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

Student number: 100250071. Blackboard ID: fxg18asu

Tuesday 12th May, 2020 12:21

Contents

1 Part 1: Form a Dictionary and Word Frequency Count

This reads in a document of words, finds the set of unique words to form a dictionary, counts the occurrence of each word, then saves the words and counts to file;

1.1 formDictionary

Algorithm 1 formDictionary(*in*) Loads a treemap with all the words used and the amount they are used

Require: An array, *in*. This is an array of strings from the file
Ensure: A TreeMap, *dictionary*. This is a dictionary storing all the different words with no duplicates and the amount of times each word appears.

```
1: for every string in in do
2:     if dictionary_string = NULL then
3:         dictionary_string ←string, 1
4:     else
5:         dictionary_string ← dictionary_string + (string, 1)
   return dictionary
```

1.2 saveToFile

Algorithm 2 saveToFile(*File*, *dictionary*) Saves the treemap previously generated to a txt file defined from the inputs

Require: A string, *File*. This stores the txt file to write to. Also *dictionary*, this stores all the words and the amount they've been used in a TreeMap. Treemaps are automatically sorted into order of their key.

```
1: for every string in dictionary do
2:     File ← File + dictionary_string + dictionary_string(amount)
```

2 Part 2: Implement a Trie Data Structure

A trie data structure for storing a prefix and methods to manipulating it

2.1 add

Algorithm 3 add(*string*) Adds a word to the Trie

Require: A character array, *key* of length *length*

```
1:  nullVisited ← FALSE
2:  search ← root
3:  next ← null
4:  for i ← 0 to length do
5:      if nullVisited = FALSE then
6:          next ← search.getOffspring
7:          if next = null then
8:              next ← new TrieNode(key_i)
9:              temp.setOffspring(key_i)
10:             nullVisited ← TRUE
11:         temp ← next
12:     else
13:         next ← new TrieNode(key_i
14:         temp.setOffspring(key_i)
15:         temp ← next
16: temp.setIsKey ← TRUE
```

2.2 contains

Algorithm 4 contains(*string*) Checks to see if a word is in the Trie

Require: A character array, *string* of length *length*

```
1: temp ← root
2: for i ← 0 to length do
3:     temp ← temp.getOffspring(string_i)
4:     if temp = null then
5:         return FALSE
6: return TRUE
```

2.3 outputBreadthFirstSearch

Algorithm 5 outputBreadthFirstSearch() Performs a breadth first search of the trie using a queue data structure

Ensure: A string, *result* which is result of the search.

```
1: queue ← queue + root
2: while queue != null do
3:     temp ← queue.remove
4:     if temp != root then
5:         result ← result + temp.getChar
6:     for every node in temp.getAllOffSpring do
7:         if node != null then
8:             queue ← queue + node
9: return result
```

2.4 outputDepthFirstSearch

Algorithm 6 outputDepthFirstSearch() Performs a depth first search of the trie using a stack data structure

Ensure: A string, *result* which is result of the search.

```
1: stack.push(root)
2: while stack != null do
3:     temp ← stack.pop
4:     if temp != root then
5:         result ← result + temp.getChar
6:     for i ← temp.getAllOffspring.length-1 to 0 do
7:         if temp.getOffspring_i != null then
8:             stack.push(temp.getOffspring_i)
9: return result
```

2.5 getSubTrie

Algorithm 7 getSubTrie($prefix$) Makes a new Trie based on the offspring of a node.

Require: A character array, $prefix$ of length $length$

Ensure: A Trie, $subTrie$ which is the subtrie

```
1: temp ← root
2: for i ← 0 to length do
3:     temp ← temp.getOffspring(prefix_i)
4: subTrie ← new Trie
5: subTrie.root ← temp
6: subTrie.root.char ← null
7: subTrie.root.isKey ← FALSE
```

2.6 getAllWords

Algorithm 8 getAllWords($node$, $string$)

Require: A TrieNode, $node$ and a string, $string$

Ensure: A list of strings, $list$ which are the complete words

```
1: if node != root then
2:     word ← word + node.char
3: for every trieNode in node.offspring do
4:     if trieNode != null then
5:         stack.push(trieNode)
6: while stack != null do
7:     list ← list + getAllWords(stack.pop, word)
8: if node.isKey = TRUE then
9:     list ← list + word
10: return list
```

3 Part 3: Word Auto Completion Application

A word auto-completion system using a dictionary and a Trie data Structure

3.1 getLast

This will be added to Trie

Algorithm 9 getLast(*string*). Gets the last node in a word.

Require: A Character array, *string*.

```
1: last ← root
2: for every char in string do
3:     last ← last.getOffspring(char)
4: return last
```

3.2 populateTrie

Algorithm 10 populateTries(*trie*,*dictionary*)

Require: A Trie, *trie* and a HashMap, *dictionary* where *dictionary* links strings to values.

```
1: for every string in dictionary do
2:     trie.add(string)
3:     trie.getLast(string).setFrequency(string_value)
```

3.3 probabadilityWords

Algorithm 11 probabiltyWords($prefix$)

Require: A string, $prefix$. This is the word to find the probabilties for.

```
1: subTrie ← trie.getSubtrie(prefix)
2: words ← subTrie.getAllWords(subTrie.getRoot(), "")
3: for every string in words do
4:     totalWords ← totalWords + string_value
5: for every string in words do
6:     probabiltyWords_probability,word ←
7:     string_value/totalWords, prefix + string
```

3.4 saveToFile

Algorithm 12 saveToFile($file$, *probabiltyWords*, $prefix$) Saves the 3 most probable to a txt file

Require: A string, $File$. This is the name of the txt file to write to. Also *probabilityWords*, this stores all the words and the probability. The treemap will be sorted into descending order so that the most probable words are at the front. Also $prefix$, this is the word that the probabilities have been found for

```
1: if thenprobabiltyWords.length ¿ 3
2:     max ← 3
3: else
4:     max ← probabilityWords.length
5: for i ← 0 to max do
6:     File ← ProbabilityWords_word(i) +
7:     ProbabilityWords_probability(i)
```

4 Code Listing

4.1 Dictionary Finder

Listing 1: DictionaryFinder.java

```
1
2
3  import java.io.BufferedReader;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6  import java.io.FileReader;
7  import java.io.FileWriter;
8  import java.io.IOException;
9  import java.io.PrintWriter;
10 import java.io.StreamTokenizer;
11 import java.util.*;
12
13 /**
14  *
15  * @author ajb
16  */
17 public class DictionaryFinder {
18     private TreeMap<String, Integer> dictionary;
19     /**
20      * All the different strings from the input file
21      */
22     private ArrayList<String> in;
23
24     /**
25      * The classes constructor
26      */
27     public DictionaryFinder(){
28         this.dictionary = new TreeMap<>();
29         this.in = new ArrayList<>();
30     }
31
32     /**
33      * Reads all the words in a comma separated text document into an Array
34      * @param file
35      */
36     public static ArrayList<String> readWordsFromCSV(String file) throws
        ↪ FileNotFoundException {
37         Scanner sc=new Scanner(new File(file));
38         sc.useDelimiter(" |,");
39         ArrayList<String> words=new ArrayList<>();
40         String str;
41         while(sc.hasNext()){
42             str=sc.next();
43             str=str.trim();
44             str=str.toLowerCase();
45             words.add(str);
46         }
47         return words;
48     }
49
50     /**
51      * This adds all the words to the treemap and tallies up the amount each word
        ↪ occurs
52      */
53     public void formDictionary(){
```

```
54          for (String string: in){
55              if (this.dictionary.get(string) == null) {
56                  this.dictionary.put(string, 1);
57              }
58              else {
59                  this.dictionary.put(string, this.dictionary.get(string)+1);
60              }
61          }
62      }
63
64      /**
65       * This saves the results into a file specified and is ordered in alphabetical
             ↪ order
66       * @param file the file to save to
67       * @throws IOException
68       */
69      public void saveToFile(String file) throws IOException{
70          /*
71          https://howtodoinjava.com/sort/java-sort-map-by-key/
72          treemaps are naturally sorted
73           */
74          FileWriter fileWriter = new FileWriter(file);
75          PrintWriter printWriter = new PrintWriter(fileWriter);
76          for(String string: dictionary.keySet()) {
77              printWriter.println(string + ", " + this.dictionary.get(string));
78          }
79          printWriter.close();
80
81      }
82
83      /**
84       * gets the words to be inputted to the treemap
85       * @return The list of words
86       */
87      public ArrayList<String> getIn() {
88          return in;
89      }
90
91      /**
92       * sets the words to be inputted to the treemap
93       * @param in the words to be inputted into the treemap
94       */
95      public void setIn(ArrayList in) {
96          this.in = in;
97      }
98
99      /**
100      * This gets the treemap
101      * @return the treemap
102      */
103     public TreeMap<String, Integer> getDictionary() {
104         return dictionary;
105     }
106
107     /**
108      * Sets the treemap
109      * @param dictionary what the treemap is to be set as
110      */
111     public void setDictionary(TreeMap<String, Integer> dictionary) {
112         this.dictionary = dictionary;
```

```
113     }
114
115     /**
116      * Used for testing
117      * @param args
118      * @throws Exception
119      */
120     public static void main(String[] args) throws Exception {
121         DictionaryFinder df=new DictionaryFinder();
122         ArrayList<String> in=readWordsFromCSV("TextFiles/lotr.csv");
123         df.setIn(in);
124         df.formDictionary();
125         df.saveToFile("TextFiles/testDictionary2.csv");
126     }
127
128 }
```

4.2 Trie

Listing 2: Trie.java

```
1  import java.util.*;
2
3  public class Trie {
4
5      private TrieNode root;
6      public static void main(String[] args) {
7          Trie test = new Trie();
8          test.add("cheers");
9          test.add("cheese");
10         test.add("chat");
11         test.add("cat");
12         test.add("bat");
13         test.add("batch");
14         if (test.contains("chat")) {
15             System.out.println("True");
16         }
17         else {
18             System.out.println("False");
19         }
20         System.out.println("test.outputBreadthFirstSearch() = " +
               ↪ test.outputBreadthFirstSearch());
21         System.out.println("test.outputDepthFirstSearch() = " +
               ↪ test.outputDepthFirstSearch());
22         Trie subTrie = test.getSubTrie("ch");
23         System.out.println("subTrie.root.getOffspring('h').getChar() = " +
               ↪ subTrie.getRoot().getChar());
24         char[] test1 = null;
25
26         List<String> strings = test.getAllWords(test.getRoot(), "");
27         for(int i=0;i<strings.size();i++) {
28             System.out.println("strings.get(i) = " + strings.get(i));
29         }
30
31
32     }
33
34     /**
35      * Gets the root of the trie
36      * @return the root
37      */
38     public TrieNode getRoot() {
39         return this.root;
40     }
41
42     /**
43      * Sets the root of the trie
44      * @param root the node to set the root as
45      */
46     public void setRoot(TrieNode root) {
47         this.root = root;
48     }
49
50     /**
51      * The constructor for a Trie
52      */
53     public Trie() {
54         this.root = new TrieNode();
```

```
55      }
56
57      /**
58       * Adds a string to the trie
59       * @param str the string to add
60       */
61      public void add(String str) {
62          Boolean nullVisited = false;
63          TrieNode temp = root;
64          TrieNode next = null;
65          char[] stringToCharArray = str.toCharArray();
66          for (char c: stringToCharArray) {
67              if (nullVisited == false) {
68                  next = temp.getOffspring(c);
69                  if (next == null) {
70                      next = new TrieNode(c);
71                      temp.setOffspring(next.getChar());
72                      nullVisited = true;
73                  }
74                  temp = temp.getOffspring(c);
75              }
76              else {
77                  next = new TrieNode(c);
78                  temp.setOffspring(next.getChar());
79                  temp = temp.getOffspring(c);
80              }
81
82
83          }
84          temp.setIsKey(true);
85      }
86
87      /**
88       * Checks if a string is in the trie
89       * @param str the string to check for
90       * @return true if it is in the trie, false if it is not
91       */
92      public Boolean contains(String str) {
93          TrieNode temp = root;
94          char[] stringToCharArray = str.toCharArray();
95          for (int i=0;i<stringToCharArray.length;i++) {
96              temp=temp.getOffspring(stringToCharArray[i]);
97              if (temp==null) {
98                  return false;
99              }
100         }
101         return true;
102     }
103
104     /**
105      * performs a breadth first search
106      * @return a string consisting of the search result
107      */
108     public String outputBreadthFirstSearch() {
109         String result = "";
110         Queue<TrieNode> queue = new LinkedList<>();
111         queue.add(this.getRoot());
112         while(!queue.isEmpty()) {
113             TrieNode temp = queue.remove();
114             if (!temp.equals(this.getRoot())) {
```

```
115                 result += temp.getChar();
116             }
117             for (TrieNode node : temp.getAllOffspring()) {
118                 if (node!=null) {
119                     queue.add(node);
120                 }
121             }
122         }
123         return result;
124     }
125 
126     /**
127      * Performs a depth first search
128      * @return the search result in the form of a string
129      */
130     public String outputDepthFirstSearch() {
131         String result = "";
132         Stack<TrieNode> stack = new Stack<>();
133         stack.push(this.getRoot());
134         while(!stack.isEmpty()) {
135             TrieNode temp = stack.pop();
136             if (!temp.equals(this.getRoot())) {
137                 result += temp.getChar();
138             }
139             for (int i = temp.getAllOffspring().length-1;i>=0;i--) {
140                 if (temp.getOffspring((char)(i+97)) != null) {
141                     stack.push(temp.getOffspring((char)(i+97)));
142                 }
143             }
144         }
145         return result;
146     }
147 
148     /**
149      * Makes a subtrie after the prefix
150      * @param prefix the prefix to find the subtrie after
151      * @return the subtrie
152      */
153     public Trie getSubTrie(String prefix) {
154         TrieNode temp = this.root;
155         char[] stringToCharArray = prefix.toCharArray();
156         for (int i=0;i<stringToCharArray.length;i++) {
157             temp = temp.getOffspring(stringToCharArray[i]);
158         }
159         Trie subTrie = new Trie();
160         subTrie.setRoot(new TrieNode(temp));
161         subTrie.getRoot().setChar('\0');
162 //        subTrie.getRoot().setIsKey(false);
163         return subTrie;
164     }
165 
166     /**
167      * gets all the words from a trie
168      * @param node the node to start on
169      * @param word the word being made
170      * @return a list of the words
171      */
172     public List<String> getAllWords(TrieNode node, String word){
173         List<String> list = new ArrayList<>();
174         Stack<TrieNode> stack = new Stack<>();
```

```
175         if (!node.equals(this.getRoot())) {
176             word += node.getChar();
177         }
178         for(TrieNode trieNode : node.getAllOffspring()){
179             if(trieNode!=null) {
180                 stack.push(trieNode);
181             }
182         }
183         while(!stack.empty()){
184             list.addAll(getAllWords(stack.pop(),word));
185         }
186         if(node.getIsKey()){
187             list.add(word);
188         }
189         return list;
190     }
191 
192 }
```

4.3 TrieNode

Listing 3: TrieNode.java

```
/*
https://www.programcreek.com/2014/05/leetcode-implement-trie-prefix-tree-java/
suggests using array for offspring for faster performance
 */
public class TrieNode {
    private TrieNode[] offspring;
    private boolean isKey;
    private char s;
    /**
     * method for making an offspring
     * @param s is the character to set
     */
    public TrieNode(char s) {
        this.offspring = new TrieNode[26];
        this.isKey = false;
        this.s = s;
    }
    /**
     * method for making the root of the trie
     */
    public TrieNode() {
        this.offspring = new TrieNode[26];
        this.isKey = false;
        this.s = Character.MIN_VALUE;
    }

    /**
     * A method for duplicating a TrieNode
     * @param node the node to be duplicated
     */
    public TrieNode(TrieNode node) {
        this.offspring = node.getAllOffspring();
        this.isKey = node.getIsKey();
        this.s = node.getChar();
    }

    /**
     * Returns the offsping node at a character
     * @param x the character to find the node at
     * @return the node the be returned
     */
    public TrieNode getOffspring(char x) {
        return offspring[(int)x-97];
    }

    /**
     * Returns the offspring array
     * @return the array
     */
    public TrieNode[] getAllOffspring() {
        return this.offspring;
    }

    /**
     * Sets the offspring at a character
     * @param x The char to set at the node
     * @return true if was null, false if it was already assigned
```

```
58       */
59      public Boolean setOffspring(char x) {
60          if (this.offspring[(int)x-97] == null) {
61              this.offspring[(int)x-97] = new TrieNode(x);
62              return true;
63          }
64          return false;
65      }
66
67      /**
68       * Sets the offspring to a certain node
69       * @param x the node to it as
70       * @return True if the node was null, false if it wasnt
71       */
72      public Boolean setOffSpring(TrieNode x) {
73          if (this.offspring[(int)x.getChar()-97] == null) {
74              this.offspring[(int)x.getChar()-97] = new TrieNode(x.getChar());
75              return true;
76          }
77          return false;
78      }
79
80      /**
81       * Returns if the node is a key
82       * @return true if it is a key, false if it isnt
83       */
84      public Boolean getIsKey() {
85          return isKey;
86      }
87
88      /**
89       * Sets the nodes key status
90       * @param x the status to set it as
91       */
92      public void setIsKey(Boolean x) {
93          isKey = x;
94      }
95
96      /**
97       * Gets the char of the node
98       * @return the char
99       */
100     public char getChar() {
101         return s;
102     }
103
104     /**
105      * Sets the char at a node
106      * @param x the char to set it as
107      */
108     public void setChar(char x) {
109         this.s = x;
110     }
111
112
113
114 }
```

4.4 AutoCompletionTrie

Listing 4: AutoCompletionTrie.java

```
1  import java.util.*;
2
3  public class AutoCompletionTrie {
4
5      private AutoCompletionTrieNode root;
6      public static void main(String[] args) {
7          AutoCompletionTrie test = new AutoCompletionTrie();
8          test.add("cheers");
9          test.add("cheese");
10         test.add("chat");
11         test.add("cat");
12         test.add("bat");
13         test.add("cheers");
14         test.add("batch");
15         System.out.println("test.getLast(\"cheers\") = " + test.getLast("cheers"));
16         if (test.contains("chat")) {
17             System.out.println("True");
18         }
19         else {
20             System.out.println("False");
21         }
22         System.out.println("test.outputBreadthFirstSearch() = " +
               ↪ test.outputBreadthFirstSearch());
23         System.out.println("test.outputDepthFirstSearch() = " +
               ↪ test.outputDepthFirstSearch());
24         AutoCompletionTrie subTrie = test.getSubTrie("ch");
25         System.out.println("subTrie.root.getOffspring('h').getChar() = " +
               ↪ subTrie.getRoot().getChar());
26         char[] test1 = null;
27
28         TreeMap<String, Integer> words = test.getAllWords(test.getRoot(), "");
29 //         List<String> strings = test.getAllWords(test.getRoot(), "");
30         for(int i=0;i<words.size();i++) {
31             System.out.println("words.keySet().toArray()[i] = " +
                   ↪ words.keySet().toArray()[i]);
32             System.out.println("words.values().toArray()[i] = " +
                   ↪ words.values().toArray()[i]);
33         }
34
35
36     }
37
38     /**
39      * Gets the last node in a word
40      * @param string the string to find the last node of
41      * @return the last node
42      */
43     public AutoCompletionTrieNode getLast(String string){
44         char[] strToChar = string.toCharArray();
45         AutoCompletionTrieNode last = this.root;
46         for (int i = 0; i<strToChar.length; i++){
47             last = last.getOffspring(strToChar[i]);
48         }
49         return last;
50     }
51
52     /**
```

```
53      * Gets the root of the trie
54      * @return the root
55      */
56     public AutoCompletionTrieNode getRoot() {
57         return this.root;
58     }
59 
60     /**
61      * Sets the root of the trie
62      * @param root the node to set the root as
63      */
64     public void setRoot(AutoCompletionTrieNode root) {
65         this.root = root;
66     }
67 
68     /**
69      * The constructor for a Trie
70      */
71     public AutoCompletionTrie() {
72         this.root = new AutoCompletionTrieNode();
73     }
74 
75     /**
76      * Adds a string to the trie
77      * **** Also increases the frequency if the word is already in the trie
78      * @param str the string to add
79      */
80     public void add(String str) {
81         Boolean nullVisited = false;
82         AutoCompletionTrieNode temp = root;
83         AutoCompletionTrieNode next = null;
84         char[] stringToCharArray = str.toCharArray();
85         for (char c: stringToCharArray) {
86             if (nullVisited == false) {
87                 next = temp.getOffspring(c);
88                 if (next == null) {
89                     next = new AutoCompletionTrieNode(c);
90                     temp.setOffspring(next.getChar());
91                     nullVisited = true;
92                 }
93                 temp = temp.getOffspring(c);
94             }
95             else {
96                 next = new AutoCompletionTrieNode(c);
97                 temp.setOffspring(next.getChar());
98                 temp = temp.getOffspring(c);
99             }
100 
101 
102         }
103         temp.setIsKey(true);
104         temp.setFrequency(temp.getFrequency()+1);
105     }
106 
107     /**
108      * Checks if a string is in the trie
109      * @param str the string to check for
110      * @return true if it is in the trie, false if it is not
111      */
112     public Boolean contains(String str) {
```

```
113         AutoCompletionTrieNode temp = root;
114         char[] stringToCharArray = str.toCharArray();
115         for (int i=0;i<stringToCharArray.length;i++) {
116             temp=temp.getOffspring(stringToCharArray[i]);
117             if (temp==null) {
118                 return false;
119             }
120         }
121         return true;
122     }
123 
124     /**
125      * performs a breadth first search
126      * @return a string consisting of the search result
127      */
128     public String outputBreadthFirstSearch() {
129         String result = "";
130         Queue<AutoCompletionTrieNode> queue = new LinkedList<>();
131         queue.add(this.getRoot());
132         while(!queue.isEmpty()) {
133             AutoCompletionTrieNode temp = queue.remove();
134             if (!temp.equals(this.getRoot())) {
135                 result += temp.getChar();
136             }
137             for (AutoCompletionTrieNode node : temp.getAllOffspring()) {
138                 if (node!=null) {
139                     queue.add(node);
140                 }
141             }
142         }
143         return result;
144     }
145 
146     /**
147      * Performs a depth first search
148      * @return the search result in the form of a string
149      */
150     public String outputDepthFirstSearch() {
151         String result = "";
152         Stack<AutoCompletionTrieNode> stack = new Stack<>();
153         stack.push(this.getRoot());
154         while(!stack.isEmpty()) {
155             AutoCompletionTrieNode temp = stack.pop();
156             if (!temp.equals(this.getRoot())) {
157                 result += temp.getChar();
158             }
159             for (int i = temp.getAllOffspring().length-1;i>=0;i--) {
160                 if (temp.getOffspring((char)(i+97)) != null) {
161                     stack.push(temp.getOffspring((char)(i+97)));
162                 }
163             }
164         }
165         return result;
166     }
167 
168     /**
169      * Makes a subtrie after the prefix
170      * @param prefix the prefix to find the subtrie after
171      * @return the subtrie
172      */
```

```
173     public AutoCompletionTrie getSubTrie(String prefix) {
174         AutoCompletionTrieNode temp = this.root;
175         char[] stringToCharArray = prefix.toCharArray();
176         for (int i=0;i<stringToCharArray.length;i++) {
177             temp = temp.getOffspring(stringToCharArray[i]);
178         }
179         AutoCompletionTrie subTrie = new AutoCompletionTrie();
180         subTrie.setRoot(new AutoCompletionTrieNode(temp));
181         subTrie.getRoot().setChar('\0');
182 //          subTrie.getRoot().setIsKey(false);
183         return subTrie;
184     }
185 
186     /**
187      * gets all the words from a trie
188      * @param node the node to start on
189      * @param word the word being made
190      * @return a list of the words
191      */
192     public TreeMap<String, Integer> getAllWords(AutoCompletionTrieNode node, String
        ↪ word){
193         TreeMap<String, Integer> words = new TreeMap<>();
194         Stack<AutoCompletionTrieNode> stack = new Stack<>();
195         if (!node.equals(this.getRoot())) {
196             word += node.getChar();
197         }
198         for(AutoCompletionTrieNode trieNode : node.getAllOffspring()){
199             if(trieNode!=null) {
200                 stack.push(trieNode);
201             }
202         }
203         while(!stack.empty()){
204             words.putAll(getAllWords(stack.pop(),word));
205         }
206         if(node.getIsKey()){
207             words.put(word, node.getFrequency());
208         }
209         return words;
210     }
211 
212 }
```

4.5 AutoCompletionTrieNode

Listing 5: AutoCompletionTrieNode.java

```
1  public class AutoCompletionTrieNode {
2      private AutoCompletionTrieNode[] offspring;
3      private boolean isKey;
4      private char s;
5      private int frequency;
6
7      /**
8       * method for making an offspring
9       * @param s is the character to set
10      */
11     public AutoCompletionTrieNode(char s) {
12         this.offspring = new AutoCompletionTrieNode[26];
13         this.isKey = false;
14         this.s = s;
15         this.frequency = 0;
16     }
17
18     /**
19      * method for making the root of the trie
20      */
21     public AutoCompletionTrieNode() {
22         this.offspring = new AutoCompletionTrieNode[26];
23         this.isKey = false;
24         this.s = Character.MIN_VALUE;
25         this.frequency = 0;
26     }
27
28     /**
29      * A method for duplicating a TrieNode
30      * @param node the node to be duplicated
31      */
32     public AutoCompletionTrieNode(AutoCompletionTrieNode node) {
33         this.offspring = node.getAllOffspring();
34         this.isKey = node.getIsKey();
35         this.s = node.getChar();
36         this.frequency = node.getFrequency();
37
38     }
39
40     /**
41      * Returns the offsping node at a character
42      * @param x the character to find the node at
43      * @return the node the be returned
44      */
45     public AutoCompletionTrieNode getOffspring(char x) {
46         return offspring[(int)x-97];
47     }
48
49     /**
50      * Returns the offspring array
51      * @return the array
52      */
53     public AutoCompletionTrieNode[] getAllOffspring() {
54         return this.offspring;
55     }
56
57     /**
```

```
58      * Sets the offspring at a character
59      * @param x The char to set at the node
60      * @return true if was null, false if it was already assigned
61      */
62     public Boolean setOffspring(char x) {
63         if (this.offspring[(int)x-97] == null) {
64             this.offspring[(int)x-97] = new AutoCompletionTrieNode(x);
65             return true;
66         }
67         return false;
68     }
69 
70     /**
71      * Sets the offspring to a certain node
72      * @param x the node to it as
73      * @return True if the node was null, false if it wasnt
74      */
75     public Boolean setOffSpring(AutoCompletionTrieNode x) {
76         if (this.offspring[(int)x.getChar()-97] == null) {
77             this.offspring[(int)x.getChar()-97] = new
                ↪ AutoCompletionTrieNode(x.getChar());
78             return true;
79         }
80         return false;
81     }
82 
83     /**
84      * Returns if the node is a key
85      * @return true if it is a key, false if it isnt
86      */
87     public Boolean getIsKey() {
88         return isKey;
89     }
90 
91     /**
92      * Sets the nodes key status
93      * @param x the status to set it as
94      */
95     public void setIsKey(Boolean x) {
96         isKey = x;
97     }
98 
99     /**
100     * Gets the char of the node
101     * @return the char
102     */
103    public char getChar() {
104        return s;
105    }
106 
107    /**
108     * Sets the char at a node
109     * @param x the char to set it as
110     */
111    public void setChar(char x) {
112        this.s = x;
113    }
114 
115    /**
116     * gets the frequency of a word
```

```
117         * @return the frequency
118         */
119        public int getFrequency() {
120            return frequency;
121        }
122
123        /**
124         * Sets the frequency of a word
125         * @param f1 the frequency to set it as
126         */
127        public void setFrequency(int f1) {
128            this.frequency = f1;
129        }
130
131
132
133    }
```

4.6 AutoComplete

Listing 6: AutoComplete.java

```
1  import java.io.FileNotFoundException;
2  import java.io.FileWriter;
3  import java.io.IOException;
4  import java.io.PrintWriter;
5  import java.io.File;
6  import java.util.*;
7
8  public class AutoComplete {
9      private DictionaryFinder df;
10     private AutoCompletionTrie trie;
11
12     /**
13      * The classes constructor
14      */
15     public AutoComplete() {
16         this.df = new DictionaryFinder();
17         this.trie = new AutoCompletionTrie();
18     }
19
20     /**
21      * This populates the trie with all the words in the dictionary finder hashmap
           ↪ and then finds the key node
22      * and assigns the amount of times the word occurs to it
23      */
24     public void populateTrie() {
25         for (String string : this.df.getDictionary().keySet()) {
26             this.trie.add(string);
27             this.trie.getLast(string).setFrequency(this.df.getDictionary().get(string));
28         }
29     }
30
31     /**
32      * This populates the hashmap for the dictionary finder and tallies the amount
           ↪ each word comes up
33      * @param file this is the file that the words come from
34      * @throws FileNotFoundException
35      */
36     public void setDictionaryFinder(String file) throws FileNotFoundException {
37         ArrayList<String> in=DictionaryFinder.readWordsFromCSV(file);
38         this.df.setIn(in);
39         this.df.formDictionary();
40     }
41
42     /**
43      * This finds all words that come after the query then finds the probabilty of
           ↪ each word by dividing number of
44      * times the word comes up by the total number of words possible within that query
45      * @param prefix The query prefix
46      * @return probabilties followed by the word
47      */
48     public TreeMap<Double, String> probabilityWords(String prefix) {
49         TreeMap<Double, String> probabiltyWords = new TreeMap<Double,
               ↪ String>(Collections.reverseOrder());
50         AutoCompletionTrie subTrie = trie.getSubTrie(prefix);
51         TreeMap<String, Integer> words = subTrie.getAllWords(subTrie.getRoot(),"");
52         Double totalWords = 0.0;
53         for (String string : words.keySet()) {
```

```
54              totalWords += words.get(string);
55
56          }
57          for (String string : words.keySet()){
58              probabiltyWords.put((words.get(string)/totalWords), (prefix +string));
59          }
60          return probabiltyWords;
61      }
62
63      /**
64       * Returns all the probabilties
65       * @param probabiltyWords The Treemap containing the words and the probabilities
66       * @return The probabilties
67       */
68      public List<Double> getProbabilities(TreeMap<Double, String> probabiltyWords) {
69          return new ArrayList(probabiltyWords.keySet());
70      }
71
72      /**
73       * Returns all the possible words matching that query
74       * @param probabiltyWords The Treemap containing the words and the probabilities
75       * @return all words
76       */
77      public List<String> getValues(TreeMap<Double, String> probabiltyWords) {
78          return new ArrayList(probabiltyWords.values());
79      }
80
81      /**
82       * reads all the queries from the file specified
83       * @param file the file
84       * @return all the queries
85       * @throws FileNotFoundException
86       */
87      public ArrayList<String> readQueries(String file) throws FileNotFoundException {
88          Scanner sc=new Scanner(new File(file));
89          ArrayList<String> words=new ArrayList<>();
90          String str;
91          while(sc.hasNext()){
92              str=sc.next();
93              str=str.trim();
94              str=str.toLowerCase();
95              words.add(str);
96          }
97          return words;
98      }
99
100     /**
101      * Saves the 3 most probable to a file specified
102      * @param file the file to save to
103      * @param probabiltyWords All the words sorted in order of probability
104      * @param prefix The query prefix
105      * @throws IOException
106      */
107     public void saveToFile(String file, TreeMap<Double, String> probabiltyWords,
        ↪ String prefix) throws IOException{
108         FileWriter fileWriter = new FileWriter(file, true);
109         PrintWriter printWriter = new PrintWriter(fileWriter);
110 //        int i =0;
111         int max = 0;
112         if (probabiltyWords.size() > 3) {
```

```
113             max = 3;
114         }
115         else {
116             max = probabiltyWords.size();
117         }
118         printWriter.print(prefix + ", ");
119         for (int i = 0;i<max;i++) {
120             printWriter.print(probabiltyWords.values().toArray()[i] + ", " +
121                     probabiltyWords.keySet().toArray()[i] + ", ");
122         }
123         printWriter.println();
124         printWriter.close();
125     }
126 
127     /**
128      * Used to show functionality
129      * @param args
130      * @throws Exception
131      */
132     public static void main(String[] args) throws Exception{
133         AutoComplete autoComplete = new AutoComplete();
134         autoComplete.setDictionaryFinder("TextFiles/lotr.csv");
135         autoComplete.populateTrie();
136 
137         ArrayList<String> queries =
               ↪ autoComplete.readQueries("TextFiles/lotrQueries.csv");
138         for (int i=0;i<queries.size();i++) {
139             TreeMap<Double, String> probabiltyWords =
                   ↪ autoComplete.probabilityWords(queries.get(i));
140             List<Double> getProbabilities =
                   ↪ autoComplete.getProbabilities(probabiltyWords);
141             List<String> getValues = autoComplete.getValues(probabiltyWords);
142             for (int j = 0; j<getProbabilities.size();j++) {
143                 System.out.println(getValues.get(j) + ", " + getProbabilities.get(j)
                       ↪ + "\n");
144             }
145             autoComplete.saveToFile("TextFiles/lotrAutoCompleted", probabiltyWords,
                   ↪ queries.get(i));
146         }
147 
148 
149 
150 
151 
152 
153     }
154 
155 
156 
157 
158 }
```