Goal. This assignment asks you to build a small database application for managing information related to rooms and reservations. The main goal of this assignment is to gain experience using one or more application programming interfaces (APIs) to SQLite. 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. This homework should provide a concrete motivation for these topics. Some aspects of the assignment will need clarifications on the class newsgroup.

The Programming Environment. An important part of this assignment is learning the interface between a typical programming environment and the database system. For programming your application, you should use Java with JDBC. If you wish to use a different programming language or environment, you should make arrangements with me *no later than* the class immediately following the one in which this assignment is given. In general, exceptions will only be granted if there are compelling reasons. 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.

The Application Program. As described further in the packaging instructions below, your submission should produce an executable file called `roomboss`, that uses SQLite 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 *aturing* (the machine used for class accounts). If you use some other machine for development, *please check very carefully that your code runs on aturing.* Code that does not properly compile and run on aturing will get little or no credit even if the problems are due to some peculiarity of aturingk 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 (␣:␣, using ␣ to represent a 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 prompts or diagnostic messages.

The interface is used to invoke a set of functions, described below, 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
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 $[0, 10^5]$. Date-time values are in UTC, with the format `YYYY-MM-DD HH:MM:SS`, along the lines of the ISO 8601 standard.[**?**] For example, `2016-03-20 04:29:02` denotes two seconds past 4:29 a.m. UTC on the 20th of March, 2016 (the Vernal Equinox). Unless otherwise specified, you may assume that all string-valued attributes contain at most 100 characters. An exception is the `comments` attribute of `Facilities`, which may contain up to 100,000 characters.

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. 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.

Database Tables. The application uses the database tables of Figure 1, where you should make suitable choices for the missing information, marked with `?` signs, based on the rest of the application description. The intended semantics of the tables should be clear from the column names and the descriptions that follow; if not, ask for clarifications.

Rooms

id	building	floor	room	directions
?	varchar(20)	integer	integer	varchar(100)
?	Neville	1	110	Near the north entrance...
?	Neville	2	210	Take the stairs...
?	Neville	2	227	Large room near...
?	Corbett	1	101	Right there...
?	Corbett	2	201	Left, then right...

Facilities

id	building	room	chairs	screens	comments
?	varchar(20)	integer	integer	integer	?
?	Neville	210	25	0	Now also 208. Weird setup.
?	Neville	227	35	1	The old workhorse; quite nice.
?	Corbett	101	2	0	Let me tell you something...

Reservations

cnum	rstart	rend	building	room	name
?	?	?	?	?	?
?	?	?	?	?	?

Figure 1: A database of rooms.

Functions. The functions that your program should implement are described below. 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). A *session* begins when your application program is invoked and ends when the end of input is reached on the standard input. This function will be the first one invoked in any session, and it will be invoked exactly once per session. 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.

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 the database tables described on page 2, if they do not already exist in the database. 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`, if they are present in the database. 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.

addRoom(b, f, r, d). When this function is invoked, your application should add a row (i, b, f, r, d) to the `Rooms` table, where b, f, r, and d denote, respectively, the building, floor, room number, and directions, 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.

findRoom(s). This function should search for rooms for which s occurs as a substring of either building name or directions (or both). This search, and **all searches on string-valued attributes**, should be case-insensitive unless specified otherwise. The matching room records should be printed one per line. Each line should contain the room's identifier (see above), building name, and room number, separated using the separator string described earlier (page 1). The output should be sorted in ascending lexicographic order of building name (case insensitive) and ascending order of room numbers (secondary sort order). Output lines here and elsewhere should be terminated by a single newline character.

describeRoom(i). This function should print descriptive information about the room identified by the given identifier i (*exact*, *case-sensitive* string match). If there is no room with identifier i, no output should be produced and this condition is not an error. If the room identified by i exists, the following information should be printed on a single line (in this order): building name, room number, floor, number of chairs, and number of screens.

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 1). 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.

addFacilities(b, r, c, s). When this function is invoked, your application should add a row (f, b, r, c, s) to the `Facilities` table, where b, r, c, and s denote, respectively, the building name, room number, number of chairs, and number of screens, and where f is an identifier similar to that used in `addRoom`. The output of this function is the identifier f.

addReservation(s, e, b, r, n). When this function is invoked, your application should add a row (c, s, e, b, r, n) to the `Reservations` table, where s, e, b, r, and n denote, respectively, the start and end times of the reservation, the building name, the room number, and the name of the person making the reservation. The identifier c is a reservation confirmation number (identifier) that is also the output of this function.

Note on Identifiers. The identifiers generated by your program in response to the `addRoom`, `addFacilities`, and `addReservation` 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 room's identifier, say, by printing it as output, the identifier may be presented as an argument of the `describeRoom` function at any point in the future. These identifiers must persist between sessions. For example, if your program exposes a room identifier xyzzy182 during one session a `describeRoom` 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 `descrbeRoom` 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 is a similar constraint on reservation identifiers: Once exposed, they must permit lookup using the `describeReservation` function below.

describeReservation(c). This function should print descriptive information about the room reservation identified by the given identifier (confirmation number) c (*exact*, *case-sensitive* string match). If there is no reservation matching identifier c, no output should be produced and this condition is not an error. If the reservation by c exists, the following information should be printed on a single line (in this order): building name, room number, start time, end time, and name of reservation holder.

makeReservations(p, l, b, r, n). This function is similar to `addReservation`, differing only in how the begin and end times of the reservation are specified. Instead of these times being specified explicitly, they are specified using a *time-pattern* p and time-length l as described below. The other arguments (b, r, and n) are treated as in `addReservation`. Recall that all times are in UTC.

The begin times are based on interpreting the time-pattern p as a *crontab*[**?**] 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 common era] |
>
> 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, `2016-2018` specifies a reservation for years 2016, 2017, and 2018 C.E. We will assume a temporal granularity of one minute. Thus the time-pattern `* * 1 1 * 2016` specifies the time points marking each minute of each hour of January 1st, 2016 (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 `makeReservations` creates, in general, several entries in the `Reservations` table. We will refer to this group of reservation entries created by a `makeReservations` invocation as a *reservation group*. The time-length argument specifies the duration of each

reservation in the reservation group, in minutes. For example, the following invocation results in 24 reservations for January 15th, 2016 for the first 10 minutes of each hour of that day:

```
makeReservation(0 * 15 1 * 2016, 10, Neville, 210, Alice)
```

Time-Limit Assumption. You may ignore reservations beyond 2018-12-31 23:59:59. For patterns that specify reservations both before and after this timestamp, 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:

```
makeReservation(0 * 15 1 * *, 10, Neville, 210, Alice)
```

matchReservation(p, b, r). This function finds reservations for room number r in building b such that there is at least one instant of time that matches the time-pattern p and that lies between the begin and end times of the reservation (including the end-points). The interpretation of p is identical to that used by `makeReservations`. The output consists of the confirmation numbers of the matching reservations, one per line.

getFreeRoom(p, l, c, s). This function finds rooms that are unreserved at all times matching the time-pattern p and l, which are interpreted as in `makeReservations`. In addition, the matching rooms are required to have at least c chairs and s screens. The output consists of the building and room-number for each matching room, one per line. A room is considered unreserved at a time matching p only if it is unreserved for the entire length of each interval specified by p and l. If the facilities (number of chairs and screens) of a room are not known, the room does not match.

Packaging and Submission. You should submit your work via the interface at `http://cs.umaine.edu/~chaw/u/` as a single file named using the scheme `cos480-hw01-`*L*`-`*F*`-`*N*`.tgz`, replacing L and F with your last-name and first-name, and N by an arbitrary 4-digit integer.

Unpacking your submission should create a directory `cos480-hw01-`*L*`-`*F*`-`*N*. (as a subdirectory of the working directory). Typing `make` at the Unix shell prompt in this directory should result in the complete compilation of your program, producing an executable file called `roomboss`. 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 does not make any assumptions on the nature of stdin and stdout and, in particular, that it 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: `roomboss < datafile`, but we may also invoke `roomboss` using an interactive terminal. If you program using the proper conventions, `roomboss` should just work in all such cases without any special effort on your part, but if you bypass those conventions, it may not. Please check carefully that your file satisfies these requirements. *Proper packaging and submission is an important part of this assignment* and your score will suffer greatly if your submission falls short in this area.

⋆ **Advanced Work.** [This portion of the assignment is required for COS 580 and optional (extra credit) for COS 480.] For each of the following, you must (1) include a clear and precise description of your solution in your README file (or an accompanying PDF file), (2) implement that solution, and (3) include suitable test data and instructions.

1. Do away with the time-limit assumption on page 6. In your README file, clearly describe your scheme for managing reservations that extend indefinitely into the future.

2. Add a feature to allow the selective modification and cancellation of reservations, including canceling some, but not all, of those created by a single `makeReservations` invocation.

Quantify the space and time costs of the methods you describe above by

1. providing suitable expressions for the running times of the operations, the database space requirements, and the computation (working) space requirements in terms of suitable input parameters, and

2. conducting a brief but precise and well documented experimental evaluation.