Lecture 21: Linearizability II

COSC 273: Parallel and Distributed Computing

Spring 2023

Announcements

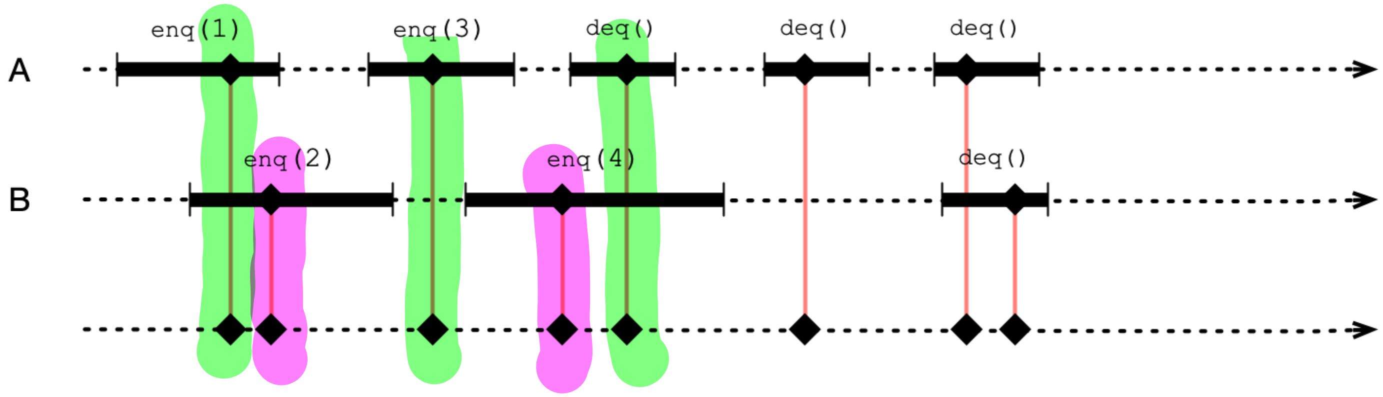
1. Quiz this Friday
    - sequential consistency
    - linearizability
    - recall **stack** operations
        - `push(x)`, `pop()`

Last Time: Linearizability

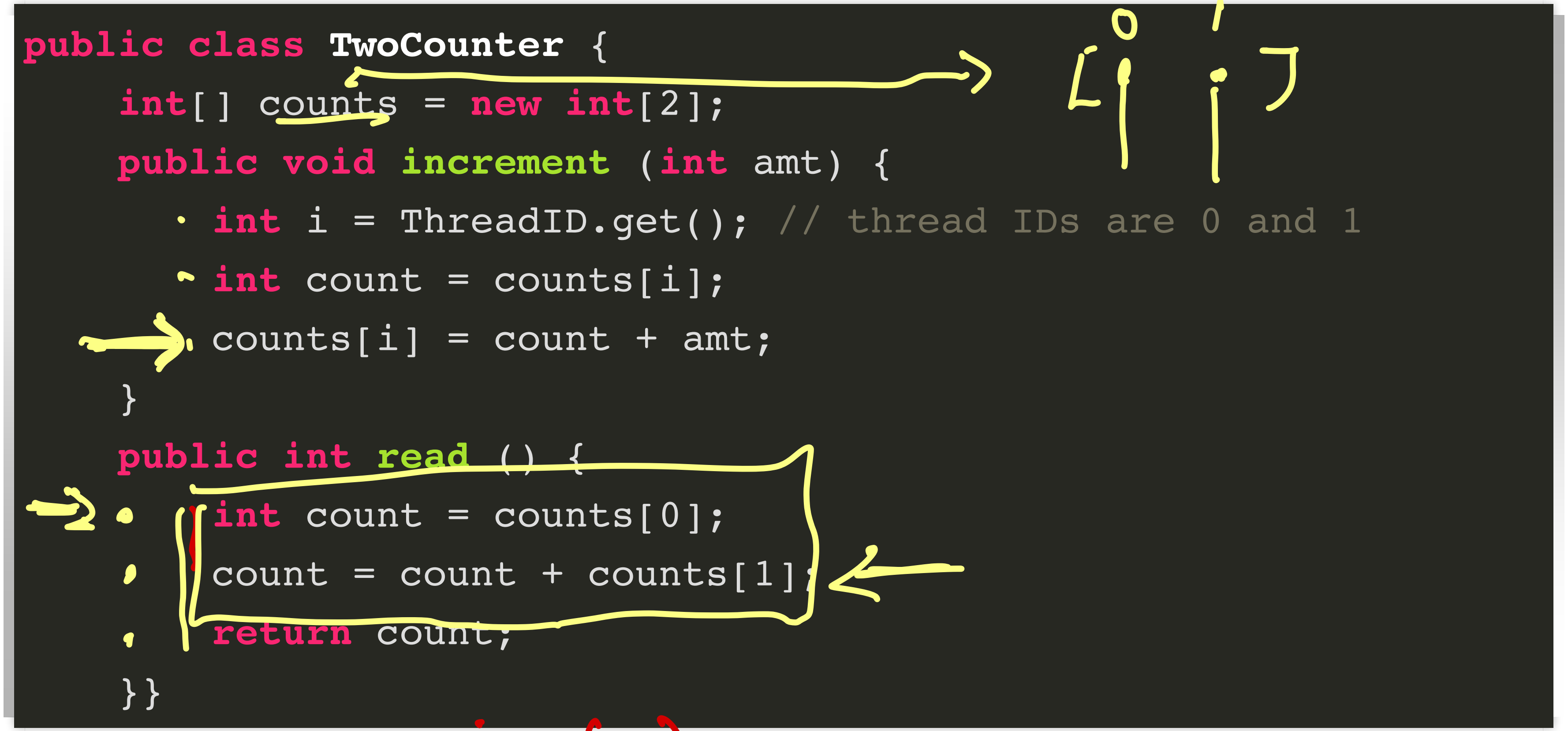
An execution of a shared object is **linearizable** if:

- exists a *linearization point* in each method call such that execution is consistent with sequential execution where method calls occur in order of corresponding linearization points

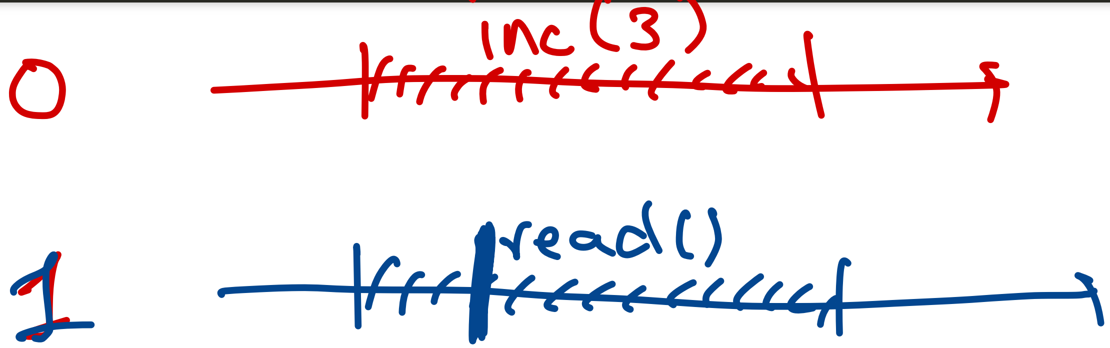
An implementation of an object is linearizable if every execution is linearizable.



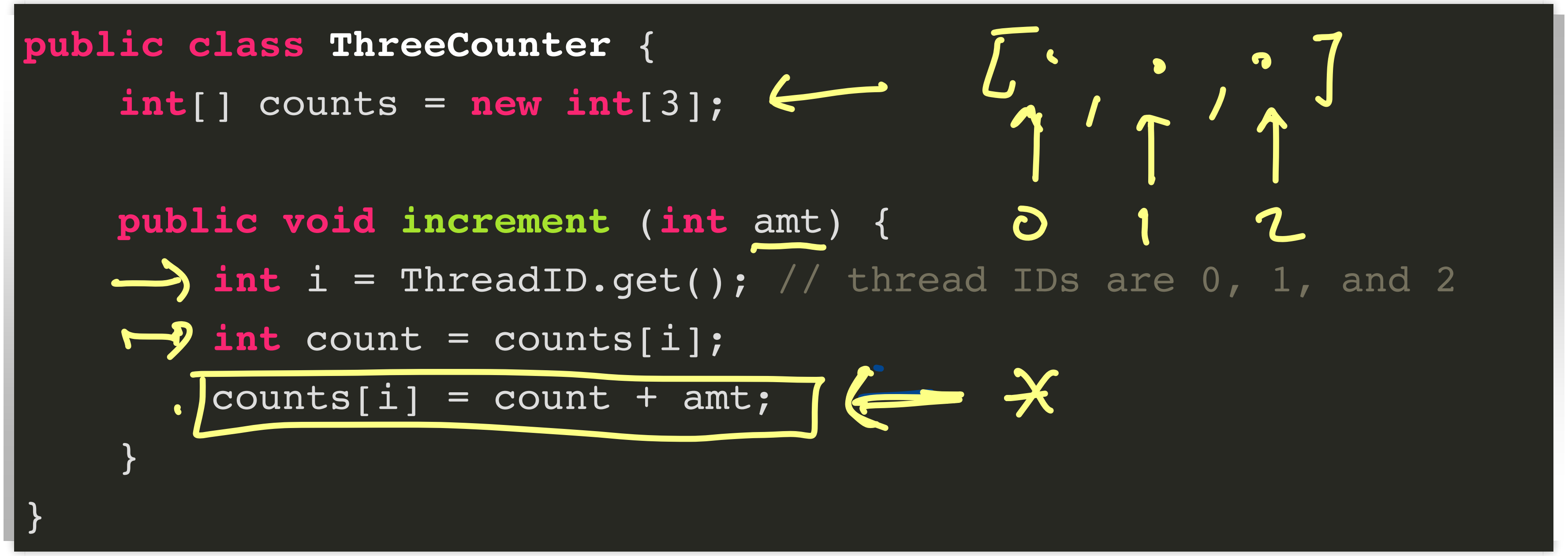
Linearizable TwoCounter



```
public class TwoCounter {
    int[] counts = new int[2];
    public void increment (int amt) {
        int i = ThreadID.get(); // thread IDs are 0 and 1
        int count = counts[i];
        counts[i] = count + amt;
    }
    public int read () {
        int count = counts[0];
        count = count + counts[1];
        return count;
    }}
```



ThreeCounter Example

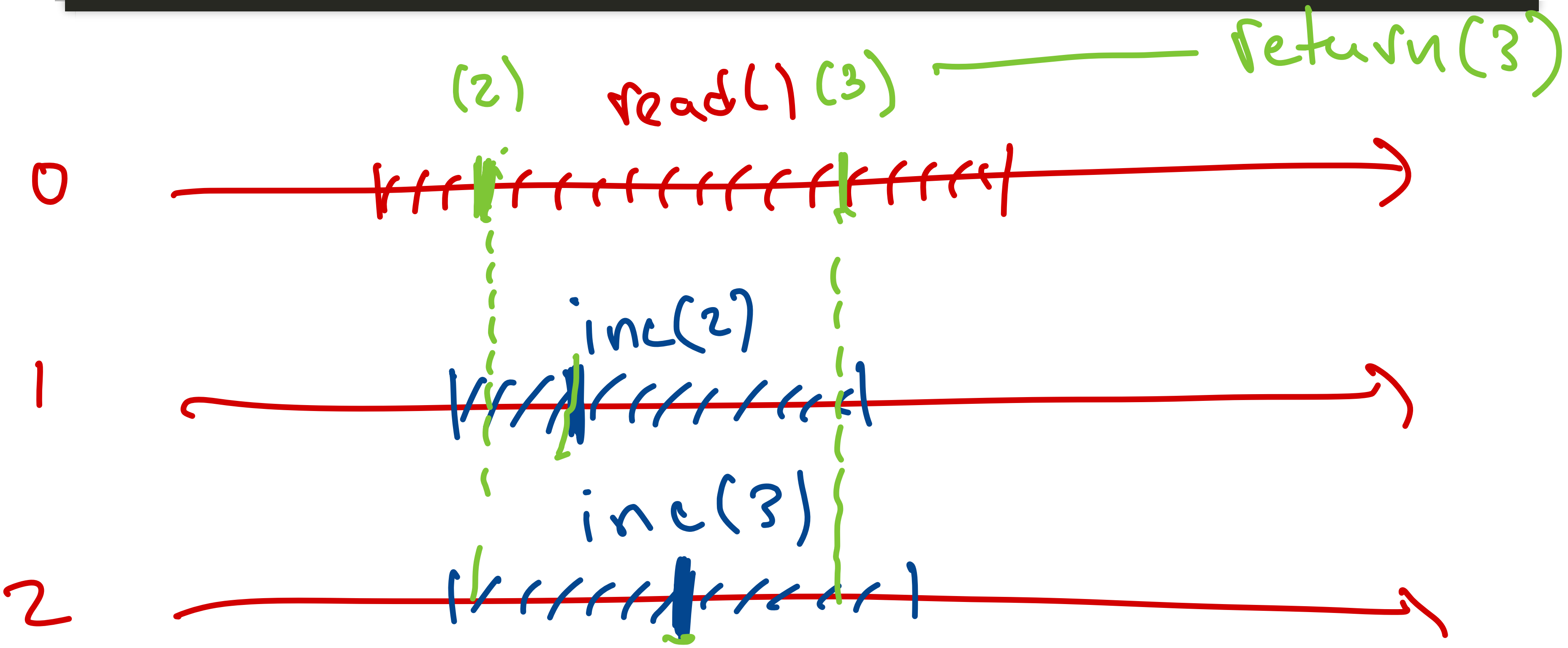


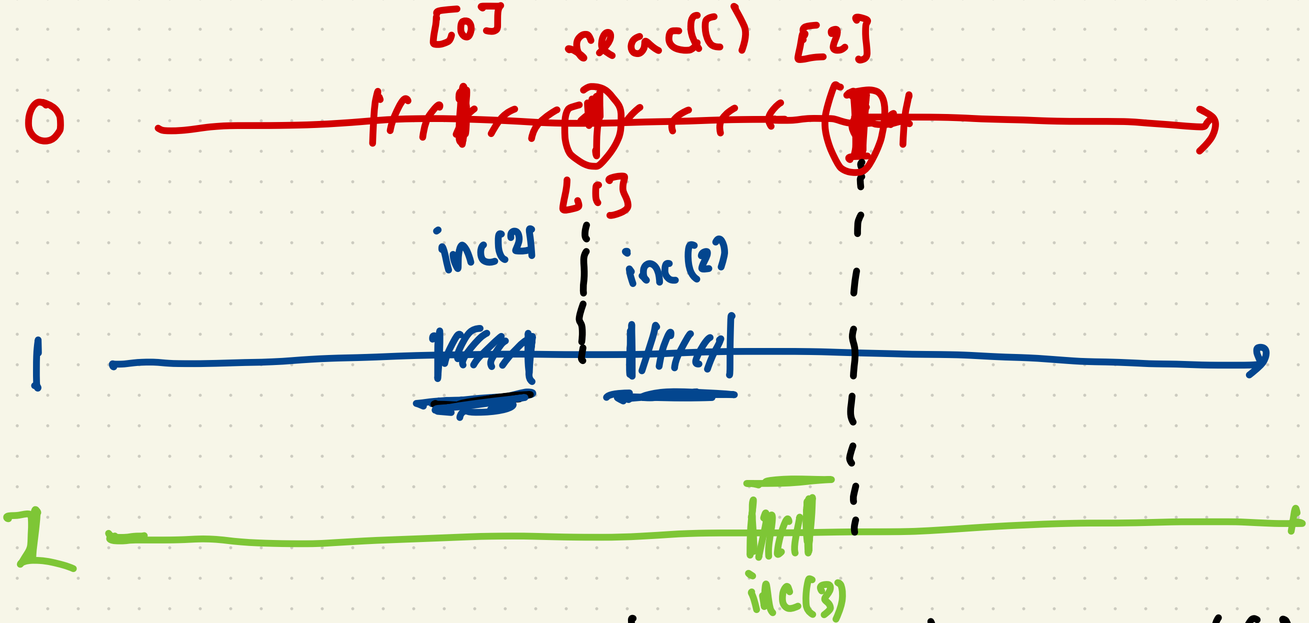
```
public class ThreeCounter {
    int[] counts = new int[3];

    public void increment (int amt) {
        int i = ThreadID.get(); // thread IDs are 0, 1, and 2
        int count = counts[i];
        counts[i] = count + amt;
    }
}
```

A read Method

(1)
(2)
(3)

```
public int read () {
    int count = counts[0];
    count = count + counts[1];
    count = count + counts[2];
    return count;
}
```





What value is returned by read()?

→ 5

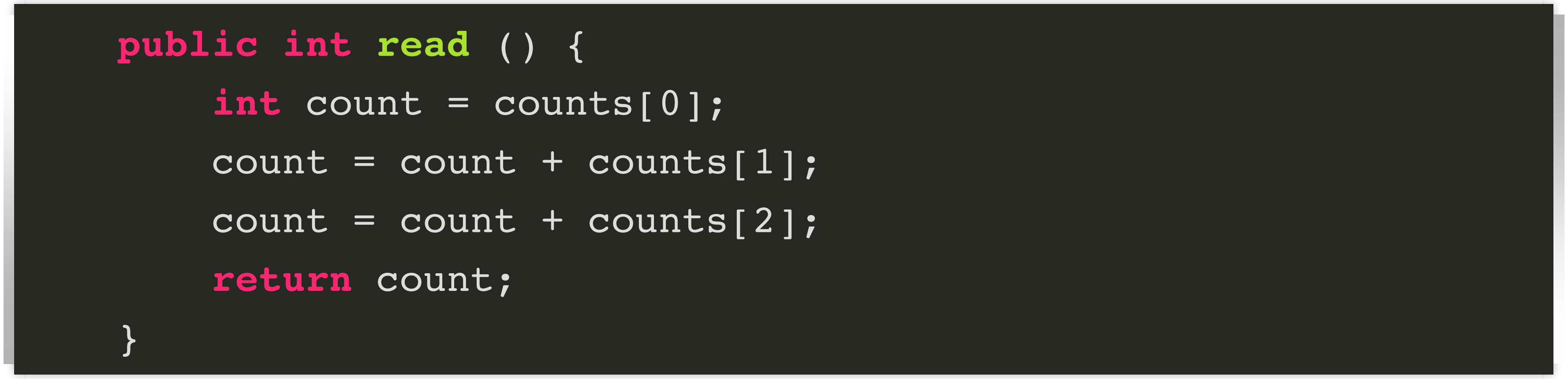
What possible values from linearizable exec?

0, 2, 4, 7

Is ThreeCounter Linearizable?

Nope.

Writing Between the Lines



```
public int read () {
    int count = counts[0];
    count = count + counts[1];
    count = count + counts[2];
    return count;
}
```

Sequentially Consistency

Questions.

1. Is the previous execution sequentially consistent?
2. Is `ThreeCounter` sequentially consistent?

A Queue Again

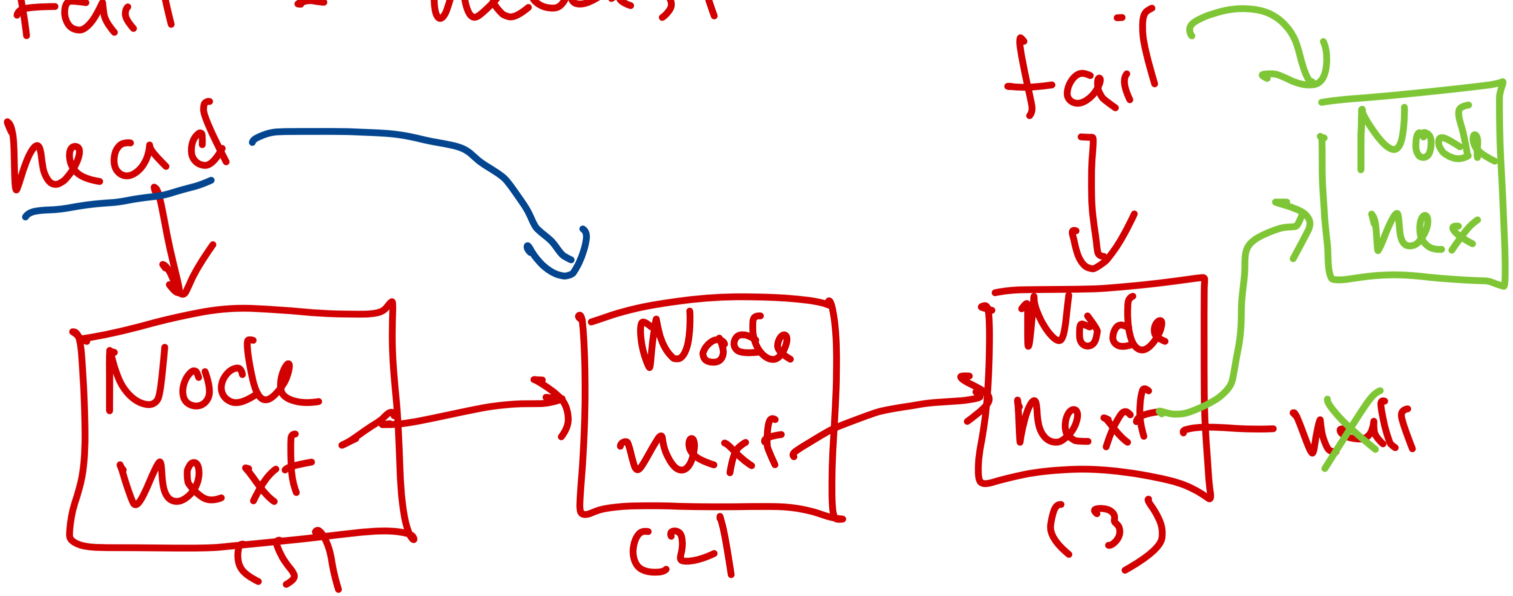
Question. How to implement a (non-concurrent) queue with a linked list?

enq ← add an item

deq ← remove "oldest" item.

(doubly) linked list

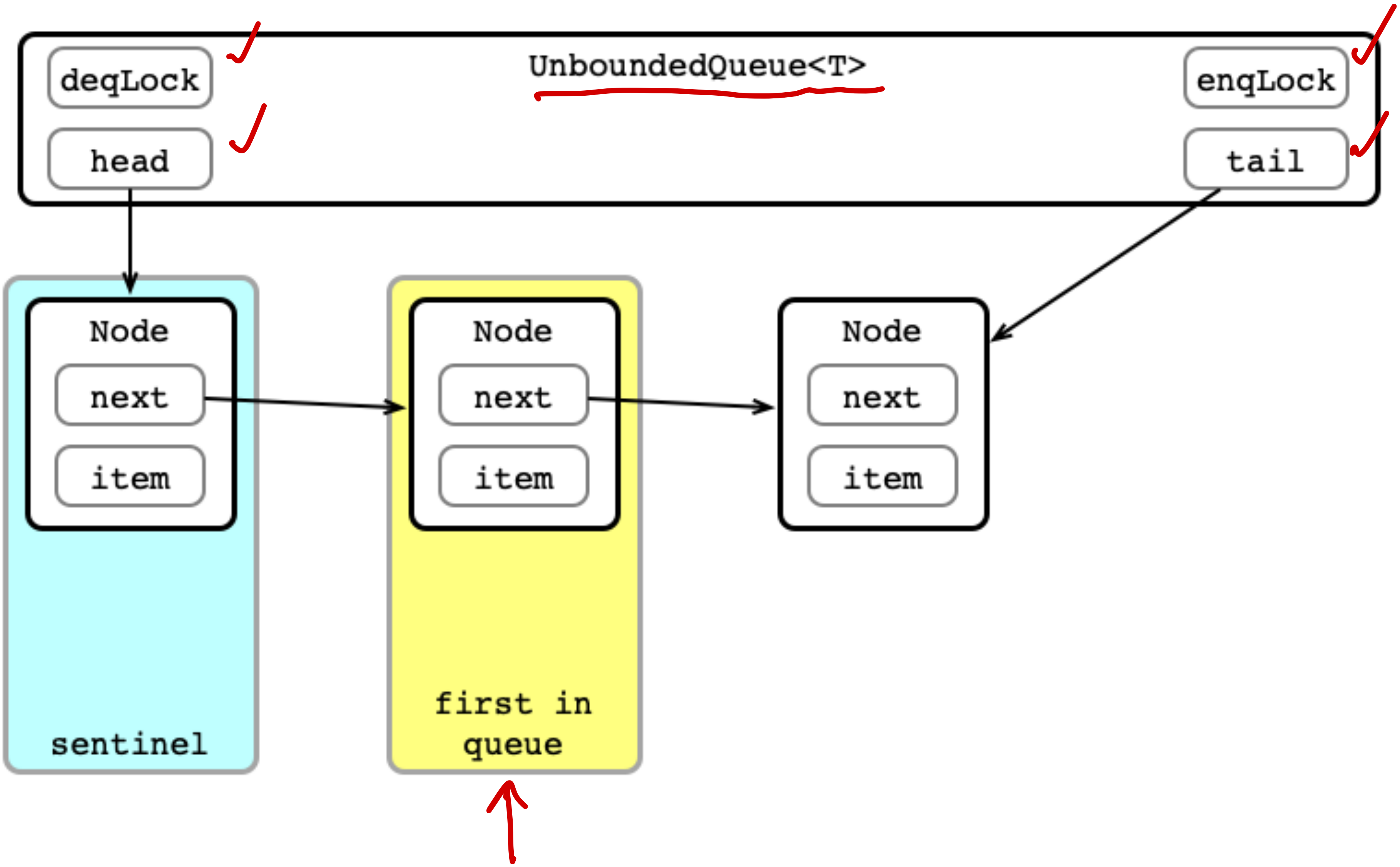
- head = oldest item
- tail = newest



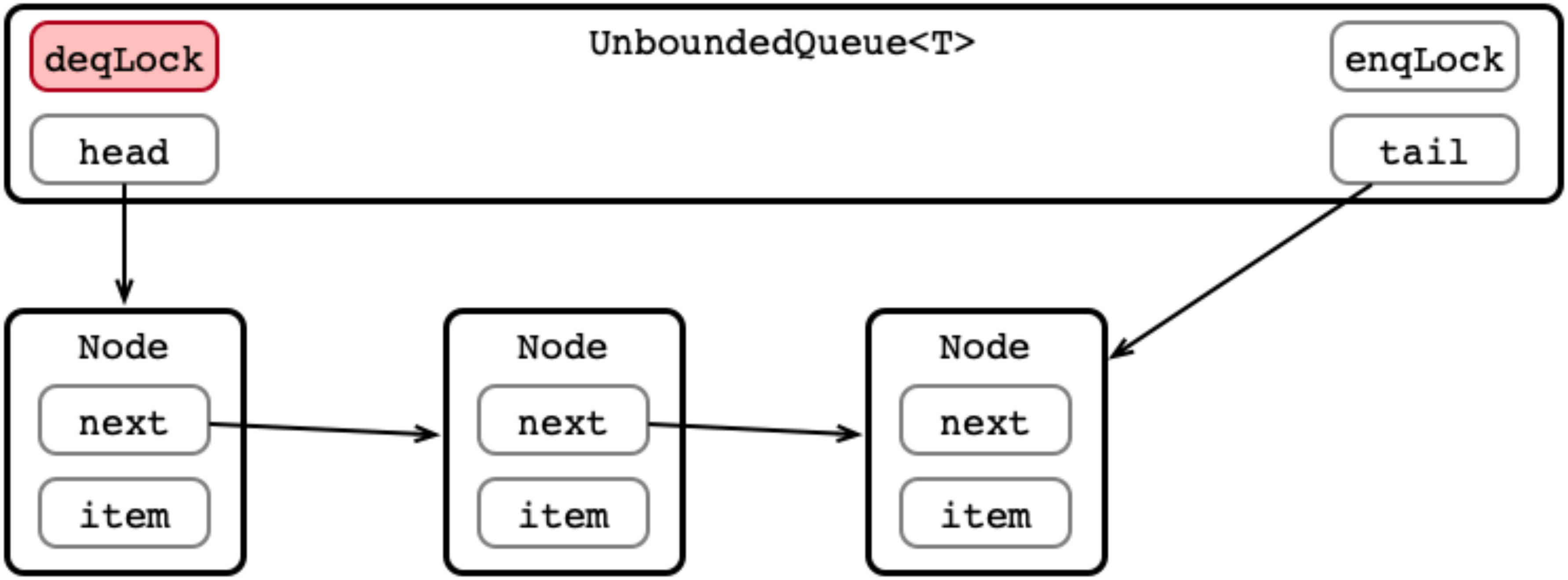
A Concurrent Queue

- Use linked list implementation of queue
- Store:
  - Node head sentinal
    - deq returns head.next value (if any), updates head
  - Node tail
    - enq updates tail.next, updates tail
- Locks:
  - enqLock locks enq operation
  - deqLock locks deq operation
  - individual Nodes are *not* locked

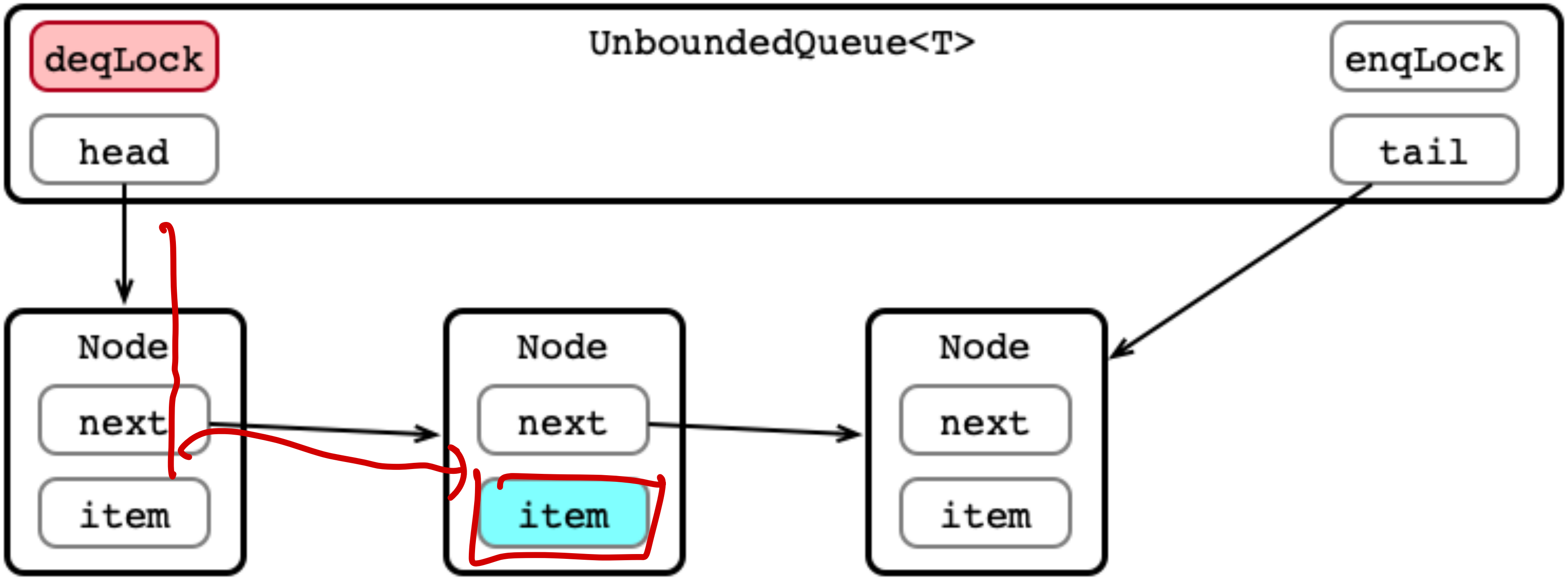
Unbounded Queue in Pictures



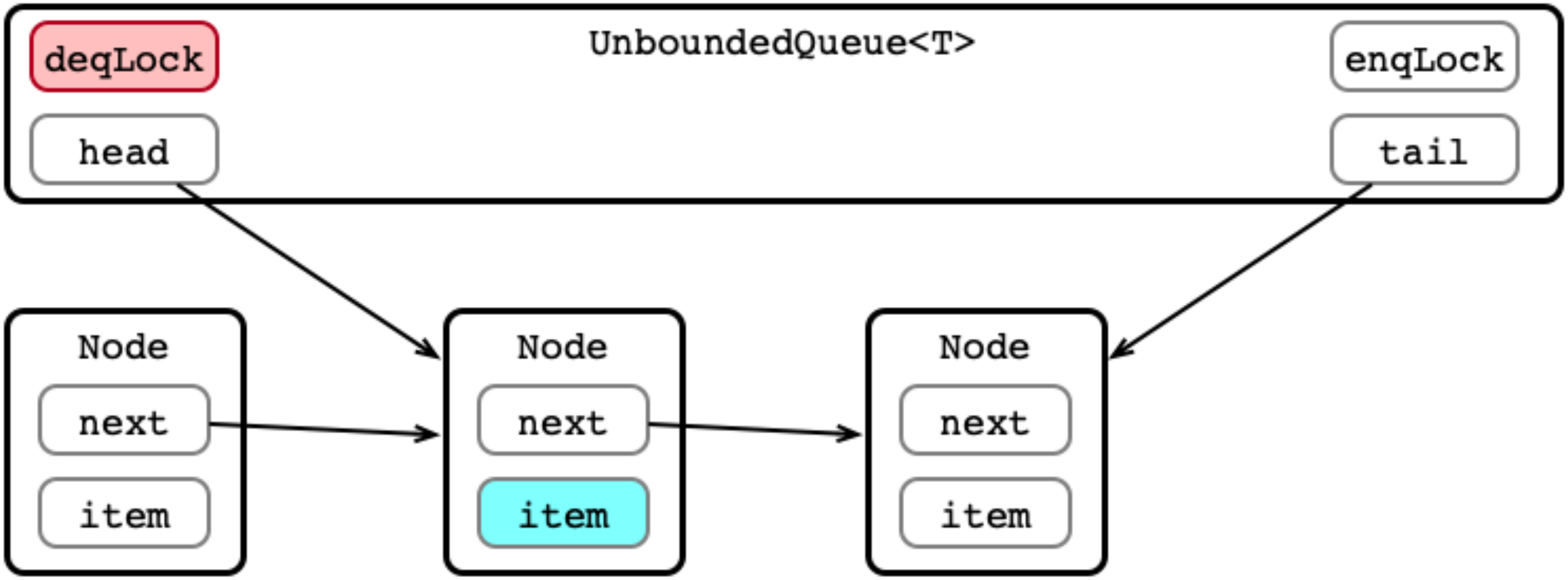
Dequeue 1: Aquire deqLock



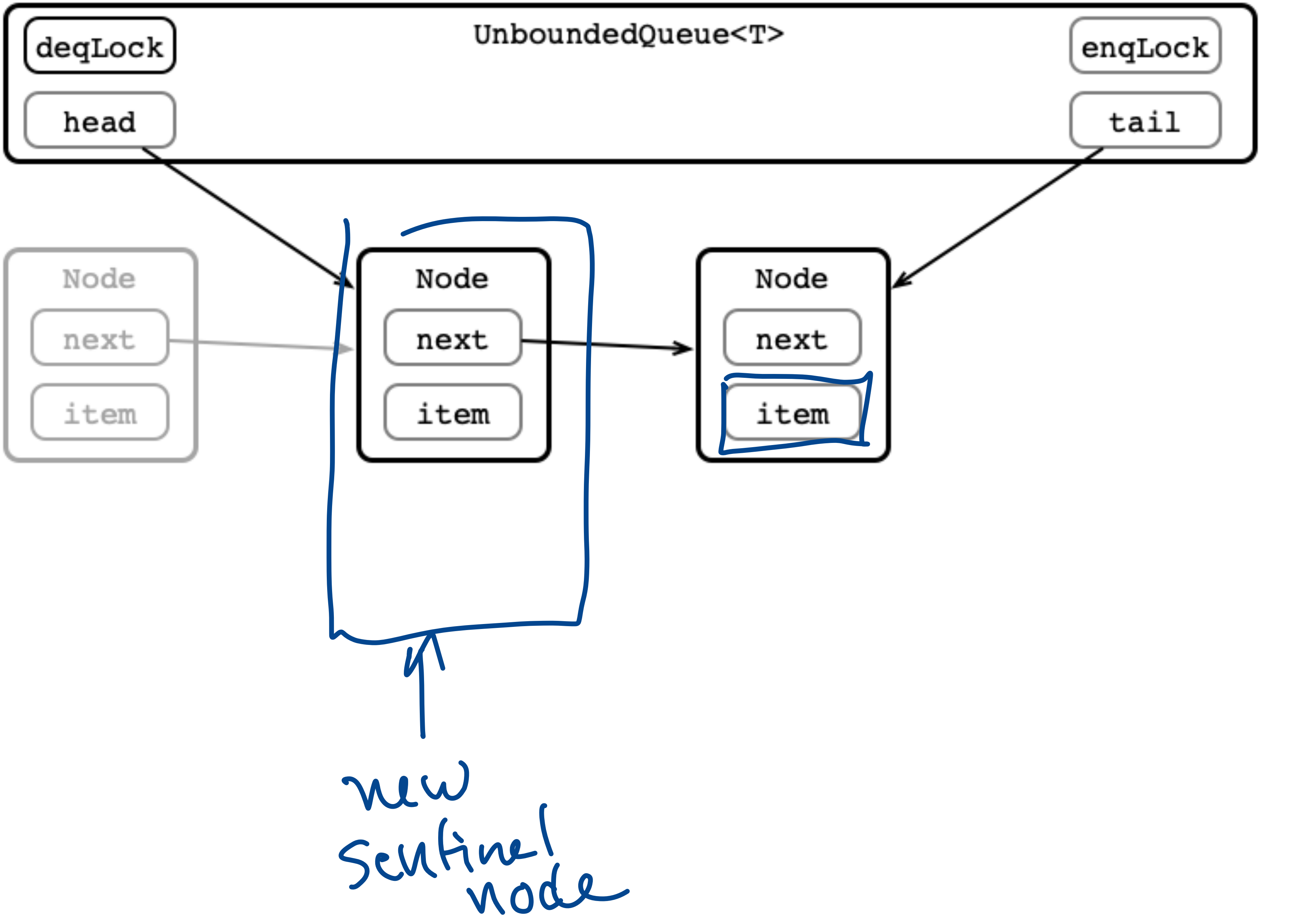
Dequeue 2: Get Element (or Exception)



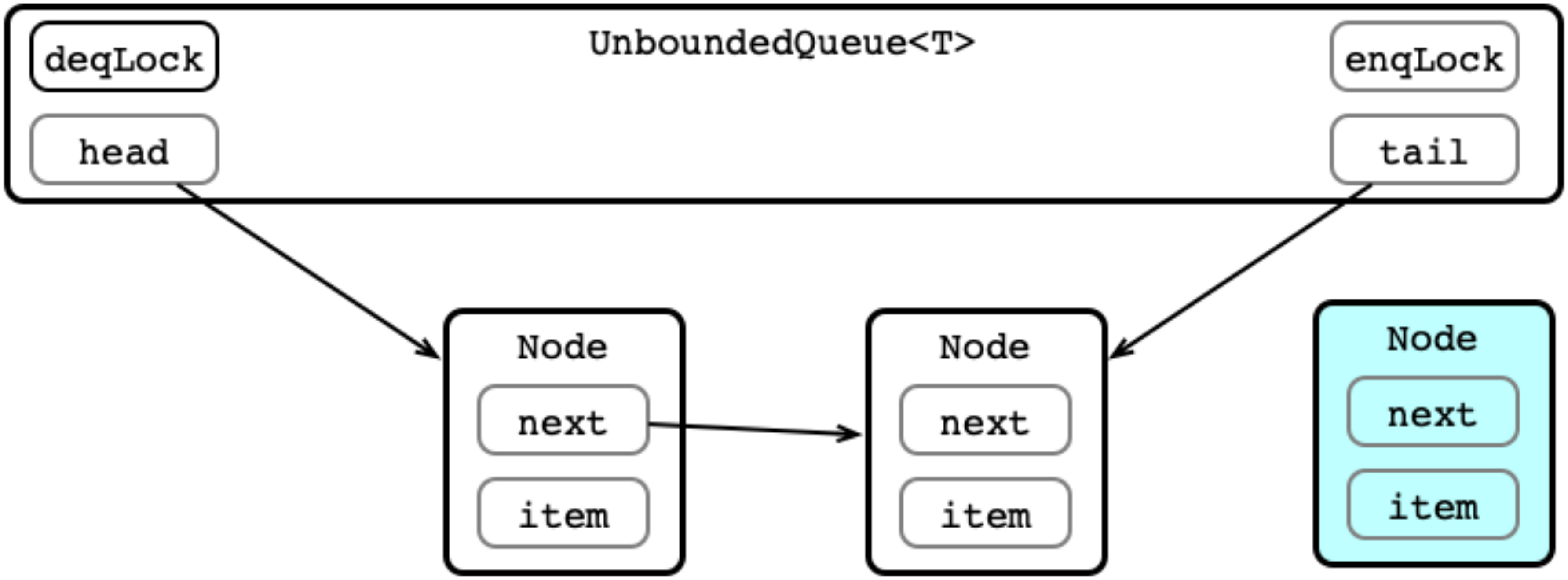
Dequeue 3: Update head



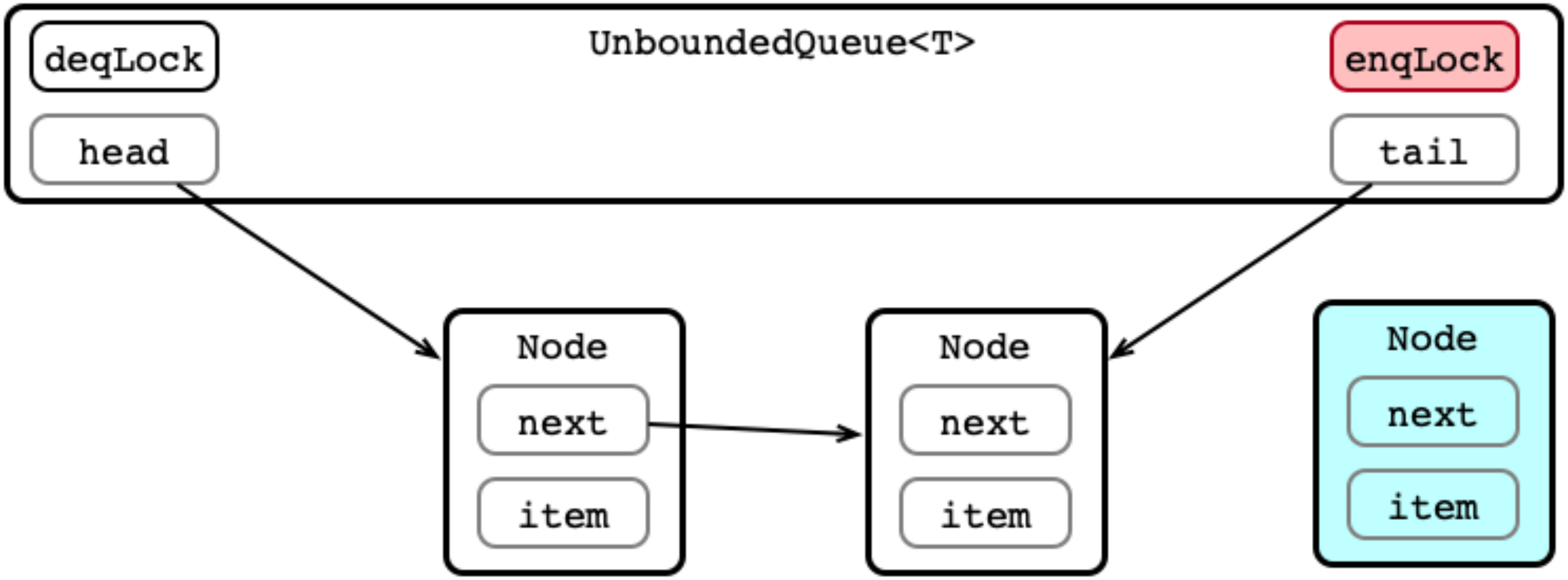
Dequeue 4: Release Lock



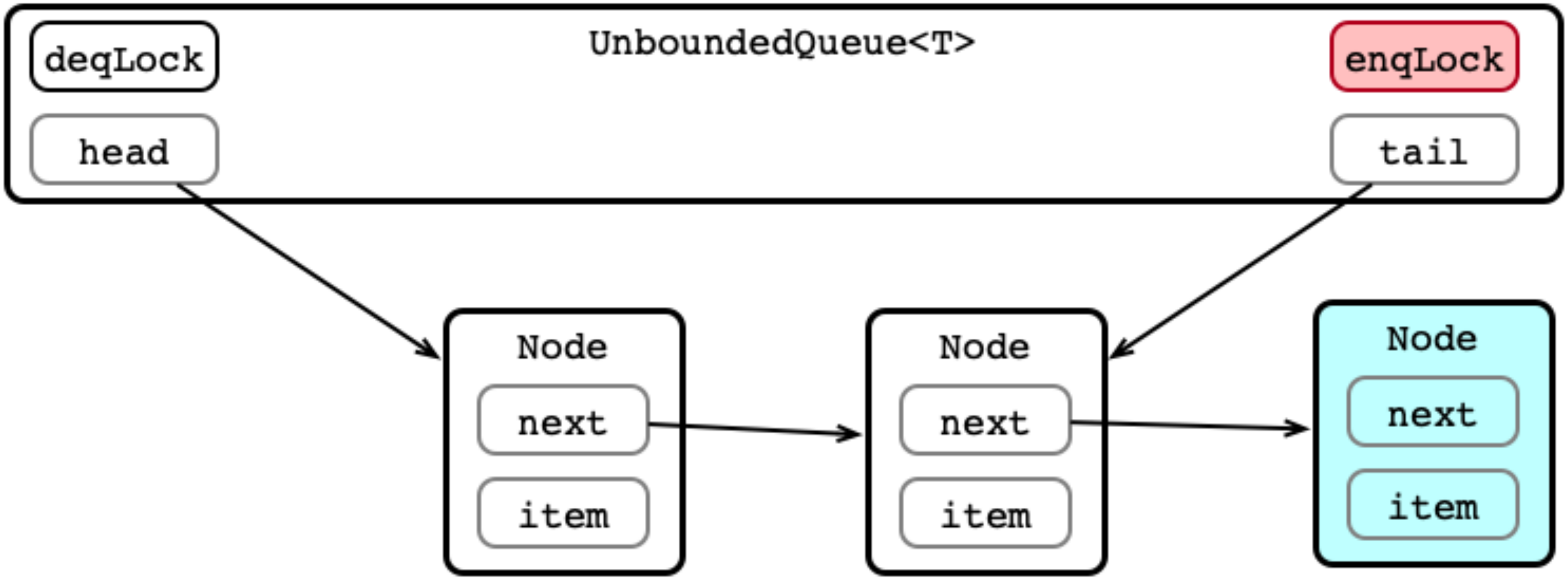
Enqueue 1: Make Node



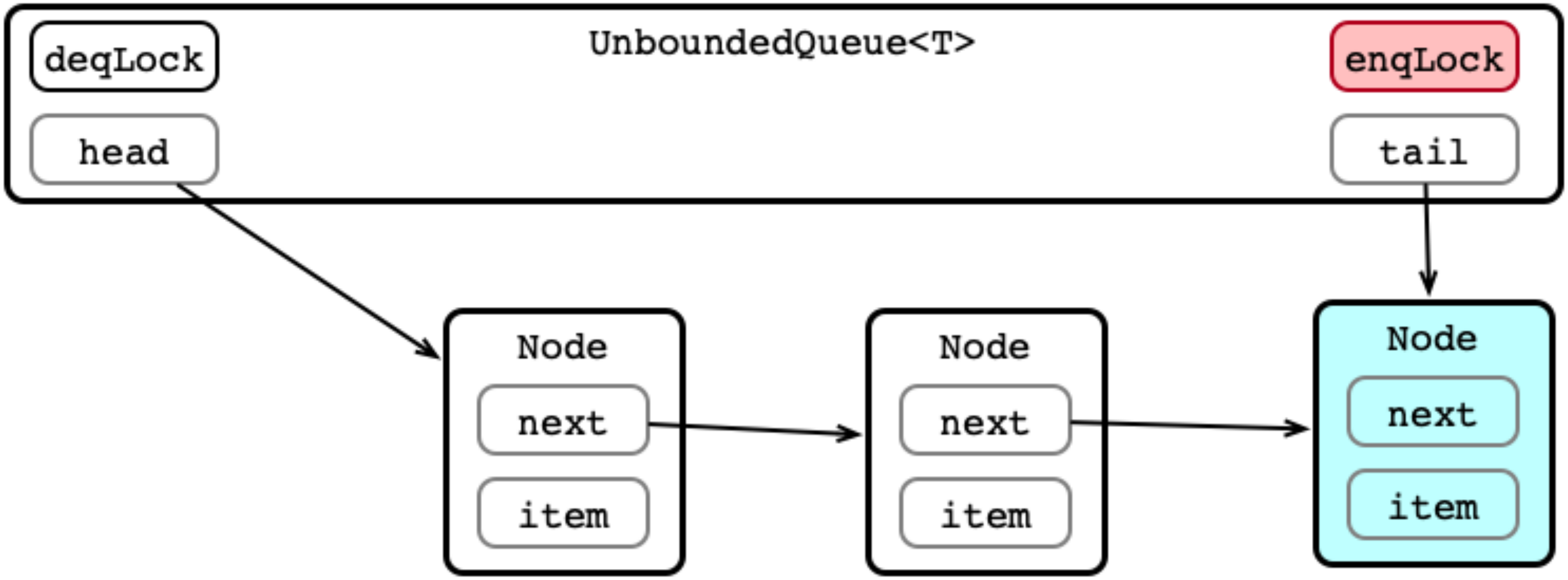
Enqueue 2: Acquire enqLock



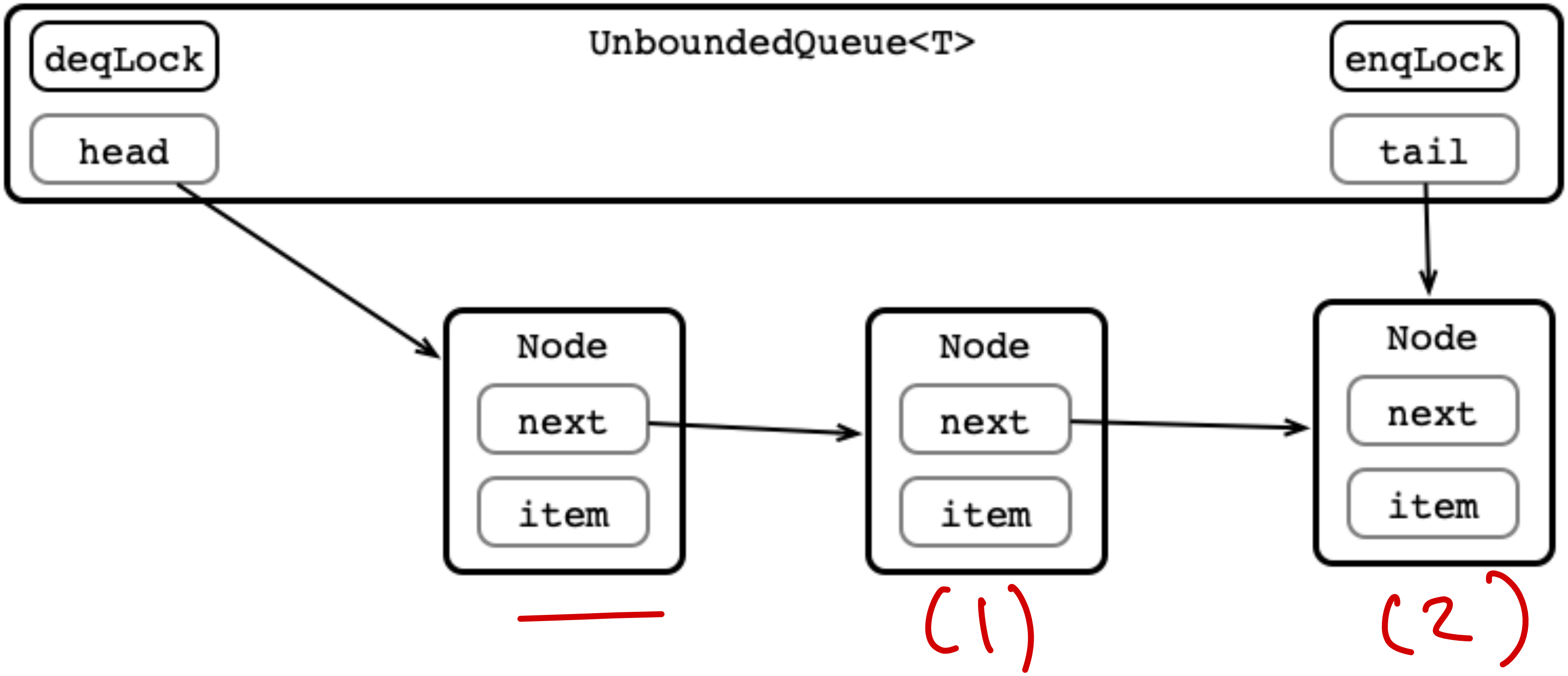
Enqueue 3: Update `tail.next`



Enqueue 4: Update `tail`



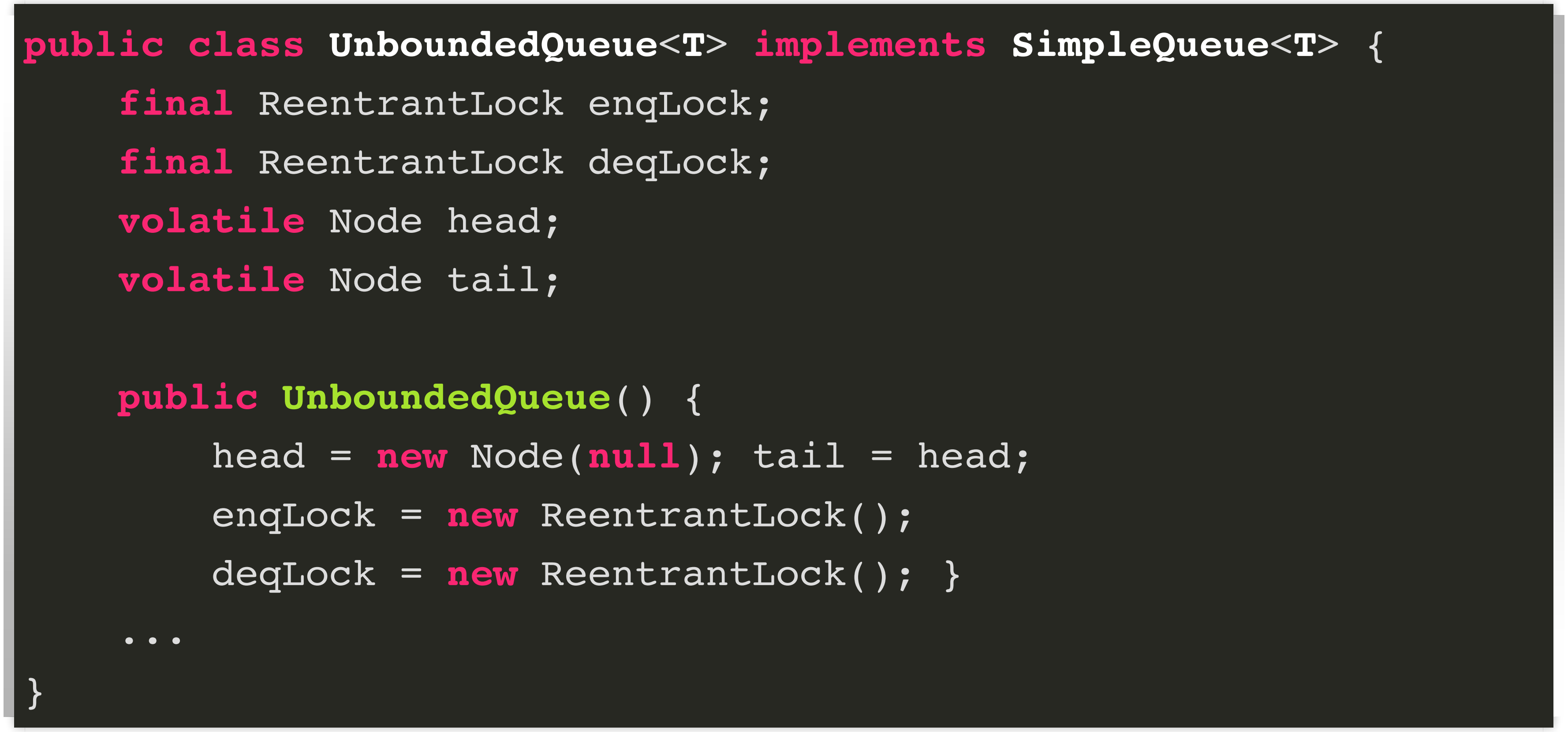
Enqueue 5: Release Lock



Question

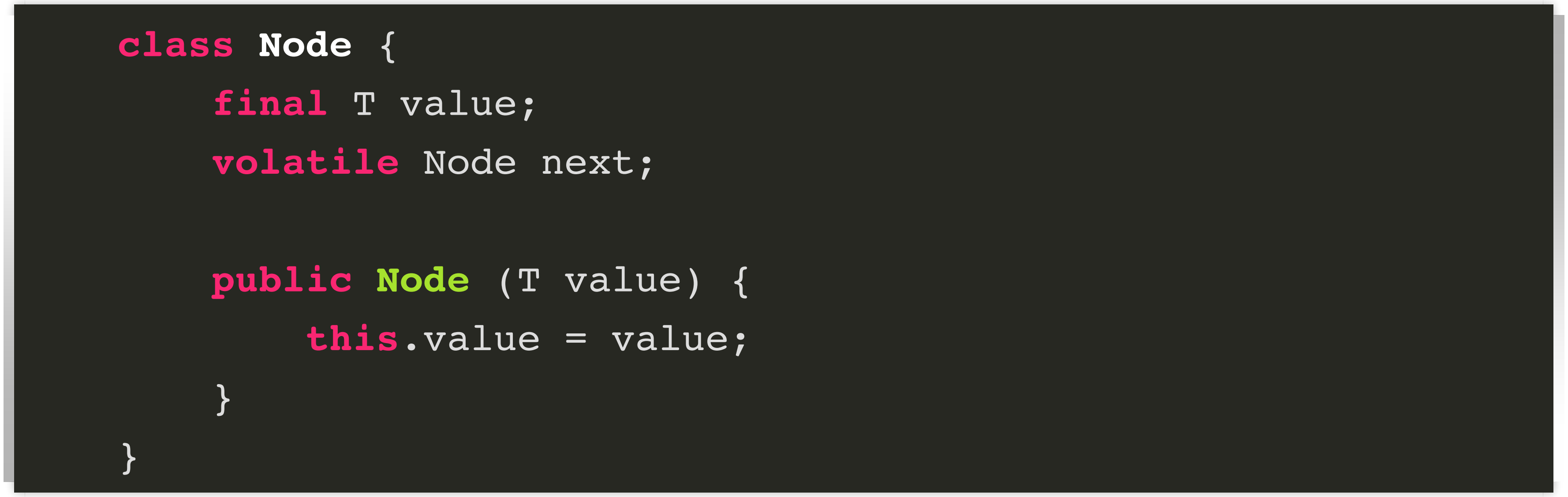
Why do we need the sentinel node?

UnboundedQueue in Code



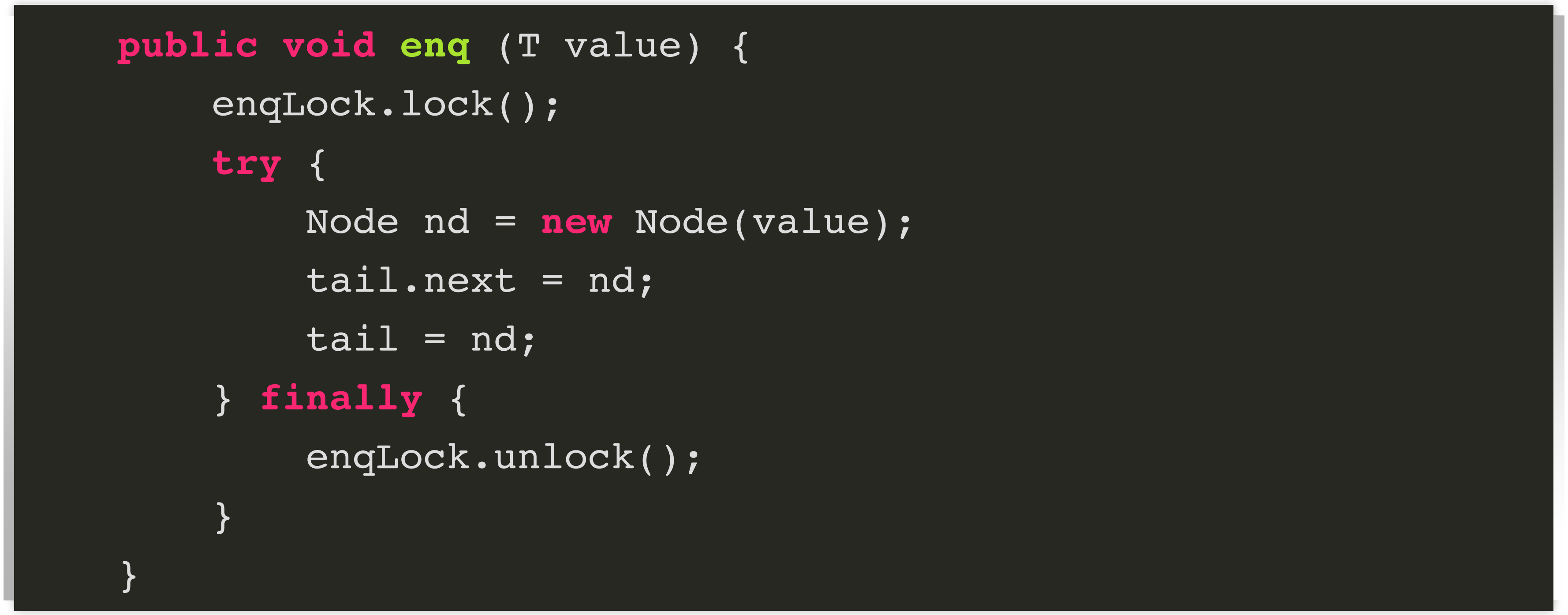
```
public class UnboundedQueue<T> implements SimpleQueue<T> {
    final ReentrantLock enqLock;
    final ReentrantLock deqLock;
    volatile Node head;
    volatile Node tail;

    public UnboundedQueue() {
        head = new Node(null); tail = head;
        enqLock = new ReentrantLock();
        deqLock = new ReentrantLock(); }
    ...
}
```

Node Class



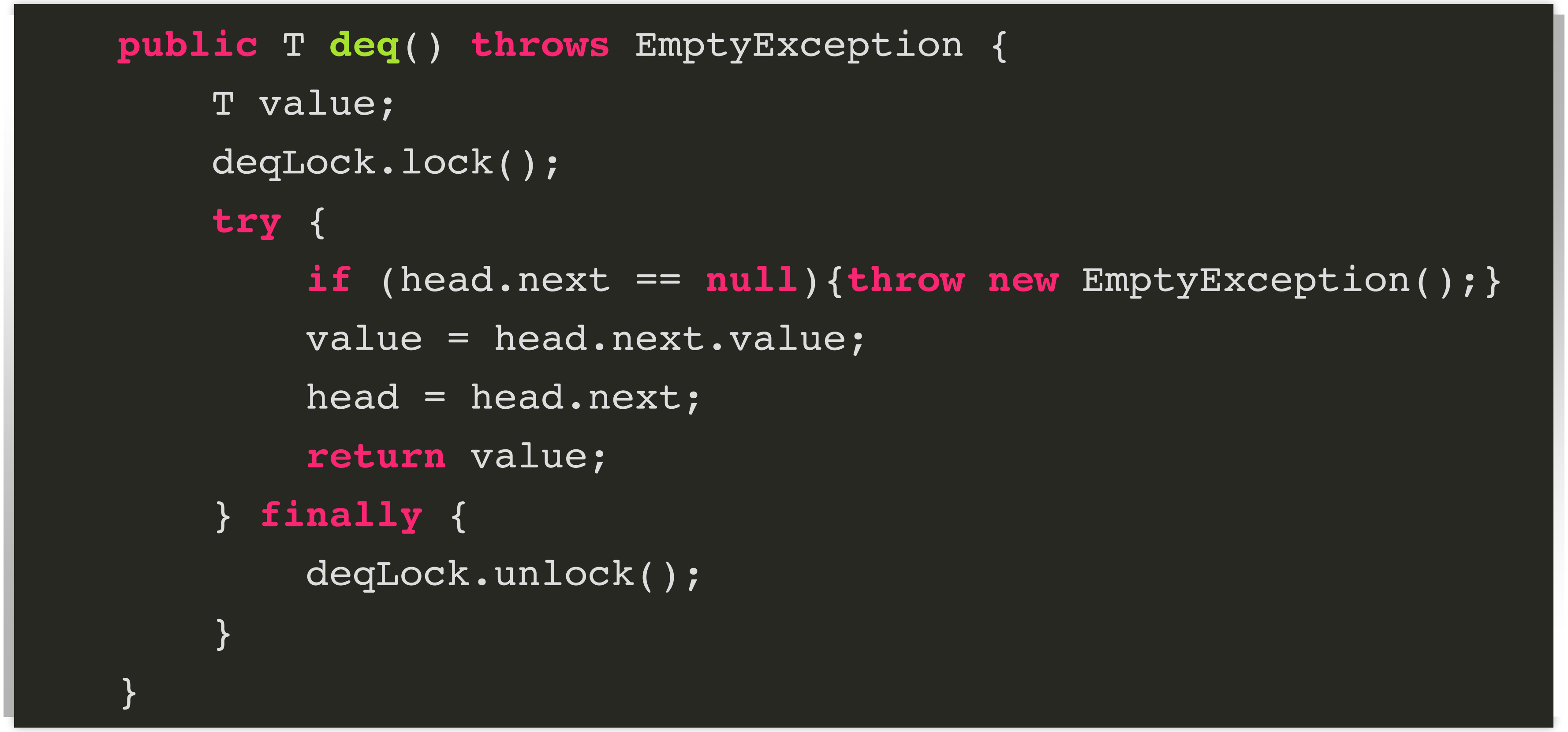
```
class Node {
    final T value;
    volatile Node next;

    public Node (T value) {
        this.value = value;
    }
}
```

enq Method



```
public void enq (T value) {
    enqLock.lock();
    try {
        Node nd = new Node(value);
        tail.next = nd;
        tail = nd;
    } finally {
        enqLock.unlock();
    }
}
```

deq Method

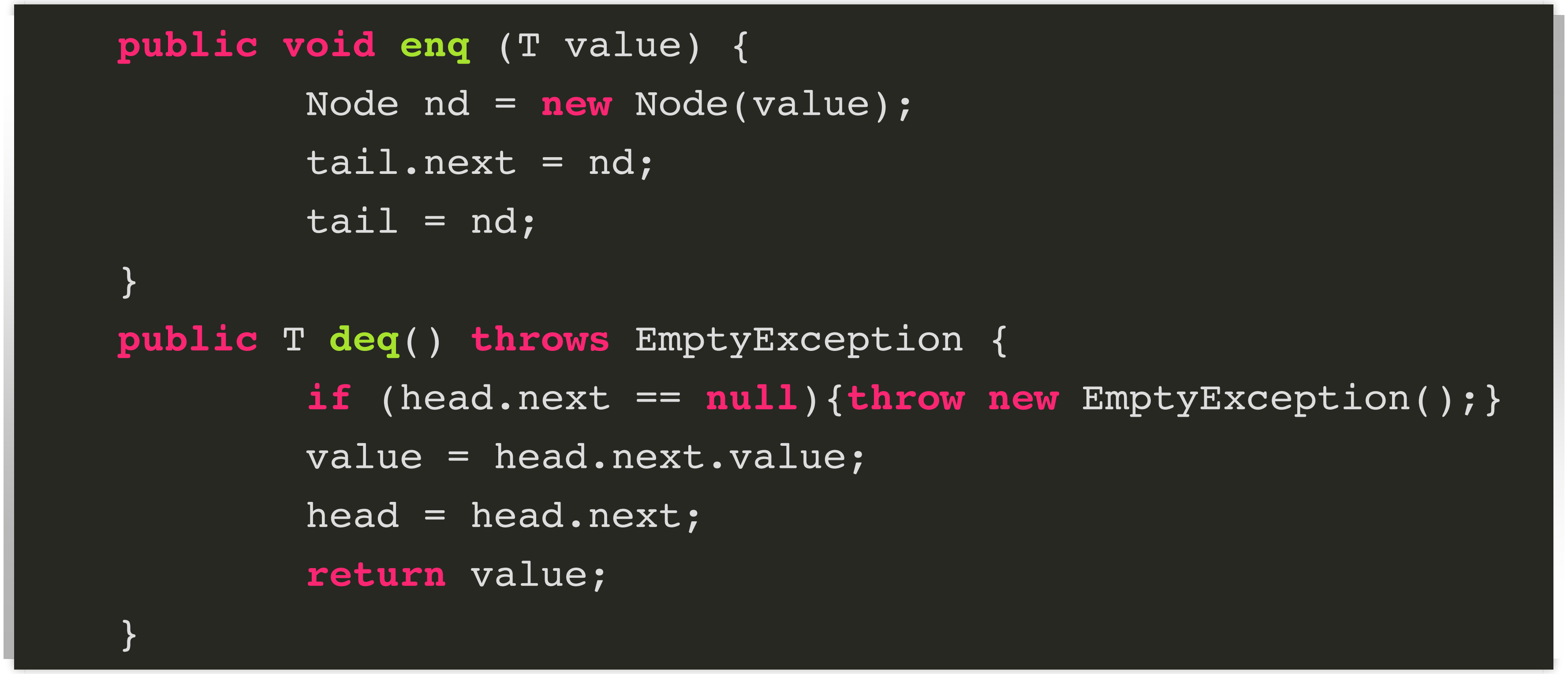


```
public T deq() throws EmptyException {
    T value;
    deqLock.lock();
    try {
        if (head.next == null){throw new EmptyException();}
        value = head.next.value;
        head = head.next;
        return value;
    } finally {
        deqLock.unlock();
    }
}
```

Is `UnboundedQueue` Linearizable?

1. What concurrent operations do we need to consider?
2. What internal states do we need to consider?
3. What are the linearization points (if any)?

Pertinent Lines



```
public void enq (T value) {
        Node nd = new Node(value);
        tail.next = nd;
        tail = nd;
}
public T deq() throws EmptyException {
        if (head.next == null){throw new EmptyException();}
        value = head.next.value;
        head = head.next;
        return value;
}
```

Next Time

Concurrent queues without locks?!?!