CSE250, Fall 2009 Project 2 Timing Tests and Report Due Fri. 12/4, 11:59pm

`online submission only---no hardcopy`

Short Task Statement

Complete the client sketched in the file `ParsingCodeBlocks.txt` in the ∼`regan/cse250/PROJECTS/PROJ2/` directory, calling it `StockClientMMMNNN.cpp` where `MMM` and `NNN` are your initials (Hasher then Heaper). It is mainly an event loop being simulated by reading lines from a file that has both data and commands, the latter distinguished by question marks: `add?`, `pause?`, `printTopByVolume?` k where k is an integer... Among the data files in that directory, the featured one for the report's timing runs is `Main100k-25.data`.

Implement `pause?` to stop and read a timer that has been timing previous commands. Use this to conduct timing runs of three competing implementations for processing the stock trades: (1) your new `ChainedHash` class at sizes 1,024, 2,620, and 7,860, (2) your previous `Valli` class or one of the provided ones with ratios 8 and 3, and (3) the provided `BST` class. Also time the processing of commands to obtain top-k lists for the various statistical categories, using (a) your `Heap` class, (b) the provided `TopPQ` class, and (*for extra credit*), (c) the provided `BST` class again! The times in (a)–(c) can be the sum of getting the 6 lists at the end of the `Main100k-25.data` file (or 7 if you do the extra-credit one), i.e. you need not time each one individually. (The timing trials (1)–(3) and (a)–(c) are logically separate—i.e., a combo such as "(2a)" has no meaning apart from timing (2) and (a) separately–because the phases of doing stock trades and then compiling top-k lists are separately represented and timed in the data files.) Repeat each of the five-or-six tests 10 times, both on your home machine (one per team is fine) with `CentiTimer.h` and on `timberlake` with `HiResTimer.h`, and report the average times over the 10 trials. The files mentioned in this paragraph are in ∼`regan/cse250/PROJECTS/F09BASE/`.

Team members may work jointly on the client, and may conduct trials on `timberlake` and a home machine jointly—or if none, on a different CSE machine (e.g. `nickelback` or `metallica`). Then answer the report questions below—note that some are individual to `Hasher` and `Heaper`, while others are joint. Various extra-credit options are also specified, and marked individual/joint as-appropriate. An expanded description of the client and timing runs, with *options* for making them easier to do, follows below.

What To Submit

The only required names are those specified on the initial project description: `ChainedHashMMM.h`, `HeapNNN.h`, `StockNNN.h`, and `StockClientMMMNNN.cpp`. Putting method bodies into `StockNNN.cpp` and/or supplying a makefile `StockClient.make` is *optional*. Any other files you include–such as `StringExtra.{h,cpp}`, `BST.h`, function-object classes (though putting them in `StockNNN.h` is sensible), `Cardbox.h`, any of the `Sifter` adapters—must also be submitted. Your `StockClientMMMNNN.cpp` file must be submitted in a form that (a) uses your `ChainedHash` and `Heap` classes as implementations, (b) has no screen-interaction lines (such as `getline(cin,line);`), and (c) implements all commands listed on the original project description (including `printAll?`, plus now `typeout?`, but not necessarily including e.g. `printDS?` or `whisper?` which were options to possibly aid your debugging). Your answers to the report questions below should go at the bottom of your client file, as previously stated.

How the Client and Timers Work

The client and data files give a rudimentary text-based simulation of an event handler. The "events" are either stock-trades or commands. All commands begin with a single space-free alpha-word followed by a question mark, than a space, then an argument. The "event loop" is a while-loop that reads a single line from the data/command-file, parses it, and feeds into a big `if-else` construct with blocks that process the various commands. The required commands are those described on the initial project spec, plus we have made `typeout?`—which simply echoes the following text to the screen so you can see what stage you're on—required for the timing runs. (Find the space after the `?`, then copy the remaining string to `cout` followed by `endl`—note that `getline` does not already include the carriage-return.) Output is manually buffered into a string `result` rather than displayed after each operation, to minimize the interference caused by screen or file output in measuring processing time.

Directions for filling in the blocks to handle the `add?` command for inserting, the `printTopBy---?` k commands for calling `make_heap` and extracting a "Top-k" list—are as described in the original spec and `ParsingCodeBlocks.txt` file. It remains to describe the timer and means of handling `pause?`. The block for `pause?` should begin by reading the timer, and end by resetting/restarting the timer (if necessary). Optionally, one may pause processing so you can print and see the timing, and/or print out the accumulated `result` string (resetting it to the empty string for the next iteration).

Both the `CentiTimer` and `HiResTimer` classes have public methods `void reset()`, `double elapsedTime()`, and `double timeSinceReset()`, plus constructors (which include a final `reset`) and `string getUnits() const`. `CentiTimer` defaults to units of milliseconds, `HiResTimer` to microseconds. Since they use the same interface, a single line `typedef CentiTimer StockTimer;` atop the client file can control which one is used. As previously stated, `HiResTimer` will probably *not* compile on your home machine, and requires a final `-lrt` option to compile on `timberlake` with `g++` (or Sun `CC`, for that matter). `CentiTimer` gives only 1/100 sec. resolution on `timberlake`, but gives high resolution (after all!) on Macintosh `g++`, and may do likewise on your home machine. The `elapsedTime()` method records the time since the previous call to `elapsedTime()` or `reset()`, while `timeSinceReset` allows you to accumulate time through several `elapsedTime()` readings. This is like a stopwatch that can record both the time since the beginning of a race and the time since the last quarter-pole. However, using screen interaction makes any accumulated-time reading meaningless, so if you use something like `getline(cin,---)` in your `pause?` block, then you should end it with a call to `reset`.

The point of this organization is to minimize the editing *of code* needed to do the various tests and trials—or even simply debug. The next section describes various options.

Ways to Change Implementations for the Timing Tests

The latter two of these three ways are more "software-engineering correct," but the first is not onerous and hence is AOK.

1. Refer to your `ChainedHash` and `Heap` classes directly in the client, changing the former to `Valli` and `BST`, the latter to `TopPQ`, and recompiling for the comparison runs.

2. Use the `Cardbox` and/or `Sifter` "façade classes" to switch the implementations. This still requires re-compiling, but may involve fewer edits with easier handling than modifying the client.

3. Move the event loop from `main` into a *generic function* above `main`, of the kind exemplified in several testing clients. Then `main` itself can be short and simply call the function for the various combinations. This avoids re-compilation but requires re-reading the data-file lines.

It is obviously horrible to reduplicate the hundred-line event-loop in `main`, hence goes-without-saying forbidden. If you use option 3., a possible header is

```
template <class ARG, class CARDBOX, class SIFTER>
void processFile(CARDBOX& cont, SIFTER& heap, [other args?]) { ...
```

Note the `&` to avoid copying the containers, and they are not `const`-references because they do get modified. Depending on your compiler, you might-or-might-not have to use angle-brackets also in the call, as in [for example—you may have `Stock*` or `StockProxy` in relevant places]:

```
processFile<Stock, ChainedHash<Stock,StockHash,StockMatch>, Heap<Stock> >
          (table, heap, [other?]);
```

Or just `processFile(table,heap,...);` might work. This is only relevant for option 3., since the "Adapter" idea 2. (demo'ed in lecture) already involves this kind of generic step and lets the client see only one table-concept (and heap-concept) at a time. (Ideas 2. and 3. figure into lecture coverage of K-W chapters 9–10 anyway, and will be referenced on the last "Assignment 8," so choosing option 1. (as originally allowed) and ignoring "Cardbox/Sifter" does not circumvent them entirely.)

The report questions below apply to any of these three methods equally. Completing the client is 30 pts. to each, and the two regular-credit report questions are 49 pts. to each, bringing the total regular-credit points to each to 150 + 50 + 21 (quiz) + 30 + 49 = 300.

YMMV / Not-YMMV Notes

We believe we've narrowed the tasks and testing-specifics to give meaningful results on a variety of PC/mainframe platforms, depsite the way high CPU speeds mask issues that were felt more obviously in previous decades. [For a case in point, the provided `Sifter` class, when switched to a `BST` implementation to get the top-k lists, has to avoid using `BST` for the initial inserts. This is because it is using your statistical function objects rather than a less-than on ticker symbols for its comparisons, but initially the stats are all zeroed out, so the tree winds up doing the horrible just-like-a-singly-linked-list thing talked about in lecture. How horrible, actually? With the full slate of 7,860 stocks, what used to be a big part of an hour is now under a minute, and on the smallest data files—just a blip. Holding the items initially in a `vector` makes the initial heap-inserts almost-instantaneous. We *can* say, however, that if your runs don't finish within a few seconds even on the largest data file, there is something markedly non-optimal in your hash-table or heap design.]

The timer classes do *not* distinguish between CPU-time and overall system-process time, as the "`Stopwatch`" class used in previous years did; I have not found a portable way to do this. However, `timberlake` is multi-core enough that system overheads seem not to interfere with results, at least when it's not being strained. That said, if you get "0.000" as timing readings with `CentiTimer` on `timberlake` from your `Heap` class, even on the largest data files, it's probably not a bug! At least the provided `TopPQ` text-like implementation does *not* give zeroes on the larger files, even when compiled with highest optimization on `timberlake` via `g++ -O5` (dash-Oh-five). That said also, there doesn't seem to be much point anymore to the smaller files for timing testing—old tradeoff points are hard to spot. Hence only the one largest file is "official." Note also that upping k from 25 (or 30) to 100 or beyond might only obscure rather than enhance the difference, because it is involved only in the "$k \log(N)$" part of the running time of extracting the top-k items. The main issue, running "heapify," (STL `make_heap`, "buildHeap" in the notes from Weiss' text) depends only on "N" which is here the number of stocks, and the `allstocks` files already include all ones that were traded on US exchanges. In future we may have to widen the slate of stocks by including overseas exchanges, but the essential points do remain valid this year, even under low timing resolution albeit barely...

Report Questions

The first two are separate checklists for "Hasher" and "Heaper," and count for 24 pts. separate credit. They might influence you to change implementation choices after (or for) the Monday 11/23 initial submission, and there is nothing wrong with that... All answers go in comments at the bottom of your 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 proxies?

(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`, `Cell`, 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 re-computing 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?

(2) (Joint answer, 25 pts.) Conduct the timing trials stated at the outset, on your home machine(s) and on `timberlake`. Absolute times do not matter (so long as each is a matter of seconds at most), but be consistent with optimization—i.e. if you use the `-O5` option for one trial on `timberlake`, do so for all. The trials on each machine, each with the lone data file `Main100k-25.data`, are:

- Time to process the 100,000 stock trades with your team's `ChainedHash` class, at target sizes 1,024, 2,620, and 7,860 (load factors of almost-8, 3, and 1 respectively).
- Time to process the 100,000 stock trades with one of your `Valli` classes, or a provided one, ratios 8 and 3.
- Time to process 100,000 stock trades with the provided `BST` class—coding and passing-in a `lessThan` on the ticker symbols.
- Time to compile the "Top-25" lists with your `Heap` class.
- Time to compile the lists with the provided `TopPQ` class.
- (extra credit) Time to compile the lists by instantiating `BST` with your pointers-or-proxies, passing in your statistical function-object comparisons rather than `lessThan` on ticker symbols.

Report each result as an average of 10 trials. Then answer:

(a) Did the theoretical "$O(1)$ vs. $O(\log n)$" per-find advantage of a hash table manifest itself, compared to `Valli` and `BST`?

(b) What difference did increasing the table-size (i.e. the size of the `buckets` array, not the number of items) make? Suppose we figure that a doubling of table size is worthwhile if it saves 25% off the time—thus a 2^a increase in table-size should make the new time less than $(3/4)^a$ of the old time. Was the increase from 1,024 to 2,620 worthwhile? From 2,620 to 7,860?

(c) Did the theoretical "$O(n + k \log n) vs. O(n \log n + k)$" advantage of `make_heap` versus the priority-queue implementation come out in your results?

(d) Are your relative comparisons similar on the home machine(s) versus `timberlake`?

Extra-Credit Options

These can be joint-efforts, counting for each even if you split up the work—though if you split up work, work done individually must be close-to-balanced. Please make clear in your report section which one(s) you have attempted, if any.

(a) The BST-for-heap option in (2) above is worth 24 pts., more because it really requires using one of the provided `Sifter` classes to make the adaptation tolerable. Also answer: assuming that the trees stayed reasonably well-balanced, making "BST-Sort" $O(n \log n)$ in theory, do you see evidence of a markedly higher principal constant compared to the heaps? [Note: If you use pointers rather than proxy-objects for heaping, technically putting the pointers in the BST violates its REQ of a member-`str()` function. However, you will not get a compile

error provided you don't actually try to print out the tree via its own `str()` method. This shows incidentally that C++ uses a more laid-back type-enforcement mechanism than Java's interface-checking; C++ creator Bjarne Stroustrup even calls it "Duck Typing." This is a major issue in the recent postponement/axing of the "Concepts" type-checking feature for C++.]

(b) Implementing `printTopByTrendShares?` with a full $O(1)$-time analysis a-la Assignment 5, problem (1), is 18 pts. extra.

(c) For 24 pts. extra, implement both `void erase(iterator itr)` and `rehash(size_t forSize)` in the hash table—this can be a joint effort. Test it by implementing a `delist? SYMBOL` command in the client, figuratively to remove a stock from the stock exchange. (Since erasing by marking a node a dummy is "too easy," and the savings are realized only if/when you rehash, these are bundled.)

(d) Bring the STL red-black tree into the timing comparisons by adapting your client to run with any of `set, multiset, map, or multimap`. [Because its `find` method returns a `const_iterator`, you may have problems with `const`... in which case the 3-pts.-per-fix-that-doesn't-cast (up to max of 9 extra-extra points) policy applies too.] Does your hash table beat the "official" tree? when code is optimized? (30 pts.)