```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[])new Object[capacity];
  }
  public void enq(T x) throws FullException {
    lock.lock();
    try {
      if (tail - head == items.length)
        throw new FullException();
      items[tail % items.length] = x;
      tail++;
    } finally {
      lock.unlock();
    }
  }
  public T deq() throws EmptyException {
    lock.lock();
    try {
      if (tail == head)
        throw new EmptyException();
      T x = items[head % items.length];
      head++;
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

FIGURE 3.1 A lock-based FIFO queue. The queue's items are kept in an array items, where head is the index of the next item (if any) to dequeue, and tail is the index of the first open array slot (modulo the capacity). The lock field contains a lock that ensures that methods are mutually exclusive. Initially head and tail are zero, and the queue is empty. If enq() finds the queue is full (i.e., head and tail differ by the queue capacity), then it throws an exception. Otherwise, there is room, so enq() stores the item at array entry for tail, and then increments tail. The deq() method works in a symmetric way.