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

Student number: 100242165. Blackboard ID: dvd18scu

Wednesday 13th May, 2020 13:44

Contents

1 Part 1: Form a Dictionary and Word Frequency Count

Part 1 deals with document file reading and dictionary writing, and creating dictionaries for the auto complete. Dictionaries are CSV files where each line has a word, as well as a number, the frequency that word appears in a document.

1.1 formDictionary Pseudocode

formDictionary is a function that reads a document file, a CSV file with words, and creates a dictionary and writes it to file. A Tree Map is used as it is efficient in both accessing from and adding items to it. The word itself is used as the key since it is unique, and the frequency the data. The function uses readWordsFromCSV, which returns an array of words.

Algorithm 1 formDictionary(*readPath*,*writePath*)

```
Require: A valid file to read located at $readPath$, and a file to write to located at $writePath$.
Ensure: The file located at$writePath$ contains a list of words from the file located at $readPath$, along with the frequency
    of each word's appearance.
$\mathbf{M} \leftarrow Map()$                                  ▷ M is a map, where a string is the key, and the frequency is the data
$\mathbf{L} \leftarrow readWordsFromCSV(readPath)$                            ▷ L is a list of words read from readPath
for $i \leftarrow 1$ to $\mathbf{L}$.size() do
    $\mathbf{M}$.put($\mathbf{L}_i$,frequency($\mathbf{L}_i$))   ▷ $frequency()$ linearly counts the frequency.   ▷ Adds the words in L, along with the word
    count for each word to M.
$saveToFile(\mathbf{M}, writePath)$                                  ▷ saveToFile writes M to the file at writePath.
```

1.2 formDictionary Analysis

1. The fundamental operation for formDictionary is:
   $\mathbf{M}$.put($\mathbf{L}_i$,frequency($\mathbf{L}_i$))

2. Treemap is $\mathcal{O}(log(n))$ when adding item to it. All cases, worst, best and average are the same. Where n is the size of $\mathbf{L}$

3. $$f(g(n)) = \sum_{i=1}^{n} log(g(n)) \tag{1}$$

4. g(n) is the runtime complexity of $frequency()$, a linear scanning function, with a complexity of $\mathcal{O}(n)$. This makes the final runtime complexity

5. $$f(n) = \sum_{i=1}^{n} log(n^2) \tag{2}$$
   $$t(n) = nlog(n^2) \tag{3}$$

6. This simplifies further to 2nlog(n), log linear. Ignoring constants, the runtime complexity function is $\mathcal{O}(nlog(n))$

1.3 saveToFile Pseudocode

saveToFile is a simple function that writes a map with a string for its key and an integer for it's value to file. It saves the data as a CSV, with each line being a word and it's frequency.

Algorithm 2 saveToFile($\mathbf{M}$, *writePath*)

```
Require: A map M with a string as the key, and an integer as the data, and a file located at $writePath$
Ensure: A file at $writePath$ that contains all keys (words) and data (frequency) to be written to file.
$openFile \leftarrow open(writePath)$                                                  ▷ Opens the file
for all i in M do
    $tempString \leftarrow getKey(\mathbf{i}) + "," + getValue(\mathbf{i})$          ▷ Joining two strings, with a comma in-between.
    $writeline(openFile, tempString)$
$close(writePath)$                                                              ▷ Closes the file
```

1.4 saveToFile Analysis

1. Fundamental operation:
   $tempString \leftarrow getKey(\mathbf{e}) + "," + getValue(\mathbf{e})$

2. Getting items from a treemap is $\mathcal{O}(log(n))$. All cases, worst, best and average are the same. The number of times the loop loops, n is how many items/entries are in the treemap **M**.

3. $$f(n) = \sum_{i=1}^{n} log(n) + \sum_{i=1}^{n} log(n) \tag{4}$$
   $$t(n) = 2nlog(n) \tag{5}$$

4. Ignoring constants, the order of the runtime complexity is log linear, and the runtime complexity function is $\mathcal{O}(nlog(n))$.

2 Part 2: Implement a Trie Data Structure

This section is about implementing a Trie data structure. It's implementation is where each "node" is an array of size 26, where the index determines what letter it will be. For example, a is index 0, b is 1, and so on. An Ascii offset is mentioned a number of times in this section. This is for converting Ascii's representation of the letters into the node's representation for them. For example, the Ascii char "a" has the integer value of 97 instead of 0.

2.1 add Pseudocode

This function adds a string to the Trie structure. It returns true only if the word that was added was newly added. It works by traversing the trie, adding nodes if necessary, and returning the correct boolean value based on flags.

Algorithm 3 add(**key**) return **true or false**

```
Require: $key$, a string.
Ensure: a boolean value, true or false
alreadyIn ← true
parent ← root                                         ▷ root is the the root node of the trie
for level ← 1 to key.size() do                        ▷ goes through all letters of the key.
    index ← charToInt(level)
    if $parent.getOffspring()_{index}$ = null then    ▷ if the node for the current letter doesn't exist.
        $parent.getOffspring()_{index} = TrieNode()$  ▷ make a new one
        alreadyIn ← false
    parent ← $parent.getOffspring()_{index}$          ▷ advanced parent node to one further down
if alreadyIn && parent.getIsEnd() then                ▷ if everything was already done, return false
    return false
parent.setIsEnd(true)                                 ▷ sets last node for key as end.
return true
```

2.2 contains Pseudocode

This function checks if a word is present in the trie. It returns true only if the word is both in the trie and is marked as a key. It also works by traversing the trie.

Algorithm 4 contains(**key**) return **true or false**

```
Require: $key$, a string.
Ensure: a boolean value, true or false
alreadyIn ← true
parent ← root                                         ▷ root is the the root node of the trie
for level ← 1 to key.size() do                        ▷ goes through all letters of the key.
    index ← charToInt(level)
    if $parent.getOffspring()_{index}$ = null then    ▷ if the node for the current letter doesn't exist.
        return false
    parent ← $parent.getOffspring()_{index}$          ▷ advanced parent node to one further down
return ($parent \neq$ null && parent.getIsEnd())      ▷ returns true only if parent exists AND is a key
```

2.3 outputBreadthFirstSearch Pseudocode

This function explores and prints the contents of the trie in breadth first order. It uses a queue. It goes through the queue, originally starting with the root, adding it's offspring to the queue. It repeats this until there are no more nodes.

Algorithm 5 outputBreadthFirstSearch() **r**eturn (*result*)

```
Require: Access to a valid Trie root node.
Ensure: A string $result$, which contains all characters in a trie in breadth first order.
$\mathbf{Q} \leftarrow Queue()$                                  ▷ Q is a queue of trie nodes
$\mathbf{Q}.add(\mathbf{root})$                                  ▷ Add the root node to the front of the queue
while $\mathbf{Q} \neq$ empty do
    $\mathbf{tempNode} \leftarrow \mathbf{Q}.remove()$                ▷ remove() removes the top trie node
    $\mathbf{O} \leftarrow \mathbf{tempNode}.getOffspring()$          ▷ getOffspring() returns an array of the node's children
    for $i \leftarrow 1$ to $\mathbf{O}.size()$ do
        if $\mathbf{O}_i \neq$ null then
            $\mathbf{Q}.add(\mathbf{O}_i)$
            $result \leftarrow result + intToChar(i + 97)$    ▷ append character to result. 97 is the offset for Ascii letters
return $(result)$
```

2.4 outputDepthFirstSearch Pseudocode

This is the helper function to outputDepthFirstSearch. It provides the actual logic with a string to start appending to, as it is recursive.

Algorithm 6 outputDepthFirstSearch() **r**eturn (*result*)

```
Require: Access to a valid Trie root node.
Ensure: A string $result$, which contains all characters in a trie in depth first order.
return $outputDepthFirstSearch(result, \mathbf{root})$
```

This is the actual logic of outputDepthFirstSearch. It linearly goes through every child of a node, and calls itself on the child as if it were the parent. It does this until there are no more nodes.

Algorithm 7 outputDepthFirstSearch(*result*, **root**) **r**eturn (*result*)

```
Require: $result$, a string to build on, and root, a valid root node to a trie
Ensure: A string $result$, which contains all characters in a trie in depth first order.
$\mathbf{O} \leftarrow \mathbf{root}.getOffspring()$                  ▷ getOffspring() returns an array of the node's children
for $i \leftarrow 1$ to $\mathbf{O}.size()$ do                        ▷ For every child
    if $\mathbf{O}_i \neq$ null then
        $result \leftarrow result + intToChar(i + 97)$       ▷ append character to result. 97 is the offset for Ascii letters
        $outputDepthFirstSearch(result, \mathbf{O}_i)$
return $result$
```

2.5 getSubTrie Pseudocode

This function gets all the words from a trie starting with a given prefix, and returns a new trie from it. It simply explores the trie to make sure the prefix is in the trie, then creates a trie with the node for the last letter in the prefix as the root.

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

```
Require: root, a valid node of a trie, and $prefix$, a prefix for a word/words.
Ensure: subTrie is a trie that starts from $prefix$.
tempNode ← root
for $i \leftarrow 1$ to $prefix.size()$ do
    $index \leftarrow charToInt(prefix_i) - 97$    ▷ charToInt(character) converts an Ascii character to its int version
    if tempNode = null then
        return null
    tempNode ← tempNode.$getOffSpring()_{index}$
return $Trie$(tempNode)
```

2.6 getAllWords Pseudocode

This is a helper function for getAllWords, providing the main recursive function with an ArrayList of strings as well as an empty StringBuilder.

Algorithm 9 getAllWords() return **W**

```
Require: Access to a valid trie's root.
Ensure: W, an array of strings of varying length.
$getAllWords(\mathbf{W}, builder, \mathbf{root})$    ▷ builder is an empty string, and root is the root of the Trie to be used.
return W
```

This is the main function, where it is to build an array of all the words present in the Trie it is called recursively. It works quite similarly to outputDepthFirstSearch, but whilst building an array of the trie's words.

Algorithm 10 getAllWords(**W**, *builder*, **root**)

```
Require: W, an array of strings of varying length, builder, a string to append to, and root, the root of the given Trie
    structure.
Ensure: W has been filled with all the words from the Trie.
if root.$getIsEnd()$ then
    W.$add(builder)$
$\mathbf{O} \leftarrow$ root.$getOffspring()$
for $i \leftarrow 1$ to O.$size()$ do
    if $\mathbf{O}_i \neq$ null then
        $builder \leftarrow builder + intToChar(97 + i)$    ▷ 97 is the Ascii offset. intToChar converts the int to it's char equivalent
        $getAllWords(\mathbf{W}, builder, \mathbf{O}_i)$
        $builder \leftarrow builder.shorten(1)$    ▷ Removes last character from builder - "goes back up a level"
```

3 Part 3: Word Auto Completion Application

This part is the autocomplete program. It uses part 1 and a modified part 2. The Trie structure now also implements the frequency count of each word. This affects add and contains slightly as well. It also features a nested static class, fullInfo. Each fullInfo object simply stores a word and its frequency, and getAllInfo acts like getAllWords but returning fullInfo objects instead. This is used to implement sorting to generate the top 3 results for a word. It also features another saveToFile, which simply saves the generated string output of the function to file. Finally, it implements a function addToTrie, used to add a dictionary to a trie, using add().

3.1 getAllInfo Pseudocode

This is a helper function, providing the main recursive function with an ArrayList of strings as well as an empty StringBuilder. The fullInfo object used consists of an int and a string, being frequency and the word itself.

Algorithm 11 getAllInfo() **return I**

Require: Access to a valid **root** to a Trie
Ensure: **I**, an array of fullInfo objects of varying length.

```
$getAllInfo(\mathbf{I}, builder, \mathbf{root})$    ▷ $builder$ is an empty string, and $root$ is the root of the Trie to be used.
$\mathbf{I}.sortByFrequency()$    ▷ $sortByFrequency()$ is a function that sorts by fullInfo's frequency, followed by key.
return I
```

This is the main function, where it is to build an array of all the words present, as well as each word's frequency. The fullInfo object used consists of an int and a string, being frequency and the word itself. The function works similarly to getAllWords, which in turn is similar to outputDepthFirstSearch.

Algorithm 12 getAllInfo(**I**, **builder**, **root**)

Require: **I**, an array of fullInfo objects of varying length, **builder**, a string to append to, and **root**, the root of the given Trie structure.
Ensure: **I** has been filled with all the words and information from the Trie.

```
if $\mathbf{root}.getIsEnd()$ then
    $tempInfo \leftarrow fullInfo(builder, \mathbf{root}.getFrequency())$
    $\mathbf{I}.add(tempInfo)$
else
    $\mathbf{O} \leftarrow \mathbf{root}.getOffspring()$
    for $i \leftarrow 1$ to $\mathbf{O}.size()$ do
        if $\mathbf{O}_i \neq$ null then
            $builder \leftarrow builder + intToChar(97 + i)$    ▷ 97 is the Ascii offset. intToChar converts ints to it's char equivalent
            $getAllInfo(\mathbf{I}, builder, \mathbf{O}_i)$
            $builder \leftarrow builder.shorten(1)$    ▷ Removes last character from builder - "goes back up a level"
```

3.2 addToTrie Pseudocode

This is quite a simple function. It reads in a dictionary formatted file and calls the add() function on each word and it's frequency. (add() in part 3 takes in both values)

Algorithm 13 addToTrie(*readPath*)

Require: Access to a correctly formatted dictionary file, located with the string *readPath*. Access to a trie to add to.
Ensure: The contents of the dictionary file to be added to the trie.

```
$openFile \leftarrow open(readPath)$
while $openFile \neq end$ do    ▷ while not the end of file
    $line \leftarrow openFile.readLine()$
    $splitLine \leftarrow line.split(","\,)$    ▷ Splits the line by the comma character
    $add(splitLine_1, splitLine_2)$    ▷ Calls the add function
$openFile.close()$
```

3.3 saveToFile Pseudocode

This is another saveToFile function that simply writes a string to a file as a line in the file.

Algorithm 14 saveToFile(*line*,*writePath*)

Require: Access to a file to write to, located at *writePath*.
Ensure: *line* has been written to the file at *writePath*.

```
$openFile \leftarrow open(writePath)$
$openFile.writeLine(line)$
$openFile.close()$
```

3.4 autoCompletion Pseudocode

This function uses the prior functions to generate the top three results for a prefix, then generates a formatted line to have it be written to file by saveToFile(). It makes a subTrie for every prefix, then calculates and generates the ideal output for it.

Algorithm 15 completion(**masterTrie**, **P**,**n**, **writePath**)

Require: masterTrie, the complete trie structure, and P, an array of prefixes (strings), of length n. A location for the file to write to, *writePath*.
Ensure: The printing of the top three most frequent words for each prefix in *prefixes*, along with their frequency and probability.

```
$infoString \leftarrow$ ””                                        ▷ This is the string that will be written to file
for $i \leftarrow 1$ to $n$ do                                      ▷ For every prefix
    currentSubTrie ← masterTrie.$getSubTrie(\mathbf{P}_i)$
    I ← currentSubTrie.$getAllInfo()$            ▷ I is an array of info about each word beginning with the prefix
    $totalFreq \leftarrow 0$
    for $j \leftarrow 1$ to I.$size()$ do
        $totalFreq \leftarrow totalFreq + \mathbf{I}_j.getFreq()$
    if I.$size() < 3$ then                                   ▷ If there are less results than the top 3
        $printLimit \leftarrow \mathbf{I}.size()$
    else
        $printLimit \leftarrow 3$
    for $j \leftarrow 1$ to $printLimit$ do
        $prob \leftarrow \mathbf{I}_j.getFreq()/totalFreq$                    ▷ Calculates the probability ratio
        $print(\mathbf{P}_i, \mathbf{I}_j.getKey() + ”,” + prob)$
        $infoString \leftarrow infoString + \mathbf{P}_i + \mathbf{I}_j.getKey() + ”,” + prob + ”,”$
    $infoString \leftarrow infoString + ”/n”$                              ▷ End the line
$saveToFile(infoString, writePath)$
```

4 Code Listing

4.1 DictionaryMaker

Listing 1: DictionaryMaker.java

```java
/*
By/Modified by Robin Rai (100242165)
V.1.0.0
Created on 05/03/2020
*/
package dsacoursework2;

import java.io.*;
import java.util.*;

public class DictionaryMaker {

        public DictionaryMaker() {
        }

        public static ArrayList<String> readWordsFromCSV(String file, String delim)
                        throws FileNotFoundException {
                Scanner sc = new Scanner(new File(file));
                sc.useDelimiter(delim);
                ArrayList<String> words = new ArrayList<>();
                String str;
                while (sc.hasNext()) {
                        str = sc.next();
                        str = str.trim();
                        str = str.toLowerCase();
                        words.add(str);
                }
                return words;
        }

        public static void saveToFile(Map<String, Integer> map, String file)
                        throws IOException {
                //goes through any map and writes it to file
                FileWriter fileWriter = new FileWriter(file);
                PrintWriter printWriter = new PrintWriter(fileWriter);
                for (Map.Entry<String, Integer> entry : map.entrySet()) {
                        //System.out.println(entry.getKey() + "," + entry.getValue());
                        printWriter.println(entry.getKey() + "," + entry.getValue());
                        //writes key/word, followed by its value/frequency
                }
                printWriter.close();
        }

        public static void saveToFile(String line, String file) throws IOException {
                //simply writes the string to file
                FileWriter fileWriter = new FileWriter(file);
                PrintWriter printWriter = new PrintWriter(fileWriter);
                printWriter.println(line);
                printWriter.close();
        }

        //form a set of words that exist and count the frequency of each word
        public void formDictionary(String fileDirectory, String saveDirectory) {
                TreeMap<String, Integer> map = new TreeMap<String, Integer>();
                //a treeMap is the most efficient for this. The string is the key,
                // and the int is the value, since
                //all string's are unique
                try {
                        ArrayList<String> temp = readWordsFromCSV(fileDirectory, ",");
                        //takes in a big ol array of words
                        for (int i = 0; i < temp.size(); i++) {
                                map.put(temp.get(i), Collections.frequency(temp, temp.get(i)));
                                //for every word, put them in the treeMap as the key, with the
                                // frequency as the value
                        }
                } catch (Exception e) {
                        System.out.println("formDictionary: readWordsFromCSV");
                }
                //System.out.println(map);
                try {
                        saveToFile(map, saveDirectory);
                        //save it to file
                } catch (Exception e) {
                        System.out.println("formDictionary: saveToFile");
                }
        }


        public static void main(String[] args) throws Exception {
                dsacoursework2.DictionaryMaker df = new dsacoursework2.DictionaryMaker();
                ArrayList<String> in = readWordsFromCSV(
                                "src\\TextFiles\\testDocument.csv", ",");
                System.out.println("Array: " + in);

                df.formDictionary("src\\TextFiles\\testDocument.csv",
                                "src\\TextFiles\\formDictionaryTest.csv");
                saveToFile("Text to write",
                                "src\\TextFiles\\formDictionaryTest2.csv");
                System.out.println("Saved to file (both methods) successfully.");
        }

}
```

4.2 Trie

Listing 2: Trie.java

```
/*
By Robin Rai (100242165)
V.1.0.0
Created on 05/03/2020
*/
package dsacoursework2;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class Trie {
        private TrieNode root;

        public Trie() {
                root = new TrieNode();
```

```java
        }

        public Trie(TrieNode input) {
                root = input;
        }

        boolean add(String key) {
                if (key.equals("")) {
                        return false;
                }
                int index;
                boolean alreadyIn = true;
                TrieNode temp = root;
                for (int level = 0; level < key.length(); level++) {
                        //goes through all letters of the key
                        index = key.charAt(level) - 'a';
                        if (temp.getOffspring()[index] == null) {
                                //if the node for the current letter doesn't exist
                                temp.getOffspring()[index] = new TrieNode();
                                //make a new one
                                alreadyIn = false;
                                //set the flag
                        }
                        temp = temp.getOffspring()[index];
                        //advanced temp node to one further down
                }
                if (alreadyIn && temp.getIsEnd()) {
                        //if everything was already done, return false
                        return false;
                }
                temp.setIsEnd(true);
                //sets last node for key as end.
                return true;
        }

        boolean contains(String key) {
                int level;
                int length = key.length();
                int index;
                TrieNode temp = root;
                for (level = 0; level < length; level++) {
                        //goes through all letters of the key
                        index = key.charAt(level) - 'a';
                        //index is current char's number
                        if (temp.getOffspring()[index] == null) {
                                //if the node for the current letter doesn't exist
                                return false;
                                //return false, it isn't there
                        }
                        temp = temp.getOffspring()[index];
                        //advance down a level
                }
                return (temp != null && temp.getIsEnd());
                //returns true only if it exists AND is a key - not just a prefix.
        }


        String outputBreadthFirstSearch() {
                Queue<TrieNode> queue = new LinkedList<>();
                StringBuilder builder = new StringBuilder();
                queue.add(root);
                //starts with root
                while (!queue.isEmpty()) {
                        TrieNode temp = queue.remove();
                        //takes first thing in queue
                        for (int i = 0; i < 26; i++) {
                                if (!(temp.getOffspring()[i] == null)) {
                                        //for all of it's children
                                        queue.add(temp.getOffspring()[i]);
                                        //add them to the queue so it goes through
                                        //their children - width first
                                        builder.append((char) (i + 97));
                                        //adds it to the output
                                }
                        }
                }
                return builder.toString();
        }

        public String outputDepthFirstSearch() {
                //helper function, provides string to append to
                StringBuilder builder = new StringBuilder();
                return outputDepthFirstSearch(builder, this.root);
        }

        String outputDepthFirstSearch(StringBuilder builder, TrieNode trieNode) {
                TrieNode[] children = trieNode.getOffspring();
                //makes an array of all children in the tree
                for (int i = 0; i < 26; i++) {
                        //for all of a node's children
                        if (children[i] != null) {
                                //if there's a child
                                builder.append((char) (i + 97));
                                //add it's character
                                outputDepthFirstSearch(builder, children[i]);
                                //call the function again on it's children
                        }
                }
                return builder.toString();
        }

        Trie getSubTrie(String key) {

                TrieNode temp = root;
                for (int level = 0; level < key.length(); level++) {
                        //for the key's length
                        int index = key.charAt(level) - 'a';
                        //goes to right character/node
                        if (temp.getOffspring()[index] == null) {
                                //checks if whole key is present
                                return null;
                        }
                        temp = temp.getOffspring()[index];
                }
                return new Trie(temp);
                //returns only if it exists AND is a key - not just a prefix.

        }

        public ArrayList<String> getAllWords() {
                //helper function, is used just for the ArrayList and
                   ↪ String/stringBuilder
                ArrayList<String> listOfWords = new ArrayList<>();
                getAllWords(listOfWords, new StringBuilder(), root);
                return listOfWords;
        }

        private void getAllWords(ArrayList listOfWords, StringBuilder builder,
                                                         TrieNode root) {
                if (root.getIsEnd()) {
                        //checks if it's reached the end
                        //System.out.println(builder.toString());
                        listOfWords.add(builder.toString());
                        //print it and add it to the array
                }
                TrieNode[] offspring = root.getOffspring();
                //get's current node's children
                for (int i = 0; i < offspring.length; i++) {
                        //for all the children
                        if (offspring[i] != null) {    //if it's a valid node
                                getAllWords(listOfWords, builder.append((char)
                                                (97 + i)), offspring[i]);
                                //call itself, with the child as the root
                                builder.setLength(builder.length() - 1);
                                //goes up a level - resets stringbuilder to right
                                   ↪ place
                        }
                }
        }

        public static void main(String args[]) {
                Trie potato = new Trie();
                System.out.println("Adding...");
                System.out.println(potato.add("bat"));
                System.out.println(potato.add("cat"));
                System.out.println(potato.add("chat"));
                System.out.println(potato.add("cheese"));
                System.out.println(potato.add("cheers"));

                System.out.println("Checking...");
                System.out.println(potato.contains("yeet"));
                System.out.println(potato.contains("ca"));
                System.out.println(potato.contains("cat"));

                System.out.println("Breadth...");
                System.out.println(potato.outputBreadthFirstSearch());
                System.out.println("Depth...");
                System.out.println(potato.outputDepthFirstSearch());

                System.out.println("getSubTrie \"ch\"");
                Trie potato2 = potato.getSubTrie("ch");

                System.out.println("Breadth...");
                System.out.println(potato2.outputBreadthFirstSearch());
                System.out.println("Depth...");
                System.out.println(potato2.outputDepthFirstSearch());

                System.out.println("Checking...");
                System.out.println(potato2.contains("eese"));
                System.out.println(potato2.contains("eers"));
                System.out.println(potato2.contains("at"));
                System.out.println(potato2.contains("yeet"));
                System.out.println(potato.getAllWords());
        }
}
```

4.3 TrieNode

Listing 3: TrieNode.java

```
/*
By Robin Rai (100242165)
V.1.0.0
Created on 05/03/2020
*/
package dsacoursework2;

public class TrieNode {
        private TrieNode[] offspring;    //a is 0, b is 1, etc
        private boolean isEnd;    //if the node is the end of a word/key

        public TrieNode() {
                isEnd = false;
                this.offspring = new TrieNode[26];
                //I don’t think we’re getting any extra letters soon
        }

        public TrieNode[] getOffspring() {
                return this.offspring;
        }

        public boolean getIsEnd() {
                return this.isEnd;
        }

        public void setIsEnd(boolean input) {
                this.isEnd = input;
        }

}
```

4.4 AutoCompletionTrie

Listing 4: AutoCompletionTrie.java

```
/*
By Robin Rai (100242165)
V.1.0.0
Created on 05/03/2020
*/
package dsacoursework2;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;

public class AutoCompletionTrie {
        private AutoCompletionTrieNode root;

        public AutoCompletionTrie() {
                root = new AutoCompletionTrieNode();
        }
```

```java

        public AutoCompletionTrie(AutoCompletionTrieNode input) {
                root = input;
        }

        boolean add(String key, int frequency) {
                if (key.equals("")) {
                        return false;
                }
                int index;
                boolean alreadyIn = true;
                AutoCompletionTrieNode temp = root;
                for (int level = 0; level < key.length(); level++) {
                        //goes through all letters of the key
                        index = key.charAt(level) - 'a';
                        if (temp.getOffspring()[index] == null) {
                                //if the node for the current letter doesn't exist
                                temp.getOffspring()[index] = new
                                   ↪ AutoCompletionTrieNode();
                                //make a new one
                                alreadyIn = false;
                                //set the flag
                        }
                        temp = temp.getOffspring()[index];
                        //advanced temp node to one further down
                }
                if (alreadyIn && temp.getIsEnd()) {
                        //if everything was already done, return false
                        return false;
                }
                temp.setIsEnd(true);
                //sets last node for key as end.
                temp.setFrequency(frequency);
                return true;
        }

        boolean contains(String key) {
                int level;
                int length = key.length();
                int index;
                AutoCompletionTrieNode temp = root;
                for (level = 0; level < length; level++) {
                        //goes through all letters of the key
                        index = key.charAt(level) - 'a';
                        //index is current char's number
                        if (temp.getOffspring()[index] == null) {
                                //if the node for the current letter doesn't exist
                                return false;
                        }
                        temp = temp.getOffspring()[index];
                        //advance down a level
                }
                return (temp != null && temp.getIsEnd());
                //returns true only if it exists AND is a key - not just a prefix.
        }

        String outputBreadthFirstSearch() {
                if (root == null) {
                        return null;
                }
                Queue<AutoCompletionTrieNode> queue = new LinkedList<>();
                StringBuilder builder = new StringBuilder();
                queue.add(root);
                //starts with root
                while (!queue.isEmpty()) {
                        AutoCompletionTrieNode temp = queue.remove();
                        //takes first thing in queue
                        for (int i = 0; i < 26; i++) {
                                if (!(temp.getOffspring()[i] == null)) {
                                        //for all of it's children
                                        queue.add(temp.getOffspring()[i]);
                                        //add them to the queue so it goes through
                                           ↪ their
                                        // children - width first
                                        builder.append((char) (i + 97));
                                        //adds it to the output
                                }
                        }
                }
                return builder.toString();
        }

        String outputDepthFirstSearch() {
                //helper function, provides string to append to
                StringBuilder builder = new StringBuilder();
                return outputDepthFirstSearch(builder, root);
        }

        private String outputDepthFirstSearch(StringBuilder builder,
                                                                                  AutoComple
                                                                                     ↪ trie
                                                                                     ↪ {
                AutoCompletionTrieNode[] children = trieNode.getOffspring();
                //makes an array of all children in the tree
                for (int i = 0; i < 26; i++) {
                        //for all of a node's children
                        if (children[i] != null) {
                                //if there's a child
                                builder.append((char) (i + 97));
                                //add its character
                                outputDepthFirstSearch(builder, children[i]);
                                //call the function again on it's children
                        }
                }
                return builder.toString();
        }

        AutoCompletionTrie getSubTrie(String key) {

                AutoCompletionTrieNode temp = root;
                for (int level = 0; level < key.length(); level++) {
                        //for the key's length
                        int index = key.charAt(level) - 'a';
                        //goes to right character/node
                        if (temp.getOffspring()[index] == null) {
                                //checks if whole key is present
                                return null;
                        }
                        temp = temp.getOffspring()[index];
                }
                return new AutoCompletionTrie(temp);
                //returns only if it exists AND is a key - not just a prefix.
        }

        public ArrayList<String> getAllWords() {
                //helper function, is used just for the ArrayList
                // and String/stringBuilder
                ArrayList<String> listOfWords = new ArrayList<>();
                getAllWords(listOfWords, new StringBuilder(), root);
                return listOfWords;
        }

        private void getAllWords(ArrayList listOfWords, StringBuilder sb,
                                                         AutoCompletionTrieNode root)
                                                            ↪ {
                if (root.getIsEnd()) {
                        //checks if it's reached the end
                        System.out.println(sb.toString());
                        listOfWords.add(sb.toString());
                        System.out.println(root.getFrequency());
                        //print it and add it to the array
                }
                AutoCompletionTrieNode[] children = root.getOffspring();
                //get's current node's children
                for (int i = 0; i < children.length; i++) {
                        //for all the children
                        if (children[i] != null) {    //if it's a valid node
                                getAllWords(listOfWords, sb.append((char)
                                                (97 + i)), children[i]);
                                //call itself, with the child as the root
                                sb.setLength(sb.length() - 1);
                                //goes up a level - resets stringbuilder to right
                                   ↪ place
                        }
                }
        }

        public static class fullInfo {
                //objects to store a word's full info - the word
                // as well as it's frequency. Done for the comparator.
                private String key;
                private int freq;

                fullInfo(String key, int freq) {
                        this.key = key;
                        this.freq = freq;
                }

                fullInfo(String key, String freq) {
                        this.key = key;
                        this.freq = Integer.parseInt(freq);
                }

                int getFreq() {
                        return freq;
                }

                String getKey() {
                        return key;
                }
```

```java
        }

        static class testComp implements Comparator<fullInfo> {
                public int compare(fullInfo m1, fullInfo m2) {
                        //compares by frequency, then by string
                        if (m1.getFreq() < m2.getFreq()) {
                                return 1;
                        } else if (m1.getFreq() > m2.getFreq()) {
                                return -1;
                        } else {
                                return m1.getKey().compareTo(m2.getKey());
                        }
                }
        }

        public ArrayList<fullInfo> getAllInfo() {
                //helper function to provide recursive method with string and array
                ArrayList<fullInfo> listOfInfo = new ArrayList<>();
                getAllInfo(listOfInfo, new StringBuilder(), root);
                listOfInfo.sort(new testComp());
                return listOfInfo;
        }

        private void getAllInfo(ArrayList listOfInfo, StringBuilder sb,
                                                        AutoCompletionTrieNode root) {
                if (root.getIsEnd()) {
                        listOfInfo.add(new fullInfo(sb.toString(),
                           ↪ root.getFrequency()));
                        //if the end of the word's been reached, add it to the list
                }
                AutoCompletionTrieNode[] children = root.getOffspring();
                //get's current node's children
                for (int i = 0; i < children.length; i++) {
                        //for all the children
                        if (children[i] != null) {
                                //if it's a valid node
                                getAllInfo(listOfInfo, sb.append((char)
                                                (97 + i)), children[i]);
                                //call itself, with the child as the root
                                sb.setLength(sb.length() - 1);
                                //goes up a level - resets stringbuilder to right
                                   ↪ place
                        }
                }
        }

        public void addToTrie(String file) throws FileNotFoundException {
                //adds a dictionary file to a trie using trie's add function
                Scanner sc = new Scanner(new File(file));
                sc.useDelimiter("\n");
                String str;
                while (sc.hasNext()) {
                        str = sc.next();
                        str = str.trim();
                        str = str.toLowerCase();
                        //reads in and trims line
                        String[] parts = str.split(",");
                        //splits word and frequency, and adds them to the trie
                        //System.out.println("adding: " + parts[0] + " " + parts[1]);
                        //System.out.println(add(parts[0],
                             ↪ Integer.parseInt(parts[1])));
                        add(parts[0], Integer.parseInt(parts[1]));
                }
        }
}
```

4.5 AutoCompletionTrieNode

Listing 5: AutoCompletionTrieNode.java

```
/*
By Robin Rai (100242165)
V.1.0.0
Created on 05/03/2020
*/
package dsacoursework2;

public class AutoCompletionTrieNode {
        private AutoCompletionTrieNode[] offspring;
        //a is 0, b is 1, etc
        private boolean isEnd;
        //if the node is the end of a word/key
        private int frequency;
        //added frequency of word - is to be used at the end of a word like isEnd


        public AutoCompletionTrieNode() {
                isEnd = false;
                this.offspring = new AutoCompletionTrieNode[26];
                frequency = 0;
        }

        public AutoCompletionTrieNode[] getOffspring() {
                return this.offspring;
        }

        public boolean getIsEnd() {
                return this.isEnd;
        }

        public void setIsEnd(boolean input) {
                this.isEnd = input;
        }

        public int getFrequency() {
                return frequency;
        }

        public void setFrequency(int input) {
                this.frequency = input;
        }
}
```

4.6 AutoComplete

Listing 6: AutoComplete.java

```
/*
By Robin Rai (100242165)
V.1.0.0
Created on 05/03/2020
*/
```

```
package dsacoursework2;

import java.io.IOException;
import java.util.ArrayList;

import static dsacoursework2.DictionaryMaker.*;
import static dsacoursework2.AutoCompletionTrie.*;

public class AutoComplete {

        public static void autoCompletion(AutoCompletionTrie masterTrie,
           ↪ ArrayList<String> prefixes, String saveLocation) throws IOException {
                //generates top three results for each query, prints them to console
                   ↪ and file
                String infoString = "";
                //string to be written to file
                for (int prefix = 0; prefix < prefixes.size(); prefix++) {
                        //for every prefix
                        System.out.println("\nResults for: " + prefixes.get(prefix));
                        AutoCompletionTrie currentSubTrie =
                           ↪ masterTrie.getSubTrie(prefixes.get(prefix));
                        //gets the subTrie for the prefix
                        ArrayList<fullInfo> info = currentSubTrie.getAllInfo();
                        //gets the information for the subTrie
                        int totalFreq = 0;
                        //counter
                        for (int i = 0; i < info.size(); i++) {
                                totalFreq += info.get(i).getFreq();
                                //sums up frequencies to calculate ratios
                        }
                        infoString += prefixes.get(prefix) + ",";
                        //starts string to write to file
                        for (int i = 0; i < 3 && i < info.size(); i++) {
                                //gets top three or all available sub words,
                                   ↪ whichever's smallest
                                infoString += prefixes.get(prefix) +
                                   ↪ info.get(i).getKey() + "," +
                                   ↪ info.get(i).getFreq() + "," + (double)
                                   ↪ info.get(i).getFreq() / totalFreq + ",";
                                //creates a line with all words under that prefix
                                   ↪ along with their info to write to file
                                System.out.println(prefixes.get(prefix) +
                                   ↪ info.get(i).getKey() + " (probability " +
                                   ↪ (double) info.get(i).getFreq() / totalFreq +
                                   ↪ ")");
                                //prints said info to console as well
                        }
                        infoString = infoString.substring(0, infoString.length() - 1);
                        //removes comma from last result, unnecessary.
                        infoString += "\n";
                        //new line for new prefix
                }
                saveToFile(infoString, saveLocation);
                //calls saveToFile from DictionaryMaker to save to file
        }

        public static void main(String[] args) throws IOException {
                dsacoursework2.DictionaryMaker df = new
                   ↪ dsacoursework2.DictionaryMaker();
                df.formDictionary("src\\TextFiles\\lotr.csv",
                   ↪ "src\\TextFiles\\lotrDic.csv");
                AutoCompletionTrie completionTest = new AutoCompletionTrie();
                completionTest.addToTrie("src\\TextFiles\\lotrDic.csv");
                autoCompletion(completionTest,
                   ↪ readWordsFromCSV("src\\TextFiles\\lotrQueries.csv", "\n"),
                                "src\\TextFiles\\lotrMatches.csv");

        }

}
```

4.7 AutoComplete output

Listing 7: lotrMatches.csv

```
ab,about,17,0.5666666666666667,above,9,0.3,able,3,0.1
go,going,15,0.2777777777777778,go,13,0.24074074074074073,good,9,0.16666666666666666
the,the,460,0.6267029972752044,they,113,0.1539509536784741,them,50,0.0681198910081744
mer,merry,36,0.9473684210526315,merely,1,0.02631578947368421,merrily,1,0.02631578947368421
fro,frodo,27,0.4909090909090909,from,24,0.43636363636363634,front,4,0.07272727272727272
gr,great,13,0.19696969696969696,ground,12,0.18181818181818182,grass,10,0.15151515151515152
gol,goldberry,3,0.6,golden,2,0.4
sam,sam,18,1.0
```