Overview

A matching engine facilitates trading financial instruments at an exchange by taking order requests from clients and matching them against one another to produce trades. We would like you to write an application that takes a sequence
of order requests (add and remove), runs them through a matching engine, and produces messages describing the trades and order state changes that result from the matching operation.

The “Details” section of this document gives information about how your application should match orders. The “Messages” section defines the input consumed by and output produced by your application. The “Example” section contains a simple example sequence of messages (both input and output). Finally, the “Problem” section provides details on the requirements for your application.

Details

Orders are offers by market participants to buy (sell) up to a given quantity at or below (above) a specified price. Generally there are a number of orders to buy at low prices and a number of orders to sell at high prices since everybody is trying to get a good deal. If at any point the exchange has a buy order at the same or higher price than a sell order, the two orders are "matched" with each other producing a trade for the smaller of the two order quantities. An incoming order message that causes the buy and sell prices to cross is called an aggressive order. The orders that are already in the market are called resting or passive orders.

If there are multiple resting orders that could be matched with the aggressive order, the resting orders are prioritized first by price and then by age. For price, priority goes to the resting order that gives the best possible price to the aggressive order (i.e., the lowest-price resting sell for an aggressive buy or the highest-price resting buy for an aggressive sell). When there are multiple resting orders at the best price, the resting order that was placed first (the oldest resting order) matches first.

Once the correct resting order has been identified, a trade occurs at the price of the resting order, and for the smaller of the quantities of the two orders (aggressive and resting). A trade "fills" the orders involved - fully if the trade quantity is equal to the order quantity and partially if the order quantity is larger. A trade will always fully fill at least one of the orders.
A partial fill leaves the remaining quantity available to be filled later.

If matching partially fills the aggressive order, the matching procedure continues down the list of resting orders in the sequence determined by the price-then-age priority defined above until either the aggressive order is fully filled, or there are no more resting orders that match. In the latter case, if there is still quantity on the aggressive order, the remaining quantity becomes a resting order in the book.

For example, if the resting orders are:

```
price  orders (oldest to newest, B = buy, S = sell)
1075   S  1            // one order to sell up to  1 at 1075
1050   S 10            // one order to sell up to 10 at 1050
1025   S  2   S  5     // one order to sell up to  2 at 1025,
                       // and second newer order to sell up to 5

1000   B  9   B  1     // buy up to 9 at 1000, second to buy up to 1
975    B 30            // buy up to 30 at 975
```

The best buy order is at a price of 1000 and the best sell order is at a price of 1025. Since no seller is willing to sell low enough to match a buyer and no buyer is willing to buy high enough to match a seller there is no match
between any of the existing orders.

If a new buy order arrives for a quantity of 3 at a price of 1050 there will be a match. The only sells that are willing to sell at or below a price of 1050 are the S10, S2, and S5. Since the S2 and S5 are at a better price, the new order will match first against those. Since the S2 arrived first, the new order will match against the full S2 and produce a trade of 2. However, the 1 remaining quantity will still match the S5, so it matches and produces a trade of 1 and the S5 becomes an S4. Two trade messages will be generated when this happens, indicating a trade of size 2 at price 1025 and a trade of size 1 at price 1025. Two order-related messages will also be generated: one to remove the S2, and one to note the modification of the S5 down to an S4. The new set of standing orders will be:

```
price  orders
1075   S  1
1050   S 10
1025   S  4

1000   B  9   B  1
975    B 30
950    B 10
```

Note that if a new sell order arrives at a price of 1025 it will be placed to the right of the S4 (i.e. behind it in the queue). Also, although there are only a few price levels shown here, you should bear in mind that buys and sells can arrive at any price level.

Messages

MessageType

There are five types of message involved in this problem, two messages your application shall accept as input and three that it will produce as output. The
message types are identified by integer IDs:

```
0: AddOrderRequest (input)
1: CanceOrderRequest (input)
2: TradeEvent (output)
3: OrderFullyFilled (output)
4: OrderPartiallyFilled (output)
```

Input

There are two input message types (requests):

```
AddOrderRequest: msgtype,orderid,side,quantity,price (e.g. 0,123,0,9,1000)
  msgtype   = 0
  orderid   = unique positive integer to identify each order;
              used to reference existing orders for remove/modify
  side      = 0 (Buy) or 1 (Sell)
  quantity  = positive integer indicating maximum quantity to buy/sell
  price     = double indicating max price at which to buy/min price to sell

CancelOrderRequest: msgtype,orderid (e.g. 1,123)
  msgtype   = 1
  orderid   = ID of the order to remove
```

Output

```
TradeEvent: msgtype,quantity,price (e.g. 2,2,1025)
  msgtype   = 2
  quantity  = amount that traded
  price     = price at which the trade happened
```

Every pair of orders that matches generates a TradeEvent. If an aggressive order has enough quantity to match multiple resting orders, a TradeEvent is output for each match.

```
OrderFullyFilled: msgtype,orderid (e.g. 3,123)
  msgtype   = 3
  orderid   = ID of the order that was removed
```

Any order that is fully satisfied by a trade is removed from the book and no longer available to match against future orders. When that happens, an OrderFullyFilled message is output. Every TradeEvent should result in at least
one OrderFullyFilled message.

```
OrderPartiallyFilled: msgtype,orderid,quantity (e.g. 4,123,3)
  msgtype   = 4
  orderid   = ID of the order to modify
  quantity  = The new quantity of the modified order.
```

Any order that has quantity remaining after a trade is only partially filled and the remaining quantity is available for matching against other orders in the future. When that happens, an OrderPartiallyFilled message is output. Each TradeEvent should result in at most one OrderPartiallyFilled message.

Problem

(1) Given a sequence of order messages as defined above, construct an in-memory representation of the current state of the order book. You will need to generate your own dataset to test your code. Messages must be read from stdin.

(2) When an aggressive order matches one or more resting orders, write the sequence of trade messages and order removals/modifications that result to stdout. If there are multiple trades, the output should be in the same sequence
as the resting orders were matched. For every pair of orders matched, three messages should be output in the following sequence:

1. The TradeEvent
2. A full or partial fill message for the aggressive order
3. A full or partial fill message for the resting order

(3) Your code should be clean, easy to understand, efficient, robust, and well designed (appropriate abstractions, code reuse, etc.). No input should cause your application to crash; any errors should be be clearly logged to stderr.

(4) You are responsible for testing your solution. Please include any datasets
and/or supporting code that you used in your testing.

(5) Please include a description of the performance characteristics of your solution, especially for determining whether an AddOrderRequest results in a match, removing filled orders from the resting order book, and removing a cancelled order (due to a CancelOrderRequest) from the orderbook.

Example

This is a complete annotated example of input and output corresponding to the example described in the Details section.

NB: Comments are not part of the input or output, they are just used to add clarity to the example.

Input/Output:

The following messages on stdin:

```
0,100000,1,1,1075
0,100001,0,9,1000
0,100002,0,30,975
0,100003,1,10,1050
0,100004,0,10,950
BADMESSAGE          // An erroneous input
0,100005,1,2,1025
0,100006,0,1,1000
1,100004            // remove order
0,100007,1,5,1025   // Original standing order book from Details
0,100008,0,3,1050   // Matches! Triggers trades
```

The final message should cause your program to produce the following ouput on stdout:

```
2,2,1025
4,100008,1
3,100005
2,1,1025
3,100008
4,100007,4          // order quantity reduced for partial fill
```

In response to the erroneous input, your application should write some kind of
error message on stderr, maybe like this:

```
Unknown message type: BADMESSAGE    // Your error message may vary
```

Notes

The input messages up through order 1000007 build the initial book from the example in the Details section above. Note that order 100004 is removed, so it doesn't appear in the Details section. None of these order messages cause a match, so they are just added to the resting order book and generate no output on stdout, and a single error message on stderr.

The next order (ID 100008, a buy of 3 at 1050) triggers the matching engine to produce a trade. When this message is processed, two trades are produced from three orders interacting (the newest order and two of the orders that are resting in the order book). The output from this event should explicitly state each trade as well as posting how the involved orders are affected by the match. In this case, the inbound order is fully filled, so it no longer affects the book (so it is removed), as is one of the two resting orders. The other resting order is only partially filled, so part of the order is still resting in the book, represented as a modify message in the output.