08 Concurrent Queues and Stacks

CS 6868: Concurrent Programming

KC Sivaramakrishnan

Spring 2026, IIT Madras

The Five-Fold Path

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

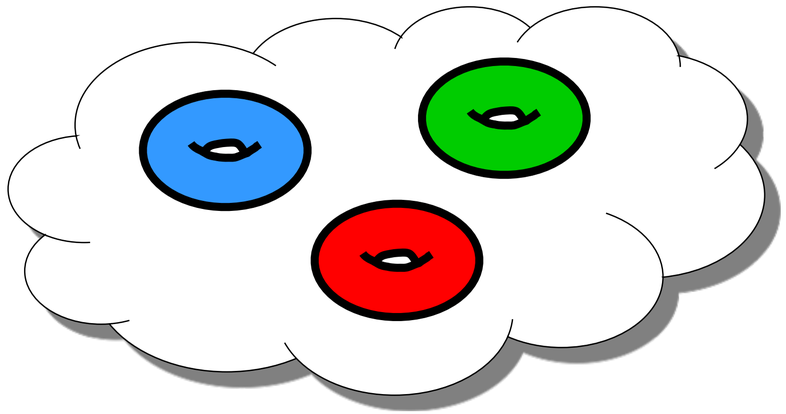
Another Fundamental Problem

- We told you about
  - Sets implemented by linked lists
- **Next:** queues
- **Later:** stacks

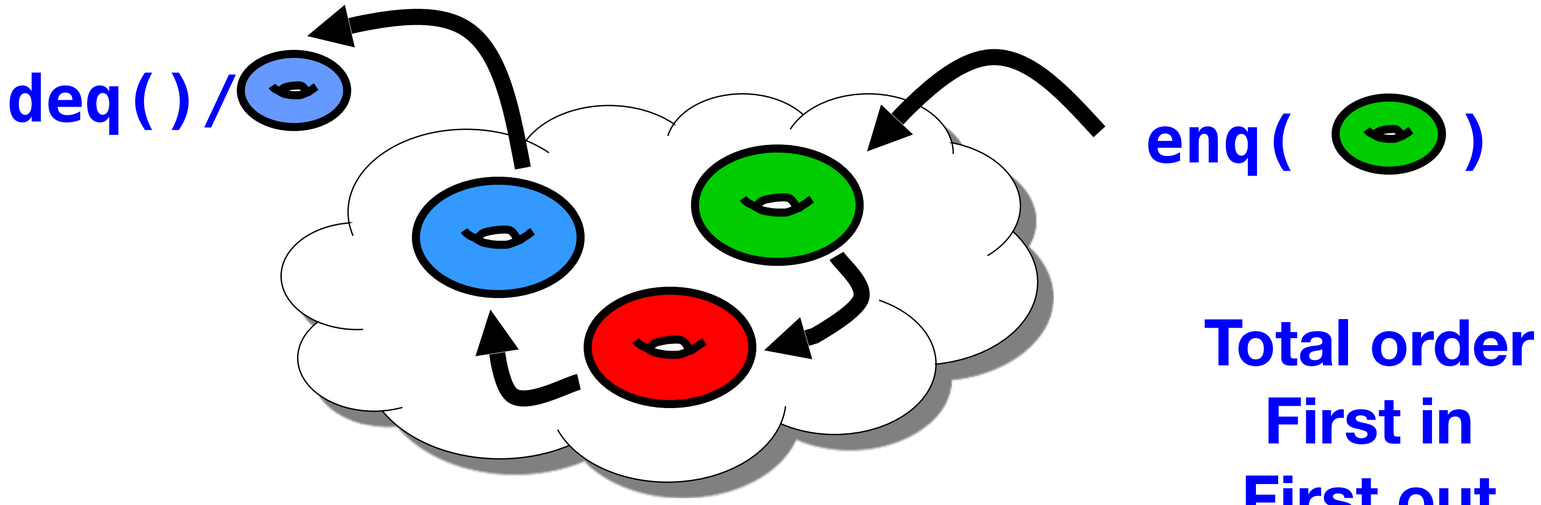
Concurrent Queues

Queues & Stacks

- pool of items

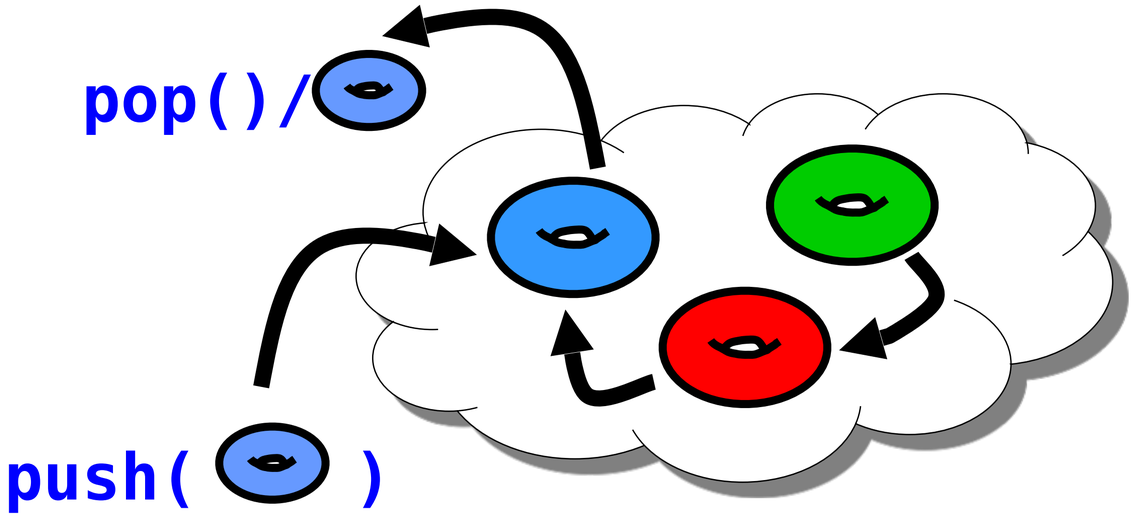


Queues



Total order
First in
First out

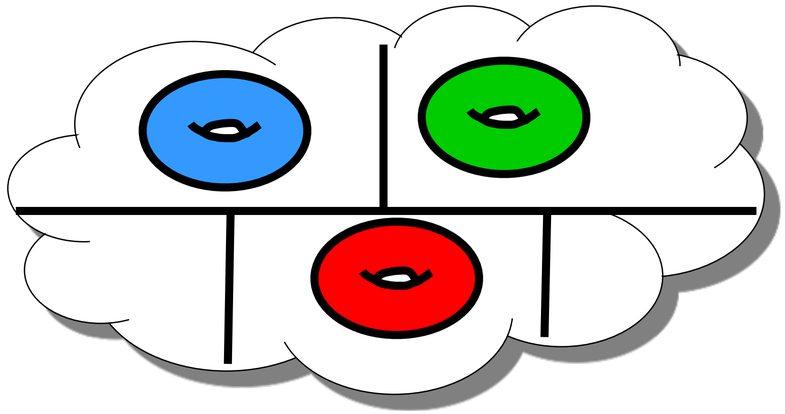
Stacks



Total order
Last in
First out

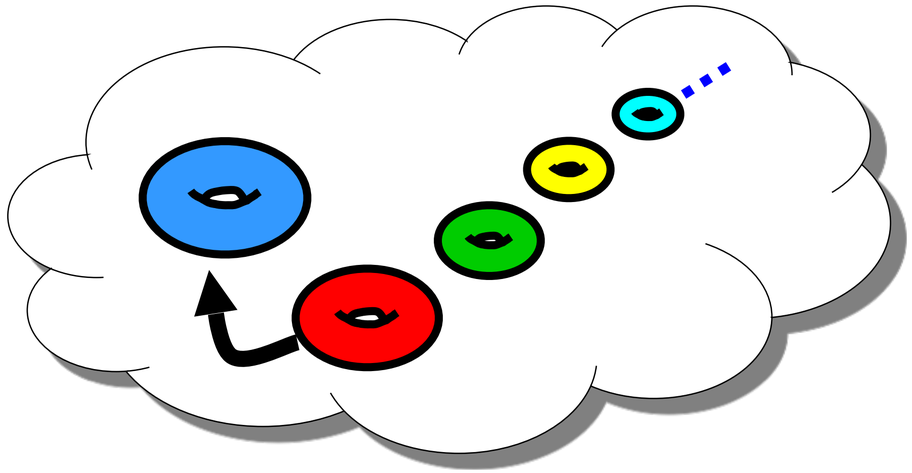
Bounded

- Fixed capacity
- Good when resources an issue

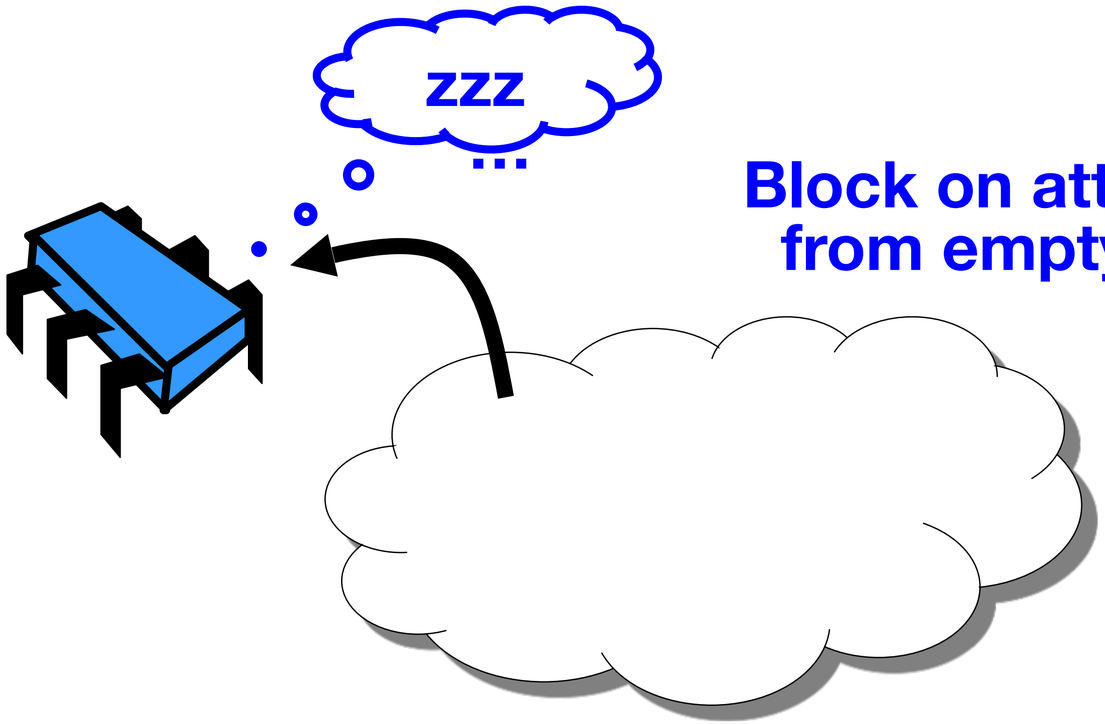


Unbounded

- Unlimited capacity
- Often more convenient

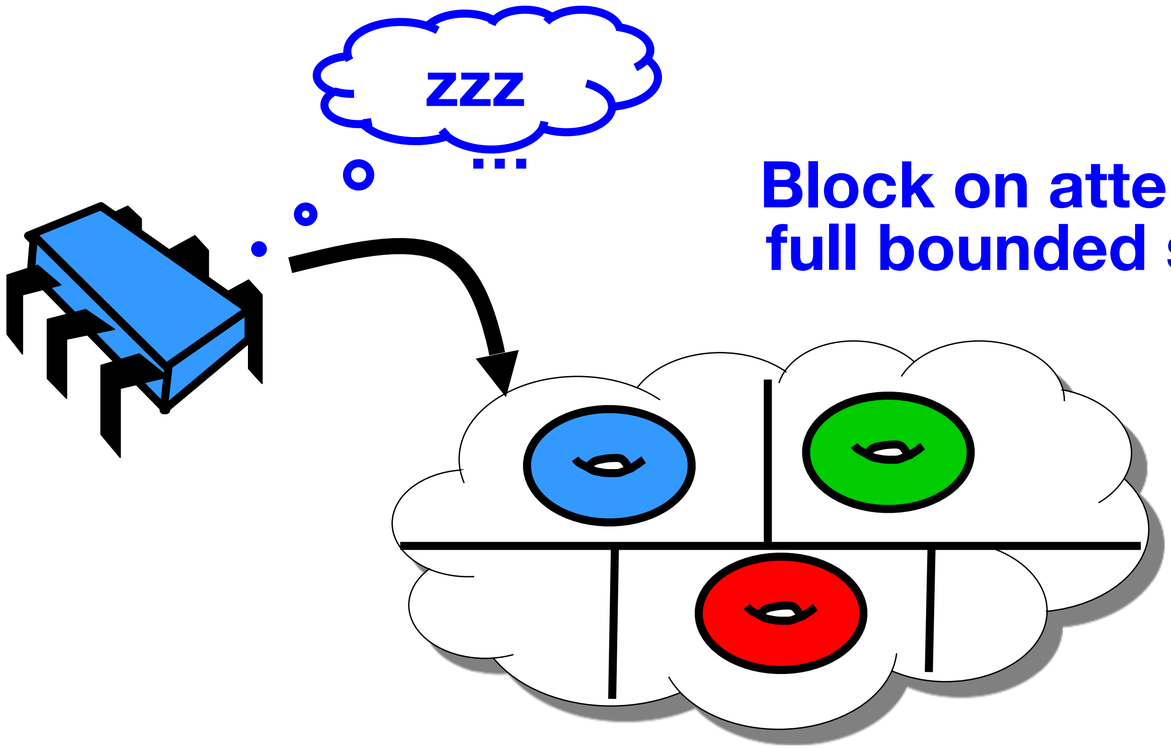


Blocking



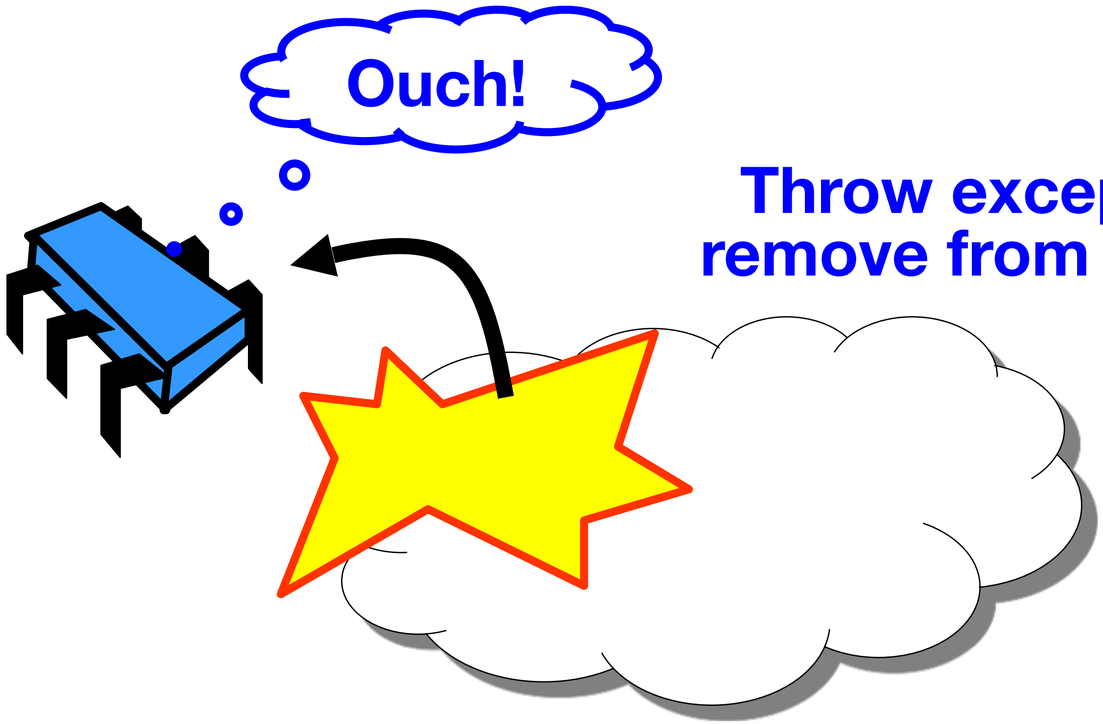
Block on attempt to remove from empty stack or queue

Blocking



Block on attempt to add to full bounded stack or queue

Non-Blocking

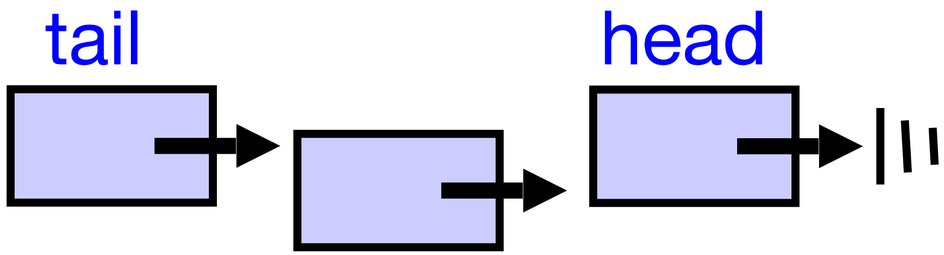


Throw exception on attempt to remove from empty stack or queue

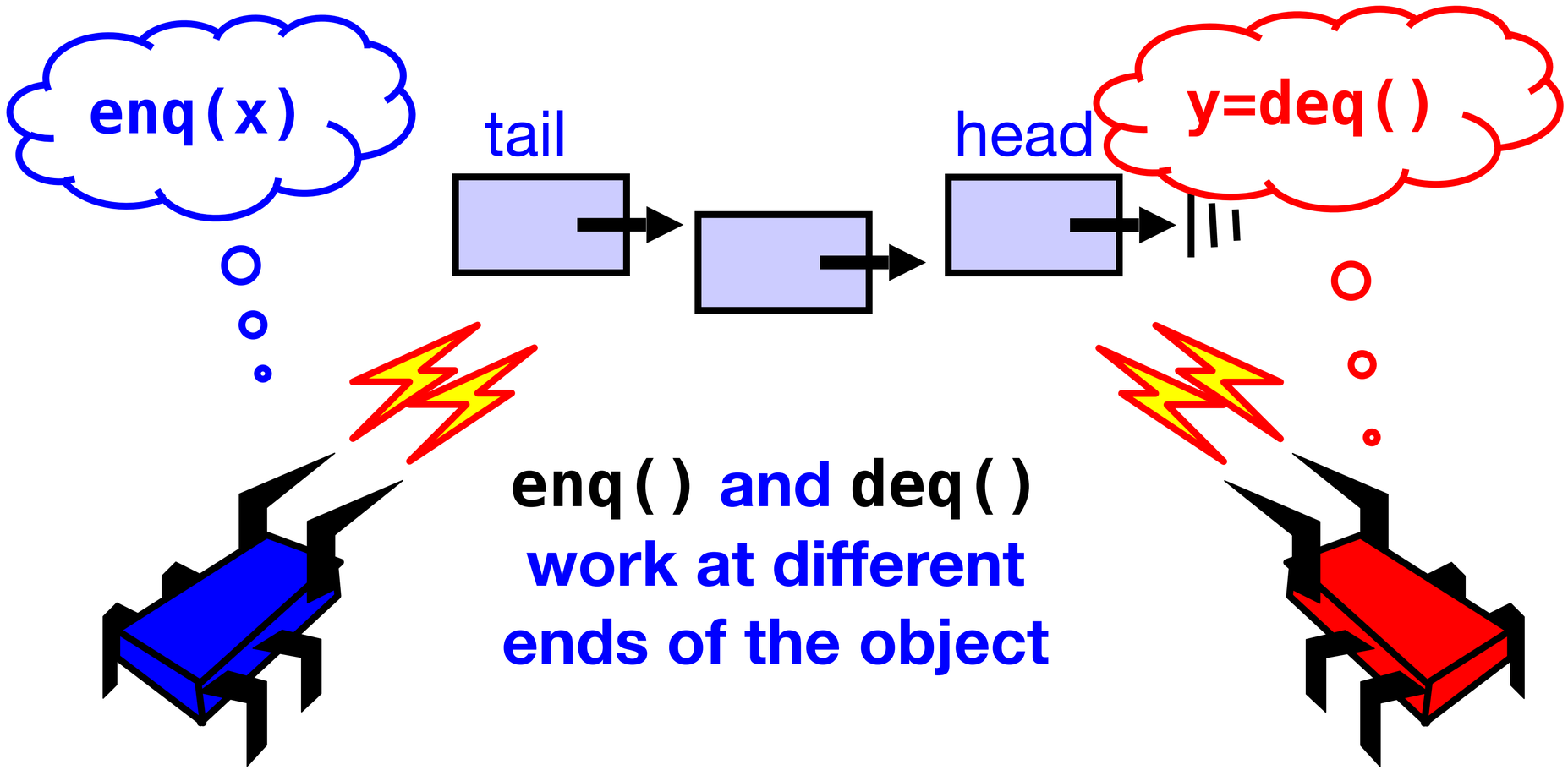
This Lecture

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

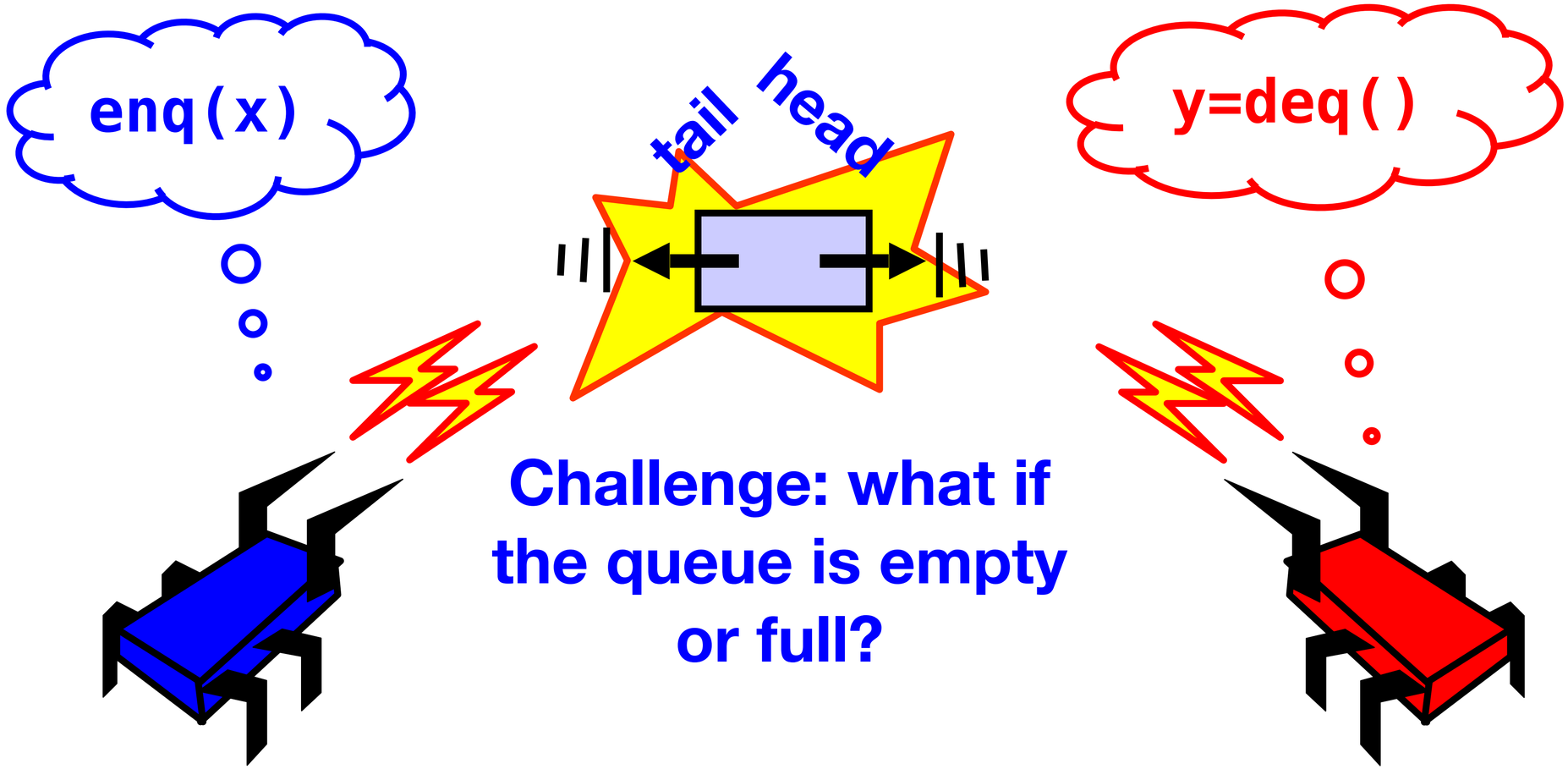
Queue: Concurrency



Queue: Concurrency

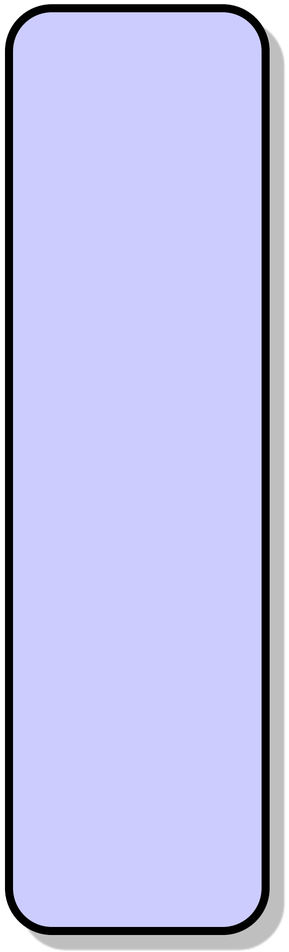


Queue: Concurrency

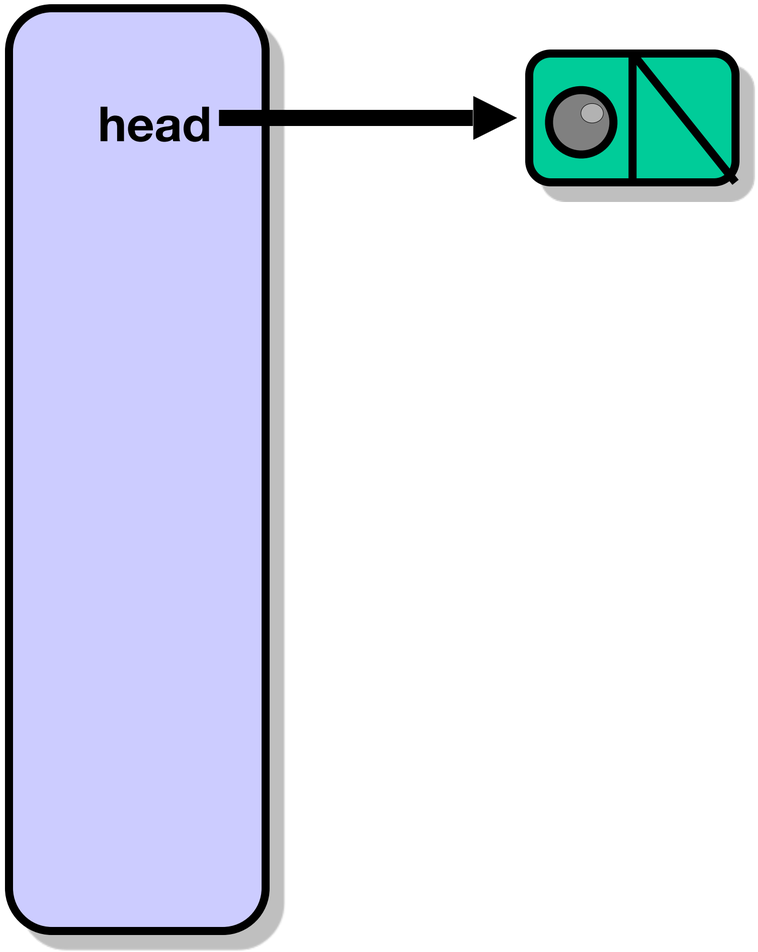


Challenge: what if the queue is empty or full?

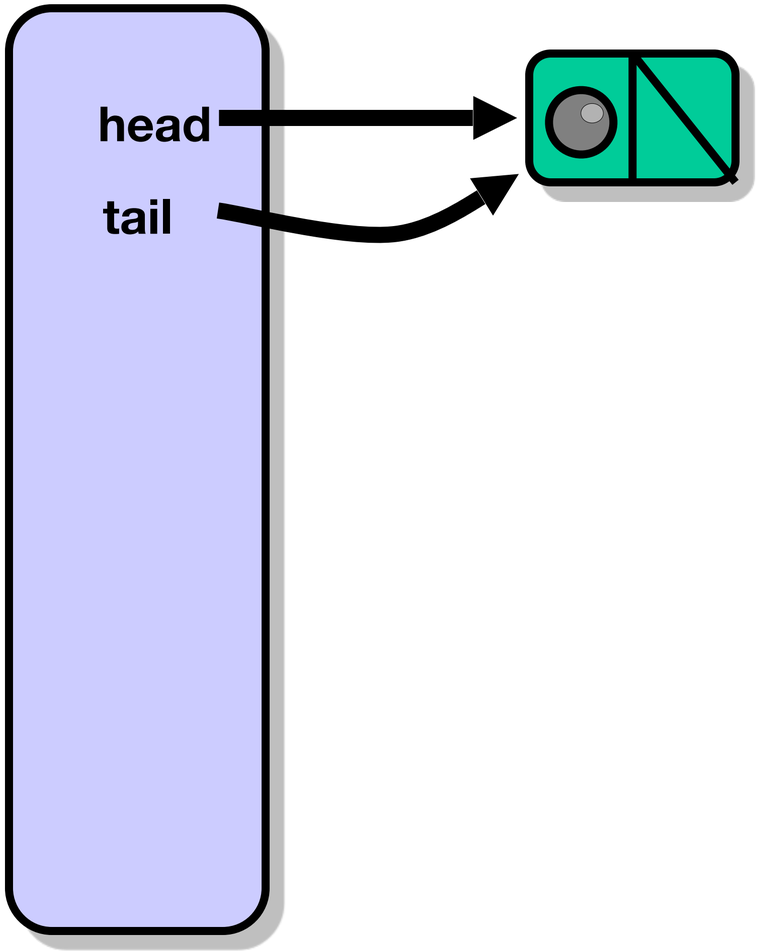
Bounded Queue



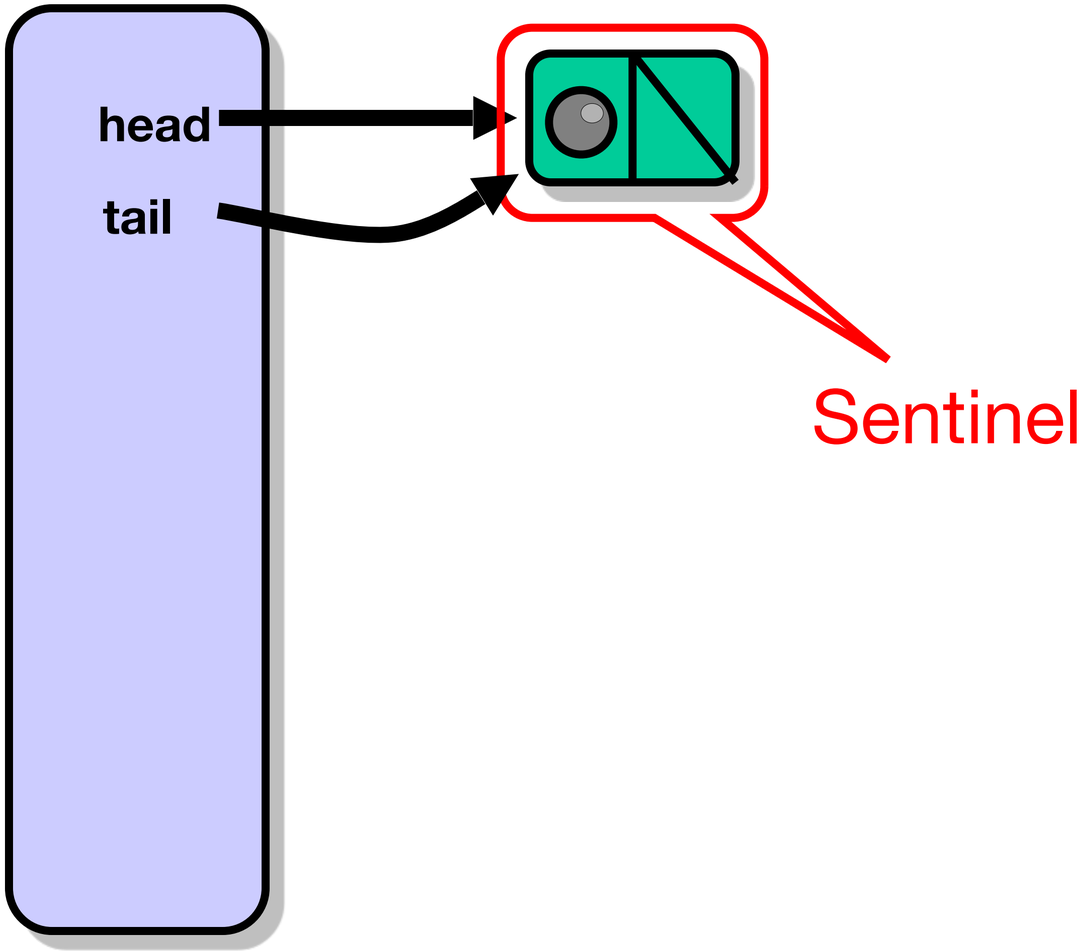
Bounded Queue



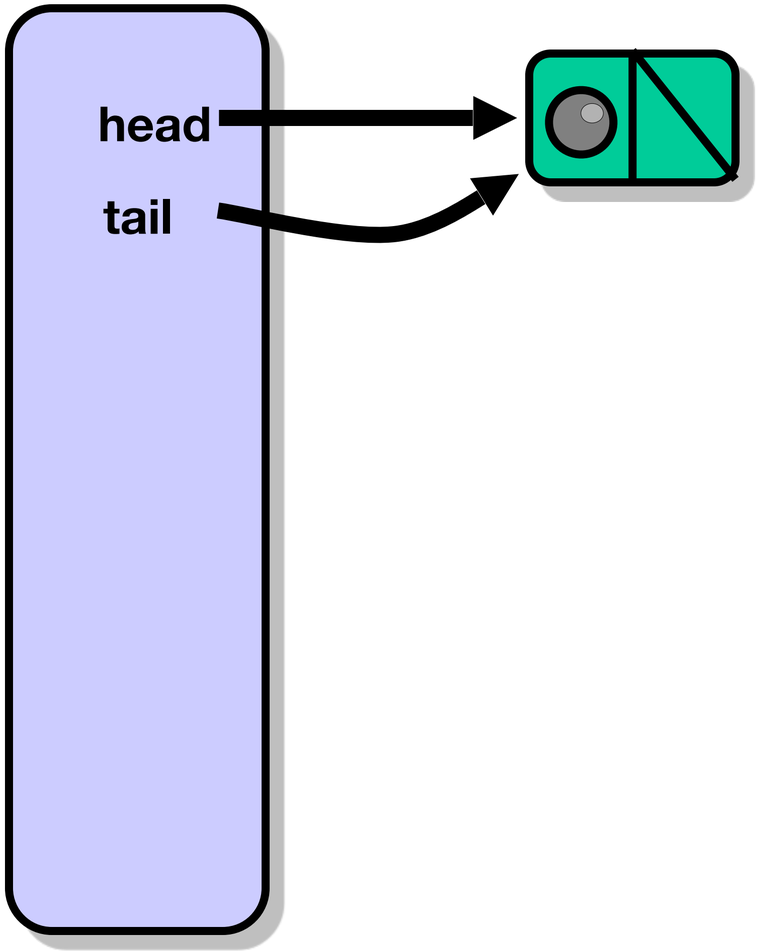
Bounded Queue



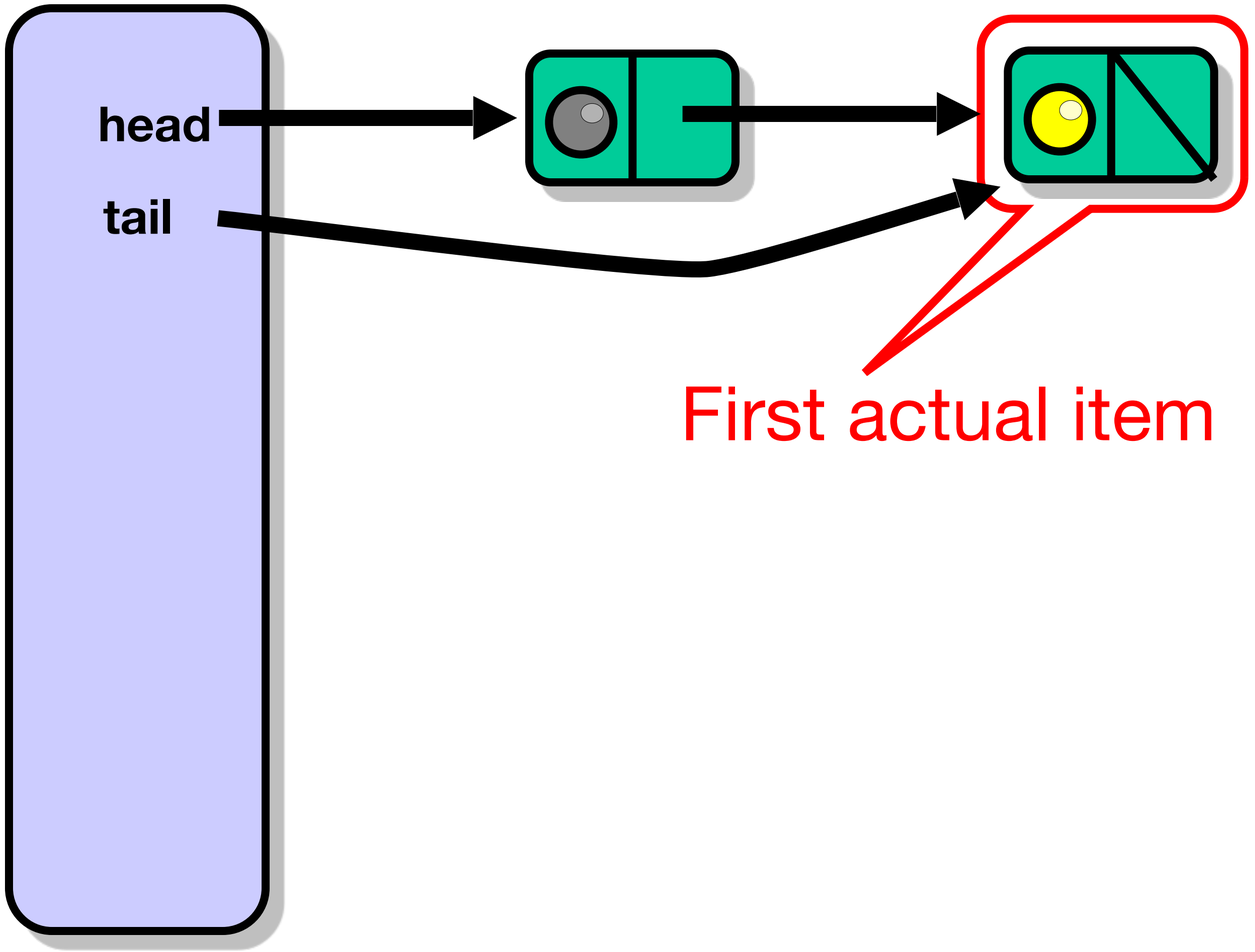
Bounded Queue



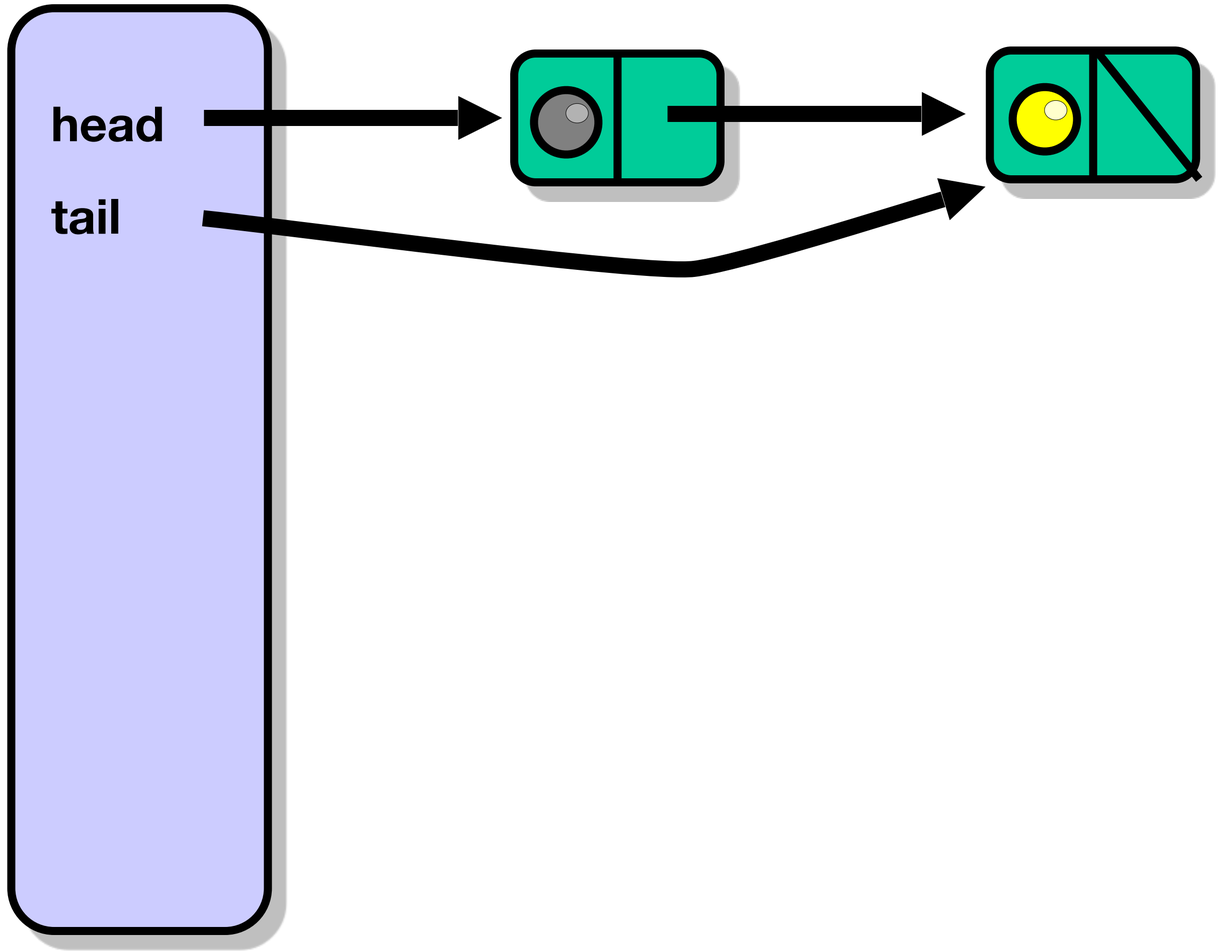
Bounded Queue



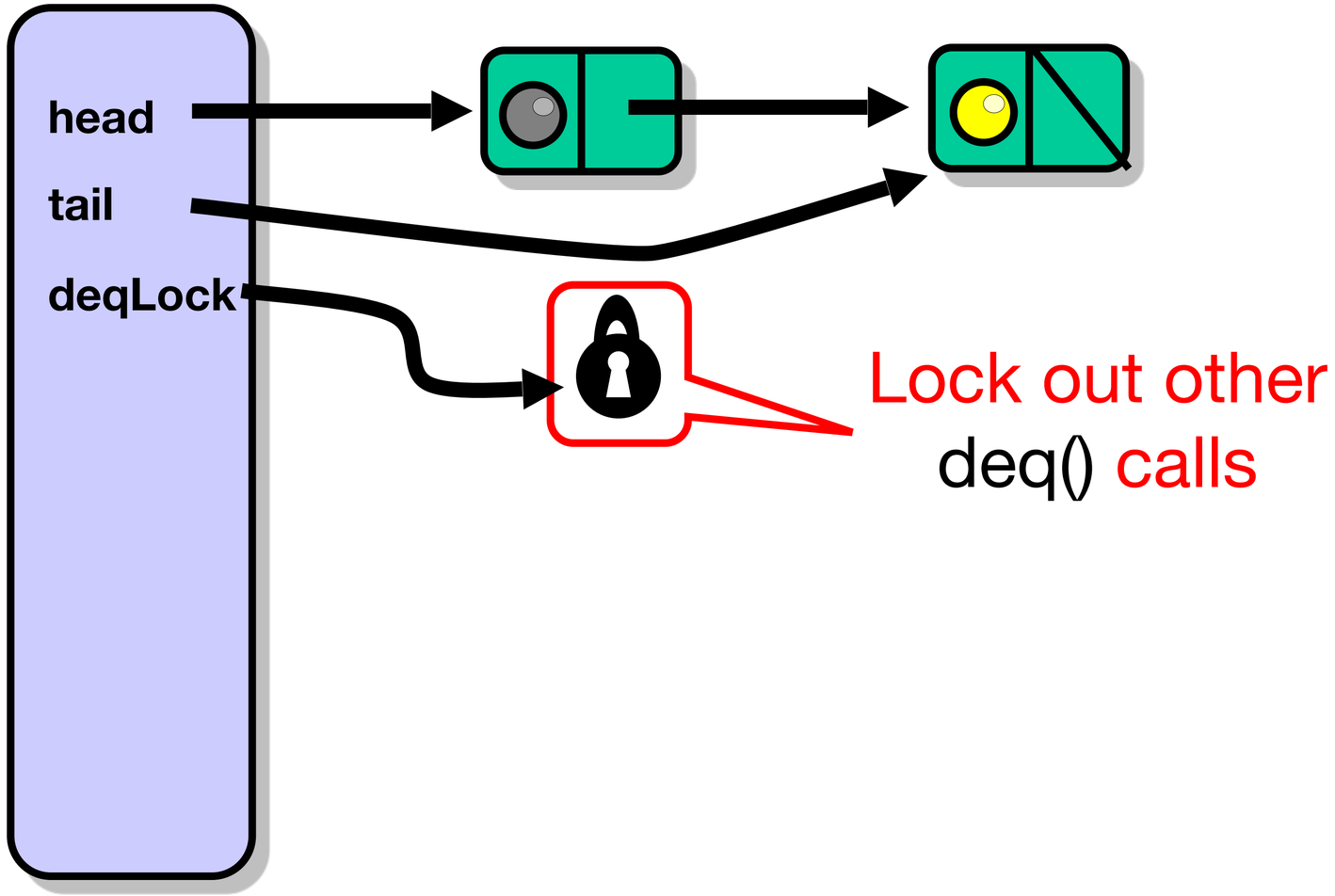
Bounded Queue



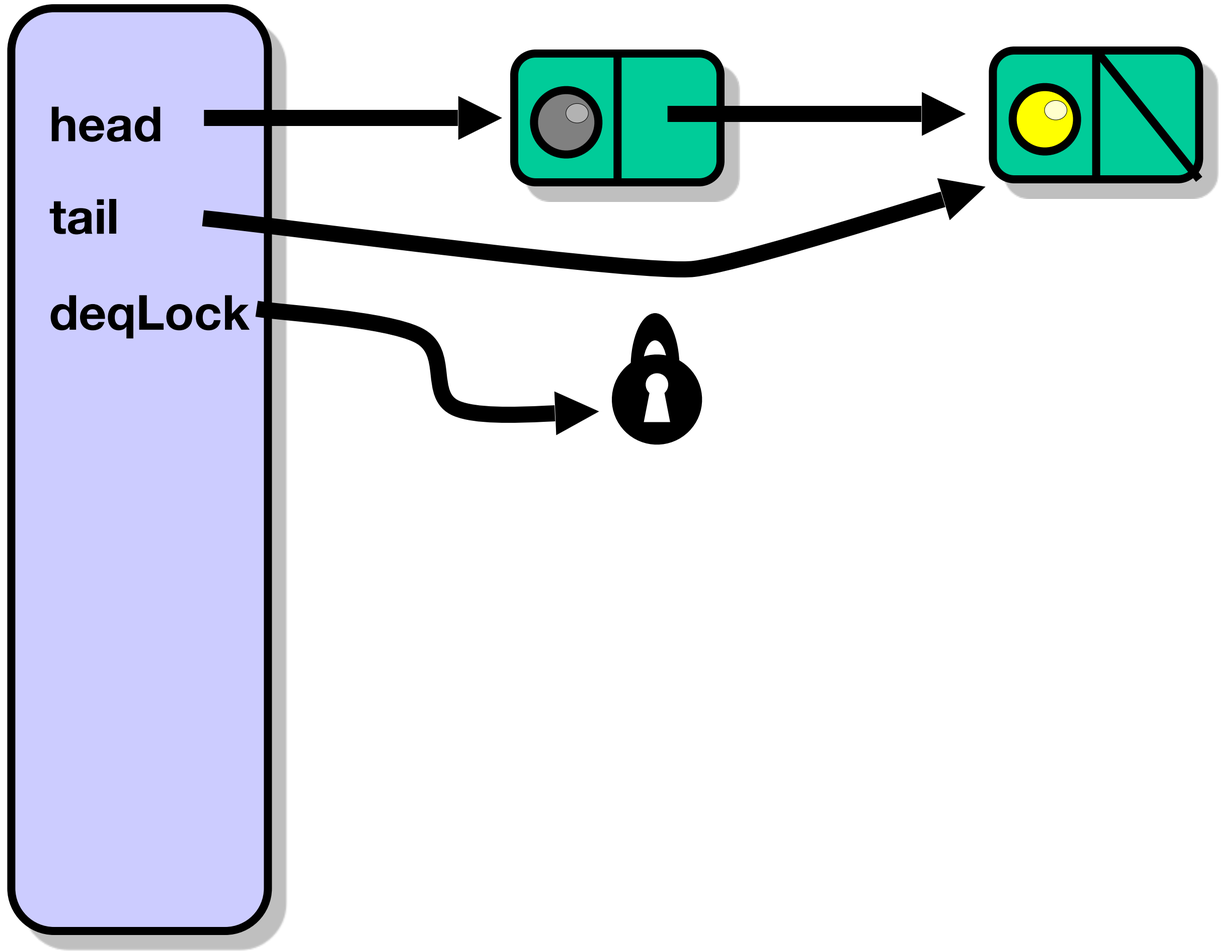
Bounded Queue



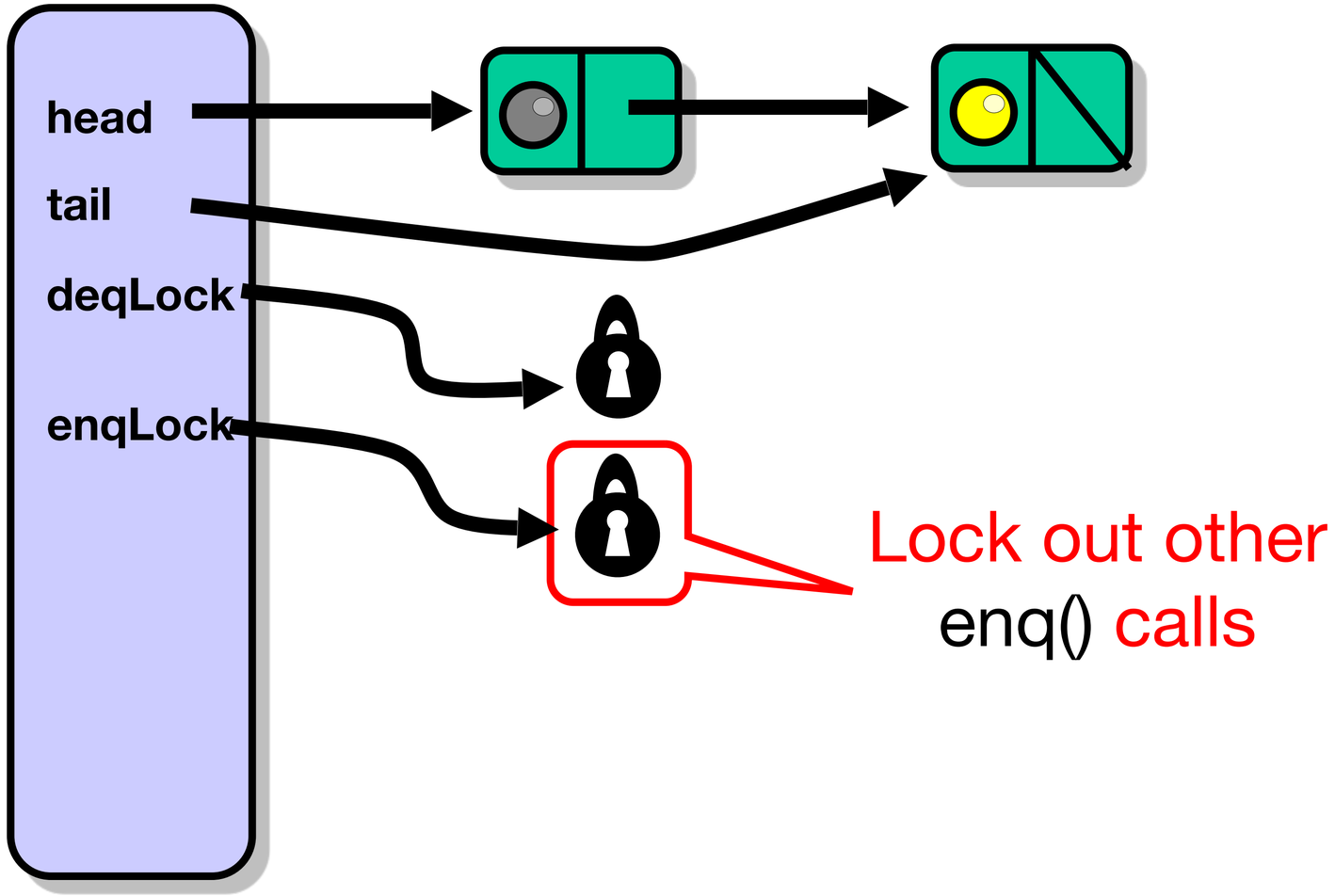
Bounded Queue



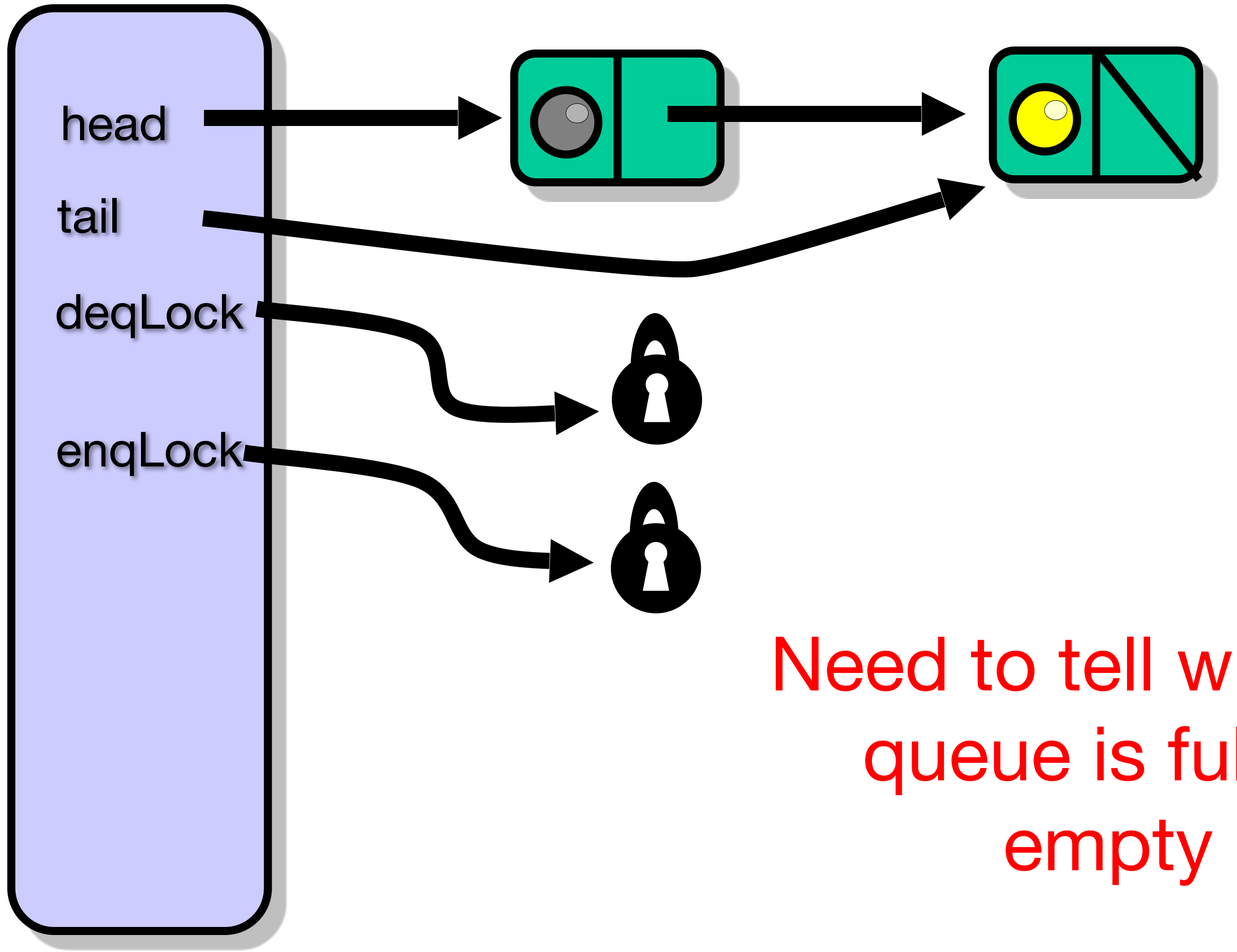
Bounded Queue



Bounded Queue

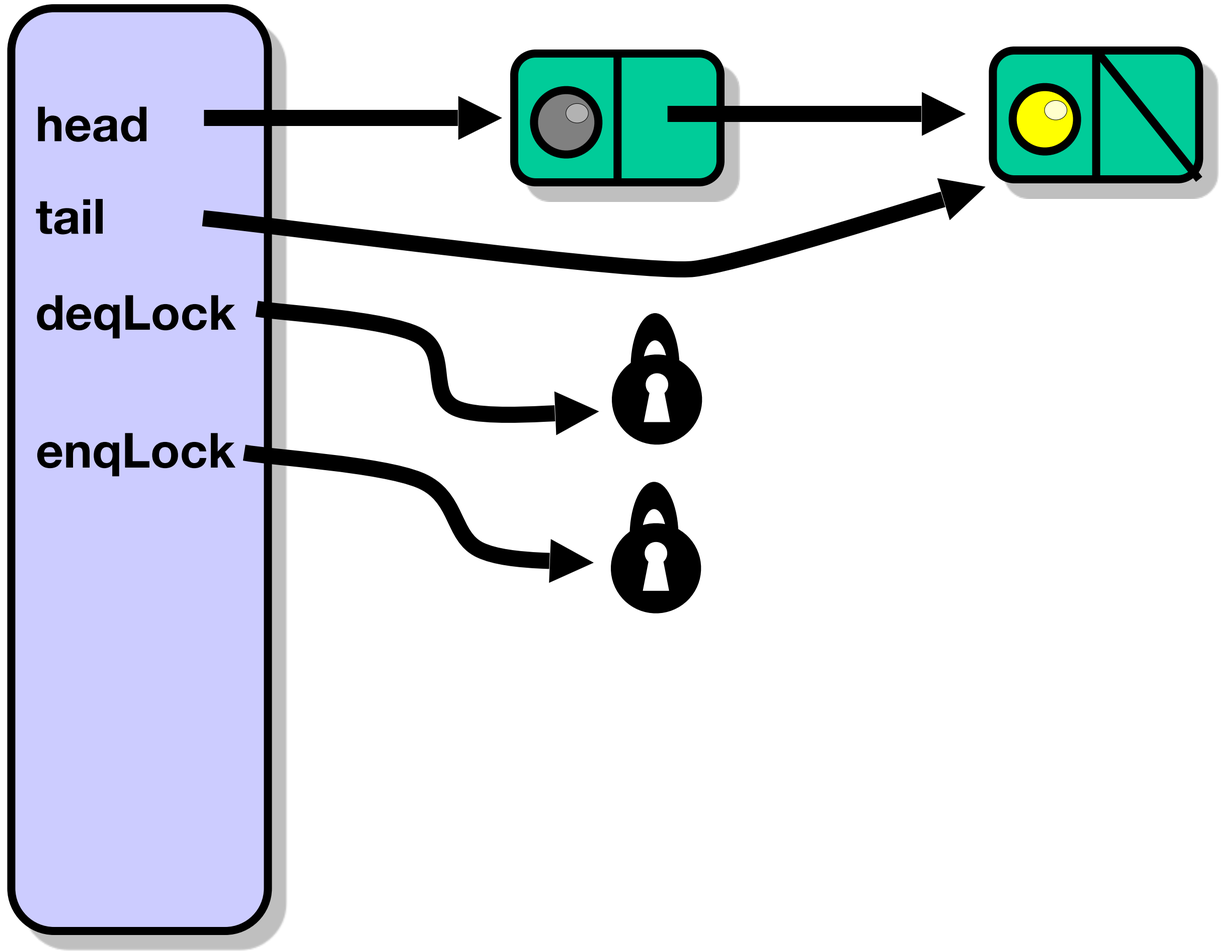


Not done yet

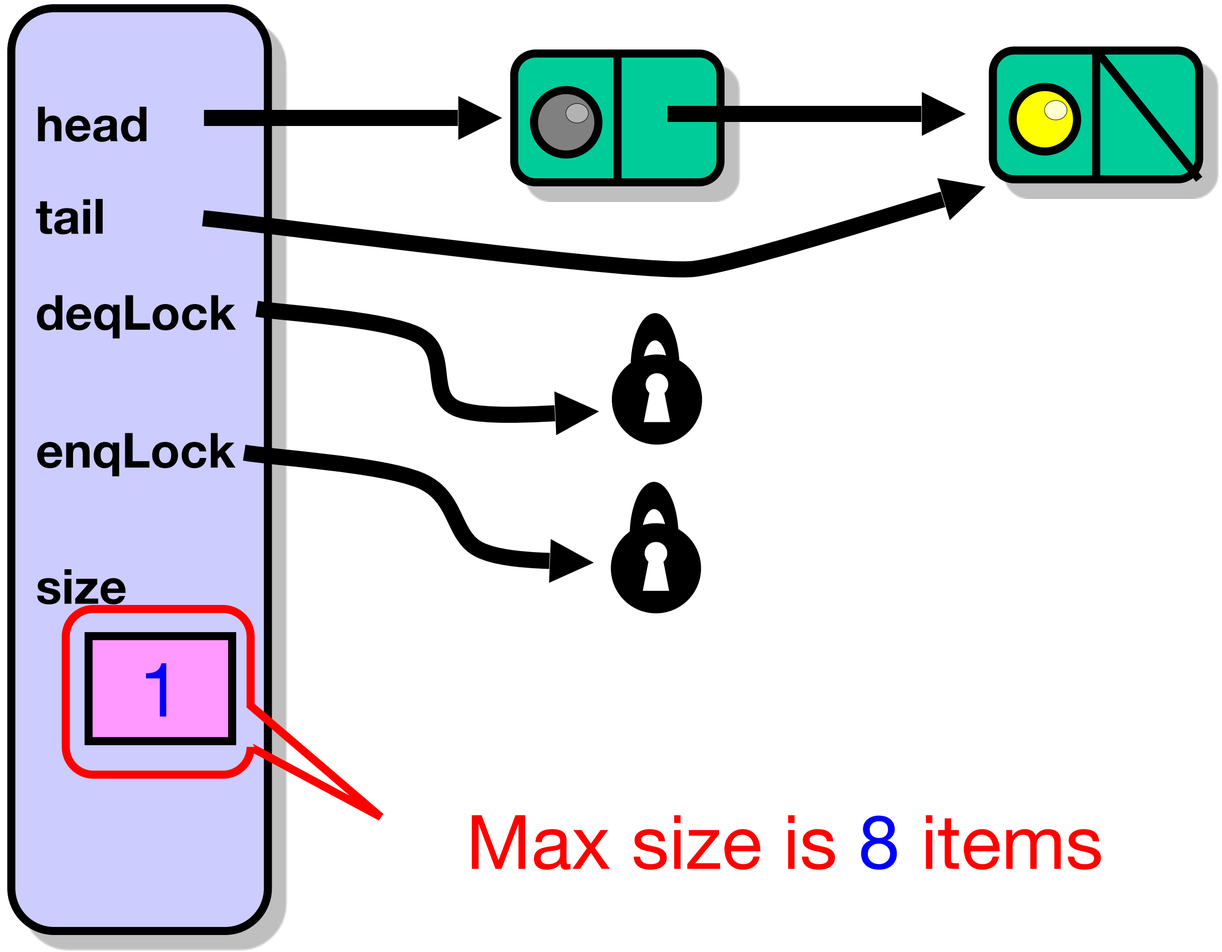


Need to tell whether queue is full or empty

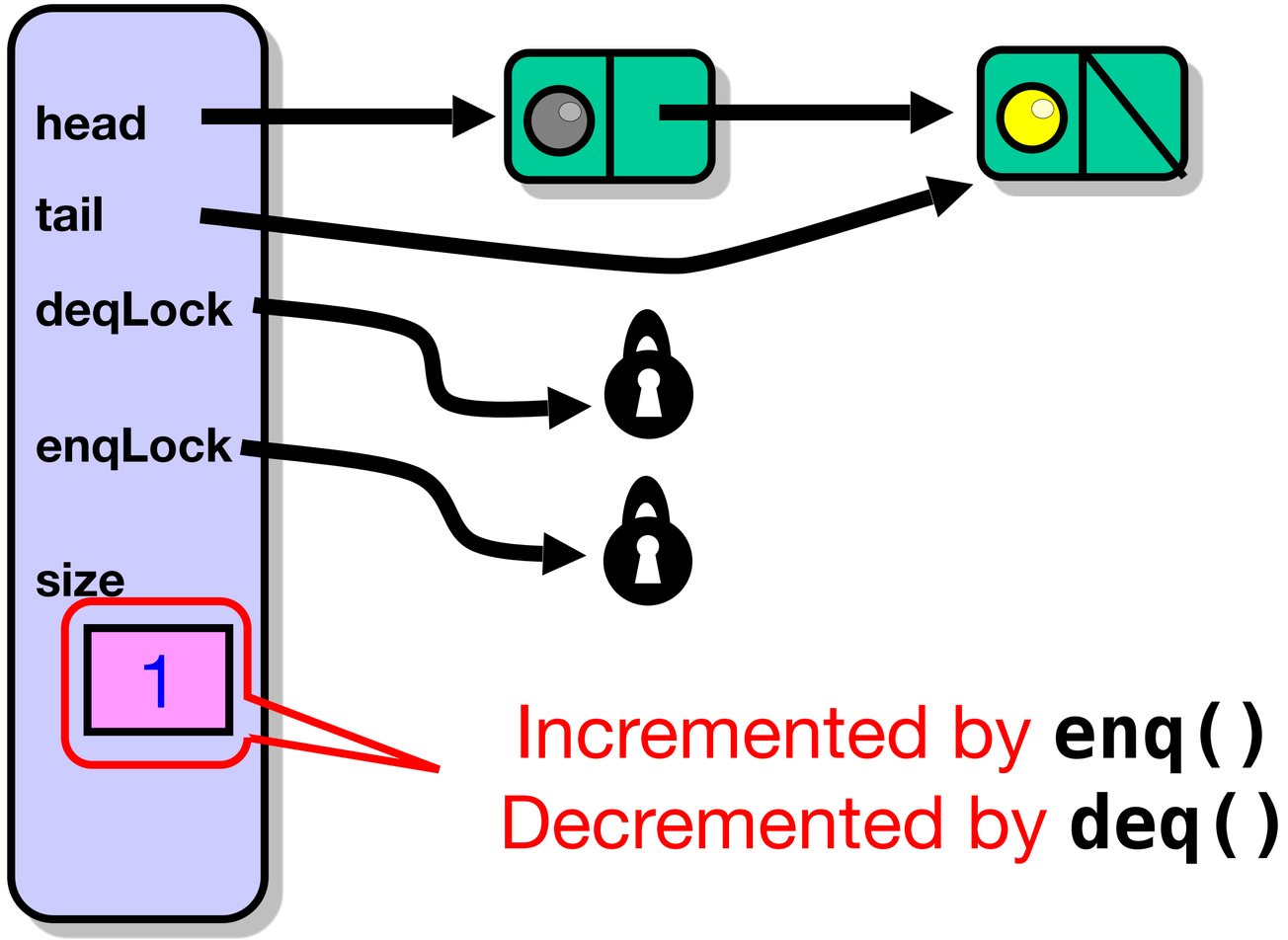
Not done yet



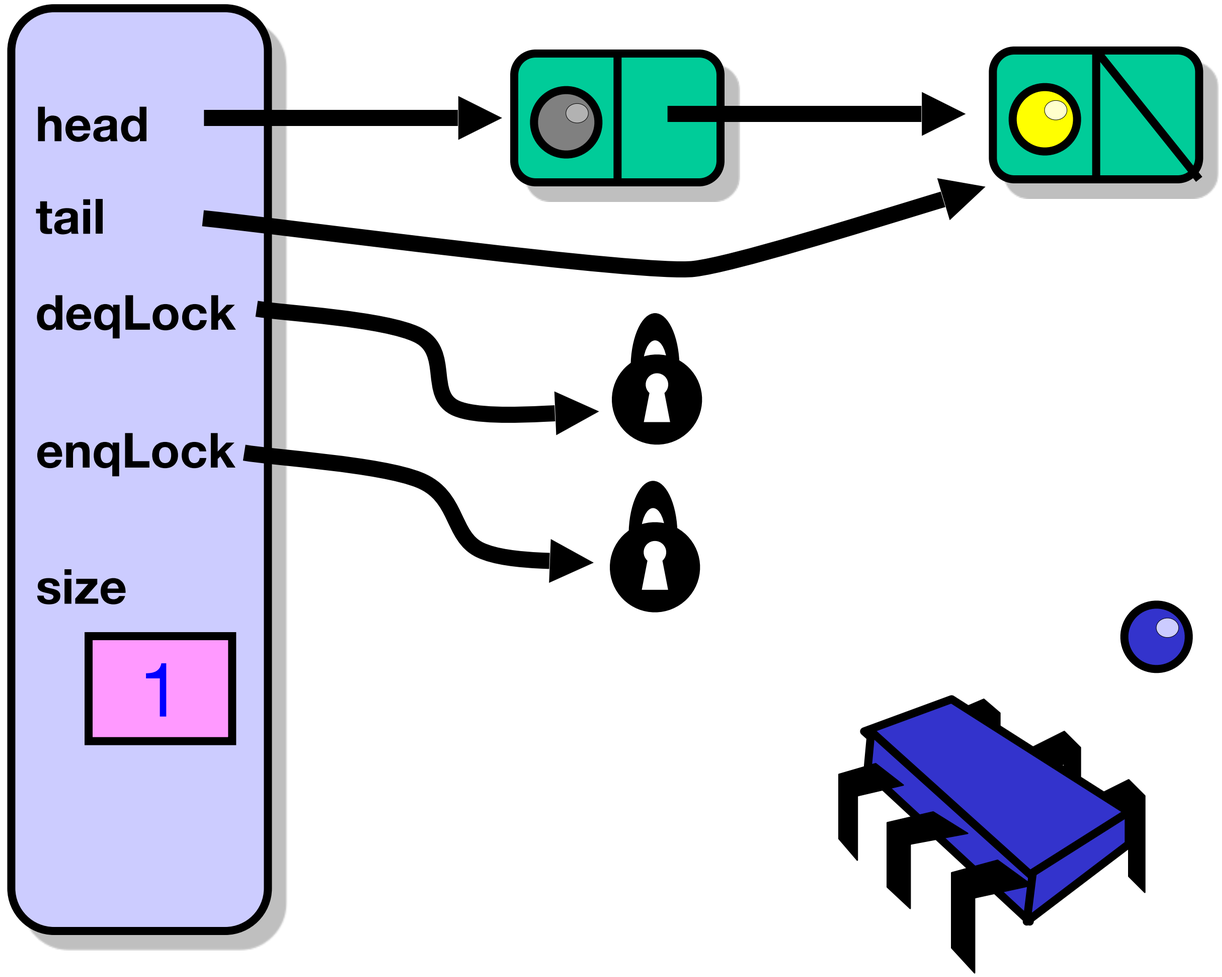
Not done yet



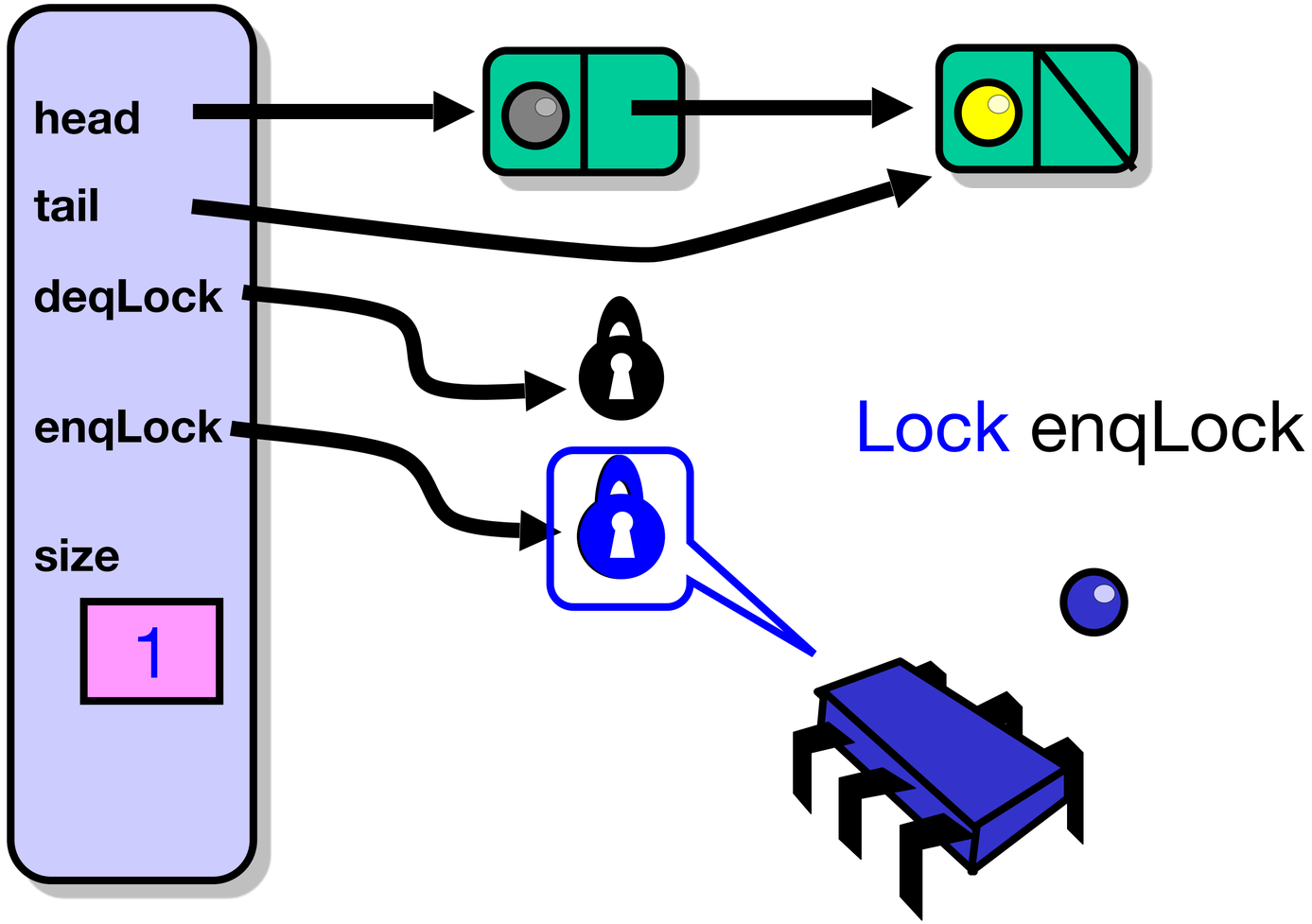
Not done yet


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

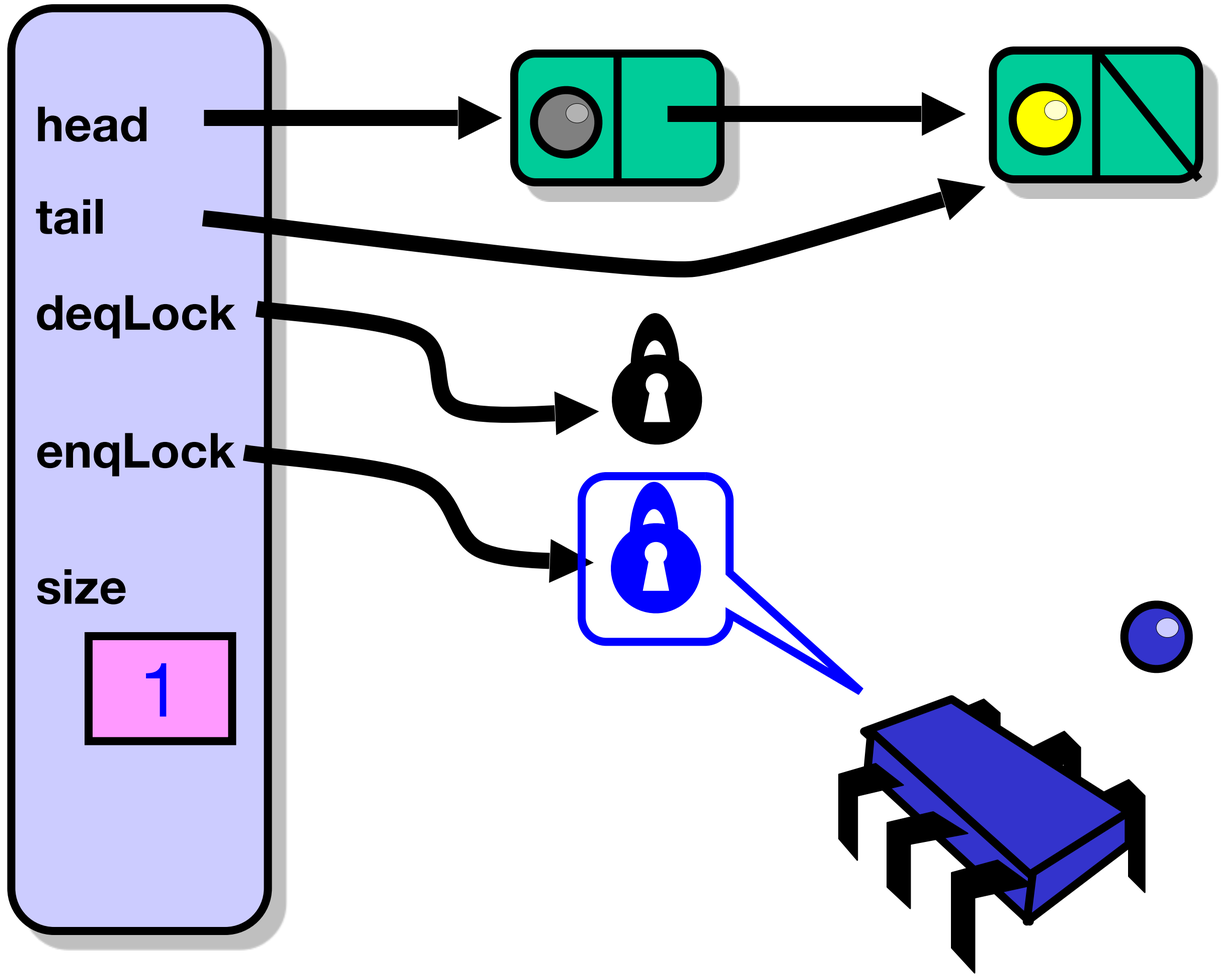
Enqueuer



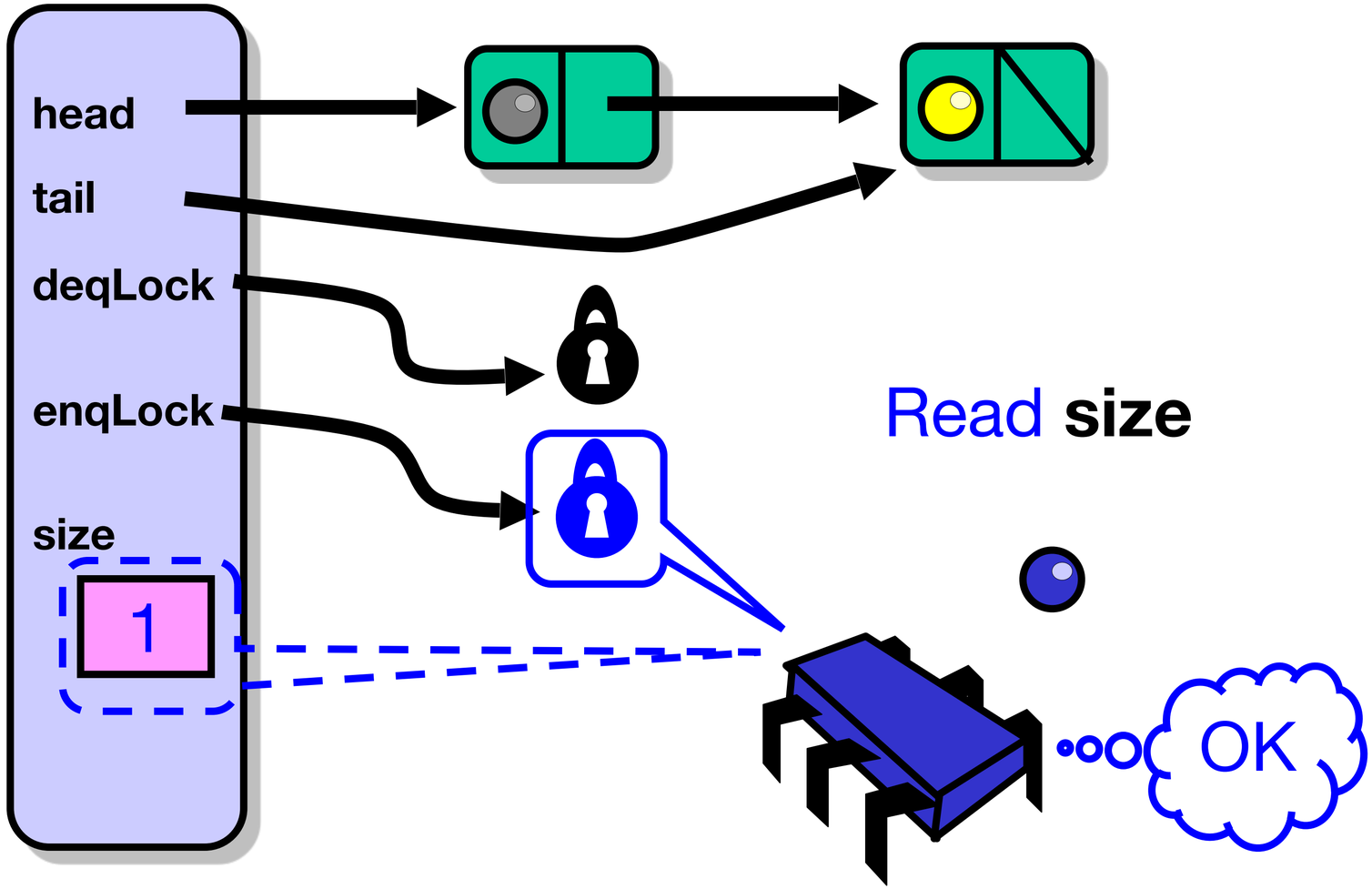
Enqueuer



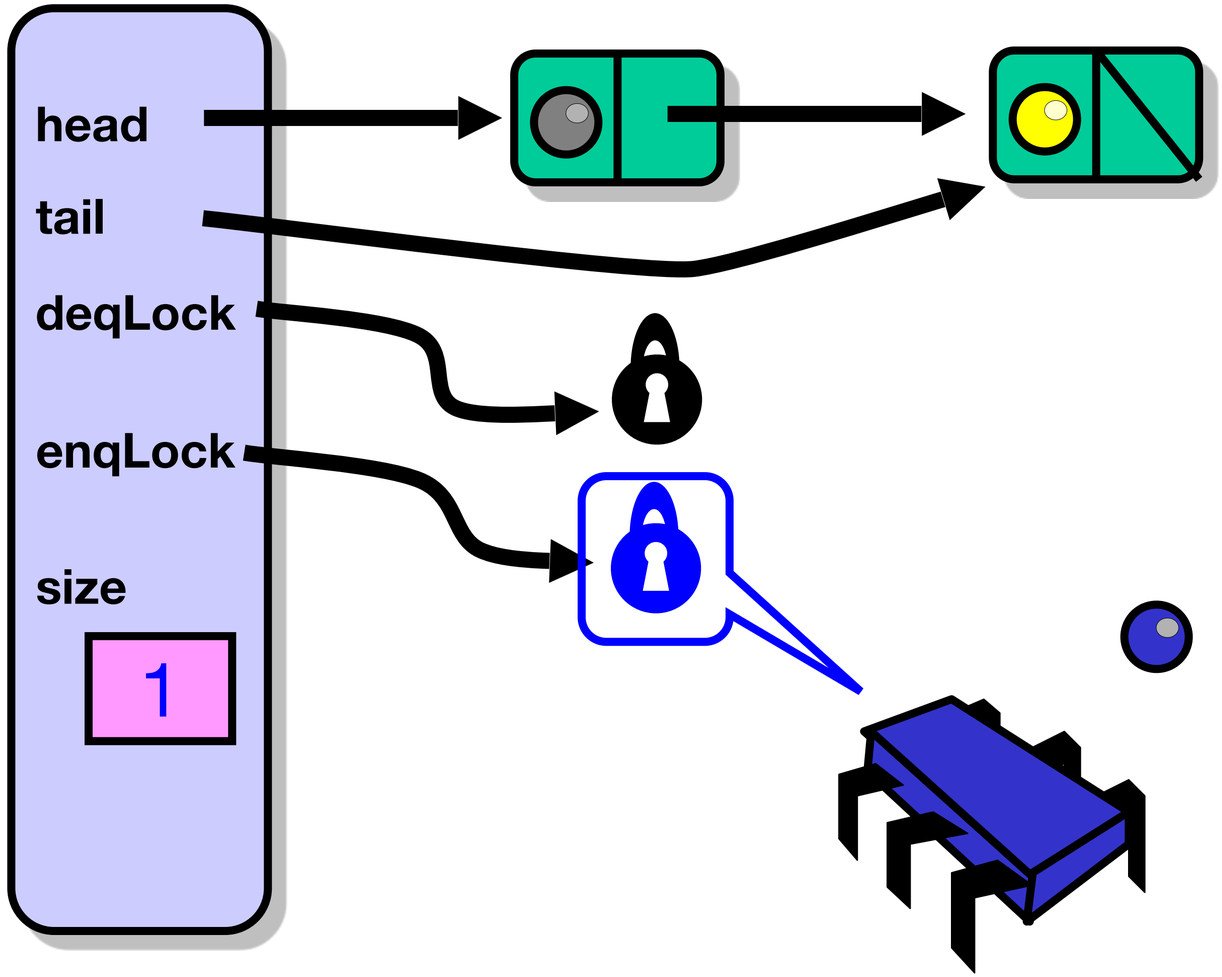
Enqueuer



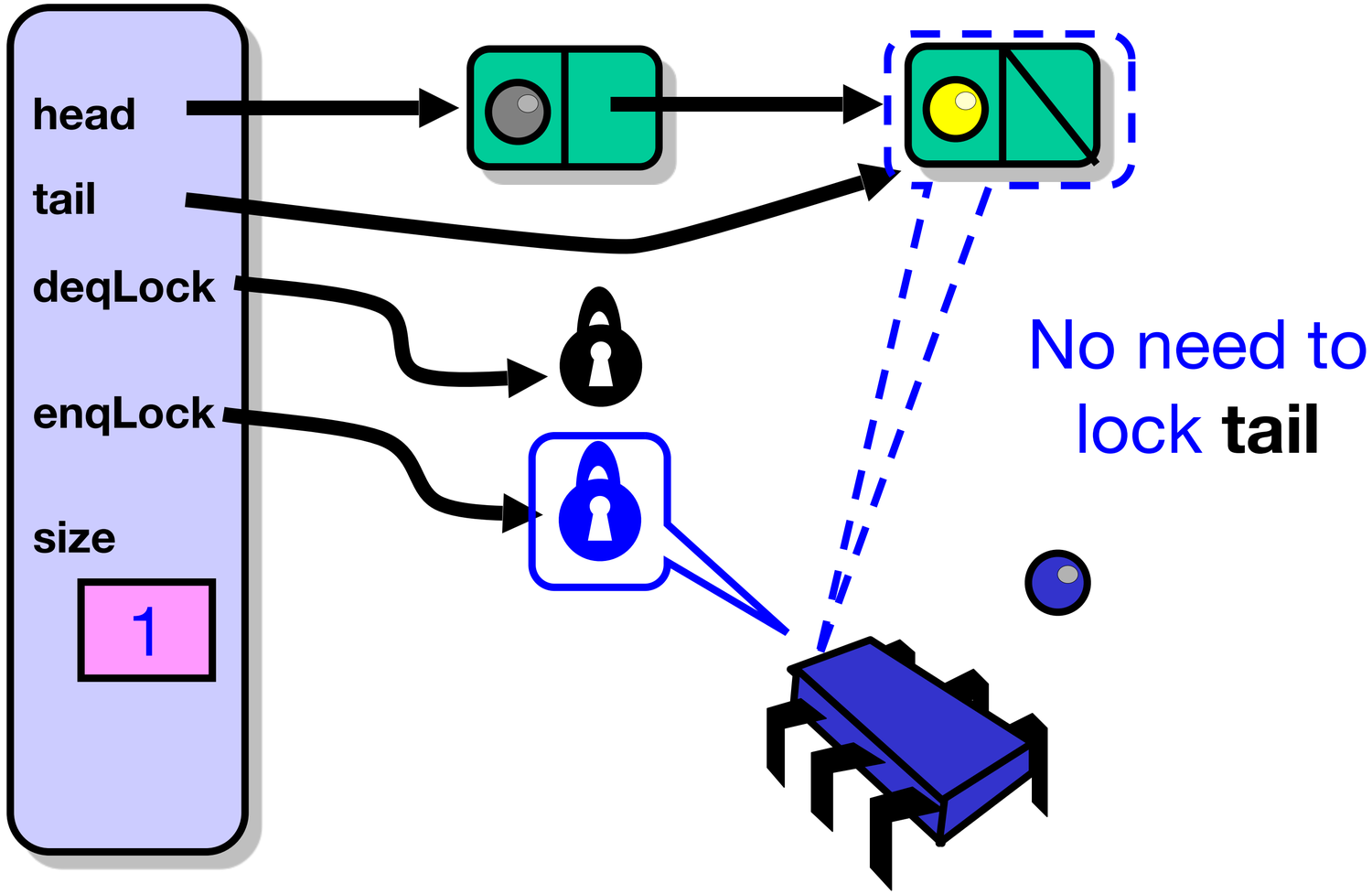
Enqueuer



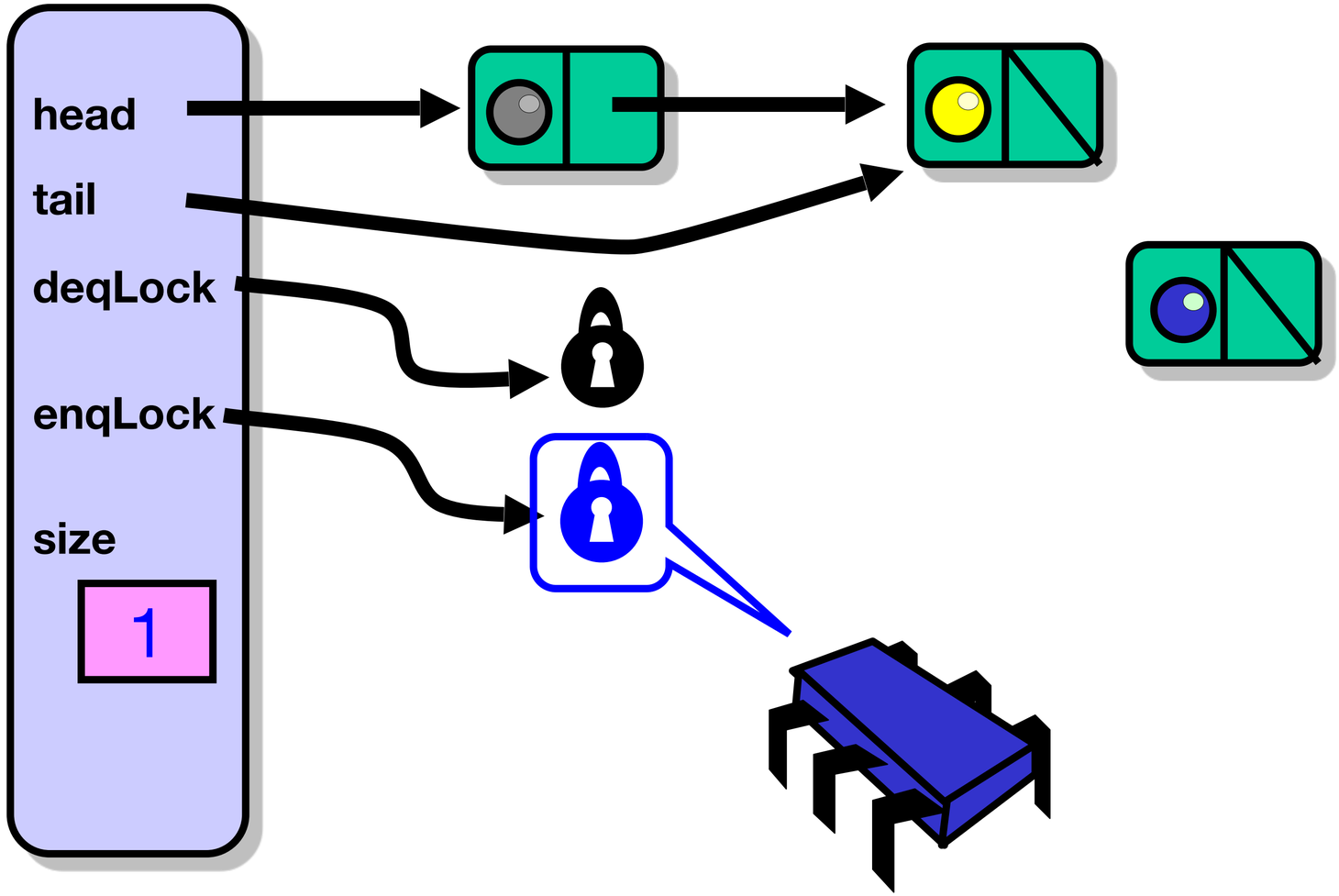
Enqueuer



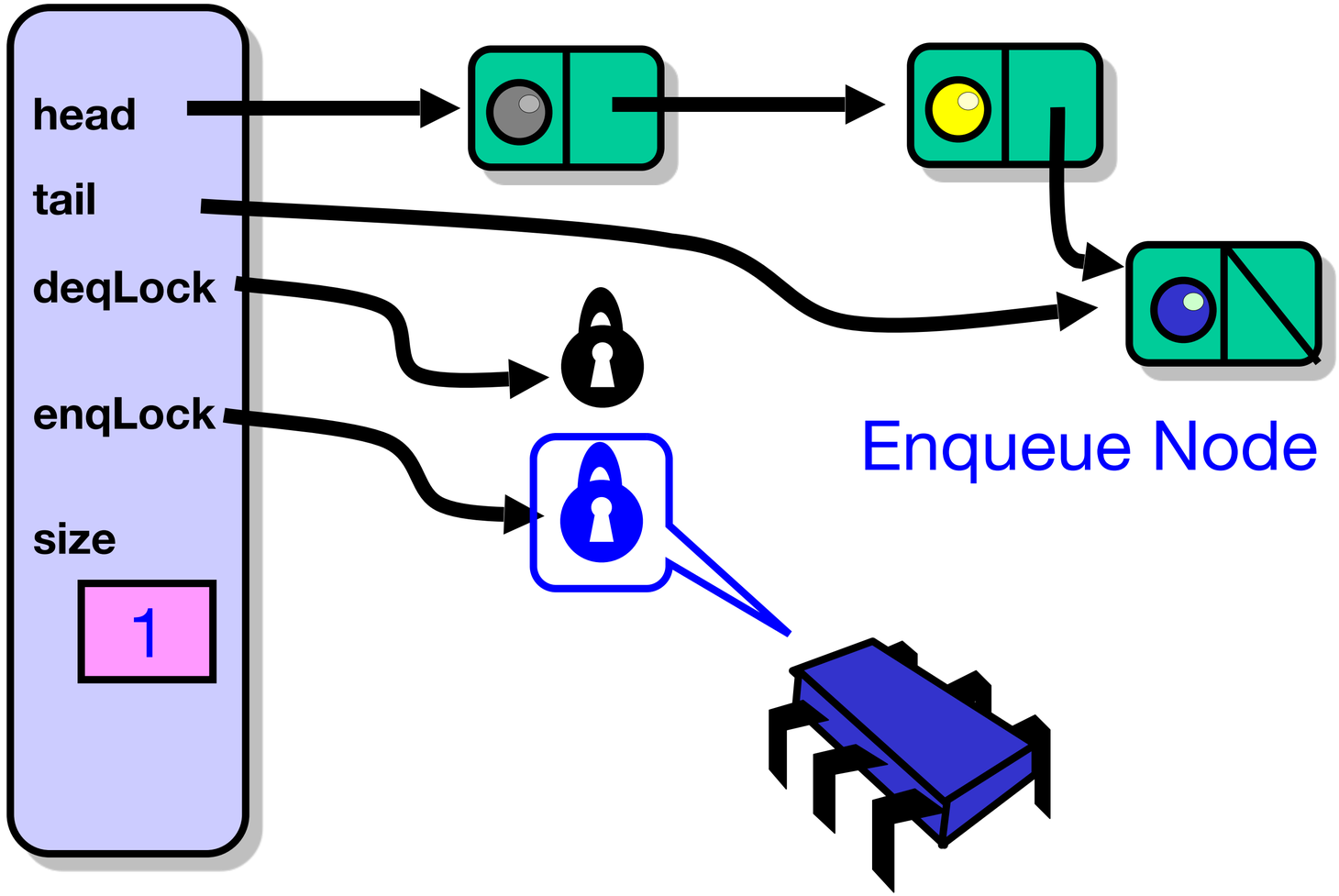
Enqueuer



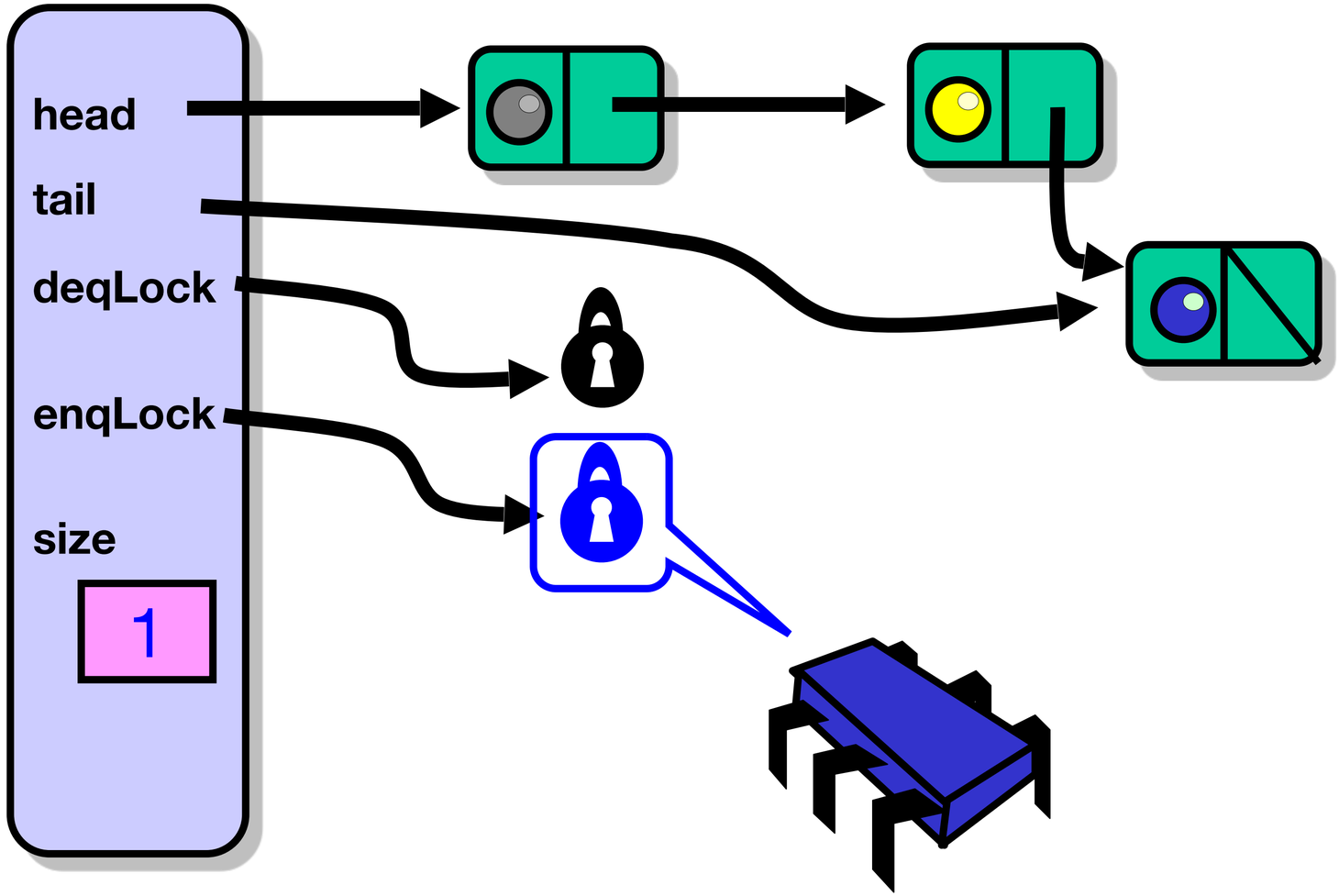
Enqueuer



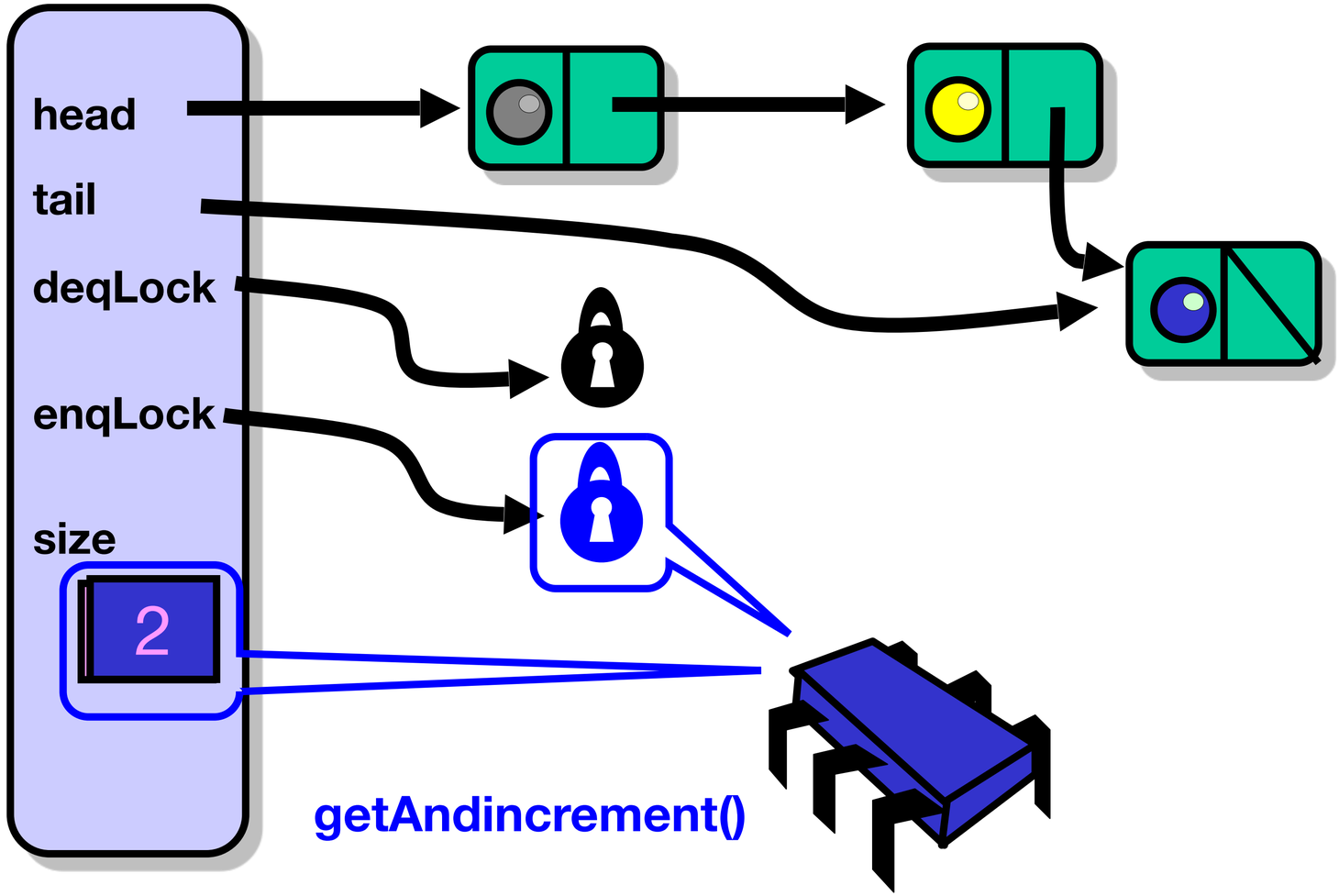
Enqueuer



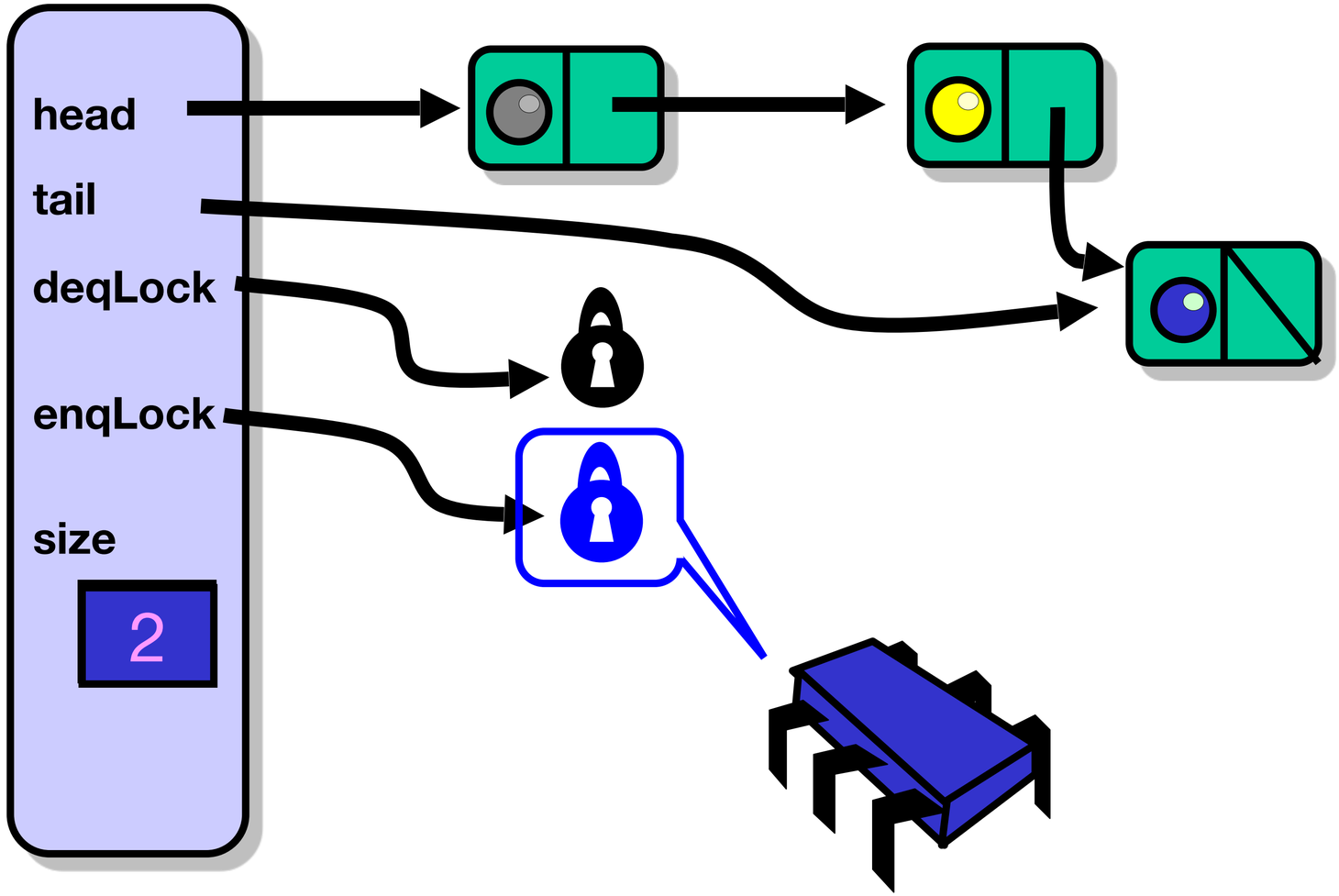
Enqueuer



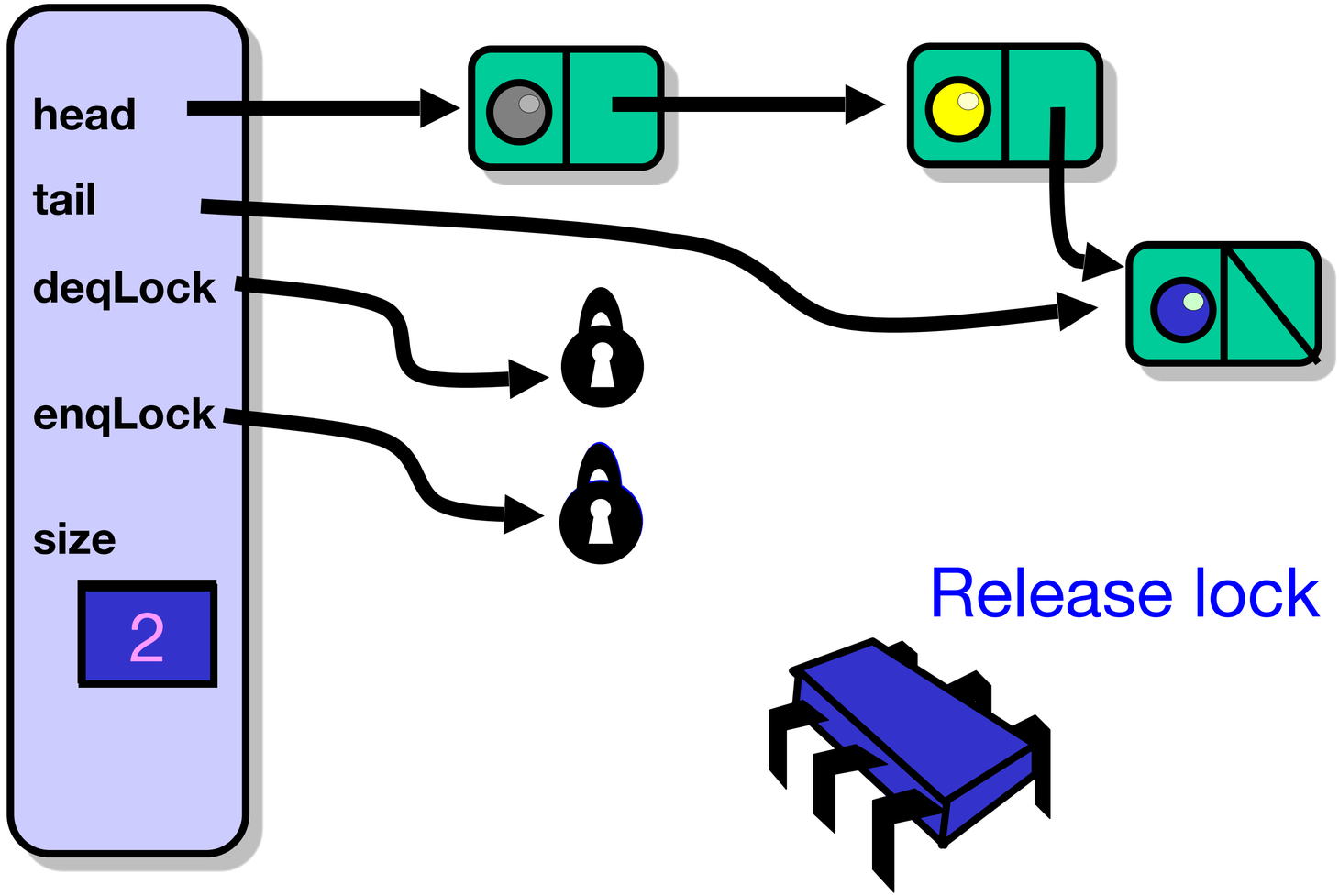
Enqueuer



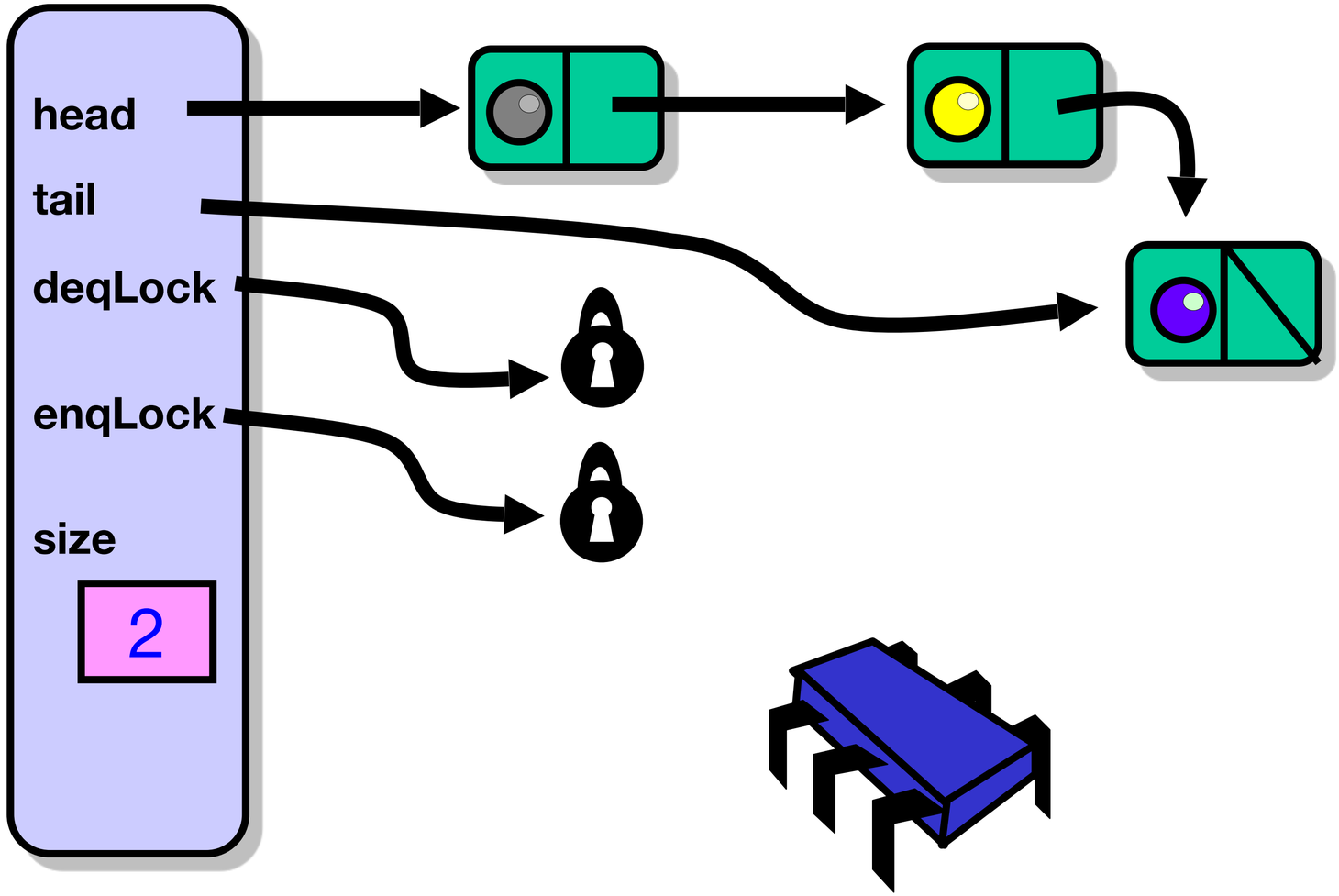
Enqueuer



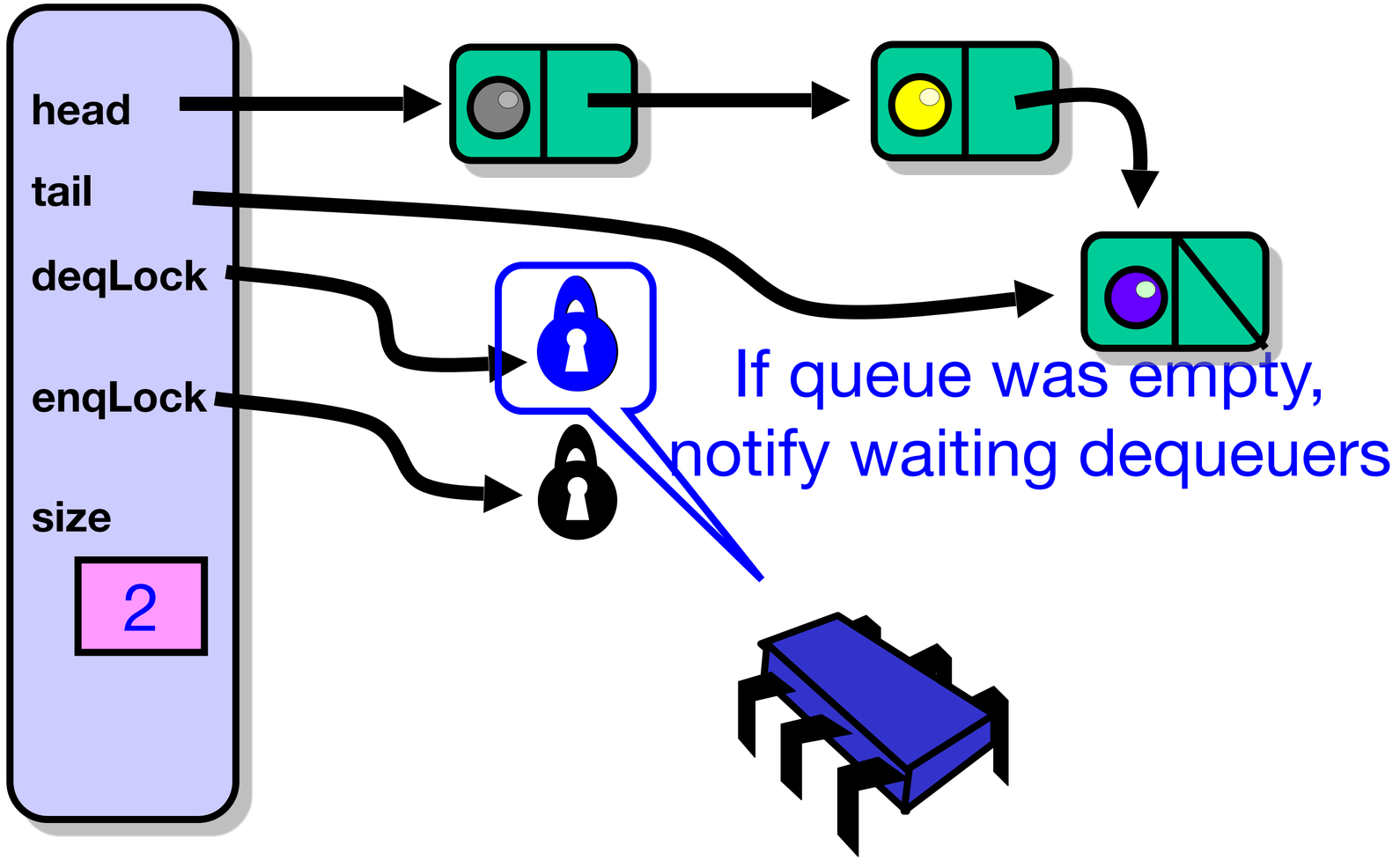
Enqueuer



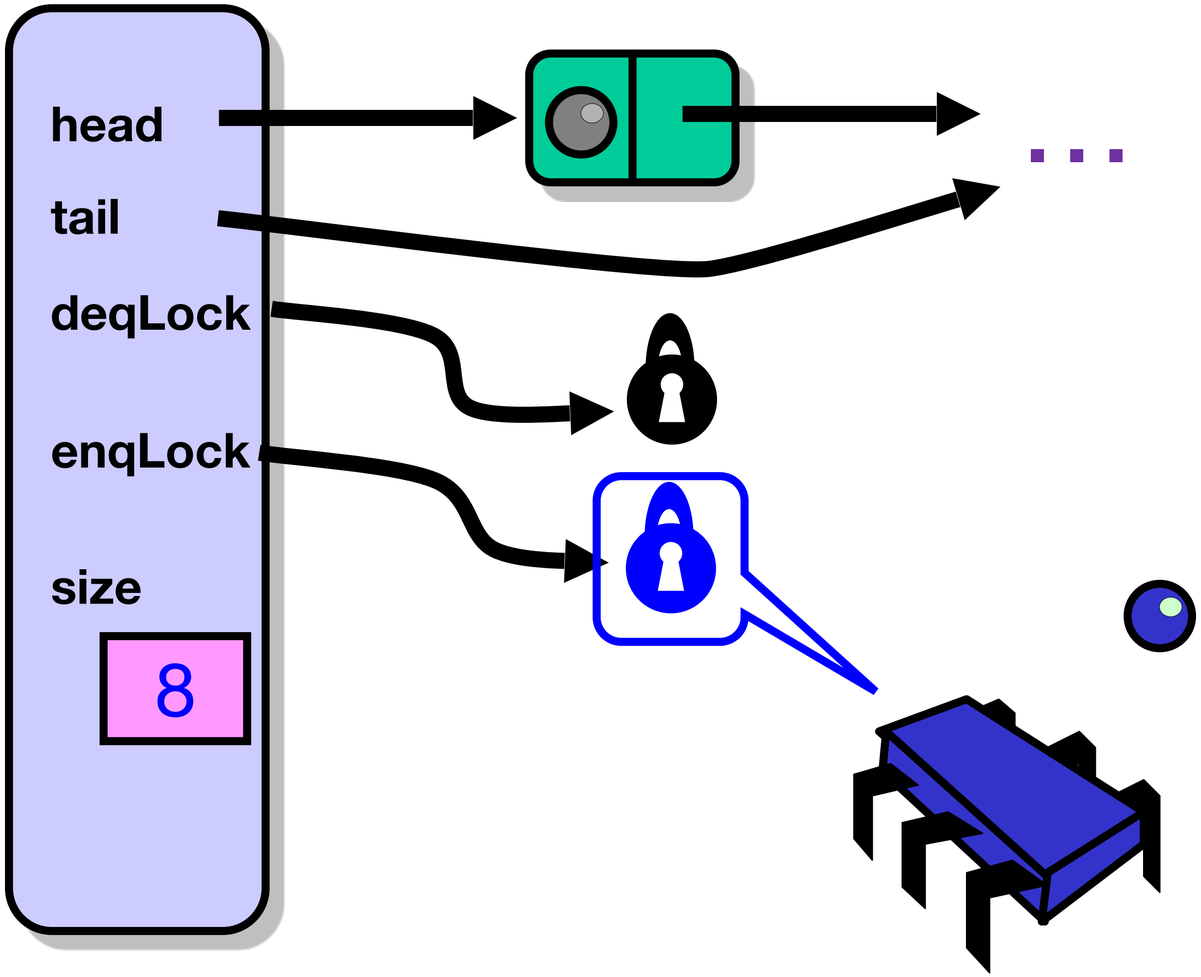
Enqueuer



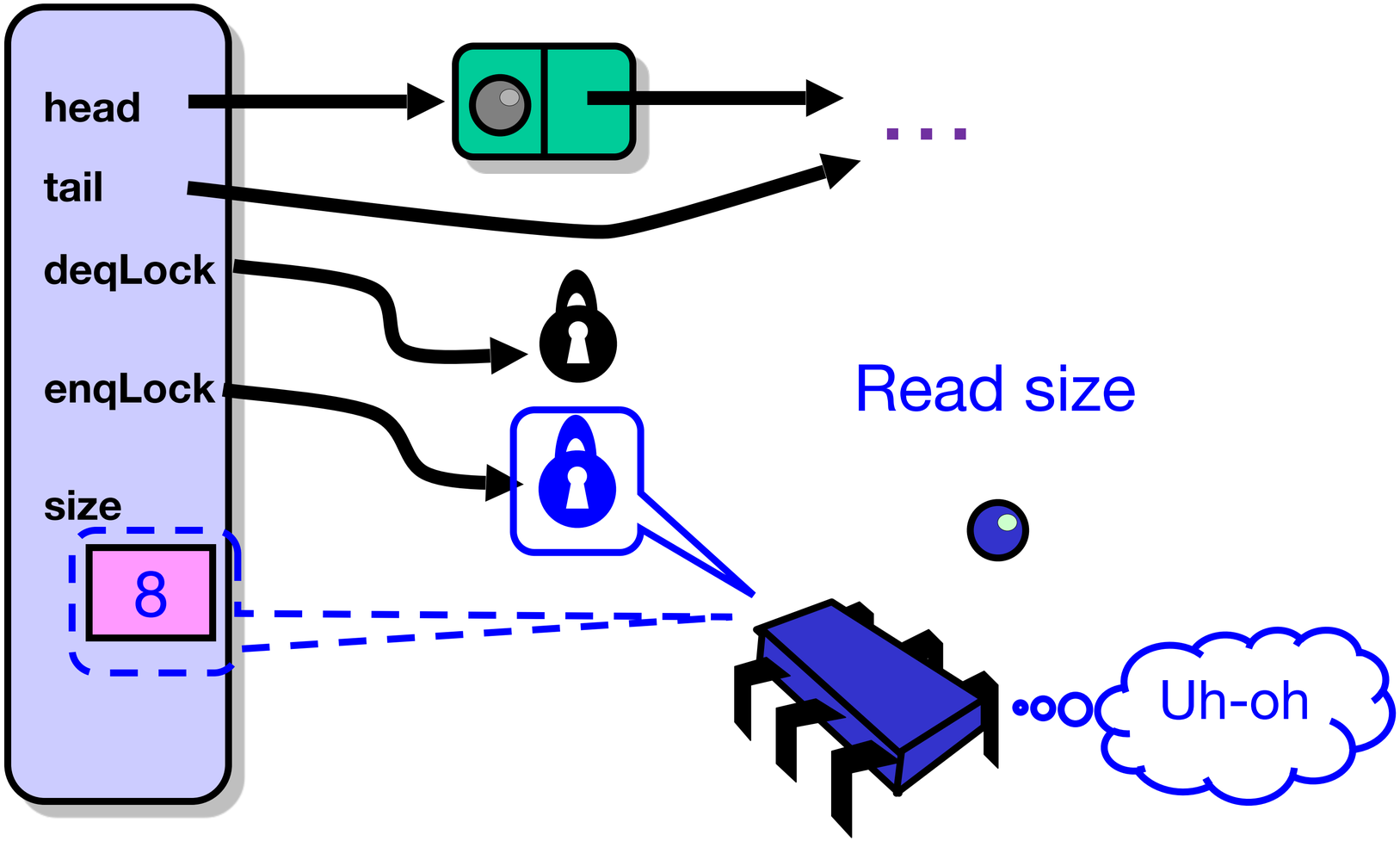
Enqueuer


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

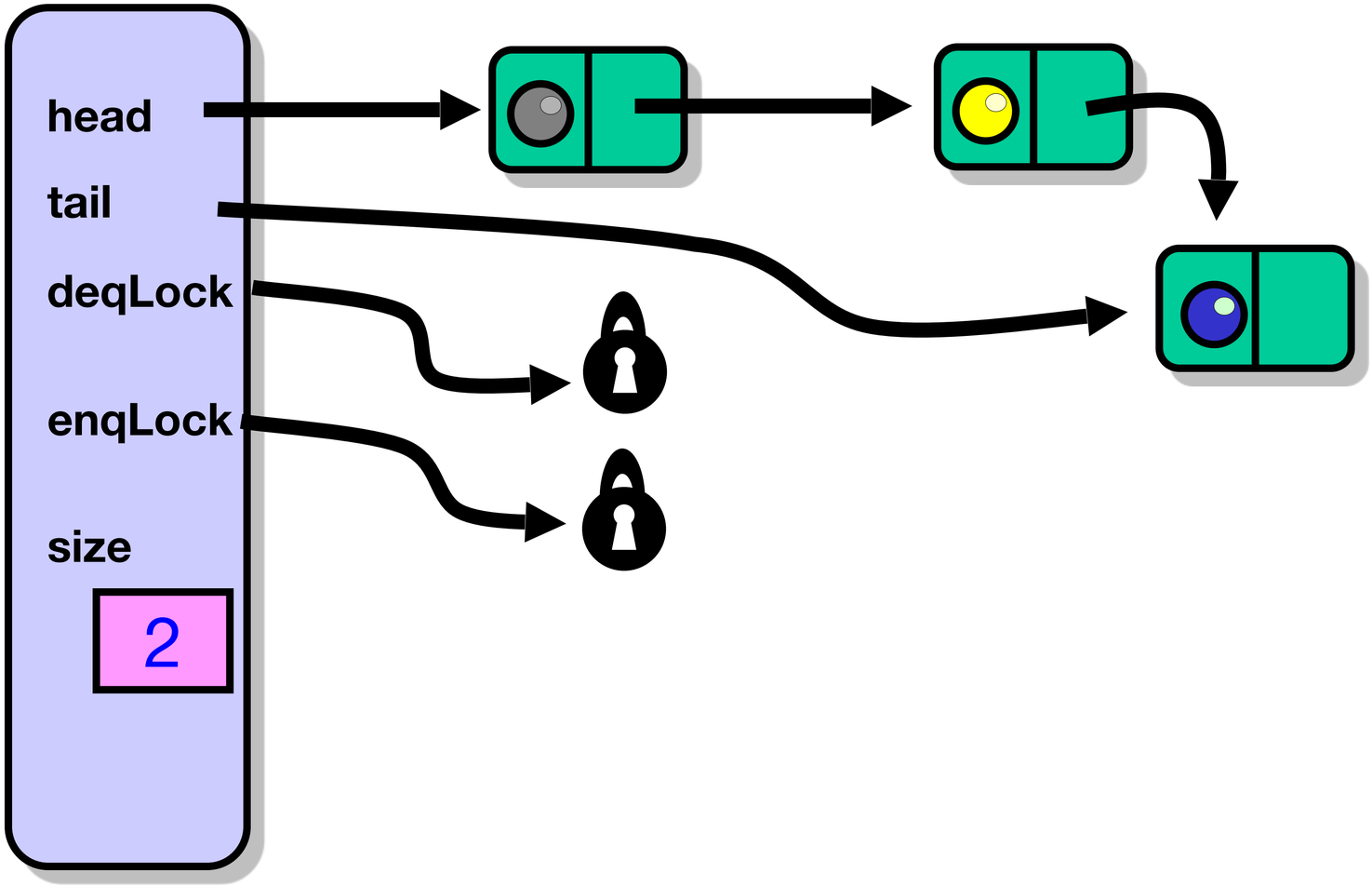
Unsuccessful Enqueuer



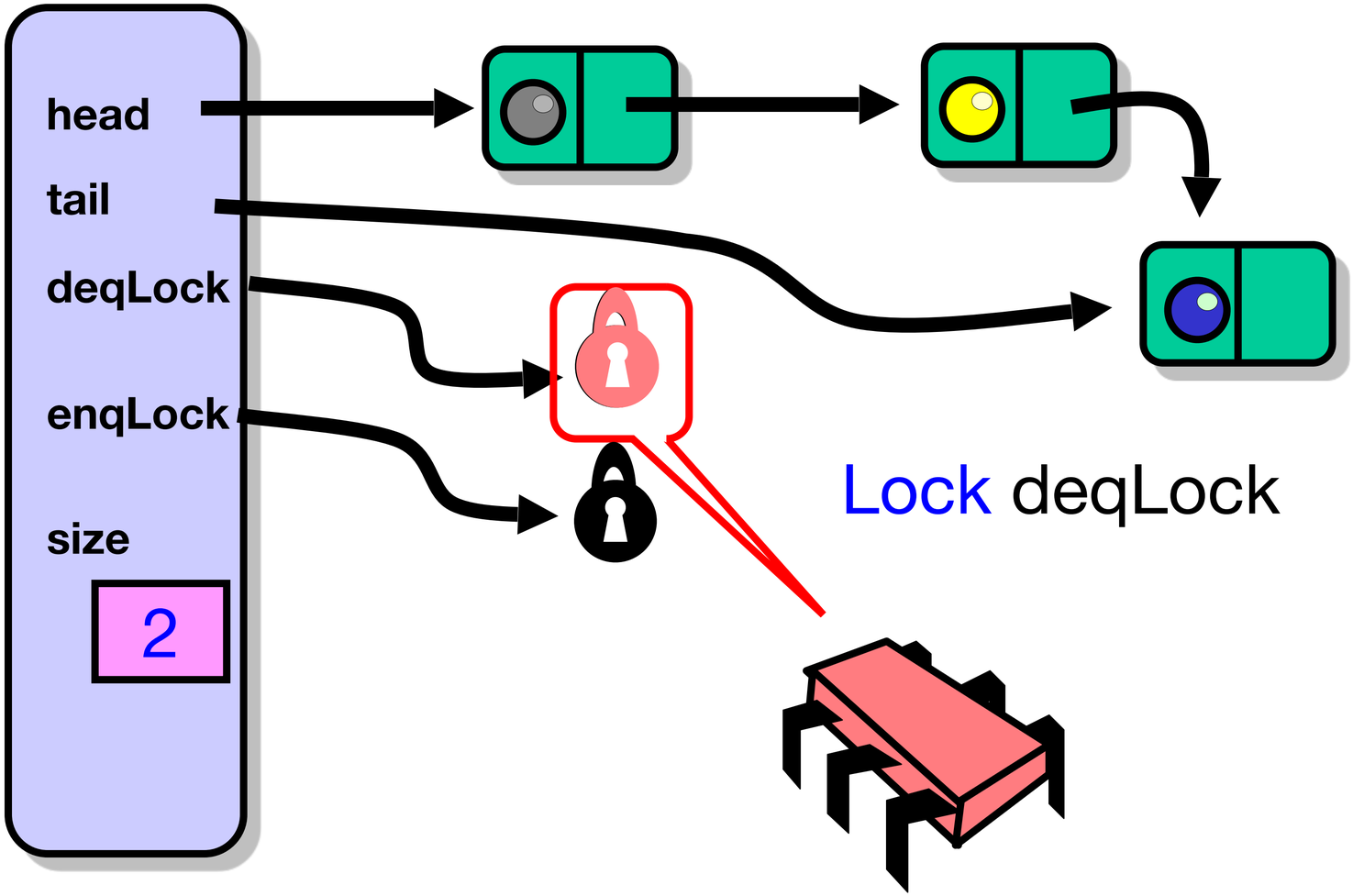
Unsuccessful Enqueuer



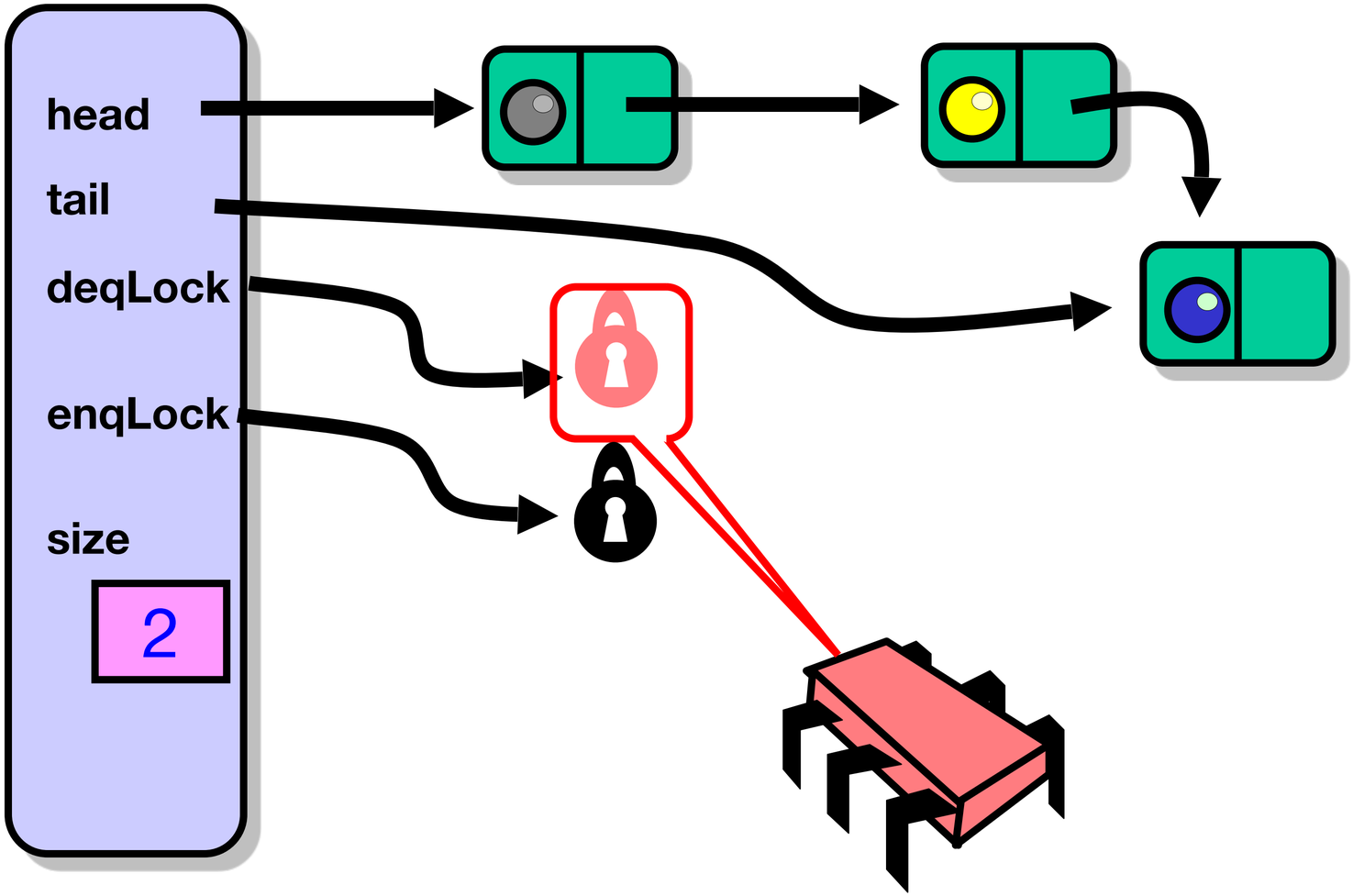
Dequeuer



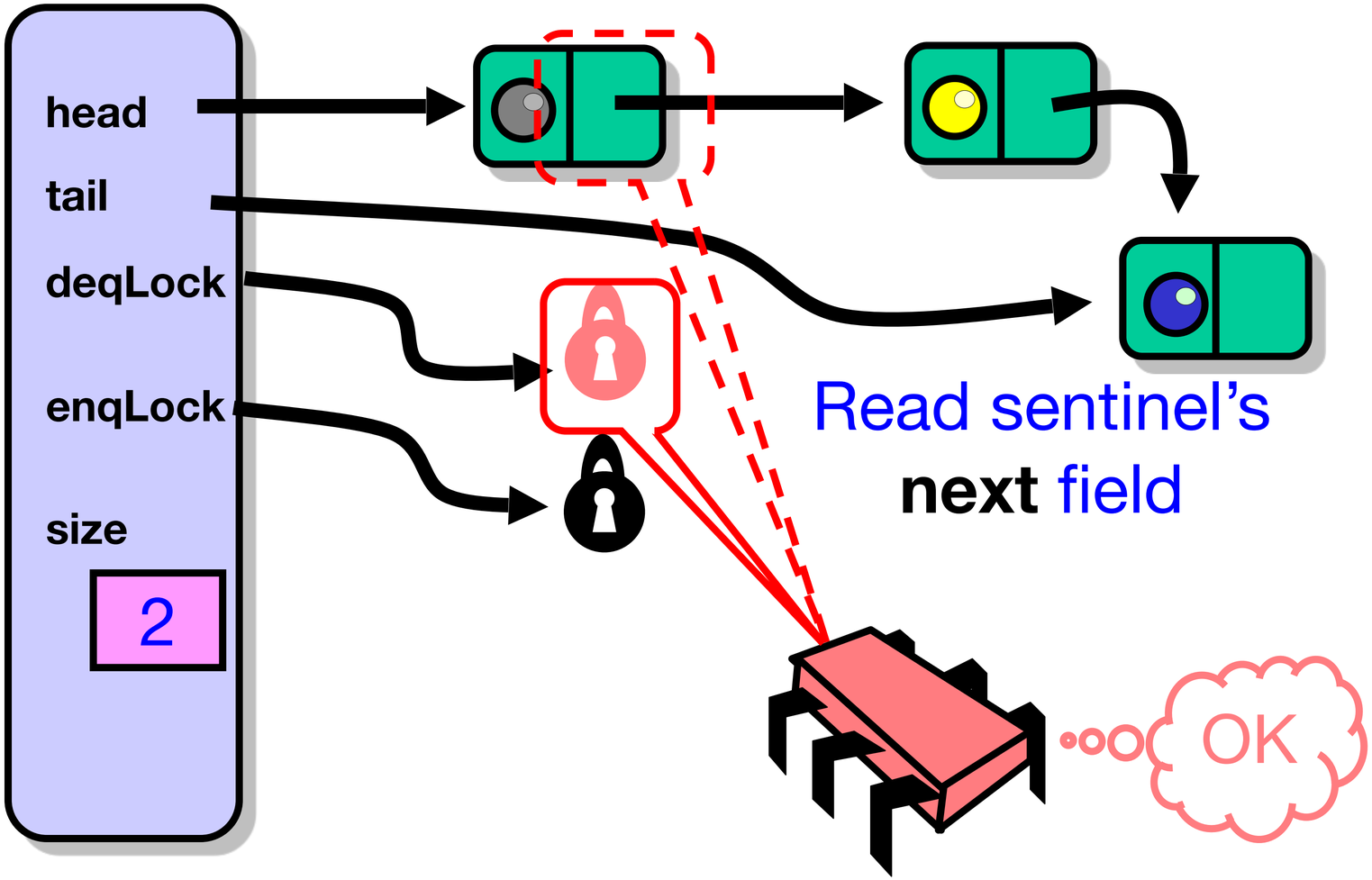
Dequeuer



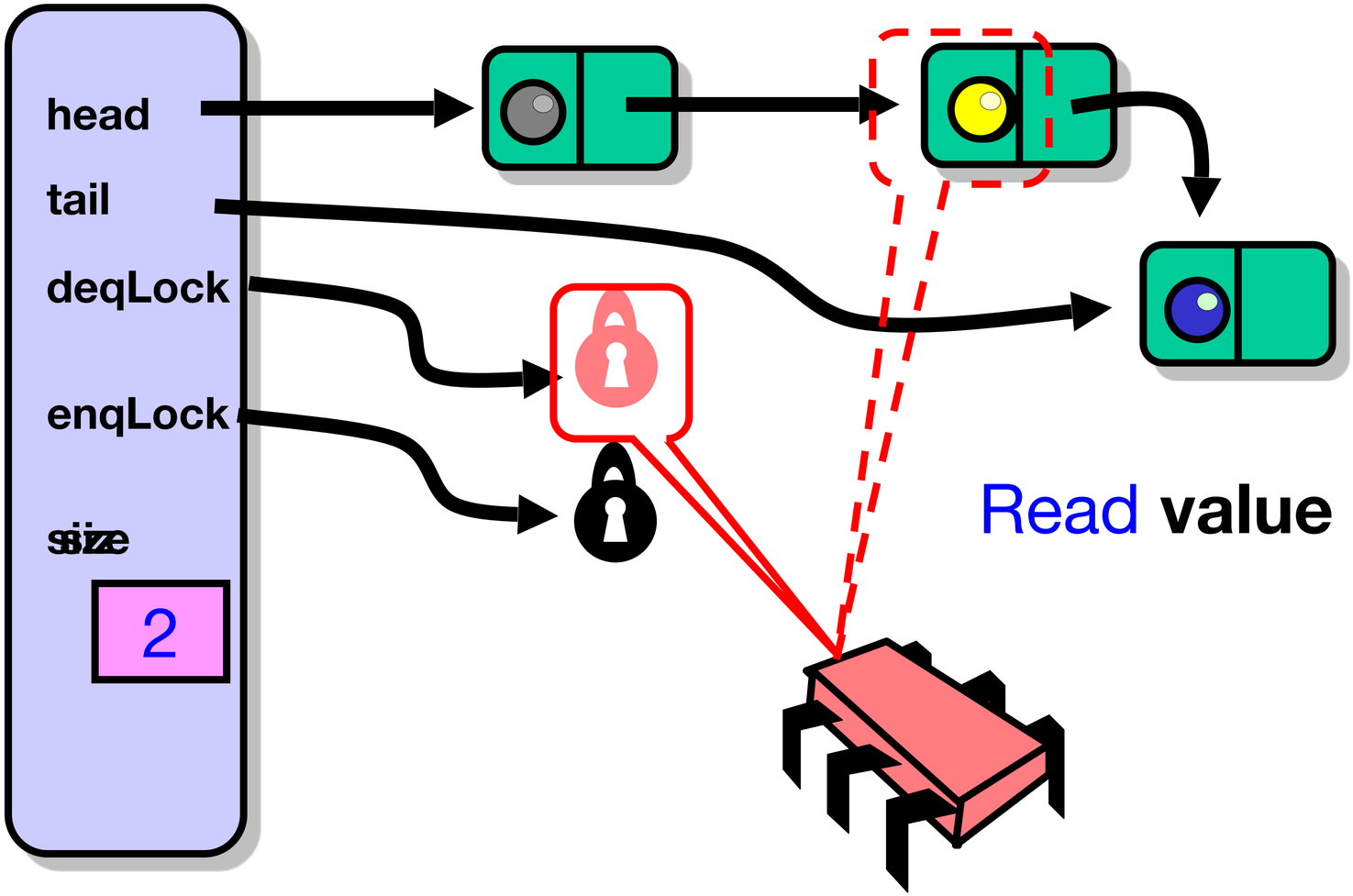
Dequeuer



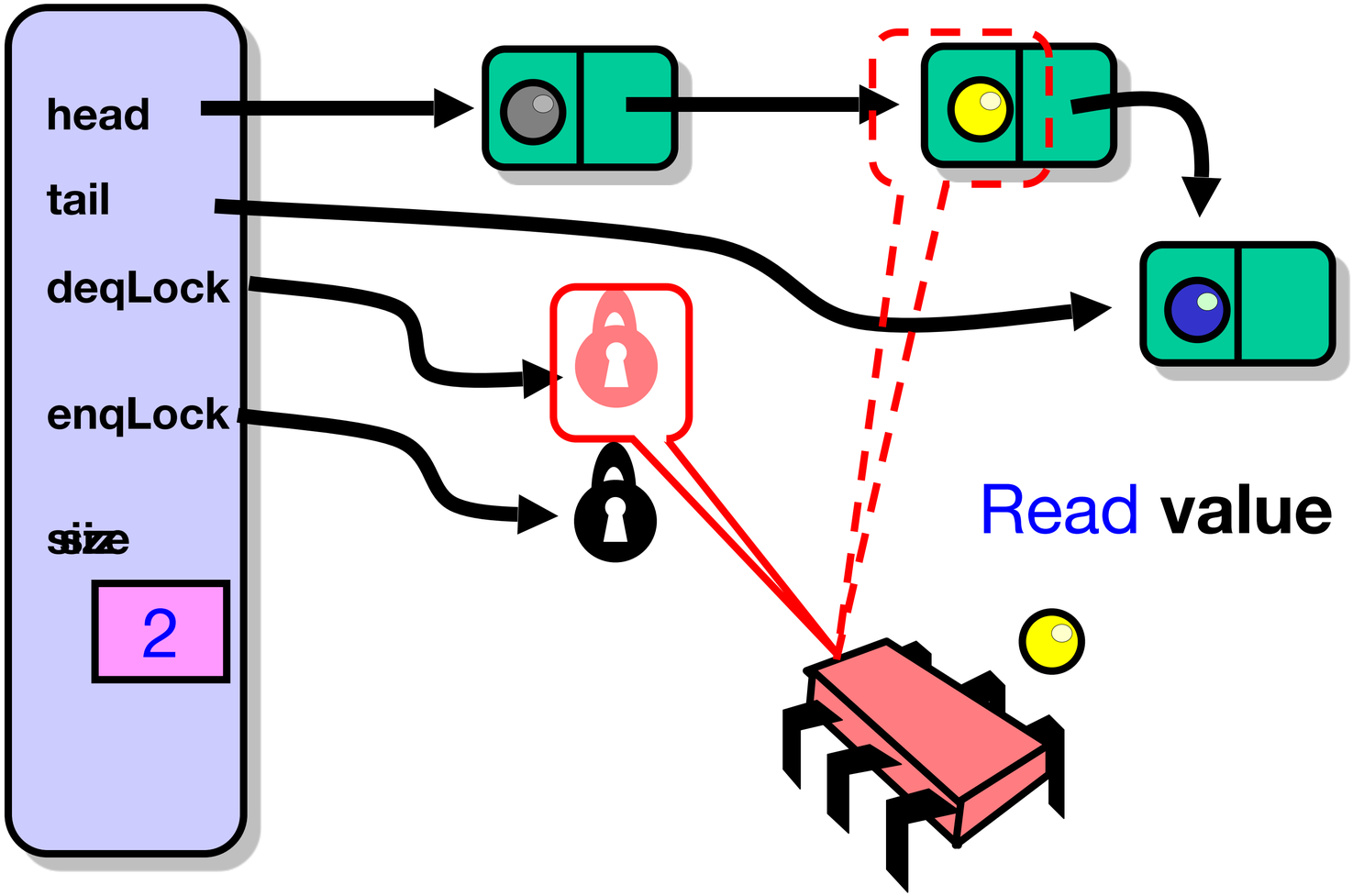
Dequeuer


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

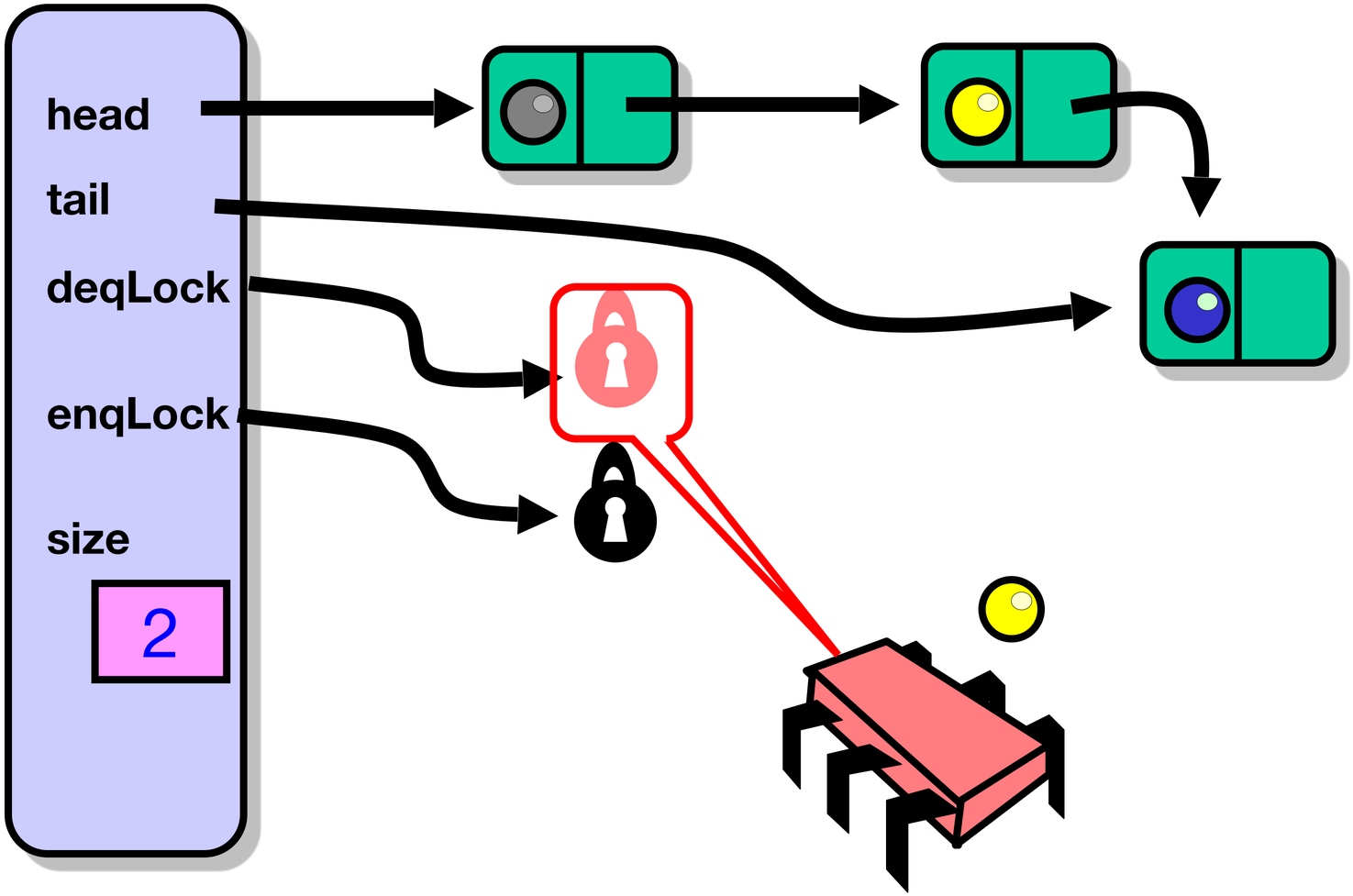
Dequeuer



Dequeuer

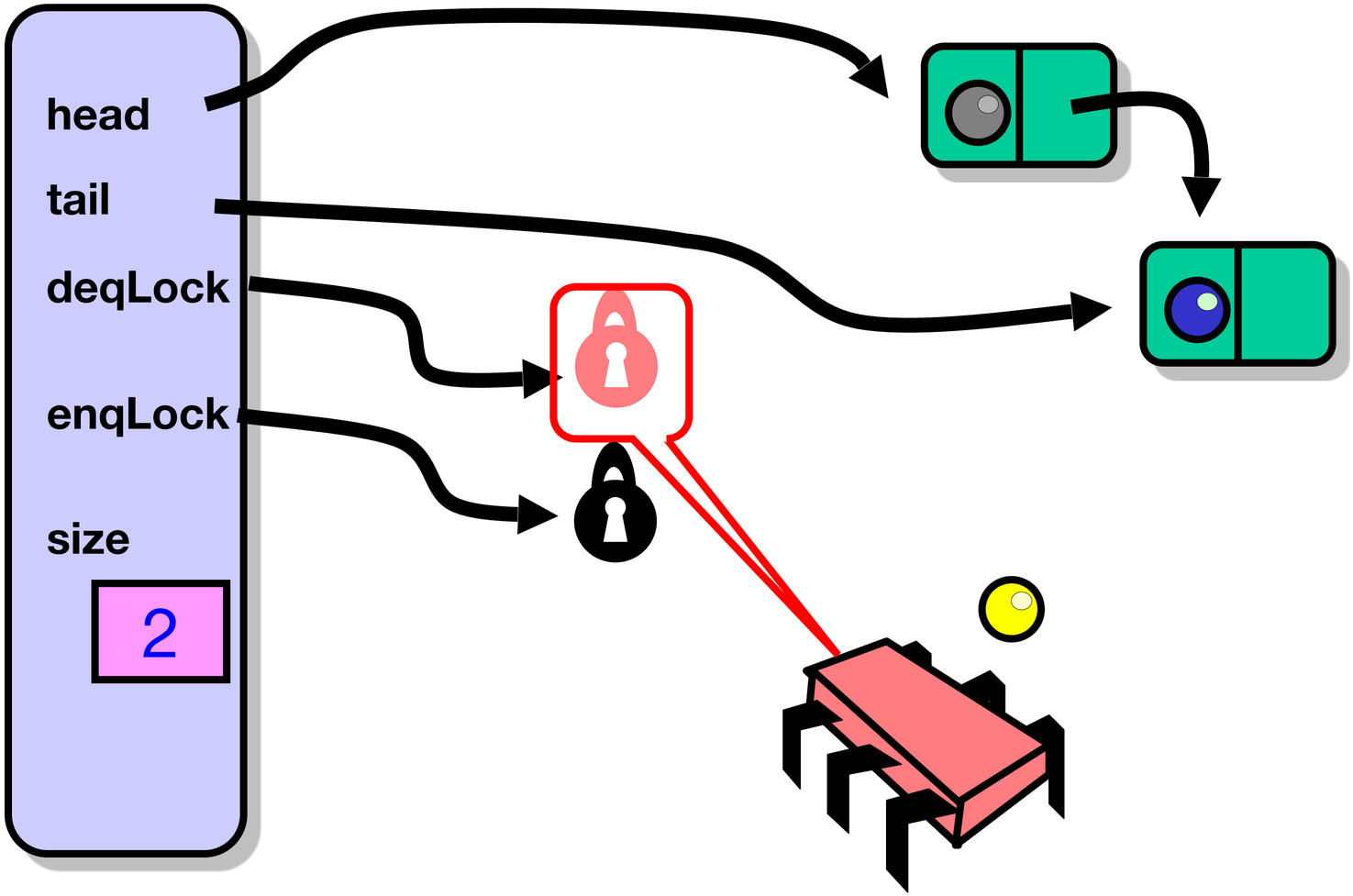


Dequeuer

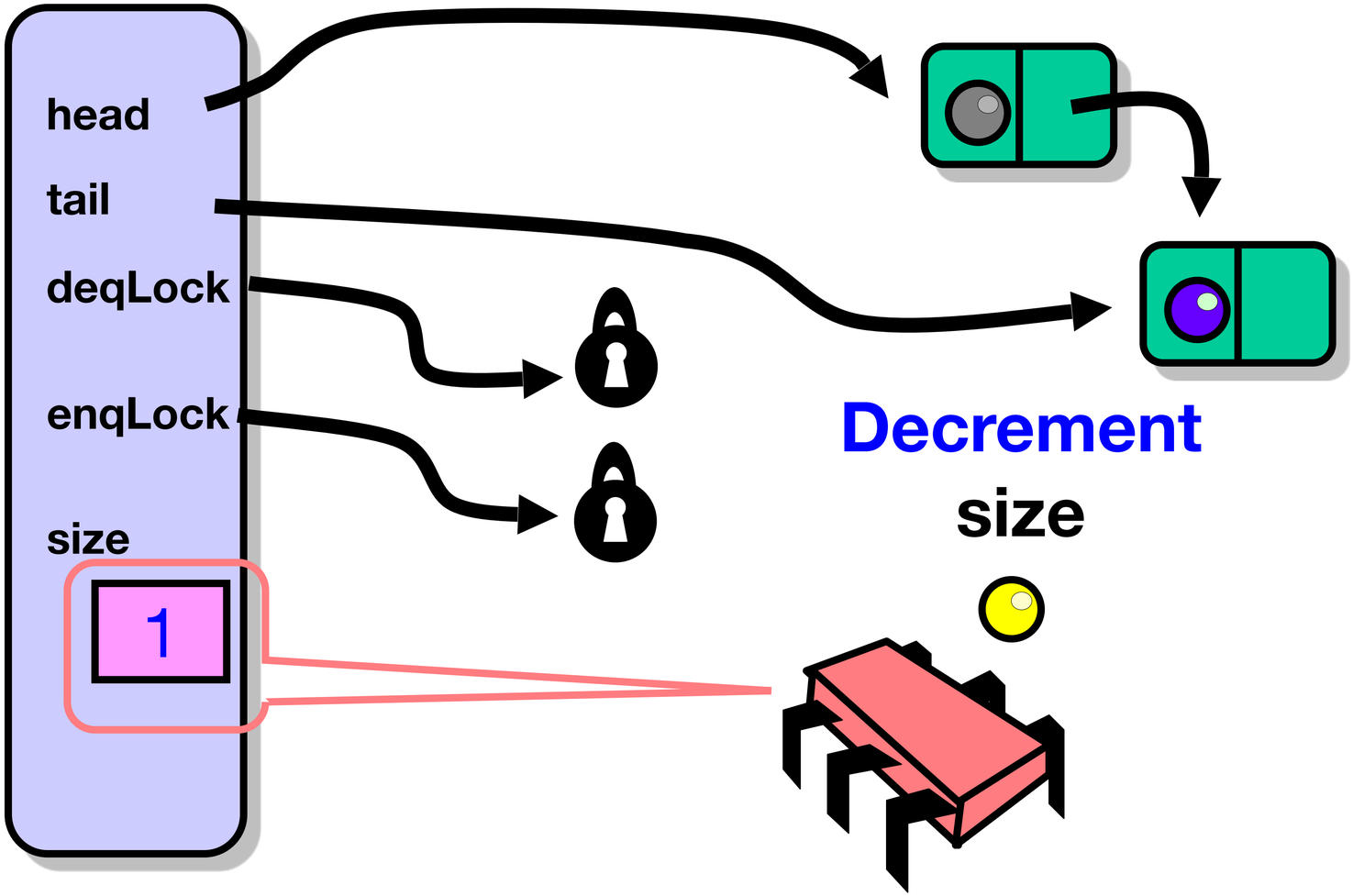


Dequeuer

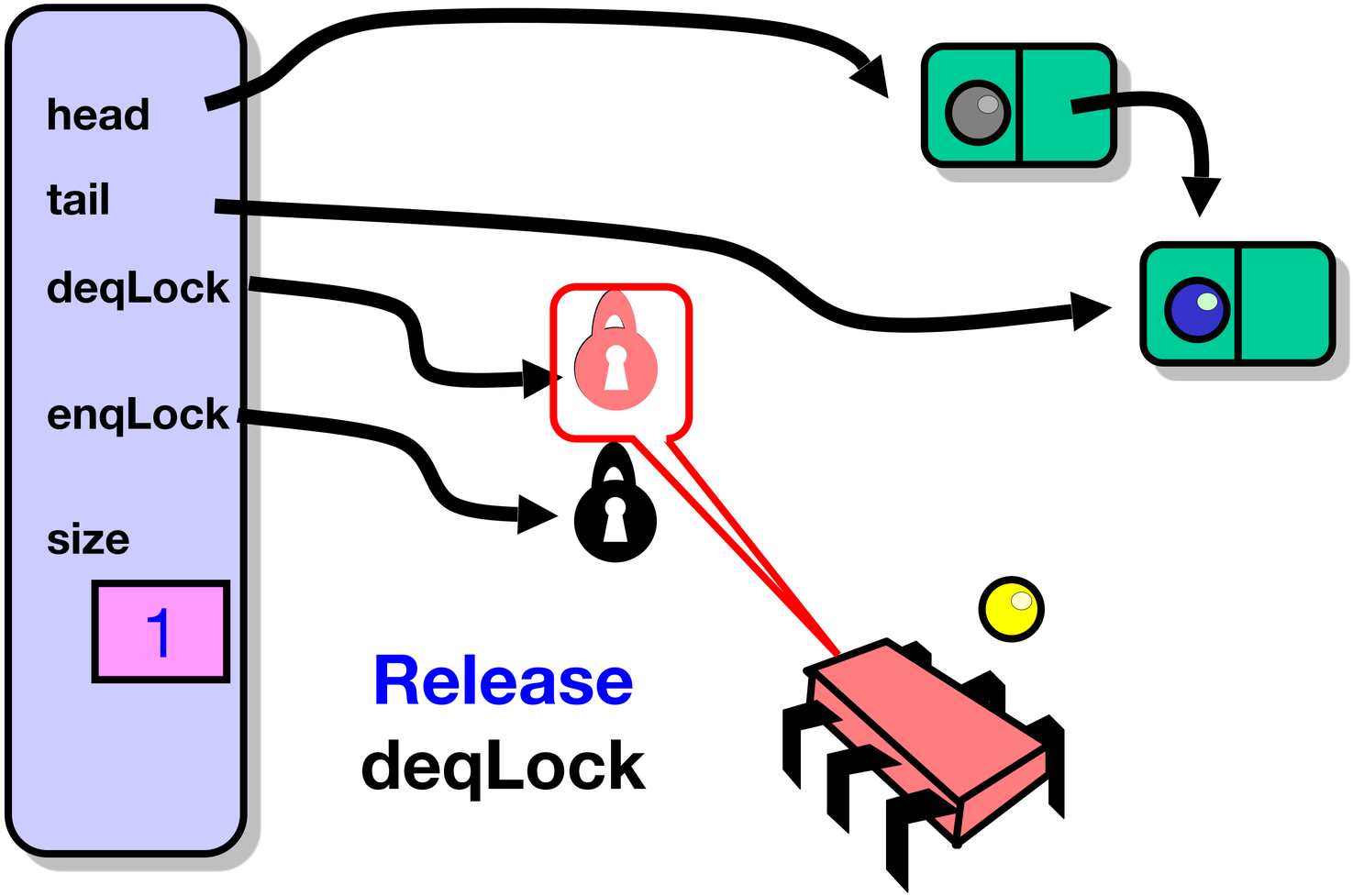
Make first Node
new sentinel



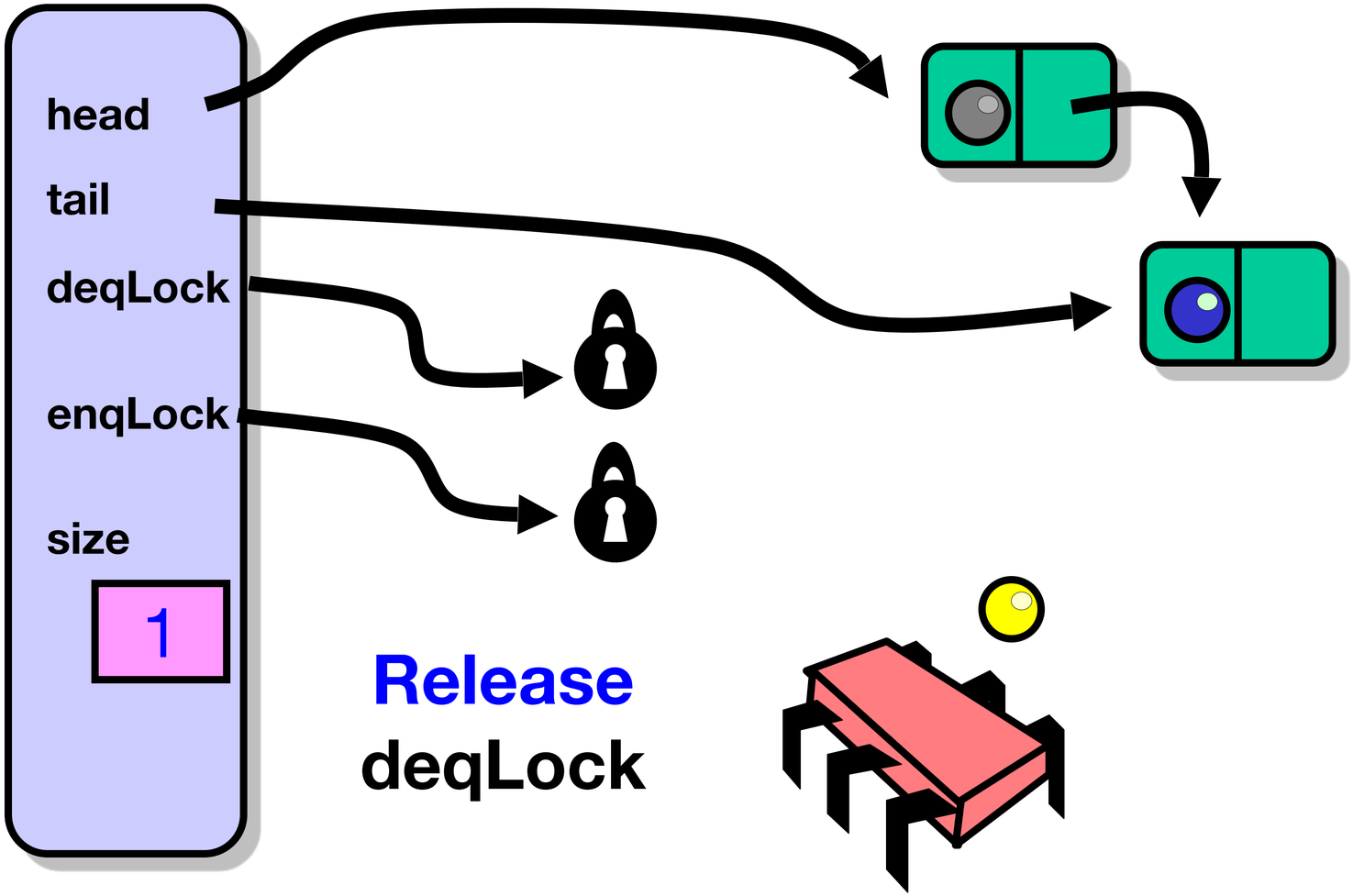
Dequeuer



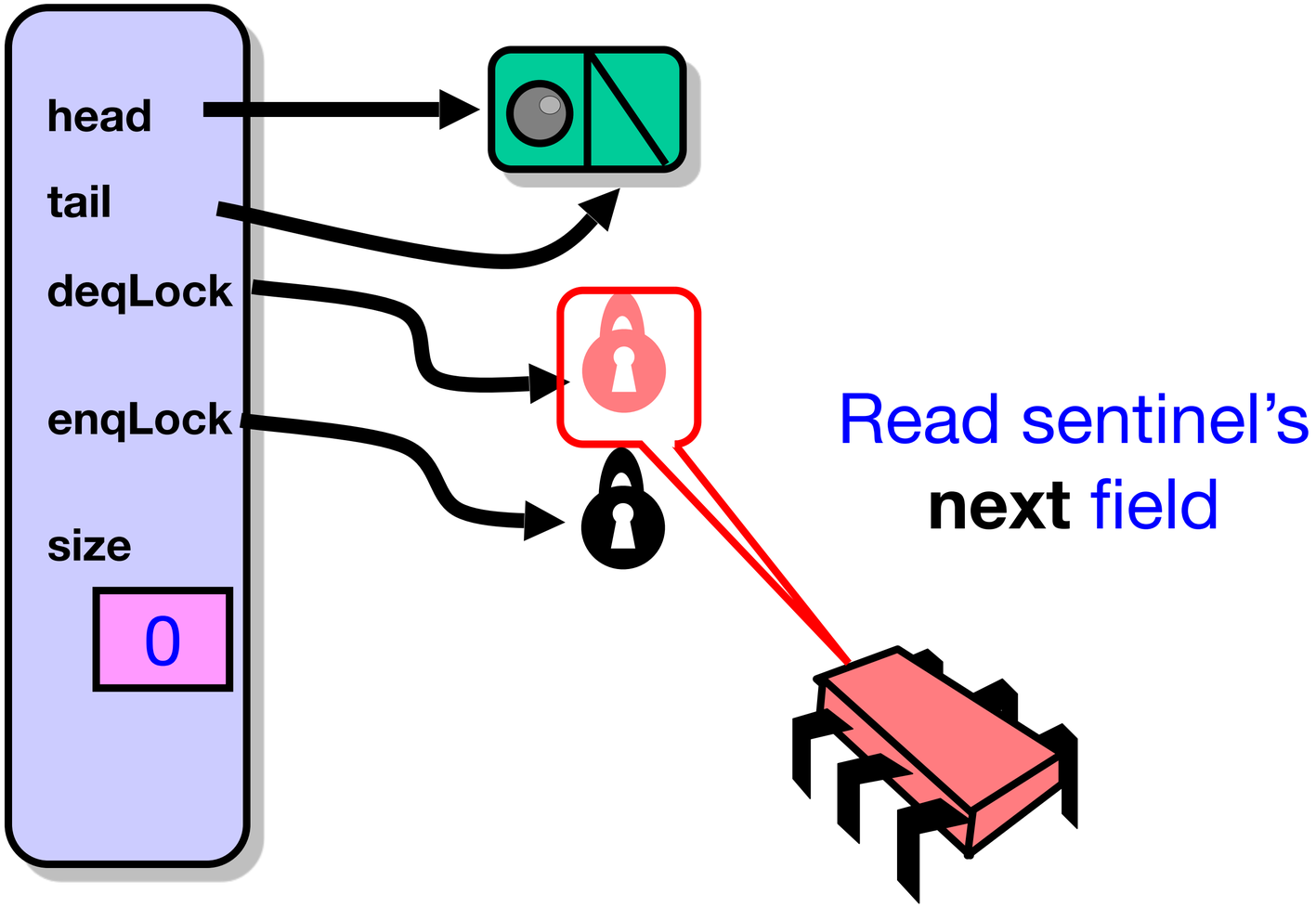
Dequeuer



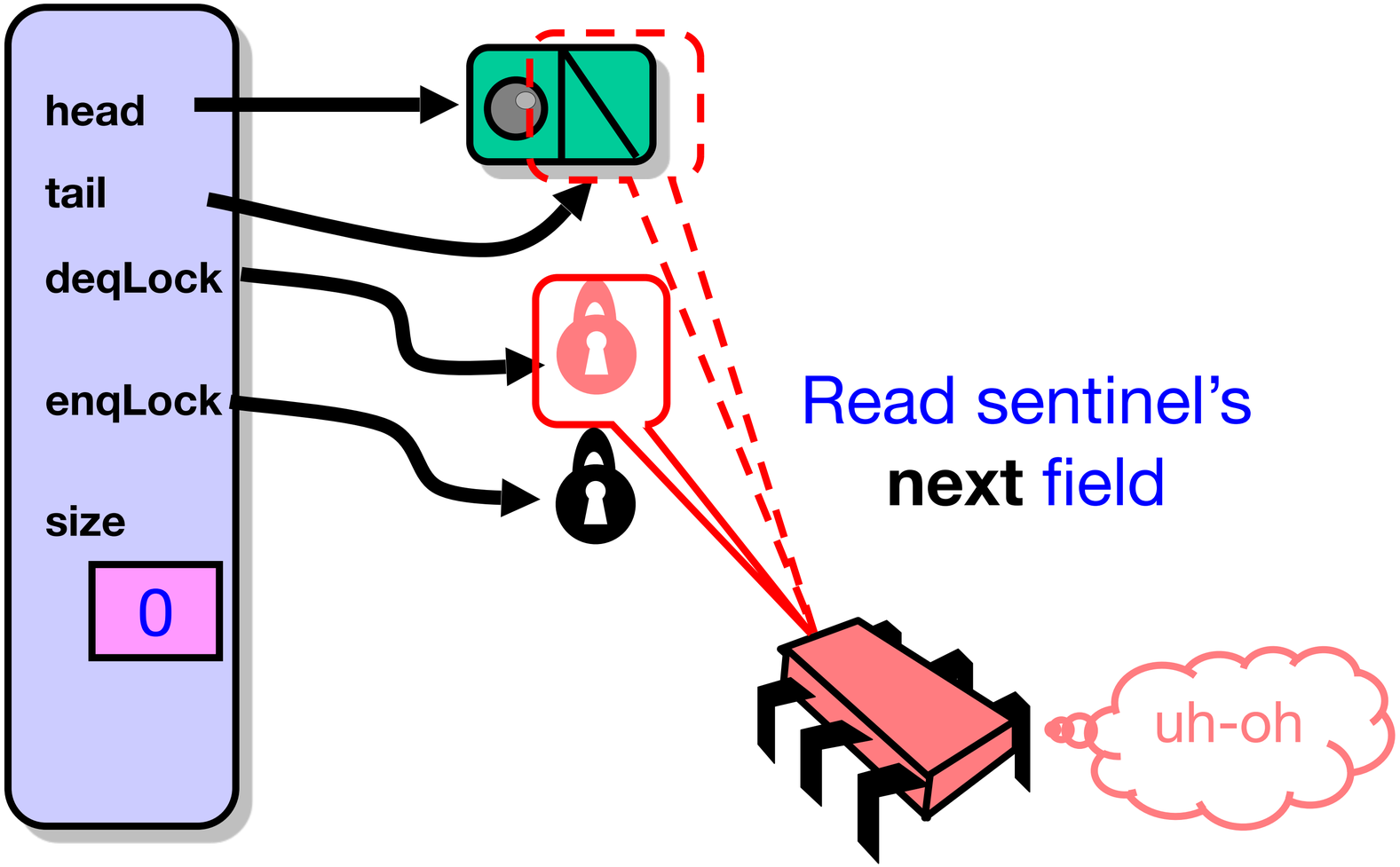
Dequeuer



Unsuccessful Dequeuer



Unsuccessful Dequeuer



Walk through the code

bounded_queue.ml

Note that the 2nd edition of the code is different from the 1st edition. We are using the 2nd edition.

The `enq()` and `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**

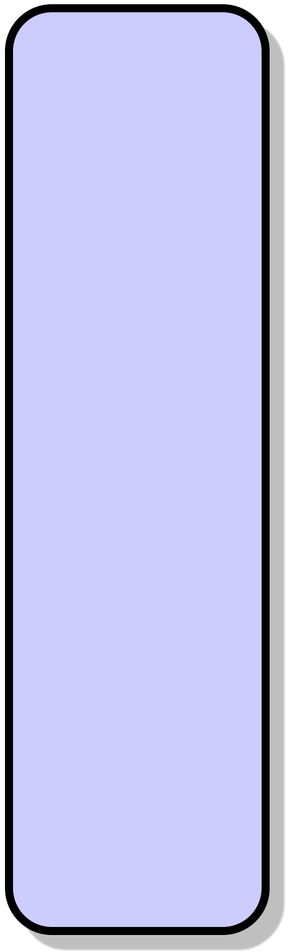
Split the counter

- Enqueuer increments **`enqSize`**
- Dequeuer decrements **`deqSize`**
- When the enqueuer hits capacity
  - Locks **`deqLock`**
  - Sets **`size = enqSize + DeqSize; DeqSize = 0`**
- Intermittent synchronization
  - Not with each method call
  - Need both locks! (careful …)

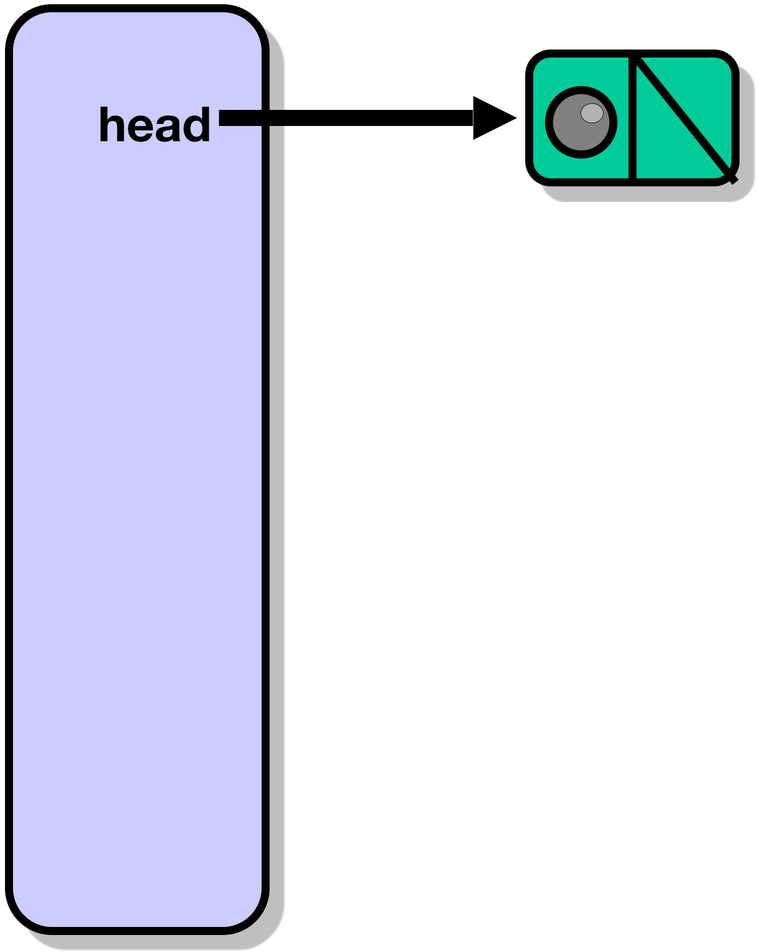
Walk through the code

bounded_queue_split_counter.ml

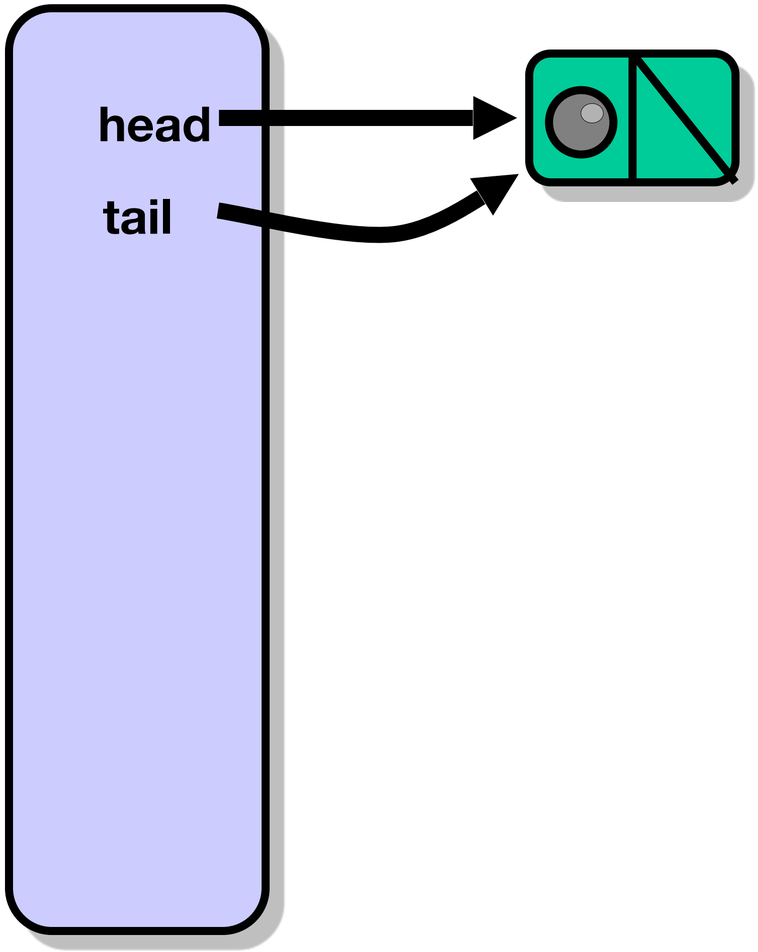
A Lock-free queue



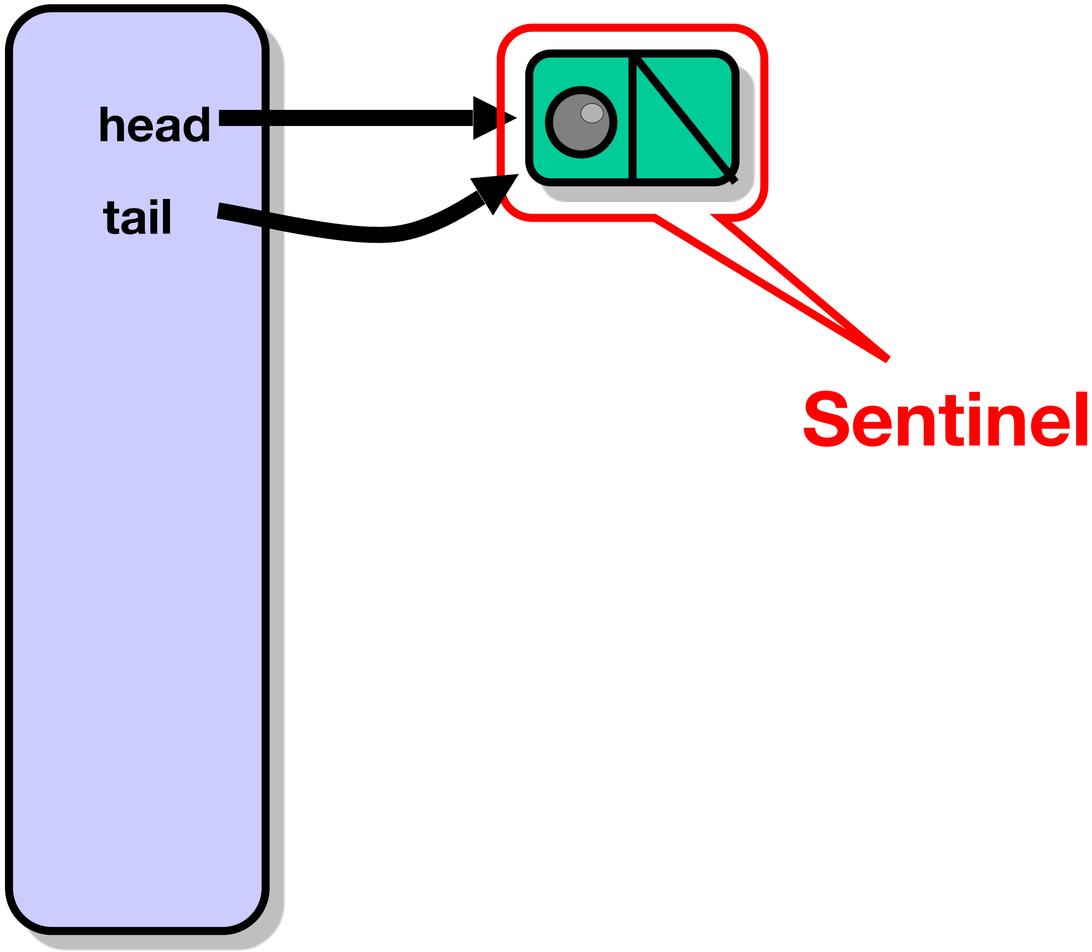
A Lock-free queue



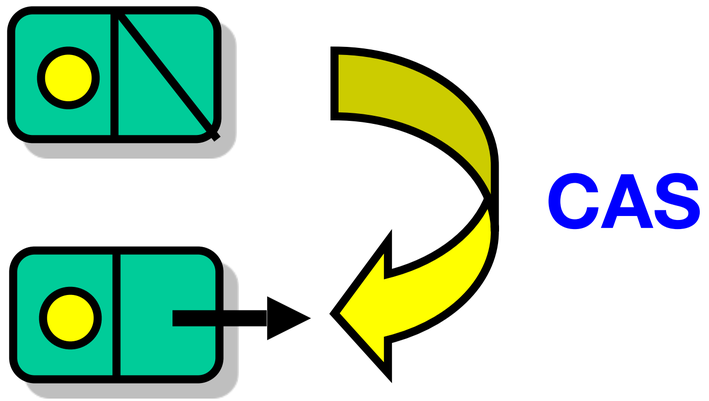
A Lock-free queue



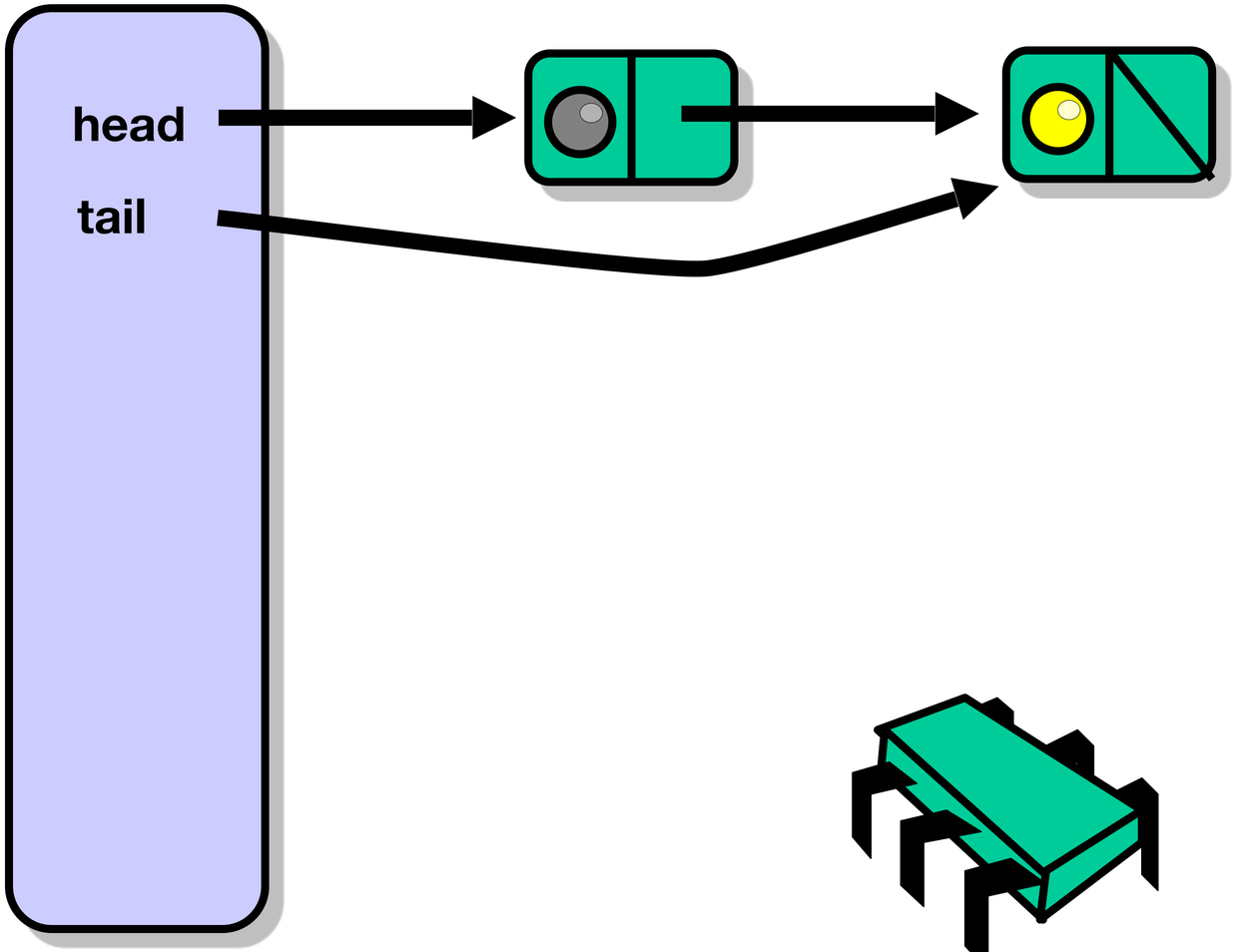
A Lock-free queue



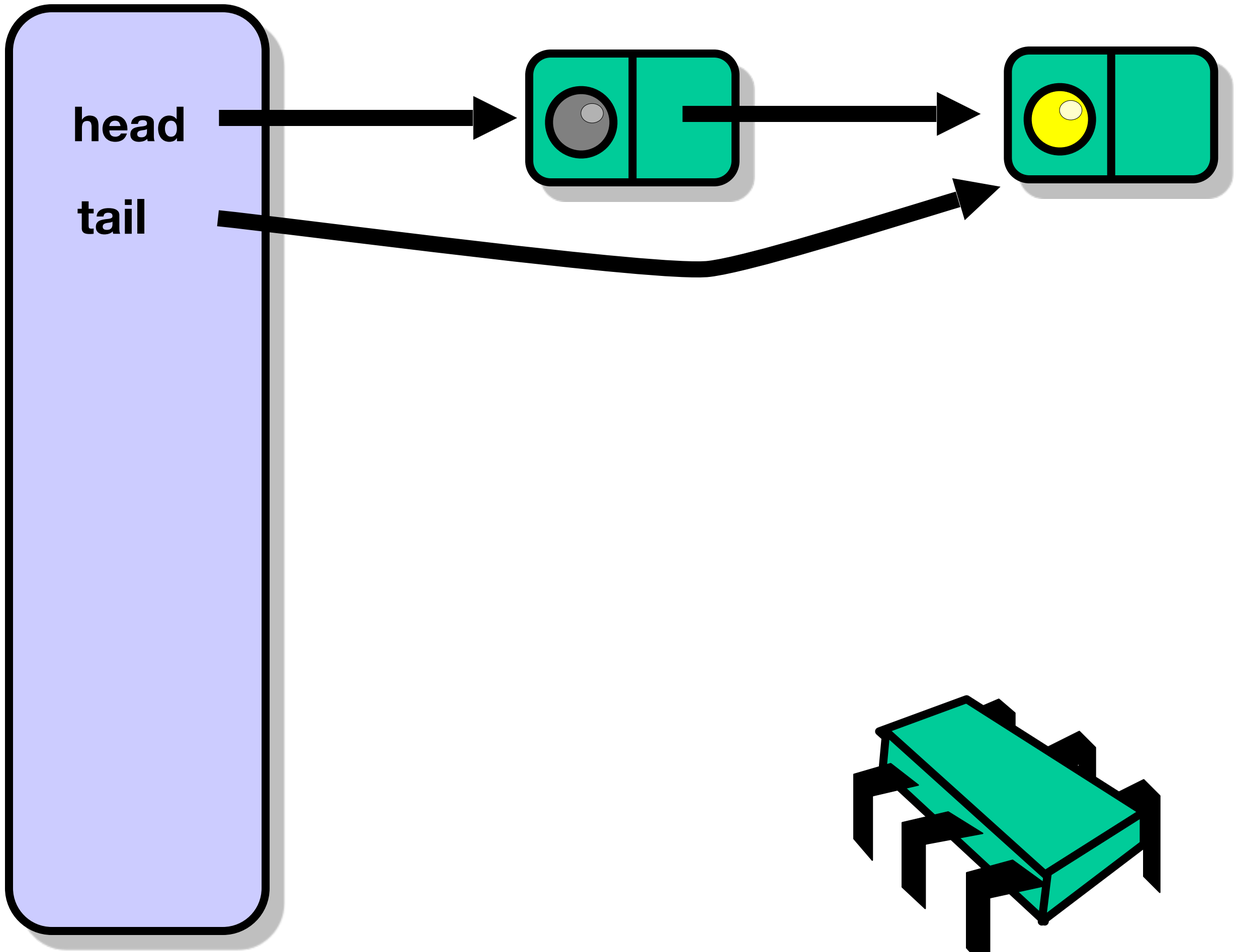
Compare and set



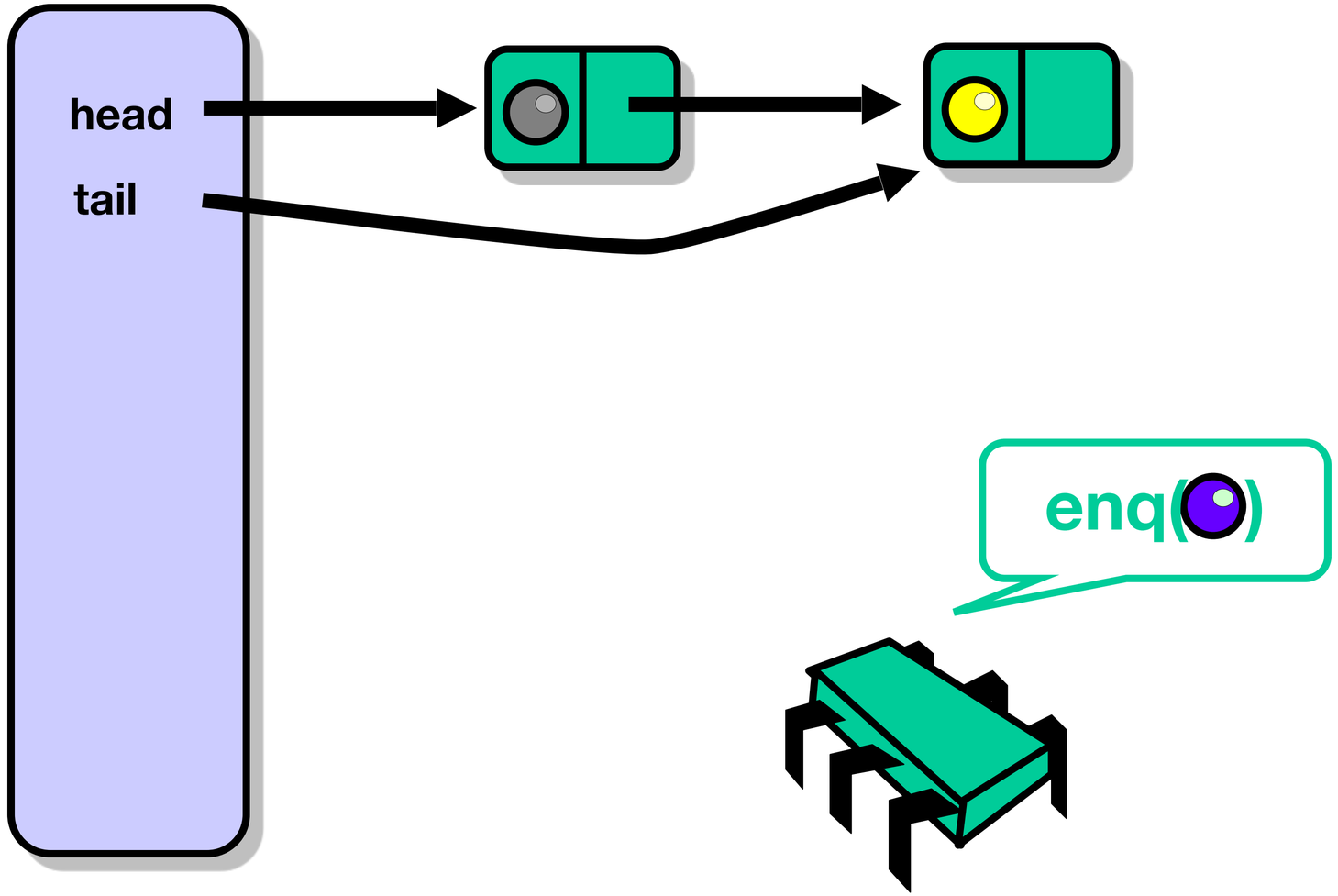
Enqueue



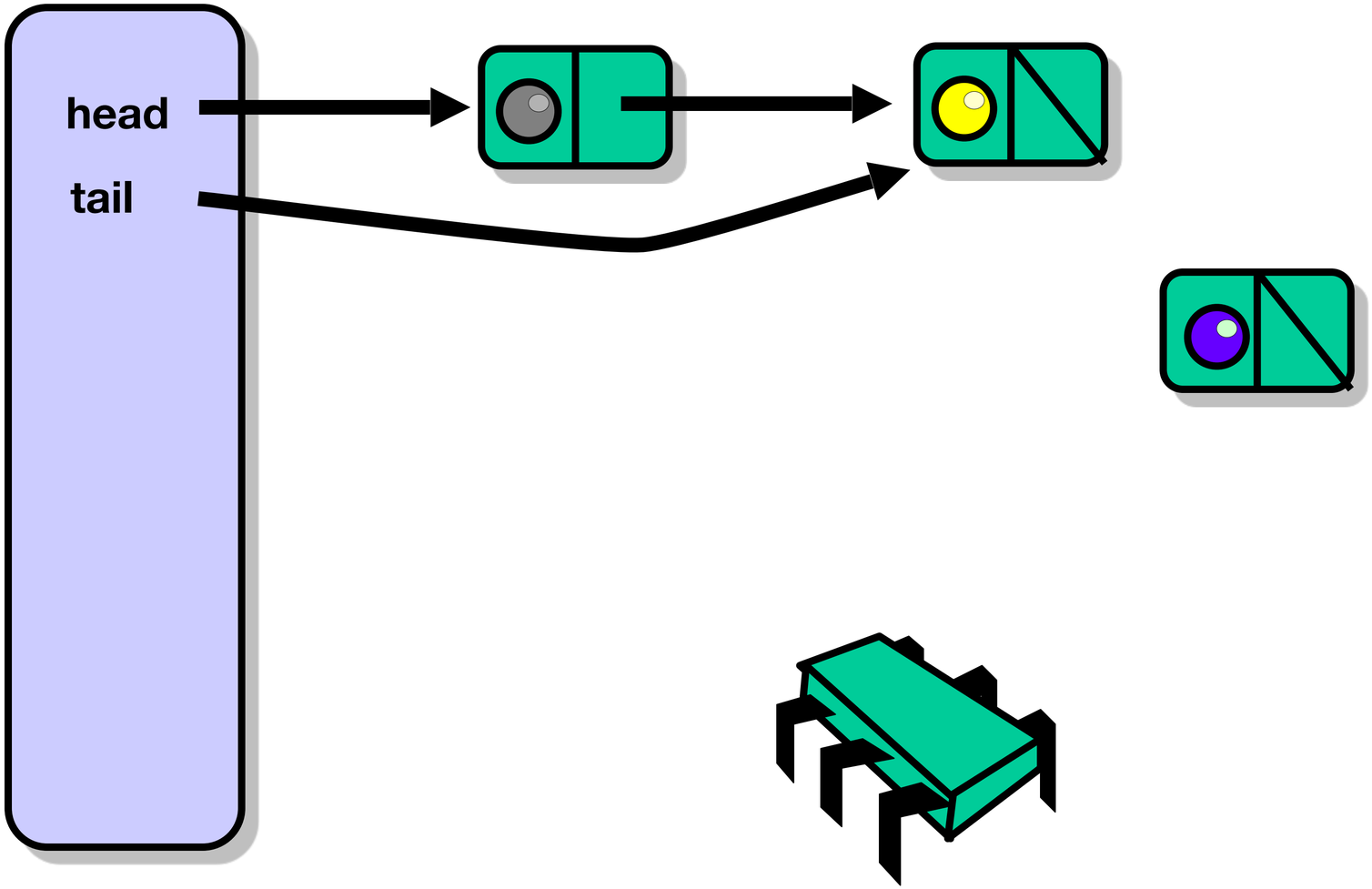
Enqueue



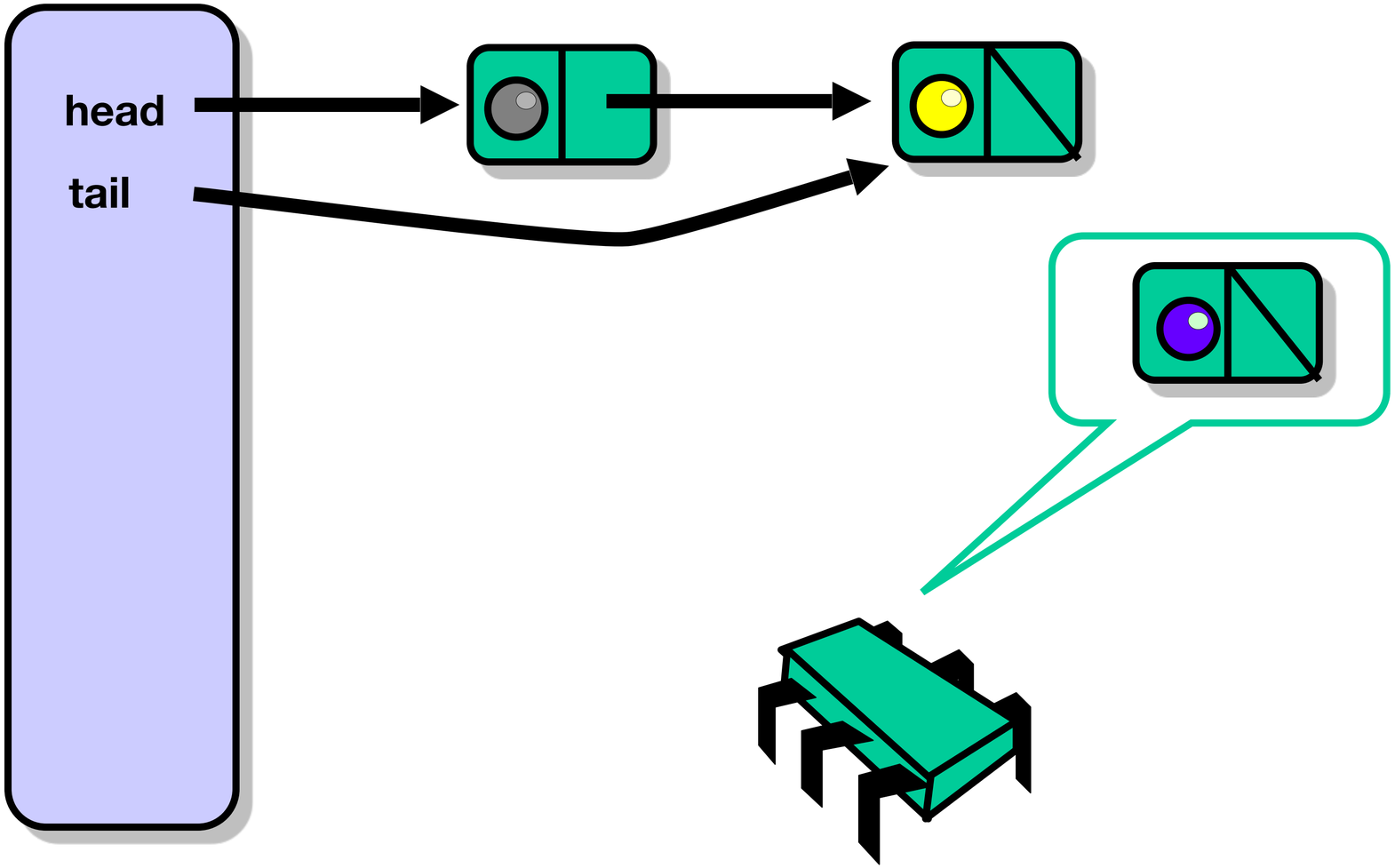
Enqueue



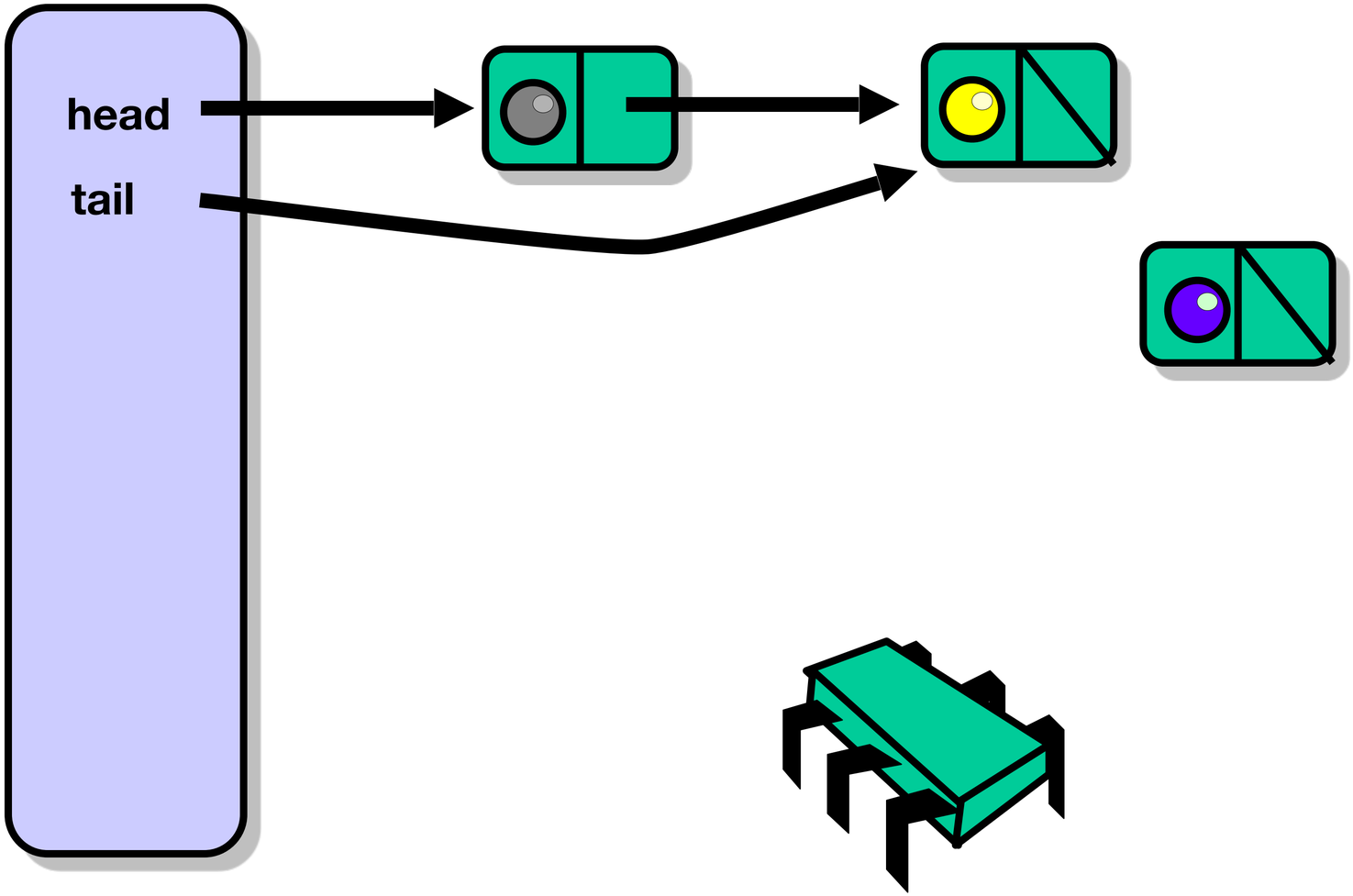
Enqueue



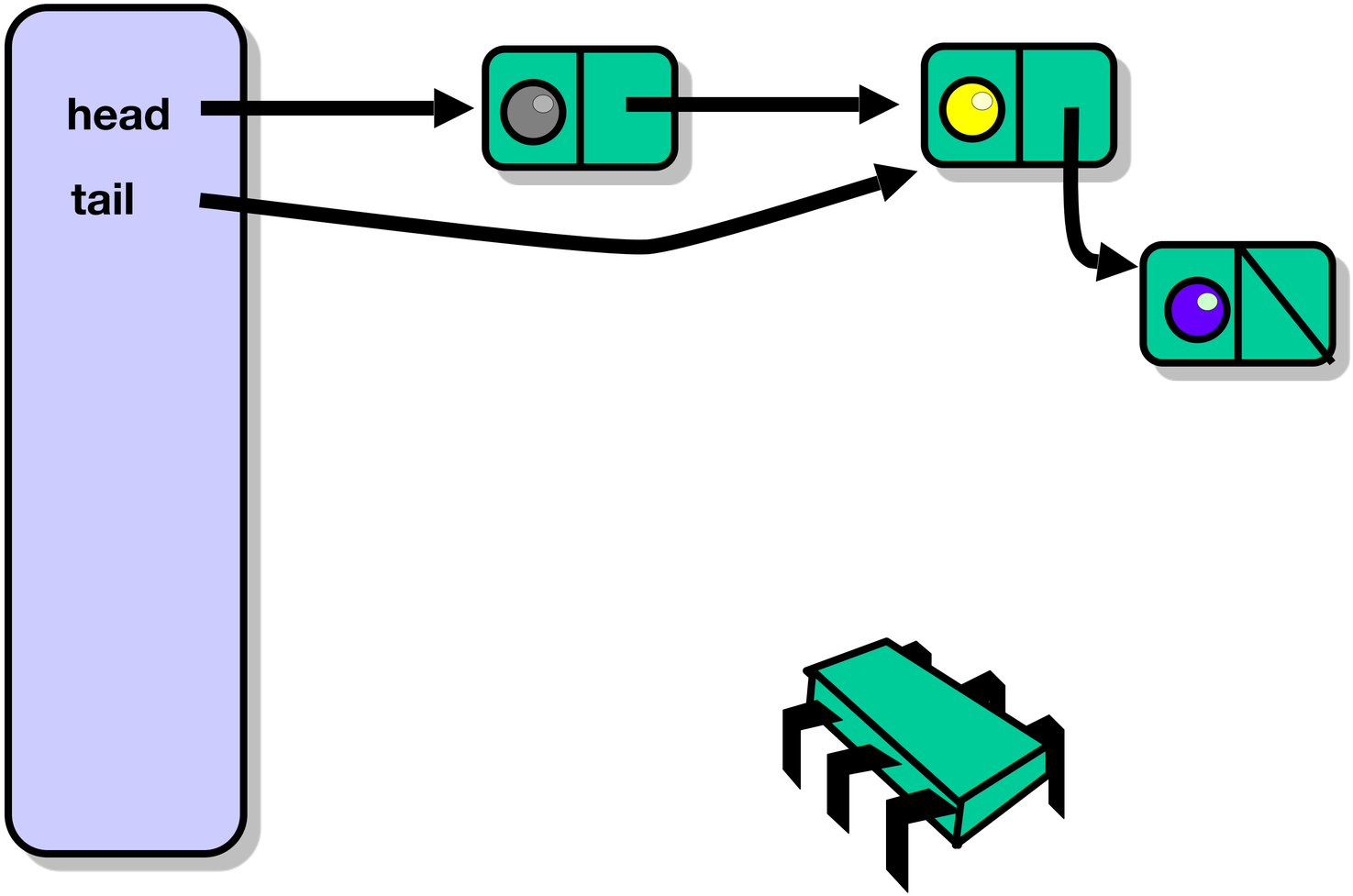
Enqueue



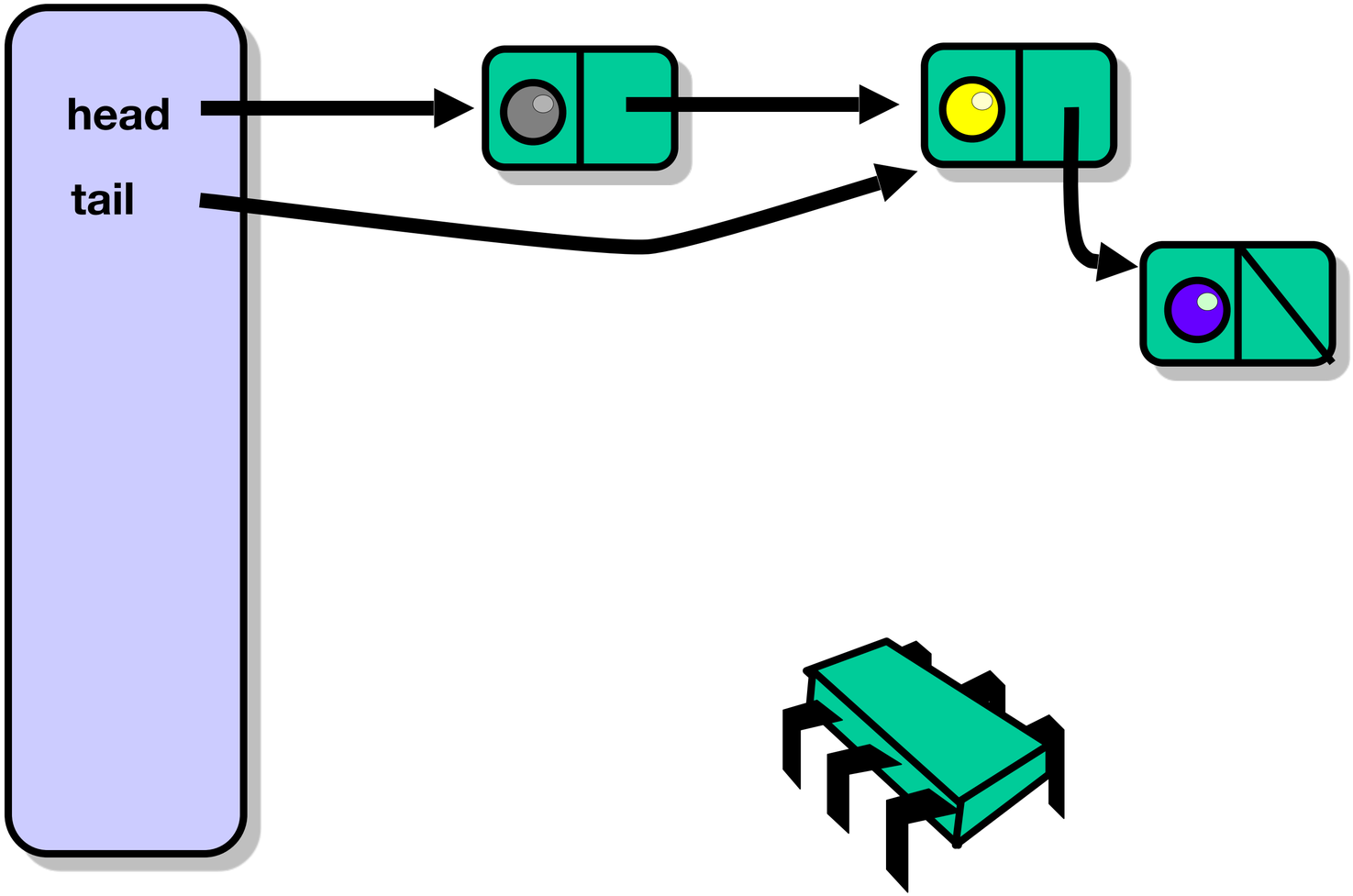
Logical Enqueue



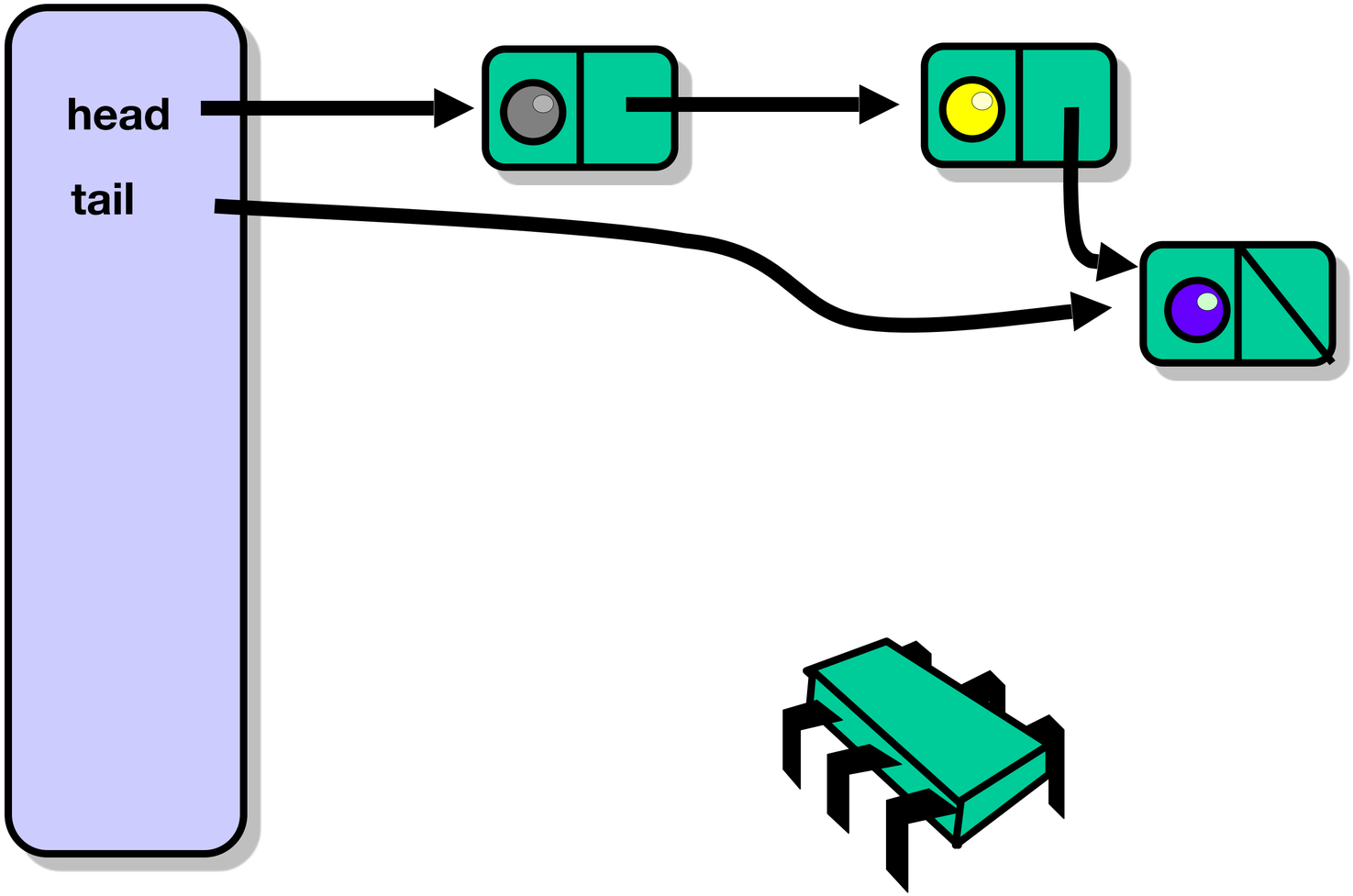
Logical Enqueue



Physical Enqueue



Physical Enqueue



Enqueue

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

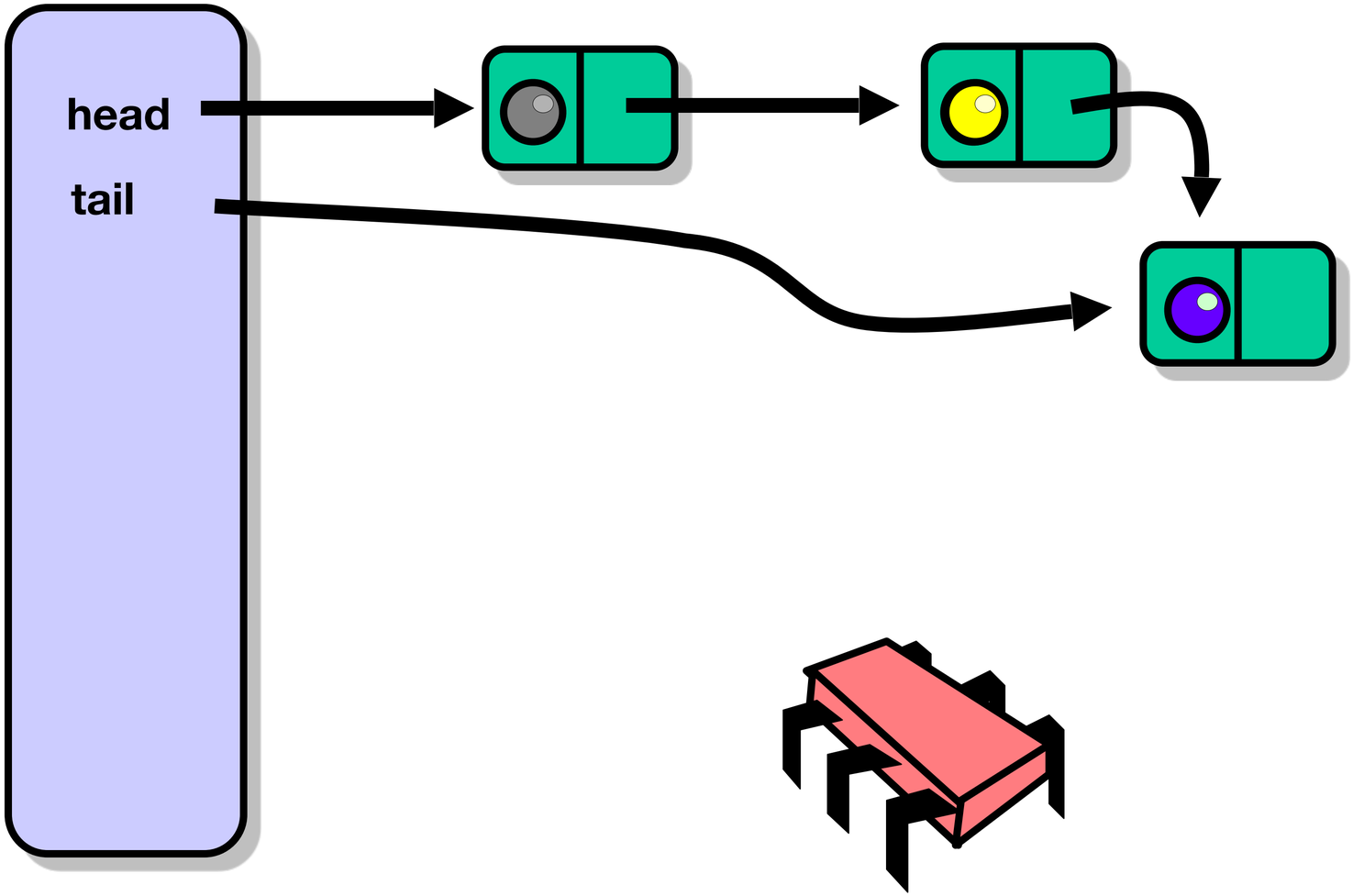
Enqueue

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

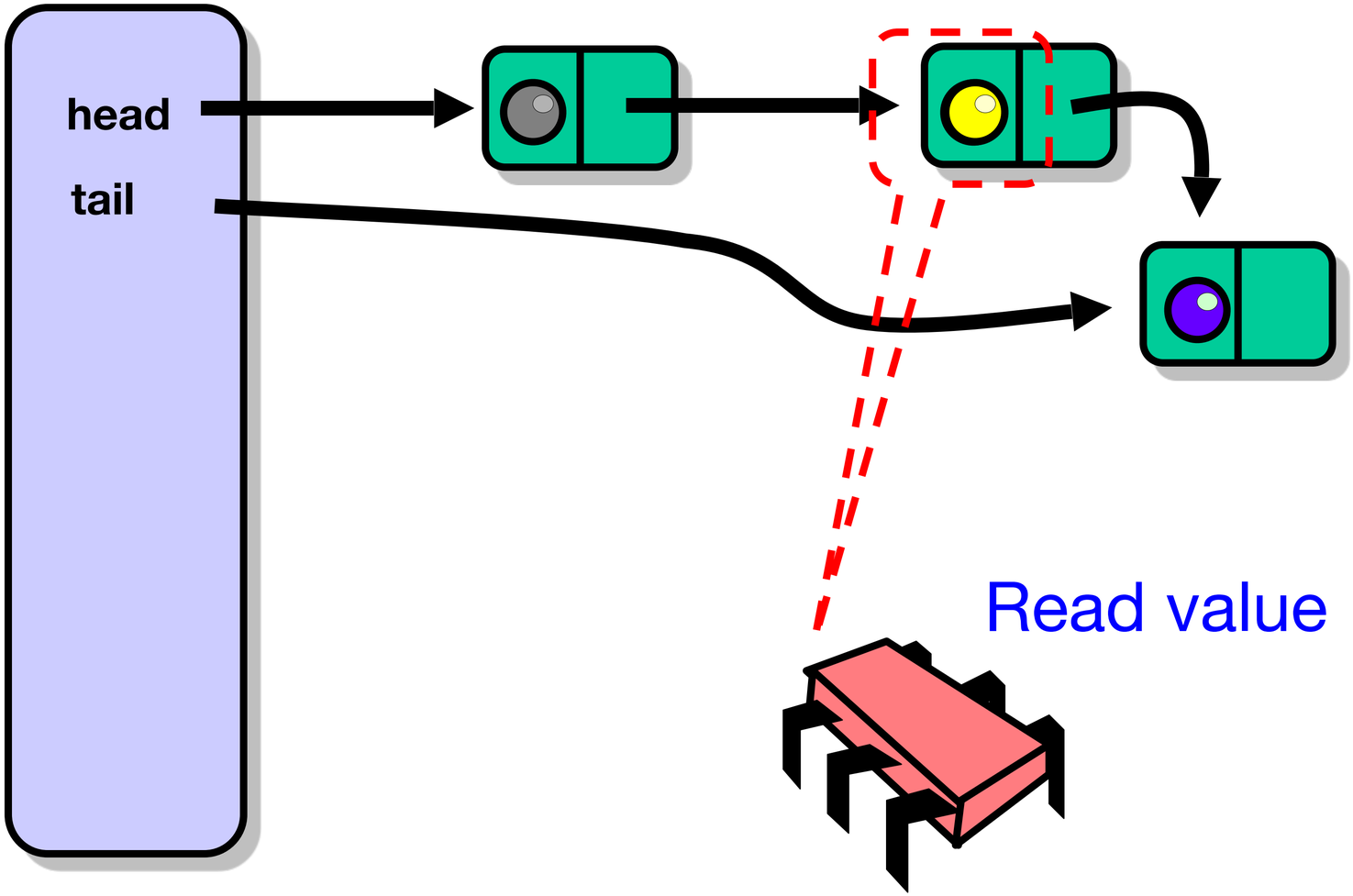
When CASes Fail

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

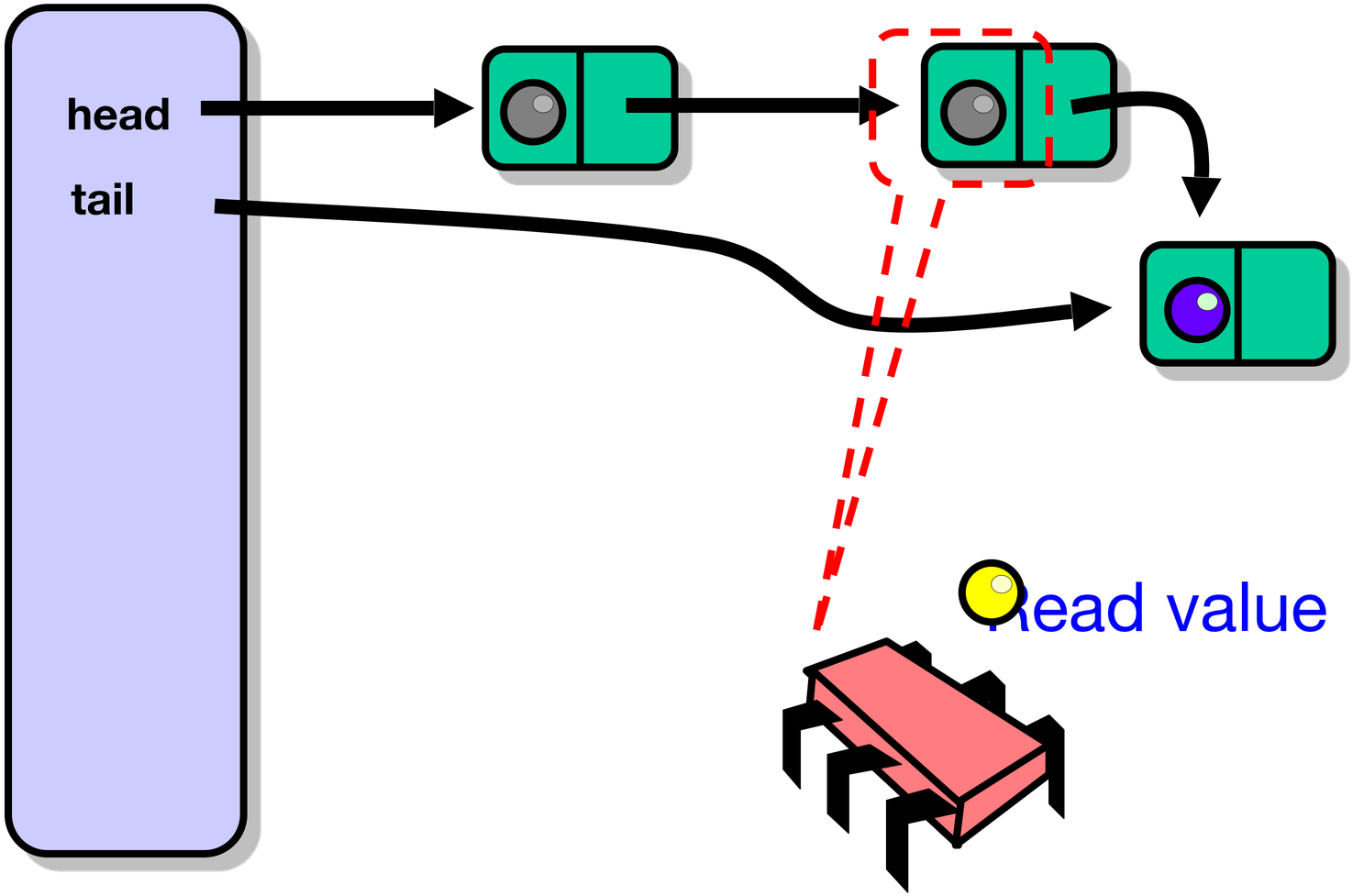
Dequeuer



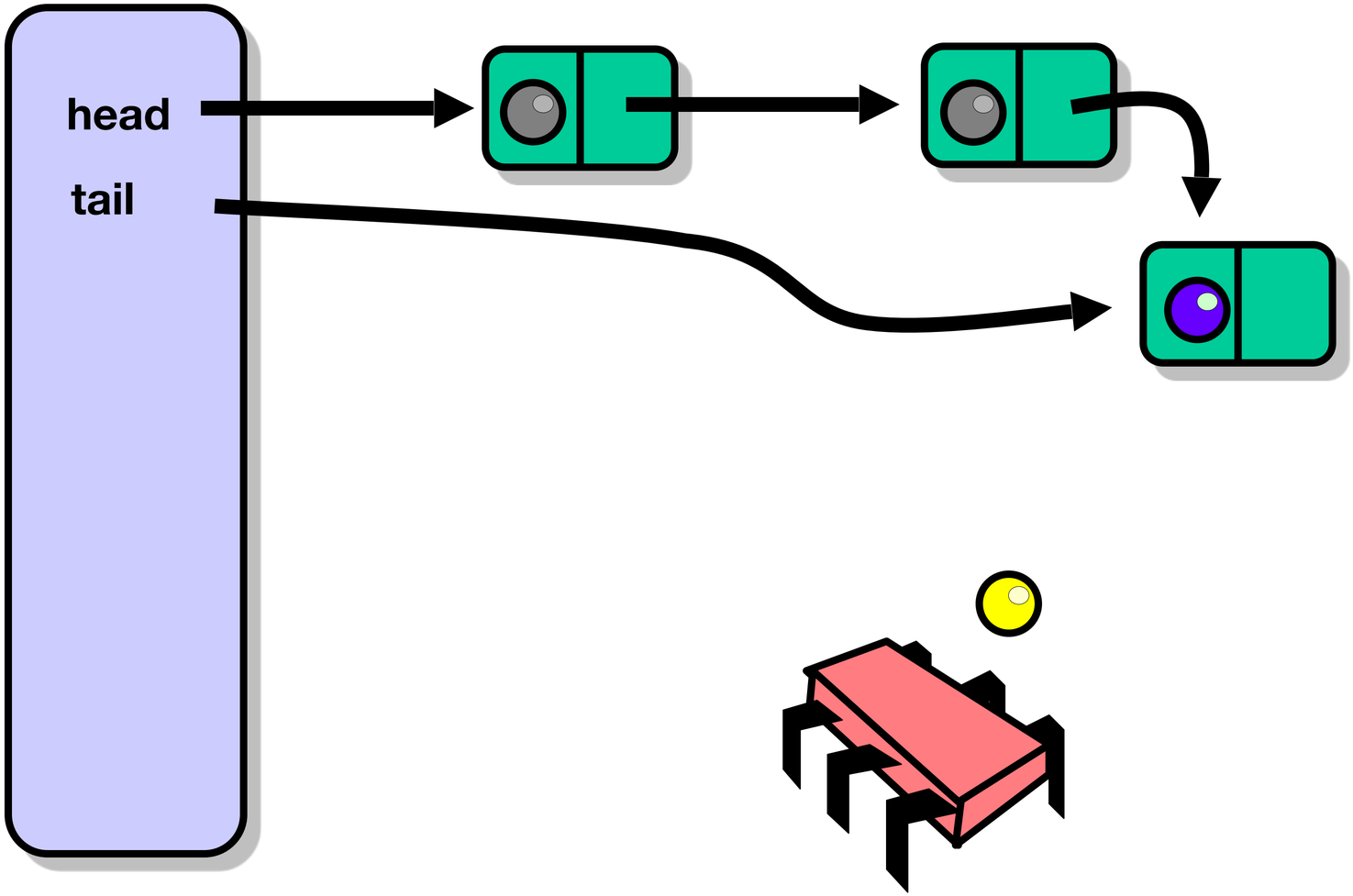
Dequeuer



Dequeuer

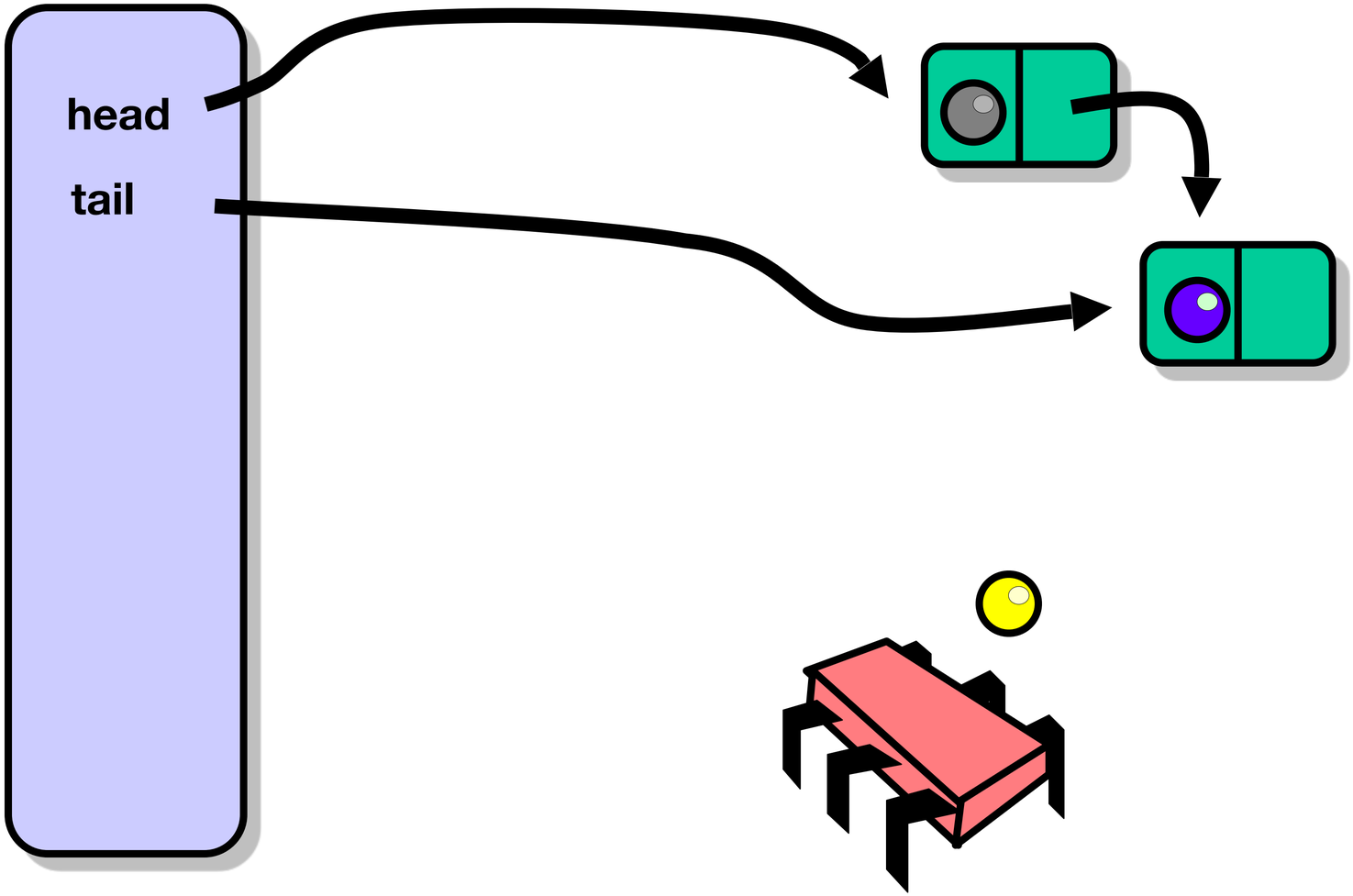


Dequeuer



Dequeuer

Make first Node
new sentinel



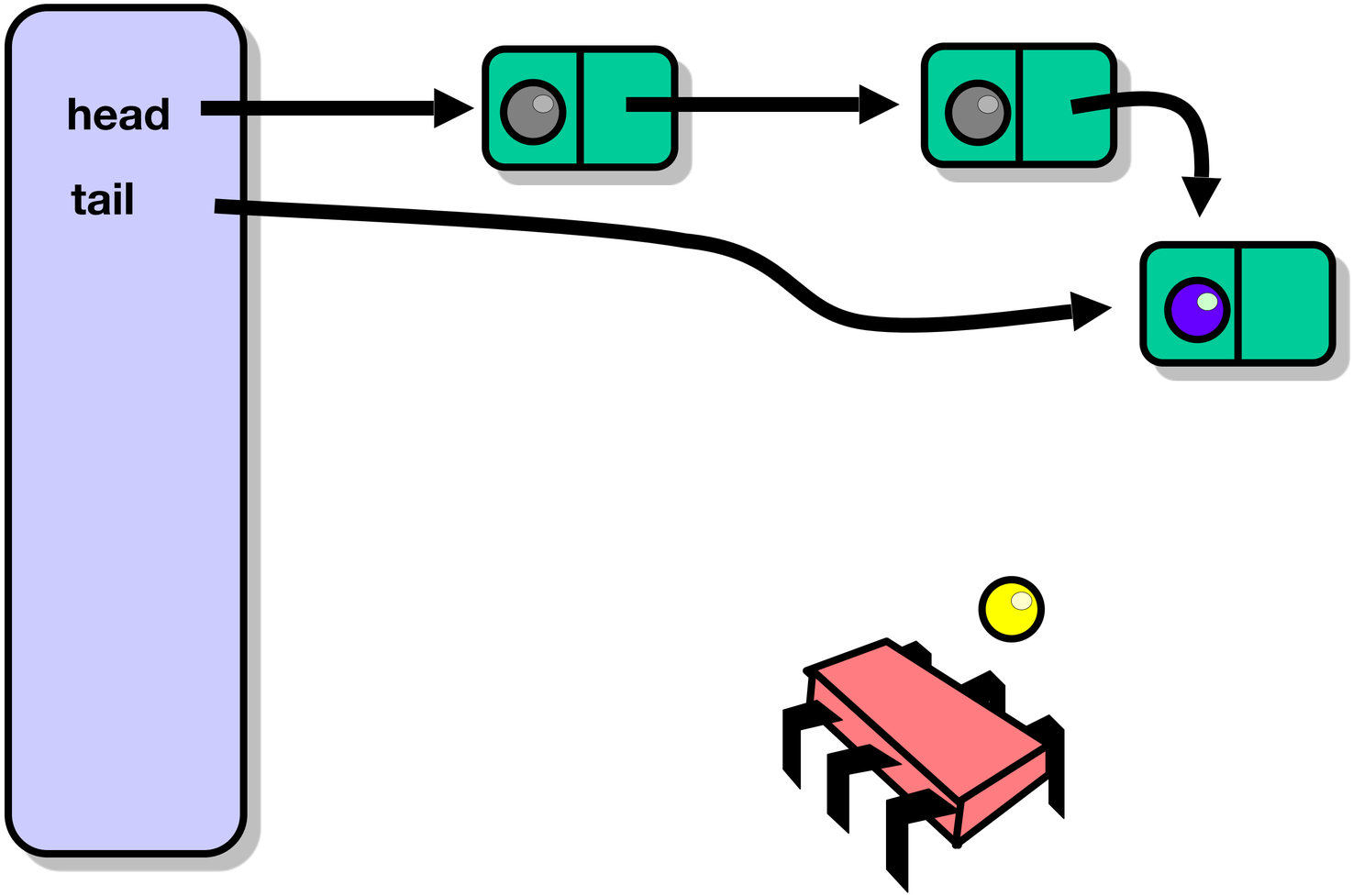
Walk through the code

lockfree_queue.ml

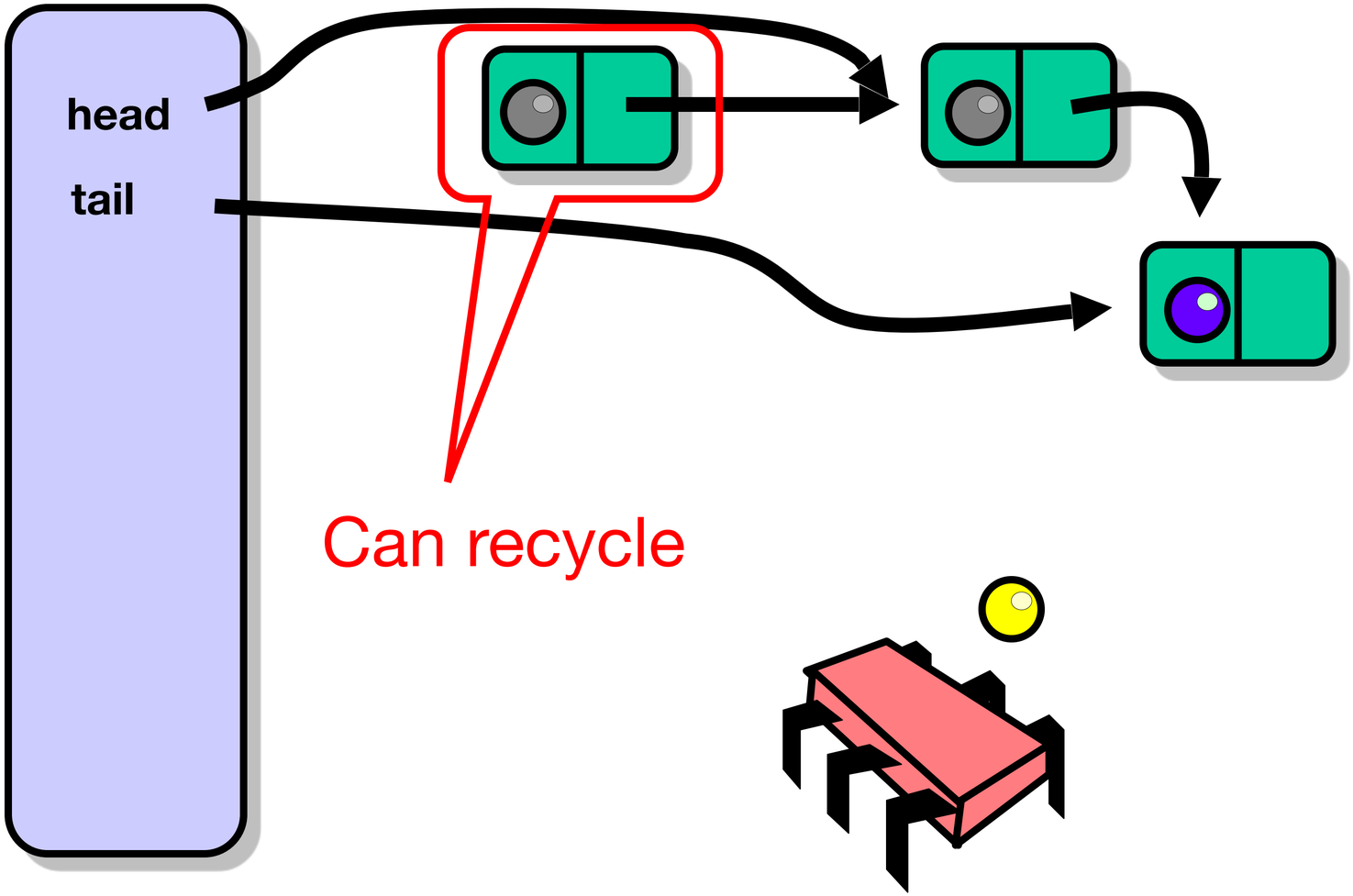
Memory Reuse?

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

Dequeuer



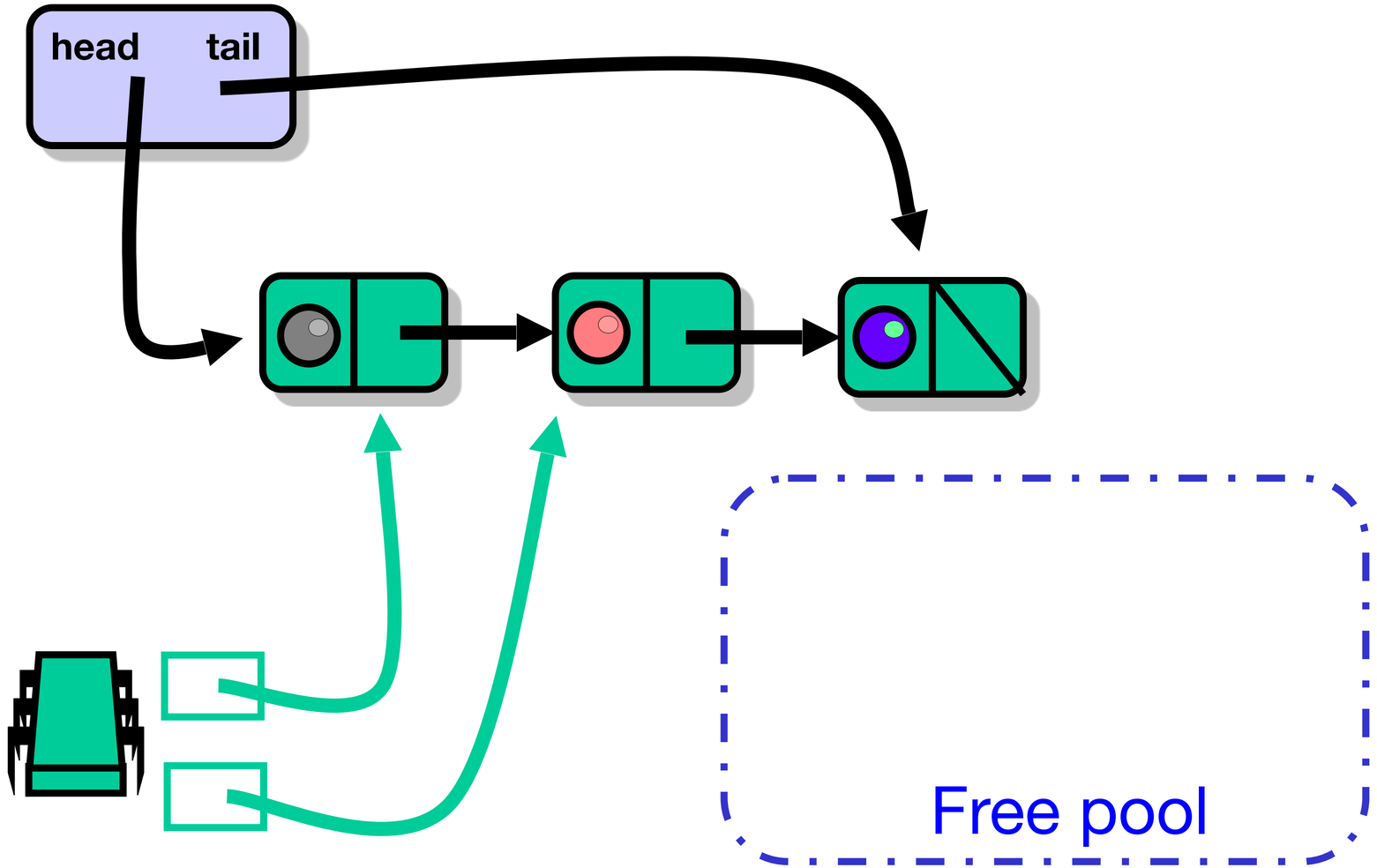
Dequeuer



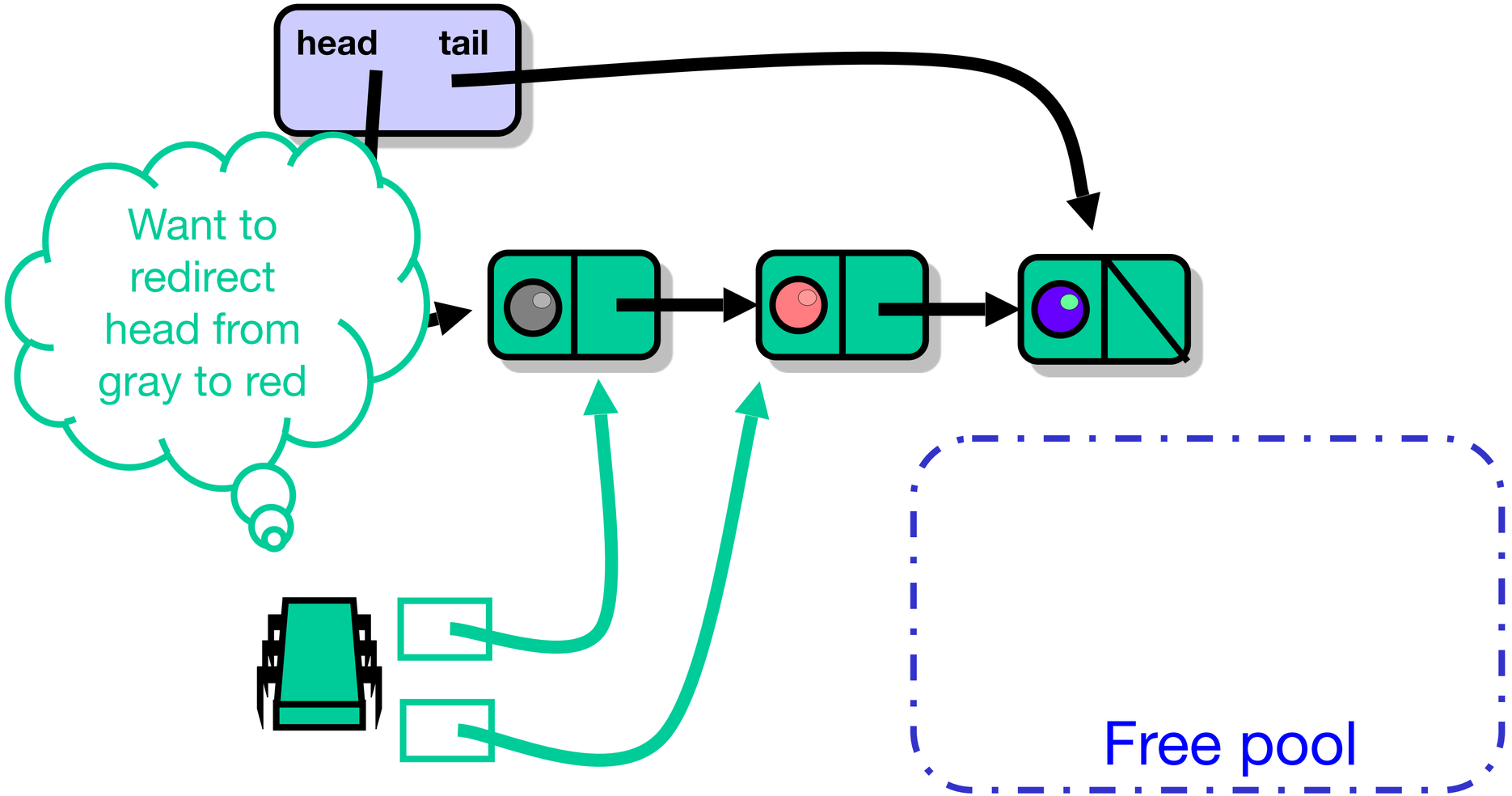
Simple Solution

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

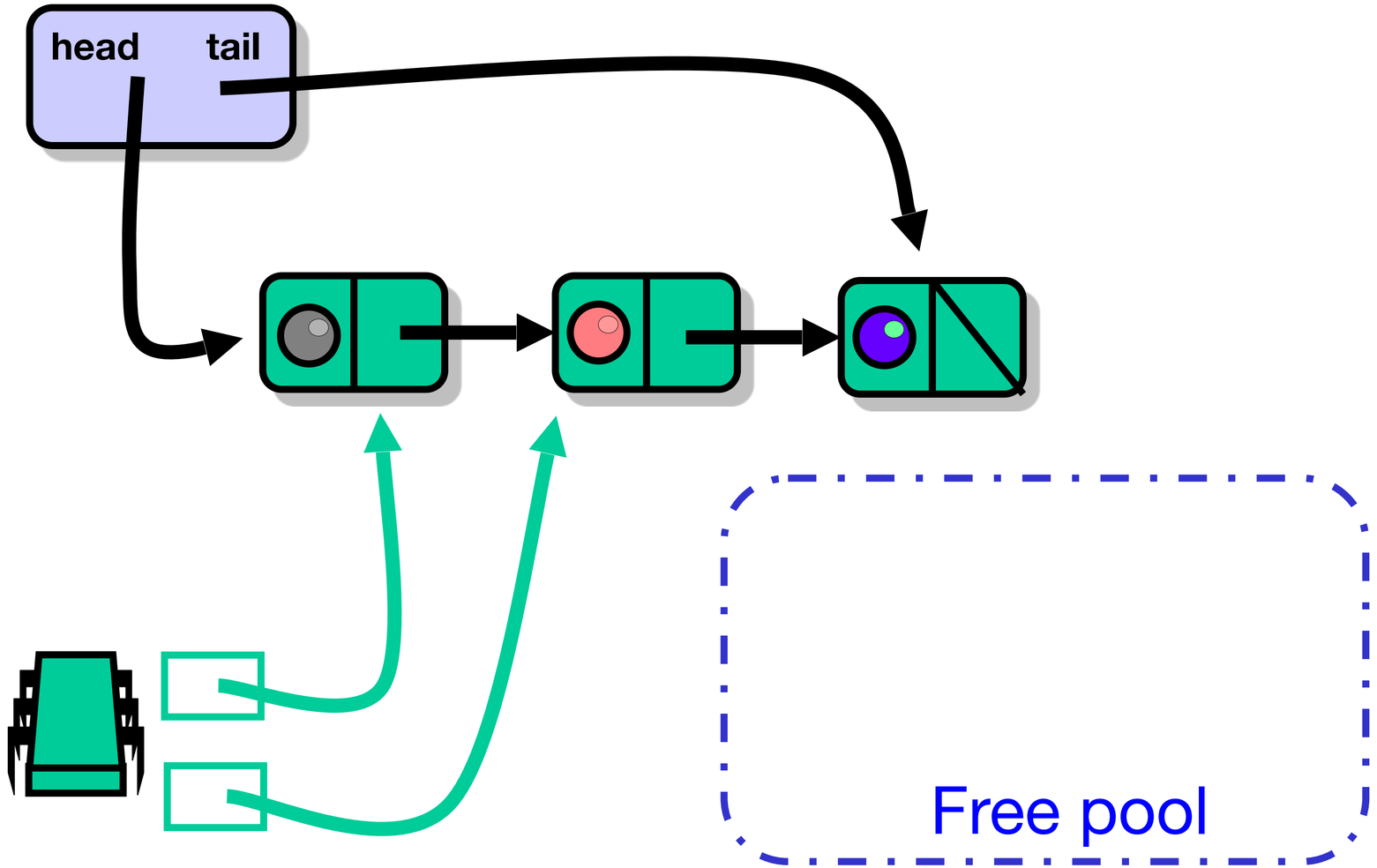
Why Recycling is Hard



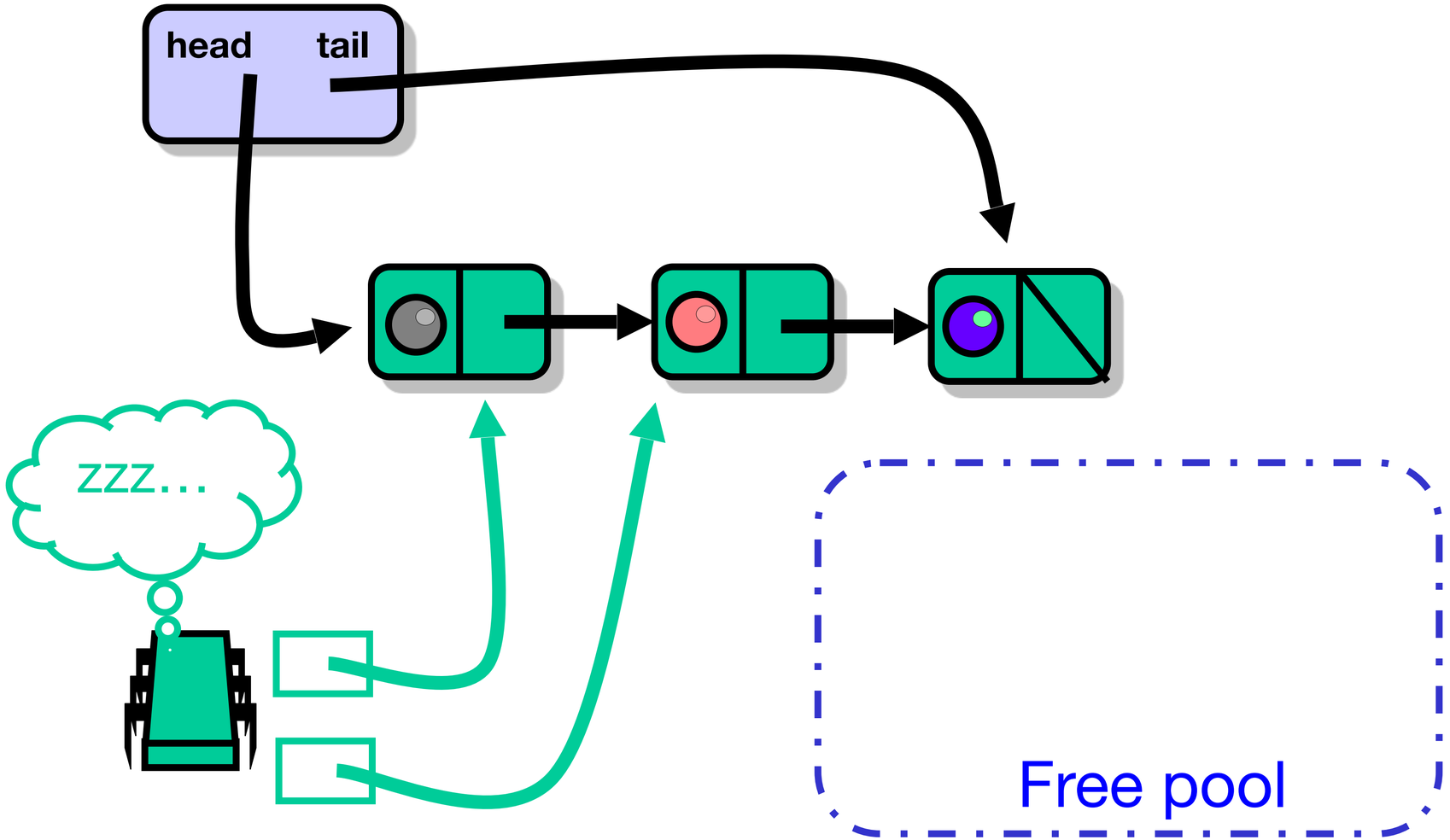
Why Recycling is Hard



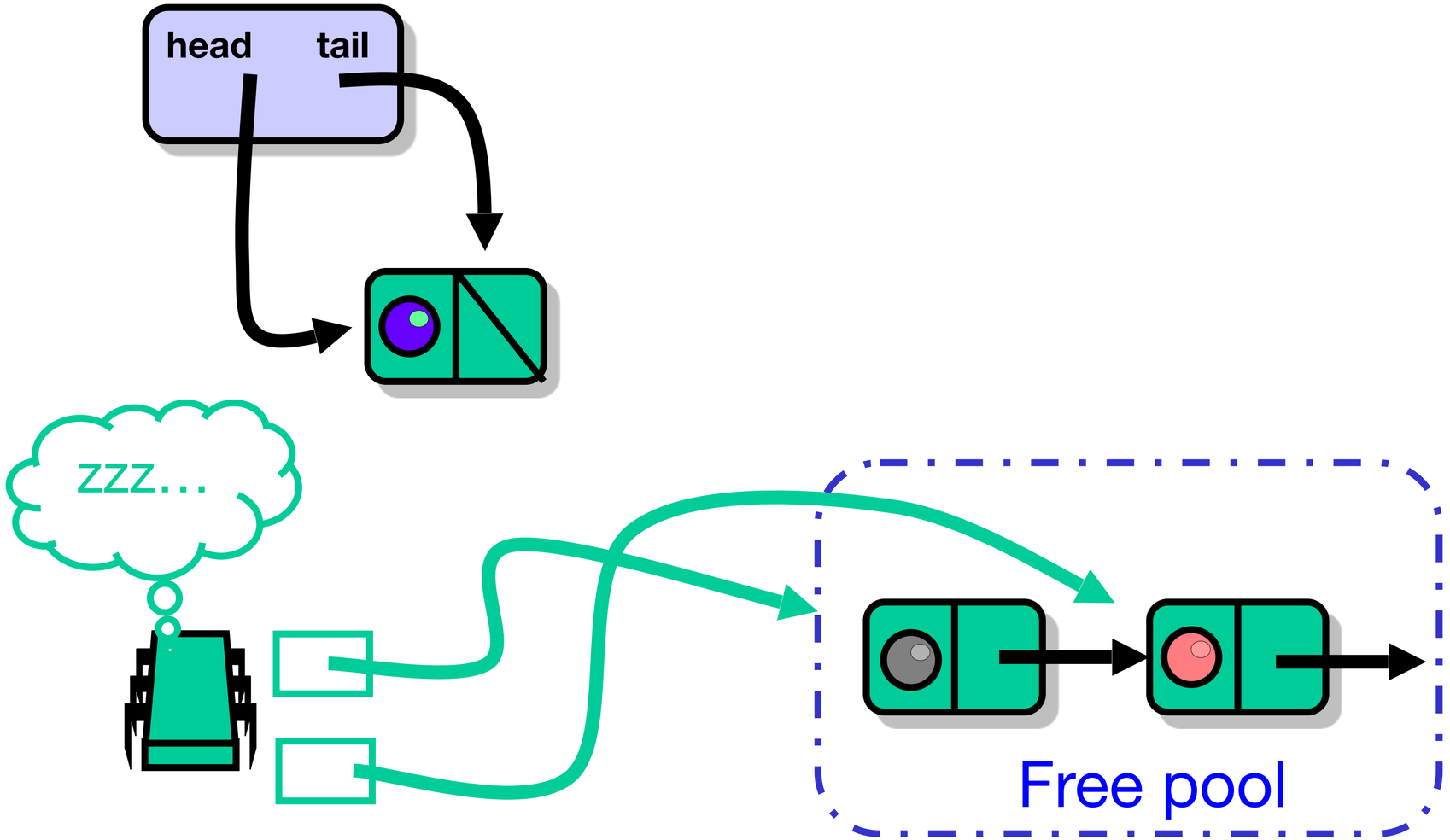
Why Recycling is Hard



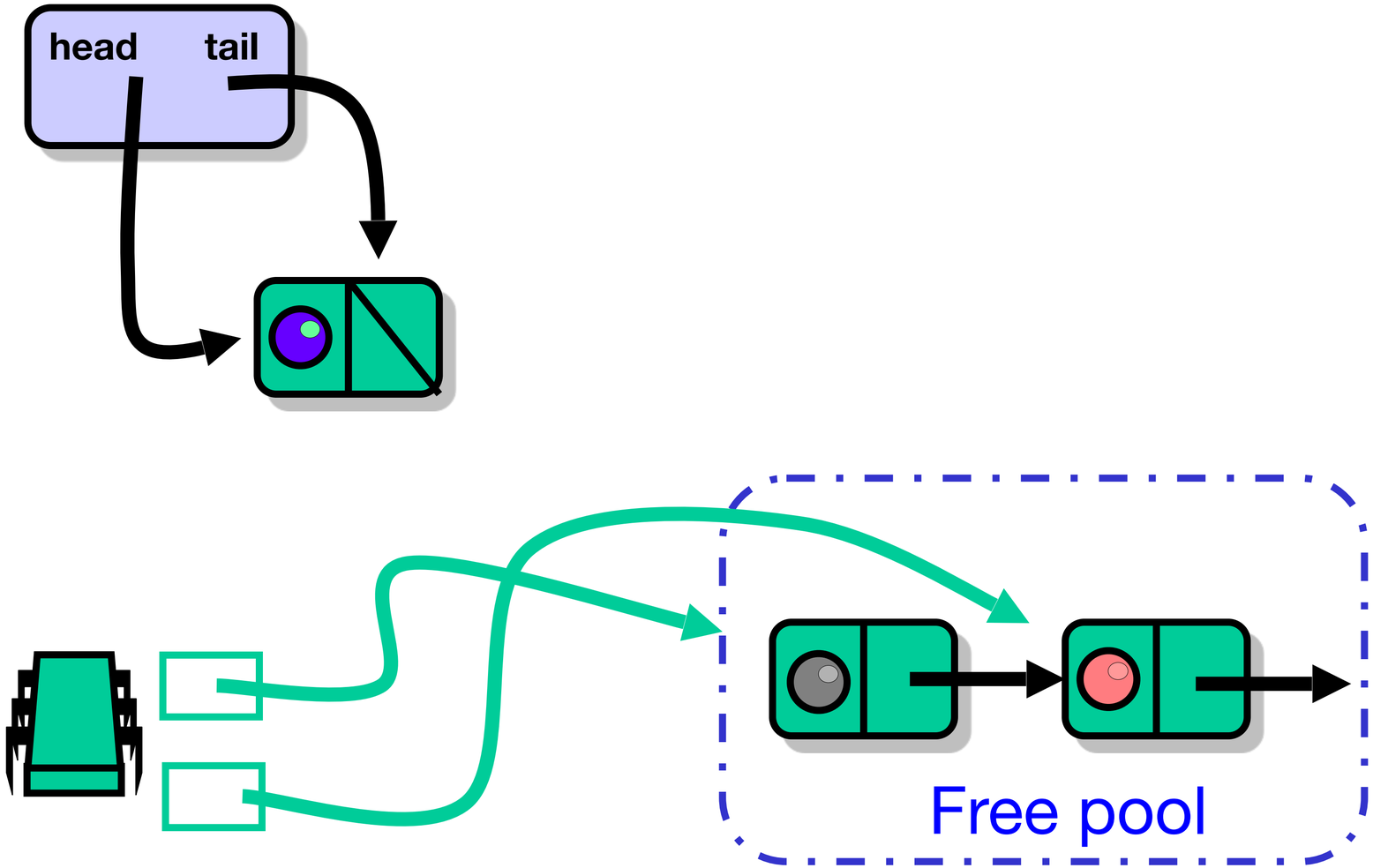
Why Recycling is Hard



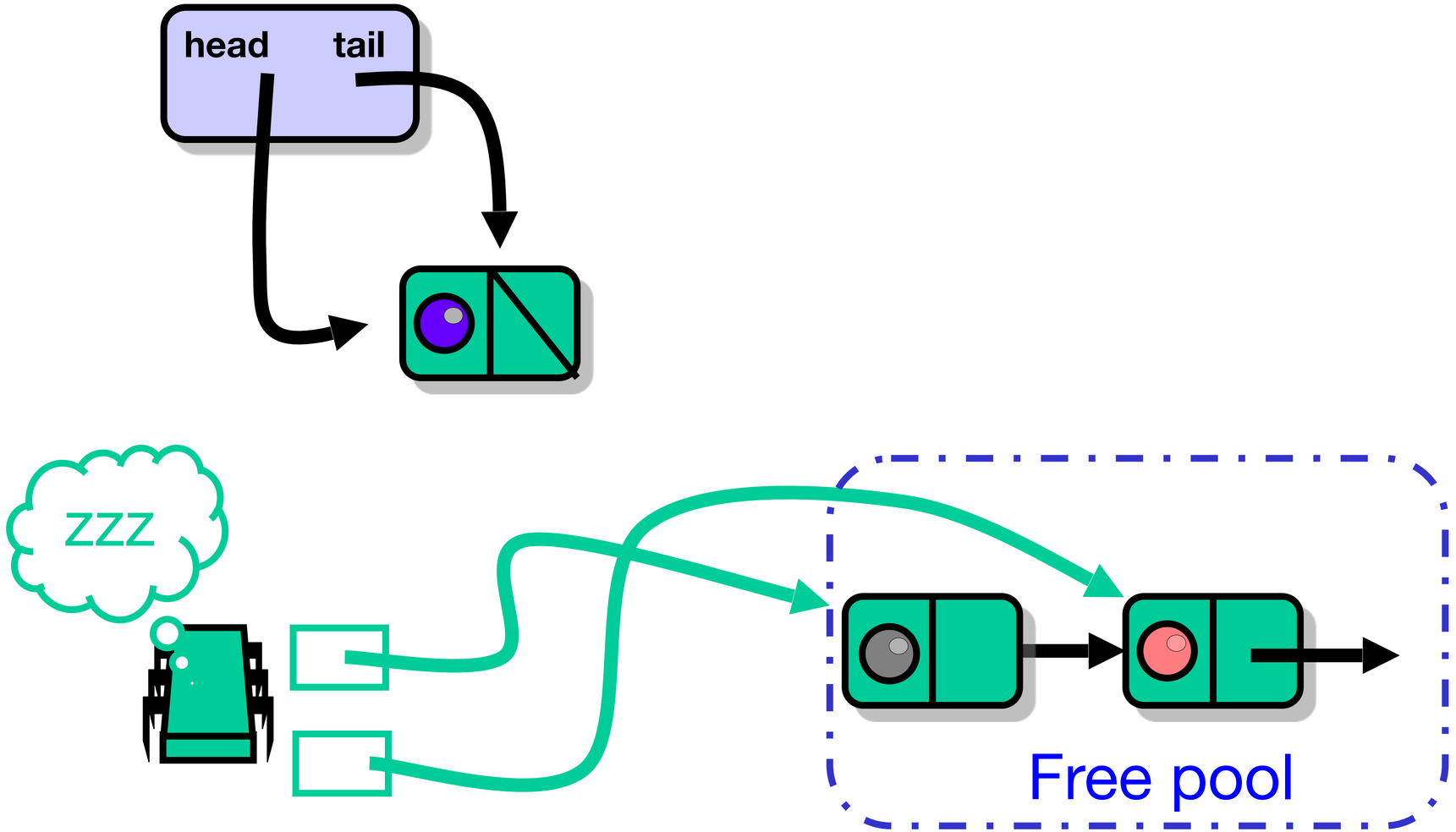
Why Recycling is Hard



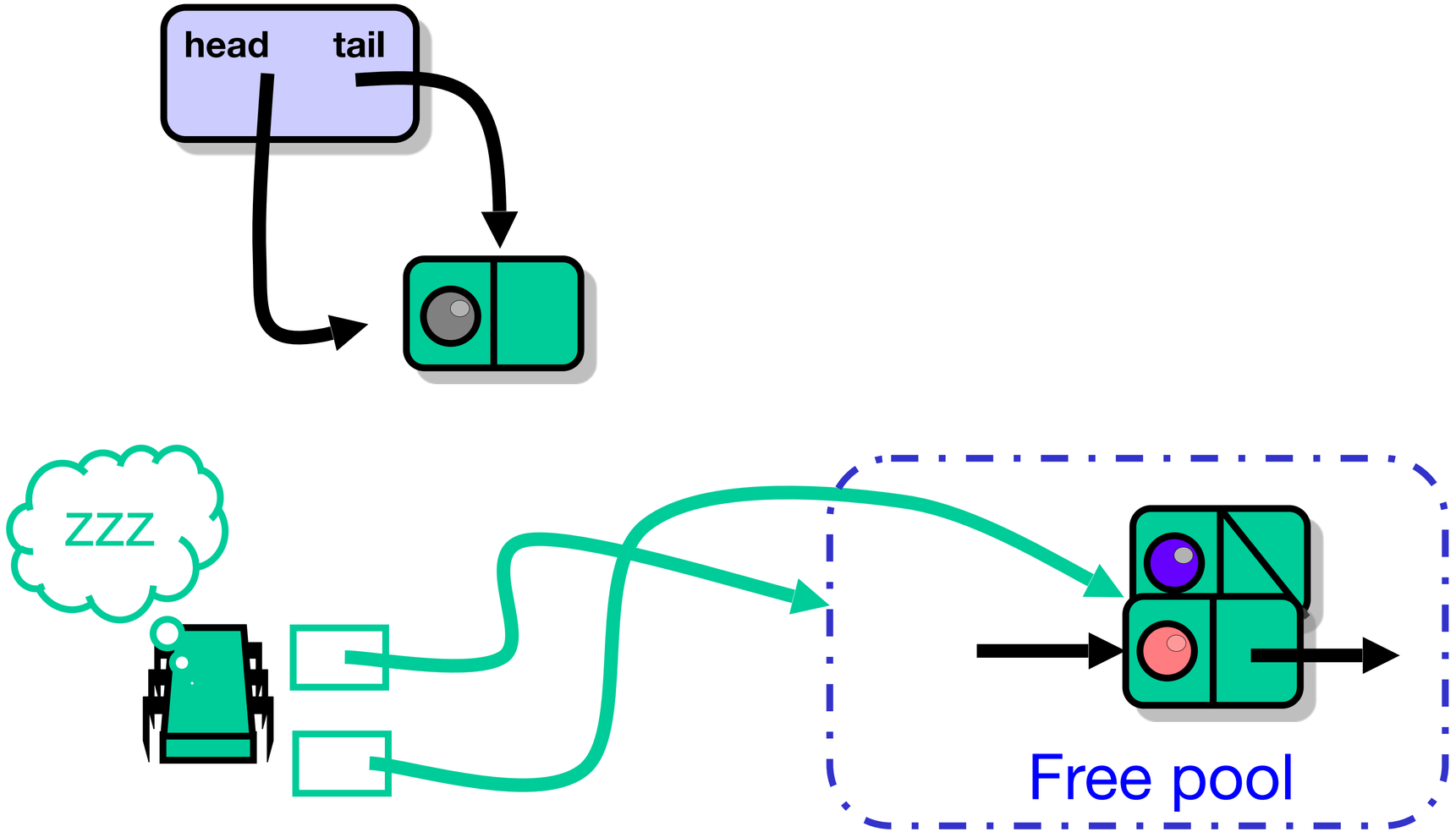
Why Recycling is Hard



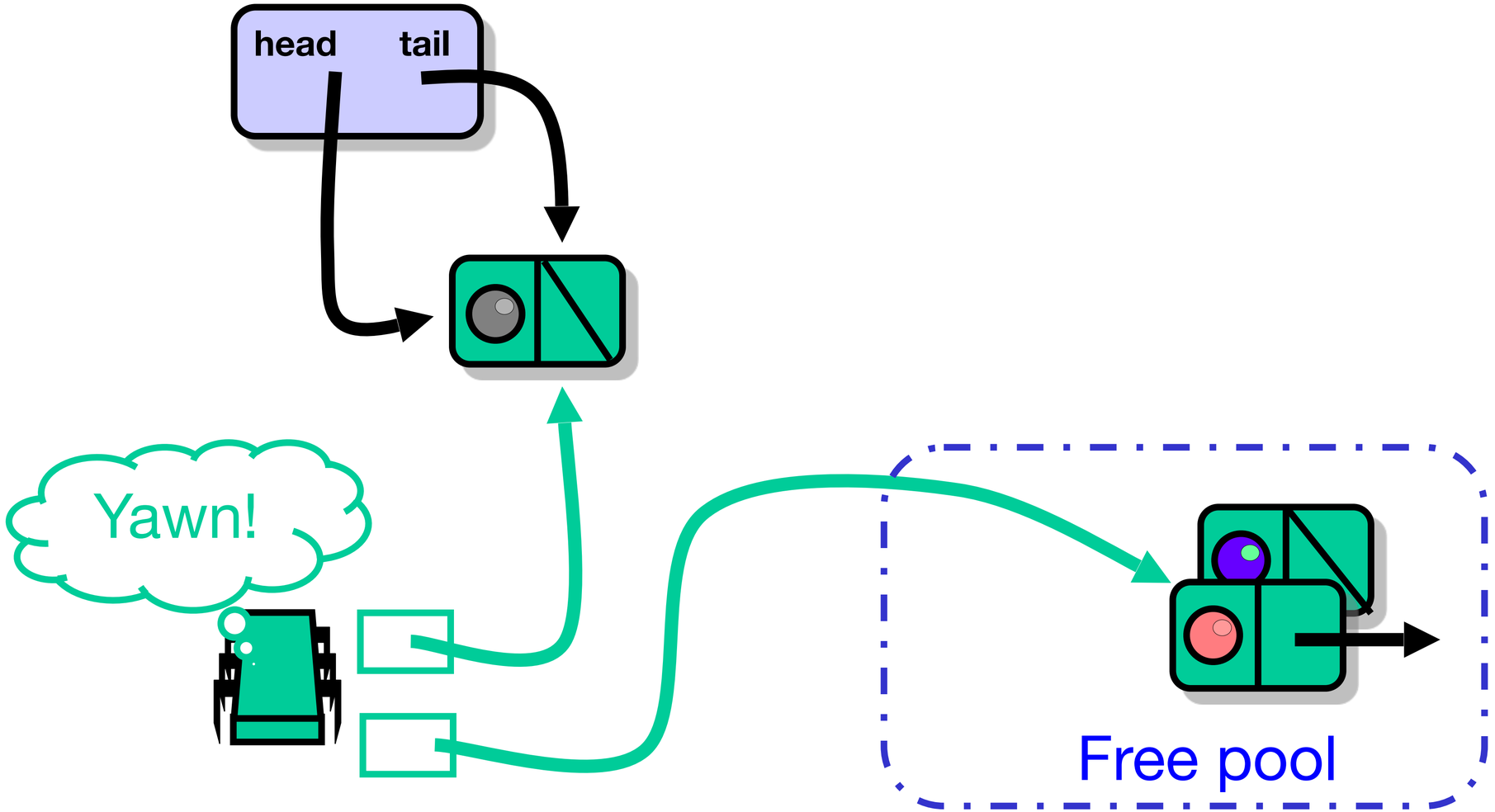
Both Nodes Reclaimed



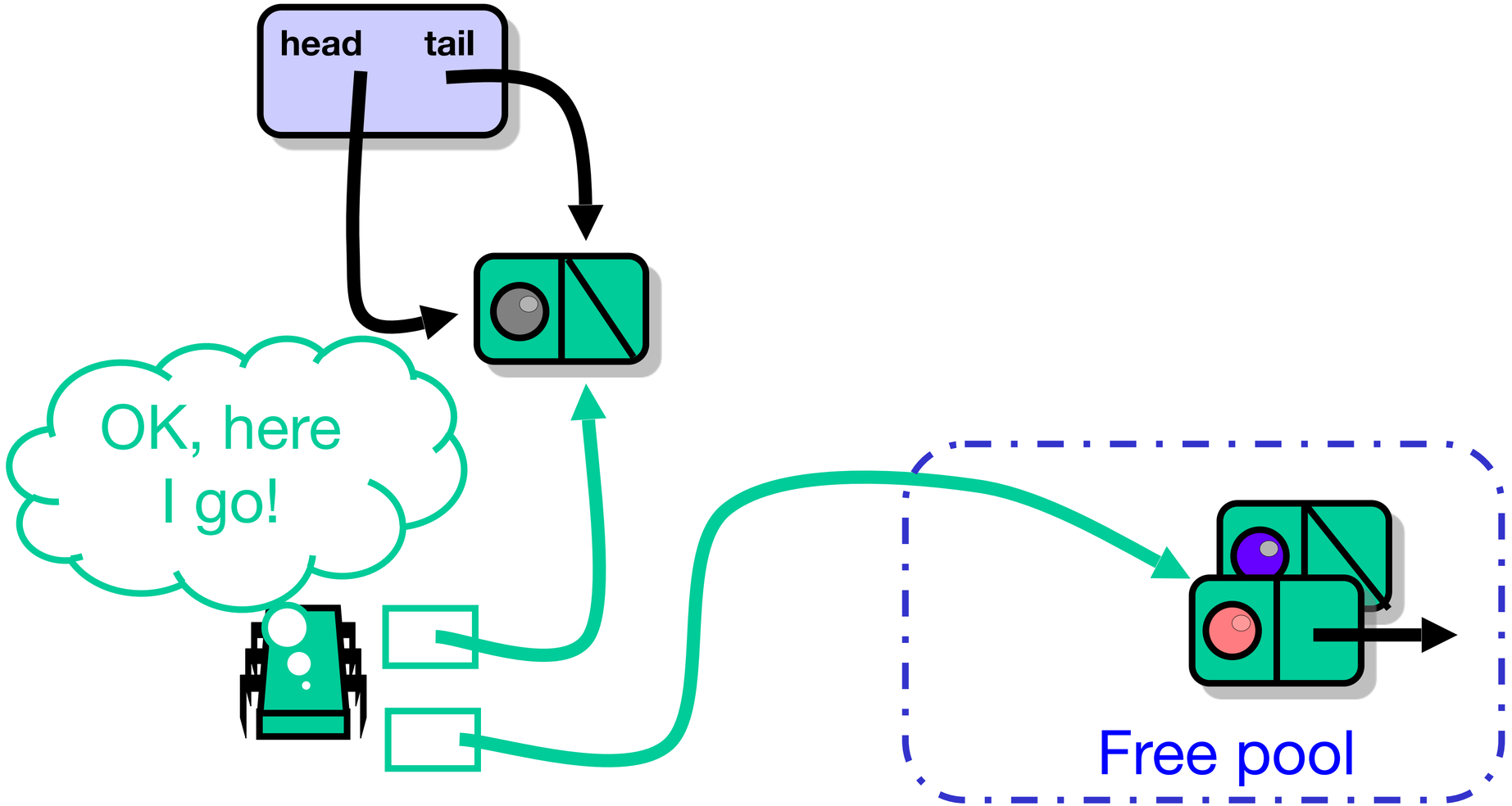
Both Nodes Reclaimed



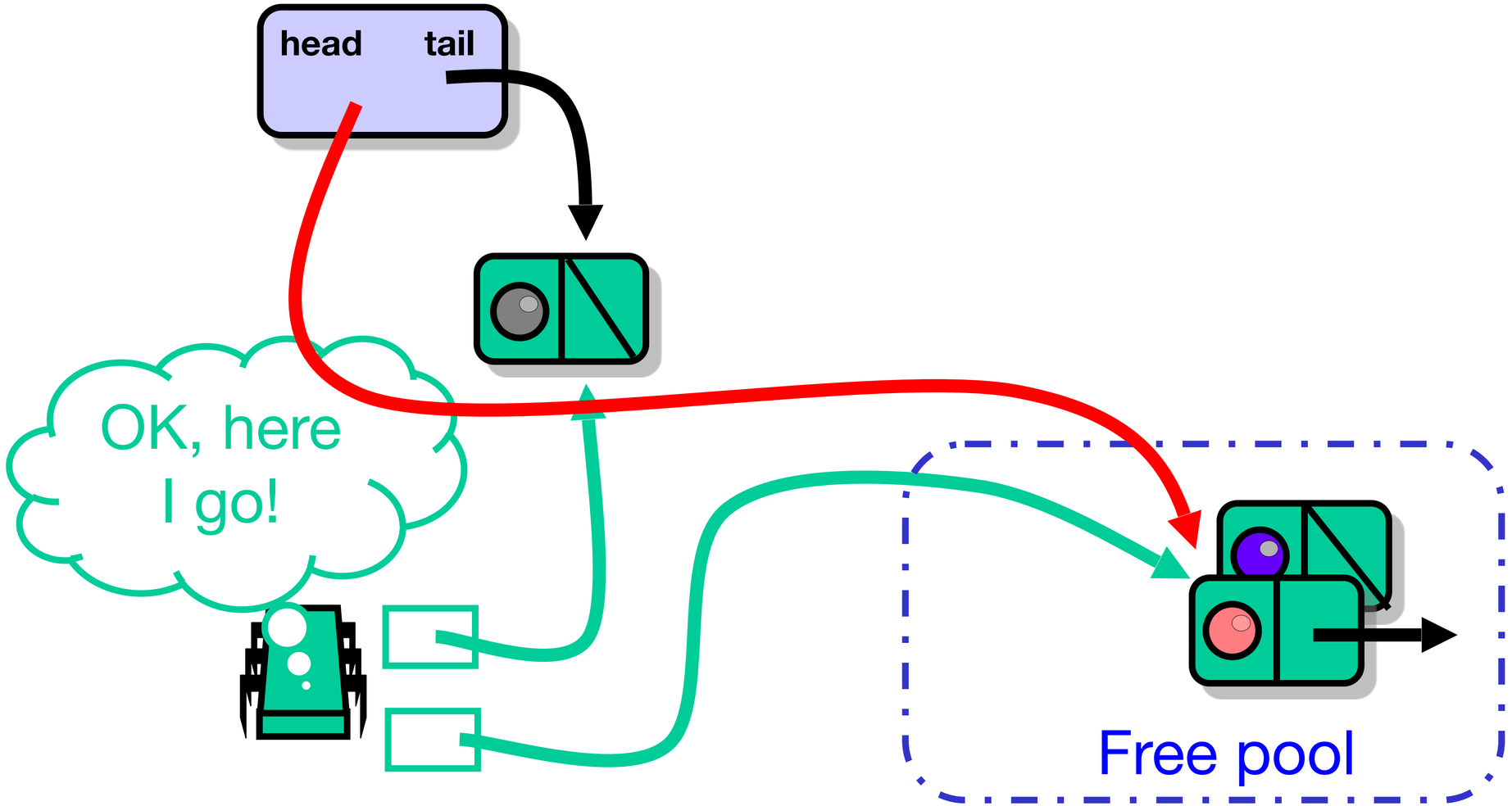
One Node Recycled



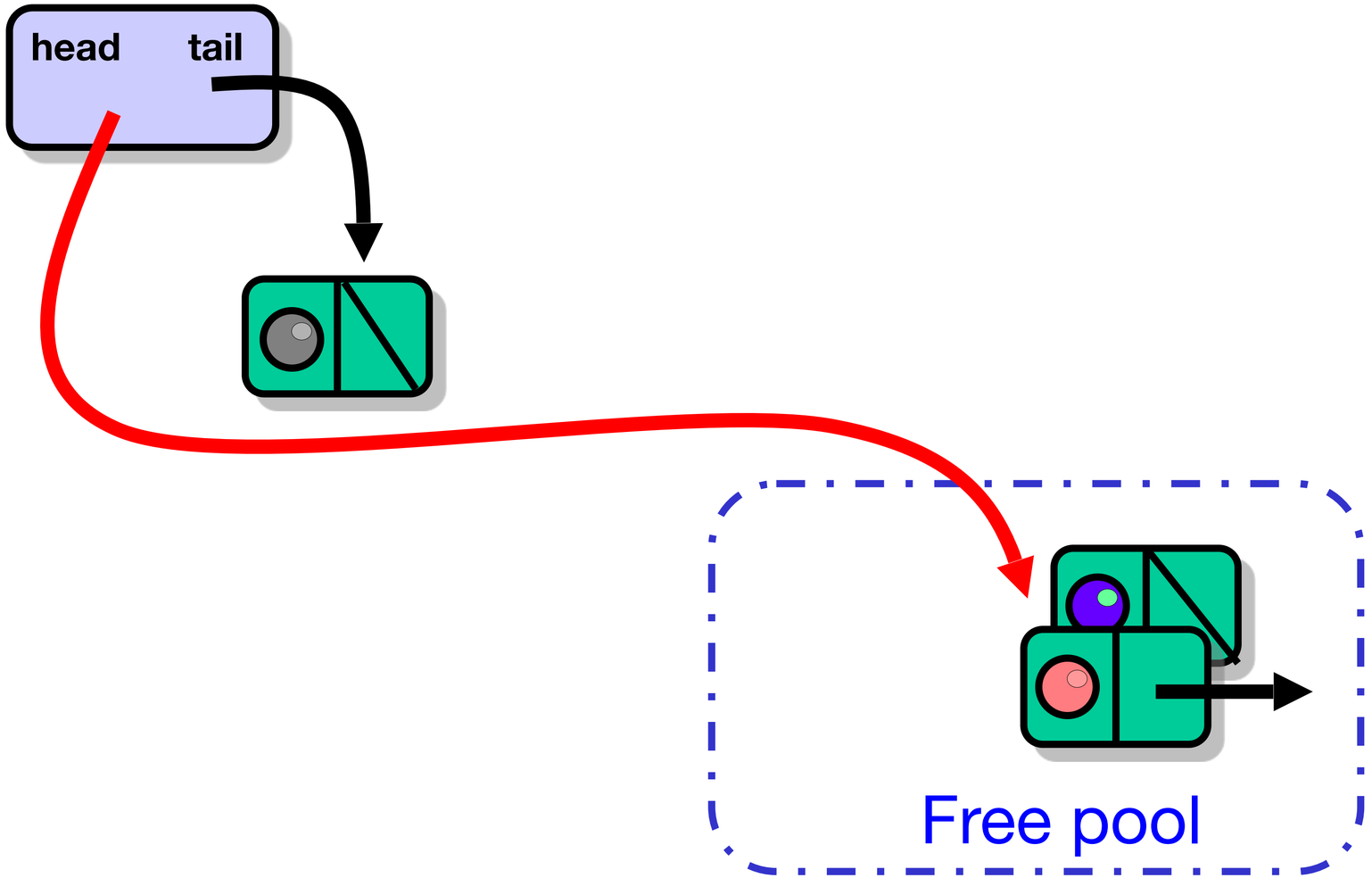
Why recycling is Hard



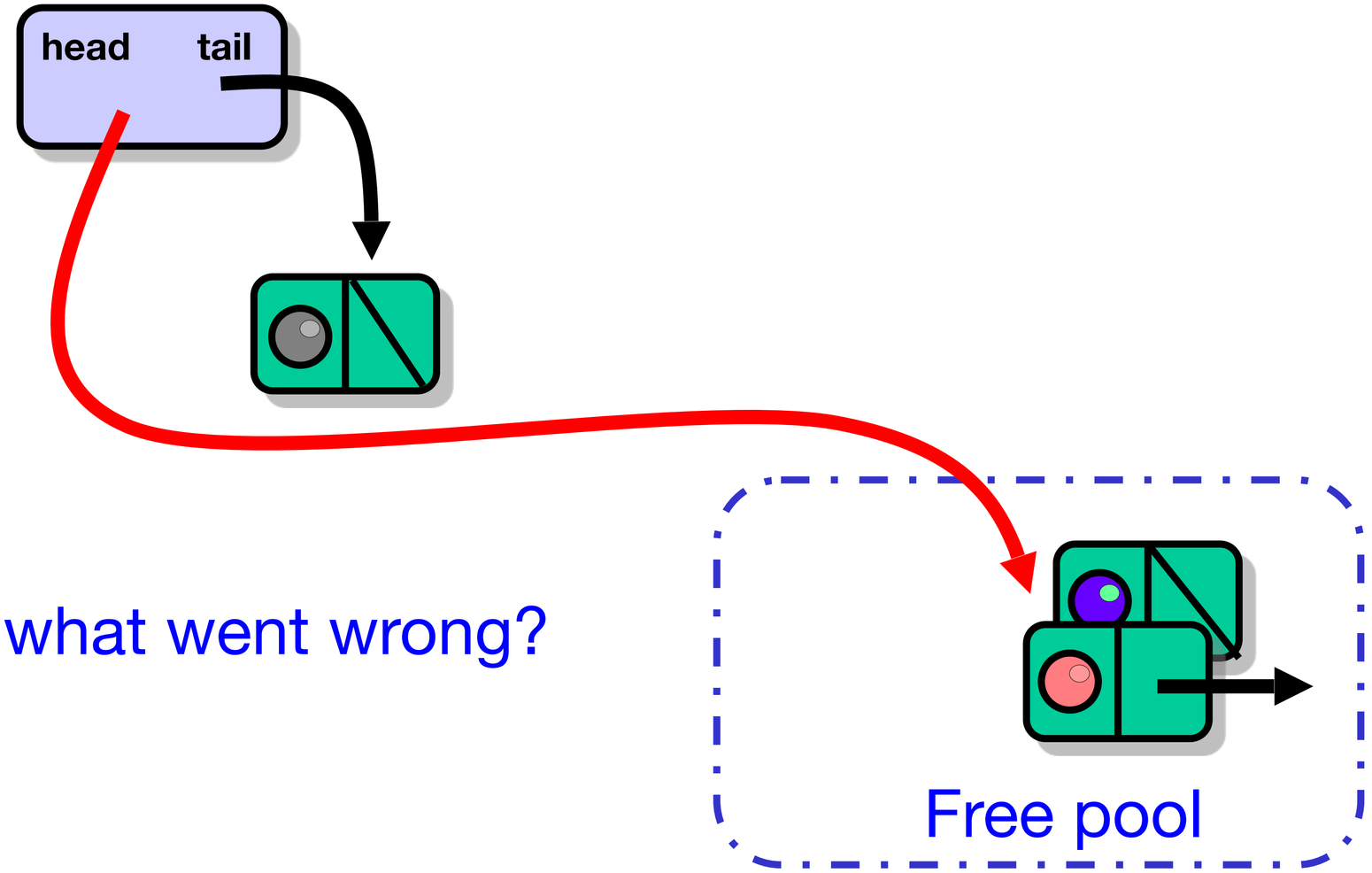
Why recycling is Hard



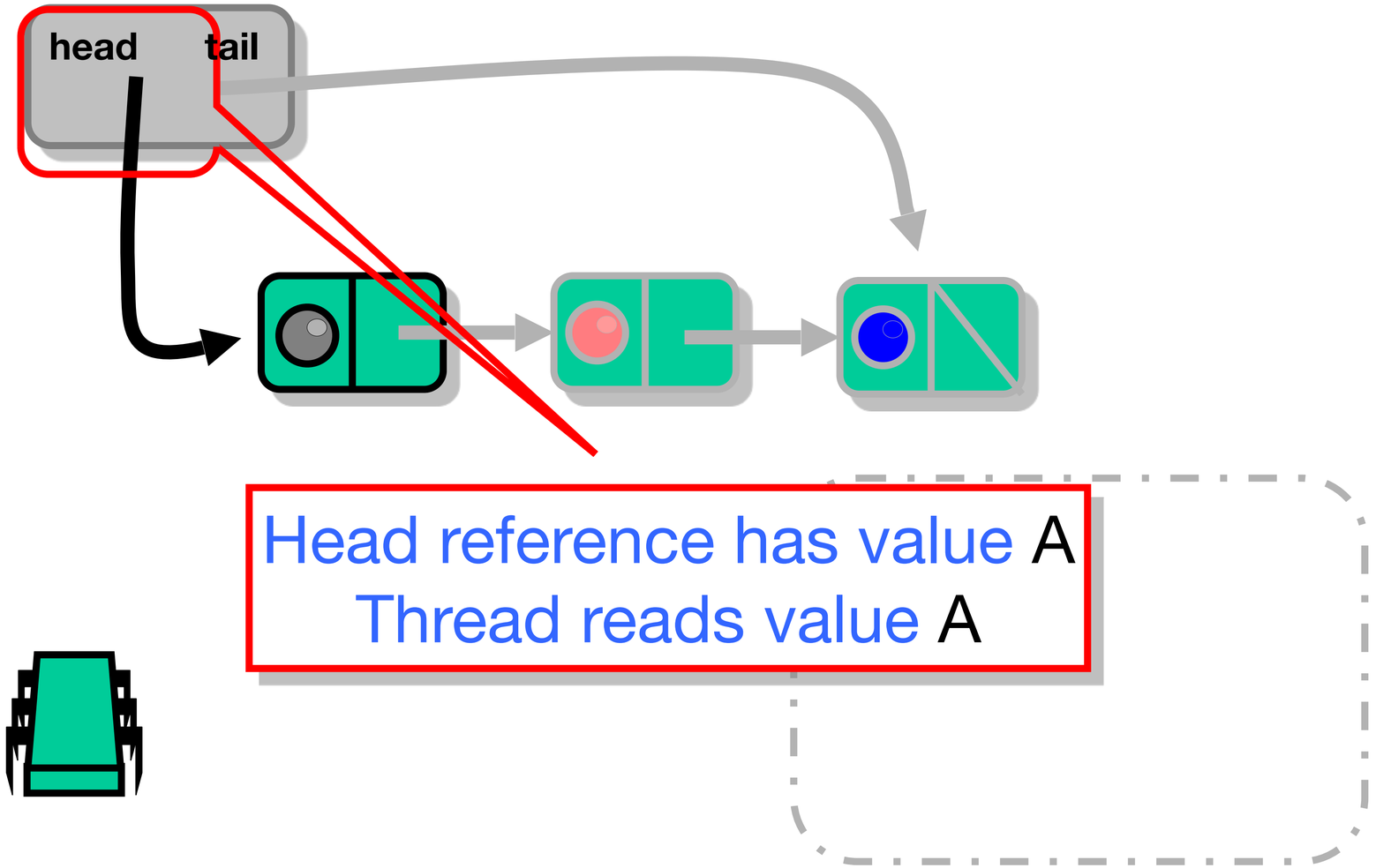
Recycle FAIL



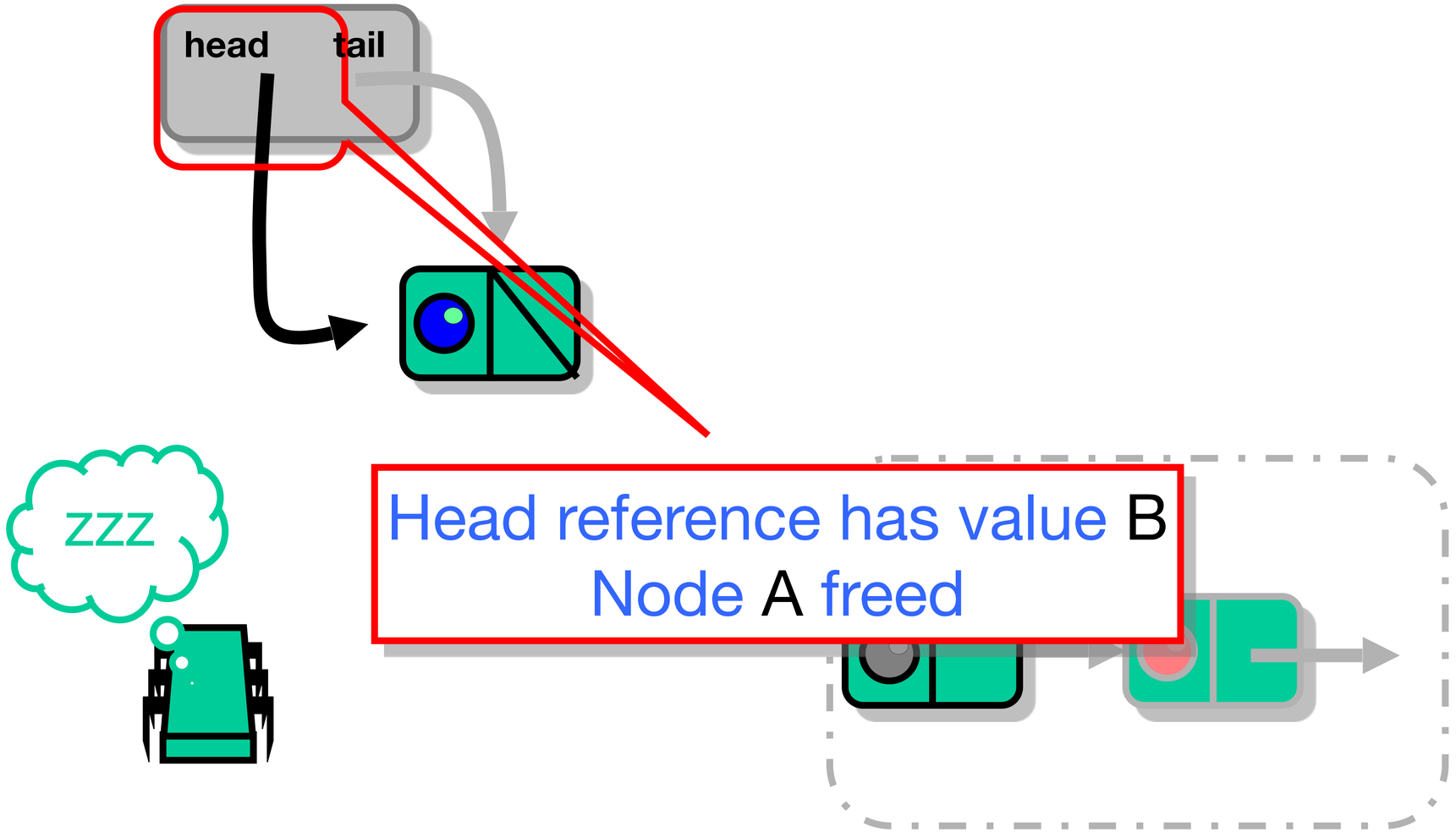
Recycle FAIL



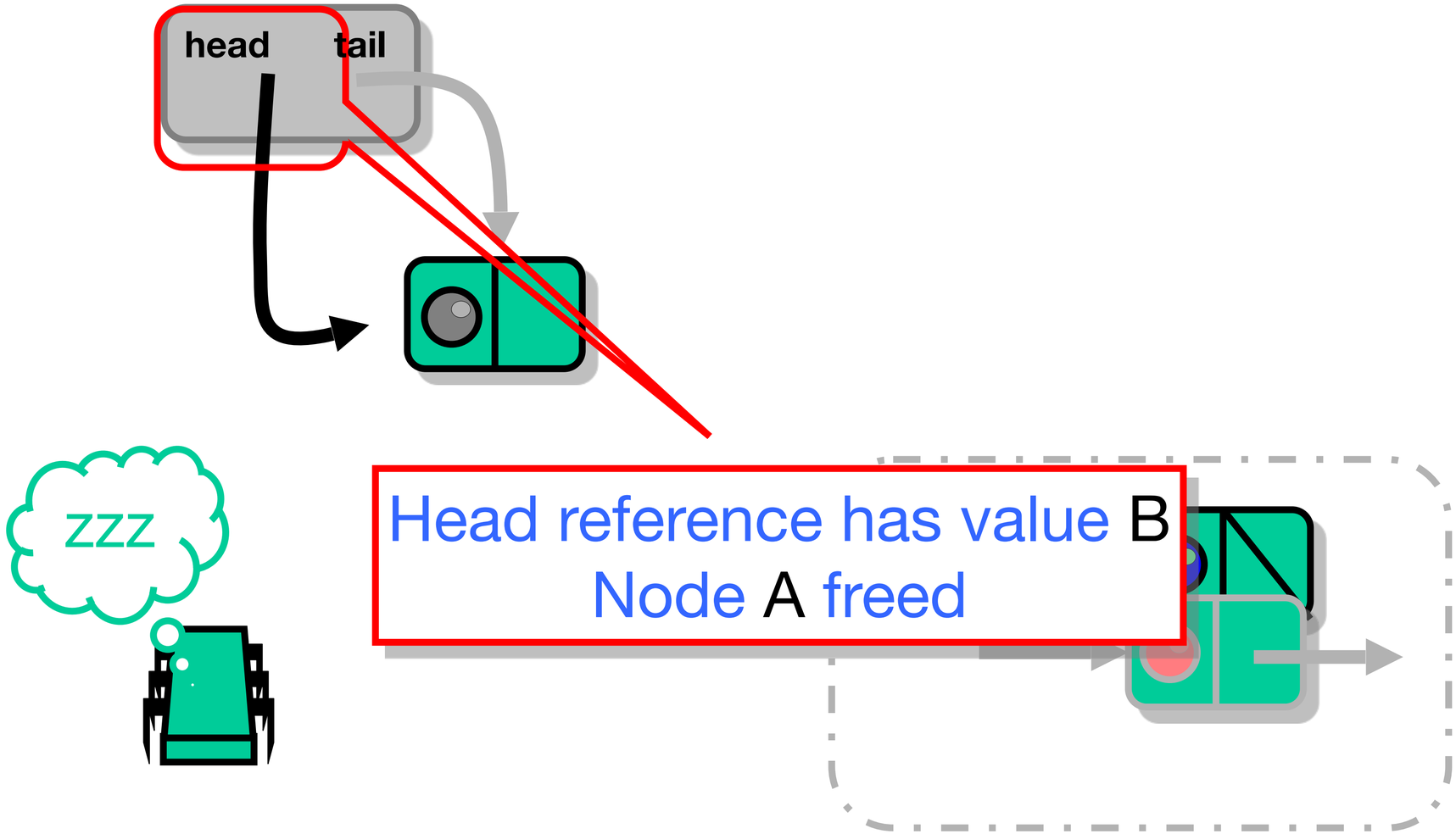
The dreaded ABA problem



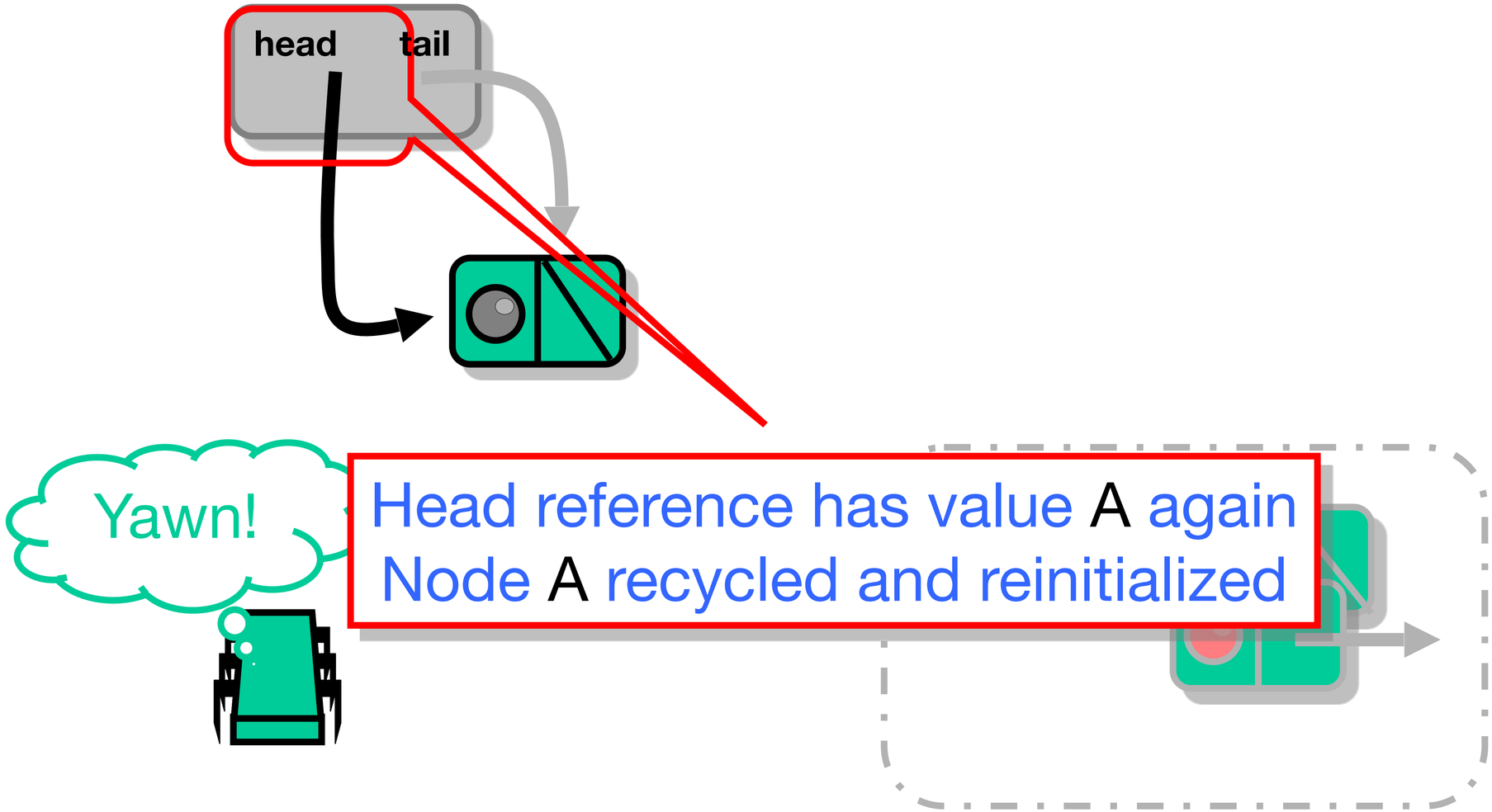
Dreaded ABA continued



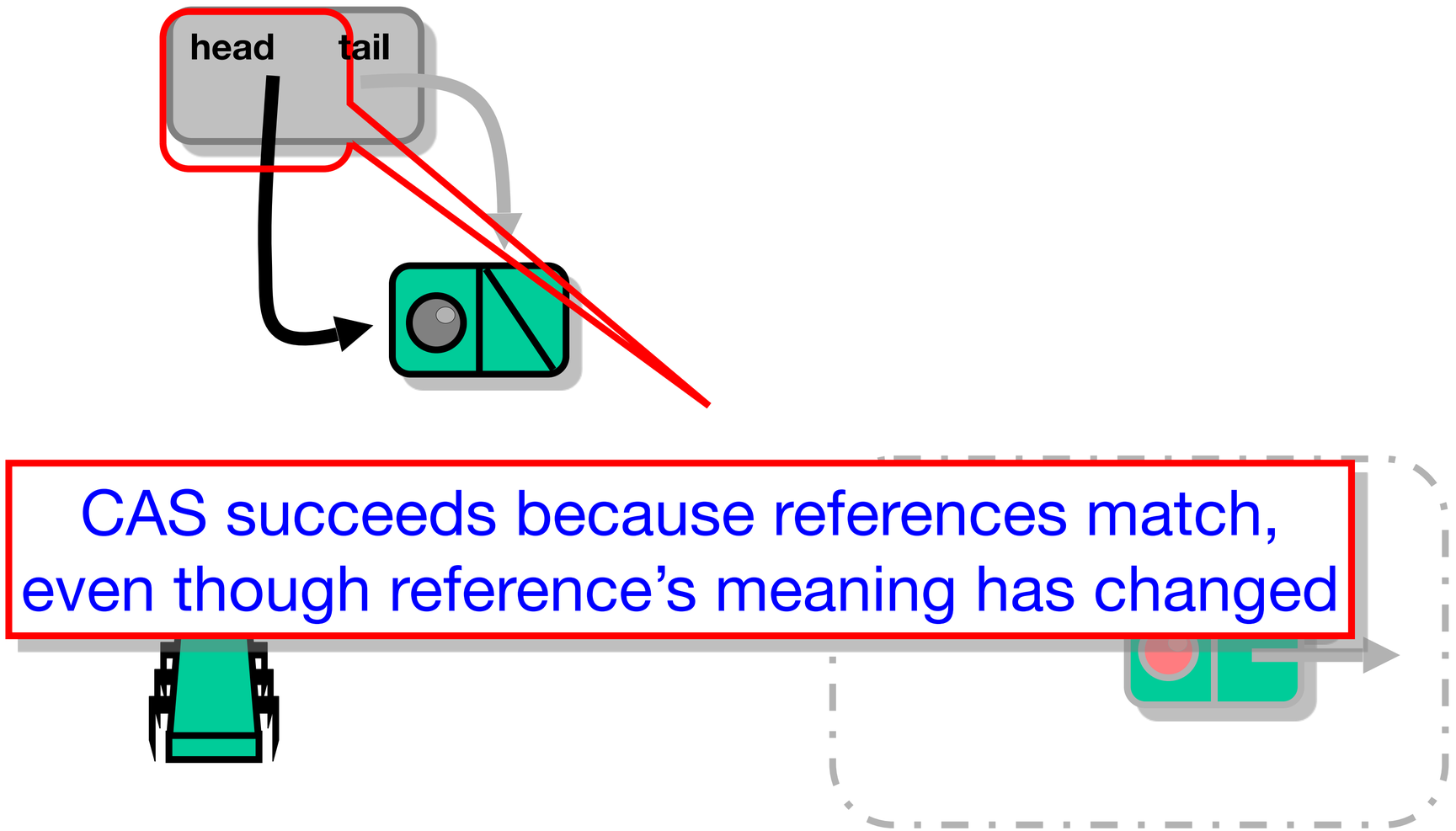
Dreaded ABA continued



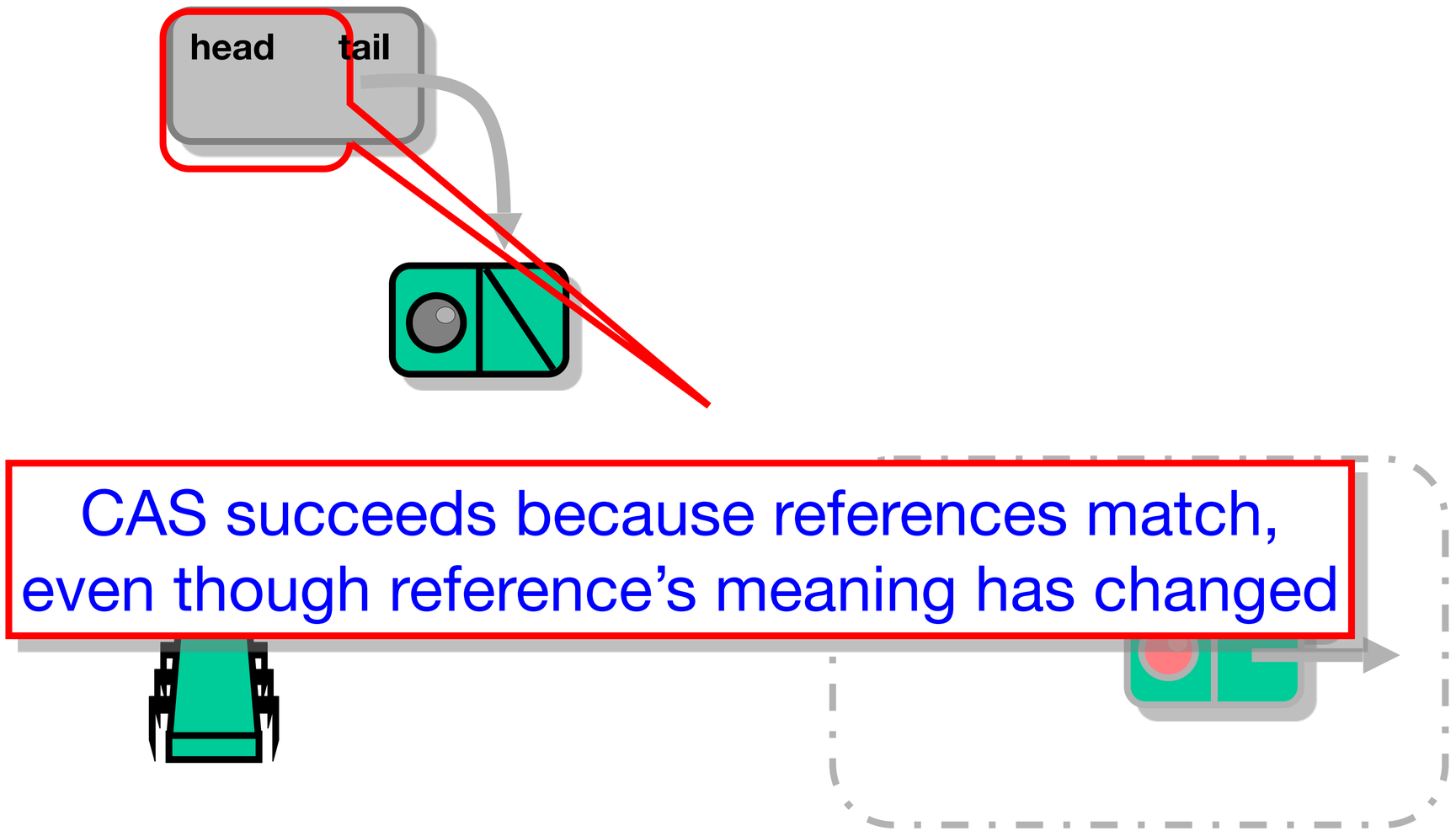
Dreaded ABA continued



Dreaded ABA continued



Dreaded ABA continued



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

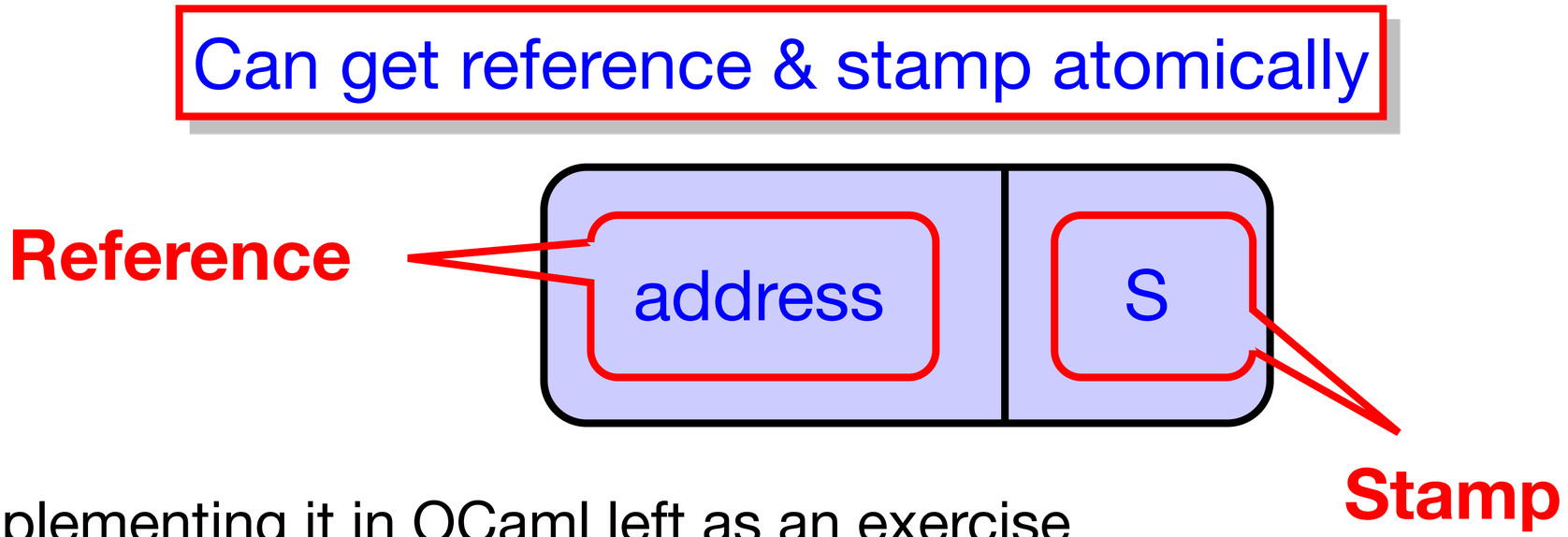
AtomicStampedReference

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

  - Implementing it in OCaml left as an exercise

AtomicStampedReference

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



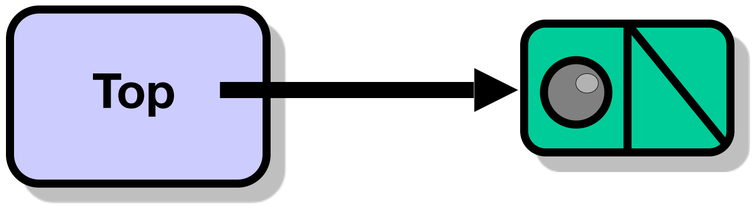
  - Implementing it in OCaml left as an exercise

Concurrent Stacks

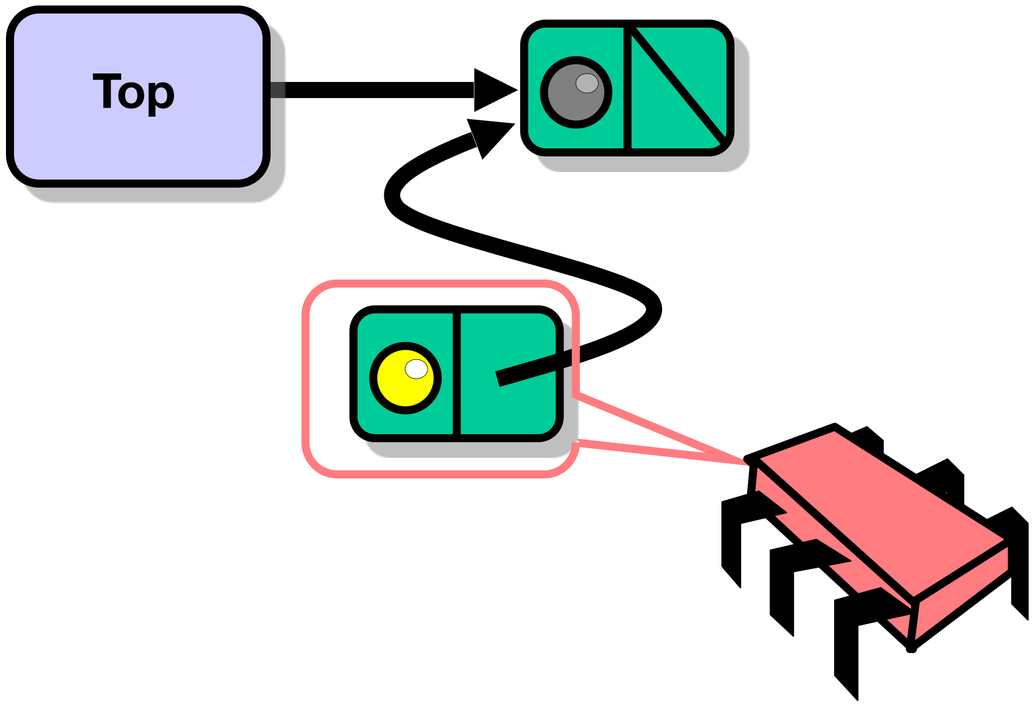
Concurrent Stack

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

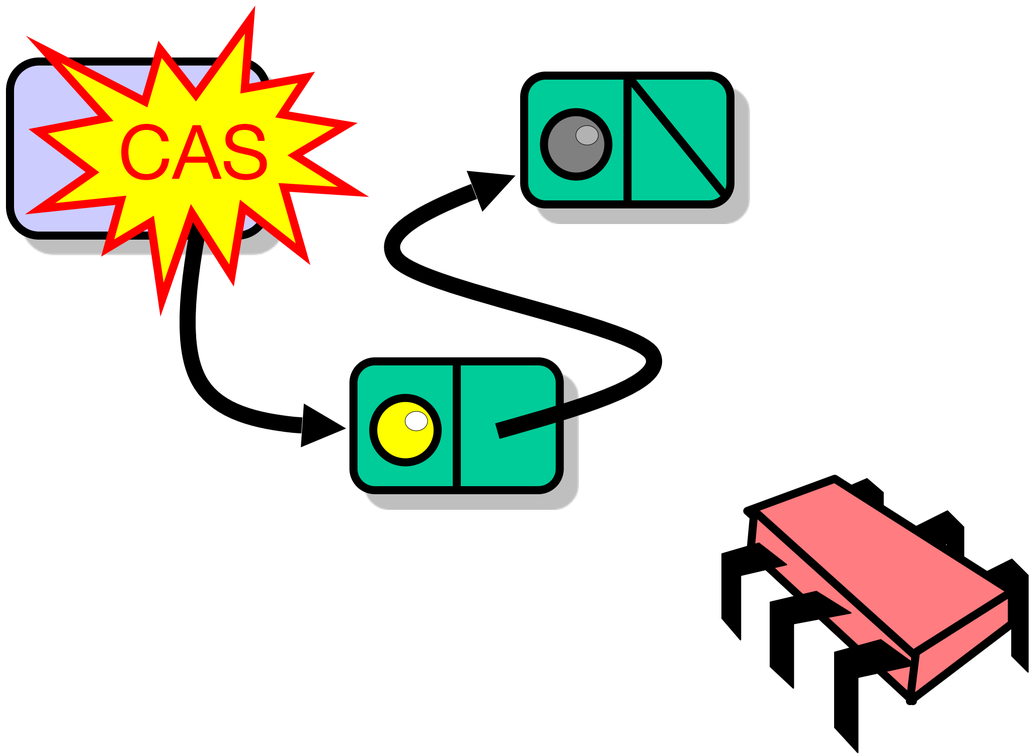
Empty Stack



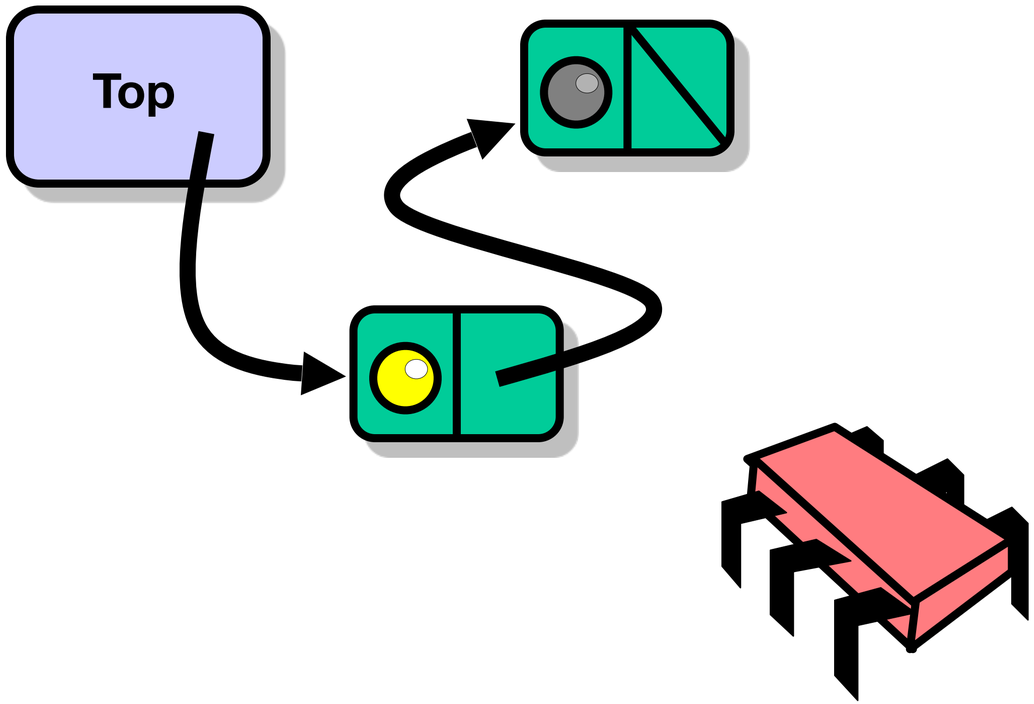
Push



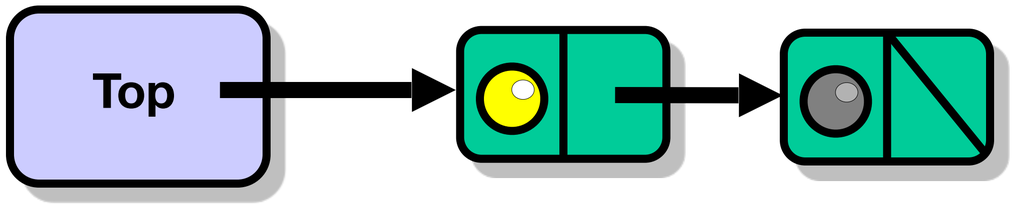
Push

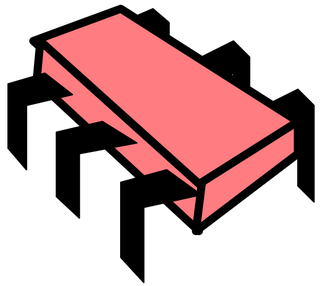


Push

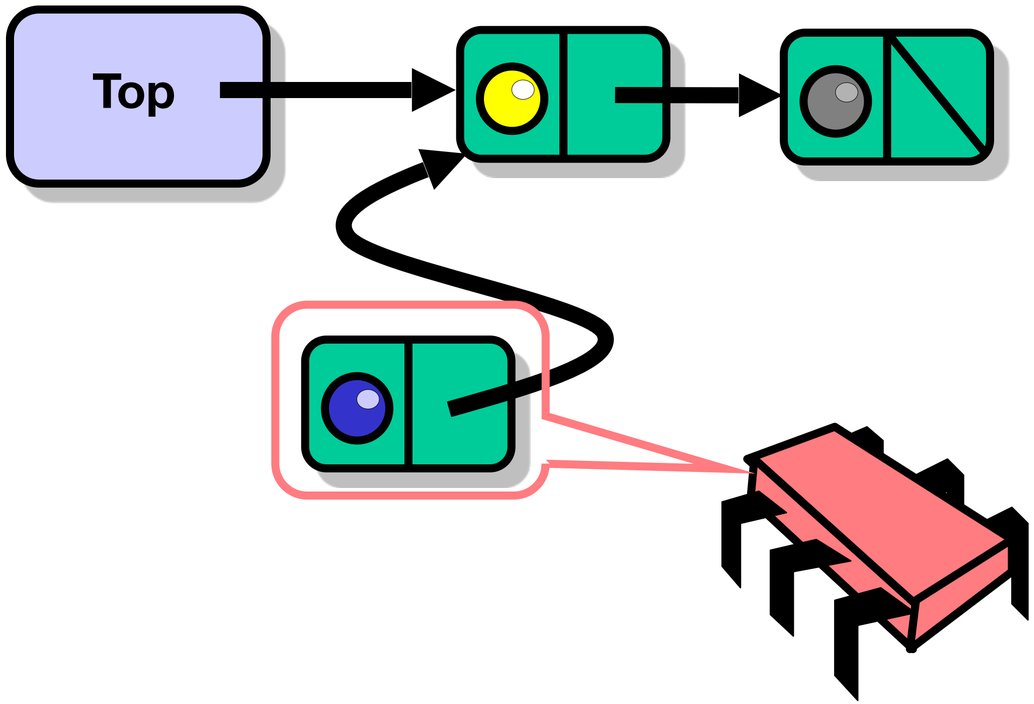


Push

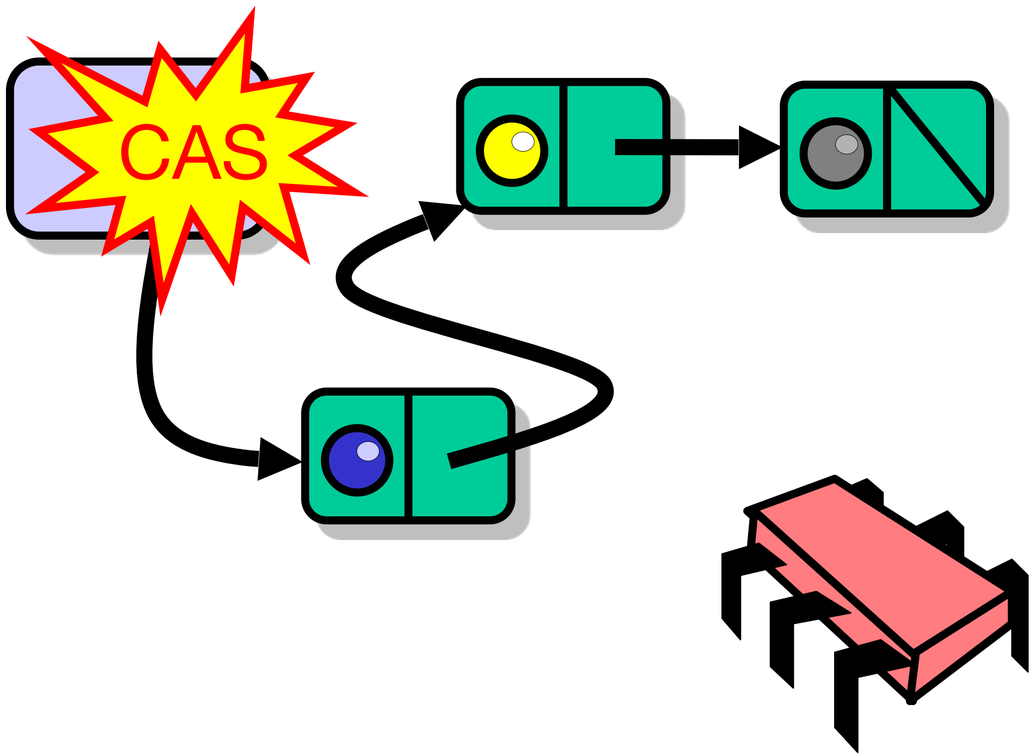




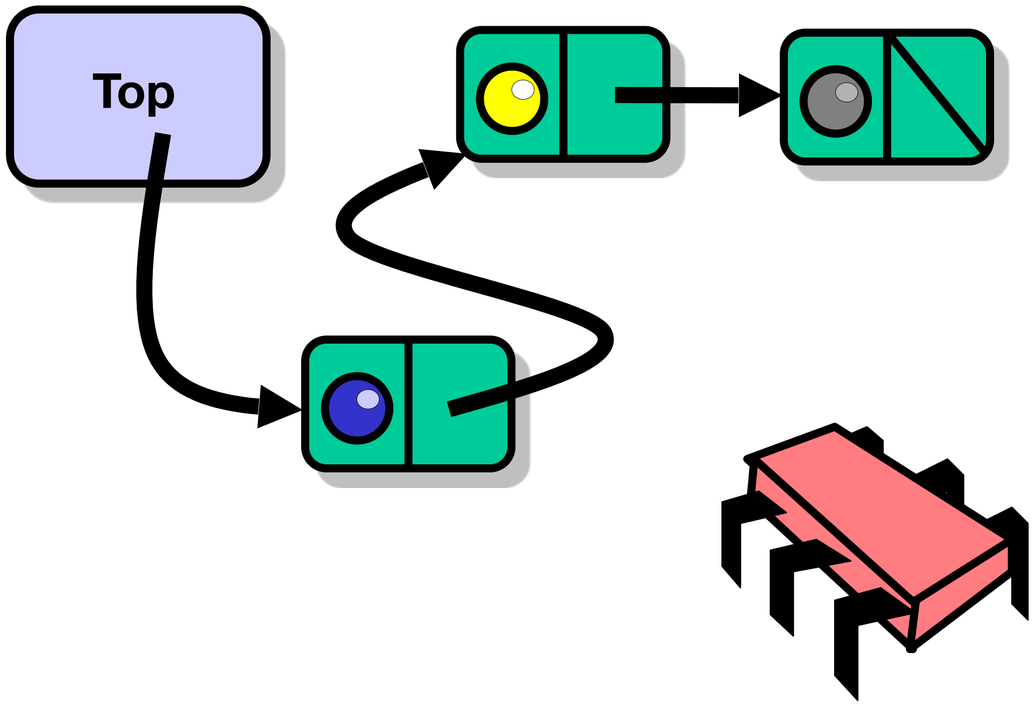
Push



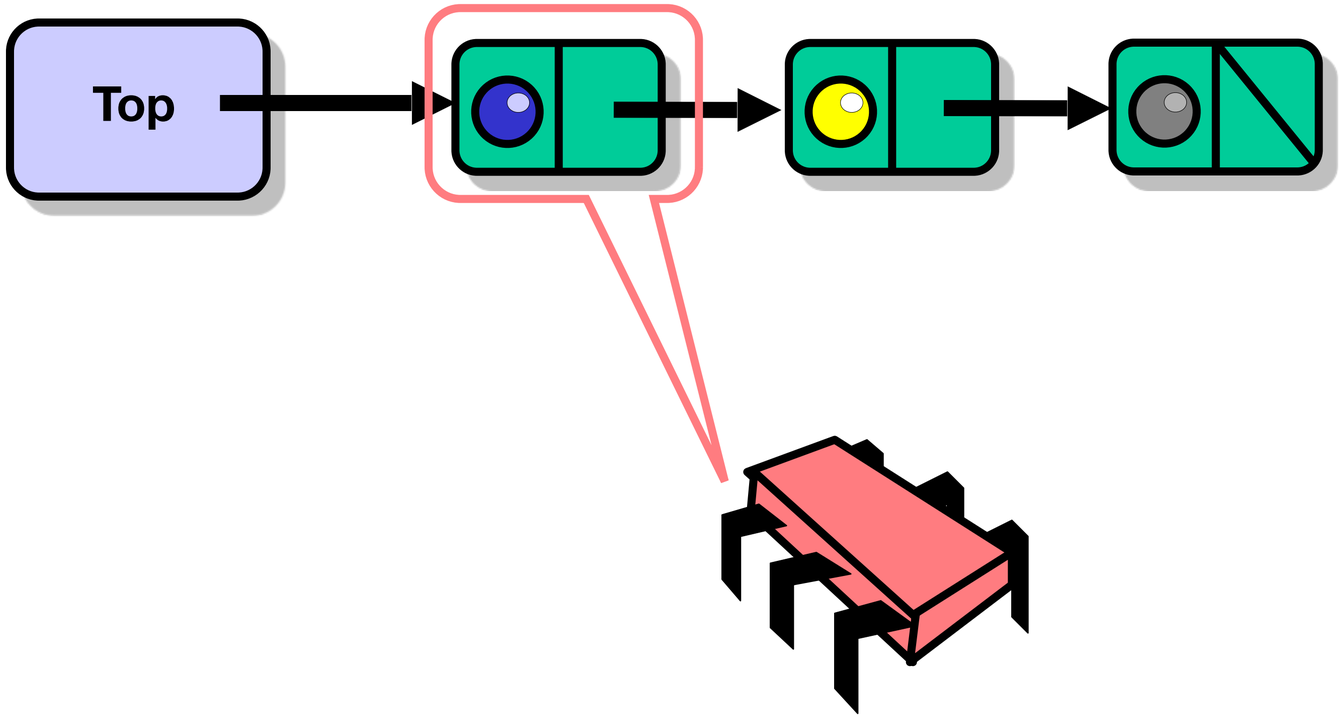
Push



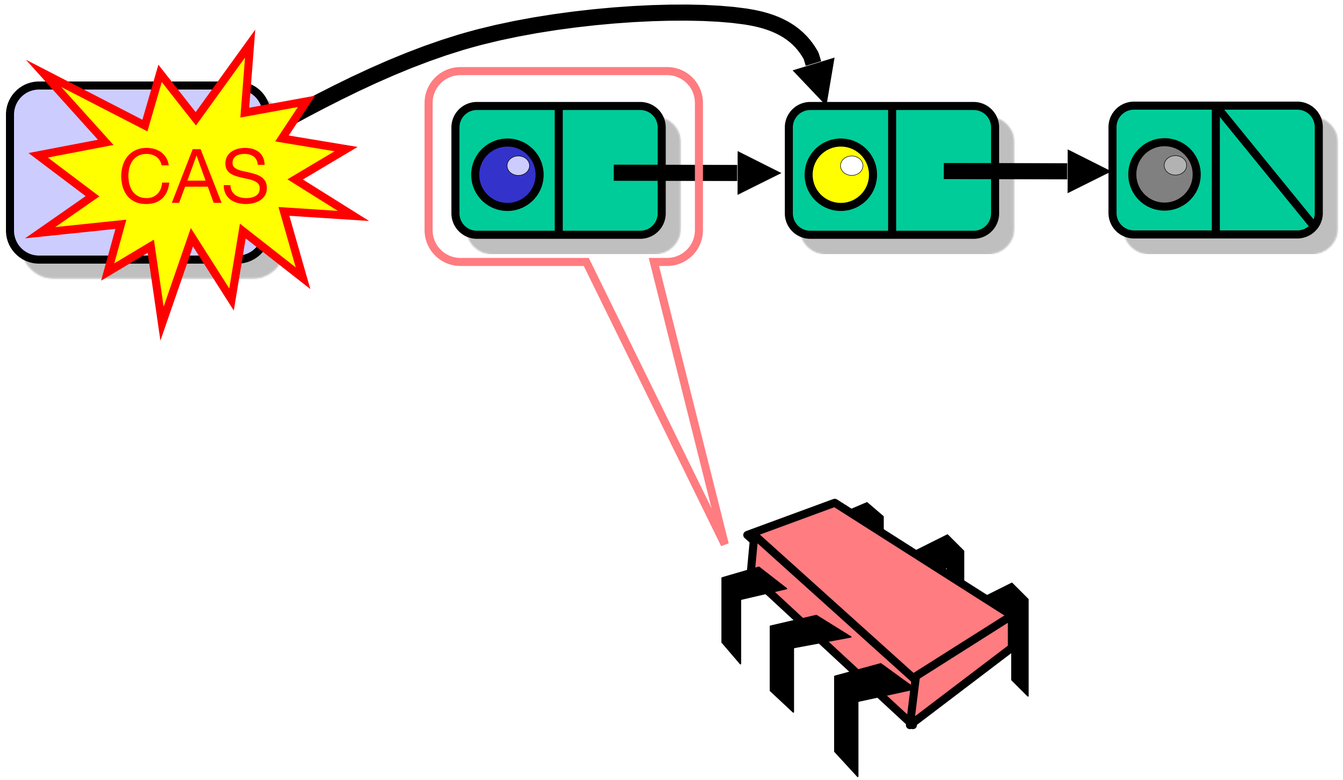
Push



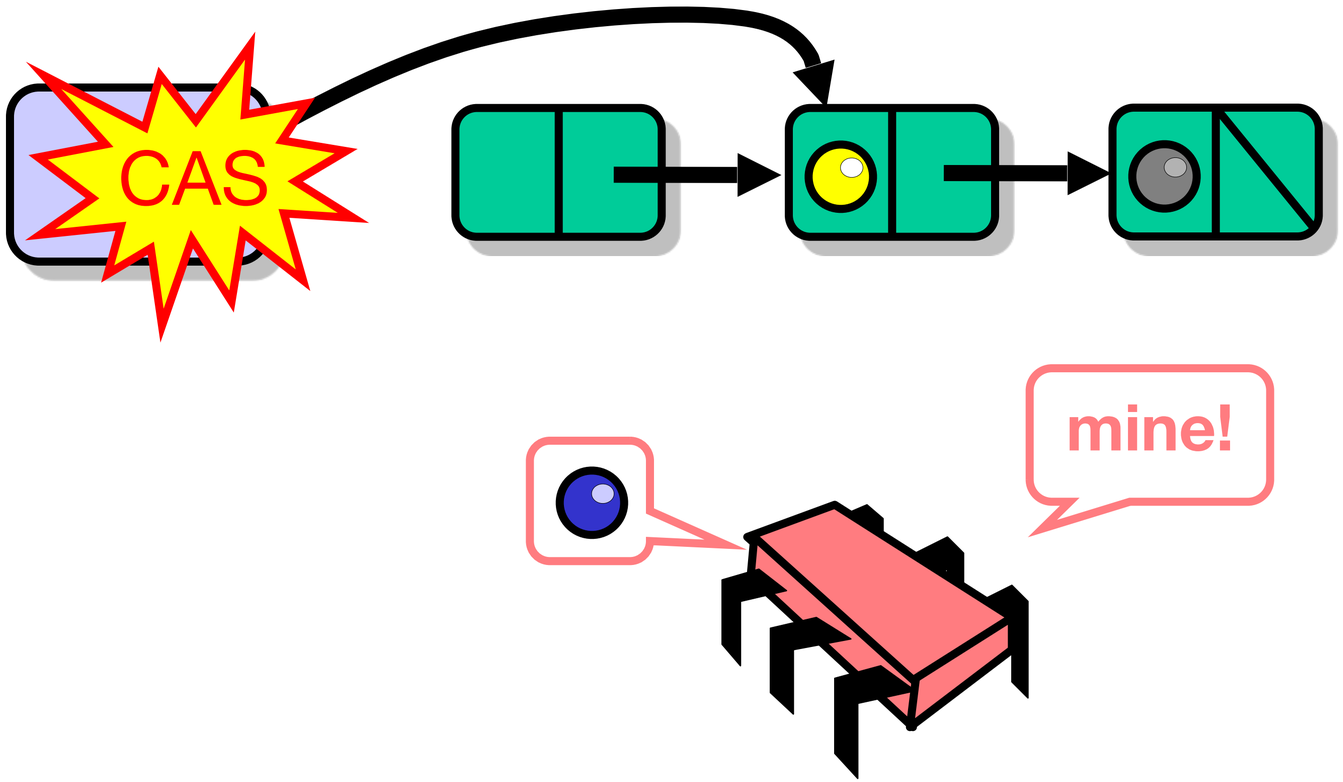
Pop



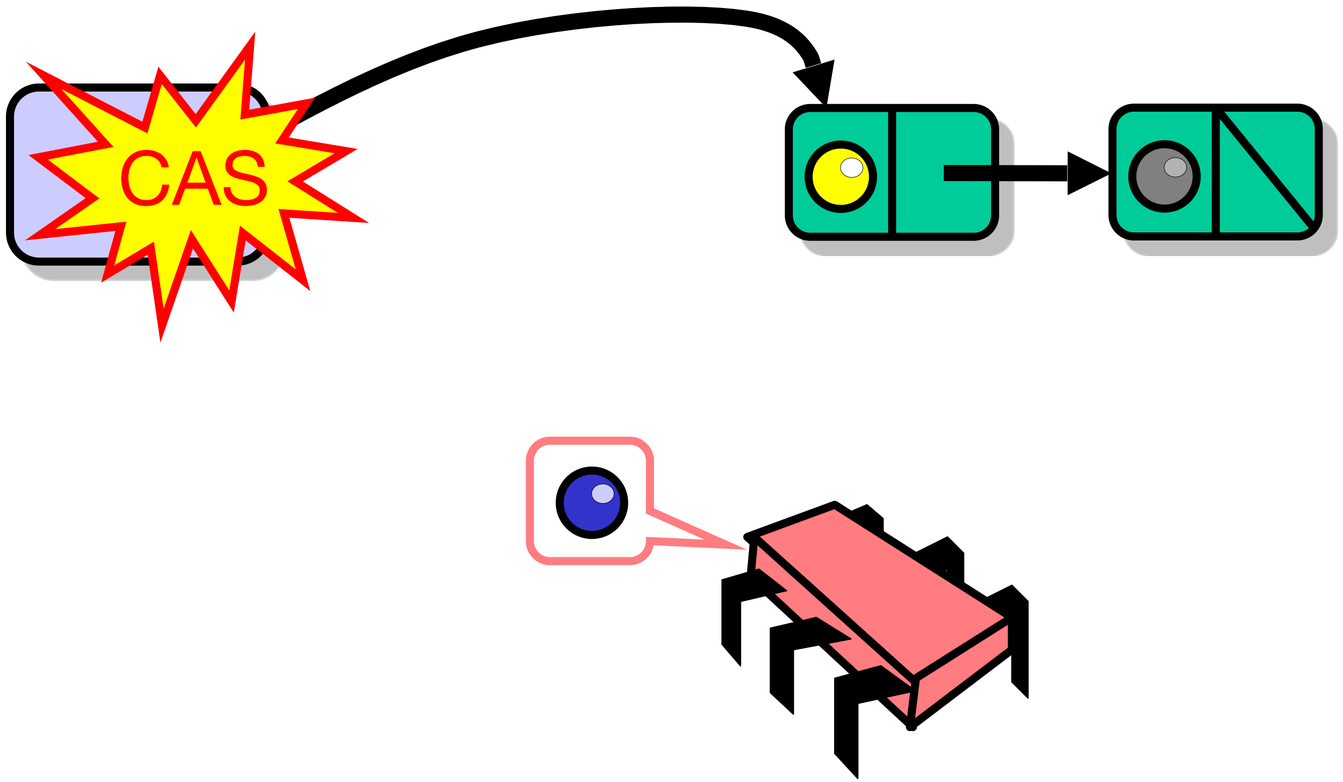
Pop



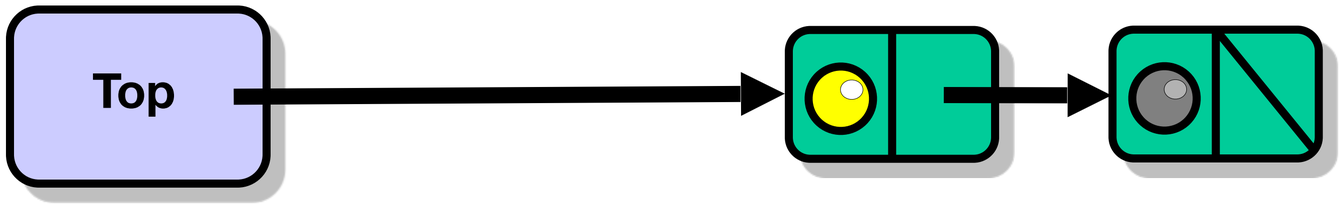
Pop

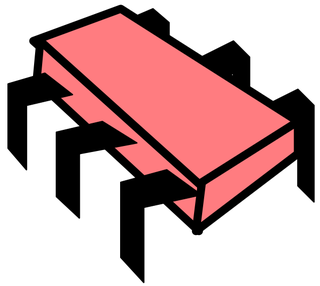


Pop



Pop





Walk through the code

lockfree_stack.ml

Walk through the code

lockfree_stack_builtin_list.ml

Lockfree Stack

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

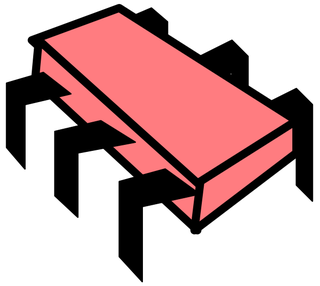
Big Question

- 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**

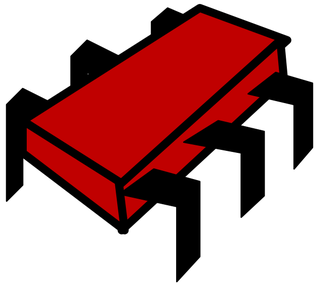
Observation



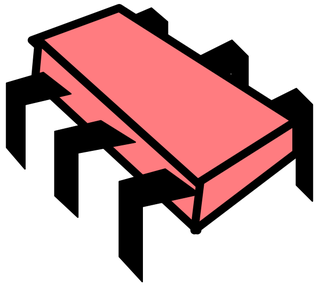
Push( )

linearizable stack

Pop()

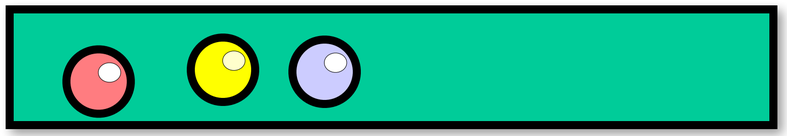


Observation

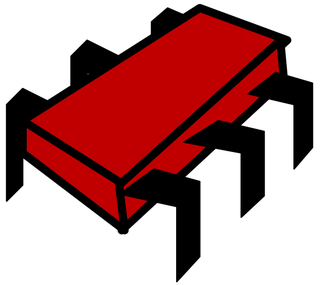


Push( )

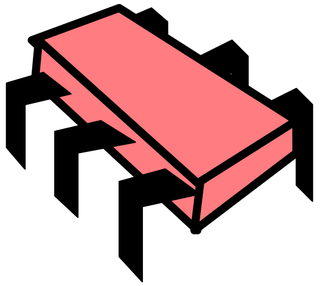
linearizable stack



Pop()



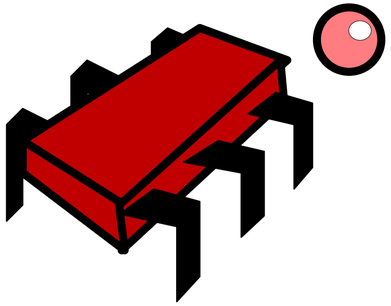
Observation



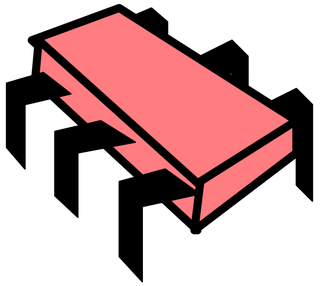
Push( )

linearizable stack

Pop()



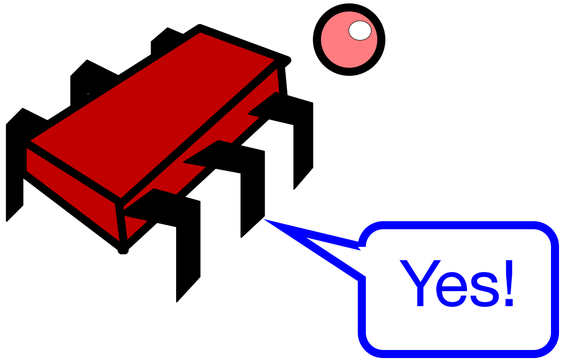
Observation



Push( )

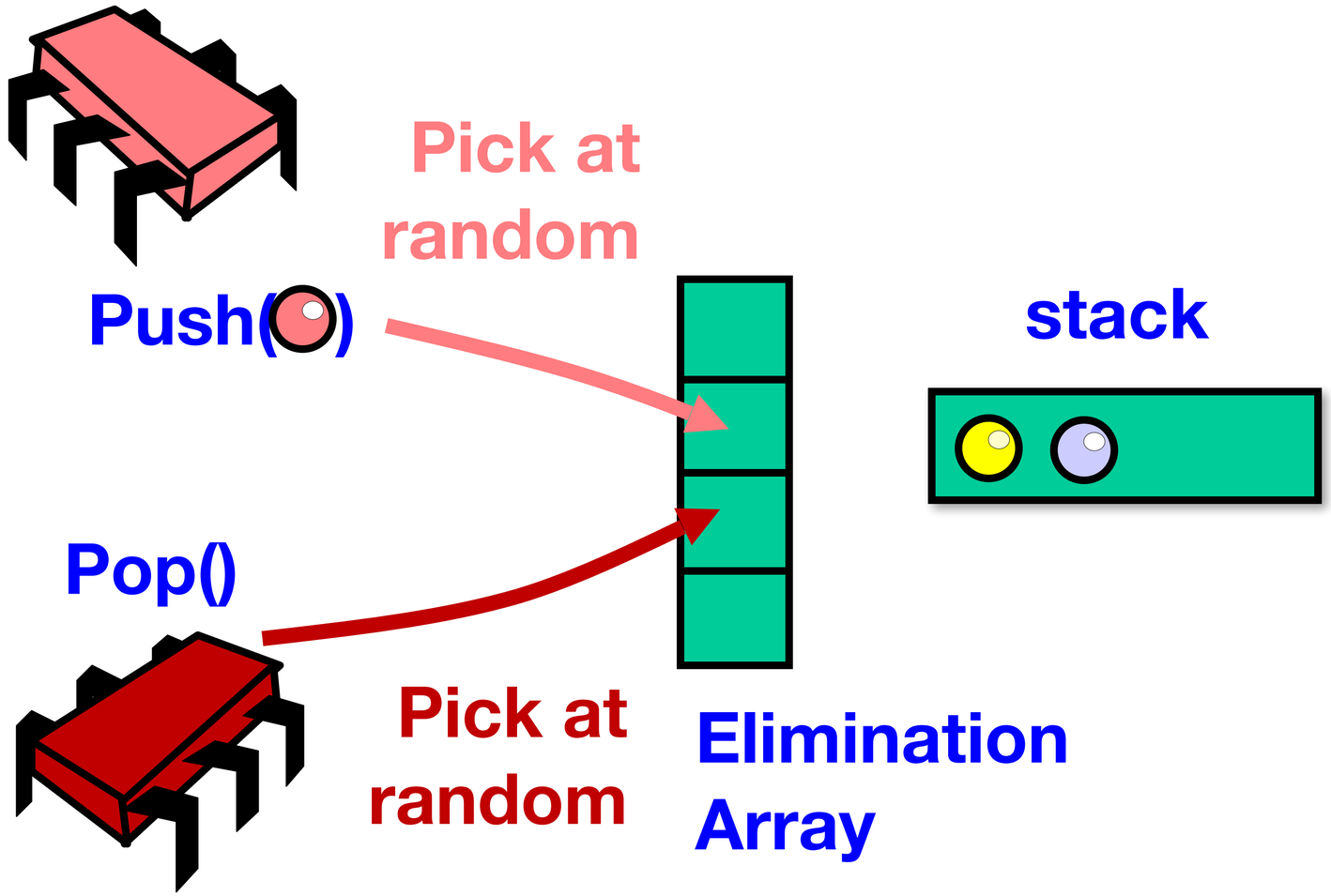
linearizable stack

Pop()

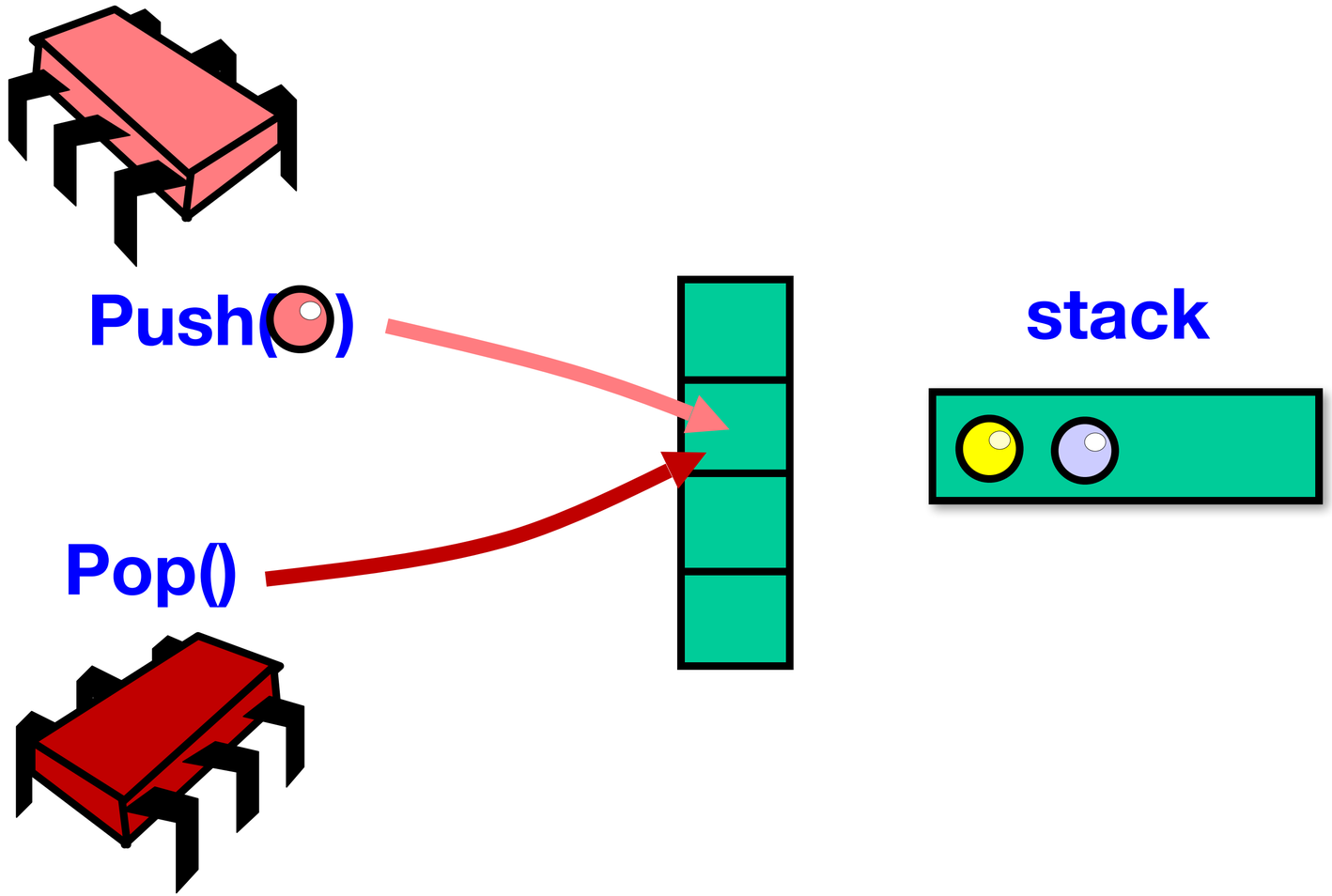


After an equal number of pushes and pops, stack stays the same

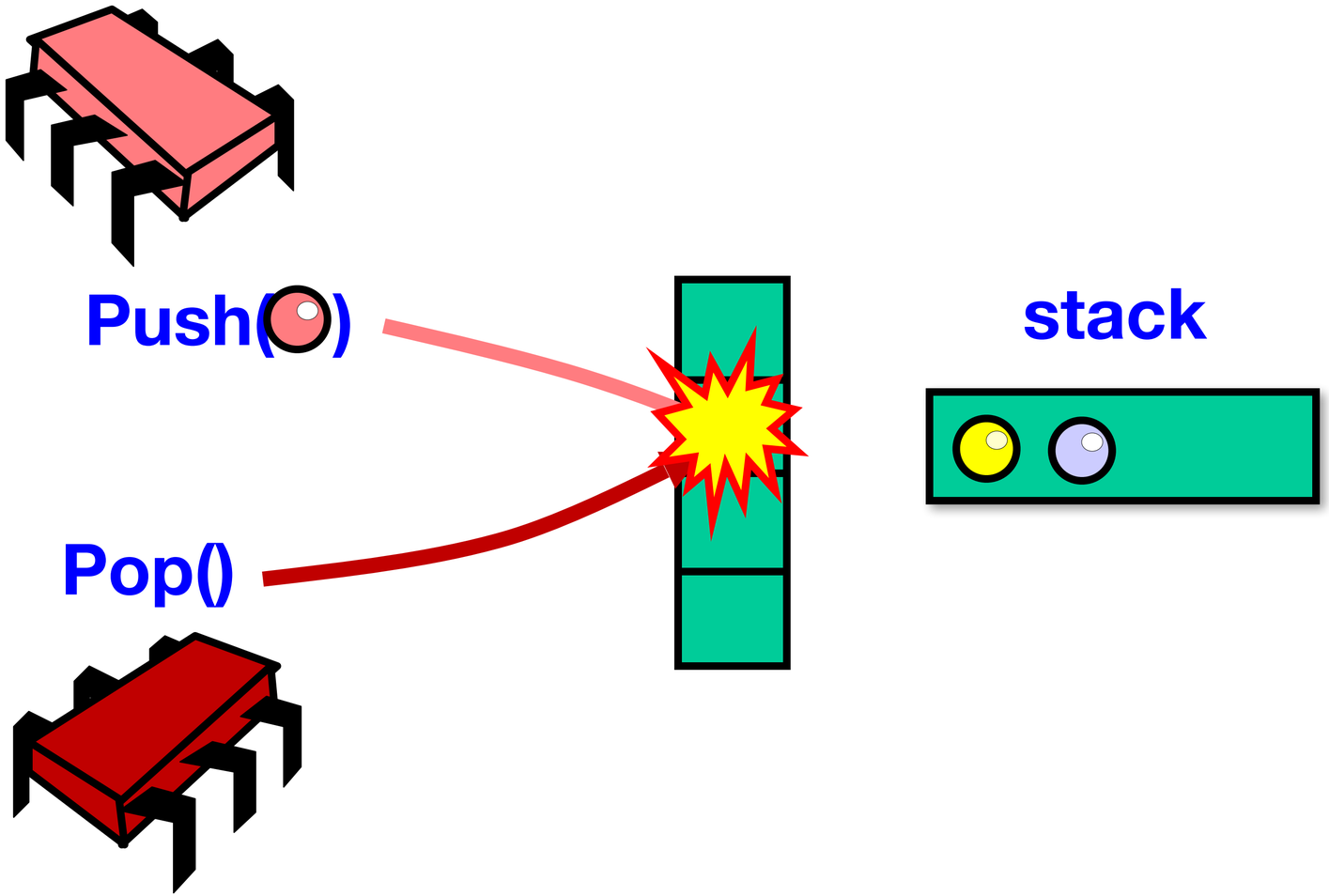
Idea: Elimination Array



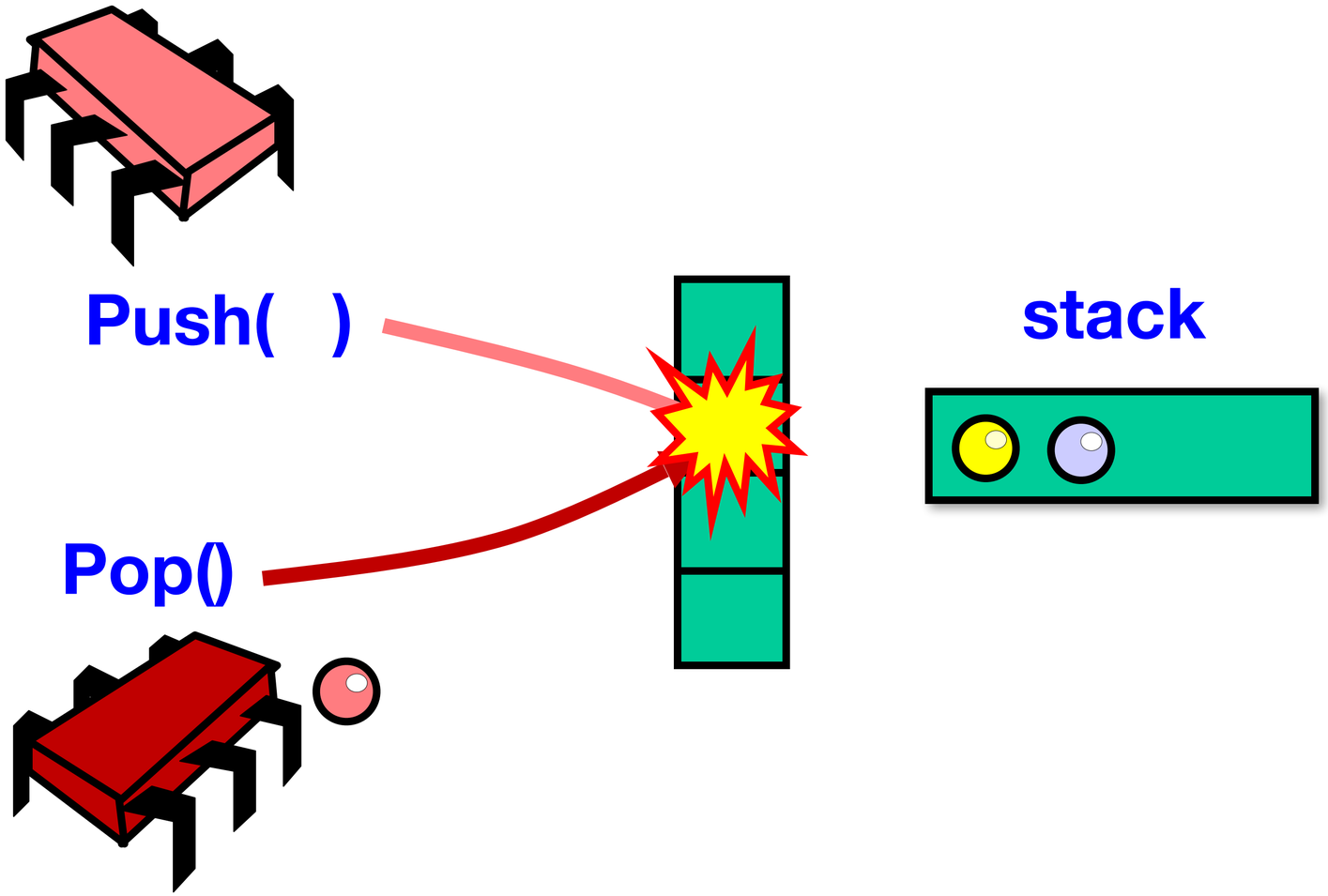
Push Collides With Pop



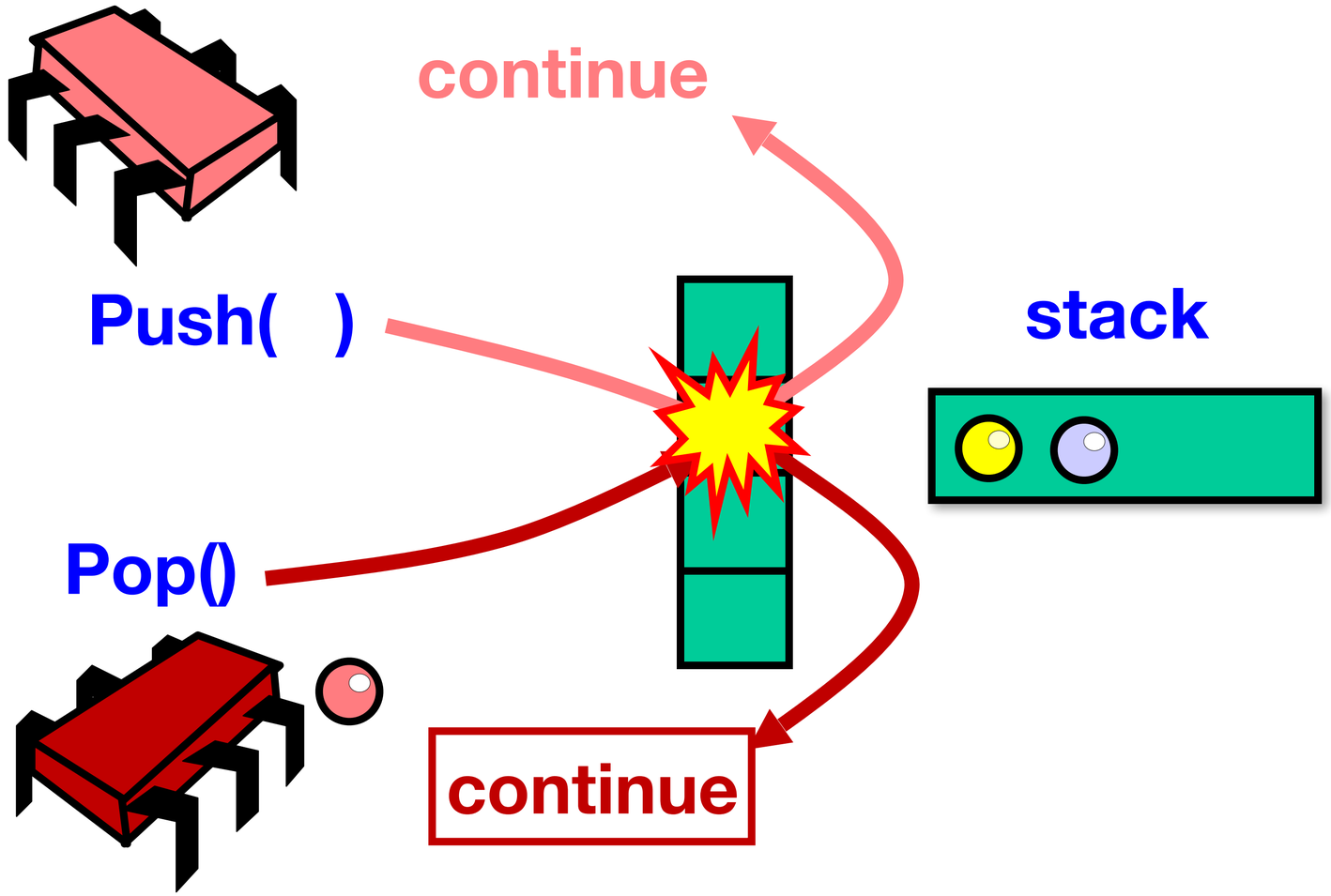
Push Collides With Pop



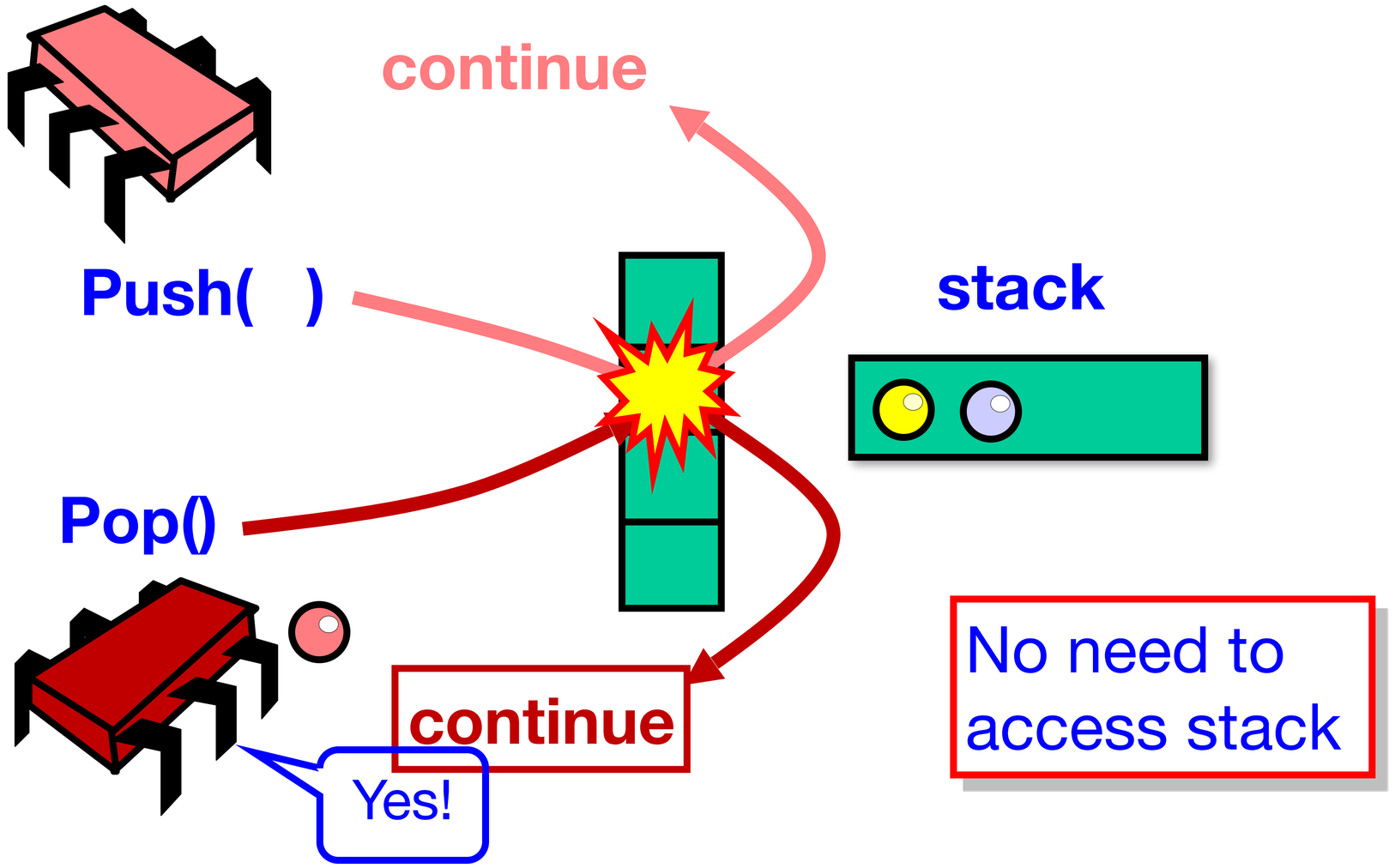
Push Collides With Pop



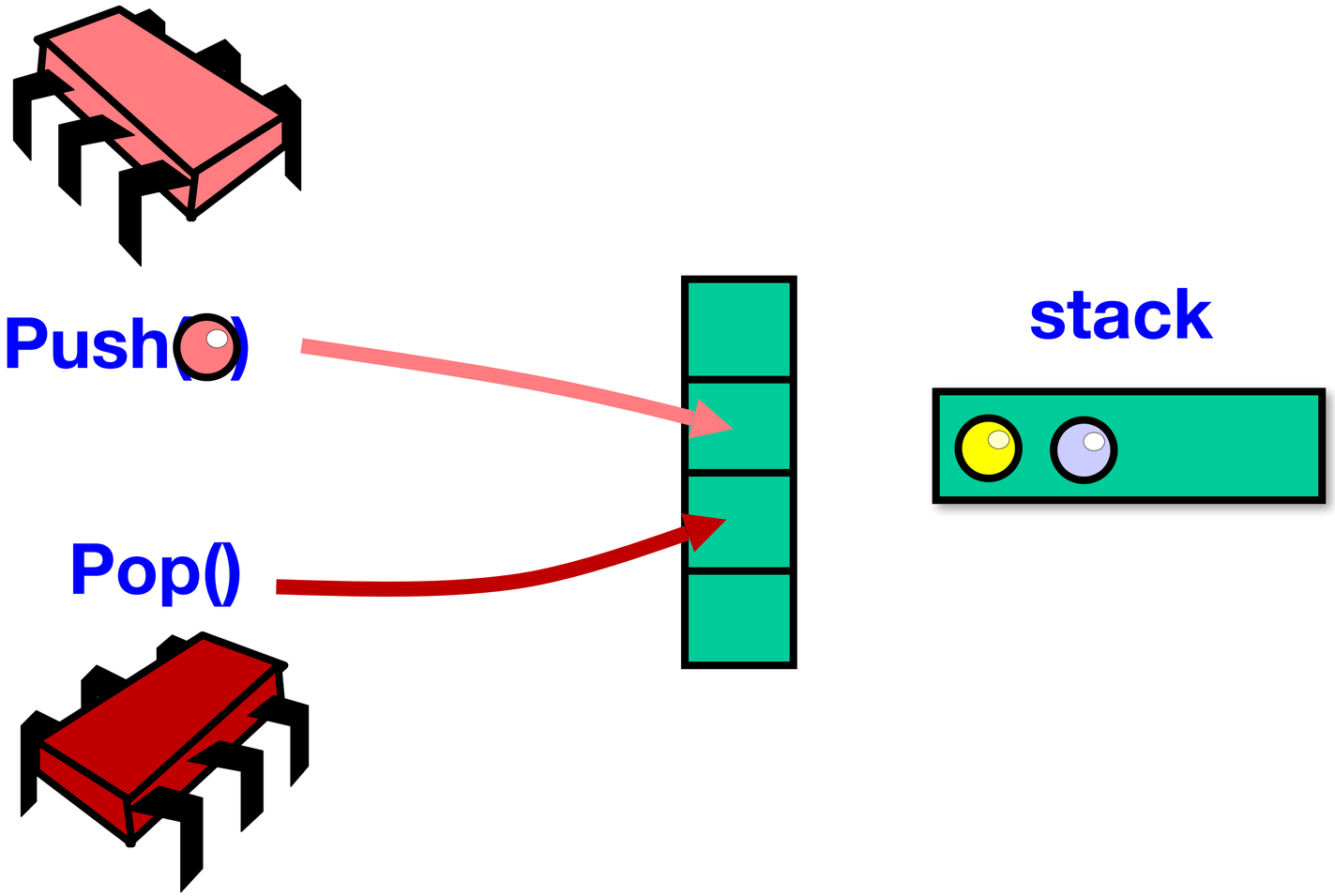
Push Collides With Pop



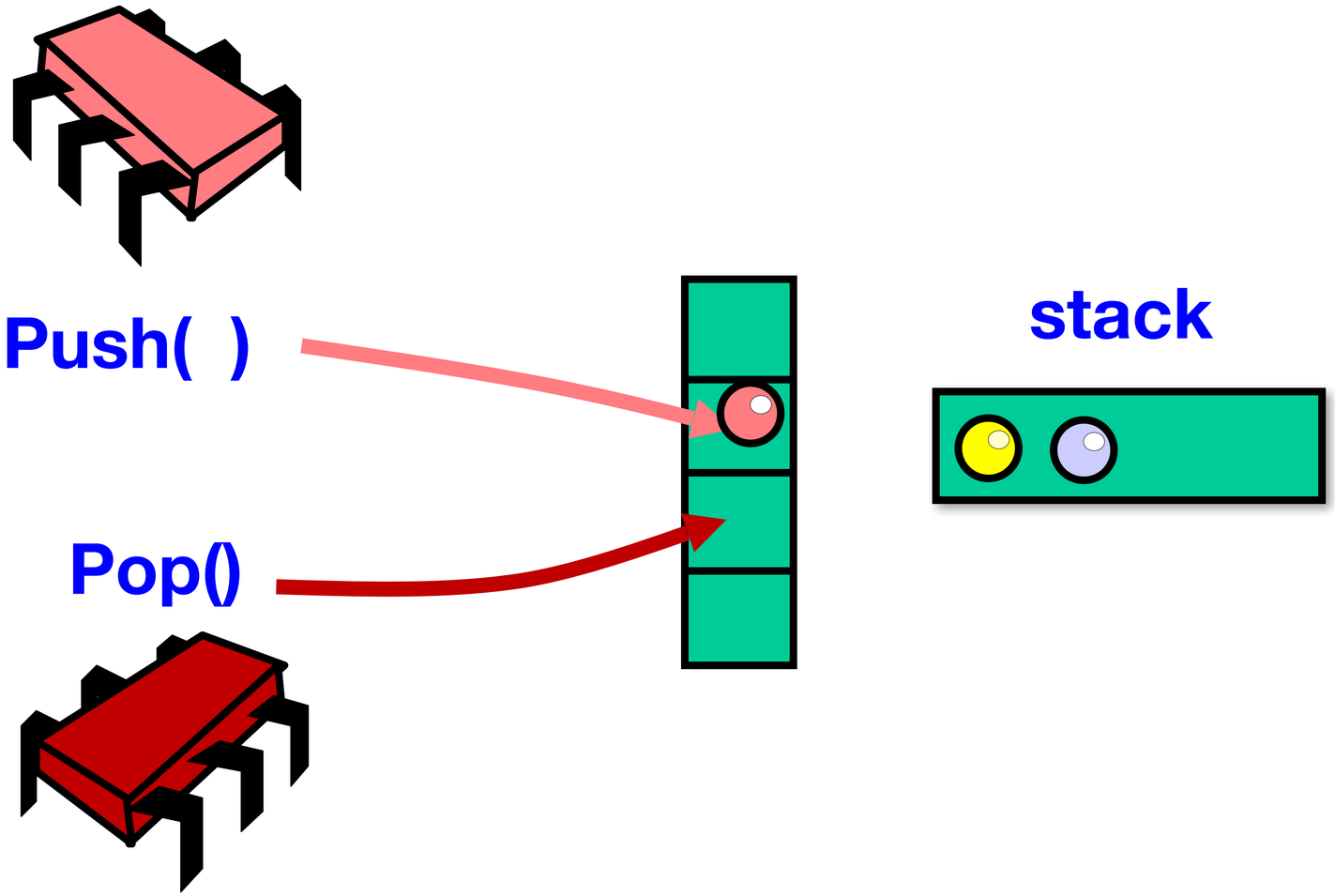
Push Collides With Pop



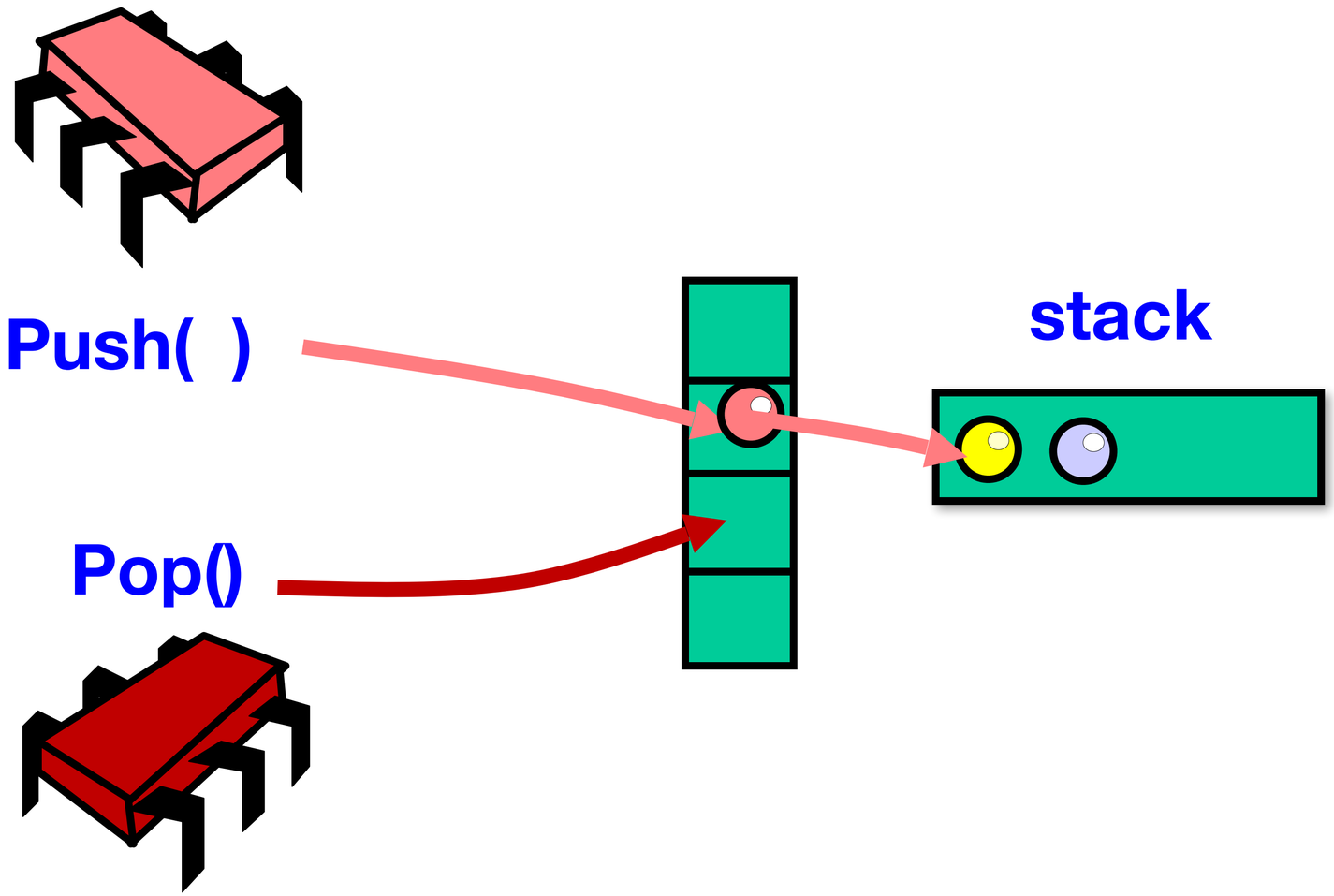
No Collision



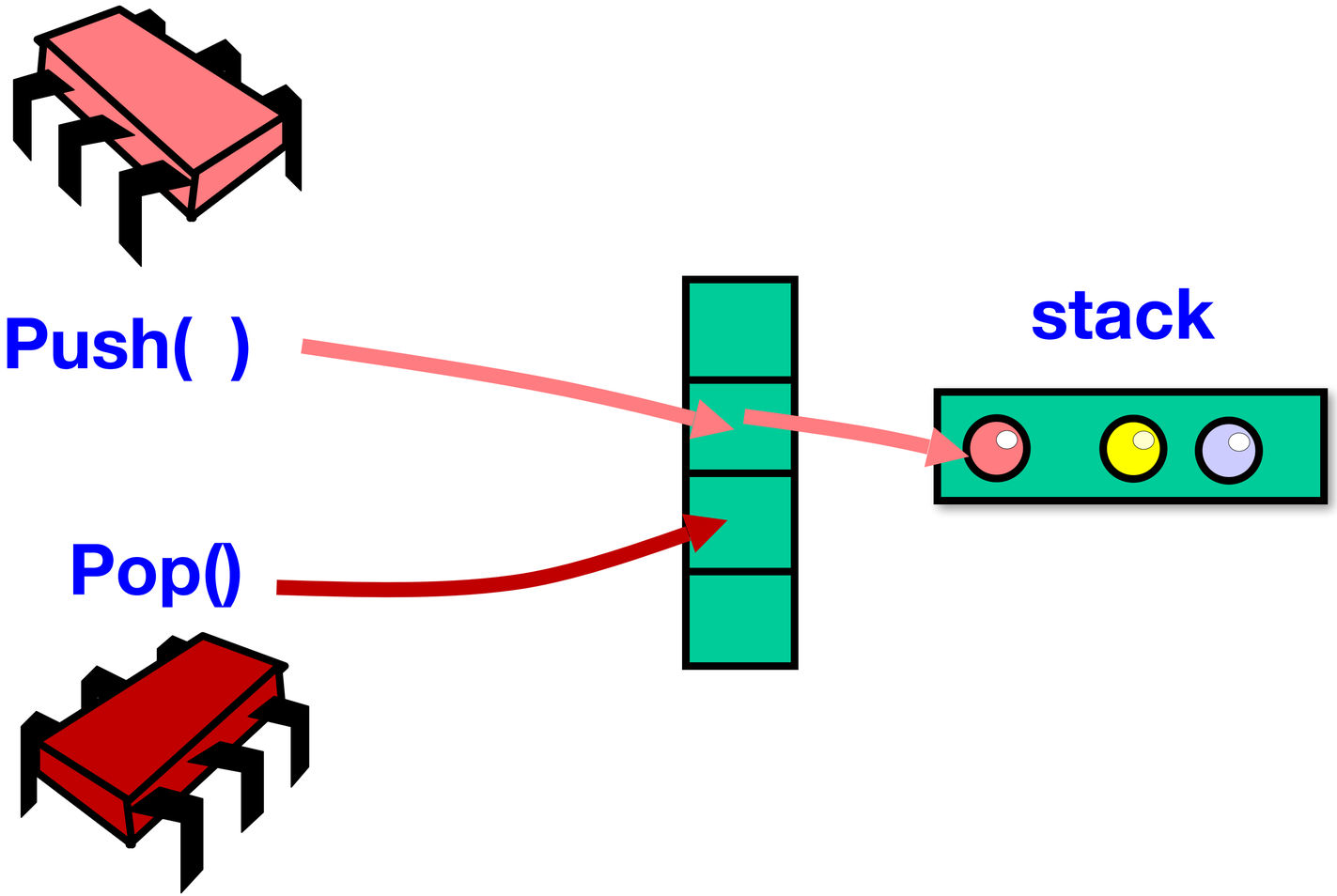
No Collision



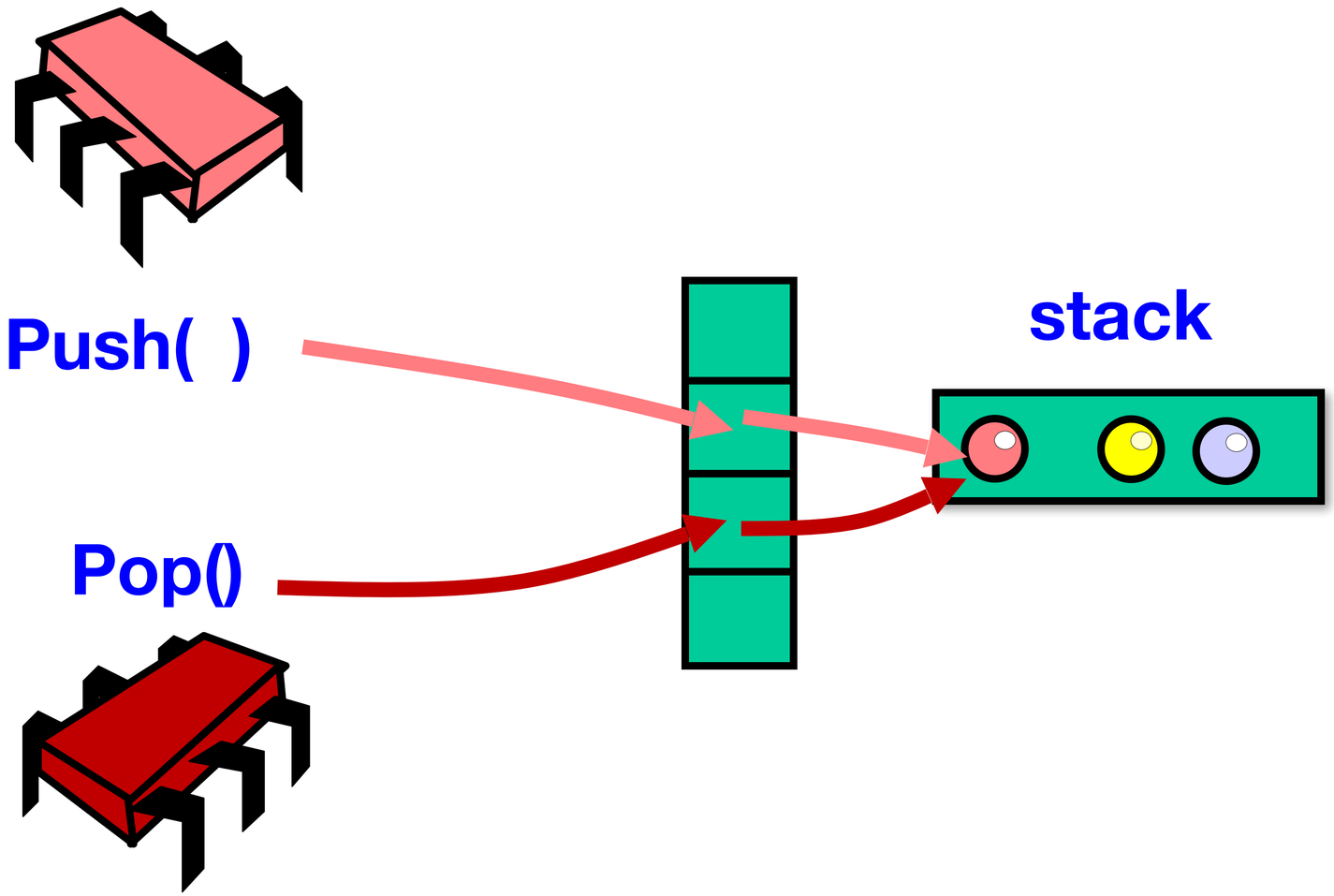
No Collision



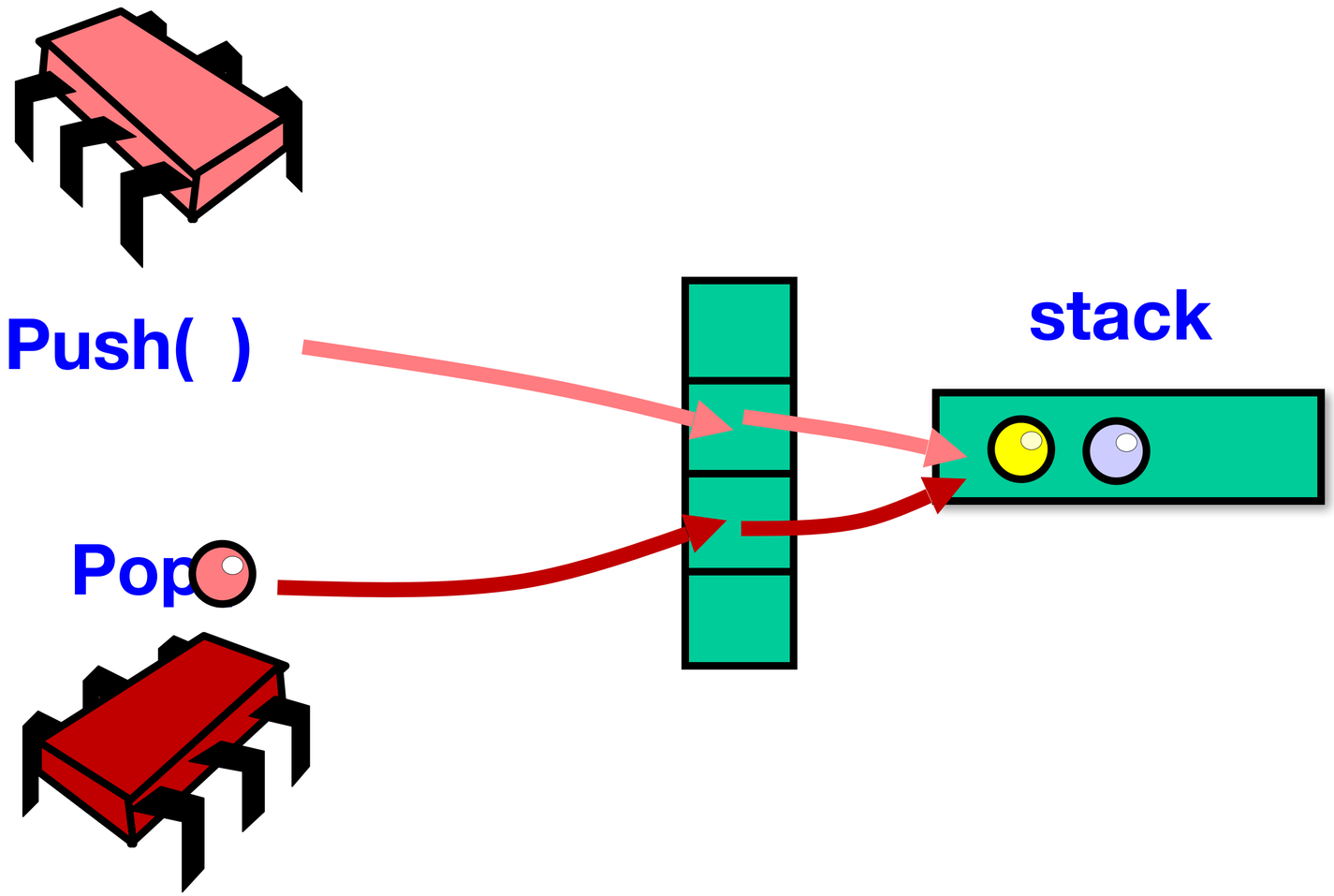
No Collision



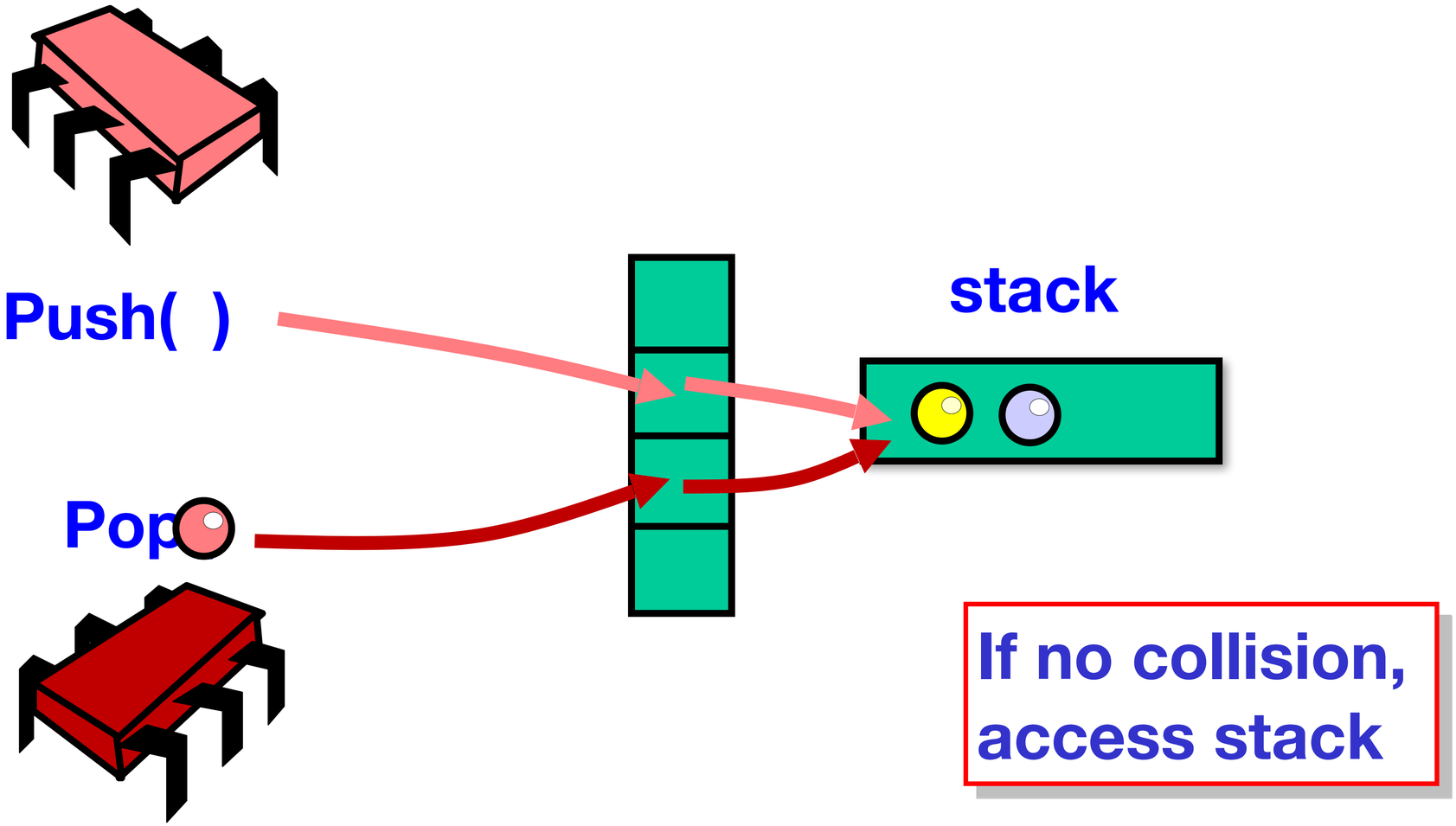
No Collision



No Collision

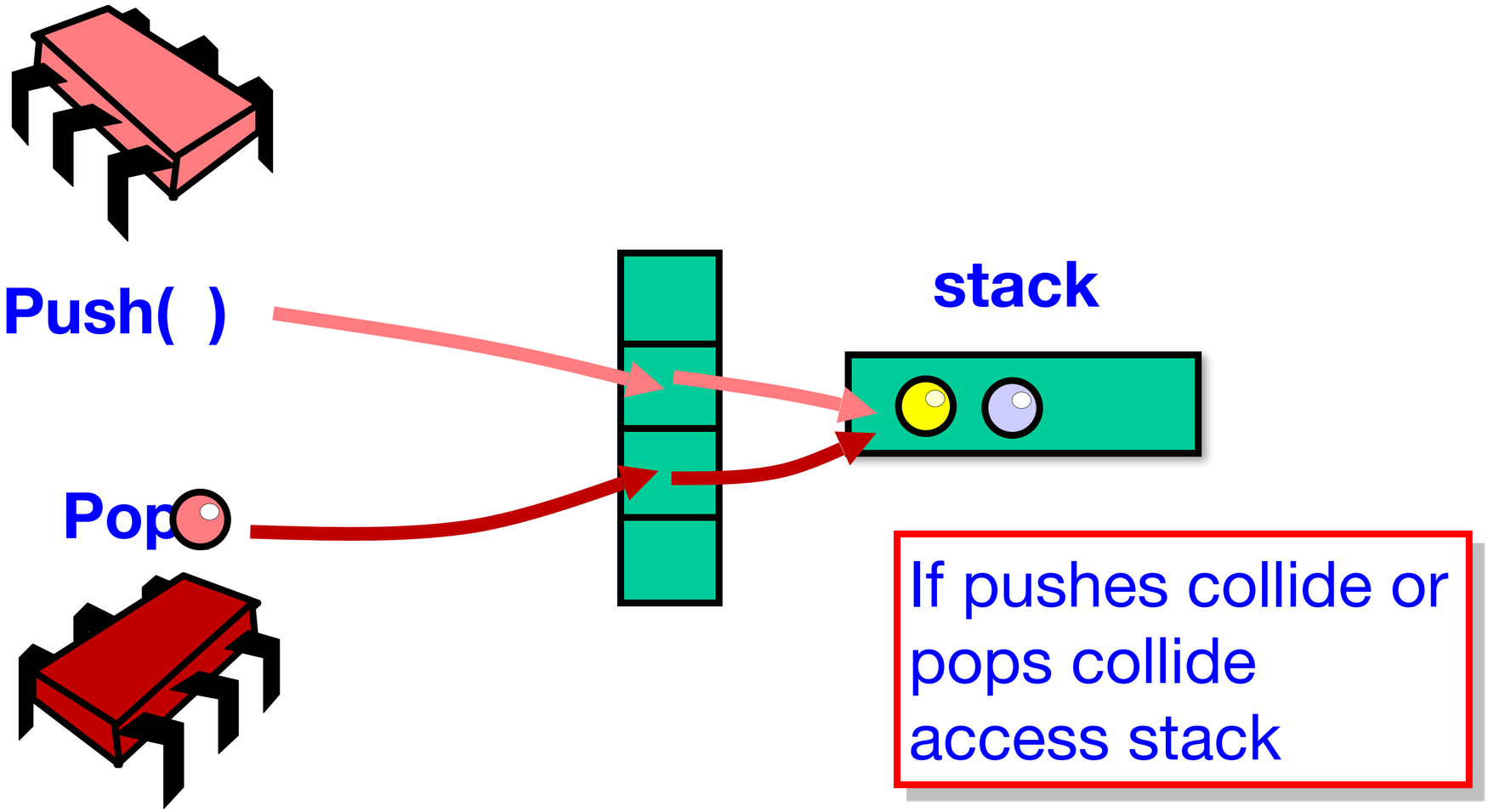


No Collision



If no collision, access stack

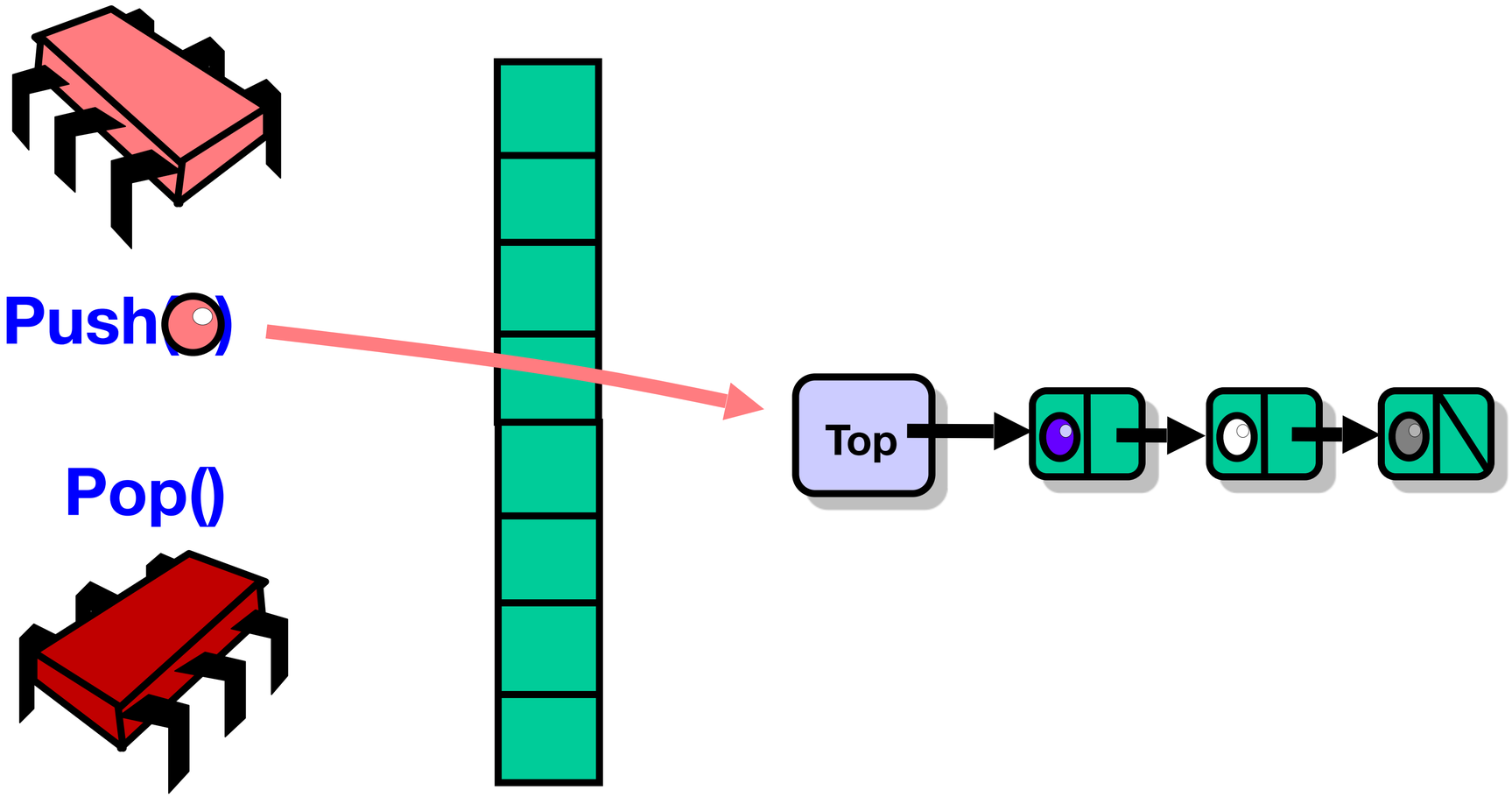
No Collision



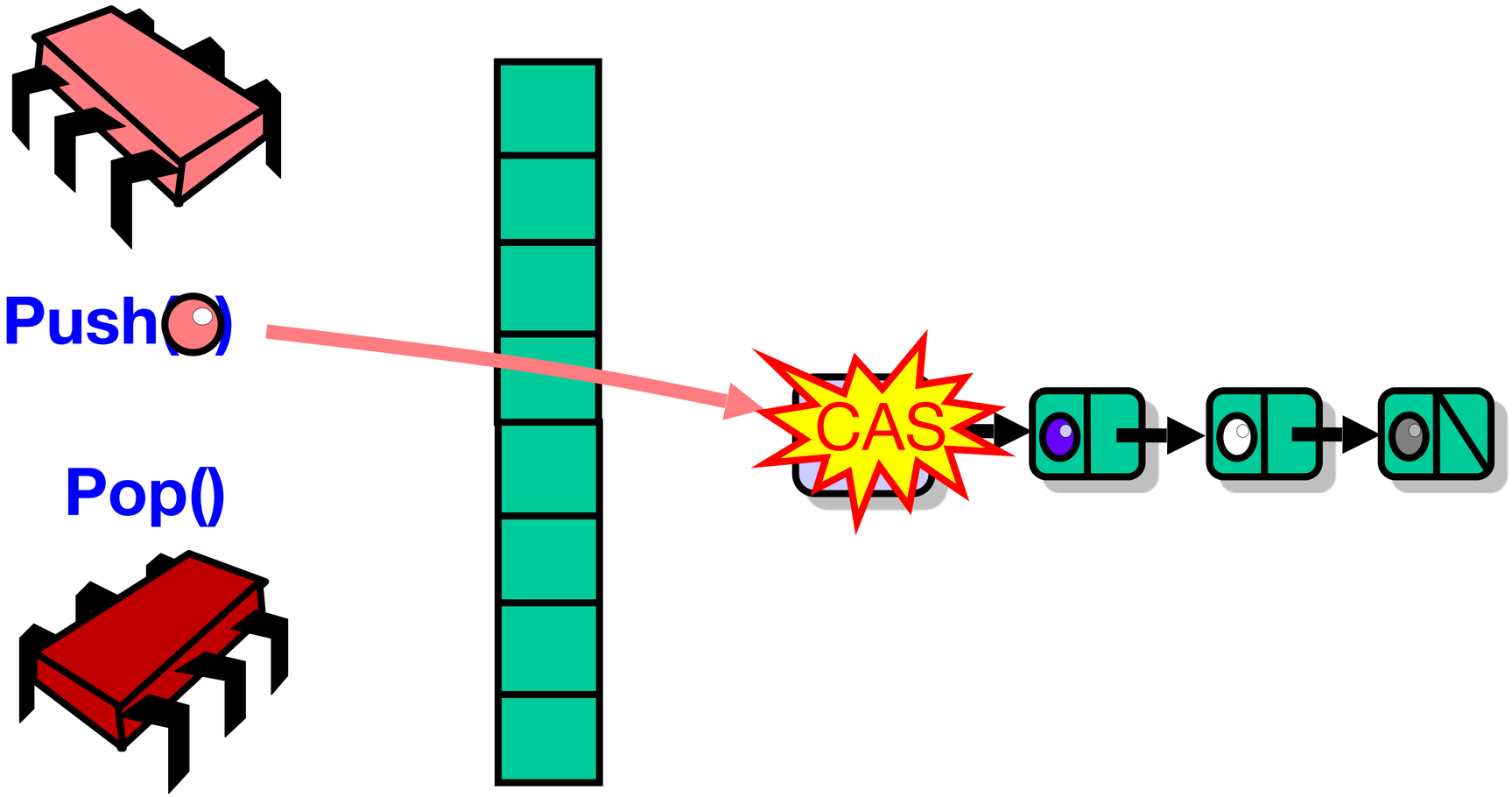
Elimination-backoff Stack

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

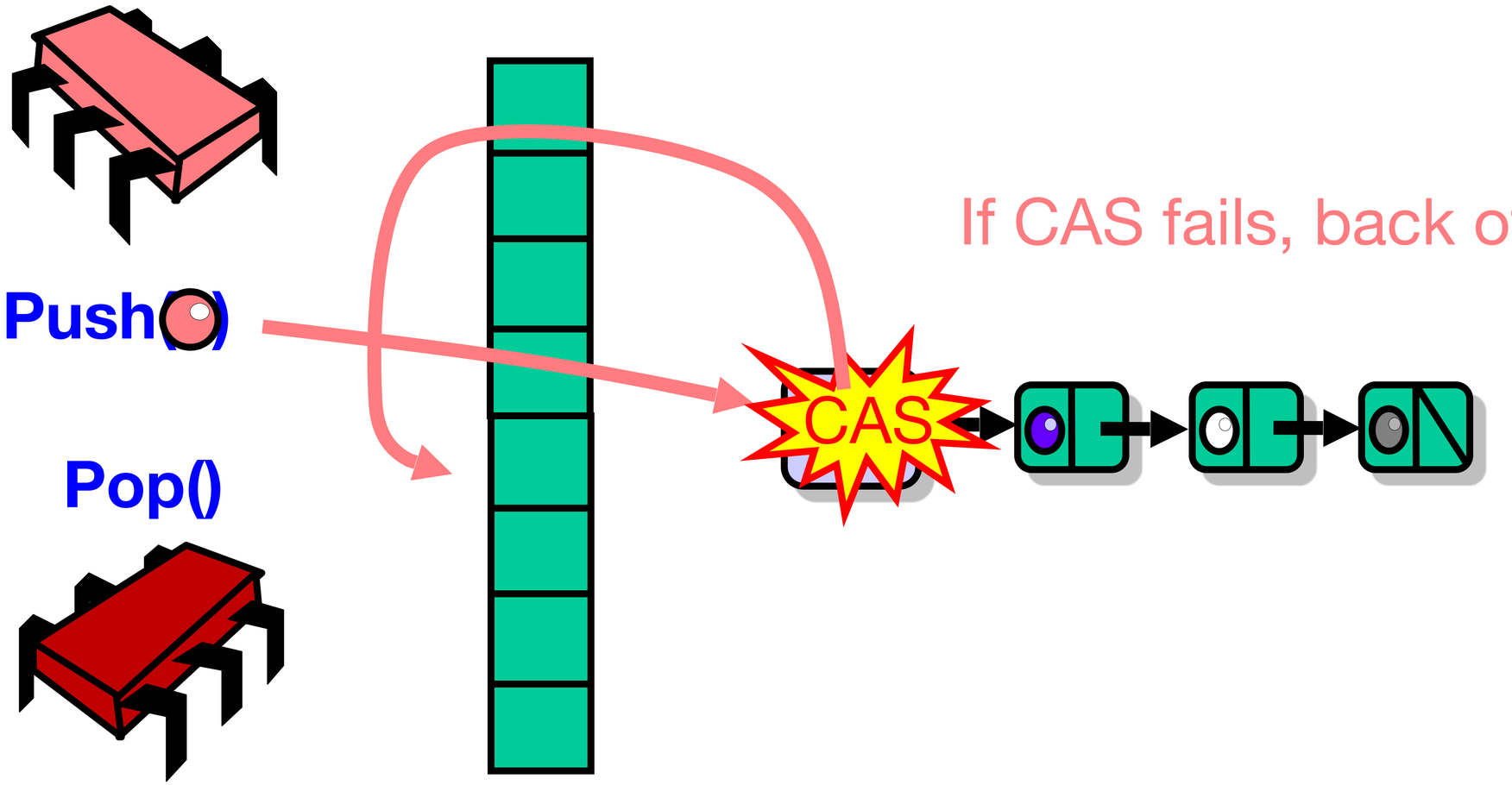
Elimination-backoff Stack



Elimination-backoff Stack

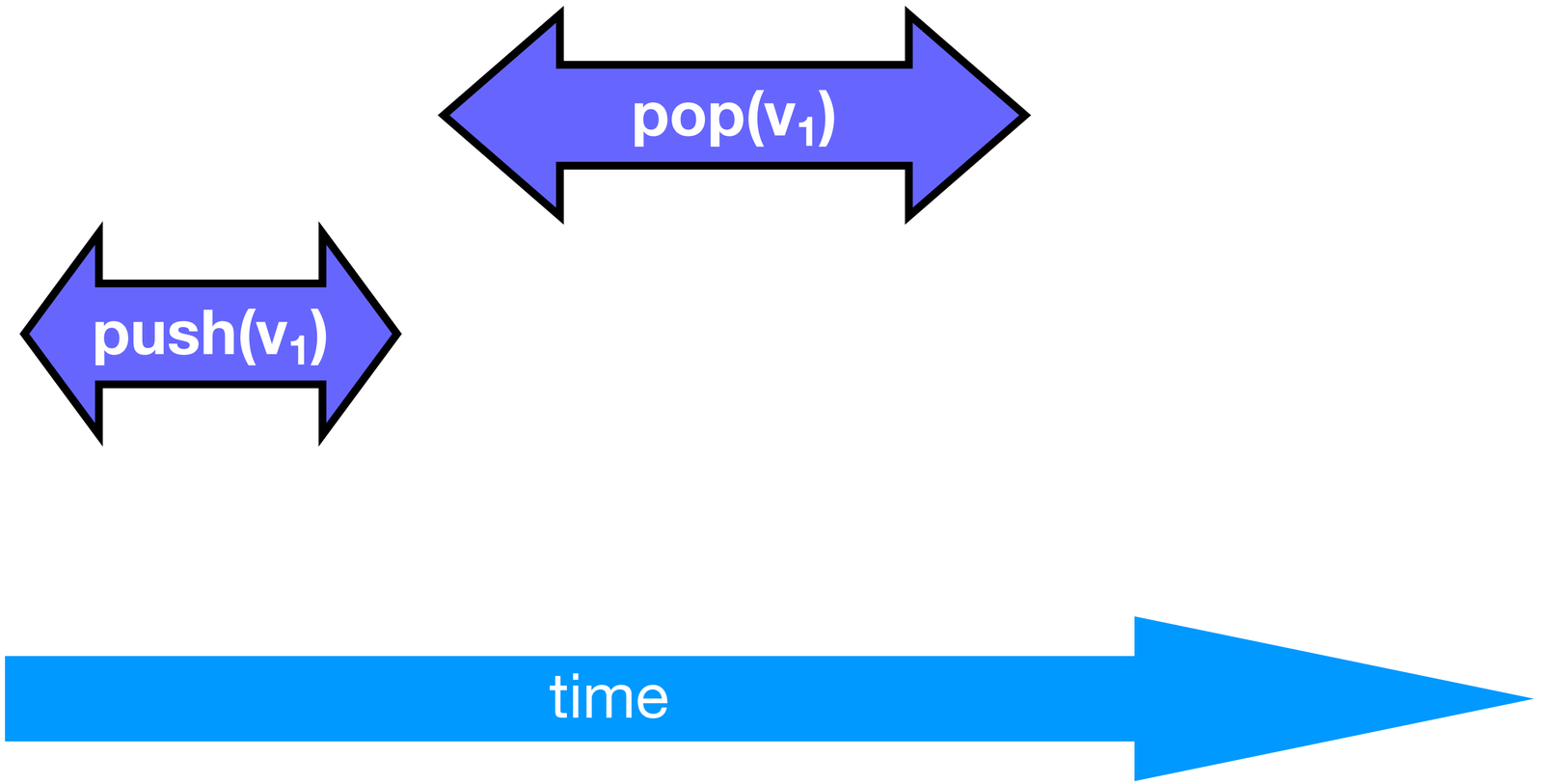


Elimination-backoff Stack

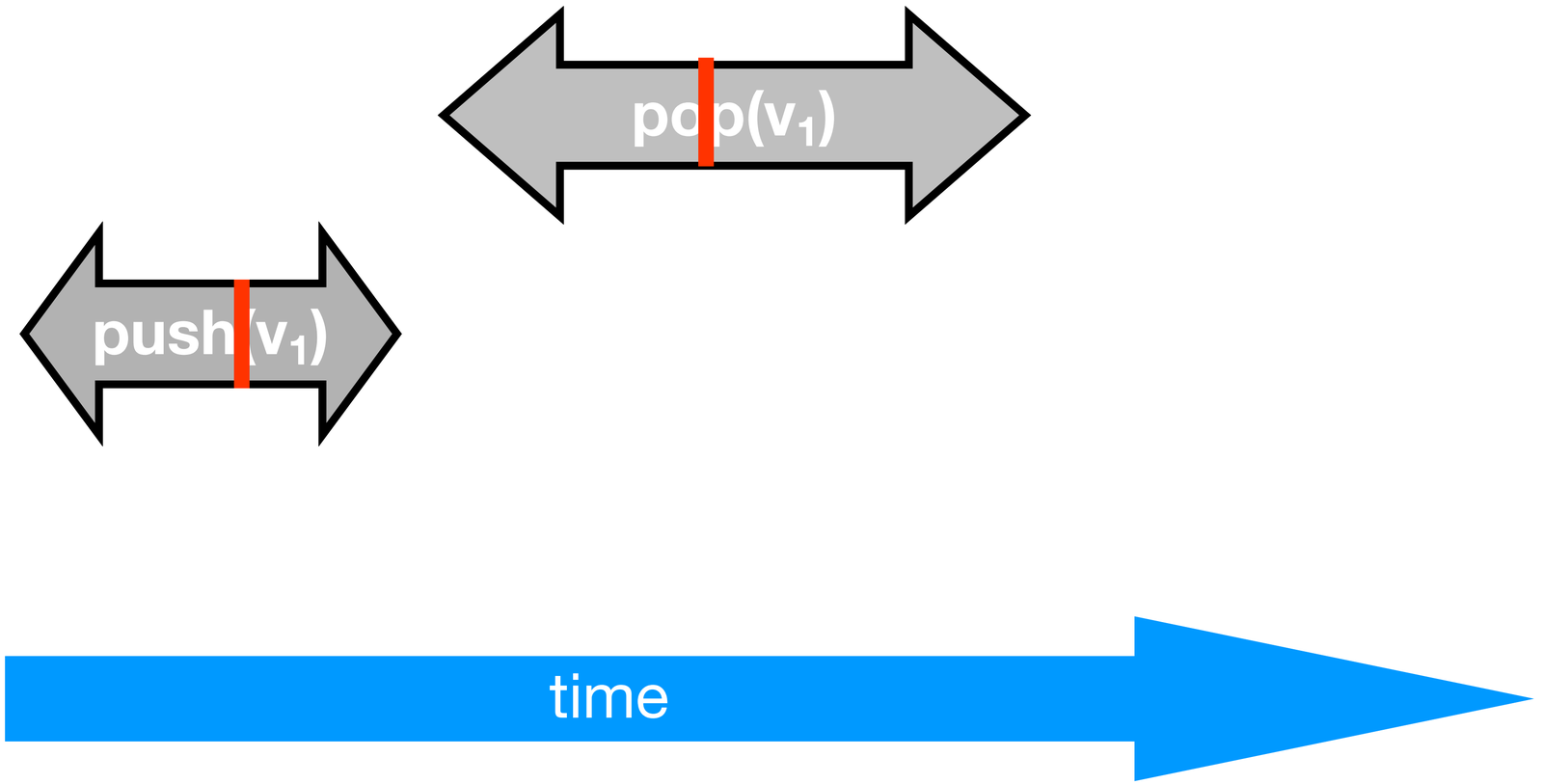


Uneliminated Linearizability

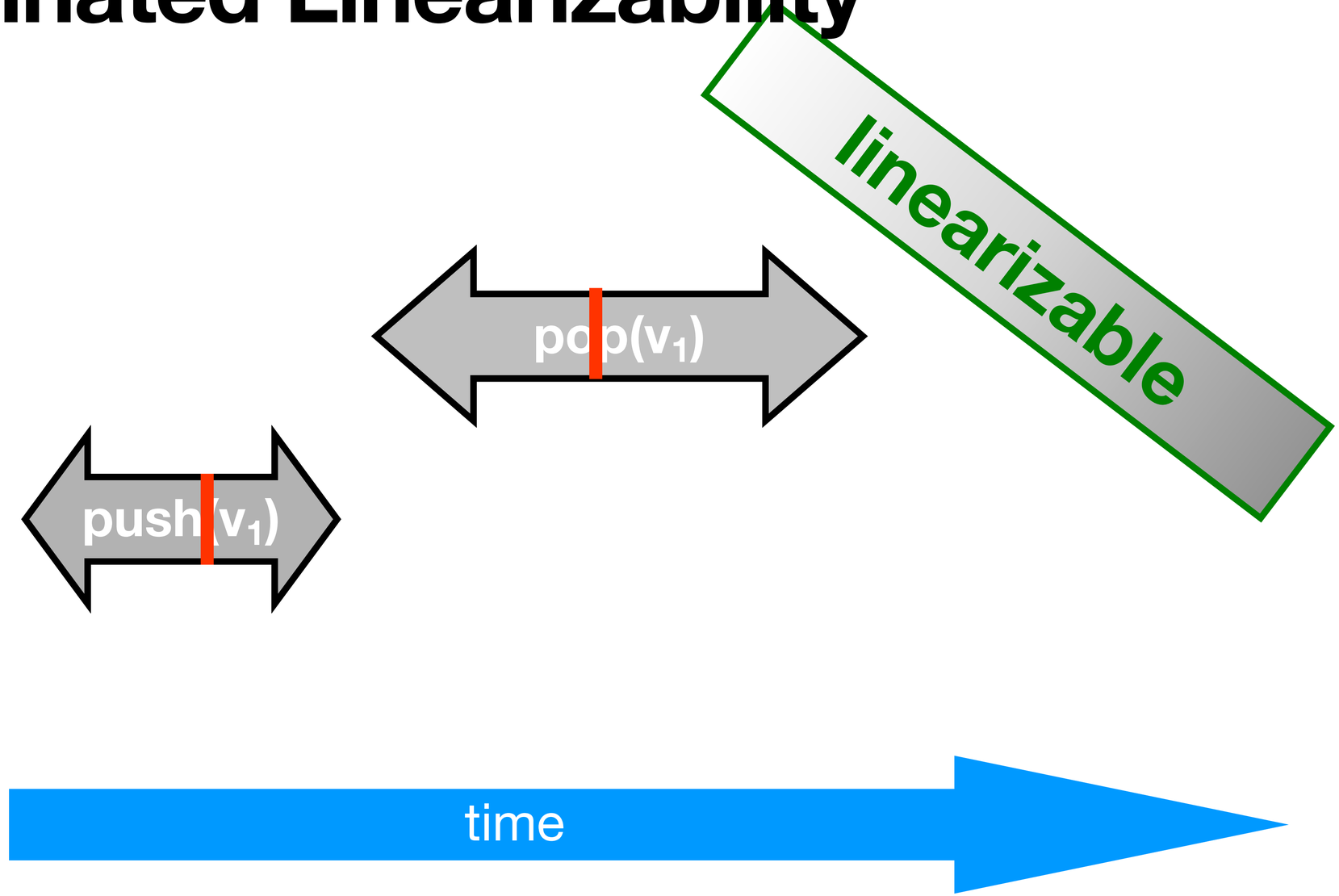
Uneliminated Linearizability



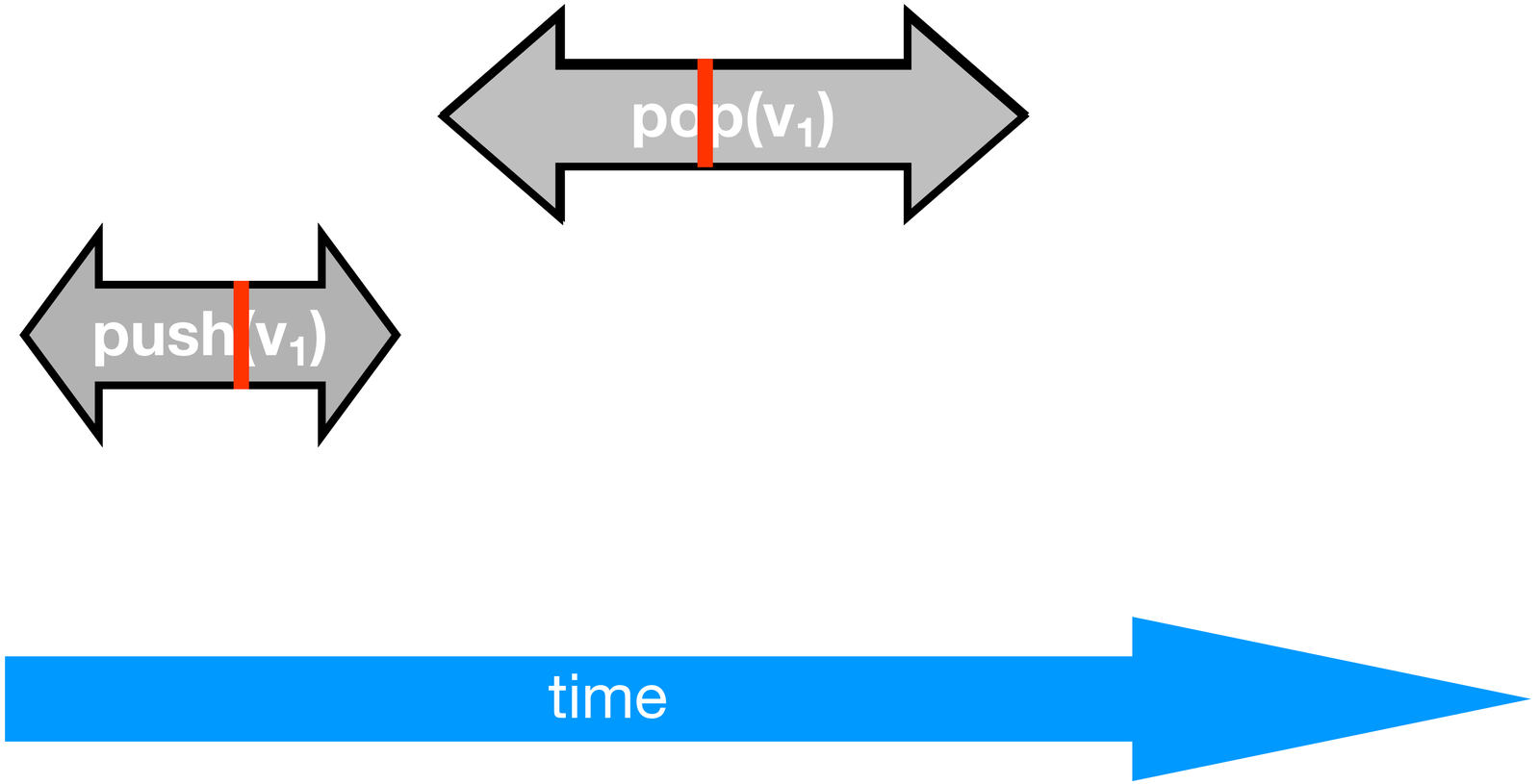
Uneliminated Linearizability



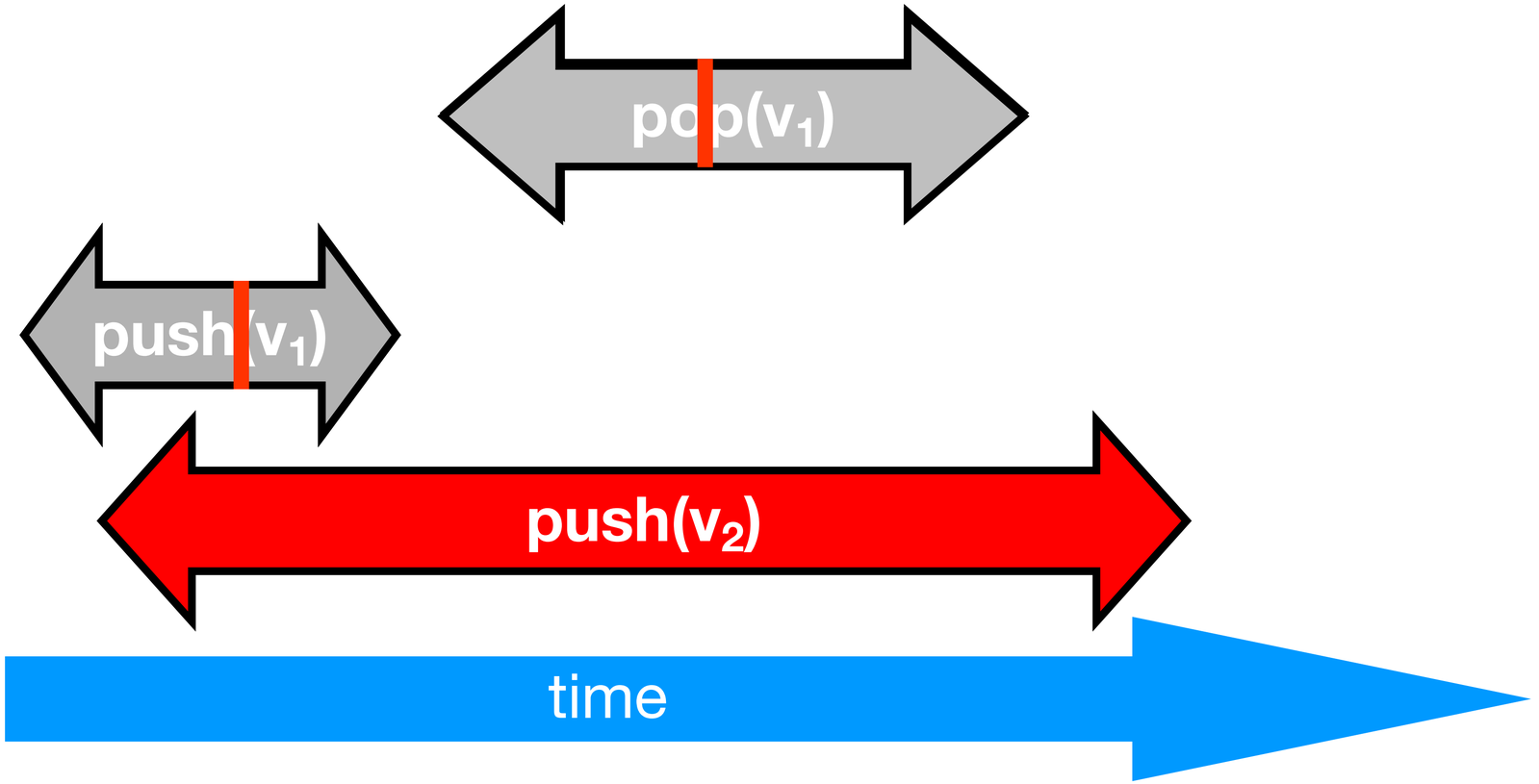
Uneliminated Linearizability



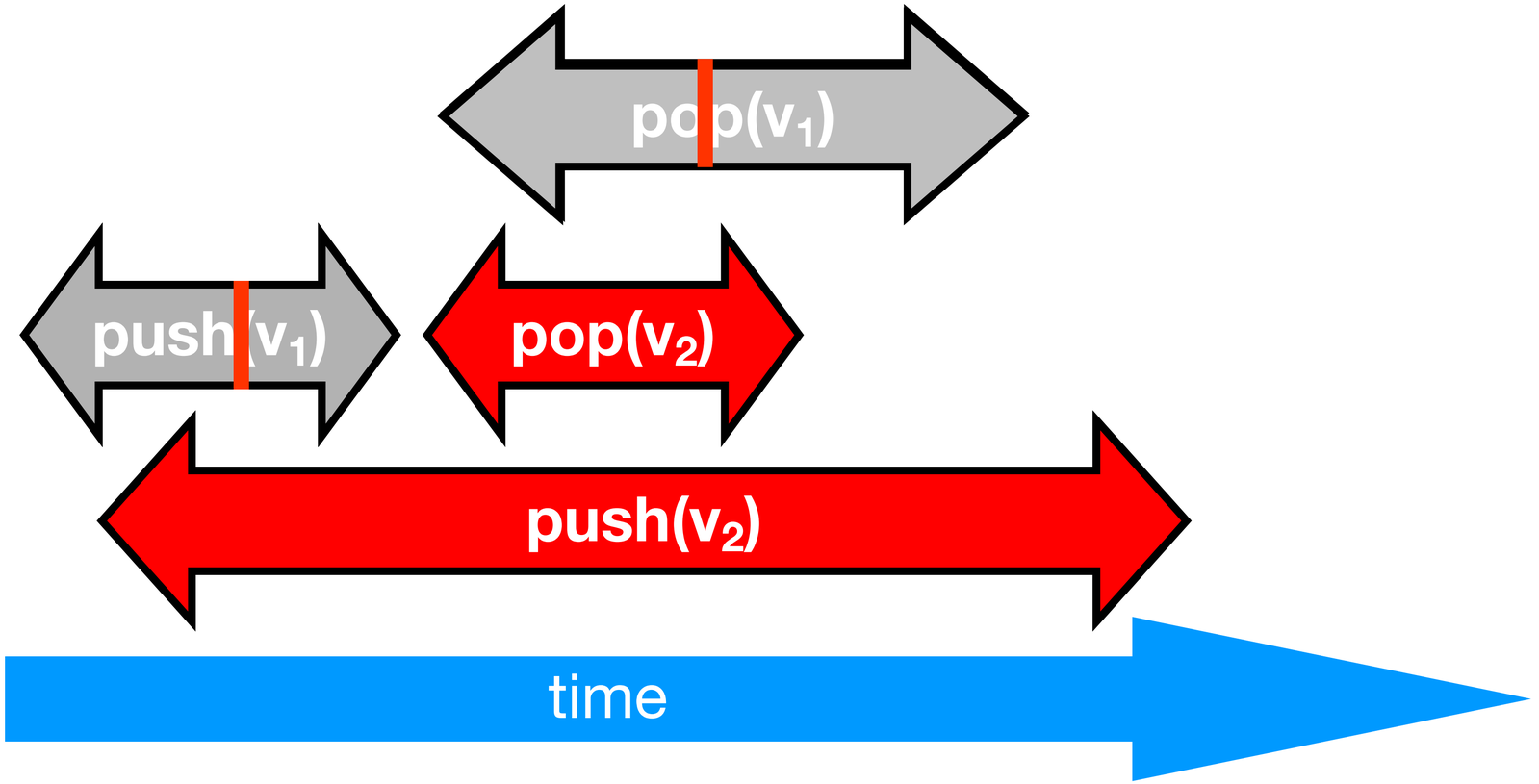
Eliminated Linearizability



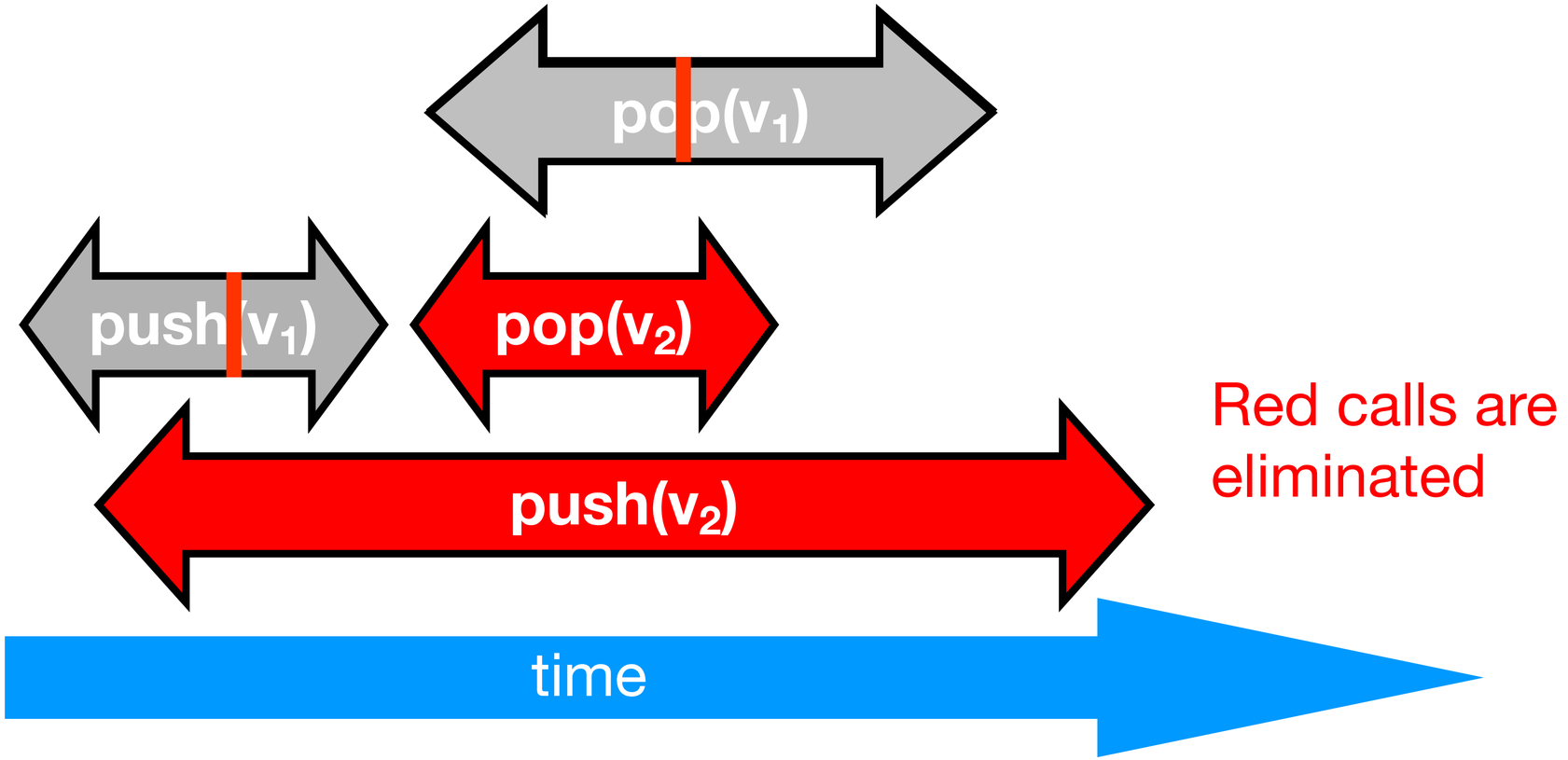
Eliminated Linearizability



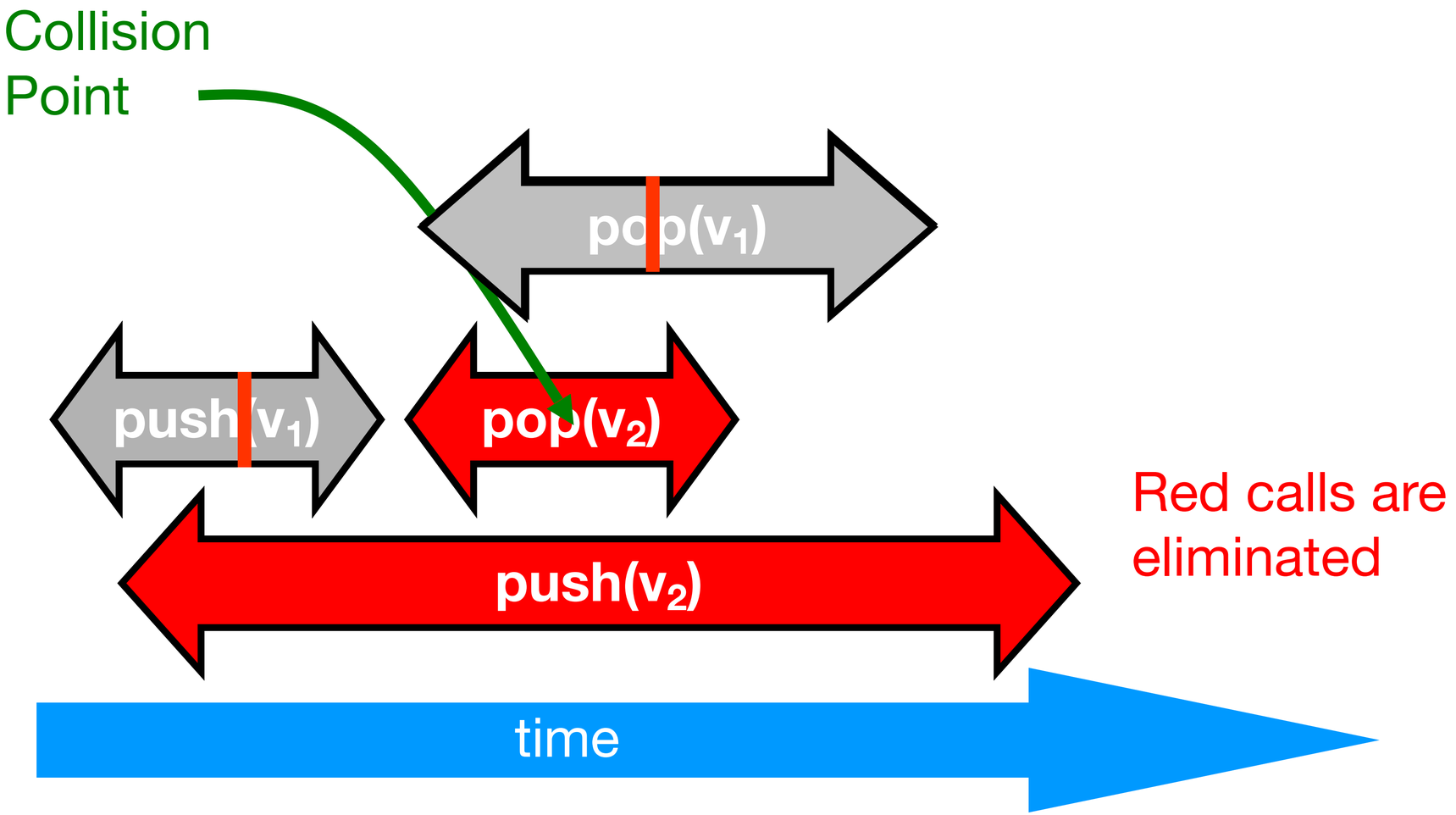
Eliminated Linearizability



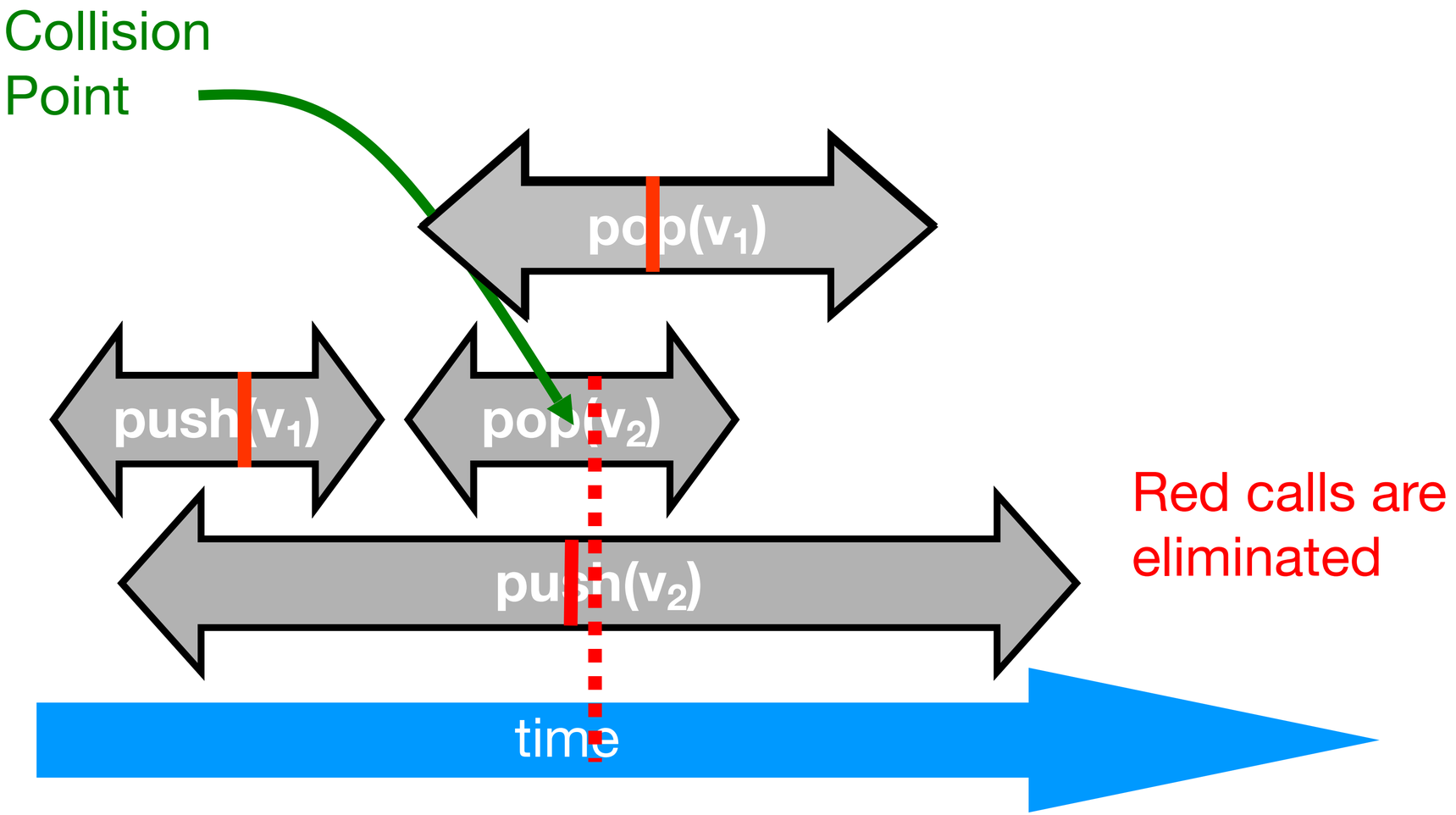
Eliminated Linearizability



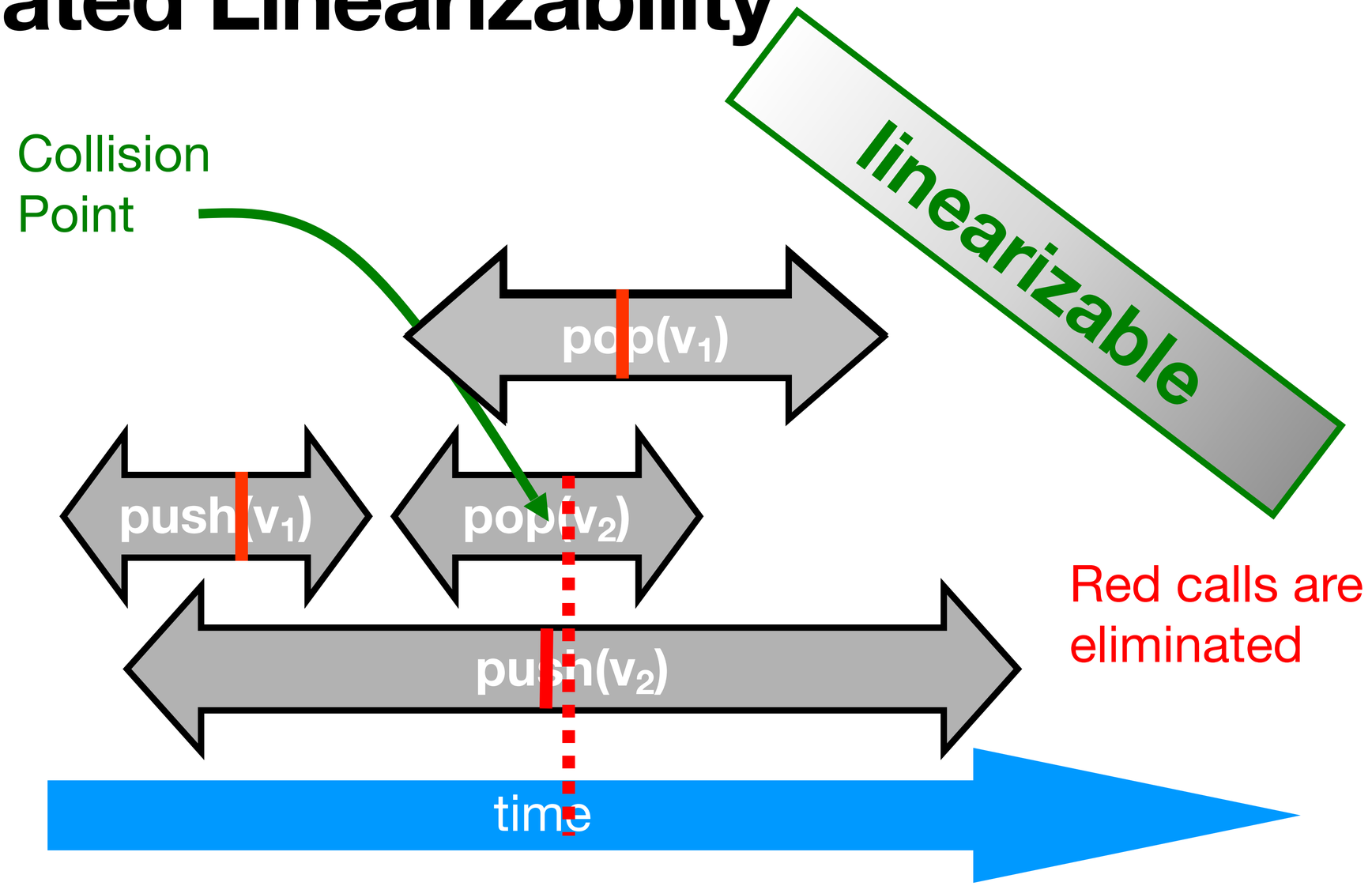
Eliminated Linearizability



Eliminated Linearizability



Eliminated Linearizability



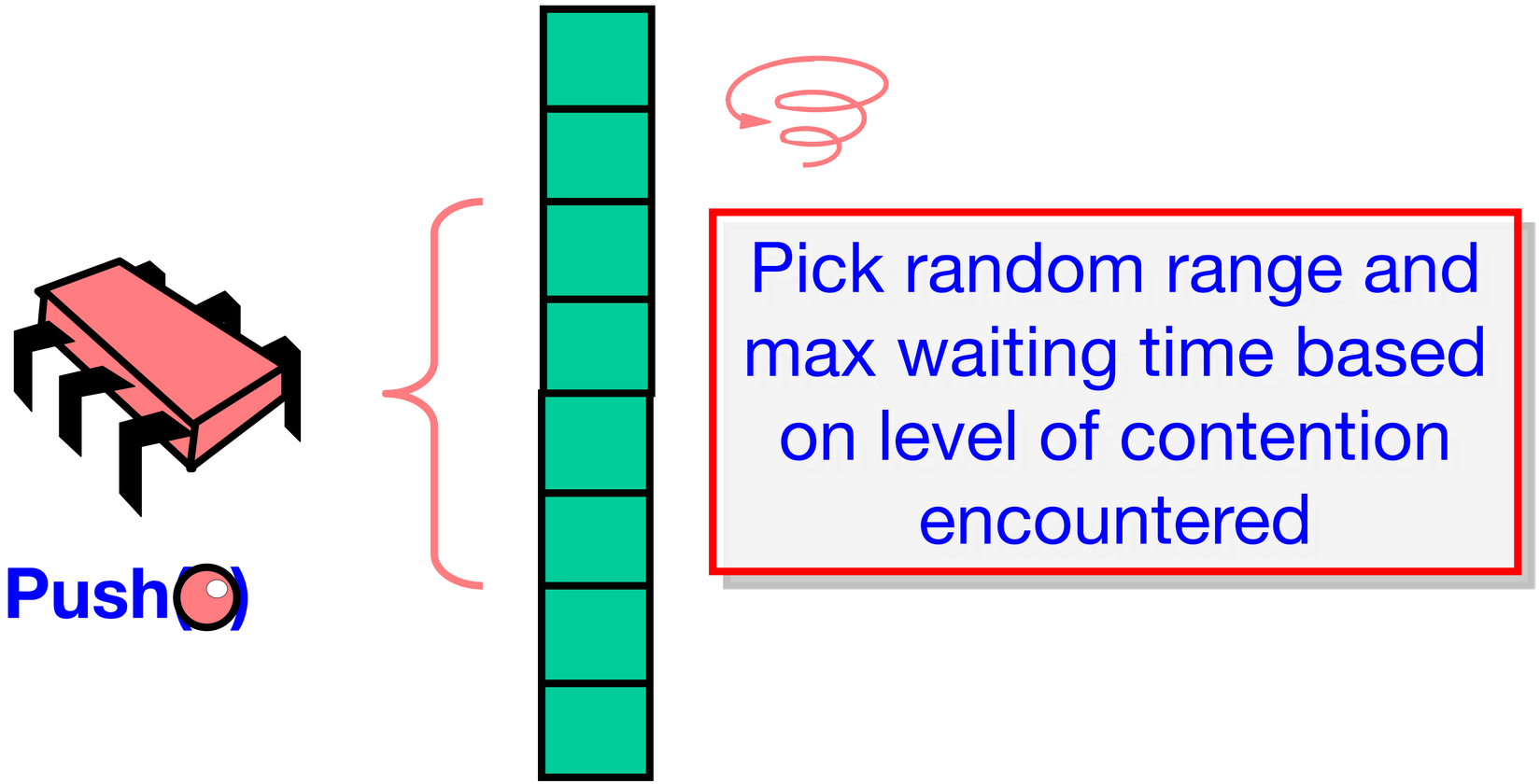
Linearizability

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

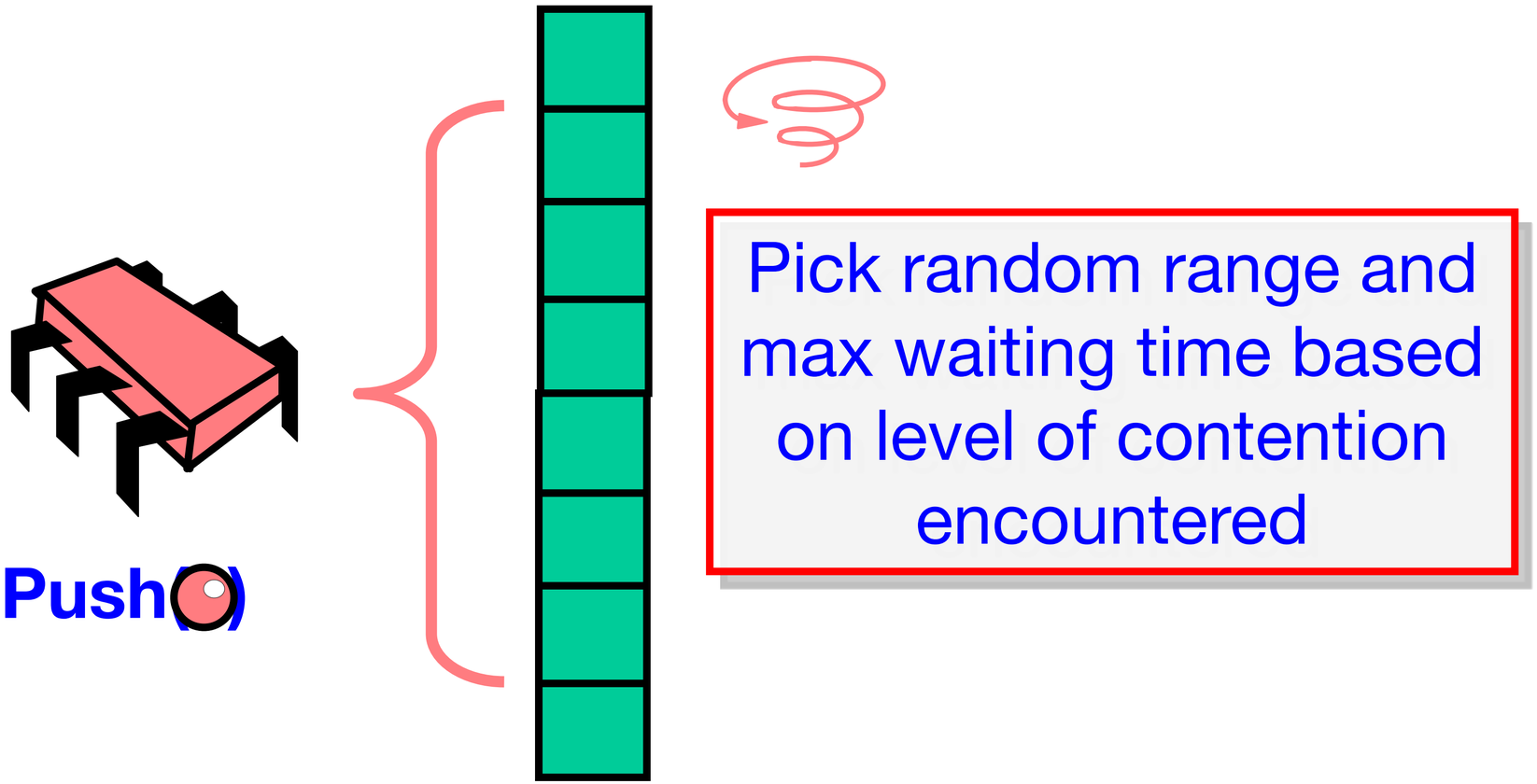
Backoff has dual effect

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

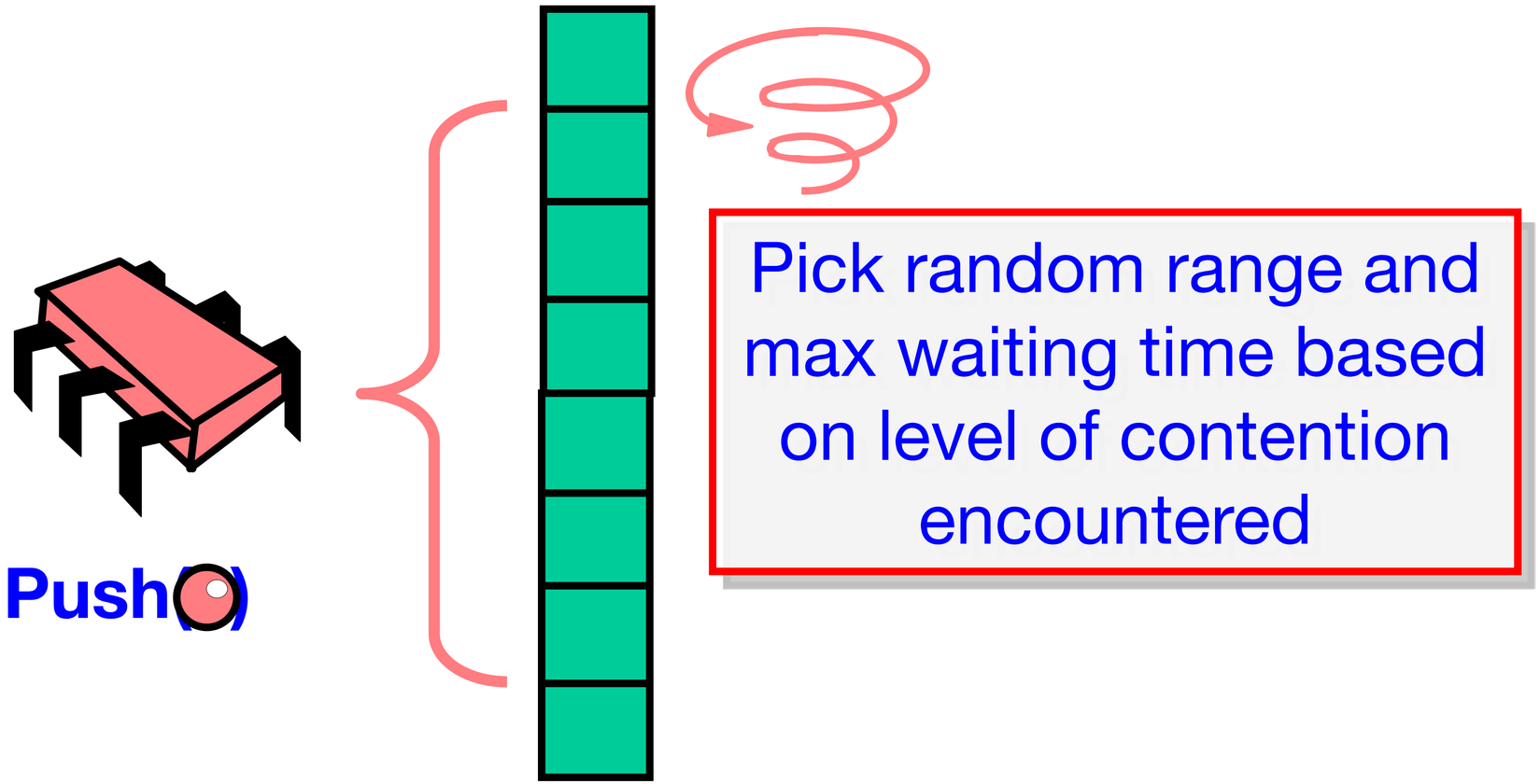
Dynamic Range and Delay



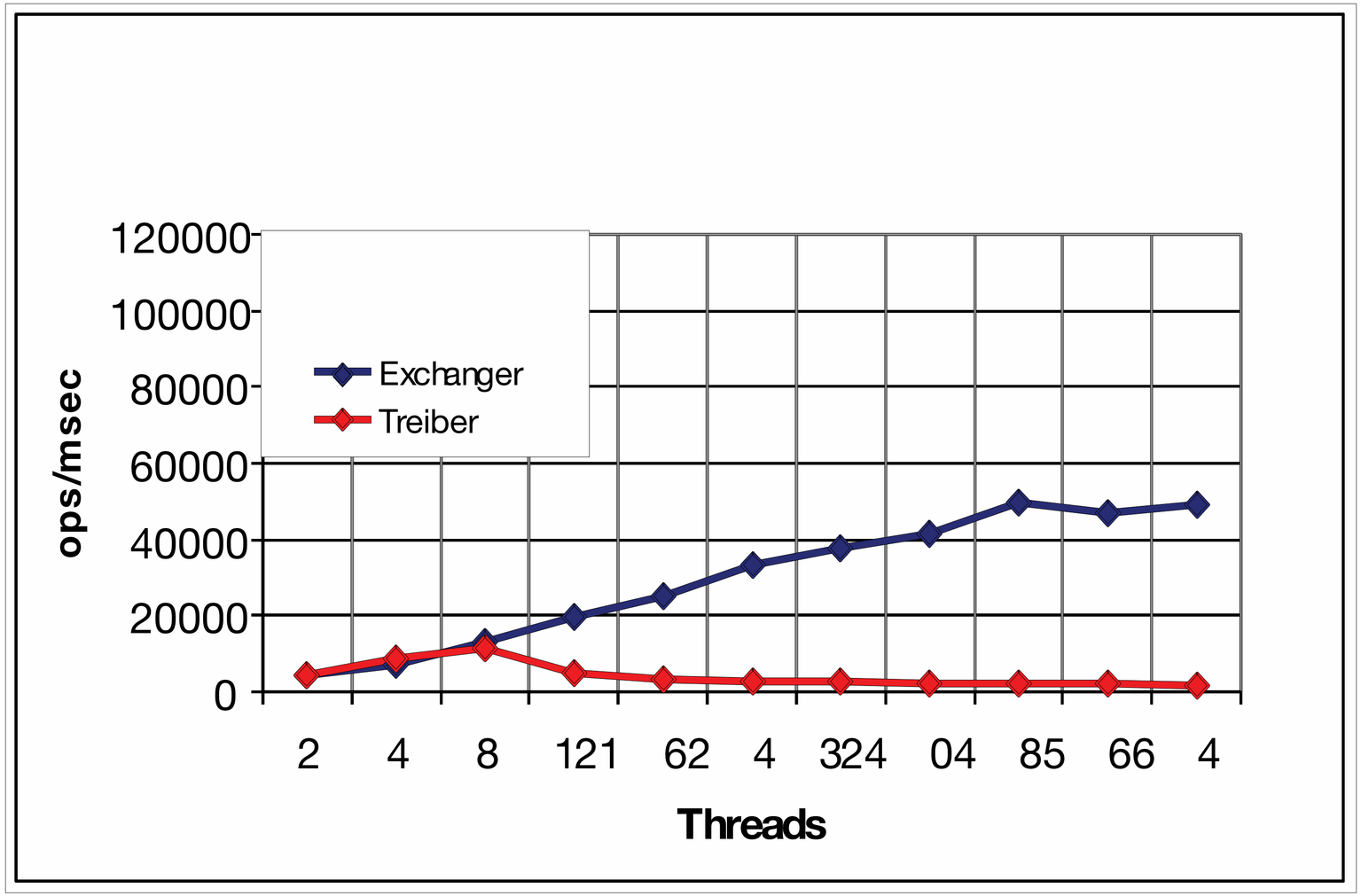
Dynamic Range and Delay



Dynamic Range and Delay

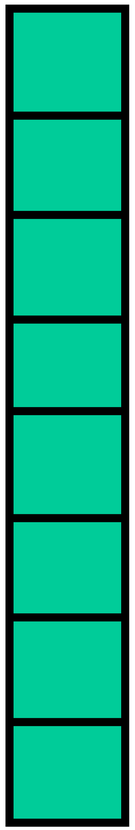


50/50, Random Slots

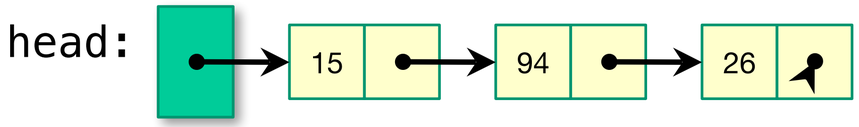


Asymmetric Rendezvous

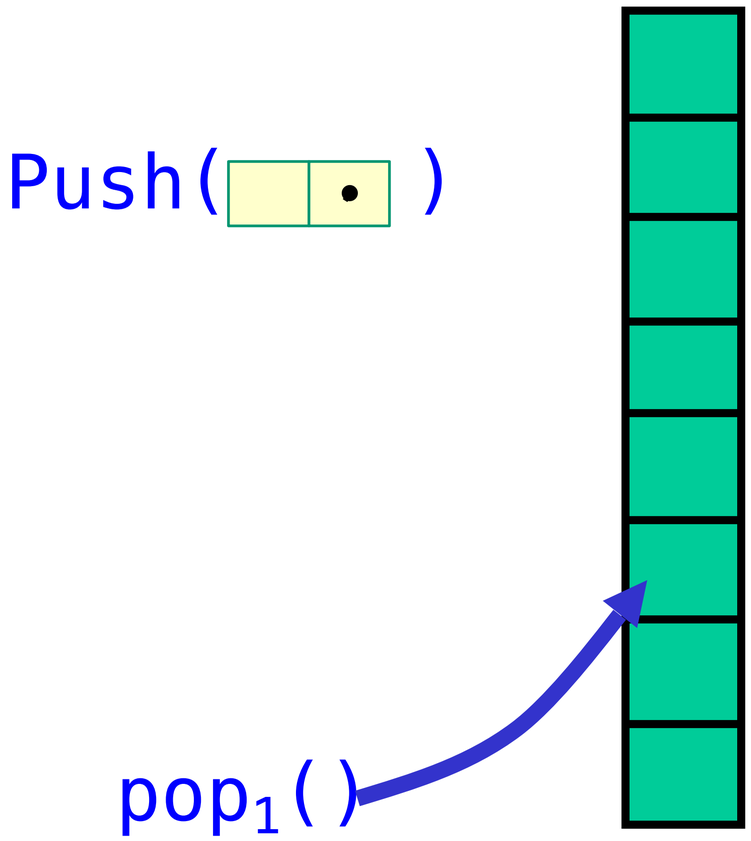
Push(•)



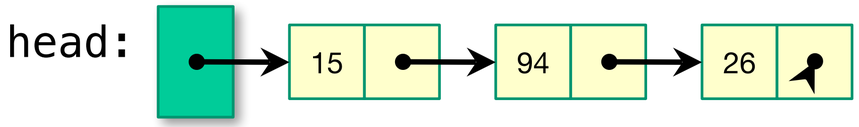
Pops find first vacant slot and spin. Pushes hunt for pops.



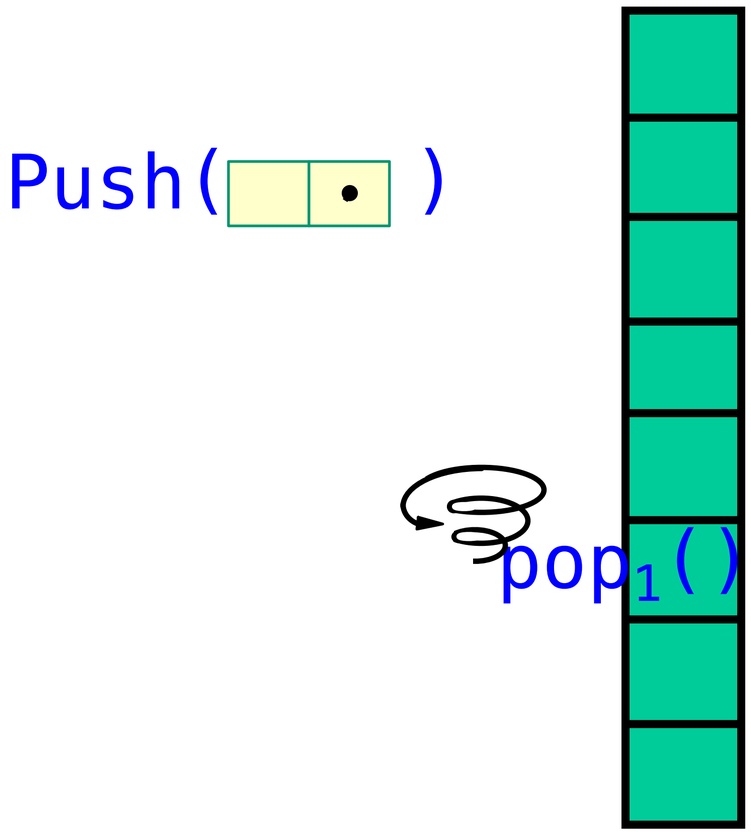
Asymmetric Rendezvous



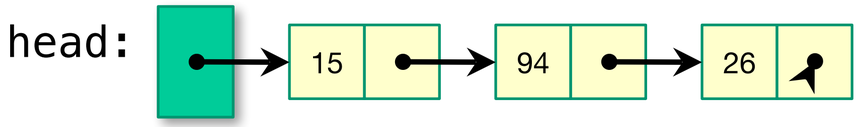
Pops find first vacant slot and spin. Pushes hunt for pops.



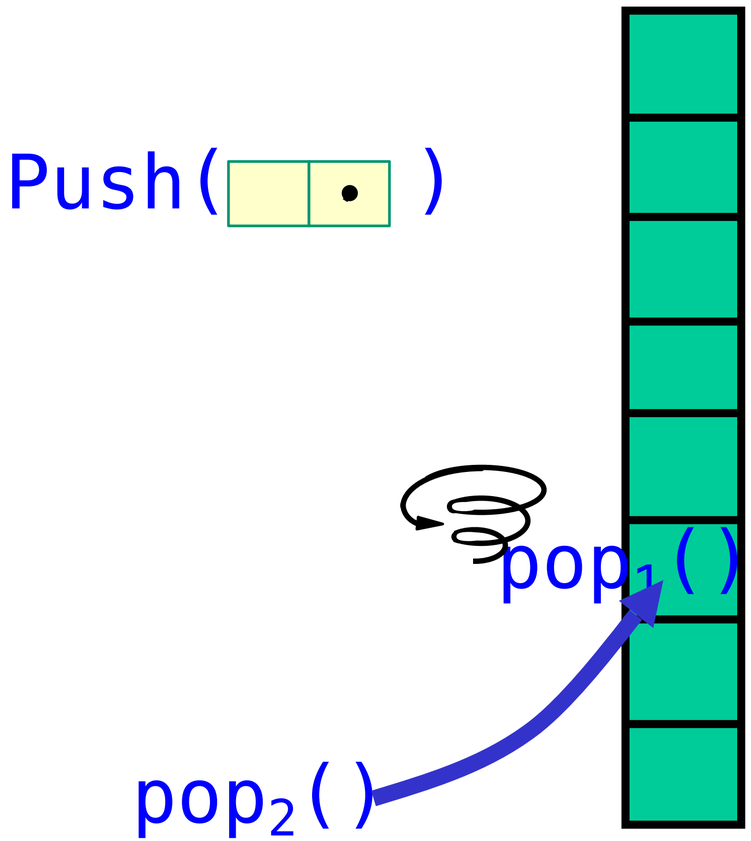
Asymmetric Rendezvous



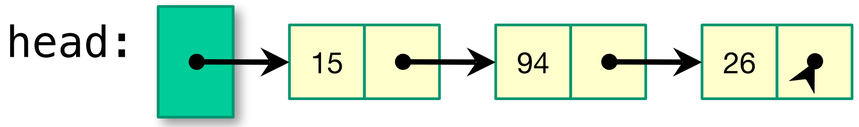
Pops find first vacant slot and spin. Pushes hunt for pops.



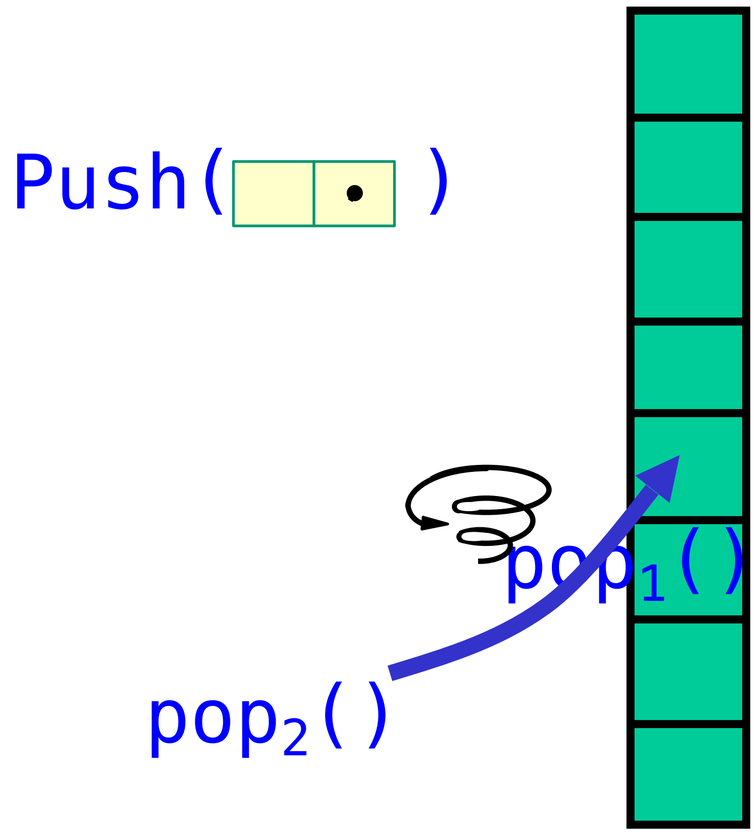
Asymmetric Rendezvous



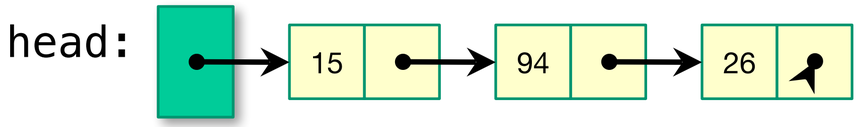
Pops find first vacant slot and spin. Pushes hunt for pops.



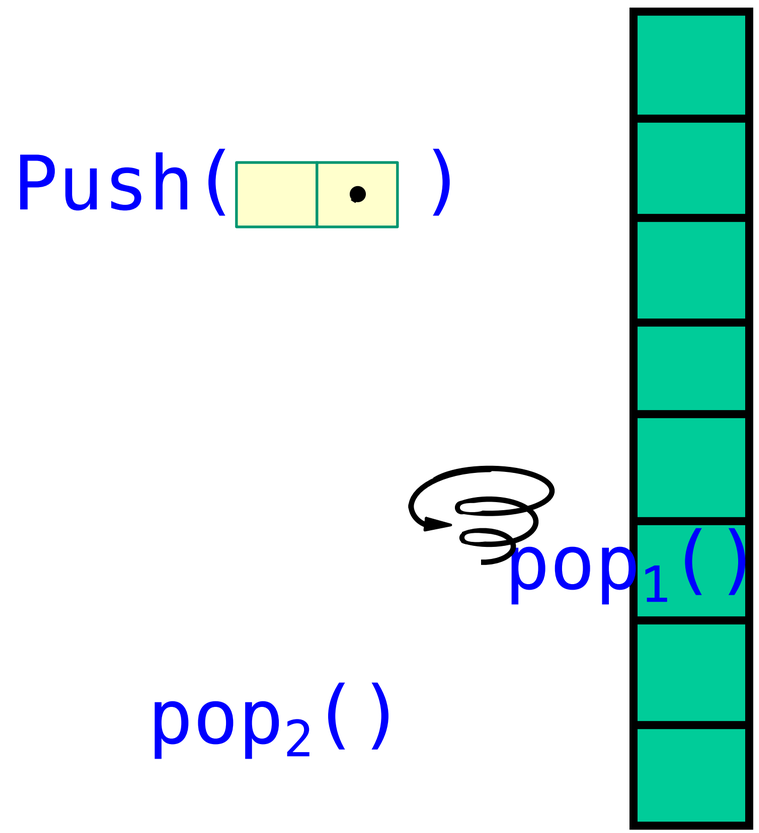
Asymmetric Rendezvous



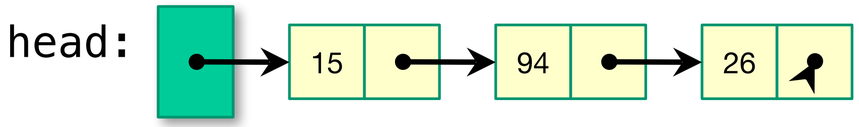
Pops find first vacant slot and spin. Pushes hunt for pops.



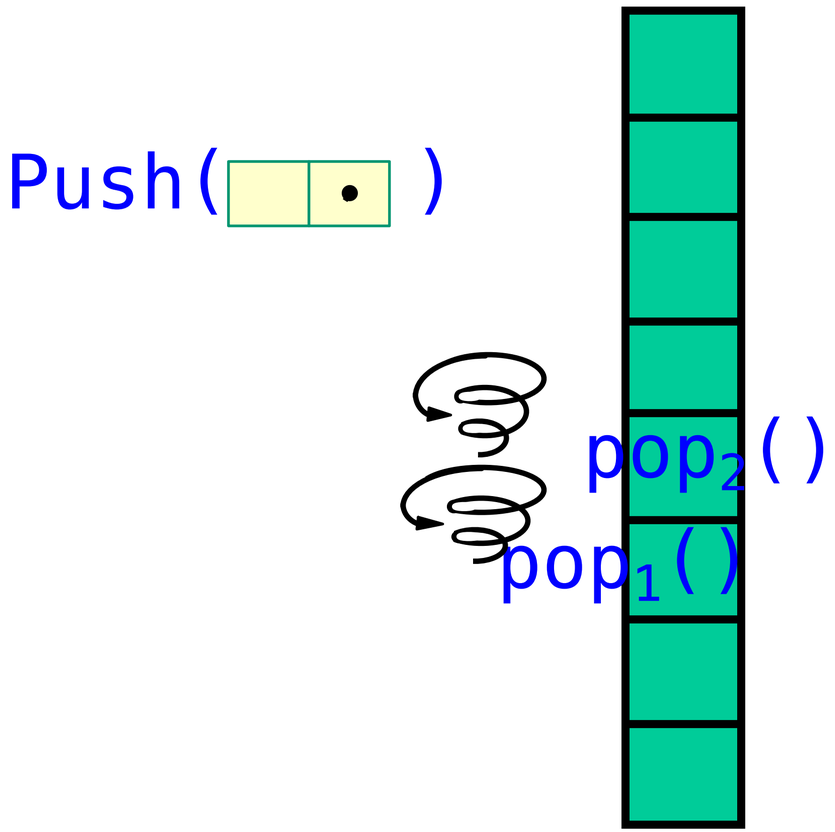
Asymmetric Rendezvous



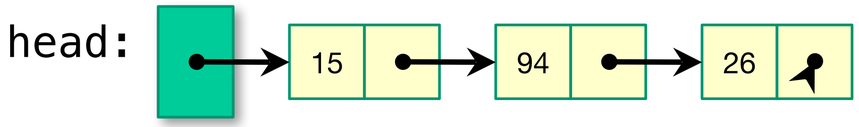
Pops find first vacant slot and spin. Pushes hunt for pops.



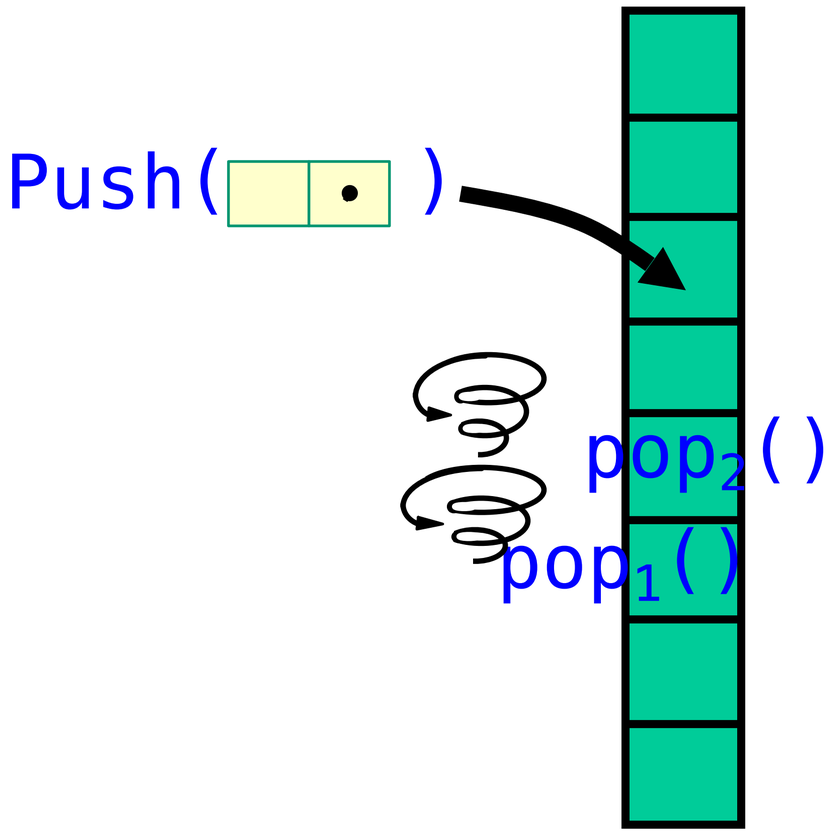
Asymmetric Rendezvous



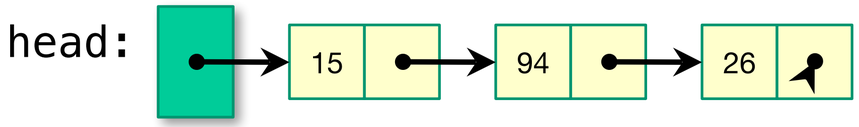
Pops find first vacant slot and spin. Pushes hunt for pops.



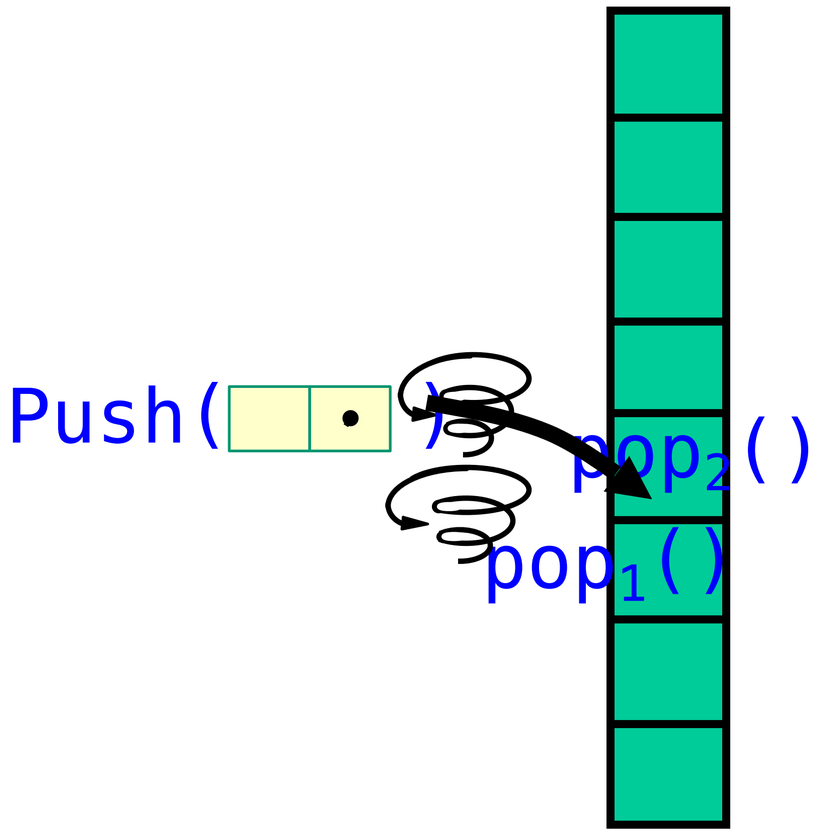
Asymmetric Rendezvous



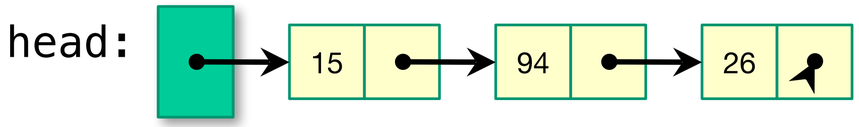
Pops find first vacant slot and spin. Pushes hunt for pops.



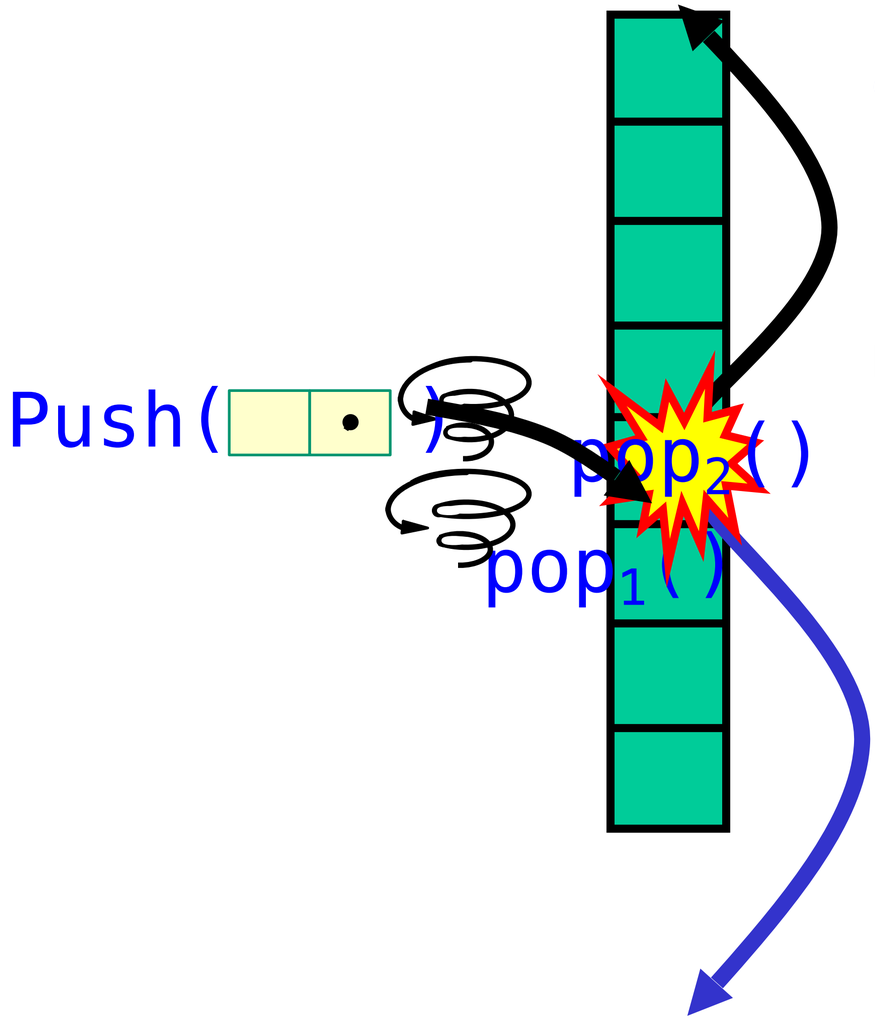
Asymmetric Rendezvous



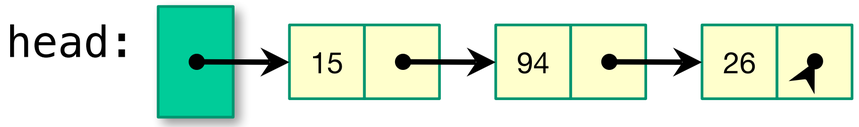
Pops find first vacant slot and spin. Pushes hunt for pops.



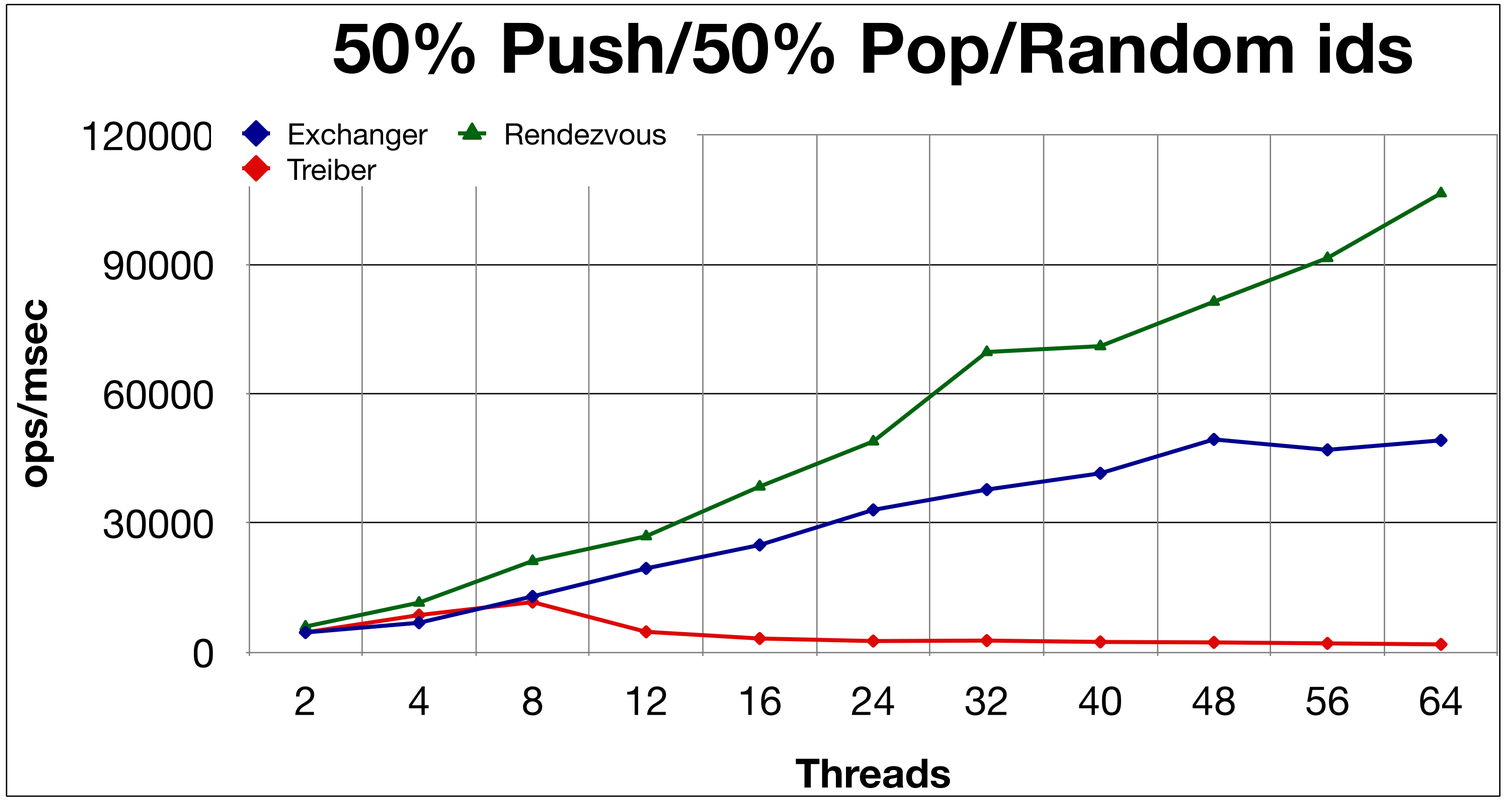
Asymmetric Rendezvous



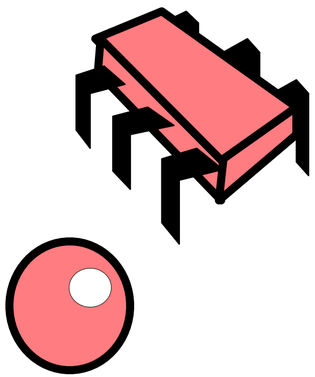
Pops find first vacant slot and spin. Pushes hunt for pops.

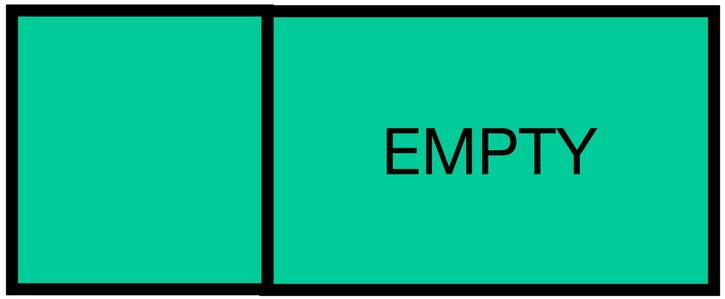


Asymmetric vs Symmetric

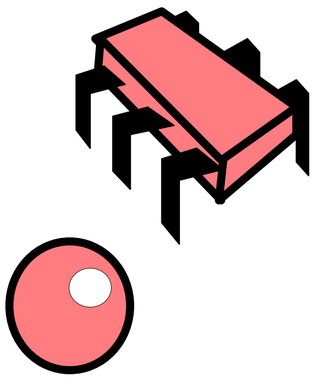


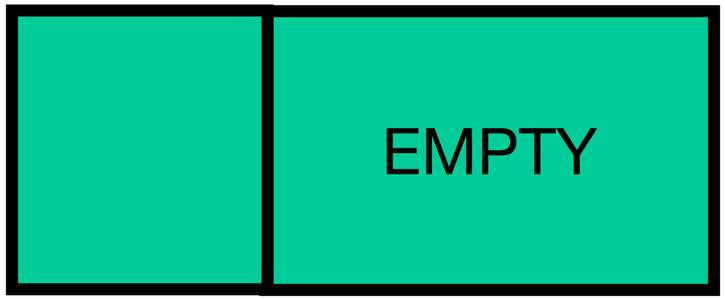
Lock-free Exchanger



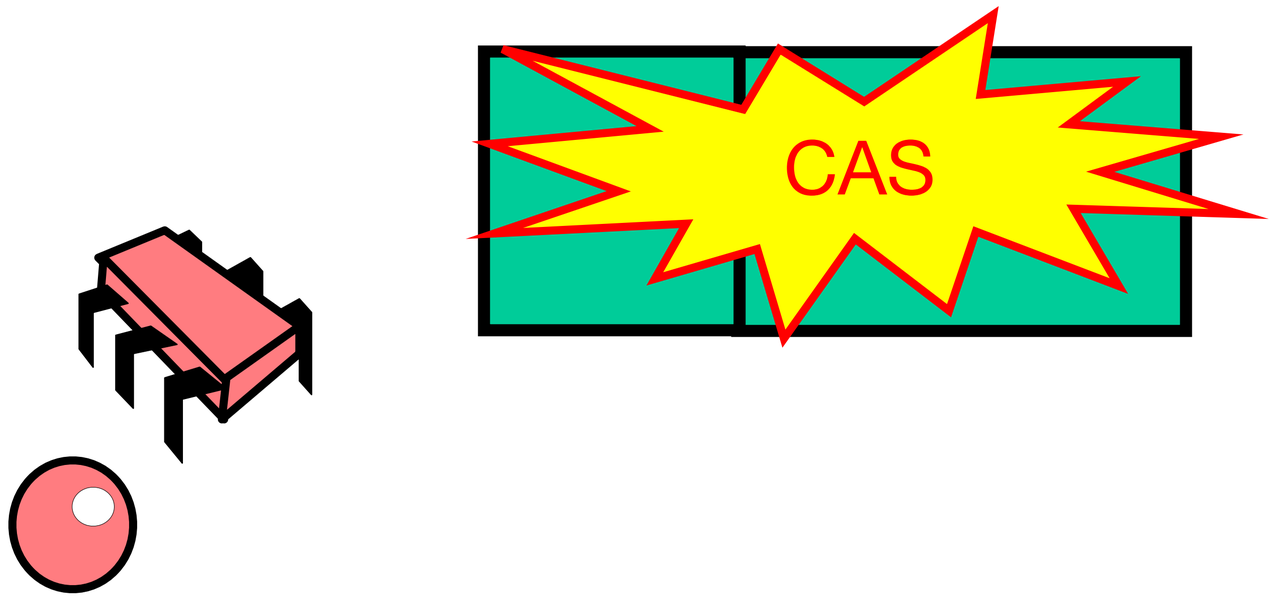


Lock-free Exchanger

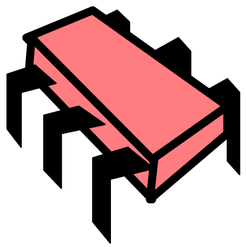


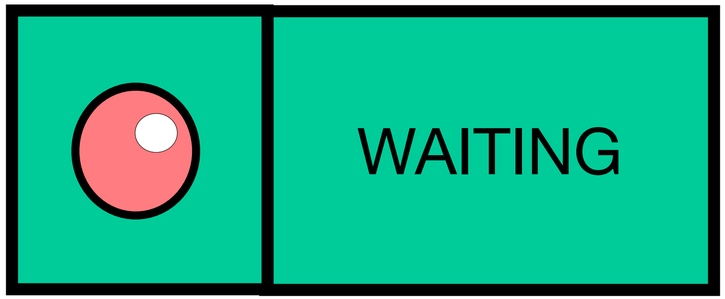


Lock-free Exchanger

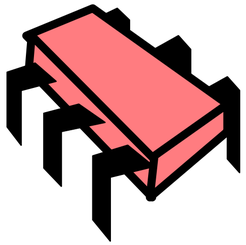


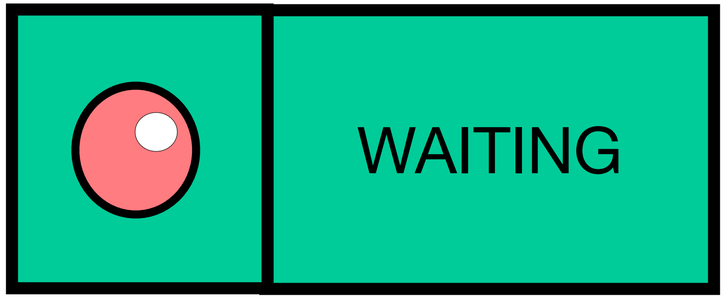
Lock-free Exchanger



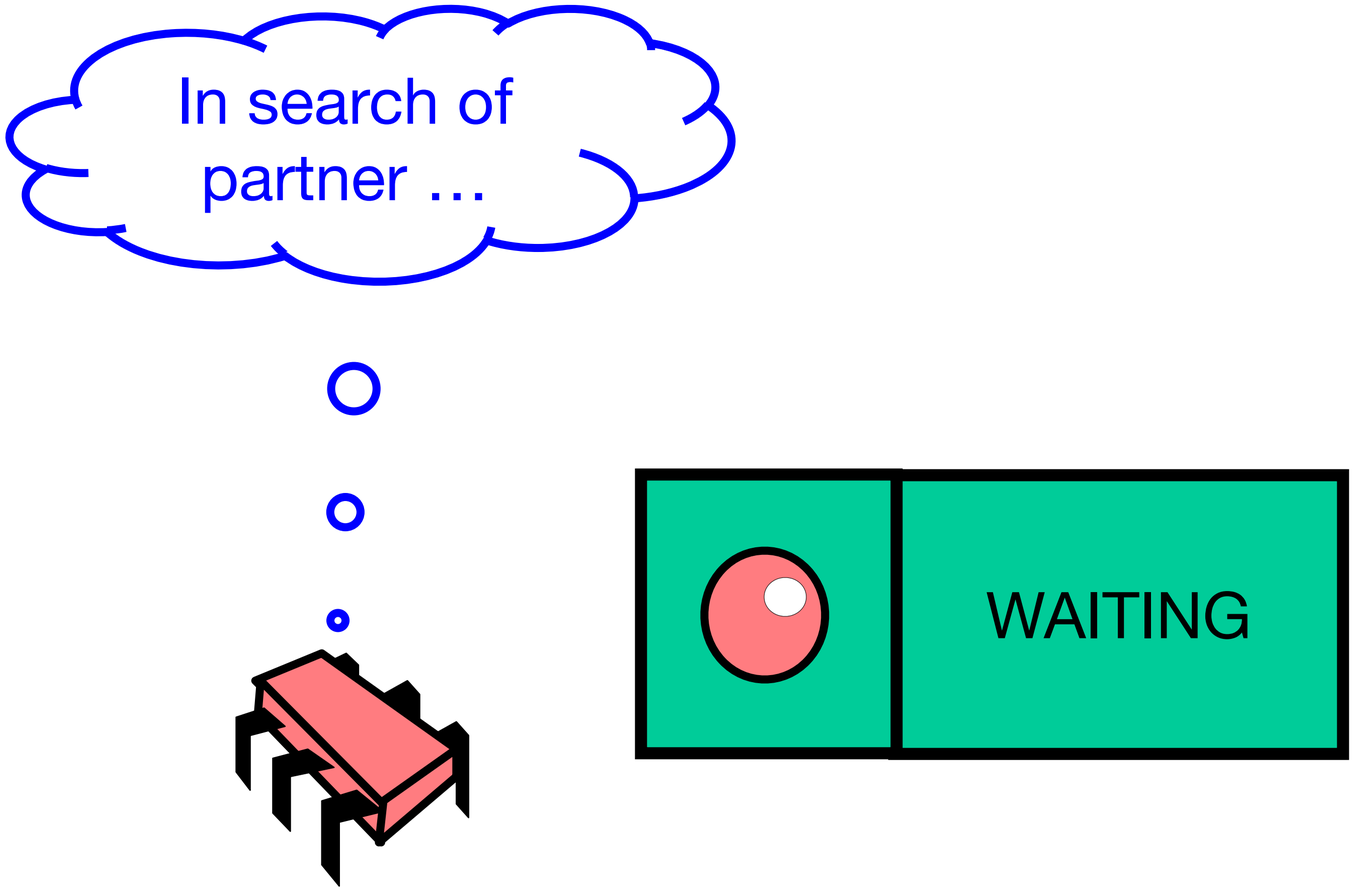


Lock-free Exchanger

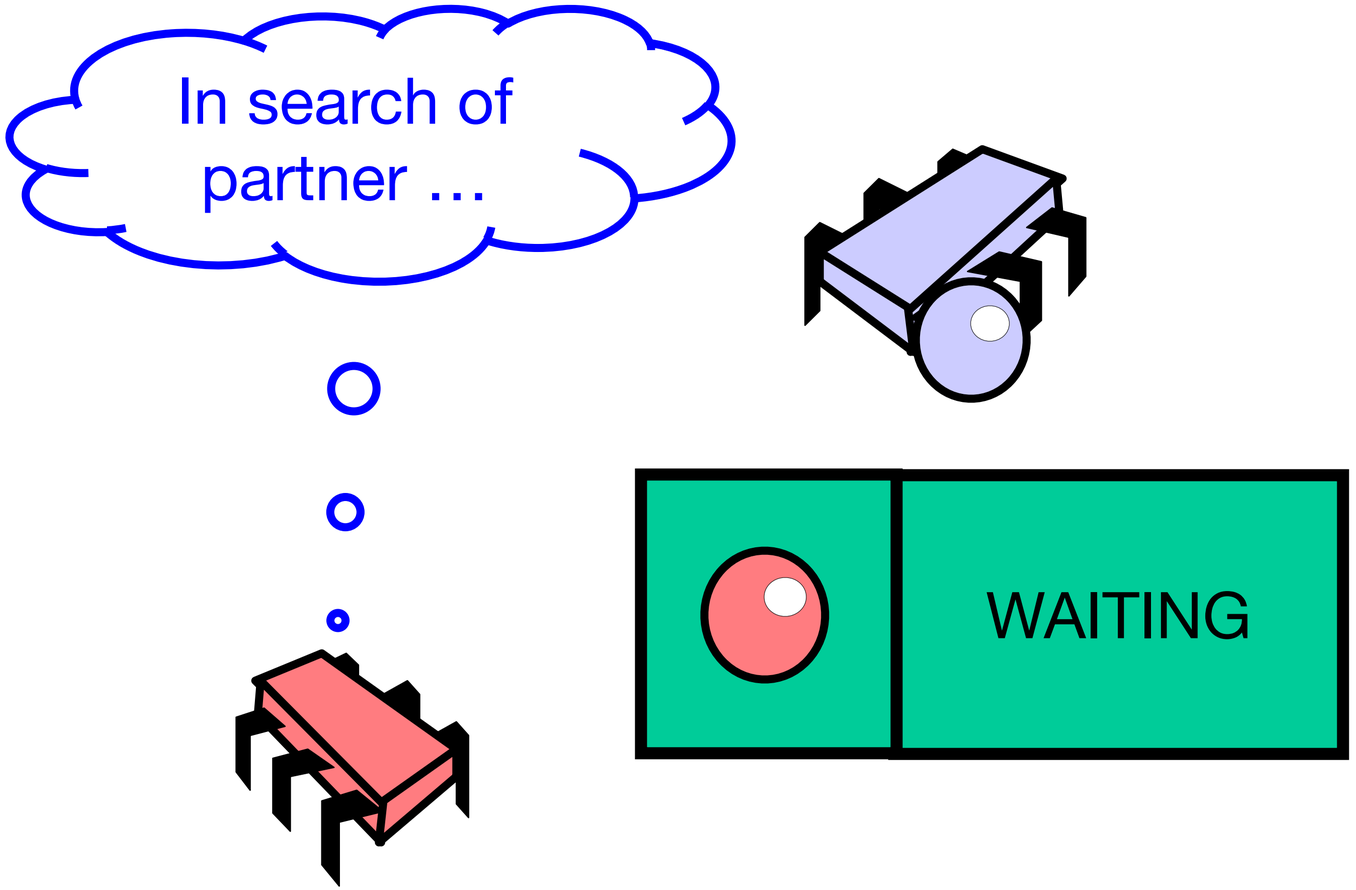




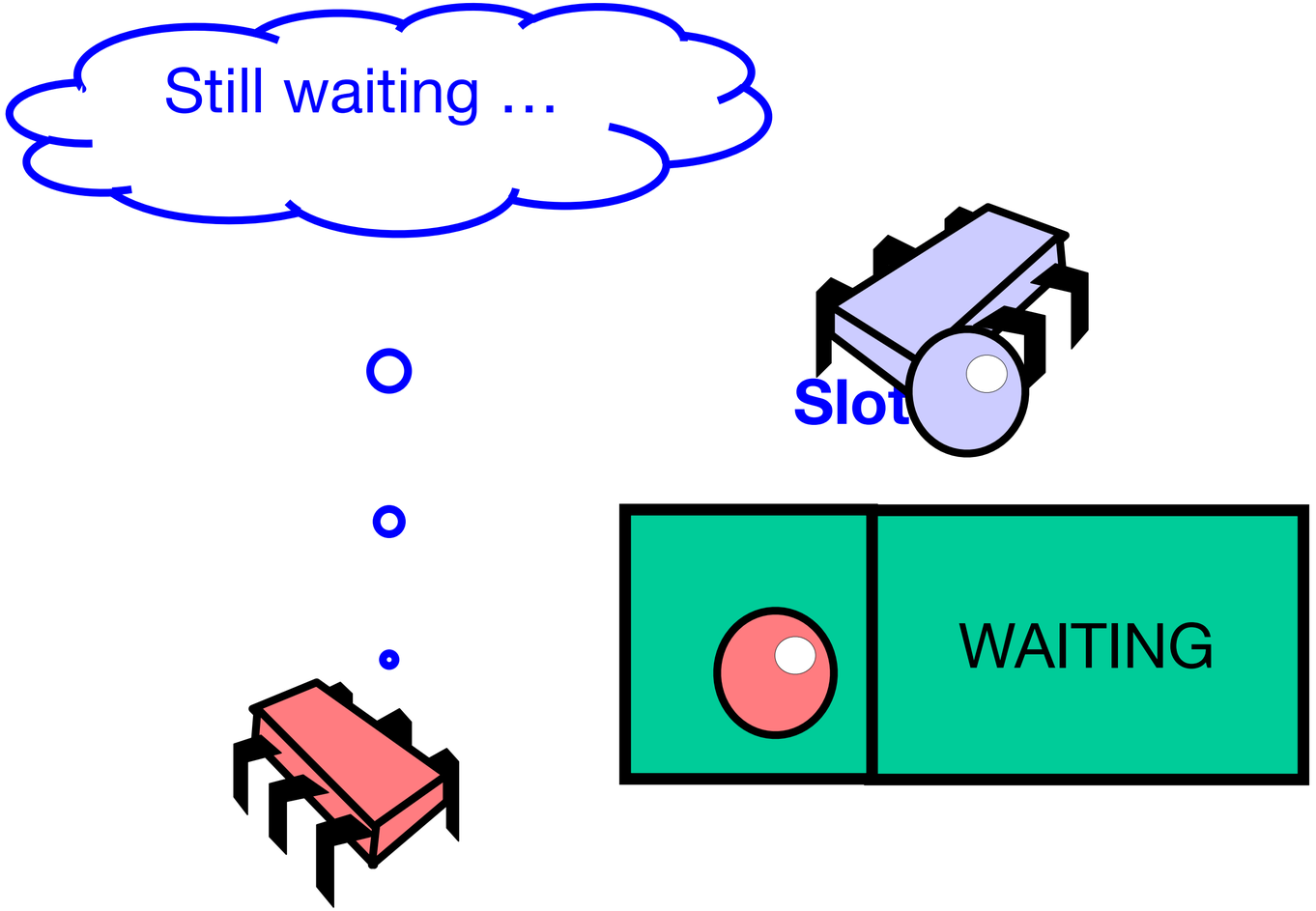
Lock-free Exchanger



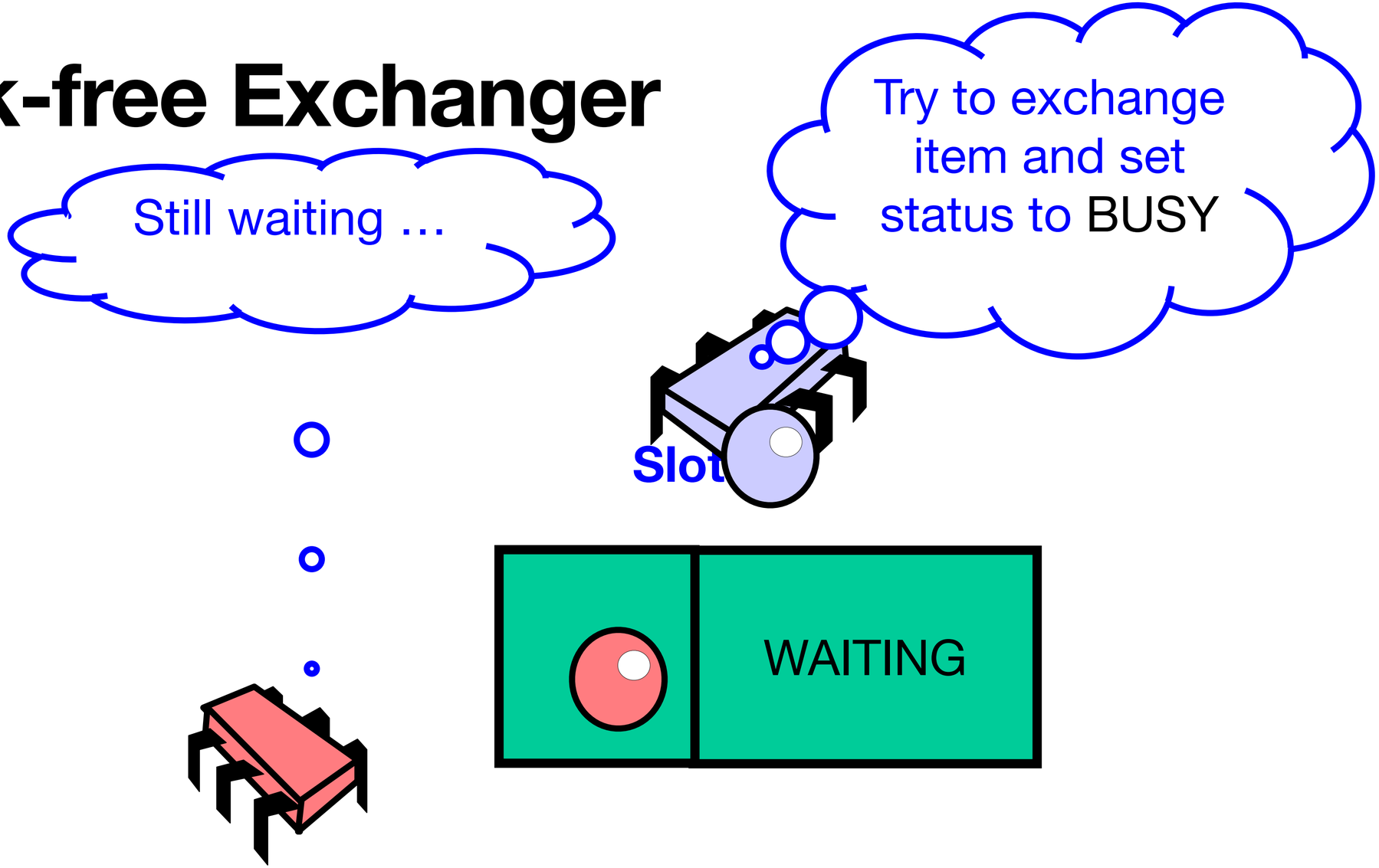
Lock-free Exchanger



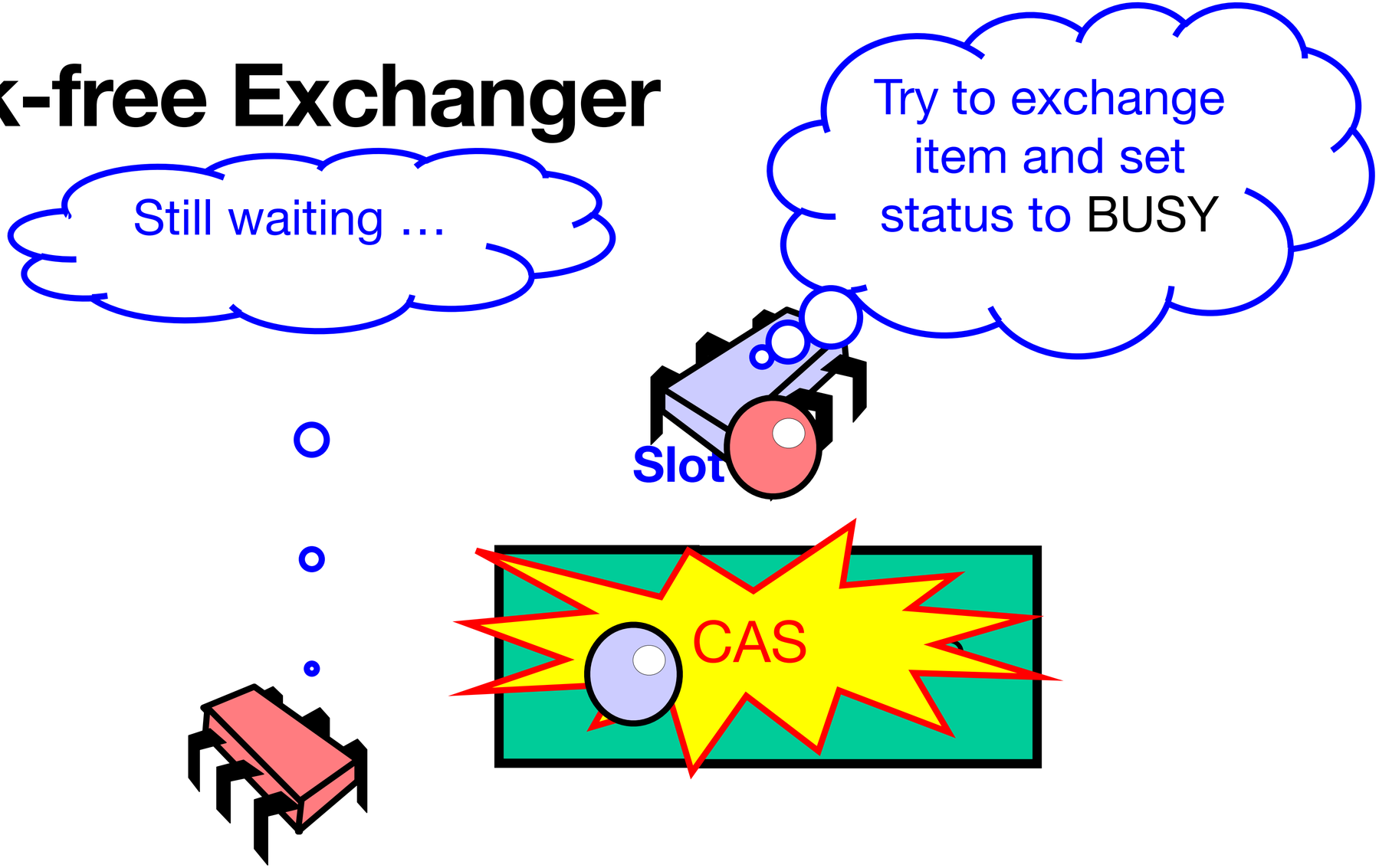
Lock-free Exchanger



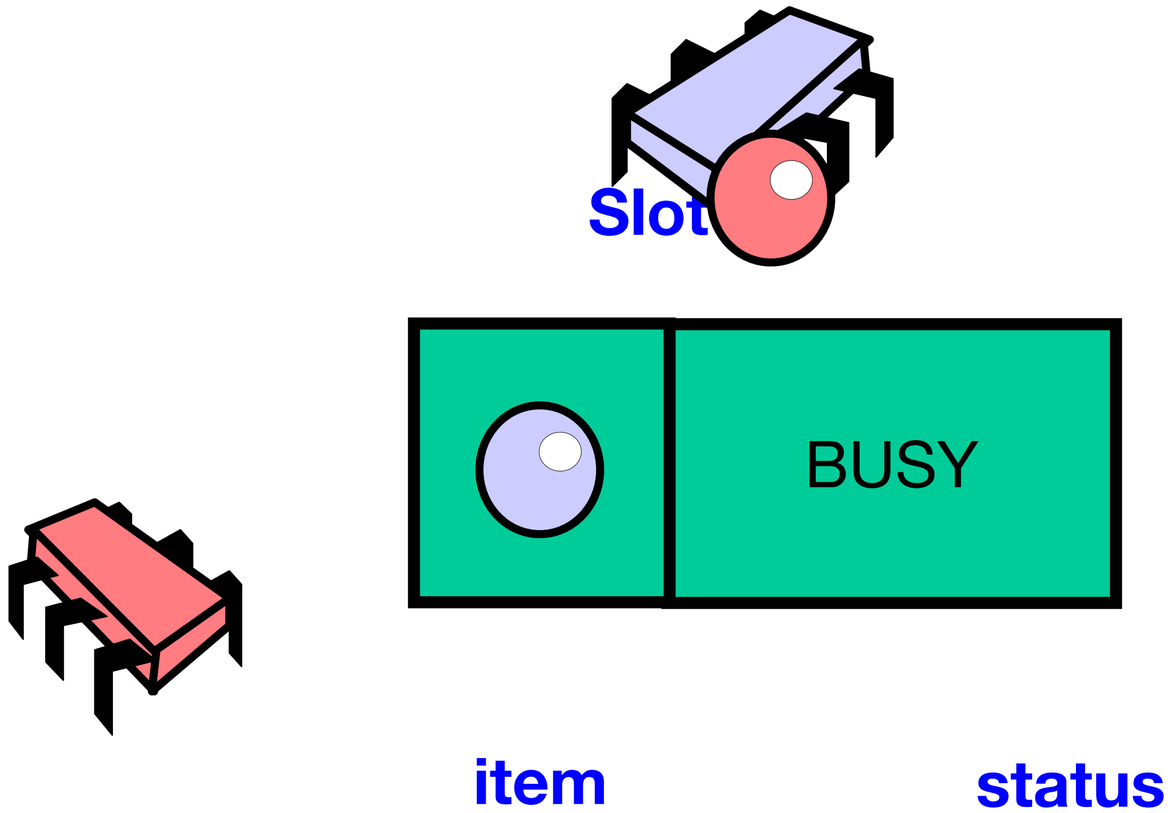
Lock-free Exchanger



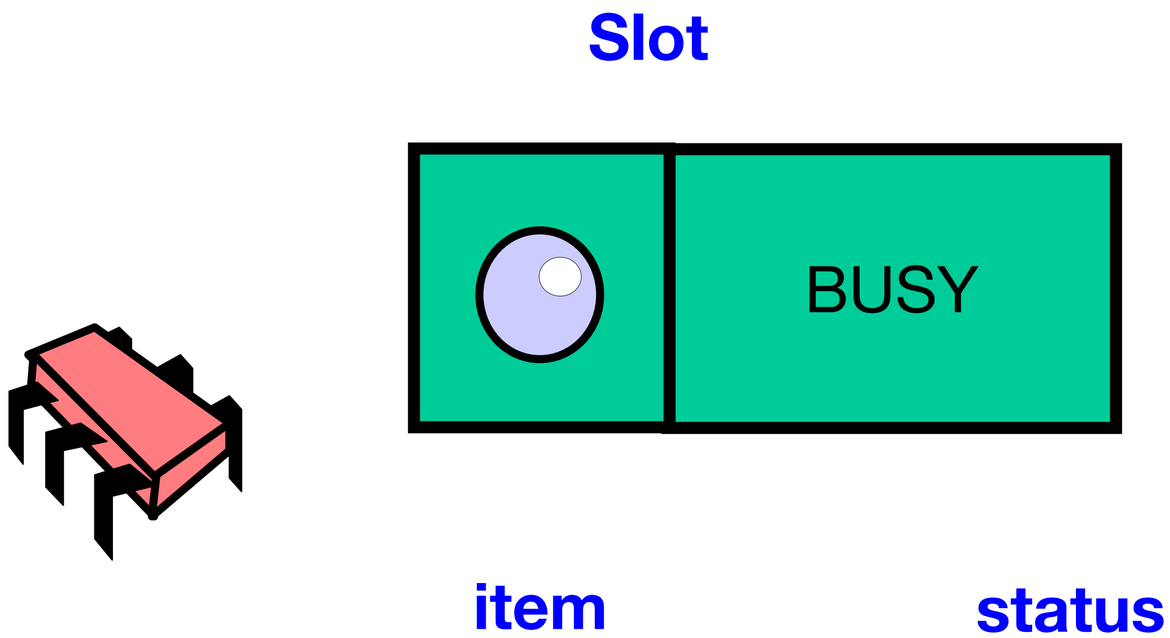
Lock-free Exchanger



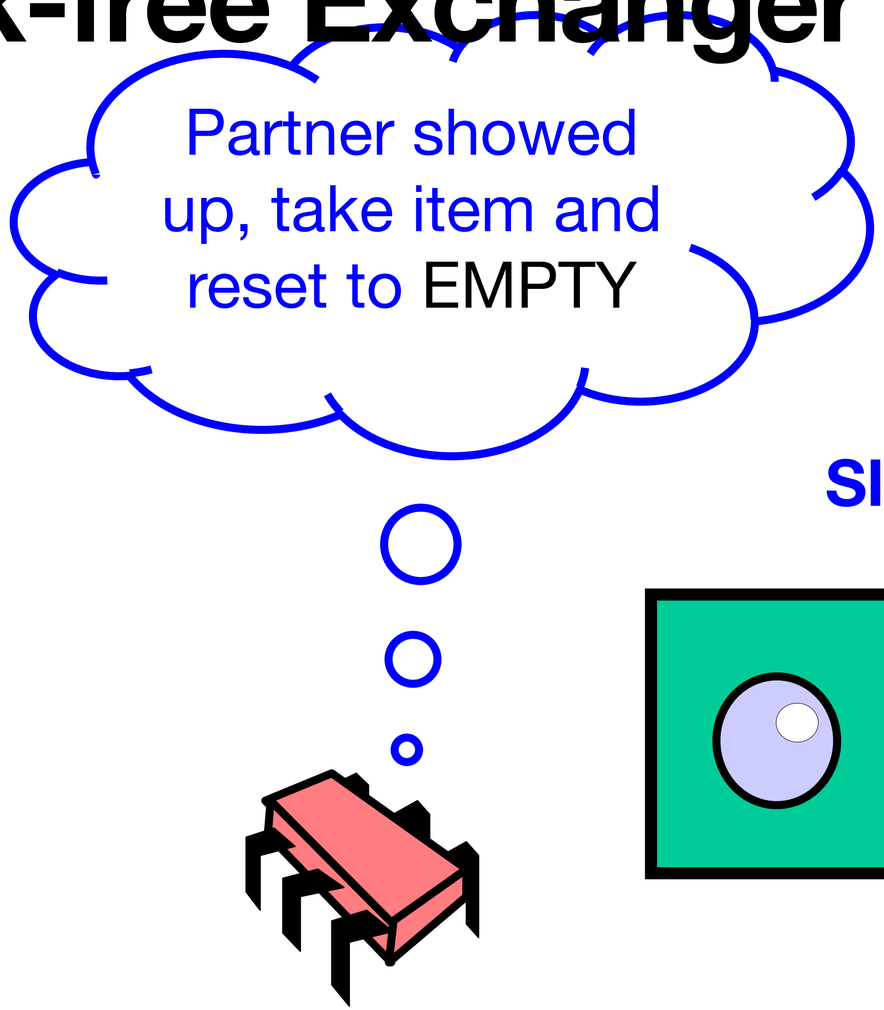
Lock-free Exchanger



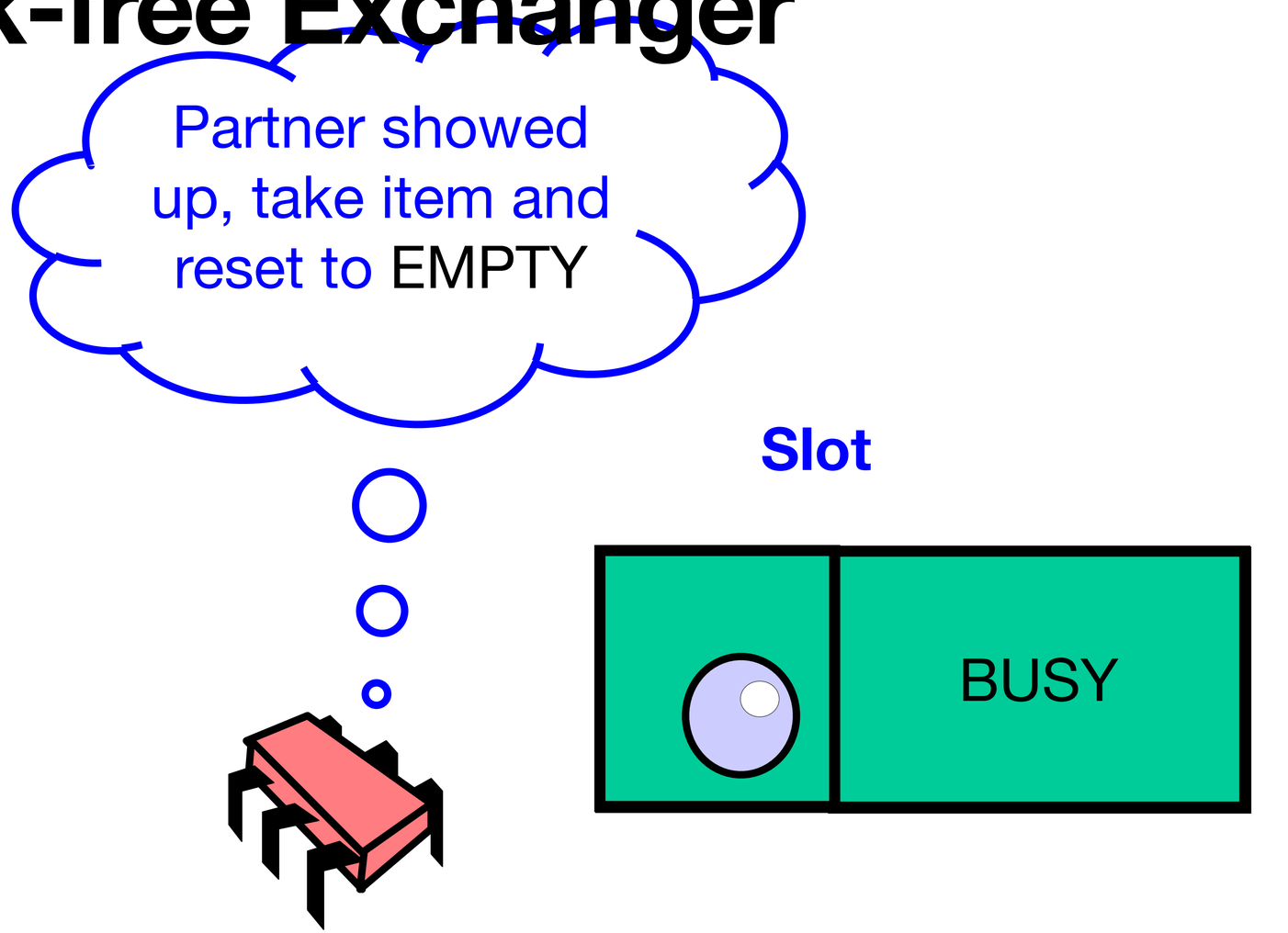
Lock-free Exchanger



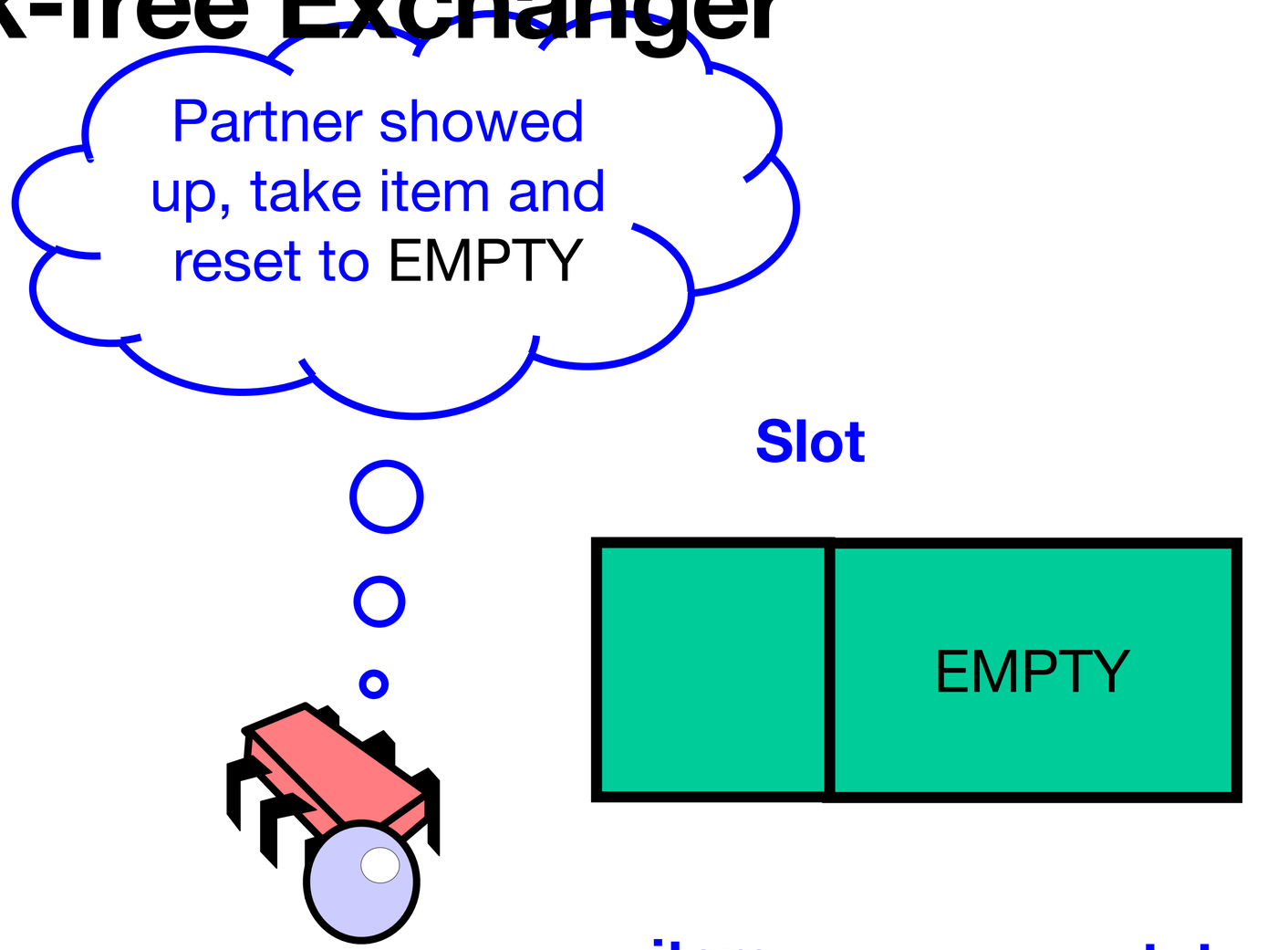
Lock-free Exchanger



Lock-free Exchanger



Lock-free Exchanger



Exchanger

- Exchanger is lock-free
- Because the only way an exchange can fail is if others repeatedly succeed or no one shows up
- The slot we need does not require symmetric exchange
  - Only **push** needs to pass an item, not **pop**

Walk through the code

lockfree_exchanger.ml

elimination_array.ml

elimination_backff_stack.ml

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