T9

Fredrik Berzins

Fall 2023

Introduction

T9 i s away of decoding numbers into letters and vice versa. It was predominately used for older phones when typing with the key pad where each number would correspond to a few letters. it would then match the possible letter combinations with a dictionary to get a actual word in the end. In this implementation it each number 1 to 9 will correspond to tree letters a to ö and to not have to work with more then tree letters per key/number q and w will be removed.

Implementation

T9 explanation

The T9 implementation has three phases initial fill, add and decode. The T9 implementation is based on a tree of nodes that each can have 27 branches one for each letter that follows.

Initiation construction

For the initial construction of the tree, all words are read and isolated as char arrays for easier incrimination thru the array. when adding the letters to the tree later. As seen below the constructor for the tree is mostly code to read the file and reformat it to a char array for each add operation.

```
public T9_tree(String file) {
    root = new Node();
    try (BufferedReader br = new BufferedReader(new FileReader(file))) {
        String line;
        char[] word;
        while ((line = br.readLine()) != null) {
            word = line.toCharArray();
            root.add(root, word, 0);
        }
```

```
            br.close();
        } catch (Exception e) {
            System.out.print("\t file " + file + " not found");
        }
    }
```

Add

The add function works recursively and and has two methods two it the first is at the tree level and calls for the node level add with starting case of root, and index = 0;. The second part is the node level part, this part checks if it's done with that word array if so it changes the valid boolean to true. If the word array is not gone thru it check the index of the letter, this is to be able to create the new node in the correct spot in the array. The last step is to recursively call the add function with the newly created node as input argument and increment wordIndx. For clarity all add related code is below beginning with tree class add method.

```
public void add(String line) {
    char[] word = line.toCharArray();
    root.add(root, word, 0);
}
```

Below this is the node class add method.

```
private void add(Node curr, char[] word, int wordIndx) {
    char[] letters = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm
    if (wordIndx >= word.length) {
        curr.valid = true;
        return;
    }
    char letter = word[wordIndx];
    int index = - 1;
    for (int i = 0; i < letters.length; i++) {
        if (letters[i] == letter) {
            index = i;
            break;
        }
    }
    if (index == -1) {
        return;
    }
    if (curr.next[index] == null) {
        curr.next[index] = new Node();
    }
```

```
    add(curr.next[index], word, (wordIndx + 1));
}
```

Decode

Decode is quite a lot more complicated and split into three methods decode, lookup and searchBranch. The initial method is decode it takes in keystrokes as one string convert it to a int array and then calls lookup with a starting condition. Lookup is goes thru all three options if they arr not null until the input keys are gone thru, after all keys are looped thru. When theirs no redistribution from keys searchBranch takes over and searches thru each branch. The code for lookup and searchBranch are all below. It clearly seen how it calls for searchBranch if it's gone thru the key inputs and if not it calls recursively for the three next letters. It also shows how searchBranch recursively calls for all non null sub branches. and adds results if the nodes are marked valid.

```
private ArrayList<String> lookup(Node curr, int[] word, int wordIndx, String currWo
    if (wordIndx >= word.length) {
        searchBranch(curr, currWord, results);
        return results;
    }
    int index = ((word[wordIndx]-1)*3);
    String[] letters = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
    if (curr.next[index] != null)
        lookup(curr.next[index], word, (wordIndx + 1), currWord + letters[index], re
    if (curr.next[index + 1] != null)
        lookup(curr.next[index + 1], word, (wordIndx + 1), currWord + letters[index
    if (curr.next[index + 2] != null)
        lookup(curr.next[index + 2], word, (wordIndx + 1), currWord + letters[index
    return results;
}

private void searchBranch(Node curr, String currWord, ArrayList<String> results) {
    if (curr.valid) {
        results.add(currWord);
    }
    String[] letters = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
    for (int i = 0; i < curr.next.length; i++) {
        Node next = curr.next[i];
        if (next != null) {
            searchBranch(next, currWord + letters[i], results);
        }
    }
}
```

Result

The result is a function that allows a inputs from solely a keypad and matches it with all words in a list. It can also be easily expandable with a larger list and the down side is a bit longer start up time since the start up time is proportional to O(n).

Conclusion

T9 is a good way of showing the use cases of a well structured large tree and how it doesn't have to be complex to work with such a large tree as long as it has some structure to it. It also shows how data can be combined without making it difficult to work with like how the tree of letters can use the same nodes for different words and solve that by having a valid marker.