Table of Contents

Speller

Be sure to read this specification in its entirety before starting so you know what to do and how to do it!

tl;dr

Implement a program that spell-checks a file, a la the below, using a hash table.

```
$ ./speller texts/lalaland.txt
MISSPELLED WORDS

[...]
AHHHHHHHHHHHHHHHHHHHHHHHHHHHT
[...]
Shangri
[...]
fianc
[...]
Sebastian's
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN TEXT:
TIME IN load:
TIME IN check:
TIME IN size:
TIME IN unload:
TIME IN TOTAL:
```

Distribution

Downloading

Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

1. Execute `cd` to ensure that you're in `~/` (aka your home folder).

2. Execute `mkdir pset4` to make (i.e., create) a directory called `pset4` in your home directory.

3. Execute `cd pset4` to change into (i.e., open) that directory.

4. Execute `wget https://cdn.cs50.net/2018/fall/psets/4/speller/hashtable/speller.zip (https://cdn.cs50.net/2018/fall/psets/4/speller/hashtable/speller.zip)` to download a (compressed) ZIP file with this problem's distribution.

5. Execute `unzip speller.zip` to uncompress that file.

6. Execute `rm speller.zip` followed by `yes` or `y` to delete that ZIP file.

7. Execute `ls`. You should see a directory called `speller`, which was inside of that ZIP file.

8. Execute `cd speller` to change into that directory.

9. Execute `ls`. You should see this problem's distribution:

```
dictionaries/  dictionary.c  dictionary.h  keys/  Mak
```

Understanding

Theoretically, on input of size *n*, an algorithm with a running time of *n* is "asymptotically equivalent," in terms of *O*, to an algorithm with a running time of 2*n*. Indeed, when describing the running time of an algorithm, we typically focus on the dominant (i.e., most impactful) term (i.e., *n* in this case, since *n* could be much larger than 2). In the real world, though, the fact of the matter is that 2*n* feels twice as slow as *n*.

The challenge ahead of you is to implement the fastest spell checker you can! By "fastest," though, we're talking actual "wall-clock," not asymptotic, time.

In `speller.c`, we've put together a program that's designed to spell-check a file after loading a dictionary of words from disk into memory. That dictionary, meanwhile, is implemented in a file called `dictionary.c`. (It could just be implemented in `speller.c`, but as programs get more complex, it's often convenient to break them into multiple files.) The prototypes for the functions therein, meanwhile, are defined not in `dictionary.c` itself but in `dictionary.h` instead. That way, both `speller.c` and `dictionary.c` can `#include` the file. Unfortunately, we didn't quite get around to implementing the loading part. Or the checking part. Both (and a bit more) we leave to you! But first, a tour.

`dictionary.h`

Open up `dictionary.h`, and you'll see some new syntax, including a few lines that mention `DICTIONARY_H`. No need to worry about those, but, if curious, those lines just ensure that, even though `dictionary.c` and `speller.c` (which you'll see in a moment) `#include` this file, `clang` will only compile it once.

Next notice how we `#include` a file called `stdbool.h`. That's the file in which `bool` itself is defined. You've not needed it before, since the CS50 Library used to `#include` that for you.

Also notice our use of `#define`, a "preprocessor directive" that defines a "constant" called `LENGTH` that has a value of `45`. It's a constant in the sense that you can't (accidentally) change it in your own code. In fact, `clang` will replace any mentions of `LENGTH` in your own code with, literally, `45`. In other words, it's not a variable, just a find-and-replace trick.

Finally, notice the prototypes for four functions: `check`, `load`, `size`, and `unload`. Notice how two of those take a pointer as an argument, per the `*`:

```
bool check(const char *word);
bool load(const char *dictionary);
```

Recall that `char *` is what we used to call `string`. So those two prototypes are essentially just:

```
bool check(const string word);
bool load(const string dictionary);
```

And `const`, meanwhile, just says that those strings, when passed in as arguments, must remain constant; you won't be able to change them, accidentally or otherwise!

`dictionary.c`

Now open up `dictionary.c`. Notice how, atop the file, we've defined a `struct` called `node` that represents a node in a hash table. And we've declared a global array, `hashtable`, with enough room for `N` pointers to nodes, where `N` is initialized above that array to `26`, one bucket for each letter in the (English) alphabet.

A bit below those lines have we implemented a function called `hash` that hashes a word, returning `0` for any word that starts with an `a` (or `A`), `1` for any word that starts with a `b` (or `B`), `25` for any word that starts with a `z` (or `Z`), and so on for every letter in between.

Below that have we written part of a function called `load` that will soon (thanks to you!) load a dictionary of words into the hash table. We've written some code that initializes all of the buckets (i.e., pointers) in `hashtable` to `NULL`. And we've written some code that opens `dictionary`, which is the file name of a dictionary to load. And we've also written some code that iterates over that dictionary and reads the words therein, one at a time, into a buffer (i.e., `string`) called `word`. But we stop short of inserting those words into `hashtable`. Thereafter, we do close the file, though, and then return `true` to indicate (we hope!) success.

As for `check`, `size`, and `unload`, well, we've only just barely implemented those, enough for the file to compile.

`speller.c`

Okay, next open up `speller.c` and spend some time looking over the code and comments therein. You won't need to change anything in this file, and you don't need to understand its entirety, but do try to get a sense of its functionality nonetheless. Notice how, by way of a function called `getrusage`, we'll be "benchmarking" (i.e., timing the execution of) your implementations of `check`, `load`, `size`, and `unload`. Also notice how we go about passing `check`, word by word, the contents of some file to be spell-checked. Ultimately, we report each misspelling in that file along with a bunch of statistics.

Notice, incidentally, that we have defined the usage of `speller` to be

```
Usage: speller [dictionary] text
```

where `dictionary` is assumed to be a file containing a list of lowercase words, one per line, and `text` is a file to be spell-checked. As the brackets suggest, provision of `dictionary` is optional; if this argument is omitted, `speller` will use `dictionaries/large` by default. In other words, running

```
./speller text
```

will be equivalent to running

```
./speller dictionaries/large text
```

where `text` is the file you wish to spell-check. Suffice it to say, the former is easier to type! (Of course, `speller` will not be able to load any dictionaries until you implement `load` in `dictionary.c`! Until then, you'll see **Could not load**.)

Within the default dictionary, mind you, are 143,091 words, all of which must be loaded into memory! In fact, take a peek at that file to get a sense of its structure and size. Notice that every word in that file appears in lowercase (even, for simplicity, proper nouns and acronyms). From top to bottom, the file is sorted lexicographically, with only one word per line (each of which ends with `\n`). No word is longer than 45 characters, and no word appears more than once. During development, you may find it helpful to provide `speller` with a `dictionary` of your own that contains far fewer words, lest you struggle to debug an otherwise enormous structure in memory. In `dictionaries/small` is one such dictionary. To use it, execute

```
./speller dictionaries/small text
```

where `text` is the file you wish to spell-check. Don't move on until you're sure you understand how `speller` itself works!

Odds are, you didn't spend enough time looking over `speller.c`. Go back one square and walk yourself through it again!

`texts/`

So that you can test your implementation of `speller`, we've also provided you with a whole bunch of texts, among them the script from *La La Land*, the text of the Affordable Care Act, three million bytes from Tolstoy, some excerpts from *The Federalist Papers* and Shakespeare, the entirety of the King James V Bible and the Koran, and more. So that you know what to expect, open and skim each of those files, all of which are in a directory called `texts` within your `pset4` directory.

Now, as you should know from having read over `speller.c` carefully, the output of `speller`, if executed with, say,

```
./speller texts/lalaland.txt
```

will eventually resemble the below. For now, try executing the staff's solution (using the default dictionary) with the below.

```
~cs50/2019/x/pset4/speller dictionaries/large texts/l
```

Below's some of the output you'll see. For information's sake, we've excerpted some examples of "misspellings." And lest we spoil the fun, we've omitted our own statistics for now.

```
MISSPELLED WORDS

[...]
AHHHHHHHHHHHHHHHHHHHHHHHHHHHT
[...]
Shangri
[...]
fianc
[...]
Sebastian's
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN TEXT:
TIME IN load:
TIME IN check:
TIME IN size:
TIME IN unload:
TIME IN TOTAL:
```

`TIME IN load` represents the number of seconds that `speller` spends executing your implementation of `load`. `TIME IN check` represents the number of seconds that `speller` spends, in total, executing your implementation of `check`. `TIME IN size` represents the number of seconds that `speller` spends executing your implementation of `size`. `TIME IN unload` represents the number of seconds that `speller` spends executing your implementation of `unload`. `TIME IN TOTAL` is the sum of those four measurements.

Note that these times may vary somewhat across executions of `speller`, depending on what else CS50 IDE is doing, even if you don't change your code.

Incidentally, to be clear, by "misspelled" we simply mean that some word is not in the `dictionary` provided.

`Makefile`

And, lastly, recall that `make` automates compilation of your code so that you don't have to execute `clang` manually along with a whole bunch of switches. However, as your programs grow in size, `make` won't be able to infer from context anymore how to compile your code; you'll need to start telling `make` how to compile your program, particularly when they involve multiple source (i.e., `.c`) files, as in the case of this problem. And so we'll utilize a `Makefile`, a configuration file that tells `make` exactly what to do. Open up `Makefile`, and you should see four lines:

1. The first line tells `make` to execute the subsequent lines whenever you yourself execute `make speller` (or just `make`).
2. The second line tells `make` how to compile `speller.c` into machine code (i.e., `speller.o`).
3. The third line tells `make` how to compile `dictionary.c` into machine code (i.e., `dictionary.o`).
4. The fourth line tells `make` to link `speller.o` and `dictionary.o` in a file called `speller`.

Be sure to compile `speller` by executing `make speller` (or just `make`). Executing `make dictionary` won't work!

Specification

Alright, the challenge now before you is to implement, in order, `load`, `size`, `check`, and `unload` as efficiently as possible using a hash table in such a way that `TIME IN load`, `TIME IN check`, `TIME IN size`, and `TIME IN unload` are all minimized. To be sure, it's not obvious what it even means to be minimized, inasmuch as these benchmarks will certainly vary as you feed `speller` different values for `dictionary` and for `text`. But therein lies the challenge, if not the fun, of this problem. This problem is your chance to design. Although we invite you to minimize space, your ultimate enemy is time. But before you dive in, some specifications from us.

- You may not alter `speller.c` or `Makefile`.
- You may alter `dictionary.c` (and, in fact, must in order to complete the implementations of `load`, `size`, `check`, and `unload`), but you may not alter the declarations (i.e., prototypes) of `load`, `size`, `check`, or `unload`. You may, though, add new functions and (local or global) variables to `dictionary.c`.
- You may alter `dictionary.h`, but you may not alter the declarations of `load`, `size`, `check`, or `unload`.
- Your implementation of `check` must be case-insensitive. In other words, if `foo` is in dictionary, then `check` should return true given any capitalization thereof; none of `foo`, `foO`, `fOo`, `fOO`, `fOO`, `Foo`, `FoO`, `FOo`, and `FOO` should be considered misspelled.
- Capitalization aside, your implementation of `check` should only return `true` for words actually in `dictionary`. Beware hard-coding common words (e.g., `the`), lest we pass your implementation a `dictionary` without those same words. Moreover, the only possessives allowed are those actually in `dictionary`. In other words, even if `foo` is in `dictionary`, `check` should return `false` given `foo's` if `foo's` is not also in `dictionary`.

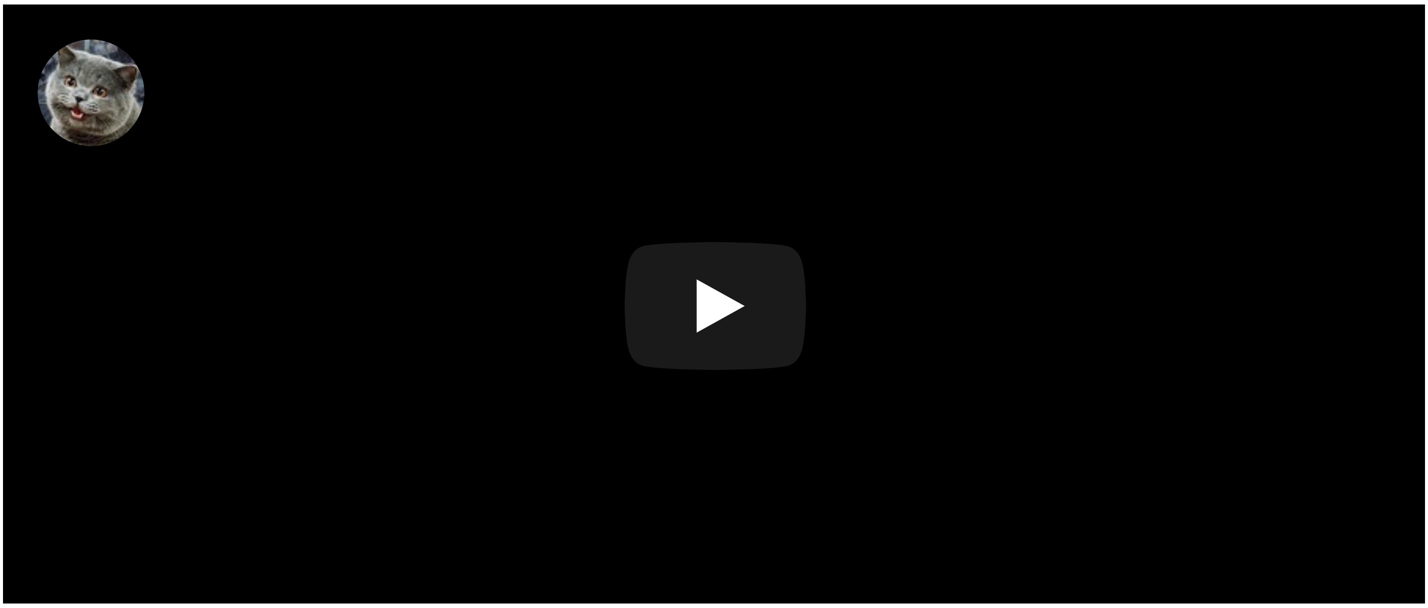
- You may assume that any `dictionary` passed to your program will be structured exactly like ours, alphabetically sorted from top to bottom with one word per line, each of which ends with `\n`. You may also assume that `dictionary` will contain at least one word, that no word will be longer than `LENGTH` (a constant defined in `dictionary.h`) characters, that no word will appear more than once, that each word will contain only lowercase alphabetical characters and possibly apostrophes, and that no word will start with an apostrophe.
- You may assume that `check` will only be passed words that contain (uppercase or lowercase) alphabetical characters and possibly apostrophes.
- Your spell checker may only take `text` and, optionally, `dictionary` as input. Although you might be inclined (particularly if among those more comfortable) to "pre-process" our default dictionary in order to derive an "ideal hash function" for it, you may not save the output of any such pre-processing to disk in order to load it back into memory on subsequent runs of your spell checker in order to gain an advantage.
- You may alter the value of `N` and the implementation of `hash`.
- Your spell checker must not leak any memory. Be sure to check for leaks with `valgrind`.

Alright, ready to go?

1. Implement `load`.
2. Implement `size`.
3. Implement `check`.
4. Implement `unload`.

Walkthroughs

In these walkthroughs, Zamyla discusses not only hash tables but tries as well.



Hints

Ultimately, be sure to `free` in `unload` any memory that you allocated in `load`! Recall that `valgrind` is your newest best friend. Know that `valgrind` watches for leaks while your program is actually running, so be sure to provide command-line arguments if you want `valgrind` to analyze `speller` while you use a particular `dictionary` and/or text, as in the below. Best to use a small text, though, else `valgrind` could take quite a while to run.

```
valgrind ./speller texts/cat.txt
```

If you run `valgrind` without specifying a `text` for `speller`, your implementations of `load` and `unload` won't actually get called (and thus analyzed).

If unsure how to interpret the output of `valgrind`, do just ask `help50` for help:

```
help50 valgrind ./speller texts/cat.txt
```

Testing

How to check whether your program is outting the right misspelled words? Well, you're welcome to consult the "answer keys" that are inside of the `keys` directory that's inside of your `speller` directory. For instance, inside of `keys/lalaland.txt` are all of the words that your program *should* think are misspelled.

You could therefore run your program on some text in one window, as with the below.

```
./speller texts/lalaland.txt
```

And you could then run the staff's solution on the same text in another window, as with the below.

```
~cs50/2019/x/pset4/speller dictionaries/large texts/l
```

And you could then compare the windows visually side by side. That could get tedious quickly, though. So you might instead want to "redirect" your program's output to a file, as with the below.

```
./speller texts/lalaland.txt > student.txt
~cs50/2019/x/pset4/speller dictionaries/large texts/l
```

You can then compare both files side by side in the same window with a program like `diff`, as with the below.

```
diff -y student.txt staff.txt
```

Alternatively, to save time, you could just compare your program's output (assuming you redirected it to, e.g., `student.txt`) against one of the answer keys without running the staff's solution, as with the below.

```
diff -y student.txt keys/lalaland.txt
```

If your program's output matches the staff's, `diff` will output two columns that should be identical except for, perhaps, the running times at the bottom. If the columns differ, though, you'll see a `>` or `|` where they differ. For instance, if you see

```
MISSPELLED WORDS

TECHNO
L

Prius

L
```

that means your program (whose output is on the left) does not think that `Thelonious` or `MIA` is misspelled, even though the staff's output (on the right) does, as is implied by the absence of, say, `Thelonious` in the lefthand column and the presence of `Thelonious` in the righthand column.

`check50`

To test your code less manually (though still not exhaustively), you may also execute the below.

```
check50 cs50/problems/2019/x/speller
```

Note that `check50` will also check for memory leaks, so be sure you've run `valgrind` as well.

Staff's Solution

How to assess just how fast (and correct) your code is? Well, as always, feel free to play with the staff's solution, as with the below, and compare its numbers against yours.

```
~cs50/2019/x/pset4/speller dictionaries/large texts/l
```

Big Board

And if you'd like to put your code to the test against classmates' code (just for fun), execute the command below to challenge the Big Board before or after you submit.

```
submit50 cs50/problems/2019/x/challenges/speller
```

Then visit the URL that `submit50` outputs (after a few minutes!) to see where you rank!

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2019/x/speller
```

You can then go to https://cs50.me/cs50x (https://cs50.me/cs50x) to view your current scores!