

Concurrent Queues and Stacks

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

The Five-Fold Path

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

Another Fundamental Problem

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

Queues & Stacks

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

Bounded vs Unbounded

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

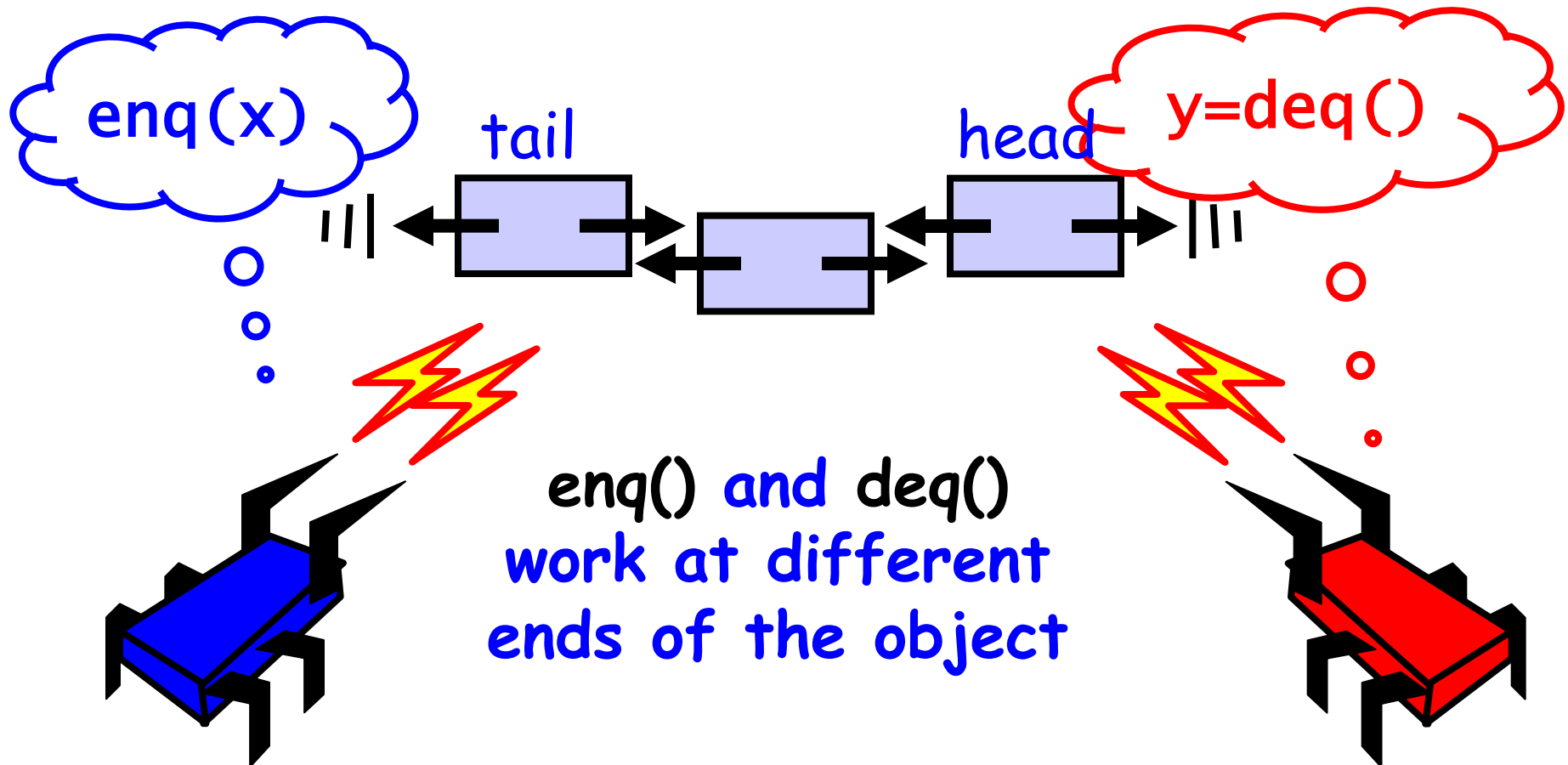
Blocking vs Non-Blocking

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

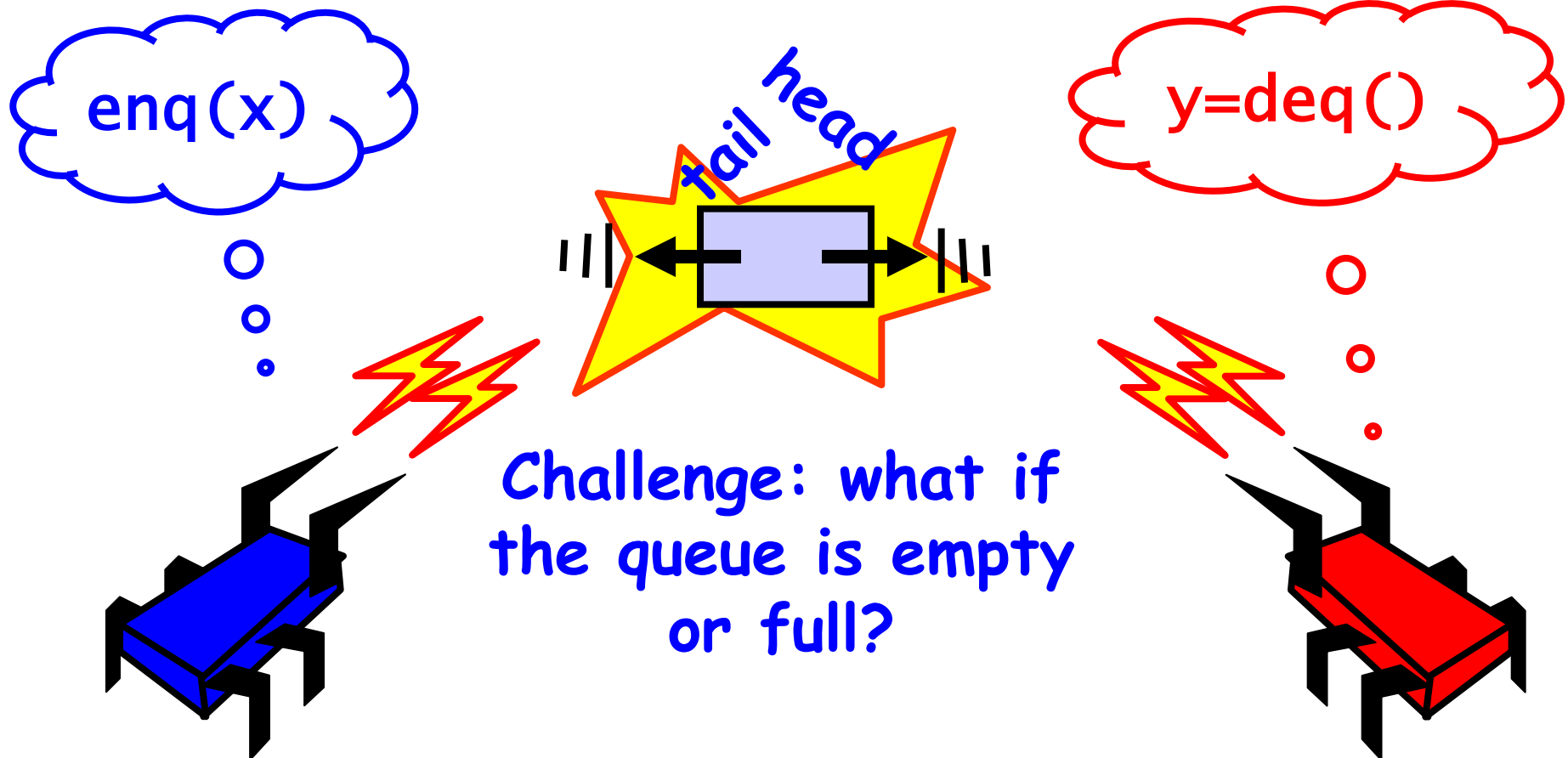
This Lecture

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

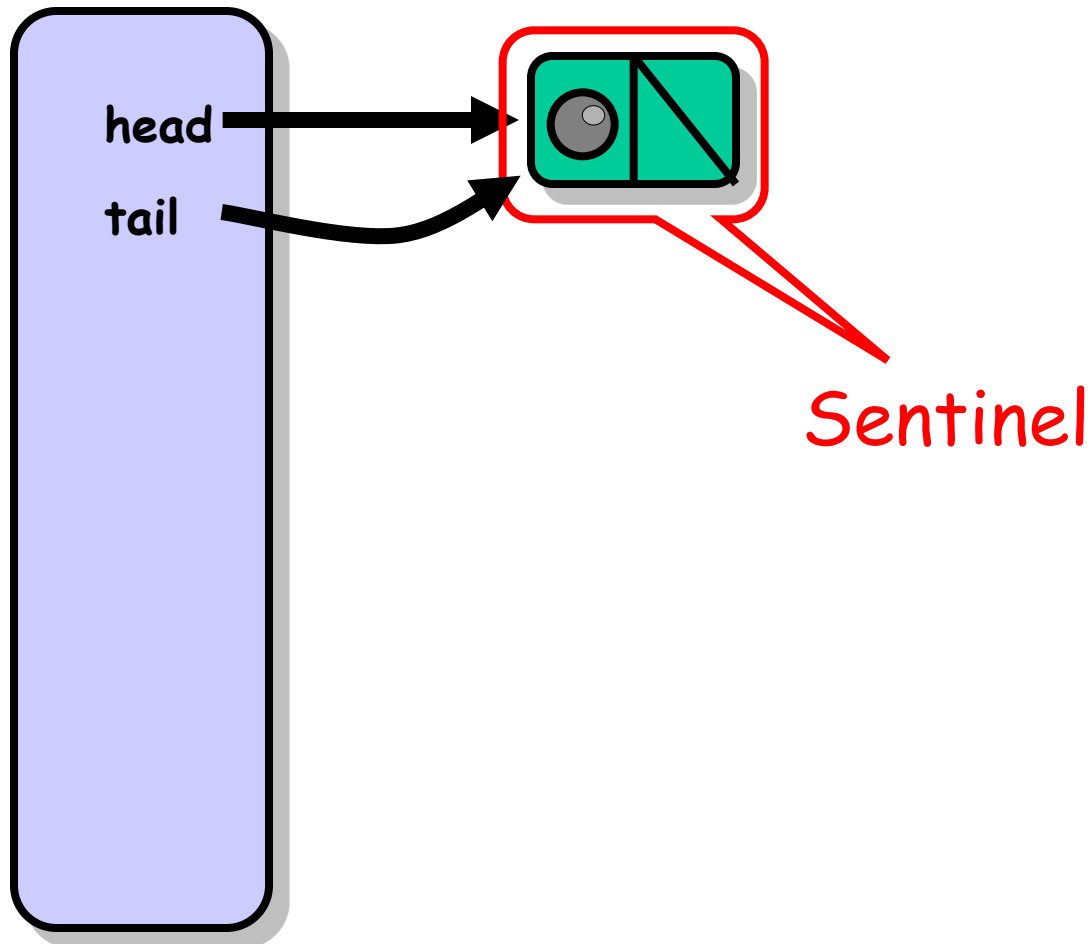
Queue: Concurrency



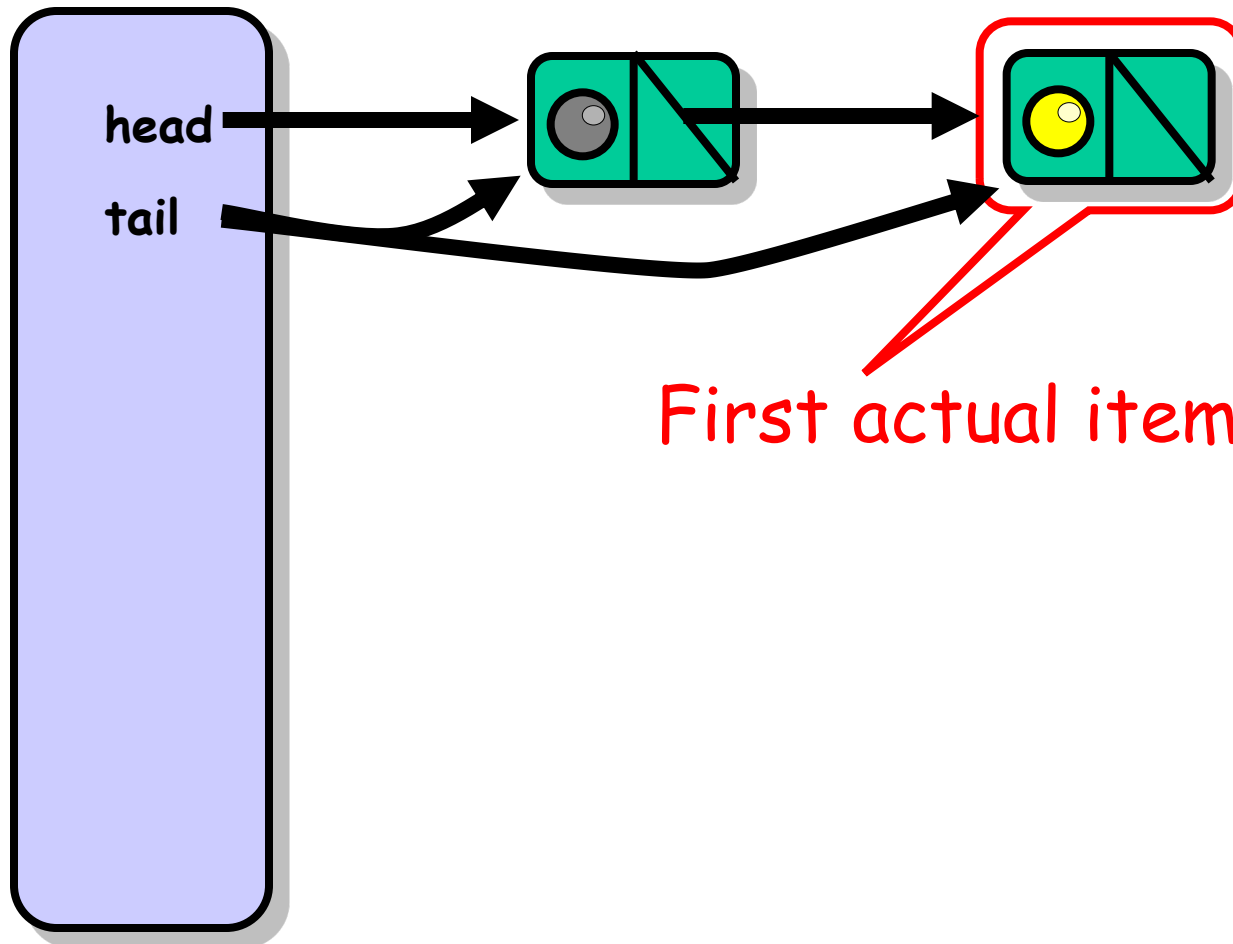
Concurrency



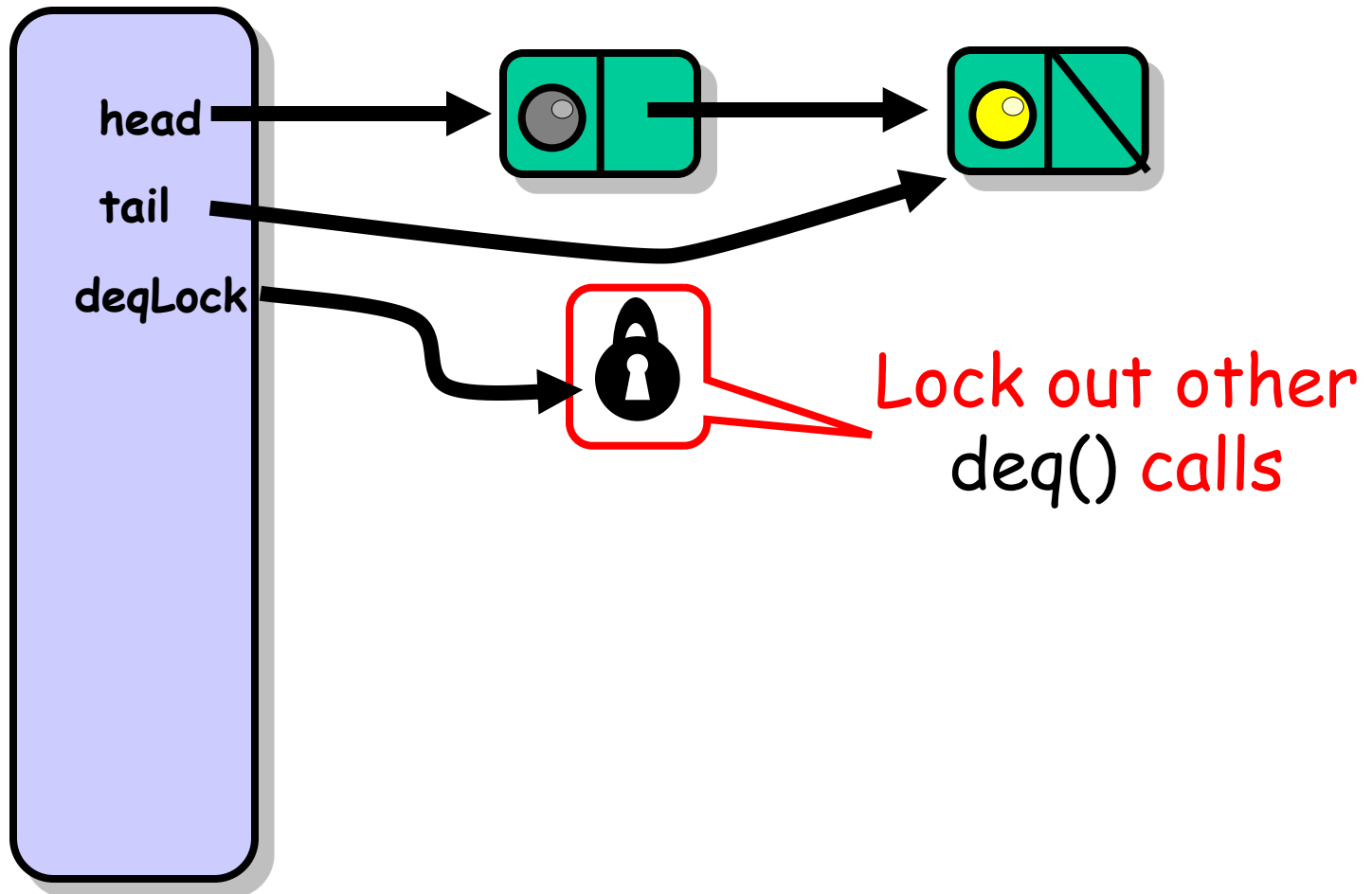
Bounded Queue



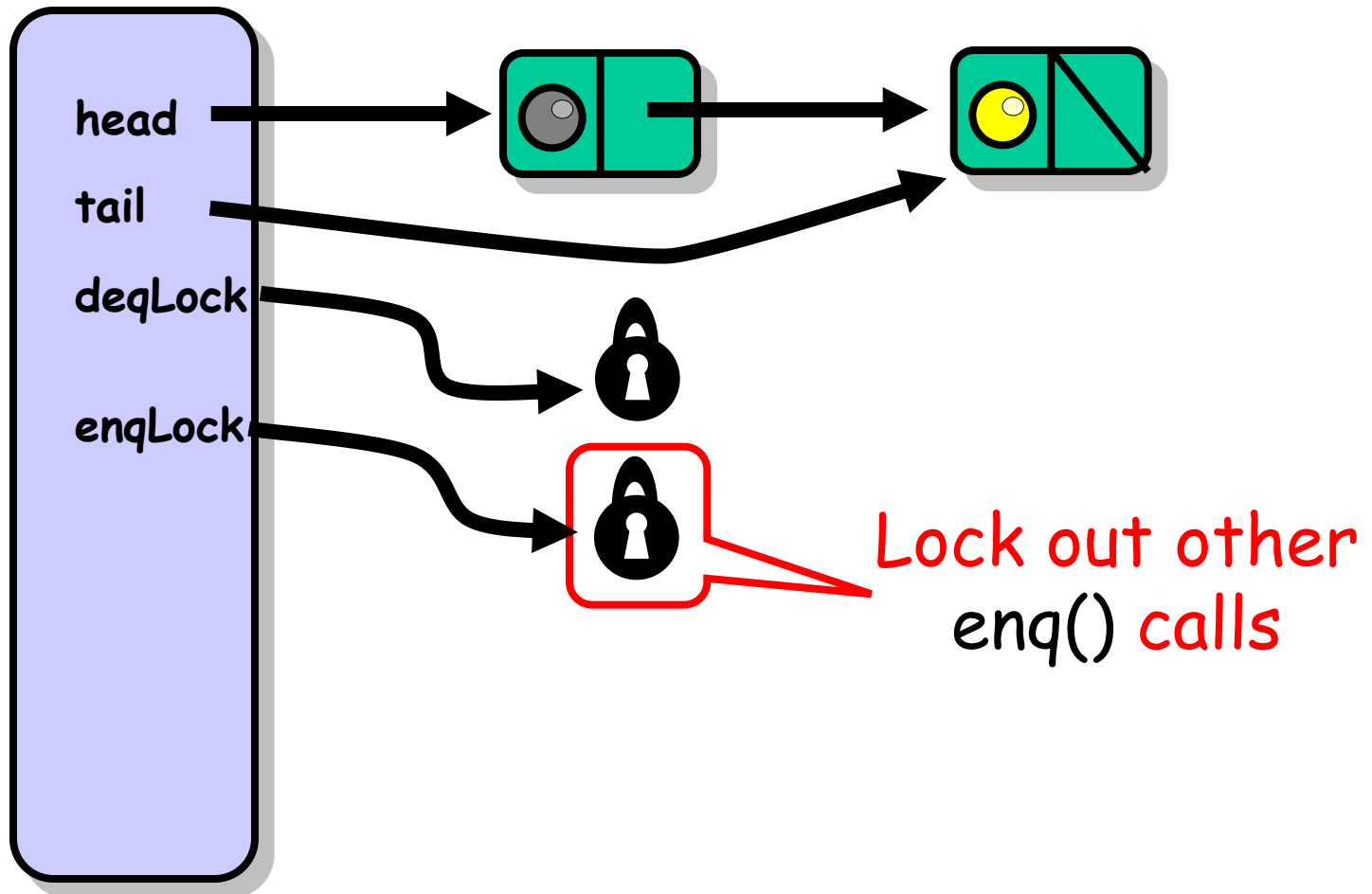
Bounded Queue



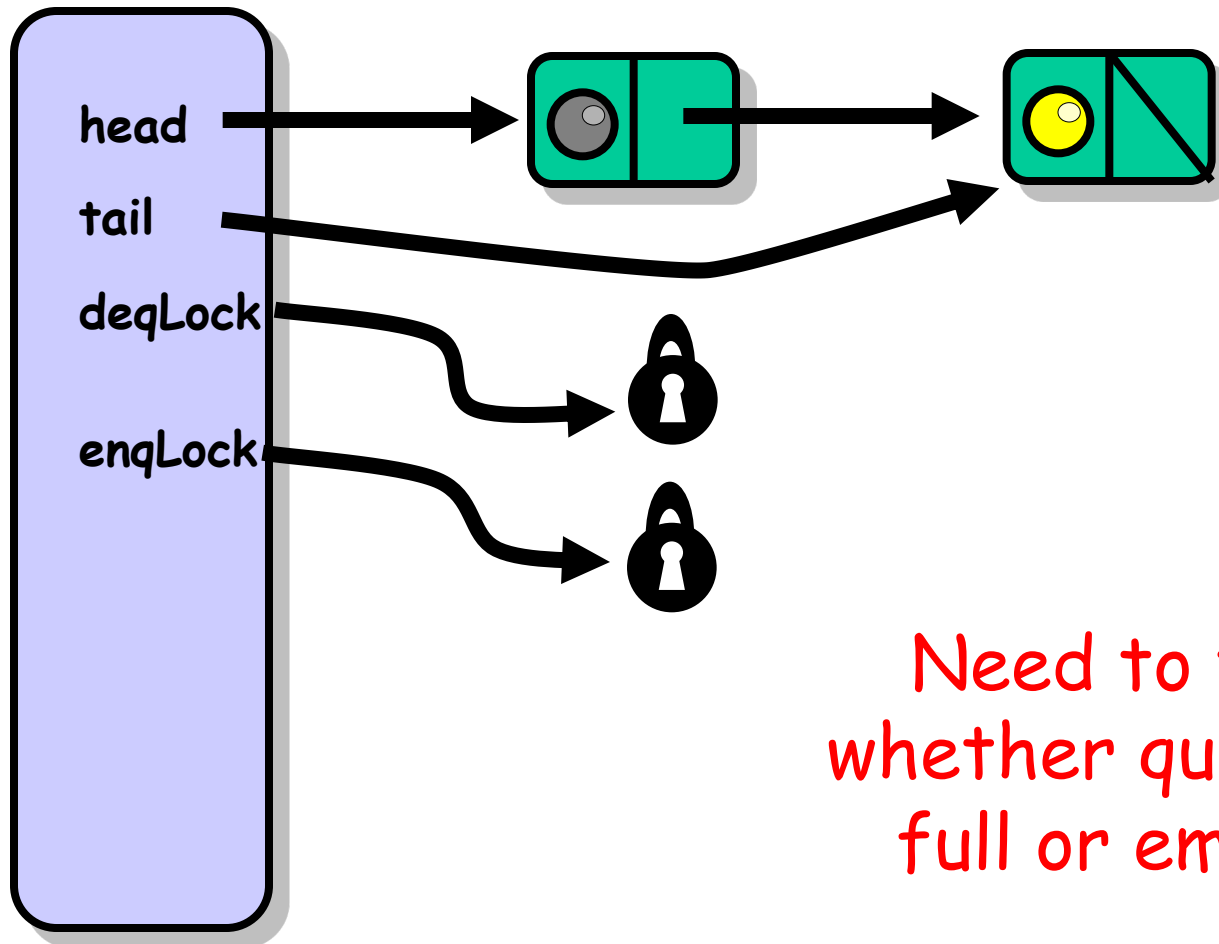
Bounded Queue



Bounded Queue

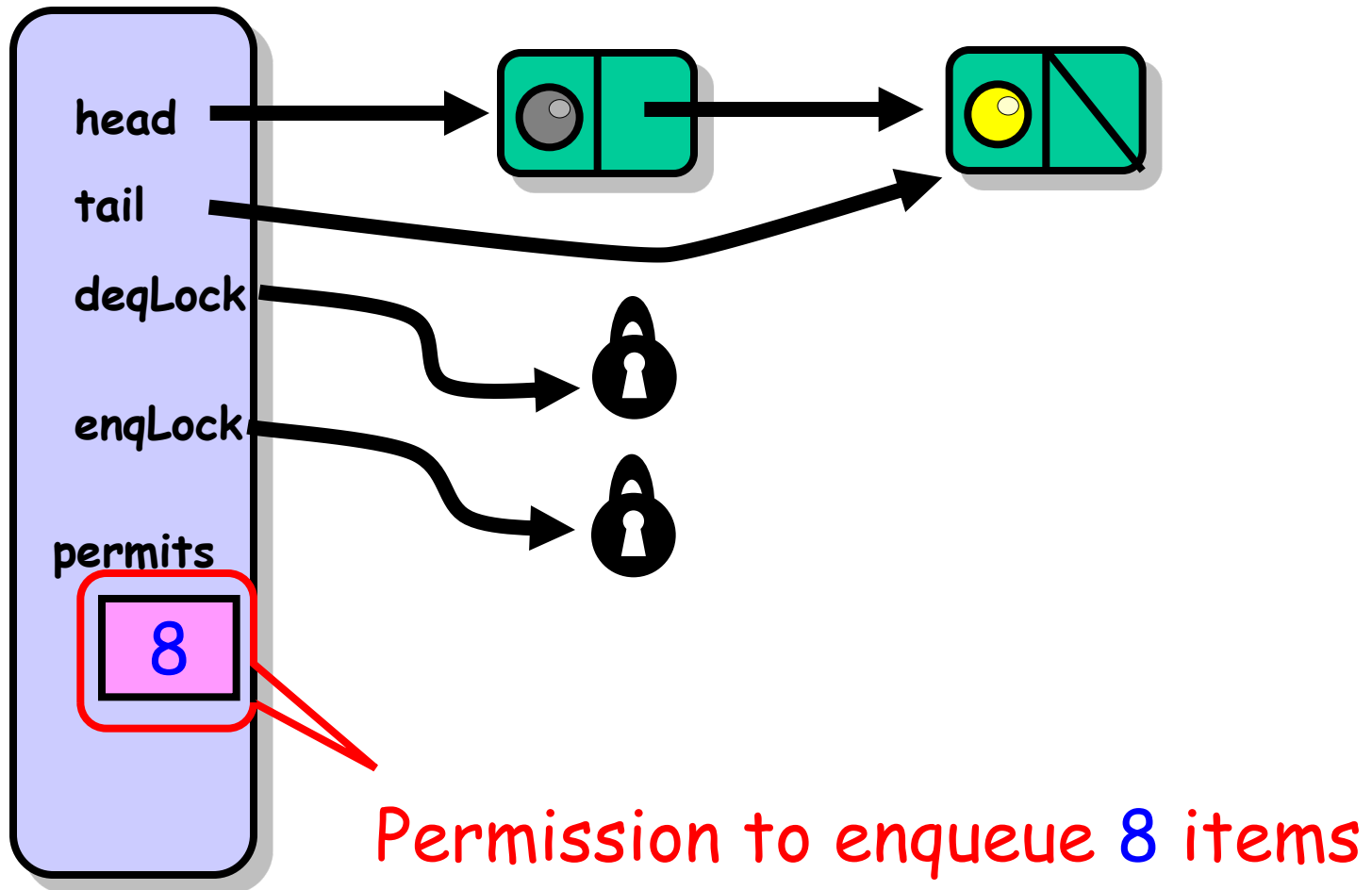


Not Done Yet

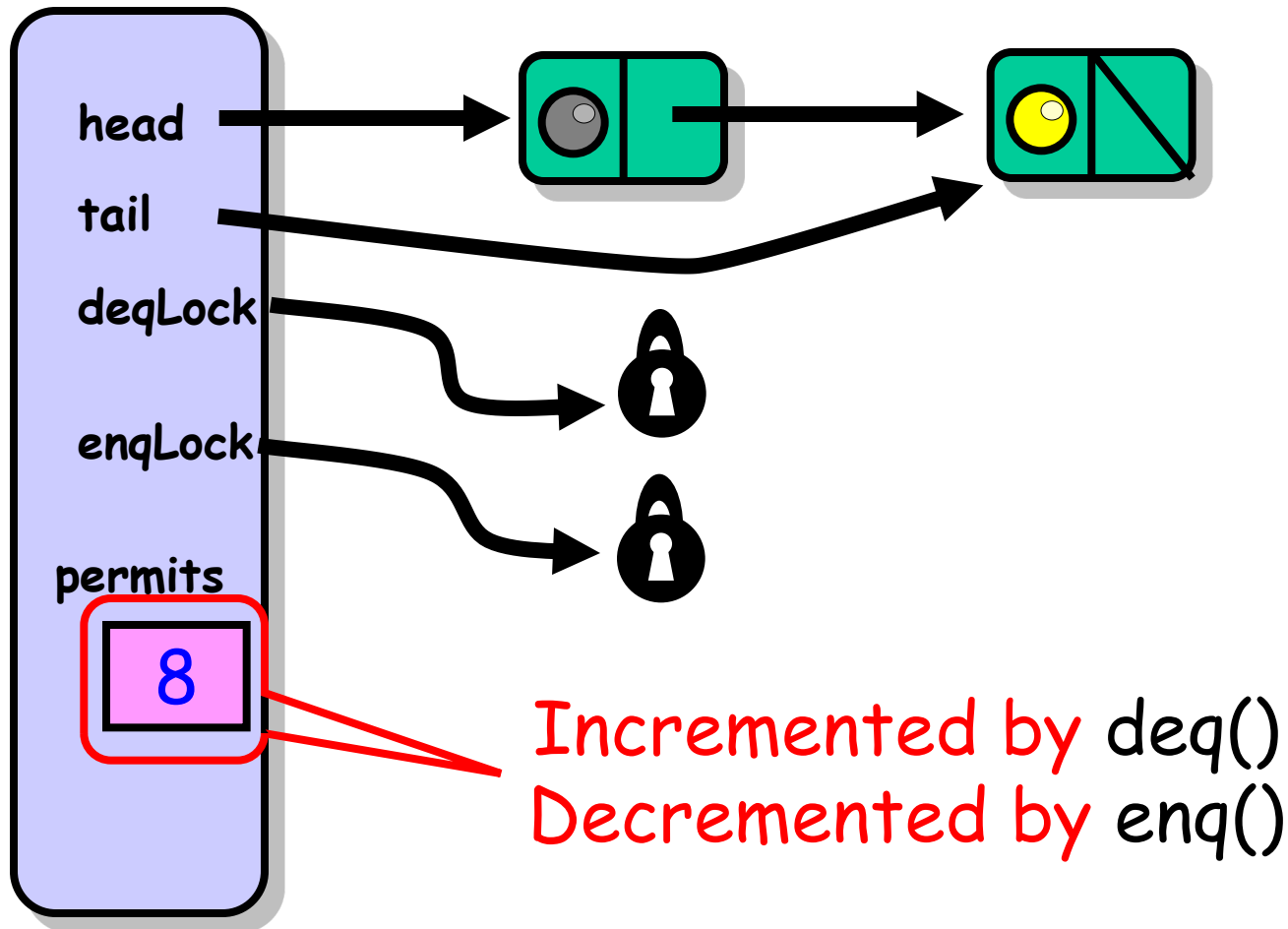


Need to tell
whether queue is
full or empty

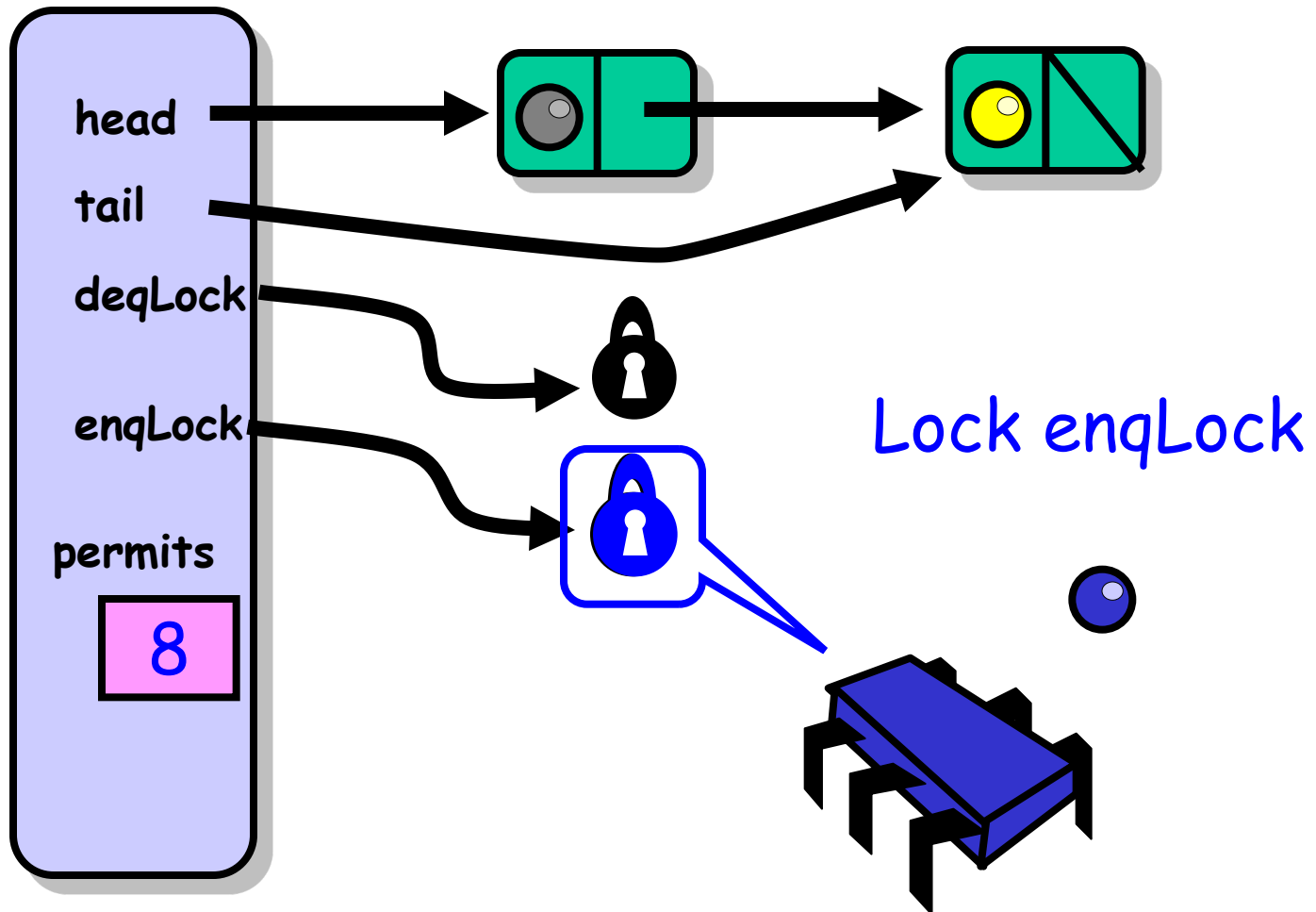
Not Done Yet



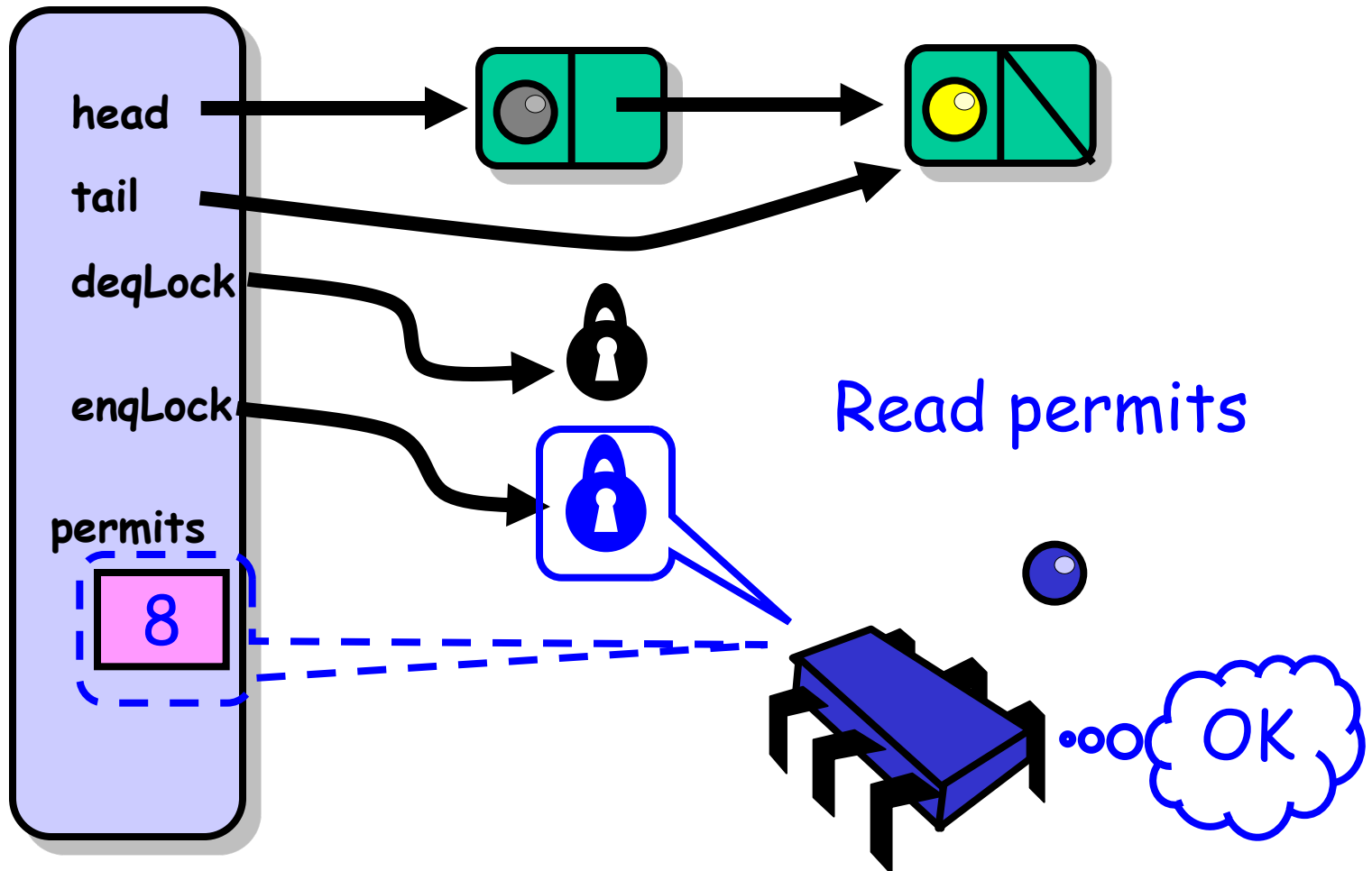
Not Done Yet



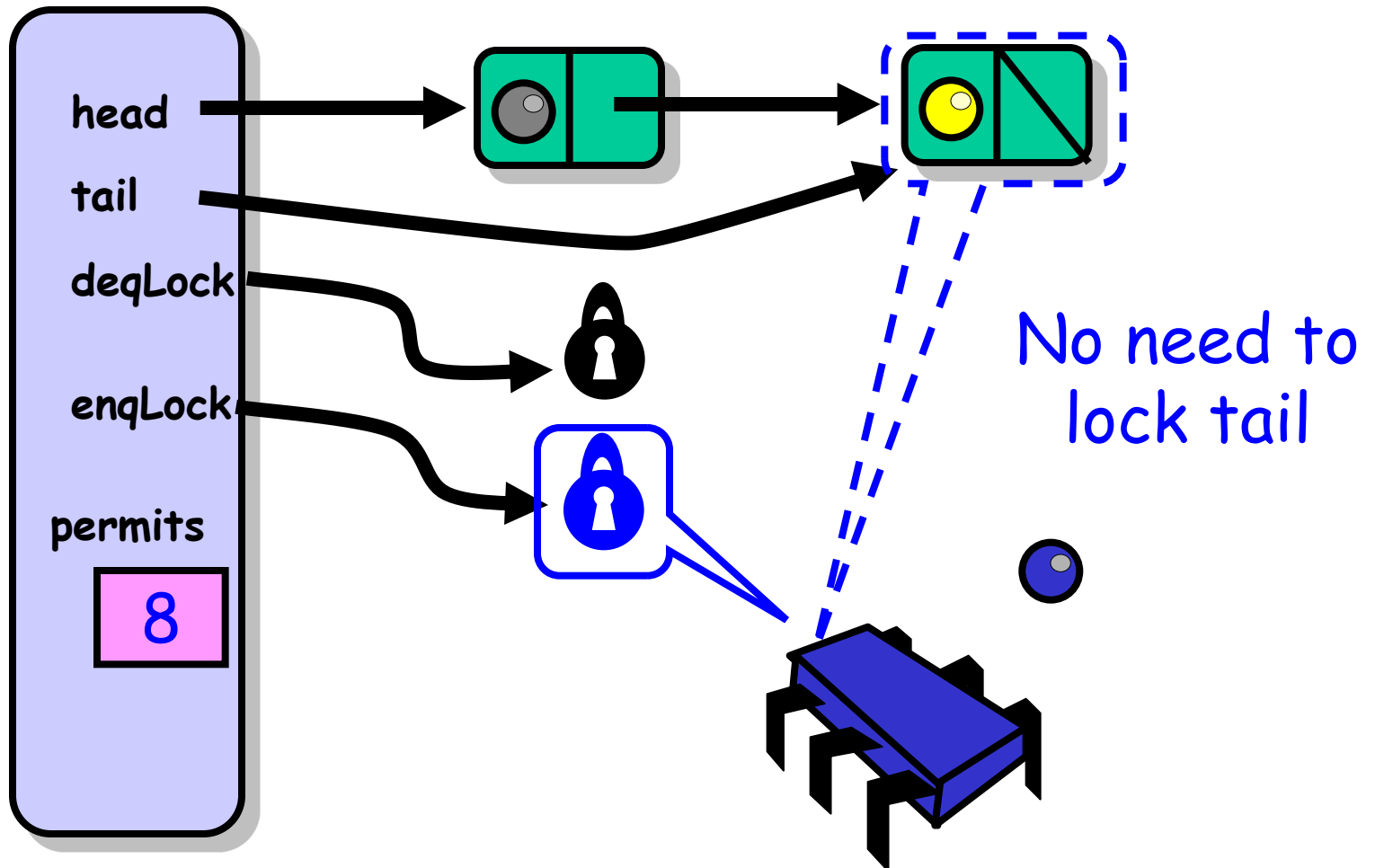
Enqueuer



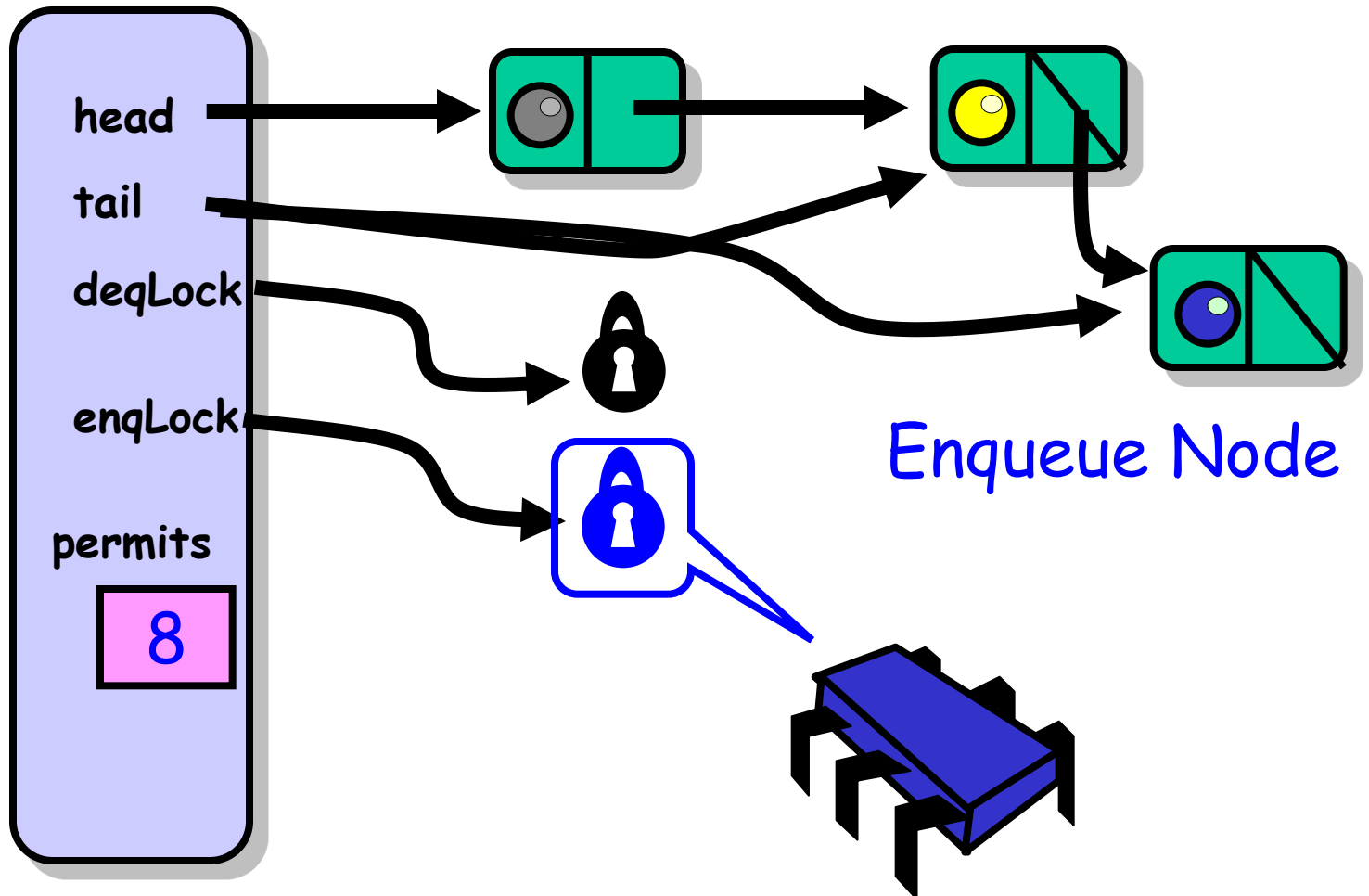
Enqueuer



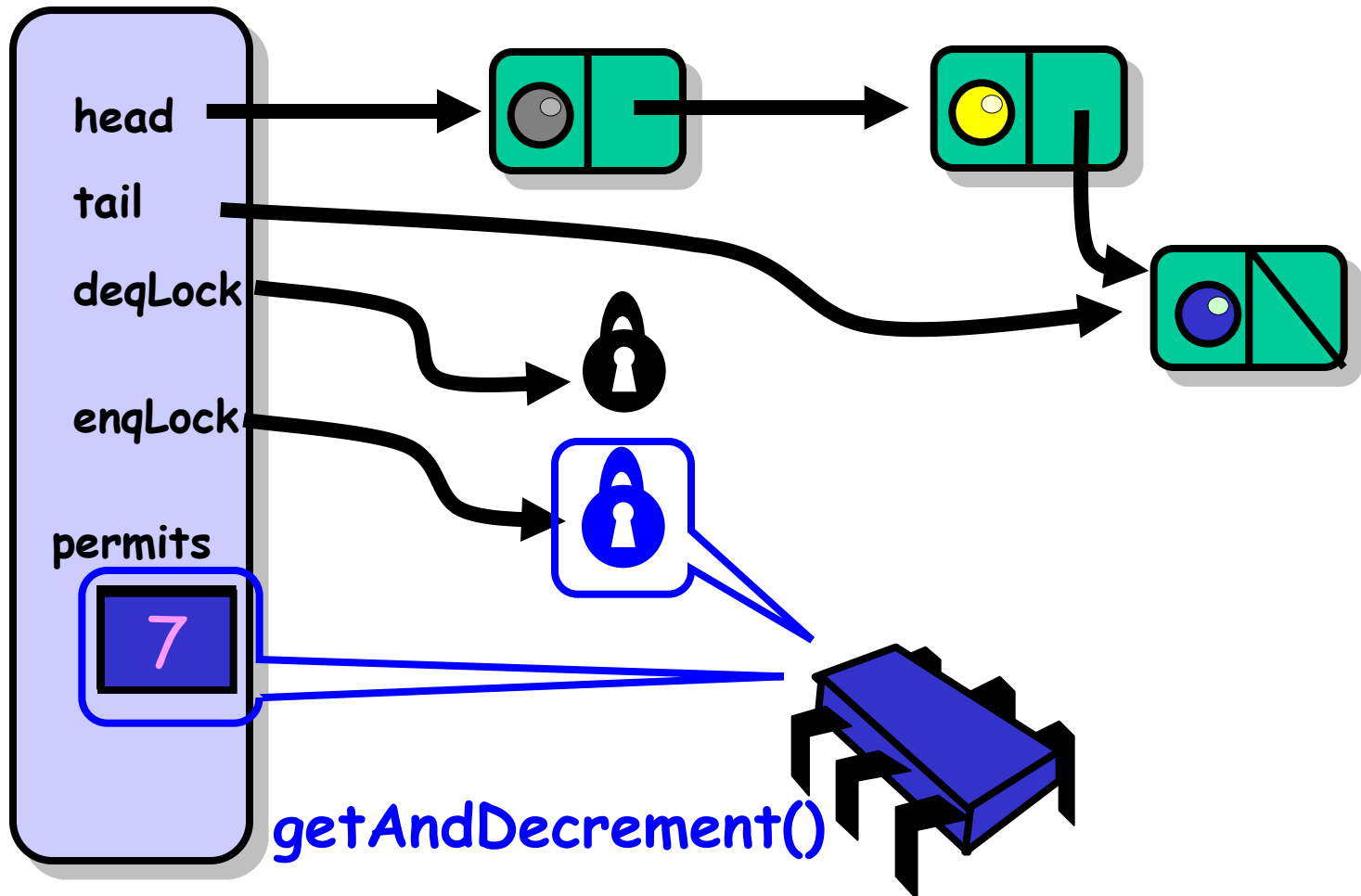
Enqueuer



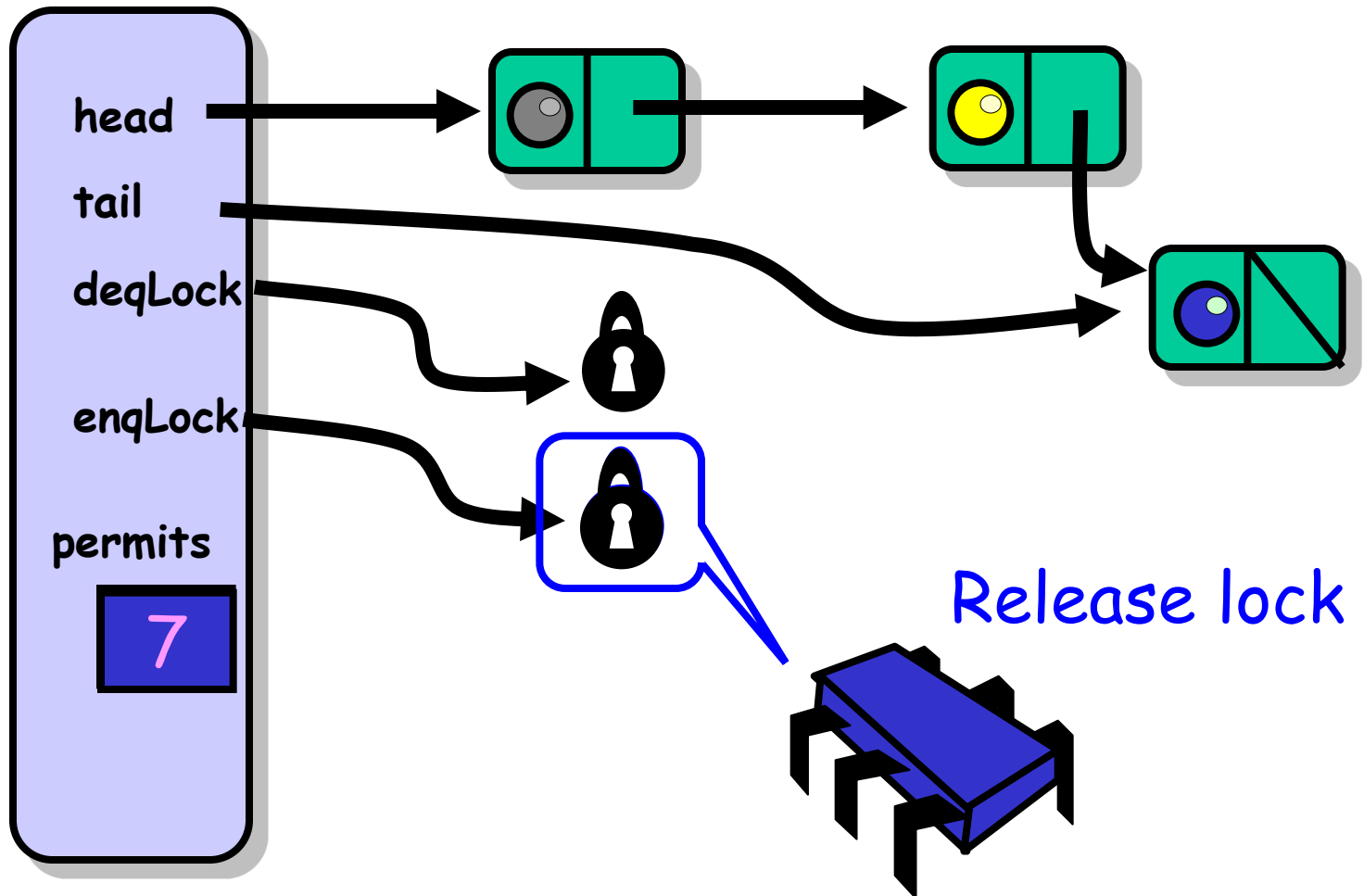
Enqueuer



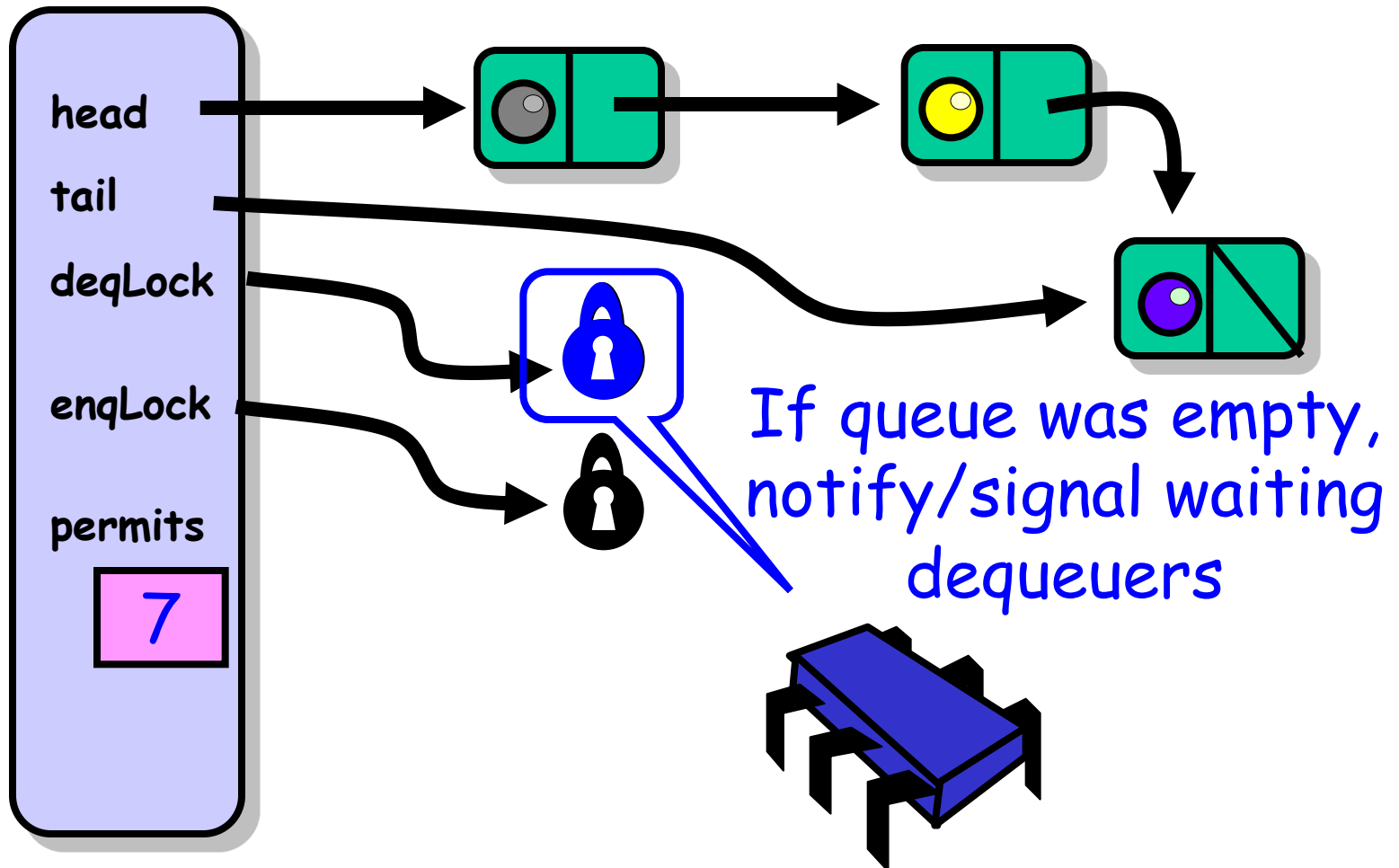
Enqueuer



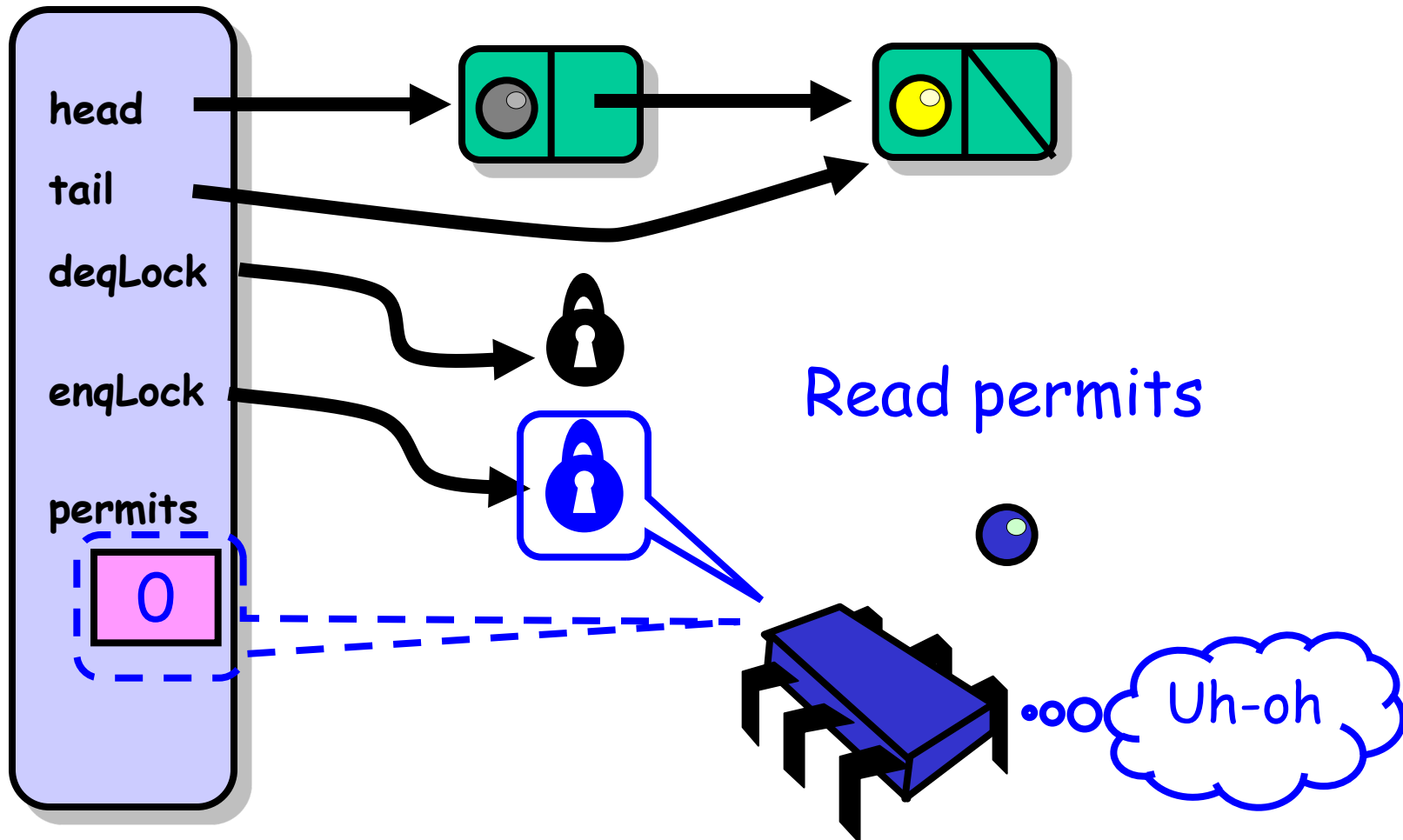
Enqueuer



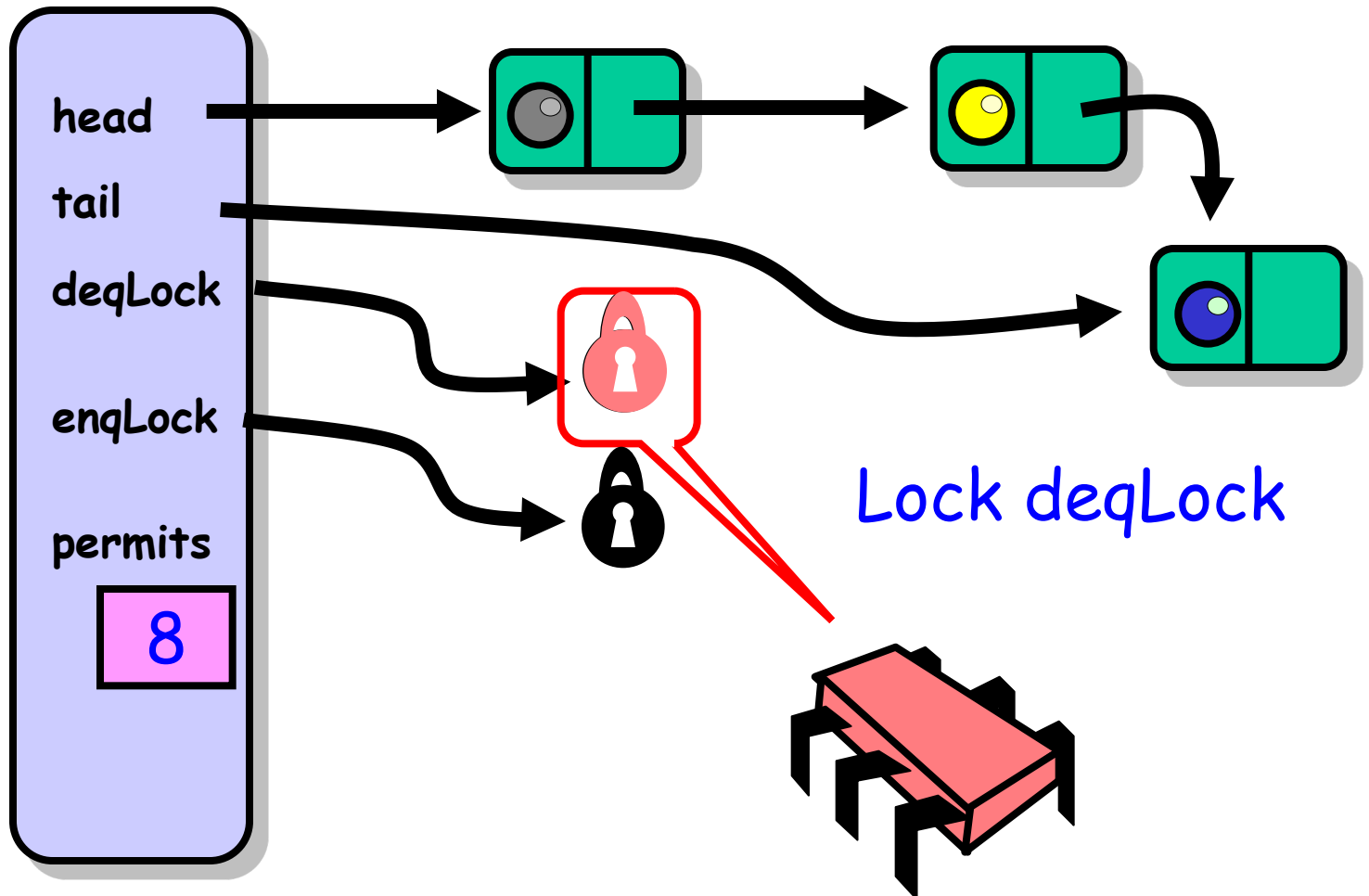
Enqueuer



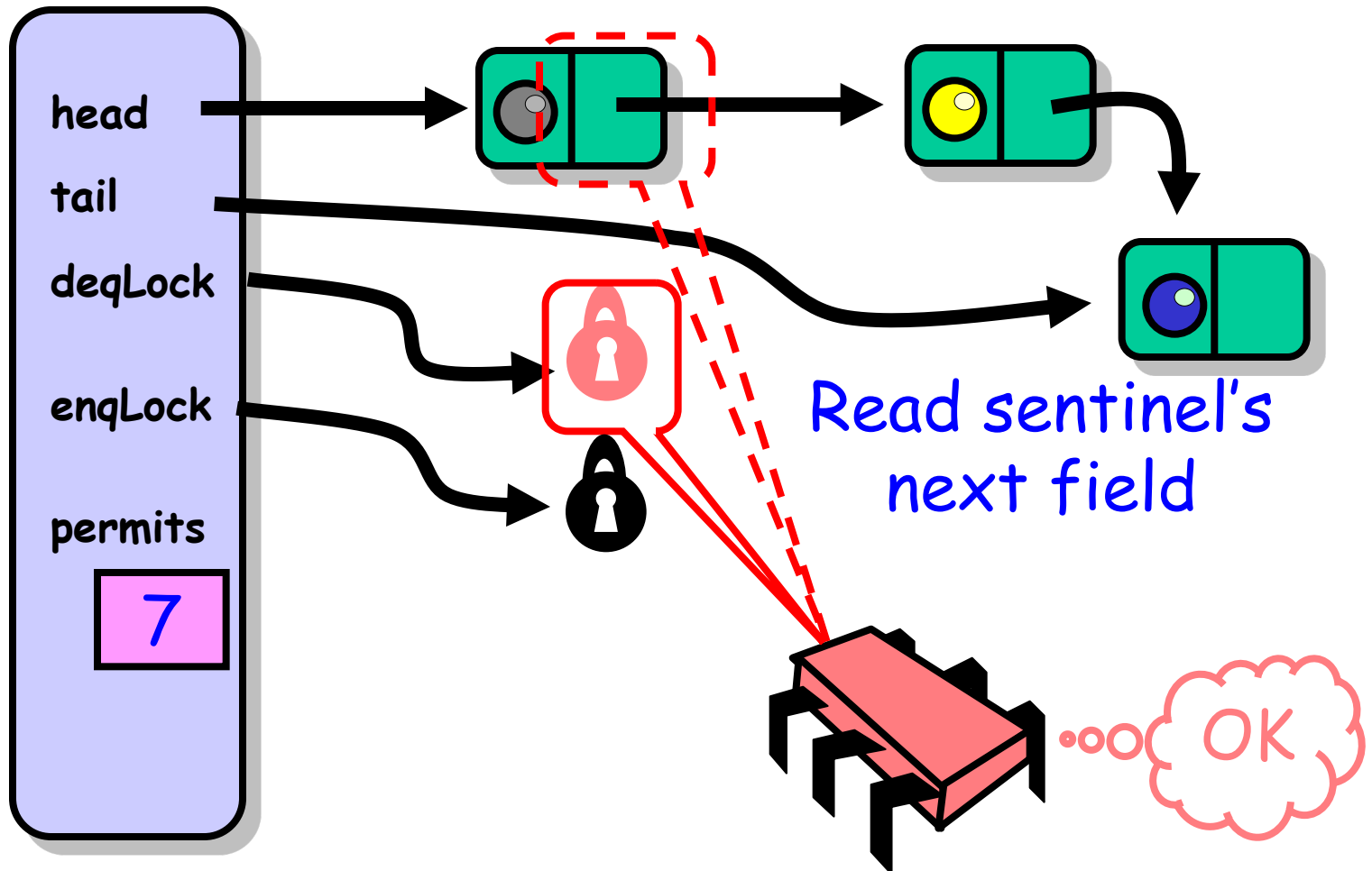
Unsuccessful Enqueuer



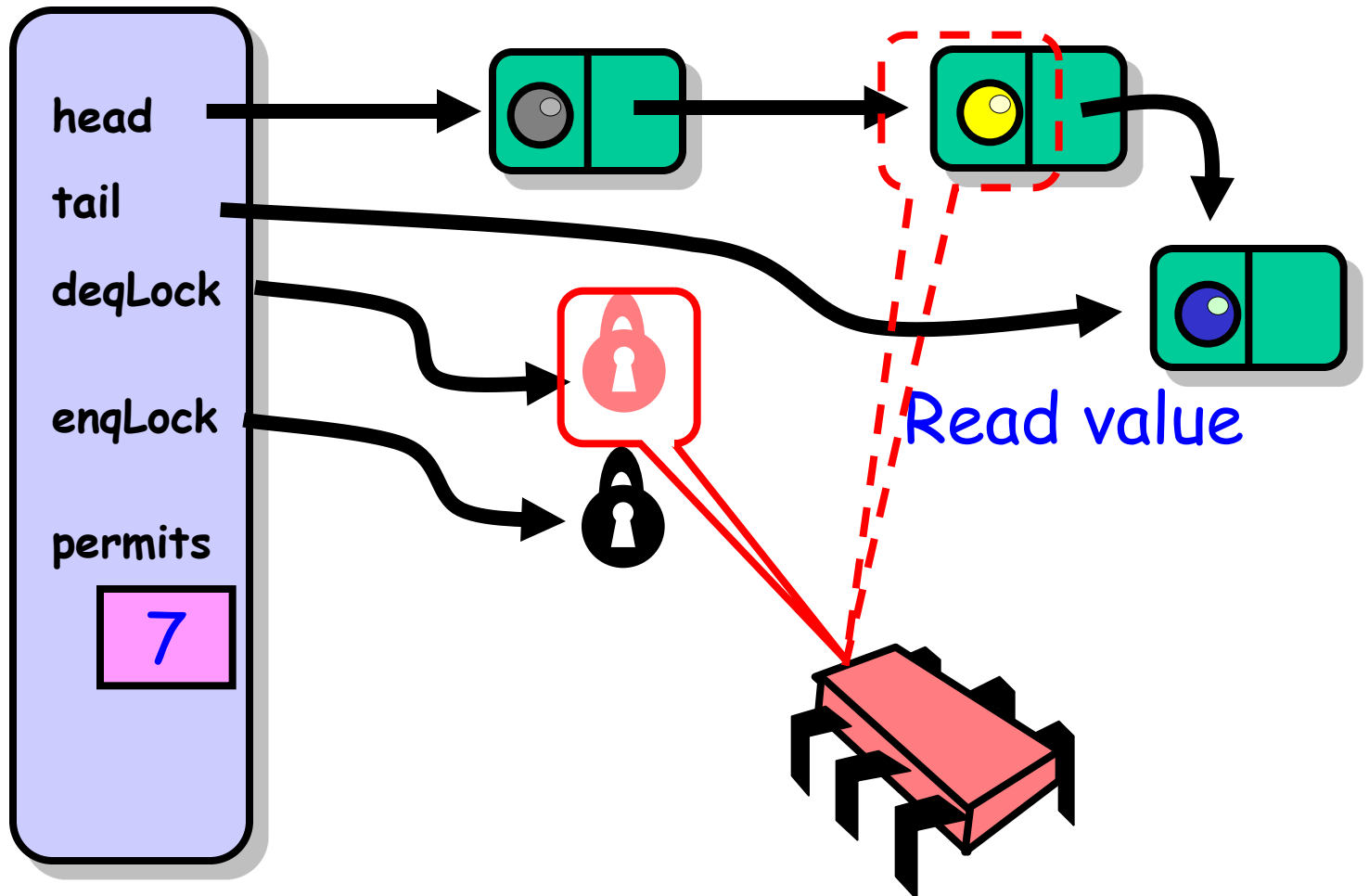
Dequeuer



Dequeuer

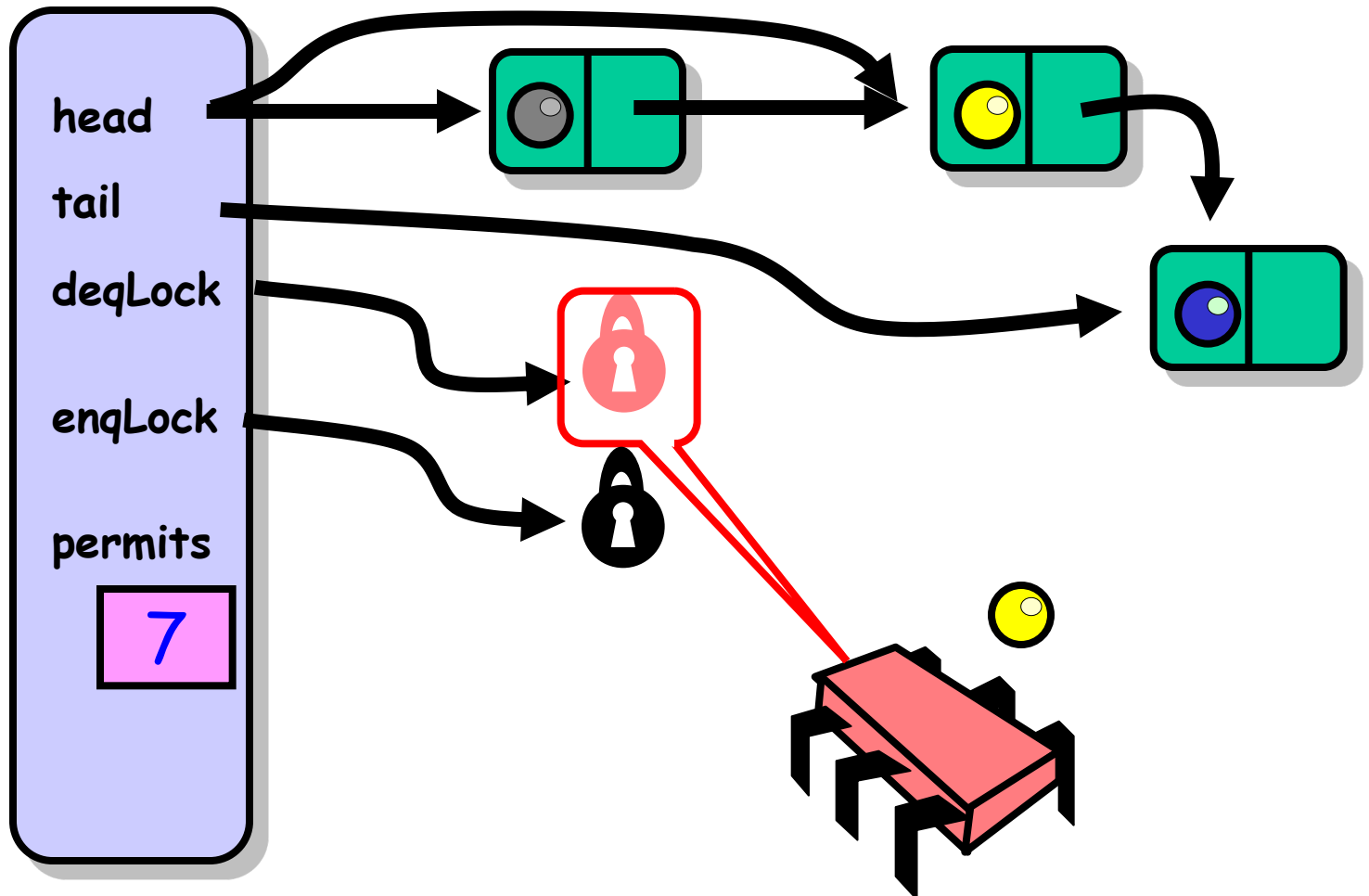


Dequeuer

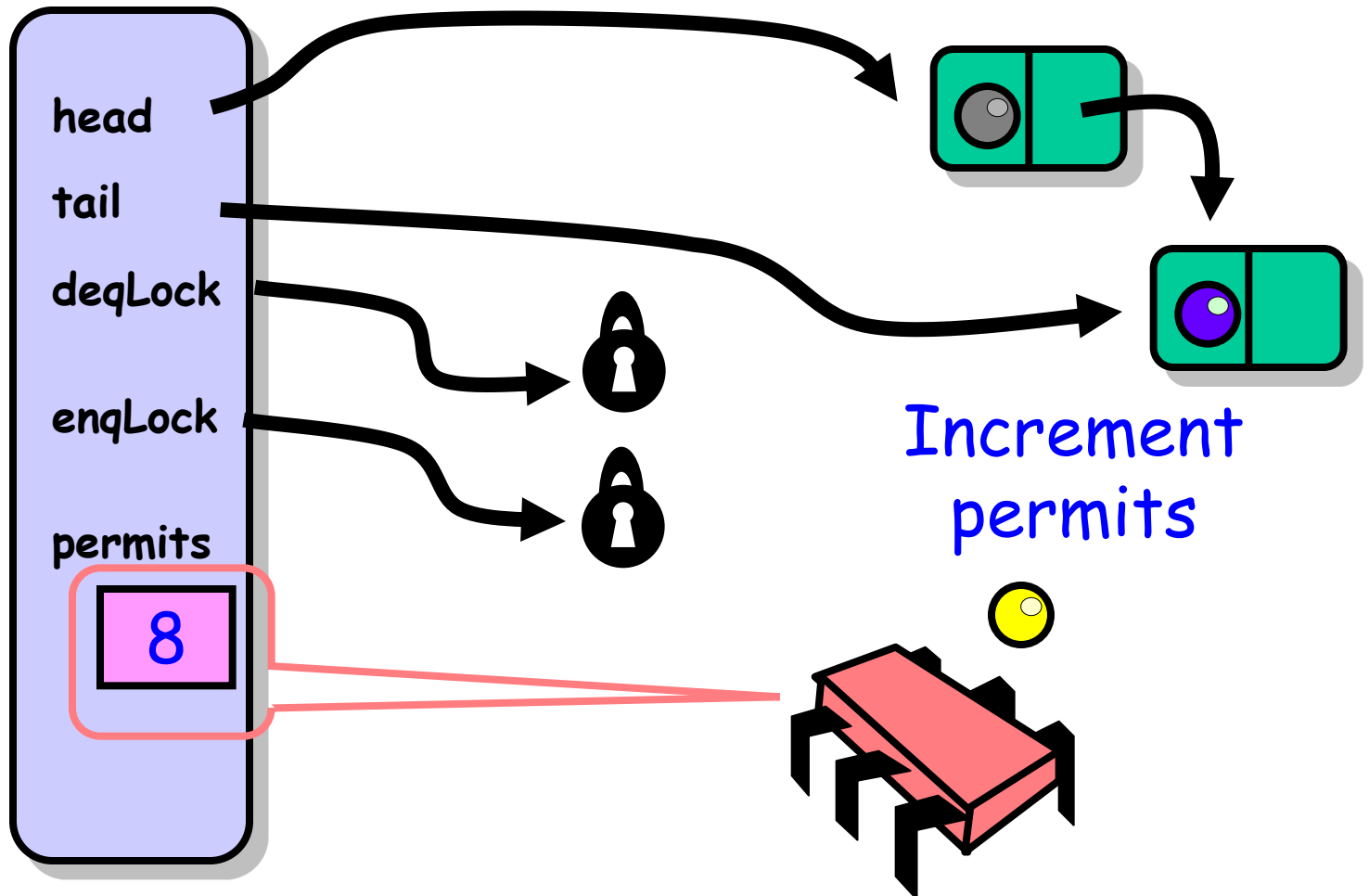


Make first Node
new sentinel

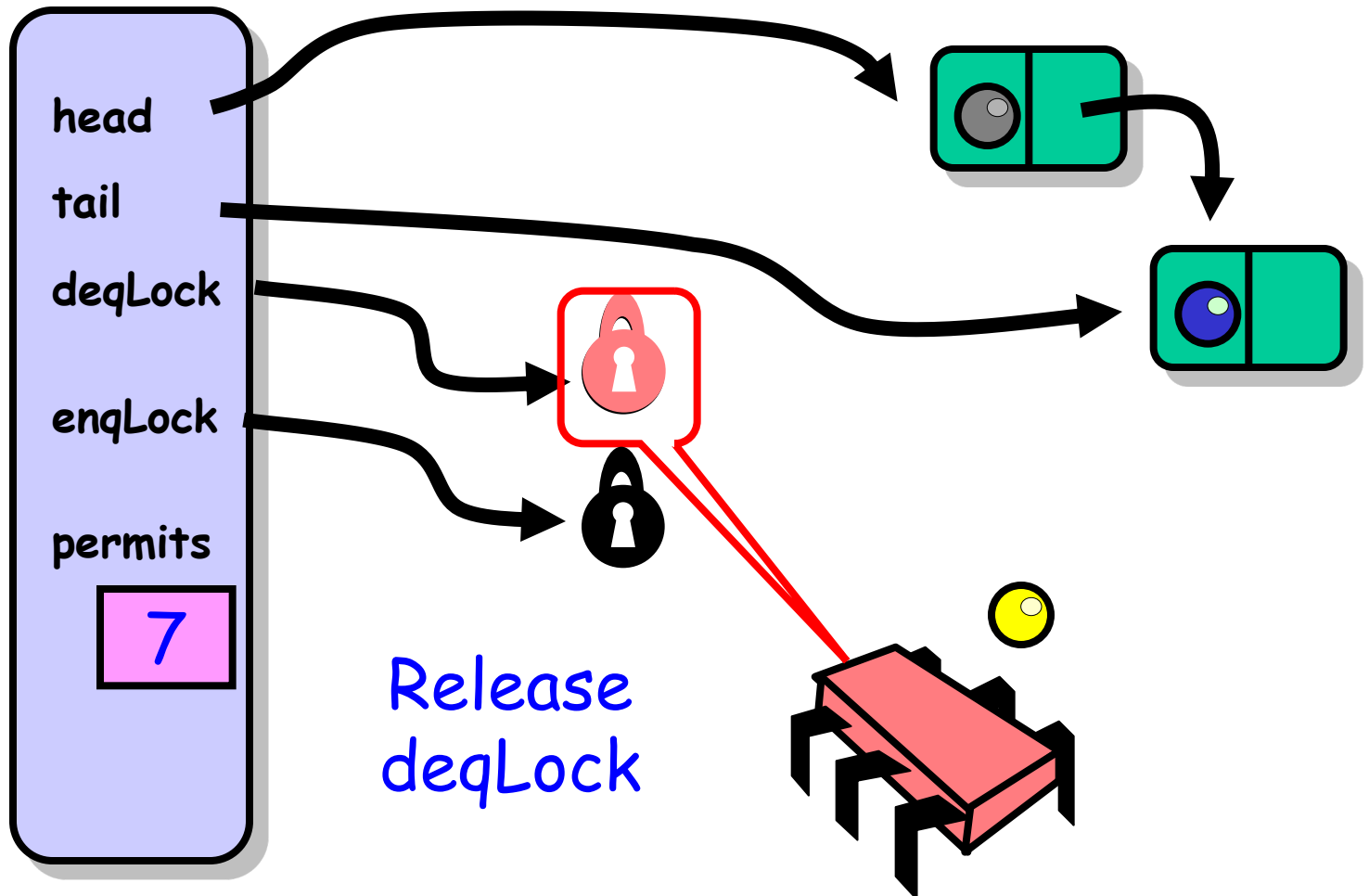
Dequeuer



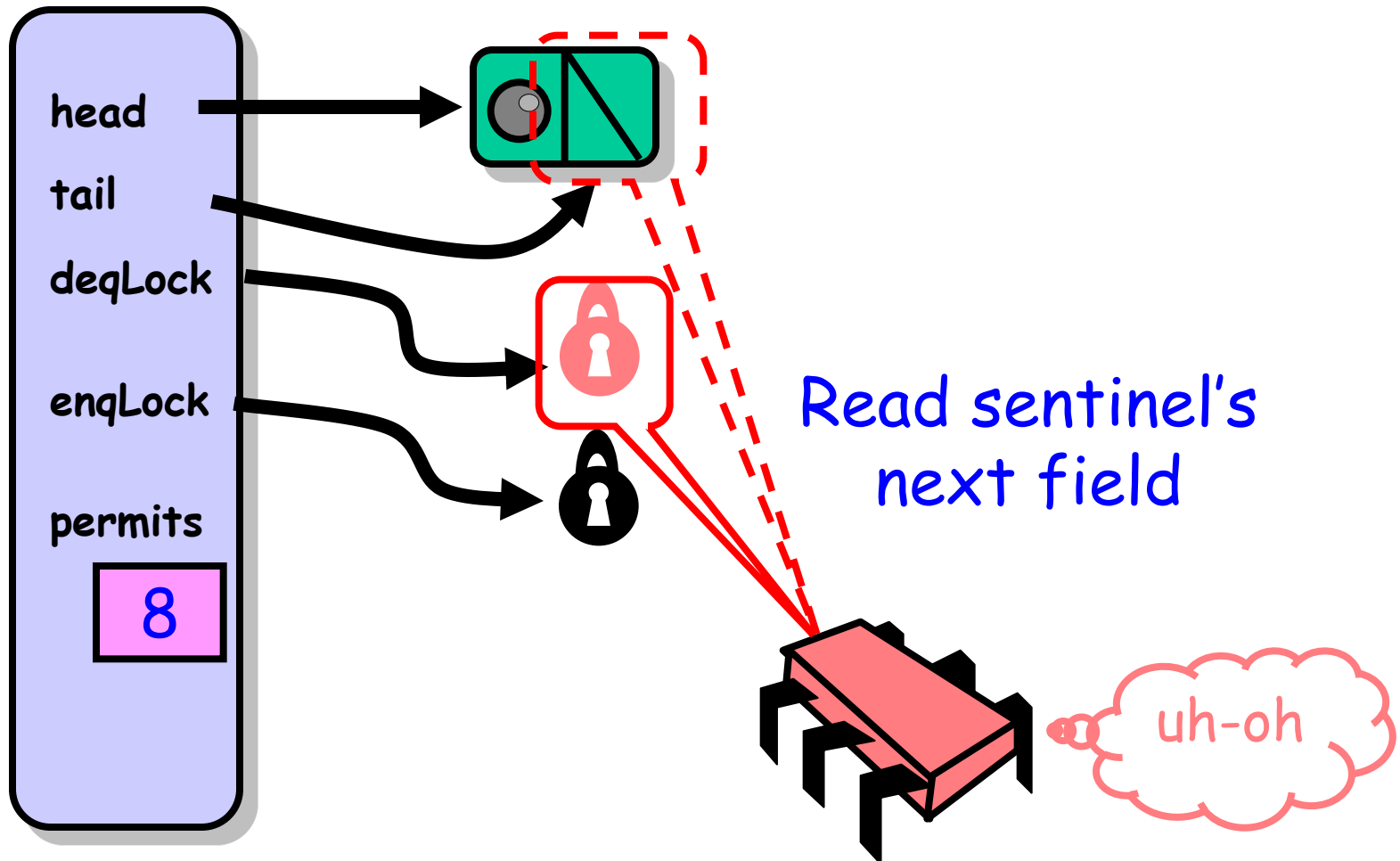
Dequeuer



Dequeuer



Unsuccessful Dequeueer

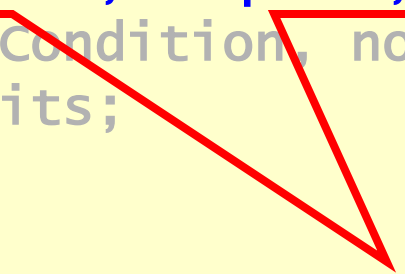


Bounded Queue

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

Bounded Queue

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



Enq & deq locks

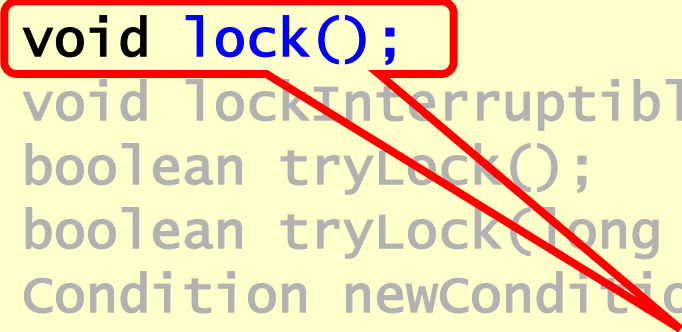
Digression: Monitor Locks

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

The Java Lock Interface

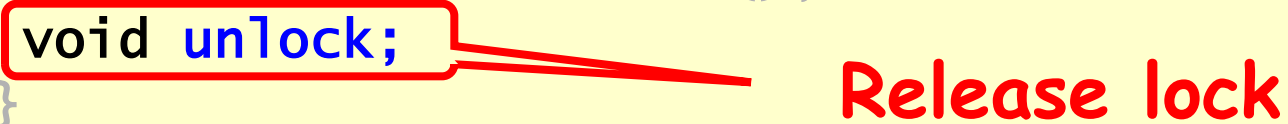
```
public interface Lock {  
    void lock();  
    void lockInterruptibly() throws InterruptedException;  
    boolean tryLock();  
    boolean tryLock(long time, TimeUnit unit);  
    Condition newCondition();  
    void unlock;  
}
```

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();  
}
```

Guess what this method does?

Lock Conditions

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

Lock Conditions

```
public interface Condition {
```

```
    void await();
```

```
    boolean await(long time, TimeUnit unit);
```

```
    ...
```

```
    void signal();
```

```
    void signalAll();
```

```
}
```

**Release lock and
wait on condition**

Lock Conditions

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

Wake up one waiting thread

Lock Conditions

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

Wake up all waiting threads

Await

`q.await()`

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

Signal

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

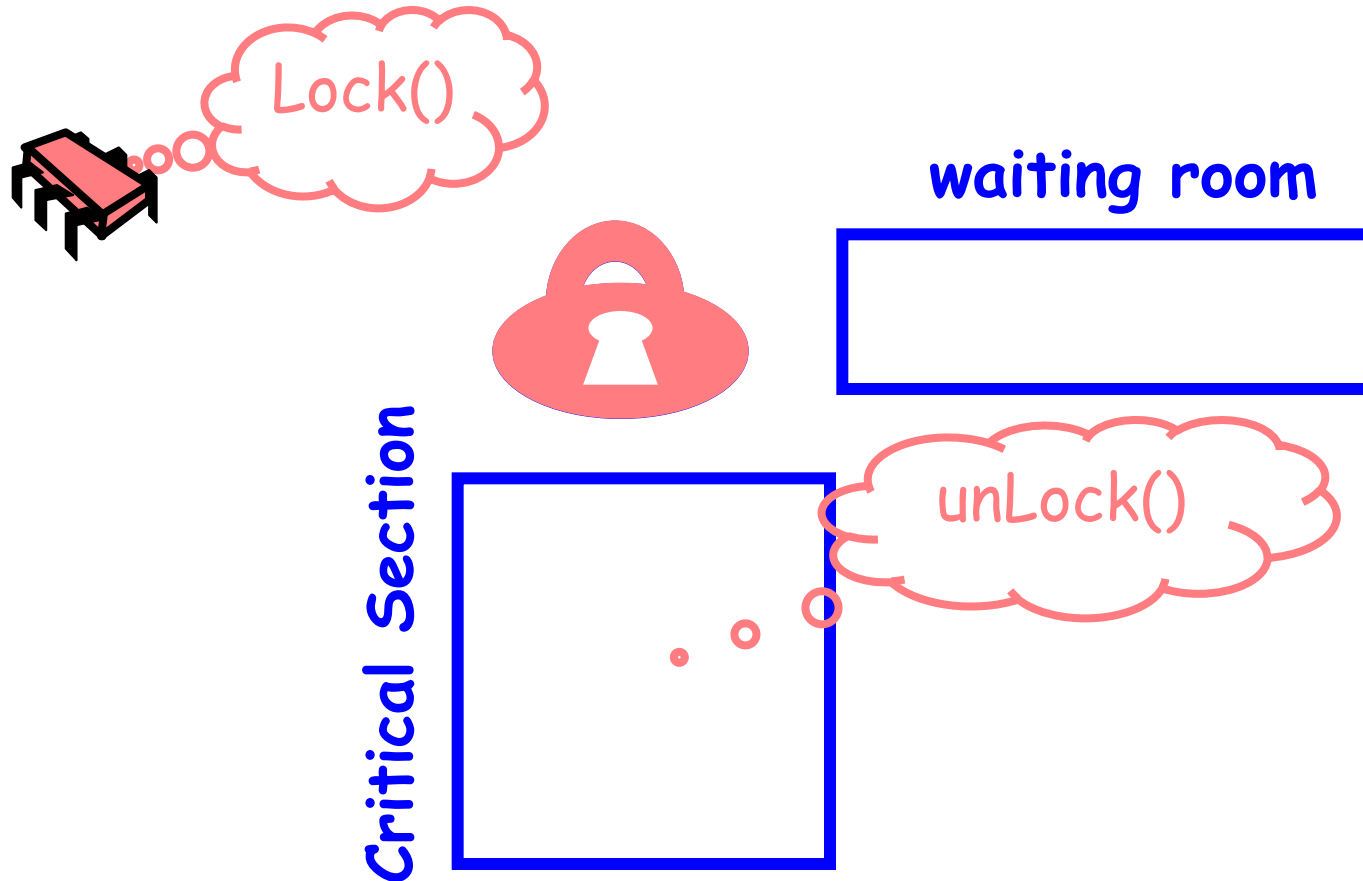
- Awakens one waiting thread
 - Which will reacquire lock

Signal All

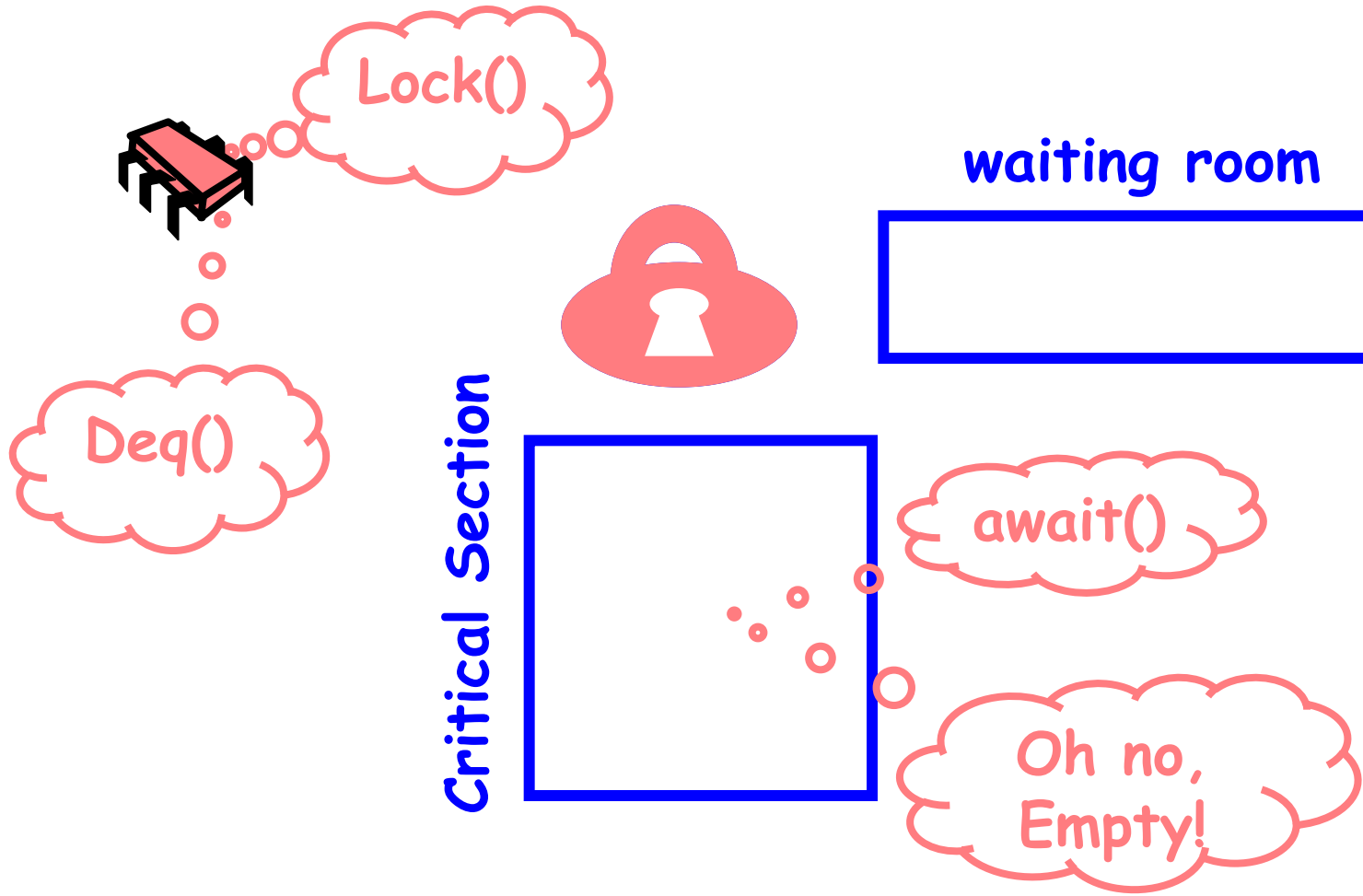
```
q.signalAll();
```

- Awakens all waiting threads
 - Which will each reacquire lock

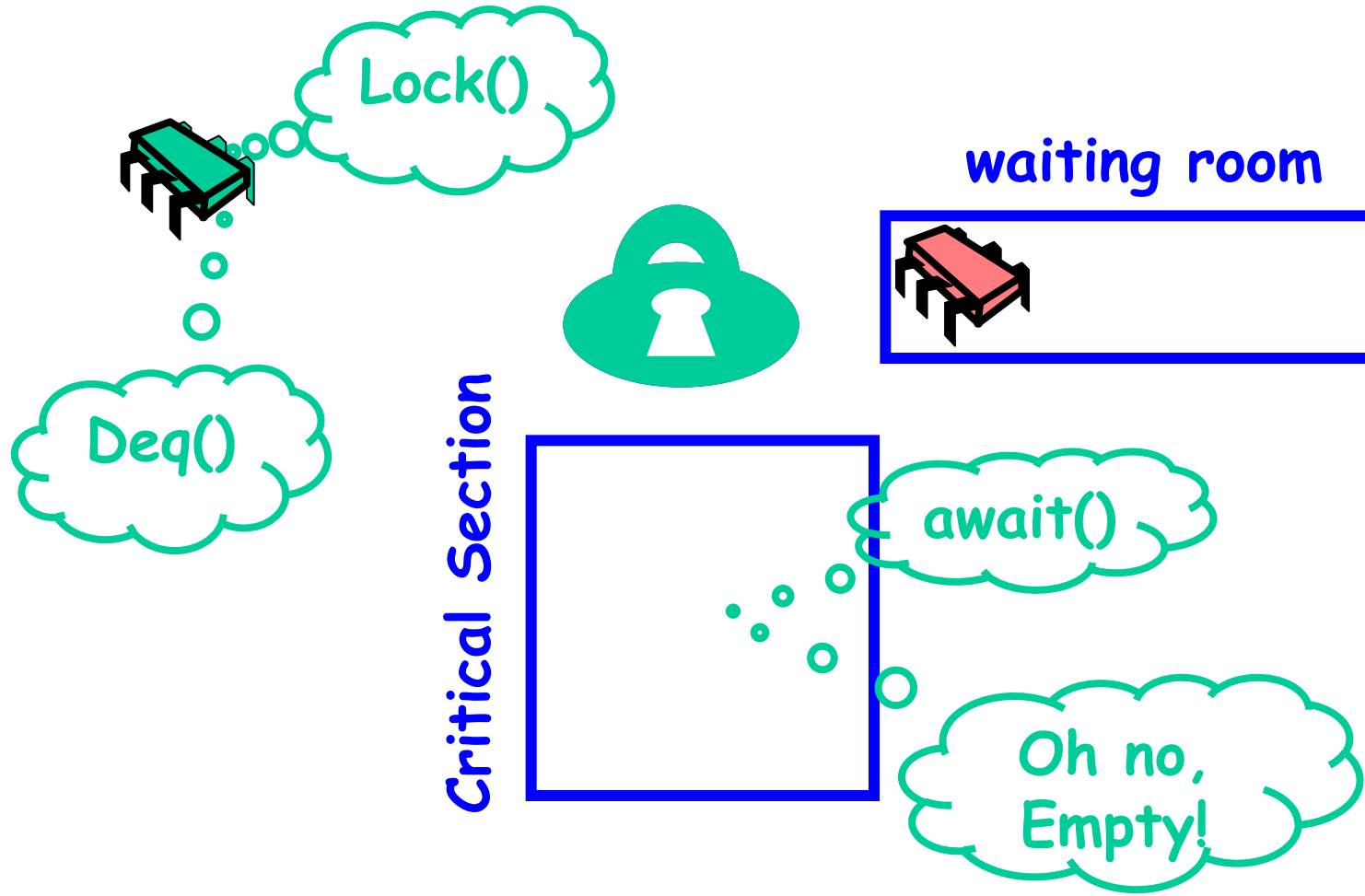
A Monitor Lock



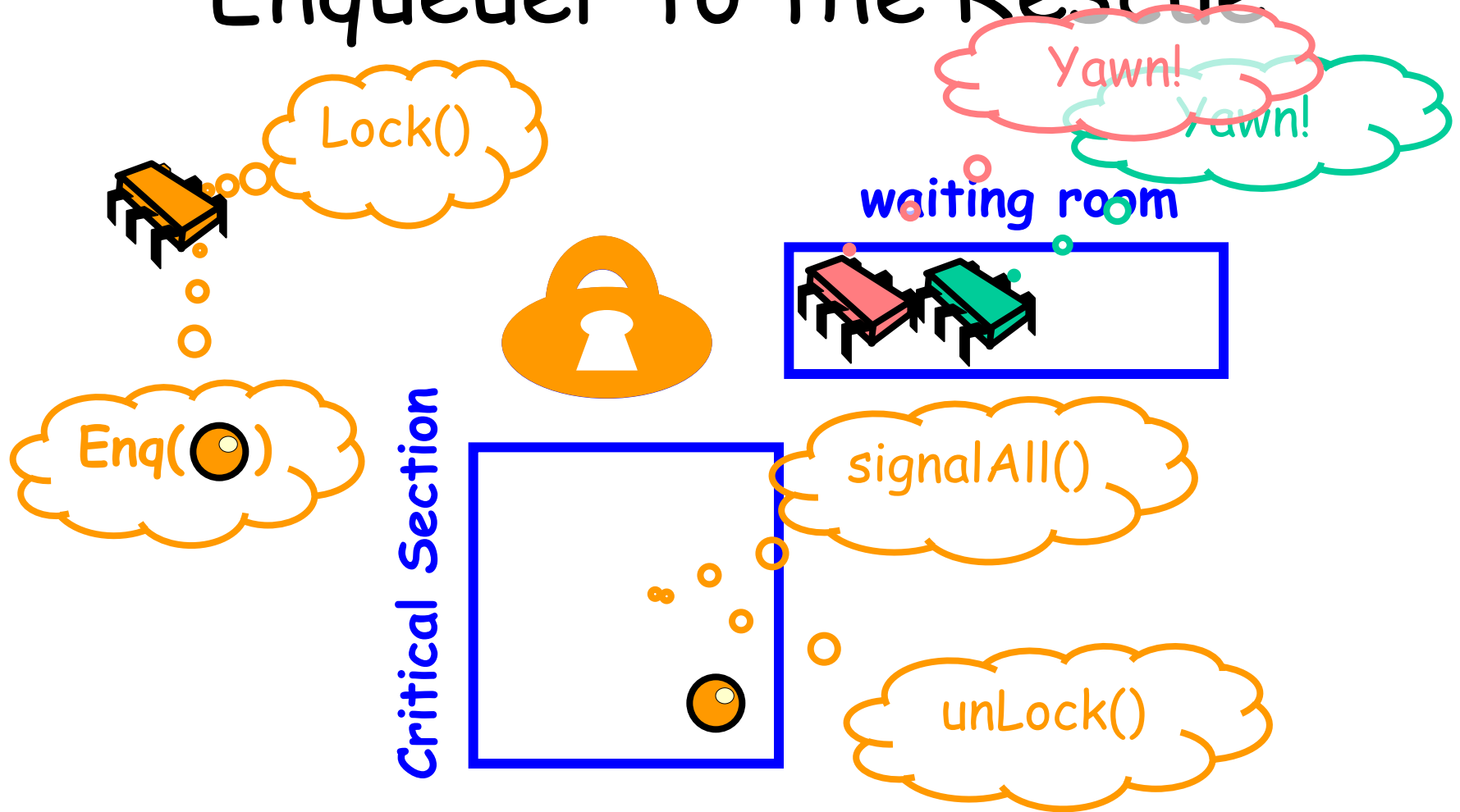
Unsuccessful Deq



Another One



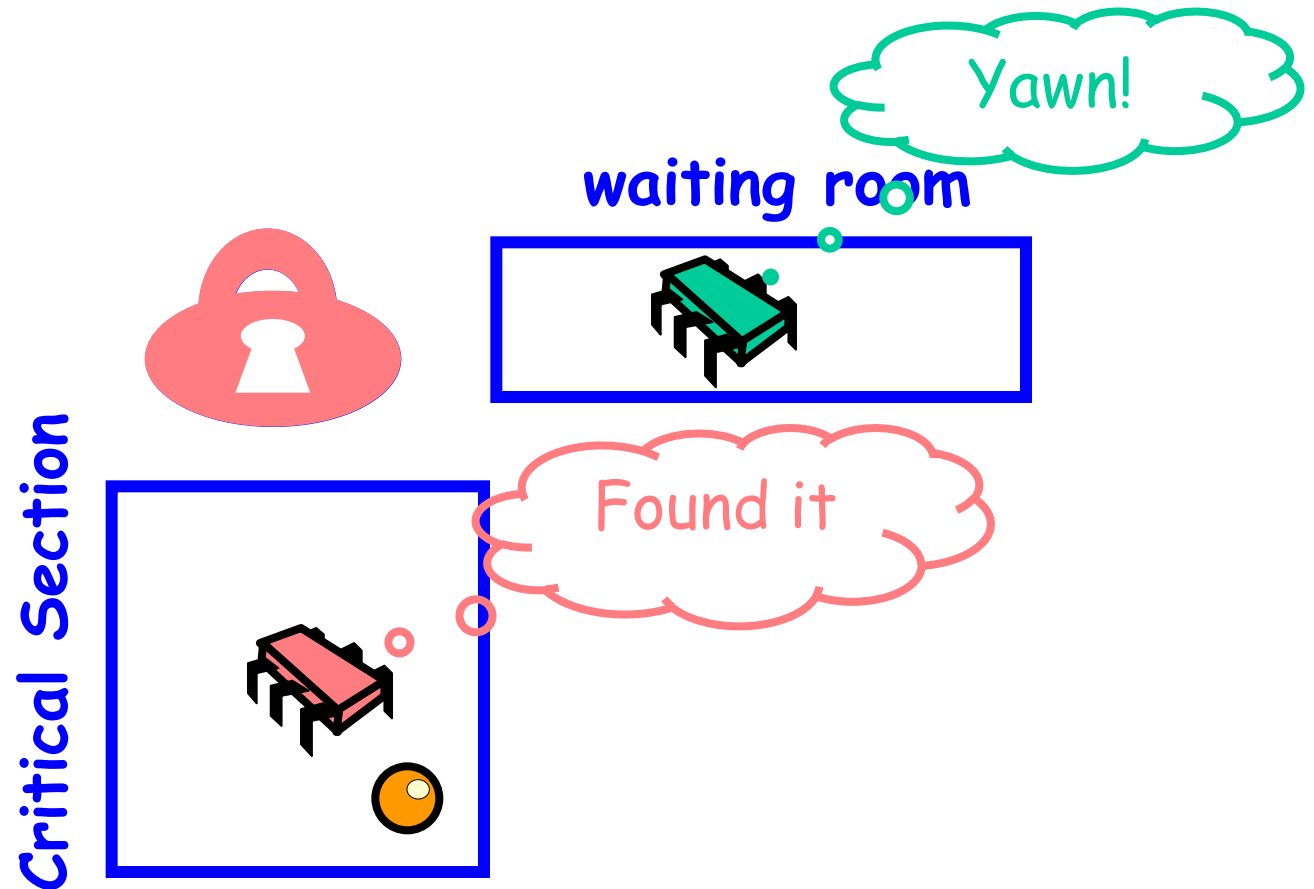
Enqueuer to the Rescue



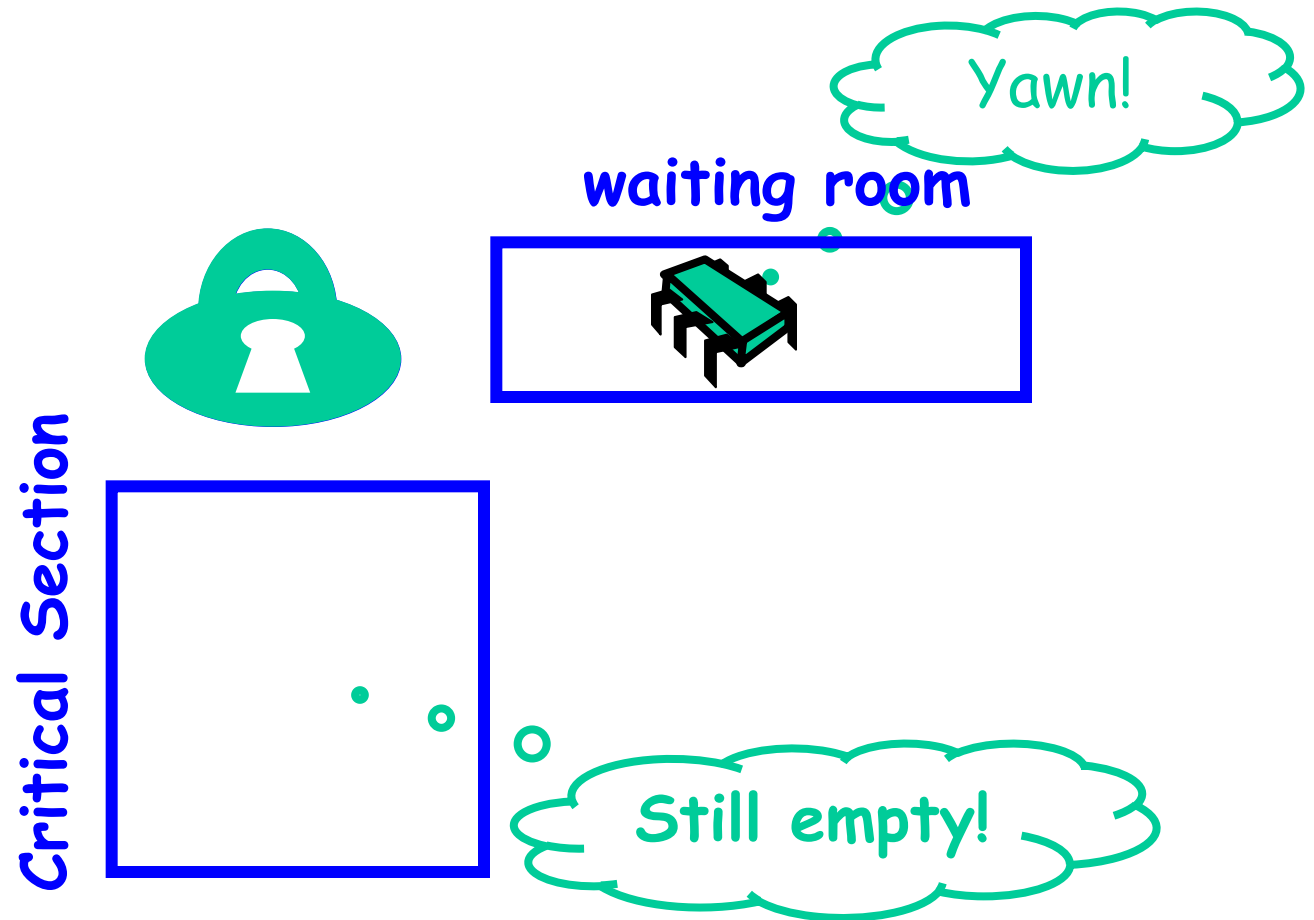
Monitor Signalling



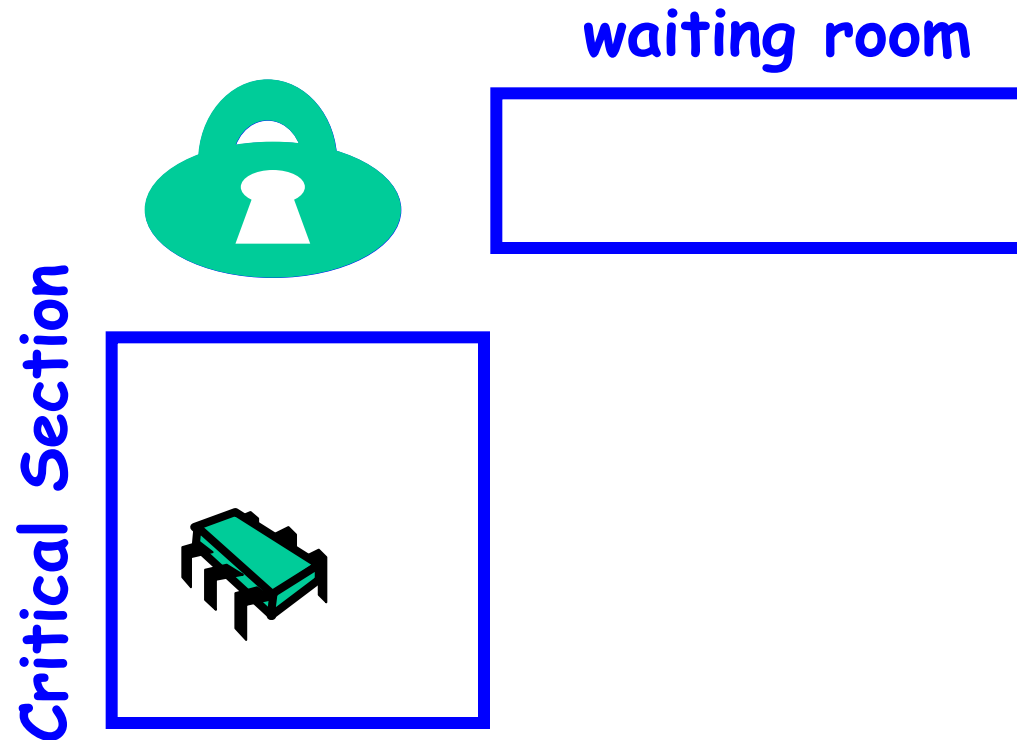
Dequeurs Signalled



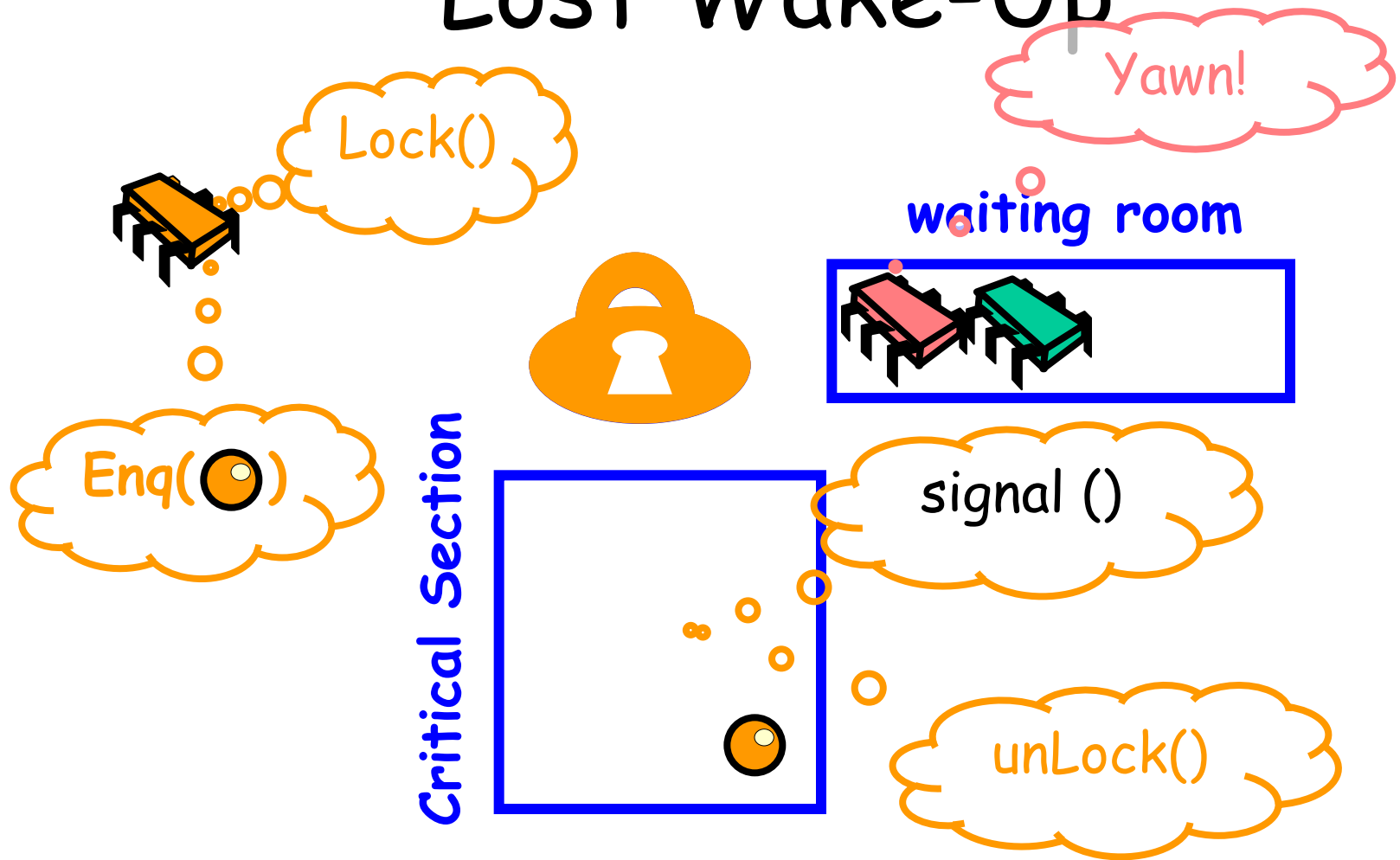
Dequeurs Signalled



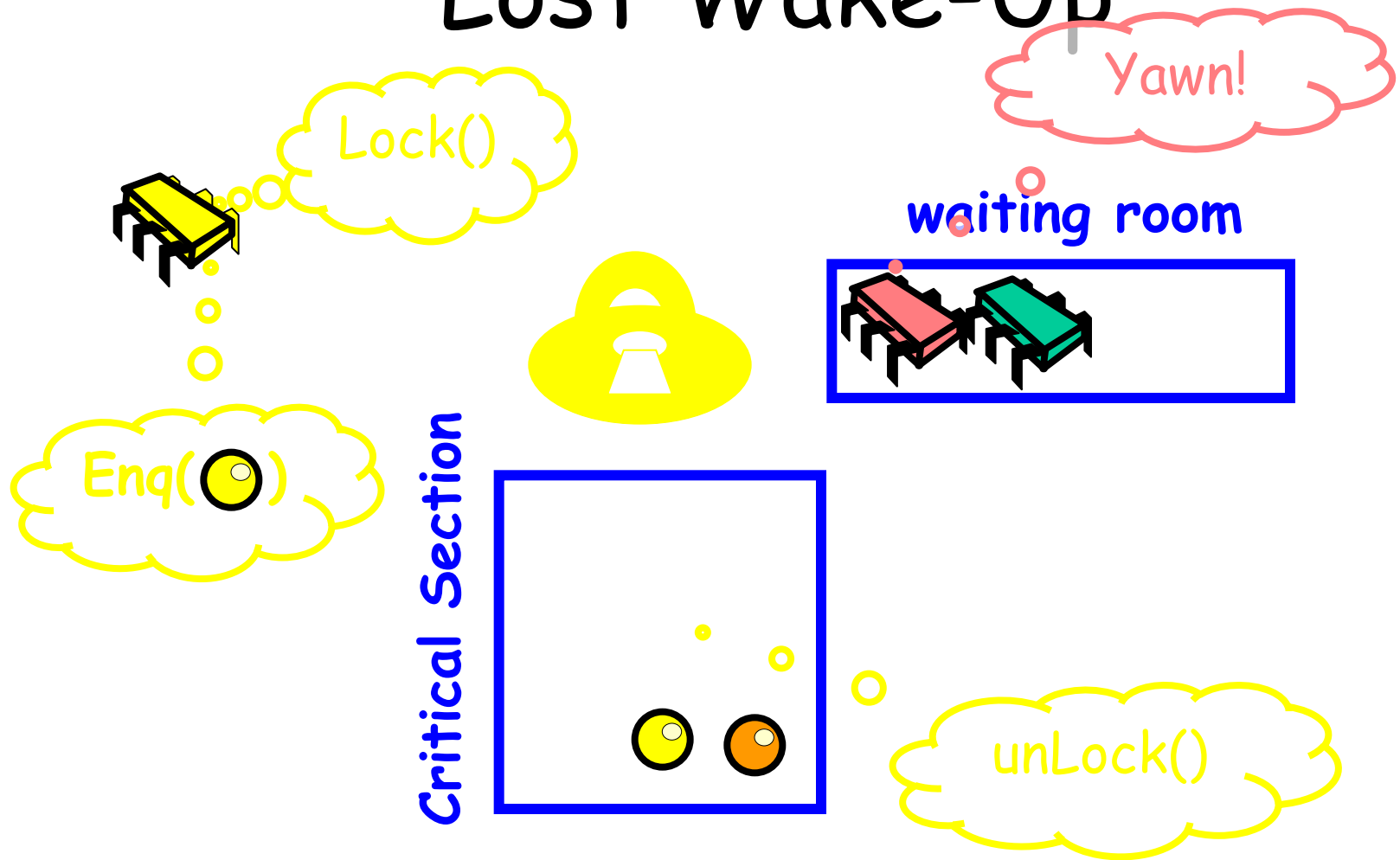
Dollar Short + Day Late



Lost Wake-Up



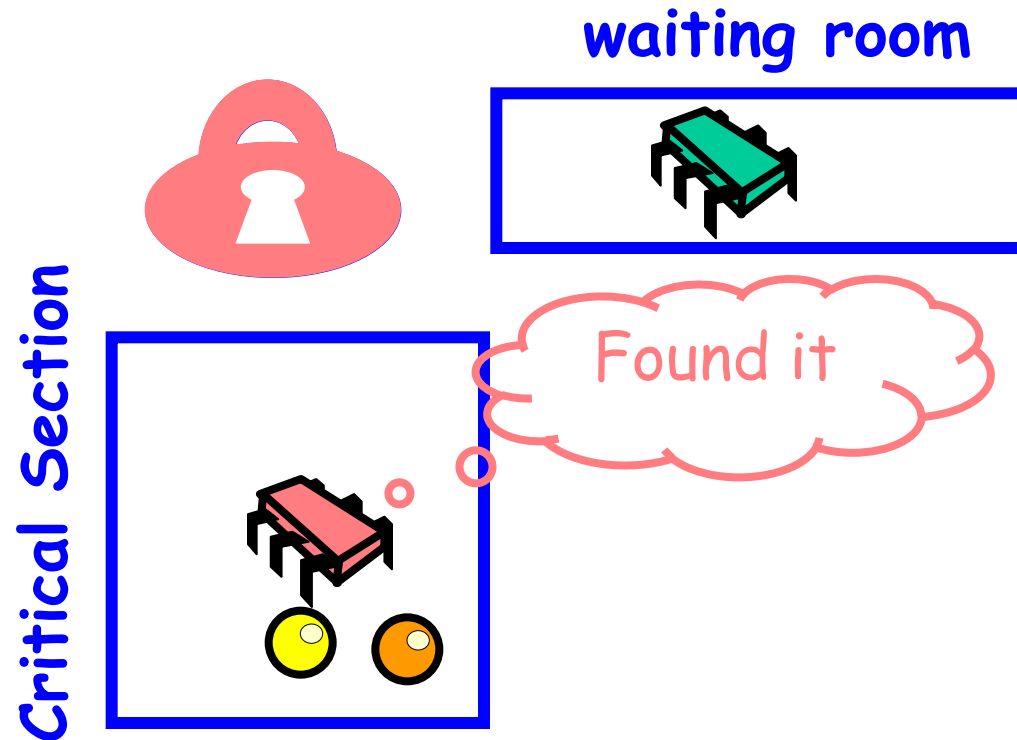
Lost Wake-Up



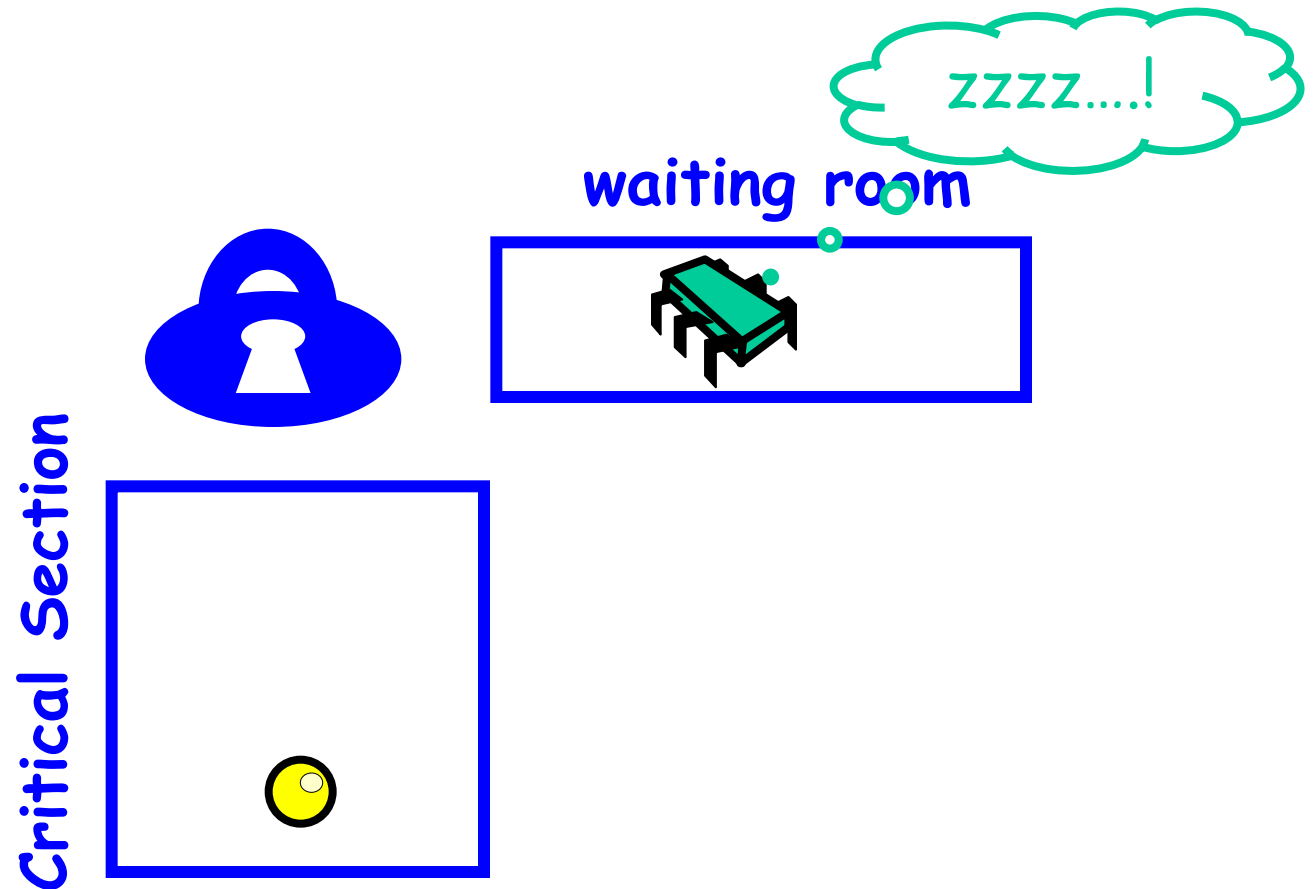
Lost Wake-Up



Lost Wake-Up



What's Wrong Here?



Solution to Lost Wakeup

- Always use signalAll and notifyAll
- Not signal and notify

Java Synchronized Methods

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

Java Synchronized Methods

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

Each object has an implicit lock with an implicit condition

Java Synchronized Methods

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

Lock on entry,
unlock on return



Java Synchronized Methods

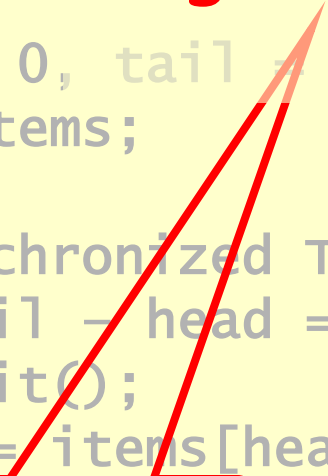
```
public class Queue<T> {  
    int head = 0, tail = 0;  
    T[QSIZE] items;  
  
    public synchronized T deq() {  
        while (tail - head == 0)  
            this.wait();  
        T result = items[head % QSIZE]; head++;  
        this.notifyAll();  
        return result;  
    }  
    ...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

Wait on implicit condition

Java Synchronized Methods

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

Signal all threads waiting on condition

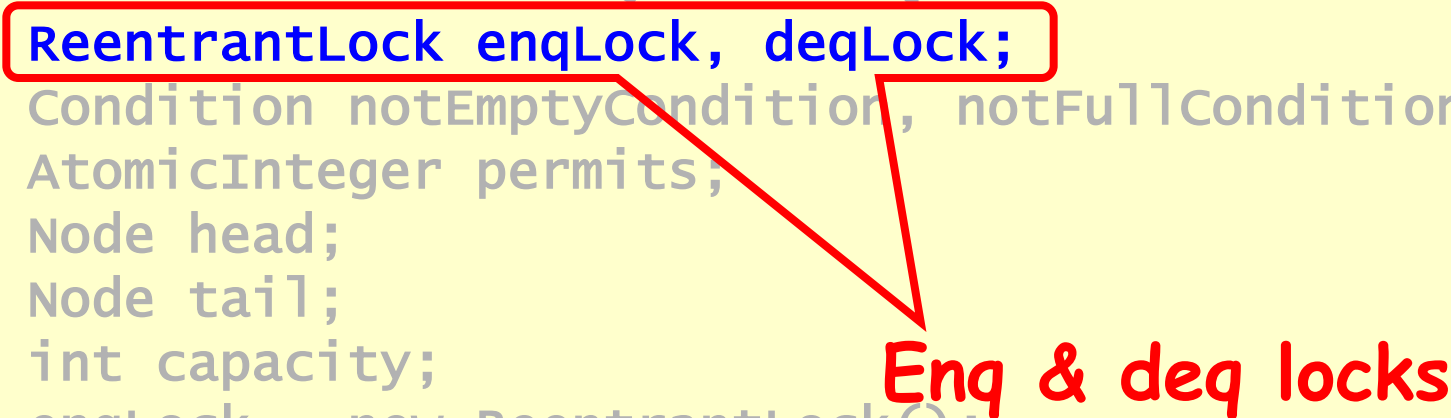


(Pop!) The Bounded Queue

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

Bounded Queue Fields

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

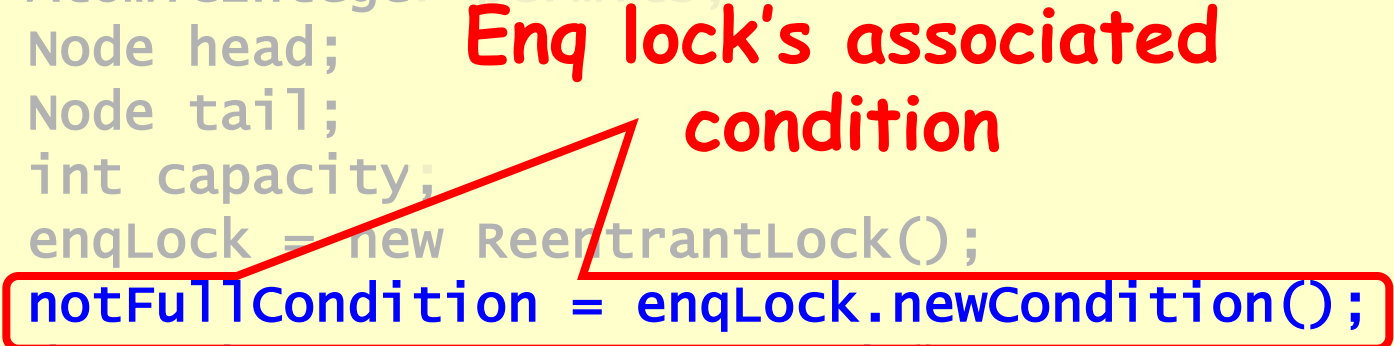


Enq & deq locks

Bounded Queue Fields

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

Enq lock's associated condition



Bounded Queue Fields

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

Num permits: 0 to capacity

Bounded Queue Fields

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

Head and Tail

Enq Method Part One

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

Enq Method Part One

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

Lock and unlock
enq lock

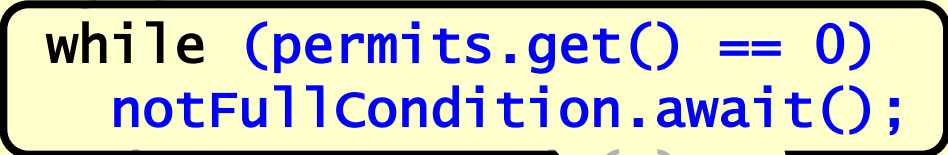
Enq Method Part One

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

**If queue is full, patiently
await further instructions ...**

Be Afraid

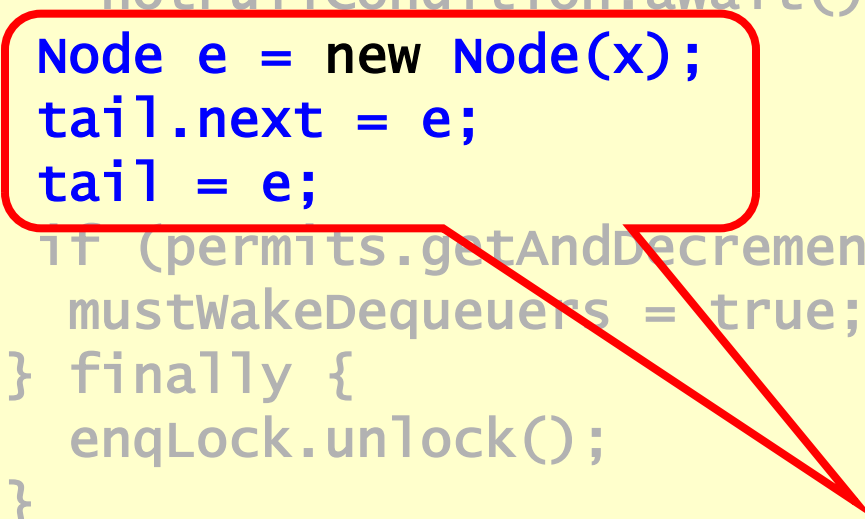
```
public void enq(T x) {  
    boolean mustWakeDequeueuers = false;  
    enqLock.lock();  
    try {  
        while (permits.get() == 0)  
            notFullCondition.await();  
        Node e = new Node(x);  
        tail.next = e;  
        tail = e;  
        if (permits.getAndDecrement() ==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



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

Enq Method Part One

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



Add new node

Enq Method Part One

```
public void enq(T x) {  
    boolean mustWakeDequeuers = false;  
    enqLock.lock();  
    try {  
        while (permits.get() == 0)  
            notFullCondition.await();  
        Node e = new Node(x);  
        tail.next = e;  
        tail = e;
```

```
        if (permits.getAndDecrement() == capacity)  
            mustWakeDequeuers = true;  
    } finally {  
        enqLock.unlock();  
    }  
    ...  
}
```

**If queue was empty, wake
frustrated dequeuers**

Enq Method Part Deux

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

Enq Method Part Deux

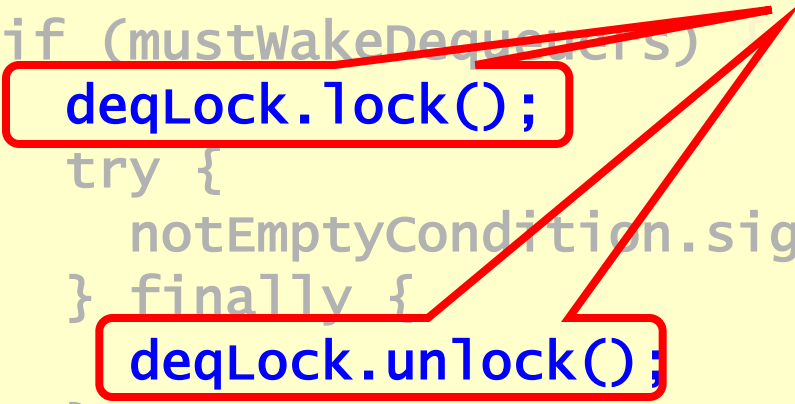
```
public void enq(T x) {  
    ...  
    if (mustWakeDequeuers) {  
        deqLock.lock();  
        try {  
            notEmptyCondition.signalAll();  
        } finally {  
            deqLock.unlock();  
        }  
    }  
}
```

Are there dequeuers to be signaled?

Enq Method Part Deux

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

**Lock and unlock
deq lock**



Enq Method Part Deux

Signal dequeuers that
queue 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?

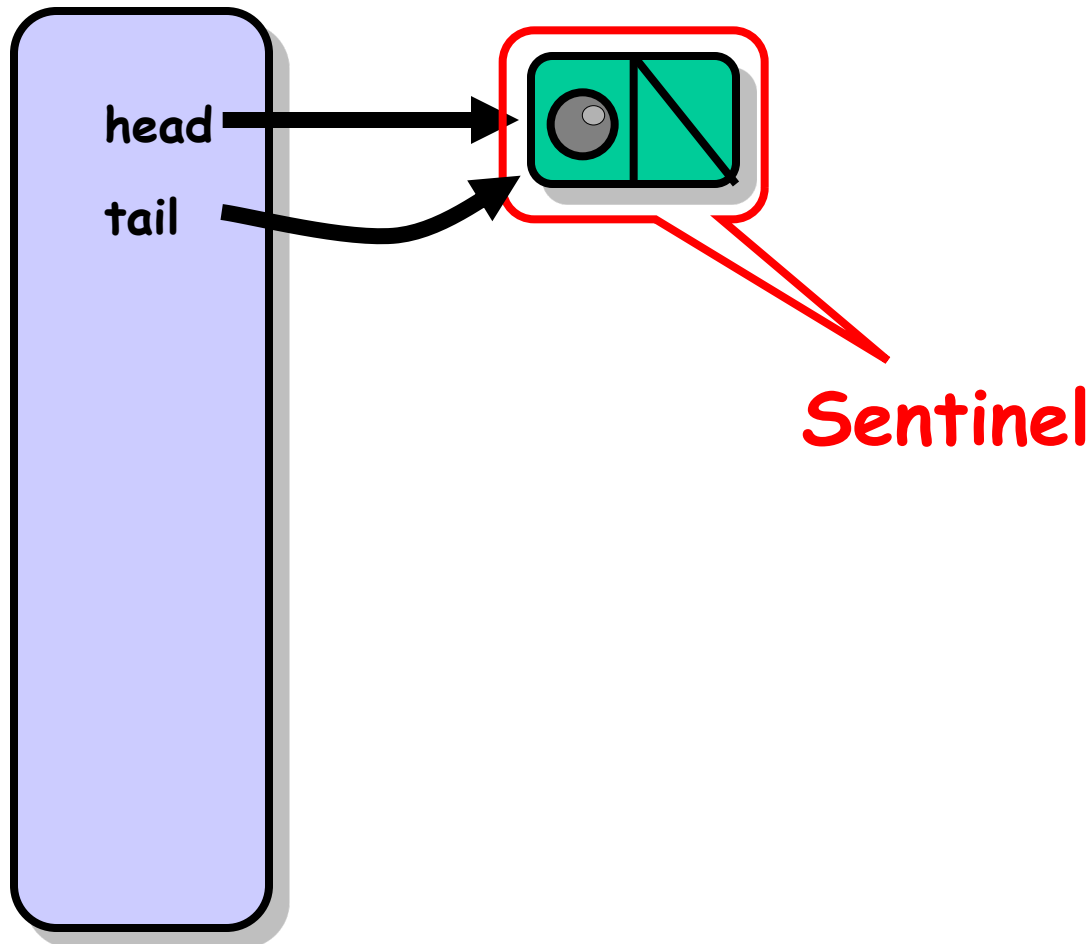
Split the Counter

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

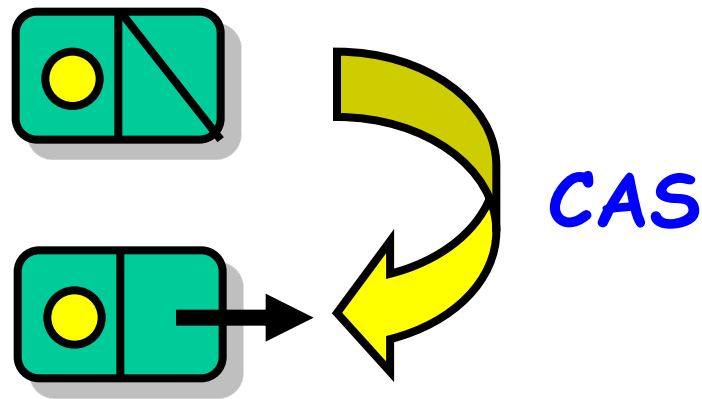
Split Counter

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

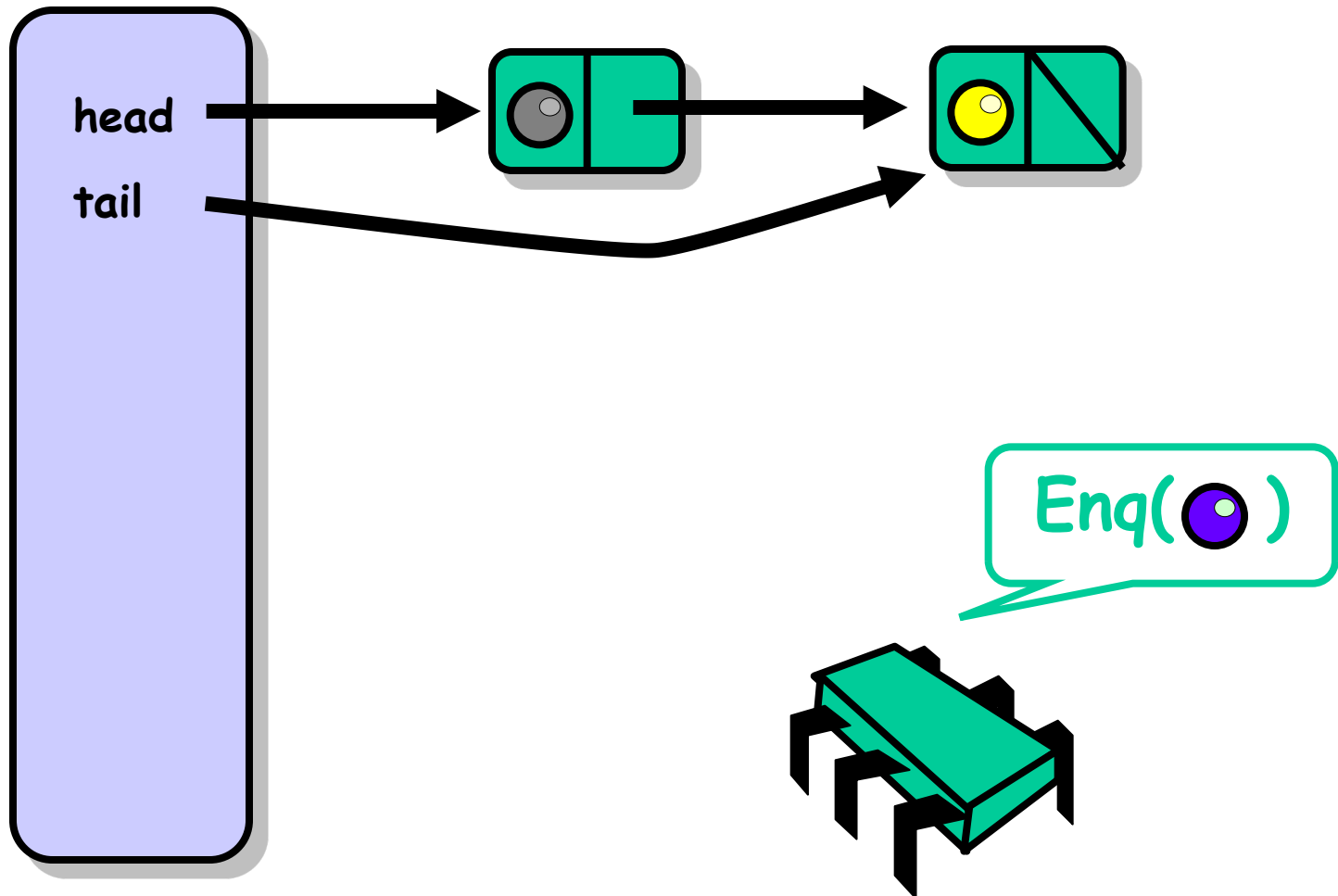
A Lock-Free Queue



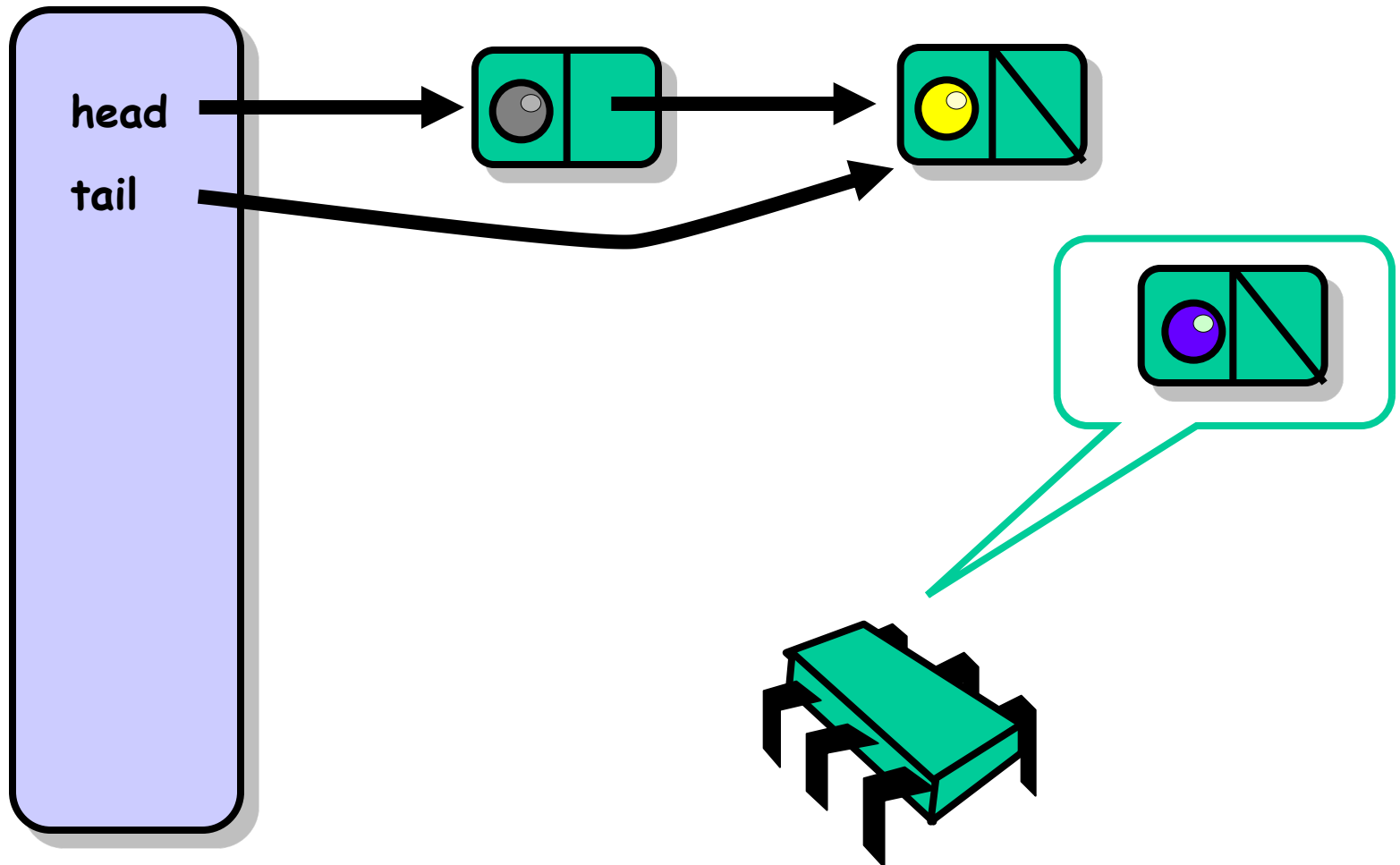
Compare and Set



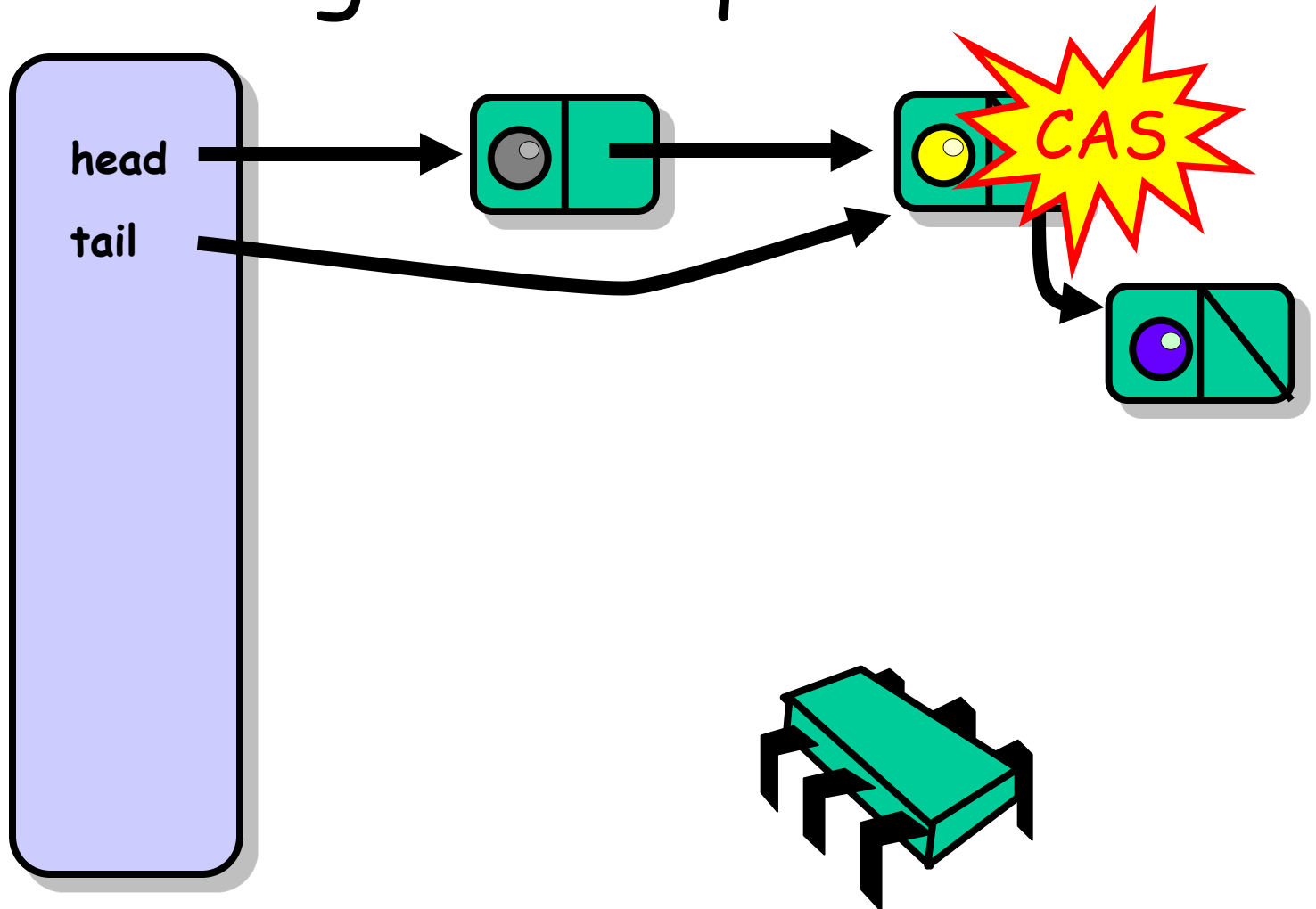
Enqueue



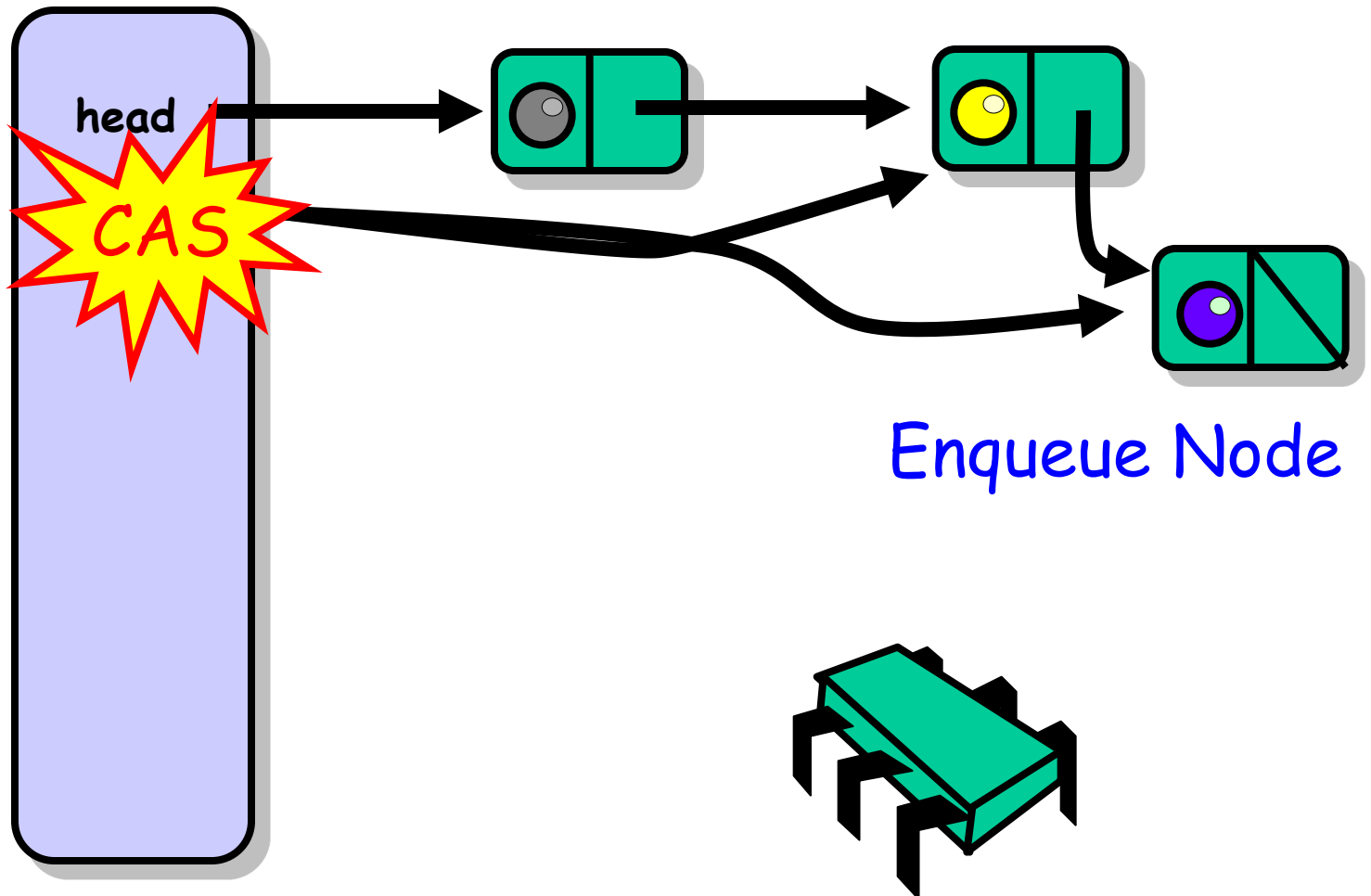
Enqueue



Logical Enqueue



Physical Enqueue



Enqueue

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

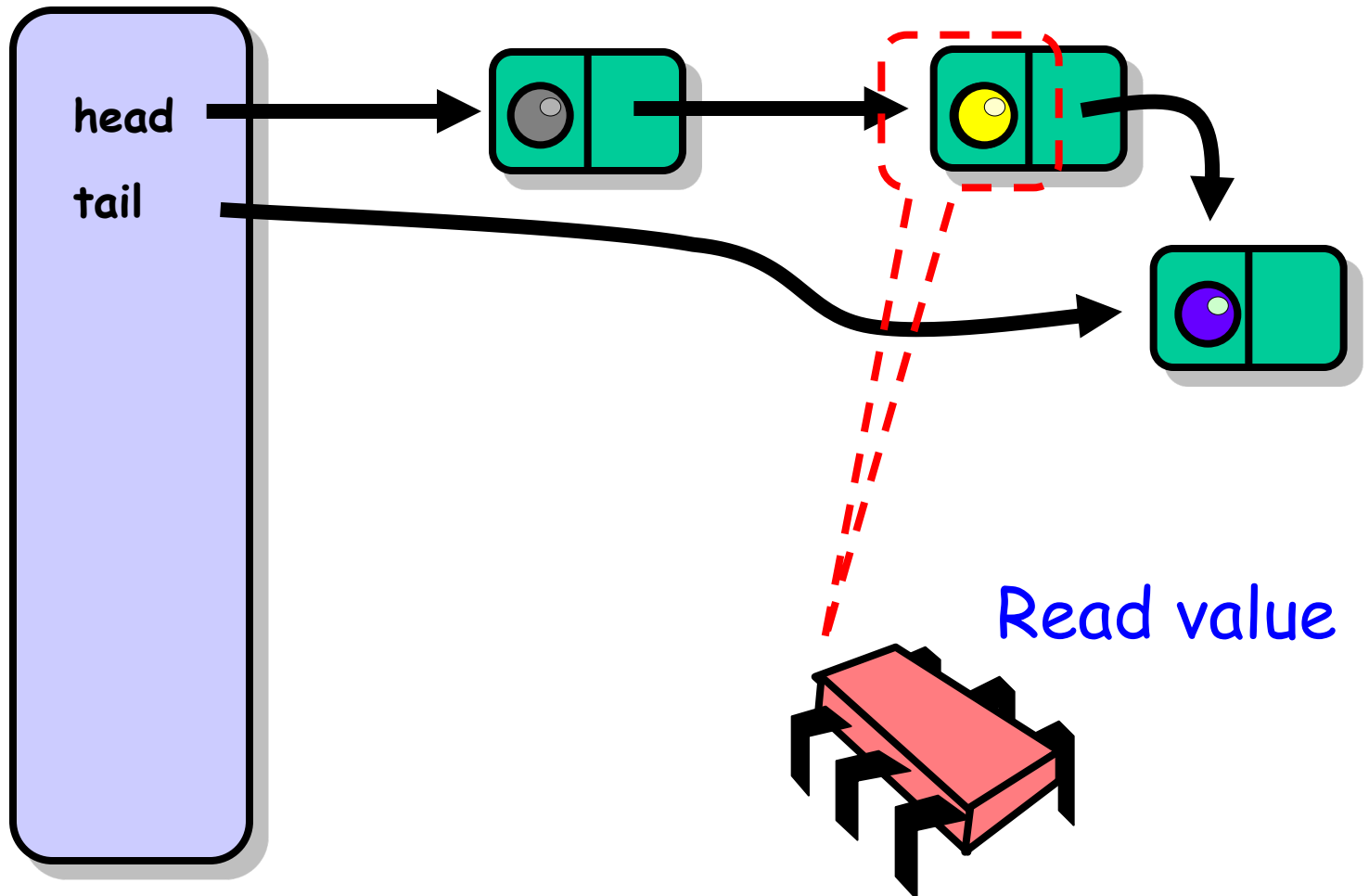
Enqueue

- What do you do if you find
 - A trailing **tail**?
- Stop and 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

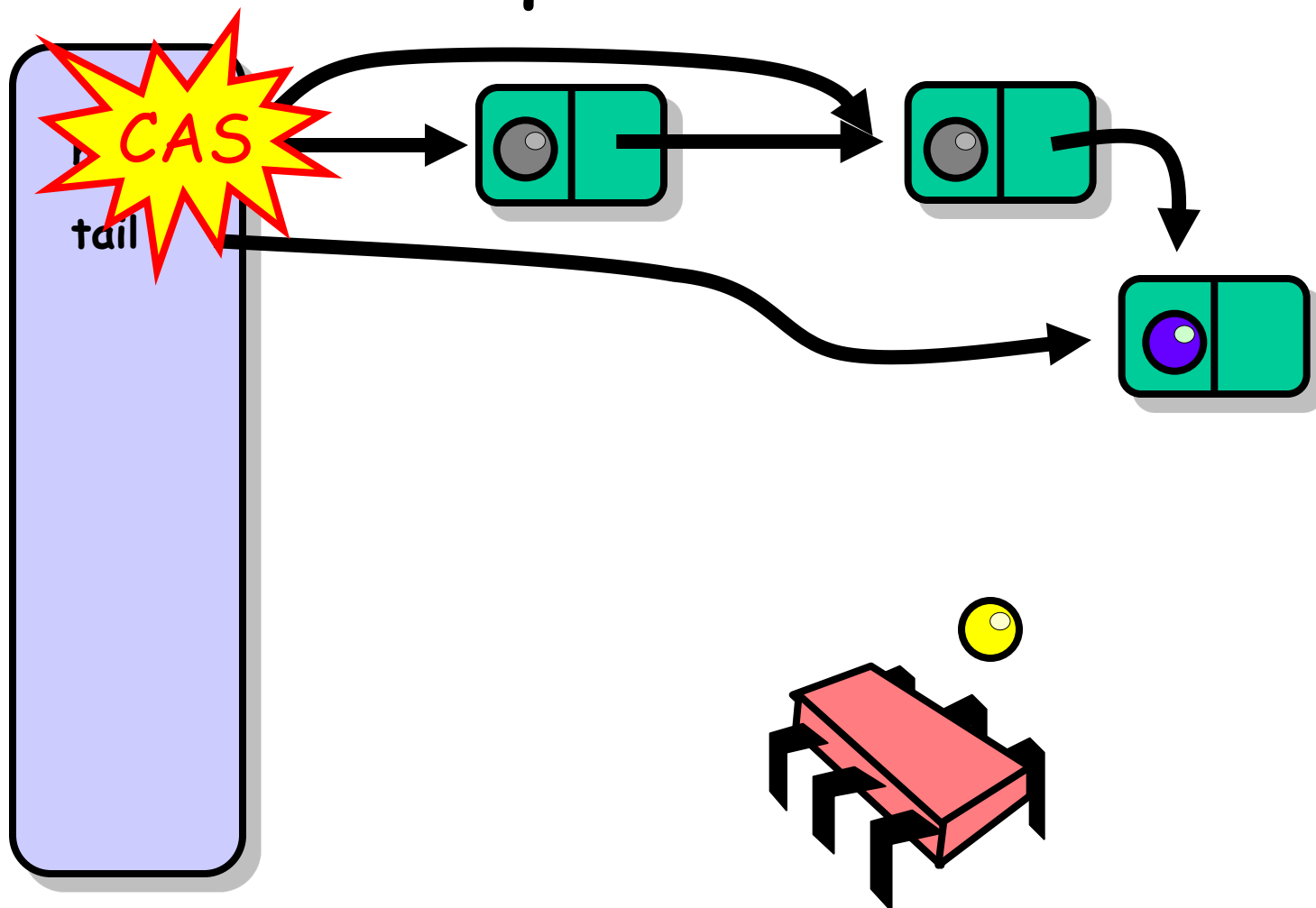
- During logical enqueue
 - Abandon hope, restart
 - Still lock-free (why?)
- During physical enqueue
 - Ignore it (why?)

Dequeuer



Make first Node
new sentinel

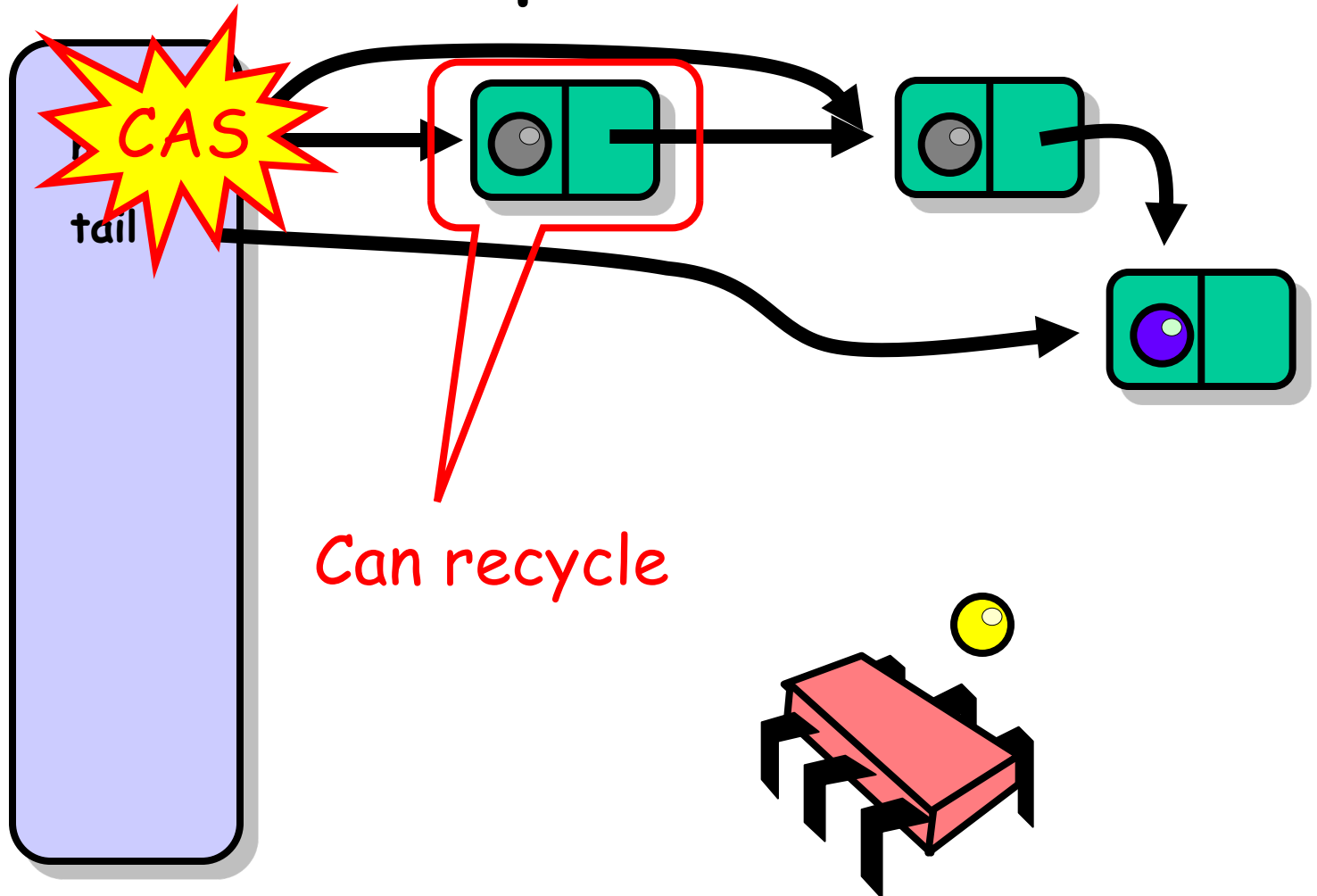
Dequeuer



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?

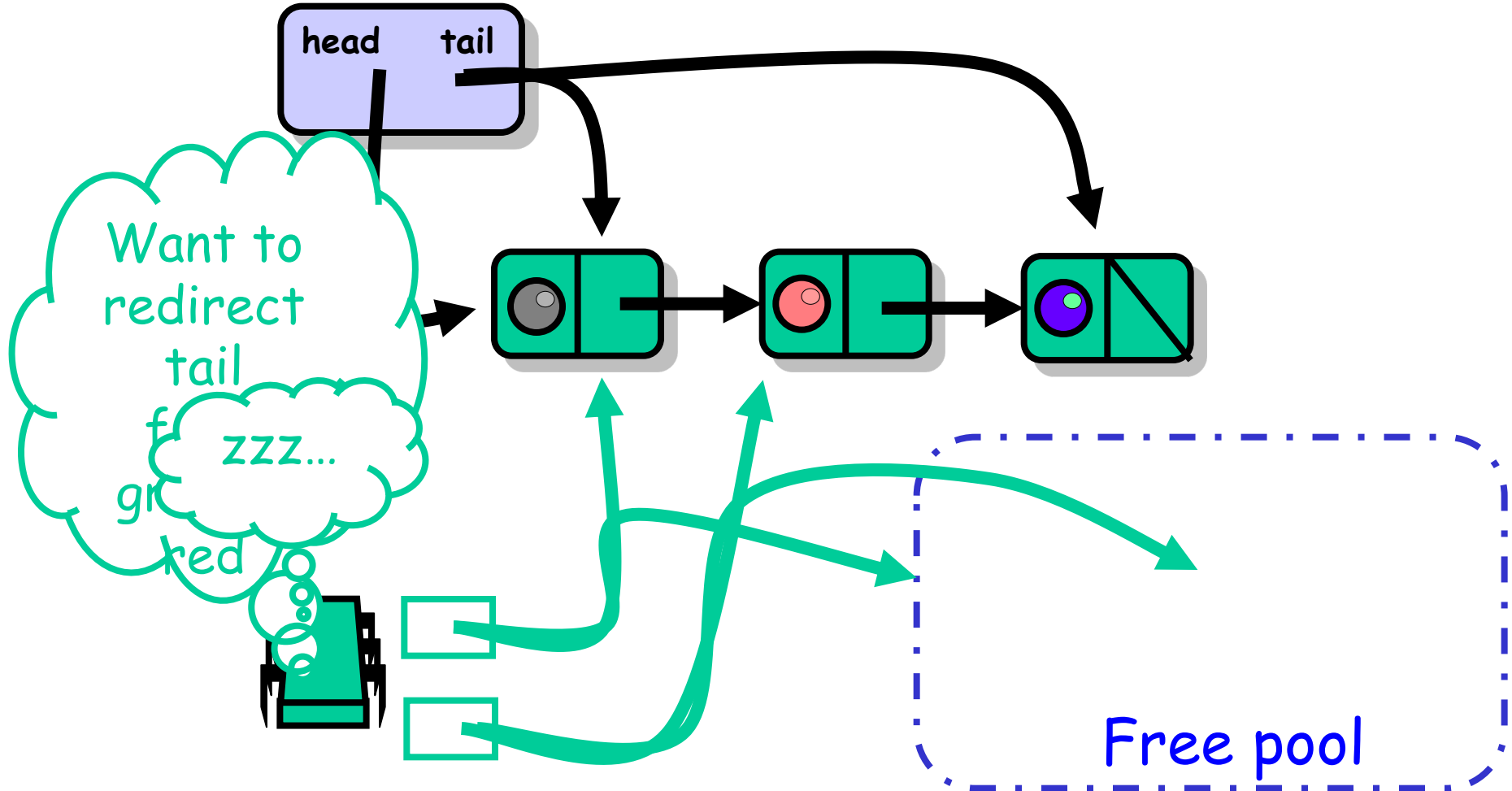
Dequeuer



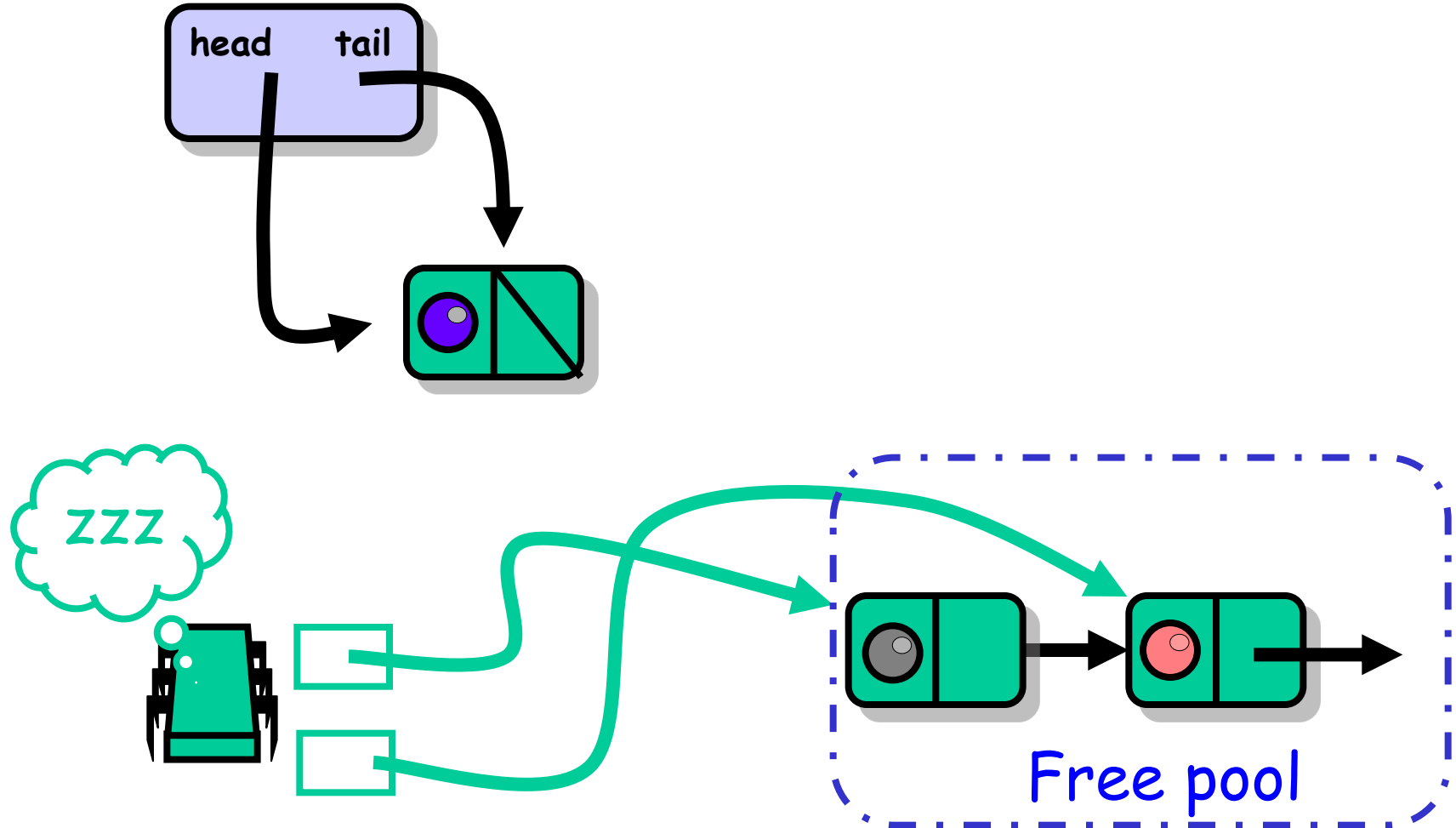
Simple Solution

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

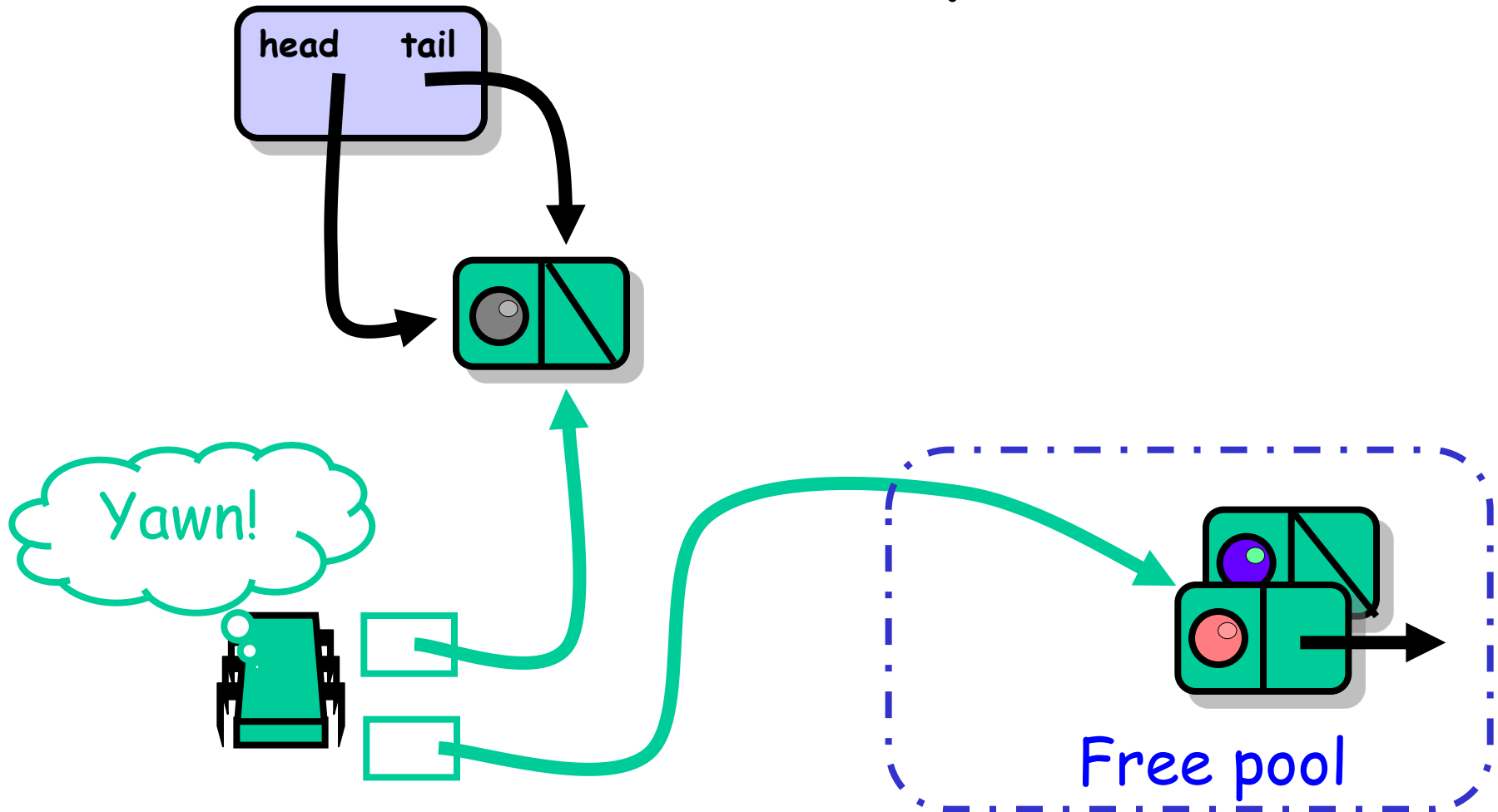
Why Recycling is Hard



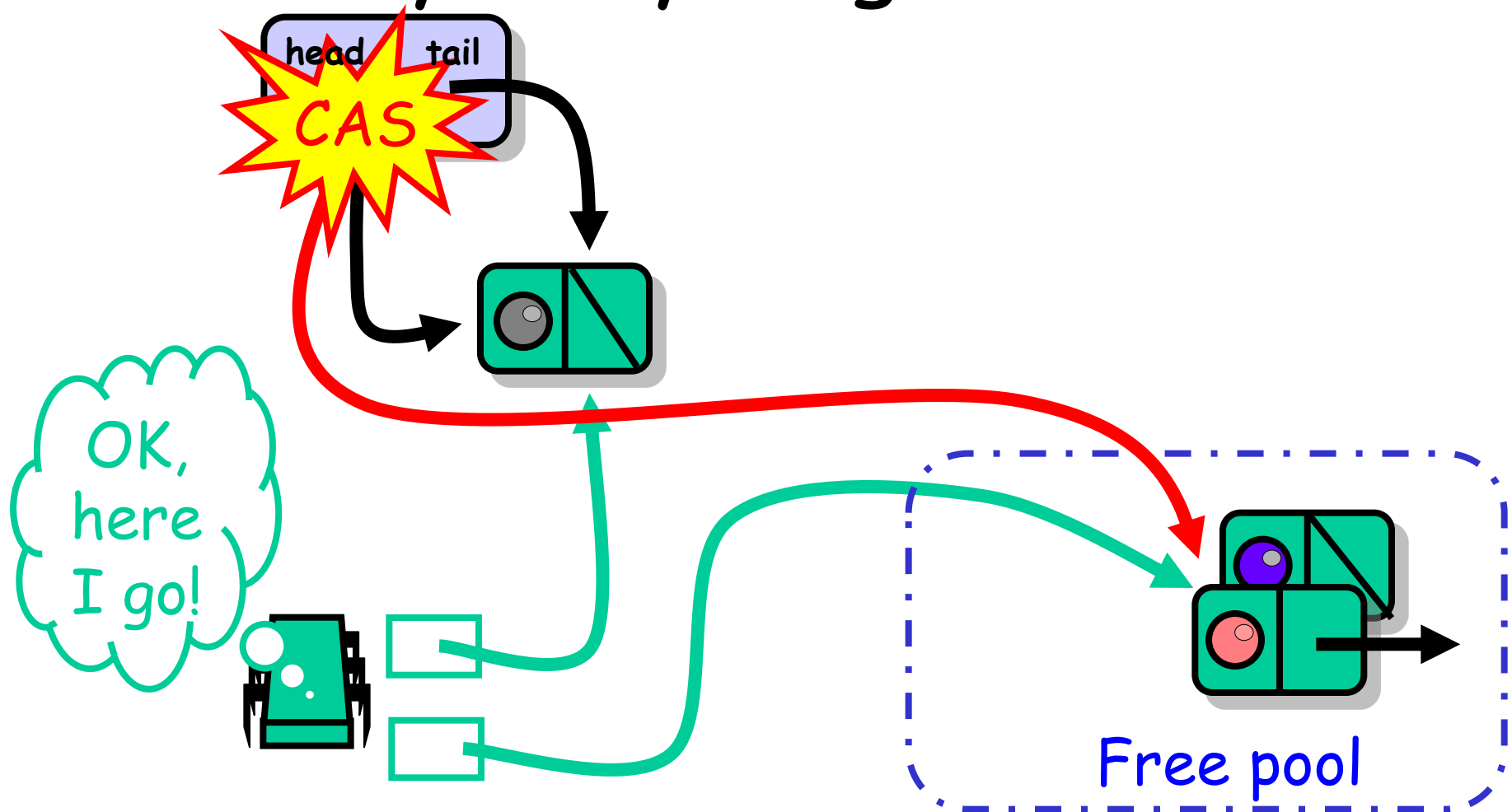
Both Nodes Reclaimed



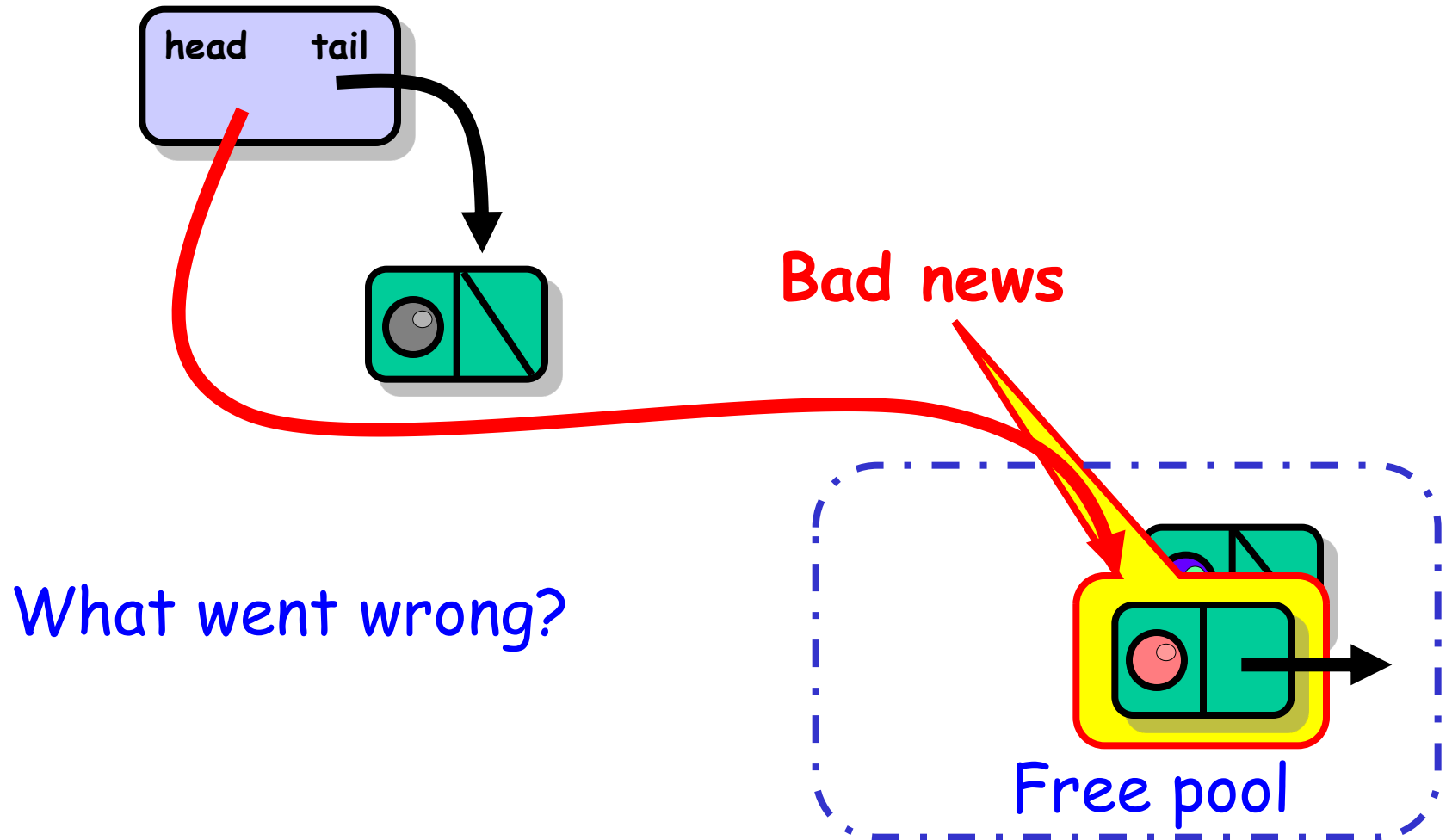
One Node Recycled



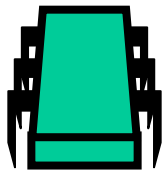
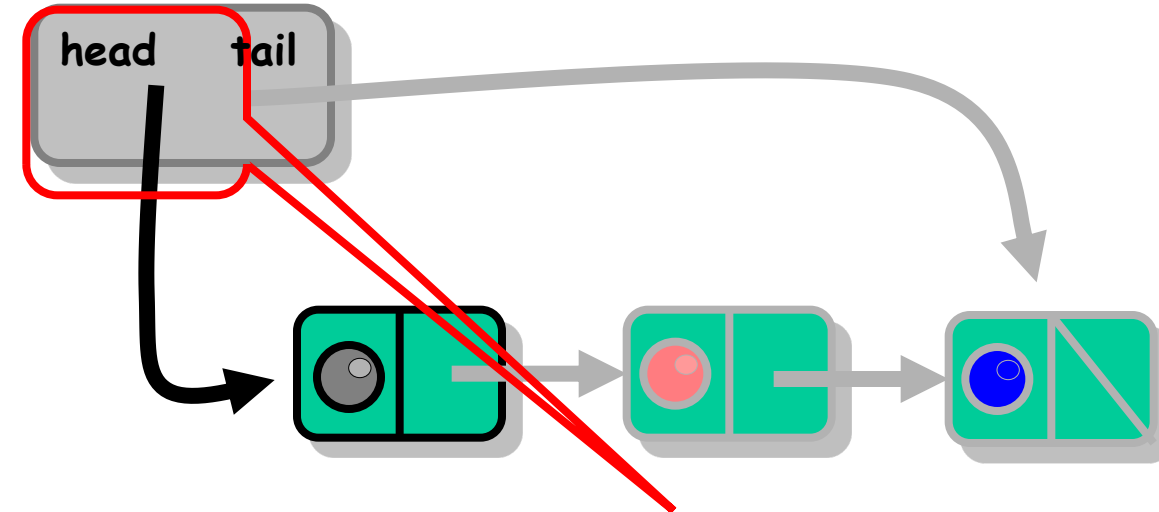
Why Recycling is Hard



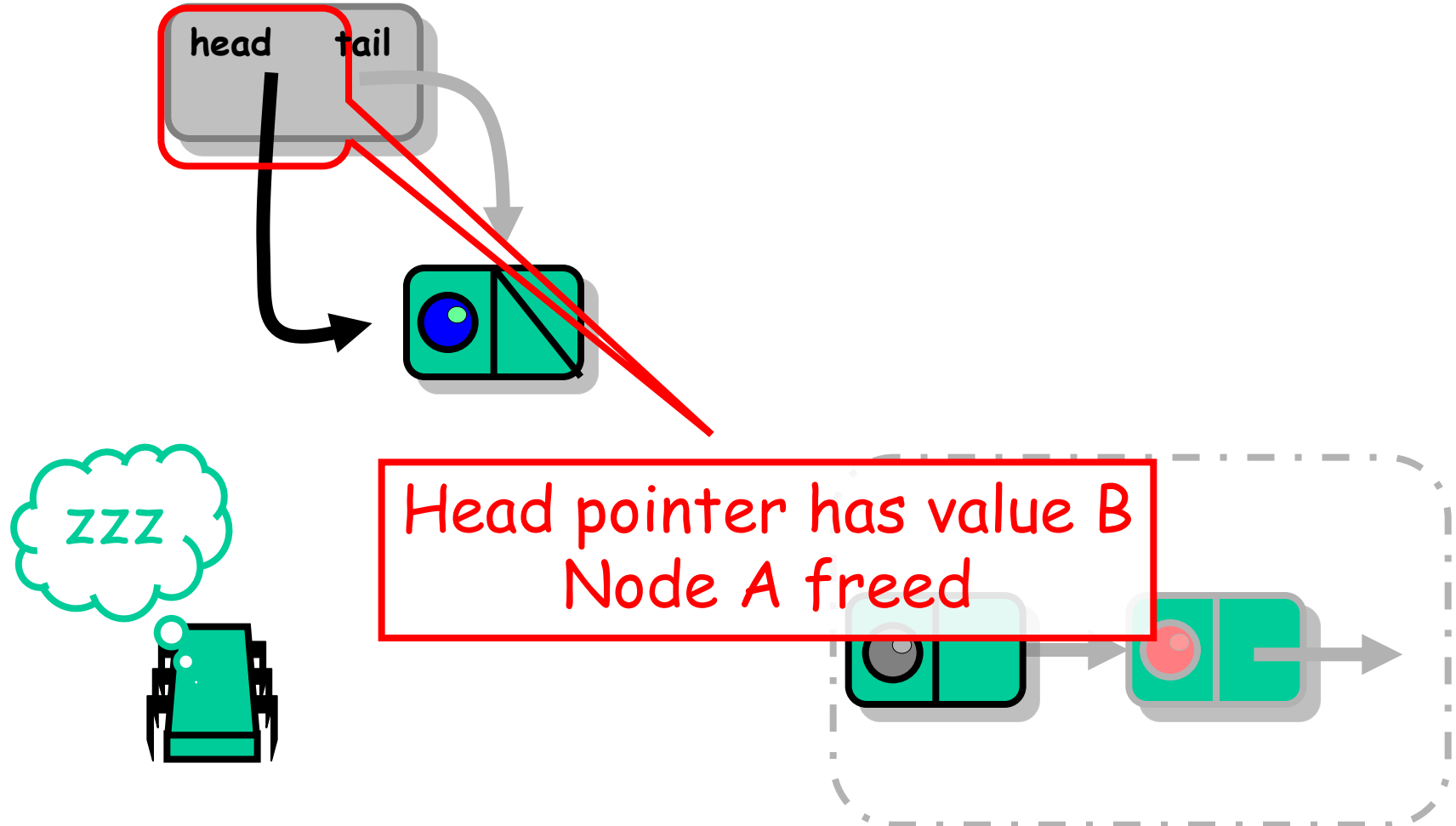
Final State



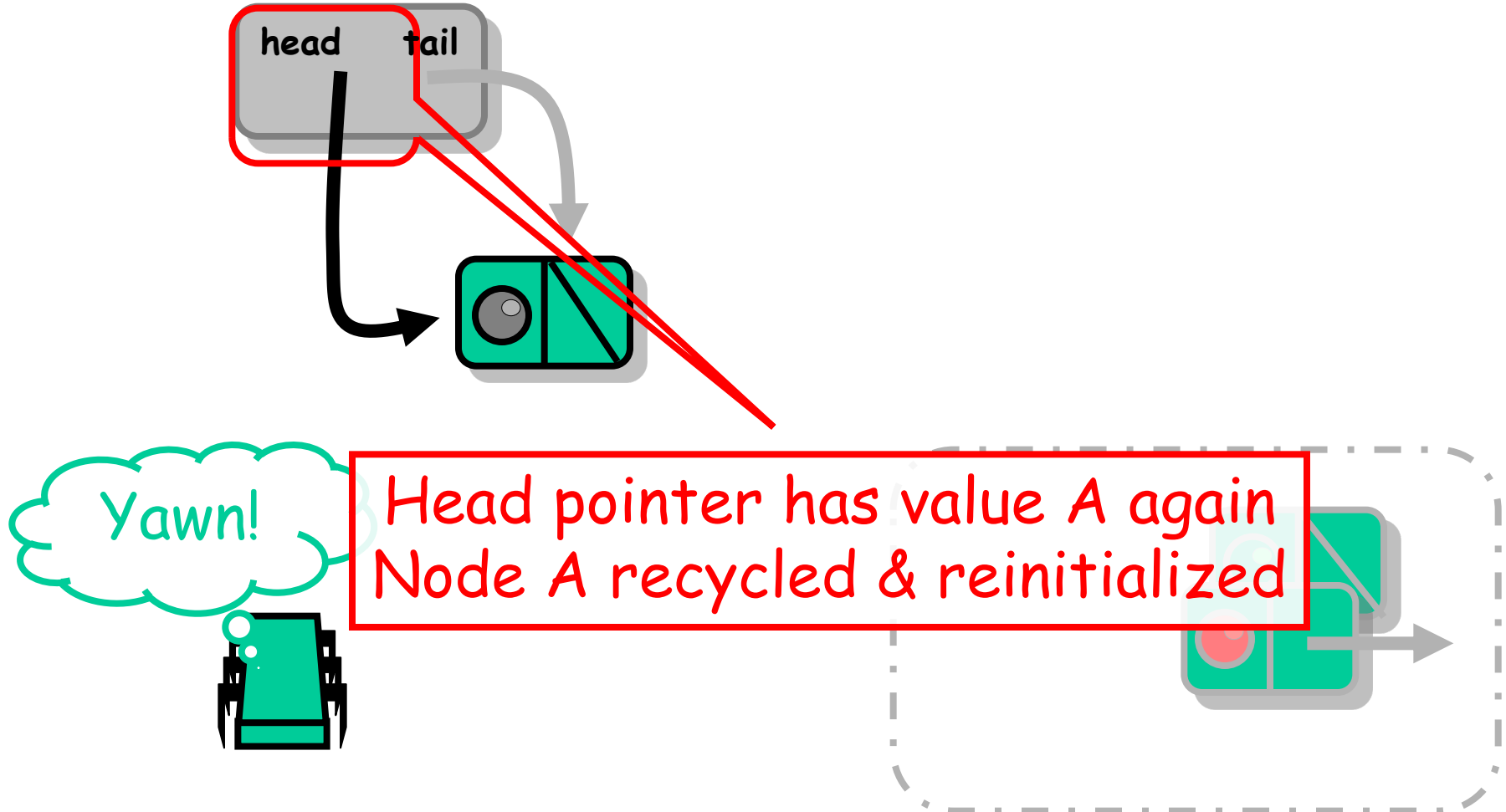
The Dreaded ABA Problem



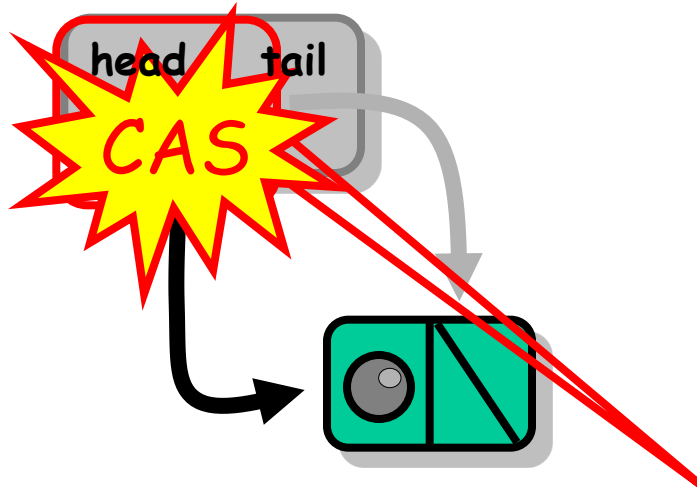
Dreaded ABA continued



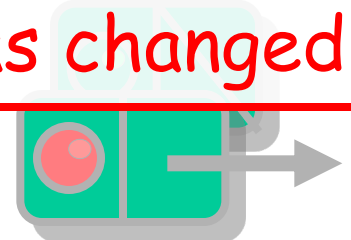
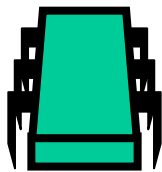
Dreaded ABA continued



Dreaded ABA continued



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



The Dreaded ABA Problem

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

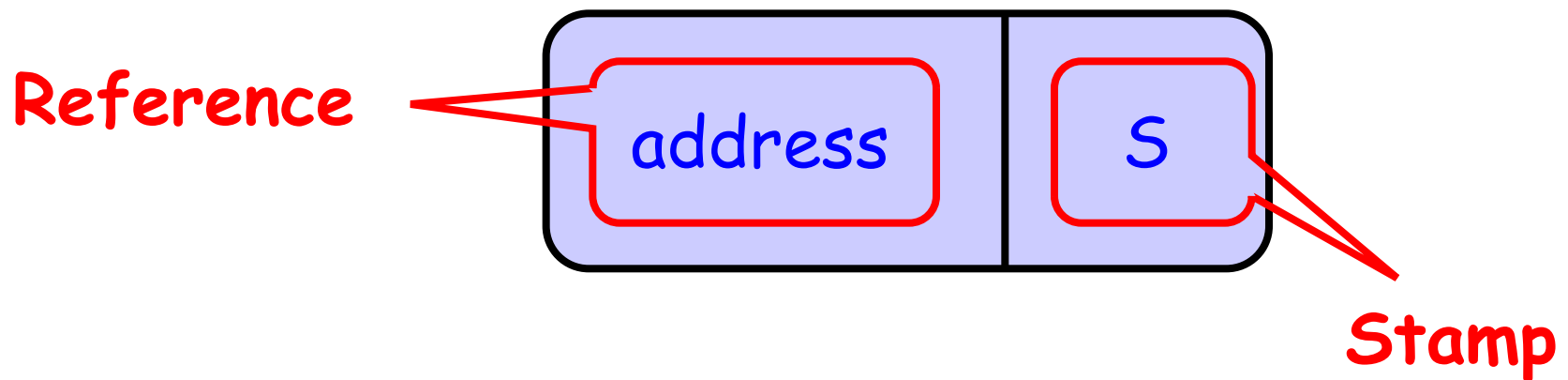
Dreaded ABA - A Solution

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

Atomic Stamped Reference

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

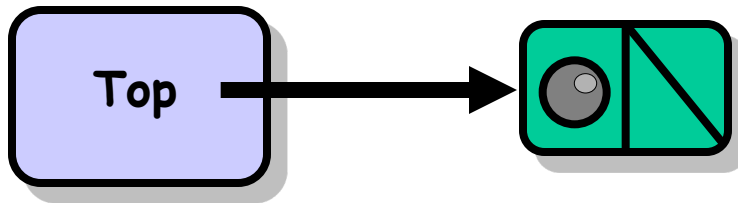
Can get reference and stamp atomically



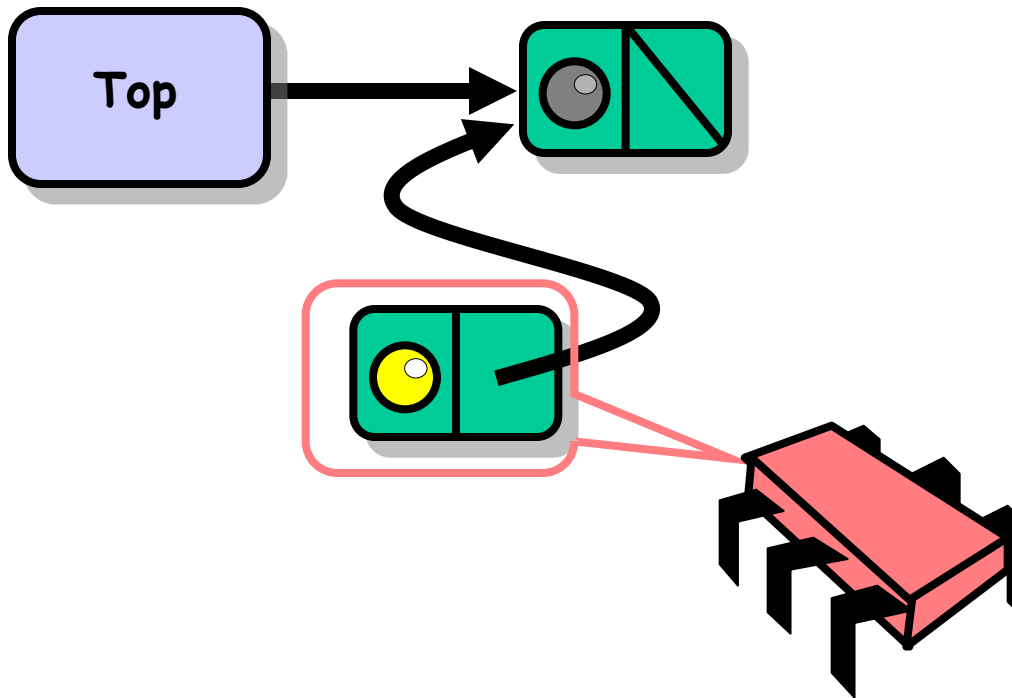
Concurrent Stack

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

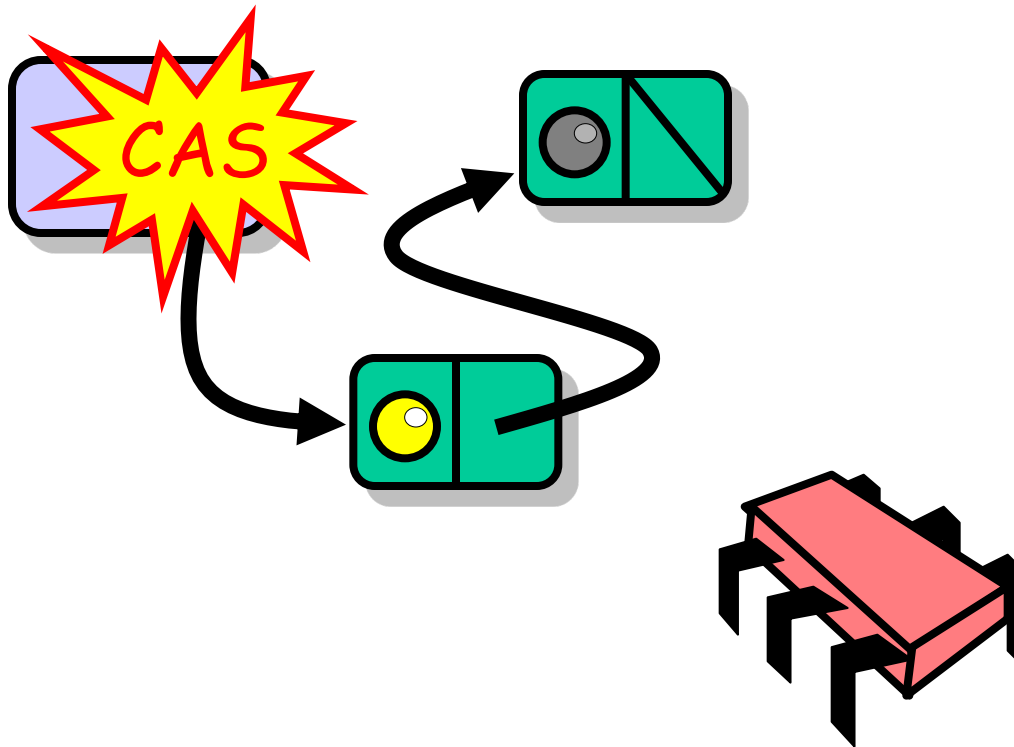
Empty Stack



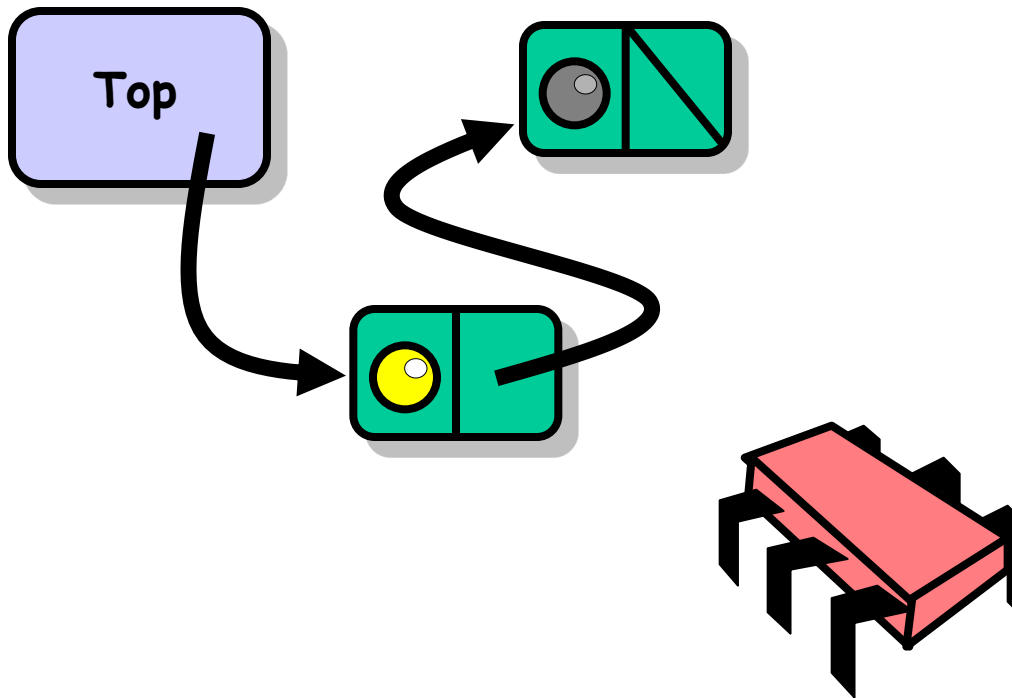
Push



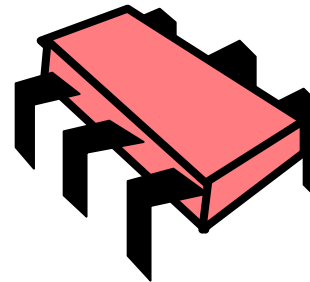
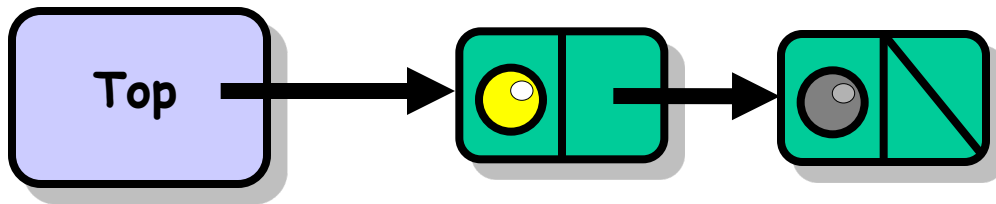
Push



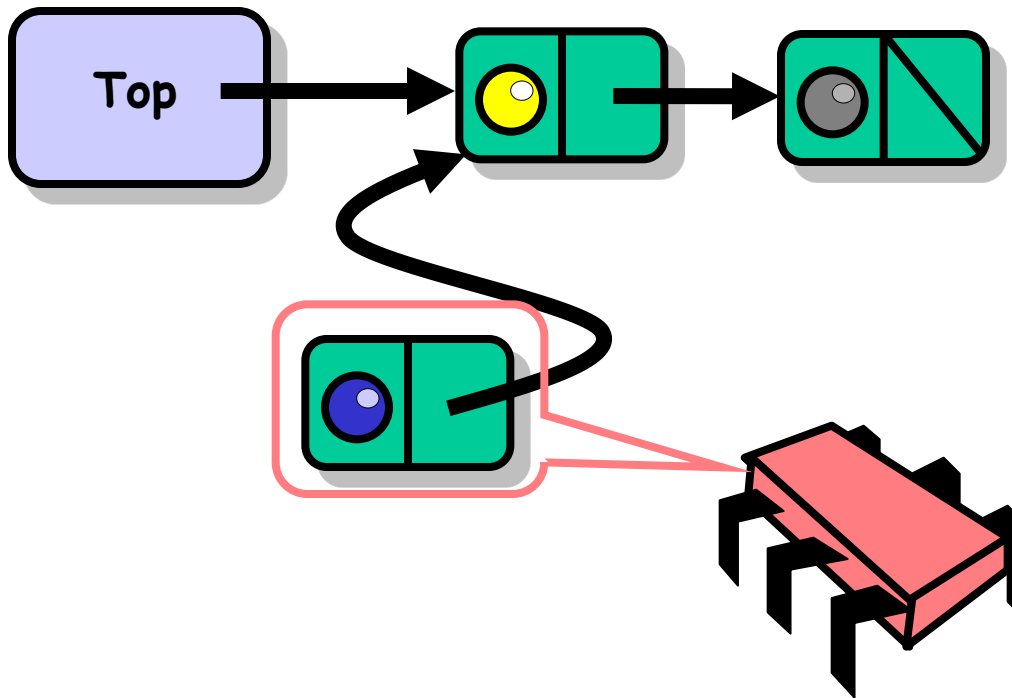
Push



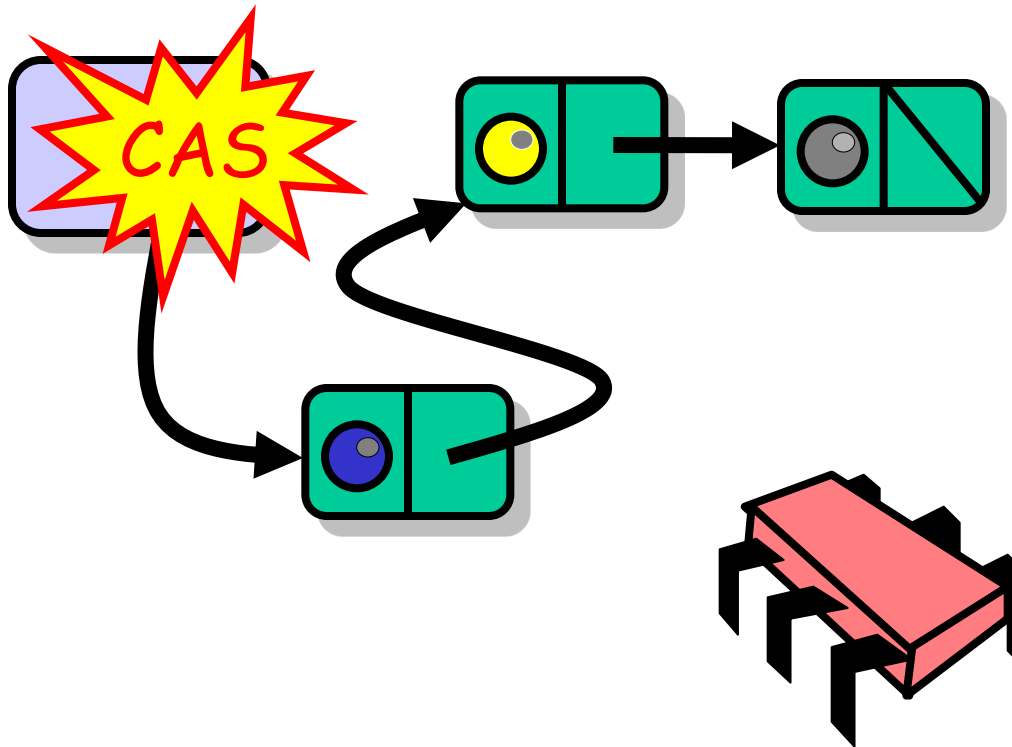
Push



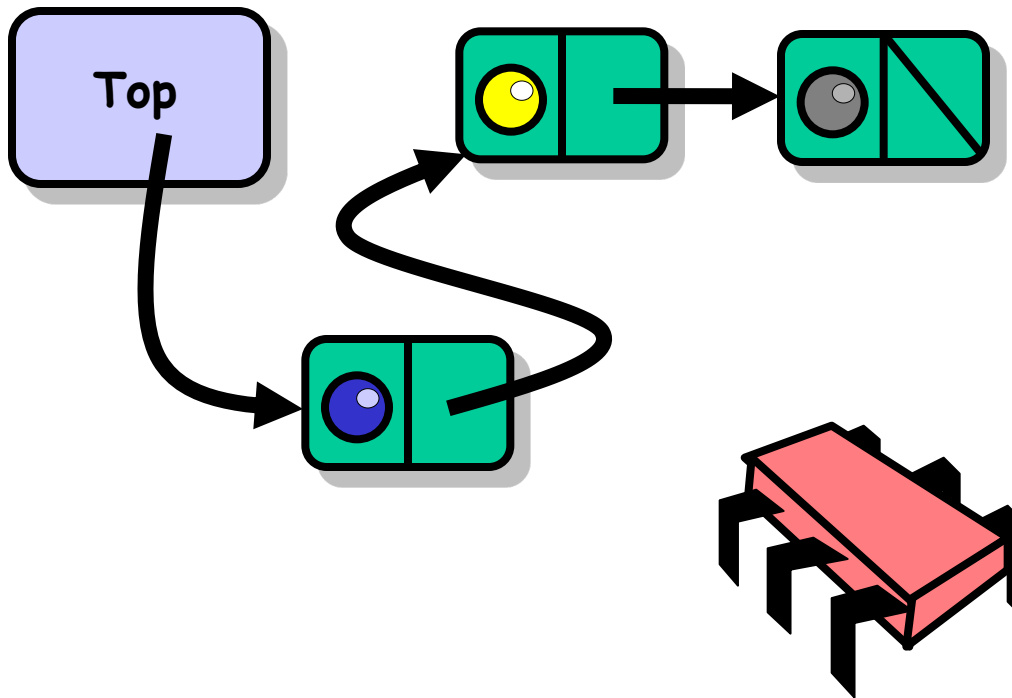
Push



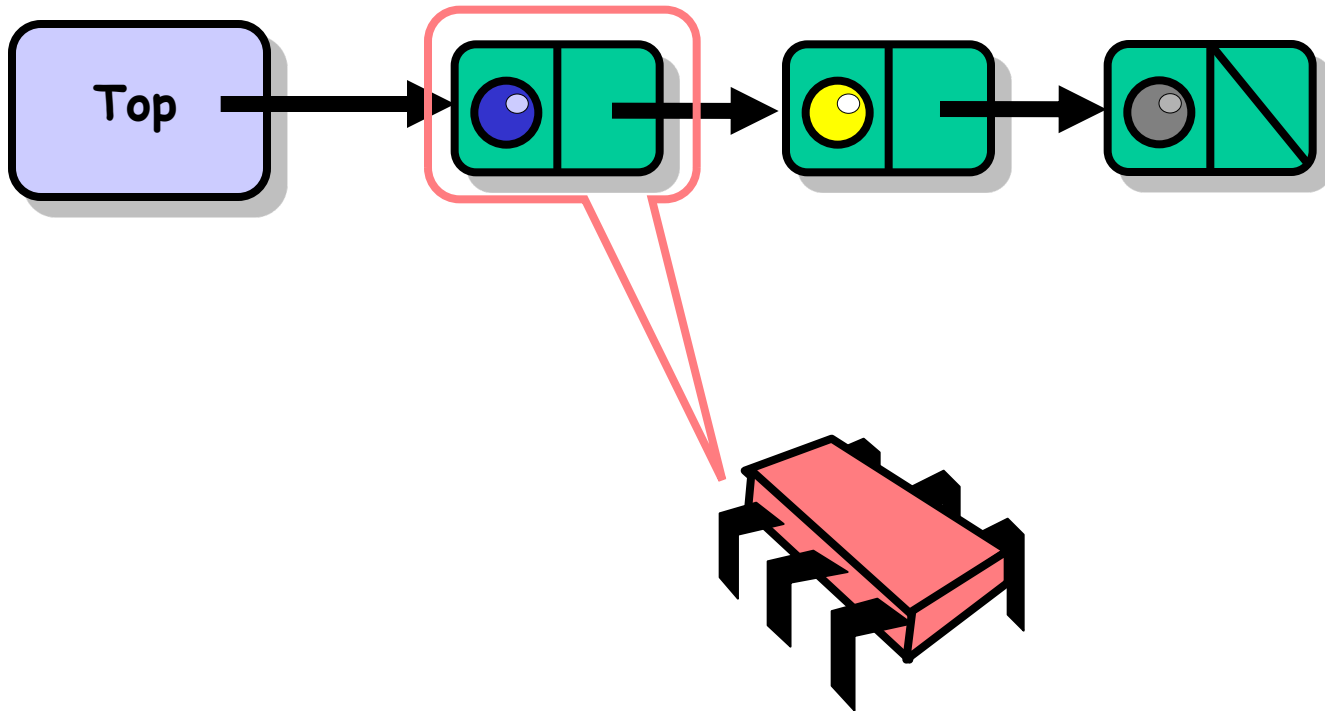
Push



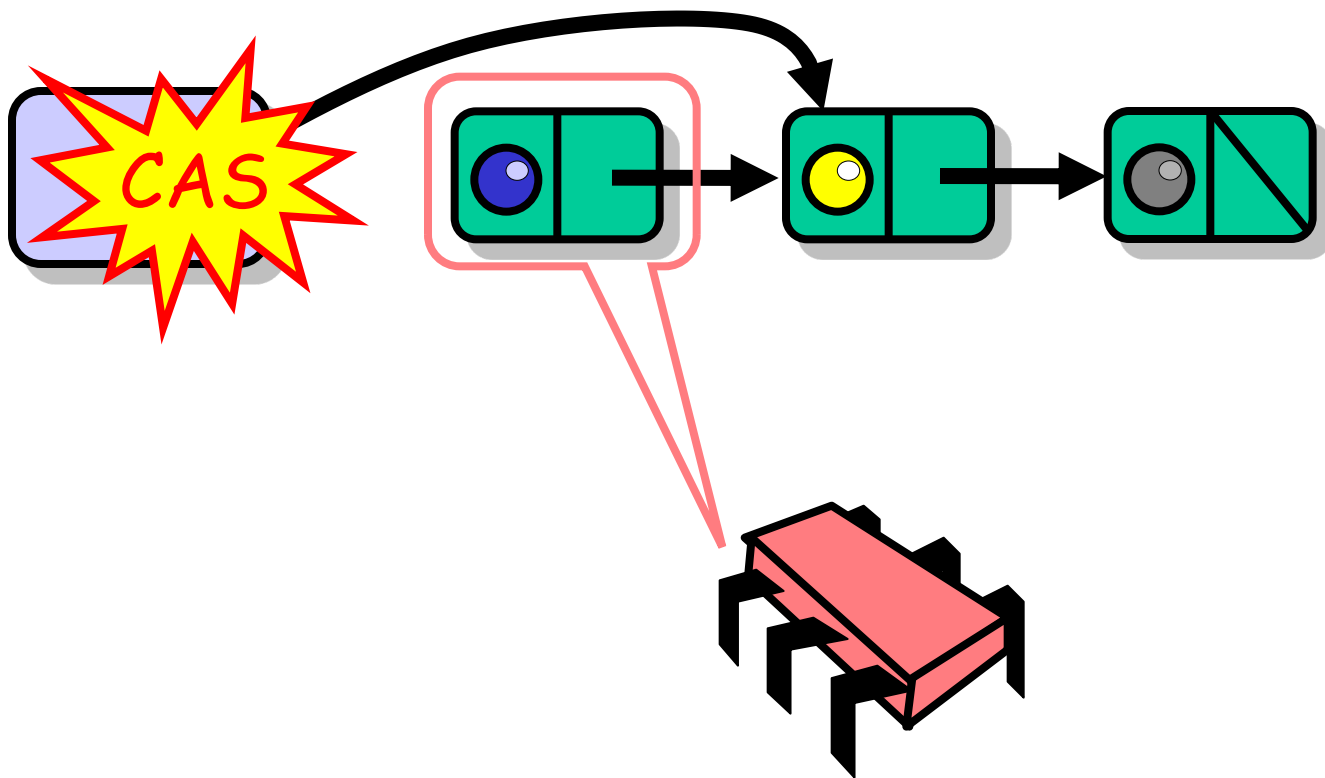
Push



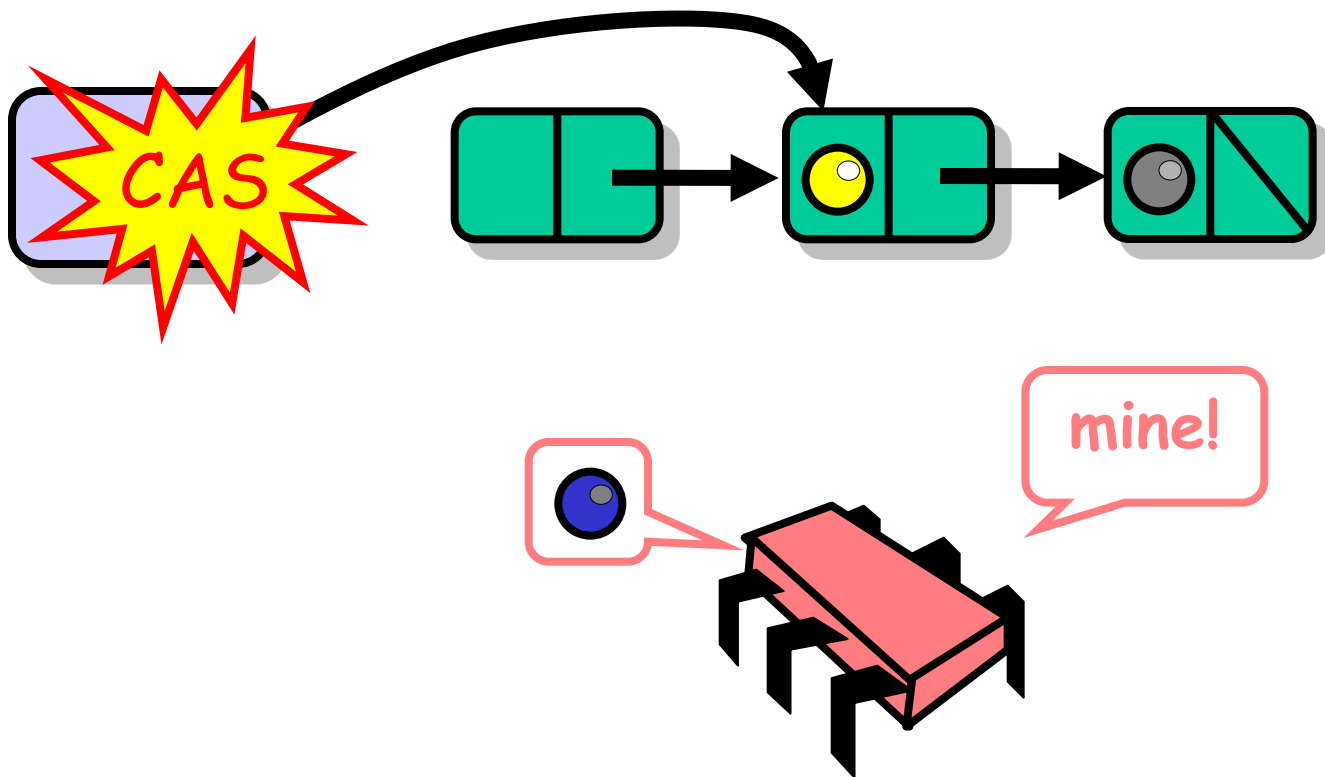
Pop



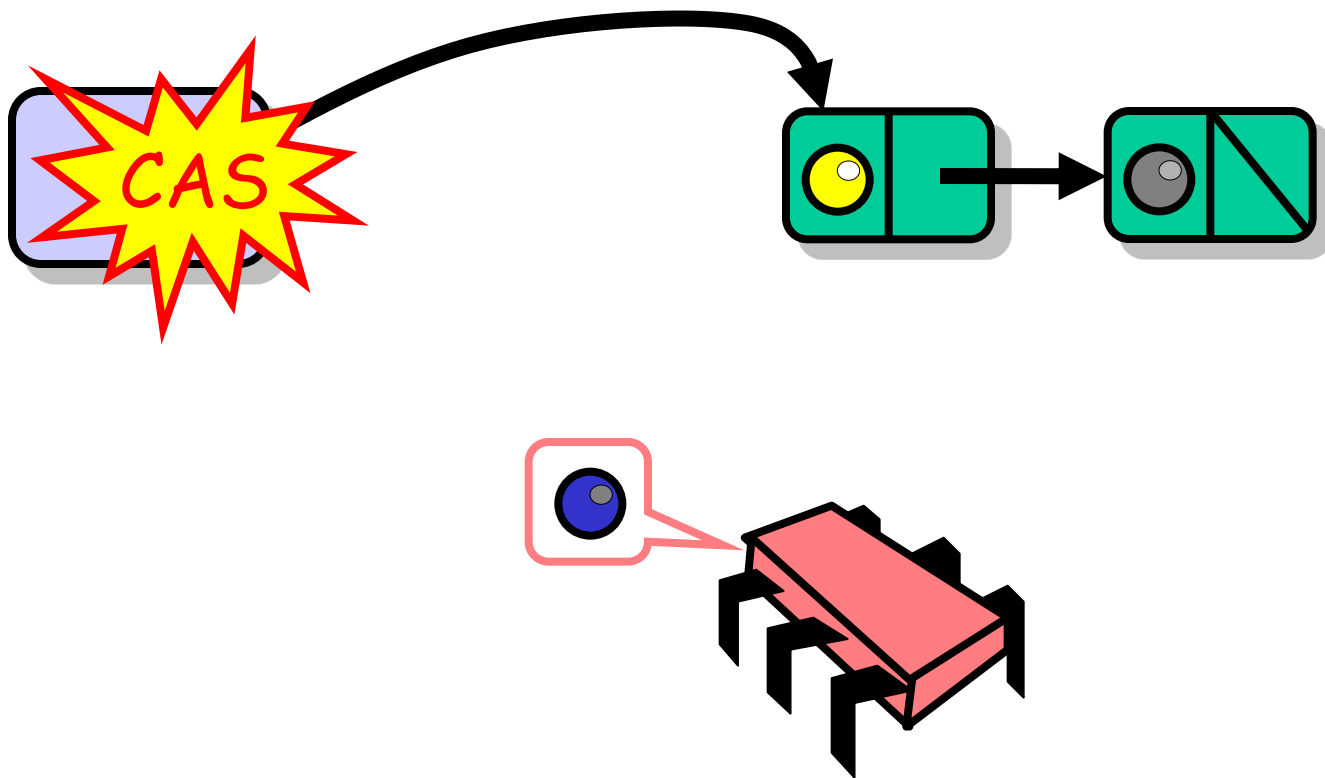
Pop



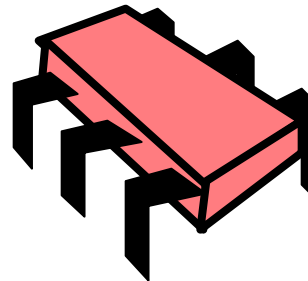
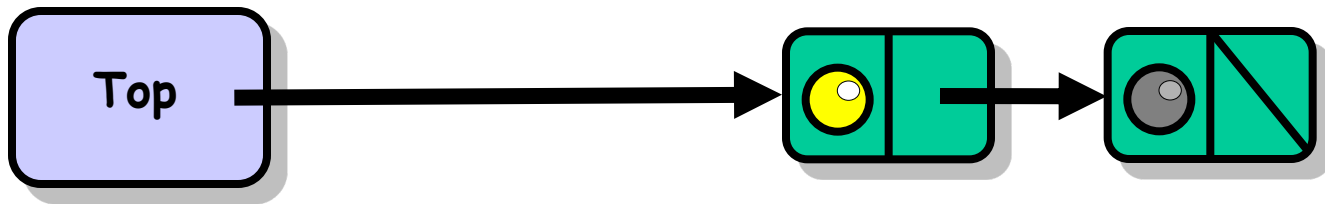
Pop



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();  
        }  
    }  
}
```

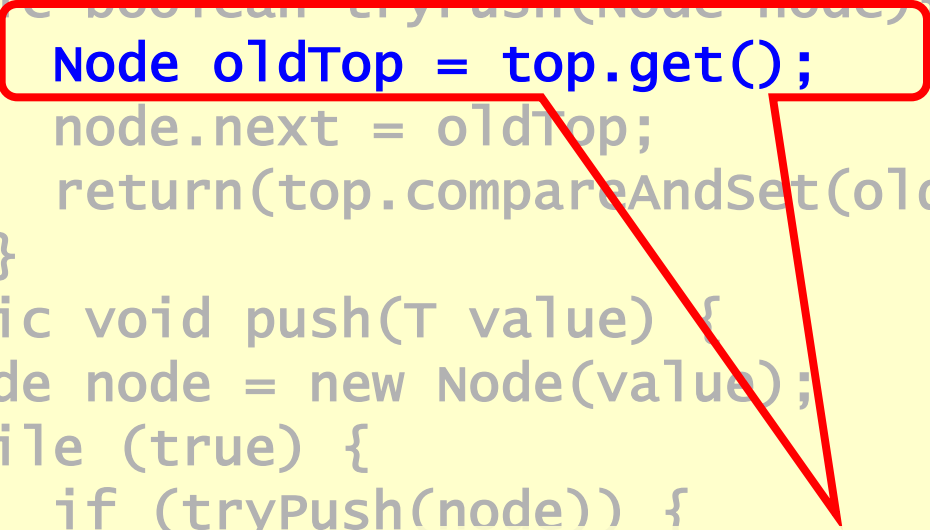
Lock-free Stack

```
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

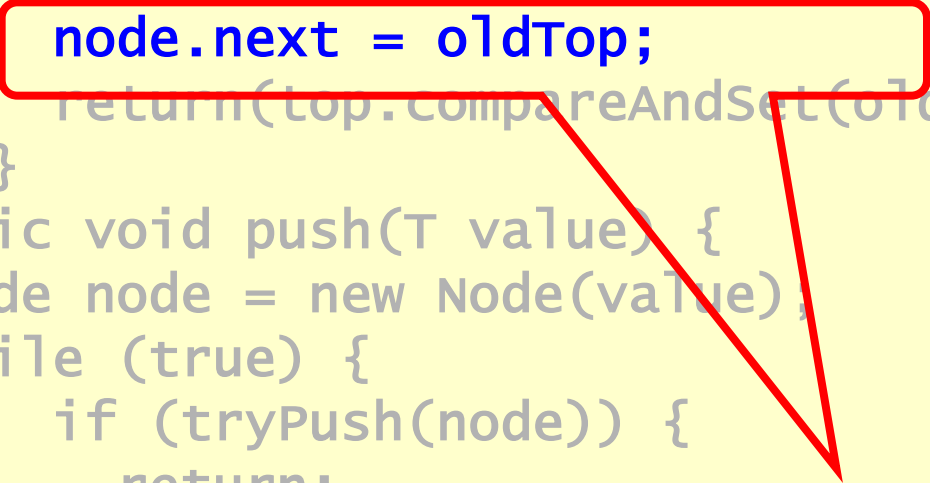
```
public class LockFreeStack {
    private AtomicReference top = new
AtomicReference(null);
    public boolean tryPush(Node node){
        Node oldTop = top.get();
        node.next = oldTop;
        return(top.compareAndSet(oldTop, node))
    }
    public void push(T value) {
        Node node = new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



Read top value

Lock-free Stack

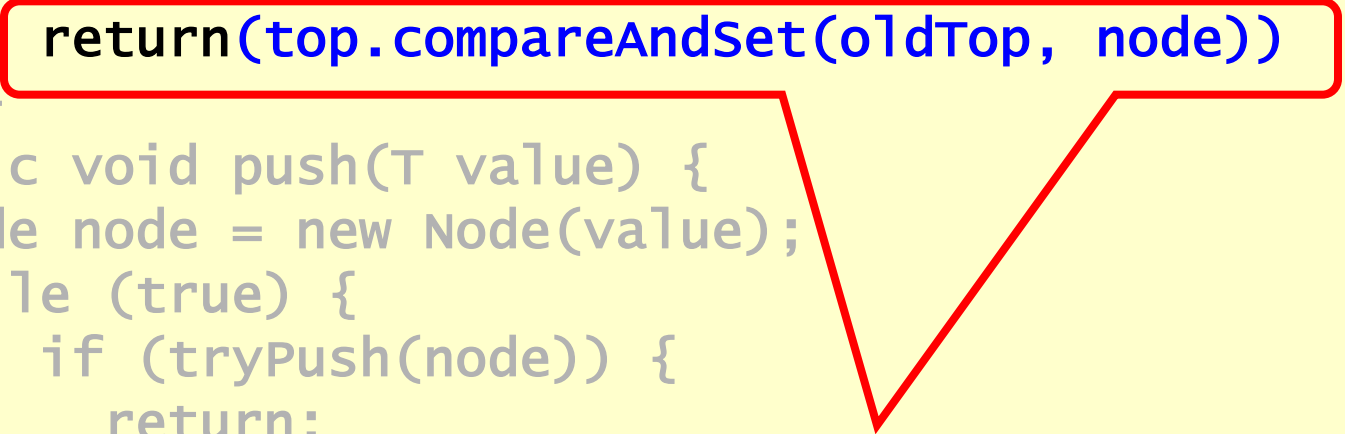
```
public class LockFreeStack {
    private AtomicReference top = new
AtomicReference(null);
    public boolean tryPush(Node node){
        Node oldTop = top.get();
        node.next = oldTop;
        return(top.compareAndSet(oldTop, node))
    }
    public void push(T value) {
        Node node = new Node(value);
        while (true) {
            if (tryPush(node)) {
                return;
            }
        }
    }
}
```



current top will be new node's successor

Lock-free Stack

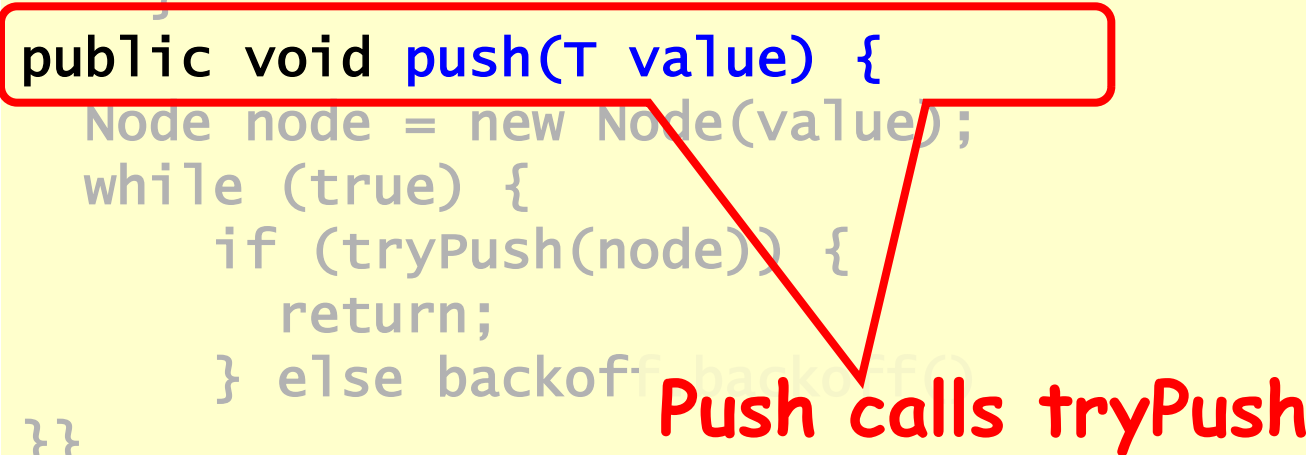
```
public class LockFreeStack {  
    private AtomicReference top = new  
AtomicReference(null);  
    public boolean tryPush(Node node){  
        Node oldTop = top.get();  
        node.next = oldTop;  
        return(top.compareAndSet(oldTop, node))  
    }  
    public void push(T value) {  
        Node node = new Node(value);  
        while (true) {  
            if (tryPush(node)) {  
                return;  
            }  
        }  
    }  
}
```



Try to swing top, return success or failure

Lock-free Stack

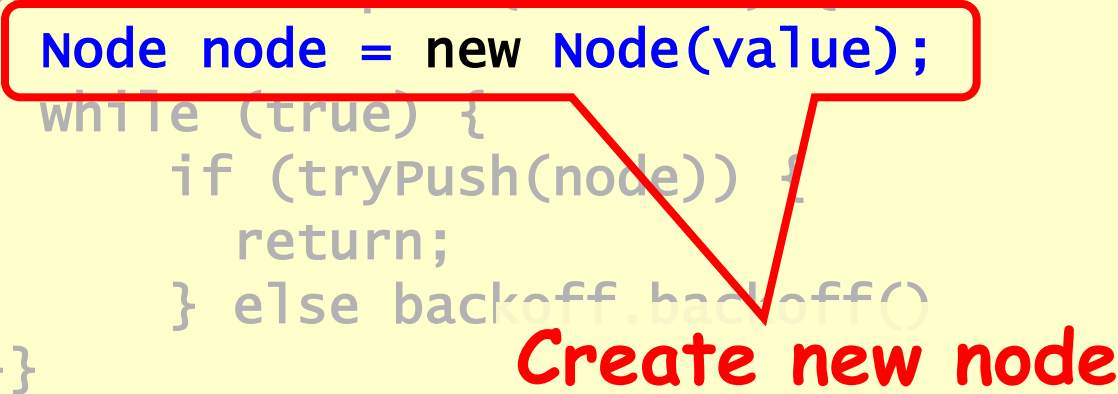
```
public class LockFreeStack {
    private AtomicReference top = new
AtomicReference(null);
    public boolean tryPush(Node node){
        Node oldTop = top.get();
        node.next = oldTop;
        return(top.compareAndSet(oldTop, node))
    }
    public void push(T value) {
        Node node = new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



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

**Makes scheduling benevolent so
Method is effectively wait-free**

**If tryPush() fails,
back off before retrying**

```
public class LockFreeStack {  
    private AtomicReference<Node> top = new AtomicReference<Node>();  
    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

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

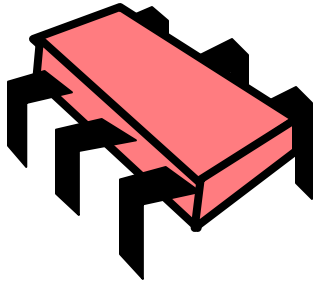
Big Question

- 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

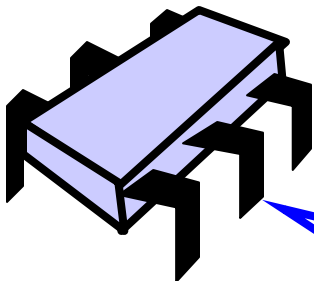


Push(●)

linearizable stack



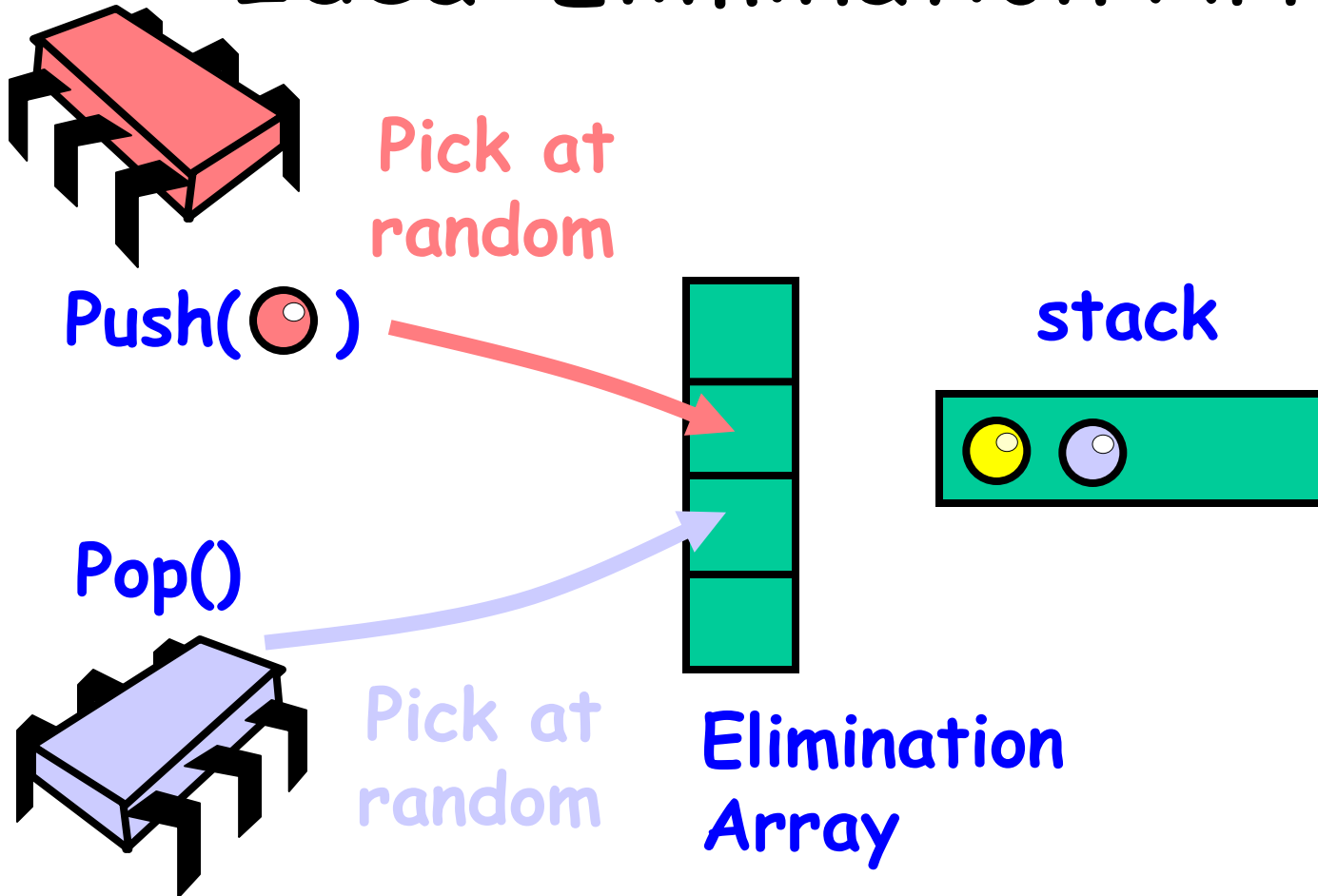
Pop()



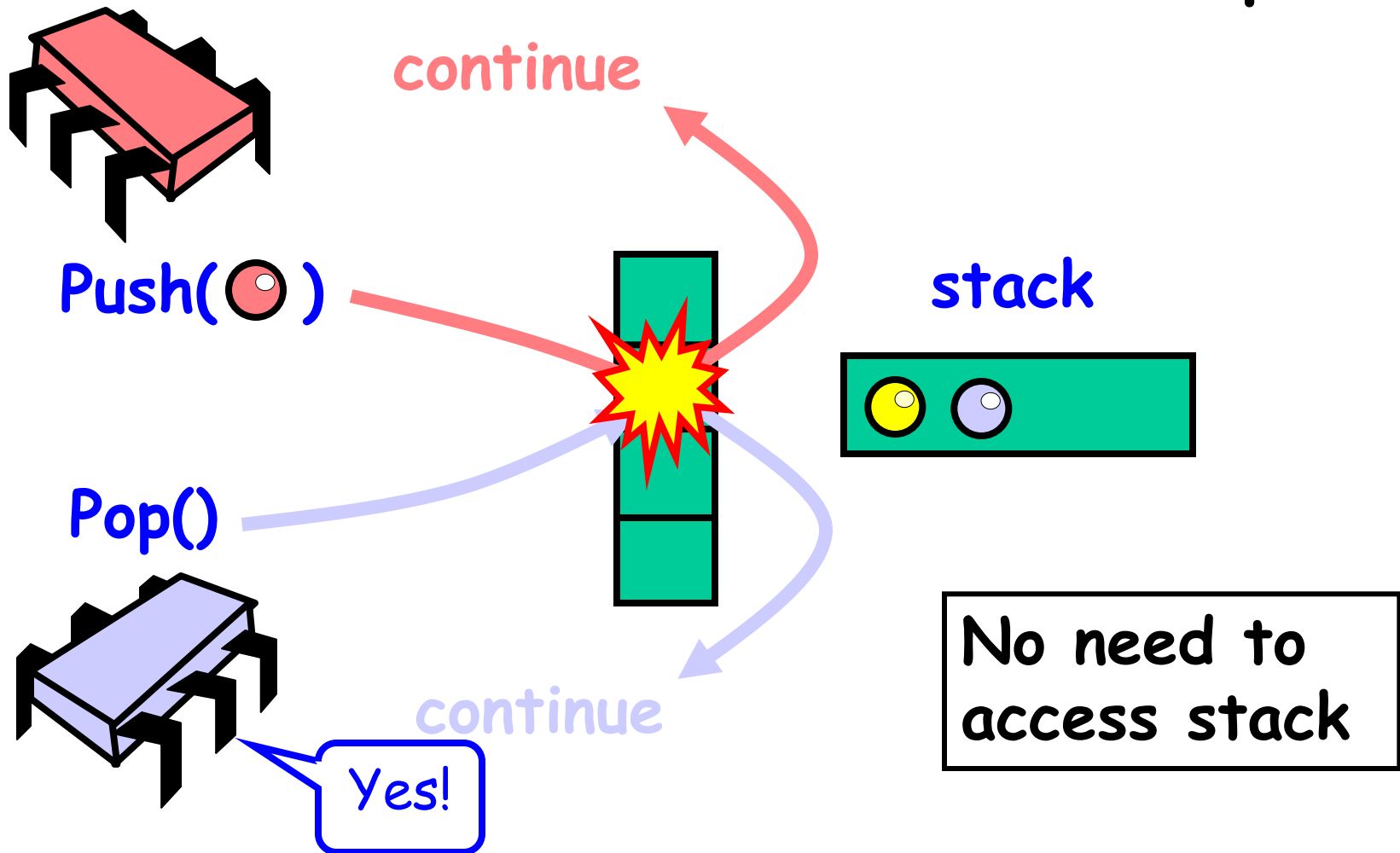
Yes!

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

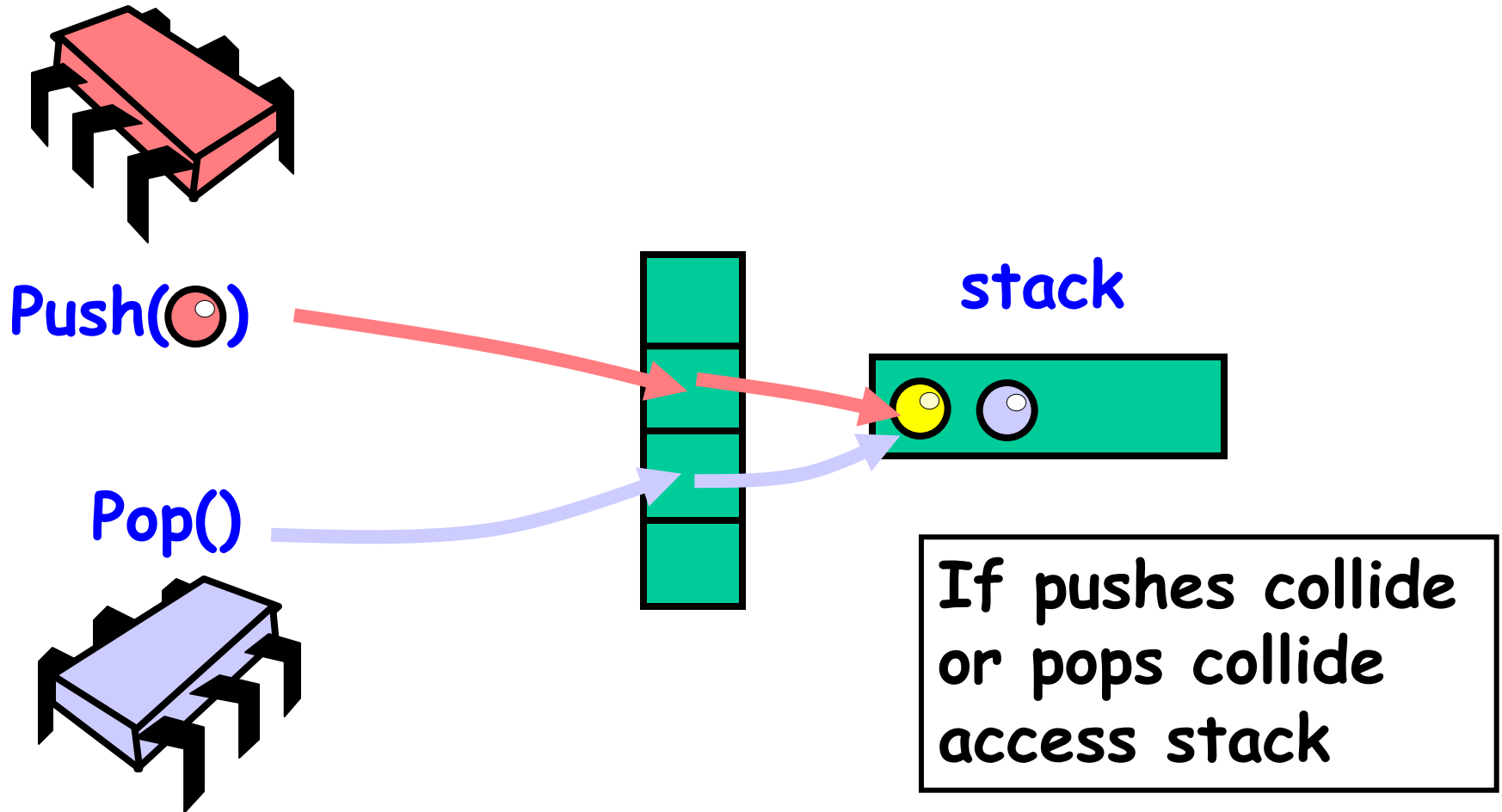
Idea: Elimination Array



Push Collides With Pop



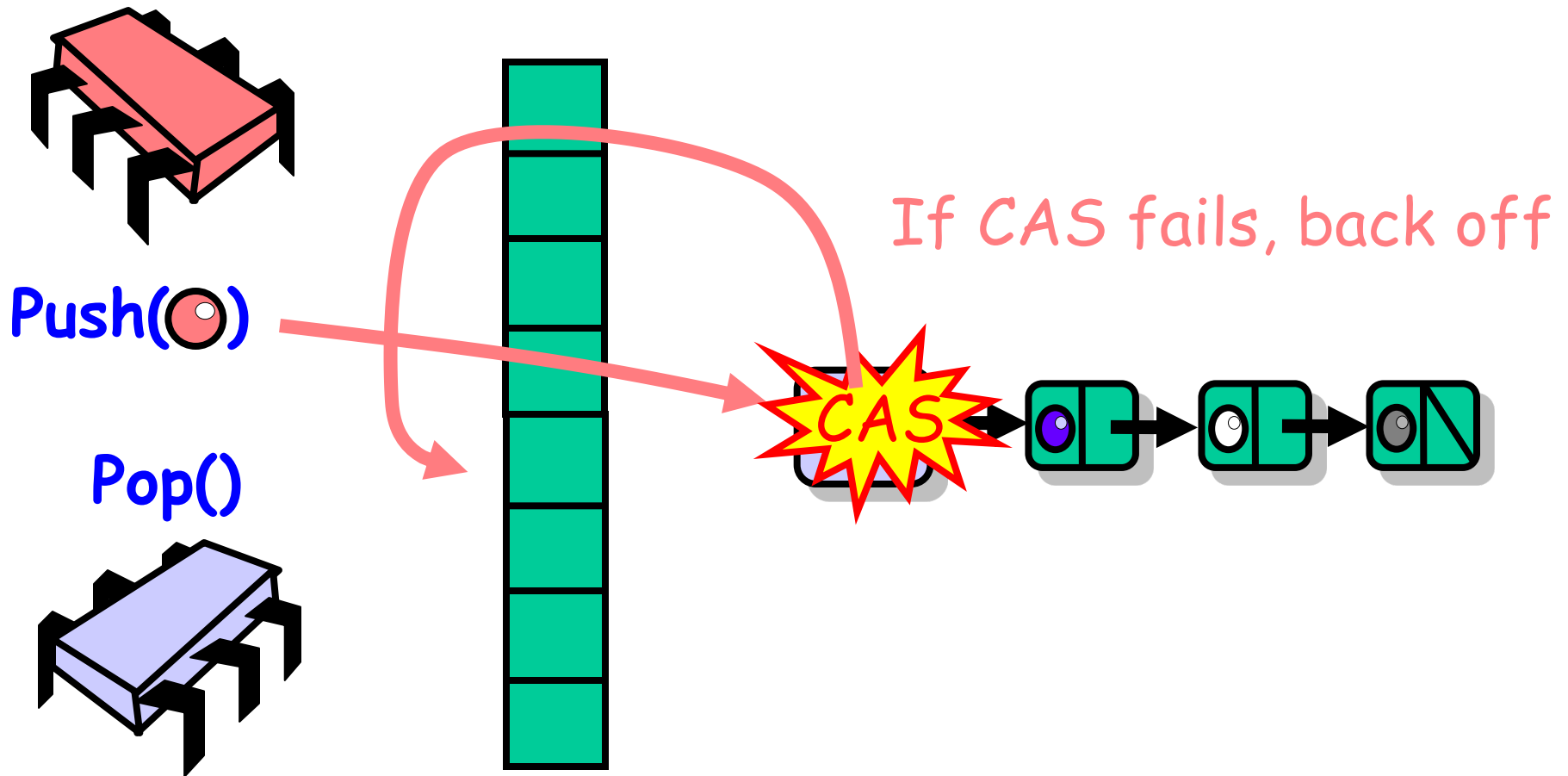
No Collision



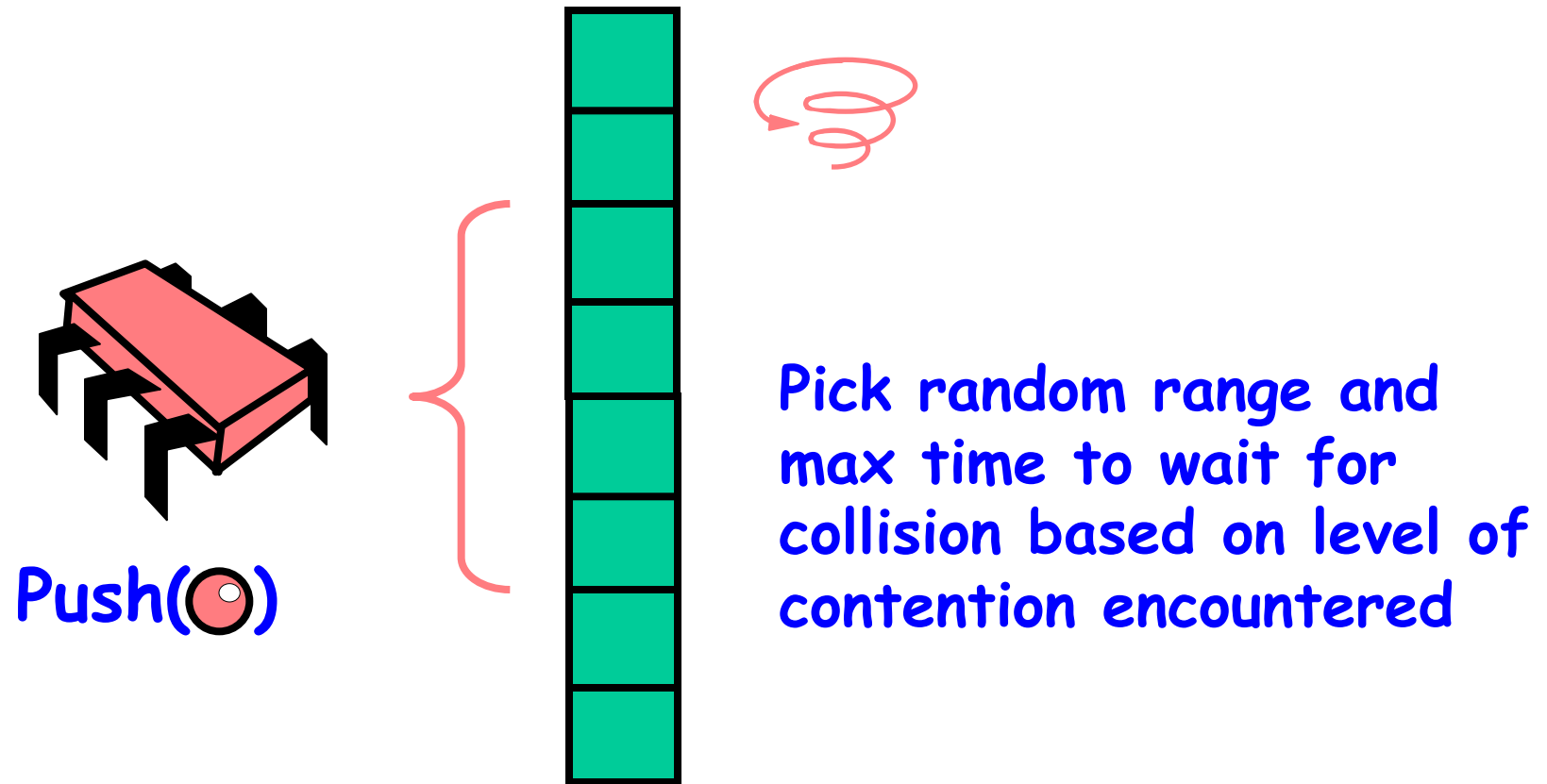
Elimination-Backoff Stack

- 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

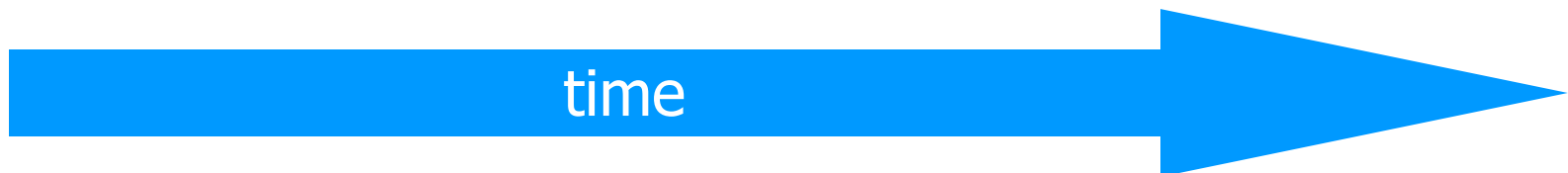
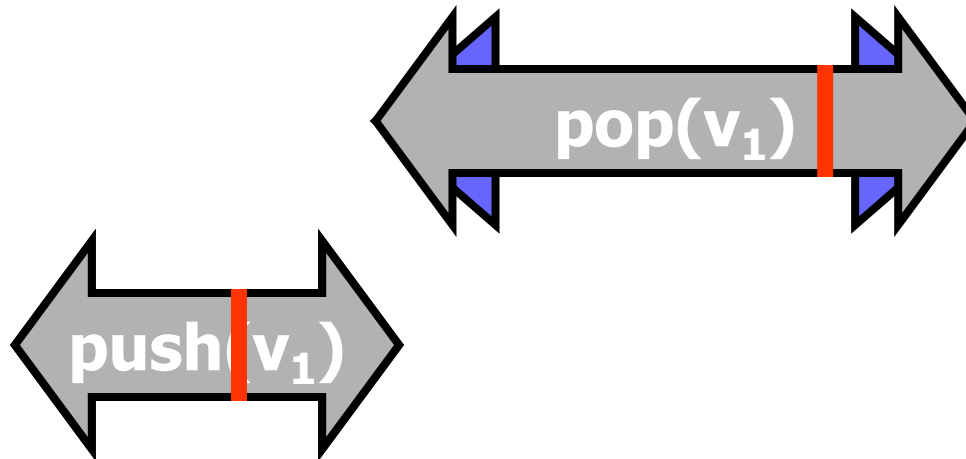


Linearizability

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

Un-Eliminated Linearizability

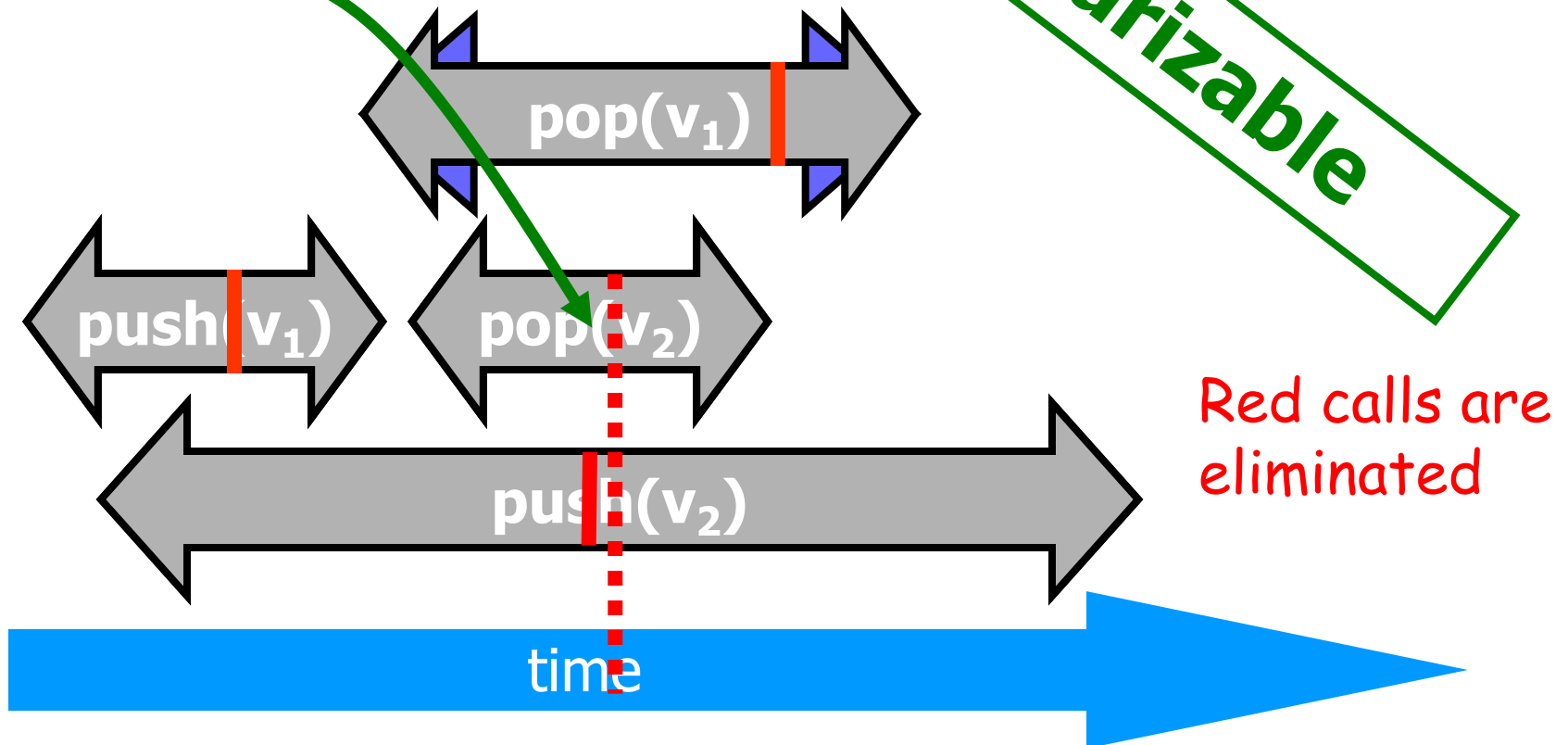
linearizable



Eliminated Linearizability

Collision
Point

linearizable



Backoff Has Dual Effect

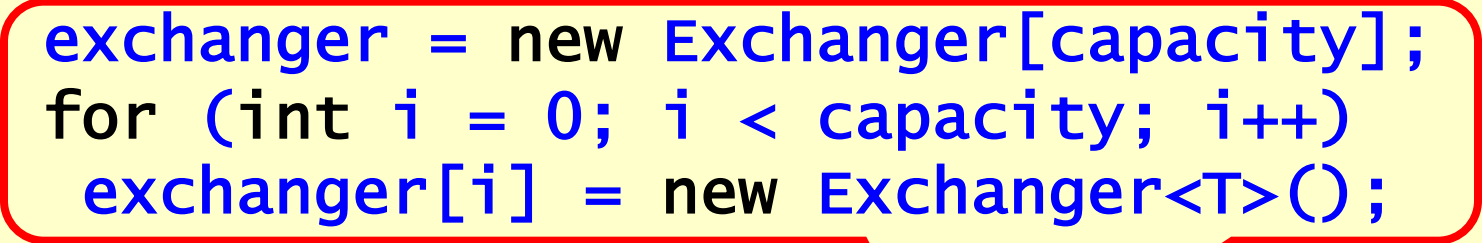
- Elimination introduces parallelism
- Backoff onto array cuts contention on lock-free stack
- Elimination in array cuts down total number of threads ever accessing lock-free stack

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

A Lock-Free Exchanger

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

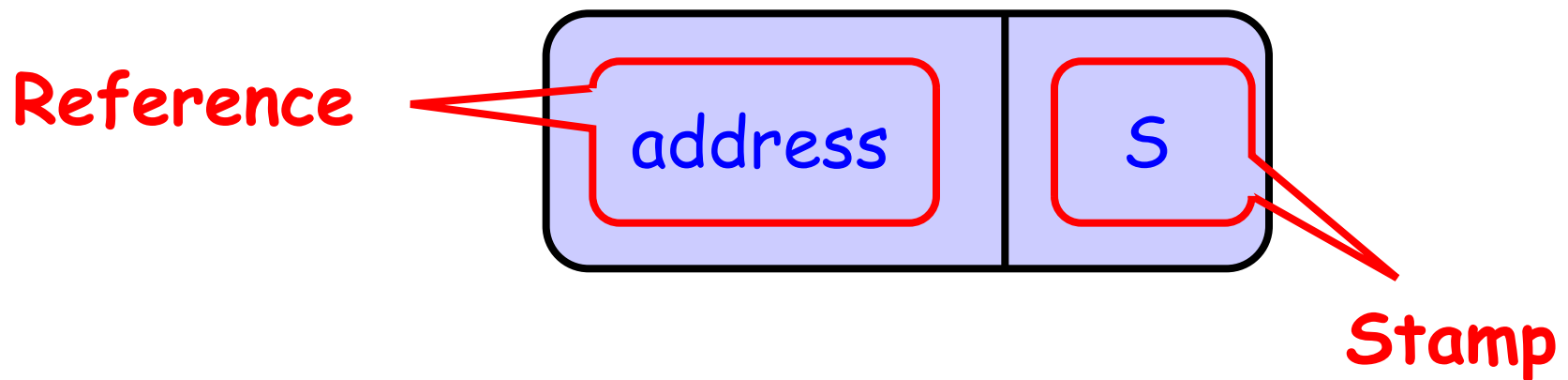
A Lock-Free Exchanger

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

Atomically modifiable
reference + time stamp

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

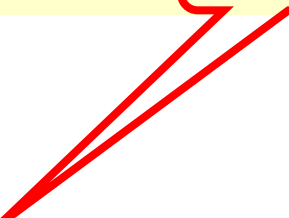
Returns stamp at
array index 0!

Exchanger Status

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

Exchanger Status

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



Nothing yet

Exchange Status

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



Nothing yet

One thread is waiting
for rendez-vous

Exchange Status

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

Nothing yet

One thread is waiting
for rendez-vous

Other threads busy
with rendez-vous

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 & timeout

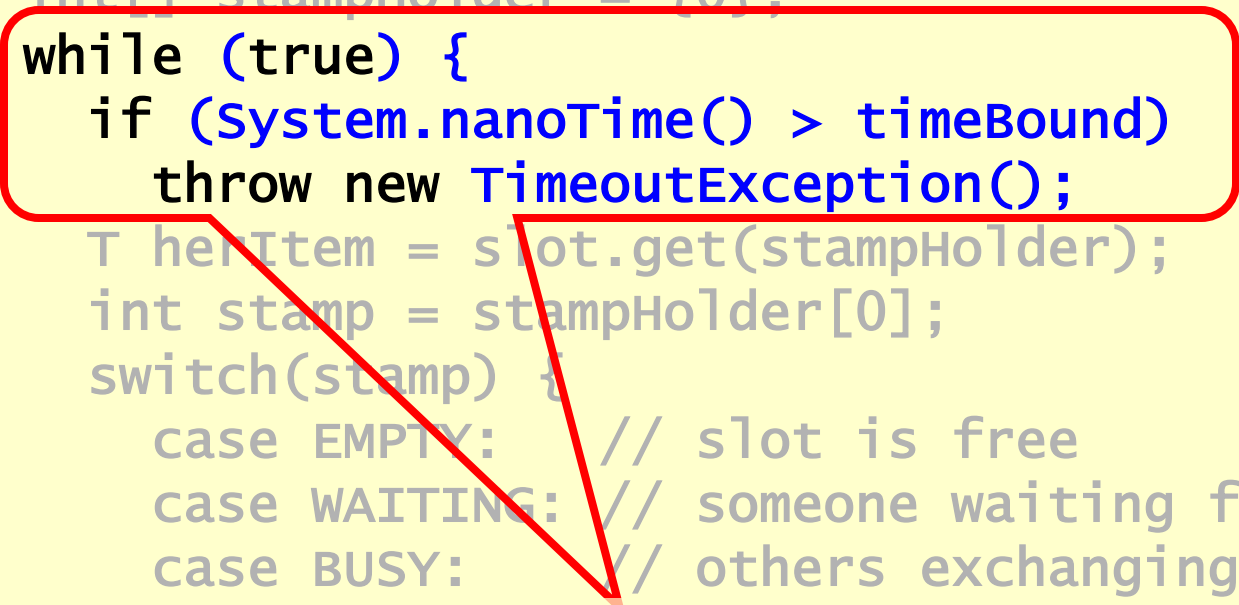
The Exchange

```
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 to hold timestamp

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 timestamp

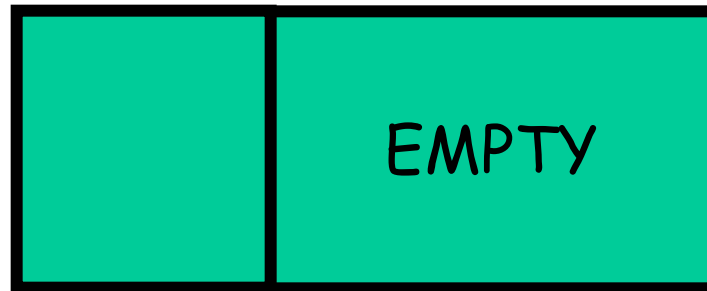
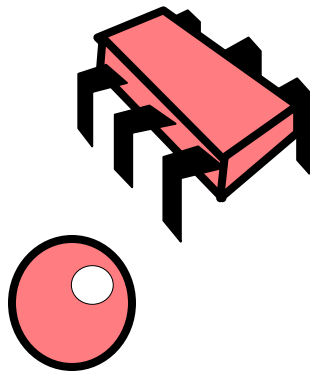
The Exchange

```
public T Exchange(T myItem, long nanos) throws  
TimeoutException {  
    long timeBound = System.nanoTime() + nanos;
```

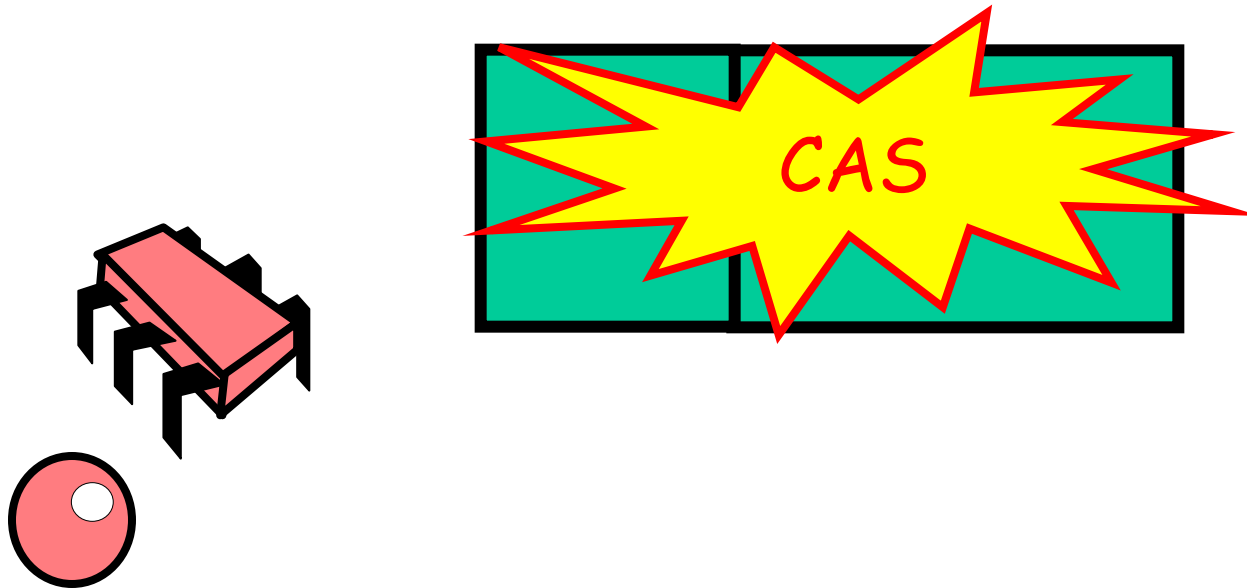
Exchanger slot has three states

```
    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  
        }  
    }  
}
```

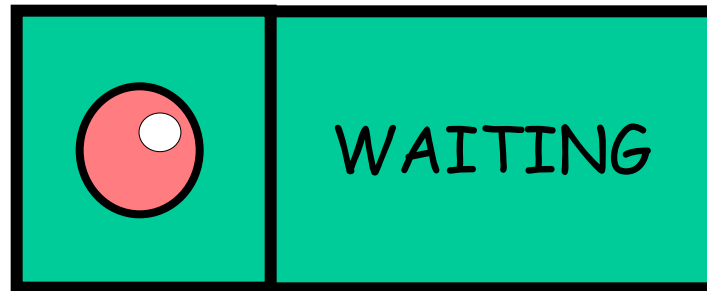
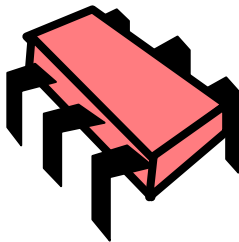
Lock-free Exchanger



Lock-free Exchanger

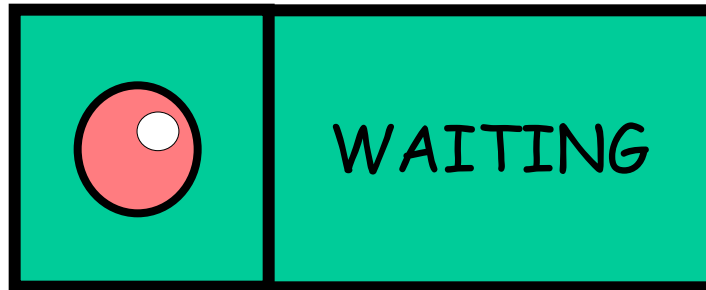
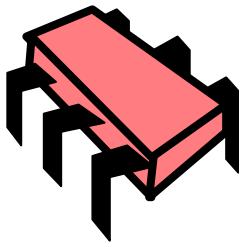


Lock-free Exchanger



Lock-free Exchanger

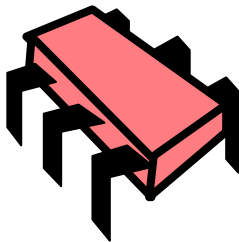
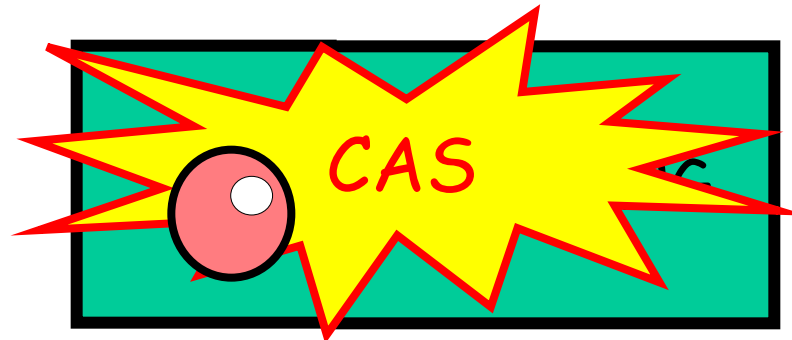
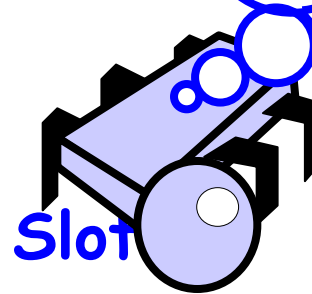
In search of
partner ...



Lock-free Ex

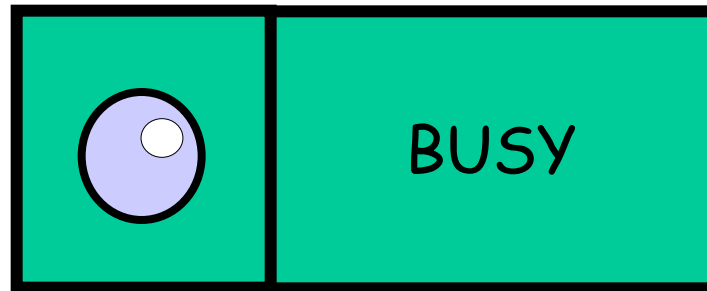
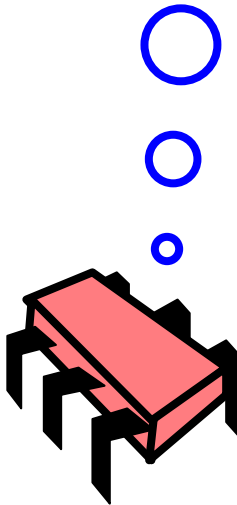
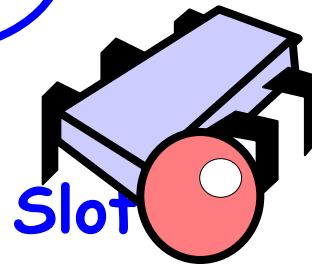
Still waiting ...

Try to
exchange item
and set state
to BUSY



Lock-free Exchanger

Partner showed
up, take item and
reset to EMPTY



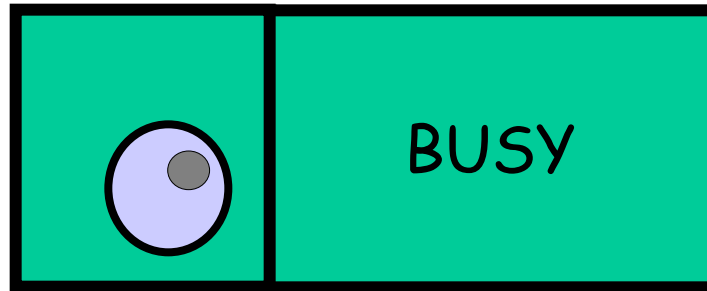
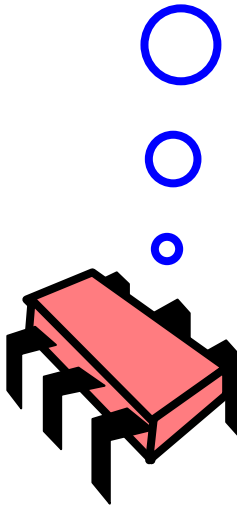
item

stamp/state

Lock-free Exchanger

Partner showed
up, take item and
reset to EMPTY

Slot



item

stamp/state

Exchanger State EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(herItem, myItem, EMPTY,
WAITING)) {
        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;
```

Exchanger State EMPTY

```
case EMPTY: // slot is free
```

```
    if (slot.compareAndSet(herItem, myItem, EMPTY,  
WAITING)) {
```

```
        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;
```

**Slot is free, try to insert
myItem and change state
to WAITING**

Exchanger State EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(herItem, myItem, EMPTY,
        WAITING)) {
        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;
```

Loop while still time left to attempt exchange

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;
```

**Get item and stamp in slot
and check if state changed
to BUSY**

Exchanger State EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(herItem, myItem, EMPTY,
WAITING)) {
        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 successful reset slot
state to EMPTY**

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;
```

and return item found in slot

Exchanger State EMPTY

Otherwise we ran out of time, try to reset state to EMPTY, if successful time out

```
if (slot.compareAndSet(myItem, null, WAITING, EMPTY)){throw new TimeoutException();
```

```
    } else {  
        herItem = slot.get(stampHolder);  
        slot.set(null, EMPTY);  
        return herItem;  
    }  
} break;
```

Exchanger State EMPTY

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

```
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;
```

Exchanger State EMPTY

Set slot to EMPTY with new timestamp and return item found

```
case EMPTY: // slot is free
    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;
```

Exchanger State EMPTY

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

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

States WAITING and BUSY

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

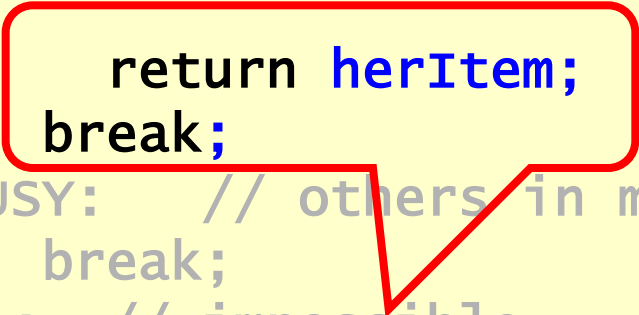
States WAITING and BUSY

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

**state WAITING means
someone is waiting for an
exchange, so attempt to
CAS my item in and change
state to BUSY**

States WAITING and BUSY

```
case WAITING:    // someone waiting for me
    if (slot.compareAndSet(herItem, myItem,
                           WAITING, BUSY))
```



```
    return herItem;
    break;
```

```
case BUSY:       // others in middle of exchanging
    break;
default:         // impossible
    break;
```

If successful return her item, state is now BUSY, otherwise someone else took her item so try again from start

States WAITING and BUSY

```
case WAITING:    // someone waiting for me
    if (slot.compareAndSet(herItem, myItem,
        WAITING, BUSY))
        return herItem;
    break;
```

```
case BUSY:      // others in middle of exchanging
    break;
```

```
default:        // impossible
    break;
}
}
}
}
```

**If state is BUSY then
some other threads are
using slot to exchange 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

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 a value
and a range (duration to wait is not
dynamic)

Elimination Array

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

Pick a random array entry

Elimination Array

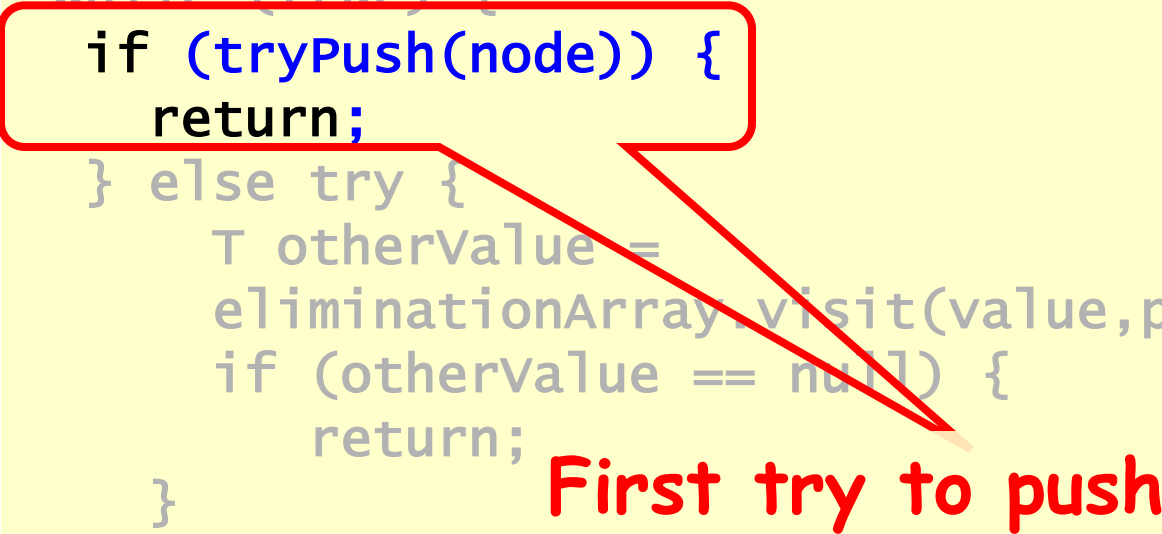
```
public class EliminationArray {  
    ...  
    public T Exchange value or time out  
    TimeoutException {  
        int slot = random.nextInt(Range);  
        int nanodur = convertToNanos(duration, timeUnit);  
        return (exchanger[slot].exchange(value, nanodur)  
    }  
}
```

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

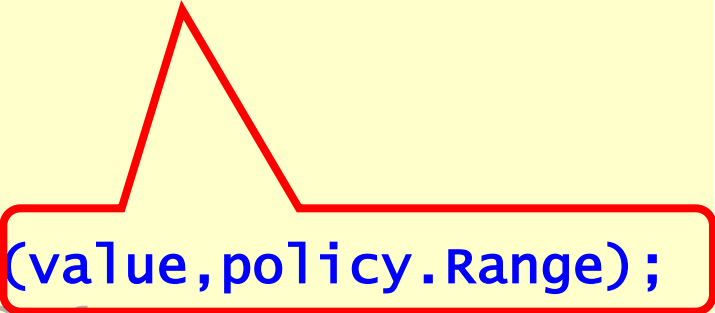
Elimination Stack Push

```
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 failed back-off to try to eliminate

Elimination Stack Push

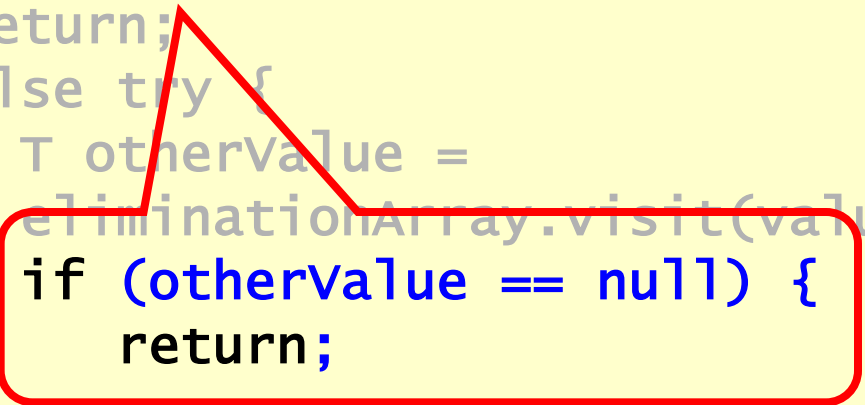
```
public void push(T value) {  
    ...  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value, policy.Range);  
            if (otherValue == null) {  
                return;  
 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



Elimination Stack Push

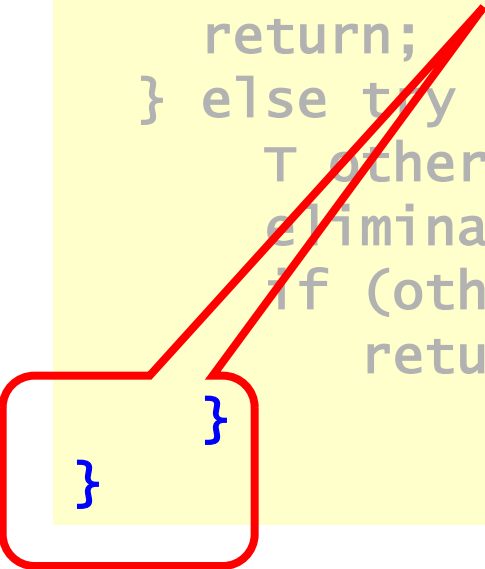
```
public void push(T value) {  
    ...  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value, policy.Range);  
            if (otherValue == null) {  
                return;  
 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

**Only a pop has null value
so elimination was successful**



Elimination Stack Push

```
public void push(T value) {  
... Else retry push on lock-free stack  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value, policy.Range);  
            if (otherValue == null) {  
                return;  
            }  
        }  
    }  
}
```



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) {  
        if (otherValue == null) {  
            return otherValue;  
        } else {  
            try {  
                T otherValue =  
                    eliminationArray.visit(null, policy.Range;  
                if (otherValue != null) {  
                    return otherValue;  
                }  
            }  
        }  
    }  
}
```

Same as push, if non-null other thread must have pushed so elimination succeeds

if (otherValue != null) {
 return otherValue;
}

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
- ABA is a real problem, pay attention

This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](https://creativecommons.org/licenses/by-sa/3.0/).

- **You are free:**
 - **to Share** — to copy, distribute and transmit the work
 - **to Remix** — to adapt the work
- **Under the following conditions:**
 - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
 - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
 - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.