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

Student number: 100225776. Blackboard ID: gny17hvu

Contents

1 Part 1: Form a Dictionary and Word Frequency Count

Part one required me to implement the methods formDictionary and saveToFile. For the former, after trying a few different methods with different data structures I've settled on using a single hash table to store the dictionary, as this allows me to store both the word and it's count as a pair. A hash table data structure also allows for a very good average time complexity for searching and inserting of $\theta(1)$ (constant time) while maintaining a decent space complexity of O(n).

The design of saveToFile followed a similar path as formDictionary, whereby I experimented with designing and implementing many different solutions simultaneously with those of formDictionary before settling on what I feel is the most efficient implementation. This final version takes the key-value pairs one by one from the hash table previously mentioned, forms a string from them and adds them to a list data structure. This list is then sorted and sent to the saveCollectionToFile method given.

In addition to the above methods, DictionaryFinder also contains readWordsFromCSV, which produces a list from a given CSV format file.

Algorithm 1 formDictionary($\mathbf{A}$,l) **return D**

Require: list $\mathbf{A}$ of length l containing strings.
Ensure: $\mathbf{D}$, the dictionary formed from $\mathbf{A}$.

$\mathbf{D} \leftarrow$ new map ▷ *initialise map to store dictionary*
for $i \leftarrow 1$ **to** l **do**
    $found \leftarrow$ **false**
    for each (k,v) pair $\in$ **D do**
        if $A_i := k$ **then**
            $v \leftarrow v + 1$ ▷ *add one to the value on this entry*
            $found \leftarrow$ **true**
            break
    if $found$:=**false then**
        $\mathbf{D} \leftarrow (k \leftarrow A_i,\ v \leftarrow 1)$ ▷ *add a new (k,v) pair to the map*
return D

In analysing formDictionary, we see that the fundamental operations are ($v \leftarrow v + 1$, $found \leftarrow$, break). The case we are interested in is the worst case - Order O. The Runtime Complexity function is as follows - the three fundamental operations only occur if there is a match therefore only a check of the map entry happens every loop, and this happens within a loop. Therefore for any given value of l and for any size of map D (represented as m), we get

$$t_{formDictionary} = \sum_{i=1}^{l}\sum_{j=1}^{m} 1 \tag{1}$$

For solving the summations, we first remove the rightmost summation and apply the summation rule

$$\sum_{i=1}^{p} 1 = p \tag{2}$$

on the innermost loop to give m, which leaves

$$t_{formDictionary} = \sum_{i=1}^{l} m \tag{3}$$

Then we deal with the remaining summation using the rule

$$\sum_{i=1}^{p} a = a \sum_{i=1}^{p} 1 = ap \tag{4}$$

to get

$$t_{formDictionary} = lm \tag{5}$$

where $m \leq l$.

By characterising the above Runtime Function we find that the worst case for this algorithm is Order nm or O(nm).

Algorithm 2 saveToFile(**D**)

Require: map **D**.

A $\leftarrow$ new collection ▷ *initialise collection to store strings*

for each (k,v) pair $\in$ **D** **do**

  $a \leftarrow k$ & "," & v ▷ *form string from k and v in requested format*

  A $\leftarrow a$

sort(**A**) ▷ *ensure A is sorted alphabetically with mergesort*

saveCollectionToFile(**A**,"A.csv") ▷ *pass collection into the provided method*

In analysing saveToFile, we see that the fundamental operation is either ($a \leftarrow k$ & "," & v, **A** $\leftarrow a$) or sort(**A**), so we will start with analysis of the former. The case we are interested in is the worst case - Order O. The Runtime Complexity function is as follows - the two fundamental operations each happen once per loop of D. Forming a string is a single operation and the complexity of adding it to a collection is dependant on the collection used, so we will assume a Linked List which inserts in constant time. Therefore if we assign the size of D to be l we get

$$t_{saveToFile} = \sum_{i=1}^{l} 2 \tag{6}$$

The constant on the right can be collapsed and using the rule

$$\sum_{i=1}^{p} 1 = p \tag{7}$$

we get

$$t_{saveToFile} = l \tag{8}$$

By characterising this first Runtime Function candidate, we find that the worst case for this algorithm is Order n or O(n). For the second candidate, we know that the best algorithm for sorting strings, mergesort, gives Order $n \log(n)$ worst-case performance, which is worse than Order n. Therefore from this analysis we can tell that the fundamental operation of the saveToFile algorithm is sort(A) and that it's characterisation is O($n \log(n)$).

2 Part 2: Implement a Trie Data Structure

Part two required the design and implementation of a Trie data structure comprised of TrieNodes to store a list of given words. Each node represents one character in a word and given words are chained together though storing lists of child nodes in each node. Methods were then written and implemented which store, split and recall words within the Trie. The TrieNode structure itself contains four variables:

1. *value* - This is a letter from a to z which this node represents.
2. *offspring* - This is a fixed list of 26 spaces, where each space can be assigned a child node representing a letter which would follow the current value in a given word.
3. *isKey* - This is a flag that is true when the current node is the end of a stored word and otherwise is false.
4. *level* - This is a number which shows how deep the current node is in the Trie structure.

The TrieNode also contains getter and setter methods, the following are the more complex examples of these:

- getOffspring(TrieNode **N**, char *c*) - We take 97 away from the ASCII value of c to get a numerical value between 0 and 25 for all letters of the alphabet, then if it exists, return the child node of **N** at that location in the offspring list.
- getAllOffspring(TrieNode **N**) - Add each node from **N**'s offspring list to a list and return it.
- setOffspring(TrieNode **N**, char *c*) - We take 97 away from the ASCII value of c and check the offspring list on **N** for an entry. If no entry exists we add a new node to the list and return true, otherwise we return false.

The Trie data structure holds only one variable, *root*, which is the address of the first TrieNode in the Trie and is assigned when the Trie is constructed. The Trie has a number of methods which control how nodes are added and and manipulated within it. I will detail these in full below.

Algorithm 3 add(**T K**,l) **return** f

Require: Trie **T**, string **K** of length l.
Ensure: f, a flag indicating if **K** was added to the Trie or not.

```
TrieNode C ← K.root                ▷ initialise node C with the value of this Trie's root
f ← false
for i ← 1 to l do
    k ← K_i                        ▷ char k holds the value of the current character in K
    TrieNode N ← getOffspring(C, k)
    if N := null then
        setOffspring(C, k)
        N ← getOffspring(C, k)
        setLevel(N, i + 1)         ▷ call setter for level variable on node N
        f ← true
    C ← N
if C := not null then
    setKey(C)       ▷ set the isKey flag on node C to true when the last letter of K is added
return f
```

Algorithm 4 contains(**T** **K**,l) **return** f

Require: Trie **T**, string **K** of length l.

Ensure: f, a flag indicating if **K** exists in **T** or not.

```
TrieNode C ← T.root
f ← false
for i ← 1 to l do
    char k ← K_i
    TrieNode N ← getOffspring(C, k)
    if N := null then return false
    C ← N
    f ← getKey(C)        ▷ set the value of f to the value of node C's getKey variable
return f
```

Algorithm 5 outputBreadthFirstSearch(**T**) **return S**

Require: Trie **T**.

Ensure: **S**, a string of characters taken from reading the Trie breadth-first.

```
V ← new list
l ← 1000
S ← new string with length of l
p ← 0        ▷ integer to hold position within S
V.add(T.root)
while V := not empty do
    TrieNode C ← V.remove()
    S_p ← C.value
    list L ← getAllOffspring(C)
    for each TrieNode N ∈ L do
        V.add(N)
    p ← p + 1
    if p := l then        ▷ increase the length of S by a multiple of 10 if it's too short
        String E ← S
        l ← l × 10
        S ← new string with length of l
        for i ← 1 to (l ÷ 10) do
            S_i ← E_i
return S
```

Algorithm 6 outputDepthFirstSearch(**T**) **return S**

Require: Trie **T**.
Ensure: **S**, a string of characters taken from reading the Trie breadth-first.

```
V ← new stack
l ← 1000
S ← new string with length of l
p ← 0
V.push(T.root)
while V := not empty do
    TrieNode C ← V.pop()
    list L ← getAllOffspring(C)
    m ← (length of N) −1
    while L:= not empty do
        V.push(N.get(m))
        m ← m − 1
    S_p ← C.value
    p ← p + 1
    if p := l then                ▷ increase the length of S by a multiple of 10 if it's too short
        String E ← S
        l ← l × 10
        S ← new string with length of l
        for i ← 1 to (l ÷ 10) do
            S_i ← E_i
return S
```

Algorithm 7 getSubTrie(**T P**,l) **return U**

Require: Trie **T**, string **P** of length l.
Ensure: **U**, the Subtrie rooted at **P**.

```
U ← new Trie
TrieNode S ← T.root
if P := empty then return U
for i ← 1 to l do
    c ← P_i
    TrieNode N ← getOffspring(S, c)
    if N := null then return S
    S ← N
U.root ← S return U
```

Algorithm 8 getAllWords(**T**) **return L**

Require: Trie **T**
Ensure: **L**, a list of strings formed from **T**.

```
L ← new list
V ← new stack
l ← 35                                      ▷ maximum length of an expected word
S ← new string of length l
list O ← getAllOffspring(T.root)
m ← (length of O) −1
while O := not empty do
    V.push(O.get(m)
    m ← m − 1
while V := not empty do
    TrieNode C ← V.pop()
    p ← C.level + 1                         ▷ position of the next character
    S_p ← C.value
    if C.isKey := true then
        F ← new string
        for i ← 1 to p do
            F ← F + S_i
        L.add(F)
    O ← getAllOffspring(C)
    m ← (length of O) −1
    while O := not empty do
        V.push(O.get(m))
        m ← m − 1
return L
```

3 Part 3: Word Auto Completion Application

The following main() algorithm relies on the data structures and algorithms explained in parts 1 and 2.

Algorithm 9 main()

```
                              ▷ 1. form a dictionary file of words and counts from the file lotr.csv.
D ← new DictionaryFinder
list L ← readWordsFromCSV(D, "lotr.csv")
D.formDictionary(L)
                              ▷ 2. construct a trie from the dictionary using your solution from part 2.
T ← new Trie
for each (k,v) pair ∈ D do
    T.add(k)
                              ▷ 3. load the prefixs from lotrQueries.csv
list P ← readWordsFromCSV("lotrQueries.csv")
                              ▷ 4. for each prefix query:
F ← new list                  ▷ list to store strings to be saved to file
for each p ∈ P do
                              ▷ 4.1. Recover all words matching the prefix from the trie.
    Trie S ← getSubTrie(T, p)
    list M ← getAllWords(S)
                              ▷ 4.2. Choose the three most frequent words and display to standard output.
    C ← new list              ▷ list of lists to store frequency and position of words found
    c ← 0
    if D.contains(p) then
        M.add("")
    l ← size of M
    for i ← 1 to l do
        string a ← p+ M.get(i)
        b ← D.get(a)          ▷ retrieve from dictionary the value for key matching the above string
        c ← c + b
        E ← new list
        E.add(b)
        E.add(i)
        C.add(E)
    while C := unsorted do
        using mergesort sort C where C.get(i).get(1) > C.get(i + 1).get(1)
    string H ← p
    for i ← 1 to l and i < 4 do
        z ← C.get(i).get(1) ÷ c
        Y ← p+ M.get(C.get(i).get(2))
        print Y + " (probability " +z+ ")"     ▷ see below for console output
        H ← H + "," + Y + "," +z
    F.add(H)
    print ""
                              ▷ 4.3. Write the results to lotrMatches.csv.
saveCollectionToFile(F, "lotrMatches.csv")
```

3.1 Console Output

about (probability 0.56666666)
above (probability 0.3)
able (probability 0.1)

going (probability 0.2777778)
go (probability 0.24074075)
good (probability 0.16666667)

the (probability 0.626703)
they (probability 0.15395096)
them (probability 0.06811989)

merry (probability 0.94736844)
merely (probability 0.02631579)
merrily (probability 0.02631579)

frodo (probability 0.4909091)
from (probability 0.43636364)
front (probability 0.07272727)

great (probability 0.1969697)
ground (probability 0.18181819)
grass (probability 0.15151516)

goldberry (probability 0.6)
golden (probability 0.4)

sam (probability 1.0)

Dictionary saved to file lotrMatches.csv

4 Code Listing

Listing 1: DictionaryFinder.java

```java
/******************************************************************************

 Project     : CMP-5014Y - Word Auto Completion with Tries:
               AutoCompletion.

 File        : DictionaryFinder.java

 Date        : Friday 06 March 2020

 Author      : Martin Siddons

 Description : This class fulfills the requirements of part 1. It supplies
 methods that read in a text document into a list of Strings, form a set of
 words that exist in the document and count how many times each word occurs,
 sort the list alphabetically and write those words and frequencies to file.

 History     :
 06/03/2020 - v1.0 - Initial setup, copied over readWordsFromCSV and
 saveCollectionToFile.
 07/03/2020 - v1.1 - Eventually settled on a implementation that works well in
 Java. Editing pseudocode to fit.
 ******************************************************************************/

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

public class DictionaryFinder {
    HashMap<String, Integer> dict;

    // Constructor
    public DictionaryFinder() {
        this.dict = null;
    }

    /**
     * Static method to read in files from a CSV file and output an ArrayList of
     * Strings, where each string is a line from the given file. Adapted from
     * code given by AJB.
     *
     * @param file : File to read in.
     * @return ArrayList of Strings read in from file.
     */
```

```java
    public static ArrayList<String> readWordsFromCSV(String file) {
        ArrayList<String> words = new ArrayList<>();
        try {
            Scanner sc = new Scanner(new File(file));
            sc.useDelimiter("[ ,\r]");
            String str;
            while (sc.hasNext()) {
                str = sc.next();
                str = str.trim();
                str = str.toLowerCase();
                words.add(str);
            }
        }
        catch (FileNotFoundException e) {
            System.out.println("Error: File not found. Aborting");
        }
        return words;
    }

    /**
     * Static method to write a given collection to file.
     * Adapted from code given by AJB.
     *
     * @param c    : Collection to save to file.
     * @param file : Name of file to be created.
     */
    public static void saveCollectionToFile(Collection<?> c, String file) {
        try {
            FileWriter fileWriter = new FileWriter(file);
            PrintWriter printWriter = new PrintWriter(fileWriter);
            for (Object w : c) {
                printWriter.println(w.toString());
            }
            printWriter.close();
        }
        catch (IOException e) {
            System.out.println("Error: Unable to write file. Aborting.");
        }
        System.out.println("\nDictionary saved to file " + file);
    }

    /**
     * Take an ArrayList of Strings and process each string into a dictionary.
     *
     * @param in : Arraylist to be formed into a dictionary.
     */
    public void formDictionary(List<String> in){
        this.dict = new HashMap<>(in.size());

```

```
        // For each string in the ArrayList, check if it exists in the
        // dictionary. If it doesn't, add it with a count of 1. If it does
        // exist, add 1 to the value on that entry.
        for (String s : in) {
            this.dict.put(s, this.dict.getOrDefault(s, 0) + 1);
        }
    }

    /**
     * Call the Dictionary previously created with formDictionary and produce a
     * formatted list. Ensure the list is sorted and send to
     * saveCollectionToFile.
     */
    public void saveToFile() {
        List<String> entries = new ArrayList<>();

        // Take each entry from the map, format it and add it to the list.
        for (Map.Entry<String, Integer> e : this.dict.entrySet()){
            entries.add(e.getKey() + "," + e.getValue());
        }
        Collections.sort(entries); // Sort the list alphabetically.

        // Call saveCollectionToFile to save the sorted, formatted list to file.
        String filename = "output.csv";
        saveCollectionToFile(entries, filename);
    }

    // Test Harness.
    public static void main(String[] args) {
        DictionaryFinder df = new DictionaryFinder();

        // 1. read text document into a list of strings
        ArrayList<String> in = readWordsFromCSV("testDocument.csv");

        // 2. form a set of words that exist in the document and count the
        // number of times each word occurs in a method called FormDictionary
        df.formDictionary(in);
        Collection<Integer> e = df.dict.values();
        System.out.print("counts for words in dictionary (before sort): ");
        System.out.println(e);

        // 3. sort the words alphabetically; and
        // 4. write the words and associated frequency to file.
        df.saveToFile();
```

```
        }
}
```

Listing 2: Trie.java

```java
/*****************************************************************************

 Project     : CMP-5014Y - Word Auto Completion with Tries :
               AutoCompletion.

 File        : Trie.java

 Date        : Thursday 12 March 2020

 Author      : Martin Siddons

 Description : Part 2 of the assignment is to make a Trie Data Structure to
 hold strings and write methods to manipulate the trie.

 History     : 12/03/2020 - Initial setup
 14/03/2020 - Completed Q1.
 13/05/2020 - Lost track on filling this in, just finished Q2.
 14/06/2020 - Fixed implementation of getAllWords for part 3.
 15/06/2020 - Finalised implementation by clearing up some methods.
 *****************************************************************************/

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Stack;

class TrieNode {
    private       char       value;
    private final TrieNode[] offspring;
    private       boolean    isKey;
    private       int        level; // track how deep this node is in the trie

    // Constructors
    public TrieNode() {
        this.offspring = new TrieNode[26];
    }

    public TrieNode(char c) {
        this.value     = c;
        this.offspring = new TrieNode[26];
    }

    public char getValue()       {return value;}

    public boolean getKey()      {return this.isKey;}

    public int getLevel()        {return this.level;}

    public void setKey()         {this.isKey = true;}
```

```java

    public void setLevel(int l) {this.level = l;}


    /**
     * Check if a given letter exists as a child on this node.
     *
     * @param c : Character to find linked from this trie.
     * @return : The TrieNode representing the given letter, or null if that
     * letter is not an offspring.
     */
    public TrieNode getOffspring(char c) {
        int pos = c - 97; // turn ASCII character value to numeric (so a = 0)
        if (pos < 0) {
            System.out.println("Invalid char num: " + c);
        }
        return this.offspring[pos];
    }

    /**
     * Retrieve a list of nodes of which the called-upon node is a parent.
     *
     * @return : List of TrieNodes that are offspring of the given node.
     */
    public LinkedList<TrieNode> getAllOffspring() {
        LinkedList<TrieNode> offspring = new LinkedList<>();

        for (TrieNode node : this.offspring) {
            if (node != null) {
                offspring.add(node);
            }
        }
        return offspring;
    }

    /**
     * Set a node for the given character, if it doesn't exist.
     *
     * @param c : Character to add to the trie.
     * @return : True if character has been added, false if not.
     */
    public boolean setOffspring(char c) {
        int pos = c - 97; // turn ASCII char to numeric value (where a = 0)
        // Create the node if it doesn't exist.
        if (this.offspring[pos] == null) {
            this.offspring[pos] = new TrieNode(c);
            return true;
```

```java
        }
        return false;
    }

}

public class Trie {
    private TrieNode root;

    // Constructor
    public Trie() {this.root = new TrieNode();}

    /**
     * Add a given string to the Trie.
     *
     * @param key : String to be added to the Trie.
     * @return : True if string has been added. If the string already exists,
     * return false.
     */
    public boolean add(String key) {
        TrieNode curNode = this.root;
        boolean wasAdded = false;

        for (int i = 0; i < key.length(); i++) {
            char curChar = key.charAt(i);

            // Check if the current character is in the trie.
            TrieNode next = curNode.getOffspring(curChar);

            // If not, add it. Switch to the node for this letter.
            if (next == null) {
                curNode.setOffspring(curChar);
                next = curNode.getOffspring(curChar);
                next.setLevel(i + 1);
                wasAdded = true;
            }
            curNode = next; // move to this node.
        }

        // Once at the end of the key String, set the final node to be a key.
        if (curNode != null) {
            curNode.setKey();
        }
        return wasAdded;
    }

    /**
     * Check if a given string exists in the trie.
     *
     * @param key : String to be checked.
```

```java
     * @return : True if the whole string exists (and is not a prefix or only
     * part of an existing word), false if not.
     */
    public boolean contains(String key) {
        TrieNode curNode = this.root;
        boolean isKey = false;

        for (int i = 0; i < key.length(); i++) {
            char c = key.charAt(i);
            TrieNode next = curNode.getOffspring(c);
            if (next == null) {
                return false;
            }
            curNode = next;
            isKey   = curNode.getKey();
        }
        return isKey;
    }

    /**
     * Traverse the trie in breadth-first order and return the result.
     *
     * @return : String of characters from reading the trie breadth-first.
     */
    public String outputBreadthFirstSearch() {
        LinkedList<TrieNode> toVisit = new LinkedList<>();
        int stringSize = 1000; // fixed starting size of 2000 bytes
        char[] trieString = new char[stringSize];
        int stringPos = 0;
        toVisit.add(this.root);

        while (! toVisit.isEmpty()) { // while the visit list is not empty:
            TrieNode curNode = toVisit.remove(); // get the next node from list
            trieString[stringPos] = curNode.getValue(); // add node to string
            toVisit.addAll(curNode.getAllOffspring()); // get all child nodes
            stringPos++;

            // If the string has reached it's limit, increase that by 10 times
            // and copy the old string over to the new one.
            if (stringPos == stringSize) {
                char[] temp = trieString;
                stringSize = stringSize * 10;
                trieString = new char[stringSize];
                for (int i = 0; i < stringSize / 10; i++) {
```

```
                    trieString[i] = temp[i];
                }
            }
        }
        return new String(trieString);
    }

    /**
     * Traverse the trie in depth-first order and return the result.
     *
     * @return String of characters from reading the trie depth-first
     */
    public String outputDepthFirstSearch() {
        Stack<TrieNode> toVisit = new Stack<>();
        int stringSize = 1000; // starting size for output string
        char[] trieString = new char[stringSize];
        int stringPos = 0;
        toVisit.push(this.root);

        while (! toVisit.isEmpty()) { // while the visit stack is not empty:
            TrieNode curNode = toVisit.pop(); // take the node off the stack

            // Retrieve all the offspring from the current node and add them to
            // the toVisit stack in reverse order (due to stack being FILO)
            LinkedList<TrieNode> node = curNode.getAllOffspring();
            while (! node.isEmpty()) {
                toVisit.push(node.removeLast());
            }
            trieString[stringPos] = curNode.getValue(); // add node to String
            stringPos++;

            // If the string has reached it's limit, increase that by 10x and
            // copy the old string over to the new one.
            if (stringPos == stringSize) {
                char[] temp = trieString;
                stringSize = stringSize * 10;
                trieString = new char[stringSize];
                for (int i = 0; i < temp.length; i++) {
                    trieString[i] = temp[i];
                }
            }
        }
        return new String(trieString);
    }

    /**
```

```java
     * returns a new trie rooted at the prefix, or null if the prefix is not
     * present in this trie.
     *
     * @param prefix : String which denotes the root of the returned trie.
     * @return : Trie structure of all branches of the given trie which branch
     * from the given prefix or empty trie if prefix not found.
     */
    public Trie getSubTrie(String prefix) {
        Trie subTrie = new Trie();
        TrieNode curNode = this.root;

        if (prefix.equals("")) {
            return subTrie;
        }

        // Find the prefix within the main trie while building the subtrie:
        for (int i = 0; i < prefix.length(); i++) {
            char c = prefix.charAt(i);

            TrieNode next = curNode.getOffspring(c);
            if (next == null) {
                return subTrie;
            }
            curNode = next;
        }
        subTrie.root = curNode;

        return subTrie;
    }

    /**
     * Returns all words held in the trie it is called on. This is written to be
     * as time efficient as possible without regard to memory usage.
     *
     * @return List of strings corresponding to each word in the trie. Returns
     * an empty list if there are no words in the trie it's called on.
     */
    List<String> getAllWords() {
        List<String> words = new ArrayList<>(); // words to be returned
        Stack<TrieNode> toVisit = new Stack<>(); // stack of Nodes to visit next
        int maxWordLength = 35; // there's no words over 35 letters in English
        char[] curWord = new char[maxWordLength]; // hold the word
```

```java
            ↪ being formed

        // Add the first node (root) to the toVisit stack.
        LinkedList<TrieNode> offspring = this.root.getAllOffspring();
        while (! offspring.isEmpty()) {
            toVisit.push(offspring.removeLast());
        }

        while (! toVisit.isEmpty()) {
            // Retrieve the next node and find it's position in the trie
            TrieNode curNode = toVisit.pop();
            int charPos = curNode.getLevel() + 1; // position of the next char
            curWord[charPos] = curNode.getValue(); // add current letter

            // Check if the current node is the end of a word and if so, add
            // the word to our list of words. We need to ensure we only copy the
            // letters that represent this word due to a persistent version of
            // curWord being used, however this is still quicker than making and
            // storing either a deep or shallow copy of curWord in a stack.
            if (curNode.getKey()) {
                StringBuilder foundWord = new StringBuilder();
                for (int i = 0; i <= charPos; i++) {
                    char curChar = curWord[i];
                    if (curChar != '\u0000') {
                        foundWord.append(curChar);
                    }
                }
                words.add(foundWord.toString());
            }

            // Retrieve all the offspring from the current node and add them to
            // the toVisit stack in reverse order (due to stack being FILO).
            offspring = curNode.getAllOffspring();
            while (! offspring.isEmpty()) {
                toVisit.push(offspring.removeLast());
            }
        }
        return words;
    }

    // Test Harness
    public static void main(String[] args) {
```

```java
        // Testing add(), should return "true" for all.
        Trie t = new Trie();
        System.out.println(t.add("cheers"));
        System.out.println(t.add("cheese"));
        System.out.println(t.add("chat"));
        System.out.println(t.add("cat"));
        System.out.println(t.add("can")); // REMOVE
        System.out.println(t.add("bat") + "\n");

        // Testing contains().
        System.out.println(t.contains("cheese")); // should return
            ↪ "true"
        System.out.println(t.contains("chose")); // should return
            ↪ "false"
        System.out.println(t.contains("ch") + "\n"); // should
            ↪ return "false"

        // Testing outputBreadthFirstSearch(), should return
            ↪ "bcaahttaetersse".
        System.out.println(t.outputBreadthFirstSearch() + "\n");

        // Testing outputDepthFirstSearch(), should return
            ↪ "batcathateersse".
        System.out.println(t.outputDepthFirstSearch() + "\n");

        // Testing getSubTrie, should return "aetersse".
        System.out.println(t.getSubTrie("ch").outputBreadthFirstSearch()
            ↪ +
                "\n");

        // Testing getAllWords, should return the list of words from
            ↪ add() test.
        System.out.println(t.getAllWords());
    }
}
```

Listing 3: AutoCompletion.java

```java
/******************************************************************************

 Project     : CMP-5014Y - Word Auto Completion with Tries :
               AutoCompletion.

 File        : AutoCompletion.java

 Date        : Thursday 14 May 2020

 Author      : Martin Siddons

 Description : Part 3 of the assignment is use Trie.java and
 DictionaryFinder.java to read in a large dictionary and a list of prefixes then
 output the three most common words on each prefix and save that to a file.

 History     : 14/05/2020 - Initial setup

 ******************************************************************************/

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class AutoCompletion {

    public static void main(String[] args) {
        // 1. form a dictionary file of words and counts from the file lotr.csv.
        DictionaryFinder lotr = new DictionaryFinder();
        lotr.formDictionary(DictionaryFinder.readWordsFromCSV("lotr.csv"));

        // 2. construct a trie from the dictionary using your solution from
        // part 2.
        Trie lotrTrie = new Trie();
        for (String k : lotr.dict.keySet()) {
            lotrTrie.add(k);
        }

        // 3. load the prefixs from lotrQueries.csv
        ArrayList<String> prefixs =
                DictionaryFinder.readWordsFromCSV("lotrQueries.csv");

        // 4. for each prefix query:
        List<String> toFile = new ArrayList<>();
        for (String s : prefixs) {
            // 4.1. Recover all words matching the prefix from the trie.
            Trie subTrie = lotrTrie.getSubTrie(s);
```

```java
            List<String> matches = new ArrayList<>(subTrie.getAllWords());

            // 4.2. Choose the three most frequent words and display to
            // standard output.
            // 2d array to store word frequency and position in matches list.
            // +1 ensures if we add the prefix, it won’t run out of bounds.
            int[][] count = new int[matches.size() + 1][2];
            float totalCount = 0;

            if (lotr.dict.containsKey(s)) { // ensure prefix is checked
                matches.add("");
            }

            for (int i = 0; i < matches.size(); i++) {
                String str = s + matches.get(i); // complete the word
                int wordCount = lotr.dict.get(str); // get this word’s count
                totalCount += wordCount; // inc the word counter
                count[i][0] = wordCount; // store the count
                count[i][1] = i; // store the word’s position
            }
            Arrays.sort(count, (a, b) -> Integer.compare(b[0], a[0]));
            StringBuilder topThree = new StringBuilder();
            topThree.append(s);

            // output and save top three results
            for (int i = 0; i < matches.size() && i < 3; i++) {
                float prob = count[i][0] / totalCount;
                String word = s + matches.get(count[i][1]);
                System.out.println(word + " (probability " + prob + ")");
                topThree.append(",").append(word).append(",").append(prob);
            }
            toFile.add(topThree.toString());
            System.out.println();
        }
        // 4.3. Write the results to lotrMatches.csv.
        DictionaryFinder.saveCollectionToFile(toFile, "lotrMatches.csv");
    }
}
```