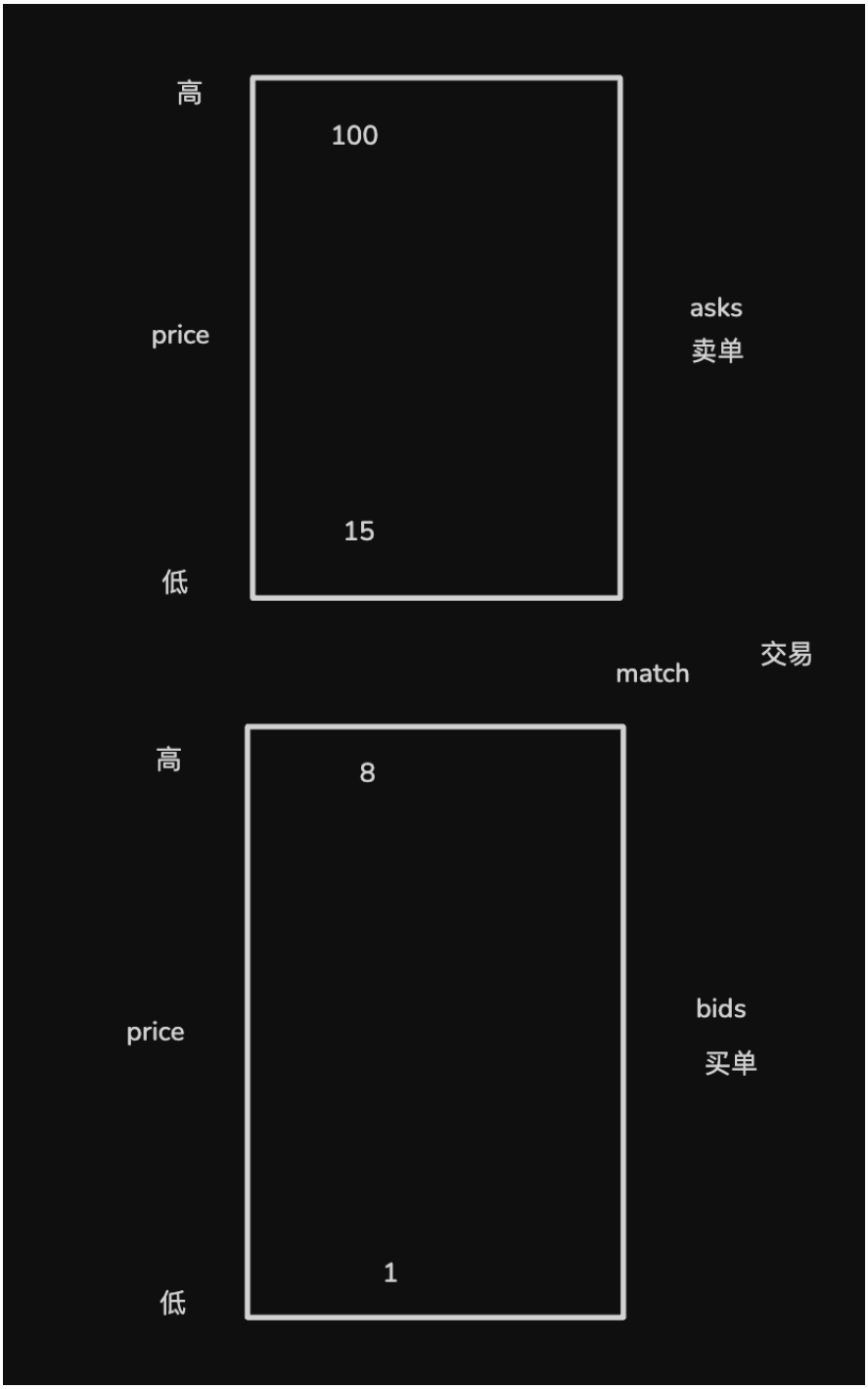
杨淇

数据结构

买得高 和 卖得低 的匹配



可以确定用有序数据结构存储订单 key 为 price，我们还需要确定value的数据类型

首先不可以用Order，因为 无法处理 价格相同 时间不同 或者 价格相同 人不同的情况

这里用list 双向链表，为了方便：

1. 中间删除 时间复杂度 O(1) （vector是O(n)）
2. 用时间戳进行 先进先出 ，都是O(1)的（vector头部插入是O(n)）

```
class OrderBook {
private:

    // 卖单：价格低到高
    std::map<double, std::list<Order> > asks;

    // 买单：价格高到低
    std::map<double, std::list<Order>, std::greater<double> > bids;
}
```

展示深度时，需要用key找到价格，将list里面的剩余标的数量求和

```
count = 0;
for (auto it = bids.begin(); it != bids.end() && count < 5; ++it, ++count) {
    uint32_t level_qty = 0;
    for (auto lit = it->second.begin(); lit != it->second.end(); ++lit) level_qty += lit->quantity;
    std::cout << std::setw(10) << it->first << " | " << level_qty << std::endl;
}
```

撮合

功能：

void match(新加入的订单，对手订单map)

1. 能成交的话，成交对手已有订单
2. 不能成交时返回

流程：

1. 新加入订单时 判断时候能成交（买单的话 价格要 高于 最小的已有的卖单）
2. 能成交的话 可能消耗多笔交易（按照价格 时间的顺序消耗）
3. 最终可能新加入的订单完全成交，也可能还有剩余（用quantity记录）

在添加订单到订单簿前调用

时间复杂度为O(n) 因为可能会消耗多笔对手盘的订单

```
    // 撮合函数
    //  这里用泛型传入 bids 和 asks T 相当于map<double, list<Order>>
    // incoming 新进入的订单 counter_side_map对手订单(map<double, list<Order>>)
    template<typename T>
    void match(Order& incoming, T& counter_side_map) {
        // 越靠前的越先交易

        // 先看价格
        // 以订单为买单 对手订单为卖单（从小到大） 为例
        // 先看卖得最便宜的

        // 这里需要遍历 来实现大买单吃掉多个小单的情况
        auto it = counter_side_map.begin();
        while (it != counter_side_map.end() && incoming.quantity > 0) {
            double best_price = it->first;

            bool can_match = (incoming.side == BUY) ? (incoming.price >= best_price) : (incoming.price <= best_price);
            // 买的话 价格要>= 卖得最便宜的
            // 卖的话 价格要<= 买得最贵的

            // 没匹配上
            if (!can_match) break;


            std::list<Order>& queue = it->second;

            // 同理需要遍历，这里的遍历是按照时间顺序的
            auto q_it = queue.begin();
            while (q_it != queue.end() && incoming.quantity > 0) {
                Order& maker = *q_it; // 成交的对手订单

                uint32_t fill_qty = std::min(incoming.quantity, maker.quantity); // 成交量

                std::cout << "MATCH: Order " << incoming.id << " (" << (incoming.side == BUY ? "BUY" : "SELL")
                          << ") matched with Order " << maker.id << " (" << (maker.side == BUY ? "BUY" : "SELL")
                          << ") | Qty: " << fill_qty << " @ Price: " << maker.price << std::endl;

                incoming.quantity -= fill_qty;
                incoming.filledQuantity += fill_qty;
                maker.quantity -= fill_qty;
                maker.filledQuantity += fill_qty;

                if (maker.quantity == 0) {
                    order_lookup.erase(maker.id);
                    q_it = queue.erase(q_it);
                } else {
```

```
                    ++q_it;
                }
            }

            if (queue.empty()) {
                counter_side_map.erase(it++);
            } else {
                ++it;
            }
        }
    }
```

Add和Delete

ADD

添加限价单（挂单）（这里指实现了限价单，这种最常用... 因为市价直接吃单手续费高，不太适合高频）

市价单的话 不用price 直接买，一路math，直到自己单用完

流程：

1. 构造Order
2. 撮合一下 看看能不能直接交易了
3. 订单还在（交易部分 或者 没交易）放到 map<double, list>里面

时间复杂度为 m + logn，n为已有 自己类型的订单的价格数量（map导致），m为match时候 消耗的对手订单数

```
void limitOrder(uint64_t id, double price, uint32_t quantity, Side side,
std::string symbol) {
      if (order_lookup.count(id)) {
          throw std::invalid_argument("Order ID already exists.");
      }
      if (quantity == 0) return;

      Order new_order(id, price, quantity, side, symbol);

      // 看看能不能直接买到 或者 直接卖出
      if (side == BUY) {
          match(new_order, asks);
      } else {
          match(new_order, bids);
      }

      // 假如订单还在 则需要加入订单簿
      if (new_order.quantity > 0) {
          if (side == BUY) {
              std::list<Order>& queue = bids[price];
              queue.push_back(new_order);
            // 这个是删除时用的索引
```

```
            order_lookup[id] = (OrderEntry){price, side, --queue.end()};
        } else {
            std::list<Order>& queue = asks[price];
            queue.push_back(new_order);
            order_lookup[id] = (OrderEntry){price, side, --queue.end()};
        }
    }
}
```

DELETE

我们要查到 map<double, list> 里面的某个order

如何我们提前存储 [价格, 订单交易方向, list里面的位置] 我们直接用O(1)的时间查到，要删除的订单，并且list删除是O(1)的 整个的复杂度就为O（1）

我们使用 key为 orderId，value 为 [价格, 订单交易方向, list里面的位置] 的unordered_map来进行处理

（不用map，map是红黑树key有序，unordered_map是一个hash表，key无序。这里不需要id有序，用unordered_map更快查询插入是 O(1)，map查询删除其实是O(logn)）

这个unordered_map存储了现在在orderbook中有效的全部订单

我们在ADD的时候提前存好索引 `order_lookup[id] = (OrderEntry){price, side, --queue.end()}` 这样就可以维护好。match函数中在订单交易完成后删除。

时间复杂度为logn，n为自己类型的订单的价格数量（用price查map时候导致）

```
void cancelOrder(uint64_t id) {
    auto it = order_lookup.find(id);
    if (it == order_lookup.end()) {
        std::cout << "Warning: Order ID " << id << " not found." << std::endl;
        return;
    }

    const OrderEntry& entry = it->second;
    if (entry.side == BUY) {
        bids[entry.price].erase(entry.it);
        if (bids[entry.price].empty()) bids.erase(entry.price);
    } else {
        asks[entry.price].erase(entry.it);
        if (asks[entry.price].empty()) asks.erase(entry.price);
    }

    order_lookup.erase(it);
    std::cout << "Order " << id << " cancelled." << std::endl;
}
```

cases

打印

```
197
198         for (int i = 0; i < 7; ++i) {
199             ob.limitOrder(100 + i, 8.0 - i * 0.1, 10, BUY, "TEST");
200             ob.limitOrder(200 + i, 15.0 + i * 0.1, 10, SELL, "TEST");
201         }
202         ob.displayDepth();
203
204
```

Problems Output Debug Console Terminal Ports zsh

```
========== 深度 ==========
    卖单
     Price | Quantity
      15.4 | 10
      15.3 | 10
      15.2 | 10
      15.1 | 10
        15 | 10
--------------------------------
         8 | 10
       7.9 | 10
       7.8 | 10
       7.7 | 10
       7.6 | 10
    买单
================================
```

删除

```
196     try {
197         // 删除订单
198         ob.limitOrder(5, 12, 10, SELL, "TEST");
199         ob.displayDepth();
200
201         ob.cancelOrder(5);
202         ob.displayDepth();
203
204
```

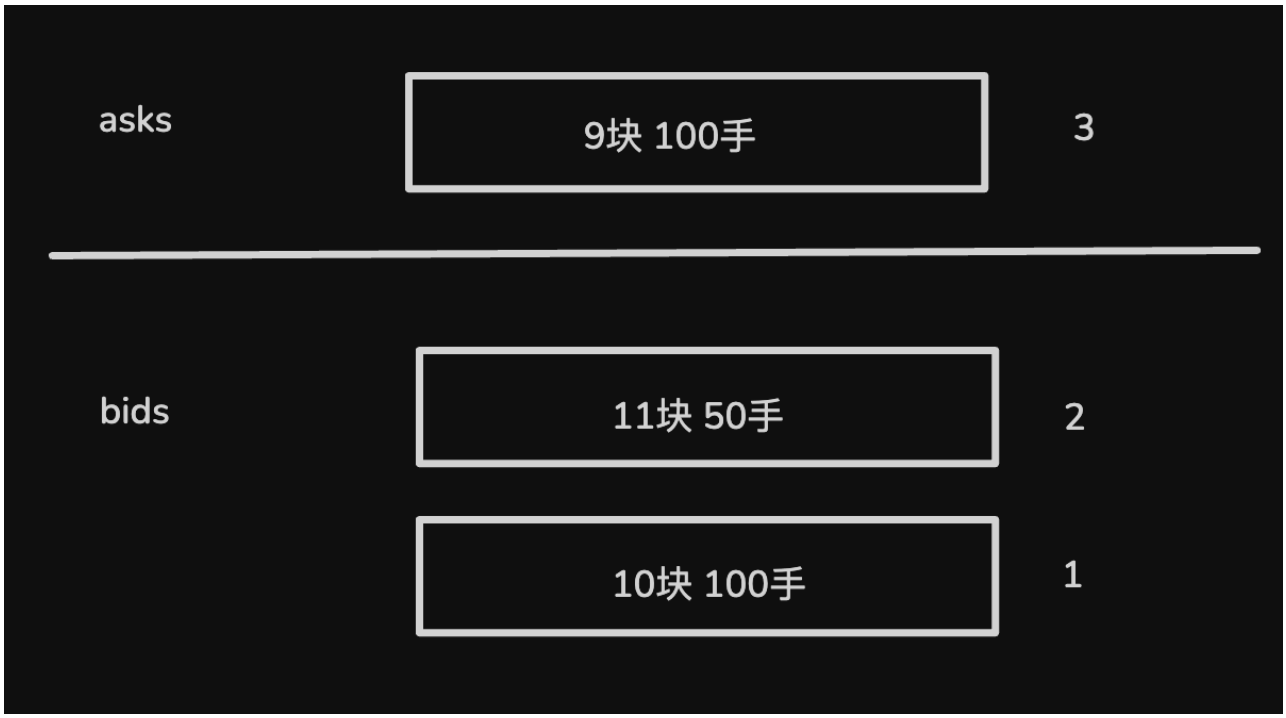
```
Problems  Output  Debug Console  Terminal  Ports
(base) ~/Code2/test/ g++ -O3 -std=c++11 main.cpp -o orderbook && ./orderbook

========== 深度 ==========
    卖单
     Price | Quantity
        12 | 10
--------------------------------
    买单
================================

Order 5 cancelled.

========== 深度 ==========
    卖单
     Price | Quantity
--------------------------------
    买单
================================
```

撮合逻辑



1、2、3既是订单id 也是时间顺序

但订单3出现时，是可以发生交易的（卖出价格较低）

交易顺序是 11块50手 交易完成后 是 10块 100手（先价格后时间）

结果符合预期

```
196     try {
197         ob.limitOrder(1, 10, 100, BUY, "TEST");
198         ob.limitOrder(2, 11, 50, BUY, "TEST");
199
200         ob.limitOrder(3, 9, 100, SELL, "TEST");
201
202         ob.displayDepth();
203
204
205         // ob.limitOrder(5, 12, 10, SELL, "TEST");
206         // ob.displayDepth();
```

```
Problems  Output  Debug Console  Terminal  Ports                    zsh
MATCH: Order 3 (SELL) matched with Order 1 (BUY) | Qty: 50 @ Price: 10

========== 深度 ==========
    卖单
     Price | Quantity
--------------------------------
        10 | 50
    买单
================================

(base) ~/Code2/test/ ./orderbook
--- Initializing Orders (Example from prompt) ---
MATCH: Order 3 (SELL) matched with Order 2 (BUY) | Qty: 50 @ Price: 11
MATCH: Order 3 (SELL) matched with Order 1 (BUY) | Qty: 50 @ Price: 10

========== 深度 ==========
    卖单
     Price | Quantity
--------------------------------
        10 | 50
    买单
================================
```

实习计划

1. 2027年末毕业
2. 实习可以持续6个月或者以上，本人2026年9月要去香港理工大学上学
3. 每周工作5-7天