Cosc 242 Assignment

Due: 4pm Friday September 15th 2017

Group work

For this assignment we require you to work in groups of three people. You may select your own group and inform us of your choice via email by August 16th.

Send an email to ihewson@cs.otago.ac.nz letting us know the names and University user codes of all students in your group. Any students who don't select their own group will be assigned to one and informed via email on August 18th. Groups will be assigned in a pseudo-random manner (students will be teamed up with others who have completed a similar amount of internal assessment).

The related labs, as well as 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.

Overview

In this assignment you will combine some of the code written in the labs to produce a larger program. This program will be used to process two groups of words. The first group will be read from a file specified on the command line. The second group will be read from stdin. If any word read from stdin is not contained in the file then it will get printed to stdout. A moment of reflection will confirm that this program could be used as a rudimentary spell-checker. Running your program with a command like this:

```
./asgn dictionary.txt < 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).

Basic outline

The data structure that you will use to hold the words is a hash table. Each position in the hash table will be a container which can hold multiple items (i.e. chaining). You will create two types of container, one using a very simple chaining method, and one using a more robust chaining method. The idea behind this is to make sure that your hash table won't be too slow

to search, even if the hash function performs poorly. If your hash table size is reasonable and you have a decent hash function then the simple chaining method will work fine. However, if your hash function hashes lots of items to the same position then the hash table will become too slow for practical use.

You should use the hash function given in lab 12 for this assignment. We can simulate a bad hash function simply by reducing the size of the hash table. This will ensure that lots of items hash to the same position in the table.

Your program will create a hash table and fill it with words read from a file specified on the command line.

Your hash table will be made up of *containers* which can hold any amount of items. A container is really just a wrapper for a flexarray or a red-black tree. So, for example, adding something to a container will either append it to the underlying flexarray, or insert it into the underlying red-black tree.

Once all of the words in the file have been put into the hash table, words get read from stdin. If a word read from stdin is not found in the hash table then it gets printed to stdout, otherwise nothing happens.

When there are no more words to read from stdin your program finishes.

Specific requirements

In order to complete this assignment successfully there are a number of requirements you will need to meet, including the following:

- The default size of the hash table should be 3877.
- All memory allocated should be deallocated before your program finishes.
- Change a copy of your flexarray from lab 10 so that it works with strings instead of integers. Add an `is_present` function, and a `visit` function (this should take a function as a parameter which is used to print each item in your flexarray in a similar manner to the RBT preorder function).
- 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.
- Modify your RBT to make it possible to add duplicate words.
- 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.
- When printing out containers from your hash table you should print the contents of each non-empty container on a single line with each item separated by a space. You should also print the index of the container in the hash table at the start of the line. If the container type is an RBT then the items should be printed in preorder.

- The container type should be defined by an enumerated type declaration in container.h like this:

```
typedef enum container_e {FLEX_ARRAY, RED_BLACK_TREE} container_t;
```

- Your program should take a number of options on the command line to alter its behaviour as specified in this table.

Option	Action performed
-r	Use a robust chaining method. This tells your hash table to use a red-black tree as its container type.
-s *table-size*	Use *table-size* as the size of your hash table. You can assume that table-size will be a number greater than 0.
-p	Print your hash table to stdout one line per non-empty container. The index of the container in the hash table should be printed at the start of the line. Don't read anything from stdin or print anything else out when this option is used.

-i	Print information (to stderr) about how long it took to fill the hash table, how long it took to search the hash table, and how many unknown words were found like this:
	`Fill time : 1.390000`
	`Search time : 0.450000`
	`Unknown words : 8690`

-h	Print a help message to *stderr* describing how to use the program and then exit

- Create a "container" data structure which is just a wrapper for a flexarray or a red-black tree. It should contain a struct which looks like this:

```
struct containerrec {
   container_t type;
   void *contents;
};
```

- Each of the functions in `container.c` will usually just call the appropriate function in either the flexarray or red-black tree. e.g.

```
void container_add(container c, char *word) {
   if (c->type == RED_BLACK_TREE) {
      c->contents = rbt_insert(c->contents, word);
   } else {
      flexarray_append(c->contents, word);
   }
}
```

- Use the *getopt* library to help you process the 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`).

- If `-h` (or an invalid option) is given as a command-line argument then a usage message should be printed to *stderr* and your program should exit.

Submission

In order to submit your assignment files open a terminal and change into the directory which contains all of your files. Type the command *asgn-submit* and press return. You should see this list of files printed.

```
asgn.c            flexarray.h       mylib.h
container.c       htable.c          rbt.c
container.h       htable.h          rbt.h
flexarray.c       mylib.c
```

If the file list looks correct then just press return, and you will be asked to rate the contribution of the members of your group. Choose the appropriate rating and you should then see the message:

```
Submission complete.
```

- Your assignment should be submitted before 4pm on the due date.
- All group members should submit a copy of the assignment

Marking

This assignment is worth 15% of your final mark for Cosc 242. A lot of the code will have already been written as you completed labs during the course. If you have completed all of the labs and are careful to write clean, well-commented code which meets the specification you can get full marks.

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

Allocation of marks	
Implementation	9
Style/Readability	6
Total	15

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 asgn
  ```

  If your code does not compile, it is a considered to be a very, very bad thing!
- Your program should use good C layout as demonstrated in the lab book.
- No line should be 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.

Any assignments submitted after the due date and time 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 Computer Science 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 ihewson@cs.otago.ac.nz.