Middle East Technical University | Department of Computer Engineering

CENG 140

C Programming

Spring' 2024-2025
Take-Home Exam 2

Emre Külah
kulah@ceng.metu.edu.tr
Due date: June 8, 2025, Sunday, 23:59

CENG STOCK MARKET

1 Introduction

In this assignment, you will implement a simplified stock market simulation using the C programming language. The goal is to practice using `structs`, dynamic memory allocation, arrays, and linked lists to model a real-world system.

Each company in the market is represented by a name, a stock abbreviation, and a display order. The population consists of individuals (represented with a common `Person` structure) who can submit buy and sell requests for company shares.
The stock market maintains two linked lists for each company:

- **Buy requests**, sorted by **decreasing** price-per-share (highest bidder first)
- **Sell requests**, sorted by **increasing** price-per-share (lowest offer first)

When a new buy request is submitted:

- It is matched against the lowest-priced sell requests.
- If the buyer's price is greater than or equal to a sell price, a trade occurs.
- The trade continues until the buyer's full requested amount is fulfilled or no suitable sellers remain.

When a new sell request is submitted:

- It is matched against the highest-priced buy requests.
- If the seller's price is less than or equal to a buy price, a trade occurs.
- The trade continues until the seller's full offered amount is sold or no suitable buyers remain.

Important constraints include:

- Matching logic must respect the sorted ordering of both request queues.

You will be given a partial codebase including struct definitions and function prototypes. Your task is to complete the logic, maintain list consistency, and implement the trading mechanism faithfully.

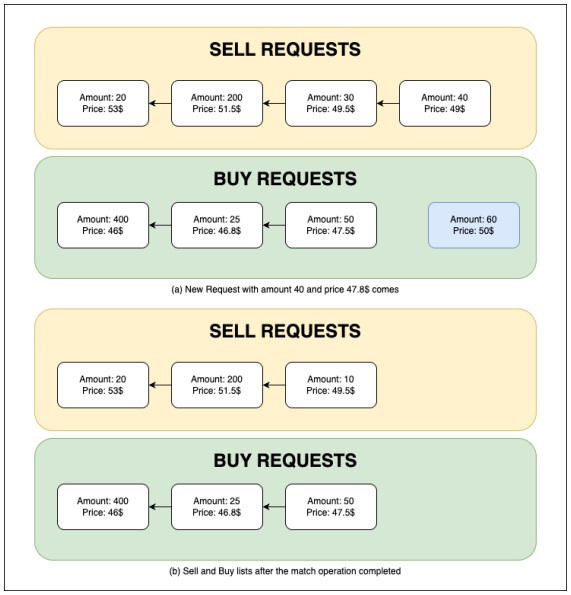

Figure 1: (a) Example of an incoming buy request fully fulfilled.

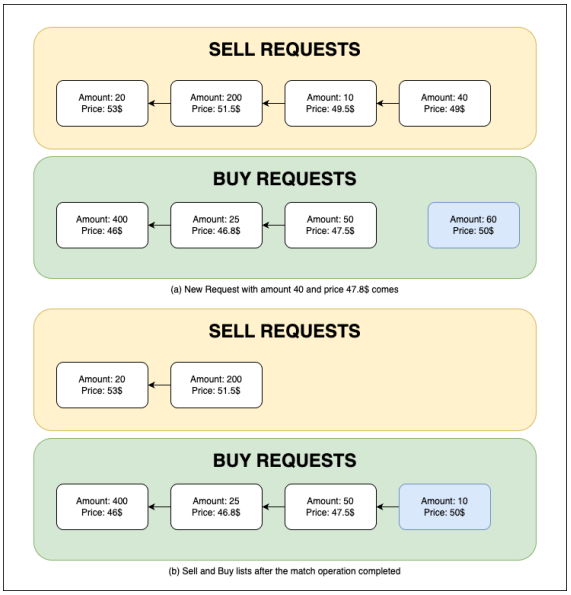

Figure 2: (b) Example of an incoming buy request partially fulfilled.

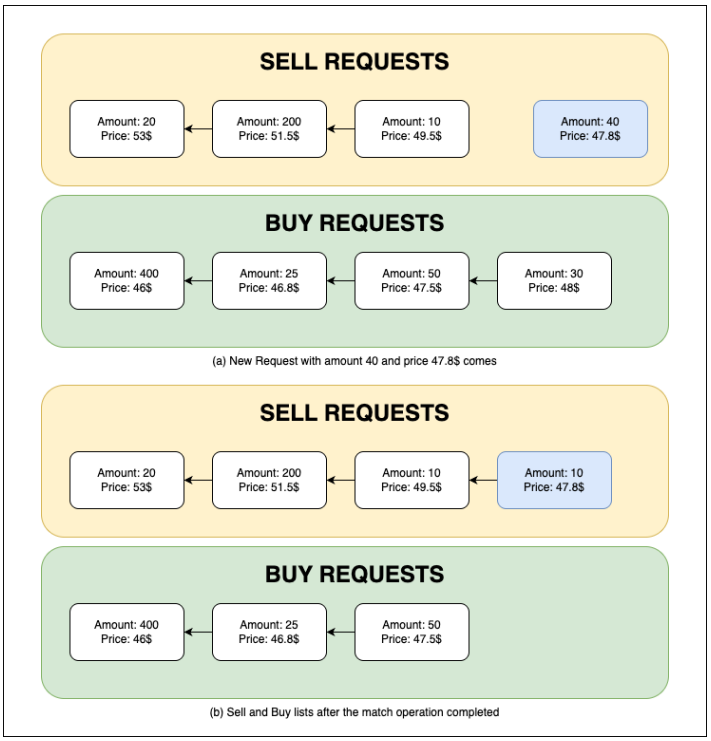

Figure 3: (c) Example of an incoming sell request partially fulfilled.

Figure 4: Illustration of different match outcomes in the stock market simulation.

2 Data Structures

The following data structures are used to represent the core components of the stock market system. You are provided with these structures in the `stock_market.h` file.

- `Person`
  Represents an individual in the population.
  - `char name[MAX_NAME_LEN];` – name of the person
  - `int id;` – unique identifier
- `Request`
  Holds the basic data for a buy or sell request.

  - `Person *person;` – requester
  - `int amount;` – number of shares
  - `float price_per_share;` – price per share

- `RequestNode`
  A node in a singly linked list used for buy/sell queues.
  - `Request data;` – request information
  - `struct RequestNode *next;` – pointer to next node

- `Company`
  Represents a company listed on the stock market.
  - `char name[MAX_NAME_LEN];` – full name of the company
  - `char stock_abbreviation[10];` – ticker code
  - `int market_order;` – position in the stock market entries

- `StockMarketEntry`
  Contains all buy and sell requests related to a specific company.
  - `Company *company;` – pointer to the company
  - `RequestNode *buy_requests;` – head of the buy request list
  - `RequestNode *sell_requests;` – head of the sell request list

- `StockMarket`
  The overall stock market containing multiple company entries.
  - `StockMarketEntry entries[MAX_COMPANIES];` – array of tracked companies
  - `int company_count;` – number of companies registered

3 Tasks

You are expected to implement the following main functions using the provided data structures. You are free to define additional helper functions as needed to structure your implementation.

- Create Person (10 points)
- Create Company (10 points)
- Create Market (20 points)
- Buy/Sell Request (60 points)

Task details are defined below;

- `void insert_buy_request(StockMarket *market, Person **people, int people_count, char *buyer_name, char *company_abbr, int amount, float price);`
  Arguments: A pointer to the `StockMarket`, an array of pointers to people, the number of people, the buyer's name, the company's abbreviation, the amount to buy, and the bid price per share.
  Returns: Nothing.
  Purpose: Finds the buyer in the population and submits a buy request. Matches it against available sell requests. Unmatched remainder is inserted into the sorted buy queue.
  Print: For each match:

  ```
  Trade executed: <amount> shares of <stock_abbr> at <sell_price> between <seller>, <buyer>
  ```

- `void insert_sell_request(StockMarket *market, Person **people, int people_count, char *seller_name, char *company_abbr, int amount, float price);`
  Arguments: A pointer to the `StockMarket`, an array of pointers to people, the number of people, the seller's name, the company's abbreviation, the amount to sell, and the ask price per share.
  Returns: Nothing.
  Purpose: Finds the seller in the population and submits a sell request. Matches it against available buy requests. Unmatched remainder is inserted into the sorted sell queue.
  Print: For each match:

  ```
  Trade executed: <amount> shares of <stock_abbr> at <sell_price> between <seller>, <buyer>
  ```

- `void print_market(StockMarket *market);`
  Arguments: A pointer to the `StockMarket`.
  Returns: Nothing.
  Purpose: Prints the current state of all buy and sell queues per company.
  Note1: Printing format is shown in 4.
  Note2: Indentations will be created with tab character not with whitespaces.

- `Person* create_person(char *name, int id);`
  Arguments: A name and unique ID.
  Returns: A pointer to a new `Person`.
  Purpose: Dynamically allocates and initializes a person struct.
- `Company* create_company(char *name, char *abbr, int order);`
  Arguments: A company name, stock abbreviation, and order index.
  Returns: A pointer to a new `Company`.
  Purpose: Dynamically allocates and initializes a company struct.
- `void add_company_to_market(StockMarket *market, Company *company);`
  Arguments: A pointer to the market and the new company.
  Returns: Nothing.
  Purpose: Registers the company in the market and initializes its order book.

4 Example

Sample Code

```
int main(void) {
    StockMarket market;
    Person *population[MAX_PEOPLE];
    int people_count = 0;
    Company *c1;

    market.company_count = 0;

    population[people_count++] = create_person("Alice", 1);
    population[people_count++] = create_person("Bob", 2);
    population[people_count++] = create_person("Charlie", 3);
    population[people_count++] = create_person("Diana", 4);
    population[people_count++] = create_person("Eve", 5);
    population[people_count++] = create_person("Frank", 6);
    population[people_count++] = create_person("Grace", 7);

    c1 = create_company("Alpha Corp", "ALP", 1);
    add_company_to_market(&market, c1);

    insert_sell_request(&market, population, people_count, "Bob", "ALP", 40, 49.0);
    insert_sell_request(&market, population, people_count, "Charlie", "ALP", 30, 49.5);
    insert_sell_request(&market, population, people_count, "Diana", "ALP", 20, 50.0);

    insert_buy_request(&market, population, people_count, "Alice", "ALP", 60, 50.0);
    insert_buy_request(&market, population, people_count, "Eve", "ALP", 25, 49.0);
    insert_buy_request(&market, population, people_count, "Frank", "ALP", 15, 48.0);

    insert_sell_request(&market, population, people_count, "Grace", "ALP", 10, 48.5);

    print_market(&market);
    return 0;
}
```

Expected Output

```
Trade executed: 40 shares of ALP at 49.00 between Bob, Alice
Trade executed: 20 shares of ALP at 49.50 between Charlie, Alice
Trade executed: 10 shares of ALP at 48.50 between Grace, Eve
Company: Alpha Corp (ALP)
    Buy Requests:
        Eve wants 15 @ 49.00
        Frank wants 15 @ 48.00
    Sell Requests:
        Charlie sells 10 @ 49.50
        Diana sells 20 @ 50.00
```

5 Specifications

- **All person names and company abbreviations will be unique.** You can use them as lookup keys during matching.
- **A person can submit multiple buy or sell requests.** Each request is treated independently, and matching is based solely on the price and order of requests in the queue.
- **Buy requests must be matched with the lowest available sell prices**, and sell requests must be matched with the highest available buy prices, following the sorting rules described.
- **Matching is price-first; differences in price are ignored beyond whether a match condition is satisfied.** For example, if a buy offer at 50 is matched with a sell offer at 48, the trade is executed and the difference is not tracked.
- **Partial matching is allowed.** If a request cannot be fully matched (due to insufficient available offers), it may still be partially executed. Any remaining amount should remain in the queue in correct sorted position.
- **No budgets or portfolios are maintained.** You are not expected to check whether a seller owns the shares or whether a buyer can afford the trade. Focus only on matching logic and queue management.

- **Floating point numbers must be printed with two-digit precision.** Use formatting for price values.
- **There must be no extra whitespaces at the end of lines in your outputs.** Each printed line (e.g., trades or queue listings) should end with a newline character, but not with additional spaces.

6 Regulations

- **Programming Language:** C
- **Libraries and Language Elements:**
  You should not use any library other than *"stdio.h"*, *"stdlib.h"*, *"string.h"*. You can use conditional clauses (switch/if/else if/else), loops (for/while), allocation methods (malloc, calloc, realloc). **You can NOT use any further elements beyond that (this is for students who repeat the course).** You can define your own helper functions.
- **Compiling and running:**
  DO NOT FORGET! YOU WILL USE ANSI-C STANDARTS. You should be able to compile your codes and run your program with given **Makefile**:

```
>_ make stock_market
>_ ./stock_market
```

- **Submission:**
  You will use CengClass system for the homework just like Lab Exams. You can use the system as an editor or work locally and upload the source files. Late submission IS NOT allowed, it is not possible to extend the deadline and **please do not ask for any deadline extensions**.
- **Evaluation:** Your codes will be evaluated based on several input files including, but not limited to the test cases given to you as an example. You can check your grade with sample test cases via CengClass system but do not forget it is not your final grade. Your output must give the exact output of the expected outputs. It is your responsibility to check the correctness of the output with the invisible characters. Otherwise, you can not get a grade from that case. If your program gives correct outputs for all cases, you will get 100 points.
- **Cheating: We have zero-tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations and will get 0. Sharing code between each other or using third party code is strictly forbidden. Even if you take a "part" of the code from somewhere/somebody else - this is also cheating. Please be aware that there are "very advanced tools" that detect if two codes are similar. **Be mindful that the use of automated assistance or code-generation tools does not exempt you from responsibility**. So please do not think you can get away with by changing a code obtained from another source.