CS2223, Project 3
Implement a Securities Market*

Due Tuesday, 20 Nov 2012

In this project, you will implement a simple market for financial securities, such as stocks and bonds. The technical core is to use a pair of priority queues (heaps), based on different orderings. Thus, you must *parameterize* the heap code to allow different ordering relations. The client code you write to use these heaps will maintain the orders to buy and sell securities. For instance, suppose:

- Buyer B wants to buy up to 100 shares of Universal Manufacturing for \$182 per share (or less), and
- Seller S wants to sell 50 shares of Universal Manufacturing for \$181.

Then a *transaction* can occur. B gets half of his *buy order* fulfilled, at the quoted price of \$181. The remaining half of the buy order is still waiting for another transaction at any price up to \$182. S's completed sell order is removed from the market.

A market is in *equilibrium* when no transactions can occur. That means that all the buy orders for a stock are bidding too little money to match any sell order for that stock. So no transaction will be triggered.

This could happen when there are no sell orders, or when there are no buy orders. It also happens when all the sellers are greedy and the buyers are too stingy. We'll use some terminology:

Buy order: A buyer places a *buy order*. Buy orders contain (at least) the name of the security, the name of the buyer, a quantity (i.e. the number of shares), and a *bid*.

Bid: The *bid* is the maximum money that the buyer is willing to pay on this buy order. A transaction may occur for less.

Sell order: A seller places a *sell order*. Sell order contain (at least) the name of the security, the name of the seller, a quantity (i.e. the number of shares), and a *quote*.

*Joshua Guttman, FL 137, `mailto:guttman@wpi.edu`. Include [cs2223] in the Subject: header of email messages. Due midnight at the end of 20 Nov.

Quote: The *quote* is the exact price at which the seller will sell this lot of stock.

Transaction record: A transaction record contains (at least) the an identifier for the security, identifiers for the buyer and seller, a quantity (i.e. the number of shares), and a price. This price equals the quote from the sell order that the transaction was generated from.

Book: The collection of not-yet-fulfilled buy and sell orders for a single security.

In real life, there are also other kinds of information, for instance an expiration time for a buy or sell order, but we will just work with this.

At one time, the book was a physical book, and a person with a seat on the stock exchange (a "specialist" trader) would copy orders into it, manually matching buyers and sellers to create transactions. This was called *making a market* in that security. Now making a market is a purely electronic activity.

When the market is not in equilibrium, the *book* may contain many buy and sell orders, in which the buyers are willing to pay at least the price the sellers have quoted. Which transactions should occur?

One principle is the lowest quote on a security should be made available first. That's because every buyer should have an opportunity to have their orders fulfilled at the lowest available price. If a low quote is available, some bidder should be able to get the security at that price.

The other principle is that the highest bid should be available first. That's not because the trade will take place at that price: The sell order's quote determines the actual price. However, a buyer making a higher bid is taking the risk of paying more if the sell quotes are high. So they should also get a benefit, namely that their bids should be fulfilled before lower bids.

These two principles will dictate your implementation strategy:

- The book for each security will be a pair of priority queues, one for the sell orders, and one for the buy orders.
- The sell orders will form a min-heap, so that the lowest quote is always at the top of that heap.
- The buy orders will form a max-heap, so that the highest bid is always at the top of the heap.
- To drive the market to equilibrium, just look at the tops of the two heaps. If no transaction is possible, because the bid is below the quote, change nothing.
- Otherwise, take the smaller of the two quantities. If there's some of the buy order remaining, decrease its quantity, and discard the sell order. If the sell order has a larger quantity, then decrease that, and discard the buy order. If they're equal, you've used them both up. Insert the transaction record at the end of the transaction log.
- Continue to look for more transactions until no transaction is possible.

- When a new buy or sell order arrives, place it in the right priority queue (heap), and drive the market to equilibrium.

This strategy, which only adds a new order to the book when the market has returned to equilibrium, isn't the right one in a world of high volumes in a distributed market—not to mention high-frequency trading—but it's perfectly reasonable for this project.

To implement this in Lua, we make the following assumptions:

1. Assume there is only one security, let's say "Universal," the Universal Manufacturing Company. They make everything. You will implement a single book, the one for Universal. (See Extra Credit, below.)

2. You should assume each buy order is a table of the form:

   ```
   {security = "Universal", bid = 186.14,
         amt = 100, buyer = "Mr B"},
   ```

   and each sell order is a table of the form:

   ```
   {security = "Universal", quote = 185.32,
         amt = 300, seller = "Ms S"}.
   ```

3. Each transaction will be a table of the form:

   ```
   {security = "Universal", sale_price = 185.32,
         amt = 100, buyer = "Mr B", seller = "Ms S" }
   ```

Retrieve `http://web.cs.wpi.edu/~cs2223/b12/proj/proj3.zip`. You will need to implement the following:

Modularize heaps: Modify the code for heaps (you can start with the code in `min_heaps.lua`) so that you can build heaps which use different ordering relations. A recommended way to do this is to add a method[1] to each table representing a heap. This function is the predicate that returns `true` if its first argument should be above its second argument in well-formed heaps and otherwise returns `false`.

For instance, the standard min-heap which uses $<$ corresponds to inserting a function `function (a,b) return a<b end` as the value in this field.

You will need to replace calls to $<$ between heap elements to calls to this field of the heap object.

Implement the book which consists of a pair of heaps as described above.

Implement the transaction log as the array containing the transactions that have occurred, in order.

[1] Or "field." For our purposes, a method just means a field of an object that happens to contain a function value.

Program an individual transaction as a procedure that inspects the tops of the two heaps in the book, and makes a transaction as described above if a bid equals or exceeds a quote.

Find equilibrium by writing a procedure to make transactions until no more can be made.

Accept a new buy order using a procedure named `BuyOrder` that takes as argument a buy order table value, inserts it into the buy queue in the book, and then drives the book to equilibrium.

Accept a new sell order using a procedure named `SellOrder` that takes as argument a sell order table value, inserts it into the sell queue in the book, and then drives the book to equilibrium.

The new `util.lua` has a `serialize` function that lets you inspect most Lua table objects easily. However, it doesn't automatically indent. In `util.lua` and `min_heaps.lua`, I've switched to a different mechanism for supporting modules, which should work on all platforms. Let me know in case of problems.

Your program should then be able to execute against files that look like this:

```
BuyOrder{security = "Universal", bid = 186.14,
       amt = 100, buyer = "Mr B"}
BuyOrder{security = "Universal", bid = 187,
       amt = 200, buyer = "Mr C"}
SellOrder{security = "Universal", quote = 185.32,
       amt = 300, seller = "Ms S"}
```

etc. Lua lets you just load this as a file of Lua code, using `dofile`, and then runs the procedures `BuyOrder` and `SellOrder` on these tables as arguments, just as if there were parentheses around the tables.

Extra Credit. For 20 points extra credit, modify your implementation so that it works for multiple securities.

Define a variable `market` to maintain the market as a table that associates each security—a string such as "Universal"—with the book that represents its state. When you get a new buy or sell order for a security you've already seen before, retrieve the book and process it as before. When you get a new buy or sell order for a security that doesn't yet have a book in the `market`, create a new book and install it in the market; then process it as before.