VE281 Homework IV

Electronic Trading

Name: Tianyi Ge **Stu Number:** 516370910168

1 Appendix

A Source Codes

Program code 1: a4.cpp

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <string>
#include <sstream>
#include <getopt.h>
#include <vector>
#include "trade.h"

bool v_flag = false, m_flag = false, p_flag = false, t_flag = false, g_flag =
    ↪ false;
unsigned g_num = 0;
std::vector<std::string> tttList;
std::unordered_set<std::string> tttEquitySet;

extern unsigned currentTimeStamp;

Trade& trade = Trade::getInstance();

void getoptions(const int &argc, char **argv) {
    static option long_options[] = {
          {"verbose",   no_argument,       0, 'v'},
          {"median",    no_argument,       0, 'm'},
```

```
            {"midpoint",  no_argument,       0, 'p'},
            {"transfers", no_argument,       0, 't'},
            {"ttt",       required_argument, 0, 'g'},
            {0, 0, 0, 0}
        };

        int option_index = 0, c = -1;
        while ((c = getopt_long(argc, argv, "vmptg:", long_options,
        ↪  &option_index)) != -1) {
            switch(c) {
                case 'v':
                    v_flag = 1; break;
                case 'm':
                    m_flag = 1; break;
                case 'p':
                    p_flag = 1; break;
                case 't':
                    t_flag = 1; break;
                case 'g': {
                    g_flag = 1;
                    g_num++;
                    tttEquitySet.emplace(optarg);
                    tttList.emplace_back(std::move(optarg));
                    break;
                }
                default: break;
            }
        }
    }

    void execute() {
        unsigned id = 0;
        string cmd;
        while (getline(std::cin, cmd) && !cmd.empty()) {
            std::istringstream sin(cmd);
            int duration;
            unsigned timeStamp, price, quantity;
            std::string name, sell_buy, equity;
```

```
60          char nop;
61          sin >> timeStamp >> name >> sell_buy >> equity >> nop >> price >> nop
            ↪  >> quantity >> duration;
62          if (timeStamp != currentTimeStamp) {
63              if (m_flag) trade.medianPrint();
64              if (p_flag) trade.midpointPrint();
65              currentTimeStamp = timeStamp;
66          }
67          if (sell_buy == "SELL") {
68              trade.sellerMatch(id++, timeStamp, std::move(name),
                ↪  std::move(equity), price, quantity, duration);
69          } else {
70              trade.buyerMatch(id++, timeStamp, std::move(name),
                ↪  std::move(equity), price, quantity, duration);
71          }
72      }
73      if (m_flag) trade.medianPrint();
74      if (p_flag) trade.midpointPrint();
75      trade.summary();
76  }
77
78  int main(int argc, char *argv[]) {
79      std::ios::sync_with_stdio(false);
80      std::cin.tie(0);
81      getoptions(argc, argv);
82      trade.setTTTList(std::move(tttList), std::move(tttEquitySet), g_num);
83      trade.setFlag(v_flag, m_flag, p_flag, t_flag, g_flag);
84      execute();
85      return 0;
86  }
```

Program code 2: trade.h

```
1   #ifndef _TRADE_H_
2   #define _TRADE_H_
3
4   #include <vector>
5   #include <unordered_map>
```

```cpp
#include <unordered_set>
#include <queue>
#include <set>
#include <map>
#include <memory>

using std::unordered_map;
using std::map;
using std::multimap;
using std::string;
using std::priority_queue;
using std::vector;
using std::move;
using std::unordered_set;
using std::set;

class Trade {
private:
    typedef string EquityNameType;
    struct client {
        typedef std::shared_ptr<client> Ptr;
        unsigned id, timeStamp;
        string name;
        EquityNameType equity;
        unsigned price;
        unsigned quantity;
        int duration;
        client(unsigned &id, unsigned &timeStamp, string &&name, string
        ↪ &&equity, unsigned &price, unsigned &quantity, int &duration):
        ↪ id(id), timeStamp(timeStamp), name(move(name)),
        ↪ equity(move(equity)), price(price), quantity(quantity),
        ↪ duration(duration) {}
    };

    struct greater_t {
        bool operator()(const client::Ptr x, const client::Ptr y) const {
            if (x->price == y->price) return x->id > y->id;
            return x->price > y->price;
```

```
40          }
41      };
42
43      struct less_t {
44          bool operator()(const client::Ptr x, const client::Ptr y) const {
45              if (x->price == y->price) return x->id > y->id;
46              return x->price < y->price;
47          }
48      };
49
50      struct PriceMedian {
51          unsigned n;
52          priority_queue<unsigned, vector<unsigned>, std::greater<unsigned> >
            ↪ minheap;
53          priority_queue<unsigned, vector<unsigned>,    std::less<unsigned> >
            ↪ maxheap;
54          PriceMedian(): n(0) {}
55          void insert(unsigned x);
56          unsigned getMedian() const {return (n&1) ? maxheap.top() :
            ↪ ((maxheap.top()+minheap.top()) >> 1);}
57      };
58
59      struct TransferInfo {
60          unsigned bought, sold;
61          int netValue;
62          TransferInfo() {bought = sold = 0; netValue = 0;}
63      };
64
65      struct orderBook {
66          priority_queue<client::Ptr, vector<client::Ptr>, greater_t> seller; //
            ↪ seller, price increasing
67          priority_queue<client::Ptr, vector<client::Ptr>, less_t> buyer; //
            ↪ buyer, price decreasing
68      };
69
70      struct tttOrder{
71          typedef std::shared_ptr<tttOrder> Ptr;
72          bool isBuy;
```

```
73          unsigned price, timeStamp;
74          tttOrder(bool isBuy, unsigned &price, unsigned &timeStamp):
            ↪ isBuy(isBuy), price(price), timeStamp(timeStamp) {}
75      };
76
77      unordered_map<EquityNameType, orderBook> orderBookMap;
78
79      unordered_set<EquityNameType> equitySet;
80      set<EquityNameType> equityList; // for midpoint output
81
82      unordered_set<string> nameSet;
83      map<string, TransferInfo> transfers; // for transfers
84
85      map<EquityNameType, PriceMedian> pricemedian; // EquityNameType dictionary
        ↪ order
86
87      vector<EquityNameType> tttList; // for ttt
88
89      unordered_set<EquityNameType> tttEquitySet;
90
91      unordered_map<EquityNameType, vector<tttOrder::Ptr> > tttMap; // from
        ↪ string to order list
92
93      unsigned c_earnings = 0, total = 0, trade_num = 0, share_num = 0;
94
95      bool v_flag, m_flag, p_flag, t_flag, g_flag;
96
97      void transferPrint();
98
99      void tttPrint();
100
101 private:
102     static std::unique_ptr<Trade> instance;
103     Trade() = default;
104
105 public:
106     Trade &operator=(const Trade &) = delete;
107         Trade &operator=(Trade &&) = delete;
```

```
108         Trade(const Trade &) = delete;
109         Trade(Trade &&) = delete;
110         ~Trade() = default;
111
112     static Trade &getInstance();
113
114     void setTTTList(vector<string> &&x, unordered_set<string> &&y, unsigned
        ↪ g_num) {tttList = move(x); tttEquitySet = move(y);
        ↪ tttMap.reserve(g_num);}
115
116     void setFlag(const bool &vflag, const bool &mflag, const bool &pflag,
        ↪ const bool &tflag, const bool &gflag) {
117         v_flag = vflag;
118         m_flag = mflag;
119         p_flag = pflag;
120         t_flag = tflag;
121         g_flag = gflag;
122     }
123
124     void sellerMatch(unsigned id,
125                      unsigned timeStamp,
126                      string &&name,
127                      string &&equity,
128                      unsigned price,
129                      unsigned quantity,
130                      int duration);
131
132     void buyerMatch(unsigned id,
133                      unsigned timeStamp,
134                      string &&name,
135                      string &&equity,
136                      unsigned price,
137                      unsigned quantity,
138                      int duration);
139
140     void medianPrint() const;
141
142     void midpointPrint();
```

```

    void summary();
};

#endif
```

Program code 3: trade.cpp

```
#include <iostream>
#include <algorithm>
#include "trade.h"
#define log_v(buyer, q, equity, seller, p) std::cout << buyer << " purchased " << q << " shares of " << equity << " from " << seller << " for £" << p << "/share\n"

#define expired(x) (x->duration > 0 && x->duration + x->timeStamp <= currentTimeStamp)

#define log_p()\
if (equitySet.find(equity) == equitySet.end()) {\
    equitySet.emplace(equity);\
    equityList.insert(equity);\
}

#define log_g(x)\
if (tttEquitySet.find(equity) != tttEquitySet.end()) {\
    tttMap[equity].emplace_back(std::make_shared<tttOrder>(x, price, timeStamp));\
}

#define log_m() pricemedian[equity].insert(p)

#define log_t()\
if(nameSet.find(name) == nameSet.end()) {\
    nameSet.emplace(name);\
    transfers.emplace(std::make_pair(name, TransferInfo()));\
}

```

```cpp
const unsigned INF = (1 << 30);

std::unique_ptr<Trade> Trade::instance = nullptr;

unsigned currentTimeStamp = 0;

Trade &Trade::getInstance() {
    if (Trade::instance == nullptr) {
        instance = std::unique_ptr<Trade>(new Trade);
    }
    return *instance;
}

void Trade::PriceMedian::insert(unsigned x) {
    if (n&1) { // odd
        if (x >= maxheap.top()) minheap.push(x);
        else {
            unsigned vic = maxheap.top();
            maxheap.pop();
            minheap.push(vic);
            maxheap.push(x);
        }
    } else { // even
        if (n == 0 || x <= minheap.top()) maxheap.push(x);
        else {
            unsigned vic = minheap.top();
            minheap.pop();
            maxheap.push(vic);
            minheap.push(x);
        }
    }
    n++;
}

void Trade::sellerMatch(unsigned id,
                        unsigned timeStamp,
                        string &&name,
                        string &&equity,
```

```
65                          unsigned price,
66                          unsigned quantity,
67                          int duration) {
68      if (t_flag) log_t();
69      if (p_flag) log_p();
70      if (g_flag) log_g(false);
71      auto &buyer = orderBookMap[equity].buyer;
72      while (!buyer.empty() && quantity > 0) {
73          if (expired(buyer.top())) { // expire
74              buyer.pop();
75              continue;
76          }
77          if (buyer.top()->price < price) break; // price not match, the latter
        ↪  buyer price are lower, cannot match anyway
78
79          // do transaction
80          unsigned q = std::min(quantity, buyer.top()->quantity);
81          unsigned p = buyer.top()->price; // buyer comes first
82          unsigned money = (p * q);
83          unsigned c_fee = money / 100;
84          buyer.top()->quantity -= q;
85          quantity -= q;
86          //default output
87          c_earnings += (c_fee << 1);
88          total += money;
89          ++trade_num;
90          share_num += q;
91
92          // options
93          if (v_flag) log_v(buyer.top()->name, q, equity, name, p);
94          if (m_flag) log_m();
95          if (t_flag) {
96              transfers[name].sold += q;
97              transfers[name].netValue += money;
98              transfers[buyer.top()->name].bought += q;
99              transfers[buyer.top()->name].netValue -= money;
100         }
101         // pop
```

```
102             if (buyer.top()->quantity == 0) buyer.pop();
103         }
104 
105         if (quantity > 0 && duration != 0) { // add to book
106             orderBookMap[equity].seller.emplace(std::make_shared<client>(id,
                ↪  timeStamp, move(name), move(equity), price, quantity, duration));
107         }
108     }
109 
110     void Trade::buyerMatch(unsigned id,
111                            unsigned timeStamp,
112                            string &&name,
113                            string &&equity,
114                            unsigned price,
115                            unsigned quantity,
116                            int duration) {
117         if (t_flag) log_t();
118         if (p_flag) log_p();
119         if (g_flag) log_g(true);
120         auto &seller = orderBookMap[equity].seller;
121         while (!seller.empty() && quantity > 0) {
122             if (expired(seller.top())) { // expire
123                 seller.pop();
124                 continue;
125             }
126             if (seller.top()->price > price) break; // price not match, the latter
                ↪  seller price are higher, cannot match anyway
127 
128             // do transaction
129             unsigned q = std::min(quantity, seller.top()->quantity);
130             unsigned p = seller.top()->price; // seller comes first
131             unsigned money = (p * q);
132             unsigned c_fee = money / 100;
133             seller.top()->quantity -= q;
134             quantity -= q;
135             //default output
136             c_earnings += (c_fee << 1);
137             total += money;
```

```
138             ++trade_num;
139             share_num += q;
140 
141             // options
142             if (v_flag) log_v(name, q, equity, seller.top()->name, p);
143             if (m_flag) log_m();
144             if (t_flag) {
145                 transfers[name].bought += q;
146                 transfers[name].netValue -= money;
147                 transfers[seller.top()->name].sold += q;
148                 transfers[seller.top()->name].netValue += money;
149             }
150             // pop
151             if (seller.top()->quantity == 0) seller.pop();
152         }
153 
154         if (quantity > 0 && duration != 0) { // add to book
155             orderBookMap[equity].buyer.emplace(std::make_shared<client>(id,
                ↪  timeStamp, move(name), move(equity), price, quantity, duration));
156         }
157     }
158 
159     void Trade::medianPrint() const {
160         for (auto &equity : pricemedian) {
161             std::cout << "Median match price of " << equity.first << " at time "
                ↪  << currentTimeStamp << " is $" << equity.second.getMedian() <<
                ↪  '\n';
162         }
163     }
164 
165     void Trade::midpointPrint() {
166         for (auto &equity : equityList) {
167             auto &buyer = orderBookMap[equity].buyer;
168             auto &seller = orderBookMap[equity].seller;
169             while (!buyer.empty() && expired(buyer.top())) buyer.pop();
170             while (!seller.empty() && expired(seller.top())) seller.pop();
171             std:: cout << "Midpoint of " << equity << " at time " <<
                ↪  currentTimeStamp << " is";
```

```
172             if (buyer.empty() || seller.empty()) {
173                 std::cout << " undefined\n";
174             } else {
175                 std::cout << " $" << ((buyer.top()->price + seller.top()->price)
                    ↪  >> 1) << '\n';
176             }
177         }
178     }
179
180     void Trade::summary() {
181         std::cout << "---End of Day---\n";
182         std::cout << "Commission Earnings: $" << c_earnings << '\n';
183         std::cout << "Total Amount of Money Transferred: $" << total << '\n';
184         std::cout << "Number of Completed Trades: " << trade_num << '\n';
185         std::cout << "Number of Shares Traded: " << share_num << '\n';
186         if (t_flag) transferPrint();
187         if (g_flag) tttPrint();
188     }
189
190     void Trade::transferPrint() {
191         for (auto &info : transfers) {
192             std::cout << info.first << " bought " << info.second.bought << " and
                    ↪  sold " << info.second.sold << " for a net transfer of $" <<
                    ↪  info.second.netValue << '\n';
193         }
194     }
195
196     void Trade::tttPrint() {
197         for (auto &equity : tttList) {
198             auto &orders = tttMap[equity]; // vector of tttOrder::Ptr
199             int ts1_cand, ts1 = -1, ts2 = -1;
200             int profit = -INF;
201             unsigned seller_price = INF;
202             bool has_seller = false;
203             for (auto &order : orders) {
204                 if (!order->isBuy) { // seller
205                     has_seller = true;
206                     if (order->price < seller_price) { // update seller price
```

```
207                     ts1_cand = order->timeStamp;
208                     seller_price = order->price;
209                 }
210             } else { // buyer
211                 if (has_seller && (signed)(order->price - seller_price) >
                    ↪ profit) { // new profit
212                     ts1 = ts1_cand;
213                     ts2 = order->timeStamp;
214                     profit = (signed)(order->price - seller_price);
215                 }
216             }
217         }
218         std::cout << "Time travelers would buy " << equity << " at time: " <<
            ↪ ts1 << " and sell it at time: " << ts2 << '\n';
219     }
220 }
```