Lecture 22: More on Concurrent Queues

COSC 273: Parallel and Distributed Computing

Spring 2023

Announcements

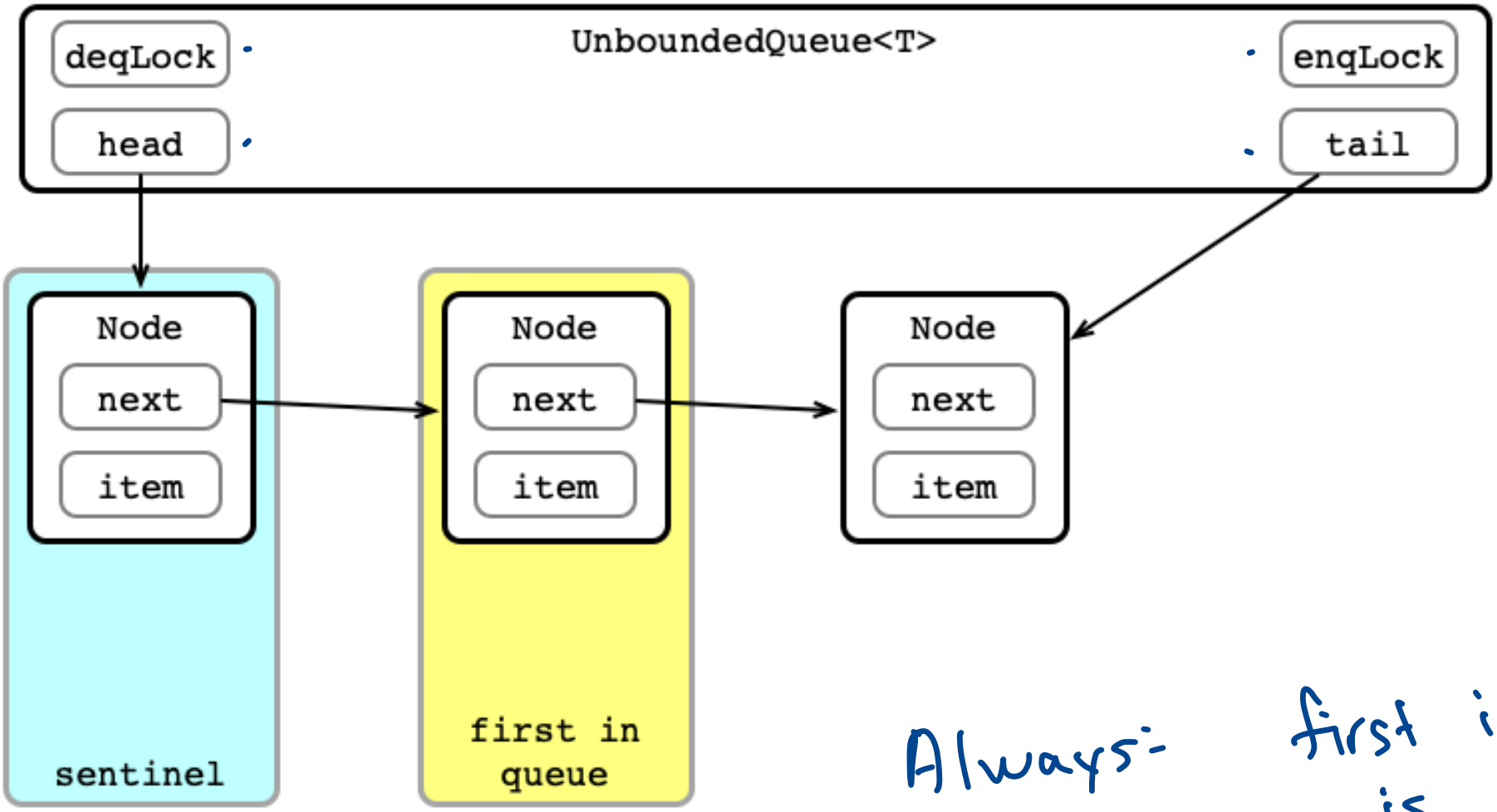
1. Homework 03 Posted Soon
    - due Friday, April 14th
2. Final Projects Announced Soon ←
    - small groups

Last Time

`UnboundedQueue` ←

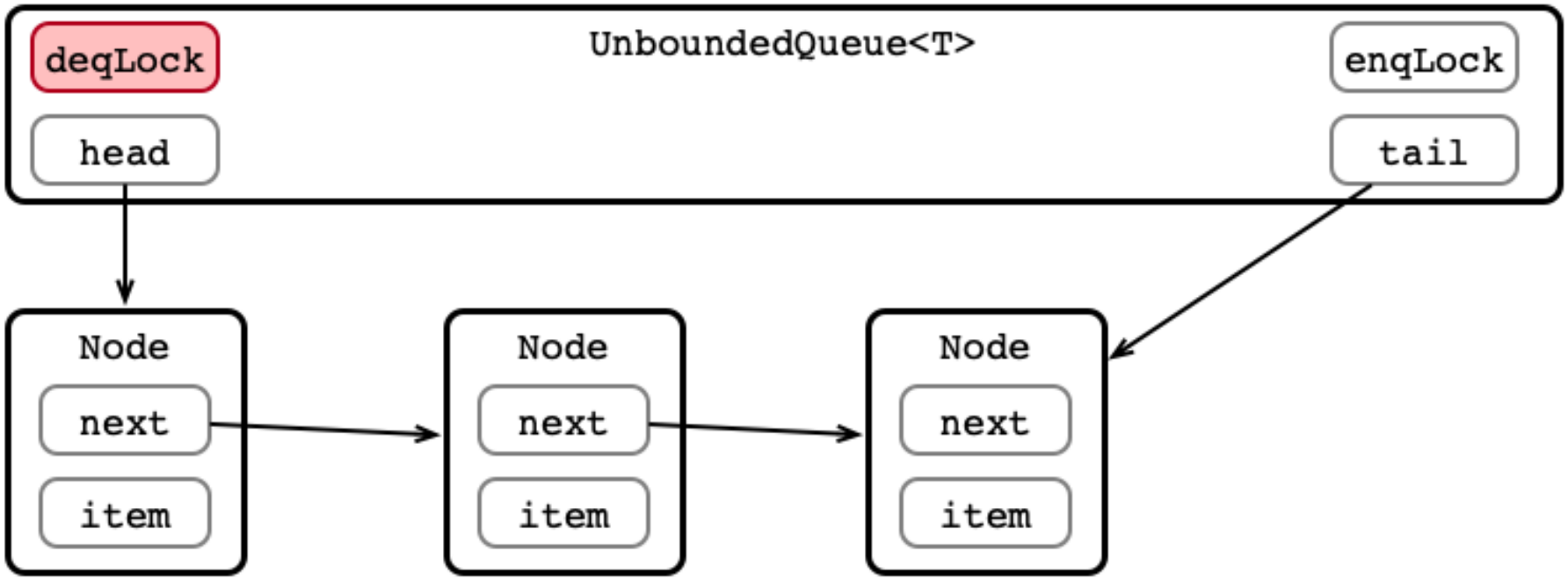
- concurrent linked-list implementation of a queue
- lock enq and deq *operations*, not nodes

Unbounded Queue in Pictures

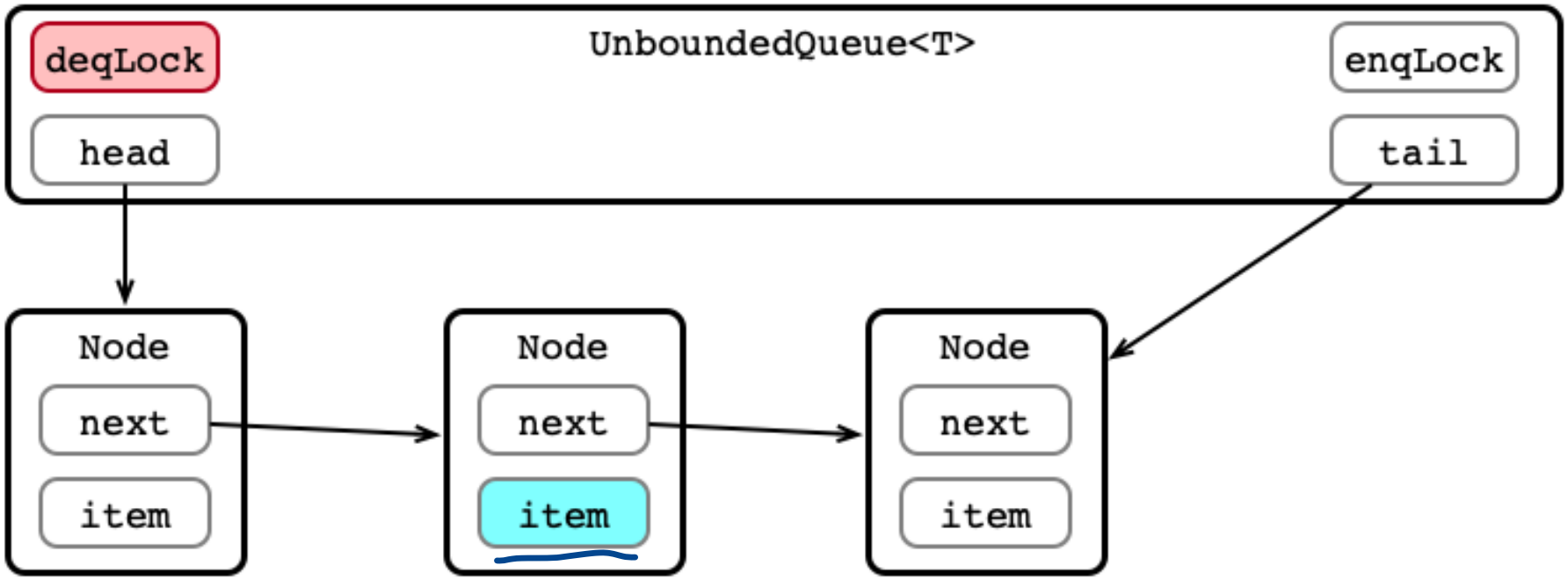


Always: first item in queue is stored in head.next

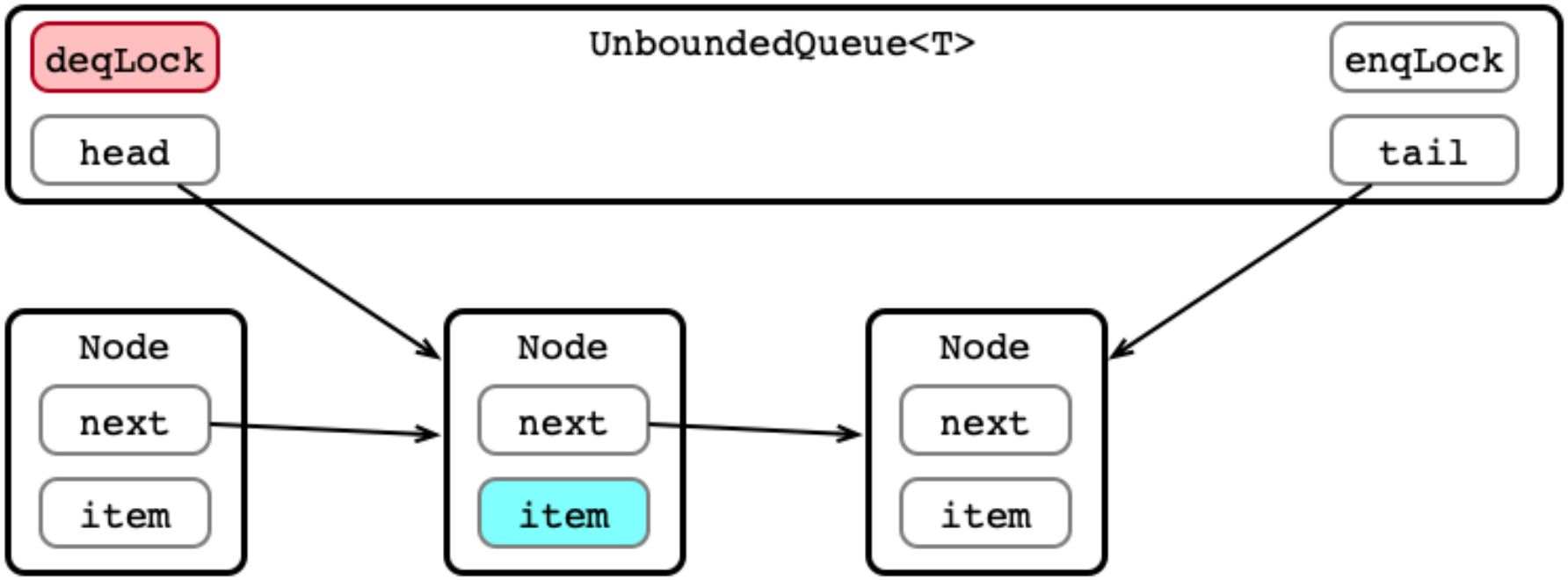
Dequeue 1: Aquire deqLock



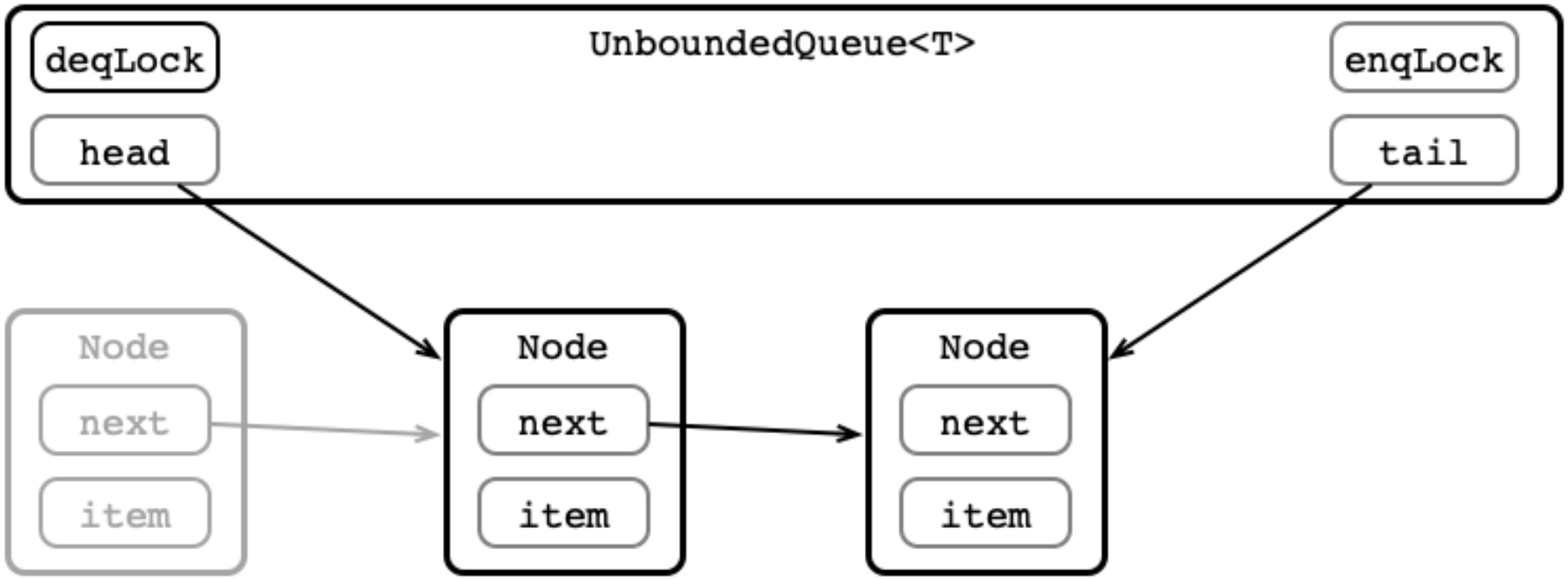
Dequeue 2: Get Element (or Exception)



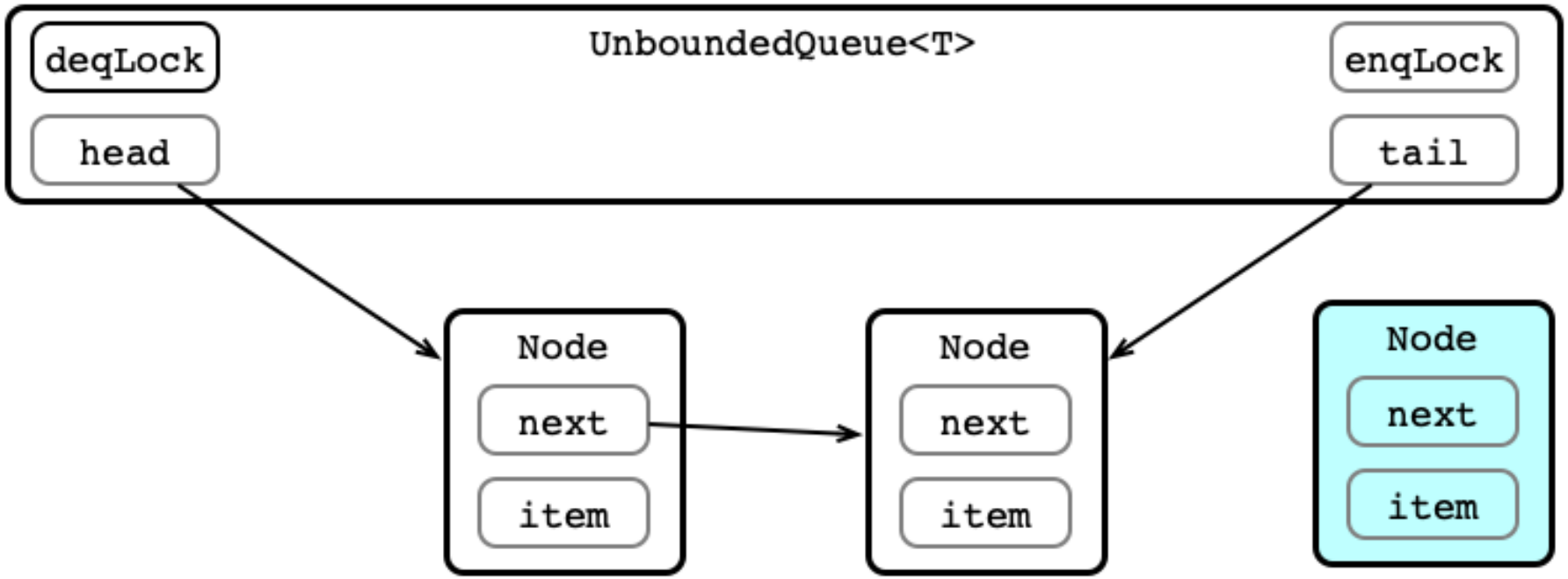
Dequeue 3: Update head



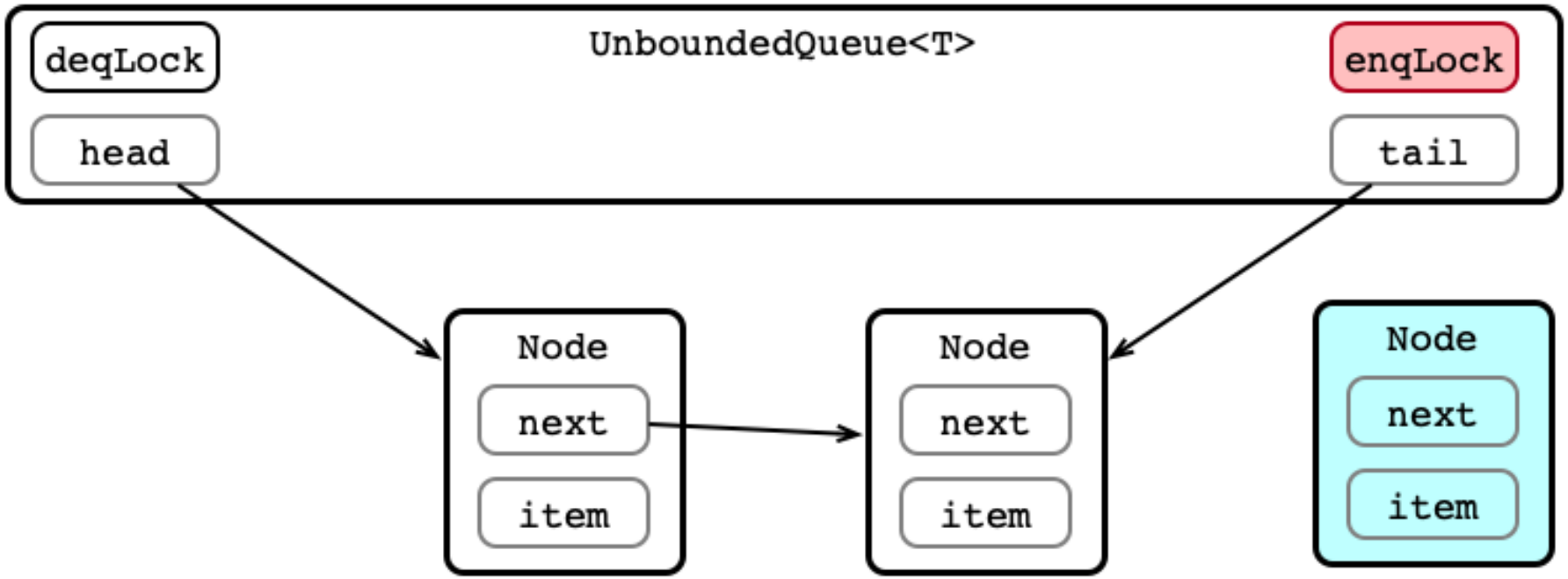
Dequeue 4: Release Lock



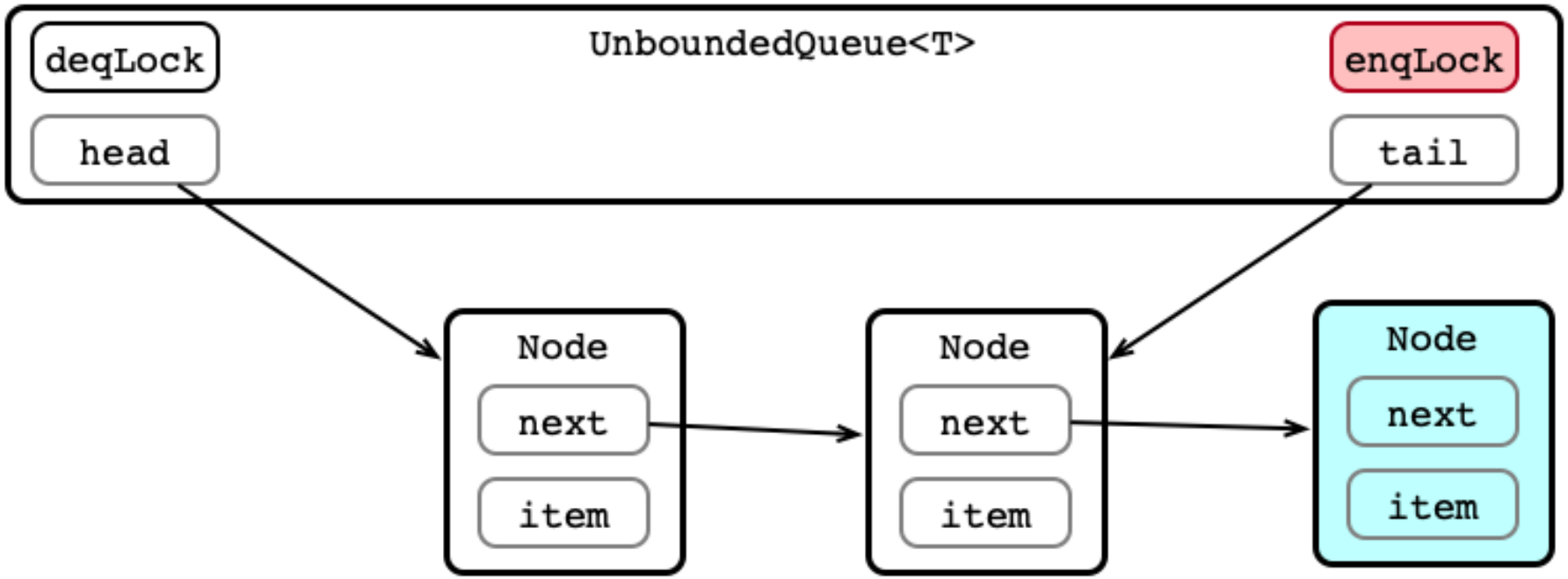
Enqueue 1: Make Node



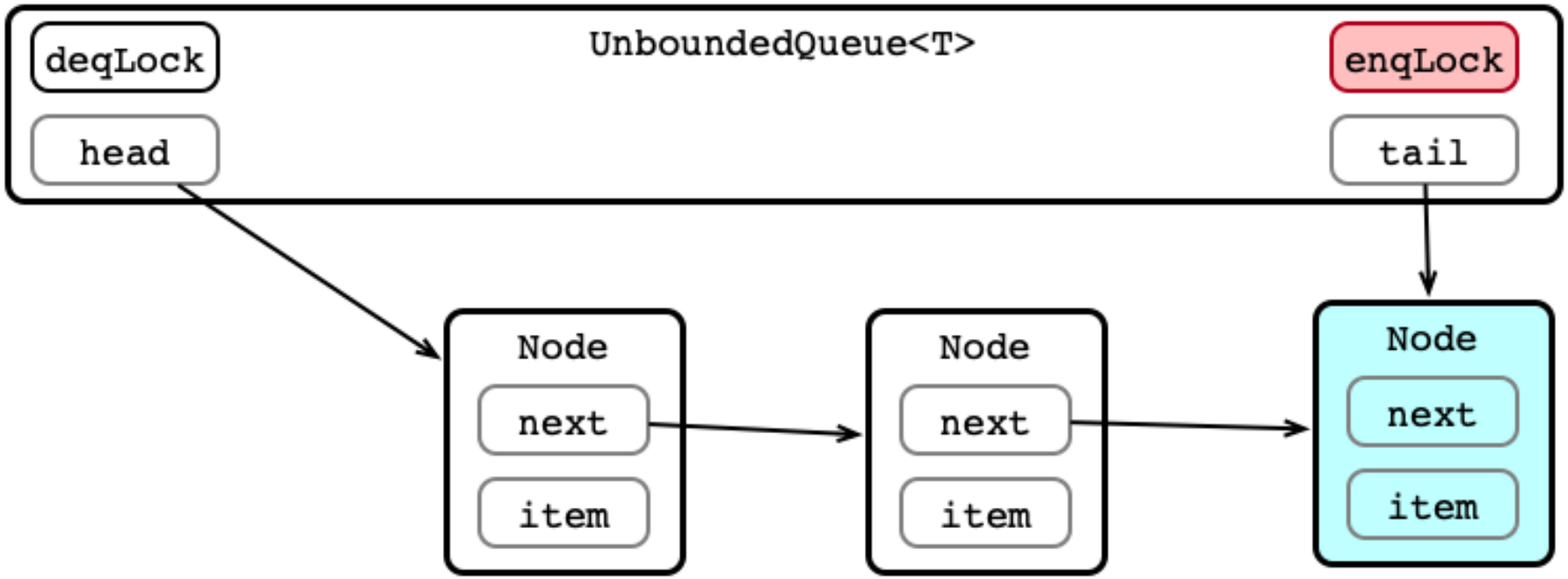
Enqueue 2: Acquire enqLock



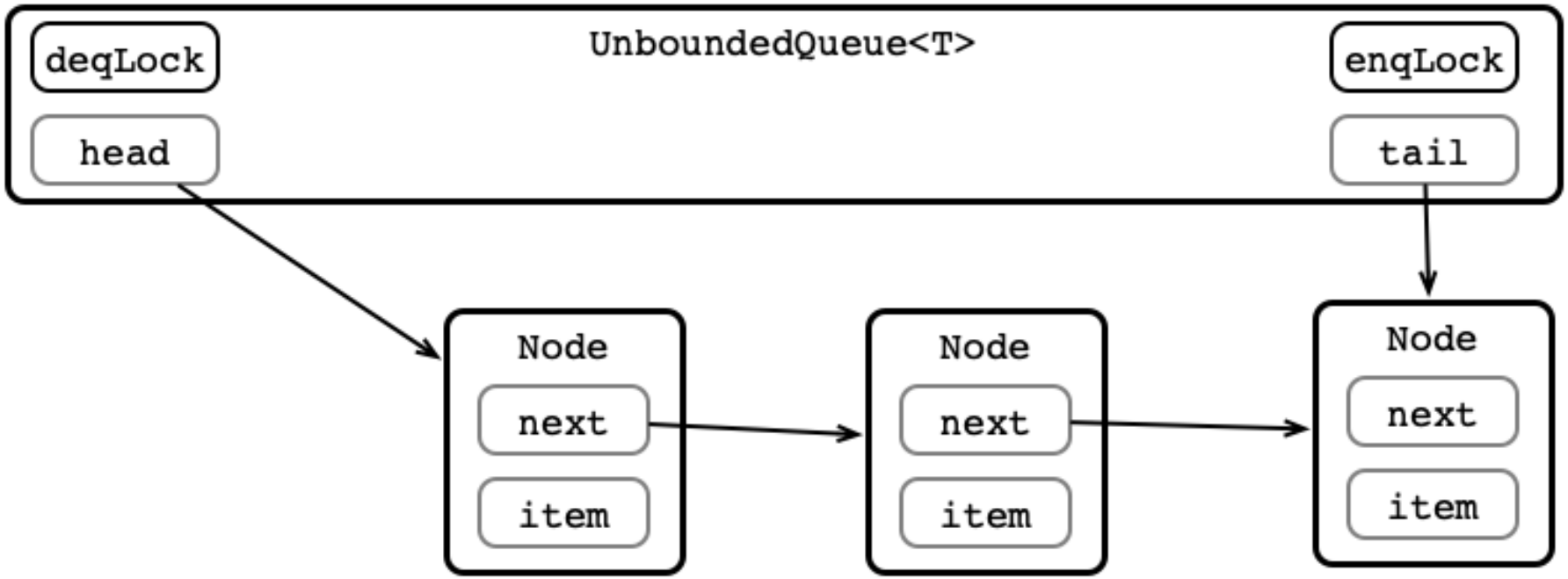
Enqueue 3: Update `tail.next`



Enqueue 4: Update `tail`



Enqueue 5: Release Lock



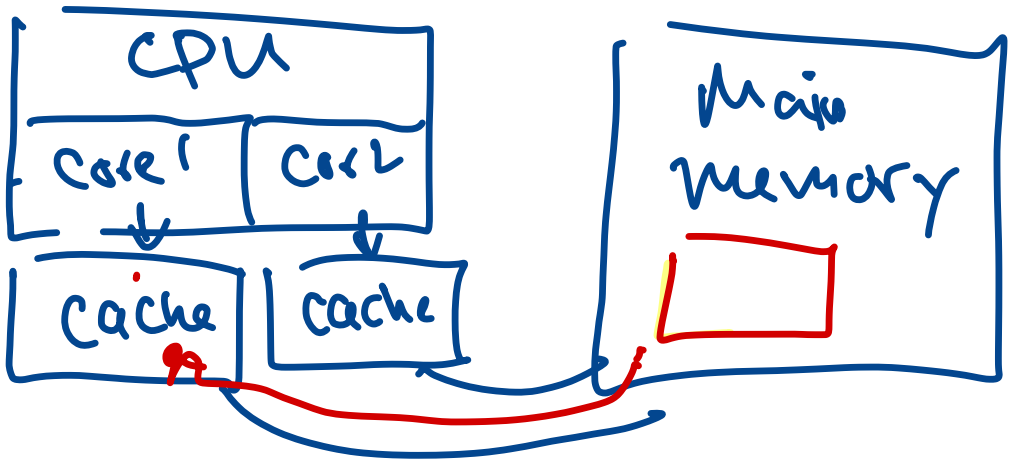
UnboundedQueue in Code

```
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(); }
    ...
}
```

Java built-in lock implementation

sentinel node



"cache coherence" ⟹ volatile always read/write to main memory

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();
  }
}
```

obtain lock

Critical section

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?

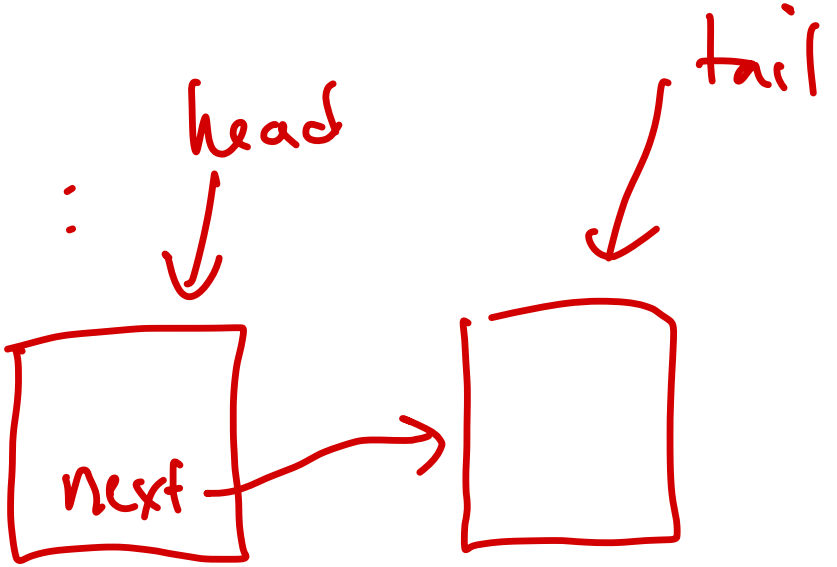
1. What concurrent operations do we need to consider? ←
2. What internal states do we need to consider?
3. What are the linearization points (if any)?

enq/enq ⇒ no concurrent crit sections

deq/deq ⇒ same

T1 enq
T2 deq

Suppose 1 elt :



Pertinent Lines

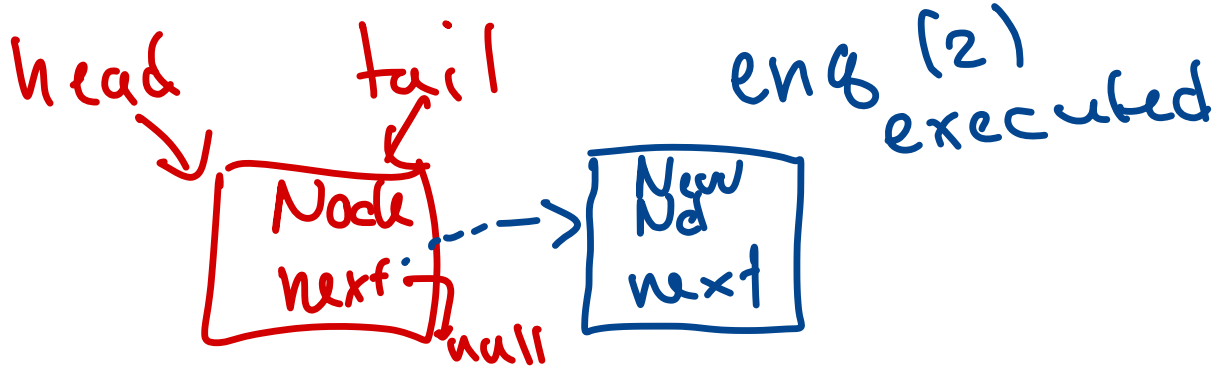
```
public void enq (T value) {
  (e1) Node nd = new Node(value);
  (e2) tail.next = nd;
  (e3) tail = nd;
}
public T deq() throws EmptyException {
  if (head.next == null){throw new EmptyException();}
  value = head.next.value;
  head = head.next;
  return value;
}
```

enq C.S.

deq C.S.

If queue is empty
head / tail → sentinel node

linearization Points



Conclusion

`UnboundedQueue`:

- is linearizable
- allows for concurrent *progress* on calls to `enq` and `deq`
  - if two threads each obtain `enq` and `deq` locks, both terminate in a finite number of steps independent of the actions of the other
- uses locks
  - concurrent calls to `enq` (or `deq`) are *blocking:* a thread cannot make progress while another thread holds the lock

Question

Is it possible to implement a (sequentially consistent? linearizable?) queue without locks?

Enqueue Without Locks

What could go wrong with concurrent enq?

```
public void enq (T value) {
    Node nd = new Node(value);
    tail.next = nd;
    tail = nd;
}
```

Possible Linearization Point?

```
public void enq (T value) {
    Node nd = new Node(value);
    tail.next = nd;
    tail = nd;
}
```

New Tech: `AtomicReferences`

```
// an AtomicReference pointing to someNode
var nd = new AtomicReference<Node>(someNode);

// try to update nd to refer to updated
nd.compareAndSet(expected, update);
```

Effect of `compareAndSet(expected, update)`:

- if `nd`'s current value is `expected`, then update value to `update`
  - return `true`
- if `nd`'s current value is not `expected`, do not update its value
  - return `false`

How Could Atomic References Help?

```
public void enq (T value) {
    Node nd = new Node(value);
    tail.next = nd;
    tail = nd;
}
```

Enqueue Idea

To do:

1. update `tail.next` to `nd`
2. update `tail` to `nd`

How?

- Can update `tail.next` *only* if `tail.next == null`
- Try to update `tail.next` to `nd`:
  1. set `last` to `tail`, `next` to `tail.next`
  2. check if `last` is still `null`
  3. update `last.next` to `nd` *only if* `last.next` is still `null`
  4. if 3 fails, try to update `tail` to `next`

LockFreeQueue

```
public class LockFreeQueue<T> implements SimpleQueue<T> {
    private AtomicReference<Node> head;
    private AtomicReference<Node> tail;
    ...
    public void enq(T item) {...}
    public T deq() throws EmptyException {...}
    class Node {
        public T value;
        public AtomicReference<Node> next;
        ...
    }
}
```

Lock Free enq

```
public void enq(T item) {
    if (item == null) throw new NullPointerException();
    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);}}}}
```

Linearization Point (if any)?

```
public void enq(T item) {
    if (item == null) throw new NullPointerException();
    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);}}}}
```

Questions (Next Time)

1. How to dequeue?
2. Which is better, locked or lock-free?