

Name: _____

Submission file name: _____

(Fill in the above information on a hardcopy of this document and submit it at the beginning of class on the due date. See below for details.)

This assignment asks you to build a small (toy) database application for storing information about books loaned by a university library. The goal is to gain experience using an application programming interface (API) to Oracle and PostgreSQL. Transforming the following specification into an application and learning all the required details (e.g., linking libraries, Makefiles) are important parts of this assignment.

As described in detail in the packaging instructions below, your submission should produce two executable files. The first, called `libO` should implement this application using Oracle to store data. The second, called `libP`, should be identical in behavior to `libO`; however, it should use PostgreSQL to store data. You may use either the C/C++ interface described in Chapter 3 of the Oracle8 textbook, or a Java interface (JDBC or SQLJ). You may also use any other interface of your choice (e.g., Perl DBI); however, you'll have to install any additional software you need yourself. Reminder: Please start working on this assignment early; use the newsgroup for questions and clarifications.

Since we haven't studied schema design methods yet, we will, for now, store all the necessary information in a single table called *BookLoans*. This table should have one column for each of the following attributes:

- student's name
- student's address
- student's phone number
- student's account number
- book's title
- book's author
- book's publisher
- date and time the book was loaned
- due date and time

The semantics for the columns should be evident from their names. Pick suitable types for each column. Note that the loan date and time is a single attribute, as is the return date and time. The types you choose should be neither too restrictive nor too permissive. For

example, do not assume that phone numbers are in U.S. standard form; they may be any alphanumeric string (e.g., 1-800-GET-RICH, 1.91.22.555.HELP). On the other hand, do not store dates and times as strings.

You must implement your application program as a Unix command-line program that reads from standard input and writes to standard output. This application must implement the user functions described below. When the work (both internal processing and output to user) for each function is done, your application should write (to standard output) five dashes (-----) followed by a single newline character. We will refer to this string of five dashes followed by a newline as the *function termination string*. The following description also refers to a *separator string*, which consists of the three character sequence space-colon-space.

These functions will be invoked from standard input by listing the function name followed by its arguments, one per line. For example, the *connect* function described below takes two arguments and may be invoked as follows (using example values for the arguments):

```
connect
sc42401
xyzzzy
```

String arguments will be listed verbatim, with no quotes or other demarcation. You may assume that function arguments do not contain any newline characters. Numeric data will be listed in a format 123.45. (That is, numbers are rounded to two places after the decimal point and there are as many digits before the decimal point as are needed, with no 0-padding.) You may assume that all numbers are in the range $[-10,000 \dots 10,000]$, with at most two digits after the decimal point. Date-time values are in the format YYYY-MM-DD HH-MM-SS. For example, 2001-05-04 14-01-03 denotes three seconds past 2:01pm on the 4th of May, 2001.

The input will contain, in general, several function calls in the above format, listed one after the other. Your program should ignore lines with # (pound sign) as the first character. It should also ignore blank lines, but blank lines separating function invocations are *not* required. Since you know the number of arguments each function takes, there is no need for such separation. (Note that the function termination string is used only for output, not in the input.) Your application should read and process the functions in the order in which they appear in the input and should terminate gracefully (e.g., by closing open database connections) when the end of input is reached. There is no special end-of-input marker.

connect(foo, bar): This function will be the first one invoked in any test run, and it will be invoked exactly once per run. In response, your application should perform all necessary initialization and connect to the Oracle server as user `foo` using password `bar`.

We will test your program using a temporary account `foo` that is *not* your class account. You may assume that the database for account `foo` initially contains no user tables. Make sure you do not assume anything specific to your own class account. For example, you cannot rely on any initialization you have in your `.login` or `.tcshrc` files, since these files will not be the same for the test account. Please be sure to understand the implications of this requirement. Creating code that can be easily run by someone else is an important part of this homework. For testing, you should use your own account name and password

in place of `foo` and `bar`. (Hint: You should test your submission by temporarily replacing your customized account files, if any, with the default ones that came with your account.)

createTable() This function should result in the creation of the BookLoans table described above. This function will be called before any of the functions below.

destroyTable() This function should result in the BookLoans table and all its contents being destroyed. The database should now be in its initial pristine state (with no user tables). You may assume that after this function is called, a call to `createTable` will precede a call to any of the functions described below.

add($a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9$): When this function is invoked, your application should add a tuple ($a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9$) to the BookLoans table (where the columns are listed in the order in which they were described earlier). You should check for duplicates; that is, if the tuple denoted by an `add` function invocation already exists in the BookLoans table, your program should not add a duplicate and should *not* flag an error.

searchByStudent(`foo`): This function should search for student names that contain the substring `foo`. This search, and all searches on string attributes, should be case-insensitive unless specified otherwise. The matching names should be printed one per line, sorted in ascending lexicographic order. On each line, the sender name should be followed by the separator string (described earlier), in turn followed by a *unique identifier*, called the *SID* (described below). Output lines here and elsewhere should be terminated by a single newline character. If a student has borrowed more than one books, their name should appear once, and after the separation string there should be a comma-separated list of SIDs, that correspond to the loans this student has done.

details(`foo`): This function should print all the information for the record identified by the SID `foo` (*exact*, case-sensitive string match) on a single line. If there is no record with SID `foo`, no output should be produced. The output (if nonempty) should print the attributes in the order they were originally described.

For this and other functions, attributes values and other items printed on an output line should be separated using the separator string. Strings should be printed literally (with no quotes, padding, or other artifacts). Numbers should be printed in the form \$123.45 (as in the input format), with an optional prefix of “-” to denote negative values. Date-and-time attributes should be printed in the input format described above.

Note on SIDs: SIDs are identifiers (generated by your program) that uniquely identify a record in the BookLoans table. You are responsible for generating and managing these identifiers. Once you have exposed a SID (by printing it as output), this SID may be presented as an argument of the `details` function at any point in the future. SIDs must persist between sessions. For example, if your program exposes a SID 192 during one session (say, in the output of the `searchByStudent` function), a `details` function call with 192 as

the argument must produce details of the record identified by this SID. Unless this record has been deleted or otherwise modified in the interim, the output of this details function invocation should be the same as if it had been invoked in the original session. In short, SIDs should be persistent and should uniquely identify a book loan record.

searchByBookTitle(foo): This function should search for records with book titles that contain the substring foo. The output should be in a format similar to that used for the searchByStudent function, except for the fact that you should print all records that match, no matter if they are duplicates or not. For each match, there should be a line of output consisting of the matching address (in its entirety) and the corresponding SID. The output records should be sorted in ascending lexicographic order by address.

searchByStudentPhone(foo): This function should search for records with student phone numbers that contain the substring foo. For each match, there should be a line of output consisting of the matching phone number and the corresponding SID. The output records should be sorted in ascending lexicographic order by phone.

searchByStudentAcct(foo): This function should search for records with student account numbers that have foo as a *prefix* (case insensitive). For each match, there should be a line of output consisting of the matching account number and the corresponding SID. The output records should be sorted in ascending lexicographic order by account number.

searchBy... You should implement two more functions, **searchByAuthor**, **searchByPublisher** that perform the corresponding searches in the same way that searchByBookTitle works (duplicates reported, substring, case insensitive.)

searchByLoanTimeRange(foo, bar): This function searches for records with loan date-times no earlier than foo and no earlier than bar. The arguments foo and bar are date-time specifications in the format described earlier. For each matching record, there should be a line of output that lists the matching loaning date-time and the SID of the corresponding record.

searchByLoanTimeExpr(foo) This function performs a wildcard-based matching on the loaning date-time field. The search parameter (foo) will be of the form YYYY-MM-DD HH-MM-SS, which is the format we've described earlier for date-time fields. However, in this function zero or more of the components (YYYY, MM, DD, HH, MM, and SS) may be the special wildcard *, denoting a "don't care." Thus, searching for *-01-2001 02-*- should list records with a pickup date-time that is between 2:00:00pm and 2:59:59pm (inclusive) on some day in January 2000.

searchByDueTimeRange and searchByDueTimeExpr: These two functions are analogous to searchByLoanTimeRange and searchByLoanTimeExpr, respectively, searching the due date-time field instead of the loan date-time field.

listDelayed(): This function should search for records where the due date-time has passed. For each match, there should be a line of output consisting of the matching due date and the corresponding SID. The output records should be sorted in ascending order by due date-time (i.e. chronologically.)

updateDueTime(foo, bar): This function should assign the due date-time bar to the record with SID foo. If such a record does not exist, the function should not perform any update (but *not* flag an error). This function should produce no output other than the function termination string.

extendDueTime(foo, bar): This function should add bar days to the due date-time of the record with SID foo. If such a record does not exist, the function should not perform any update (but *not* flag an error). This function should produce no output other than the function termination string.

deleteRecord(foo) This function should delete the record with SID foo. If such a record does not exist, the function should not perform any database modifications (but *not* flag an error). This function should produce no output other than the function termination string.

Submission: The submission procedure (including file naming convention) is similar to the one used for the first homework, PHW01.

Packaging You must submit a gzipped tar file containing the source files (*not* object files or machine code) required to compile and run your program. The file should be named `foo.tar.gz` (where foo is replaced with something like HendrixJM-1101, as described in PHW01). Unzipping and untarring `foo.tar.gz` should result in the creation of a single directory (in the current working directory) called `phw02`. Typing *make* at the Unix shell prompt in the `phw02` directory should result in the complete compilation of your program, producing two executable files (machine code, shell script, Perl script, etc.) called `lib0` and `libP`, for the Oracle and PostgreSQL implementations, respectively. Obviously, you will need to include a Makefile in the `phw02` directory. You should also include a short README file describing the files in your submission. This README file is a fallback. If your program does not work perfectly, we will look at the README file and *if it is well written and includes some special instructions* we will try to get your program working by following these instructions.

Please test very carefully that this unpacking and compilation procedure works with your submission. Your score will suffer greatly if it does not, or if your submission contains object files or machine code. (If you use Java, submit the `.java` files, not the `.class` files; your makefile should be designed to produce the `.class` files. The make procedure should also result in a executable files that run the Oracle and PostgreSQL versions of the application, perhaps by calling “java classname.”) Recap: The sequence of commands `gunzip foo.tar.gz; tar xf foo.tar; cd phw02; make` should result in the final executables `lib0` and `libP`.

Test Input You may wish to use this sample input to test your program by replacing `dummyAcct` and `dummyPassword` with your own account name and password. (You should test your work thoroughly by generating test inputs that exercise all the functions above.) Note that spaces are significant in string arguments (e.g., passwords, comments) and should not be ignored or modified. For clarity, the following uses `␣` to denote the space character. There is a newline character at the end of each input line.

```
connect
dummyAccount
#␣This␣line␣should␣be␣ignored
dummyPassword
createTable
#␣We␣may␣destroy␣and␣create␣the␣table␣repeatedly...
destroyTable
createTable

#␣The␣above␣blank␣line␣and␣the␣one␣following␣add␣should␣be␣ignored.
add

John␣N.␣Doe
1600␣K␣St␣NW,␣Suite␣101,␣Washington,␣DC.
202.111.2323
DC01X29a
The␣Art␣of␣Computer␣Programming␣-␣Sorting␣and␣Searching
Donald␣Knuth
Springer
2002-09-12␣17-05-01
2002-09-15␣15-00-00

searchByLoanTimeRange
2001-09-02␣00-00-01
2003-09-01␣23-59-59

details
X1438a
```

Test Output On the above input, your program should produce the following output. The SID 1438 is arbitrary; your program may produce a different identifier. All the text between the last two function termination lines (-----) below is a single line (which has been wrapped below in order to fit on this page).

```
-----
-----
-----
-----
```

2002-09-01_17-05-01:_X1438a

John_N._Doe:_1600_K_St_NW,_Suite_101,_Washington,_DC._:
_202.111.2323:_DC01X29a._:
_The_Art_of_Computer_Programming_-_Sorting_and_Searching._:
_Donald_Knuth:_Springer._:2002-09-12_17-05-01._:
_2002-09-15_15-00-00

Note: For the previous input, even when the input ends (Ctrl-D, or EOF), the program should disconnect from the database but *not* destroy the table, unless explicitly asked. Therefore, any subsequent execution of the program is supposed to start with an existing table, which can be altered, extended, destroyed, searched etc.