Spelling correction algorithm: using hash table, separate chaining with linked list, and move-to-front technique

Program Overview

By Leonardo Linardi (linkedin.com/in/leonardolinardi)

Section 1 - File Overview

This program consists of the following code and data files:

Filename	Description
`main.c`	Entry point to program.
`list.h, list.c`	Singly linked list module.
`spell.h, spell.c`	The 4 main functions of the spelling correction algorithm.
`strhash.h, strhash.c`	Module providing several hash functions for strings
`hashtbl.h, hashtbl.c`	Hash table with separate chaining and move-to-front technique
`Makefile`	To assist with compilation.
data/	Directory containing some lists of words considered 'correctly spelled'.

Section 2 - Data structures

Throughout this program, 'string' refers to a null-terminated array of `char` suitable for use with the functions declared in `string.h` (standard library string manipulation functions). Furthermore, every such string will only consist of lower-case alphabetic characters (and the null byte at the end).

The functions in `spell.h` work well with the data structures defined in `list.h`.
Here's a brief overview: A List is a linked list of `Node`s, with each `Node` containing a pointer to a single piece of data and to the next `Node` in the linked list. These lists are easily traversed using an appropriate `while` or `for` loop.

The data with each list node is usually a string. That is, the node's `data` field (of type `void *`) can be cast to type `char *` and can be passed to the functions from `string.h`.

Section 3 - The Program

I. Computing edit distance

Function Name : `print_edit_distance()`

Parameters :

1. `word1` (string)
2. `word2` (string)

This function computes the Levenshtein edit distance between the strings `word1` and `word2`. That is, this program computes the minimum number of 'edits' required to transform `word1` into `word2`, where edits are defined as letter substitutions, deletions or insertions.

This function will print the result (a single integer) on a single line to the standard output. It will not print anything else to the standard output.

II. Enumerating all possible edits

Function Name : `print_all_edits()`

Parameters :

1. `word` (string)

This function will generate and print all lower-case alphabetic strings within a Levenshtein edit distance of 1 from `word`. That is, all the strings made of alphabetic characters ('abcdefghijklmnopqrstuvwxyz') that can be made from substituting a single letter in word, inserting a single letter into word, or deleting a single letter from word.

This function will print the results (lots of strings) to the standard output, one string per line. It will not print anything else to the standard output. The function might print some words multiple times and it may also print the string word itself.

III. Spell Checking

Function Name : `print_checked()`

Parameters :

1. `dictionary` (a `List` pointer), a list of strings representing correctly spelled words in approximate order of decreasing probability of occurrence
2. `document` (another `List` pointer), a list of strings to check for spelling mistakes

For each string (word) in `document`, this function will check if the string occurs in `dictionary` (that is, if it's a correctly spelled word).

If the string does occur in `dictionary`, the function should print the string directly to the standard output on its own line. If the string does not occur in `dictionary`, then the function should print the string to the standard output followed by a question mark, on its own line. Strings will be printed in their order of occurrence in `document`.

IV. Spelling Correction

Function Name : `print_corrected()`

Parameters :

1. `dictionary` (a `List` pointer), a list of strings representing correctly spelled words in approximate order of decreasing probability of occurrence
2. `document` (another `List` pointer), a list of strings to check for spelling mistakes (and attempt to correct)

For each string (word) in document, in order, this function will do the following:

1. If the string occurs in `dictionary` (that is, if it's a correctly spelled word) the function should print this word directly to the standard output on its own line.

2. If the string does not occur in `dictionary` the function will try to find a `corrected' version of the string with a Levenshtein edit distance of 1 in `dictionary`.
    - If such a word exists, the function will print this word to the standard output on its own line.
    - If multiple such words exist, the function will print only the word that occurs first among them in `dictionary`.
3. If the string does not occur in `dictionary` and no words in `dictionary` have a Levenshtein edit distance of 1 with the string, then the program will try to find a `corrected' version of the string with a Levenshtein edit distance of 2 in `dictionary`.
    - If such a word exists, the function will print this word to the standard output on its own line.
    - If multiple such words exist, the function will print only the word that occurs first among them in `dictionary`.
4. If none of the first three conditions are met, then the function will attempt to find a `corrected` version of the string with a Levenshtein edit distance of 3 in `dictionary`.
    - If such a word exists, the function will print this word to the standard output on its own line.
    - If multiple such words exist, the function will print only the word that occurs first among them in `dictionary`.
5. If none of the previous conditions are met, then the function will print the original string to the standard output followed by a question mark, on its own line.

Section 4 - Running the program

I. Computing edit distance

To execute the function, run a command of the following form:

```
./a2 dist word1 word2
```

where `word1` and `word2` are the two alphabetic strings to be used as the two arguments to the function (`word1` and `word2`, respectively). For example, the command `./a2 dist sweet sleep` will run the function with the strings ‘`sweet`’ as `word1` and "`sleep`" as `word2`.

II. Enumerating all possible edits

To execute the function, run a command of the following form:

```
./a2 edits word
```

where `word` is an alphabetic string to be passed as the argument to the function. For example, the command `./a2 edits soup` will run the function with the string ‘`soup`‘as the argument `word`.

III. Spell Checking

To execute the function, run a command of the following form:

```
./a2 check dictionary_filename < document_filename
```

where `dictionary_filename` is the path of a text file containing ‘correctly-spelled’ lower-case alphabetic words, one per line; and `document_filename` is the path of a text file containing words which may or may not be ‘correctly-spelled’, one per line. The strings in these files will be read into the linked lists arguments passed into your function (`dictionary` and `document`, respectively).

For example, the command `./a2 check data/words.txt < document.txt` will run the function with the list of “correctly-spelled” words coming from the file `words.txt` inside the local directory `data`, and input words coming from the local file `document.txt`.

Alternatively, run a command of the following form:

```
./a2 check dictionary filename
```

and type a sequence of lower-case alphabetic words (one per line) directly into your program's standard input. Once you have finished entering words to check, enter a blank line or an end of file marker (either input will cause the program to stop reading from the standard input and start running your function).

IV. Spelling Correction

To execute the function, run a command of the following form:

```
./a2 spell dictionary_filename < document_filename
```

```
./a2 spell dictionary filename
```

with similar meanings as the commands for running the spell-checking function above.