Description of Data Structure(s)

The primary data structure used is the `exchange` unordered_map, mapping instrument names to their respective OrderBook instances. Each OrderBook contains two linked lists for buy and sell orders. The linked list structure in OrderBook is implemented using the Order class, where each node represents an order. `idOrderMapper` is an unordered_map that associates order IDs with a reference to the order. Mutexes within each node are employed for synchronisation, ensuring thread safety when modifying shared data structures.

Explanation of How Data Structures Enable Concurrency

Instrument Level Concurrency:
This level of concurrency is facilitated by maintaining separate OrderBook instances for each instrument in the `exchange` unordered_map. Each instrument operates independently, allowing for concurrent processing of orders related to different instruments.

Phase Level Concurrency:
Use of separate linked lists for buy and sell orders within each OrderBook. Within each list, the orders are sorted according to their priority, such as the highest price at the head for “Buy List” and lowest price at the head for “Sell List”. Additionally, every Order in the OrderBook has its respective mutex. This allows for fine-grain parallelism where multiple orders for the same sides can be processed simultaneously for a given instrument. Further explanation on this will be described in the next section.

Mutexes, such as `exchange_mutex`, are strategically placed to guard critical sections where shared data structures are accessed or modified. This prevents data races and ensures that operations on different instruments are thread-safe.

Supporting Concurrent Execution of Orders

Execution of Orders
The explanation will be based on buy orders but will be the same for sell orders the other way around. The code execute_buy_orders (execute_sell_orders) within the OrderBook class handles the execution of buy and sell orders in the matching engine.

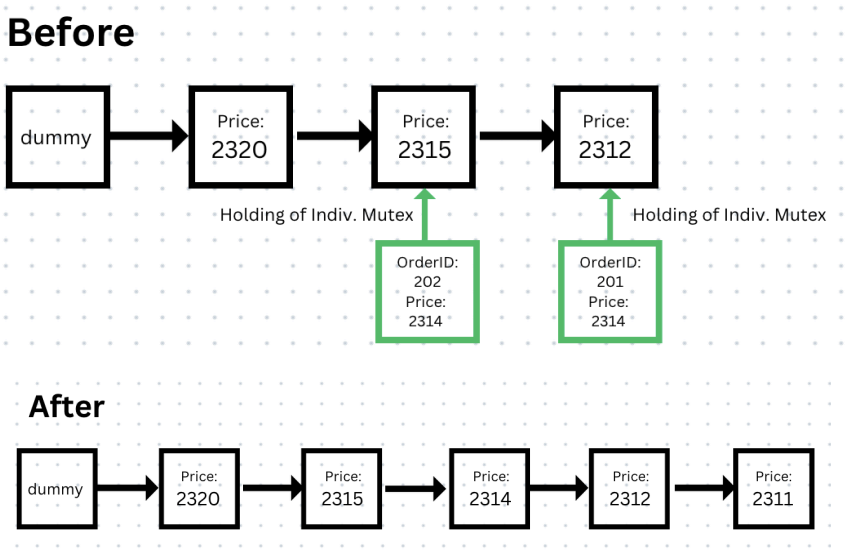
Bridge Mechanism:
The code begins with a while loop that locks the order book (std::lock_guard obLock(this->mut)) to ensure exclusive access to the orderbook and unlocks when it goes out of scope. It then checks a shared variable bridge using atomic operations. The bridge is a synchronisation mechanism, controlling if buy (sell) orders are allowed to be executed. Bridge value of more (less) than or equal to 0 will allow buy (sell) orders to execute. It uses atomic operations fetch_add (fetch_sub) of 1 upon entry and fetch_sub (fetch_add) of 1 upon exit. This method allows multiple orders of the same type to execute matching and place into the orderbook. Hence, achieving phase concurrency.

Order Execution Loop:
The head of the “Sell List” is acquired using std::unique_lock to ensure that there is only 1 matching done at a time and flexibility of unlocking.

Order Book Manipulation:
The push_buy_order method is called to add a new order to the buy side of the order book if there's remaining quantity after executing against sell orders. This is the part where there can be multiple buy orders being pushed to the orderbook concurrently

Adding of Orders
In the event where there are no opposite orders for execution, an order will have to be added back to their respective side linked list. Given our implementation of Order having their own mutexes, it allows for multiple orders of the same type to be concurrently added back to the list.

Scenario For Adding of Orders
Let's assume there are two orders being 'B 201 AMZN 2311 10' and 'B 202 AMZN 2314 10'. The current head of the "Buy List" has a price of 2320 and there are multiple orders with prices ranging from 2320 to 2310. Order 201 will first obtain the mutex for the head of the mutex and check if it can be added as the next node. If it is unable, it will release the mutex and proceed to the next node. At this instance, Order 202 will obtain the head of the mutex and check accordingly. This recursive checking and releasing of mutexes will continue for both orders until their respective positions are located and they are added to the OrderBook.



Description of Testing Methodology
The testing primarily concentrated on ensuring the semantic accuracy of the output. Initially, we validated the fundamental behaviour of the engine using the given test cases. Subsequently, our emphasis shifted towards confirming correctness within a multi-threaded, high-load setting. We generated several test scenarios involving 5 threads, each executing a thousand orders. The execution and verification of these tests were conducted using the supplied grader.