Answers exercises 7

7.1)
7.1
The implementation only uses atomic integers and atomic references and uses Compare and Swap to avoid using locks. Its safe in the sense that the operation will conclude and the threads do not get into race conditions as they all perform atomic operations when handling shared data. The only non atomic operations are to set up what will be put in the shared data, and then we check if some other thread has changed the state that we expected to find. If some other thread has changed it, we do our whole operation again. Therefore its impossible to have an operation where another thread changes something unexpectedly midway through a method.

7.3
We didnt implement this method but not having locks is faster. The more threads you have you tend to go faster (until some point and not linearly) and the less locks the better as the overhead is just so much smaller.
7.2)
We implemented the required but we struggled to do the readerUnlock as we struggled to remove objects from a linked list. The writers lock works fine and we believe the implementation of readerLock to also work fine. We tested the writer lock amongst writers only but not the reader x writer scenario.

Answers exercises 8
Exercise 8.1
We differ on the first one only

Matheus's
1.
The execution is sequentially consistent as we can place the q.deq(x) after the whole execution of thread A. This way, the ordering within each thread is maintained and the invariant of the FIFO queue is maintained.

A —--|q.enq(x)| —---- —--|q.enq(y)| —----
B —--|q.deq(x)|

Laura's
1.
Not sequentially consistent.
Since in a sequentially consistent execution, the order of operations from different threads should appear consistent with the real-time order. In this execution, we have enq(x) followed by enq(y) in thread A, but q.deq(x) in thread B happens before the enqueueing of y in thread A, which violates the real-time order.

2.
Not linearizable.

Since it does not satisfy the real-time order, which is required for linearizability, a valid linearization cannot be provided.

3.
Linearizable.
<q.enq(x), q.deq(x)>

This linearization represents a scenario where the enqueue of x happens first, and then the dequeue of x follows. This is consistent with the standard specification of a sequential FIFO queue. Also per Hierly: if one method call precedes another in the original history, that ordering must be preserved in the linearization.

4.
Not linearizable.
Thread B dequeues y before it was enqueued in Thread A. This execution does not satisfy the standard specification of a sequential FIFO queue.

Exercise 8.2
1.
- Push has one linearization point:
 - P3 - if succesfully executed, the new element is inserted after the compareAndSet operation succeeds.

- Correctness
 - if there is two threafs executing push while reading the same top. A race condition occurs, where only one thread succeeds updating the top reference. The other fails and repeat the push.

- Poplinearization points:
 - P4 if stack is empty, after its execution, the evaluation of P5 is determined and whether the method will return null.
 - P8 -if succesfully executed, the old head of the stack is atomically replaced with the new head. This is after compareAndSet operation but before the new value is returned.

- Correctness
 - If two threads execute pop concurrently before the head is updated (P4 succeeds for both) and the stack is not empty (P5 returns null), then P8 succeeds for only one of them. The other restarts the pop.
 - If a thread executes pop after another thread updated the head, then P4 fails and it restarts the pop