OOP MIDTERM REPORT – ADVISOR BOT

1. **An introduction describing the purpose of your advisorbot**

The purpose of my version of the advisor bot is similar to the requirements: to help the user to analyze the cryptocurrency exchange data and emulate an exchange that should have occurred. I decided not to repeat the "merkelrex" and try to learn and practice myself, making this application a little different.

First major difference: I decided to separate time into two parts – days and timestamps. In my version, all orders have a lifetime of one day, so they can be matched with orders from different timestamps inside the same day.

Second major difference: I decided to use the hierarchical data model to store and access data. The hierarchy looks like this: Order book > Dates > Timestamps > Products > Order type groups > Orders. This way, I can access and analyze specific data without the need to parse all orders. Also, I can store some statistical data in group objects, and save a lot of calculation time. And, somehow, I ended up with 3 additional commands for my application.

2. **A table reporting which commands you were able to implement and which you were not able to implement**

Table 1 – Command implementation results

Command	Result	Notes
C1) help	Implemented	
C2) help <cmd>	Implemented	
C3) prod	Implemented	
C4) min	Implemented	
C5) max	Implemented	
C6) avg	Implemented	I decided to use not a simple arithmetic average, but a weighted average. So if there are X orders with price 2 and Y orders with price 10, then average will be Avg = (X *2 + Y * 10) / (X + Y). Also, it should be noted that my implementation clears out orders that have been matched. And all calculations are based on active orders. So to help a user better analyze statistical data, this command also prints information about average sale results

		for the requested product, if there are some (try to loop application one time and check period that hold "sales" type orders).
C7) predict	Implemented	I decided to use the weighted moving average, which uses weighted averages modified by the weight, that signifies the relativity of the data in terms of time. Also, for this function – the <steps> argument is continuous (so the first step is Date 1, Period 1, and so on), and the 'predict' command can collect data across different dates and timestamps if argument is large enough.
C8) time	Implemented	Prints both date and timestamp.
C9) step	Implemented	Match all orders in the current period, before moving to the next one. Find matches across all timestamps of the current date (even previous ones), if several orders have the same price the algorithm uses the FIFO principle and matches old ones first.
C10) plot	Implemented	Custom command 1
C11) struct	Implemented	Custom command 2
C12) dates	Implemented	Custom command 3

3. **A description of your command parsing code. In particular, you should explain how you validated user input and how you converted user inputs to appropriate data types to execute the commands**

Each program cycle starts from the small system message and user prompt. After the user enters a command, the application tokenizes user input and checks it using the 'processUserInput' function. This command checks the size of the tokenized cmd line and reroutes the execution next to the appropriate handler function (based on the number of lines), and if the user passed an empty line, or line too long > respond with an error message. All valid commands can be categorized into 4 categories: single token commands ('help', 'prod', 'time', etc), commands with 1 argument (2 tokens total - 'cmd <cmd_name>'), commands with 2 arguments (3 tokens - 'min' and 'max'), and commands with 3 arguments (4 tokens - 'avg', 'predict', 'plot'. Each handler functions check that the initial command is valid, and fires it with arguments in the form of the string. Each appropriate function that uses arguments converts (when needed) and checks arguments themselves. When I try to convert data

types I mostly use standard functions like 'stoi', 'stoul', or 'stod' inside the try block, to catch potential program crashes and switch them with simple error messages. When I try to check for the requested data I use custom functions that return boolean values inside the 'if - else' statements.

4. **A description of your custom command and how you implemented it.**

I implemented 3 custom commands: 'plot', 'struct', and 'dates'.

- 'dates' is the simplest ones. It checks the order book map for all key values (dates) and prints them to the user, similar to the 'prod' command.
- 'struct' is also pretty simple. It moves across each hierarchy level and prints the appropriate data. So for each date, command prints each timestamp, for each timestamp print every product, etc, following nested structure. This helps to show all contents in the more logical form.
- 'plot' command is the hardest one. This command prints data for the requested product collected across selected number of steps from selected date. It was my take on data visualization and market charts. I read about different kinds of graphs and decided to use a market depth chart, that visualizes buy and sell orders. First of all, this function collects vectors of addresses to the applicable product order types - bids and asks. Based on this data we can prepare the x-axis ('getXInfo' function) - find minimal and maximal price values, that will be displayed, and calculate the price difference (step) between each column. After that, we prepare buckets (vectors of doubles) that hold the volume of all orders which price falls into the bucket range (i.e. first double > amount of all applicable orders that fall into a range between the minimal price and minimal price plus column step price). We use 'allPagesToBuckets' function for ask and bid orders to get this information. Next, the program prepares the y-axis ('getYInfo' function) - same as the x-axis, but based on the amount values stored in the buckets. When we get all data and information for both axes we fire 'plotData' function to prepare data visualization in the form of the string. Iterating over each row, adding axes data, and then iterating over every column to visualize based on the data in each cell. The last step > just print data and simple information about the graph.

5. **A description of how you optimized the exchange code**

Because I decided to change the life of the order to a single day I had to use a different exchange method. In my application order is applicable for the exchange the whole day, but to optimize the calculations I used the fact that my book is structured and every structure unit holds necessary metadata stored in the private variable.

For the first period of the date: we only compare current asks and bids. And for the next period there is no need to check ask and bids from the first period between themselves again. So I prepare two groups of pointers to the applicable order groups and find the highest bid and lowest ask across all timestamps (but for that we don't iterate inside the group, but check the metadata - 'minPrice' and 'maxPrice' for each order type group). Min ask and max bid are then checked between themselves (similar to 'merkelrex'). If prices match > we start checking the amount to decide the result (fully executed orders leave order book, while program updates metadata). If we erased one of the orders (min ask or max bid) we find the next 'min ask' or 'max bid' and continue the cycle. We repeat the process until there is a price match. So if the price of the highest bid is less than the price of the smallest ask > there is no point to continue with the rest of the orders and we stop matching. This way, we never make any unnecessary comparisons. And to find the next 'max bid' or 'min ask' across several timestamps I compare only metadata values (that saves a lot of time).