CSCI 3357: Database System Implementation
Homework Assignment 4
Due Monday, September 30

Your task this week is to implement non-quiescent checkpointing. You will need to do the following things:

- Create a new class `NQCheckpoint` to implement non-quiescent checkpoint records.
- Modify the interface `LogRecord` to have a constant `NQCKPT` with the value 6. You will need to update the constant definitions and the method `createLogRecord`.
- Modify the class `RecoveryMgr` in two ways:
    - Add a method `checkpoint` that writes a non-quiescent checkpoint record to the log. This method will be called from the `Transaction` class; its argument will be a list of ids of the active transactions.
    - Modify the method `doRecover` to handle non-quiescent checkpoint records. This is the most difficult part of the assignment; think carefully about what needs to happen. As an aid to debugging, your code should print the log records as they are encountered.
- As a further aid to debugging, add a `println` statement in the `undo` methods of `SetIntRecord` and `SetStringRecord` so you can see when the `recover` and `rollback` methods actually undo a value.

In addition, the `Transaction` constructor must be modified to keep track of the currently-active transactions and to periodically call the recovery manager's checkpoint method. I have written this code for you, to ensure that everyone does checkpointing at the same times (and thus has the same output). Download and use my revised *Transaction.java* file.

To test your code, you can use my `HW4TestA` and `HW4TestB` programs. Program `HW4TestA` does an NQ checkpoint in the middle of running some transactions, and stops before all transactions have finished. Program `HW4TestB` preforms recovery on the log file. When I run `HW4TestA`, I get the following output:

```
new transaction: 1
new transaction: 2
transaction 2 setint old=0 new=10002
transaction 2 committed
new transaction: 3
transaction 3 setint old=0 new=10003
new transaction: 4
transaction 4 setint old=0 new=10004
NQ CHECKPOINT: Transactions 1 3 4  are still active
new transaction: 5
transaction 5 setint old=0 new=10005
```

```
transaction 3 committed
transaction 5 committed
new transaction: 6
transaction 6 setint old=0 new=10006
new transaction: 7
transaction 7 setint old=0 new=10007
new transaction: 8
transaction 8 setint old=0 new=10008
transaction 4 committed
transaction 7 committed
new transaction: 9
transaction 9 setint old=0 new=10009
NQ CHECKPOINT: Transactions 1 6 8 9  are still active
new transaction: 10
transaction 10 setint old=0 new=100010
new transaction: 11
transaction 11 setint old=0 new=100011
transaction 6 committed
transaction 9 committed
transaction 11 committed
new transaction: 12
transaction 12 setint old=0 new=100012
new transaction: 13
transaction 13 setint old=0 new=100013
new transaction: 14
transaction 14 setint old=0 new=100014
transaction 8 committed
transaction 12 committed
transaction 14 committed
NQ CHECKPOINT: Transactions 1 10 13  are still active
new transaction: 15
transaction 15 setint old=0 new=100015
new transaction: 16
transaction 16 setint old=0 new=100016
new transaction: 17
transaction 17 setint old=0 new=100017
transaction 10 committed
transaction 15 committed
transaction 17 committed
transaction 1 committed
```

Running `HW4TestB` gives the following output:

```
new transaction: 1
Initiating Recovery
Here are the visited log records
<START 1>
<COMMIT 1>
<COMMIT 17>
<COMMIT 15>
<COMMIT 10>
<SETINT 17 [file testfile, block 15] 99 0>
<START 17>
<SETINT 16 [file testfile, block 14] 99 0>
      undoing record
<START 16>
```

```
<SETINT 15 [file testfile, block 13] 99 0>
<NQCKPT 1 10 13 >
<START 15>
<COMMIT 14>
<COMMIT 12>
<COMMIT 8>
<SETINT 14 [file testfile, block 12] 99 0>
<START 14>
<SETINT 13 [file testfile, block 11] 99 0>
      undoing record
<START 13>
```

The first log record visited, `<START 1>`, is the log record for the `HW4TestB` transaction itself. The second log record, `<COMMIT 1>`, is the commit record for the first `HW4TestA` transaction. These transaction numbers repeat because SimpleDB starts transaction numbers from 1 each time it begins. That is kind of annoying, but not worth changing.

The last two values in the `SETINT` records are 99 and 0. The 99 denotes an offset of the block, and the 0 denotes that the previous value at that offset is 0.

Note that only the updates for transactions 13 and 16 need to get undone. The recovery method uses the `NQCHECKPOINT` record to know that it can stop after seeing the `START` record for transaction 13.

This test file gives you the flavor of how you can test your code. You probably should begin with a stripped down version of it, and increase complexity as you work out the bugs. Another debugging aid is the program `PrintLogFile`, which is in the recovery directory of the SimpleDB distribution. After running `HW4TestA`, run `PrintLogFile` to ensure that the log records are what you expect.

Once you get your code working, you might have fun doing the following: Run several JDBC programs that make (non-conflicting) changes to the database, modified so that some of them sleep before committing. When all are running, stop the server. Then bring up the server again, and see what gets printed during recovery.

WARNING: This assignment is substantially harder than previous ones. It's not that you need to write a lot of code. It's that you will need to think hard to figure out what code you need to write and where it should go. Start early.