ID1217 Programming with Processes
Lecture 13

Concurrent Objects

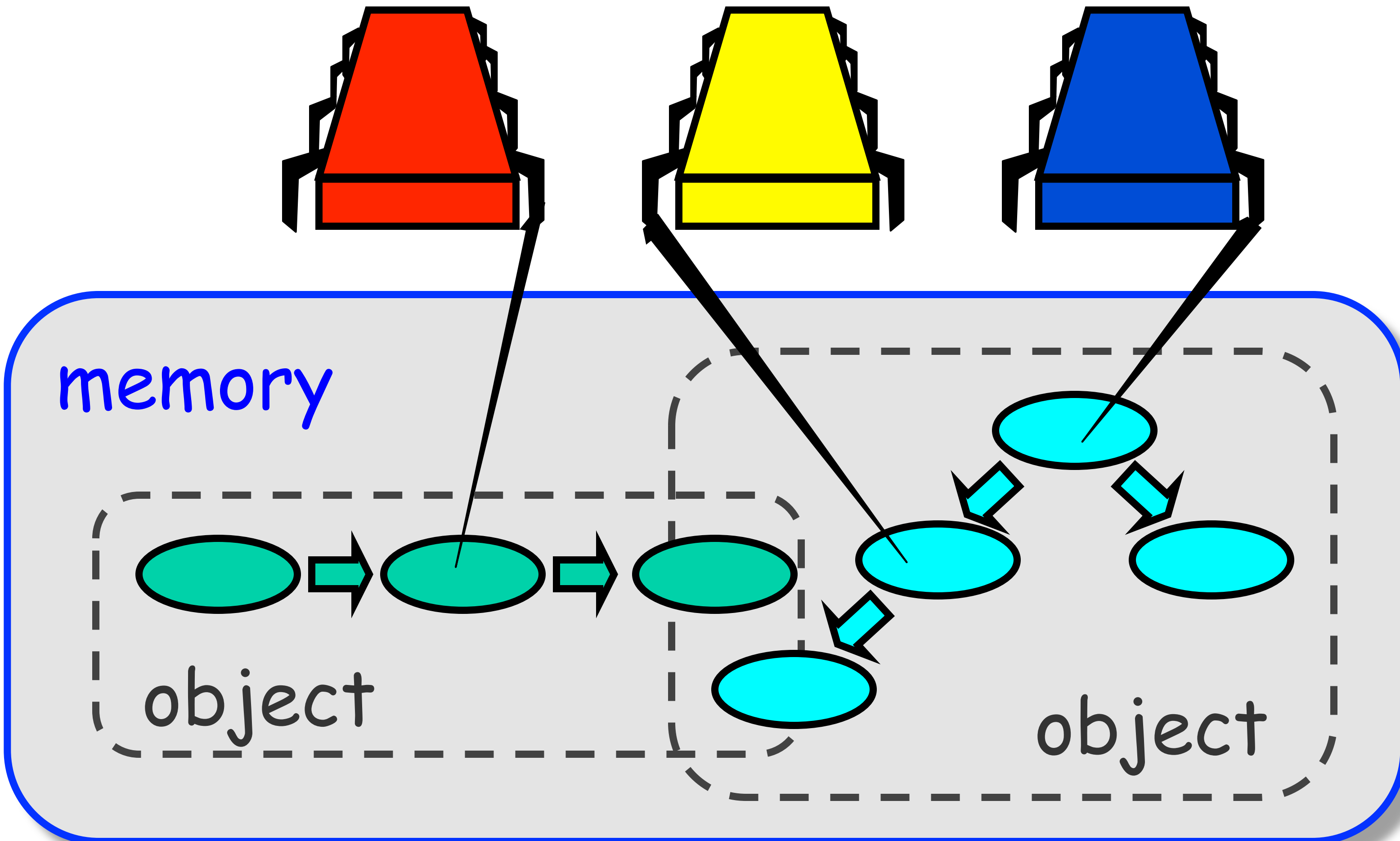
Vladimir Vlassov
KTH/ICT/EECS

Outline

- Concurrent objects
  - Design space
  - Linearization
  - Patterns
  - Case study: Concurrent set based on ordered linked list

- Based on "***The Art of Multiprocessor Programming***" by Maurice Herlihy & Nir Shavit
  - Chapter 3 Concurrent objects
  - Chapter 9 Concurrent Lists: The Role of Locking
- *Acknowledgement*: slides are adapted from slides provided by the authors M. Herlihy and Nir Shavit

Concurrent Computation with Shared Objects



Shared Object Design Space (1/3)

- **Lock-based** vs **Lock-free** objects
  - It's about mutual exclusion synchronization on shared objects: whether and how accesses to shared objects are synchronized
  - Lock-based (synchronized)
    - *Monitors*: A lock per object; One caller (thread) at a time with mutual exclusion;
  - Lock-free (unsynchronized)
    - *Concurrent objects*: No locks; Allow concurrent access by multiple callers (threads)

Shared Object Design Space (2/3)

- **Coarse-grained** vs **Fine-grained** synchronization
  - It's about the number of locks protecting a shared object and its components
  - Coarse-grained synchronization: one lock for the entire object (Monitor model)
  - Fine-grained synchronization: multiple locks: each object component has a lock
    - Split object into independently-synchronized components
    - E.g. each element of a set has its own lock.

Shared Object Design Space (3/3)

- **Blocking** vs **Non-blocking (wait-free)** objects
  - It's about condition synchronization: await or not on a condition in an object
  - Blocking
    - Caller waits until state changes, e.g. a dequeuer awaits until a queue is not empty;
    - Spinning (busy waiting) vs blocking (using wait/signal)
  - Non-Blocking
    - No wait: Method throws exception, e.g. EmptyException
    - Methods return null
  - Wait-free
    - Every method call completes in a finite number of steps
    - Implies no mutual exclusion

Objectivism (1/2)

- What is a **concurrent object**?
  - How do we **describe** one?
  - How do we **implement** one?
  - How do we **tell if we’re right**?
    - Safety (correct behavior) and liveness (progress) properties
- First we focus on
  - how to describe, and
  - how to reason about safety (correctness) and liveness (progress) properties of an implementation algorithm.

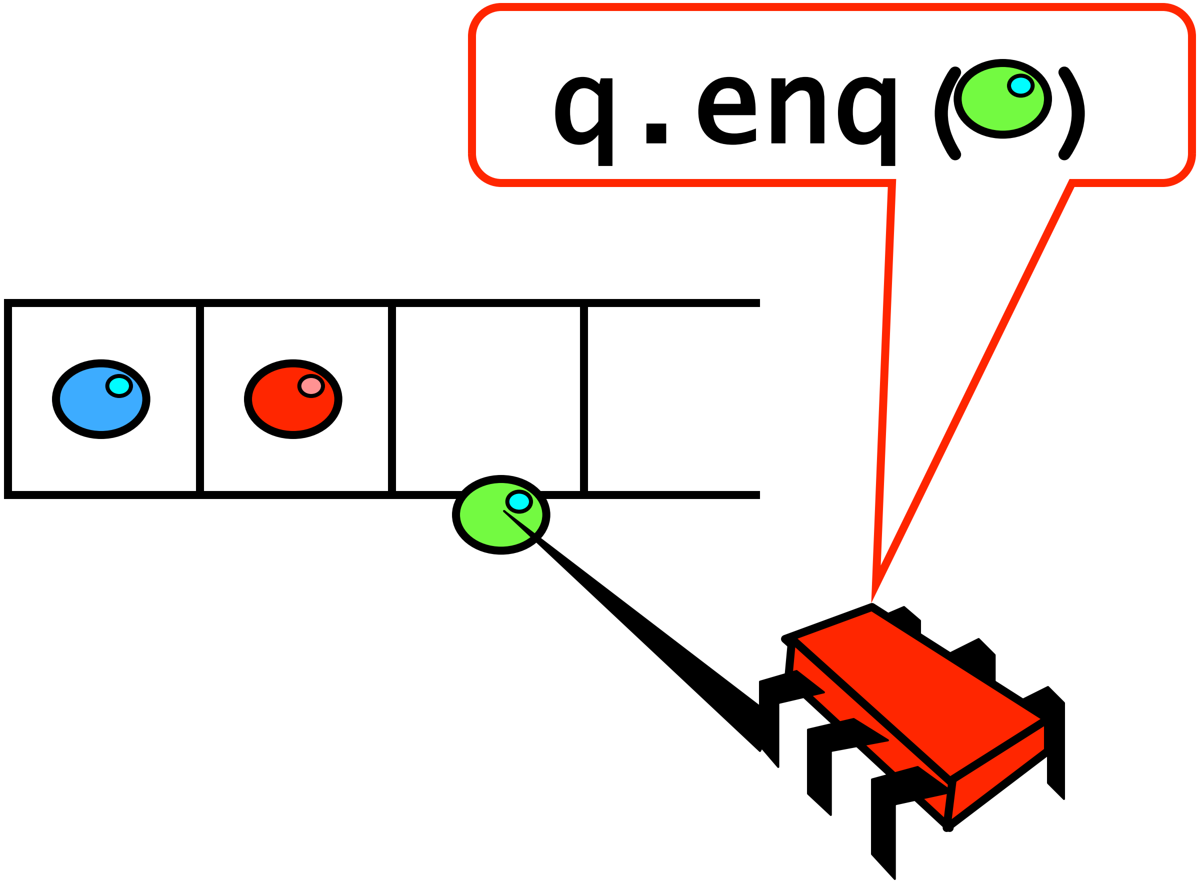
Objectivism (2/2)

- Each object has a **state**
  - Usually given by a set of **fields**
  - Queue example: sequence of items
- Each object has a set of **methods**
  - Only way to manipulate (inspect, alter) state
  - Queue example: **`enq`** and **`deq`** methods
- Methods take time: **A method call is a time interval**
- Concurrent method calls may overlap, may be nested
  - How do we describe and reason about concurrent objects?
  - How do we know that the implementation is correct and threads make progress?

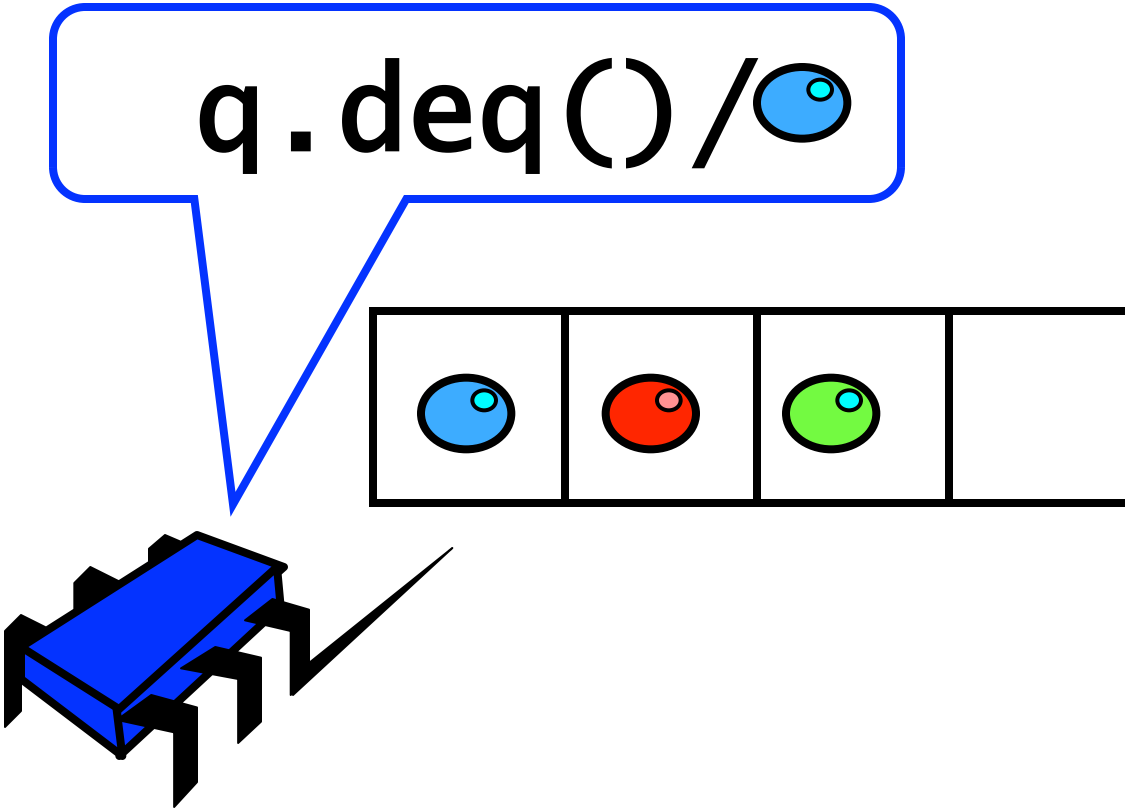
FIFO Queue Example

- Queue
  - pool of items
  - `enq()` & `deq()` methods
  - First-In-First-Out (FIFO) order

FIFO Queue: Enqueue Method



FIFO Queue: Dequeue Method



Consider Two Implementations of a Bounded Queue

1. Lock-based (synchronized) bounded FIFO queue
2. Lock-free (wait-free) bounded FIFO queue

- How to check whether implementations are correct?

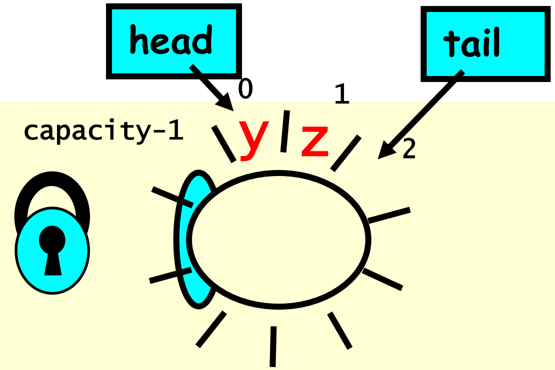
Lock-based Bounded FIFO Queue

- Lock-based (synchronized)
  - The Monitor model: Lock to enqueue/dequeue; unlock when done or on exception
  - Modifications are mutually exclusive
- Bounded
  - Fixed capacity
  - Good when resources are an issue
- Non-blocking (no busy-waiting)
  - **`enq()`** throws **`FullException`** if the queue is full
  - **`deq()`** throws **`EmptyException`** if the queue is empty
  - Versus methods that wait …

Lock-based Bounded Queue

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Lock-based Bounded Queue




```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Queue fields protected by single shared lock

Lock-based Bounded Queue

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Initially head = tail

Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Method calls mutually exclusive

Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

If queue empty throw exception

Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Queue not empty: remove item and update head

Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Return result

Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Release lock no matter what!

Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Should be correct (i.e. FIFO) because modifications are mutually exclusive, i.e. one at a time

Lock-free Wait-free Bounded FIFO Queue

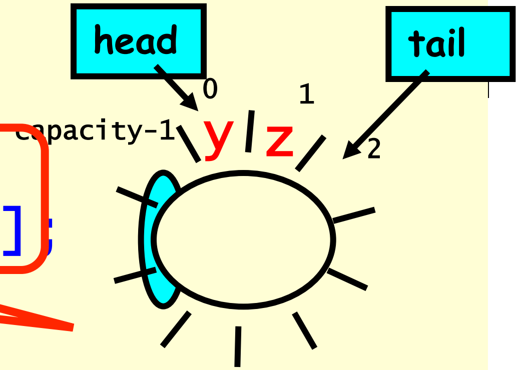
- Lock-free (unsynchronized)
  - The same thing without mutual exclusion
- For simplicity, only **two threads**
  - One thread **enq only**
  - The other **deq only**

Wait-free 2-Thread Queue

```
public class WaitFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    if (tail-head == capacity) throw
         new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```

Wait-free 2-Thread Queue

```
public class WaitFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    if (tail-head == capacity) throw
          new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
          new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```



Wait-free 2-Thread Queue

```
public class WaitFreeQueue {

  int head = 0, tail
  items = (T[]) new O

  public void enq(Item                {
    if (tail-head == capacity) throw
         new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     Iter
     return item;
}}
```

How do we define "correct" when modifications are not mutually exclusive?

Queue is updated without a lock!

Defining Concurrent Queue Implementations

- We saw two types of queue implementations,
  - one that used locks and intuitively felt like a queue,
  - and one that did not use locks, it was in fact, wait-free.
- What properties are they actually providing, are they both implementations of a concurrent queue?
- In what ways are they similar and in what ways do they differ?

Defining Concurrent Queue Implementations (cont'd)

- Need a way to specify a concurrent queue object
- Need a way to prove that an algorithm implements the object's specification, i.e. verify correctness
- Lets talk about object specifications …

Correctness and Progress

- In a concurrent setting, we need to specify both the **safety** and the **liveness** properties of an object
  - Safety: Properties that state that nothing bad ever happens
  - Liveness: Properties that state that something good eventually happens
- Need a way to define
  - when an implementation is **correct** (safety)
  - the conditions under which it guarantees **progress** (liveness)

Lets begin with correctness

Sequential Objects

- Each object has a ***state***
  - Usually given by a set of ***fields***
  - Queue example: sequence of items
- Each object has a set of ***methods***
  - Only way to manipulate state
  - Queue example: **enq** and **deq** methods

Sequential Specifications

- If (precondition)
  - the object is in such-and-such a state
  - before you call the method,
- Then (postcondition)
  - the method will return a particular value
  - or throw a particular exception.
- and (postcondition, con't)
  - the object will be in some other state
  - when the method returns,

Pre and PostConditions for Dequeue

- Precondition:
  - Queue is non-empty
- Postcondition:
  - Returns first item in queue
- Postcondition:
  - Removes first item in queue

- Precondition:
  - Queue is empty
- Postcondition:
  - Throws Empty exception
- Postcondition:
  - Queue state unchanged

Why Sequential Specifications Totally Rock

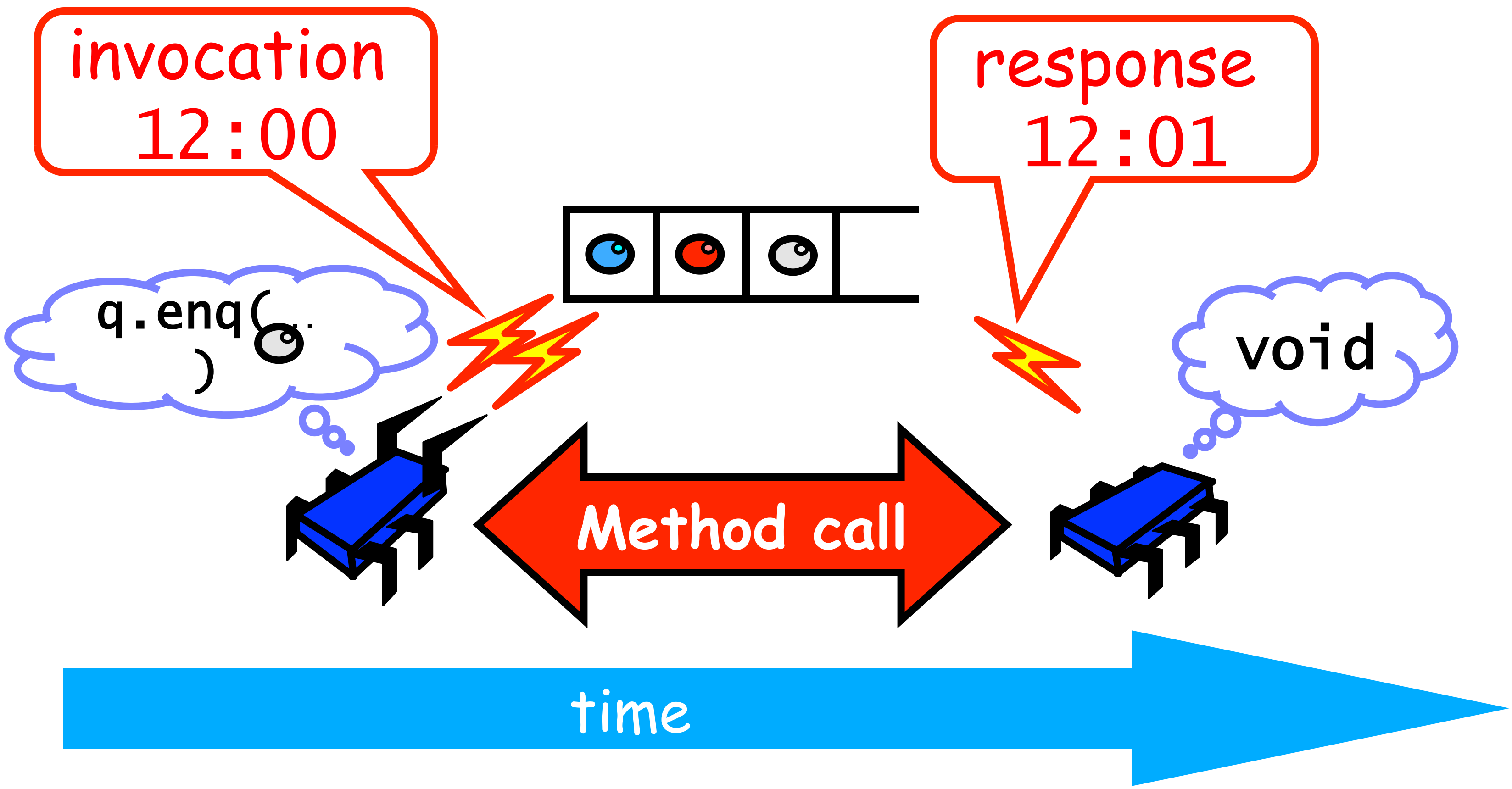
- Interactions among methods captured by side-effects on object state
  - State meaningful between method calls
  - E.g. after enq, an empty queue becomes non-empty; deq from the non-empty queue can make it empty
- Documentation size linear in number of methods
  - Each method described in isolation
- Can add new methods
  - Without changing descriptions of old methods

What About Concurrent Specifications?

- Methods?
- Documentation?
- Adding new methods?

Methods Take Time

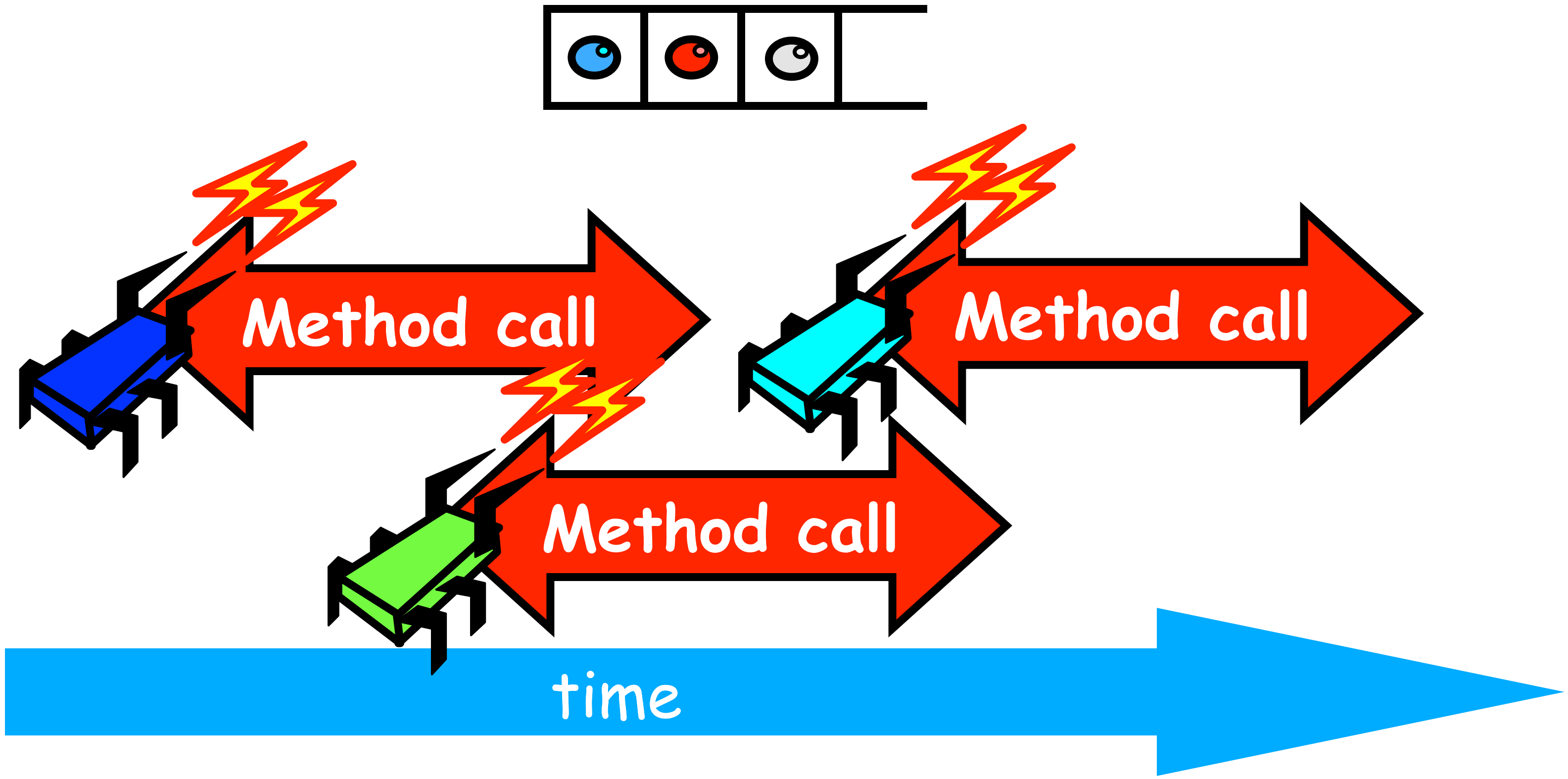
In the sequential world, we treat method calls as if they were instantaneous, but in fact, they take time.



Sequential vs Concurrent

- Sequential
  - Methods take time? Who knew?
- Concurrent
  - Method call is not an event
  - Method call is an interval.
    - Invocation and response events
    - Method takes effect in between the invocation and response events.

Concurrent Methods Take Overlapping Time



Sequential vs Concurrent

Sequential	Concurrent
• Object needs meaningful state only ***between*** method calls	• Because method calls overlap, object might ***never*** be between method calls
• Each method described in isolation	• Must characterize ***all*** possible interactions with concurrent calls – What if two `enq`s overlap? – Two `deq`s? `enq` and `deq`? …
• Can add new methods without affecting older methods	• Everything can potentially interact with everything else

Sequential vs Concurrent

Sequential	Concurrent
• Object needs meaningful state only ***between*** method calls	• Because method calls overlap, object might ***never*** be between method calls
• Each method described in isolation	• Must characterize ***all*** possible interactions with concurrent calls – What if two `enq`s overlap? – Two `deq`s? `enq` and `deq`? …
• Can add new methods without affecting older methods	• Everything can potentially interact with everything else

Panic!

The Big Question

- **What does it mean for a *concurrent* object to be correct?**
  - What *is* a concurrent FIFO queue?
  - FIFO means strict temporal order
  - Concurrent means ambiguous temporal order

Intuitively…

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

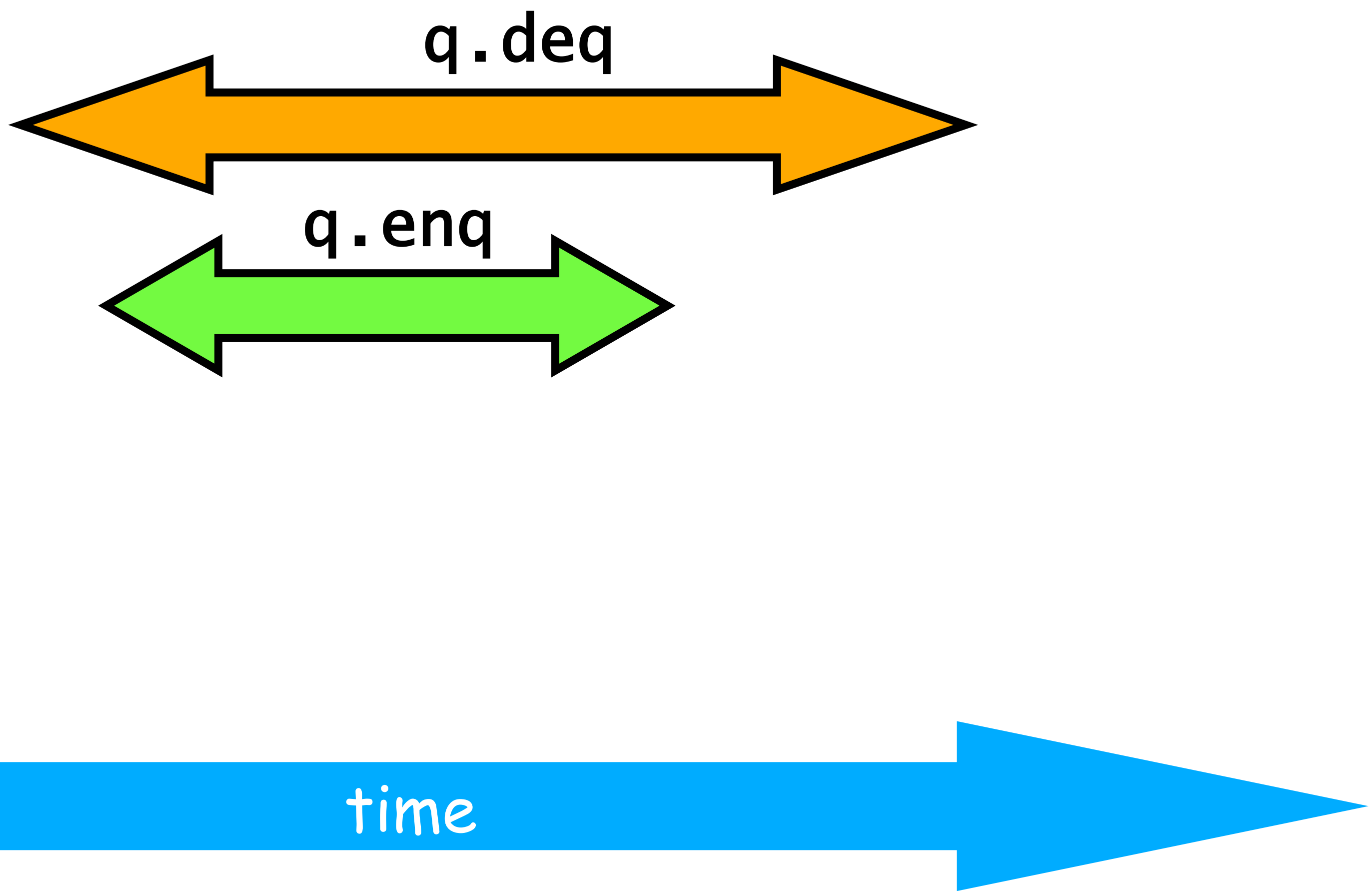
Intuitively…

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

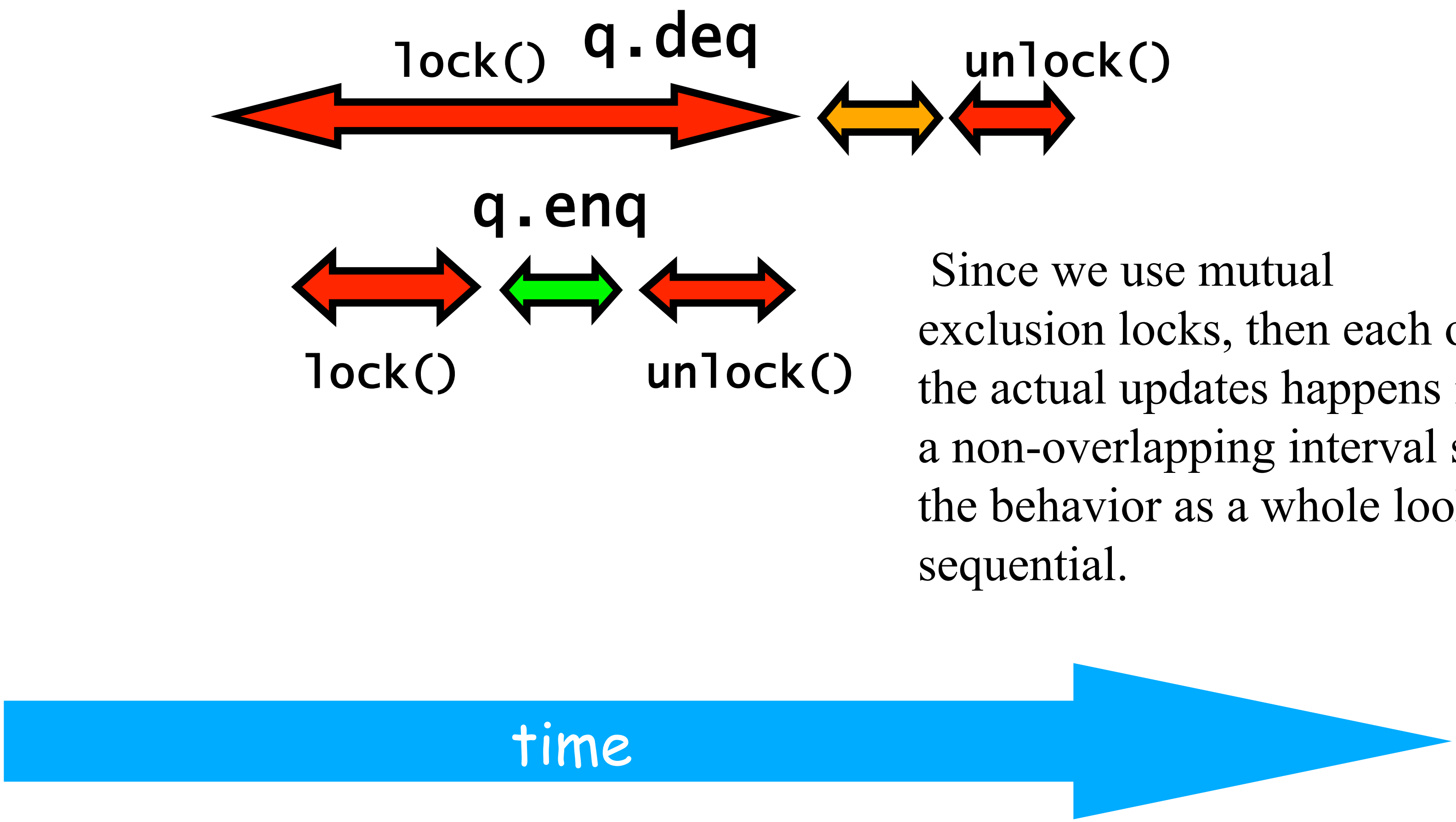
All modifications of queue are done mutually exclusive

Intuitively

Assume method calls overlap.

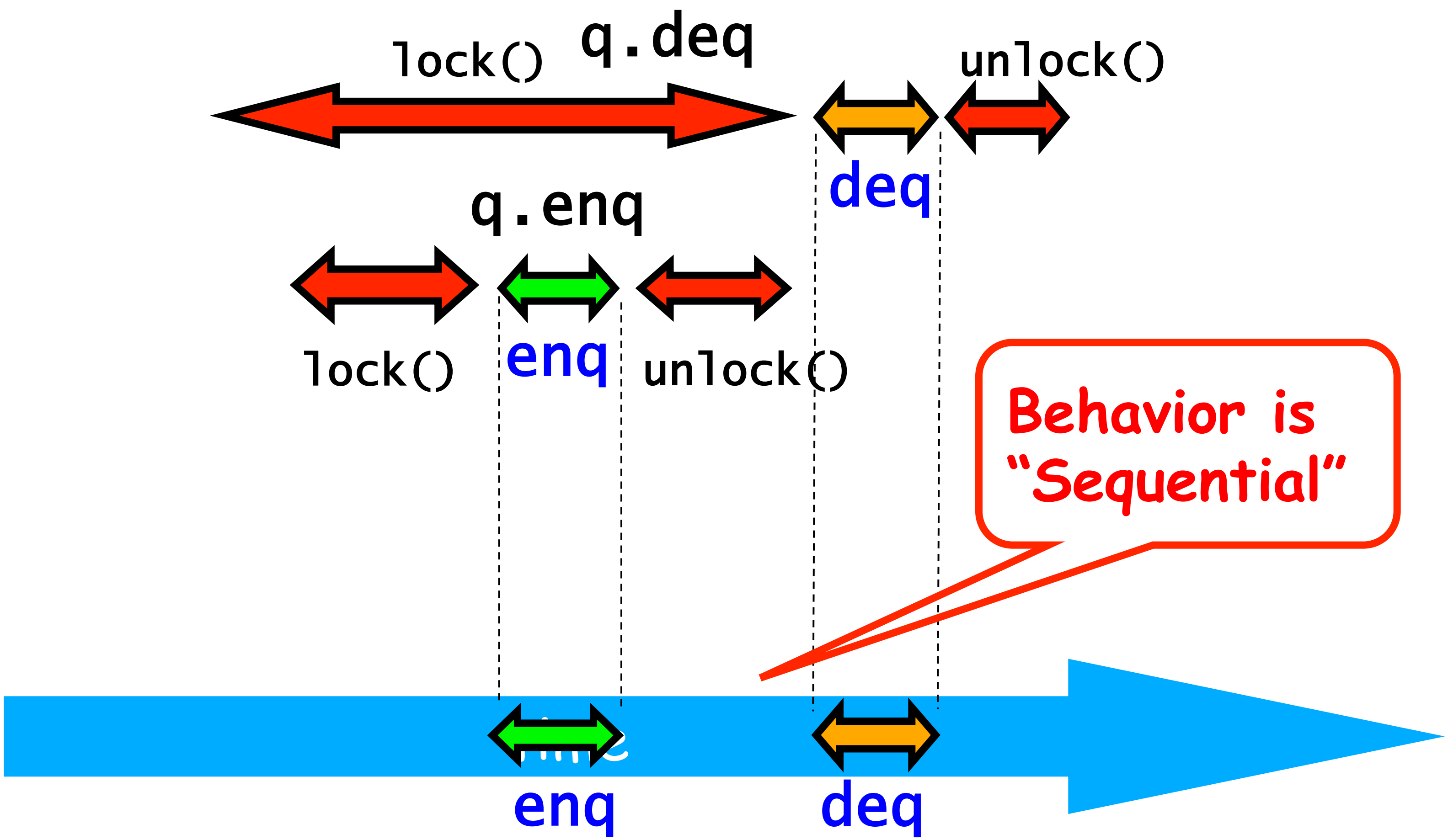


Intuitively

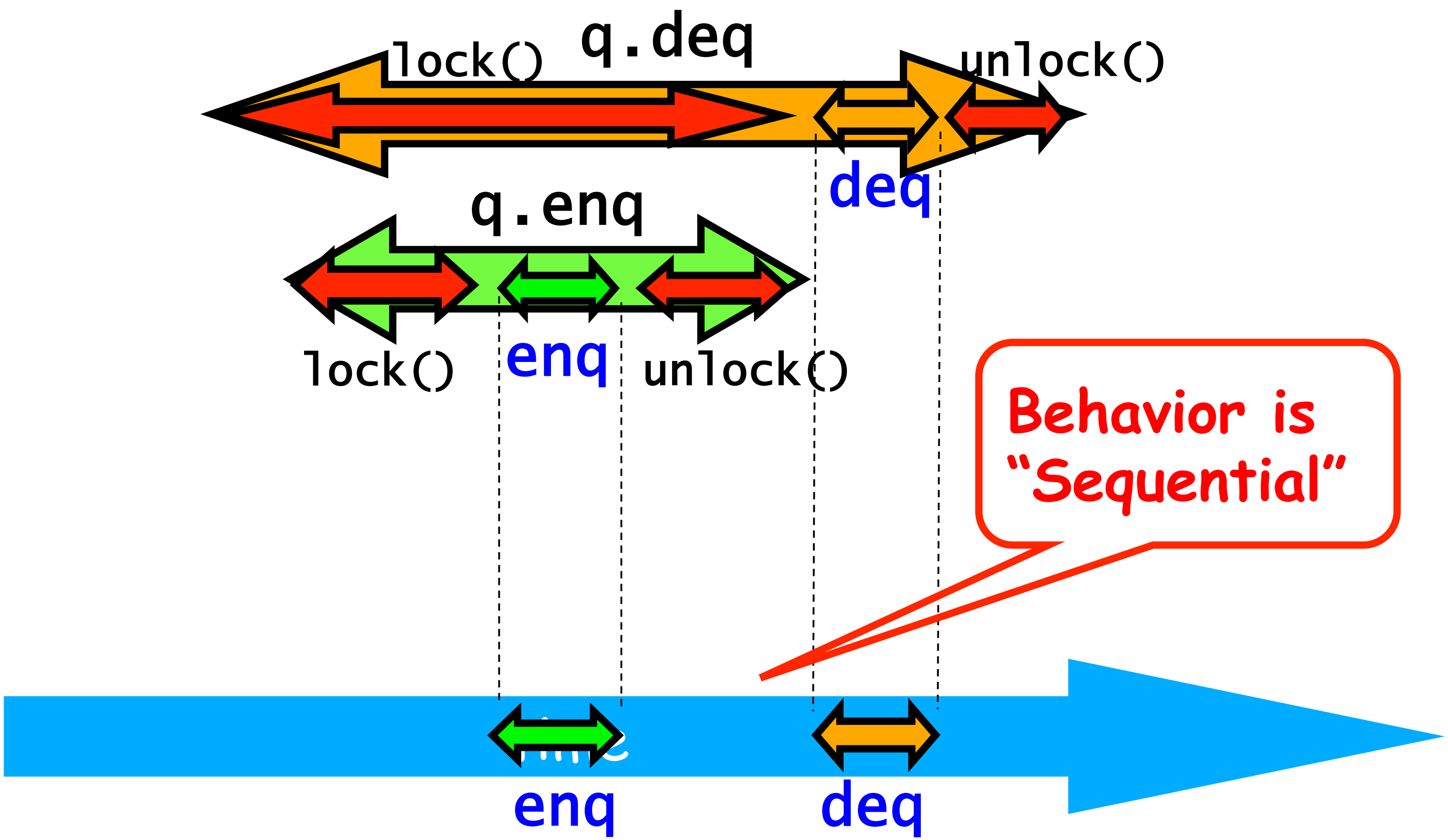


Since we use mutual exclusion locks, then each of the actual updates happens in a non-overlapping interval so the behavior as a whole looks sequential.

Intuitively



Lets capture the idea of describing the concurrent via the sequential



Linearizability (1/4)

- Each method should
  - **"take effect"** (changes become visible to other threads) **instantaneously between invocation and response events** (within the invocation interval)
  - **Linearization points** at which the method "takes effect"
  - A method may have several linearization points
- A concurrent object execution can be viewed as an **interleaving of linearization points** of method invocations on the object by multiple threads

Linearizability (2/4)

- **Every concurrent execution history is equivalent to some sequential execution history** (which can be either legal or not)
  - If one method call precedes another, then the earlier call must have taken effect before the later call.
  - By contrast, if two method calls overlap, then their order is ambiguous, and we are free to order them in any convenient way.
- **A history is legal (correct) if it confirms to the sequential specification for the object**
- **We check the linearized execution (the sequential history) for correctness**

Linearizability (3/4)

- Map the concurrent execution to the sequential one and check the latter for correctness
- **Object is correct if its "sequential" behavior (i.e. its linearized execution) is correct**
- Any such concurrent object is **Linearizable™**

- **Linearizability is a correctness condition for concurrent objects.**

Linearizability (4/4)

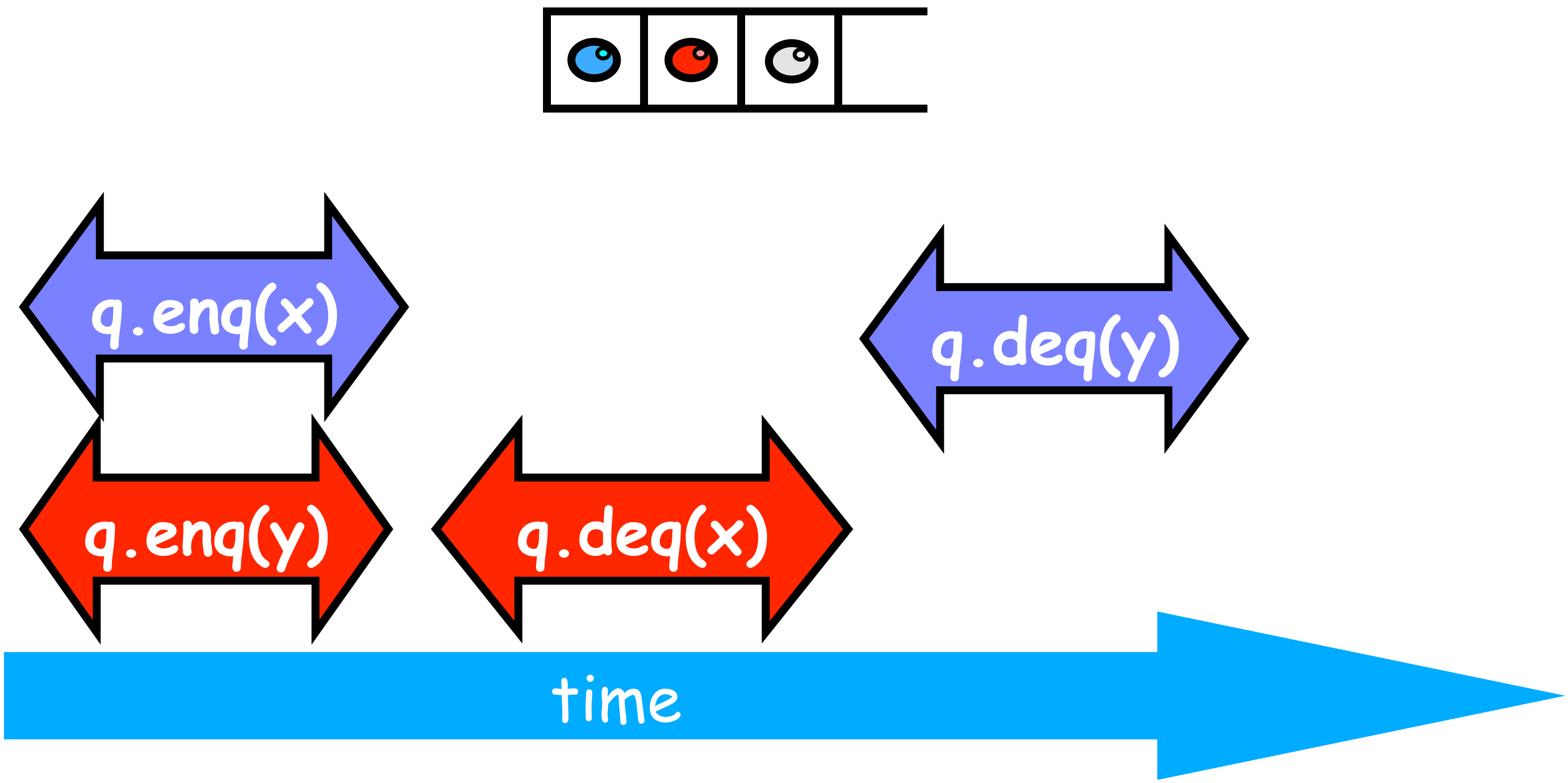
- Is it really about the object?
  - Each method should
  - “take effect”
  - instantaneously
  - between invocation and response events.
- Sounds like a property of an execution…
- **A linearizable object is an object all of whose possible executions are linearizable**

Critical Sections

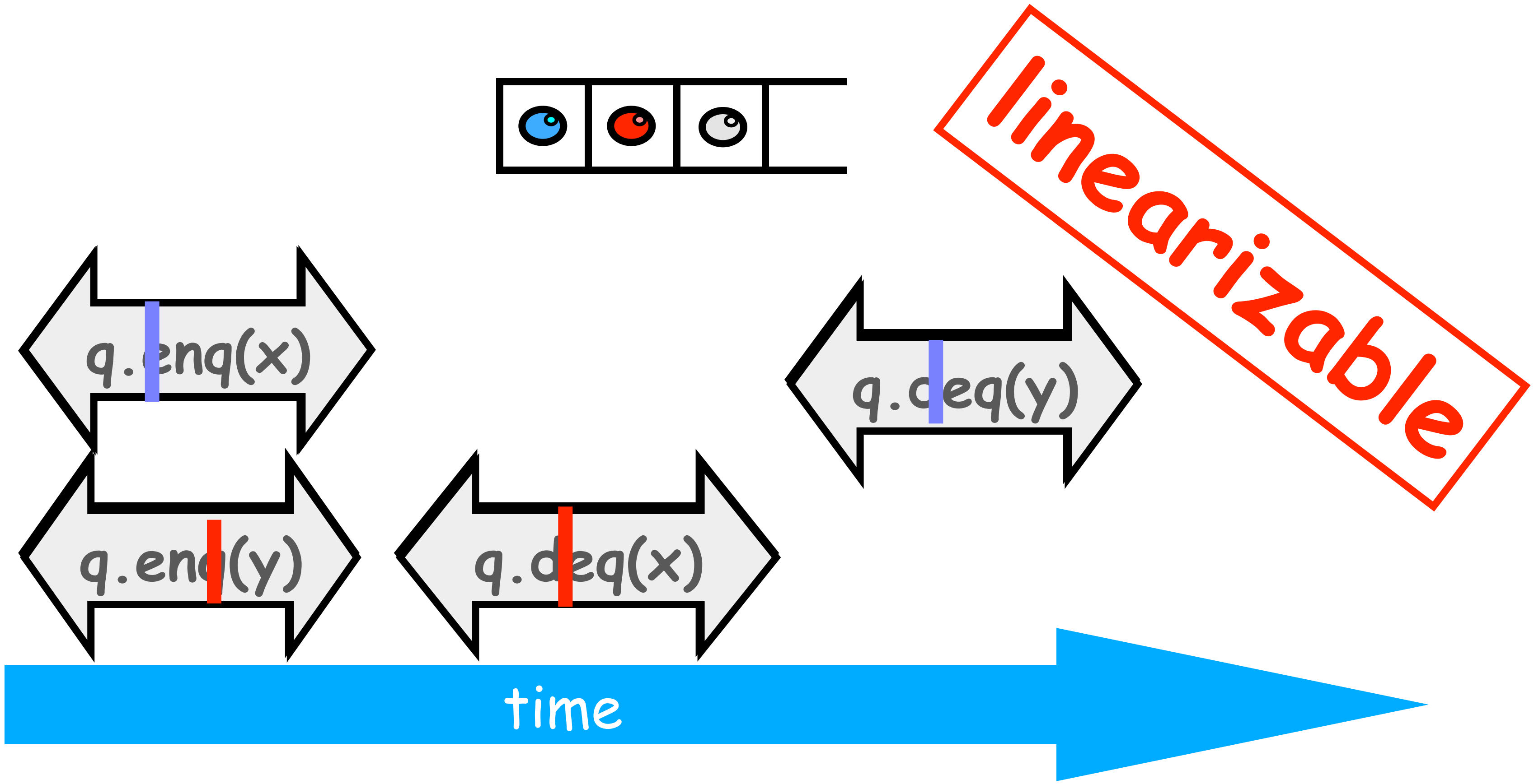
- Easy way to implement linearizability
  - Take sequential object
  - Make each method a critical section
- Problems
  - Blocking
  - No concurrency

Example 1

- Assume the following execution history.
- Is it linearizable?

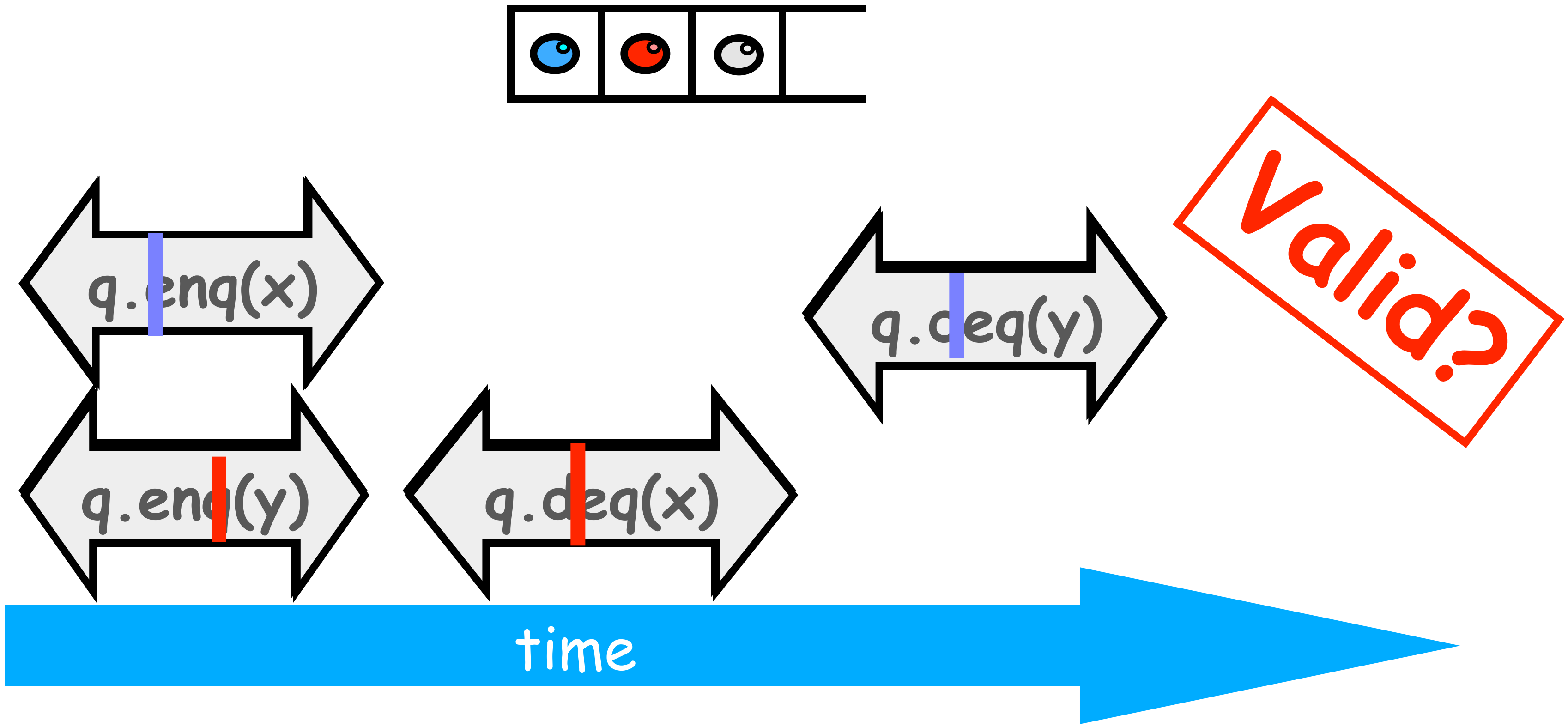


Example 1



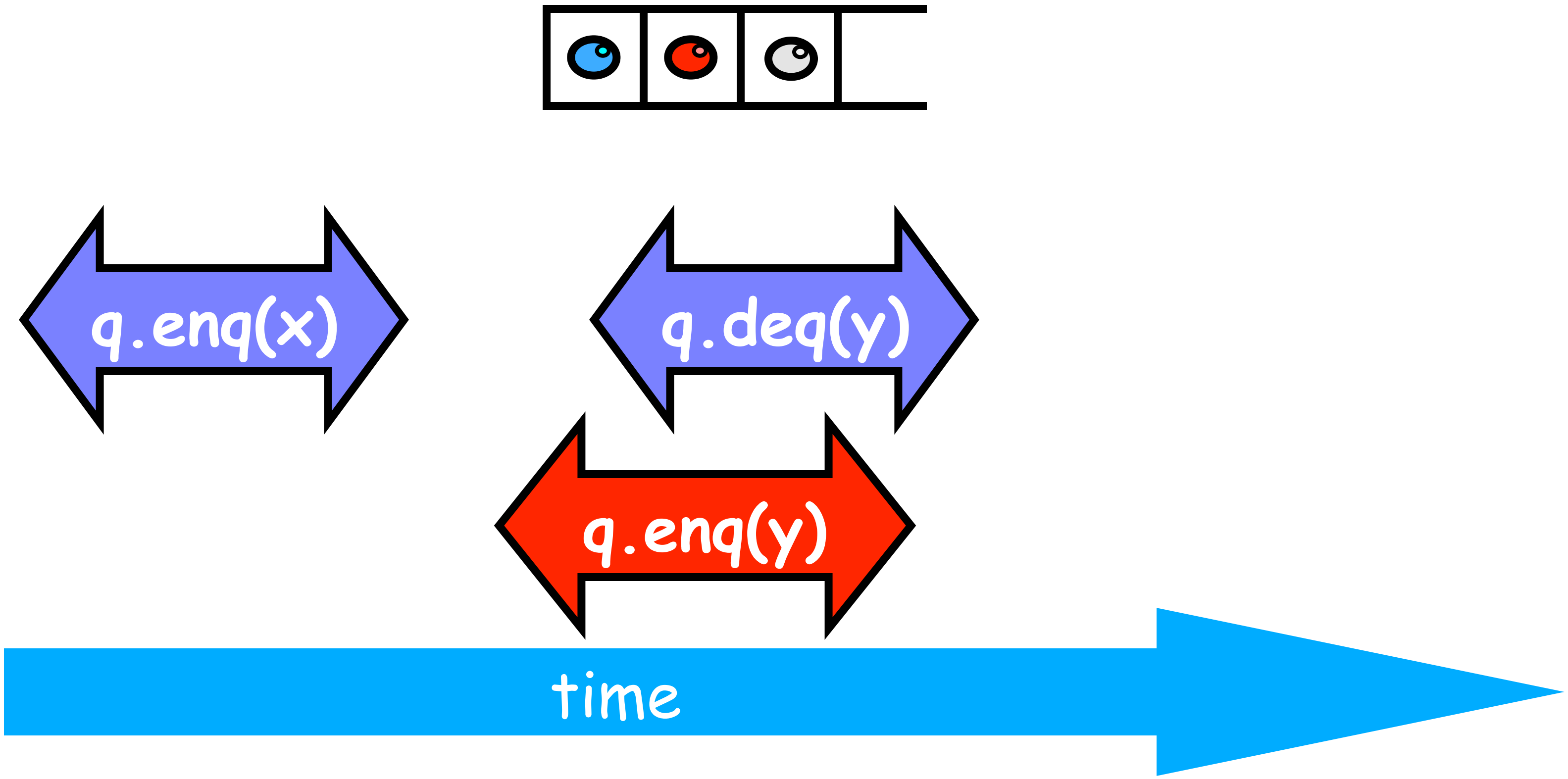
Example 1

- What if we choose other enq linearization points?

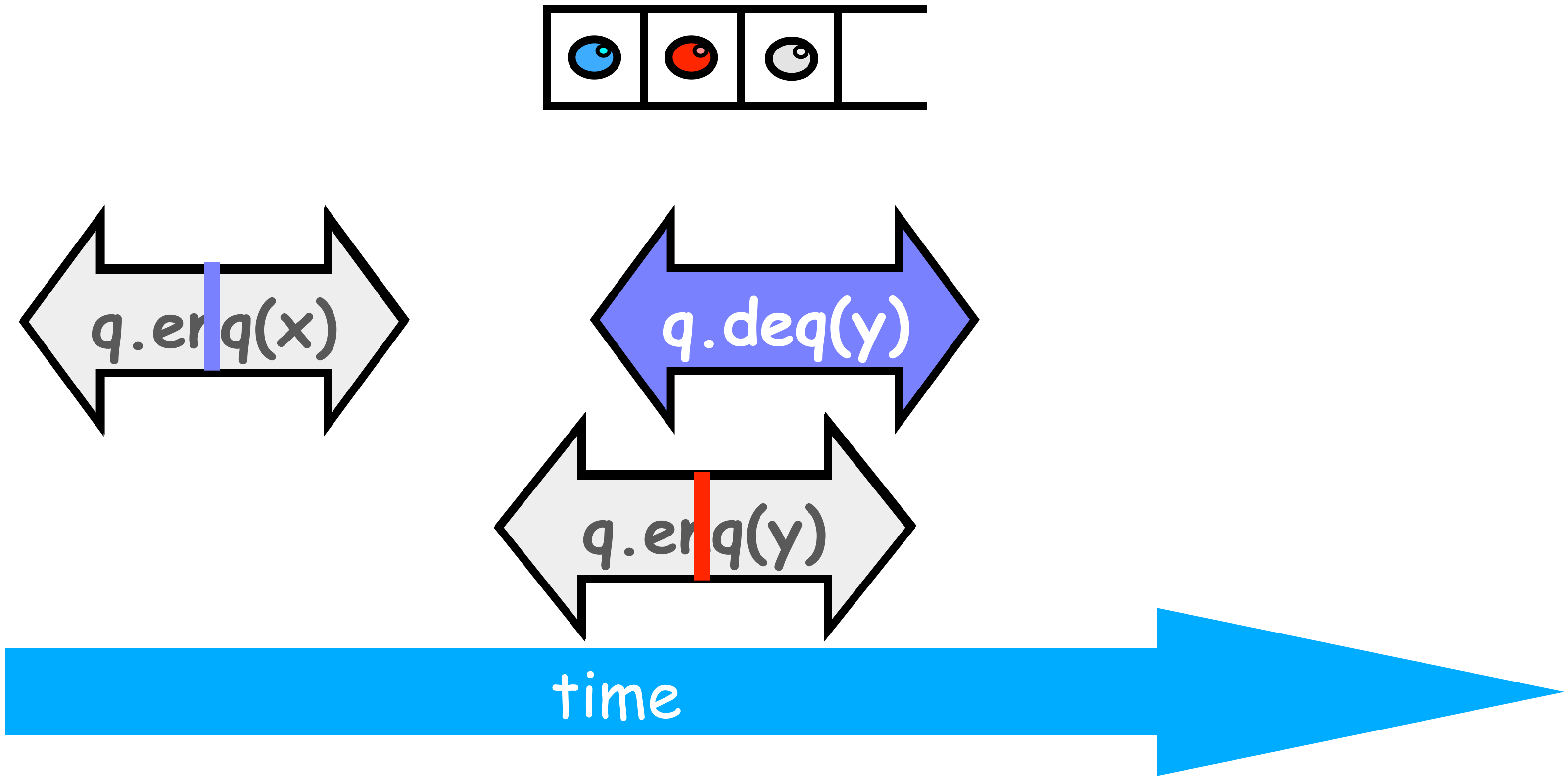


Example 2

- Assume the following execution history.
- Is it linearizable?

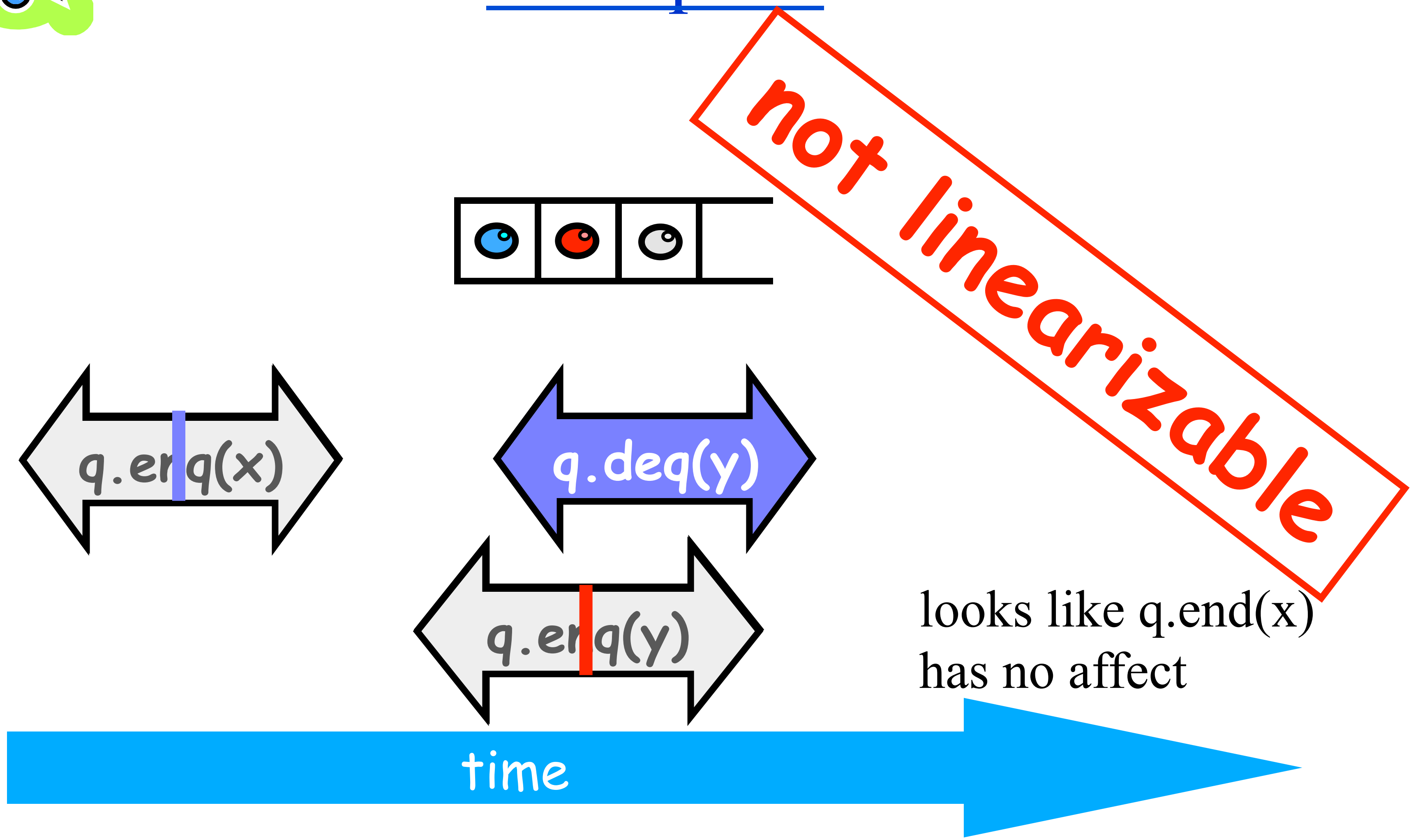


Example 2



(2/3)

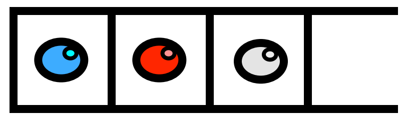
Example 2

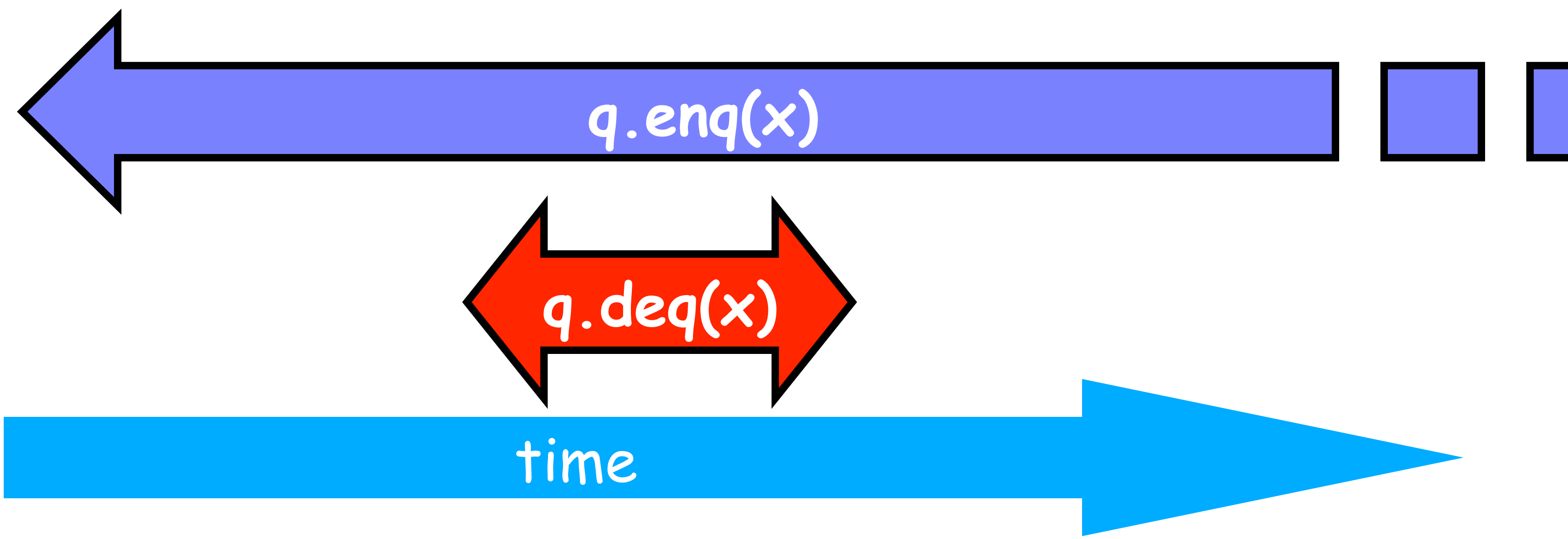


looks like q.end(x)
has no affect

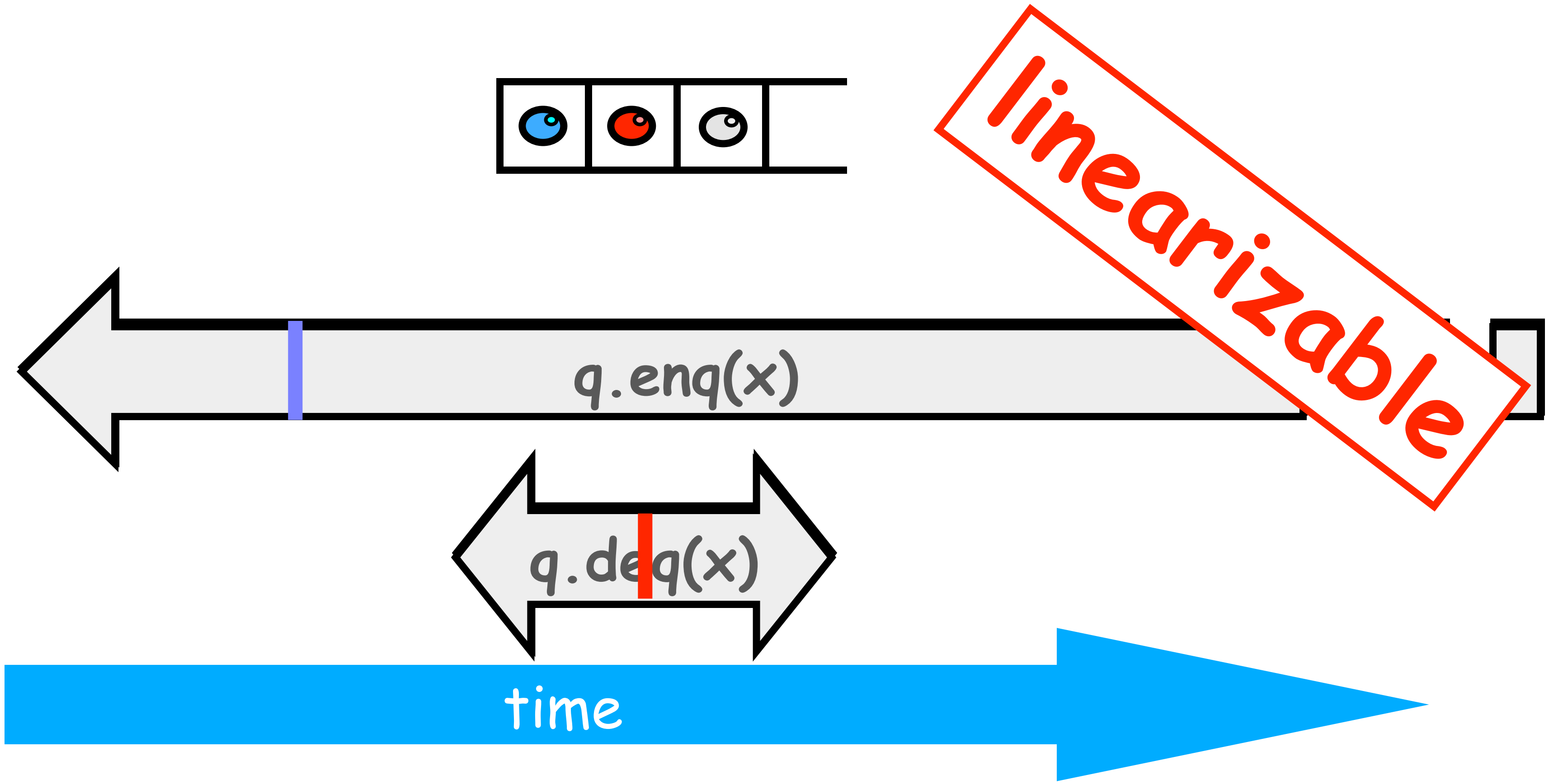
Example 3

- Is this execution history linearizable?





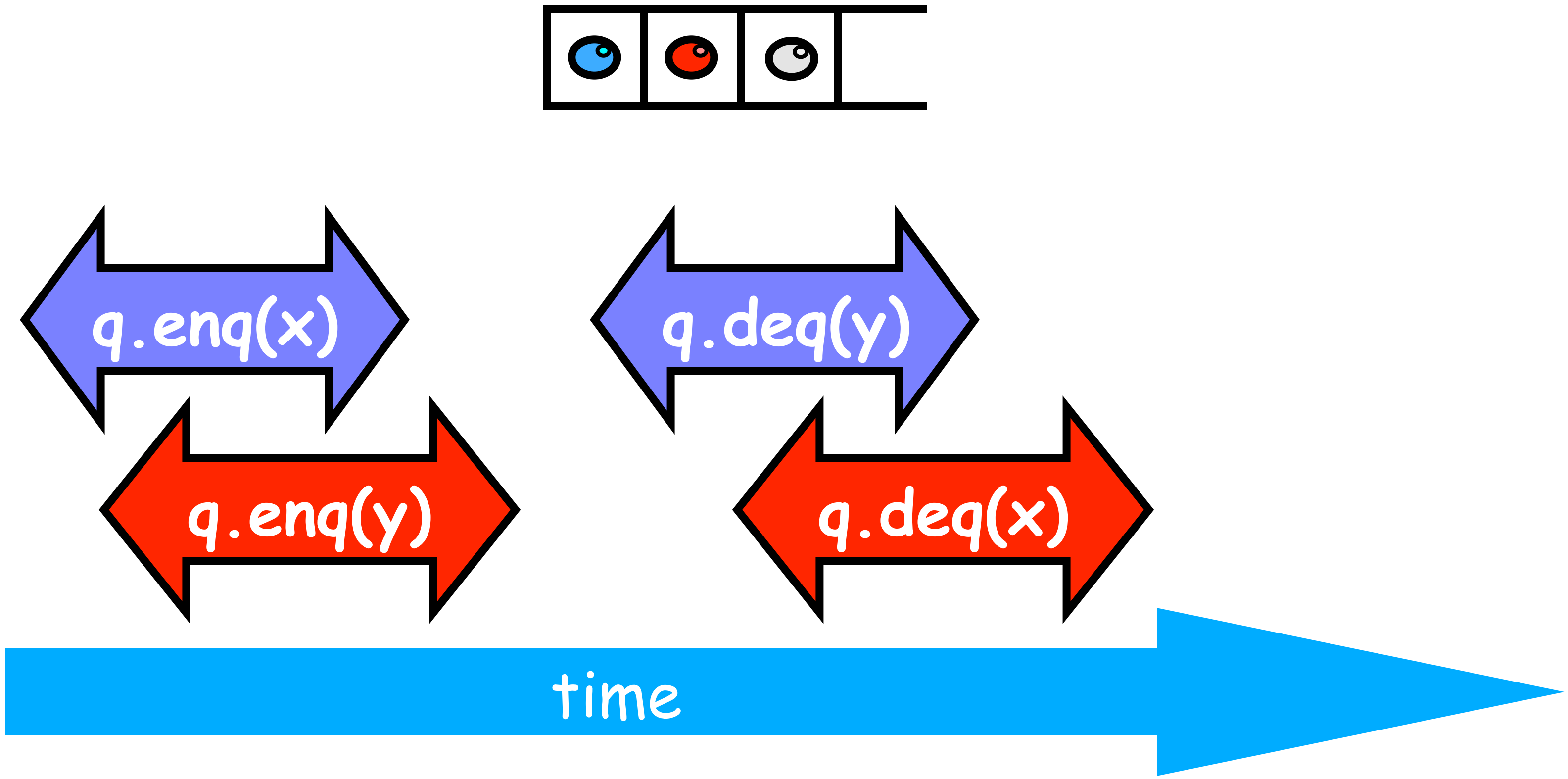
Example 3



(2/2)

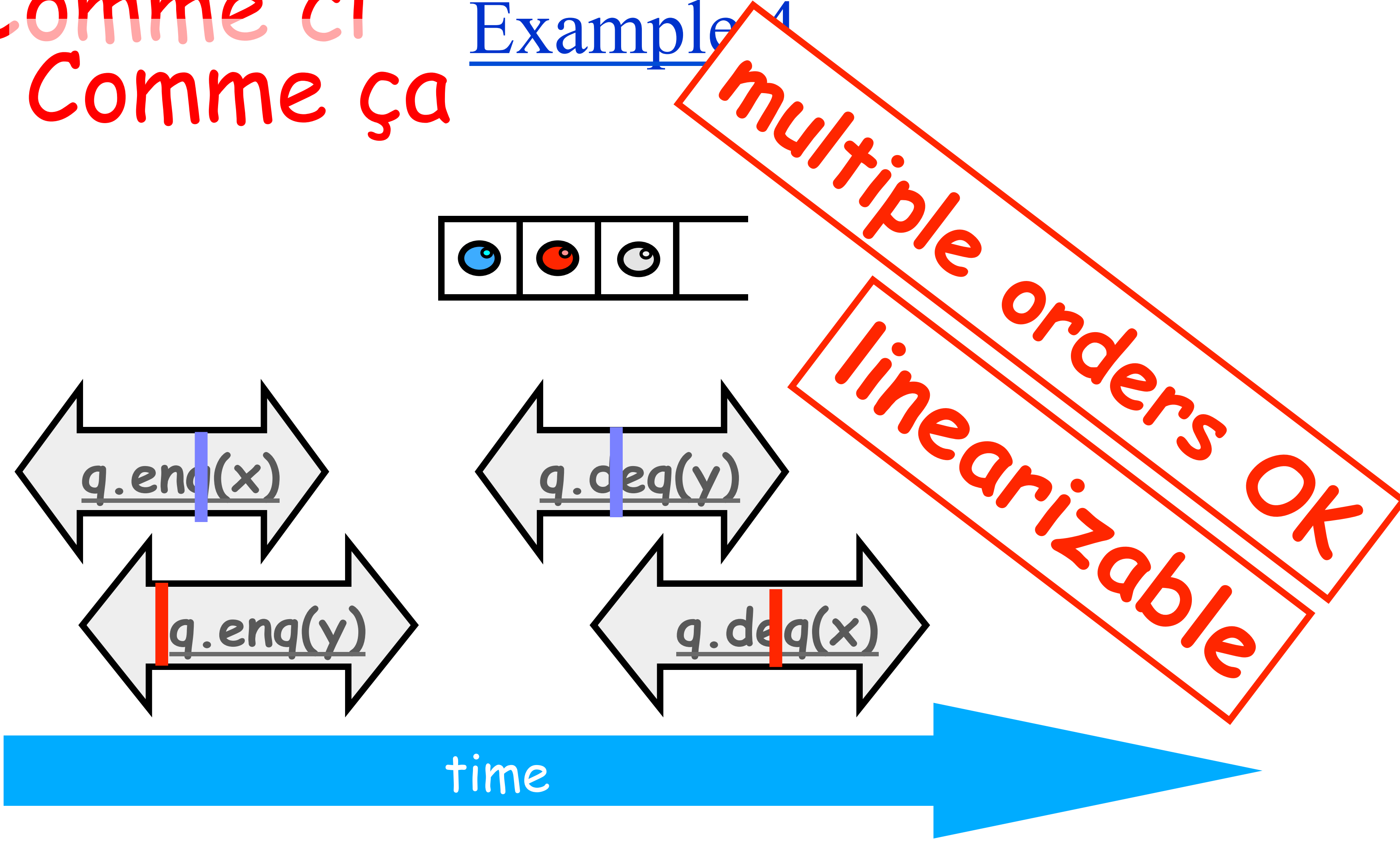
Example 4

- Is this history linearizable?

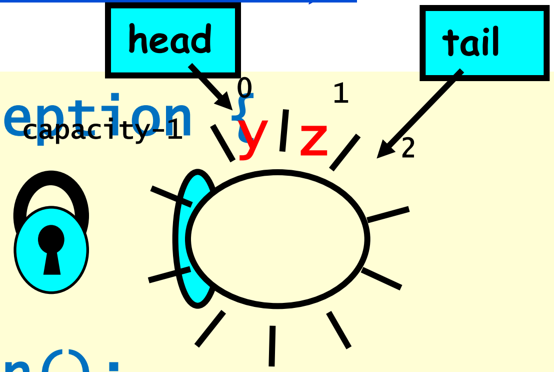


Comme ci
Comme ça

Example 4



Reasoning About Lineraizability: Locking (Lock-based Queue Example)




```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

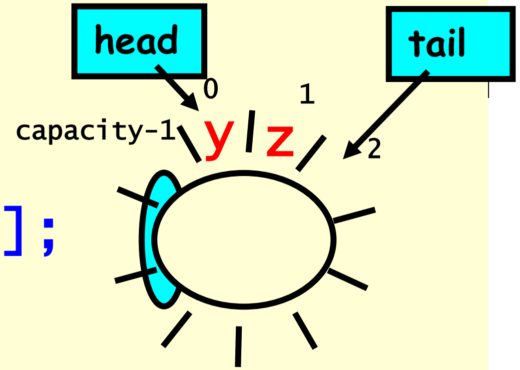
Reasoning About Lineraizability: Locking

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Linearization points are when locks are released

More Reasoning: Wait-free

```
public class WaitFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    if (tail-head == capacity) throw
         new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```



More Reasoning: Wait-free

```
public clas  aitFreeQueue {

  int          ail =
  it           new Ob

          id enq(Item x) {
       (tail-head == capacity) throw
           new FullException();
     items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
           new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```

Linearization order is order head and tail fields modified

Remember that there is only one enqueuer and only one dequeuer

Talking About Executions

- Why?
  - Can't we specify the linearization point of each operation without describing an execution?
- Not Always
  - In some cases, linearization point depends on the execution

Strategy

- Identify one atomic step where method "happens"
  - Critical section
  - Machine instruction
- Doesn't always work
  - Might need to define several different steps for a given method

Linearizability: Summary

- Linearization points: Operation takes effect instantaneously between invocation and response
- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being "atomic"
- **Helps to describe, develop and use a concurrent object, and reason about its properties using a sequential specification**
- **Linearization is a local property**
- **Locality implies composability: a composition of linearizable objects is linearizable**
- Good for high level objects

Why Does Composability Matter?

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects

Correctness via Linearizability

- Reason about correctness using *linearizability*
  - Find linearization points
  - Linearize the concurrent execution to the sequential one and check the latter for correctness
- **Linearizability is a correctness condition for concurrent objects.**
- **Object is correct if its “sequential” behavior (i.e. linearized execution) is correct**

Progress

- There might be implementation whose methods are lock-based (deadlock-free)
- There might be implementation whose methods did not use locks (lock-free)
- How do they relate?

Progress Conditions

- Informal definitions
- ***Deadlock-free***: some thread trying to acquire the lock eventually succeeds.
- ***Starvation-free***: every thread trying to acquire the lock eventually succeeds.
- ***Lock-free***: some thread calling a method eventually returns.
- ***Wait-free***: every thread calling a method eventually returns.

Progress Conditions

	Non-Blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free

Concurrent Object Case Study: A Concurrent Set based on Linked List

Concurrent Objects

- Shared objects are potential sequential bottlenecks
- Scalable concurrent data structures: how to?
- Adding threads should not lower throughput of object methods
  - Contention effects
  - Mostly fixed by using Queue locks
- Should increase throughput
  - Not possible if inherently sequential
  - Surprising things are parallelizable

Concurrent Objects: Design Space

- **Lock-based** (synchronized) vs **Lock-free** (non-synchronized)
  - It's about mutual exclusion
- **Blocking** vs **Non-Blocking**
  - It's about condition synchronization
- **Wait-Free (lock-free and non-blocking)**
  - Each call takes a finite number of steps
- **Fine-grained** vs **coarse-grained** synchronization
  - It's about granularity of synchronized objects
  - Fine-grained vs coarse-grained locking

Coarse-Grained Lock-Based Synchronization

- Synchronized class: Take a sequential implementation of a class, add a scalable lock field, and ensure that each method call acquires and releases that lock.
- Each method locks the object
  - Avoid contention using queue locks
  - Easy to reason about
    - In simple cases
  - Standard Java model: Monitors
    - **Synchronized** blocks and methods
- So, are we done with concurrent objects?

Coarse-Grained Lock-Based Synchronization (cont'd)

- Sequential bottleneck
  - Threads "stand in line"
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse
- So why even use a multiprocessor?
  - Well, some apps inherently parallel …

“Traffic Jam”

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And “eats the big muffin” (stops running)
    - Cache miss, page fault, descheduled …
  - Everyone else using that lock is stuck!
  - Need to trust the scheduler….

Four Patterns for Concurrent Objects

1. Fine-Grained Synchronization
2. Optimistic Synchronization
3. Lazy Synchronization (not considered here)
4. Lock-Free Synchronization

- For ***highly-concurrent*** objects
  - Concurrent access
  - More threads, more throughput
- Course-Grained Synchronization (the Monitor model) was considered earlier

First: Fine-Grained Synchronization

- Instead of using a single lock …
- Split object into
  - Independently-synchronized components
- Methods conflict when they access
  - The same component …
  - At the same time

Second: Optimistic Synchronization

- Search without locking …
- If you find it, lock and check (validate)…
  – OK: we are done
  – Oops: start over
- Evaluation
  – Usually cheaper than locking, but
  – Mistakes are expensive

Third: Lazy Synchronization

- Rather object-specific
  - Might work on collections, e.g. list
- Postpone hard work, e.g. physical removals
- Removing components is tricky
  - *Logical* removal
    - Mark component to be deleted
  - *Physical* removal
    - Do what needs to be done

Fourth: Lock-Free Synchronization

- Don't use locks at all
  - Use **CompareAndSet()** & relatives …
  - Transactional style: load shared state, modify locally, then CAS: Compare And Set

```
do {
  old = state.get();
  new = modify(old);
} until (state.compareAndSet(old, new))
```

- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead

Case Study: Concurrent Set Based on Linked List

- To illustrate some of the patterns…
- Using a **list**-based **Set**
  - Common application
  - Building block for other apps

- Consider three implementation patterns
  1. Course-grained locking (the Monitor model)
  2. Fine-grained locking
  3. Lock-free

  - For other patterns – optimistic and lazy synchronizations – see the book ***The Art of Multiprocessor Programming***, by Maurice Herlihy, and Nir Shavit

The Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - `add(x)` put `x` in set
  - `remove(x)` take `x` out of set
  - `contains(x)` tests if `x` in set

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

Add item to set

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(Tt x);
}
```

Remove item from set

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

Is item in set?

List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

item of interest

List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

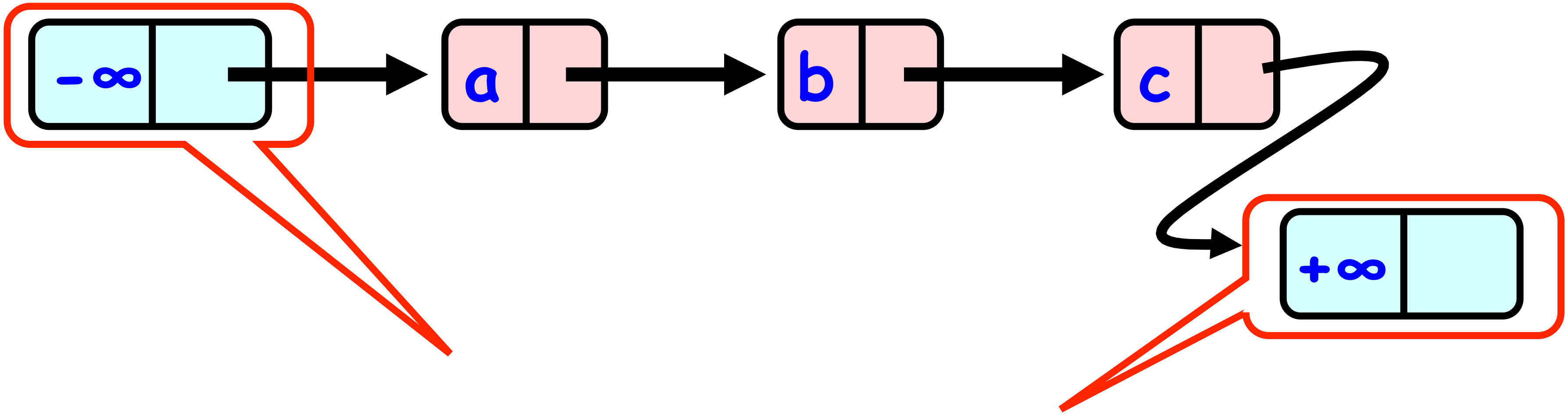
Usually hash code

List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

Reference to next node

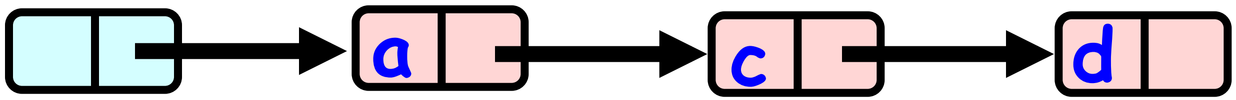
The List-Based Set



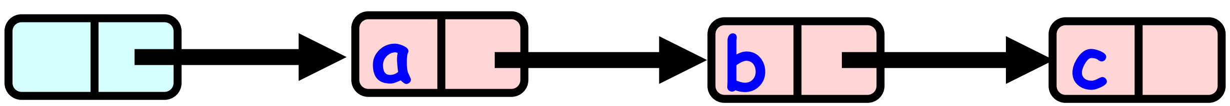
Sorted using keys with Sentinel nodes (min & max possible keys)

Sequential List Based Set

Add()

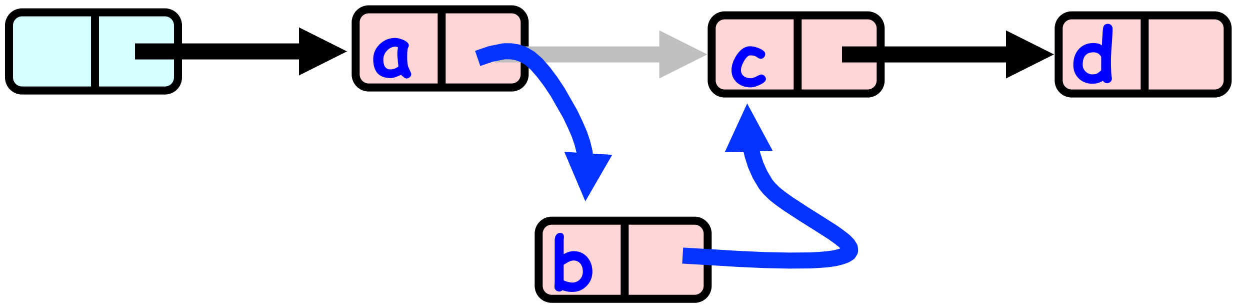


Remove()

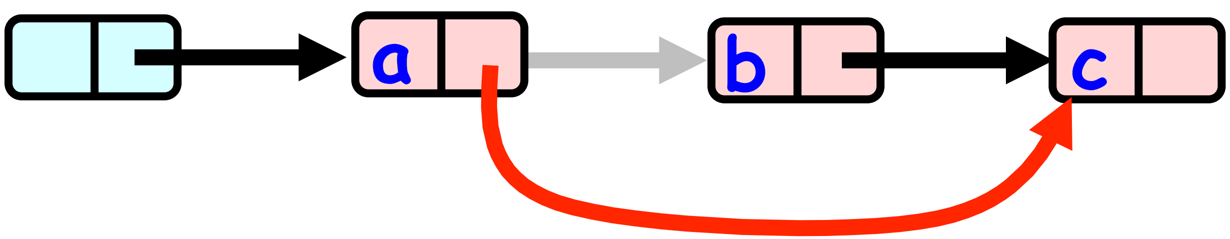


Sequential List Based Set

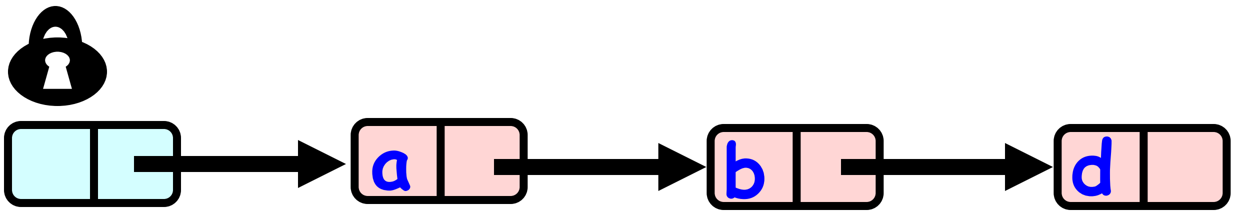
Add()



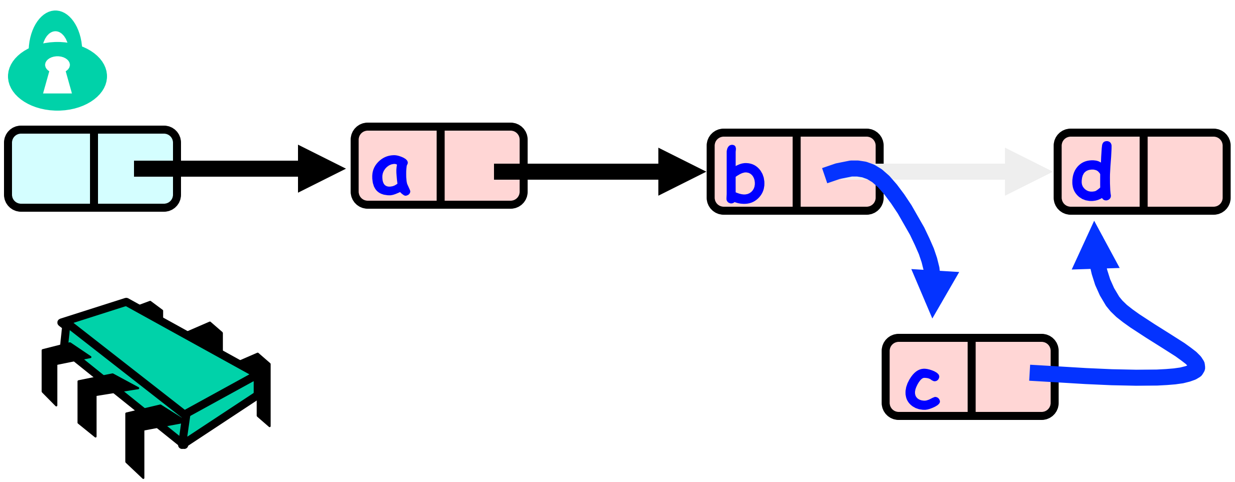
Remove()



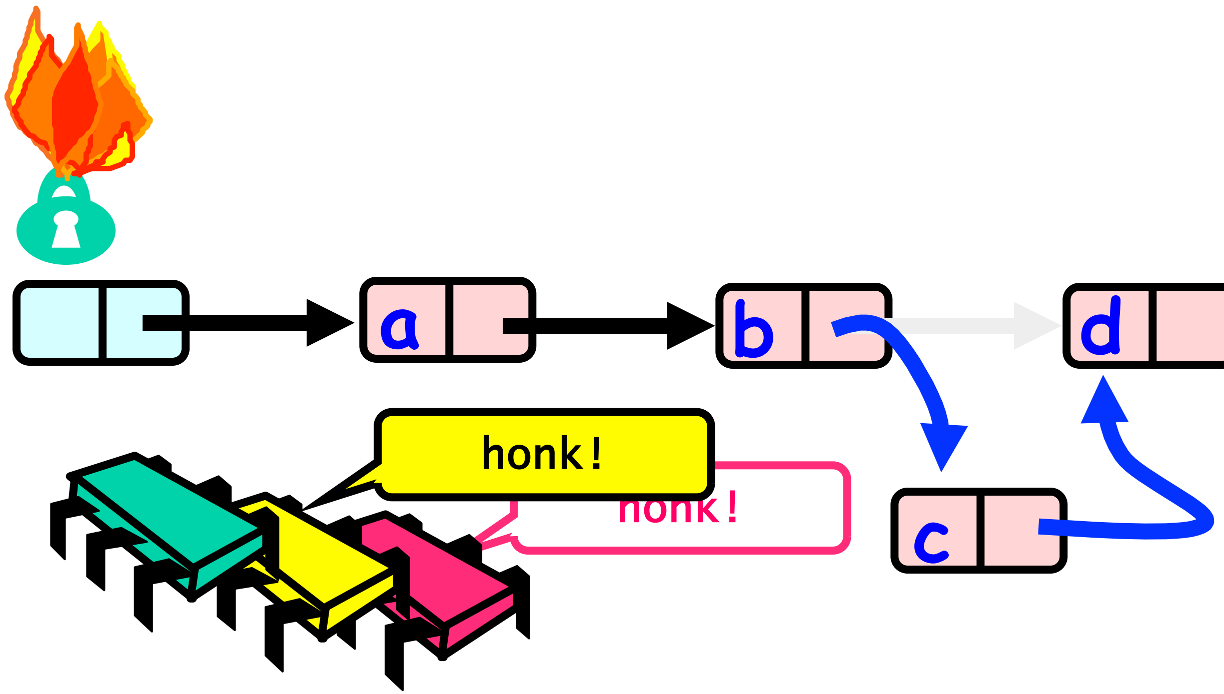
Coarse-grained Locking: Coarse-Grained Synchronized Set



Coarse-grained Locking



Coarse-grained Locking



Simple but hotspot + bottleneck

Coarse-grained Locking

- Easy, same as synchronized methods
  - **“One lock to rule them all …”**
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention
  - Queue locks help
  - But bottleneck still an issue

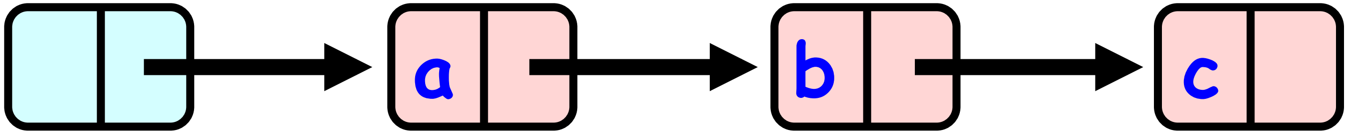
First Pattern: Fine-grained Locking

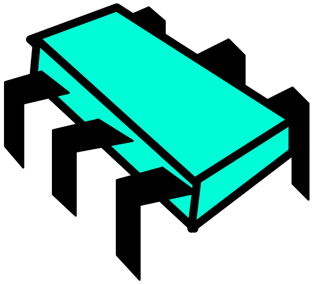
- Requires **careful** thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"
    - J.R.R. Tolkien, "The Fellowship of the Ring"
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Fine-grained Locking Idea

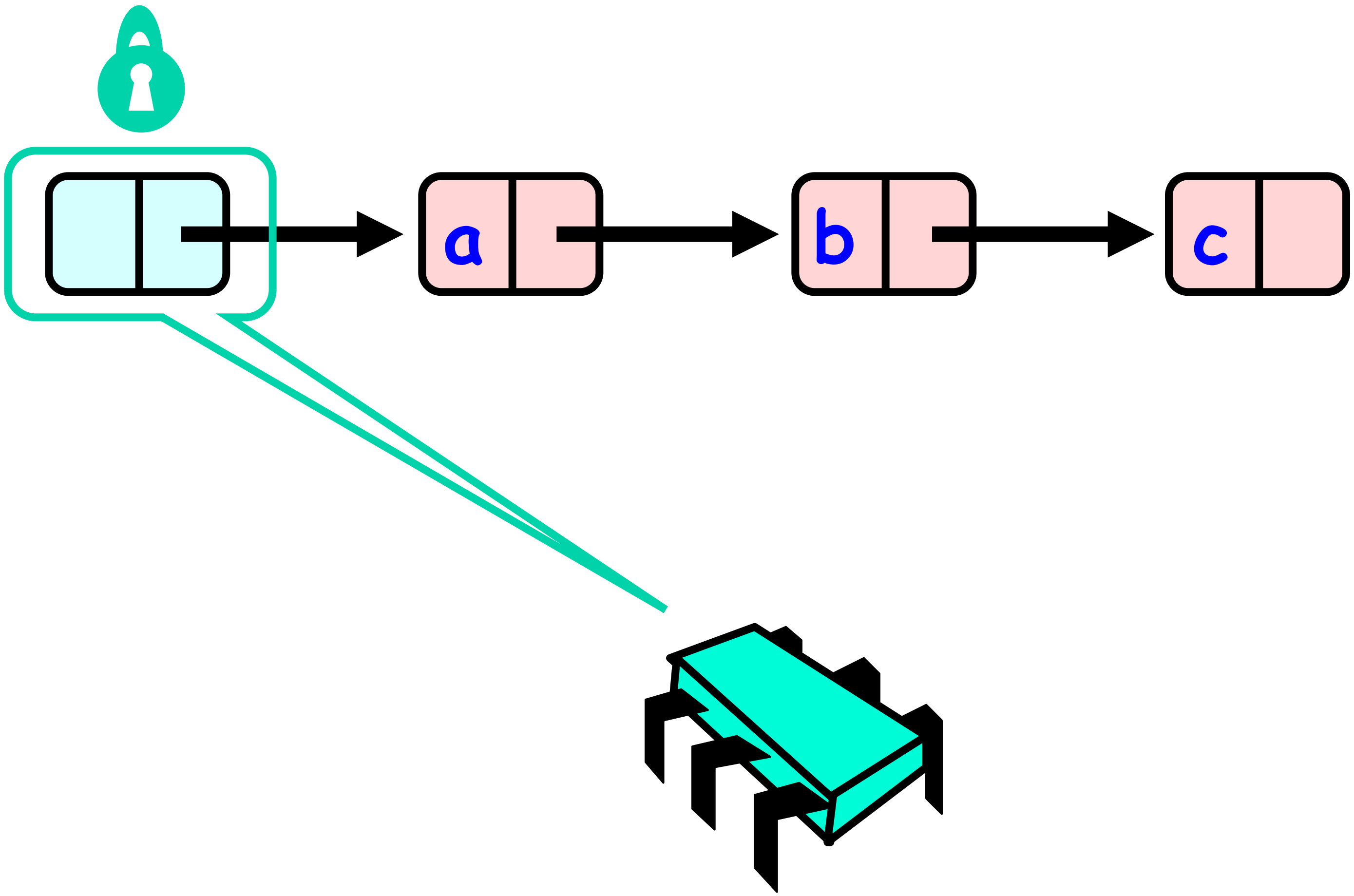
- Improve concurrency by locking individual entries, rather than locking the entire list.
- Instead of a single lock on the entire list, add a lock to each entry.
- As a thread traverses the list, it locks each entry when it first visits, and some time later releases it.
- Such fine-grained locking allows concurrent threads to traverse the list together in a pipelined fashion.

Hand-over-Hand Locking

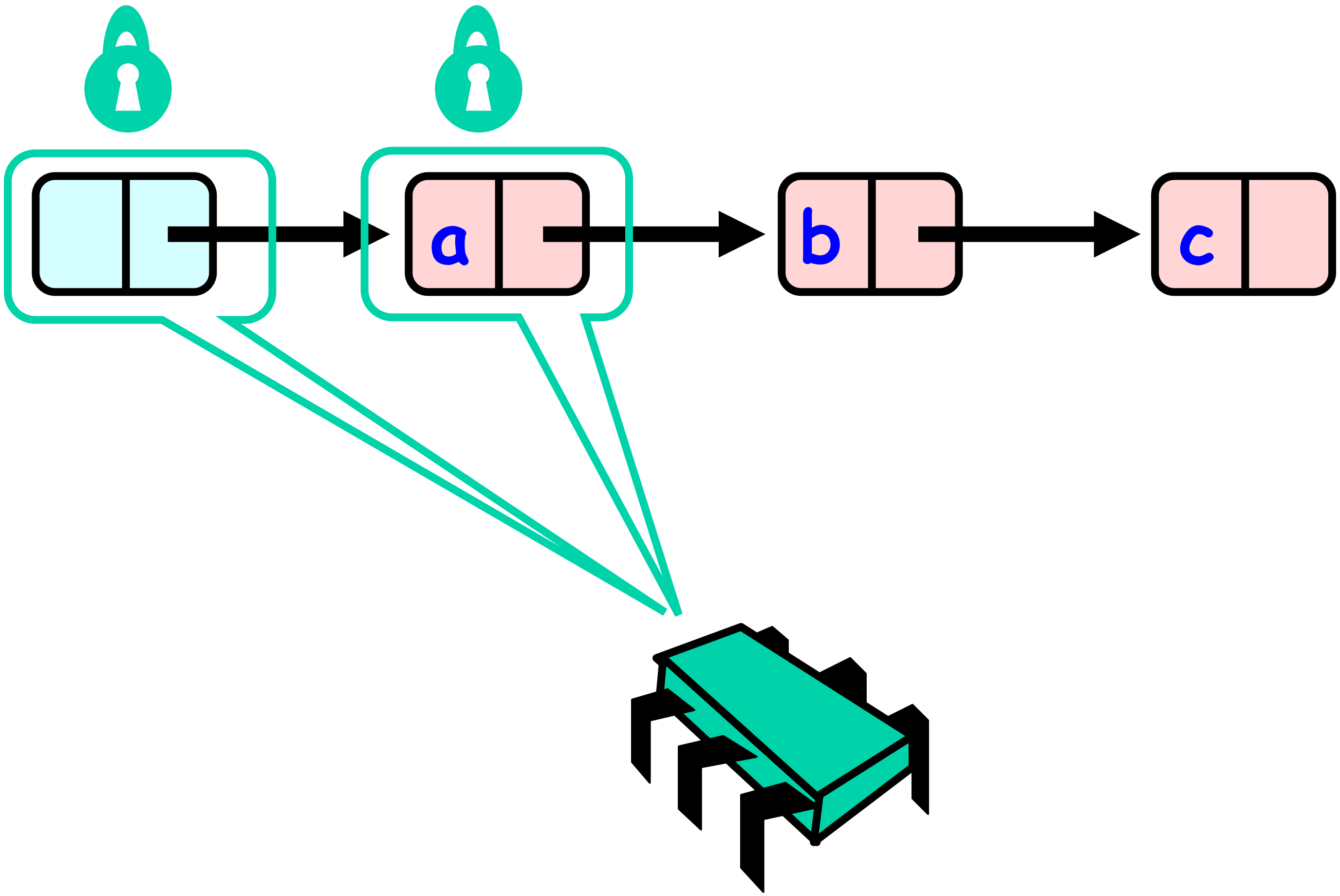




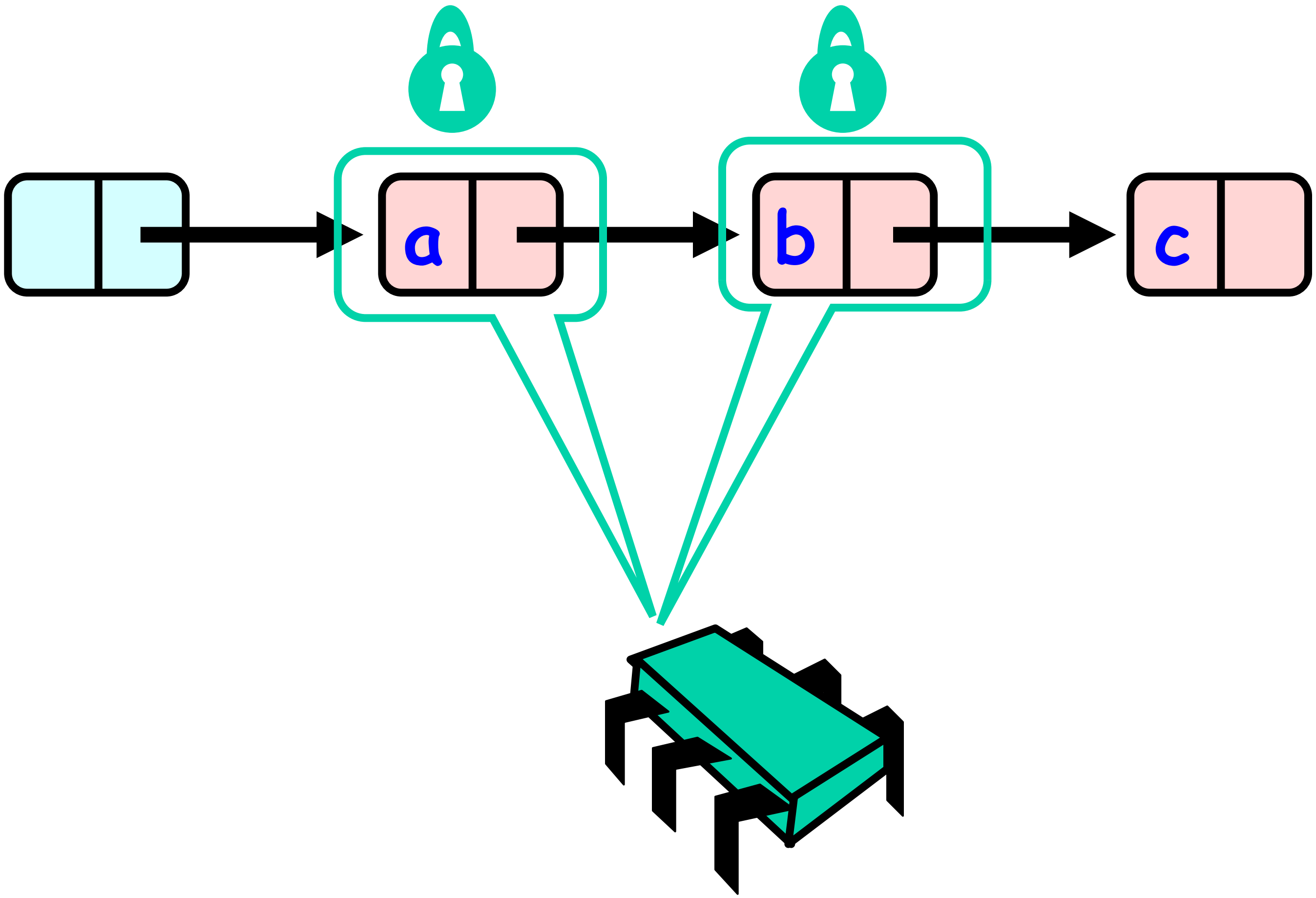
Hand-over-Hand Locking



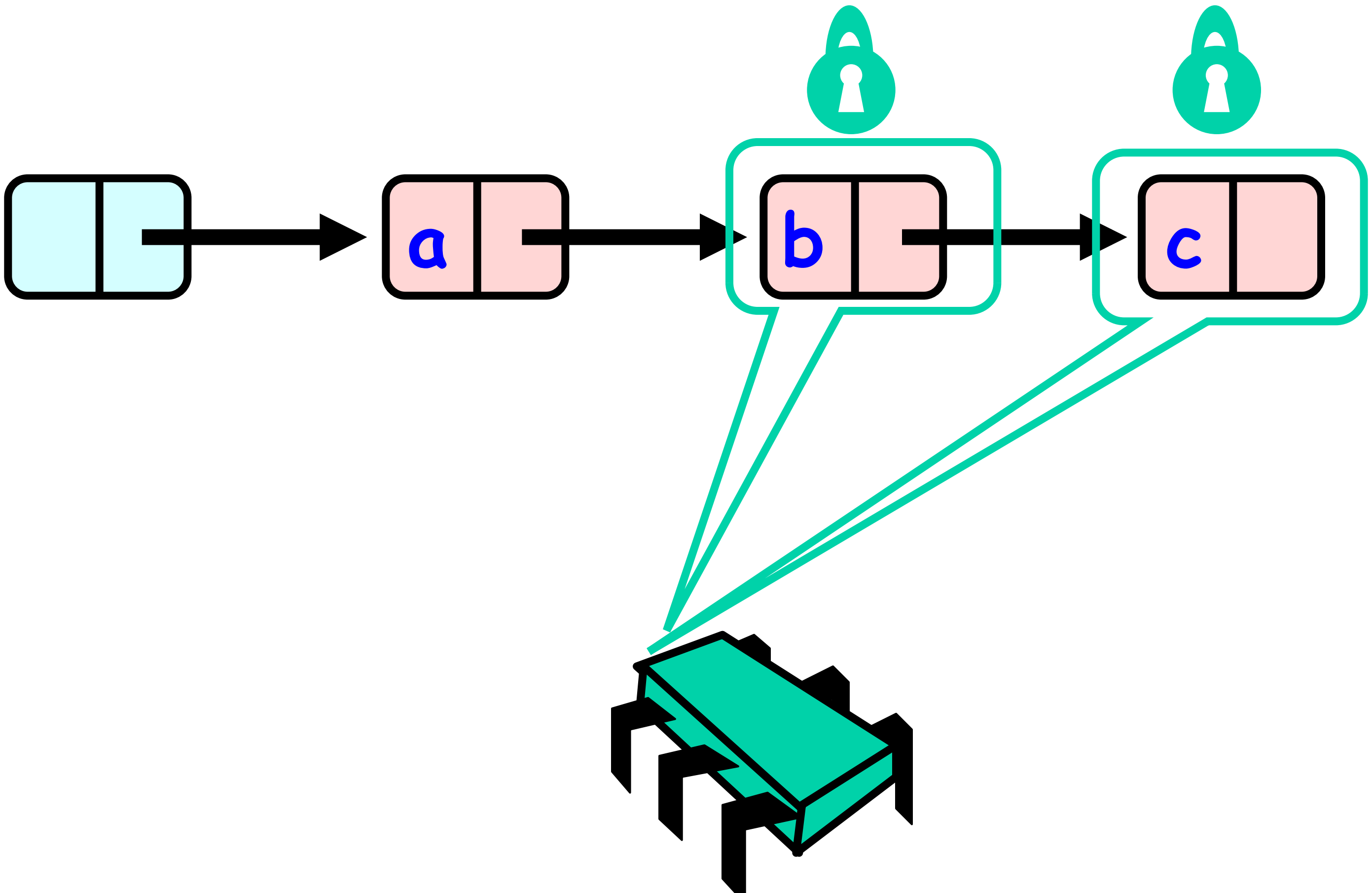
Hand-over-Hand Locking



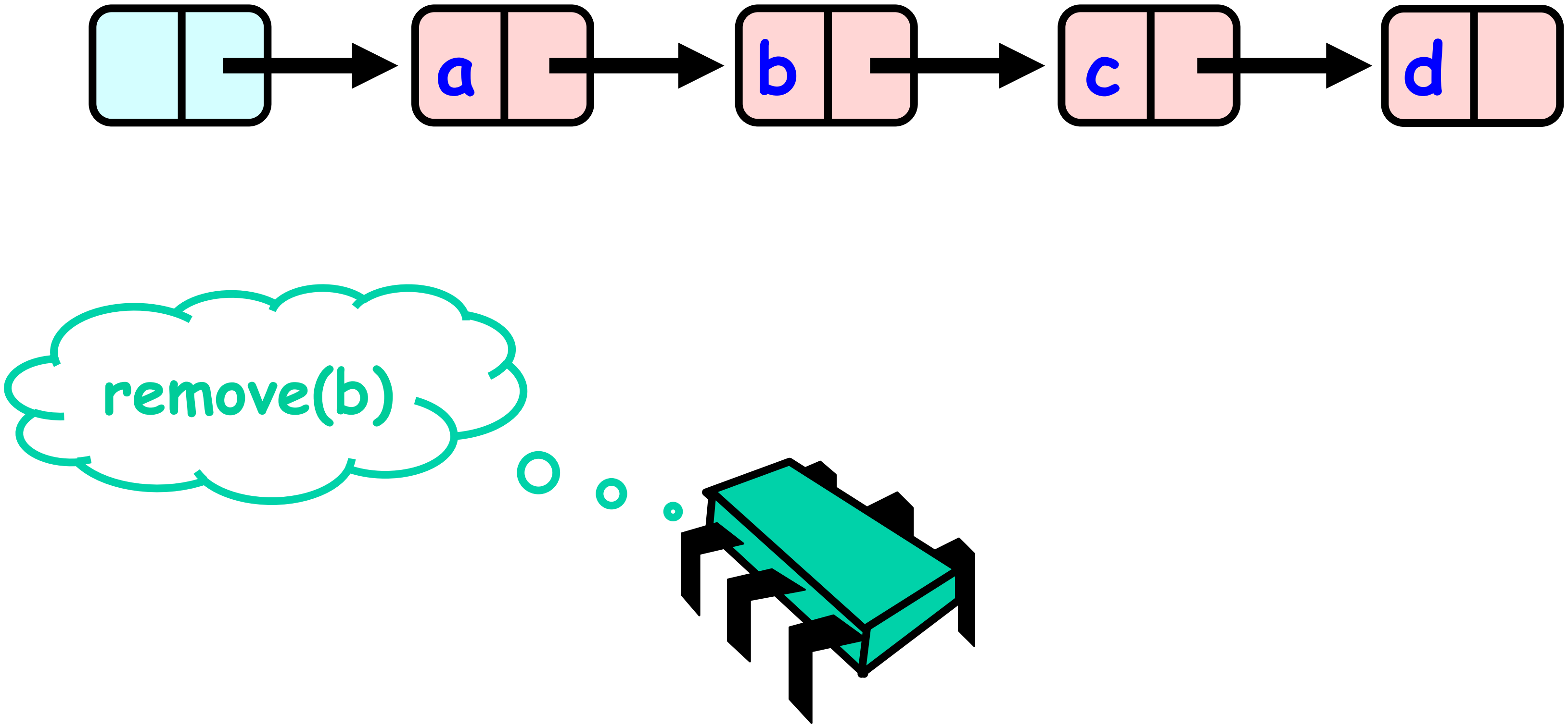
Hand-over-Hand Locking



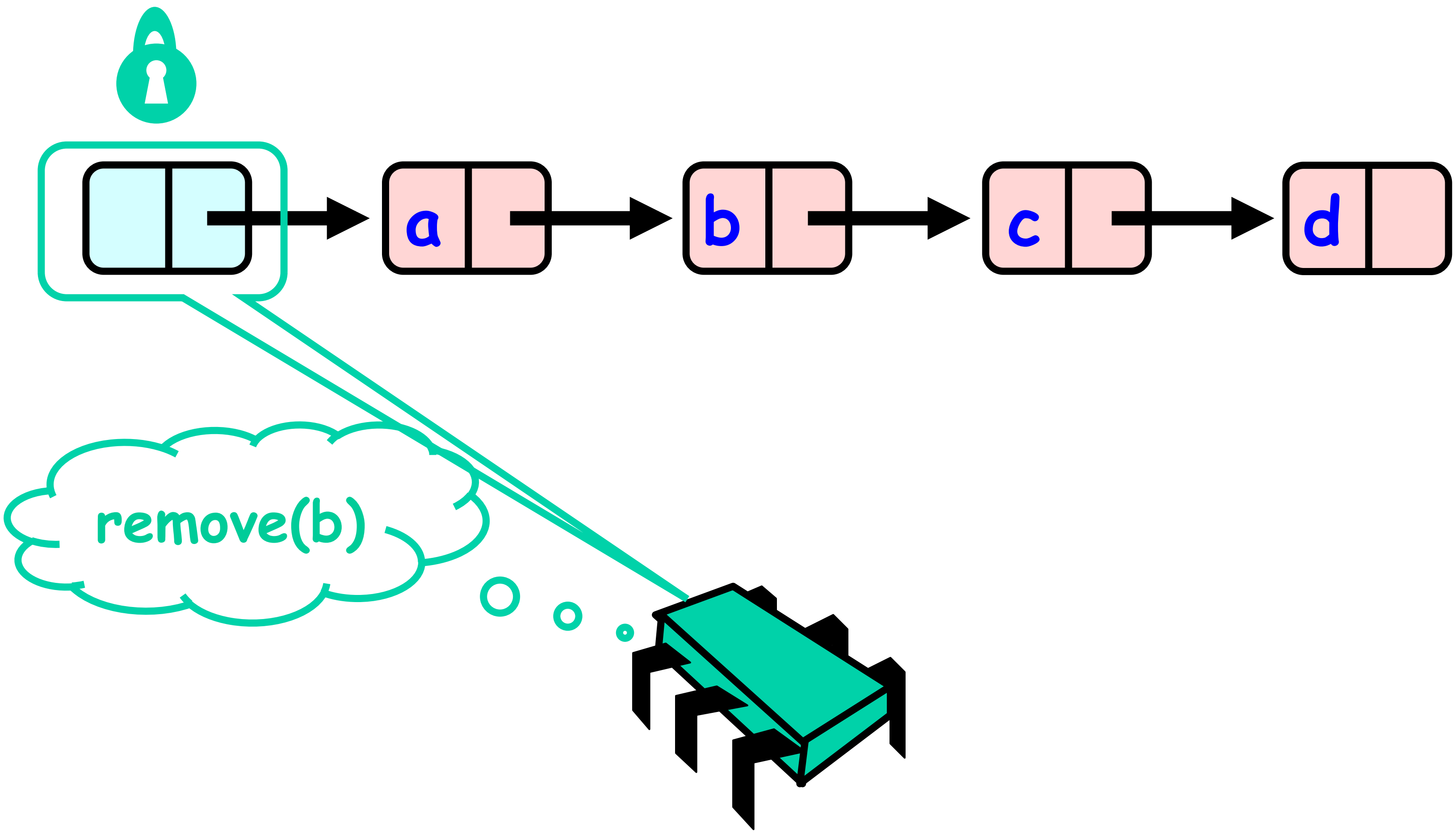
Hand-over-Hand Locking



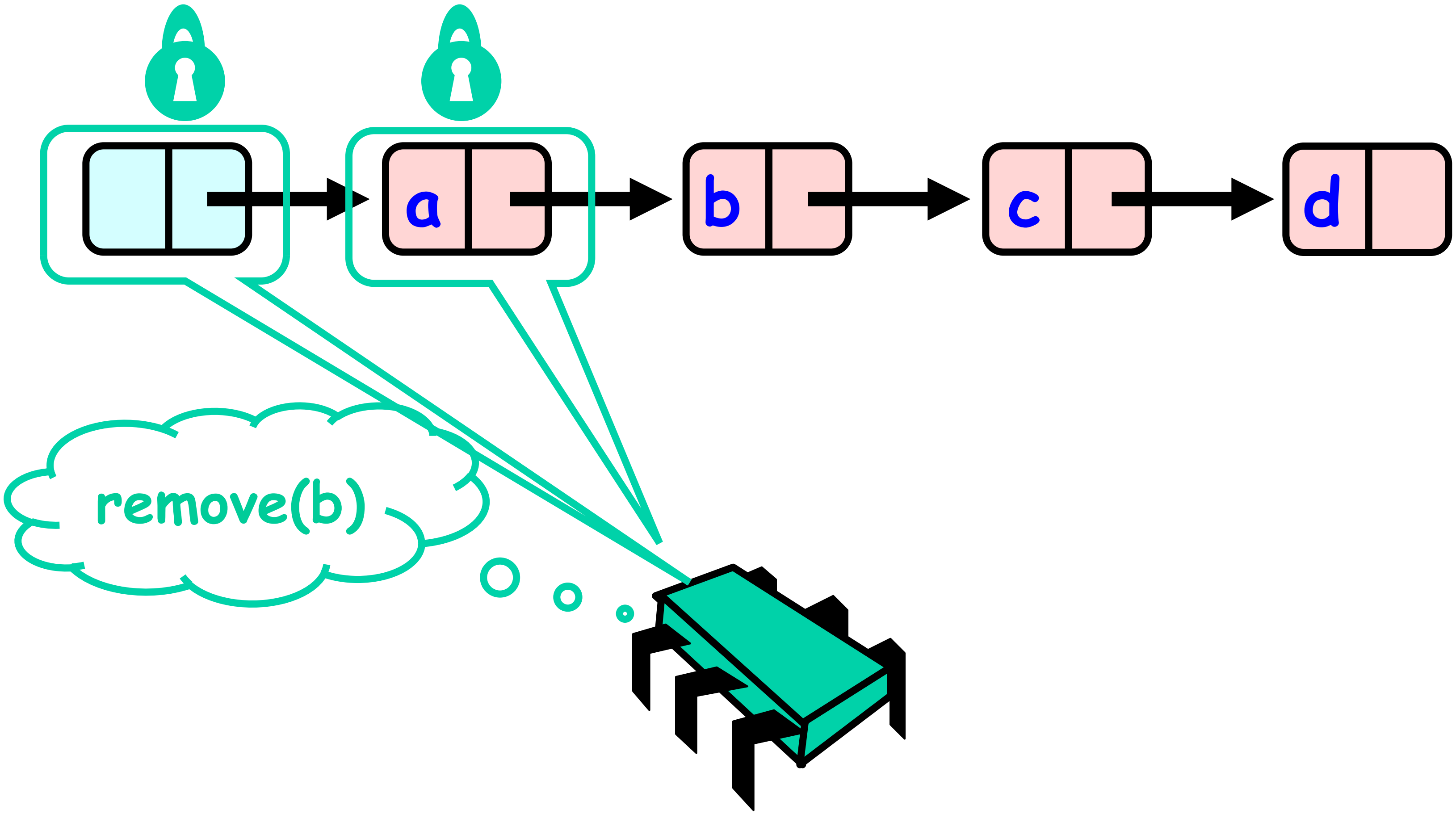
Removing a Node



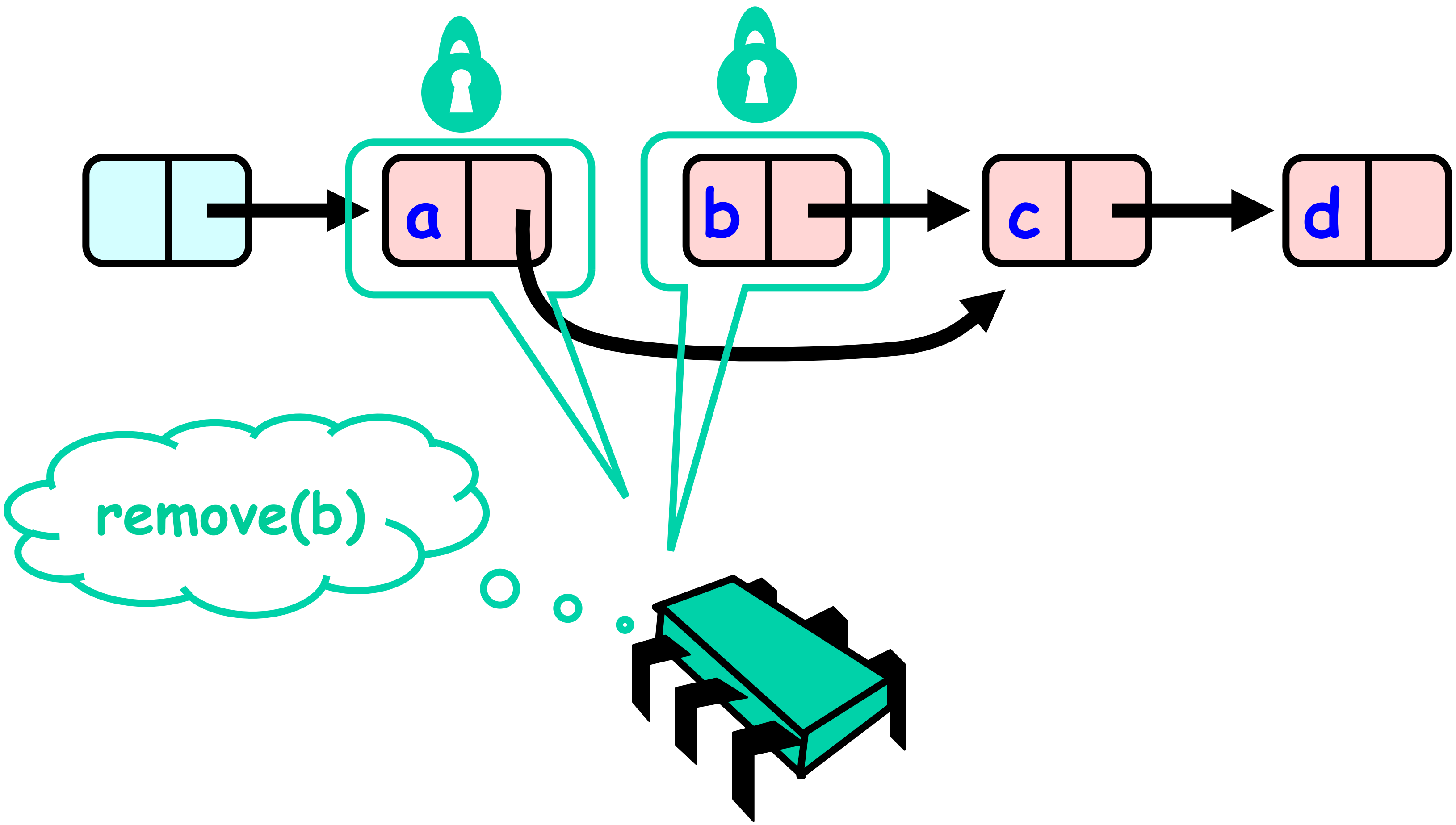
Removing a Node



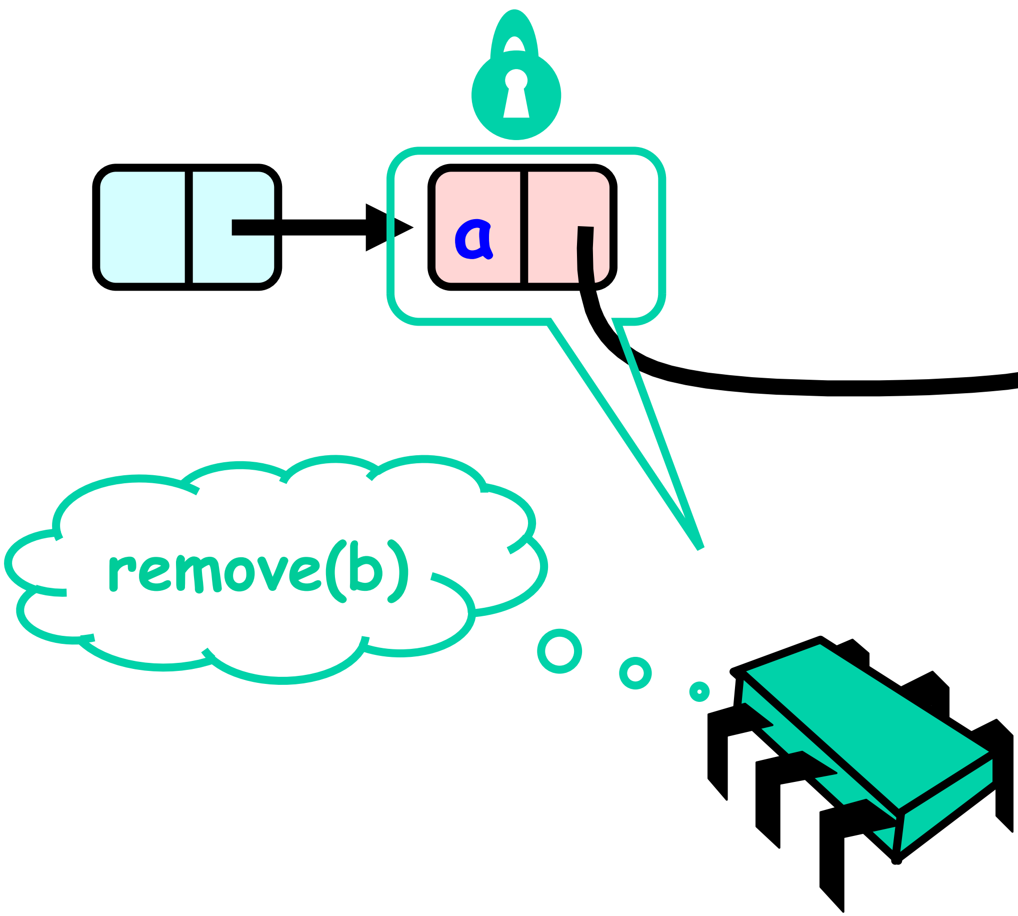
Removing a Node



Removing a Node



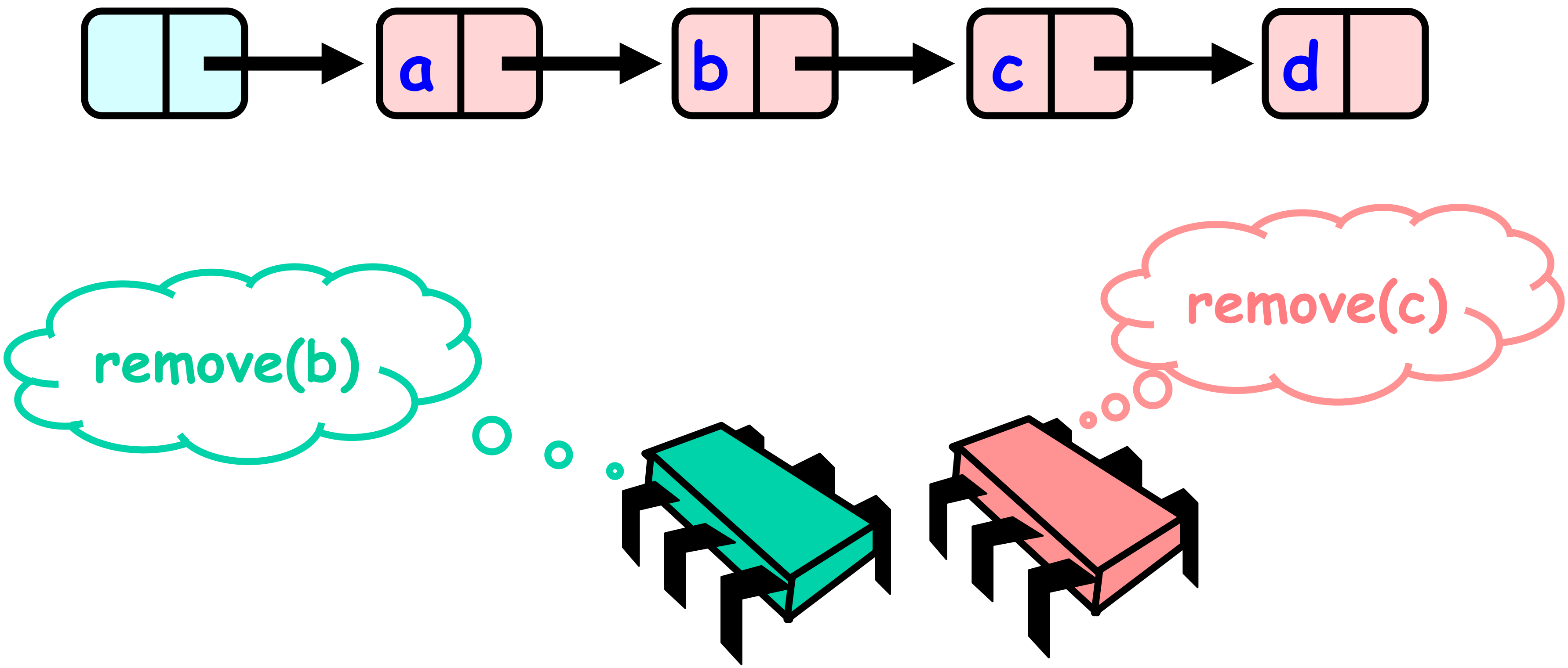
Removing a Node



Why do we need to always hold 2 locks?

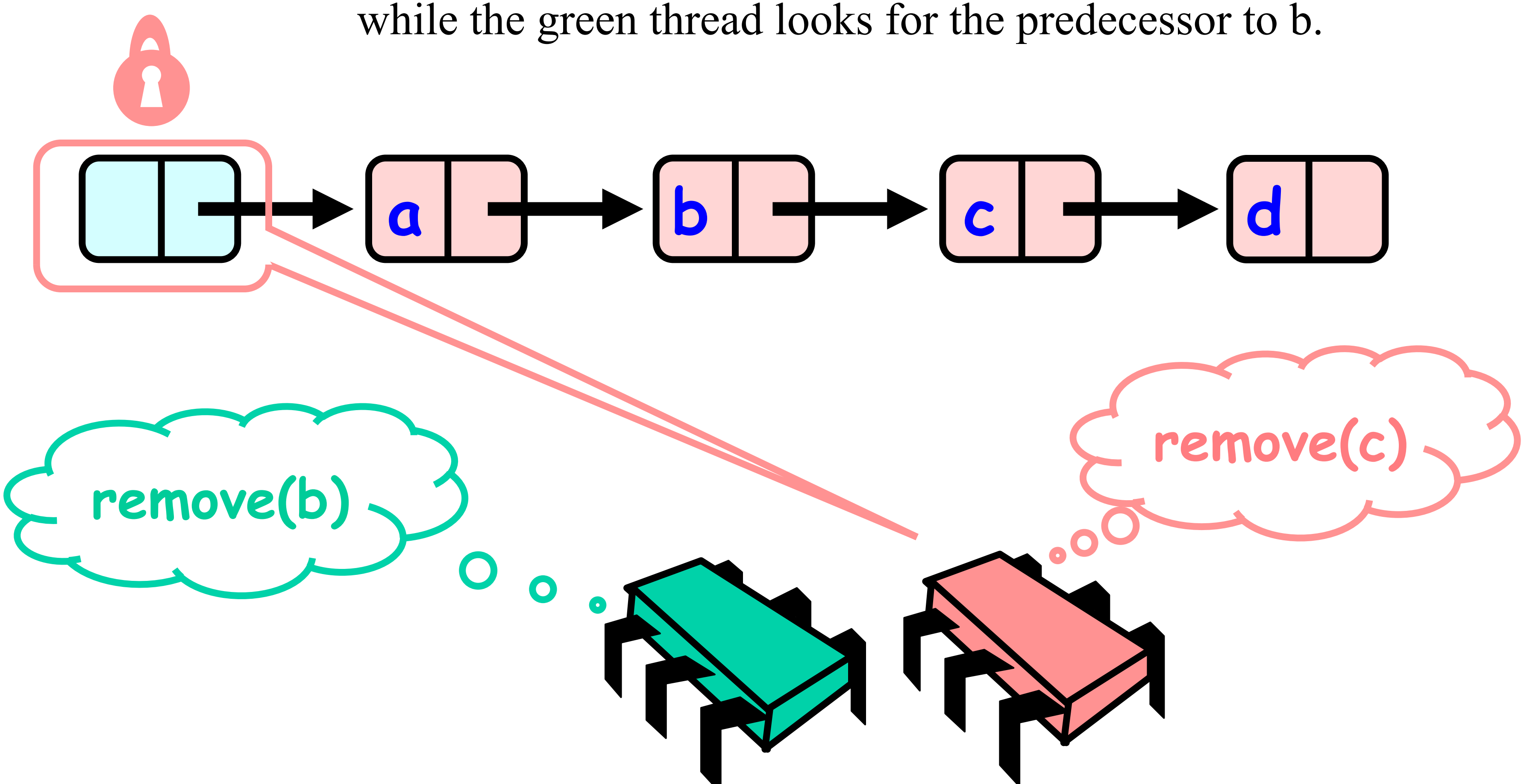
Concurrent Removes (each locks one node)

- Supposed we only locked one node and not two.
- Consider two concurrent threads that want to remove adjacent nodes.

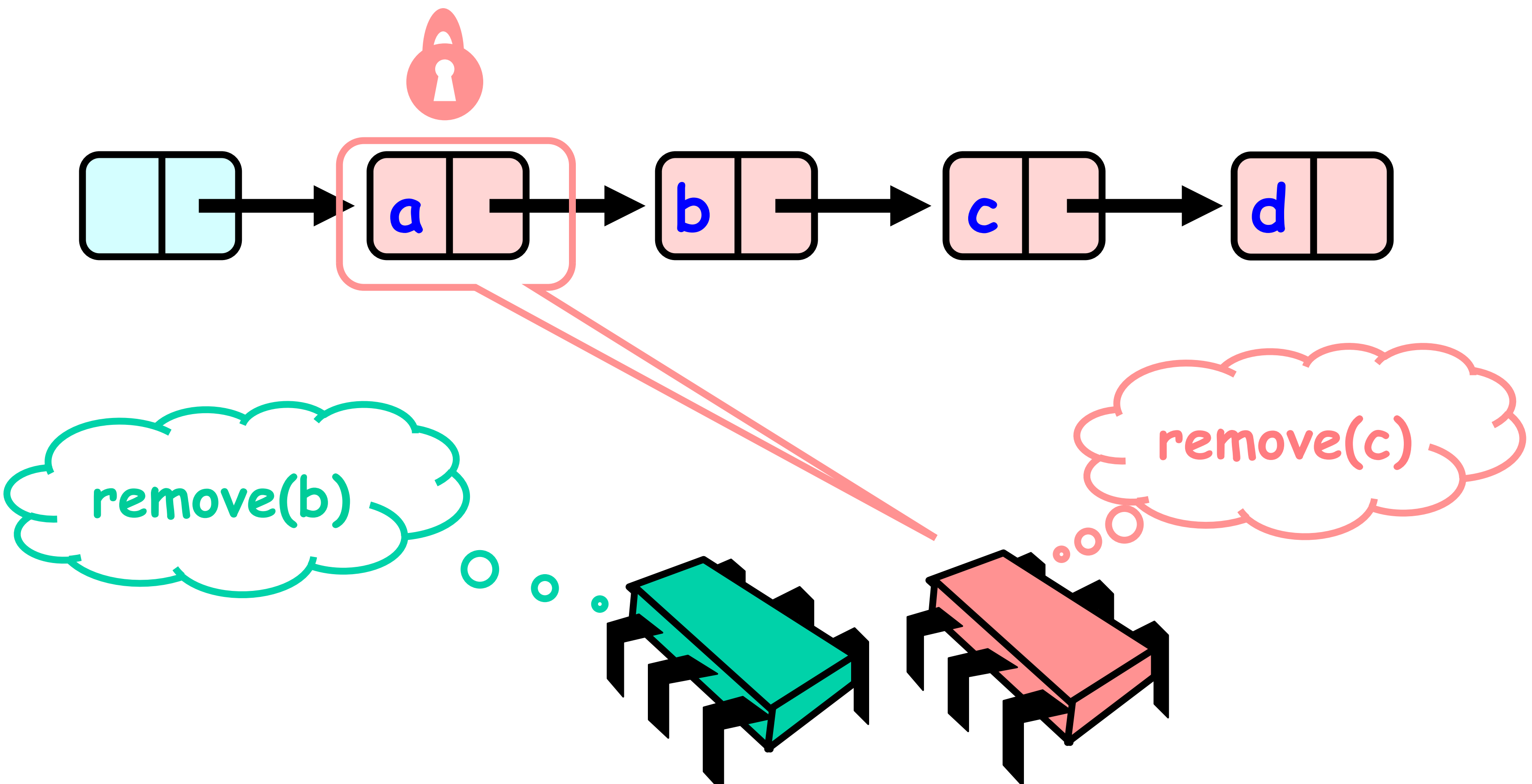


Concurrent Removes

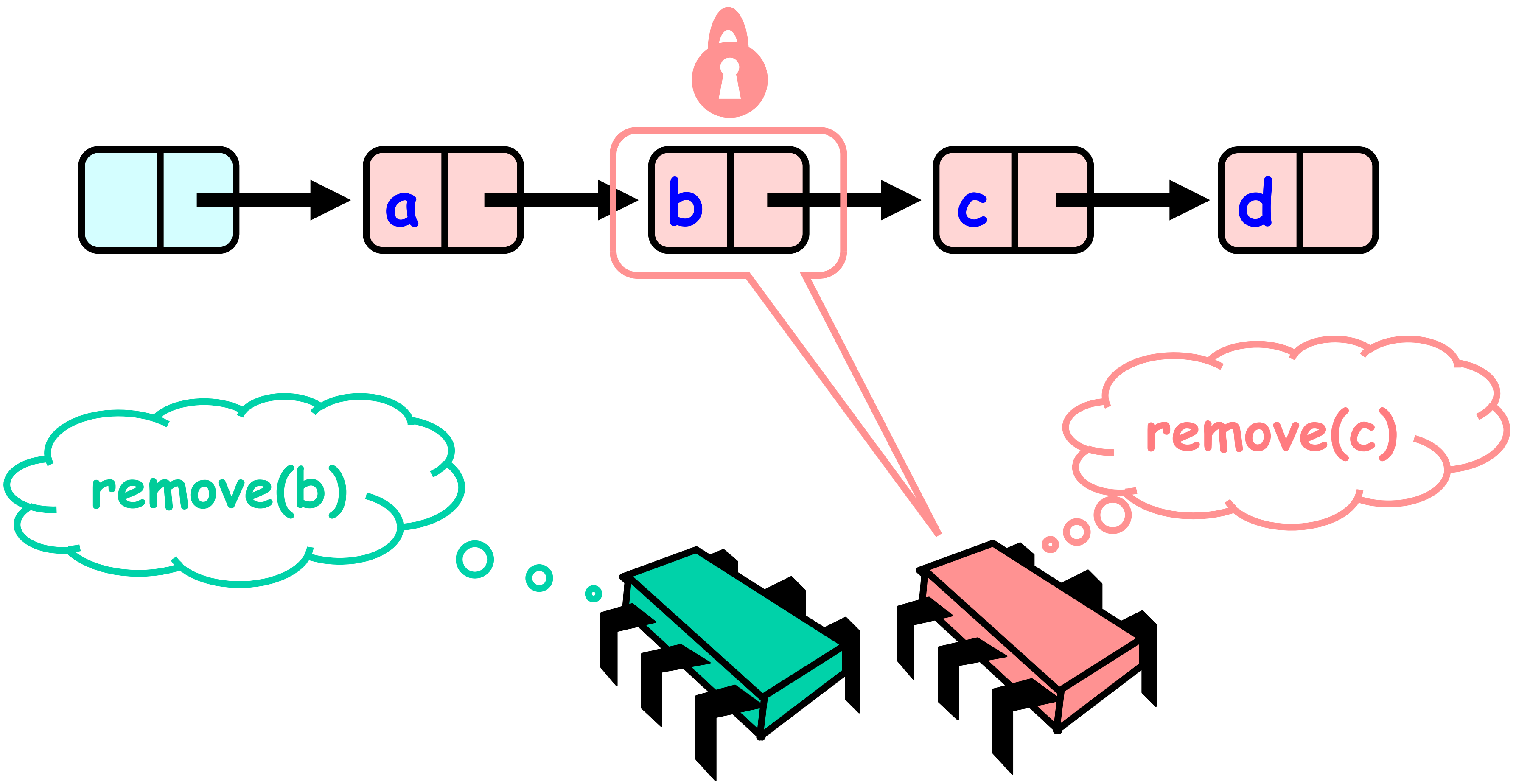
- The red thread looks for the predecessor to c, while the green thread looks for the predecessor to b.



Concurrent Removes

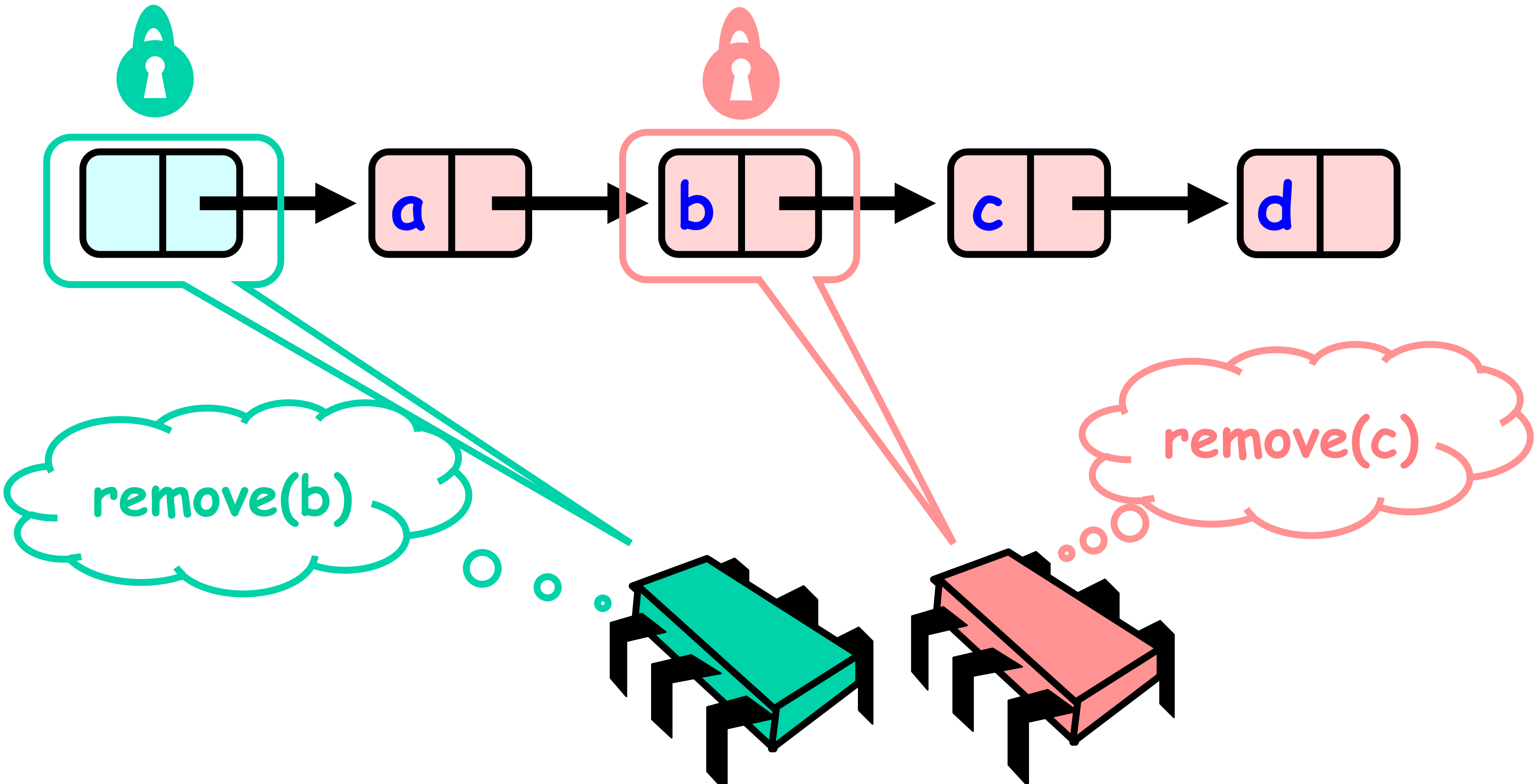


Concurrent Removes

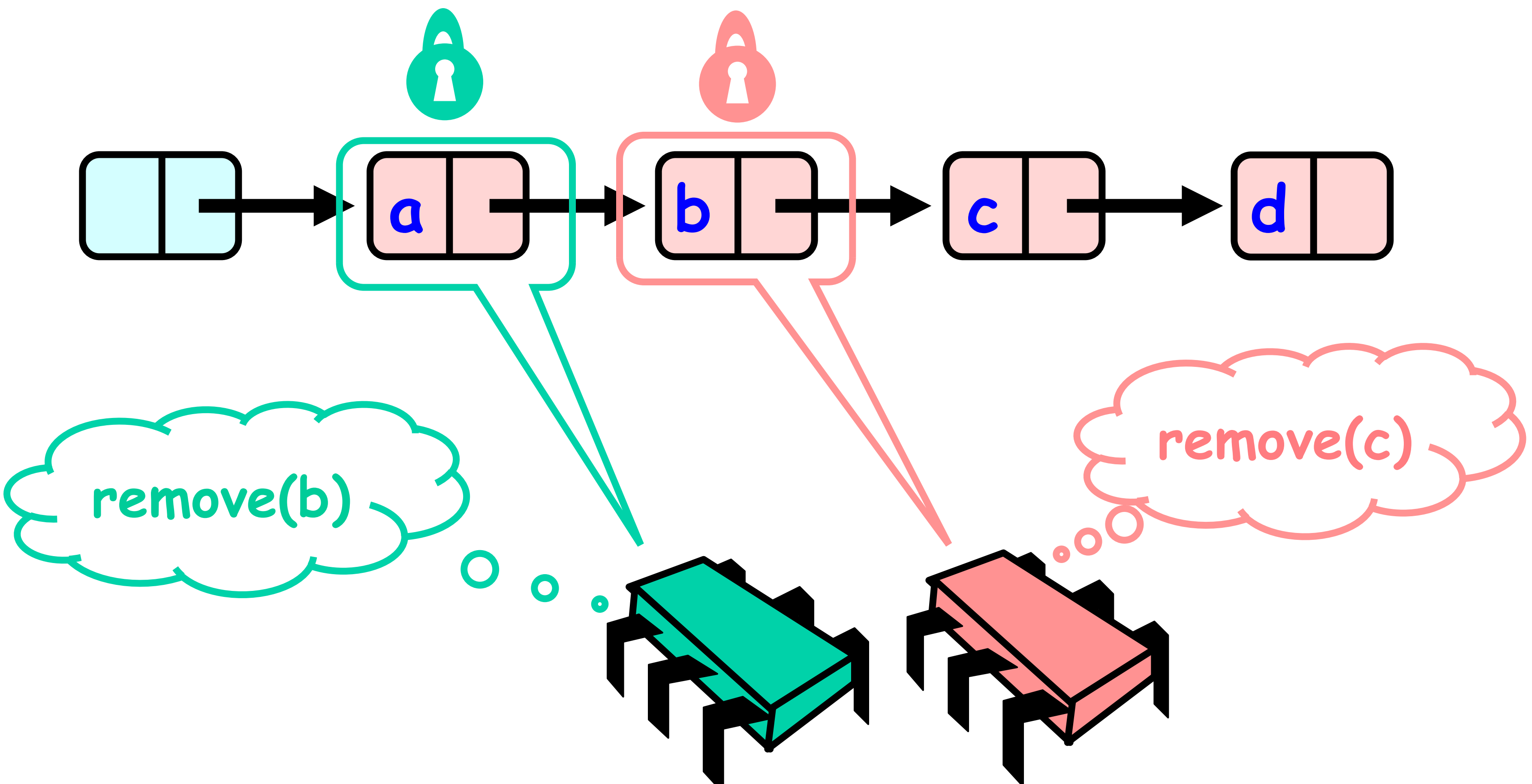


Concurrent Removes

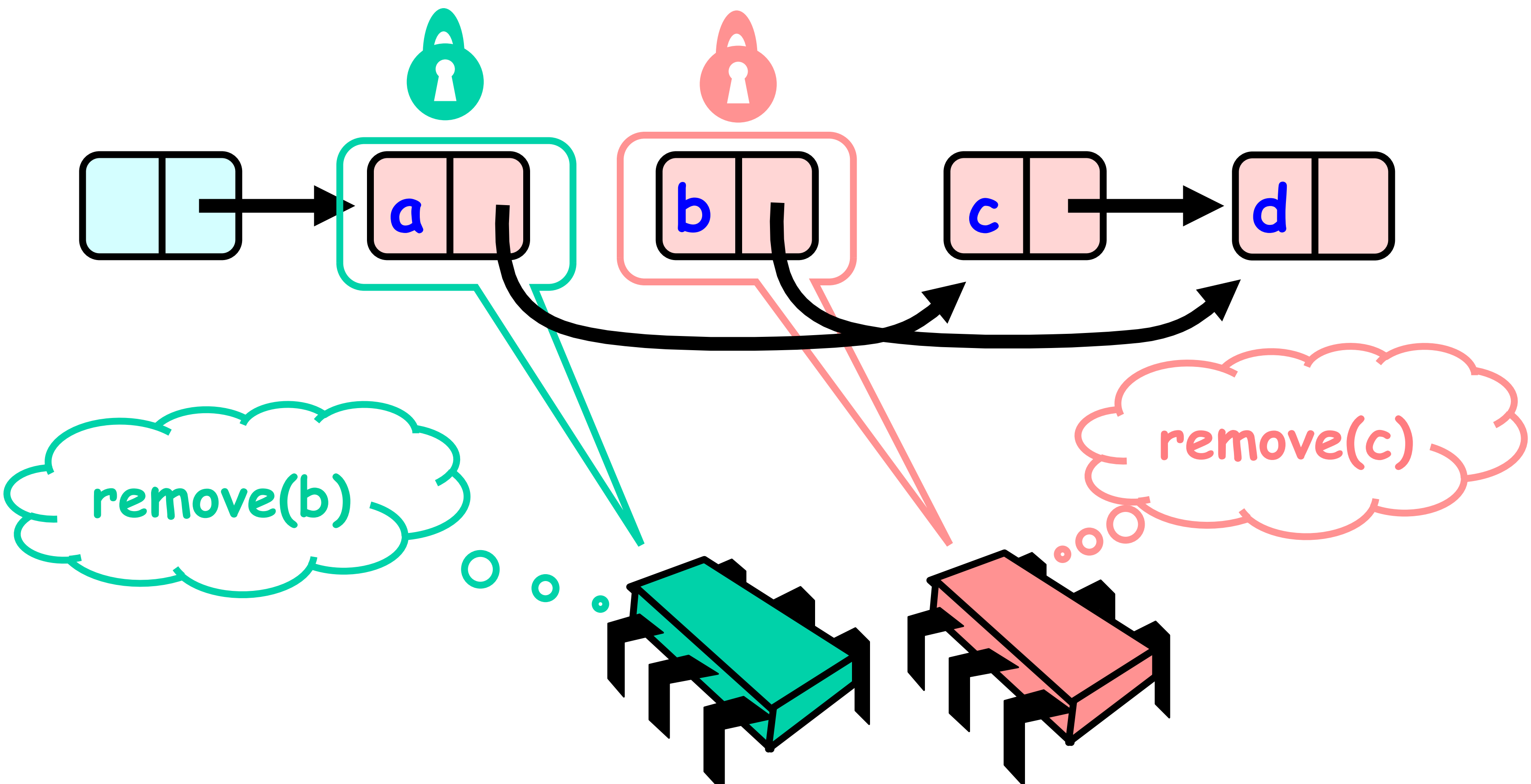
- The green thread looks for the predecessor to b.



Concurrent Removes

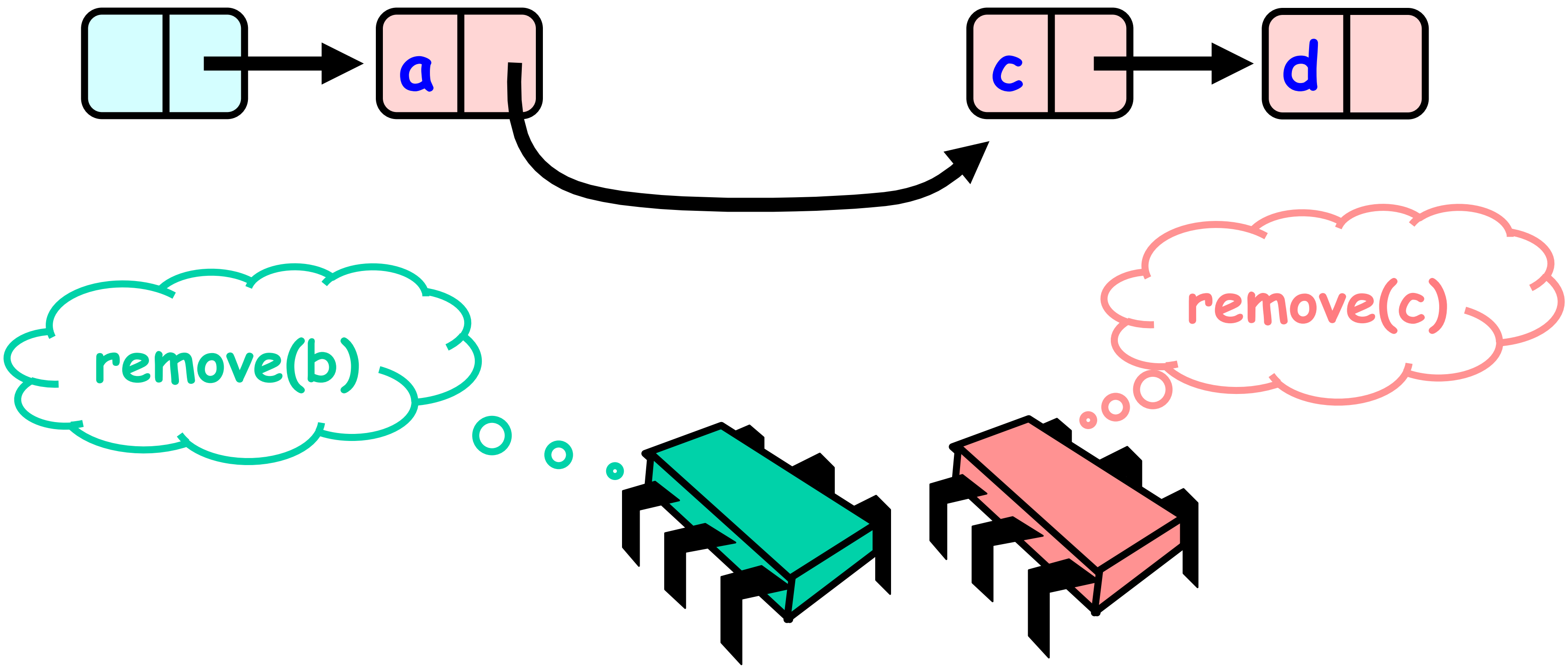


Concurrent Removes

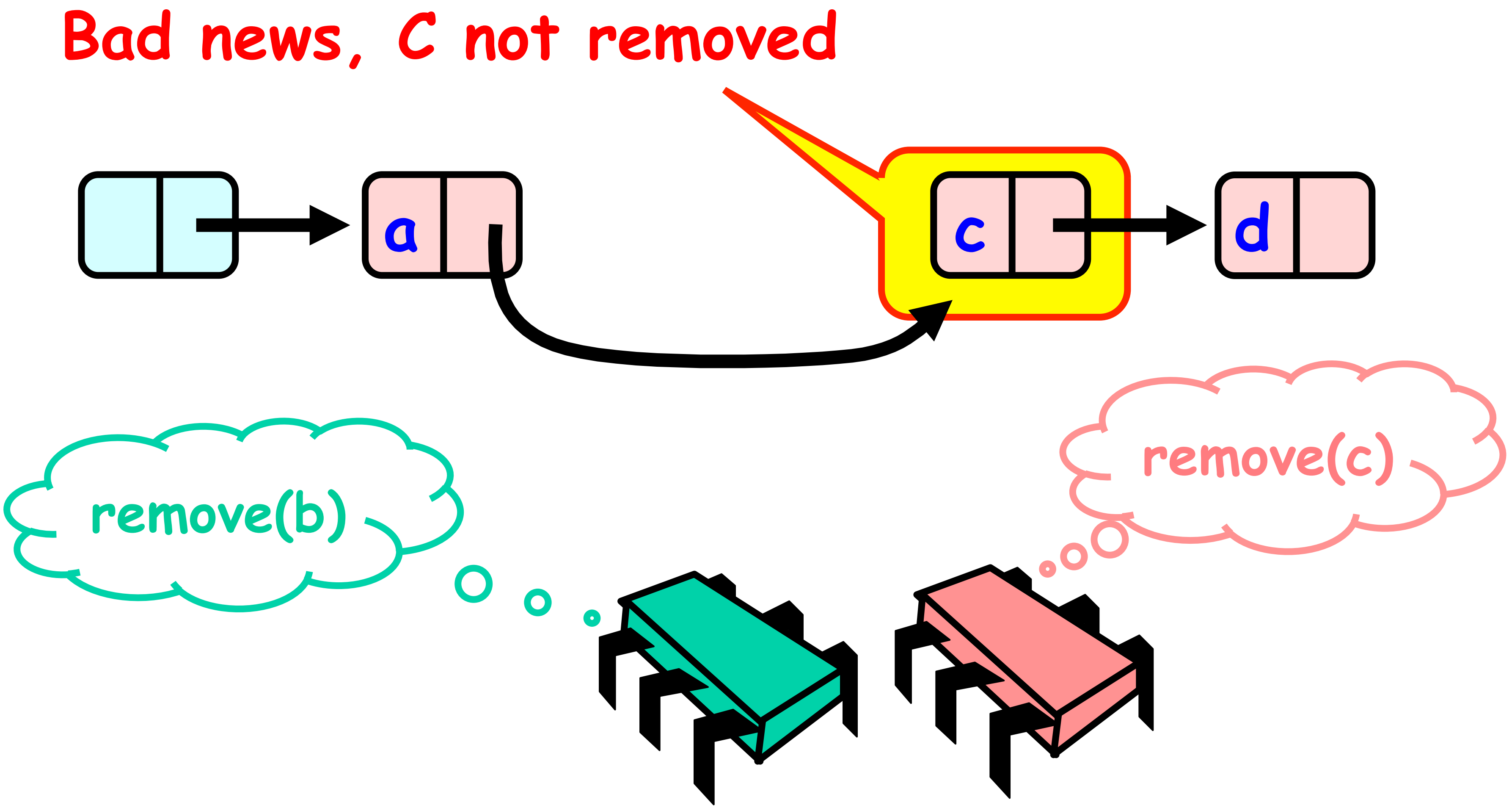


Uh, Oh

- But the final effect is to remove b but not c!



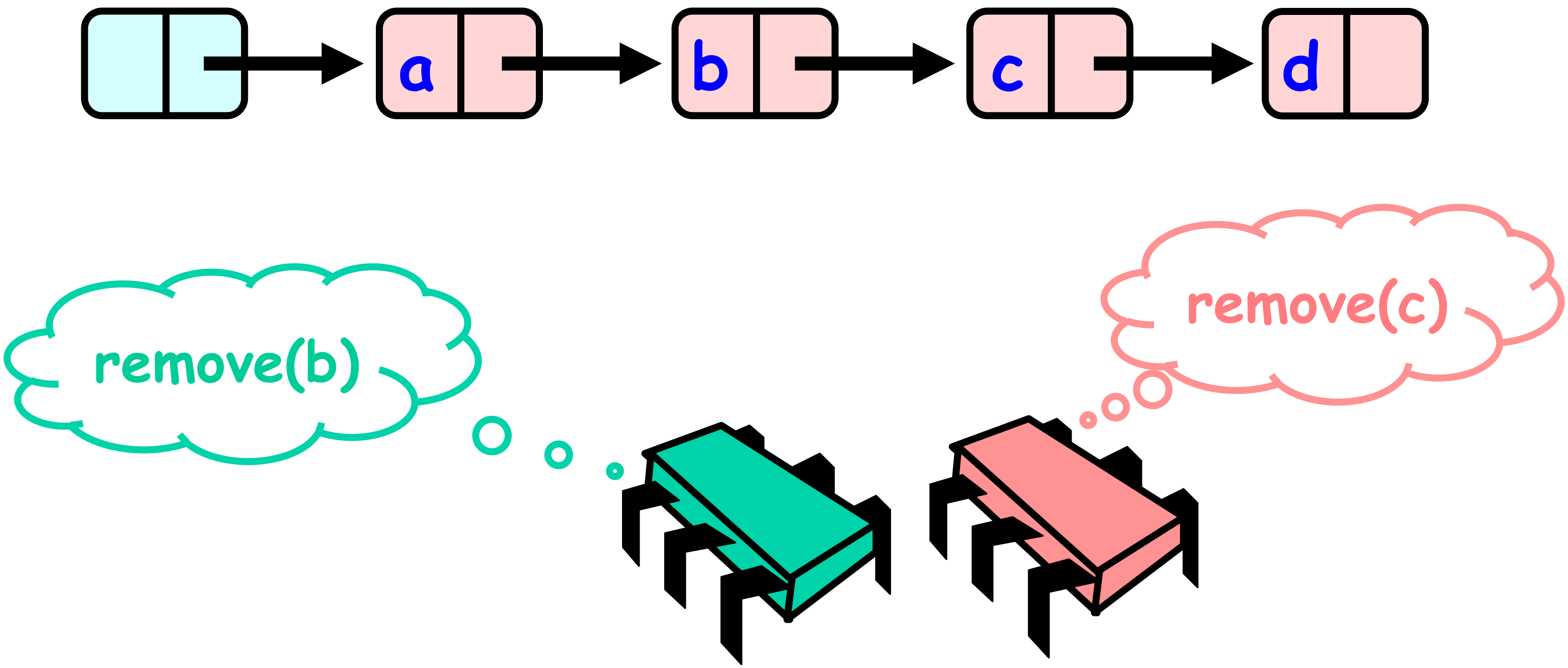
Uh, Oh



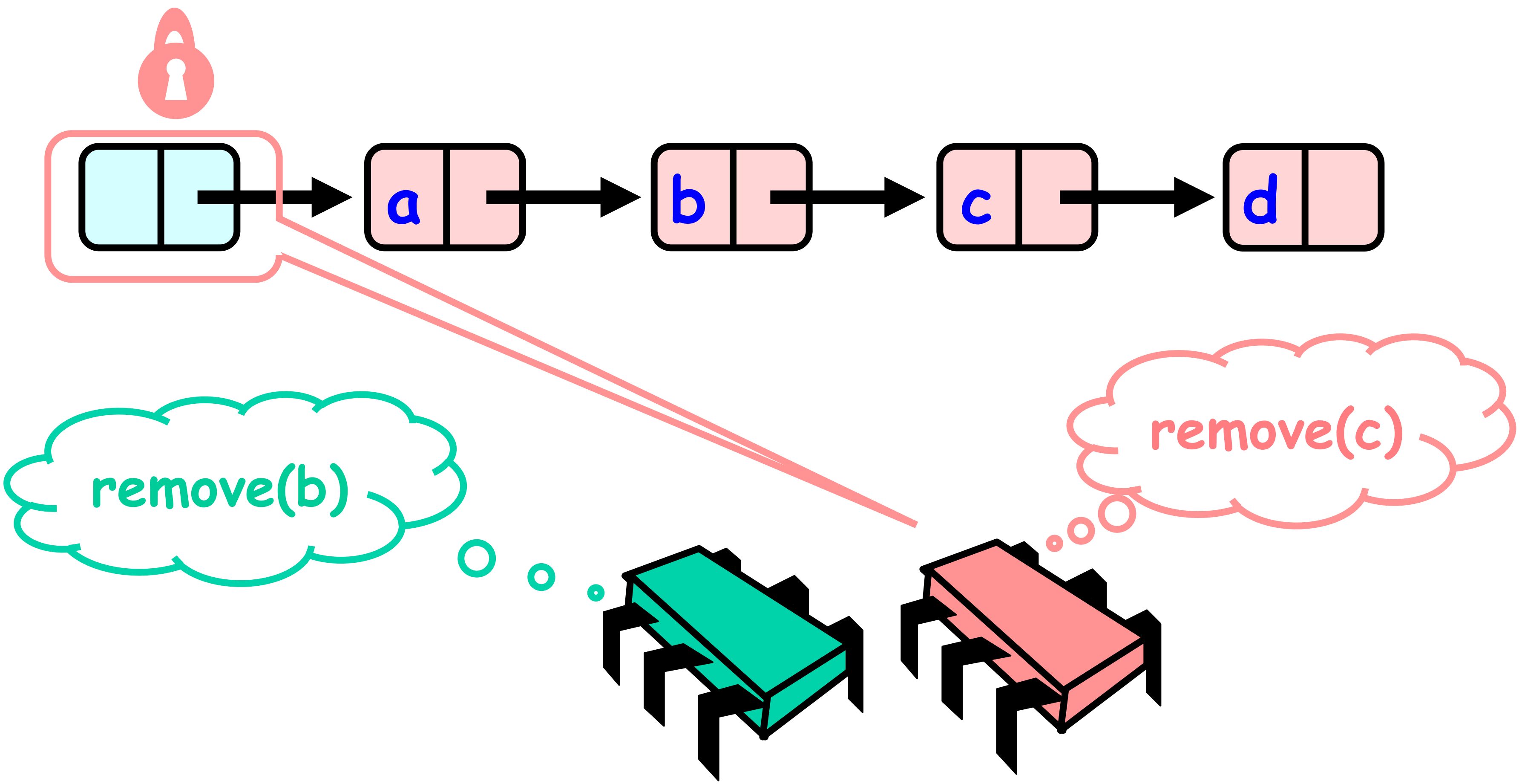
Insight

- If a node is locked
  - No one can delete node's *successor*
- If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works

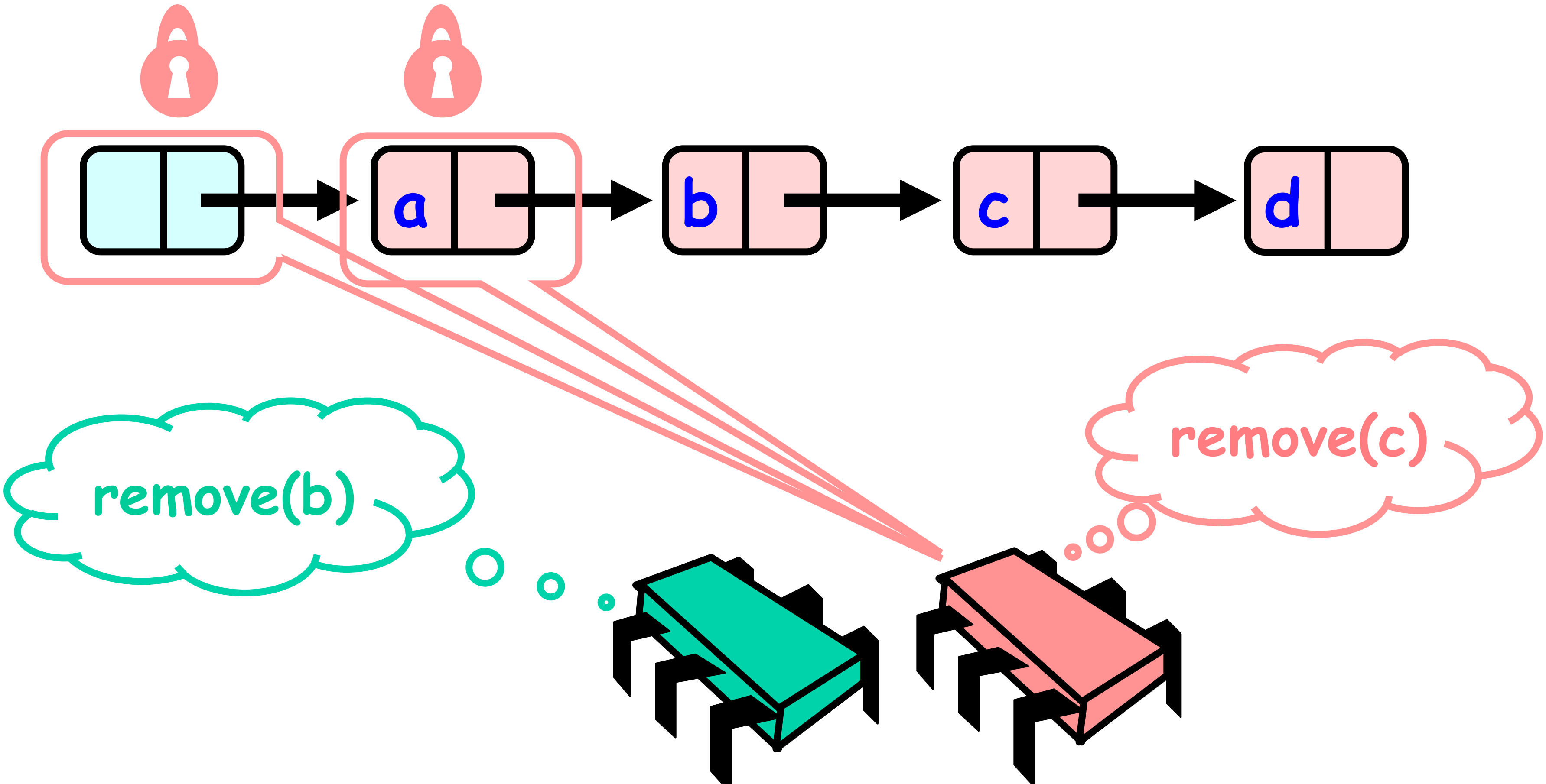
Concurrent Removes Again (each locks two nodes)



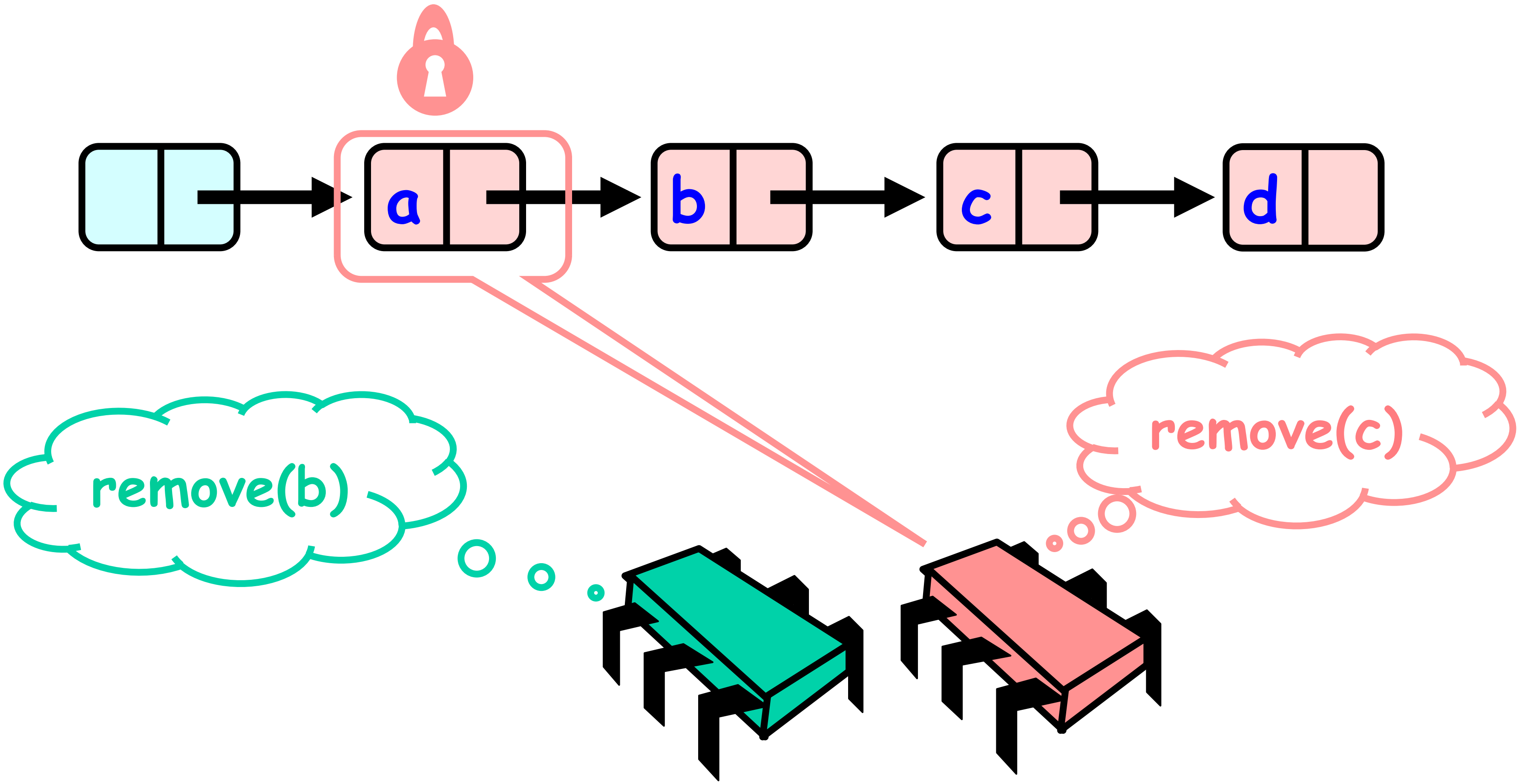
Removing a Node



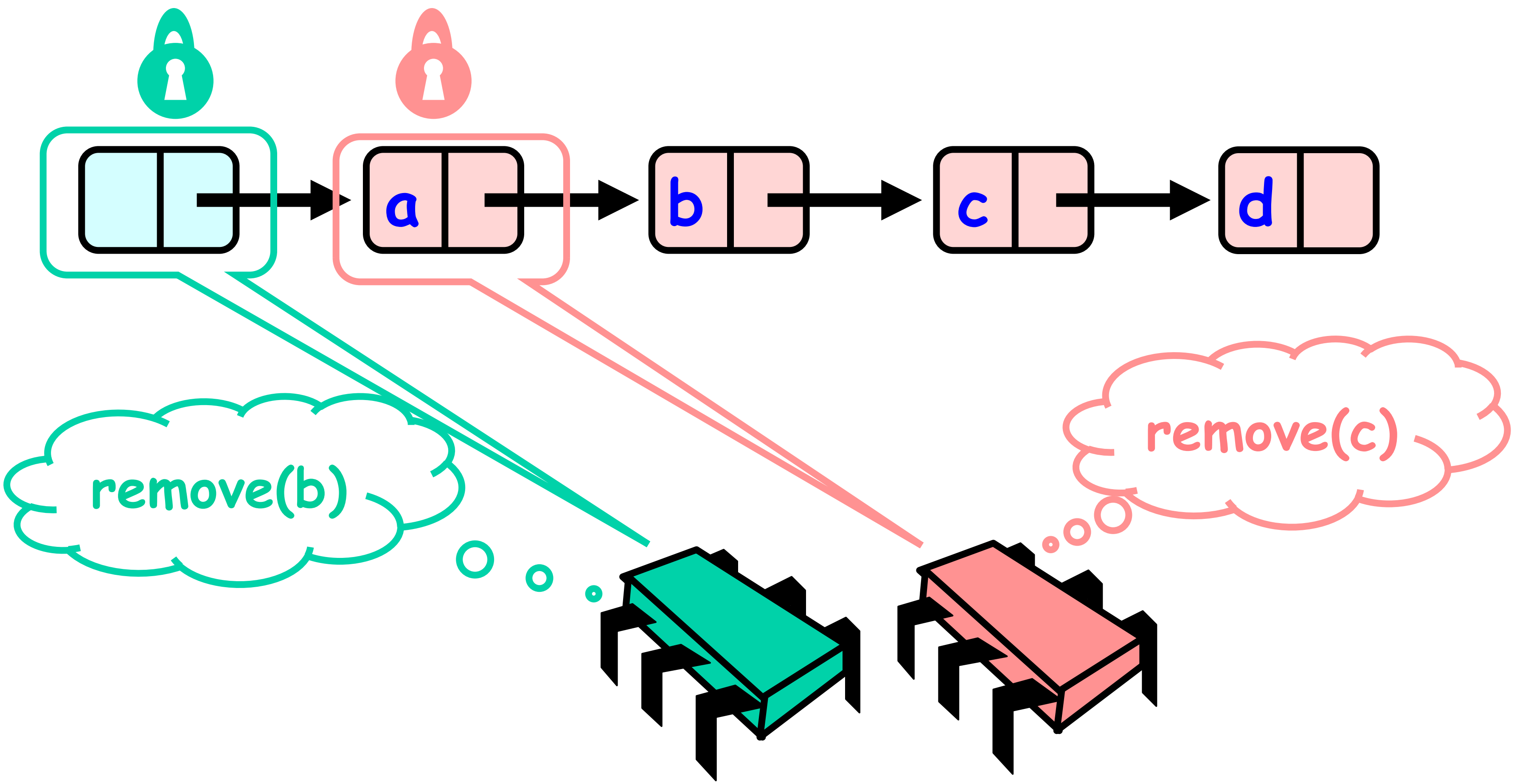
Removing a Node



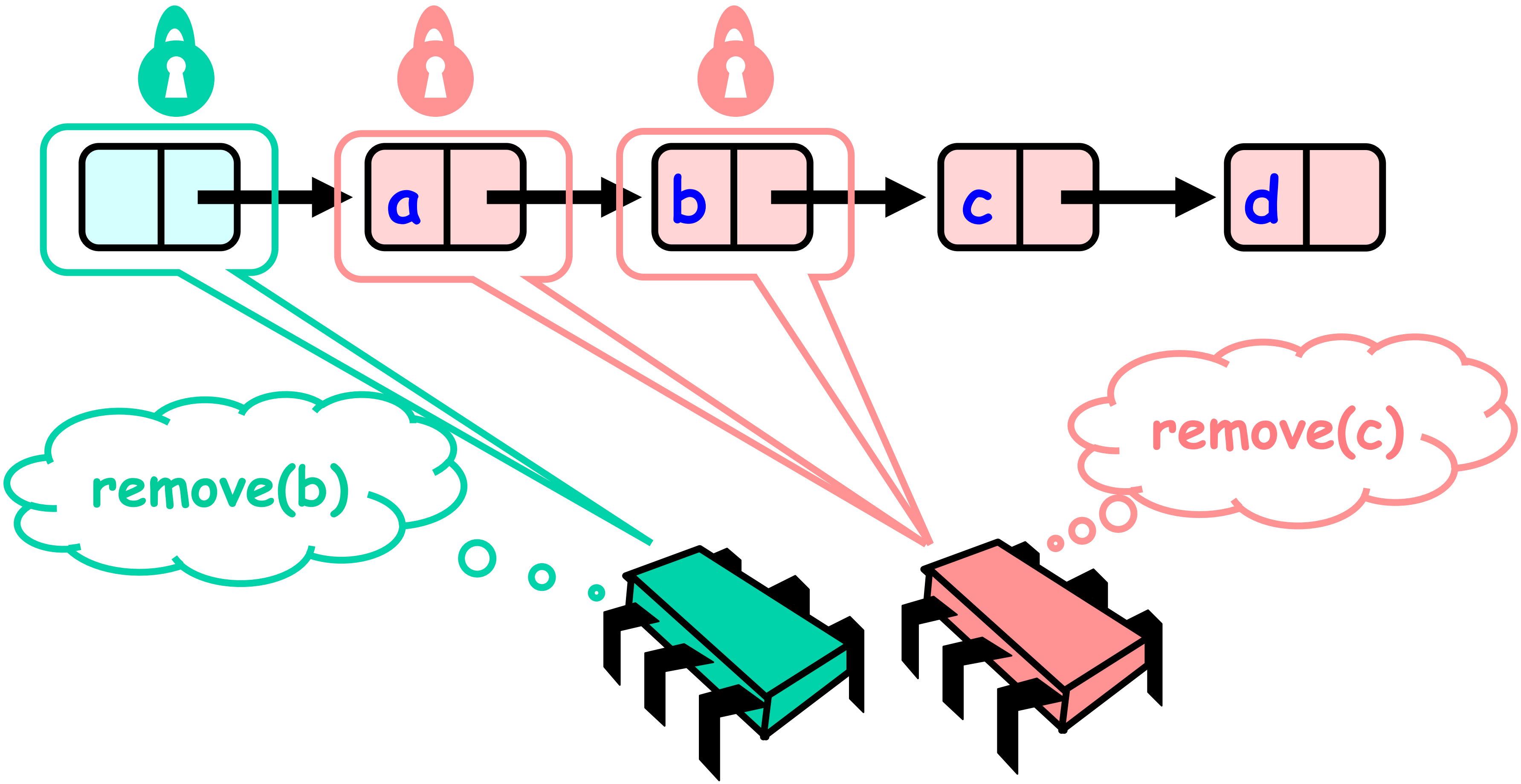
Removing a Node



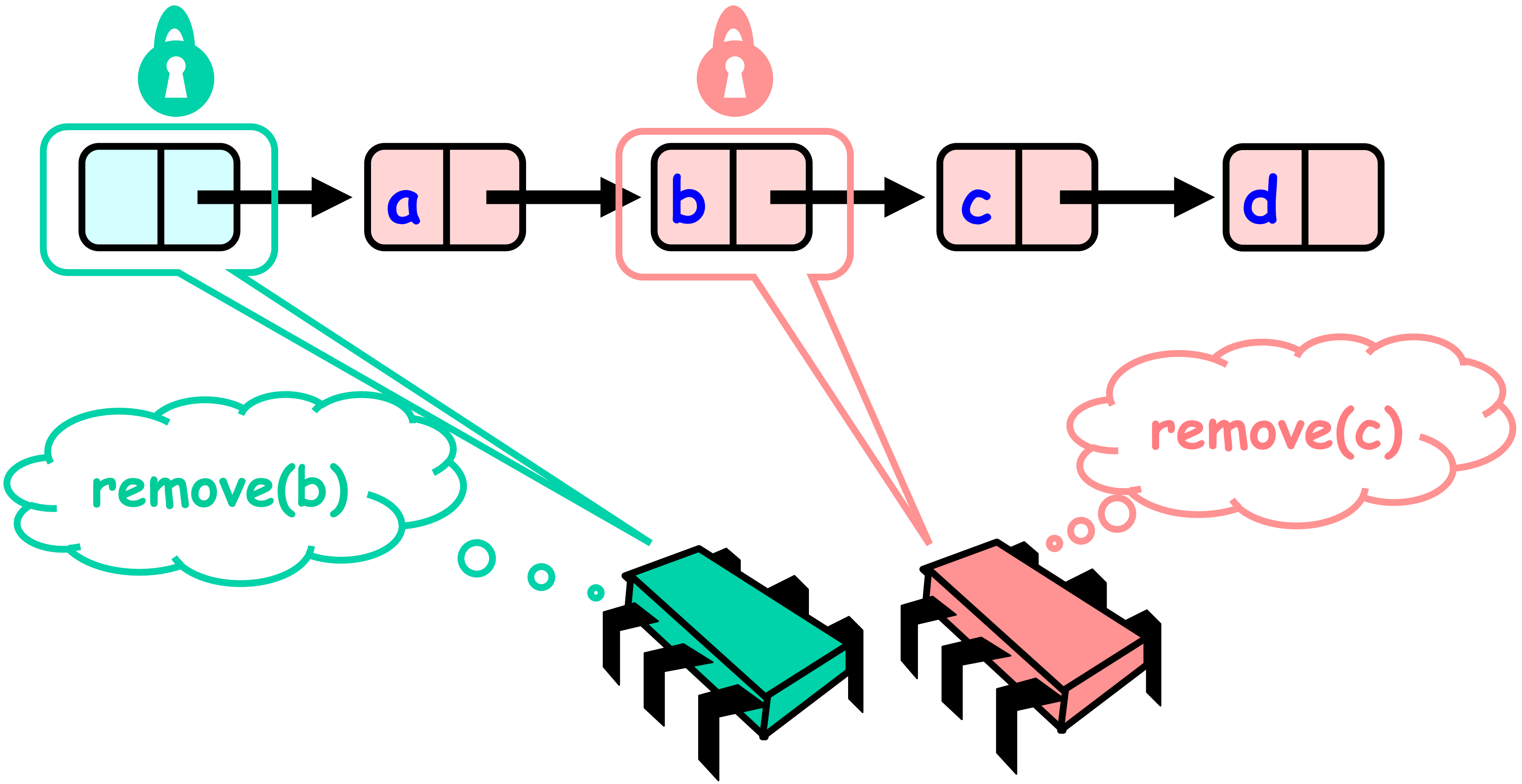
Removing a Node



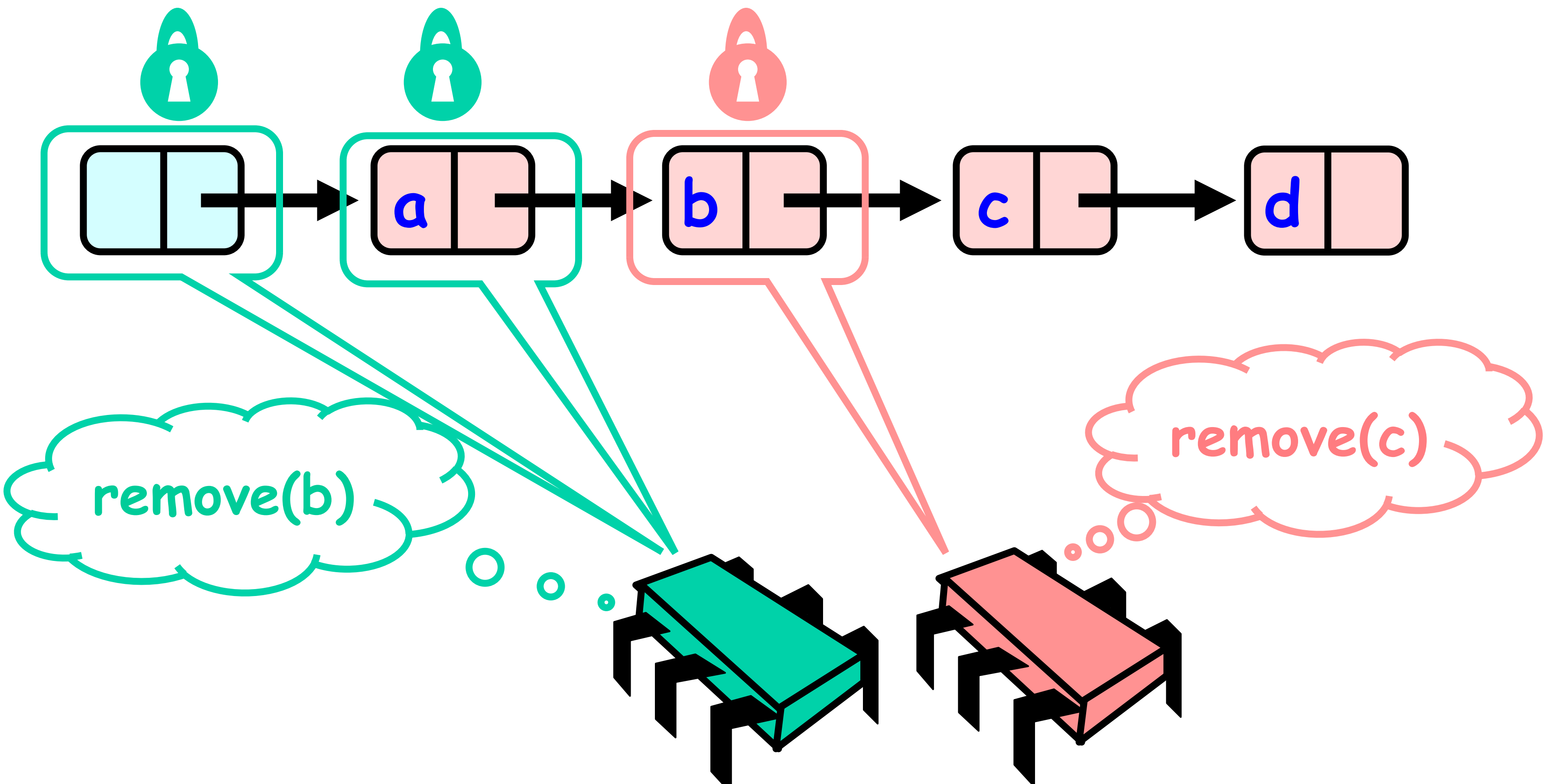
Removing a Node



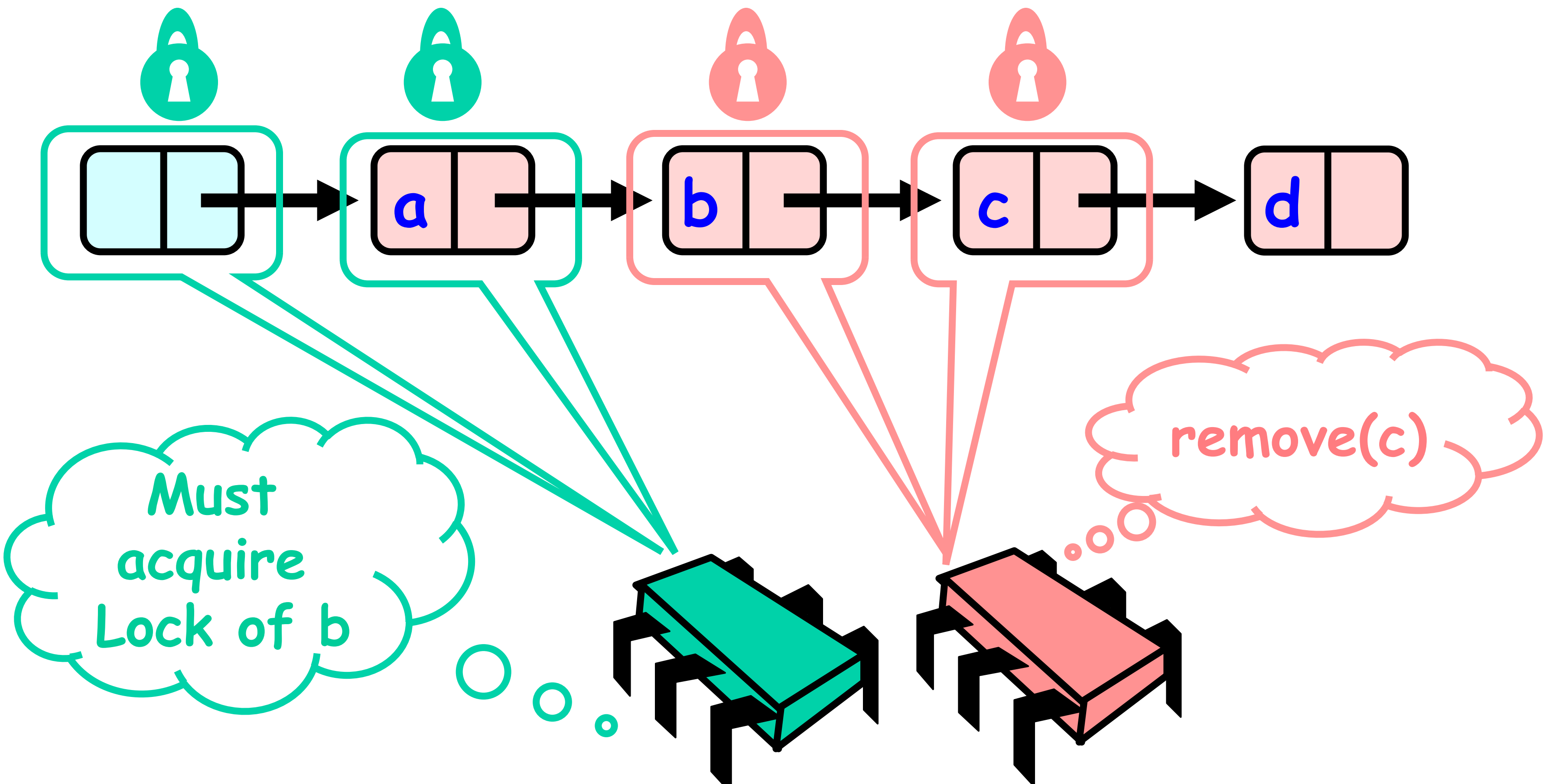
Removing a Node



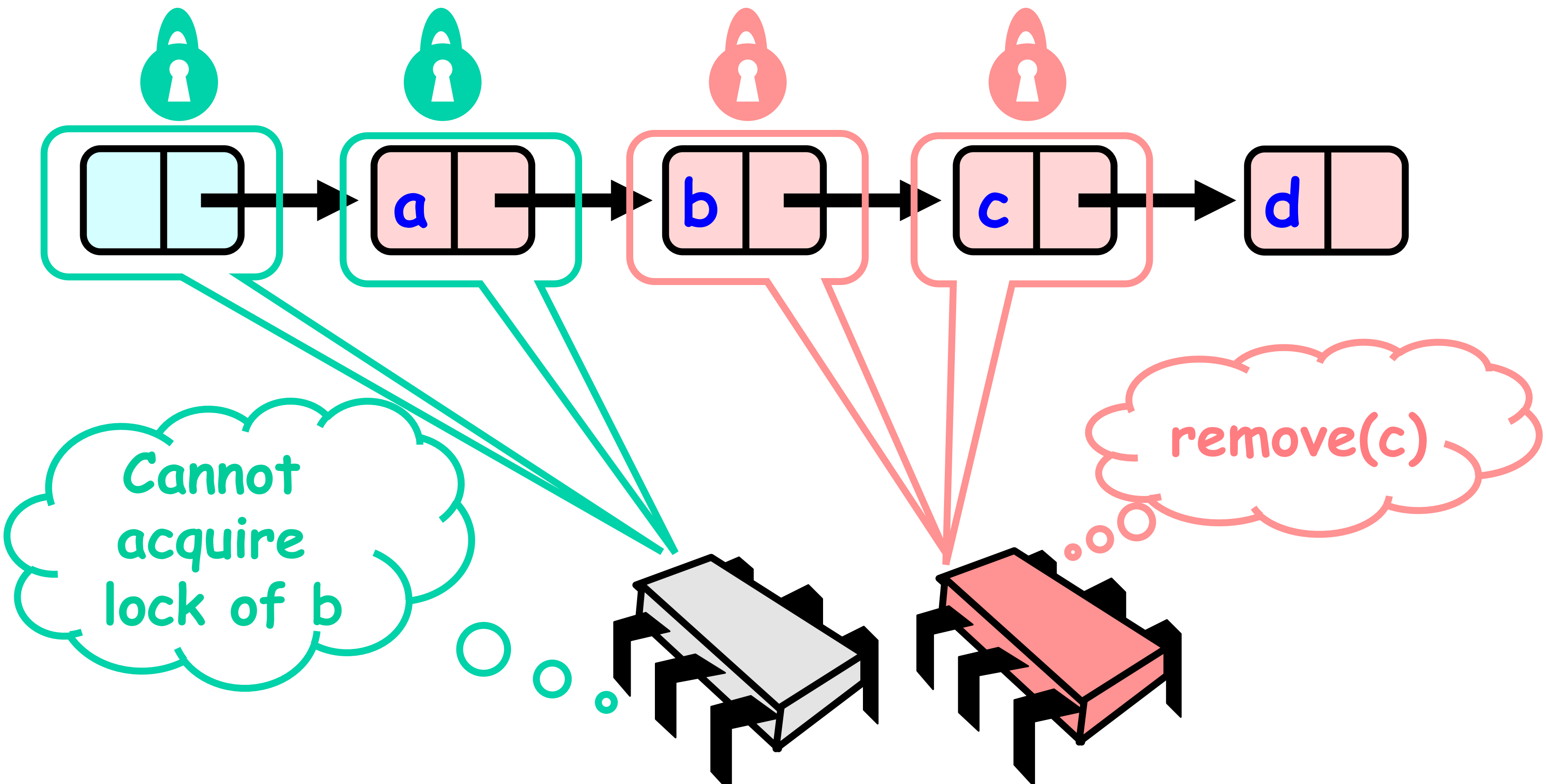
Removing a Node



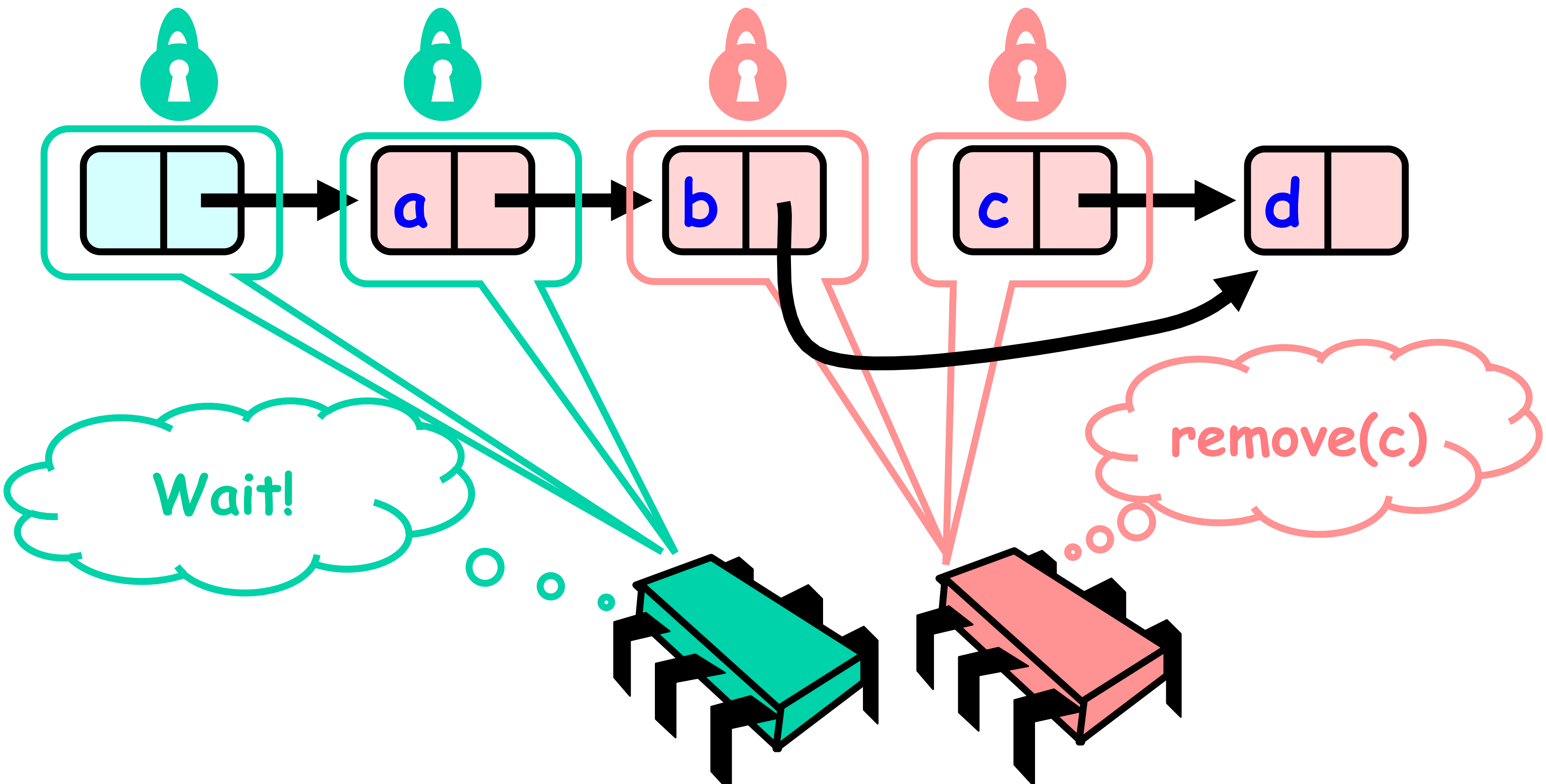
Removing a Node



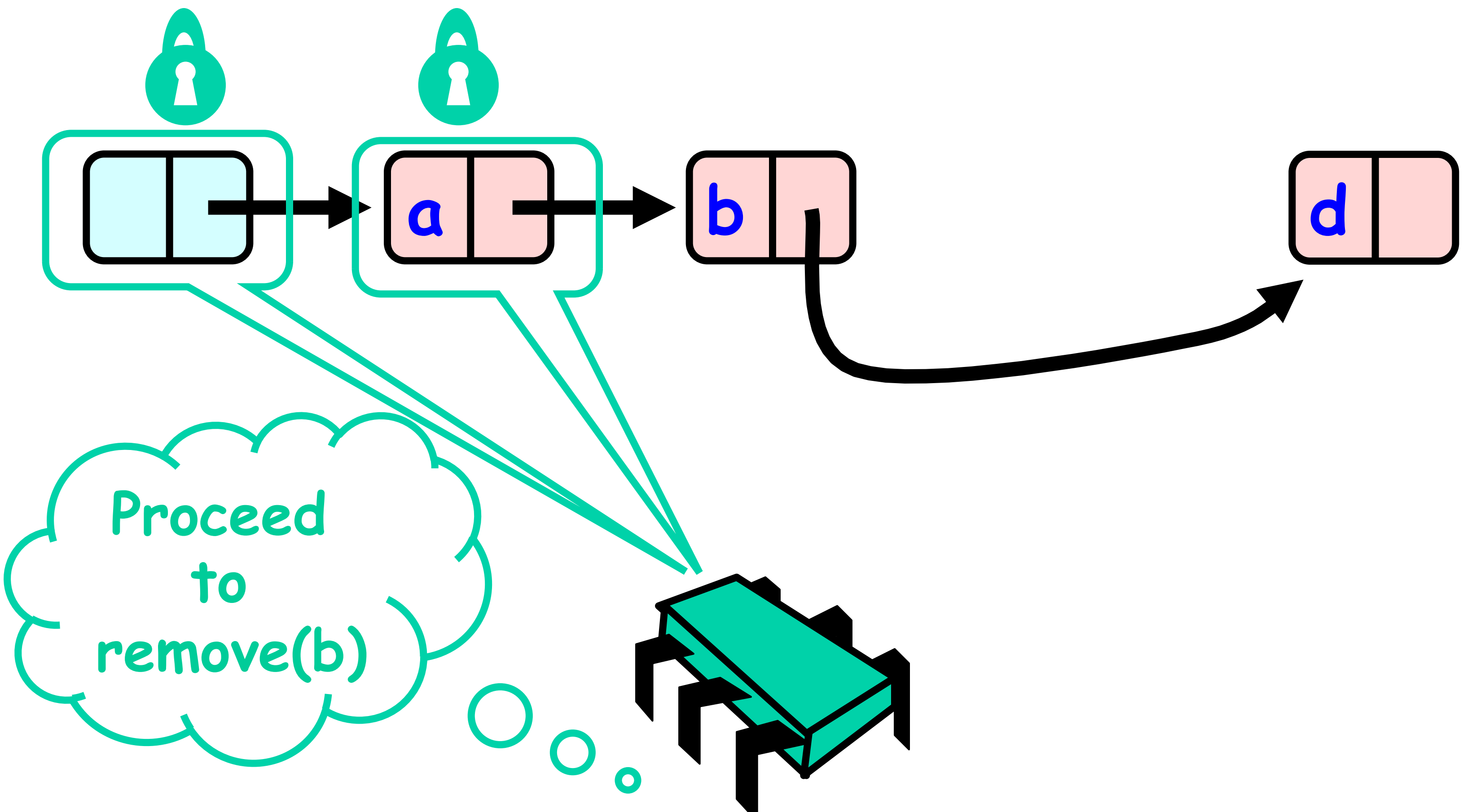
Removing a Node



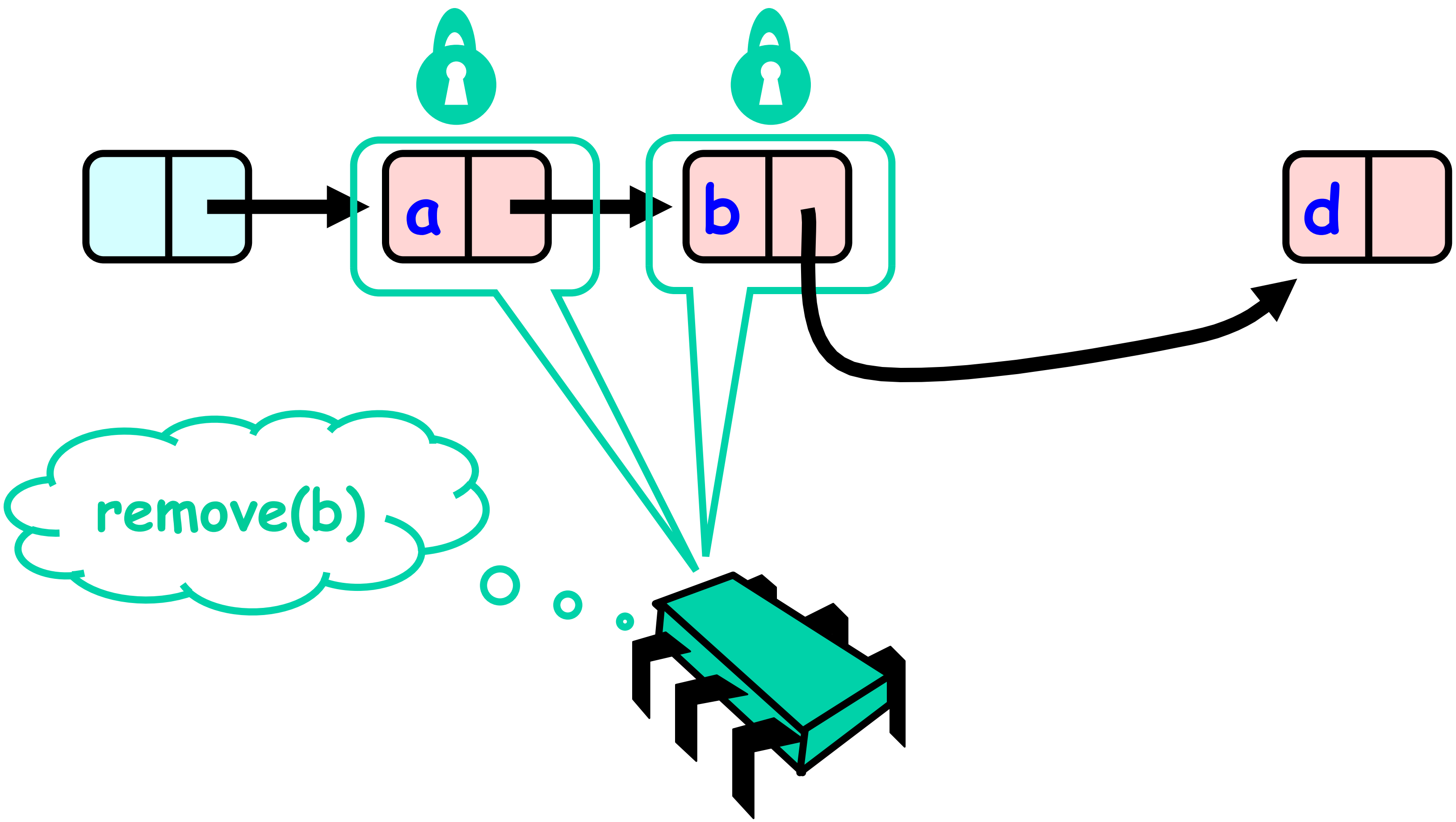
Removing a Node



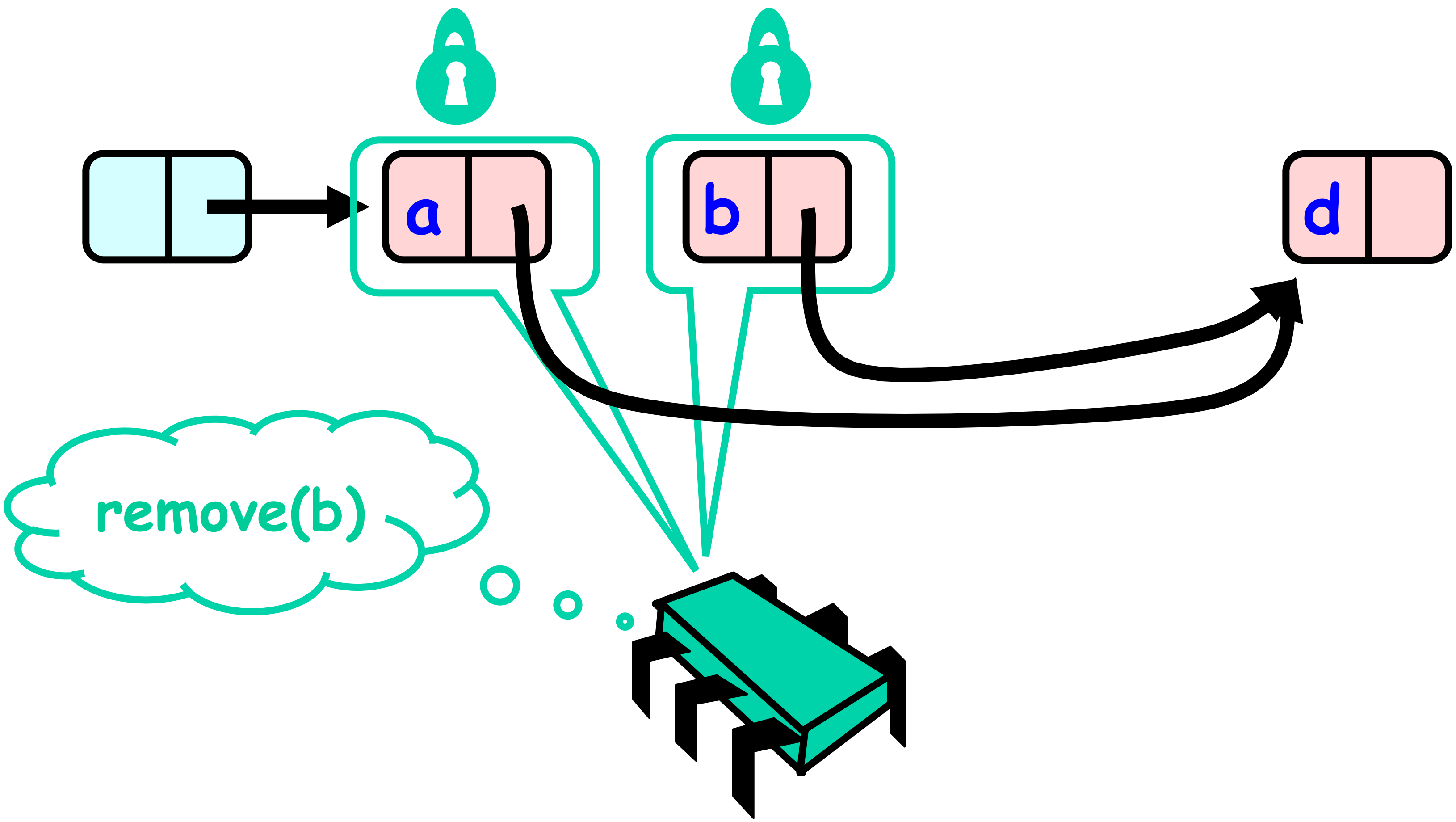
Removing a Node



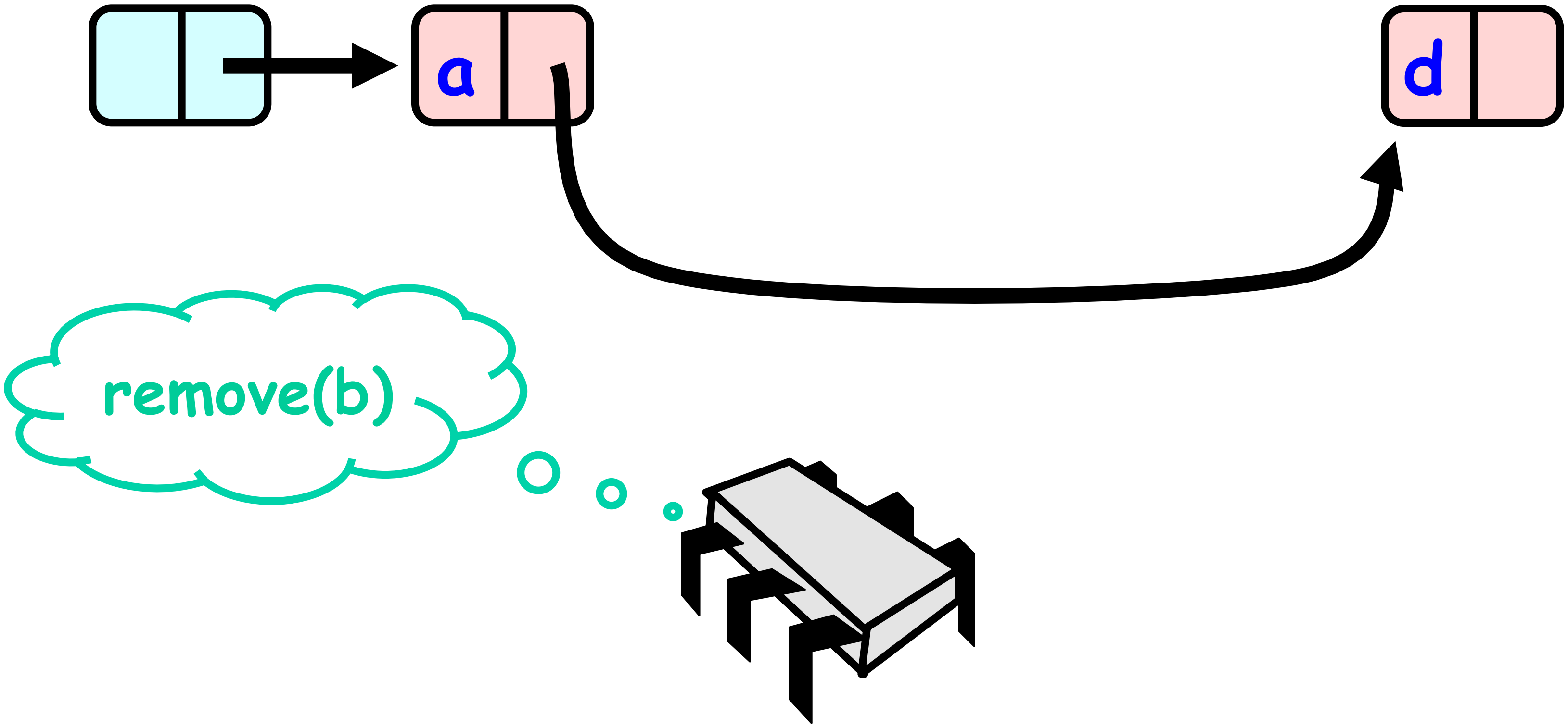
Removing a Node



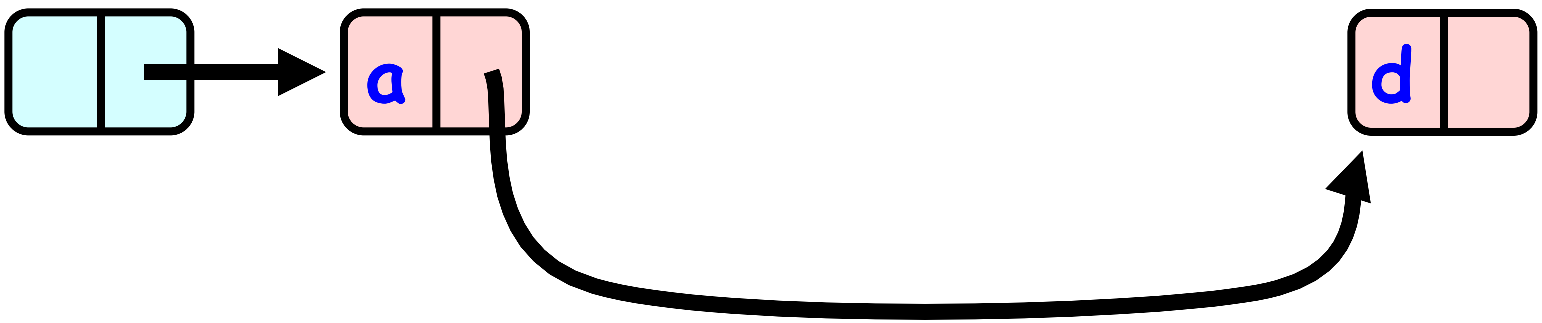
Removing a Node



Removing a Node



Removing a Node



Remove method

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

Remove method

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

Key used to order node

Remove method

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  currNode.unlock();
  predNode.unlock();
 }}
```

Predecessor and current nodes

Remove method

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

Make sure locks released

Remove method

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
  …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

Everything else
(on next slide)

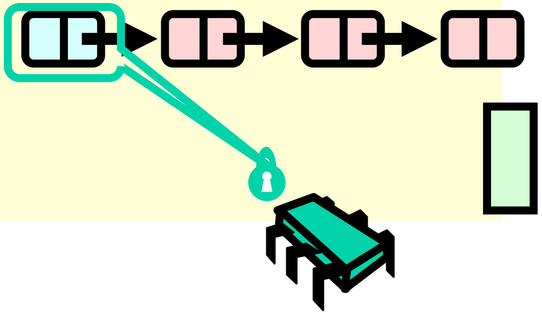
Remove method

```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

Remove method

lock pred == head

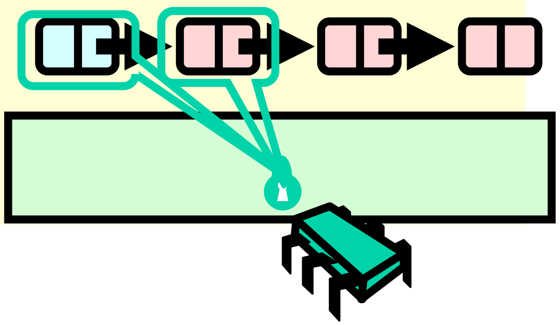
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```



Remove method

```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

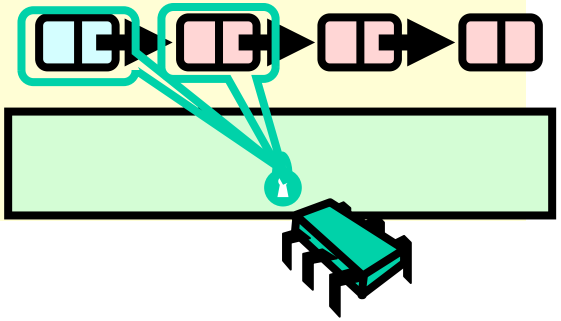
Lock current



Remove method

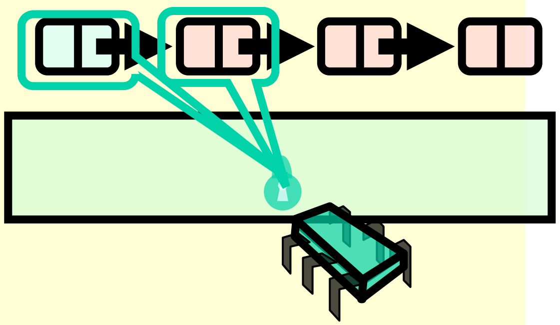
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 ...
} finally { … }
```

Traversing list



Remove: searching

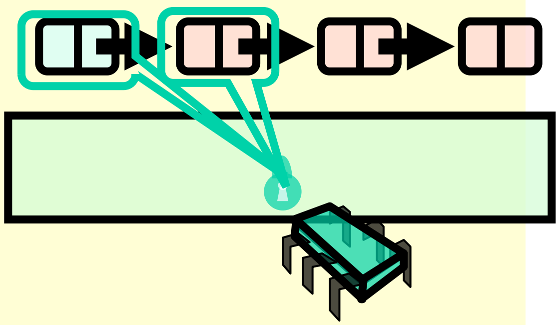
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next; // remove
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

```
while (curr.key <= key) {
   if (item == curr.item) {
    pred.next = curr.next;
    return true;
   }
   pred.unlock();
   pred = curr;
   curr = curr.next;
   curr.lock();
 }
 return false;
```

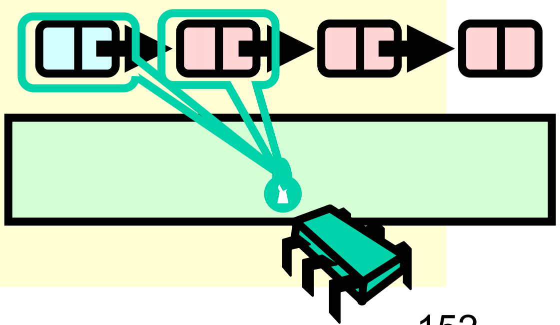
Search key range



Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

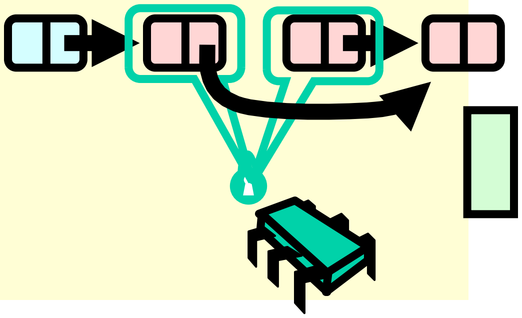
At start of each loop: curr and pred locked



Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
```

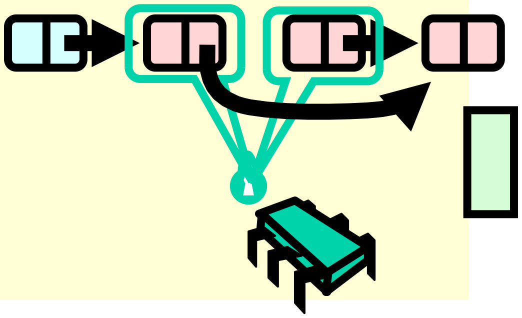
If item found, remove node



Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
```

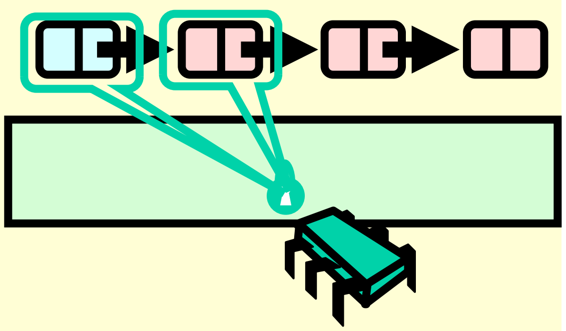
If node found, remove it



Remove: searching

Unlock predecessor

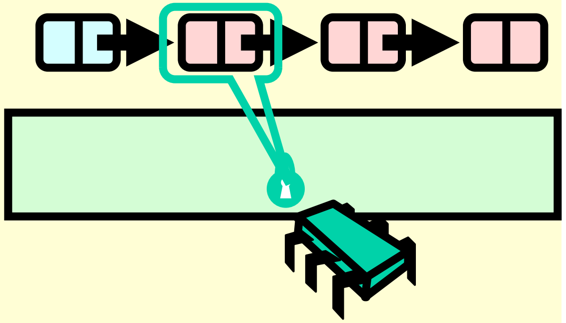
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

Only one node locked!

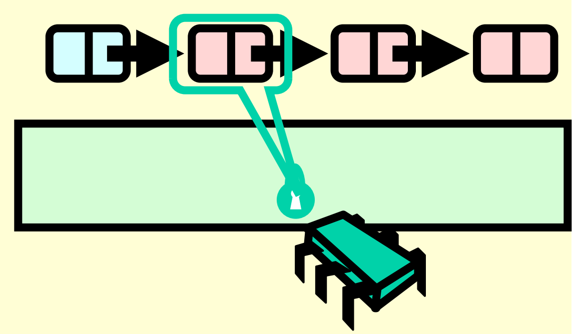
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

```
while (curr.key <= key) {
   if (item == curr.item) {
    pred.next = curr.next;
    return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

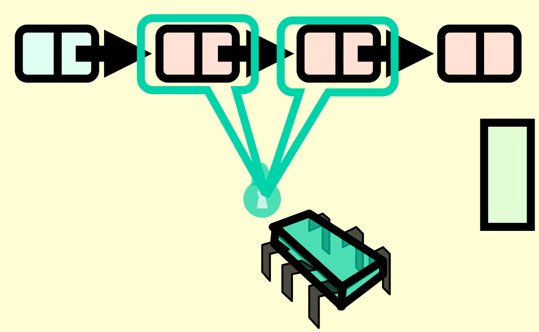
demote current



Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = currNode;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

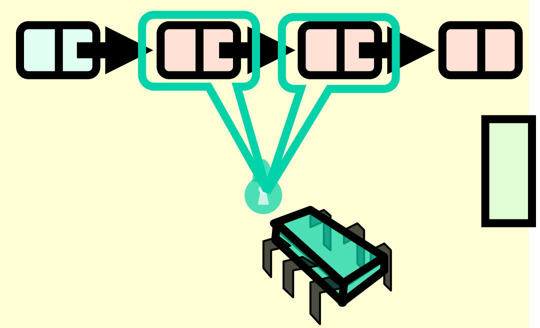
Find and lock new current



Remove: searching

Lock invariant restored

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = currNode;
  curr = curr.next;
  curr.lock();
}
return false;
```



Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Otherwise, not present

Why does this work?

- To remove node e
  - Must lock e
  - Must lock e’s predecessor
- Therefore, if you lock a node
  - It can’t be removed
  - And neither can its successor

Linearization points in remove()

```
while (curr.key <= key) {
   if (item == curr.item) {
     pred.next = curr.next;
     return true;
   }
   pred.unlock();
   pred = curr;
   curr = curr.next;
   curr.lock();
 }
 return false;
```

Linearization point if item is present

Why remove() is linearizable

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Linearization point if item not present (before return false)

Adding Nodes

- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted
  - (Is successor lock actually required?)
  - successor lock not actually required!

Drawbacks of Fine-grained Locking

- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient

Traffic Jam

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And “eats the big muffin” (stops running)
    - Cache miss, page fault, descheduled …
  - Everyone else using that lock is stuck!
  - Need to trust the scheduler….

Fourth Pattern: Lock-Free

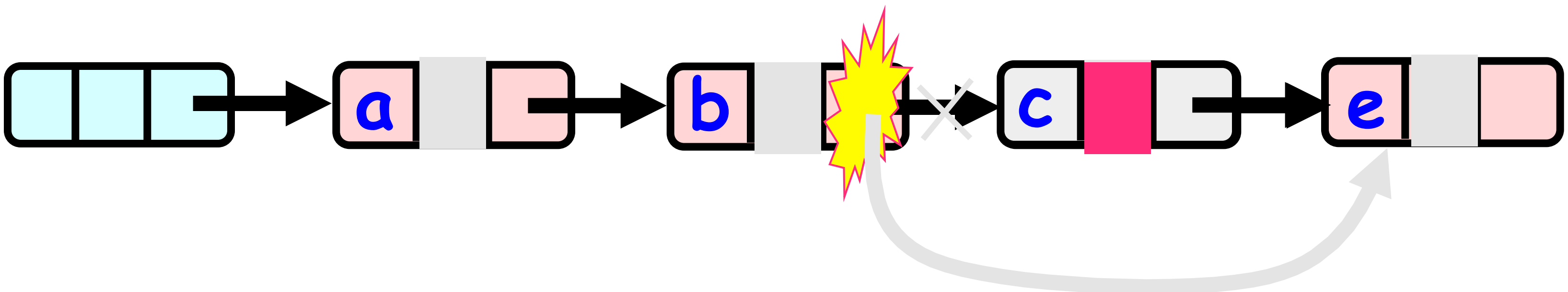
- Reminder: Lock-Free Data Structures
- No matter what …
  - Guarantees minimal progress in any execution
  - i.e. Some thread will always complete a method call
  - Even if others halt at malicious times
  - Implies that implementation can’t use locks

A Lock-free List (Set)

- Next logical step
  - Wait-free `contains()`
    - Search by traversing the list from head to tail without locking
  - Lock-free `add()` and `remove()`
    - Search by traversing the list from head to tail without locking
    - Use CAS to complete the operation
- Use only **CAS: compareAndSet()**
  - What could go wrong?

Lock-free List

Logical Removal =
Set Mark Bit

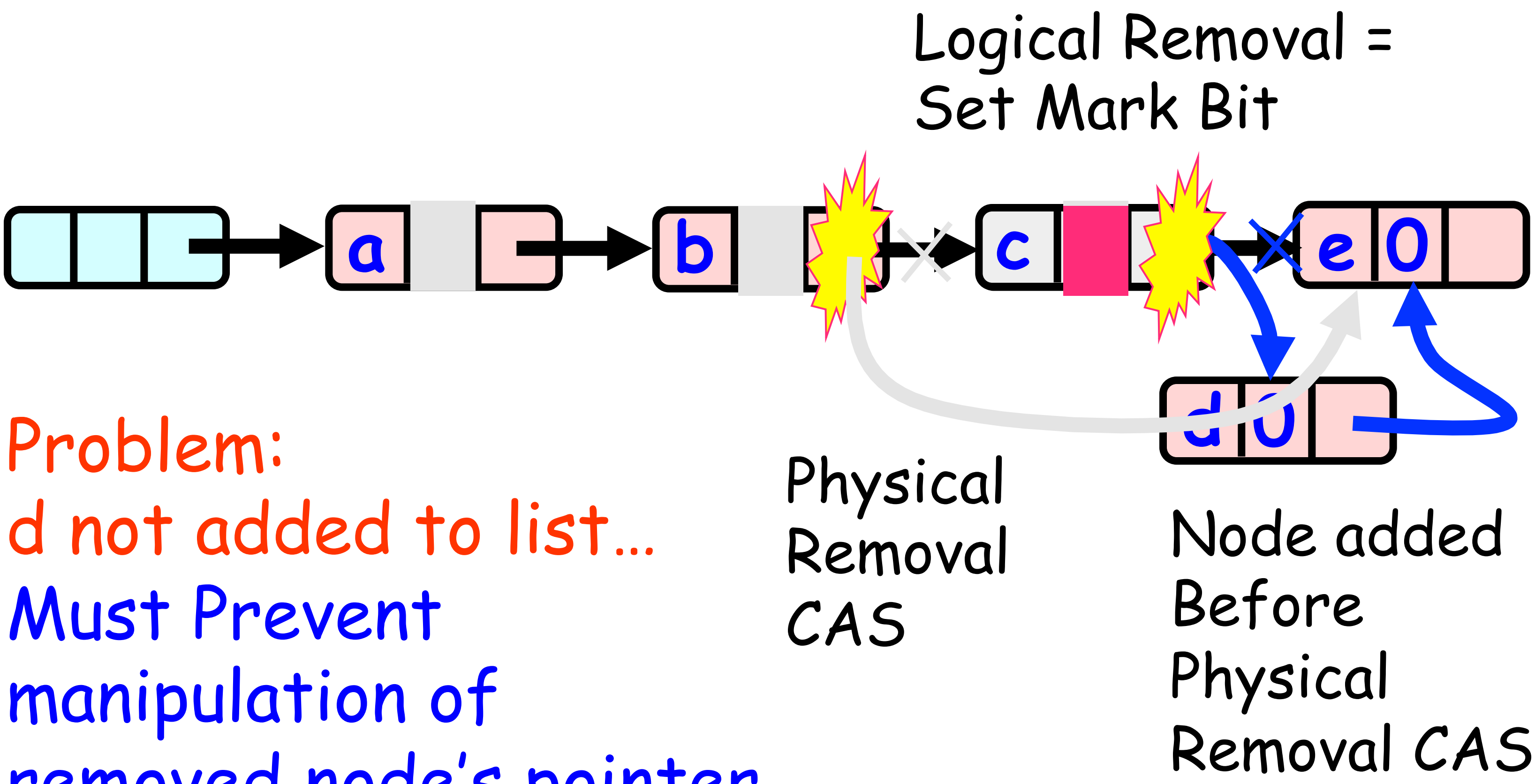


Use CAS to verify pointer
is correct

Physical
Removal
CAS pointer

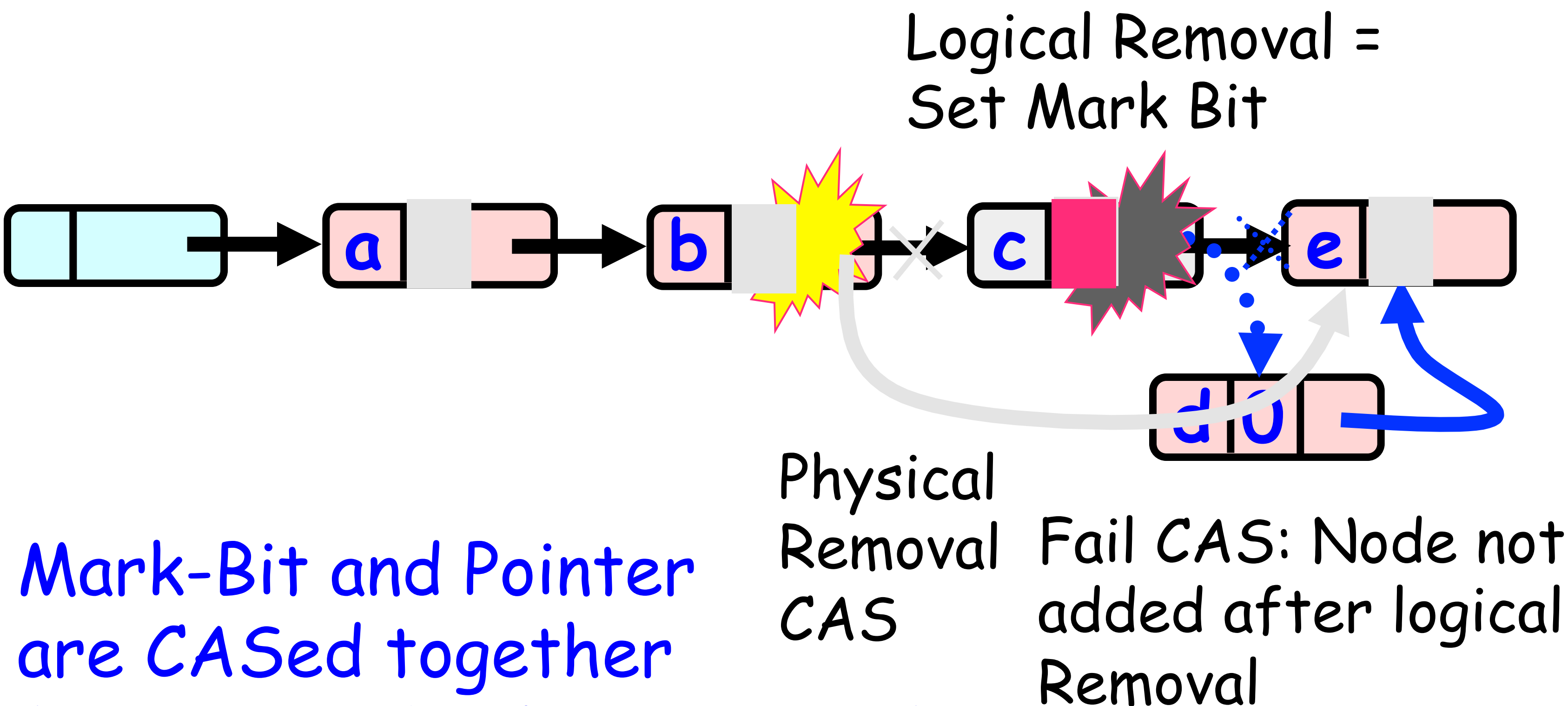
Not enough!

Problem…



Problem:
d not added to list…
Must Prevent
manipulation of
removed node's pointer

The Solution: Combine Bit and Pointer



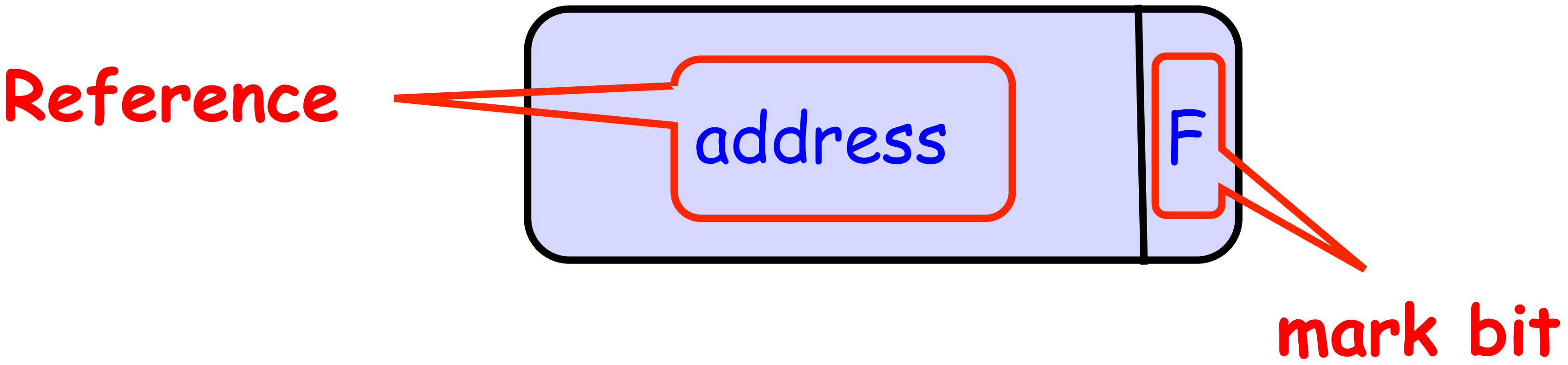
Mark-Bit and Pointer are CASed together (AtomicMarkableReference)

Solution

- Use `AtomicMarkableReference`
- Atomically
  - Swing reference and
  - Update flag
- Remove in two steps
  - Set `mark` bit in `next` field
  - Redirect predecessor's pointer

Marking a Node

- **AtomicMarkableReference** class
  - `java.util.concurrent.atomic` package



Getting Reference (& Mark)

- Getting Reference & Mark

```
Public Object get(boolean[] marked);
```

```
Public Object get(boolean[] marked);
```

Returns reference

Returns mark at array index 0!

- Getting Mark only

```
public boolean isMarked();
```

Value of mark

Changing State

```
Public boolean compareAndSet(
  Object expectedRef,
  Object updateRef,
  boolean expectedMark,
  boolean updateMark);
```

Changing State

If this is the current reference …

```
Public boolean compareAndSet(
  Object expectedRef,
  Object updateRef,
  boolean expectedMark,
  boolean updateMark);
```

And this is the current mark …

Changing State

…then change to this new reference …

```
Public boolean compareAndSet(
  Object expectedRef,
  Object updateRef,
  boolean expectedMark,
  boolean updateMark);
```

… and this new mark

Changing State

```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```

Changing State

```
public boolean attemptMark(
   Object expectedRef,
   boolean updateMark);
```

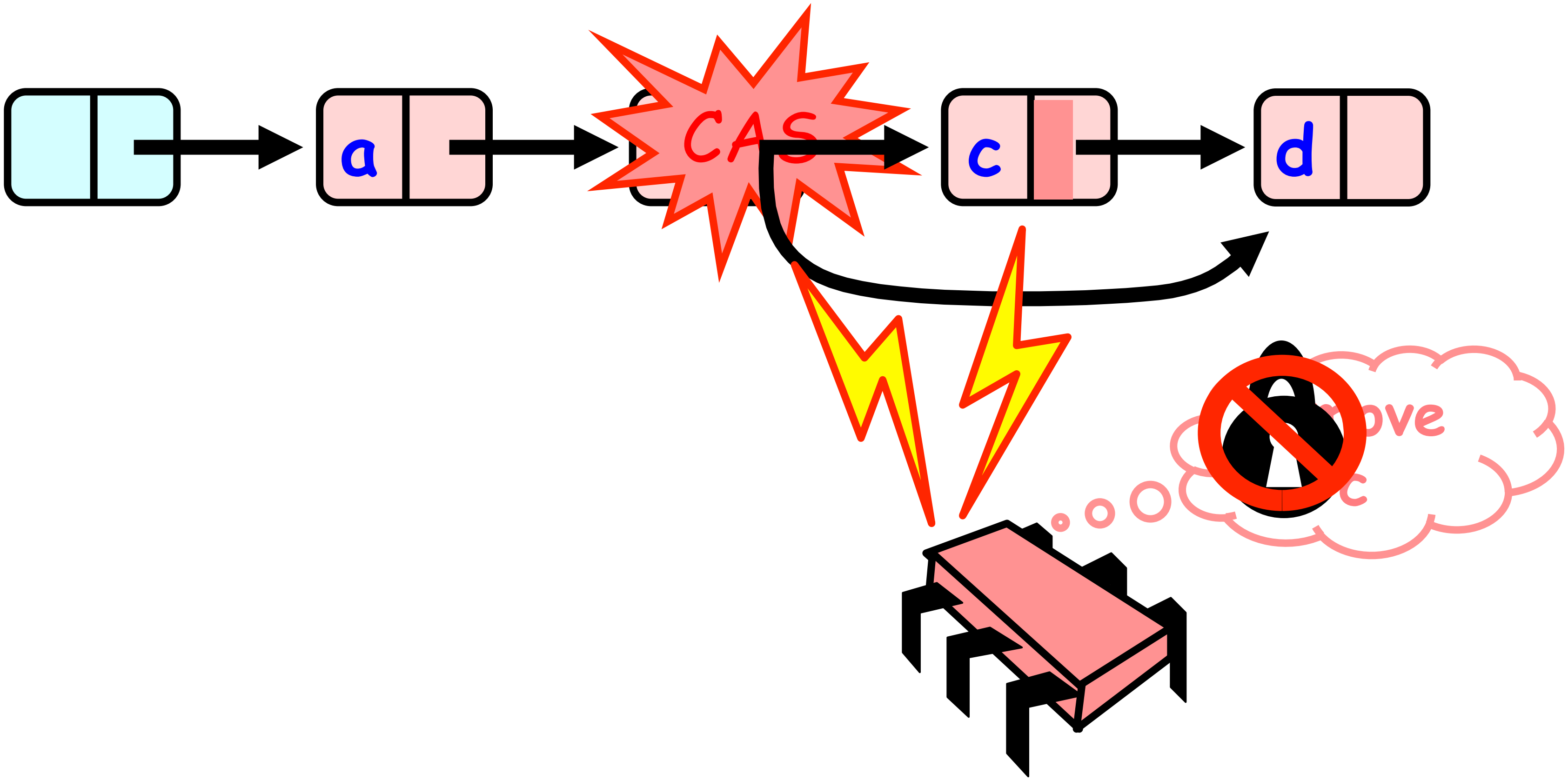
If this is the current reference …

Changing State

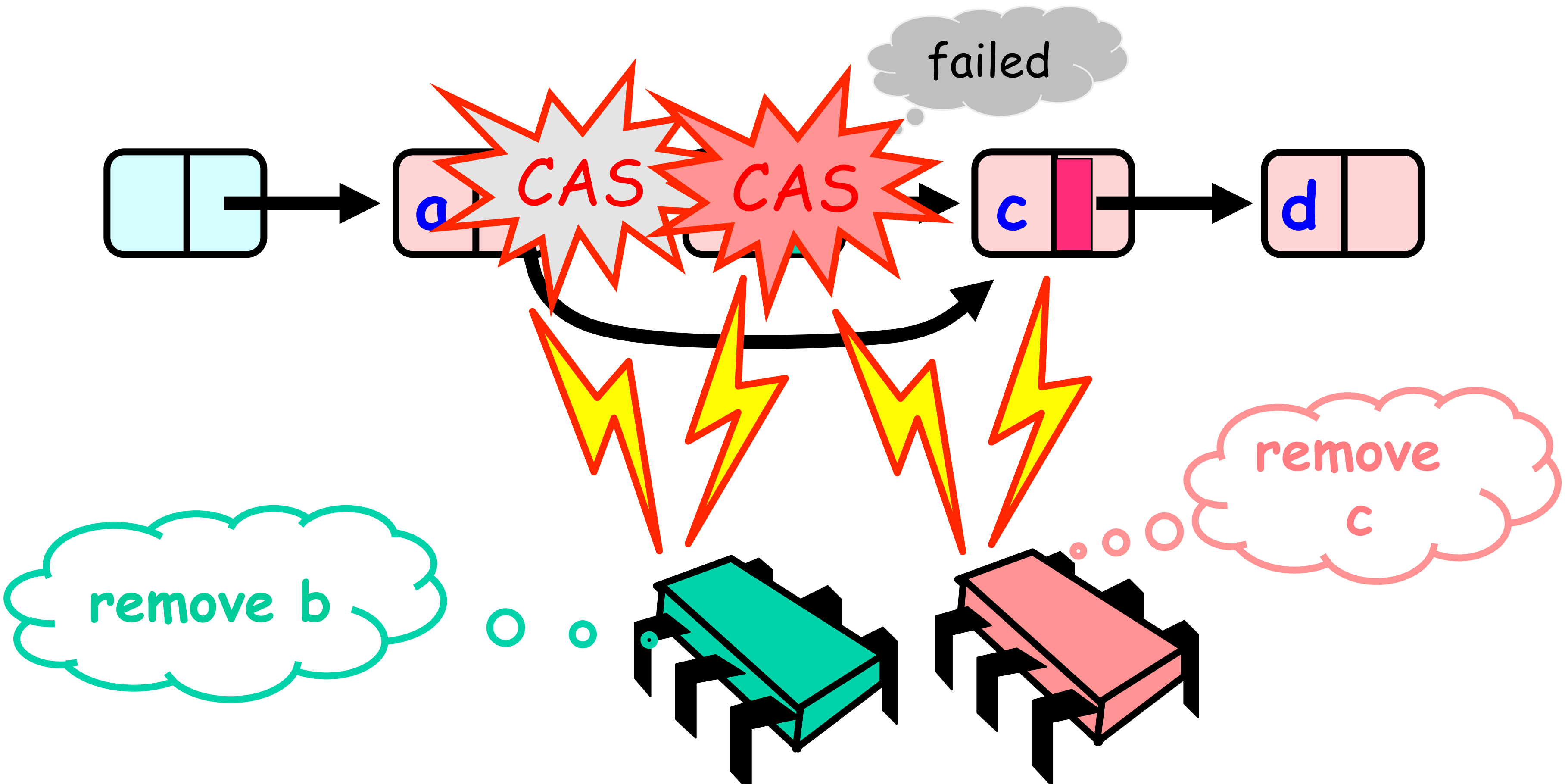
```
public boolean attemptMark(
   Object expectedRef,
   boolean updateMark);
```

.. then change to
this new mark.

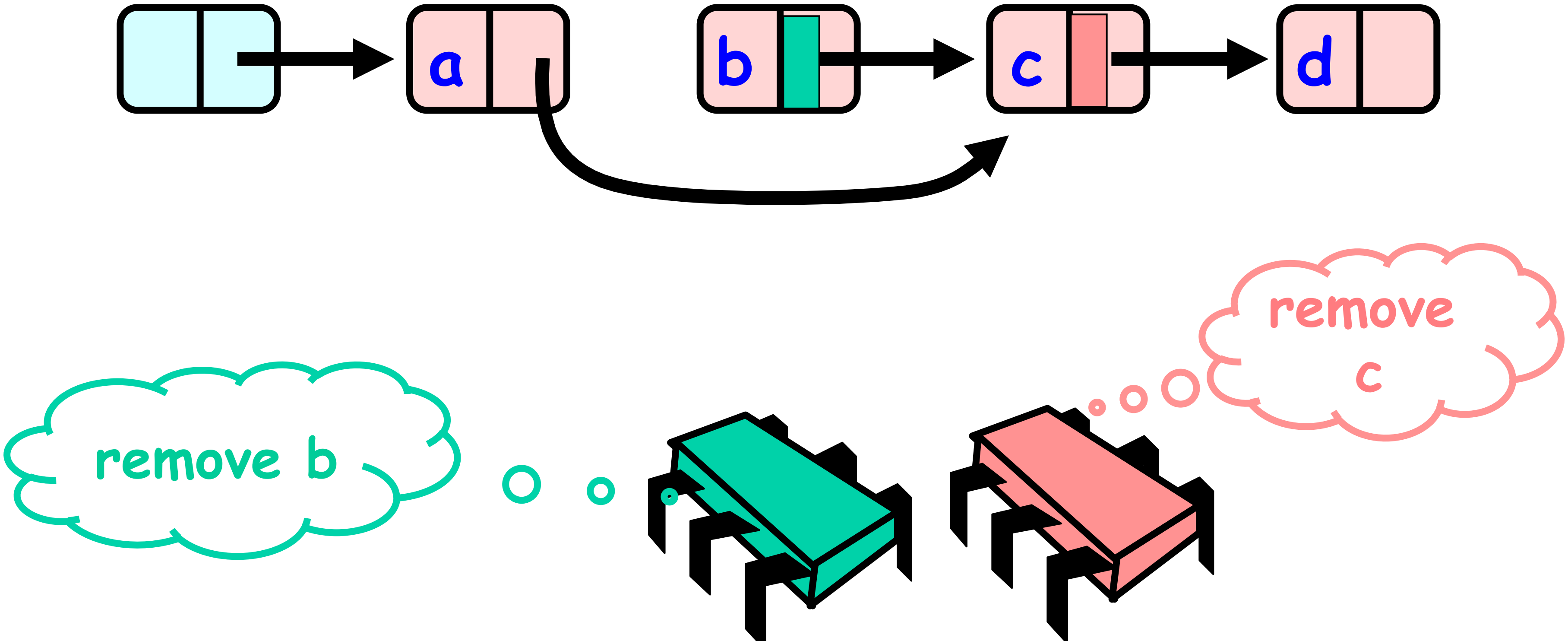
Removing a Node



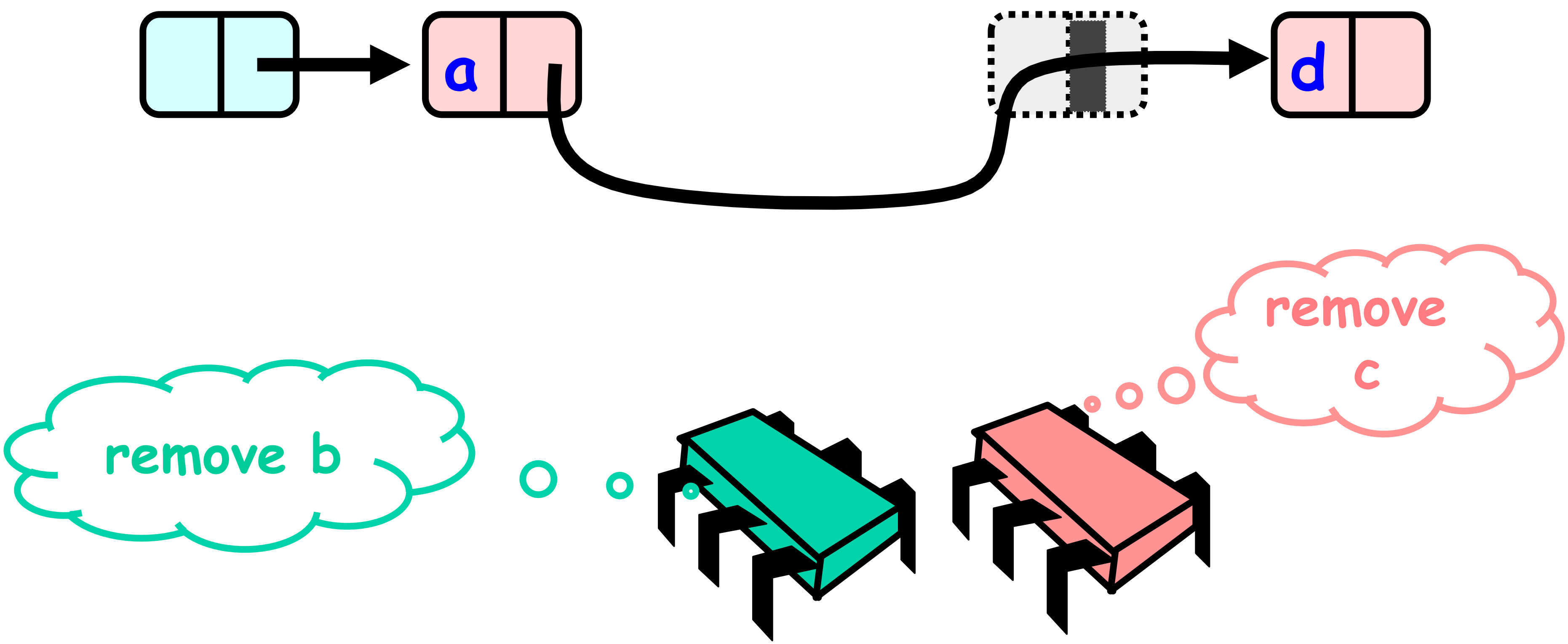
Removing a Node



Removing a Node



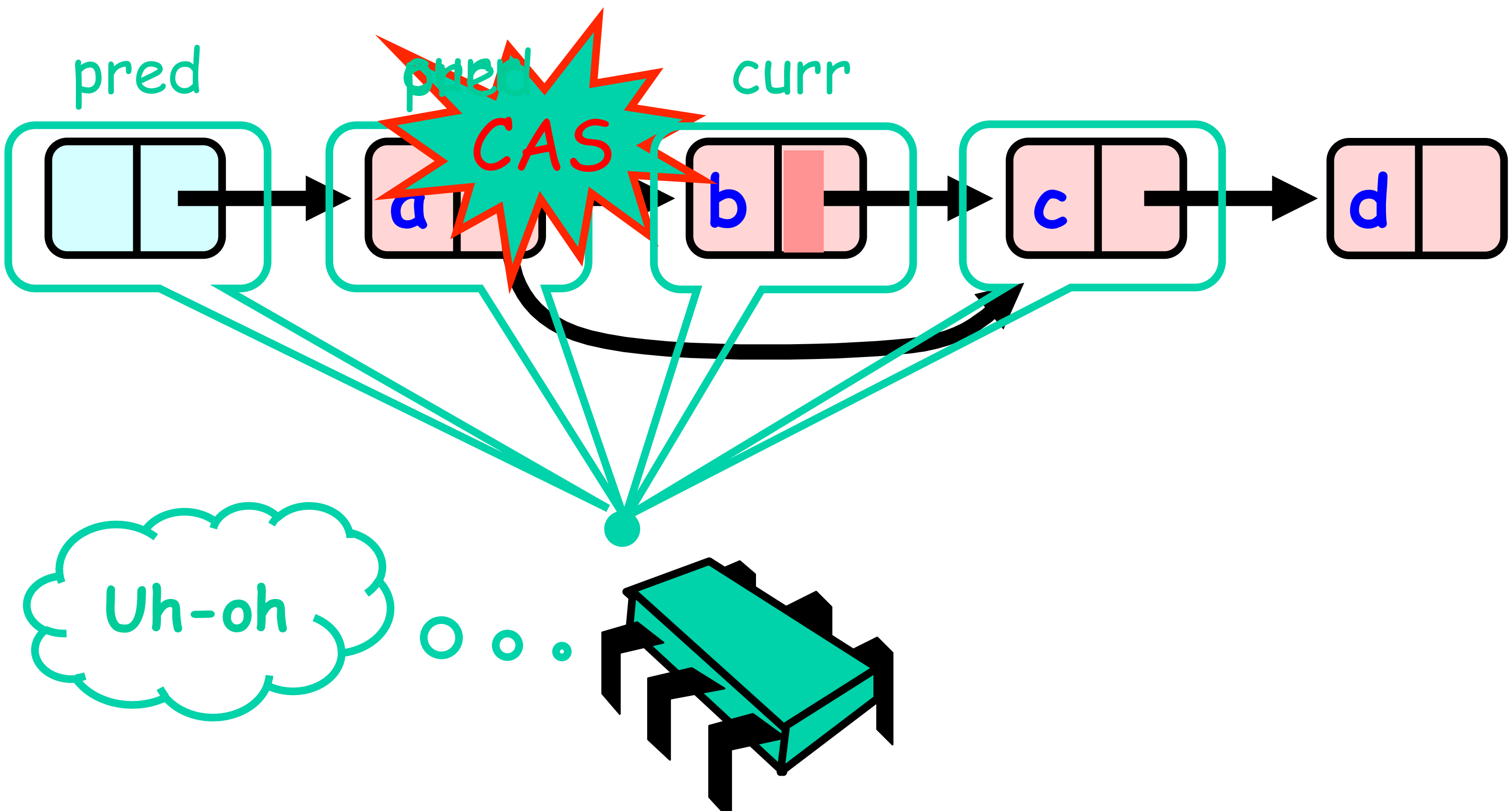
Removing a Node



Traversing the List

- Q: what do you do when you find a "logically" deleted node in your path?
- A: finish the job.
  - CAS the predecessor's next field
  - Proceed (repeat as needed)

Lock-Free Traversal (only Add and Remove)



The Window Class

```
class Window {
 public Node pred;
 public Node curr;
 Window(Node pred, Node curr) {
   this.pred = pred; this.curr = curr;
 }
}
```

The Window Class

```
class Window {
 public Node pred;
 public Node curr;
 Window(Node pred, Node curr) {
   this.pred = pred; this.curr = curr;
 }
}
```

A container for pred and current values

Using the Find Method

```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

- The `find()` method returns a structure containing the nodes on either side of the key. It removed marked nodes when it encounters them

Using the Find Method

```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

Find returns window

Using the Find Method

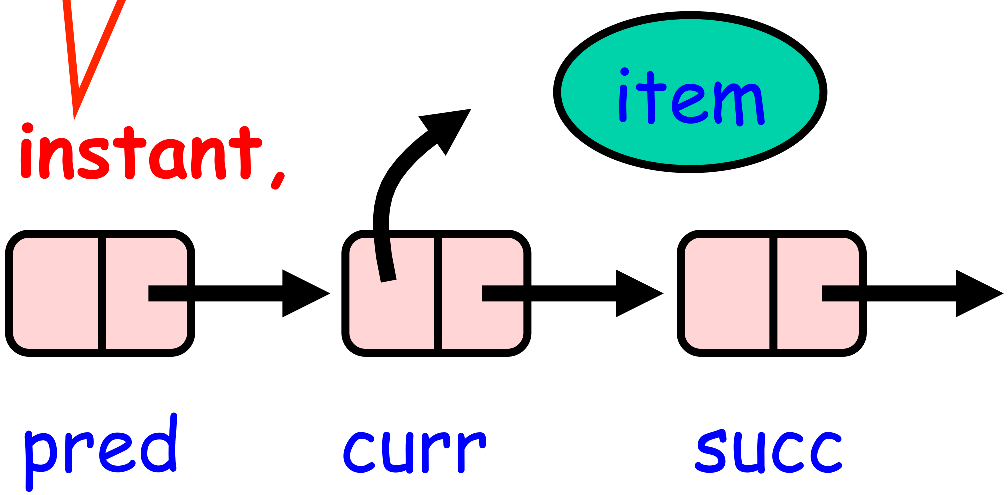
```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

Extract pred and curr

The Find Method

```
Window window = find(item);
```

At some instant,

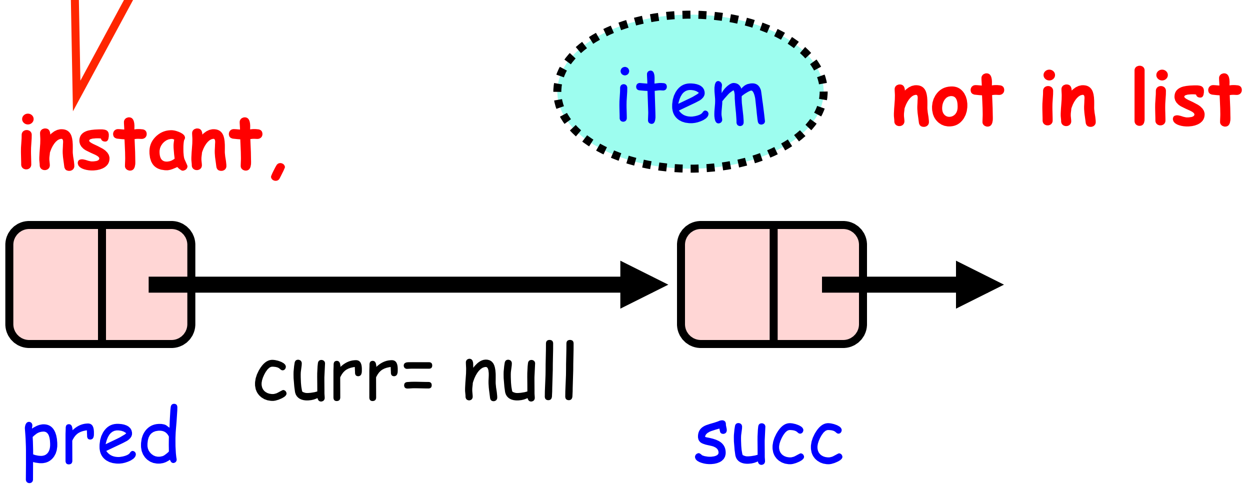


or …

The Find Method

```
Window window = find(item);
```

At some instant,



Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet (succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

Keep trying

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
Window window = find(head, key);
Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet (succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

Find neighbors

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

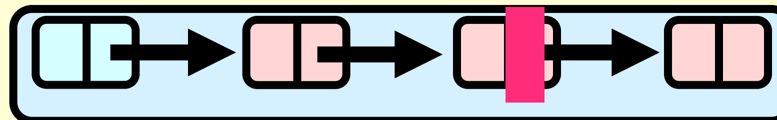
She's not there …

Remove

Try to mark node as deleted

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

Remove

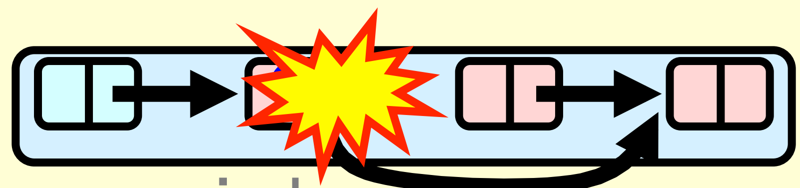


```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

If it doesn't work, just retry, if it does, job essentially done

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head,
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
   return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
   return true;
}}}
```



Try to advance reference
(if we don't succeed, someone else did or will).

Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

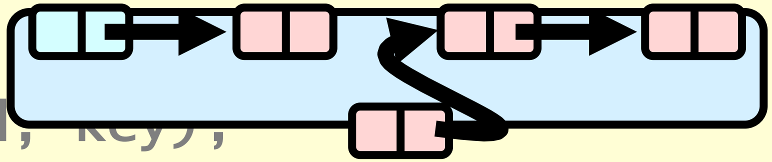
Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

Item already there.

Add

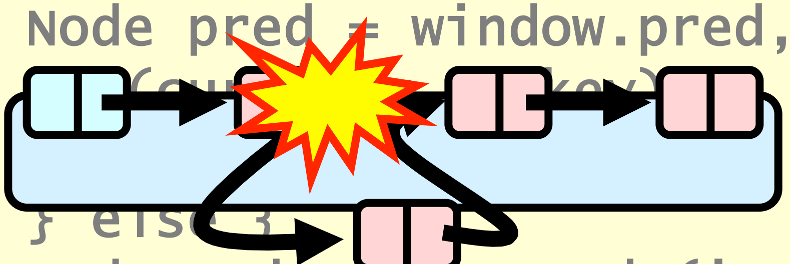
```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```



create new node

Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
```



```
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false, 
false)) {return true;}
}}}
```

Install new node, else retry loop

Wait-free Contains

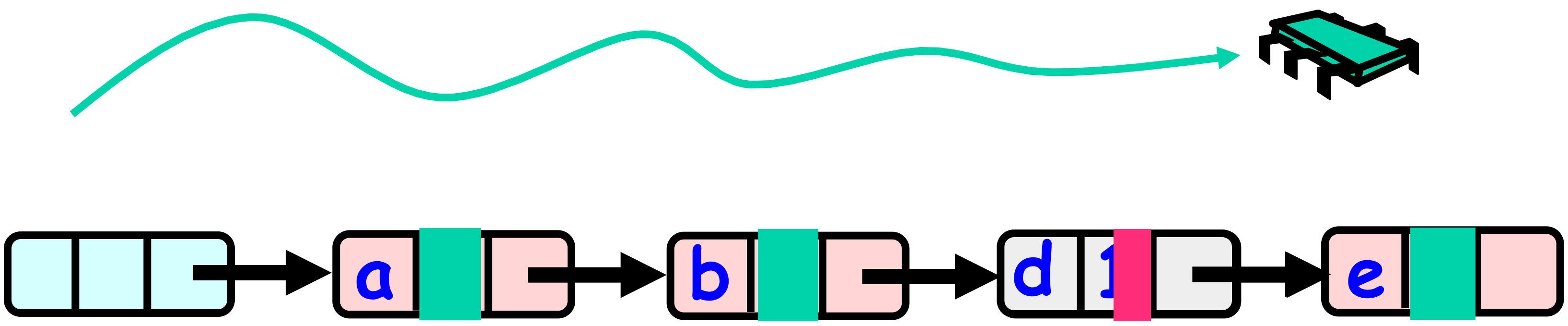
```
public boolean contains(T item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key)
      curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
  }
```

Wait-free Contains

```
public boolean contains(T item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key)
      curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
  }
```

get and check marked together with key

Summary: Wait-free Contains



Use Mark bit + list ordering

1. Not marked → in the set
2. Marked or missing → not in the set

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
          return new Window(pred, curr);
        pred = curr;
        curr = succ;
     }
}}
```

Lock-free Find

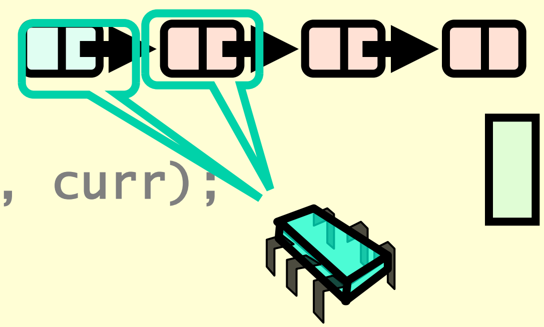
```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
         return new Window(pred, curr);
        pred = curr;
        curr = succ;
     }
}}
```

If list changes while traversed, start over Lock-Free because we start over only if someone else makes progress

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
          return new Window(pred, curr);
        pred = curr;
        curr = succ;
     }
}}
```

Start looking from head



Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
   if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
}}
```

Move down the list

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
          return new Window(pred, curr);
        pred = curr;
        curr = succ;
     }
}}
```

Get ref to successor and current deleted bit

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
         return new Window(pred, curr);
       pred = curr;
```

Try to remove deleted nodes in path…code details soon

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
          return new Window(pred, curr);
        pred = curr;
        curr = succ;
     }
}}
```

If curr key that is greater or equal, return pred and curr

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
     …
    }
    if (curr.key >= key)
          return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
}}
```

Otherwise advance window and loop again

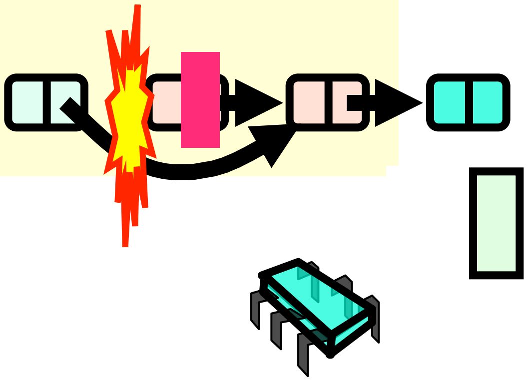
Lock-free Find

```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                                    succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```

Lock-free Find

Try to snip out node

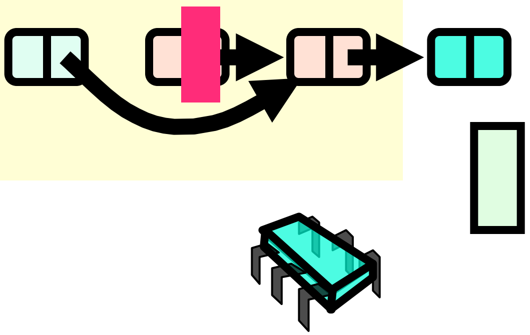
```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                              succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```



Lock-free Find

if predecessor's next field changed must retry whole traversal

```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```



Lock-free Find

Otherwise move on to check if next node deleted

```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                                    succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```

“To Lock or Not to Lock”

- Locking vs. Non-blocking: Extremist views on both sides
- The answer: nobler to compromise, combine locking and non-blocking
  - Remember: Blocking/non-blocking is a property of a method
  - Individual methods of a given item implementation don’t necessarily provide the same properties