Cosc 242 Assignment 2

Due: 4pm Friday October 1st 2021

Overview

In this assignment you will expand and modify code written during the labs to produce a program based around a tree data structure. This program can be used to process two groups of words. The first group of words will be read from *stdin* and will be inserted into the tree. The second group of words will be read from a file specified on the command line. If any word read from the file is not contained in the tree then it should get printed to *stdout*. A moment of reflection will confirm that this program can be used to perform a basic spell check. You can begin working on this assignment while you are completing the labs it is based on.

Assignment groups

Everyone in the class should form groups of three to work on the this assignment. You have until 1pm Wednesday August 8th to select your groups. Please send an email to iain.hewson@otago.ac.nz with the names and University user codes of your group members (e.g. stuad123, brofr456, jonsu789). If you don't select your own group then one will be chosen for you containing students who have completed a similar amount of internal assessment. You will be informed of your group members via email, so please check your University email regularly and let Iain know if there are any problems. The related labs, and all of the programming for this assignment should be done working in your groups.

You must not collaborate with, or discuss issues related to the assignment with anyone who is not a member of your group.

Version Control

All code for this assignment must be stored in the departmental GitLab server `https://altitude.otago.ac.nz/`. You can log in using your regular Computer Science usercode and password. One group member should create a new blank project called "242-a2" and initialize it with a README containing the names and usercodes of the other group members. The project should have private visibility. After creating the project go to "Project information -> members" and add the other group members and Iain Hewson "ihewson" giving everyone the role of "Maintainer". Make sure that you have a top level directory in your project called "asgn" containing all of the .c and ,h files needed for your assignment. You can put other files in the project as you see fit e.g. separate directories for each group member or combined lab work etc. The only files that will be marked are those found in the top level asgn directory.

Part of the marks for this assignment come from appropriate use of version control. There is a comprehensive freely available book "Pro Git" that you might find helpful https://git-scm.com/book/en/v2. Note that we don't require you to use branching.

Provided files

We have provided some files for you to use when completing this assignment. The files can be found in the directory `/home/cshome/coursework/242/asgn-files/` and are as follows:

- `output-dot.txt` — contains extra functions which you should add to your `tree` files.
- `sample-asgn2` — an executable, compiled on Linux, which you can use to see if your program is working correctly. If your program is correct then it should produce exactly the same output as the sample program.

Comparing BSTs and RBTs

We require you to combine the code you have written for binary search trees and red-black trees in the labs into a tree data structure as described below. Once you have done this, the only difference between a bst and an rbt is that the rbt calls a `tree_fix()` function after every insertion to make sure that the tree is balanced.

This program needs to meet the following requirements:

- Create a combination `tree` ADT which can be either an ordinary bst or a balanced rbt depending on an enumerated type which gets passed to `tree_new()`.
- Make a static `type_t` variable called `tree_type` which can hold the value passed to the tree 'constructor'. The definition of the enumerated type in `tree.h` should look like this:

  ```
  typedef enum tree_e { BST, RBT } tree_t;
  ```

- You don't need to implement a `tree_remove()` function since removing nodes from an rbt can be a bit tricky. In fact you might as well remove your old `bst_remove()` code since using it would break an rbt.
- Add a `tree_depth()` function which should return the length of the longest path between the root node and the furthest leaf node.
- Add a `frequency` field to your `struct tree_node` and update the frequency whenever a duplicate item is added to the tree.
- Add the two `output_dot` graph printing functions which we have provided to your tree ADT. These will enable you to visualise what your tree looks like – including the colours and frequencies. You might find it useful to use these functions when completing your tree labs. When you run your tree program with the `-o` option it should produce a file (`tree-view.dot` by default) containing a representation of your tree using the "*dot*" language[1]. You can produce a nice image of your tree by running the command

[1] Dot is a plain text graph description language. For more information see http://www.graphviz.org.

```
dot -Tpdf < tree-view.dot >| tree-view.pdf
```

in a terminal.

- You may have noticed that the rbt you implemented in labs doesn't ensure that the root is always black. This doesn't affect the structure of the tree at all, but you should try to fix this problem in your program.

The asgn2.c file

You need to create a main file called `asgn2.c` which uses a tree to perform a number of tasks. By default, words are read from stdin and added to your tree before being printed out alongside their frequencies to stdout. This should be done by passing a `print_info` function, shown below, to `tree_preorder`.

```
static void print_info(int freq, char *word) {
   printf("%-4d %s\n", freq, word);
}
```

All memory allocated should be deallocated before your program finishes.

All words should be read using the `getword()` function from the lab book. Helper functions like `getword` and `emalloc` should be in your mylib.c file.

You should use the *getopt* library to help you process options given on the command line. Here is an example of how to use it:

```
const char *optstring = "ab:c";
char option;

while ((option = getopt(argc, argv, optstring)) != EOF) {
   switch (option) {
      case 'a':
         /* do something */
      case 'b':
         /* the argument after the -b is available
            in the global variable 'optarg' */
      case 'c':
         /* do something else */
      default:
         /* if an unknown option is given */
   }
}
```

You need to include `getopt.h` to use the getopt library. The letters listed in optstring are possible valid options. The colon following the letter `b` indicates that `b` takes an argument. As the options are being processed by getopt, they get shifted to the front of the argv array.

After processing, the index of the first non-option argument is available in the global variable optind. For more information have a look at the man page for getopt.h (type `man 3 getopt`).

When given the command line option `-c filename`, your program will be used to process two groups of words. The first group will be read from *stdin* and put into the tree as usual. These words will function as a dictionary. The second group of words will be read from the file specified on the command line. If any word read from the file is not contained in the dictionary then it should get printed to *stdout*. Running your program with a command like this

```
./asgn2 < dictionary.txt -c document.txt
```

should print out a list of every word from `document.txt` which is not found in `dictionary.txt`. If there is no output then document.txt has no misspelled words (as defined in dictionary.txt).

The exact behaviour of your program when given the `-c filename` option should be as follows:

1. Take `filename` to be the plain text file that we want to check the spelling in.
2. Read words from *stdin* (using the `getword()` function) and put them into our tree. The tree will now function as our dictionary.
3. For each word we read from `filename` (using `getword()`) check to see if it is in our dictionary. If it is then don't do anything. If it is not then print the word to *stdout*.
4. When finished checking `filename` for unknown words print timing information and unknown word count to *stderr* like this:

```
Fill time     : 0.320000
Search time   : 0.180000
Unknown words = 8690
```

When your program is given the `-h` option, or an invalid option is given, then a usage message should be printed and your program should exit.

- Your program should respond to command line arguments as specified in the following table:

Option	Action performed
-c *filename*	Check the spelling of words in *filename* using words read from stdin as the dictionary. Print all unknown words to stdout. Print timing information and unknown word count to stderr. When this option is given then the `-d` and `-o` options should be ignored.
-d	Print the depth of the tree to stdout and don't do anything else
-f *filename*	Write the "*dot*" output to *filename* instead of the default file name if `-o` is also given.
-o	Output a representation of the tree in "*dot*" form to the file 'tree-view.dot' using the functions given in `output-dot.txt`.
-r	Make the tree an rbt instead of the default bst.
-h	Print a help message describing how to use the program

Submission

Instead of requiring you to submit your assignment before the due date, we will pull the latest version from the shared repository. Your project should have a top level directory called `asgn` which contains the following files:

```
asgn2.c            tree.c
mylib.c            tree.h
mylib.h
```

- Your assignment should be ready for collection by 4pm on the due date.
- If you have any concerns about the contributions of any of your group members please send an email to iain.hewson@otago.ac.nz.

Marking

This assignment is worth 20% of your final mark for Cosc 242. It is possible to get full marks. In order to do this you must write correct, well-commented code which meets the specifications, and use the departmental GitLab server as specified.

Program marks are awarded for both implementation and style (although it should be noted that it is very bad style to have an implementation that doesn't work).

Allocation of marks	
Implementation	10
Style/Readability	6
Use of version control	4
Total	20

In order to maximise your marks please take note of the following points:

- Your code should compile without warnings on the Linux lab machines using the command:

  ```
  gcc -O2 -W -Wall -ansi -pedantic -lm *.c -o asgn2
  ```

  If your code does not compile, it is considered to be a very, very bad thing!
- Your program should use good C layout as demonstrated in the lab book, including no lines more than 80 characters long.
- Most of your comments should be in your function headers. A function header should include:

- A description of what the function does.
- A description of all the parameters passed to it.
- A description of the return value if there is one.
- Any special notes.

Overdue assignments will lose marks at a rate of 10% per day late.

You should not discuss issues pertaining to the assignment with anyone not in your group. All programs will be checked for similarity.

Part of this assignment involves you clarifying exactly what your program is required to do. Don't make assumptions, only to find out that they were incorrect when your assignment gets marked.

You should check your University email regularly, since email clarifications may be sent to the class email list.

If you have any questions about this assignment, or the way it will be assessed, please see Iain or send an email to iain.hewson@otago.ac.nz.