Here are a couple of ways to store the data and keep track of changes using simple data structures alongside the main data structures of the system.

Undo functionality:
You can use an *Enum* to check which **type** of operation has been performed. The types could be remove, add, change, etc.

```
from enum import Enum
class OperationType(Enum):
    ADD = 1
    UPDATE = 2
    DELETE = 3
```

Make a class that you can call ***OperationInfo***. That class needs variables to store relevant information, like what data changed and what its values were when this operation happened.

```
class OperationInfo:
    def __init__(self, type, data):
        self.type = type
        self.data = data
```

Whenever an operation is performed in the system put a new instance of ***OperationInfo*** on a **stack** (a list in python can be used as a stack, using just ***append()*** and ***pop()***). A student can decide if the operation type stored is the operation that happened or the inverse operation (the operation needed in order to undo the operation). All relevant data needed to perform that operation so that the system is exactly as it was again must be stored here. *It is not necessary to perfectly replicate information that does not affect the user, such as a unique id that is only used by the system (e.g. if that is usually auto-generated).*

```
info = OperationInfo(OperationType.DELETE, {"id": 11})
stack.append(info)
info = OperationInfo(OperationType.ADD, {"name": "Jon", "phone": "123"})
stack.append(info)
```

Students can of course decide to store the relevant information in a different way. They can make a specific class or store it in separate variables. Here the built-in dict functionality is used.

Whenever the user select the undo operation, take the top off the **stack** and perform the operation stored there (or its inverse) using the type and data in the ***OperationInfo*** object.

```
some_info = stack.pop()
if some_info.type == OperationType.DELETE:
    #perform a delete using some_type.data
elif some_info.type == OperationType.ADD:
    #perform an add using some_type.data
```

Wait-list functionality:

This description uses the ContactList as a base, but use whatever assignment you are working on, sport (or group) and member, etc.

- For each instance of ***ContactList*** in the system make a **queue**.
    - This instance can reside inside the instance of ContactList or in a separate part of the system, but must be connected to that particular contact list.
- Whenever a contact is added to that contact list and that list is full (max number of contacts per list) add the contact to the **queue** *instead*.
- Whenever a contact is removed from the contact list take the first member off the **queue** and add them to the contact list proper.
    - *Maybe have the system notify the user that this has happened, or ask the user whether they want this to happen or not.*