First programming project: Key-Word indexer

1 Indexing books by key words

For this assignment, you will create a keyword reference index and answer queries from this index. Each query will be a keyword (e.g., "Congress") or keyword phrase (e.g., "pathological liars"). The program should return the the bibliographic reference of every work (if any) which has that keyword/keyword phrase.

The program will need to be able to add and delete bibliographic references, answer queries, and print out either the list of keywords or the list of books in alphabetical order (by author in the case of books).

Your program will take a single command-line argument, which will give the name of an input file.

Your program will write output to the terminal.

You must use turnin and provide us with a makefile. Your makefile can specify either the CC or the g++ compiler.

The input file will contain the following commands:

- ADD n
  Read the following line, which contains a bibliographic reference in the format (title; author; year) and the n lines after that, each of which is a keyword for that reference. The keywords should be added to the table of keywords (stored in a binary search tree) if they aren't already there. Also, you'll want somehow to remember that the keyword refers to this reference.

  Note that bibliographic references and keyword phrases may be arbitrarily long. You will have no difficulties with this as long as you use the standard C++ type `string` and avoid the C type `char*`.

- FIND
  Read the next line, which contains a keyword, and print out the bibliographic reference(s), if any, for it. If there are no such references, print a message saying so.

- LIST
  Print out all the keywords in the table in alphabetical order.

- BOOKS
  Print out all the references in alphabetical order.

- DELETE
  Read the next line, and delete the work with that author.

- QUIT
  End the program.

2 Data structures

Both the keywords and the book data should be stored in binary search trees. You should build your own binary search tree class, because as explained in the next paragraph, you will need some special data members that are not in most binary search tree classes. You may use the class from the text, or some other source as a starting point if you wish. **If you do this, please document in your comments that you based your work on some existing class** and cite a source (either a book or a URL.

You should have only one binary search tree class.

In order to save storage space, we will have only one entry per book or keyword in each tree, no matter how many keywords reference a given book or vice versa. Therefore, each element in the tree of keywords will contain a linked list of ref-pointer nodes (or a pointer to the linked list if you think that's better design.) (Note that in this linked list, the "data" field will contain a *pointer*.)

Each ref-pointer will point to the appropriate element in the tree storing bibliographic references. This allows many keywords to point to the same bibliographic reference. (In a real world system, bibliographic references might be *much* longer than keywords.)

Similarly, the entries for books will contain a pointer to a linked list of pointers to keyword elements. (You'll need this information when you delete a book.)

2.1 Using standard String and List classes

You are encouraged to use the STL linked list class; you are not required to do so. (However, I would be giving slightly more time for this assignment were I not relying on the fact that you can use the linked list class and the string class.)

Here are a few quick examples of using the STL list class. For this example, I am assuming a list of `char`'s.

```
#include <list>

list<char> myList;
list<char>::iterator currentPos;

currentPos = myList.begin();
myList.insert(currentPos, 'o');
myList.insert(currentPos, 'f');     // List is now f, o
```

The iterator can be moved forward or backward using `++` and `--`. The iterator's begin() method gives the position of the front of the list; its end() method gives a position *immediately after* the end of the list.

The list class has a method erase whose argument is an iterator that gives the position at which to delete a list element.

2.2 Miscellaneous program requirements

You must not "leak memory." That is, when an item is deleted, you need to be careful to free all of its binary tree node, and anything else you may have allocated d dynamically with `new`.

(Which may not be anything else if you use the STL String and List classes, depending on implantation choices you make.)

2.3 Documentation

For this program, 65% of the grade will depend on correct functionality.

The remaining 35% will be based on a correctly functioning makefile, adherence to the course coding standards, and general design quality.

Here is some sample input. We probably won't use this particular input file for grading; it's just an example.

```
ADD 7
Godel, Escher, Bach; Hofstader; 1980
undecidability
artificial intelligence
knowledge
fugues
puzzles
art
music
ADD 2
Prince of the Blood; Feist; 1989
fantasy
adventure
FIND
puzzles
ADD 4
Computational Learning Theory: New Models and Algorithms; Sloan; 1989
machine learning
artificial intelligence
concept learning
noise
FIND
artificial intelligence
FIND
neurons
LIST
DELETE
Feist
BOOKS
QUIT
```

3 Submission

Deadline: Your code must be electronically submitted using the turnin program on the departmental machines by *10 p.m. Thursday, September 12*.

You must provide us with a working makefile.

Partial credit for unfinished work (that compiles!) will be given; no credit will be given for late work.