CS-2303, System Programming Concepts, C-term 2017

Project 3 (40 points)
Assigned: Monday, January 30, 2017
Due: Monday, February 6, 2017, 6:00 PM

Programming Assignment #3 — Binary Trees

Abstract

Write a program in *C* to scan one or more text files and count the number of occurrences of each word in those files. Use a binary tree to keep track of all of the words. When all input files have been scanned, print out a sorted list of the words to another file, along with the number of occurrences of each one. For extra credit, set up your tree to be an AVL tree, so that it is always balanced.

This project is similar to Assignment #HW5 in CS-1004, Introduction to Programming for Non-majors, in A-term 2016 as described here:– docx, pdf. In that assignment, students wrote in *Python* and used a *Python dictionary* as the principal data structure. This assignment is in *C* and requires a *binary tree* as the principal data structure.

Outcomes

After successfully completing this assignment, you should be able to:–

- Define a **`struct`** and organize your *C* program in an object-oriented style.
- Build a massive, recursive data structure comprising those objects.
- Search for an item in that data structure and, if it is not found, add it to the structure.
- Create a file for your output and write to it.
- Carry out simple string manipulation
- Put together a non-trivial program in *C*.

Before Starting

Review the following sections in Kernighan and Ritchie:–

- §5.1–5.2 regarding pointers;
- §§6.1–6.5 regarding **`structs`** and self-referential data structures;
- §7.5 regarding the creation, opening, and writing of files; and
- §7.8.5 regarding **`malloc()`** and **`free()`**.

This Assignment

Your program should be called **`PA3`**. Your program must accept an indeterminate number of arguments on the command line. The first argument specifies the output file, and the remaining arguments specify a sequence of input files. Thus, a user could invoke your program from a command line by typing

```
./PA3 outputFile inputFile1 inputFile2 ...
```

Under this command, the program would open and read each input file in turn, building up a *binary tree* of words and counts as it progresses. Once all files have been read and closed, the program must

create the output file and write out the words in the tree *in alphabetical order*, one word per line, along with the number of occurrences of that word. Your program should ignore the case of the words, so that "**`This`**" and "**`this`**" are considered the same. However, words that are actually spelled differently — such as "**`car`**" and "**`cars`**" — are considered to be different words. You should recognize contractions such as "**`won't`**" and possessives such as "**`Bob's`**" as words. You should also treat hyphenated words such as "**`hard-hearted`**" as one word, not two.

A sample output might look like the following:–

```
   166    a
    25    and
    11    as
     3    command
    15    each
     2    file
     4    files
   109    in
     4    input
    98    it
     1    it's
    99    of
     3    open
     6    program
    18    read
   152    the
    41    this
     3    under
    30    would
------------
    19    Distinct words
   790    Total words counted (including duplicates)
```

To allow for very long input files, the field width of the number of occurrences of each word should be at least six decimal digits. You must total and print the number of distinct words *and also* the total number of words counted (including duplicates).

For debugging, you may use the following text files:–

http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/Kennedy.txt
http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/Obama.txt
http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/MartinLutherKing.txt

You may also use other non-trivial documents (e.g., essays, newspaper or magazine articles, and speeches). Here are three Shakespeare plays:–

http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/Macbeth.txt
http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/MuchAdoAboutNothing.txt
http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/TamingOfTheShrew.txt

Finally, the following is a particularly long file containing the great American novel *Moby Dick*:–

http://www.cs.wpi.edu/~cs2303/c17/Resources_C17/MobyDick.txt.

Implementation in C

You must implement this project in an object-oriented style. Think, for example, how you might have done it in Java, and then use that approach as a guideline for organizing your *C* program.

At the very minimum, you should define two or more **`.h`** files and a corresponding **`.c`** file for each **`.h`** file. Each **`.h`** file corresponds to a different part of your program — e.g., the tree management, input and output, etc. — much the same way that you would organize your classes in *Java*. In addition, you should also have a separate **`.c`** file for your **`main()`** function and its supporting functions.

For example, you might define a **`tree.h`** to be the interface to the binary tree. This would include the headers to functions for adding nodes to the tree, iterating through the tree, and deleting the tree (and all of its nodes). The companion **`tree.c`** would define the tree data structure (including the **`struct`** for the nodes the pointer to the root). It would contain the implementations of these functions. This would be the intellectual equivalent of a *BinaryTree* class in Java.

For input and output, you should use *C* functions such as **`fgets()`** and **`fprintf()`**.

Compiling C programs

To compile your *C* program, create a **`makefile`** along the lines of Laboratory #1. The compiler command line for **`.c`** files should be

```
gcc $(CFLAGS) -c yourFile.c
```

The make variable **`CFLAGS`** should default to **`-g0`**.

Note: Don't forget to delete your tree and free all of the nodes and strings in it before your program exits. Failure to do so can result in memory leaks. The graders will be looking for this.

String manipulation

A traditionally challenging part of this assignment is the handling of strings. Recall that a *string* in *C* is an array of characters terminated by the null character (i.e., **`'\0'`**). The recommended approach for this assignment is to do something similar to the *Python* version assigned to the CS-1004 class. That is:–

- read the text of each input document;
- partition it into substrings at "whitespace" boundaries;
- for each substring, "strip" the punctuation from the beginning and end;
- if the remaining substring is non-null, enter it into the binary tree data structure as a word.

"Whitespace" is defined at the space, newline, tab, vertical tab, form feed, and carriage return characters — i.e., any of the characters in the string **`" \n\t\v\r\f"`**. These define the boundaries between words and word-like substrings. An algorithm for splitting may be discussed in class.

Punctuation characters to strip from the beginnings and end of words typically include the characters **`".,?-;:()[]!\"\'"`**. If you find other characters that should be stripped in the input files, you may include them. An algorithm for stripping may be discussed in class.

Binary tree

Kernighan and Ritchie describe binary tree data structures in §6.5. You may base your implementation on the code in this chapter, but blindly copying that code will probably not meet the needs of

this assignment. AVL trees (for extra credit) are *not* covered in Kernighan & Ritchie, but they were covered in CS-2102, Object-Oriented Programming.

Individual or Team Project

You may carry out this project individually, or you may optionally carry it out in a two-person team. In either case, you may consult classmates and others on the algorithms for creating and manipulating binary trees — for example by drawing pictures on a whiteboard or other temporary medium — but you must implement them in your program on your own or just with your teammate.

If you wish to work as a team, *you must register your team* with the professors so that it can be entered into *Canvas*. Your team name will consist of the WPI usernames of both members, in alphabetical order and separated by a hyphen. Once registered, either teammate may submit on behalf of the team.

Deliverables

You must carry out this project on the course virtual machine. You may use any tools, but you are strongly encouraged to use an IDE such as *Eclipse*. When you are ready to submit, clean the project and then export it to a *zip* file named **`PA3_userName.zip`** or **`PA3_teamname.zip`**, where **`userName`** is replaced by your WPI username and where teamname is the name registered in *Canvas*. The zip file should contain the following:–

- All of the *C* header and program files of your project, including your **`.c`** and **`.h`** files.
- A **`makefile`**. The target name should be **`PA3`**.
- Text files representing at least two non-trivial test cases other than the test cases cited above.
- The output of your test cases showing that the program works.

You must also submit a document called **`README.txt`**, **`README.pdf`**, **`README.doc`**, or **`README.docx`** summarizing your program, how to run it, and detailing any problems that you had. If you borrowed from or consulted with anyone on any part of the algorithm for this assignment, be sure to cite your sources. Your **`README`** file does not need to be part of the *zip* file.

Before submitting your assignment, execute **`make clean`** to get rid of extraneous files. The penalty for an "unclean" submission is 10 points. Submit your program to *Canvas* under the course named *CS2303-C17-LABS*. This assignment is named *PA3*. Programs submitted after the due date (Monday, February 3, 2017, 6:00 PM) will be tagged as late, and will be subject to the late homework policy.

Grading

This assignment is worth forty (40) points. *Your program* must *compile without errors in order to receive* any *credit.*

- Correctly build from the **`makefile`** without warnings (with **`-Wall`** switch – 5 points
- Organization of program into at least three modules – 5 points
- Correct construction of binary tree and insertion of nodes – 5 points
- Correct traversal of binary tree and output of information according to specified format – 5 points
- Correct use of **`malloc()`** and correctly **`free`**ing all **`malloc`**'ed data – 5 points
- Proper destruction of tree and all of its objects before exiting – 5 points
- Correct execution with graders' test cases – 5 points

- Satisfactory **`README`** file, including output of two non-trivial test cases – 5 points

Note: A penalty of ten **points** will be assessed if your project is not *clean* before creating the zip file or for submitting your programs in some format other than a *zip* file.[1]

Note 2: If your program does not compile correctly on the course virtual machine using the **`makefile`**, the graders will attempt to contact you via e-mail. You will have 24 hours to resubmit a corrected version, and a penalty of 25% will be assessed (in addition to other penalties).

Extra Credit — AVL tree

For 25% extra credit, make your binary tree into an AVL tree, so that it is balanced at all times. In addition, submit a graph or a plot that shows the running times of the AVL tree algorithm *versus* the ordinary binary tree algorithm for at least three test cases including at least one small, one medium, and one large input. You should do this *after* you get the program working with an ordinary binary tree. Be sure to explain in your **`README`** file how your AVL tree works and how you know that it is balanced.

If you elect to do this extra credit, it is *strongly* suggested that you get the basic version of the project working first. Submit it as a zip file named **`PA3_userName.zip`**. Next, develop and debug the code to implement the AVL tree, and export that to a file named **`PA3_userName_AVL.zip`**. The graders will attempt to grade your AVL version first. If that does not work, they will fall back to the non-AVL version.

[1] The Professor's computer is **`tar`**-file challenged, **`rar`**-file challenged, as well as challenged for all other kinds of archive files.