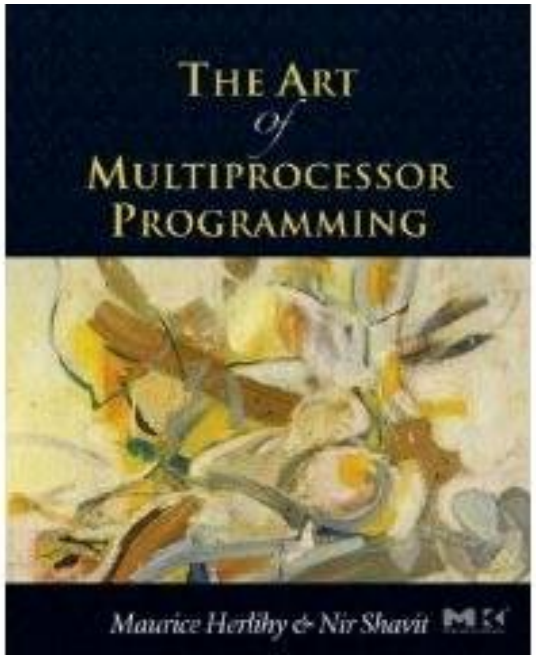
Department of Computer Science

COS 226

Chapter 10

Concurrent Queues and the ABA Problem

Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit

Pools

- Similar to Set class of Chapter 9 except:
  - Pool object does not necessarily have a contains() method
  - Same item can appear more than once

Bounded vs Unbounded

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

Total, partial or synchronous

- Total:
  - A method is total if calls do not wait for conditions to become true
  - For example:
    - An attempt to dequeue from an empty queue throws an exception or failure code

Total, partial or synchronous

- Partial:
  - A method may wait for conditions to hold
  - For example:
    - Thread that tries to dequeue from an empty queue blocks (suspends) until item becomes available

Total, partial or synchronous

- Synchronous:
  - A method is synchronous if it waits for another method to overlap its call interval
  - For example:
    - A method call that adds an item to the pool is blocked until that item is removed by another method call
    - Used by CSP and Ada for threads to rendevous and exchange values

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

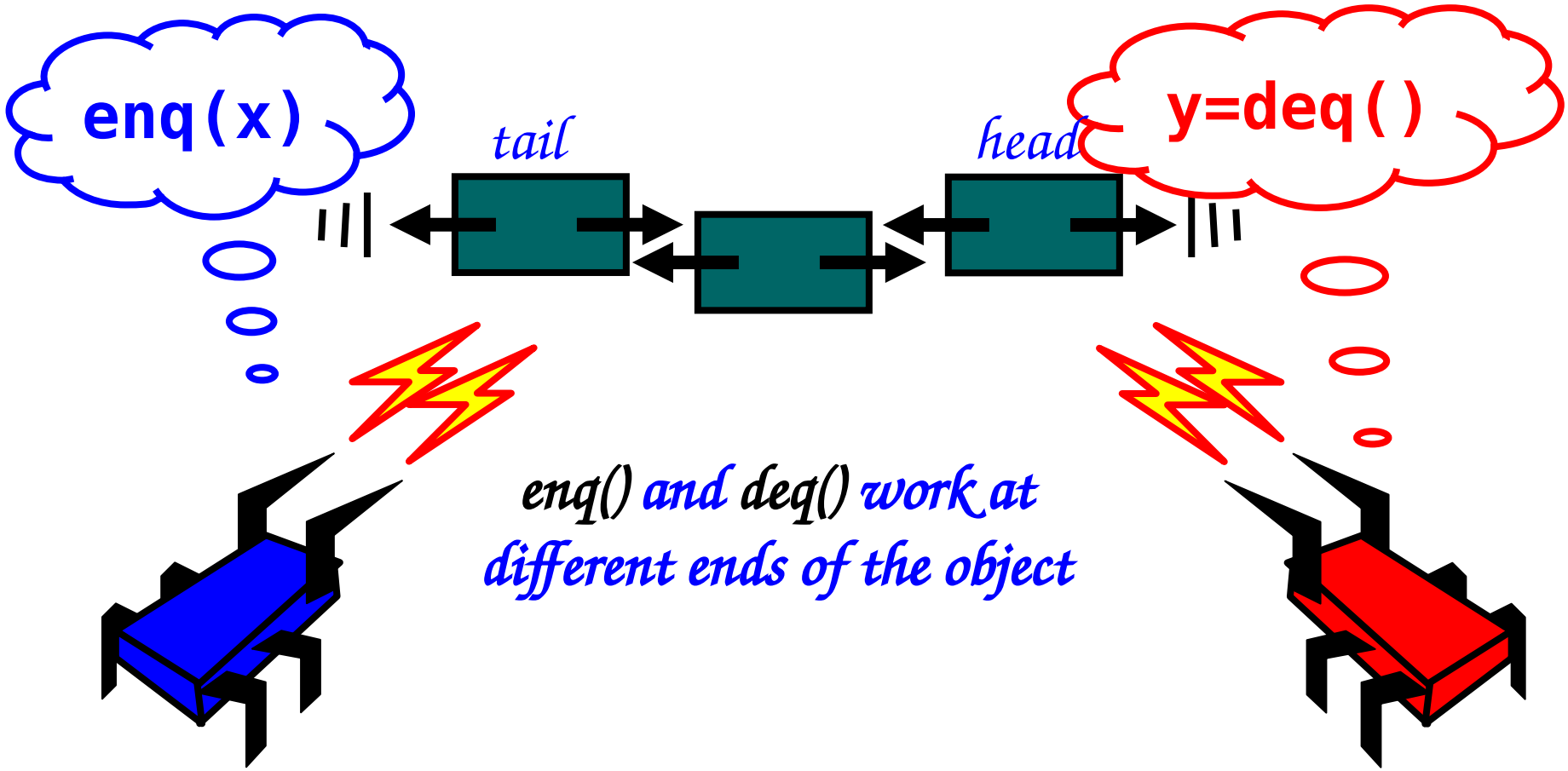
Bounded Queues

- How much concurrency can we expect a bounded queue implementation with multiple concurrent enqueuers and dequeuers to have?

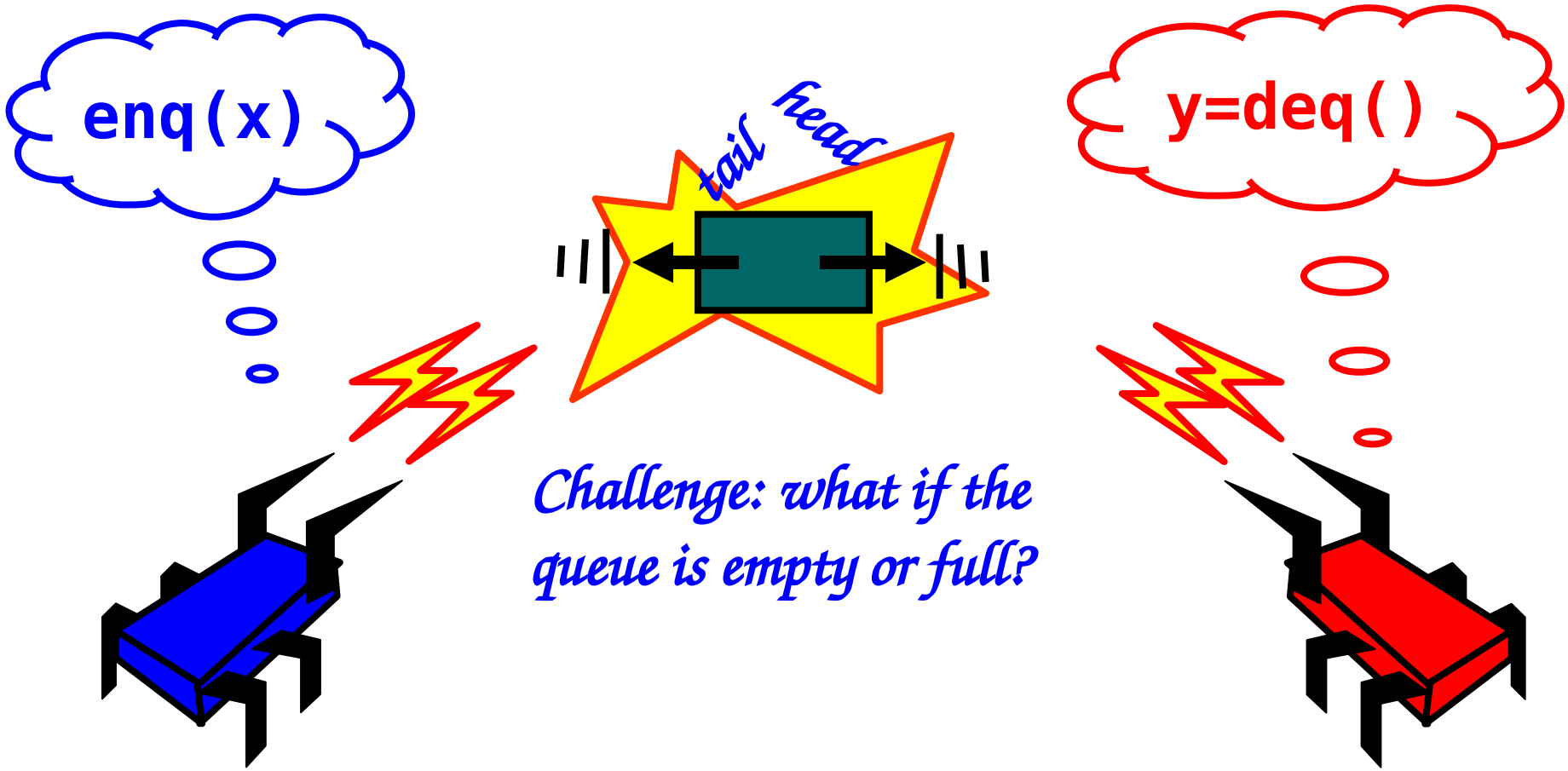
Bounded Queues

- Informally:
  - enq() and deq() methods operate on opposite ends of the queue
  - So an enq() call and a deq() call should be able to proceed concurrently without interference
  - Concurrent deq() calls and concurrent enq() calls will however interfere

Queue: Concurrency



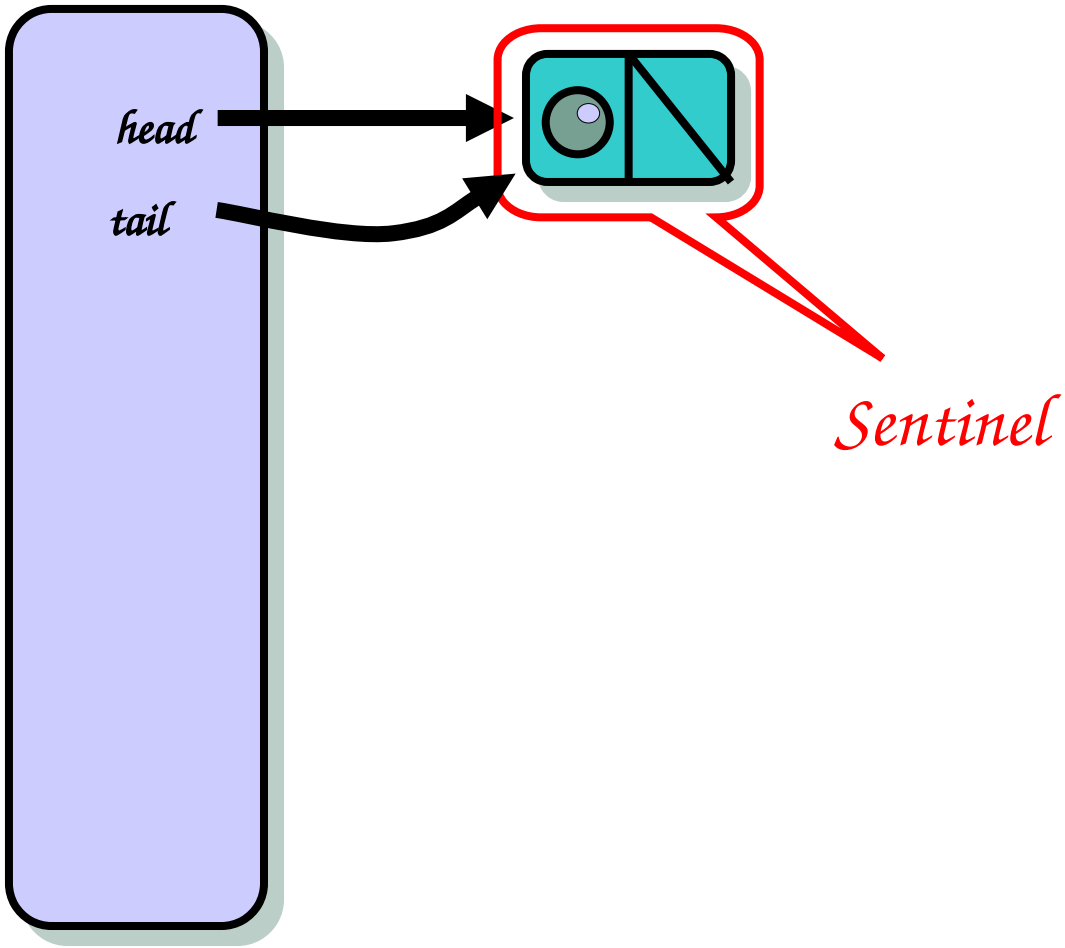
Concurrency



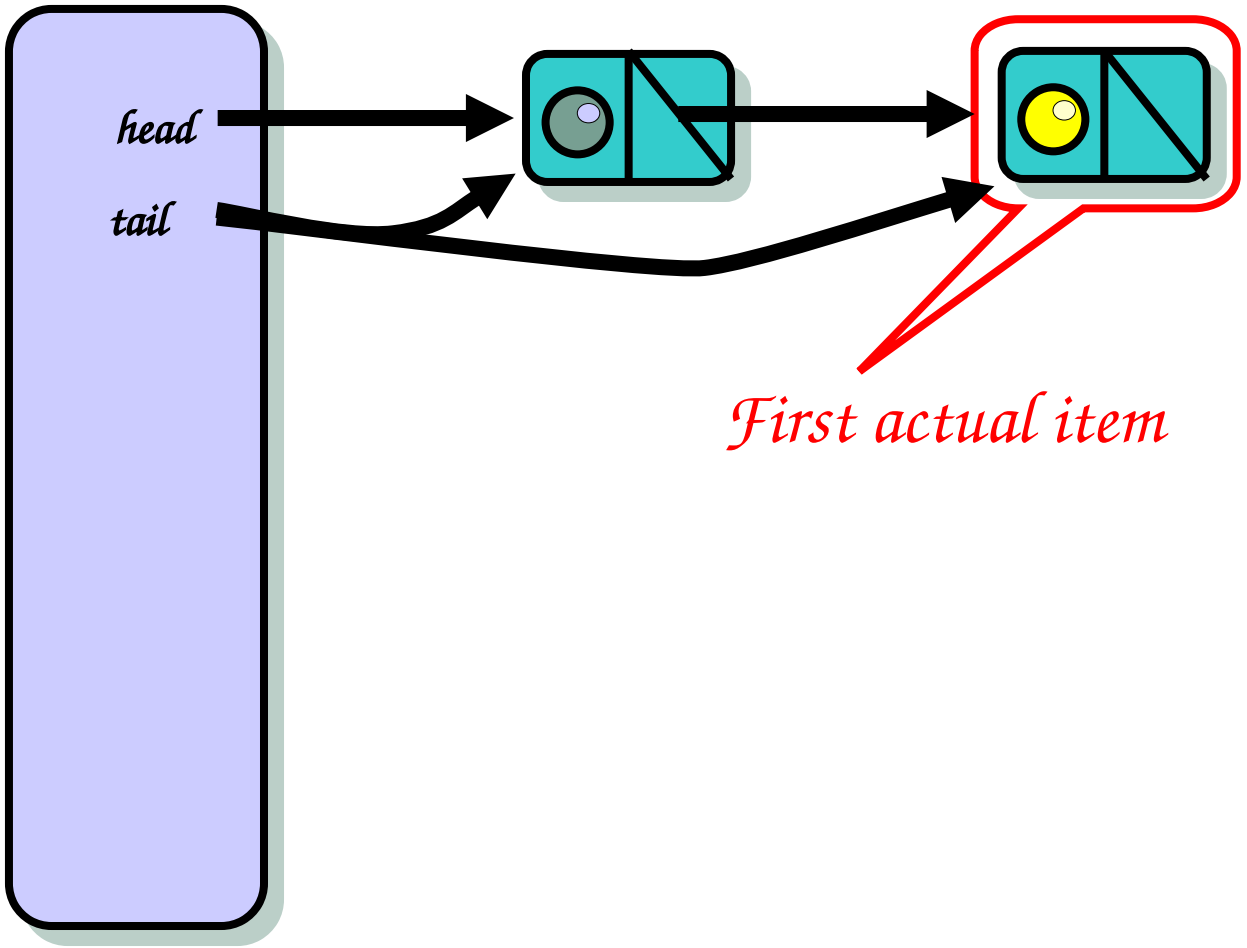
Bounded lock-based queue

- Implement queue as linked list
- Queue has head and tail fields that refer to first and last nodes in list
- The queue contains a sentinel node – a place-holder

Bounded Queue



Bounded Queue



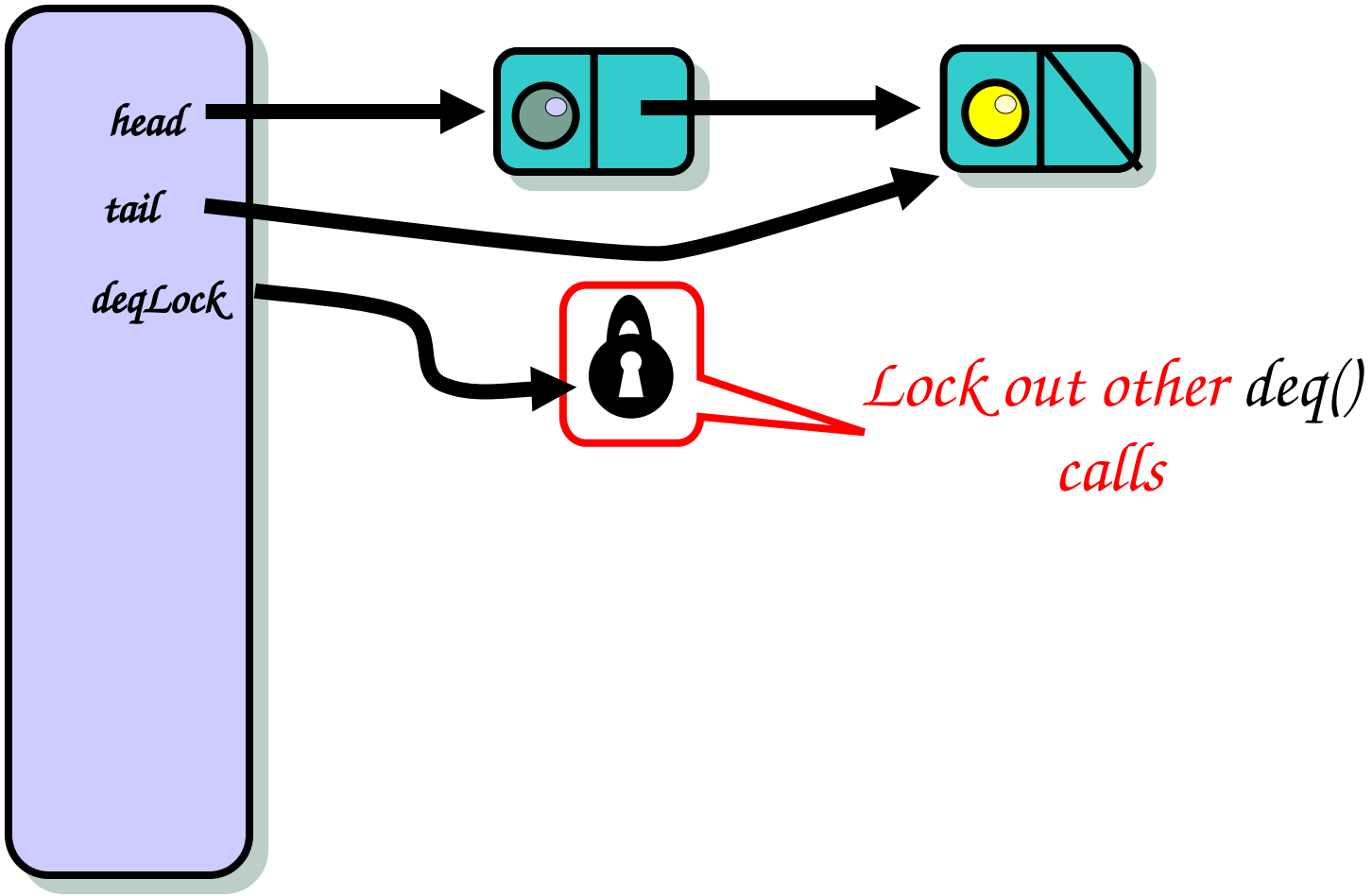
Bounded Queue

- Two distinct locks are used:
  - deqlock – to lock the front of the queue to ensure that there is not more than one dequeuer at a time
  - enqlock – to lock the end of the queue to ensure that there is not more than one enqueuer at a time

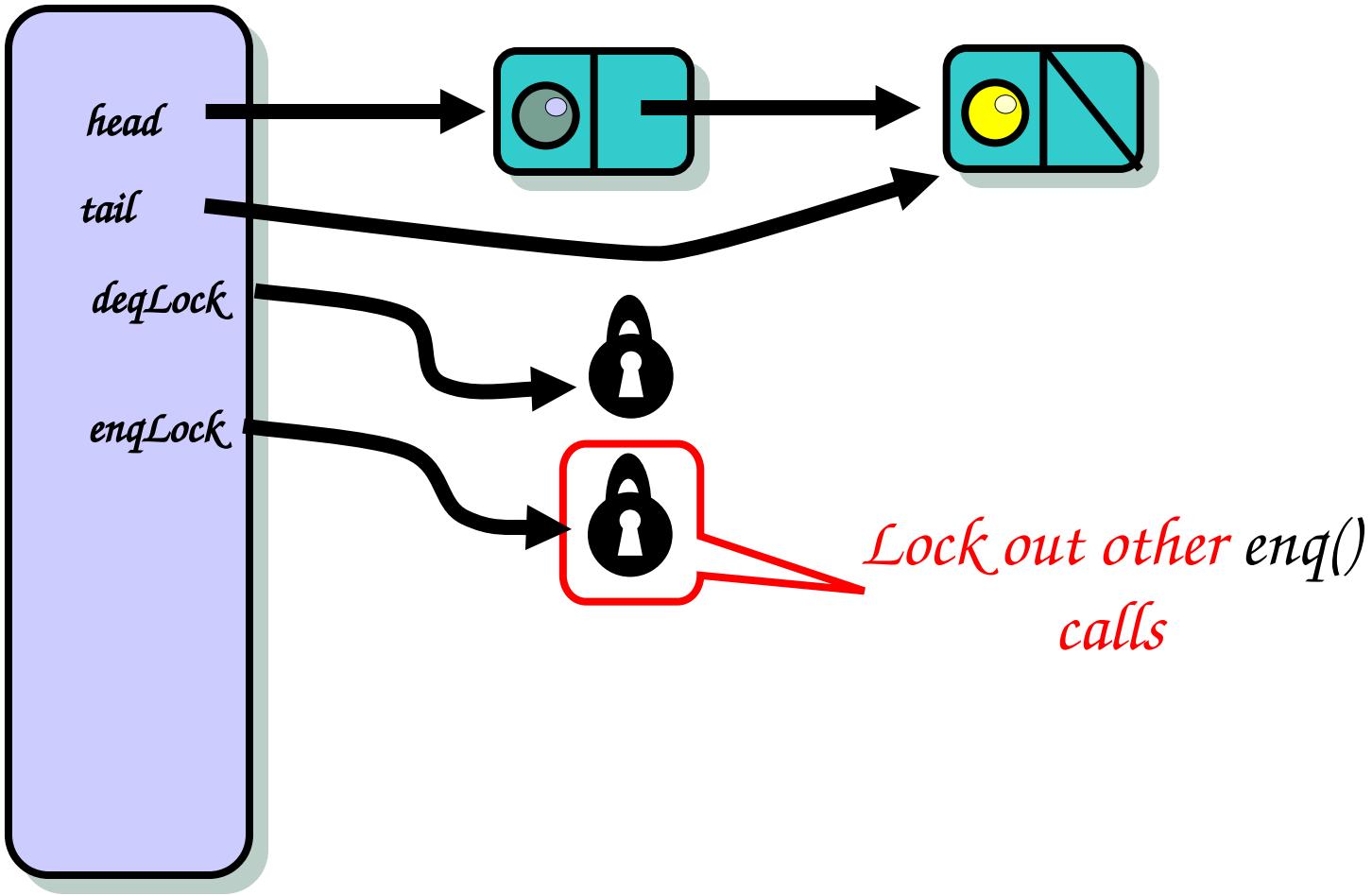
Bounded Queue

- Using two locks instead of one means that enqueuers and dequeuers can operate independently and does not lock one another out unnecessarily

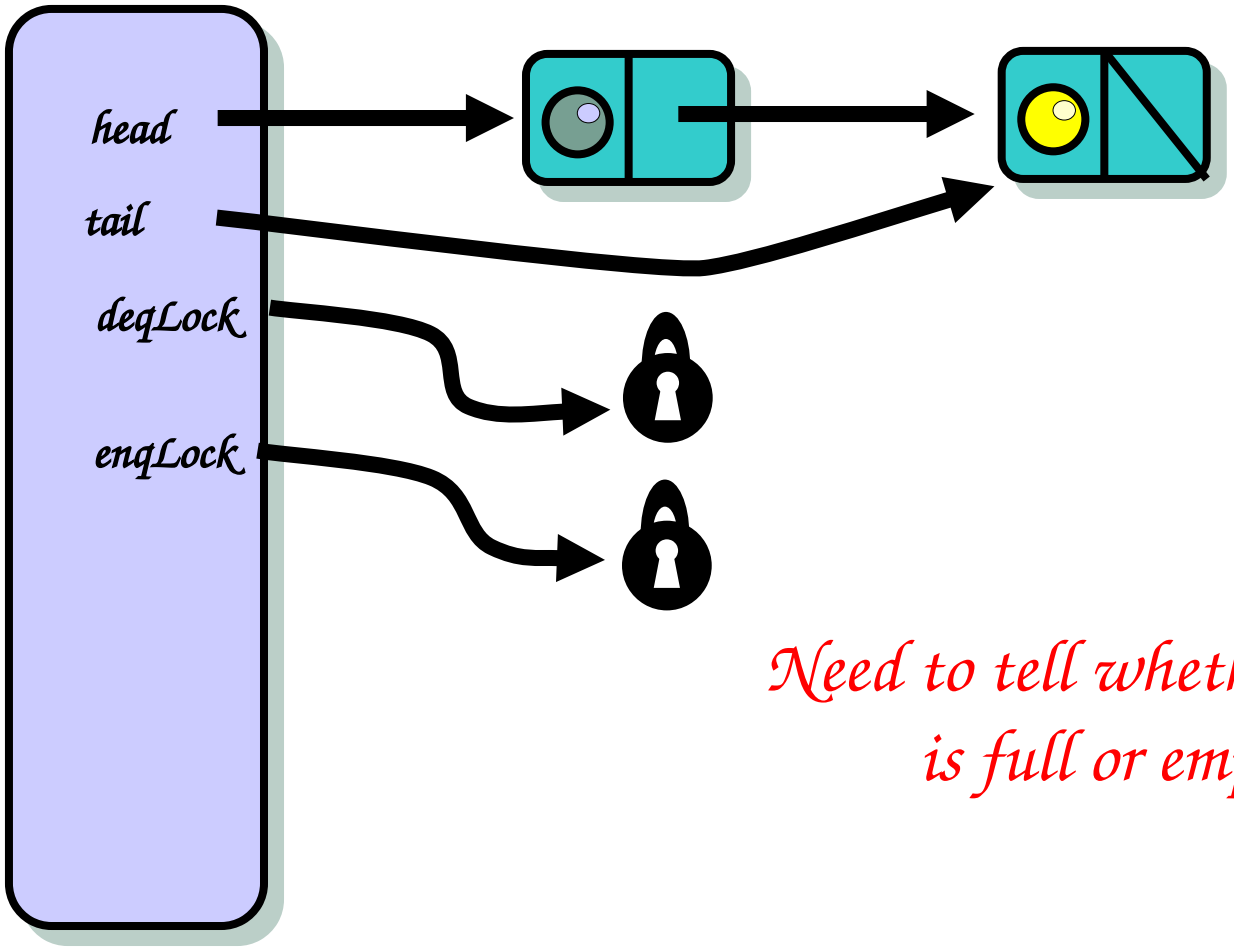
Bounded Queue



Bounded Queue



Not Done Yet



Need to tell whether queue is full or empty

Bounded Queues

- enq() calls cannot continue while list is full
- deq() calls cannot continue while list is empty

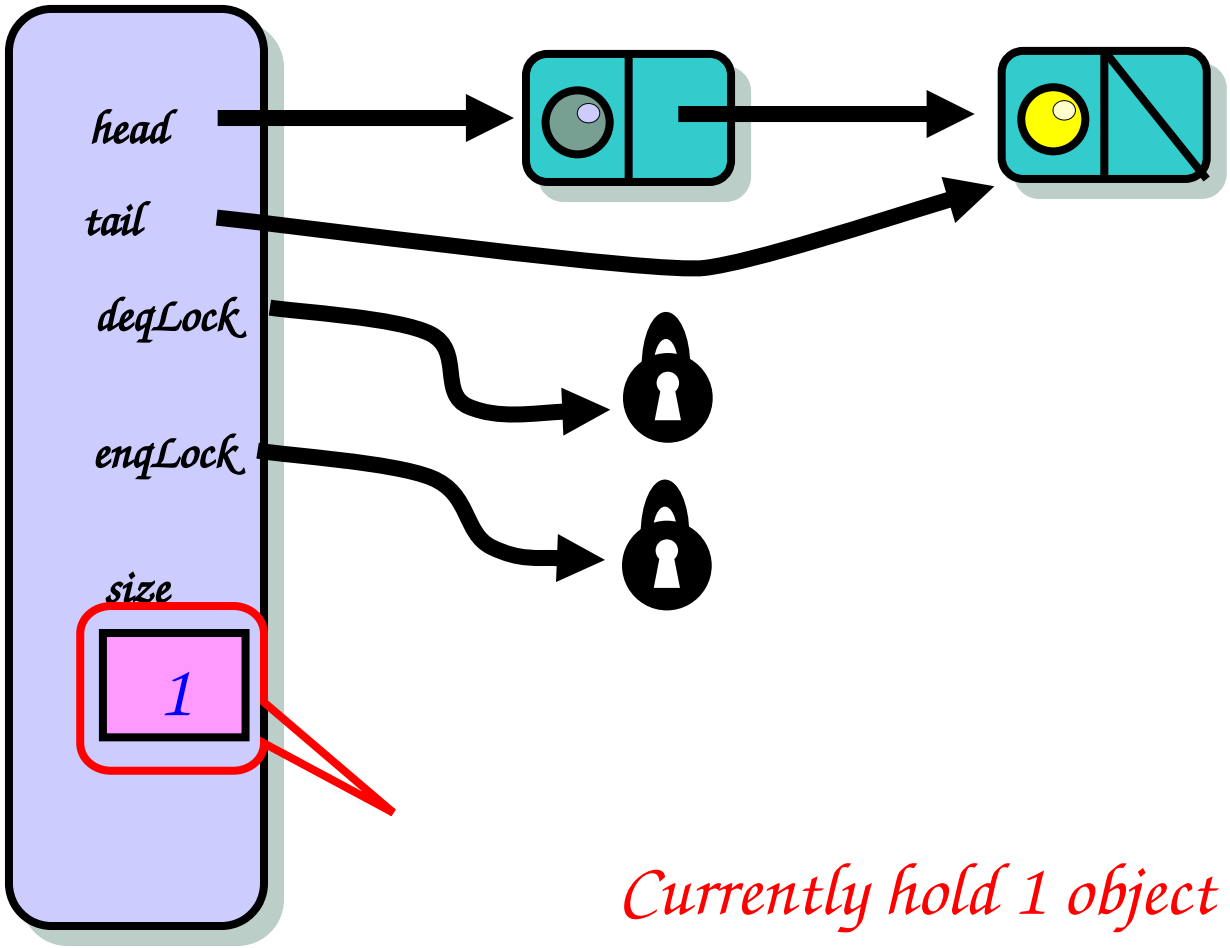
Bounded Queue

- Each lock has an associated condition field:
  - enqlock is associated with the notFullCondition
    - Notifies enqueuers when list is not full anymore
  - deqlock is associated with the notEmptyCondition
    - Notifies dequeuers when list is not empty anymore

Bounded Queue

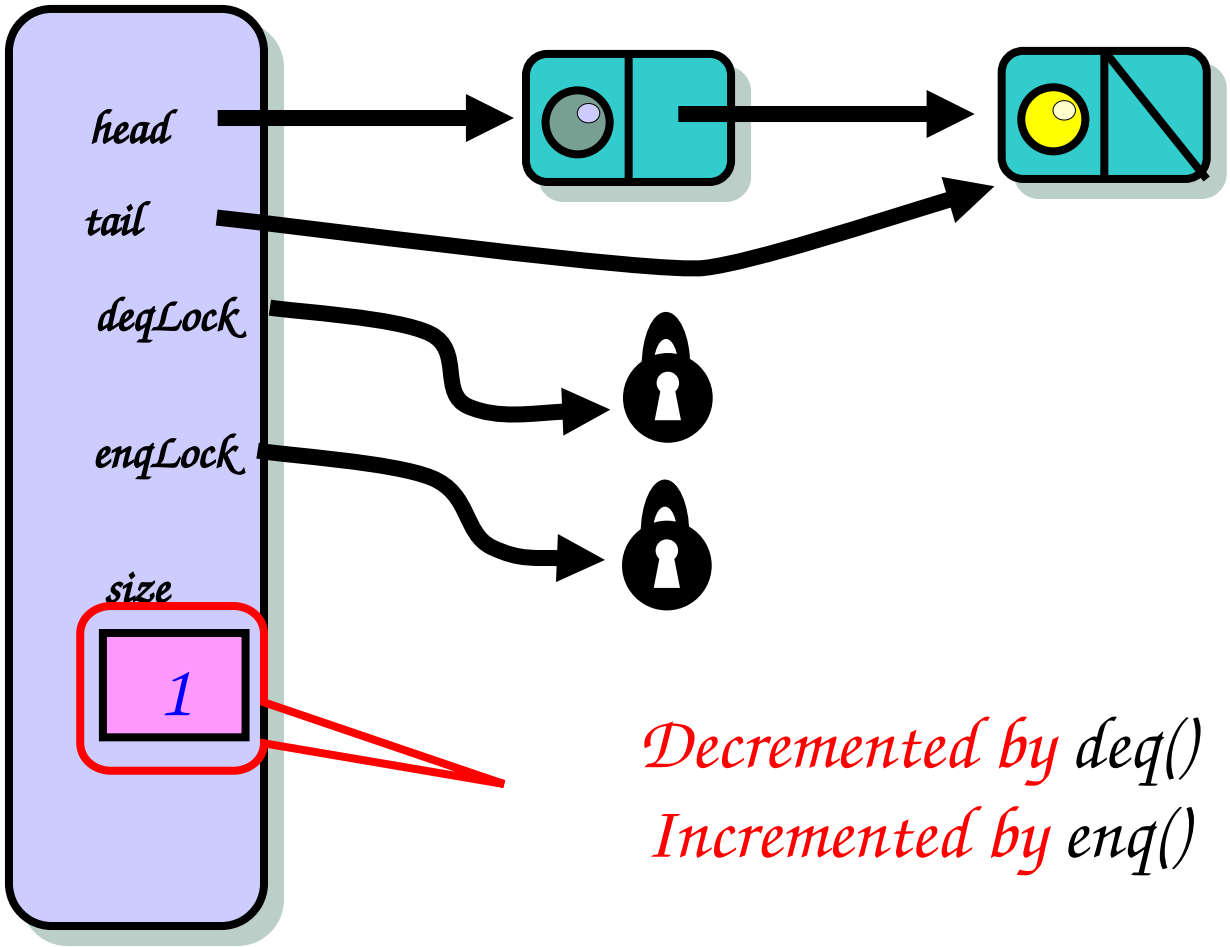
- Since the queue is bounded we must also keep track of the number of empty/available slots
- The size field keeps track of the number of objects currently in the queue
- The size field is an AtomicInteger

Not Done Yet

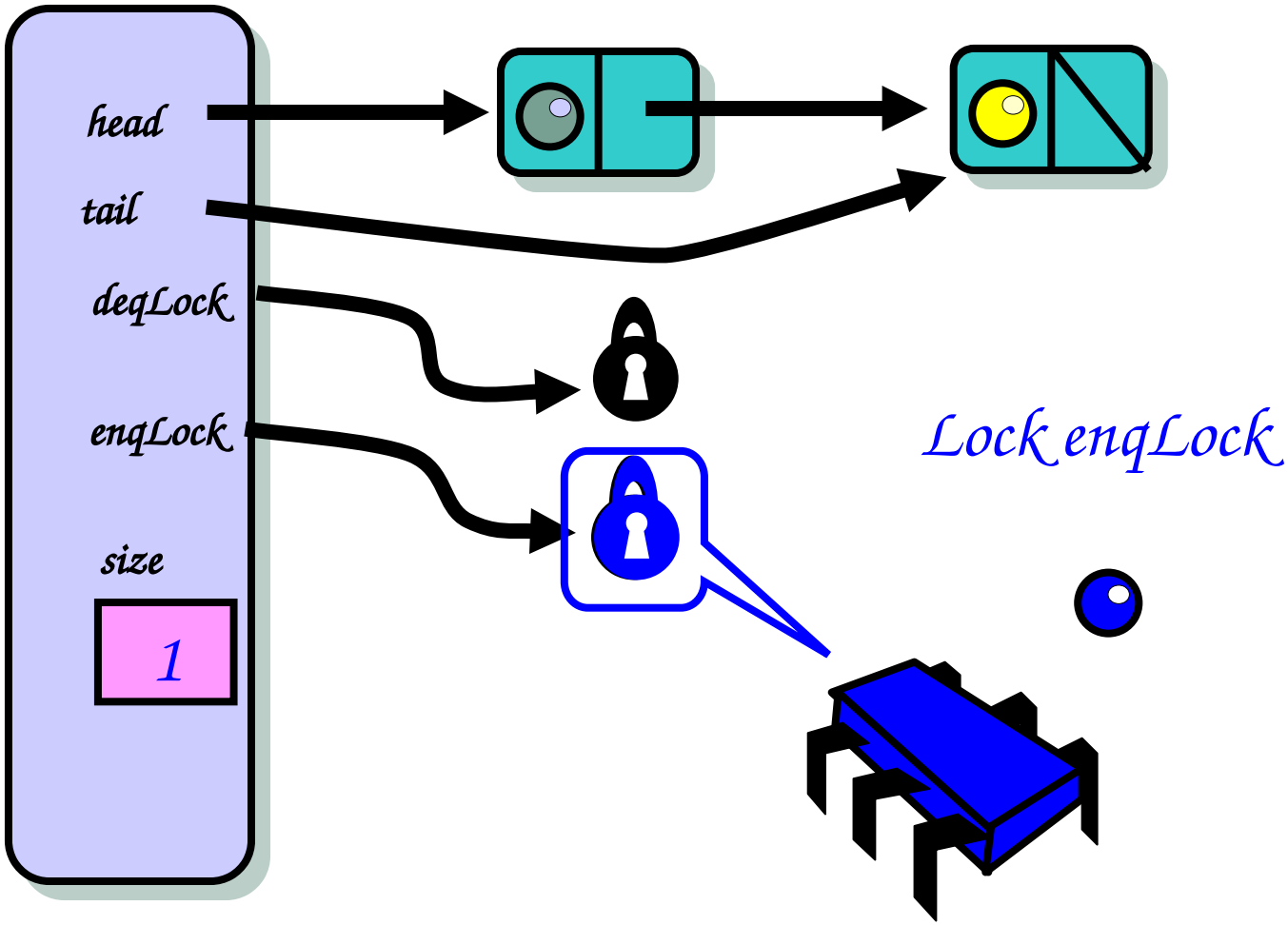


Currently hold 1 object

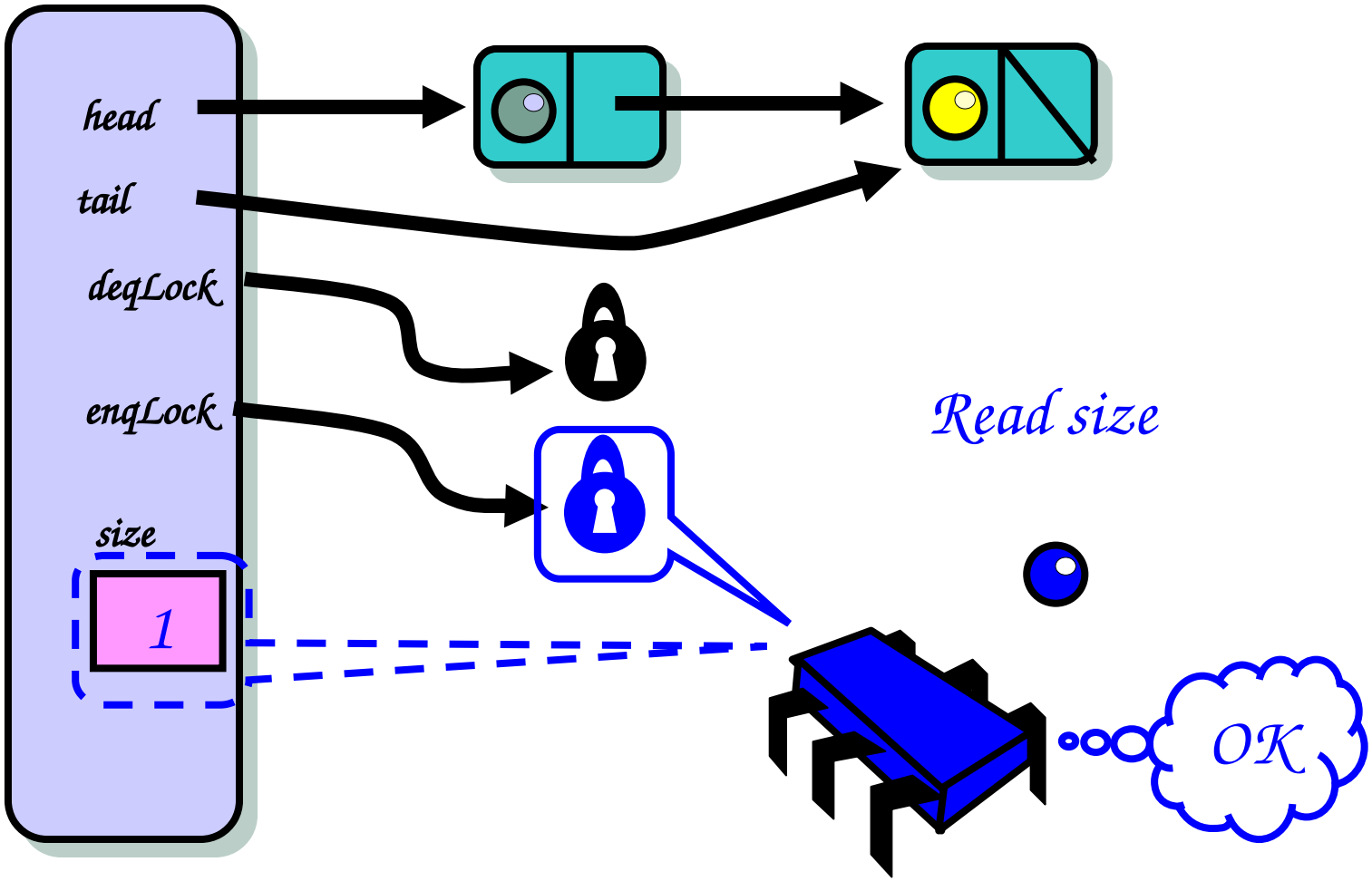
Not Done Yet


head
tail
deqLock
enqLock
size
1
Decremented by deq()
Incremented by enq()

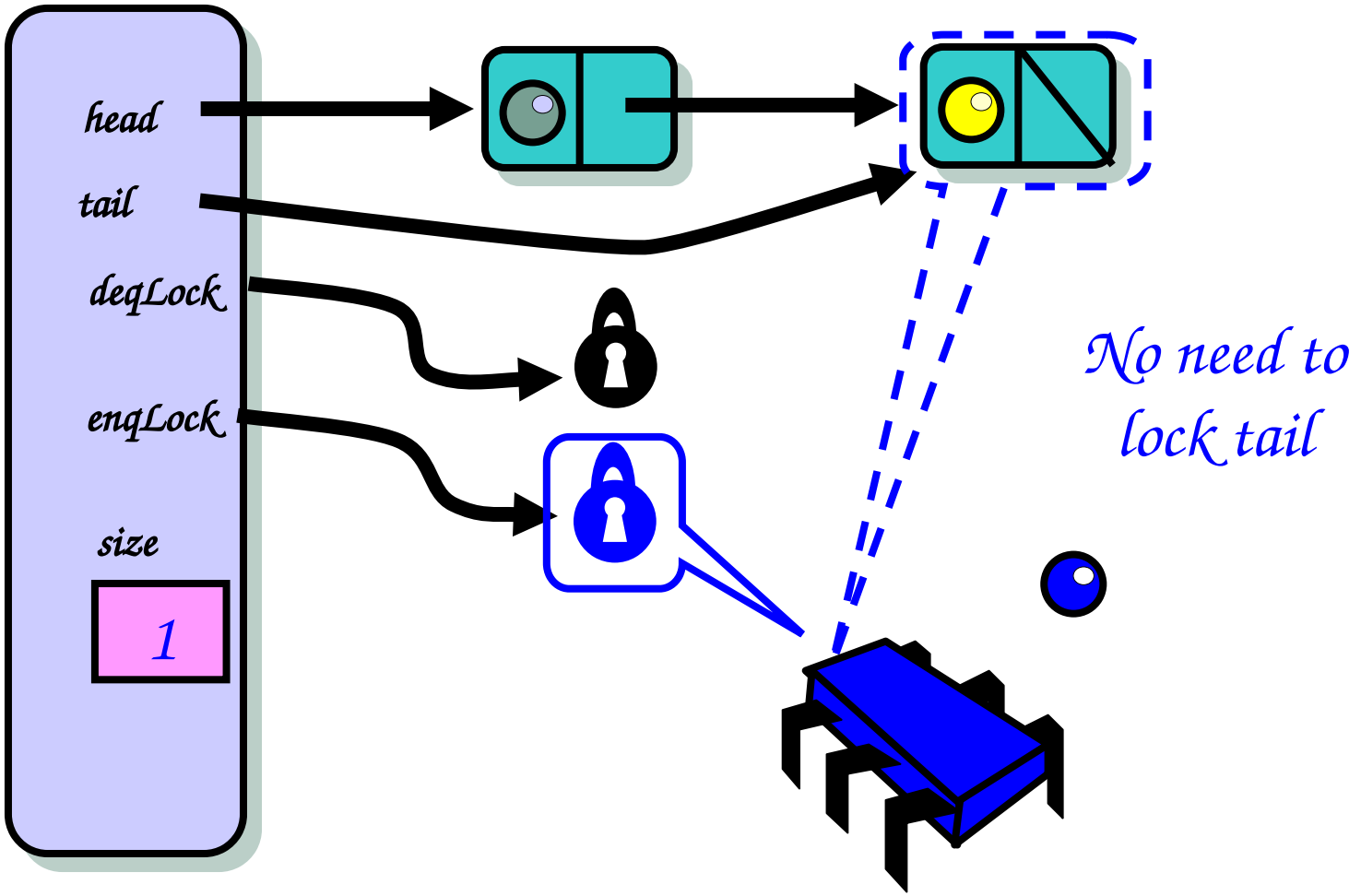
Enqueuer



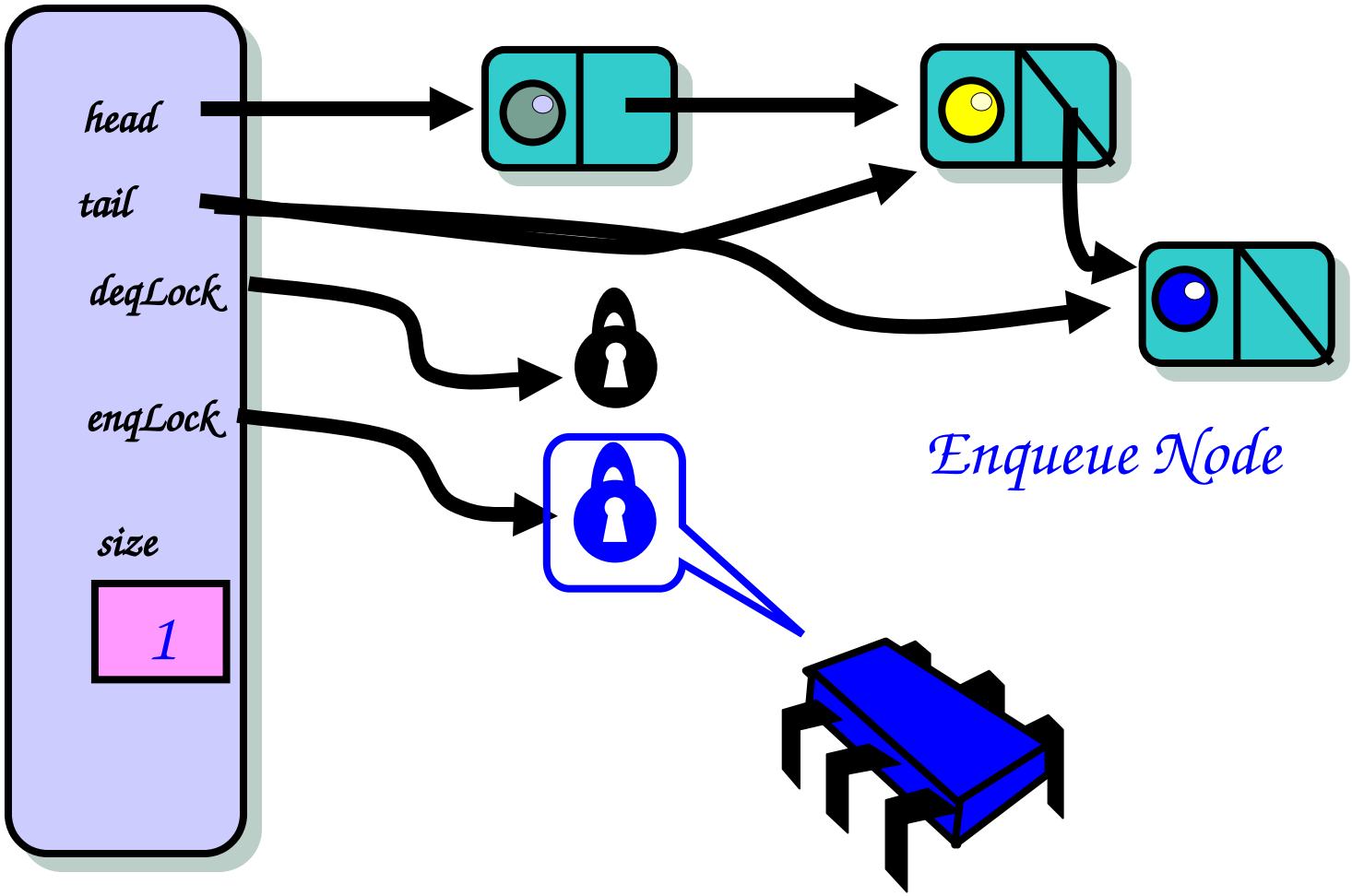
Enqueuer



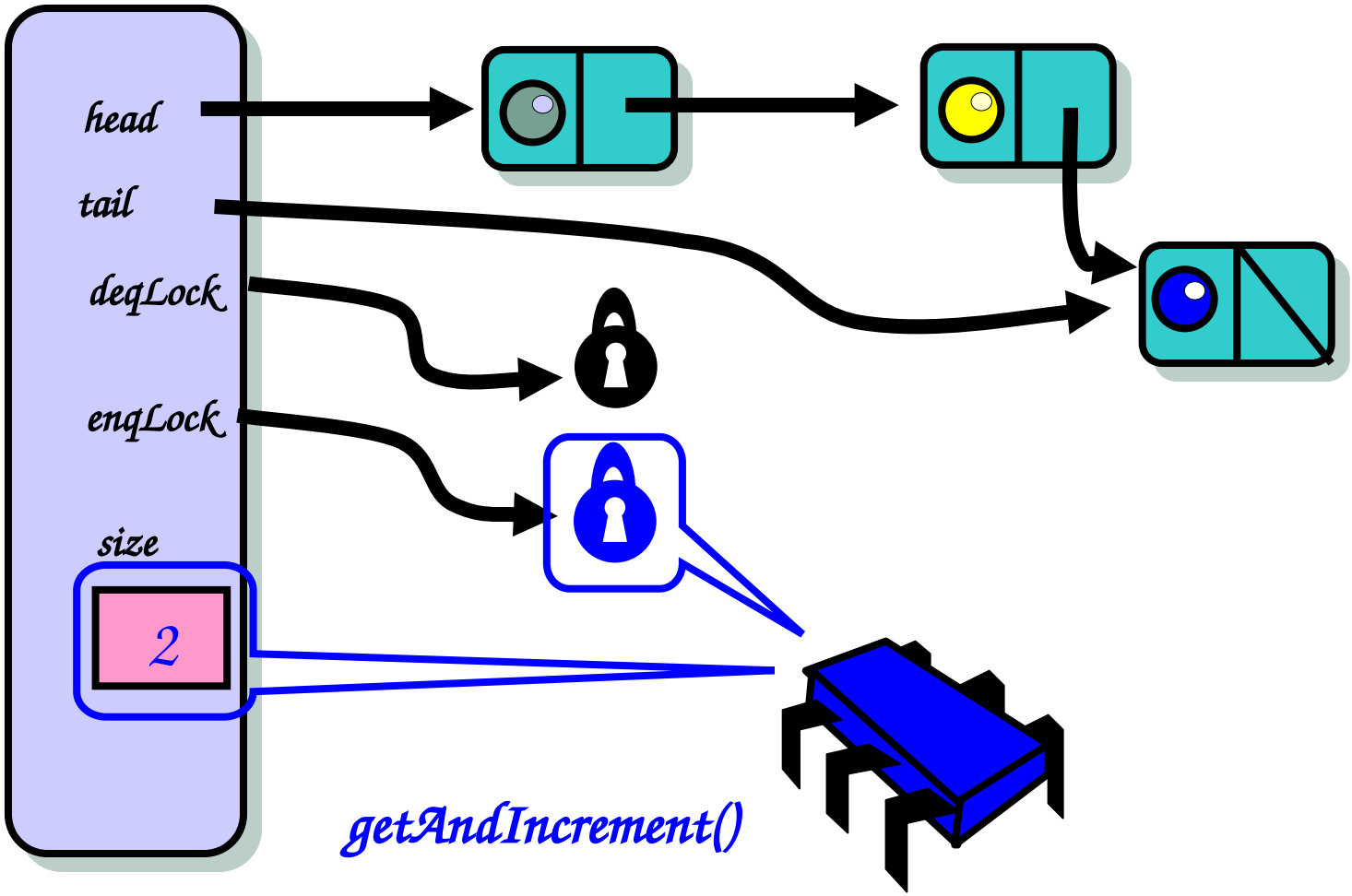
Enqueuer



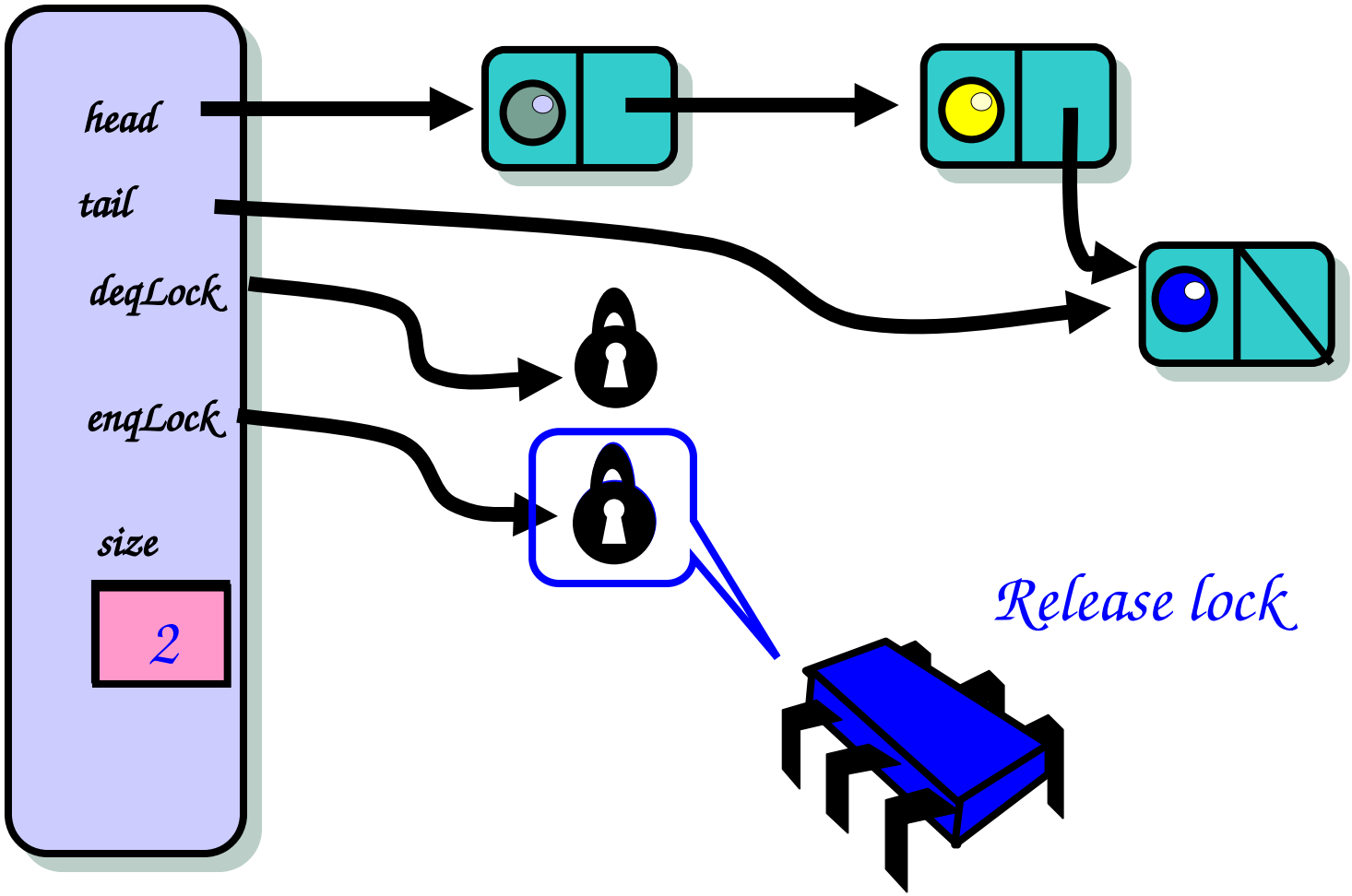
Enqueuer



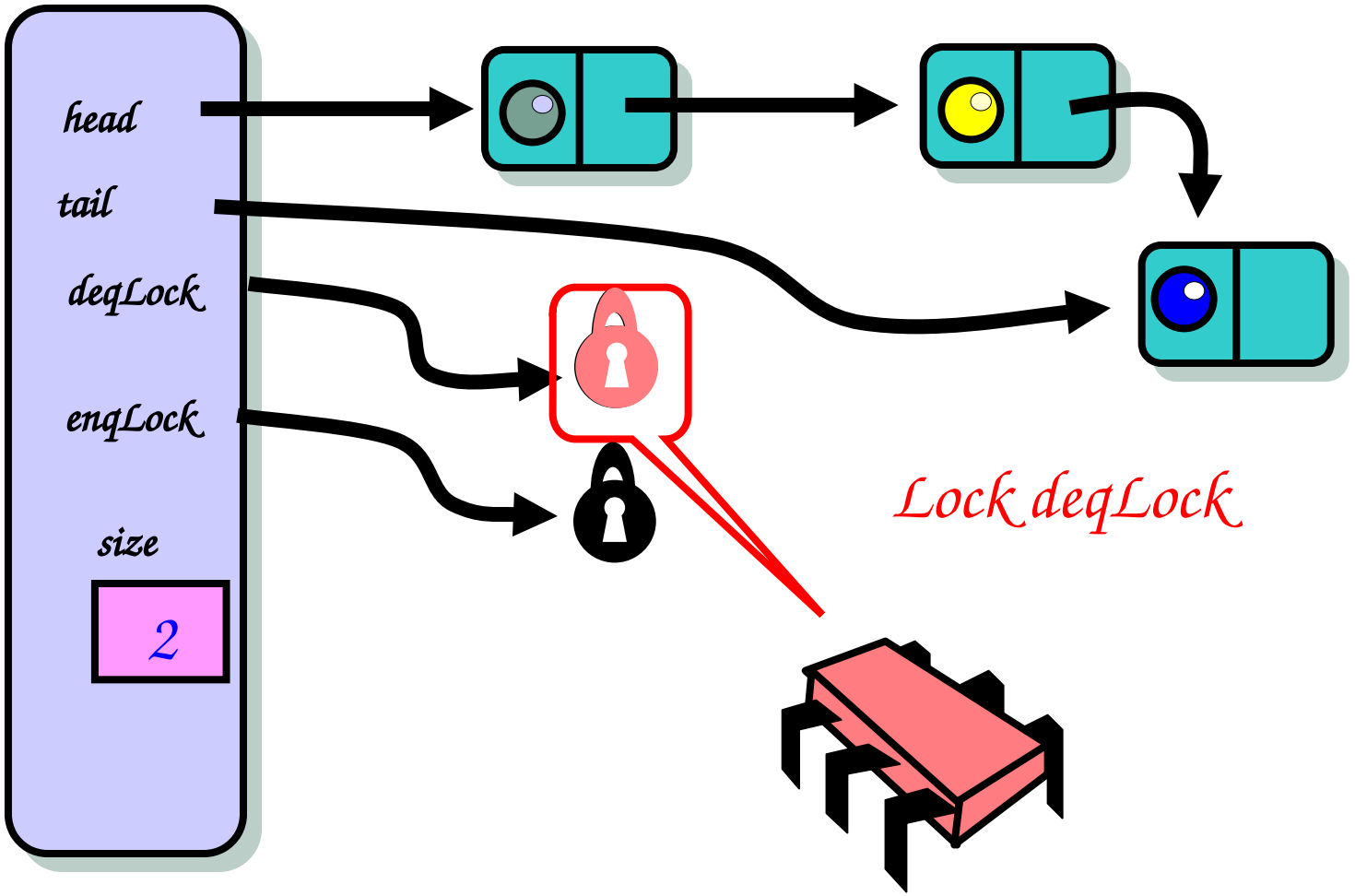
Enqueuer



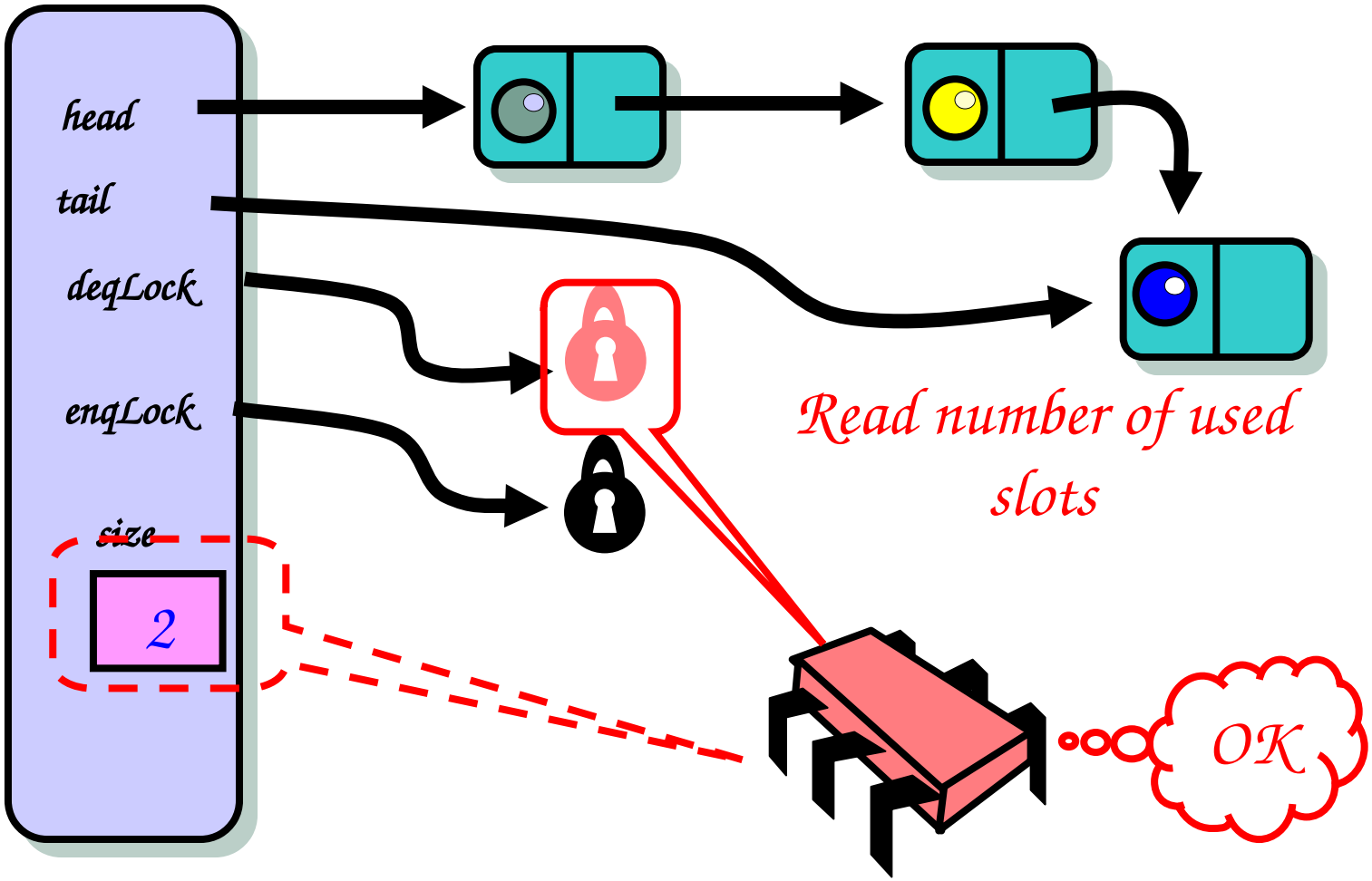
Enqueuer



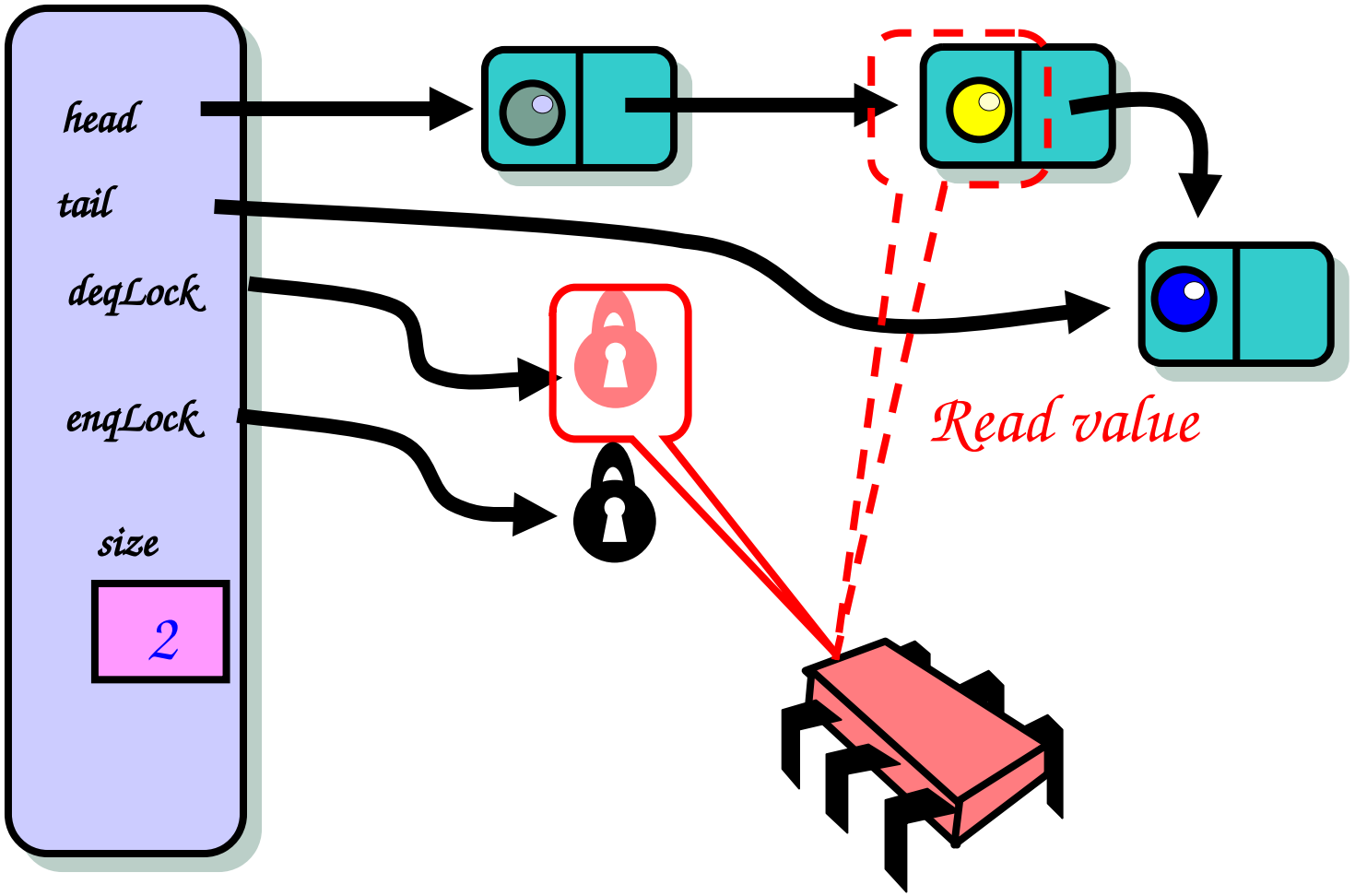
Dequeuer



Dequeuer

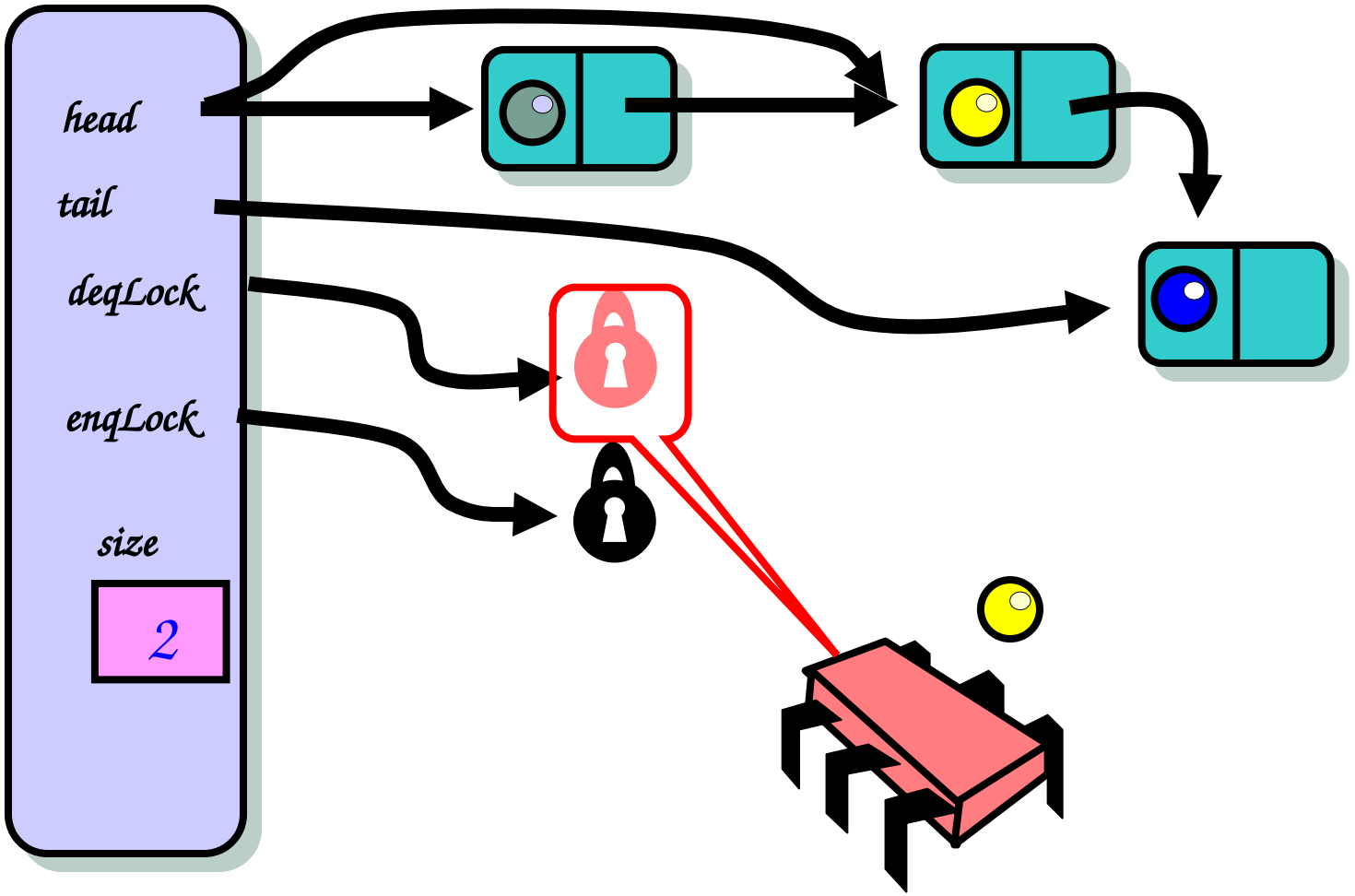

head
tail
deqLock
enqLock
size
2
Read number of used slots
OK

Dequeuer

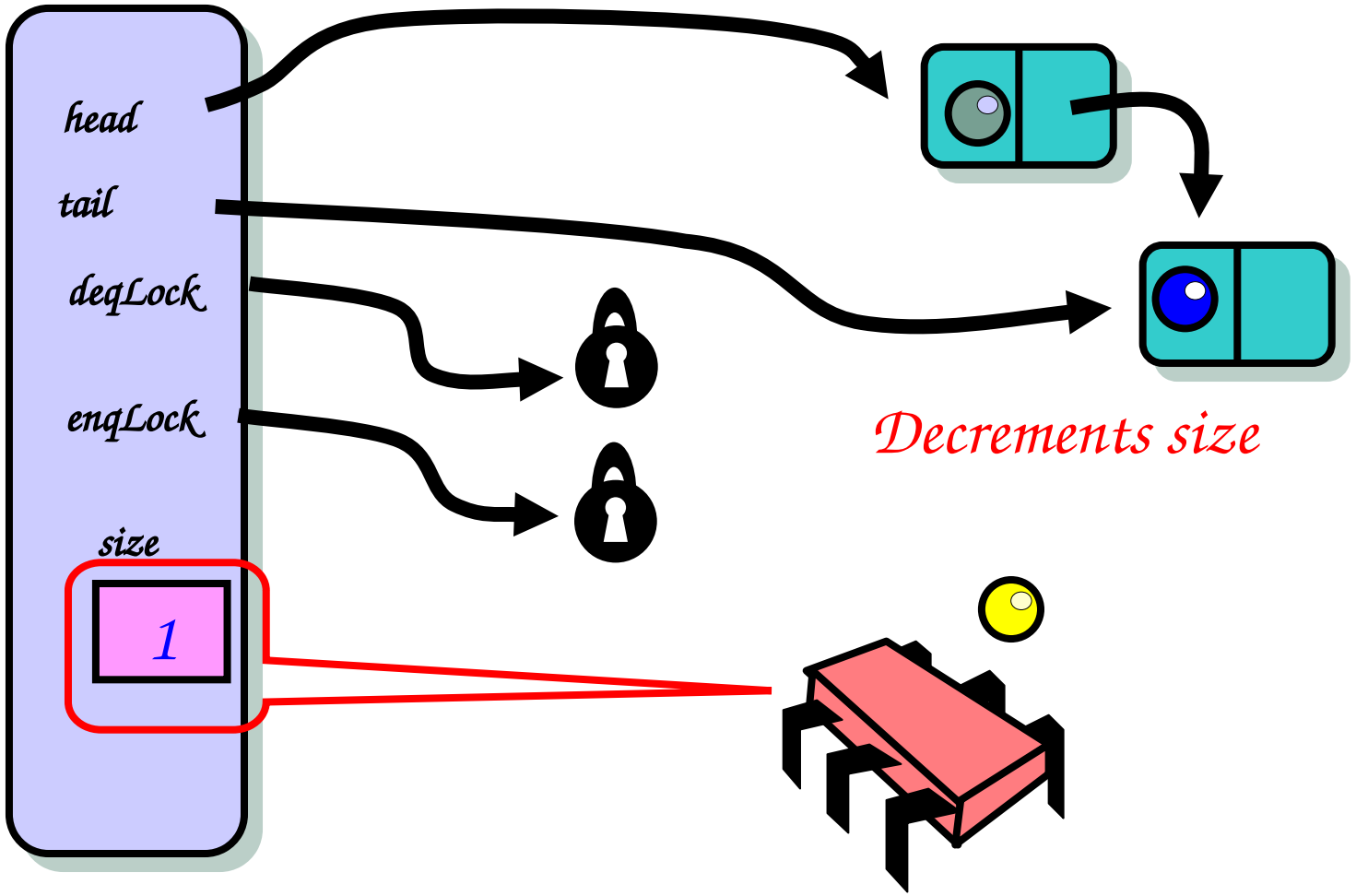


Dequeuer

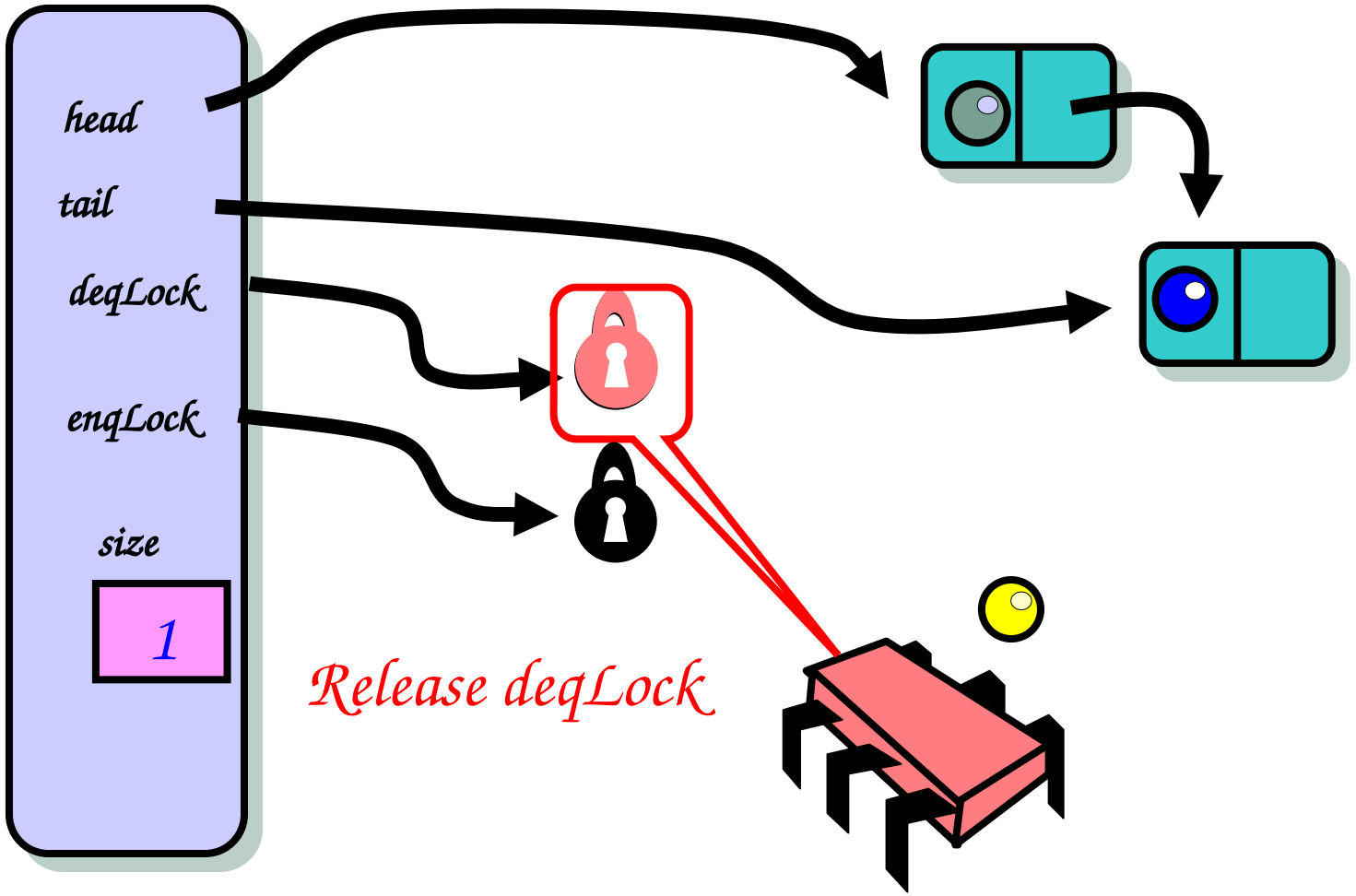
Make first Node new sentinel



Dequeuer



Dequeuer



Bounded Queue

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

Enq & deq locks

Bounded Queue

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

Conditions

Bounded Queue

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

Associate conditions with locks

Enq Method Part One

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

Enq Method Part One

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

If queue is full, patiently await further instructions …

Enq Method Part One

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

Add new node

Enq Method Part One

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

If queue was empty, allows dequeuers to be woken

Enq Method Part Deux

```
public void enq(T x) {
 …
    if (mustWakeDequeuers)
notEmptyCondition.signalAll();
    }
  }
```

Enq Method Part Deux

```
public void enq(T x) {
 …
    if (mustWakeDequeuers) {
       notEmptyCondition.signalAll();
    }
   }
```

Are there dequeuers to be signaled?

Enq Method Part Deux

```
public void enq(T x) {
 …
    if (mustWakeDequeuers) {
     notEmptyCondition.signalAll();
    }
  }
```

Signal dequeuers that queue no longer empty

Deq Method Part One

```
public T deq() {
  T result;
  boolean mustWakeEnqueuers = false;
  deqLock.lock();
  try {
    while (size.get() == 0)
      notEmptyCondition.await();
    result = head.next.value;
    head = head.next;
    if (size.getAndDecrement() == capacity)
     mustWakeEnqueuers = true;
  } finally {
    deqLock.unlock();
  }
 …
}
```

Deq Method Part Deux

```
public T deq() {
 …
    if (mustWakeEnqueuers) {
      notFullCondition.signalAll();
    }
  return result;
  }
```

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 into two counters

- enqSideSize
  - Decremented by deq()
- deqSideSize
  - Incremented by enq()

Two counters

- The enq() method:
  - Checks enqSideSize and proceeds as long as less than capacity
  - Cares only if value is capacity
- The deq() method:
  - Checks deqSideSize and proceeds as long as greater than zero
  - Cares only if value is zero

Unbounded Queues

- Queue can hold unbounded number of items:
  - enq() always enqueues its item
  - deq() throws an EmptyException if there is no item to dequeue
- Same representation as bounded queue, only no need to count number of items

Unbounded queue enq()

```
public void enq(T x) {
  enqLock.lock();
  try {
    Node e = new Node(x);
    tail.next = e;
    tail = tail.next;
  } finally {
    enqLock.unlock();
  }
}
```

Unbounded queue deq()

```
public T deq() throws EmptyException {
  T result;
  deqLock.lock();
  try {
    if (head.next == null)
      throw new EmptyException();
    result = head.next.value;
    head = head.next;
  } finally {
    deqLock.unlock();
  }
  return result;
}
```

Only need to check that queue is not empty

Now an unbounded lock-free queue

- Natural extension of the unbounded queue
- However it prevents method calls from starving by having quicker threads help slower threads

Unbounded Lock-free Queue

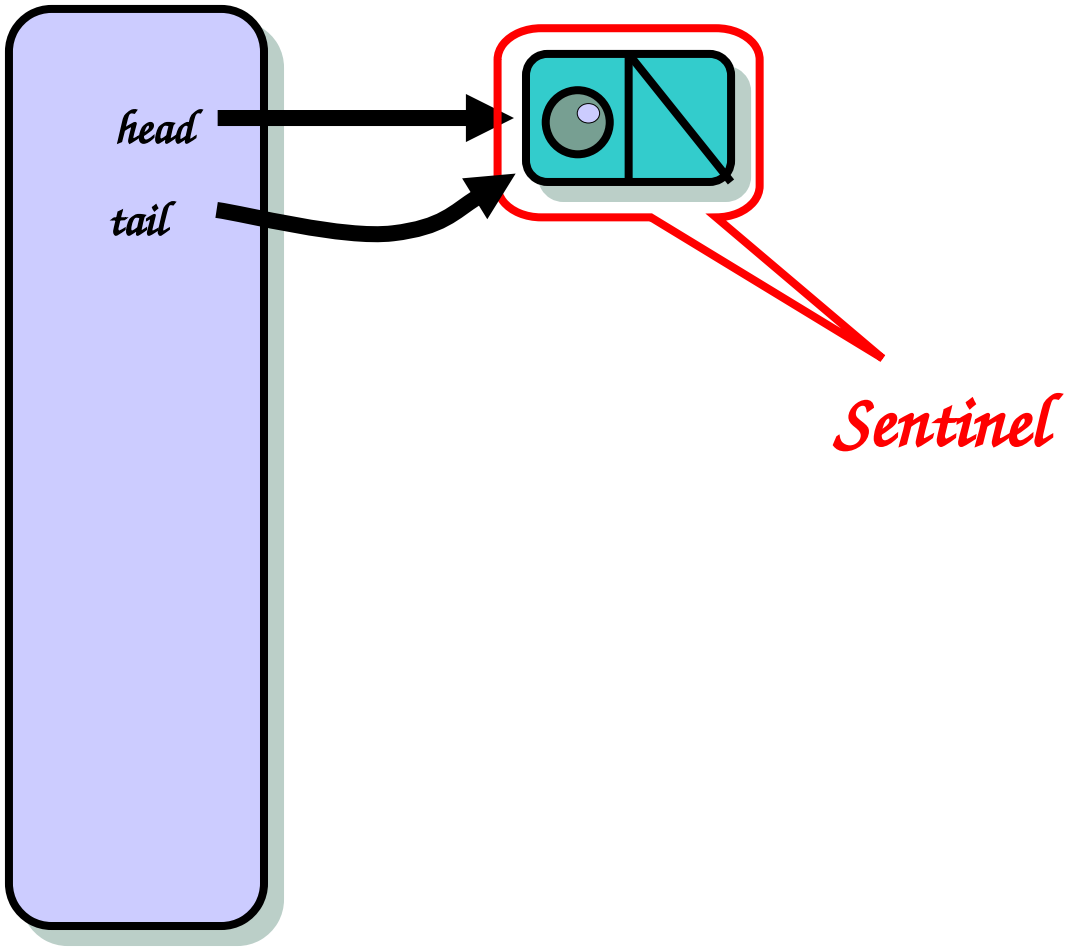
- Again represented as a Linked List
- However each node’s next field is now an AtomicReference (to facilitate lock-free operations)

Lock-free queue Node class

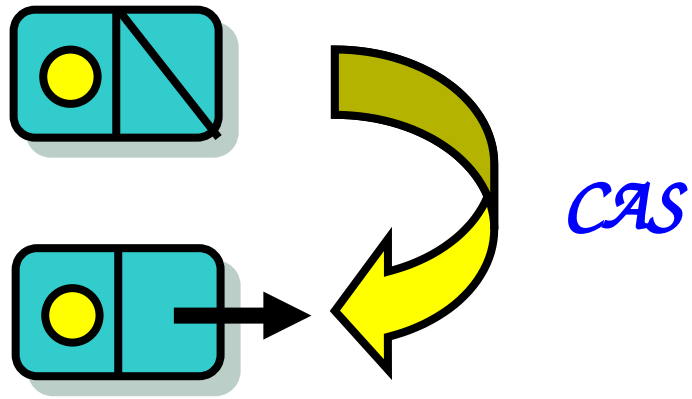
```
public class Node {
  public T value;
  public AtomicReference<Node> next;

  public Node(T value) {
    value = value;
    next = new AtomicReference<Node>(null);
  }
}
```

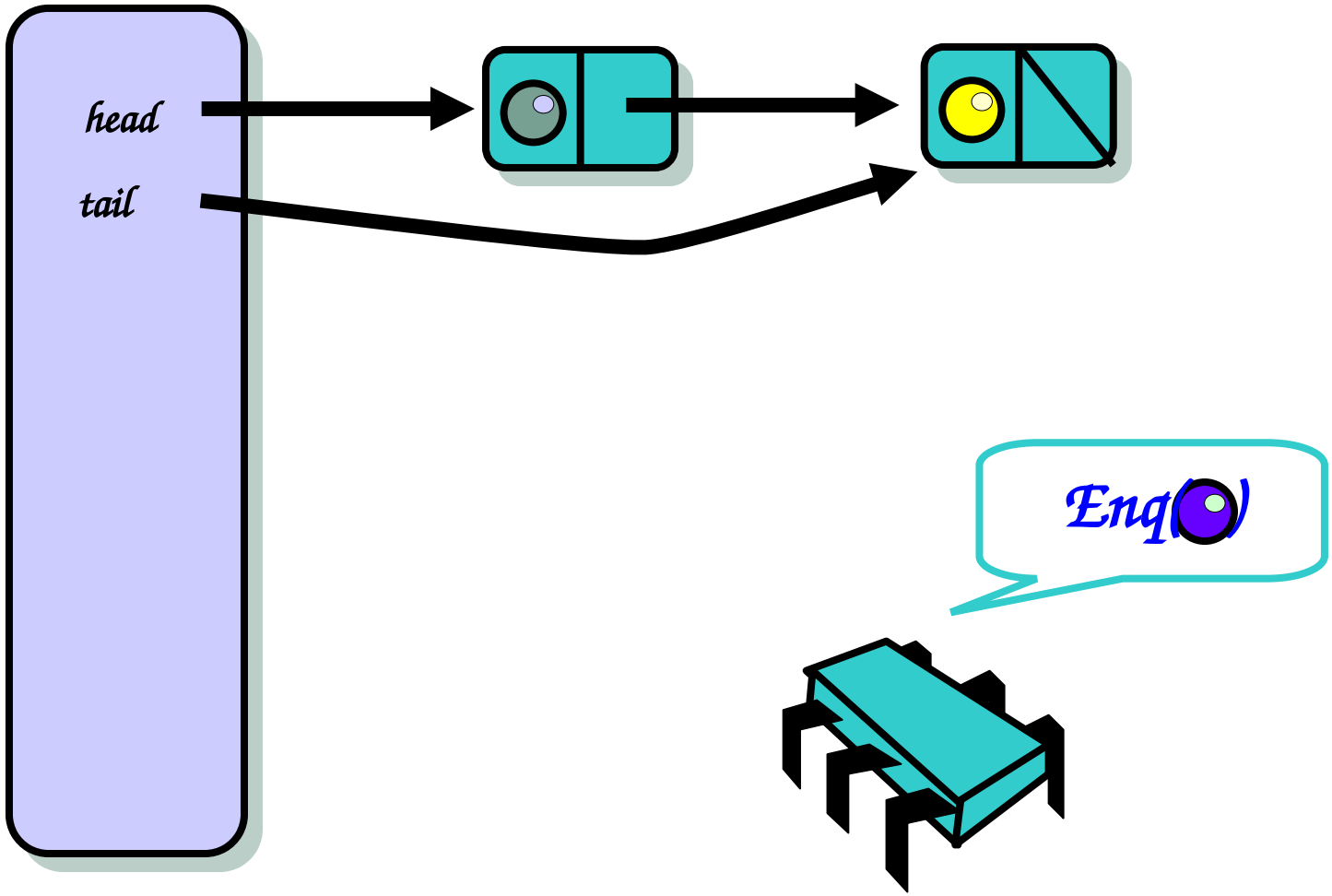
A Lock-Free Queue



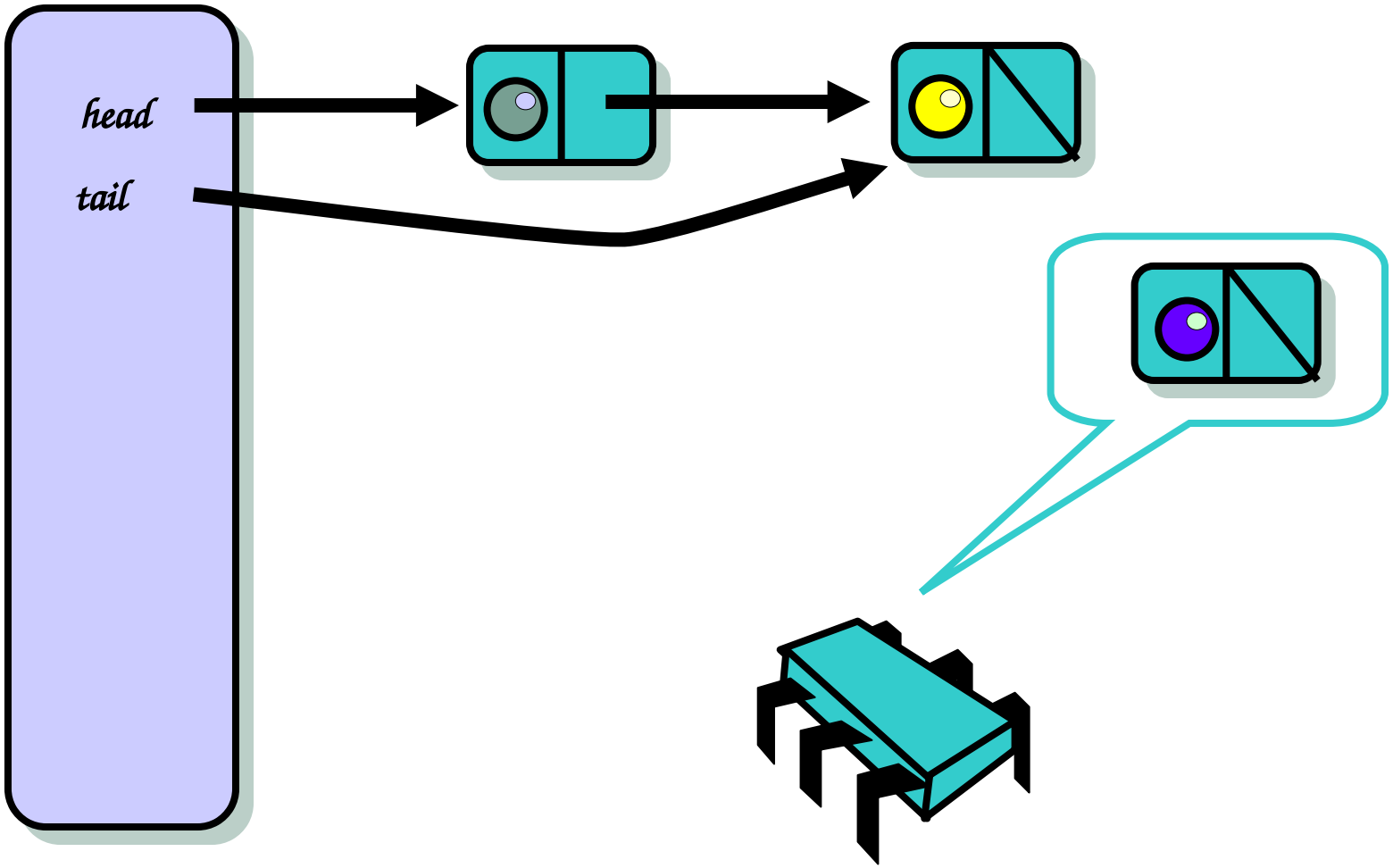
Compare and Set



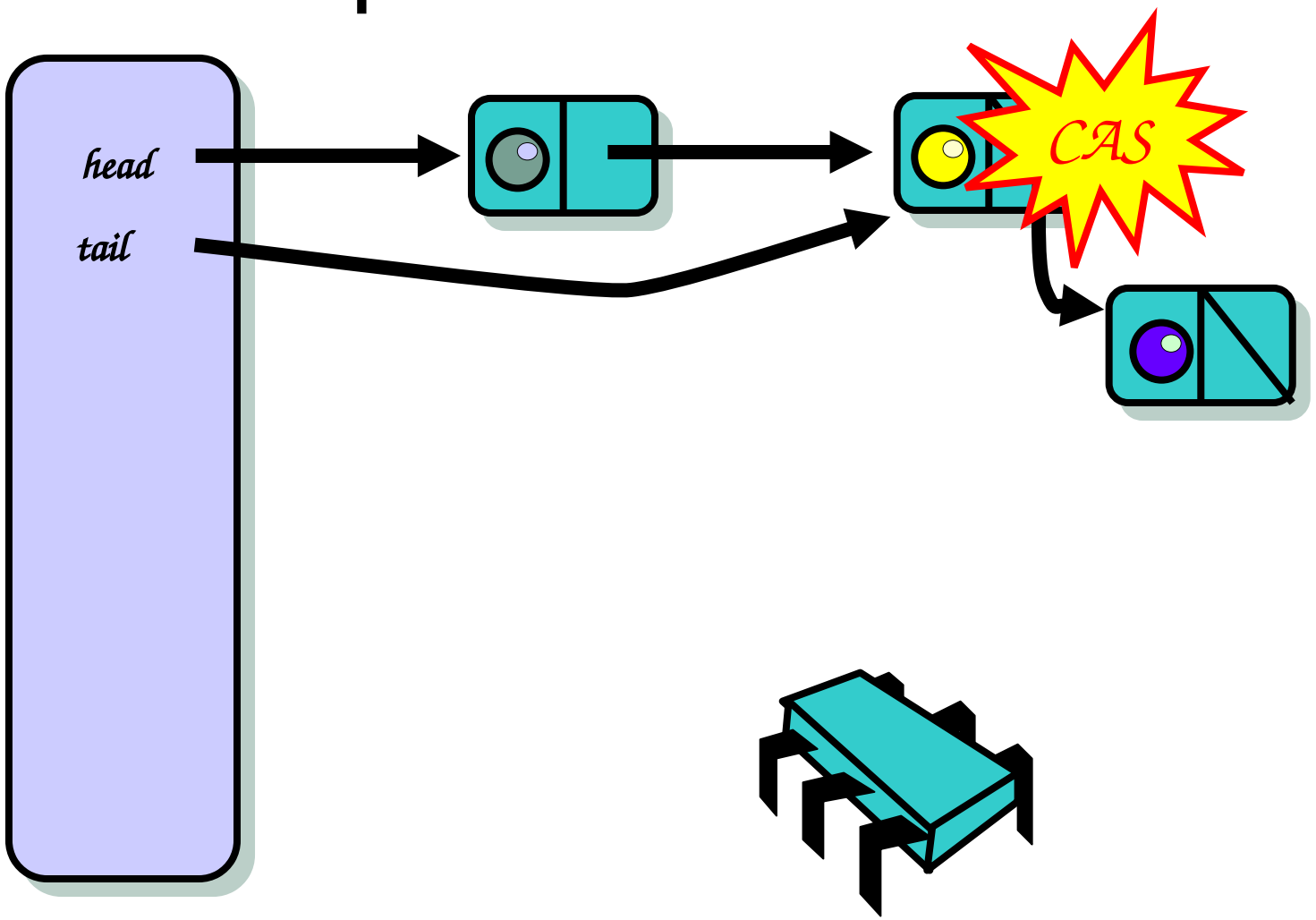
Enqueue



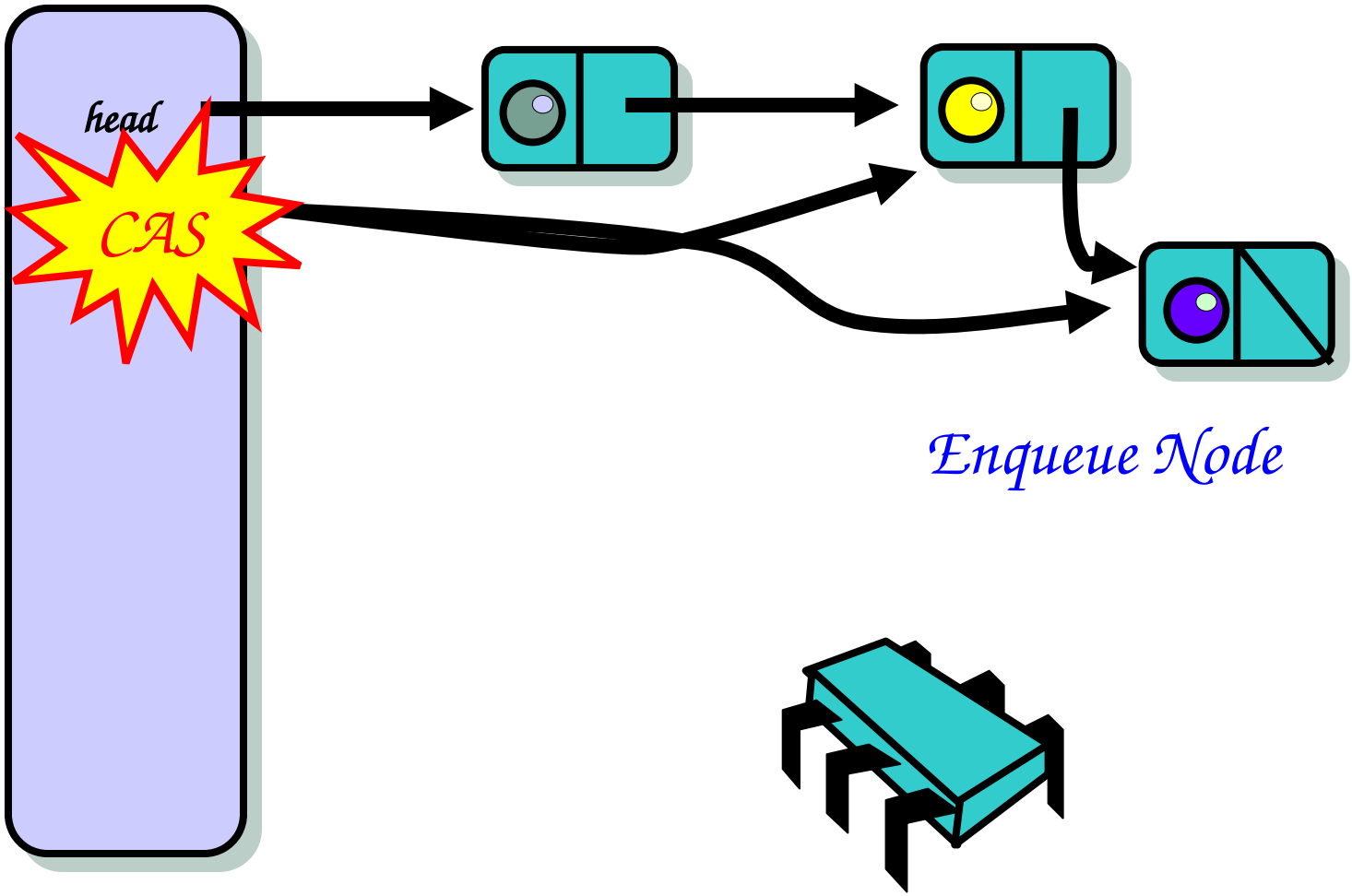
Enqueue



Logical Enqueue



Physical Enqueue



Enqueue

- The enq() method is lazy
- Enqueue consists of two steps:
  - Logical enqueue – appends the node to the linked list
  - Physical enqueue – changes the tail field to point to the new node
- These two steps are not atomic

Enqueue

- It is thus possible that a new item is reachable by other threads before it actually becomes the tail
- The tail field can thus refer to either
  - The actual last node or
  - The penultimate node

Enqueue

- When enqueueing you have to make sure that you are adding an item to the actual end of the queue
- So check whether the tail you are working with has a successor

Enqueue

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

Lock-free enqueue method

```
public void enq(T value) {
  Node node = new Node(value);
  while (true) {
    Node last = tail.get();
    Node next = tail.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        } else
          tail.compareAndSet(last, last.next);
      }
    }
  }
}
```

Lock-free enqueue method

```
public void enq(T value) {
  Node node = new Node(value);
  while (true) {
    Node last = tail.get();
    Node next = tail.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        } else
          tail.compareAndSet(last, last.next);
      }
    }
  }
}
```

What does tail point to

Lock-free enqueue method

```
public void enq(T value) {
  Node node = new Node(value);
  while (true) {
    Node last = tail.get();
    Node next = tail.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        } else
          tail.compareAndSet(last, last.next);
      }
    }
  }
}
```

What does tail's next point to

Lock-free enqueue method

```
public void enq(T value) {
  Node node = new Node(value);
  while (true) {
    Node last = tail.get();
    Node next = tail.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        } else
          tail.compareAndSet(last, last.next);
      }
    }
  }
}
```

If the value pointed to by tail has not changed…

Lock-free enqueue method

```
public void enq(T value) {
  Node node = new Node(value);
  while (true) {
    Node last = tail.get();
    Node next = tail.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        } else
          tail.compareAndSet(last, last.next);
      }
    }
  }
}
```

…and tail's next points to null…

Lock-free enqueue method

```
public void enq(T value) {
  Node node = new Node(value);
  while (true) {
    Node last = tail.get();
    Node next = tail.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        } else
          tail.compareAndSet(last, last.next);
      }
    }
  }
}
```

Logical enqueue

Lock-free enqueue method

```
public void enq(T value) {
  Node node = new Node(value);
  while (true) {
    Node last = tail.get();
    Node next = tail.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        } else
          tail.compareAndSet(last, last.next);
      }
    }
  }
}
```

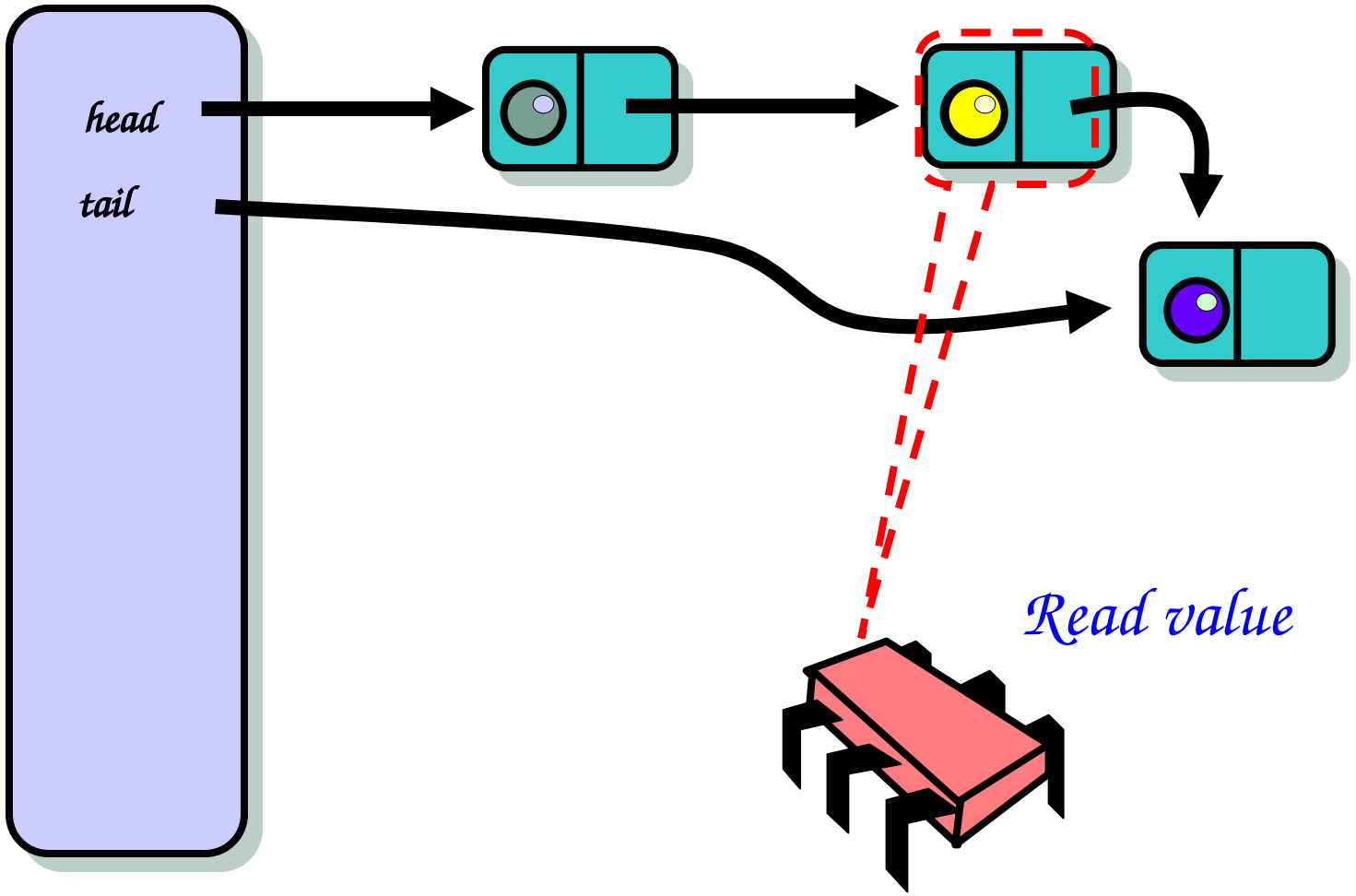
Physical enqueue

Lock-free enqueue method

```
public void enq(T value) {
  Node node = new Node(value);
  while (true) {
    Node last = tail.get();
    Node next = tail.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        } else
          tail.compareAndSet(last, last.next);
      }
    }
  }
}
```

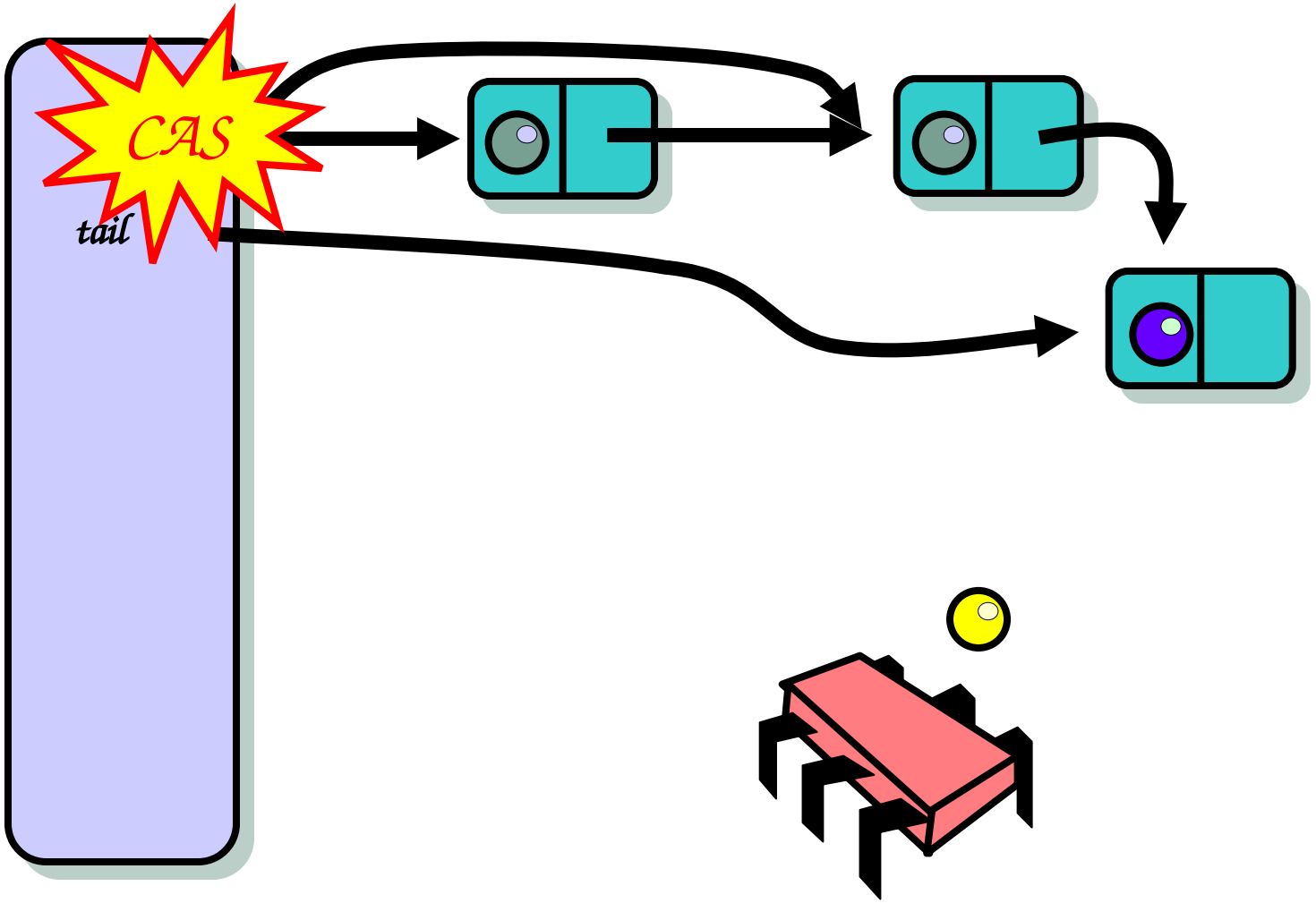
If tail has a successor it 'helps' other nodes by advancing tail to refer to the successor

Dequeuer



Dequeuer

Make first Node new sentinel



Dequeuer

- Although queue is unbounded, you always have to check that queue is not empty
- Without counter, queue is empty when head = tail = sentinel

Lock-free dequeue method

```
public T deq() throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) {
          throw new EmptyException();
        }
        tail.compareAndSet(last, next);
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next))
          return value;
      }
}}}
```

Lock-free dequeue method

```
public T deq() throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) {
          throw new EmptyException();
        }
        tail.compareAndSet(last, next);
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next))
          return value;
      }
}}}
```

What does head point to

Lock-free dequeue method

```
public T deq() throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) {
          throw new EmptyException();
        }
        tail.compareAndSet(last, next);
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next))
          return value;
      }
}}}
```

What does tail point to

Lock-free dequeue method

```
public T deq() throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) {
          throw new EmptyException();
        }
        tail.compareAndSet(last, next);
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next))
          return value;
      }
}}}
```

Is there a node after head?

Lock-free dequeue method

```
public T deq() throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) {
          throw new EmptyException();
        }
        tail.compareAndSet(last, next);
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next))
          return value;
      }
}}}
```

Has the value of head changed?

Lock-free dequeue method

```
public T deq() throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) {
          throw new EmptyException();
        }
        tail.compareAndSet(last, next);
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next))
          return value;
      }
}}}
```

Is the queue empty?

Lock-free dequeue method

```
public T deq() throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) {
          throw new EmptyException();
        }
        tail.compareAndSet(last, next);
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next))
          return value;
      }
}}}
```

Is the queue still empty?

Lock-free dequeue method

```
public T deq() throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) {
          throw new EmptyException();
        }
        tail.compareAndSet(last, next);
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next))
          return value;
      }
}}}
```

If a new value node have arrived, 'help' others by advancing tail

Lock-free dequeue method

```
public T deq() throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) {
          throw new EmptyException();
        }
        tail.compareAndSet(last, next);
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next))
          return value;
      }
}}}
```

If the queue is not empty

Lock-free dequeue method

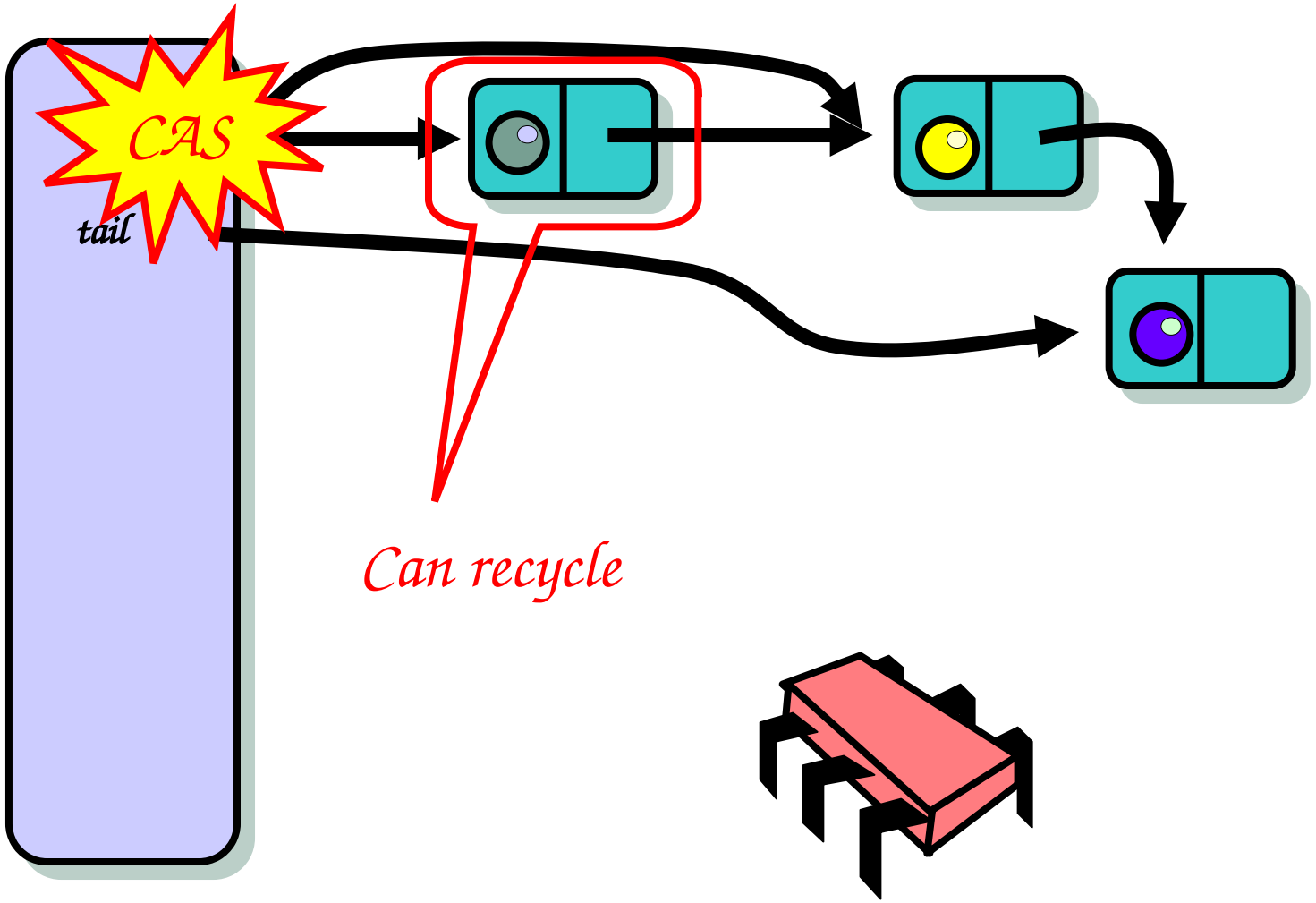
```
public T deq() throws EmptyException {
  while (true) {
    Node first = head.get();
    Node last = tail.get();
    Node next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) {
          throw new EmptyException();
        }
        tail.compareAndSet(last, next);
      } else {
        T value = next.value;
        if (head.compareAndSet(first, next))
          return value;
      }
}}}
```

Try to remove the first value from the queue

Memory reuse

- What do we do with nodes after we dequeue them?
- With current implementation nodes are unlinked from linked list, but we depend on built-in garbage collection to remove them from memory
- Suppose there is no garbage collector?

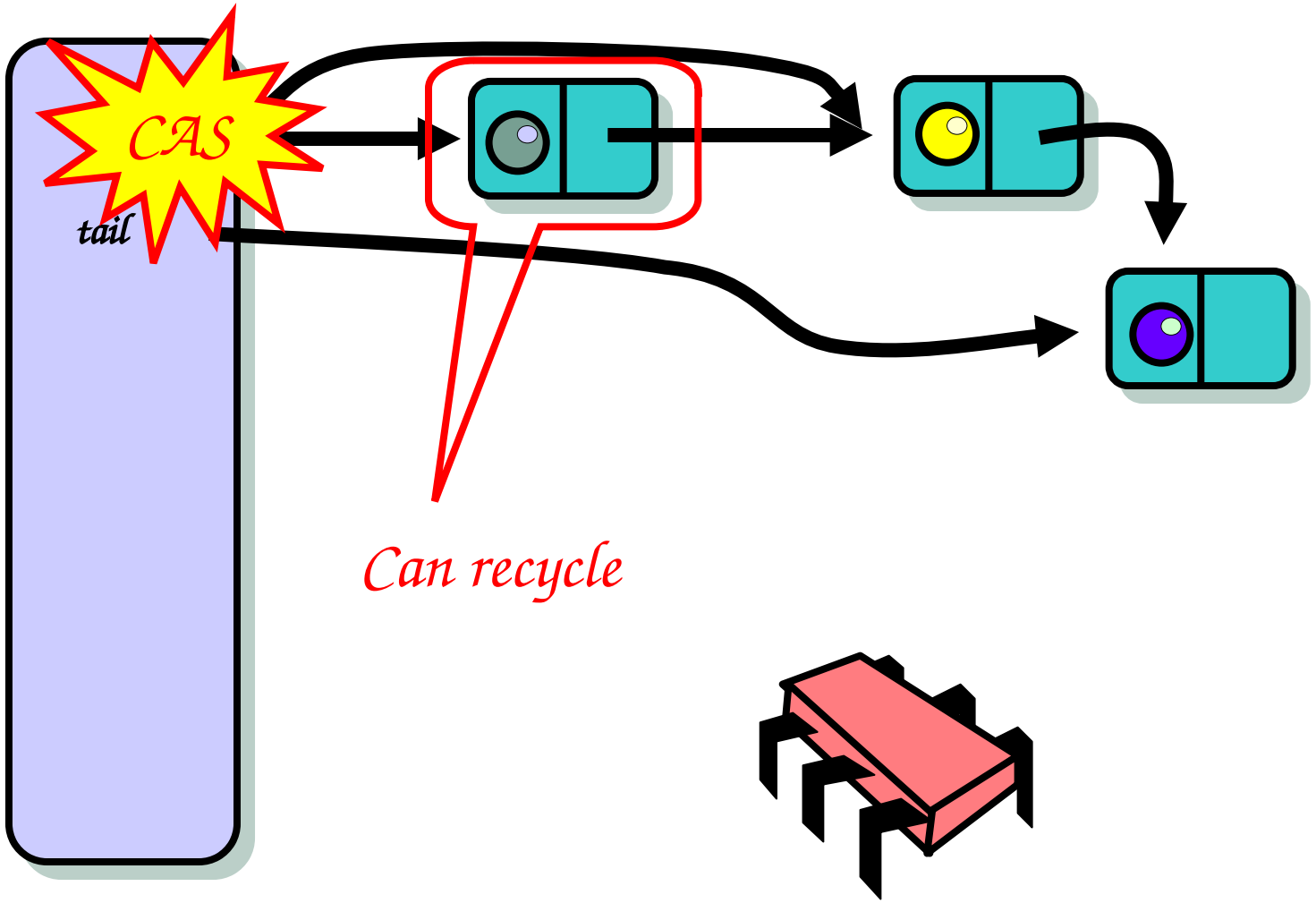
Dequeuer



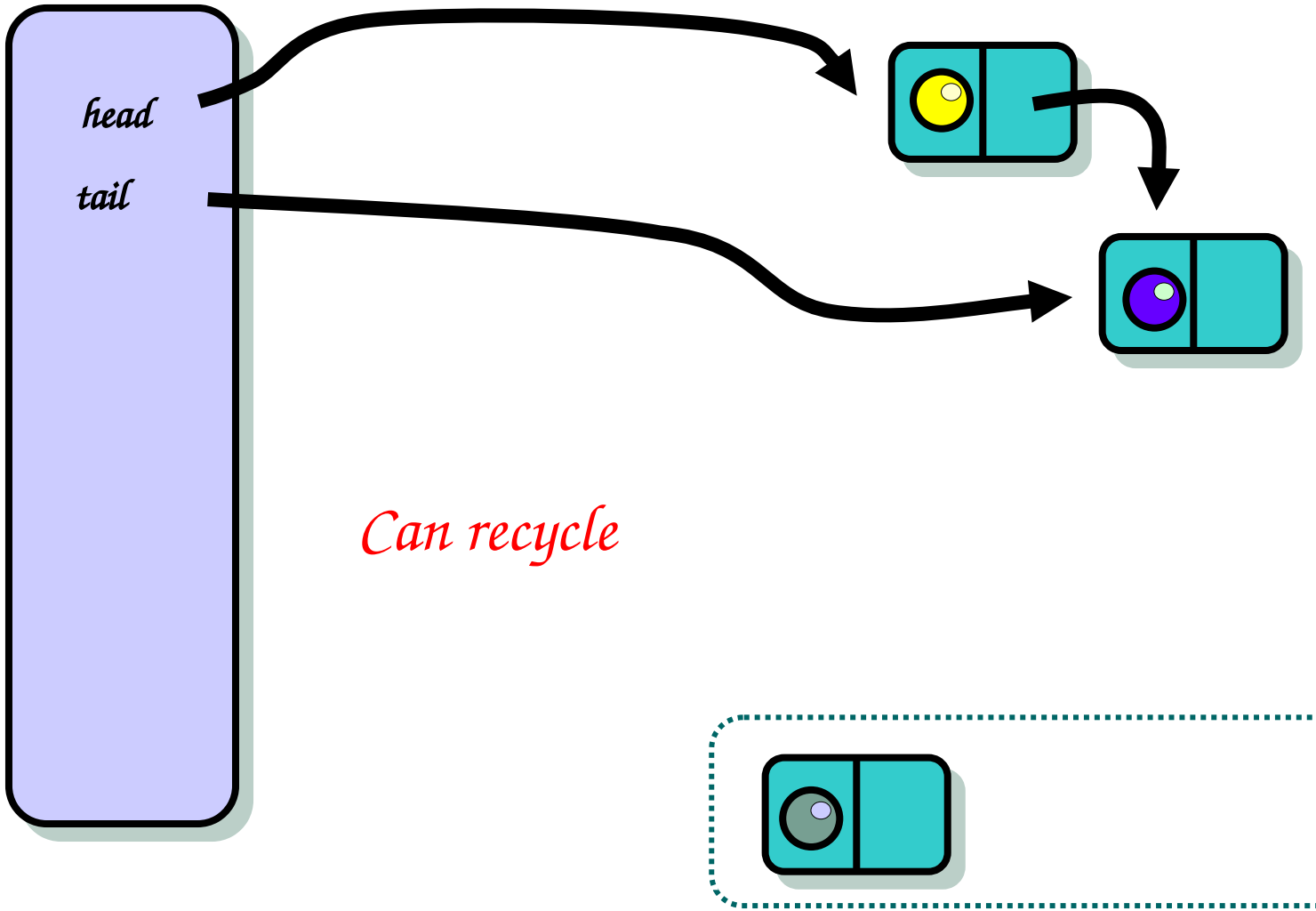
Simple solution

- Each thread maintains its own private free-list of unused queue entries
  - When enqueue – get a node from the free-list
  - When dequeue – add removed node to free-list
- If free-list is empty, simply allocate new node through dynamic memory allocation

Dequeuer



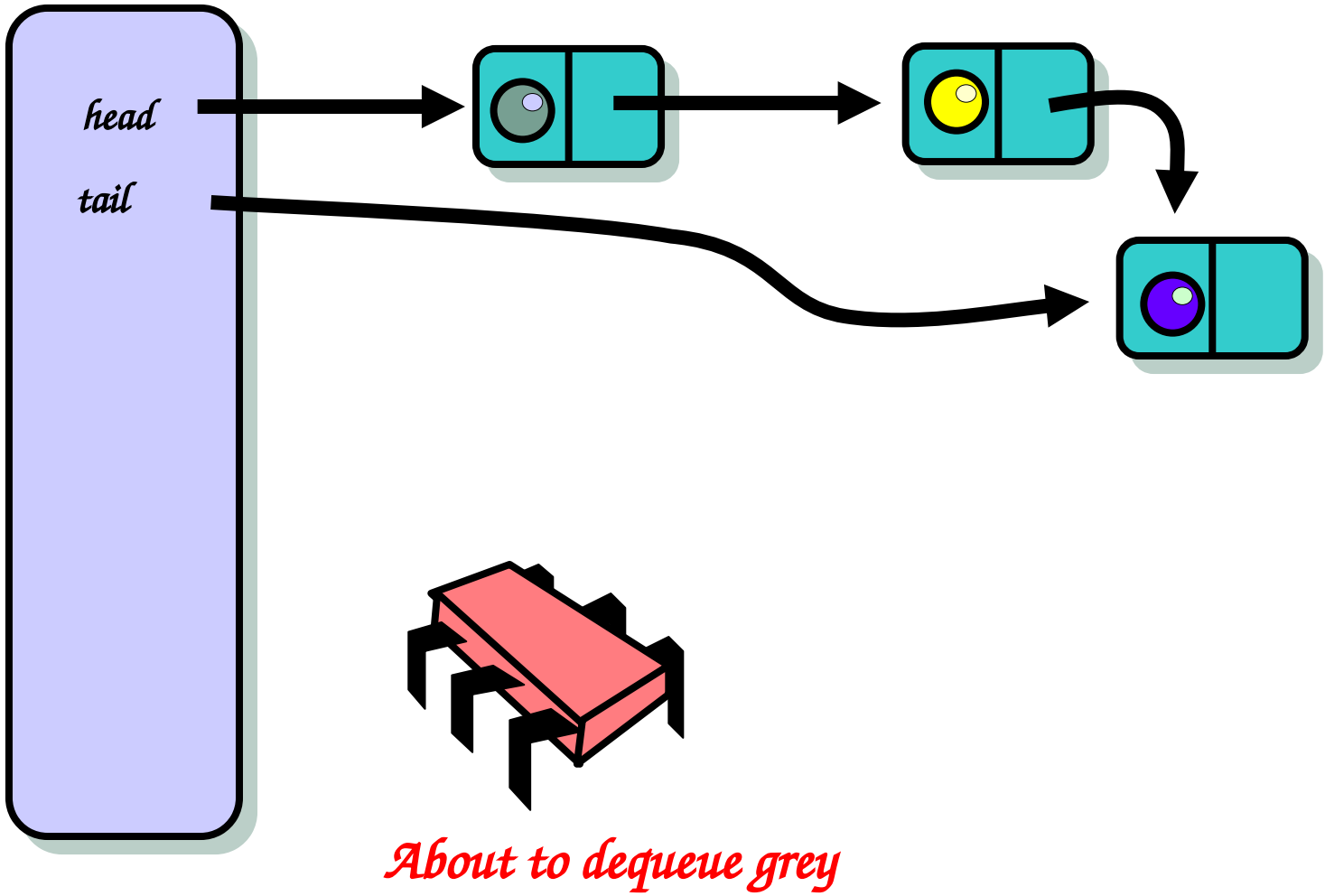
Dequeuer



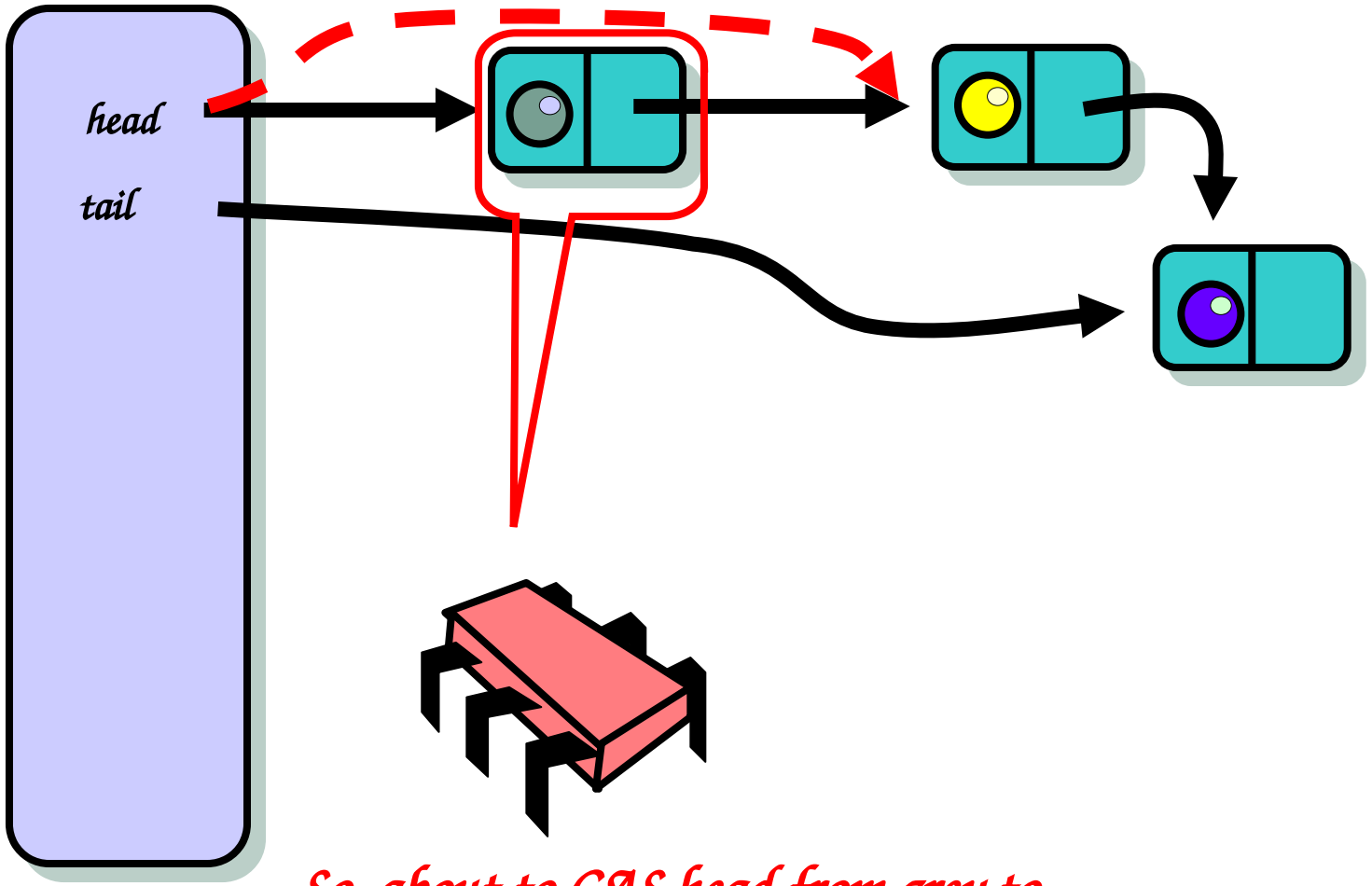
Potential problem

- When nodes are recycled – removed from the queue and later added again
  - Dreaded ABA problem

Dequeuer

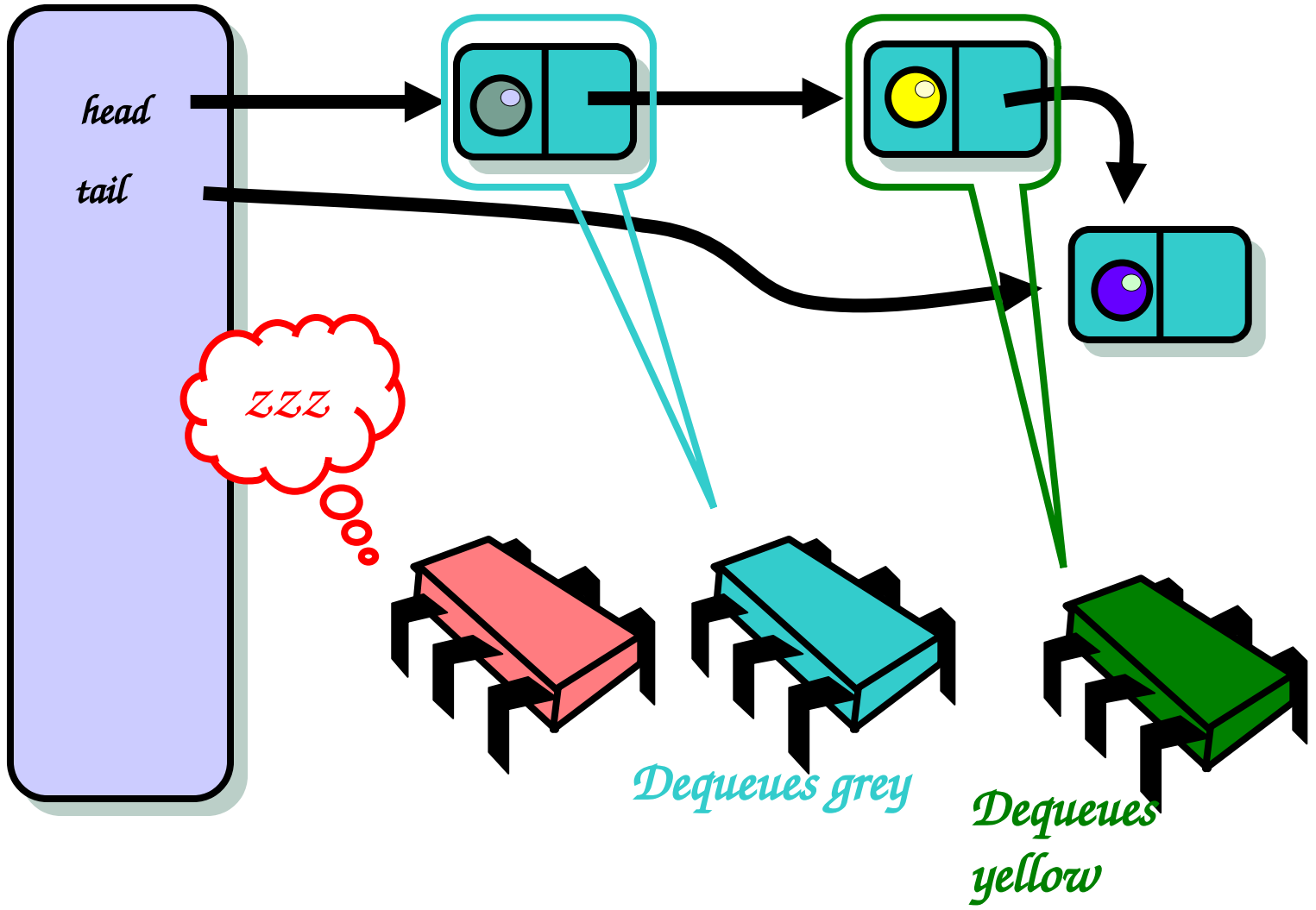


Dequeuer

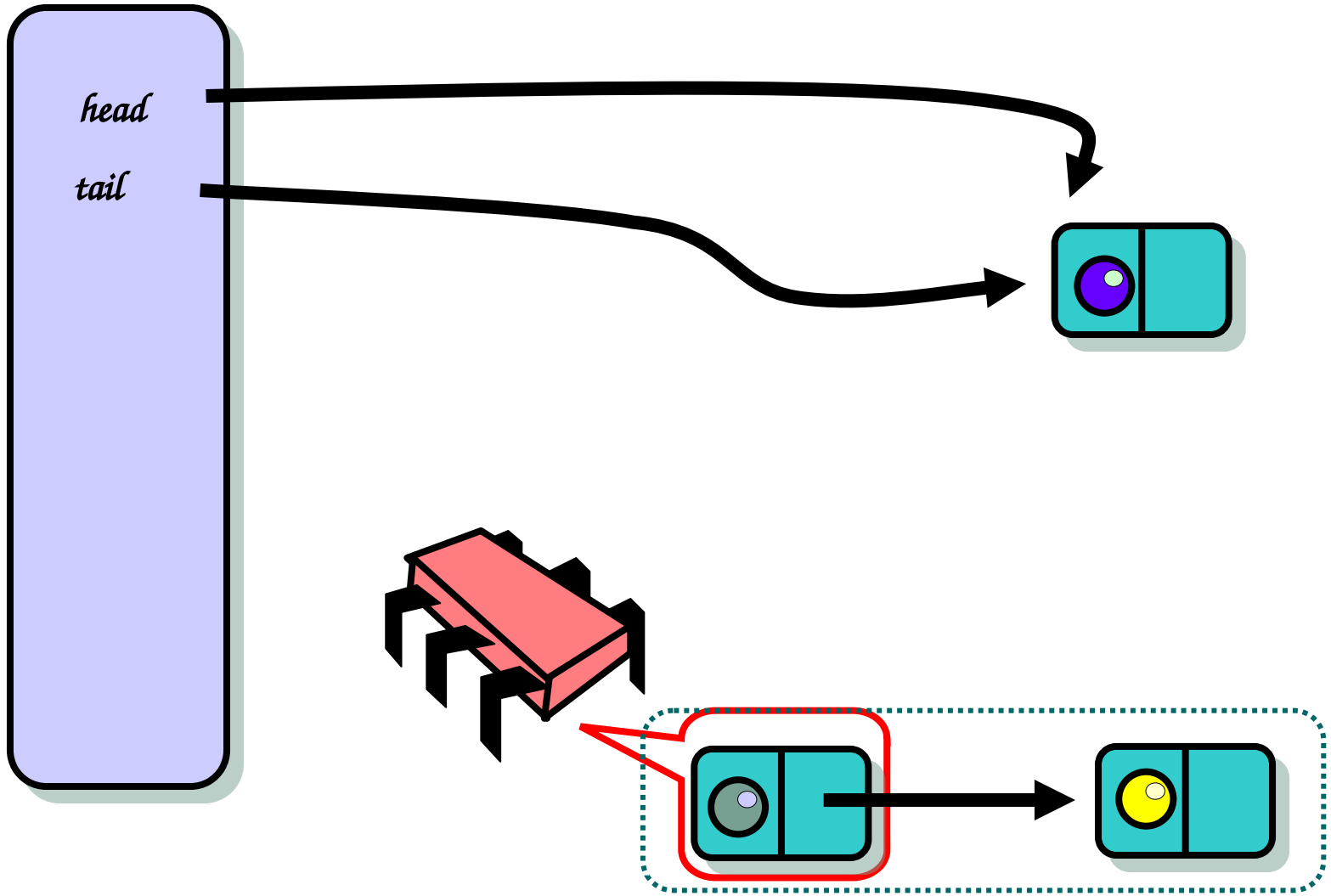


So, about to CAS head from grey to yellow

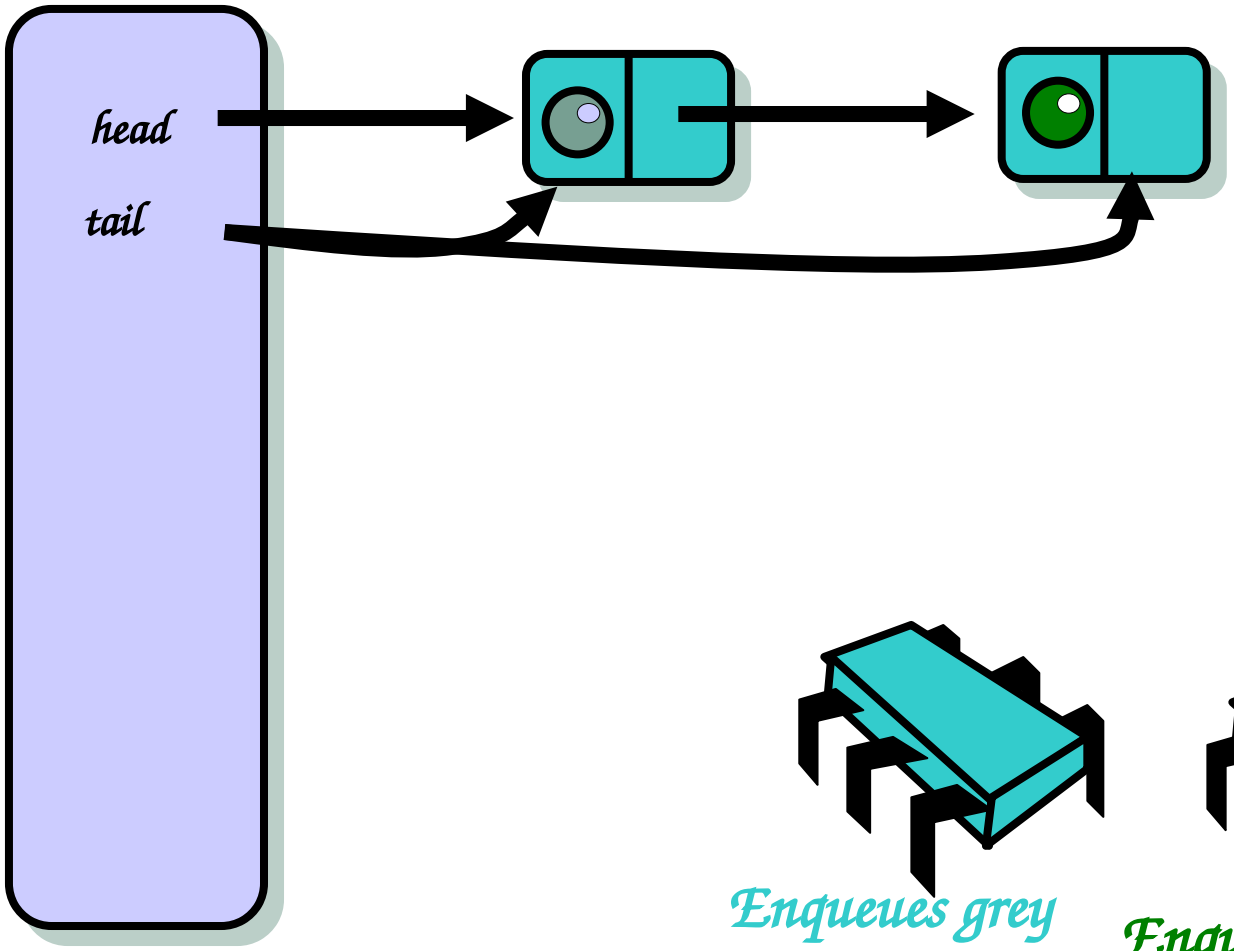
Dequeuer



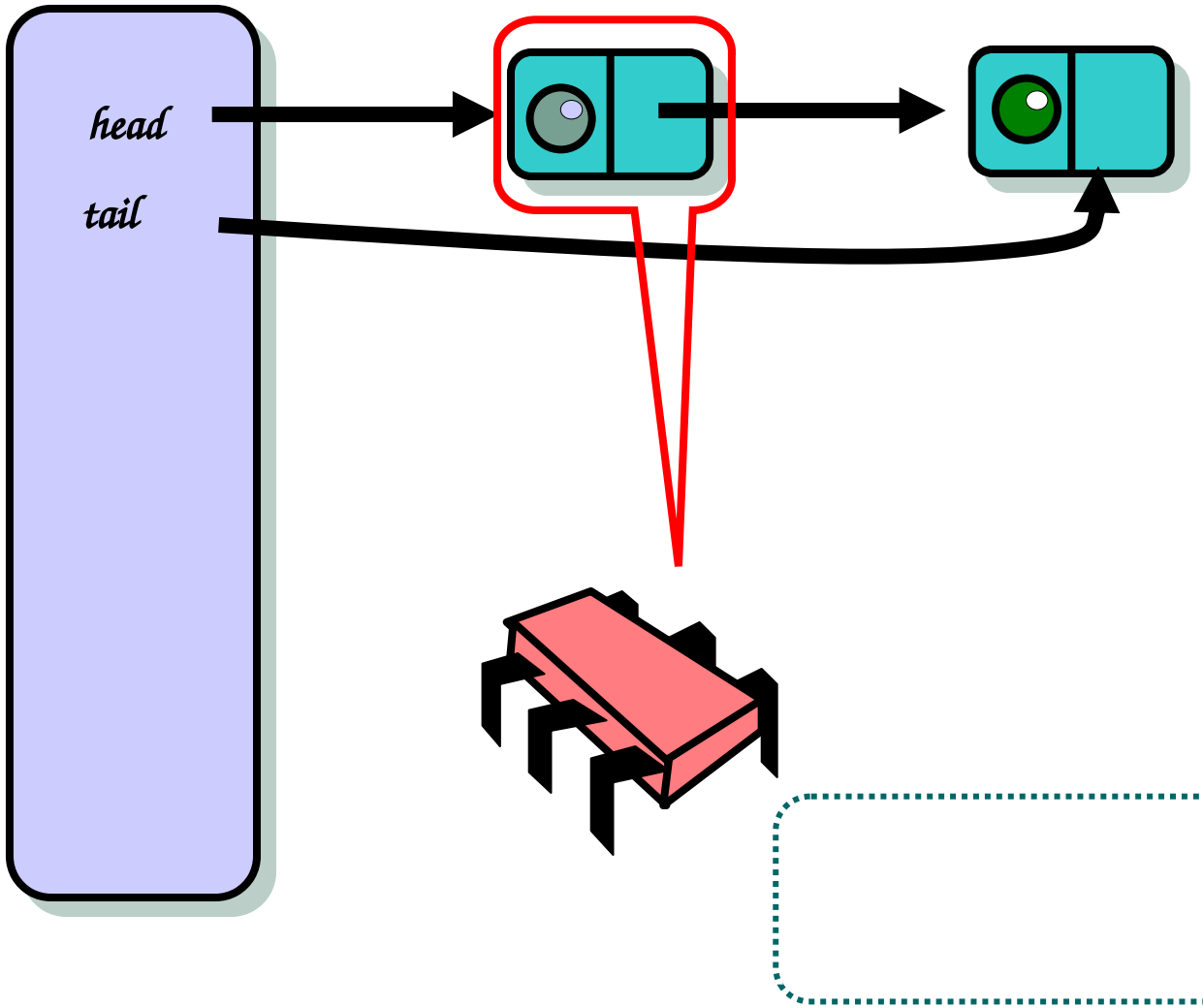
Dequeuer



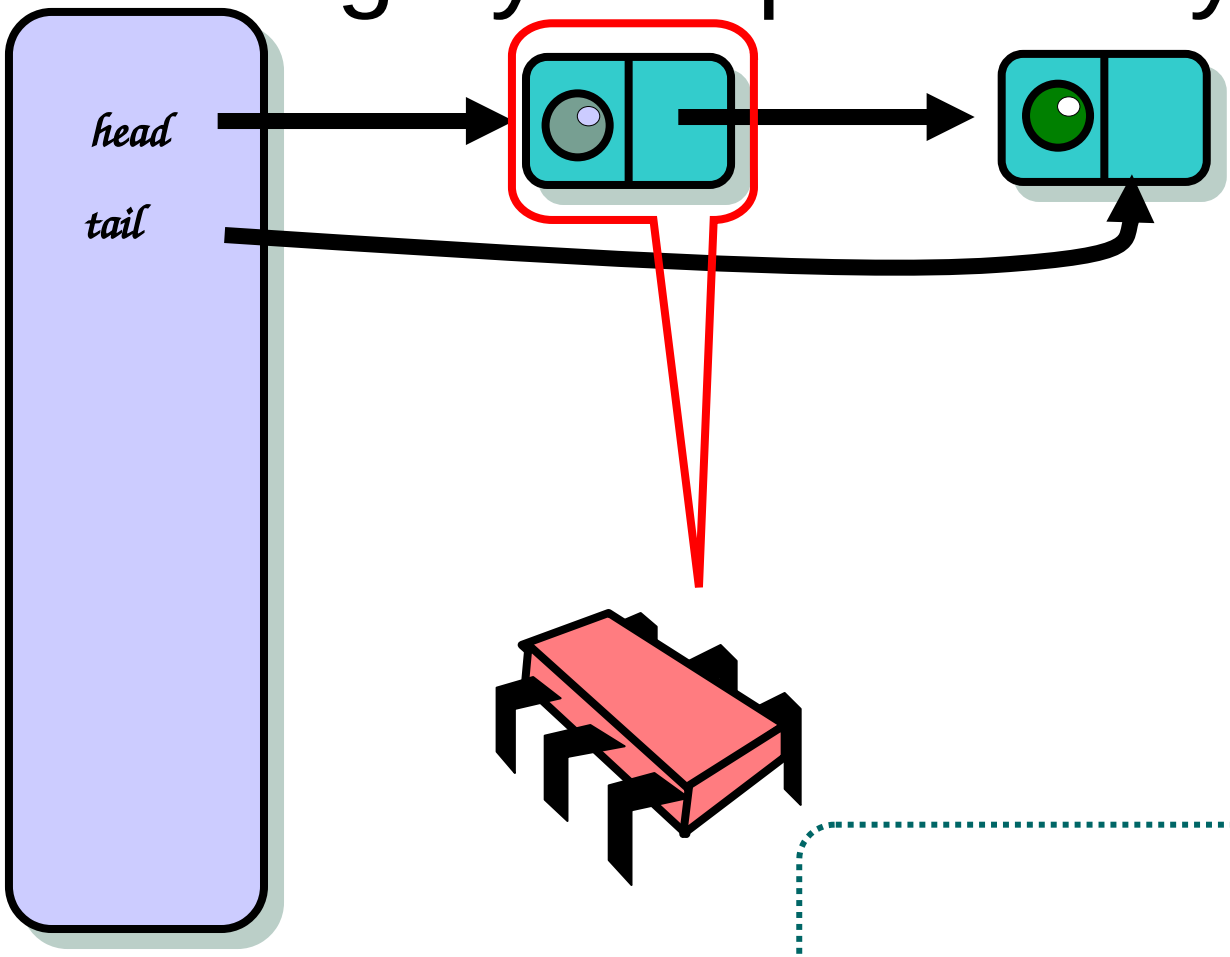
Dequeuer



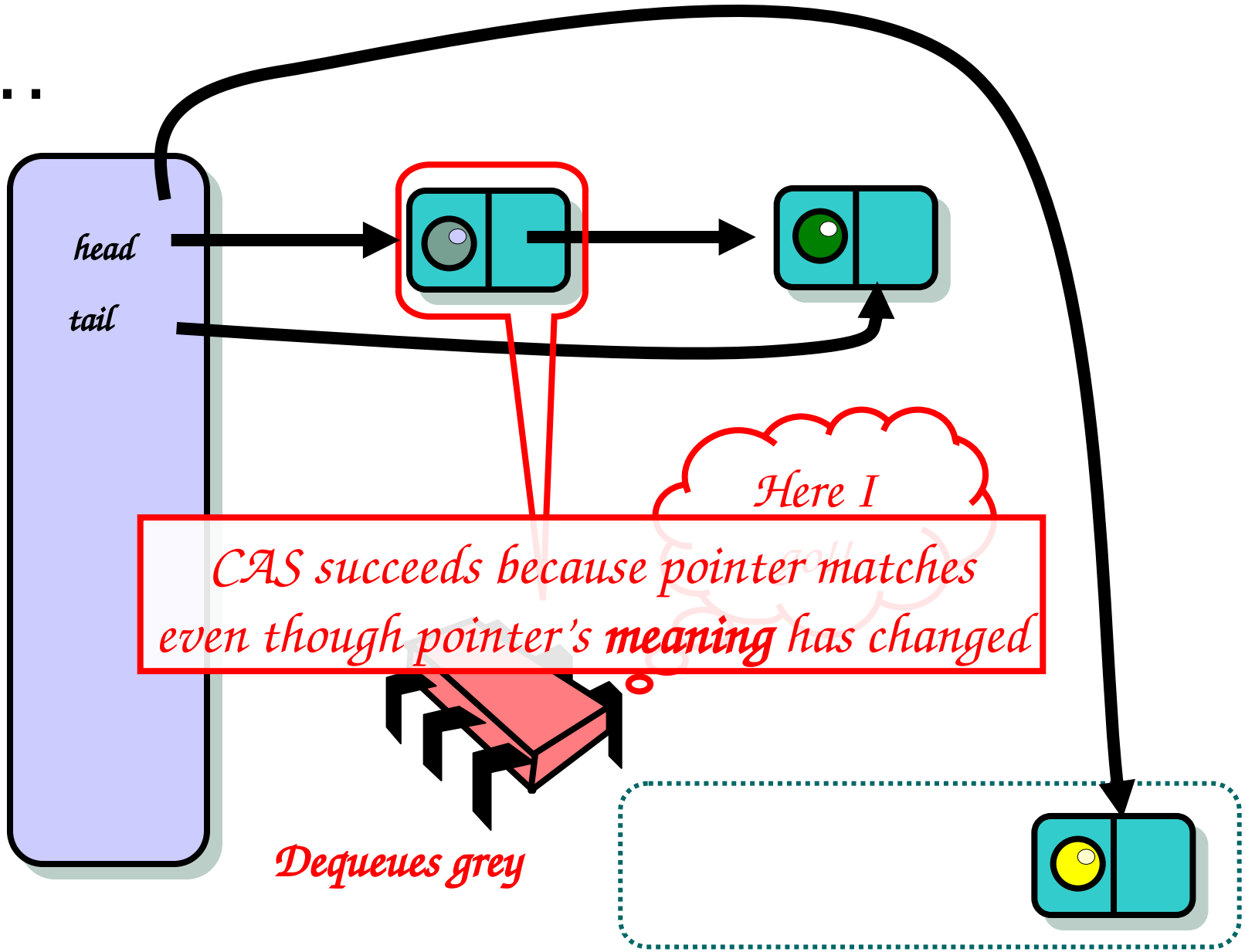
But the red thread still had grey marked



And according to its previous calculation grey still points to yellow



So…


head
tail
Here I
CAS succeeds because pointer matches
even though pointer's meaning has changed
Dequeues grey

ABA problem

- Head ends up pointing to a node that is not in the queue anymore, but has been added to the free-list

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

Dual Data Structures

- To reduce the synchronization overhead of the synchronous queue we split enq() and deq() into two steps

deq() method

- If a dequeuer tries to remove an item from an empty queue:
  - It puts a reservation object in the queue to indicate that the deq() is waiting for an enq() to rendezvous with
  - deq() spins on flag in reservation

enq() method

- When enq() sees reservation:
  - Fulfills the reservation by depositing an item
  - Notifies deq() by setting object's flag

Dual Data Structure

- An enq() can also add its own reservation to the queue and spin on the flag waiting for a deq()
- At any time the queue contains either enq() reservations, deq() reservations or is empty

Dual Data Structure

- Methods take effect in two stages:
  - Reservation and fulfillment

Dual Data structures

- Good things:
  - Waiting threads spin on a locally cached flag
  - Ensures fairness
  - Linearizable

Node class

```
private enum NodeType {ITEM, RESERVATION};
private class Node {
      volatile NodeType type;
      volatile AtomicReference<T> item;
      volatile AtomicReference<Node> next;

      Node (T myItem, NodeType myType) {
            item = new AtomicReference<T> (myItem);
            next = new AtomicReference<Node>(null);
            type = myType;
      }
}
```

enq()

```
public void enq(T e) {
      Node offer = new Node (e, ITEM);
      while (true) {
        Node t = tail.get(), h = head.get();
        if (h == t || t.type == ITEM) {
          Node n = t.next.get();
          if (t == tail.get()) {
            if (n != null) {
              tail.compareAndSet(t, n);
            } else if (t.next.CAS(n, offer)) {
              tail.compareAndSet(t, offer);
              while (offer.item.get() == e) {}
              h = head.get();
              if (offer == h.next.get())
                head.compareAndSet(h, offer);
              return; }}
```

enq()

Create new node

```
public void enq(T e) {
      Node offer = new Node (e, ITEM);
      while (true) {
        Node t = tail.get(), h = head.get();
        if (h == t || t.type == ITEM) {
          Node n = t.next.get();
          if (t == tail.get()) {
            if (n != null) {
              tail.compareAndSet(t, n);
            } else if (t.next.CAS(n, offer)) {
              tail.compareAndSet(t, offer);
              while (offer.item.get() == e) {}
              h = head.get();
              if (offer == h.next.get())
                head.compareAndSet(h, offer);
              return; }}
```

enq()

If the queue is empty OR if it contains only ITEMS…

```
public void enq(T e) {
      Node offer = new Node (e, ITEM);
      while (true) {
        Node t = tail.get(), h = head.get();
        if (h == t || t.type == ITEM) {
          Node n = t.next.get();
          if (t == tail.get()) {
            if (n != null) {
              tail.compareAndSet(t, n);
            } else if (t.next.CAS(n, offer)) {
              tail.compareAndSet(t, offer);
              while (offer.item.get() == e) {}
              h = head.get();
              if (offer == h.next.get())
                head.compareAndSet(h, offer);
              return; }}
```

enq()

…then no reservations only add ITEM

```
public void enq(T e) {
      Node offer = new Node (e, ITEM);
      while (true) {
        Node t = tail.get(), h = head.get();
        if (h == t || t.type == ITEM) {
          Node n = t.next.get();
          if (t == tail.get()) {
            if (n != null) {
              tail.compareAndSet(t, n);
            } else if (t.next.CAS(n, offer)) {
              tail.compareAndSet(t, offer);
              while (offer.item.get() == e) {}
              h = head.get();
              if (offer == h.next.get())
                head.compareAndSet(h, offer);
              return; }}
```

enq()

If tail not last value move it up

```
public void enq(T e) {
      Node offer = new Node (e, ITEM);
      while (true) {
        Node t = tail.get(), h = head.get();
        if (h == t || t.type == ITEM) {
          Node n = t.next.get();
          if (t == tail.get()) {
            if (n != null) {
              tail.compareAndSet(t, n);
            } else if (t.next.CAS(n, offer)) {
              tail.compareAndSet(t, offer);
              while (offer.item.get() == e) {}
              h = head.get();
              if (offer == h.next.get())
                head.compareAndSet(h, offer);
              return; }}
```

enq()

Else add new ITEM node to queue

```
public void enq(T e) {
      Node offer = new Node (e, ITEM);
      while (true) {
        Node t = tail.get(), h = head.get();
        if (h == t || t.type == ITEM) {
          Node n = t.next.get();
          if (t == tail.get()) {
            if (n != null) {
              tail.compareAndSet(t, n);
            } else if (t.next.CAS(n, offer)) {
              tail.compareAndSet(t, offer);
              while (offer.item.get() == e) {}
              h = head.get();
              if (offer == h.next.get())
                head.compareAndSet(h, offer);
              return; }}
```

enq()

Now we wait until it gets dequeued

```
public void enq(T e) {
      Node offer = new Node (e, ITEM);
      while (true) {
        Node t = tail.get(), h = head.get();
        if (h == t || t.type == ITEM) {
          Node n = t.next.get();
          if (t == tail.get()) {
            if (n != null) {
              tail.compareAndSet(t, n);
            } else if (t.next.CAS(n, offer)) {
              tail.compareAndSet(t, offer);
              while (offer.item.get() == e) {}
              h = head.get();
              if (offer == h.next.get())
                head.compareAndSet(h, offer);
              return; }}
```

enq()

After it is dequeued
remove node from queue

```
public void enq(T e) {
      Node offer = new Node (e, ITEM);
      while (true) {
        Node t = tail.get(), h = head.get();
        if (h == t || t.type == ITEM) {
          Node n = t.next.get();
          if (t == tail.get()) {
            if (n != null) {
              tail.compareAndSet(t, n);
            } else if (t.next.CAS(n, offer)) {
              tail.compareAndSet(t, offer);
              while (offer.item.get() == e) {}
              h = head.get();
              if (offer == h.next.get())
                head.compareAndSet(h, offer);
              return; }}
```

enq()

```
…
  } else {
    Node n = h.next.get();
    if (t != tail.get() || h != head.get() || n ==
      null) {
      continue;
    }
    boolean success = n.item.compareAndSet(null,e);
    head.compareAndSet(h, n);
    if (success)
      return;
}}}
```

enq()

Otherwise fulfill reservation

```
…
  } else {
    Node n = h.next.get();
    if (t != tail.get() || h != head.get() || n ==
      null) {
      continue;
    }
    boolean success = n.item.compareAndSet(null,e);
    head.compareAndSet(h, n);
    if (success)
      return;
}}}
```

enq()

Make sure nothing has changed

```
…
  } else {
    Node n = h.next.get();
    if (t != tail.get() || h != head.get() || n ==
      null) {
      continue;
    }
    boolean success = n.item.compareAndSet(null,e);
    head.compareAndSet(h, n);
    if (success)
      return;
}}}
```

enq()

Change item in reservation node to enq item

```
…
  } else {
    Node n = h.next.get();
    if (t != tail.get() || h != head.get() || n ==
      null) {
      continue;
    }
    boolean success = n.item.compareAndSet(null,e);
    head.compareAndSet(h, n);
    if (success)
      return;
}}}
```