1.
The current implementation only tracks the number of readers. To allow lock upgrades, it would have to track the identities of the individual reader threads. This would require a data structure like a hash table to store the thread identities along with their read lock count. The lock upgrade could only occur if the requesting thread is the only reader holding the lock. The implementation would have to verify this condition and ensure no other threads accquire a read lock during the upgrade process. Permitting lock upgrades might lead to writer starvation, where write lock requests are perpetually blocked by read lock upgrades. To avoid this situation, the implementation should condifer priority management to balance read and write access fairly. The upgrade process would need to be atomic and thread-safe to ensure consistent lock state transtitions.
2.
https://github.com/lvu5/454hw3
3.
The ineffective lazy method involves eliminating a node by simply setting its next field to null, which results in a loss of information, specifically the remainder of the list. To make this approach viable, we need to make two adjustments. Firstly, preserve a reference to the rest of the list after nullifying the removed node's next field. Secondly, use a unique sentinel node to indicate the list's end instead of a null pointer. However, this "correction" is futile, as we are not actually setting the next field to null but merely relocating it.

Furthermore, both the LockFreeList algorithm and the solution mentioned earlier would fail if we try to delete a node by setting its next field to null, particularly due to the compareAndSet() operation. To address this issue, we would have to lock the list's tail, but this would render the list non-lock-free.

4.
Indeed, holding the lock is crucial. We can assess if head.next is null without possessing the lock. If it is null, it's safe to raise an EmptyException. However, if it's not null, we cannot assume the queue remains non-empty once we obtain the lock.

5.
Base case: When the tree has a depth of d = 1, there is only one balancer. In this case, the balancer's output satisfies the step property by definition, as it divides the input tokens evenly between its two output wires.

Inductive step: We assume that the claim holds for trees of depth d - 1. A tree of depth d has a root balancer with left and right subtrees of depth d - 1. When there are m input values, the root balancer sends $\lceil m/2 \rceil$ values to the left subtree and $\lfloor m/2 \rfloor$ values to the right subtree. The left subtree's leaves correspond to the even-numbered outputs, while the right subtree's leaves correspond to the odd-numbered outputs.

We now consider two cases: m is even and m is odd.

a. If m is even, both subtrees have the same number of input values. Each subtree has a "step point" c, where the number of output tokens decreases by one. For the left subtree, this occurs at wire 2c, and for the right subtree, at wire 2c + 1.

b. If m is odd, the left subtree's step point occurs at wire 2c, while the right subtree's step point occurs at wire 2c - 1.

In both cases, the step property is preserved, as the output tokens are distributed in such a way that the first ⌈m/2⌉ outputs have one more token than the last ⌊m/2⌋ outputs.

Thus, by induction on the depth d of the tree, the step property is maintained for trees of any depth.