CSE250, Spring 2013 **Project 2 (Assgt. 9)** **Due 4/22 & 4/29**

Stock Market News Service Simulation

Groups of 2, `online submission only---no hardcopy`

This project has a "staggered due dates" policy, also called a "checkpoint" policy, to combat procrastination. A full draft of the main data-structure components are due **Monday, April. 22, 11:59pm**; labs that week will help debug and interface timing code using the `HiResTimer` class. The components are a hash-table plus hash function from one team member; a heap-rebuilder, `Stock` class, and comparison function objects from the other. The full application client, timing experiments, and final project report are due **Monday, April. 29, 11:59pm**.

Reading

For Wednesday, read this project description, including parts of the text *and* online sources it references.

Short Task Statement

Your software is to provide speedy recording of stock market transactions as they come over a simulated *stock ticker*, and speedy reporting of certain market statistics requested by subscribers to your service. The requests are for "Top-k" lists in certain categories. Both the ticker's stock-trades and the requests are simulated by lines in a text file.

Your project must store class objects for the stocks in a templated STL-conformant hash table, hashed according to their *ticker symbol*, and pointers or "proxy objects" for the stock objects in templated heap containers corresponding to the categories. For initial insertion into the hash table, the data files begin with lines such as

```
add? NCF 5000000s23.875
add? AAM 6203000s2.0
add? AIR 27181000s15.8125
add? RNT 19909000s13.9375
```

The `add?` is to help you process this as `insert` not `find`. Then follows the ticker symbol between spaces. Then comes a relatively large integer giving the total number of shares that exist (called the *capitalization*), the letter `s` for "shares' as a divider, and finally a floating-point number representing a starting dollar price per share. *Trades* have the same format without the `add?` token, e.g.:

```
AIR 100s15.7625
IHI 1400s26.262499
SRR 100s8.0625
KEG 400s9.2375
```

means 100 shares of AIR were traded at (about) \$15.76, then 1400 shares of another stock IHI were traded at (about) \$26.26, next 100 shares of SRR and 400 shares of KEG at the prices shown. The *requests* you must provide for are the following—the final number "k" is arbitrary:

```
printTopByVolume? 30
printTopByPercentUp? 10
printTopByPercentDown? 50
printTopByPercentChange? 100
printTopByMomentum? 50
printTopByTrendTrades? 50
```

The *volume* is the total number of shares that have been traded thus far. Percent up and down are based on the starting price when the initial inserts were done. Percentage down should have a

negative sign, but percent-change takes its absolute value (use `std::fabs`) to compare with cases of percent-up. The *momentum* is defined by

$$m = (\text{percentage change})\frac{\text{volume}}{\text{capitalization}}.$$

Finally, the *trend* is the latest sequence of trades during which the stock has either been rising or falling. A rising trend is defined to begin with a trade in which the price rose, but is not stopped by trades at the same price thereafter—it is stopped only by a trade in which the price falls. A falling trend is defined simlarly.[1] The query `printTopByTrendTrades? 50` asks to print the 50 stocks with the largest number of trades in their current trend, whether rising or falling. *For extra credit*, you may implement the related query `printTopByTrendShares?`, which adds up the volume of shares over the trades in the trend, rather than just count the trades—the extra credit being specifically for maintaining this in $O(1)$ time. Together with momentum, the trend information is most sought after in stock market "technical analysis."

Finally, there is the query

`printAll?`

which calls for printing out the current (or closing) prices of all the stocks *sorted by ticker symbol*. Note that iterating through the hash table will first give you a haphazard "hash order." To sort them, the team member in charge of the heap must either use it to sort (this is HeapSort), or alternatively may re-use the `Valli` class from Project 1 ("ValliSort"!). In addition you can/should implement some commands along lines of Assignment 8, problem (4) to help you debug, and on the final part there will be `pause?` to help set timing code (`CentiTimer.h`, optionally `HiResTimer.h` on `timberlake` only). This completes the required functionality.

How It Works

You will have one hash table, various heap objects organized according to the corresponding category, function objects for the hash function and the category comparisons, and a `Stock` class. You may define and use additional classes, such as a small "proxy" for the `Stock` class which the heaps can store without needing to swap entire `Stock` objects. The proxy will (be or) store a pointer into the hash table's record for the stock, so that the current price and volume and other comparison information can be read on-the-fly. Thus the order of events is:

- To process a stock trade, look up the stock in the hash table and update its info. The heaps need not be involved (i.e., you need not "fix them up" yet).
- To process a "top-k" query, re-make the corresponding heap, and extract the top k elements—putting them back into the heaps after you've printed them.
- To print-all, iterate through the hash table to pull off the stock objects into a `vector` or etc., sort them, and iterate again to print them in sorted order by ticker symbol.

The most delicate part is getting the stock-comparison function objects to work with the proxies in the heaps, not just the `Stock` objects by-value. The proxies *can* just be `Stock*` pointers, so compared to the `Phrase` function objects, you would have a pointer-dereference of variables such as "`lhs`" and "`rhs`" in the body. But since raw pointers can be a debugging hazard, you may choose to make a proxy object that embeds and manages such a pointer. *Note that storing the* `Stock`

[1] With reference to the similar concept of phrases with ascending/descending word lengths, a falling trend does *not* overlap the end of the previous rising trend.

objects themselves in the heap(s) is impractical, because heaps do not offer $O(1)$ or even $O(\log n)$ time lookup.

Rules and Grading

One person is "Hasher," the other "Heaper." The Hasher must initially submit by 4/22 a file `ChainedHashMMM.h` that implements a hash table *with chaining*, and with the same templated interface as on Project 1. Note especially:

> The Hasher may not obtain a hash table from the Internet or any other third-party source—the regulation proof of originality is that this file has recognizably been "morphed" from one of the team's `Valli` files, with the hash-buckets chained together into one list. Similar remarks apply for the Heaper—in particular, you must write your own `make_heap` function based on code shown in lecture.

As tentative shortcuts before the 4/22 first due date:

- `erase` need not be coded—this will be permanent.
- the integer argument for the constructor `explicit ChainedHash(size_t sz)` can be interpreted as a predetermined size, rather than a load-factor limit—with no `refresh/rehash` needed, and
- the "hole" with the iterator `operator*` will be left unfixed, even though it applies equally to a hash table as to a sorted data structure.

The Heaper is responsible for `HeapNNN.h`, `StockNNN.h`, and various function-object classes (which need not have initials, but need both teammates' names inside). The `Heap` class must be templated and have a function-object template parameter, so that the single `Heap` class can be re-used for each statistical category. The one restriction is that the `Heap` class may not simply delegate to the STL's `priority_queue` class, but must instead have a `vector` field on which the make-heap algorithm from lecture is applied. These components are likewise due 4/22, 11:59pm.

Team members may of course collaborate on design decisions (method names and prototypes, etc.), but the coding effort must be substantially as indicated. The client file, `StockClientMMMNNN.cpp` with both sets of initials, and with answers to report-questions (to be specified) as a comment at the bottom, may be an unrestricted joint effort. That, along with any needed fixes/revisions to components, is due Monday 4/29, 11:59pm.

Points on Project 2 Components:

- *Hasher:* 120 for the hash table, 15 for appropriate logic comments, and 15 for coding the hash function as a function object.
- *Heaper:* 60 for the heap class, 60 for the `Stock` class, 30 for the function objects.
- *Each gets 50 pts. credit for the other person's work*—nominally this is for collaboration on the overall design.

This makes 200 points credit to each for components. The client file will be 30 pts. for each, and there will be 70 pts. divided among timing code to be specified, report questions, and a possible pop-quiz for reading this assignment. Here is one definite report question—as before, answers are expected to go in comments at the bottom of your joint client-file.

(1A) ($8 \times 3 = 24$ pts. individual credit) *For "Hasher":*

(a) Does your hash table intend to store values (`I data;` in the bucket nodes), or pointers (`I* data;`)? Or possibly "proxy objects" that would have a pointer as a field?

(b) How did you initialize the hash table to a specified size `sz`? Did you allocate up-front `sz`-many nodes, one per bucket?

(c) Did you employ dummy nodes? Did they have a Boolean field marking them dummies, or did you rely on identifying the dummy default-constructed object `I()` by `itemMatch` to a stored dummy (similar to the text's handling of "DELETED")? Or if you used `I*` pointers, was having `data == NULL` your way of testing-for and skipping over dummies?

(d) Did you try to "recycle" a dummy node by assigning the first item hashed into its bucket to it? Or did dummies stay dummies throughout?

(e) Was your iterator ever allowed to stop on a dummy node? How did you handle the node (if any) for `end()`?

(f) How did you test items for equality, especially in the `find` method? Did your hash table accept a general function object for matching, or did it explicitly rely on `operator==` for the `string` class? (Ultimately you applied `operator==` to the stocks' ticker symbols somewhere—the question is where?)

(g) Did you add any extra functionality to the `iterator` class compared to `Valli`? (Such as coding `operator->` to return a pointer to the current cell's `data` field, which would enable the client to store `itr.operator->()` rather than do `&(*itr)` to get a pointer to the `data` field.)

(h) Did any `CLASS INVs` (for any of your `ChainedHash`, `DNode`, or `iterator` classes) help make your code simpler and/or faster?

(1B) *For "Heaper":*

(a) Does your heap class store pointers, or proxy objects?

(b) Does it `#include` your partner's hash-table class file in order to `find` statistical information via the ticker symbol, or does it fetch the information more directly?

(c) Was your heap always a heap at any point in time?

(d) Did your function-objects take arguments of type `const Stock&`, `const Stock*`, or something else?

(e) Did your `Stock` class `friend` the various function objects, or did the function objects use only public (getter) methods of your `Stock` class?

(f) How did your `Stock` class process a transaction? Does `Stock` itself try to parse a string transction line into a number-of-shares and price, or assume the client will do it?

(g) Does your `Stock` class store all previous transactions in order to figure out the number of trades in the current upward or downward trend? Or does it maintain fields to do so?

(h) Did you consider (allowing "Hasher" to talk you into) computing the hash code of the ticker symbol at construction time and storing it in a `hashCode` field, rather than recomputing the hash function every time it is accessed? (Then the actual passed-in hash-function object would just be a getter for this field.) Is this an application where you could get away with that?