Q3

Lock Implementation:

The Lock Implementation tends to generally have a sequential equivalent ordering to its output in adds-to-deletes down to the nanosecond. When the time in nanoseconds are equivalent between two operations (which does occur, although rare) then it may show that a delete of an item occurred before it's added, however can be inferred that this occurred the other way around.

The reason why the output follows a relatively nice ordering is due to the total locking of each individual bucket. Only one thread may operate on a single priority at a time, whether it be a *put* or *get,* thus making operations truly occur sequentially on that priority. Operations may occur in parallel on different priorities, which may have the timing of such operations interleave, however this would not obstruct the integrity of adding to/removing from the priority queue. Additionally, timestamps are generated in the locked area, thus limiting overlaps in time and give more accurate occurrences of operations, as other threads would not have been able to operate on the bucket at the time of the stamping.

However, when a thread is attempting to get the highest priority item, if it has already looked at a higher priority bucket and moves onto the next priority, another thread may add to a higher priority, where the thread performing the *get* will continue to look for the next highest priority item. This may occur, but generally because each bucket has its own lock each thread will only operate on it when its alone in that bucket. Thus operations will be "real" in that bucket at that time of execution.

Lock-free Implementation

The Lock-free Implementation does not necessarily have a nice, equivalent sequential ordering to it, although it can be pieced together in small chunks to make sense.

One reason for this is the time at which actions are stamped. Because only swaps occur atomically, the timestamp statements may not be run directly after the swap is confirmed as another thread may have taken control of the CPU. This other thread can complete a different operation and stamp itself prior to that initial thread. Thus an actions timestamp is not extremely accurate. This makes certain operations seem to occur out of order, although this would not truly be the case.

Similarly to the Lock Implementation, higher priority items may be added to the queue after a thread has checked that priority, where it wont go back to fetch it. Thus, a lower priority item may be taken over a higher priority item if a thread is in the middle of its search.

Q4

The graph below describes the execution time ratio of the Lock implementation vs. the Lock-free implementation (Ltime/LFtime). As you can see, the two implementations run a similar execution time, however the Lock implementation out-performs the lock-free implementation ever so slightly. This is likely due to the additional timeout created by the Lock-free implementation in the exchange algorithm.

Whenever a thread fails to normally get/put an item into the priority queue it attempts to make an exchange using the exchange algorithm from the LockFreeExchange class. A thread that runs the exchange algorithm when its state is EMPTY waits an allotted time for another thread to come in and complete the swap. If a thread does not enter within this set time then the waiting thread leaves the function. Therefore, there is an extra thread delay on top of the delay *d* that is incurred after every operation. If the timeout time is made to be too short, then an exchange is less likely to occur, and if its made too long, more swaps may occur however it will increase the run time.

It seems as though the lock-free techniques alone make modifying shared data faster than lock-based techniques, however in this implementation the extra delay from the exchanger increases the run time. Thus, the faster lock-free algorithm compensates for the exchanger delay and runs in similar time to the lock implementation.