In this project, you will implement one of the foundational pieces that drive capital markets. Billions of dollars worth of assets are traded on exchanges worldwide daily. Most of those exchanges operate an [order book](https://en.wikipedia.org/wiki/Order_book_(trading)) for each instrument that can be traded.

This project is more than just implementing a single function where we test correctness. You will first need to synthesize a possibly foreign subject matter (how order books work), then craft a design based on the description provided, and finally ensure correctness.

We will test for correctness in this project via automated test cases. Inputs are fed into your program via CSV files. Your program will need to generate the correct CSV outputs.

This entire document should be read, sequentially, before you start coding and/or designing.

**Definitions**

First, we should review some definitions before diving into more specifics.

1. **Orders**  
   An order is an instruction a person gives to either buy or sell something. Orders have several required fields. Critically, an order has the price at and size (number of shares) you are willing to buy or sell.
2. **Bids**  
   "Bids" is the common name used to identify buy-orders. A bid is a single buy-order. Bids are a set of buy-orders.
3. **Asks**  
   "Asks", or "offers", is the common name used to identify sell-orders. An ask is a single sell-order. Asks are a set of sell-orders.
4. **Price**  
   The price specified on a bid or ask. The price specified on a bid is the maximum price a buyer is willing to pay to purchase the stock. The price specified on an ask is the minimum price a seller is willing to pay to sell the stock. Naturally, a buyer will be willing to buy for a price less than their bid, and a seller will be willing to sell at a higher price than their ask.
5. **Trades**  
   The result of a bid-price being ≥ to an ask-price results in a *trade*. The amount traded is always equal to the ask-price.
6. **Order Book**  
   An [order book](https://en.wikipedia.org/wiki/Order_book_(trading)) is a collection of bids and asks, where the bids are sorted such that the top order has the highest price, and the asks are sorted such that the top has the lowest price. The purpose of an order book is to manage the life-cycle of all buy-orders and sell-orders. When an order book processes an incoming order, it has a predefined matching algorithm to figure out what to do with the order.
7. **BBO**  
   The top bid (highest buy price) is called *best-bid*, and the top ask (lowest sell price) is called *best-offer*. The *BBO* is the pair of both the best-bid and best-offer, called the *best-bid-offer*, or *BBO* for short. Note that the BBO can contain multiple buy orders (or sell orders), if there are multiple orders at the same price.

**Implementation**

Your job is to implement an order book that can manage the life cycle of orders. Orders can either be *inserted* or *cancelled*. An order that is inserted can update the BBO, and possibly generate trades depending on the price of the order. An order that is cancelled can possibly update the BBO, again depending on the price of the order. Once an order is inserted, the outstanding size of the order can change if it is partially fulfilled. If an order that was previously inserted is completely fulfilled, it is removed from the order book.

You will read insert or cancel requests from a CSV file and apply them to your order book using your matching algorithm, described below.

**Matching Algorithm**

An order-book's *matching algorithm* is responsible for generating trades when an incoming order's price locks or crosses the opposing side of the order book. For example, if the incoming order is to buy 100 shares at $1.00, then the order book will first check if there is an existing sell order with a price less-than-or-equal to $1.00. If there is, then a trade should be generated between the buyer and seller. If not, it will add the buy order to the list of existing bids yet to be fulfilled. When you insert an order, you must ensure that your order book does not become locked or crossed. A locked or crossed order book is where the best-bid price ≥ best-ask price. That should never happen because at least one of the crossing orders should have been fulfilled. A locked or crossed book should never happen if the matching algorithm is working correctly.

Orders can also be partially filled and fulfilled at different prices. For example, consider a bid for 10 shares at $10 with the following three asks already in the order book: (a) 2 shares at $8, (b) 3 shares at $8, and (c) 2 shares at $9. Then, three trades would be generated: (a) 2 shares at $8, (b) 3 shares at $8, and (c) 2 shares at $9. In this case, the bid would have 3 shares remaining at $10. It will then be placed on the bid side of the order book for the remaining 3 shares at $10.

Your matching algorithm should be implemented with price *then* time priority. This means if you receive two insert orders with the same price, the order with the earliest timestamp takes priority since the prices are the same. If the price and timestamp are the same, then priority should match the ordering in the input file, i.e. whichever order you read first takes priority. This will impact your matching algorithm: if an incoming buy order's price ≥ the existing sell price, and there are multiple sell orders with the same price, then you will match the incoming buy order to the sell order with the earliest timestamp first.

**Example**

Given the following state of an order book with 4 asks and 2 bids:

| **Side** | **Timestamp** | **OrderID** | **Price** | **Size** |
| --- | --- | --- | --- | --- |
| sell | 1602556616 | 999 | 200 | 50 |
| sell | 1602556615 | 998 | 200 | 50 |
| sell | 1602556611 | 997 | 200 | 50 |
| sell | 1602556611 | 996 | 150 | 25 |
| buy | 1602556609 | 888 | 125 | 50 |
| buy | 1602556619 | 887 | 120 | 50 |

Our best-bid is 50 shares @ $125, and our best-ask is 25 shares @ $150. If we inserted buy 100 shares @ $200 (orderID 1000, timestamp 1602556620) then, we will lock/cross with existing sell orders. After applying the matching algorithm, the resulting order book should look like:

| **Side** | **Timestamp** | **OrderID** | **Price** | **Size** |
| --- | --- | --- | --- | --- |
| sell | 1602556616 | 999 | 200 | 50 |
| sell | 1602556615 | 998 | 200 | 25 |
| buy | 1602556609 | 888 | 125 | 50 |
| buy | 1602556619 | 887 | 120 | 50 |

With the following trades being generated:

* 25 shares @ $150, sell orderID 996, buy orderID 1000
* 50 shares @ $200, sell orderID 997, buy orderID 1000
* 25 shares @ $200, sell orderID 998, buy orderID 1000

And a new best-ask of 75 shares @ $200 (sum of all shares at the $200 price-level).

Since orderID 996 has the lowest ask price and is lower than the bid price, we fulfill it first, removing it from the order book and leaving 75 remaining shares in orderID 1000. Since orderIDs 997 and 998 have the same ask price and both are lower than the bid price, we use timestamp as a tiebreaker. OrderID 997 has an earlier timestamp, so we fulfill it, removing it from the order book and leaving 25 remaining shares in orderID 1000. Now, since OrderID 998 has the lowest ask price in the order book, we fulfill the remaining 25 shares of orderID 1000, leaving 25 shares remaining in orderID 998. Since OrderID 998 was only partially fulfilled, it remains on the order book with a reduced size of 25 shares. The incoming order, orderID 1000, is completely fulfilled so is never added to the order book.

If the incoming order had been to buy, say, 1000 shares instead of 100, then all the sell orders would have been fulfilled, leaving no sell orders, and the 825 outstanding shares of the incoming order would be added to the order book, resulting in the following order book:

| **Side** | **Timestamp** | **OrderID** | **Price** | **Size** |
| --- | --- | --- | --- | --- |
| buy | 1602556620 | 1000 | 200 | 825 |
| buy | 1602556609 | 888 | 125 | 50 |
| buy | 1602556619 | 887 | 120 | 50 |

**Inputs/Outputs**

As mentioned earlier, your application will be provided a path to a CSV file that contains order inserts and cancels. As you process this input, you should generate two CSV output files.

The first output file is a BBO output file. This file should end up with the same number of lines as your input file. Each time you insert or cancel an order, you should unconditionally write a row to this file with the current BBO information.

The second output file is a trade output file. Each time you insert an order, if any trades are generated as a result, you should write them to this file.

To facilitate automated testing, all CSV files should follow these conventions:

* The first line should always have the headers
* No extraneous spaces
* No leading commas after the last column
* A newline with no other characters as the last line in the file

Note: We will *not* test against input sanitization. Your application can assume that input CSV files are well-formed, and that they adhere to the specifications found below.

**Input File Specification**

The input CSV file your application reads will have orders that are either being inserted into the order book or cancelled. All prices are in cents and stored as ints. Timestamps will increase monotonically (i.e. each successive timestamp will be ≥ the previous). The shape of the CSV is as follows:

| **Column** | **Description** | **Type** |
| --- | --- | --- |
| timestamp | Timestamp of when the order was created, seconds since UNIX epoch | int |
| action | Either insert or cancel. | string |
| order\_id | Unique ID for the order | int |
| side | Either buy or sell | string |
| price | The price for the order | int |
| size | The size for the order | int |

Note: If the action is cancel, then side, price, and size are ignored; those columns should be empty. If there is not an existing insert with the same order\_id, the cancel should have no effect on the order book.

**BBO Output File Specification**

The BBO output your application produces should have one line for every line in the input CSV file. After you process an order in the input file, your application should write a row in the BBO output file that shows the current BBO. The shape of the CSV is as follows:

| **Column** | **Description** | **Type** |
| --- | --- | --- |
| bid\_price | Best bid-price | int |
| bid\_size | Net size on the best bid-price level | int |
| ask\_price | Best ask-price | int |
| ask\_size | Net size on the best ask-price level | int |

Note: If there are no orders on a particular side, then both the price and size for that particular side should be set to 0.

**Trade Output File Specification**

The Trade output your application produces will show a line everytime a trade occurs in the order book. The shape of the CSV file is as follows:

| **Column** | **Description** | **Type** |
| --- | --- | --- |
| trade\_price | The price traded | int |
| trade\_size | The size traded | int |
| buy\_order\_id | The order\_id that bought | int |
| sell\_order\_id | The order\_id that sold | int |

**Getting Started**

This project is encapsulated in its own filesystem. You can freely browse the filesystem to discover sample tests that we have provided for you. Importantly, depending on the language, you should find an app.py (if you're in python3) or src/main/java/com/app/App.java (if you're in java). This is where the test driver calls your implementation.

Important: Each invocation of your application should maintain no state between runs. The test driver assumes no memory is maintained between successive calls to your application.

There are 11 tests for this project located at path tests for python3 and src/tests/resources for java. The first tests 1-5 are "sample" test cases where we provide both the input and output files. These tests are visible to help you understand the input/output formats, in addition to understanding the baseline functionality.

With tests 6-11, both the input and output files are hidden from you. When you hit "Run" within the CodeSignal IDE, only the basic tests are executed which use the "sample" test cases. Once you hit "Submit" in the CodeSignal IDE, both the basic and advance tests are executed. The advanced tests test against the hidden test cases, but only the final score will be shown. The test output will not be visible.

You are able to "Submit" multiple times. Only your final submission will be viewable to us. Once you have submitted your final solution, please navigate back to the tasks screen and select "Finish The Test".

Your output files are verified using a simple diff between the expected output and the output you generated. Therefore, it's important you pay attention to how your output is formatted to avoid test failures due to bad CSV formatting.

We encourage you to write your own test cases before submitting your task. You can set your own test entry points by opening the dropdown next to "RUN" and clicking "Run Custom Script" in the CodeSignal IDE. This will give you a bash script as your entrypoint where you can invoke any custom tests you build. For example, if I'm using Java + Maven and I only want to invoke test1 from AppTest, my bash script would look like:

mvn test -Dtest=AppTest#test1

In python, the bash script would look like:

python3 -m unittest discover './tests' -p '\*.py' -k 'test01'

**Resources**

If you need further help understanding the business logic behind an order book, referencing a live example might be helpful. Crypto exchanges often feature animated order books and provide free sandbox environments where you can play with inserting / cancelling orders. [Coinbase](https://pro.coinbase.com/trade/BTC-USD) is one such exchange.