Word Freak with a hashtable of BSTs

Project Profile

The high level goal of this project is to write a program called "wordfreak" that takes "some files" as input, counts how many times each word occurs across them all (considering all letters to be lower case), and writes those words and associated counts to an output file in alphabetical order.
We provide you some example book text files to test your program on. For example, if you ran

```
$ ./wordfreak aladdin.txt
```

Then the contents of the output file would be:

```
$ cat output.txt
a            :  49
aback        :   1
able         :   1
...
required     :   1
respectfully :   1
retraced     :   1
...
that         :  11
the          : 126
their        :   2
...
you          :  20
young        :   1
your         :   7
```

The words from all the input files will be counted together. If a word appears 3 times in one input file and 4 times in another, it will be counted 7 times between the two.

Input

wordfreak needs to be able to read input from 3 sources: standard input, files given in argv, and a file given as the environment variable. It should read words from all these that are applicable (always standard in, sometimes the other 2).

A working implementation must be able to accept input entered directly into the terminal, with the end of such input signified by the EOF character (^D (control+d)):

```
$ ./wordfreak
I can write words here,
and end the file with control plus d
$ cat output.txt
and     : 1
can     : 1
control : 1
d       : 1
end     : 1
file    : 1
here    : 1
i       : 1
plus    : 1
the     : 1
with    : 1
words   : 1
write   : 1
```

However, it should alternately be able to accept a file piped in to standard input via bash's operator pipe (for more details on the bash pipe | operator see this tutorial):

```
$ cat aladdin.txt | ./wordfreak
```

It should be noted that your program has no real way to tell which of these two situations is occuring, it just sees information written to standard input. However, by just treating standard input like a file, you will get both of these behaviours.

A working implementation must also accept files as command line arguments:

```
$ ./wordfreak aladdin.txt iliad.txt odyssey.txt
```

Finally, a working implementation must also accept an environment variable called **WORD_FREAK** set to a single file from the command line to be analyzed:

```
$ WORD_FREAK=aladdin.txt ./wordfreak
```

And of course, it should be able to do all of these at once

```
$ cat newton.txt | WORD_FREAK=aladdin.txt ./wordfreak iliad.txt odyssey.txt
```

Words

Words should be comprised of only alpha characters, and all alpha characters should be taken to be lower case.
For example "POT4TO???" would give the words "pot" and "to". And the word "isn't" would be read as "isn" and "t". While this isn't necessarily intuitively correct, this is what your code is expected to do:

```
$ echo "Isn't that a POT4TO???" | ./wordfreak
$ cat output.txt
a    : 1
isn  : 1
pot  : 1
t    : 1
that : 1
to   : 1
```

You are required to store the words in a specific data structure. You should have a binary search tree for each letter 'a' to 'z' that stores the words starting with that letter (and their counts). This can be thought of as a hash function from strings to binary search trees, where the hashing function is just first_letter - 'a'. Note that these BSTs will not likely be balanced; that is fine.

Output

The words should be written to the file alphabetically (the BSTs make this fairly trivial). Each word will give a line of the form "[word][additional space] : [additional space][number]\n". The caveat is that all the colons need to line up. The words are left-aligned and the longest will have a single space between its end and the colon (note "respectfully" in the example below); the numbers are right-aligned and the longest will have a single space between the colon and its beginning (note 126 in the example below).

```
$ ./wordfreak aladdin.txt
$ cat output.txt
a           :  49
...
respectfully :   1
...
the         : 126
...
your        :   7
```

The output file should be named output.txt. Note that when opening the file to write to, you will either need to create the file or remove all existing contents, so make use of open()'s O_CREAT and O_TRUNC. Moreover, you will want the file's

permissions to be set so that it can be read. open()'s third argument determines permissions of created files, something like 0644 will make it readable.

Restrictions

Your submission is **restricted to only using** the following system calls: open(), close(), read(), write(), and lseek() for performing I/O. You are allowed to use other C library calls (e.g., malloc(), free()). However, all I/O is restricted to the Linux kernel's direct API support for I/O. You are also allowed to use sprintf() to make formatting easier.

Submission

You must submit the following to Gradescope by the assigned due date:

- **Source Files** - source and header files that provide the interface and implementation of the solution to this project.
- **`Makefile`** - the build file to use with the **`make`** program. In particular, the executable created by this Makefile must be named "wordfreak".
- **`README.txt`** - this is a text file containing an overview/description of your submission highlighting the important parts of your implementation. It should explicitly explain where in your implementation your code satisfies the requirements of having BSTs and the "hashing function". The goal is to make it easy and obvious for the person grading your submission to find the important rubric items. This text file should also clearly include a URL to your video for us to review.
- **Video** - You must provide a link to a 2-minute video demonstration. Your video should be short and get to the point. It should show your code being compiled from scratch using your Makefile. It should also show the program being run on four types of inputs: standard input via entering in terminal when prompted, standard input via piping in, filenames as arguments, and a file name in the environment variable. For the latter two, since you always need to enter something for standard input, you may choose to do something like the following
    - `$ echo "" | ./wordfreak [filenames]`
    - `$ echo "" | WORD_FREAK=[filename] ./wordfreak`

Your code must compile on the EdLab Linux machine. Make sure you submit this project to the correct assignment on Gradescope!

Grading Breakdown

1. GradeScope Submission (1 point)
    a. You automatically get a point for submitting to gradescope. (1 point)
2. Deliverables (10 points)
    a. Source and header files (2 points)
    b. Makefile (2 points)
    c. Executable generated by Makefile named "wordfreak" (2 points)
    d. README.txt (2 points)
    e. Video link in README.txt (2 points)
3. Output Requirements (25 points)
    a. stdin: file piped in or words typed in to standard in are read, words are counted, alphabetised, and written to output.txt (5 points)
    b. arguments: files passed in as arguments are read, words are counted, alphabetised, and written to output.txt (5 points)

   c. environment: file passed in as environment variable is read, words are counted, alphabetised, and written to output.txt (5 points)
   d. all: program can handle files piped in or words typed in to standard input, files passed in via argv, and a file as the environment variable all at once (5 points)
   e. format: output.txt adheres to the formatting rules (5 points)
4. Structure Used (20 points)
   a. Binary search trees are used as specified (5 points)
   b. The specified "hash function" is used, there is a BST for each letter 'a' to 'z' (5 points)
   c. The only file I/O functions used are open(), close(), read(), write(), and lseek() (10 points)
5. Design (15 points)
   a. Minimal use of global variables, say no more than 5 (2.5 points)
   b. Code organised: broken into files in a reasonable way, separated into functions in a reasonable way (2.5 points)
   c. Data structures used to keep things organised (2.5 points)
   d. Reasonably efficient (things like no obvious places where a for loop could be used to simplify code) (2.5 points)
   e. Error Checking: I/O functions are error checked, especially open(). If you fail to open a file, you should gracefully move on to the next (5 points)
6. Style/Aesthetic (5 points)
   a. Consistent bracketing style (2.5 points)
   b. Consistent indentation style (2.5 points)
7. Comments (9 points)
   a. Functions: comments describing purpose, inputs, and returns of functions where applicable (3 points)
   b. Variables: comments describing the meaning/purpose of non-trivial and non-obvious variables (3 points)
   c. Misc Code: comments describing non-trivial regions of code/control flow (3 points)
8. Video (10 points)
   a. Video shows code being compiled from scratch with "make" (use "ls" to show there are no object files/executables before "make") and the result is replicable (2 points)
   b. stdin pipe: Video shows running wordfreak with file piped in to standard in (shows output.txt afterwards) and the result is replicable (2 points)
   c. stdin inputs: Video shows running wordfreak with words typed in to standard in (shows output.txt afterwards) and the result is replicable (2 points)
   d. arguments: Video shows running wordfreak with files as arguments (shows output.txt afterwards) and the result is replicable (2 points)
   e. environment: Video shows running wordfreak with file passed as environment variable (shows output.txt afterwards) and the result is replicable (2 points)
9. README (5 points)
   a. Overview/description of the overall structure of the program (2 points)
   b. Explains where in code BST requirement is satisfied (1.5 points)
   c. Explains where in code "hash function" requirement is satisfied (1.5 points)