Problem 1. The AtomicInteger class is a container for an integer value. This class provides
\begin{verbatim}
boolean compareAndSet(int expect, int update).
\end{verbatim}
This method compares the object’s current value to \texttt{expect}. If the values are equal, then it atomically replaces the object’s value with \texttt{update} and returns \texttt{true}. Otherwise, it leaves the object’s value unchanged, and returns \texttt{false}. This class also provides
\begin{verbatim}
int get()
\end{verbatim}
which returns the object’s actual value.

Consider the FIFO queue implementation shown in Fig. 1. It stores its items in an array \texttt{items}, which, for simplicity, we will assume has unbounded size. It has two AtomicInteger fields: \texttt{tail} is the index of the next slot from which to remove an item, and \texttt{head} is the index of the next slot in which to place an item. Give an example showing that this implementation is not linearizable.

\begin{figure}[h]
\centering
\begin{verbatim}
class MyQueue<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;
    }
}
\end{verbatim}
\caption{MyQueue implementation.}
\end{figure}
public class HWQueue<T> {
    AtomicReference<T>[] items;
    AtomicInteger tail;
    static final int CAPACITY = 1024;

    public HWQueue() {
        items = (AtomicReference<T>[]) Array.newInstance(AtomicReference.class, CAPACITY);
        for (int i = 0; i < items.length; i++) {
            items[i] = new AtomicReference<T>(null);
        }
        tail = new AtomicInteger(0);
    }

    public void enq(T x) {
        int i = tail.getAndIncrement();
        items[i].set(x);
    }

    public T deq() {
        while (true) {
            int range = tail.get();
            for (int i = 0; i < range; i++) {
                T value = items[i].getAndSet(null);
                if (value != null) {
                    return value;
                }
            }
        }
    }
}

Figure 2: Herlihy/Wing queue.

Problem 2. This exercise examines the queue implementation shown in Fig. 2. 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.

Part 1: True or False? There are only two memory references in enq(), at Line 15 and at Line 16. If we can show that neither one is always the linearization point for enq(), then the enq() method is not linearizable. Justify your answer in one short sentence.

Part 2: True or False? Line 15 is the linearization point for enq(). If not, give an execution where two calls are not executed in the order they execute that line.

Part 3: True or False? Line 16 is the linearization point for enq(). If not, give an execution where two calls are not executed in the order they execute that line.
Problem 3. For each of the histories shown, are they sequentially consistent? Linearizable? Justify your answer.