list
121         addChildNodeToQueue(root,q);//adds the child nodes of the root to the queue
122         while(q.size()>0){//while there is still nodes in the queue
123             TrieNode current = (TrieNode) q.peek();//get first element
124             q.remove();//remove the first item in the queue
125             q = addChildNodeToQueue(current,q);//add all the child nodes of the
                    ↪ current element to the queue
126             str.append(current.getLetter());//add the current letter value to the
                    ↪ output string
127         }
128         return str.toString();
129     }
130     /**
131      * Helper function for breadth first search
132      *
133      * Gets the children of the node
134      * For each child
135      * If the value of the child is not null, add it to the queue
136      *
137      * @param node node whose children need adding
138      * @param q queue to add them too
139      * @return the queue with the child nodes added
140      */
141     private Queue addChildNodeToQueue(TrieNode node, Queue q){
142         TrieNode[] children = node.getChildren();//get all the children of the node
143         for (TrieNode child:children) {//for each child
144             if (child!=null){//if the child isnt null
145                 q.add(child);//add the child to the queue
146             }
147         }
148         return q;
149     }
150
151     /**
152      * 4. String outputDepthFirstSearch(): returns a string representing a pre order
             ↪ depth
153      * first traversal.
154      *
155      * Appends all the strings of the results of all the children together
156      *
157      * @return output of depth first traversal
158      */
159     public String outputDepthFirstSearch(){
160         StringBuilder str = new StringBuilder();//string builder to append to
161         str.append(depthRecursive(root));
162         return str.toString();
163     }
164     /**
```

```
165         * Helper function for depth first
166         *
167         * Adds the current letter to the string to return
168         * Gets the children of the node
169         * For each child of the node
170         * Append onto the return string the recursive function call on each child
171         *
172         * @param node Current node
173         * @return string of depth first search
174         */
175        private String depthRecursive(TrieNode node){
176            StringBuilder str = new StringBuilder();//create a new string builder
177            str.append(node.getLetter());//add the node value first
178            TrieNode[] children = node.getChildren();//get the children
179            for (TrieNode child:children) {//for each child node
180                if (child!=null){//if the child exists
181                    str.append(depthRecursive(child));//recursivly call the function
182                }
183            }
184            return str.toString();
185        }
186
187        /**
188         * 5. Trie getSubTrie(String prefix): returns a new trie rooted at the prefix, or
               ↪ null if
189         * the prefix is not present in this trie
190         *
191         * Traverse down the trie until the end of the prefix
192         * Create a new trie with the root being the current node
193         *
194         * @param prefix prefix of the new sub trie
195         * @return sub trie of the prefix
196         */
197        public Trie getSubTrie(String prefix){
198            TrieNode current = root;//start at the root of the trie
199            for (int i = 0; i < prefix.length(); i++) {//for each letter in the prefix
200                char c = prefix.charAt(i);//get the character
201                TrieNode node = current.getChildNode(c);//get the child node of the
                       ↪ character
202                if (node == null) {//if at any point the trie doesnt contain the prefix
                       ↪ it returns null
203                    return null;
204                }
205                current = node;//set the current node to the child node
206            }
207            return new Trie(current);//creates a new trie with the root being the current
                   ↪ node
208        }
209
210        /**
211         * 6. List getAllWords(): returns a list containing all words in the trie.
212         *
213         * Check if the root is a end of word
214         * Add all the root children to the stack (actually a Linked List)
215         * While there is still nodes to process
216         * Get the next node to process off the stack
217         * Add all the children of the current node onto the stack
218         * Check if the top node is done with, if it is pop it and check the new top one
219         * repeat till it isnt done with
220         * This will be the prefix of the next word
```

```
221         * If it is the end of a word
222         * Add the word to the list of all the words
223         *
224         * @return List containing all the words in the trie
225         */
226        public List<String> getAllWords(){
227            List words = new ArrayList();//create new list
228            if (root.getIsWordEnd()){//This is so that when a prefix of a sub-trie is a
                   ↪ whole word in the trie that word isnt missed out
229                words.add("");
230            }
231            LinkedList<TrieNode> todo = new LinkedList<>();//create a new linked list
                   ↪ which im using as a stack
232            todo = addAllChildNodesToStack(root,todo);//add all the child nodes of the
                   ↪ root to the stack
233            ArrayList<TrieNode> fullyProcessedNodes = new ArrayList<>();//create an array
                   ↪ list of all the done nodes
234            LinkedList<TrieNode> currentWord = new LinkedList<>();//current word is also
                   ↪ a stack
235            while(todo.size()>0){//while there is still nodes to process
236                TrieNode current = todo.peek();//peek at the top node on the stack to
                       ↪ process
237                todo.remove();//remove the top node
238                todo = addAllChildNodesToStack(current,todo);//add all the children of
                       ↪ the current node to the stack
239                //calculates the word by removing the correct letters from the word stack
240                while (currentWord.size()>0){//while the current word still has
                       ↪ characters in
241                    TrieNode top = currentWord.peek();//peek at the top one
242                    if (isNodeDoneWith(top,fullyProcessedNodes)){//if all the children
                           ↪ have been done with or it has no children
243                        fullyProcessedNodes.add(top);//add to processed nodes
244                        currentWord.pop();//remove letter from the word stack
245                    } else {
246                        break;//ends the while if the top letter is still valid and needs
                               ↪ to stay there
247                    }
248                }
249                //adds word to the all words list if it is the end of a word
250                currentWord.push(current);//push the current letter onto the stack
251                if (current.getIsWordEnd()){//if its an end of a word
252                    StringBuilder newWord = new StringBuilder();//creates a new string
                           ↪ builder
253                    for (TrieNode node : currentWord) {//for all the nodes in the current
                           ↪ word stack
254                        newWord.append(node.getLetter());//append to the word string the
                               ↪ node letter
255                    }
256                    words.add(newWord.reverse().toString());//add the new word string to
                           ↪ the list
257                }
258            }
259            return words;
260        }
261        /**
262         * Helper function to get all words
263         *
264         * Get all the children
265         * Adds all the children of the node to the stack
266         *
```

```
267         * @param node node to add all the children
268         * @param stk stack to add all the children onto
269         * @return the new stack
270         */
271        private LinkedList<TrieNode> addAllChildNodesToStack(TrieNode node,
               ↪ LinkedList<TrieNode> stk){
272            TrieNode[] children = node.getChildren();//get the children
273            for (TrieNode child:children){//for each child
274                if (child!=null){//if the child isnt null
275                    stk.push(child);//push child onto the stack
276                }
277            }
278            return stk;
279        }
280        /**
281         * Helper function to get all words
282         *
283         * Check if the node has children
284         * If it does, for each child
285         * If there is a child that isnt in the fully processed nodes array,
286         * then the node cannot be fully processed so return false
287         *
288         * @param node the node to work out if its done with
289         * @param fullyProcessedNodes the arraylist of fully processed nodes
290         * @return true if the node is done with in the word, false if not
291         */
292        private boolean isNodeDoneWith(TrieNode node, ArrayList<TrieNode>
               ↪ fullyProcessedNodes){
293            if (node.hasChildren()){//the top in current word has children
294                TrieNode[] topChildren = node.getChildren();//get the children
295                for (TrieNode topChild : topChildren) {//for all of the top nodes children
296                    if (topChild != null) {
297                        if (!fullyProcessedNodes.contains(topChild)) {//if fully
                               ↪ processed nodes doesnt contain a child of top
298                            return false;//we are not done with the top node
299                        }
300                    }
301                }
302            }
303            return true;
304        }
305    }
```

Listing 3: TrieNode.java

```
1  package DSA2;
2
3  public class TrieNode {
4      private TrieNode[] children;//the children
5      private char letter;//value of the node
6      private boolean isWordEnd;//boolean to tell if it is the end of the word
7
8      /**
9       * Constructor
10      * @param letter value of the node
11      */
12     public TrieNode(char letter) {
13         this.children = new TrieNode[26];//constructs a new array of trie 26 trie
               ↪ nodes where all are set to null
14         this.letter=letter;
15         this.isWordEnd=false;
16     }
17     //Accessors
18     /**
19      * Gets the children array
20      *
21      * @return children array
22      */
23     public TrieNode[] getChildren(){
24         return children;
25     }
26     /**
27      * Gets value of the node
28      *
29      * @return the letter value
30      */
31     public char getLetter() {
32         return letter;
33     }
34     /**
35      * Get end of word boolean
36      *
37      * @return return if the node is an end of a word
38      */
39     public boolean getIsWordEnd(){
40         return isWordEnd;
41     }
42     /**
43      * Null if no child node of that letter
44      *
45      * @param letter the letter value of the child node
46      * @return TrieNode of the child node of letter
47      */
48     public TrieNode getChildNode(char letter){
49         return children[(int)letter-'a'];
50     }
51     //Modifiers
52     /**
53      * Set the is word End
54      *
55      * @param isWordEnd what to set the isWordEnd to
56      */
57     public void setWordEnd(boolean isWordEnd){
58         this.isWordEnd=isWordEnd;
```

```
59      }
60      /**
61       * Add a child
62       *
63       * Create a new node
64       * @param letter child letter
65       */
66      public void addChild(char letter){
67          this.children[(int)letter-'a'] = new TrieNode(letter);
68      }
69      //Methods
70      /**
71       * Checks if a letter is a child
72       *
73       * @param letter letter to test
74       * @return true if letter is a child
75       */
76      public boolean isChild(char letter){
77          return (getChildNode(letter)!=null);
78      }
79      /**
80       * Determines if this node has any children
81       *
82       * For each letter in the alphabet
83       * If the children array is not null at that letter
84       * Return True
85       *
86       * @return true if there is any children
87       */
88      public Boolean hasChildren(){
89          for(int i = 0;i<26;i++){
90              if (this.children[i] != null){
91                  return true;
92              }
93          }
94          return false;
95      }
96  }
```

Listing 4: AutoCompletionTrie.java

```
1  package DSA2;
2
3  import java.io.FileNotFoundException;
4  import java.util.*;
5
6  public class AutoCompletionTrie{
7      private AutoCompletionTrieNode root;//root of the trie
8      /**
9       * Test harness
10      * @param args
11      */
12     public static void main(String[] args) {
13         System.out.println("Testing Part 3");
14         //1. Load all the queries file called queries.csv from the project directory.
15         ArrayList<String> prefixes = new ArrayList<>();
16         try {
17             prefixes =
                   ↪ DictionaryFinder.readWordsFromCSV("TextFiles\\testQueries.csv");//read
                   ↪ in the prefixes
18         } catch (FileNotFoundException e) {
19             e.printStackTrace();
20         }
21         ArrayList<String> wordsAndFrequencies = new ArrayList<>();
22         try {
23             wordsAndFrequencies =
                   ↪ DictionaryFinder.readWordsFromCSV("TextFiles\\Results\\mytestDictionary.cs
                   ↪ in the prefixes
24         } catch (FileNotFoundException e) {
25             e.printStackTrace();
26         }
27         HashMap<String,Number> dict = new LinkedHashMap<>();
28         dict.put((wordsAndFrequencies.get(0)),Integer.parseInt((wordsAndFrequencies.get(1).s
29         for (int i = 1;i < wordsAndFrequencies.size() - 1;i++){
30             dict.put((wordsAndFrequencies.get(i).split("\n"))[1],Integer.parseInt((wordsAndF
31         }
32         AutoCompletionTrie t = new AutoCompletionTrie(dict);
33         LinkedHashMap<String,Number> pro = new LinkedHashMap<>();
34         for (String pre:prefixes){//for each prefix
35             System.out.println("For: " + pre);
36             //2. For each query, find the best three matches (at most) with the most
                   ↪ likely first, and with
37             //associated estimated probability of correctness. If words have equal
                   ↪ probability, choose
38             //the first occurring word as determined by a breadth first search.
                   ↪ Probabilities should be
39             //calculated from the frequencies (see example below).
40             pro.putAll(t.getTopThreeProbability(pre));
41             DictionaryFinder.saveToFile(pro,"TextFiles\\Results\\myTestMatches.csv");//3.
                   ↪ Write the results into a file called matches.csv in exactly the
                   ↪ specified format.
42         }
43     }
44
45     /**
46      * Constructor
47      *
48      * Sets the dictionary, prefix and trie
49      *
50      * @param dict dictionary with frequencies
```

```
51       */
52      public AutoCompletionTrie(HashMap<String,Number> dict){
53          root = new AutoCompletionTrieNode(' ',null);
54          for (Map.Entry<String, Number> entry : dict.entrySet()) {//descend through
                ↪ the treemap
55              this.add(entry.getKey(), (Integer) entry.getValue());
56          }
57      }
58
59      //Trie Clone
60      /**
61       * Constructor
62       *
63       * For a trie that doesnt have any nodes on it already
64       * Creates a TrieNode that is the root of the trie
65       */
66      public AutoCompletionTrie(){
67          root = new AutoCompletionTrieNode(' ',null);
68      }
69      /**
70       * Constructor
71       * For a sub trie
72       * @param root the node that needs to be the root of the trie
73       */
74      public AutoCompletionTrie(AutoCompletionTrieNode root){
75          this.root = root;
76      }
77      /**
78       * ALtered to take a quantity of word aswell as a word
79       *
80       * Checks if the word is already in the trie
81       * For each character in the word
82       * Create child character if needed
83       * Then look at the child node
84       * When it gets to the end of the word set the node at the end to have the
             ↪ quantity of words finishing there
85       *
86       * @param key the word to add to the trie
87       * @return true if it was already in the trie
88       */
89      public boolean add(String key,int quantity) {
90          if (!contains(key)){
91              AutoCompletionTrieNode current = root;
92              key = key.toLowerCase();
93              for (int i = 0; i < key.length(); i++) {
94                  if (!current.isChild(key.charAt(i))){
95                      current.addChild(key.charAt(i));
96                  }
97                  current = current.getChildNode(key.charAt(i));
98              }
99              current.alterWordsEnding(quantity);
100             return true;
101         }
102         return false;
103     }
104     /**
105      * 2. boolean contains(String key): returns true if the word passed is in the
             ↪ trie as a
106      * whole word, not just as a prefix
107      *
```

```
108         * For each characters in the key
109         * Try to get the child of the next character
110         * If it is null return false right away
111         * If all the charaters are there return if the last charater is an end of the
                ↪ word
112         *
113         * @param key The word being looked for
114         * @return true if the word is in the trie, false if not
115         */
116        public boolean contains(String key){
117            AutoCompletionTrieNode current = root;//start at the root
118            key=key.toLowerCase();
119            for (char c:key.toCharArray()) {//goes through each character
120                current = current.getChildNode(c);//attempt to ge the child node
121                if (current == null) {//is the child null
122                    return false;
123                }
124            }
125            return current.getQuantityOfWordsEnding()>0;//if all the characters are there
126        }
127        /**
128         * 3. String outputBreadthFirstSearch(): returns a string representing a breadth
                ↪ first
129         * traversal.
130         *
131         * Get the children nodes of the root
132         * Add the children to the queue
133         * Take off the queue then look at the children of that node and add them to the
                ↪ queue
134         * Keep going until the queue is empty
135         *
136         * @return returns the letters in breath first order
137         */
138        public String outputBreadthFirstSearch(){
139            StringBuilder str = new StringBuilder();//make a string builder to return
140            Queue q = new LinkedList<AutoCompletionTrieNode>();//make a queue with a
                   ↪ linked list
141            addChildNodeToQueue(root,q);//adds the child nodes of the root to the queue
142            while(q.size()>0){//while there is still nodes in the queue
143                AutoCompletionTrieNode current = (AutoCompletionTrieNode) q.peek();//get
                       ↪ first element
144                q.remove();//remove the first item in the queue
145                q = addChildNodeToQueue(current,q);//add all the child nodes of the
                       ↪ current element to the queue
146                str.append(current.getLetter());//add the current letter value to the
                       ↪ output string
147            }
148            return str.toString();
149        }
150        /**
151         * Helper function for breadth first search
152         *
153         * Gets the children of the node
154         * For each child
155         * If the value of the child is not null, add it to the queue
156         *
157         * @param node node whose children need adding
158         * @param q queue to add them too
159         * @return the queue with the child nodes added
160         */
```

```
161     private Queue addChildNodeToQueue(AutoCompletionTrieNode node, Queue q){
162         AutoCompletionTrieNode[] children = node.getChildren();//get all the children
                ↪ of the node
163         for (AutoCompletionTrieNode child:children) {//for each child
164             if (child!=null){//if the child isnt null
165                 q.add(child);//add the child to the queue
166             }
167         }
168         return q;
169     }
170     /**
171      * 4. String outputDepthFirstSearch(): returns a string representing a pre order
                ↪ depth
172      * first traversal.
173      *
174      * Appends all the strings of the results of all the children together
175      *
176      * @return output of depth first traversal
177      */
178     public String outputDepthFirstSearch(){
179         StringBuilder str = new StringBuilder();//string builder to append to
180         str.append(depthRecursive(root));
181         return str.toString();
182     }
183     /**
184      * Helper function for depth first
185      *
186      * Adds the current letter to the string to return
187      * Gets the children of the node
188      * For each child of the node
189      * Append onto the return string the recursive function call on each child
190      *
191      * @param node Current node
192      * @return string of depth first search
193      */
194     private String depthRecursive(AutoCompletionTrieNode node){
195         StringBuilder str = new StringBuilder();//create a new string builder
196         str.append(node.getLetter());//add the node value first
197         AutoCompletionTrieNode[] children = node.getChildren();//get the children
198         for (AutoCompletionTrieNode child:children) {//for each child node
199             if (child!=null){//if the child exists
200                 str.append(depthRecursive(child));//recursivly call the function
201             }
202         }
203         return str.toString();
204     }
205     /**
206      * 5. Trie getSubTrie(String prefix): returns a new trie rooted at the prefix, or
                ↪ null if
207      * the prefix is not present in this trie
208      *
209      * Traverse down the trie until the end of the prefix
210      * Create a new trie with the root being the current node
211      *
212      * @param prefix prefix of the new sub trie
213      * @return sub trie of the prefix
214      */
215     public AutoCompletionTrie getSubTrie(String prefix){
216         AutoCompletionTrieNode current = root;//start at the root of the trie
217         for (int i = 0; i < prefix.length(); i++) {//for each letter in the prefix
```

```
218             char c = prefix.charAt(i);//get the character
219             AutoCompletionTrieNode node = current.getChildNode(c);//get the child
                    ↪ node of the character
220             if (node == null) {//if at any point the trie doesnt contain the prefix
                    ↪ it returns null
221                 return null;
222             }
223             current = node;//set the current node to the child node
224         }
225         return new AutoCompletionTrie(current);//creates a new trie with the root
                ↪ being the current node
226     }
227     /**
228      * Modified to return the fequency of the word too
229      * Also is a linked hashmap os the order matters
230      * This is a breadth first search to determine the order for any words that
             ↪ occurs equal number of times
231      *
232      * Check if the root is a end of word
233      * Add all the root children to the queue (actually a Linked List)
234      * While there is still nodes to process
235      * Get the next node to process out of the queue
236      * Add all the children of the current node into the queue
237      * If it is an end of a word
238      * Work out what the word is by using the parents of it
239      * Add the word and the frequencies to the hashmap
240      *
241      * @return Linked hashmap containing all the words in the trie with their
             ↪ frequencies
242      */
243     public LinkedHashMap<String,Integer> getAllWords(){
244         LinkedHashMap<String,Integer> words = new LinkedHashMap<>();//create new list
245         if (root.getQuantityOfWordsEnding()>0){//This is so that when a prefix of a
                ↪ sub-trie is a whole word in the trie that word isnt missed out
246             words.put("",root.getQuantityOfWordsEnding());
247         }
248         Queue q = new LinkedList<AutoCompletionTrieNode>();//make a queue with a
                ↪ linked list
249         addChildNodeToQueue(root,q);//adds the child nodes of the root to the queue
250         while(q.size()>0){//while there is still nodes in the queue
251             AutoCompletionTrieNode current = (AutoCompletionTrieNode) q.peek();//get
                    ↪ first element
252             q.remove();//remove the first item in the queue
253             q = addChildNodeToQueue(current,q);//add all the child nodes of the
                    ↪ current element to the queue
254             if (current.getQuantityOfWordsEnding()>0){
255                 StringBuilder sb = new
                        ↪ StringBuilder(getWordFromNode(current));//builds the word from
                        ↪ end to start
256                 words.put(sb.reverse().toString(),current.getQuantityOfWordsEnding());//reve
                        ↪ the string builder and adds it to the map
257             }
258         }
259         return words;
260     }
261     /**
262      * Helper function to get all words
263      *
264      * If the node is not the root (the character space)
265      * Add the node letter value to the start of the string
```

```
266         * Call recursivly on the current nodes parent
267         *
268         * @param node current node
269         * @return the word as a string in reverse backtracking up through the parents
270         */
271        private String getWordFromNode(AutoCompletionTrieNode node){
272            String str = "";//new string to return
273            if (node.getLetter()!=' '){//if node isnt the root node
274                str += node.getLetter() + getWordFromNode(node.getParentNode());//add
                       ↪ letter to string plus recursive call on the parant node
275            }
276            return str;
277        }
278
279        //Additional
280        /**
281         * Gets all probabilities of all the words from the prefix
282         *
283         * Get the sub trie of the prefix
284         * Set the total number of words equal to the root of the sub trie quantity (for
                ↪ words that are the prefix)
285         * Add all the root nodes to the queue to process
286         * While there is still nodes to process
287         * Get the first node in the queue
288         * Add all of its children to the queue
289         * Add the number of words finishing on that node to the total
290         * Get a linked hashmap of all the words and frequencies with getAllWords function
291         * Go through the map
292         * If the entry is "" add the prefix and the frequency divided by the total to
                ↪ the Linked hashmap to return
293         * else just add the key entry and the frequency divided by the total for it to
                ↪ the linked hashmap
294         *
295         * @param prefix the prefix to get all the probabilities from
296         * @return the linked hashmap of the probability and word
297         */
298        private LinkedHashMap<String,Number> getAllProbabilites(String prefix){
299            LinkedHashMap<String,Number> r = new LinkedHashMap<>();//create new hashmap
                   ↪ to return
300            //getting subtrie
301            AutoCompletionTrie subTrie = this.getSubTrie(prefix);//gets the list of words
                   ↪ after the prefix that are in the subtrie
302            if (subTrie!=null){
303                //getting total number of words in sub trie
304                int total = subTrie.root.getQuantityOfWordsEnding();//set the total to
                       ↪ the sub trie root node quantity of words ending
305                Queue q = new LinkedList<AutoCompletionTrieNode>();//make a queue with a
                       ↪ linked list
306                addChildNodeToQueue(subTrie.root,q);//adds the child nodes of the root to
                       ↪ the queue
307                while(q.size()>0){//while there is still nodes in the queue
308                    AutoCompletionTrieNode current = (AutoCompletionTrieNode)
                           ↪ q.peek();//get first element
309                    q.remove();//remove the first item in the queue
310                    q = addChildNodeToQueue(current,q);//add all the child nodes of the
                           ↪ current element to the queue
311                    total += current.getQuantityOfWordsEnding();//add the number of words
                           ↪ ending there to the total
312                }
313                //calculating the probabilities
```

```
314             LinkedHashMap<String,Integer> allWords = subTrie.getAllWords();//gets all
                    ↪ the words and frequencies
315             for (Map.Entry<String,Integer> entry : allWords.entrySet()) {//through
                    ↪ the map
316                 if (entry.getKey().equals("")){//if the prefix was a word
317                     r.put(prefix,(double) entry.getValue()/total);//put the prefix
                            ↪ and frequencies divided by total in map
318                 } else {
319                     r.put(entry.getKey(),(double) entry.getValue()/total);//put the
                            ↪ word and the frequencies difvided by the total in the map
320                 }
321             }
322         }
323         return r;
324     }
325     /**
326      * Get the three most likely words from a given prefix
327      *
328      * Get all the probabilities for the prefix
329      * While the map of all the words still has entrys
330      * and 3 havent yet been put in
331      * Check for the first highest one
332      * Only change the highest one if the next one is greater than any previous (not
                ↪ greater than or equal to)
333      * Print out the highest one that round
334      * Add it to the map to return
335      * Remove that element from the all words map
336      *
337      * @param prefix prefix of the word
338      * @return linked hash map of the top three probabilities
339      */
340     public LinkedHashMap<String,Number> getTopThreeProbability(String prefix){
341         LinkedHashMap<String,Number> r = new LinkedHashMap<>();//create new hashmap
                ↪ to return
342         LinkedHashMap<String,Number> all = getAllProbabilites(prefix);//get all the
                ↪ probabilies
343         int i = 0;
344         while (all.size()>0&&i++<3){//while there is still entrys left to go through
                ↪ and less than three entries have been put in
345             double highestProbability = 0.0;
346             String highestKey = "";
347             for (String key : all.keySet()) {//goes through the map of words and
                    ↪ probabilities
348                 if ((double) all.get(key)>highestProbability){//if the probability of
                        ↪ the current word is greater than the current highest probability
349                     highestProbability=(double) all.get(key);//set the highest
                            ↪ probability to the current map
350                     highestKey = key;//set the highest key too
351                 }
352             }
353             r.put(highestKey,highestProbability);//put the highest one in the map to
                    ↪ return
354             System.out.println(highestKey + "=" + highestProbability);//print out the
                    ↪ highest word and the highest probability
355             all.remove(highestKey);//remove the highest one from the map of all words
356         }
357         return r;
358     }
359 }
360
```

```
361 class AutoCompletionTrieNode {
362     private AutoCompletionTrieNode[] children;//the children
363     private char letter;//value of the node
364     private int quantityOfWordsEnding;
365     private AutoCompletionTrieNode parent;
366 
367     /**
368      * Constructor
369      *
370      * @param parent the parent node
371      * @param letter value of the node
372      */
373     public AutoCompletionTrieNode(char letter,AutoCompletionTrieNode parent) {
374         this.children = new AutoCompletionTrieNode[26];//constructs a new array of
              ↪ trie 26 trie nodes where all are set to null
375         this.letter=letter;//sets the letter value
376         this.quantityOfWordsEnding =0;//sets the number of words ending on this node
              ↪ to 0
377         this.parent = parent;//sets the parent to the parent passed in
378     }
379     //Accessors
380     /**
381      * Gets the children array
382      *
383      * @return children array
384      */
385     public AutoCompletionTrieNode[] getChildren(){
386         return children;
387     }
388     /**
389      * Gets value of the node
390      *
391      * @return the letter value
392      */
393     public char getLetter() {
394         return letter;
395     }
396     /**
397      * Get quantity
398      *
399      * @return number of words that end at this node
400      */
401     public int getQuantityOfWordsEnding(){
402         return quantityOfWordsEnding;
403     }
404     /**
405      * Null if no child node of that letter
406      *
407      * @param letter the letter value of the child node
408      * @return TrieNode of the child node of letter
409      */
410     public AutoCompletionTrieNode getChildNode(char letter){
411         return children[(int)letter-'a'];
412     }
413     /**
414      * Gets the parent node
415      *
416      * @return the parent node
417      */
418     public AutoCompletionTrieNode getParentNode(){
```

```
419         return parent;
420     }
421     //Modifiers
422     /**
423      * Add a child
424      *
425      * Create a new node in the appropreate place in the children array
426      *
427      * @param letter child letter
428      */
429     public void addChild(char letter){
430         this.children[(int)letter-'a'] = new AutoCompletionTrieNode(letter,this);
431     }
432     /**
433      * Change the number of words finishing at this node
434      *
435      * @param i number of additional words ending at this node
436      */
437     public void alterWordsEnding(int i){
438         quantityOfWordsEnding += i;
439     }
440     //Methods
441     /**
442      * Checks if a letter is a child
443      *
444      * @param letter letter to test
445      * @return true if letter is a child
446      */
447     public boolean isChild(char letter){
448         return (getChildNode(letter)!=null);
449     }
450 }
```

ADD ANY OTHER CODE HERE

Listing 5: Main.java

```
package DSA2;

import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;

public class Main {
    public static void main(String[] args) {
        System.out.println("Task");
        //1. form a dictionary file of words and counts from the file lotr.csv
        DictionaryFinder df=new DictionaryFinder();
        ArrayList<String> in= new ArrayList<>();
        try {
            in = DictionaryFinder.readWordsFromCSV("TextFiles\\lotr.csv");//reads
                ↪ from lotr into the arraylist in
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        LinkedHashMap<String,Number> wordsAndFreq = df.formDictionary(in); //form a
            ↪ dictionary with frequencies from the text file
        df.saveToFile(wordsAndFreq,"TextFiles\\Results\\myGollem.csv");//save the
            ↪ dictionary with frequencies
        //2. construct a trie from the dictionary using your solution from part 2
        AutoCompletionTrie t = new AutoCompletionTrie(wordsAndFreq);
        //3. load the prefixs from lotrQueries.csv
        ArrayList<String> prefixLine = new ArrayList<>();
        try {
            prefixLine =
                ↪ DictionaryFinder.readWordsFromCSV("TextFiles\\lotrQueries.csv");//read
                ↪ in the prefixes
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        String prefixes[] = prefixLine.get(0).split("\\r?\\n");
        //4. for each prefix query
        LinkedHashMap<String,Number> pro = new LinkedHashMap<>();
        for (String pre:prefixes){//for each prefix
            System.out.println("For: " + pre);
            AutoCompletionTrie subT = t.getSubTrie(pre);
            subT.getAllWords();//4.1. Recover all words matching the prefix from the
                ↪ trie.
            AutoCompletionTrie ACT = new AutoCompletionTrie(wordsAndFreq);//create
                ↪ new object
            pro.putAll(ACT.getTopThreeProbability(pre));//4.2. Choose the three most
                ↪ frequent words and display to standard output.
            DictionaryFinder.saveToFile(pro,"TextFiles\\Results\\mylotrMatches.csv");//4.3.
                ↪ Write the results to lotrMatches.csv
        }
    }
}
```