Lighthouse *2019 Feb 01*

Task 1
The task is to implement an order book. Exchange matching engines use an order book to match orders from buyers and sellers. It is a simple mechanism which can be best described using example. Let's consider order flow for an imaginary company traded under symbol XYZ. Initially, an order book has no order. An anonymous market participant places a sell order for 5 shares at a price of 110 USD, order book will become :

```
ASK
110: 5
---
BID
```

A moment after that some other market participant places a buy order for 10 shares at a price of 90 USD :

```
ASK
110: 5
---
90: 10
BID
```

So far we have not had any trades as the lowest price from the seller (best ask) is 110 USD, and the highest price from the buyer (best bid) is 90 USD in other words currently there is no match between buyers and sellers, difference between best bid and best ask is called the spread and in this case equals to 20 USD. A few moments later, a new sell order of 10 shares at 110 USD is added :

```
ASK
110: 5 10
---
90: 10
BID
```

Note that a new order at the price level 110 has been added to the end of the order queue. It means that in case of match, the previous order (5 shares) will be executed before newly added order. A few moments later, several more orders from buyers and sellers are added to our order book, but no trades happened yet.

```
ASK
110: 5 10
105: 3 7
---
100: 4 6
90: 10 2 3
BID
```

Let's imagine some buyer places an aggressive order to buy 4 shares at the price of 105 USD. It will be an order which actually matches the best price from sellers (namely lowest price at 105 USD). The seller's order of 3 shares at price 105 USD was added earlier than another seller's order of 7 shares at 105 USD. Therefore seller's order of 3 shares will be matched first. As a result we will see a trade: 3 shares of XYZ were sold at 105 USD and the order book will look like :

```
ASK
110: 5 10
105: 7
---
100: 4 6
90: 10 2 3
BID
```

But we still have 1 share left from an aggressive buyer who wants to buy at 105 USD. Thus we will have partial match with seller's order at 105 USD. We will see second trade happening, 1 share of XYZ were sold at 105 USD and order book will now look like :

```
ASK
110: 5 10
105: 6
---
100: 4 6
90: 10 2 3
BID
```

In other words, that order from the aggressive buyer crossed the book and two trades were generated. Now imagine that some other aggressive seller wants to sell 23 shares at 80 USD. In other words, he wants to sell 23 shares no cheaper than 80 USD. First trade (or execution in other terminology) will be 4 shares at price of 100 USD because buy order of 4 shares at price level of 100 was added before buy order of 6 shares at price level 100. The order book will look like :

```
ASK
110: 5 10
105: 6
---
100: 6
90: 10 2 3
BID
```

Second trade will be 6 shares at price 100 USD :

```
ASK
110: 5 10
105: 6
------------
90: 10 2 3
BID
```

Third trade will be 10 shares at price 90 USD :

```
ASK
110: 5 10
105: 6
---
90: 2 3
BID
```

Forth trade will be 2 shares at price 90 USD :

```
ASK
110: 5 10
105: 6
---
90: 3
BID
```

And finally the last trade will be 1 share at price 90 USD (partial match or partial execution) :

```
ASK
110: 5 10
105: 6
---
90: 2
BID
```

Now let's say we have got a new buy order of 8 shares at the price 107 USD, 6 shares of XYZ were traded at 105 USD :

```
ASK
110: 5 10
---
90: 2
BID
```

We still have 2 more shares from a buyer who is willing to buy no higher than 107 USD but the best sell order right now is at 110 USD so we place a new order at the level of 107 USD and the order book will look like :

```
ASK
110: 5 10
---
107: 2
90: 2
BID
```

As you might noticed the first priority is price, the second priority is time when order was placed at certain price level. It is called price/time limit order book. There are delete orders if a trader wants to withdraw his or her order from market, each order has unique order id. If trader deletes order and then adds order, new order will be added to the end of queue. Your implementation of the order book should read market data file and build order book, print to stdout trades which happened and final state of order book. Market data file format : each line represents order added in chronological order. First byte of each line represents message type, `A` for new order and `X` for delete order. The rest of a line : order_id, side, quantity, price. For example : `A,16113575,B,18,585`. It decoded as :

```
A = add new order
16113575 = order id
B = buy order
18 = number of shares to buy
585 = price
```

Example of input file:

```
A,100000,S,1,1075
A,100001,B,9,1000
A,100002,B,30,975
A,100003,S,10,1050
A,100004,B,10,950
A,100005,S,2,1025
A,100006,B,1,1000
X,100004,B,10,950
A,100007,S,5,1025
A,100008,B,3,1050
X,100008,B,3,1050
X,100005,S,2,1025
```

Task 2

Imagine you are quoting your orders in an exchange, you have outstanding buy and sell orders registered in exchange order book, now market condition have been changed and you want to have a little different set of outstanding orders in exchange order book, so you are about to send the smallest possible number of orders (new, delete, modify) to exchange in order to transform your initial set of order into desired one. For example, here are your current outstanding orders :

```
ASK
110: 5 3
---
90: 10 2
85: 6
BID
```

And your desired state of outstanding orders is :

```
ASK
110: 4 3
108: 8
---
90: 10 2
85: 6
BID
```

What you would do is to modify order :

```
M,111222333,S,4,110
```

`M` is a new order type (modify) which does not exist in previous task. It means to modify order with id `111222333` to have new quantity of `4`. Where `111222333` is order id of price `110` and quantity `5`. Second order would be :

```
A,999888777,S,8,108
```

You cannot use executions (trades) to transform initial set into final set of outstanding orders. Now implement the algo.

Answer

```
namespace algo
{
    struct order_info
    {
        unsigned long order_id;
        bool is_buy_order;
        unsigned short share;
        unsigned short price;
    };

    struct order
    {
        char action;
        order_info info;
    };
```

Remark 1 : Five elements in an order

```
    struct trade
    {
        unsigned short transaction_share;
        unsigned short transaction_price;
        unsigned long bid_order_id;
        unsigned long ask_order_id;
    };
```

Remark 2 : Four elements in a trade

```
    inline std::ostream& operator<<(std::ostream& os, const order_info& x)
    {
        os << x.order_id << "," << (x.is_buy_order? 'B':'S') << ","  << x.share << "," << x.price;
        return os;
    }

    inline std::ostream& operator<<(std::ostream& os, const order& x)
    {
        os << x.action << "," << x.info;
        return os;
    }

    inline std::ostream& operator<<(std::ostream& os, const trade& x)
    {
        os << "trade " << x.transaction_share << " @" << x.transaction_price << " between "
                        << x.bid_order_id << " and " << x.ask_order_id;
        return os;
    }

    inline std::istream& operator>>(std::istream& is, order_info& x)
    {
        char temp,side;
        is >> x.order_id >> temp >> side >> temp >> x.share >> temp >> x.price;

        if (side == 'B') x.is_buy_order = true;
        else x.is_buy_order = false;
        return is;
    }

    inline std::istream& operator>>(std::istream& is, order& x)
    {
        char temp;
        is >> x.action >> temp >> x.info;
        return is;
    }

    // ************** //
    // *** TASK 1 *** //
    // ************** //
    class order_book
    {
    public:
        order_book(std::ifstream& ifs)
        {
            order x;
            while(!ifs.eof()) // assume that the datafile does not end with newline char
            {
                ifs >> x;
                add_order(x);
            }
        }

        inline void print() const
        {
            for(const auto& x : trades) std::cout << "\n" << x;
            for(const auto& x : warns)  std::cout << "\n" << x;
            std::cout << "\n================";
            std::cout << "\n[ask]";
            for(auto x = ask_side.rbegin(); x!=ask_side.rend(); ++x)
            {
                std::cout << "\n" << x->first << " : ";
                for(const auto& y : x->second)
                {
                    std::cout << y.order_id << "(" << y.share << ") ";
                }
            }
            std::cout << "\n-----------------";
            for(auto x = bid_side.begin(); x!=bid_side.end(); ++x)
            {
                  std::cout << "\n" << x->first << " : ";
                  for (const auto& y : x->second)
                  {
                        std::cout << y.order_id << "(" <<  y.share << ") ";
                  }
            }
            std::cout << "\n[bid]";
            std::cout << "\n================"; std::cout << "\n\n";
      }
```

Remark 3 : Three cases in add order

(1) hit no order

(2) hit some orders and totally filled

(3) hit some orders and remaining live

```
private:
      inline void add_order(const order& x)
      {
            if (x.action == 'A')
            {
                  order_info temp = x.info;
                  if (temp.is_buy_order)
                  {
                        while(hit_one_ask(temp));
                        if (temp.share > 0) bid_side[temp.price].push_back(temp);
                  }
                  else
                  {
                        while(hit_one_bid(temp));
                        if (temp.share > 0) ask_side[temp.price].push_back(temp);
                  }
            }
            else if (x.action == 'X')
            {
                  bool status = false;
                  if (x.info.is_buy_order) status = cancel_one_bid(x.info);
                  else status = cancel_one_ask(x.info);

                  if (!status)
                  {
                        std::stringstream ss;
                        ss << x << " failed";
                        warns.push_back(ss.str());
                  }
            }
      }

      inline bool hit_one_ask(order_info& x)
      {
            trade t;
            if (x.share > 0 && !ask_side.empty() && x.price >= ask_side.begin()->first)
            {
                  if (x.share >= ask_side.begin()->second.front().share)
                  {
                        t.bid_order_id = x.order_id;
                        t.ask_order_id = ask_side.begin()->second.front().order_id;
                        t.transaction_share = ask_side.begin()->second.front().share;
                        t.transaction_price = ask_side.begin()->first;

                        // update share, pop queue and pop map
                        x.share -= t.transaction_share;
                        ask_side.begin()->second.pop_front();
                        if (ask_side.begin()->second.empty()) ask_side.erase(ask_side.begin());
                  }
                  else
                  {
                        t.bid_order_id = x.order_id;
                        t.ask_order_id = ask_side.begin()->second.front().order_id;
                        t.transaction_share = x.share;
                        t.transaction_price = ask_side.begin()->first;

                        // update share
                        ask_side.begin()->second.front().share -= x.share;
                        x.share = 0;
                  }
                  trades.push_back(t);
                  return true;
            }
            return false;
      }

      inline bool hit_one_bid(order_info& x)
      {
            trade t;
            if (x.share > 0 && !bid_side.empty() && x.price <= bid_side.begin()->first)
            {
                  if (x.share >= bid_side.begin()->second.front().share)
                  {
                        t.bid_order_id = bid_side.begin()->second.front().order_id;
                        t.ask_order_id = x.order_id;
                        t.transaction_share = bid_side.begin()->second.front().share;
                        t.transaction_price = bid_side.begin()->first;
                        // update share, pop queue and pop map
                        x.share -= t.transaction_share;
                        bid_side.begin()->second.pop_front();
                        if (bid_side.begin()->second.empty()) bid_side.erase(bid_side.begin());
                  }
                  else
                  {
                        t.bid_order_id = bid_side.begin()->second.front().order_id;
                        t.ask_order_id = x.order_id;
                        t.transaction_share = x.share;
                        t.transaction_price = bid_side.begin()->first;

                        // update share
                        bid_side.begin()->second.front().share -= x.share;
                        x.share = 0;
                  }
                  trades.push_back(t);
                  return true;
            }
            return false;
      }

      inline bool cancel_one_bid(const order_info& x)
      {
            if (bid_side.find(x.price)!=bid_side.end())
            {
                  for(auto iter = bid_side[x.price].begin(); iter!=bid_side[x.price].end(); ++iter)
                  {
                        if (x.order_id == iter->order_id)
                        {
                              bid_side[x.price].erase(iter);
                              if (bid_side[x.price].empty()) bid_side.erase(x.price);
                              return true;
                        }
                  }
            }
            return false;
      }

      inline bool cancel_one_ask(const order_info& x)
      {
            if (ask_side.find(x.price)!=ask_side.end())
            {
                  for(auto iter = ask_side[x.price].begin(); iter!=ask_side[x.price].end(); ++iter)
                  {
                        if (x.order_id == iter->order_id)
                        {
                              ask_side[x.price].erase(iter);
                              if (ask_side[x.price].empty()) ask_side.erase(x.price);
                              return true;
                        }
                  }
            }
            return false;
      }

public:
      std::map<unsigned short, std::deque<order_info>, std::greater<unsigned short> > bid_side;
      std::map<unsigned short, std::deque<order_info> > ask_side;
      std::vector<trade> trades;
      std::vector<std::string> warns;
};

// ************** //
// *** TASK 2 *** //
// ************** //
namespace imp // implementation by recursion (dynamic programming stuff)
{
      // index 0 denotes current container
      // index 1 denotes desired container
      template<typename ITER>
      void min_dist(ITER i0_begin, ITER i0_end, ITER i1_begin, ITER i1_end, std::vector<order>& orders)
      {
            if (i0_begin == i0_end)
            {
                  if (i1_begin == i1_end) return;
                  else
                  {
                        for(ITER i=i1_begin; i!=i1_end; ++i) { order x {'A', *i}; orders.push_back(x); }
                  }
            }
            else
            {
                  if (i1_begin == i1_end)
                  {
                        for(ITER i=i0_begin; i!=i0_end; ++i) { order x {'X', *i}; orders.push_back(x); }
                  }
                  else if (i0_begin->share == i1_begin->share)
                  {
                        min_dist(i0_begin+1, i0_end, i1_begin+1, i1_end, orders);
                  }
```

```cpp
                    else
                    {
                         // Not efficient as it involves copy of vector, need improvement
                         std::vector<order> orders0; // if we del the first current order
                         std::vector<order> orders1; // if we add the first desired order
                         std::vector<order> orders2; // if we modify the front

                         min_dist(i0_begin+1, i0_end, i1_begin, i1_end, orders0);
                         min_dist(i0_begin, i0_end, i1_begin+1, i1_end, orders1);
                         min_dist(i0_begin+1, i0_end, i1_begin+1, i1_end, orders2);

                         if (orders0.size() <= orders1.size() && orders0.size() <= orders2.size())
                         {
                              order x {'X', *i0_begin};
                              orders.push_back(x);
                              orders.insert(orders.end(), orders0.begin(), orders0.end());
                         }
                         else if (orders1.size() <= orders0.size() && orders1.size() <= orders2.size())
                         {
                              order x {'A', *i1_begin};
                              orders.push_back(x);
                              orders.insert(orders.end(), orders1.begin(), orders1.end());
                         }
                         else
                         {
                              order x {'M', *i0_begin};
                              x.info.share = i1_begin->share;
                              orders.push_back(x);
                              orders.insert(orders.end(), orders2.begin(), orders2.end());
                         }
                    }
               }
          }

          void min_dist(const deque<order_info>& q0, const deque<order_info>& q1, vector<order>& orders)
          {
               return min_dist(q0.begin(), q0.end(), q1.begin(), q1.end(), orders);
          }
     }

     inline void min_distance_between(const order_book& bk0, const order_book& bk1, std::vector<order>& orders)
     {
          std::deque<order_info> empty_q;
          auto i0 = bk0.bid_side.begin();
          auto i1 = bk1.bid_side.begin();
          auto j0 = bk0.ask_side.begin();
          auto j1 = bk1.ask_side.begin();

          // *** Bid side *** //
          while(i0!=bk0.bid_side.end() && i1!=bk1.bid_side.end())
          {
            if (i0->first > i1->first)      { imp::min_dist(i0->second, empty_q, orders);      ++i0;      }
            else if (i0->first < i1->first) { imp::min_dist(empty_q, i1->second, orders);      ++i1;      }
            else                            { imp::min_dist(i0->second, i1->second, orders);   ++i0; ++i1; }
          }
          while(i0!=bk0.bid_side.end())     { imp::min_dist(i0->second, empty_q, orders);      ++i0;      }
          while(i1!=bk1.bid_side.end())     { imp::min_dist(empty_q, i1->second, orders);      ++i1;      }

          // *** Ask side *** //
          while(j0!=bk0.ask_side.end() && j1!=bk1.ask_side.end())
          {
            if (j0->first < j1->first)      { imp::min_dist(j0->second, empty_q, orders);      ++j0;      }
            else if (j0->first > j1->first) { imp::min_dist(empty_q, j1->second, orders);      ++j1;      }
            else                            { imp::min_dist(j0->second, j1->second, orders);   ++j0; ++j1; }
          }
          while(j0!=bk0.ask_side.end())     { imp::min_dist(j0->second, empty_q, orders);      ++j0;      }
          while(j1!=bk1.ask_side.end())     { imp::min_dist(empty_q, j1->second, orders);      ++j1;      }
     }
}

// *************** //
// *** Testing *** //
// *************** //
void test_orderbook()
{
     // Please input 2 orderbook, then ....
     std::ifstream ifs1("E:/DOC/library/Alglib/AlgoLib/source_test/order1.txt");
     std::ifstream ifs2("E:/DOC/library/Alglib/AlgoLib/source_test/order2.txt");
     algo::order_book obk1(ifs1);
     algo::order_book obk2(ifs2);

     std::cout << "\n[Orderbook1]"; obk1.print();
     std::cout << "\n[Orderbook2]"; obk2.print();
     std::cout << "\n\n\norderbook distance is : ";

     // min distance algo can help to find the min distance between the orderbook using recursion
     std::vector<algo::order> orders;
     algo::min_distance_between(obk1, obk2, orders);
     for(const auto& x : orders) std::cout << "\n" << x;
}
```