Concurrent Systems and Applications 2004
Paper 4 Question 8 (TLH)

This question is examining material from the 'Concurrent systems' part of the course, in particular the lectures titled 'Mutual exclusion', 'Condition synchronization' and 'Worked examples'.

(a) A multi-threaded application is using a long linked list of integers. The list is accessed through `synchronized` methods on a `ListSet` object.

The list itself comprises a chain of `ListNode` objects in ascending numerical order. The chain always starts and ends with special *sentinel* nodes conceptually containing $-\infty$ and $+\infty$ respectively. This simplifies the implementation of operations on the list: they do not have to deal with inserting elements at the very start or at the very end.

Sketch the definition of `ListSet` and `ListNode` as Java classes. You need only give appropriate field definitions and the implementation of an `insert` method on `ListSet`. [4 marks]

```
class ListNode {
  int value;
  ListNode next;
}

class ListSet {
  ListNode first;

  synchronized void insert(int value) {
    ListNode nn = new ListNode(value);
    ListNode at = first;
    while (value > at.next.value) {
      at = at.next;
    }
    at.next = nn;
  }
}
```

(b) An engineer suggests that, instead of holding a lock on a `ListSet` object, threads only need to lock a pair of `ListNode` objects in the region that they are working.

(i) Define methods `lock` and `unlock` for your `ListNode` class to allow a thread to acquire a mutual exclusion lock on a given node. [6 marks]

```
boolean locked = false;

synchronized void lock() throws InterruptedException {
  while (locked) wait();
  locked = true;
}

synchronized void unlock() {
  locked = false;
  notifyAll();
}
```

(*ii*) Show how your insert method could be updated to incorporate the engineer's idea. [6 marks]

```
void insert(int value) throws InterruptedException {
  ListNode nn = new ListNode(value);
  ListNode at = first;
  at.lock();
  at.next.lock();
  while (value > at.next.value) {
    at.unlock();
    at = at.next;
    at.next.lock();
  }
  at.next = nn;
  at.unlock();
  at.next.unlock();
}
```

A particularly good answer would deal with interruption more carefully and release any locks held.

(*iii*) Do you think the new implementation will be faster than the original one? Justify your answer. [2 marks]

The new implementation removes the need for mutual exclusion on the entire list – this may increase the amount of concurrency available. However, the vast number of `lock` and `unlock` operations is likely to negate these benefits. Further problems such as *lock convoying* exist, but that is beyond the scope of this course.