Concurrent Queues

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

The Five-Fold Path

- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization
- Lazy synchronization
- Lock-free synchronization

Another Fundamental Problem

- We told you about
  - Sets implemented using linked lists
- Next: queues
- Next: stacks

Queues & Stacks

- Both: pool of items
- Queue
  - enq() & deq()
  - First-in-first-out (FIFO) order
- Stack
  - push() & pop()
  - Last-in-first-out (LIFO) order

Bounded vs Unbounded

- Bounded
  - Fixed capacity
  - Good when resources an issue
- Unbounded
  - Holds any number of objects

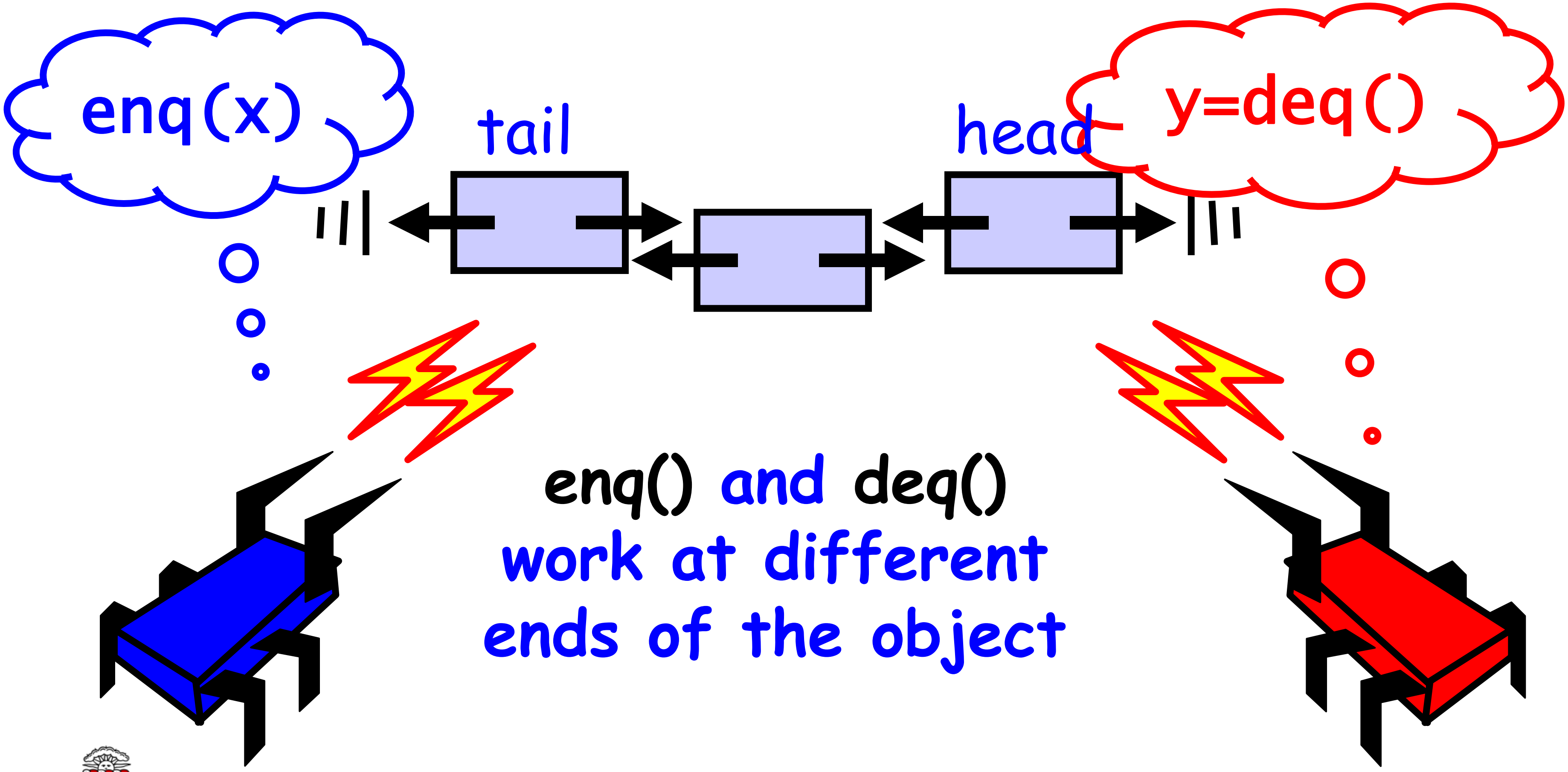
Blocking vs Non-Blocking

- Problem cases:
  - Removing from empty pool
  - Adding to full (bounded) pool
- Blocking
  - Caller waits until state changes
- Non-Blocking
  - Method throws exception

This Lecture

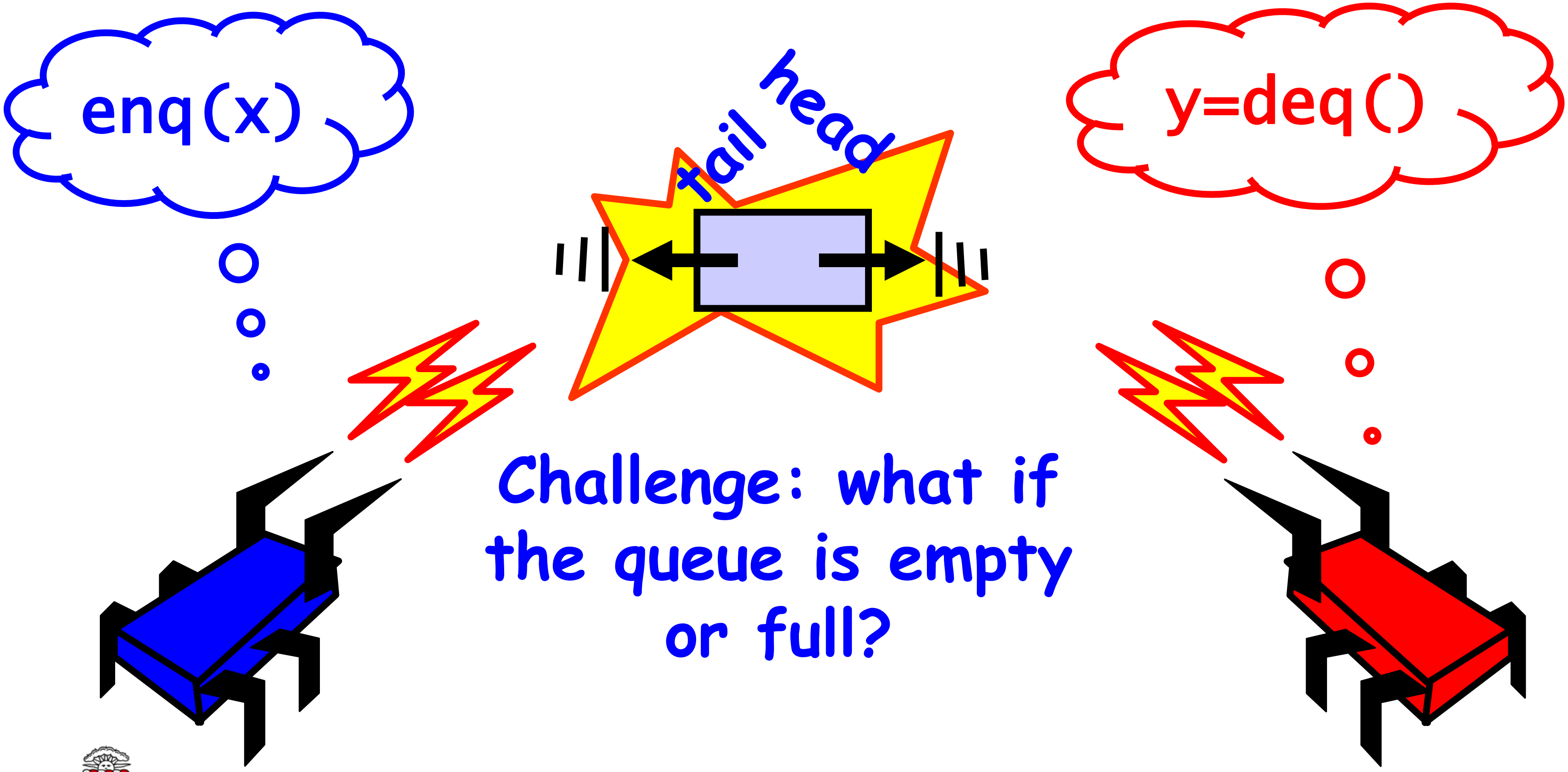
- Bounded, Blocking, Lock-based Queue
- Unbounded, Non-Blocking, Lock-free Queue
- ABA problem
- Unbounded Non-Blocking Lock-free Stack
- Elimination-Backoff Stack

Queue: Concurrency



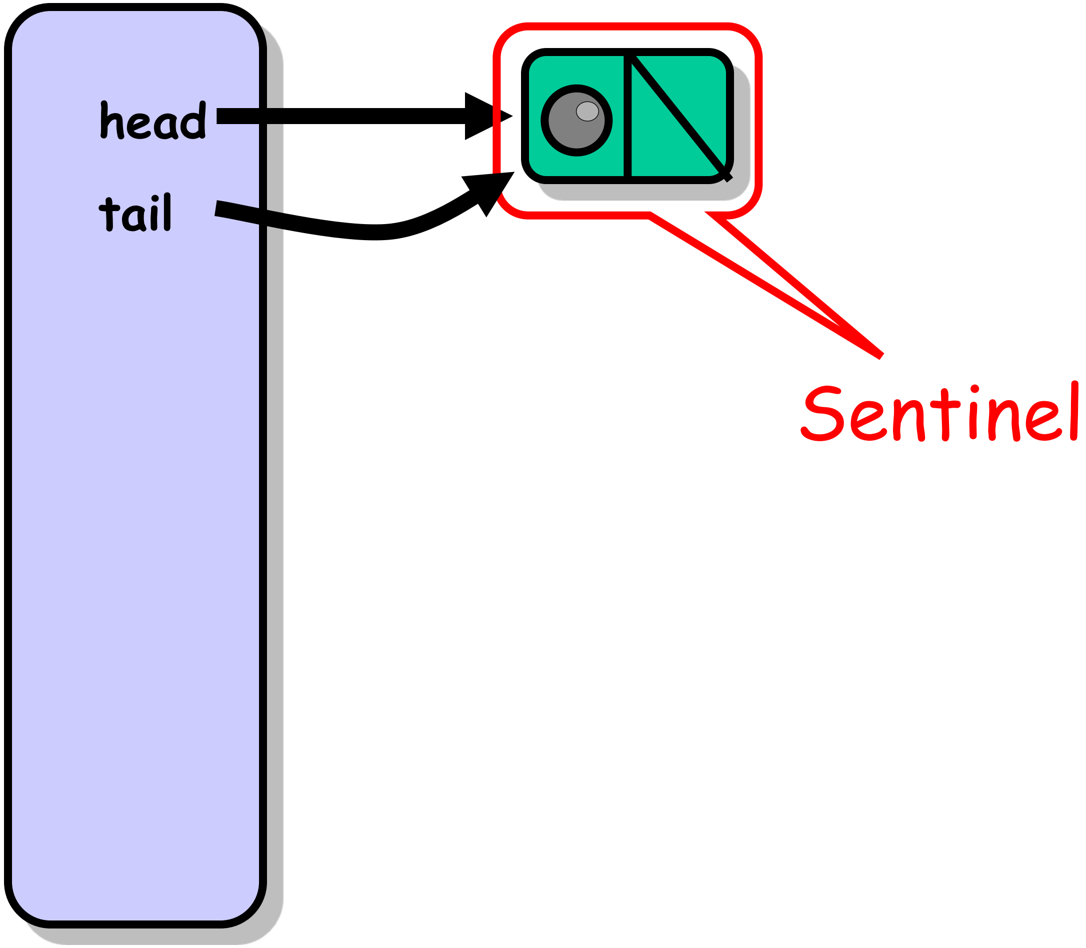
enq() and deq() work at different ends of the object

Concurrency

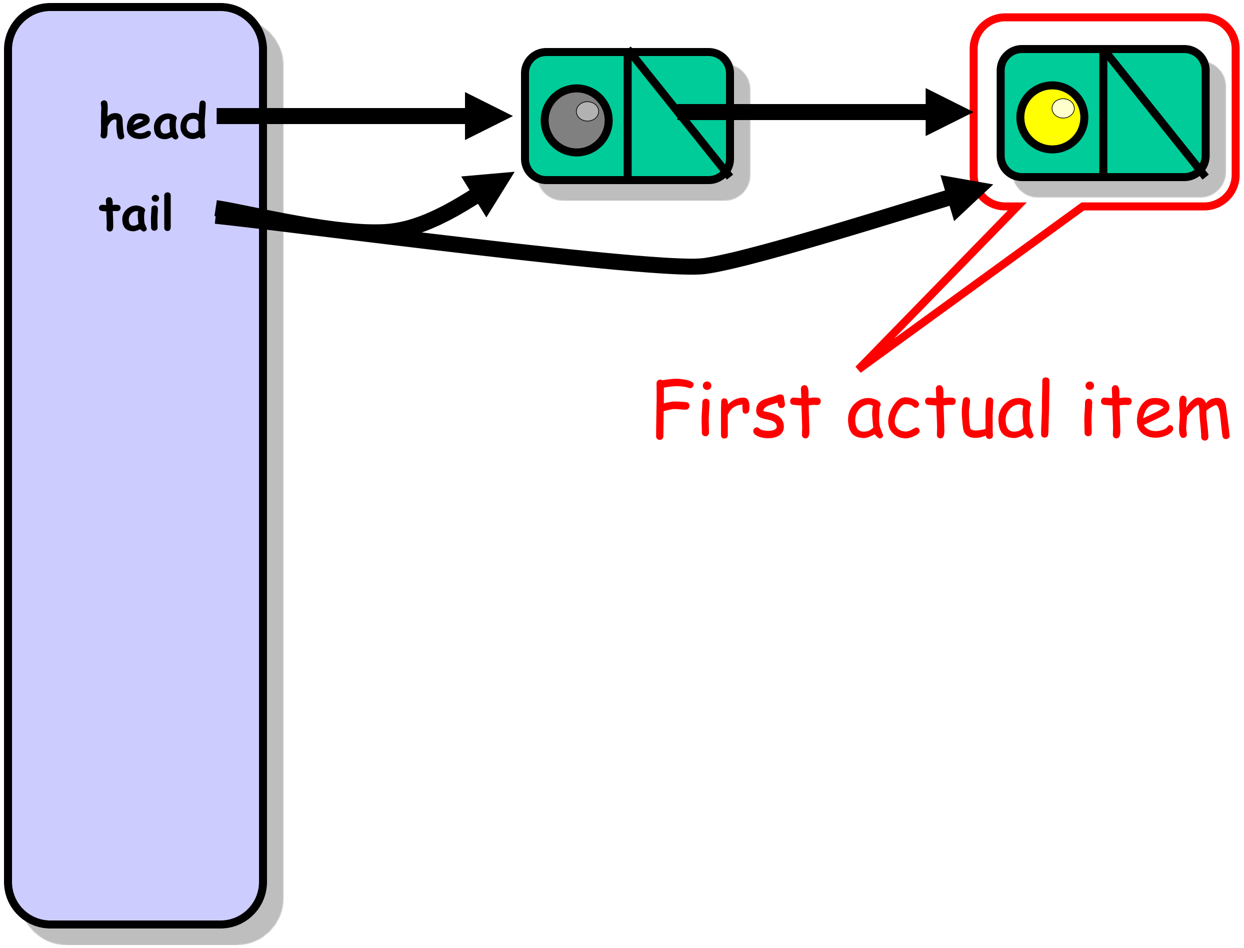


Challenge: what if the queue is empty or full?

Bounded Queue

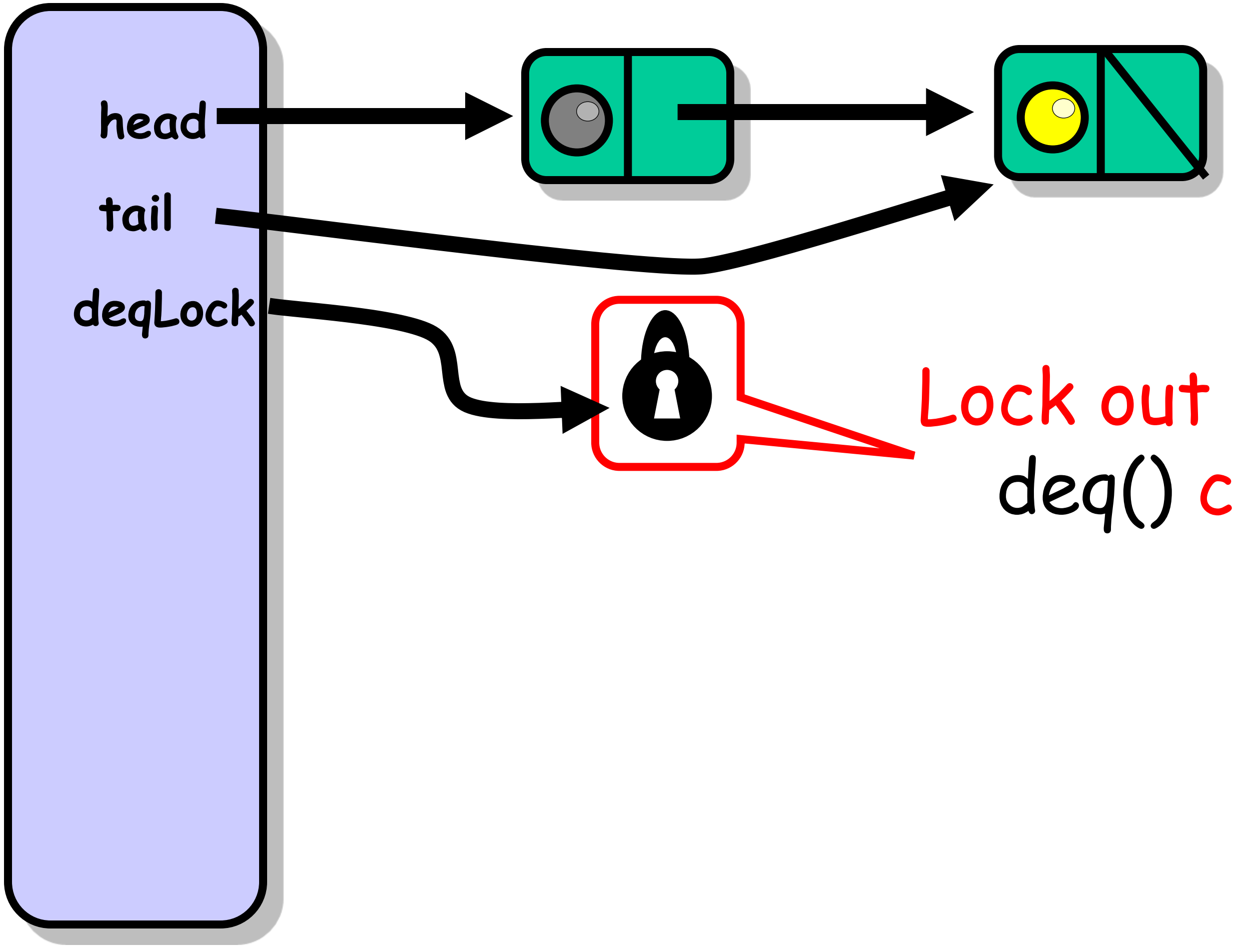


Bounded Queue



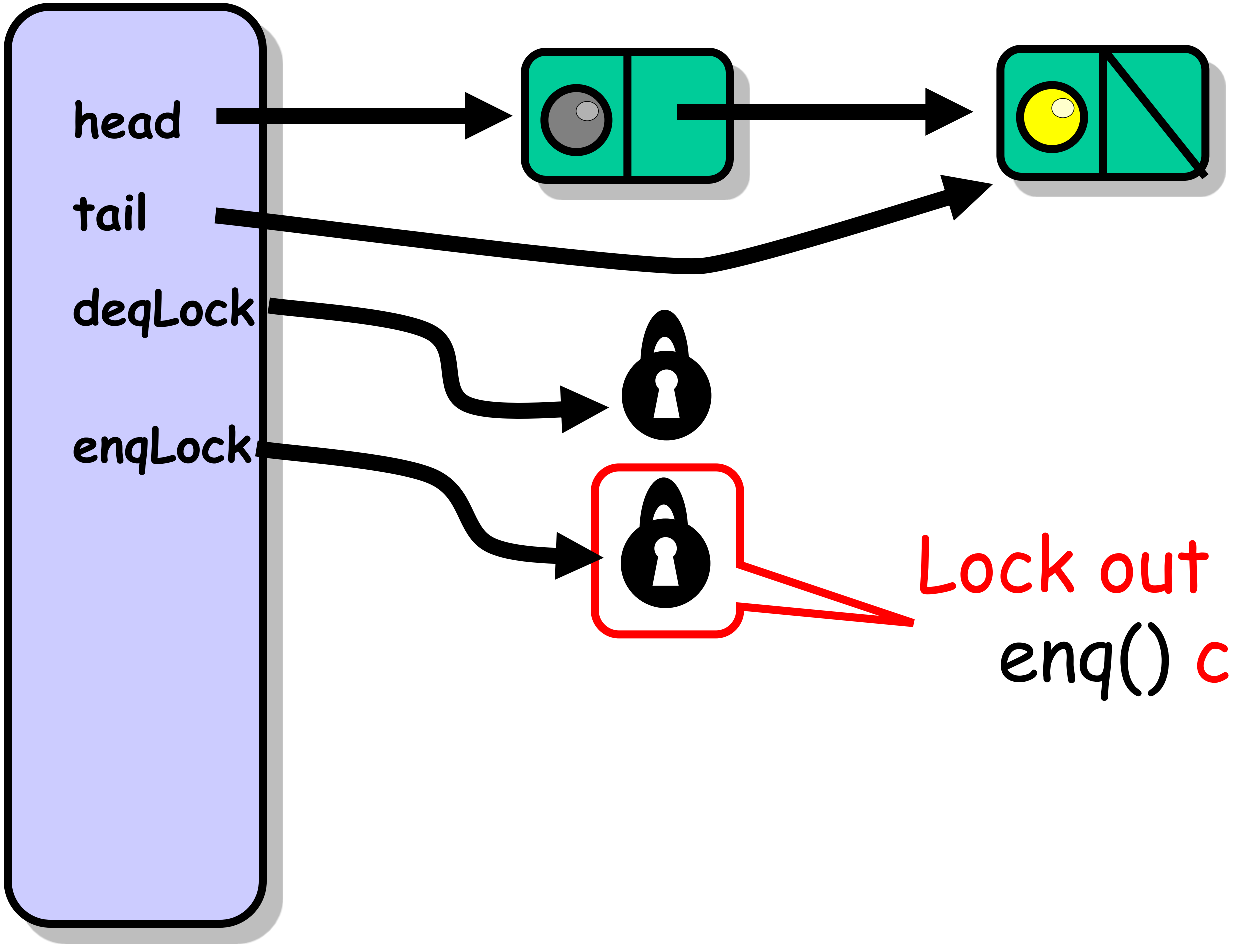
First actual item

Bounded Queue



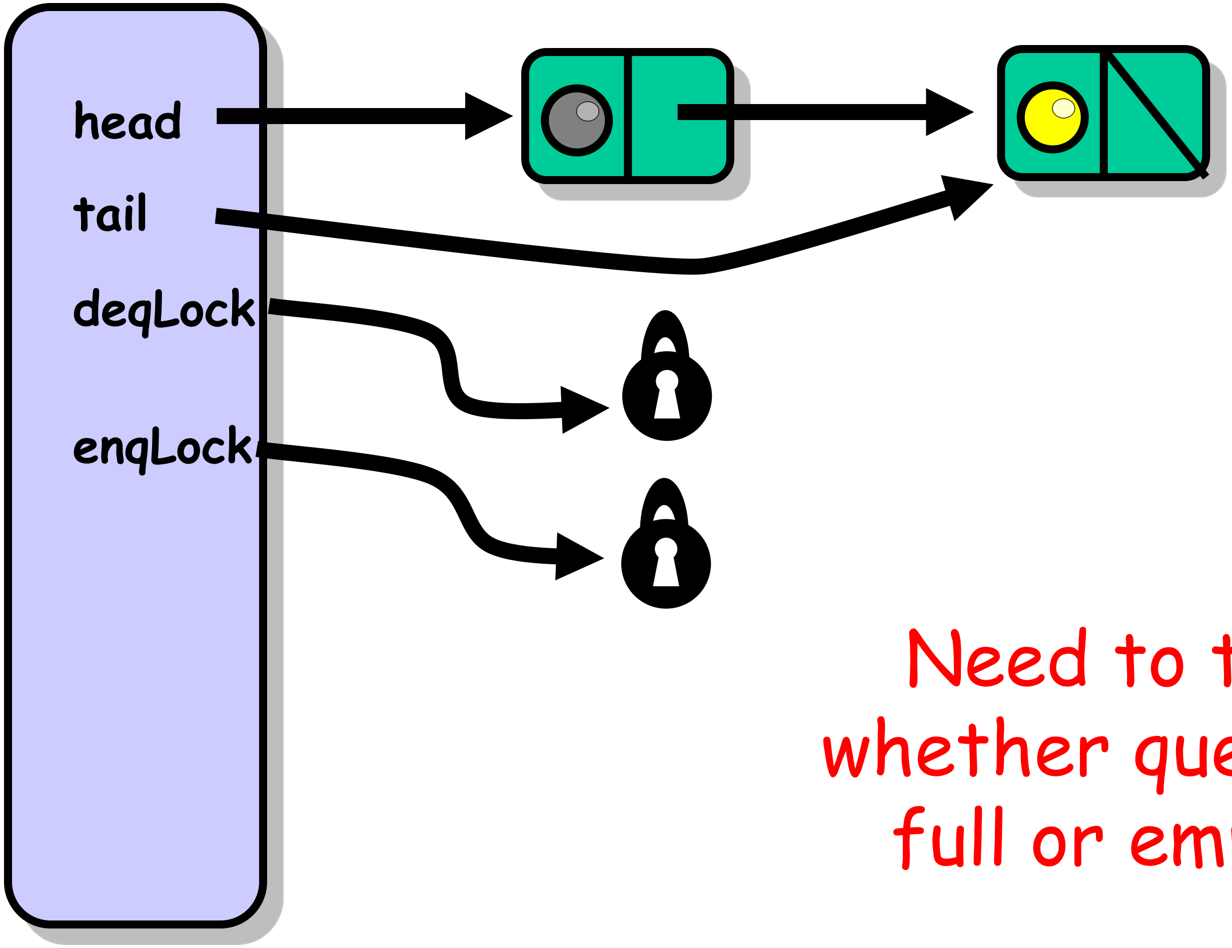
Lock out other deq() calls

Bounded Queue



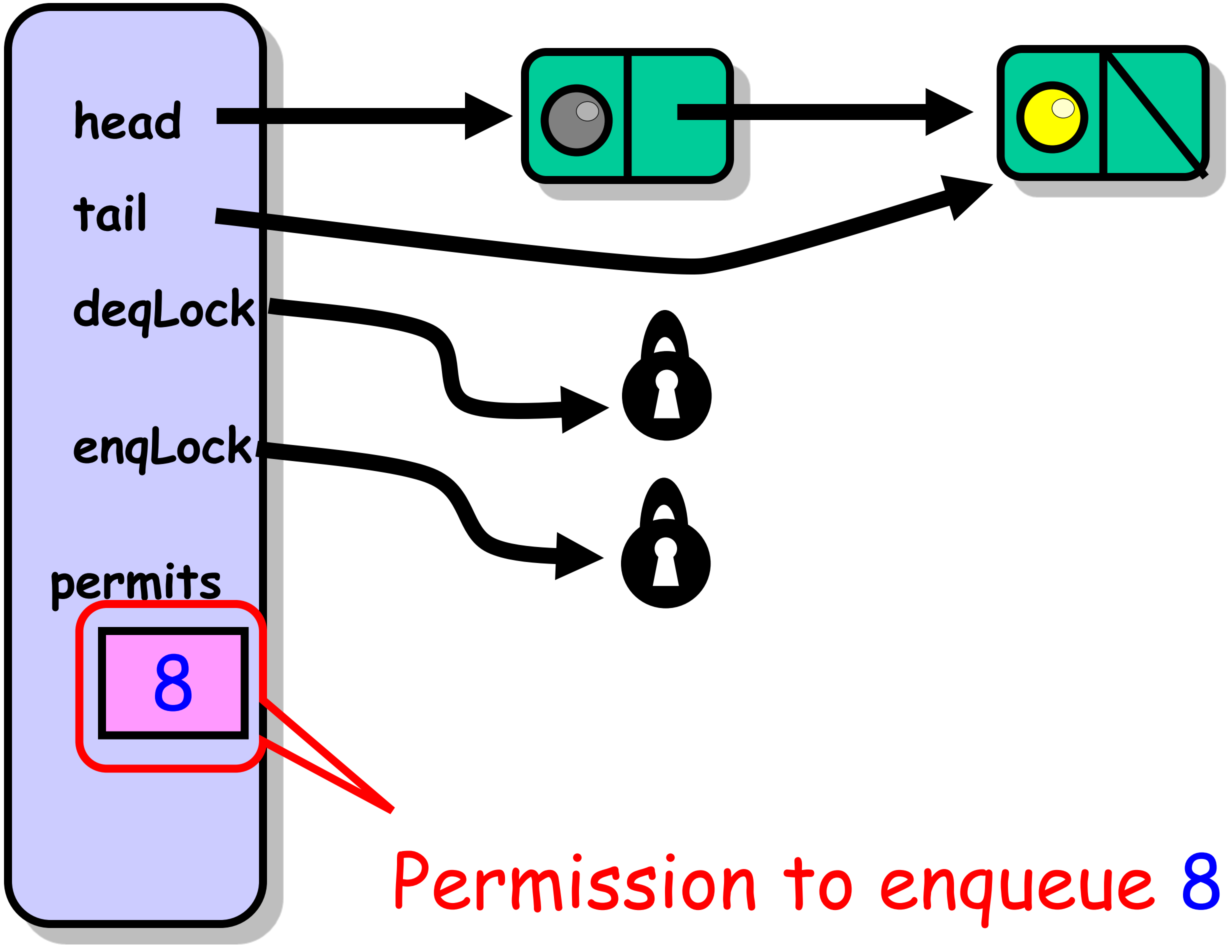
Lock out other enq() calls

Not Done Yet



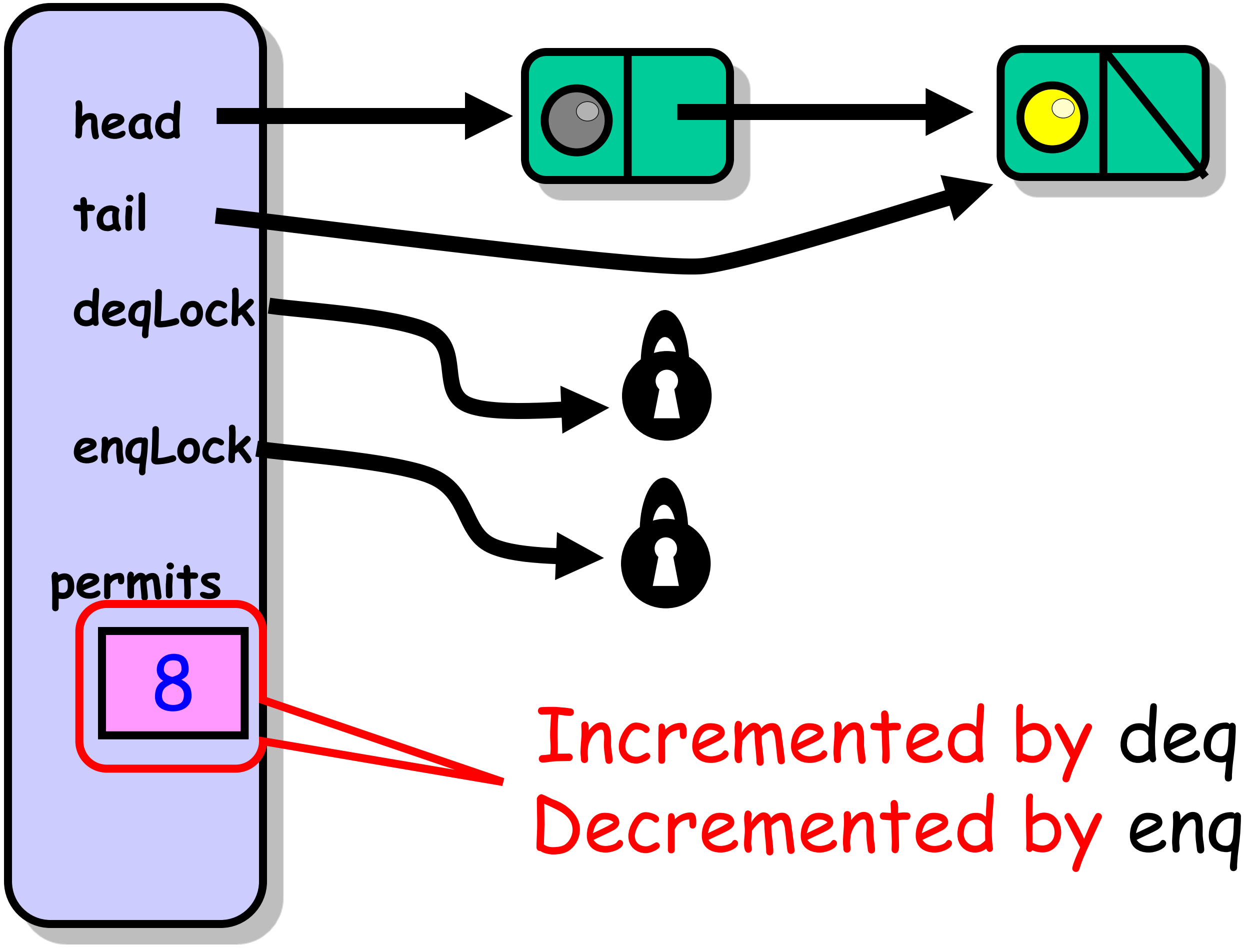
Need to tell whether queue is full or empty

Not Done Yet



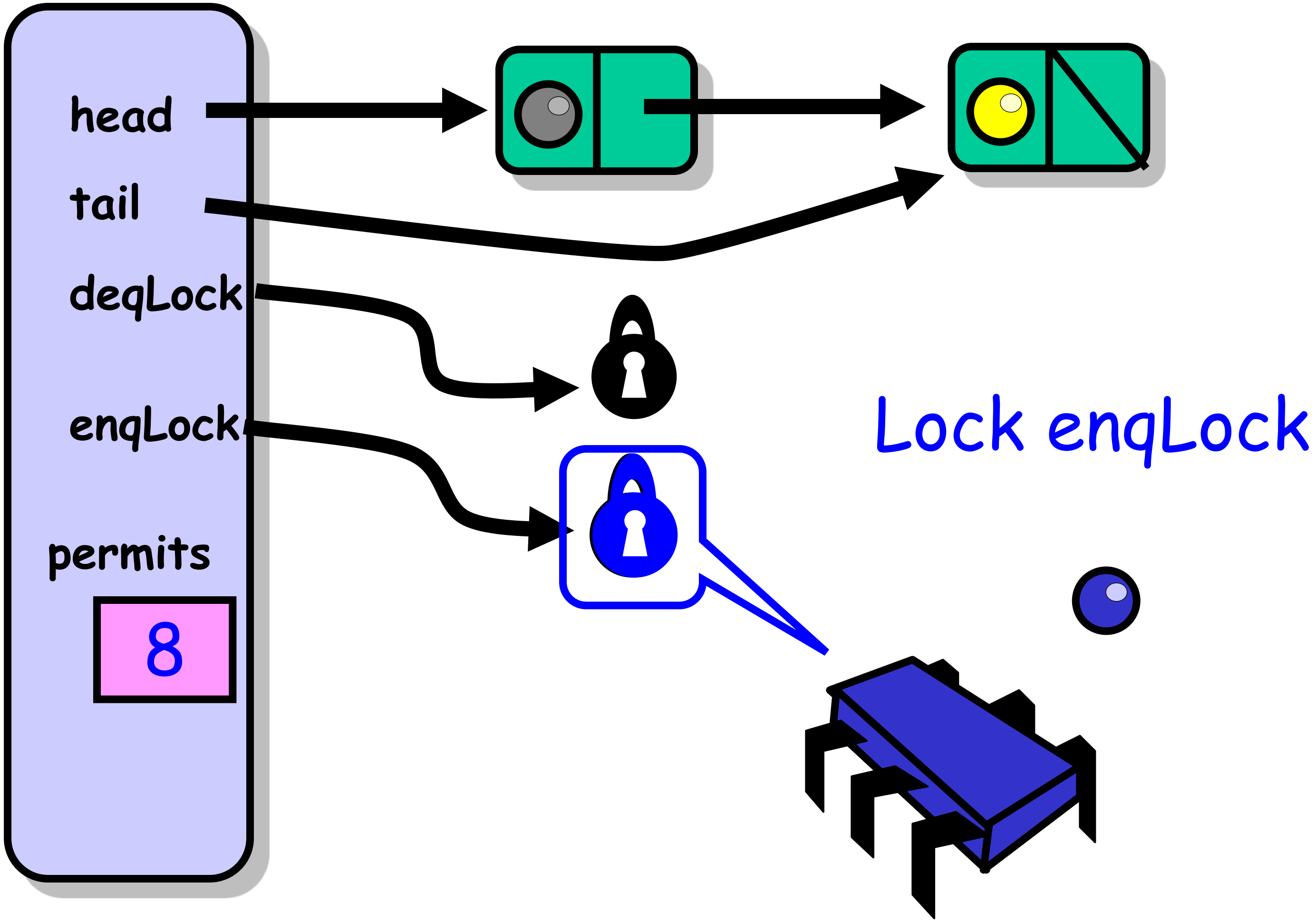
Permission to enqueue 8 items

Not Done Yet



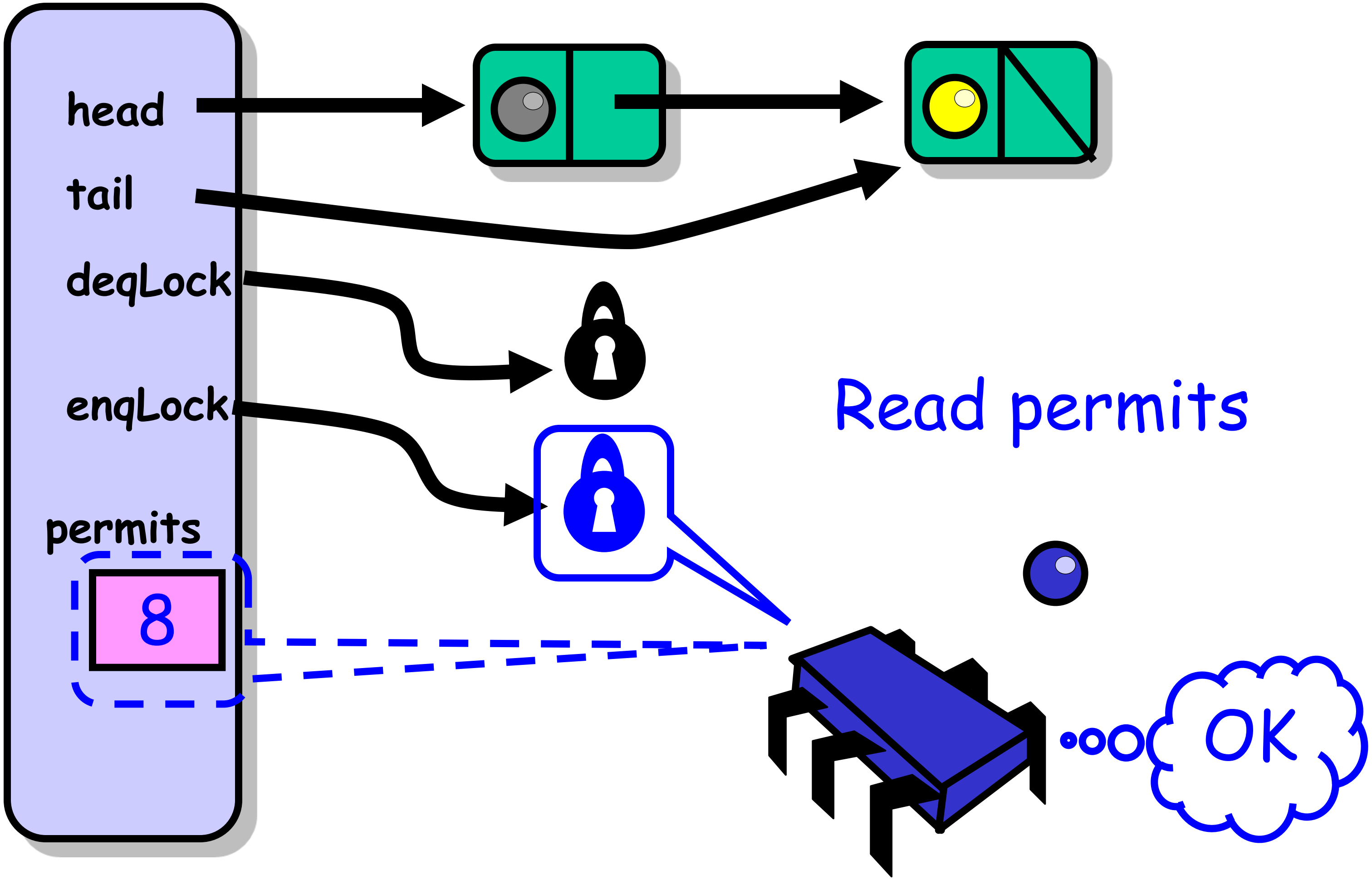
Incremented by deq()
Decremented by enq()

Enqueuer

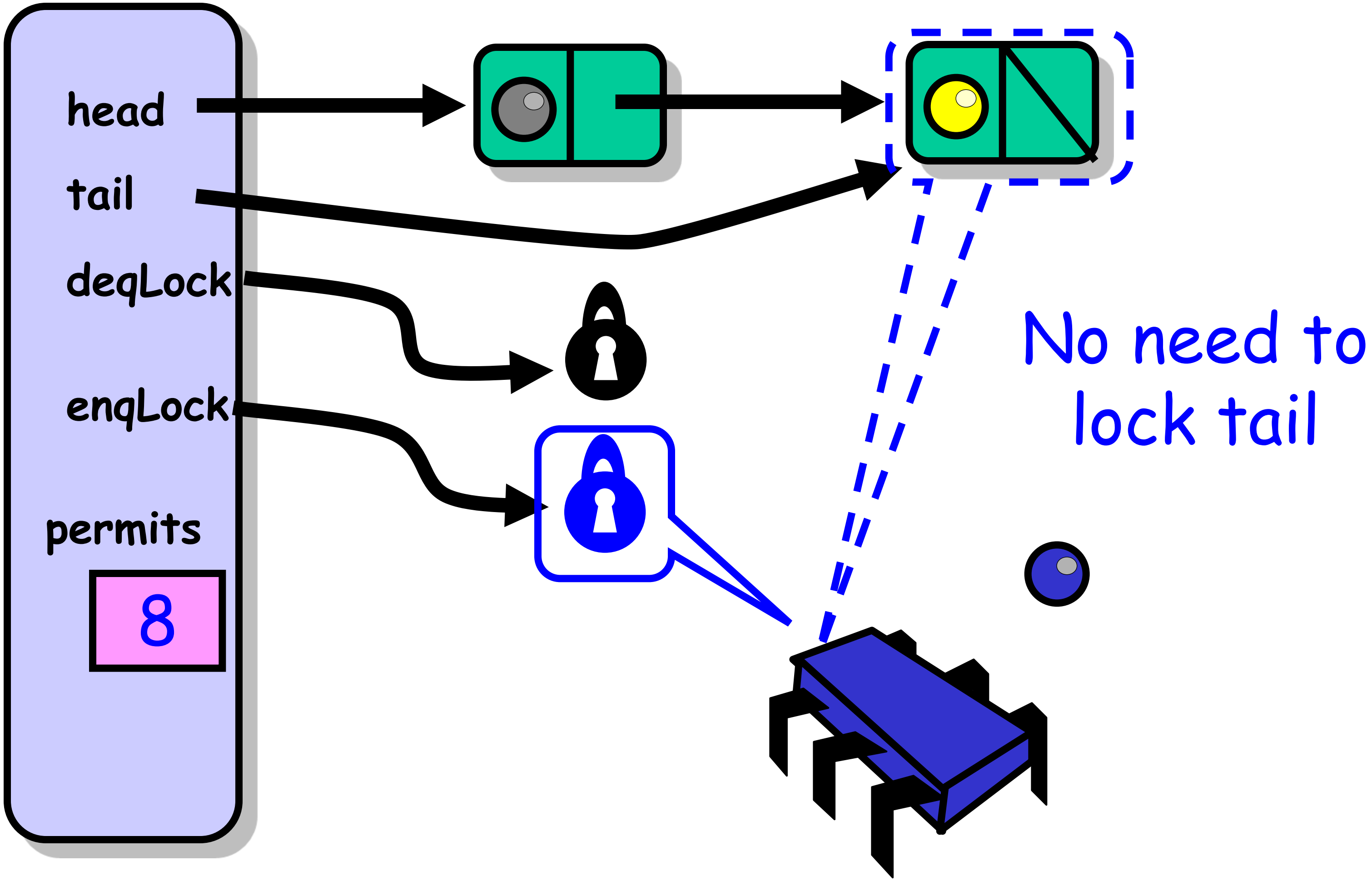


Lock enqLock

Enqueuer

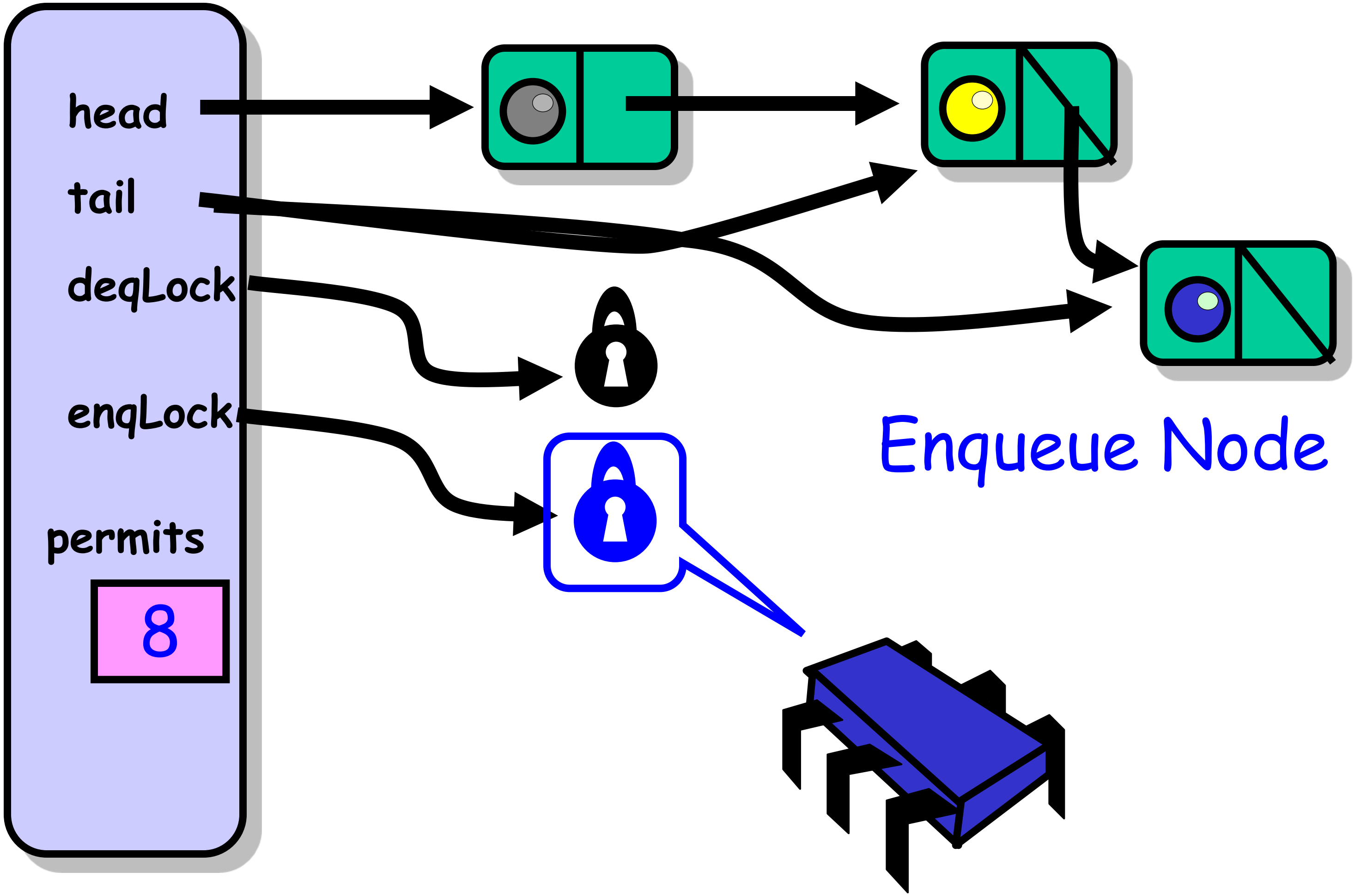


Enqueuer

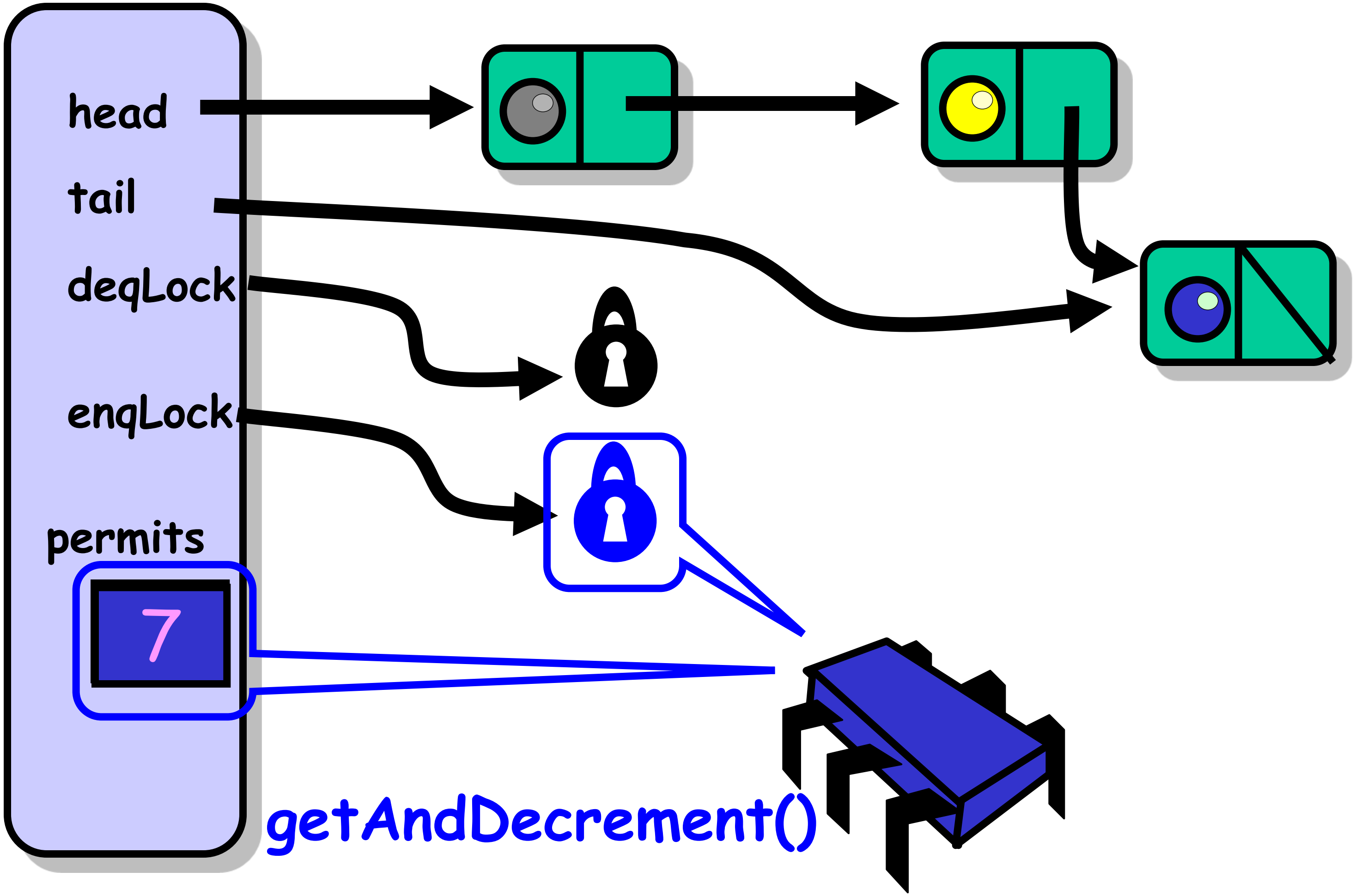


No need to lock tail

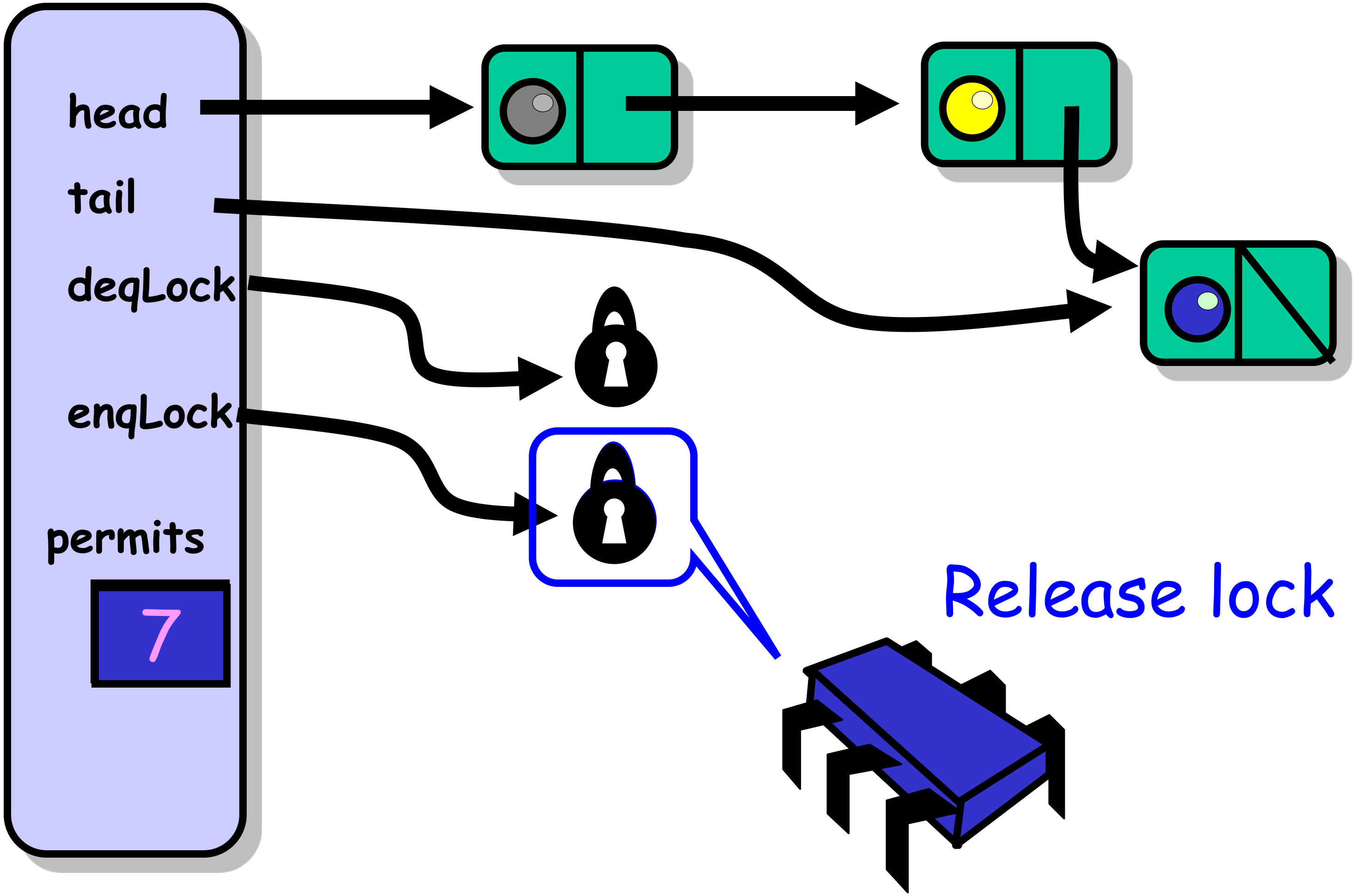
Enqueuer



Enqueuer

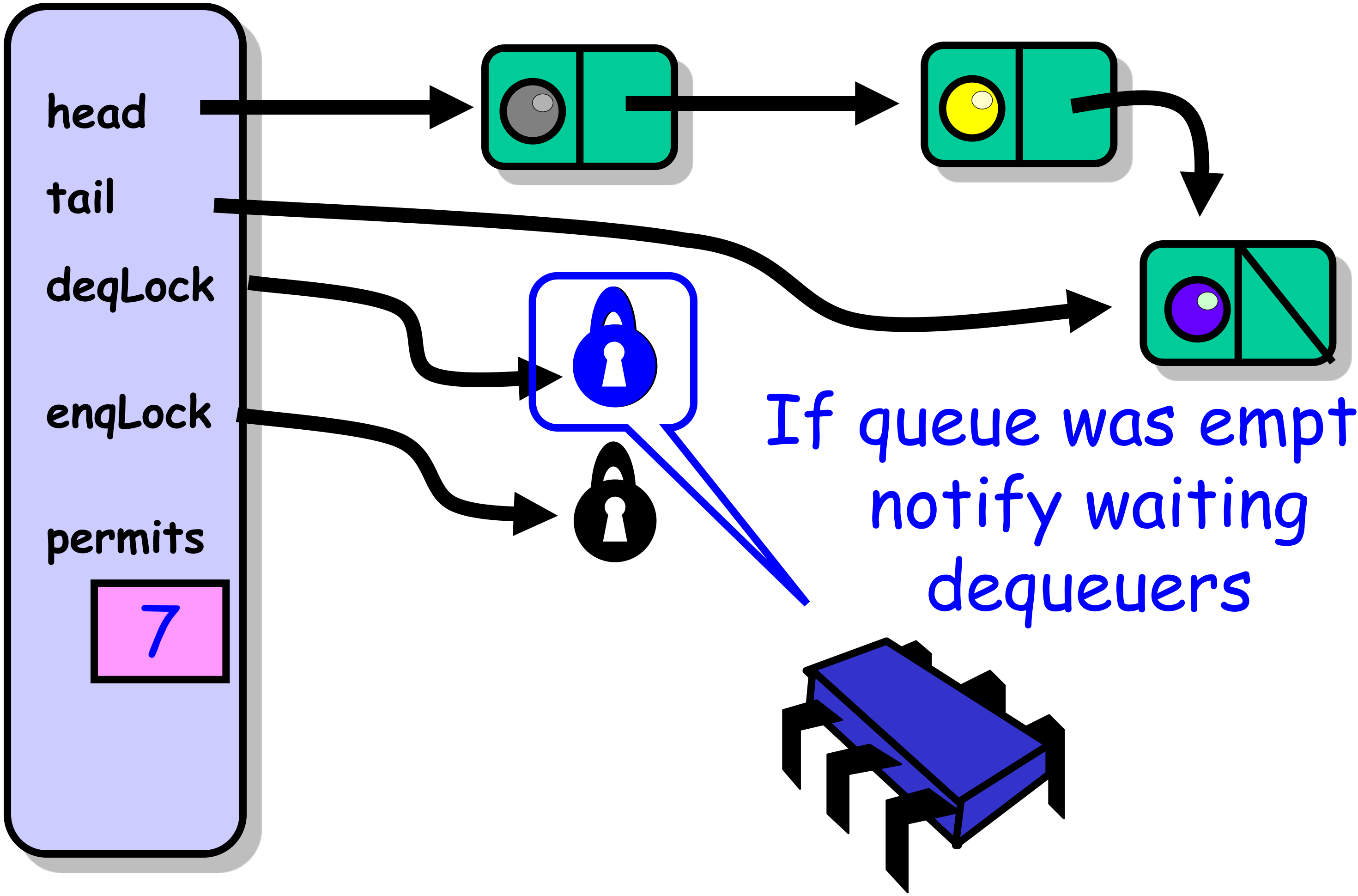


Enqueuer



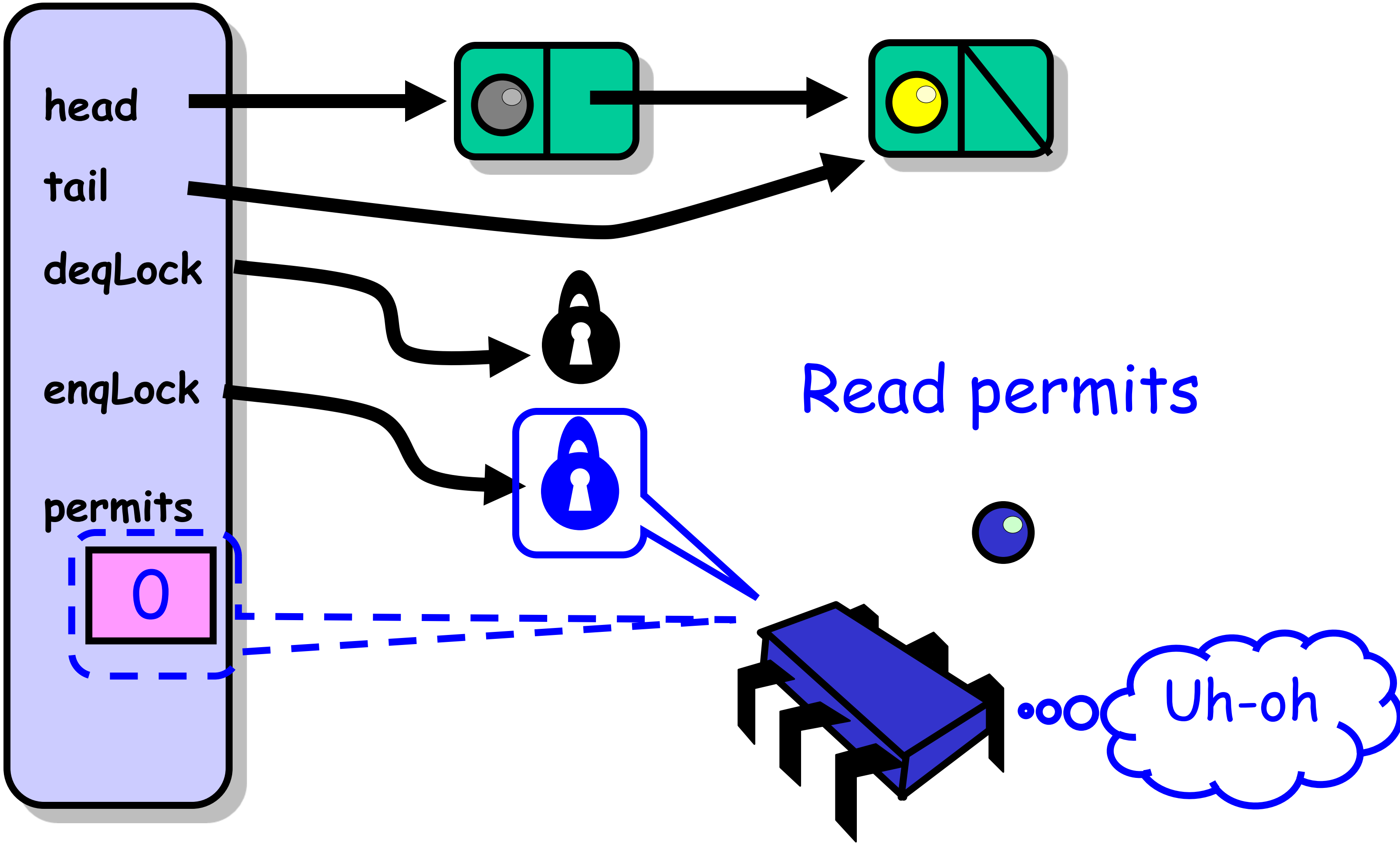
Release lock

Enqueuer

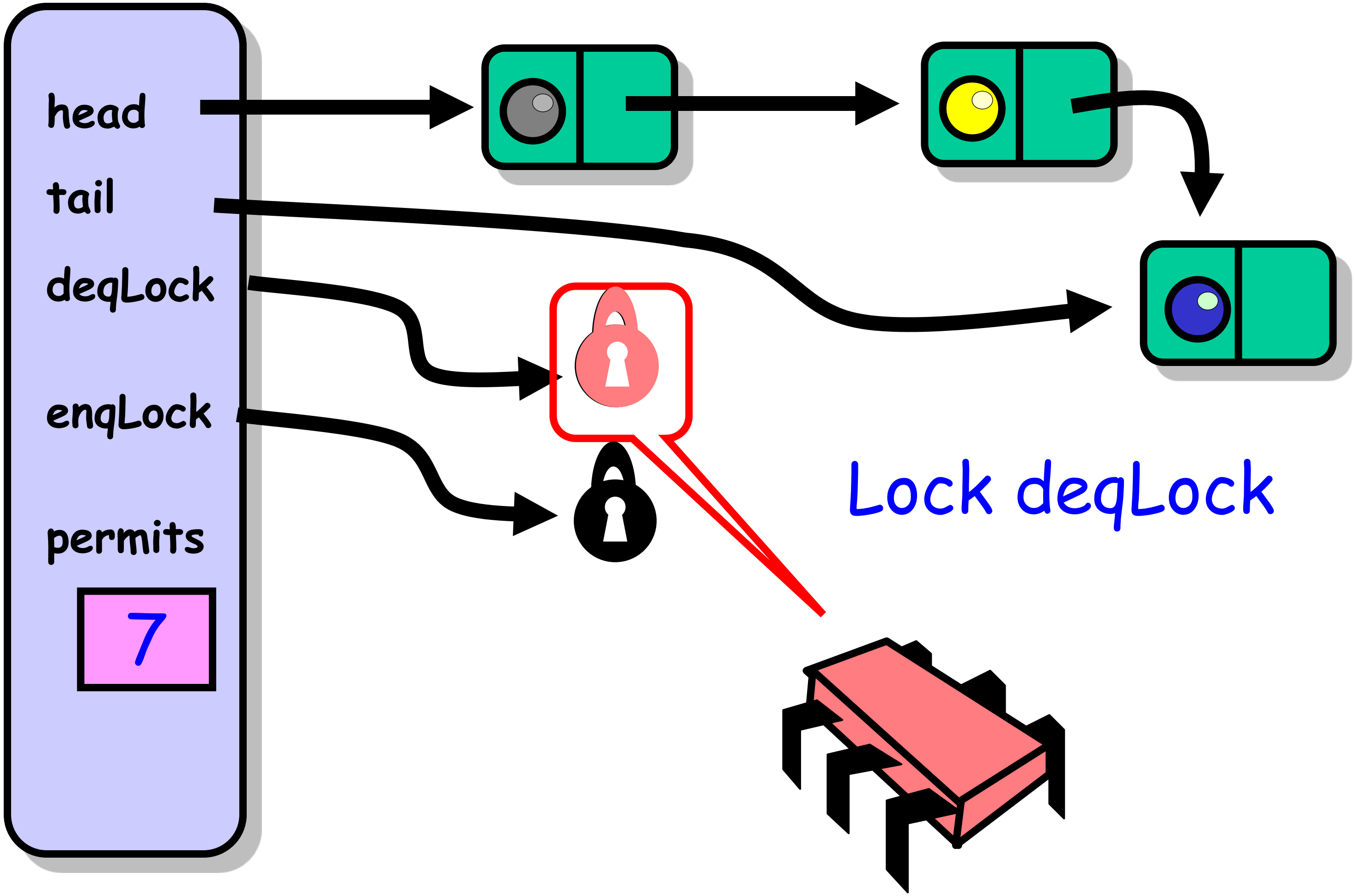


If queue was empty, notify waiting dequeuers

Unsuccesful Enqueuer

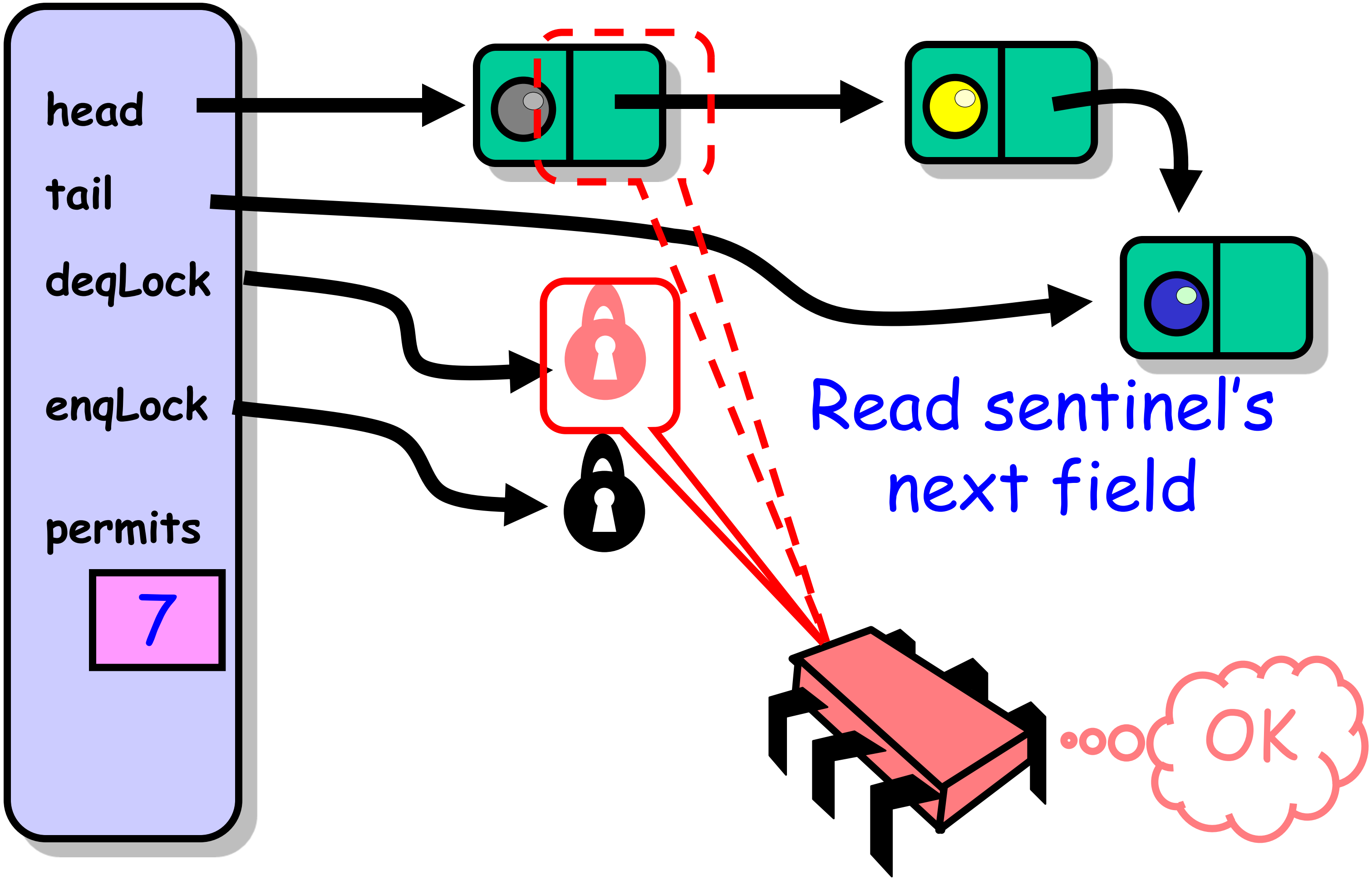


Dequeuer



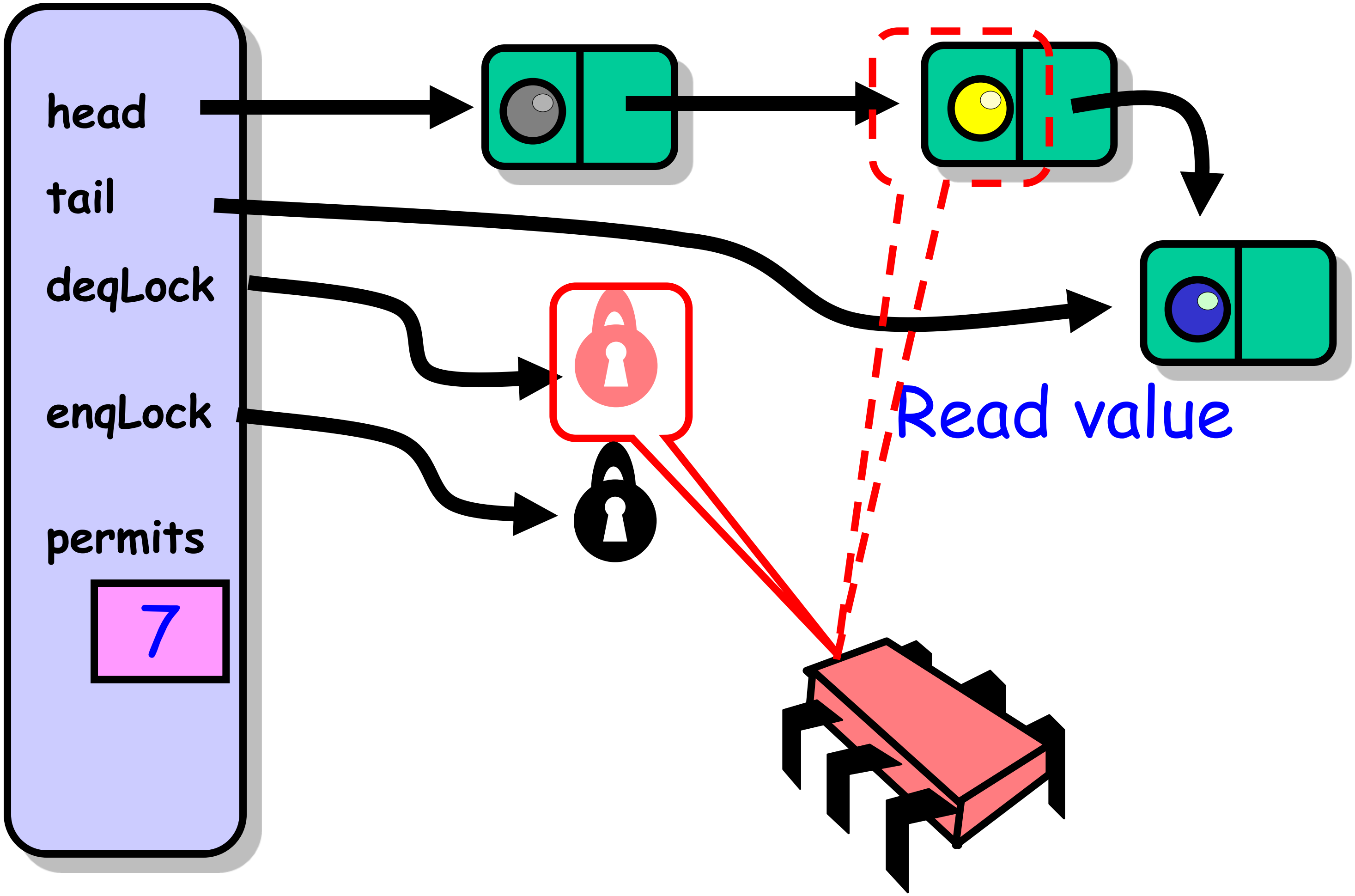
Lock deqLock

Dequeuer



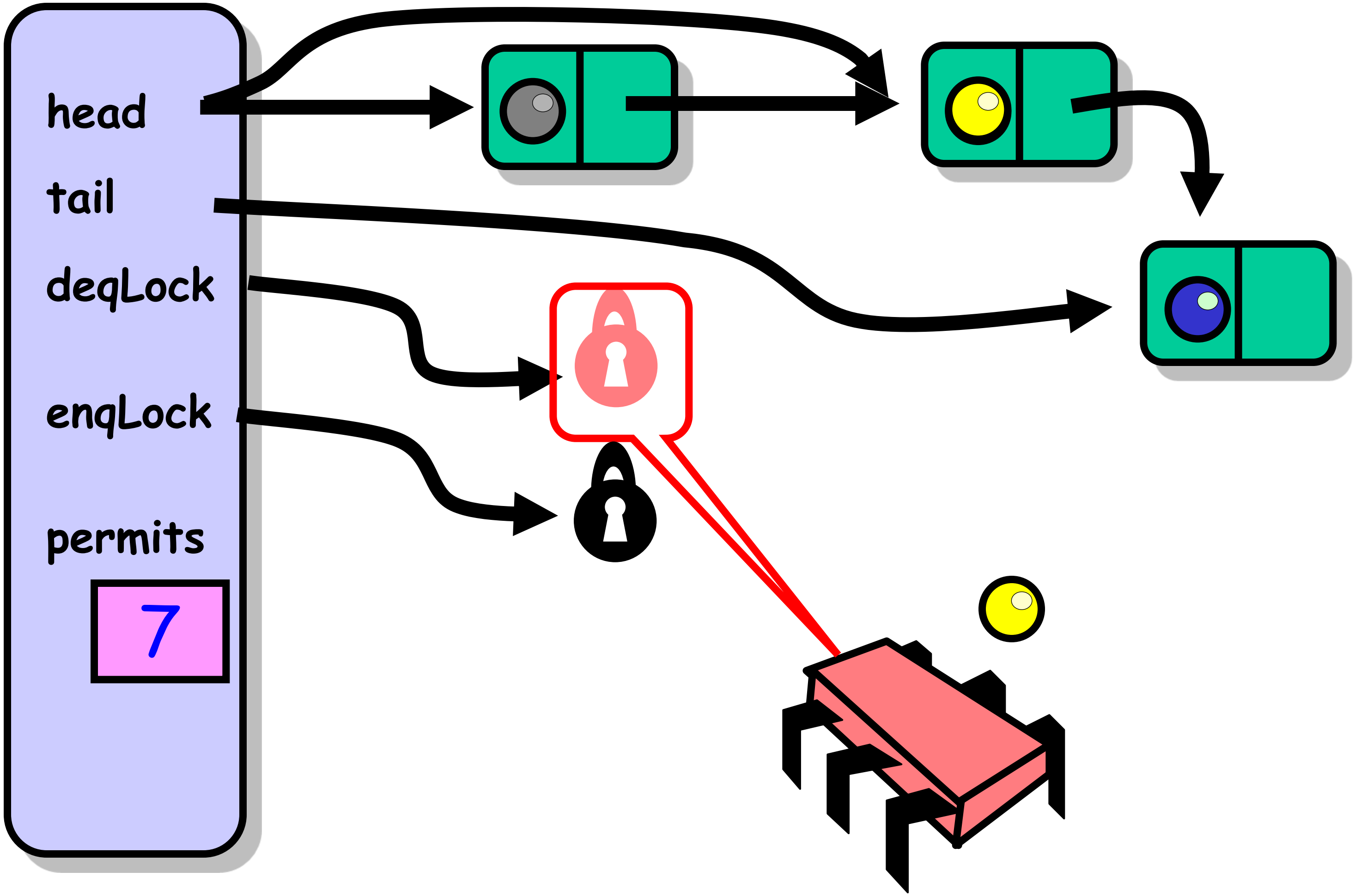
Read sentinel's next field

Dequeuer

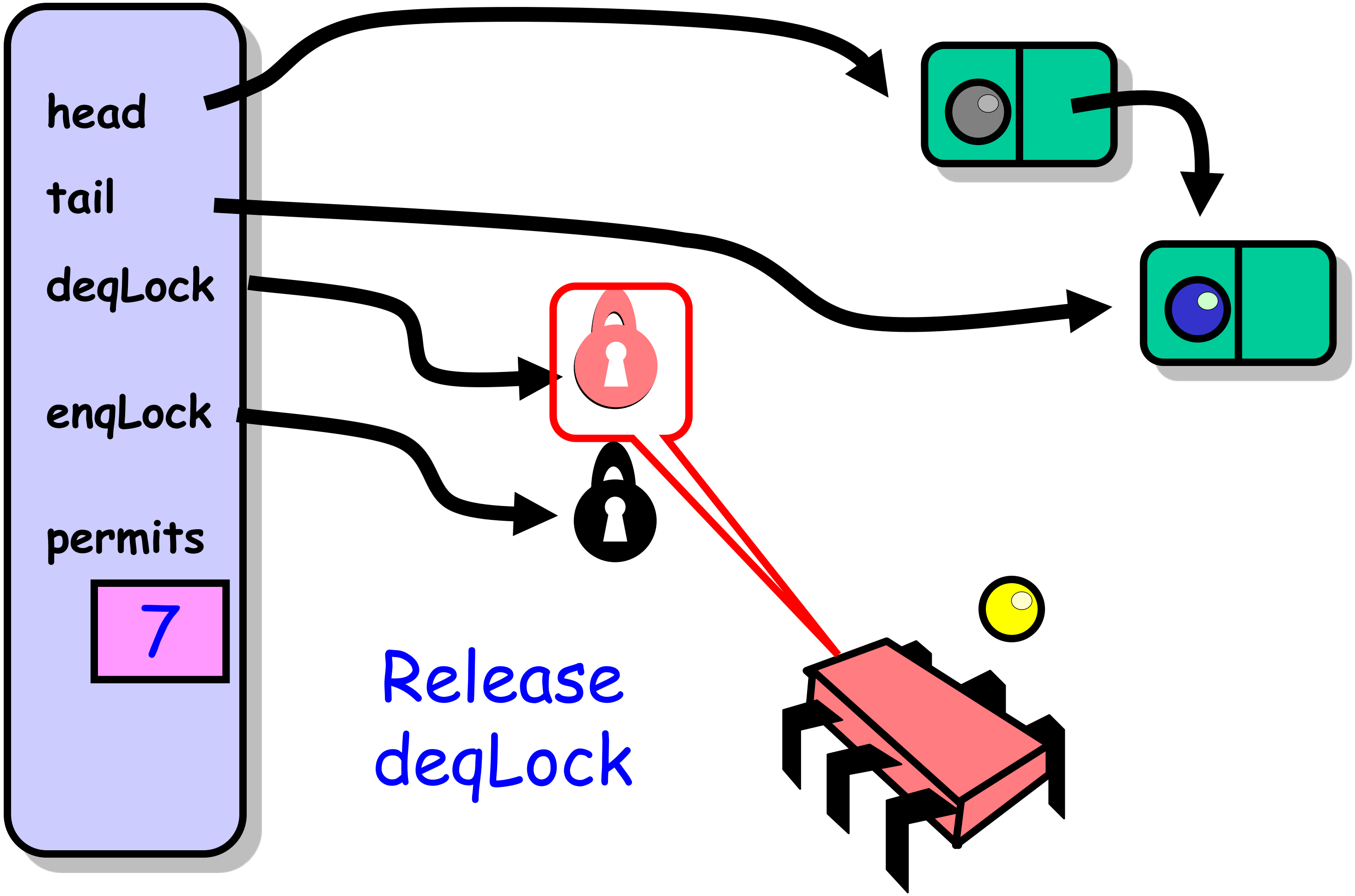


Make first Node new sentinel

Dequeuer

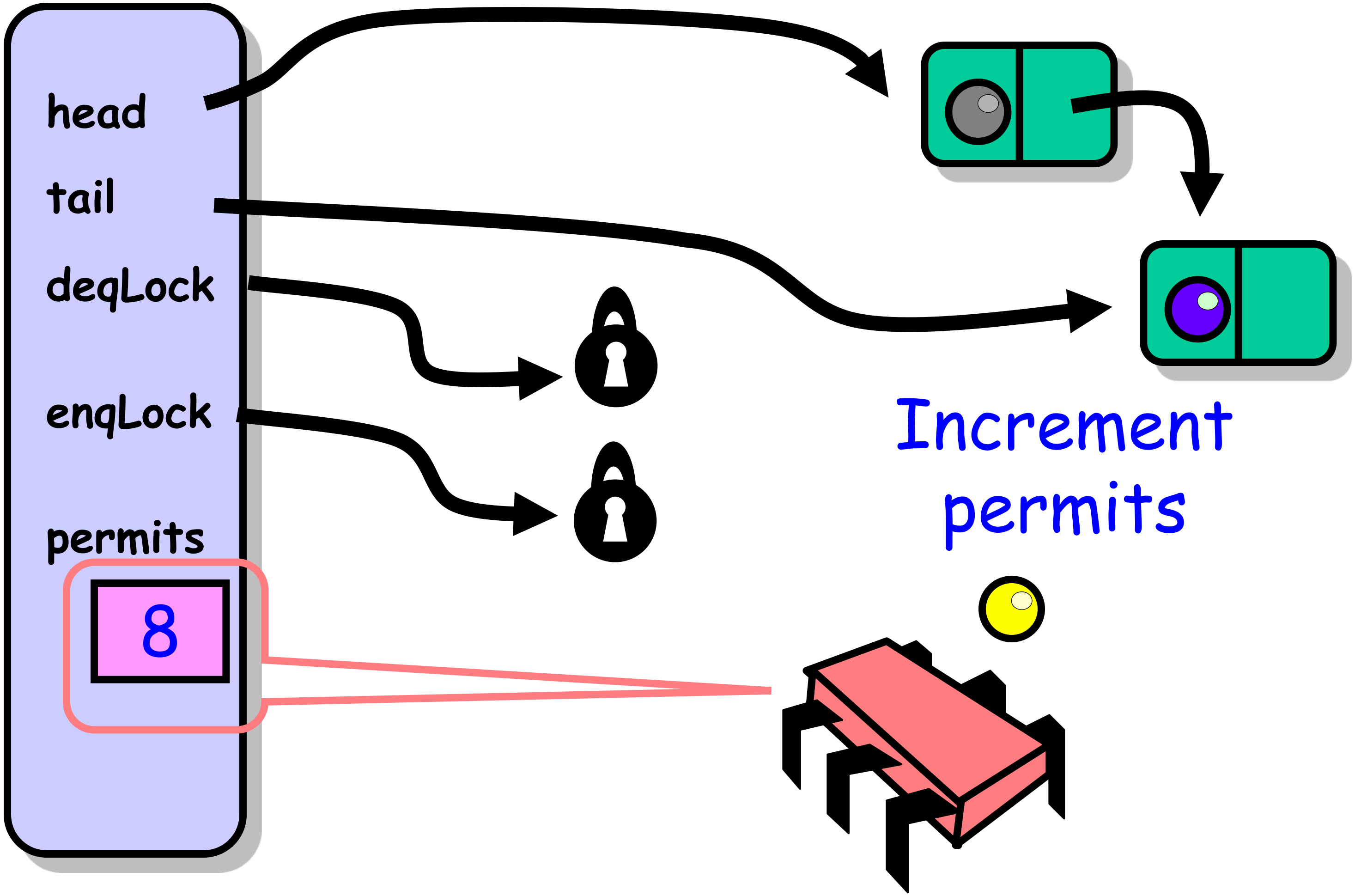


Dequeuer



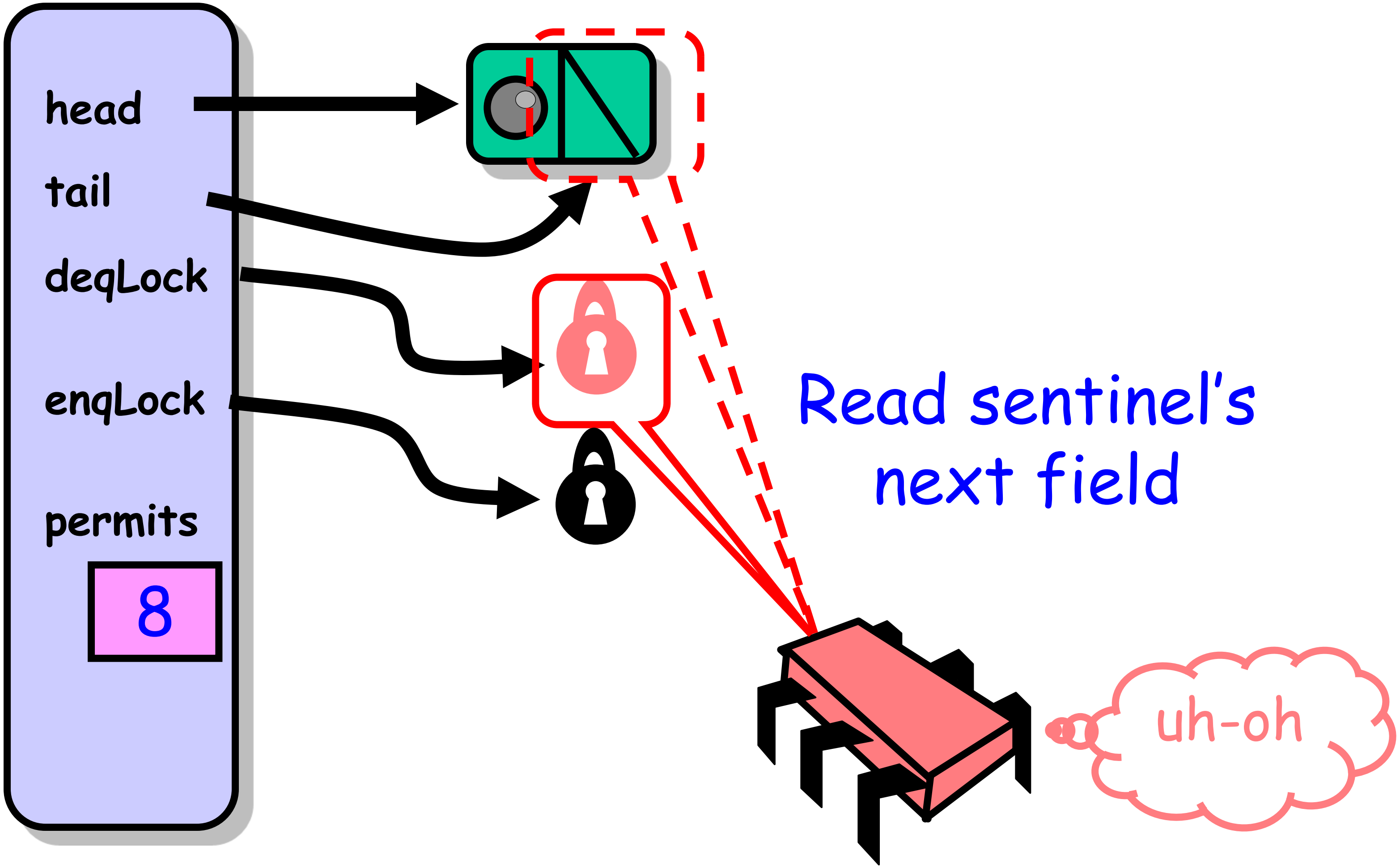
Release deqLock

Dequeuer



Increment permits

Unsuccesful Dequeuer



Read sentinel's next field

Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Enq & deq locks

Digression: Monitor Locks

- The ReentrantLock is a monitor
- Allows blocking on a condition rather than spinning
- Threads:
  - acquire and release lock
  - wait on a condition

The Java Lock Interface

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Acquire lock

The Java Lock Interface

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Release lock

The Java Lock Interface

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Try for lock, but not too hard

The Java Lock Interface

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Create condition to wait on

The Java Lock Interface

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Guess what this method does?

Lock Conditions

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
 }
```

Lock Conditions

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
}
```

Release lock and wait on condition

Lock Conditions

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
 }
```

Wake up one waiting thread

Lock Conditions

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
}
```

Wake up all waiting threads

Await

```
q.await()
```

- Releases lock associated with *q*
- Sleeps (gives up processor)
- Awakens (resumes running)
- Reacquires lock & returns

Signal

```
q.signal();
```

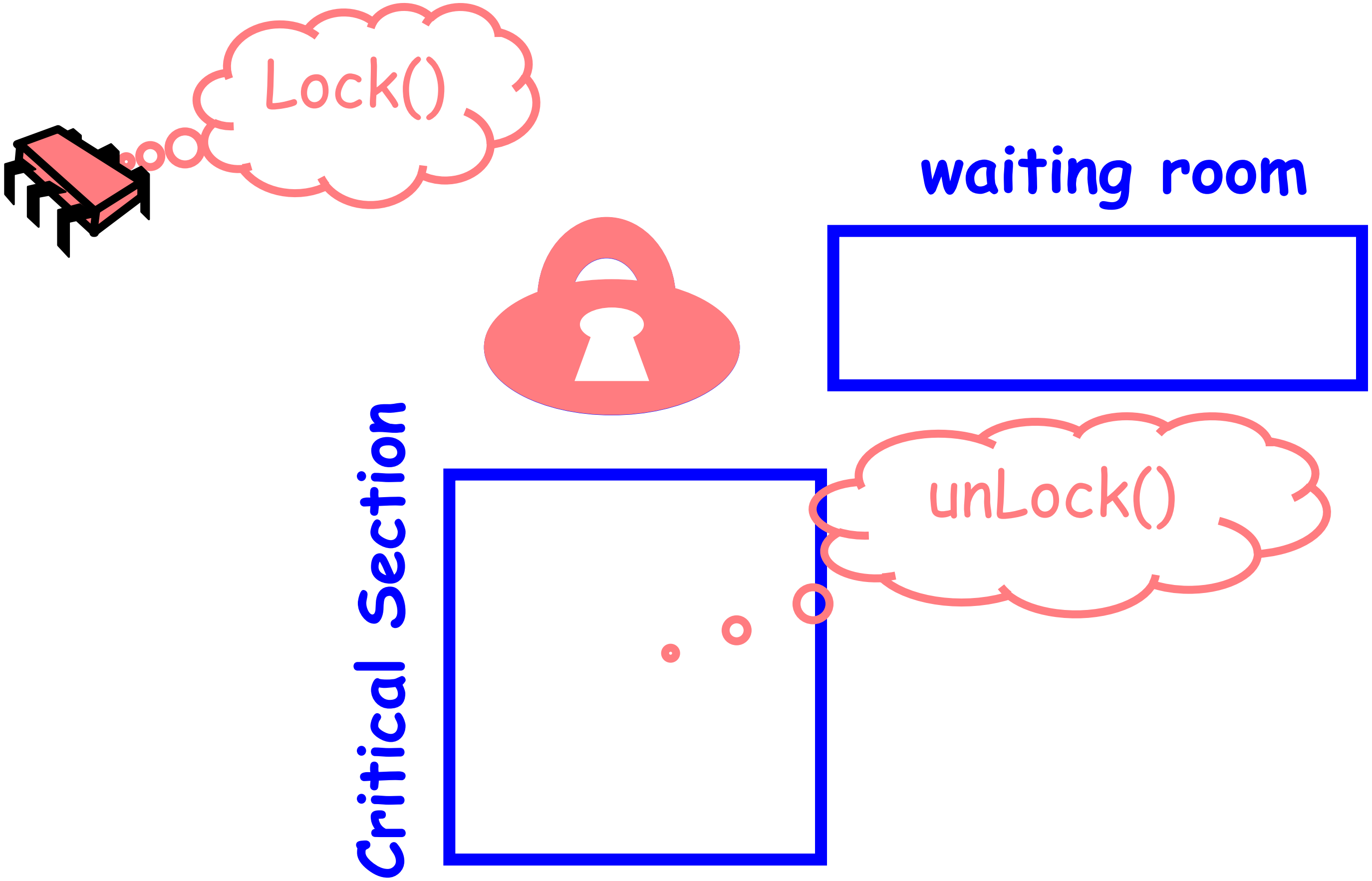
- Awakens one waiting thread
  - Which will reacquire lock

Signal All

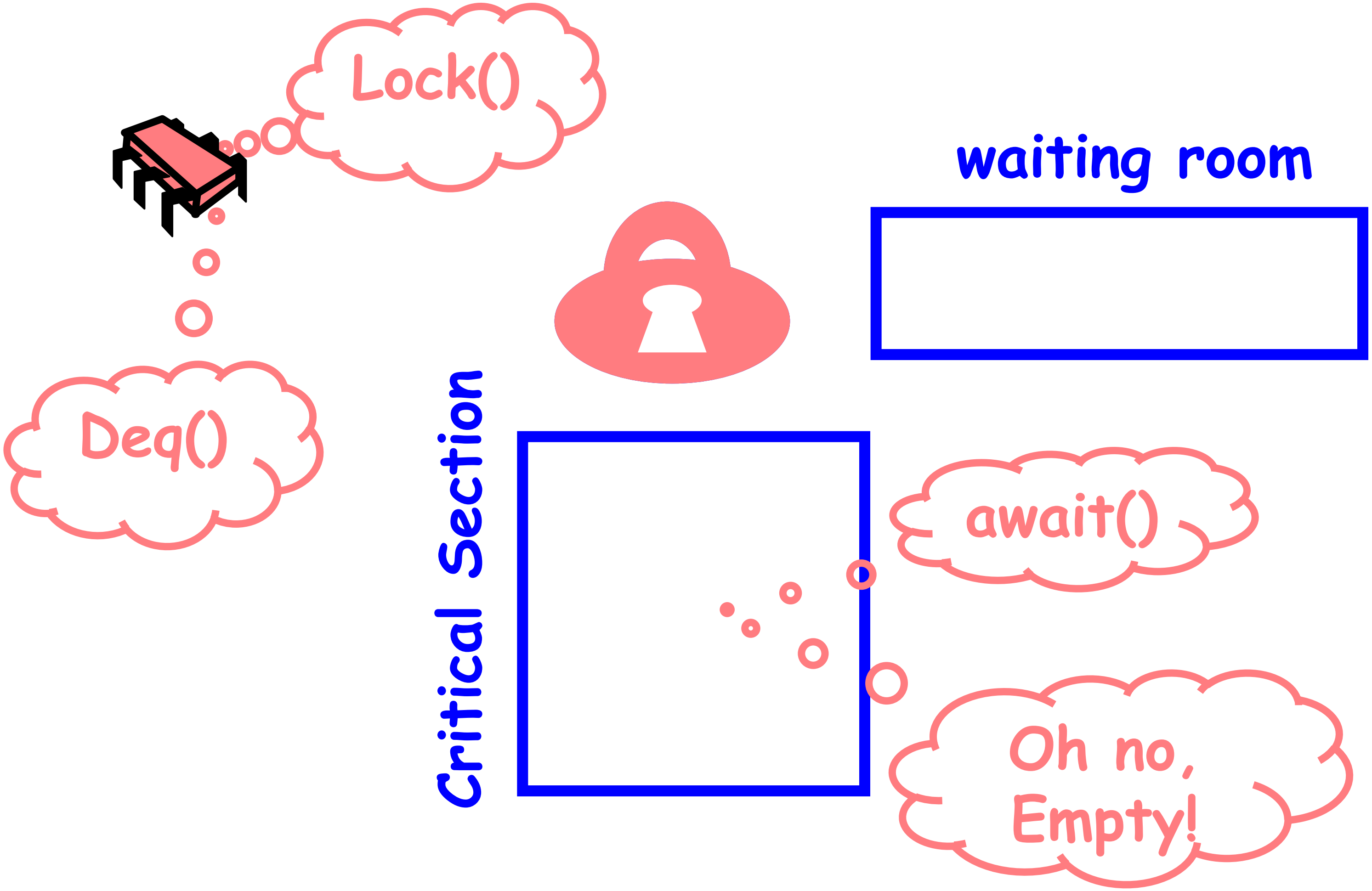
```
q.signalAll();
```

- Awakens all waiting threads
  - Which will each reacquire lock

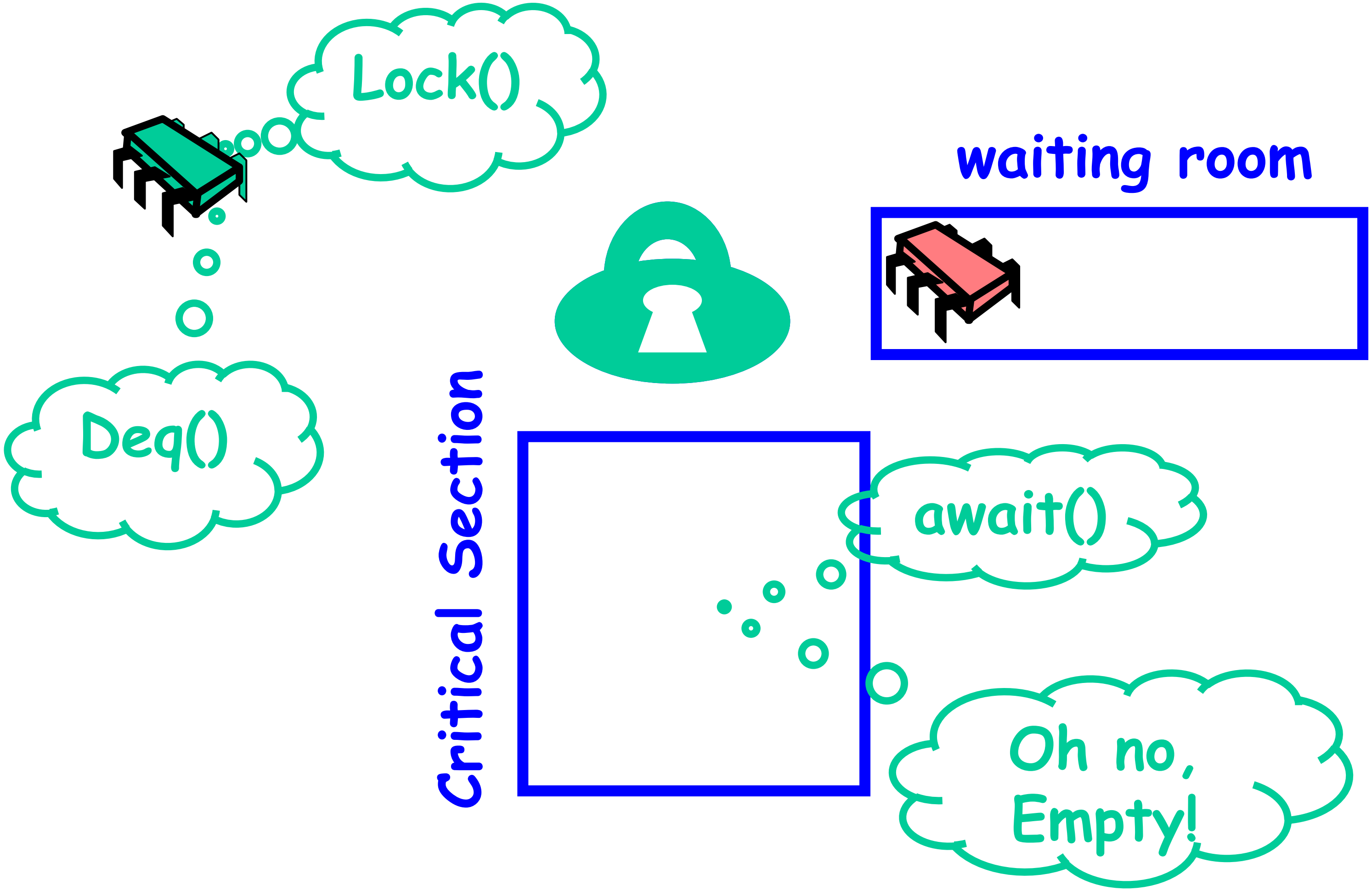
A Monitor Lock



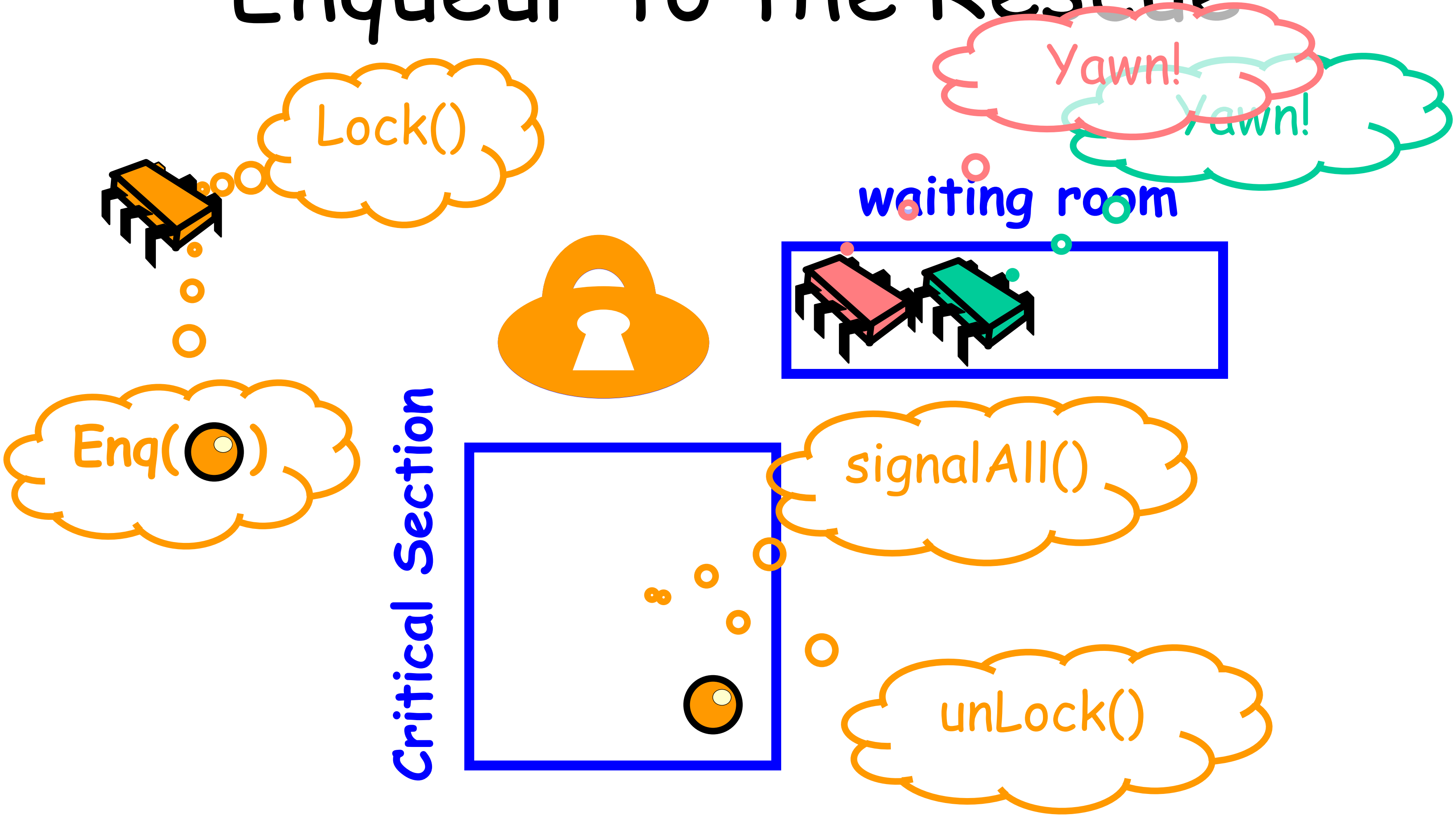
Unsuccessful Deq



Another One



Enqueur to the Rescue

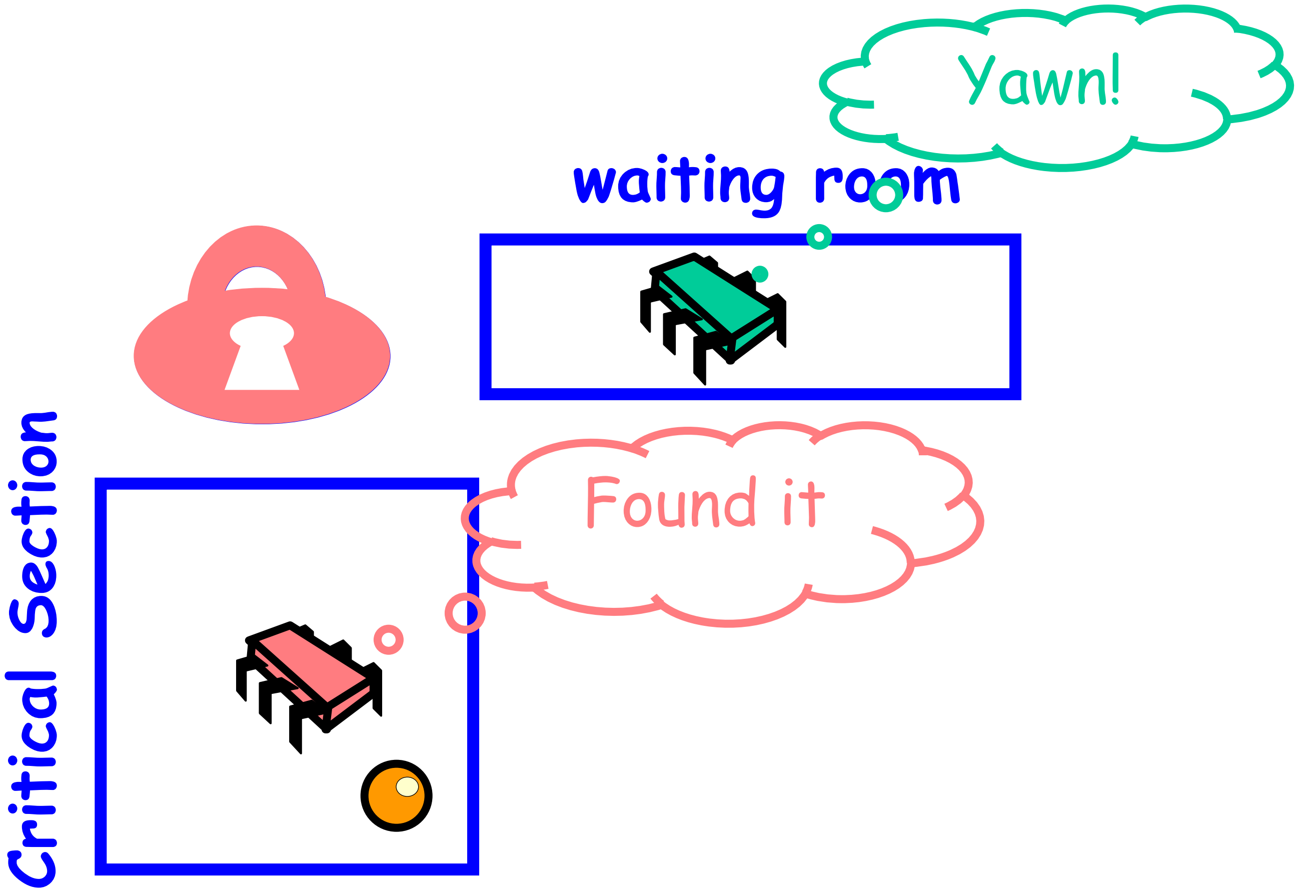


Monitor Signalling

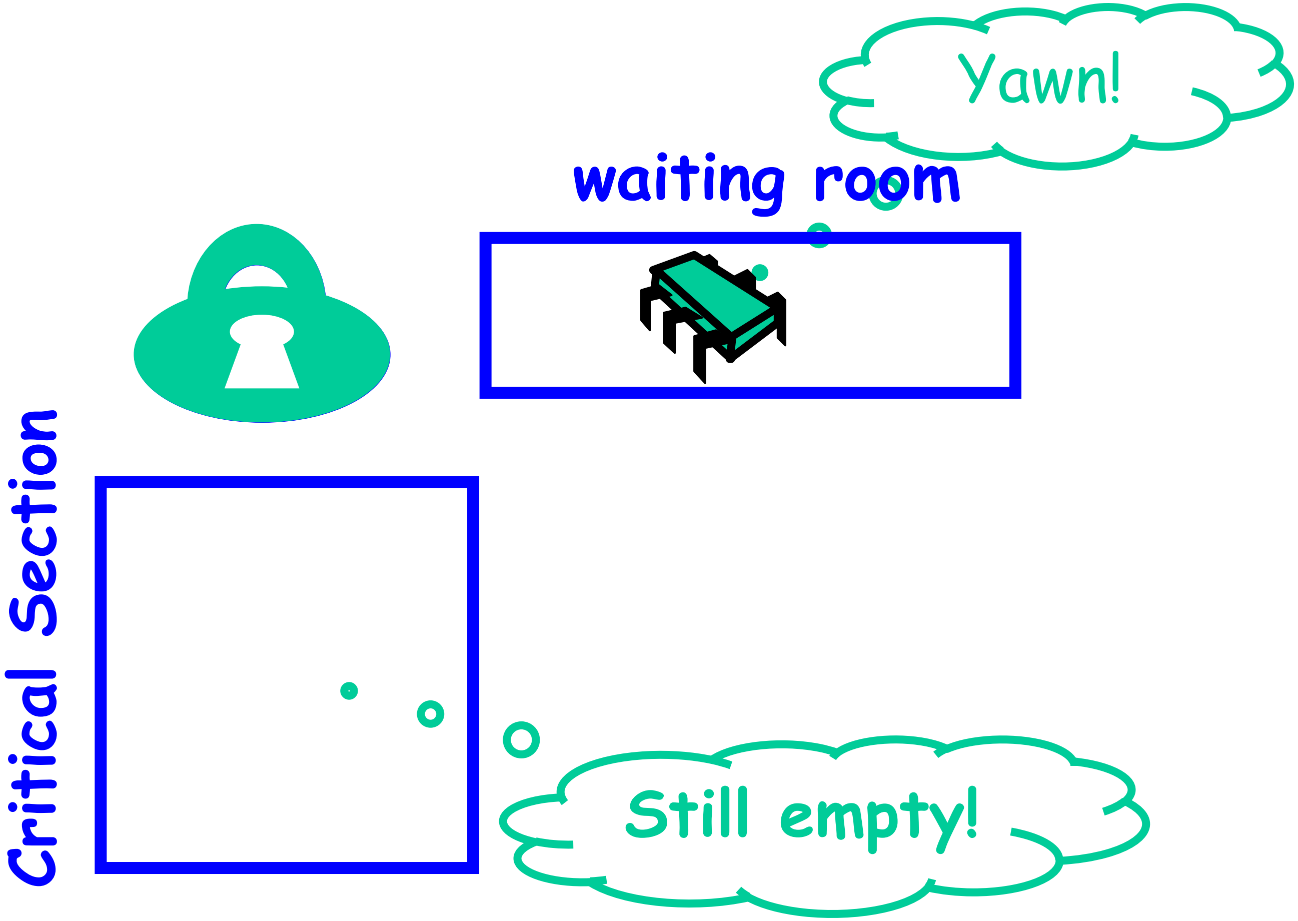
Yawn!
Yawn!
waiting room
Critical Section

Awakend thread might still lose lock to outside contender…

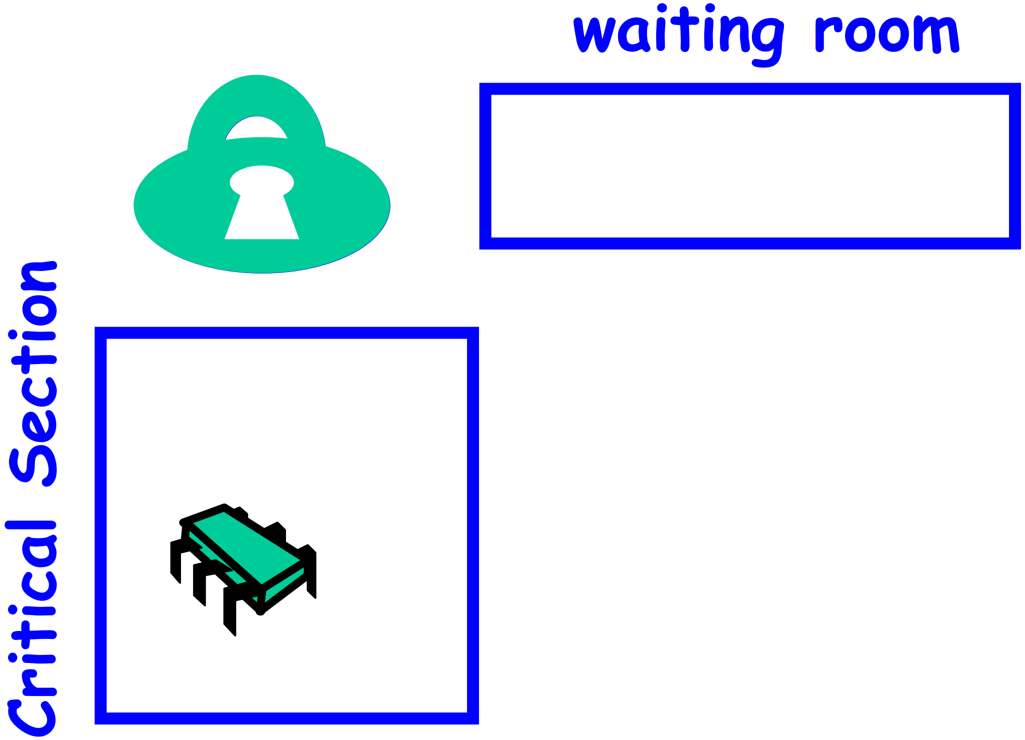
Dequeurs Signalled



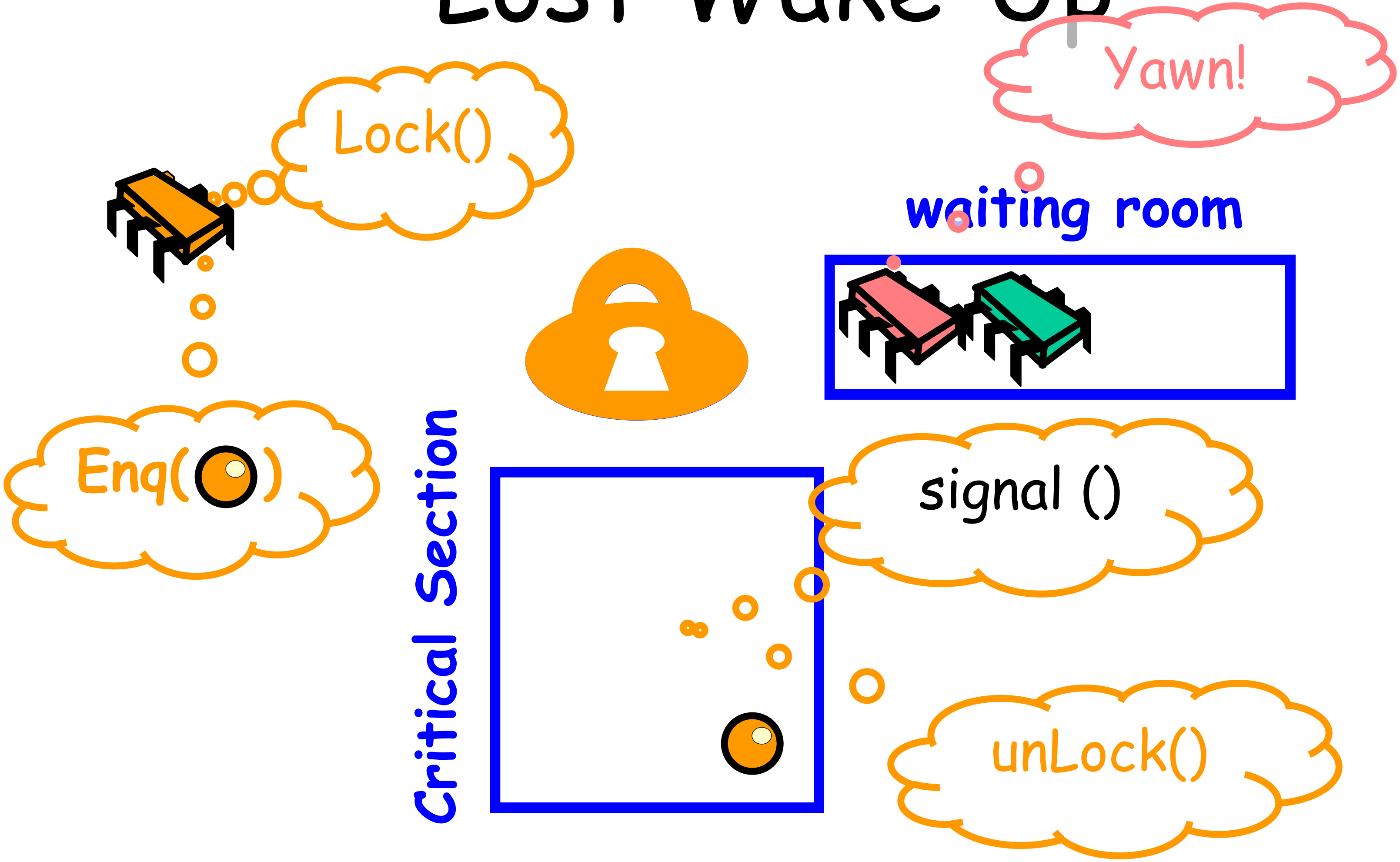
Dequeurs Signalled



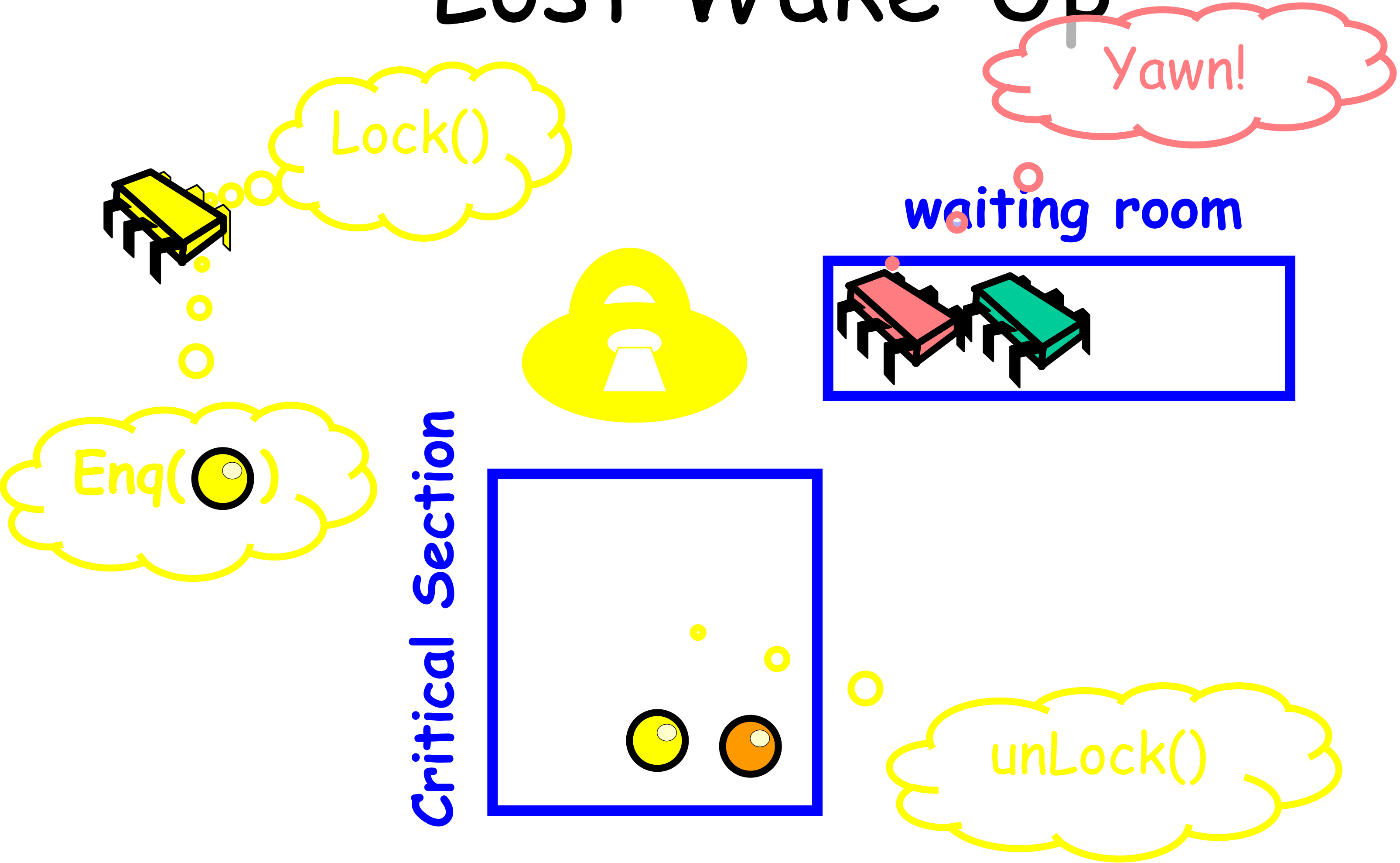
Dollar Short + Day Late



Lost Wake-Up



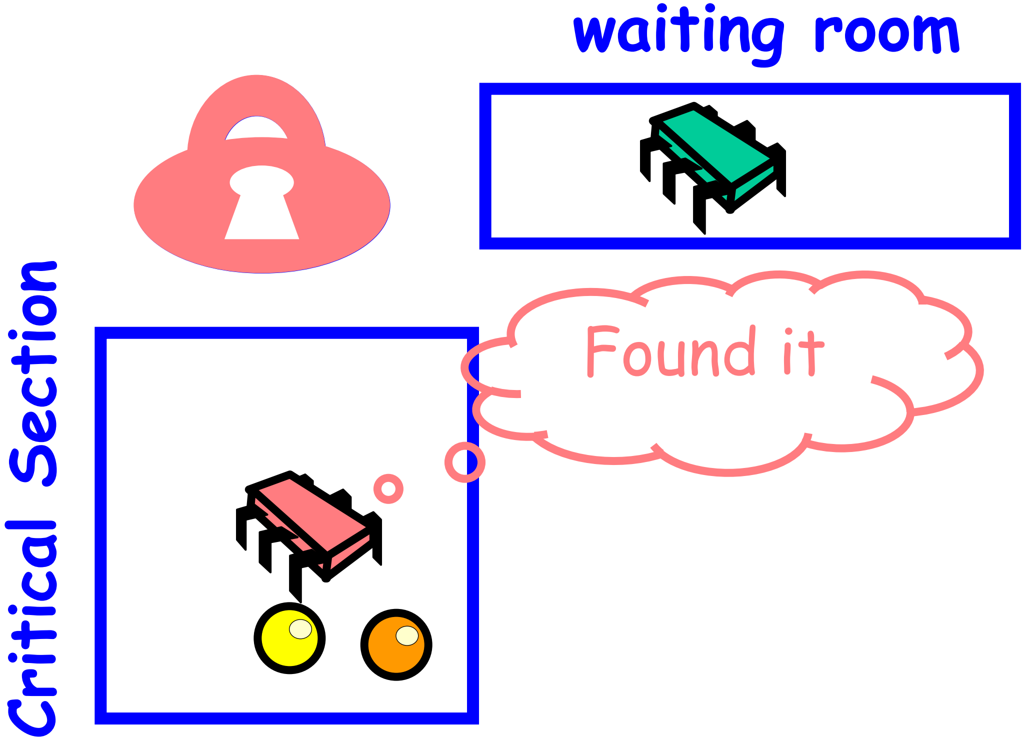
Lost Wake-Up



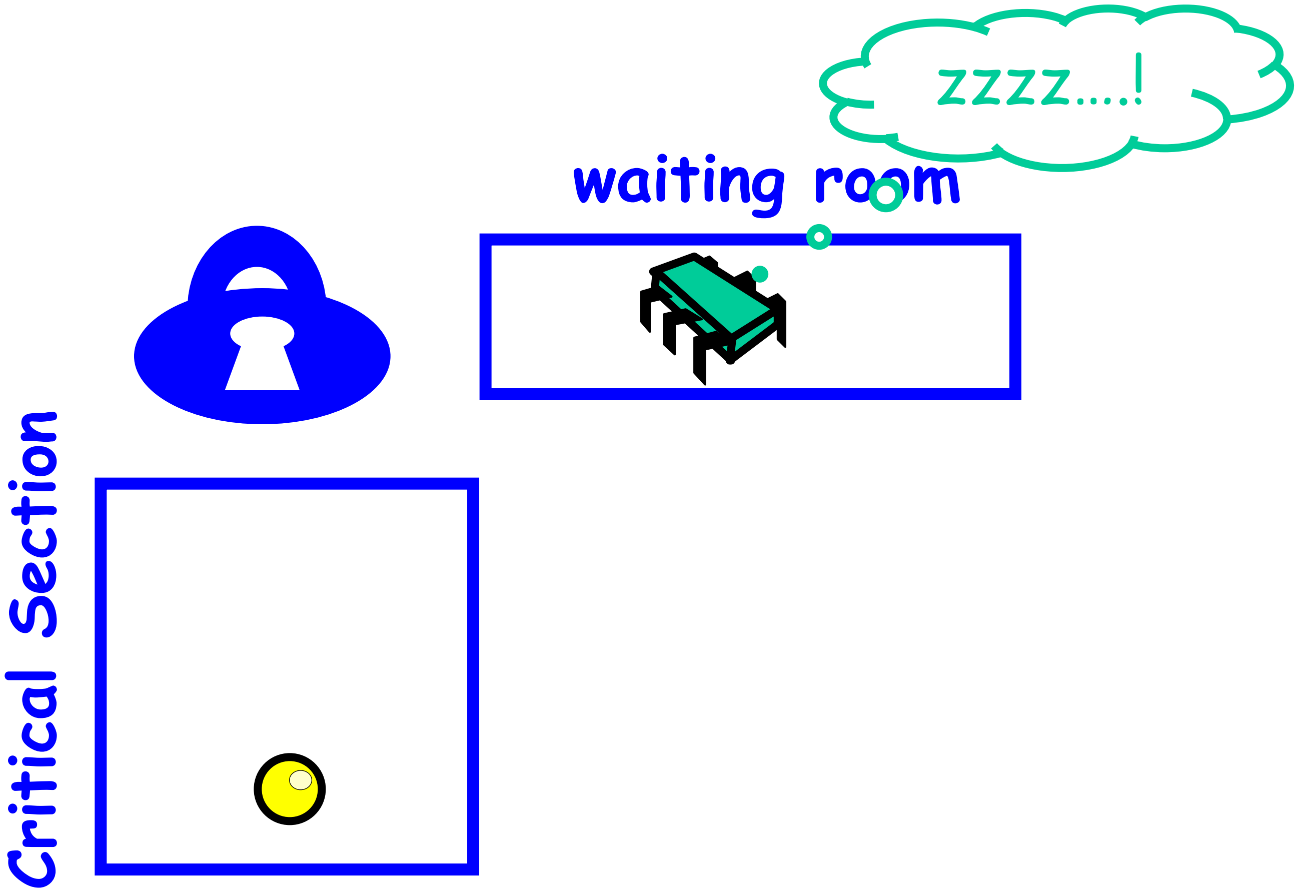
Lost Wake-Up

Yawn!
waiting room
Critical Section

Lost Wake-Up



What's Wrong Here?



Java Synchronized Methods

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail – head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Java Synchronized Methods

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail – head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Each object has an implicit lock with an implicit condition

Java Synchronized Methods

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail – head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Lock on entry, unlock on return

Java Synchronized Methods

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail – head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Wait on implicit condition

Java Synchronized Methods

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail – head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Signal all threads waiting on condition

(Pop!) The Bounded Queue

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Bounded Queue Fields

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Enq & deq locks

Bounded Queue Fields

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Enq lock's associated condition

Bounded Queue Fields

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Num permits: 0 to capacity

Bounded Queue Fields

```
public class BoundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Condition notEmptyCondition, notFullCondition;
  AtomicInteger permits;
  Node head;
  Node tail;
  int capacity;
  enqLock = new ReentrantLock();
  notFullCondition = enqLock.newCondition();
  deqLock = new ReentrantLock();
  notEmptyCondition = deqLock.newCondition();
}
```

Head and Tail

Enq Method Part One

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
  while (permits.get() == 0)
    notFullCondition.await();
  Node e = new Node(x);
  tail.next = e;
  tail = e;
  if (permits.getAndDecrement() == capacity)
   mustWakeDequeuers = true;
 } finally {
   enqLock.unlock();
 }
 …
}
```

Enq Method Part One

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
  while (permits.get() == 0)
    notFullCondition.await();
  Node e = new Node(x);
  tail.next = e;
  tail = e;
  if (permits.getAndDecrement() == capacity)
   mustWakeDequeuers = true;
 } finally {
   enqLock.unlock();
 }
 …
}
```

Lock and unlock enq lock

Enq Method Part One

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
  while (permits.get() == 0)
    notFullCondition.await();
  Node e = new Node(x);
  tail.next = e;
  tail = e;
  if (permits.getAndDecrement() == capacity)
   mustWakeDequeuers = true;
 } finally {
   enqLock.unlock();
 }
 …
}
```

If queue is full, patiently await further instructions …

Be Afraid

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
  while (permits.get() == 0)
    notFullCondition.await();
  Node e = new Node(x);
  tail.next = e;
  tail = e;
  if (permits.getAndDecrement() == capacity)
   mustWakeDequeuers = true;
 } finally {
   enqLock.unlock();
 }
 …
}
```

How do we know the permits field won't change?

Enq Method Part One

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
  while (permits.get() == 0)
    notFullCondition.await();
  Node e = new Node(x);
  tail.next = e;
  tail = e;
  if (permits.getAndDecrement() == capacity)
   mustWakeDequeuers = true;
 } finally {
   enqLock.unlock();
 }
 …
}
```

Add new node

Enq Method Part One

```
public void enq(T x) {
 boolean mustWakeDequeuers = false;
 enqLock.lock();
 try {
  while (permits.get() == 0)
    notFullCondition.await();
  Node e = new Node(x);
  tail.next = e;
  tail = e;
  if (permits.getAndDecrement() == capacity)
   mustWakeDequeuers = true;
 } finally {
   enqLock.unlock();
 }
 …
}
```

If queue was empty, wake frustrated dequeuers

Enq Method Part Deux

```
public void enq(T x) {
 …
    if (mustWakeDequeuers) {
      deqLock.lock();
      try {
        notEmptyCondition.signalAll();
      } finally {
        deqLock.unlock();
      }
    }
  }
```

Enq Method Part Deux

```
public void enq(T x) {
 …
    if (mustWakeDequeuers) {
      deqLock.lock();
      try {
        notEmptyCondition.signalAll();
      } finally {
        deqLock.unlock();
      }
    }
}
```

Are there dequeuers to be signaled?

Enq Method Part Deux

```
public void enq(T x) {
 …
    if (mustWakeDequeuers)
      deqLock.lock();
      try {
        notEmptyCondition.signalAll();
      } finally {
        deqLock.unlock();
      }
    }
  }
```

Lock and unlock deq lock

Enq Method Part Deux

Signal dequeuers that queue no longer empty

```
public void enq(T x) {
  ...
  if (mustWakeDequeuers) {
    deqLock.lock();
    try {
      notEmptyCondition.signalAll();
    } finally {
      deqLock.unlock();
    }
  }
}
```

The Enq() & Deq() Methods

- Share no locks
  - That's good
- But do share an atomic counter
  - Accessed on every method call
  - That's not so good
- Can we alleviate this bottleneck?

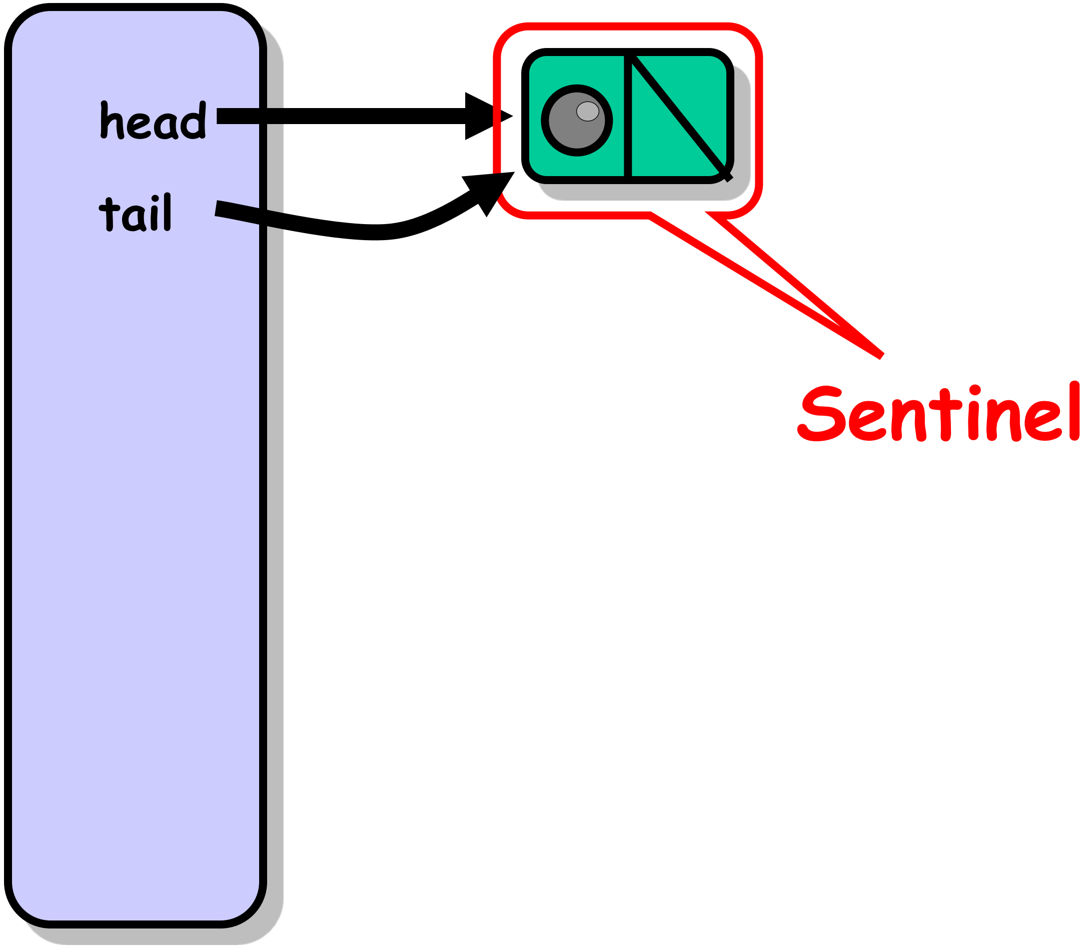
Split the Counter

- The **enq()** method
  - Decrements only
  - Cares only if value is **zero**
- The **deq()** method
  - Increments only
  - Cares only if value is **capacity**

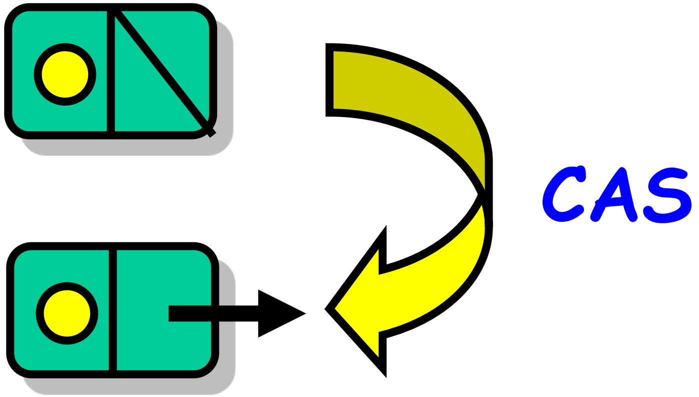
Split Counter

- Enqueuer decrements enqSidePermits
- Dequeuer increments deqSidePermits
- When enqueuer runs out
  - Locks **deqLock**
  - Transfers permits
- Intermittent synchronization
  - Not with each method call
  - Need both locks! (careful …)

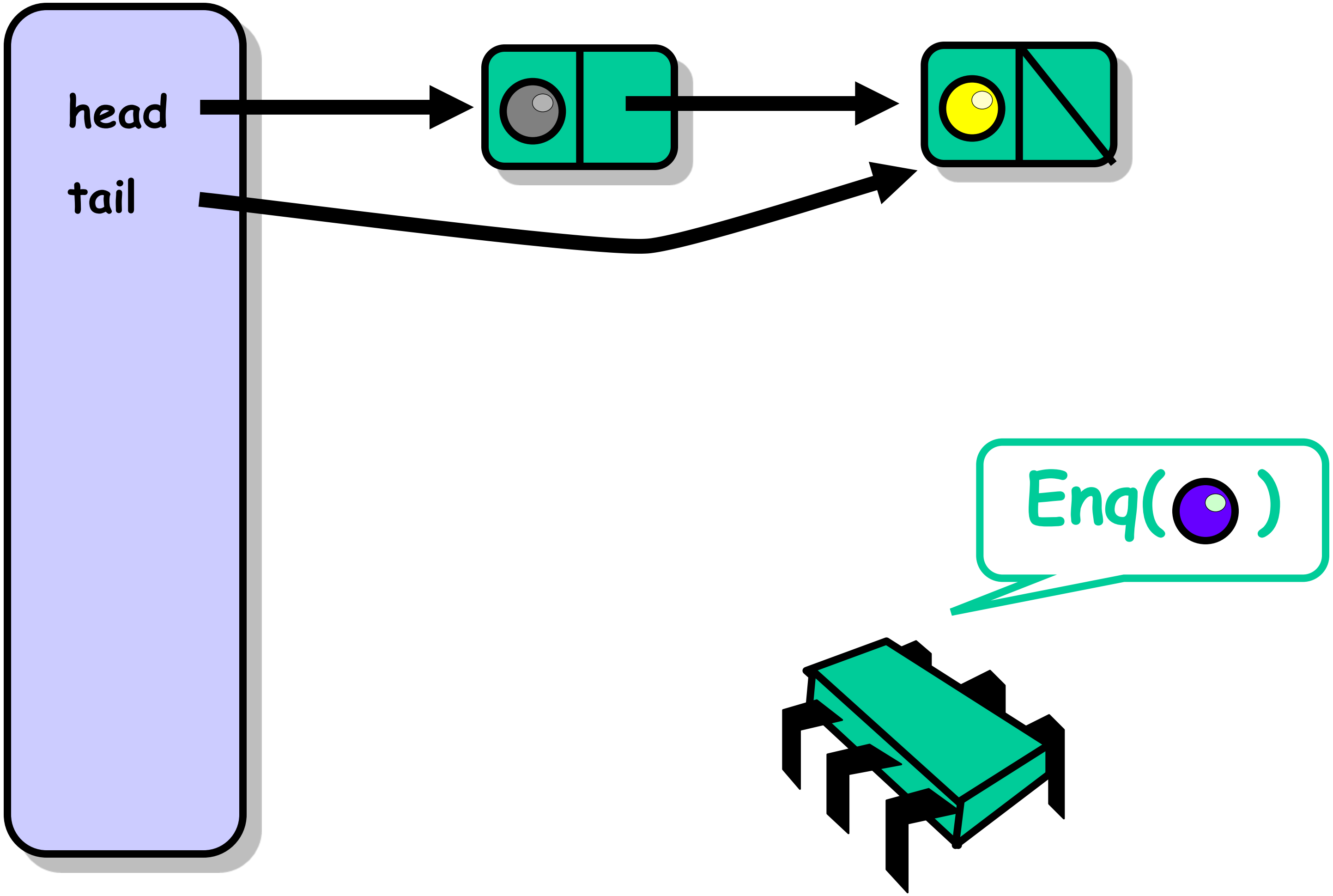
A Lock-Free Queue



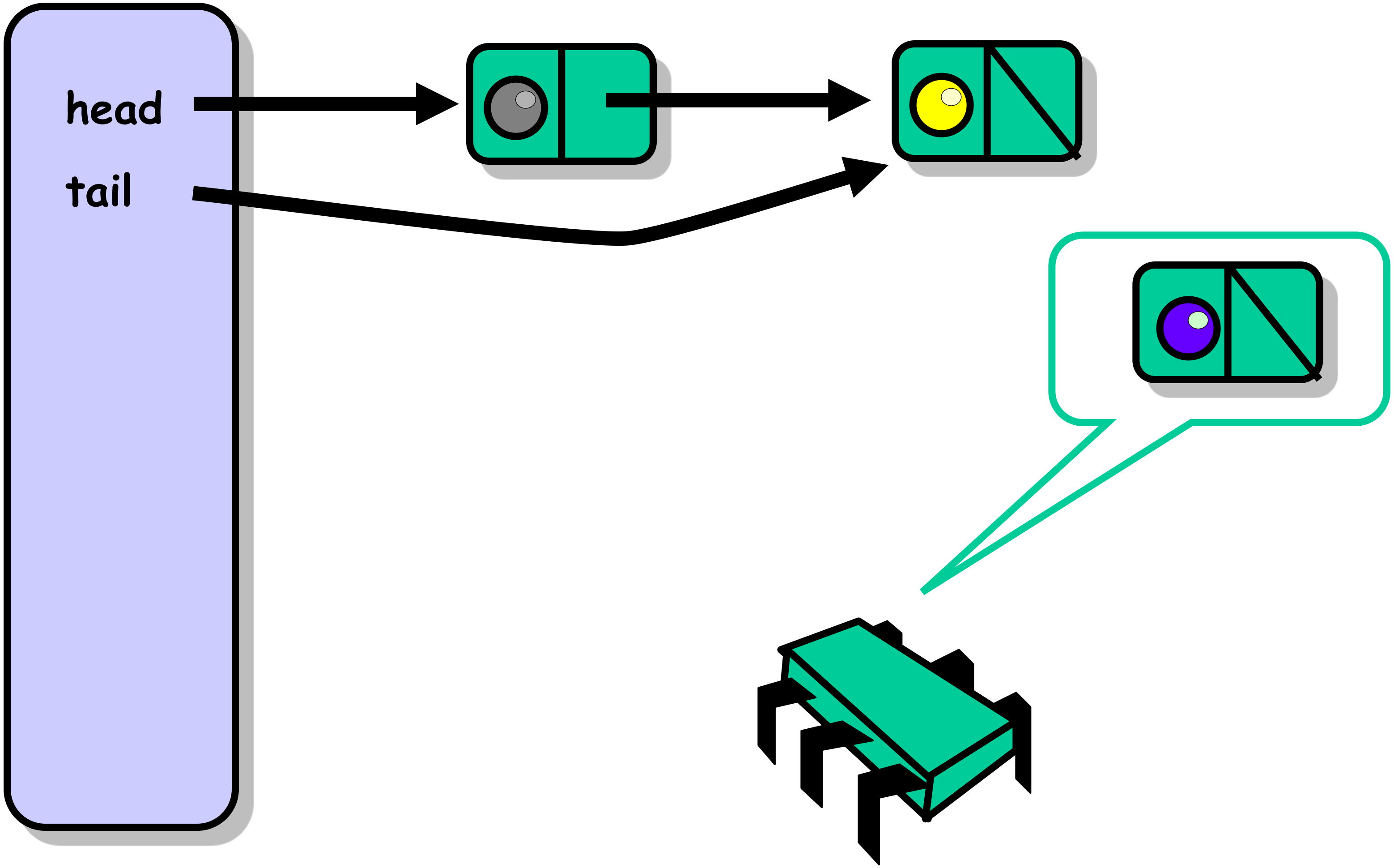
Compare and Set



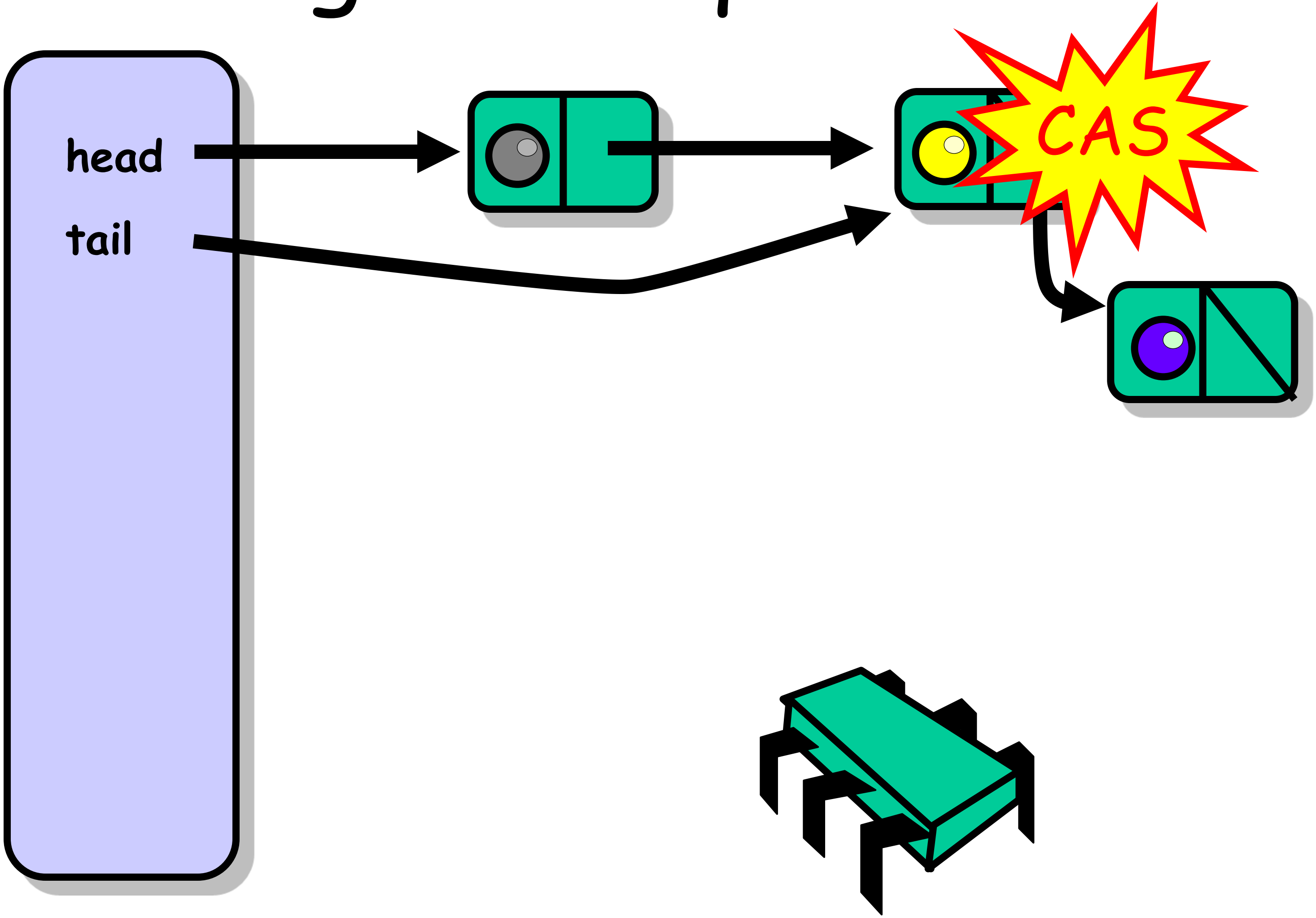
Enqueue



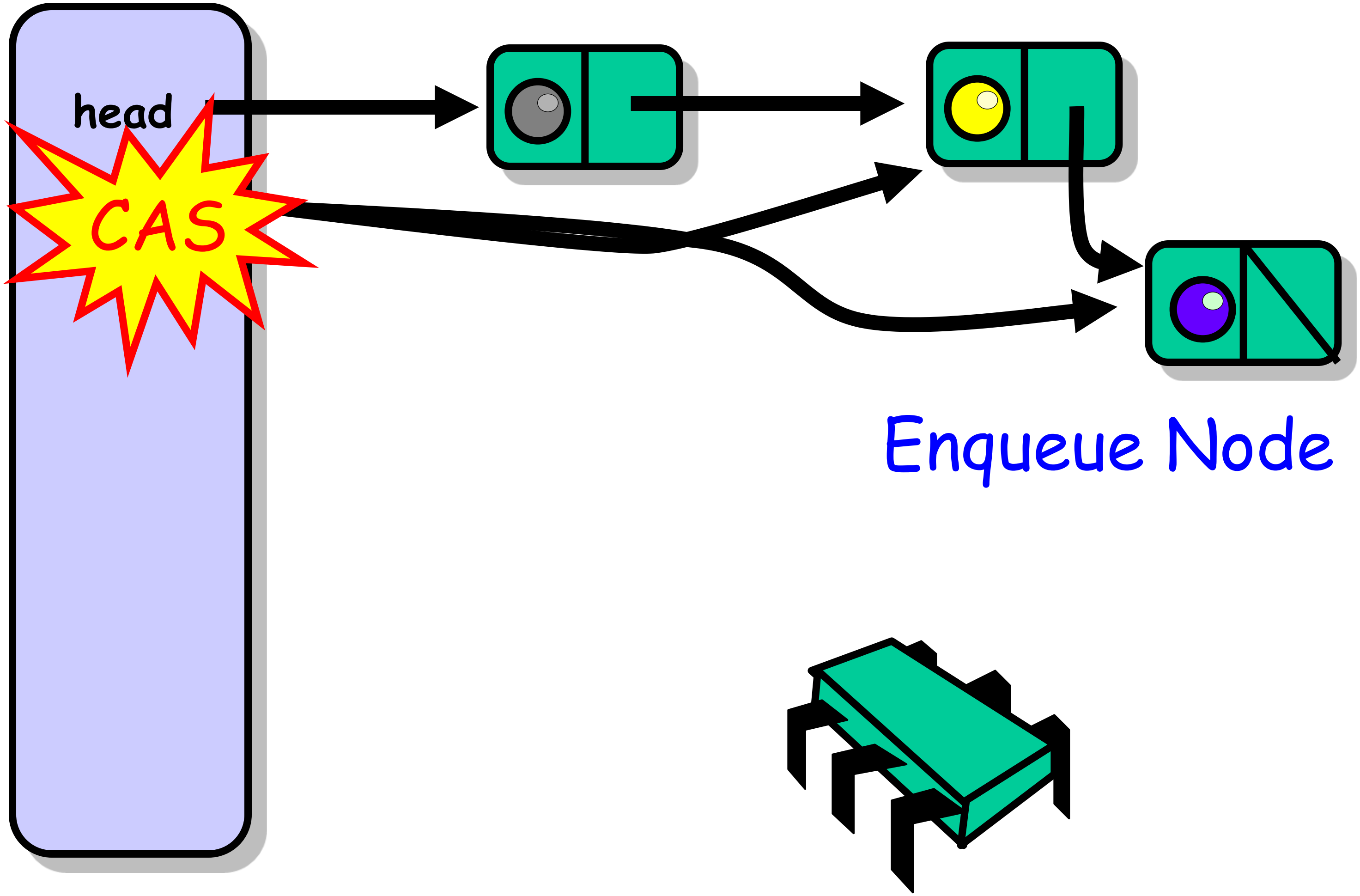
Enqueue



Logical Enqueue



Physical Enqueue



Enqueue

- These two steps are not atomic
- The tail field refers to either
  - Actual last Node (good)
  - Penultimate Node (not so good)
- Be prepared!

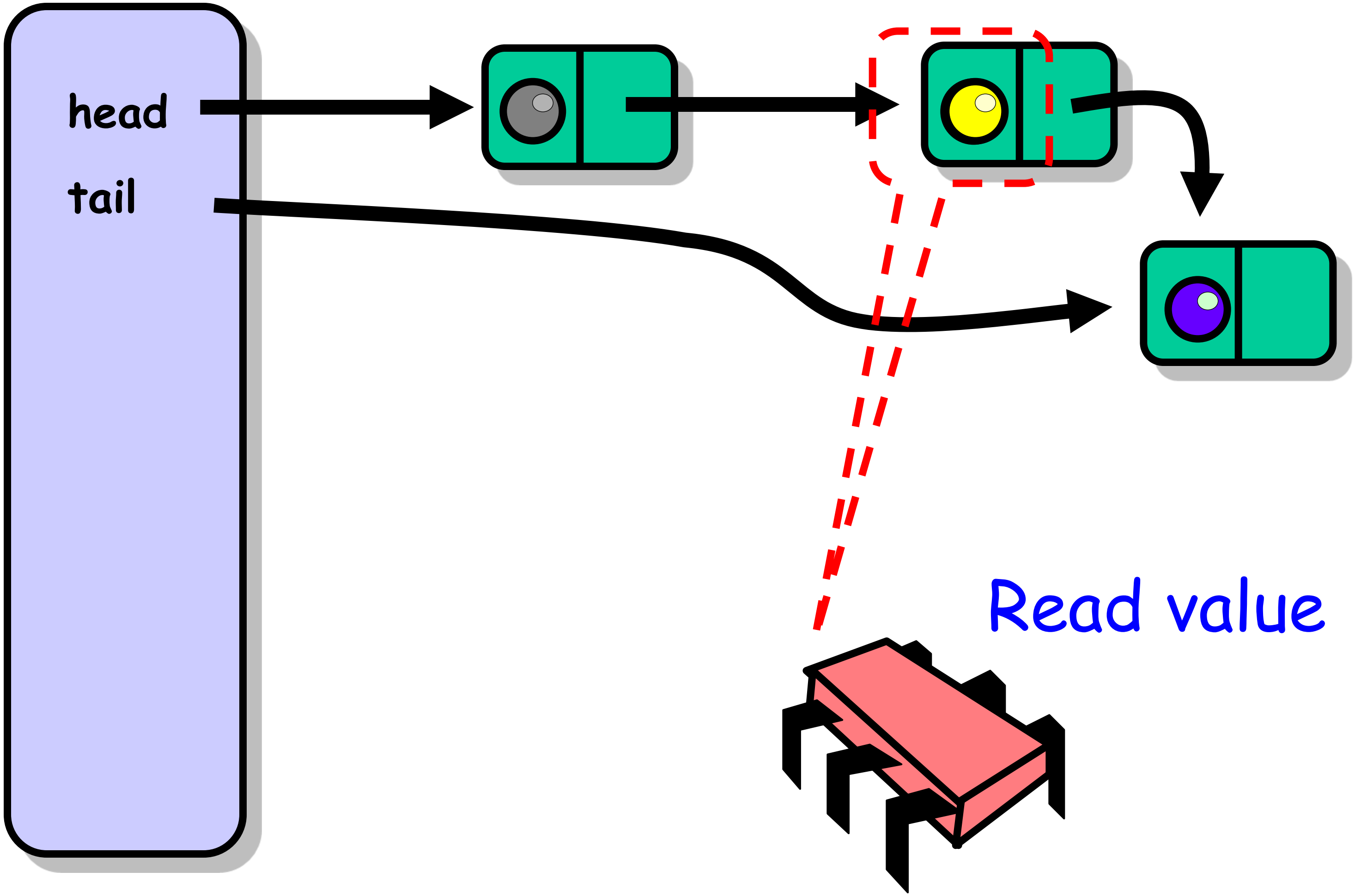
Enqueue

- What do you do if you find
  - A trailing **tail**?
- Stop and fix it
  - If **tail** node has non-*null* next field
  - CAS the queue's **tail** field to **tail.next**
- As in the universal construction

When CASs Fail

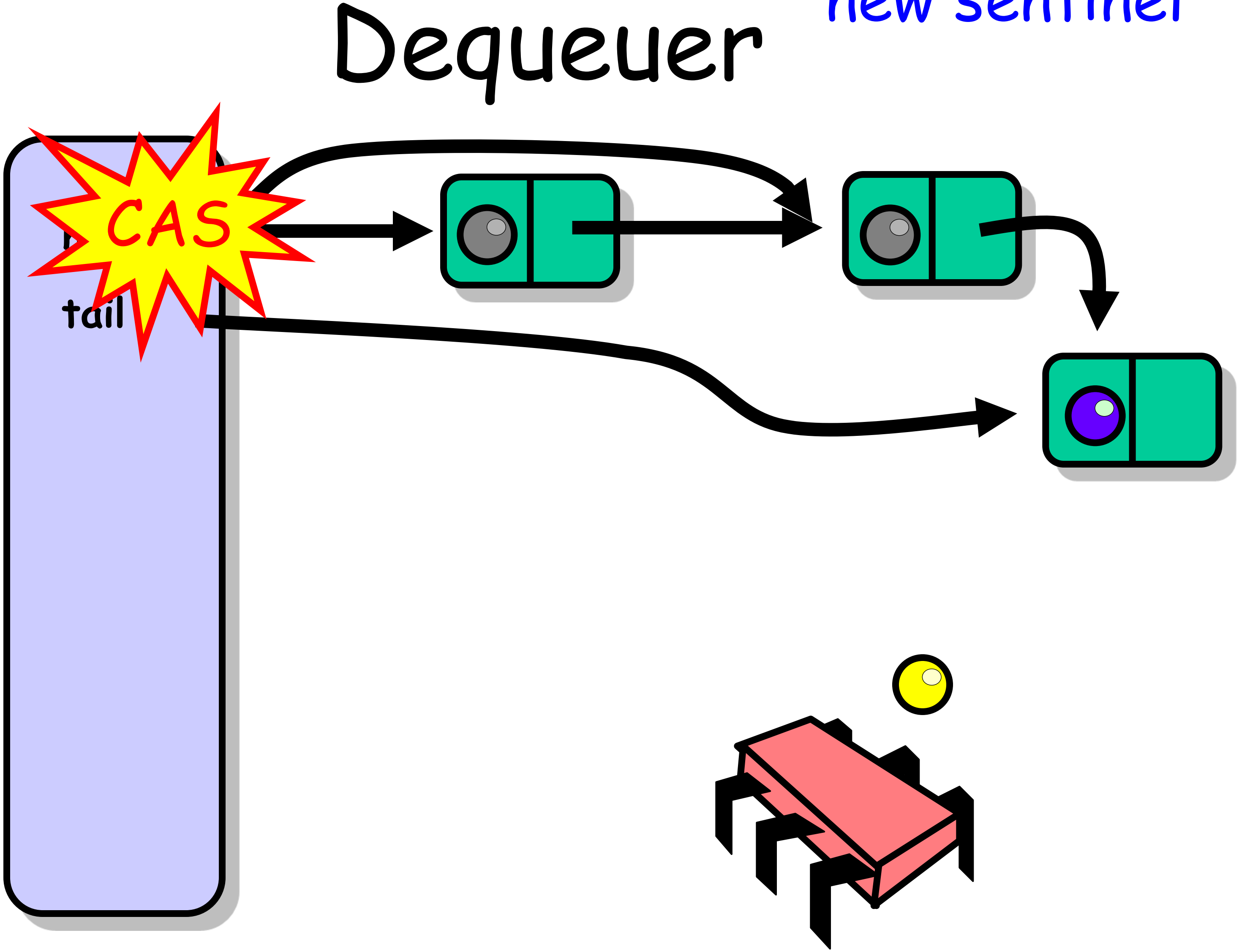
- During logical enqueue
  - Abandon hope, restart
  - Still lock-free (why?)
- During physical enqueue
  - Ignore it (why?)

Dequeuer



Read value

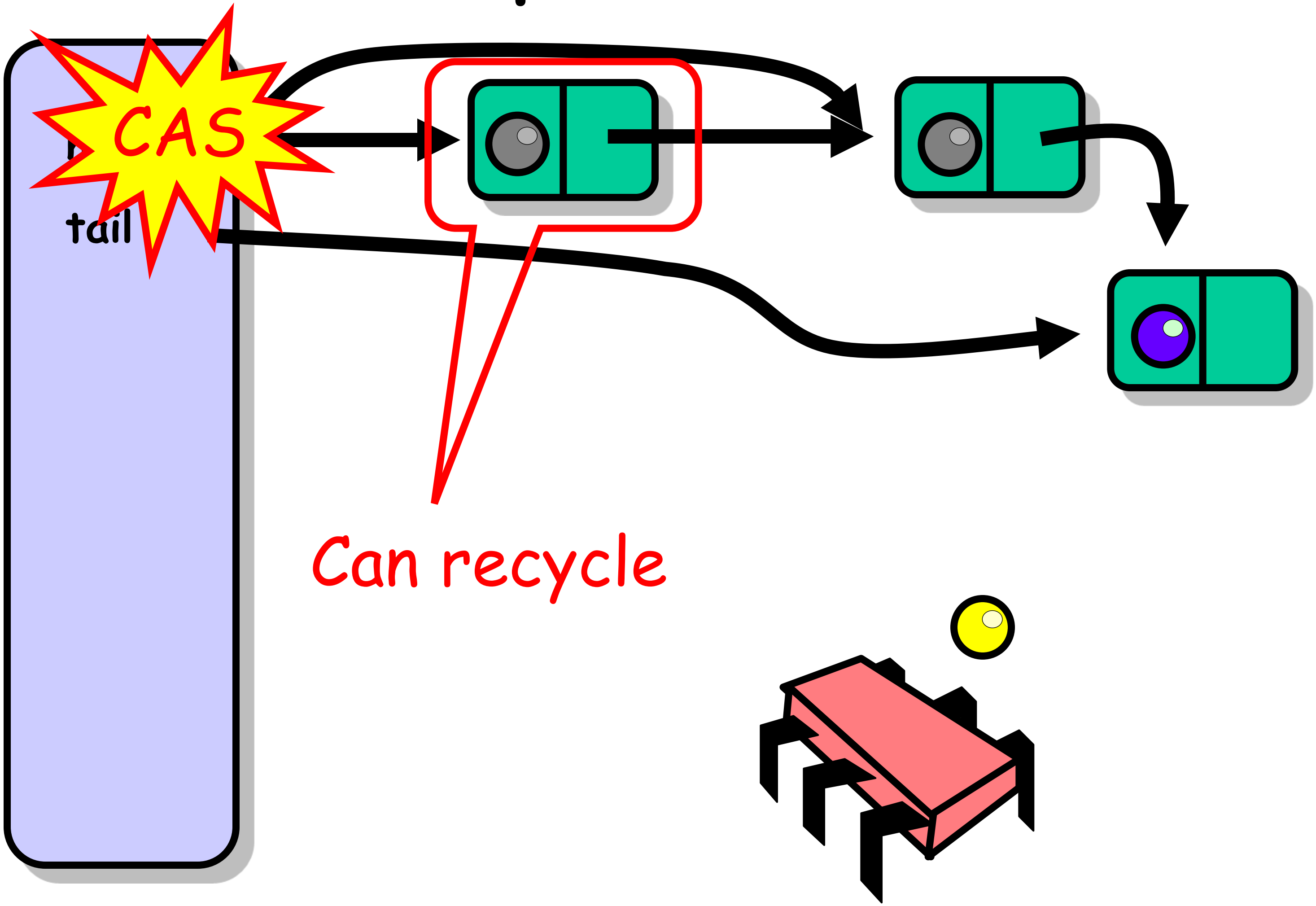
Make first Node new sentinel



Memory Reuse?

- What do we do with nodes after we dequeue them?
- Java: let garbage collector deal?
- Suppose there is no GC, or we prefer not to use it?

Dequeuer

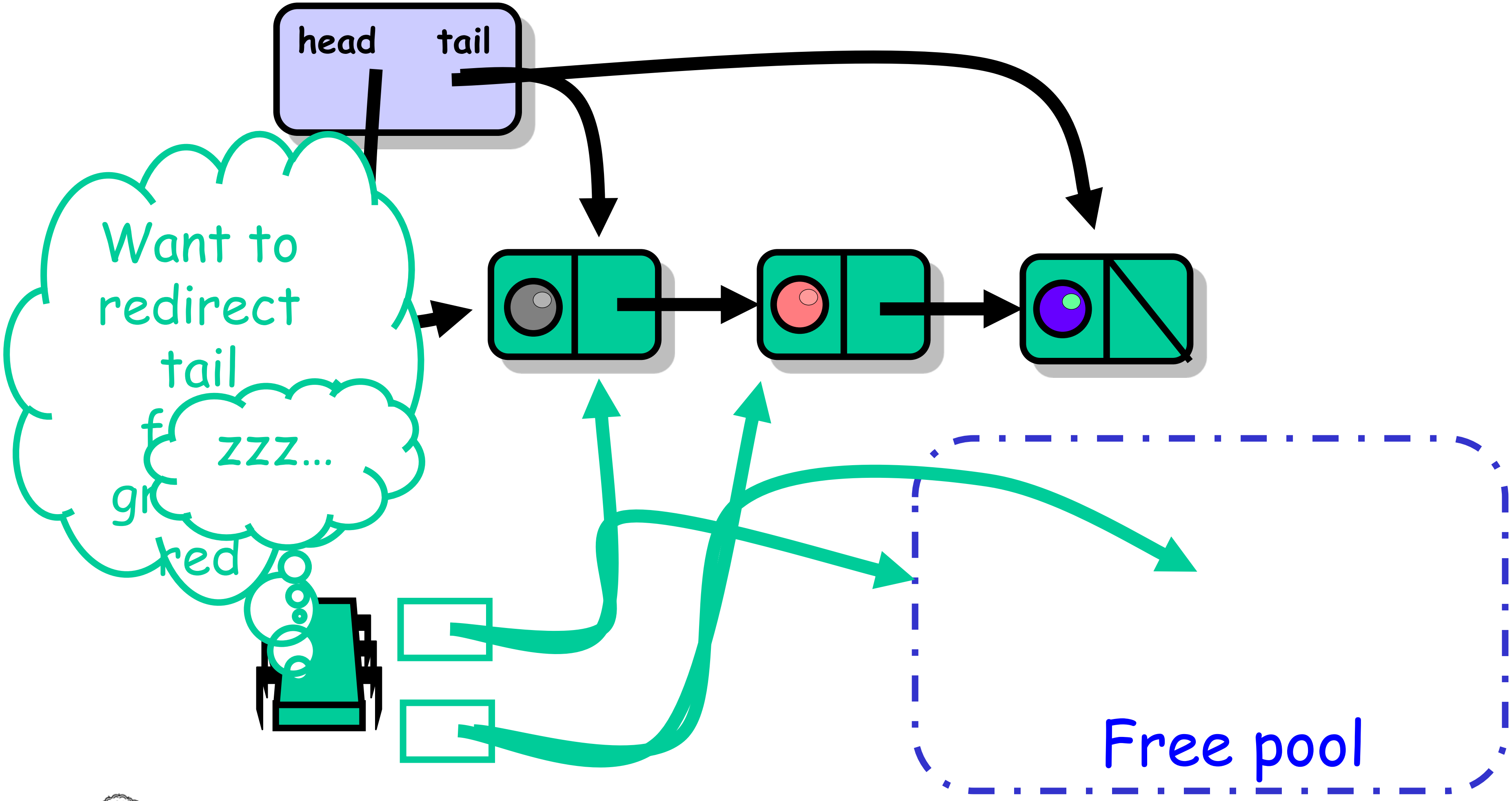


Can recycle

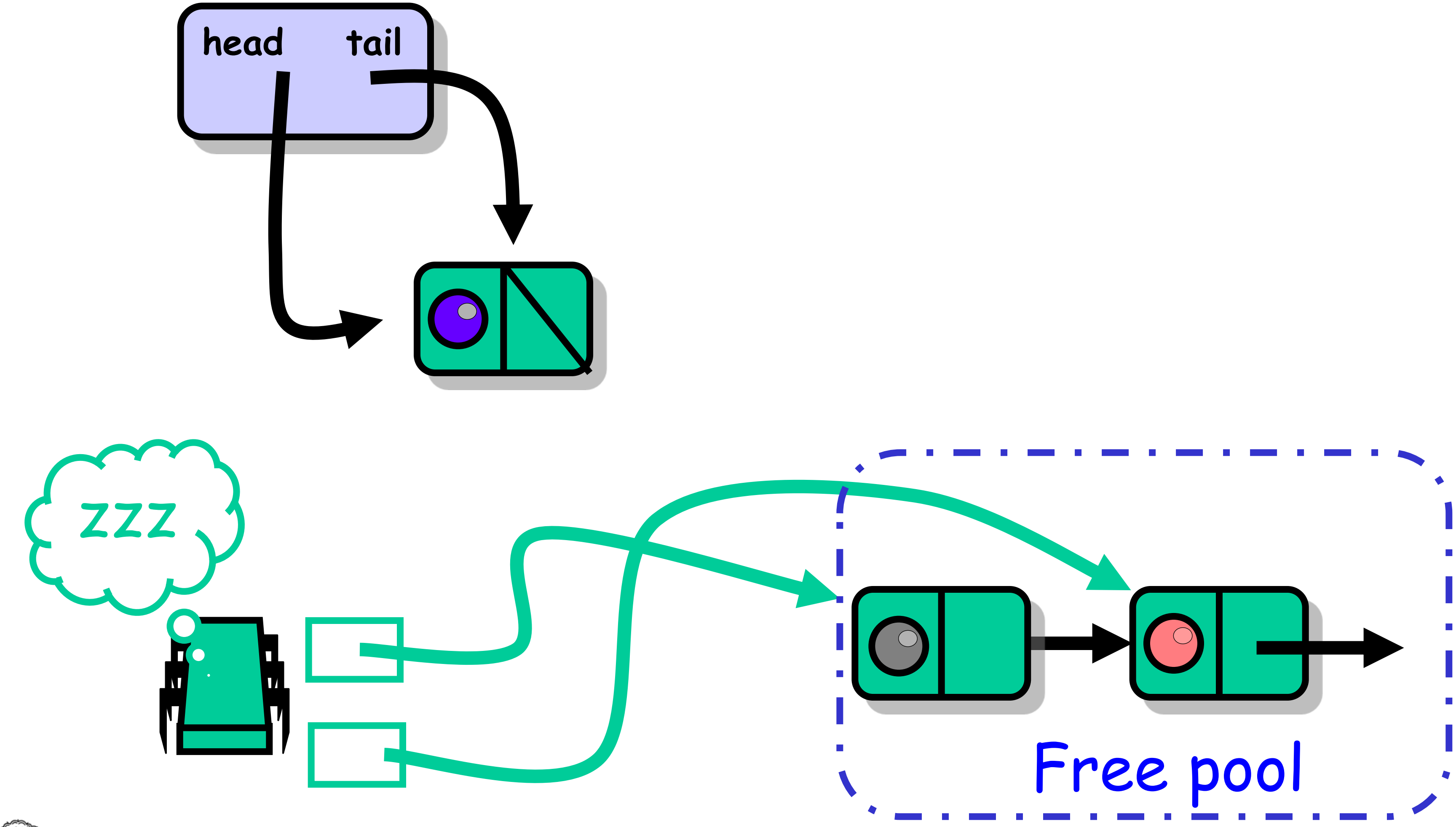
Simple Solution

- Each thread has a free list of unused queue nodes
- Allocate node: pop from list
- Free node: push onto list
- Deal with underflow somehow …

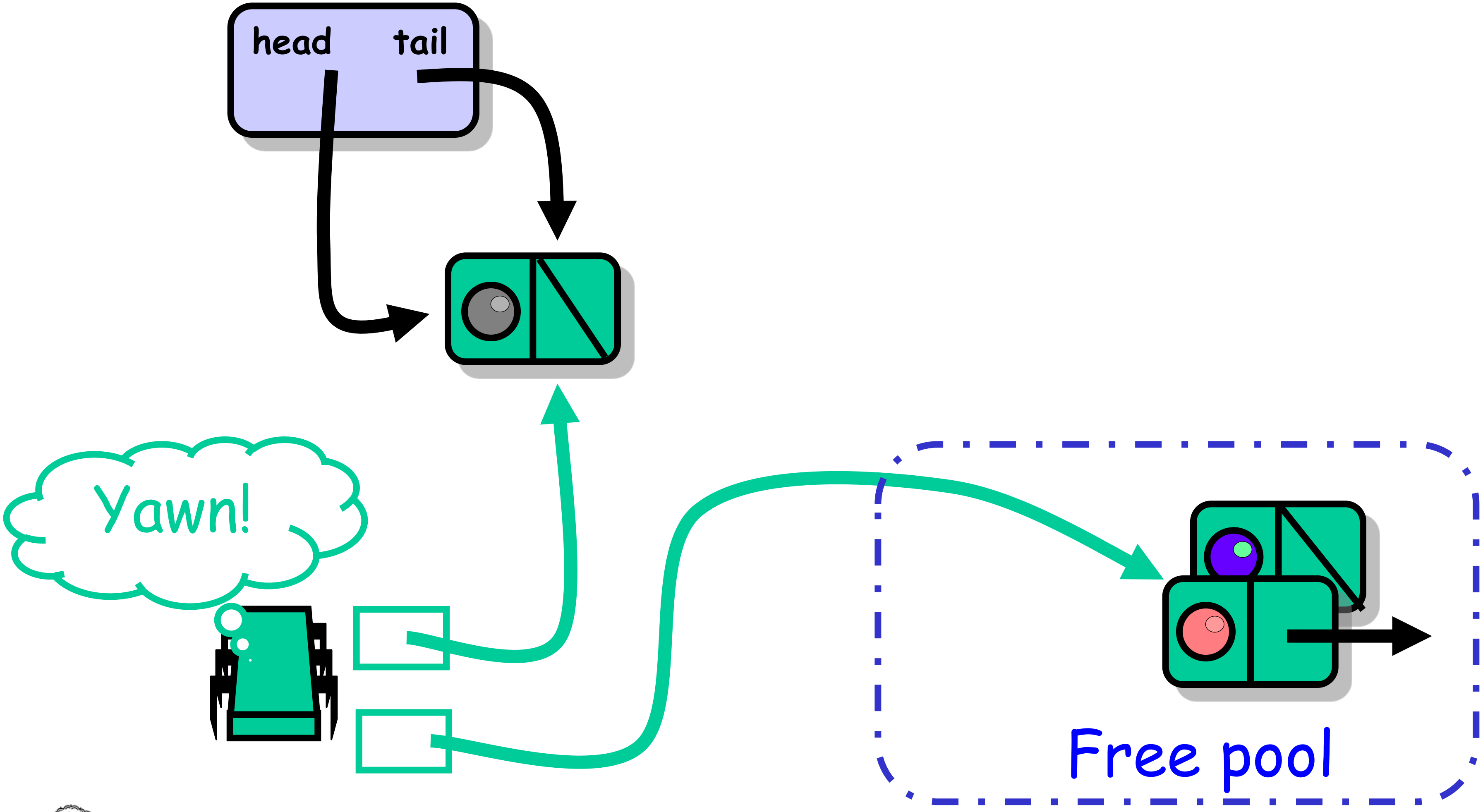
Why Recycling is Hard



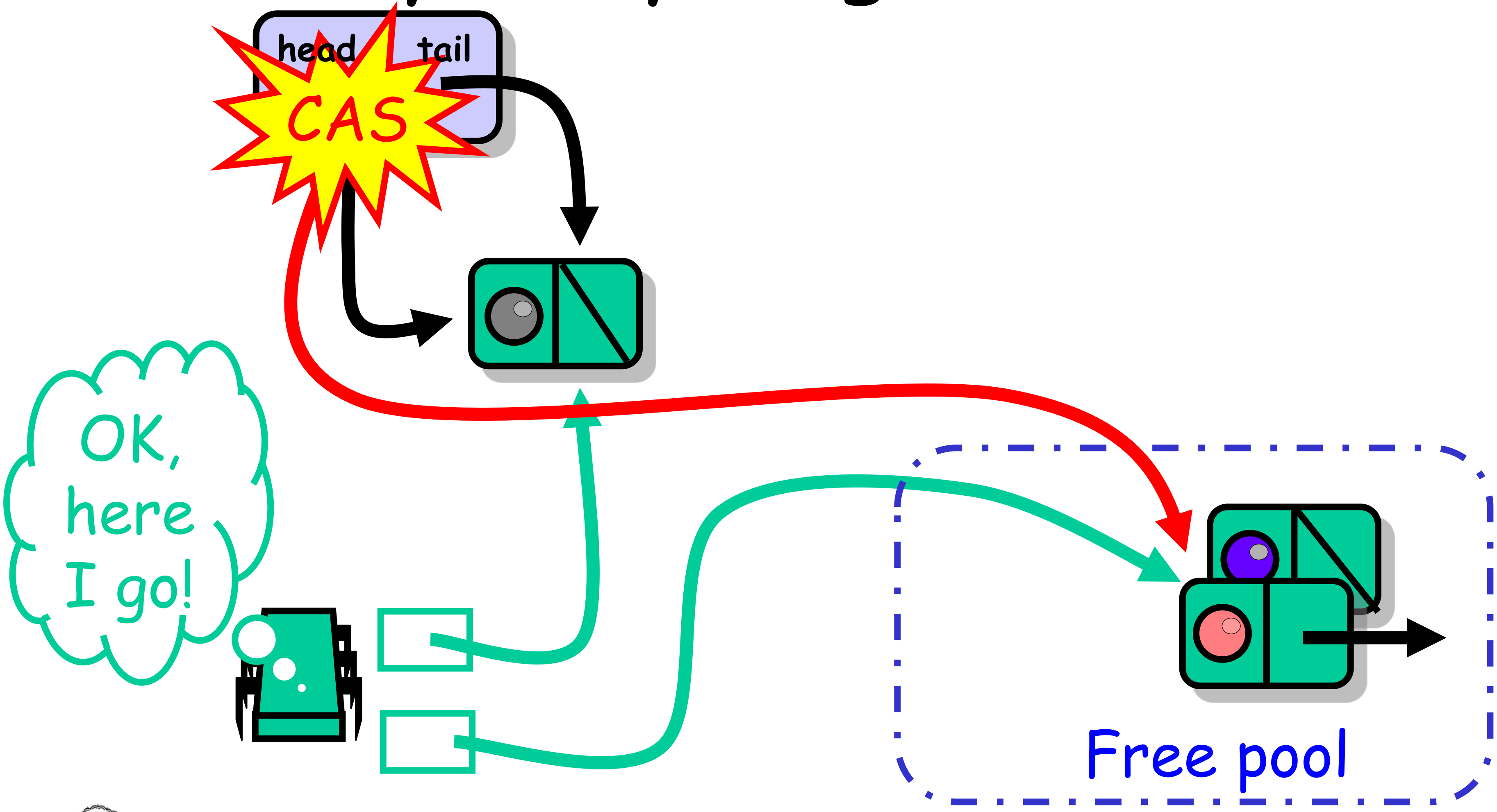
Both Nodes Reclaimed



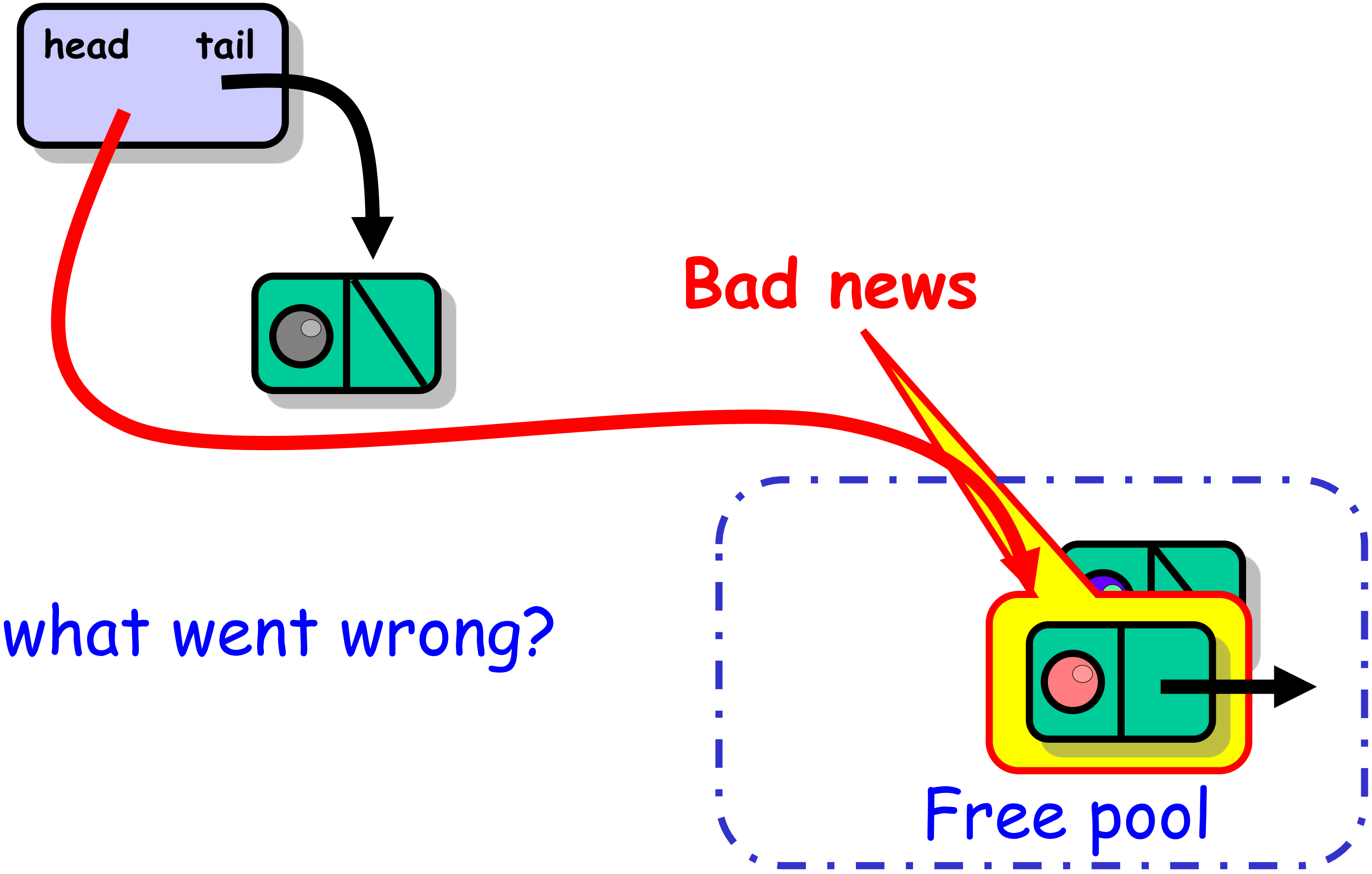
One Node Recycled



Why Recycling is Hard

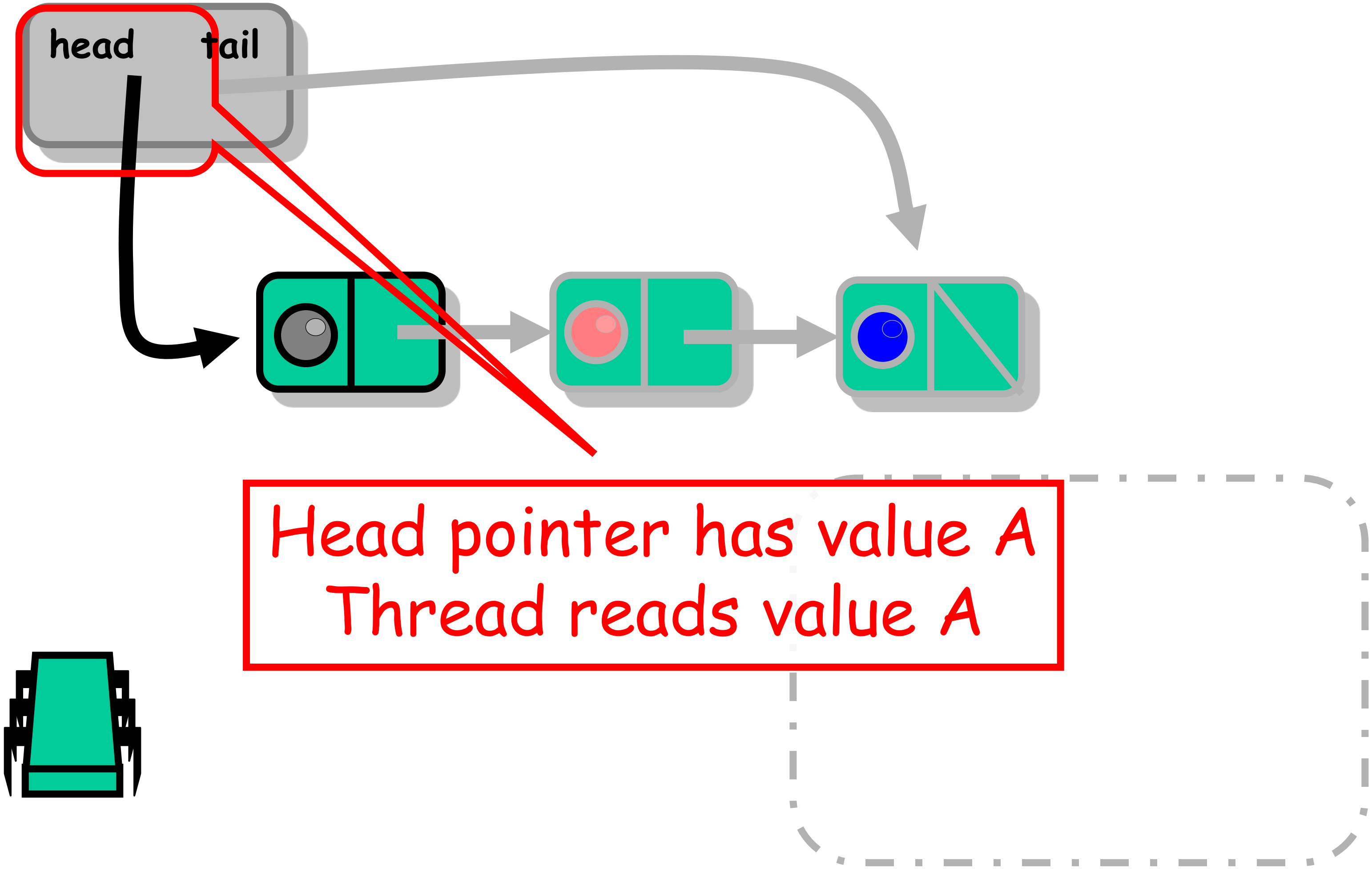


Final State



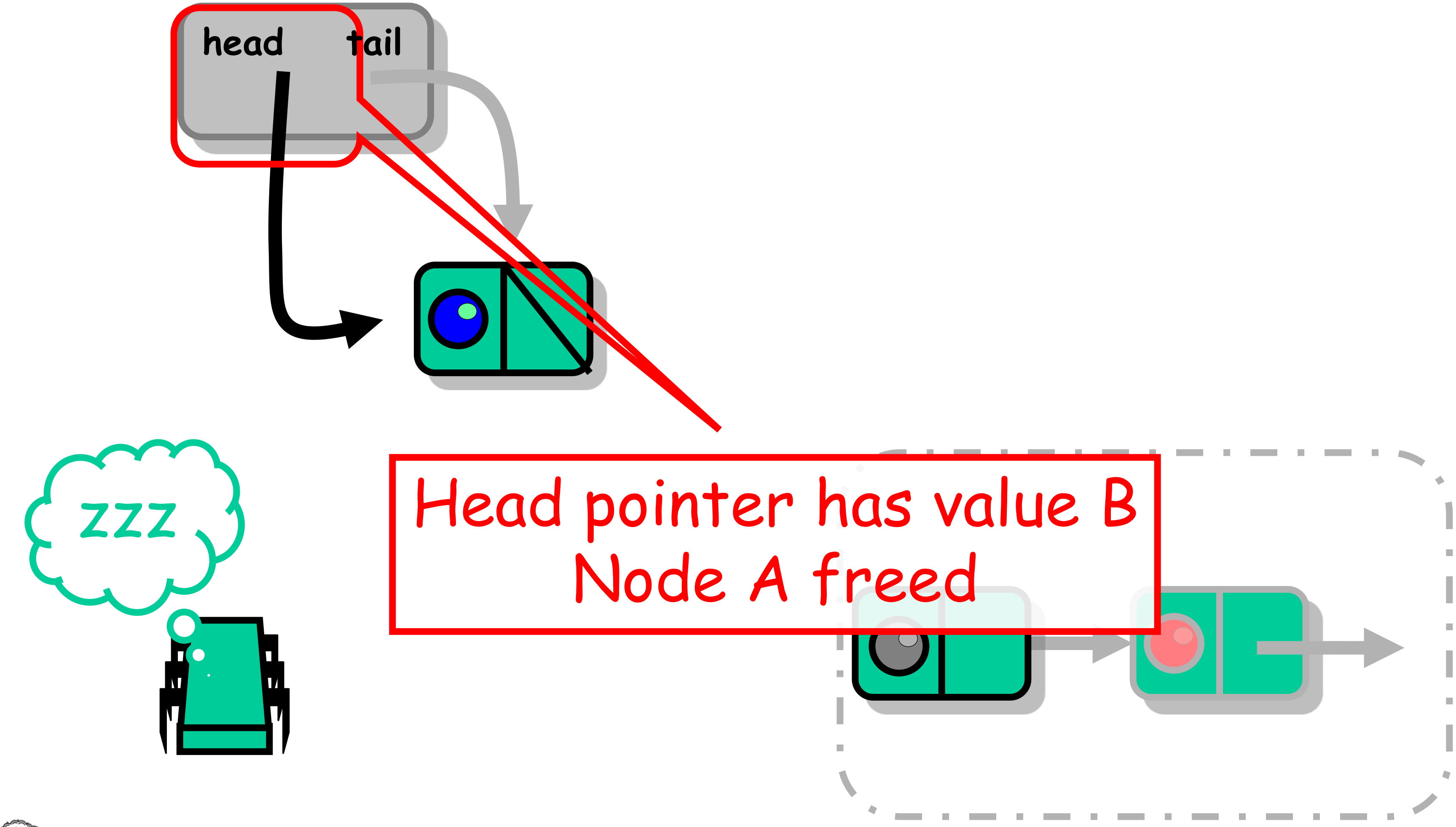
zOMG what went wrong?

The Dreaded ABA Problem



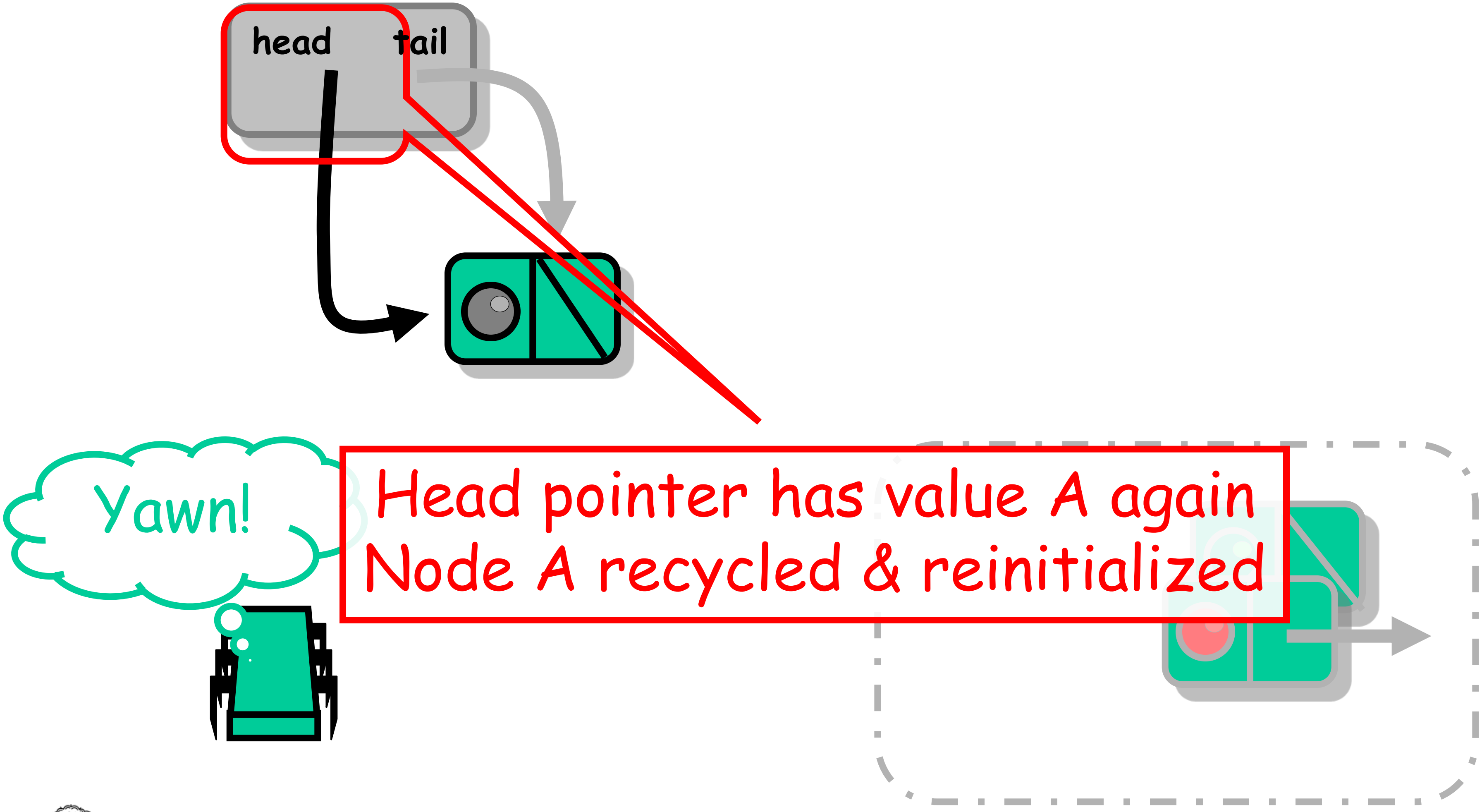
Head pointer has value A
Thread reads value A

Dreaded ABA continued



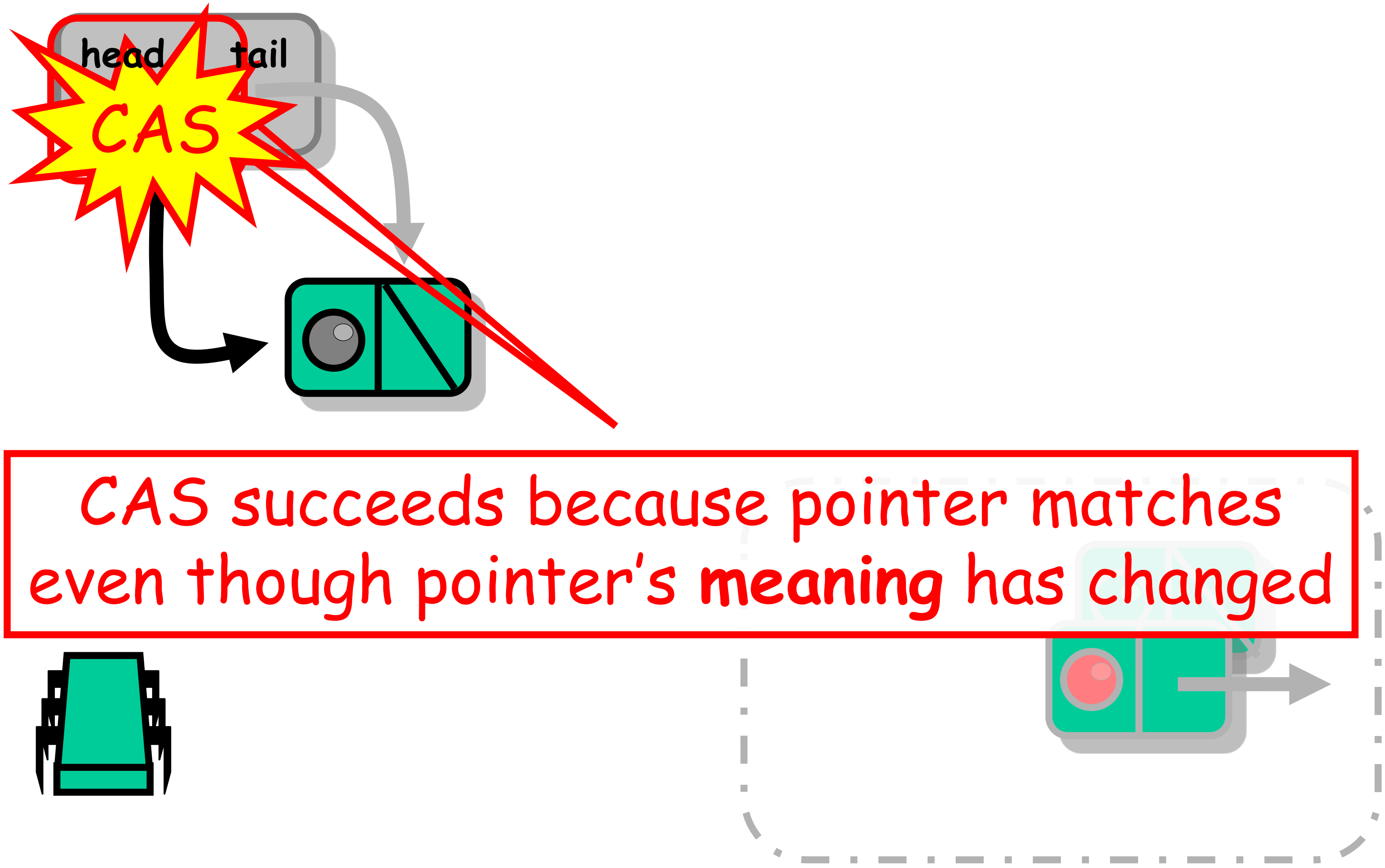
Head pointer has value B
Node A freed

Dreaded ABA continued



Head pointer has value A again
Node A recycled & reinitialized

Dreaded ABA continued



CAS succeeds because pointer matches even though pointer's **meaning** has changed

The Dreaded ABA Problem

- Is a result of CAS() semantics
  - I blame Sun, Intel, AMD, …
- Not with Load-Locked/Store-Conditional
  - Good for IBM?

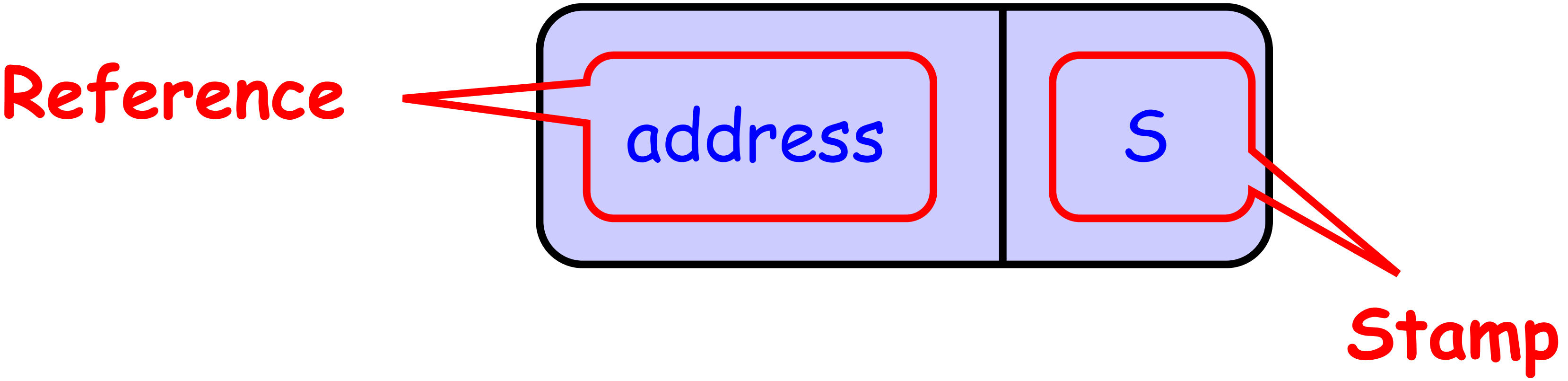
Dreaded ABA – A Solution

- Tag each pointer with a counter
- Unique over lifetime of node
- Pointer size vs word size issues
- Overflow?
  - Don't worry be happy?
  - Bounded tags?
- AtomicStampedReference class

Atomic Stamped Reference

- AtomicStampedReference class
  - – Java.util.concurrent.atomic package

Can get reference and stamp atomically, details soon



Summary

- We saw both lock-based and lock-free implementations of
- queues
- Don't be quick to declare a data structure inherently sequential
  - Linearizable stack is not inherently sequential
- ABA is a real problem, pay attention