T9 ”predicative texting technology” in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment we implement something similar to the T9[1] system that was used in old mobile phones for texting. We will not follow its implementation strictly (e.g. we won’t be as memory efficient) but the main task is the same. We have a list of the 8000 most common Swedish words for our database, and we will only use the Swedish letters a to ö without q and w.

Implementation

chars,codes,keys and indices

First we implemented some supporting methods for translation between characters[2], codes[3], keys[4] and indices[5]. I implemented a method called toCode that takes an ascii character and converts it to a number in the range 0..26.

```
public int toCode(int a){
    int[] chars = {97,98,99,100,101,102,103,104,105,106,107,108,
    109,110,111,112,114,115,116,117,118,120,121,122,229,228,246};
    for (int i = 0; i<27; i++){
        if (chars[i]==a) return i;
    }
    return -1;
}
```

I also implemented the reverse very similarly, that converts a code to an ascii character.

[1] T9 wikipedia site

[2] the characters ’a’..’ö’

[3] the integer representation of the characters, numbers 0..26

[4] the key you would press on a phone, so 1 for a,b,c, 2 for d,e,f etc

[5] the integer representation of our keys in the range 0..8 used as indices for the search

My toIndex method takes a key and returns index=key-1.
And finally toKey takes a code and returns which key has to be pressed for it. This will be useful for testing my program in the end for the entire data set. For these two methods I didn't need to check if the argument is in the correct range, since toKey gets its value from the toCode method so it will be a valid number and the toIndex gets its value from toKey (or from the keyboard, but then I trust myself to enter valid numbers).

```
public int toKey(int k){
    int[] s = {0,3,6,9,12,15,18,21,24};
    int i = 0;
    while (i<s.length && s[i]<=k){
        i++;
    }
    return i;
}
```

the trie

The main data structure will be a specific type of tree called a `trie`, where the words will be stored as paths. It will consist of nodes that have a boolean valid variable indicating whether the path up to that point is a valid word and a reference to the next (up to 27) branches. The index of a node in the current level is the code for its char value, so for a word that starts with 'a' we would look at the first element of the first level and look for the second character in current[0].next[x]. All elements start out as null and will be added as we read the file. This implementation makes searching for words quick and easy.

populating the trie

My main problem was working with the Swedish characters (åäö). My windows terminal seemed to have a problem displaying them, so I had a hard time determining whether the internal structure worked correctly. Reading from the file as I did before also didn't work, but I found a solution on stackoverflow (link) and used the following setup for reading from the file:

```
public T9(){
    String file = "kelly.txt";
    this.root=new Node();
    try (BufferedReader br = new BufferedReader(new
    InputStreamReader(new FileInputStream(file), "UTF-8"))) {
        String word;
        while ((word = br.readLine()) != null) {
            Node current = this.add(toCode((int)word.charAt(0)));
            for(int i = 1; i < word.length()-1; i++) {
                current = current.add(toCode((int)word.charAt(
                i)));
            }
            current.addFinal(toCode((int)word.charAt(
            word.length()-1)));
        }
    } catch (Exception e) {
        System.out.println(e.toString());
    }
}
```

As you can see, I decided to separate adding the last character of a word which is when the node needs to be marked valid. I did it like this because I thought it would be more efficient than checking if we're at the end when adding each node.

searching for words given a sequence

Now that our `trie` was set up and ready to go, we could finally focus on the main functionality of the T9, which is finding all possible words for a given sequence of keys. This is slightly more complicated, so I decided to draw it out on paper to see what needs to be done. I implemented a decode method that takes a string of the sequence, initializes an `arraylist` for the words we will return and calls collect on the root. My collect method will be the one running recursively to collect all words up to a node.
`Collect()` takes four parameters, the string with the sequence, an integer for the position of the current character in the String, a reference to the `arraylist` that we will write to, and another String with the path. At each node, it will call itself up to 3 times, for the three possible letters of the corresponding key. When we reach the end of the sequence, we have our position variable be equal to the length of the string. This is when we check whether the path is a valid word and if so, we add it to our `arraylist`.

```
public ArrayList<String> decode(String s){
    ArrayList<String> result = new ArrayList<String>();
    String path="";
    int pos=0;
    this.root.collect(s,pos,result,path);
    return result;
}

public void collect(String s, int pos, ArrayList<String> result,
String path){
    if (pos>=s.length()){ //reached the end of the sequence
        if (this.valid==true){
            result.add(path);
            return;
        }
        else return;
    }
    int index = toIndex(s.charAt(pos)-'0');
    pos++;

    String one = path;
    String two = path;
    String three = path;

    if(this.next[index*3]!=null){
        one+=(char)fromCode(index*3);
        this.next[index*3].collect(s, pos, result, one);
    }
    if(this.next[index*3+1]!=null){
        two+=(char)fromCode(index*3+1);
        this.next[index*3+1].collect(s, pos, result, two);
    }
    if(this.next[index*3+2]!=null){
        three+=(char)fromCode(index*3+2);
        this.next[index*3+2].collect(s, pos, result, three);
    }
}
```

tests

To test if my program produces the desired outcome for the entirity of my data, I read the same text file again with the 8000 words, ”encode” the words as sequences of keys and print the resulting `arraylists` as shown in the code below:

```
try (BufferedReader br = new BufferedReader(new InputStreamRea
der(new FileInputStream("kelly.txt"), "UTF-8"))) {
    String word;
    while ((word = br.readLine()) != null) {
        System.out.print(word+": ");
        String seq="";
        for (int i = 0; i<word.length(); i++){
            seq+=t.toKey(t.toCode((int)word.charAt(i)));
        }
        System.out.println(t.decode(seq).toString());
    }
```

For most words this returns just the word itself, but for some key sequences it can return more than one, for example "4341" results in [kika, kila, lika, lila] being printed. This confirmed that my program performs the intended task.

Summmary

In this assignment I got familiar with a new data structure (`trie`) and implemented the base of the T9 system and practiced depth-first searching trees. I also learned a lot about encodings on the way there. In reality, in my case the datas spacial complexity grew by implementing the tree, but it was good for practice. The T9 system used careful engineering and a lot of tricks to fit all this data into a much smaller memory.