CMPS 101 - Programming assignment 4 (version 2)
Finding Books (May 13, 2014)

Due 11:59 pm Friday May 23.

May 13, 2014

Main Changes:

- recommend generic list option
- revised diagram
- corrected assignment number in submission instructions
- allow g++ compiler

1 Introduction

Libraries have a large number of books. In this assignment you will write a program that quickly looks up which libraries currently have a particular book. This assignment will use nested ADTs, where the data stored in ADT includes handles for another type of ADT.

For this assignment you are encouraged to work in groups of two, but not with a previous programming partner. Partnerships must rotate – you may not use the same partner as a previous written or programming assignment. If you work in a group of 2, both student's are responsible for ensuring that both partners completely understand the code and constructs used. Every file should begin with a block comment indicating who the partners are (names and UCSC accounts) and *any* help received on that part of the program. Significant outside help must also be acknowledged in the README file. Only one partner should submit the zip file described below, the other partner should watch the submission to ensure it is done correctly and on time. The partner not submitting the program should instead just submit a copy of the partnership's README file (*not* zip'ed).

The goals of this assignment include:

- Compose several levels of abstract data types.
- Use both keys and data in your ADTs.
- Gain practice with hashing.

This assignment has many pieces, and ensuring you understand everything early will be essential.

2 Program Specifications

Your program is to read in and process a data file (given as the first command line argument, use `argc` and `argv` in C) and then read in a act on a series of data requests in a second file (given as the second command line argument) and answer the data requests to the standard output. You may use either ANSI C, C-99 or C++ for this assignment. However, if you use C++ you must implement all of the ADTs "by hand" as part of your assignment rather than using libraries for the linked list or hash table modules.

The first input file will consist of a number of lines formatted as follows:

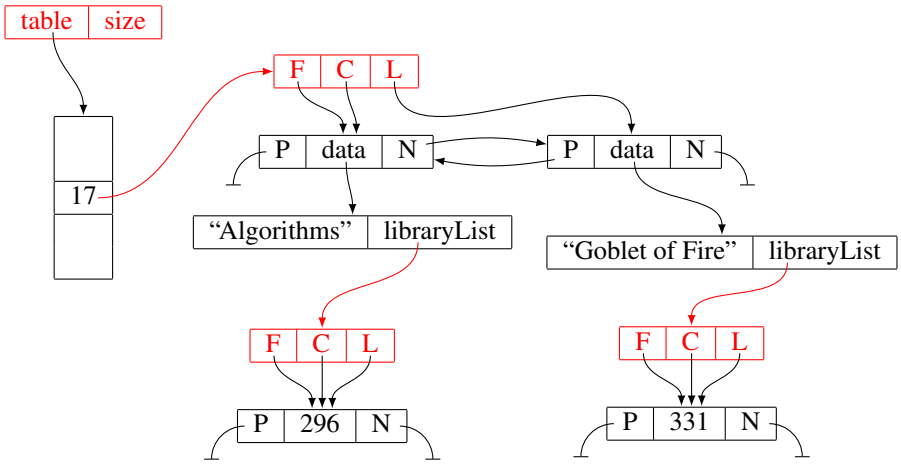
- The first line will give the number of data lines in the file (not counting this first line) and the size (number of slots) to use for your hash table.
- Each remaining line represents a library having a book, and will contain two values separated by a single comma (“,”)
  - The first value is an integer library ID
  - The remainder of the line is a book title (you can assume that book titles will be at most 40 characters long).

The second input file will contain a number of lines representing book requests. Each line will contain (only) the title of a book.

Your program should read the first file and initialize a data structure as described below. Then it should read the second file, and for each line, look up the requested book, and print out the libraries (in any order) that have the requested book.

3 Program Design – top down description

The program should keep track of which libraries contain a given book using a hash table. The hash function should map the book title to an index, see `http://www.cse.yorku.ca/~oz/hash.html` for some example hash functions for strings (note that these functions compute large integers – you still need to take modulo the hash table size). **Use chaining for handling collisions**, so each bucket in the hash table will be a list of data elements, and each data element will be the key (book title) and a list of libraries having the book. Thus the data structure might look like the following when both “Algorithms” and ”Goblet of Fire” hash to slot 17, and “Algorithms” is held by library ID 296 (only) and (only) library ID 331 contains “Goblet of Fire”. The handles and header records are in red.



This means you will need two kinds of linked lists: Linked lists of integers (as part of the data elements) and linked lists of data elements (for the hash table buckets). You can implement this with either two copies of your linked list module (with functions appropriately re-named, not recommended) or with one generic linked list module keeping a linked list of generic (`void*`) pointers. With the recommended second implementation, you must be very careful to only insert the correct kinds of things on each list. Note also that with linked list of generic pointers implementation, the above diagram is slightly incorrect: the list nodes at the bottom should contain pointers to library IDs rather than the library IDs themselves.

4 Implementation Details - bottom up

You should implement your program with a number of modules using the ADT methodology. Get started early and implement the modules in a bottom-up way.

The recommended option is to implement a generic linked list (i.e. a list of generic `void *` pointers) where the list users can put whatever pointers they want on to the list. Allocating the structs that these pointers point to and keeping the lists straight (so that the wrong kind of pointer doesn't get inserted into the list) becomes the list user's responsibility. This option creates a single very flexible linked list module that would be used by all the linked lists in the assignment.

A second option is to create specialized linked list modules for each kind of linked list in the assignment. Thus you would have a `libraryList.c` module with its own header file for keeping track of lists of libraries (integers) and a `bookLibrariesList.c` module (again with its own header file) for keeping track of book-libraryList pairs. This module would probably have separate access functions to get the book title and the handle for the linked list of libraries. Alternatively, you could view it as a list of structs where each struct has a book-title field and a library-list-handle field.

1. At the bottom level, you will need lists of libraries (ints). This could be very much like your program 1 list module. These lists will be controlled and manipulated not by the hash table, but by the clients of the hash table. (The hash table is providing the client with a way to look up books and retrieve the appropriate list.)

2. You will need to implement a hash table to store the library lists for the various books. The key will be the book, and the data will be the handle for a library list. When a key is inserted into the hash table, the user must provide a valid and initialized handle for a list of libraries. When a key is looked up in the hash table, the hash table should return a pointer; either null if the book is not in the table, or the listHandle[1] stored with the book.

As you read the entries in the library-book pairs in the first data file, you must lookup the indicated book in your hash table. If the book is already there, then add the libraryID to the associated list of libraries. If the book is not yet in the hash table, then create a new (empty) list of libraries, put the library on the list, and insert the book with the newly created list into the hash table.

Once you have processed the data file, you are ready to process the requests. After reading a request, look up the book in your hash table. If it is there, print out the list of library-IDs that have the book. If it is not there, then print an appropriate message.

5 Submission

As with program 3, zip up all of your files (including drivers, Makefile, and README) into a file named *lastName-FirstInitial*`Prog4.zip` and submit it through E-commons. By all your files, I mean: the main program for the assignment, your Makefile, and your README. You should also submit the .h files, .c files, and drivers for all of your modules that are needed to run your main program. Your Makefile should compile all of the .c files and create executables for the main program and each of the drivers. FIles used by other programs (such as program 1) should not be submitted unless they are also needed by this program.

I have specified the design a little less strictly than in previous assignments, but all exported functions should be described (complete with pre- and post- conditions) in the appropriate `.h` file. I expect each submission will contain at least one list module (and possibly two or three), a hash table module, drivers and header files for each module, and a main program. Feel free to break the problem into additional modules if it makes things easer.

As always, you will need to make sure that the programs compile and run correctly on the UCSC unix systems. You will lose points for compile issues, and should not expect that graders will take much time trying to fix them. If the program does not compile, graders may choose to grade the code on non-execution aspects and this will significantly cost your grade.

[1] Note that the hash table only stores (copies of) list handles. Although the list handles will stay the same, the state (number of elements and their counts) of the lists will change during the program's execution.

5.1 Grading Guide - check back for possible changes

15 points maximum:

- 1 point for correct submission with all files (including driver programs, README, and Makefile).
- 1 point for using separate ADT Files with information hiding and good descriptions of the exported functions including pre- and post- conditions clearly specified
- 1 point for well organized, commented, readable code and generally good style,
- 2 points for *a good, self-contained* Makefile (i.e. not using 12b scripts that the grader may not have access to).
- 1 point for either a generic linked list or two separate linked lists (with associated driver(s)).
- 2 points for a working hash table (as evidenced by the hash table driver)
- 7 points for correctly listing the libraries containing the desired books (using the required design)

Bonus points: These can be used to make-up lost points, but cannot increase your assignment total above 10.

- 1 point for good C memory management (each module frees the memory it allocates)

6 Notes and Hints:

- There is a lot to do in this assignment so **get started early**.
- Visualize the program top-down to understand what needs to be done, but implement bottom-up, using a driver to check that each small part of the implementation is working well before moving on to the next small chunk.
- One of the first chunks to write for each module is a print function that will help check correctness and aid debugging.
- Advanced students might find it interesting to time your implementation on a large file with various hash table sizes. If using re-hashing you might experiment with different rehashing schemes and how full to let the table get before doubling the size.