DSA Coursework 2

100172935 (`ngx16ybu`)

Wed, 25 Apr 2018 13:16

PDF prepared using PASS version 1.12 running on `Windows 10 10.0` (`amd64`).

☑ I agree that by submitting a PDF generated by PASS I am confirming that I have checked the PDF and that it correctly represents my submission.

Contents

DictionaryFinder.java

```java
package dsacoursework2;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Scanner;
import java.util.SortedMap;
import java.util.TreeMap;

/**
 *
 * @author ngx16ybu
 */
public class DictionaryFinder {

    private SortedMap<String, Integer> treeMap;

    public DictionaryFinder() {
        treeMap = new TreeMap<>();
    }

    public void setTreeMap(SortedMap<String, Integer> treeMap) {
        this.treeMap = treeMap;
    }

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

    /**
     * Saves the file to a fileWriter
     *
     * @param c, file
     */
    public static void saveCollectionToFile(Collection<?> c, String file)
            throws IOException {
        FileWriter fileWriter = new FileWriter(file);
        PrintWriter printWriter = new PrintWriter(fileWriter);
        for (Object w : c) {
            printWriter.println(w.toString());
```

```
        }
        printWriter.close();
    }

    /**
     * Forms a dictionary of words
     *
     * @param words
     */
    public void formDictionary(List<String> words) {
        for (String word : words) {
            if (!(this.treeMap.containsKey(word))) {
                this.treeMap.put(word, 1);
            } else {
                this.treeMap.put(word, treeMap.get(word) + 1);
            }

        }
    }

    /**
     * Saves the dictionary with a printwriter
     *
     * @param f
     */
    public void saveToFile() throws IOException {
        StringBuilder sb = new StringBuilder();
        try (PrintWriter printWriter = new PrintWriter("output.csv")) {
            for (String w : treeMap.keySet()) {
                sb.append(w).append(" ").append(treeMap.get(w)).append("\n");
            }
            printWriter.write(sb.toString());
        }
    }

    public static void main(String[] args) throws Exception {
        DictionaryFinder df = new DictionaryFinder();
        ArrayList<String> in = readWordsFromCSV("testDocument.txt");
        df.formDictionary(in);
        df.saveToFile();
    }
}
```

TrieNode.java

```java
/*
 *TrieNode class and a root for Trie
 */
package dsacoursework2;

/**
 *
 * @author ngx16ybu
 */
public class TrieNode {

    public char letter;
    public TrieNode[] offspring;
    public boolean complete;

    /**
     * Default constructor with special character used for root node
     */
    public TrieNode() {
        this.letter = '\0';
        this.offspring = new TrieNode[26];
        this.complete = false;
    }

    /**
     * Constructor for creating a TrieNode with a given character
     *
     * @param c
     */
    public TrieNode(char c) {
        this.letter = c;
        this.offspring = new TrieNode[26];
        this.complete = false;
    }

    /**
     * Method to check whether the node is a complete word
     *
     */
    public boolean isComplete() {
        return this.complete;
    }

    /**
     * method to set a node as a complete word
     *
     * @param set
     */
    public void setComplete(boolean set) {
        this.complete = set;
    }

    /**
     * Accessor Method to get the offspring of the node
     *
     */
    public TrieNode[] getOffspring() {
        return this.offspring;
    }

    /**
```

```java
     * Method to get the index of the specified character
     *
     * @param c
     */
    public static int getCharIndex(char c) {
        return c - 'a';
    }

    /**
     * Accessor method to get the want node of an offspring
     *
     * @param c
     */
    public TrieNode getNode(char c) {
        return this.offspring[getCharIndex(c)];
    }

    /**
     * Method to set the wanted node of an offspring
     *
     * @param c
     */
    public void setNode(char c) {
        this.offspring[getCharIndex(c)] = new TrieNode(c);
    }

    /**
     * Accessor method to return the letter of a node
     *
     * @param c
     */
    public char getChar() {
        return letter;
    }

}
```

Trie.java

```java
/*
 * Trie data structure
 */
package dsacoursework2;

import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

/**
 *
 * @author ngx16ybu
 */
public class Trie extends TrieNode {

    TrieNode root, node;

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

    public void setRootNode() {
        this.root = node;
    }

    /**
     * Method to add a key to the trie
     *
     * @param key
     */
    public boolean add(String key) {
        //root trienode
        TrieNode currentNode = root;
        //iterate through all the keys
        for (int i = 0; i < key.length(); i++) {
            //get their chars at i positions
            char current = key.charAt(i);
            if (currentNode != null) {

                TrieNode child = currentNode.getNode(current);
                if (child == null) {
                    currentNode.setNode(current);
                    child = currentNode.getNode(current);
                }
                currentNode = child;
            }
        }
        //if node is complete return false
        if (currentNode.isComplete()) {
            return false;
        }
        //set node as complete and return true
        currentNode.setComplete(true);
        return true;

    }

    /**
     * Method to check if the word passed in the trie is a whole word
     *
     * @param key
```

```java
     */
    public boolean contains(String key) {
        // set current node to root
        TrieNode currentNode = root;
        //iterate through the key
        for (char c : key.toCharArray()) {
            //if next node is not null return false
            if (currentNode.getNode(c) == null) {
                return false;
            }
            //else if it exists, assign currentNode to it
            currentNode = currentNode.getNode(c);
        }
        //return whether the current Node is complete
        return currentNode.isComplete();
    }

    /**
     * Method to output the string by breadth first search
     *
     * @param key
     */
    public String outputBreadthFirstSearch() {
        //a string for the result
        String result = "";
        //initialiase a queue for the nodes
        Queue nodes = new LinkedList();
        //add a root node
        nodes.add(root);
        //iterate through nodes until they are empty
        while (!nodes.isEmpty()) {
            //get the first node
            TrieNode temp = (TrieNode) nodes.poll();
            //add all temp node letters to result
            result += temp.getChar();
            //iterate through every node
            for (TrieNode node : temp.getOffspring()) {
                //if node is not null
                if (node != null) {
                    //add a node to nodes linkedlist
                    nodes.add(node);
                }
            }
        }
        //return result
        return result.toLowerCase();
    }

    /**
     * Method to output the string by depth first search
     *
     * @param trienode
     * @return result
     */
    public String depthFirstSearch(TrieNode trienode) {
        //initialise an empty string
        String result = "";
        //iterate through every node
        for (TrieNode node : trienode.getOffspring()) {
            //if node is not null
            if (node != null) {
                //add all the nodes to result from depthfirstseach
                result += depthFirstSearch(node);
```

```
            }
        }
        //append all letters of trienode to result
        result += trienode.getChar();
        return result;
    }

    public String outputDepthFirstSearch() {
        //initialise an empty strin
        String result = "";
        //if root is not empty
        if (root != null) {
            //append result root from depthfirstsearch
            result += depthFirstSearch(root);
        }
        return result;
    }

    /**
     * Method to return a trie rooted at the prefix
     *
     * @param prefix
     */
    public Trie getSubTrie(String prefix) {
        //set current node to root
        TrieNode currentNode = root;
        Trie result = new Trie();
        //iterate through all the prefixes
        for (int i = 0; i < prefix.length(); i++) {

            int index = (int) prefix.charAt(i) - 'a';

            if (currentNode.getNode(prefix.charAt(i)) != null) {
                result.root = currentNode.getNode(prefix.charAt(i));
            }
            currentNode = currentNode.offspring[index];
        }
        return result;
    }

    public List<String> getAllWords() {
        List<String> output = new LinkedList<>();
        getAllWords("\0", root, output);
        return output;
    }
    /**
     * Method to get All the words out of the prefix
     *
     * @param prefix, trienode, nodes
     */
    private void getAllWords(String prefix, TrieNode trienode,
            List<String> nodes) {
        //iterate through every trienode
        for (TrieNode temp : trienode.getOffspring()) {
            //if temp is not null
            if (temp != null) {
                //word from prefix  and temp char
                String prefix2 = prefix + temp.getChar();

                getAllWords(prefix2, temp, nodes);
            }
        }

```

```
        if (trienode.isComplete()) {
            nodes.add(prefix);
        }
    }

    public static void main(String[] args) {
        Trie test = new Trie();
        System.out.println(test.add("cheers"));
        System.out.println(test.add("cheese"));
        System.out.println(test.add("chat"));
        System.out.println(test.add("cat"));
        System.out.println(test.add("bat"));
        System.out.println("---------- ");
        System.out.println(test.contains("chee"));
        System.out.println(test.contains("afc"));
        System.out.println(test.contains("ba"));
        System.out.println(test.contains("cheese"));
        System.out.println(test.contains("bat"));
        System.out.println("---------- ");
        System.out.println(test.outputBreadthFirstSearch());
        System.out.println("---------- ");
        System.out.println(test.outputDepthFirstSearch());
        System.out.println("---------- ");
        Trie subtrie = test.getSubTrie("ch");
        System.out.println(subtrie.outputBreadthFirstSearch());
        System.out.println("---------- ");
        System.out.println(test.getAllWords());

    }
}
```

AutoCompletion.java

```java
/*
 * Word completion program
 */
package dsacoursework2;

import static dsacoursework2.DictionaryFinder.readWordsFromCSV;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.NavigableMap;
import java.util.TreeMap;

/**
 *
 * @author ngx16ybu
 */
public class AutoCompletion {

    public static void main(String[] args) throws FileNotFoundException,
            IOException {
        DictionaryFinder df = new DictionaryFinder();
        DictionaryFinder df1 = new DictionaryFinder();
        //printwriter to save the output to "lotrMatches.csv"
        PrintWriter pw = new PrintWriter("lotrMatches.csv");
        //ArrayList in that reads all the words from "lotr.csv"
        ArrayList<String> in = readWordsFromCSV("lotr.csv");
        //form a dictionary of ArrayList in
        df.formDictionary(in);
        ArrayList<String> LotrQueries = new ArrayList();
        ArrayList<Integer> test = new ArrayList();
        //add all the prefixes LotrQueries ArrayList
        LotrQueries.addAll(readWordsFromCSV("lotrQueries.csv"));
        //form a dictionary of prefixes
        df1.formDictionary(LotrQueries);
        Trie wordstrie = new Trie();
        HashMap<String, Integer> words = new HashMap<String, Integer>();
        NavigableMap<String, Integer> storeAuto
                = new TreeMap<String, Integer>();
        NavigableMap<String, Integer> storeAutoTemp
                = new TreeMap<String, Integer>();
        //used to count frequency
        int count = 0;
        //iterate through all the words in ArrayList in from "lotr.csv"
        for (int i = 0; i < in.size(); i++) {
            //add those words to the trie
            wordstrie.add(in.get(i));

        }
        //
        List<String> lotr = new ArrayList<>();
        Trie temp;
        String prefix;
        String auto;
        //iterate through arraylist in to put the words and their frequencies
        //to hashmap words
        for (String str : in) {
            if (words.containsKey(str)) {
```

```
                words.put(str, words.get(str) + 1);
            } else {
                words.put(str, 1);
            }
        }
        //iterate through all the prefixes
        for (int i = 0; i < LotrQueries.size(); i++) {
            //get all the prefixes and add them to the lotr List
            lotr.add(LotrQueries.get(i));
            //Using the getSubtrie method to get the sub trie of the words
            temp = wordstrie.getSubTrie(lotr.get(i));
            //get all the words that start with associated prefixes
            List<String> list = temp.getAllWords();
            //get all the prefixes into prefix String
            prefix = lotr.get(i);
            //iterate through the list of words
            for (int j = 0; j < list.size(); j++) {
                //adds the prefix to the other part of the word
                auto = prefix.trim() + list.get(j).trim();
                //iterate through all the words from entry set
                for (Map.Entry<String, Integer> entry : words.entrySet()) {
                    //if words that are in auto equal to the words in entry map
                    if (auto.equals(entry.getKey())) {
                        //store those words in a storeAuto map
                        storeAuto.put(entry.getKey(), entry.getValue());

                    }

                }

            }

        }

        int counter = 0;
        int numberOfPrefixes = 2;
        //iterate through all the words from storeAutp
        for (Map.Entry<String, Integer> entry : storeAuto.entrySet()) {
            //if words are equal to the first entry of StoreAuto
            if (entry.equals(storeAuto.firstEntry())) {
                //store them into storeAuto temp
                storeAutoTemp.put(entry.getKey(), entry.getValue());
            } //if prefixes are equal and count is less than 3 words
            else if (storeAutoTemp.lastEntry().getKey().substring(0, 2).equals
            (entry.getKey().substring(0, 2)) && counter < numberOfPrefixes) {
                //store words into storeAuto temp
                storeAutoTemp.put(entry.getKey(), entry.getValue());
                //increase counter by 1
                counter++;
            } //if prefixes are not equal to each other
            else if (!(storeAutoTemp.lastEntry().getKey().substring(0, 2).equals
            (entry.getKey().substring(0, 2)))) {
                //set counter to 0
                counter = 0;
                //store words into storeAuto temp
                storeAutoTemp.put(entry.getKey(), entry.getValue());
            }

        }
        //create an empty string
        String key = "  ";
        //iterate through all the words that are in the storeAutoTemp map
        for (Map.Entry<String, Integer> entry : storeAutoTemp.entrySet()) {
```

```
            //if the prefix of key is not equal to entry map prefix,which store
            //storeAuto words
            if (!(key.substring(0, 2).equals(entry.getKey().substring(0, 2)))) {
                //if frequency is not equal 0
                if (count != 0) {
                    //add the frequency to test arraylist
                    test.add(count);

                }
                count = 0;
            }
            //get all the keys from entry map that store storeAuto words
            key = entry.getKey();
            //get all the values of keys from entry map
            Integer value = entry.getValue();
            //counts the frequency of each prefix
            count += value;
            //if entry map equals last entry of th, add it
            if (entry.equals(storeAutoTemp.lastEntry())) {
                test.add(count);
            }

        }
        //attempt to sort
        Map.Entry<String, Integer> previousEntry = null;
        Map.Entry<String, Integer> tempEntry;
        for (Map.Entry<String, Integer> entry : storeAutoTemp.entrySet()) {

            if (!entry.equals(storeAutoTemp.firstEntry())) {
                if (entry.getKey().substring(0, 2).equals(previousEntry.getKey()
                        .substring(0, 2)) && entry.getValue() >
                        previousEntry.getValue()) {
                    tempEntry = previousEntry;
                    previousEntry = entry;
                    entry = tempEntry;

                }
            } else {
                previousEntry = entry;
            }

        }
        //initialise an empty string
        key = "  ";
        //used to count for probability
        double probability;
        //i is used to switch between prefixes
        int i = -1;
        //iterate through all the words that are in the storeAutoTemp map
        for (Map.Entry<String, Integer> entry : storeAutoTemp.entrySet()) {
            //if the prefix of key is not equal to entry map prefix,which store
            //storeAuto words
            if (!(key.substring(0, 2).equals(entry.getKey().substring(0, 2)))) {
                //used to separate words with different prefixes
                i++;
                //System.out.println(keyCount);
            }
            //frequency of each prefixes
            double total = test.get(i);
            //get the key value
            key = entry.getKey();
            //get value of key
            double value = entry.getValue();
```

```
                //count the probability
                probability = value / total;
                //print out key and their probability
                System.out.println(key + " : " + probability);
                //print to "lotrMatches.csv" using printwriter
                pw.print(key + " : " + probability);
                pw.print("\n");

            }
            pw.close();

        }

}
```

AutoCompletionTrie.java

File not found.

AutoCompletionTrieNode.java

File not found.

100172935.pdf

(Included PDF starts on next page.)

CMP-5014Y Data Structures and Algorithms

100172935

April 25, 2018

1 Form a dictionary

Algorithm 1 formDictionary algorithm

Input: List of String words
Output: SortedMap treeMap

```
TrieNode currentNode ← root
for String word in words do
    if treeMap does not contain word then
        Add word,key to treeMap
    else
        Add word, key by n+1 to treeMap
    end if
end for
return treeMap
```

1.1 Fundamental Operation

The fundamental operation for the algorithm is Add word,key to treeMap and Add word, key by n+1 to treeMap.

1.2 Run time complexity function

$$\sum_{i=1}^{n} log(i-1)1$$

1.3 Worst case scenario

Worst case scenario is that the words that have been added to the treeMap are new words.

2 Trie data structure

2.1 Add method for adding a key to the trie

Algorithm 2 add algorithm

Input: String key
Output: true if key was successfully added to the trie, false otherwise

```
TrieNode currentNode ← root
for every letter current in key do
    TrieNode child ←currentNode.getNode(current)
    if child is not equal null then
        currentNode.setNode(current)
        child ←currentNode.getNode(current)
    end if
    currentNode ← child
end for
if currentNode.isComplete() = true then
    return false
end if
currentNode.setComplete()←true
return true
```

2.2 Contains method to check whether the word that is passed is a full word and not prefix

Algorithm 3 contains algorithm

Input: String key
Output: true if the whole word was in the trie, false otherwise

```
TrieNode currentNode ← root
for every letter c in key do
    if currentNode.getNode(c) is equal null then
        return false
    else
        currentNode ← currentNode.getNode(c)
    end if
end for
return currentNode.isComplete()
```

2.3 Output by Breadth First Search Method

Algorithm 4 outputBreadthFirstSearch algorithm

Input: No Input
Output: String result

```
String result ← empty String
Queue nodes ← empty LinkedList
nodes.add(root)
while nodes.isEmpty() is equal false do
    TrieNode temp ← nodes.poll()
    append result with temp.getChar()
    for each offspring nodein temp.getOffSpring() do
        if node is not equal null then
            nodes.add(node)
        end if
    end for
end while
return result
```

2.4 Depth First Search Method

Algorithm 5 DepthFirsSearch algorithm

Input: Trienode trienode
Output: result

```
String result ← empty String
Queue nodes ← empty LinkedList
for each offspring nodein trienode.getOffSpring() do
    if node is not equal null then
        append result with depthFirstSearch(node)
    end if
end for
append result with trienode.getChar()
```

2.5 Output by Depth First Search Method

Algorithm 6 OutputDepthFirsSearch algorithm

Output: result

```
String result ← empty String
if root is not equal null then
    append result with depthFirstSearch(root)
end if
return result
```

2.6 get SubTrie Method to return a trie rooted at the prefix

Algorithm 7 getSubTrie algorithm

Input: String prefix
Output: Trie result

```
TrieNode currentNode ← root
Trie result ←new Trie()
for every prefix i in prefix.lenght() do
    int index←prefix.charAt(i) - 'a'
    if currentNode.getNode(prefix.charAt(i) not equal null then
        result.root ← currentNode.getNode(prefix.charAt(i)
    end if
    currentNode ← currentNode.offspring[index]
end for
return result
```

2.7 get AllWords function to get the all the words in the trie

Algorithm 8 getAllWords function algorithm

Input: String prefix, TrieNode trienode, List of String Nodes

```
for each offspring temp in trienode.getOffspring() do
    if temp is not equal null then
        String prefix2 ← prefix + temp.getChar()
        getAllWords(prefix2, temp, nodes)
    end if
end for
if trienode.isComplete() then
    nodes.add(prefix)
end if
```

2.8 get AllWords function to return the all the words in the trie

Algorithm 9 getAllWords algorithm

Output: List of Strings output

```
List of Strings output ←new LinkedList
getAllWords("",root,output)
return output
```

3 Word Auto Completion

3.1 Auto Competion program

Algorithm 10 AutoCompletion algorithm

```
ArrayList of Strings LotrQueries ←a list of prefixes
List of Strings lotr ←new ArrayList
Trie wordstrie ←a trie of all words
for each prefix i in LotrQueries.size() do
  lotr.add(LotrQueries.get(i))
  temp ←wordstrie.getSubTrie(lotr.get(i))
  List of Strings list ← temp.getAllWords()
  prefix ← lotr.get(i)
  for each word j in list.size() do
    auto ← prefix + list.get(j)
    for every entry of Map of String and Integer in words.entrySet() do
      if auto.equals(entry.getKey()) then
        storeAuto.put(entry.getKey(),entry.getValue())
      end if
    end for
  end for
end for
```

3.2 AutoCompletion output

Word	Probability
able	0.14285714285714285
abominable	0.047619047619047616
about	0.8095238095238095
frodo	0.4909090909090909
from	0.43636363636363634
front	0.07272727272727272
go	0.7647058823529411
goblins	0.058823529411764705
goes	0.17647058823529413
grasp	0.07692307692307693
grass	0.7692307692307693
grasses	0.15384615384615385
merely	0.02631578947368421
merrily	0.02631578947368421
merry	0.9473684210526315
sam	1.0
the	0.8471454880294659
their	0.06077348066298342
them	0.09208103130755065