# 4.2 Lecture Summary

## 4.2 Concurrent Queues

**Lecture Summary:** In this lecture, we studied *concurrent queues*, an extension of the popular queue data structure to support concurrent accesses. The most common operations on a queue are *enq(x)*, which enqueues object *x* at the end (tail) of the queue, and *deq()* which removes and returns the item at the start (head) of the queue. A correct implementation of a concurrent queue must ensure that calls to *enq()* and *deq()* maintain the correct semantics, even if the calls are invoked concurrently from different threads. While it is always possible to use locks, isolation, or actors to obtain correct but less efficient implementations of a concurrent queue, this lecture illustrated how an expert might implement a more efficient concurrent queue using the optimistic concurrency pattern.

A common approach for such an implementation is to replace an object reference like *tail* by an *AtomicReference*. Since the *compareAndSet()* method can also be invoked on *AtomicReference* objects, we can use it to support (for example) concurrent calls to *enq()* by identifying which calls to *compareAndSet()* succeeded, and repeating the calls that failed. This provides the basic recipe for more efficient implementations of *enq()* and *deq()*, as are typically developed by concurrency experts. A popular implementation of concurrent queues available in Java is *java.util.concurent.ConcurrentLinkedQueue.*

## Optional Reading:

1. Documentation on Java's AtomicReference class

2. Documentation on Java's ConcurrentLinkedQueue class

Mark as completed