

Name: _____

Goal: This assignment asks you to build a small database application for storing information about cardboard boxes used by a small warehousing, packing, and shipping store called *FooPS*. This store maintains a small inventory of products for its customers and, on receiving instructions from the customers, packs and ships their products. 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 two database tables: **Boxes** and **Products**. The **Boxes** table stores information about types of cardboard boxes used for packing, and has the following columns:

1. **name:** This column stores a human-readable name for the box type. Names identify boxes uniquely.
2. **width, depth, and height:** These three columns store the size of the box, in inches.
3. **load:** This column stores the maximum permissible weight of the box's contents (in pounds weight).
4. **color:** This column stores a human-readable description of the box's color.
5. **cprice** and **sprice:** These columns are used to store the cost price (price paid by FooPS) and selling price (price charged to FooPS customer) of a box, respectively.

6. **num**: This column stores the number of boxes of this type available at the store (i.e., inventory).

The `Products` table has the following columns:

1. **PID**: a unique identifier for each product. This identifier is *not* provided by the customer. Instead, it should be generated by your program. (Details below.)
2. **name**: the name of the product.
3. **customer**: the name of the customer for whom this product is being stored and shipped.
4. **cphone**: the customer's contact phone number. You should pick a datatype that allows you to store sufficiently general phone numbers, not just 10-digit ones. For example, your application should be able to manage numbers such as 11 91 22 55 55 12 12 and 1-800-BIG-BOXS.
5. **description**: a description of the product. You may assume that the description is at most 10000 characters long.
6. **weight**: the weight of the product, in pounds.
7. **width, depth, and height**: These three columns store the size of the product, in inches. (If the products is not box-shaped, these dimensions refer to its bounding box.)

Unless otherwise specified, you may assume that all string-valued attributes contain at most 100 characters.

The Application Programs: As described further in the packaging instructions below, your submission should produce two executable files. The first, called `foodbo` (FooPS Data Base in Oracle) should implement this application using Oracle to manage the data. The second, called `foodbp`, should be identical in behavior to `foodbo`; 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 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. 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 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
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 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 $[0 \dots 100,000]$, with at most two digits after the decimal point.

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 `Boxes` and `Products` 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 **Boxes** and **Products** 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.

addBox($n, w, d, h, l, c, b, s, i$): When this function is invoked, your application should add a tuple $t_1 = (n, w, d, h, l, c, b, s, i)$ to the **Boxes** table, where the attribute values are listed in the order the attributes were introduced earlier. However, *duplicates*, in the following sense, should be avoided: If a tuple t_2 exists in **Boxes** with $t_1.n = t_2.n$, then instead of simply adding t_1 to the table, you should instead *replace* t_2 with t_1 . This function does not return any results.

addProduct(n, c, p, t, l, w, d, h): When this function is invoked, your application should add a tuple $t_1 = (i, n, c, t, l, w, d, h)$ to the **Products** table, where the attribute values are listed in the order the attributes were introduced earlier, and where i is a unique identifier generated by your program. (See the note on identifiers below.) Duplicates should be avoided in the following way: if a tuple t_2 exists in **Products** such that all attributes of t_1 , except the first (identifier), agree with those of t_2 , then instead of simply adding t_1 to the table, you should instead *replace* t_2 with t_1 . This function does not return any results.

searchProductByName(n): This function should search for products with a name that includes the *substring* n . This search, and all searches on string-valued attributes, should be *case-insensitive* unless specified otherwise. The matching product records should be printed one per line, sorted in ascending lexicographic order of names. On each line, the name should be followed by the separator string (described earlier), in turn followed by the PID of the picture. Output lines here and elsewhere should be terminated by a single newline character.

detailProduct(i): This function should print all the information for the product identified by the PID i (*exact, case-sensitive* string match) on a single line. If there is no record with PID 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 above.

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). Prices should be printed in the form \$123.45. Integers, reals, and dates should be printed in the format used for the input.

Note on Identifiers (PIDs): PIDs are identifiers (generated by your program) that uniquely identify records in the **Products** table. You are responsible for generating and managing these identifiers. Once you have exposed an identifier (by printing it as output), the identifier may be presented as an argument of the **detailProduct** function at any point in the future. These identifiers must persist between sessions. For example, if your program

exposes a PID 192 during one session (say, in the output of the `searchProductsByName` function), a `detailProduct` function call with 192 as the argument must produce details of the product identified by this PID. Unless this record has been deleted or otherwise modified in the interim, the output of this `detailProduct` function invocation should be the same as if it had been invoked in the original session. In short, PIDs should be persistent and should identify the product records (respectively) uniquely. All matching for identifiers should be exact. (If you use strings as identifiers, the match should be case-sensitive, exact string match, for example.)

detailBox(*n*): This function is to `Boxes` what `detailProduct` is to `Products`, with records identified using the given name *n*. (Thus, a case-sensitive, exact string match should be used.) Recall that names uniquely identify boxes.

searchBoxesByLoad(*l*₁, *l*₂): This function searches for boxes with load in [*l*₁, *l*₂]. Matching boxes are presented in ascending order of load values. For each such box, there is a line that lists its load followed by the box name.

searchBoxesForProduct(*i*): This function searches for up to 10 *best* boxes for packing the product identified by the PID *i*. A box can be used for packing a product if it is in stock at FooPS, the weight of the product is no greater than the box's load (permissible weight) and, the product fits in the box with 1 inch of extra space on each side (for padding). Among such boxes, we order boxes using their selling prices. Cheaper boxes are better. In case there are ties, we break them using the cost prices. Again, cheaper boxes are better. Any remaining ties are broken using a lexicographic ordering on the box names. The output of this function is a list of box names satisfying these conditions, sorted in descending order of goodness, in the above sense. If there are fewer than 10 such boxes, then they form the output. Otherwise, only the 10 best ones are listed.

searchProductsByPhone(*p*): This function searches for product records with phone numbers matching the given pattern *p*. The pattern language is as follows: All characters except the period (`.`), asterisk (`*`), and backslash (`\`) are interpreted literally, and match only the corresponding character in the phone number (with case-insensitive matches for letters). A period matches any one character in the phone number, while an asterisk matches zero or more consecutive characters. A backslash is used as an escape character to override the special meaning of the three special characters: A literal period, asterisk, and backslash are represented by `\.`, `*`, and `\\`, respectively. For each product that has a phone number matching the given pattern *p*, the output has a line listing the phone number followed by the PID. The output should be sorted by ascending lexicographic order of phone numbers.

searchProductByDesc(*t*): This function searches for products with a `description` attribute that matches the given pattern *t* (case insensitive substring match). For each match-

ing record, the output has a line listing the description followed by the PID.

useOneBox(n): This function models one box of the type named n being used for packing. It should result in the inventory (`num`) for the box-type n being decremented by one. This function produces no output.

setProductDesc(i, t): This function updates the description of the product with PID i to the given text t . You may assume that i is a valid PID exposed by your program earlier. (See the note on identifiers above.) This function produces no output.

setCustomerPhone(c, p): This function updates the phone number of the customer with name c to the given phone number p . This function produces no output.

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 `foodbo` and `foodbp`, 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 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 `foodbo` and `foodbp`. You must make sure you program works with redirecting input. For example, we may run your program by the command `foodbo < datafile`, where `datafile` is a text file contains the input of the program.

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
createTables
#␣We␣may␣destroy␣and␣create␣the␣table␣repeatedly...
destroyTables
createTables

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

rare␣original␣IBM␣PC
junk␣buyers,␣antique␣sellers
1-800-buy-junk
This␣is␣the␣gold␣standard␣of␣all␣computers.␣␣Don't␣miss...
#␣this␣line␣is␣ignored;␣the␣above␣may␣be␣quite␣long
50
30
35
10.25
searchProductsByPhone
1-800-...-JU*

detailProduct
4297
```

Test Output On the above input, your program should produce the following output. The PID 1121 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).

```
-----
-----
-----
-----
-----
1-800-by-junk␣:␣4297
-----
4297␣:
```

```
rare_original_IBM_PC:junk_buyers,antique_sellers:1-800-buy-junk...  
-----
```

In the penultimate line above, the ... denotes the listing of the rest of the fields of the matching record.

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 e.t.c.