Concurrent programming

Concurrent Queues and Stacks

Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear

Modified by Piotr Witkowski

The Five-Fold Path

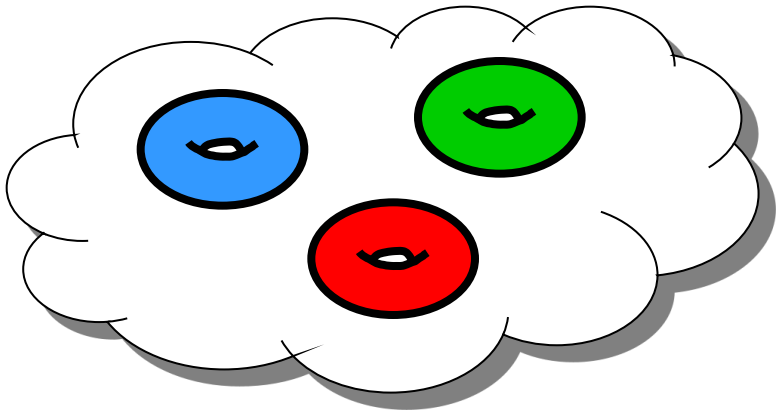
- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization
- Lazy synchronization
- Lock-free synchronization

Another Fundamental Problem

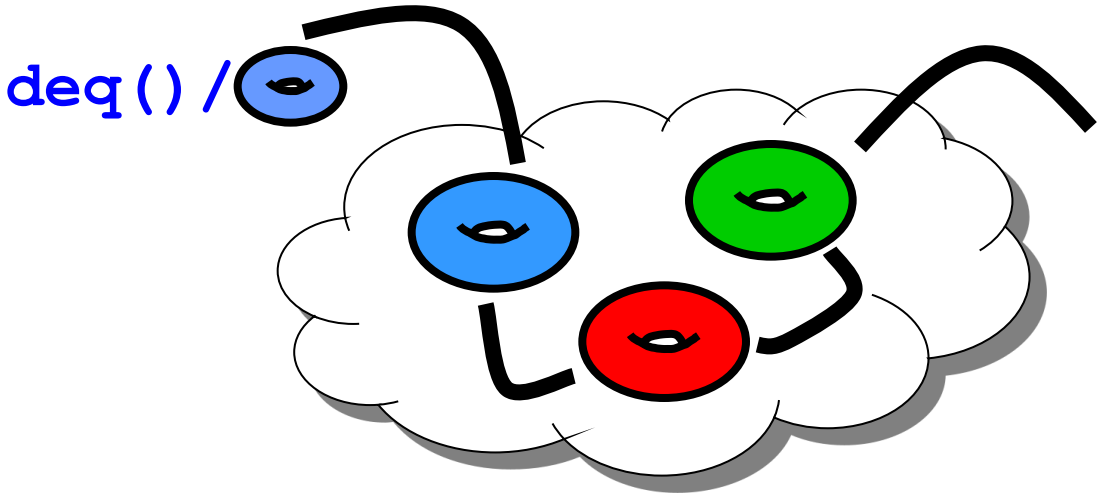
- We told you about
  - Sets implemented by linked lists
  - ~~Hash Tables~~
- Next: queues
- Next: stacks

Queues & Stacks

- pool of items

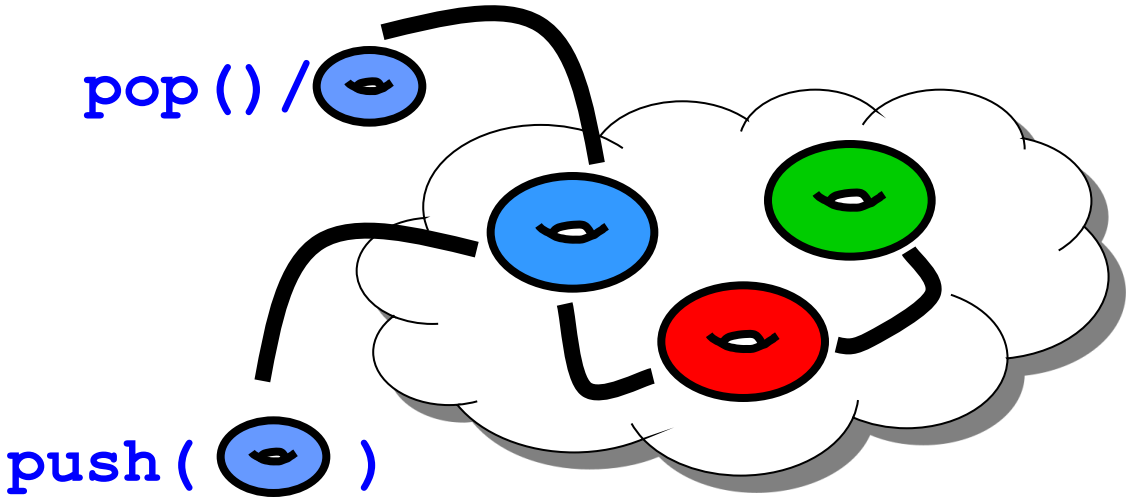


Queues



Total order
First in
First out

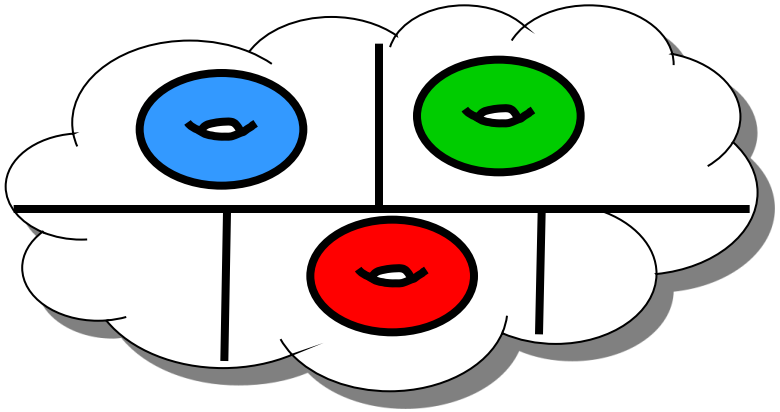
Stacks



Total order
Last in
First out

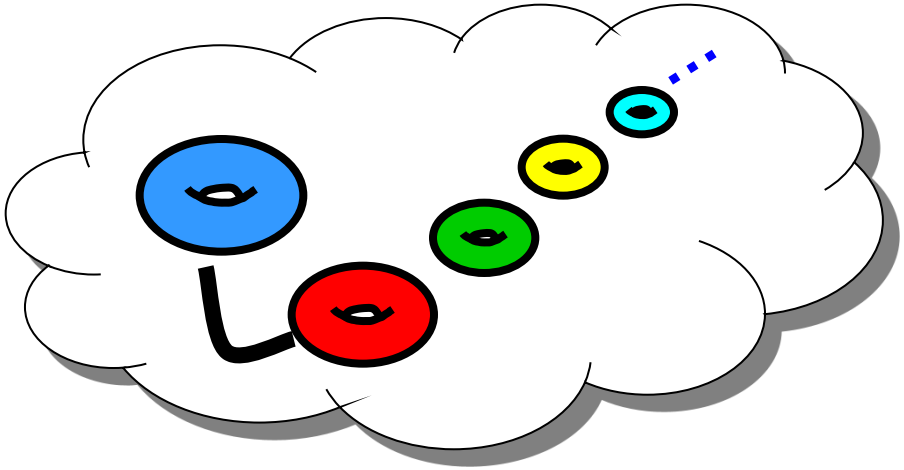
Bounded

- Fixed capacity
- Good when resources an issue

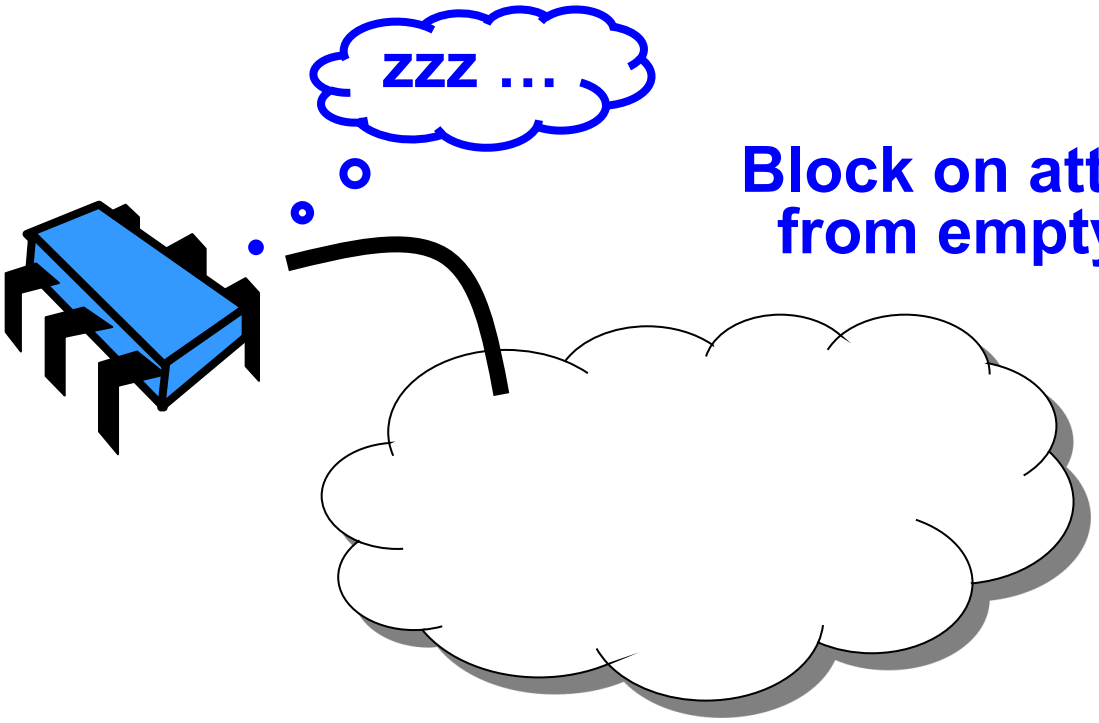


Unbounded

- Unlimited capacity
- Often more convenient

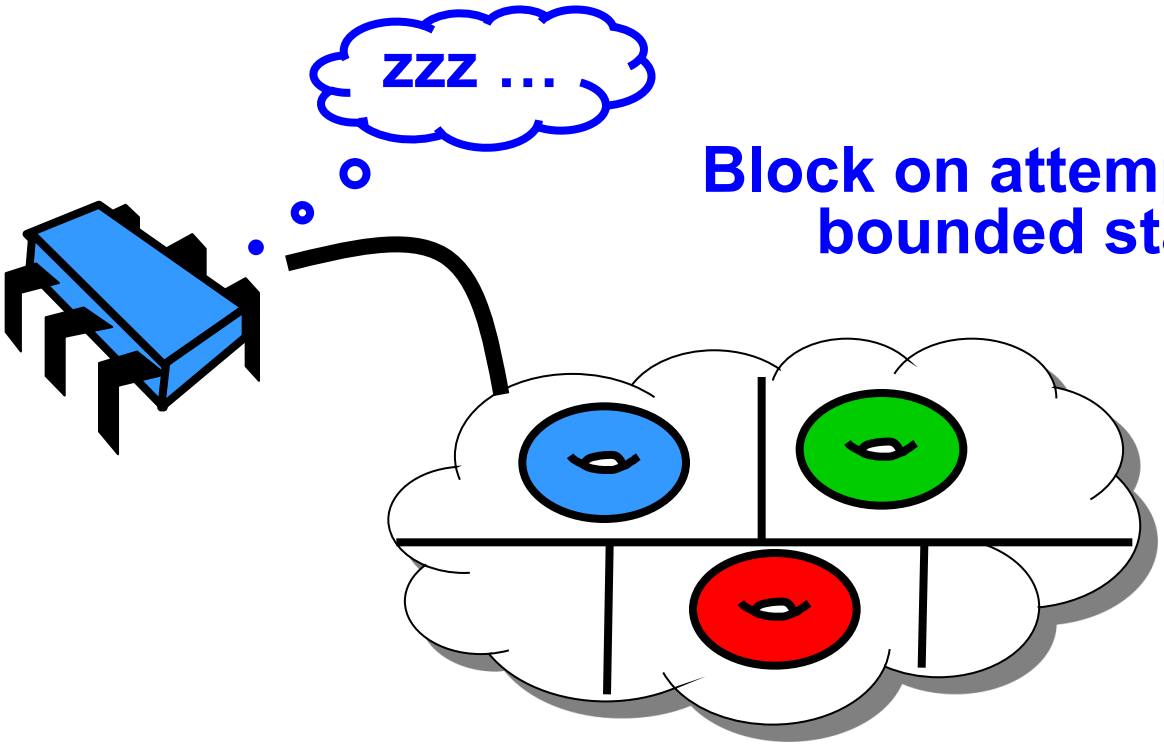


Blocking



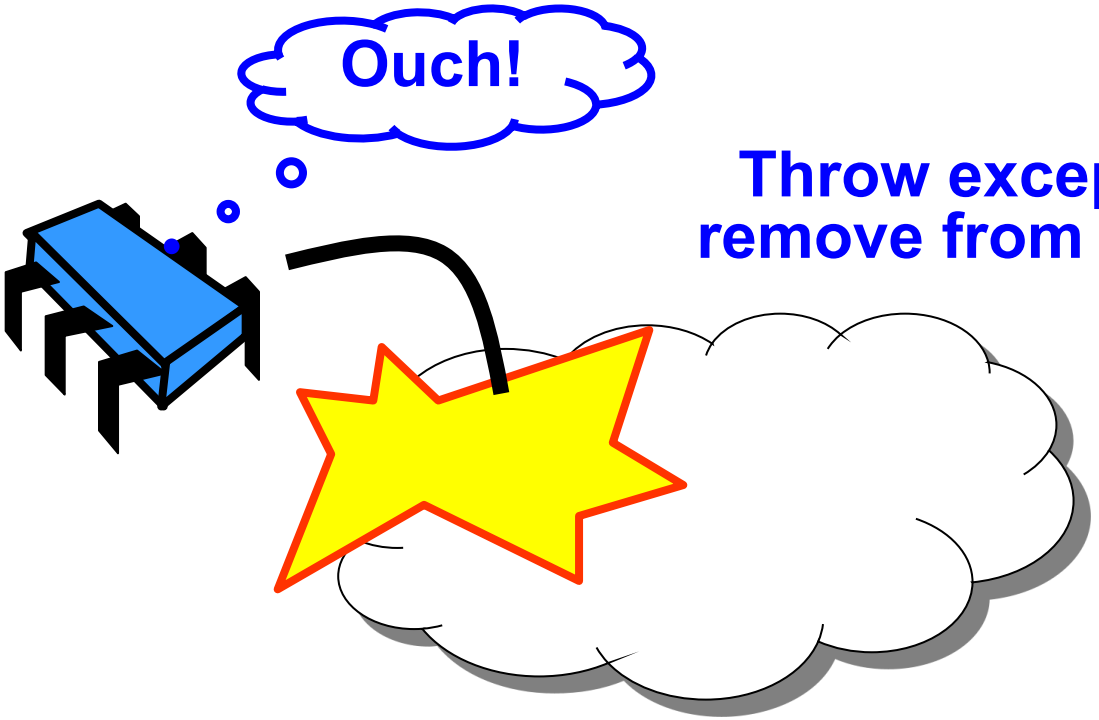
Block on attempt to remove from empty stack or queue

Blocking



Block on attempt to add to full bounded stack or queue

Non-Blocking

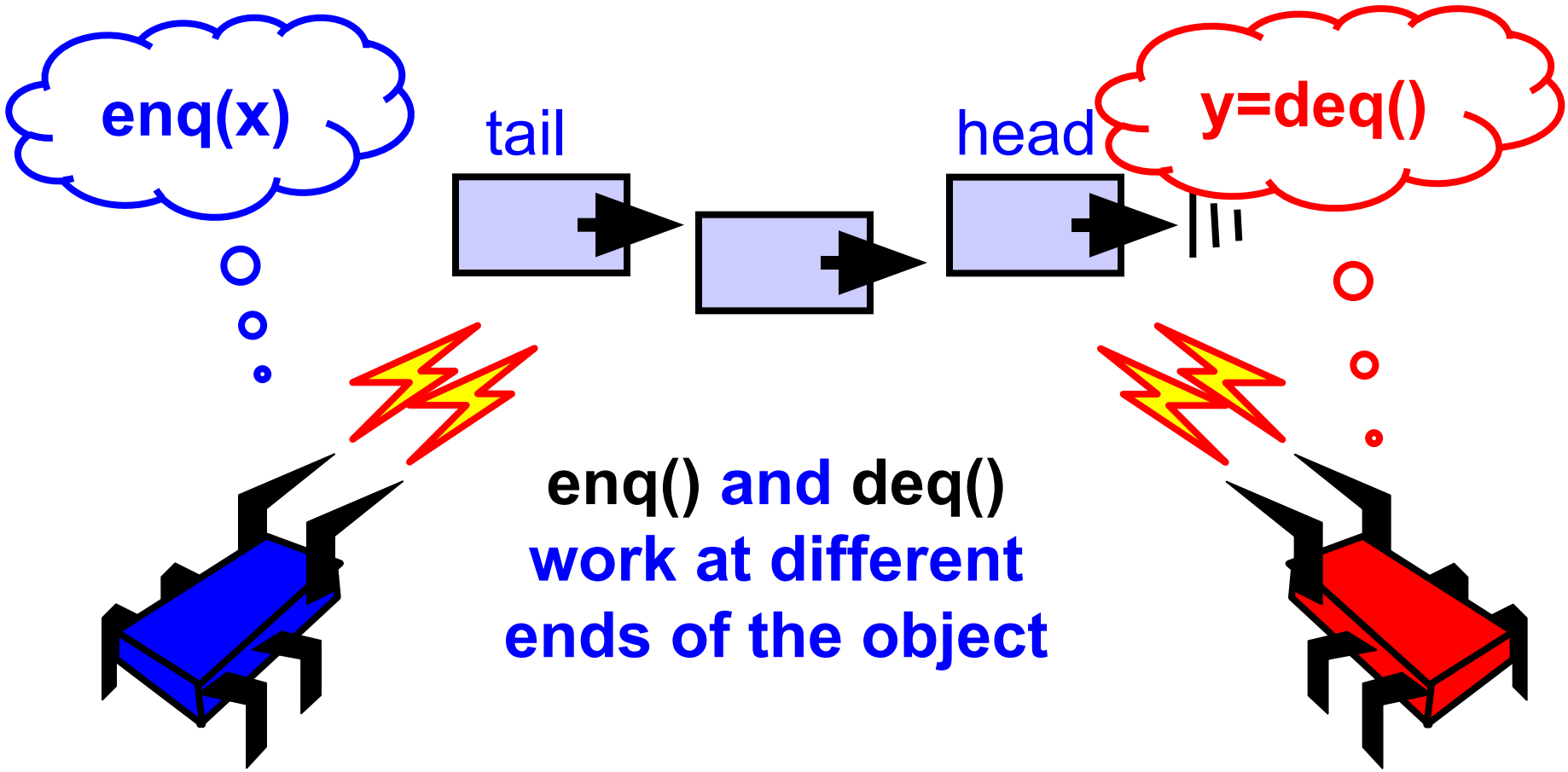


Throw exception on attempt to remove from empty stack or queue

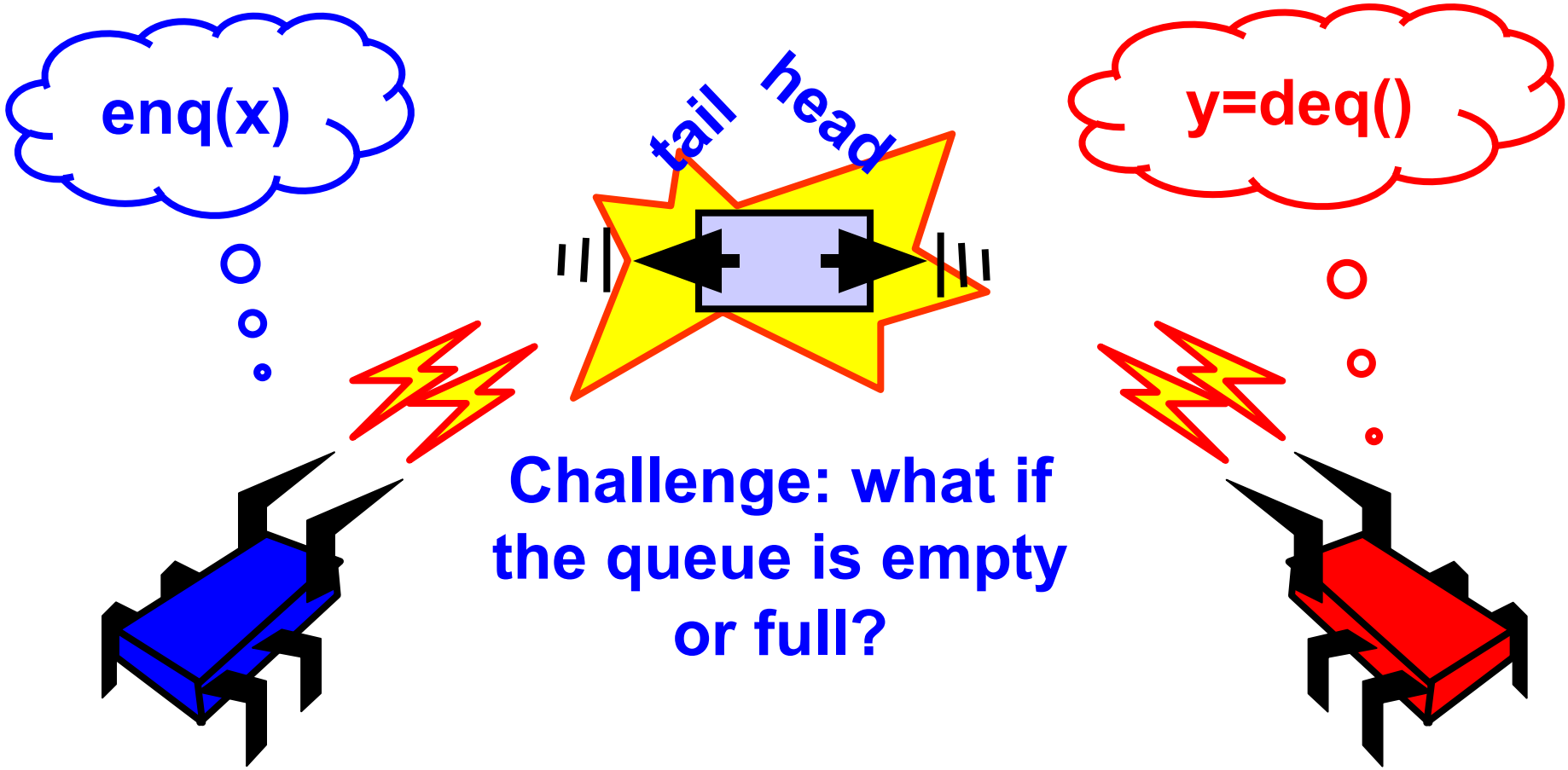
This Lecture

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

Queue: Concurrency

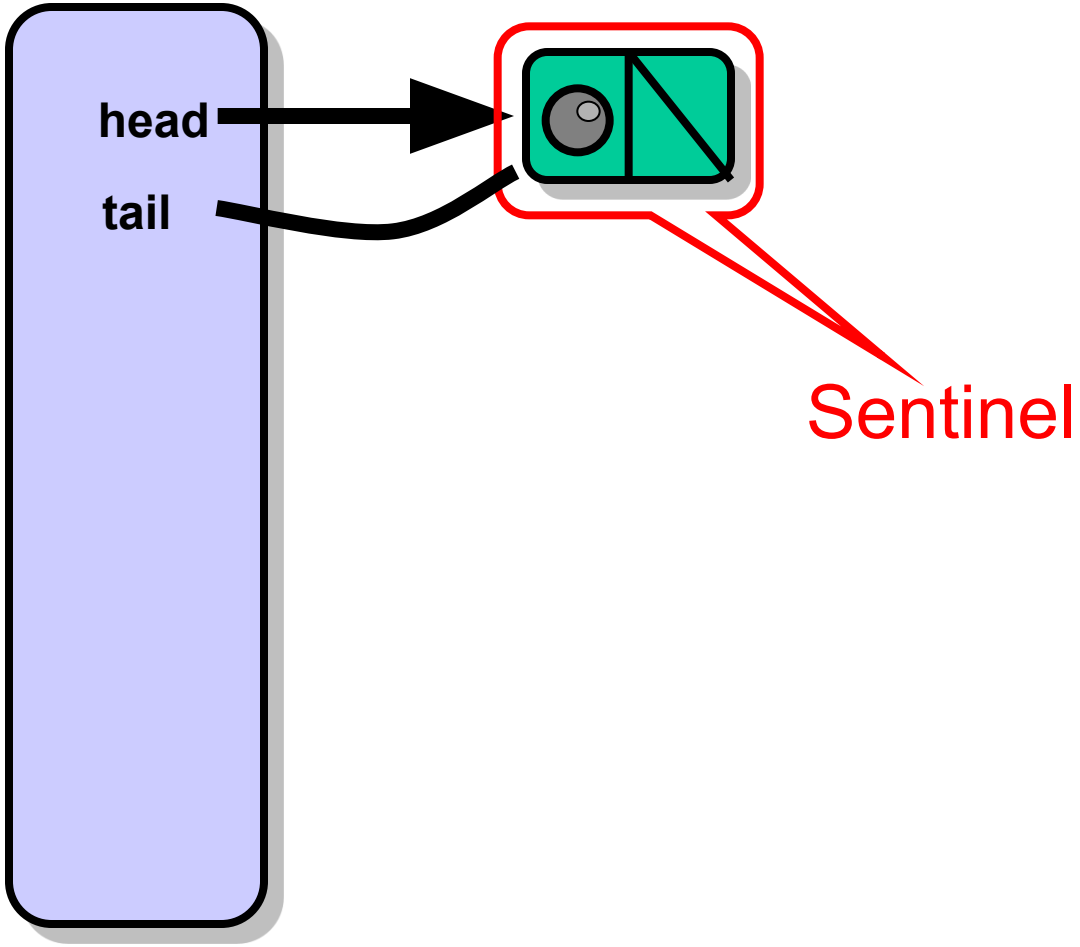


Concurrency

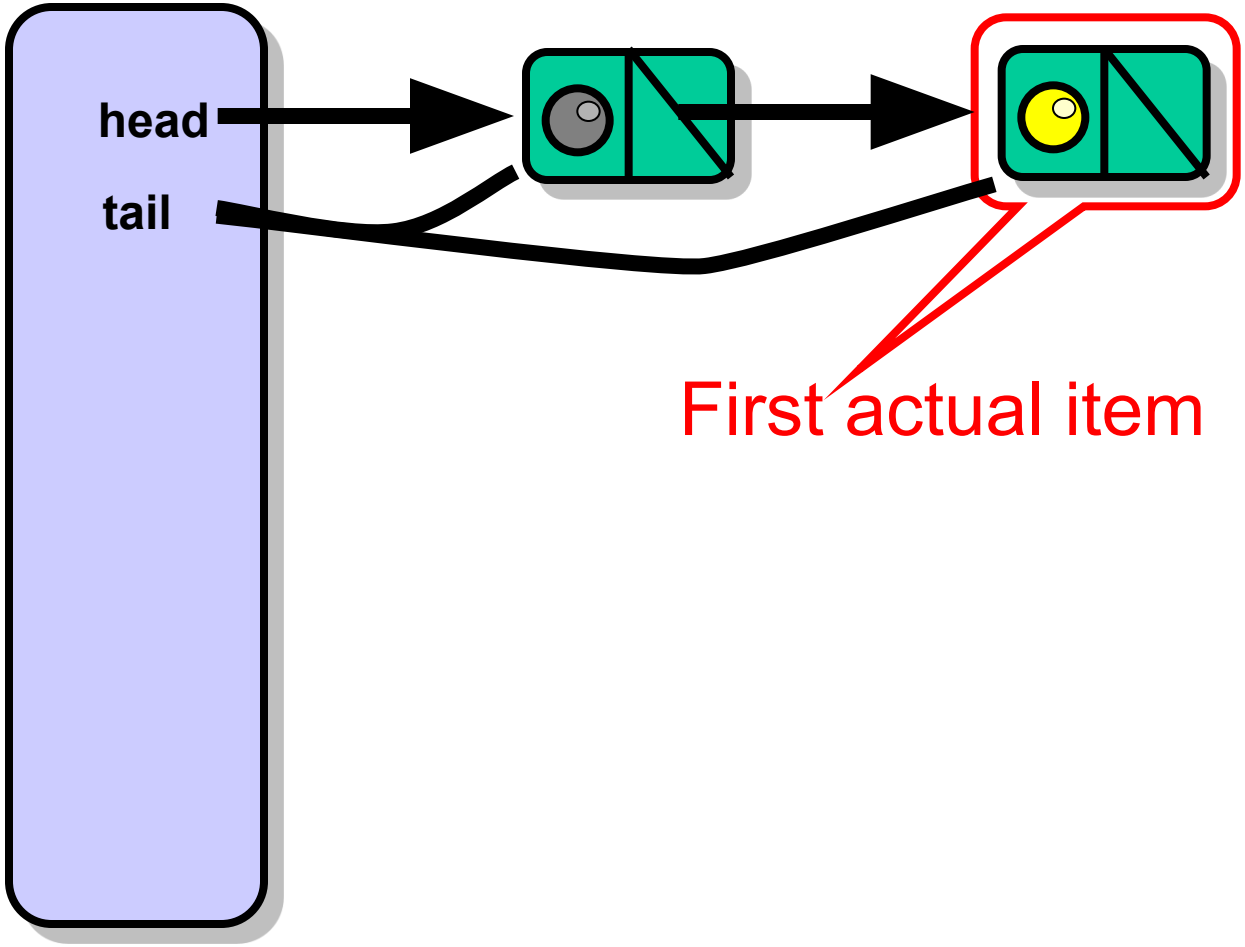


Challenge: what if the queue is empty or full?

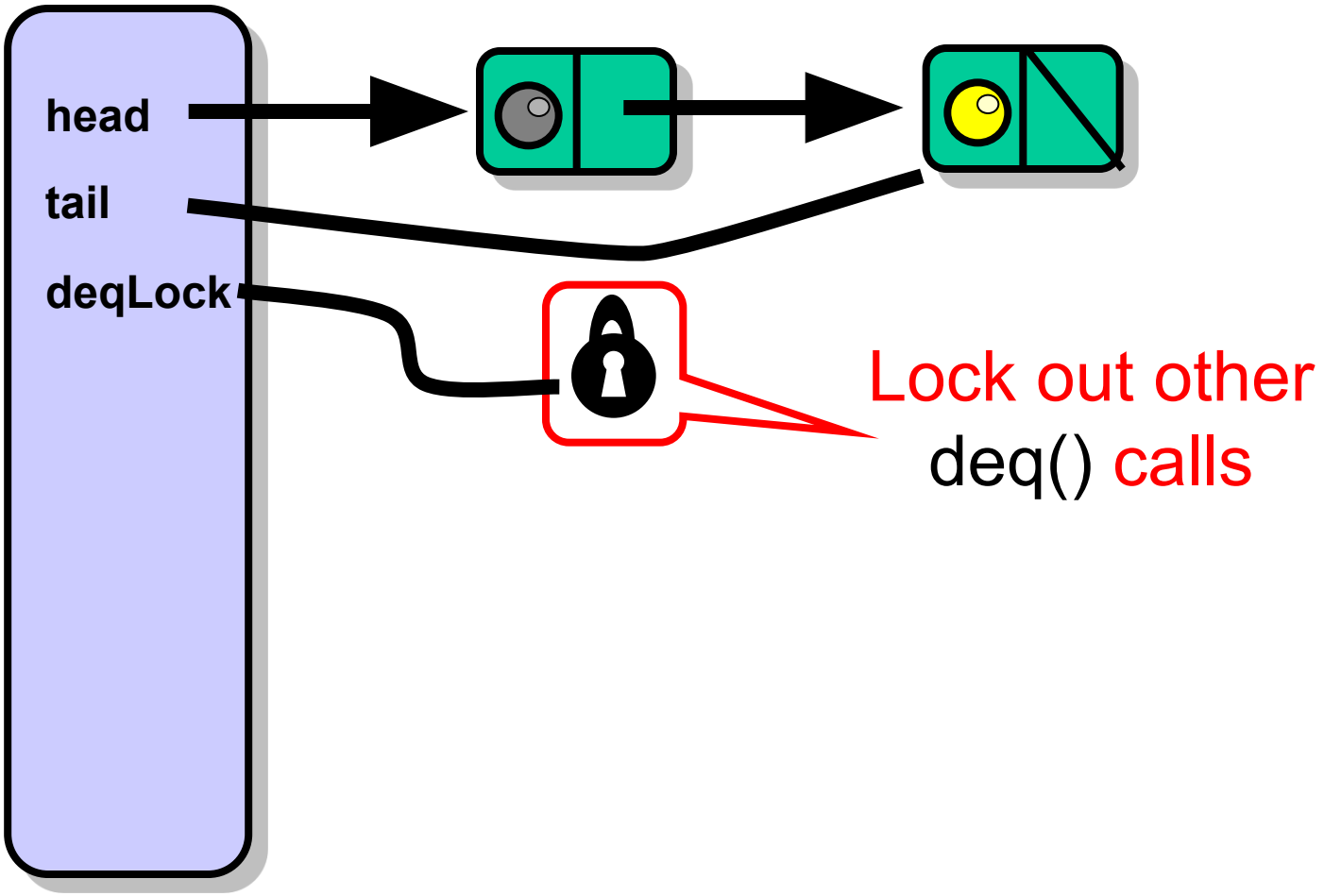
Bounded Queue



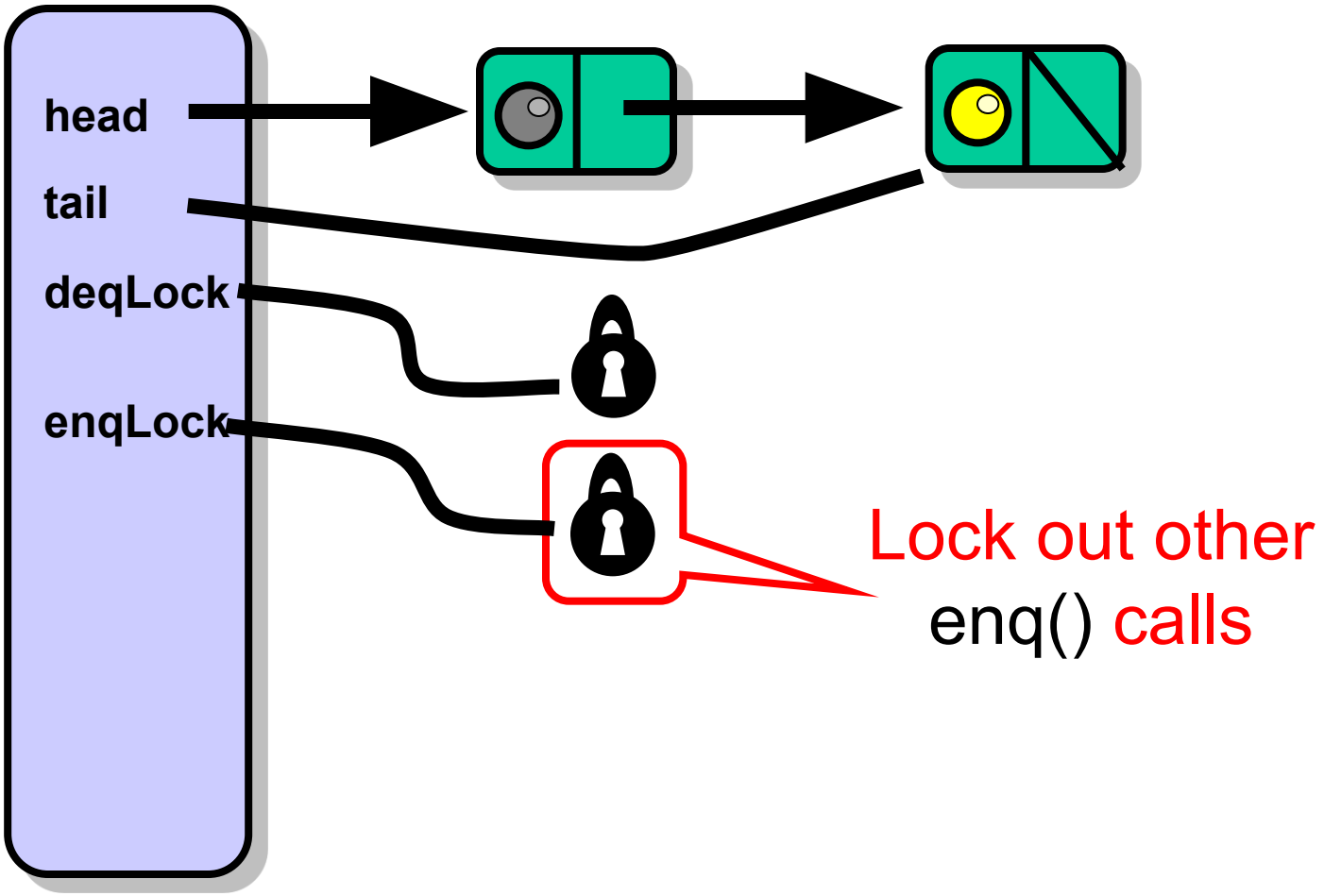
Bounded Queue



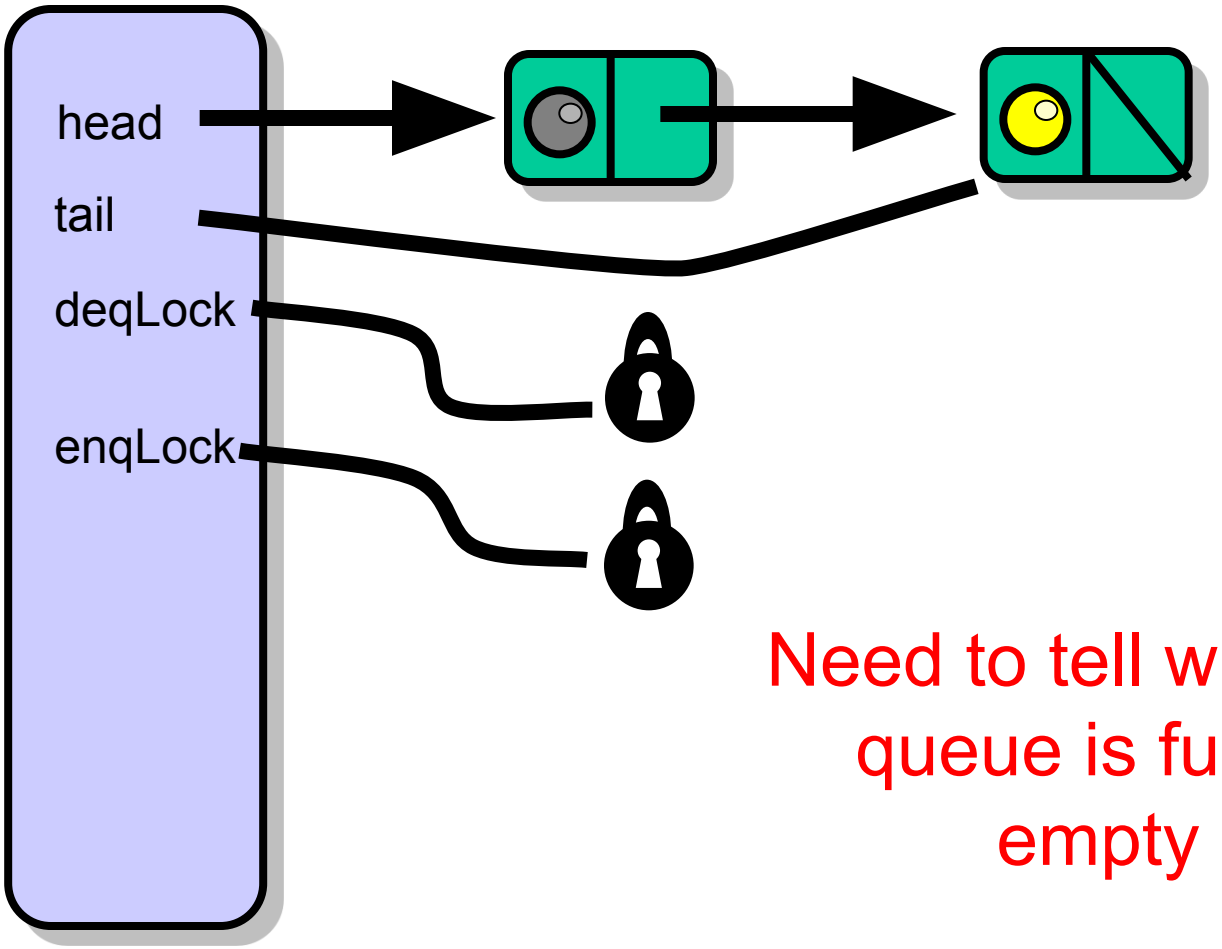
Bounded Queue



Bounded Queue

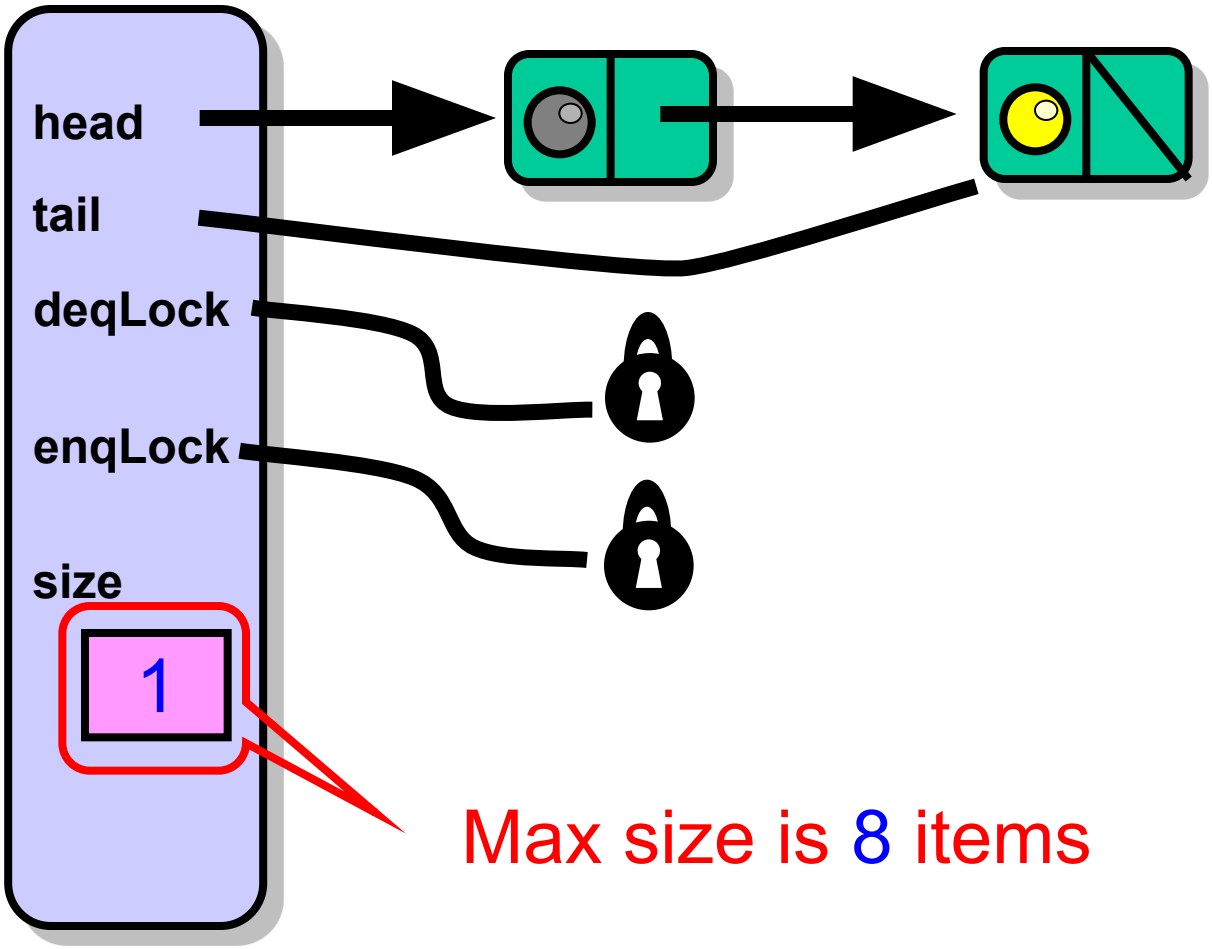


Not Done Yet

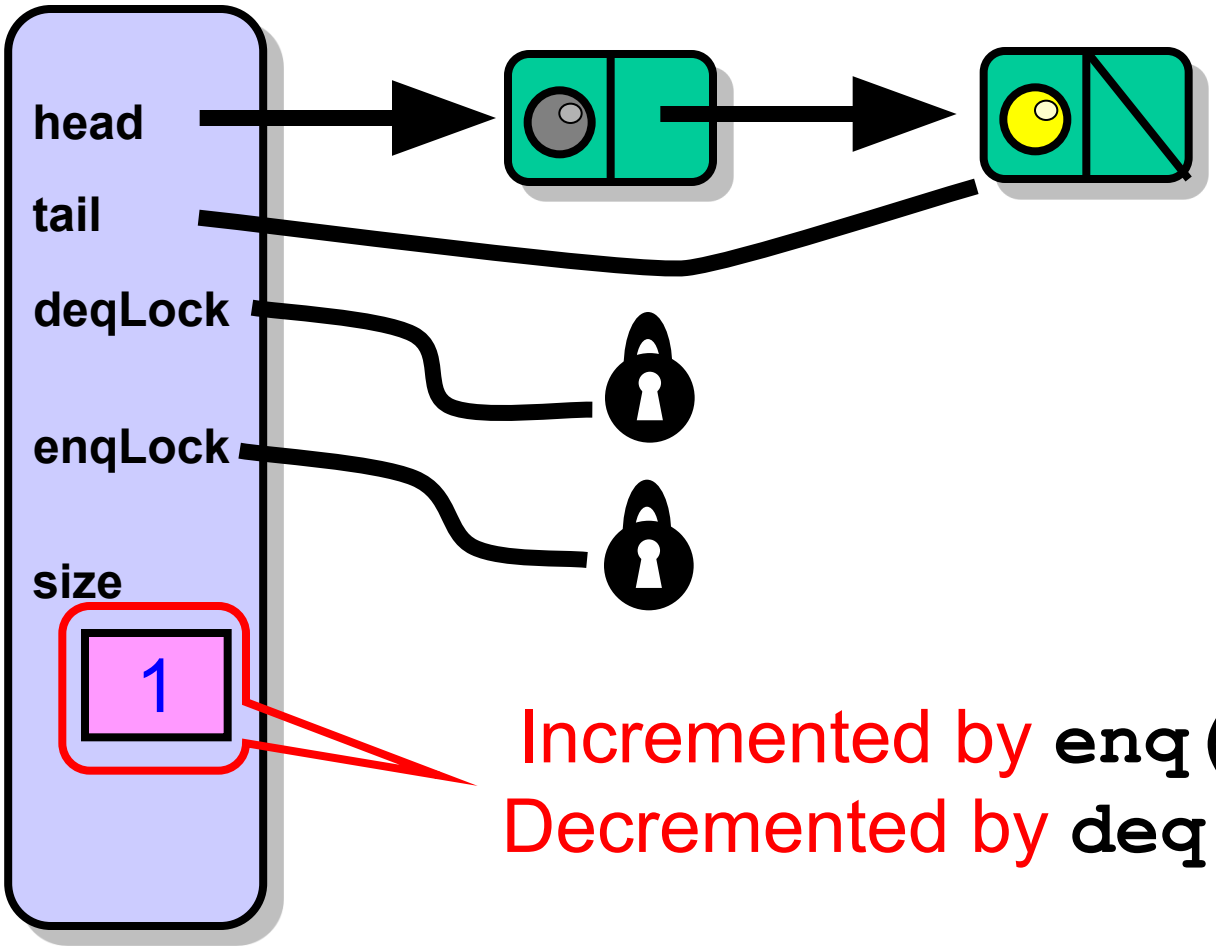


Need to tell whether queue is full or empty

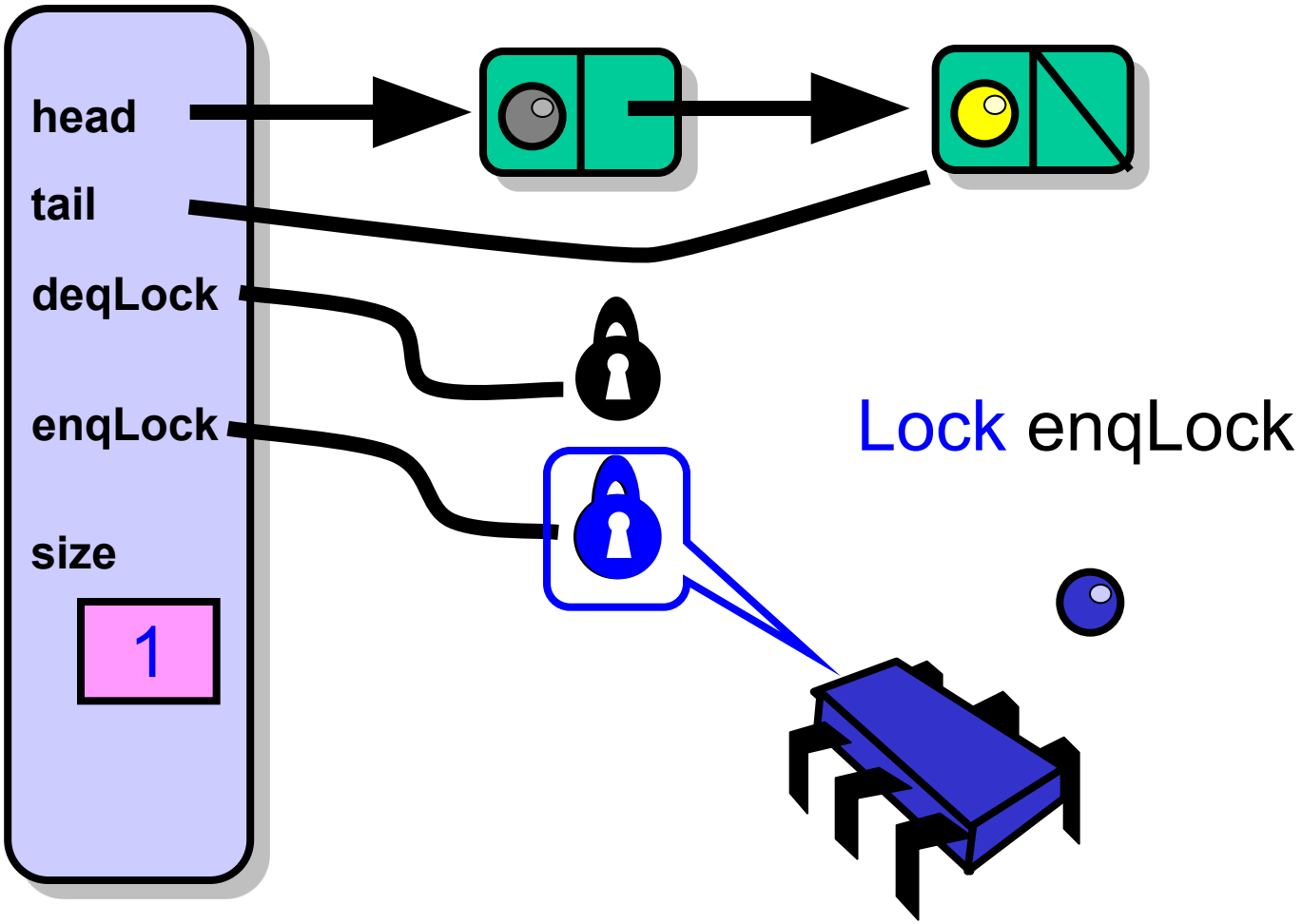
Not Done Yet



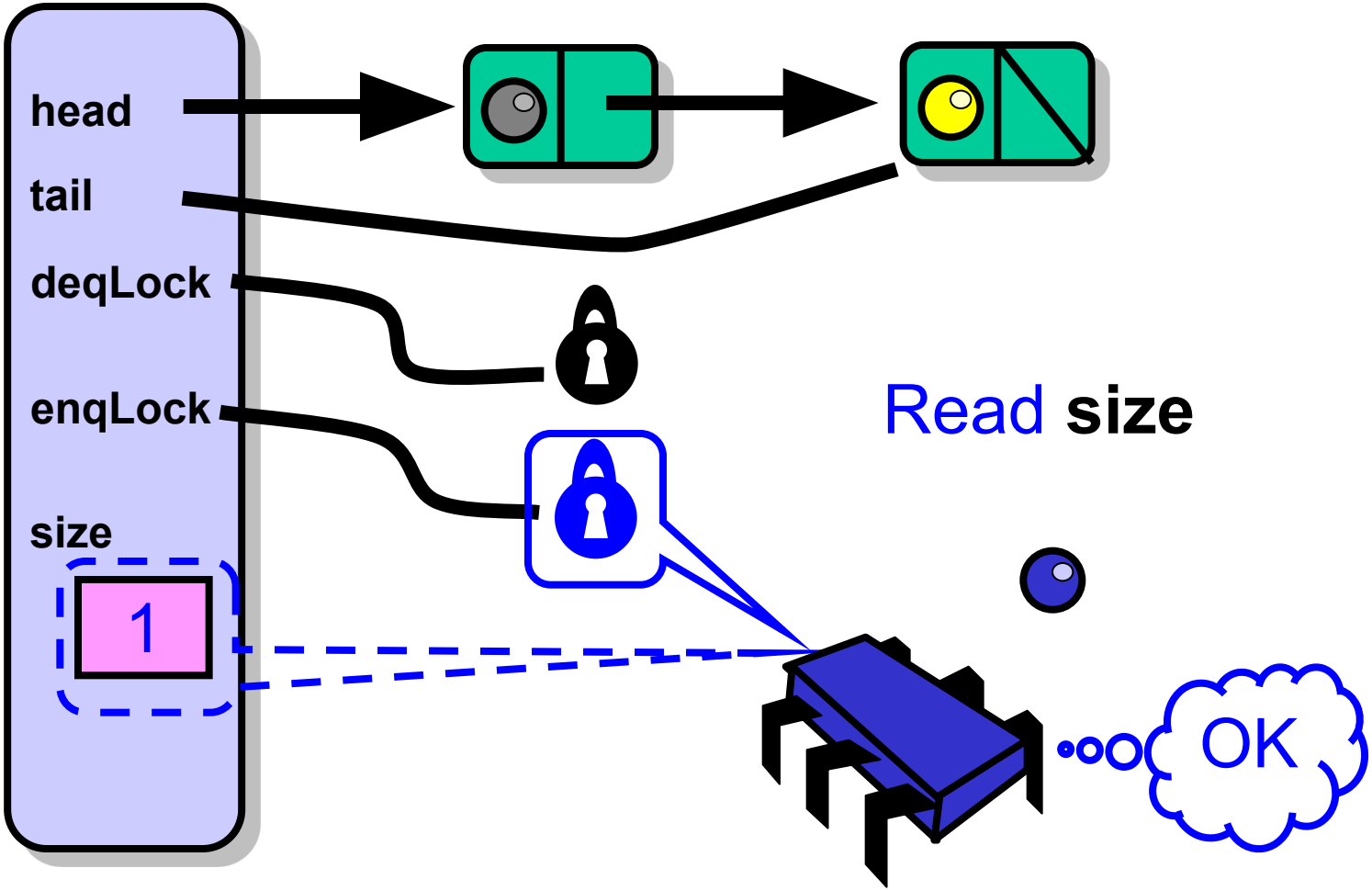
Not Done Yet


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

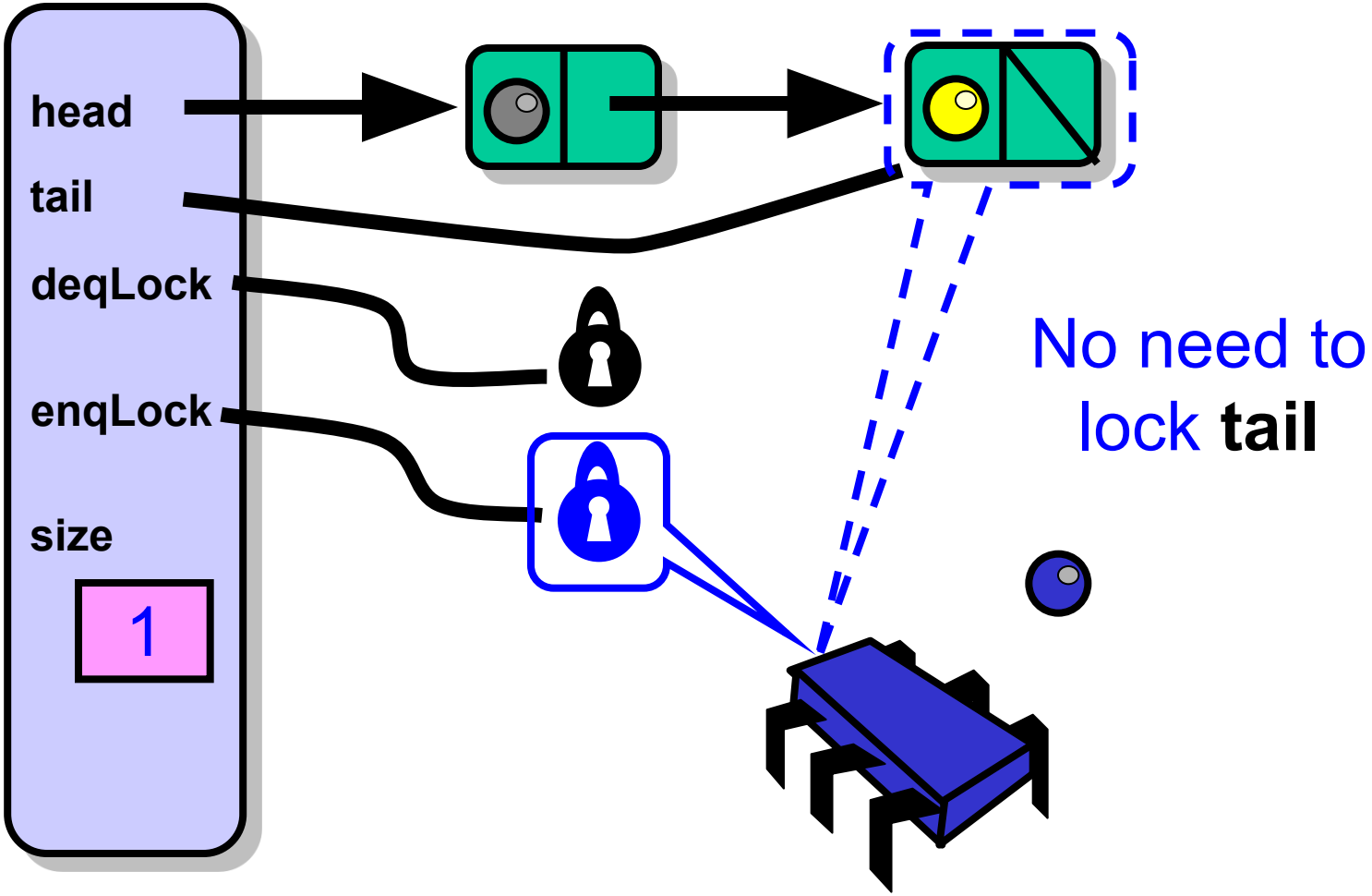
Enqueuer



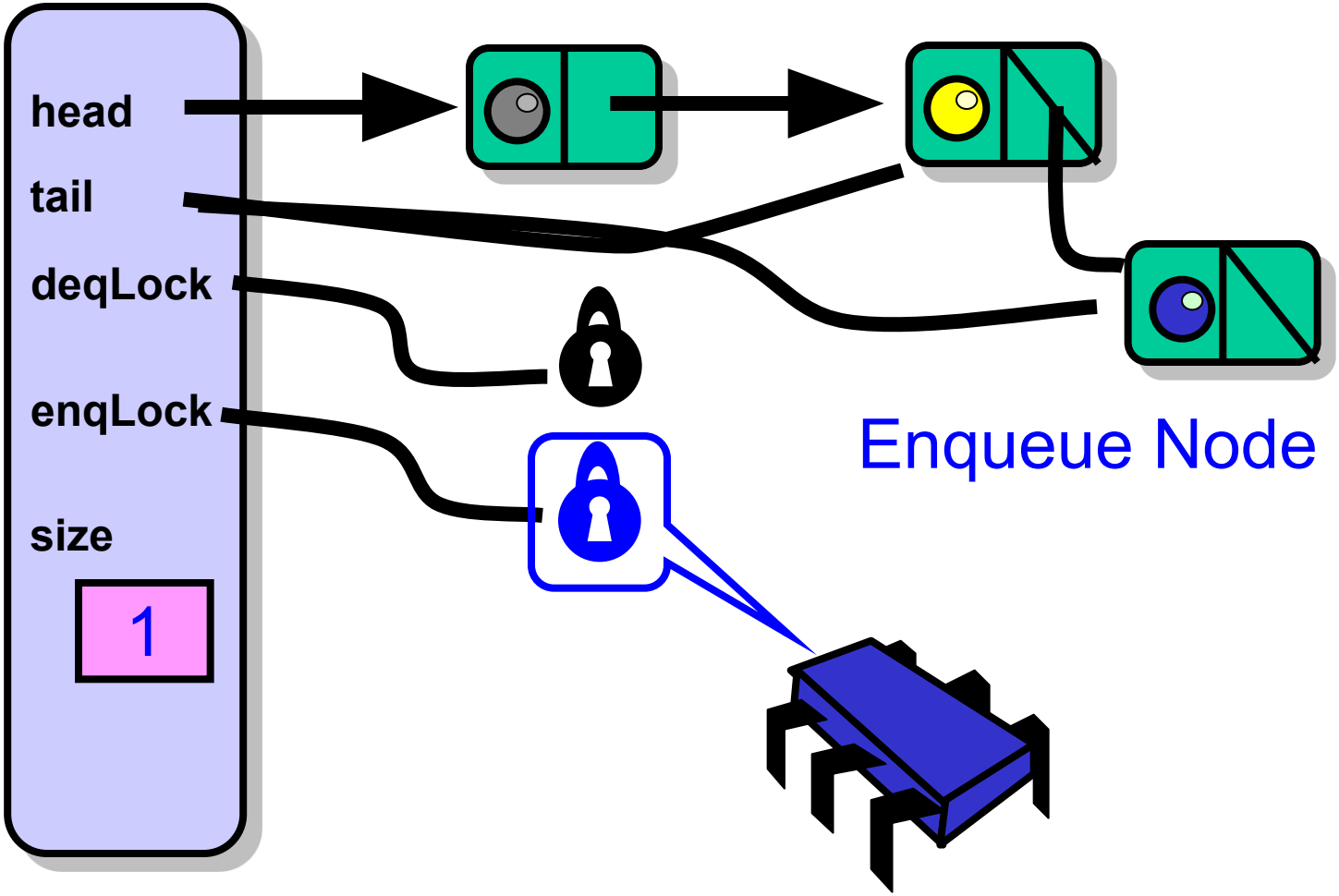
Enqueuer



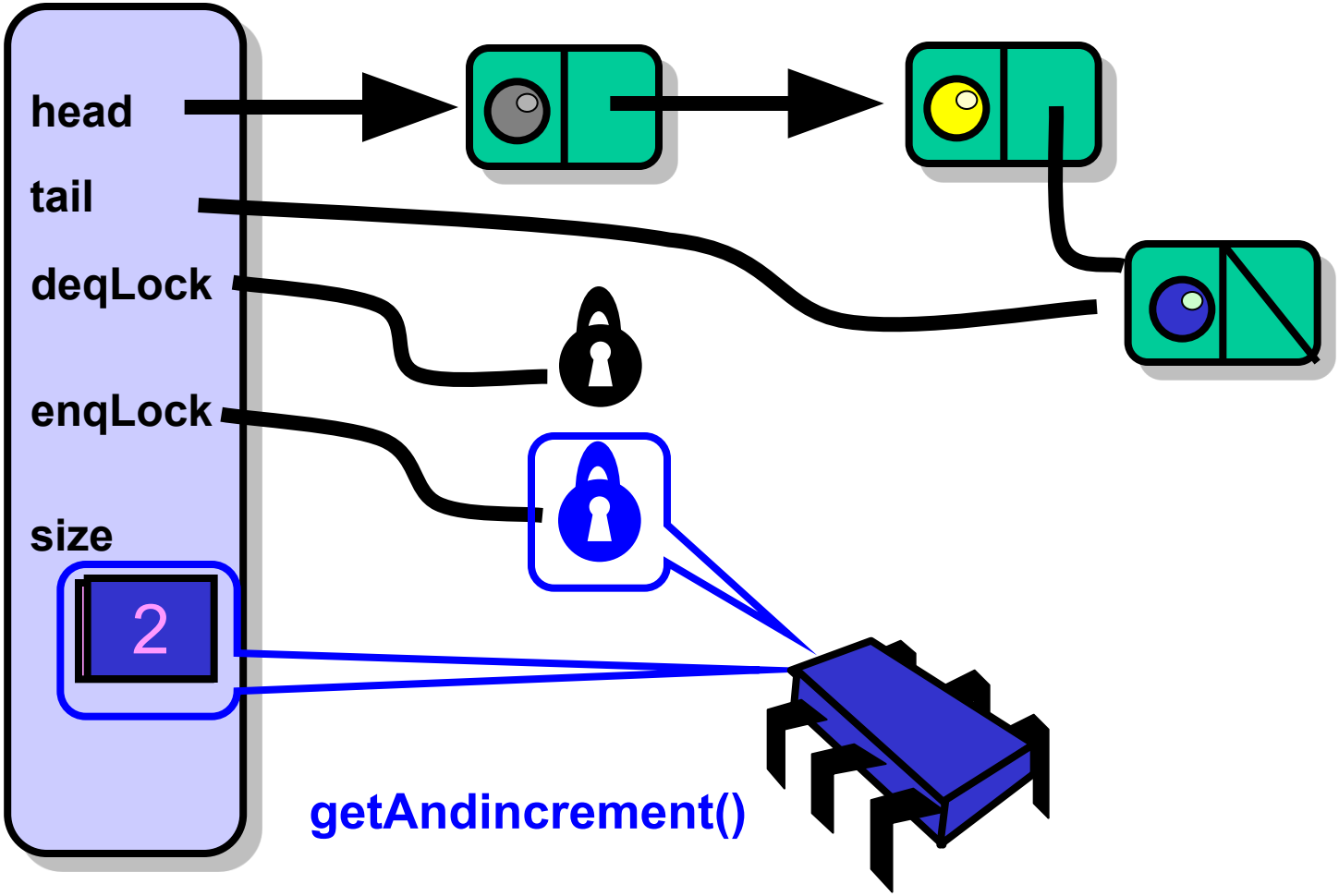
Enqueuer



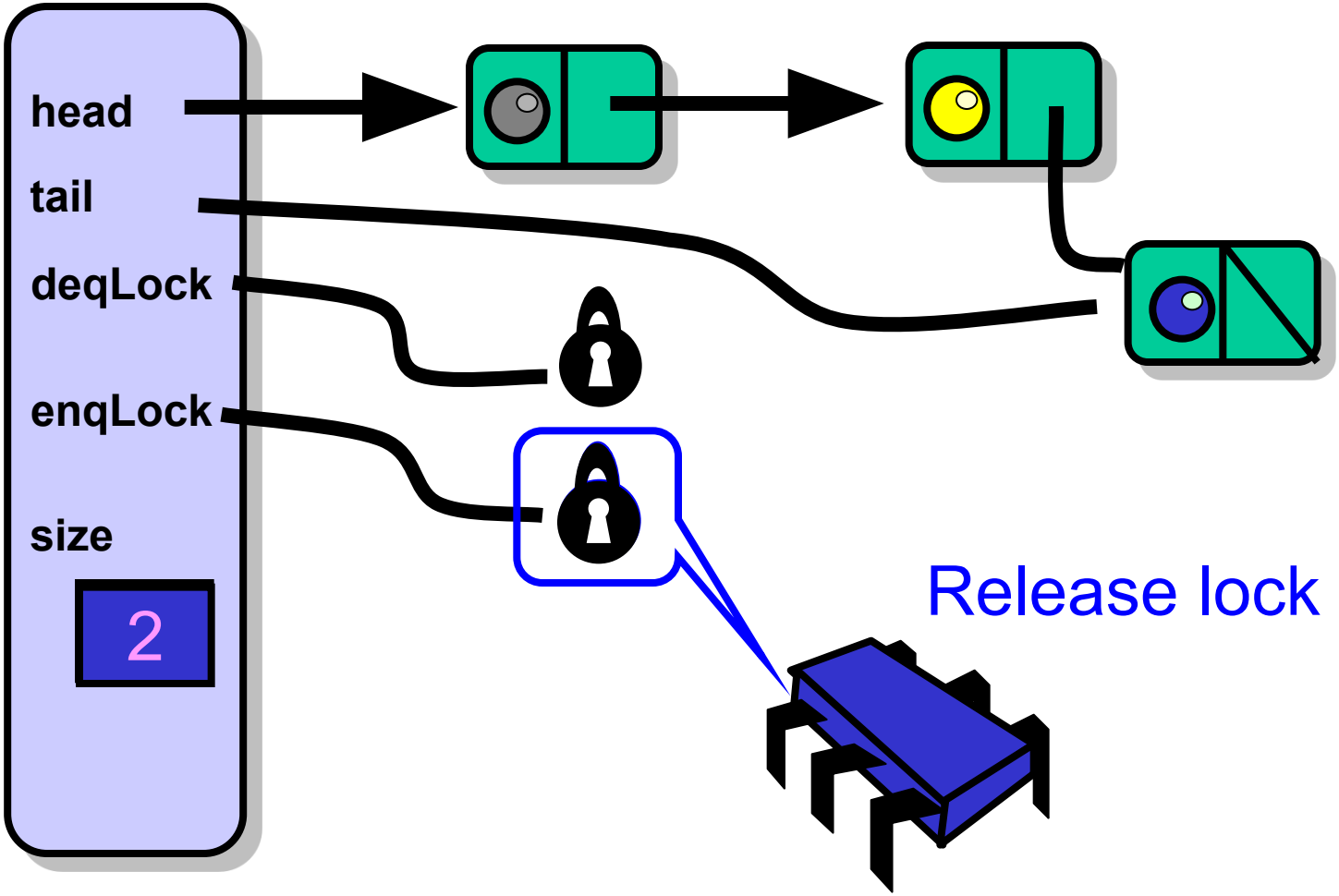
Enqueuer



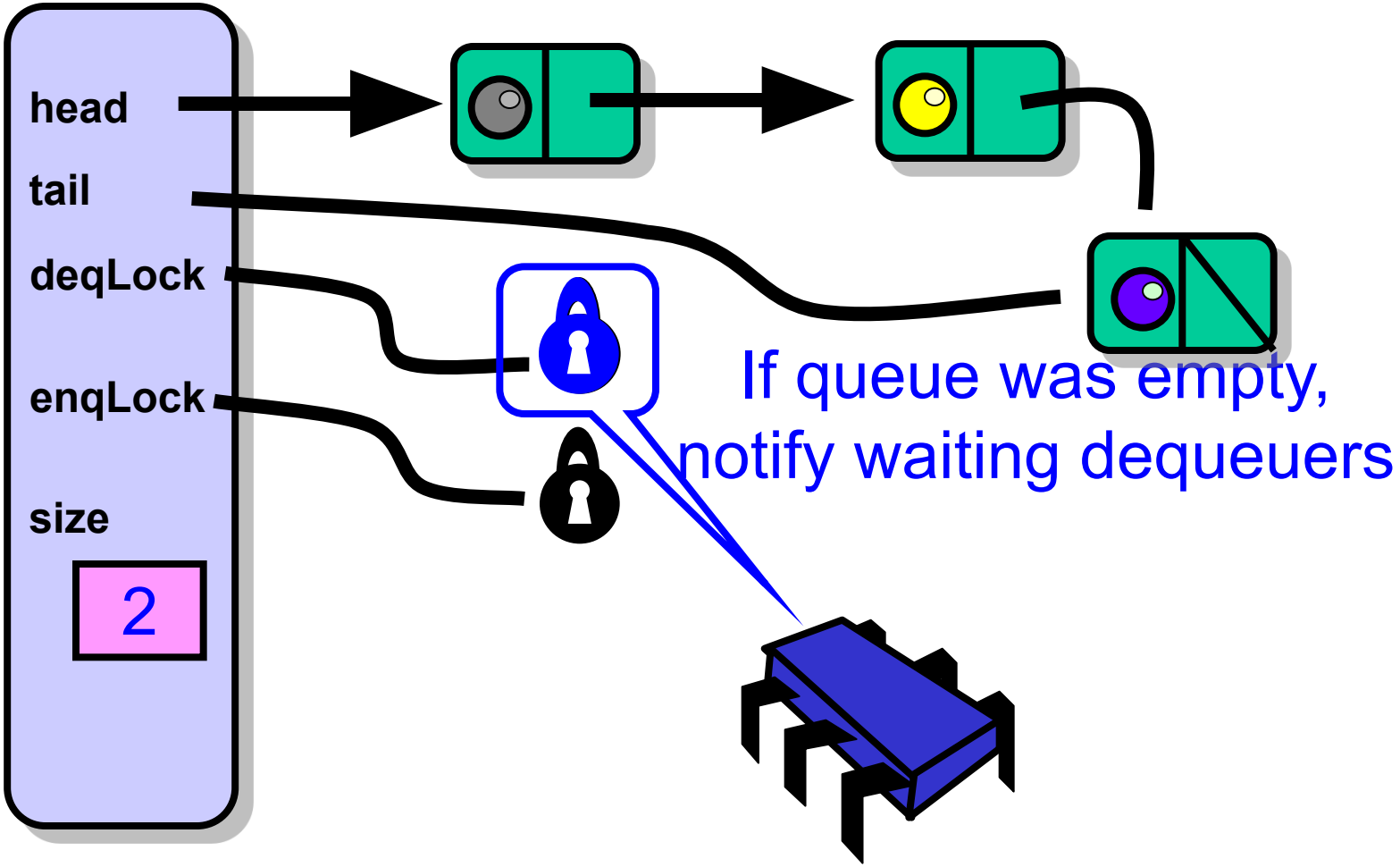
Enqueuer



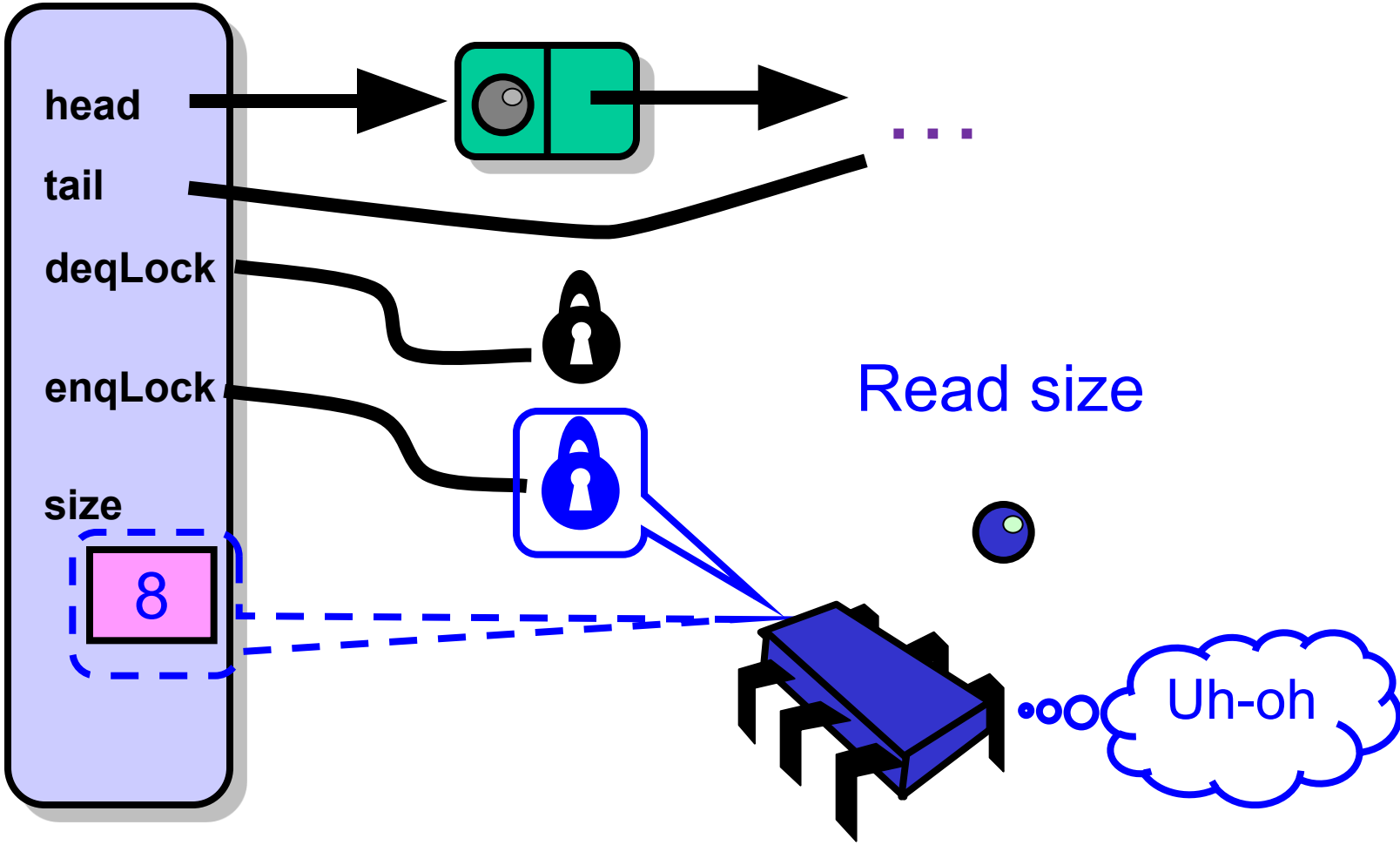
Enqueuer



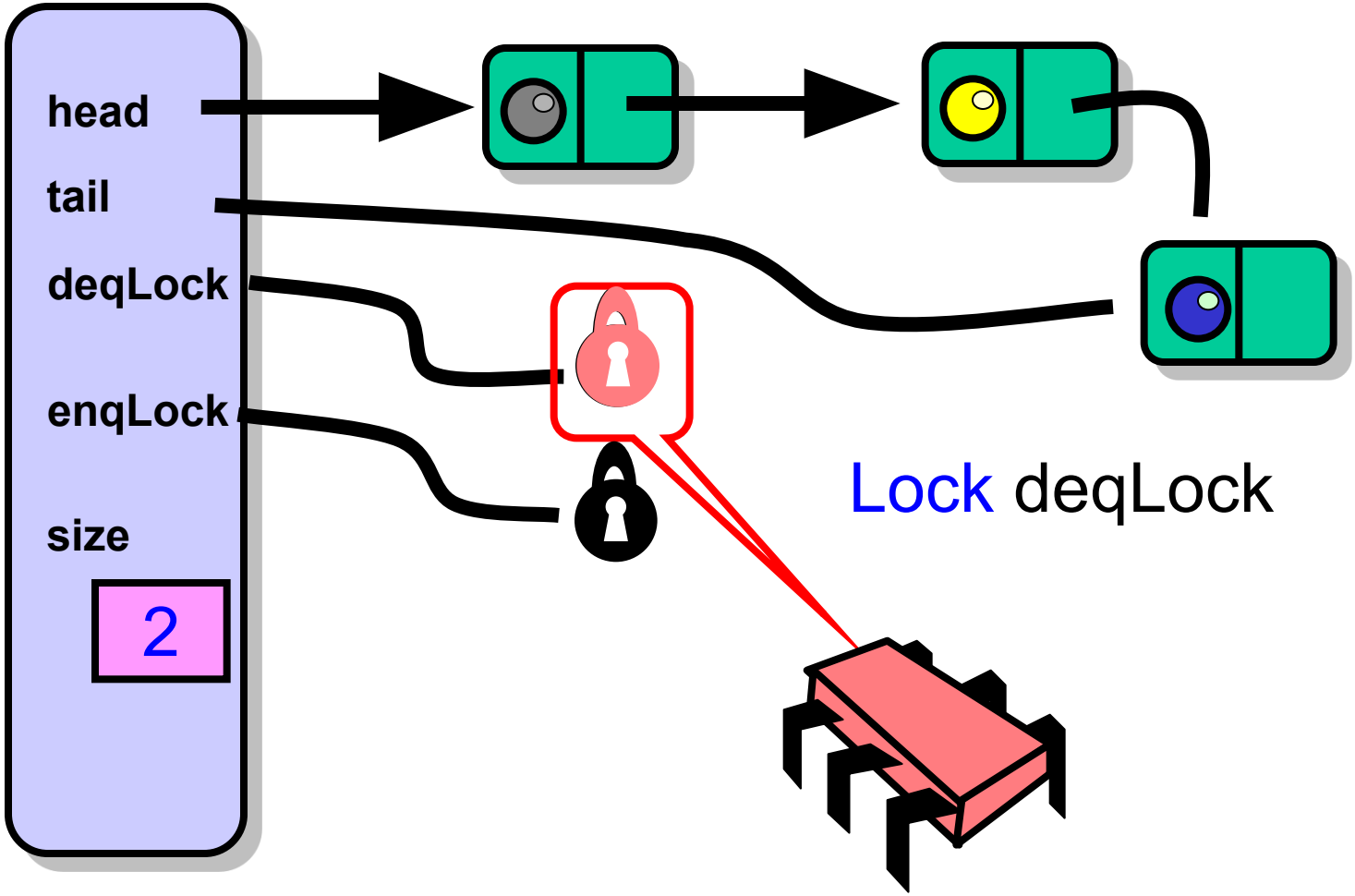
Enqueuer


head
tail
deqLock
enqLock
size
2
If queue was empty,
notify waiting dequeuers

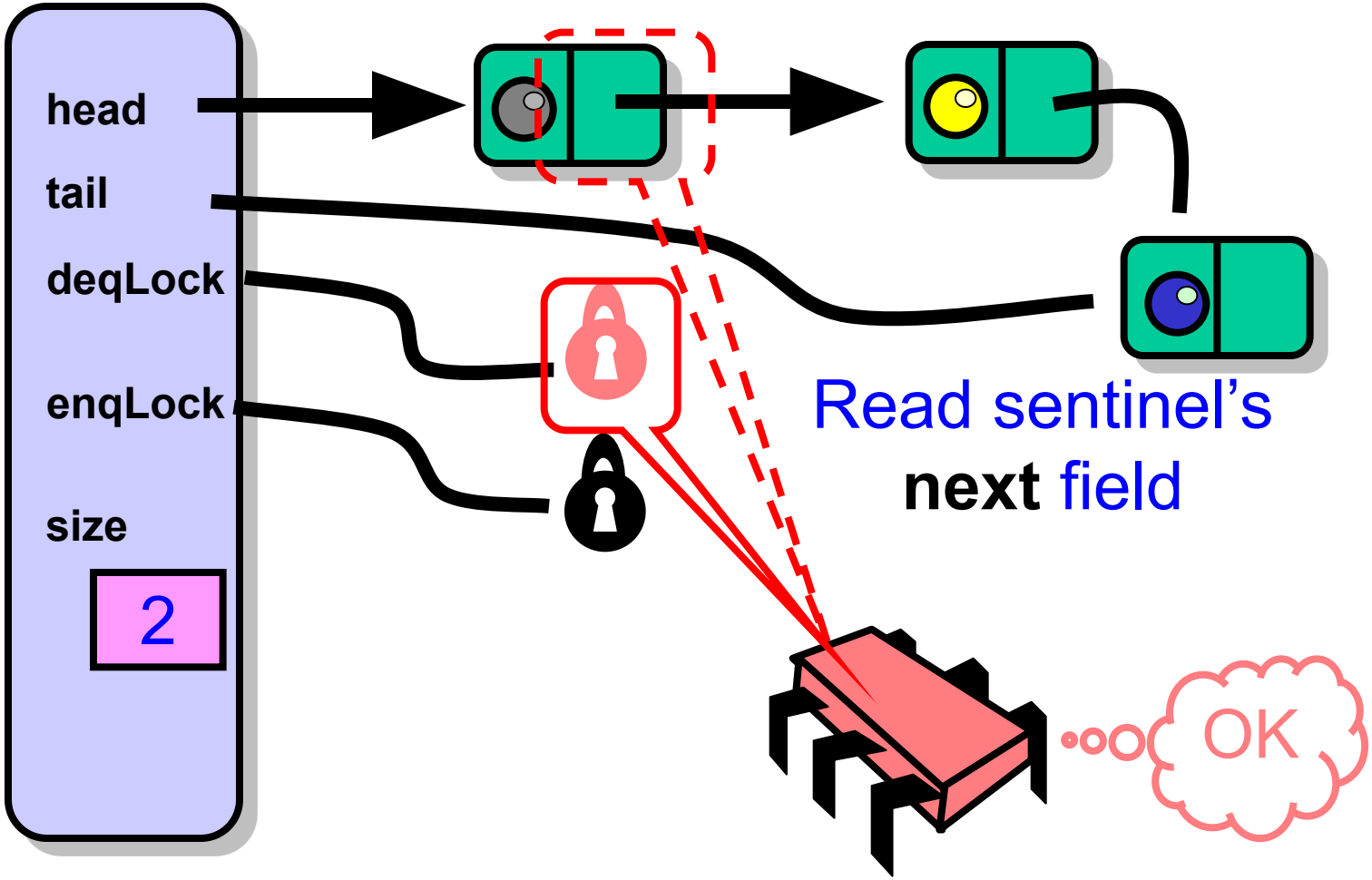
Unsuccesful Enqueuer



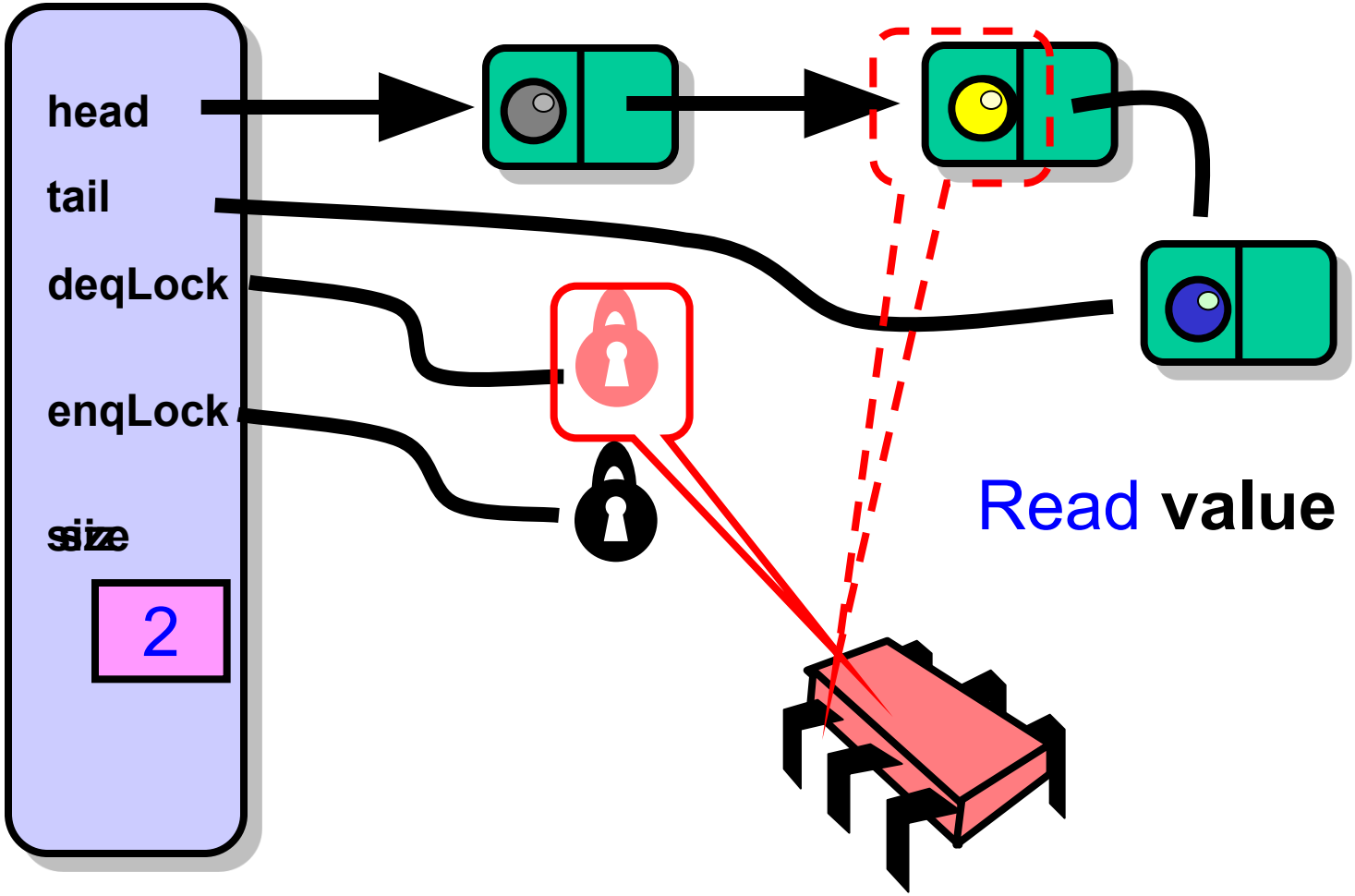
Dequeuer



Dequeuer

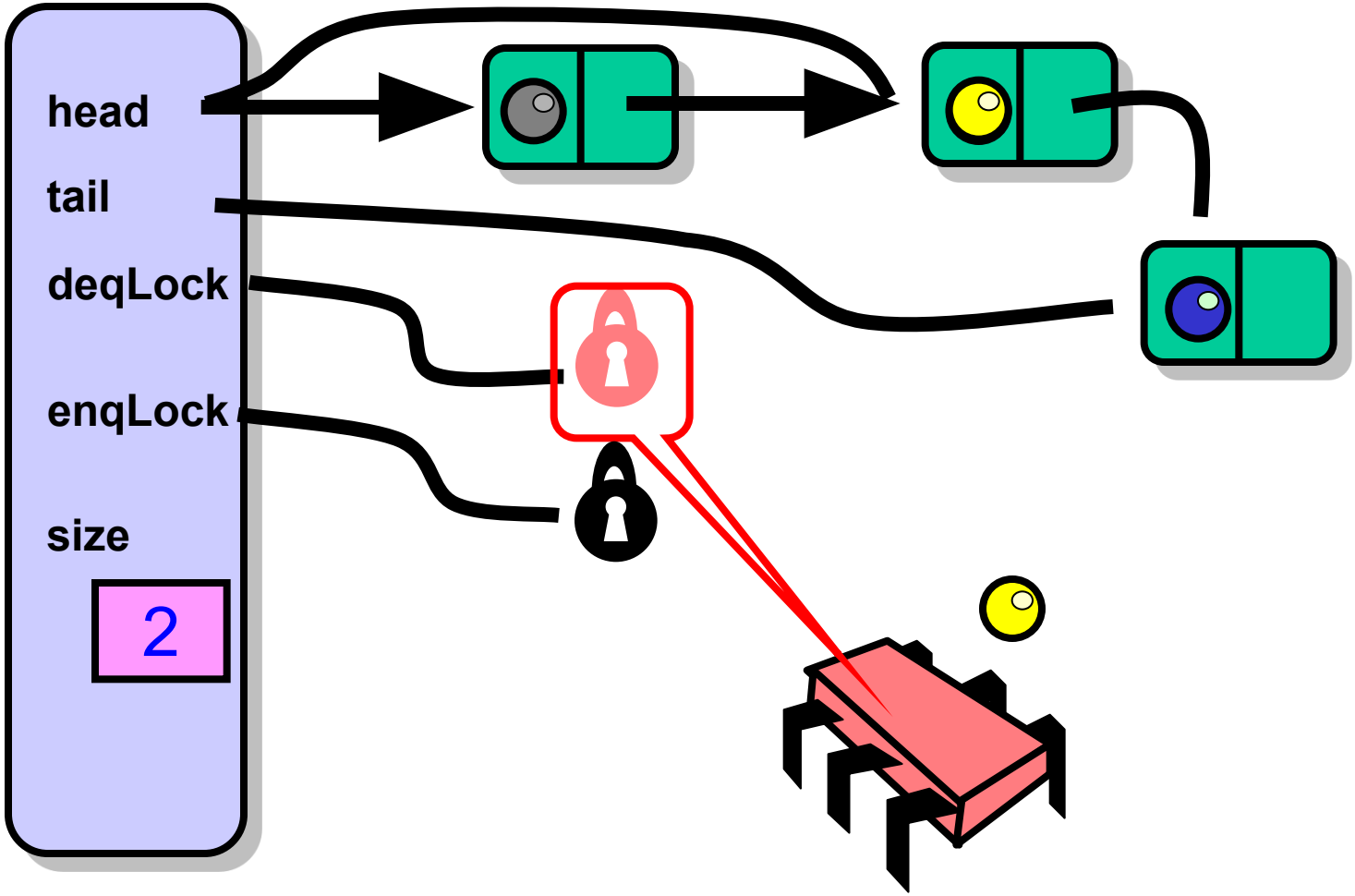

head
tail
deqLock
enqLock
size
2
Read sentinel's
next field
OK

Dequeuer

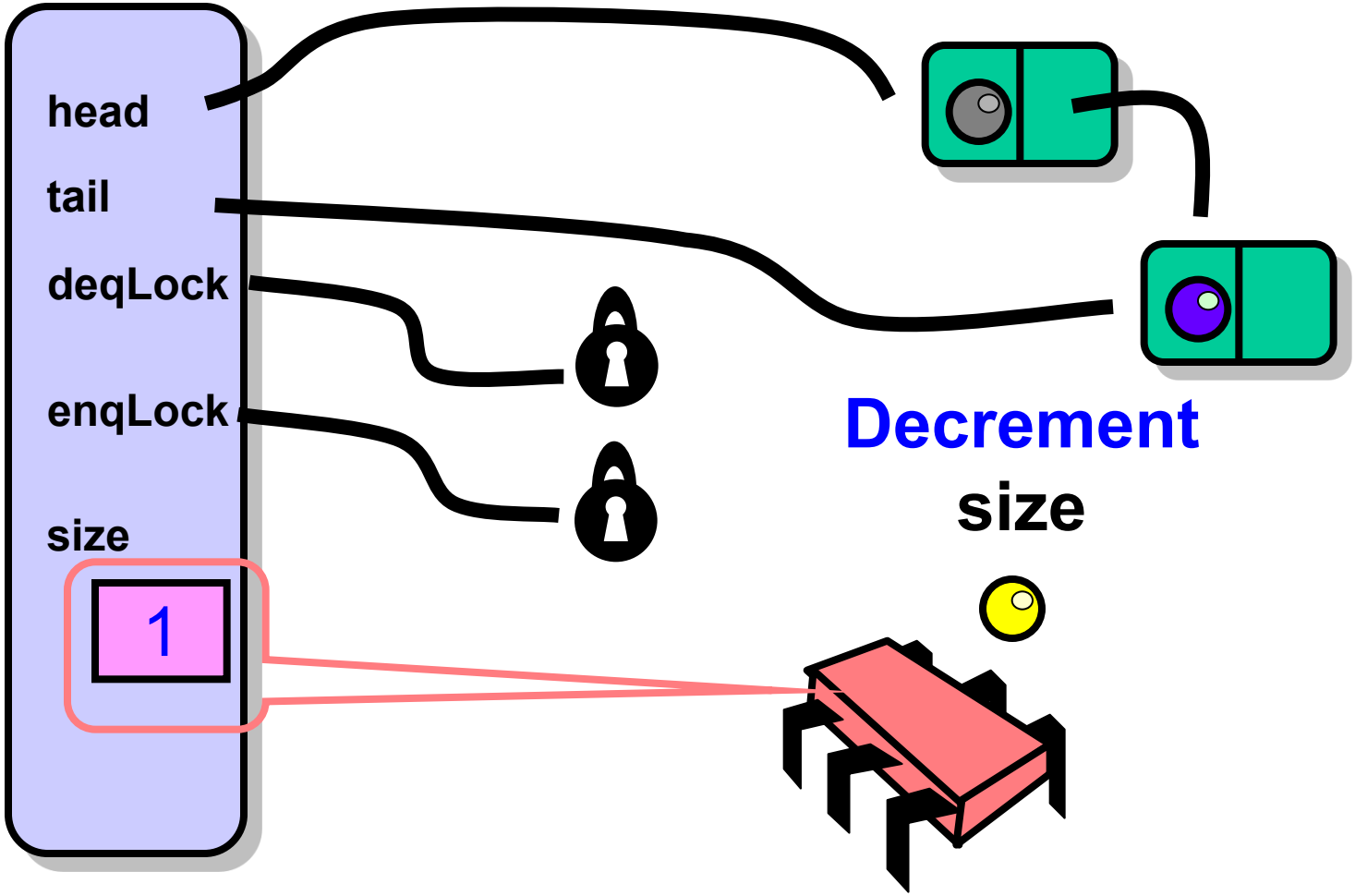


Make first Node
new sentinel

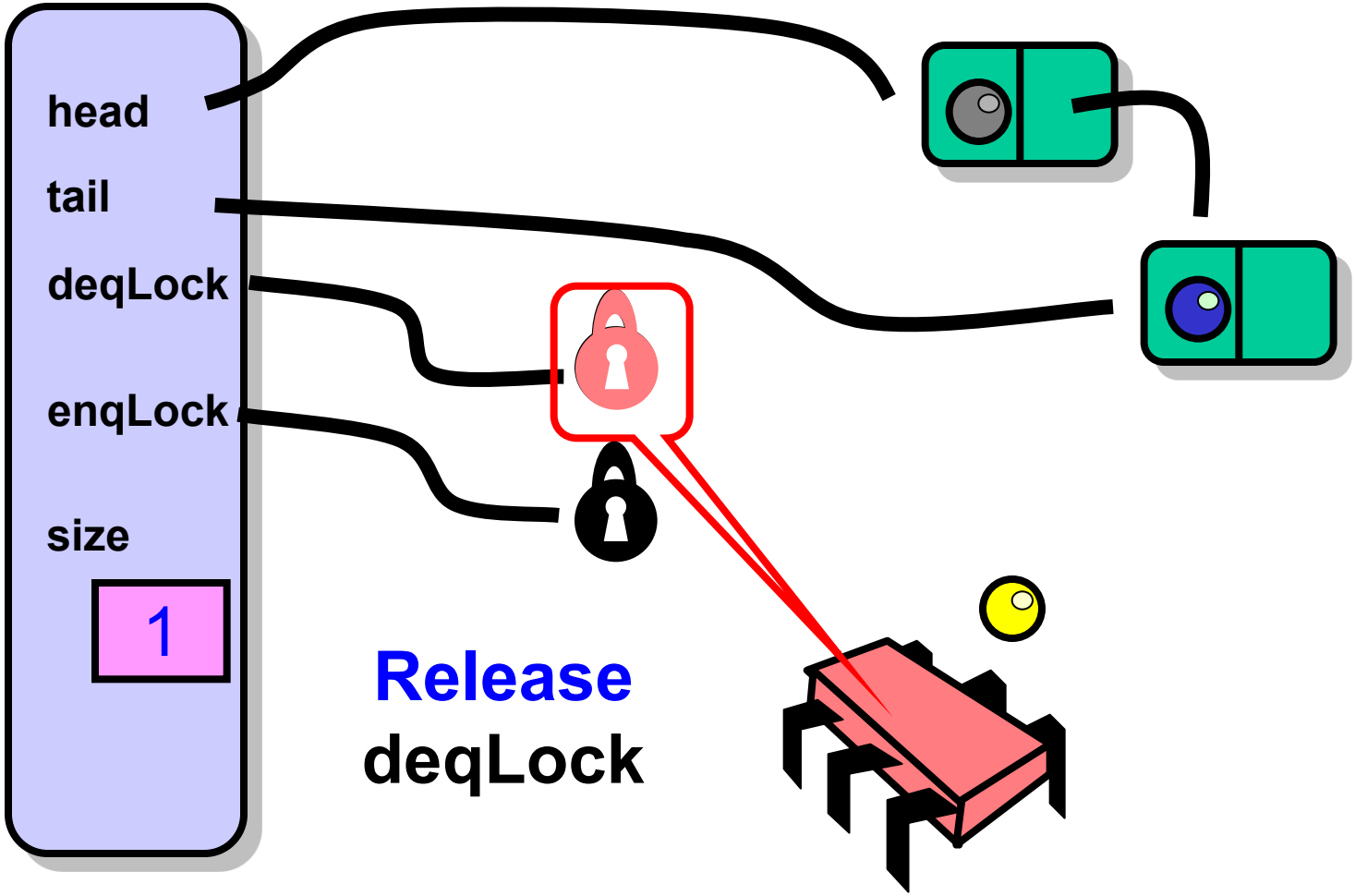
Dequeuer



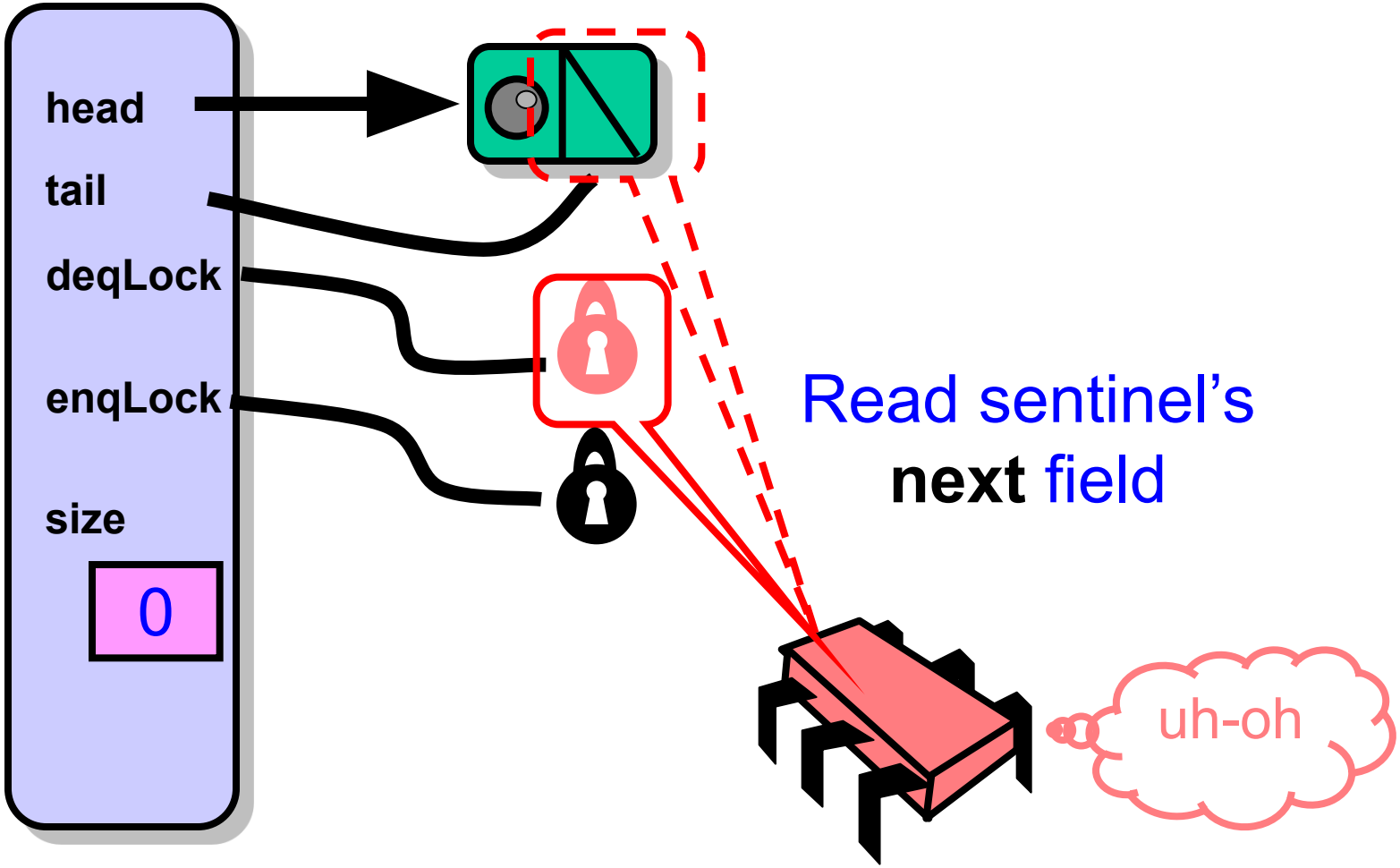
Dequeuer



Dequeuer



Unsuccesful 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

(push)

Digression: Monitor Locks

- Java `synchronized` objects and `ReentrantLocks` are monitors
- Allow blocking on a condition rather than spinning
- Threads:
  - acquire and release lock
  - wait on a condition

The Java Lock Interface

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

Acquire lock

The Java Lock Interface

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

Release lock

The Java Lock Interface

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

Try for lock, but not too hard

The Java Lock Interface

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

Create condition to wait on

The Java Lock Interface

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

Never mind what this method does

Lock Conditions

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

Lock Conditions

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

Release lock and wait on condition

Lock Conditions

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

Wake up one waiting thread

Lock Conditions

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

Wake up all waiting threads

Await

```
q.await()
```

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

Signal

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

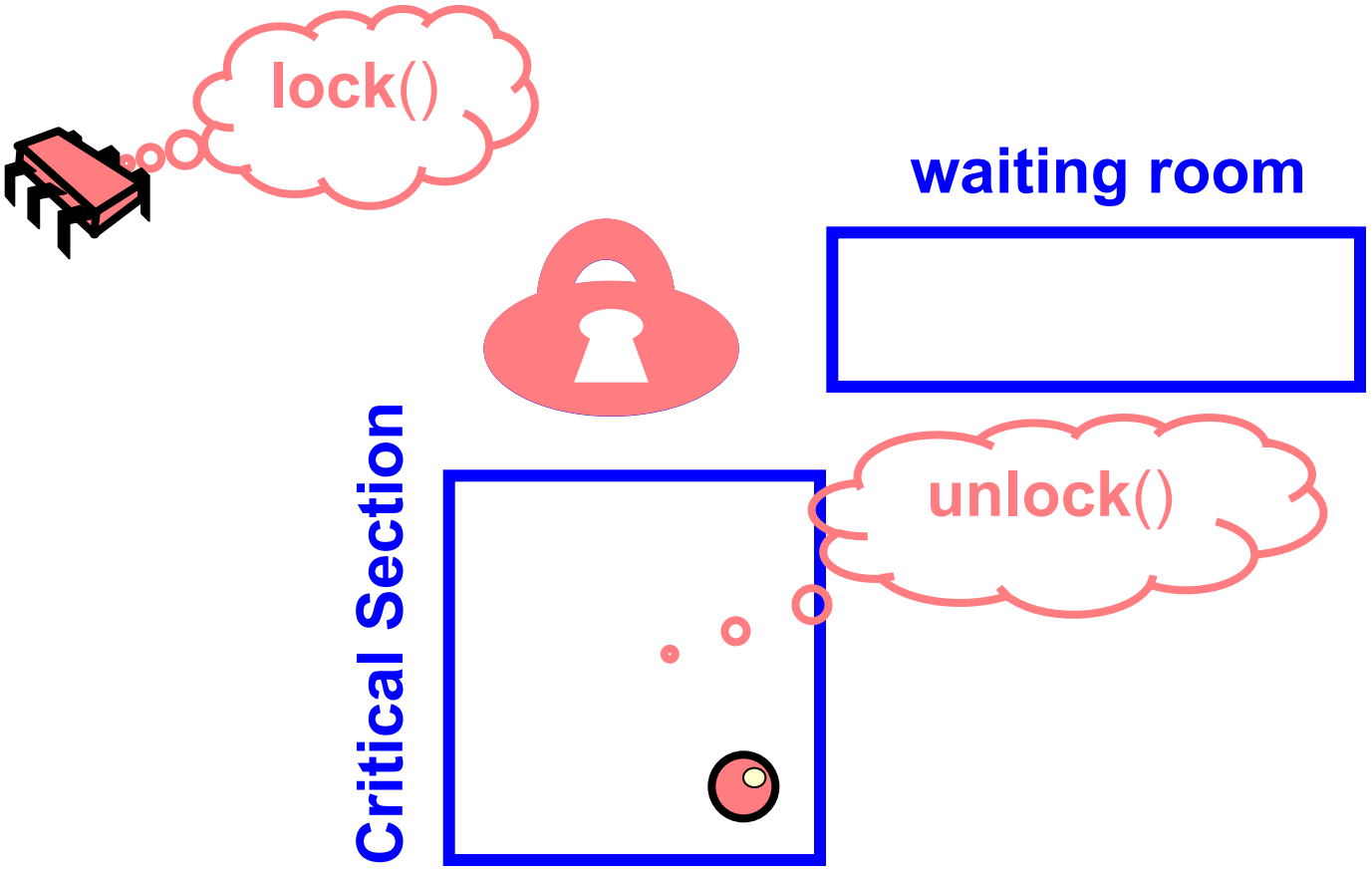
- Awakens one waiting thread
  - Which will reacquire lock

Signal All

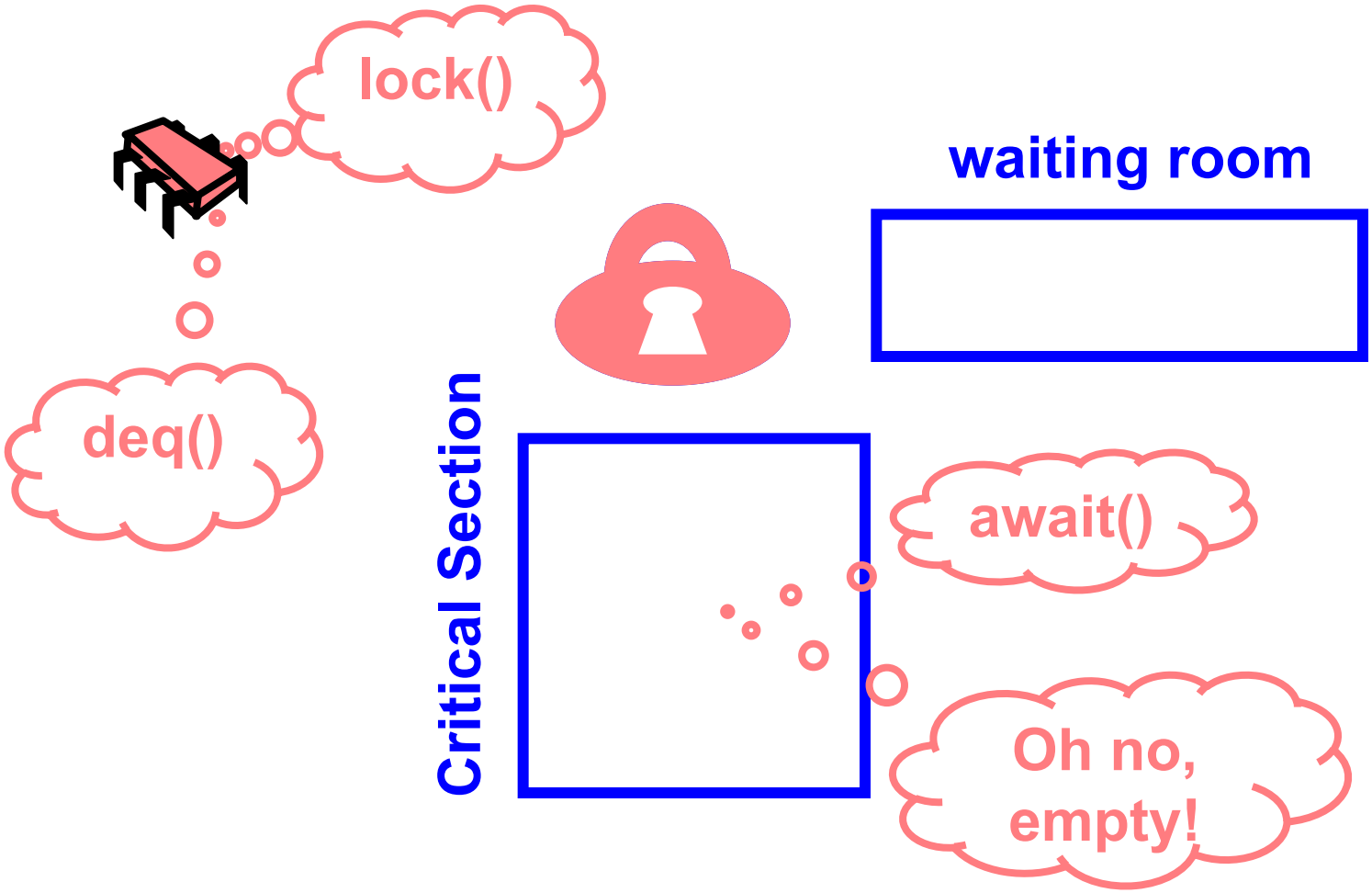
```
q.signalAll();
```

- Awakens all waiting threads
  - Which will each reacquire lock

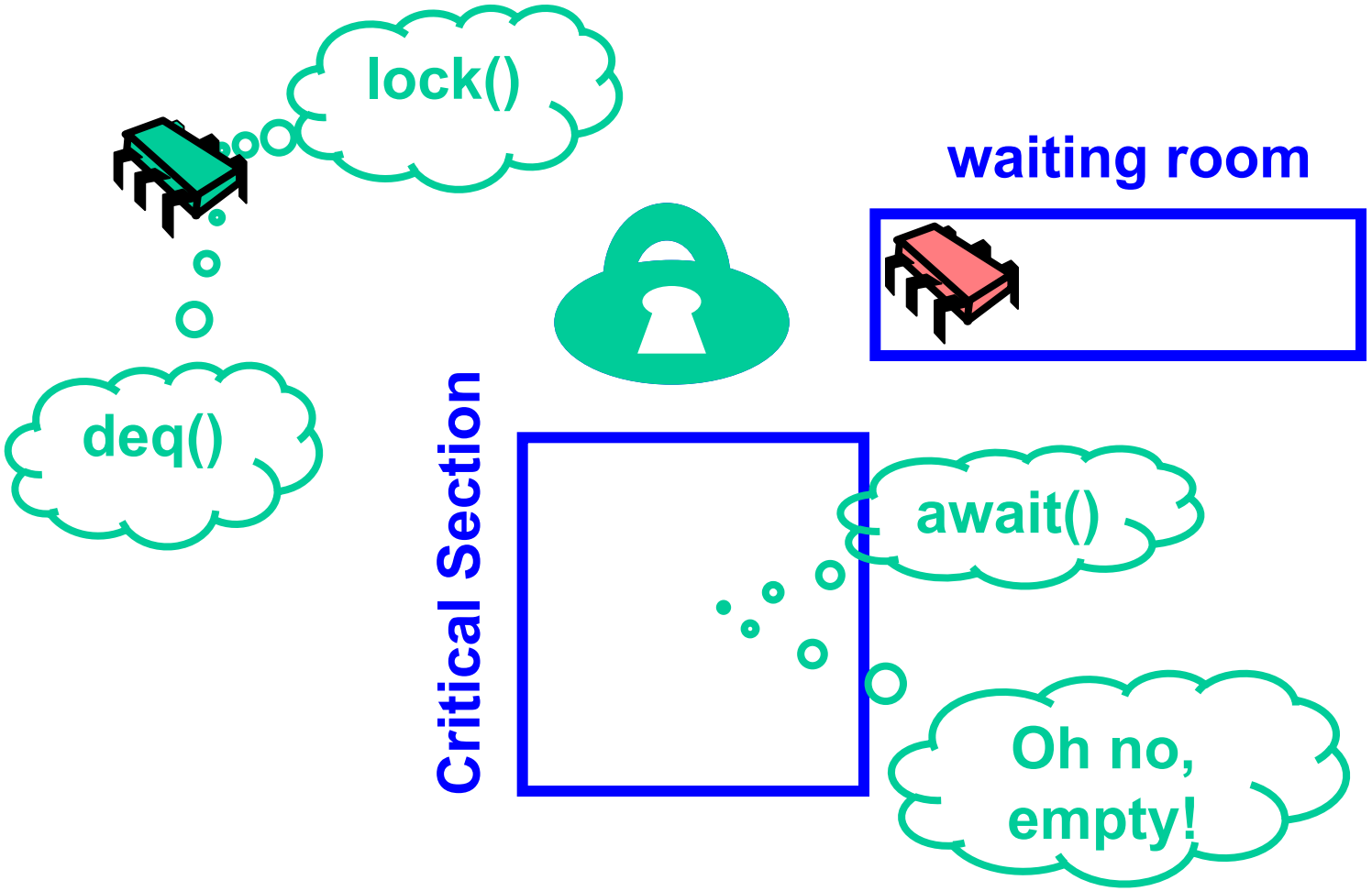
A Monitor Lock



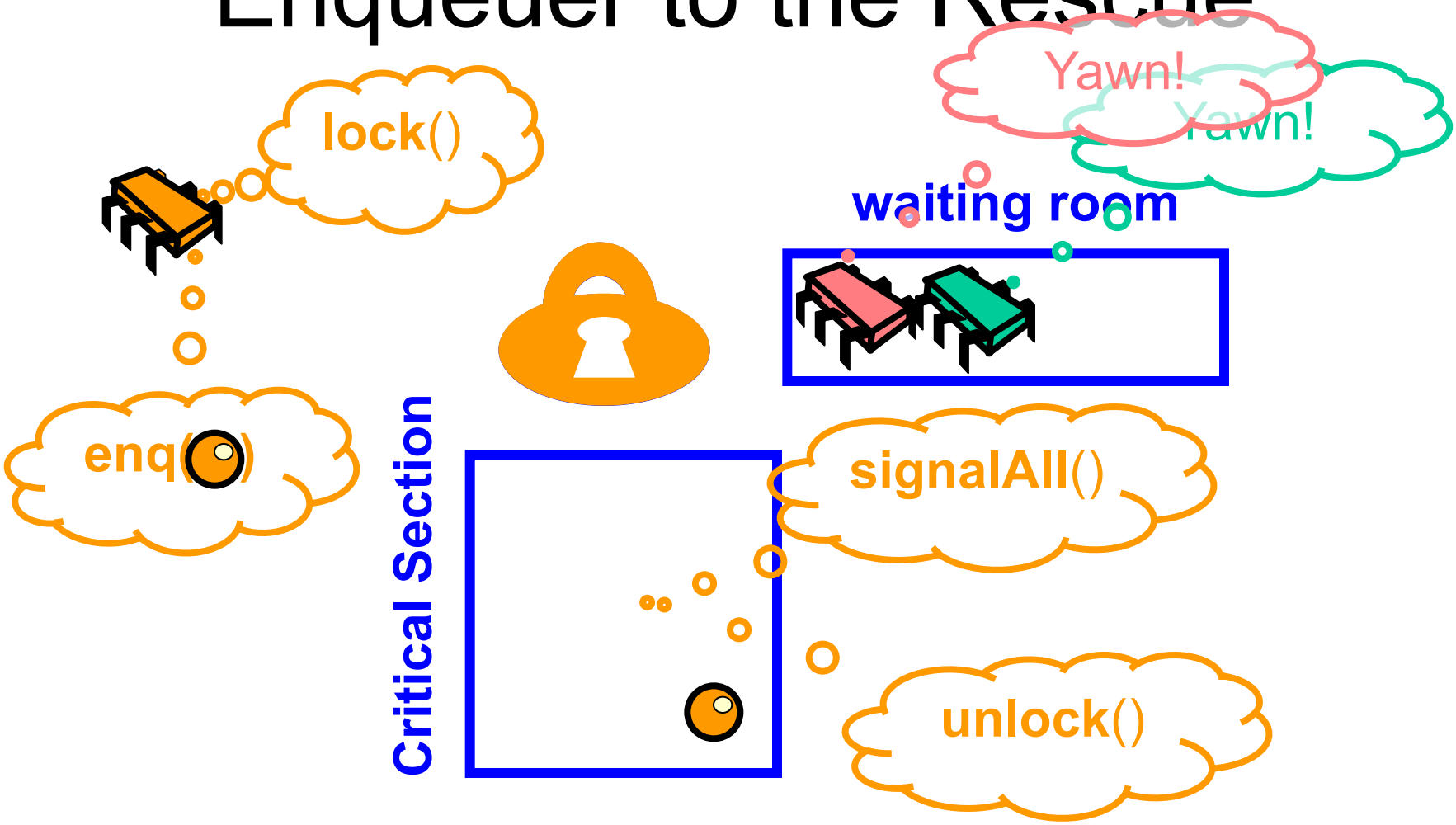
Unsuccessful Deq



Another One



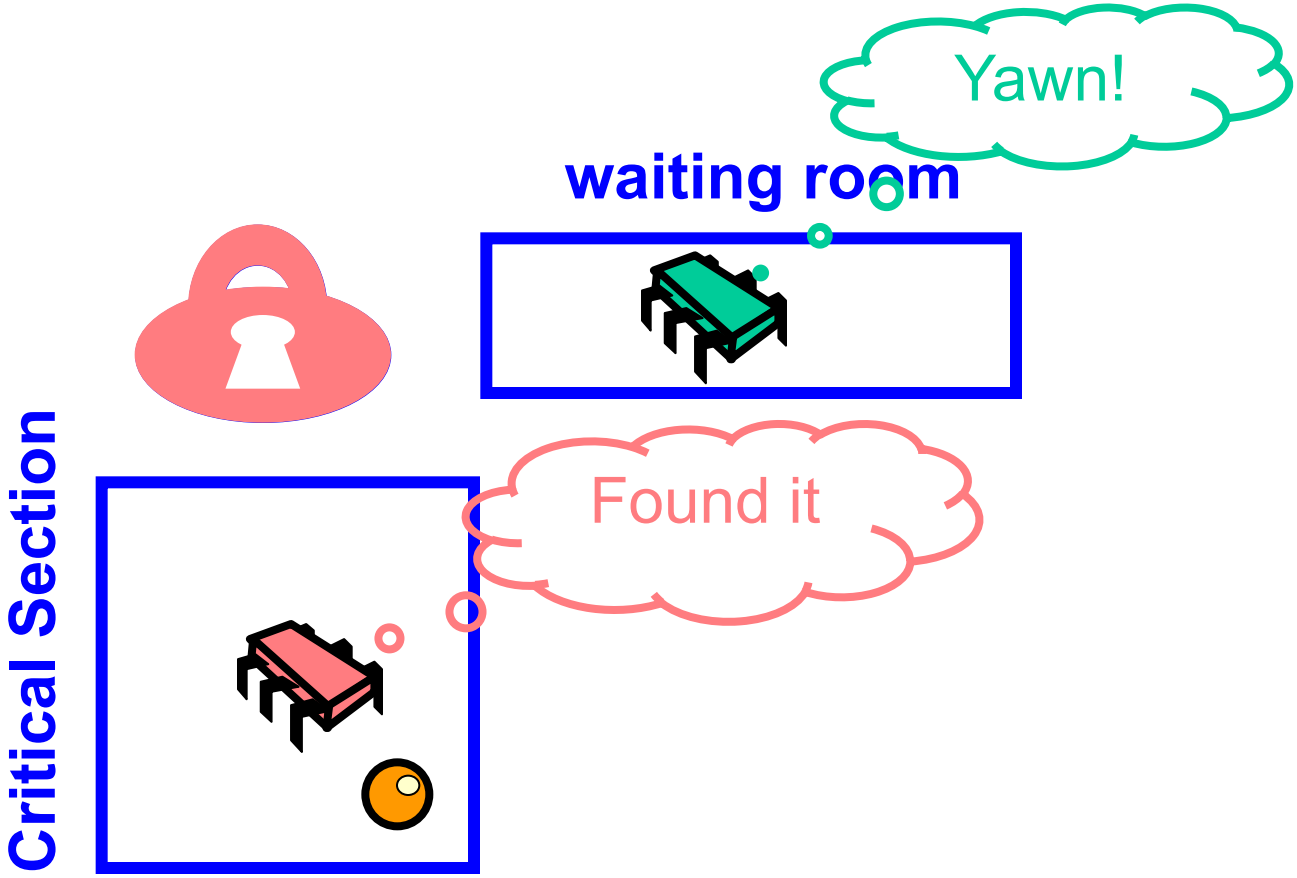
Enqueuer to the Rescue



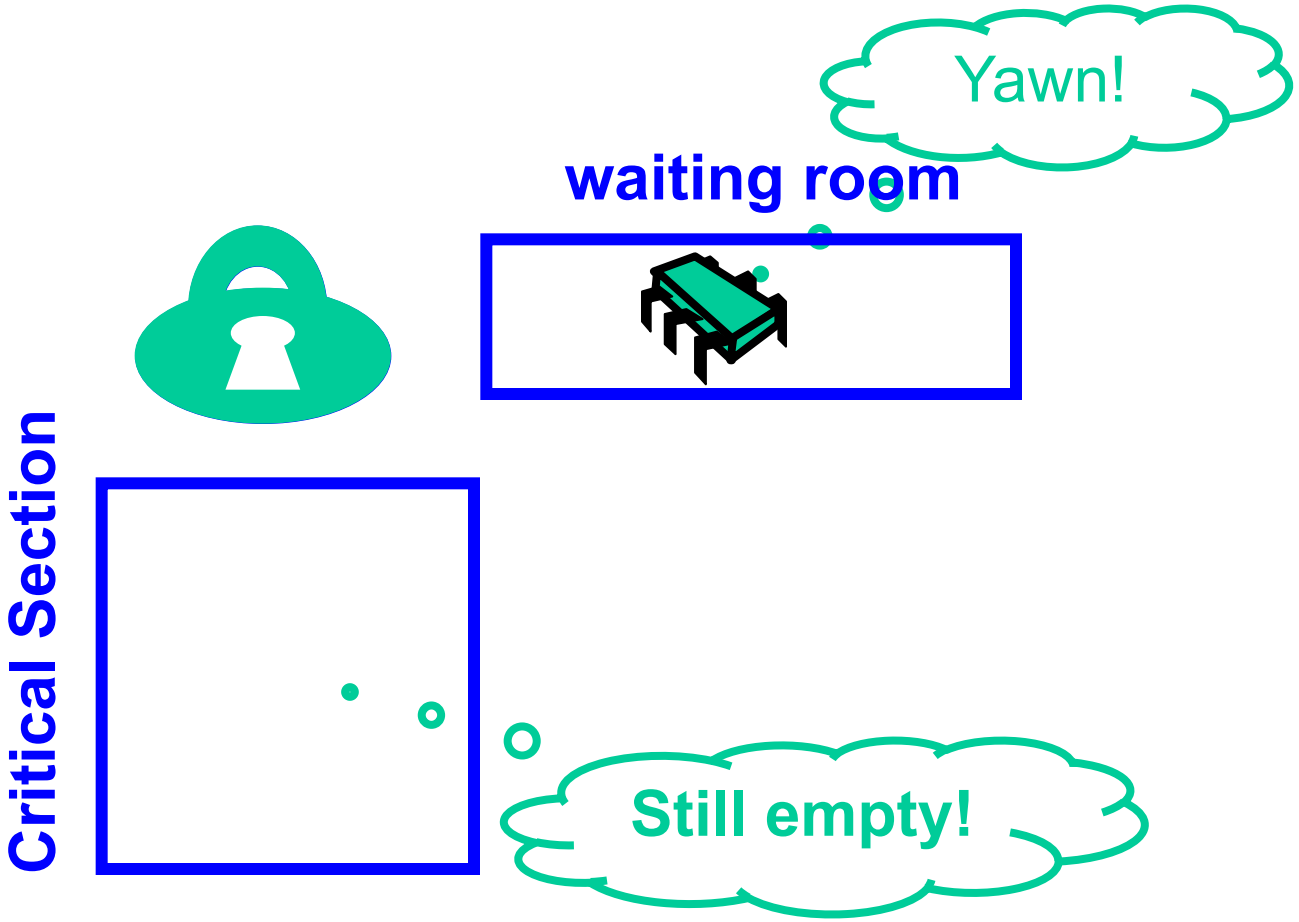
Monitor Signalling

Awakened thread might still lose lock to outside contender…

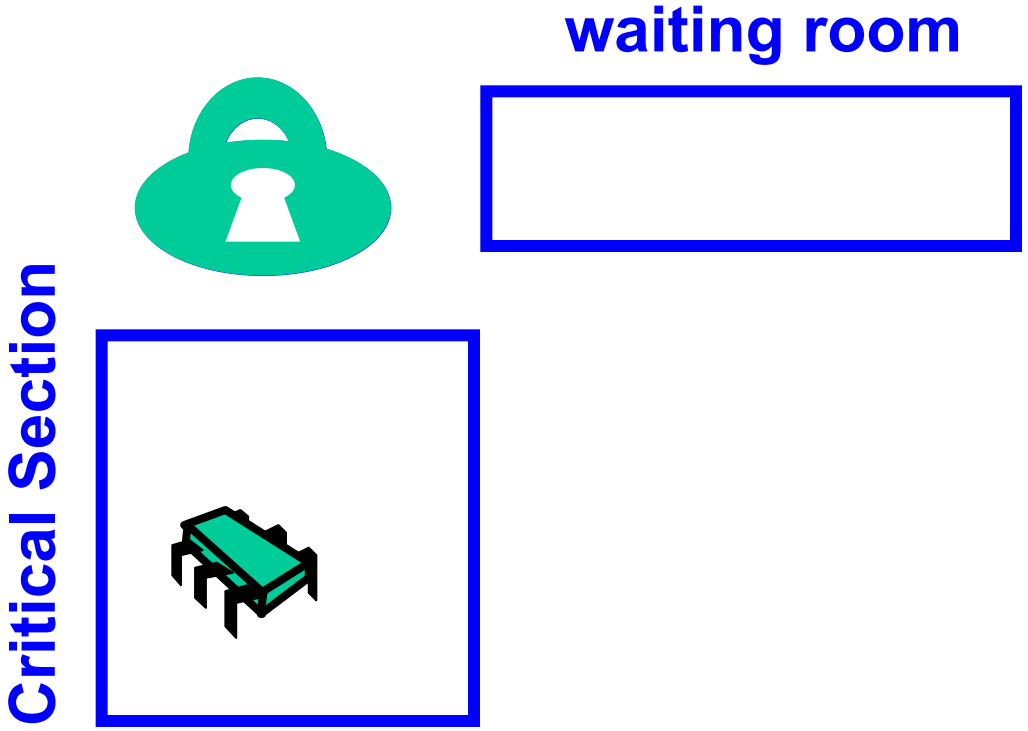
Dequeuers Signalled



Dequeuers Signaled



Dollar Short + Day Late



Java Synchronized Methods

```
public class Queue<T> {

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

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

Java Synchronized Methods

```
public class Queue<T> {

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

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

Each object has an implicit lock with an implicit condition

Java Synchronized Methods

```
public class Queue<T> {

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

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

Lock on entry, unlock on return

Java Synchronized Methods

```
public class Queue<T> {

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

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

Wait on implicit condition

Java Synchronized Methods

```
public class Queue<T> {

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

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

Signal all threads waiting on condition

(Pop!) The 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 Fields

```
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 Fields

```
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 lock's associated condition

Bounded Queue Fields

```
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();
}
```

size: 0 to capacity

Bounded Queue Fields

```
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();
}
```

Head and Tail

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();
 }
 …
}
```

Wait while queue is full …

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();
 }
 …
}
```

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

Be Afraid

```
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();
 }
 …
}
```

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

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();
 }
 …
}
```

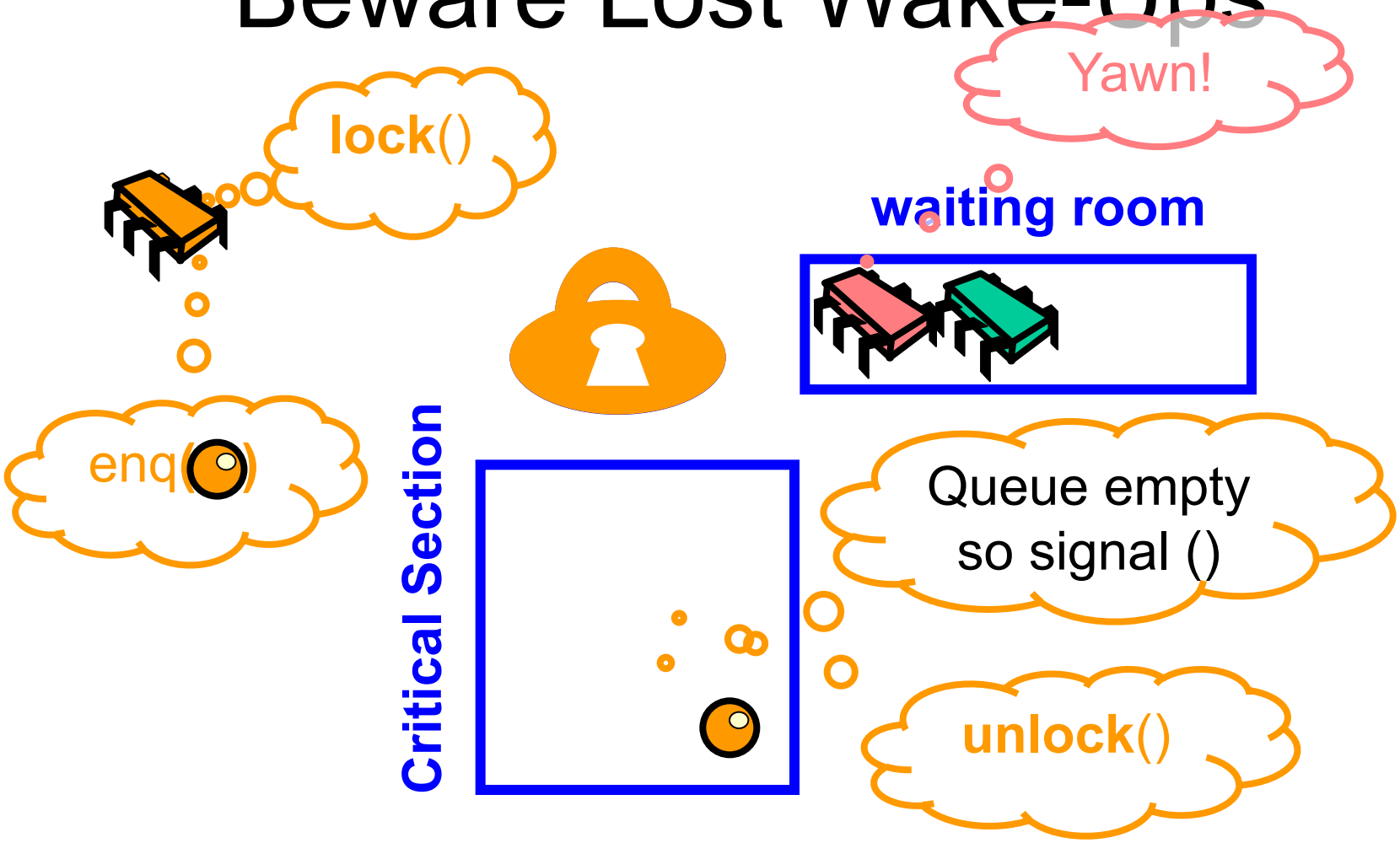
Add new node

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, wake frustrated dequeuers

Beware Lost Wake-Ups

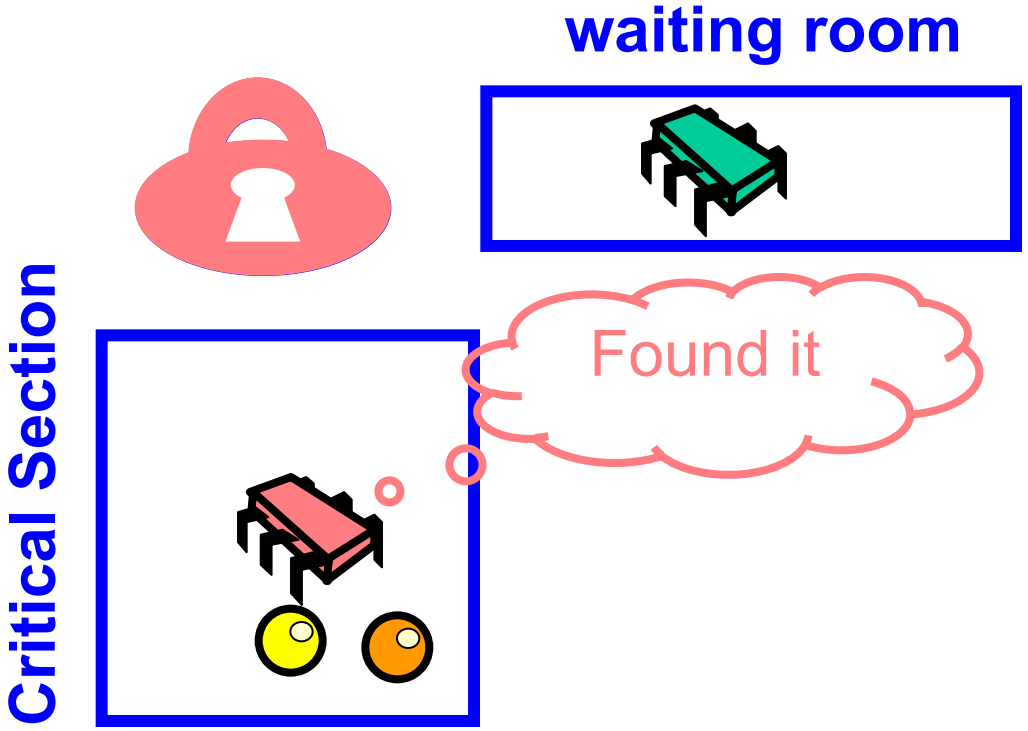


Lost Wake-Up

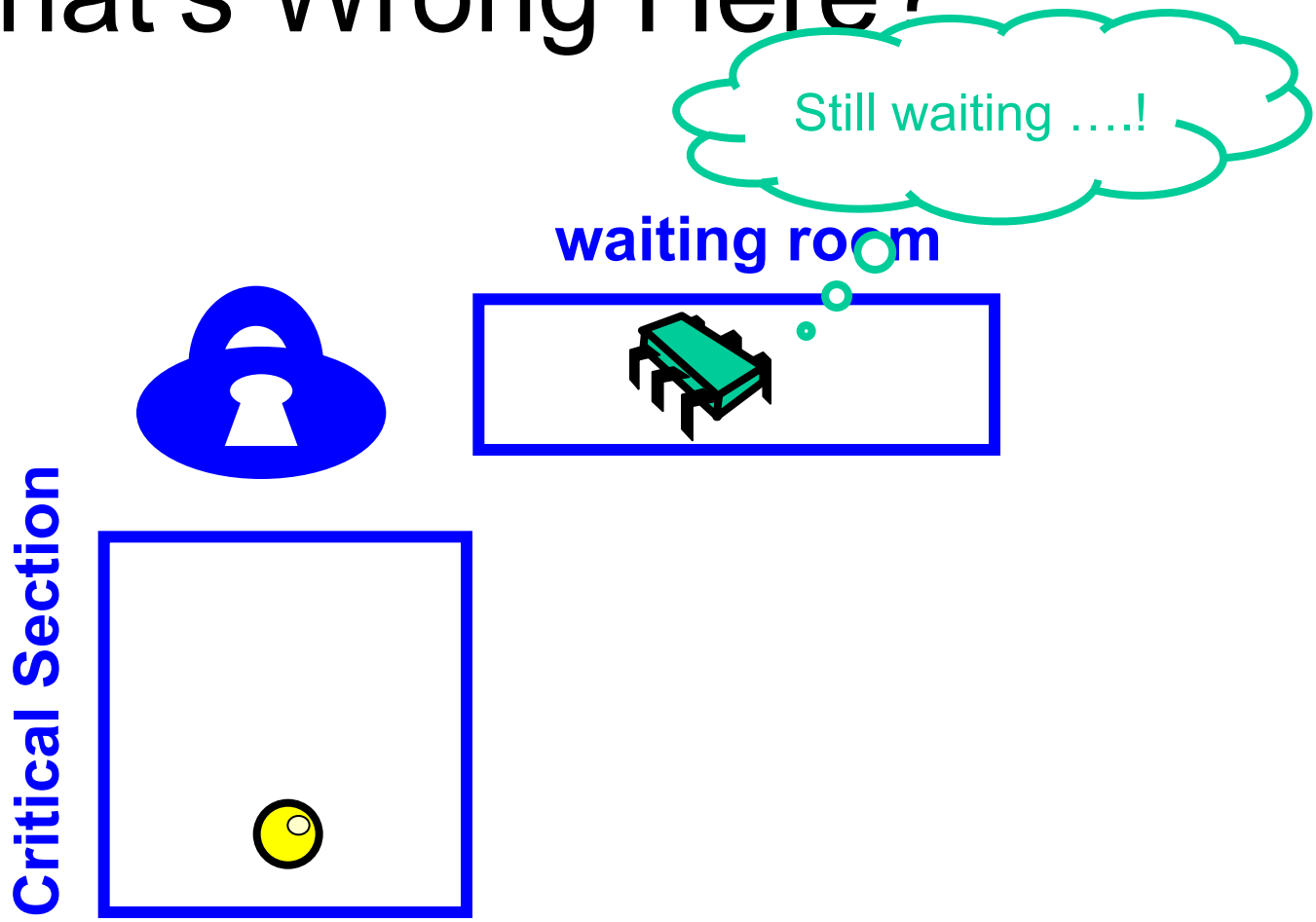

Yawn!
lock()
waiting room
enq()
Critical Section
Queue not empty so no need to signal
unlock()

Lost Wake-Up

Lost Wake-Up



What's Wrong Here?



Solution to Lost Wakeup

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

(pop)

Enq Method Part Deux

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

Enq Method Part Deux

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

Are there dequeuers to be signaled?

Enq Method Part Deux

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

Lock and unlock deq lock

Enq Method Part Deux

Signal dequeuers that queue is no longer empty

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

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

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

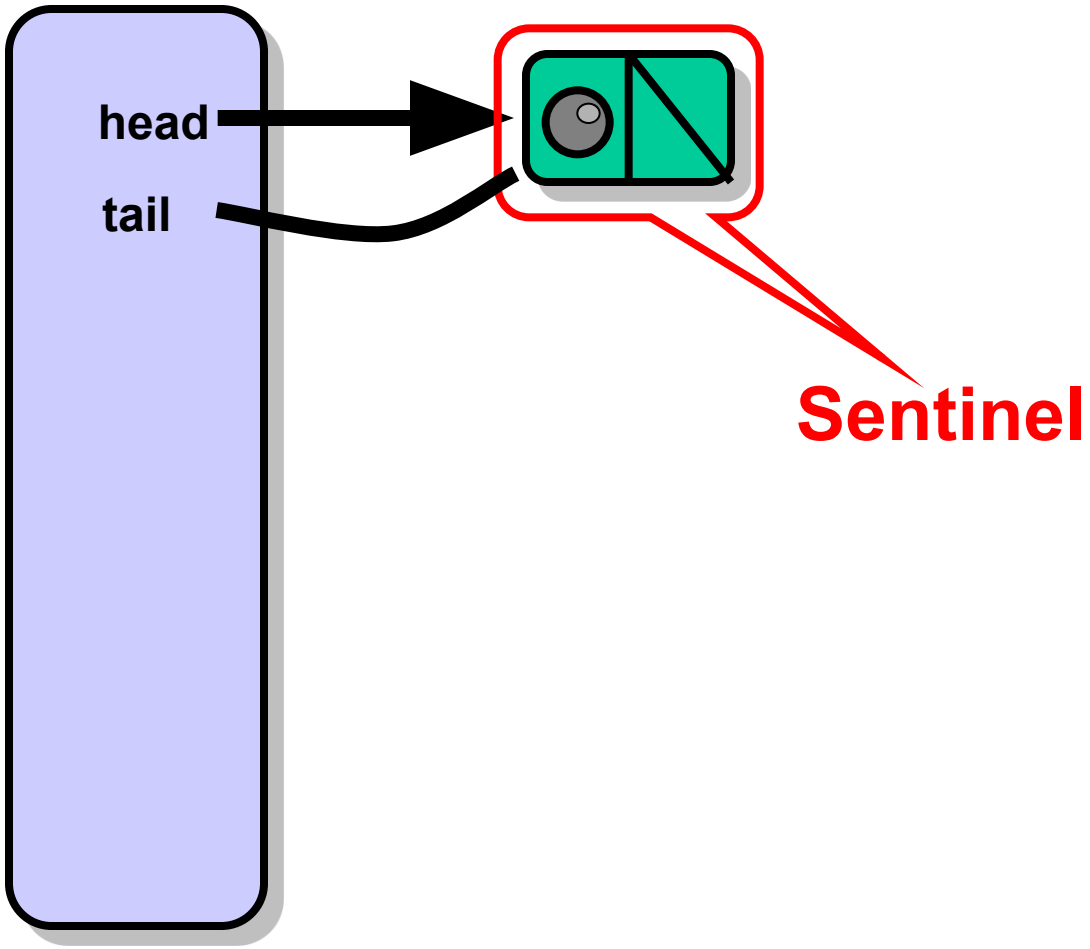
Split the Counter

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

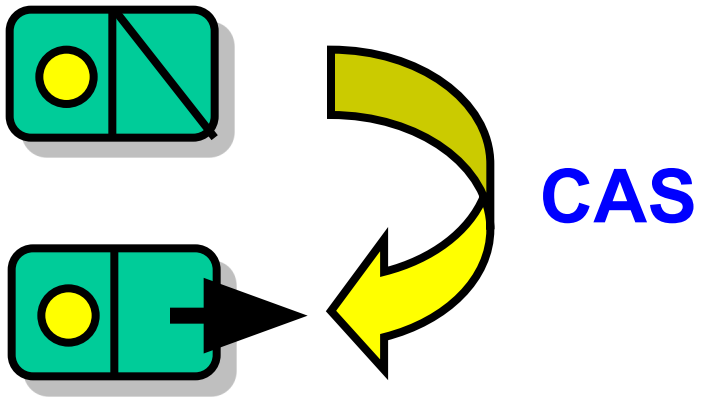
Split Counter

- 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 …)

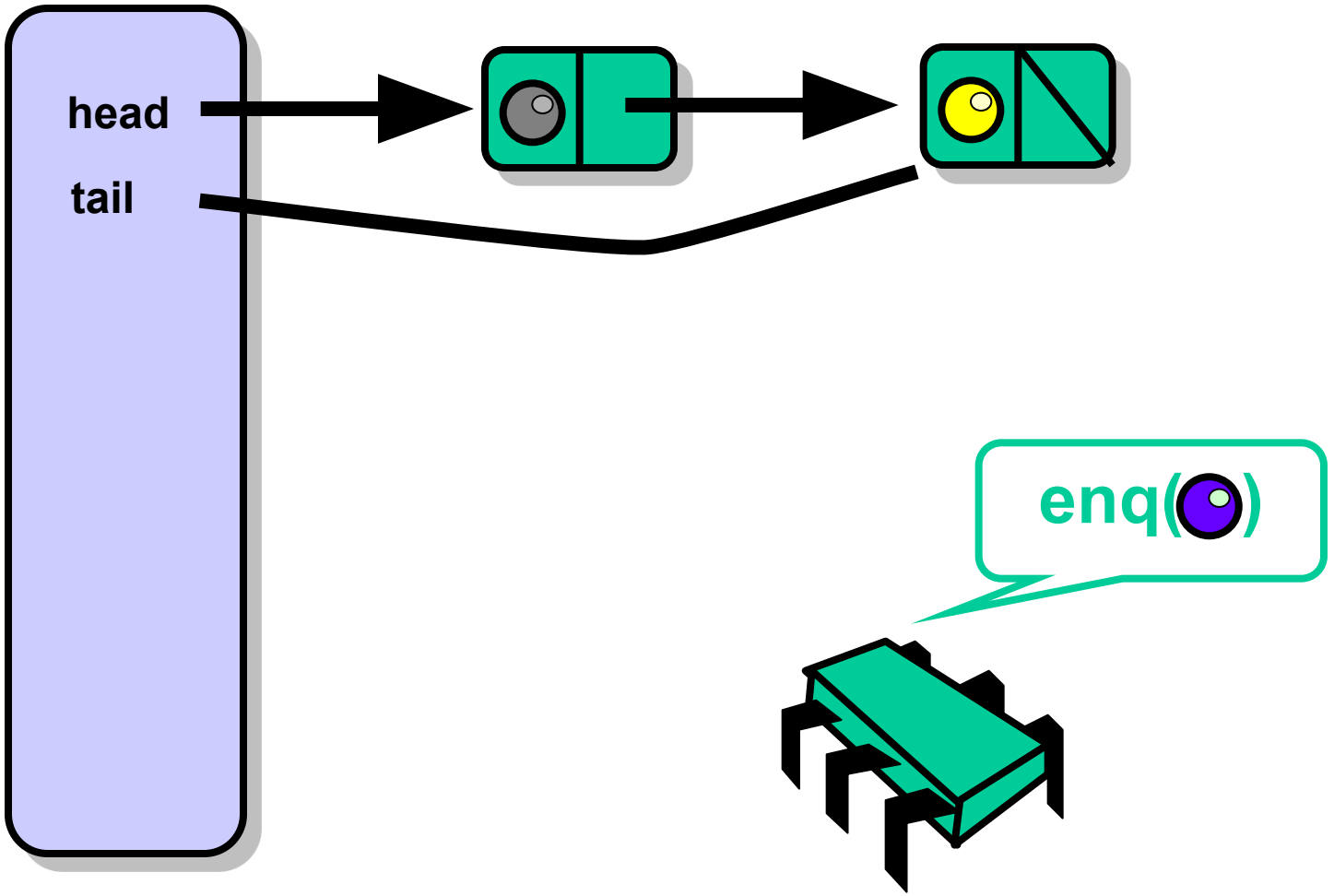
A Lock-Free Queue



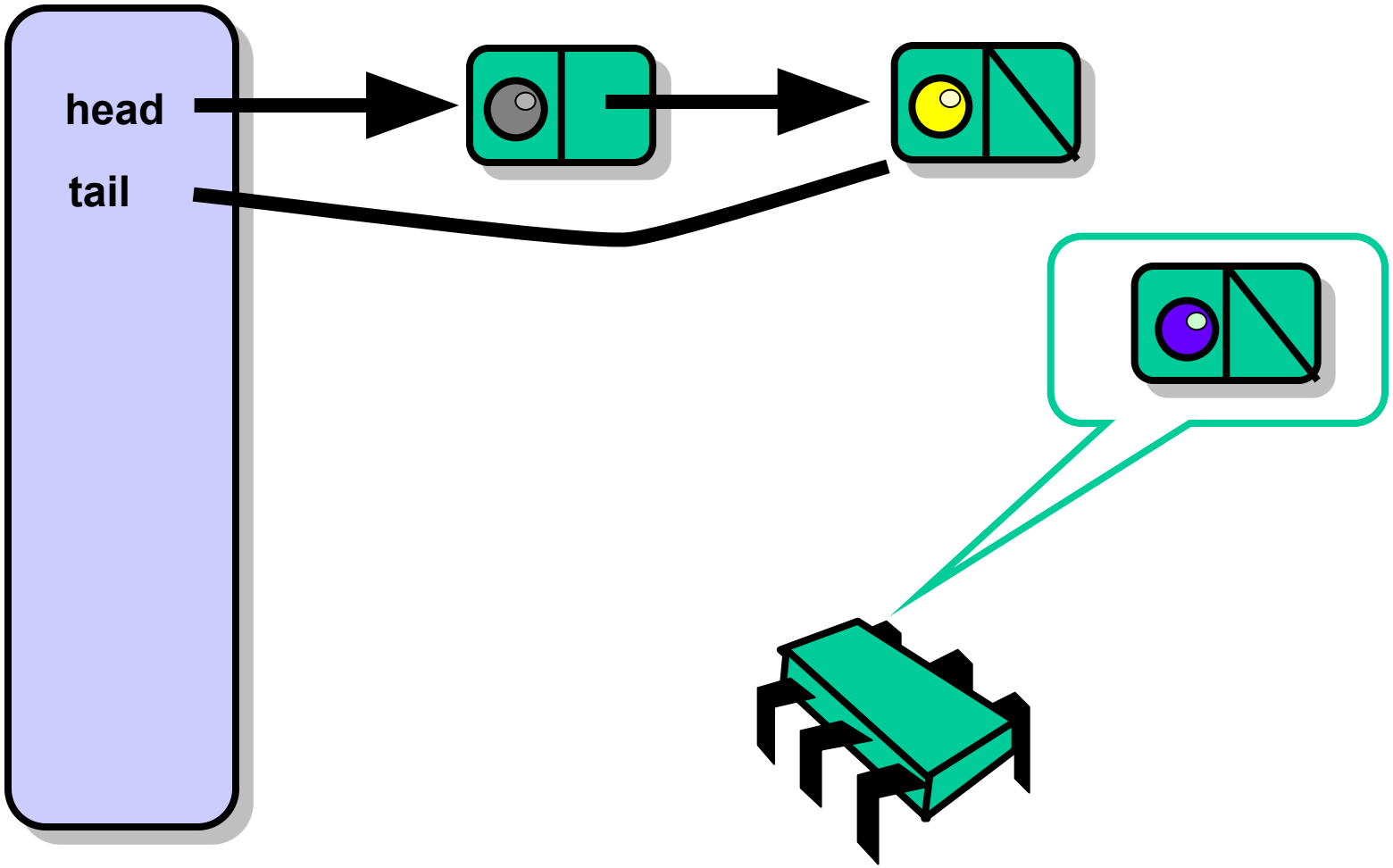
Compare and Set



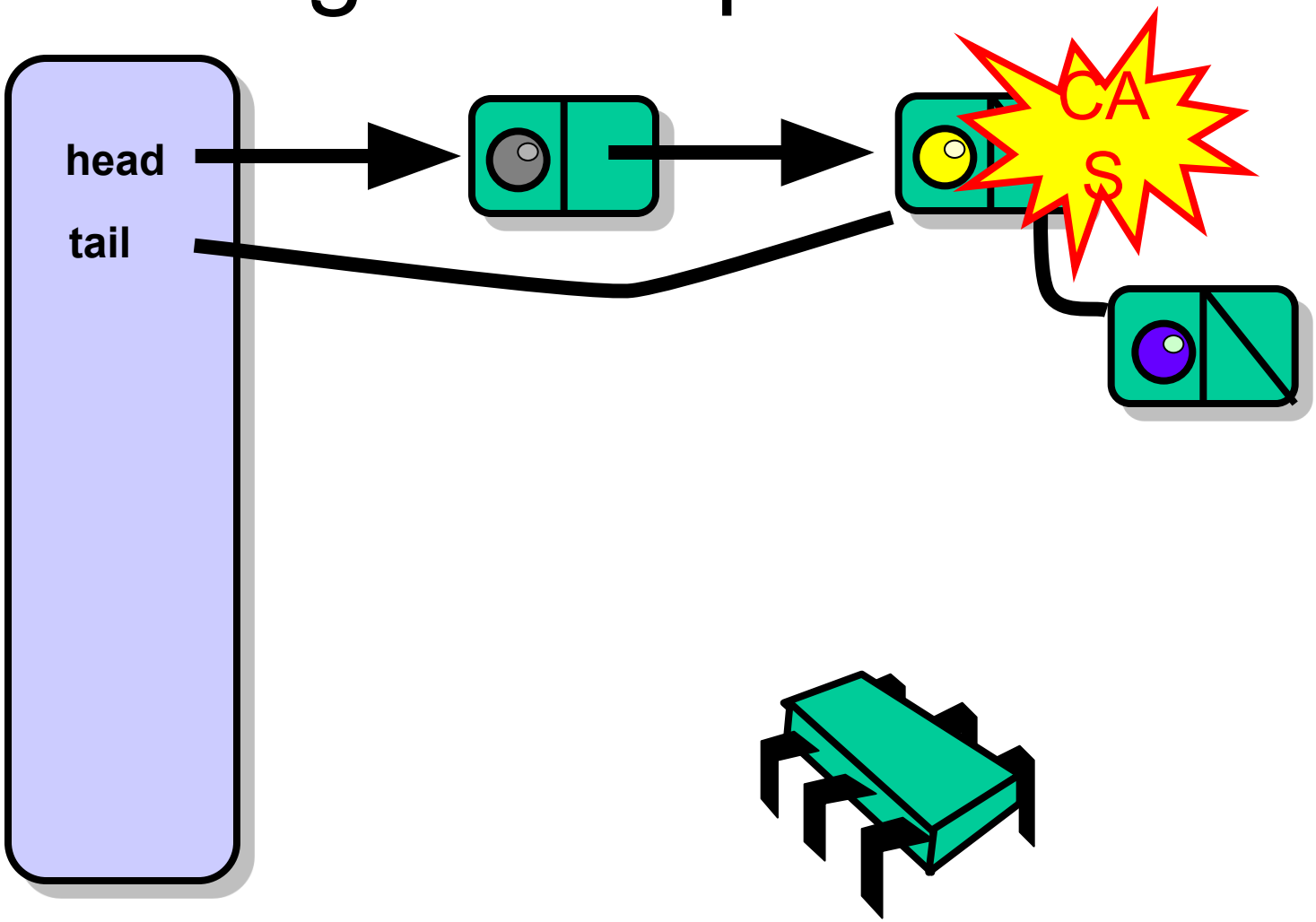
Enqueue



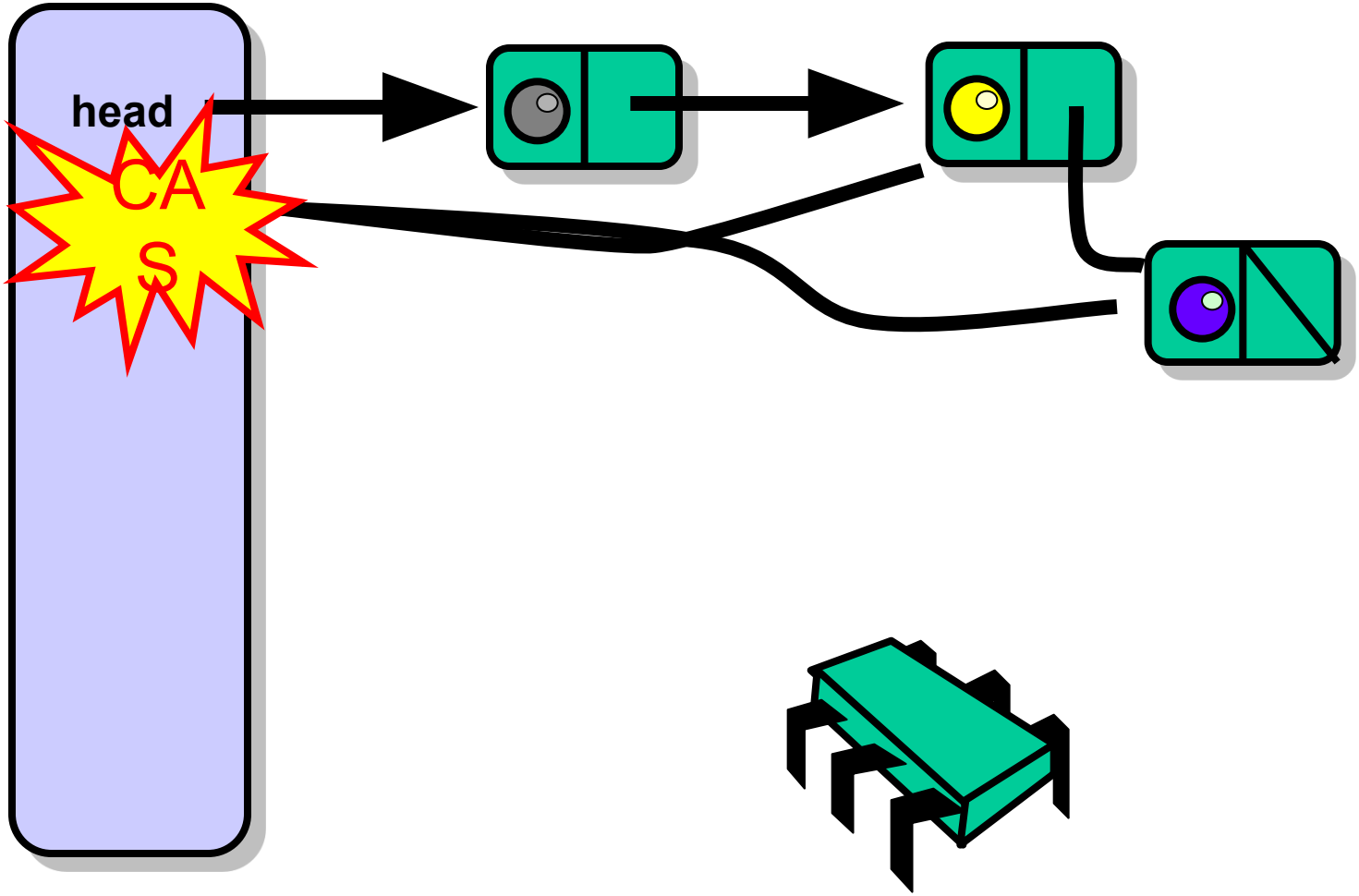
Enqueue



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!

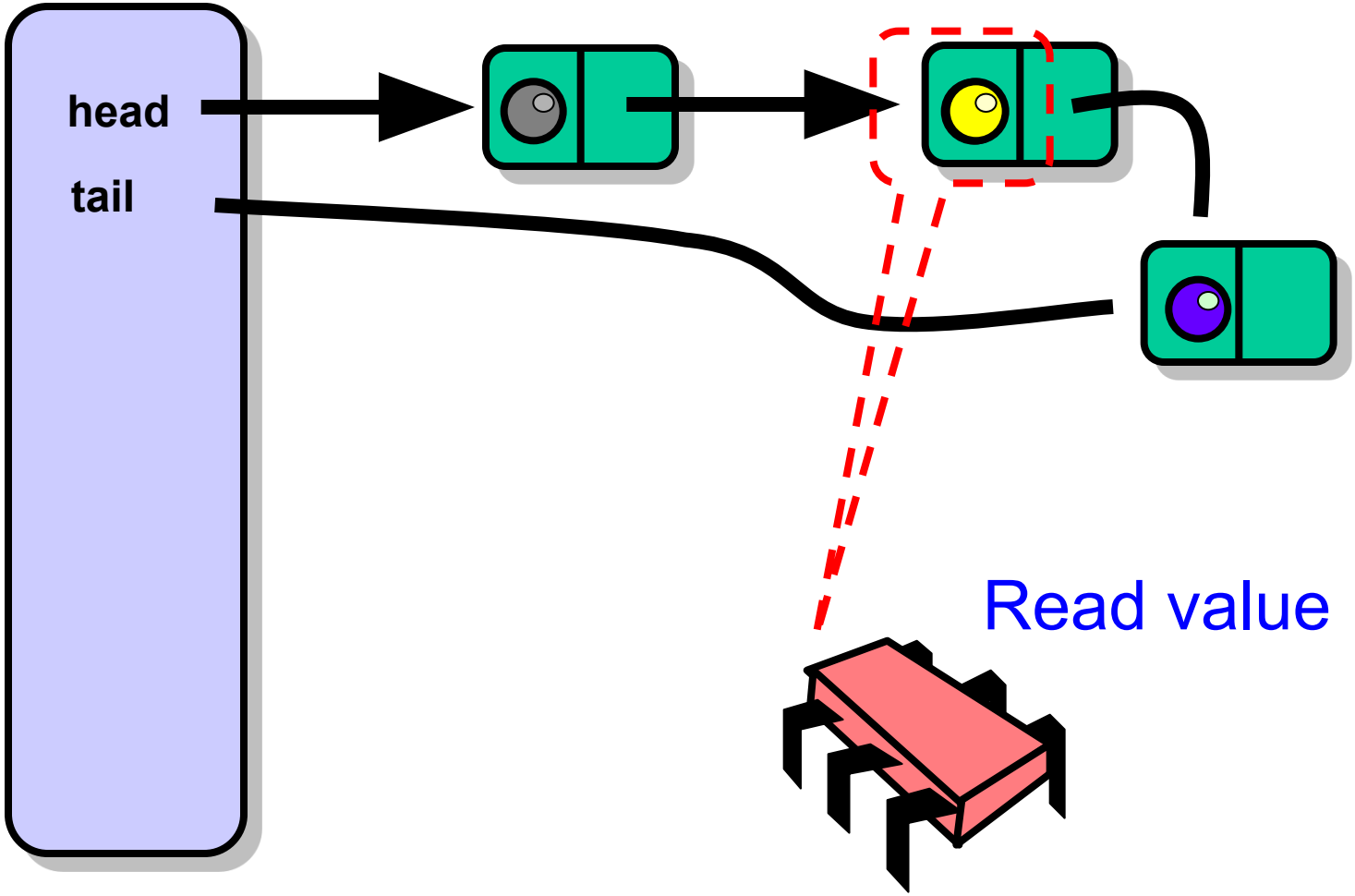
Enqueue

- 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**
- As in the universal construction

When CASs Fail

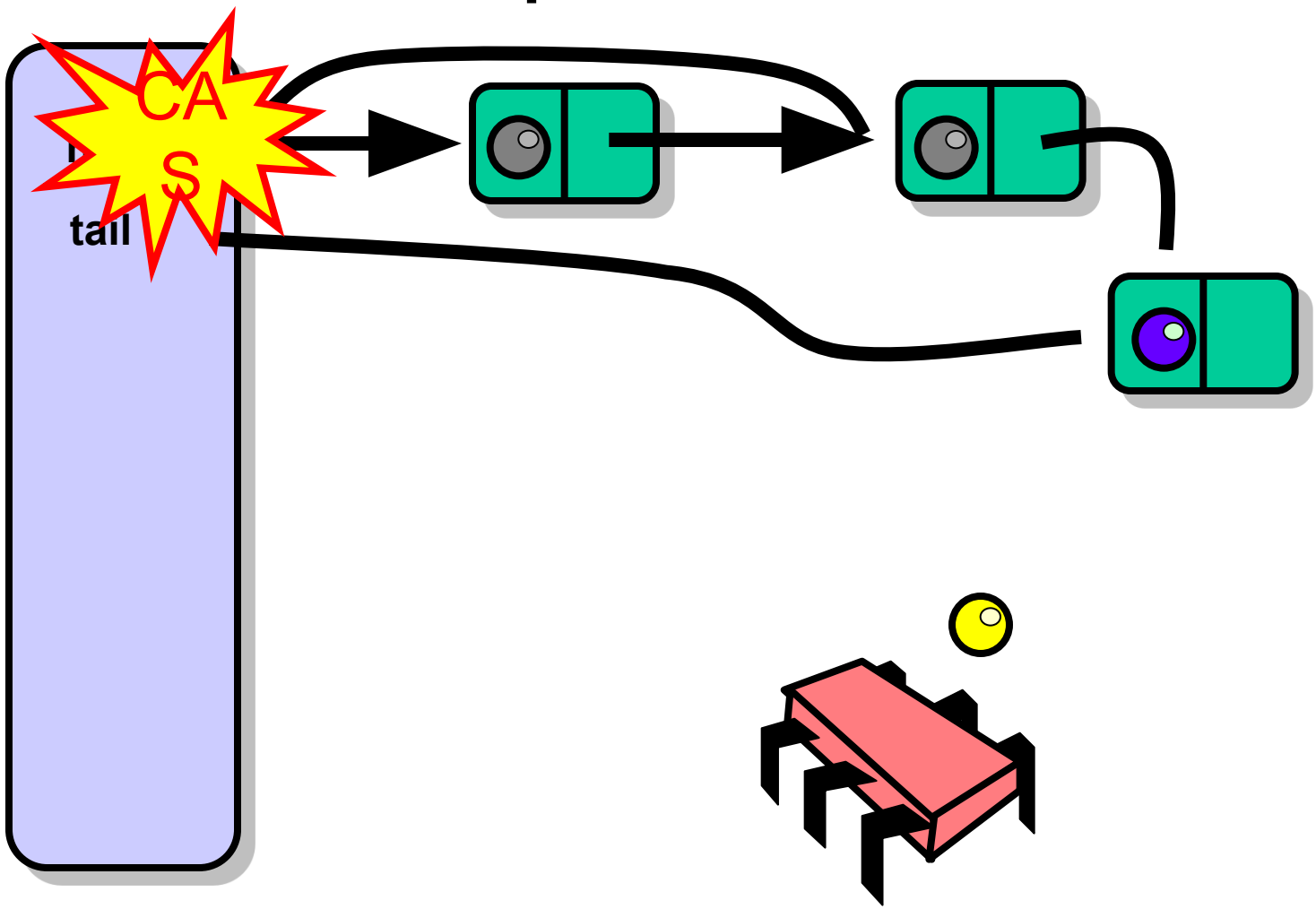
- During logical enqueue
  - Abandon hope, restart
  - Still lock-free (why?)
- During physical enqueue
  - Ignore it (why?)

Dequeuer



Make first Node
new sentinel

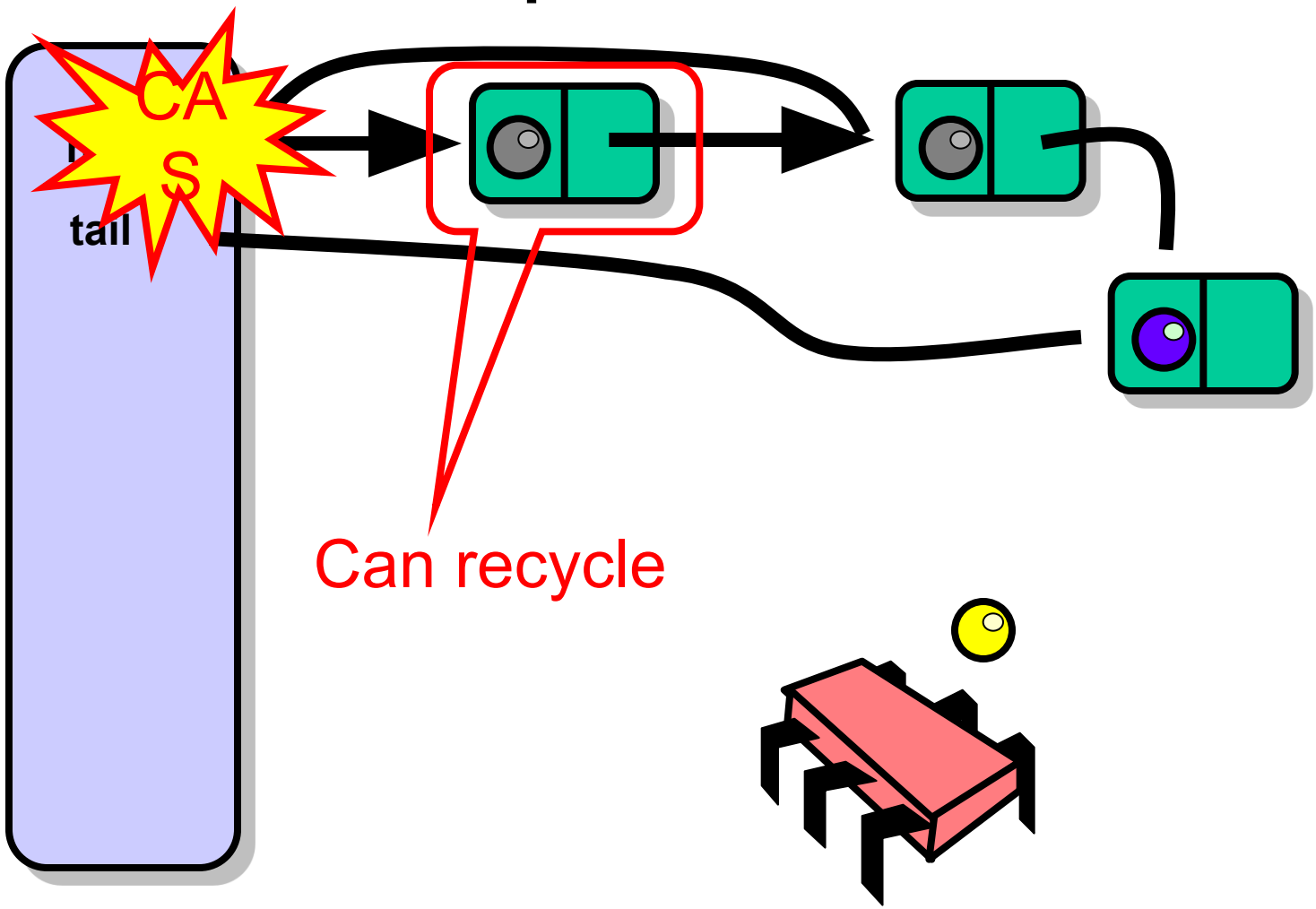
Dequeuer



Memory Reuse?

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

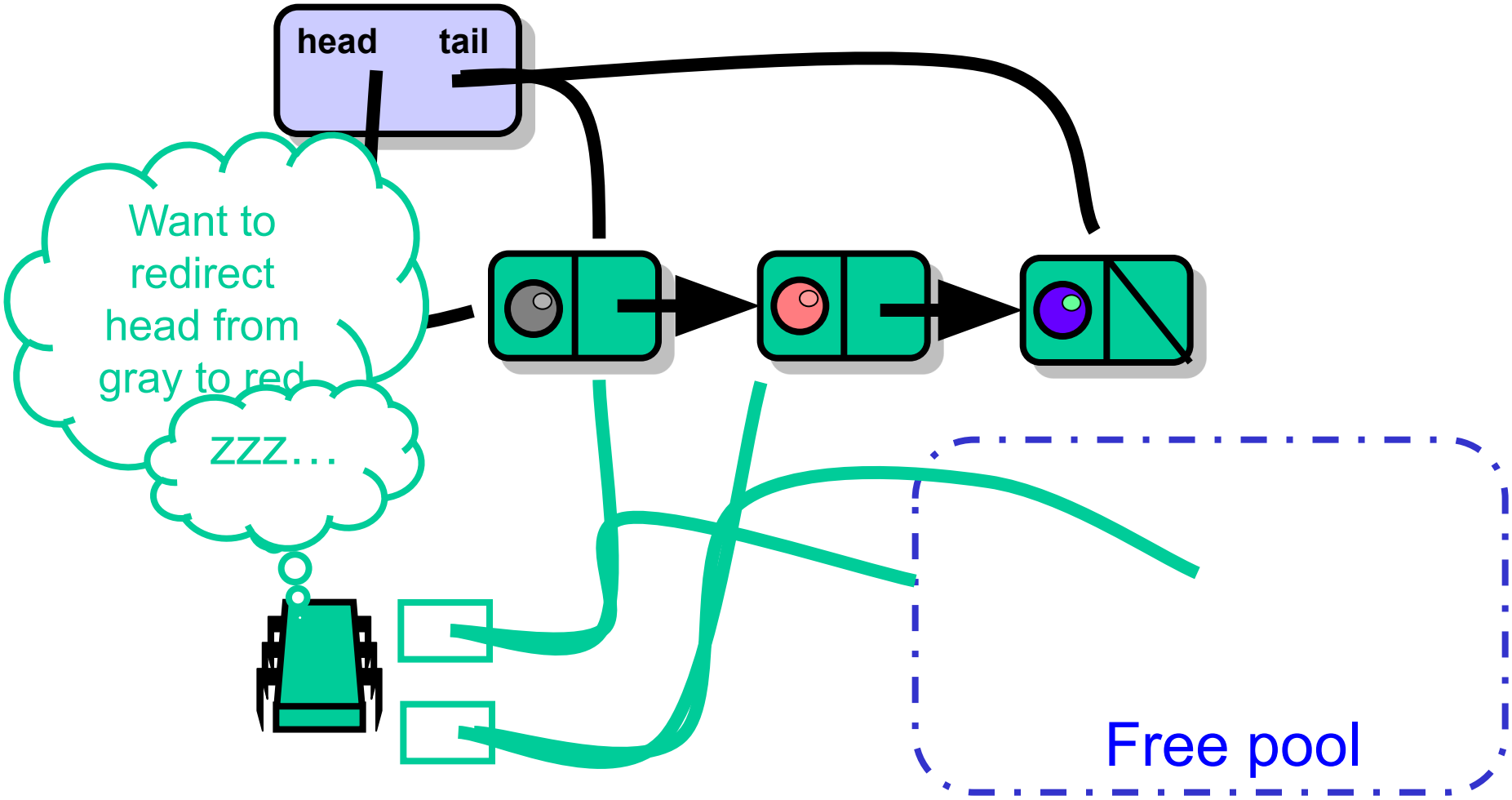
Dequeuer



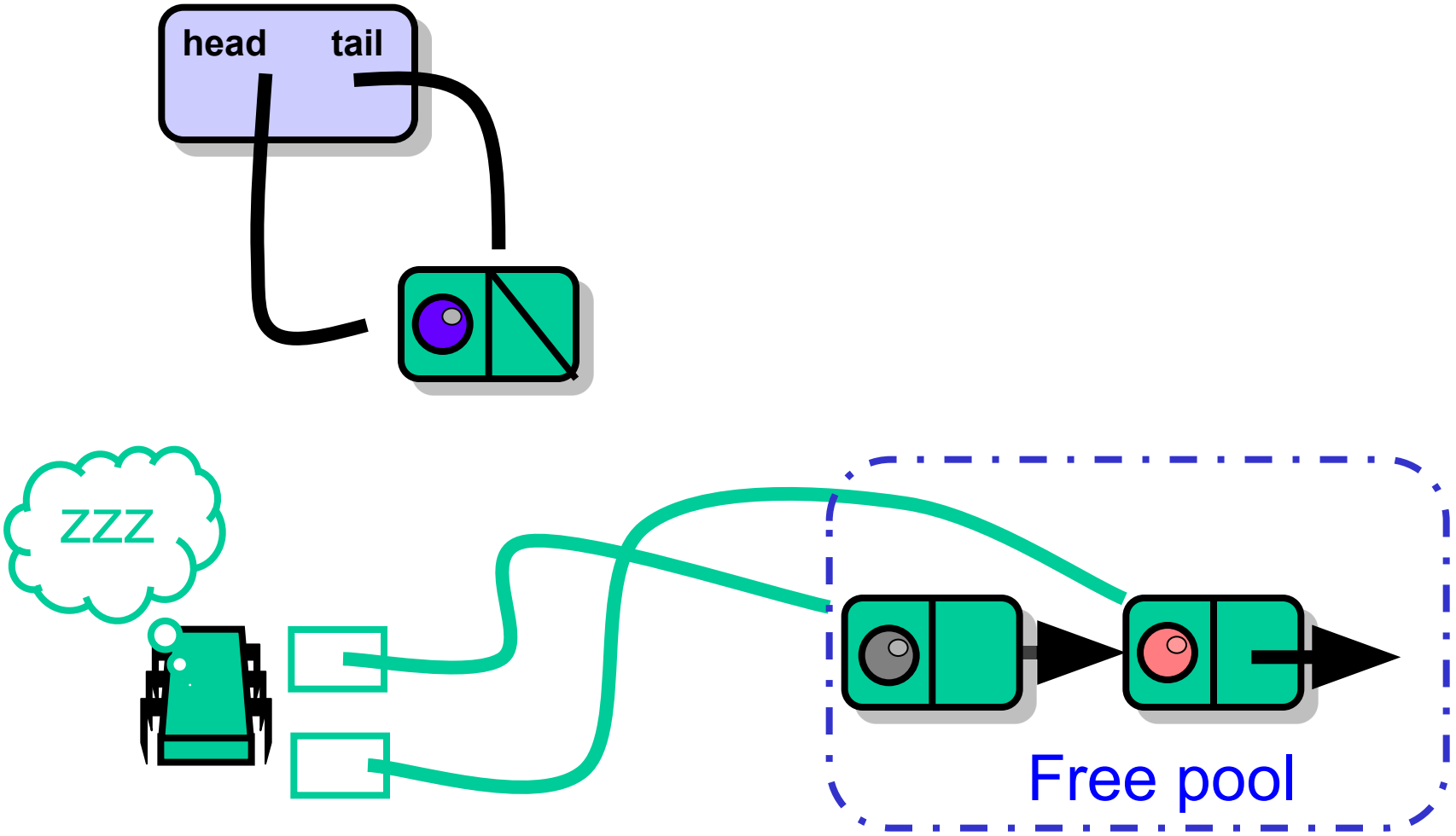
Simple Solution

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

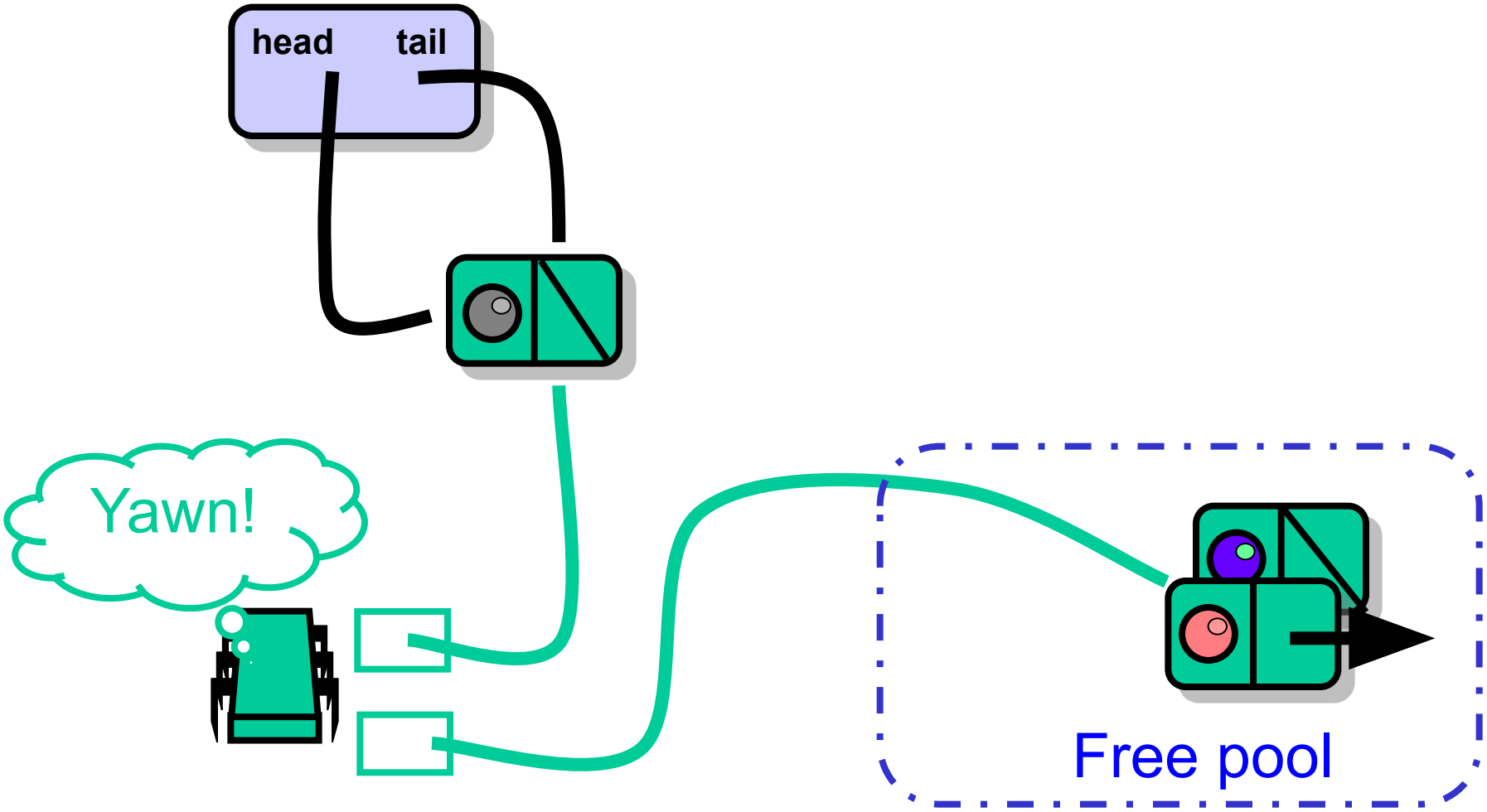
Why Recycling is Hard



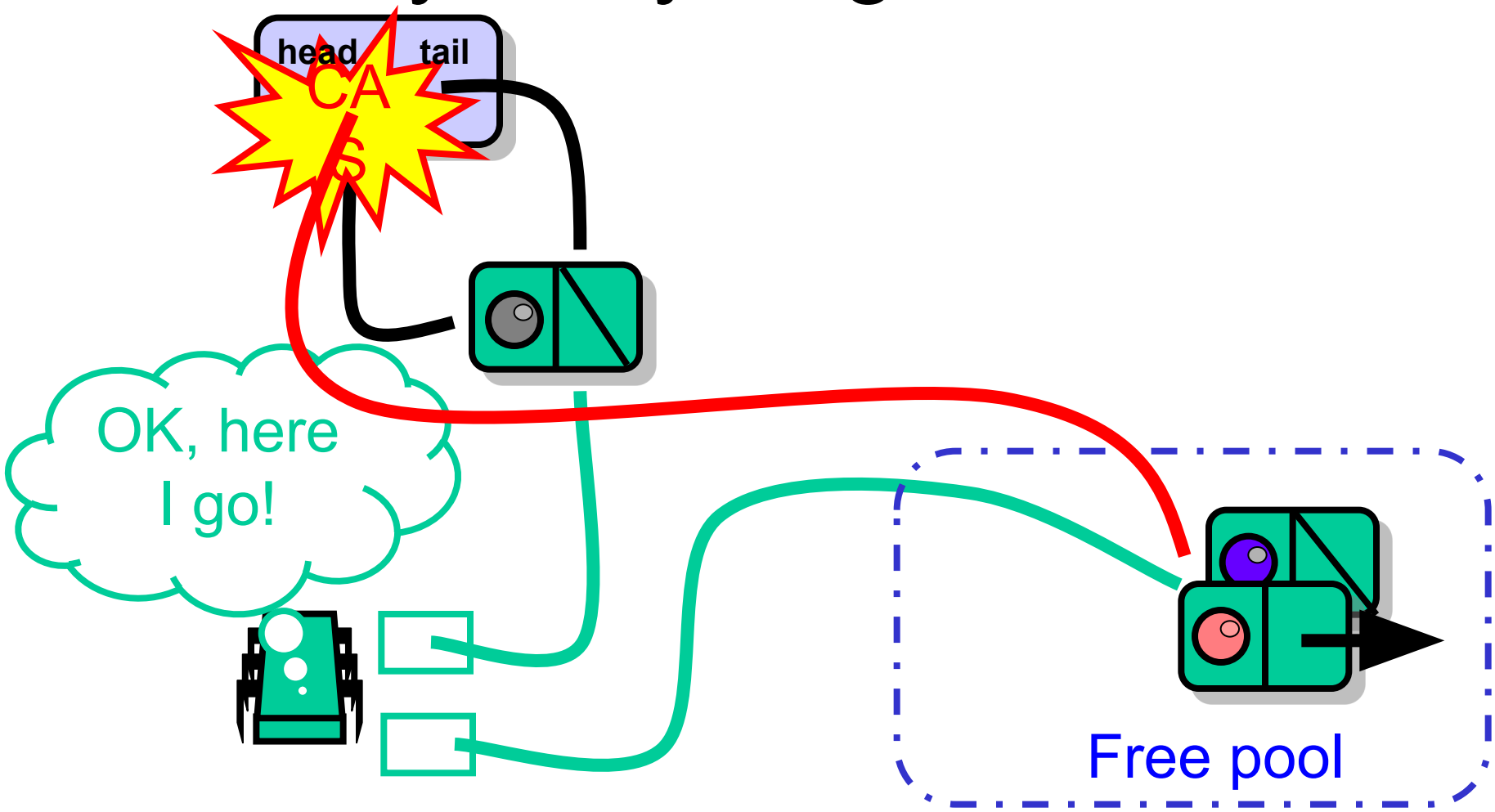
Both Nodes Reclaimed



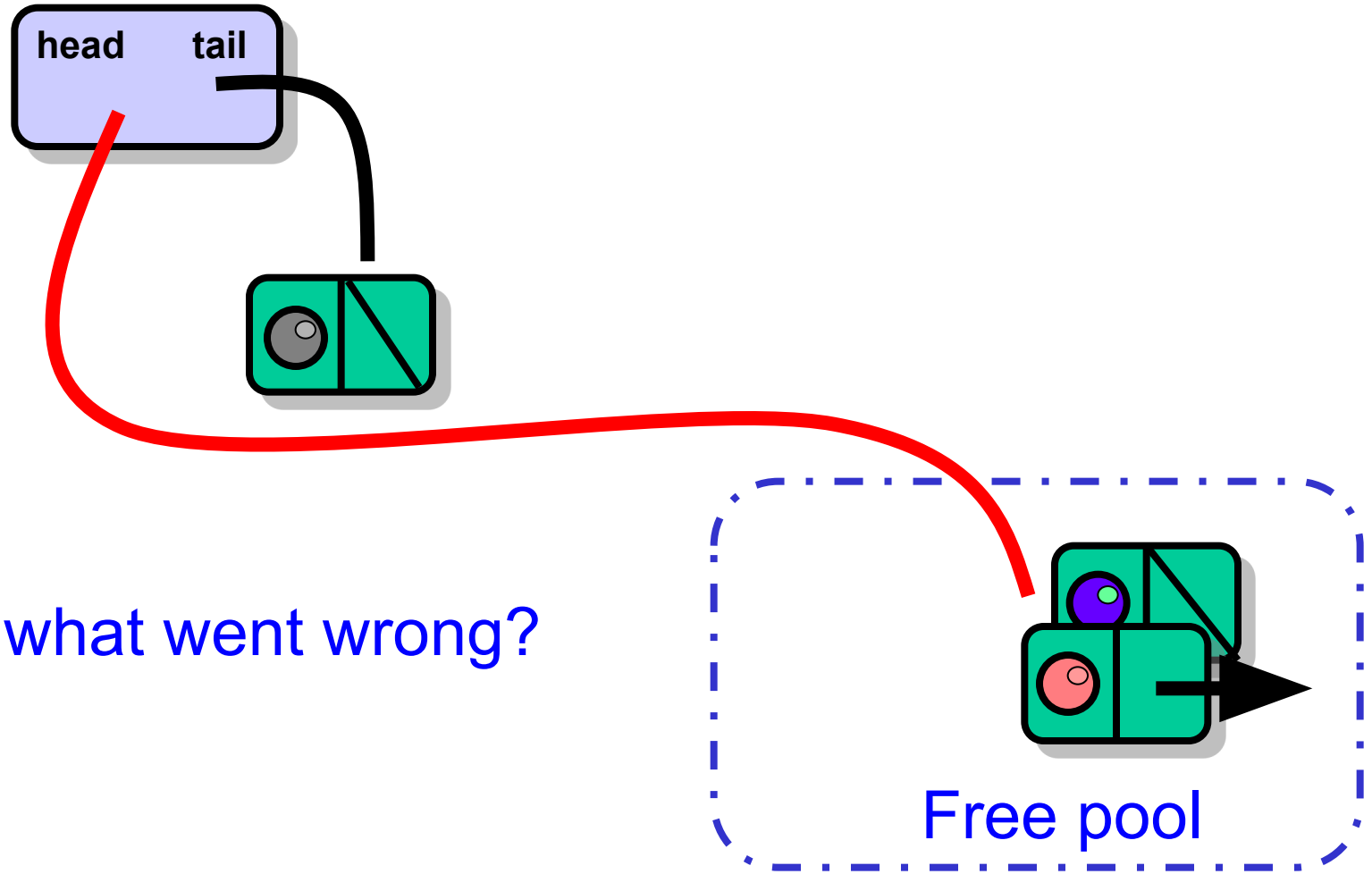
One Node Recycled



Why Recycling is Hard

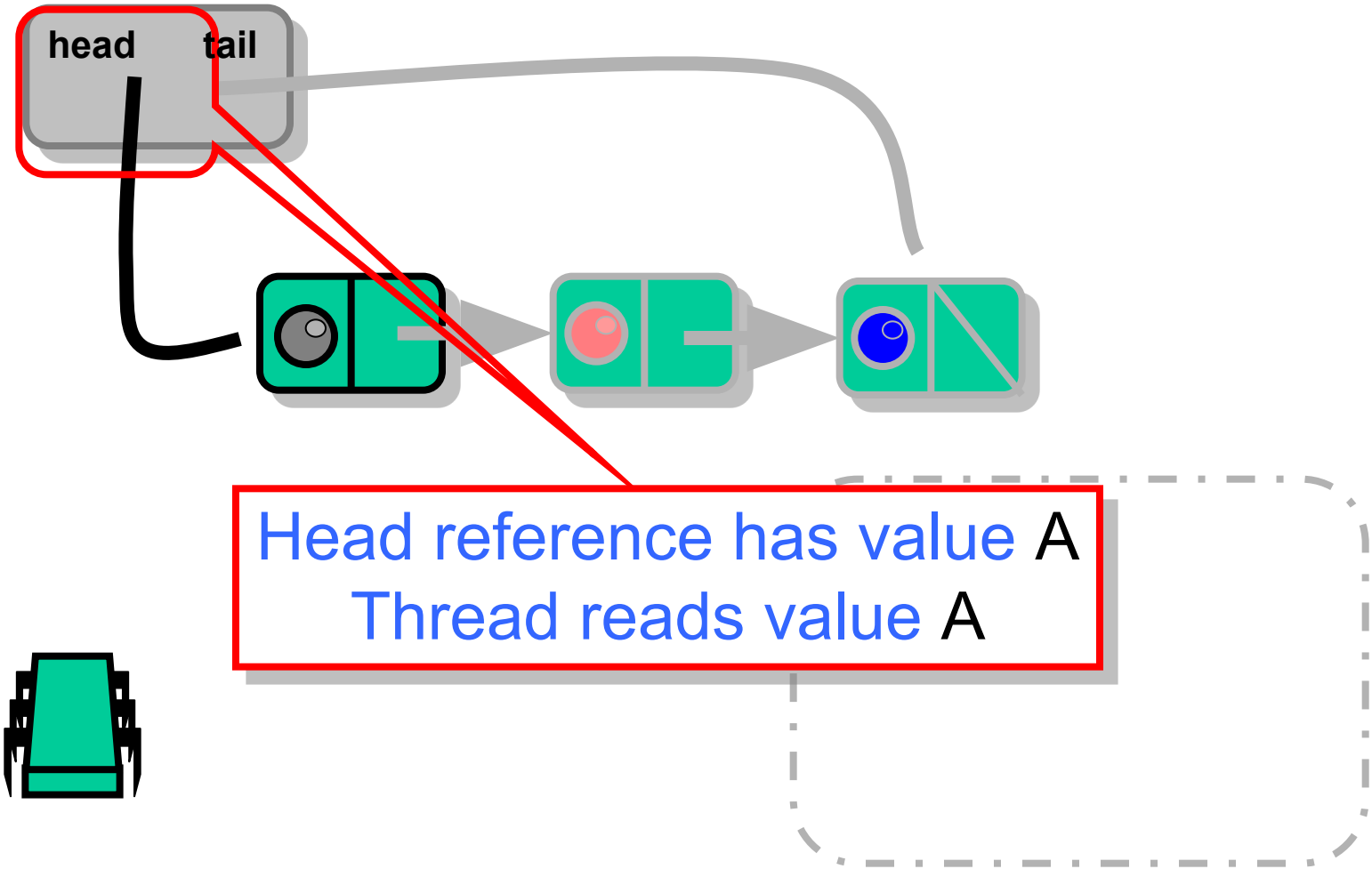


Recycle FAIL

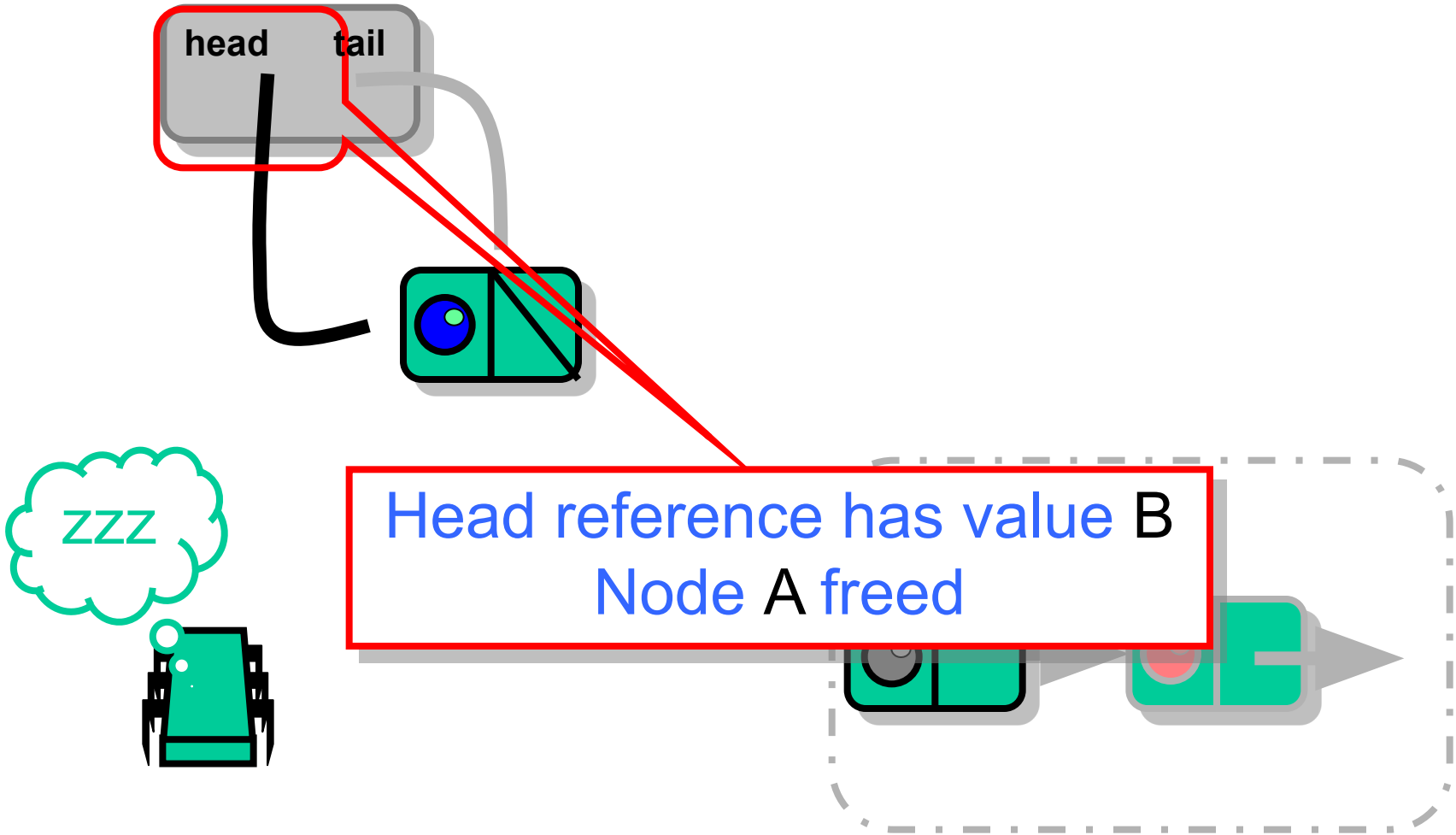


zOMG what went wrong?

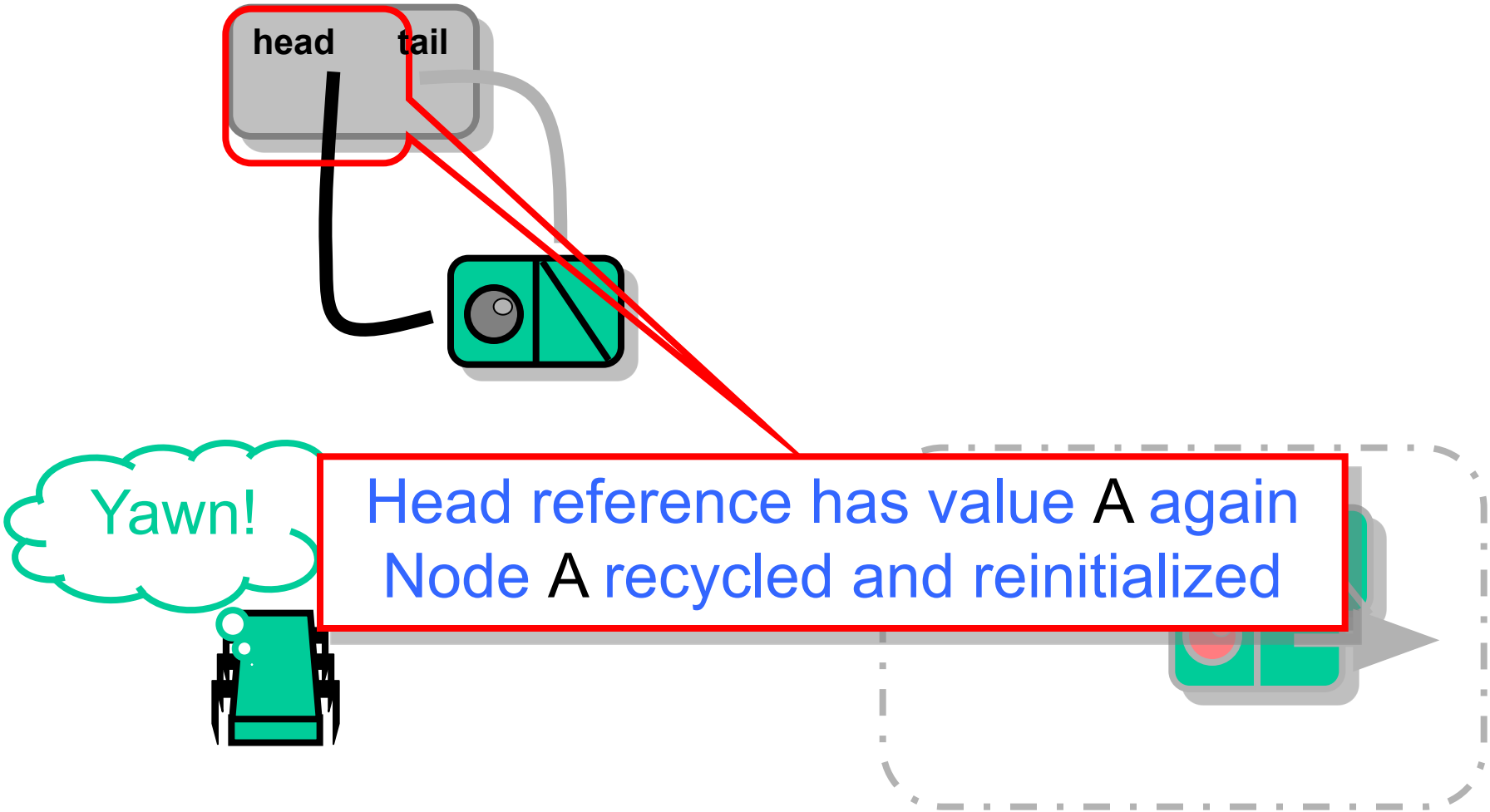
The Dreaded ABA Problem



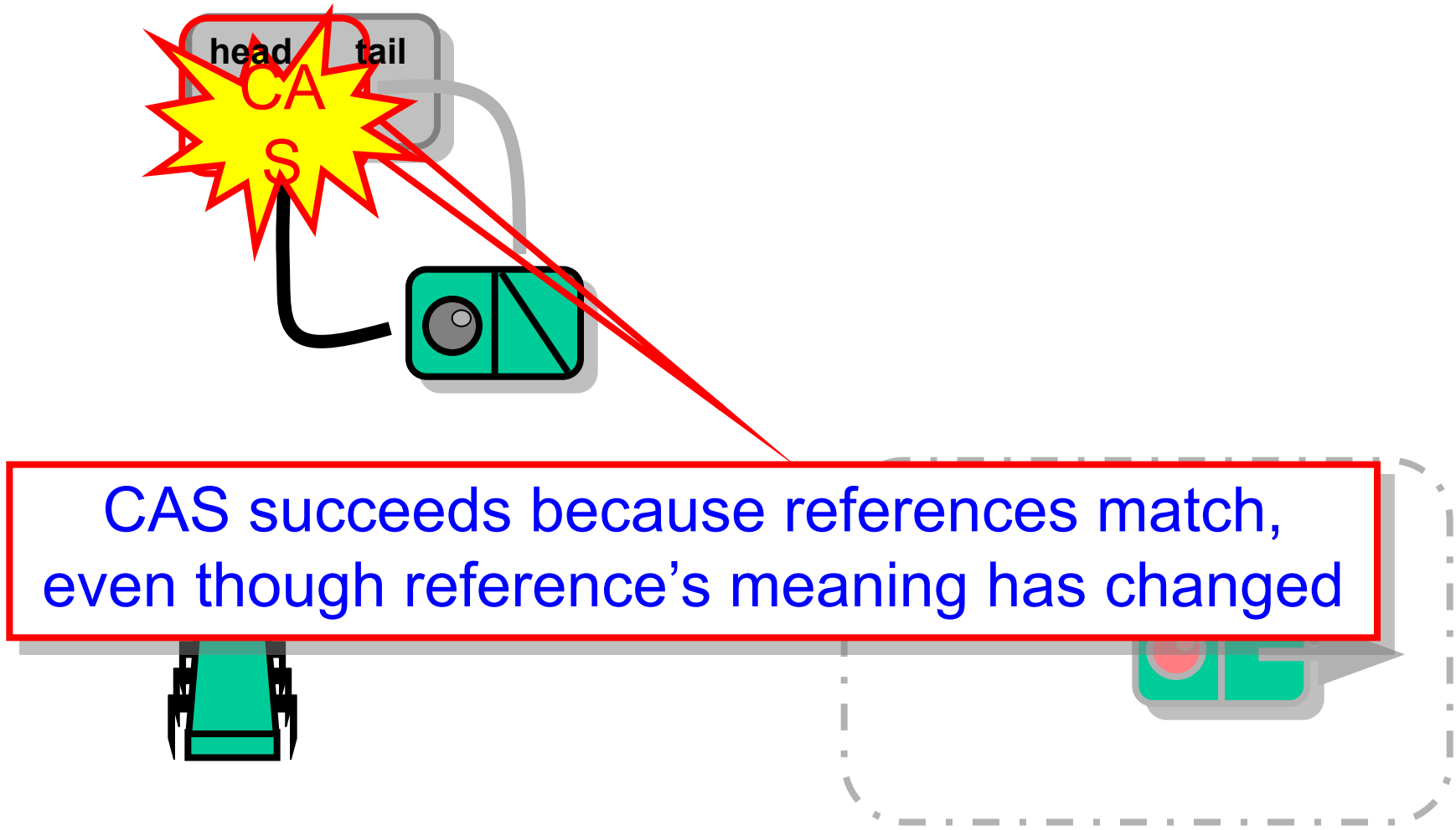
Dreaded ABA continued



Dreaded ABA continued



Dreaded ABA continued



The Dreaded ABA FAIL

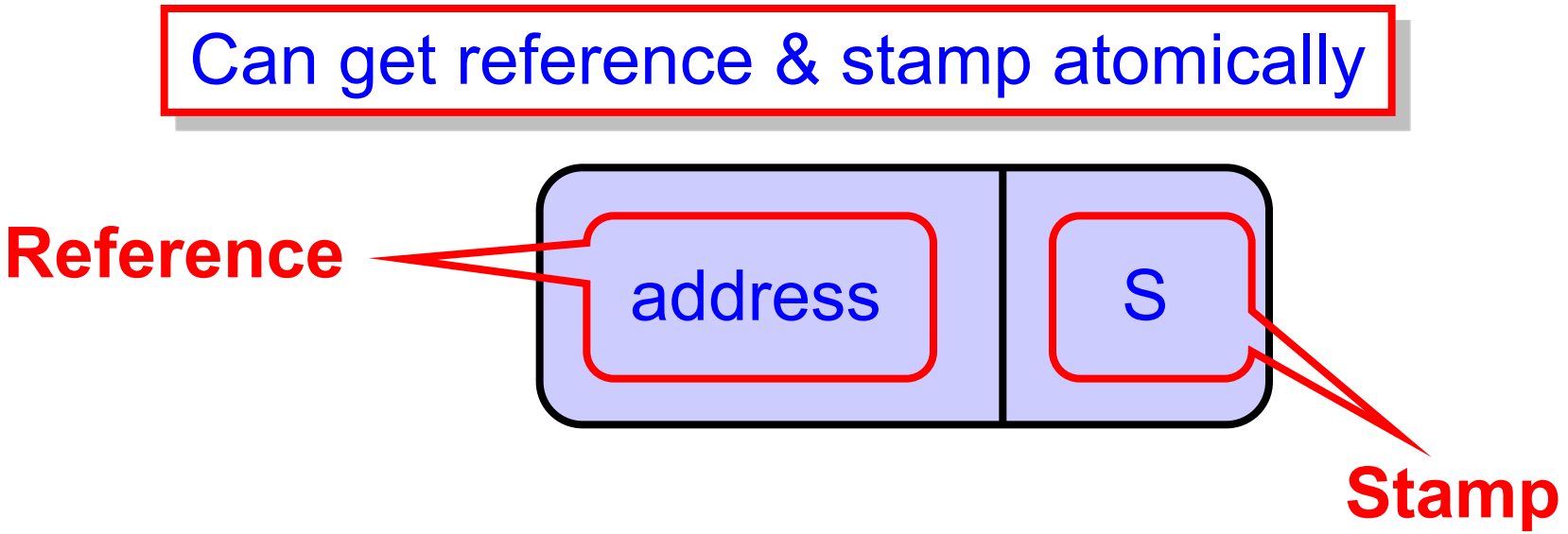
- Is a result of CAS() semantics
  - Oracle, Intel, AMD, …
- Not with Load-Locked/Store-Conditional
  - IBM …

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

Atomic Stamped Reference

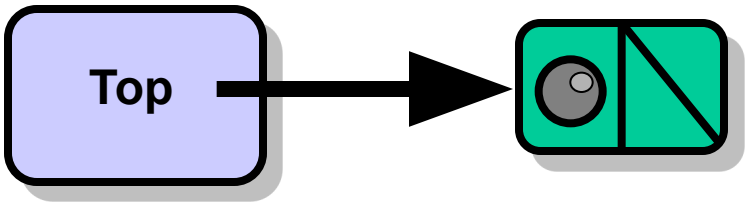
- **AtomicStampedReference** class
  - Java.util.concurrent.atomic package



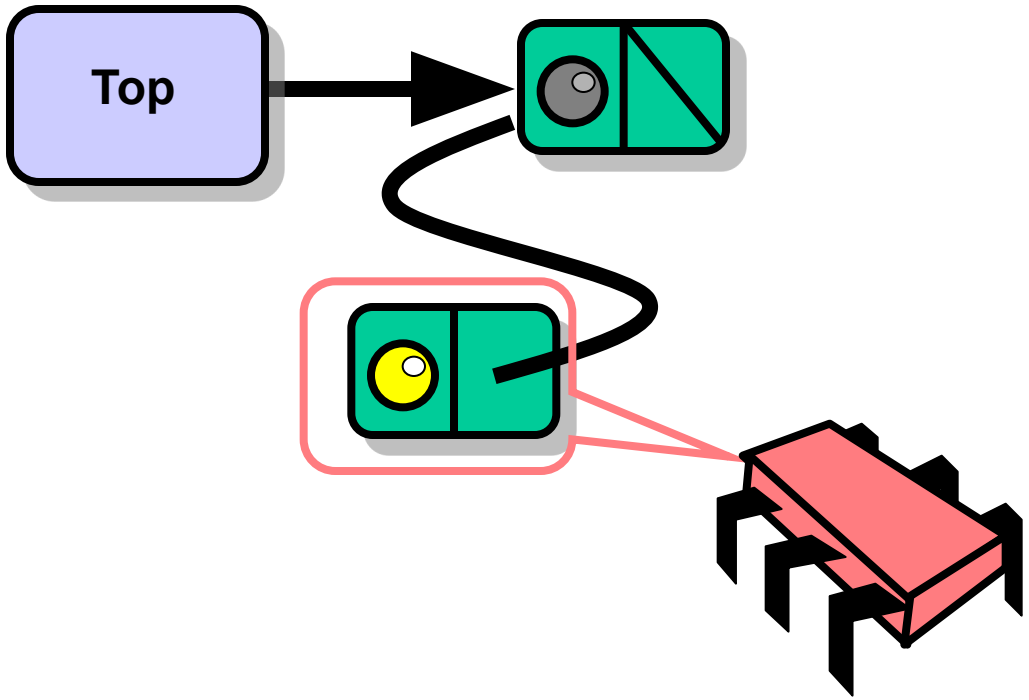
Concurrent Stack

- Methods
  - **push(x)**
  - **pop()**
- Last-in, First-out (LIFO) order
- Lock-Free!

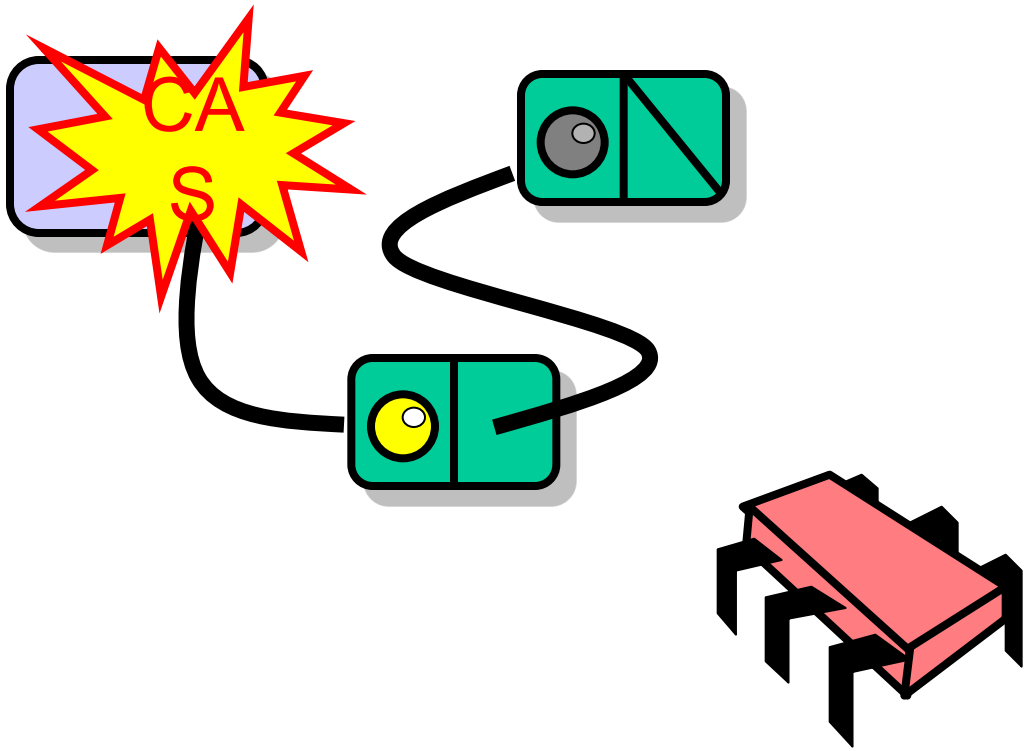
Empty Stack



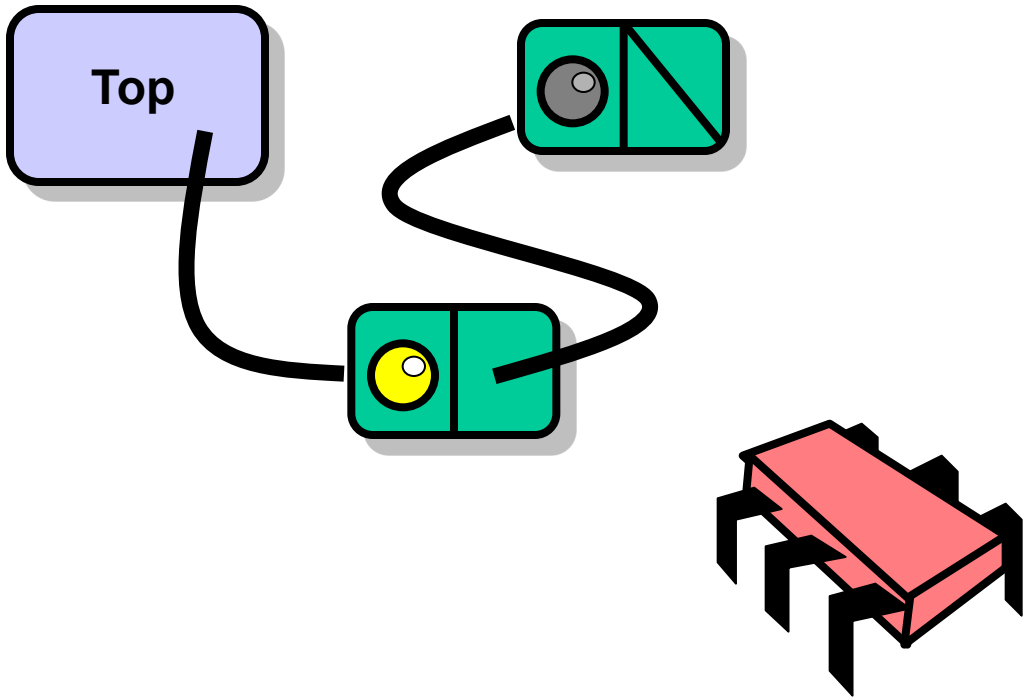
Push



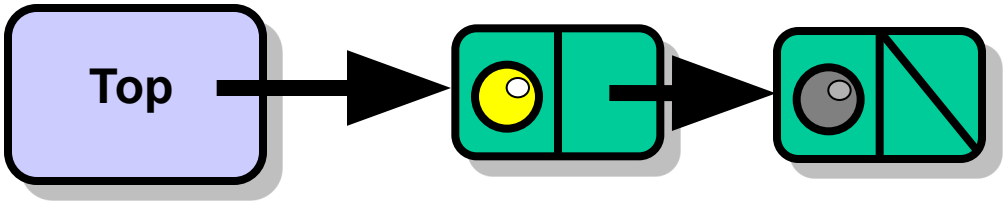
Push

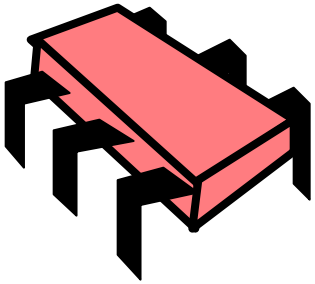


Push

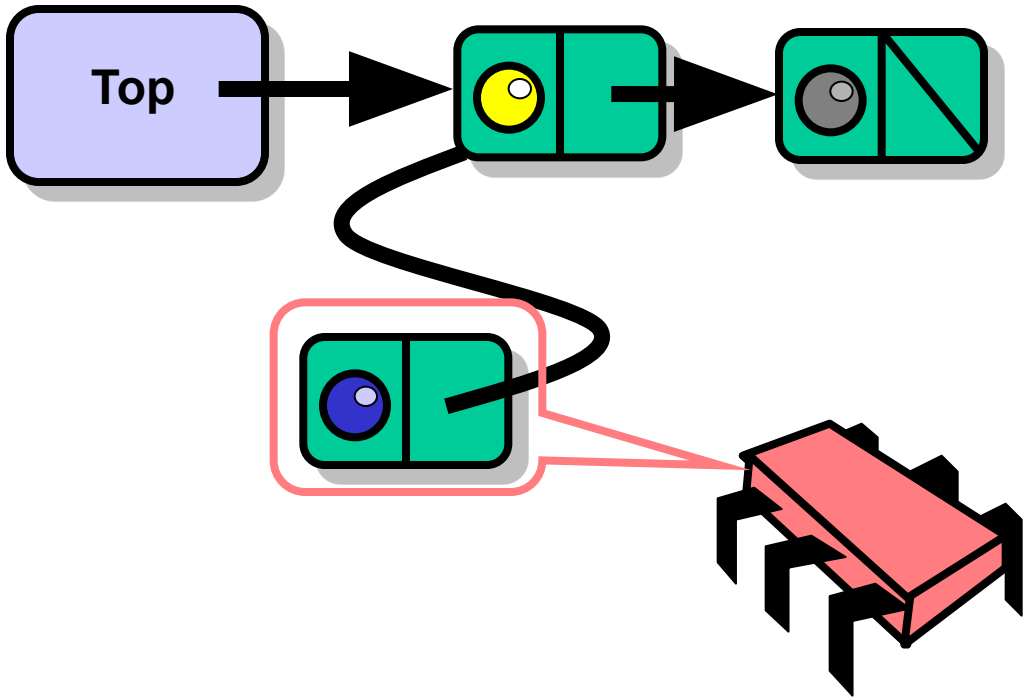


Push

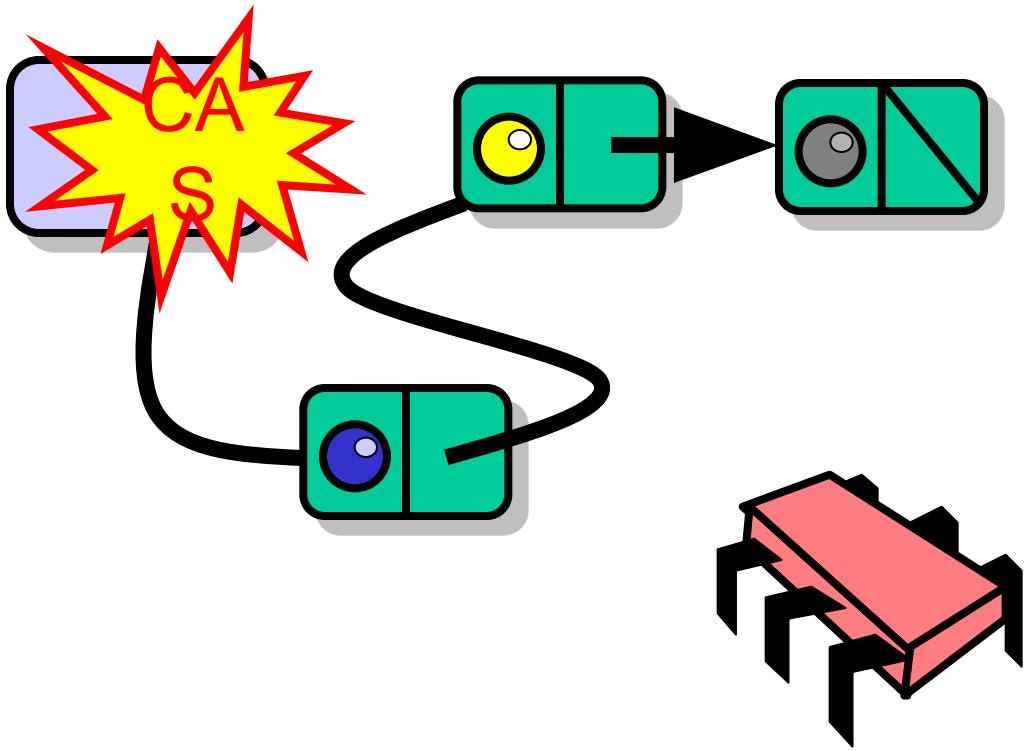




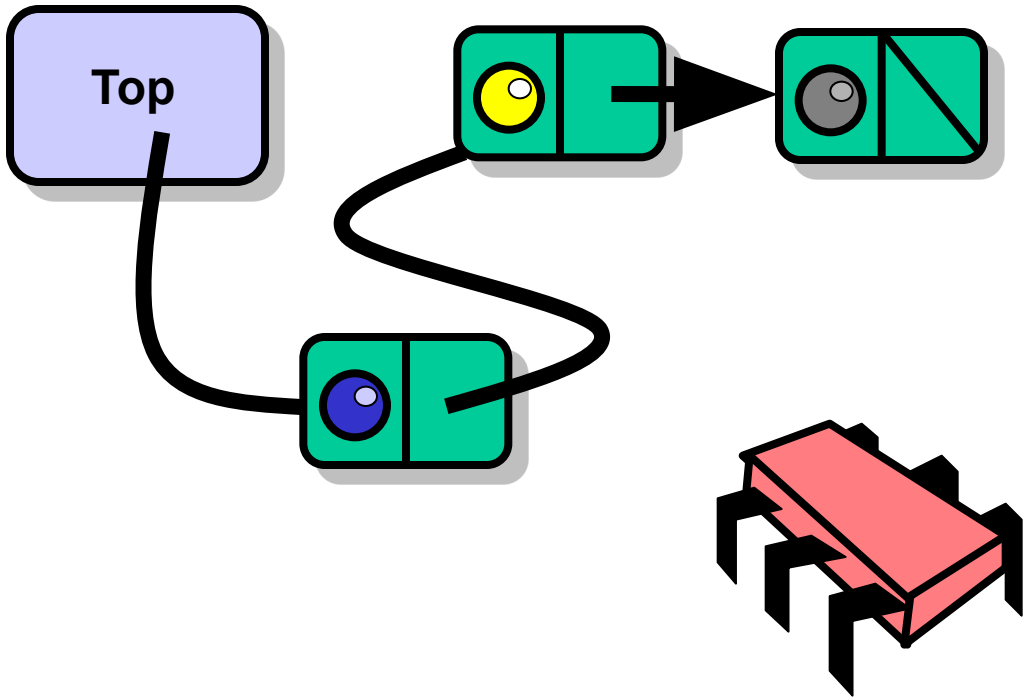
Push



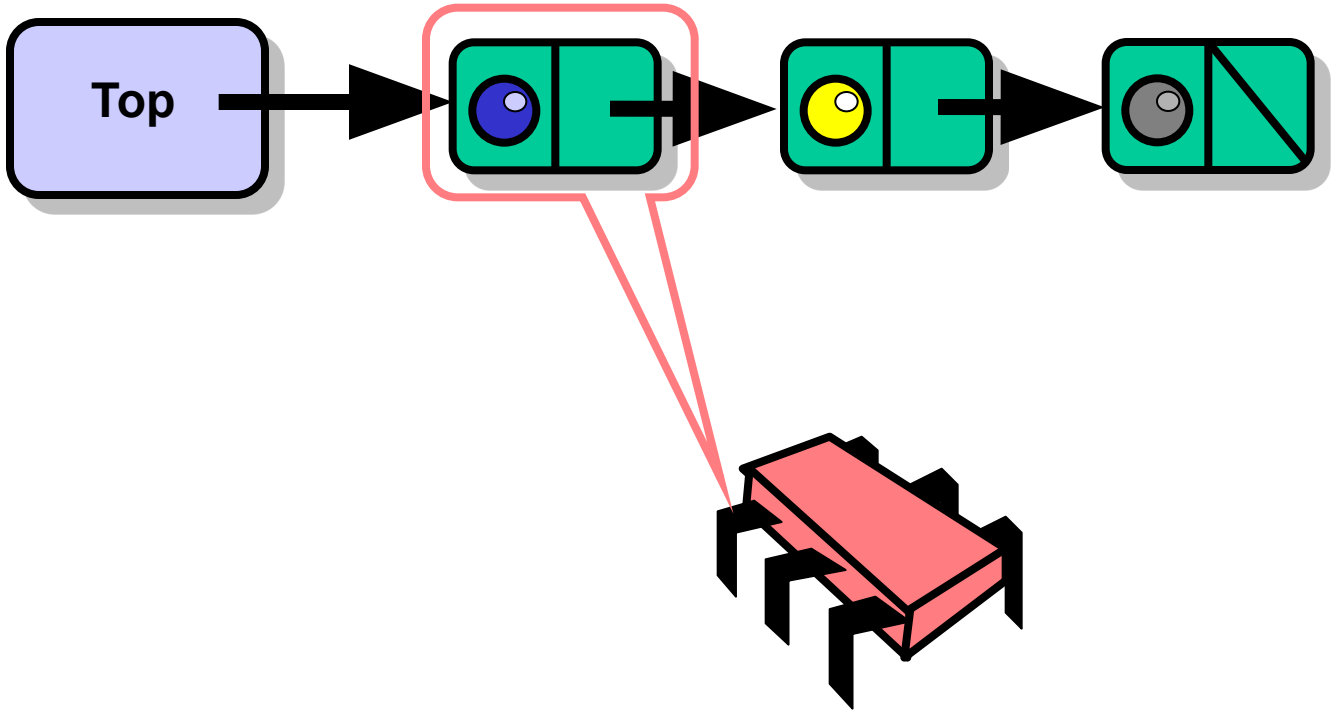
Push



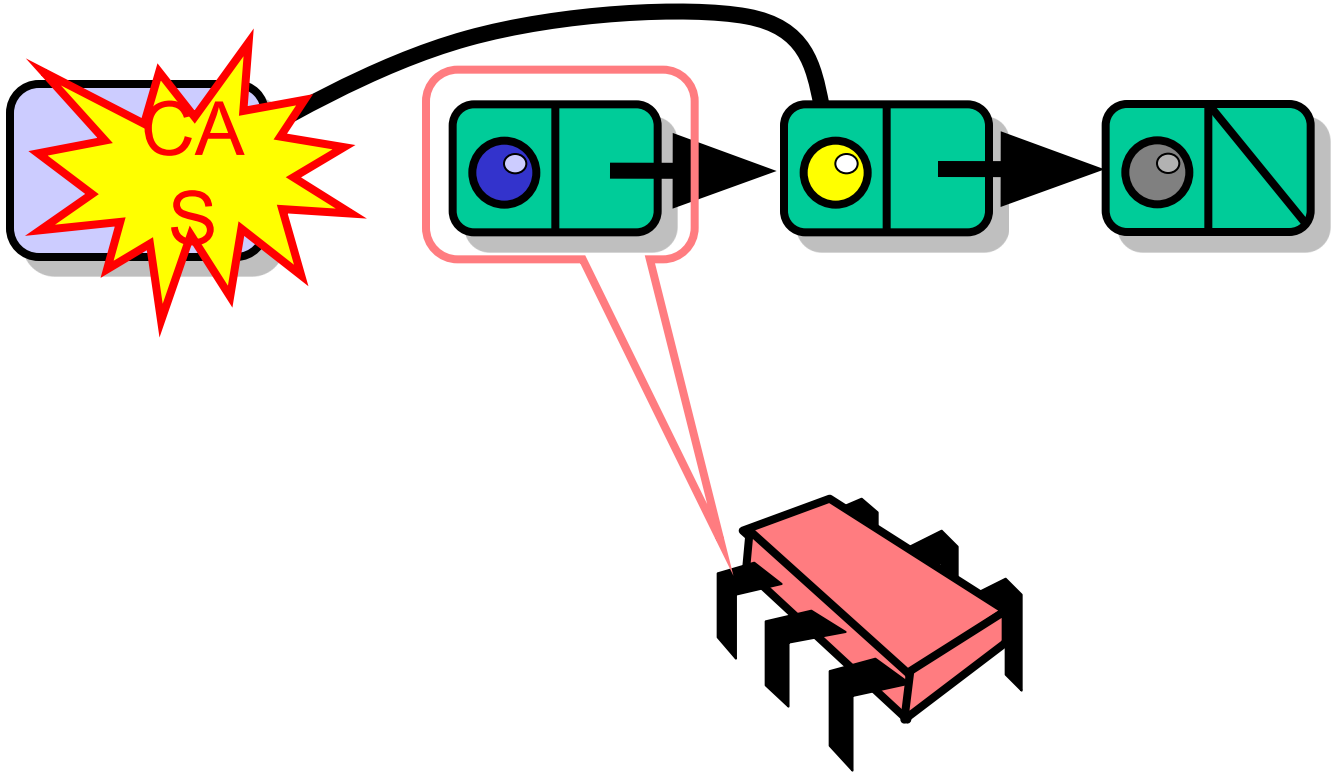
Push



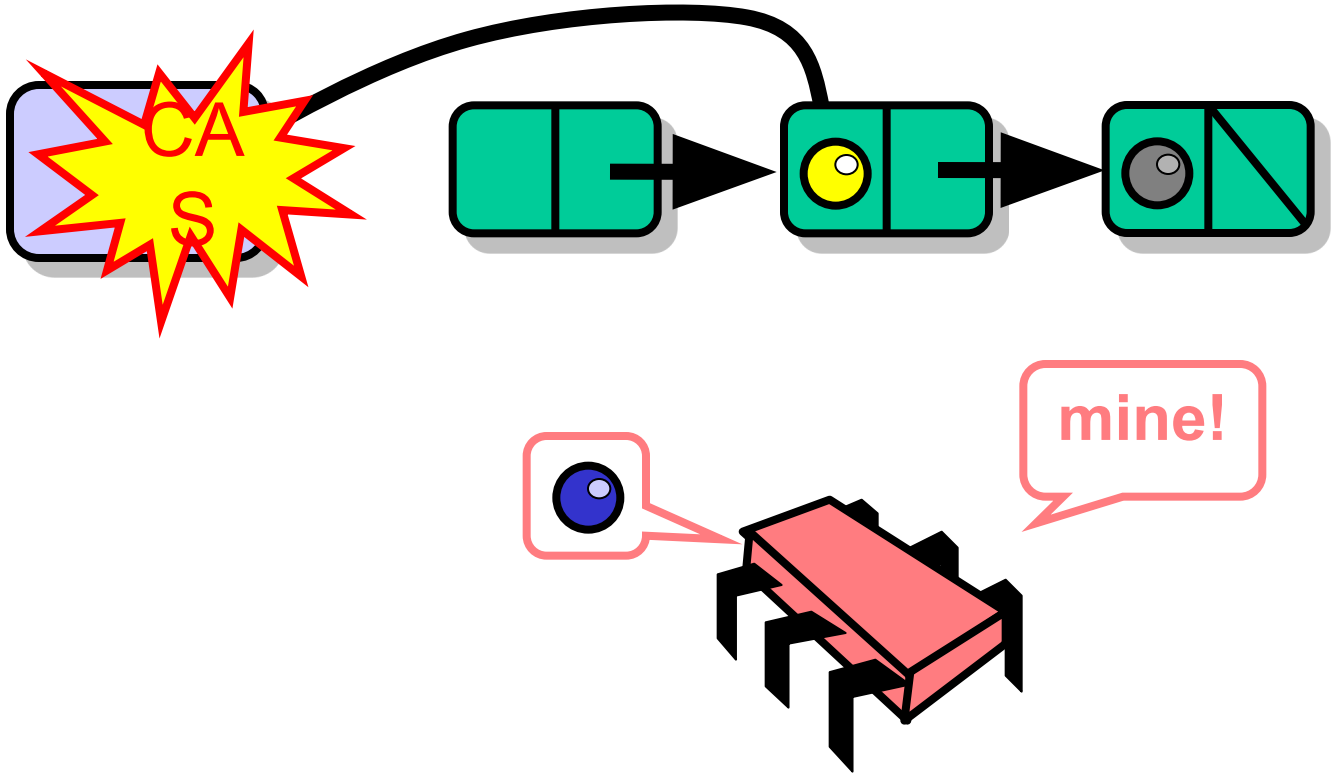
Pop



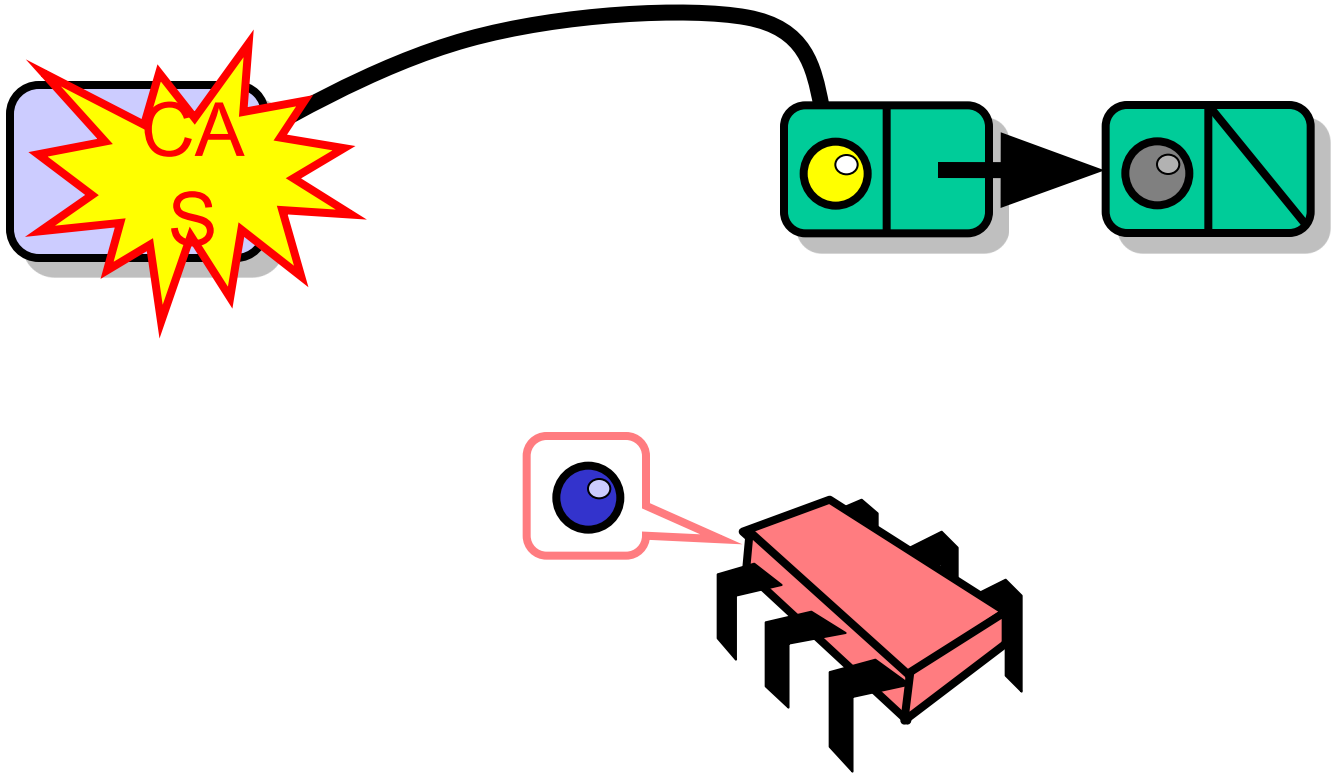
Pop



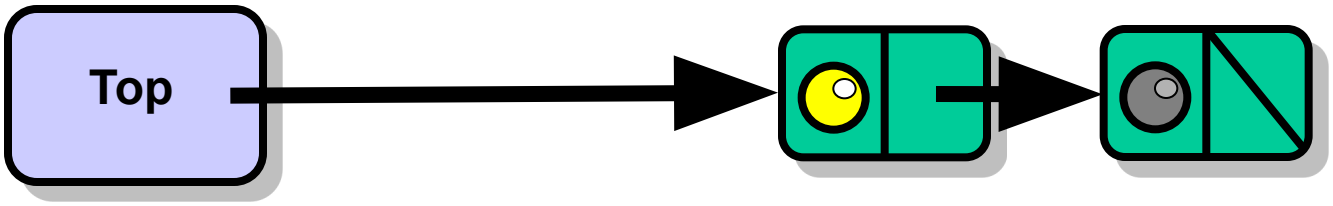
Pop

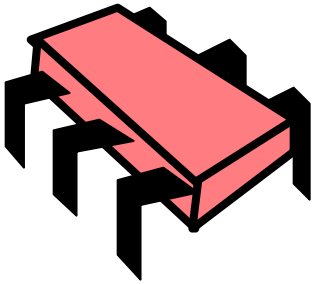


Pop



Pop





Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top =
    new AtomicReference(null);
  public 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();
  }}
```

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public 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;
}}
```

tryPush attempts to push a node

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public 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()
}}
```

Read top value

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public 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;
}}
```

current top will be new node's successor

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public 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;
```

Try to swing top, return success or failure

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public 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();
}}
```

Push calls tryPush

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public 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()
}}
```

Create new node

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
public 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()
}}
```

If tryPush() fails, back off before retrying

Lock-free Stack

- Good
  - No locking
- Bad
  - Without GC, fear ABA
  - Without backoff, huge contention at top
  - In any case, no parallelism

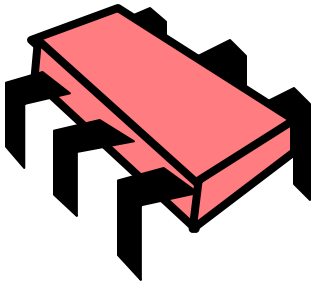
Big Question

- Are stacks *inherently* sequential?
- Reasons why
  - Every **pop()** call fights for top item
- Reasons why not
  - Stay tuned …

Elimination-Backoff Stack

- How to
  - "turn contention into parallelism"
- Replace familiar
  - **exponential backoff**
- With alternative
  - **elimination-backoff**

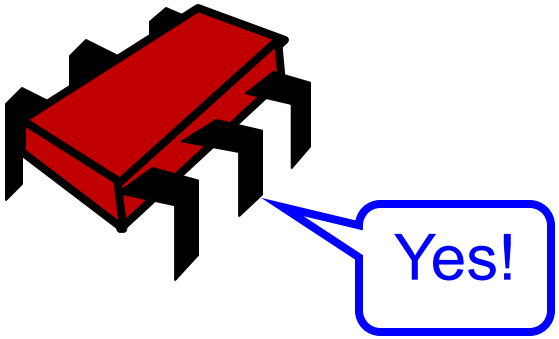
Observation



Push( )

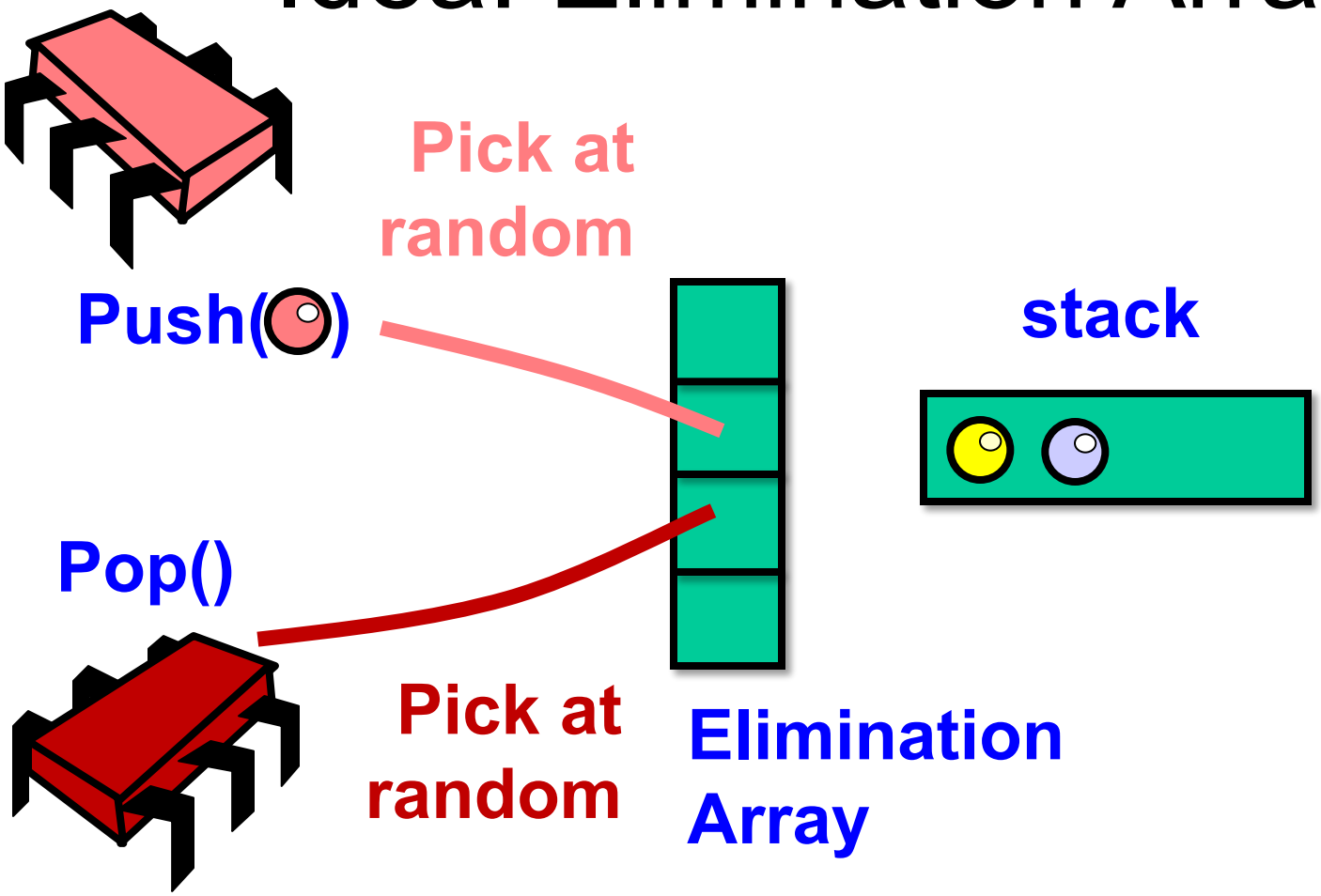
linearizable stack

Pop()

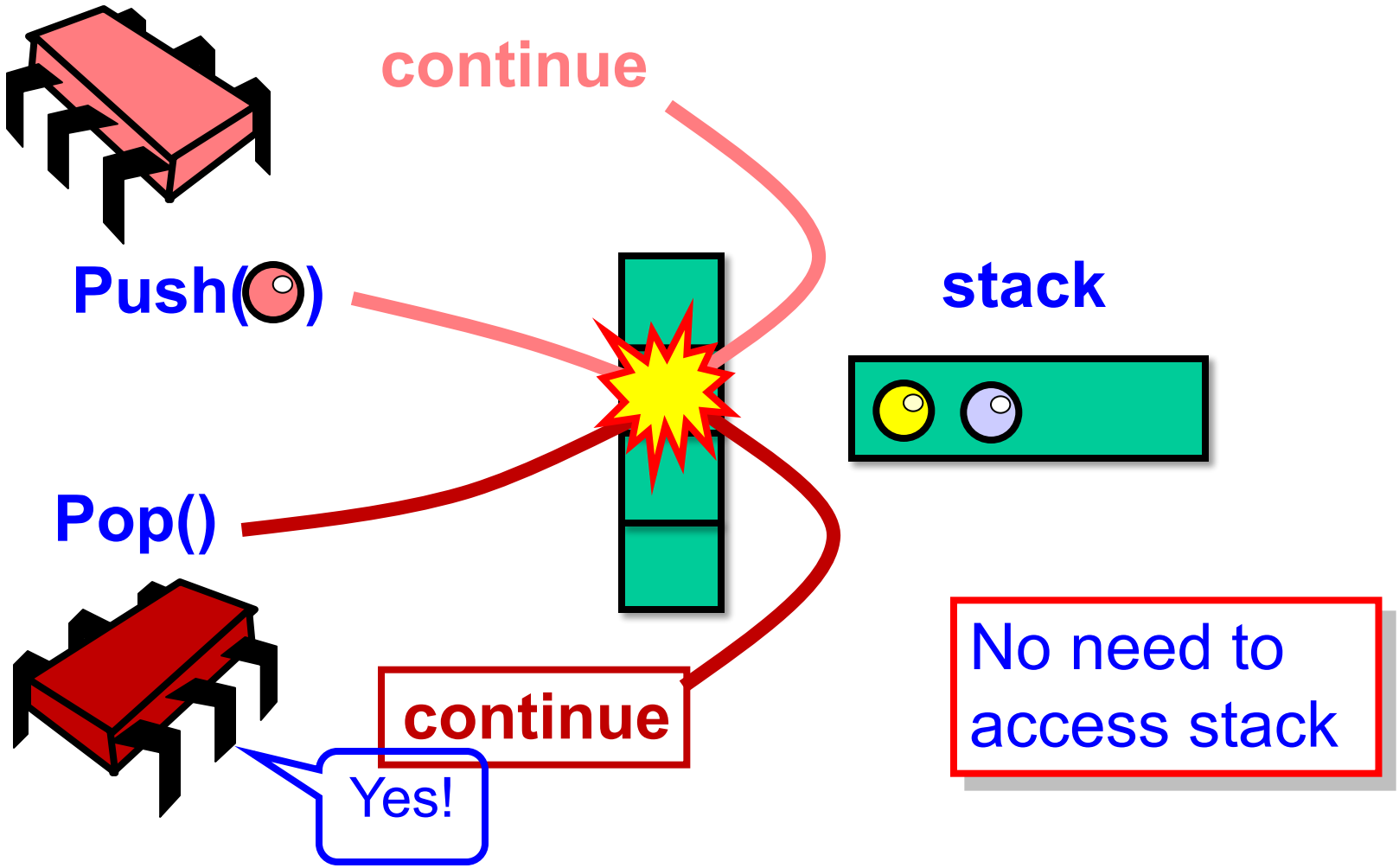


After an equal number of pushes and pops, stack stays the same

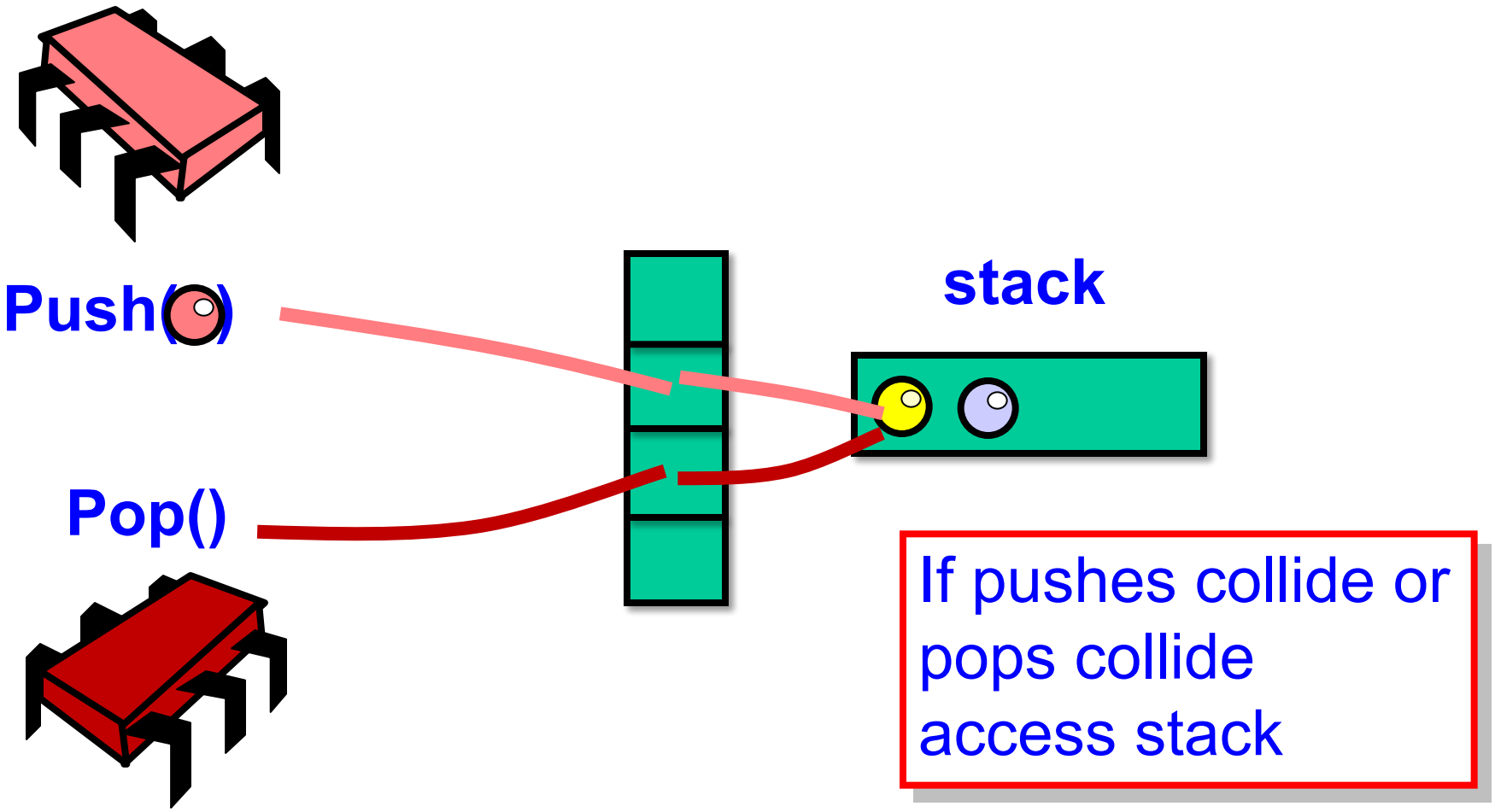
Idea: Elimination Array



Push Collides With Pop



No Collision

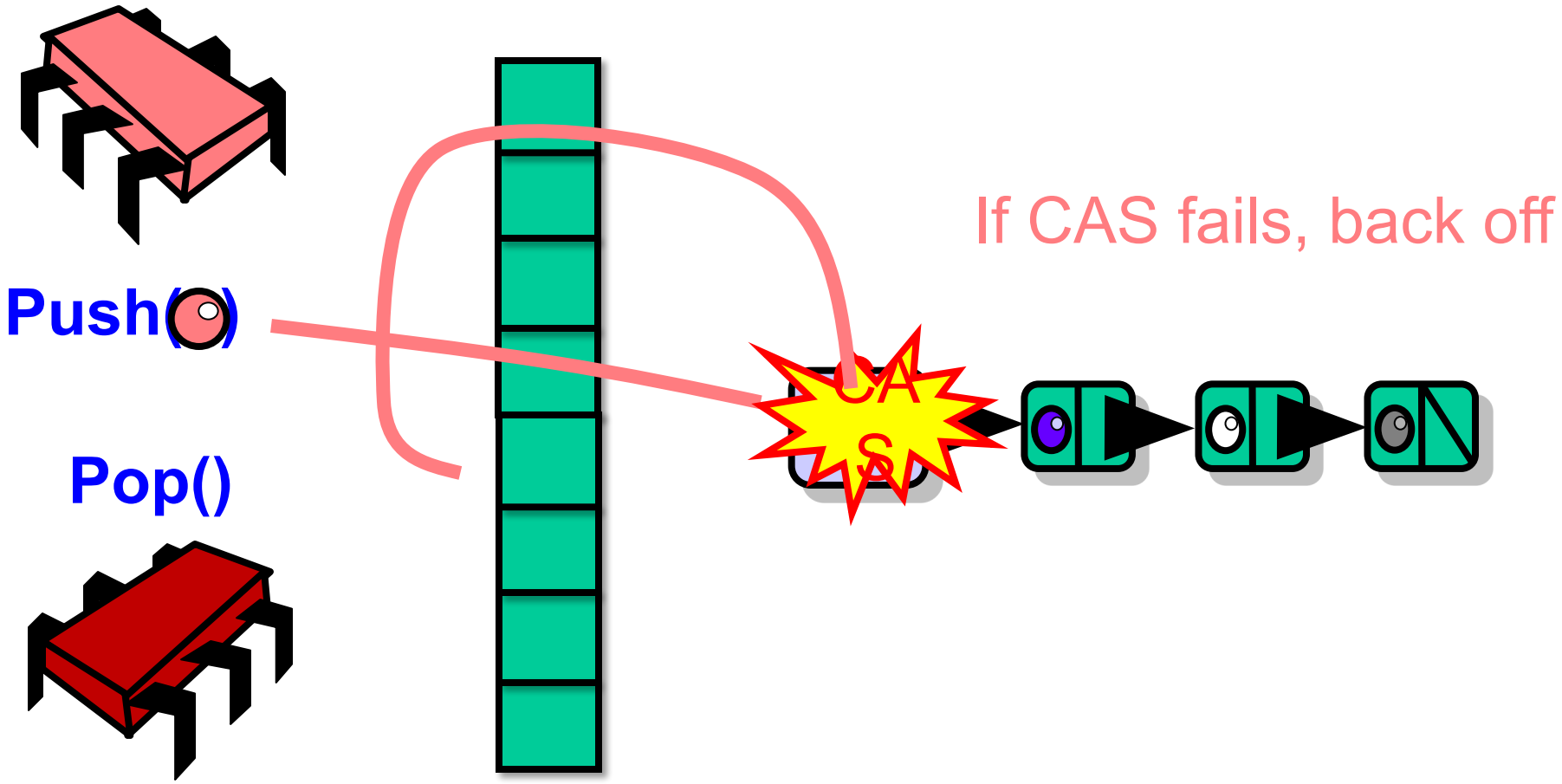


If pushes collide or pops collide access stack

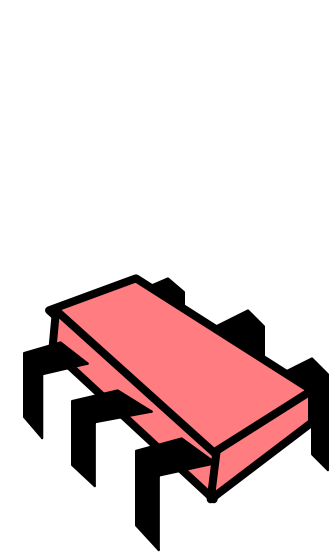
Elimination-Backoff Stack

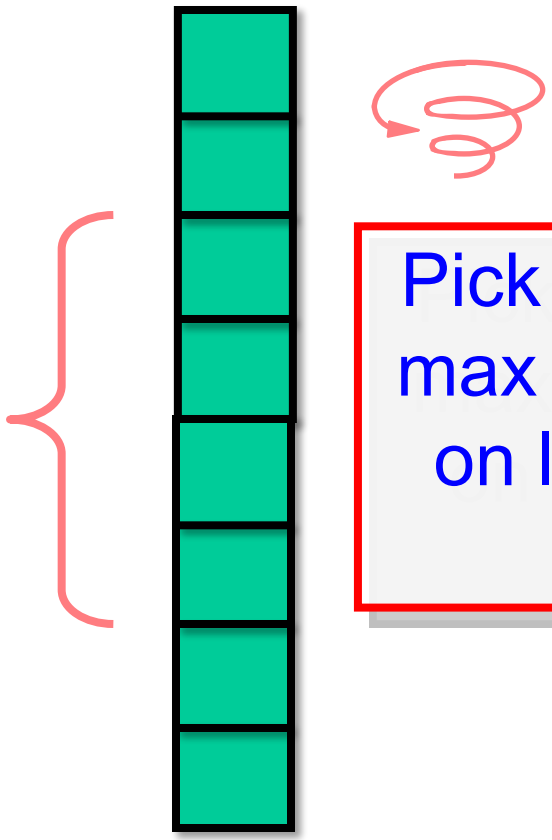
- Lock-free stack + elimination array
- Access Lock-free stack,
  - If **uncontended**, apply operation
  - if **contended**, back off to elimination array and attempt elimination

Elimination-Backoff Stack



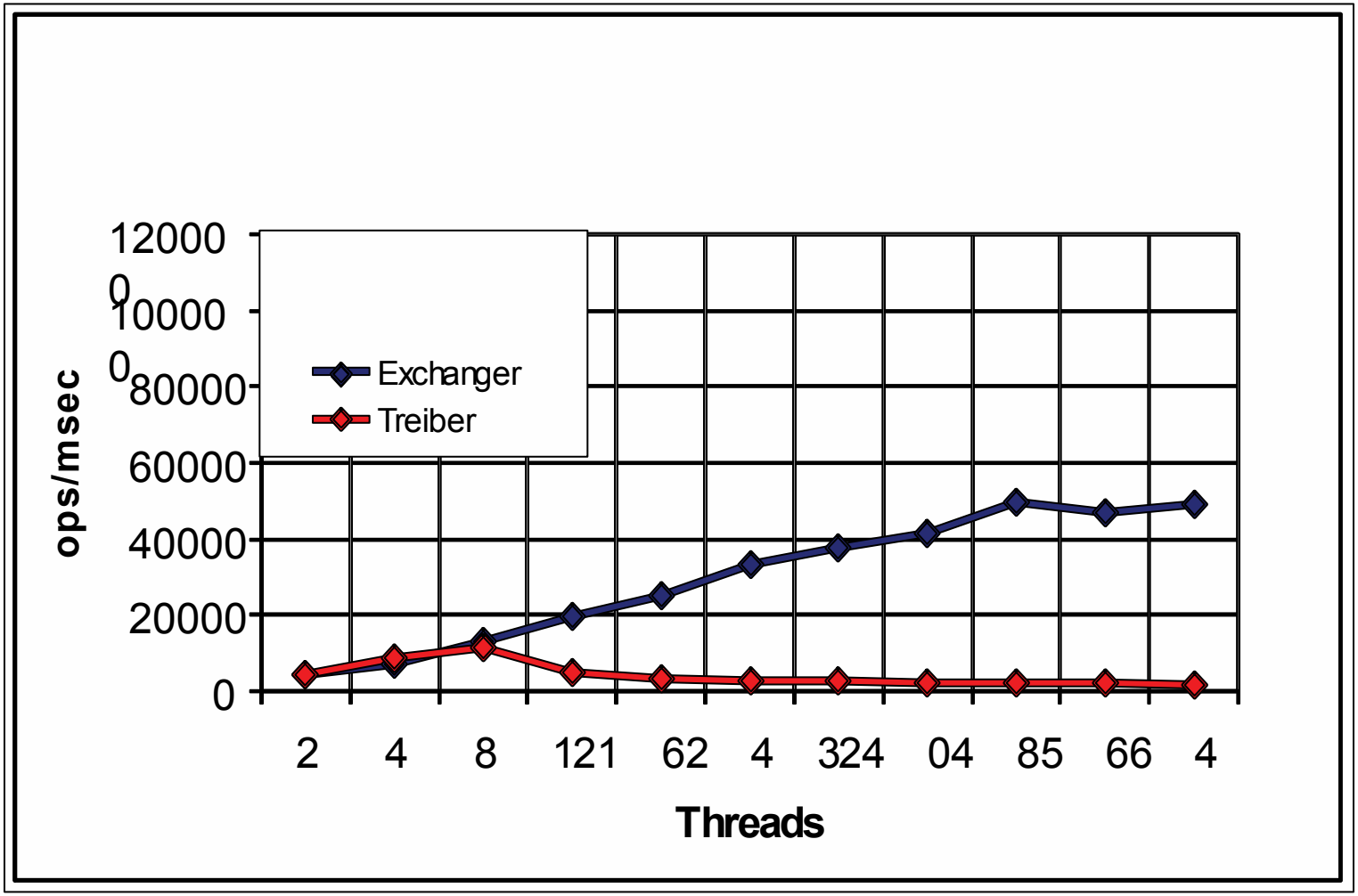
Dynamic Range and Delay



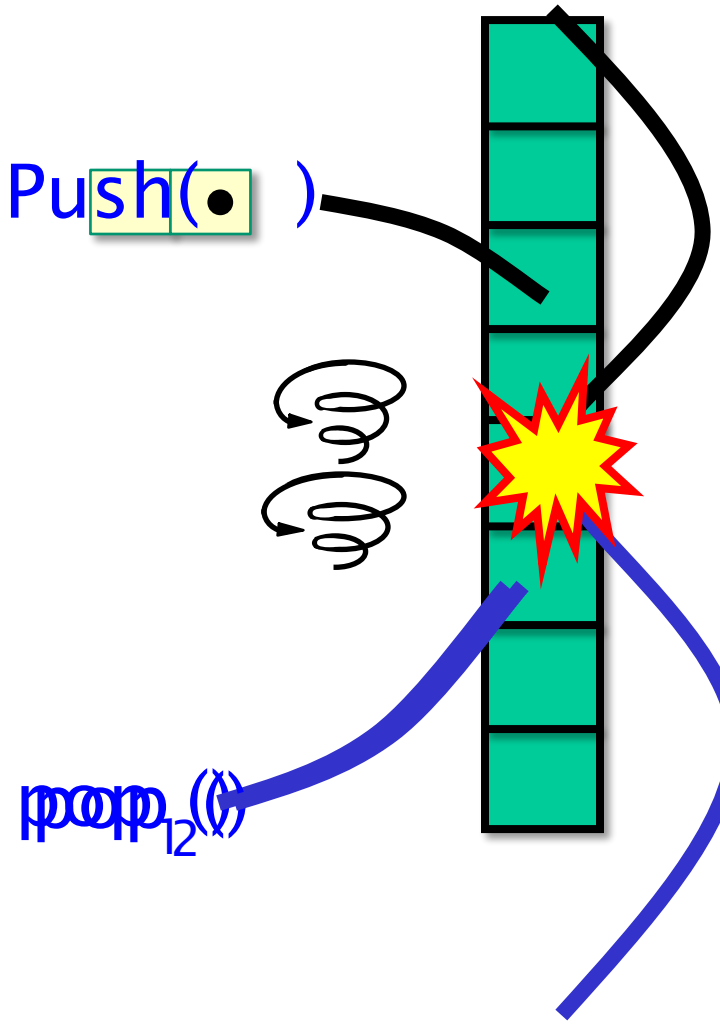


Pick random range and max waiting time based on level of contention encountered

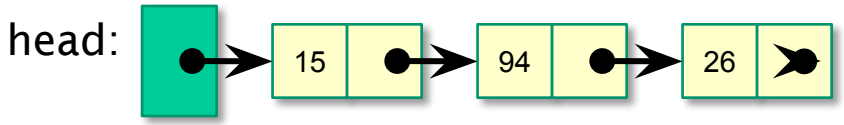
50-50, Random Slots

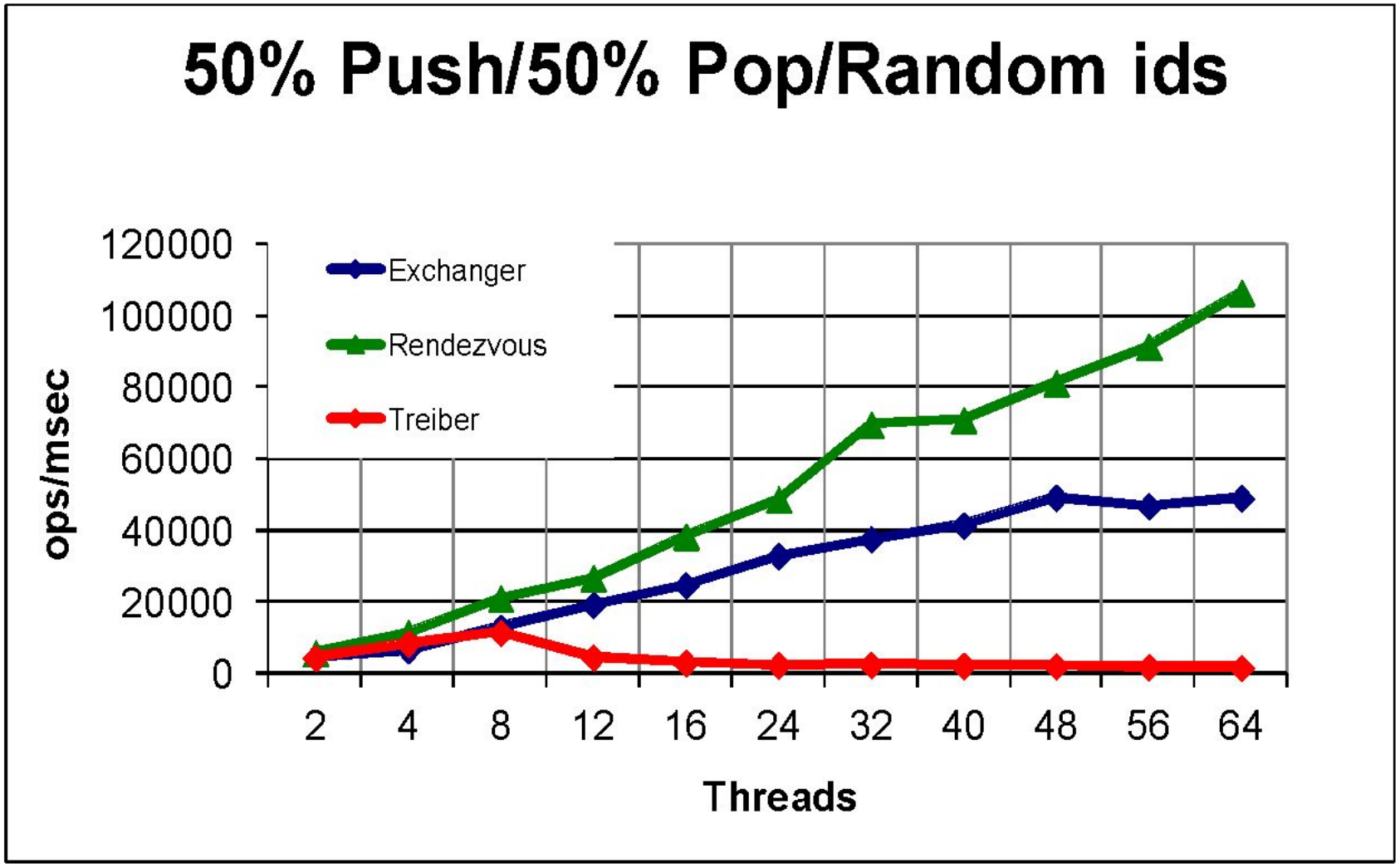


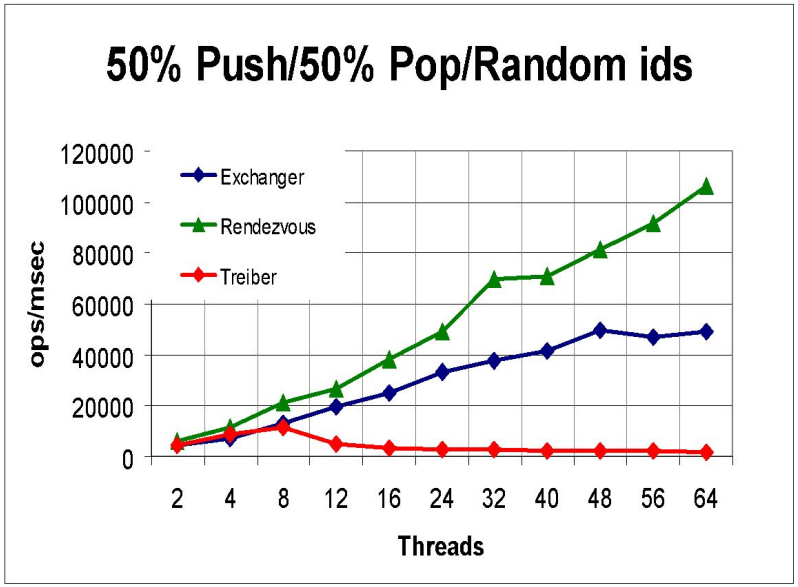
Asymmetric Rendevous

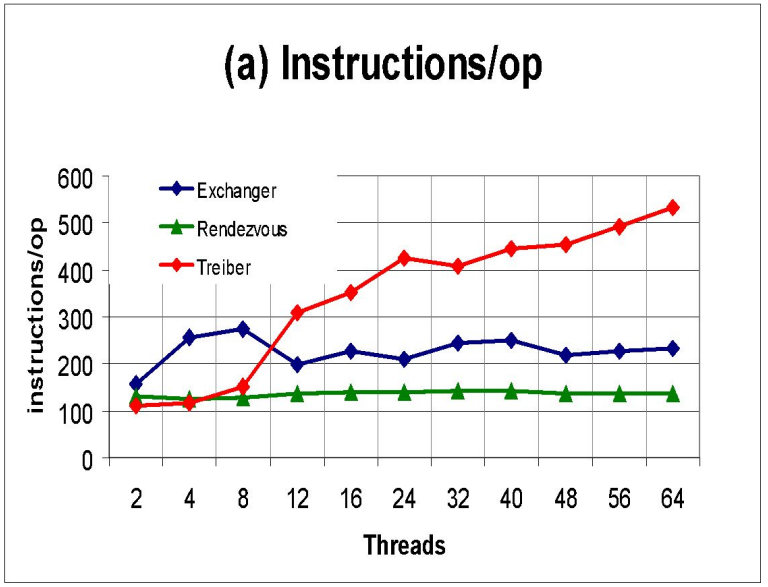


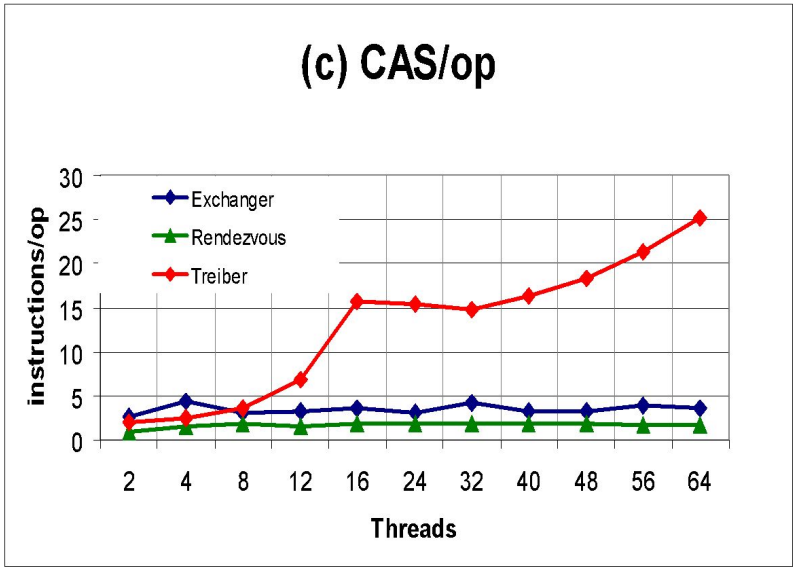
Pops find first vacant slot and spin. Pushes hunt for pops.

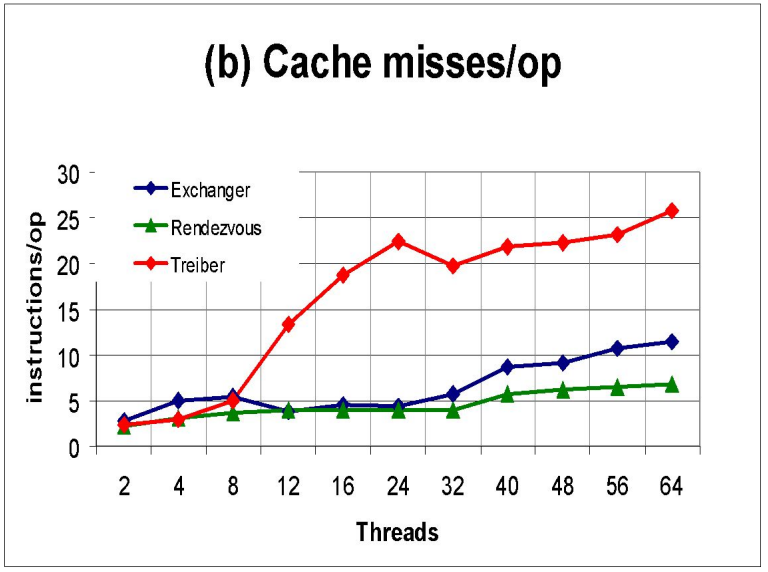



50% Push/50% Pop/Random ids
ops/msec
120000
100000
80000
60000
40000
20000
0
Exchanger
Rendezvous
Treiber
2
4
8
12
16
24
32
40
48
56
64
Threads

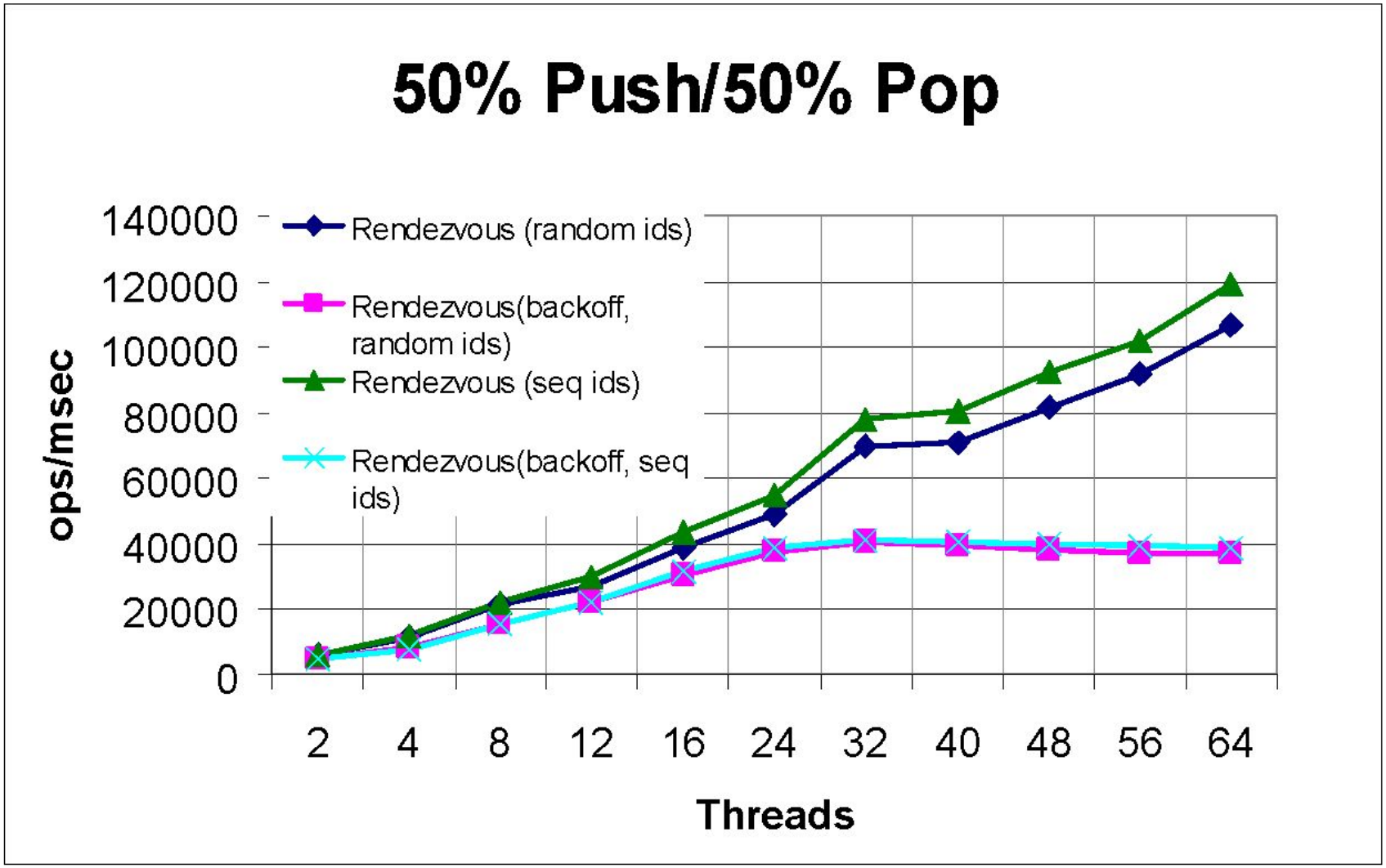

50% Push/50% Pop/Random ids
ops/msec
120000
100000
80000
60000
40000
20000
0
Exchanger
Rendezvous
Treiber
2 4 8 12 16 24 32 40 48 56 64
Threads


(a) Instructions/op
instructions/op
600
500
400
300
200
100
0
Exchanger
Rendezvous
Treiber
2 4 8 12 16 24 32 40 48 56 64
Threads


(c) CAS/op
instructions/op
30
25
20
15
10
5
0
Exchanger
Rendezvous
Treiber
2 4 8 12 16 24 32 40 48 56 64
Threads

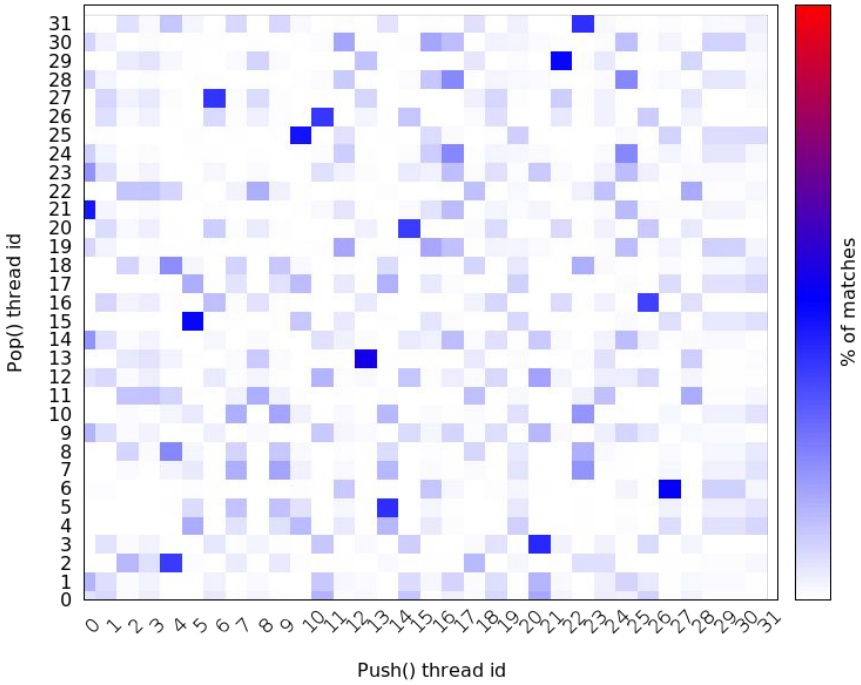

(b) Cache misses/op
instructions/op
30
25
20
15
10
5
0
Exchanger
Rendezvous
Treiber
2 4 8 12 16 24 32 40 48 56 64
Threads

Effect of Backoff and Slot Choice

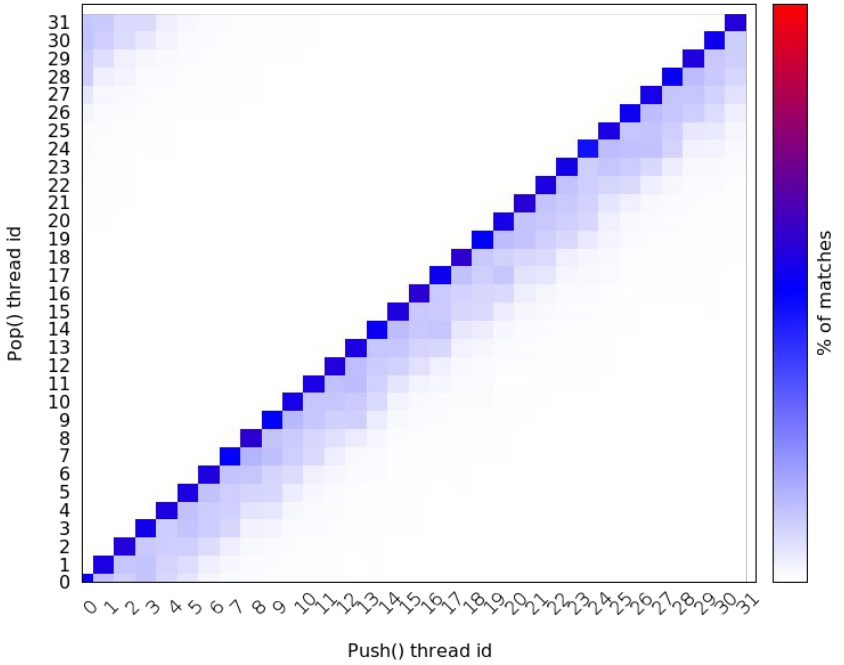


Effect of Slot Choice

Random choice

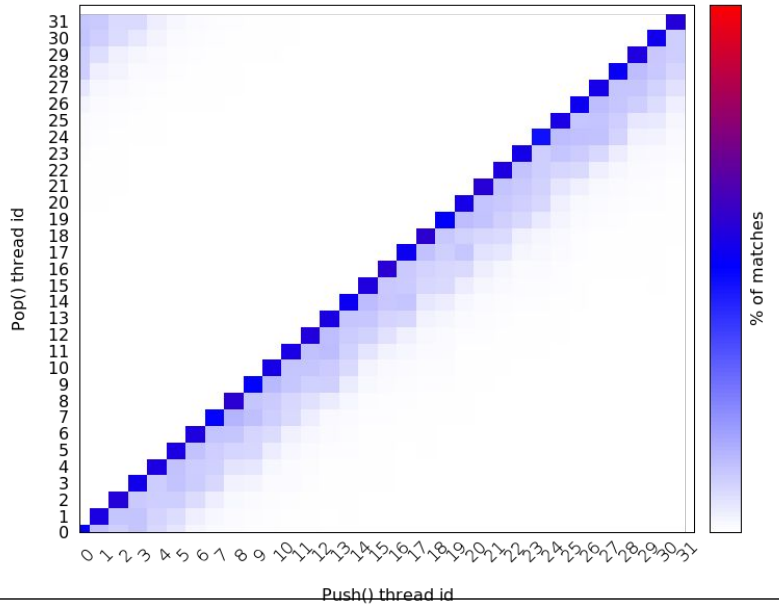


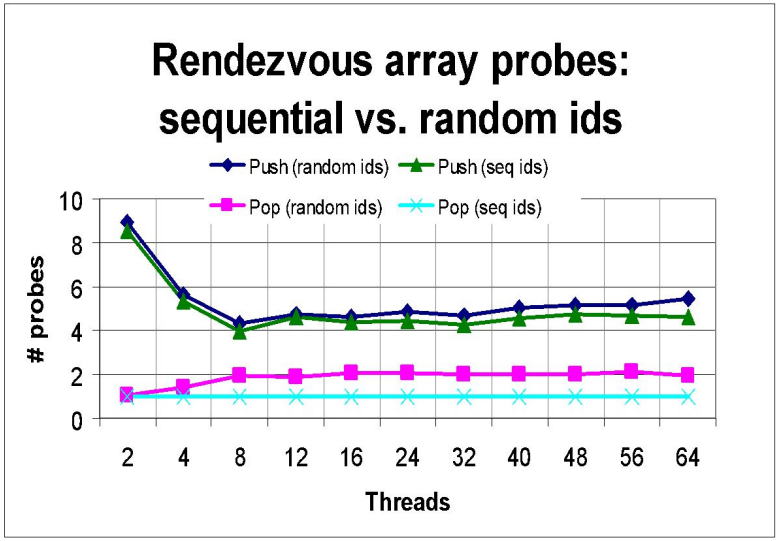
Sequential choice

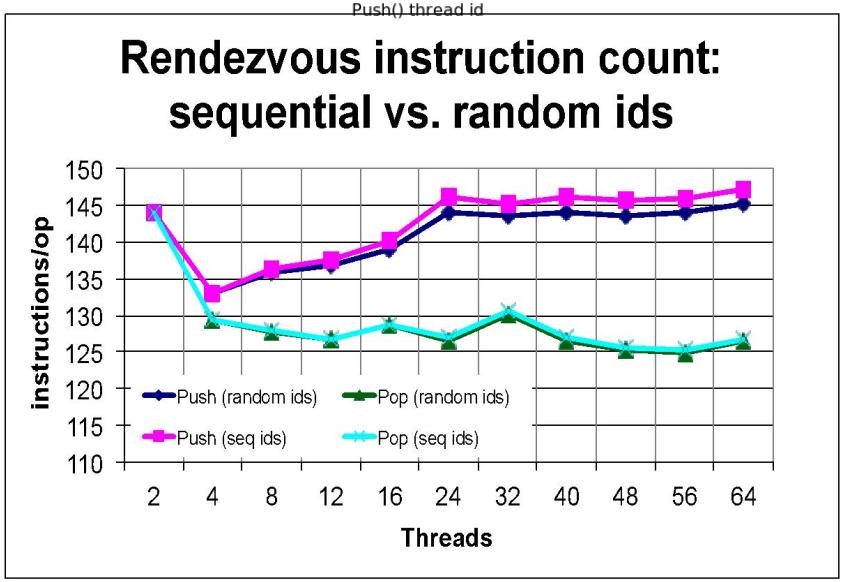


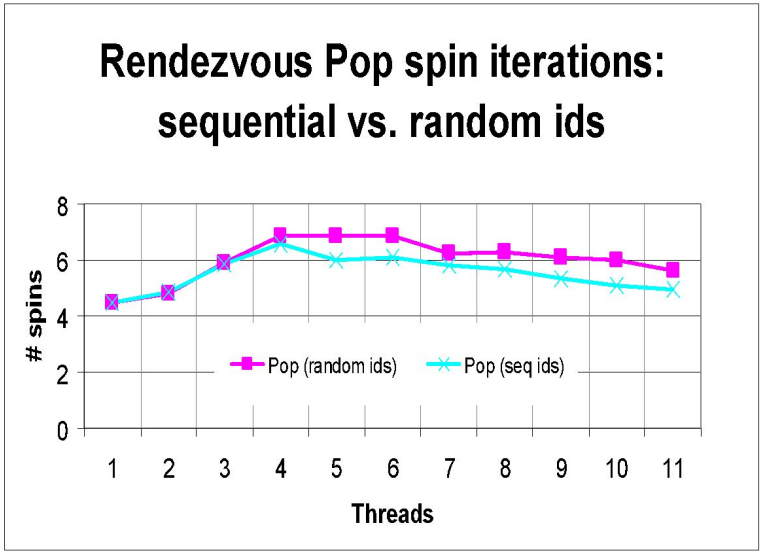
Darker shades mean more exchanges

Effect of Slot Choice





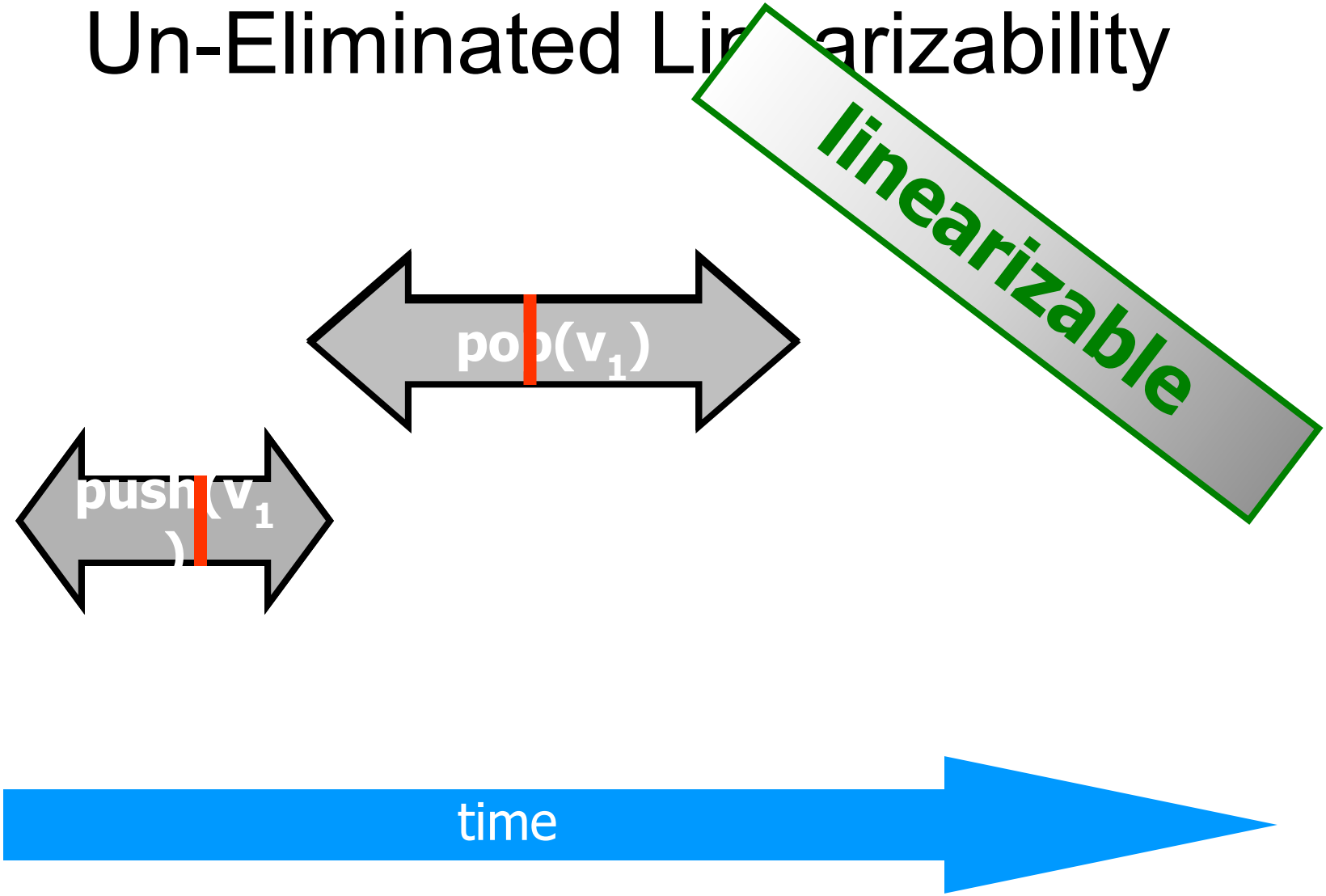




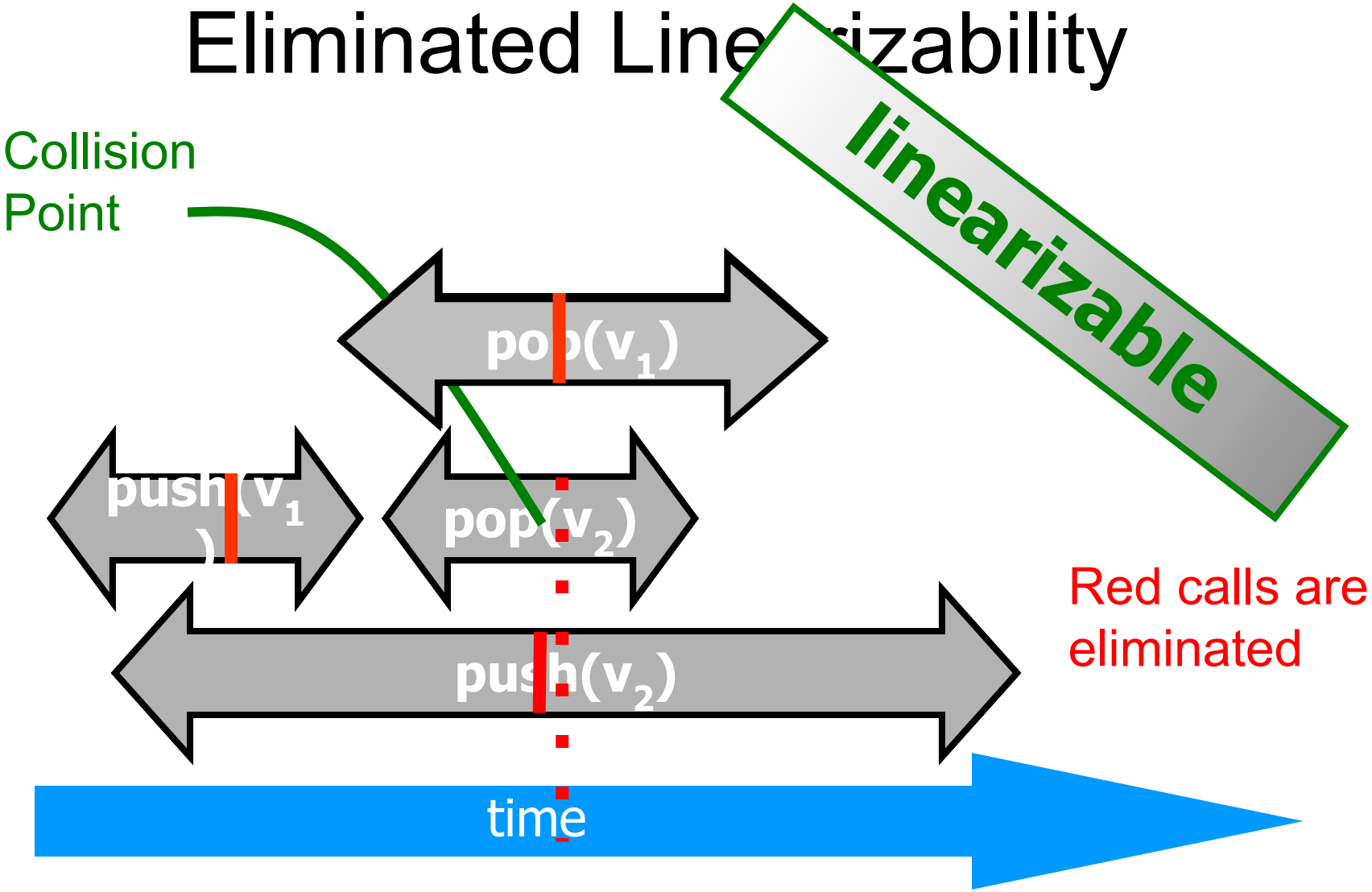
Linearizability

- **Un-eliminated calls**
  - **linearized as before**
- **Eliminated calls:**
  - **linearize pop() immediately after matching push()**
- **Combination is a linearizable stack**

Un-Eliminated Linearizability



Eliminated Linearizability



Backoff Has Dual Effect

- Elimination introduces parallelism
- Backoff to array cuts contention on lock-free stack
- Elimination in array cuts down number of threads accessing lock-free stack

Elimination Array

```
public class EliminationArray {
 private static final int duration = ...;
 private static final int timeUnit = ...;
 Exchanger<T>[] exchanger;
 public EliminationArray(int capacity) {
  exchanger = new Exchanger[capacity];
  for (int i = 0; i < capacity; i++)
   exchanger[i] = new Exchanger<T>();
  …
  }
  …
}
```

Elimination Array

```
public class EliminationArray {
 private static final int duration = ...;
 private static final int timeUnit = ...;
 Exchanger<T>[] exchanger;
 public EliminationArray(int capacity) {
  exchanger = new Exchanger[capacity];
  for (int i = 0; i < capacity; i++)
   exchanger[i] = new Exchanger<T>();
  ...
  }
  ...
}
```

An array of *Exchangers*

Digression: A Lock-Free Exchanger

```
public class Exchanger<T> {
 AtomicStampedReference<T> slot
  = new AtomicStampedReference<T>(null, 0);
```

A Lock-Free Exchanger

```
public class Exchanger<T> {
  AtomicStampedReference<T> slot
    = new AtomicStampedReference<T>(null, 0);
```

Atomically modifiable reference + status

Atomic Stamped Reference

- **AtomicStampedReference** class
  - Java.util.concurrent.atomic package
- In C or C++:

Extracting Reference & Stamp

```
public T get(int[] stampHolder);
```

Extracting Reference & Stamp

```
public T get(int[] stampHolder);
```

Returns reference to object of type T

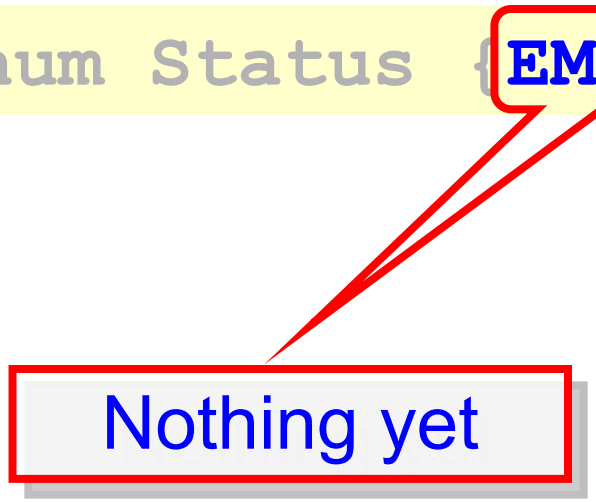
Returns stamp at array index 0!

Exchanger Status

```
enum Status {EMPTY, WAITING, BUSY};
```

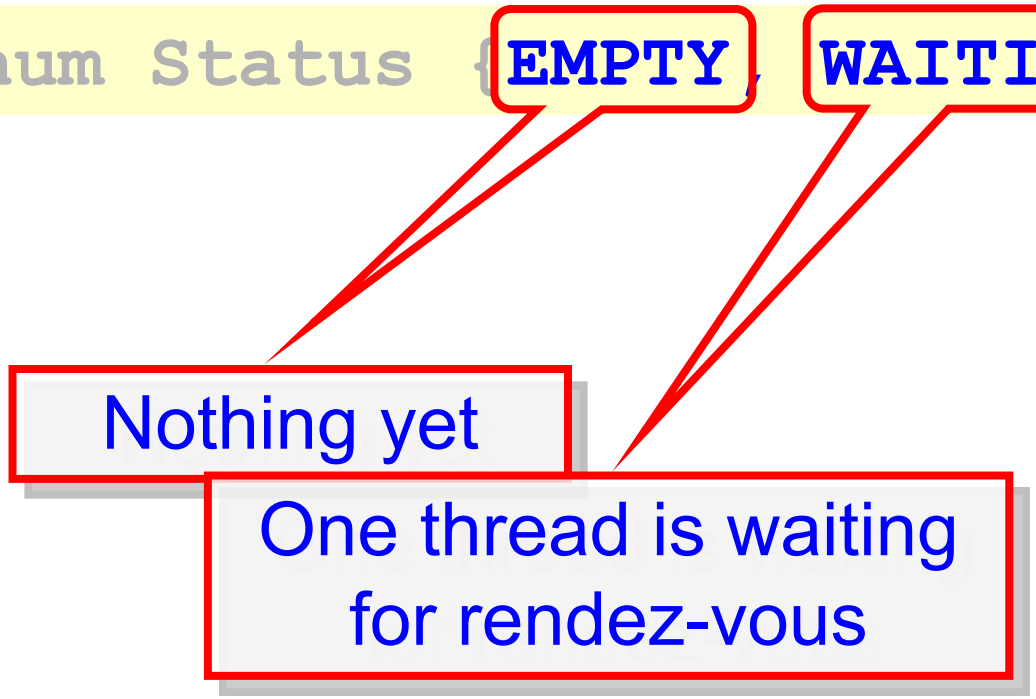
Exchanger Status

```
enum Status {EMPTY, WAITING, BUSY};
```

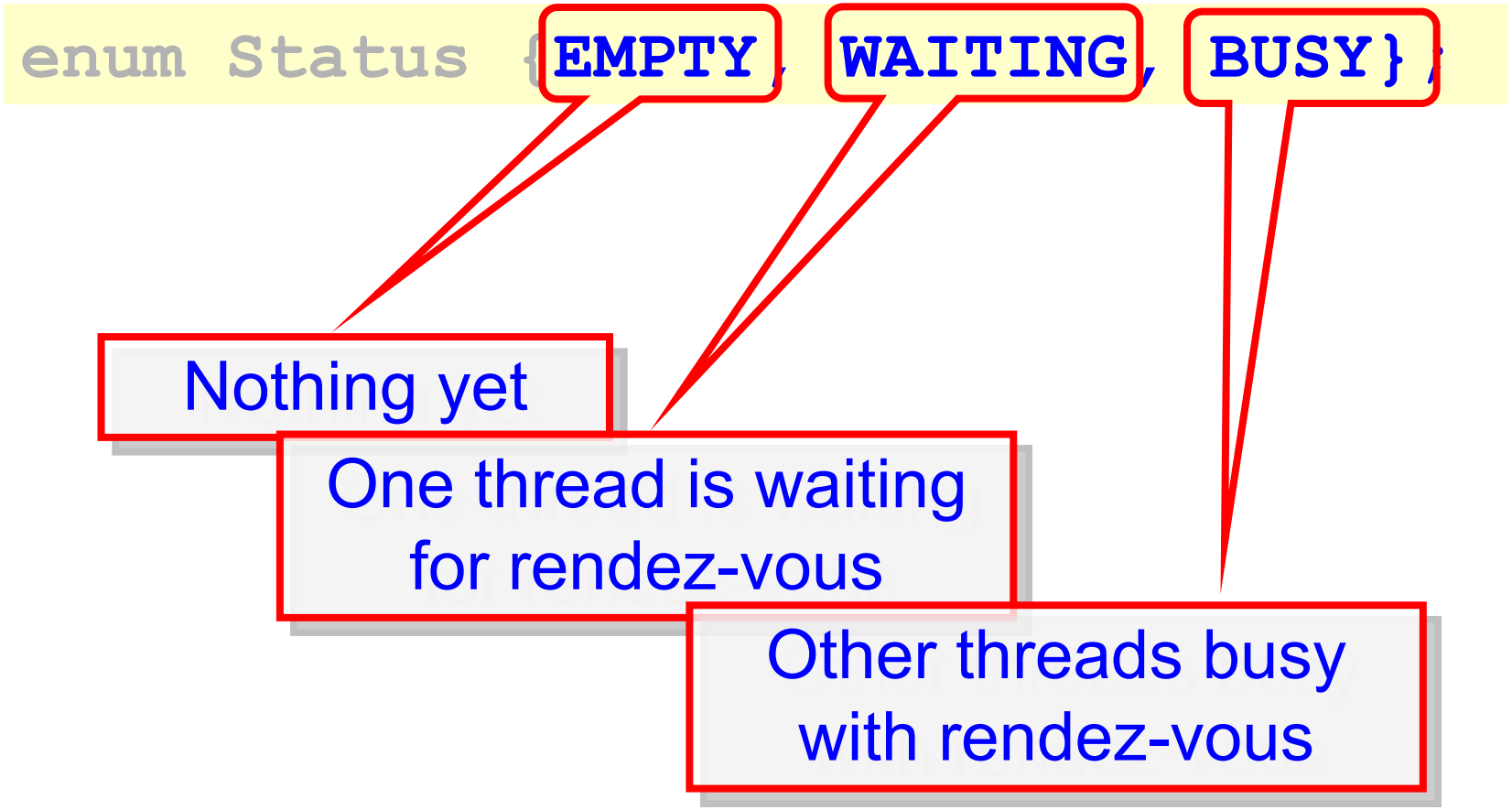


Exchange Status

```
enum Status {EMPTY, WAITING, BUSY};
```



Exchange Status



The Exchange

```
public T Exchange(T myItem, long nanos)
   throws TimeoutException {
 long timeBound = System.nanoTime() + nanos;
 int[] stampHolder = {EMPTY};
 while (true) {
  if (System.nanoTime() > timeBound)
    throw new TimeoutException();
  T herItem = slot.get(stampHolder);
  int stamp = stampHolder[0];
  switch(stamp) {
   case EMPTY: …    // slot is free
   case WAITING: … // someone waiting for me
   case BUSY: …     // others exchanging
  }
 }
```

The Exchange

```
public T Exchange(T myItem, long nanos)
    throws TimeoutException {
  long timeBound = System.nanoTime() + nanos;
  int[] stampHolder = {EMPTY};
  while (true) {
    if (System.nanoTime() > timeBound)
      throw new TimeoutException();
    T herItem = slot.get(stampHolder);
    int stamp = stampHolder[0];
    switch(stamp) {
      case EMPTY: …    // slot is free
      case WAITING: … // someone waiting for me
      case BUSY: …     // others exchanging
    }
  }
```

Item and timeout

The Exchange

```
public T Exchange(T myItem, long nanos)
   throws TimeoutException {
 long timeBound = System.nanoTime() + nanos;
 int[] stampHolder = {EMPTY};
 while (true) {
  if (System.nanoTime() > timeBound)
    throw new TimeoutException();
  T herItem = slot.get(stampHolder);
  int stamp = stampHolder[0];
  switch(stamp) {
   case EMPTY: …    // slot is free
   case WAITING: … // someone waiting for me
   case BUSY: …     // others exchanging
  }
 }
```

Array holds status

The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
    long timeBound = System.nanoTime() + nanos;
    int[] stampHolder = {0};
    while (true) {
      if (System.nanoTime() > timeBound)
        throw new TimeoutException();
      T herItem = slot.get(stampHolder);
      int stamp = stampHolder[0];
      switch(stamp) {
        case EMPTY:   // slot is free
        case WAITING: // someone waiting for me
        case BUSY:    // others exchanging
        }
    }}
```

Loop until timeout

The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
    long timeBound = System.nanoTime() + nanos;
    int[] stampHolder = {0};
    while (true) {
      if (System.nanoTime() > timeBound)
        throw new TimeoutException();
      T herItem = slot.get(stampHolder);
      int stamp = stampHolder[0];
      switch(stamp) {
        case EMPTY:   // slot is free
        case WAITING: // someone waiting for me
        case BUSY:    // others exchanging
        }
    }}
```

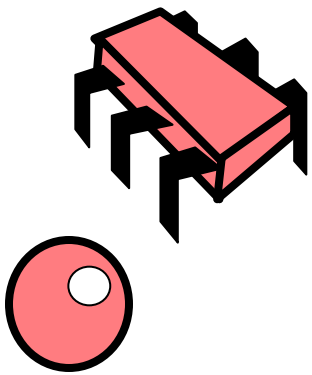
Get other's item and status

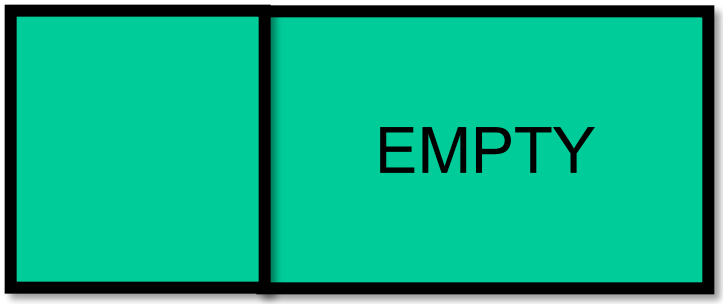
The Exchange

```
public T Exchange(T myItem, long nanos) throws
TimeoutException {
    long timeBound = System.nanoTime() + nanos;
    int[] stampHolder = {EMPTY};
    while (true) {
      if (System.nanoTime() > timeBound)
        throw new TimeoutException();
      T herItem = slot.get(stampHolder);
      int stamp = stampHolder[0];
      switch(stamp) {
        case EMPTY: …    // slot is free
        case WAITING: … // someone waiting for me
        case BUSY: …     // others exchanging
        }
    }}
```

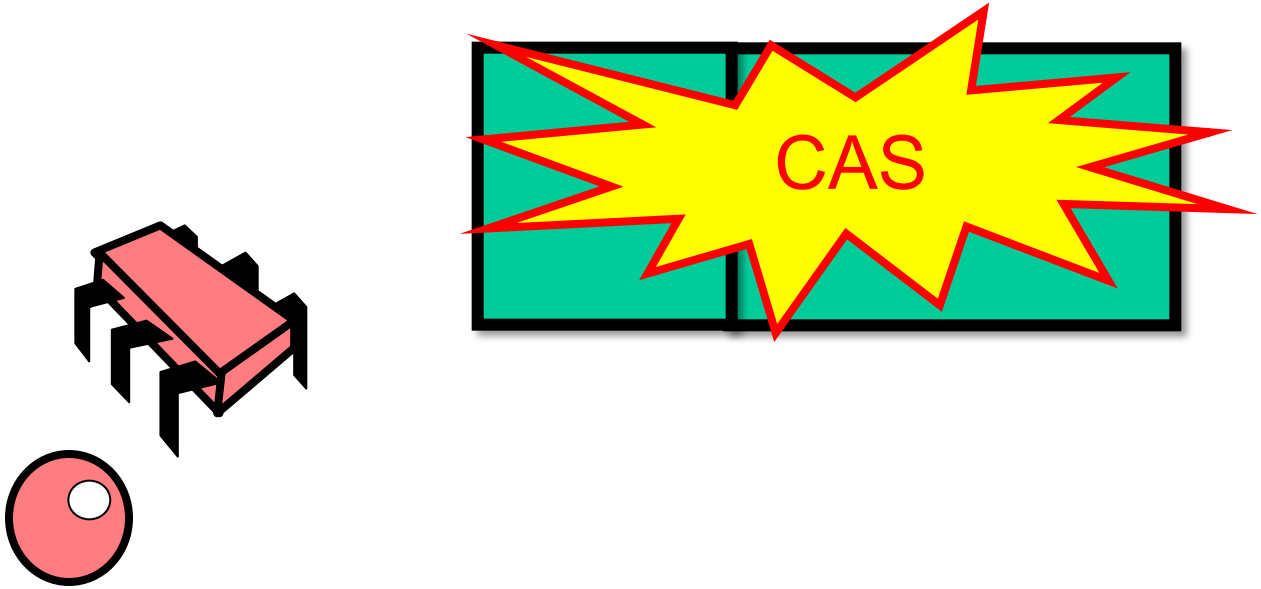
An *Exchanger* has three possible states

Lock-free Exchanger

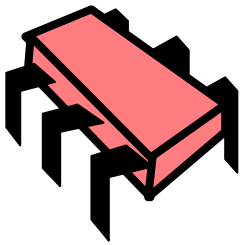


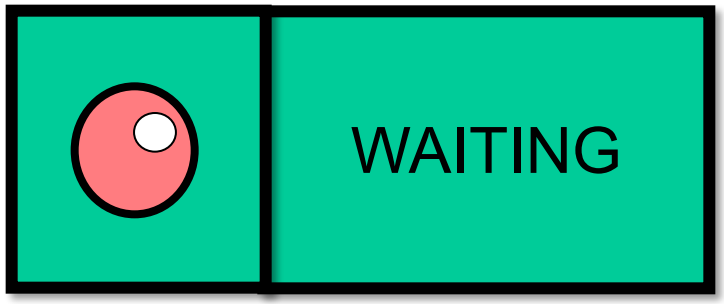


Lock-free Exchanger

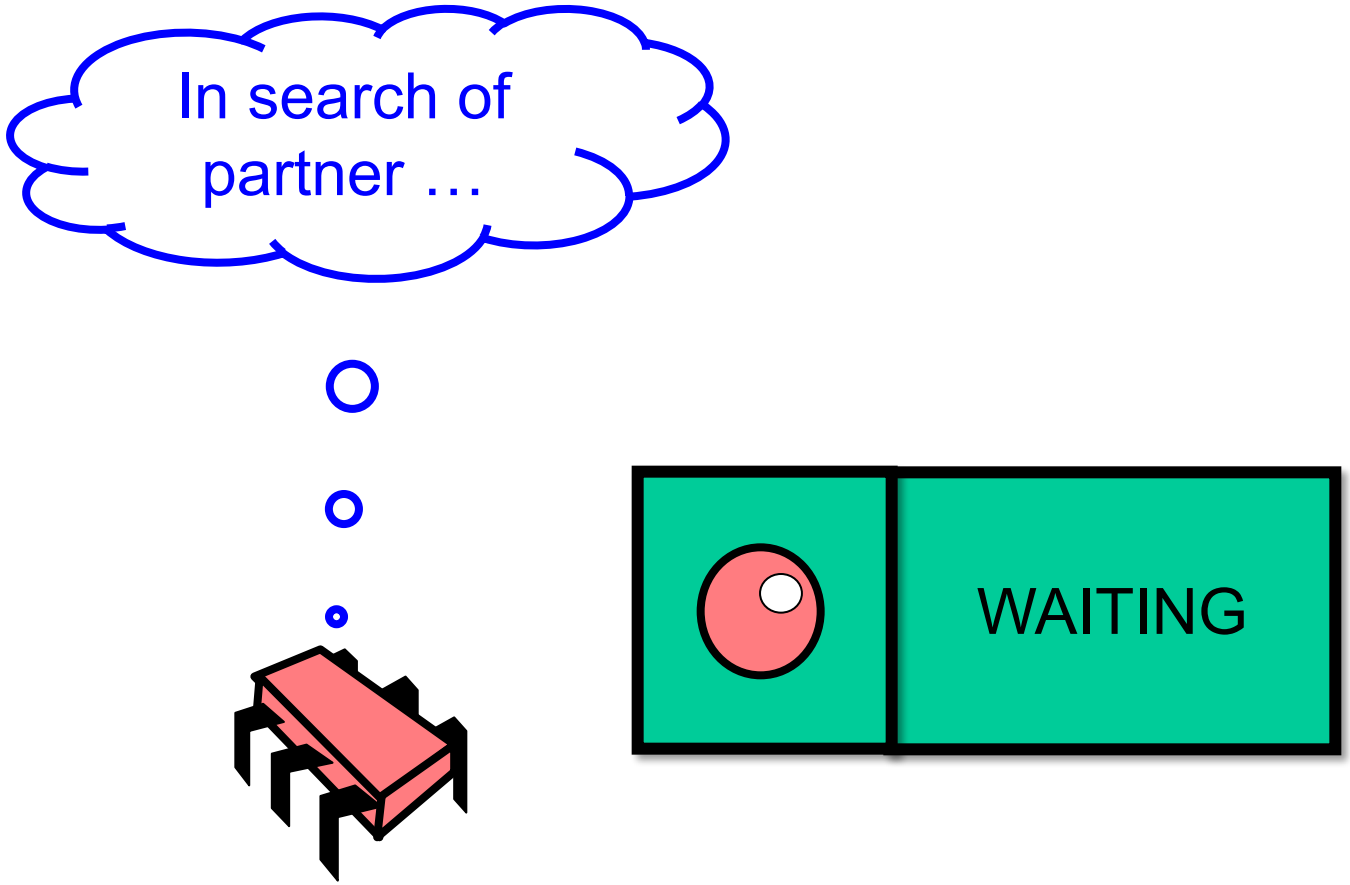


Lock-free Exchanger

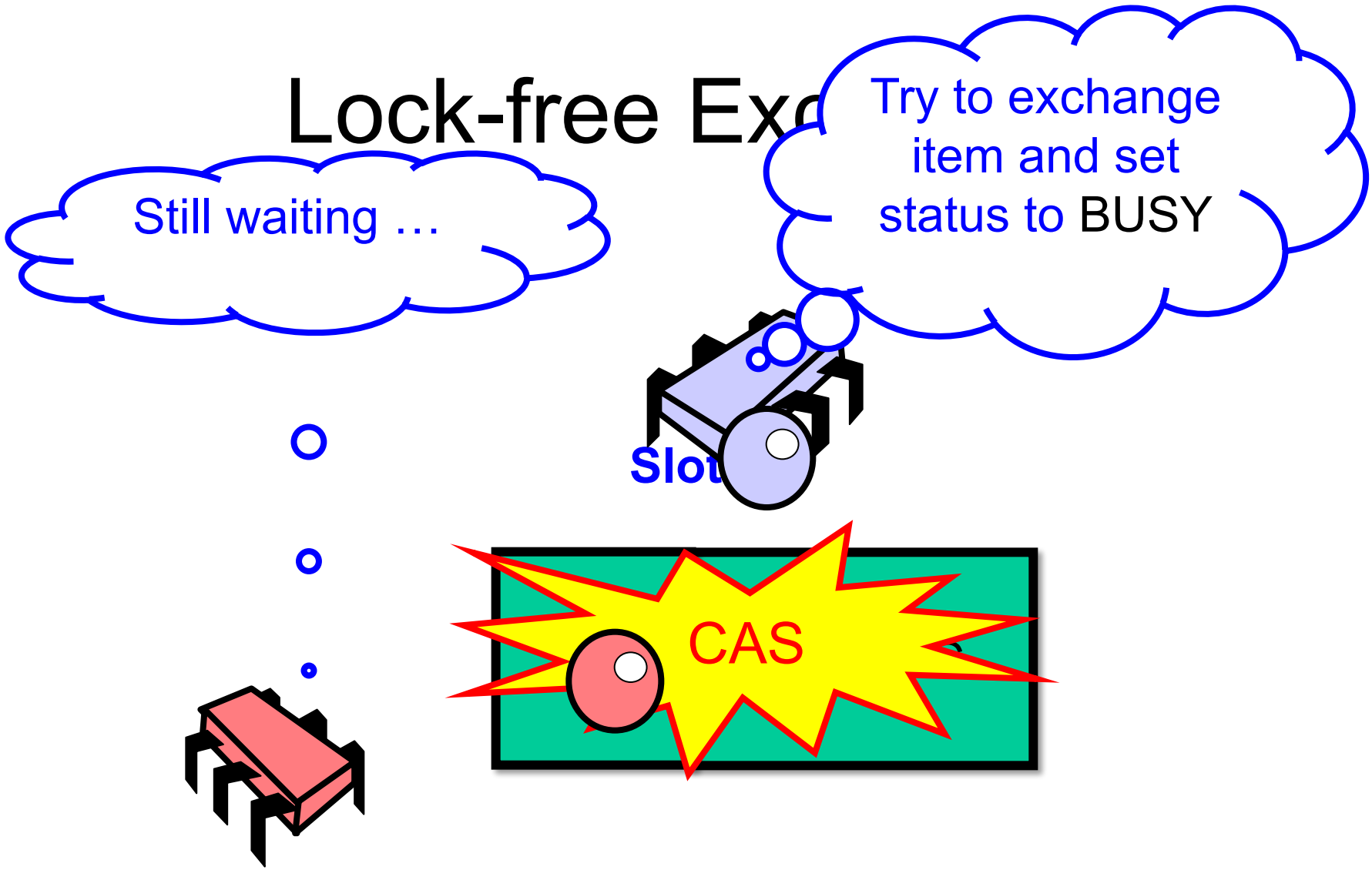




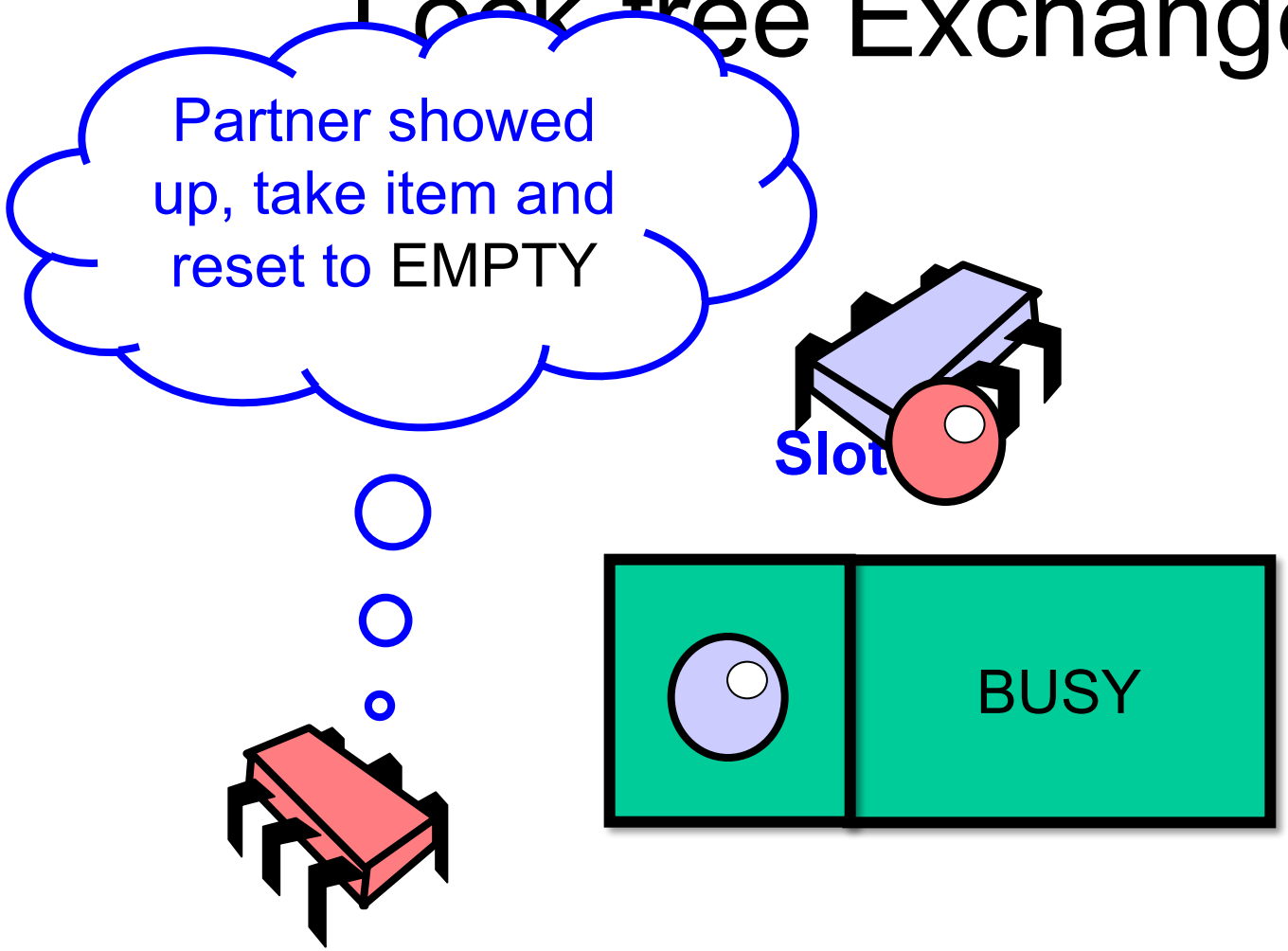
Lock-free Exchanger



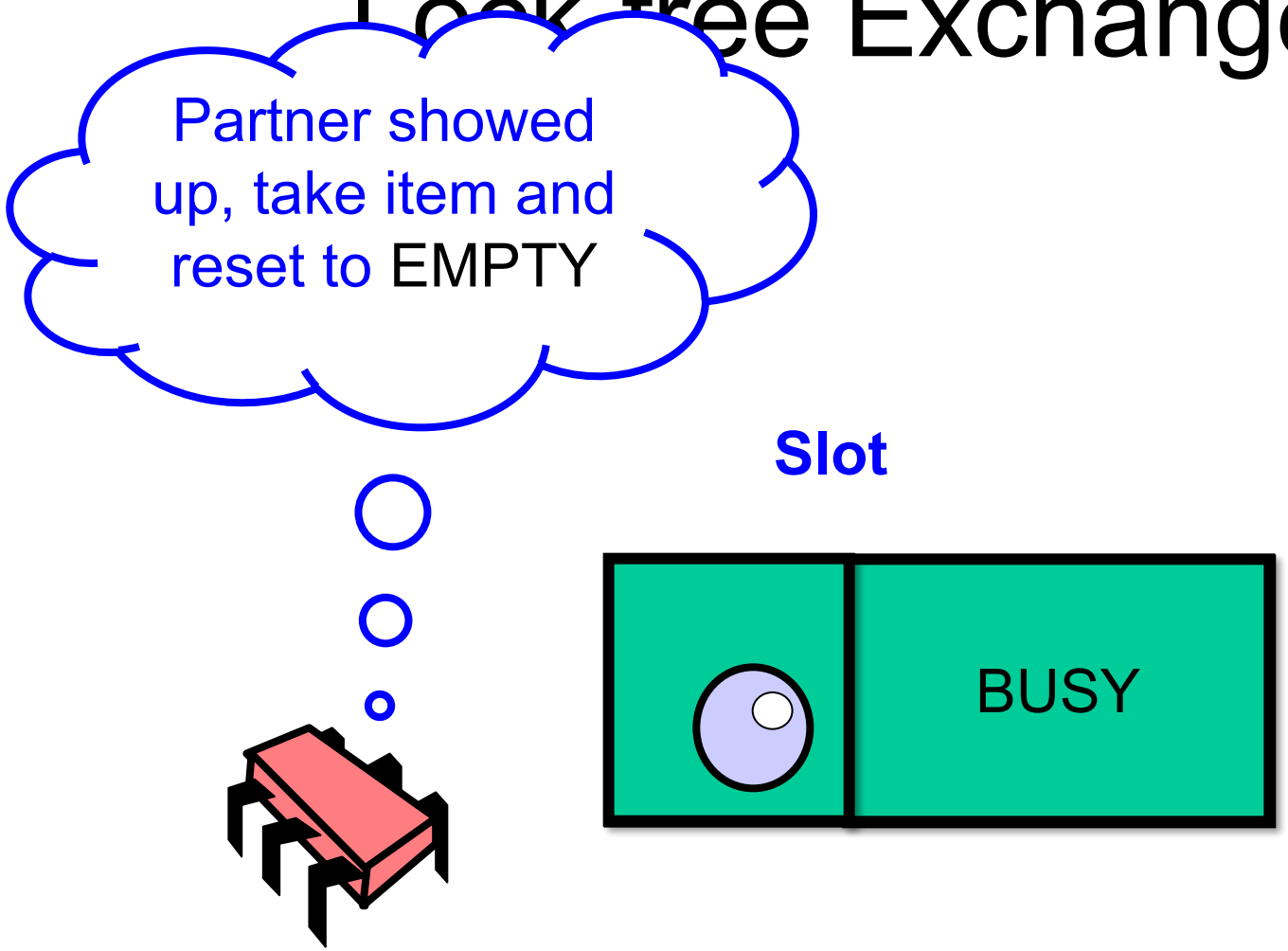
Lock-free Exc



Lock-free Exchanger



Lock-free Exchanger



Exchanger State EMPTY

```
case EMPTY: // slot is free
  if (slot.CAS(herItem, myItem, EMPTY, WAITING)) {
   while (System.nanoTime() < timeBound){
     herItem = slot.get(stampHolder);
     if (stampHolder[0] == BUSY) {
       slot.set(null, EMPTY);
       return herItem;
     }}
   if (slot.CAS(myItem, null, WAITING, EMPTY)){
      throw new TimeoutException();
   } else {
     herItem = slot.get(stampHolder);
     slot.set(null, EMPTY);
     return herItem;
   }
} break;
```

Exchanger State EMPTY

```
case EMPTY: // slot is free
  if (slot.CAS(herItem, myItem, EMPTY, WAITING)) {
   while (System.nanoTime() < timeBound){
     herItem = slot.get(stampHolder);
     if (stampHolder[0] == BUSY) {
       slot.set(null, EMPTY);
       return herItem;
     }}
   if (slot.CAS(myItem, null, WAITING, EMPTY)){
      throw new TimeoutException();
   } else {
     herItem = slot.get(stampHolder);
     slot.set(null, EMPTY);
     return herItem;
   }
} break;
```

Try to insert *myItem* and change state to *WAITING*

Exchanger State EMPTY

```
case EMPTY: // slot is free
  if (slot.CAS(herItem, myItem, EMPTY, WAITING)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == BUSY) {
        slot.set(null, EMPTY);
        return herItem;
      }}
    if (slot.CAS(myItem, null, WAITING, EMPTY)){
       throw new TimeoutException();
    } else {
      herItem = slot.get(stampHolder);
      slot.set(null, EMPTY);
      return herItem;
    }
} break;
```

Spin until either
myItem is taken or timeout

Exchanger State EMPTY

```
case EMPTY: // slot is free
  if (slot.CAS(herItem, myItem, EMPTY, WAITING)) {
   while (System.nanoTime() < timeBound){
     herItem = slot.get(stampHolder);
     if (stampHolder[0] == BUSY) {
       slot.set(null, EMPTY);
       return herItem;
     }}
   if (slot.CAS(myItem, null, WAITING, EMPTY)){
      throw new TimeoutException();
   } else {
     herItem = slot.get(stampHolder);
     slot.set(null, EMPTY);
     return herItem;
   }
} break;
```

myItem was taken,
so return *herItem*
that was put in its place

Exchanger State EMPTY

Otherwise we ran out of time, try to reset status to EMPTY and time out

```
case EMPTY: // slot is free
  if (slot.compareAndSet(myItem, null, EMPTY, WAITING)) {
    while (System.nanoTime() < timeBound){
      herItem = slot.get(stampHolder);
      if (stampHolder[0] == BUSY) {
        slot.set(null, EMPTY);
        return herItem;
      }}
    if (slot.CAS(myItem, null, WAITING, EMPTY)){
      throw new TimeoutException();
    } else {
      herItem = slot.get(stampHolder);
      slot.set(null, EMPTY);
      return herItem;
    }
} break;
```

Exchanger State EMPTY

```
case EMPTY: // slot is free
     if (slot.compareAndSet(herItem, myItem, WAITING,
BUSY)) {
     while (System.nanoTime() < timeBound) {
       herItem = slot.get(stampHolder);
       if (stampHolder[0] == BUSY) {
         slot.set(null, EMPTY);
         return herItem;
       }}
     if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
      } else {
        herItem = slot.get(stampHolder);
        slot.set(null, EMPTY);
        return herItem;
     }
} break;
```

If reset failed,
someone showed up after all,
so take that item

Exchanger State EMPTY

```
case EMPTY: // slot is free
  if (slot.CAS(herItem, myItem, EMPTY, WAITING)) {
   while (System.nanoTime() < timeBound){
     herItem = slot.get(stampHolder);
     if (stampHolder[0] == BUSY) {
       slot.set(null, EMPTY);
       return herItem;
     }}
   if (slot.CAS(myItem, null, WAITING, EMPTY)){
      throw new TimeoutException();
   } else {
     herItem = slot.get(stampHolder);
     slot.set(null, EMPTY);
     return herItem;
   }
} break;
```

Clear slot and take that item

Exchanger State EMPTY

```
case EMPTY: // slot is free
  if (slot.CAS(herItem, myItem, EMPTY, WAITING)) {
   while (System.nanoTime() < timeBound){
     herItem = slot.get(stampHolder);
     if (stampHolder[0] == BUSY) {
       slo
       return he
     }
   if (slot.CAS(myItem, null, WAITING, EMPTY)){
     throw new TimeoutException();
   } else {
     herItem = slot.get(stampHolder);
     slot.set(null, EMPTY);
     return herItem;
   }
} break;
```

If initial CAS failed,
then someone else changed status
from EMPTY to WAITING,
so retry from start

States WAITING and BUSY

```
case WAITING:    // someone waiting for me
  if (slot.CAS(herItem, myItem, WAITING, BUSY))
    return herItem;
  break;
case BUSY:       // others in middle of exchanging
  break;
default:         // impossible
  break;
      }
    }
  }
}
```

States WAITING and BUSY

```
case WAITING:    // someone waiting for me
  if (slot.CAS(herItem, myItem, WAITING, BUSY))
    return herItem;
  break;
case BUSY:       // others in middle of exchanging
  break;
default:         // impossible
  break;
      }
    }
  }
}
```

someone is waiting to exchange,
so try to CAS my item in
and change state to BUSY

States WAITING and BUSY

```
case WAITING:    // someone waiting for me
  if (slot.CAS(herItem, myItem, WAITING, BUSY))
    return herItem;
  break;
case BUSY:       // others in middle of exchanging
  break;
default:         // impossible
  break;
      }
    }
  }
}
```

If successful, return other's item,
otherwise someone else took it,
so try again from start

States WAITING and BUSY

```
case WAITING:    // someone waiting for me
  if (slot.CAS(herItem, myItem, WAITING, BUSY))
    return herItem;
  break;
case BUSY:       // others in middle of exchanging
  break;
default:         // impossible
  break;
      }
    }
  }
}
```

If BUSY,
other threads exchanging,
so start again

The Exchanger Slot

- Exchanger is lock-free
- Because the only way an exchange can fail is if others repeatedly succeeded or no-one showed up
- The slot we need does not require symmetric exchange

Back to the Stack: the Elimination Array

```
public class EliminationArray {
…
public T visit(T value, int range)
  throws TimeoutException {
    int slot = random.nextInt(range);
    int nanodur = convertToNanos(duration, timeUnit));
    return (exchanger[slot].exchange(value, nanodur)
}}
```

Elimination Array

```
public class EliminationArray {

public T visit(T value, int range)
    throws TimeoutException {
      int slot = random.nextInt(range);
      int nanodur = convertToNanos(duration, timeUnit));
      return (exchanger[slot].exchange(value, nanodur);
}}
```

visit the elimination array with fixed value and range

Elimination Array

```
public class EliminationArray {
…
public T visit(T value, int range)
  throws TimeoutException {
    int slot = random.nextInt(range);
    int nanodur = convertToNanos(duration, timeUnit));
    return (exchanger[slot].exchange(value, nanodur)
}}
```

Pick a random array entry

Elimination Array

```
public class EliminationArray {
…
public T visit(T value, int range)
   throws TimeoutException {
     int slot = random.nextInt(range);
     int nanodur = convertToNanos(duration, timeUnit));
     return (exchanger[slot].exchange(value, nanodur)
}}
```

Exchange value or time out

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.range);
      if (otherValue == null) {
         return;
    }
}
```

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.range);
      if (otherValue == null) {
         return;
    }
}
```

First, try to push

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.range);
      if (otherValue == null) {
          return;
      }
}
```

If I failed, backoff & try to eliminate

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.range);
      if (otherValue == null) {
         return;
    }
}
```

Value pushed and range to try

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.range);
      if (otherValue == null) {
        return;
      }
    }
}
```

Only `pop()` leaves null, so elimination was successful

Elimination Stack Push

```
public void push(T value) {
...
 while (true) {
  if (tryPush(node)) {
    return;
  } else try {
      T otherValue =
eliminationArray.visit(value,policy.range);
      if (otherValue == null) {
        return;
      }
}
```

Otherwise, retry **`push()`** on lock-free stack

Elimination Stack Pop

```
public T pop() {
  ...
 while (true) {
  if (tryPop()) {
   return returnNode.value;
   } else
      try {
        T otherValue =
        eliminationArray.visit(null,policy.range;
        if (otherValue != null) {
         return otherValue;
         }
      }
}}
```

Elimination Stack Pop

```
public T pop() {
  ...
 while (true) {
   ...
   return returnNode.value;
  } else
      try {
        T otherValue =
        eliminationArray.visit(null,policy.range;
        if ( otherValue != null) {
         return otherValue;
        }
      }
}}
```

If value not null, other thread is a **push()**, so elimination succeeded

Summary

- We saw both lock-based and lock-free implementations of
- queues and stacks
- Don't be quick to declare a data structure inherently sequential
  - Linearizable stack is not inherently sequential (though it is in the worst case)
- ABA is a real problem, pay attention