This is CS50x.

Problem Set 5: Mispellings

This is the Hacker Edition of Problem Set 5. It cannot be submitted for credit.

Objectives.

- Allow you to design and implement your own data structure.
- Optimize your code's (real-world) running time.

Recommended Reading.

- Sections 18 – 20, 27 – 30, 33, 36, and 37 of http://www.howstuffworks.com/c.htm.
- Chapter 17 of *Programming in C*.

diff pset5 hacker5.

- Hacker Edition challenges you with a different **Section of Questions**.

A Section of Questions.

- This section of questions comes with some distribution code that you'll need to download before getting started. Go ahead and execute

  ```
  cd ~/Dropbox
  ```

  in order to navigate to your `~/Dropbox` directory. Then execute

  ```
  wget http://cdn.cs50.net/2012/fall/sections/6/hacker6.zip
  ```

  in order to download a ZIP (i.e., compressed version) of this section's distro. If you then execute

  ```
  ls
  ```

  you should see that you now have a file called `hacker6.zip` in your `~/Dropbox`

directory. Unzip it by executing the below.

```
unzip hacker6.zip
```

If you again execute

```
ls
```

you should see that you now also have a `hacker6` directory. You're now welcome to delete the ZIP file with the below.

```
rm -f hacker6.zip
```

Now dive into that hacker6 directory by executing the below.

```
cd hacker6
```

Now execute

```
ls
```

and you should see that the directory contains the below.

```
queue.c  sll.c  stack.c
```

- A "stack" is one of the basic, fundamental data structures of computer science. We use stacks when we're modeling collections of elements that follow a "last-in, first-out" (LIFO) pattern of insertion and retrieval. Think about the piles of trays in the dining halls: when the dining staff put trays out before meals, they pile them from the bottom to the top, and then you take the top-most tray when you arrive. The last tray that the staff put on the pile is the first one taken off of the pile.

  But wait, isn't the "stack" a segment of memory? Yep, that's right too! It's no coincidence that the data structure and the memory segment share the same name, since the stack memory segment behaves just like the stack data structure, whereby functions' "stack frames" are the elements being stored in the stack memory segment. When a function is called, a new stack frame is placed on the "top" of the stack memory segment, and when that function returns, its stack frame is removed from the top of the segment.

  Unlike arrays, which allow you to access any element in the array whenever you like, with stacks you only ever access the element at the top of the stack. A stack's two primary operations are called `push` and `pop`: `push` places a new element on the top of the stack (like a dining hall's tray or a function's stack frame), and `pop` retrieves the topmost element from the stack, decrementing the stack's size in the process.

  Your task here is to implement `push` and `pop` for a `stack` that stores `char*`s.

Per `stack.c`, we've defined a `stack` as

```
typedef struct
{
    char** strings;
    int size;
    int capacity;
}
stack;
```

where `strings` is a pointer to a dynamically-allocated array of `char*`s (that you'll need to expand appropriately as more and more `char*`s are pushed on to the stack); `size` is the number of elements currently in the stack (that you'll need to adjust appropriately so that you can track the location of the "top" of the stack); and `capacity` is the allocated capacity of the `strings` array (which you'll need to adjust, along with the `strings` array itself, as the stack fills up).

Notice, now, that `push` is declared as

```
bool push(char* str);
```

whereby it should return `true` if it can successfully put `str` on the top of the stack and `false` otherwise.

On the other hand, `pop` is declared as

```
char* pop(void);
```

whereby `pop` should return the `char*` from the top of the stack if there is one and `NULL` otherwise.

Notice, too, that we've provided code that will test your stack's functionality so that you know when you're on the right track.

Alright, implement `push` and `pop`!

- A queue, another fundamental data structure, is used for modeling collections of elements that follow a "first-in, first-out" (FIFO) pattern of insertion and retrieval. Just as you'd expect with the line of fanboys at most any Apple Store, the first one in line is the first one to get in and get out with a new iPhone.

  Like stacks (and unlike arrays), queues typically don't allow access to elements in the middle. Moreover, whereas `push` and `pop` adjust a stack's top, a queue's `enqueue` function places a new element at a queue's "tail" end, while `dequeue` retrieves the element at a queue's "head" (i.e., front).

  Notice how, in `queue.c`, we've defined a queue for `char*`s:

```
typedef struct
{
    int head;
    char** strings;
    int size;
    int capacity;
}
queue;
```

Notice how a `queue`, like a `stack`, encapsulates `strings` and `size`. The `head` field is new, though. We could consider the element at `strings[0]` to be the head of the queue and the element at `strings[size - 1]` to be the tail, but this would require us to shift all of the elements from `strings[1]` to `strings[size - 1]` down by one position every time we call `dequeue`. That's time-wasting work, though, especially if we've got a long queue! Therefore, we'll store the index of the queue's head element and adjust it as we dequeue elements.

Your job is to implement `enqueue` and `dequeue`, whose prototypes are the below.

```
bool enqueue(char* s);
char* dequeue(void);
```

Note that `enqueue` should return `true` if `str` is successfully enqueued and `false` otherwise. Likewise, `dequeue` should return the `char*` at the queue's head if there is one and `NULL` if the queue is empty.

- One of the main downsides to storing data in an array is that inserting or deleting an element in the middle of an array requires shifting other elements to make (or fill in) a gap. In situations where insertion and deletion of elements is more critical than the retrieval of them, a linked list is a fantastic tool.

  Your task this time around is to implement some functions for the provided singly-linked list of `int`s in `sll.c`. Recall that a list is just a sequence of nodes, so there's no `list` structure; rather, there's just a `node` structure that we'll define as

  ```
  typedef struct node
  {
      int i;
      struct node* next;
  }
  node;
  ```

  where `i` is the integer to be stored in the `node` and `next` is a pointer (i.e., a link) to the next `node` in the list. By convention, the last `node` in a list has its `next` pointer set to `NULL`.

  Alright, it's now up to you to implement these functions in sll.c:

```
/**
 * Returns the length of the list.
 */
int length(void);

/**
 * Returns true if a node in the list contains the value i and false
 * otherwise.
 */
bool contains(int i);

/**
 * Puts a new node containing i at the front (head) of the list.
 */
void prepend(int i);

/**
 * Puts a new node containing i at the end (tail) of the list.
 */
void append(int i);

/**
 * Puts a new node containing i at the appropriate position in a list
 * sorted in ascending order.
 */
void insert_sorted(int i);
```

- You will probably find it helpful to craft a couple of helper functions for such tasks as building a new node and inserting a node immediately following another one!

Getting Started.

- Start up your appliance and, upon reaching John Harvard's desktop, open a terminal window (remember how?) and execute

  ```
  update50
  ```

  to ensure that your appliance is up-to-date!

- Like Problem Set 4, this problem set comes with some distribution code that you'll need to download before getting started. Go ahead and execute

  ```
  cd ~/Dropbox
  ```

  in order to navigate to your `~/Dropbox` directory. Then execute

  ```
  wget http://cdn.cs50.net/2012/fall/psets/5/pset5.zip
  ```

in order to download a ZIP (i.e., compressed version) of this problem set's distro. If you then execute

```
ls
```

you should see that you now have a file called pset5.zip in your ~/Dropbox directory. Unzip it by executing the below.

```
unzip pset5.zip
```

If you again execute

```
ls
```

you should see that you now also have a `pset5` directory. You're now welcome to delete the ZIP file with the below.

```
rm -f pset5.zip
```

Now dive into that `pset5` directory by executing the below.

```
cd pset5
```

Now execute

```
ls
```

and you should see that the directory contains the below.

```
dictionary.c  dictionary.h  Makefile  questions.txt  speller.c
```

Interesting! Let's get started.

- 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*. In the real world, though, the fact of the matter is that the latter feels twice as slow as the former.

  The challenge ahead of you is to implement the fastest spell-checker you can! By "fastest," though, we're talking actual, real-world, noticeable seconds—none of that asymptotic stuff this 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. 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!

Before we walk you through `speller.c`, go ahead and open up `dictionary.h` with `gedit`. Declared in that file are four functions; take note of what each should do. Now open up `dictionary.c`. Notice that we've implemented those four functions, but only barely, just enough for this code to compile. Your job for this problem set is to re-implement those functions as cleverly as possible so that this spell-checker works as advertised. And fast!

Let's get you started.

- 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 set. And so we'll utilize a `Makefile`, a configuration file that tells make exactly what to do. Open up `Makefile` with `gedit`, and let's take a tour of its lines.

  The line below defines a variable called `CC` that specifies that make should use `clang` for compiling.

  ```
  CC = clang
  ```

  The line below defines a variable called `CFLAGS` that specifies, in turn, that `clang` should use some flags, most of which should look familiar.

  ```
  CFLAGS = -ggdb –O0 -Qunused-arguments -std=c99 –Wall –Werror
  ```

  The line below defines a variable called `EXE`, the value of which will be our program's name.

  ```
  EXE = speller
  ```

  The line below defines a variable called `HDRS`, the value of which is a space-separated list of header files used by `speller`.

  ```
  HDRS = dictionary.h
  ```

  The line below defines a variable called `LIBS`, the value of which is should be a space-separated list of libraries, each of which should be prefixed with `-l`. (Recall our use of `-lcs50` earlier this term.) Odds are you won't need to enumerate any libraries for this problem set, but we've included the variable just in case.

  ```
  LIBS =
  ```

  The line below defines a variable called `SRCS`, the value of which is a space-separated list of C files that will collectively implement speller.

```
SRCS = speller.c dictionary.c
```

The line below defines a variable called `OBJS`, the value of which is identical to that of `SRCS`, except that each file's extension is not `.c` but `.o`.

```
OBJS = $(SRCS:.c=.o)
```

The lines below define a "target" using these variables that tells make how to compile speller.

```
$(EXE): $(OBJS) Makefile
    $(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)
```

The line below specifies that our `.o` files all "depend on" `dictionary.h` and `Makefile` so that changes to either induce recompilation of the former when you run `make`.

```
$(OBJS): $(HDRS) Makefile
```

Finally, the lines below define another target for cleaning up this problem set's directory.

```
clean:
    rm -f core $(EXE) *.o
```

Know that you're welcome to modify this `Makefile` as you see fit. In fact, you should if you create any `.c` or `.h` files of your own. But be sure not to change any tabs (i.e., `\t`) to spaces, since make expects the former to be present below each target. To be safe, uncheck **Use Spaces** under **Tab Width** at the bottom of `gedit`'s window before modifying `Makefile`.

The net effect of all these lines is that you can compile `speller` with a single command, even though it comprises quite a few files:

```
make speller
```

Even better, you can also just execute:

```
make
```

And if you ever want to delete speller plus any core or .o files, you can do so with a single command:

```
make clean
```

In general, though, anytime you want to compile your code for this problem set, it should suffice to run:

```
make
```

- Okay, next open up `speller.c` with `gedit` and spend some time looking over the code and comments therein. You won't need to change anything in this file, but you should understand it nonetheless. Notice how, by way of `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 `/home/cs50/pset5/dictionaries/large` by default. In other words, running

  ```
  ./speller text
  ```

  will be equivalent to running

  ```
  ./speller ~cs50/pset5/dictionaries/large text
  ```

  where `text` is the file you wish to spell-check. Suffice it to say, the former is easier to type!

  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, as with `gedit`. 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 `/home/cs50/pset5/dictionaries/small` is one such dictionary. To use it, execute

  ```
  ./speller ~cs50/pset5/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!

- Okay, technically that last problem induced an infinite loop. But we'll assume you broke out of it. Open up `questions.txt` with `gedit` and answer each of the following questions in one or more sentences.

    0. What is pneumonoultramicroscopicsilicovolcanoconiosis?
    1. According to its man page, what does `getrusage` do?
    2. Per that same man page, how many members are in a variable of type `struct rusage`?
    3. Why do you think we pass `before` and `after` by reference (instead of by value) to `calculate`, even though we're not changing their contents?
    4. Explain as precisely as possible, in a paragraph or more, how `main` goes about reading words from a file. In other words, convince us that you indeed understand how that function's `for` loop works.
    5. Why do you think we used `fgetc` to read each word's characters one at a time rather than use `fscanf` with a format string like `"%s"` to read whole words at a time? Put another way, what problems might arise by relying on `fscanf` alone?
    6. Why do you think we declared the parameters for `check` and `load` as `const`?

- 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 *Austin Powers: International Man of Mystery*, a sound bite from Ralph Wiggum, three million bytes from Tolstoy, some excerpts from Machiavelli and Shakespeare, the entirety of the King James V Bible, and more. So that you know what to expect, open and skim each of those files, as with `gedit`. For instance, to open `austinpowers.txt`, open a terminal window and execute the below.

    ```
    gedit ~cs50/pset5/texts/austinpowers.txt
    ```

    Alternatively, launch `gedit`, select **File > Open...**, click **File System** at left, double-click **home** at right, double-click **cs50** at right, double-click **pset5** at right, double-click **texts** at right, then double-click **austinpowers.txt** at right. (If you get lost, simply start these steps over!)

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

    ```
    ./speller ~cs50/pset5/texts/austinpowers.txt
    ```

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

```
~cs50/pset5/speller ~cs50/pset5/texts/austinpowers.txt
```

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

```
MISSPELLED WORDS

[...]
Bigglesworth
[...]
Fembots
[...]
Virtucon
[...]
friggin'
[...]
shagged
[...]
trippy
[...]

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.

Incidentally, to be clear, by "misspelled" we mean that some word is not in the `dictionary` provided. "Fembots" might very well be in some other (swinging) dictionary.

- Alright, the challenge ahead of you is to implement `load`, `check`, `size`, and `unload` as efficiently as possible, 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 set. This problem set 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`.
- You may alter `dictionary.c` (and, in fact, must in order to complete the implementations of `load`, `check`, `size`, and `unload`), but you may not alter the declarations of `load`, `check`, `size`, or `unload`.
- You may alter `dictionary.h`, but you may not alter the declarations of `load`, `check`, `size`, or `unload`.
- You may alter `Makefile`.
- You may add functions to `dictionary.c` or to files of your own creation so long as all of your code compiles via `make`.
- 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 `check` will only be passed strings with alphabetical characters and/or apostrophes.
- You may assume that any `dictionary` passed to your program will be structured exactly like ours, lexicographically 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, and that each word will contain only 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 research hash functions in books or on the Web, so long as you cite the origin of any hash function you integrate into your own code.

Alright, ready to go?

- Implement `load`!

  Allow us to suggest that you whip up some dictionaries smaller than the 143,091-word default with which to test your code during development.

- Implement `check`!

  Allow us to suggest that you whip up some small files to spell-check before trying out, oh, War and Peace.

- Implement `size`!

  If you planned ahead, this one is easy!

- Implement `unload`!

  Be sure to free any memory that you allocated in `load`!

- In fact, be sure that your spell-checker doesn't leak any memory at all. 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.

  ```
  valgrind -v --leak-check=full ./speller ~cs50/pset5/texts/austinpowers
  ```

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

- Don't forget about your other good buddy, `gdb`.

- And CS50 Discuss.

- How to assess just how fast your code is? Well, as always, feel free to play with the staff's solution, as in the below, and compare your output to its!

  ```
  ~cs50/pset5/speller ~cs50/pset5/texts/austinpowers.txt
  ```

  And if you'd like to check the correctness of your program with `check50`, you may execute the below.

  ```
  check50 2012/pset5/speller dictionary.c dictionary.h Makefile
  ```

- Congrats! At this point, your speller-checker is presumably complete (and fast!), so it's time for a debriefing. In `questions.txt`, answer each of the following questions in a short paragraph.

    7. What data structure(s) did you use to implement your spell-checker? Be sure not to leave your answer at just "hash table," "trie," or the like. Expound on what's inside each of your "nodes."

    8. How slow was your code the first time you got it working correctly?

    9. What kinds of changes, if any, did you make to your code over the course of the week in order to improve its performance?

    10. Do you feel that your code has any bottlenecks that you were not able to chip away at?