T9

Emanuel Paraschiv

Fall 2024

Introduction

In this assignment we will implement a prototype of the T9 mechanism used for cell phones. Compared to older cell phones when we had to press multiple times on specific buttons in order to type a letter, T9 came as an innovation which allowed the user to type only once making use of predictive text.

Implementation

Our goal is to use the buttons of a phone to write text. We will name them 'keys' each of them having 3 letters, and they will take values from 1 (abc) to 9 (åäö). For example pressing a sequence of keys `32445` should return `'hello'`. Notice however that in this experiment we start from key 1 and not 2 as in a traditional phone.

In order to represent the words in our program we will use a new tree data structure called 'Trie'. This tree will first consist of one Node called 'root' that has 27 branches, being connected to the letters of the alphabet:

```
public Node() {
    next = new Node[27];
    valid = false;  // will talk about this later
}
```

Representing words in this Trie follows a simple pattern: we start from the root and go down in depth, one level for each letter. This way, every word is just a 'path' from the root to the last letter in it.

The logic of representing words can be more simple to conceptualize with an example. Let's say we want to represent the word 'java'. In order to do so, we start from the root, then we access 'j'. Then, because the next value is `null` we will initialize a new array of 27 nodes, from which we select 'a'. We continue in the same manner for 'v' and 'a' and in the end we do a simple trick: to confirm that this is a word we will set the last node 'a' as `true`.

With this logic in mind we can start thinking about what we need. Firstly, because we want to access different letters, we can assign to each of them a code, which is simply the position in the array. For example, we can come up with a method like this:

```
private static int char2code(char w)
        switch (w)
            case 'a' :return 0;
            ...
            case 'ö' :return 26;
        return -1;
```

We will also need one method which does the exact opposite, called `code2char()` (given the code it will return the corresponding letter).

Adding Words

Now that we have a Trie and understand the logic, we can start to think of a way to add the words using Java. If we are given a String we will have to turn it into individual characters and add them to the Trie in the manner described previously.

We will begin by storing the characters in an array. Then we will start from the root with the first character, `chars[0]`. We will turn that into an integer using `char2code` and find that corresponding position in the tree. Now root has a path to the first char: `(root` → `chars[0])`.

Then we will move to next position in the array of characters of the word, `chars[1]`. Again we will turn it into an int, but because the tree has next branch `null` we will have to construct it (i.e. a new array of 27 elements). This way we dynamically construct he Trie as needed to fit the given words.

Then we continue the process until the last character, and when we reach the end, we set the it as valid to know that this is a real word.

The path is the tree would look something like:

`(root` → `chars[0]` → `...` → `chars[n] )` for a word with n letters.

```
public void add(String word){
    Node current=root;
    char[] chars=word.toCharArray();
    int j;

    for(int i=0; i<chars.length;i++){
        j=char2code(chars[i]);
        if (current.next[j]==null){
            current.next[j]= new Node();
        }
```

```
            current=current.next[j];
        }
        current.valid=true;
    }
```

Here comes the problem with the memory, if we have a lot of words then we'll have to construct a big number of arrays. Also, the longer the word is, the deeper the branches of the Trie.

Lookup

Now the lookup method will make use of the Trie but can be a bit more tricky. This is happening because now instead of working with one word only we have to retrieve multiple words that match the given key sequence. For example, there can be multiple words given the sequence '5643' and the `lookup()` method should make sure that it returns all of them.

To start with, we will need 1 more method to deal with converting the key buttons of the phone (from 1 to 9) to indexes (0 to 8). We could build something simple like a method to return `'value - 1'` given a `value` input. However, we will see that it is more advantageous to work with chars directly (so we don't have to convert them after), thus we will make use of switch statements once again to get an int from a char input:

```
     public static int key2index(char key)
        switch (key)
            case '1': return 0;
            ...
            case '9': return 8;
            default: return -1;
```

We will start by implementing a method called `decode()`:

```
    public ArrayList<String> decode (String seq){
        ArrayList<String> list=new ArrayList<>();
        char[] chars=seq.toCharArray();
        int[] ints = new int[chars.length];
        int j;
        for(int i=0; i<chars.length;i++){
            j=key2index(chars[i]);
            ints[i]=j;
        }
        // add words to the list
        collect(root,list,ints,0,"");
        return list;
    }
```

As we can see, the input will be a String of the form "12345", a sequence of digits from which we have to find the matching words.

Every time we find a word, we will add it to an ArrayList which will be returned by the method at the end. This list will contain all possible words.

We will take our string input and turn it into an array of characters. Then we will convert them to integers (codes to access elements in the Trie) and then call a method to `collect` all possible words, presented below.

This method will take all possible words, add them to the list, and after it is done we will return the list from `decode()`.

```
public void collect(Node current,ArrayList<String> list,
int[] ints, int i, String s){
   //base case
   if (i == ints.length) {
       if (current.valid) {
           list.add(s);
       }
       return;
   }
   //current code
   int cur=ints[i];
   int k=0;
   while(k<3){
       //check each of the 3 digits
       Node next= current.next[cur*3+k];
       if(next!=null){
           char nextc=code2char(cur*3+k);
           collect(next,list,ints, i+1,s+nextc);
       }
       k++;
   }
}
```

The `collect()` method is recursive and works in the following way: start from the root and examine multiple paths to find valid words.

As we know already, every key holds 3 letters(1-abc, 2-def etc.). What we do is convert that key to its corresponding index ($1 \to 0$, $2 \to 1$ etc).

As 1 contains abc (abc has codes 012), in order to access the letters we will use the operation ($\texttt{index of key 1} = 0) * 3$ for 0 (which is 'a') then ($\texttt{index of key 1} = 0) * 3 + 1$ for 1 (which is 'b') then ($\texttt{index of key 1} = 0) * 3 + 2$ for 2 (which is 'c').

Then the method will call itself recursively and each time letters are concatenated into a string which grows in size with every iteration. When we reach the input length we check the last bit, and if it is valid, then we have a word to add to the list. If not we simply explore the other cases.

This method will therefore explore every path (every combination of 3 letters) and return only the valid words.

Testing specific cases

Now it is time to do some testing to confirm our implementation. In order to do that, we will use the file 'kelly.txt' present in the assignment.

First we can populate our tree by creating an instance and call the `add()` method on it.

```
T9 t9 = new T9();
// create buffer reader for the file, then run add:
t9.add(word.trim());
```

For the decoding, we can create a list to store the words:

```
ArrayList<String> list = t9.decode(sequence);
```

Now we can randomly choose a word present in the list, such as 'lott'.

Manually we can figure out that this word has the code '4577'. Now, if we call the decode method with this input, we get 2 outputs: 'lott' and 'knut'. This means that the word 'knut' also has the same sequence of keys, and our program is able to return it.

Let us now take another example, let it be '4567'. If we call the decode with this sequence, the words 'kort', 'korv' and 'kost' are returned. And for '327' we get the words 'get' and 'het'.

This confirms that our implementation of T9 is able to not only return one word that matches the sequence, but all of them.

Conclusion

In this experiment we discovered a thorough way of implementing the T9 system with its basic functionalities. By using a tree we were able to implement functions that are able to add and search, encode and decode, words.