CS/240/Project/2

Project 2: The "inside quote"

In the last project, we built an order book to track all the outstanding orders in the market. To make buying or selling decisions, trading programs need know what the "inside quote" is at any time. The "inside quote" is the combination of the lowest sell price and the highest buy price at any given moment. In this project, you will take your order book code and modify it to always track the "inside quote".

Input

Your program will check the command line arguments for a `-i input_filename` flag. If it is present, your program will read the input stream from the file named there. If it is not, your program will read the input from standard input.

Stream format

As before, your program will read in input as ASCII text, of which the format is as follows:

- `A <id> <side> <symbol> <quantity> <price>` - A new order is added with the supplied details. Side is either B for buy or S for sell
- `X <id> <symbol>` - An order is cancelled
- `T <id> <symbol> <quantity>` - An order is executed for the given quantity (remove this quantity from the existing order in your records). If an order is excuted for the full amount (the resulting order quantity is 0), then remove the order from the order book.
- `C <id> <symbol> <quantity>` - An order is (partially) cancelled for the given quantity (remove this quantity from the existing order in your records)
- `R <id> <symbol> <quantity> <price>` - An order is changed to have the given price and quantity

For example:

```
A 344532111 S SPY 300 117.880000
R 344532111 SPY 300 117.840000
T 344532111 SPY 100
C 344532111 SPY 100
A 344533172 B SPY 200 117.110000
A 344533348 B SPY 280 118.050000
X 344533348 SPY
T 344533172 SPY 200
```

This would be a new order to sell 300 shares of the stock SPY at $117.88. It is followed by a message to change the previous order to sell at a price of $117.84. The next two messages indicate that the order are partially executed and cancelled by 100 respectively. The next message is a new order to buy 200 shares of

the stock SPY at $117.11. The next two messages add a buy order and then cancel it. The last messages executes an order for the full amount, so it should be removed from the order book. If this was the entire message stream, your program would print out the following output:

```
344532111 S 100 117.840000
```

In the data stream for this project we will have more than one symbol included. Your program will check the command line arguments for a `-s symbol` flag. When it is present, your program will only process orders related to the symbol listed afterwards (symbol names are never more than 4 letters, and always capitalized). If it is not present, then your program should state the correct command-line format, and then exit.

The data stream as the traders read it in is not actually in ASCII format. Your program will check the command line arguments for a `-b` flag. If it is present, your program will read the input stream in binary format.

Reading in binary can be done with the fread() function.

```
size_t fread ( void * ptr, size_t size, size_t count, FILE * stream );

Read block of data from stream
Reads an array of count elements, each one with a size of size bytes, from the
stream and stores them in the block of memory specified by ptr.
The position indicator of the stream is advanced by the total amount of bytes read.
The total amount of bytes read if successful is (size * count).
```

Thus if we wanted to read in a `float` and an `int`, we could use the following code in our program:

```
struct mystruct {
  float f;
  int i;
} __attribute__ ((__packed__));

FILE *input;
input = fopen("myfile.bin", "r");
struct mystruct s;
fread(&s,sizeof(struct mystruct),1,input);
```

For this project, the binary sizes are as follows:

- `<message type>` - `char`
- `<id>` - `unsigned int`
- `<side>` - `char`
- `<symbol>` - `char[4]`
- `<quantity>` - `unsigned int`
- `<price>` - `double`

The message patterns are the same in the binary file they are in the ASCII format, except that all values come directly after one another. There are no spaces, newlines, or other formatting like in the ASCII file.

Data structure alignment and `__attribute__ ((__packed__))`

Care must be taken when read and writing binary data. Compilers don't necessarily pack data as tightly as possible. Let's take a look:

```
struct mystruct {
  char c;
  int i;
};

printf("%d\n", (int)sizeof(struct mystruct));
```

What does this print? On many modern systems it will print 8 and not 5. Why? When most modern computers read or write a memory address, they do it in word sized chunks (e.g. 4 byte chunks on a 32-bit system). To enable efficient access, data is stored at an offset equal to a multiple of the word size. To enable this alignment it can be necessary to insert meaningless bytes between variables or at the end of the last data structure and the start of the next. These bytes are known as data structure padding.
Most C compilers will let you control this. For gcc, the keywords `__attribute__ ((__packed__))` after a struct definition will ensure that the minimum required memory be used to represent the type. So:

```
struct mystruct {
  char c;
  int i;
} __attribute__ ((__packed__));

printf("%d\n", (int)sizeof(struct mystruct));
```

will report 5 bytes. Keep this in mind since we do not want padding in our binary file. Padding bytes in the binary file would defeat compact representation that binary formats allow. The binary files we supply you do not have any padding.

Internal storage

For this project, the only data structure you will use to store the order book is a hash map. You should be able to reuse your code from the previous project.

In order to keep track of the lowest sell price and the highest buy price at any time, you need to track all buy and sell orders in two linked lists, and the orders in each list should be sorted based on the price. The sorted lists should be maintained at all times, which means, whenever there is a new order added or a price adjustment, the list should be adjusted accordingly. As such, the lowest/highest sell/buy order will always locate at one end of the list whenever we need it.

Remember though that speed and memory are critical to high-frequency trading. We cannot afford to keep two copies of the data. Both the buy and the sell linked lists should be made of nodes within your hash map. This should only require a small modification to your node structure. Your the node structure should allow the memory location of a node to be inserted not only in the hash map but also into the sorted list at the same time.

Output format

For this project you will output two files. The first should be named "ob.txt" and contain a printout of the order book as it exists at the end of the hour's trading data. The format will be the same as in the previous project.

```
176986 B 24 168.090000
108976 B 39 15.230000
18740 S 2 197.230000
12780 B 52 1098.120000
:
```

For each order in the hash map, print out the orderID, side, quantity, and price. Place each order on a separate line, and use a space between values. We will leave out the symbol since all our data is for the same stock. Again, the order of the printed entries doesn't matter.

The second file will contain a series of inside quotes. This file will be named "iq.txt". After every `n` messages are processed for the selected symbol, your program should print the current inside quote to this file. The default value for `n` should be 1000. Your program will check the command line arguments for a `-n message_count` flag. If it is present, your program will use the value `message_count` for `n`. The inside quote is just the lowest sell price followed by one space and the highest buy price. Each inside quote will be on its own line. It should look as follows:

```
168.120000 168.090000
168.150000 168.110000
168.140000 168.130000
:
```

Makefile
Create a Makefile. Target `all` builds `order_book` with whatever files you need.

Turning in

For your submission, pack your files as follows:

```
tar zcf turnin.tgz Makefile source1.c source2.c ...
```

The project is due Monday, April 2nd before midnight. No late projects accepted.

Grading criteria

- Makefiles correctly builds target `all` that builds `inside_quotes`
- code compiles and runs without error
- program can handle input from either stdin or an input file
- program can handle input in either ASCII format or binary format
- program outputs a correct order book to a file named `ob.txt` before termination
- program outputs correct inside quotes to a file named `iq.txt` for every `n` messages (where the default is 1000, but `n` can be provided in on the command line)
- program stores the order book in a hash map
- program does not store use additional memory allocations to store the buy and sell lists

Please note that the autograder cannot fully detect all requirements. Thus your autograder grade is not final. All projects will be inspected by the TAs to determine your final grade.