# In-Memory Implementation of
# Order Book & Matcher
# For
# Jump Trading, Chicago

**Submitted By - Rupinder Ghotra**

**Waterloo, Canada**

# Executive Summary

A fast, efficient, and reliable orderbook is the first practical step to build a fully functional automated exchange. In order to achieve the given mandate of low-latency, highly-robust and maintainable order book I chose to go with the data structure of List of Structs. Further, each Struct in the list has a list of struct as a data member.

Choice of data structure plays a key role in resolving complex problems relatively easier. Different types of information could be represented in several ways.

I considered several data structures in order to come up with the most efficient in terms of access - insertion, deletion, searching - and user friendly in terms of readability(for debugging) and maintenance. After analysis - I decided to choose **LinkedList** implementation - which provides the most efficient solution. But for the purpose of validation of my solution, I am also submitting another implementation but in less efficient data structure - **Set.** LinkedList vs Set implementation's performance is measured and is included in this report.

# Design

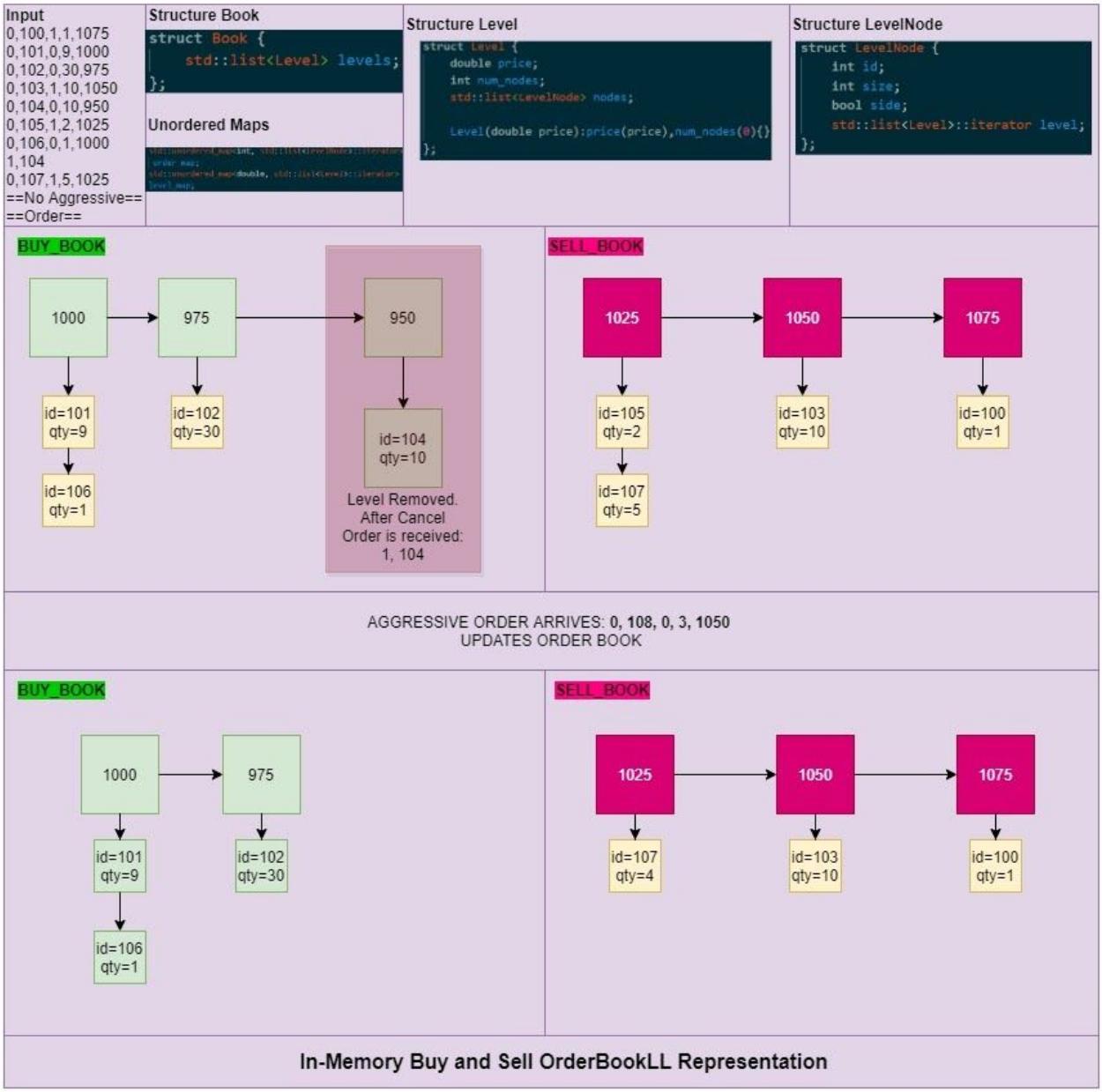
The overall flow of the program is as follows. The main executable reads lines from standard input and passes each line to the MessageParser, which parses the input and, if the input parses correctly, passes the appropriate message to OrderBook. OrderBook is responsible for maintaining the book for buy and sell limit orders and detect trades and notify ExecutionListener of trades and fills.

The OrderBook and ExecutionListener classes are abstract to support different implementations. We provide two implementations of OrderBook:

- OrderBookRBT, a set based implementation whose correctness is easy to verify from definition, whose operations take O(log n) time where n is the number of orders already in the book. We used this implementation mainly for verification.
- OrderBookLL, a linked list based implementation where each operation takes O(1) time. Further details are provided in the next section.
- ExecutionListener, an abstract class implemented by ExecutionLogger class to display output from the OrderBook.
- MessageParser, a standard class being used to receive incoming orders and send the data for validation and triggers OrderBook methods accordingly.
- Utility: Data validation and conversion class used by MessageParser class to validate incoming orders.

# Data Structure & Algorithm

In OrderBookLL, we group the open limit orders of buy and sell side into levels based on price. The book is represented as a linked list of levels (ordered by price), and each level in turn consists of a linked list of nodes corresponding to individual orders (ordered by recency). In addition to these lists, we maintain two hash tables (order_map and level_map) to lookup the node from an order's id and the level from price.A simple in-memory representation of OrderBook using LinkedList data structure is shown below.



**Add Order**: When a buy order arrives, we iterate over the sell side book to find existing orders that match with this order. If the sell order is completely filled by current order, we remove its node from the sell side book. If it was the last sell order in its level, we remove that level and continue to the next level. We continue to iterate until either the buy order is completely filled or we reach a sell order whose price exceeds the buy order's price. If the buy order has some size remaining after trades, the remaining size is added to the buy side book. Sell orders are handled analogously.

**Cancel Order**: When a cancel order arrives, we lookup the order's id in order_map and remove the node if it exists. As when removing nodes due to trade add order, we ensure that if this was the last node for the level, we remove that level from the book.

# Logging

A logger class name Logger.h is implemented and inherited by OrderBookLL, OrderBookRBT and MessageParser classes.

Four logging levels can be set in **main.cpp's line number 9**. Each level is self-explanatory as per their names. By-default, logging level is set to **ERROR.**

1) ERROR
2) INFO
3) DEBUG
4) WARNING

# Testing

## Concerned Directories & Files

- **/testData** = Contains files related to testing of application.
- **/testData/input** = Suite of 12 test cases covering smoke testing of application.
- **/testData/expected** = Suite of expected output of test-runs executed by test-runner.sh
- **test-runner.sh** = A bash script to execute testSuite in /testData/input dir. Test-runner.sh iterates over the files of /input dir and feeds the content of incoming test files to the application and generates corresponding output files.test-runner.sh then compares the output data from test run with the corresponding expected output file in /testData/expected directory. If the data matches - test passes else test is marked as failure.

Smoke tests can be triggered comfortably by triggering the test-runner.sh file using bash on Linux and Windows-Subsystem-For-Linux(WSL). Therefore - making it platform independent.

### Test Case Description

| Test Number | Test Case Name | Test Case Description |
|---|---|---|
| **0** | **SimpleInput** | simple input triggering order matcher |
| **1** | **SimpleInputWithError** | simple input triggering order matcher and garbage input |
| **2** | **OrderGenTradeInMultipleOrders** | order causing trade in multiple orders |
| **3** | **OrderGenTradeInMultipleLevels** | order causing trade in multiple orders in multiple levels(price) |
| **4** | **GarbageInput** | garbage input (invalid code, not csv, not convertible to int/double etc) |
| **5** | **InvalidPrice** | input with invalid price (price <= 0) |
| **6** | **InvalidSize** | input with invalid size (size <= 0) |
| **7** | **InvalidOrderId** | input with invalid id (id <= 0) |
| **8** | **AddOrderWhichExists** | add_order with id that already exists in system |
| **9** | **CancelOrder** | cancel_order with arbit id |
| **10** | **CancelCompletedOrder** | cancel_order with completed order |
| **11** | **OutOfRangeInput** | out of range input |

## Performance Characteristics

Both asymptotic and practical performance experiments were conducted and readings are incorporated.

### Asymptotic Performance Analysis

Theoretical performance of both implementations is given below. Maintaining order_map and level_map enabled me to achieve constant time lookup.

n = size of input

| **OperationName** | **OrderBookLL** | **OrderBookRBT** |
|---|---|---|
| AddOrder | O(1) | O(log n) |
| CancelOrder | O(1) | O(log n) |
| MatchOrder | O(1) | O(log n) |

## Experimental Performance Analysis

### Setup

- Test suite will be run first using OrderBookLL implementation and then OrderBookRBT implementation.
- Each test is run three times on each implementation and runtime is captured in milliseconds.
- Average of three runs for each test case is calculated.
- Winner is declared accordingly.

### Machines

Three different machines were used to perform the experiment. OS and processor description is below:

| OS Name | Processor | Expermement Run # |
|---|---|---|
| Windows 10, 16GB | Intel Core i7-4810MQ CPU @ 2.80GHz | 1 |
| Windows 10 8GB | Intel Core i5-8250U CPU @ 1.60GHZ | 2 |
| Mac OSX 10.15, 8 GB | Intel Core i5 CPU @2.3GHZ | 3 |

### Readings:

Experimental readings can be found on the next page.

| Test # | OrderBookLL | | | | OrderBookRBT | | | |
|---|---|---|---|---|---|---|---|---|
| | Run1 | Run2 | Run3 | **Avg** | Run1 | Run 2 | Run3 | **Avg** |
| 0 | 1998 | 1161 | 484 | **1214** | 2009 | 471 | 155 | **878** |
| 1 | 1011 | 1310 | 218 | **846** | 1790 | 655 | 411 | **952** |
| 2 | 1999 | 1056 | 253 | **1102** | 1999 | 452 | 243 | **898** |
| 3 | 1998 | 1687 | 277 | **1320** | 1998 | 668 | 168 | **944** |
| 4 | 2985 | 2105 | 491 | **1860** | 2998 | 1709 | 473 | **1726** |
| 5 | 2998 | 1830 | 355 | **1727** | 2998 | 1693 | 333 | **1674** |
| 6 | 1999 | 1390 | 220 | **1203** | 2010 | 655 | 160 | **941** |
| 7 | 1998 | 1239 | 562 | **1266** | 1997 | 811 | 255 | **1021** |
| 8 | 1997 | 1028 | 274 | **1099** | 1998 | 444 | 187 | **876** |
| 9 | 1910 | 1454 | 420 | **1261** | 1998 | 864 | 543 | **1135** |
| 10 | 1999 | 811 | 180 | **925** | 1990 | 447 | 340 | **966** |
| 11 | 1010 | 1038 | 298 | **782** | 1998 | 1299 | 302 | **1199** |

Time: in milliseconds.

## Conclusion

We can observe from the readings that OrderBookRBT still performed better than OrderBookLL even though the asymptotic analysis tells the opposite story. The result is not a surprise because for a small input size Set can perform better than LinkedList implementation of OrderBook.
is still decently competitive. But this gap will increase as the size of the order book increases. For small input - the impact seems tiny but in the world of low-latency electronic trading every nanosecond matter. Therefore, OrderBookLL is the clear winner in the world of fintech.

In the end - I would like to say that I thoroughly enjoyed the given exercise and I would like to mention that I'm honored to be considered by Jump Trading to develop their supreme trading platform, tools, infrastructure and trading models.