CS/240/Project/1

Over the course of four projects, you will explore how C can be used to construct data structures and implement algorithms involved in high-frequency trading. No understanding of trading algorithms 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; that number has increased since then. As a field, HFT succeeds because it uses 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 this 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. By 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 in order 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 since we use an index number to directly 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 indexing into an arrays using the orderId, however. First, 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 an index in the array where we can add, store, or modify the order data. This solves the problem of wasted space.

The second problem is that there may be 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.
First, 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).
Second, 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 both representations of internal storage (linked lists and hash tables). We will look in your source code to make sure both implementations exist. If the flag -dh is passed to your program then the hash should use the function `default_hash` as default value for your hash structure.

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
:
```

Implementation

In this section we will give you instructions to build modules to get project1 completed.

The project will have three main parts:

- `order_book.c`
- hash.c
- `list.c`

A. list

Part A is a doubly linked list implementation similar to what you have done in lab4. For convenience, we will provide a reference library with all the functionality needed to get the assignment done. **If your submission misses linked list implementation of yours then 30 points will be deducted from your grade. You should be able to submit the other parts by using the reference library.**

Specifications:
There are some functions required to be implemented (the ones listed in `list.h`). You need to make sure that you implement all these methods since it will be the way the auto grader will interface with your linked list.

- The list has nodes of `struct onode` shown in `list.h`. `onode` holds a pointer to the actual data stored in `struct order`.
- It makes perfect sense to think of a hash table as an array of linked lists. If you have an array of size 1 then you are reducing the hash is reduced to function as a single linked list.

Beside the set of the setters and getters you need to implement for the `onode` and `order` structs, there are couple of new functions to implement.

```
struct onode* newNode (struct order* data);
```

Description: Returns a new linked list node filled in with the given order, The function allocates a new order and copy the values stored in data then allocate a linked list node. If you are implementing this function make sure that you duplicate, as the original data may be modified by the calling function.

```
struct onode* getOrderNode (struct onode* head, int id);
```

Description: Given the head of a linked list head and the id of the order we are looking for. `getOrderNode` will iterate on the linked list and returns `node` having the order data with that id.

```
void printList (struct onode *node,
            void(*printItem)(struct order *, FILE *),
            FILE *out);
```

Description: Accepts the head of the linked list. `prinList` is generic in the sense that it takes the output stream as a parameter which means it can actually print to a file, `stderr`, or `stdin` based on the value of the out stream. Finally, printList can take a function pointer that specifies exactly the format of how each item should be printed to the output stream. `printList` will iterate on all the elements of the list and apply `printItem` on each node in the list.

B. Hash

Part B is to implement a hash structure. You do not need to finish part A to get this part done. You can use the reference library to start testing your hash implementation. You have to implement the functions listed in `hash.h` .
Hint: Think of the hash as an array of linked lists. A hash can also be viewed as a linked list if the size of the array is 1.

Specifications:
`hash.h` defines `struct hashStorage`. The latter is a struct that will hold all the information for a single instance of hash. To start storing orders in a hash you need to implement `createHash` which returns the new instance of the hash and sets all the fields of the hash. After the hash is successfully created, it should be ready for usage given the hash function and size you passed to `createHash`.

```
struct hashStorage* createHash(int size,
            int (*myHash)(int),
            void(*printOrder)(struct order *, FILE *));
```

Description: Create a new instance of `struct hashStorage` and returns it. It sets the size of the table to be of length "size" which will be the number of the entries in the hash. It takes also an argument to specify the format of the order printed to an output stream. If myHash parameter is `NULL` then this means that the hash should be a linked list. When `myHash` is `NULL` the `size` parameter should be ignored. It returns the new `onode`.

```
struct onode* addOrder(struct hashStorage* hash, struct order* data);
```

Description: Add an order to the hash structure. Remember that you should copy the data before adding it to the hash as data can be modified (hint: look at `newNode` in list)

```
void cancelOrder(struct hashStorage* hash, struct order* data);
```

Description: Cancel an order from the hash. It will look for the `onode` having order with the same `data->id` and remove it from the list, then destroy its allocation.

```
void reduceOrderQty(struct hashStorage* hash, struct order* data);
```

Description: Reduce the quantity of an order by the amount stored in the `data->quantity`. It will look for the `onode` having the same order id as `data->id` then subtract the `data->quantity` from `node->data->quantity`. If `(node->data->quantity==0)` the node should be removed from the hash and its memory allocation should be destroyed.

```
void changeOrder(struct hashStorage* hash, struct order* data);
```

Description: Change the quantity and the price of an order. It will look for the `onode` having the same order id as `data->id` then set the `(node->data->quantity=data->quantity)` and `(node->data->price=data->quantity) node->quantity`. If `(node->data->quantity==0)` the node should be removed from the hash and its memory allocation should be destroyed.

```
void freeOrderBook(struct hashStorage** hash);
```

Description: Delete the order book and free all the memory allocated for that hash.

```
void printOrderBook(struct hashStorage* hash, FILE *out);
```

Description: Given the `hash` and the output stream, print the order book. The function should print each item given the function `hash->printItem` and the output stream required for the print.

```
int getHashSize(struct hashStorage* hash);
```

Description: returns the size of the hash table

```
struct onode** getHashTable(struct hashStorage* hash);
```

Description: returns the table of the hash.

C. order book

This file is the only file that has main. It parses the argument and set the default values if any. The order book then should read the input and store all the orders in an internal storage. If the parameter `-h` was present then the internal storage has to be a hash and you have to pass a default hashing function to the hash during the creation.
If the parameter `-dh` was present then your internal storage has to be a hash and you have to pass `default_hash` as a parameter to your hash. You do not have to implement `default_hash`. Instead, it will be provided in a reference library. Feel free to implement the default hash function as you like. We will test to make sure that you are actually using a hash function that does not generate a single linked list when passing `-h` flag. So, make sure that you do it correctly and you are not hashing all the orders to a single linked list.

As specified in previous sections, use the `#ifndef` macro to set a default value for the hash size. `order_book.c` should be the place to set that default value. After parsing the input stream, the order book should be printed using a call to `printOrderBook` then finally the data should be deleted `freeOrderBook` before terminating the main function.

Turning in

For submissions, you can submit `hash.c` and `list.c` to get 30% of the grade.
to submit the rest of the assignment you can pack your files as follows:

```
tar zcf turnin.tgz list.c hash.c order_book.c
```

If you are using the reference library and you are not submitting list.c then pack your files as follows:

```
tar zcf turnin.tgz hash.c order_book.c
```

Make sure that you decide whether you are submitting list.c or not. You need to make sure that you pick the way that gives you the highest grade (in case your list.c is buggy)

The project is due Wed., Oct. 31st before midnight. No late projects accepted.

Grading criteria

- a list.c and hash.c that compiles and has all the required functionality `(30% of the grade)`

- program can handle all arguments mentioned in previous sections `-i -o -h -dh`
- 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
- correct hash implementation that can be used to handle all the order operations mentioned `A, X, T, C and R`.

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 data arguments
- Add to a program that will correctly parse the input messages from an ASCII file
- use the reference list library to start implementing a dummy hash implementation that encapsulates a single linked list.
- run a program that will read some input and then print them and use the printOrder to make sure that you get the correct output.
- when you run the input make sure that you can do it in steps. first, focus on adding the orders correctly. After you make sure that you can add the orders correctly, move to the next step of cancellation, updates..etc
- After you have the feeling that your code works well for a dummy hash. Start, fixing and making sure that the code works when you make the complete hash solution.
- Test your hash map implementation
- Try to test different hash size and see the effect of that on the performance.

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