# Algorithmic Trading Developer Technical Assignment –part 2 Report

Applicant: Vishlen V. Pillay

## Introduction

The assignment's goal is to write a limit order book (LOB), with the following components.

1. OrderBook (Submission 1) – due 16 August 2024 (before close of business)
2. Matching Engine (Submission 2) – due 22 August 2024 (before close of business)

## Efficiency Mechanisms:

### Iterators over *for*loops for map data structure

Both *for*loops and Iterators provide mechanisms to traverse through collections of elements. Although they both serve the same purpose, they differ in their applicability.

*for*loops aren't directly compatible with removing elements from the collection being traversed. Modifying the collection during a *for*loop iteration can lead to unpredictable behavior as the size of the collection is modified. This often results in ConcurrentModificationException or incorrect indices.

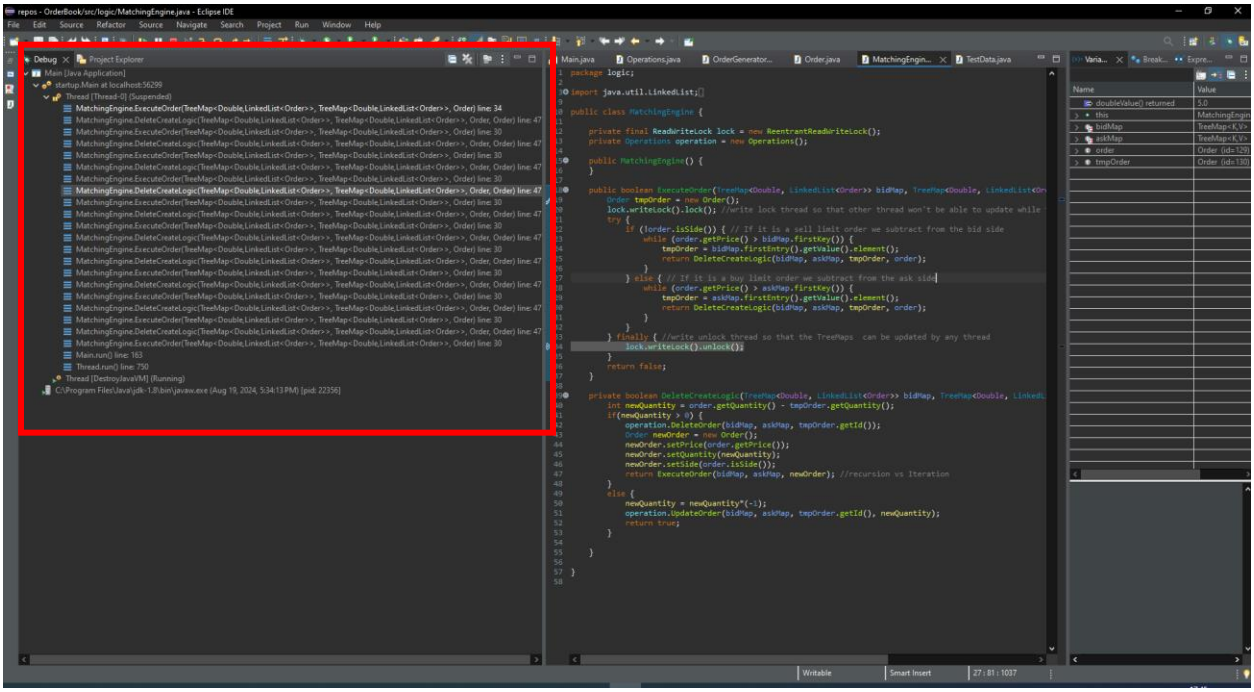
Iterators, on the other hand, provide a safe and reliable way to remove elements during iteration using the remove() method. Iterator internally maintains a cursor or a position within the collection. When we call remove(), it knows exactly which element to remove based on its internal state. This prevents concurrent modification issues and ensures the integrity of the iteration process.

*for*loops are more prone to errors due to their reliance on index-based access. Incorrect index values or modifications to the collection during iteration can lead to various exceptions and unexpected behavior. For example, for loop can lead to IndexOutOfBoundException if the index value is outside the bounds of the collection. This can happen if the index variable isn't properly initialized or if the collection size is modified during iteration.

On the other hand, Iterator enforces hasNext() checks before accessing elements, preventing null pointer exceptions. This ensures that the Iterator points to a valid element before attempting to access it.

## Iterations over recursion for large data sets and improved performance in memory and space

Initially while coding this matching functionality recursion seemed to be the simplest solution to use. While debugging a test I noticed that with recursion each iteration created a new instance of the class, if we have a larger data set this could lead to a high memory intensive application. This can be seen by the screenshot below. Highlighted in the red rectangle are the many instances of the class created for this recursion run.



Recursion is slower due to the overhead of multiple function calls. Iterations involve just 1 function call, and the function will iterate within the for/ while loop within the same function, thus reducing overhead.

## Solution Approach:

It is important to note that when matching/ executing a sell order we are subtracting from the 'bid' side, and when matching/ executing a buy order we subtract from the 'ask' side. The Limit Order Book needs to start subtracting the quantity from the highest priority order first before moving down the list to the next priority order and only if the next priority order is within the limit order price. The orders are executed at or better than the given limit price. Executing a buy order will start from the lowest price (better than the limit price). Executing a sell order will start from the highest price(better than the limit price).

Using this information, I had to reverse order my bidMap, so that it would start executing from the highest price going to the lowest prices for execution of sell orders.

1. My algorithm is to take in a limit order to be executed (input price, quantity, buy/sell). If this is a Sell order, we will use the bidMap, if it is a Buy order we will use the askMap. Because I am using TreeMap datastructure the maps are already ordered ascending using the price as the key. The bid Map is ordered descending by Key.
2. When Iterating through the askMap, the first check is to see if the limit order price is greater than or equal to the first key. If not, then the order cannot be executed as the limit price is too low.
   When Iterating through the bidMap, the first check is to see if the limit order price is less than or equal to the first key. If not, then the order cannot be executed as the limit price is too high.
3. Next, we will iterate through the LinkedList of orders for the first price using a listIterator. This LinkedList is already in priority order (FIFO).
4. Taking the first element of the Linked List, I subtract the limit order quantity from the 1st element quantity.
   a. If the result == zero, remove this element and jump out of the iteration.
      i. This means the limit order quantity was filled by just the 1st element order.
   b. Else If the result > zero, remove element and modify the limit order with the new quantity, go to the next element and repeat checks.
      i. This means the 1st element order was not enough to complete the limit order, we need to go to the next priority order.
   c. Else if the result < zero, multiply quantity by -1 (to make it positive), update the element to this new quantity and break out of the loop.
5. We continue to loop through each order starting from highest priority and going down, and subtracting the quantity until the limit order quantity is zero, that is how we match the limit order to the orders from the order book.
6. At the end of each Key iteration, we do a check to see if the value of the Key-Value pair is empty and remove the key from the map if it is. This is how we keep the first element being the next key, if the current key's orders are all removed, thus taking us back to #3 where we check the limit order price against the first element of the map.
7. The whole while loop repeats.

## Data Structures:

The choice of data structures used in part 1 of this assignment was perfectly chosen for the functionality to match the limit orders.

The TreeMap datastructure was chosen because of its Key-Value pair architecture and especially because each element is added in natural order of the key. This ordering played a big part in traversing through each key ascendingly for the askMap. The TreeMap data structure also provided me with the ability to reverse the order of the map so that the bidMap can be accurately represented as going from highest price to lowest price.

The LinkedList was used because of the ability to easily modify and remove elements from anywhere in the list. This became beneficial when deleting and modifying an order. It also enabled us to keep all orders ordered in priority order (FIFO).

Iterators are used to iterate through elements of the TreeMap and a listIterator is used to iterate through elements of the LinkedList.

As mentioned previously iterators provide a safe and reliable way to modify or remove elements from a map or list, without the fear of running into ConcurrentModificationException of modify or removing an element while iterating through the collection.

A ListIterator was used to traverse the LinkedList to be able delete, add or modify elements during an iteration of the collection without a ConcurrentModificationException being thrown. The ordinary Iterator does not allow deletion or modification of elements and throws ConcurrentModificationException when attempting to add an element during an iteration of the collection.