# Application Overview

## Unit tests:

119 unit tests passing. 0 failing.

I have broken down the unit tests into individual classes. The orderBookManagerImpl has separate classes for each method. Within each class, I have used consistent patterns and naming schemas to test edge cases for each method.

Tests are generally repeated using Side as a variable. As Side is used to map items into HashMaps, it is a heavily dependent variable within the application for correct functioning. It is therefore tested at each method.

# Design:

## Classes Implemented

* OrderBookManagerImpl
* OrderBook
* OrderLinkedList, OrderNode
* InstrumentProperty, LevelProperty

## OrderBook

The OrderBook is used for adding, removing, and modifying orders. The OrderBook extends from TreeMap<Long, OrderLinkedList> to ensure the order books are always kept in a sorted order based upon price (Long).

The bid book differs from the ask book in one way: when the one-argument constructor is called, it will call the super(Comparator) constructor of treemap. The one-arg constructor is only used when creating a bid book, and the Comparator.reverseOrder() Comparator is passed as an argument. The constructor will ensure the bid book orders the levels with the highest price first. See createNewBookForOrdermethod in the OrderBookManagerImpl class.

## OrderBookManagerImpl

The OrderBookManagerImpl contains 3 hashmaps:

1. orderHashMap to store the orderId and Order as a lookup table
2. instrumentPropertyHashMap to store the instrumentProperty instances for different types of orders (instrument, side)
3. orderBookHashMap to store the different OrderBooks that are identified by a string that is concatenated with the instrument and side to form a key.

The three hashmaps provide constant time access, adding, and removing of entries. The possibility to create unique keys with the instrument and side of the order, as well as the orderId means they are optimal data structures to use for storing lookup values.

When I had noticed the property requests (getOrderAtLevel, bestPrice, orderNumAtLevel, totalQuantityAtLevel, totalVolumeAtLevel) all used the orderId as an argument, I used the orderHashMap to retrieve access to the order itself to further find properties such as the side, instrument, and price. The orderHashMap acts as an intermediary between the orderId and identifying the instance of the InstrumentProperty or OrderBook to operate upon. The instrumentPropertyHashMap and orderBookHashMap use a key that is a concatenation of the instrument and side of the order to return the InstrumentProperty instance or OrderBook instance respectively.

## OrderLinkedList, OrderNode

The OrderBook class stores the value of the treeMap as an OrderLinkedList instance. Each instance relates to a price, instrument, and side combination. The OrderLinkedList class is my implementation of a LinkedList to store the Orders. The OrderLinkedList contains OrderNode instances, which are instances of a class that store the Order, next Order, and previous Order in the doubly-linked LinkedList.

By using a LinkedList adaptation, I am able to achieve constant time for appending to the LinkedList. I am also able to delete orders within the chain without having to update all the other orders. If I had used an ArrayList, removing an order within the body of the list will result in a recursive function to update the index position of all the other orders at the level. The Big Theta for updating the OrderLinkedList is less than that of the ArrayList.

## InstrumentProperty, LevelProperty

The InstrumentProperty class is used to return values for the get requests for each instrument at the instrument level. The InstrumentProperty class can return the levels for the instrument to provide a further level of granularity. The LevelProperty class stores information relating to the level of each instrument and side. LevelProperty is used to group the data from same price levels for each instrument and side.

## Time and Space Complexity

The operations of the application perform at constant time complexity o(1) with the exception of the modifyOrder, deleteOrder, and getOrdersAtLevel methods. The three methods have a complexity that is equal to the number of Orders at the level it is searching on:

*O(n) where n is number of orders at a level for an instrument and side.*

The non-constant time complexity is created due to the LinkedList style of storing and ordering Orders at each level. The application must loop through the orders until it finds the correct one.