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

Student number: 100251167. Blackboard ID: afz18mcu

Wednesday 29th April, 2020 13:24

Contents

1 Part 1: Form a Dictionary and Word Frequency Count

An algorithm that takes as input a list of words and returns a dictionary of words and the frequency count of each word after sorting them alphabetically.

1.1 Dictionary Finder

Algorithm 1 : Dictionary Finder

```
Require: $readWordsFromCSV$
Require: $AbsoluteFilePath$
  $wordRead \leftarrow readWordsFromCSV(AbsoluteFilePath)$
  $dictionaryFill \leftarrow new\ TreeMap$
  $sort(wordRead)$                                        ▷ sorting the words
  for each word in wordRead do
      if dictionaryFill contains word then
          $+1$ to occurrences of the word in dictionaryFill
      else
          put word at 1 in dictionaryFill
      end if
  end for
  for each key in keySet of dictionaryFill do
      print key and Number of times it occurred
  end for
```

1.2 Analyse the Worst Case Runtime Complexity

We want to form the worst case runtime complexity function for DictionaryFinder. The fundamental operation happens once on every loop. The time-complexity of the for loop is O(n) and Collections.sort is a modified mergesort, therefore it is n log(n), so for any given n, we perform.

$$f(n) = \sum_{i=1}^{n} log(n) + \sum_{i=1}^{n} 1 + \sum_{j=1}^{n} 1 \tag{1}$$

$$f(n) = \sum_{i=1}^{n} log(n) + 2\sum_{i=1}^{n} n \tag{2}$$

$$f(n) = nlog(n) + 2n \tag{3}$$

$$f(n) = O(nlog(n)) \tag{4}$$

2 Part 2: Implement a Trie Data Structure

Here we create a trie data structure to hold string keys and also create methods to manipulate the trie.

2.1 Add function

The add function adds a key to the trie and returns true if add was successful i.e. returns false if key already exist in the trie

Algorithm 2 : boolean add (String key)

```
Require: $root := new\ TrieNode$
Require: $offSpring$
Require: $getOffSpring$                                    ▷ gets the offSpring with the specific character
Require: $boolean\ isEnd$                                  ▷ checks if its the last character in the string
  $TrieNode\ rootTemp \leftarrow root$                     ▷ creating a copy of root to use in add
  for $i\ in\ length\ of\ key$ do                          ▷ iterate through the length of the String key
    $TrieNode\ nextNode \leftarrow rootTemp.getOffSpring(key(charAt(i))$
    if $nextNode\ is\ null$ then
      $nextNode \leftarrow TrieNode.makeNode(key(charAt(i))$
      $rootTemp.toCharArray(nextNode)$
    end if
    $rootTemp \leftarrow nextNode$
  end for
  $rootTemp.isEnd \leftarrow true$
  return $rootTemp.isEnd$
```

2.2 Contains function

The contains function returns true if the word passed is in the trie a whole word and not just a prefix.

Algorithm 3 : boolean contains(String key)

```
Require: $boolean\ isEnd$
Require: $root := new\ TrieNode$
Require: $getOffSpring$
  $TrieNode\ rootTemp \leftarrow root$
  if $rootTemp\ is\ null\ and\ rootTemp.isEnd$ then return true
  end if
  for $i\ in\ length\ of\ key$ do
    $TrieNode\ nodeNext \leftarrow rootTemp.getOffSpring(key(charAt(i))$
    if $nodeNext\ is\ null$ then return $false$
    else
      $rootTemp \leftarrow nodeNext$ return $true$
    end if
  end for
  return $true$
```

2.3 Breadth First Search

Returns a string representing a breadth first traversal

Algorithm 4 : String outputBreadthFirstSearch

```
Require: $root$
Require: $offSpring$                          ▷ an array of Trienodes with fixed size of 26
Require: $add()$                              ▷ add function
  $Queue < TrieNode > queue \leftarrow new\ LinkedList$ ▷ queue used to add items to the end of the linked-list
  $ArrayList < Character > characterArrayList \leftarrow new\ ArrayList$
  $queue.add(root)$
  while $!queue.isEmpty$ do
      $TrieNode\ currentNode \leftarrow queue.remove()$
      if $currentNode.offSpring\ != null$ then
          for $i\ in\ the\ length\ of\ the\ offSpring$ do
              if $currentNode.offSpring[i]\ != null$ then
                  $queue.add(currentNode.offSpring[i])$
              end if
          end for
          $characterArrayList.add(currentnode.charValueLetter)$
      end if
  end while
  $StringBuilder\ buildString \leftarrow new\ StringBuilder(characterArrayList.size)$
  for $Character\ characterBFS\ in\ characterArrayList$ do
      $buildString.append(characterBFS)$
  end for
  return $buildString.toString()$
```

2.4 Depth First Search

Returns a string representing a pre-order depth first traversal (which means it visits every node in the binary tree)

Algorithm 5 : String outputDepthFirstSearch

```
Require: $root$
Require: $offSpring$
  $Stack < TrieNode > s \leftarrow new\ Stack < TrieNode >$   ▷ Stack to keep track of which nodes need to be visited
  $Stack < Integer > sInt \leftarrow new\ Stack <>$
  $String\ word \leftarrow ""$   ▷ empty string, word
  $TrieNode\ tempTN \leftarrow root$
  if $root\ is\ null$ then
  return $null$
  end if
  $s.push(root)$   ▷ pushes the current node to the top
  while $!s.isEmpty$ do
    $tempTN \leftarrow s.pop$   ▷ removes the object which is at the top of the stack
    if $!sInt.isEmpty$ then
      $word \leftarrow word + (char)(sInt.pop +' a')$
    end if
    for $i\ in\ the\ length\ of\ the\ offSpring\ of\ tempTN$ do   ▷ where i=tempTN.offSpring.length - 1, i is greater than or equal to 0 and i decrements
      if $tempTN.offSpring[i]\ is\ not\ null$ then
        $s.push(tempTN.offSpring[i]$
        $sInt.push(i)$
      end if
    end for
  end while
  return $word$
```

2.5 SubTrie

Returns a new trie rooted at the prefix, or null if the prefix is not present in this trie

Algorithm 6 : Trie getSubTrie(String prefix)

```
Require: $root$
Require: $offSpring$
Require: $getOffSpring$
  $Trie\ subTrie \leftarrow new\ Trie$                                  ▷ creating a new trie
  $TrieNode\ tempTN \leftarrow root$
  if $"\ ".equals(prefix)\ or\ prefix.isEmpty$ then
  return $null$
  end if
  for $i\ in\ length\ of\ prefix$ do
      $int\ subTrieInt \leftarrow (int)prefix.charAt(i) - 97$ ▷ gets the character from the next node and converts
  it to an integer and then removes 97(ascii value for a)    ▷ basically converts a to z to 0 to 25 and then
  storing the next at that index
      if $tempTN.offSpring[subTrieInt]\ is\ null$ then
  return $null$
      end if
      $subTrie.root \leftarrow tempTN.getOffSpring(prefix.charAt(i))$
      $tempTN \leftarrow tempTN.offSpring[subTrieInt]$
  end for
  return $subTrie$
```

2.6 GetAllWords

Returns a list containing all words in the trie.

Algorithm 7 : List getAllWords

```
Require: $root$
Require: $offSpring$
Require: $charValueLetter$                ▷ The character that is stored inside the node
  $List\ allWords \leftarrow new\ ArrayList$
  for $each\ getWords(TrieNode)\ in\ offSpring\ of\ root$ do
    if $getWords\ is\ not\ null$ then
      $getWords(allWords, getWords.charValueLetter + "", getWords)$
    end if
  end for
  return $allWords$
```

Algorithm 8 : void getAllWords(List allWords, String words, TrieNode getWords)

```
Require: $add()$
Require: $offSpring$
Require: $charValueLetter$
Require: $isEnd$
  if $getWords.isEnd$ then
    $allWords.add(words)$
  end if
  for $each\ tempNode(TrieNode)\ in\ offSpring\ of\ getWords$ do
    if $tempNode\ is\ not\ null$ then
      $getAllWords(allWords, words + tempNode.charValueLetter, tempNode)$
    end if
  end for
```

3 Part 3: Word Auto Completion Application

Algorithm 9 : AutoCompletion

```
Require: $readWordsNewLine$    ▷ reads the file and gets rid of the new line
Require: $readFromCSV$
Require: $getWordFrequency$    ▷ same as subTrie but returns the frequency
Require: $add$    ▷ add function
Require: $getAllWords$
  $PrintWriter\ writeToFile \leftarrow new\ PrintWriter(lotrMatches.csv)$    ▷ writes the matches along with their probabilities to lotrMatches.csv
  $ArrayList < String > lotrQueries \leftarrow readWordsNewLine(lotrQueries)$    ▷ loads the queries into an arraylist
  $AutoCompletionTrie\ autoTrie \leftarrow new\ AutoCompletionTrie$
  for $each\ word\ in\ readFromCSV(lotr.csv)$ do
      $autoTrie.add(word)$    ▷ adds the words from lotr.csv into an AutoCompletionTrie
  end for
  for $each\ query\ in\ lotrQueries$ do
      $AutoCompletionTrie\ newSubTrie \leftarrow autoTrie.getSubTrie(query)$    ▷ push each query into another AutoCompletionTrie for efficiency
      $int\ totalFrequencies \leftarrow 0$
      $ArrayList < String > wordsToSort \leftarrow newSubTrie.getAllWords$    ▷ pushing all the words from newSubTrie into an arraylist
      for $i\ in\ the\ size\ of\ wordsToSort$ do    ▷ i increments]
          $wordsToSort.set(i, query + wordsToSort.get(i))$    ▷ returns the element at the given position, i in wordsToSort
      end for
      if $autoTrie.getWordFrequency(query)\ is\ greater\ than\ 0$ then    ▷ if the frequency of a query if more than 0
          $wordsToSort.add(query))$    ▷ adds the query to wordsToSort
      end if
      $wordsToSort.sort((String\ word1, String\ word2) - > -Integer.compare(autoTrie.getWordFrequency(wo$
$(autoTrie.getWordFrequency(word2))))$    ▷ sort the words using a lambda and Integer.compare to compare the frequency of the words
      for $each\ newWord\ in\ wordsToSort$ do
          $totalFrequencies = totalFrequencies + autoTrie.getWordFrequency(newWord)$    ▷ add the frequency of the newWord to the totalFrequenices for the specific query
      end for
      $writeToFile.print(query)$    ▷ prints the queries to lotrMatches.csv
      for $i\ in\ Math.min\ of\ 3\ or\ the\ size\ of\ wordsToSort$ do    ▷ only prints the top three frequencies or less if the prefix appears less times
          $writeToFile.print(wordsToSort.get(i))$    ▷ prints the words at i
          $writeToFile.print(float)autoTrie.getWordFrequency(Probability\ of\ the\ word)$    ▷ prints the probability of the number of times each word is likely in autoTrie(probability is calculated by dividing the number of occurences for each word by the totalFrequency)
      end for
  end for
  $writeToFile.close$
```

4 Code Listing

4.1 Part 1: DictionaryFinder

Listing 1: DictionaryFinder.java

```java
package CW2;

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

/**
 *
 * @author ajb
 */
public class DictionaryFinder {

    ArrayList<String> wordRead;
    TreeMap<String, Integer> dictionaryFill;

    public DictionaryFinder()
    {
    }

    /**
     * Reads all the words in a comma separated text document into an Array
     * @param
     */

    public static ArrayList<String> readWordsFromCSV(String file) throws FileNotFoundException
    {
        Scanner sc=new Scanner(new File(file));
        sc.useDelimiter(" |,");
        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 saveCollectionToFile(Collection<?> c,String file) throws IOException
    {
        PrintWriter printWriter = new PrintWriter(file);
        for(Object w: c)
        {
            printWriter.println(w.toString());
        }
        printWriter.close();
    }

    public void formDictionary() throws Exception
    {
        // reading the words from lotr.csv into the arraylist, wordRead
        wordRead = readWordsFromCSV
            ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\lotr.csv");
        // sort the wordRead alphabetically
        Collections.sort(wordRead);
        // dictionaryFill is a new TreeMap
        dictionaryFill = new TreeMap();

        //for each word in the arraylist, wordRead
        for(String word : wordRead)
        {
            //if dictionaryFill contains the word then add 1 to the
              ↪ frequency of the word
            if(dictionaryFill.containsKey(word))
            {
                dictionaryFill.put(word, dictionaryFill.get(word)+1);
            }
            // else it leaves the frequency at 1
            else
            {
                dictionaryFill.put(word, 1);
            }
        }

        // for each key in the keySet of dictionaryFill, print out the
          ↪ word with its frequency
        for(String key : dictionaryFill.keySet())
            System.out.println(key + " : " + dictionaryFill.get(key));
    }

    public void saveToFile() throws IOException
    {
        try (BufferedWriter writeToFile = new BufferedWriter(new
          ↪ FileWriter
        ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\Output.txt")))
        {
            for (Map.Entry<String, Integer> entry :
              ↪ this.dictionaryFill.entrySet()) {
                System.out.println("Word = " + entry.getKey() + ", Value
                  ↪ = " + entry.getValue());
                writeToFile.write(entry.getKey() + " = " +
                  ↪ entry.getValue() + " times, \n");
            }
        }
    }

    public static void main(String[] args) throws Exception
    {
        DictionaryFinder df=new DictionaryFinder();
        ArrayList<String> in=readWordsFromCSV
        ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\lotr.csv");
        df.formDictionary();
        df.saveToFile();
    }
}
```

4.2 Part 2: TrieNode

Listing 2: TrieNode.java

```java
package CW2;

import java.util.*;

public class TrieNode
{
    //the character that is stored inside the node
    public char charValueLetter;
    // variable to check if this is the last character in the string
    public boolean isEnd;
    // variable to check whether the node has been node
    public boolean visitedNode;
    //creating an array of fixed size 26
    public TrieNode[] offSpring;

    public TrieNode()
    {
        // initialize the array of fixed size 26
        this.offSpring = new TrieNode[26];
        //sets both the booleans to false
        this.isEnd = false;
        this.visitedNode = false;
        // iterates through the length of the array and sets all the
           ↪ elements to null
        for(int i = 0; i < offSpring.length; i++)
            offSpring[i] = null;
    }

    public TrieNode(char c)
    {
        charValueLetter = c;
    }

    public static TrieNode makeNode(char cNode)
    {
        //create a new TrieNode
        TrieNode newTrieN = new TrieNode();
        newTrieN.isEnd = false;
        newTrieN.charValueLetter = cNode;

        return newTrieN;
    }

    // getting the offSpring with the specific character
    public TrieNode getOffSpring (char cOffSpring)
    {
        for(int i = 0; i < offSpring.length; i++)
        {
            //make sure its not invalid or null and is equal to the the
               ↪ specific character, then adding and updating each item
               ↪ in the array
            if (offSpring[i] != null && offSpring[i].charValueLetter ==
               ↪ cOffSpring)
                return offSpring[i];
        }
        return null;
    }

    // gets character from next node and converts it to an integer and
       ↪ then removes 97(cos ascii for a)
    // basically turning a to z to 0 to 25 and then storing next at that
       ↪ index
    public void toCharArray(TrieNode nextNode)
    {
        int node = (int)nextNode.charValueLetter - 97;
        offSpring[node] = nextNode;
    }

    public char getCharValueLetter()
    {
        return charValueLetter;
    }
}
```

4.3 Part 2: Trie

Listing 3: Trie.java

```java
package CW2;

import java.util.*;
import java.lang.*;

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


    //adds a key to trie and returns true if the addition was successful
       ↪ i.e. returns false if key already exist in the Trie
    public boolean add(String key)
    {
        TrieNode rootTemp = root;
        for (int i = 0; i < key.length(); i++)
        {
            // charAt returns the character at the specified index
            TrieNode nextNode = rootTemp.getOffSpring(key.charAt(i));
            if (nextNode == null)
            {
                nextNode = TrieNode.makeNode(key.charAt(i));
                rootTemp.toCharArray(nextNode);
            }
            rootTemp = nextNode;
        }
        rootTemp.isEnd = true;
        return rootTemp.isEnd;
    }


    public boolean contains(String key)
    {
        TrieNode rootTemp = root;
        if (rootTemp != null && rootTemp.isEnd )
            return true;
        for(int i  = 0; i < key.length(); i++)
        {
            TrieNode nodeNext = rootTemp.getOffSpring((key.charAt(i)));
            if (nodeNext == null)
                return false;
            else
            {
                rootTemp = nodeNext;
                return true;
            }
        }
        return true;
    }

    // returns a string representing a breadth first traversal
    public String outputBreadthFirstSearch()
    {
        Queue<TrieNode> queue = new LinkedList<>();
        ArrayList<Character> characterArrayList = new ArrayList<>();
        queue.add(root);
        //while the linkedlist is full, we take the item in front of the queue and add it to the list
        // add items that aren't in the list to the back of the queue.
        while (!queue.isEmpty())
        {
            TrieNode currentNode = queue.remove();
            //check that offSpring has another element
            if (currentNode.offSpring != null)
            {
                for(int i = 0; i < currentNode.offSpring.length; i++)
                {
                    if (currentNode.offSpring[i] != null)
                        queue.add(currentNode.offSpring[i]);
                }
                characterArrayList.add(currentNode.charValueLetter);
            }
        }
        // StringBuilders is like an array of strings.
        // creating a new stringbuilder of the size of characterArrayList
        StringBuilder buildString = new StringBuilder(characterArrayList.size());
        //adding each character in characterArrayList to buildString
        for ( Character characterBFS : characterArrayList)
            buildString.append(characterBFS);
        return buildString.toString();
    }

    //returns a string representing a pre-order depth first traversal
    // make it recursive
    public String outputDepthFirstSearch()
    {
        if (root == null)
            return null;
        // create a stack for DFS i.e. which nodes to visit
        Stack<TrieNode> s = new Stack<TrieNode>();
        Stack<Integer> sInt = new Stack<>();
        String word = "";
        // push the current node to top
        s.push(root);
        TrieNode tempTN = root;
        while (!s.isEmpty())
        {
            // removes the object at the top of the stack
            tempTN = s.pop();
            if(!sInt.isEmpty())
                word = word + (char)(sInt.pop() + 'a');
            for (int i = tempTN.offSpring.length - 1; i >= 0; i--)
            {
                if (tempTN.offSpring[i] != null)
                {
                    s.push(tempTN.offSpring[i]);
                    sInt.push(i);
                }
            }
        }
        return word;
    }


    // returns a new Trie rooted at the prefix
    public Trie getSubTrie (String prefix)
    {
        //create a new TrieNode
        Trie subTrie = new Trie();
        TrieNode tempTN = root;
        if (" ".equals(prefix) || prefix.isEmpty())
            return null;
        for (int i = 0; i < prefix.length(); i++)
        {
            // same concept as toCharArray
            int subTrieInt = (int)prefix.charAt(i) - 97;
            if(tempTN.offSpring[subTrieInt] == null)
            {
                return null;
            }
            subTrie.root = tempTN.getOffSpring(prefix.charAt(i));
            tempTN = tempTN.offSpring[subTrieInt];
        }
        return subTrie;
        // searches letters in trie if yes creates a new trie
        // assign root to characters of the prefix
    }

    // returns a list containing all words in the Trie
    public List getAllWords()
    {
        List allWords = new ArrayList();
        for (TrieNode getWords : root.offSpring )
        {
            if (getWords != null)
                getAllWords(allWords, getWords.charValueLetter + "",
                    ↪ getWords);
        }
        return allWords;
    }

    public void getAllWords(List allWords, String words, TrieNode
       ↪ getWords)
    {
        if(getWords.isEnd)
            allWords.add(words);
        for (TrieNode tempNode : getWords.offSpring)
        {
            if (tempNode != null)
                getAllWords(allWords, words + tempNode.charValueLetter,
                    ↪ tempNode);
        }
    }

    public static void main(String[] args) {
        Trie newTrie = new Trie();
        newTrie.add("cheers");
        newTrie.add("cheese");
        newTrie.add("chat");
        newTrie.add("cat");
        newTrie.add("bat");

        System.out.println(newTrie.outputBreadthFirstSearch());
        System.out.println(newTrie.outputDepthFirstSearch());
        System.out.println(newTrie.getSubTrie("ch").getAllWords());
        System.out.println(newTrie.getAllWords());

    }
}
```

4.4 Part 3: AutoCompletionTrieNode

Listing 4: AutoCompletionTrieNode.java

```java
package CW2;

public class AutoCompletionTrieNode
{
    //the character that is stored inside the node
    public char charValueLetter;
    // variable to check if this is the last character in the string
    public boolean isEnd;
    // variable to check whether the node has been node
    public boolean visitedNode;
    //creating an array of fixed size 26
    public AutoCompletionTrieNode[] offSpring;
    int frequency = 0;

    public AutoCompletionTrieNode()
    {
        // initialize the array of fixed size 26
        this.offSpring = new AutoCompletionTrieNode[26];
        //sets both the booleans to false
        this.isEnd = false;
        this.visitedNode = false;
        // iterates through the length of the array and sets all the
           ↪ elements to null
        for(int i = 0; i < offSpring.length; i++)
            offSpring[i] = null;
    }

    public AutoCompletionTrieNode(char c)
    {
        charValueLetter = c;
    }

    public static AutoCompletionTrieNode makeNode(char cNode)
    {
        //create a new TrieNode
        AutoCompletionTrieNode newTrieN = new AutoCompletionTrieNode();
        newTrieN.isEnd = false;
        newTrieN.charValueLetter = cNode;

        return newTrieN;
    }

    // getting the offSpring with the specific character
    public AutoCompletionTrieNode getOffSpring (char cOffSpring)
    {
        for(int i = 0; i < offSpring.length; i++)
        {
            //make sure its not invalid or null and is equal to the the specific character, then adding and updating each item in the array
            if (offSpring[i] != null && offSpring[i].charValueLetter == cOffSpring)
                return offSpring[i];
        }
        return null;
    }

    // gets character from next node and converts it to an integer and then removes 97(cos ascii for a)
    // basically turning a to z to 0 to 25 and then storing next at that index
    public void toCharArray(AutoCompletionTrieNode nextNode)
    {
        int node = (int)nextNode.charValueLetter - 97;
        offSpring[node] = nextNode;
    }

    public char getCharValueLetter()
    {
        return charValueLetter;
    }
}
```

4.5 Part 3: AutoCompletionTrie

Listing 5: AutocompletionTrie.java

```java
package CW2;

import java.util.*;
import java.lang.*;

public class AutoCompletionTrie
{
    AutoCompletionTrieNode root = new AutoCompletionTrieNode();
    int wordFrequency;


    //adds a key to trie and returns true if the addition was successful i.e. returns false if key already exist in the Trie
    public boolean add(String key /*,int addFrequency*/)
    {
        AutoCompletionTrieNode rootTemp = root;
        for (int i = 0; i < key.length(); i++)
        {
            // charAt returns the character at the specified index
            AutoCompletionTrieNode nextNode = rootTemp.getOffSpring(key.charAt(i));
            if (nextNode == null)
            {
                nextNode = AutoCompletionTrieNode.makeNode(key.charAt(i));
                rootTemp.toCharArray(nextNode);
            }
            rootTemp = nextNode;
        }
        rootTemp.isEnd = true;
        rootTemp.frequency++;
        //rootTemp.frequency = addFrequency;
        return rootTemp.isEnd;
    }

    public boolean contains(String key)
    {
        AutoCompletionTrieNode rootTemp = root;
        if (rootTemp != null && rootTemp.isEnd )
            return true;
        for(int i  = 0; i < key.length(); i++)
        {
            AutoCompletionTrieNode nodeNext = rootTemp.getOffSpring((key.charAt(i)));
            if (nodeNext == null)
                return false;
            else
            {
                rootTemp = nodeNext;
                return true;
            }
        }
        return true;
    }

    // returns a string representing a breadth first traversal
    public String outputBreadthFirstSearch()
    {
        Queue<AutoCompletionTrieNode> queue = new LinkedList<>();
        ArrayList<Character> characterArrayList = new ArrayList<>();
        queue.add(root);
        //while the linkedlist is full, we take the item in front of the
           ↪ queue and add it to the list
        // add items that aren't in the list to the back of the queue.
        while (!queue.isEmpty())
        {
            AutoCompletionTrieNode currentNode = queue.remove();
            //check that offSpring has another element
            if (currentNode.offSpring != null)
            {
                for(int i = 0; i < currentNode.offSpring.length; i++)
                {
                    if (currentNode.offSpring[i] != null)
                        queue.add(currentNode.offSpring[i]);
                }
                characterArrayList.add(currentNode.charValueLetter);
            }
        }
        // StringBuilders is like an array of strings.
        // creating a new stringbuilder of the size of characterArrayList
        StringBuilder buildString = new
           ↪ StringBuilder(characterArrayList.size());
        //adding each character in characterArrayList to buildString
        for ( Character characterBFS : characterArrayList)
            buildString.append(characterBFS);
        return buildString.toString();
    }

    //returns a string representing a pre-order depth first traversal
    // make it recursive
    public String outputDepthFirstSearch()
    {
        if (root == null)
            return null;
        // create a stack for DFS i.e. which nodes to visit
        Stack<AutoCompletionTrieNode> s = new
```

```
            ↪ Stack<AutoCompletionTrieNode>();
        Stack<Integer> sInt = new Stack<>();
        String word = "";
        // push the current node to top
        s.push(root);
        AutoCompletionTrieNode tempTN = root;
        while (!s.isEmpty())
        {
            // removes the object at the top of the stack
            tempTN = s.pop();
            if(!sInt.isEmpty())
                word = word + (char)(sInt.pop() + 'a');
            for (int i = tempTN.offSpring.length - 1; i >= 0; i--)
            {
                if (tempTN.offSpring[i] != null)
                {
                    s.push(tempTN.offSpring[i]);
                    sInt.push(i);
                }
            }
        }
        return word;
    }

    // returns a new Trie rooted at the prefix
    public AutoCompletionTrie getSubTrie (String prefix)
    {
        //create a new TrieNode
        AutoCompletionTrie subTrie = new AutoCompletionTrie();
        AutoCompletionTrieNode tempTN = root;
        if (" ".equals(prefix) || prefix.isEmpty())
            return null;
        for (int i = 0; i < prefix.length(); i++)
        {
            // same concept as toCharArray
            int subTrieInt = (int)prefix.charAt(i) - 97;
            if(tempTN.offSpring[subTrieInt] == null)
            {
                return null;
            }
            subTrie.root = tempTN.getOffSpring(prefix.charAt(i));
            tempTN = tempTN.offSpring[subTrieInt];
        }
        return subTrie;
        // searches letters in trie if yes creates a new trie
        // assign root to characters of the prefix
    }

    public ArrayList<String> getAllWords()
    {
        ArrayList<String> allWords = new ArrayList();
        for (AutoCompletionTrieNode getWords : root.offSpring )
        {
            if (getWords != null)
                getAllWords(allWords, getWords.charValueLetter + "",
                  ↪ getWords);
        }
        return allWords;


    }

    public void getAllWords(List allWords, String words,
      ↪ AutoCompletionTrieNode getWords)
    {
        if(getWords.isEnd)
            allWords.add(words);
        for (AutoCompletionTrieNode tempNode : getWords.offSpring)
        {
            if (tempNode != null)
                getAllWords(allWords, words + tempNode.charValueLetter,
                  ↪ tempNode);
        }
    }

    // same concept as getSubTrie except in this case it returns the
      ↪ int, frequency rather than the subTrie.
    public int getWordFrequency(String word)
    {
        // creating anew autocompletiontrie called subTrie
        AutoCompletionTrie subTrie = new AutoCompletionTrie();
        // creating a temporary TrieNode
        AutoCompletionTrieNode tempTN = root;
        if (" ".equals(word) || word.isEmpty())
            return 0;
        for (int i = 0; i < word.length(); i++)
        {
            // same concept as toCharArray
            int subTrieInt = (int)word.charAt(i) - 97;
            if(tempTN.offSpring[subTrieInt] == null)
            {
                return 0;
            }
            subTrie.root = tempTN.getOffSpring(word.charAt(i));
            tempTN = tempTN.offSpring[subTrieInt];
        }
        // returns the frequency
        return tempTN.frequency;
    }
}
```

4.6 Part 3: AutoCompletion

Listing 6: AutoCompletion.java

```java
package CW2;

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

import static CW2.DictionaryFinder.readWordsFromCSV;

public class AutoCompletion
{
    // reads the queries from lotrQueries.csv and uses the new line character as a delimiter.
    public static ArrayList<String> readWordsNewLine(String file) throws FileNotFoundException
    {
        Scanner sc=new Scanner(new File(file));
        sc.useDelimiter("\n");
        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 main(String[] args) throws Exception
    {
        // PrintWriter prints the matches and their probabilities to the specified
        PrintWriter writeToFile= new PrintWriter
        ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\lotrMatches.csv
        // reading and pushing all the queries from lotrQueries.csv to an arraylist called lotrQueries
        ArrayList<String> lotrQueries = readWordsNewLine
        ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\lotrQueries.csv
        //System.out.println(lotrQueries);
        // creating a new AutoCompletionTrie called autoTrie
        AutoCompletionTrie autoTrie = new AutoCompletionTrie();
        // for each word in the lotr.csv, add it to autoTrie
        for (String word: readWordsFromCSV
        ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\lotr.csv"))
            autoTrie.add(word);
        //System.out.println(autoTrie.getAllWords());
        //System.out.println(autoTrie.getWordFrequency("yellow"));
        // for each query in the arraylist, lotrQueries
        for (String query: lotrQueries)
        {
            // push the query into another AutoCompletionTrie called
              ↪ newSubTrie
            AutoCompletionTrie newSubTrie = autoTrie.getSubTrie(query);
            // initializing totalFrequencies to 0;
            int totalFrequencies=0;
            // prints out all the words in newSubTrie
            //System.out.println(newSubTrie.getAllWords());
            // pushing all the words in newSubTrie to an arraylist
              ↪ called, wordsToSort
            ArrayList<String> wordsToSort = newSubTrie.getAllWords();
            // for loop that goes through the size of the arraylist,
              ↪ wordsToSort
            for (int i =0; i<wordsToSort.size(); i++)
                // returns the element at the given index after going
                  ↪ through wordsToSort
                wordsToSort.set(i, query+wordsToSort.get(i));
            // if the frequency of a query is more than 0, it adds it to
              ↪ the arraylist, wordsToSort
            if(autoTrie.getWordFrequency(query) > 0)
                wordsToSort.add(query);
            // sort the words using a lambda and Integer.compare to
              ↪ compare the frequency of the words and then sort.
            wordsToSort.sort((String word1, String
              ↪ word2)->-Integer.compare(autoTrie.getWordFrequency(word1),
              ↪ (autoTrie.getWordFrequency(word2))));
            //for each newWord in wordsToSort add the frequency of the
              ↪ newWord to the totalFrequencies
            for (String newWord : wordsToSort)
            {
                totalFrequencies += autoTrie.getWordFrequency(newWord);
            }
            // writeToFile.print() is used to write to lotrMatches.csv
            writeToFile.print(query+",");
            // for loop that only prints the top three frequencies or
              ↪ less if the prefix appears less times
            for (int i = 0; i < Math.min(3,wordsToSort.size()); i++)
            {
                writeToFile.print(wordsToSort.get(i)+",");
                // prints the probabilities of the number of time each
                  ↪ word is likely to occur
                writeToFile.print((float)autoTrie.getWordFrequency
                        (wordsToSort.get(i))/totalFrequencies+",");
            }
            // makes sure each query and its occurrences and its
              ↪ probabilities
            writeToFile.println();
            }
            writeToFile.close();
        }
}
```

5 Answer

5.1 Part 3: lotrMatches.csv

Listing 7: lotrMatches.csv

```
ab,about,0.56666666,above,0.3,able,0.1,
go,going,0.2777778,go,0.24074075,good,0.16666667,
the,the,0.626703,they,0.15395096,them,0.06811989,
mer,merry,0.94736844,merely,0.02631579,merrily,0.02631579,
fro,frodo,0.4909091,from,0.43636364,front,0.07272727,
gr,great,0.1969697,ground,0.18181819,grass,0.15151516,
gol,goldberry,0.6,golden,0.4,
sam,sam,1.0,
```