Lab 4: Dates

Early Tue 7-March-2023

On-Time: Thu 9-March-2023

Late until Fri 10-March-2023

No labs to do over break! **Take** a break.

Contents

Overview

This is the same basic simulation as lab 3. You can use the lab 3 reference code with the same rules as before. The lab 3 reference code does have a deficiency or two that you will need to fix.

Added features:

- Dynamic memory failures
- Linked list code
- File I/O

- Command line arguments
- Playable bonus mode

See the various other documents for details on the above.

Plan of attack

As soon as you have code that compiles and doesn't instantly crash, create a zip file. Perhaps this is just your prototypes to start with. Do those – they are worth 2 points – and create a zip.

Each time you get something working, rebuild your zip. You should always have a zip ready to turn in. It will reduce your stress and it might save your lab grade.

The lab has multiple independent features, so you can pick and choose what to work on at any given time.

- The linked list code can be added to an existing lab 3 simply by adding one file to the project.
    - Do each function as a separate work item
    - Each time one of them works, recreate your zip file
    - Consider doing iterate first. As soon as it works, rebuild your zip file.
    - Then do insert, but if you can't get it working comment out the code and switch to another category of tasks.
    - Do deltesSome and sort last
- File I/O can be implemented without processing command line parameters by adding a file pointer to the sim structure, initializing it to stdin, and then changing every input call to use that file pointer.
- Once file I/O is in, you can add command line argument processing to set those the file pointer using the filename given on the command line.
    - First do argc validations, one at a time , rebuild your zip after each thing works
    - Start with arc must be at least 2
    - Add code to handle argc = 3, which is nuanced
    - Add code to actually open the file given and the code to close that file
- Adding unreliable allocation is much smaller in scope than the others. It's best done once you have your own insert and deleteSome operational.
    - Start by changing the function call and adding reliable.o to your lab4 entry in the makefile
    - Replicate that makefile entry, call it lab4u, change reliable to unreliable.o
    - Test both heavily, zip as soon as it works

Do not get bogged down! If deleteSome is blowing up your brain, do unreliable allocation and get back to deleteSome later. You do not need a feature-complete lab to get worthwhile points! Consult the spreadsheet when it comes out.

Consider using prototypes to test your linked list code. You may have some old lab 3 prototypes that you can use exactly as they are to test your code. If they test your code instead of the library code, they can be counted for lab 4.

Scoring details

Makefile

You need to have 2 build targets, lab4 and lab4u in your makefile. They will be identical except that lab4 will need `reliable.o` and lab4u will need `unreliable.o`

Do not make it hard on the graders. They must be able to "make –r lab4" and "make –r lab4u" to generate your executables. Failures here can be catastrophic to your lab score.

Linked List

- Linked list functions that allocate and free nodes are marked static
- Only the linked list code has any idea of what a node is
- Linked list must not know what a sim or process structure is

Dynamic Memory

- Both allocation routines check for NULL (and work with unreliable allocation) and properly handle failures, including printing an error message. Your code must use the alternative calls such as alternative_free().
- Both (ball & node) allocators and both free routines count and print the count
- Free counts == allocate counts

I/O and Arguments

- Arg count is checked and code reacts properly
- Gracefully handles missing file / fail to open
- Opens the file and uses it
- Closes the file when done

The Usual

- Readme
- One job comments

No Global Variables! Code Must Compile!

Global variables is a -10 penalty. Errors or warnings in compilation means no credit for the lab, though you might get 2 points if your prototypes are there and compile cleanly.

Submission

Effectively: Same as before. Your zip file needs to contain README_LAB4, **all** the code to be graded, and a makefile sufficient to build all of the targets that are to be graded. **Those targets include lab4** and at least 4 working prototypes and lab4u. Do not include any of your .o files (do include n2.o)or the lab4 executable. Any file you edited by hand (other than test data) probably needs to be included. It's nice if you include the libraries (*.a) in your zip.

Your lab is counted late if the makefile doesn't self-test the zip

All files that you edit by hand must have your name in them. Machine generated headers need your name in them.

Any code that comes from the instructor's lab 2 o 3 must retain proper attribution.

Defects in the submission can cause a zero score for the lab or worse.

README_LAB4 text:

THIS IS THE README FILE FOR LAB 4

BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE PERFORMED ALL OF THE WORK TO DETERMINE THE ANSWERS FOUND WITHIN THIS FILE MYSELF WITH NO ASSISTANCE FROM ANY PERSON OTHER THAN THE INSTRUCTOR OF THIS COURSE OR ONE OF OUR UNDERGRADUATE GRADERS.

The readme should contain your **name**, the number of **hours** you worked on the lab, and any comments you want to add. Of particular interest are what was hard or easy about the lab or places where programming reinforced what we went over in class.