Problem 1. The AtomicInteger class (in the java.util.concurrent.atomic package) is a container for an integer value. One of its methods is

\[
\text{boolean compareAndSet(int expect, int update)}.\]

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

\[
\text{int get()}\]

which returns the object’s actual value.

Consider the FIFO queue implementation shown in Fig. 1. 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.

```java
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 1: IQueue implementation.
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] = new AtomicReference<T>(null);
    }

    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 a queue implementation (Fig. 2) 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.

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. Explain why this this does mean enq() is not linearizable.
Problem 3. Prove that sequential consistency is nonblocking.
Problem 4. For this exercise you will create a Java program that will spawn several threads to concurrently increment a shared counter a million times in an unsynchronized manner. More specifically your program will have each thread concurrently increment an integer a hundred-thousand times for a grand total of one million increments.

The format of your handin will consist of three files in a directory: Counter.java, TestCounter.java, and a README.

1. In Counter.java create a Counter class that implements Runnable. Have the class initialize a counter value to 0, provide a public run() method that increments the counter value a hundred-thousand times, and a provide a public getValue() method that returns the current value of the counter.

2. In TestCounter.java create a TestCounter class that will house the main() method. This method should then create a Counter object, spawn ten threads, have each thread increment the counter a hundred-thousand times, and then print out the final value of the Counter once all threads finish.

3. In your README, proceed to describe the output of your program. Did it print the expected grand total of a million? If not, provide your reasoning as to why the discrepancy occurs. Please note that as you do this problem we expect neither your answer for your README nor your code is expected to be very long.

When you ready to turn in your assignment, navigate to the directory containing your solution and run cs1760_handin counter to hand in.