T9 in C

Ying Pei Lin

Fall 2024

The encoding of the characters

After reading the input file, we need to change the characters from UTF-8 to unicode. However, the three swedish characters, å, ä, and ö, are originally encoded in UTF-8 as two bytes, turn out to be two separate UTF-8 characters in the `kelly.txt` file. For example, the character å is encoded as two bytes, 0xc3a5, in UTF-8, but in the `kelly.txt` file, it is encoded as Ã¥, which are 0xc3 and 0xa5 in UTF-8. So we need to combine them manually to get the correct UTF-8 encoding, then we can convert them to unicode. Down below is the code snippet to combine the two bytes to get the correct UTF-8 encoding.

```
void combine_unicode(wchar_t *input, wchar_t *output) {
  while (*input) {
    // Check if current character is 0xC3 (Ã) and next is available
    if (*input == 0x00C3 && *(input + 1)) {
      // Get the next character
      wchar_t next = *(input + 1);

      // Calculate the actual character e.g. For Ã¥ (0xC3 0xA5) -> å (0xE5)
      wchar_t combined = 0;
      if (next == 0x00A5) {  // å
        combined = 0x00E5;
      } else if (next == 0x00A4) {  // ä
        combined = 0x00E4;
      } else if (next == 0x00B6) {  // ö
        combined = 0x00F6;
      } else if (next == 0x0085) {  // Å
        combined = 0x00C5;
      } else if (next == 0x0084) {  // Ä
        combined = 0x00C4;
      } else if (next == 0x0096) {  // Ö
        combined = 0x00D6;
      }
```

```
      if (combined) {
        *output++ = combined;
        input += 2;  // Skip both characters
        continue;
      }
    }

    // Copy character
    *output++ = *input++;
  }
  *output = L'\0';
}
```

The trie data structure

Trie is a tree structure used for storing the letters in a way that allows for fast retrieval of the words. It contains a root node, and each node has a boolean value to indicate if the node is the end of a word, and an array of pointers to the next nodes. The trie data structure is defined as follows:

```
typedef struct node {
  bool valid;
  struct node *next[30];
} node;

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

Inserting a word into the trie

To insert a word into the trie, I first extract the word from the input file using the provided `dict()` function. Then, inside the loop(shown below), I remove the newline character, convert it to wide characters, and combine the swedish characters.

```
while (fgets(buf, sizeof(buf), fptr) != NULL) {
  // Remove newline character
  buf[strcspn(buf, "\n")] = 0;

  // Convert to wide character
  mbstowcs(wbuf, buf, sizeof(buf));
  combine_unicode(wbuf, result);
```

```
  // Add word to trie
  kelly->root = add(kelly->root, result);
}
```

Then we use `*add(node *nd, wchar_t *rest)` to insert the word into the trie by traversing the trie and adding the characters to the trie. If the node is null, we create a new one and initialize the next thirty nodes to null. Then we check if the character is the last character of the word. If not, we continue to add the next character to the trie. . If the character is the last character of the word, we set the node to be valid. Below is the code snippet for adding a word to the trie.

```
node *add(node *nd, wchar_t *rest) {
  if (nd == NULL) {
    nd = (node*)malloc(sizeof(node));
    nd->valid = false;
    for (int i = 0; i < 30; i++) {
      nd->next[i] = NULL;
    }
  }

  if (*rest == L'\0') {
    nd->valid = true;
    return nd;
  }

  int c = code((int)*rest);
  if (c == -1) {
    // printf("Character: %lc (0x%x)\n", *rest, (unsigned int)*rest);
    return nd;
  }
  nd->next[c] = add(nd->next[c], rest + 1);
  return nd;
}
```

The `code()` function is used to convert the wide character to an integer value.

```
int code(wchar_t w) {
  if (w >= L'a' && w <= L'z') return w - L'a';
  if (w >= L'A' && w <= L'Z') return w - L'A';

  switch (w) {
    case L'å': case L'Å': return 26;
    case L'ä': case L'Ä': return 27;
```

```
    case L'ö': case L'Ö': return 28;
    default:
      return -1;
  }
}
```

Search for words

To find a word in the trie, we need check the all possible in the range of the characters in the trie. Each integer may represent three to four characters. To get the range of characters for a key, we use the `key_start[]` and `key_length[]` arrays. The `key_start[]` array contains the starting index of the characters for each key, and the `key_length[]` array contains the number of characters for each key. For example, the key 1 represents the characters 'abc', so the starting index is 0. By the help of these two arrays, we narrow down the search range for each key. The arrays are defined as follows:

```
// Mapping of T9 keys to character ranges
static const int key_start[] = {
  0,    // 1: abc (0,1,2)
  3,    // 2: def (3,4,5)
  6,    // 3: ghi (6,7,8)
  9,    // 4: jkl (9,10,11)
  12,   // 5: mno (12,13,14)
  15,   // 6: pqrs (15,16,17,18)
  19,   // 7: tuv (19,20,21)
  22,   // 8: wxyz (22,23,24,25)
  26    // 9: åäö (26,27,28)
};

static const int key_length[] = {
  3,  // 1: abc
  3,  // 2: def
  3,  // 3: ghi
  3,  // 4: jkl
  3,  // 5: mno
  4,  // 6: pqrs
  3,  // 7: tuv
  4,  // 8: wxyz
  3   // 9: åäö
};
```

We then use the `collect()` function to search for the word in the trie. If the character is the last character of the word and the node is valid, we

add the word to the result list. Otherwise, we continue to search for the next character in the trie. The code snippet is shown below.

```
void collect(node* current, const char* keys, int key_pos,
                  wchar_t* prefix, int prefix_pos, word_list* results) {
  if (current == NULL) return;

  // End of key, check if current node is valid
  if (keys[key_pos] == '\0') {
    if (current->valid) {
      prefix[prefix_pos] = L'\0';
      add_word(results, prefix);
    }
    return;
  }

  // Convert key to index (2->1, 3->2, ....)
  int key_idx = keys[key_pos] - '1';
  if (key_idx < 0 || key_idx > 8) return;  // Invalid key

  // Get the range of characters for this key
  int start = key_start[key_idx];
  int len = key_length[key_idx];

  // Try each possible character for this key
  for (int i = 0; i < len; i++) {
    int char_idx = start + i;
    if (current->next[char_idx] != NULL) {
      // Add character to prefix
      prefix[prefix_pos] = L'a' + char_idx;
      // Special handling for Swedish characters
      if (char_idx == 26) prefix[prefix_pos] = L'å';
      else if (char_idx == 27) prefix[prefix_pos] = L'ä';
      else if (char_idx == 28) prefix[prefix_pos] = L'ö';

      // Recurse with next key
      collect(current->next[char_idx], keys, key_pos + 1,
              prefix, prefix_pos + 1, results);
    }
  }
}
```