Description of Data Structures used

The exchange (aka Bridge for the Single-lane bridge problem) consists of:

- 2 **linked lists**, namely the buy and sell linked list to manage all the resting buy and sell orders.
- Another **linked list** to store ReturnedData because with our implementation, there might be an issue of interleaved output printing. For example, a later thread, B, finds an order to cancel first as compared to an earlier thread A. If we do not serialize the output printing, that later thread, thread B, will be printed first before the earlier thread, thread A.
- A **queue** to store the new orders to be added after there are no more orders from the same direction, similar to *lazy addition*. We will only add it to the opposing resting list when it is ready to prevent data race.

Synchronization Primitives used

The synchronization primitives used are **conditional variables**, **mutexes** and **hand-over-hand-locking**.

Explanation of level of concurrency

The level of concurrency is **phase-level concurrency**, specifically solving the single-lane bridge problem. We group multiple orders of the same type and allow them to execute concurrently. Additionally, within these orders there are multiple instruments which allow orders of different instruments to execute concurrently.

If the number of threads processing is 0, we will take any random order and set the bridge direction to be that and increment threads processing by 1. If it is not 0, we allow threads with the same input type to continue processing. This is done via **condition variables**. For the threads of other types, they are put to sleep and forced to wait until there are no more orders of the same type notified via `notify_one` or by spurious wakes.

We perform **hand-over-hand locking** on the linked list of the resting orders. Within each node, there is a mutex within. When a thread attempts to access the node, it will lock the node and mutate the data. If a later thread finishes processing before the earlier thread and attempts to mutate the same node, it will not be blocked because of the mutex. This means that multiple threads can progress at once because while an earlier thread is processing a node, later threads can process the nodes behind it concurrently.

We perform lazy deletion and addition and serialized printing. For lazy deletion, we set the `count` to 0 when it is fully executed or `is_cancelled` flag to true. However, we delay the actual deletion as there might be other threads reading it and leading to a data race. We

perform lazy addition because if we do not synchronize the addition, there might be a *producer-producer* data race. We perform serialized printing because there might be interleaved printing. For example, if a thread that starts processing later finds a node to be canceled earlier than a thread that processes earlier, it will print the timestamp of the later thread before the timestamp of the earlier thread.

Finally, we used **mutex** to synchronize the number of threads processing by counting the number of active processing threads. When we are guaranteed that the number of threads processing is 0, we know that it is safe to perform the addition, sorting, deletion and serialized printing.

Explanation of testing methodology

The basic test cases provided focus more on the correctness of our matching strategy. This ensures the correctness of our code in a single thread, taking into account the different possible scenarios. If the basic test cases fail, there is something wrong with our code and we will not be able to proceed with the intermediate cases.

Next are the written intermediate test cases that help us to check whether we have achieved phase-level concurrency. These tests include multiple threads working on a small number of orders. We also execute these tests multiple times to better ensure that there are no apparent data races and also with the `-fsanitize` flag.

Finally, we wrote a python script that generates an input file with random orders with the given number of threads, orders and instruments. We will increment the values slowly and run the input file up to 20 times to ensure that the output passes. A huge number of orders, threads and instruments are given to stress test our implementation and see if it can handle the large scale of orders and threads received.