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.

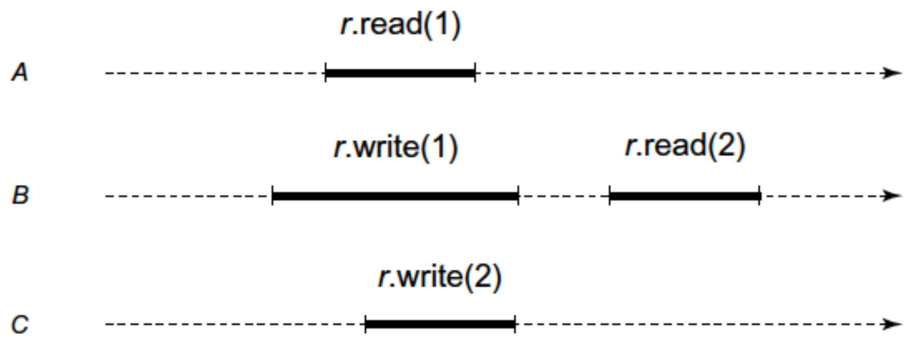

Figure 3.13 First history for Exercise 24.

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 the 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.

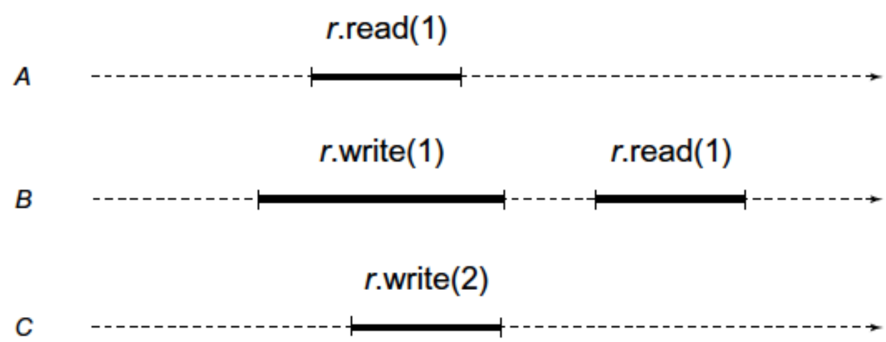

Figure 3.14 Second history for Exercise 24.

The following history is sequential consistent since a possible execution order could be: write(2), write(1), read(1), read(1). Additionally, the history is quiescently consistent since the method calls appear in a one-at-a-time sequential order and there are no pending invocations as seen in Figure 3.14. Finally, the history is linearizable since we can have a history write(2), write(1), read(1), read(1) such that the writes complete before the reads and the reads correctly return 1 twice.

Exercise 25. If we drop condition L2 from the linearizability definition, is the resulting property the same as sequential consistency? Explain.

Condition L2 states that if a method call m_0 precedes method call m_1 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.

Exercise 27. The `AtomicInteger` class (in the java.util.concurrent.atomic package) is a container for an integer value. One of its methods is

```
boolean compareAndSet(int expect, int update).
```

This method compares the object's current value to `expect`. If the values are equal, then it atomically replaces the object's value with `update` and returns *true*. Otherwise, it leaves the object's value unchanged, and returns *false*. This class also provides

```
int get()
```

which returns the object's actual value.

Consider the FIFO queue implementation shown in Fig. 3.15. It stores its items in an array `items`, which, for simplicity, we will assume has unbounded size. It has two `AtomicInteger` fields: `tail` is the index of the next slot from which to remove an item, and `head` is the index of the next slot in which to place an item. Give an example showing that this implementation is *not* linearizable.

```
class IQueue<T> {
  AtomicInteger head = new AtomicInteger(0);
  AtomicInteger tail = new AtomicInteger(0);
  T[] items = (T[]) new Object[Integer.MAX_VALUE];
  public void enq(T x) {
    int slot;
    do {
      slot = tail.get();
    } while (! tail.compareAndSet(slot, slot+1));
    items[slot] = x;
  }
  public T deq() throws EmptyException {
    T value;
    int slot;
    do {
      slot = head.get();
      value = items[slot];
      if (value == null)
        throw new EmptyException();
    } while (! head.compareAndSet(slot, slot+1));
    return value;
  }
}
```

Figure 3.15 `IQueue` implementation.

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.

Exercise 32. This exercise examines a queue implementation (Fig. 3.17) whose enq() method does not have a linearization point.

The queue stores its items in an `items` array, which for simplicity we will assume is unbounded. The `tail` field is an `AtomicInteger`, initially zero. The enq() method reserves a slot by incrementing `tail`, and then stores the item at that location. Note that these two steps are not atomic: there is an interval after `tail` has been incremented but before the item has been stored in the array.

The deq() method reads the value of `tail`, and then traverses the array in ascending order from slot zero to the tail. For each slot, it swaps *null* with the current contents, returning the first non-*null* item it finds. If all slots are *null*, the procedure is restarted.

Give an example execution showing that the linearization point for enq() cannot occur at Line 15.

Hint: give an execution where two enq() calls are not linearized in the order they execute Line 15.

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.

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail;
4    static final int CAPACITY = 1024;
5
6    public HWQueue() {
7      items =(AtomicReference<T>[])Array.newInstance(AtomicReference.class,
8          CAPACITY);
9      for (int i = 0; i < items.length; i++) {
10       items[i] = new AtomicReference<T>(null);
11     }
12     tail = new AtomicInteger(0);
13   }
14   public void enq(T x) {
15     int i = tail.getAndIncrement();
16     items[i].set(x);
17   }
18   public T deq() {
19     while (true) {
20       int range = tail.get();
21       for (int i = 0; i < range; i++) {
22         T value = items[i].getAndSet(null);
23         if (value != null) {
24           return value;
25         }
26       }
27     }
28   }
29 }
```

Figure 3.17 Herlihy/Wing queue.

Give another example execution showing that the linearization point for enq() cannot occur at Line 16.

Since these are the only two memory accesses in enq(), we must conclude that enq() has no single linearization point. Does this mean enq() is not linearizable?

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.