Order Matching Engine

Swetha Shanmugam
Zhejian Jin
Fenglei Gu

Part 0: Introduction

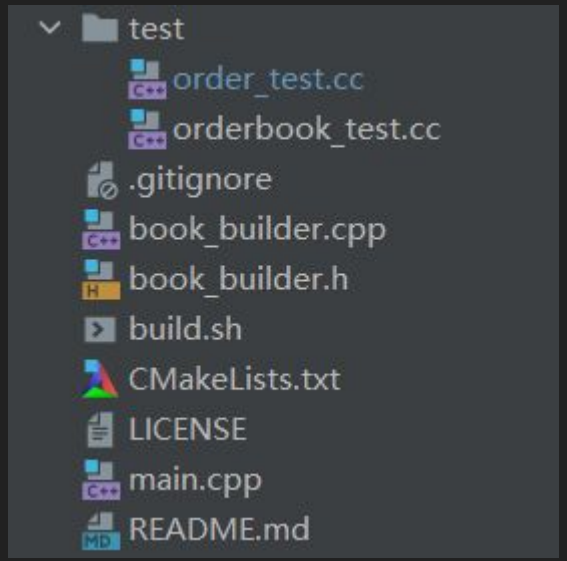
Intro to parser - 8 mins
Order book stuff - 15 mins

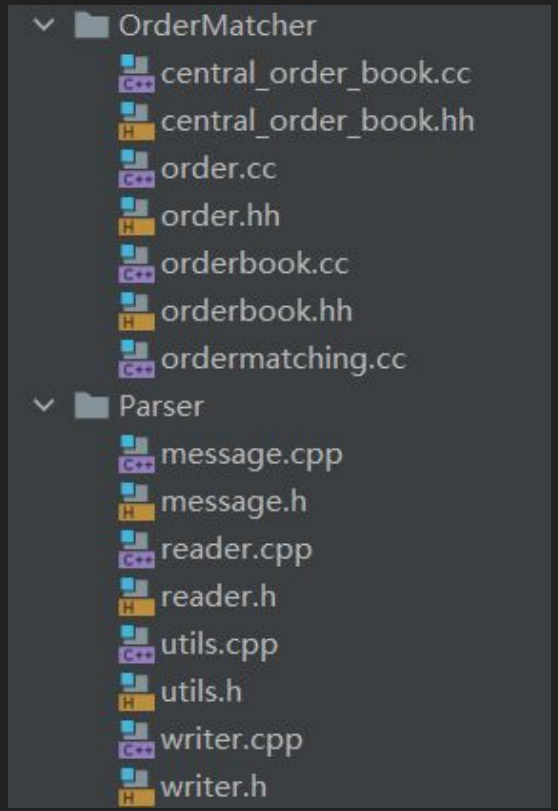
Exchange

- What is an exchange?
- What is electronic trading？
- What is an order?
- What is order matching?

Components

- Parser: takes financial data and parses it
- Order Matcher





Part 1: Parser

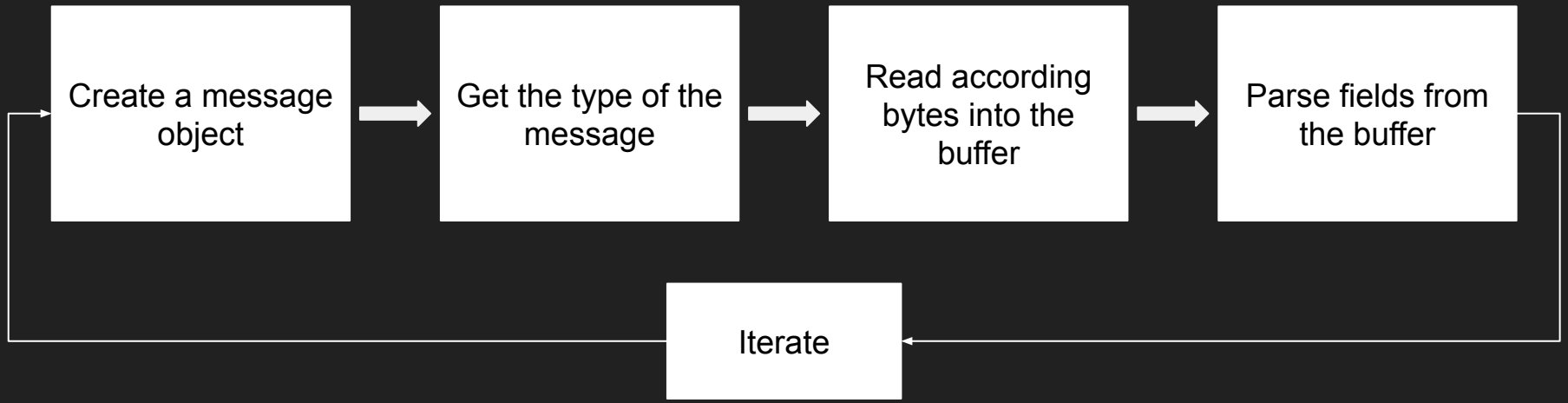
Parser

- Data Source: Nasdaq ITCH
  - Sample data: emi.nasdaq.com - /ITCH/Nasdaq ITCH
  - Specifications: NQTVITCHSpecification.pdf
  - SBE (Simple Binary Encoding)
- How the messages look like?
  - E.g. : Add Order Message
- ~ 61784531 messages in a 1.8G sample

Parser adapted:
https://github.com/martinobdl/ITCH

Add Order Message			
Name	**Offset**	**Length**	**Value**
Message Type	0	1	"A"
Stock Locate	1	2	Integer
Tracking Number	3	2	Integer
Timestamp	5	6	Integer
Order Reference Number	11	8	Integer
Buy/Sell Indicator	19	1	Alpha
Shares	20	4	Integer
Stock	24	8	Alpha
Price	32	4	Price (4)

Parser flow diagram - Reader Class



Parser - Reader Class

```
Reader(std::string fileName);

bool isValid() const;
virtual ~Reader();

Message createMessage();
bool eof();
void printProgress();
virtual void readBytesIntoMessage(const long &);
virtual void skipBytes(const long &);
virtual char getKey();
```

Parser Results

```
*;Ø◆◆O'RØ
eØ◆◆◆A        N dNCZ PN 1NN'RØ
eØ◆◆YAA       N dNCZ PN 1NØN'RØ
eØ◆$iAAAU     P dNQI PN 2YØN'RØ
eØ◆◆ØAABA     QNdNCQ PNN2NØN'RØ
eØ◆Ø◆AAC      N dNCZ PN 2NNØHØ
eØ◆Ø◆A        T     ØYØ
eØ◆'◆A        0'RØ
eØ◆◆=AADR     P dNQI PN 2YN'R
eØ◆CbAAL      QNdNCZ PNN1NN'R
eØ◆◆[AAMC     A dNCZ PN 2NN'R
eØ◆^%AAME     GNdNCZ PNN2NN'R
```

```
ticker:       CHK
Message type   :A
Id             :11211
timestamp      :28800007241329
Side           :1
Price          :31500
Remaining size :1000

ticker:       QQQ
Message type   :A
Id             :11212
timestamp      :28800013687753
Side           :0
Price          :1.7898e+06
```

Part 2: Order Matcher
2.1: Orders

Types of Orders

Market

- Executed at current market price
- Executed immediately (usually)

Types of Orders

Limit

- Executed at a particular limit price specified or better
- Filled only when the price is met

Types of Orders

Limit

- Executed at a particular limit price specified or better
- Filled only when the price is met

E.g:

Buy order - Price = **$100** -> Filled at a price <= **$100**

Sell order - Price = **$100** -> Filled at a price >= **$100**

Types of Orders

Stop/Stop-Loss

- Buy/Sell when market price reaches a “stop price”
- Order converted to a **Market Order**

E.g:

Sell order - Stop Price = **$90** Market Price = **$100**

-> Order NOT EXECUTED

Types of Orders

Stop/Stop-Loss

- Buy/Sell when market price reaches a “stop price”
- Order converted to a **Market Order**

E.g:

Sell order - Stop Price = **$90** Market Price = **$90**

-> Order EXECUTED

Types of Orders

Stop Limit

- Buy/Sell when market price reaches a “stop price”
- Order converted to a **Limit Order**

E.g:

Sell order - Stop price = **$90** Market Price = **$100**

Limit price = **$90**

-> Order NOT EXECUTED

Types of Orders

Stop Limit

- Buy/Sell when market price reaches a “stop price”
- Order converted to a **Limit Order**

E.g:

Sell order - Stop Price = **$90** Market Price = **$90**

Limit Price = **$90**

-> Order EXECUTED

unsigned prices

- Precision: $0.0001
- Range: $0 to $429,496.7295

```
#include <iostream>
int main(){
    float a = 2050.1234, b = 2050.1233;
    std::cout << (a-b) << "\n";
    return 0;
}
```

Order

`unsigned order_id`

`unsigned owner_id`

`unsigned quantity`

`unsigned quote`

`unsigned stop_price` - The stop_price in case of a STOP/STOP_LIMIT order

`OrderSide order_side` - The order side (BUY/SELL)

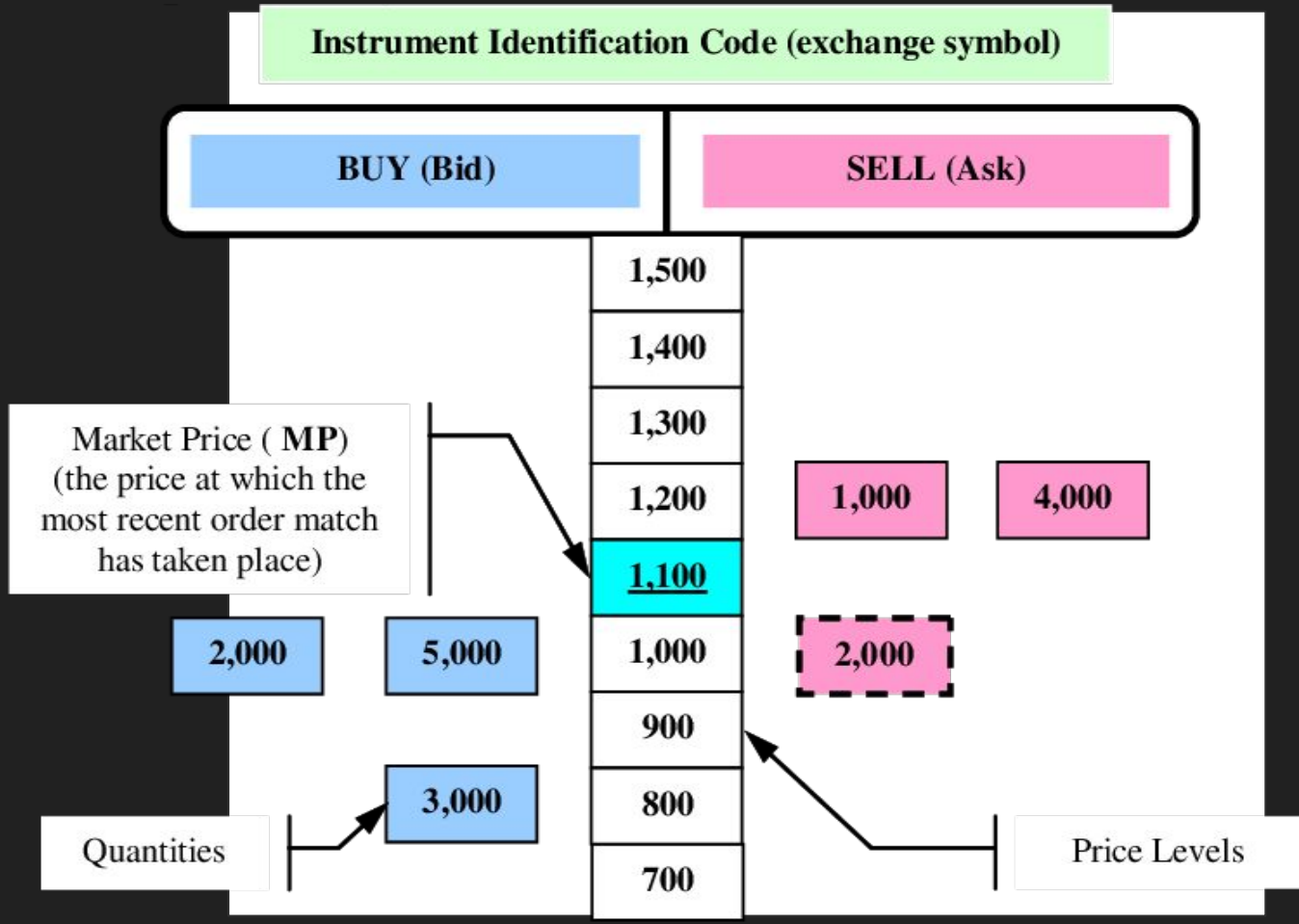
`OrderType order_type` - The order type (MARKET/LIMIT/STOP/STOP_LIMIT)

`bool all_or_none`

`std::chrono::time_point<std::chrono::system_clock> timestamp` - timestamp of added order

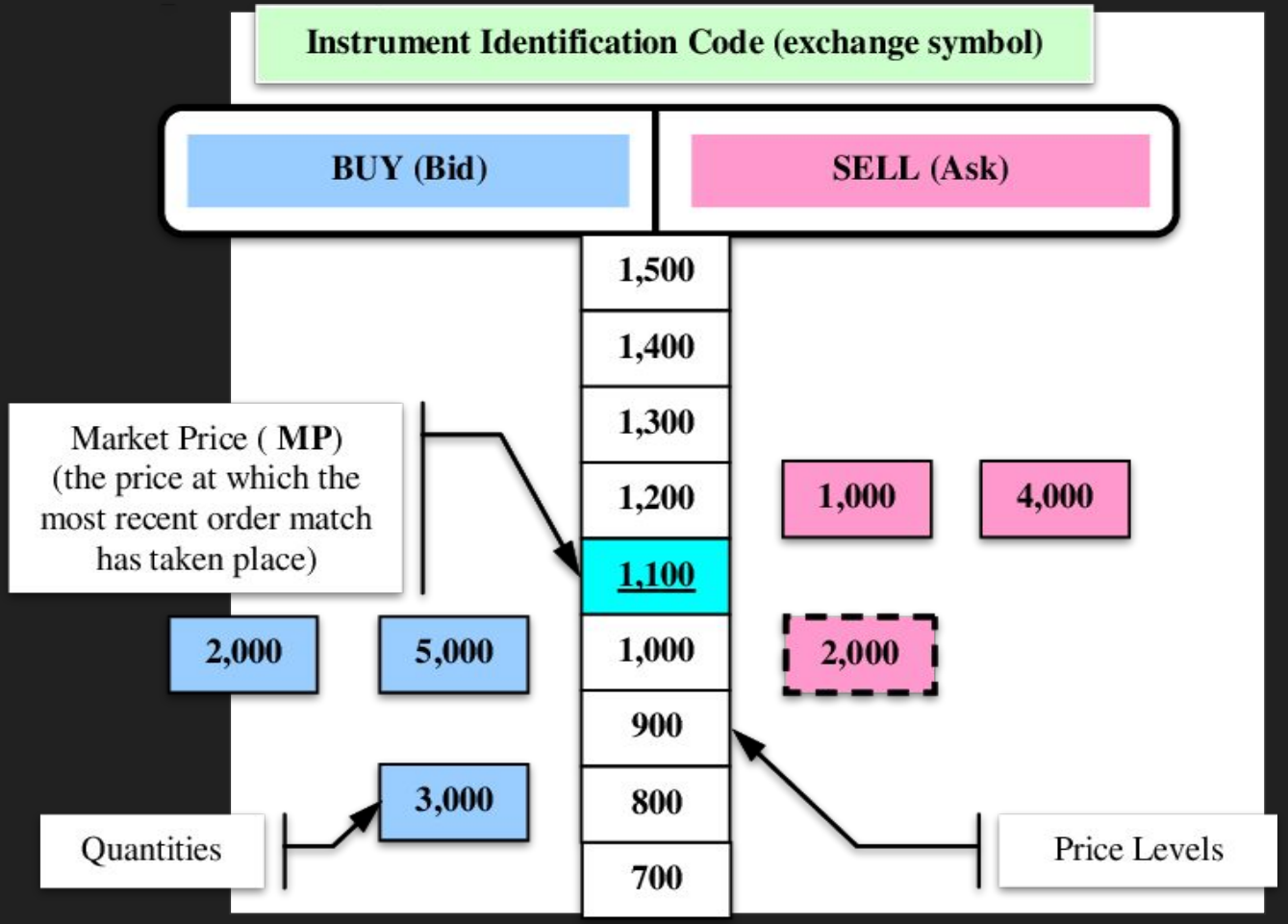
2.2: Order Book

OrderBook


Instrument Identification Code (exchange symbol)
BUY (Bid)
SELL (Ask)
1,500
1,400
1,300
1,200
1,100
1,000
900
800
700
Market Price (MP)
(the price at which the most recent order match has taken place)
1,000
4,000
2,000
2,000
5,000
3,000
Quantities
Price Levels

OrderBook

- Add Order
- Delete Order
- Get Order
- Get Best Bid
- Get Best Ask



Central Order Book - Members

```
//Hash map of symbol to order book
std::unordered_map<string, OrderBook> order_book_map;

//Hash map of order ID to stock symbol
std::unordered_map<unsigned int, string> order_ticket_map;
```

Central Order Book - Methods

```
StatusCode add_order(string stock, Order& order);

StatusCode delete_order(unsigned int order_id);

std::optional<Order> get_order(unsigned int order_id);

std::pair<StatusCode, unsigned> best_ask(string stock) const;

std::pair<StatusCode, unsigned> best_bid(string stock) const;
```

Order Book - Members: Naive Implementation

```
// key=price level; value=a list of Order
std::map<unsigned, std::list<Order>> buypool, sellpool,
stop_buy_pool, stop_sell_pool;
```

- Get min/max price: O(1)
- Insert: O(log N)

Order Book - Members: Our Implementation

```
// stores current price levels of the order pools
std::set<unsigned, std::less<unsigned>> sell_prices;
std::set<unsigned, std::greater<unsigned>> buy_prices;
```

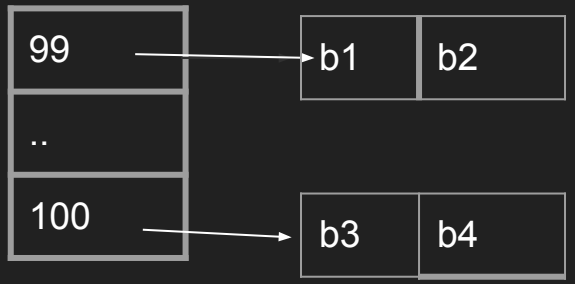
- Get min/max price: O(1)

Order Book - Members: Our Implementation

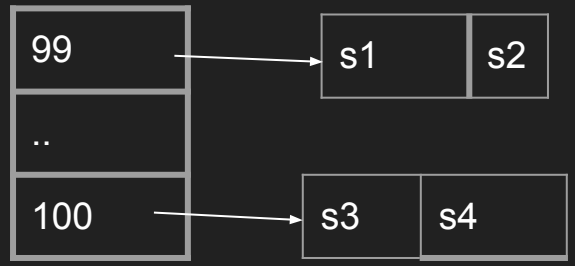
```
// key=price level; value=a list of Order
std::unordered_map<unsigned, std::list<Order>> buypool, sellpool;
```

- Insert: O(1)

Buy Pool



Sell Pool



Maintaining `std::set`s

- Update sell_prices/buy_prices when the order is the first or the last order at that price level
- O(log N) per update
- 1 update per M orders

Why `std::unordered_map` + `std::set`?

Assume: N price levels, M orders per level

Add new order:

- If buypool.count(order.quote)==0 : possibility = 1/M
  - buyprices.insert(order.quote): O(log L)
  - Average: O(1/M * log L)
- Access buypool[order.quote]: O(1)
- Insert order: O(1)
- O(1/M * log L)

If `std::map` pool…

- No need to maintain buyprices (pool keys ordered)
- Access buypool[order.quote]: O(log L)
- Insert: O(1)
- O(log L)

- unordered_map: O(1/M * log L)

Why `std::unordered_map` pool?

Delete an order:

- Access buypool[order.quote]: O(1)
- Erase order: O(M)
- If buypool[order.quote].empty() : possibility = 1/M
  - buypool.erase(order.quote): O(1)
  - buyprices.erase(order.quote): O(log L)
  - Average: O(1/M * log L)
- O(M + 1/M * log L)

If `std::map` pool…

- Access buypool[order.quote]: O(log L)
- Erase: O(M)
- O(M + log L)

- unordered_map: O(M + 1/M * log L)

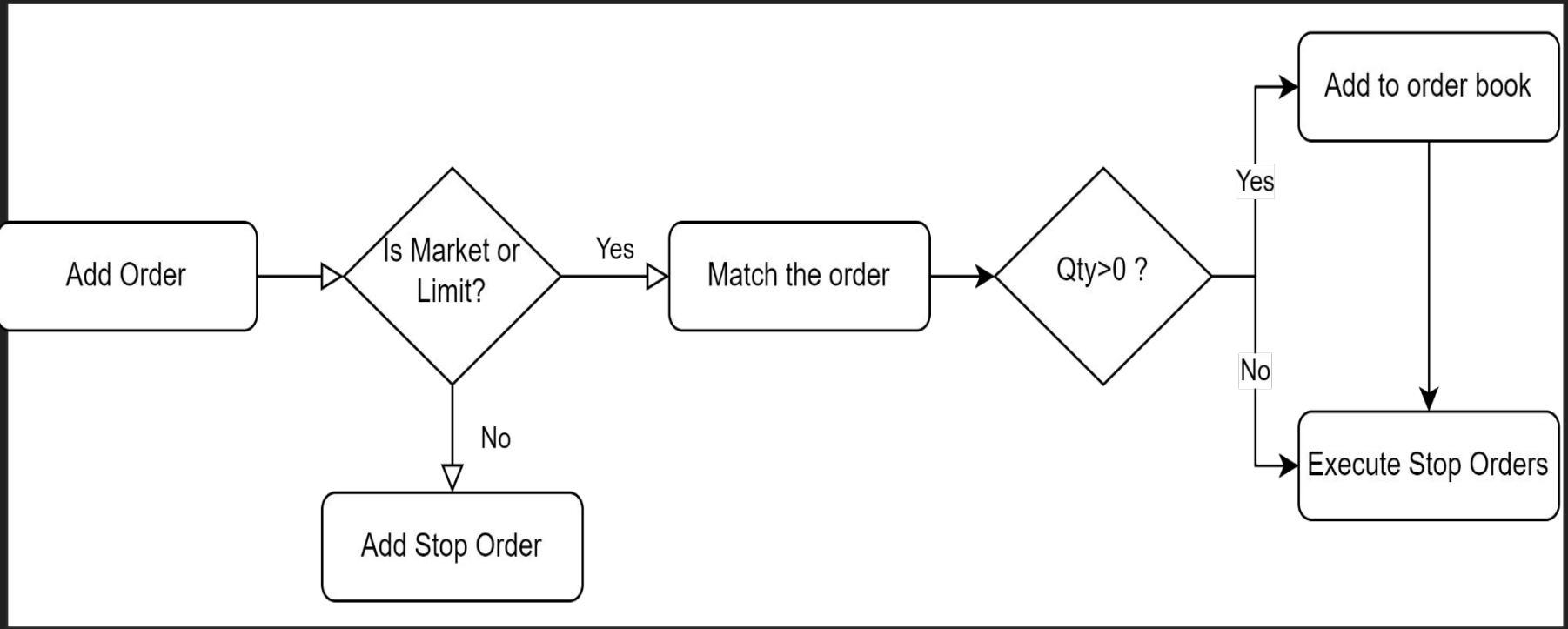
Order Book - Members (Stop-Loss Orders)

```
// stores current price levels of the order pools
std::set<unsigned, std::less<unsigned>> stop_buy_prices;
std::set<unsigned, std::greater<unsigned>> stop_sell_prices;

// key=price level; value=a list of Order
std::unordered_map<unsigned, std::list<Order>> stop_buy_pool,
stop_sell_pool;

// key=order ID, value=(orderside, price level, ordertype)
std::unordered_map<unsigned, OrderInfo> order_map;
```

Order Book - Methods

```
StatusCode add_order(Order& order);

StatusCode delete_order(unsigned int order_id);

std::optional<Order> get_order(unsigned int order_id);

std::pair<StatusCode, unsigned> best_ask() const;

std::pair<StatusCode, unsigned> best_bid() const;
```

Add Order


Add Order
Is Market or Limit?
Yes
Match the order
Qty>0 ?
Yes
Add to order book
No
Execute Stop Orders
No
Add Stop Order

Execute stop orders

```
template<typename Pred, typename Comp>
void execute_stop_orders(unsigned market_price, std::set<unsigned,Comp>&
prices,std::unordered_map<unsigned, std::list<Order>>& order_pool, Pred
p){
for (auto f = prices.begin(); f != prices.end();) {
        if(!p(*f, market_price))
            break;
     // activate all stop orders at price level *f
}
}
```

Execute stop orders

```
 //if stop price at a level <= market price, stop order is activated
    auto buy_pred = [](unsigned stop_price, unsigned sell_market_price)
{
     return stop_price <= sell_market_price;
    };
//execute stop buy orders if possible
execute_stop_orders(get_sell_market_price(), stop_buy_prices,
stop_buy_pool, buy_pred);
```

Execute stop orders

```
//if stop price at a level >= market price, stop order is activated
    auto sell_pred = [](unsigned stop_price, unsigned buy_market_price)
{
     return stop_price >= buy_market_price;
    };
//execute stop sell orders if possible
execute_stop_orders(get_buy_market_price(), stop_sell_prices,
stop_sell_pool, sell_pred);
```

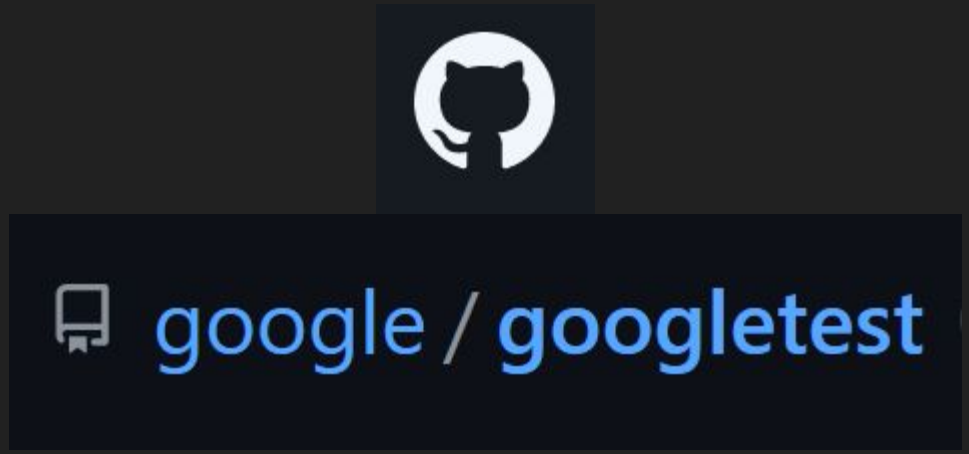
Order Matching

Sellers: ask prices

Buyers: bid prices

Part 3: Running & Results

Build and Test



```
Welcome to the Nasdaq ITCH order matching engine
----------------------start-------------------------
Opened ../ExchangeDataViewer/data/03272019.PSX_ITCH50 to read ITCH 5.0. messages.
Opened ./data/test.log for writing.
Begin building book and matching orders
Processed 10Million messages. 2.5 Mio messages per sec.
Processed 20Million messages. 2.85 Mio messages per sec.
Processed 30Million messages. 2.72 Mio messages per sec.
Processed 40Million messages. 2.66 Mio messages per sec.
Processed 50Million messages. 2.63 Mio messages per sec.
Processed 60Million messages. 2.6 Mio messages per sec.
----------------------end-------------------------
Finish building book and matching orders in 23seconds.
Total Add Order is 557615 and Total Delete Order is 523969
File ./data/test.log has been closed.
File ../ExchangeDataViewer/data/03272019.PSX_ITCH50 has been closed
Finished, processed 61784531 messages in 23seconds.

Process finished with exit code 0
```

Performance

C++ version: C++20

Compiler version: g++ (Ubuntu 12.1.0-2ubuntu1~22.04) 12.1.0

Time taken: 22 seconds

Total messages in data source: 61784531

# of orders added/deleted: 557615 and 523969

# of matched orders:

AAPL	AMZN	MSFT	TSLA
2091	4019	22707	510

Sample Matched Results

```
509798;556269;17859600;100
509954;557251;17858600;8
509953;557251;17857000;10
563382;634347;17850100;1
563382;634351;17850100;1
563453;634351;17850100;1
563453;634386;17850100;3
562292;634386;17850000;10
649010;649140;17844100;10
649010;649173;17844100;10
649010;652034;17844100;8
649010;655593;17844100;10
649010;658697;17844100;10
649010;658735;17844100;1
649010;661985;17844100;7
649010;667309;17844100;6
```

Future work

- Concurrency
  - Parser
  - Orderbook
- Support other exchange data
- Support other order types
- Support more operations like modify order.

Concurrency

```
#include <thread>
std::jthread jt(&OrderBook::add_order, &book,
std::ref(thisOrder));
jt.detach();
```

```
#include <future>
auto a = std::async(&OrderBook::delete_order, &book, id);
auto s = a.get();
```

Concurrency

```
#include <semaphore>
std::binary_semaphore booksem{1};
booksem.acquire();
booksem.release();
```

Contributions

Fenglei - Design and implementation of Order Matching, design of Order and OrderBook data structures, unit testing

Swetha - Order Book functions, Google test integration, unit tests for OrderBook, design of OrderBook data structure

Zhejian - ITCH parser, integration of parser and order matcher, test for integral performance, design of OrderBook data structure

References

- enewhuis/liquibook: Modern C++ order matching engine (github.com)
- chronoxor/CppTrader: High performance components for building Trading Platform such as ultra fast matching engine, order book processor (github.com)
- martinobdl/ITCHc: NASDAQ ITCH 50 Book Constructor (github.com)
- (63) How traders orders get matched (exchange matching algorithms) - YouTube
- CME Globex Matching Algorithm Steps - Electronic Platform Information Console - Confluence (cmegroup.com)

Thank you!