T9

in C

Algorithms and data structures ID1021

Johan Montelius

Fall 2024

Introduction

In this assignment you should implement the something that you might have used when your were a kid and had your first mobile phone. At the time when you had to text messages using a regular nine digit keypad, the T9 system made life easier. If you typed ”43556” the text message would probably say ”hello” and not ”idklo” or something else that was not a word.

T9 kept track of all words that could possibly be encoded starting in a sequence of keys. If the user has typed ’3’ followed by ’2’ it knew that this narrows the set of words down to words starting in ”ge”, ”he”, ”id” and ”if” (assuming that there are no words starting in ”gd” nor ”gf” etc). T9 would use this information to display choices for the user or, if there was only one possible word, replace the sequence with a word. Sometimes the smartness of T9 gave surprising text messages but over all it worked great.

The true implementation of T9 was very clever in how it encoded the table of words in the dictionary and this was necessary since memory was in short supply. The implementation that you should do will not be as memory efficient but you will use the same structure as the original T9 implementation.

A strange tree - a trie

In this assignment you will not follow the T9 implementation exactly. To start with you will use a list of the most common Swedish words and we will then use only the characters ’a’ to ’ö’ but not ’q’ nor ’w’. This means that we will have 27 characters which suits us fine since this is three characters for each of the keys ’1’ through ’9’.

The main class of this assignment could be called `T9`. As we have seen many examples of before it will be the public interface but hide all the implementation details.

a node

To represent all the words in the list we will use a tree structure called a *trie*. The idea is to construct a tree where each node has as many branches as there are letters in the alphabet. The root of the tree thus has 27 branches representing the words that start with: 'a', 'b', 'c' etc.

A node will apart from the set of branches also have a boolean value indicating of the path from the root to the node is a valid word. We will call this *a valid node* although all nodes have a purpose, these are the ones that represents valid words.

In C we can define the node to hold the array itself rather that a reference to an array.

```
typedef struct node {
    bool valid;
    node *next[27];
  } node;

typedef struct trie {
  node *root;
  } trie;
```

In C we will have to do some extra thinking. A slight problem that we have is that the dictionary that we will use is in Swedish and thus represented, not in plain ASCII but in UTF-8. This means that some characters ('å', 'ä' and 'ö') are represented by two bytes each. Since regular strings in C are arrays of bytes this causes some confusion.

In UTF, each character is given a number, its *code point*. The Unicode code points for 'å', 'ä' and 'ä' are given below. Since the number of characters in the world far exceeds 256 (that would fit in a byte) the codepoints are *encoded* in aclever fornat where some characters are coded using two, three or even four bytes. The UTF-8 codes for the Swedish caracters alluse two bytes as shown in the table.

character	Unicode	UTF-8
å	229	0xc3a5
ä	228	0xc3a4
ä	246	0xc3b6

If we include the library `wchar.h` and `locale.h` we can set the "locale" to UTF-8 and then read UTF-8 encoded characters from a file. In the code below the array `ws` is an array of `wchar_t` i.e. `a wide character string`. Each caracter is now represented by its Unicode code (an `int`).

The following code should get you started (note how we use the ”%S” directive in the `printf` statement since the string is now encoced with ”wide” characters.

```
trie *dict() {

  setlocale(LC_ALL,"en_US.UTF-8");

  // Open the dictionary in read mode
  FILE *fptr = fopen("kelly.txt", "r");

  if (fptr == NULL)
    return NULL;

  trie *kelly = (trie*)malloc(sizeof(trie));
  kelly->root = NULL;

  wchar_t ws[BUFFER];

  while (fgetws(ws, BUFFER, fptr) != NULL) {
    printf("adding %S", ws);
    kelly->root = add(kelly->root, ws);
  }

  fclose(fptr);
  return kelly;
}
```

A leaf in the tree will have all intries in the `next` array set to `NULL`. Note that the words themselves are never explicitly represented as strings, it’s the path to a valid node (`valid` set to `true`) that implicitly represents a word.

You might ask yourself why we use this strange tree to represent the words but it turns out that it is a very compact form if done right. In this Java implementation the set of 27 branches will take up some space but we could have coded them as 27 bits i.e. four bytes. Searching for possible words is also quite efficient even if we could find more efficient representations.

code, index and key

Some terminology and methods that could come in handy are: *character*, *code*, *key* and *index*. A code is the integer representation of our characters. Implement a method that given character (’a’. . . ’ö’) returns the code: 0. . . 26. We will use these codes since they then can be used to address the branches of an array i.e. the `next` array of a node. Also implement the reverse method that given a code returns the character.

In C you now need to encode 'å', 'ä', and 'ö' explicitly using ther Unicode code points. If you simply write 'ö' the compiler will warn you that you're using a two byte character.

When implementing these methods, `switch` statement might come in handy:

```
static int code(wchar_t w) {
  switch (w) {
  case 'a' :
    return 0;
  case 'b' :
    return 1;
   :
  case 229 :
    return 24;
  case 228 :
    return 25;
  case 246 :
    return 26;
  }
  printf("strange character in word: %d\n", w);
  return -1;
}
```

The second thing we will need is a procedure that given *a key* returns *an index*. The keys are the keys that you press: '1', '2' etc and the indices are the integers 0 . . . 8. We will use indices starting with 0 since we will use them to index an array.

The last thing we need, and this is not strictly needed but could be fun to have, is a method that returns *a key* given a character. This could be used to encode words so that you can turn "toffel" into "752224". It will come in handy when you do tests where you first insert a word and then make sure that you can actually find the word given the encoded sequence.

So, to recap the terminology:

- **character:** the characters in our alphabet `a...ö`.
- **code:** the character code in the range `0...27`.
- **key:** a character, the key that you press i.e. `'1'` , `'2'` etc.
- **index:** an int, the representation of keys in the range `0...8`.

adding words

You will populate the tree by adding all words in a list. A word is added by starting in the root and then work your way down the tree given the indices of the characters in the word. If you find a branch empty you will of course have to construct that branch. When you reach the last character you make sure that the last node is marked as a valid node.

The add method is surprisingly simple and if you only draw up what exactly is to be done you will write it down in ten lines of code.

```
node *add(node *nd, wchar_t *rest) {
  if (nd == NULL) {
    // construct a new node
  }
  int c = code((int)*rest);

  if (rest[1] == '\n') {
    // if the next character is eol, then this a valid node
     :
  } else {
    nd->next[...] = add(... , (... + 1));
  }
}
```

searching for words given a sequence

The lookup procedure is slightly more tricky but only because we are looking for all possible words and not just a single word. The following outline is one idea but you could implement it as you like.

Implement a procedure `decode`, that takes a key sequence (`"2314"`) and returns a list of all possible words that could match the sequence. You can implement this by creating an empty list and then add possible words as we find them.

Starting in the root node you should `collect` all possible alternatives given the key sequence. If the first key is '2' then this corresponds to the initial letters 'd', 'e' or 'f'. You will find these branches if you look at branches: 3, 4 and 5. Take the key '2', convert it to the index 1 and then examine branches $1*3$, $1*3+1$ and $1*3+2$. You can examine the branches by using the `collect` method recursively.

As an argument to `collect` you also provide a string and this is the string representing the path that you have taken. Initially this string is set to `""` but as you go down the trie you add characters at the end of the string. If you go down the second branch (`1*3+1`) then you add the character that corresponds to the index 4 i.e. `'e'` to the string.

When you reach the end of the sequence you then have a string that could be a valid word. Looking at the node that you have, the boolean `valid` will tell you if this is indeed a valid word. If it is you simply add it to the list of valid words. If all works well you should in the end have a list containing all possible words.

Vanligaste orden i svenska

The word "svenska" is among the two thousand most common words in Swedish text. The word "i" is in the top hundred. The words "vanligaste" and "orden" are not among the most common but the words "vanlig" and "ord" are both in the top 500.

Given to you is a file containing the eight thousand most common words. This will be your source to populate the T9 tree. The file is a version of svenska "Kelly-listan" [1] where we have removed multiple word expressions, any words containing 'w' (only "webb-", "show" and "clown") or 'q' (only "squash"), and changed words like "idé" to "ide".

To make sure that your T9 implementation work you can first populate the tree, then for each word in the list encode it as a sequence of keys and finally do a decoding of this sequence. You will of course have cases where the encoded version of a word is decoded resulting in two or three words but most words are decoded back to the original.

References

[1] Kilgarriff, Adam; Charalabopoulou, Frieda; Gavrilidou, Maria; Johannessen, Janne Bondi; Khalil, Saussan; Kokkinakis, Sofie Johansson; Lew, Robert; Sharoff, Serge; Vadlapudi, Ravikiran & Volodina, Elena Corpus-based vocabulary lists for language learners for nine languages Language Resources and Evaluation,48:121–163 DOI 10.1007/s10579-013-9251-2 2014