This programming assignment should be submitted electronically by following the instructions on page 7. You are welcome, and encouraged, to use any resources, such as Web sites, to help you with your work. However, *all such help must be clearly noted* in your submissions. Further, no matter what you use, *you must be able to explain* how and why it works.

For this homework, functions marked with $\star$ are required for COS 580 students only. The main programming task described below is worth 200 points. For COS 480 students, any points earned on $\star$ functions (potentially another 50 points) are simply added to the score. If the resulting score is greater than 200, the remaining points are extra credit. COS 580 students must implement all functions to yield a raw score (maximum 250) that will be scaled by 4/5 to obtain the score (maximum 200) on the homework. (COS 580 students who wish to work for extra credit should contact me for a suitable assignment.)

Goal: This assignment asks you to build a small database application for managing information related to trees. The main goal of this assignment is to gain experience using one or more application programming interfaces (APIs) to PostgreSQL. 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: 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, you should not assume any particulars without first checking with me. 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.

Database Tables: The application uses the two database tables `Trees` and `Places` from the previous assignment, with the following changes:

1. The `Trees` table should include a column named `features` that holds a textual description of distinguishing features of each tree, such as the following description from the Maine Tree Club[1]: "The Pitch Pine is the only 3-needled pine found in the North. Needles are usually twisted and grow at right angles to the branchlets. Needles can also grow in tufts from the trunk."

[1]Maine Tree Species Fact Sheets, Maine Tree Club. `http://www.umaine.edu/umext/mainetreeclub/` September 2006.

2. The `Places` table should include a column named `area` that holds the area of each place in square miles.

3. Each of `Trees` and `Places` should include a column named `id` that uniquely identifies rows in the that table.

We also use two additional tables:

1. `Individuals(id, botname, location, dia, height, mdate)`. Each row in this table represents an individual tree, with the columns representing, in order, an attribute that uniquely identifies an individual tree, the botanical name of the tree species, a short textual description of the individual's location (e.g., "20 feet from the library entrance"), its trunk diameter in inches, its height in feet, and the date of these measurements.

2. `OSchedule(id, obs, indid, odatetime)`. Each row in this table represents a scheduled observation, with the columns representing, in order, a unique identifier for the scheduled observation, the name of the observer (person), the individual tree to be observed (identified using the `id` attribute of `Individuals`), and the date and time of the scheduled observation.

Choose appropriate types for the columns of these tables. Unless otherwise specified, you may assume that all string-valued attributes contain at most 100 characters. Exception are the `features` attribute of `Trees` and the `location` attribute of `Individuals`, which may each contain up to 100,000 characters.

The Application Program: As described further in the packaging instructions below, your submission should *produce* (not contain) an executable file called `treedb`, that uses PostgreSQL to implement the application described below. You must implement your application program as a Unix command-line program that reads from standard input and writes to standard output. Your program will be tested and graded on Gandalf. If you use some other machine for development, please check very carefully that your code runs on Gandalf. 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. We will assume that the separator string is not a substring of any valid string input to this application. Except the output described in this homework, your program should not produce any extra output, such as diagnostic messages.

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:

```
connect
bigmoose
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. Integers will be listed in conventional format (e.g., 123, 74). You may assume that all numbers are in the range $[0 \ldots 10^5]$. Date-time values are in UTC, with the format `YYYY-MM-DD HH:MM:SS`. For example, `2006-09-27 19:04:09` denotes nine seconds past 7:04 p.m. on the 27th of September, 2006, UTC (i.e., 3:04:09 p.m. EDT).

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 are encouraged, but not required, 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 with 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 `.bashrc` 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 all the database tables required by this application, as described on page 1. This function will be called before any of the functions below. It does not return any results.

destroyTables(): This function should cause the removal of the database tables created by `createTables`. After `destroyTables`, the database should 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.

addTree(n,b,t,d,h,m,x): When this function is invoked, your application should add a row (n, b, t, d, h, m, x, i) to the `Trees` table, where n, b, t, d, h, m, and x denote, respectively, the common name, botanical name, tree type, typical trunk diameter, typical height, minimum zone, and maximum zone, and where i is an identifier of your program's choosing. (See the note on identifiers below.) The output of this function is the identifier i.

findTree(s): This function should search for trees for which s occurs as a substring of either common name or botanical name (or both). This search, and **all searches on string-valued attributes**, should be case-insensitive unless specified otherwise. The matching tree records should be printed one per line. Each line should contain the tree's identifier (see above), common name, and botanical name, separated using the separator string described earlier (page 2). The output should be sorted in ascending lexicographic order of botanical name (case insensitive). Output lines here and elsewhere should be terminated by a single newline character.

describeTree(i): This function should print descriptive information about the tree identified by the given identifier i (*exact*, *case-sensitive* string match). If there is no tree with identifier i, no output should be produced and this condition is not an error. If the tree identified by i exists, the following information should be printed on a single line (in this order): common name, botanical name, minimum zone, and maximum zone.

For this and other functions, attribute values and other items printed on an output line should be separated using the separator string described earlier (page 2). Strings should be printed literally (with no quotes, padding, or other artifacts). Integers and dates should be printed in the format used for the input.

addPlace(c,s,p,z,y,l): When this function is invoked, your application should add a row (c, s, p, z, y, l, j) to the `Places` table, where c, s, p, z, y, and l denote, respectively, the city, state, population, zone, subzone, and minimum temperature, and where j is an identifier similar to that used in `addTree`. The output of this function is the identifier j.

addIndividual(k,b,l,d,h,m): When this function is invoked, your application should add a row (k, b, l, d, h, m) to the `Individuals` table, where k is an identifier generated as in `addTree`, and b, l, d, h, and m denote values for the remaining columns of the `Individuals` table in the order they were introduced earlier. The output of this function is the identifier k.

addOSchedule(o,i,d): When this function is invoked, your application should add a row (l, o, i, d) to the `OSchedule` table, where l is an identifier generated as in `addTree`, and o, i, and d denote values for the remaining columns of the `OSchedule` table in the order they were introduced earlier. The output of this function is the identifier l.

Note on Identifiers: The identifiers generated by your program in response to the `addTree`, `addPlace`, and `addIndividual` functions must uniquely identify the rows in the respective tables. Your application is responsible for generating and managing these identifiers. Once your application has exposed a tree's identifier (by printing it as output), the identifier may be presented as an argument of the `describeTree` function at any point in the future. These identifiers must persist between sessions. For example, if your program exposes a tree identifier xyzzy182 during one session a `describeTree` function call with xyzzy182 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 `descrbeTree` 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.) There are similar constraint on identifiers in the `Places`, `Individuals`, and `OSchedule` tables: Once exposed, they must permit lookup using the `describePlace`, `describeIndividual`, and `describeSchedule` functions introduced below.

describePlace(j): This function should print descriptive information about the place identified by the identifier j (*exact*, *case-sensitive* string match). If there is no place matching identifier j, no output should be produced and this condition is not an error. If the place identified by j exists, the following information should be printed on a single line (in this order): city, state, population, zone, subzone, and minimum temperature.

describeIndividual(k): This function is to the `Individuals` table what `describePlace` is to the `Places` table. The output, if any, consists of the values of the columns of the `Individuals` table, in the order they were introduced earlier.

describeSchedule(i): This function is to the `OSchedule` table what `describePlace` is to the `Places` table. The output, if any, consists of the values of the columns of the `OSchedule` table, in the order they were introduced earlier.

addOScheduleX(o,i,p): This function is similar to `addOSchedule`, differing only in how the time (and date) of the scheduled observation is specified. Instead of it being specified explicitly, it is specified using a *time-pattern* p as described below. An noted below, a single invocation of this function may result in the creation of several scheduled observations, unlike the `addOSchedule` function, which always creates one scheduled observation. The other arguments, o and i, are treated as in `addOSchedule`. Recall that all times are in UTC.

The scheduled observation times are based on interpreting the time-pattern p as a *crontab*[2] 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 modified) 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:
>
> | field | allowed values |
> |---|---|
> | minute | 0–59 |
> | hour | 0–23 |
> | day of month | 1–31 |
> | month | 1–12 or names |
> | day of week | 0–7 (0 and 7 mean Sun.) |
> | [year | integer year of Gregorian calendar] |
>
> 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 reservation] at hours 8, 9, 10 and 11. [Similarly, `2006-2007` specifies a reservation for years 2006, 2006, and 2007 A.D. We will assume a temporal granularity of one minute. Thus the time-pattern `* * 1 1 * 2006` specifies the time points marking each minute of each hour of January 1st, 2006 (a total of 60×60 points in time.]
>
> 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.

A single invocation of `addOScheduleX` results in the creation of several scheduled observations (several entries in the `OSchedule` table). We will refer to this group of scheduled observations created by a `addOScheduleX` invocation as a *observation group*. For example, the following invocation results in 24 scheduled observations for January 15th, 2007 for the first 10 minutes of each hour of that day:

[2] Paul Vixie, *crontab*—tables for driving *cron*, Manual, 4th Berkeley Distribution January 1994.

```
addOScheduleX(Alice, t101ab, 0 * 15 1 * 2007)
```

The output of this function is a list of observation-schedule identifiers, listed one per line.

For this assignment, you may assume that reservations beyond 2008-12-31 23:59:59 may be safely ignored. For patterns that specify reservations both before and after this date, only the reservations after the date may be ignored; the earlier ones must be managed properly. This assumption may simplify your implementation of invocations such as the following:

```
addOScheduleX(Alice, t101xy, 0 * 15 1 * *)
```

matchObservations(*p*, *l*, *o*): Here p is a time-pattern as described above, while l is a floating-point number. This function finds all scheduled observations for observer (person) o in the time interval $[i, i+l]$, where i is any instant of time that matches the pattern p and l is interpreted as a time duration, in hours. The output consists of the identifiers of the matching scheduled-observations, one per line.

$\star$ **getReqdObs(*p*, *l*):** This function finds individual trees that are in need of observations in the interval specified by time-pattern p and duration l, interpreted as in `matchObservations`. We say an individual tree t needs an observation in an interval I if there is at least one time instant $i \in I$ such that there is no scheduled observation of t in the interval $[i - M, i]$, where M denotes the time duration of one month. The output of this function is a list of records, one per line, indicating the identifier of a matching individual and the earliest instant of time i satisfying the condition described above.

Packaging and Submission: You should generate a gzip-compressed tar archive file called `M-hw02-N.tgz`, replacing M with your last-name and replacing N with an arbitrary 4-digit integer (e.g., `Doe-hw01-4242.tgz`). The execution of the following sequence of commands on Gandalf (replacing `Doe-hw02-4242.tgz` with the name of your file, which resides in `/tmp`) should result in the creation of a directory `/tmp/Doe-hw02`:

```
cd /tmp; gzip -dc Doe-hw02-4242.tgz | tar xf -
```

Typing `make` at the Unix shell prompt in the `/tmp/Doe-hw02` directory should result in the complete compilation of your program, producing an executable file called `treedb`. You will need to include an appropriate Makefile for this procedure to work. You should also include a short `README` file describing the files in your submission, along with anything that may be helpful in fixing your submission if it does not work as above. You must make sure you program works when stdin and stdout are redirected. For example, we may run your program as follows, where `datafile` is a text file contains the input of the program: `treedb < datafile`. Please check carefully that your file satisfies these requirements. Submit the file using anonymous FTP as described in the previous assignment.