CS/240/Project/1

Over the course of four projects, you will explore how C can be used to make the data structures and algorithms needed involved in high-frequency trading. No understanding of stocks is needed.

High-frequency Trading
In high-frequency trading (HFT), programs analyze market data to find and take advantage of trading opportunities that often only exist for a few seconds. The programs in HFT buy and sell equities, options, futures, ETFs, currencies, and other financial instruments quickly to generate very small consistent profits from sub-second trends seen in the market data. In 2010, HFT accounted for over 70% of equity trades in US markets. As a field, HFT operates largely on having the fastest network, algorithms, operating system, programs, and coders.

Project 1: Hash maps and High-Frequency Trading

Programs involved in automated trading have to process a lot of data extremely quickly. Messages describing trade offers can reach rates of up to 250,000 messages/second. For the first project, we will create a program which will read in and process a stream of order messages for a single stock. These messages will be used to update our program's view of the market state. This state is known as the "order book". Each order consists of an order id, whether the order is to buy or sell, the stock symbol, the quantity, and the price. The symbol can be up to 4 characters long. The messages we process will either trigger the entry of a new order, change an existing order, or deletion of an order. At the end of the stream, your program will print out all the current orders in the order book.

The Order Book
The "order book" is the current state of all pending buy and sell offers for a given electronically traded financial instrument. Buy maintaining an order book, a program can see the "depth" of the market, since it can see the number of contracts that are available at each price. By analyzing the order book, a HFT program can make decisions when to buy or sell.

Input/Ouput

You 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. Likewise, your program will check the command line arguments for a `-o output_filename` flag. If it is present, your program will write the output stream from the file named there. If it is not, your program will write the input to standard output.

Stream format

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 (partially) executed for the given quantity (remove this quantity from the existing order in your records)
- `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
```

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 last two messages add a buy order and then cancel it. If this was the entire message stream, your program would print out the following output:

```
344532111 S 100 117.840000
344533172 B 200 117.110000
```

Internal storage

For this project we will experiment with two different data structures to hold the orders. By default, your program will store the order data in a *linked list*. If your program is given the command line argument `-h` then it will store the order data in a *hash map*. You should be familiar with linked lists from previous labs we will only discuss hash maps here.

Also known as hash tables, a hash map is a data structure used for quickly accessing stored values (something we need to remove or modify an order). In a linked list, we must look at each item to find what we are looking for. This can be slow when the list is long. An array, on the other had, provides direct access, we use an index number to retrieve a data value (e.g. array[5]). In fact, a standard array can be considered the simplest hash table. There are two problems with this. orderID's may not be sequential (there may an orderID 18462 and an orderID 308675, with no orderID's in between), and we don't know how many orders there will be. Using the orderID as the index into an array would thus be wasteful, and dangerous (the orderID could be larger than the array size).

To get direct access we will still use an array. Instead of using the orderID as the index, we will give the orderID as an argument to a hash function. The hash function will return a index in the array where we can add, store, or modify the order data. This solves the problem of wasted space.

This still leaves the problem having more orderID's than array locations. Whenever a hash function returns an index that is already being used by another order, this is called a "collision". There are many ways hash maps can handle this. In this project, we will use "chaining" to solve this. Each entry in the array will be a pointer to the head of a linked list. When you need to add an order to an array position that already stores an order, you will insert the new order at the head of the list. When you need to remove or

modify an order, you will need to first find it's array position with the hash function, and then search the linked list at that location for the correct order.

Hash functions and map sizes
Two things are critical to an efficient hash map, a good hash function, and a correct array size.
If the hash function is poor, then the data will not be evenly stored across the hash map. In the worst case, imagine a hash function that always returned the value 0, no matter what order id was provided. This would result in a hash map that was really just a single linked list. A good hash function should make sure that the data is evenly stored across the entire map (array).
If the hash map size is too small, then we again run into similar problems. If we used an array of size two, then our entire hash map would consist of two linked lists. While each order lookup would only take half the time of one linked list, this is still far less efficient that what we would prefer.
For this project, we will assume that the order id's are evenly distributed numbers. As such, our hash function can use a simple modulus function (hash index = order id % hash table size). This leaves the question of the best hash table size to use. Set your hash table size with a `#define` macro. Surround the `#define` with the conditional group `#ifndef` (http://gcc.gnu.org/onlinedocs/cpp/Ifdef.html#Ifdef). This will allow you to quickly and easily try your program using different hash table sizes by simply adding the `-D name=definition` compilation flag and recompiling (http://gcc.gnu.org/onlinedocs/cpp/Invocation.html#Invocation). Using the `time` command, you can time the speed of your program as a you try different hash table sizes. For small data sets, this won't make much of a difference, but for large dataset, it can be critical.

Note, your program must implement the both ways of internal storage. We will look in your source code for the correct implementation.

Output format

When the data stream has been completely processed, you have reached the end of the hour's trades. You must printout the contents of the order book. 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. Your output should look similar to the following. The order of the printed entries doesn't matter.

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

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 Mon., Mar. 19th before midnight. No late projects accepted.

Grading criteria

- Makefiles correctly builds target `all` that builds `order_book`
- code compiles and runs without error
- program can handle input from either stdin or an input file
- program can output to either stdout or an output file
- program can store the order book in a linked list or hash map, as determined by command-line flags
- program correctly outputs the order book data

Please note that the autograder cannot fully detect whether you use a hash map or a linked list. Thus your autograder grade is not final. All projects will be inspected by the graders to determine your final grade.

Working on larger programs
Starting larger programs with lots of requirements can be daunting. Start by breaking things down into smaller testable units. For this project, a simple order could be as follows:

- Start with a program that will correctly parse the input messages from an ASCII file
- Modify the program to add, remove, and modify orders stored in a linked-list
- Your program should now have the correct behavior when no command-line flags are used. Test!
- Add parsing of command-line flags. Start with the flags to accept input and output target files
- Add the ability to detect the `-h` flag, and begin adding in the hash map data structure
- Test your hash map implementation

As separate tasks, each of these problems end up being straight-forward.