Project Overview

There are **two** parts to Project 2. Part A is an emulation of an electronic stock exchange market. Part B requires you to make multiple implementations of the priority queue abstract data structure.

Part A Goals

- Gain experience using a priority queue.
- Become more proficient with object-oriented design.
- Become more proficient using the STL, especially `std::priority_queue<>`.

Part B Goals

- Implement multiple versions of the priority queue and compare their performances.
- Learn a data structure (the pairing heap) from a paper. This type of task is something that programmers have to do in their careers.
- Gain experience writing templated classes and subclasses.

Part A: Stock Market Emulation

Market Logic Overview

Your program `market` will receive a series of "**orders**," or intentions to buy or sell shares of a certain stock. An **order** consists of the following information:

- Timestamp - the timestamp that this order comes in
- Trader ID - the trader who is issuing the order
- Stock ID - the stock that the trader is interested in
- Buy/Sell Intent - whether the trader wants to buy or sell shares
- Price Limit - the max/min amount the trader is willing to pay/receive per share
- Quantity - the number of shares the trader is interested in

As each order comes in, your program should see if the new order can be matched with any previous orders. A new order can be matched with a previous order if:

- Both orders are for the same stock ID.
    - Trader ID does not matter; traders are allowed to buy from themselves.
- The buy/sell intents are different. That is, one trader is buying and the other trader is selling.
- The selling order's price limit is **less than or equal** to the buying order's price limit.

A buyer will **always** try to buy for the lowest price possible, and a seller will always try to sell for the highest price possible. Functionally, this means that whenever a possible match exists, it will **always** occur at the price of the *earlier* order. You need to use the STL's `priority_queue<>` data structure to match the *lowest-price seller* with the *highest-price buyer*.

© 2022 Regents of the University of Michigan

If the SELL order came first, then the price of the match will be at the price listed by the seller. If the BUY order came first, the match price will be the price listed by the buyer.

In the event of ties (e.g. the two cheapest sellers are offering for the same price), **always favor the order that came in earlier.**

Example Scenario

Consider the following series of orders:

Trader 1 wants to buy 10 shares of stock 2 for up to $100 each
Trader 2 wants to sell 20 shares of stock 2 for at least $10 each
Trader 3 wants to buy 10 shares of stock 2 for up to $1 each.

Actual input for this project is formatted differently. The *Input Details* section later covers how input will actually look.

Here is a detailed explanation of what would happen in the market exchange.

1. Trader #1 enters the market with the intent to buy 10 shares of stock #2 for up to $100/share
    - There are no other orders in the market yet, so trader #1 posts his order to the stock exchange.Trader #2 enters the market with the intent to sell 20 sh
    - ares of stock #2 for as low as $10 per share.
    - Trader #2 sees the order left by trader #1, and decides to first sell 10 shares of the stock to trader #1 for $100 each.
    - Even though she would have been happy selling for $10/share, she wants to make as much money as possible and takes advantage of the fact that trader #1 is willing to pay more.
    - After selling her 10 shares, she still has 10 shares she wants to get rid of. So, she now posts an order to sell her 10 leftover shares for $10 each.
2. Trader #3 enters the market with the intent to buy 10 shares of stock #2 for only $1/share.
    - Trader #3 sees the order left by trader #2 to sell for $10 each, but is not willing to pay that much.
    - As a result, Trader #3 leaves an order to express his intent.

Input

Input will arrive from standard input (`cin`). There are two input formats, *trade list* (`TL`) and *pseudorandom* (`PR`). You can assume that all numeric values within an input file will always fit within an integer variable (`int`). The first four lines of input will always be in the following format, regardless of input format, which you may assume are correctly formatted:

```
COMMENT: <COMMENT>
MODE: <INPUT_MODE>
NUM_TRADERS: <NUM_TRADERS>
NUM_STOCKS: <NUM_STOCKS>
```

`<COMMENT>` is a string terminated by a newline, which should be ignored. You should comment your test files to explain their purpose. There will be exactly one line of comments.

© 2022 Regents of the University of Michigan

<INPUT_MODE> will either be the string **"TL"** or **"PR"** (without the quote marks). TL indicates that the rest of input will be in the trade list format, and PR indicates that the rest of input will be in pseudo-random format. Details for these input formats will be explained shortly.

<NUM_TRADERS> and **<NUM_STOCKS>**, respectively, will tell you how many traders and stocks will exist. These are unsigned integers.

You may assume that the first four lines of input will be correctly formatted.

Trade List (TL) Input:

If **<INPUT_MODE>** is TL, the rest of the input will be a series of lines in the following format:

<TIMESTAMP> <BUY/SELL> T**<TRADER_ID>** S**<STOCK_NUM>** $**<PRICE>** #**<QUANTITY>**

Each line represents a unique order. For example, the line:

```
0 BUY T1 S2 $100 #50
```

Can be translated as,

"At timestamp 0, trader 1 is willing to buy 50 shares of stock 2 for up to $100/share."

You can assume that each line which describes an order will be well-formatted, meaning that it might have invalid values, but not an unreadable format. For example, there will always be a "T" before the `<TRADER_ID>`, always be an "S" before the `<STOCK_NUM>`, etc. The things you expect to be numbers will always be numbers (not strings, not `double` values with a decimal point).

When you want to read a trade, DO NOT getline() the entire line, copy it to a stringstream, then extract what you want one piece at a time: that's processing the same input three times. Instead, use >> to extract exactly what you need. Starting out, a trade needs some type of integer (the timestamp), a `string` (BUY or SELL), a `char` (the T before the trader number), some type of integer for the trader number, another `char` for the S, etc.

Errors you must check for in TL Input mode

You can assume that you will always find a number where you expect to find one (though they might be invalid values). To avoid unexpected losses of large amounts of money, you must check for each of the following:

- **<TIMESTAMP>** is non-negative integer (no one can trade before the market opens)
- **<TRADER_ID>** and **<STOCK_ID>** are respectively integers in ranges **[0, <NUM_TRADERS>)** and **[0, <NUM_STOCKS>)**
  - e.g if **<NUM_TRADERS>** is 5, then valid trader IDs are 0, 1, 2, 3, 4
- **<PRICE>** and **<QUANTITY>** are positive (non-zero) integers.
- Timestamps are non-decreasing.
  e.g. 0 cannot come after 1, but there can be multiple orders with the same timestamp.

If you detect invalid input at any time during the program, print a helpful message to cerr and `exit(1)`.
You do not need to check for input errors not explicitly mentioned here.

© 2022 Regents of the University of Michigan

Pseudorandom (PR) Input:

If **<INPUT_MODE>** is PR, the rest of input will consist of these three lines in this format:

```
RANDOM_SEED: <SEED>
NUMBER_OF_ORDERS: <NUM_ORDERS>
ARRIVAL_RATE: <ARRIVAL_RATE>
```

- **RANDOM_SEED** — An integer used to initialize the random seed.
- **NUMBER_OF_ORDERS** — The number of orders to generate. You may assume that this value will fit in an `int`.
- **ARRIVAL_RATE** — An integer corresponding to the average number of orders per timestamp.

All three of these values are unsigned integers.

PR input will always be correctly formatted.

Generating orders with `P2random.h`

In the project folder, we provide a pair of files to generate the orders in PR mode. This is to make sure that the generation of pseudo-random numbers is consistent across platforms. The class `P2random` contains the following function:

```
void P2random::PR_init(std::stringstream& ss,    unsigned int seed,
                       unsigned int num_traders, unsigned int num_stocks,
                       unsigned int num_orders,  unsigned int arrival_rate);
```

`P2random::PR_init()` will set the contents of the stringstream argument (`ss`) so that you can use it just like you would use `cin` for `TL` mode:

You may find the following C++ code helpful in reducing code duplication:

```
// Read values that exist in both files (mode, num_traders, num_stocks)
stringstream ss;

if (mode == "PR") {
    // TODO: Read PR mode values from cin (seed, num_orders, arrival_rate)
    P2random::PR_init(ss, seed, num_traders, num_stocks, num_orders, rate);
} // if

// TODO: Make a function named something like this, accepting a stream
//       reference variable.
void processOrders(istream &inputStream) {
    // Read orders from inputStream, NOT cin
    while (inputStream >> var1 >> var2 ...) {
        // process orders
```

© 2022 Regents of the University of Michigan

```
    } // while
} // processOrders()

// Call the function with either the stringstream produced by PR_init()
// or cin
if (mode == "PR")
    processOrders(ss);
else
    processOrders(cin);
```

TL & PR Comparison

The following two input files are in different modes, but consist of the same buy/sell orders; thus they generate **the same output**.

```
COMMENT: Spec example, TL mode generating some orders.
MODE: TL
NUM_TRADERS: 3
NUM_STOCKS: 2
0 SELL T1 S0 $100 #44
1 SELL T2 S0 $56 #42
2 BUY T1 S0 $73 #19
2 BUY T2 S0 $34 #50
2 SELL T2 S1 $86 #23
2 SELL T0 S0 $20 #39
2 BUY T1 S1 $49 #24
2 SELL T1 S1 $83 #45
3 SELL T0 S1 $64 #22
3 SELL T2 S1 $6 #19
3 SELL T0 S1 $42 #37
4 SELL T1 S0 $10 #44
```

```
COMMENT: Spec example, PR mode generating the same sequence of orders.
MODE: PR
NUM_TRADERS: 3
NUM_STOCKS: 2
RANDOM_SEED: 104
NUMBER_OF_ORDERS: 12
ARRIVAL_RATE: 10
```

Command Line Interface

Your program `market` should take the following case-sensitive command-line options that will determine which types of output to generate. Details about each output mode are under the **Output Details** section.

- `-v, --verbose`
  An optional flag that indicates verbose output should be generated
- `-m, --median`
  An optional flag that indicates median output should be generated
- `-i, --trader_info`
  An optional flag that indicates that the trader details output should be generated.
- `-t, --time_travelers`
  An optional flag that indicates that time traveler's output should be generated.

Examples of legal command lines:

- `./market < infile.txt > outfile.txt`
- `./market --verbose --trader_info < infile.txt`
- `./market --verbose --median > outfile.txt`
- `./market --time_travelers`
- `./market --trader_info --verbose`
- `./market -vmit`

Example of an illegal command line:

- `./market -v -q`

We will not be specifically error-checking your command-line handling; however we expect that your program conforms with the default behavior of `getopt_long()`. Incorrect command-line handling may lead to a variety of difficult-to-diagnose problems.

Output

The output generated by your program will depend on the command line options specified at runtime. With the exception of program startup and the end of program summary output, all output is optional and should not be generated unless the corresponding command line flag is set.

Program Startup

Your program should always print the following line **before** reading any orders:

```
Processing orders...
```

Verbose Option

If and only if the **`--verbose/-v`** option is specified on the command line (see above), whenever a trade is completed you should print on a single line:

```
Trader <BUYING_TRADER_NUM> purchased <NUM_SHARES> shares of Stock <STOCK_NUM> from Trader <SELLING_TRADER_NUM> for $<PRICE>/share
```

Example:

Given the following list of orders:

```
0 SELL T1 S0 $125 #10
```

```
0 BUY T2 S0 $1 #100
0 SELL T3 S0 $100 #10
0 SELL T4 S0 $80 #10
0 BUY T5 S0 $200 #4
```

No trades are possible until the **5th** order comes in. When the 5th order comes in, you should print:

```
Trader 5 purchased 4 shares of Stock 0 from Trader 4 for $80/share
```

Median Option

If and only if the **`--median/-m`** option is specified on the command line, at the times detailed in the Market Logic section, your program should print the current median match price for all stocks in ascending order by stock ID. **If no trades have been made for a given stock, it does not have a median, and thus nothing should be printed for that stock's median.** In the case that a median does exist, you should print:

```
Median match price of Stock <STOCK_ID> at time <TIMESTAMP> is $<MEDPRICE>
```

If there are an even number of trades, take the average of the middle-most and **use integer division** to compute the median**. If a particular timestamp in the input file produces no trades, you still print a median for that timestamp (limited by the rules above); see the "Detailed Algorithm" section below.**

Example

Given the following transactions:

```
Trader 5 purchased 4 shares of Stock 9 from Trader 4 for $80/share
Trader 2 purchased 1 shares of Stock 9 from Trader 10 for $50/share
```

The median match price for Stock 9 after these two transactions is $(80 + 50 / 2) or $65. If the timestamp changed and the `--median` option was specified, your program should print:

```
Median match price of Stock 9 at time 0 is $65
```

The median match price only considers transactions, and does not consider the quantity traded in each trade. You must keep a running median, and do it efficiently! Simply sorting the values every time you need the middle value(s) will take too much time.

Summary Output

After all input has been read and all possible trades completed, the following output should always be printed without any preceding newlines before any optional end of day output:

```
---End of Day---
Trades Completed: <TRADES_COMPLETED><NEWLINE>
```

`<TRADES_COMPLETED>` is the total number of trades completed over the course of the program. The number of shares traded doesn't matter; a seller and a buyer making a trade adds 1 to this count.

Trader Info Output

If and only if the **`--trader_info/-i`** option is specified on the command line, you should print the following line without any preceding newlines.

```
---Trader Info---
```

Followed by lines in the following format for ***every*** trader in ascending order (0, 1, 2, etc.):

```
Trader <TRADER_ID> bought <NUMBER_BOUGHT> and sold <NUMBER_SOLD> for a net transfer of $<NET_VALUE_TRADED>(newline)
```

These numbers are orders across all stocks. Example:

```
---Trader Info---
Trader 0 bought 0 and sold 2 for a net transfer of $166
Trader 1 bought 63 and sold 0 for a net transfer of $-5359
Trader 2 bought 24 and sold 85 for a net transfer of $5193
```

The numbers bought and sold, and net value traded include all stocks that the trader happened to trade in. The total number bought by all traders should equal the total number sold by all traders, and the total of all the net transfers should be 0. This shows that our simulation is a zero-sum game.

Time Travelers Output

In time-travel trading, we want to find the ideal time that a time traveler theoretically could have bought shares of a stock and then later sold that stock to maximize profit.

If and only if the **`--time_travelers/-t` option** is specified on the command line, you should print the following line without any preceding newlines:

```
---Time Travelers---
```

Followed by time traveler's output for every stock in ascending order in the following format:

```
A time traveler would buy Stock <STOCK_ID> at time <TIMESTAMP1> for $<PRICE1> and sell it at time <TIMESTAMP2> for $<PRICE2>
```

`TIMESTAMP1` will correspond to an actual sell order that came in during the day, and **`TIMESTAMP2`** will correspond to an actual buy order that came *after* the sell order that maximizes the time traveler's profit.

- When calculating the results for time traveler trading, the only factors are the time and price of orders that happened throughout the day. Quantity is not considered.
- If there would be more than one answer that yields the optimal result, you should prefer buy/sell

pairs with the lower **<TIMESTAMP2>**, then the lower **<TIMESTAMP1>** if they have the same **<TIMESTAMP2>**

If there are no valid buy/sell order pairs (none exist, or none result in a profit) you should print:

`A time traveler could not make a profit on Stock <STOCK_ID>`

Your time traveler must work in Θ(n) time (Θ(1) per buy/sell order), using Θ(1) memory per stock.

Detailed Algorithm

Following these steps in order will help guarantee that your program prints the correct output at the proper times. Details of the output options will be covered later under the *Output* section.

`CURRENT_TIMESTAMP` starts at 0, and its value is updated throughout the run of the program.

1. Print program startup output
2. Read the next order from input.
3. If the new order's `TIMESTAMP != CURRENT_TIMESTAMP`
    a. If the `--median` option is specified, print the median price of all stocks that have been traded on at least once by this point in ascending order by stock ID.
    b. Set `CURRENT_TIMESTAMP` to be the new order's `TIMESTAMP`.
4. Make all possible matches between the new order with any previously posted orders. If the new order cannot be fully fulfilled, save it to match with future orders.
    a. If the `--verbose` option is specified, you should print the details of each completed match to `stdout/cout`.
5. Repeat steps 2-4 until there are no more orders.
6. Output median information again if the `--median` flag is set
7. Print end-of-day output
8. Output the Trader information if the `--trader_info` flag is set.
9. Output the time traveler's output if the `--time_travelers` flag is set.

Full Example

Input File Contents:

```
COMMENT: Spec example, TL mode generating some orders.
MODE: TL
NUM_TRADERS: 3
NUM_STOCKS: 2
0 SELL T1 S0 $100 #44
1 SELL T2 S0 $56 #42
2 BUY T1 S0 $73 #19
2 BUY T2 S0 $34 #50
2 SELL T2 S1 $86 #23
2 SELL T0 S0 $20 #39
2 BUY T1 S1 $49 #24
2 SELL T1 S1 $83 #45
3 SELL T0 S1 $64 #22
3 SELL T2 S1 $6 #19
3 SELL T0 S1 $42 #37
4 SELL T1 S0 $10 #44
```

Output when run with flags: **`--verbose --median --trader_info --time_travelers`**

```
Processing orders...
Trader 1 purchased 19 shares of Stock 0 from Trader 2 for $56/share
Trader 2 purchased 39 shares of Stock 0 from Trader 0 for $34/share
Median match price of Stock 0 at time 2 is $45
Trader 1 purchased 19 shares of Stock 1 from Trader 2 for $49/share
Trader 1 purchased 5 shares of Stock 1 from Trader 0 for $49/share
Median match price of Stock 0 at time 3 is $45
Median match price of Stock 1 at time 3 is $49
Trader 2 purchased 11 shares of Stock 0 from Trader 1 for $34/share
Median match price of Stock 0 at time 4 is $34
Median match price of Stock 1 at time 4 is $49
---End of Day---
Trades Completed: 5
---Trader Info---
Trader 0 bought 0 and sold 44 for a net transfer of $1571
Trader 1 bought 43 and sold 11 for a net transfer of $-1866
Trader 2 bought 50 and sold 38 for a net transfer of $295
---Time Travelers---
A time traveler would buy Stock 0 at time 1 for $56 and sell it at time 2 for $73
A time traveler could not make a profit on Stock 1
```

Hints & Advice

The project is specified so that the various pieces of output are all independent. We strongly recommend working on them separately, implementing one command-line option at a time. The autograder test cases are named so that you can get a sense of where your bugs might be.

Write your code with verbose mode right at the beginning: if you don't, and the number of trades is wrong at the end of the day, you'll have no idea why. Work on the trader info, time traveler, and running median after you have the other parts working.

We place a strong emphasis on time budgets in this project. This means that you may find that you need to rewrite sections of your code that are performing too slowly or consider using different data structures. Pay attention to the Big-O complexities of your implementation and examine the tradeoffs of using different possible solutions. Using `perf` on this project will be incredibly helpful in finding which

© 2022 Regents of the University of Michigan

© 2022 Regents of the University of Michigan

parts of your code are taking up the most amount of time, and remember that `perf` is most effective when your code is well modularized (i.e. broken up into functions).

Running your code with `valgrind` can help you find and remove undefined (buggy) behavior and memory leaks from your code. This can save you from losing points in the final run when you mistakenly believe your code to be correct.

It is extremely helpful to compile your code with the following gcc options: `-Wall -Wextra -Werror -Wconversion -pedantic`. This way the compiler can warn you about parts of your code that may result in unintended/undefined behavior. Compiling with the provided Makefile does this for you.

To get your market logic working, you will need to create functors to help control what each priority queue defines as the maximum priority. See the About Comparators section below.

We have several videos for this project: overall project 2 including time traveler (with timestamps of each important portion), how to efficiently calculate a running median, and one about the priority queues (Part B, again including timestamps of topics).

Don't spend all your time getting part A working before thinking about part B. Start reading up on the Fredman paper to prepare for Pairing Heaps, look at the `UnorderedPQ.h`, and start working on `SortedPQ.h`: this one is mostly about using the STL, there's very little actual code to write (the best solution is on the order of 5 lines of code to write). Lecture 8 (on Heaps) is what you need for the implementation in `BinaryPQ.h`.

© 2022 Regents of the University of Michigan

Part B: Priority Queues

For this part of the project, you are required to implement and use your own priority queue containers. You will implement a **"sorted array priority queue"**, a **"binary heap priority queue"**, and a **"pairing heap priority queue"** that implements the interface defined in **Eecs281PQ.h**, which we provide**.**

To implement these priority queues, you will need to fill in separate header files, **SortedPQ.h**, **BinaryPQ.h**, and **PairingPQ.h**, containing all the definitions for the functions declared in **Eecs281PQ.h**. We have provided these files with empty function definitions for you to fill in.

We provide a good implementation of a very bad priority queue approach called the "**Unordered priority queue**" in **`UnorderedPQ.h`**, which does a linear search for the most extreme element each time it is needed. You can test your other priority queue implementations against `UnorderedPQ`. You can also use this priority queue to ensure that your other priority queues are returning elements in the correct order.

These files specify more information about each priority queue type, including runtime requirements for each method and a general description of the container.

You are **not** allowed to modify `Eecs281PQ.h` in any way. Nor are you allowed to change the interface (names, parameters, return types) that we give you in any of the provided headers. You are allowed to add your own private helper functions and variables to the other header files as needed, so long as you still follow the requirements outlined in both the spec and the comments in the provided files.

These priority queues can take in an optional comparison functor type, `COMP`. Inside the classes, you can access an instance of `COMP` with `this->compare`. All of your priority queues must default to be MAX priority queues. This means that if you use the default comparison functor with an integer PQ, `std::less<int>`, the PQ will return the *largest* integer when you call `top()`. Here, the definition of max (aka most extreme) is entirely dependent on the comparison functor. For example, if you use `std::greater<int>`, it will become a min-PQ. The definition is as follows:

If A is an arbitrary element in the priority queue, and top() returns the "most extreme" element. `compare(top(), A)` should always return false (A is "less extreme" than `top()`).

It might seem counterintuitive that `std::less<>` yields a max-PQ, but this is consistent with the way that the STL `priority_queue<>` works (and other STL functions that take custom comparators, like `sort`).

We will compile your priority queue implementations with our own code to ensure that you have correctly and fully implemented them. To ensure that this is possible (and that you do not lose credit for these tests), do not define a main function in one of the PQ headers, or any header file for that matter.

© 2022 Regents of the University of Michigan

Eecs281PQ interface

Member variables:

```
compare                 // More on this below;
                        // see "Implementing the Priority Queues"
```

Functions:

```
push(const TYPE& val)   // inserts a new element into the priority queue

top()                   // returns the highest priority element in the
                        // priority queue

pop()                   // removes the highest priority element from
                        // the priority queue

size()                  // returns the size of the priority queue

empty()                 // returns true if the priority queue is empty,
                        // false otherwise
```

Sorted Priority Queue

The *sorted priority queue* implements the priority queue interface by maintaining a **sorted** vector. Complexities and details are in `SortedPQ.h`.

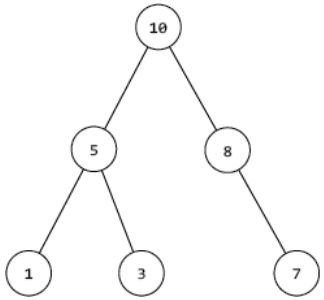
Binary Heap Priority Queue

Binary heaps will be covered in lecture. We also highly recommend reviewing Chapter 6 of the CLRS book. Complexities and details are in `BinaryPQ.h`.

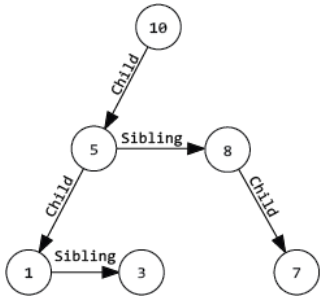
Pairing Priority Queue

Pairing heaps are an advanced heap data structure that can be quite fast. In order to implement the pairing priority queue, read the two papers we provide you describing the data structure. Complexity details can be found in `PairingPQ.h`. We have also included a couple of diagrams that may help you understand the tree structure of the pairing heap.

Below is the pairing heap modeled as a tree, in which each node is greater than each of its children:



To implement this structure, the pairing heap will use child and sibling pointers to have a structure like this:



Implementing the Priority Queues

Look through the included header files: you need to add code in `SortedPQ.h`, `BinaryPQ.h`, and `PairingPQ.h`, and this is the order that we would suggest implementing the different priority queues. Each of these files has TODO comments where you need to make changes. We wanted to provide you with files that would compile when you receive them, so some of the changes involve deleting and/or changing lines that were only placed there to make sure that they compile. For example, if a function was supposed to return an integer, NOT having a return statement that returns an integer would produce a compiler error. Also, functions which accept parameters have had the name of the parameter commented out (otherwise you would get an unused parameter error). Look at `UnorderedPQ.h` as an example, it's already done. There's also `UnorderedFastPQ.h`, which uses mutable to make the unordered PQ faster.

When you implement each priority queue, you cannot compare *data* yourself using the < operator. You can still use < for comparisons such as a vector index to the size of the vector, but you must use the provided comparator for comparing the data stored inside your priority queue. Notice that `Eecs281PQ` contains a member variable named `compare` of type `COMP` (one of the templated class types). Although

the other classes inherit from `Eecs281PQ`, you cannot access the `compare` member directly, you must always say `this->compare` (this is due to a template inheriting from a template). Notice that in `UnorderedPQ` it uses `this->compare` by passing it to the `max_element()` algorithm to use for comparisons.

When you write the `SortedPQ` you cannot use `binary_search()` from the STL, but you wouldn't want to: it only returns a `bool` to tell you if something is already in the container or not! Instead use the `lower_bound()` algorithm (which returns an iterator), and you can also use the `sort()` algorithm -- you don't have to write your own sorting function. You do however have to pass the `this->compare` functor to both `lower_bound()` and `sort()`, similar to the way that `UnorderedPQ` passes it to `max_element()`.

The `BinaryPQ` is harder to write, and requires a more detailed and careful use of the comparison functor, and you have to know how one works to write one in the first place, even for `UnorderedPQ` to use. See the About Comparators section below.

Compiling and Testing Priority Queues

You are provided with a test file, `testPQ.cpp`. `testPQ.cpp` contains examples of unit tests you can run on your priority queues to ensure that they are correct; however, it is **not** a complete test of your priority queues; for example it does not test `updatePriorities()`. It is especially lacking in testing the `PairingPQ` class, since it does not have any calls to `addNode()` or `updateElt()`. You should add tests to this file or make your own test cases in this form.

Using the 281 Makefile, you can compile `testPQ.cpp` by typing in the terminal: `make testPQ`. You may use your own Makefile, but you will have to make sure it does not try to compile your driver program as well as the test program (i.e., use at your own risk).

Logistics

The `std::priority_queue<>`

The STL `priority_queue<>` data structure is basically an efficient implementation of the binary heap which you are also coding in `BinaryPQ.h`. To declare a `priority_queue<>` you need to state either one or three types:

1) The data type to be stored in the container. If this type has a natural sort order that meets your needs, this is the only type required.
2) The underlying container to use, usually just a vector<> of the first type.
3) The comparator to use to define what is considered the highest priority element.

© 2022 Regents of the University of Michigan

If the type that you store in the container has a natural sort order (i.e. it supports `operator<()`), the `priority_queue<>` will be a max-heap of the declared type. For example, if you just want to store integers, and have the largest integer be the highest priority:

```
priority_queue<int> pqMax;
```

When you declare this, by default the underlying storage type is `vector<int>` and the default comparator is `less<int>`. If you want the smallest integer to be the highest priority:

```
priority_queue<int, vector<int>, greater<int>> pqMin;
```

If you want to store something other than integers, define a custom comparator as described below.

About Comparators

The functor must accept two of whatever is stored in your priority queue: if your PQ stores integers, the functor would accept two integers. If your PQ stores pointers to orders, your functor would accept two pointers to orders (actually two `const` pointers, since you don't have to modify orders to compare them).

Your functor receives two parameters, let's call them `a` and `b`. It must always answer the following question: **is the priority of `a` less than the priority of `b`**? What does lower priority mean? It depends on your application. For example, refer back to the section on "Market Logic Overview": if you have multiple Buy orders on the same stock, which order has the highest priority if a Sell order comes in? In the same way, Sell orders have a different way of determining the highest priority. This means you will need at least two different functors: one for a priority queue containing Buy orders and a different functor for a priority queue containing Sell orders.

When you would have wanted to write a comparison, such as:

```
if (data[i] < data[j])
```

You would instead write:

```
if (this->compare(data[i], data[j])
```

Your priority queues must work **in general**. In general, a priority queue has no idea what kind of data is inside of it. That's why it uses `this->compare` instead of `<`. What if you wanted to perform the comparison `if (data[i] > data[j])`? Use the following:

```
if (this->compare(data[j], data[i])
```

Libraries and Restrictions

We highly encourage the use of the STL for part A, with the exception of these prohibited features:

- The thread/atomics libraries (e.g., boost, pthreads, etc) which spoil runtime measurements.

© 2022 Regents of the University of Michigan

- Smart pointers (both unique and shared).

In addition to the above requirements, you may not use any STL facilities which trivialize your implementation of your priority queues, including but not limited to `priority_queue<>`, `make_heap()`, `push_heap()`, `pop_heap()`, `sort_heap()`, `partition()`, `partition_copy()`, `stable_partition()`, `partial_sort()`, or `qsort()`. However, you **may** (and probably should) use `sort()`. Your main program (Part A) **must** use `priority_queue<>`, but your PQ implementations (Part B) **must not**. If you are unsure about whether a given function or container may be used, ask on Piazza.

Testing and Debugging

Part of this project is to prepare several test files that expose defects in a solution. Each test file is an input file. We will give your test files as input to intentionally buggy solutions and compare the output to that of a correct project solution. You will receive points depending on how many buggy implementations your test files expose.

The autograder will also tell you if one of your test files exposes bugs in your solution. For the first such file that the autograder finds, it will give you the correct output and the output of your program.

Test File Details

Your test files must be Trade List (TL mode) input files, **and may have no more than 30 trades** in any one file. You may submit up to 10 test files (though it is possible to expose all buggy solutions with fewer test files).

Test files should be named in the following format:

`test-<N>-<FLAG>.txt`

- **<N>** is an integer in the range [0, 9]
- **<FLAG>** is one (and only one) of the following characters `v`, `m`, `i`, or `t`.
    - This tells the autograder which command-line option to test with (verbose, median, trader info, or time traveler).
    - These test files are run with Part A (stock market), not Part B (priority queues).

For example, `test-1-m.txt` and `test-5-v.txt` are both valid test file names.

The tests on which the autograder runs your solution are NOT limited to 30 trades in a file; your solution should not impose any size limits (as long as memory is available)

Submitting to the Autograder

Do all of your work (with all needed files, as well as test files) in some directory other than your home directory. This will be your “submit directory”. Before you turn in your code, be sure that:

- Every source code and header file contains the following project identifier in a comment at the top of the file:

- `// Project Identifier: 0E04A31E0D60C01986ACB20081C9D8722A1899B6`
- The Makefile must also have this identifier (in the first TODO block).
- DO NOT copy the above identifier from the PDF! It might contain hidden characters. Copy it from the README file instead (this file is included with the starter files).
- Your makefile is called `Makefile`. Typing ‘`make -R -r`’ builds your code without errors and generates an executable file called "`market`". (The command-line options -R and -r disable automatic build rules; these automatic rules do not exist on the autograder).
- Your `Makefile` specifies that you are compiling with the gcc optimization option `-O3`. This is extremely important for getting all of the performance points, as `-O3` can speed up code by an order of magnitude.
- Your test files are named as described and no other project file names begin with test. Up to 10 test files may be submitted.
- The total size of your program and test files does not exceed 2MB.
- You don't have any unnecessary files (including temporary files created by your text editor and compiler, etc) or subdirectories in your submit directory (e.g., the .git folder used by git source code management).
- Your code compiles and runs correctly using version 6.2.0 of the g++ compiler on the CAEN servers. To compile with g++ version 6.2.0 on CAEN you **must** put the following at the top of your `Makefile` (or use our provided `Makefile`):

```
PATH := /usr/um/gcc-6.2.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-6.2.0/lib64
LD_RUN_PATH := /usr/um/gcc-6.2.0/lib64
```

For Part A (stock market simulation), turn in all of the following files:

- All your .h and or .cpp files for the project (NOT including your priority queue implementations)
- Your Makefile
- Your test files

For Part B (priority queues), turn in all of the following files:

- Your priority queue implementations: `SortedPQ.h`, `BinaryPQ.h`, `PairingPQ.h`
- Your `Makefile` (actually optional, it includes itself, but we’ll replace this with our `Makefile`)

If **any** of your submitted priority queue files do not compile, **no** unit testing (Part B) can be performed.

You must prepare a compressed tar archive (.tar.gz file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission in one directory. In this directory, run

```
tar -czvf submit.tar.gz *.cpp *.h Makefile test-*.txt
```

This will prepare a suitable file in your working directory. Alternatively, the 281 `Makefile` has useful targets `fullsubmit` and `partialsubmit` that will do this for you. Use the command `make help` to find out what else it can do!

Submit your project files directly to either of the two autograders at:

https://g281-1.eecs.umich.edu/ or https://g281-2.eecs.umich.edu/. **When the autograders are turned on and accepting submissions, there will be an announcement on Piazza.** The autograders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to two times per calendar day, per part (A or B; more per day in Spring). If you use a late day to extend one part, the other part is automatically extended also. For this purpose, days begin and end at midnight (Ann Arbor local time). **We will use your best submission for final grading**. If you would instead like us to use your LAST submission, see the autograder FAQ page, or use this form.

Please make sure that you read all messages shown at the top section of your autograder results! These messages often help explain some of the issues you are having (such as losing points for having a bad Makefile or why you are segfaulting). Also be sure to check whether the autograder shows that one of your own test files exposes a bug in your solution.

Grading

- 60 pts total for part A (all your code, using the STL, but not using your priority queues)
  - 45 pts — correctness & performance
  - 5 pts — no memory leaks
  - 10 pts — student-provided test files
- 40 pts total for part B (our main, your priority queues)
  - 20 pts — pairing heap correctness & performance
  - 5 pts — pairing heap has no memory leaks
  - 10 pts — binary heap correctness & performance
  - 5 pts — sorted heap correctness & performance

Although we will not be grading your code for style, we reserve the right to not help you in office hours if your code is unreadable. Readability is generally defined as follows:

- Clean organization and consistency throughout your overall program
- Proper partitioning of code into header and cpp files
- Descriptive variable names and proper use of C++ idioms
- Omitting globals, `goto`s, unnecessary literals, or unused libraries
- Effective use of comments
- Reasonable formatting - e.g an 80 column display
- Code reuse/no excessive copy-pasted code blocks

When you start submitting test files to the autograder, it will tell you (in the section called "Scoring student test files") how many bugs exist, the number needed to start earning points, and the number needed for full points. It will also tell you how many are needed to start earning an extra submit/day!

Part A Test Case Legend

Some of the test cases on the autograder follow a pattern and others do not. The test files that do not fit this pattern you can think of as handwritten cases which test a particular aspect of your program,

though some can be very large.

INV*: Invalid test cases, derived from the “Errors you must check for” section.
st*: The example from the project specification, in TL mode.
sp*: The example from the project specification, in PR mode.
P*: PR mode files, of a varying number of trade orders (“PS” is smaller, “Pm” is medium to large).
K*: TL mode files, contain a large number of trades.
M*: TL mode files, tend to be small-medium number of trades.
F*: TL or PR mode files, tend to have a small-medium number of trades.

A: Runs with all flags turned on: `-i -m -t -v`.
i, *m*, *t*, *v*: Run with some of the flags (`-i`, `-m`, `-t`, `-v`).

Part B Test Case Legend

When your priority queues (`SortedPQ`, `BinaryPQ`, and `PairingPQ`) are compiled with our `main()`, we will perform unit testing on your data structures. These test cases all have three-letter names:

1) First letter: **B**inary PQ, **S**orted PQ, **P**airing PQ
2) Second letter: **P**ush, **R**ange, **U**pdate priorities, **A**ddnode, update**E**lt, **C**opy constructor, **O**perator =
3) Third letter: **S**mall, **M**edium, **L**arge, e**X**tra-large

A “push” test uses the `.push()` member function to insert numerous values into your priority queue. After that, the values will be checked via `.top()` and `.pop()` until the container is empty, to make sure that every value came out in the correct order. If the “push” test goes over on time, it *might* be the fault of your `.pop()`, not your `.push()`, because both must be called to verify that your container works properly.

A “range” test uses the range-based constructor, from `[start, end)` to insert values into your container, then the test proceeds as described above for the “push” test. The start iterator is inclusive while the end is exclusive, as is normal for the STL.

The “update priorities” tests use `.push()` to fill your container, then half of the values that were given to you are modified (hint, this is accomplished with pointers). After `.updatePriorities()` is called, all of the values are popped out and tested as above.

The first three tests above are run for every priority queue type. The ones below are run only on the `PairingPQ`.

The “addNode” tests use `.addNode()` to fill the container instead of `.push()`, and every value is checked through the returned pointer to make sure that it matches.

The “updateElt” tests use `.addNode()` to fill the container, then half of the values have their priority

increased by a random amount using `.updateElt()`. After that, values are popped off one at a time, checking to make sure that each value is correct.

The “copy constructor” and “operator=” tests first fill one `PairingPQ` using `.push()`. Then they use the stated method to create a second `PairingPQ` from the first. Lastly every value is popped from **both** priority queues, making sure that every value is correct (and also ensuring that a deep copy was performed, not a shallow copy).

Hints for the PQ Implementations

When you implement the priority queues, read **UnorderedPQ.h** (and **UnorderedFastPQ.h**) first, understand them. Write `SortedPQ<>` first, then `BinaryPQ<>`, then `PairingPQ<>`. Remember to modify the testPQ.cpp file, add more tests to it, specifically for the `PairingPQ<>`. When writing the PQ implementations, remember that each header file must #include whatever it needs, don’t count on `main()` doing it for you, since we will be testing your PQ implementations with our `main()`. For example, if you need `swap()` for a particular PQ be sure to `#include <utility>` inside that PQ.

When compiling your unit testing (`make testPQ` for `testPQ.cpp`), it’s important to realize how templates work: if the code in your cpp file doesn’t call a particular member function, such as `updatePriorities()`, the compiler doesn’t even compile it!

When you’re implementing your PQs, remember that the `vector` class has a `.back()` member function! Don’t write `data[data.size() - 1]`, it’s much harder to read.

For the `BinaryPQ<>`, when you want to translate from 1-based to 0-based indexing, there's 3 options that you can consider:

1. Change all the formulas from the slides to be 0-based. Hardest but best option.
2. Push a "dummy" element at the start of the vector, and then lie about the size, empty, etc. Easy but poor option (it wastes space, is sometimes hard to get the syntax correct, and adds some confusion because the size of the data vector is not the same as the size of the PQ).
3. Use the functions provided below, put them inside the private section of the class declaration. When the slides say to access `heap[k]`, use `getElement(k)` instead. Easy and 2nd-best solution.

```
// Translate 1-based indexing into a 0-based vector
TYPE &getElement(std::size_t i) {
    return data[i - 1];
} // getElement()

const TYPE &getElement(std::size_t i) const {
    return data[i - 1];
} // getElement()
```


Remember, don’t leave the Pairing Heap until the last minute! Be reading and thinking about it before you start coding, and draw out examples on paper to help while you’re coding. Here’s some more tips:

- You can add other private member variables and functions, such as the current size and a `meld()` function.
- Make sure ALL of your constructors initialize ALL of your member variables.
- Even if you think it's working, run your Pairing Heap with `valgrind` to check for memory errors and/or leaks. This is good advice for the entire project.
- You are not allowed to change from sibling and child pointers to a `std::deque<>` or `std::vector<>` of children pointers: this is slower than using sibling/child., and the autograder timings are based on the sibling/child approach.
- You are allowed (and need) to add one more pointer to the `Node` structure. You can use either a parent (pointing up) or a previous (points left except leftmost points to parent). You can also add a public function to return it if you want to. We didn't put this variable or function in the starter file because some students want parent, some want previous.
- Pointers passed to `meld()` must each point to the "root" node of a Pairing Heap. Root nodes never have siblings! This is important to making `meld()` as simple and as fast as it should be.
- Don't write a copy constructor and copy the code for `operator=()`! Use the copy-swap method outlined in lecture 5.
- You are allowed to write a public function, such as `print()` to display the entire pairing heap, and only use it for testing.
- When you need to traverse an entire pairing heap (print, copy constructor, destructor, etc.), think about the project 1 approach (see below)!
- When you copy a pairing heap, it must match in values and behavior, but not necessarily structure.
- Don't use recursion, it's very easy to have a tree structure that causes a stack overflow.

What was the Project 1 approach?

- Add the “start” location to a deque
- While the deque is not empty:
    - Set the current location to the “next” in the deque
    - Pop the “next” location from the deque
    - Add current's “nearby” locations to the deque
    - Do something with the current location (print it, copy it, destroy it, etc.)

What is the “start” of a Pairing Heap? The root. What is “nearby” the current node? Its child and sibling, but NOT the parent/previous: this would violate the Project 1 approach of never adding anything to the deque twice. In P1 you skipped adding invalid choices, in P2 skip adding null pointers.