T9

Ayda Ghalkhanbaz

Fall 2023

Introduction

This assignment is about implementing an algorithm called "Text on 9 keys" or briefly, "T9". This algorithm is used in mobile phones with a numeric keypad, where keys 1 through 9 represent sets of letters. In this way, by entering the input "43556", we can output the word "hello". This algorithm has made texting easier for the user due to its predictive features. For instance, if the user starts to input "32..." the algorithm predicts the user tries to write something starting with "ge", "he", "id" or "if", and it would simply suggest relevant words. In the following sections, we will delve into the implementation details of such an algorithm.

Trie: a strange tree

In implementing this algorithm, a tree structure is utilised. By employing a file containing Swedish words, we can populate a strange tree (which we will call it "Trie" from now on). Notably, we use all Swedish letters including "åäö" while excluding the letters 'q' and 'w'. The implementation of the `Trie` class needed some definitions which we will talk about in detail shortly. These definitions include: a `Node` class and methods such as `add`, `decode`, `collect` etc.

Node

The idea of this class might be a little confusing at first, but drawing a representation of such a tree made it easier to understand. Each node in the `Trie`, has two key attributes; a list of nodes called `next` and a boolean `valid`.

Since we are dealing with 27 letters ('a' to 'ö', excluding 'q' and 'w'), then each node should have 27 possible `next` paths to represent all letters. The `valid` attribute is `true` only if we reach a leaf indicating the presence of a valid word which corresponds to our file. For instance, if we have the word "hej" in the list but not "he", only the node representing the letter 'j'

along the path "h>e>j" would have a `true` `valid` attribute. Below you can find the fundamental structure of the `Node` class.

```
private class Node{
        public Node[] next;
        public boolean valid;

        public Node(){
            next = new Node[27];
            valid = false;
        }
    }
```

Code, Index and Key

This part of the task might be the simplest segment of the assignment. It involves implementing some helper methods that will come in handy during the implementation.

Firstly, a method named `getCode()` was implemented. This method takes a character as a parameter and returns an integer representation of it. For example, the letter 'a' is represented by 0, 'b' by 1 and so on , with 'ö' represented by 26. This method consists of multiple if statements and utilises the corresponding ASCII code of each character in order to calculate the returned integer.

The code snippet below demonstrates the `getCode()` method.

```
private static int getCode(char c){
        int ret = -1;
        if(c >= 'a' && c <= 'p') // a to p
            ret = (int) c - 'a';
        else if(c >= 'r' && c <= 'v') // r to v
            ret = (int) c - 'a' - 1;
        else if(c >= 'x' && c <= 'z') // x to z
            ret = (int) c - 'a' - 2;
        else if(c == 'å')
            ret = 24;
        else if(c == 'ä')
            ret = 25;
        else if(c == 'ö')
            ret = 26;
```

```
        return ret;
    }
```

The second helper method works in the opposite way, in other words it takes an integer as argument and returns the corresponding character. Below you'll find the implementation of this method.

```
    private static char getChar(int c){
            char ret = ' ';
            if(c >= 0 && c <= 15) // a to p
                ret = (char) (97 + c);
            else if(c >= 16 && c <= 20) // r to v
                ret = (char) (97 + c + 1);
            else if(c >= 21 && c <= 23) // x to z
                ret = (char) (97 + c + 2);
            else if(c == 24)
                ret = 'å';
            else if(c== 25)
                ret = 'ä';
            else if(c == 26)
                ret = 'ö';

            return ret;
        }
```

The next method, called `getKeyofChar()`, returns an integer representing the buttons 1 to 9 on the phone, which is associated with characters. Each key on the phone corresponds to 3 letters. The code snippet below shows how the method is implemented.

```
    private static int getKeyofChar(char c){
            return (getCode(c)/3) + 1;
        /* we want to divide chars into sets of 3.
         +1 ensures that we stay in range 1 to 9 (keys)
         1      2     3     4     5     6     7     8     9
         abc   def   ghi   jkl   mno   prs   tuv   xyz   åäö*/
        }
```

The last but not least method is called `getIndexofKey()`, which simply takes a key (1 to 9) and returns the index of the key in an array which is `key-1`.

Adding Words

After an unsuccessful attempt for implementing the `add()` method recursively, I decided to implement the add method as the code snippet below illustrates.

```
public void add(String word){
        if(root == null){
            root = new Node();
        }
        int i = 0;
        Node curr = root;
        //go through each char in the word
        while(i != word.length()){
            int index = getCode(word.charAt(i));
            // create a new branch if it's empty
            if(curr.next[index] == null){
                curr.next[index] = new Node();
            }
            // set curr to the recently added
            curr = curr.next[index];
            i++;
        }
        // set the flag valid to true (a valid word)
        curr.valid = true;
    }
```

The `add()` method accepts a string as a parameter. Starting from the root of the tree, we iterate through each character in the string, find its corresponding index in an array using the `getCode()` method and try to find a suitable place for it.

For example, when adding the word "hej" to the tree, we begin with the letter 'h'. The corresponding index of this letter in the array is 7. With the help of the `curr` variable, we navigate through the `Trie`. If the 7th spot in the array is null, then we create a new node there and continue with the letter 'e'. This process continues until there are no other letters left in the string. At this point, we set the `valid` flag to `true`, indicating that this path ends with a valid word.

Searching for Words Given a Sequence

This part was likely the most challenging task of the assignment. The idea is that given a key sequence, we should find all words corresponding to the

sequence. At first glance, it might seem simple, but it took a considerable amount of time to debug and understand what was happening.

For implementing this part, I chose to use the recursive approach. Firstly, a method named `decode()` was implemented in the `Trie` class. This method is really simple. It creates an empty list and an empty string, calls another method named `collect`, and finally returns a list with words associated with the key sequence. You'll find the code for the `decode()` method below.

```
    public ArrayList<String> decode(String keySeq) {
            ArrayList<String> list = new ArrayList<>();
            String word = "";
            root.collect(keySeq, word, list, 0);
            return list;
        }
```

The magic of filling the array with possible words occurs in a method defined in the `Node` class. The `collect` method takes two strings, a list and an index as arguments, and adds the valid words to the list. The code for the `collect` method can be found below.

```
public void collect(String keySeq, String word,
                ArrayList<String> list, int index){
        // base case: if the word is valid then add it
        // to the list,otherwise return(no word found)
        if(index == keySeq.length()){
            if(valid){
                list.add(word);
                return;
            }
            else{
                return;
            }
        }
        // keySeq = "324" -> '3' -> key = int 3
        int key = Character.getNumericValue(keySeq.charAt(index));
        // get the index of key in the tree
        // ex: if key is 3 then index is 2
        int indx = getIndexofKey(key);
        // iterate over all three branches for the given key
        for (int i = 0; i < 3; i++){
            String possiblePath = "";
            if(next[(indx*3)+i] != null){
```

```
        //indx*3+i iterates over all branches of a key
                possiblePath = word + getChar((indx*3) + i);
                next[(indx*3)+i].collect(keySeq,
                                possiblePath,list, index+1);
            }
        }
    }
```

First, a base case is defined. It checks if the end of the `keySeq` has been reached. If this is the case and there is a valid word, then we should add the word to the list. Otherwise if there is no valid word, the method exits the call.

However, until we reach the end of the `keySeq`, we take each character in the `keySeq` string, convert it to an integer and find its index in the array using the implemented method `getIndexofKey()`. After that, we iterate over all three branches that the index is associated with to find the relevant words.

For each iteration we add the corresponding letter to an empty string named `possiblePath` and then call the `collect()` method recursively to continue this process until it reaches the end.

To clarify the function of this method, here comes an example. If we want to find the words associated with the key sequence "324", we begin with '3'. The '3' in the sequence indicates that the user has pressed button 3 on the phone and is trying to type a word starting with letters 'g', 'h' or 'i'. These letters exist in the branches 6, 7 and 8 (that's why we calculate `indx*3 + i` in the for loop). For each of the branches, if there are more branches to examine, we go deeper and deeper in the `Trie`, until we reach a valid word or we find no valid words. In case of the existence of a valid word, it is added to the list. This process iterates over all branches, and finally the `collect()` method returns a list hopefully filled with relevant words.

In conclusion, I have run the program with the example above, decoding "324". The result is a list containing the words "hej" and "hel", which these words are valid words according to the provided file in the assignment.