T9

William Frid

Fall 2023

Introduction

Back in the days when phones had 3x4 numerical keypads, there was a technology called T9 released. Its name meant "Text on 9 keys" and was a lightweight (compared to nowadays) text prediction technology. Text precision has since come a long way, but yet, we're going to explore T9 since it can be written in a somewhat simple way using hashing and tries (a type of strange tree).

This report will focus on how the code was written and how it works, rather than benchmarking since how one might build different data structures is in focus rather than optimizing speed. Note that the code written in this assignment is Java 8 and that some might be left out as its purpose is clearly described in the text and/or is simply helper functions.

T9 Basics

Before diving into building the text prediction program one must understand the 3x4 numpad. The basic idea is that each number (1-9) corresponds to 3 letters. Number 1 corresponds to the letters "a", "b", and "c", number 2 to "d", "e", and "f", and so on... This project made use of an array containing all the letters used. That is, all letters in the Swedish alphabet excluding "q" and "w" (thus making it 27 long). Why 27? Because that makes the word easier since we can simply calculate the possible letters using the formulas $(n * 3 + 0)$, $(n * 3 + 1)$, and $(n * 3 + 2)$. Similar formulas will be seen in this report. This can also be carried out backward, that is convert a character into the number it would be represented by on the number. More on this under the section "Testing The Tie" as we there make use of this.

A Trie

A trie - as mentioned in the introduction - is a strange tree-like data structure, similar to what we've seen in the previous assignment of `ID1021`. What makes the trie so unique is that instead of having a right and left reference,

it has in this case 27 and a boolean that tells if it is the end of a valid word. Each of the 27 nodes it can refer to stands for a letter in the alphabet array. The structure of each node is seen below.

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

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

    private void addWord(String s) { ...}
}
```

Populating The Trie

This tree was populated with words loaded from a given CSV file upon initialization using a recursive method inside the node class called `addWord()`. This method started off with selecting the first character in the string and based on that went further down the tree via the node with an index equal to the selected character's code (see method `getCode()` below) with the given string as argument minus the first character. It does so recursively until there are no more letters left in the string, and when that happens, it marks the current node as valid. Marking it has completed a word.

```
private static int getCode(char character) {
    for (int i = 0; i < alphabet.length; i++)
        if (Character.compare(alphabet[i], character) == 0)
            return i;
    return -1;
}
```

Searching/Decoding The Trie

To search in a trie is more complicated than populating it. Recursion is once again needed, but this time a different helper functions similar to `getCode()` (seen above). Since the words imported are more encoded into the tree than actually stored in it directly, we'll going from here on out, call it "seconding" instead of "searching".

One way of writing the method `decode()` is seen in the snippet below. The main idea is to generate an ArrayList holding strings. Then pass this list, alongside the given input to a method (with the same name in this

case), that will call upon itself recursively. In that method, the first step is checking if a word is found, if so, it only adds the word to the list. But how does it build the string saved to the list? This is where the other part of the if-statement comes into play.

The other part of the if statement (first part), starts off by getting the integer number of the first character in the input. Generate three new integers. These integers all correspond to the characters the input digit could lead to on a T9 numpad. It then selects the three nodes with the generated indices, and checks if they each exist (separately). Let's assume that node `nn0` exists, then the method calls upon itself, but this time with modified arguments. The node is no longer the root (as it represents the next character), the input has been stripped of the first character, and finally, the empty string is given a character. The character added to the string is based on the current node index and is decoded using the method `getChar()`.

```
private ArrayList<String> decode(String input){
    ArrayList<String> lst = new ArrayList<String>();
    decode(root, lst, input, "");
    return lst;
}
private void decode(Node n, ArrayList<String> last,
    String input, String s) {
    if (input.length() > 0) {
        int key = input.charAt(0) - 48;

        int i0 = key*3 + 0;
        int i1 = key*3 + 1;
        int i2 = key*3 + 2;

        Node nn0 = n.next[i0];
        // Same for nn1 & nn2

        if (nn0 != null) {
            String key_rest = input.substring(1, input.length());
            decode(nn0, lst, key_rest, s + getChar(i0));
        }
        // Same for nn1 & nn2

    } else if (n.valid) { // Word found.
        lst.add(s);
    }
}
```

Upon completion, the method returns the ArrayList it once created in the beginning, containing all possible words. The data given and encoded

into the tree is approximately 8262 of the most common words. Meaning that in some cases, decode might actually return multiple words. Especially when decoding a word that is very similar to other words.

Testing The Trie

To test the code, a printing method was written. It once again loaded all the data from the given CSV file and encoded each of the words into the numbers they would be represented by on a T9 numpad. The is called upon decode for each of these. The output was the correct amount of words, and as expected, in some cases the printed ArrayList contained additional words that were close to the correct one. To encode the characters into numbers, a method was written called `charToKey()` (shown in the snipped below).

```
private static int charToKey(char character) {
    int i = 0;
    int j = getCode(character);
    while (j >= 3) {
        i++;
        j-=3;
    }
    return i;
}
```

Discussion

Writing the code for this report was straightforward. Thanks to clear instructions and just enough wiggle room, there was barely any room for mistakes. the biggest challenge was getting the decoding to work properly and figuring out how to check that the words had been properly encoded into the trie since there was no simple build method to print it out.

Something that could be interesting for the report would be testing how the speed of the algorithm would be affected by adding more words to the trie. One could expect that the increase should be large at all since the algorithm stops its decoding when there are no more inputted numbers to search with. Thus, adding words would probably only add time to the decoding for cases where multiple words would be suitable for the given input.

Conclusion

T9 is a genius lightweight word completion software that's relatively simple to code. It makes great use of a modified tree-like structure that allows for unique ways of encoding data. Since working with this type of encoding

and decoding requires a lot of recursive calculation, helper functions are a must to avoid a large amount of repeated code. Furthermore, the trie structure requires less space than multiple separate strings and comes with great speeds.