

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	2
1.1	Dictionary Finder	2
1.2	Analyse the Worst Case Runtime Complexity	2
2	Part 2: Implement a Trie Data Structure	3
2.1	Add function	3
2.2	Contains function	3
2.3	Breadth First Search	4
2.4	Depth First Search	5
2.5	SubTrie	6
2.6	GetAllWords	7
3	Part 3: Word Auto Completion Application	8
4	Code Listing	10
4.1	Part 1: DictionaryFinder	10
4.2	Part 2: TrieNode	13
4.3	Part 2: Trie	15
4.4	Part 3: AutoCompletionTrieNode	19
4.5	Part 3: AutoCompletionTrie	21
4.6	Part 3: AutoCompletion	25
5	Answer	28
5.1	Part 3: lotrMatches.csv	28

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*)

\triangleright *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.

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

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

$$f(n) = n \log(n) + 2n \quad (3)$$

$$f(n) = O(n \log(n)) \quad (4)$$

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 Trianodes with fixed size of 26

Require: *add()*

▷ add function

Queue < *TrieNode* > *queue* ← *new LinkedList* ▷ queue used to add items to the end of the linked-list

ArrayList < *Character* > *characterArrayList* ← *new ArrayList*

queue.add(root)

while *!queue.isEmpty* **do**

TrieNode currentNode ← *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 ← *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 \leftarrow *TrieNode* \triangleright *s* \leftarrow *new Stack* \leftarrow *TrieNode* \triangleright Stack to keep track of which nodes need to be visited

Stack \leftarrow *Integer* \triangleright *sInt* \leftarrow *new Stack* \leftarrow $\langle \rangle$

String word \leftarrow "" \triangleright empty string, word

TrieNode tempTN \leftarrow *root*

if *root is null* **then**

return *null*

end if

s.push(root) \triangleright pushes the current node to the top

while *!s.isEmpty* **do**

tempTN \leftarrow *s.pop* \triangleright 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** \triangleright 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 $< \text{String} >$ *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 $< \text{String} >$ *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) \rightarrow $-Integer.compare(autoTrie.getWordFrequency(word1), autoTrie.getWordFrequency(word2))$) ▷ 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 totalFrequencies 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 occurrences for each word by the totalFrequency)
 end for
end for
writeToFile.close

4 Code Listing

4.1 Part 1: DictionaryFinder

Listing 1: DictionaryFinder.java

```
1 package CW2;
2
3 import java.io.*;
4 import java.util.*;
5
6 /**
7  *
8  * @author ajb
9  */
10 public class DictionaryFinder {
11
12     ArrayList<String> wordRead;
13     TreeMap<String, Integer> dictionaryFill;
14
15     public DictionaryFinder()
16     {
17     }
18
19     /**
20      * Reads all the words in a comma separated text document into an
21      *   ↪ Array
22      * @param
23      */
24     public static ArrayList<String> readWordsFromCSV(String file) throws
25         ↪ FileNotFoundException
26     {
27         Scanner sc=new Scanner(new File(file));
28         sc.useDelimiter(" |,");
29         ArrayList<String> words=new ArrayList<>();
30         String str;
31         while(sc.hasNext())
32         {
33             str=sc.next();
34             str=str.trim();
35             str=str.toLowerCase();
36             words.add(str);
37         }
38         return words;
39     }
40
41     public static void saveCollectionToFile(Collection<?> c,String file)
42         ↪ throws IOException
43     {
```

```

42     PrintWriter printWriter = new PrintWriter(file);
43     for(Object w: c)
44     {
45         printWriter.println(w.toString());
46     }
47     printWriter.close();
48 }
49
50 public void formDictionary() throws Exception
51 {
52     // reading the words from lotr.csv into the arraylist, wordRead
53     wordRead = readWordsFromCSV
54         ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\lotr.csv");
55     // sort the wordRead alphabetically
56     Collections.sort(wordRead);
57     // dictionaryFill is a new TreeMap
58     dictionaryFill = new TreeMap();
59
60     //for each word in the arraylist, wordRead
61     for(String word : wordRead)
62     {
63         //if dictionaryFill contains the word then add 1 to the
64         ↪ frequency of the word
65         if(dictionaryFill.containsKey(word))
66         {
67             dictionaryFill.put(word, dictionaryFill.get(word)+1);
68         }
69         // else it leaves the frequency at 1
70         else
71         {
72             dictionaryFill.put(word, 1);
73         }
74     }
75
76     // for each key in the keySet of dictionaryFill, print out the
77     ↪ word with its frequency
78     for(String key : dictionaryFill.keySet())
79         System.out.println(key + " : " + dictionaryFill.get(key));
80 }
81
82 public void saveToFile() throws IOException
83 {
84     try (BufferedWriter writeToFile = new BufferedWriter(new
85         ↪ FileWriter
86         ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\Output.txt")))
87     {
88         for (Map.Entry<String, Integer> entry :
89             ↪ this.dictionaryFill.entrySet()) {
90             System.out.println("Word = " + entry.getKey() + ", Value
91                 ↪ = " + entry.getValue());

```

```

87         writeToFile.write(entry.getKey() + " = " +
88             ↪ entry.getValue() + " times, \n");
89     }
90 }
91
92 public static void main(String[] args) throws Exception
93 {
94     DictionaryFinder df=new DictionaryFinder();
95     ArrayList<String> in=readWordsFromCSV
96     ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\lotr.csv");
97     df.formDictionary();
98     df.saveToFile();
99 }
100 }

```

4.2 Part 2: TrieNode

Listing 2: TrieNode.java

```
1 package CW2;
2
3 import java.util.*;
4
5 public class TrieNode
6 {
7     //the character that is stored inside the node
8     public char charValueLetter;
9     // variable to check if this is the last character in the string
10    public boolean isEnd;
11    // variable to check whether the node has been node
12    public boolean visitedNode;
13    //creating an array of fixed size 26
14    public TrieNode[] offSpring;
15
16    public TrieNode()
17    {
18        // initialize the array of fixed size 26
19        this.offSpring = new TrieNode[26];
20        //sets both the booleans to false
21        this.isEnd = false;
22        this.visitedNode = false;
23        // iterates through the length of the array and sets all the
24        // ↪ elements to null
25        for(int i = 0; i < offSpring.length; i++)
26            offSpring[i] = null;
27    }
28
29    public TrieNode(char c)
30    {
31        charValueLetter = c;
32    }
33
34    public static TrieNode makeNode(char cNode)
35    {
36        //create a new TrieNode
37        TrieNode newTrieN = new TrieNode();
38        newTrieN.isEnd = false;
39        newTrieN.charValueLetter = cNode;
40
41        return newTrieN;
42    }
43
44    // getting the offSpring with the specific character
45    public TrieNode getOffSpring (char cOffSpring)
46    {
```

```

46     for(int i = 0; i < offSpring.length; i++)
47     {
48         //make sure its not invalid or null and is equal to the the
         ↳ specific character, then adding and updating each item
         ↳ in the array
49         if (offSpring[i] != null && offSpring[i].charValueLetter ==
         ↳ cOffSpring)
50             return offSpring[i];
51     }
52     return null;
53 }
54
55 // gets character from next node and converts it to an integer and
         ↳ then removes 97(cos ascii for a)
56 // basically turning a to z to 0 to 25 and then storing next at that
         ↳ index
57 public void toCharArray(TrieNode nextNode)
58 {
59     int node = (int)nextNode.charValueLetter - 97;
60     offSpring[node] = nextNode;
61 }
62
63 public char getCharValueLetter()
64 {
65     return charValueLetter;
66 }
67 }

```

4.3 Part 2: Trie

Listing 3: Trie.java

```
1 package CW2;
2
3 import java.util.*;
4 import java.lang.*;
5
6 public class Trie {
7     TrieNode root = new TrieNode();
8
9
10    //adds a key to trie and returns true if the addition was successful
11    ↪ i.e. returns false if key already exist in the Trie
12    public boolean add(String key)
13    {
14        TrieNode rootTemp = root;
15        for (int i = 0; i < key.length(); i++)
16        {
17            // charAt returns the character at the specified index
18            TrieNode nextNode = rootTemp.getOffSpring(key.charAt(i));
19            if (nextNode == null)
20            {
21                nextNode = TrieNode.makeNode(key.charAt(i));
22                rootTemp.toCharArray(nextNode);
23            }
24            rootTemp = nextNode;
25        }
26        rootTemp.isEnd = true;
27        return rootTemp.isEnd;
28    }
29
30    public boolean contains(String key)
31    {
32        TrieNode rootTemp = root;
33        if (rootTemp != null && rootTemp.isEnd )
34            return true;
35        for(int i = 0; i < key.length(); i++)
36        {
37            TrieNode nodeNext = rootTemp.getOffSpring((key.charAt(i)));
38            if (nodeNext == null)
39                return false;
40            else
41            {
42                rootTemp = nodeNext;
43                return true;
44            }
45        }
46    }
```

```

46         return true;
47     }
48
49     // returns a string representing a breadth first traversal
50     public String outputBreadthFirstSearch()
51     {
52         Queue<TrieNode> queue = new LinkedList<>();
53         ArrayList<Character> characterArrayList = new ArrayList<>();
54         queue.add(root);
55         //while the linkedlist is full, we take the item in front of the
56         //    ↪ queue and add it to the list
57         // add items that aren't in the list to the back of the queue.
58         while (!queue.isEmpty())
59         {
60             TrieNode currentNode = queue.remove();
61             //check that offSpring has another element
62             if (currentNode.offSpring != null)
63             {
64                 for(int i = 0; i < currentNode.offSpring.length; i++)
65                 {
66                     if (currentNode.offSpring[i] != null)
67                         queue.add(currentNode.offSpring[i]);
68                 }
69                 characterArrayList.add(currentNode.charValueLetter);
70             }
71             // StringBuilders is like an array of strings.
72             // creating a new stringbuilder of the size of characterArrayList
73             StringBuilder buildString = new
74                 ↪ StringBuilder(characterArrayList.size());
75             //adding each character in characterArrayList to buildString
76             for (Character characterBFS : characterArrayList)
77                 buildString.append(characterBFS);
78             return buildString.toString();
79         }
80
81         //returns a string representing a pre-order depth first traversal
82         // make it recursive
83         public String outputDepthFirstSearch()
84         {
85             if (root == null)
86                 return null;
87             // create a stack for DFS i.e. which nodes to visit
88             Stack<TrieNode> s = new Stack<TrieNode>();
89             Stack<Integer> sInt = new Stack<>();
90             String word = "";
91             // push the current node to top
92             s.push(root);
93             TrieNode tempTN = root;
94             while (!s.isEmpty())

```



```

94     {
95         // removes the object at the top of the stack
96         tempTN = s.pop();
97         if(!sInt.isEmpty())
98             word = word + (char)(sInt.pop() + 'a');
99         for (int i = tempTN.offSpring.length - 1; i >= 0; i--)
100         {
101             if (tempTN.offSpring[i] != null)
102             {
103                 s.push(tempTN.offSpring[i]);
104                 sInt.push(i);
105             }
106         }
107     }
108     return word;
109 }

110
111 // returns a new Trie rooted at the prefix
112 public Trie getSubTrie (String prefix)
113 {
114     //create a new TrieNode
115     Trie subTrie = new Trie();
116     TrieNode tempTN = root;
117     if (" ".equals(prefix) || prefix.isEmpty())
118         return null;
119     for (int i = 0; i < prefix.length(); i++)
120     {
121         // same concept as toCharArray
122         int subTrieInt = (int)prefix.charAt(i) - 97;
123         if(tempTN.offSpring[subTrieInt] == null)
124         {
125             return null;
126         }
127         subTrie.root = tempTN.getOffSpring(prefix.charAt(i));
128         tempTN = tempTN.offSpring[subTrieInt];
129     }
130     return subTrie;
131     // searches letters in trie if yes creates a new trie
132     // assign root to characters of the prefix
133 }

134
135 // returns a list containing all words in the Trie
136 public List getAllWords()
137 {
138     List allWords = new ArrayList();
139     for (TrieNode getWords : root.offSpring )
140     {
141         if (getWords != null)
142             getAllWords(allWords, getWords.charValueLetter + "",

```

```

144         ↪ getWords);
145     }
146     return allWords;
147 }
148 public void getAllWords(List allWords, String words, TrieNode
149     ↪ getWords)
150 {
151     if(getWords.isEnd)
152         allWords.add(words);
153     for (TrieNode tempNode : getWords.offSpring)
154     {
155         if (tempNode != null)
156             getAllWords(allWords, words + tempNode.charValueLetter,
157                 ↪ tempNode);
158     }
159 }
160 public static void main(String[] args) {
161     Trie newTrie = new Trie();
162     newTrie.add("cheers");
163     newTrie.add("cheese");
164     newTrie.add("chat");
165     newTrie.add("cat");
166     newTrie.add("bat");
167
168     System.out.println(newTrie.outputBreadthFirstSearch());
169     System.out.println(newTrie.outputDepthFirstSearch());
170     System.out.println(newTrie.getSubTrie("ch").getAllWords());
171     System.out.println(newTrie.getAllWords());
172 }
173 }

```

4.4 Part 3: AutoCompletionTrieNode

Listing 4: AutoCompletionTrieNode.java

```
1 package CW2;
2
3 public class AutoCompletionTrieNode
4 {
5     //the character that is stored inside the node
6     public char charValueLetter;
7     // variable to check if this is the last character in the string
8     public boolean isEnd;
9     // variable to check whether the node has been node
10    public boolean visitedNode;
11    //creating an array of fixed size 26
12    public AutoCompletionTrieNode[] offSpring;
13    int frequency = 0;
14
15    public AutoCompletionTrieNode()
16    {
17        // initialize the array of fixed size 26
18        this.offSpring = new AutoCompletionTrieNode[26];
19        //sets both the booleans to false
20        this.isEnd = false;
21        this.visitedNode = false;
22        // iterates through the length of the array and sets all the
23        // ↪ elements to null
24        for(int i = 0; i < offSpring.length; i++)
25            offSpring[i] = null;
26    }
27
28    public AutoCompletionTrieNode(char c)
29    {
30        charValueLetter = c;
31    }
32
33    public static AutoCompletionTrieNode makeNode(char cNode)
34    {
35        //create a new TrieNode
36        AutoCompletionTrieNode newTrieN = new AutoCompletionTrieNode();
37        newTrieN.isEnd = false;
38        newTrieN.charValueLetter = cNode;
39
40        return newTrieN;
41    }
42
43    // getting the offSpring with the specific character
44    public AutoCompletionTrieNode getOffSpring (char cOffSpring)
45    {
46        for(int i = 0; i < offSpring.length; i++)
```

```

46     {
47         //make sure its not invalid or null and is equal to the the
         ↳ specific character, then adding and updating each item
         ↳ in the array
48         if (offSpring[i] != null && offSpring[i].charValueLetter ==
         ↳ cOffSpring)
49             return offSpring[i];
50     }
51     return null;
52 }
53
54 // gets character from next node and converts it to an integer and
         ↳ then removes 97(cos ascii for a)
55 // basically turning a to z to 0 to 25 and then storing next at that
         ↳ index
56 public void toCharArray(AutoCompletionTrieNode nextNode)
57 {
58     int node = (int)nextNode.charValueLetter - 97;
59     offSpring[node] = nextNode;
60 }
61
62 public char getCharValueLetter()
63 {
64     return charValueLetter;
65 }
66 }

```

4.5 Part 3: AutoCompletionTrie

Listing 5: AutocompletionTrie.java

```
1 package CW2;
2
3 import java.util.*;
4 import java.lang.*;
5
6 public class AutoCompletionTrie
7 {
8     AutoCompletionTrieNode root = new AutoCompletionTrieNode();
9     int wordFrequency;
10
11
12     //adds a key to trie and returns true if the addition was successful
13     ↪ i.e. returns false if key already exist in the Trie
14     public boolean add(String key /*,int addFrequency*/)
15     {
16         AutoCompletionTrieNode rootTemp = root;
17         for (int i = 0; i < key.length(); i++)
18         {
19             // charAt returns the character at the specified index
20             AutoCompletionTrieNode nextNode =
21                 ↪ rootTemp.getOffSpring(key.charAt(i));
22             if (nextNode == null)
23             {
24                 nextNode =
25                     ↪ AutoCompletionTrieNode.makeNode(key.charAt(i));
26                 rootTemp.toCharArray(nextNode);
27             }
28             rootTemp = nextNode;
29         }
30         rootTemp.isEnd = true;
31         rootTemp.frequency++;
32         //rootTemp.frequency = addFrequency;
33         return rootTemp.isEnd;
34     }
35
36     public boolean contains(String key)
37     {
38         AutoCompletionTrieNode rootTemp = root;
39         if (rootTemp != null && rootTemp.isEnd )
40             return true;
41         for(int i = 0; i < key.length(); i++)
42         {
43             AutoCompletionTrieNode nodeNext =
44                 ↪ rootTemp.getOffSpring((key.charAt(i)));
45             if (nodeNext == null)
46                 return false;
```

```

43         else
44         {
45             rootTemp = nodeNext;
46             return true;
47         }
48     }
49     return true;
50 }
51
52 // returns a string representing a breadth first traversal
53 public String outputBreadthFirstSearch()
54 {
55     Queue<AutoCompletionTrieNode> queue = new LinkedList<>();
56     ArrayList<Character> characterArrayList = new ArrayList<>();
57     queue.add(root);
58     //while the linkedlist is full, we take the item in front of the
59     //    ↪ queue and add it to the list
60     // add items that aren't in the list to the back of the queue.
61     while (!queue.isEmpty())
62     {
63         AutoCompletionTrieNode currentNode = queue.remove();
64         //check that offSpring has another element
65         if (currentNode.offSpring != null)
66         {
67             for(int i = 0; i < currentNode.offSpring.length; i++)
68             {
69                 if (currentNode.offSpring[i] != null)
70                     queue.add(currentNode.offSpring[i]);
71             }
72             characterArrayList.add(currentNode.charValueLetter);
73         }
74     }
75     // StringBuilders is like an array of strings.
76     // creating a new stringbuilder of the size of characterArrayList
77     StringBuilder buildString = new
78     //    ↪ StringBuilder(characterArrayList.size());
79     //adding each character in characterArrayList to buildString
80     for (Character characterBFS : characterArrayList)
81         buildString.append(characterBFS);
82     return buildString.toString();
83 }
84
85 //returns a string representing a pre-order depth first traversal
86 // make it recursive
87 public String outputDepthFirstSearch()
88 {
89     if (root == null)
90         return null;
91     // create a stack for DFS i.e. which nodes to visit
92     Stack<AutoCompletionTrieNode> s = new

```

```

    ↪ Stack<AutoCompletionTrieNode>();
91 Stack<Integer> sInt = new Stack<>();
92 String word = "";
93 // push the current node to top
94 s.push(root);
95 AutoCompletionTrieNode tempTN = root;
96 while (!s.isEmpty())
97 {
98     // removes the object at the top of the stack
99     tempTN = s.pop();
100     if(!sInt.isEmpty())
101         word = word + (char)(sInt.pop() + 'a');
102     for (int i = tempTN.offSpring.length - 1; i >= 0; i--)
103     {
104         if (tempTN.offSpring[i] != null)
105         {
106             s.push(tempTN.offSpring[i]);
107             sInt.push(i);
108         }
109     }
110 }
111 return word;
112 }
113
114 // returns a new Trie rooted at the prefix
115 public AutoCompletionTrie getSubTrie (String prefix)
116 {
117     //create a new TrieNode
118     AutoCompletionTrie subTrie = new AutoCompletionTrie();
119     AutoCompletionTrieNode tempTN = root;
120     if (" ".equals(prefix) || prefix.isEmpty())
121         return null;
122     for (int i = 0; i < prefix.length(); i++)
123     {
124         // same concept as toCharArray
125         int subTrieInt = (int)prefix.charAt(i) - 97;
126         if(tempTN.offSpring[subTrieInt] == null)
127         {
128             return null;
129         }
130         subTrie.root = tempTN.getOffSpring(prefix.charAt(i));
131         tempTN = tempTN.offSpring[subTrieInt];
132     }
133     return subTrie;
134     // searches letters in trie if yes creates a new trie
135     // assign root to characters of the prefix
136 }
137
138 public ArrayList<String> getAllWords()
139 {

```

```

140     ArrayList<String> allWords = new ArrayList();
141     for (AutoCompletionTrieNode getWords : root.offSpring )
142     {
143         if (getWords != null)
144             getAllWords(allWords, getWords.charValueLetter + "",
145                 ↪ getWords);
146     }
147     return allWords;
148 }
149
150
151 public void getAllWords(List allWords, String words,
152     ↪ AutoCompletionTrieNode getWords)
153 {
154     if(getWords.isEnd)
155         allWords.add(words);
156     for (AutoCompletionTrieNode tempNode : getWords.offSpring)
157     {
158         if (tempNode != null)
159             getAllWords(allWords, words + tempNode.charValueLetter,
160                 ↪ tempNode);
161     }
162 }
163
164 // same concept as getSubTrie except in this case it returns the
165     ↪ int, frequency rather than the subTrie.
166 public int getWordFrequency(String word)
167 {
168     // creating anew autocompletiontrie called subTrie
169     AutoCompletionTrie subTrie = new AutoCompletionTrie();
170     // creating a temporary TrieNode
171     AutoCompletionTrieNode tempTN = root;
172     if (" ".equals(word) || word.isEmpty())
173         return 0;
174     for (int i = 0; i < word.length(); i++)
175     {
176         // same concept as toCharArray
177         int subTrieInt = (int)word.charAt(i) - 97;
178         if(tempTN.offSpring[subTrieInt] == null)
179         {
180             return 0;
181         }
182         subTrie.root = tempTN.getOffSpring(word.charAt(i));
183         tempTN = tempTN.offSpring[subTrieInt];
184     }
185     // returns the frequency
186     return tempTN.frequency;
187 }
188 }

```


4.6 Part 3: AutoCompletion

Listing 6: AutoCompletion.java

```
1 package CW2;
2
3 import java.io.*;
4 import java.util.*;
5
6 import static CW2.DictionaryFinder.readWordsFromCSV;
7
8 public class AutoCompletion
9 {
10     // reads the queries from lotrQueries.csv and uses the new line
    ↪ character as a delimiter.
11     public static ArrayList<String> readWordsNewLine(String file) throws
    ↪ FileNotFoundException
12     {
13         Scanner sc=new Scanner(new File(file));
14         sc.useDelimiter("\n");
15         ArrayList<String> words=new ArrayList<>();
16         String str;
17         while(sc.hasNext())
18         {
19             str=sc.next();
20             str=str.trim();
21             str=str.toLowerCase();
22             words.add(str);
23         }
24         return words;
25     }
26
27     public static void main(String[] args) throws Exception
28     {
29         // PrintWriter prints the matches and their probabilities to the
    ↪ specified
30         PrintWriter writeToFile= new PrintWriter
31         ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\lotrMatches.csv"
32         // reading and pushing all the queries from lotrQueries.csv to
    ↪ an arraylist called lotrQueries
33         ArrayList<String> lotrQueries = readWordsNewLine
34         ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\lotrQueries.csv"
35         //System.out.println(lotrQueries);
36         // creating a new AutoCompletionTrie called autoTrie
37         AutoCompletionTrie autoTrie = new AutoCompletionTrie();
38         // for each word in the lotr.csv, add it to autoTrie
39         for (String word: readWordsFromCSV
40         ("C:\\Users\\rohan\\IdeaProjects\\CW2\\src\\TextFiles\\lotr.csv"))
41             autoTrie.add(word);
42         //System.out.println(autoTrie.getAllWords());
```

```

43 //System.out.println(autoTrie.getWordFrequency("yellow"));
44 // for each query in the arraylist, lotrQueries
45 for (String query: lotrQueries)
46 {
47     // push the query into another AutoCompletionTrie called
48     ↪ newSubTrie
49     AutoCompletionTrie newSubTrie = autoTrie.getSubTrie(query);
50     // initializing totalFrequencies to 0;
51     int totalFrequencies=0;
52     // prints out all the words in newSubTrie
53     //System.out.println(newSubTrie.getAllWords());
54     // pushing all the words in newSubTrie to an arraylist
55     ↪ called, wordsToSort
56     ArrayList<String> wordsToSort = newSubTrie.getAllWords();
57     // for loop that goes through the size of the arraylist,
58     ↪ wordsToSort
59     for (int i =0; i<wordsToSort.size(); i++)
60         // returns the element at the given index after going
61         ↪ through wordsToSort
62         wordsToSort.set(i, query+wordsToSort.get(i));
63     // if the frequency of a query is more than 0, it adds it to
64     ↪ the arraylist, wordsToSort
65     if(autoTrie.getWordFrequency(query) > 0)
66         wordsToSort.add(query);
67     // sort the words using a lambda and Integer.compare to
68     ↪ compare the frequency of the words and then sort.
69     wordsToSort.sort((String word1, String
70     ↪ word2)->-Integer.compare(autoTrie.getWordFrequency(word1),
71     ↪ (autoTrie.getWordFrequency(word2))));
72     //for each newWord in wordsToSort add the frequency of the
73     ↪ newWord to the totalFrequencies
74     for (String newWord : wordsToSort)
75     {
76         totalFrequencies += autoTrie.getWordFrequency(newWord);
77     }
78     // writeToFile.print() is used to write to lotrMatches.csv
79     writeToFile.print(query+",");
80     // for loop that only prints the top three frequencies or
81     ↪ less if the prefix appears less times
82     for (int i = 0; i < Math.min(3,wordsToSort.size()); i++)
83     {
84         writeToFile.print(wordsToSort.get(i)+",");
85         // prints the probabilities of the number of time each
86         ↪ word is likely to occur
87         writeToFile.print(((float)autoTrie.getWordFrequency
88         (wordsToSort.get(i))/totalFrequencies+",");
89     }
90     // makes sure each query and its occurrences and its
91     ↪ probabilities
92     writeToFile.println();

```

```
81         }  
82         writeToFile.close();  
83     }  
84 }
```

5 Answer

5.1 Part 3: lotrMatches.csv

Listing 7: lotrMatches.csv

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