Name: ______________________________________

Goal: This assignment asks you to build a small database application for managing an online silent auction such as the one described in the previous assignment. The main goal of this assignment is to gain experience using the application programming interfaces (API) to Oracle and PostgreSQL, understanding the common and differing parts. Secondary goals include practice in writing SQL queries and a study of the impact of database design on ease of querying and updating data, and on maintaining database consistency. (Hopefully, this homework will provide a concrete motivation for these topics, which we will study soon.)

The Programming Environment (DBMS libraries): An important part of this assignment is learning the interface between a typical programming environment and the database system. You are free to use a programming language and database interface library of your choice. However, only C (with embedded or dynamic SQL) and Java (with JDBC) are supported. While we will try to help you with other languages and libraries, please note that it is not usually possible to modify software (especially the database system libraries) on the OIT and CSIC machines in the middle of the semester. Figuring out the details of the database system interface and the necessary libraries usually takes people a lot longer than they expect, so please start working on at least this part early! Please use the class newsgroup (and not email) for questions and other discussions.

Database Tables: The application uses the two database tables AuctionItems and Bids from the previous assignment. Unless otherwise specified, you may assume that all string-valued attributes contain at most 100 characters. (An exception is the longer description attribute of AuctionItems.)

The Application Programs: As described further in the packaging instructions below, your submission should produce two executable files. The first, called aucdbo (auction database in Oracle) should implement this application using Oracle to manage the data. The second, called aucdbp, should be identical in behavior to aucdbo; however, it should use PostgreSQL to manage the data.

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 singlenewline 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. Except the output described in this homework, your program should not produce any extra output such as diagnostic messages. (Before submission, please remember to remove any such messages that you use for debugging your code.)

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
xyzzy
```

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 or 123.45. (That is, integers are listed in common notation and floating point numbers are rounded to two places after the decimal point. 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 [0..100.000], with at most two digits after the decimal point. Date-time values are in the format YYYY-MM-DD HH:MM:SS. For example, 2005-05-04 14:01:03 denotes three seconds past 2:01 p.m. on the 4th of May, 2005.

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. You do not need to provide any error-handling features; your program will only be tested on valid input.

**Functions:** The functions that your program should implement are described below. Note that the descriptions use a conventional functional notation of the form $f(a, b)$, but the input is presented in the form described above.

**connect($u$, $p$):** 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 database server as user $u$ using password $p$. (Strictly speaking, your program need not perform any of these actions, since its observable behavior for this function does not depend on them. However, it is probably a good idea.)

We will test your program using a temporary account $u$ that is not your class account. You may assume that the database for account $u$ 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
u and p. (You may wish to test your submission by temporarily replacing your customized
account files, if any, with the default ones that came with your account.)

createTables(): This function should result in the creation of the AuctionItems and Bids
tables described above. This function will be called before any of the functions below. It
does not return any results.

destroyTables(): This function should result in the AuctionItems and Bids tables 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. This function does not return any results.

addItem(b, l, p, s, d): When this function is invoked, your application should add a tuple
(i, b, l, p, s, d) to the AuctionItems table, where the attribute values are listed in the order
the attributes were described in the previous assignment and where i is an identifier of your
program’s choosing. (See the note on identifiers below.) This function does not return any
results.

findItems(s): This function should search for auction items whose brief or long descrip-
tions include s as a substring. This search, and all searches on string-valued attributes,
should be case-insensitive unless specified otherwise. The matching auction-item records
should be printed one per line, sorted in ascending order of the items’ auction-end date-
times. On each line, the brief description should be followed by the separator string (de-
scribed earlier), in turn followed by the item’s identifier. Output lines here and elsewhere
should be terminated by a single newline character.

describeItem(i): This function should print all the information for the auction item iden-
tified by the given identifier i (exact, case-sensitive string match) on a single line. If there
is no record with identifier i, no output should be produced (and this condition is not an
error). The output (if nonempty) should present the attributes of a product record in the
order they were described in the previous assignment.

For this and other functions, attribute 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). Dollar amounts should be printed in the form $123.45.
Integers, reals, and dates should be printed in the format used for the input.
addBid\((b, \ i, \ a)\): When this function is invoked, your application should add a tuple \((c, b, i, a, t)\) to the Bids table, where the attribute values are listed in the order the attributes were introduced in the previous assignment, where \(t\) is a timestamp corresponding to the time of function invocation, and where \(c\) is a unique identifier generated by your program. (See the note on identifiers below.) The output of this function is a single line with the identifier \(c\).

**Note on Identifiers:** The identifiers generated by your program in response to the addItem and addBid functions must uniquely identify auction items and bids, respectively. You are responsible for generating and managing these identifiers. Once you have exposed an auction item’s identifier (by printing it as output), the identifier may be presented as an argument of the describeItem function at any point in the future. These identifiers must persist between sessions. For example, if your program exposes an auction item identifier xy101 during one session a describeItem function call with xy101 as the argument must produce details of the corresponding record. Unless this record has been deleted or otherwise modified in the interim, the output of this describeItem function invocation should be the same as if it had been invoked in the original session. All matching for identifiers should be exact. (If you use strings as identifiers, the match should be case-sensitive, exact string match, for example.)

findAvailItems\((s)\): This function differs from findItems in only one way: Only auction items whose auction-end times are in the future (with respect to the time of function invocation) should be returned.

showResults(): This function should produce output indicating the result of all auctions that have completed (whose end time is in the past). The output consists of one line for each such item, sorted in descending order of end time. Each line should contain the item identifier, its brief description, its starting price, the name of the winning bidder (the string -- in case of no bids), and the amount of the winning bid (-- in case of no bids), all separated using the separator string.

showResults\((p)\): This function differs from showResults in only one way: Only auction items whose auction end times match the time-pattern \(p\) should be returned. The matching is based on interpreting \(p\) as a crontab\(^1\) expression. The expression \(p\) consists of five fields (minute, hour, day of month, month, and day of week) separated by whitespace. The following excerpt (slightly edited) from the crontab manual describes the semantics.

> [A timestamp matches \(p\)] when the minute, hour, and month of year fields match the current time, and when at least one of the two day fields (day of month, or day of week) match the current time. The time and date fields are:

A field may be an asterisk (*), which always stands for “first–last.”

Ranges of numbers are allowed. Ranges are two numbers separated with a hyphen. The specified range is inclusive. For example, 8–11 for an “hours” entry specifies [a match] at hours 8, 9, 10 and 11.

Lists are allowed. A list is a set of numbers (or ranges) separated by commas. Examples: “1,2,5,9”; “0–4,8–12.”

Step values can be used in conjunction with ranges. Following a range with “/<number>” specifies skips of the number’s value through the range. For example, “0–23/2” can be used in the hours field to specify command execution every other hour (the alternative in the V7 standard is “0,2,4,6,8,10,12,14,16,18,20,22”). Steps are also permitted after an asterisk, so if you want to say “every two hours,” just use “*/2.”

Names can also be used for the “month” and “day of week” fields. Use the first three letters of the particular day or month (case doesn’t matter). Ranges or lists of names are not allowed.

**showHotItems(m):** This function should produce output summarizing the ten auction items that have had the most bids in the last $m$ minutes, sorted in descending order of the number of bids. Ties are to be broken by auction end times (earlier end times are preferred). Remaining ties are broken arbitrarily. (This function should never return more than ten items.) The output consists of one line per item, listing the item’s brief description, start price, highest current bid, and the number of bids on that item in the last $m$ minutes, all separated using the separator string.

**showBidDistr(m):** This function should produce a histogram of the distribution of number of bids on items in the last $m$ minutes. More precisely, the output consists of one line for each histogram bucket. On each line, there are three numbers, $l$, $u$, and $n$ (separated by the separator string), denoting a histogram bucket with range $[l, u)$ containing $n$ bids. There should be exactly 20 histogram buckets of equal range-size ($u - l$) that cover all the data.

**showBidDistrAlt(m):** This function differs from showBidDistr in only one way: Instead of the histogram buckets all having the same range-size, they are required to contain nearly identical number of bids ($n$). More precisely, if $n_1 \ldots n_{20}$ denote the number of bids in each of the 20 histogram buckets, then for all $i, j \in [1..20]$ we require $|n_i - n_j| \leq 1$. 

<table>
<thead>
<tr>
<th>field</th>
<th>allowed values</th>
</tr>
</thead>
<tbody>
<tr>
<td>minute</td>
<td>0–59</td>
</tr>
<tr>
<td>hour</td>
<td>0–23</td>
</tr>
<tr>
<td>day of month</td>
<td>1–31</td>
</tr>
<tr>
<td>month</td>
<td>1–12 (or names, see below)</td>
</tr>
<tr>
<td>day of week</td>
<td>0–7 (0 or 7 is Sun, or use names)</td>
</tr>
</tbody>
</table>
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 previous assignment). 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 aucdbo and aucdbp, 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 fall-back. 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 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 aucdbo and aucdbp. You must make sure your program works with redirecting input. For example, we may run your program by the command aucdbo < datafile, where datafile is a text file contains the input of the program.