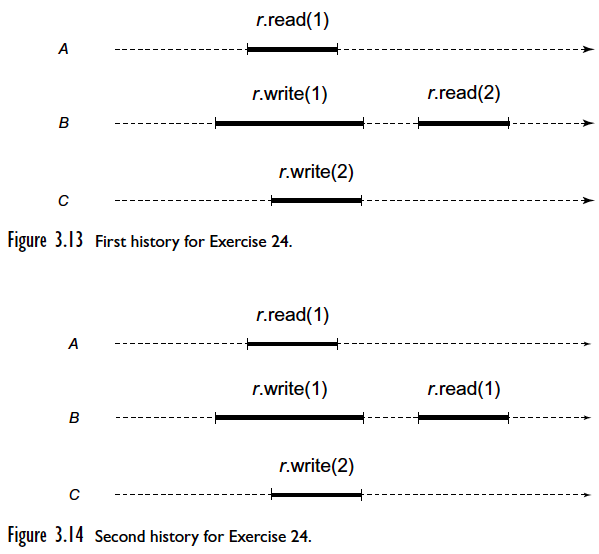
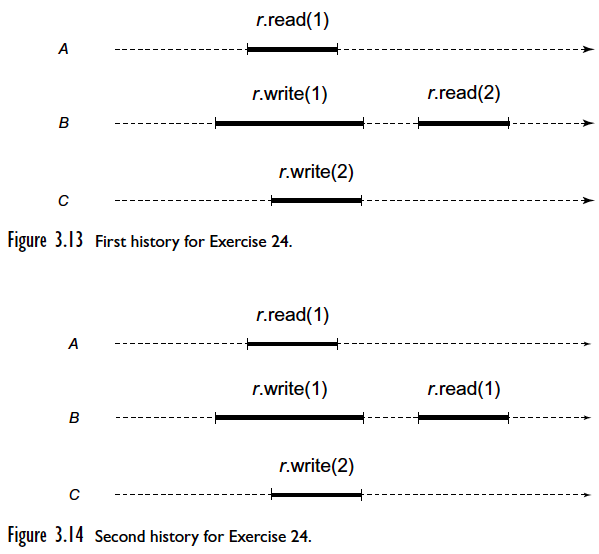
**Exercise 24.** For each of the histories shown in Figs. 3.13 and 3.14 are they quiescently consistent? Sequentially consistent? Linearizable? Justify your answer.



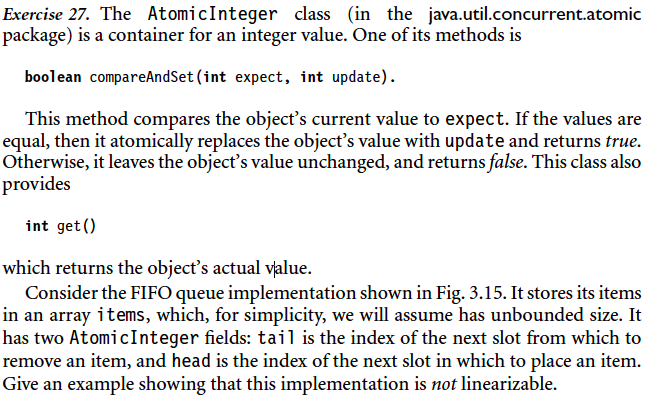
The following history is sequential consistent since a possible execution order could be: write(1), read(1), write(2), read(2). Additionally, the history is quiescently consistent requires method calls to appear in one-at-a-time sequential order and there are no pending invocations as seen in Figure 3.13. Finally, the history is linearizable since we can have a history write(1), read(1), write(2), read(2) such that the writes complete before the reads correctly return 1 then 2.

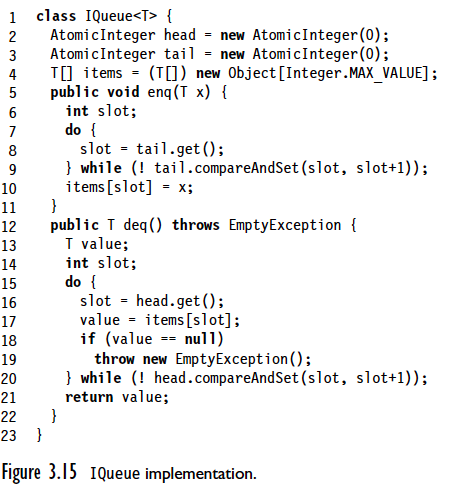


The following history



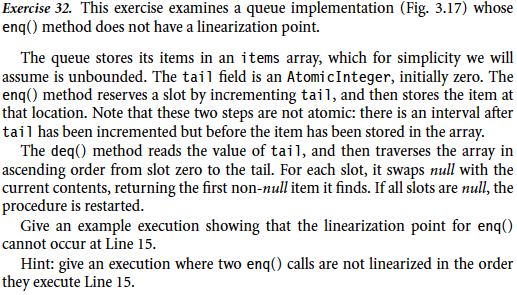
Condition L2 states that if a method call m0 precedes method call m1 in H, then the same is true in S where H is a history and S is a legal sequential history that is equivalent to the complete extension of H. Condition L2 is the quiescent property for linearization, which makes linearizability compositional. Condition L1 states a previous method call must have taken effect before a later method call and Sequential Consistency requires that method calls act as if they occurred in a sequential order consistent with program order. Condition L1 is essential the sequential consistency property for linearizability. Therefore, removing condition L2 from linearizabilty results in a property equivalent to sequential consistency.





Given threads 1 and 2, thread 1 enq(A), but stops executing before setting item[0] and never finishes. Thread 2 calls enqueue(B) and assigns item[1] to B since thread 1 was supposed to assign item[0] and the get() method returned 1. Next, Thread 2 calls dequeue() and attempts to remove A from item[0], but throws an empty exception because thread 1 never completed executing and did not set item[0] to A.

The example proves the implementation is not linearizable, since item[0] should contain a value, A, instead of throwing an empty exception. The implementation violates the condition that if one method call precedes another, then the earlier call must have taken effect before the later call.

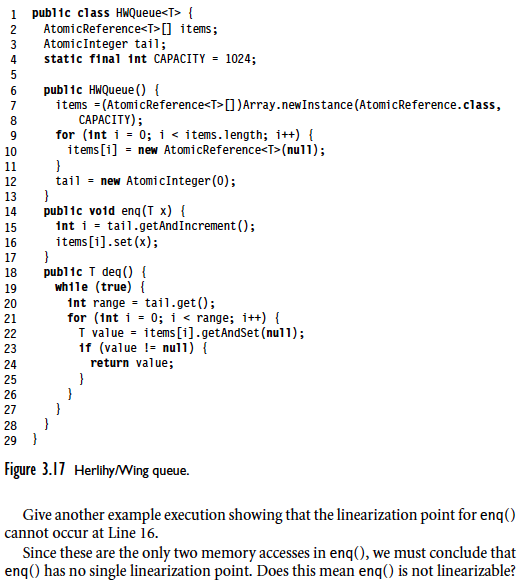


Proof by Counterexample:

Given threads A, B, C, thread A enqueues ‘a’ into the queue concurrently with thread B enqueueing ‘b’, then thread C dequeues.

1. Thread A executes tail.getAndIncrement() setting ‘i’ to 0 and tail to 1.
2. Thread B executes tail.getAndIncrement() setting ‘i’ to 1 and tail to 2.
3. Thread B sets items[1] to ‘b’.
4. Thread C dequeues at items[0] returning null.
5. Thread C moves to items[1] and dequeues returning ‘b’.
6. Thread A sets items[0] to ‘a’.
7. Thread C dequeues at items[0] returning ‘a’.

The possible history demonstrates a counterexample where ‘b’ is returned before ‘a’ when thread A executes line 15 before thread B, therefore line 15 is not a linearization point since it is not visible to other threads.



Given threads A, B, C, thread A executes concurrently with thread B, then thread C dequeues.

1. Thread A enqueues, executes tail.getAndIncrement(), so thread A gets items[0].
2. Thread B enqueues after thread A calling tail.getAndIncrement() and setting items[1] to ‘b’ before thread A sets items[0] to ‘a’.
3. Thread C dequeues at items[0] returning ‘a’.
4. Thread C dequeues again at items[1] returning ‘b’.

Line 16 is not a linearization point since thread B called items[i].set(x), line 16, before thread A even though thread A was first in the queue. Thread A called getAndIncrement before thread B, so it had items[0] even though it had not assigned ‘a’ to items[0] before thread B assigned items[1] to ‘b’.

The enq() method can be linearizable, but doesn’t have single linearization point in this case.