# Programming Exercise: Order Book processing

We are looking for a program that reads a file with fictional orders. The program should read and process **ALL** the orders in the order file and maintain order books as order messages are read and processed.

The order file *`orders.xml`* contains the fictive order messages that can be of the type ***AddOrder*** or ***DeleteOrder*** (see "Order Message" on the next page).

The file with the fictive orders will accompany this document. (Zipped as orders.7z)

The program should create order books (see "Order Book") dynamically as they are referenced in the order messages during processing of the XML file. Order books should be maintained in memory. Orders are inserted to the order books based on price and time, for examples see "Book Examples". After processing all order messages the program should print the order books according to the "output example" below.

**At start and end of processing, the timestamp should be outputted.**

All orderbooks should be printed at the end, along with the duration of processing in seconds.

Please provide your code for implementing the described functionality. Reflect and comment on performance aspects. If not implemented, what could be done to improve performance? Reflect and comment on data consistency if a multithreaded approach is used for processing the order flow.

**Steps for adding an order to an order book:**

1. Look up the order book based upon the book attribute. If the book does not exist create it.
2. Examine if the order can match one or more orders already in the book. Orders having their whole quantity being matched (i.e. `volume == 0`) are removed from the book. See matching example below.
3. If the incoming order is completely matched against orders in the book i.e. `volume == 0` the order is deleted.
4. If the incoming order is either partly or not matched the order is inserted into the book.

**Steps for deleting an order:**

1. Look up the order book based on the book attribute.
2. Locate the order in the book based upon the `orderId` attribute. If the order is not found just ignore the operation and just continue.
3. If found remove the order from the book.

## Order Messages

Orders are defined in a rather flat XML structure. Under the root element <Orders> there are only two types of elements and no deeper nesting.

```
<AddOrder book="book-2" operation="SELL" price="101.00" volume="87"
orderId="9363" />
```

- `AddOrder`, specifying that this element adds an order to an orderbook.
- book, identifies the order book to which the order should be added.
- *operation*, specifies whatever the order is a BUY or SELL order.
- `price`, specifies the price to which the order could be sold if a SELL order or bought if a BUY order.
- `volume`, specifies the amount of units to be sold or bought.
- `orderID`, unique identification of the order .

```
<DeleteOrder book="book-3" orderId="9036" />
```

- `DeleteOrder`, operation specifying that an order is to be deleted from the order-book.
- book, identifies the orderbook in which the order resides.
- `orderId`, identifies the order to be deleted.

## Order Book

An order is an interest in buying or selling an "asset". All interest (i.e. orders) for a given "asset" is kept in an order book. An order book have two sides (or lists). One side with the orders expressing a buy interest and another side with the orders expressing a sell interest. When a new order is to be inserted in to the book it must be examined if there is an interest in the book matching the interest of the incoming order. If so a *match* takes place. A match takes place if the a sell order price <= with the a buy price. The volume match is equal with the lowest volume for the both orders. The order volume for the incoming order and the order in book is reduced with the matched volume. If an incoming order cannot be matched or partly matched the order must be inserted into the book. *The orders are then inserted into the buy or sell list based upon price and time priority.*

A ***buy order*** with the price 101 € has a higher priority than a buy order with the price 100 €. On the other side a ***sell order*** with the price of 10 € has higher priority than a sell order with the price 11 €.

## Order Book Examples

An order book typically has a list of BUY and SELL orders. Orders are sorted by price-time priority, i.e. orders with a better price precedes an order with a worse price, and orders with the same price are prioritized such that the order that's been in the book for the longest time is processed first.

Assume an empty order book to which we add a BUY order with the price of 100.00 and with the volume 50. The order book will look as follows:

| Buy | Sell |
|---|---|
| 50 @ 100.0 | |

We now add a SELL order with the price of 101.50 and with a volume of 60. The order book will then look as follows:

| **Buy** | **Sell** |
|---|---|
| 50 @ 100.0 | 60 @ 101.50 |

We now add two additional BUY orders one with `price = 100.25` and `volume = 30` and one with `price = 99.50` and `volume = 45`. The order book will then look as follows:

| **Buy** | **Sell** |
|---|---|
| 30 @ 100.25 | 60 @ 101.50 |
| 50 @ 100.00 | |
| 45 @ 99.5 | |

We now add a SELL order with the `price = 99.75` and `volume = 50`. The order book will then look as follows:

| **Buy** | **Sell** |
|---|---|
| 30 @ 100.00 | 60 @ 101.50 |
| 45 @ 99.50 | |

We now add an additional SELL order with the `price = 100.00` and `volume = 50`. The order book will then look as follows:

| **Buy** | **Sell** |
|---|---|
| 45 @ 99.50 | 20 @ 100.00 |
| | 60 @ 101.50 |

We now add a BUY order with the `price = 99.50` and `volume =60`. The order book will then look as follows:

| **Buy** | **Sell** |
|---|---|
| 45 @ 99.50 | 20 @ 100.00 |
| 60 @ 99.50 | 60 @ 101.50 |

Output example of a program implementing order book:

```
Processing started at: 2019-09-06 10:03:24.123

book: book-1
            Buy -- Sell
==================================
       89@99.90 -- 13@100.30
        3@99.90 -- 84@100.30
       24@99.90 -- 62@100.40
       50@99.80 -- 45@100.40
       40@99.70 -- 53@100.50
       21@99.70 -- 88@100.50
       71@99.60 -- 84@100.50
       85@99.60 -- 85@100.50
       50@99.60 -- 76@100.50
       67@99.60 -- 20@100.60
       69@99.60 -- 34@100.60
       85@99.50 -- 55@100.70
       50@99.50 -- 63@100.80
        1@99.50 -- 38@100.90
        2@99.40 -- 69@100.90
       66@99.40 -- 35@100.90
       88@99.10 -- 73@101.00
       57@99.10 -- 19@101.20
       41@99.10 --

book: book-2
            Buy -- Sell
==================================
      11@100.00 -- 10@100.20
      20@100.00 -- 58@100.30
      54@100.00 -- 57@100.30
       20@99.90 -- 34@100.30
       44@99.90 -- 84@100.40
        8@99.80 -- 52@100.50
       14@99.80 --  9@100.50
       90@99.80 -- 45@100.50
       51@99.70 -- 54@100.60
       59@99.70 -- 90@100.60
       42@99.70 -- 94@100.60
       57@99.70 -- 48@100.60
       18@99.60 -- 35@100.60
       93@99.60 -- 96@100.70
       17@99.50 -- 89@100.70
       12@99.50 -- 68@100.80
       81@99.40 -- 53@100.80
       52@99.40 --  7@100.90
       45@99.30 -- 70@100.90
       10@99.30 --
       87@99.10 --

book: book-3
            Buy -- Sell
==================================
       99@99.90 -- 36@100.10
       85@99.90 -- 13@100.30
       56@99.80 -- 63@100.30
       56@99.60 -- 25@100.30
       17@99.60 -- 89@100.30
       38@99.60 -- 72@100.40
       66@99.60 -- 60@100.50
       70@99.60 -- 84@100.60
       53@99.50 -- 32@100.60
        4@99.50 -- 25@100.60
       99@99.50 -- 25@100.60
       51@99.50 -- 83@100.70
       82@99.40 -- 22@100.80
       10@99.40 -- 55@100.90
       60@99.40 -- 17@100.90
       46@99.20 -- 53@101.00
       86@98.80 -- 86@101.10
                    1@101.30
                   43@101.40

Processing completed at: 2019-09-06 10:03:24.123
Processing Duration: ### seconds
```