

Concepts of Concurrent Computation

Spring 2015

Lecture 7: Lock-Free Approaches

Sebastian Nanz
Chris Poskitt

Today's agenda

1. what's wrong with locks?
2. lock-free algorithms and data structures
3. transactional memory

What's wrong with locks?



They are difficult to use correctly

- forget to take a lock?
- take too many locks?
- take the locks in the wrong order?
- take the wrong lock?

They are difficult to use correctly

- forget to take a lock?

danger of data race

- take too many locks?

danger of deadlock

- take the locks in the wrong order?

danger of deadlock

- take the wrong lock?

???

Blocking, faults, and performance...

- priority inversion

=> lower-priority thread preempted while holding a lock that a higher-priority thread needs

- convoying

=> multiple threads of the same priority contend repeatedly for the same lock

- fault tolerance

=> what if a faulty process halts whilst holding a lock?

- granularity of locking

*=> **lock overhead** vs. **lock contention***

Blocking, faults, and performance...

- priority inversion

=> lower-priority thread preempted while holding a lock that a higher-priority thread needs

- convoying

=> multiple threads of the same priority contend repeatedly for the same lock

- fault tolerance

=> what if a faulty process halts whilst holding a lock?

- granularity of locking

decreases with more locks

*=> **lock overhead** vs. **lock contention***

increases with more locks

Locks are not “composable” in general

- they don't support modular programming

=> i.e. building larger programs from smaller blocks

```
class Account {  
    int balance;  
    synchronized void deposit(int amount) {  
        balance = balance + amount;  
    }  
    synchronized void withdraw(int amount) {  
        balance = balance - amount;  
    }  
}
```

*how to implement
a “transfer” method?*

Locks are not “composable” in general

- although **deposit** and **withdraw** are correctly implemented **by themselves**, the following is incorrect:



```
void transfer(Account acc1, Account acc2, int amount) {  
    acc1.withdraw(amount);  
    acc2.deposit(amount);  
}
```

Locks are not “composable” in general

- although **deposit** and **withdraw** are correctly implemented **by themselves**, the following is incorrect:



```
void transfer(Account acc1, Account acc2, int amount) {  
    acc1.withdraw(amount);  
    acc2.deposit(amount);  
}
```

*have to add explicit
locking code*

```
void transfer(Account acc1, Account acc2, int amount) {  
    synchronized (acc1) {  
        synchronized (acc2) {  
            acc1.withdraw(amount);  
            acc2.deposit(amount);  
        }  
    }  
}
```

How do we do concurrent programming without locks?

- message passing

=> no shared data at all

=> *but: overheads of messaging, slower access to data, ...*


- lock-free programming

=> *instead of locks, use stronger atomic operations*

- software transactional memory (STM)

=> *based on the idea of database transactions*

Next on the agenda

1. what's wrong with locks? 
2. lock-free algorithms and data structures
3. transactional memory

Lock-free programming

- write shared-memory concurrent programs **without using locks** (but still ensuring **thread safety**)
- idea: use **stronger atomic operations** (typically provided by the hardware)
- designing general lock-free algorithms is difficult
 - => *focus instead on developing lock-free data structures*
 - => *stack, list, queue, buffer, ...*
 - => *NB: avoids many of the problems of locks, but not the problem of compositionality*

Classes of lock-free algorithms

- typically distinguish two **classes** of lock-free algorithms

lock-free

wait-free

Classes of lock-free algorithms

- typically distinguish two **classes** of lock-free algorithms

lock-free

\Rightarrow guaranteed
system-wide progress

\Rightarrow i.e. infinitely often
some process finishes

wait-free

\Rightarrow guaranteed
per-thread progress

\Rightarrow i.e. all processes
complete in a finite
number of steps

implies

free from deadlock

**free from deadlock
and starvation**

Compare-and-swap (CAS)

- compare-and-swap (CAS) combines a **load** and a **store** into a **single atomic operation**
- takes three arguments: a memory address **x**, an **old** value, and a **new** value

CAS (**x**, **old**, **new**)

- atomically reads the contents at **x**, and, if it contains **old**, updates it to **new**

Compare-and-swap (CAS)

- CAS must **indicate** whether or not it performed the substitution

=> by returning the value read from memory

=> or by a simple Boolean response

- latter variant sometimes called **compare-and-set**:

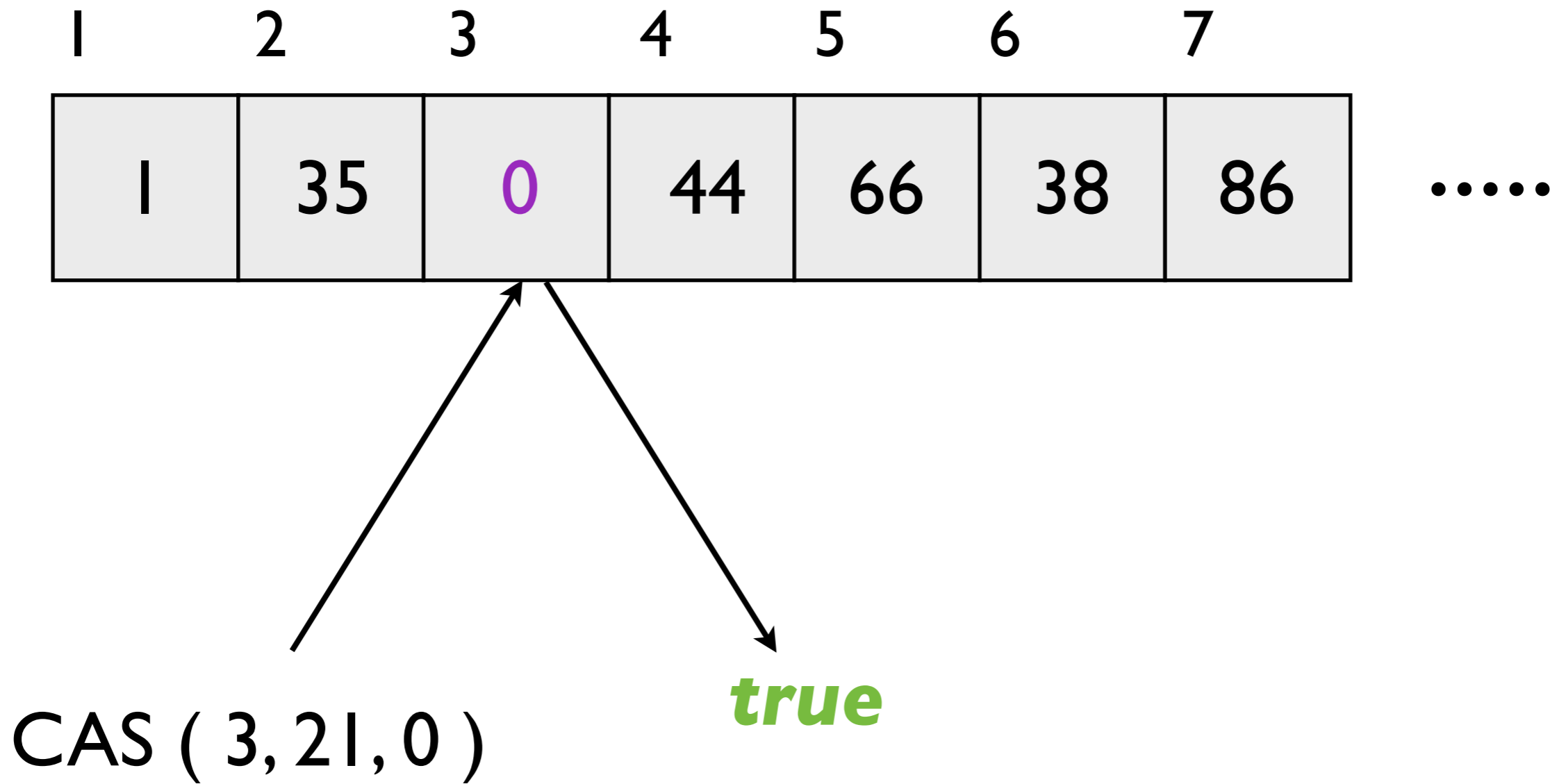
```
CAS (x, old, new)
  do-atomic
    if *x = old then
      *x := new;
      result := true
    else
      result := false
    end
  end
end
```

Compare-and-swap (CAS)



CAS (3, 21, 0)

Compare-and-swap (CAS)

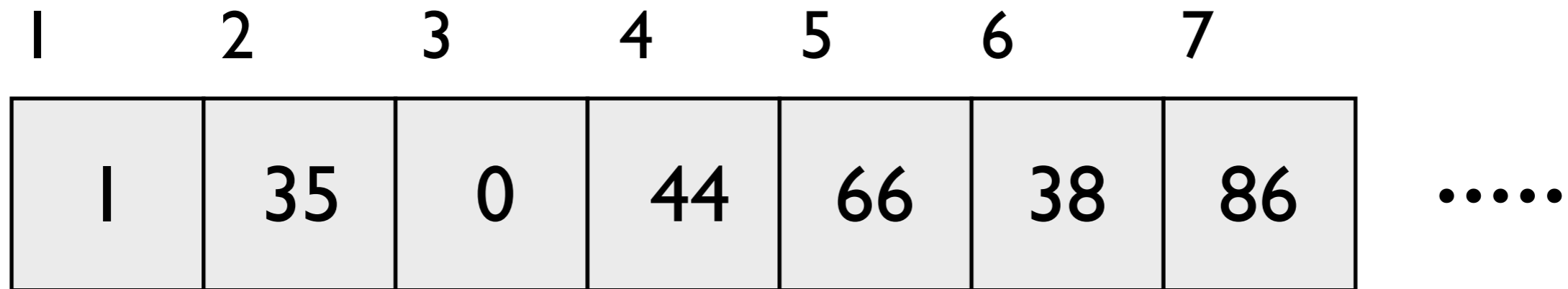


Compare-and-swap (CAS)



CAS (5, 21, 0)

Compare-and-swap (CAS)



CAS (5, 21, 0)

false

Using CAS to implement lock-free algorithms

- CAS can be used to implement **lock-free concurrent data structures**

=> we will look at a concurrent stack and queue

- typically, a thread interacting with a data structure will loop repeatedly until the CAS operation succeeds

=> not as easy to do correctly as it sounds! :-)

Treiber stack

(a simple lock-free stack)

- CAS facilitates a **lock-free stack implementation** (due to Treiber, 1986)
- **stack of integers** represented as a **linked list of nodes**; the top of the stack is given by **head**

```
class Node {  
    Node* next;  
    int item;  
}  
  
Node* head; // top of the stack
```

Treiber stack

(a simple lock-free stack)

- to implement *push* and *pop*, a common pattern is used:
 - (1) **read a value** from the current state
 - (2) compute an **updated value** based on the read one
 - (3) **atomically update** the state by swapping the new for old

Push

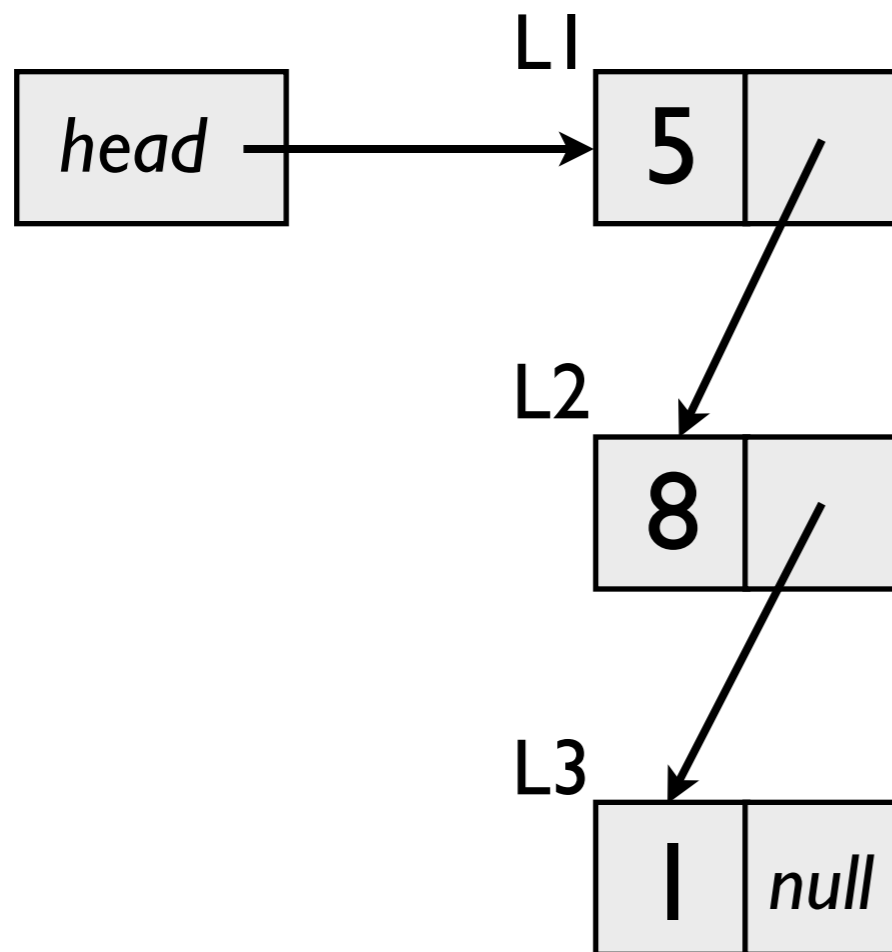
```
void push (int value) {  
    Node* oldHead;  
    Node* newHead := new Node();  
    newHead.item := value;  
    do {  
        oldHead := head;  
        newHead.next := head;  
    } while (!CAS(&head, oldHead, newHead));  
}
```

Push

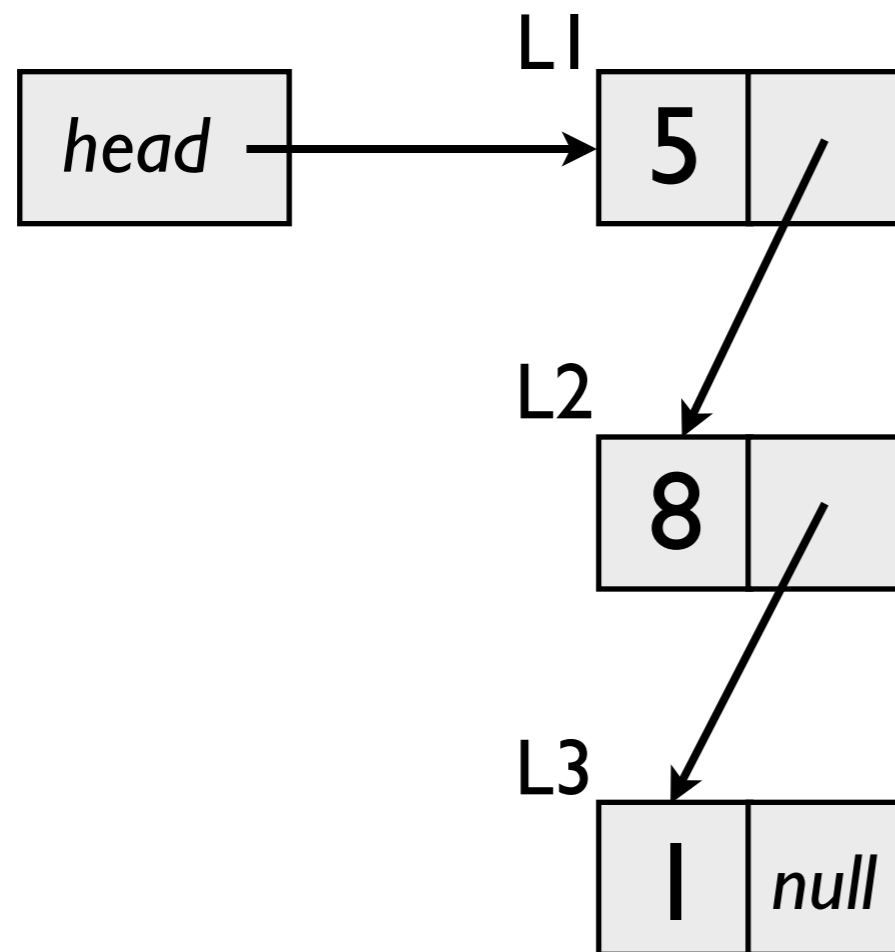
```
void push (int value) {  
    Node* oldHead;  
    Node* newHead := new Node();  
    newHead.item := value;  
    do {  
        oldHead := head;  
        newHead.next := head;  
    } while (!CAS(&head, oldHead, newHead));  
}
```

operation fails if another process has changed the head in the meantime (then loop repeats)

Push example (two threads)



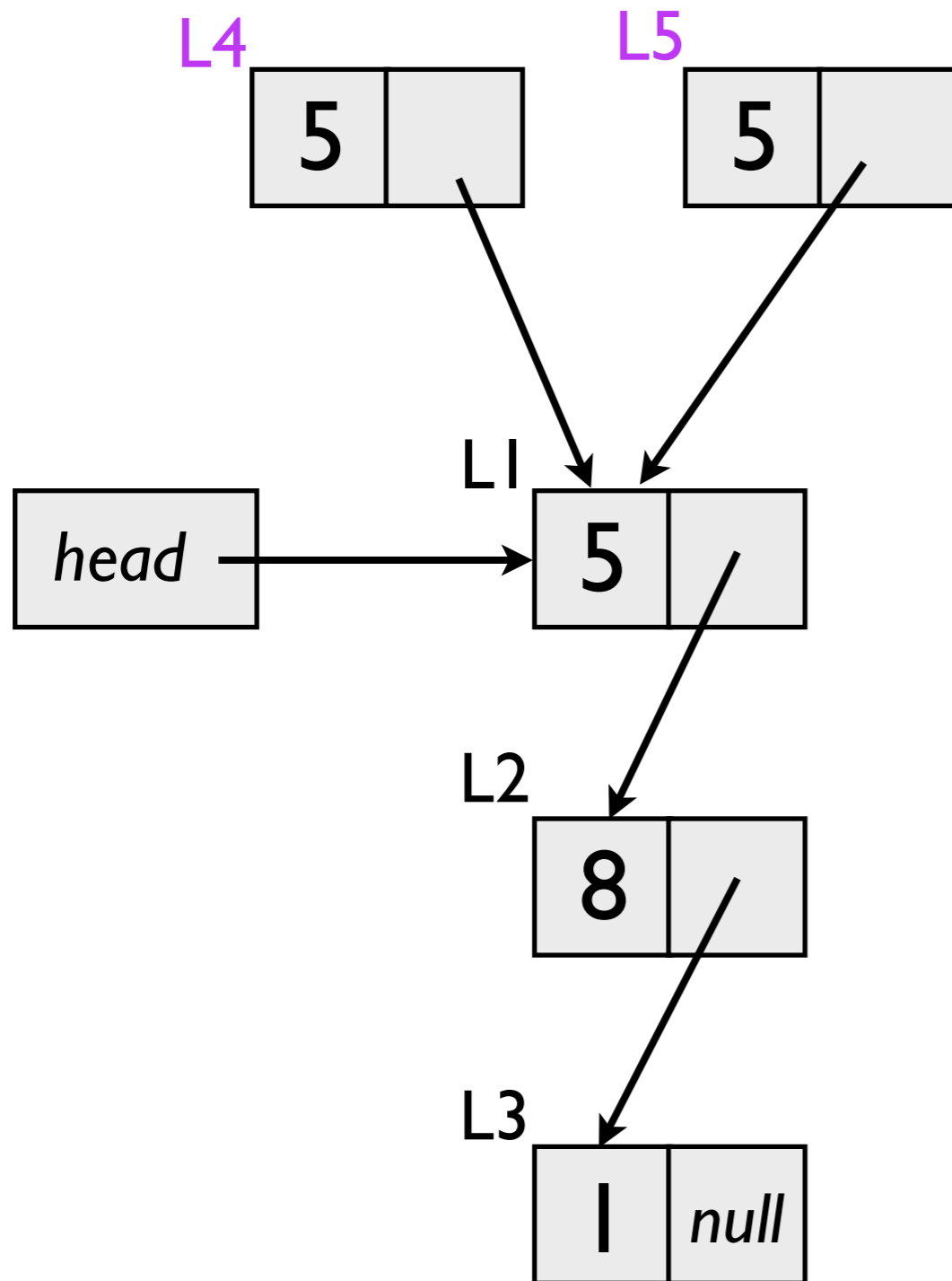
Push example (two threads)



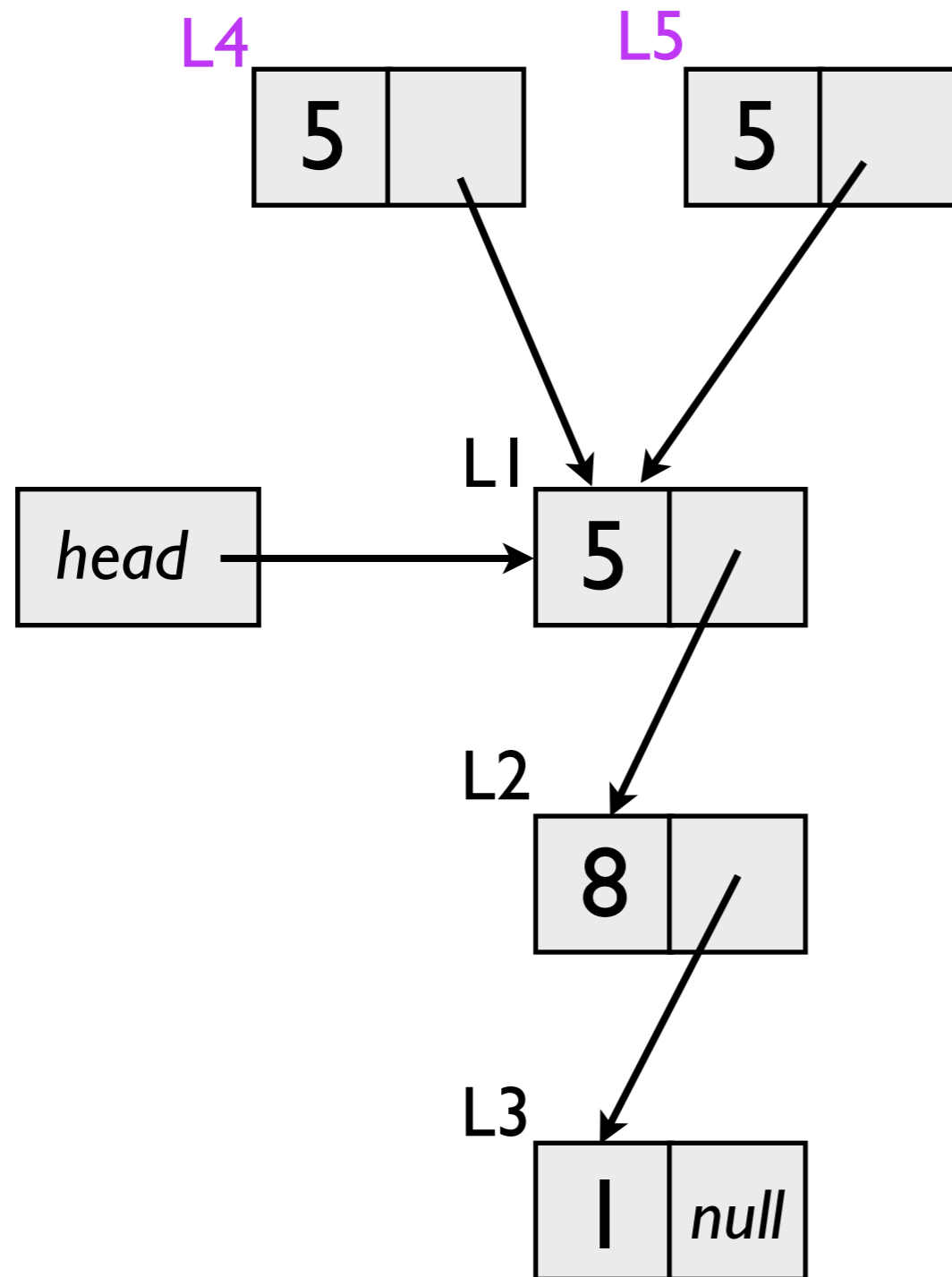
(1) Thread A: start push(5)

(2) Thread B: start push(5)

Push example (two threads)

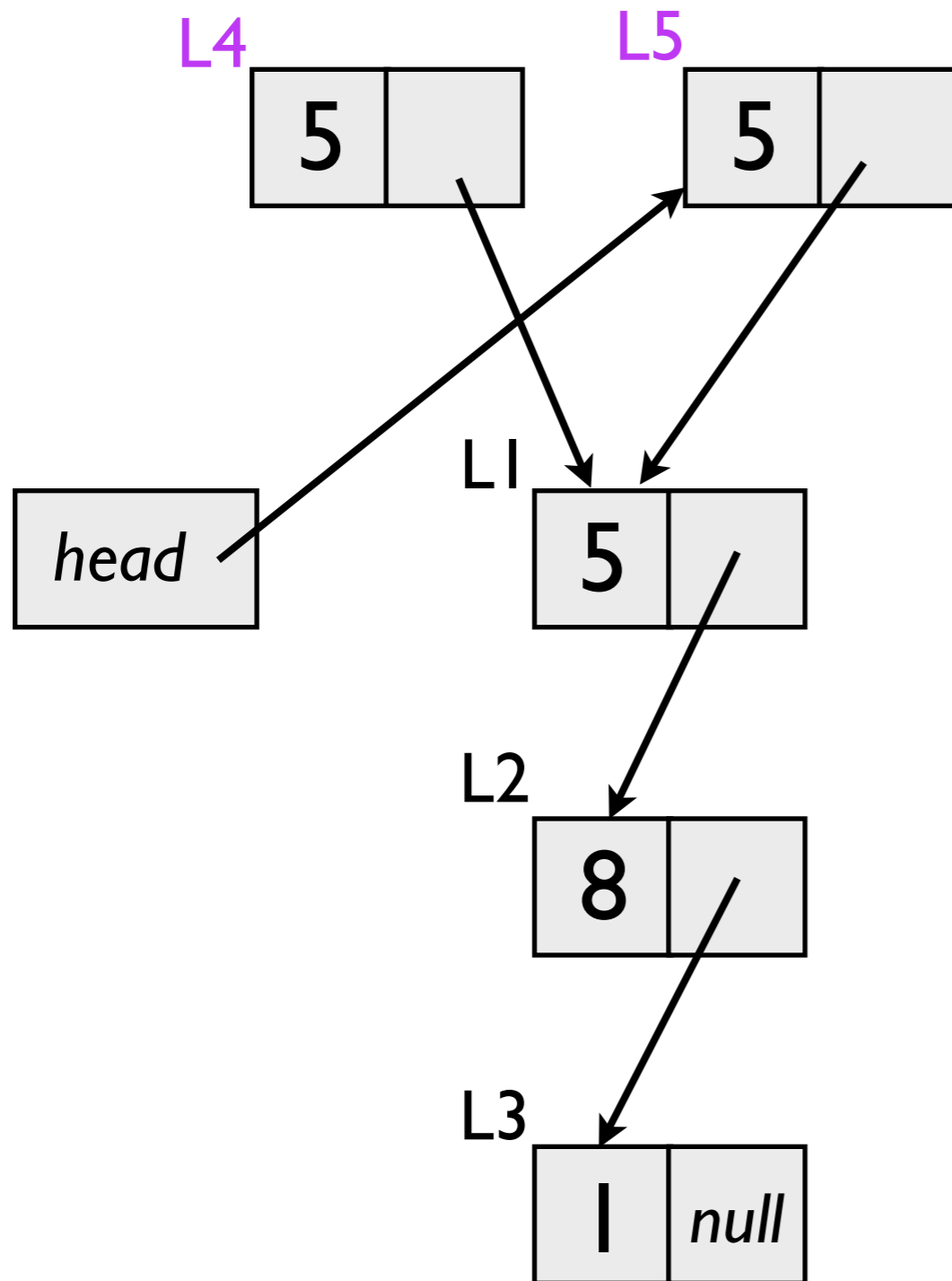


Push example (two threads)

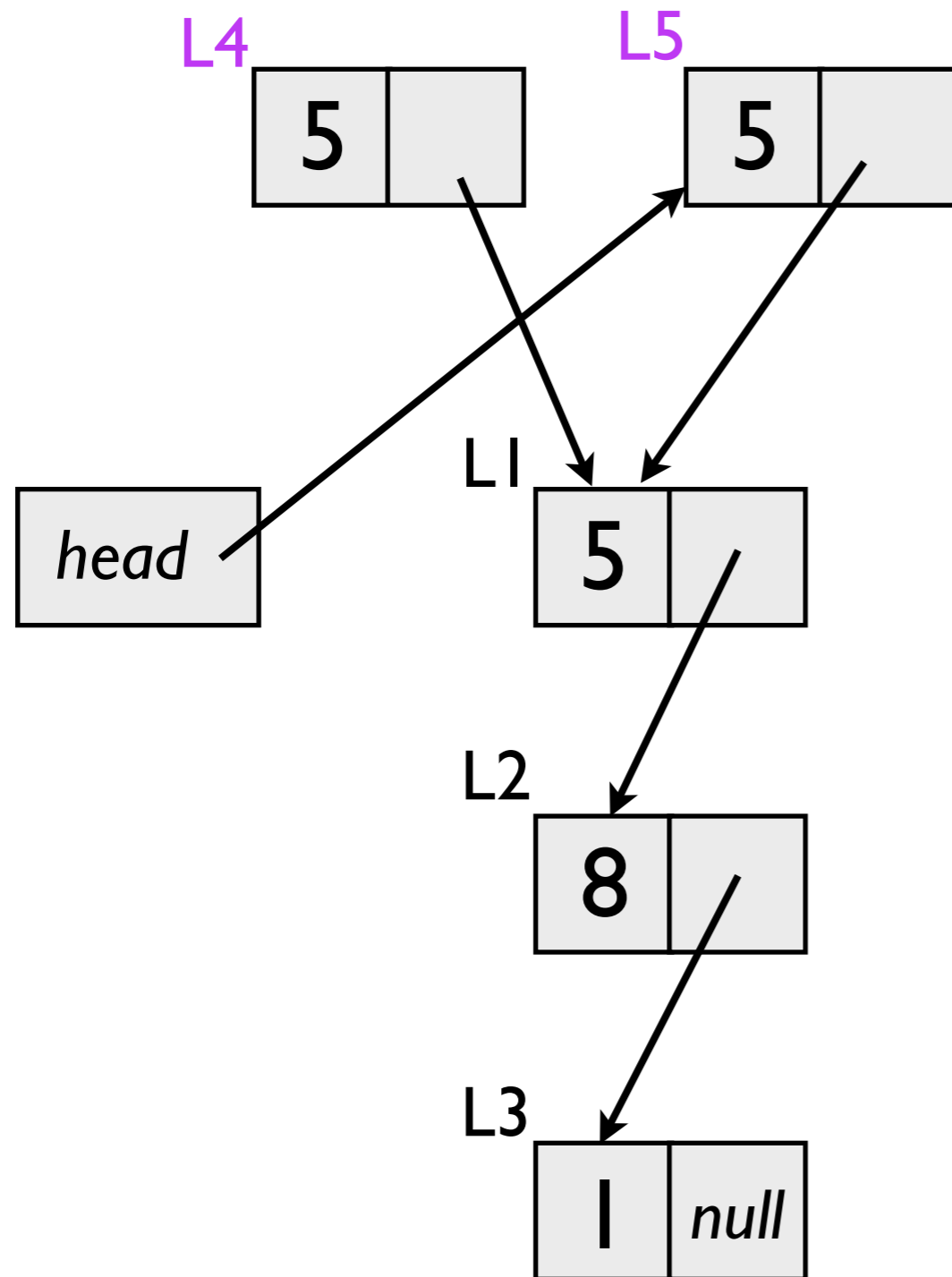


(3) Thread B: CAS **head** from L1 to L5

Push example (two threads)



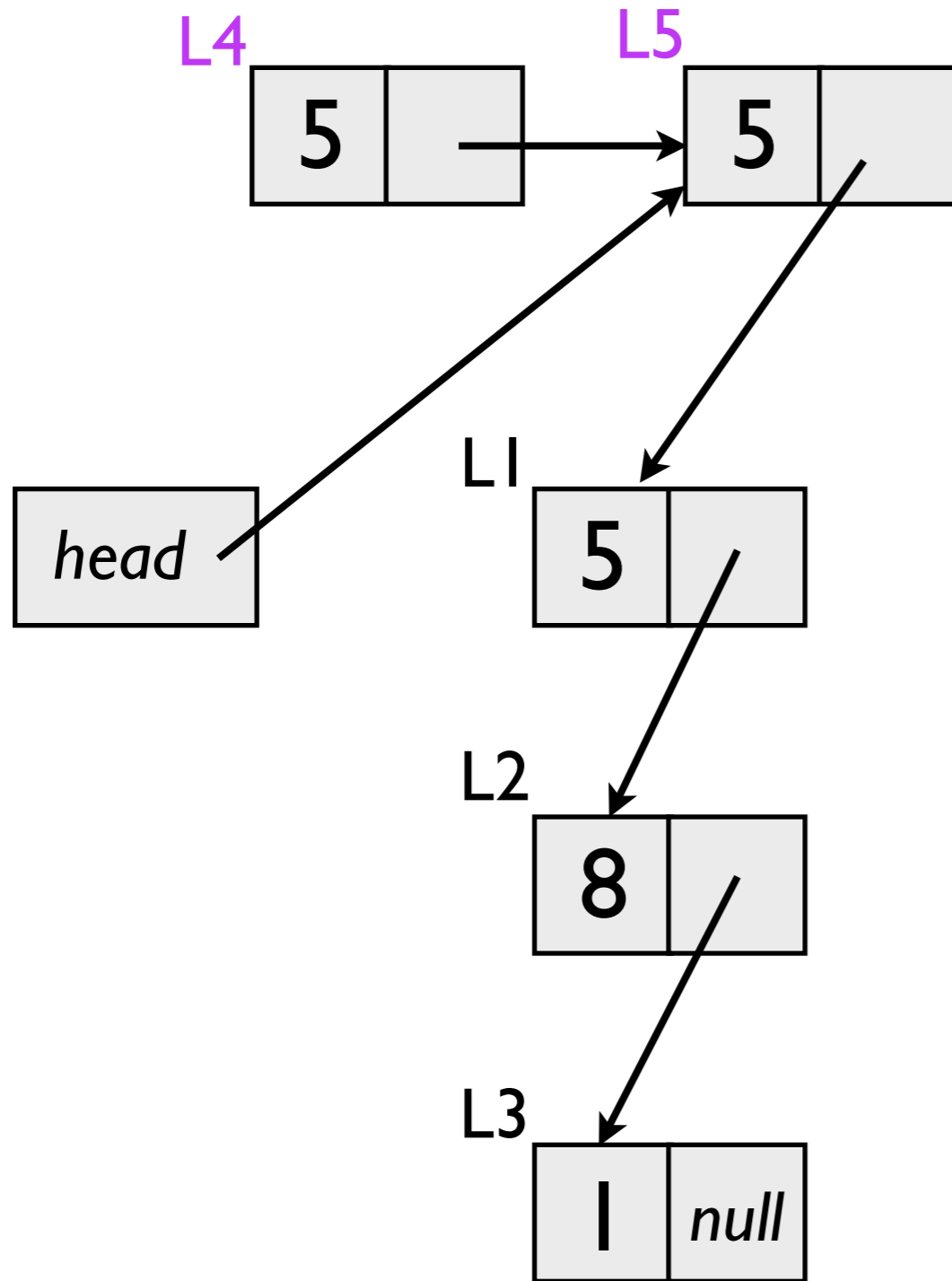
Push example (two threads)



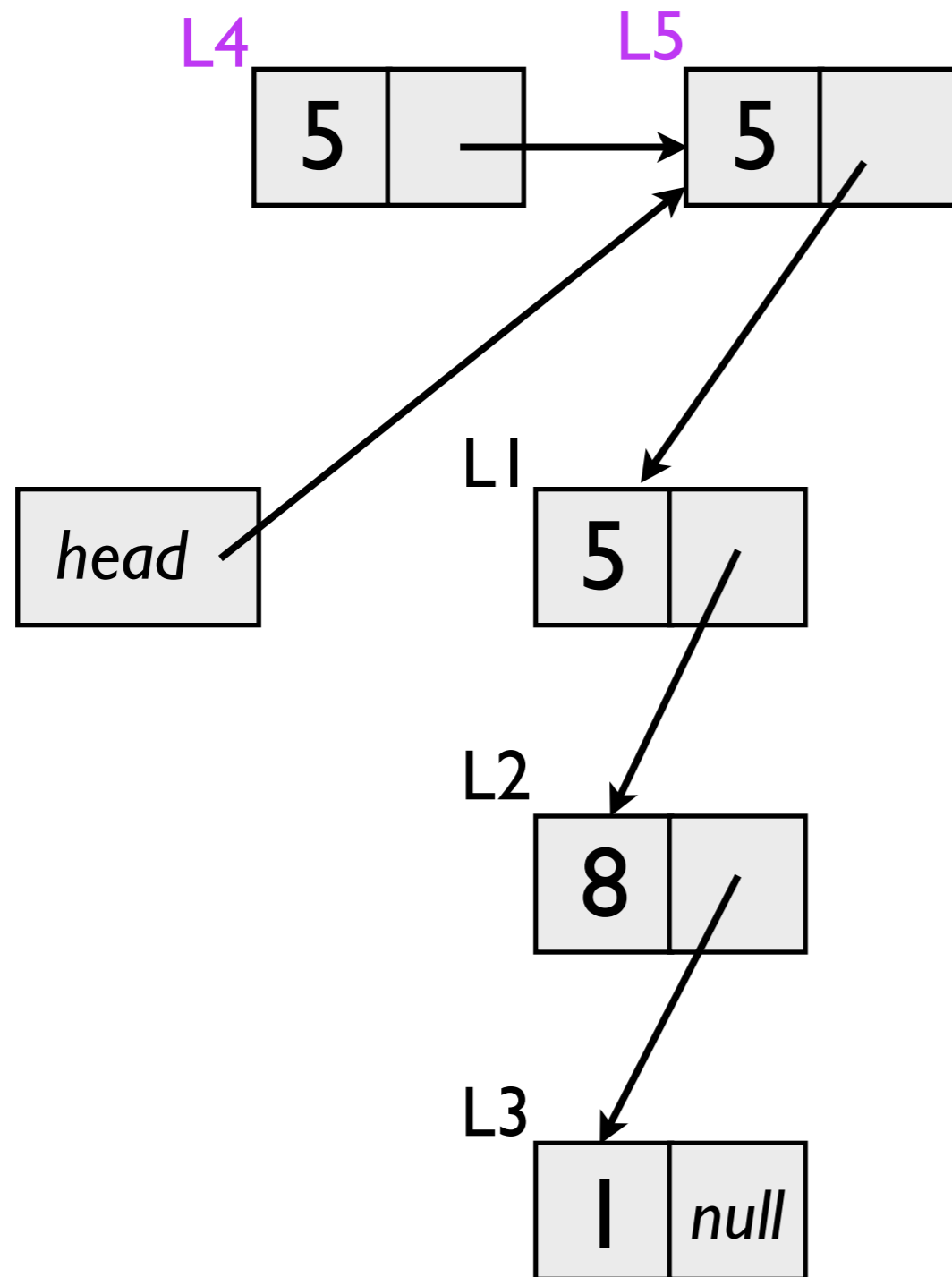
(4) Thread A: CAS **head**
from L1 to L4
=> FAILS

(5) Thread A: restart
loop; update pointer

Push example (two threads)

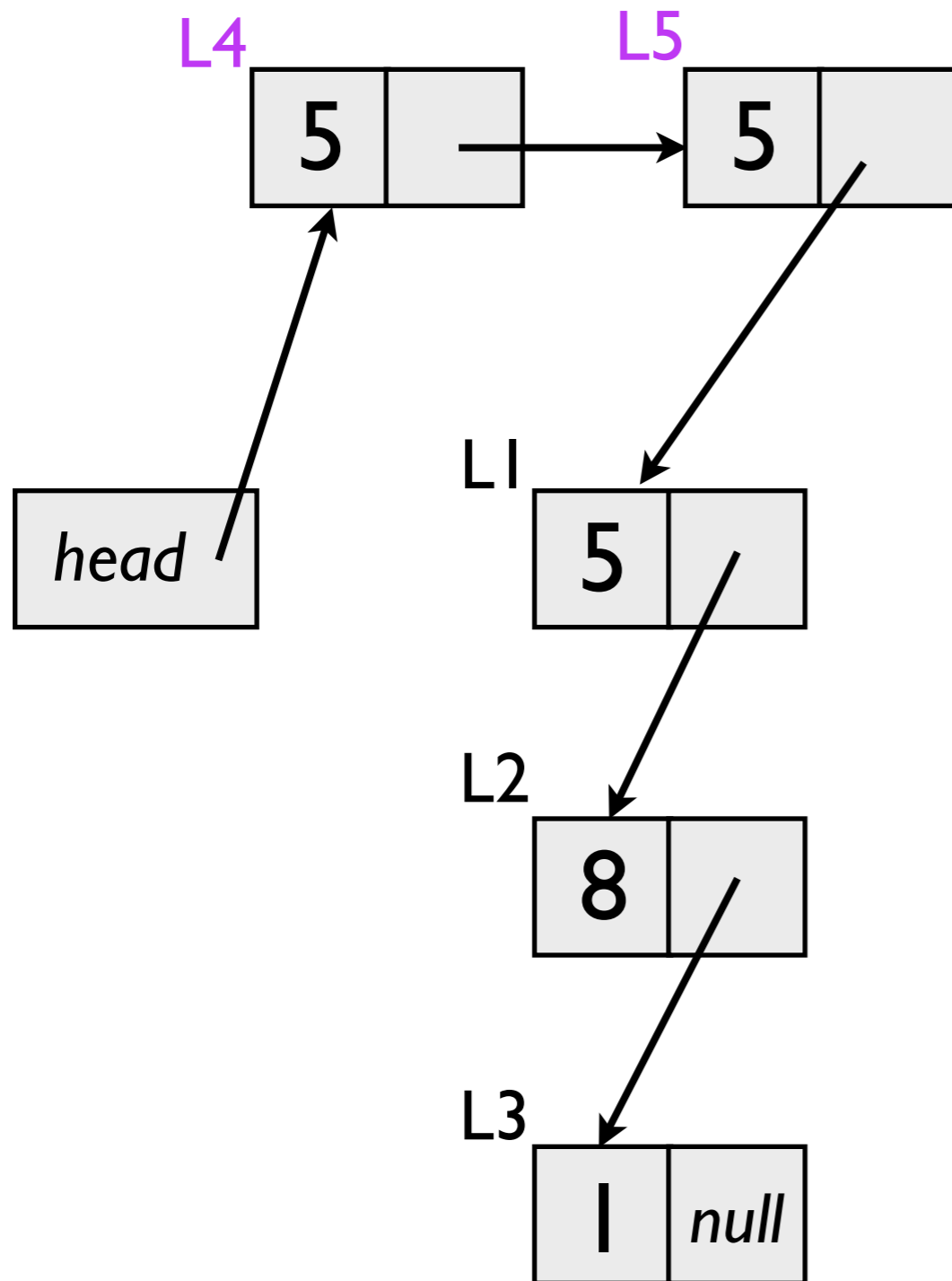


Push example (two threads)



(6) Thread A: CAS **head** from L5 to L4

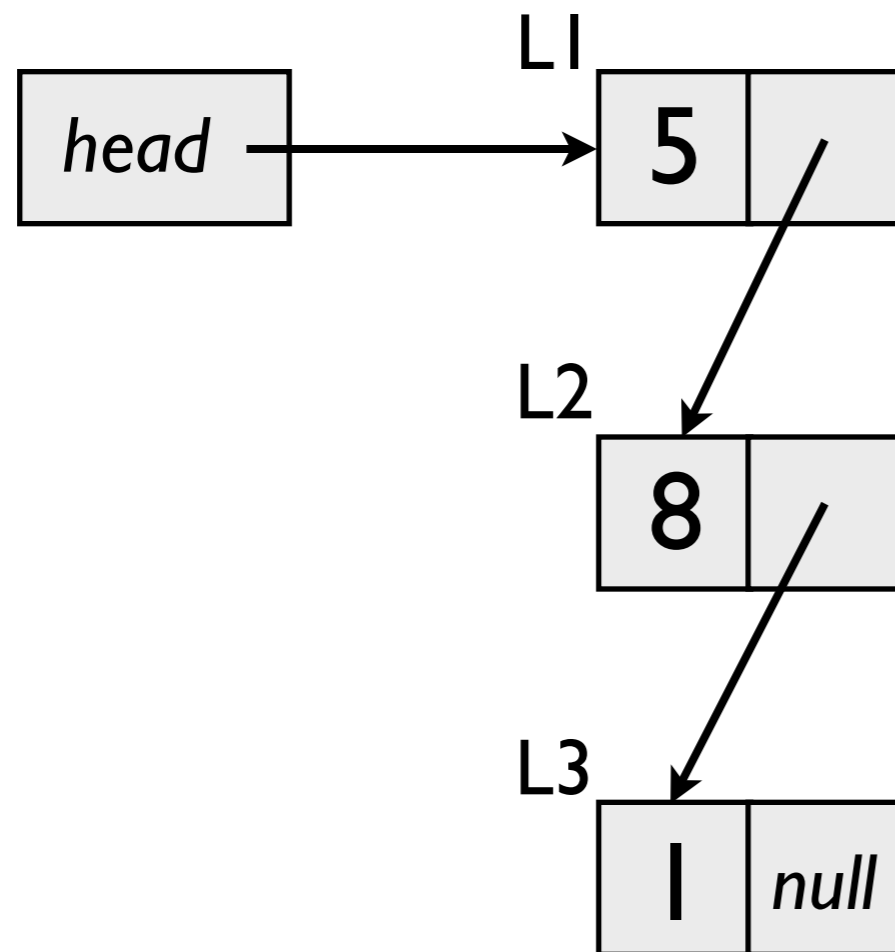
Push example (two threads)



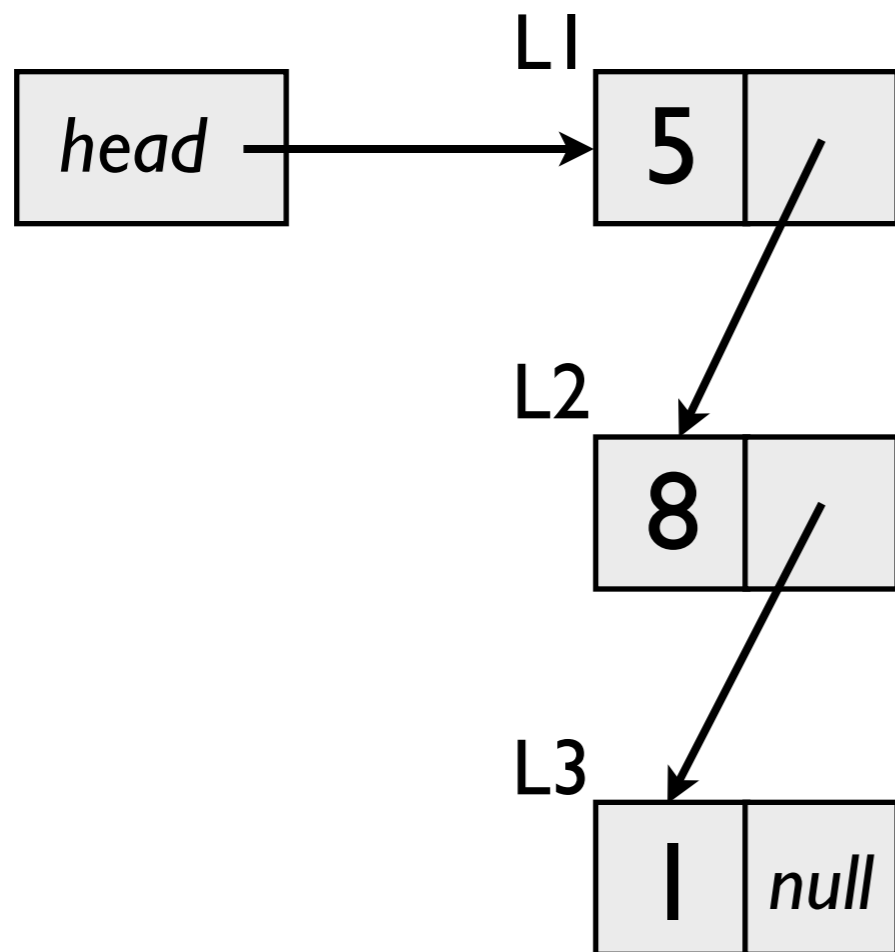
Pop

```
int pop () {  
    Node* oldHead;  
    Node* newHead;  
    do {  
        oldHead := head;  
        if(oldHead = null) return EMPTY;  
        newHead := oldHead.next;  
    } while(!CAS(&head, oldHead, newHead));  
    return oldHead.item;  
}
```

Pop example (two threads)



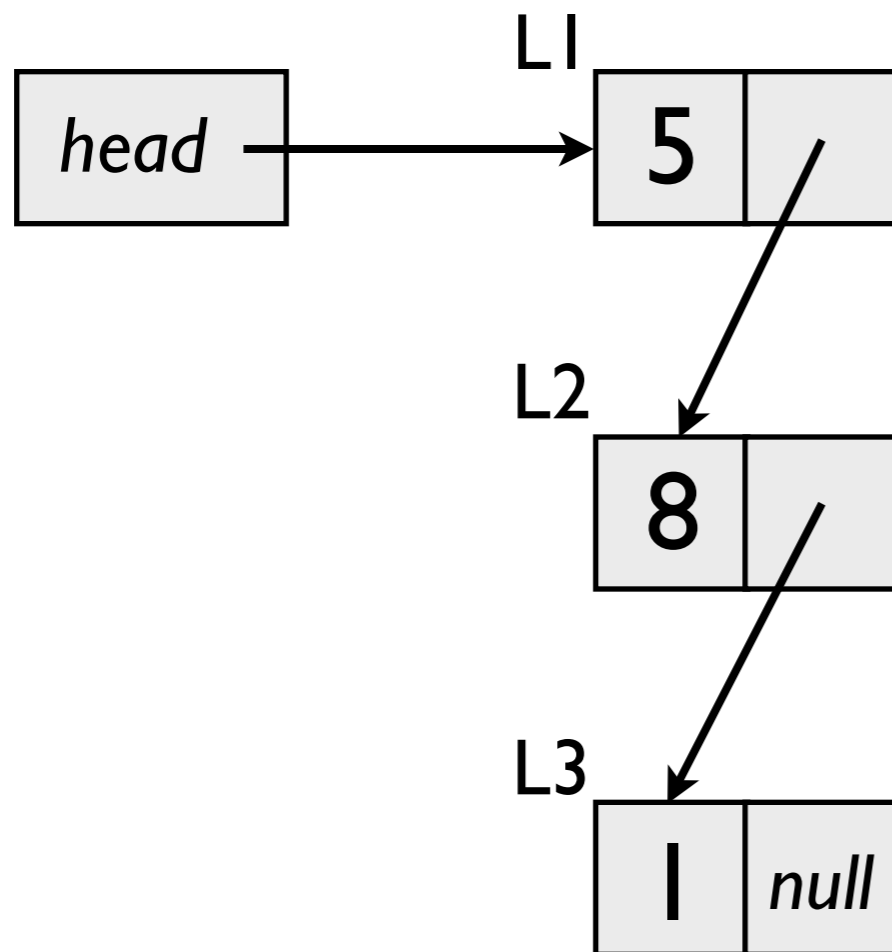
Pop example (two threads)



(1) Thread A: start pop, read 5 at L1

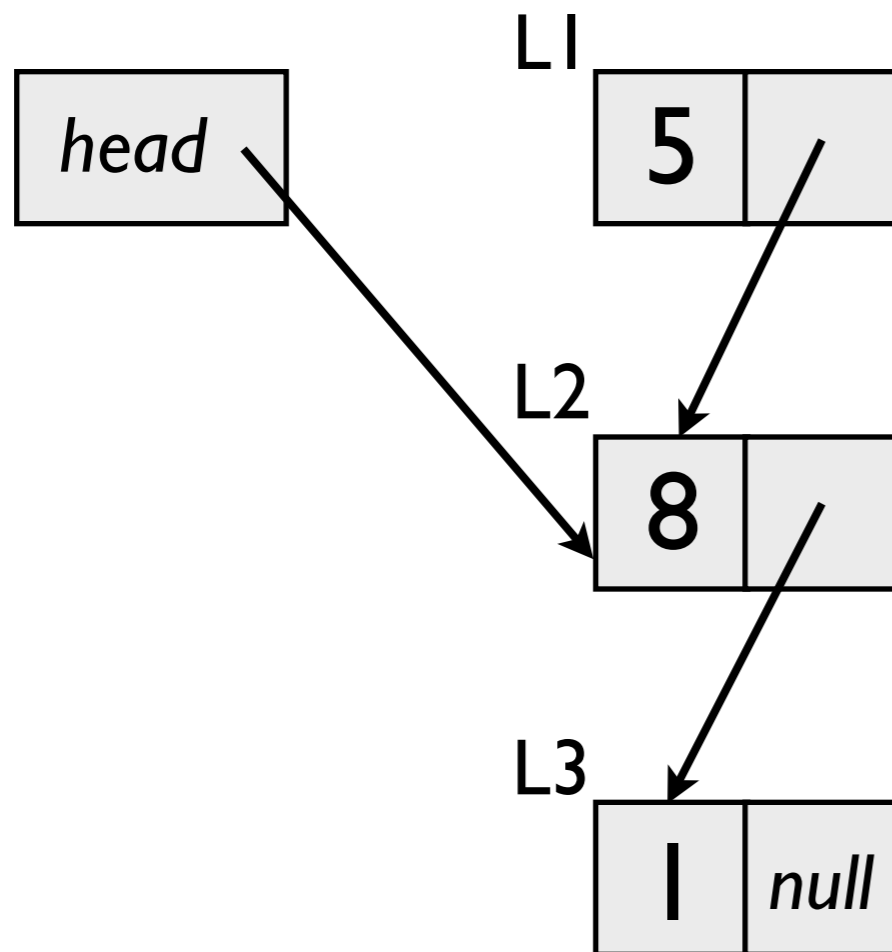
(2) Thread B: start pop, read 5 at L1

Pop example (two threads)



(3) Thread B: CAS **head** from L1 to L2

Pop example (two threads)

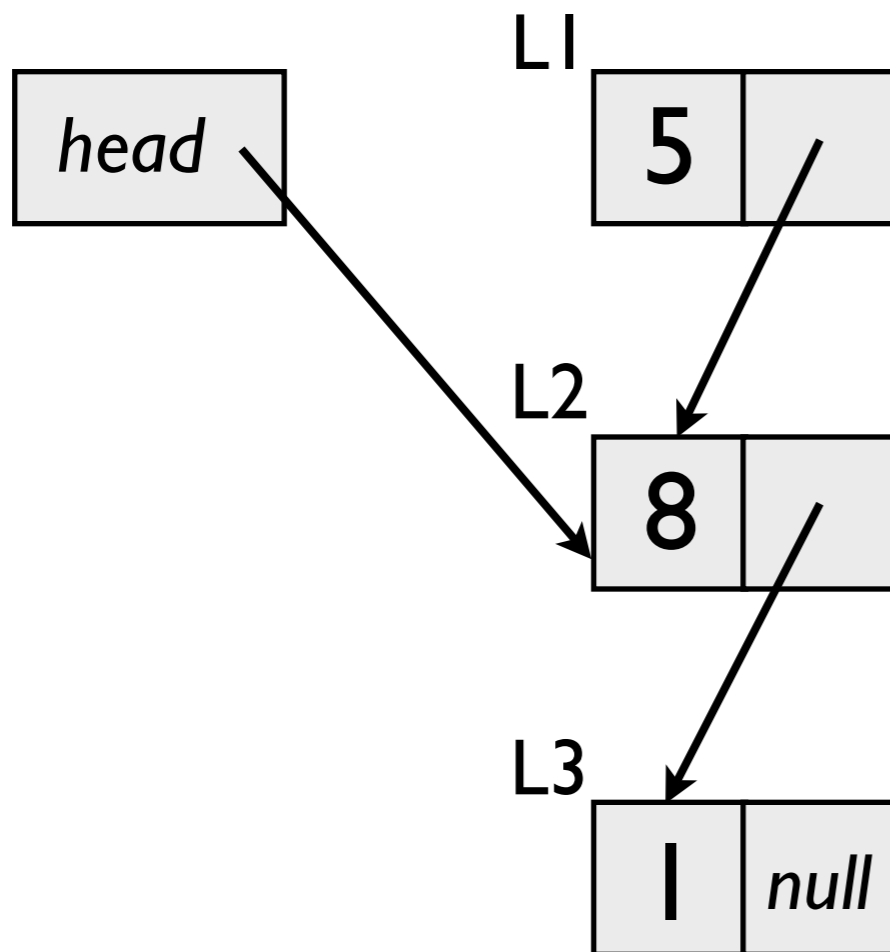


Pop example (two threads)

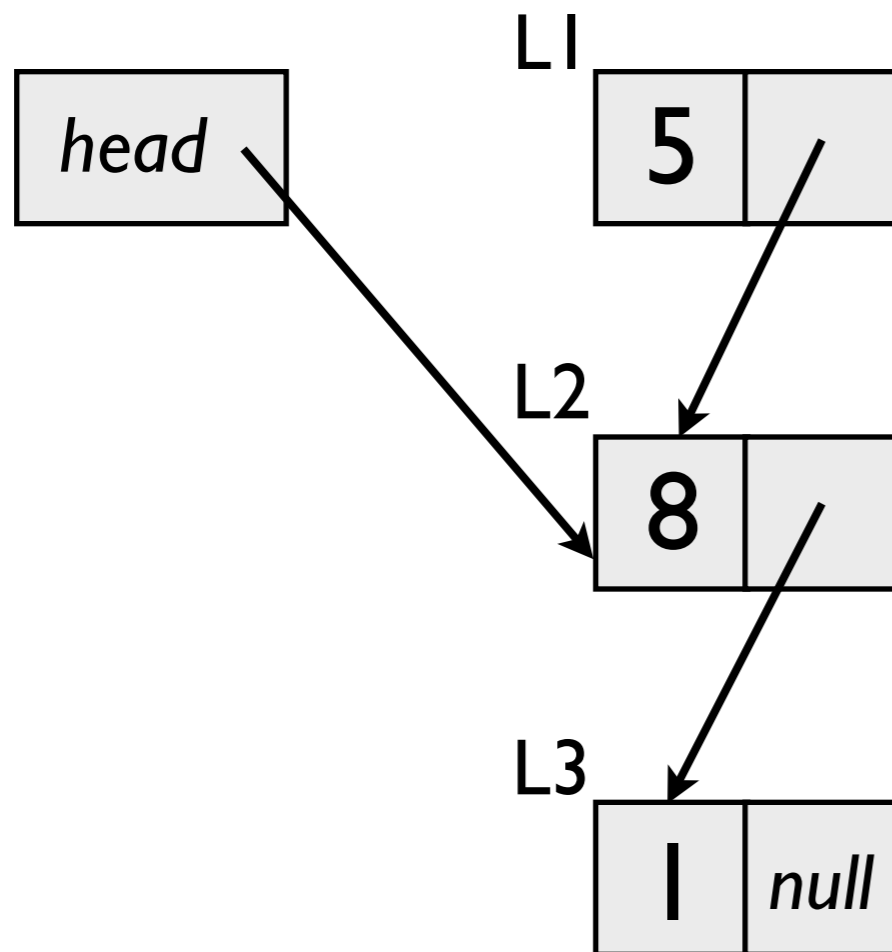
(4) Thread A: CAS **head**
from L1 to L2

=> **FAILS**

(5) Thread A: restart
loop, read 8 at L2

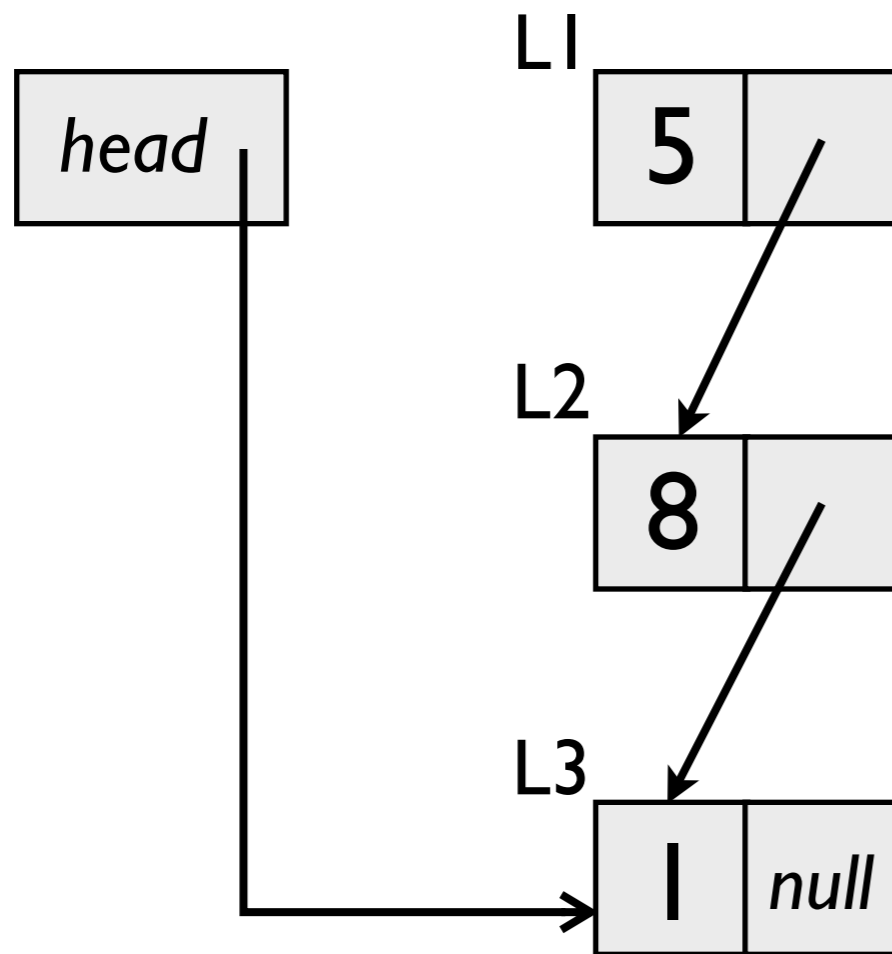


Pop example (two threads)



(6) Thread A: CAS **head**
from L2 to L3

Pop example (two threads)





CAS can be fooled!

- in the stack example, the following has to be avoided:

T_1 : starts `pop()` – reads value of current head as X

T_2 : executes `pop()`, removing X from the stack

T_2 : modifies the stack arbitrarily

T_2 : executes `push(X)`, putting X back on the stack

T_1 : finishes `pop()` – CAS succeeds, since X is on top

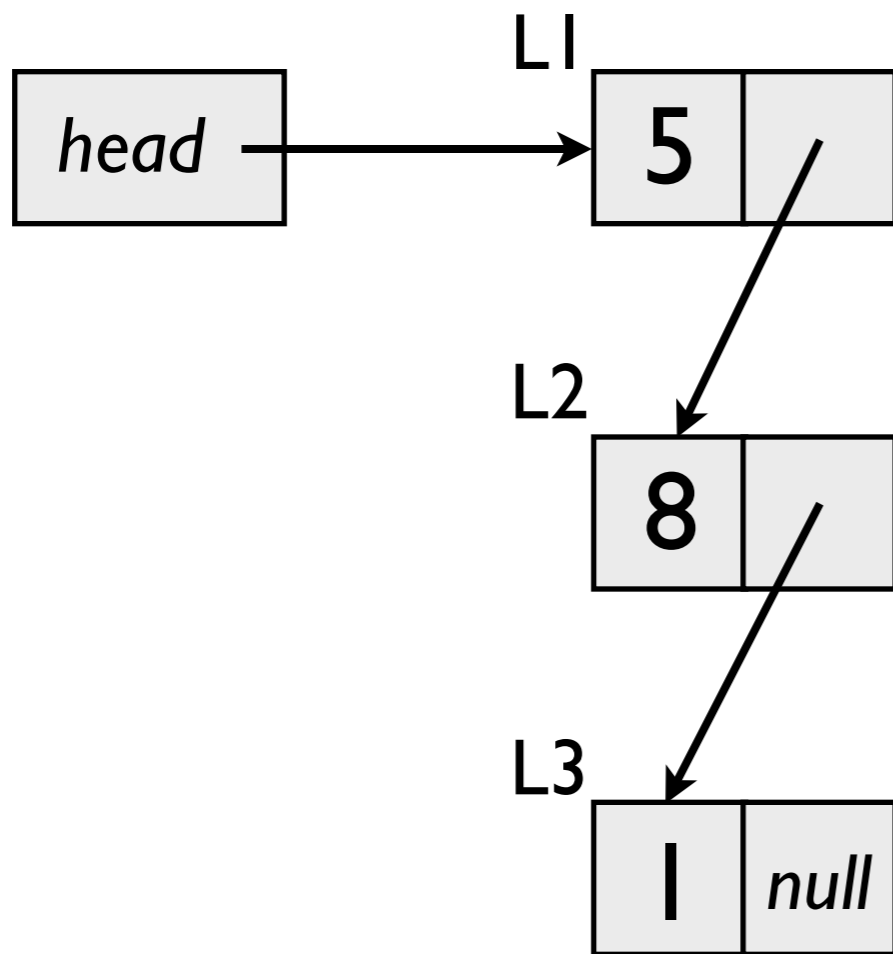


The ABA problem (I)

- this problematic pattern is called the **ABA problem**
 - T₁: a value is read from state A
 - T₂: the state is changed to state B
 - T₁: CAS operation cannot distinguish between A and B, so assumes the state is still A
- avoided in our stack since *push* always creates a **new node**, and the old node's location is **not freed**
- easy to introduce when **implementing memory management** yourself

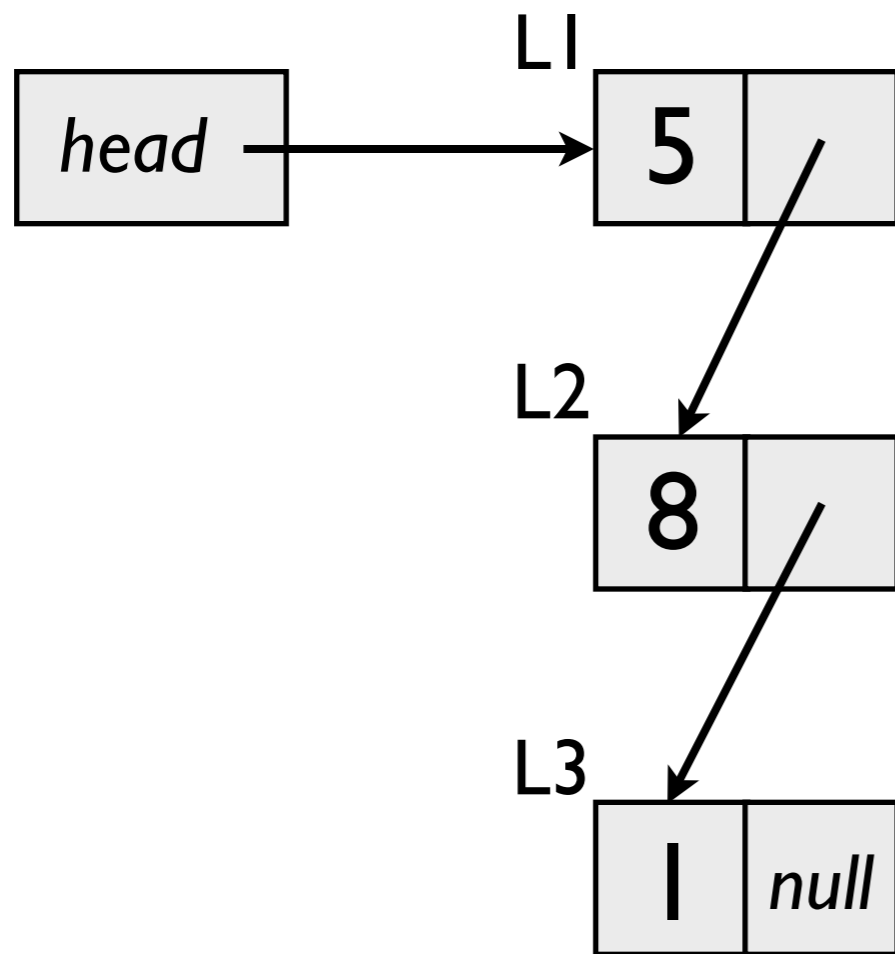


The ABA problem (2)





The ABA problem (2)

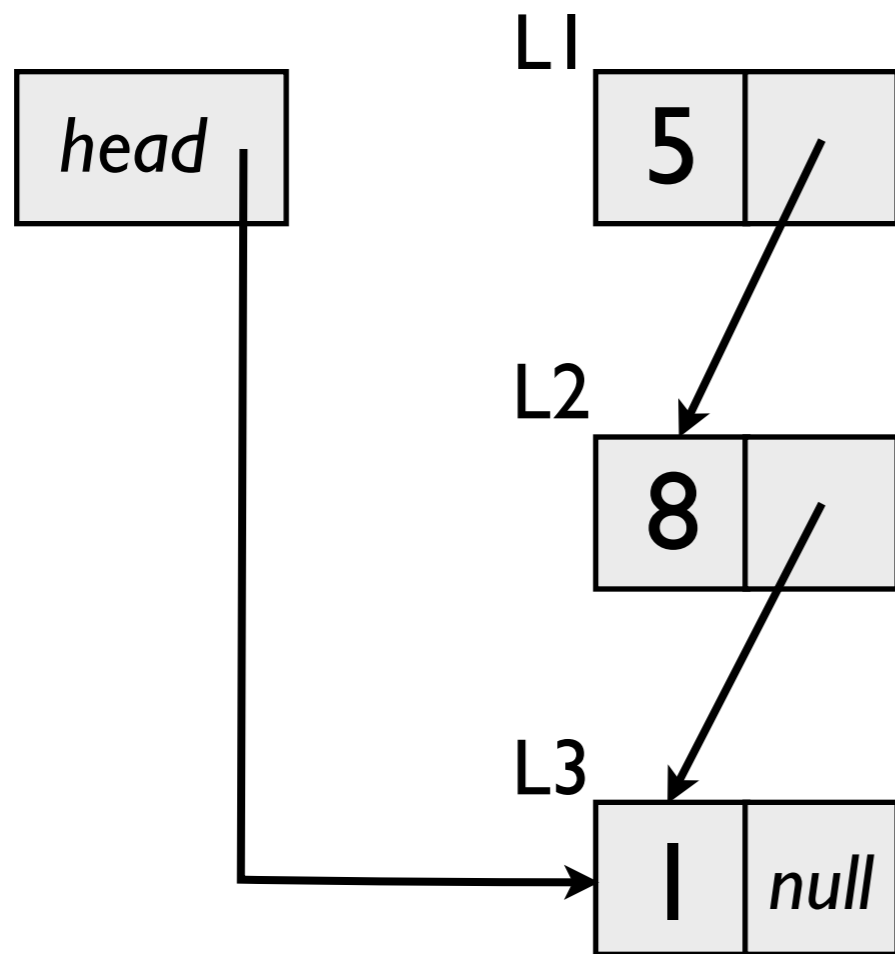


(1) thread A: started pop; about to CAS head from L1 to L2

(2) threads B, C: both do pop

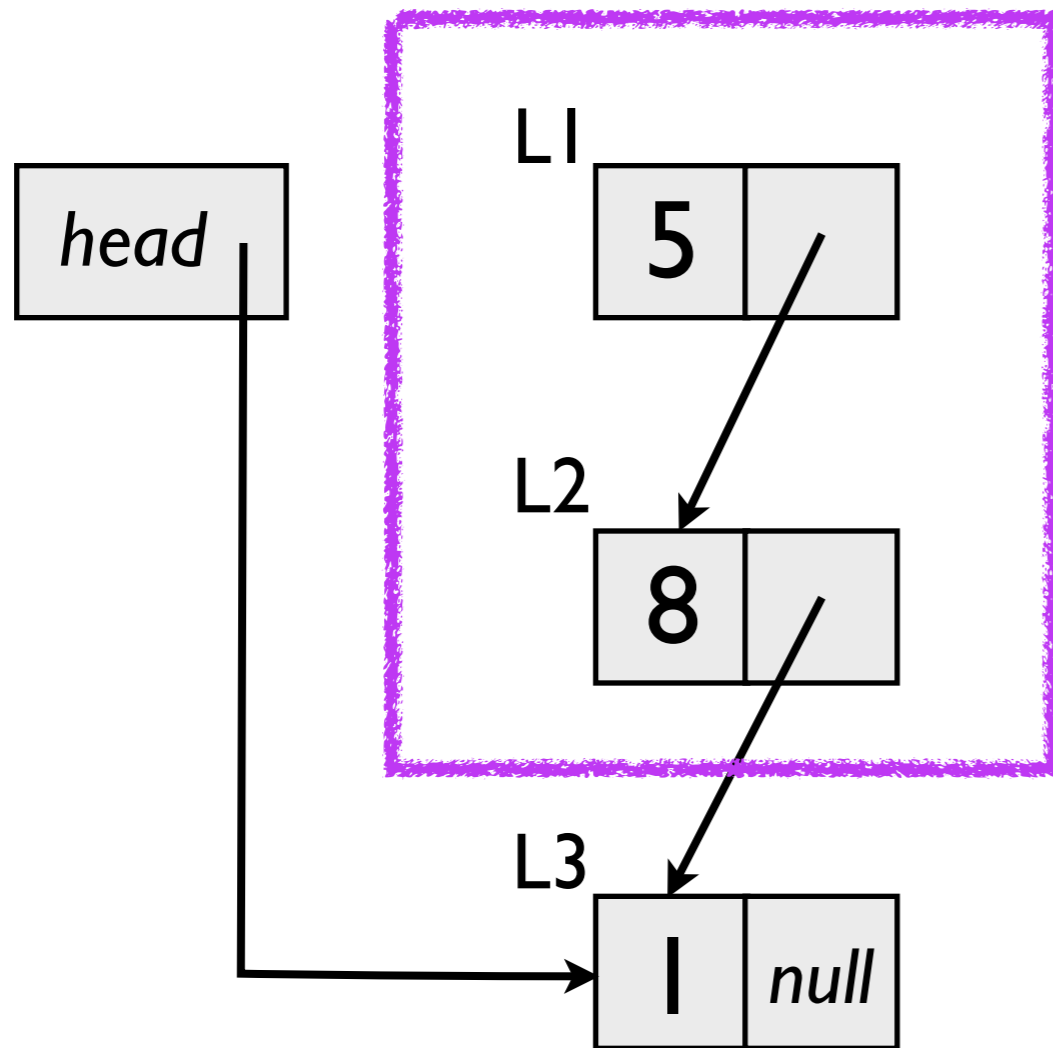


The ABA problem (2)





The ABA problem (2)

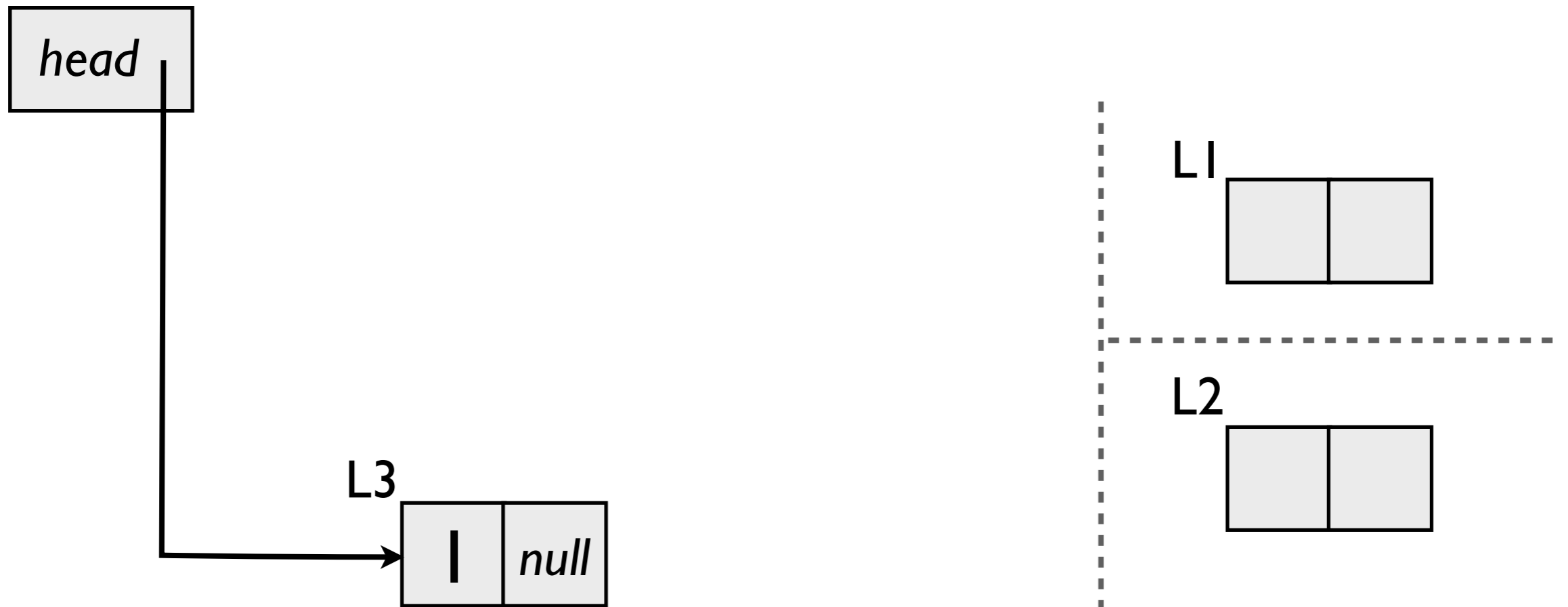


suppose we have a memory management policy that adds these to “thread-local pools” according to who did pop

threads then attempt to first recycle nodes from their local pools before creating new ones



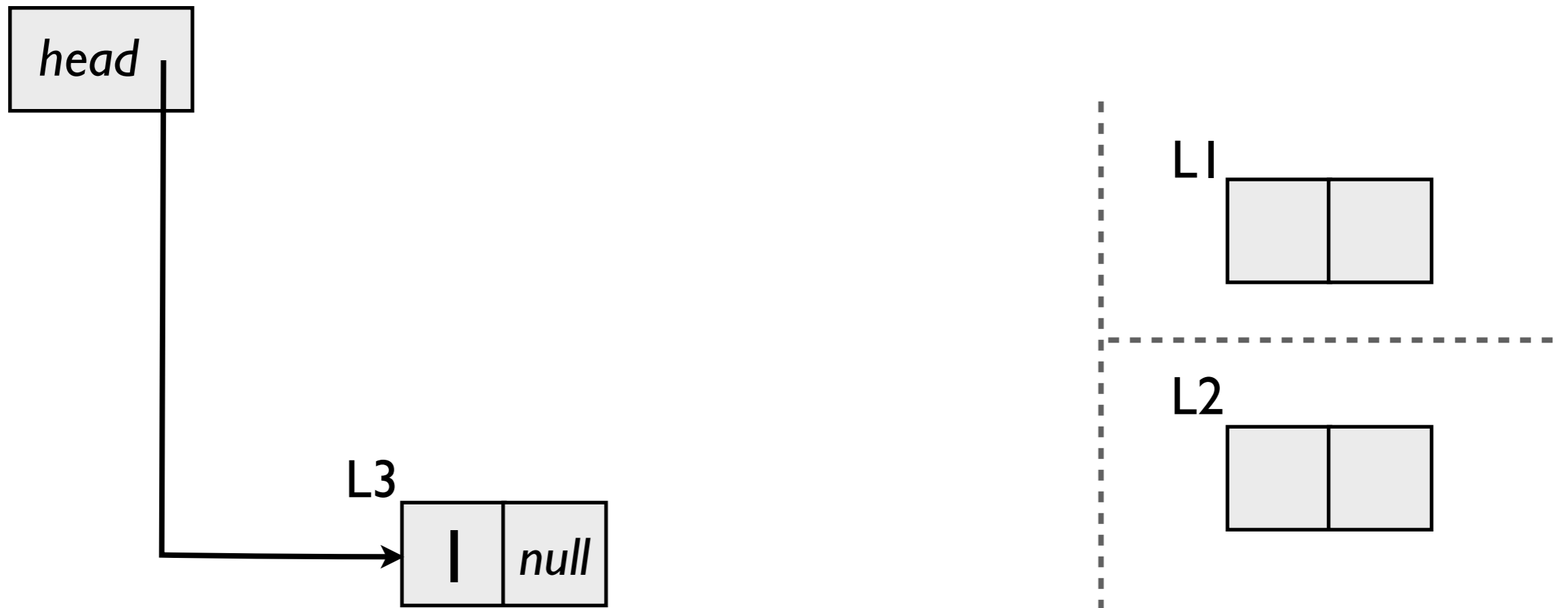
The ABA problem (2)





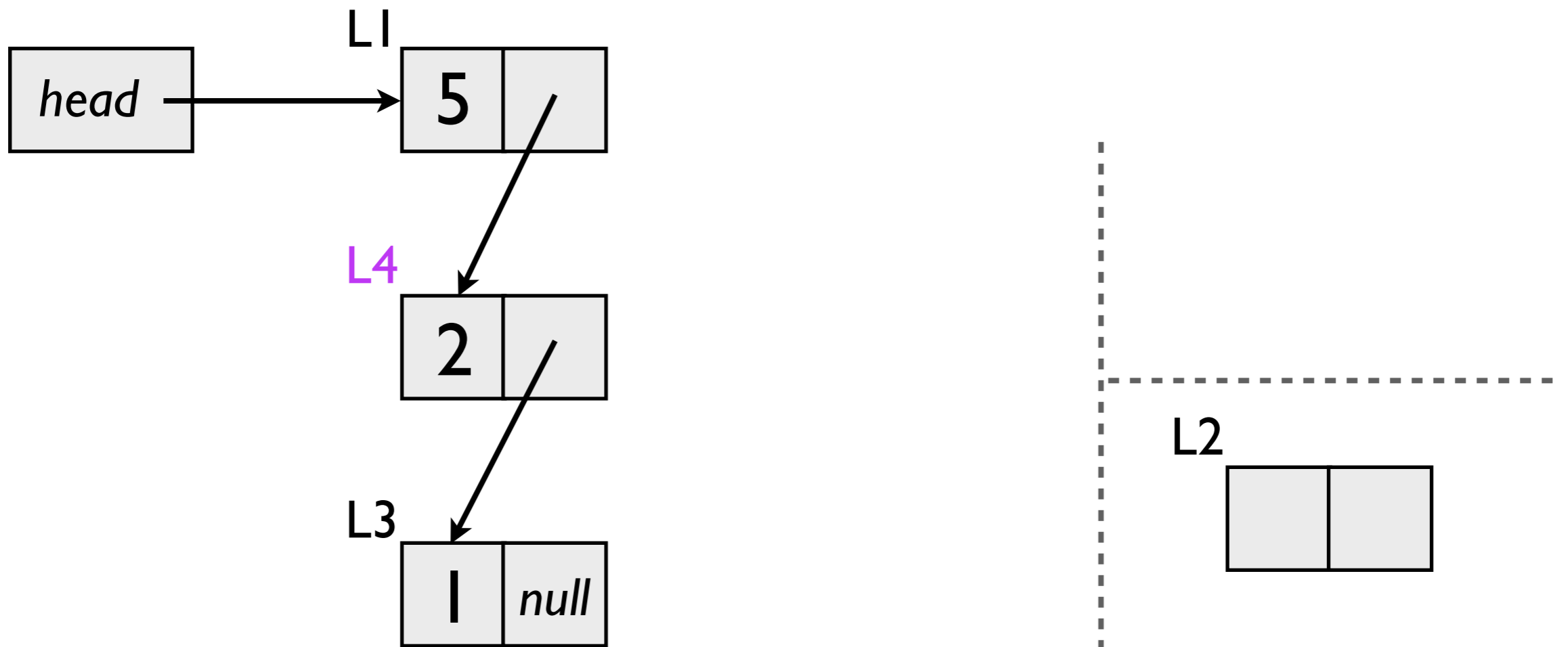
The ABA problem (2)

(3) various threads call push and pop...





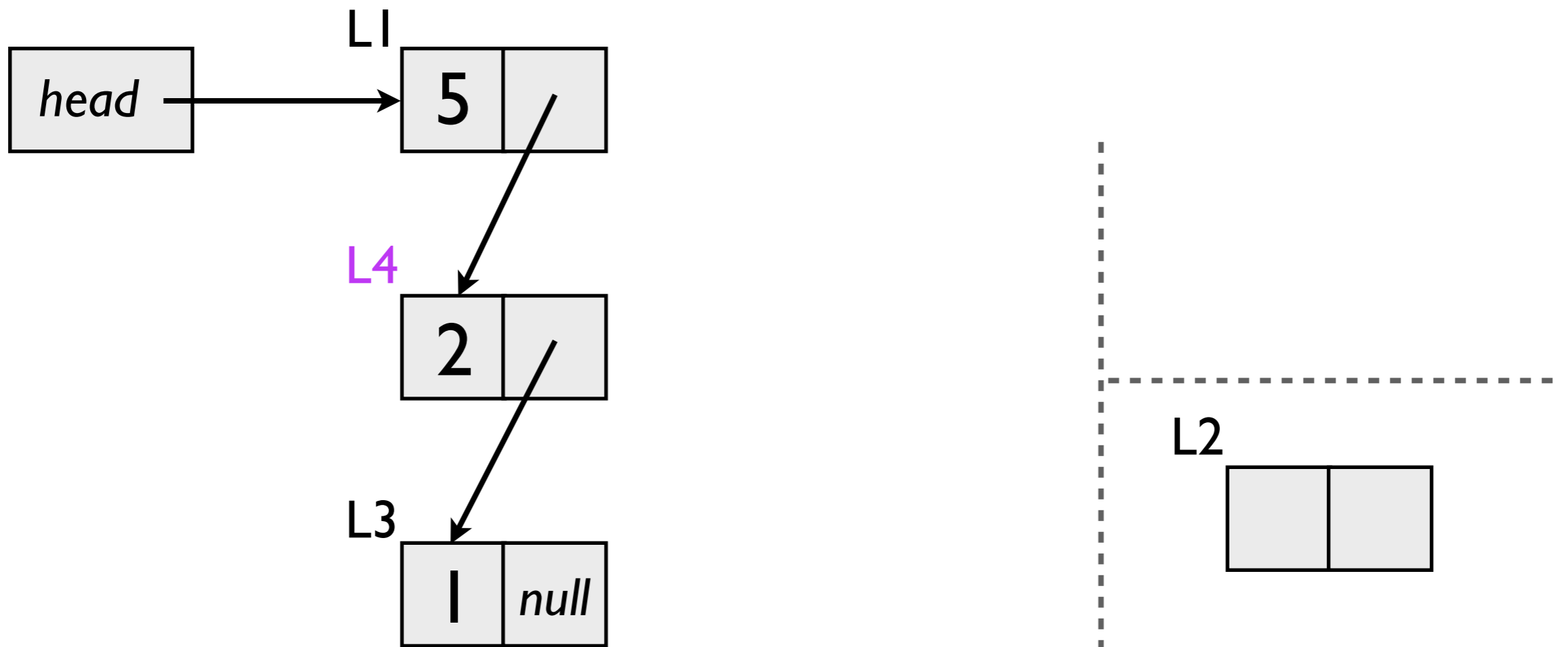
The ABA problem (2)





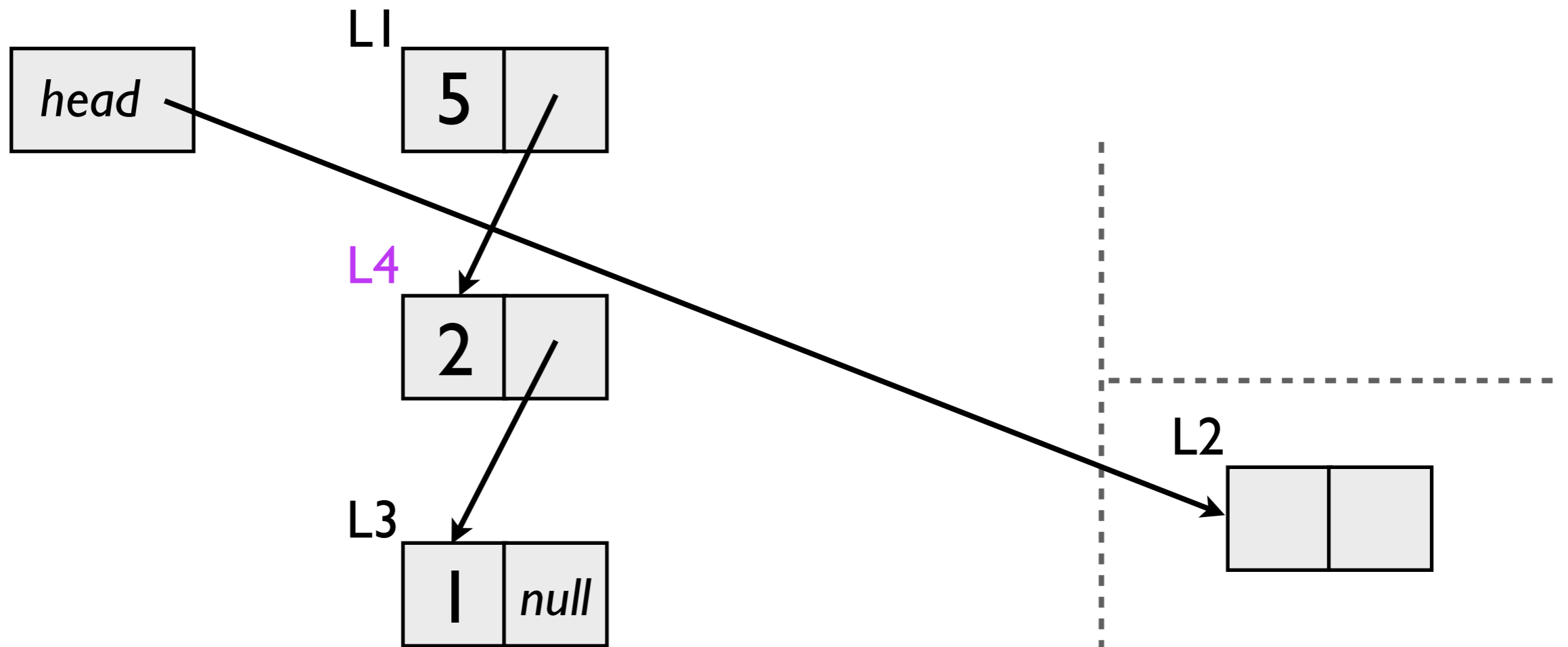
The ABA problem (2)

(4) thread A: wakes up, successfully calls CAS on head from L1 to L2





The ABA problem (2)

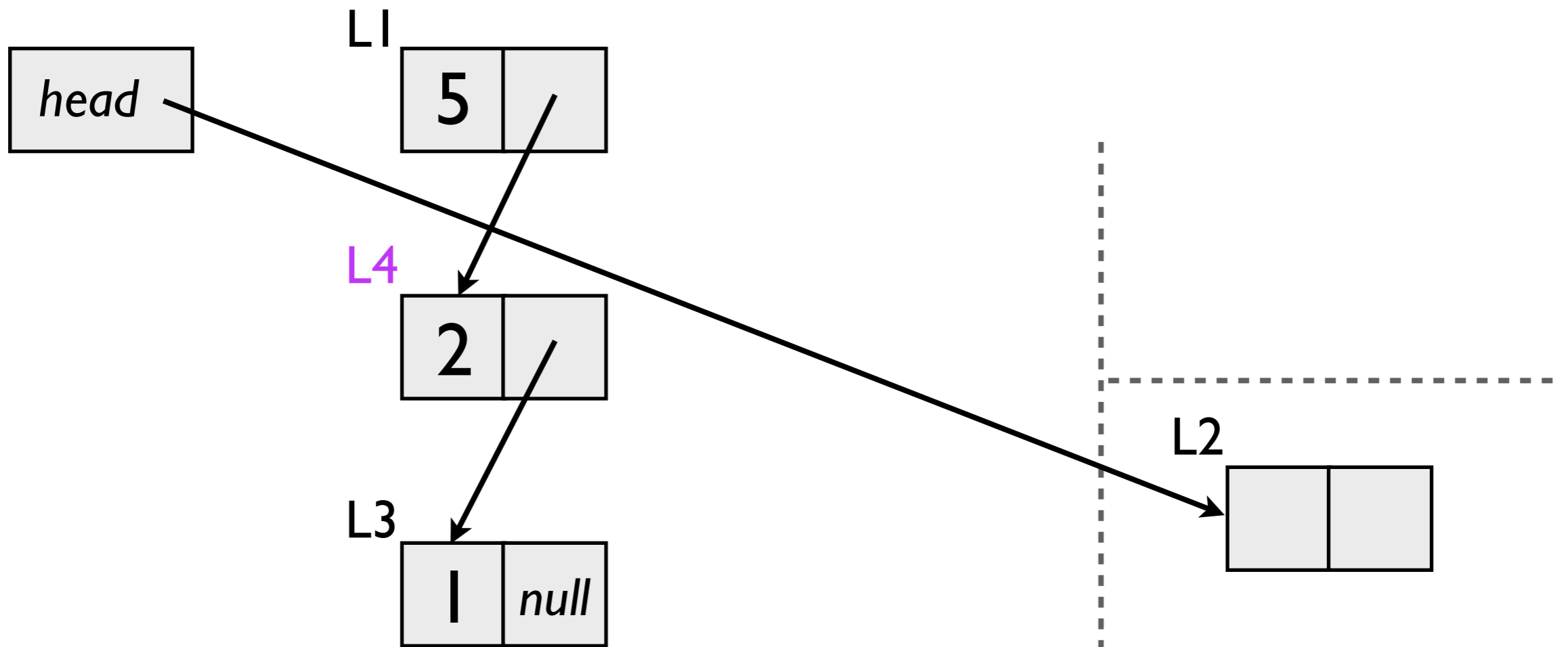




The ABA problem (2)



how can this be prevented?



Another example: a lock-free queue

- CAS can be used similarly to implement a concurrent **FIFO queue**, with enqueue and dequeue operations
- represented as a **linked list of nodes**, starting from a **sentinel** node; front of the queue at **head**, back of the queue at **tail**
- we implement it in Java (**garbage collection** prevents ABA)

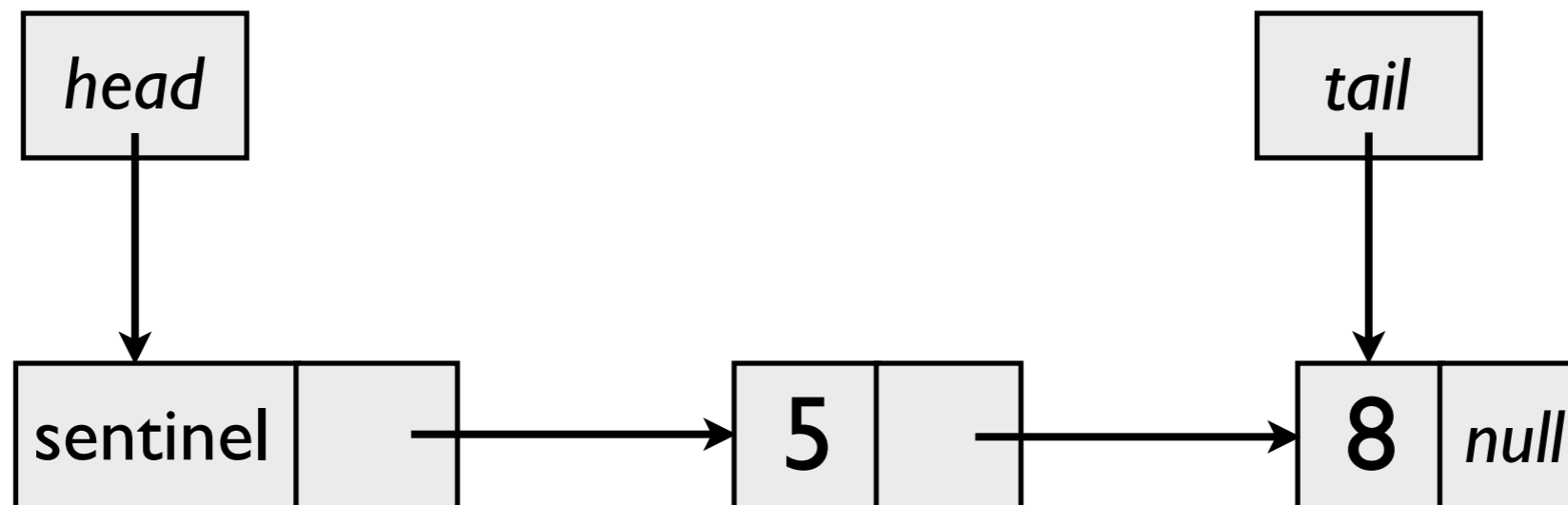
Another example: a lock-free queue

- CAS can be used similarly to implement a concurrent **FIFO queue**, with enqueue and dequeue operations
- represented as a **linked list of nodes**, starting from a **sentinel** node; front of the queue at **head**, back of the queue at **tail**
- we implement it in Java (**garbage collection** prevents ABA)

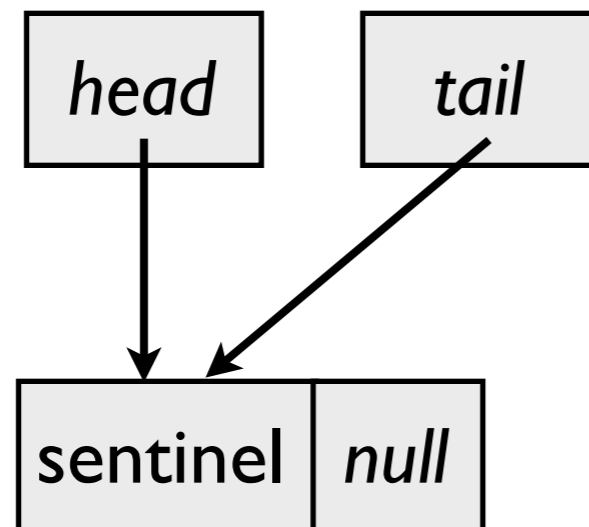
```
public class Node {  
    public int item;  
    public AtomicReference<Node> next;  
    public Node(int item) {  
        this.item = item;  
        this.next = new AtomicReference<Node>(null);  
    }  
}
```

Another example: a lock-free queue

- example queue:



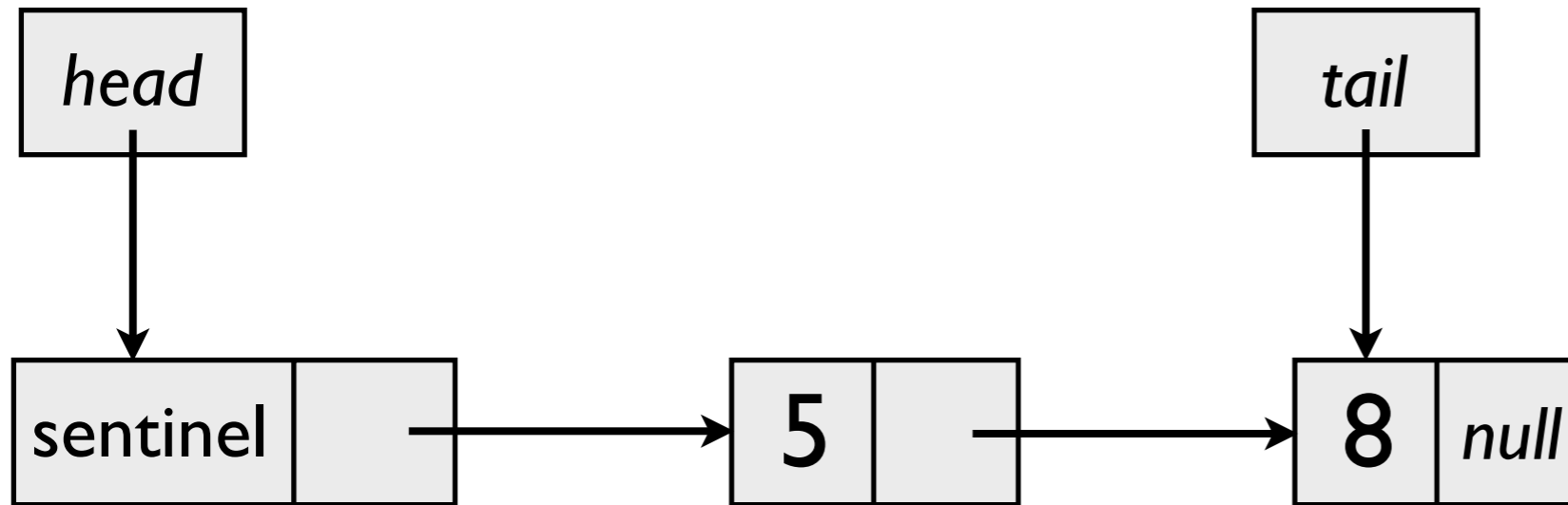
- the **sentinel** (or “dummy node”) prevents **head** and **tail** from pointing to null; for example, this is the empty queue:



Enqueue

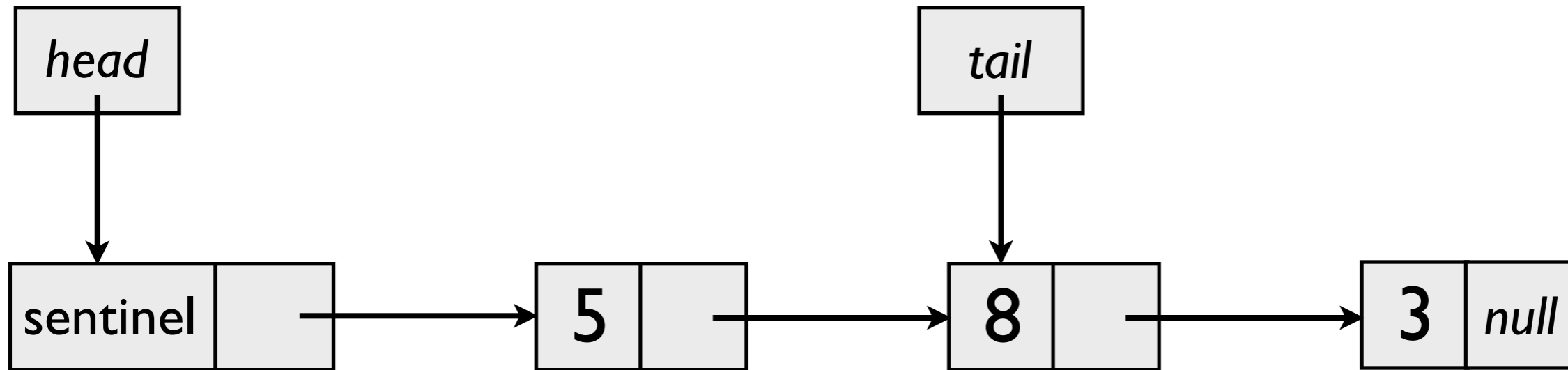
- the **enqueue** operation should add a Node to the back of the queue, updating both next for that node as well as **tail**
- requires two CAS operations
 - => must be prepared to encounter a “half-finished” enqueue operation and finish the job*
 - => “helping technique”*
- strategy: loop until you succeed... but if you don't, at least you can help fix pointers!

Enqueue example (one thread)

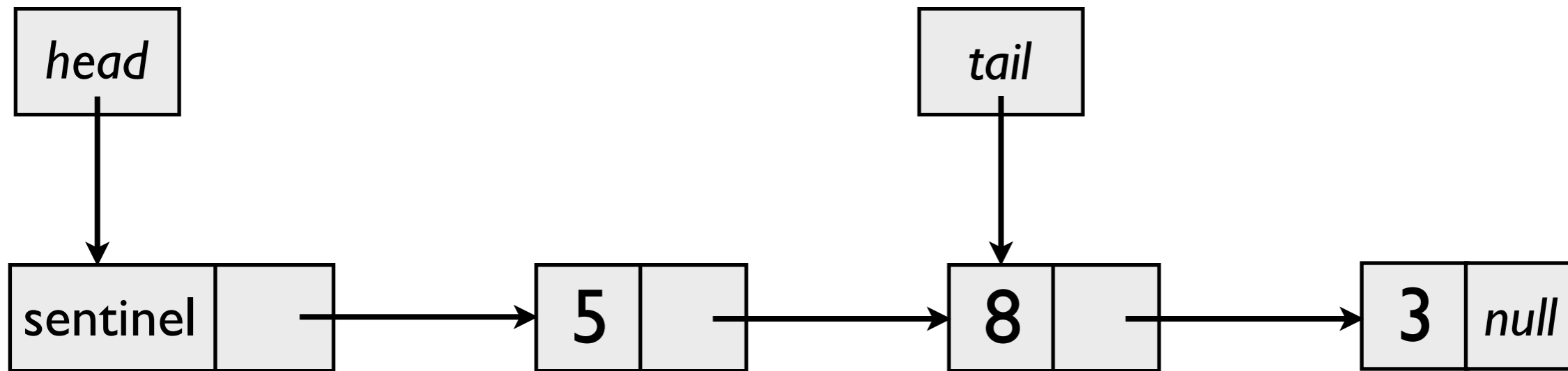


(I) Thread A: CAS next

Enqueue example (one thread)

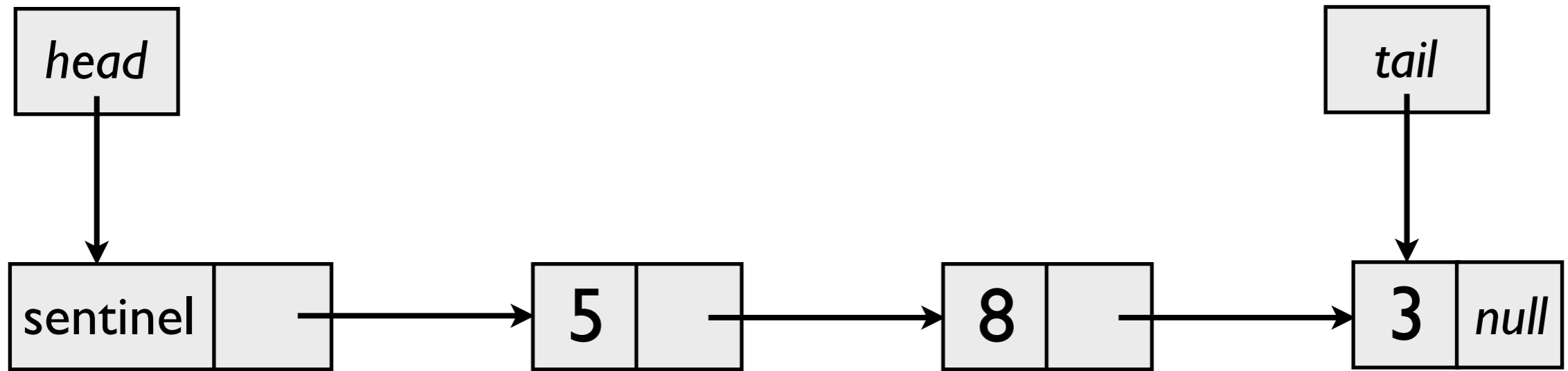


Enqueue example (one thread)



(2) Thread A: CAS **tail**

Enqueue example (one thread)



Enqueue method

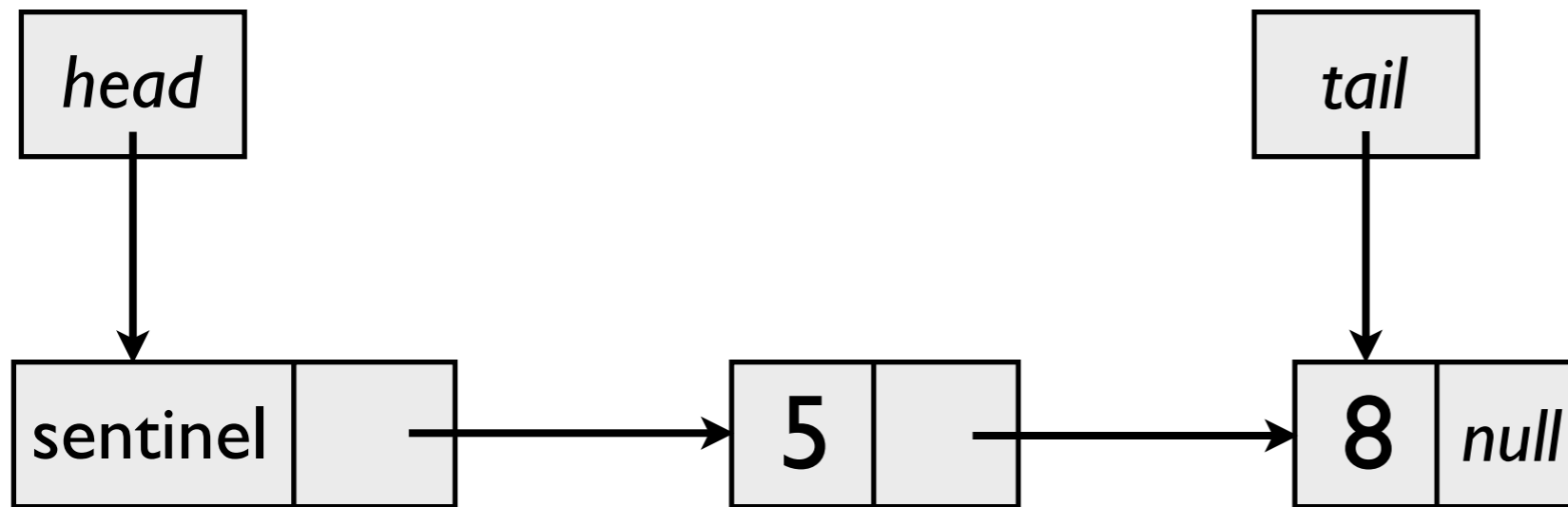
```
public void enq(int item) {
    Node node = new Node(item);
    while (true) {
        Node last = tail.get();
        Node next = last.next.get();
        if (last == tail.get()) {
            if (next == null) {
                if (last.next.compareAndSet(next, node)) {
                    tail.compareAndSet(last,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


Enqueue method

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

does it matter if this fails?

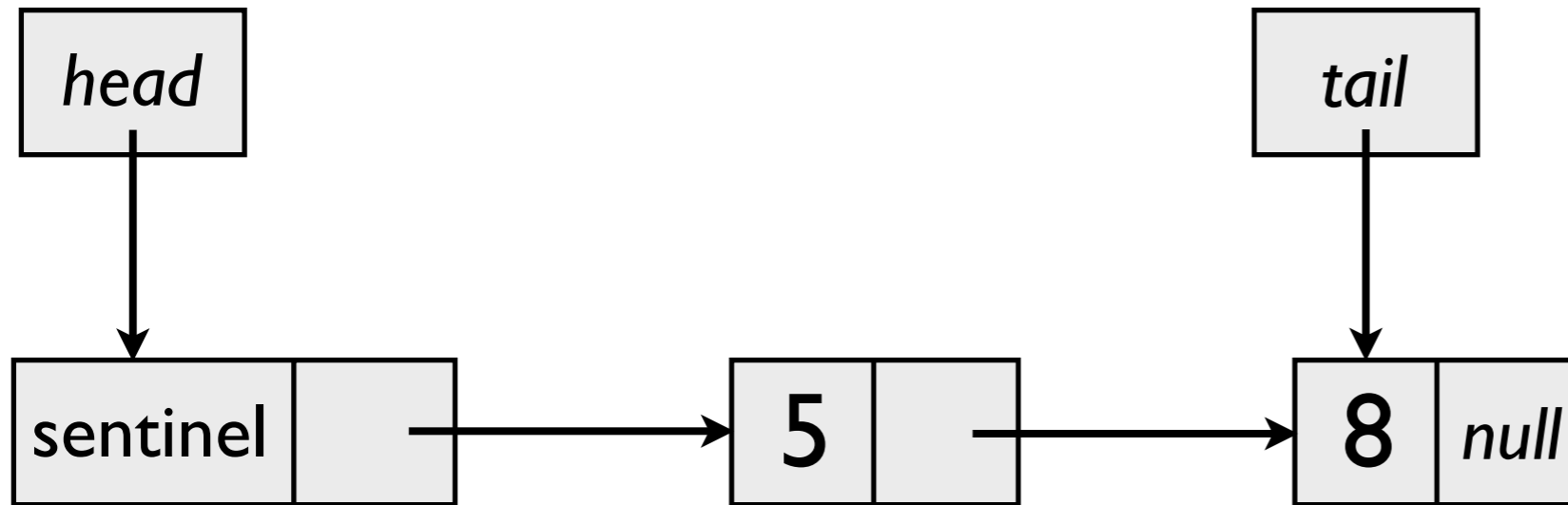
Enqueue example (two threads)



(1) Thread A: starts `enq(3)`, reads `tail/next`

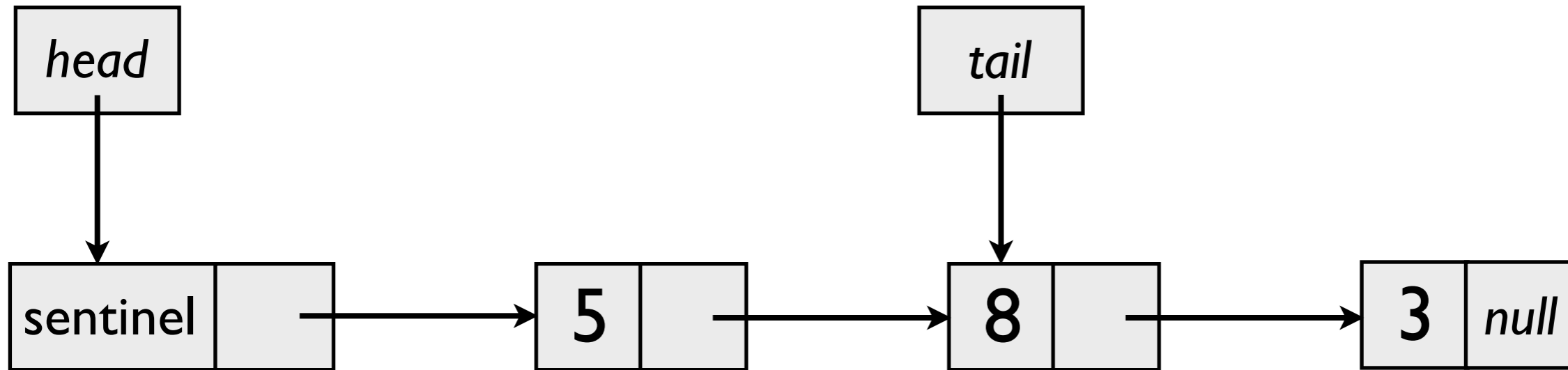
(2) Thread B: starts `enq(3)`, reads `tail/next`

Enqueue example (two threads)

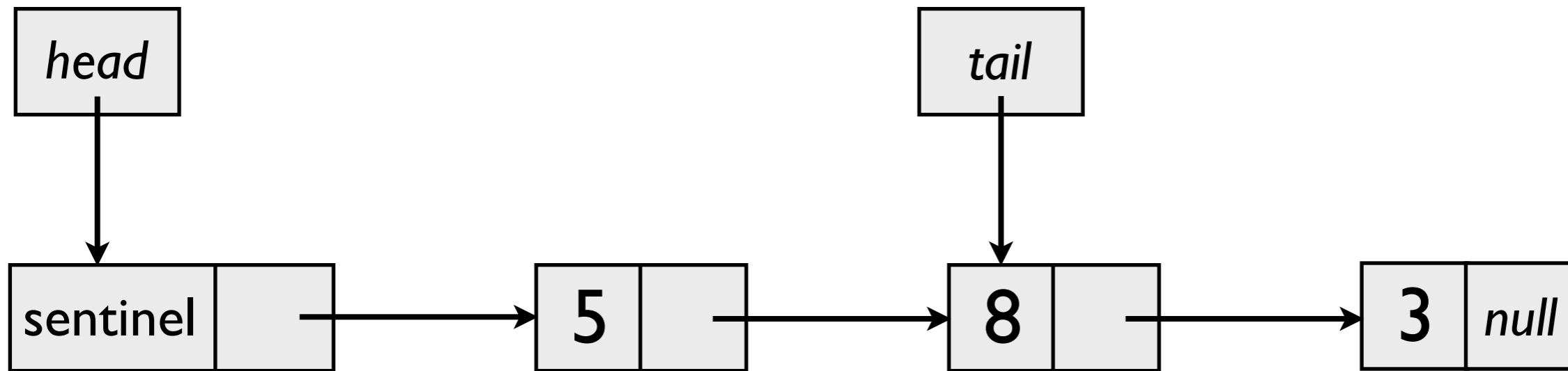


(3) Thread A: calls CAS on next

Enqueue example (two threads)



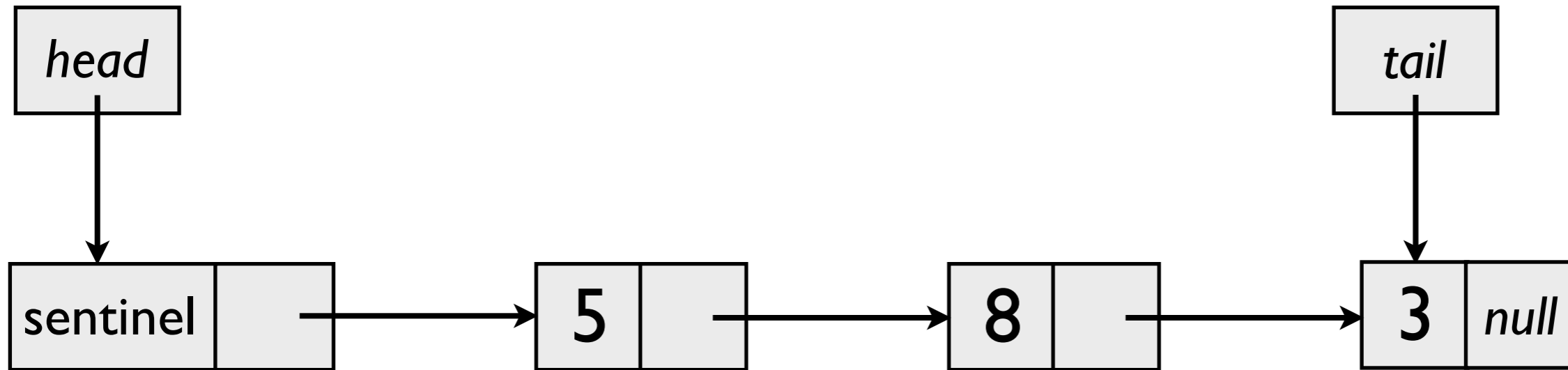
Enqueue example (two threads)



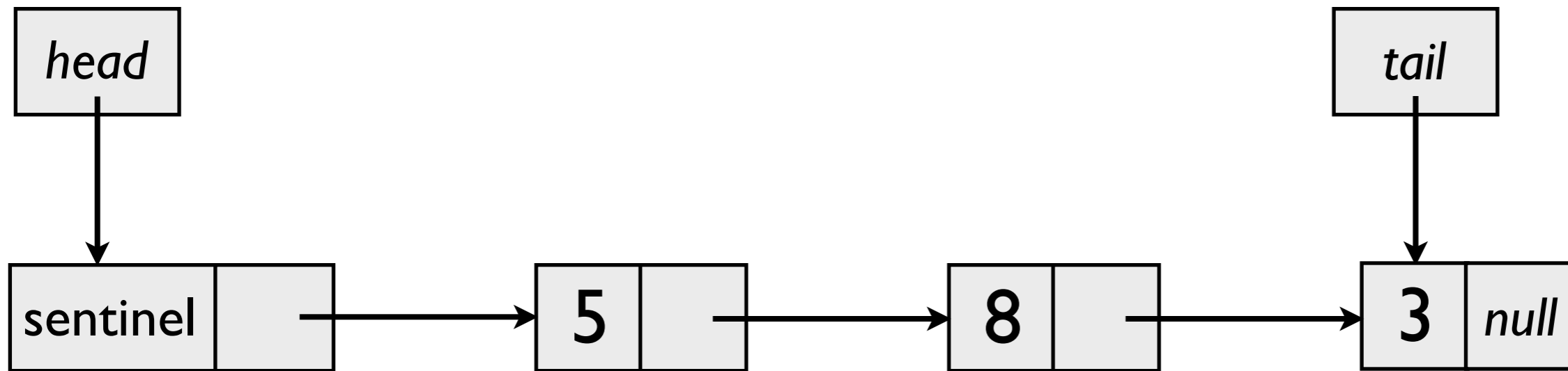
(4) Thread B: calls CAS
=> FAILS

(5) Thread B: calls CAS
to “repair” **tail**

Enqueue example (two threads)



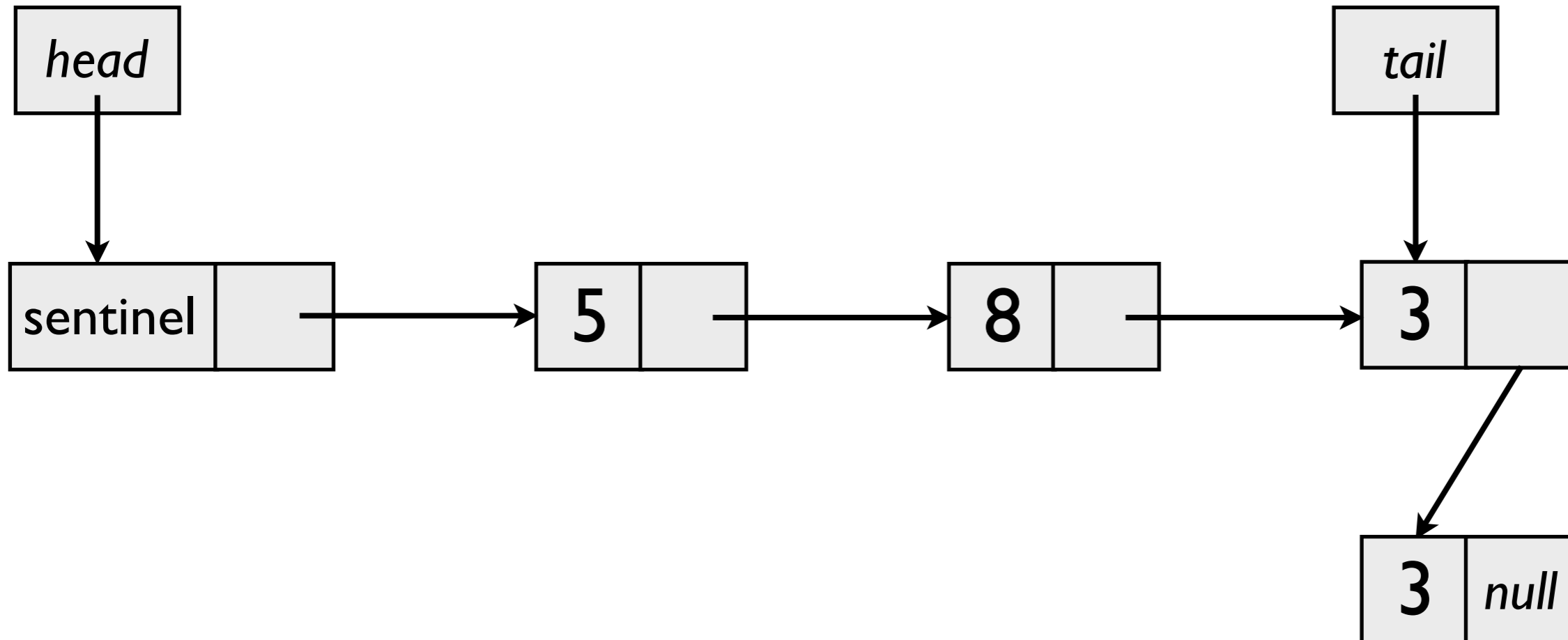
Enqueue example (two threads)



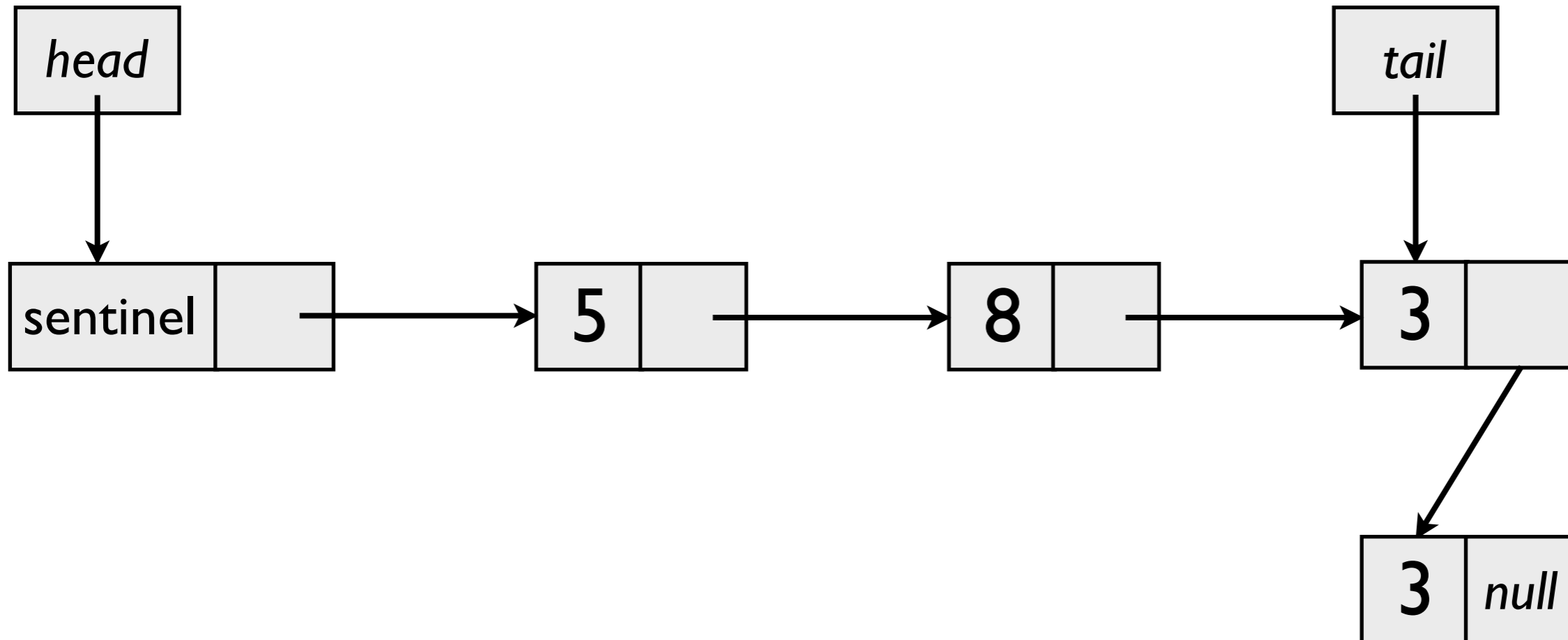
(6) Thread A: CAS to
update **tail**
=> **FAILS**, exits loop

(7) Thread B: restarts
loop, CAS on next

Enqueue example (two threads)

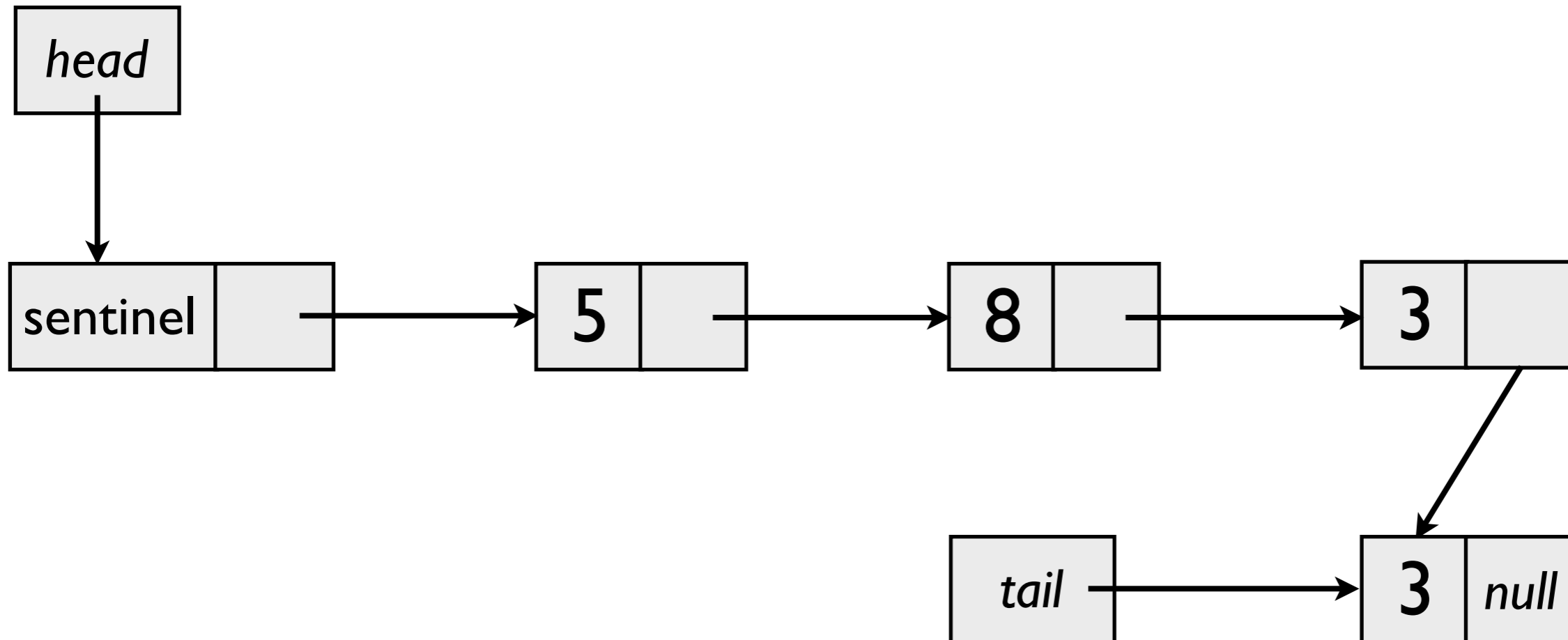


Enqueue example (two threads)



(8) Thread B: CAS
to update **tail**

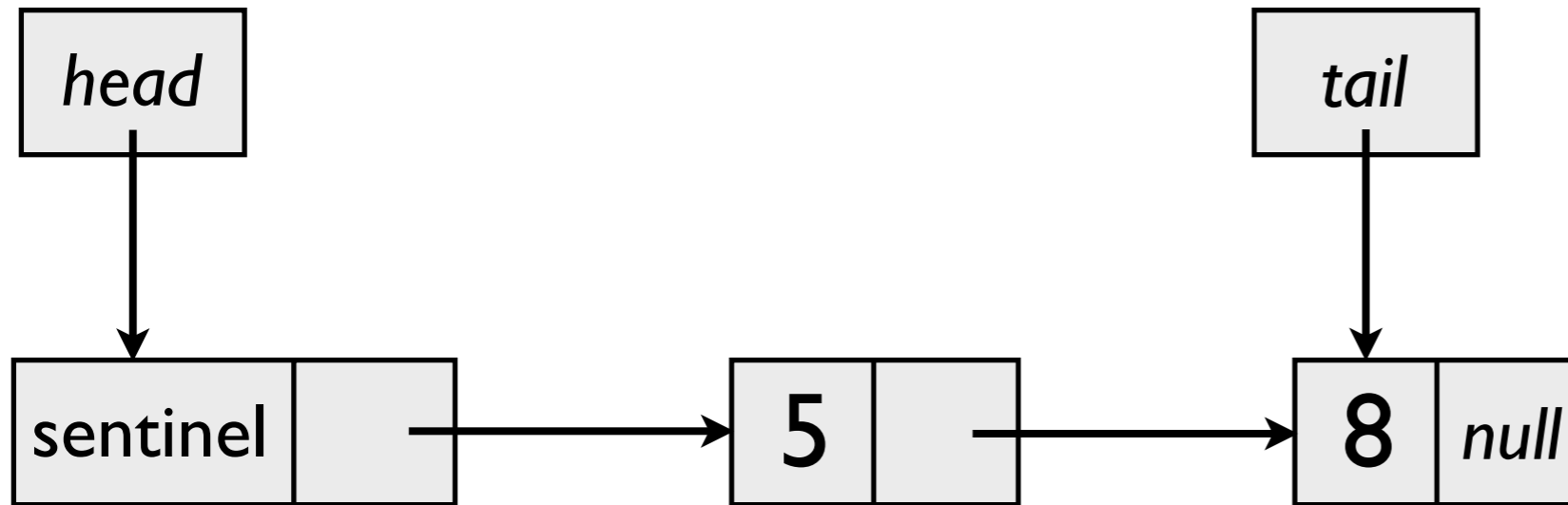
Enqueue example (two threads)



Dequeue

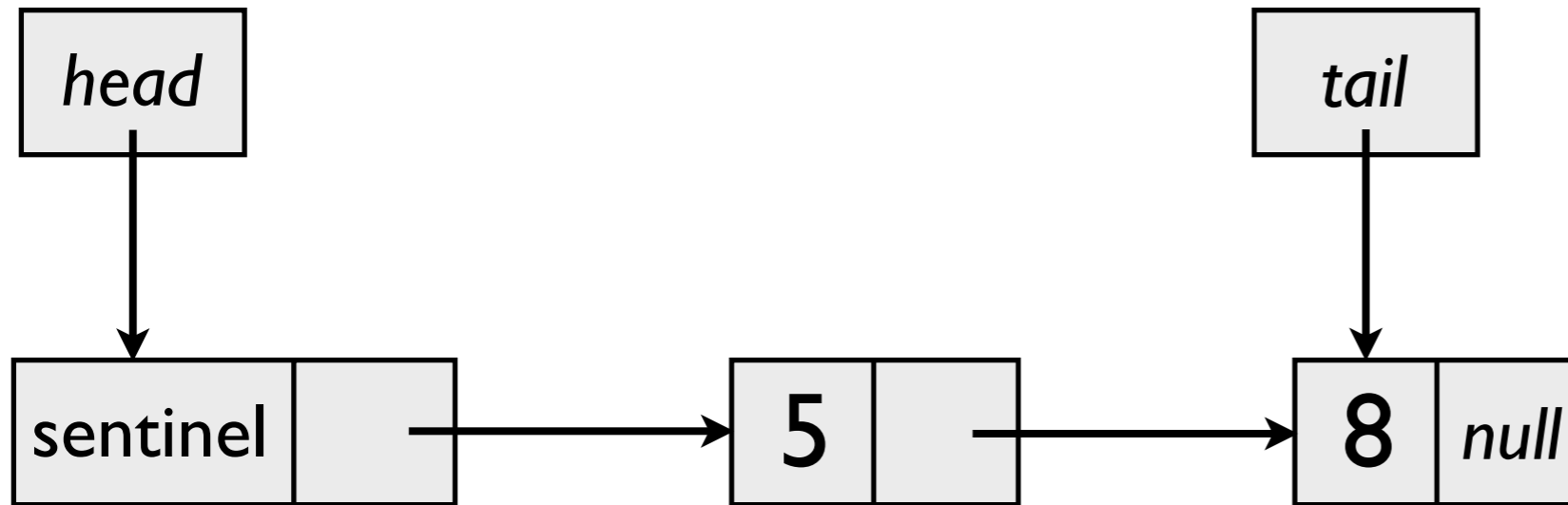
- the **dequeue** operation should return the integer stored in the Node at the front of the queue (or throw an exception if it is empty) and make the next Node the **head**
- must ensure that **head** does not “advance” past **tail**

Deque example (one thread)



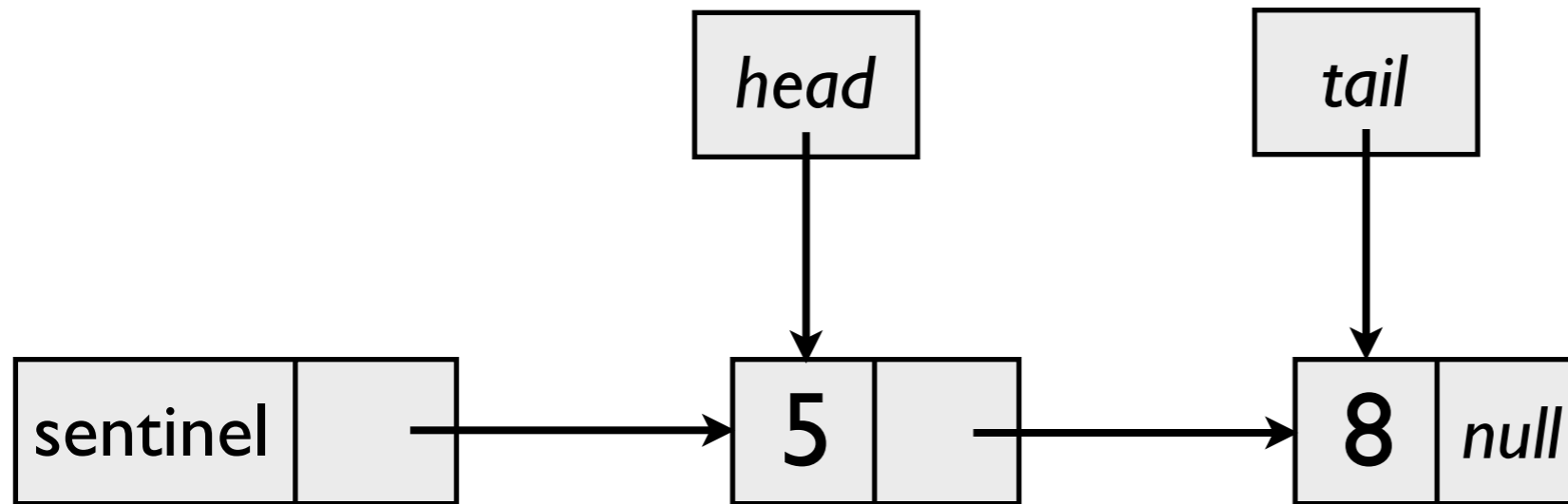
(I) Thread A: read value 5

Deque example (one thread)

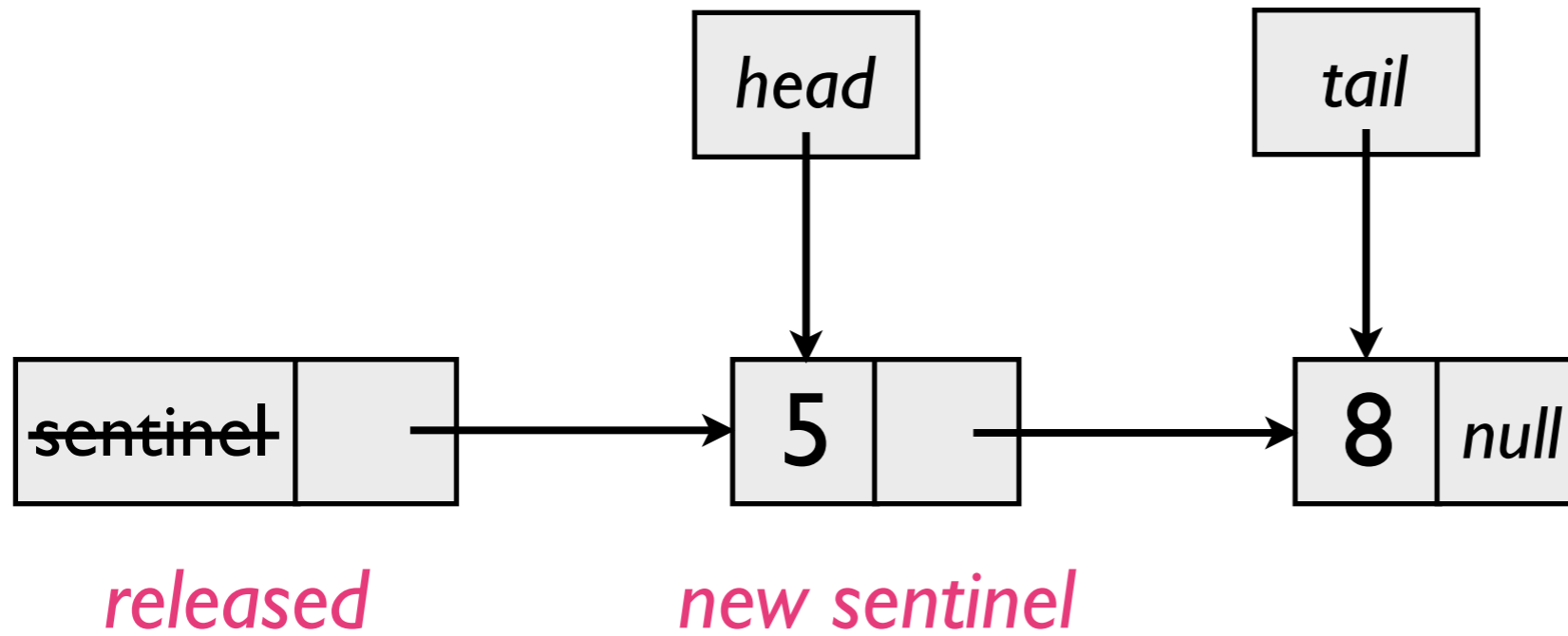


(2) Thread A: CAS **head**

Deque example (one thread)



Deque example (one thread)




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

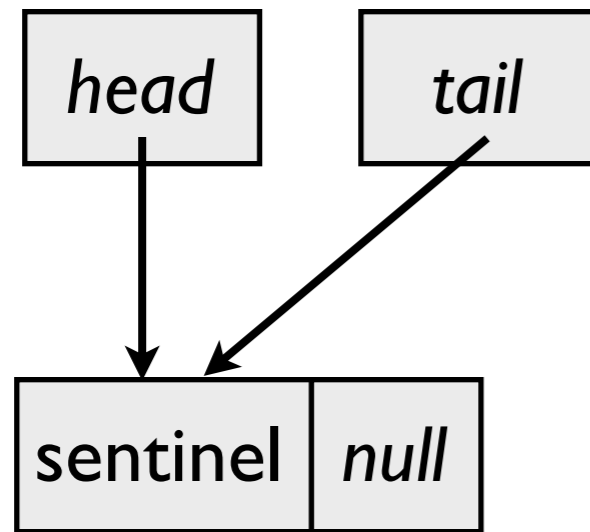


```
public int deq() throws EmptyException {
    while (true) {
        Node first = head.get();
        Node last = tail.get();
        Node next = first.next.get();
        if (first == head.get()) {
            if (first == last) {
                if (next == null) {
                    throw new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

what is this for?

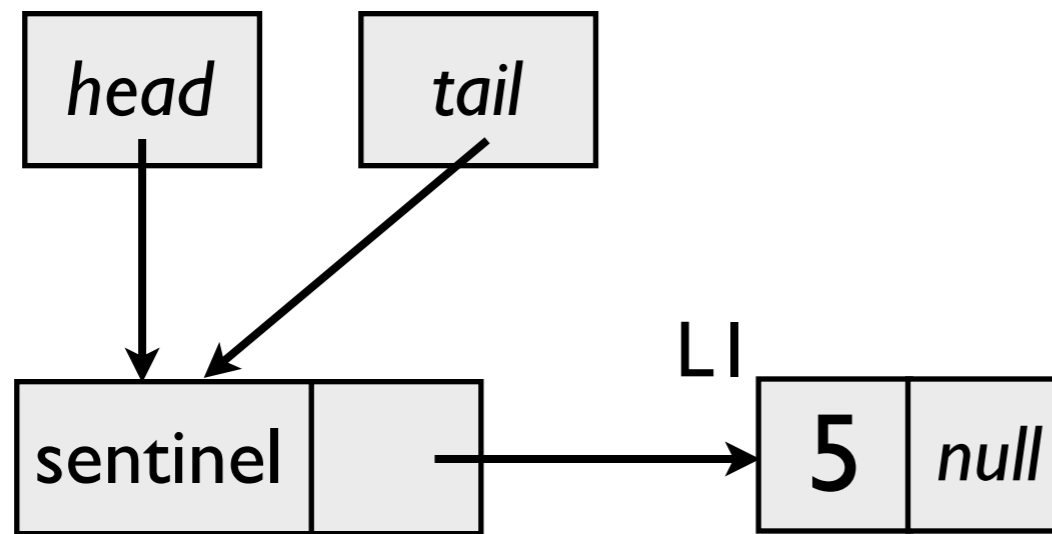


Dequeue example (two threads)

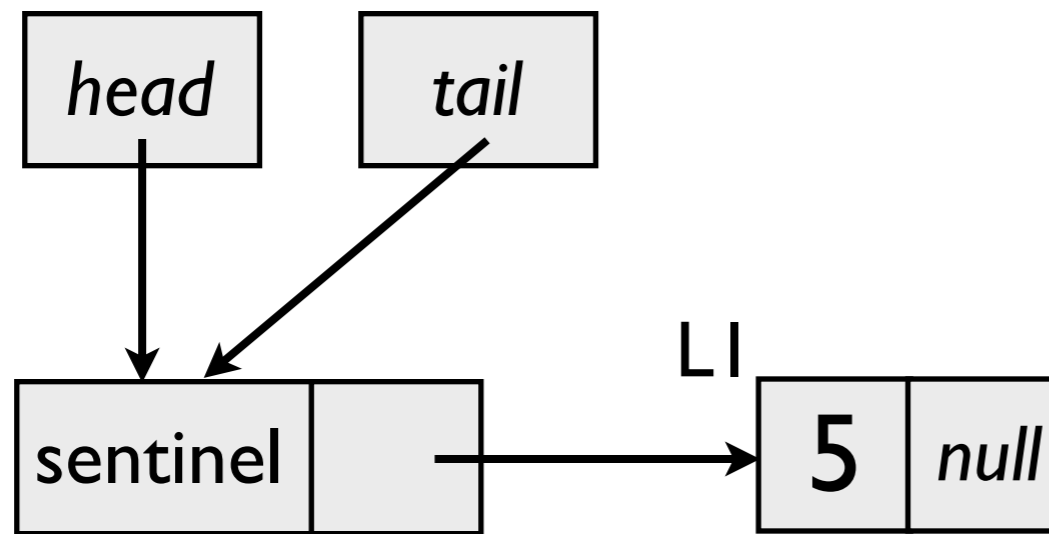


(I) Thread A: starts enqueueing LI

Dequeue example (two threads)

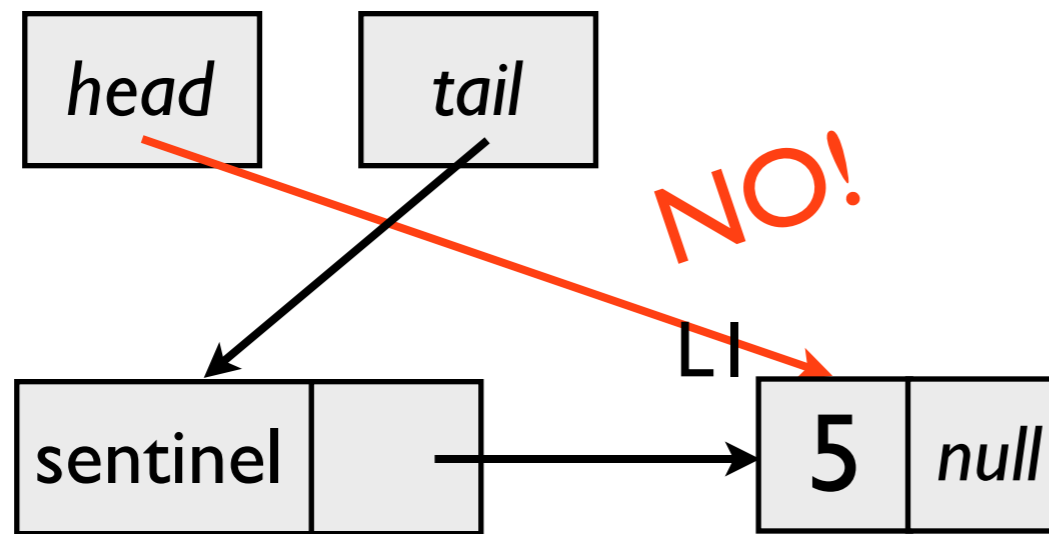


Dequeue example (two threads)



(2) Thread B: dequeue!

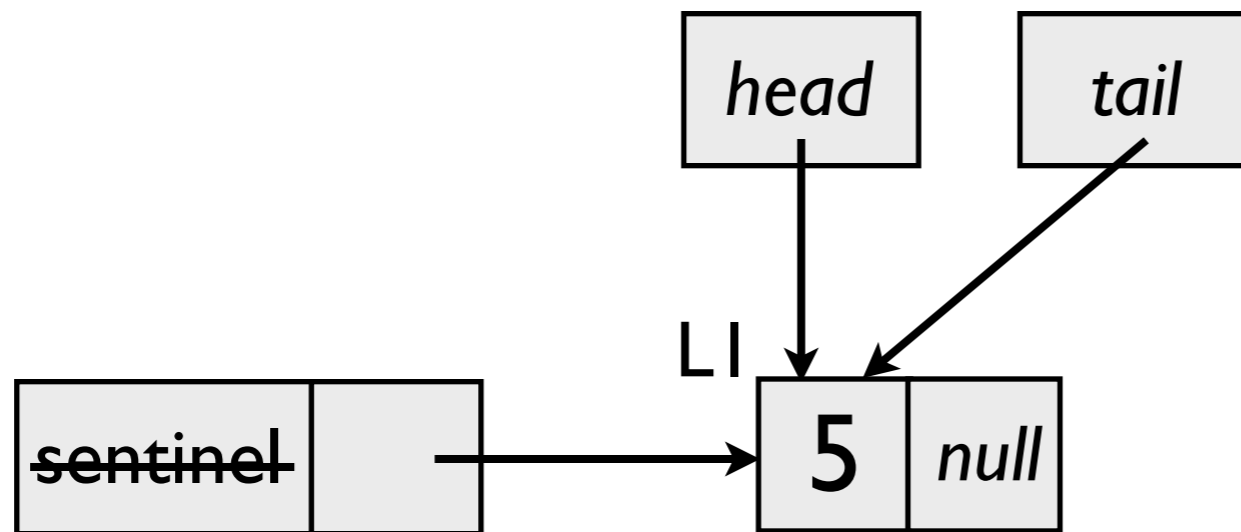
Dequeue example (two threads)



(2) Thread B: dequeue!

thread B must first redirect tail

Dequeue example (two threads)



(3) Thread A: reawakens, and exits loop since **tail** was already redirected

Lock-free programming: discussion

- good performance in some situations, avoiding many of the problems of locks
 - => *deadlock, priority inversion, ...*
- but **difficult to correctly implement** lock-free algorithms
 - => *e.g. the ABA problem*
 - => *can lead to unnatural structuring of algorithms*
 - => *not composable*
- focus on lock-free data structures (well-established algorithms and implementations available)

Next on the agenda

1. what's wrong with locks?



2. lock-free algorithms and data structures



3. transactional memory

What's wrong with CAS?

- the conventional atomic primitives of lock-free approaches operate on **one** memory location at a time
 - => algorithms can have an unnatural structure*
 - => e.g. concurrent FIFO queues: “half-finished” enq!*
- still a **lack of compositionality**
- very difficult to use in practice (and even harder to use correctly!)

Transactional memory: the future?

- **transactional memory (TM)** aims at simplifying atomic updates of **multiple** independent memory locations
 - => allows a group of instructions to execute in an atomic way*
 - => if properly implemented, does not deadlock or livelock*
- software (STM) and hardware (HTM) solutions
- inspiration: transactions in **database management systems**

The inspiration: database transactions

- a **database transaction** is a sequence of operations performed within a DBMS enjoying these properties:
 - => *Atomicity*: transactions appear to execute completely or not at all
 - => *Consistency*: transactions preserve consistency of the DB
 - => *Isolation*: other operations cannot access data modified by an incomplete transaction
 - => *Durability*: all committed transactions guaranteed to persist
- for TM, **atomicity** and **isolation** are most interesting

Software transactional memory (STM)

- research first focused on **software implementations**

=> starting with the work of Shavit & Touitou, 1995

=> based on earlier ideas of a multiprocessor *hardware architecture* to support lock-free programming (Herlihy & Moss, 1993)



- **idea:** allow code to be enclosed by an **atomic-block**

=> guarantee: executes *atomically* with respect to other atomic-blocks

Idea of Transactional Memory

```
public class TransactionalQueue<T> {
    private Node head;
    private Node tail;
    public TransactionalQueue() {
        Node sentinel = new Node(null);
        head = sentinel;
        tail = sentinel;
    }
    public void enq(T item) {
        atomic {
            Node node = new Node(item);
            tail.next = node;
            tail = node;
        }
    }
}
```

Idea of Transactional Memory

```
public void enq(T x) {  
    atomic {  
        if (count == items.length)  
            retry;  
        items[tail] = x;  
        if (++tail == items.length)  
            tail = 0;  
        ++count;  
    }  
}
```

*conditional synchronisation
via rollback*

Idea of Transactional Memory

```
atomic {  
  x = q0.deq();  
  q1.enq(x);  
}
```

composing atomic calls

Implementing STM

- a possible “**optimistic**” implementation scheme:

=> atomic-blocks run without locking; write to transaction log

=> onus placed on readers to check consistency

=> transaction can be committed, aborted, and/or re-executed

- numerous implementations of STM (mostly research prototypes; quality varies!)

=> http://en.wikipedia.org/wiki/Software_transactional_memory#Implementation_issues

- nice support in **concurrent Haskell**

=> <http://research.microsoft.com/pubs/67418/2005-ppopp-composable.pdf>

=> <https://www.fpcomplete.com/school/advanced-haskell/beautiful-concurrency/3-software-transactional-memory>

Concurrent Haskell

- starting point: “a purely declarative language is perfect for STM”
 - => type system explicitly separates computations with side-effects
 - => refine so that transactions perform memory effects but not irrevocable input/output effects
 - => most computation takes place in the pure functional world: never needs to be rolled back
- example: resource manager; `put r n` should return `n` units of resource to `r`

```
type Resource = TVar Int
putR :: Resource -> Int -> STM ()
putR r i = do { v <- readTVar r
               ; writeTVar r (v+i) }
```

Concurrent Haskell

- starting point: “a purely declarative language is perfect for STM”
 - => type system explicitly separates computations with side-effects
 - => refine so that transactions perform memory effects but not irrevocable input/output effects
 - => most computation takes place in the pure functional world: never needs to be rolled back
- example: resource manager; `put r n` should return `n` units of resource to `r`

```
type Resource = TVar Int
putR :: Resource -> Int -> STM ()
putR r i = do { v <- readTVar r
               ; writeTVar r (v+i) }
```

returns STM actions



Concurrent Haskell

- starting point: “a purely declarative language is perfect for STM”
 - => type system explicitly separates computations with side-effects
 - => refine so that transactions perform memory effects but not irrevocable input/output effects
 - => most computation takes place in the pure functional world: never needs to be rolled back
- example: resource manager; `put r n` should return `n` units of resource to `r`

```
type Resource = TVar Int
putR :: Resource -> Int -> STM ()
putR r i = do { v <- readTVar r
               ; writeTVar r (v+i) }
```

```
main = do { ...; atomic (putR r 3); ... }
```

Concurrent Haskell

- starting point: “a purely declarative language is perfect for STM”
 - => type system explicitly separates computations with side-effects
 - => refine so that transactions perform memory effects but not irrevocable input/output effects
 - => most computation takes place in the pure functional world: never needs to be rolled back
- example: resource manager; `put r n` should return `n` units of resource to `r`

```
type Resource = TVar Int
putR :: Resource -> Int -> STM ()
putR r i = do { v <- readTVar r
               ; writeTVar r (v+i) }
```

```
main = do { ...; atomic (putR r 3); ... }
```

compose actions!

*returns an I/O action
that runs the transaction
atomically wrt to all
other memory
transactions*

Hardware Transactional Memory (HTM)

- extensions to the processor's **instruction set**
- embed transactional memory into **existing cache design**
- going mainstream: **Intel Haswell** (2013) architecture includes support

Next on the agenda

1. what's wrong with locks?



2. lock-free algorithms and data structures



3. transactional memory



Conclusion

- lock-free programming can lead to good performance
- difficult to get right
 - ⇒ *CAS can modify only one memory location at a time*
 - ⇒ *ABA problem*
- well established lock-free concurrent data structures are available
- STM/HTM may yet provide a simpler model for concurrent programming
- *next lecture: correctness conditions (linearizability)*