How to Design a Limit Order Book

date: 20260521
status: #finished
tags: #computer_science #data_structure #trading #limit_order_book #order_book #market_microstructure #market_maker #liquidity

---

0. Introduction

This note aims to answer, but is not limited to, the following questions.

1. What operations interact with a limit order book?
2. What data structures are involved to build a limit order book?
3. What are market makers? and what are the design principles of limit order book?

---

1. Operations of the Limit Order Book

A limit order book stores the resting limit orders that provide liquidity to a market. Its data structures should be designed around three core operations: (i) inserting a new limit order, (ii) canceling a resting limit order, and (iii) matching an incoming market order with resting limit orders.

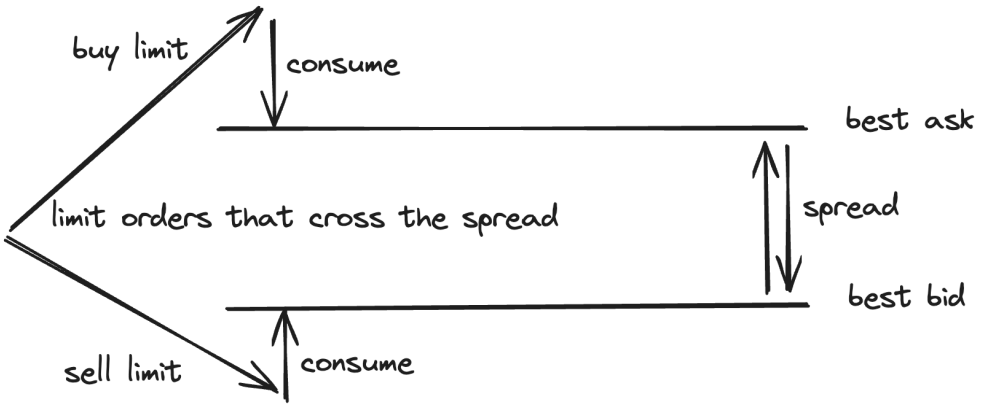
> **Note**
>
> Because each incoming order is matched the moment it arrives, this book implements continuous trading; an exchange can instead batch orders over a window and match them all at one clearing price in a call auction.

(i) A new limit order can cross the spread.

When a limit order arrives, the engine first checks whether it crosses the spread and can be matched immediately.

- A limit buy order crosses if its price is higher than or equal to the best ask.

- A limit sell order crosses if its price is lower than or equal to the best bid.



If a limit order crosses, it is treated as a market order and matched against the resting limit orders, which widens the spread. The matching process continues until the spread is no longer crossed.

- For a limit buy, matching stops when the resting ask price is higher than the limit price.
- For a limit sell, matching stops when the resting bid price is lower than the limit price.

Any remaining quantity that could not be matched is then rested in the book.

(ii) The price can move only when market orders hit the book.

A tick defines the increments across price levels in the limit order book. For example, a limit order book with the tick as 0.01 might display as the following.

```
Ask Price     Quantity
100.00         3
99.99          4
99.98          5
```

The best ask is 99.98. With the above limit order book, a buy market order with the quantity 10 will push the best ask upward to 100.00.

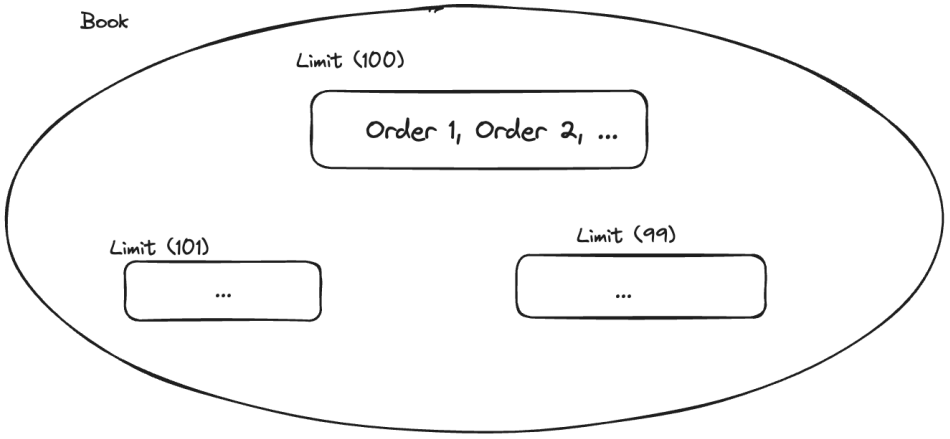
Besides, we can see a single price number displayed on the screen, which is usually the last traded price. After the above trade, the price will be pushed upward to 100.00.

---

2. The Three Levels Data Structures: `Order`, `Limit`, and `Book`

(i) What are they?

Fundamentally, there will be an `Order` object storing the information that includes but not limited to (i) type (limit v.s. market), (ii) buy or sell, (iii) quantity and (iv) price. Besides, we will need a `Limit` object associated with a price level, functioning as a FIFO queue to store all `Order` objects with the limit type at that price level. Finally, a `Book` object stores and organizes all buy and sell `Limit` objects.



Moreover, as there are buy and sell limit orders, there are two groups of `Limit` objects. One stores buy orders while the other stores sell orders. A single `Book` object stores these two groups.

(ii) Why three levels?

One might ask, why not adopt the two levels structure, in which a `Book` object organizes all `Order` objects? The reason is that the matching priority of limit order has two steps, which is known as the price-time priority.

> **Definition (Price-Time Priority)**
>
> Price alone is not enough to decide matching priority. If several limit orders have the same price, the earlier order should be matched first. This is called **price-time priority**.

That's exactly why we have the FIFO queue `Limit` to organize orders at the same price.

3. Market Makers

(i) Motivation of Market Making

A trader who sends a market order accepts whatever price the book offers, so a thin book costs them slippage — a large order walks through several price levels and its average fill drifts away from the expected quote. Keeping that drift small requires a book that stays deep and tightly quoted, which requires someone to keep quoting it.

> **Definition (Market Maker)**
>
> A market maker continuously posts limit orders on both sides of the book, a buy limit at the best bid and a sell limit at the best ask, so that a trader arriving with a market order always finds a counterparty resting there. That is what **providing liquidity** means: standing ready to take the other side.

(ii) How Market Makers Earn

The market maker is paid for standing there by the spread. Quoting a bid at 99.95 and an ask at 100.05, it buys 100 shares from a seller who hits the bid and sells them to a buyer who lifts the ask, keeping 0.10 per share without ever taking a view on direction. The two quotes rest in parallel, but the fills arrive one at a time, so the market maker holds inventory in between, and that's the risk of market making.

Besides, market makers continuously reprice — canceling resting limit orders and inserting new ones as their estimate of fair value moves — because a quote left behind at the old price will be taken at a loss.

(iii) The Design Principle to Facilitate Market Making

To encourage market making, which boosts the trading experience, the cancellation and insertion of limit orders must not search the book across a wide range of price levels. This is exactly what the two hash tables below, the price-level index and the order index, make $O(1)$.

4. Make Insertion at Existing Price Levels Direct

(i) The `limits_by_price` Index

To avoid that search, maintain a hash table `limits_by_price`:

```
limits_by_price:
    price → FIFO queue of resting orders at that limit
(price)
```

Insertion then becomes:

1. Look up `price` in `limits_by_price`.
2. If found, append the order directly to the back of that level's queue — $O(1)$.
3. If not found, create a new price level, insert it into the price-level index — $O(\log M)$ if a BST tree structure is used to organize the price index, and register it in `limits_by_price`.

(ii) The Tree Cost Is Paid Once per Price Level

The $O(\log M)$ BST insertion only occurs once per price level, when it first appears. Every subsequent order at that price is $O(1)$.

---

5. Make Cancellation Direct

(i) The `orders_by_id` Index

When a limit `Order` object is created, it is assigned an `order_id`, which is what a cancellation targets.

To avoid the scan across `Limit` objects and the `Order` objects inside them, maintain a hash table `orders_by_id`:

```
orders_by_id:
    order_id → order node
```

Cancellation then becomes:

1. Use `order_id` to find the order node.
2. Remove that node from the underlying `Limit`.
3. If the resulting `Limit` is empty, remove the `Limit` from the `Book` and delete the corresponding price key in `limits_by_price`.

(ii) Why the FIFO Queue Should Be Doubly Linked

A singly linked list is enough for ordinary FIFO behavior. It can append new orders to the tail and remove matched orders from the head efficiently:

```
head → A → B → C → D
```

However, cancellation may remove an order in the middle of a queue. If `orders_by_id` points directly to node C, a singly linked node can know its next node D, but it does not know its previous node B. Removing C still requires updating:

```
B.next = D
```

Without a previous pointer, the queue must search from the head to find B.

A doubly linked list avoids that search:

```
A ⟷ B ⟷ C ⟷ D
```

Because node C knows both its previous and next nodes, the engine can unlink it directly after the hash-table lookup.

---

6. What Data Structure Should `Book` Use?

The limit order book should efficiently expose the limit orders at the best bid and ask to enable fast market order matching.

Two reasonable implementations for `Book` are (i) a heap-based priority queue or (ii) a balanced search tree.

(i) Heap-Based Price Index

A heap is effective when the main job is fetching the best price.

- Use a min heap for sell limits.
- Use a max heap for buy limits.

However, deleting an arbitrary price level from a heap is awkward. This matters after cancellation: if every order at one price level disappears, the book should

remove that empty level.

One heap-based solution is **lazy deletion**. The engine leaves empty price levels in the heap and discards them later when matching reaches them. This can work, but it moves cleanup into the matching path. If several empty levels are near the top of the heap, the engine must skip them before it reaches active liquidity.

(ii) Balanced-Tree Price Index

A balanced search tree keeps price levels ordered while supporting insertion and deletion of price levels directly. When cancellation empties a price level, the level can be removed from the index immediately.

(iii) Fast Retrieval of the Best Bid/Ask Price

A balanced BST like a red-black tree maintains a cached pointer to the leftmost (ask)/rightmost (bid) node as part of its internal bookkeeping, giving $O(1)$ access to the best price without any tree traversal.

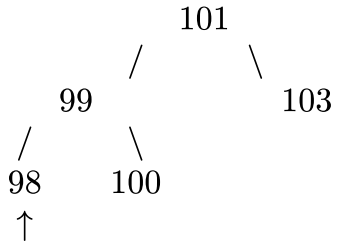
The best ask is the smallest price in the ask BST, and the best bid is the largest price in the bid BST. These are always the leftmost and rightmost nodes respectively, which by definition sit at the extreme end of the tree.

(iv) Update of the Cached Pointer Is Efficient

When the best price level is exhausted and erased, the cached pointer must advance to the next best price, and the second-best bid/ask will always be the parent or the child of the current "exhausted" best.

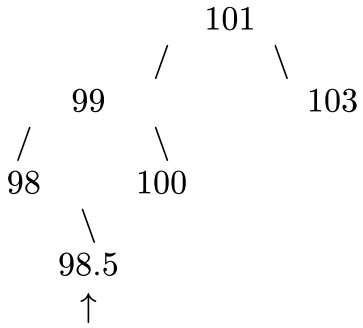
Here are the four cases:

Example (Ask, Pure Leaf (Successor Is Parent))



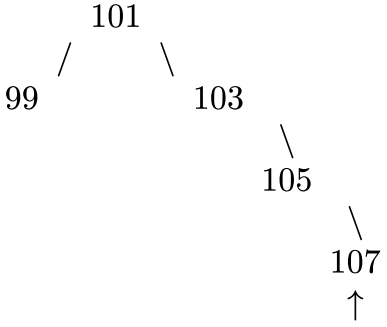
best ask erased, no children → successor is parent 99

Example (Ask, Has Right Child (Successor Is Right Child))



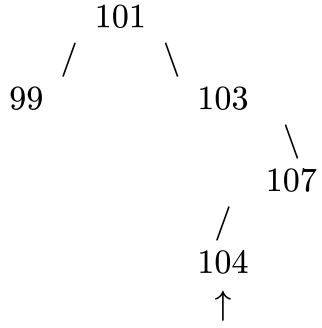
best ask 98 erased, has right child → successor is right child 98.5

Example (Bid, Pure Leaf (Successor Is Parent))



best bid erased, no children → successor is parent 105

Example (Bid, Has Left Child (Successor Is Left Child))



best bid 107 erased, has left child → successor is left child 104

In every case the update is $O(1)$ — either one step up to the parent or one step down to the child. No tree traversal needed.