Assignment 10 Report - T9 in Java

Dean Tsankov

October 20, 2024

Introduction

In this report I will present what an implementation of the T9 concept might look like. T9 is related to the way text suggestions worked on old cell phones using a 9 digit keypad. The idea was that instead of pressing each key the respective number of times in order to choose one of the three letters written bellow it, you could press the key once and leave the guess work up to the phone itself. Using this, if you wanted to write "Yo" you would simply key in "96" instead of "999666". Now in order for this to work the phone had to keep track of valid words it could give out as suggestions, but since storage space was limited and especially so for novelty features, a clever way to optimize this was thought of.

T9 refers to the way the list of words is encoded in a tree like structure instead of a list. And later this tree is compressed into a usable array representation which in turn drastically lowers the required storage space for the data set. For this assignment I will present only how the tree is generated, populated and traversed, since this is the main part of the T9 solution.

Tree organisation

The T9 tree is set up in the following way. Each node has at most 27 branches (in reference to all the supported letters for the words). Such a tree is also known as a trie. The trie's root holds a reference to the first letter of a given word. In essence if the root has a branch at let us say index "0" then we know the first letter of the word is "a". Each next branch and its node relations encode the letters for all the words in the given set. Finally in order to know what paths over the tree give the actual words and not just a random possible string, each node also holds a boolean value showing if it is the last letter to an existing word.

Structured like this this trie eliminates the duplication of parts of some of the data since in a word list ”eight” and ”eighteen” would be separate entries while in here the path to ”eighteen” is also encompassing the path to ”eight”. Now while this might be a good consideration, in reality the size of this complex structure and especially the way we are going to implement it far exceeds the size of the text list file data set. Even still this approach gets us much closer the the actual way to minimize the storage space.

Setting up the T9 class

To begin with we are provided a Node code snippet for using in the tree. The T9 class itself needs only a single root node property. And then we are asked to implement a few helper methods to ease our work on the trie.

Character to code

The method `charToCode` given a character in the alphabet returns a value from 0 to 6. We will use these codes to address the branches of a Node’s array of next references. One thing to note is that in the given data set we are told that there are no words with the letters ”q” and ”x” since in this way including the 3 Swedish specific letters their total is 27 (3 per one of the 9 keys). With this in mind we have to track the fact that these gaps yield an offset in the sequence of ASCII values I am using to get the key representation. Also the said Swedish specific letters have completely unrelated ASCII values and as such are separated cases.

```
(...)
public int charToCode(char c) {
    if ((int) c - 97 < 16) {
        return (int) c - 97;
    } else if ((int) c - 98 < 21) {
        return (int) c - 98;
    } else if ((int) c - 99 < 24) {
        return (int) c - 99;
    } else if (c == 229) {
        return 24;
    } else if (c == 228) {
        return 25;
    } else {
        return 26;
    }
}
(...)
```

Character to code

The next method - `codeToChar` has in essence the reverse functionality of the previous one. The two of them are implemented in slightly different ways while the `charToCode` uses an if-else ladder making use of the specific indexes where an offset of the ascii values occurs, the `codeToChar` method is written using a switch operator. Both of them can be implemented with the respective other way.

```
(...)
public char codeToChar(int c) {
    switch (c) {
        case 0:
        case 1:
        case 2:
        case 3:
        (...)
        case 15:
            return (char) (c + 97);
        case 16:
        (...)
        case 20:
            return (char) (c + 98);
        case 21:
        case 22:
        case 23:
            return (char) (c + 99);
        case 24:
            return (char) (229);
        case 25:
            return (char) (228);
        case 26:
            return (char) (246);
        default:
            break;
    }
    return '\n';
}
(...)
```

Key to Index

This is a simple conversion from a supposed key press on the phone dial to an array index used when retrieving suggestions.

```
(...)
public int keyToIndex(char k) {
    return (int) k - 49;
}
(...)
```

Character to Key

The last method will be useful when it comes around to testing the functionality of the assembled trie. It maps a given letter to one of the nine phone dial keys.

```
(...)
public char CharToKey(char c) {
    switch (c) {
        case 'a':
        case 'b':
        case 'c':
            return '1';
        (...)

        case 'å':
        case 'ä':
        case 'ö':
            return '9';
        default:
            break;
    }
    return '\n';
}
(...)
```

Onto the Populating

In order to populate our trie on initialisation this is the constructor method of the T9 class:

```
(...)
public T9(String file) {
    this.root = new Node();
    try (BufferedReader br = new BufferedReader(new FileReader(file))) {
        String line;
```

```
            while ((line = br.readLine()) != null) {
                char[] row = line.toLowerCase()
                        .toCharArray();
                addWord(row);
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    (...)
```

A file is read line by line and each of the lines (words) after being converted into a char array is passed to the following `addWord` method.

```
    (...)
    public void addWord(char[] word) {
        int[] wordAsCode = new int[word.length];
        for (int i = 0; i < word.length; i++) {
            wordAsCode[i] = charToCode(word[i]);
        }

        Node itr = root;
        int wordindex = 0;
        while (wordindex < word.length) {
            if (itr.next[wordAsCode[wordindex]] == null) {
                itr.next[wordAsCode[wordindex]] = new Node();
            }
            itr = itr.next[wordAsCode[wordindex]];
            wordindex++;
        }
        itr.valid = true;

    }
    (...)
```

Each given word is transformed from an array of its letters into an array of the respective letters' codes. Using this array of codes we can traverse the tree while checking if the current Node's next array at the index of the letter is constructed or not. If it is we just go down it, if not we first construct it and perform this until there are no longer any letters in the given word, at that point we set the last node as the end to a valid word and we are done.

Getting information out of the tree

Finally in order to get mileage out of the tree we would like to be able to receive all possible word suggestions given a string of dialed key pad numbers. In order to perform this we will make use of two methods. The first of which is this `decode` function:

```
(...)
public ArrayList<String> decode(String key) {
    ArrayList<String> suggestions = new ArrayList<String>();

    StringBuilder keyInCode = new StringBuilder();

    for (char c : key.toCharArray()) {
        keyInCode.append(c);
    }

    collect("", this.root, keyInCode.toString(), suggestions);

    return suggestions;
}
(...)
```

It simply provides the primary `collect` method with an ArrayList of suggestions, where all possible words are added, and a character array representation of the given key sequence.

Bellow is the mentioned `collect` function:

```
(...)
private void collect(String word, Node subtrie, String key,
        ArrayList<String> valids) {

    if (!key.isEmpty()) {

        int i = keyToIndex(key.charAt(0));

        String curKey = key.substring(1);

        if (subtrie.next[i * 3] != null) {
            collect(word + codeToChar(i * 3),
                    subtrie.next[i * 3], curKey, valids);
        }
        if (subtrie.next[i * 3 + 1] != null) {
            collect(word + codeToChar(i * 3 + 1),
                    subtrie.next[i * 3 + 1], curKey, valids);
        }
        if (subtrie.next[i * 3 + 2] != null) {
            collect(word + codeToChar(i * 3 + 2),
                    subtrie.next[i * 3 + 2], curKey, valids);
        }
    }
    if (subtrie.valid) {
        valids.add(word);
    }
}
(...)
```

This is the most complex of all the code segments in this assignment. It receives the following 4 arguments:

- a word string storing the letters from the path of the node traversed so far (meaning that when we reach a valid node this will be a proper word), and on the initial call is simply an empty string;
- a Node containing the trie or subtrie left to be traversed;
- a string with the sequence of keys corresponding to the letters of the word we are searching for;
- a reference to the array list where we add all found valid words from the procedure.

The function is implemented recursively and executes in this way. Unless there are no longer any letters in the `key` argument we calculate an index from the first of the characters in the string, and we remove the same character from the key in other to pass down a current representation of the letters left to go through. At this point since each key could invoke three different letters we do 3 separate recursive calls on each of which we pass the subtrie found at the index value timed three plus an offset of either 0, 1 or 2 to branch to each of the possibilities. Finally after all of this is performed we simply check if at the point we are in the trie the node is valid, since if it is we are certain we have found one of the words fitting our search query.

Testing the trie and closing

Logically speaking everything seems sound, but it is always reassuring to practically see that everything works. To do this I basically wrote a program to turn each of the entries of the "kelly.txt" file into their 9 key pad typed representation and ask the T9 class to decode them. In essence this would return all of the words in the file along with any other words which also fit the coded representation. This did in fact yield correct results and since it is a long output here I present only the first few lines:

```
15261 :
        andra
        an
        bo
16755 :
        arton
        art
        arv
        brunn
        brun
11462 :
        bakre
        bak
        bal
```

This assignment was very pleasant to work on and seeing the structure perform as intended, even though at first it might appear convoluted, gives a great sense of satisfaction. It also does a great job at instilling an intuition about how the specific use of data structures aids in solving complex issues, which might appear not so trivial at first glance.