YSC4231: Parallel, Concurrent and Distributed Programming

Concurrent Queues and the ABA Problem

The Five-Fold Path

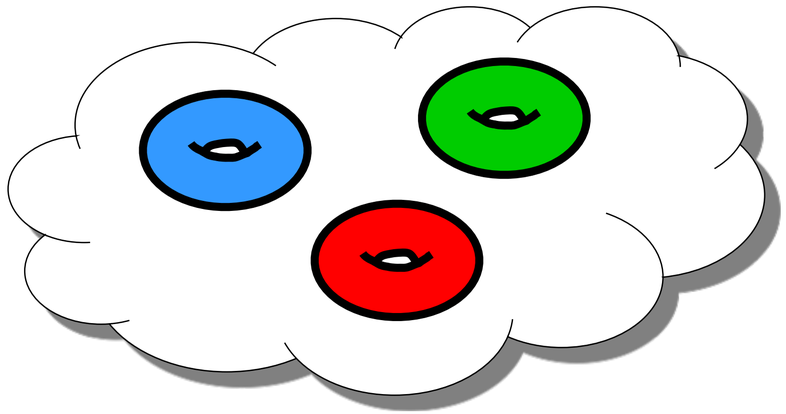
- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization
- Lazy synchronization
- Lock-free synchronization (a glimpse of)

Another Fundamental Problem

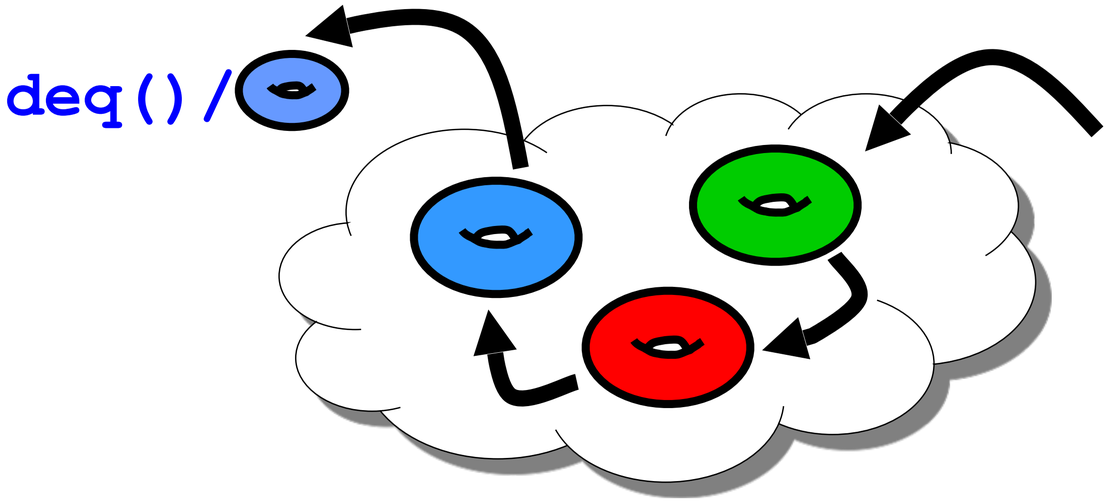
- We learned about
  - Sets implemented by linked lists
- Next: queues
- After that: stacks

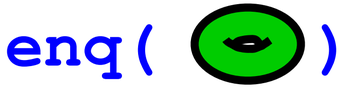
Queues & Stacks

- pool of items



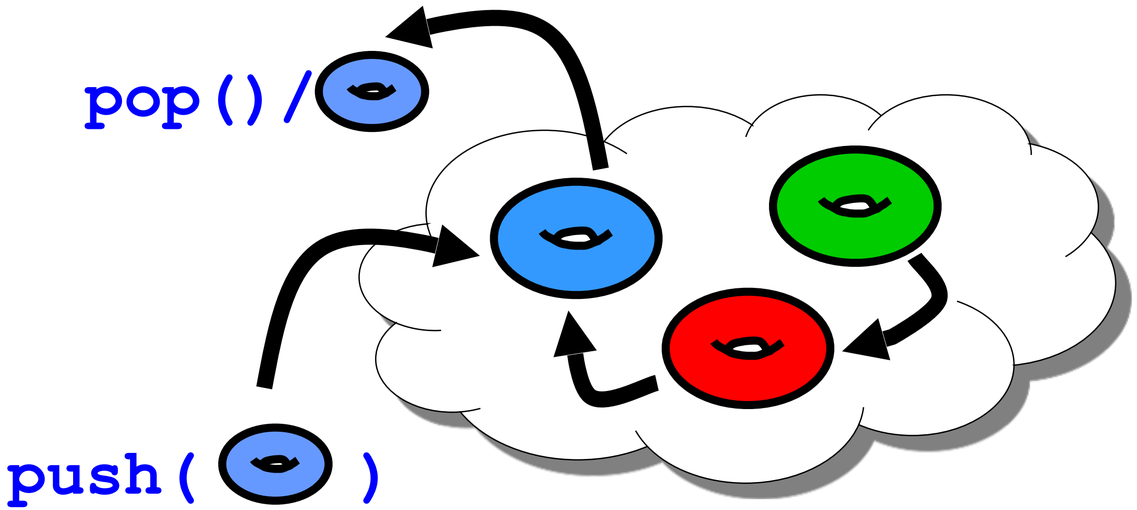
Queues





Total order
First in
First out

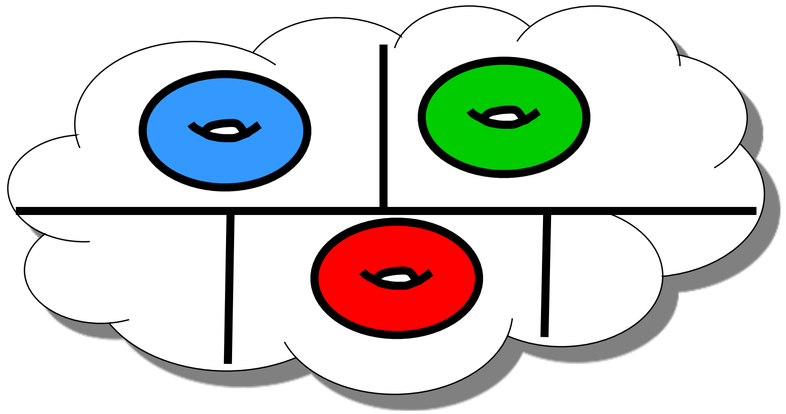
Stacks



Total order
Last in
First out

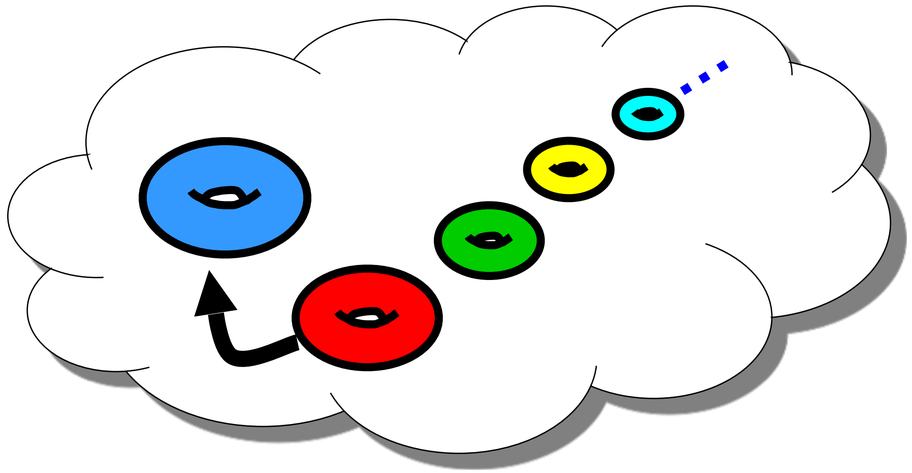
Bounded

- Fixed capacity
- Good when resources an issue

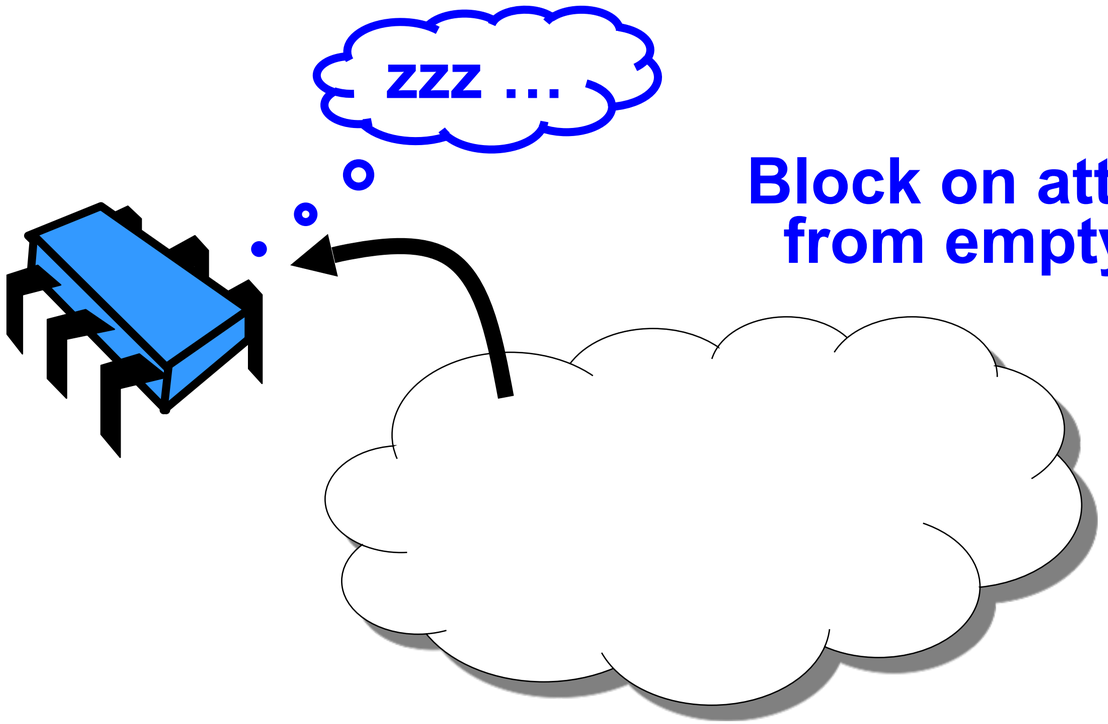


Unbounded

- Unlimited capacity
- Often more convenient

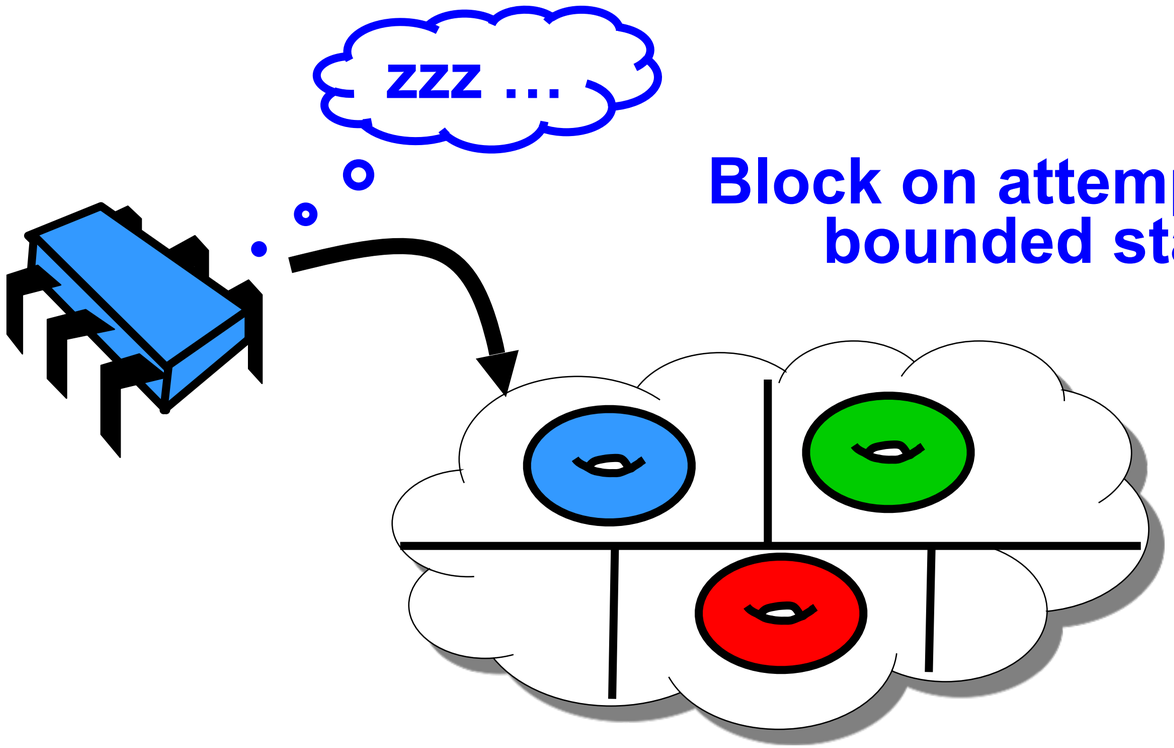


Blocking



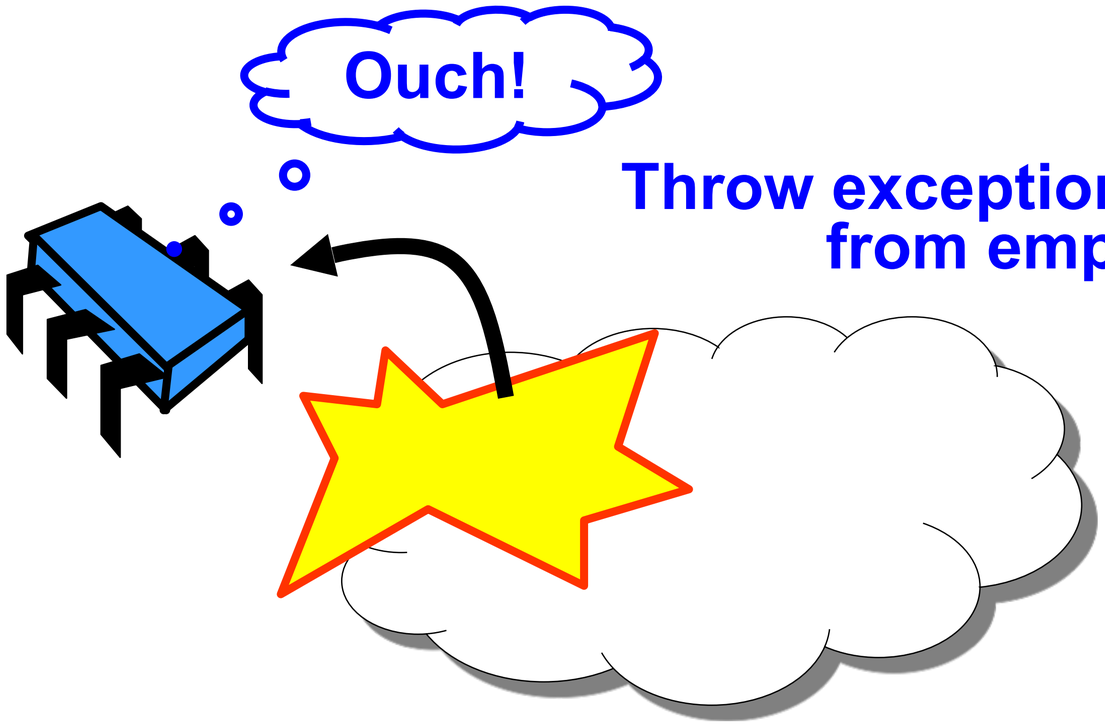
Block on attempt to remove from empty stack or queue

Blocking



Block on attempt to add to full bounded stack or queue

Non-Blocking



Throw exception on attempt to remove from empty stack or queue

This Lecture

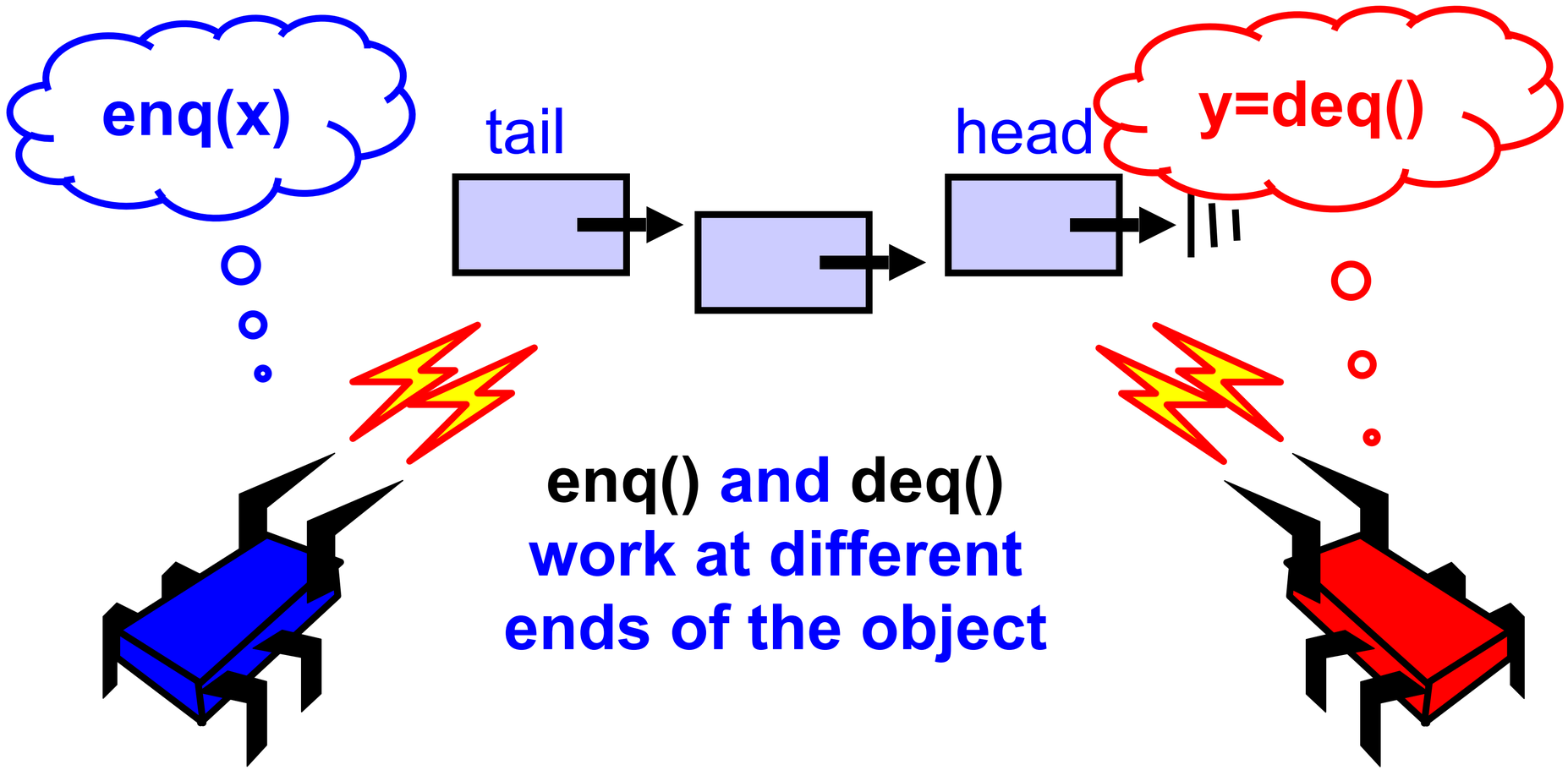
- Queue
  - Bounded, blocking, lock-based
  - Unbounded, non-blocking, lock-free
- Stack
  - Unbounded, non-blocking lock-free
  - Elimination-backoff algorithm

This Lecture

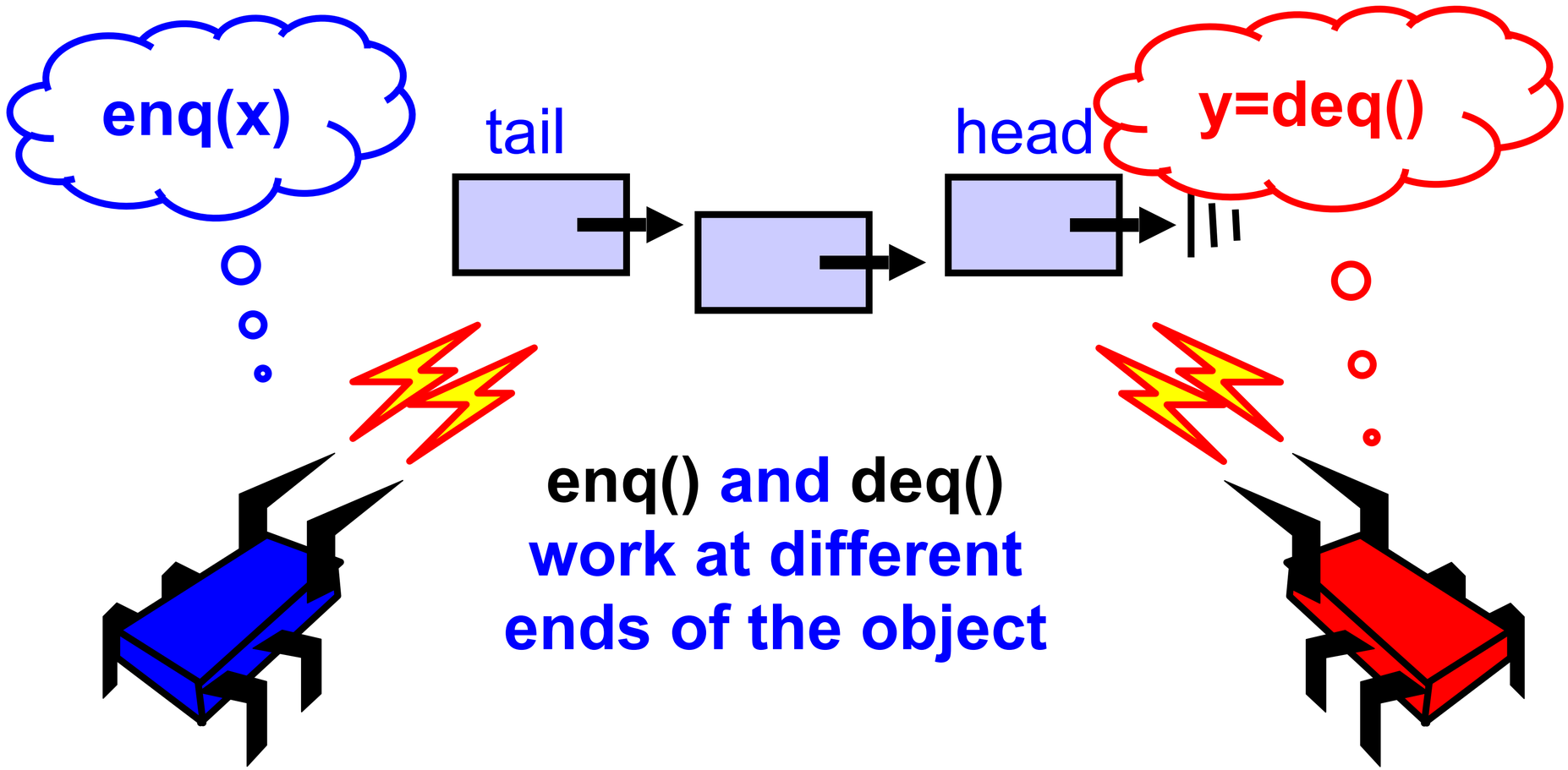
- Queue
  - Bounded, blocking, lock-based
  - Unbounded, non-blocking, lock-free
- Stack
  - Unbounded, non-blocking lock-free
  - Elimination-backoff algorithm

Warm-up (coding):
Coarse-Grained Queues and Tests

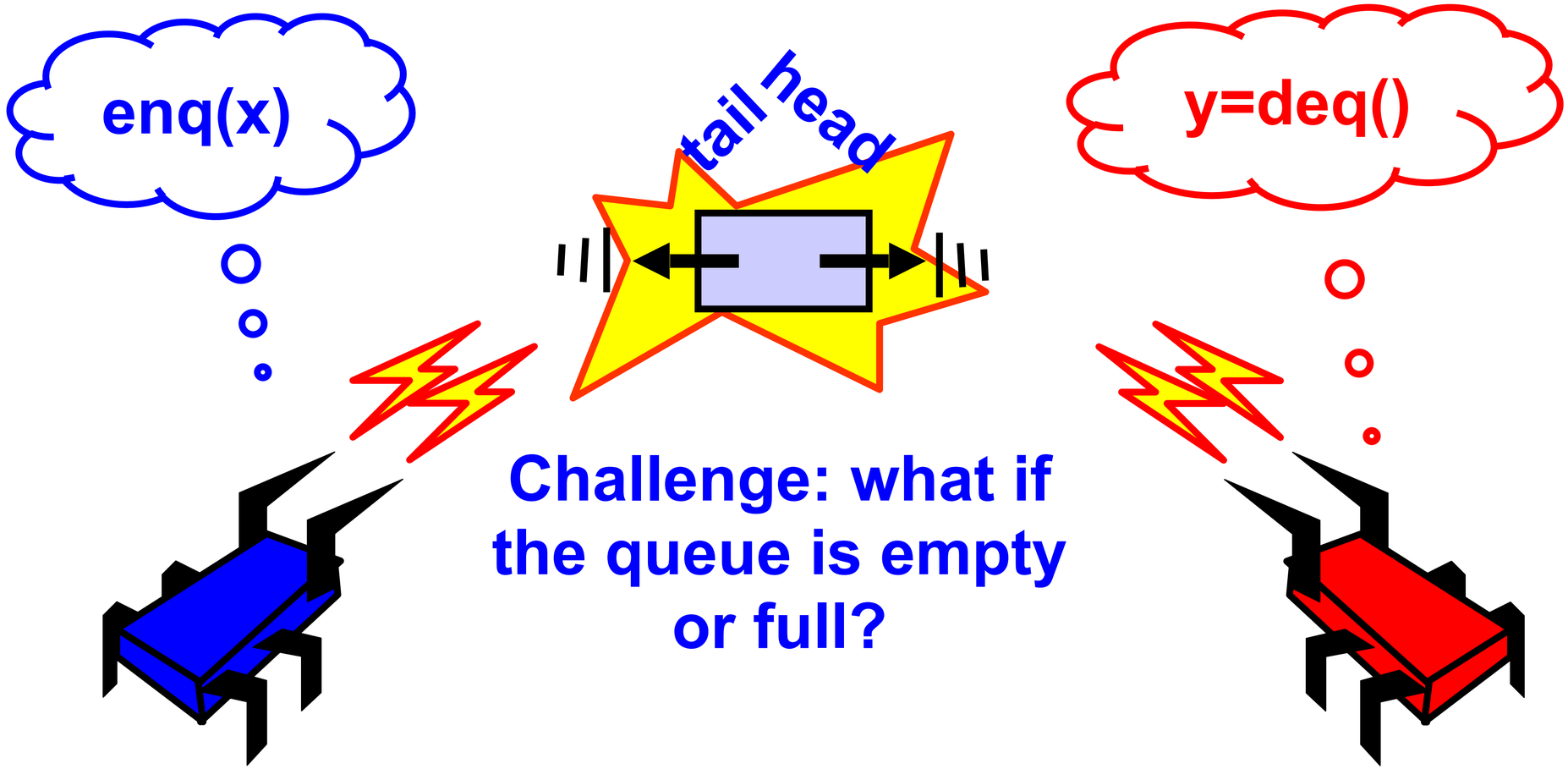
Queue: Concurrency



Queue: Concurrency



Concurrency

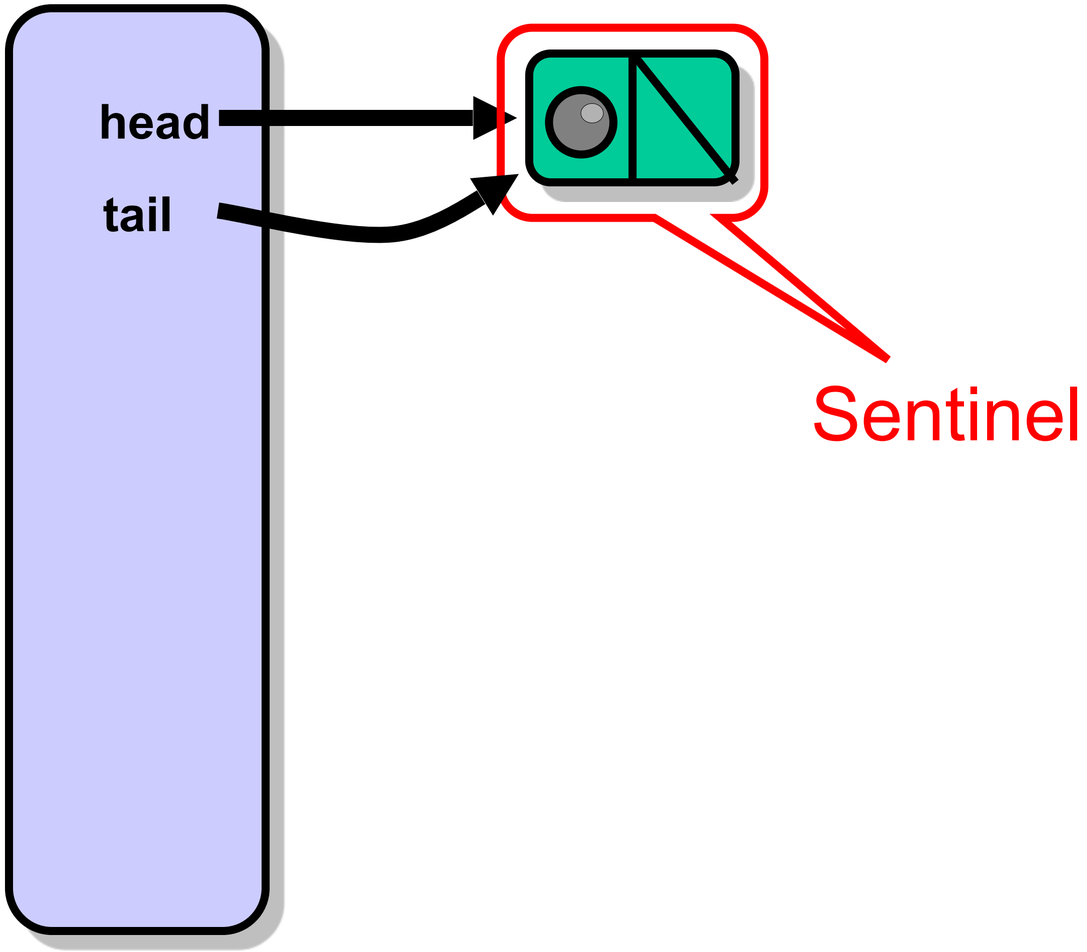


Challenge: what if the queue is empty or full?

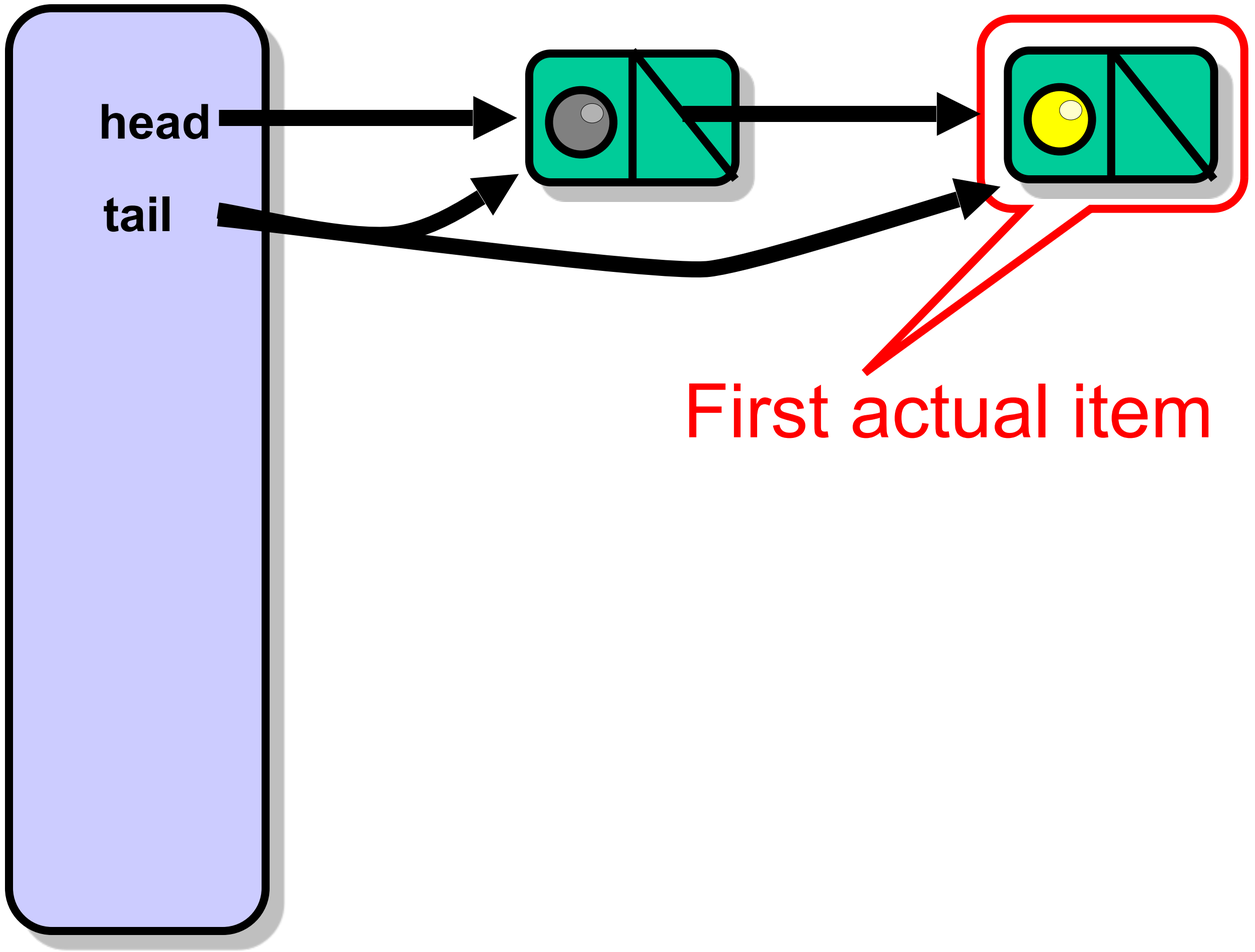
Let's Look at the Code:
Analysing Unbounded Queue

Bounded Queue

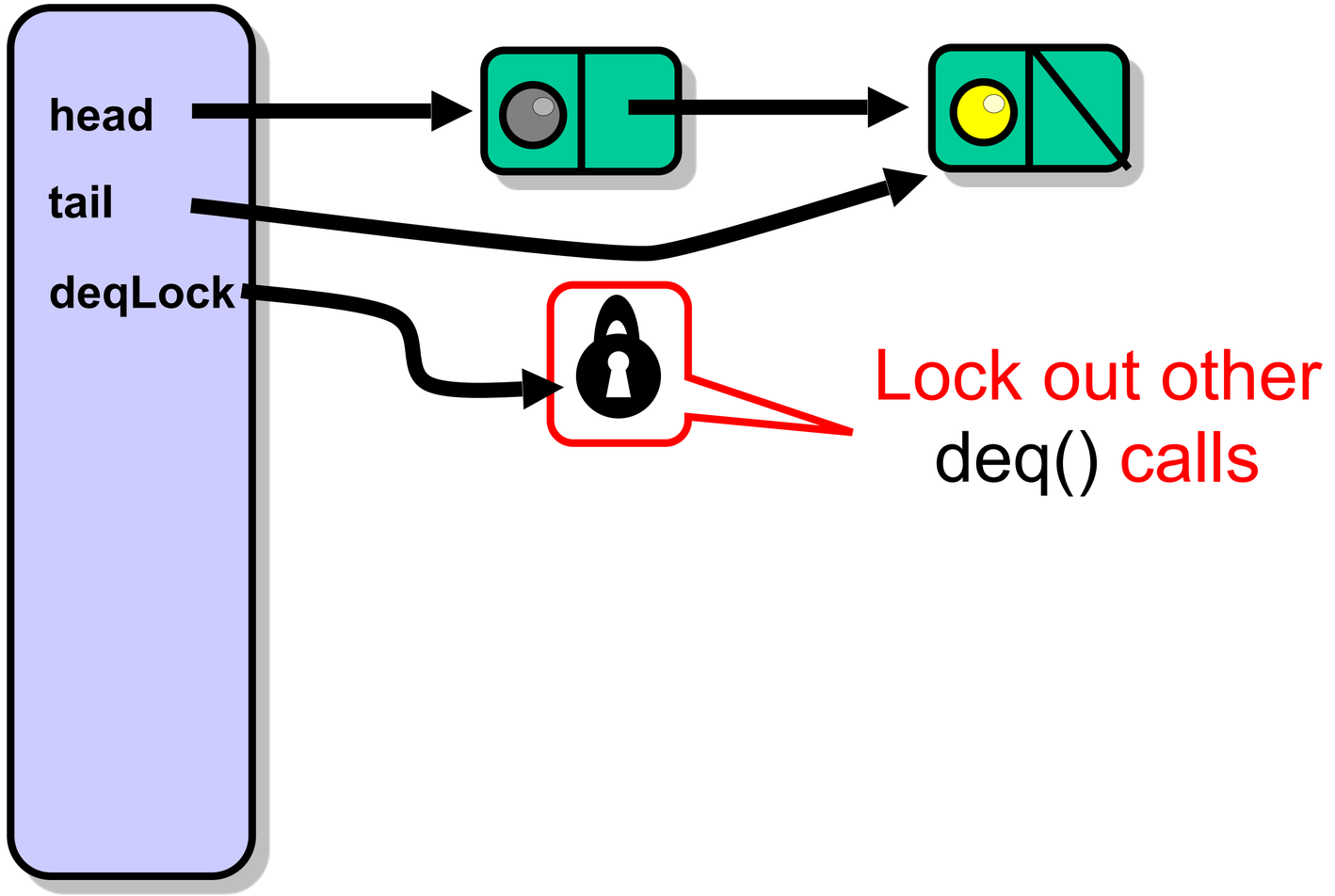
Bounded Queue



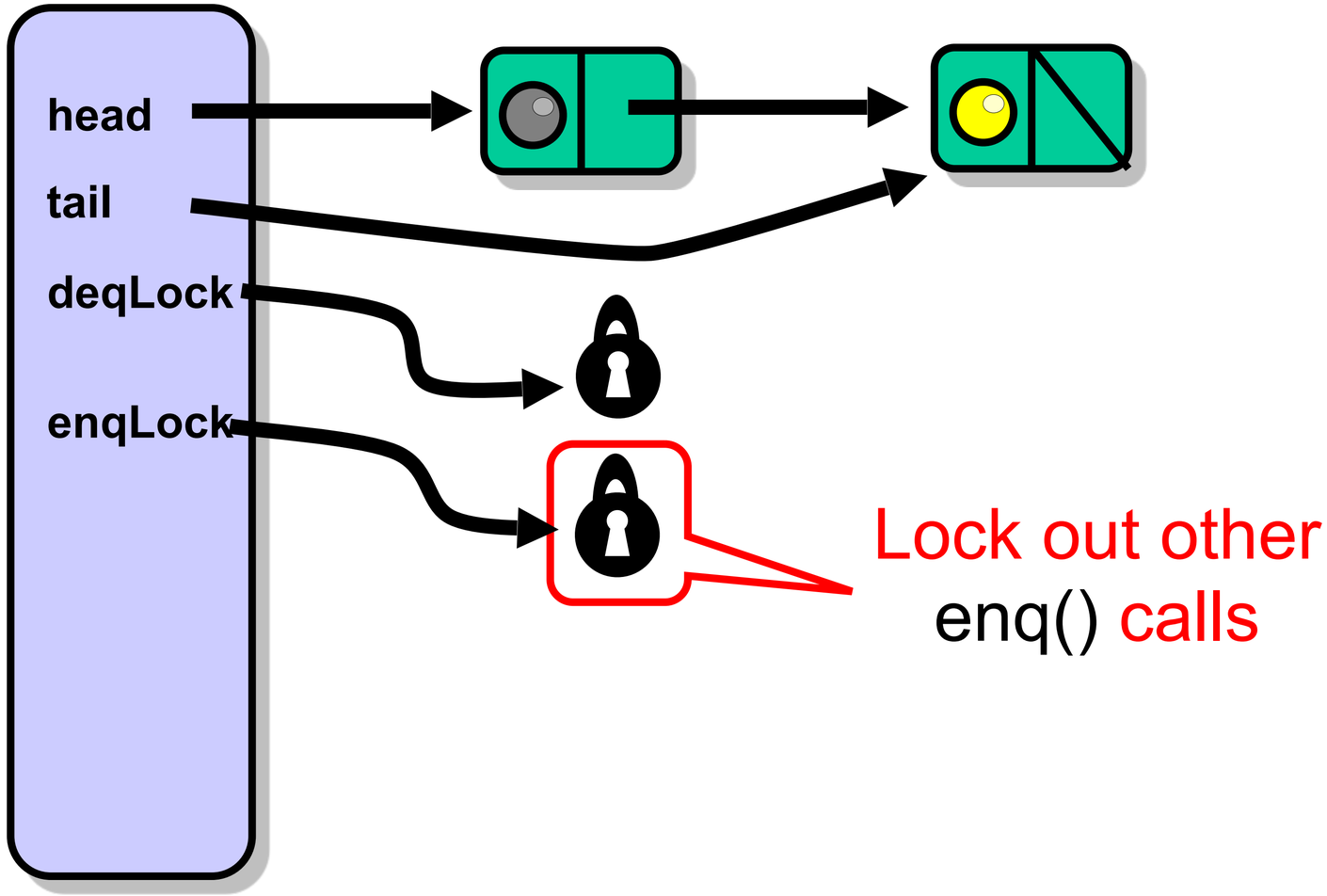
Bounded Queue



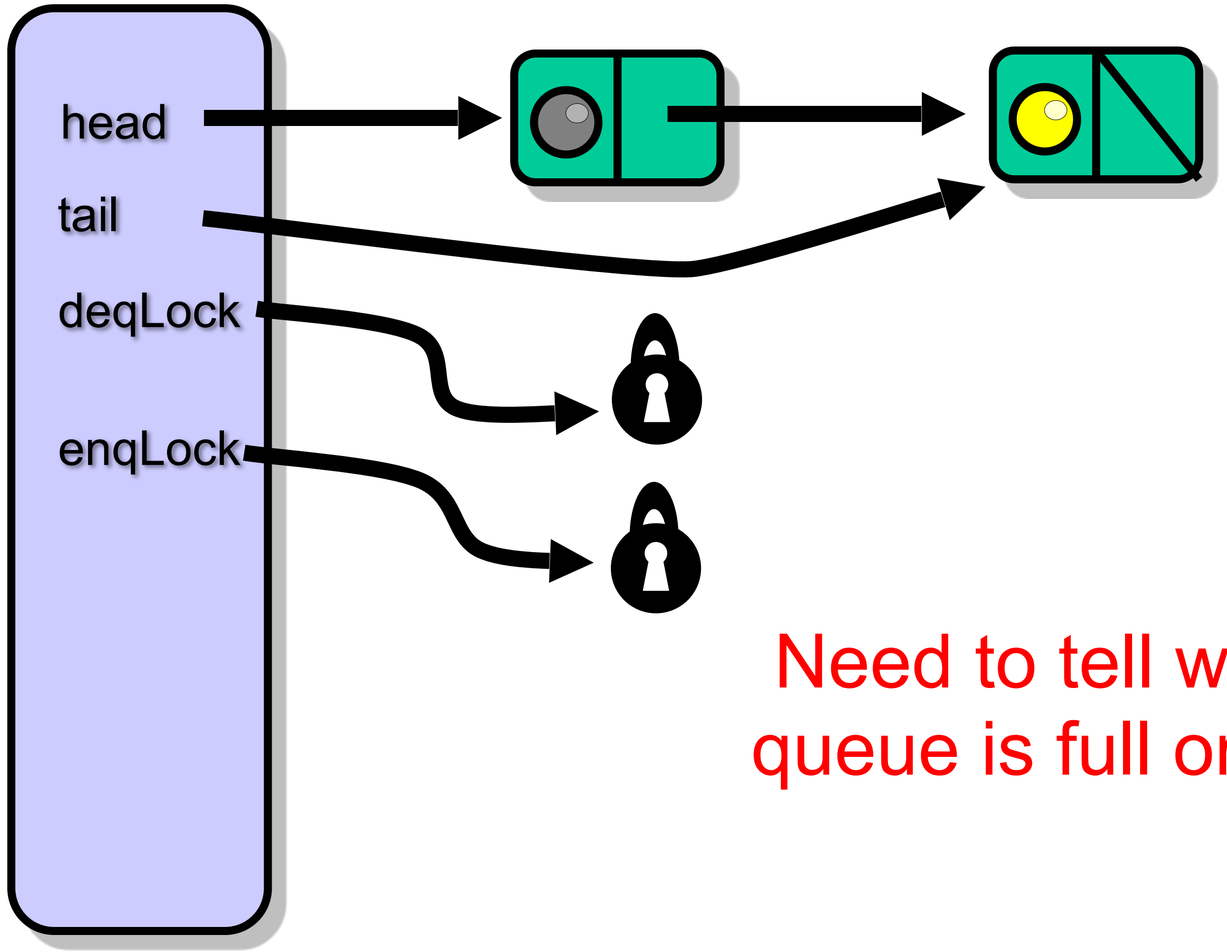
Bounded Queue



Bounded Queue

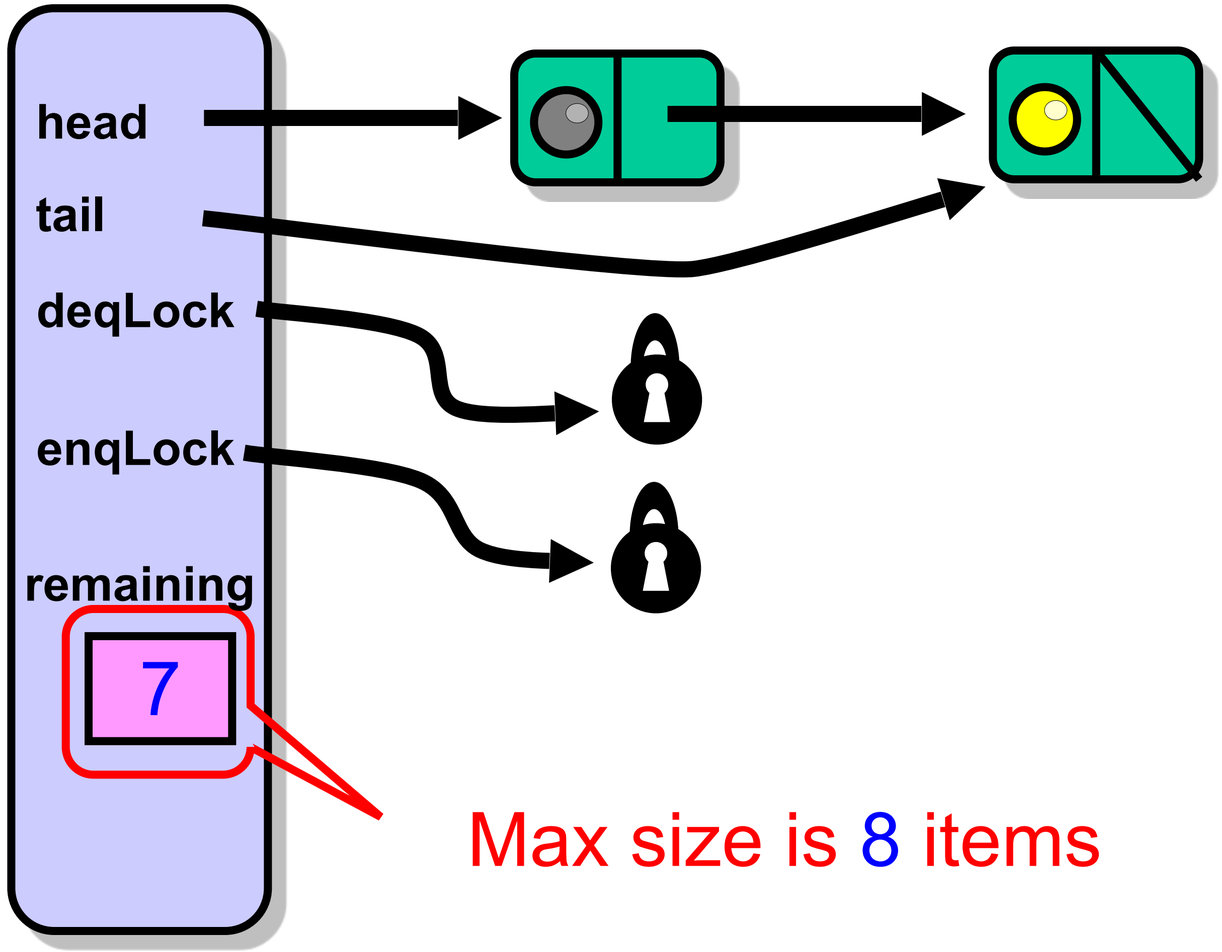


Not Done Yet

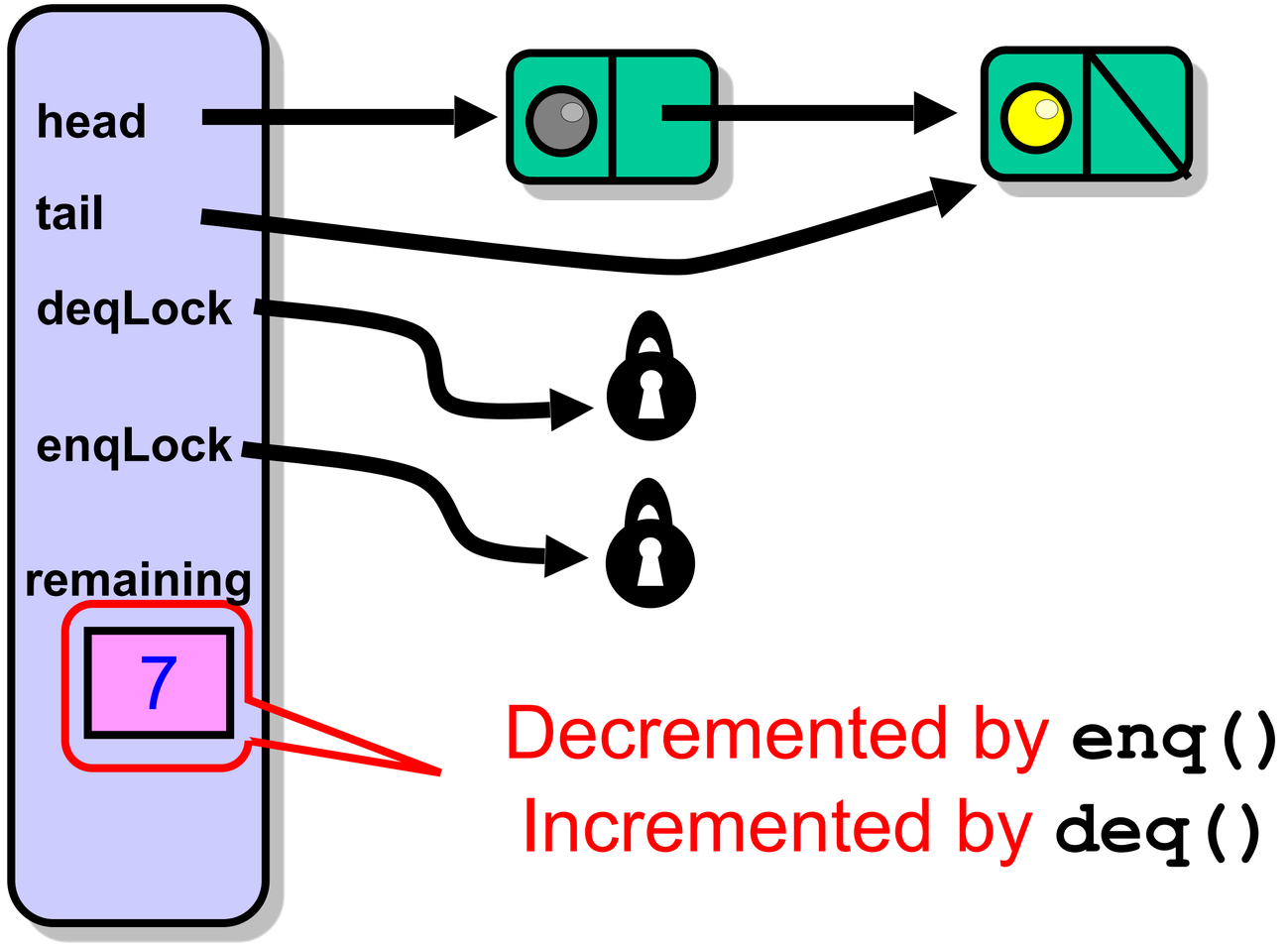


Need to tell whether queue is full or empty

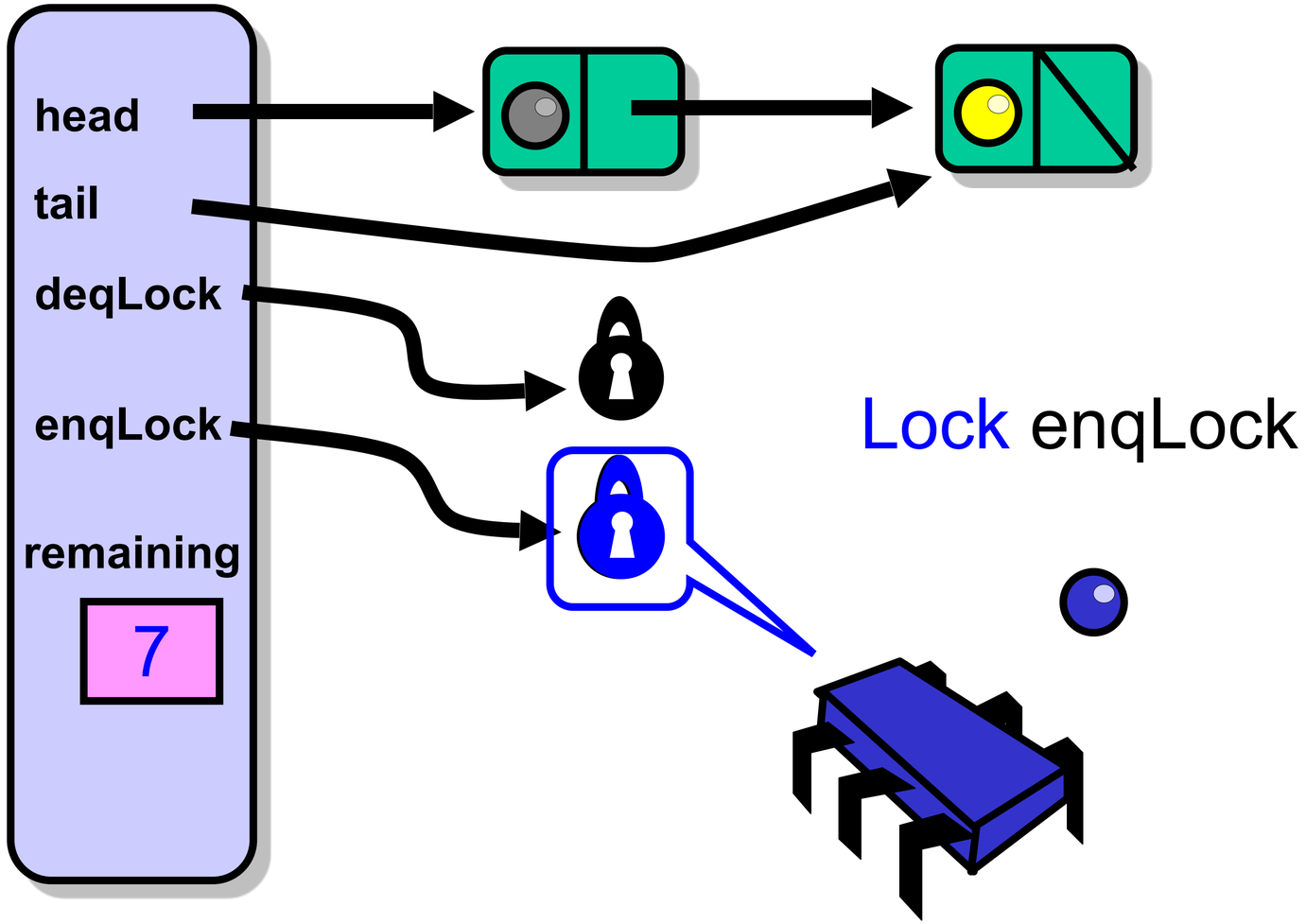
Not Done Yet



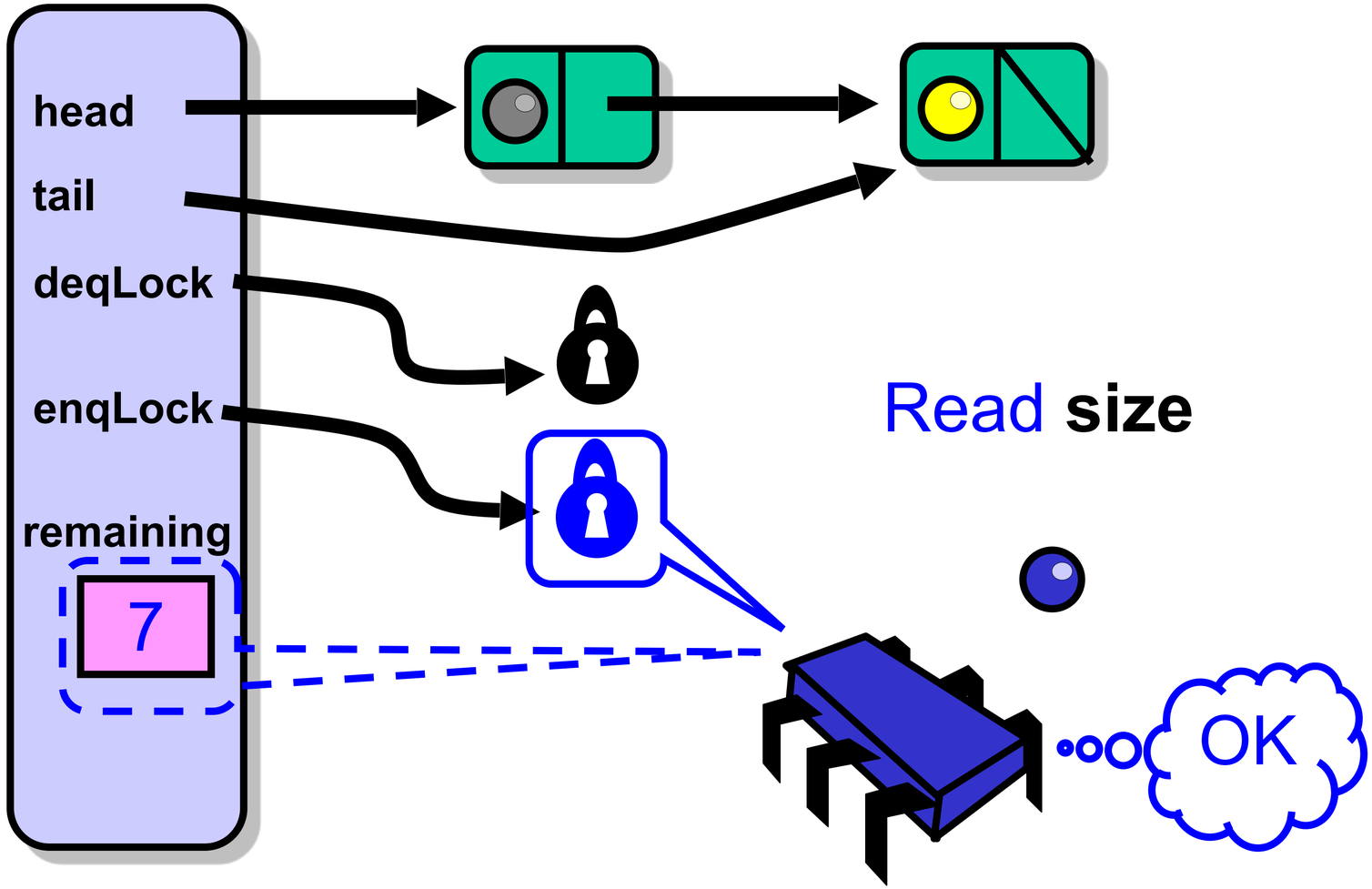
Not Done Yet


head
tail
deqLock
enqLock
remaining
7
Decremented by enq()
Incremented by deq()

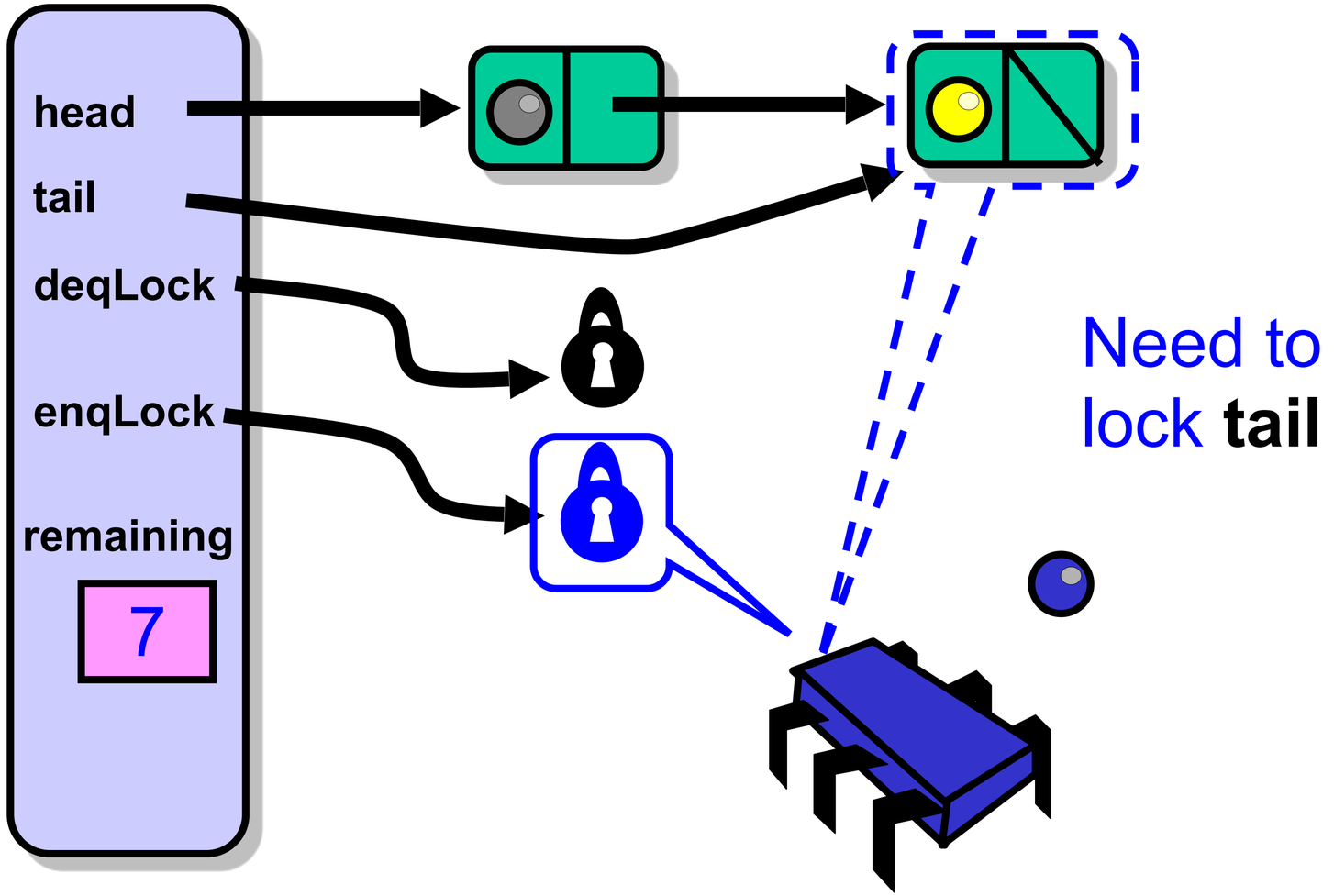
Enqueuer



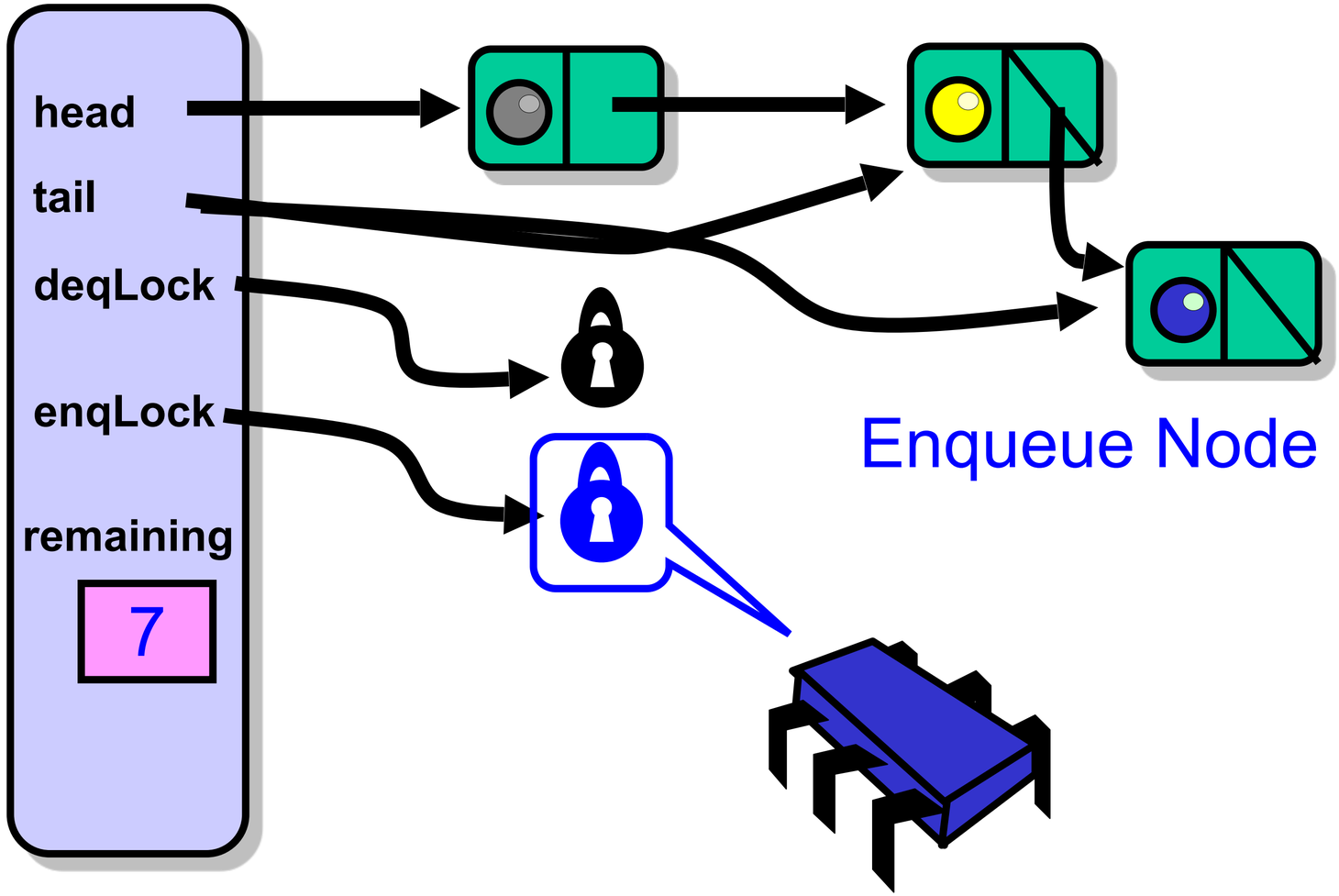
Enqueuer



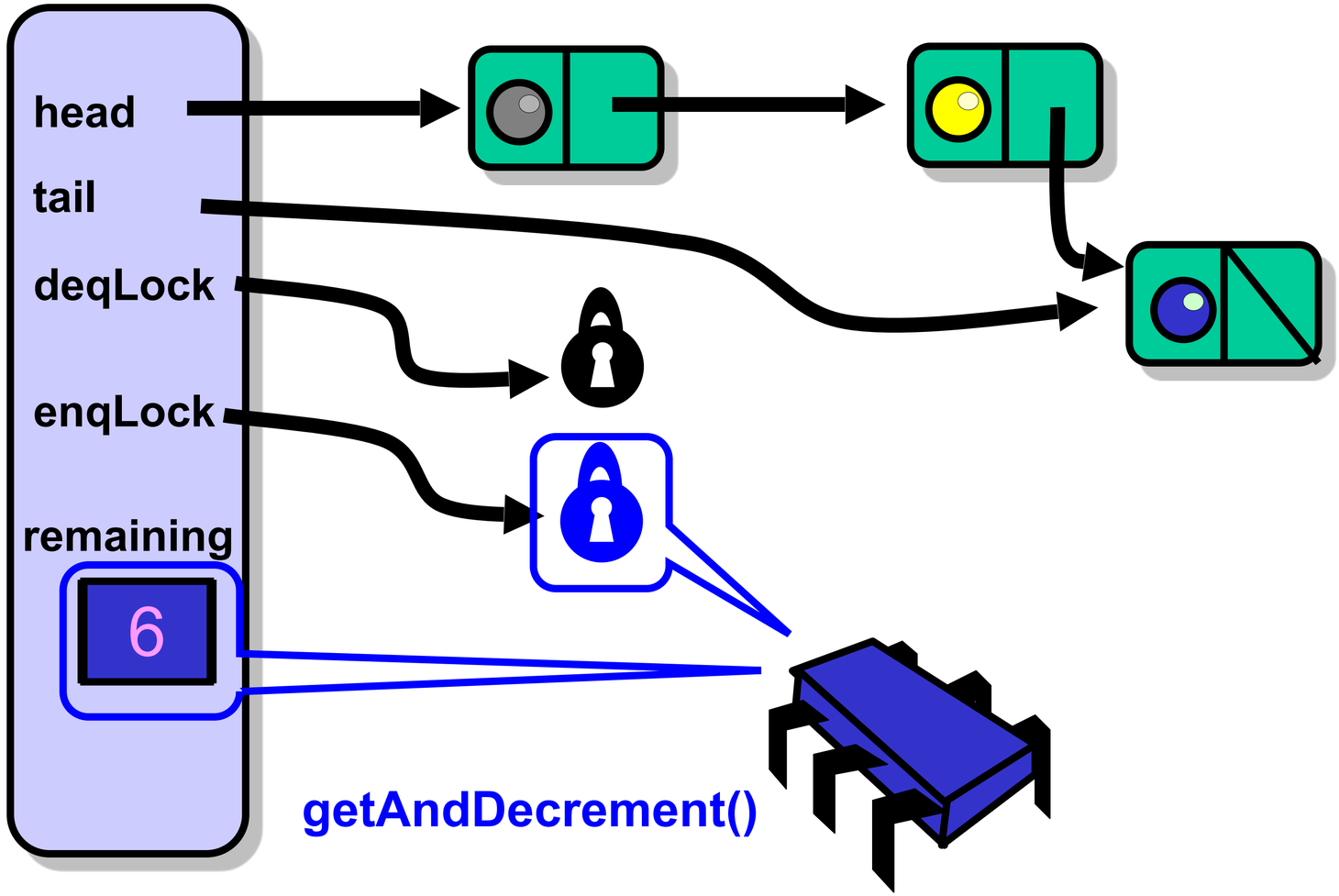
Enqueuer



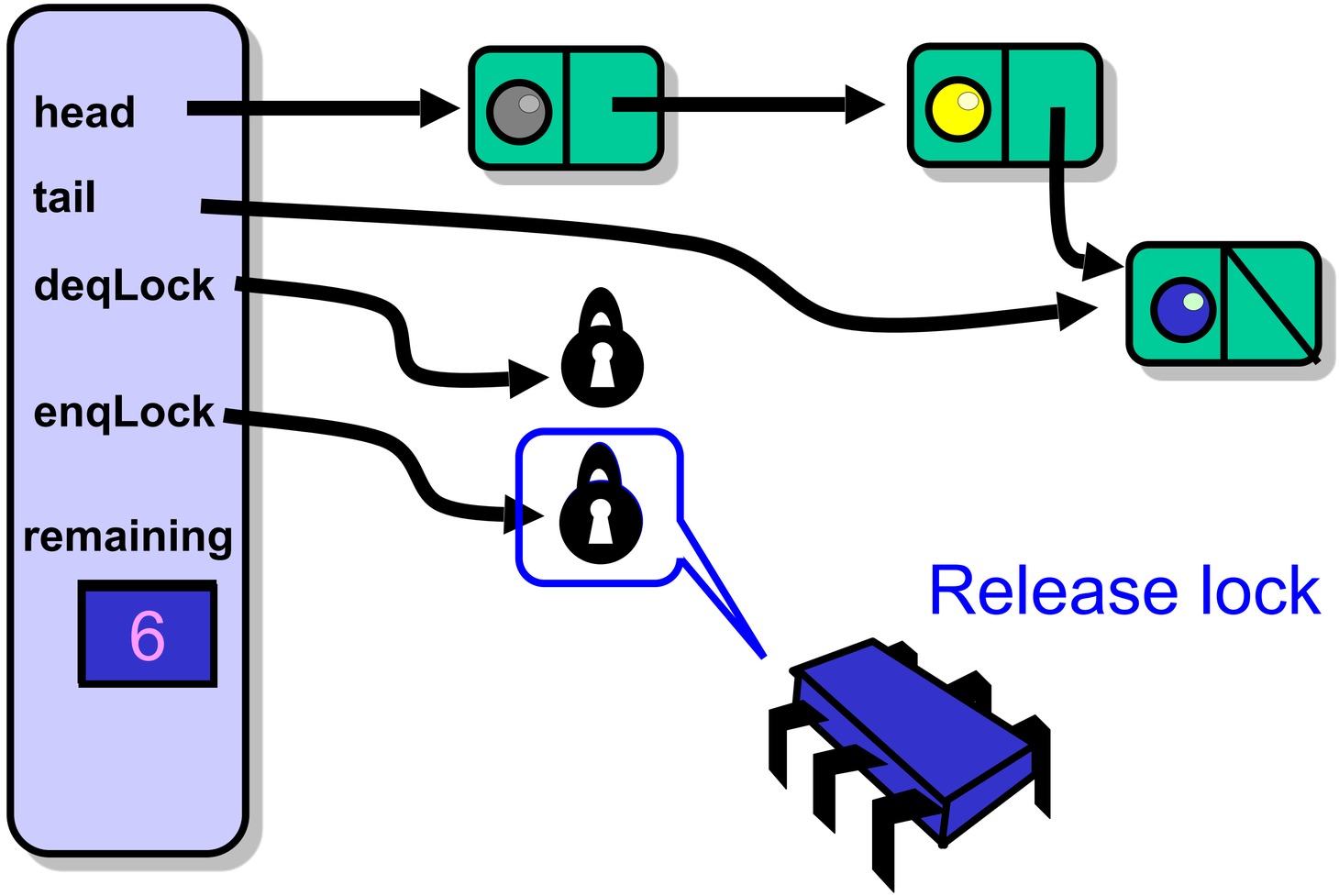
Enqueuer



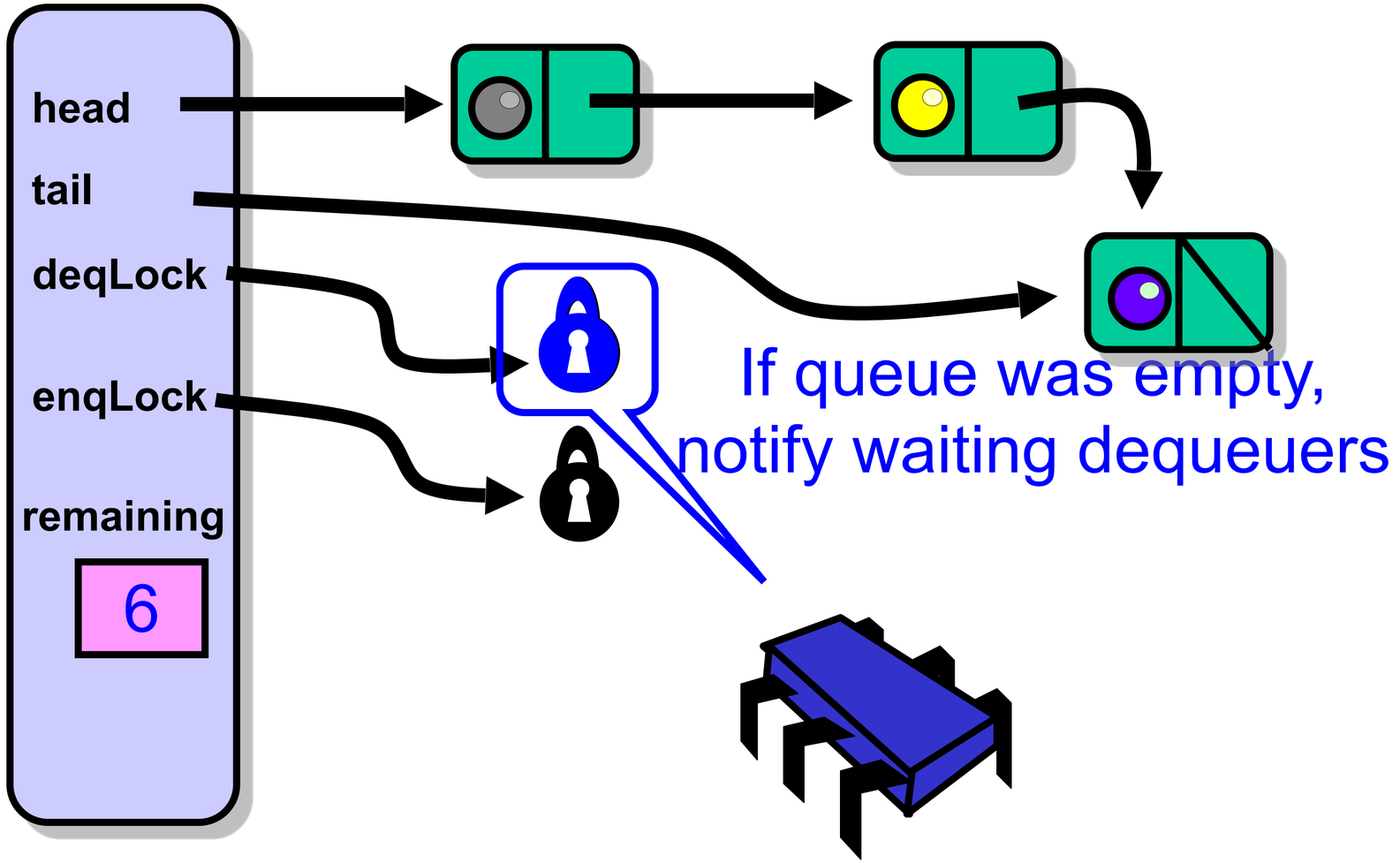
Enqueuer



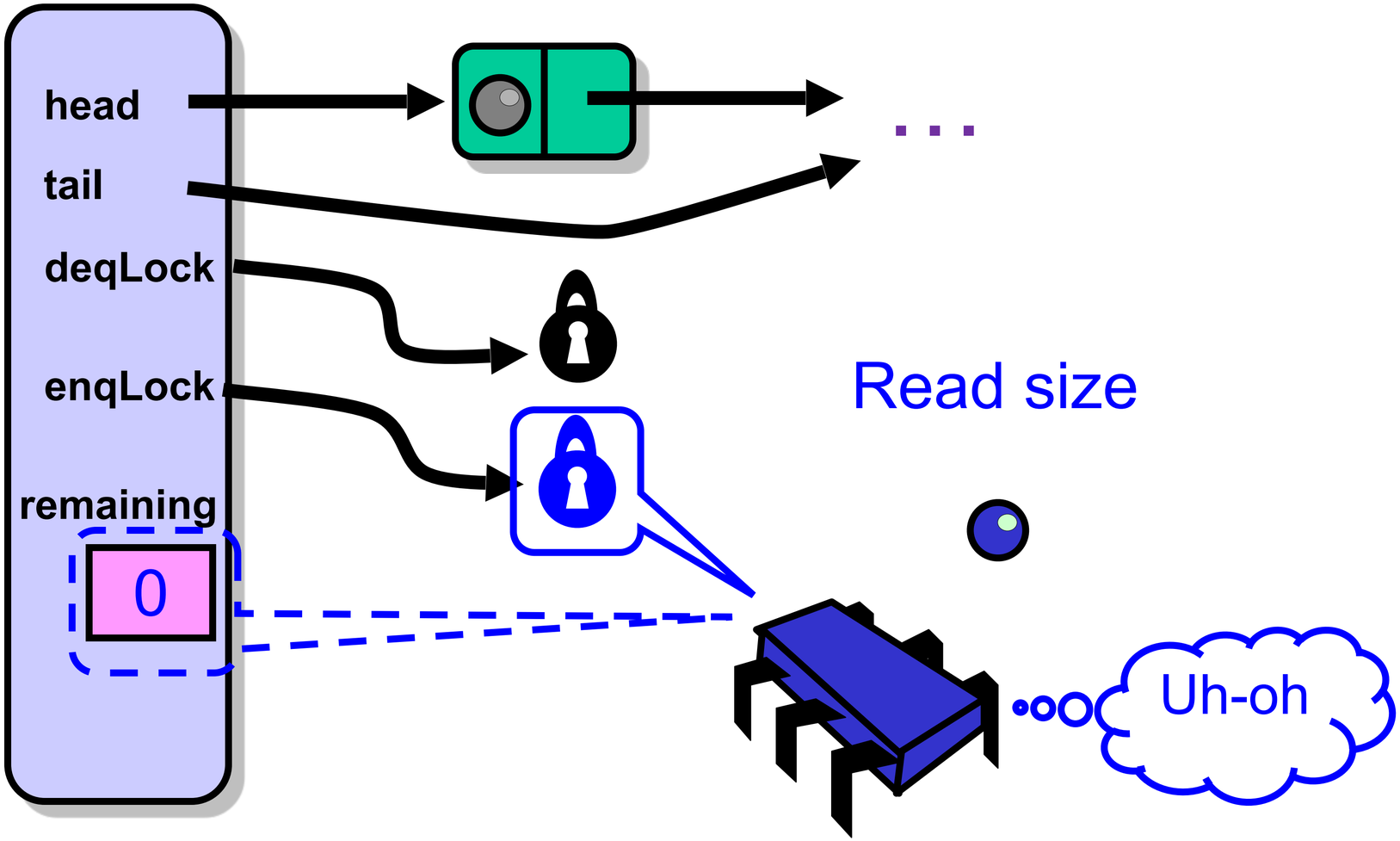
Enqueuer



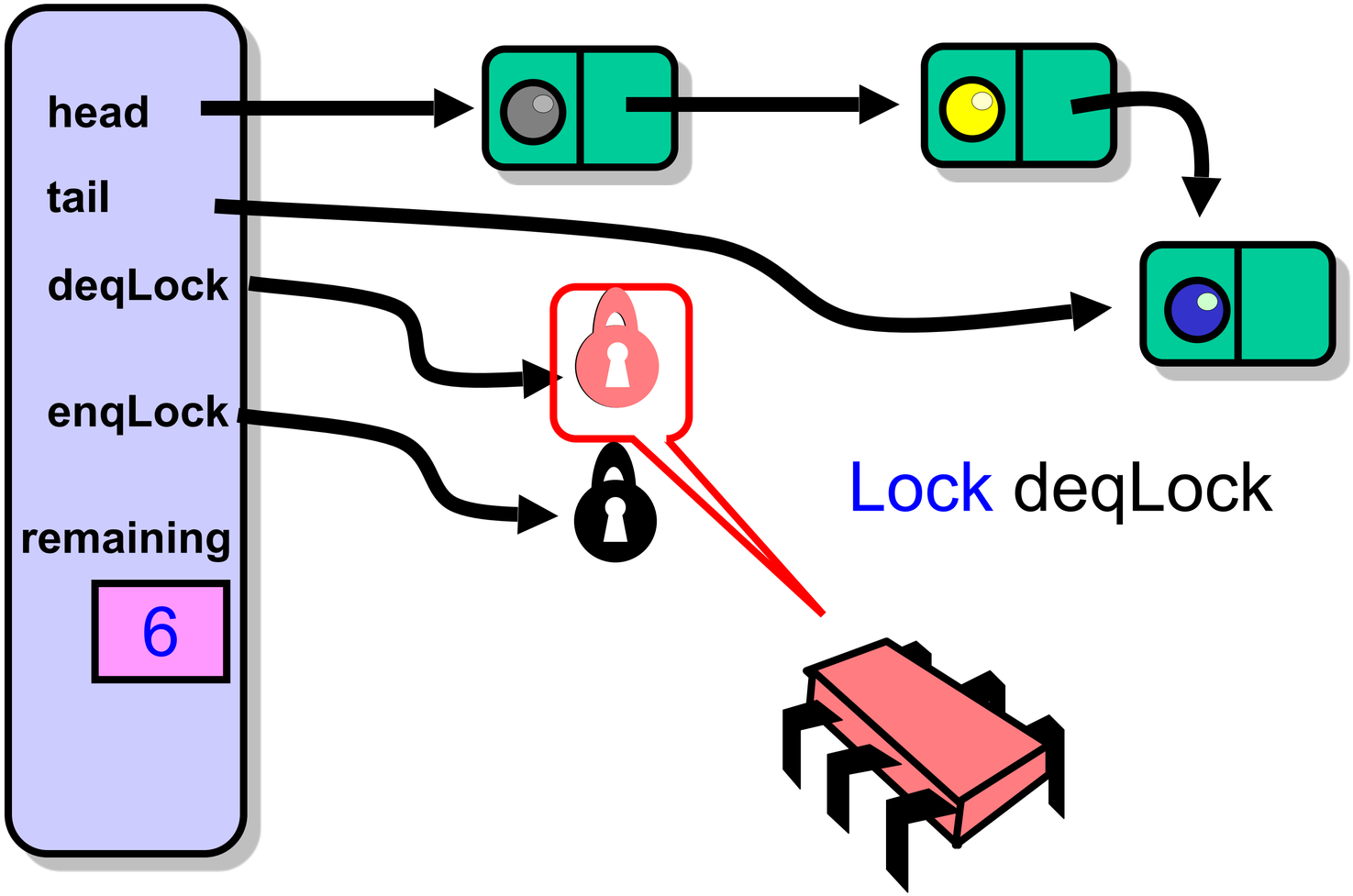
Enqueuer


head
tail
deqLock
enqLock
remaining
6
If queue was empty,
notify waiting dequeuers

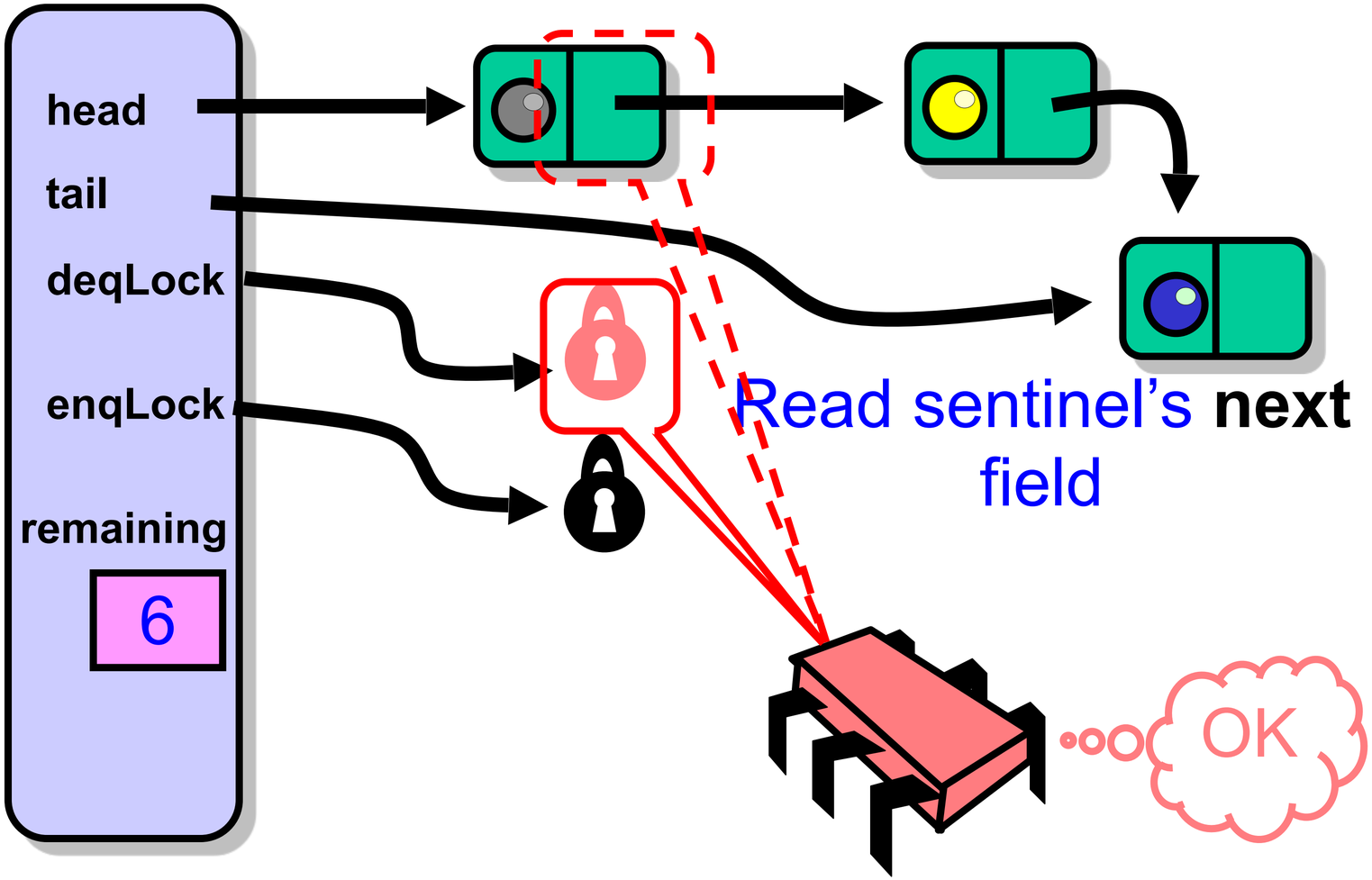
Unsuccesful Enqueuer



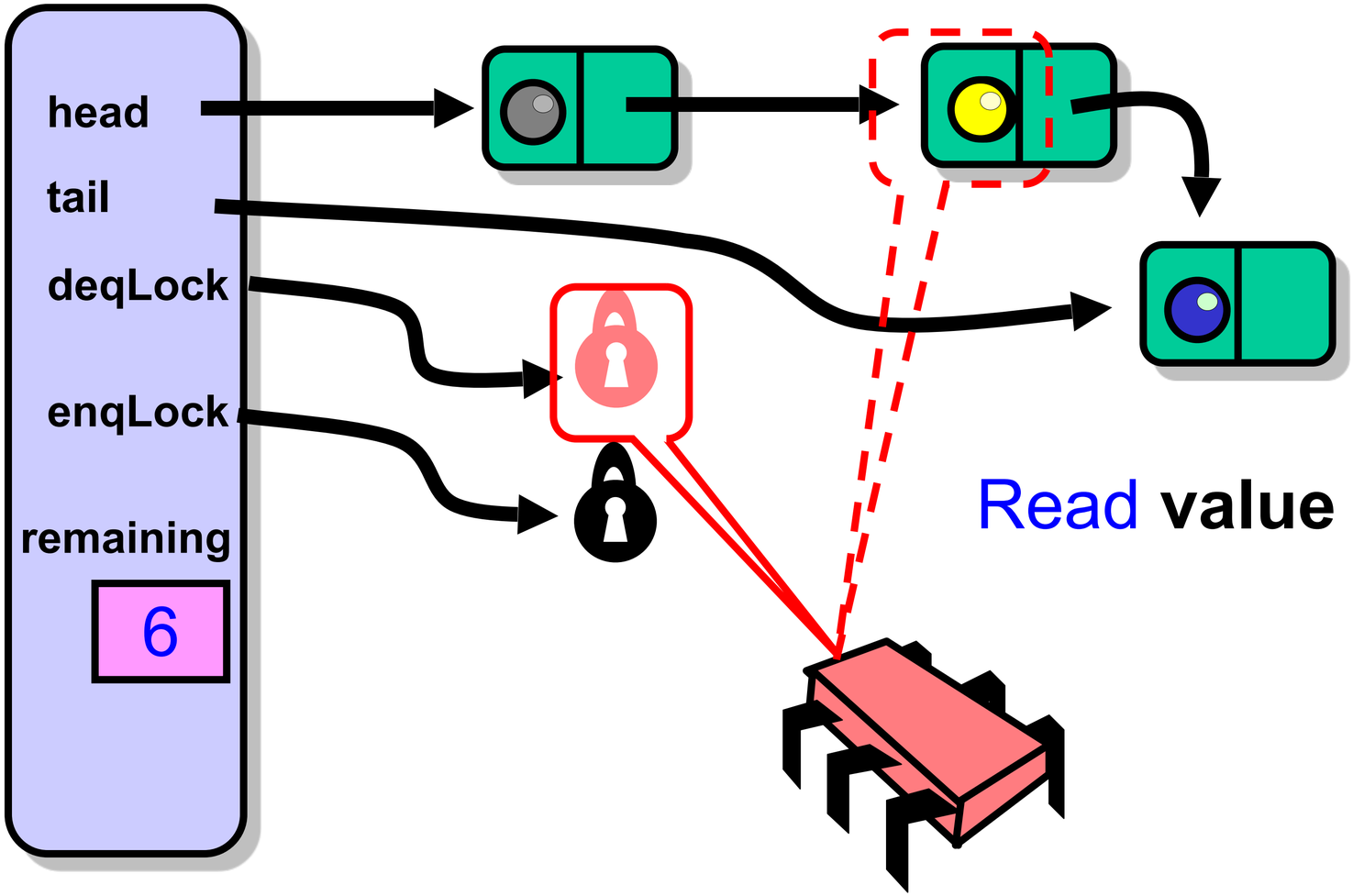
Dequeuer



Dequeuer

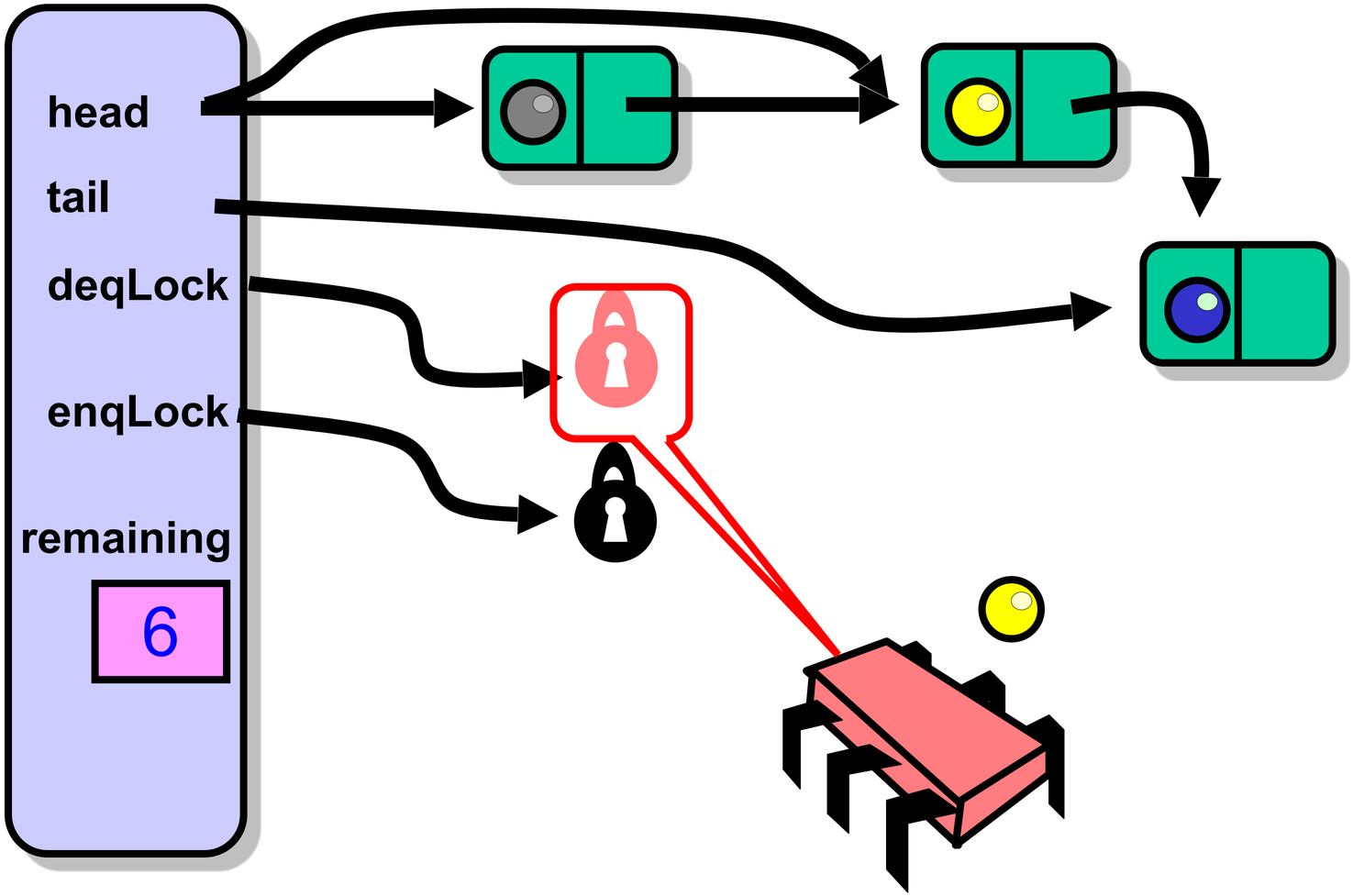

head
tail
deqLock
enqLock
remaining
6
Read sentinel's next field
OK

Dequeuer

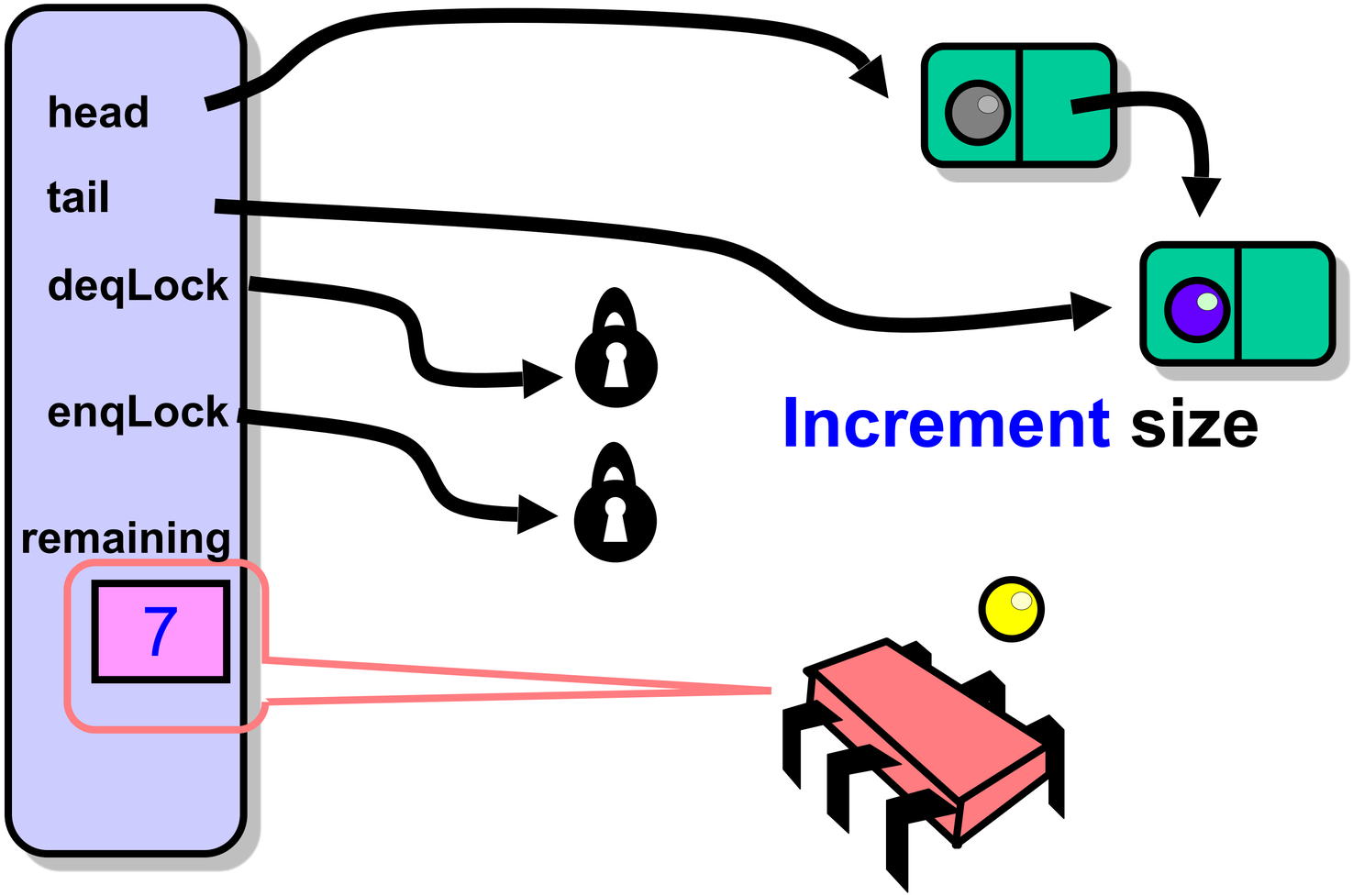


Make first Node new sentinel

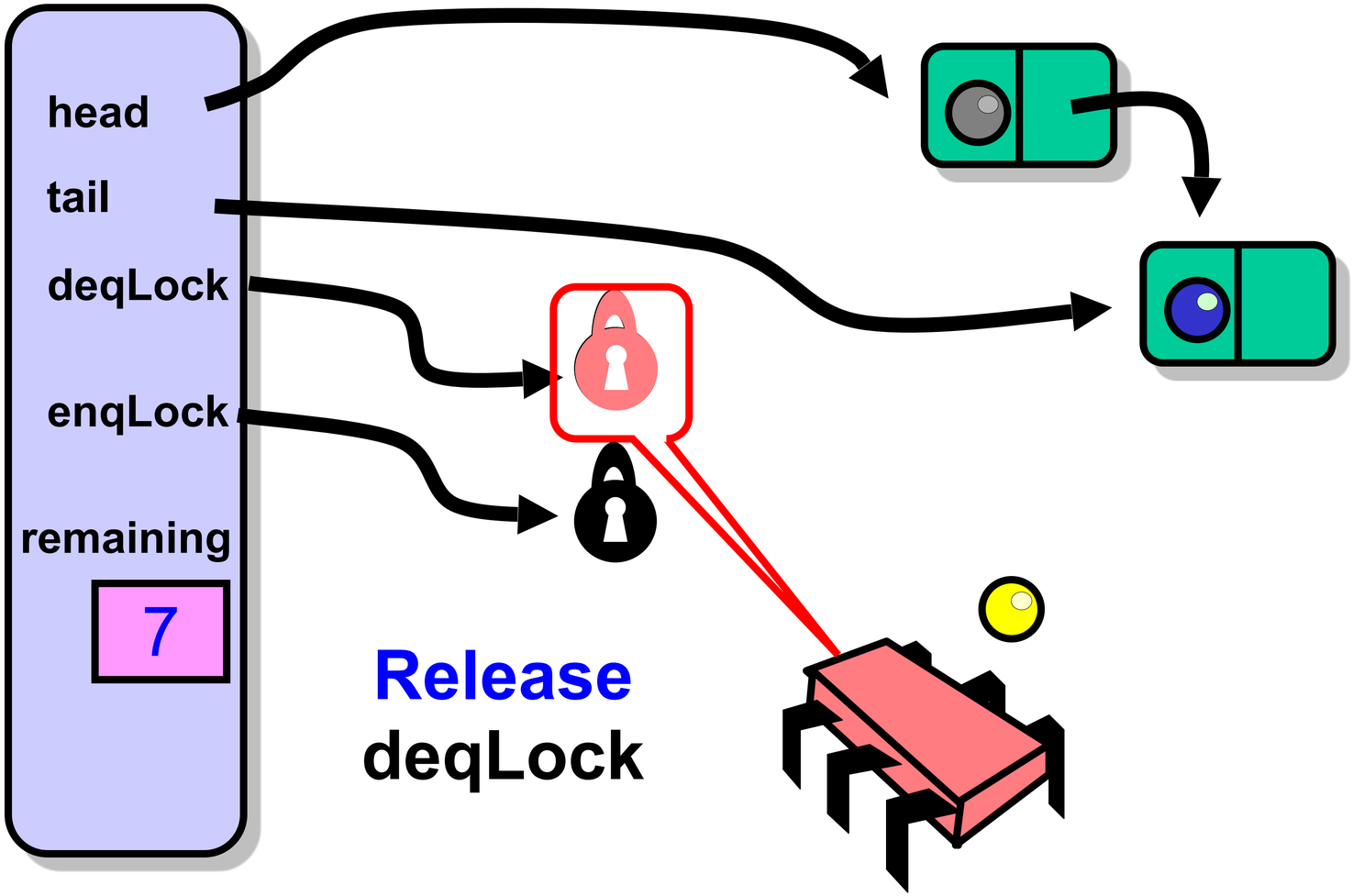
Dequeuer



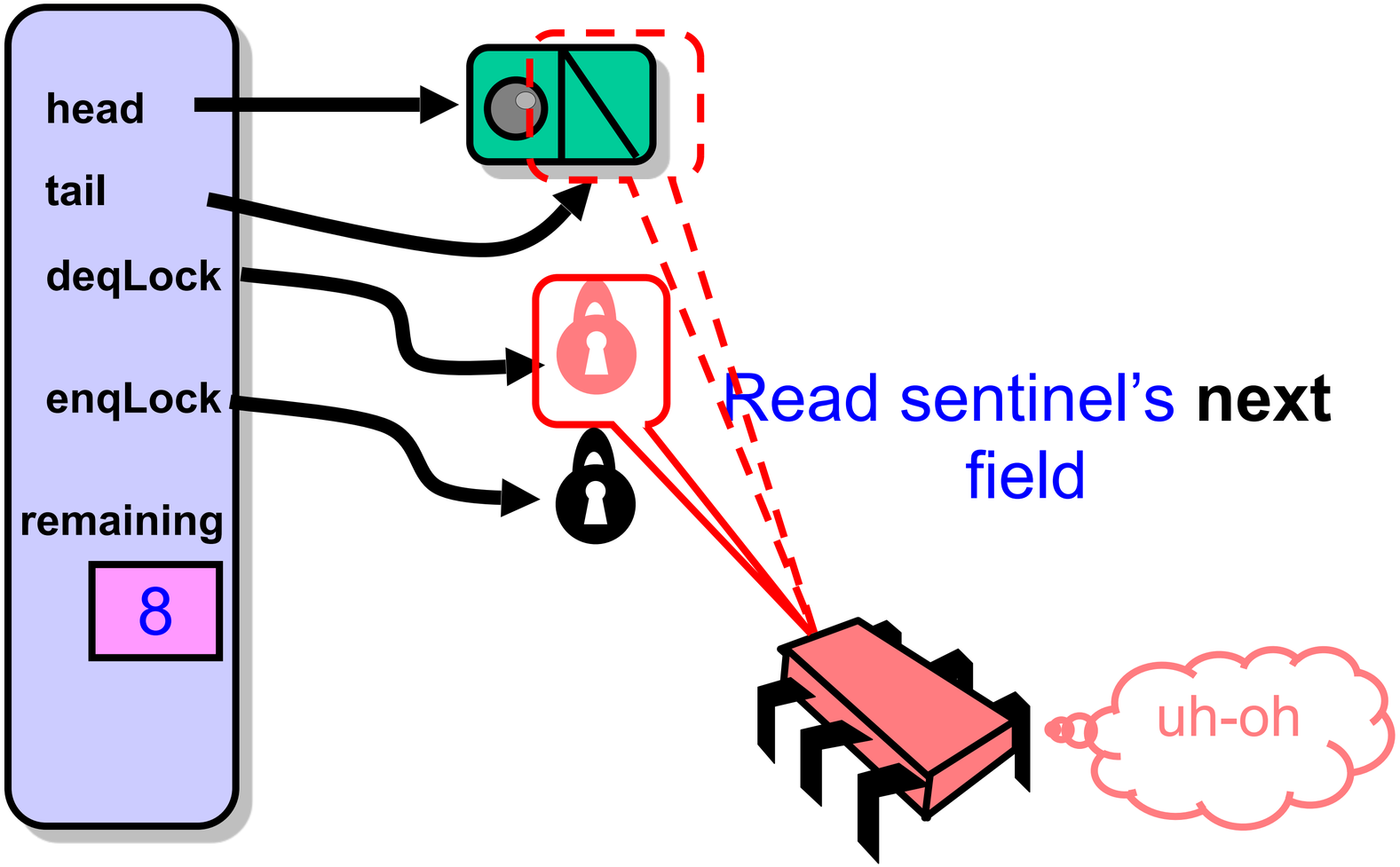
Dequeuer



Dequeuer



Unsuccesful Dequeuer



The Bounded Queue

```
class BoundedQueue[T](private val capacity: Int)
      extends ConcurrentQueue[T]{

  private val enqLock = new ReentrantLock()
  private val deqLock = new ReentrantLock()

  private val notFullCondition = enqLock.newCondition()
  private val notEmptyCondition = deqLock.newCondition()

  private val remaining = new AtomicInteger(capacity)
  private val head: Node = new Node(null)
  private val tail: Node = head
```

Bounded Queue Fields

```
class BoundedQueue[T](private val capacity: Int)
      extends ConcurrentQueue[T]{

  private val enqLock = new ReentrantLock()
  private val deqLock = new ReentrantLock()

  private val notFullCondition = enqLock.newCondition()
  private val notEmptyCondition = deqLock.newCondition()

  private val remaining = new AtomicInteger(capacity)
  private val head: Node = new Node(null)
  private val tail: Node = head
```

Enq & deq locks

Bounded Queue Fields

```
class BoundedQueue[T](private val capacity: Int)
      extends ConcurrentQueue[T]{

  private val enqLock = new ReentrantLock()
  private val deqLock = new ReentrantLock()

  private val notFullCondition = enqLock.newCondition()
  private val notEmptyCondition = deqLock.newCondition()

  private val remaining = new AtomicInteger(capacity)
  private val head: Node = new Node(null)
  private val tail: Node = head
```

Enq lock's associated condition

Bounded Queue Fields

```
class BoundedQueue[T](private val capacity: Int)
      extends ConcurrentQueue[T]{

  private val enqLock = new ReentrantLock()
  private val deqLock = new ReentrantLock()

  private val notFullCondition = enqLock.newCondition()
  private val notEmptyCondition = deqLock.newCondition()

  private val remaining = new AtomicInteger(capacity)
  private val head: Node = new Node(null)
  private val tail: Node = head
```

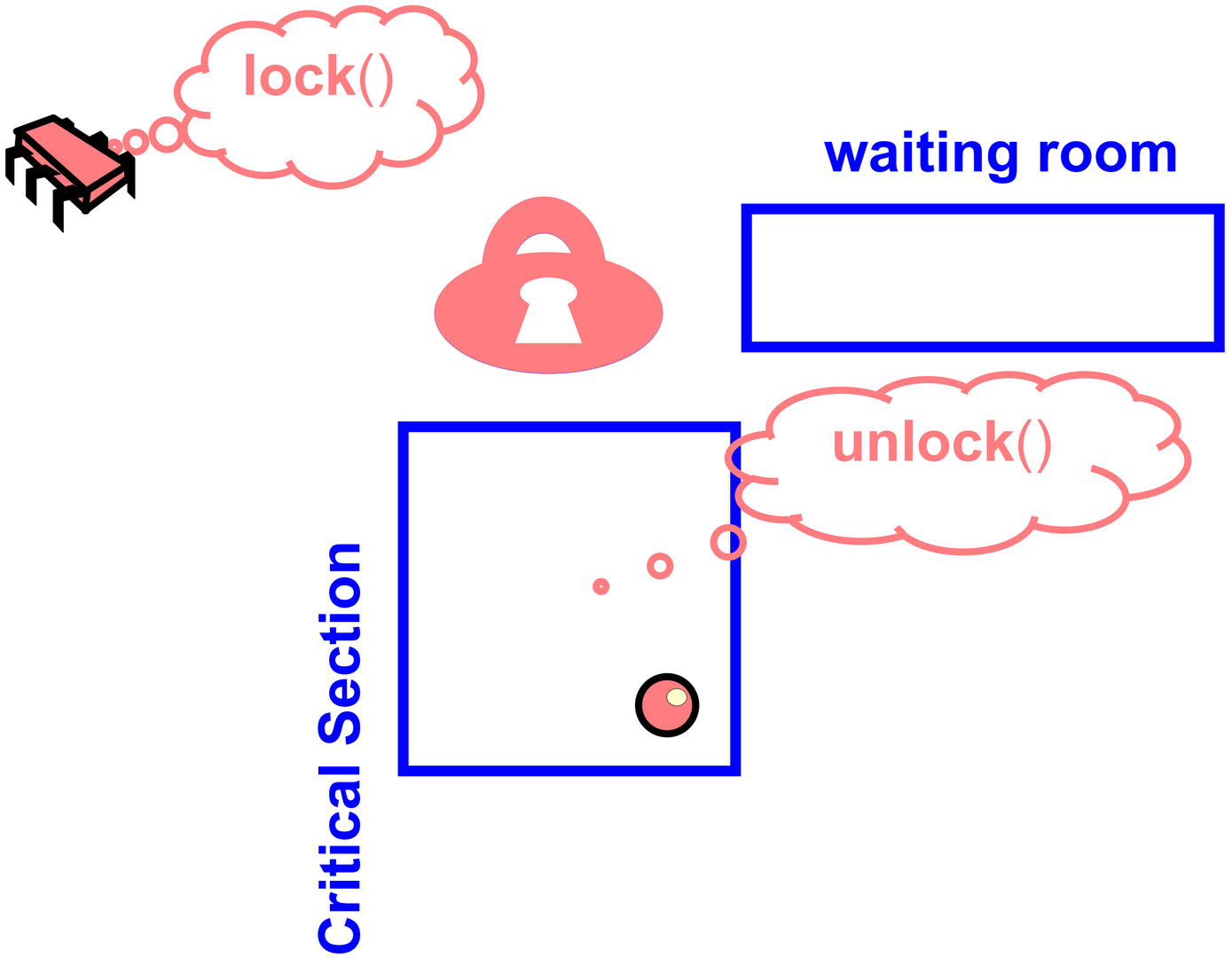
remaining slots: capacity to 0

Bounded Queue Fields

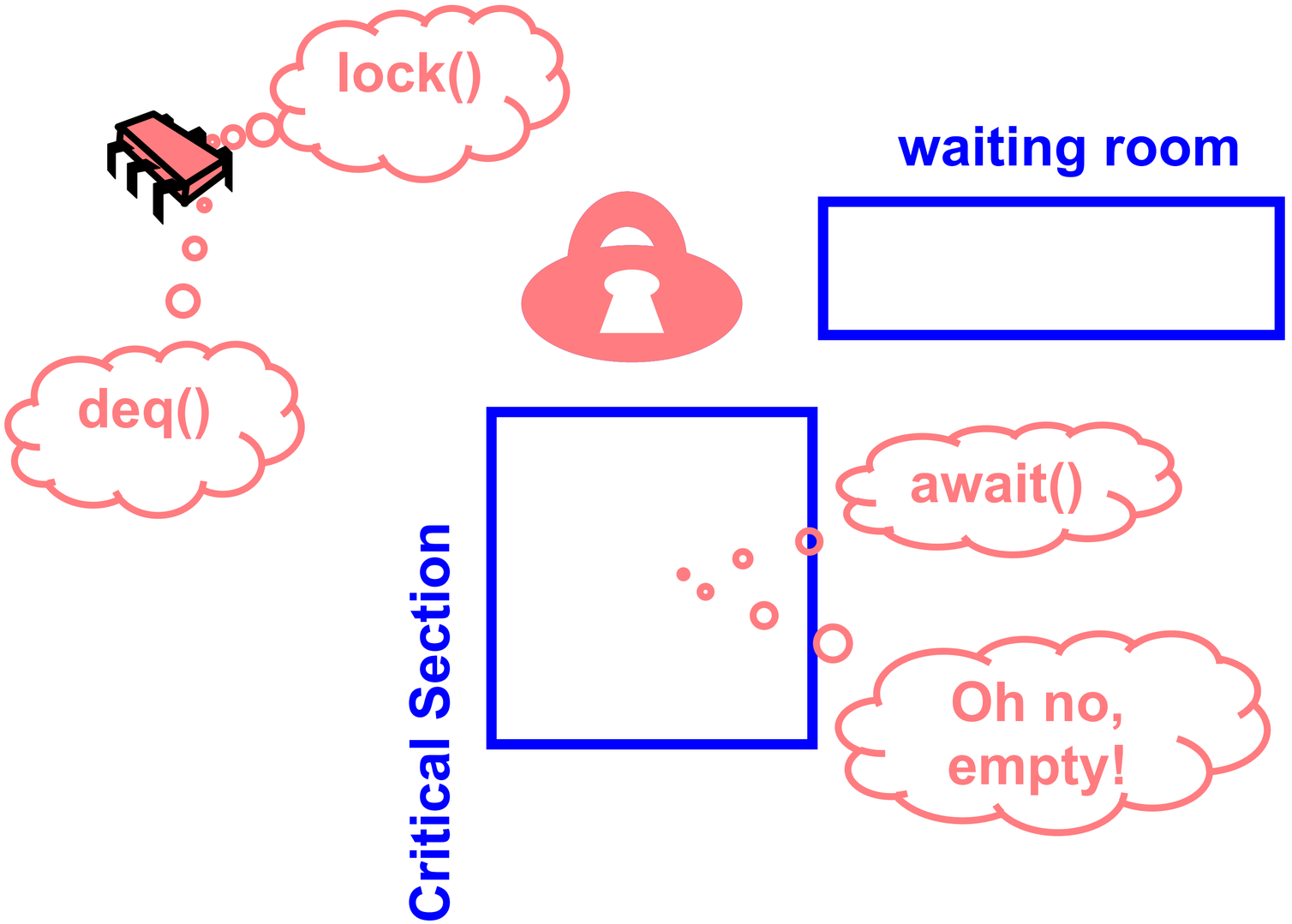
```
class BoundedQueue[T](private val capacity: Int)
      extends ConcurrentQueue[T]{

  private val enqLock = new ReentrantLock()
  private val deqLock = new ReentrantLock()

  private val notFullCondition = enqLock.newCondition()
  private val notEmptyCondition = deqLock.newCondition()

  private val remaining = new AtomicInteger(capacity)
  private val head: Node = new Node(null)
  private val tail: Node = head
```

Head and Tail

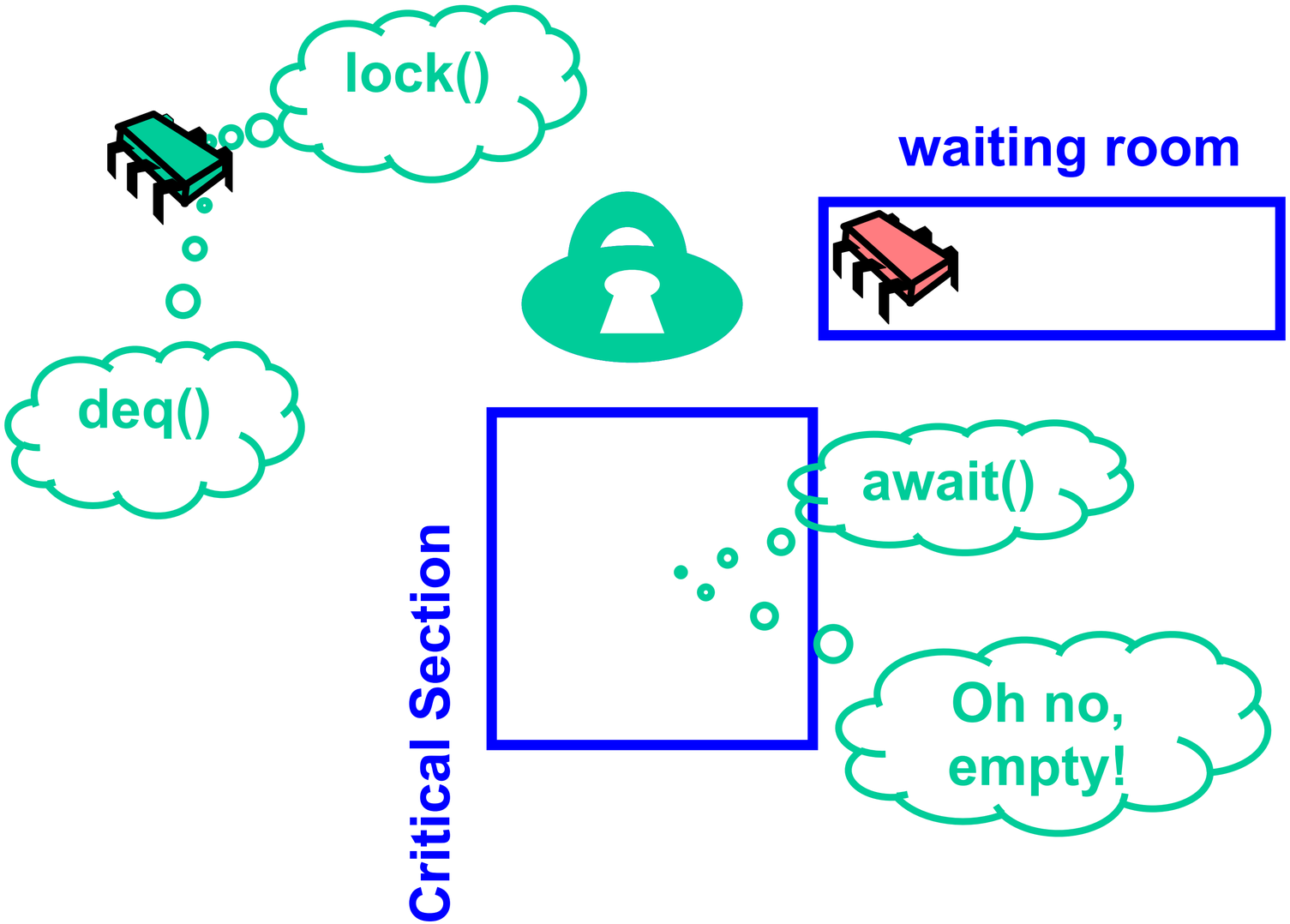
A Monitor Lock



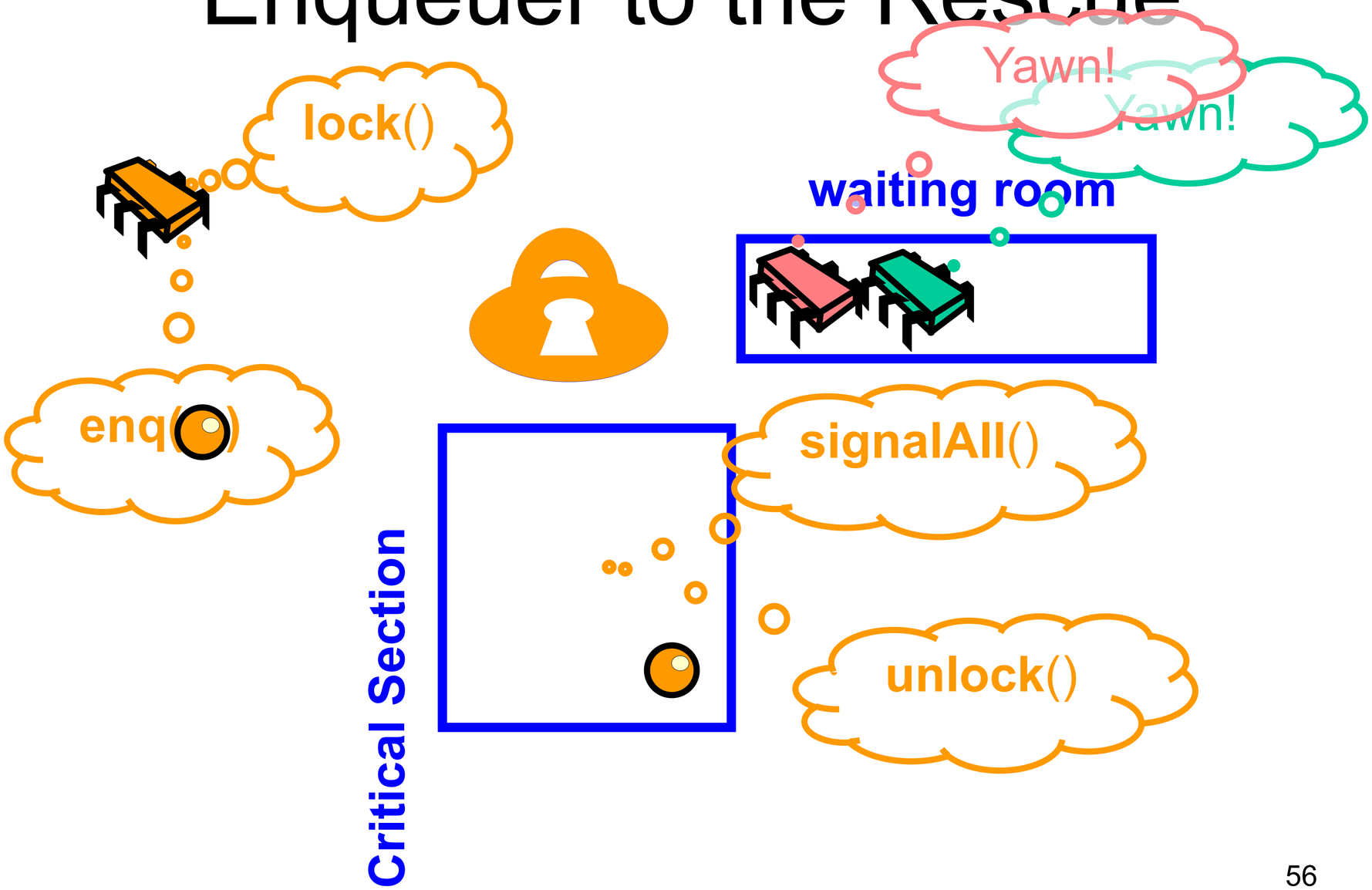
Unsuccessful Deq



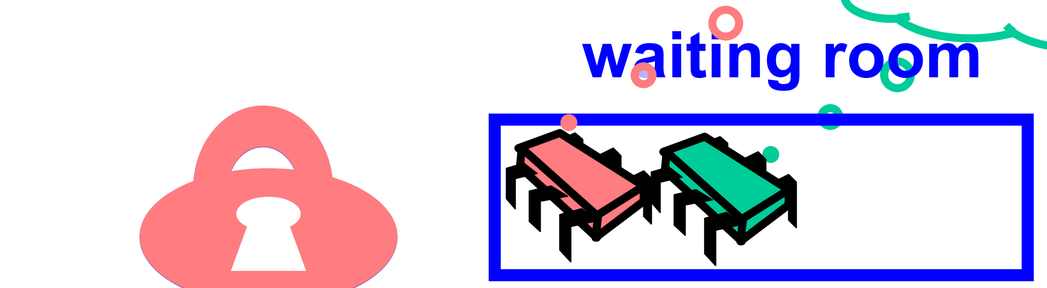
Another One

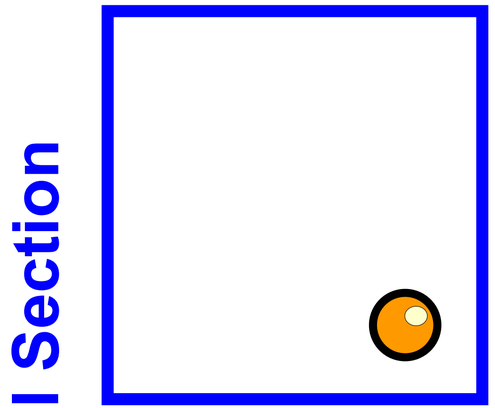


Enqueuer to the Rescue



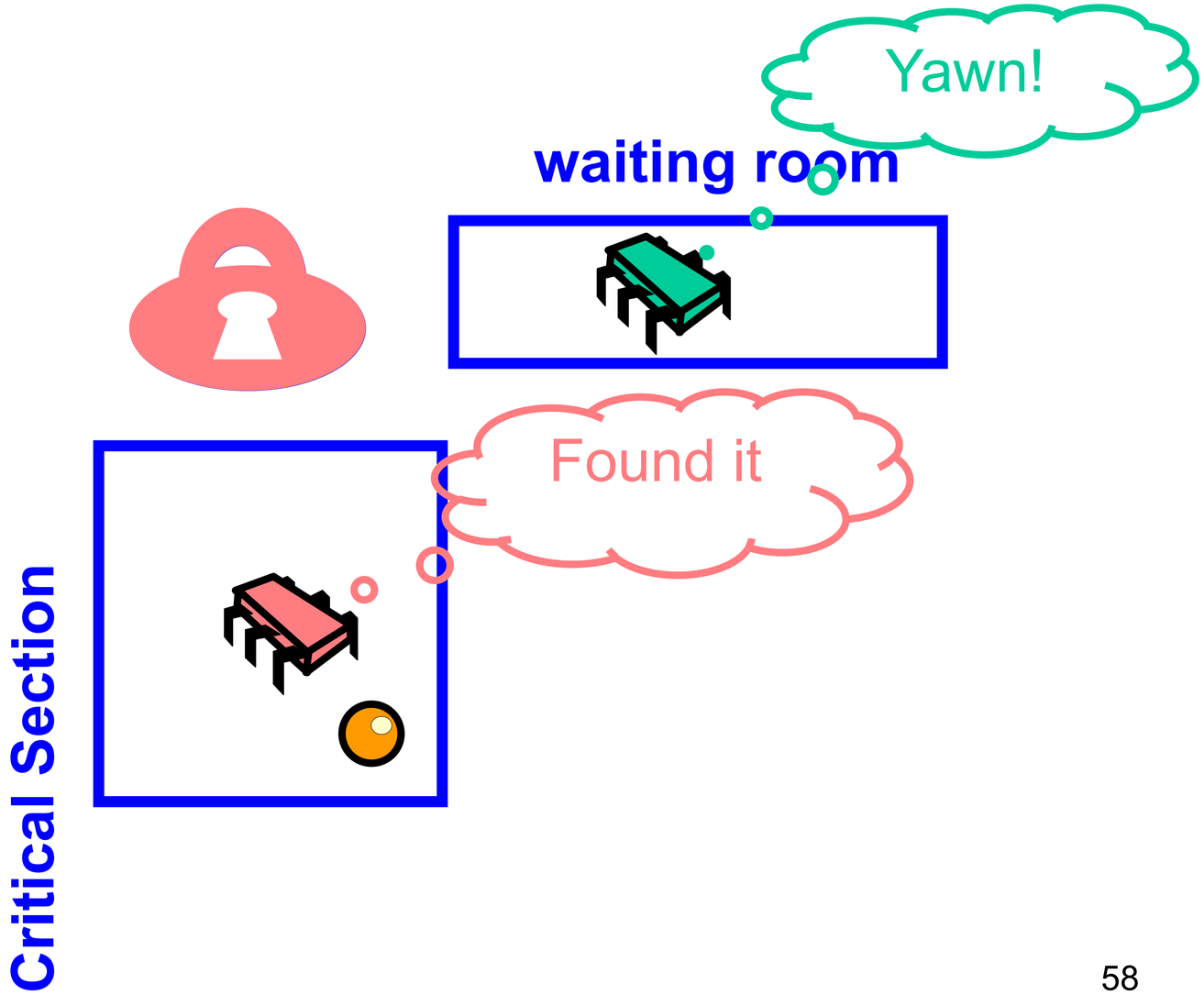
Monitor Signalling



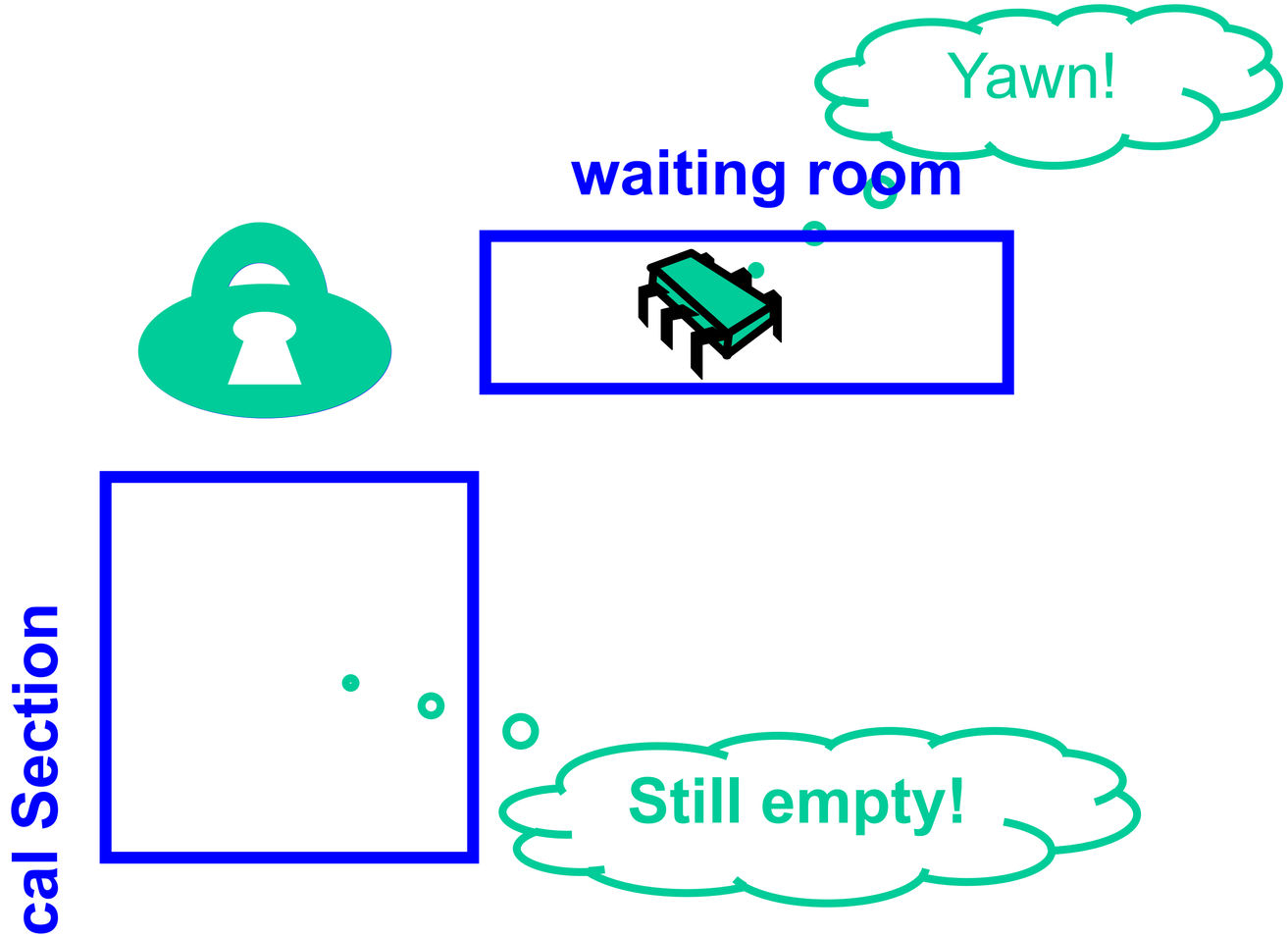


Awakened thread might still lose lock to outside contender…

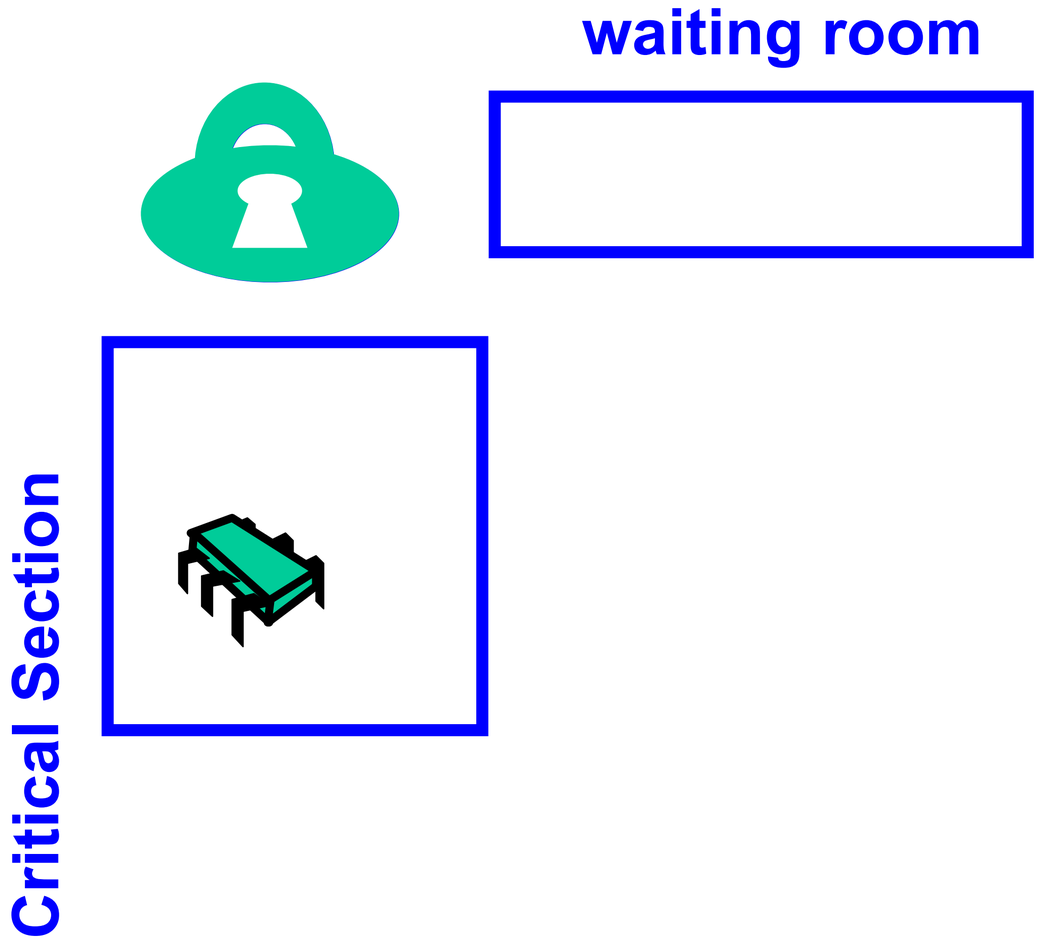
Dequeuers Signalled



Dequeuers Signaled



Dollar Short + Day Late



Enq Method Part One

```
def enq(x: T): Unit = {
  var mustWakeDequeuers = false
  enqLock.lock()
  try {
    while (remaining.get == 0) {
      notFullCondition.await()
    }
    val e = new Node(x)
    tail.next = e
    tail = e
    if (remaining.getAndDecrement == capacity)
      mustWakeDequeuers = true
  } finally {
    enqLock.unlock()
  }
  // ...
}
```

Enq Method Part One

```
def enq(x: T): Unit = {
  var mustWakeDequeuers = false
  enqLock.lock()
  try {
    while (remaining.get == 0)
      notFullCondition.await()
    }
    val e = new Node(x)
    tail.next = e
    tail = e
    if (remaining.getAndDecrement == capacity)
      mustWakeDequeuers = true
  } finally {
    enqLock.unlock()
  }
  // ...
}
```

Lock and unlock enq lock

Enq Method Part One

```
def enq(x: T): Unit = {
  var mustWakeDequeuers = false
  enqLock.lock()
  try {
    while (remaining.get == 0) {
      notFullCondition.await()
    }
    val e = new Node(x)
    tail.next = e
    tail = e
    if (size.getAndDecrement == capacity)
      mustWakeDequeuers = true
  } finally {
    enqLock.unlock()
  }
  // ..
}
```

Wait while queue is full …

Enq Method Part One

```
def enq(x: T): Unit = {
  var mustWakeDequeuers = false
  enqLock.lock()
  try {
    while (remaining.get == 0) {
      notFullCondition.await()
    }
    val e = new Node(x)
    tail.next = e
    tail = e
    if (size.getAndDecrement == capacity)
      mustWakeDequeuers = true
  } finally {
    enqLock.unlock()
  }
  // ...
}
```

when `await()` returns, you might still fail the test !

Be Afraid

```
def enq(x: T): Unit = {
  var mustWakeDequeuers = false
  enqLock.lock()
  try {
    while (remaining.get == 0) {
      notFullCondition.await()
    }
    val e = new Node(x)
    tail.next = e
    tail = e
    if (size.getAndDecrement == capacity)
      mustWakeDequeuers = true
  } finally {
    enqLock.unlock()
  }
  // ...
}
```

After the loop: how do we know the queue won't become full again?

Enq Method Part One

```
def enq(x: T): Unit = {
  var mustWakeDequeuers = false
  enqLock.lock()
  try {
    while (remaining.get == 0) {
      notFullCondition.await()
    }
    val e = new Node(x)
    tail.next = e
    tail = e
    if (remaining.getAndDecrement == capacity)
      mustWakeDequeuers = true
  } finally {
    enqLock.unlock()
  }
  // ...
}
```

Add new node

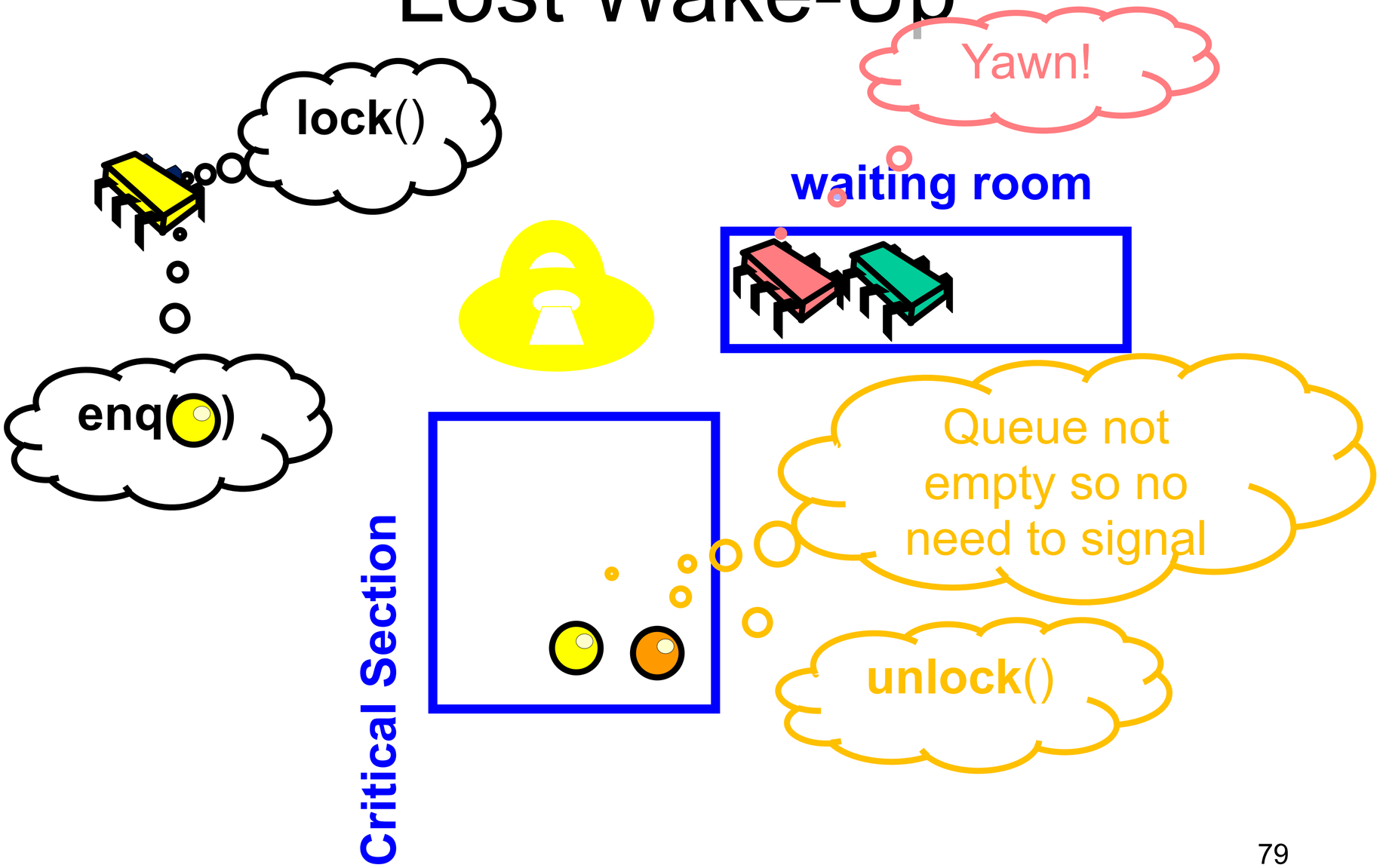
Enq Method Part One

```
def enq(x: T): Unit = {
  var mustWakeDequeuers = false
  enqLock.lock()
  try {
    while (remaining.get == 0) {
      notFullCondition.await()
    }
    val e = new Node(x)
    tail.next = e
    tail = e
    if (remaining.getAndDecrement == capacity)
      mustWakeDequeuers = true
  } finally {
    enqLock.unlock()
  }
  // ...
}
```

If queue was empty, wake frustrated dequeuers

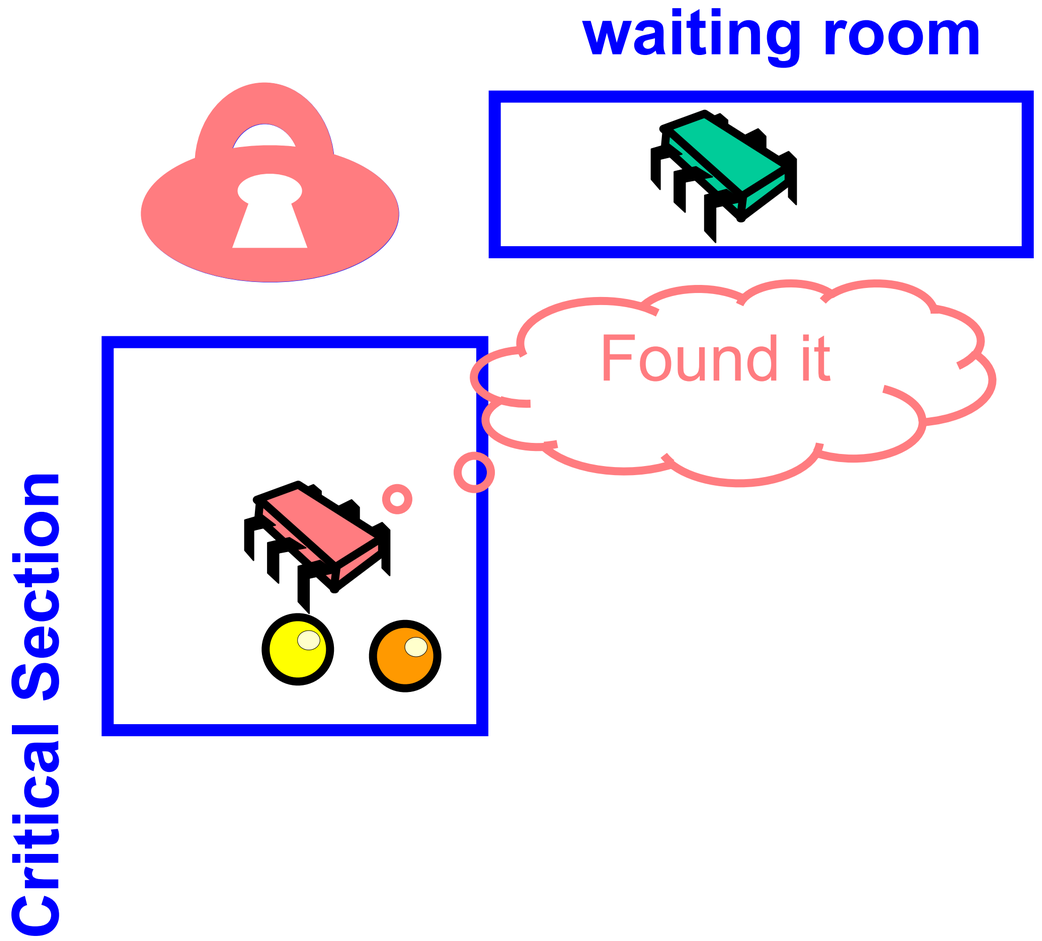
Beware Lost Wake-Ups

Lost Wake-Up

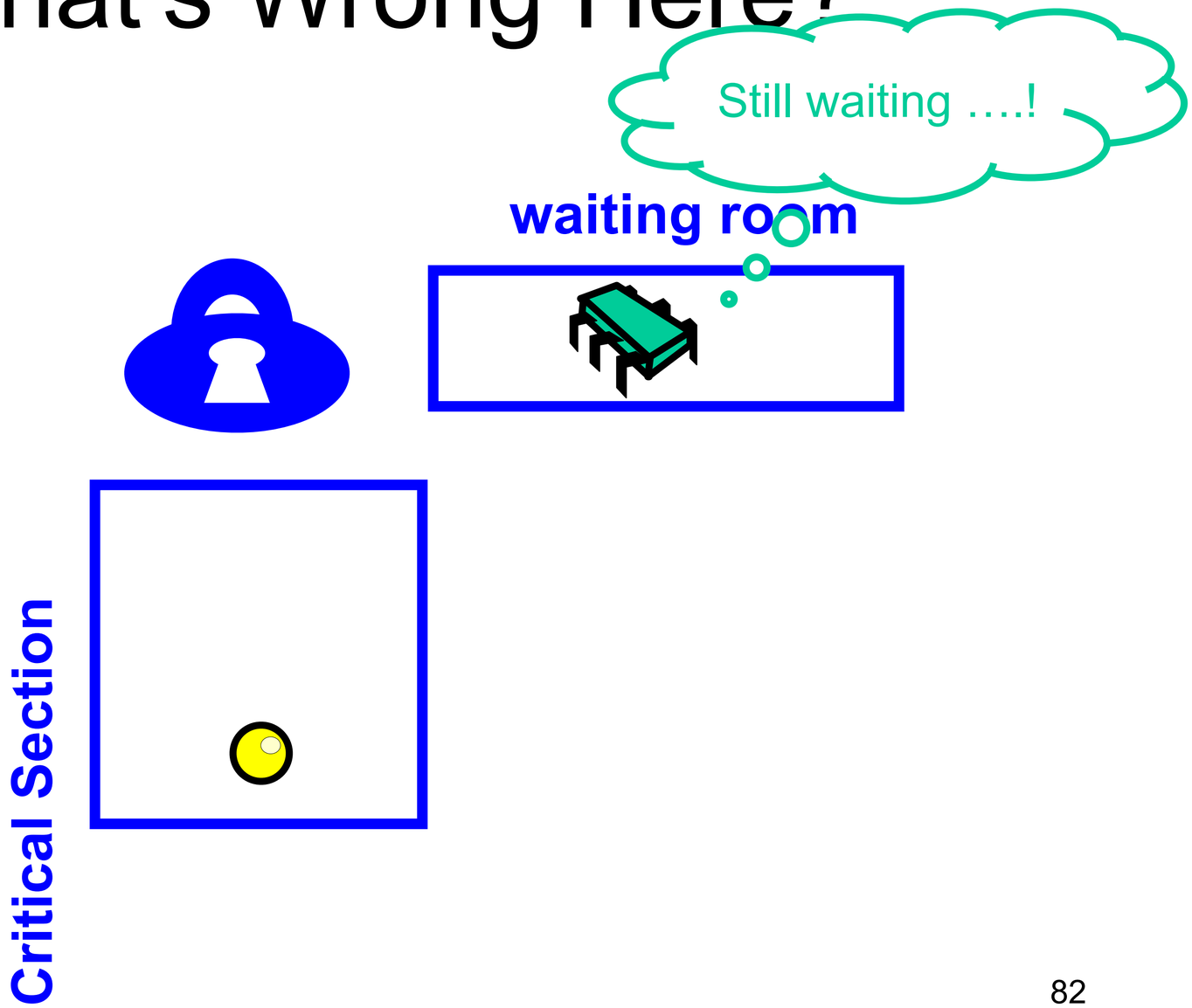

lock()
Yawn!
waiting room
enq()
Queue not empty so no need to signal
Critical Section
unlock()

Lost Wake-Up

Lost Wake-Up



What's Wrong Here?



Solution to Lost Wakeup

- Always use
  - `signalAll()` and `notifyAll()`
- Not
  - `signal()` and `notify()`

Enq Method Part Deux

```
def enq(x: T): Unit = {
  // ...
  if (mustWakeDequeuers) {
    deqLock.lock()
    try {
      notEmptyCondition.signalAll()
    } finally {
      deqLock.unlock()
    }
  }
}
```

Enq Method Part Deux

```
def enq(x: T): Unit = {
  // ...
  if (mustWakeDequeuers) {
    deqLock.lock()
    try {
      notEmptyCondition.signalAll()
    } finally {
      deqLock.unlock()
    }
  }
}
```

Are there dequeuers to be signaled?

Enq Method Part Deux

```
def enq(x: T): Unit = {
  // ...
  if (mustWakeDequeuers) {
    deqLock.lock()
    try {
      notEmptyCondition.signalAll()
    } finally {
      deqLock.unlock()
    }
  }
}
```

Lock and unlock deq lock

Enq Method Part Deux

Signal dequeuers that queue is no longer empty

```
    if (mustWakeDequeuers) {
      deqLock.lock()
      try {
        notEmptyCondition.signalAll()
      } finally {
        deqLock.unlock()
      }
    }
  }
```

‹Good place for a break›

The `enq()` & `deq()` Methods

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

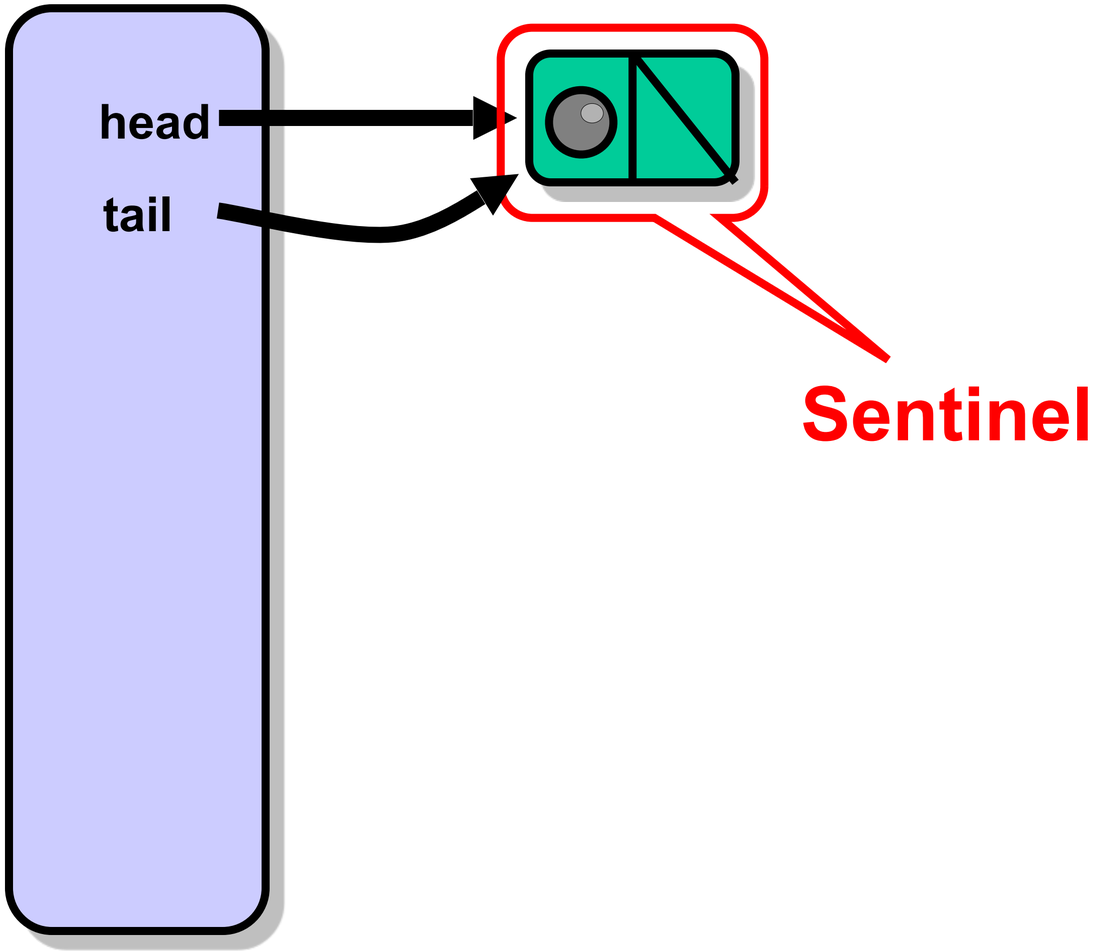
Split the Counter

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

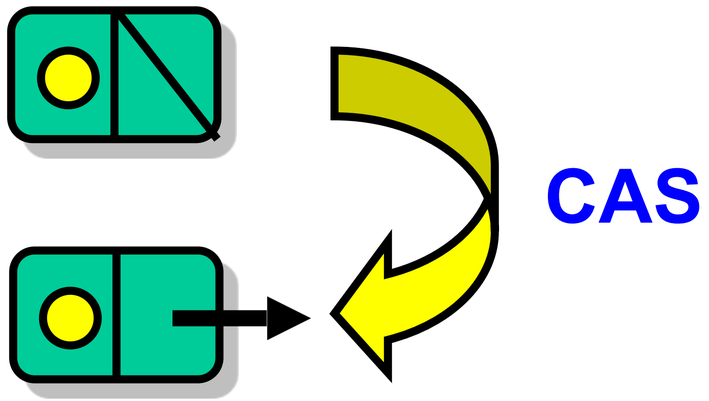
Split Counter

- Enqueuer increments **`enqSize`**
- Dequeuer increments **`deqSize`**
- When enqueuer hits capacity
  - Locks **`deqLock`**
  - Sets **`size = enqSize - deqSize`**
- Intermittent synchronization
  - Not with each method call
  - Need both locks! (careful …)

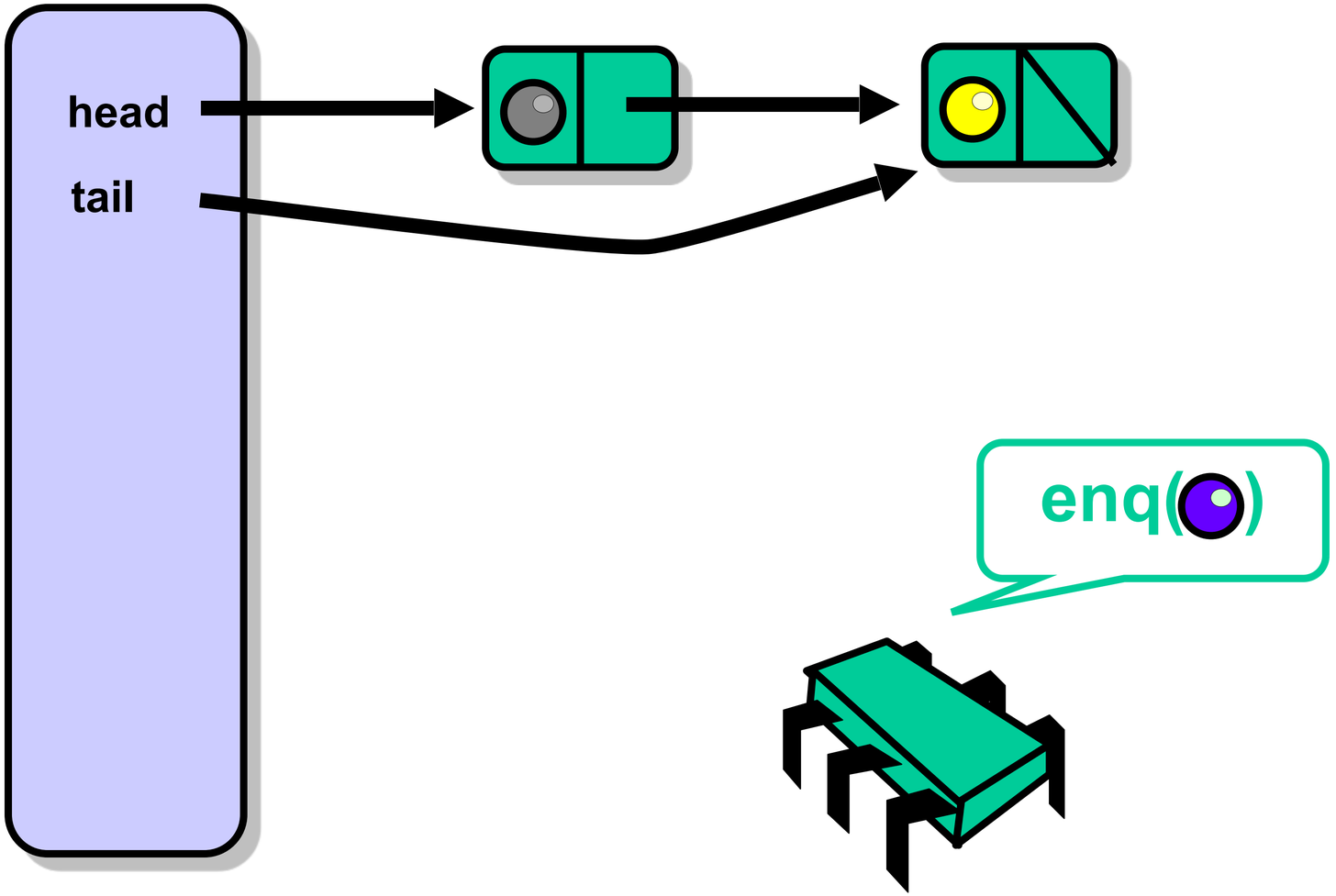
A Lock-Free Queue



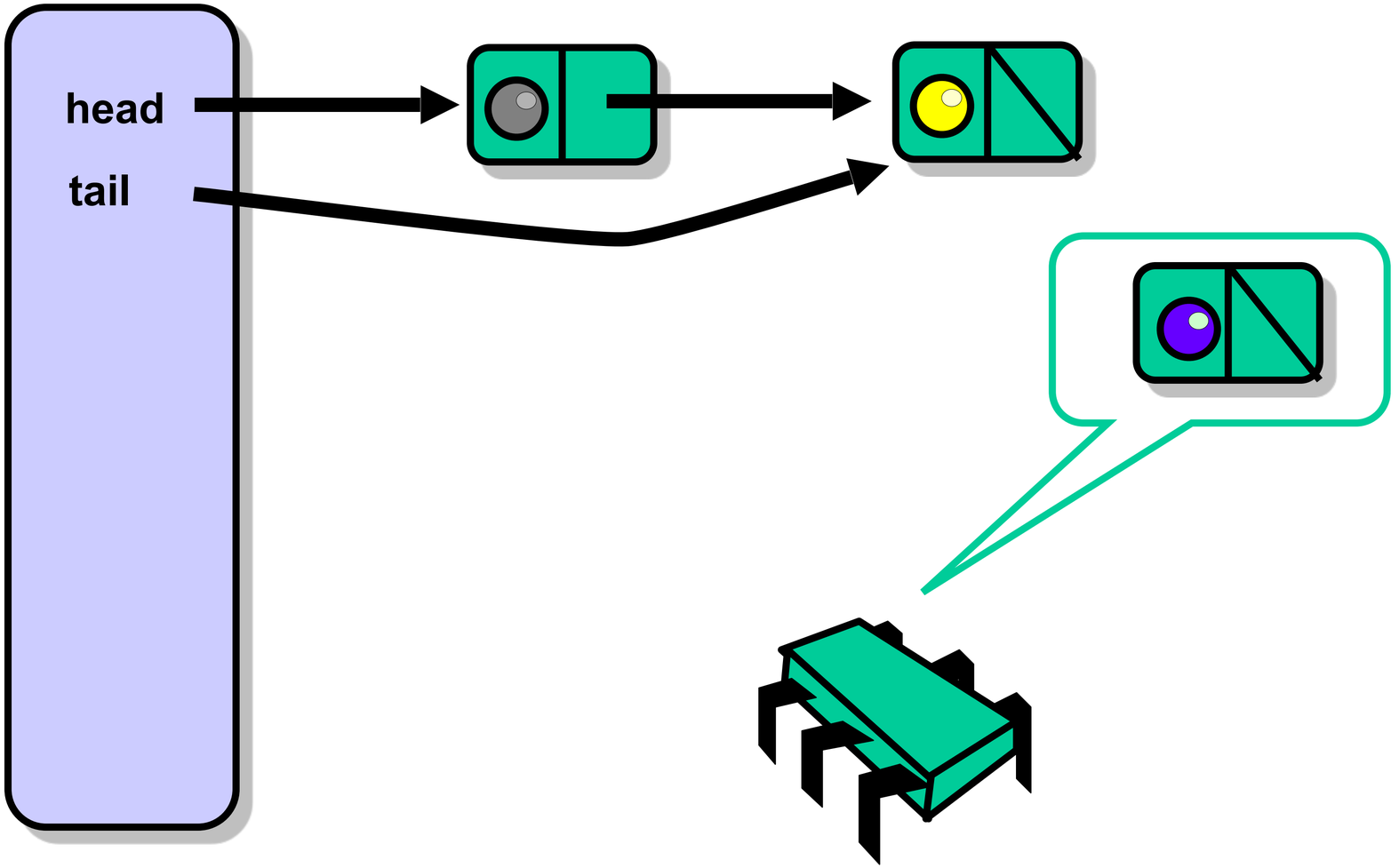
Compare and Set



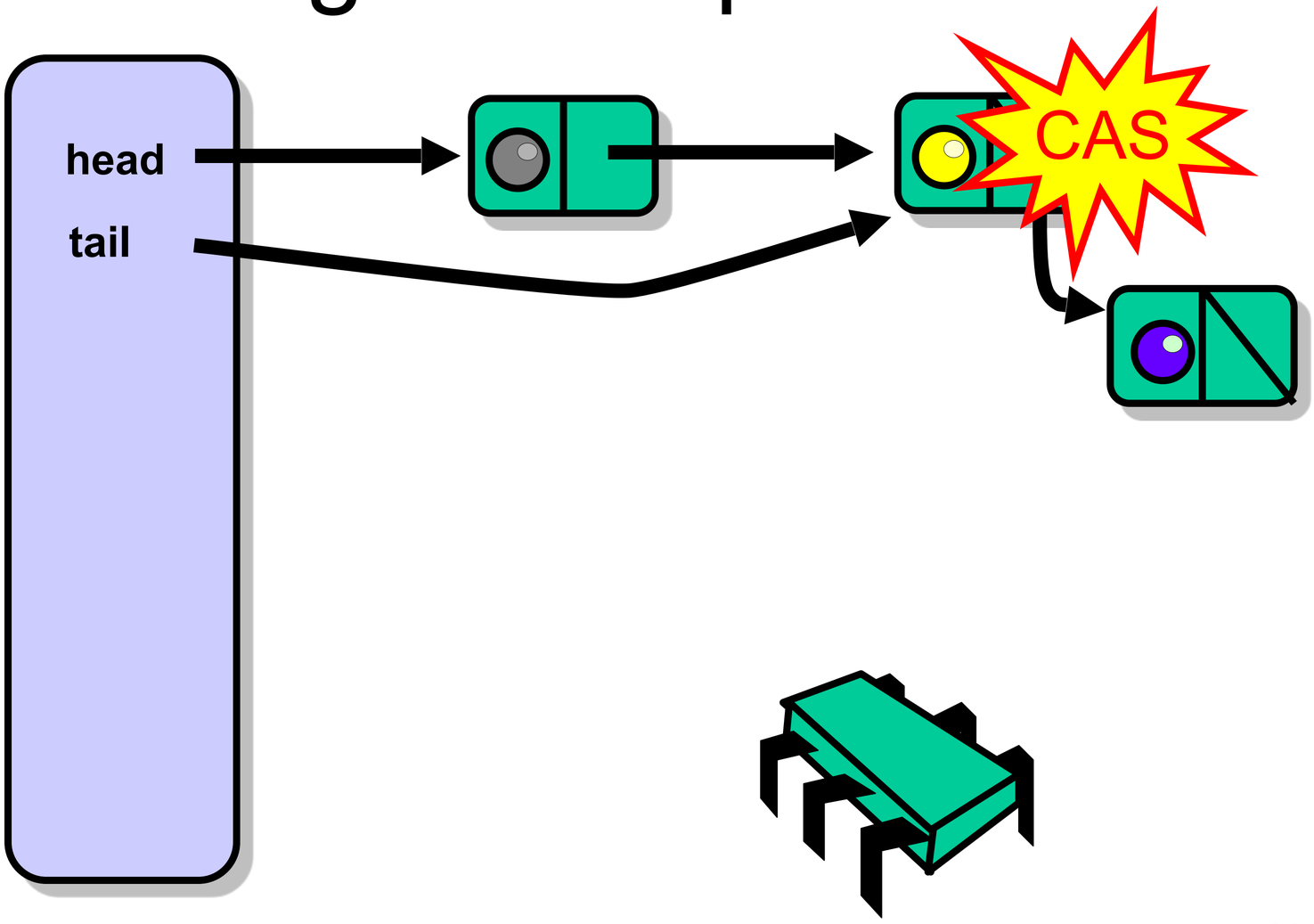
Enqueue



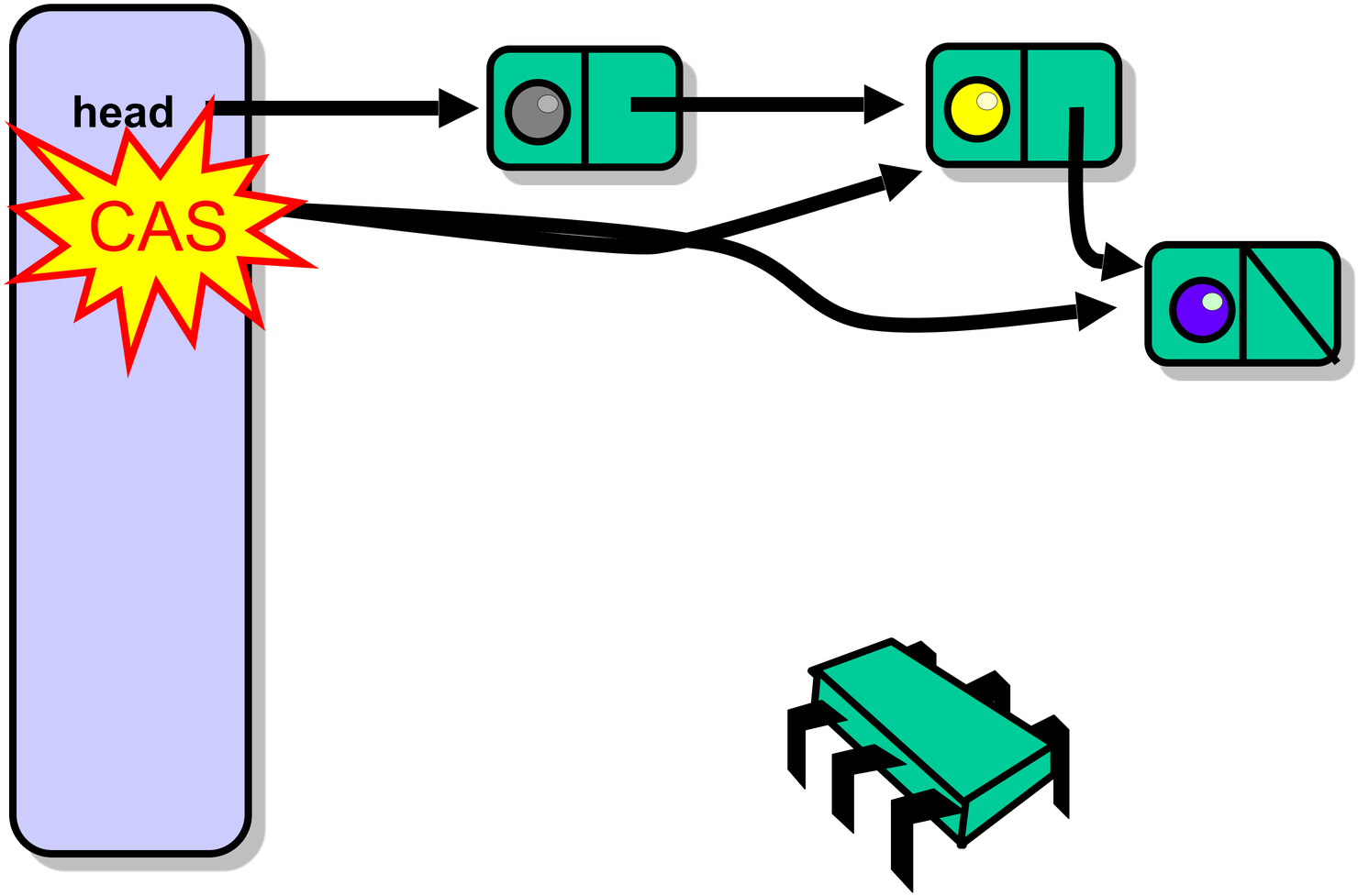
Enqueue



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!

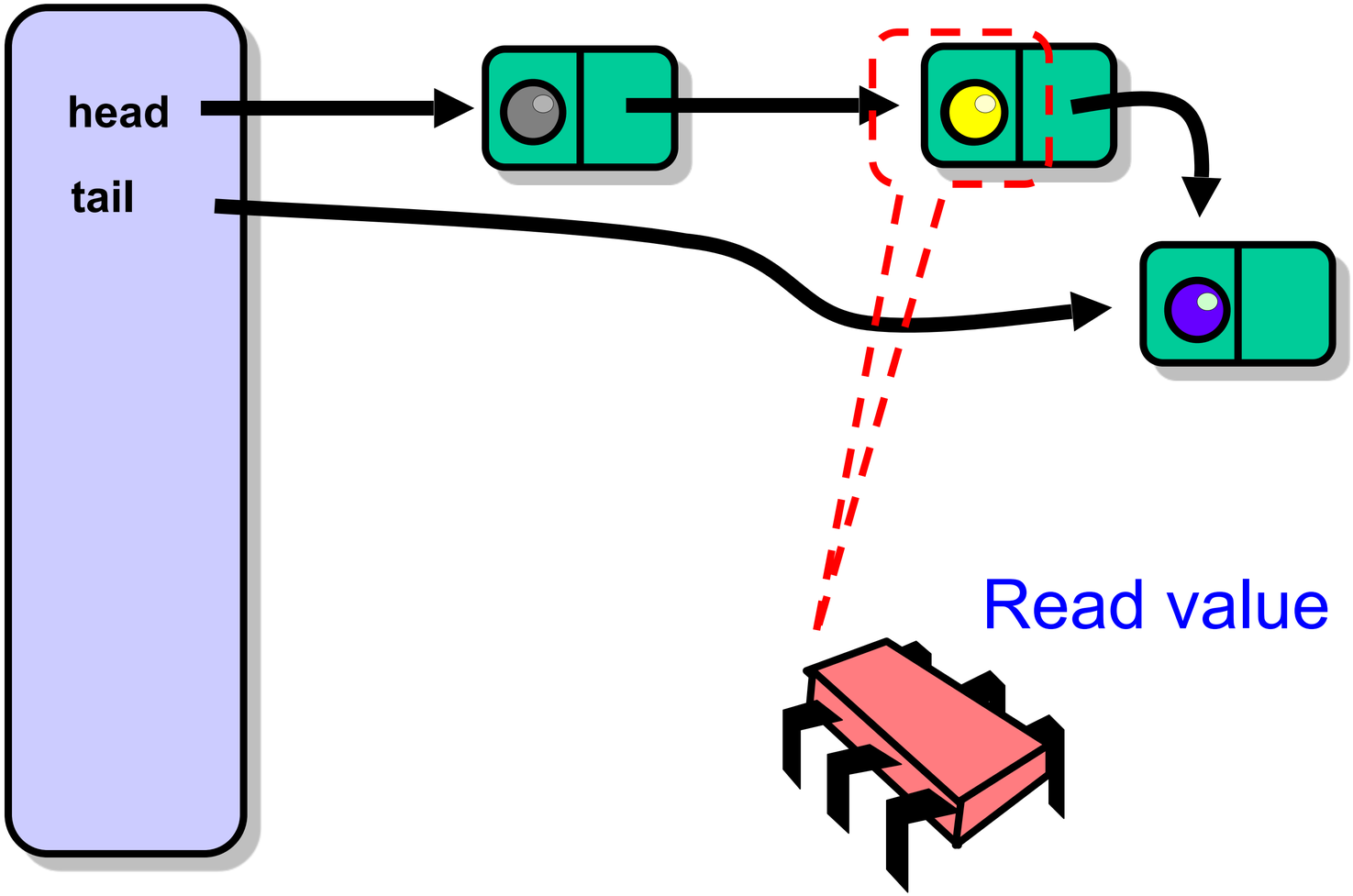
Enqueue

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

When CASs Fail

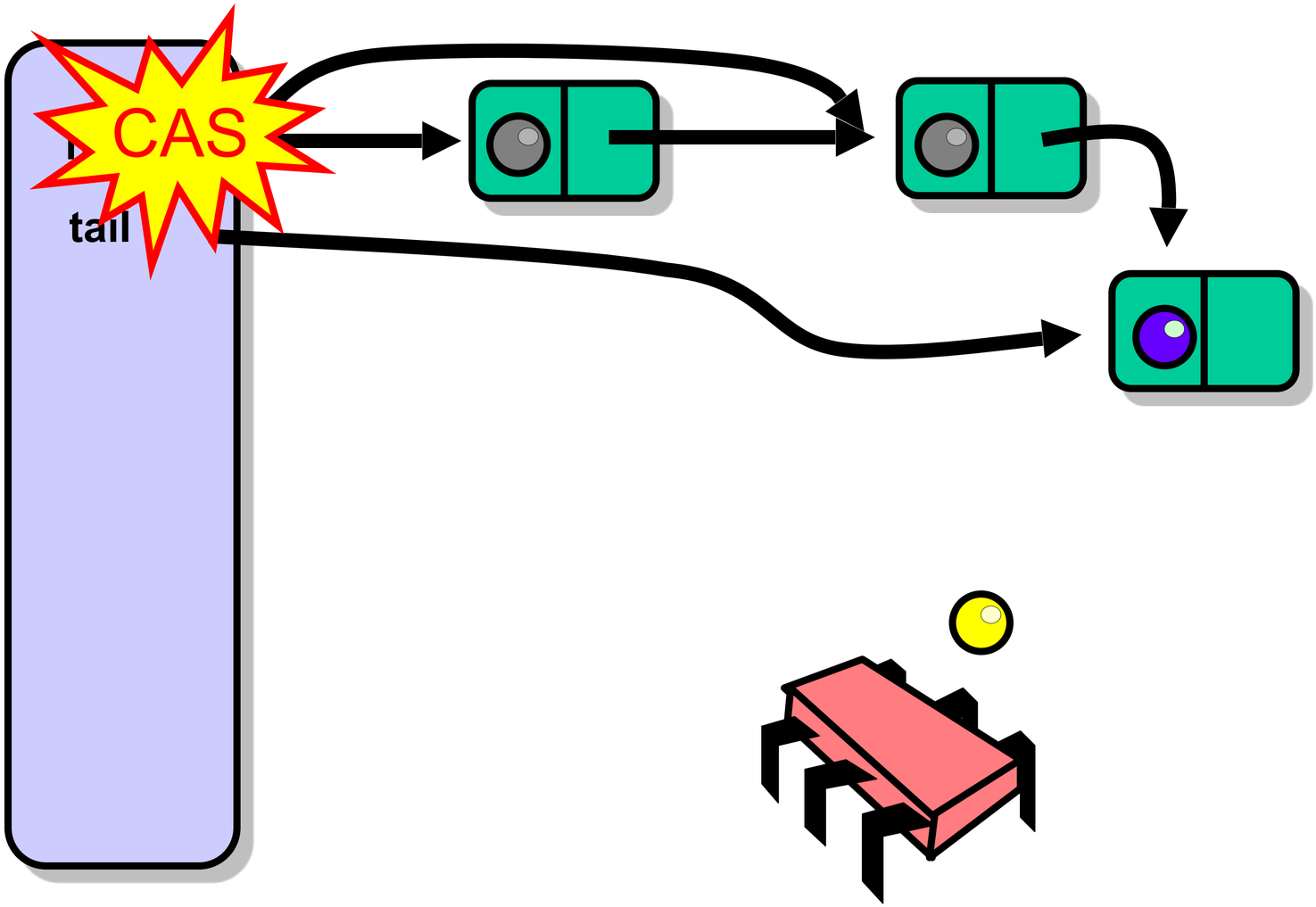
- During logical enqueue
  - Abandon hope, restart
  - Still lock-free (why?)
- During physical enqueue
  - Ignore it (why?)

Dequeuer



Make first Node new sentinel

Dequeuer

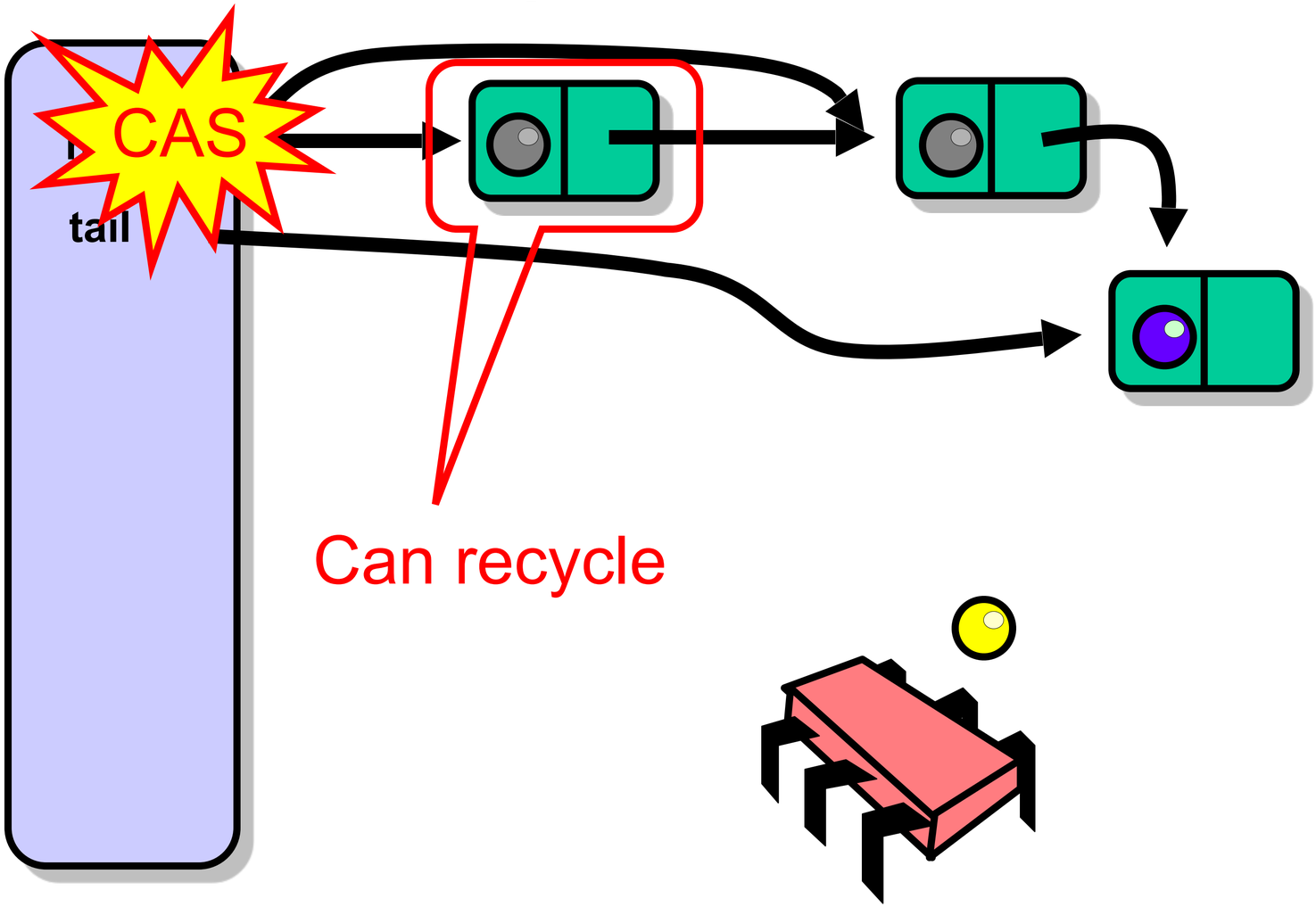


Checking the code of Lock-Free Queue

Memory Reuse?

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

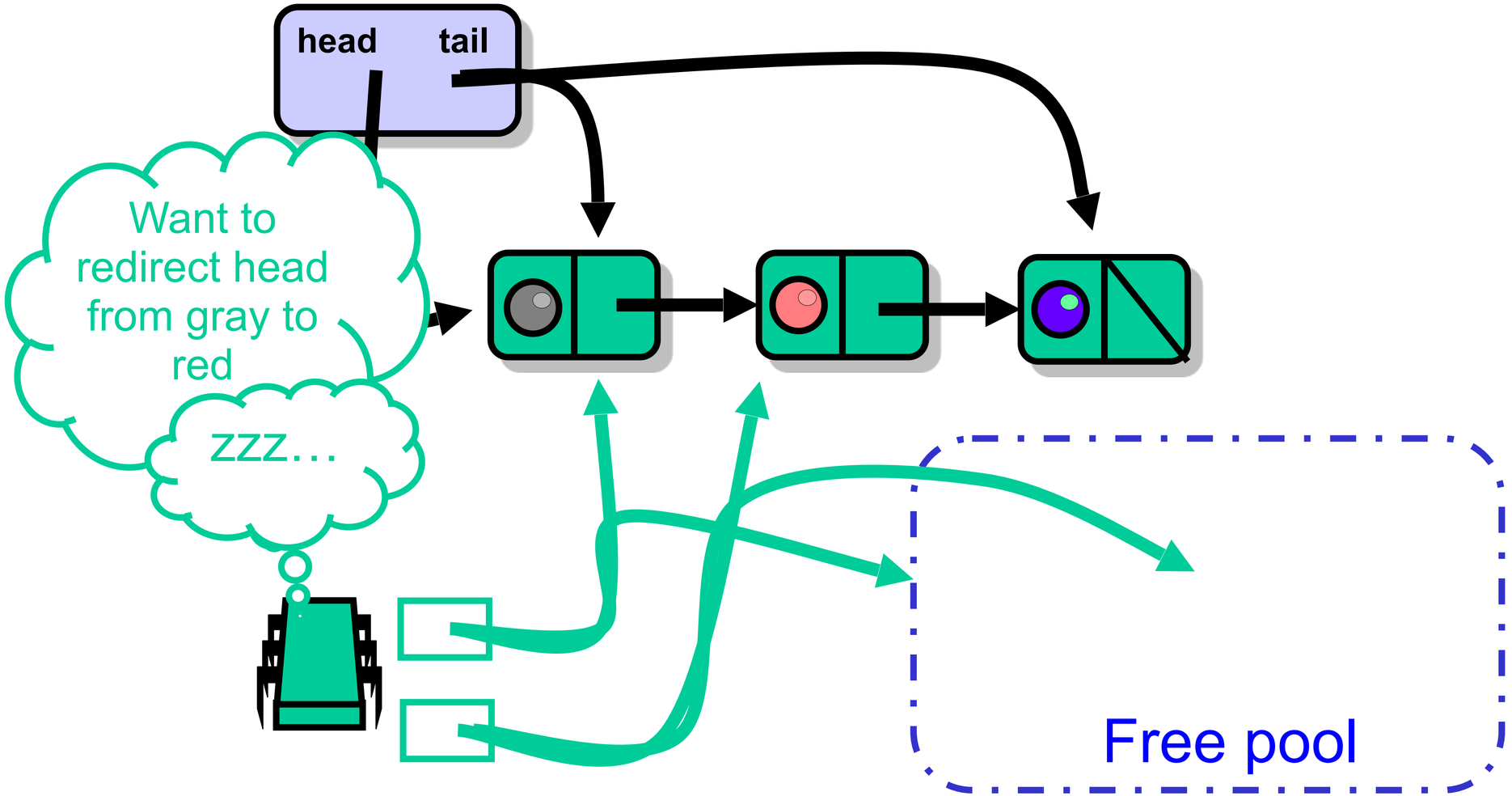
Dequeuer



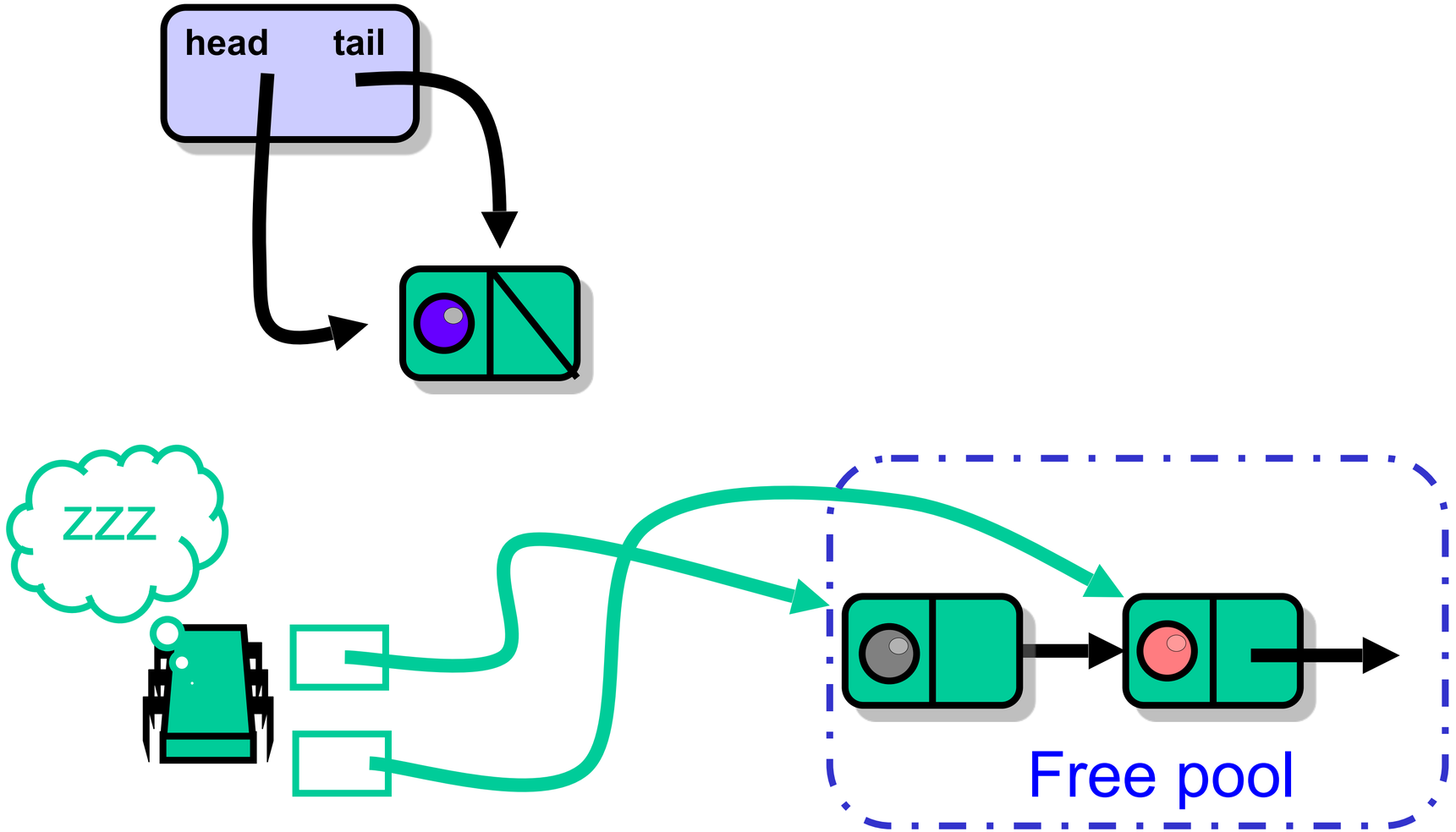
Simple Solution

- Each thread has a free list of unused queue nodes
- Allocate node: pop from list
- Free node: push onto list

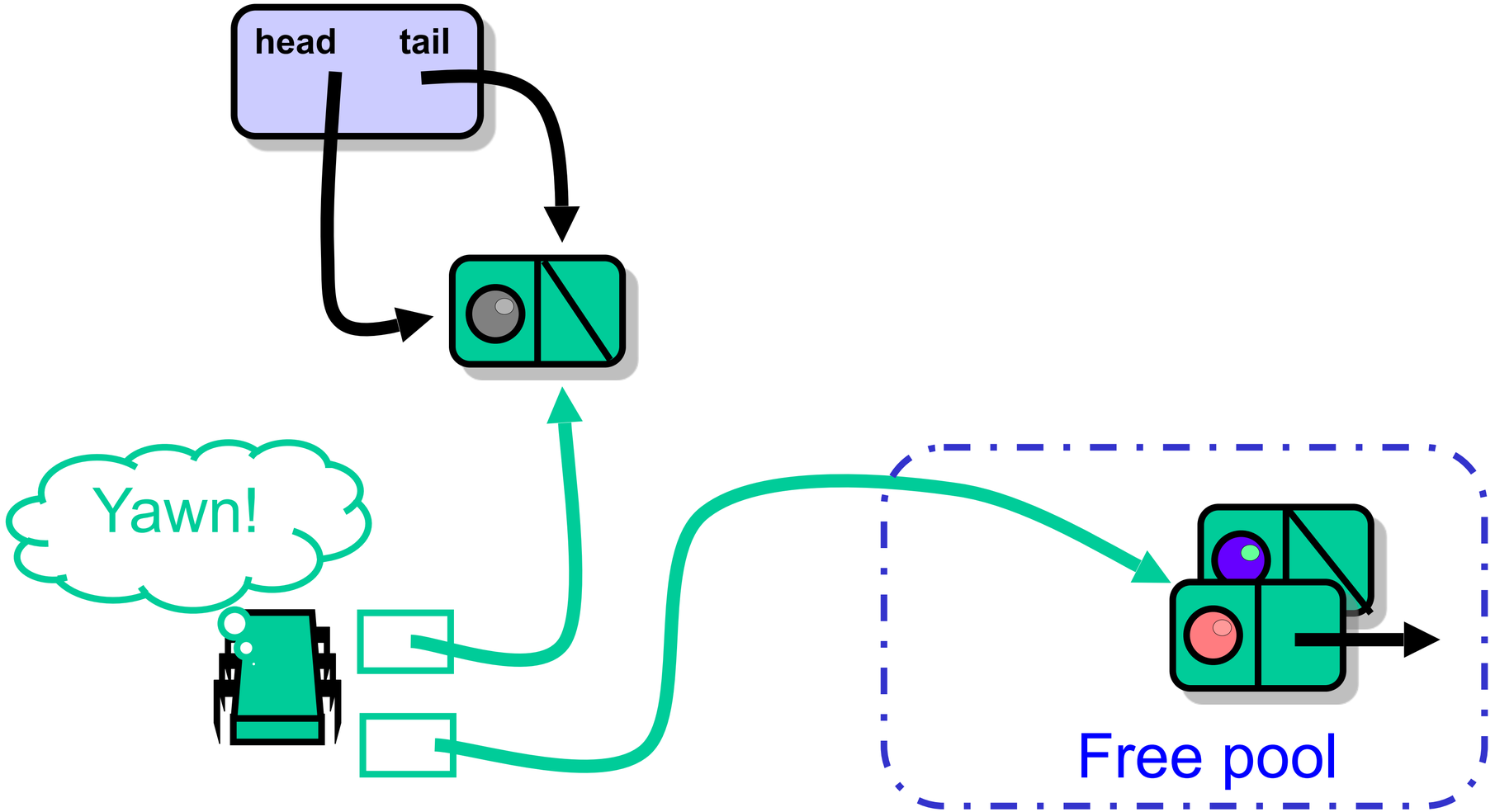
Why Recycling is Hard



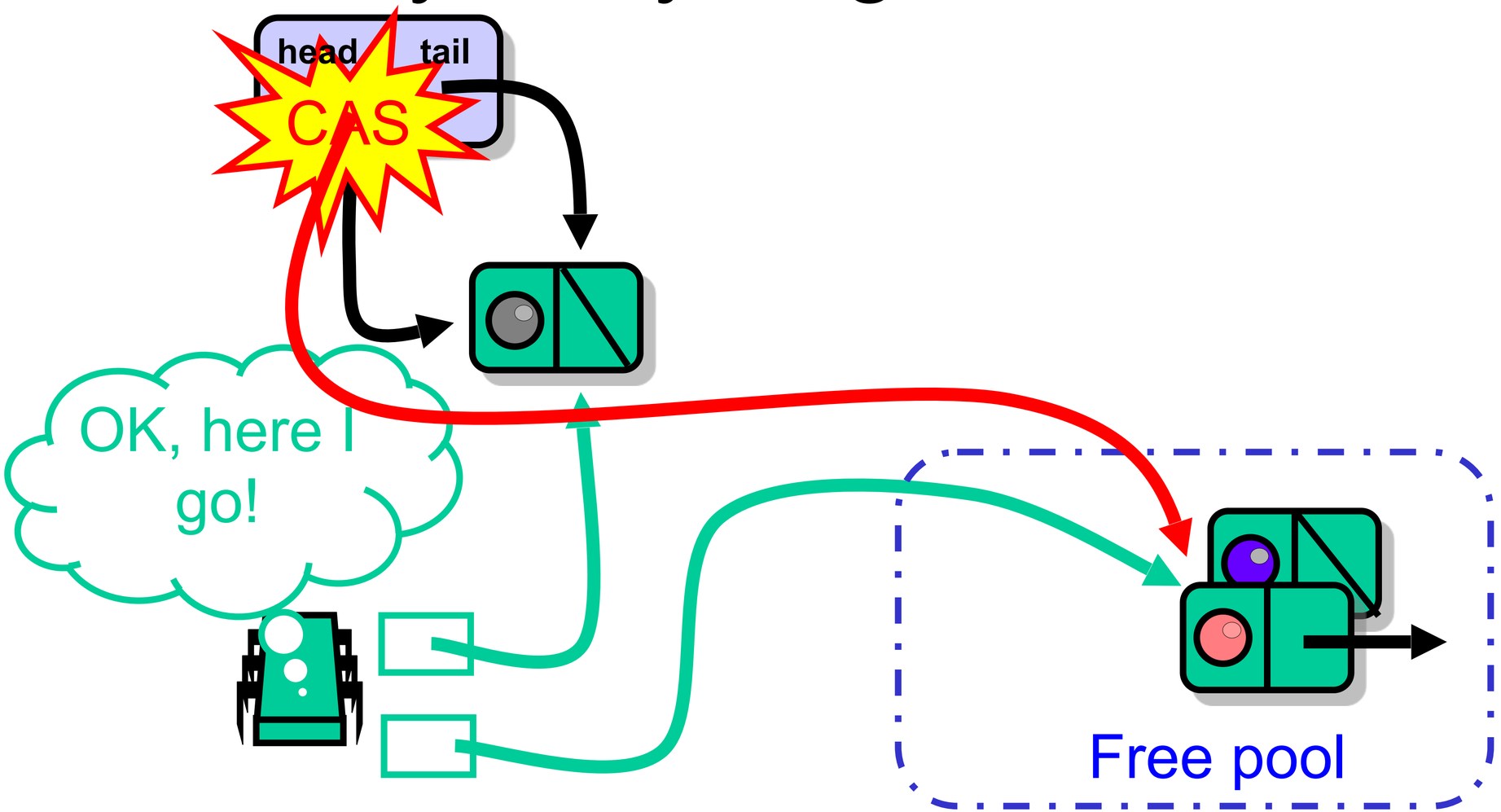
Both Nodes Reclaimed



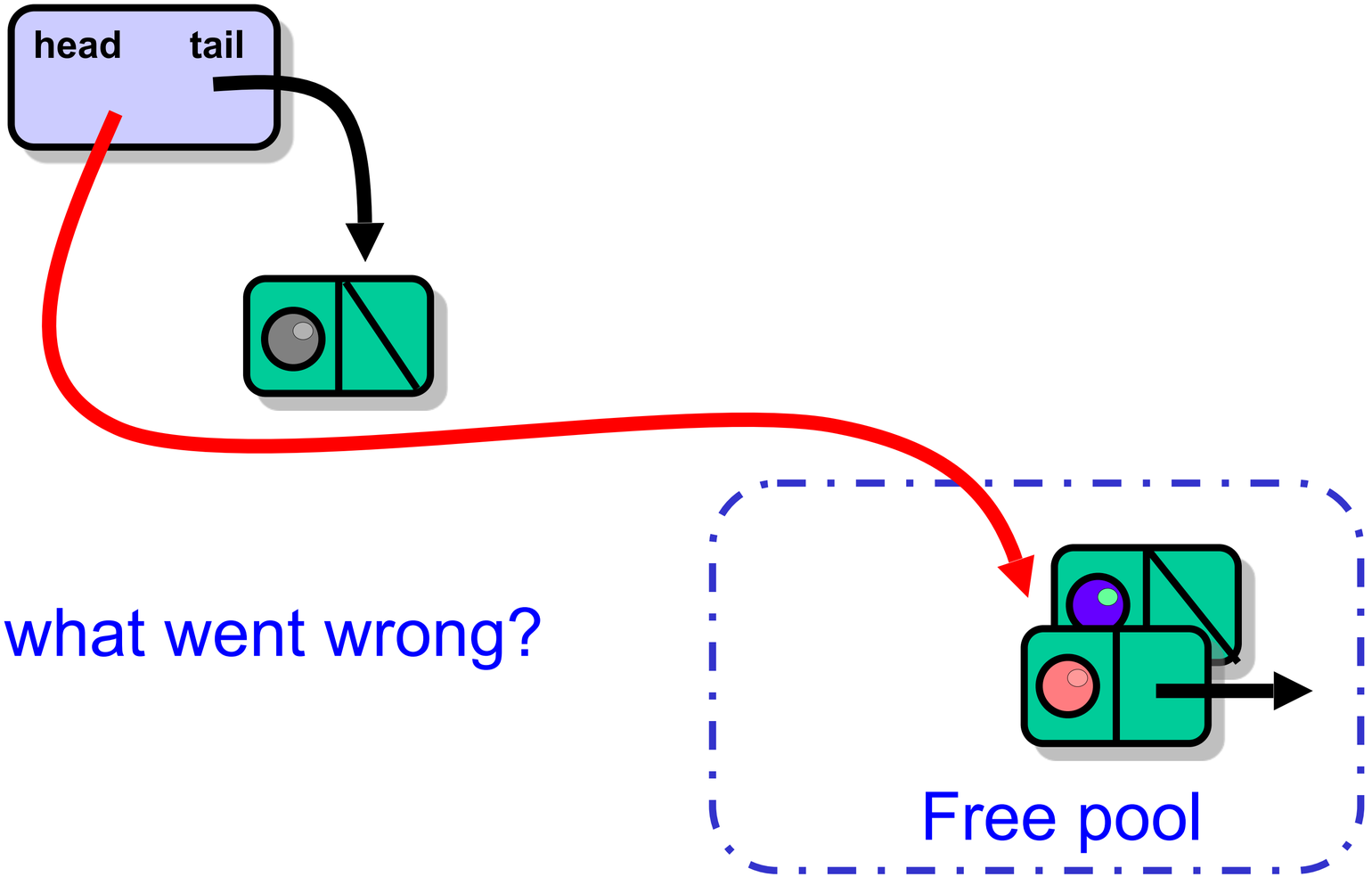
One Node Recycled



Why Recycling is Hard

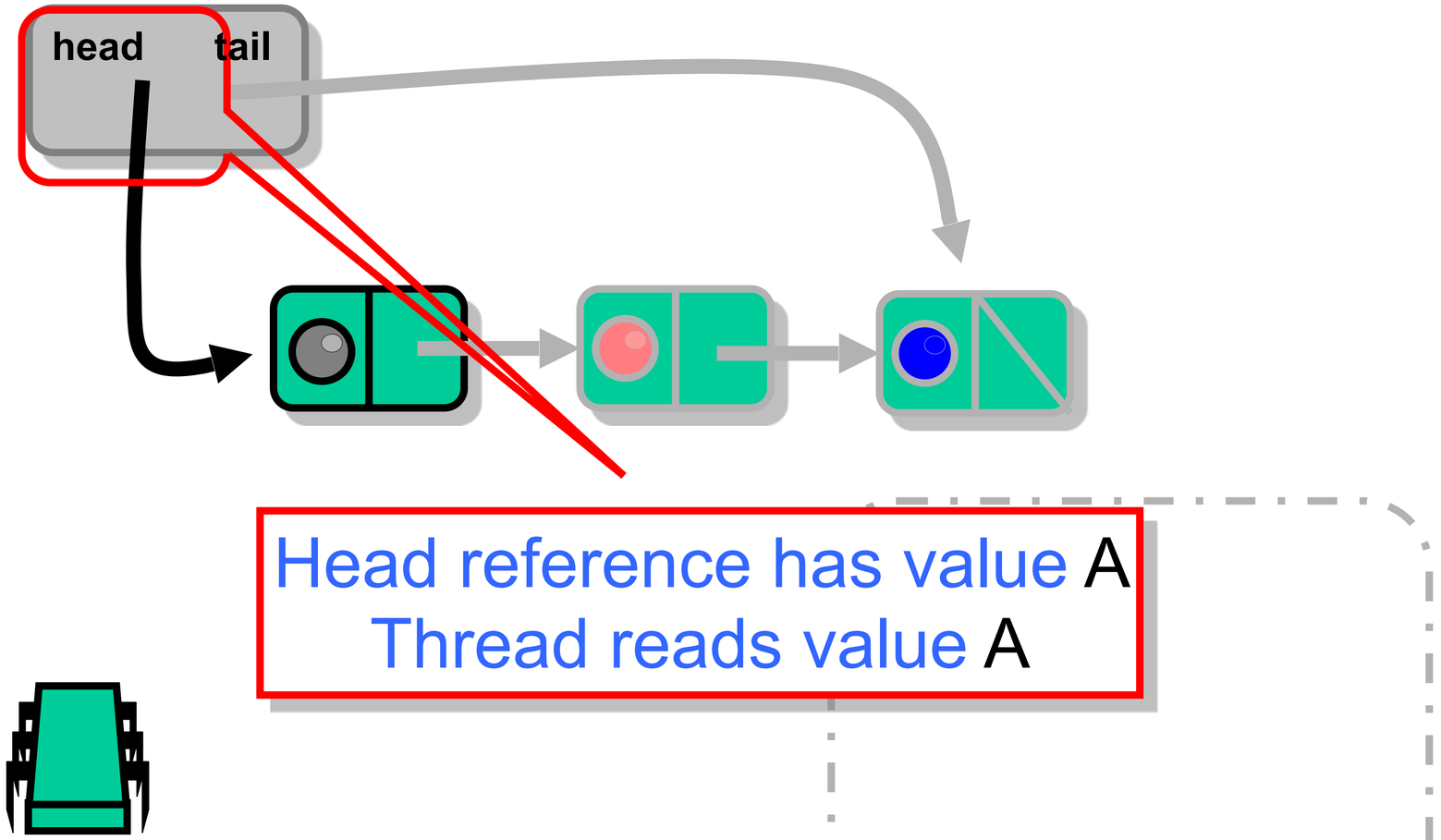


Recycle FAIL

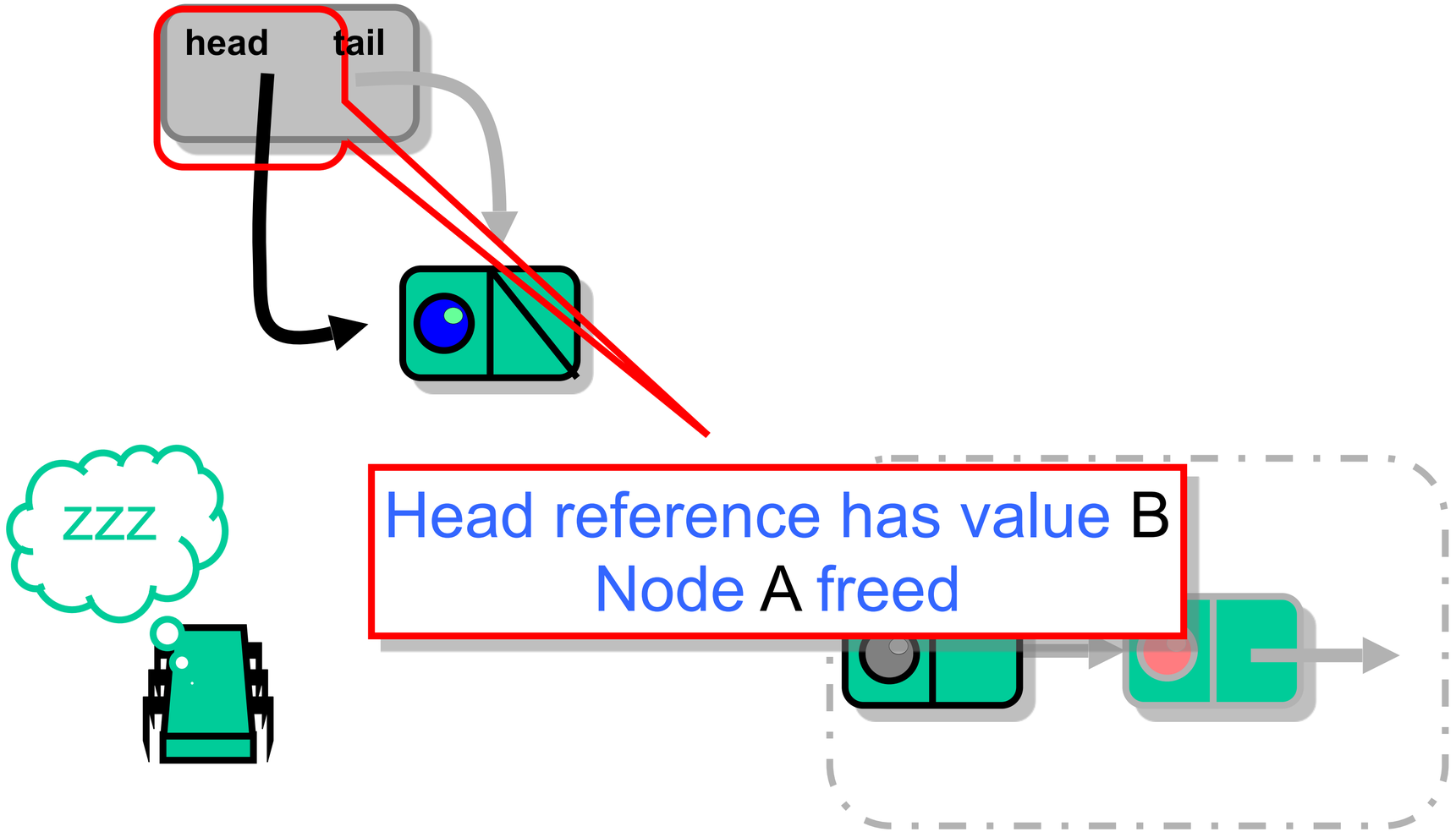


ZOMG what went wrong?

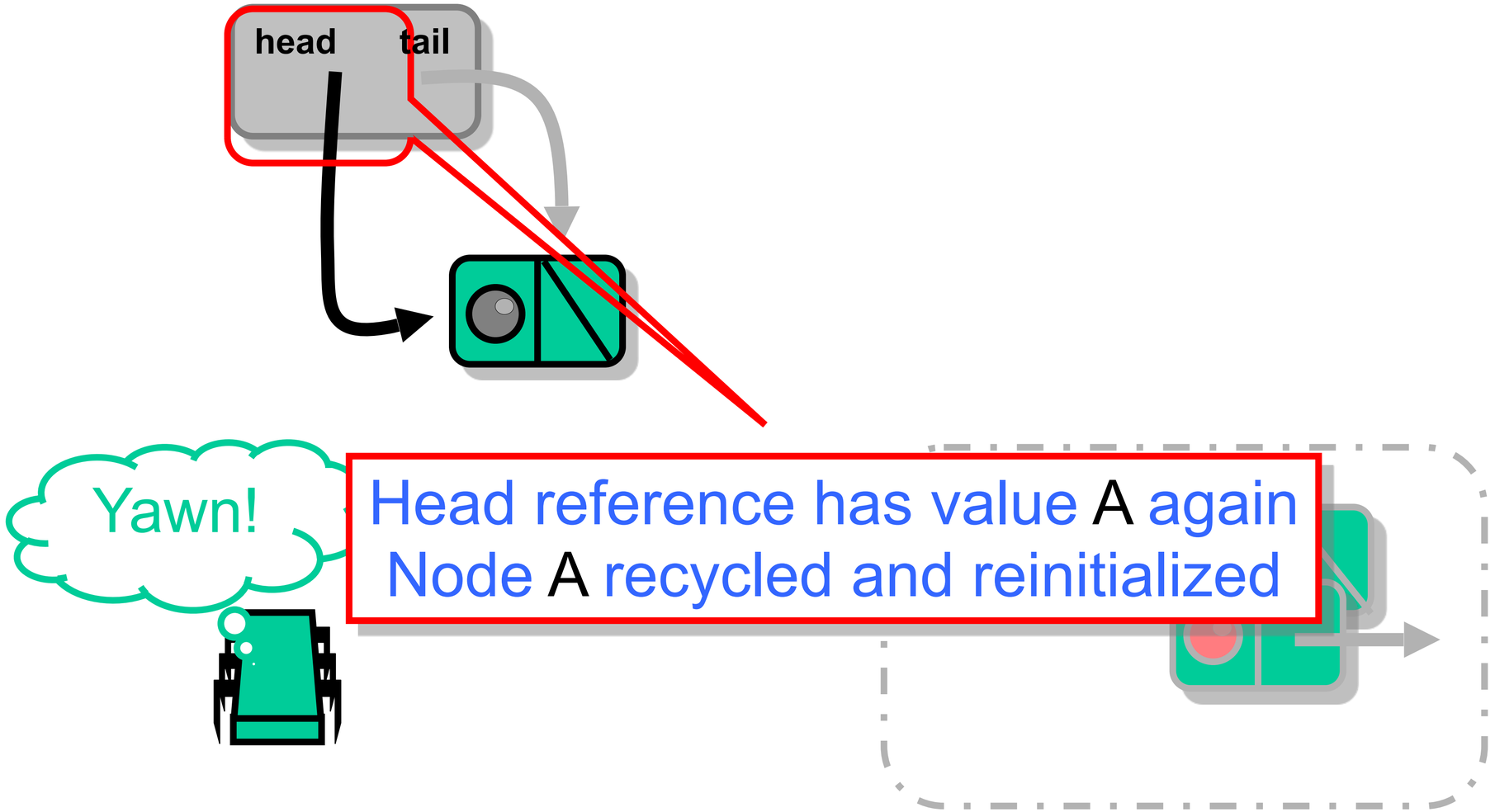
The Dreaded ABA Problem



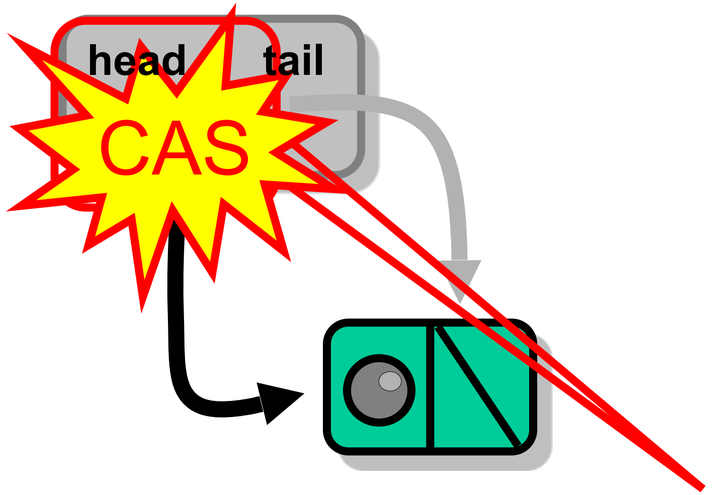
Dreaded ABA continued



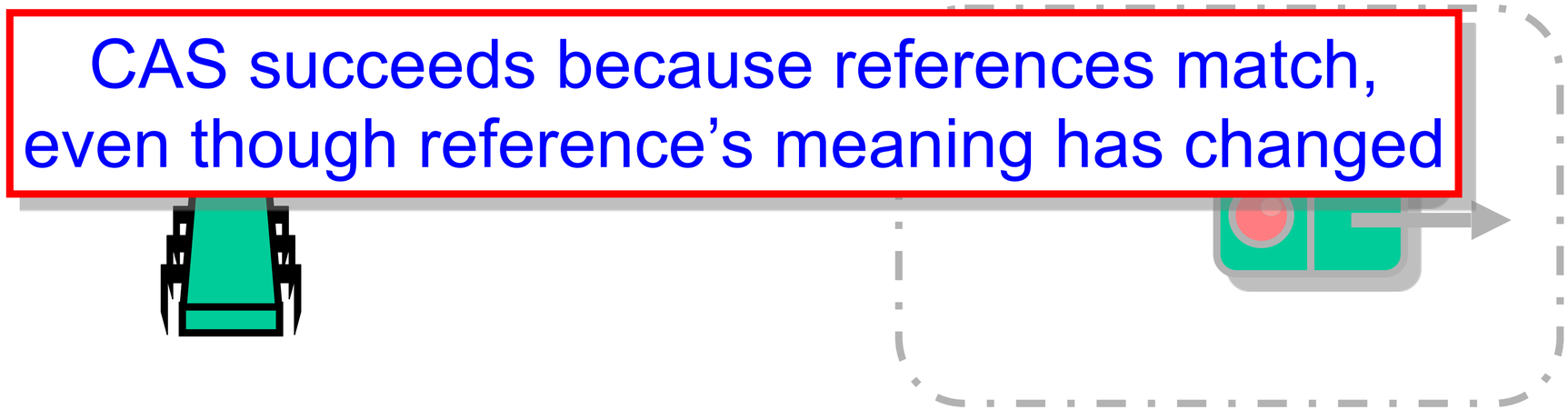
Dreaded ABA continued



Dreaded ABA continued



CAS succeeds because references match,
even though reference's meaning has changed



The Dreaded ABA FAIL

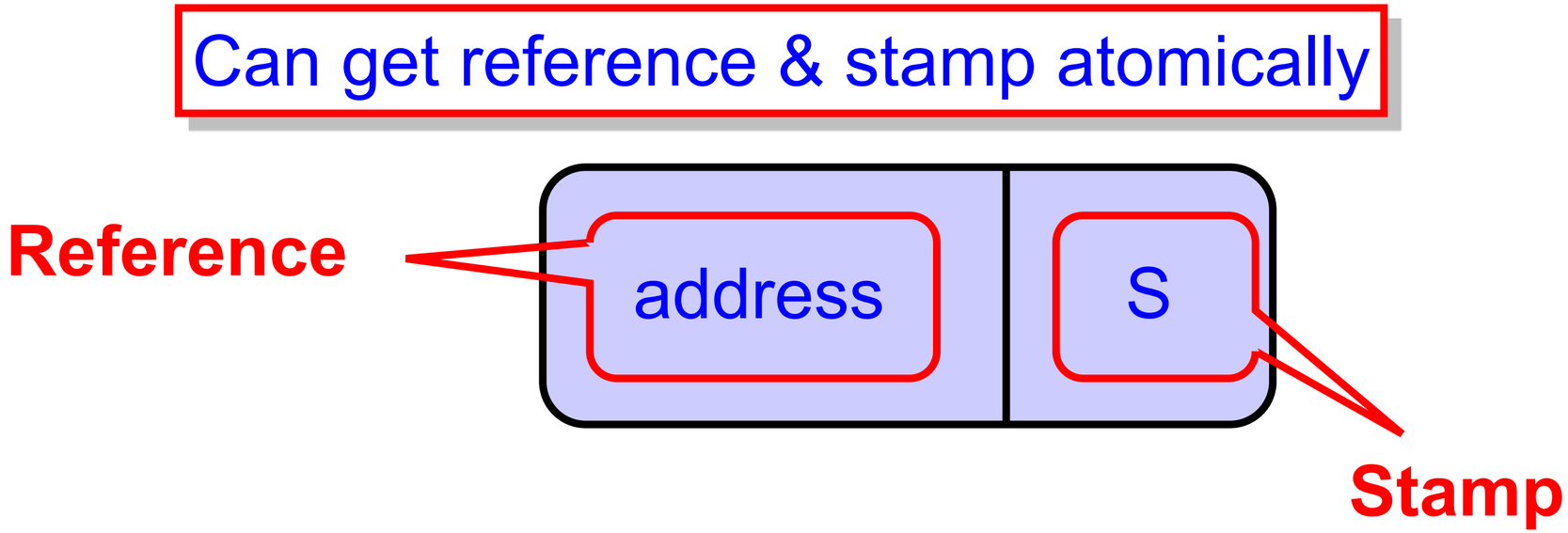
- Is a result of CAS() semantics
  - Oracle, Intel, AMD, …
- Not with Load-Locked/Store-Conditional
  - IBM …

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
- “Hazard Pointers”

Atomic Stamped Reference

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



Next:
Concurrent Stacks