T9

Fatima Mohammad Ali

Fall 21/10/2023

Introduction

A *Text on 9 keys* system, also known as *T9*, is a predictive text system that keeps track of a bunch of words and suggests those when given a sequence of keys. On older mobile phones with numerical keypads, each key represented three alphabetical letters and/or characters. Pressing the sequence "43556" could hence result in the word "hello" being suggested. In this assignment, a "Swedish-friendly" T9 system was implemented using a specific tree structure, so-called a *trie*. The trie represented all words given a word-list and with some key-value modifications, it was possible to implement something that is close to a T9 system.

T9 keys

To implement the T9 structure, several keys had to be kept track of in different methods. First off, there were the "pressed" **keys** (type `char`) ranged '1' - '9'. Those were converted to **indices** (type `int`) ranged 0-8 in a `getindx(char key)` that simply converted the key *character* to an *integer* and *decremented* it with one, which returned e.g. key '3' as index 2.

There were also 27 **characters** (type `char`) 'a'-'ö' (excluding 'q' and 'w'). Those were converted to **codes** (type `int`) ranged 0-26 ('a' returned 0,..., 'ö' returned 26). This was done in an `int code(char ch)` method that had switch cases to return each conversion. Codes 0-26 were also converted to their corresponding character in reversed switch cases, in a `char revocode(int key)` method.

Each key represented three letters. Key '1' represented 'a', 'b', and 'c', while '2' represented 'd', 'e', and 'f', and so on. To get a *key* given a character, the following method was implemented:

```
private String encode(String word){   //e.g. word = "trött"
    String keyseq = "";   //initialize an empty string to add chars on
    int indx = 0;
```

```
    while(word.length() != indx){ //for all chars of the word
        char key = word.charAt(indx); //get char at indx of word
        int k = 1;
        for(int i = 0; i <= 24 ; i+=3){ //check relevant range of characters
            if(i <= code(key) && code(key) <= i+2){ //check key for the letter
                char c = (char) (k + '0');  //converts int to char
                keyseq = keyseq + c;    //add char (letter) to string
            }
            k++;
        }
        indx++;
    }
    return keyseq;  // returns "76977"
}
```

All key methods were implemented in a `T9` class and utilized when constructing the *trie*.

Trie implementation

To implement the trie, a `Node` structure was initialized in the `T9` class. The `T9` class kept track of the `root` node, and each node in the node class had a node array (`Node[] next`) and a flag (`boolean valid`). The flag was initialized as `false` and the node array was declared with size 27, since it held all 27 characters.

Add method

Adding words to the trie required starting at the root node and working down the branches. If the branch was not null, then the character gave path to a word (the word itself should be given as a *String*). So, to add a word to the trie, the *path* through characters should be returned.

The implemented `add` method in this assignment was done recursively in the following `add` method in the `Node` class (it was called as a one line method in the `T9` class):

```
public void add(String word, int indx){ //word and its char indx (for recursion)
    int lastindx = word.length();   //length of word
    if (indx == lastindx) { //after reaching the last char
        valid = true; //set the valid flag as true
        return; //stop recursion
    }
    char ch = word.charAt(indx);    //returns char att index of string
    int charindx = code(ch); //returns corresponding index of char
```

```
    if(next[charindx] == null)  //if the branch of char is empty
        next[charindx] = new Node();    //contstruct branch

    //recursively add the next char of the string
    next[charindx].add(word, indx + 1); //indx + 1 to check the next char
}
```

When given a `String word`, the method added the path of it to the trie by checking branches. Each character of the word had to be checked individually. This was done by using the built-in method `charAt(indx)` (in `java.lang`) that retrieved a character at a given index of the string. The *index* of the *character* (letter) was also needed and retrieved by using the previously implemented `code(char ch)` method. When the index was available, the corresponding branch of a node was checked. If the branch was empty, then a new branch was constructed for the character. If the branch was not empty, then it meant that the path (the character) was already present in the trie. The recursion continued until the last character of the word was reached. When all characters were added forming the word, the `valid` flag was set to `true` and the recursion stopped. That is how a word was added to the trie.

Lookup method

Some words may have similar sub-paths and therefore be suggested when pressing a certain key sequence. To suggest/collect these potential words, a `lookup` method that returns a list of words of a key sequence was implemented. The following `collect` method was implemented recursively in the `Node` class:

```
public void collect(ArrayList<String> list, String keyseq, String word, int indx){
    int lastindx = keyseq.length(); //length of key seq.
    if (valid == true)  //when a valid word is found, add to the list
        list.add(word);
    if (indx == lastindx)   //when the key seq is done, stop recursion
        return;

    char curkey = keyseq.charAt(indx);  //gets char at indx of string
    int keyindx = getindx(curkey);  //gets key index

    int c = keyindx * 3;    //branch
    for (int i = c; i <= c + 2; i++) {  //checks the three branches
        if (next[i] != null) {  //if branch is not null
            //add it to the string
```

```
            char add = revcode(i);  //returns letter (char) on branch
        //adds character to the end of string, increments indx to check next key
            next[i].collect(list, keyseq, word + add, indx + 1);
        }
    }
}
```

The lookup method was called in the `T9` class as follows:

```
public ArrayList<String> decode(String seq){
    ArrayList<String> list = new ArrayList<String>();
    String word = "";   //initialize an empty string
    root.collect(list, seq, word,0);
    return list;    //return list with suggested words
}
```

The method `decode(String seq)` was given a key sequence and returned an `ArrayList` of all possible words that could match the sequence. This was done be sending an empty `String word`, the key sequence, and index 0, to the recursive `collect` method. `collect` started at the root node and collected all possible words given the sequence. Similiarly to the `add` method, the first character of the key sequence was retrieved using `charAt(indx)`, then the corresponding key index (integer 0-8) was retrieved. For instance, the sequence "2134" would check the character '2' and retrieve its index int = 1. This index represented three branches (characters): index*3, index*3 + 1, and index*3 + 2. For instance, branch 1*3+2 = 5 represented the letter 'f'. Then, each branch was checked. If the branch was not null (the character was a part of a word/path), its character would be added to the empty string. When going down the branches recursively, the checking and adding would slowly return valid words that were added to the list (each time the flag `valid` was `true`). Therefore, all possible words were returned when given a sequence.

Results

The trie was populated with words from the given "Kelly" list. All words were tested to be encoded, respective decoded, and all suggestions appeared correctly as they should. For instance, testing the code could be done as follows:

```
String s = "kul";
System.out.println(s + ", encoded: "  + trie.encode(s) +
                       ", decoded: " + trie.decode(trie.encode(s)));
```

And trying for different words would give:

```
kul, encoded: 474, decoded: [ju, jul, kul]
att, encoded: 177, decoded: [att, av]
programmera, encoded: 66536155261, decoded: [program, programmera]
```

Reflections

This assignment was a bit mind-boggling to grasp at first (with many papers going to waste to draw up the trie...), but it was really fulfilling when I finally got around and understood the structures. Something that made me feel rather regretful though was the use of some built-in functions (`charAt(indx)` and `String.lenght()`), since I am not sure if the usage of them is allowed or not (even though using an `ArrayList` was suggested). Regardless, this was a refreshing assignemnt to work on.