CSE/IT 122: Homework 7
Stacks and More Hashing

Download `hw7.tar.gz` from Canvas. It contains a number of files you need for this homework.

Follow the Linux coding style and make sure you comment your code using Doxygen style comments.

Any dynamic memory allocation should be checked with valgrind for errors. Make sure you free memory correctly in all problems that use dynamic memory techniques.

Do not change any provided structures.

Follow any sample output exactly.

Problems

1. Write a generic stack library. The files `stack.[c,h]` contain the function declarations and structure you need to use. You don't need any other stack functions so don't pollute the header with any of your own functions. Any helper functions needed in `stack.c` should not be exposed in the header file. The file `test_stack.c` is provided to test and demo the usage of the stack library. It generates 10 random integers and prints them. You should run this through valgrind to make sure your memory management works.

   The stack uses a sentinel node to keep track of the count of the number of nodes in the stack and a pointer to the head node in the singly linked list. Make sure you divide the responsibility of the stack between the user and the library. The user is responsible for providing and freeing any data. You pass the data in as a void * to the stack. The stack library manages the stack and handles any memory allocation's the library does.

   Do not change the name of the files.

2. The file `postfix.c` and `lex.[h.c]` give an example of tokenizing postfix input. Your task is to use the given tokenizer and write a postfix calculator using the stack library you just wrote. Use the postfix algorithm described in class.

   As you calculate the postfix expression, you can also check that a valid postfix expression was entered (details left to you). Either return an error message if for some reason the postfix expression is invalid or the answer if it was a correctly entered.

As written the output of the postfix program just returns what tokens were entered. See the sample output below. Your job is, knowing what token was entered, to carry out the postfix calculation using a stack. As everything is converted to a double, the operators you need to use are +, -, *, /, and ^ where the caret represents exponentiation (i.e. 3 5 ^ = 3^5).

Sample postfix output – note how minus signs are handled.

```
./postfix
Welcome to the postfix calculator
Enter a postfix expression -- all expressions are
converted to floating point.
Enter CTRL + C to quit or type "quit"
postfix> 3 4 5
TOK_NUM: 3 (3.000000)
TOK_NUM: 4 (4.000000)
TOK_NUM: 5 (5.000000)
TOK_NL
postfix> 3 4 5 + *
TOK_NUM: 3 (3.000000)
TOK_NUM: 4 (4.000000)
TOK_NUM: 5 (5.000000)
TOK_ADD
TOK_MULT
TOK_NL
postfix> -3 -4 -
TOK_NUM: -3 (-3.000000)
TOK_NUM: -4 (-4.000000)
TOK_SUB
TOK_NL
postfix> -3-4-
TOK_NUM: -3 (-3.000000)
TOK_NUM: -4 (-4.000000)
TOK_SUB
TOK_NL
postfix> quit
goodbye
```

3. Lisp loves parenthesis, so it's easy to write lisp with unbalanced parenthesis. Use your stack library and write a balanced parenthesis checker for Common Lisp. Test with the lisp files in the tarball. Pass in the filename as argv[1].

   To do this you can write a tokenizer that reads in the file a char at a time (use `fgetc()`). And returns a token for the left parenthesis and a token for the right parentheses and ignores all other input except new lines which it uses to keep track of line numbers.

   While you are testing you program use variations of silly.lisp before reading in the other *.lisp files.

For errors, you need to report the line number of unmatched left or right parenthesis. You can use another stack to keep track of line numbers for left parenthesis mismatches. Do you need a stack for right parenthesis mismatches? Report all errors and the line number it occurs on.

Name your program `bal_paren.c` and your lexer files `lex_lisp*`.

4. Stop Words and Hashing

   If you are trying to index a webpage or a book or any text for that matter, one strategy would be to simply store every word on the page. This, of course, allows for duplicate words which would waste space on the server. To save space, you remove all duplicate words from the text to construct your index. While saving significant space with this strategy, you soon realize you can do better if you filter out words that are common and provide little lexical meaning. Such words are called stop words. Example of such words are `'the'` and `'a'`. Searching on stop words doesn't yield useful results as all or a large portion of pages have those words. Try searching on `'the'` in Google and look at the results.

   There is little agreement about what constitutes a stop word and filtering based on stop words is problem dependent. But overall the strategy is effective. For example, this strategy has been employed by Stack Overflow to speed up their queries: "One of our major performance optimizations for the related questions query is removing the top 10,000 most common English dictionary words (as determined by Google search) before submitting the query to the SQL Server 2008 full text engine. Its shocking how little is left of most posts once you remove the top 10k English dictionary words. This helps limit and narrow the returned results, which makes the query dramatically faster." `http://blog.stackoverflow.com/2008/12/podcast-32/`

   The tarball contains a file entitled `1000_words.txt`, which contains a list, in order, of the 1000 most common words in the English language. Like all such lists, the list has a bias depending on which corpus you use (text or spoken) and are you including only American English, etc. This list is extracted from American English text.

   You will use this list to filter the words in the story The Metamorphosis by Franz Kafka (metamorphosis.txt) (a German text translated into English) and Gadsby by Ernest Wright (gadsby.txt) to create sets of unique uncommon words in the texts. For purposes here, we will consider a word as beginning with characters that make isalpha() return true. So don't store any page numbers like the ones that appear in the Gadsby text.

   You are going to write a program that filters out the 1000 most common words in the text and ultimately will print out a sorted list of unique uncommon words and their frequency of the text.

   To do this the basic idea is to:

   (a) Hash the 1000 most frequent words. (use chaining; table size 250).

   (b) Read in the file character by character (fgetc()) and tokenize the text into words (see below).

(c) If the word is a frequent word, then discard it.

(d) If the word is not a frequent word and it is not already in the list of uncommon words add it to a chained hash table. The hash table size should be a runtime decision and passed in as a command line argument. You should determine the load factor and see if the table size works. A load factor less than 5 is considered as acceptable. In your code, as a comment in the first line of main, put what table size you ultimately ran the program with for the two files.

(e) If the word is already in the uncommon word hash, just increment the word count (word frequency).

(f) sort the list of uncommon words using qsort(). Now you cannot use the hash table to sort, just transform the hash table into something you can sort (i.e. an array of char *).

(g) save the sorted uncommon word list and the word's frequency to a text file named `meta_uncommon.txt` and `gadsby_uncommon.txt`. The output should be the word, a tab, its frequency count, followed by a new line.

Name your source code file `unique_words.c`. Use Bernstein's algorithm to generate keys from strings. Pass in the filename and table size of the uncommon word hash as command line arguments so the hash table can be adjusted as necessary. Name your tokenizer `lex_word.[c,h]`. Also you should use lower case for all words.

Tokenizing Text

You are going to process the file, character by character using `fgetc()`, extracting and hashing words as you encounter them.

Now, for a majority of cases, the end of a word can be determined if you encounter a space or a line terminator. Unfortunately, extracting tokens is complicated by the fact other punctuation terminates words such as commas, periods, `', '', ?, !, :` to name a few. In addition, the text uses double quotes for phrases spoken by characters; single quotes are used as contractions. Single hyphens `-` are used for compound words which we will consider as two words (e.g. self-reproach), but hyphens are also used as a separator in the text (e.g. dozing - and) and for hyphenation. Also the texts uses `--` which should be considered as two words (e.g. school--he). You need to account for all these cases in your tokenizer. Most of the punctuation is handled by the use of `ispunct()` found in `ctype.h` (`ctype.h` contains other useful functions that will aid in this program).

When you print out the sorted list you should check to make sure you are not printing out punctuation. Adjust your tokenizer as necessary.

Submission

Tar your source code, Makefile, and output files into a file named:

`cse122_firstname_lastname_hw7.tar.gz`

and upload to Canvas before the due date.