Exercise 8.1

1. yes it is sequentially consistent: we can find a sequential execution that satisfies the standard specification of a sequential FIFO queue:
   < q.enq(x), q.deq(x), q.enq(y) >

2. no it is not linearizable: you cannot find a linearization point for enq(x) in thread A that happens before deq(x) in thread B

3. yes the execution is linearizable: we can define linearization point within each method call which map to sequential execution, for example
   < q.enq(x), q.deq(x) >

4. no it isn't: there is not linearization point such that the deque of x is before the deque of y.
   < q.enq(x), q.enq(y), q.deq(y) >
   it does not satisfy the FIFO requirement .

Exercise 8.2

1. Linearization points for push:
   - PUSH1 is the linearization point if during the creation of a new node the head has not changed (no other push or pop)

   Linearization points in pop:
   - POP1 is the linearization point if the queue is empty
   - POP2 is the point when a push is completed when queue is not empty and no other pushes or pops are performed during the creation of the new head

   Correctness:
   - If two threads execute push concurrently, then only one of them succeeds in executing PUSH1.
   - The other fails and repeats the push
   - If a thread executes push during another thread updated the head due to a pop, then PUSH1 fails and it repeats the push

   - If two threads execute pop concurrently, before the head is updated and the queue is not empty (POP1 fail), then POP2 succeeds for only one of them. The other restarts the pop
   - If the queue is empy then POP1 succeed
   - If a thread executes pop after another thread updated the head (due to a push/pop), then POP2 fail and it restarts the pop

2. See first method in file TestLockFreeStack.java for implementation.
3. See the other method in file TestLockFreeStack.java for implementation.

Exercise 8.3

1. **writerTryLock():** is wait-free: If a thread succeeds in making a call to writerTryLock(), it will succeed and get the lock if no other thread is currently holding the lock. Otherwise, it will return a false value without locking or waiting indefinitely. So it will return true or false in a finite number of step.

2. **writerUnlock():** This is wait-free, If the calling thread is the current owner of the lock, it will release the lock successfully, otherwise it will generate an exception. So in a finite number of steps it will either return true or throw an exception independently from what other threads do.

3. **readerTryLock():** This is lock-free, in any way if anyone tries to do a tryLock and fails this means that somebody else has the lock and is succeeding.

4. **readerUnlock():** This is lock-free, if a call to this method fails means someone else is succeeding.