Exercise 11: Scrabble Cheater Basic Edition

Due on Monday, July 04, 2016

M. Schütz, P. Ulc, R. Krajewski

Contents

Exercise 1

The first thing we implemented was a simple file reader that just reads a file line by line. We did that already in the histogram exercise, so there is nothing new to explain in it.

Listing 1: File Reader

```
while (currentInput != null) {
                try {
                     currentInput = fileReader.readLine();
                } catch (IOException e) {
                     e.printStackTrace();
                }
                if (currentInput != null) {
                     String normalized = normalize(currentInput);
                     int hash = generateHash(normalized);
                     putInHashTable(currentInput, hash);
                }
                //System.out.println(currentInput);
                count++;
          }
```

For every line the reader calls three methods: normalize, generateHash and putInHashTable. Normalize sorts the word alphabetically, so that for example "coffee" is converted into "ceeffo". This allows us to skip all the possible permutations a word the user inputs might have. We just generate the hash for the normalized word, normalize the users input and compare those two, so that all possible permutations end up getting recognized.

Listing 2: method normalize

```
private String normalize(String original) {
          char[] originalArray = original.toCharArray();
          Arrays.sort(originalArray);
          String sorted = new String(originalArray);
          return sorted;
     }
```

To generate the hash, we used the ascii values of the chars, but we subtract 96 from the value, to get values from 1 to 26 instead of 96 to 122. To avoid the issue that words like "c" and "aaa" have the same hash, we used a polynomial hashing function, which uses the indices of the chars as exponents. Lastly we use the modulo 1000 to get a value that fits into a reasonable sized array. We managed to get our longest collision chain down to 16, using a file with 3975 words as input.

Listing 3: method generateHash

```
private int generateHash(String word) {
          char[] chars = word.toCharArray();
          long polynom = 0;
          for (int i = 0; i < chars.length; i++) {
               int asciiPosition = chars[i] - 96; // a maps to 1
               polynom += Math.pow(asciiPosition, i );
          }
          return (int) (polynom % hashTableSize);
     }
```

Exercise 1 continued on next page...

```
Words total: 3975
Max colision: 16
Max colision position 87
```

We tested that by adding a field to the class and checking the size of the LinkedList that represents the collision chain every time we put in an element and updating the current longest collision chain size. Our hashTable is an array of LinkedLists of Strings. After generating a hash, we add the original word to the linked list using the hash as index for the array containing the linked lists.

Listing 4: method putInHashTable.

```
private void putInHashTable(String word, int hash) {
        if (hash > 0 && hash < hashTable.length) {
            hashTable[hash].add(word);
            int colisioncounter = hashTable[hash].size();
            if (colisioncounter > maxColisioncounter) {
                maxColisioncounter = colisioncounter;
                maxColisionposition = hash;
            }
        }
    }
```

Exercises 3 and 4

Firstly we wrote a method that takes a String and returns all the words of the dictionary that are permutations of the String. We did not write a method, that returns all the entries, as that is pretty pointless, because the hashes are generated from the normalized words, meaning that words at one location of the array really don't have much in common other than the hash. Our method normalizes the input before it generates the corresponding hash. This is necessary, as our whole dictionary contains words that were normalized before generating a hash from them. As mentioned above, this eliminates the problem of permutations. After the hash was generated, the method takes the linked list at the position of the hash in the array and iterates through it, comparing the normalized input and the normalized version of the words in the linked list. Whenever there is a match, the method adds the original word to an ArrayList called output. In the end the method just returns the ArrayList. To get the users input, we just implemented a small run method, that loops the readLine() method of an BufferedInputReader from System.in until the user types quit.

Listing 5: method getWords

```
public ArrayList<String> getWords(String input) {
        ArrayList<String> output = new ArrayList<>();
        String normalizedInput = normalize(input);
        int hash = generateHash(normalizedInput);
        String current = "";
        LinkedList<String> listOfPotentialWords = hashTable[hash];
        ListIterator<String> itr = listOfPotentialWords.listIterator();
        while (itr.hasNext()) {
            current = itr.next();
            if (normalize(current).equals(normalizedInput)) {
                output.add(current);
            }
        }
        return output;
    }
```

Source Code

Listing 6: class ScrabbleCheaterBE

```
package src;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.ListIterator;

public class ScrabbleCheater {
     private final int hashTableSize = 1000;
     private LinkedList<String>[] hashTable;
     private int maxColisioncounter = 0;
     private int maxColisionposition = 0;

     @SuppressWarnings("unchecked")
     public ScrabbleCheater() {
          hashTable = new LinkedList[hashTableSize];
          for (int i = 0; i < hashTable.length; i++) {
               hashTable[i] = new LinkedList<String>();
          }
     }

     public static void main(String[] args) {
          ScrabbleCheater scrabbleCheater = new ScrabbleCheater();
          scrabbleCheater.readFile();
          scrabbleCheater.run();
     }

     public void run() {
          String input = "";
          BufferedReader userInput = new BufferedReader(new InputStreamReader(
              System.in));
          while (!input.equals("quit")) {
               try {
                    input = userInput.readLine();
               } catch (IOException e) {
               }
               ArrayList<String> result = getWords(input);
               for(String s: result){
                    System.out.println(s);
               }
          }
     }

     public ArrayList<String> getWords(String input) {
```

Source Code continued on next page...

```
50          ArrayList<String> output = new ArrayList<>();
            String normalizedInput = normalize(input);
            int hash = generateHash(normalizedInput);
            String current = "";
            LinkedList<String> listOfPotentialWords = hashTable[hash];
55          ListIterator<String> itr = listOfPotentialWords.listIterator();
            while (itr.hasNext()) {
                 current = itr.next();
                 if (normalize(current).equals(normalizedInput)) {
                      output.add(current);
60               }
            }
            return output;
       }

65     public void readFile() {
            File dictionaryFile = new File("/Users/imi/Dropbox/eclipse/info1/
                ScrabbleCheaterBE/src/words.txt");
            String currentInput = "";
            BufferedReader fileReader = null;
            int count = 0;
70          try {
                 fileReader = new BufferedReader(new InputStreamReader(new
                     FileInputStream(dictionaryFile)));
            } catch (FileNotFoundException e) {
                 e.printStackTrace();
            }
75          try {
                 currentInput = fileReader.readLine();
            } catch (IOException e1) {
                 e1.printStackTrace();
            }
80
            while (currentInput != null) {
                 try {
                      currentInput = fileReader.readLine();
                 } catch (IOException e) {
85                    e.printStackTrace();
                 }
                 if (currentInput != null) {
                      String normalized = normalize(currentInput);
                      int hash = generateHash(normalized);
90                    putInHashTable(currentInput, hash);
                 }
                 //System.out.println(currentInput);
                 count++;
            }
95
            try {
                 fileReader.close();
            } catch (IOException e) {
                 e.printStackTrace();
100         }
```

```
            System.out.println("Words total: " + count);
            System.out.println("Max colision: " + maxColisioncounter);
            System.out.println("Max colision position " + maxColisionposition);
        }

        private String normalize(String original) {
            char[] originalArray = original.toCharArray();
            Arrays.sort(originalArray);
            String sorted = new String(originalArray);
            return sorted;
        }

        private void putInHashTable(String word, int hash) {
            if (hash > 0 && hash < hashTable.length) {
                hashTable[hash].add(word);
                int colisioncounter = hashTable[hash].size();
                if (colisioncounter > maxColisioncounter) {
                    maxColisioncounter = colisioncounter;
                    maxColisionposition = hash;
                }
            }
        }

        private int generateHash(String word) {
            char[] chars = word.toCharArray();
            long polynom = 0;
            for (int i = 0; i < chars.length; i++) {
                int asciiPosition = chars[i] - 96; // a maps to 1
                polynom += Math.pow(asciiPosition, i );
            }
            return (int) (polynom % hashTableSize);
        }
    }
```