

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 people's contact information. The goal is to gain experience using an application programming interface (API) to Oracle. You may use either the C/C++ interface described in Chapter 3 of the Oracle8 textbook, a Java interface (JDBC or SQLJ), or a Perl interface (DBI).

We will store all the necessary information in a single table called *Contacts*. This table should have one column for each of the following attributes: name, email address, home phone number, date of birth, money owed, and comment. Assume that for each person in the database, we have exactly one value for each of these attributes. The *money owed* attribute for a person should contain, in dollars and cents, the amount of money that person owes us (database owners). If we owe the person money, then the attribute should be negative. Pick suitable types for each column. 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).

You must implement your application program as a Unix command-line program called `contactapp` 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. No other output (except as specified below) should be produced.

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
sc42407
supersecret
```

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. You may ignore all but the two most significant digits following the decimal point. You may also assume that all numbers are in the range  $[-10,000 \dots 10,000]$ . Dates will be specified in the form DD-MM-YYYY.

The input will contain, in general, several function calls in this format, listed one after the other. Your program should ignore all blank lines, but there may not be any blank lines separating function invocations. (Since you know the number of arguments each function takes, there is no need for such separation). 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.

**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, so be sure not to assume anything account-specific. However, you should obviously test your application using your own account name and password in place of `foo` and `bar`. You may assume that the database for account `foo` initially contains no user tables.

**createTable()** This function should result in the creation of the `Contacts` table described above. It will be called only once during the entire testing process, immediately after the first call to `connect` and before calls to any of the functions below.

**add(foo, bar, baz, qux, quux, quuux):** When this function is invoked, your application should add a tuple (`foo, bar, baz, qux, quux, quuux`) to the `Contacts` table (where the columns are listed in the order `(name, email address, home phone number, date of birth, money owed, comment)`). You may assume that each such requested addition to the `Contacts` table will have a distinct name.

**details(foo):** This function should print all the information for the person with name `foo` (*exact* string match) on a single line. If there is no information about `foo`, no output should be produced. The output (if nonempty) should print the attributes in the order: `name, email address, home phone number, date of birth, money owed, comment`. All but the last attribute should be followed by a single tab character to separate it from the next one. The last attribute should be followed by a single newline character. Strings should be printed literally (with no quotes, padding, or other artifacts). Dollar amounts should be printed in the form `$123.45` (with an optional prefix of `-` to denote negative amounts). The date of birth should be printed in the format `DD-MM-YYYY`.

**searchName(foo):** This function should print a list of names that contain the substring `foo`. The names should be printed one per line, sorted in ascending (lexicographic) order. (Each line should contain (only) the string value of the name followed by a single newline character.) This search, and all searches on string attributes listed below, should be case-insensitive.

**searchEmail(foo):** This function should print a list of the names and email addresses of all people whose email address contains the substring `foo`. It should print one (email address, name) pair per line. On each line, it should print the email address followed by a single tab character followed by the name followed by a single newline character. The list should be sorted first (primary sort order) in ascending order of the domain part of the email address (e.g., `cs.umd.edu` for `testudo@cs.umd.edu`) and then (secondary sort order) in ascending order of the account name (e.g., `testudo`). You may assume that all email addresses will have a valid syntax of the form `account@domain`. (You need to assume only that neither account nor domain contains the `@` character.)

**searchPhone(foo):** This function is similar to the searchEmail function, but searches for a (substring) match for foo in the phone column. It should print the matches as a list of one (phone number, name) pair per line, with a single tab character separating the name from the phone number (and a single newline character terminating the line). The list should be sorted first in ascending order of phone numbers and then in ascending order of names.

**searchDOB(foo):** This function searches for people based on the date-of-birth (DOB). As with the searchPhone function, it should print a list of (DOB, name) pairs, sorted in ascending DOB (primary) and ascending name (secondary order), with one entry per line (using tab and newline as above). The search parameter (foo) will be of the form DD-MM-YYYY (e.g., 25-11-1995) denoting the day, month, and year components of the DOB. However, one or more of the components (DD, MM, and YYYY) may be the special wildcard \*, denoting a “don’t care.” Thus, searching for DOB \*-01-2000 should list tuples with a DOB that is in January 2000; similarly, searching for \*-04-\* should yield tuples with DOB in April.

**destroyTable()** This function should result in the Contacts 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 any function call other than connect.

**Submission:** Submission is by anonymous FTP to `ftp.cs.umd.edu`, directory `/incoming/chaw`. You must submit a gzipped tar file containing the source files (*not* object files or machine code) required to compile and run your program. You should name this file `foo.tar.gz`, replacing foo with your last name suffixed with initials (e.g., SmithJK). Unzipping and untarring `foo.tar.gz` should result in the creation of a single directory (in the current working directory) called `phw02`. Typing a `make` in the `phw02` directory should result in the complete compilation of your program, producing a final executable file (machine code, shell script, Perl script, etc.) called `contactapp`. 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. 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.<sup>1</sup> Recap: The sequence of commands `gunzip foo.tar.gz; tar xf foo.tar; cd phw02; make` should result in the final executable `contactapp`.

**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. 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`

---

<sup>1</sup>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 `contactapp` file that runs your application (perhaps by calling “java classname.”)

```

dummyAccount
dummyPassword
createTable
destroyTable
createTable

add
Jane_Q_Public
jane.q@aol.com
202.111.2323
13-12-1960
-20.12
I_should_send_Jane_some_money.
add
Jane
jane@foo.bar.edu
5-1212
12-12-1970
10
Huh?
details
Jane
searchName
public
searchDOB
*-*-1970

```

**Test Output** On the above input, your program should produce the following output. The seventh line below uses `\t` to denote a single tab character.

```

-----
-----
-----
-----
-----
-----
Jane\tjane@foo.bar.edu\tjane@foo.bar.edu\t5-1212\t12-12-1970\t10.00\tHuh?
-----
Jane_Q_Public
-----
Jane
-----

```