T9

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to introduce the step by step implementation of T9, which in the end given a valid keyboard combination, will return in a list all valid corresponding words. As a data source in order to populate the trie data structure, a file "kelly.txt" has been used which consists of approximately 8300 Swedish words.

Step 1: From letter to index

In order to get the T9 implementation started, I used the ASCII code of the letters in order to return the corresponding index. There are three special cases for the Swedish specific letters and I also checked that the character is a valid one, by looking at the ASCII code range since all small needed letters are between the 97 and 122 ASCII code. If valid, I subtract from the character the ASCII value of 'a'.

```
public int getCharCode(char c) {
//three special cases for the Swedish letters
    if (c == 'å')
        return 24;
    else if (c == 'ä')
        return 25;
    else if (c == 'ö')
        return 26;
    else if (c >= 97 && c <= 122) //checking that the character is a small letter
        return c - 'a';
    else
        return -1;
    }
```

Step 2: From index to letter

The next implemented method, was the reverse of the one previously presented. Namely, given an index, it should return the corresponding character. Once again, there are three special cases for the Swedish characters. This time instead of subtracting the ASCII code, I add it up.

```
public char getChar(int c) {
    if (c == 24)
        return 'å';
    else if (c == 25)
        return 'ä';
    else if (c == 26)
        return 'ö';
    else if (c >= 0 && c <= 24)
        return (char) (c + 'a'); //convert to char after adding the ASCII code
    else
        return ' ';
    }
```

Step 3: Converting the keyboard values to indexes

This method, given a character from 1 to 9, converts them to a corresponding index of type integer from 0 to 8. This method is useful since in the array we need to have the representation starting from index 0. Once again, first the '0' ASCII code is subtracted and then 1. This operation is possible since when doing it we use the integer values corresponding to the characters.

```
public int getKey(char c) {
    return (c - '0') - 1;
}
```

Step 4: Populating the trie data structure

In order to populate the trie with words provided in "kelly.txt", I have made use of the main class's constructor and of recursion. By and large, in the while loop, as long as the end of the file is not reached, an *addWords method* is called. What was a bit difficult for me to understand at first was that populating the trie didn't involve copying the words letter by letter, but on the right track, marking the end valid bit with true.

In the *addWords method*, the following steps have been followed:

- one line (word) in the file has been sent as parameter;

- if the word is null or empty, an excepton is thrown;
- the trie root is copied to a Node variable;
- for each word, I extract from it character by character and I use the previously shown method to map it to an index corresponding to the letter it represents;
- each node has a reference to a 27 long array where letters are marked by indexes - if the index of the character I am looking at is equal to null, it means that I have reached a leaf and the node corresponding to that letter has not yet been initialised; if so I initialise it;
- regardless of the node being or not initialised, I update the node I use to traverse the trie in order to move forward and create the path corresponding to the word length;
- when the path is at its end - we have reached the end of the word in the for loop and we set the valid bit to true;

```
public void addWords(String word) {
    if (word == null || word.isEmpty()) {
        throw new IllegalArgumentException("Invalid word!");
    }

    Node current = root;

    for (int i = 0; i < word.length(); i++) {
        char c = word.charAt(i);
        int index = getCharCode(c);

        if (current.next[index] == null) {
            Node node = new Node();
            current.next[index] = node;
        }
            current = current.next[index];
    }
    current.valid = true;
}
```

Step 5: Creating an extra method to check that the trie has been correctly populated

In order to check that the trie has been populated correctly, an extra method has been created to which a word is sent as parameter. Following the reverse

approach described above, I extracted from the word the indexes of the letters. Then, by iterating through the trie, if null was found before the end of the word, this was enough to conclude that the word was not there. In contrast, if I managed going though the whole string, in the end I returned the value of the valid bit so that I can conclude it the word existed or not in the data structure.

```
public boolean lookup(String word) {
    if (word == null || word.isEmpty()) {
        throw new IllegalArgumentException("Invalid word!");
    }

    Node current = root;

    for (int i = 0; i < word.length(); i++) {
        char c = word.charAt(i);
        int index = getCharCode(c);
        if (current.next[index] == null) {
            return false;
        }
        current = current.next[index];
    }
    return current.valid;
}
```

Step 6: Decode and Collect

In order to collect all valid and possible words given a certain T9 keyboard combination, we first have to convert the combination from a string to an int. Since in the future recursive steps, there are multiple possibilities for each key on the keyboard (3 letters for each), I decided to map it to an integer array for easy access. In addition, the list where the valid words will be stored is initialised, followed by the method call for the collection of words. In the end, we return the list, though which we go in the main method.

```
public ArrayList<String> decode(String key) {
    ArrayList<String> words = new ArrayList<String>();
    if (key == null || key.isEmpty())
        throw new IllegalArgumentException("Invalid key!");
    int[] code = new int[key.length()];

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

```
            code[i] = getKey(c);
        }

        collect(root, code, 0, "", words);
        return words;
}
```

Next, we will dive into the *collect* method, where the following steps have been taken:

- since we are going to use recursion, we start off by implementing the base case, which checks when the index reaches the end of the code array length; if so - we check the valid bit and if the word is found we add it to the list; we also have to stop the execution using "return";
- since each key is mapped to three letters, we have several scenarios and in order to access them we use the same concept as in storing a binary tree in an array; we multiply the code by three and add 0, 1 or 2 in order to find the corresponding letter in the 27 long array;
- in each recursive step we check all three different directions and combinations if they are 'null'; otherwise we keep calling the recursive function to continue checking in those directions as well; this way we get all possible key combinations, which we check in the base case whether or not they are valid;

```
public void collect(Node node, int[] code, int index, String s,
                    ArrayList<String> words) {
    if (index == code.length) {
        if (node.valid)
            words.add(s);
        return;
    }
    int res1 = code[index] * 3 + 0;
    int res2 = code[index] * 3 + 1;
    int res3 = code[index] * 3 + 2;
    if (node.next[res1] != null)
        collect(node.next[res1], code, index + 1, s + getChar(res1), words);

    if (node.next[res2] != null)
        collect(node.next[res2], code, index + 1, s + getChar(res2), words);

    if (node.next[res3] != null)
        collect(node.next[res3], code, index + 1, s + getChar(res3), words);
}
```