1. Preliminaries

Before starting on this assignment, please be sure to read the General Instructions that are on Canvas (under Files-->General Resources), and also make sure you are able to log in to the class PostgreSQL server. You'll get help on this in your Lab Section, not the Lectures, so *be sure to attend Lab Sections*.

2. Goal

The goal of the first assignment is to create a PostgreSQL data schema with 7 tables. That is all that is required in this assignment; don't do anything that's not required. The other Lab Assignments are much more difficult. In your Lab Sections, you may be given information about how to load data into a table and issue simple SQL queries, because that's fun, but loading data and issuing queries are **not** required in this assignment. (That will show up in the Lab2 assignment.)

3. Lab1 Description

3.1 Create PostgreSQL Schema Lab1

As we noted in the general instructions, you will create a Lab1 schema to set apart the database tables created in this lab from tables you will create in future labs, as well as from tables (and other objects) in the default (public) schema. Note that the meaning of schema here is specific to PostgreSQL, and distinct from the general meaning of schema. See here for more details on PostgreSQL schemas. You create the Lab1 schema using the following command:

```
CREATE SCHEMA Lab1;
```

[PostgreSQL makes all identifiers lowercase, unless you put them in quotation marks, e.g.,, "Lab1". But in CSE 180, you don't have to bother using quotation marks for identifiers. We use capitals in assignments for readability, but it's okay (and equivalent) if you use lab1 as the schema name.]

Now that you have created the schema, you want to make Lab1 be the default schema when you use psql. If you do not set Lab1 as the default schema, then you will have to qualify your table names with the schema name (e.g., by writing Lab1.customer, rather than just customer). To set the default schema, you modify your search path as follows. (For more details, see here.)

```
ALTER ROLE username SET SEARCH_PATH to Lab1;
```

You will need to log out and log back in to the server for this default schema change to take effect. (Students **often forget** to do this, and then are surprised that their tables aren't in the expected schema.) To see your current SEARCH_PATH, enter:

```
SHOW SEARCH_PATH;
```

3.2 Tables

You'll be creating tables for a very simplified version of a Adventure game database schema (which we'll call Adventure), with tables for Members, Rooms, Roles, Characters, Things, Monsters and Battles. The data types and Referential Integrity for the attributes in these 7 Adventure tables are described in the next section. No, the Adventure schema doesn't provide everything that an Adventure game database would have, but it's a decent start.

Important: To receive full credit, you must use the attribute names as given, and the attributes must be in the order given. Also, the data types and referential integrity must match the specifications given in the next section. Follow directions; do not do more than you're asked to do in this assignment.

Members(memberID, name, address, joinDate, expirationDate, isCurrent)

Rooms(roomID, roomDescription, northNext, eastNext, southNext, westNext)

Roles(role, battlePoints, initialMoney)

Characters(memberID, role, name, roomID, currentMoney, wasDefeated)

Things(thingID, thingKind, initialRoomID, ownerMemberID, ownerRole, cost,
extraBattlePoints)

Monsters(monsterID, monsterKind, name, battlePoints, roomID, wasDefeated)

Battles(characterMemberID, characterRole, characterBattlePoints,
monsterID, monsterBattlePoints)

The underlined attribute (or attributes) identifies the Primary Key of each table. A table can only have one Primary Key, but that Primary Key may involve multiple attributes.

- A member in **Members** specifies the member's ID, name, address, date they became members in the game, date their membership in the game expires, and whether their membership is current.
- A room in **Rooms** specifies the ID of the room, a description of the room, and the next room if someone moves north, east, south or west from that room.
    - Any value that appears in northNext, eastNext, southNext or westNext in a Rooms row must appear as the roomID of a row in Rooms.
    - Sometimes there is no next room in a particular direction and the value of (say) southNext will be NULL, which is an attribute value that means "does not apply" or "unknown". That's okay.
- A role in **Roles** might be (or example) a knight, a mage, a rogue or a cleric. Each role has a character string for that role (e.g., 'knight'), the number of battle points for characters in that that role), and the initial amount of money for characters who have that role.

- Each member may have more than one character in the **Characters** table, but may have only one character for each role. For example, a member may have four characters, a knight, a mage, a rogue and a cleric. But a member may not have two rogues. So each character in Characters has the memberID and role for that character. Each character also specifies the character's name, the room that the character is in, the amount of money that the character has, and whether the character has ever been defeated in a battle.
    - Any memberID that's in a Characters row must appear as a memberID of a row in the Members table.
    - Any role that's in a Characters row must appear as a role in a row in the Roles table.
    - Any roomID that's in a Characters row must appear as a roomID of a row in the Rooms table.
- Each thing in **Things** specifies its ID, the kind of thing it is (which we'll describe later), the room in which the thing is located when the game starts, the character who owns that thing (which is identified by the memberID and role of the owner), the thing's cost, and the extra points a character receives in a battle when it has that thing. Sometimes a thing is not owned by anyone; we'll explain how that works below. When a thing is owned, it moves around with its owner.
    - Any initialRoomID that's in a Things row must appear as a roomID of a row in the Rooms table.
    - Any (ownerMemberID, ownerRole) that's in a Things row must appear as a (memberID, role) of a row in the Characters table. However, if a thing isn't owned, it's okay for ownerMemberID and ownerRole to be NULL, which is an attribute value that means "does not apply" or "unknown". That's okay.
- A monster in **Monsters** specifies the id of the monster, the kind of monster (which we'll explain later), its name, its battle points, the room that it's in and whether it has ever been defeated. Unlike characters, monsters won't be able to move to a different room.
    - Any roomID that's in a Monsters row must appear as a roomID of a row in the Rooms table.
- A battle in **Battles** is between a character and a monster, so it has attributes which identify the character (characterMemberID, characterRole) and the monster (monsterID). Those 3 attributes together make up the Primary Key of Battles, so there cannot be more than one battle between a particular character and a particular monster (but there might be none). Each battle also specifies the number of points for the character and the monster in their battle. The points that a character has in a battle equals the number of points associated with its role, with extra points for any things that character owned when it found the battle.
    - Any (characterMemberID, characterRole) that's in a Battles row must appear as a (memberID, role) of a row in the Characters table.
    - Any monsterID that appears in a Battles row must appear as a monsterID in a row of the Monsters table.
    -

In this assignment, you'll just have to create tables with the correct table names, attributes, data types, Primary Keys and Referential Integrity. "Must appear as" means that there's a Referential Integrity requirement. **Be sure not to forget Primary Keys and Referential Integrity when you do Lab1!**

3.2.1 Data types.

Sometimes an attribute (such as name or battlePoints) appears in more than one table. Attributes that have the same attribute name might not have the same data type in all tables, but in our schema, they do.

- For memberID, ownerMemberID, characterMemberID, roomID, initialRoomID, thingID and monsterID, use *integer*.
- For battlePoints, characterBattlePoints, monsterBattlePoints, extraBattlePoints, northNext, eastNext, southNext and westNext, also use *integer*.
- For thingKind and monsterKind, use *character with fixed length 2*. (We'll explain the values of these attributes in a later Lab Assignment.)
- For role, characterRole and ownerRole, use *character of variable length* with maximum length 6.
- For name and address, use *character of variable length*, with maximum length 50.
- For roomDescription, , use *character of variable length*, with maximum length 30.
- cost should be *numeric,* with at most 2 decimal digits to the left of the decimal point and 2 decimal digits after it.
- currentMoney and initialMoney should be *numeric,* with at most 3 decimal digits to the left of the decimal point and 2 decimal digits after it.
- joinDate and expirationDate should be of type *date*.
- isCurrent and wasDefeated should be *boolean*.

You must write a CREATE TABLE statement for each of the 7 tables in Section 3.2. Write the statements in the same order that the tables are listed above. **Use the data types, Primary Keys and Referential Integrity described above.** You will lose credit if you do anything beyond that, even if you think that it's sensible. Save your statements in the file create_lab1.sql

Note that PostgreSQL maps all SQL identifiers (e.g., table names and attributes) to lowercase. That's okay in your CSE 180 assignments. You won't lose points for Lab1 because Persons appears in the database as persons, and conferenceName appears as conferencename. It is possible to specify case for identifiers by putting that identifier inside double-quote symbols, e.g., as "Rooms". But then every time you refer to that identifier, you'll have to use the double-quotes. "ROOMS" is not the same identifier as "Rooms", and neither is the same as Rooms, which PostgreSQL maps to lowercase, as rooms. We use capitalization for readability, but we won't bother using double-quotes in our own Lectures and Lab Assignment solutions.

4. Testing

While you're working on your solution, it is a good idea to drop all objects from the schema every time you run the create_lab1.sql script, so you can start fresh. Dropping each object in a schema may be tedious, and sometimes there may be a particular order in which objects must be dropped. The following command, which you should put at the top of your create_lab1.sql, will drop your Lab1 schema (and all the objects within it), and then create the (empty) schema again:

```
DROP SCHEMA Lab1 CASCADE;
CREATE SCHEMA Lab1;
```

Before you submit your Lab1 solution, login to your database via psql and execute your create_lab1.sql script. As you'll learn in Lab Sections, the command to execute a script is: \i <filename> Verify that every table has been created by using the command: \d Note that when you execute \d, the tables that are displayed may appear in any order, not necessarily in the order in which you created them.

To verify that the attributes of each table are in the correct order, and that each attribute is assigned its correct data type use the following command: \d <table>.

We've supplied some load data that you can use to test your solution in the file load_lab1.sql. After you've created your tables, using the command : \i create_lab1.sql, you can load that data in psql by executing the command: \i load_lab1.sql. (Why will loading the data twice always result in errors?) If your solution fails on the load data, then it's likely that your solution has an error. But although testing can demonstrate that a program is buggy, testing cannot prove that a program is correct.

You do not have to develop your solution on unix.ucsc.edu, but please recognize that we'll run your solution on unix.ucsc.edu. If your solution fails on unix.uscs.edu, you'll receive a poor grade, even if your solution worked in some other environment.

5. Submitting

1. Save your script as create_lab1.sql You may add informative comments to your scripts if you want. Put any other information for the Graders in a separate README file that you may submit.

2. Zip the file(s) to a single file with name Lab1_XXXXXXX.zip where XXXXXXX is your 7-digit student ID. For example, if a student's ID is 1234567, then the file that this student submits for Lab1 should be named Lab1_1234567.zip

   If you have a README file (which is <u>not</u> required), you can use the Unix command:

   ```
   zip Lab1_1234567 create_lab1.sql README
   ```

   If you don't have a README file, to create the zip file you can use the Unix command:

   ```
   zip Lab1_1234567 create_lab1.sql
   ```

   (Of course, you should use **your own student ID**, not 1234567.) Submit a zip file, even if you only have one file.

Submit the zip file on Canvas under Assignment Lab1. Please be sure that you have access to Canvas for CSE 180. Registered students should automatically have access; students who are not registered in CSE 180 will have Auditor access, so they can't submit solutions. You can replace your original solution, if you like, up to the Lab1 deadline. (Canvas will give the new file a slightly different name, but that's okay.) <u>No students will be admitted to CSE 180 after the Lab1 due date.</u>

If you are working on the UNIX timeshare and your zip file is located there, you will need to copy your file to your computer so that you can upload it to Canvas through your browser. To do that, you will need an FTP (File Transfer Protocol) client to securely transfer files from the UNIX timeshare. A widely used secure FTP client is Filezilla. Installation instructions are found in the site of FileZilla (make sure you install the distribution suitable for your operating system). After opening the Filezilla client, you will need to set the host field to unix.ucsc.edu, the username to your CruzId and the password to your Blue password, while the port number should be set to 22 (the default port for remote login). By clicking the Quickconnect button, if your credentials are correct, you will connect and be able to see the contents of your remote Unix folder at the right pane (under the title "Remote site"), while the left pane (under the title "Local site") will display the contents of your local file system. With the mouse, you can drag the file from the Unix folder and drop it to the desired location at your computer. This will transfer the file to your local machine, without erasing it from its original remote location. Filezilla is only one of several options for an FTP client. If you are finding it difficult to install the necessary tools and successfully do file transfers, you should <u>promptly</u> ask for help in the Lab Sections; **do not postpone this until the deadline date**. Computers in UCSC Unix Labs also have pre-installed SSH and FTP clients (PuTTY and PSFTP).

Other approaches to copy files includes using SCP (Secure Copy) and using Cut-and-Paste, where you copy the contents of the file from the unix system, and then paste contents into a file on your computer. Cut-and-Paste works well with for small files, but it's a hack that does not work well for large files.

The CSE 180 Teaching Assistants, Tanuj Gupta and Gavin Cooke, will discuss approaches to access unix remotely (SSH for Mac/Linux and PuTTY for Windows) and to move files to your computer (SCP for Mac/Linux and Filezilla for Windows/Mac/Linux) with you during Lab Sections. <u>Attend your Lab Section to ensure that you know how to handle this correctly!</u>

Lab1 is due by 11:59pm on Tuesday, October 11. **Late submissions will not be accepted (Canvas won't take them, nor will we), and there will be no make-up Lab assignments.** Always check to make sure that your submission is on Canvas, and that you've submitted the correct file. You will receive **no credit** if you accidentally submit the wrong file, even if you attempt to "prove" that you completed the correct file on time.