Predictive text using a trie data structure in C

Sam Serbouti serbouti@kth.se

October 19, 2024

This assignement is about making a simplified version of the T9 predictive text system in C using a trie data structure (which is good for storing dynamic sets of strings). One specificity is handling special Swedish characters (å, ä, ö) by using UTF-8 encoding.

1 What I need and how it needs to work

I want to input a sequence of keypresses (for example "43556" for "hello") and get all the possible words that match the sequence. Each keypress corresponds to a set of letters[1] (4 corresponds to g, h, and i). I must find all words that correspond to that sequence by traversing a data structure built from a list of swedish words:the kelly listan.

I will use a **trie** to hold the words. It's like a tree structure where each node is a character and the path from the root to a leaf node makes a word. It must have an **add function** that adds words to the trie and a **decode function** that takes a sequence of keypresses and returns all possible words from the trie that match the sequence.

2 Trie data strcture

Each trie node holds `bool valid`[2] a flag that indicates whether the node marks the end of a valid word and `next[27]` an array of pointers to the next nodes in the trie [3] (ie all the letters of our alphabet that will follow the current one).

The trie structure holds a `node *root` and words are added or searched by going through pointers in the trie. Consider the words hej and hell. They look like:

[1] our letters are all from a to ö except q and w

[2] I will use `#include <stdbool.h>`

[3] 27 because the 26 letters of the alphabet minus q and minus w and plus å, ä and ö

```
        root              root->[h]->[e]->[j(valid)]
       / |  \             root->[h]->[e]->[l]->[l(valid)]
    'a''b' ... 'h' ...
                |
               'e'
              /  |
           'l'  'j' (valid)
          /
       'l' (valid)
```

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

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

```
node *new_node() {
    node *nd =(node *)malloc(sizeof(node));
    nd->valid =false;
    for (int i =0; i<27;i++){
        nd->next[i]=NULL;
    }
    return nd;
}
```

The first thing I need to do in main is put things in our trie. I take a list of 8 thousand Swedish words used in every day life. But to add them in the trie, I need an `add` function.

3 Adding words to the trie

If I take the word "katt", I need to:

- split it into 'k', 'a', 't', 't'
- assign an index to each character (0 for 'a', 10 for 'k'...)
- create the node k, create the node for a after k etc
- flag 't' as `valid=true`

For other words like 'kar', I will reuse the k-a connection.

Encoding characters

`static int code(wchar_t w)` takes a character like 'a' and returns the index '0', 'z' to '23' and the special cases 'å' to '229', 'ä' to 228 and 'ö' to 246 using a switch-case. It returns -1 for a wrong character.

Recursive add

```
node *add(node *nd, wchar_t *rest) {
    if(nd==NULL){nd = new_node();} //build new node

    int c=code((int)*rest);
```

```
    if(c==-1) return nd; //wrong char

    if (rest[1] == '\n') {nd->valid = true;}
    else {nd->next[c]=add(nd->next[c],rest+1);}
    return nd;
}
```

Great, now that the trie is built, I want to get all the possible words from a keypress sequence

4 Decoding

`decode` takes a key seqeunce and returns a list of possible words so it needs to go through the trie to explore all possible paths based on the key sequence: `collect` will do this; and meanwhile, add the correct words (when node is flagged as valid) to the dynamic list in . For example, give `decode(kelly->root, L"5278"` (kelly being our dataset), `collect` starts at the root, then should go to 'k' ie index 10 because of the key '5'. The function `key_to_indices` will map this. (see section 5). `collect` recursively explores all paths in the trie that match the key sequence. For each key in the sequence, the function determines the range of letters that the key could represent and explores each of the corresponding branches in the trie. If a valid word is found, it is added to the result list.

Collect

`*current` is node where we currently are in the trie. `*keys` is the key sequence. `depth` is how deep we are in the trie. `*word` is the array where we build the current word. `word_len` is it's length so far.`**results` stores the words we find. `*result_count` counts the number of found words. I use the string "abcdefghijklmnoprstuvxyzääö" to link an index to the correcsponding letter of a word.

```
void collect(node *current, wchar_t *keys, int depth, char *word, int word_len,
             ..char **results, int *result_count)
{
    if (keys[depth] == '\0') { //reach end of key seaquence
        if (current->valid) { //valid word
            word[word_len] = '\0';//end word with null character
            results[*result_count]=strdup(word);//copy word into our results
            (*result_count)++;//increment result count
        }
        return;
    }
    int start, end;
    key_to_indices(keys[depth], &start, &end); //map current key to its letter
    for (int i=start; i<=end; i++){ //iterate over possible letters
        if (current->next[i]){
```

```
            word[word_len]="abcdefghijklmnoprstuvxyzääö"[i];
            collect(current->next[i], keys, depth+1, word, word_len+1,
                ..results, result_count);
        }
    }
}
```

Decode

```
char **decode(trie *t9, wchar_t *keys){
    char word[BUFFER];
    char **results =malloc(sizeof(char*)*100);
    int result_count =0;
    collect(t9->root, keys, 0, word, 0, results, &result_count);
    results[result_count]=NULL;
    return results;
}
```

5 Mapping

`key_to_indices` maps each keypress to a range of indices in the trie.

With t9, each keypress corresponds to a range of letters:

Figure 1: from touchedeclavier.com

```
void key_to_indices(int key, int *start, int *end) {
    switch (key) {
    case '2': *start = 0; *end = 2; break;  // a b c
    case '3': *start = 3; *end = 5; break;  // d e f
    case '4': *start = 6; *end = 8; break;  // g h i
    ...
    case '9': *start = 21; *end = 26; break;// x y z å ä ö
    default: *start = -1; *end = -1; break; // wrong
    }
}
```

Conclusion

Unfortunately, I couldn't get to the end of it. Working with strings was a real challenge for this assignment, especially for the decoding part of the project. I think my errors come from wrong mapping between index and letters. But other than that, it was fun work (in particular having to assemble all these moving pieces).