CSE 303, Winter 2006, Assignment 5B
Due: Wednesday 22 February, 9:00AM

Last update: 12 February

You will implement a data structure for "counting words" and unit-tests for it while other group members *independently* implement some I/O code and some "counter distance" code. The sample solution is 65-70 lines, *not including* testing code, the header file, or the definition of `struct WordCounter`.

Requirements:

- Put your code in two files, `5b.c` and `5b_test.c`. Both should include `5b.h`, which you should write. `5b.h` needs just these prototypes plus typical header-file stuff:

```
struct WordCounter;
typedef struct WordCounter * word_counter_t;

word_counter_t new_counter();
void add_word(word_counter_t counter, char* word, int wordlen);
int get_count(word_counter_t counter, char* word, int wordlen);
int longest_word(word_counter_t counter);
int longer_exist(word_counter_t counter, char * word, int wordlen);
void free_counter(word_counter_t counter);
```

- In `5b.c`, use the definition: `struct WordCounter { int count; struct WordCounter * longer; };` and implement the 6 functions declared in the header file.
- Every `word_counter_t` you create will actually point to an array of 26 `struct WordCounter` objects (one for each lower-case English letter). So `new_counter` should return a pointer to such an array, with each count initialized to 0 and each longer field initialized to `NULL`.
- You may assume that the numeric values for `'a'`, `'b'`, etc. are consecutive and increasing. So if `ch` is a lower-case English character, you may assume `ch-'a'` is between 0 and 25, inclusive.
- For `add_word`, `get_count`, and `longer_exist`, assume `word` points to `wordlen` characters each of which is a lower-case English letter. Do *not* assume a trailing `'\0'`. You *may* assume `wordlen>=1`.
- So a `word_counter_t` can take any word (holding only lower-case English letters) and return a number (its "count"). For example, the count for "cat" in `counter` would be

```
(((((counter['c'-'a']).longer)['a'-'a']).longer)['t'-'a']).count
```

  but if a `longer` field along the "path" is NULL, it must not be dereferenced (of course), and the count is 0.
- `add_word` increments the count for the word it is passed. This may require calls to `new_counter` since any NULL longer field encounterd along the word's "path" will need to be updated.
- `get_count` returns the current count for the word it is passed, as described above.
- `free_counter` deallocates all the space used by a counter *including* all the space used by the counters it points to (and the counters they point to and so on).
- `longest_word` returns the length of the longest word that has a non-zero count in the counter.
- `longer_exist` returns 1 if there is a word with non-zero count that begins with the word `longer_exist` is passed but is *strictly* longer. (For example, if passed `foo`, if `foolish` is in the counter, the result is 1.) Else it returns 0.
- In `5b_test.c` put unit tests for your code and a `main` that runs them.

Advice:

- Understand the data structure before you start coding.
- Keep `longer` fields `NULL` unless you cannot because a longer word has been added.
- Do and test `new_counter`, `add_word`, and `get_count` before proceeding. You will need loops that maintain the "current position" in the word and the "current counter in the data structure". In the sample solution, these loops return directly on their last iteration (when the position is `wordlen-1`) if not before, so they may *look* like infinite loops.
- For `longest_word`, use recursion: The longest word in a counter is:
  - 1 more than the longest word in any counter it points to, unless it does not point to any counters.
  - 1 if it points to no counters but has a count that is greater than 0.
  - 0 if it points to no counters and has all counts that are 0. (This should only happen for the "topmost" counter before any words are added.)
- `longer_exist` is very much like `get_count` except it returns 1 if at the end of the path is a non-NULL longer.

Assessment and turn-in:

Your solutions should be:

- Correct C code that compiles without warnings using `gcc -Wall` and does not have space leaks
- In good style, including indentation and line breaks
- Of reasonable size

Your test code should provide good *coverage*.

Use `turnin` for course cse303 and project hw5. If you use late-days, use project hw5late1 (for 1 late day) or hw5late2 (for 2) instead of hw4.