SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 431 Distributed Systems

Fall 2019

Concurrent Data Structures

https://www.cs.sfu.ca/~keval/teaching/cmpt431/fall19/

Instructor: Keval Vora

Reading

- [AMP] Chapter 10 & 11
- [Paper] Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms: https://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf

Slides based on book material

Queues

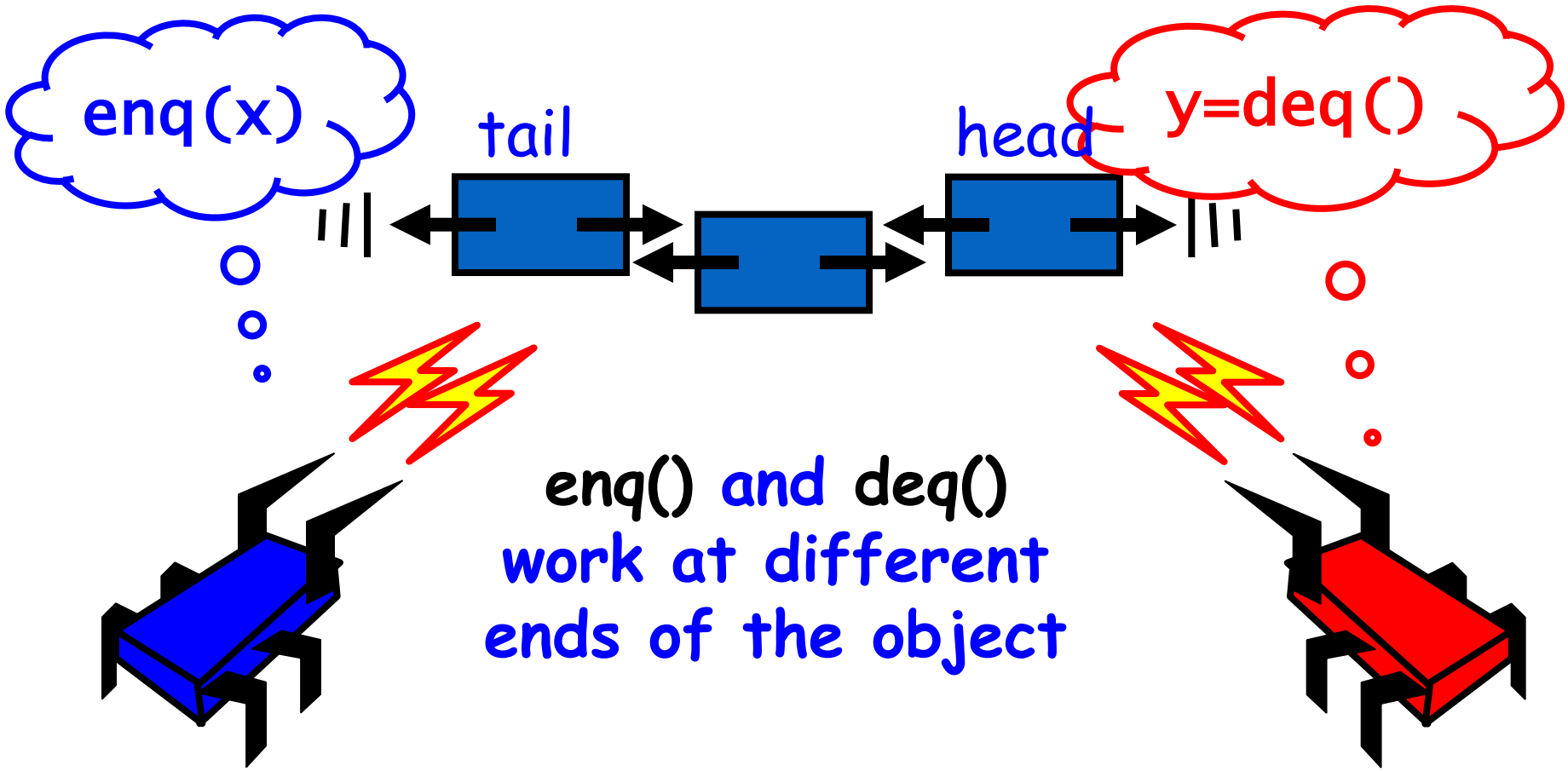
- First-in-first-out (FIFO) order
- enq() & deq()

- Bounded v/s Unbounded
  - Capacity
- Blocking (Partial) v/s Non-Blocking (Total)
  - Removal from empty queue or addition to queue that's full
  - Blocking: caller waits until state changes
  - Non-Blocking: method throws exception

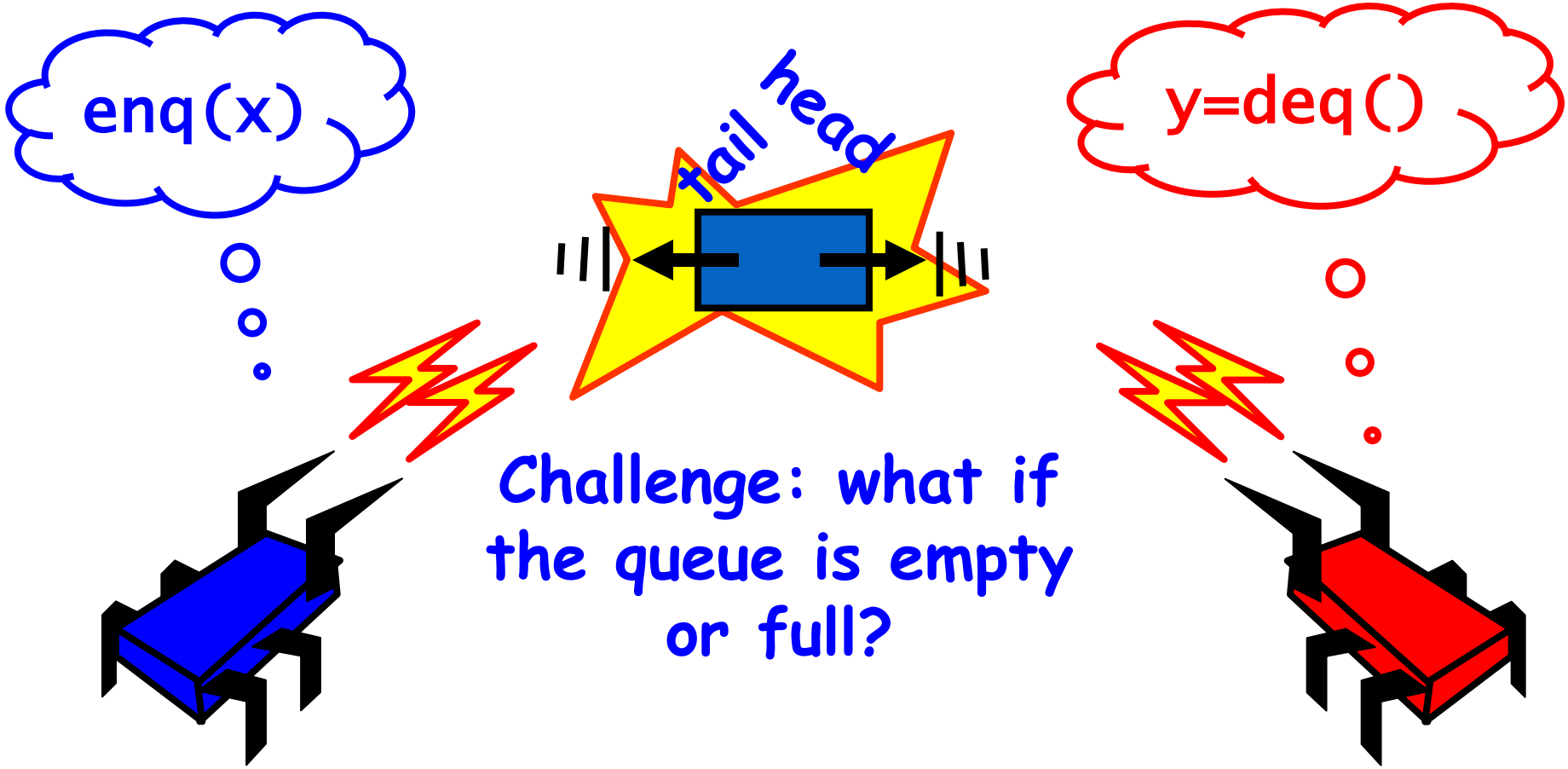
We will look at …

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

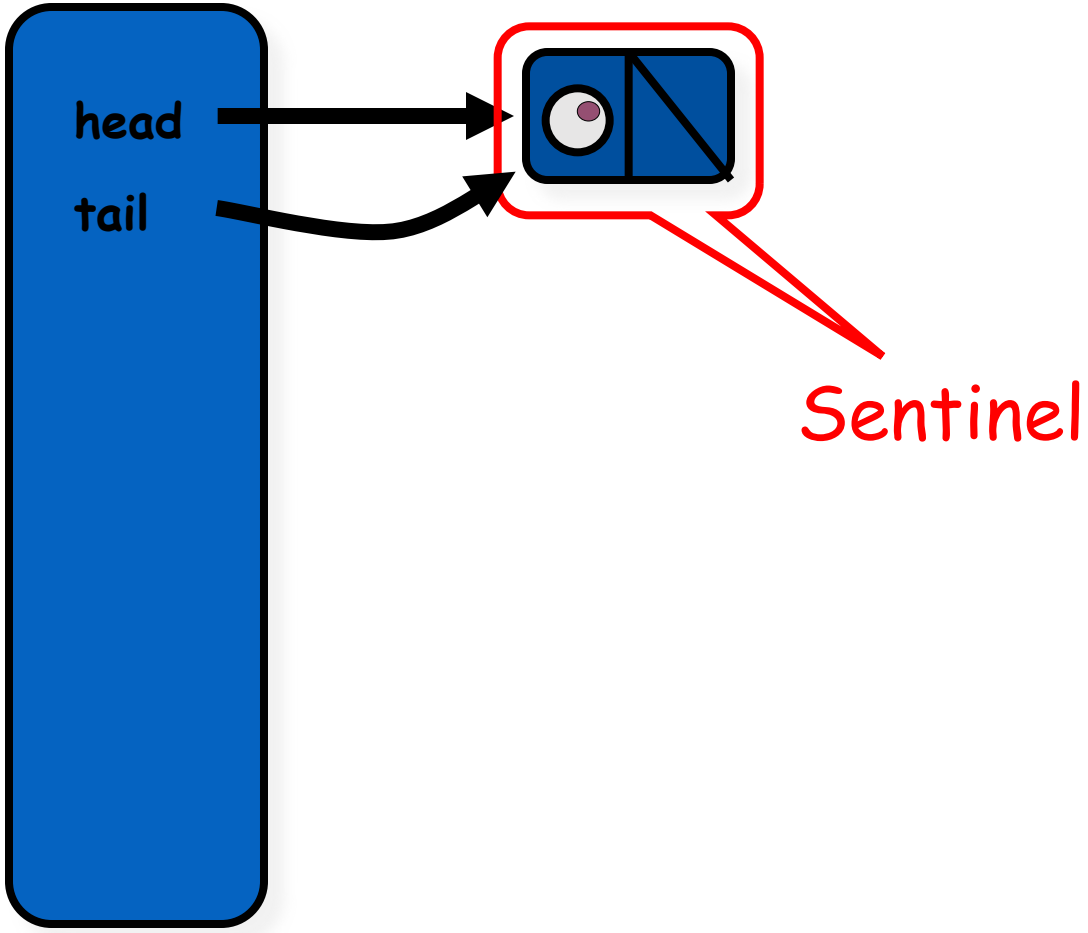
Queue: Concurrency



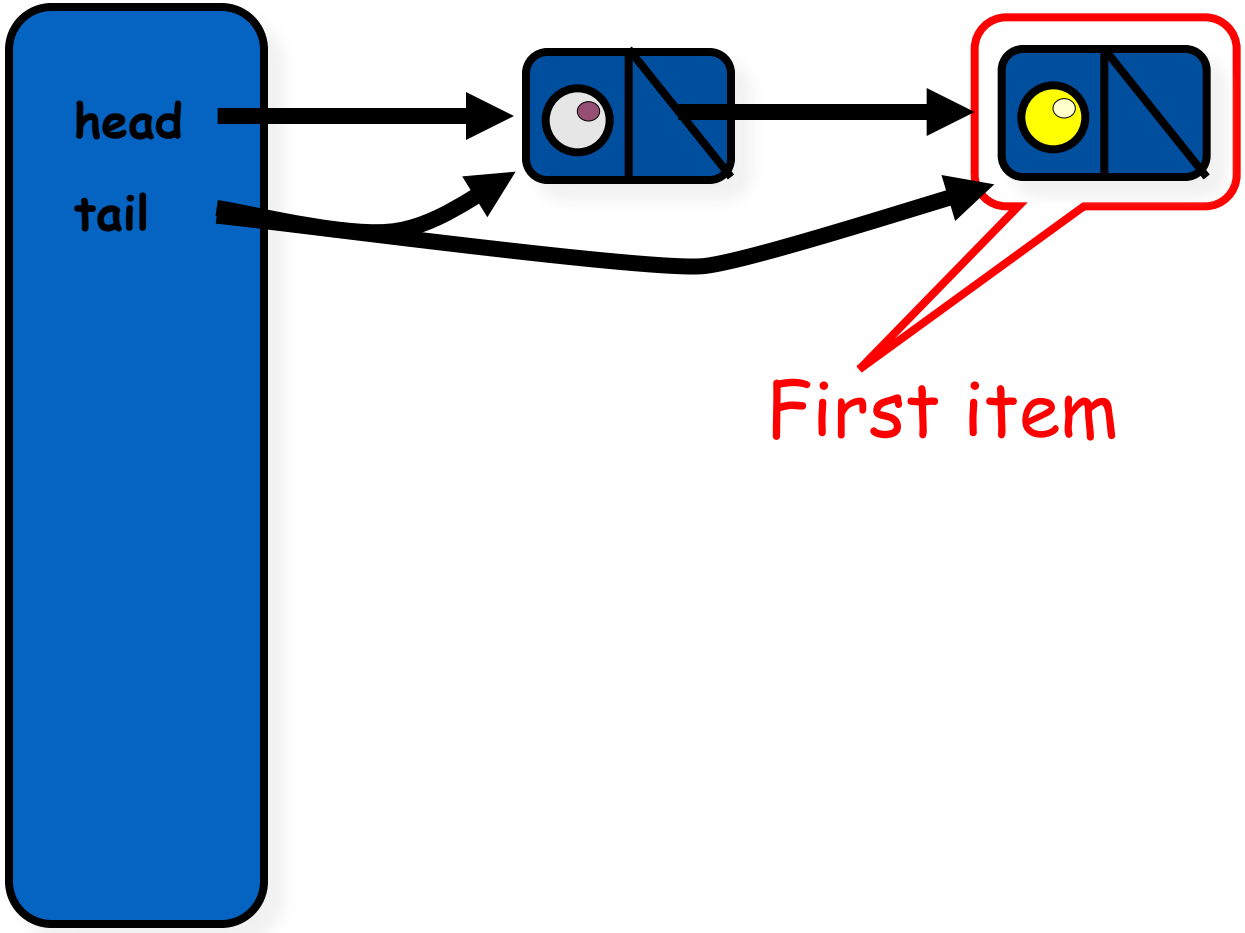
Concurrency



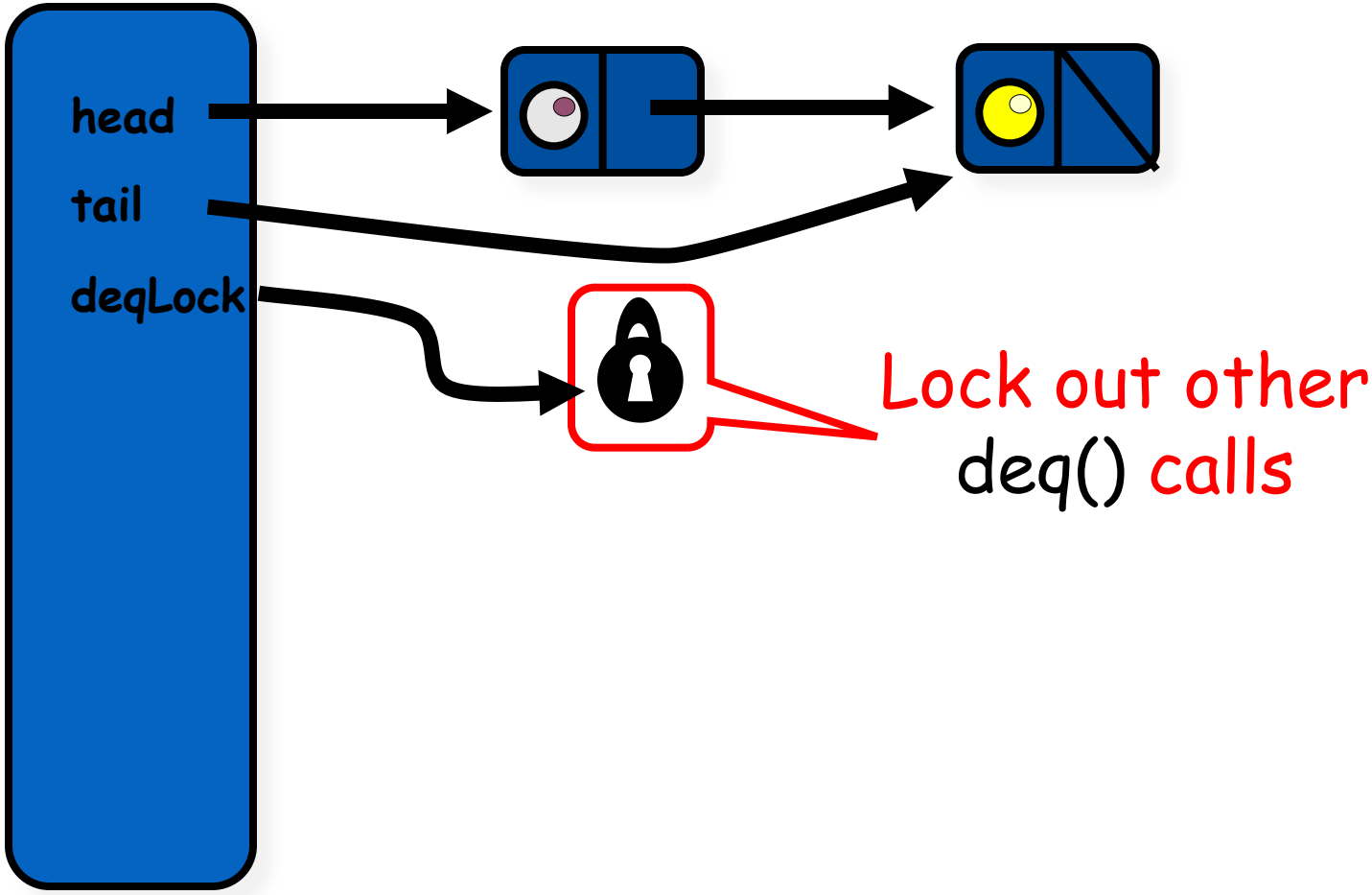
Bounded Queue



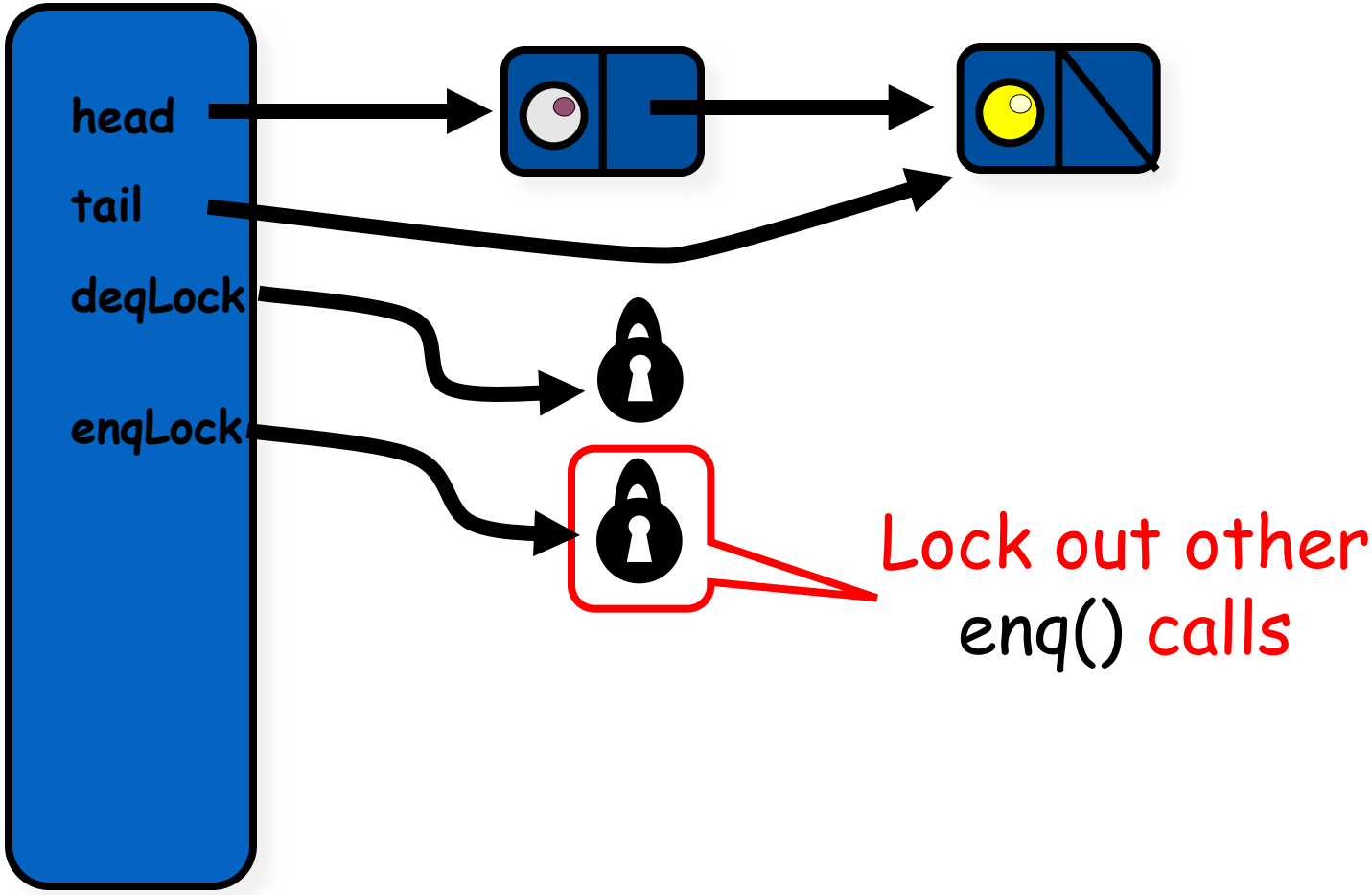
Bounded Queue



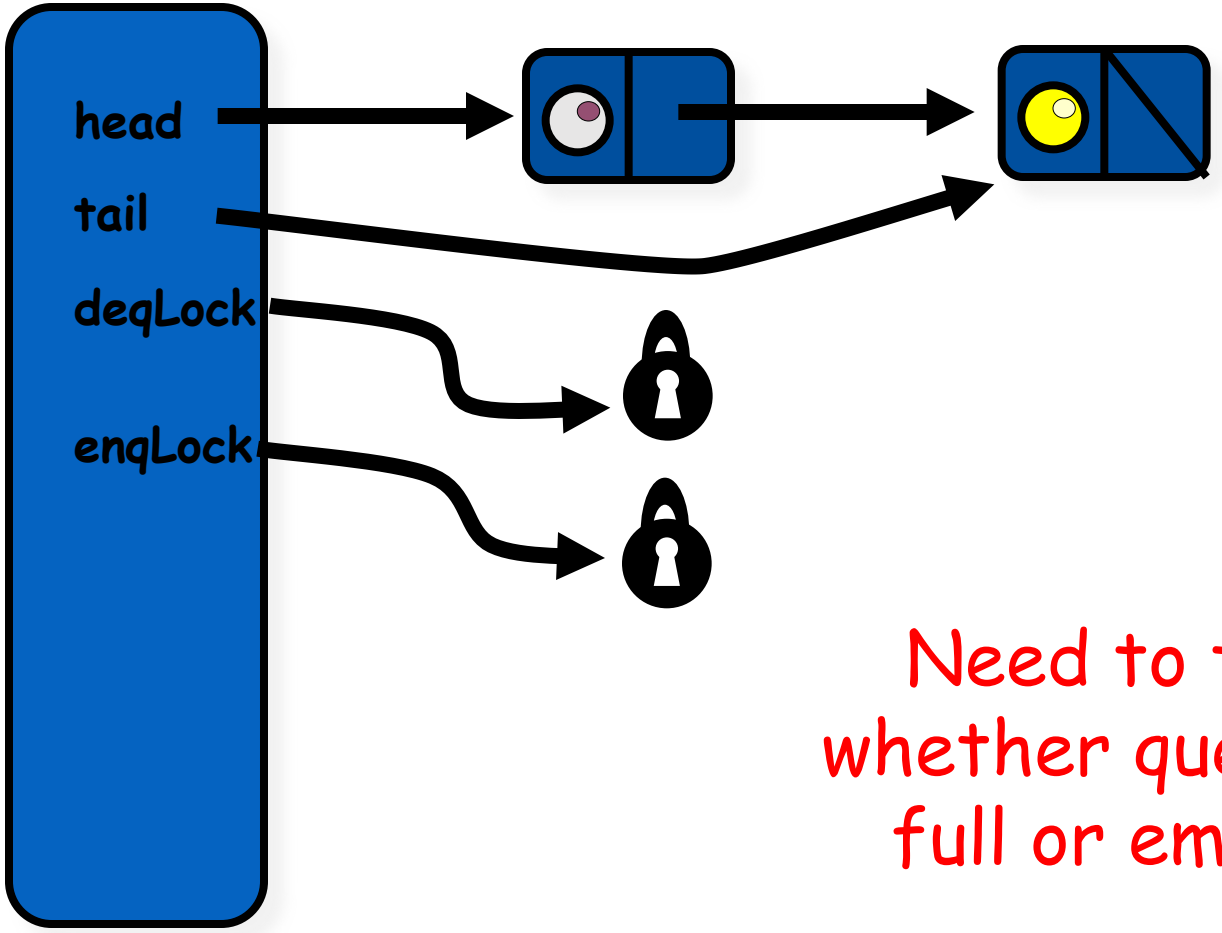
Bounded Queue



Bounded Queue

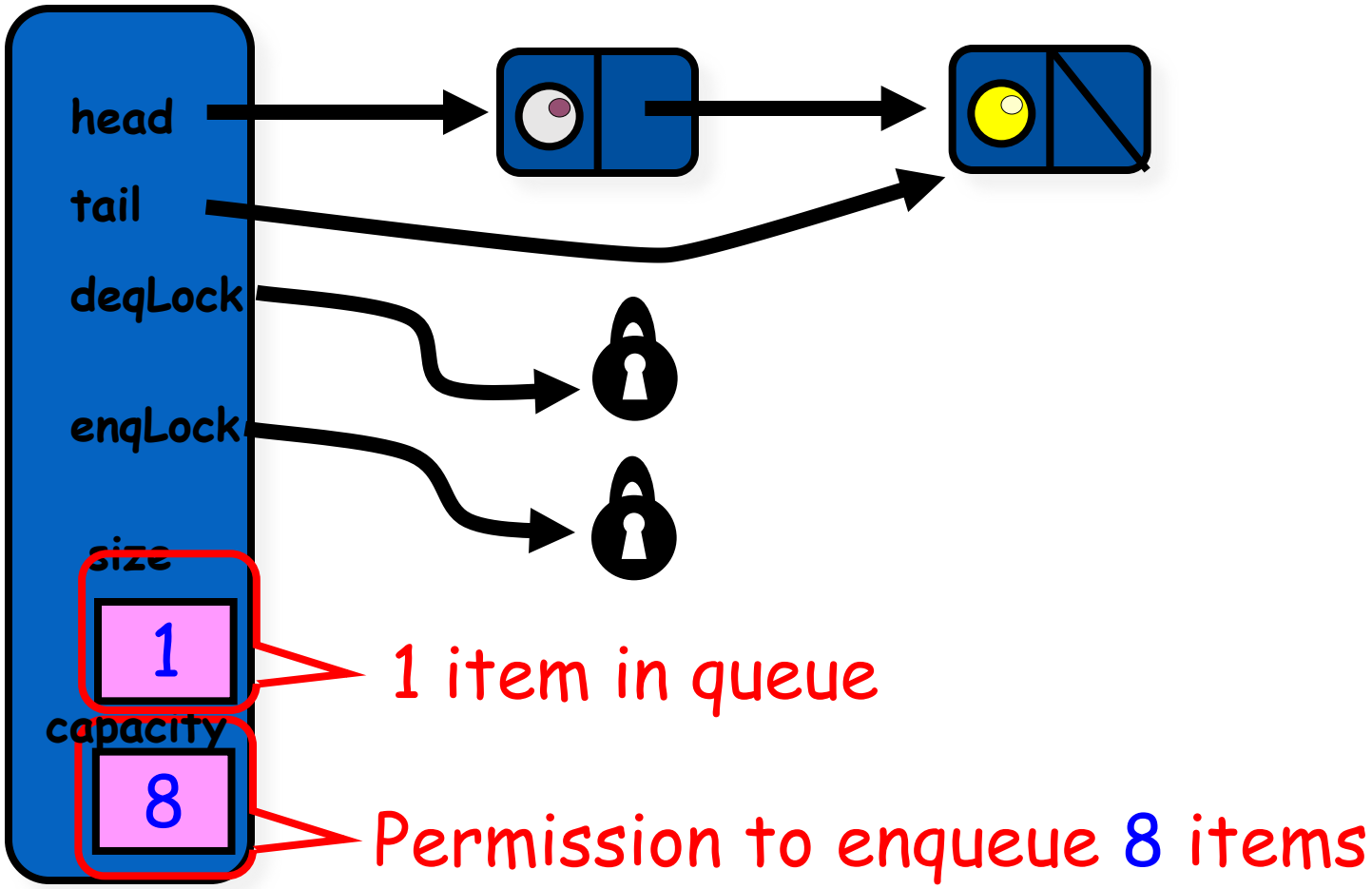


Bounded Queue

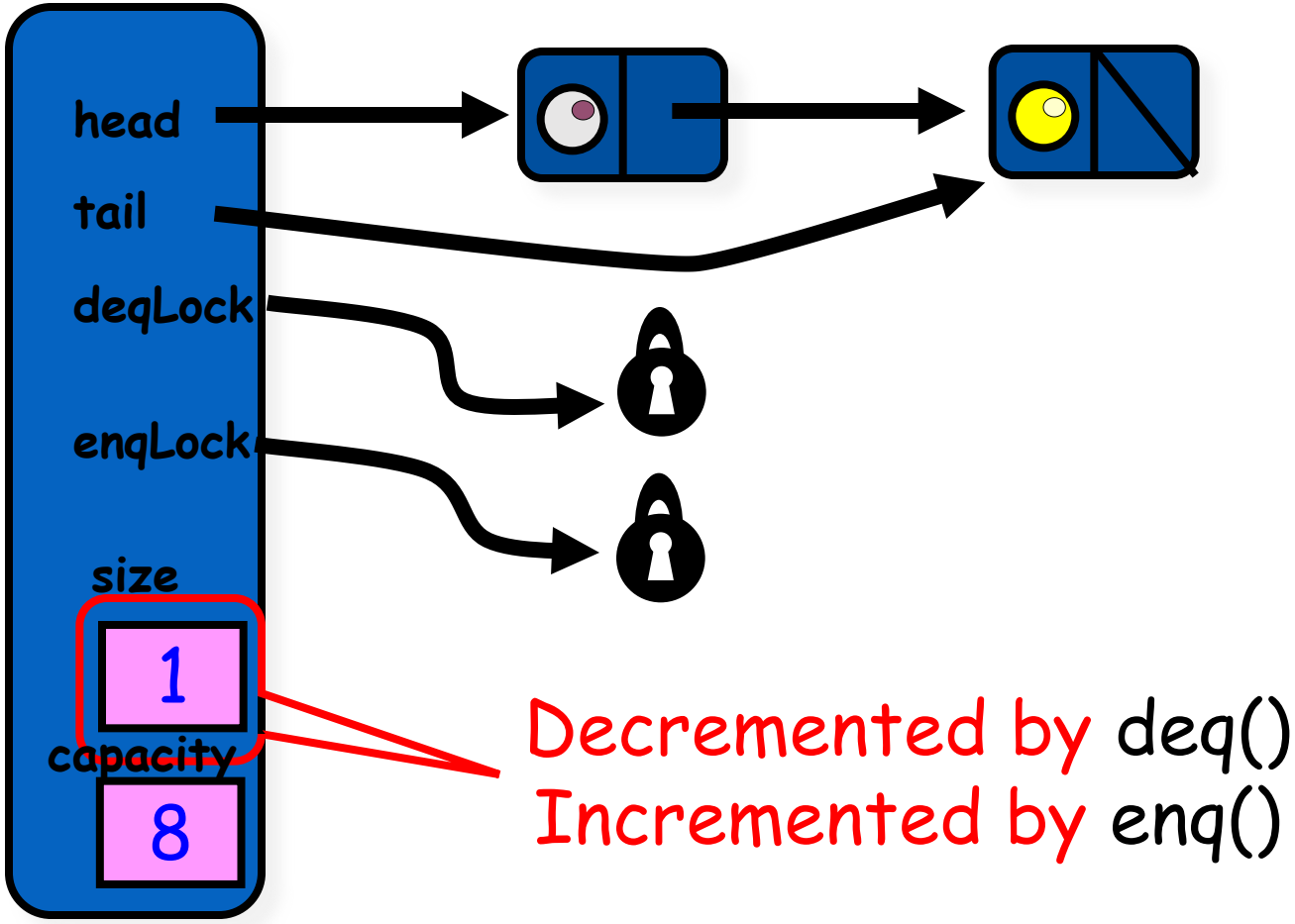


Need to tell whether queue is full or empty

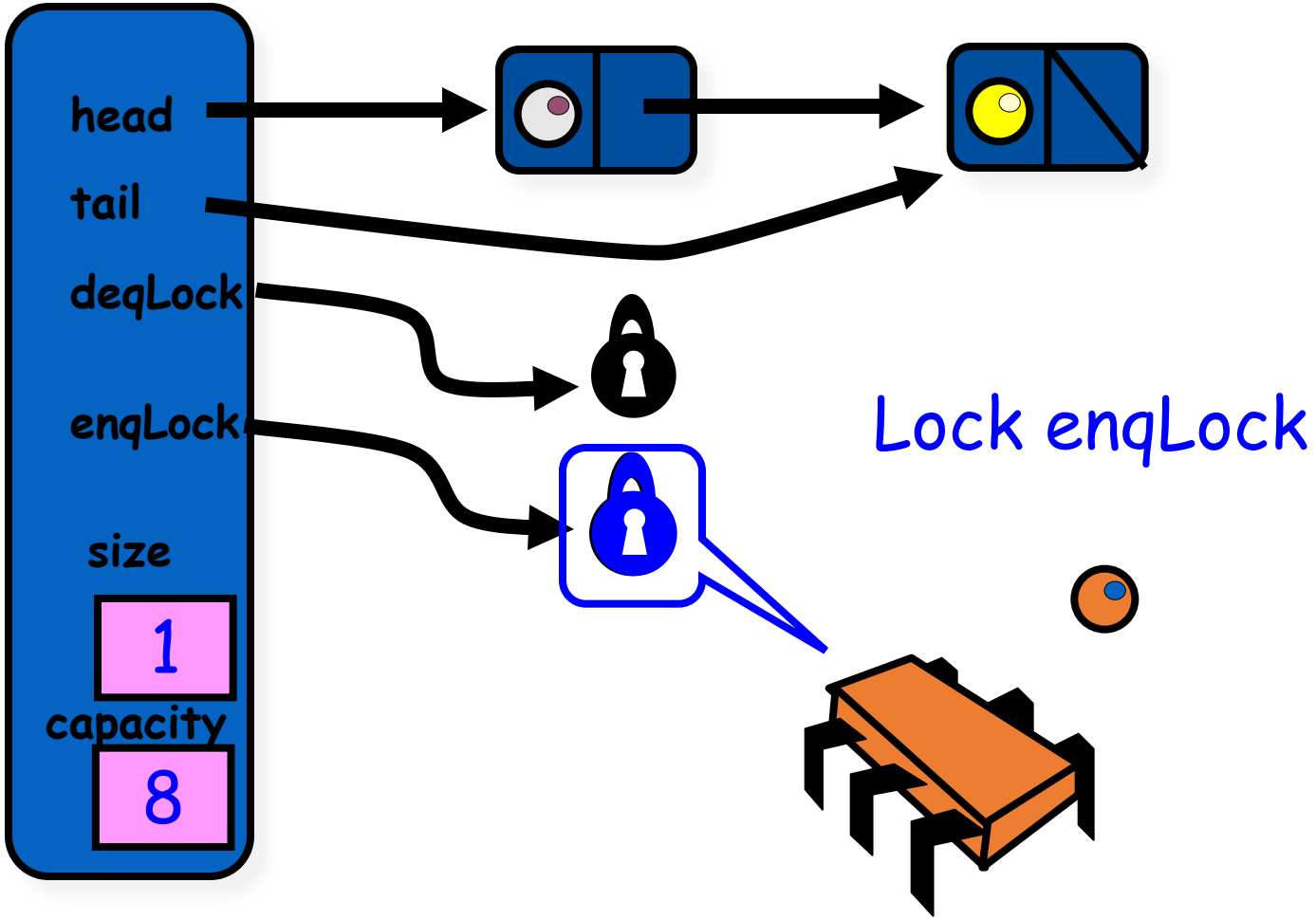
Bounded Queue



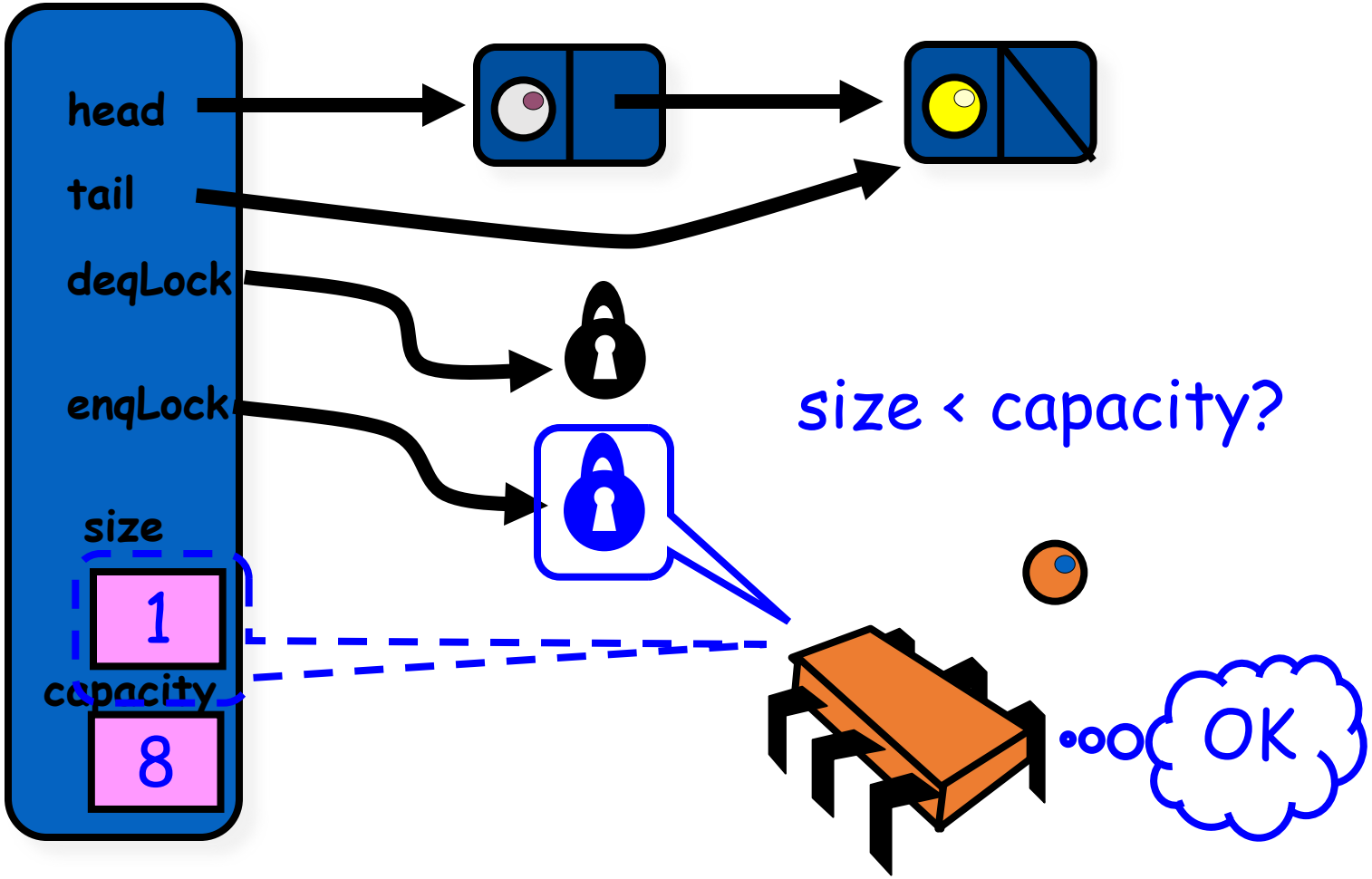
Bounded Queue



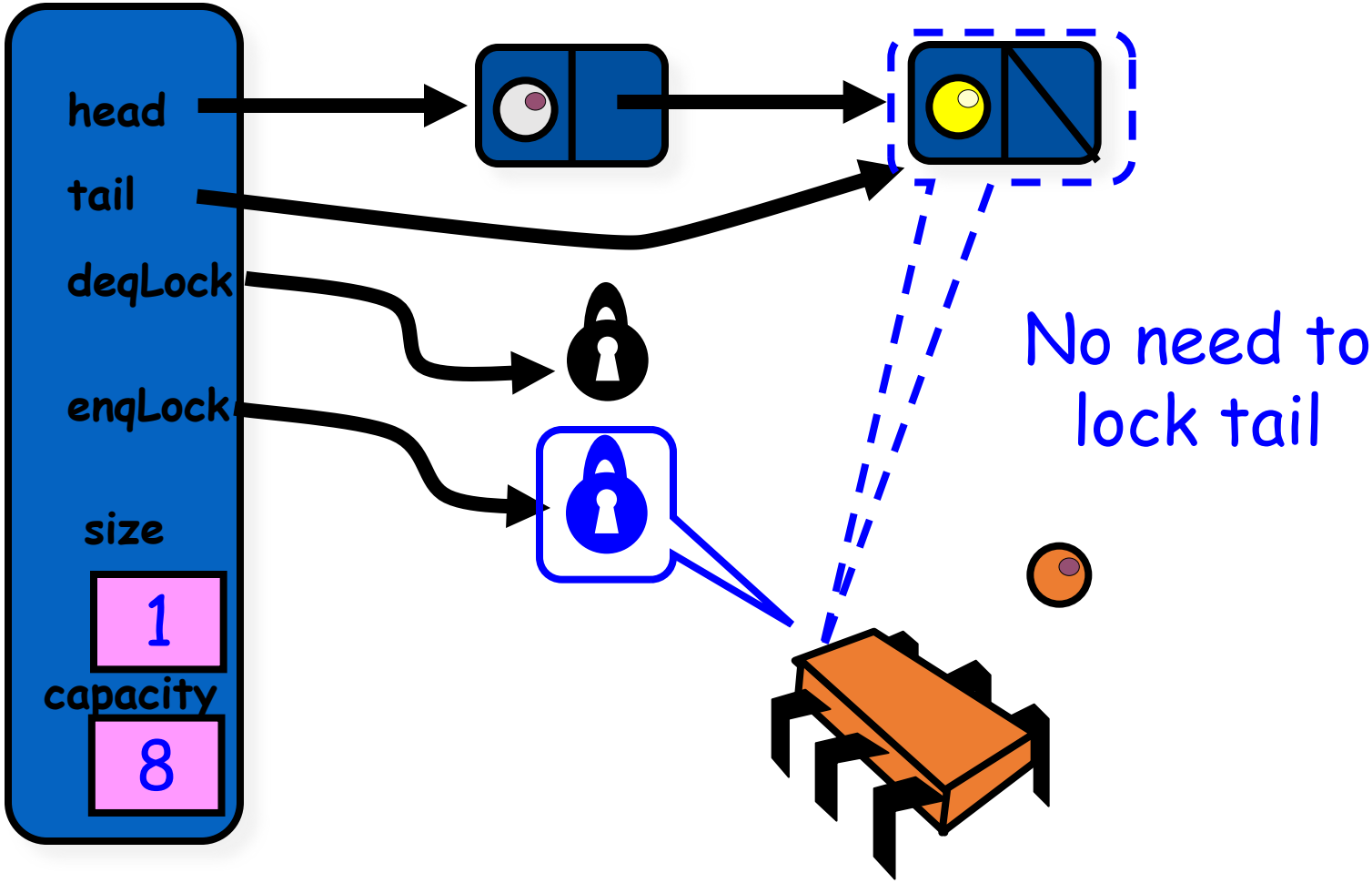
Bounded Queue: Enqueuer



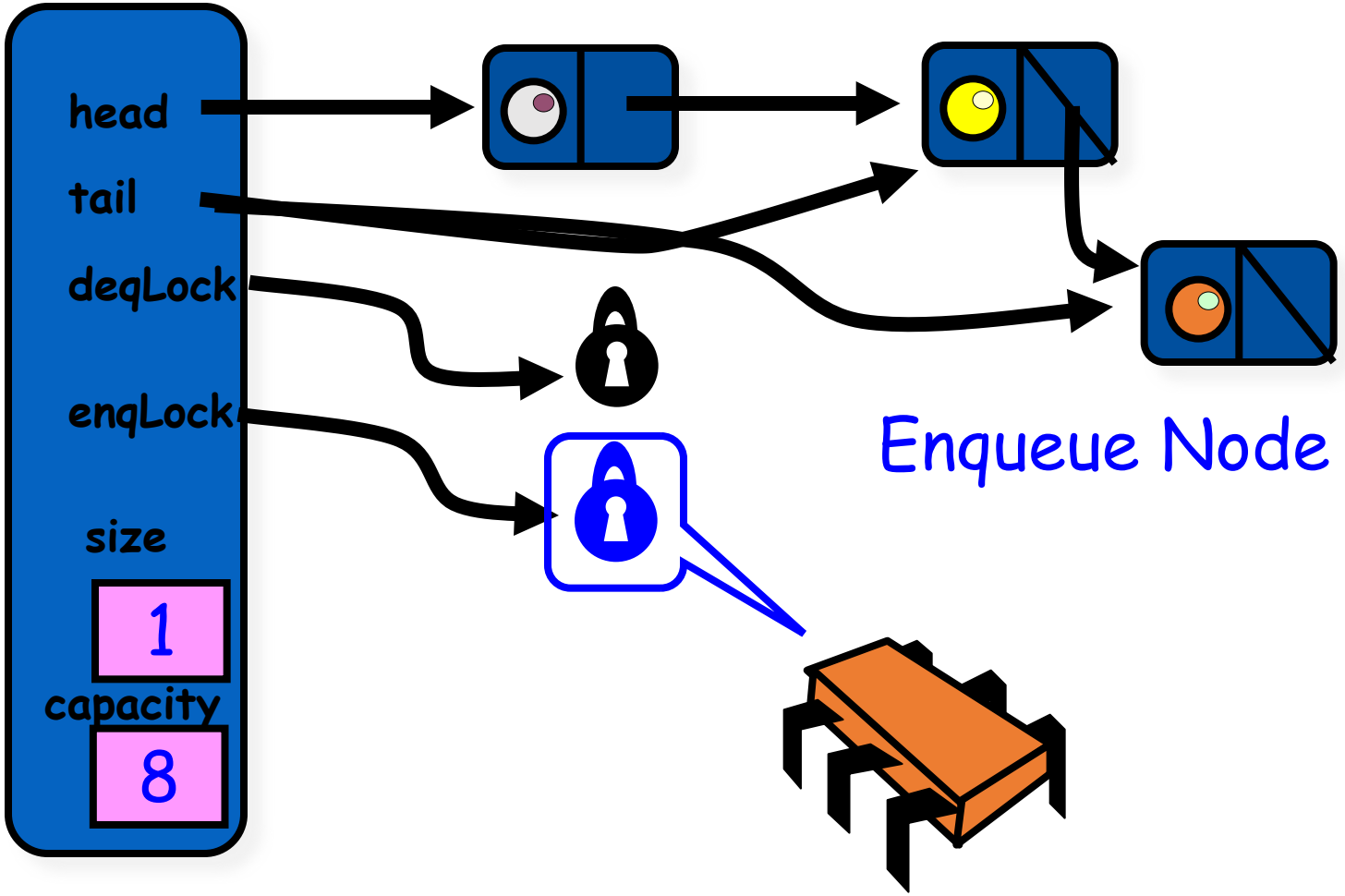
Bounded Queue: Enqueuer



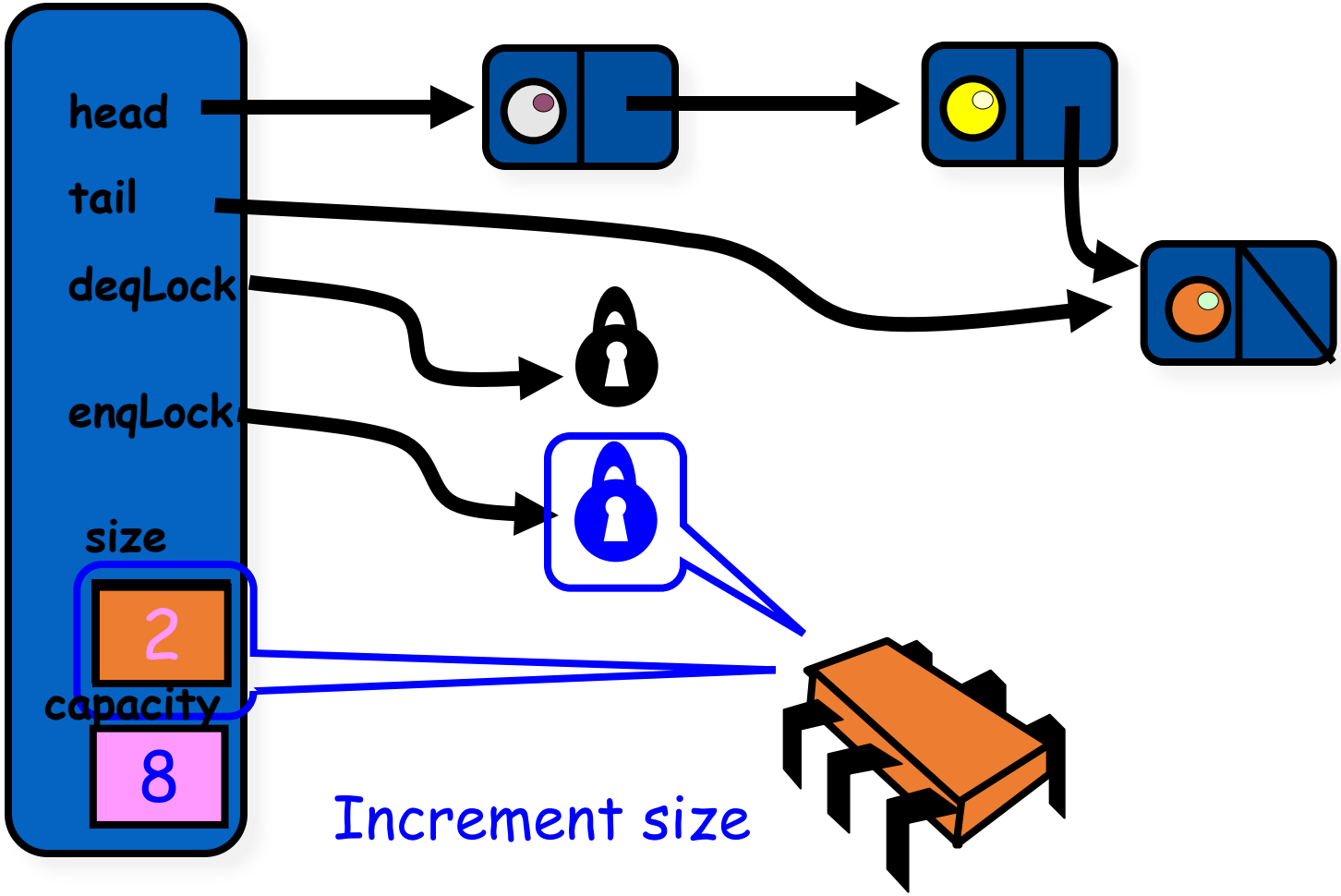
Bounded Queue: Enqueuer



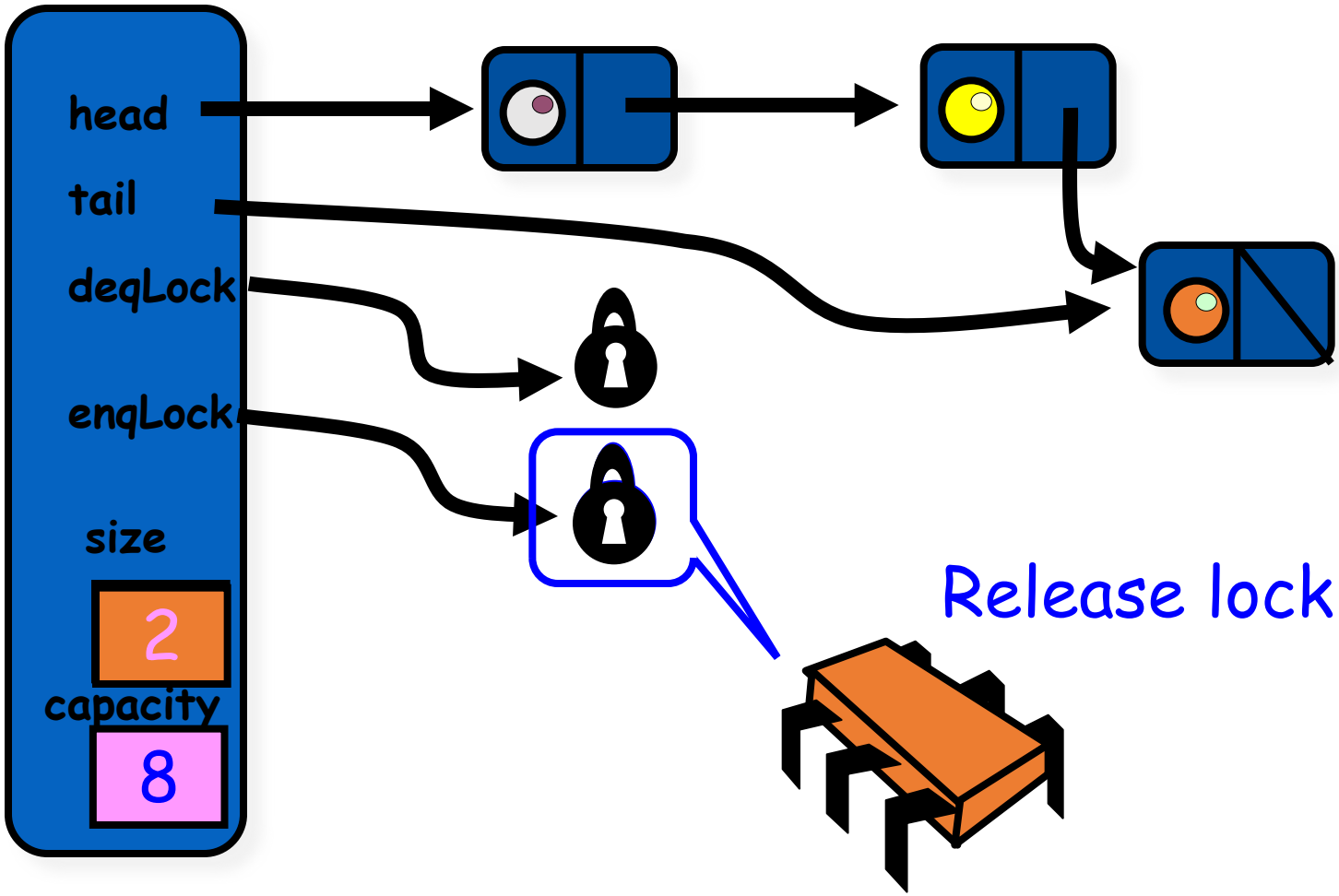
Bounded Queue: Enqueuer



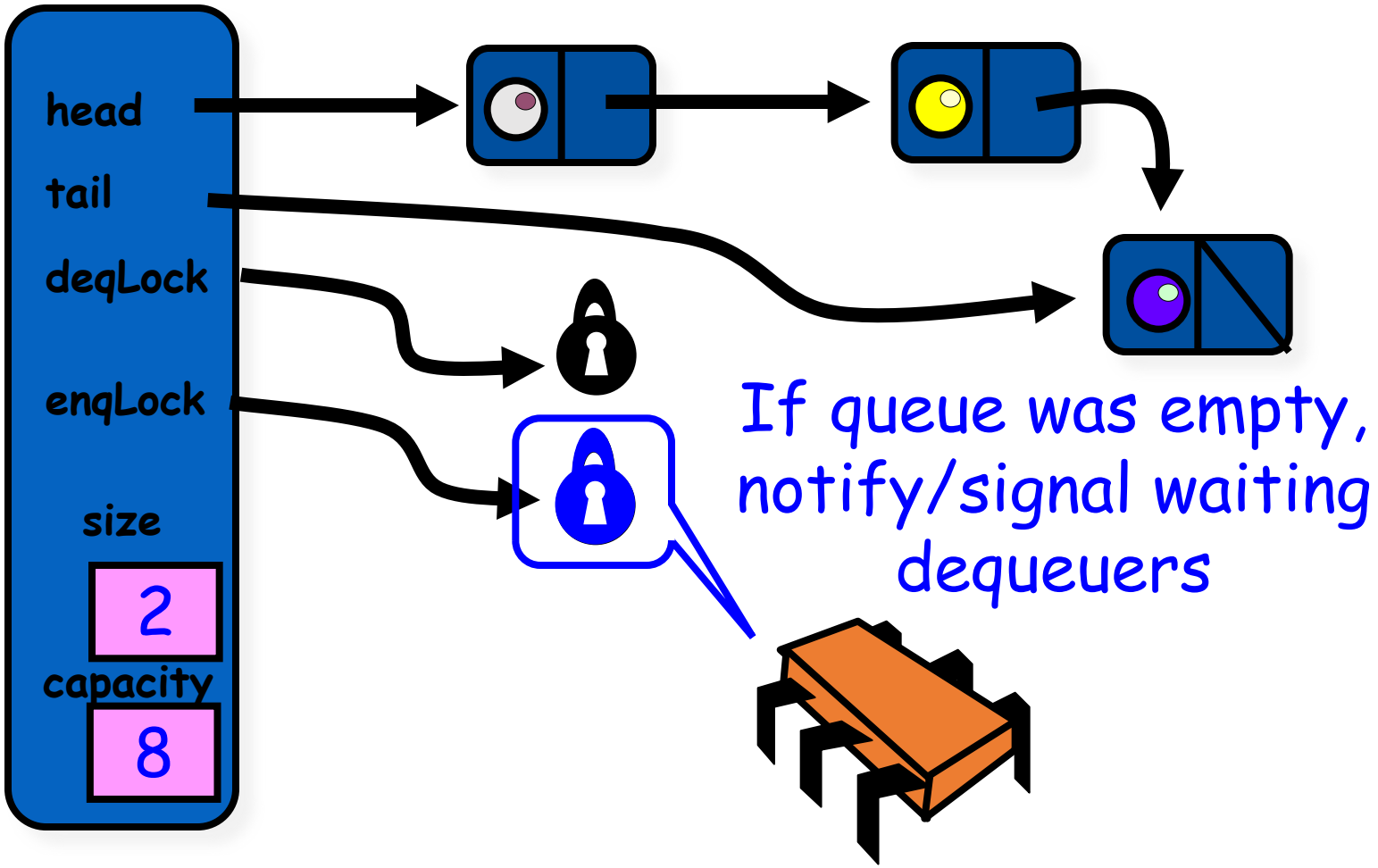
Bounded Queue: Enqueuer



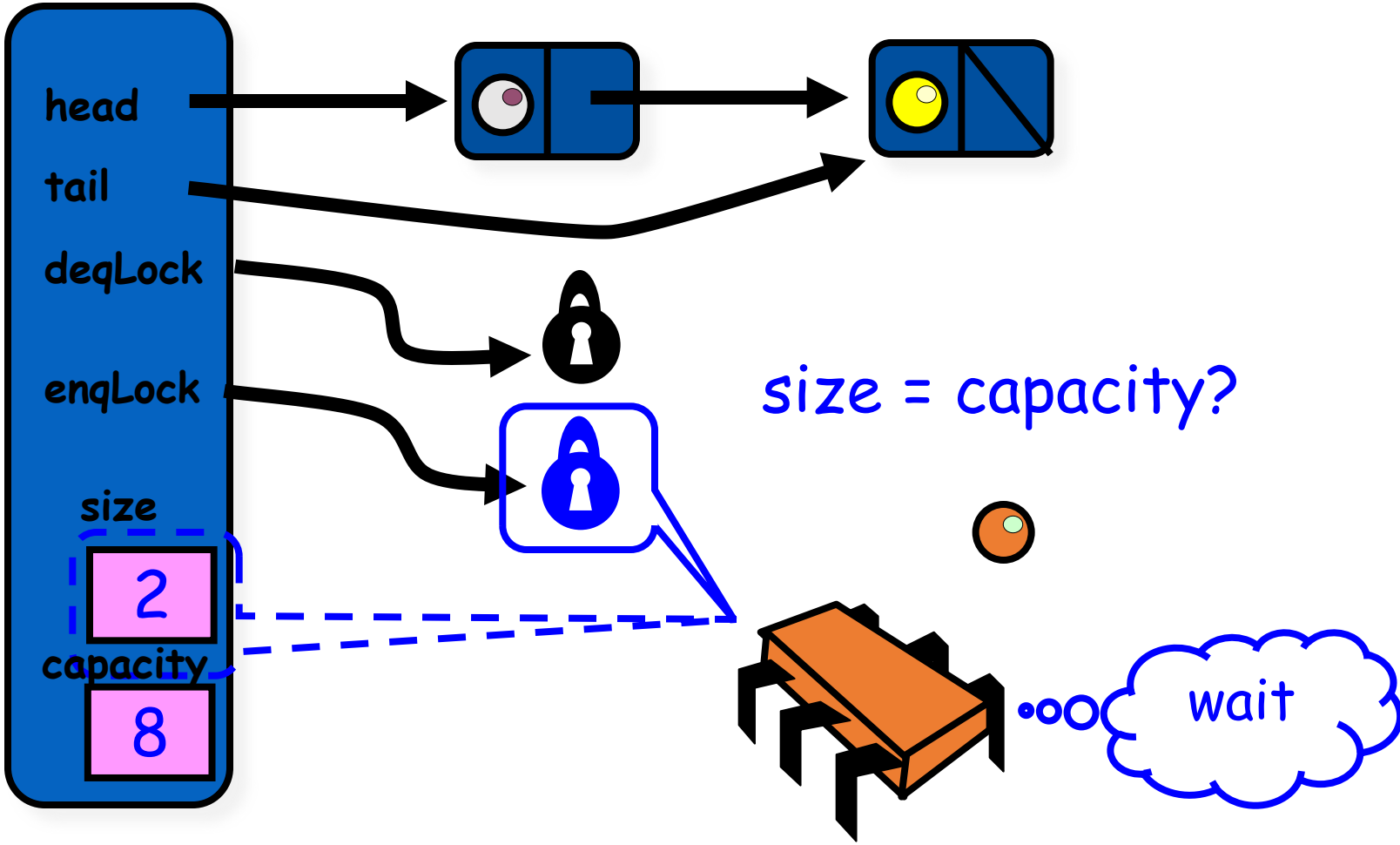
Bounded Queue: Enqueuer



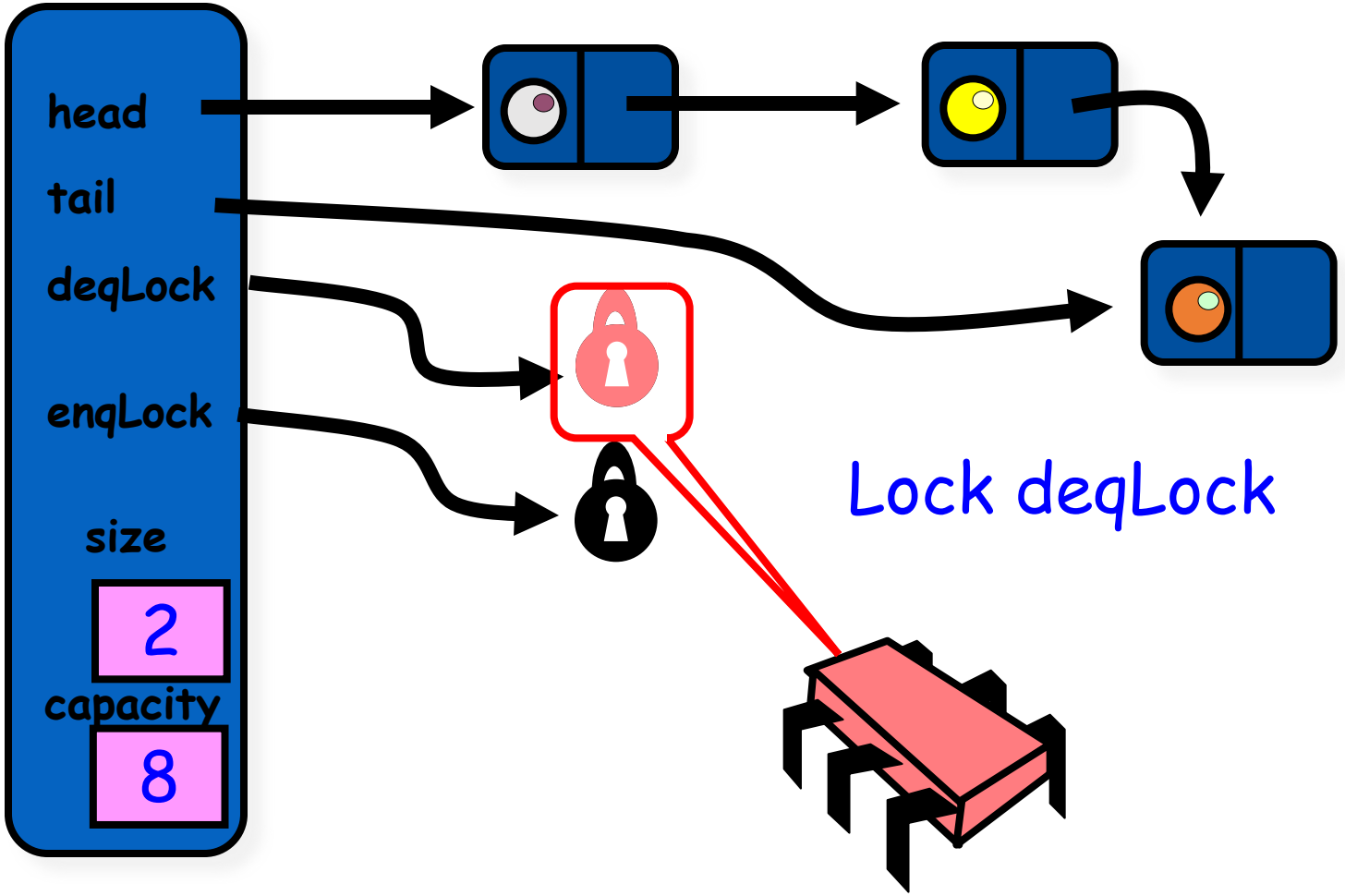
Bounded Queue: Enqueuer



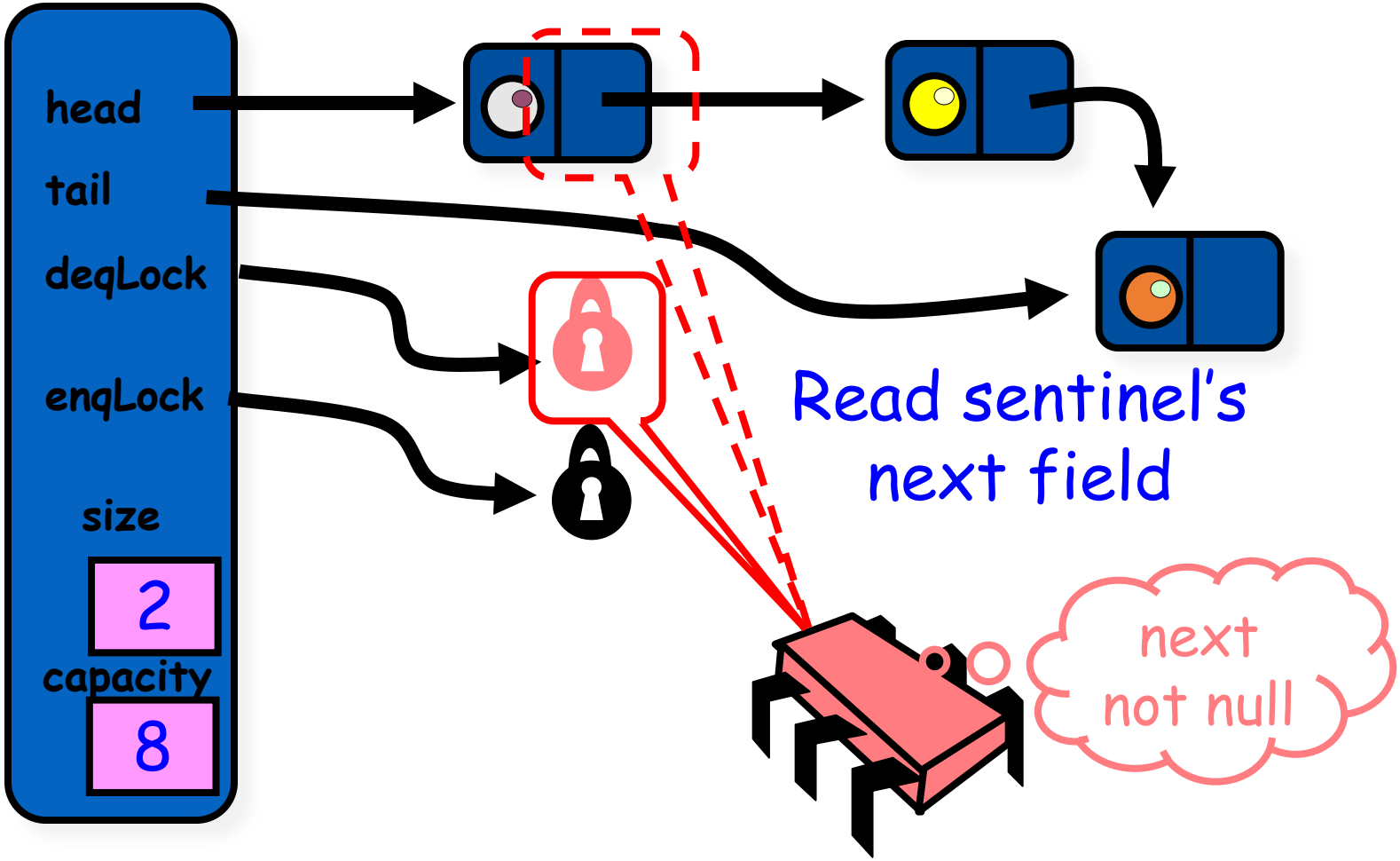
Bounded Queue: Enqueuer



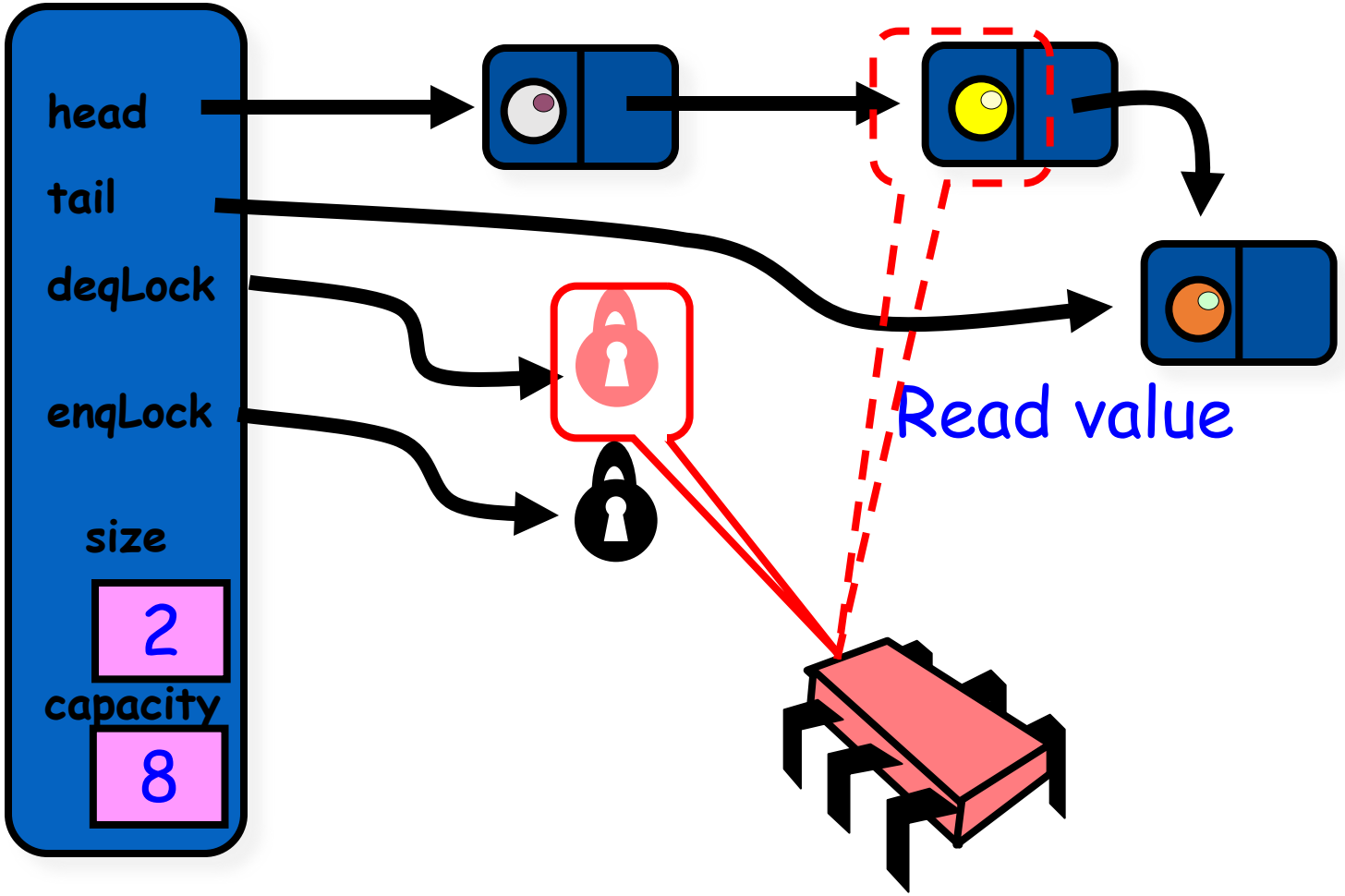
Bounded Queue: Dequeuer



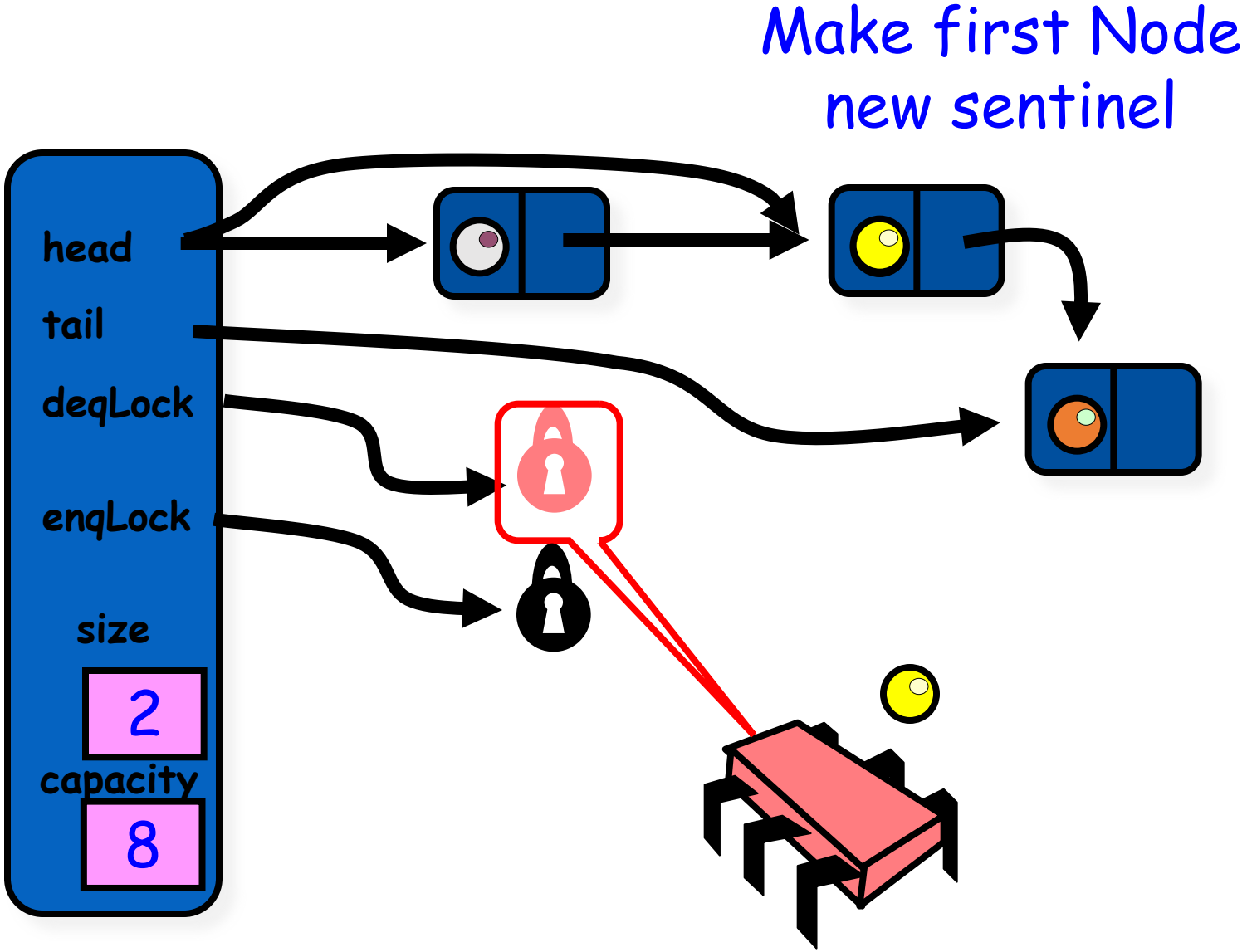
Bounded Queue: Dequeuer



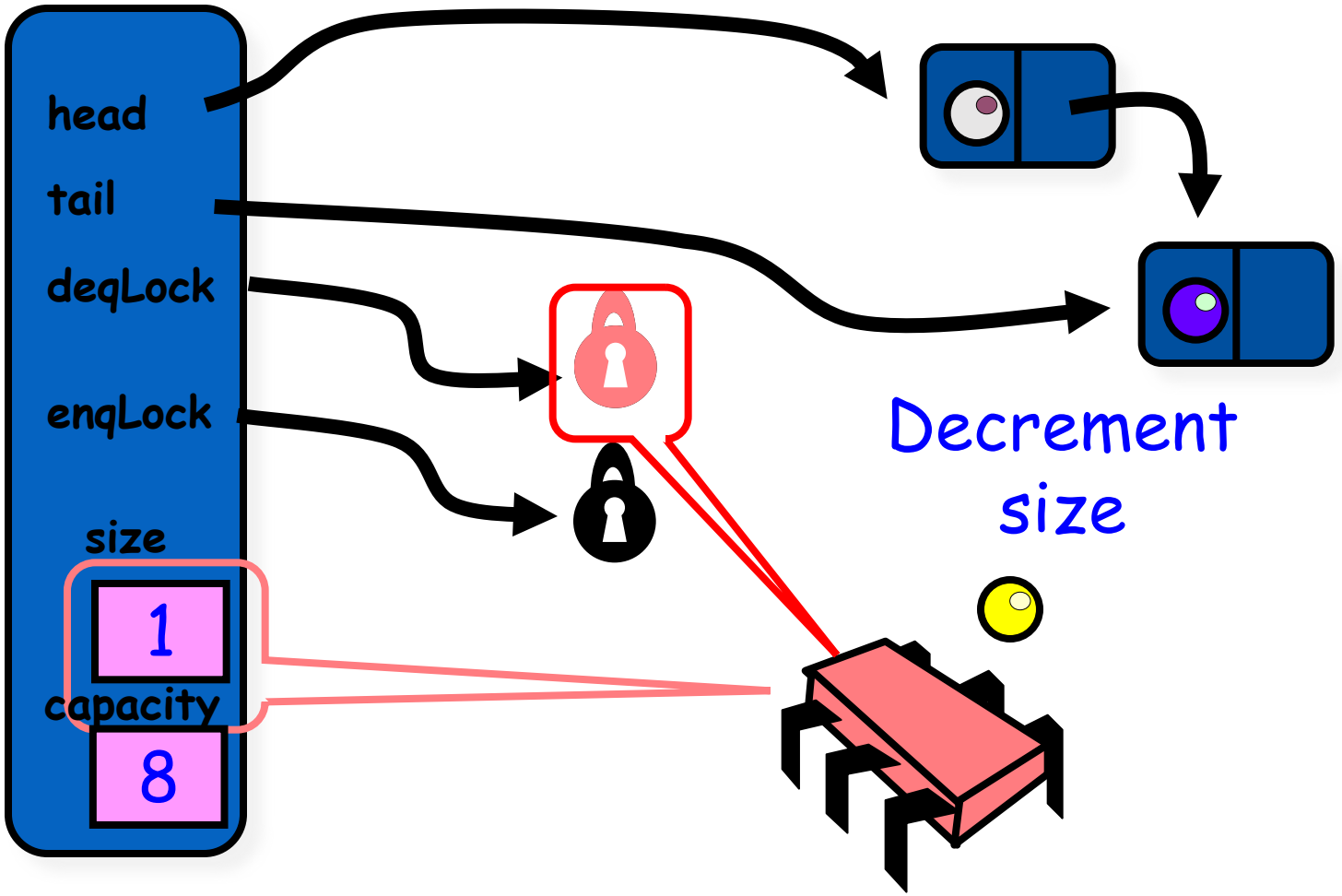
Bounded Queue: Dequeuer



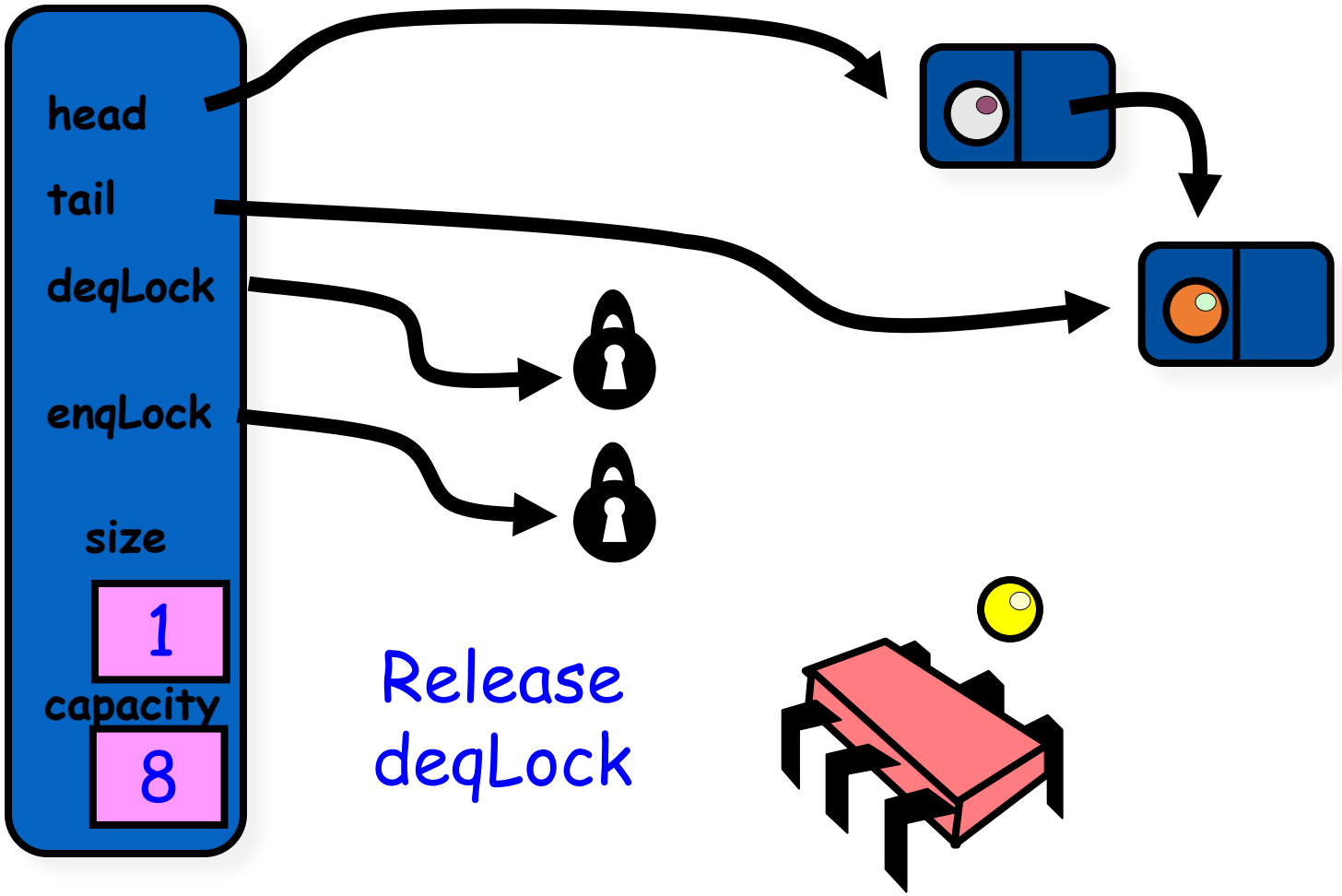
Bounded Queue: Dequeuer



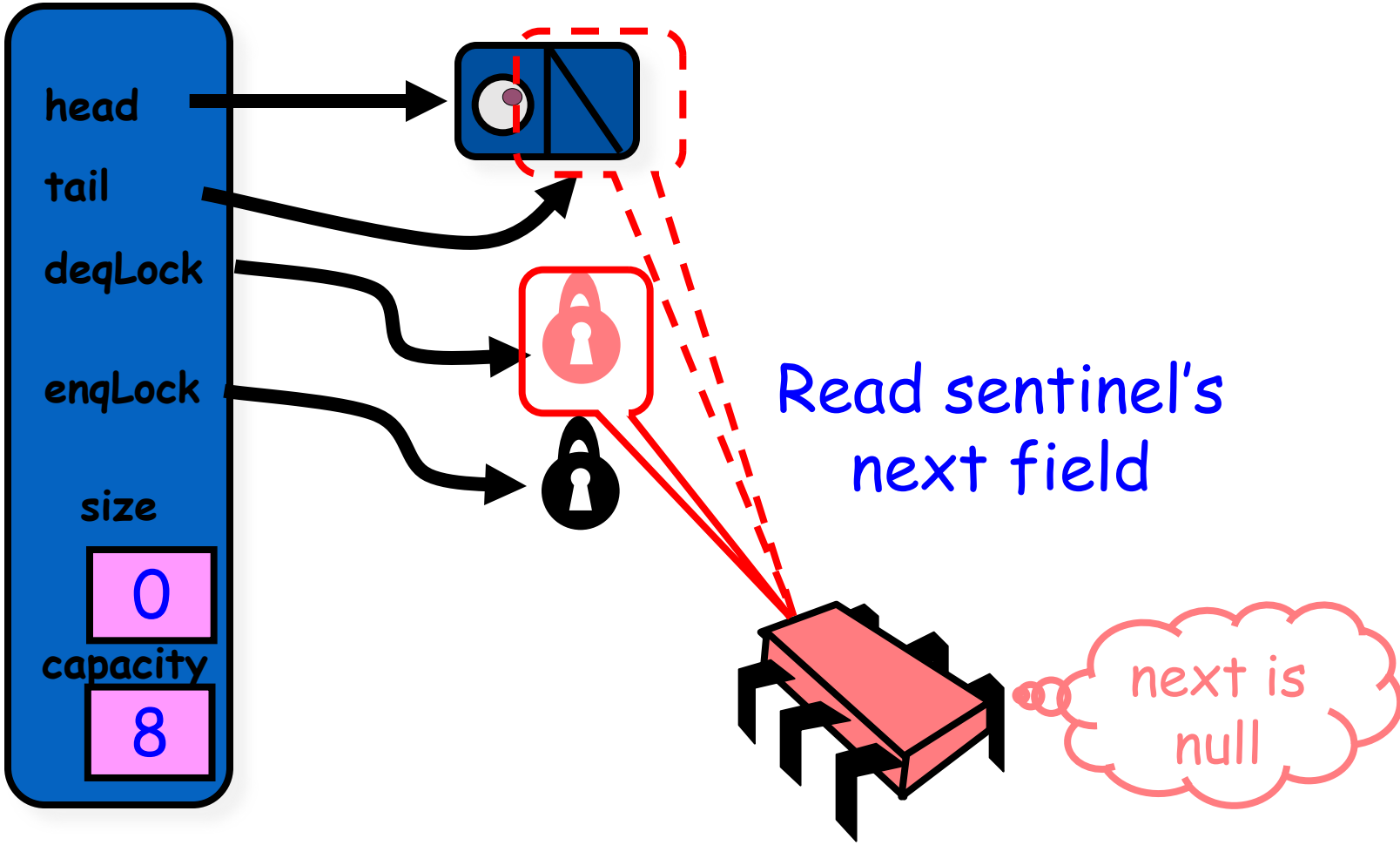
Bounded Queue: Dequeuer



Bounded Queue: Dequeuer



Bounded Queue: Dequeuer



Bounded Queue

```
1  public class BoundedQueue<T> {
2    ReentrantLock enqLock, deqLock;
3    Condition notEmptyCondition, notFullCondition;
4    AtomicInteger size;
5    Node head, tail;
6    int capacity;
7    public BoundedQueue(int _capacity) {
8      capacity = _capacity;
9      head = new Node(null);
10     tail = head;
11     size = new AtomicInteger(0);
12     enqLock = new ReentrantLock();
13     notFullCondition = enqLock.newCondition();
14     deqLock = new ReentrantLock();
15     notEmptyCondition = deqLock.newCondition();
16   }
```

Bounded Queue

Linearization point for enq()?

```
17    public void enq(T x) {
18      boolean mustWakeDequeuers = false;
19      enqLock.lock();
20      try {
21        while (size.get() == capacity)
22          notFullCondition.await();
23        Node e = new Node(x);
24        tail.next = e; tail = e;
25        if (size.getAndIncrement() == 0)
26          mustWakeDequeuers = true;
27      } finally {
28        enqLock.unlock();
29      }
30      if (mustWakeDequeuers) {
31        deqLock.lock();
32        try {
33          notEmptyCondition.signalAll();
34        } finally {
35          deqLock.unlock();
36        }
37      }
38    }
```

```
39    public T deq() {
40      T result;
41      boolean mustWakeEnqueuers = true;
42      deqLock.lock();
43      try {
44        while (size.get() == 0)
45          notEmptyCondition.await();
46        result = head.next.value;
47        head = head.next;
48        if (size.getAndDecrement() == capacity) {
49          mustWakeEnqueuers = true;
50        }
51      } finally {
52        deqLock.unlock();
53      }
54      if (mustWakeEnqueuers) {
55        enqLock.lock();
56        try {
57          notFullCondition.signalAll();
58        } finally {
59          enqLock.unlock();
60        }
61      }
62      return result;
63    }
```

Bounded Queue

```
17    public void enq(T x) {
18      boolean mustWakeDequeuers = false;
19      enqLock.lock();
20      try {
21        while (size.get() == capacity)
22          notFullCondition.await();
23        Node e = new Node(x);
24        tail.next = e; tail = e;
25        if (size.getAndIncrement() == 0)
26          mustWakeDequeuers = true;
27      } finally {
28        enqLock.unlock();
29      }
30      if (mustWakeDequeuers) {
31        deqLock.lock();
32        try {
33          notEmptyCondition.signalAll();
34        } finally {
35          deqLock.unlock();
36        }
37      }
38    }
```

Logical addition

```
39    public T deq() {
40      T result;
41      boolean mustWakeEnqueuers = true;
42      deqLock.lock();
43      try {
44        while (size.get() == 0)
45          notEmptyCondition.await();
46        result = head.next.value;
47        head = head.next;
48        if (size.getAndDecrement() == capacity) {
49          mustWakeEnqueuers = true;
50        }
51      } finally {
52        deqLock.unlock();
53      }
54      if (mustWakeEnqueuers) {
55        enqLock.lock();
56        try {
57          notFullCondition.signalAll();
58        } finally {
59          enqLock.unlock();
60        }
61      }
62      return result;
63    }
```

Bounded Queue

```
17    public void enq(T x) {
18      boolean mustWakeDequeuers = false;
19      enqLock.lock();
20      try {
21        while (size.get() == capacity)
22          notFullCondition.await();
23        Node e = new Node(x);
24        tail.next = e; tail = e;
25        if (size.getAndIncrement() == 0)
26          mustWakeDequeuers = true;
27      } finally {
28        enqLock.unlock();
29      }
30      if (mustWakeDequeuers) {
31        deqLock.lock();
32        try {
33          notEmptyCondition.signalAll();
34        } finally {
35          deqLock.unlock();
36        }
37      }
38    }
```

```
39    public T deq() {
40      T result;
41      boolean mustWakeEnqueuers = true;
42      deqLock.lock();
43      try {
44        while (size.get() == 0)
45          notEmptyCondition.await();
46        result = head.next.value;
47        head = head.next;
48        if (size.getAndDecrement() == capacity) {
49          mustWakeEnqueuers = true;
50        }
51      } finally {
52        deqLock.unlock();
53      }
54      if (mustWakeEnqueuers) {
55        enqLock.lock();
56        try {
57          notFullCondition.signalAll();
58        } finally {
59          enqLock.unlock();
60        }
61      }
62      return result;
63    }
```

Dequeuers won't see logical addition

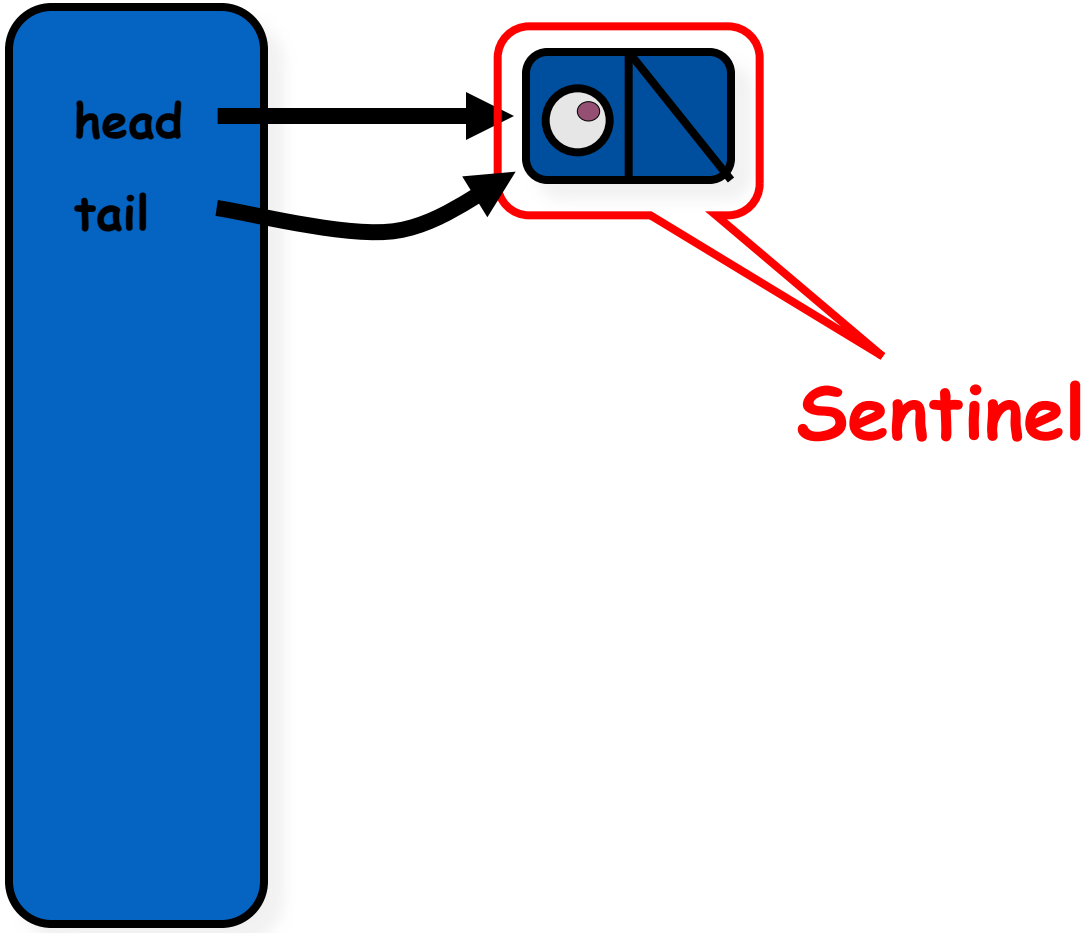
Unbounded (Total) Lock-Free Queue

- enq() always succeeds
- deq() throws exception (or returns false) on empty queue

- Rely on atomics (CAS) instead of locks

```
bool cas(pointer p, int old, int new) {
    if (*p ≠ old) { return false; }
    *p ← new;
    return true;
}
```

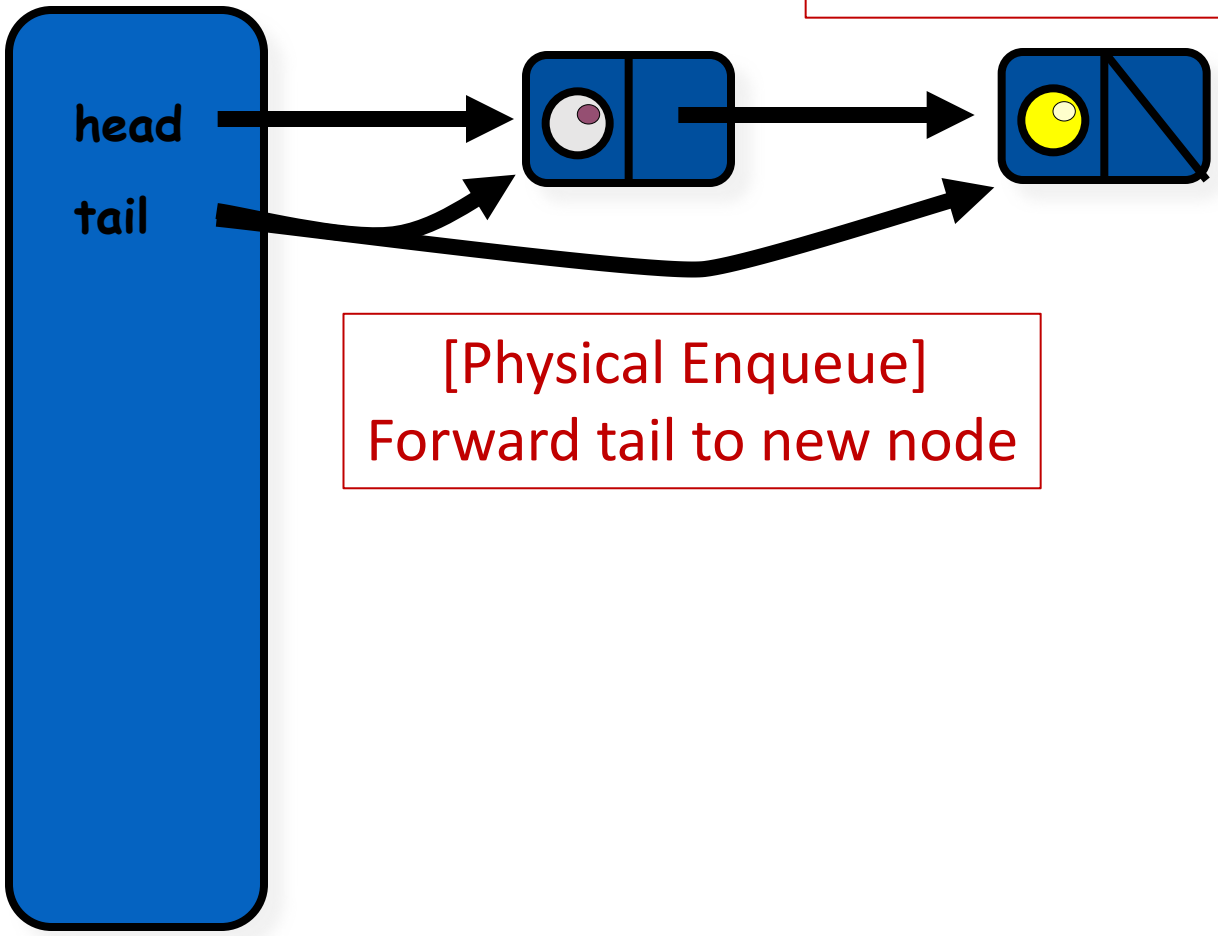
Lock-Free Queue



Lock-Free Queue: Enqueue

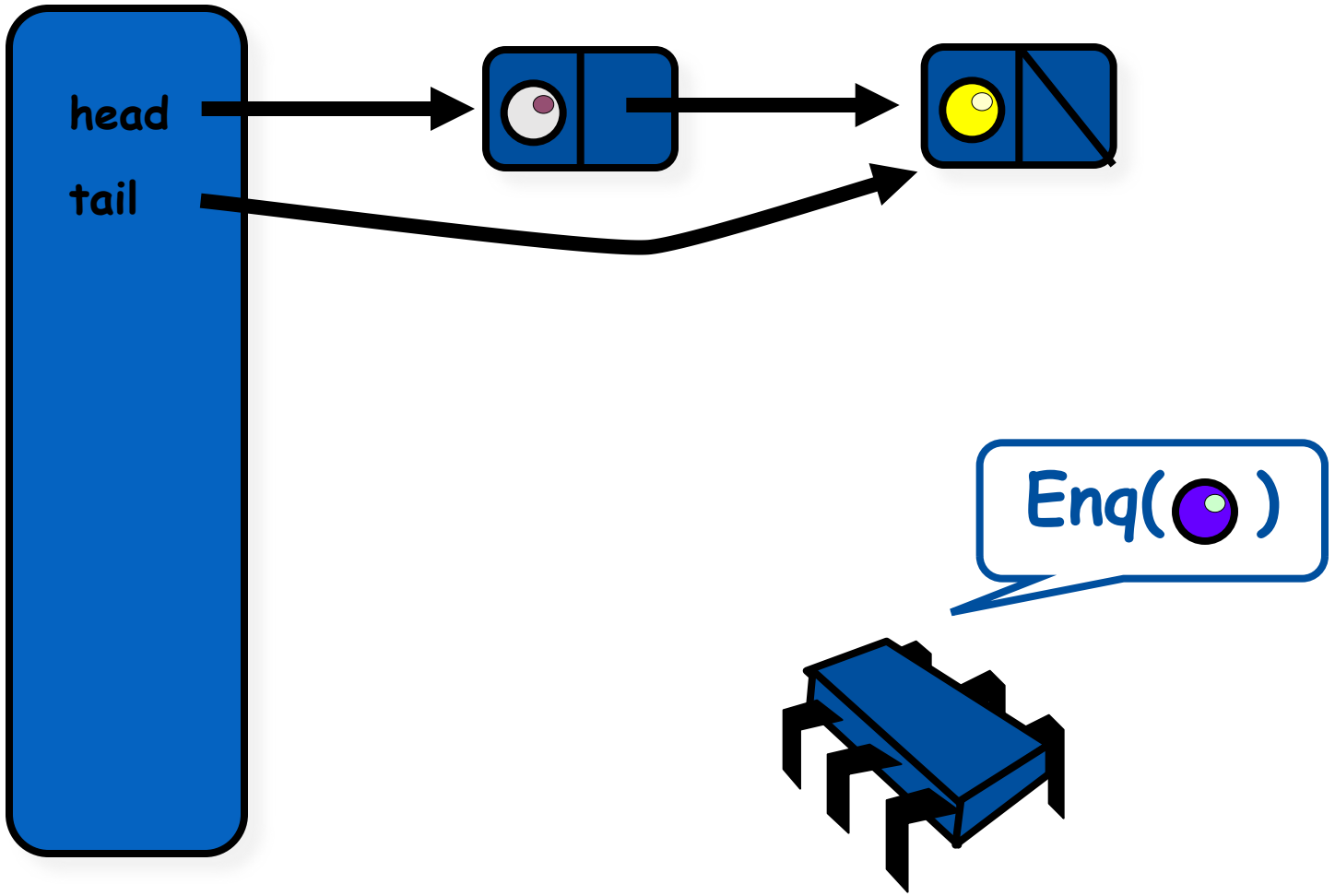
Steps for Enqueue?

[Logical Enqueue]
Point tail.next to new node

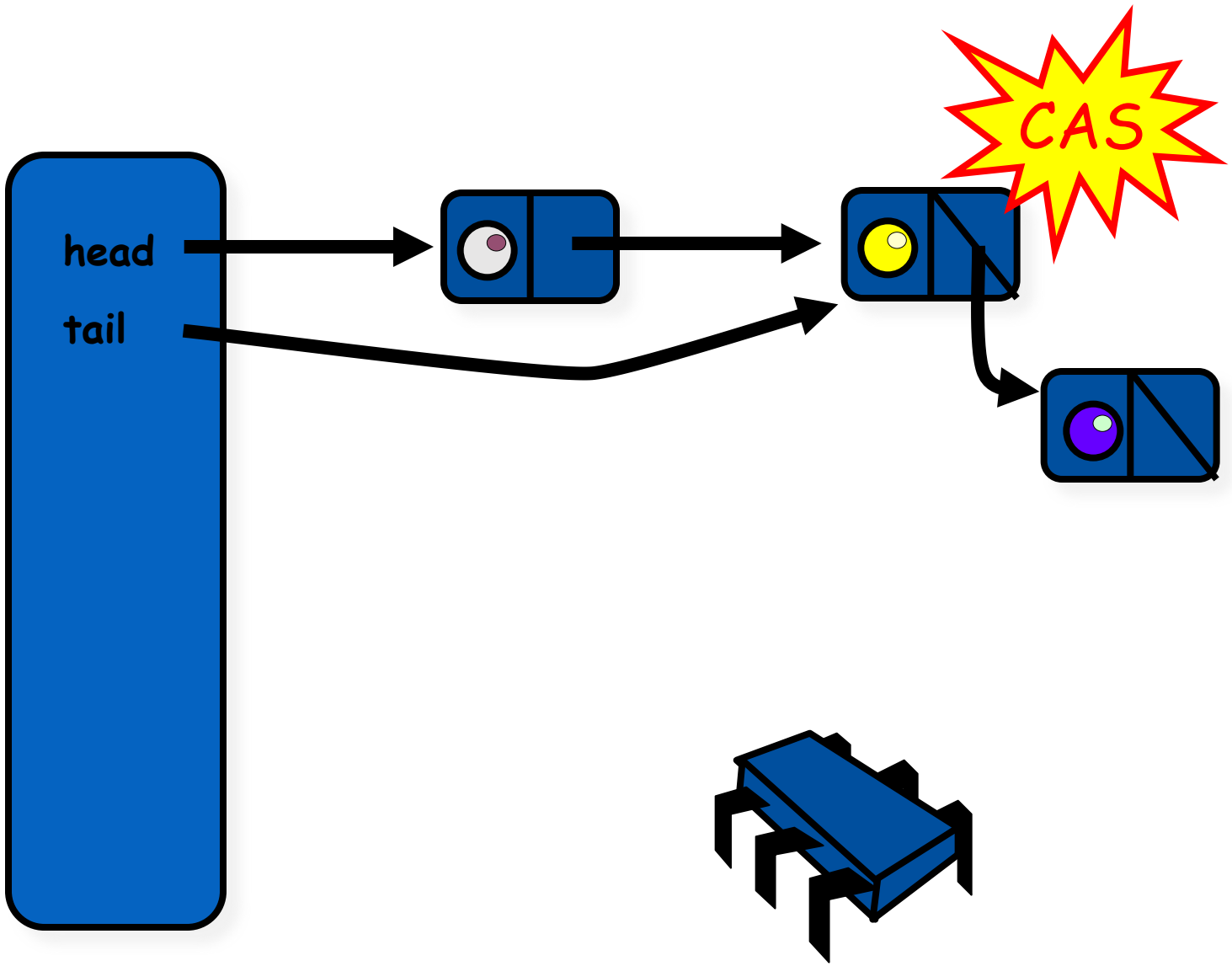


[Physical Enqueue]
Forward tail to new node

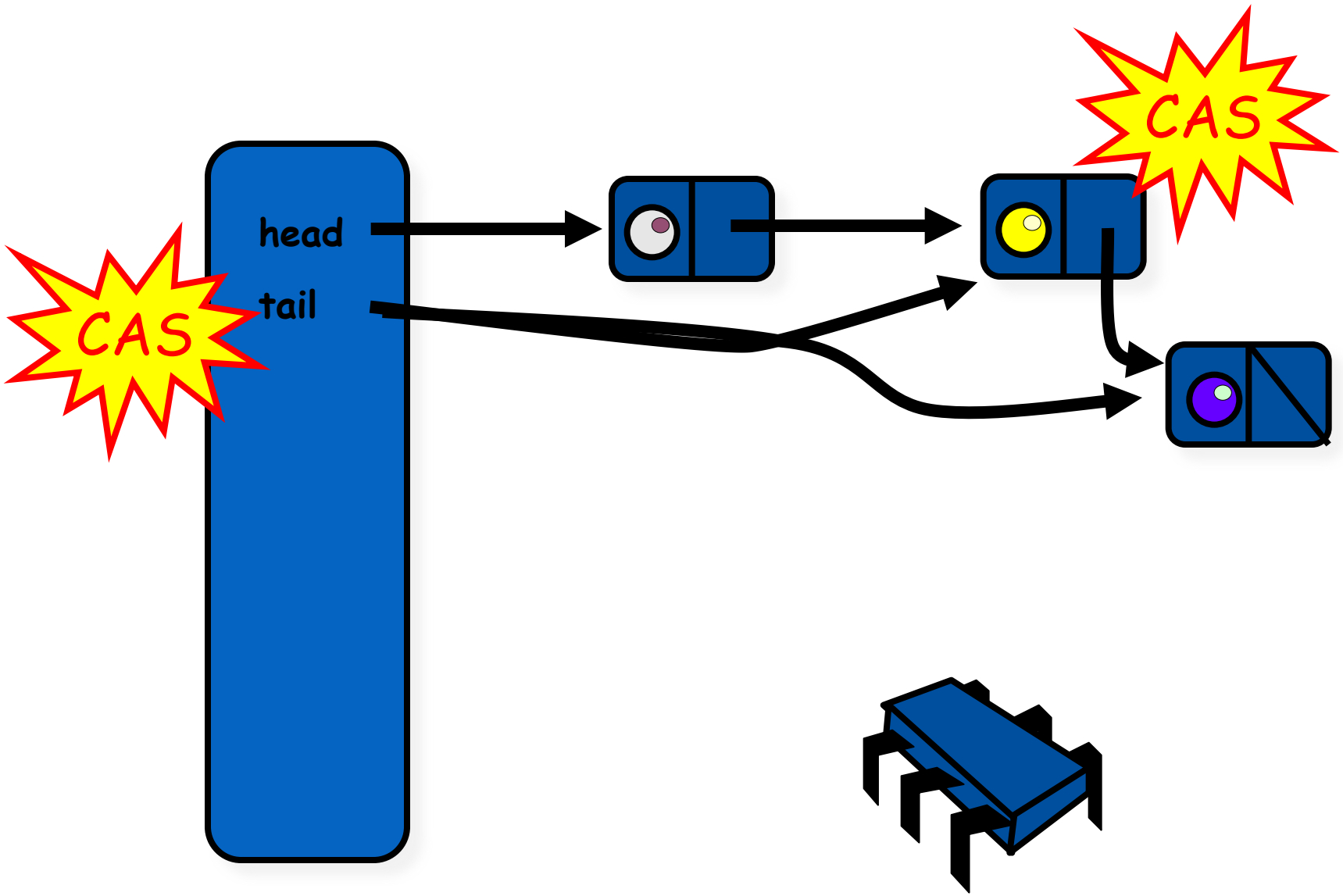
Lock-Free Queue: Enqueue



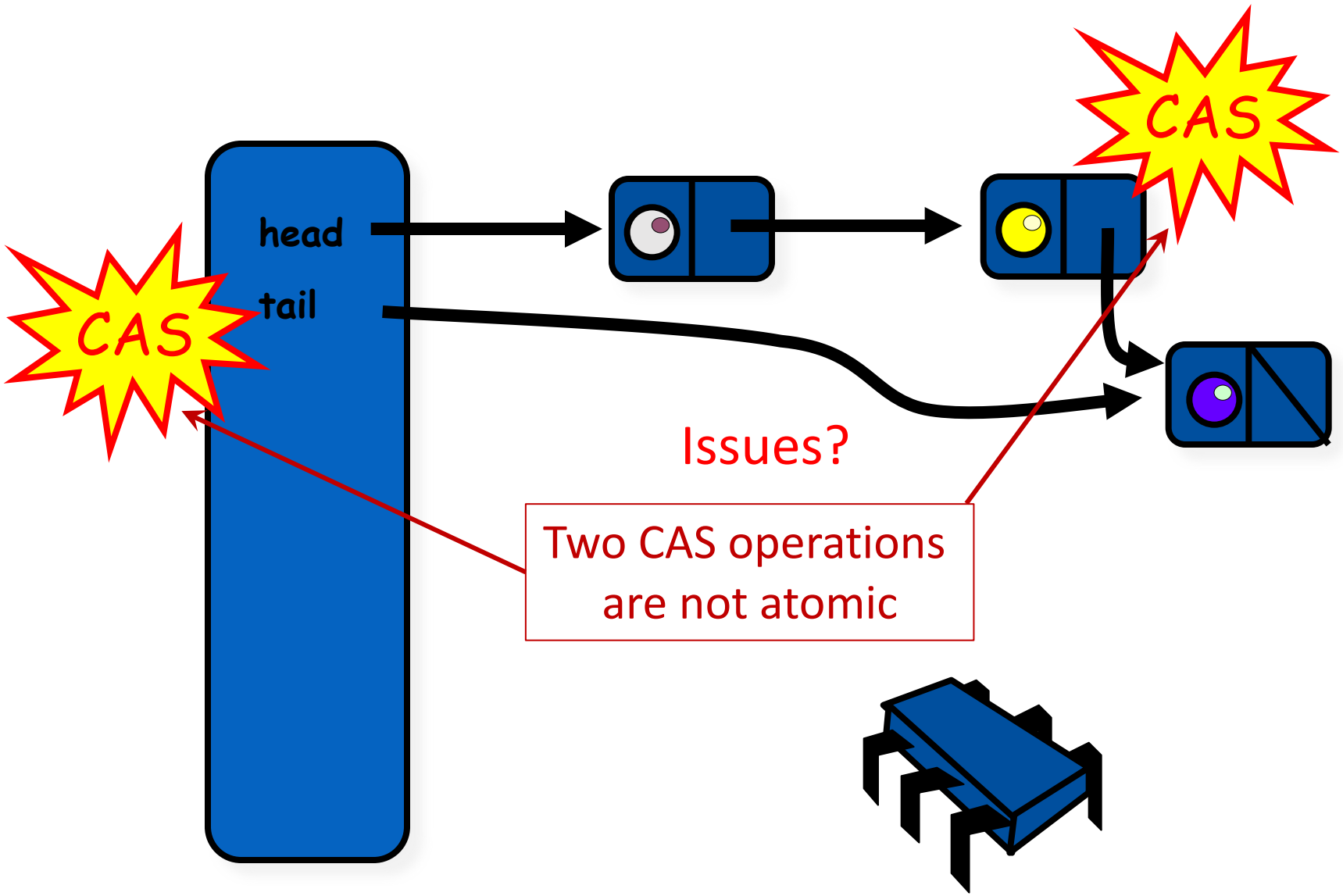
Lock-Free Queue: Logical Enqueue



Lock-Free Queue: Physical Enqueue



Lock-Free Queue: Physical Enqueue

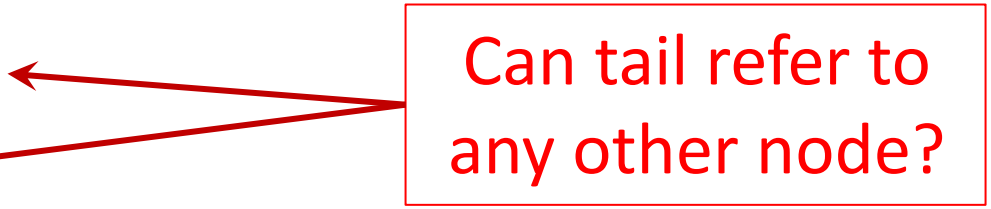


Lock-Free Queue: Enqueue

- Two CAS operations are not atomic
- When logical enqueue CAS fails
  - Restart the enqueue process
- When physical enqueue CAS fails
  - Ignore it – Why?
  - Similar to lock-free set implementation

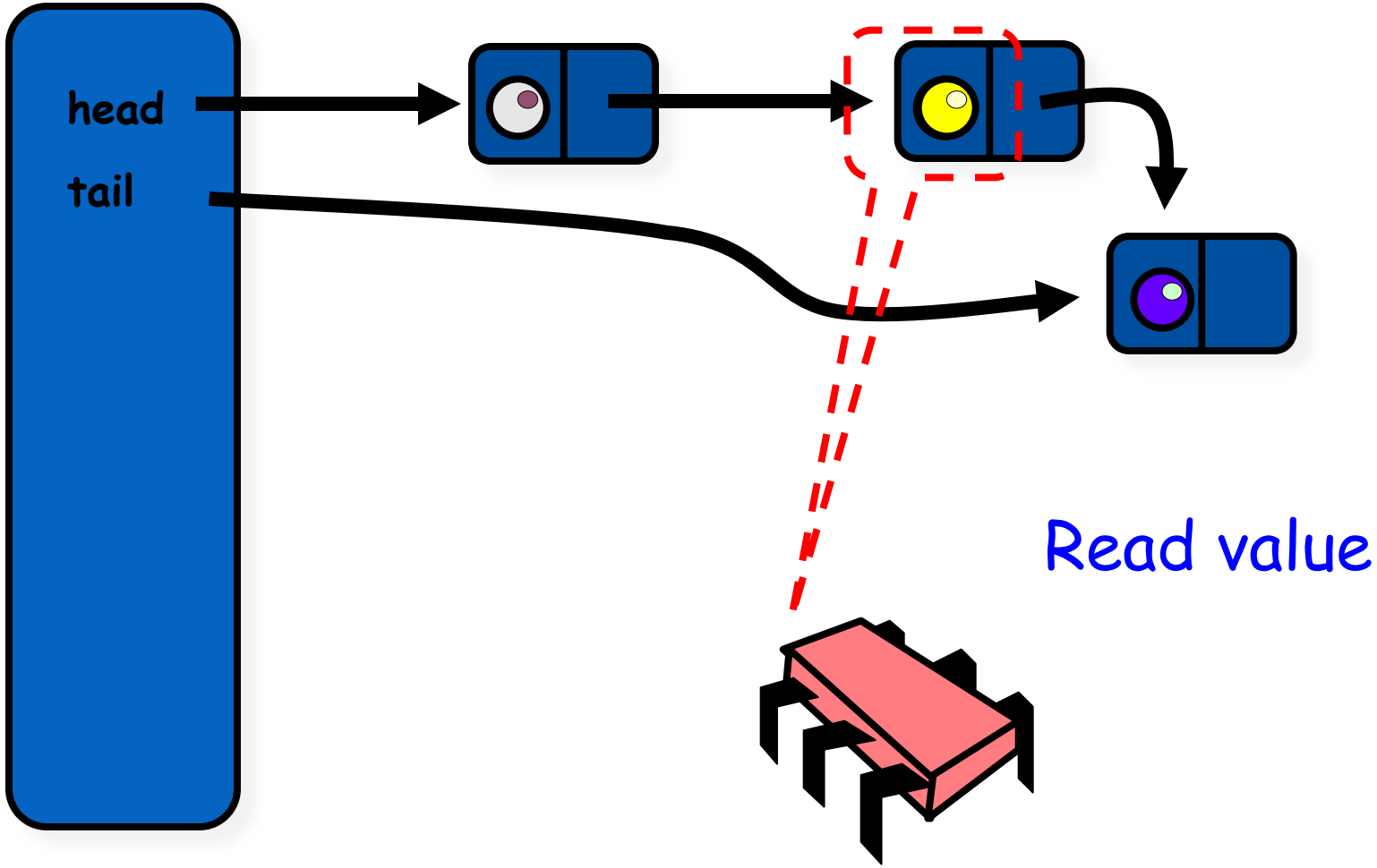
Lock-Free Queue

- The tail can refer to either:
  - Actual last node, or
  - Penultimate node

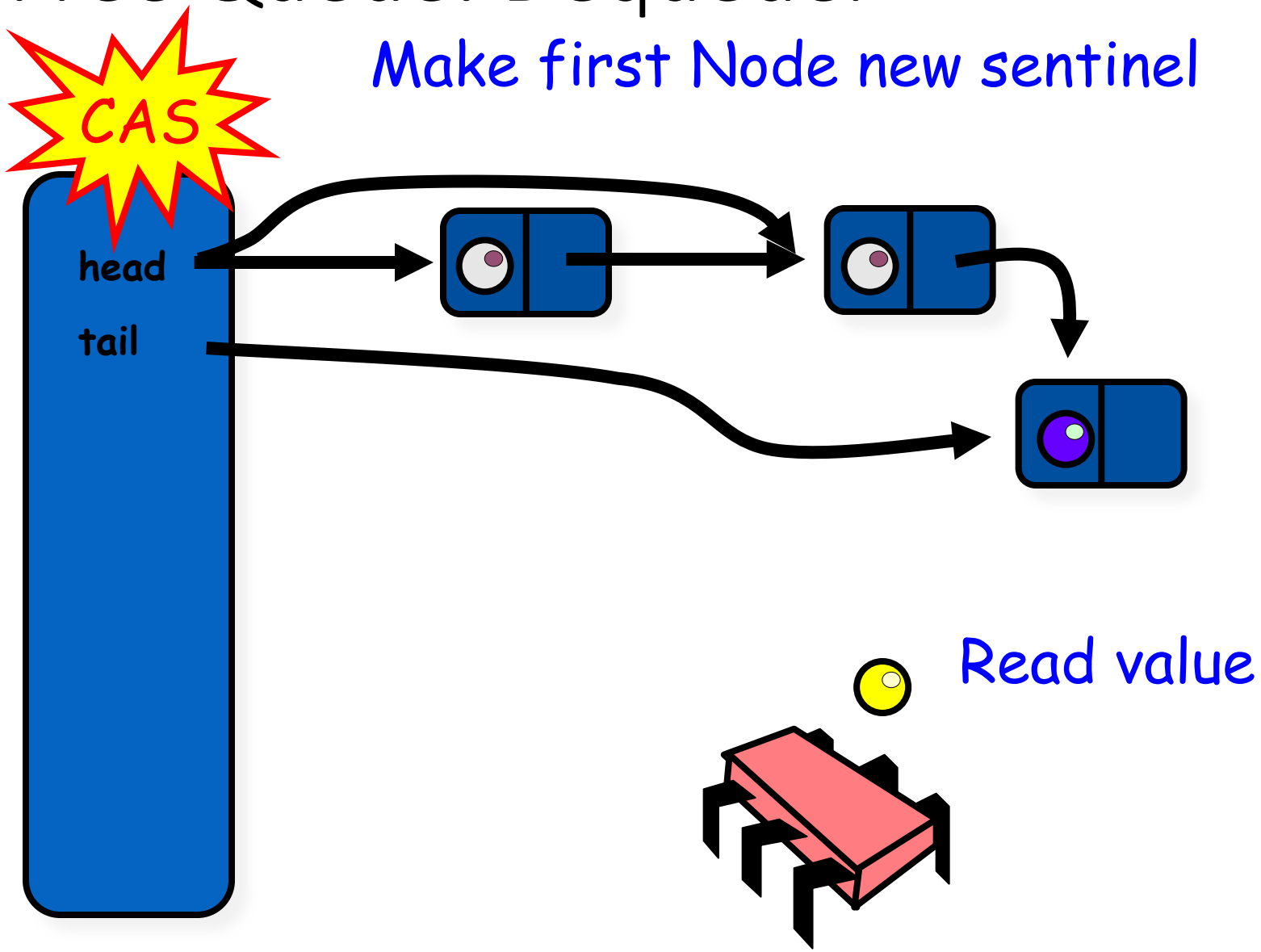


- What to do when we find a trailing tail?
- Stop and help fix it
  - CAS queue's tail field to tail.next

Lock-Free Queue: Dequeuer



Lock-Free Queue: Dequeuer



Unbounded (Total) Lock-Free Queue

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

```
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;
      }
    }
  }
}
```

Unbounded (Total) Lock-Free Queue

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

Logical enqueue

```
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;
      }
    }
  }
}
```

Unbounded (Total) Lock-Free Queue

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

Physical enqueue

```
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;
      }
    }
  }
}
```

Unbounded (Total) Lock-Free Queue

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

Forward tail

```
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;
      }
    }
  }
}
```

Unbounded (Total) Lock-Free Queue

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

```
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;
      }
    }
  }
}
```

Forward tail

Unbounded (Total) Lock-Free Queue

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

Linearization Point?

```
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;
      }
    }
  }
}
```

Unbounded (Total) Lock-Free Queue

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

```
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;
      }
    }
  }
}
```

Enqueuing thread OR helper thread succeeds on CAS to swing tail

Unbounded (Total) Lock-Free Queue

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

```
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;
      }
    }
  }
}
```

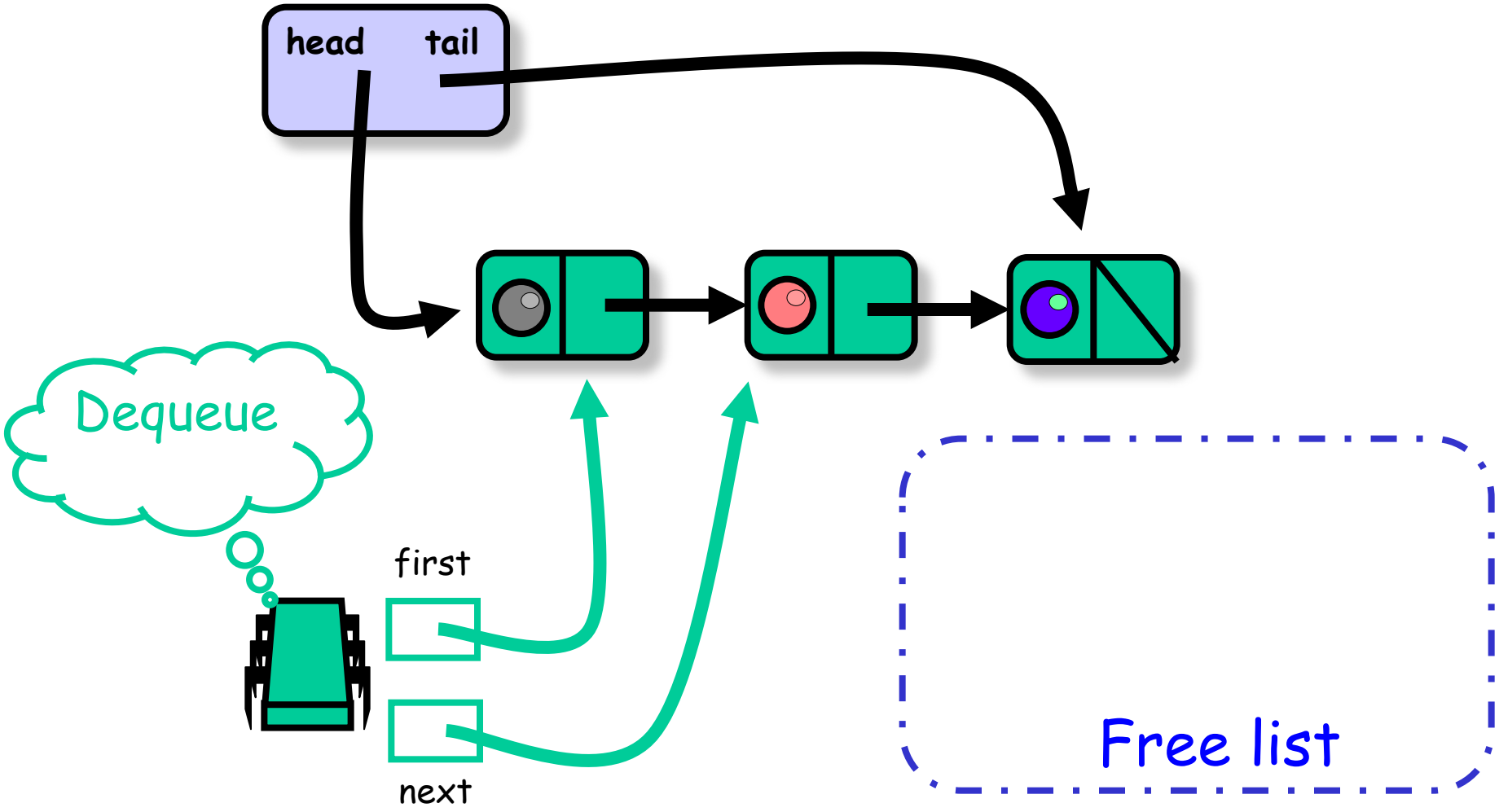
successful CAS to swing head

Memory Management

- So far we relied on GC to recycle dequeued nodes
- Let's do our own memory management
- Simplest design
  - Each thread has its own private free list
  - Enqueue(): Allocate nodes for using thread's free list (if private list is empty, call malloc/new)
  - Dequeue(): Deallocated nodes go back to thread's free list

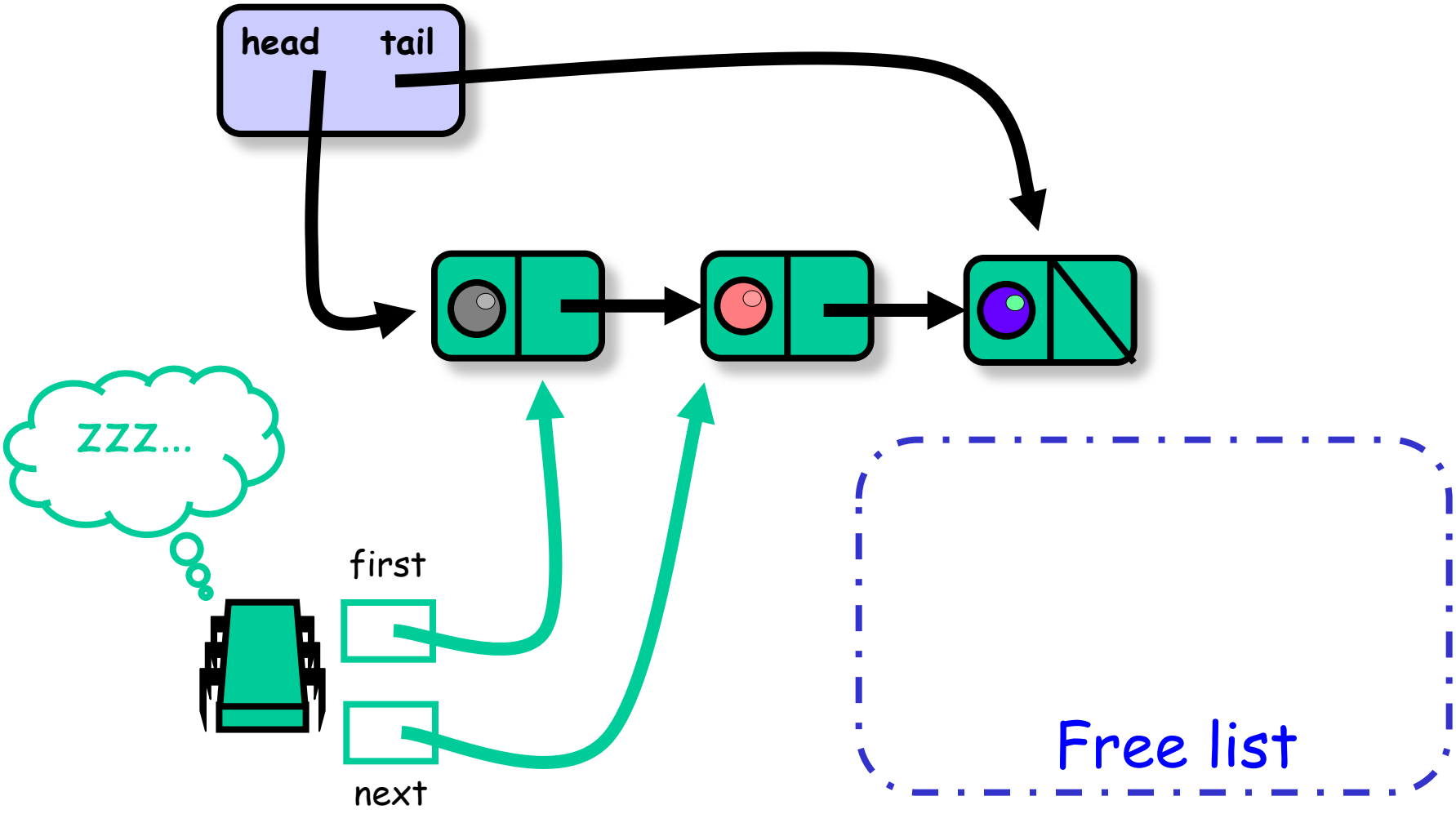
Want to redirect head from grey to red

```
if (head.compareAndSet(first, next))
  return value;
```



Want to redirect head from grey to red

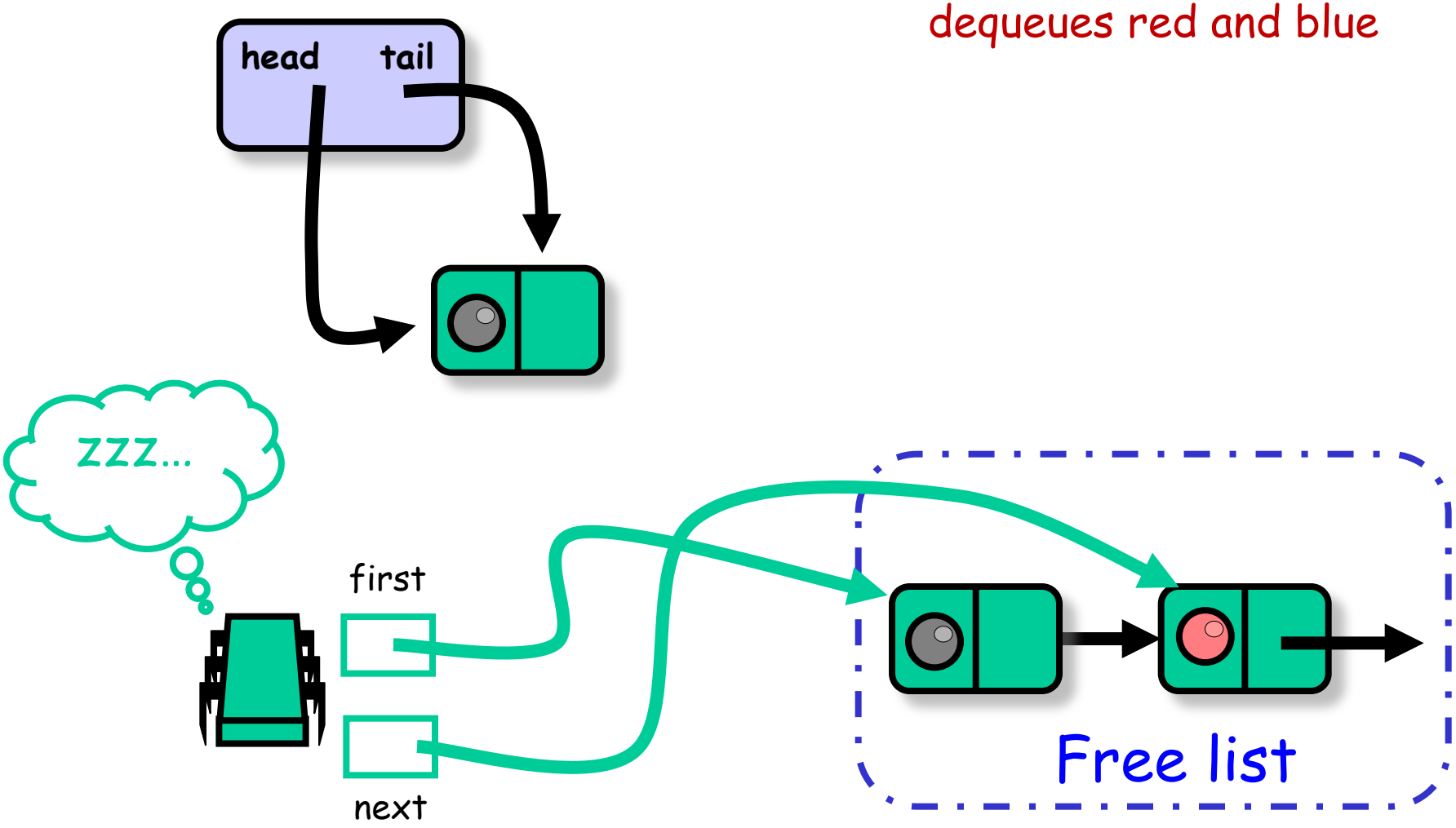
```
if (head.compareAndSet(first, next))
  return value;
```



Want to redirect head from grey to red

```
if (head.compareAndSet(first, next))
  return value;
```

Another thread dequeues red and blue

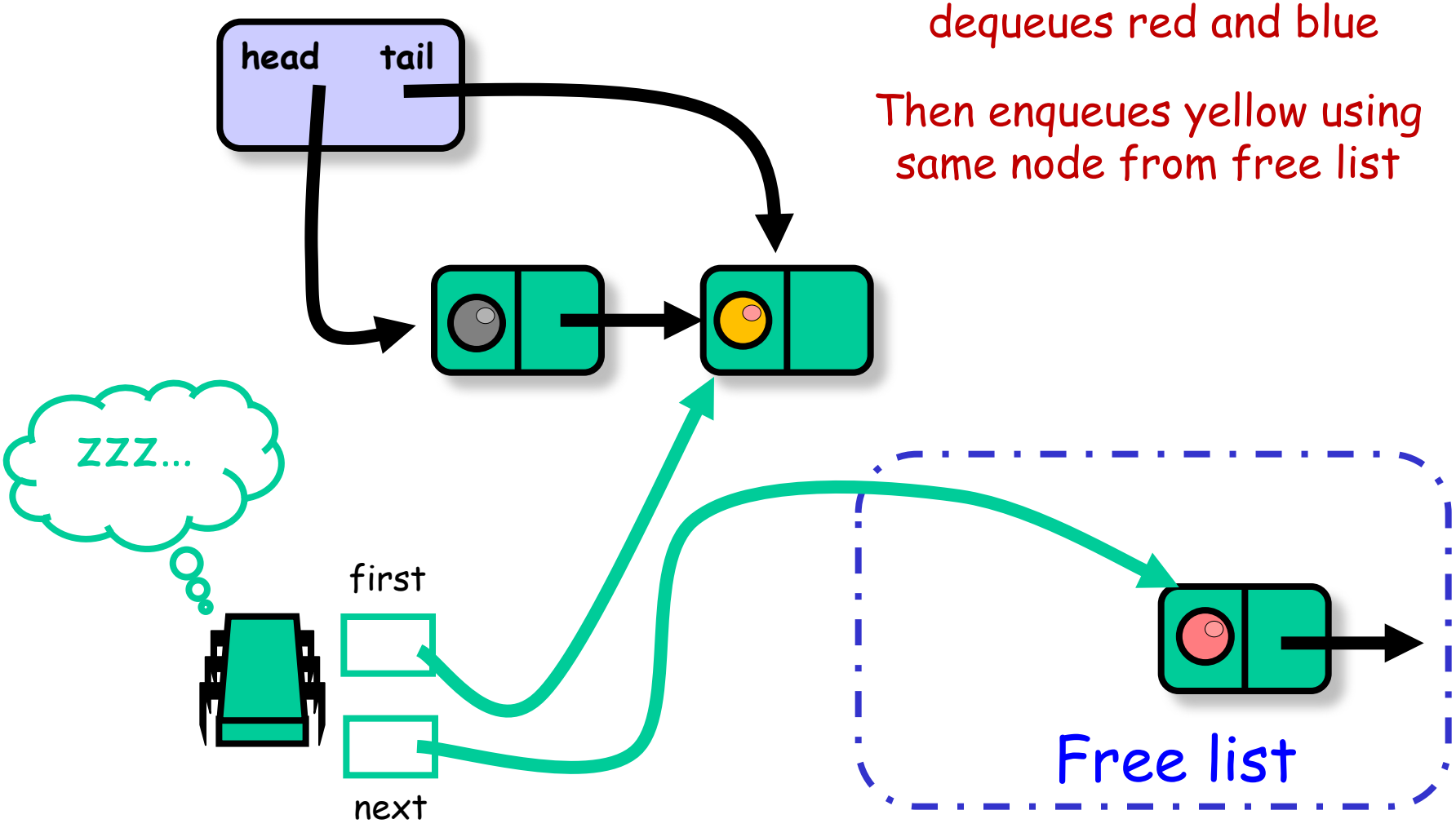


Want to redirect head from grey to red

```
if (head.compareAndSet(first, next))
  return value;
```

Another thread dequeues red and blue

Then enqueues yellow using same node from free list



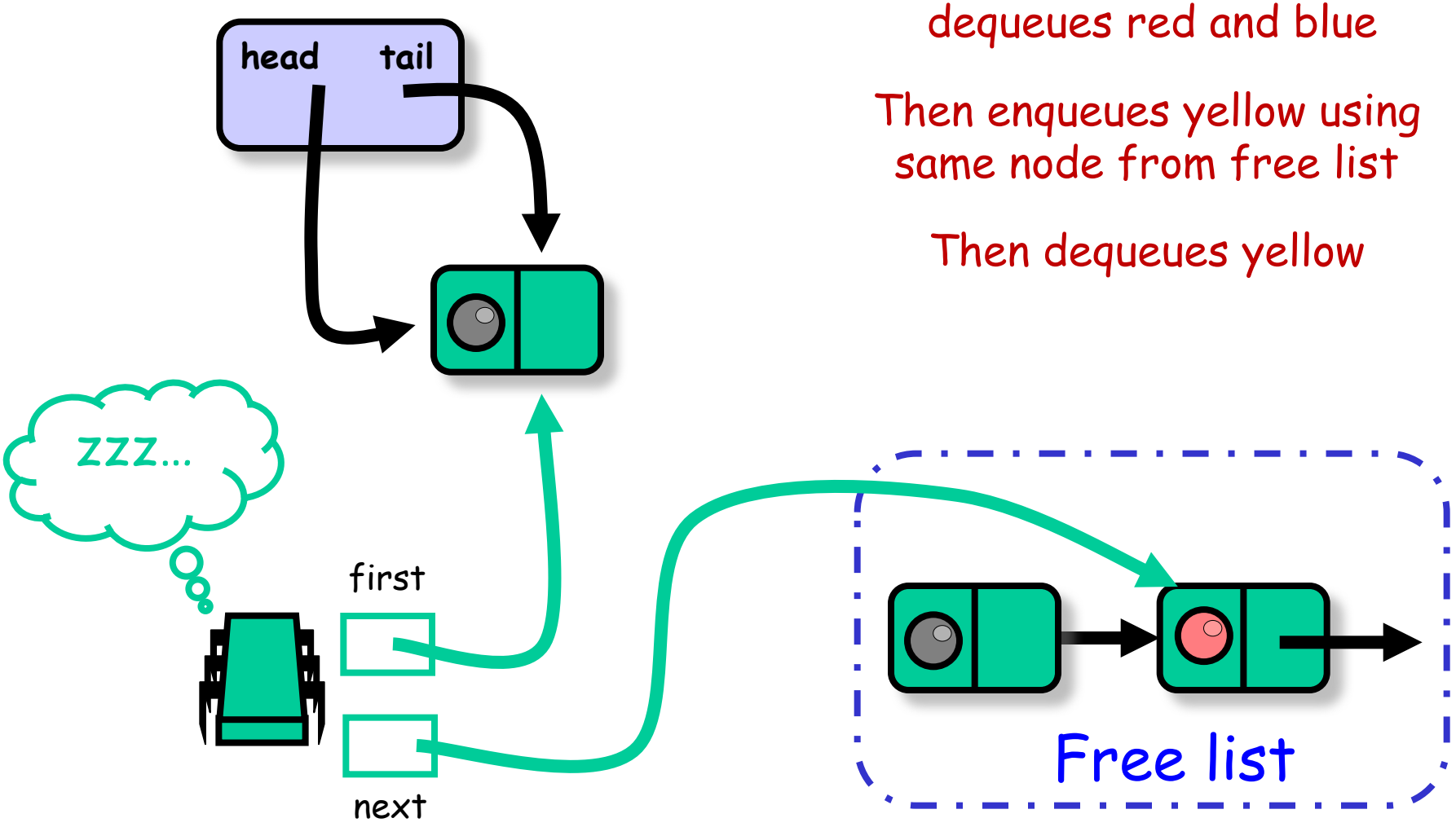
Want to redirect head from grey to red

```
if (head.compareAndSet(first, next))
  return value;
```

Another thread dequeues red and blue

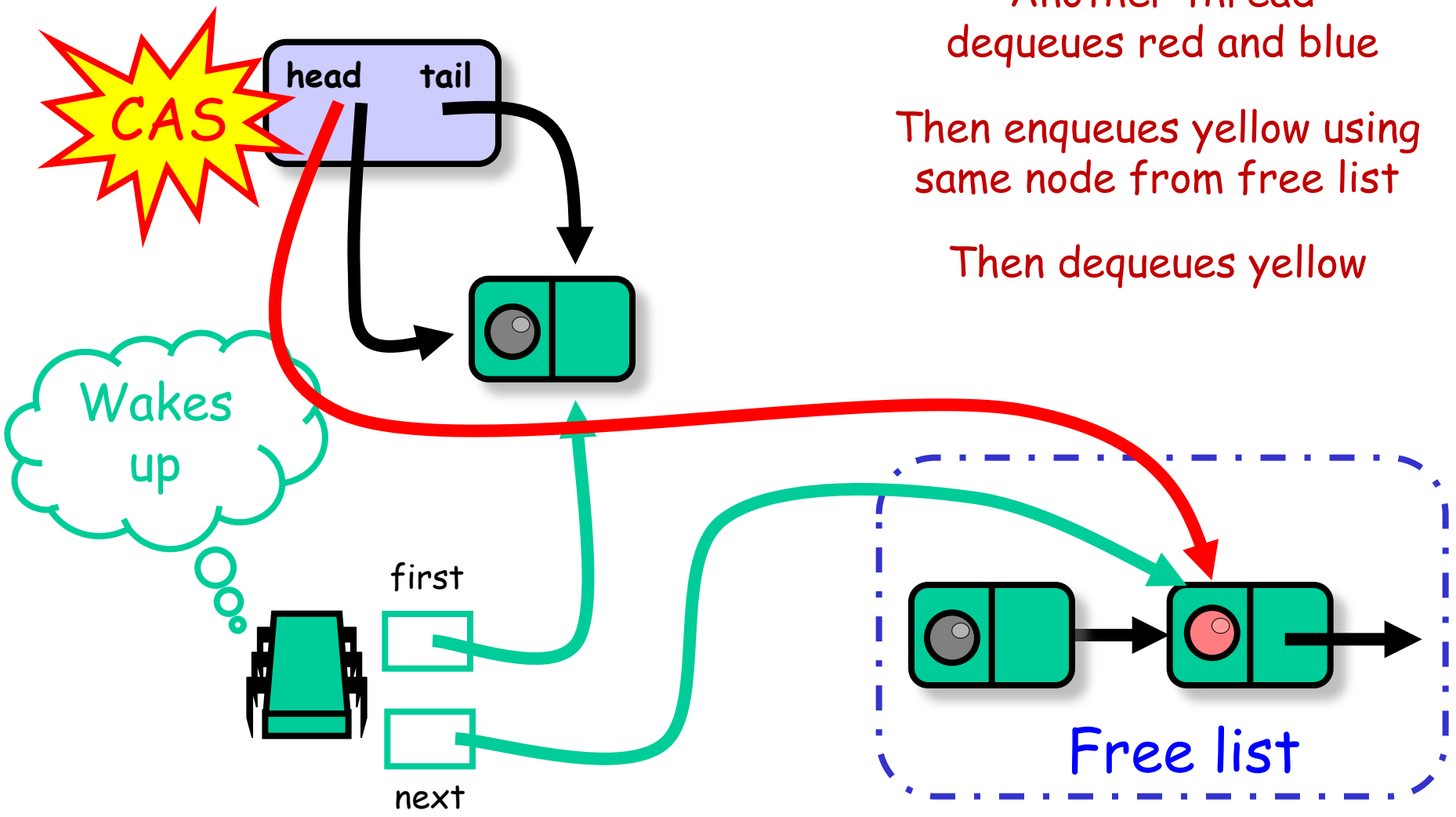
Then enqueues yellow using same node from free list

Then dequeues yellow



Want to redirect head from grey to red

```
if (head.compareAndSet(first, next))
  return value;
```

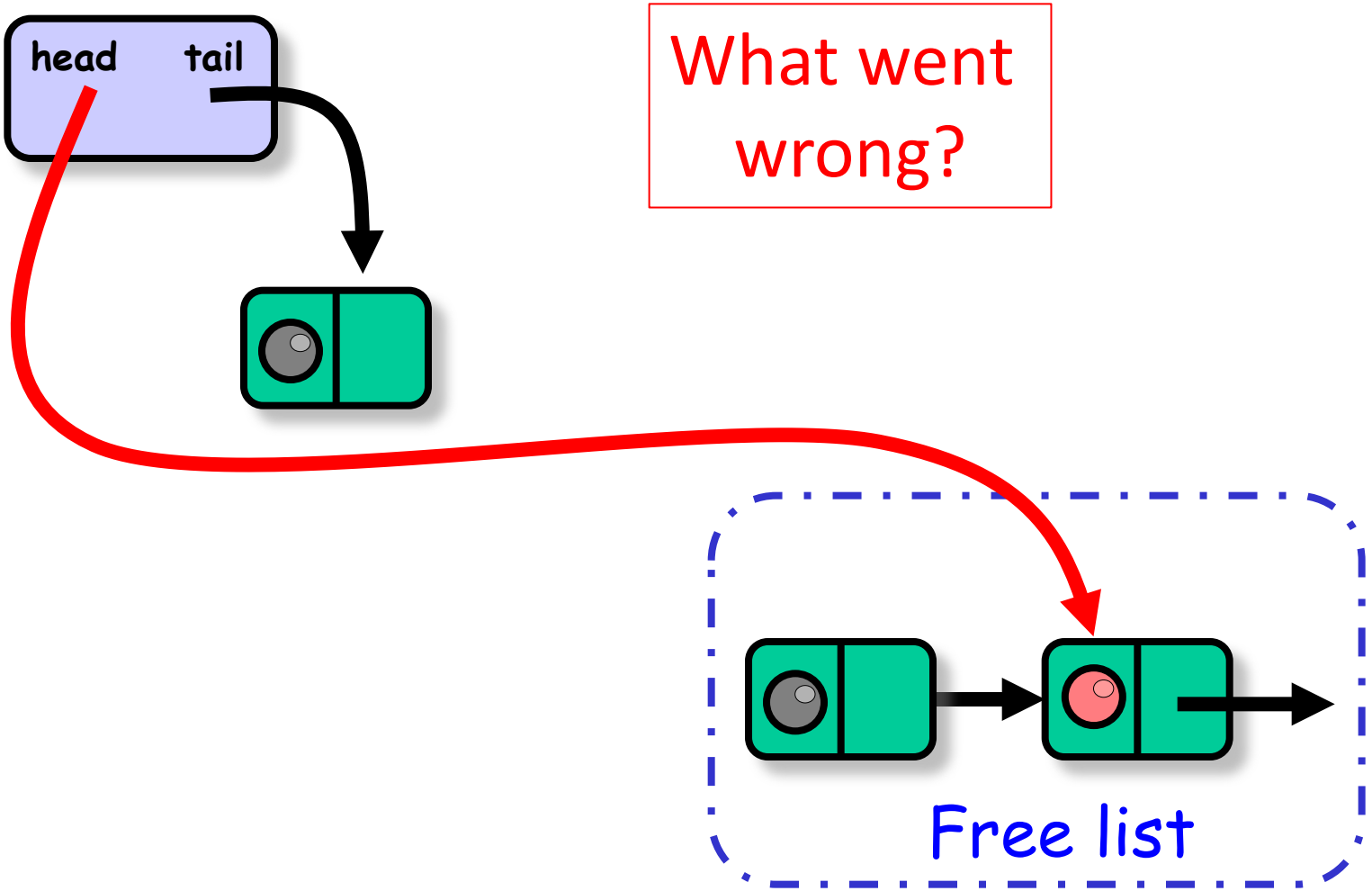


Another thread dequeues red and blue

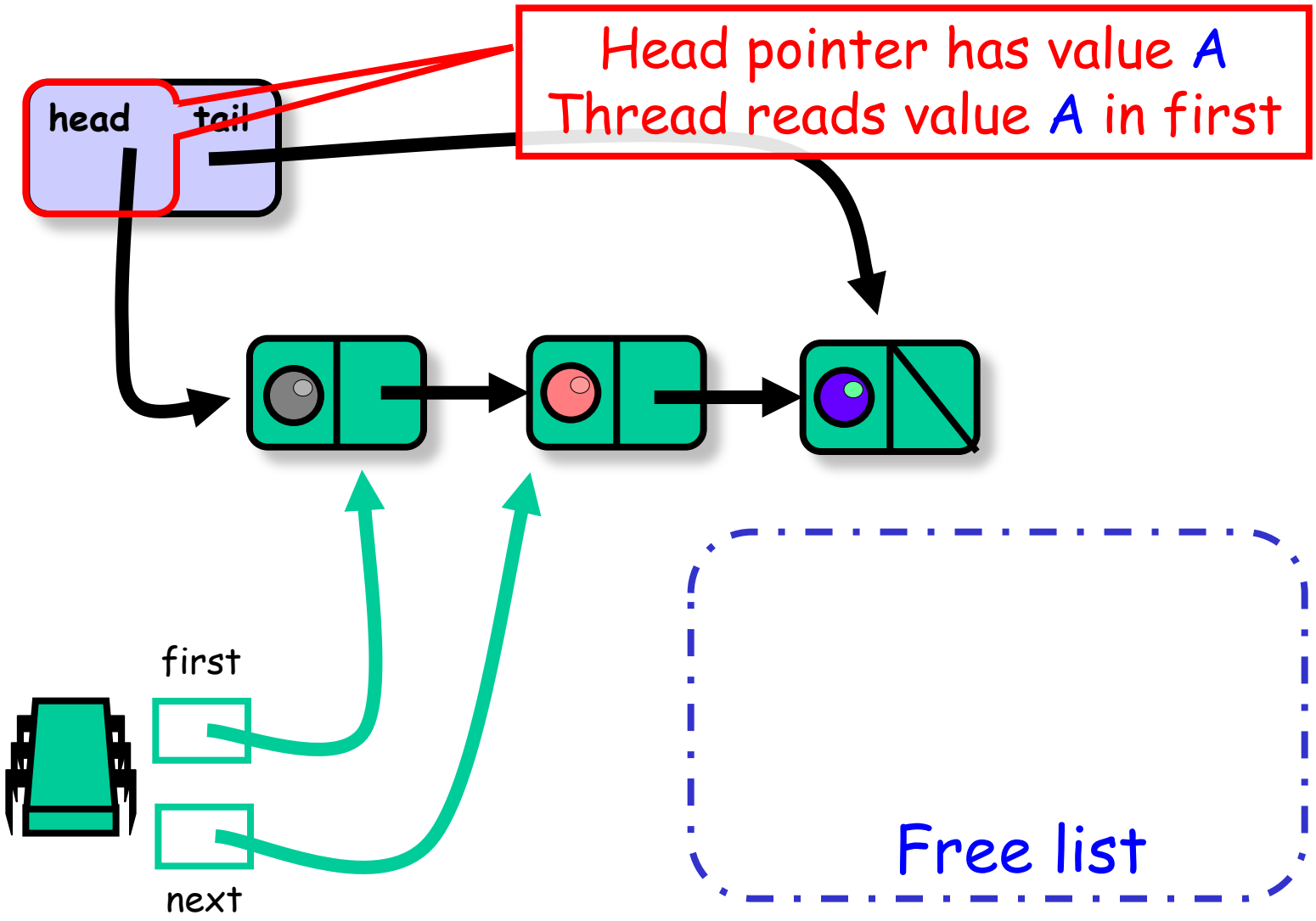
Then enqueues yellow using same node from free list

Then dequeues yellow

```
if (head.compareAndSet(first, next))
  return value;
```

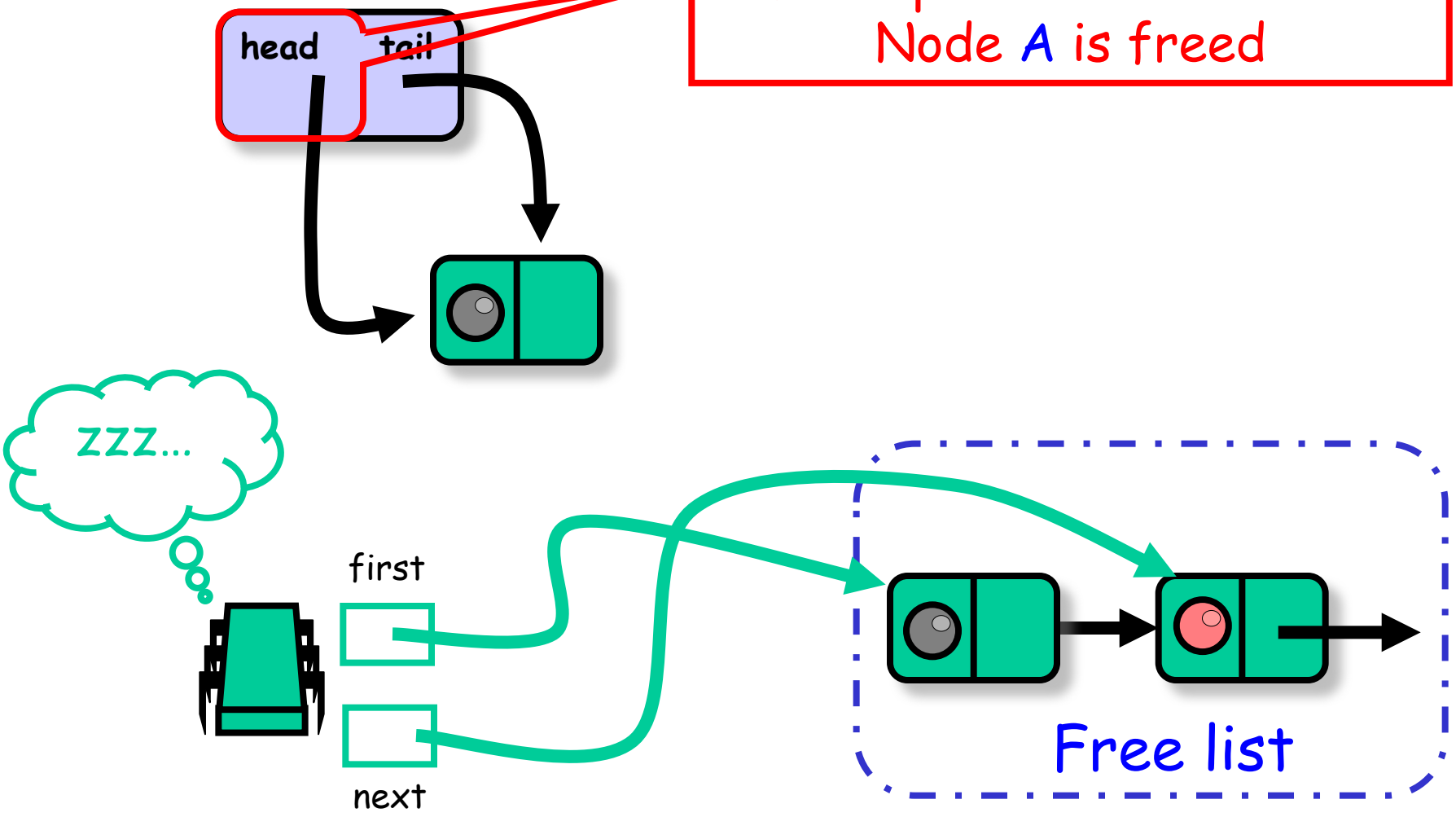


```
if (head.compareAndSet(first, next))
  return value;
```



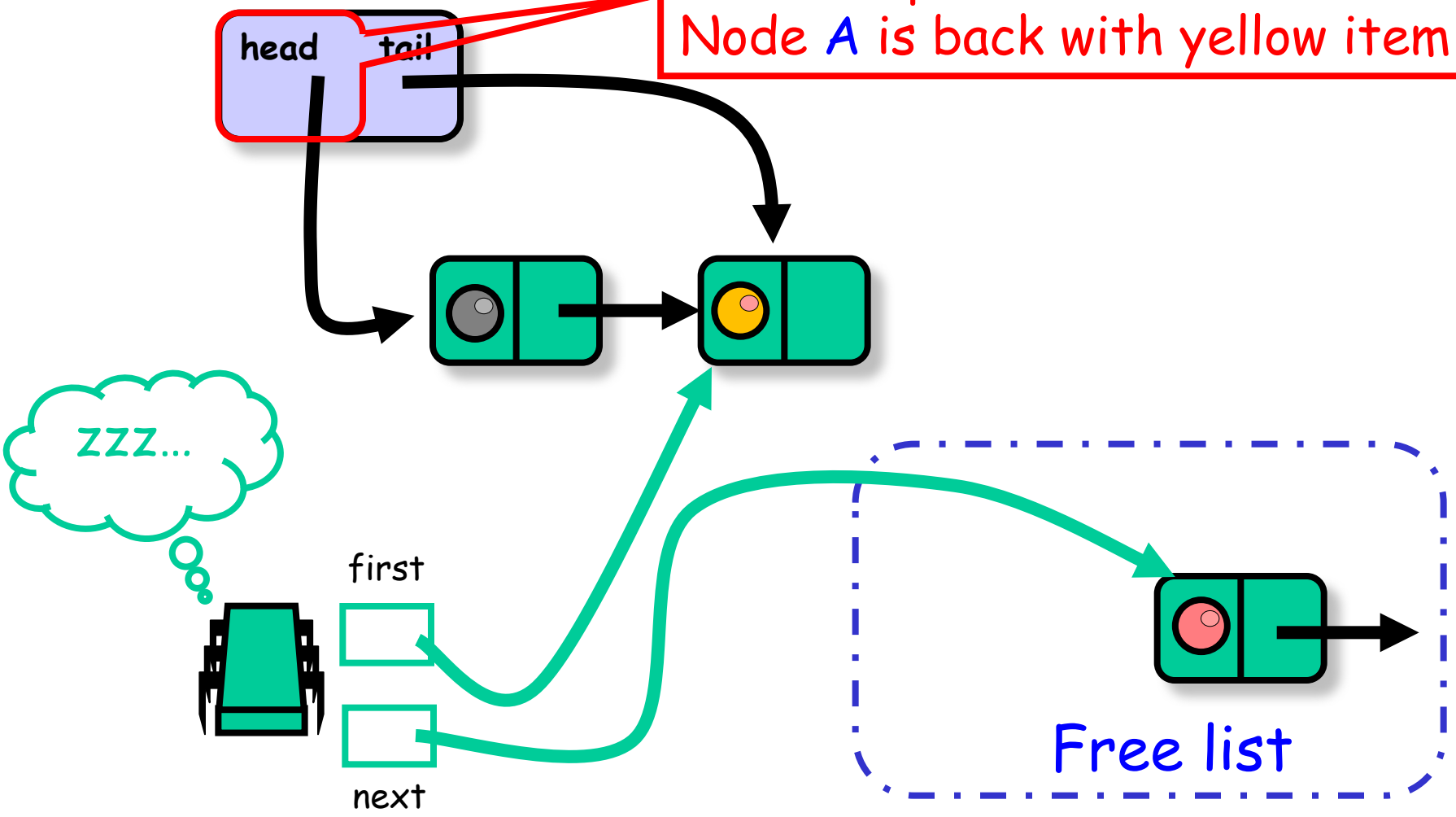
```
if (head.compareAndSet(first, next))
  return value;
```

Head pointer has value B
Node A is freed



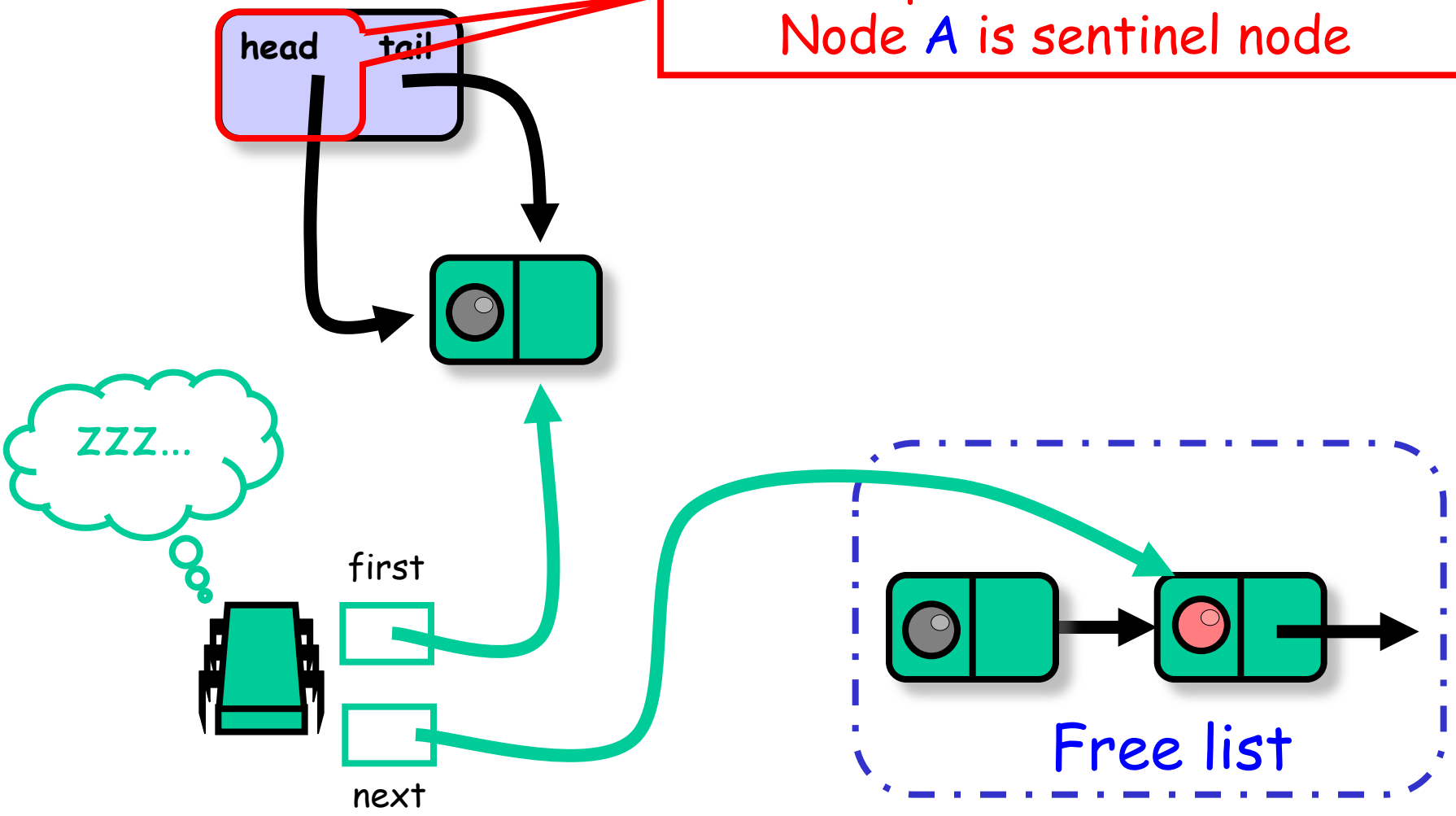
```
if (head.compareAndSet(first, next))
  return value;
```

Head pointer has value B
Node A is back with yellow item



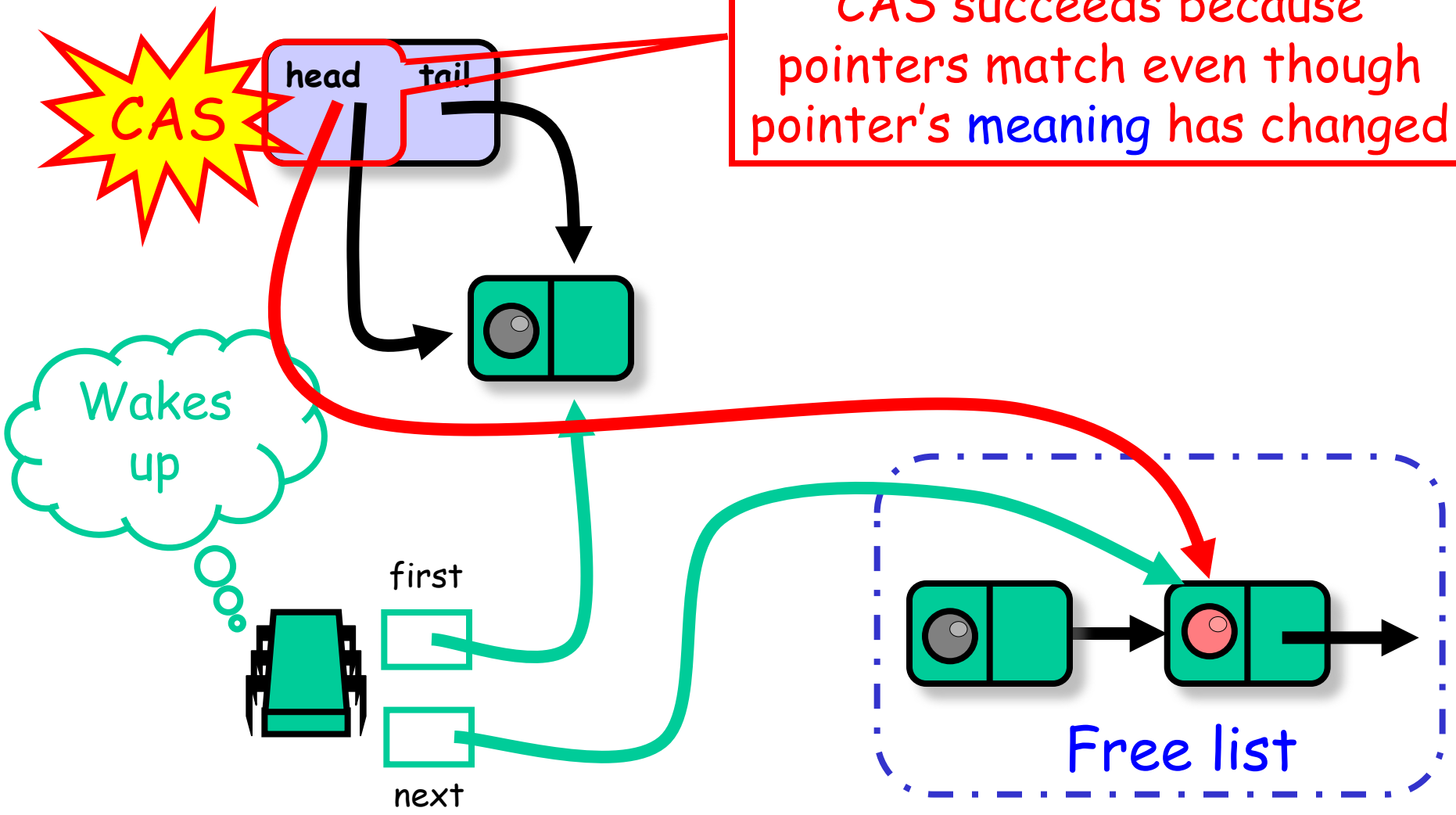
```
if (head.compareAndSet(first, next))
  return value;
```

Head pointer has value A
Node A is sentinel node



```
if (head.compareAndSet(first, next))
  return value;
```

CAS succeeds because pointers match even though pointer's meaning has changed



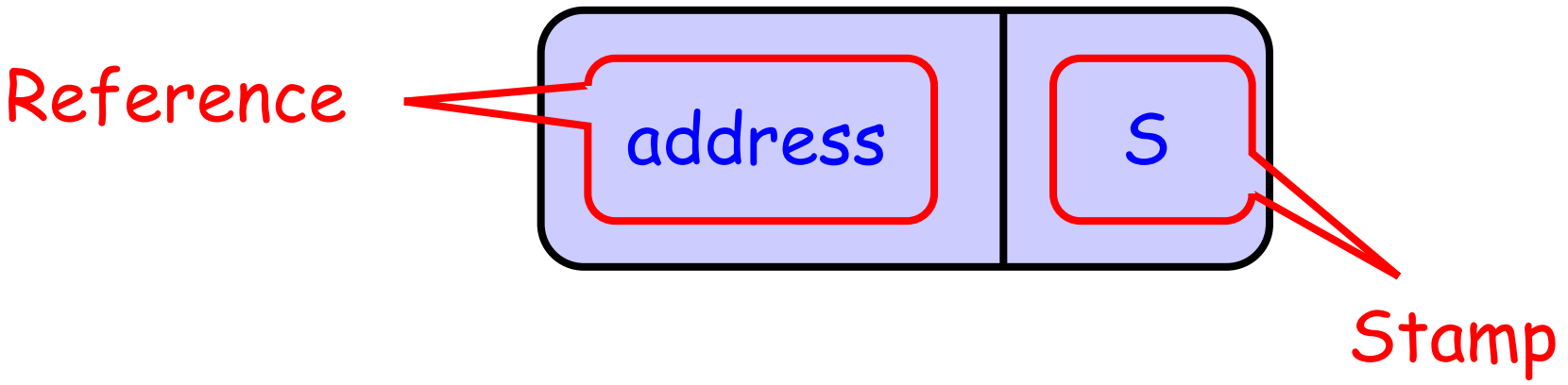
ABA Problem

- head pointer went A - B - A
- Due to semantics of CAS
  - It looks at the value only
  - Advanced: Check out Load-link/Store-conditional

Load-link/Store-conditional, Wikipedia: https://en.wikipedia.org/wiki/Load-link/store-conditional

Solution for ABA Problem

- Tag each pointer with a counter
- Pointer is unique over lifetime of node
- Pointer size v/s word size issues
  - Overflow?
- Java has AtomicStampedReference class



Reading

- [Paper] Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms: https://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf

Stacks

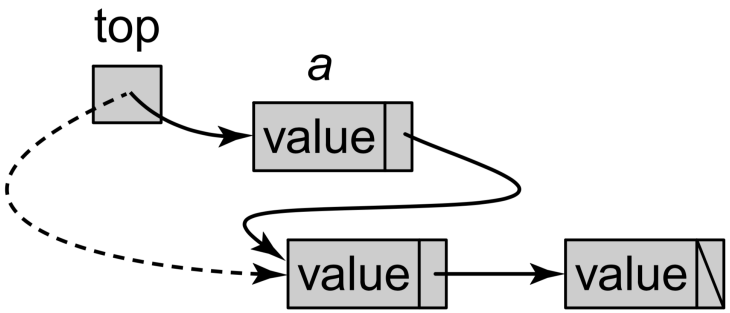
- Last-in-first-out (LIFO) order
- push() & pop()

- Appear inherently sequential?
- Little opportunity for concurrency?

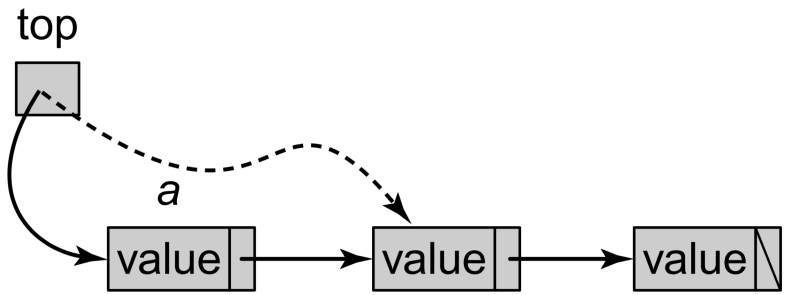
Lock-Free Stack

- Implement using linked-list
- Unbounded Total Lock-Free Stack
  - Push and pop don't block

```
public class Node {
    public T value;
    public Node next;
    public Node(T value) {
      value = value;
      next  = null;
    }
}
```

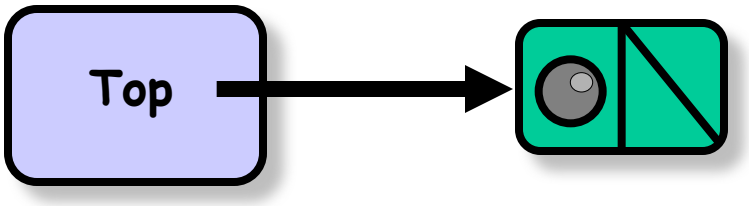


Push a

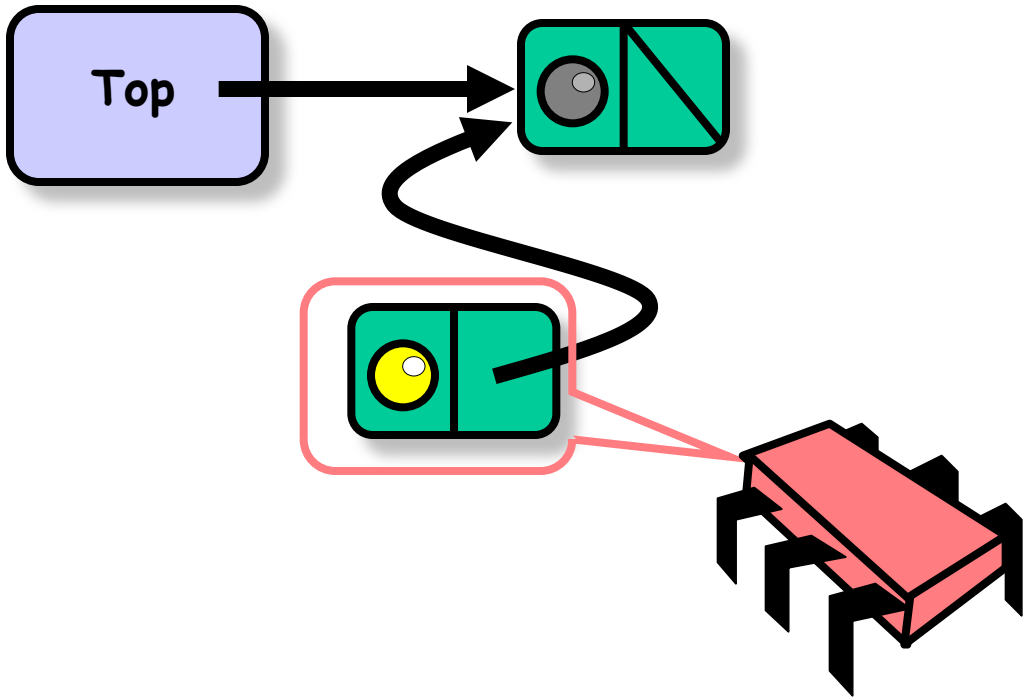


Pop a

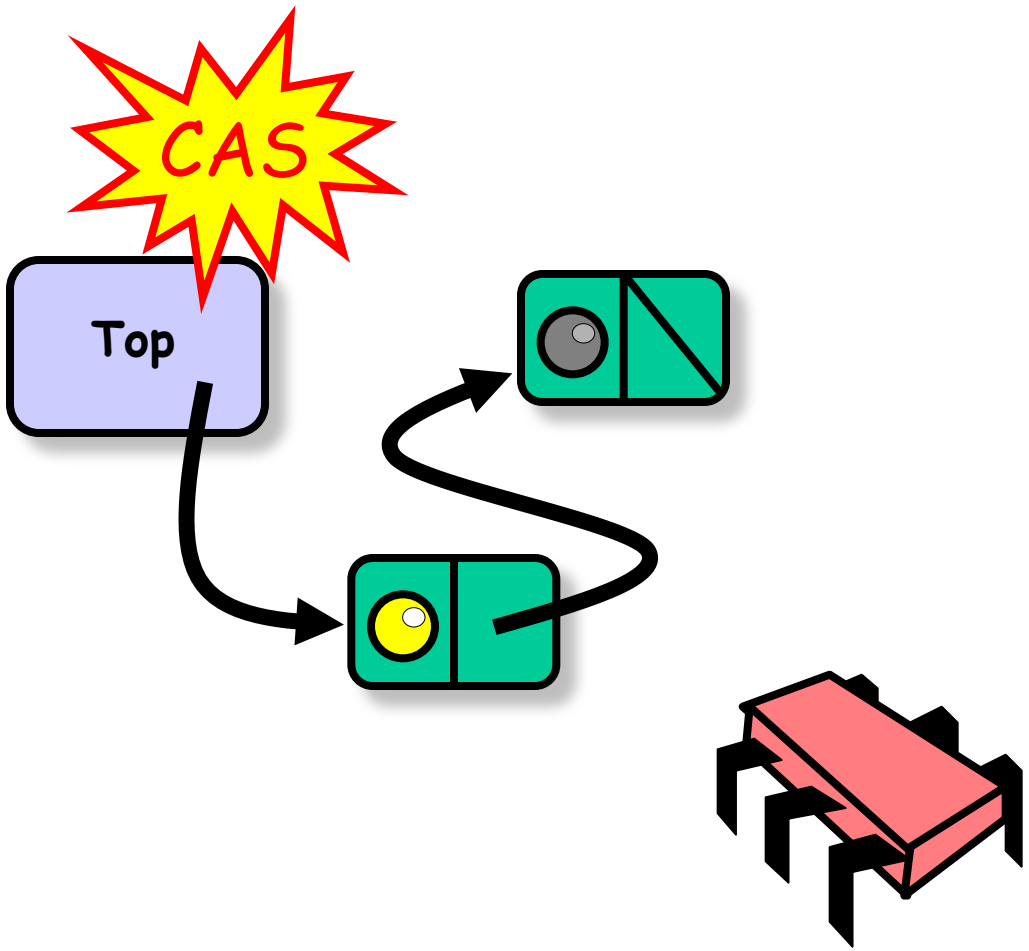
Lock-Free Stack: Empty



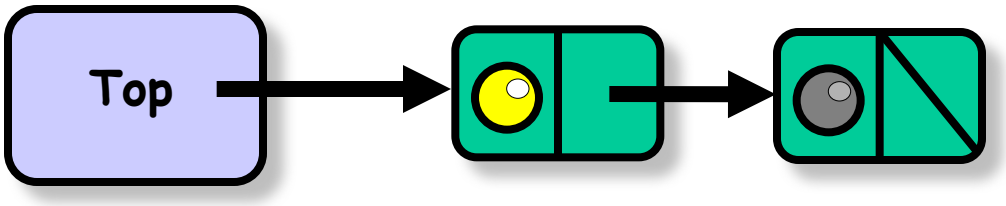
Lock-Free Stack: Push

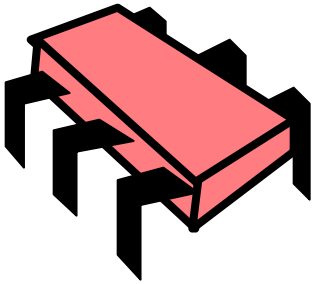


Lock-Free Stack: Push

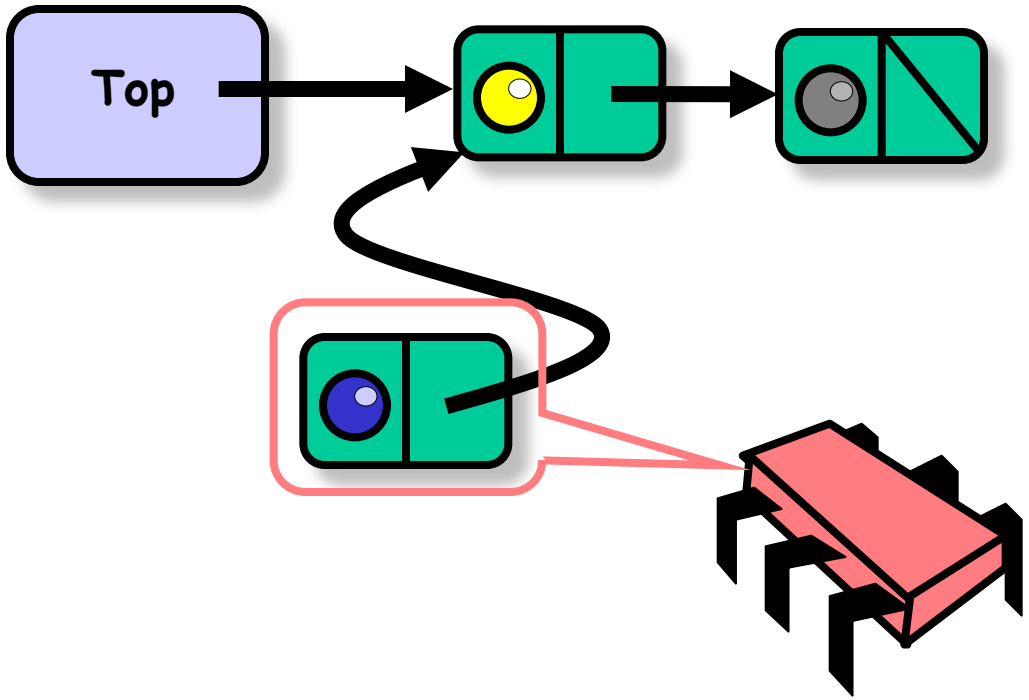


Lock-Free Stack: Push

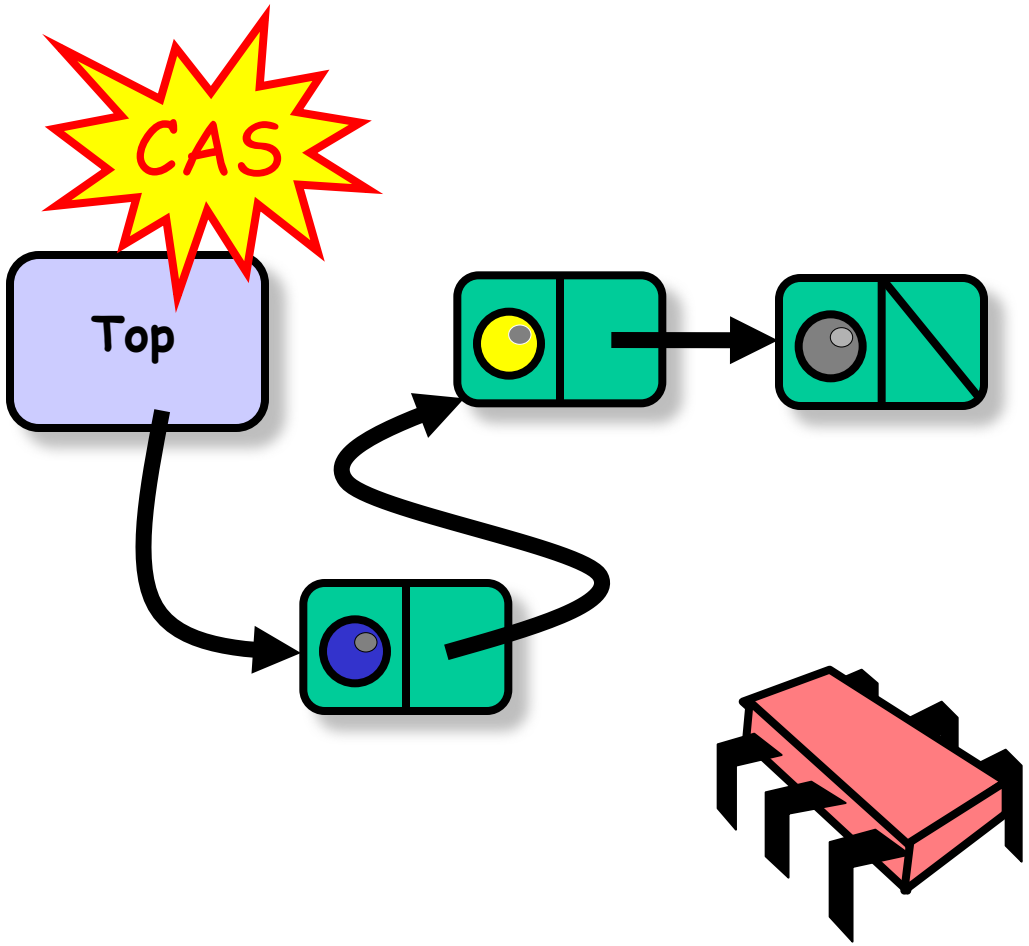




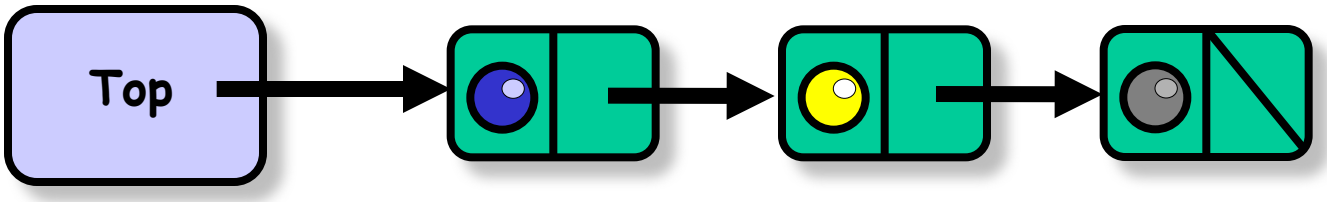
Lock-Free Stack: Push

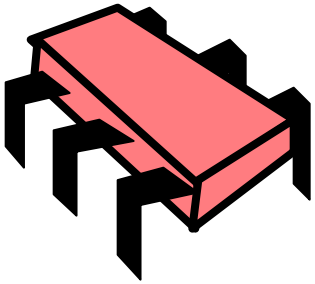


Lock-Free Stack: Push

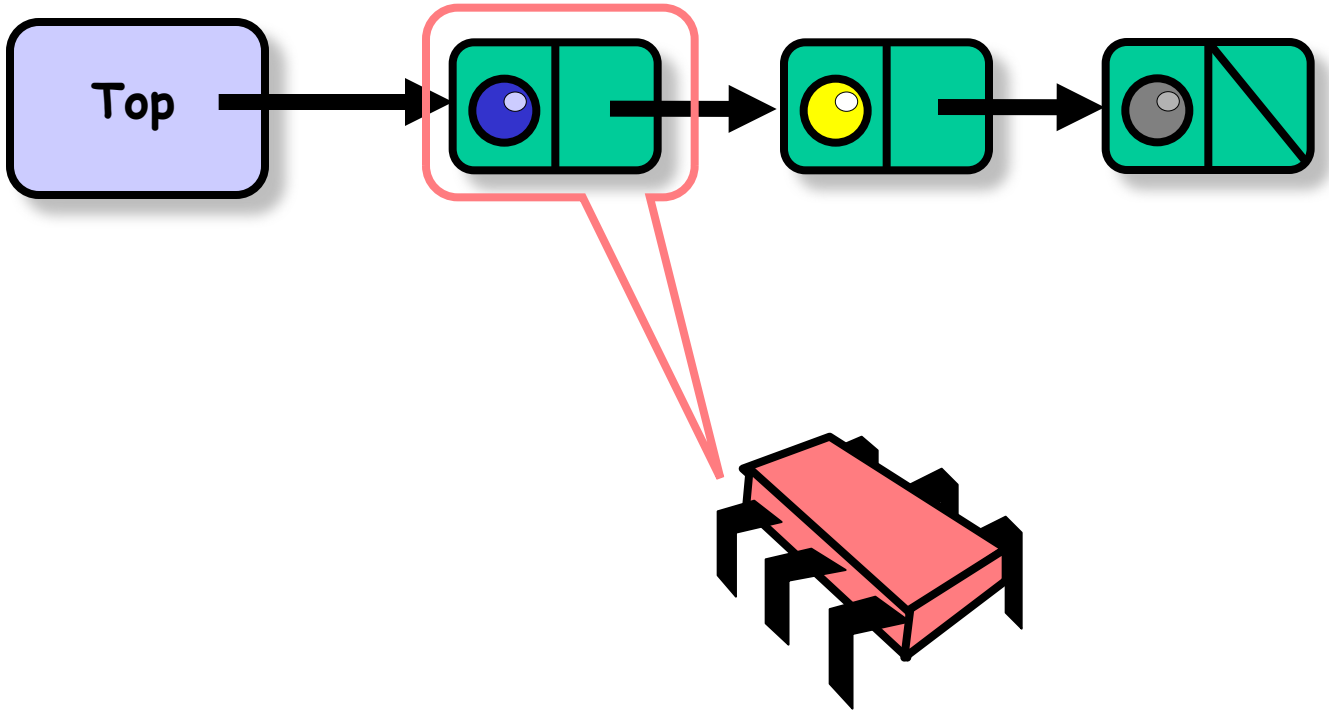


Lock-Free Stack: Push

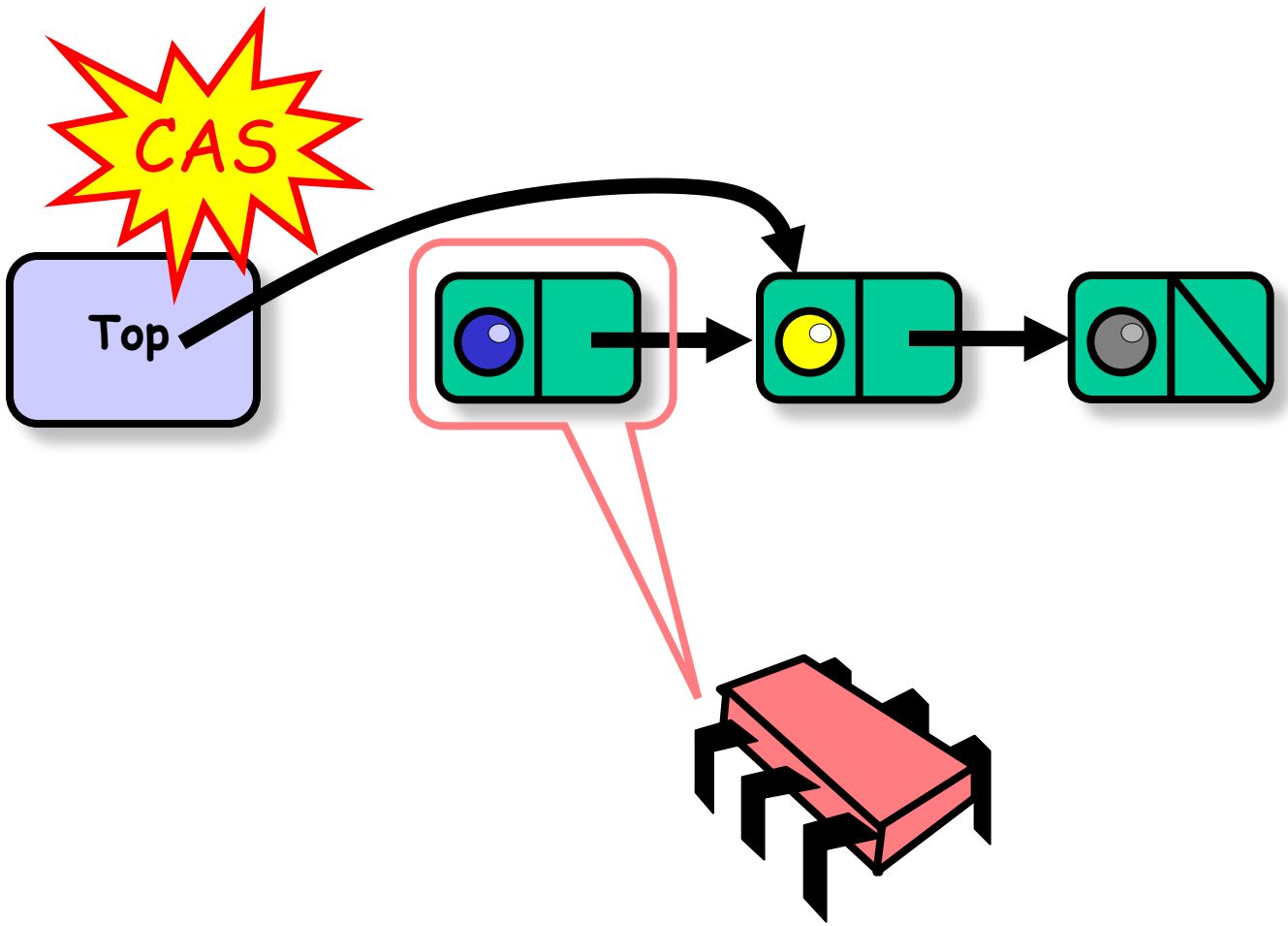




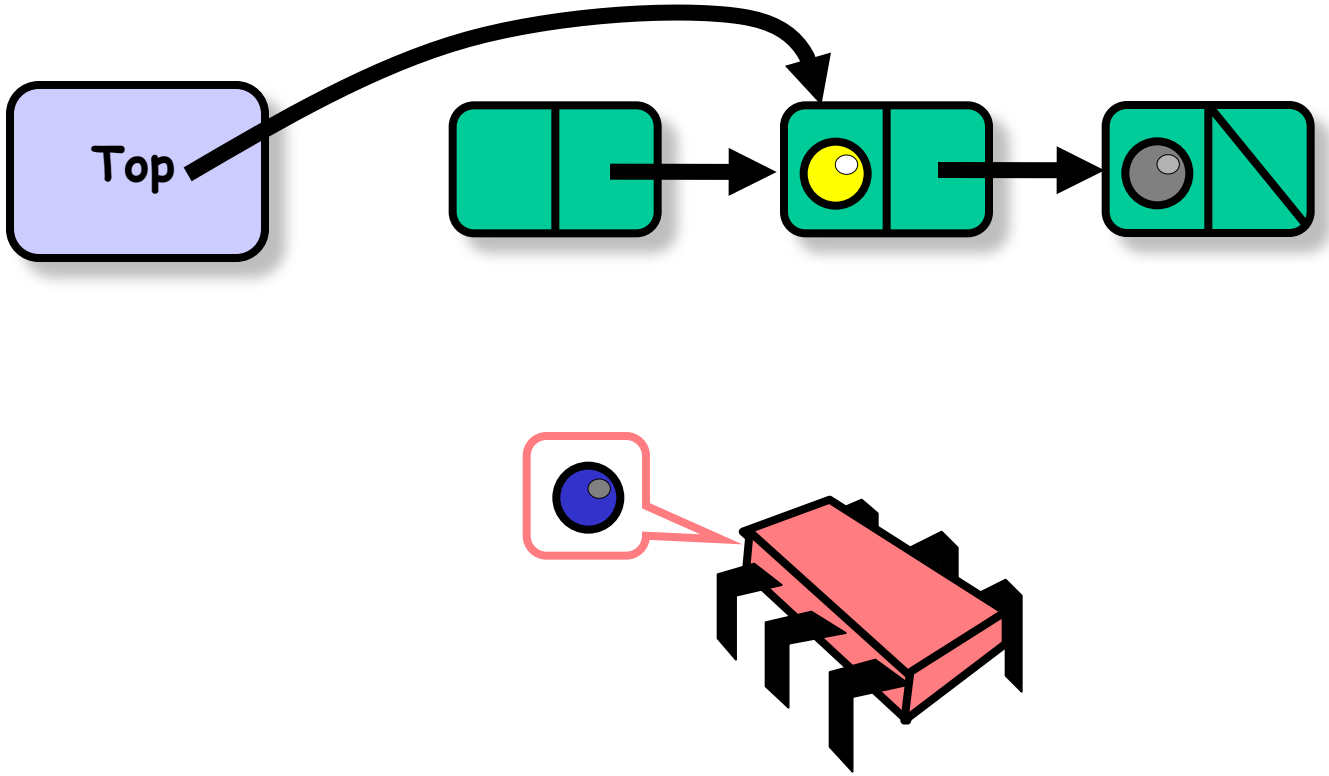
Lock-Free Stack: Pop



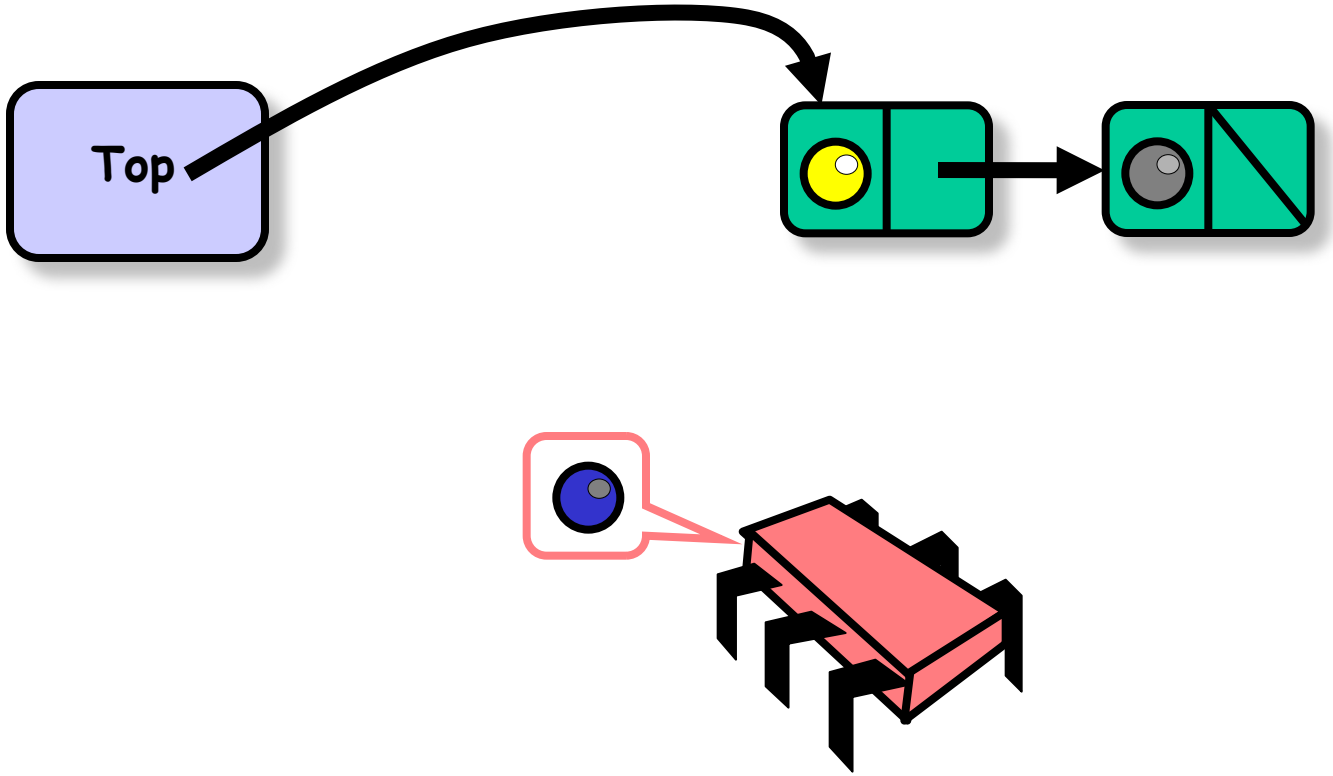
Lock-Free Stack: Pop



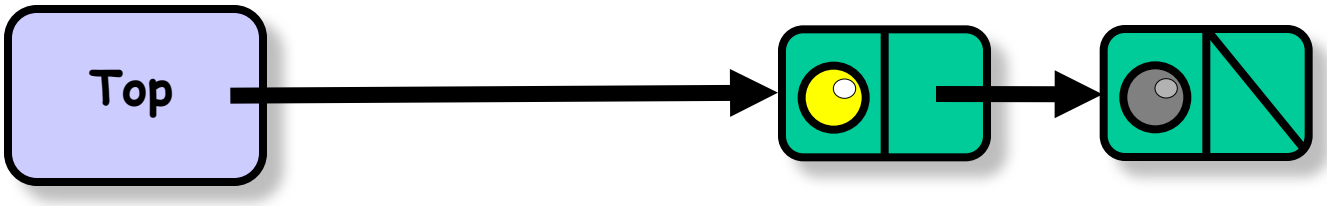
Lock-Free Stack: Pop

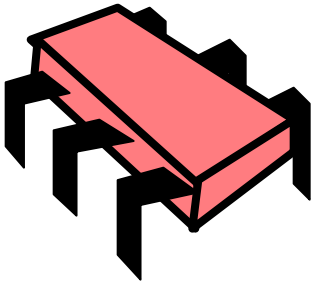


Lock-Free Stack: Pop



Lock-Free Stack: Pop





```
public class LockFreeStack<T> {
  AtomicReference<Node> top = new AtomicReference<Node>(null);
  static final int MIN_DELAY = ...;
  static final int MAX_DELAY = ...;
  Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);

  protected boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node));
  }
  public void push(T value) {
    Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else {
        backoff.backoff();
      }
    }
  }
```

```
protected Node tryPop() throws EmptyException {
  Node oldTop = top.get();
  if (oldTop == null) {
    throw new EmptyException();
  }
  Node newTop = oldTop.next;
  if (top.compareAndSet(oldTop, newTop)) {
    return oldTop;
  } else {
    return null;
  }
}
public T pop() throws EmptyException {
  while (true) {
    Node returnNode = tryPop();
    if (returnNode != null) {
      return returnNode.value;
    } else {
      backoff.backoff();
    }
  }
}
```

Reduce contention

```
public class LockFreeStack<T> {
  AtomicReference<Node> top = new AtomicReference<Node>(null);
  static final int MIN_DELAY = ...;
  static final int MAX_DELAY = ...;
  Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);

  protected boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node));
  }
  public void push(T value) {
    Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else {
        backoff.backoff();
      }
    }
  }
```

```
protected Node tryPop() throws EmptyException {
  Node oldTop = top.get();
  if (oldTop == null) {
    throw new EmptyException();
  }
  Node newTop = oldTop.next;
  if (top.compareAndSet(oldTop, newTop)) {
    return oldTop;
  } else {
    return null;
  }
}
public T pop() throws EmptyException {
  while (true) {
    Node returnNode = tryPop();
    if (returnNode != null) {
      return returnNode.value;
    } else {
      backoff.backoff();
    }
  }
}
```

Linearization Points?

```
public class LockFreeStack<T> {
  AtomicReference<Node> top = new AtomicReference<Node>(null);
  static final int MIN_DELAY = ...;
  static final int MAX_DELAY = ...;
  Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);

  protected boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node));
  }
  public void push(T value) {
    Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else {
        backoff.backoff();
      }
    }
  }
```

```
protected Node tryPop() throws EmptyException {
  Node oldTop = top.get();
  if (oldTop == null) {
    throw new EmptyException();
  }
  Node newTop = oldTop.next;
  if (top.compareAndSet(oldTop, newTop)) {
    return oldTop;
  } else {
    return null;
  }
}
public T pop() throws EmptyException {
  while (true) {
    Node returnNode = tryPop();
    if (returnNode != null) {
      return returnNode.value;
    } else {
      backoff.backoff();
    }
  }
}
```

Is this good enough?

Linearization Points?

```
public class LockFreeStack<T> {
  AtomicReference<Node> top = new AtomicReference<Node>(null);
  static final int MIN_DELAY = ...;
  static final int MAX_DELAY = ...;
  Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);

  protected boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return(top.compareAndSet(oldTop, node));
  }
  public void push(T value) {
    Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else {
        backoff.backoff();
      }
    }
  }
```

Is this good enough?

```
protected Node tryPop() throws EmptyException {
  Node oldTop = top.get();
  if (oldTop == null) {
    throw new EmptyException();
  }
  Node newTop = oldTop.next;
  if (top.compareAndSet(oldTop, newTop)) {
    return oldTop;
  } else {
    return null;
  }
}
public T pop() throws EmptyException {
  while (true) {
    Node returnNode = tryPop();
    if (returnNode != null) {
      return returnNode.value;
    } else {
      backoff.backoff();
    }
  }
}
```

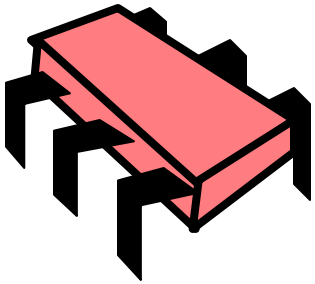
Pushes and Pops happen serially

Lock-Free Stack

- Operations happen serially
  - Push and pop fight for top
- Are stacks inherently sequential?
- How to enable concurrent operations?

Observation

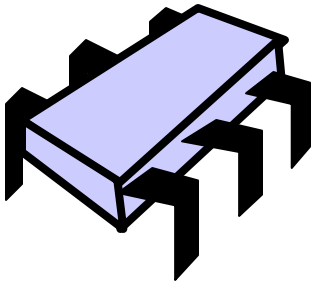
After an equal number of pushes and pops, stack stays the same



Push( )

linearizable stack

Pop()

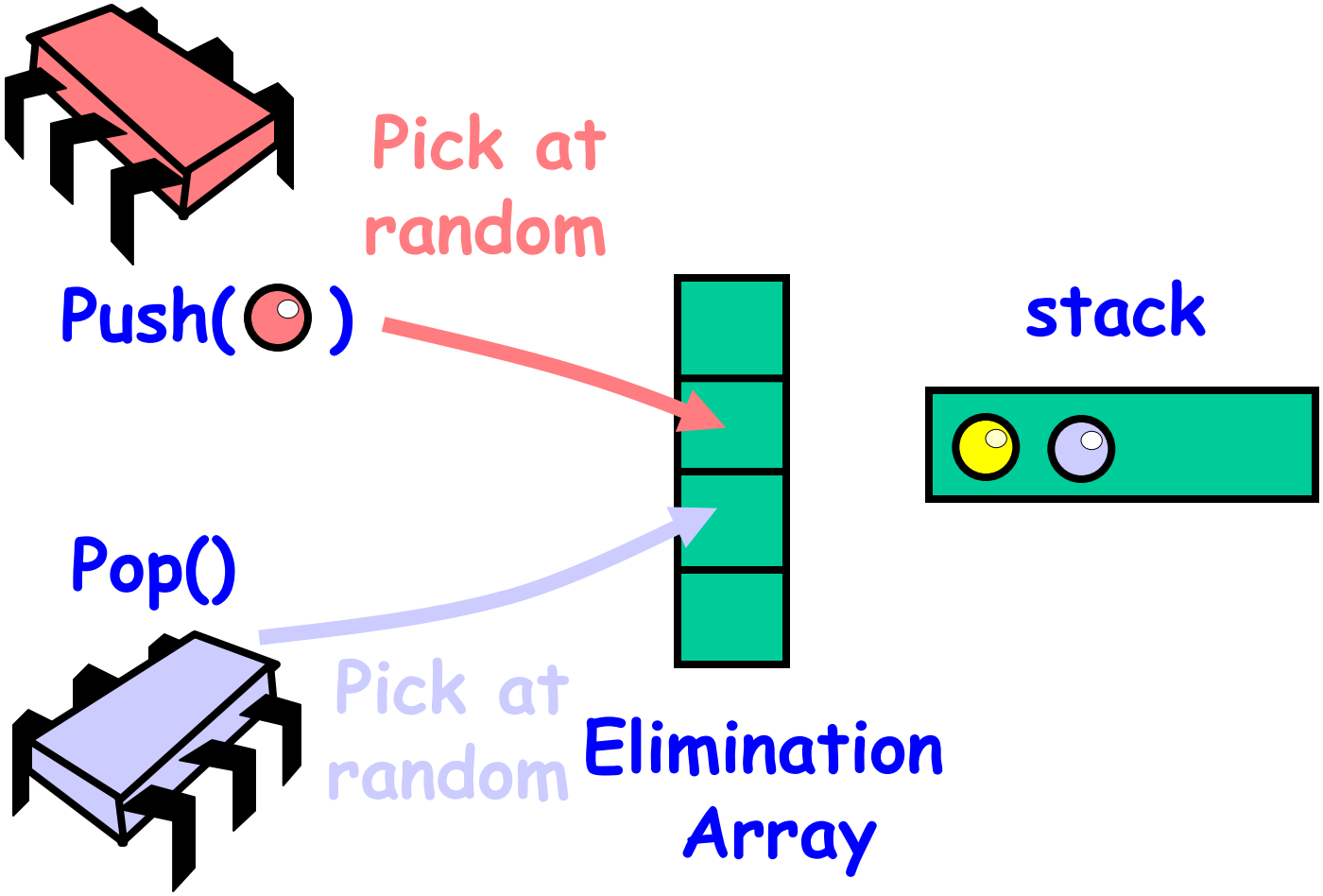


So if a push and a pop occurs concurrently, stack stays the same

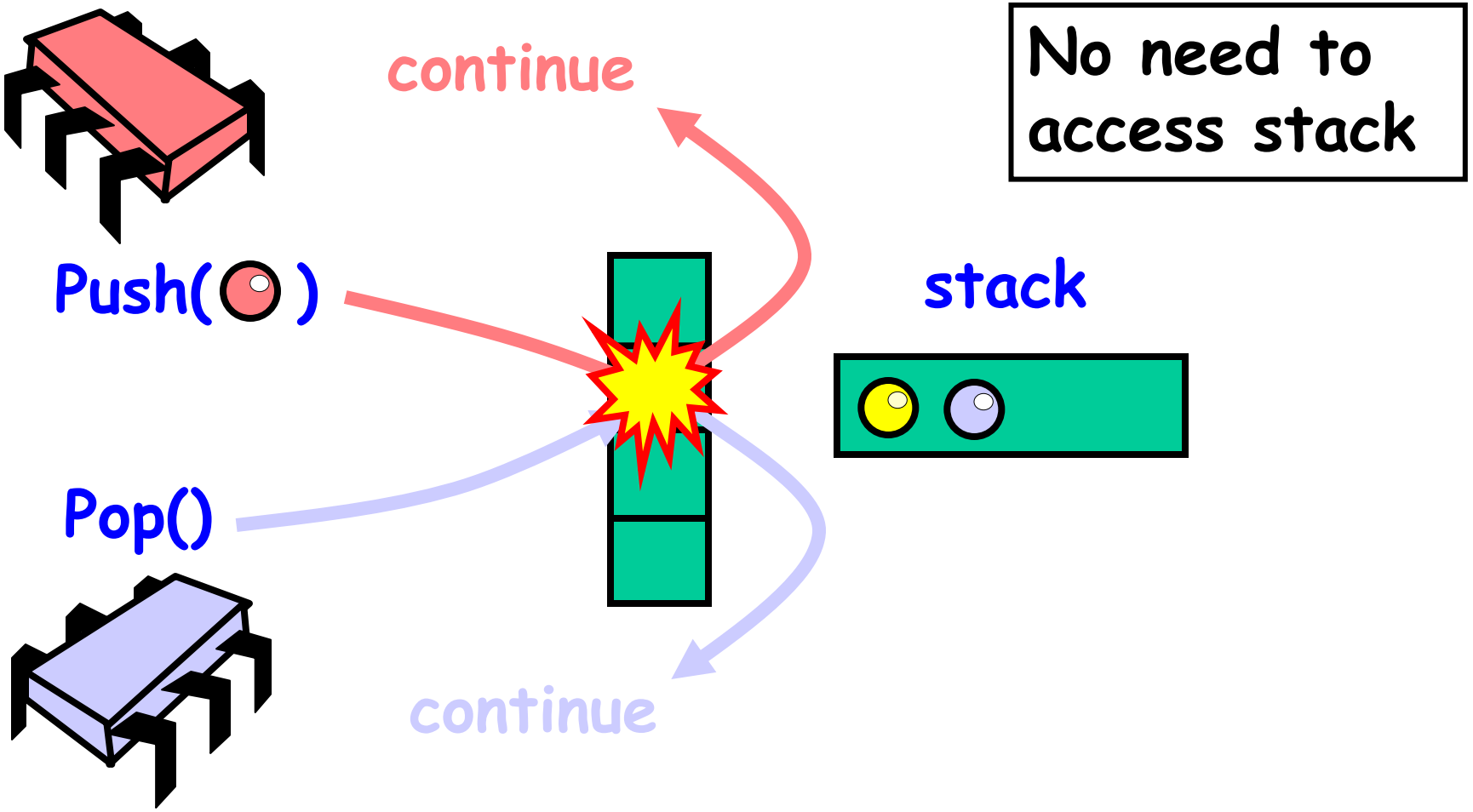
Elimination Backoff

- Turn contention into parallelism
- Concurrent push and pop operations can be matched outside the linearizable stack

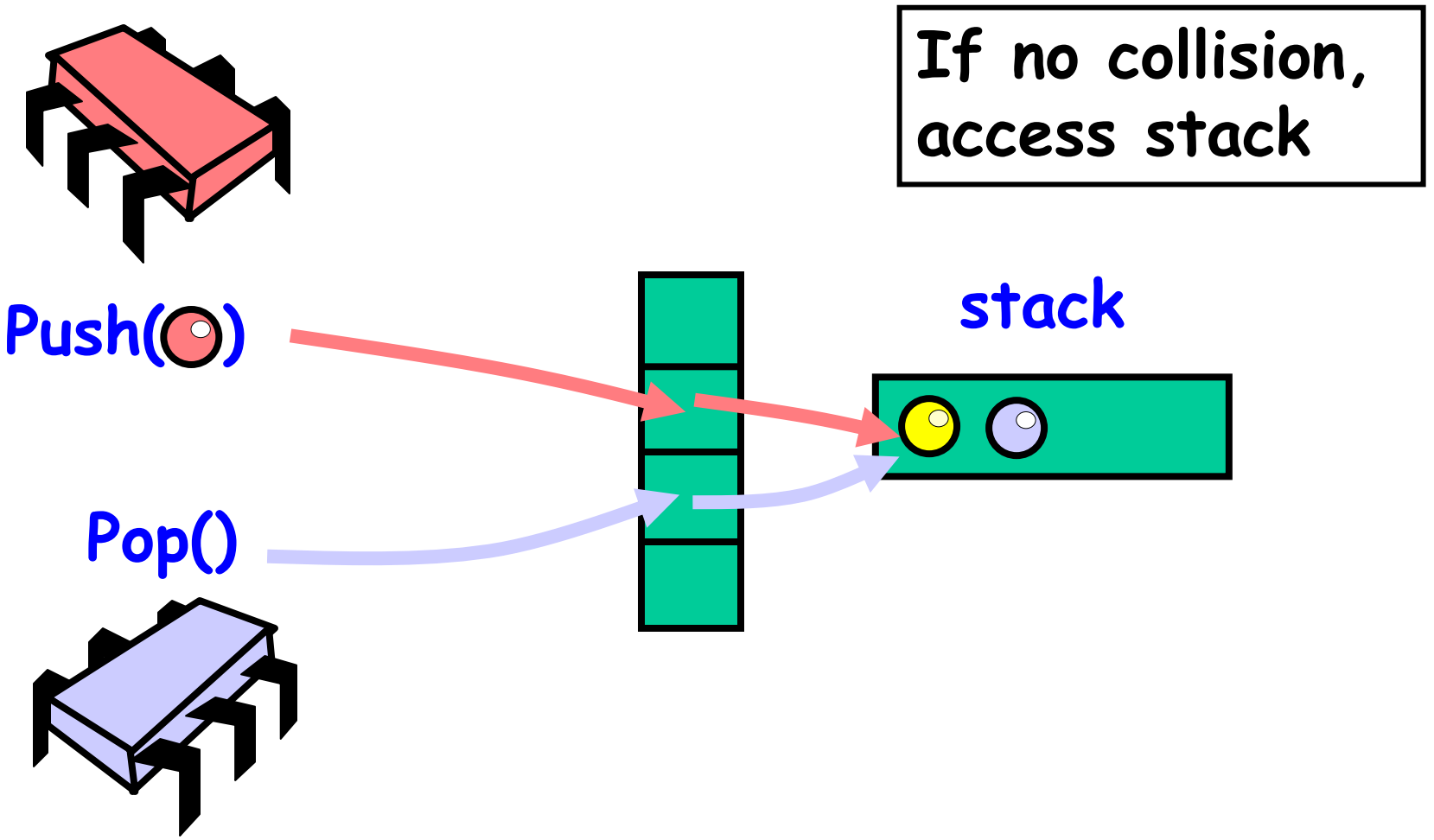
Elimination Backoff



Elimination Backoff: Collision



Elimination Backoff: No Collision

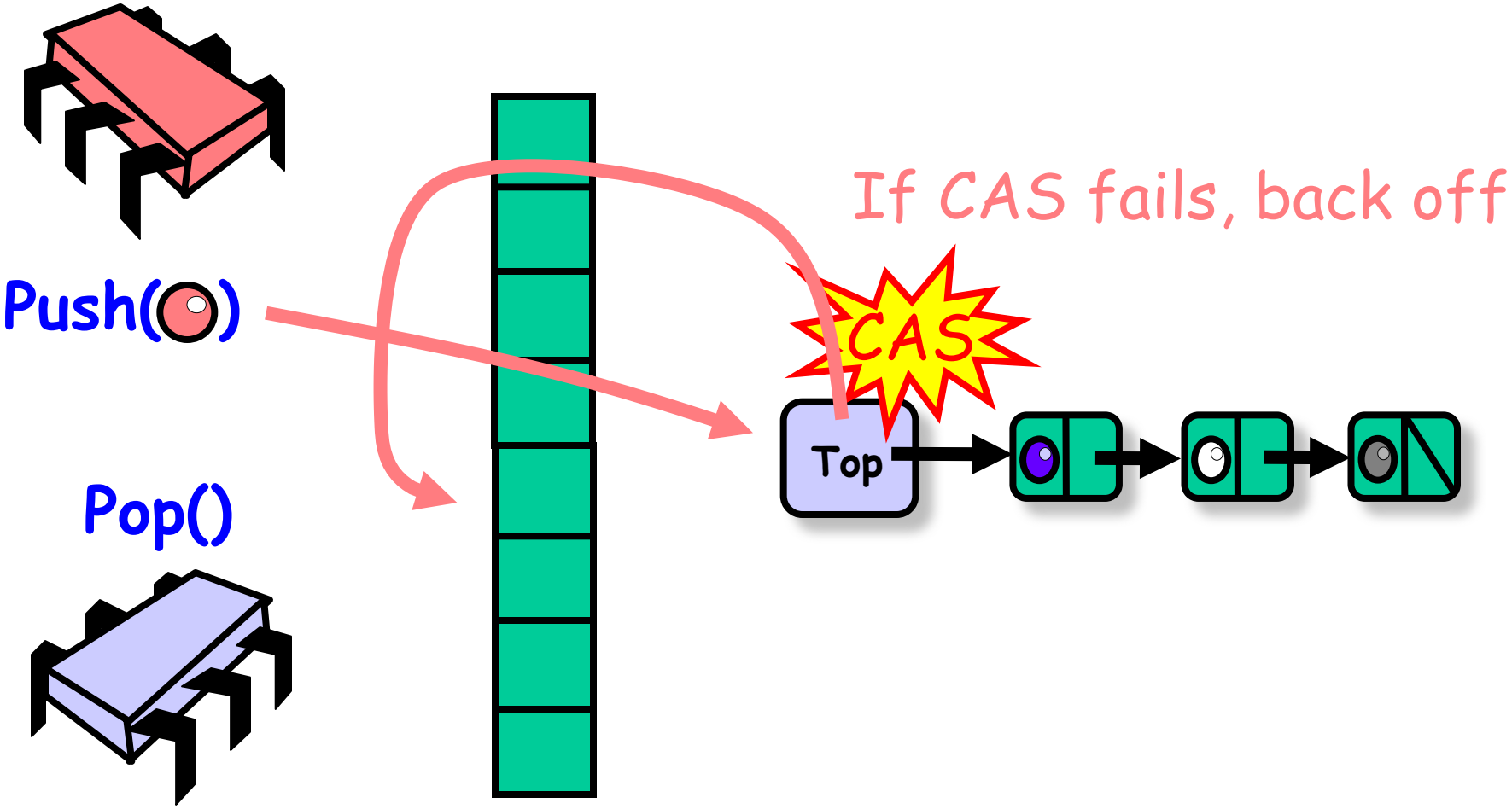


Elimination Backoff Stack

- Turn contention into parallelism
- Concurrent push and pop operations can be matched outside the linearizable stack

- Access Lock-free stack:
  - If uncontended, apply operation
  - If contended, back off to elimination array and attempt elimination

Elimination Backoff Stack



Elimination Array

Checkout code in Textbook!

- Every slot is in one of 3 states: EMPTY, WAITING, BUSY
- First thread finds an EMPTY slot:
  - Places its item and changes EMPTY to WAITING
  - Keeps on waiting for state to change (or until timeout)
- Second thread finds a WAITING slot:
  - Exchanges its item and changes WAITING to BUSY (and returns)
- First thread observes state change to BUSY:
  - Reads the item and changes state to EMPTY (and returns)

- Choices: Size of array, how to search for empty slot, etc.

Elimination Backoff Stack

- How to reason whether this is linearizable?

- Un-eliminated calls:
  - Linearized as before
- Eliminated calls:
  - Concurrent operations
  - Linearize pop immediately after matching push (i.e., when they collide)
- Combination is a linearizable stack