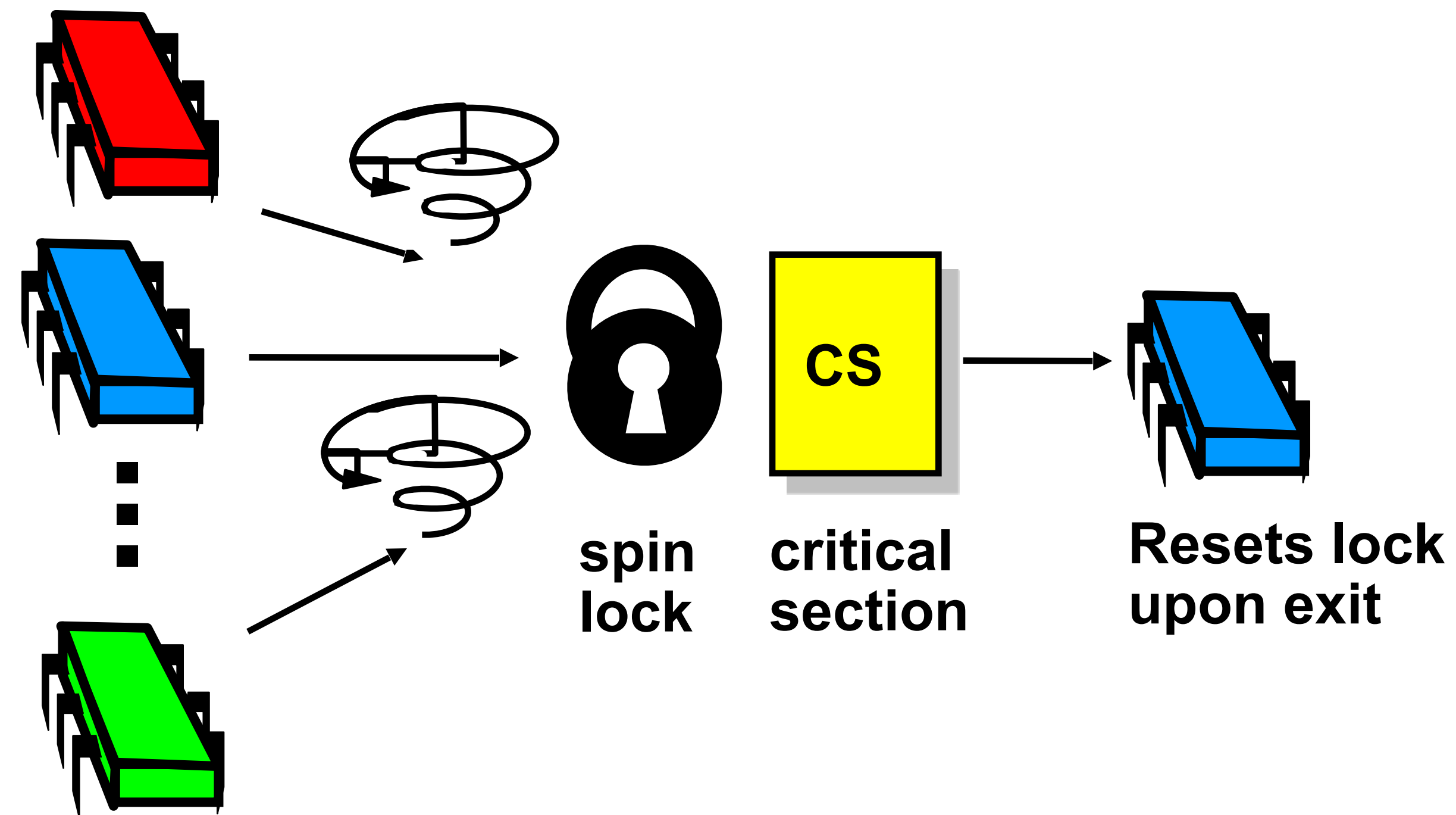


# YSC4231: Parallel, Concurrent and Distributed Programming

Concurrent Linked Lists

# Previous Lectures: Spin-Locks



# Today: More Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Can be mitigated by back-offs, arrays, etc.

# Today: More Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Can be mitigated by back-offs, arrays, etc.
- Should increase throughput
  - Not possible if inherently sequential
  - Surprising things are parallelizable

# Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using queue locks

# Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using locks
  - Easy to reason about
    - In simple cases

# Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using locks
  - Easy to reason about
    - In simple cases
- So, are we done?

# Coarse-Grained Synchronization

- Sequential bottleneck
  - Threads “stand in line”



# Coarse-Grained Synchronization

- Sequential bottleneck
  - Threads “stand in line”
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse

# Coarse-Grained Synchronization

- 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 ...

# This Lecture

- Introduce several “patterns”
  - Bag of tricks ...
  - Methods that work more than once ...

# This Lecture

- Introduce several “patterns”
  - Bag of tricks ...
  - Methods that work more than once ...
- For highly-concurrent objects
  - Concurrent access
  - More threads, more throughput

# First: Fine-Grained Synchronization

- Instead of using a single lock ...

# First: Fine-Grained Synchronization

- Instead of using a single lock ...
- Split object into
  - Independently-synchronized components

# 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 ...



# Second: Optimistic Synchronization

- Search without locking ...
- If you find it, lock and check ...
  - OK: we are done
  - Oops: start over

# Second: Optimistic Synchronization

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

# Third: Lazy Synchronization

- Postpone hard work

# Third: Lazy Synchronization

- Postpone hard work
- Removing components is tricky

# Third: Lazy Synchronization

- Postpone hard work
- Removing components is tricky
  - Logical removal
    - Mark component to be deleted

# Third: Lazy Synchronization

- Postpone hard work
- 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 ...

# Fourth: Lock-Free Synchronization

- Don't use locks at all
  - Use `compareAndSet()` & relatives ...
- Advantages
  - No Scheduler Assumptions/Support



# Fourth: Lock-Free Synchronization

- Don't use locks at all
  - Use `compareAndSet()` & relatives ...
- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead

# Linked List

- Illustrate these patterns ...
- Using a list-based Set
  - Common application
  - Building block for other apps

# Set Interface

- Unordered collection of items

# Set Interface

- Unordered collection of items
- No duplicates

# 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

Warm-up:  
Testing Concurrent Sets

# List-Based Sets

```
trait ConcurrentSet[T] {  
  def add(item: T): Boolean  
  def remove(item: T): Boolean  
  def contains(item: T): Boolean  
}
```

# List-Based Sets

```
trait ConcurrentSet[T] {  
  def add(item: T): Boolean  
  def remove(item: T): Boolean  
  def contains(item: T): Boolean  
}
```

**Add item to set**



# List-Based Sets

```
trait ConcurrentSet[T] {  
  def add(item: T): Boolean  
  def remove(item: T): Boolean  
  def contains(item: T): Boolean  
}
```

**Remove item from set**

# List-Based Sets

```
trait ConcurrentSet[T] {  
  def add(item: T): Boolean  
  def remove(item: T): Boolean  
  def contains(item: T): Boolean  
}
```

**Is item in set?**

# List Node

```
class Node (val item: T) {  
  def key : Int  
  @volatile var next: Node = _  
}
```

# List Node

```
class Node (val item: T) {  
  def key : Int  
  @volatile var next: Node = _  
}
```

**item of interest**

# List Node

```
class Node (val item: T) {  
  def key : Int  
  @volatile var next: Node = _  
}
```

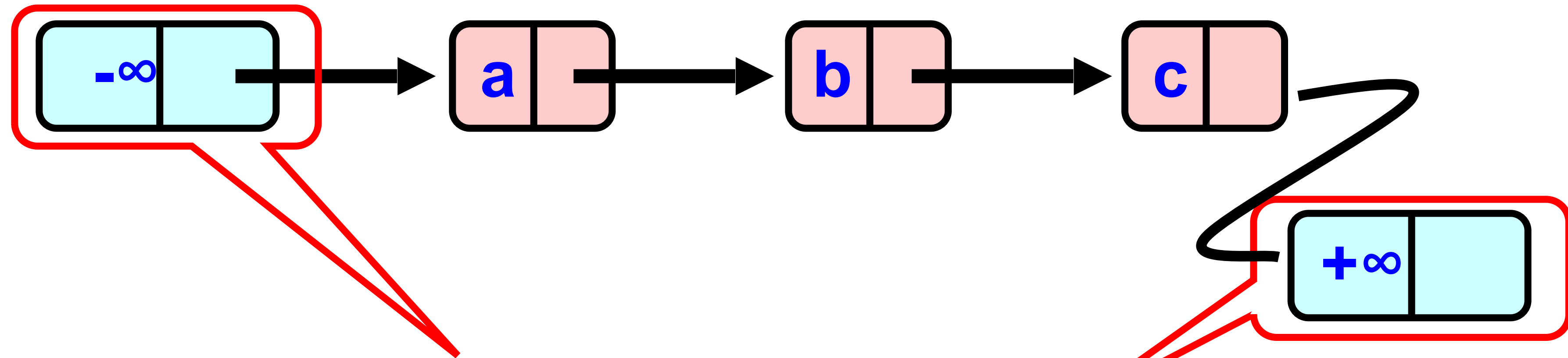
**Usually hash code**

# List Node

```
class Node (val item: T) {  
  def key : Int  
  @volatile var next: Node = _  
}
```

**Reference to next node**

# The List-Based Set



Sorted with Sentinel nodes  
(min & max possible keys)

# Reasoning about Concurrent Objects

- Invariant
  - Property that always holds



# Reasoning about Concurrent Objects

- Invariant
  - Property that always holds
- Established because
  - True when object is **created**
  - Truth **preserved** by each method
    - Each **step** of each method

# Specifically ...

- Invariants preserved by
  - `add()`
  - `remove()`
  - `contains()`

# Specifically ...

- Invariants preserved by
  - `add()`
  - `remove()`
  - `contains()`
- Most steps are trivial
  - Usually one step tricky
  - Often it is the linearization point

# Interference

- Invariants make sense only if
  - methods considered
  - are the only modifiers

# Interference

- Invariants make sense only if
  - methods considered
  - are the only modifiers
- Language encapsulation helps
  - List nodes *not visible* outside class

# Interference

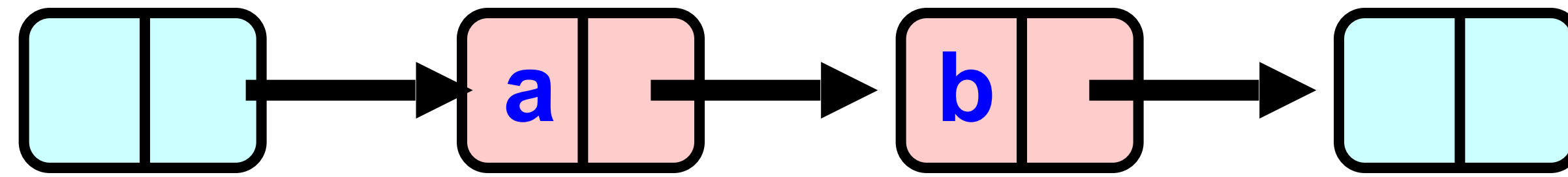
- Invariants make sense only if
  - methods considered
  - are the only modifiers
- Language encapsulation helps
  - List nodes *not visible* outside class
- Similar to *loop invariants*
  - Each method must preserve the invariant (same as each loop iteration)

# Interference

- Freedom from interference needed even for removed nodes
  - Some algorithms traverse removed nodes
  - Careful with `malloc()` & `free()`!
- We rely on garbage collection

# Recap: Abstract Data Types

- Concrete representation:



- Abstract Type:  
{**a**, **b**}



# Abstract Data Types

- Meaning of rep given by abstraction map

$$S(\text{[ ]} \rightarrow \text{[a]} \rightarrow \text{[b]} \rightarrow \text{[ ]}) = \{a, b\}$$

# Representation Invariant

- Which concrete values meaningful?
  - Sorted?
  - Duplicates?
- Representation invariant
  - Characterises legal *concrete representations*
  - Preserved by methods
  - Relied on by methods

# Blame Game

- Rep invariant is a **contract**
- Suppose
  - **add ()** leaves behind 2 copies of x
  - **remove ()** removes only 1
- Which is incorrect?

# Blame Game

- Suppose
  - **add ()** leaves behind 2 copies of x
  - **remove ()** removes only 1

# Blame Game

- Suppose
  - **add ()** leaves behind 2 copies of x
  - **remove ()** removes only 1
- Which is incorrect?
  - If rep invariant says *no duplicates*
    - **add ()** is incorrect
  - Otherwise
    - **remove ()** is incorrect

# Lists' Rep Invariant (partly)

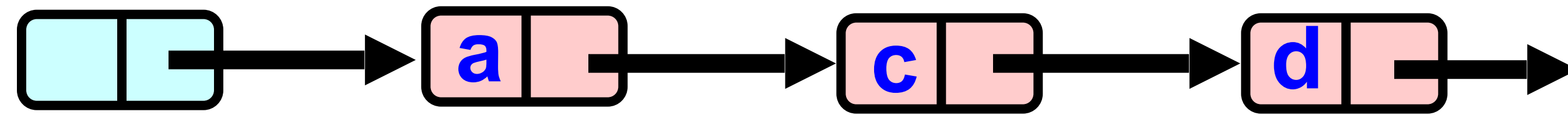
- Sentinel nodes
  - tail reachable from head
- Sorted
- No duplicates

# Abstraction Map

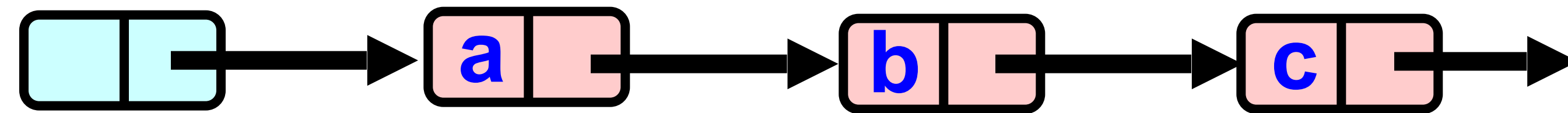
- $S(\text{head}) =$   
 $\{ x \mid \text{there exists } a \text{ such that}$ 
  - $a \text{ reachable from head and}$
  - $a.\text{item} = x$ $\}$

# Sequential List Based Set

add()



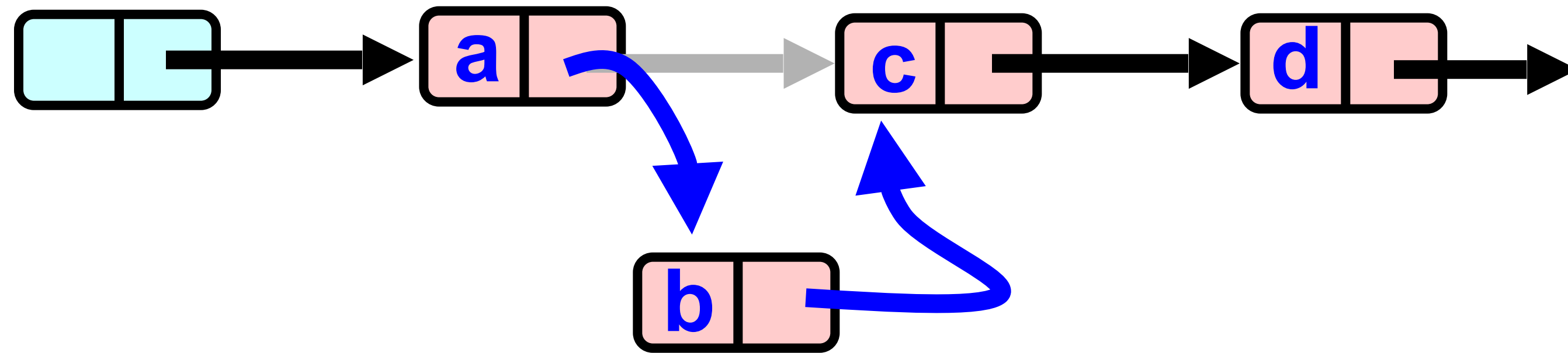
remove()



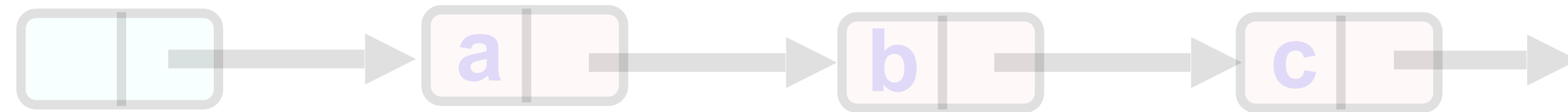


# Sequential List Based Set

add ()

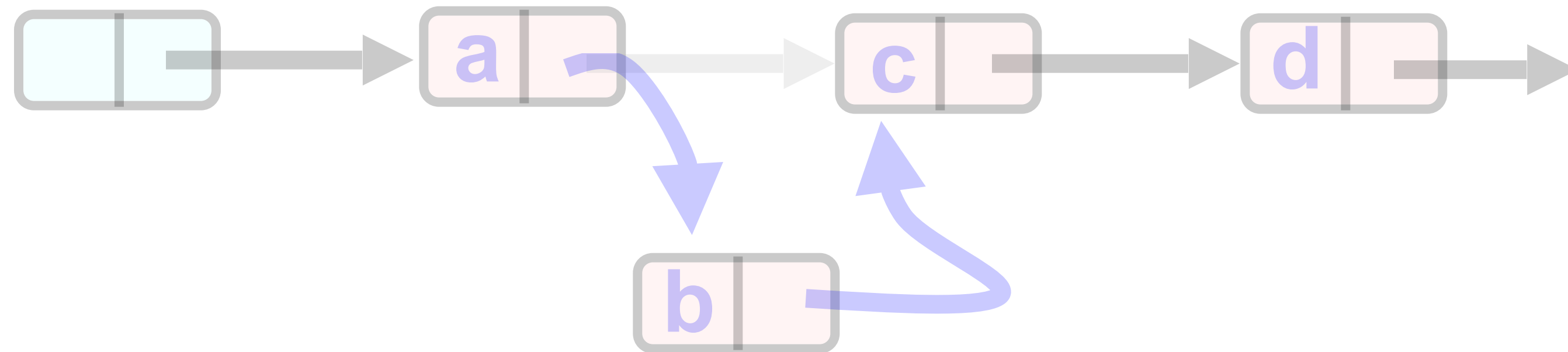


remove ()

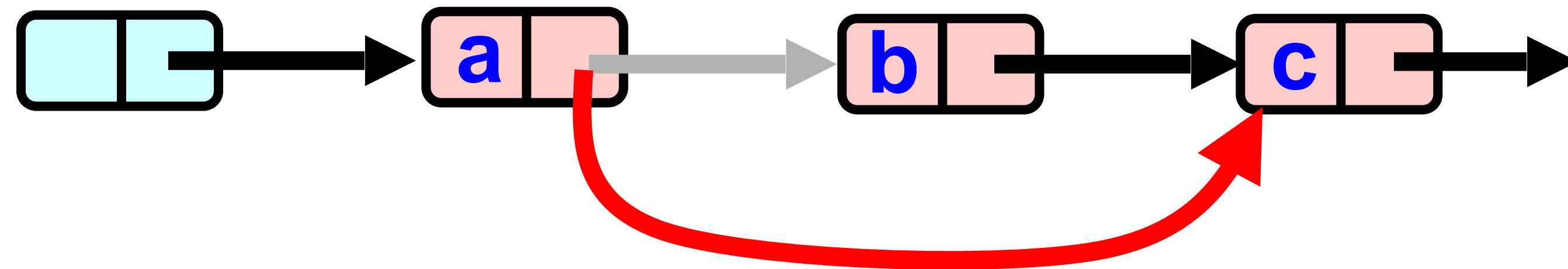


# Sequential List Based Set

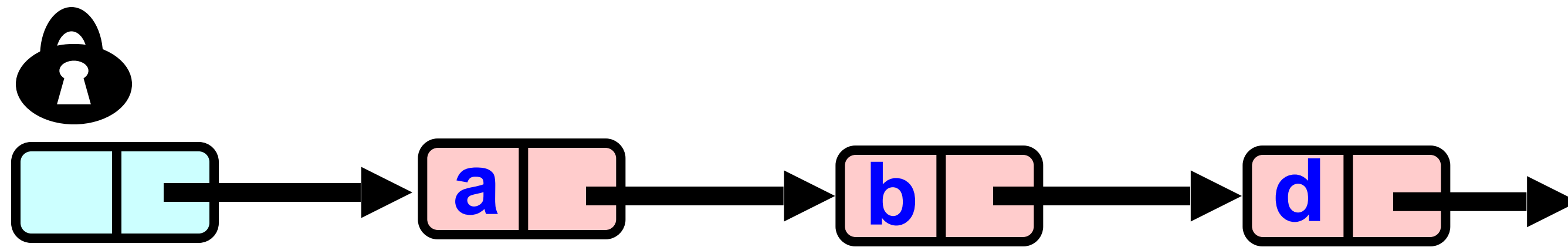
add()



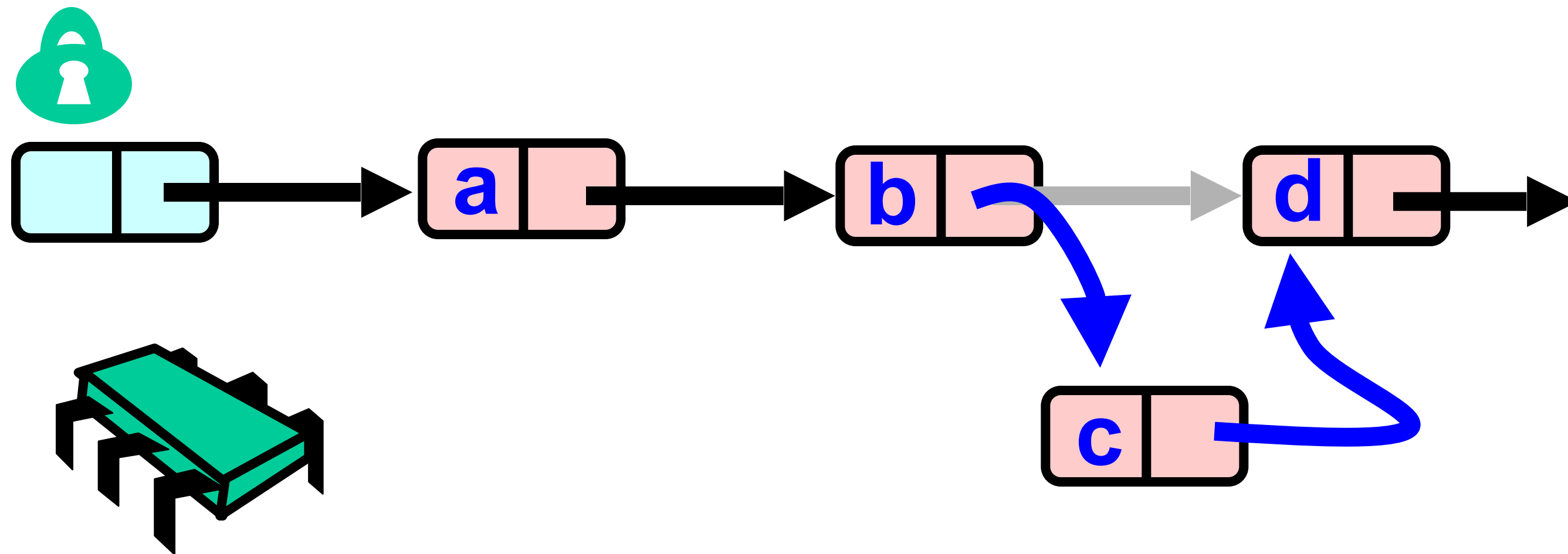
remove()



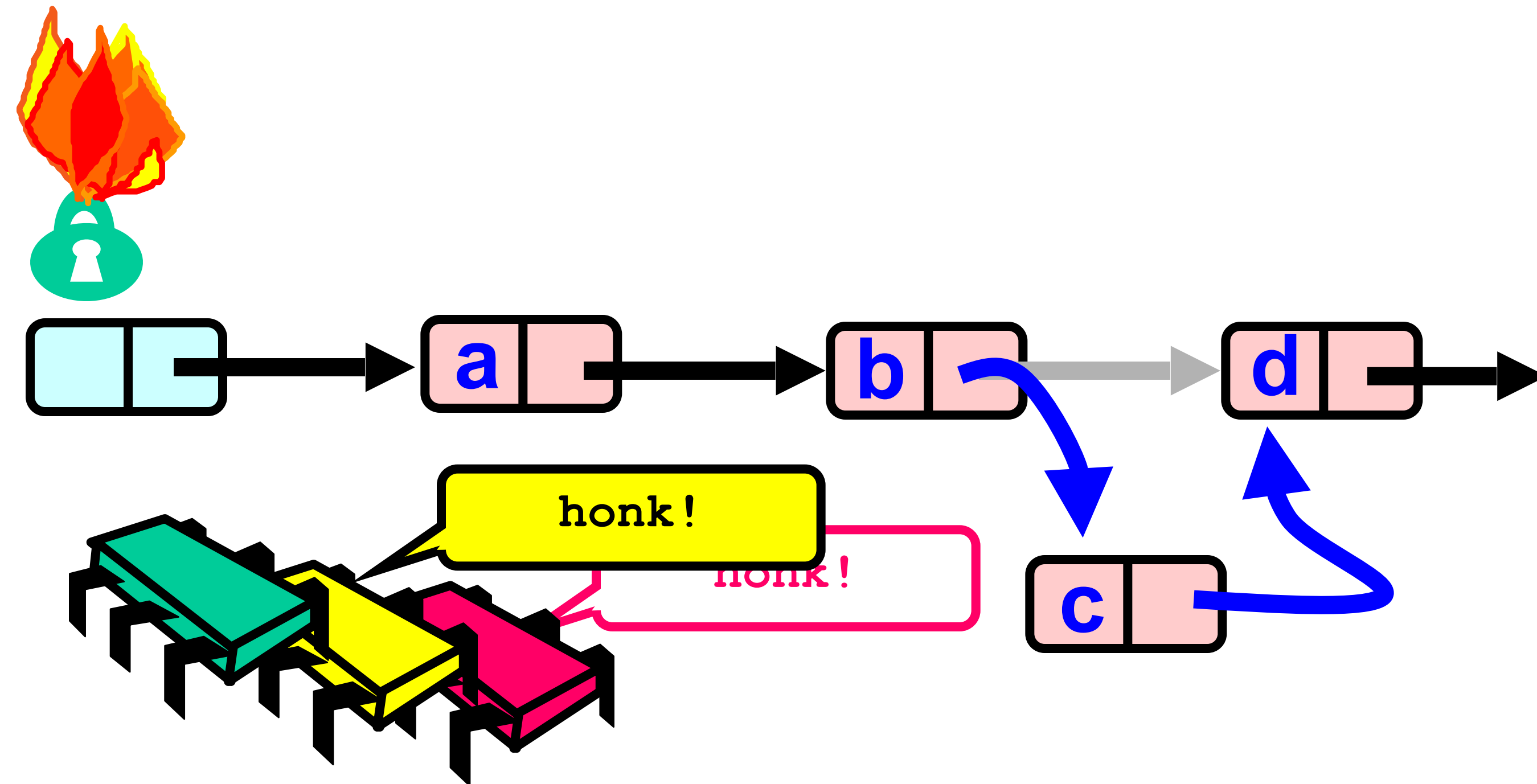
# Coarse-Grained Locking



# Coarse-Grained Locking



# Coarse-Grained Locking



Simple but hotspot + bottleneck

# Coarse-Grained Locking

- Easy, same as synchronized methods
  - “One lock to rule them all ...”

# 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

# Fine-grained Locking

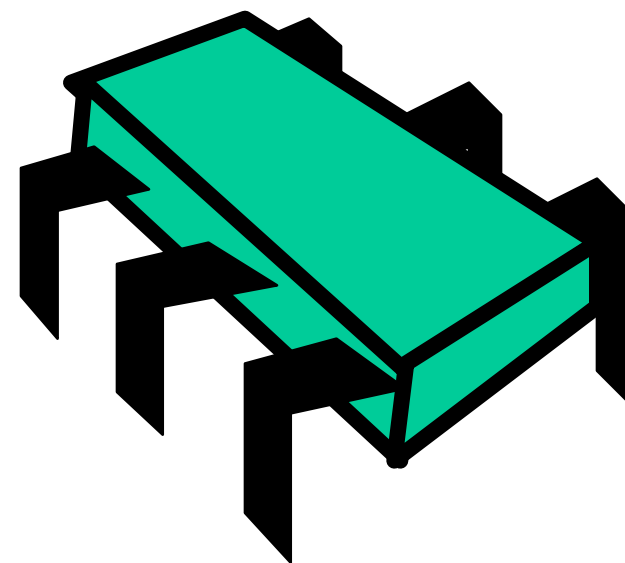
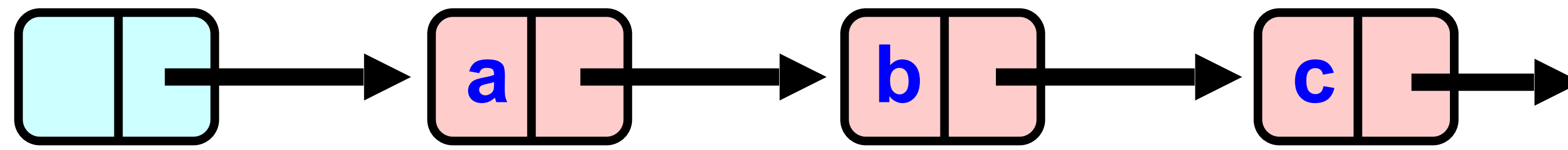
- Requires **careful thought**
  - “Do not meddle in the affairs of wizards, for they are subtle and quick to anger”



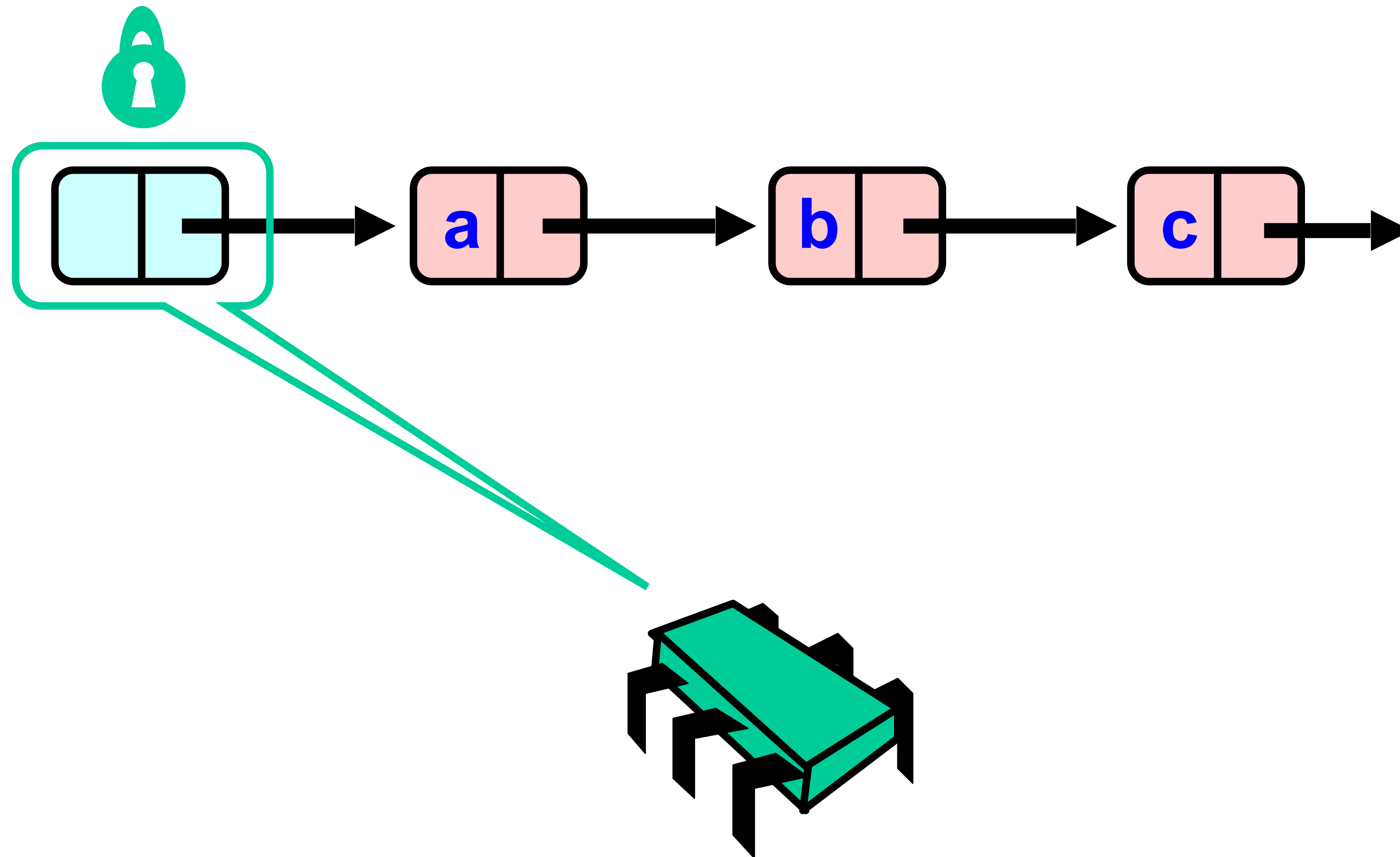
# Fine-grained Locking

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

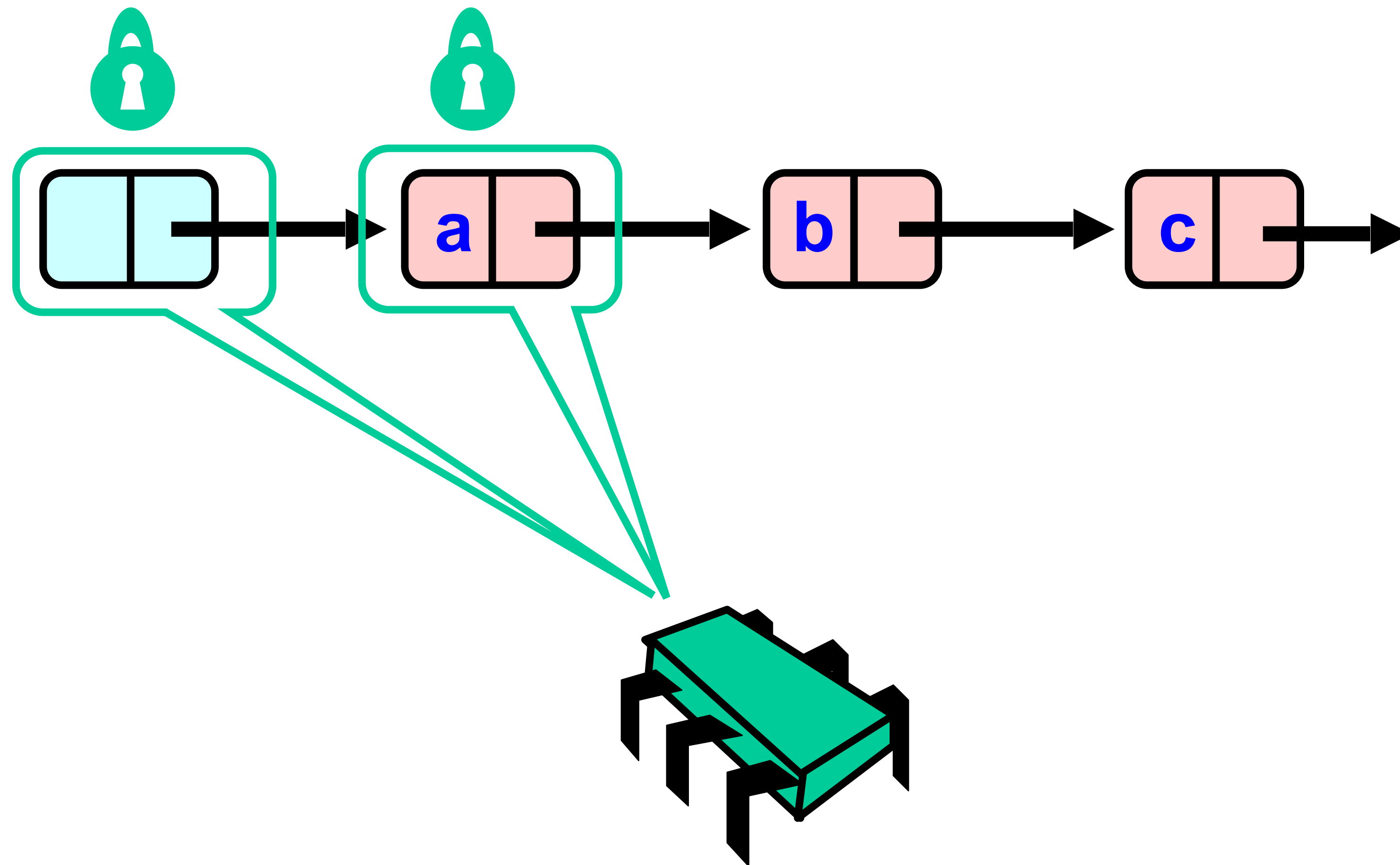
# Hand-over-Hand locking



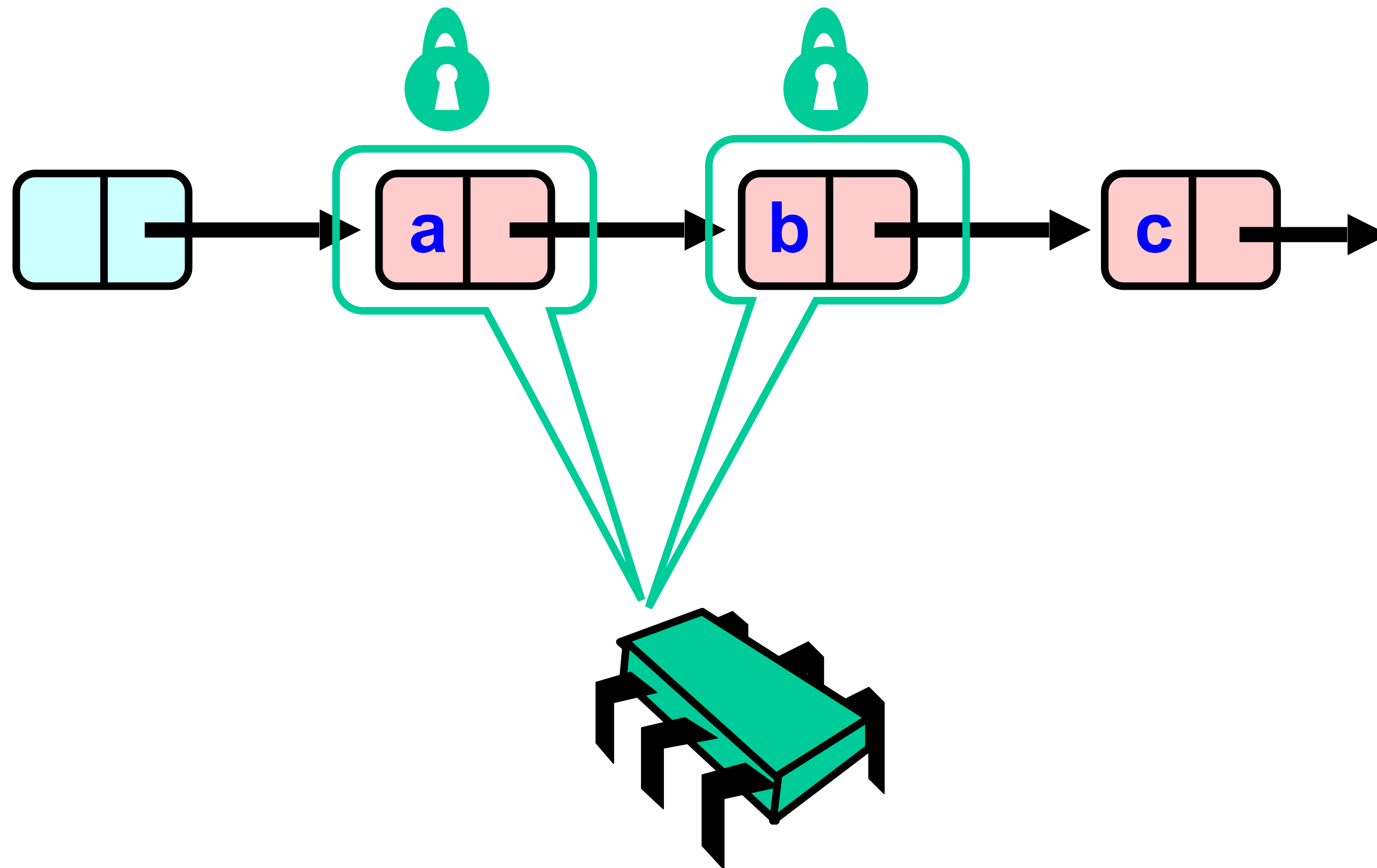
# Hand-over-Hand locking



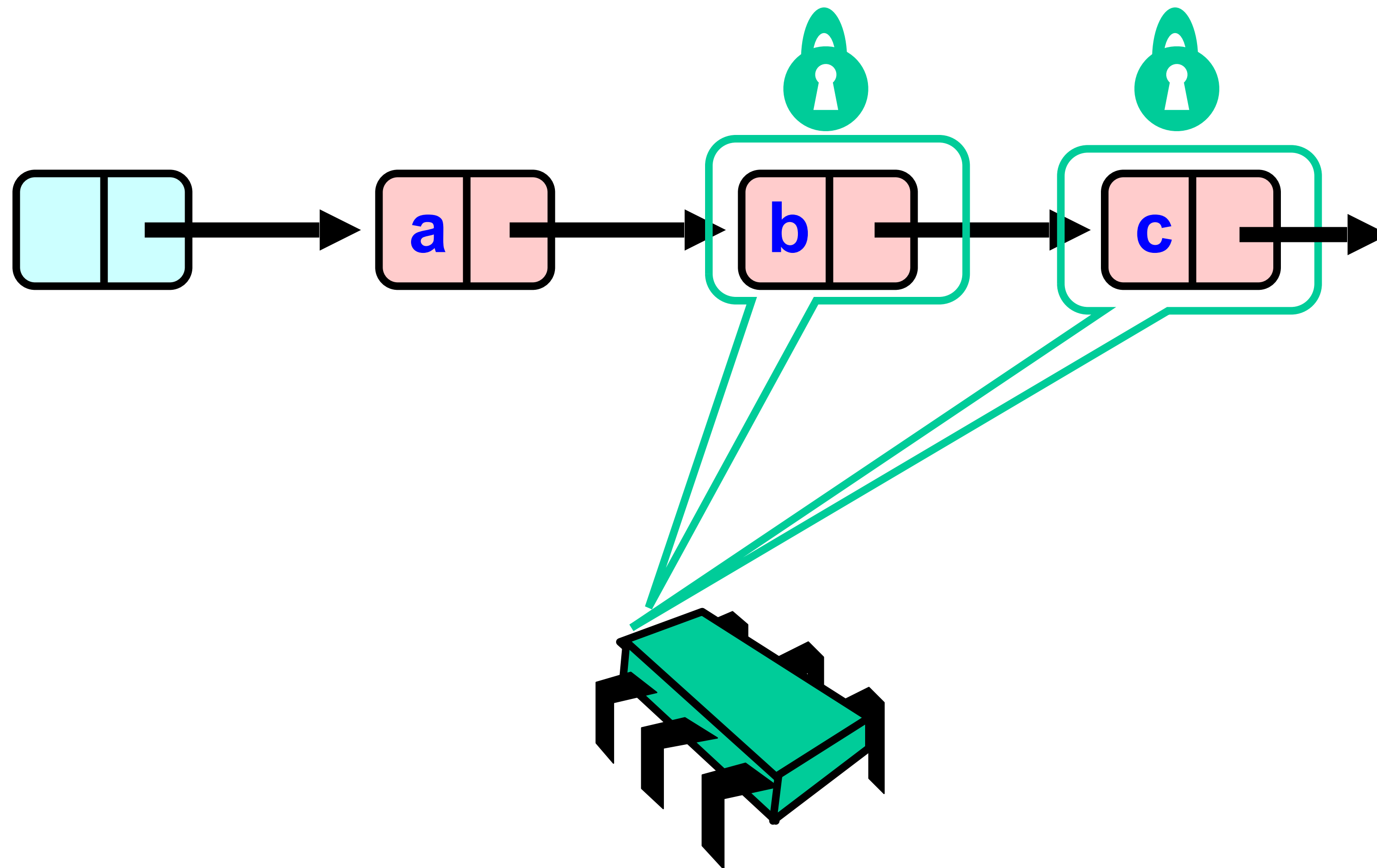
# Hand-over-Hand locking



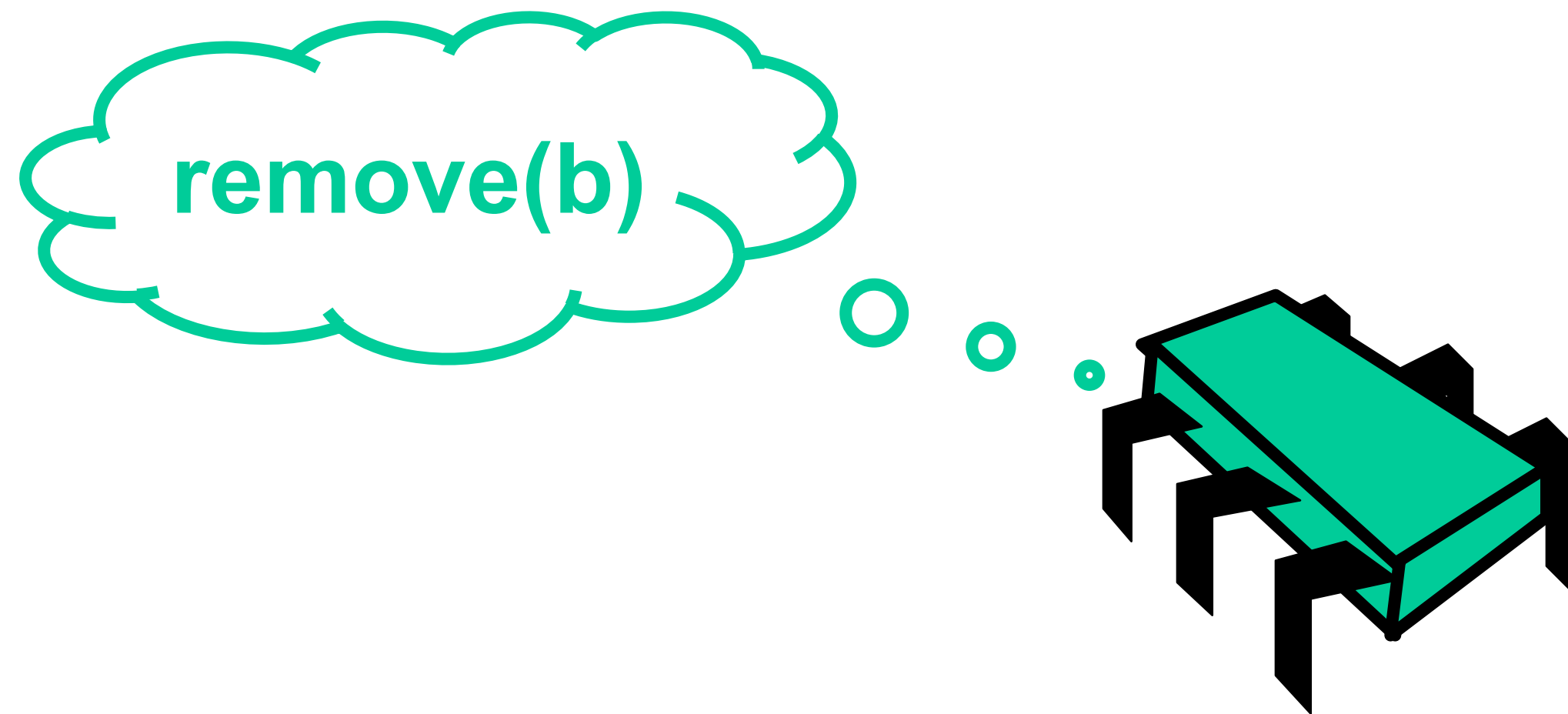
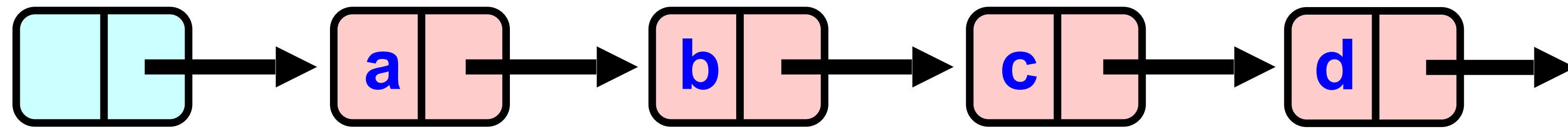
# Hand-over-Hand locking



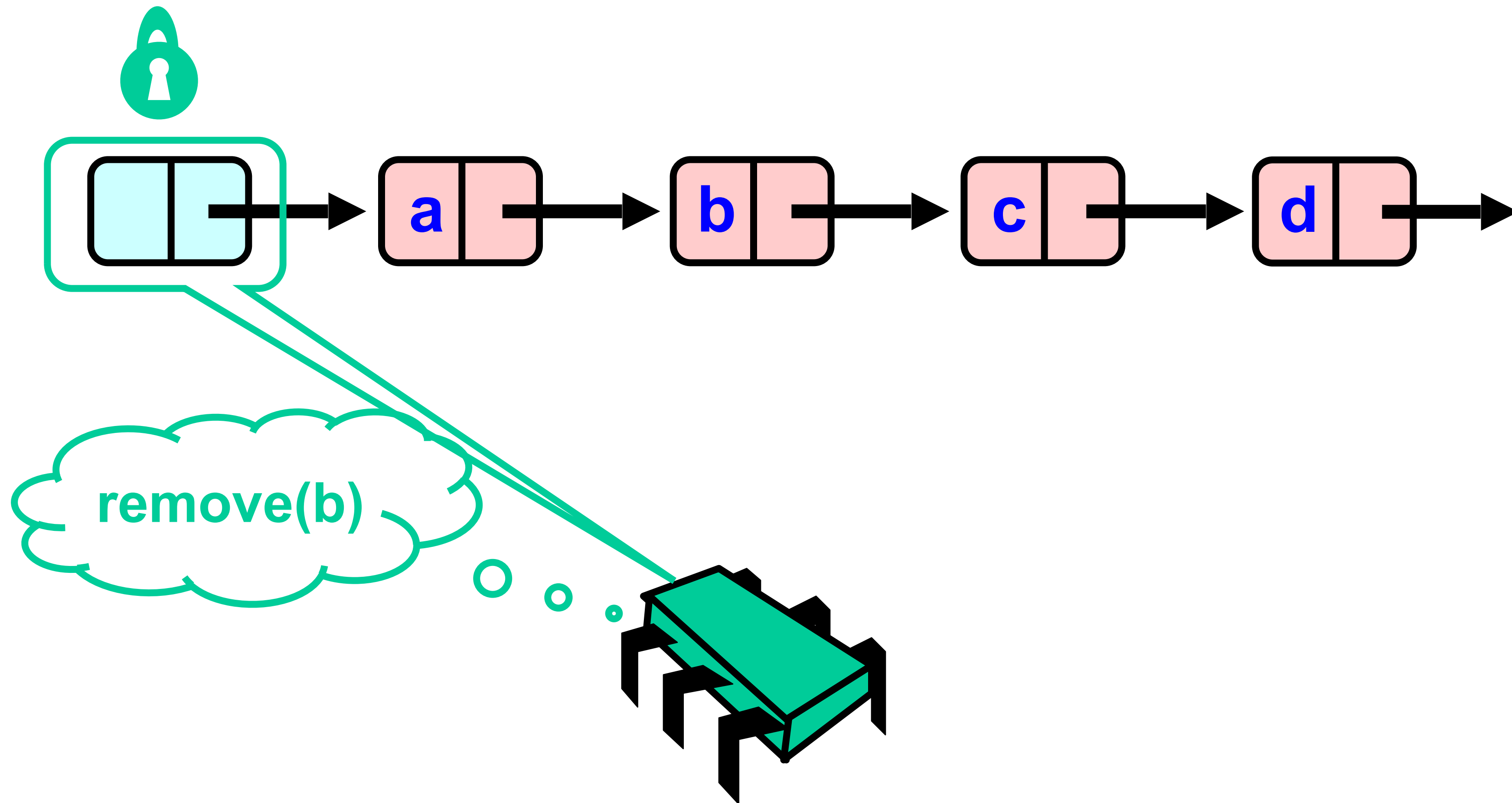
# Hand-over-Hand locking



# Removing a Node

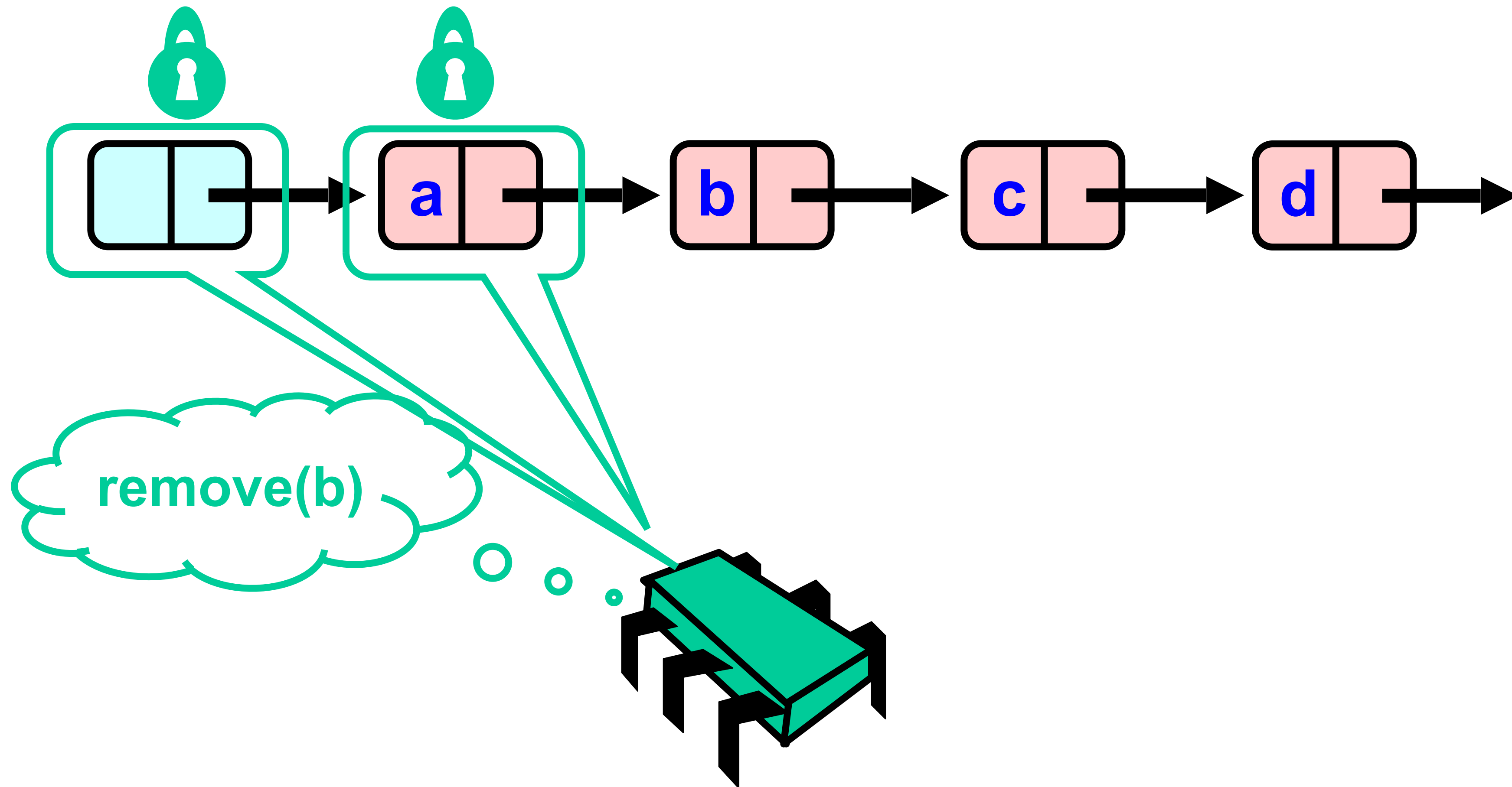


# Removing a Node

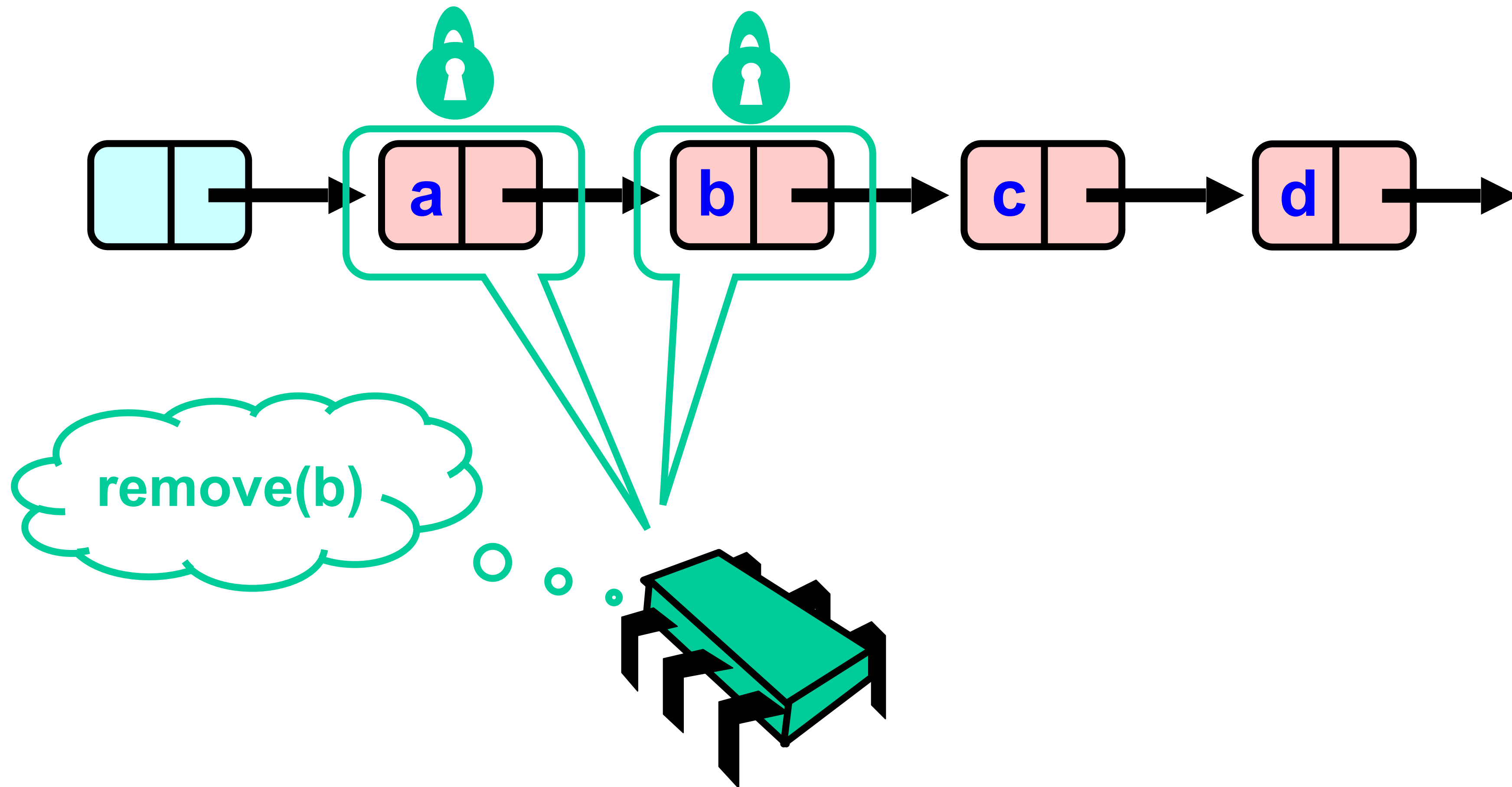




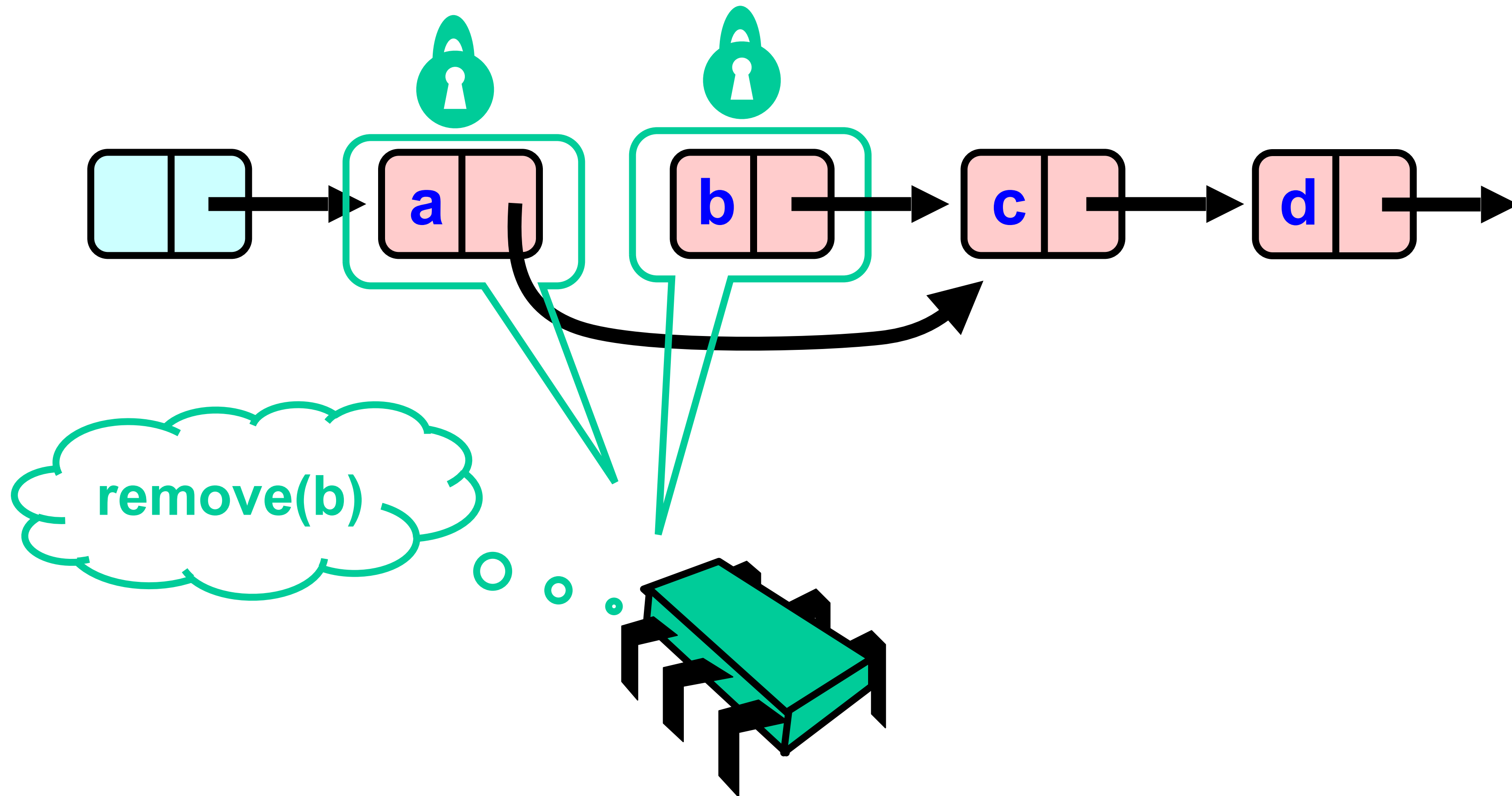
# Removing a Node



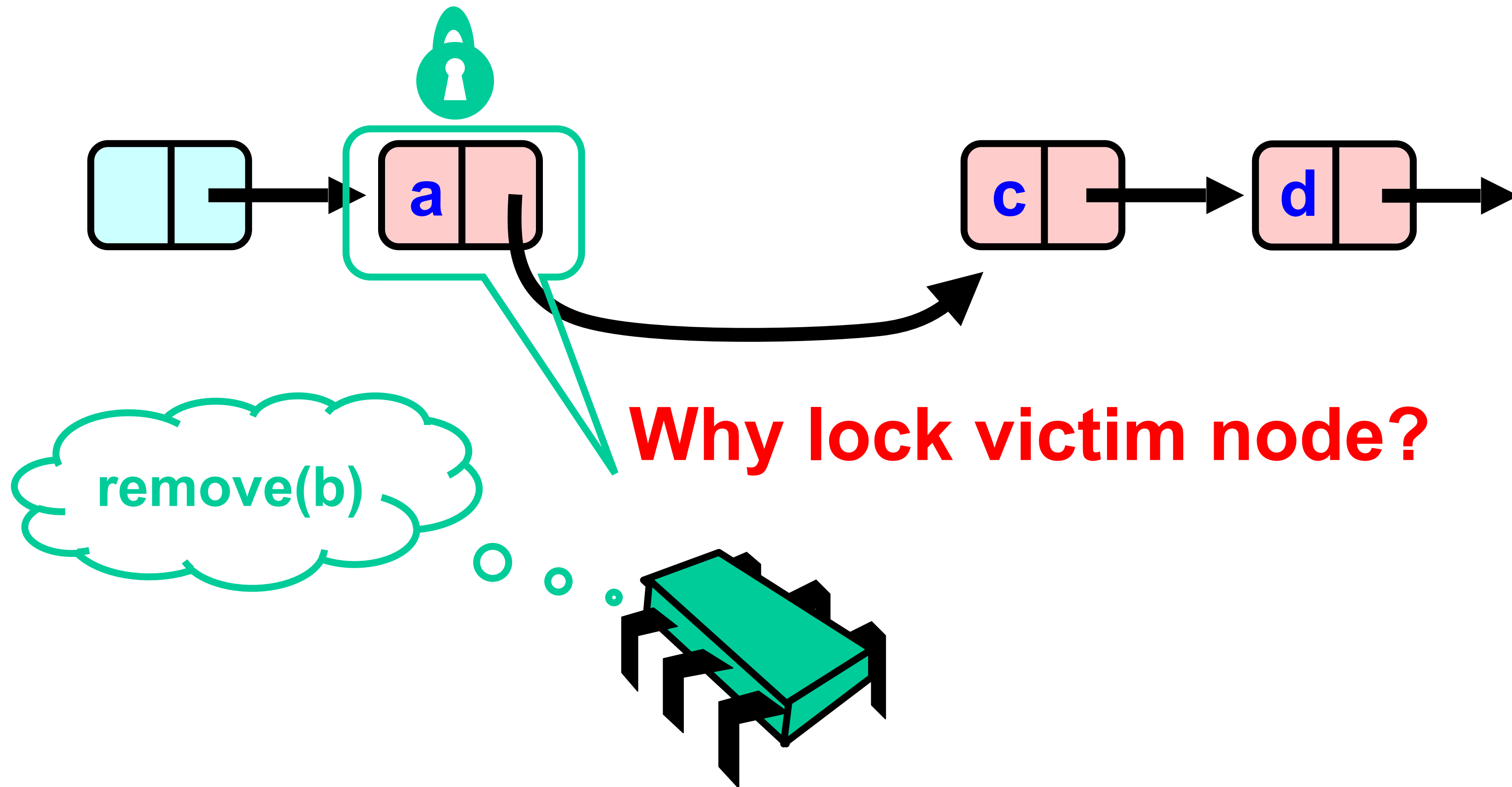
# Removing a Node



# Removing a Node

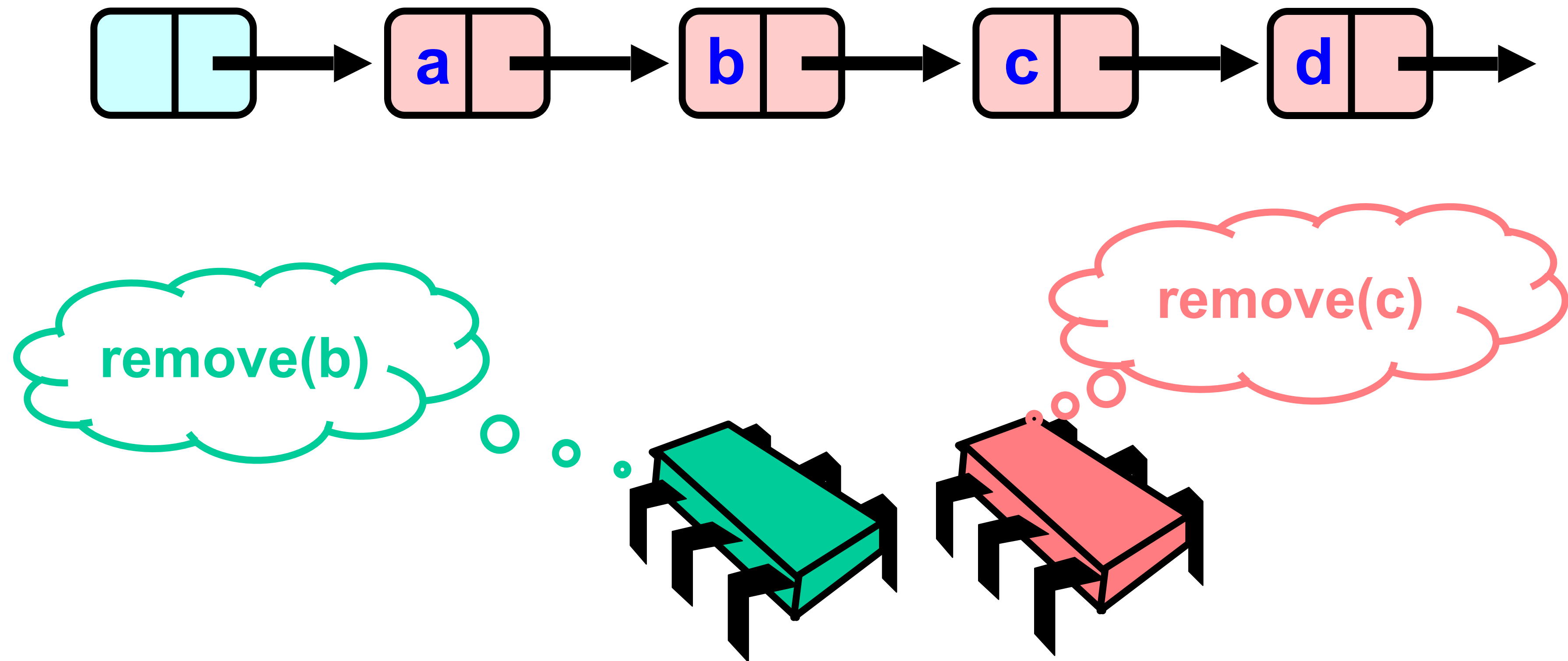


# Removing a Node

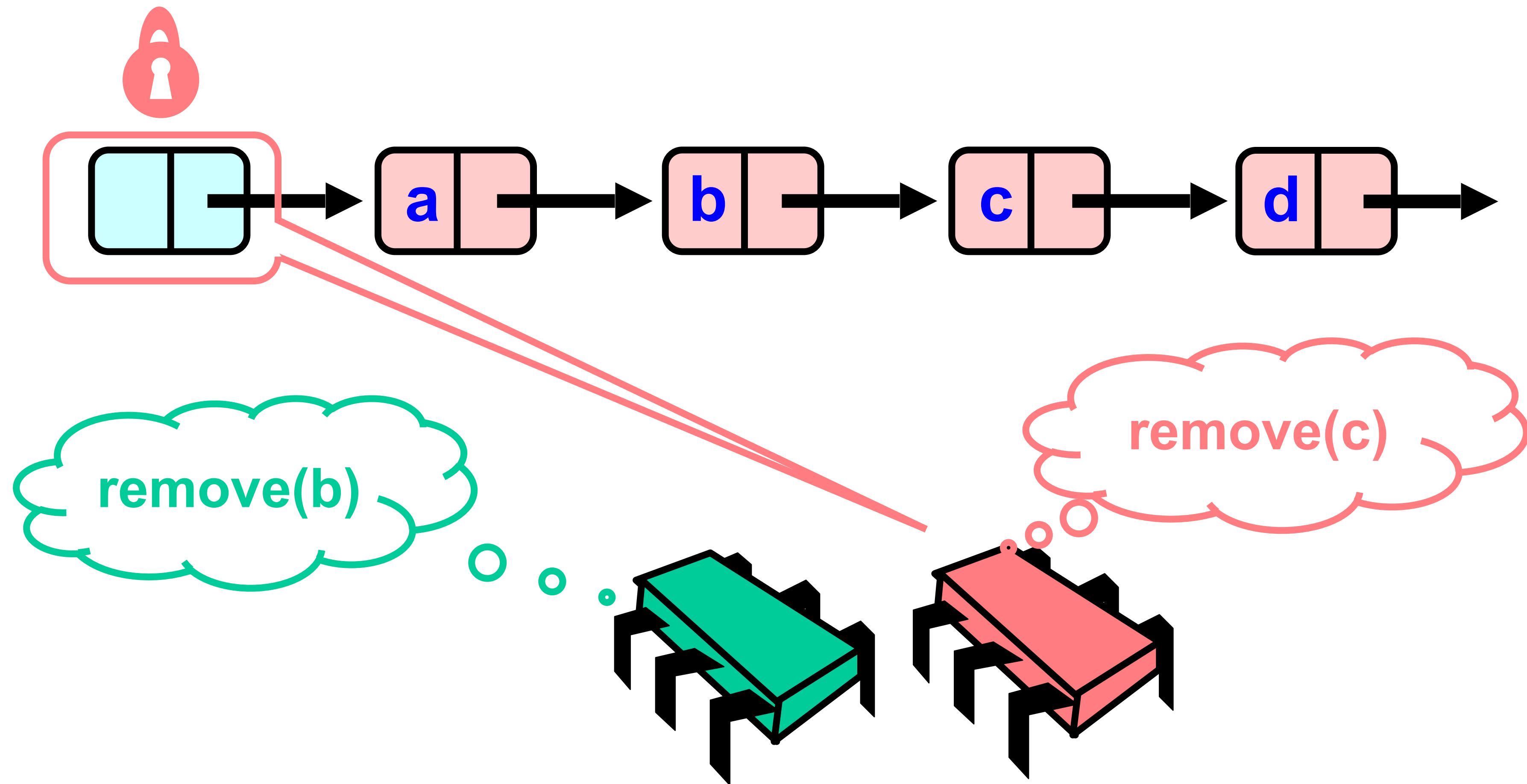


<A good place to pause>

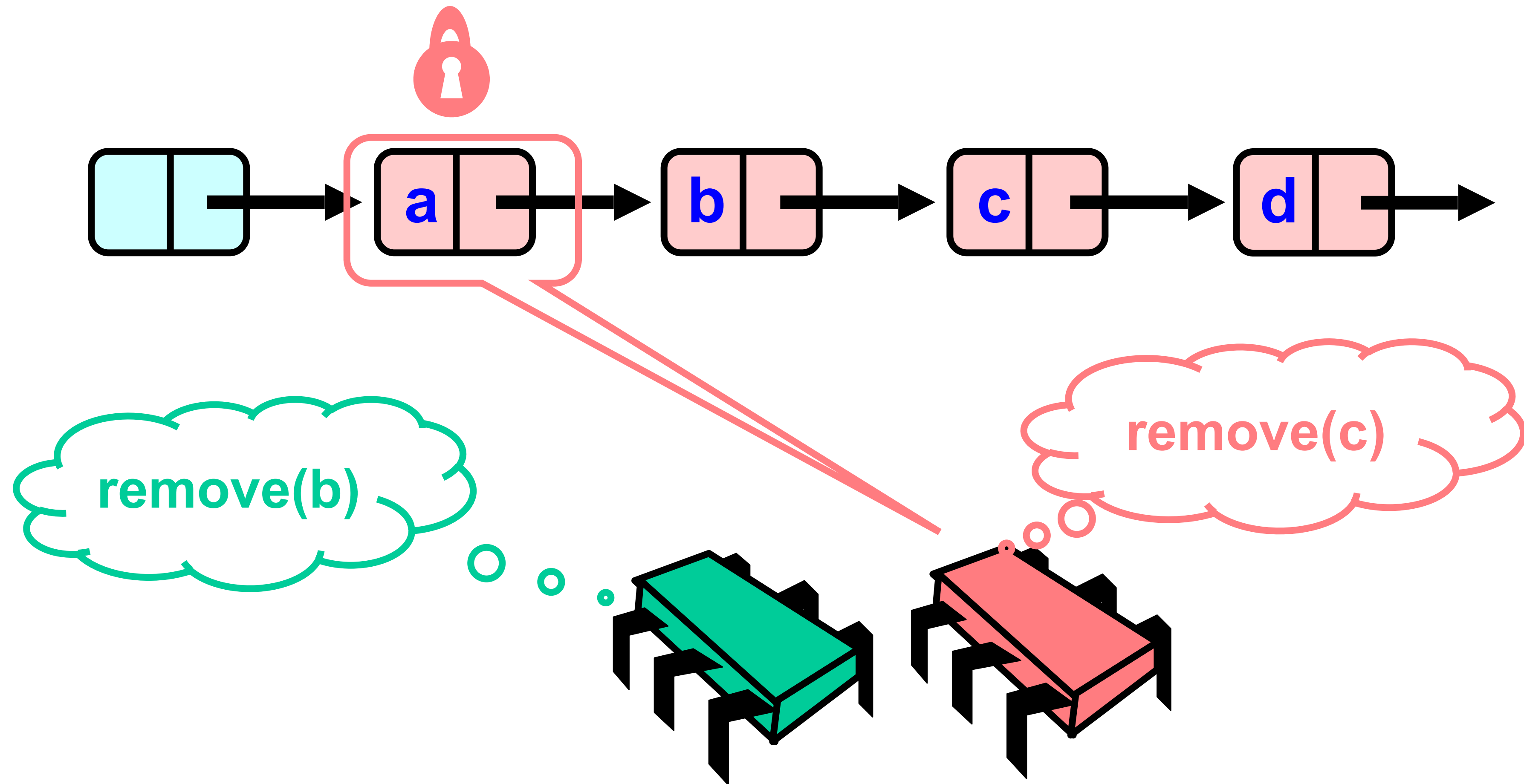
# Concurrent Removes



# Concurrent Removes

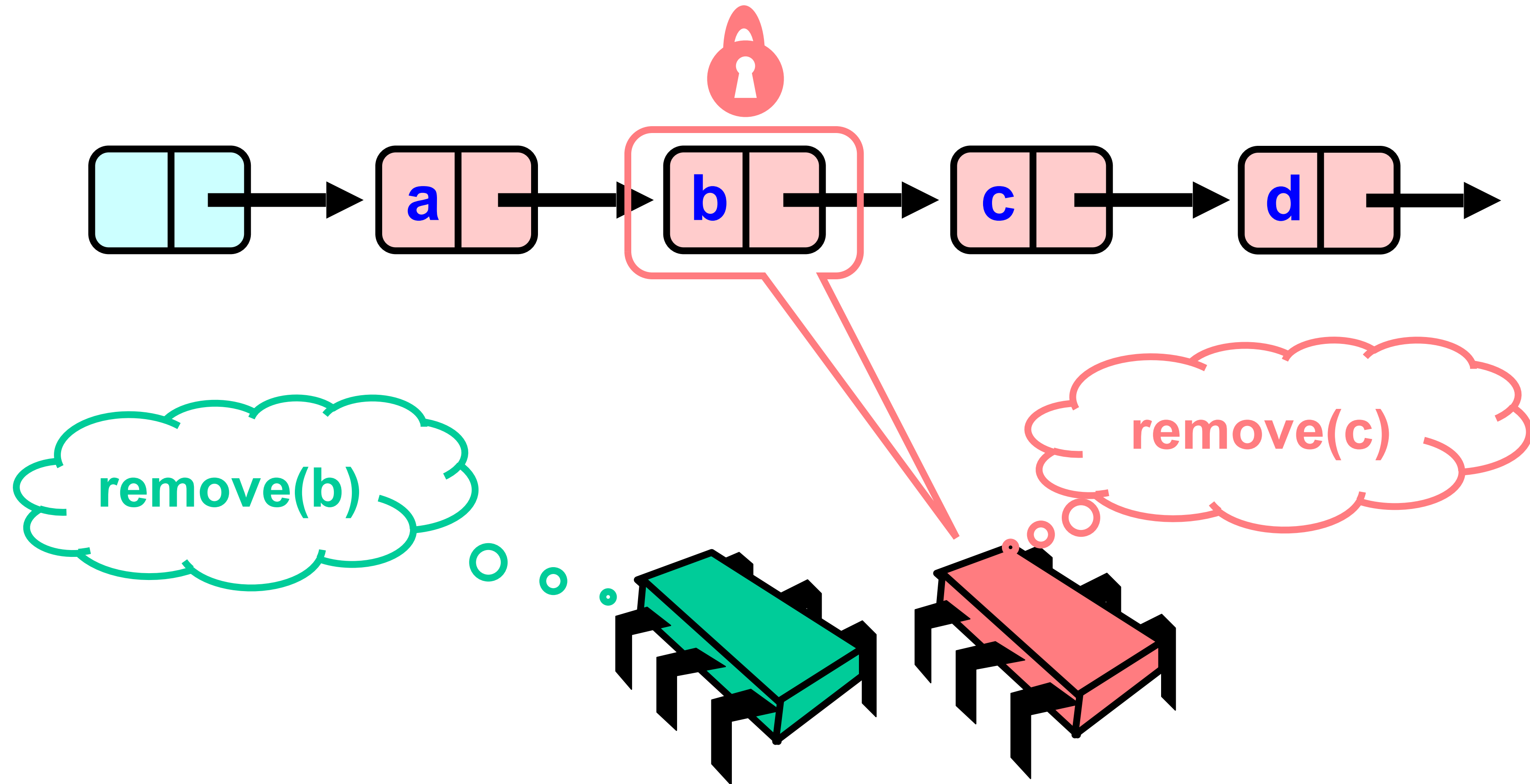


# Concurrent Removes

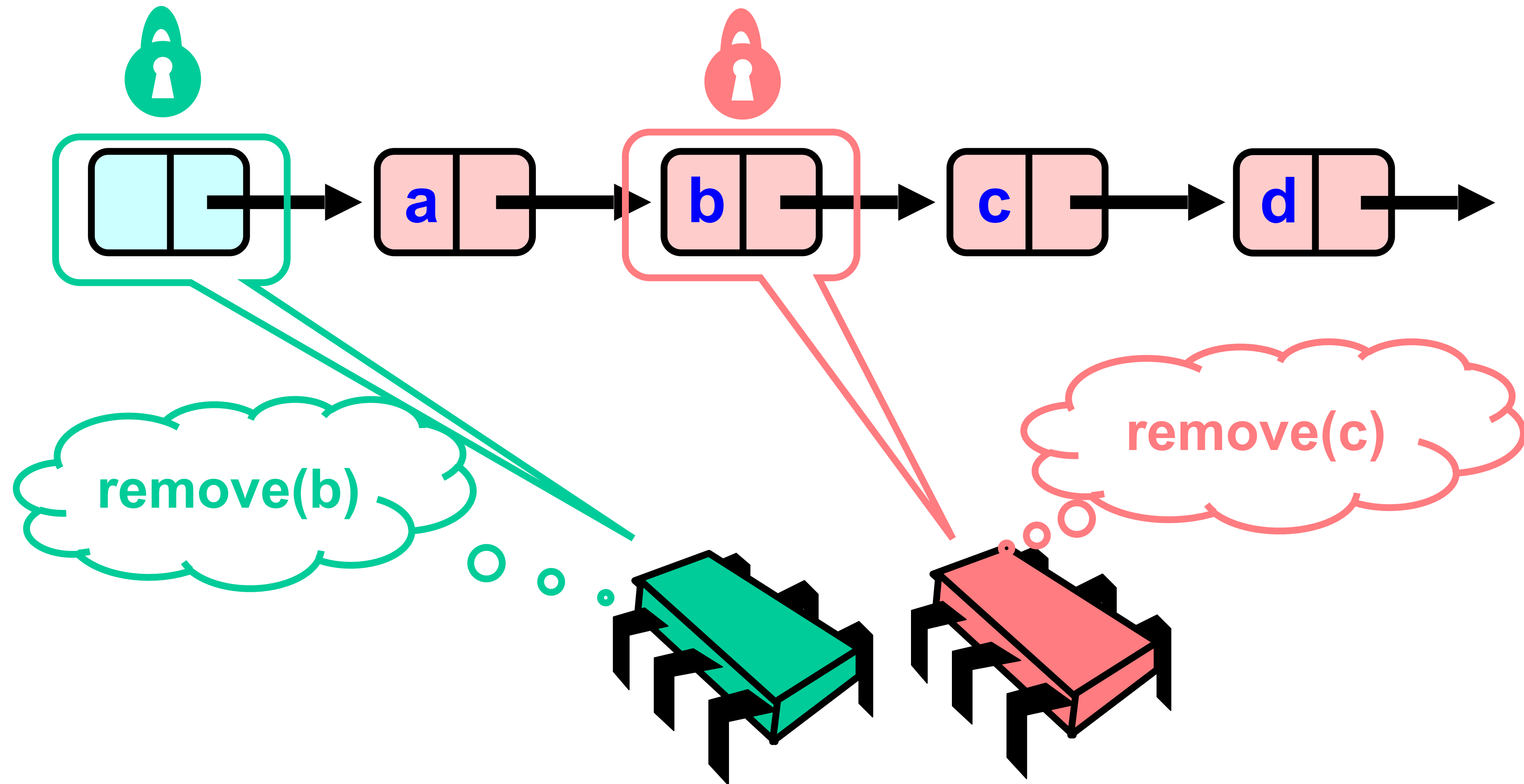




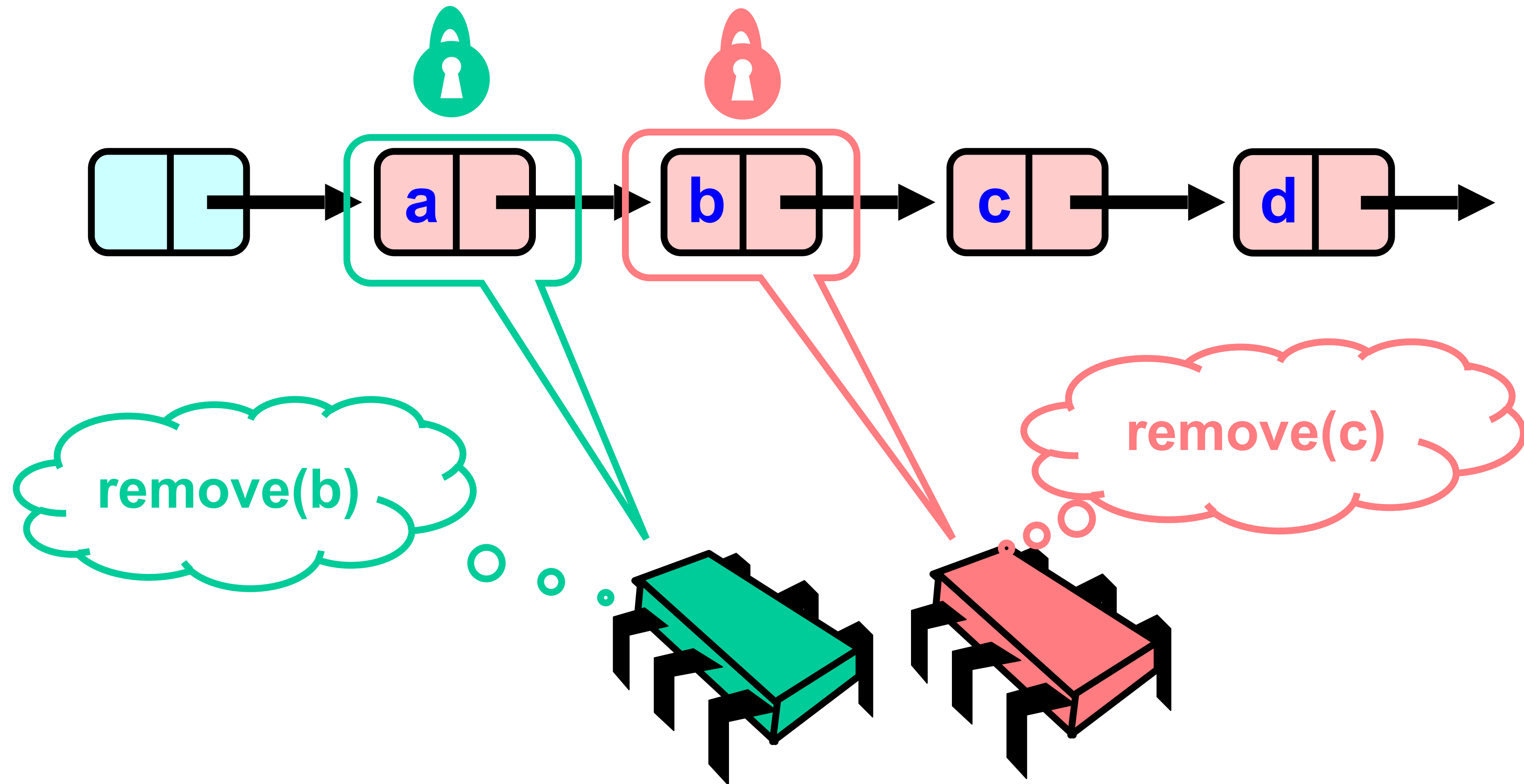
# Concurrent Removes



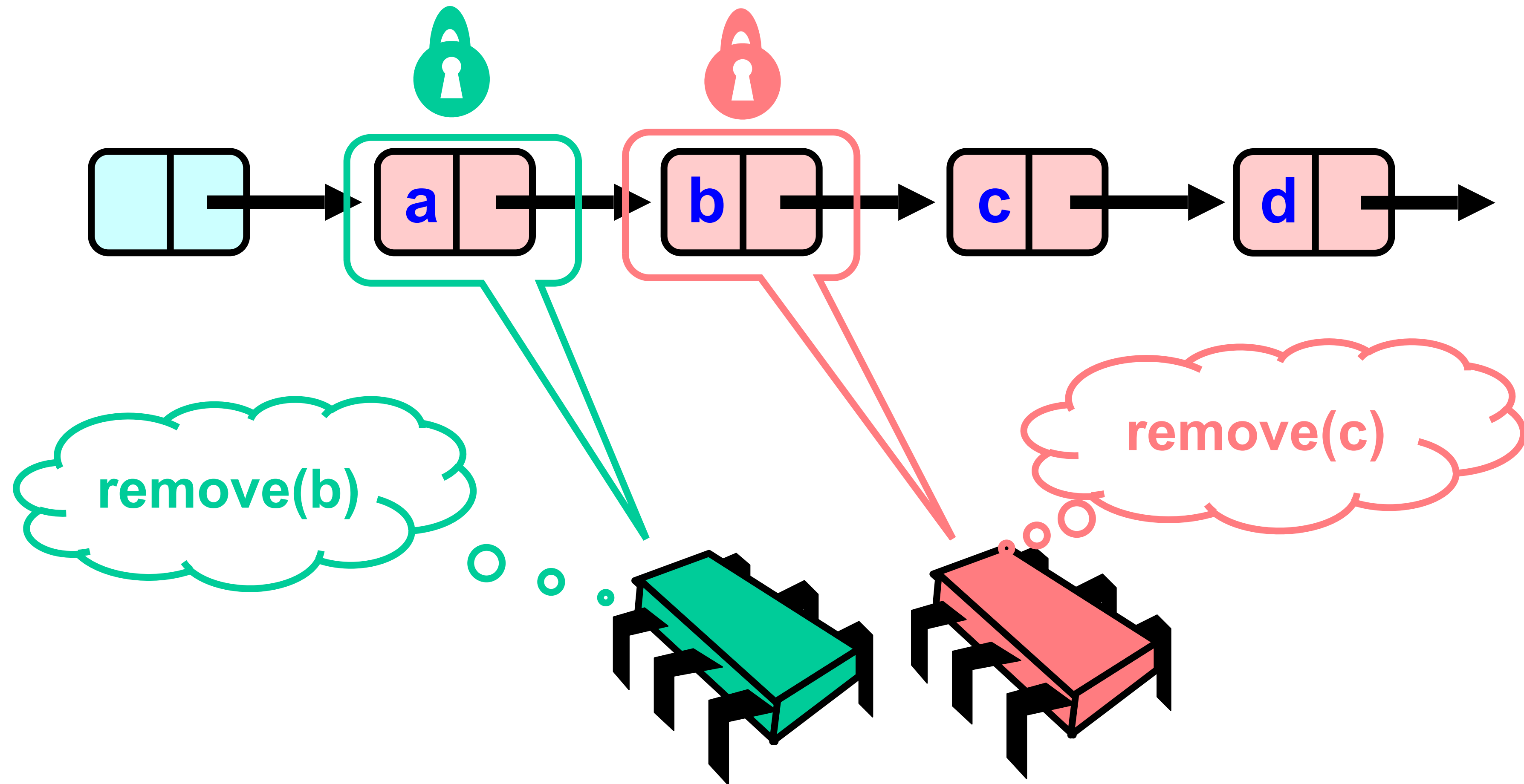
# Concurrent Removes



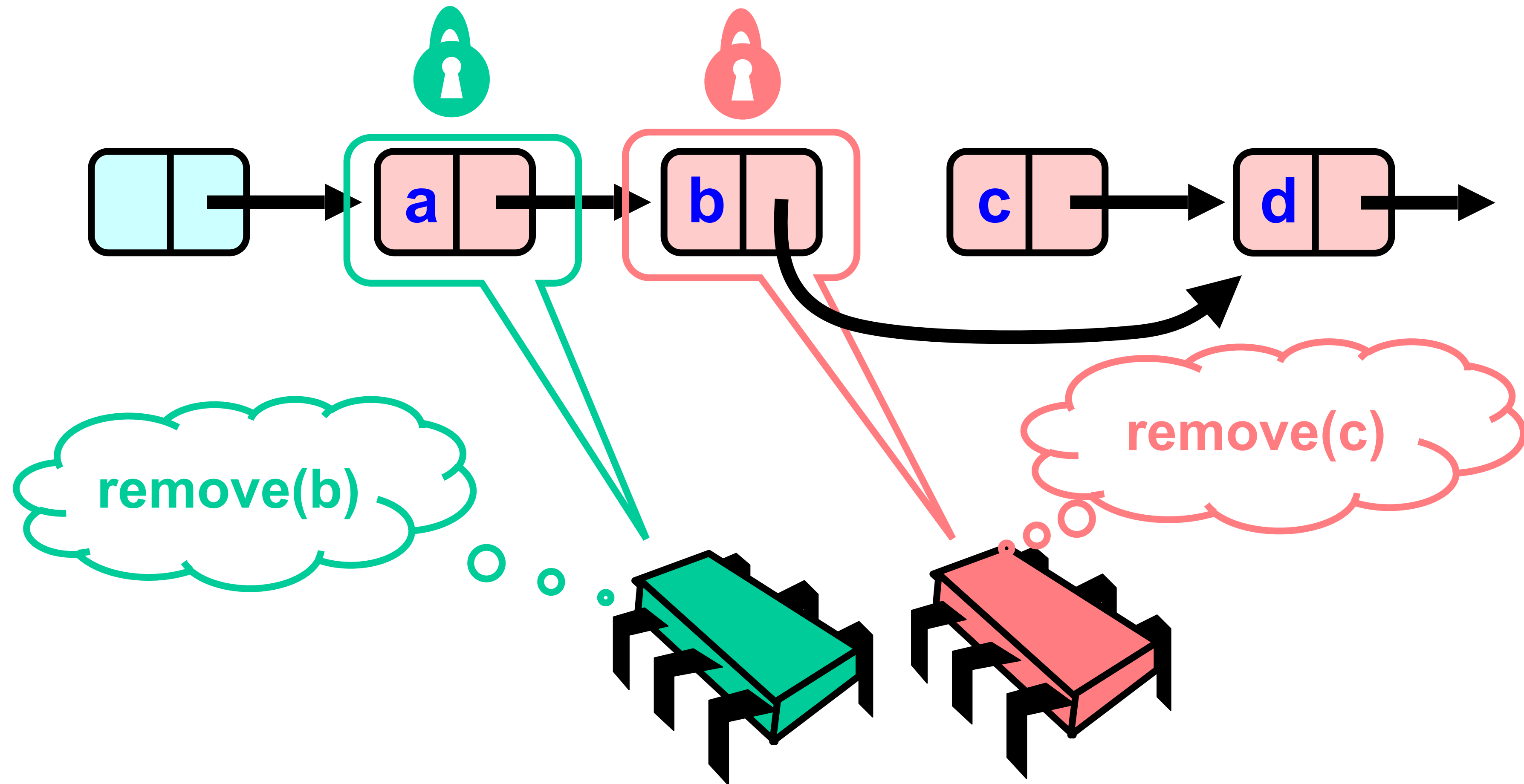
# Concurrent Removes



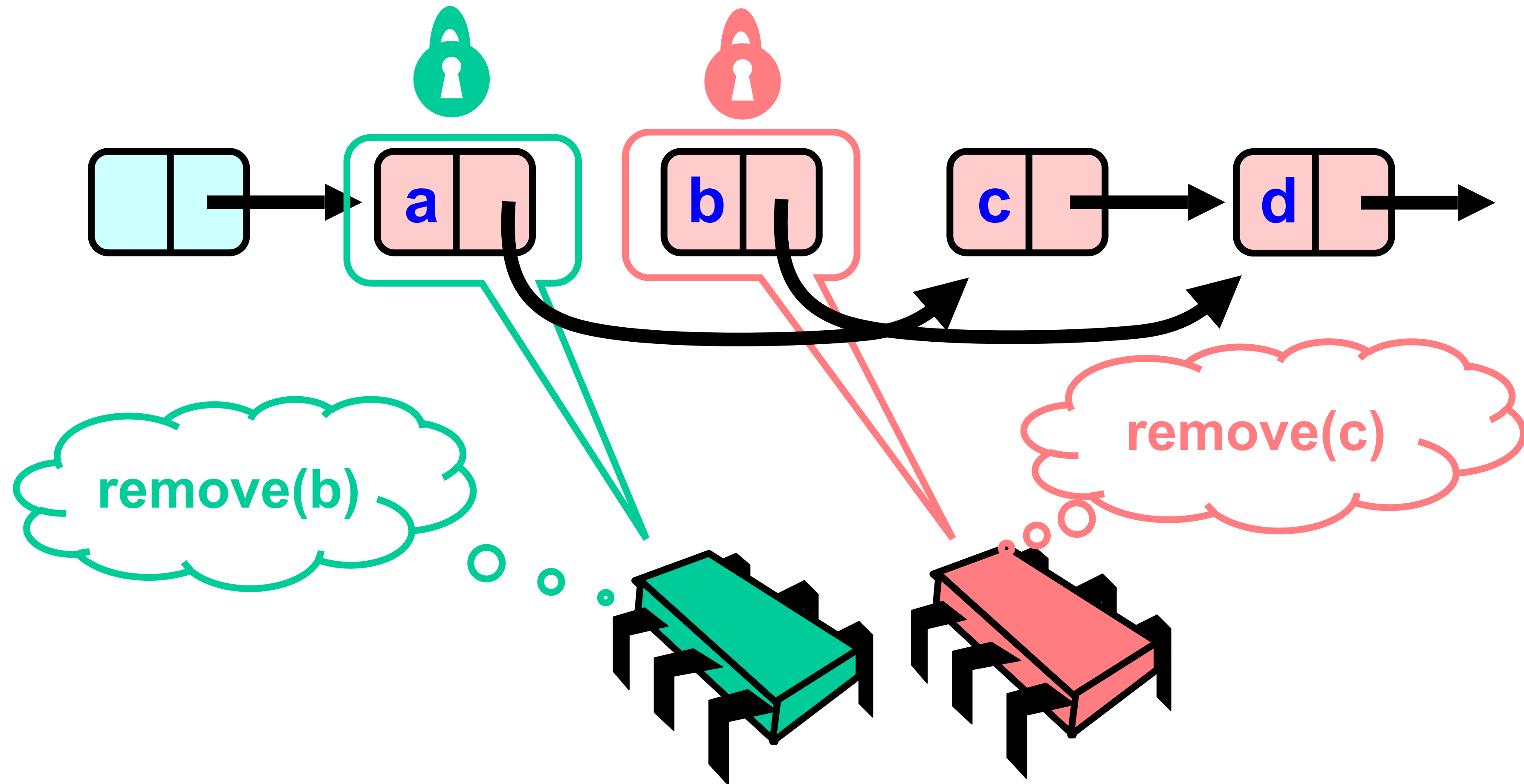
# Concurrent Removes



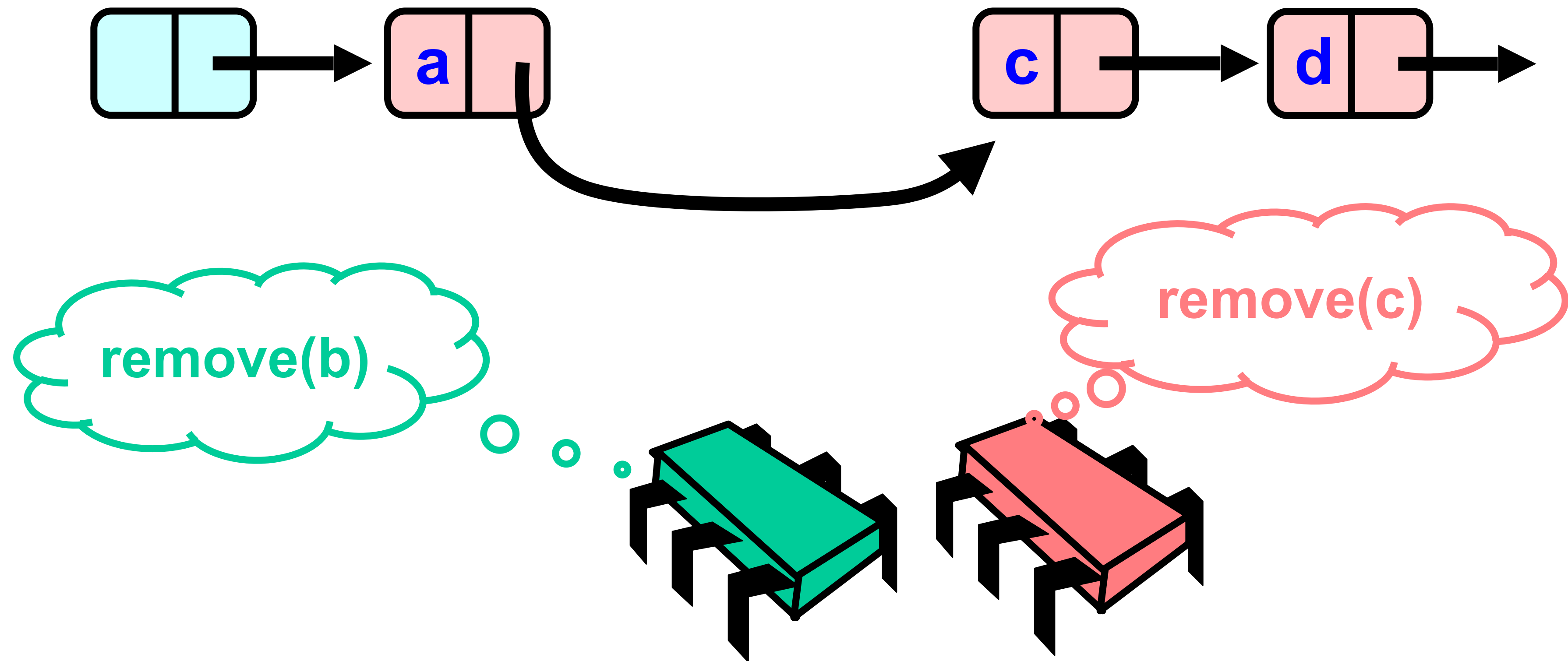
# Concurrent Removes



# Concurrent Removes

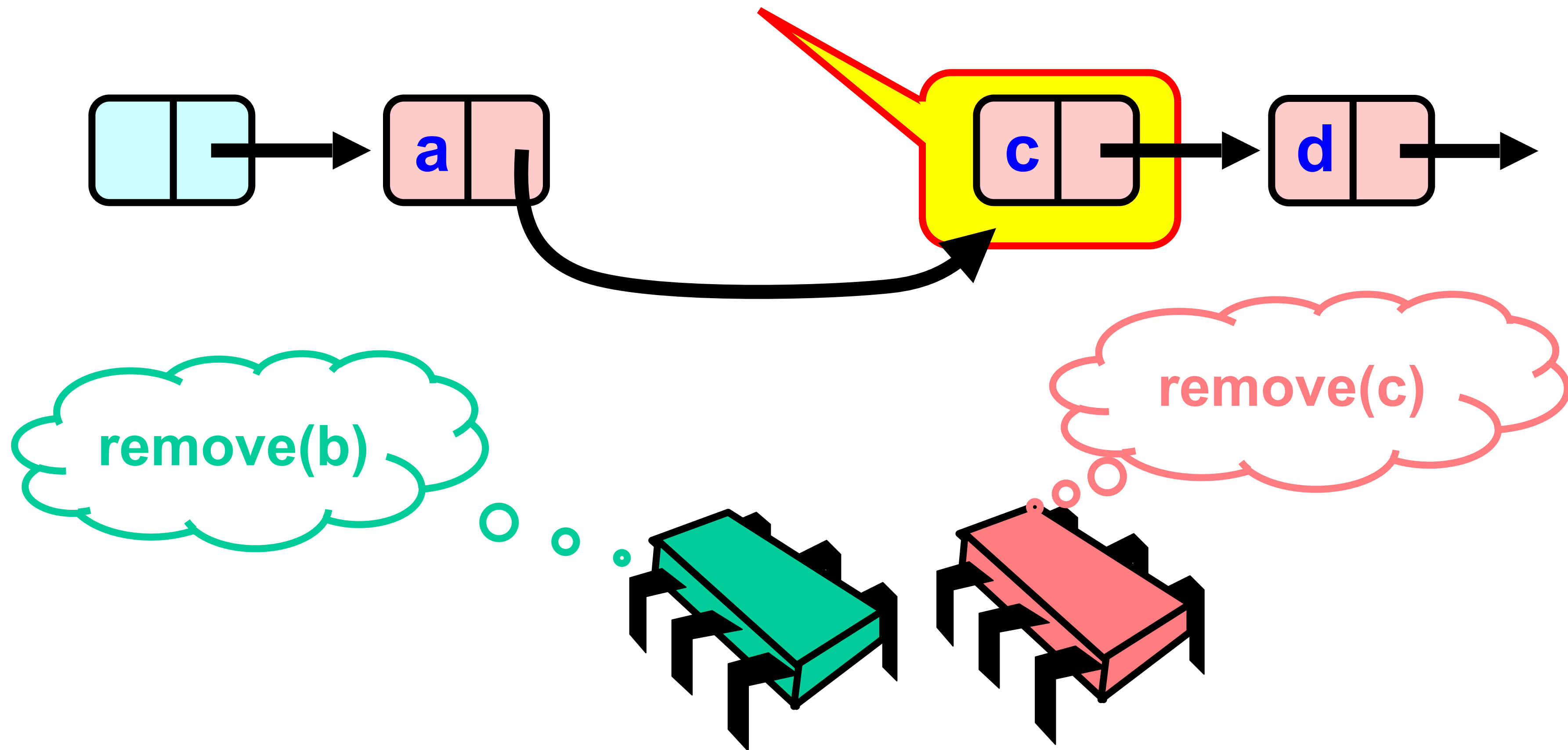


Uh, Oh



# Uh, Oh

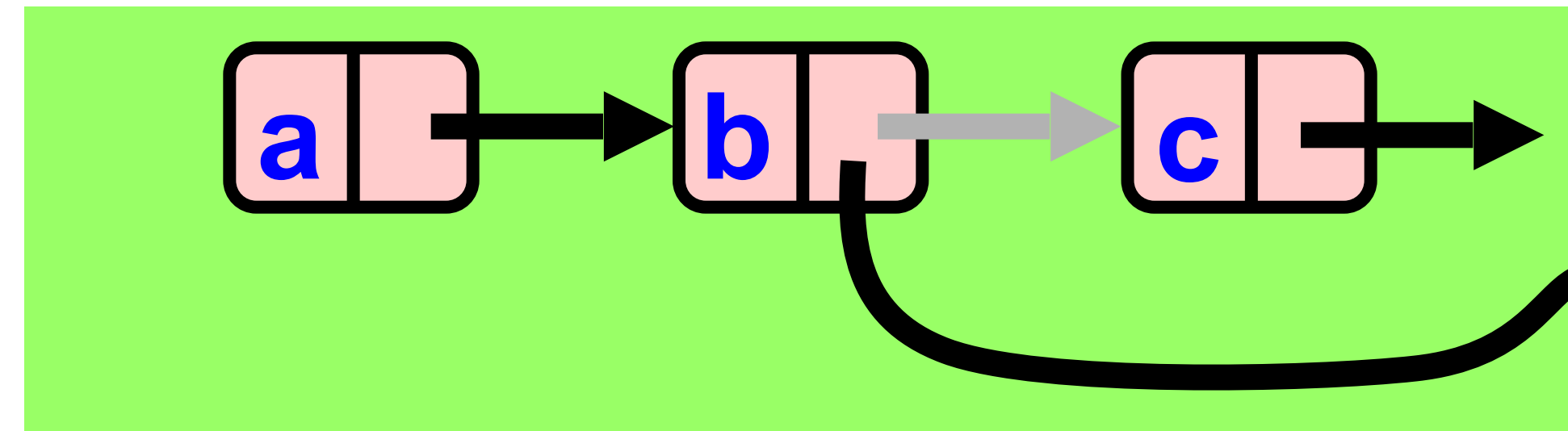
**Bad news, c not removed**



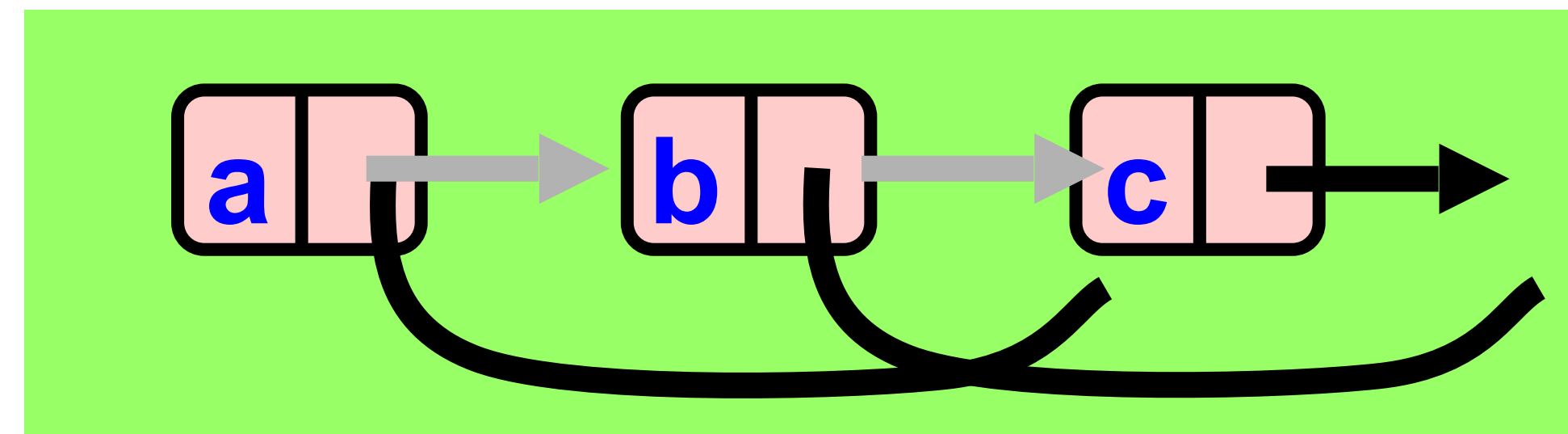


# Problem

- To delete node **c**
  - Swing node **b**'s next field to **d**



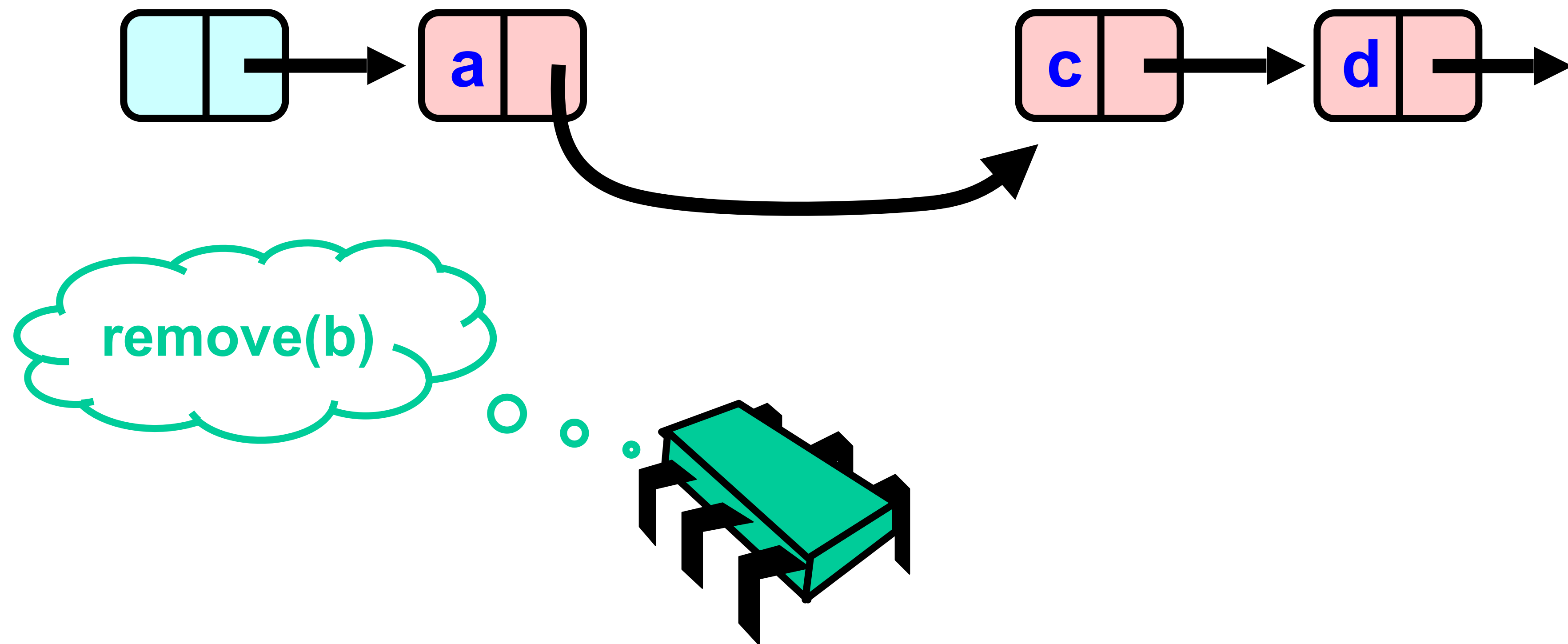
- Problem is,
  - Someone deleting **b** concurrently could direct a pointer to **c**



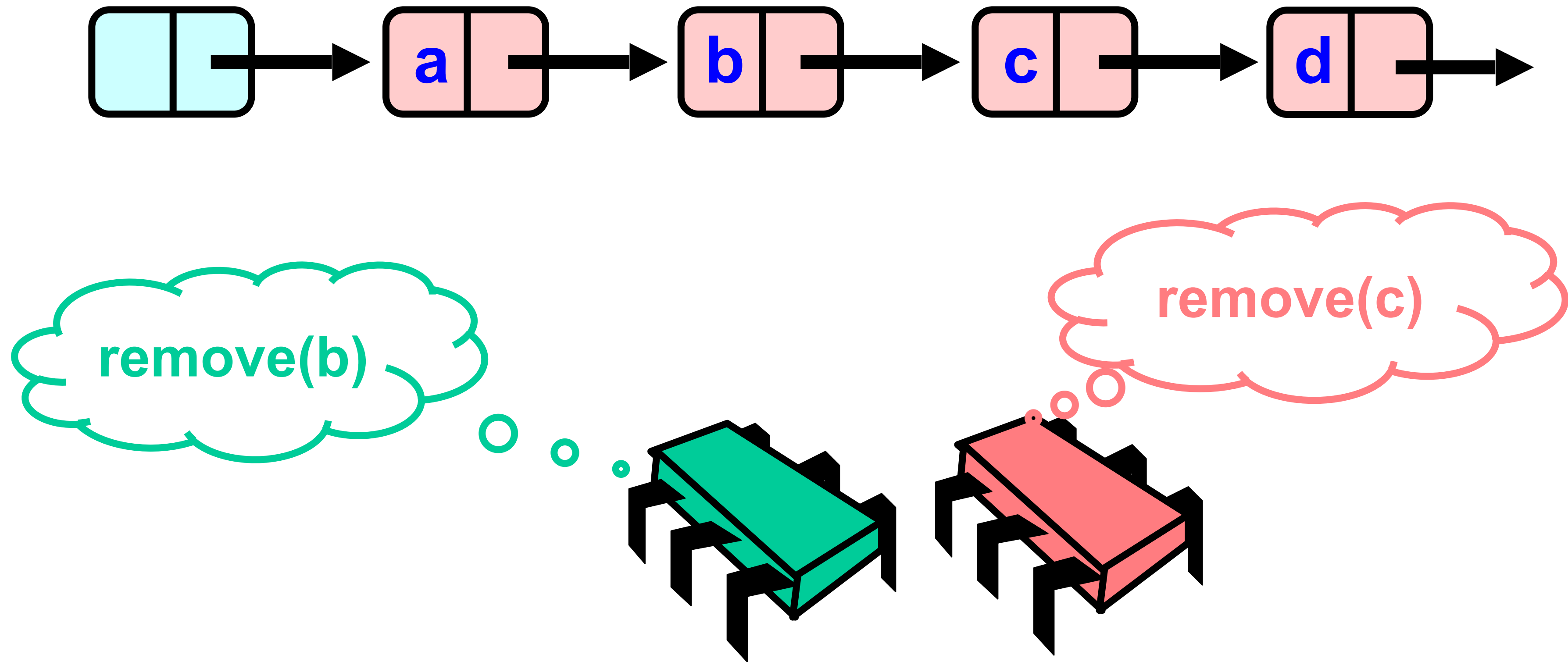
# Hand-over-Hand Locking: Insight

- 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

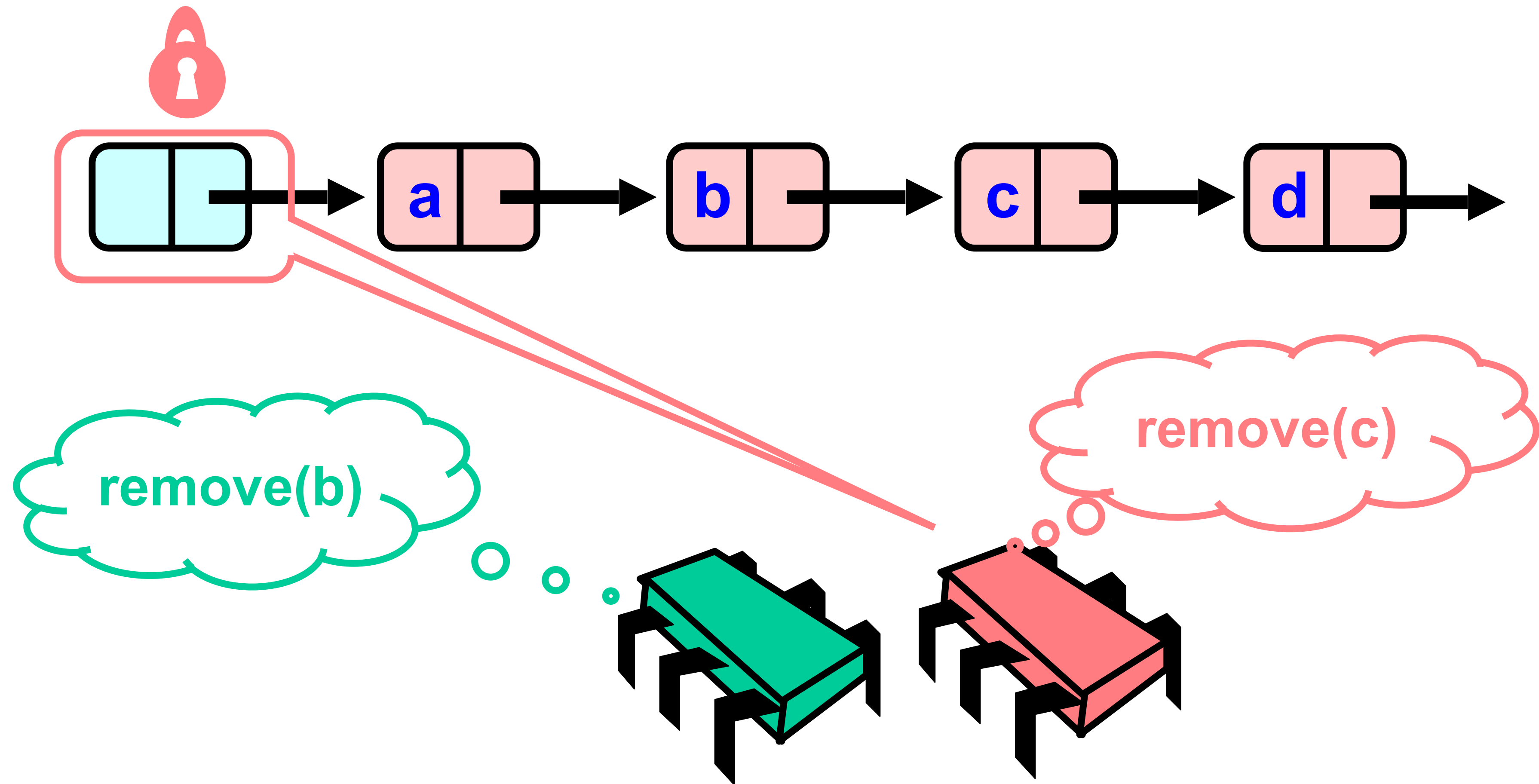
# Hand-Over-Hand Again



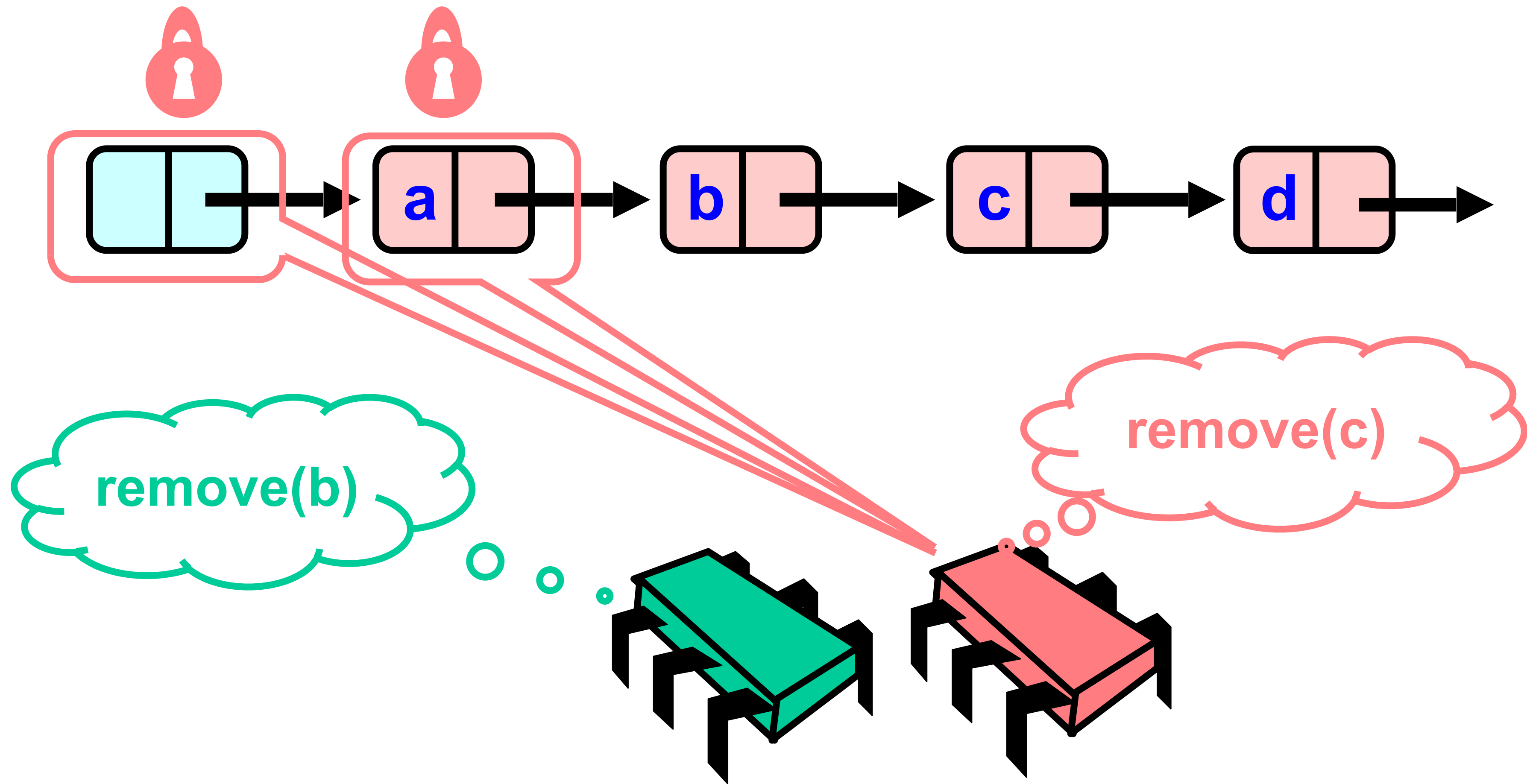
# Removing a Node



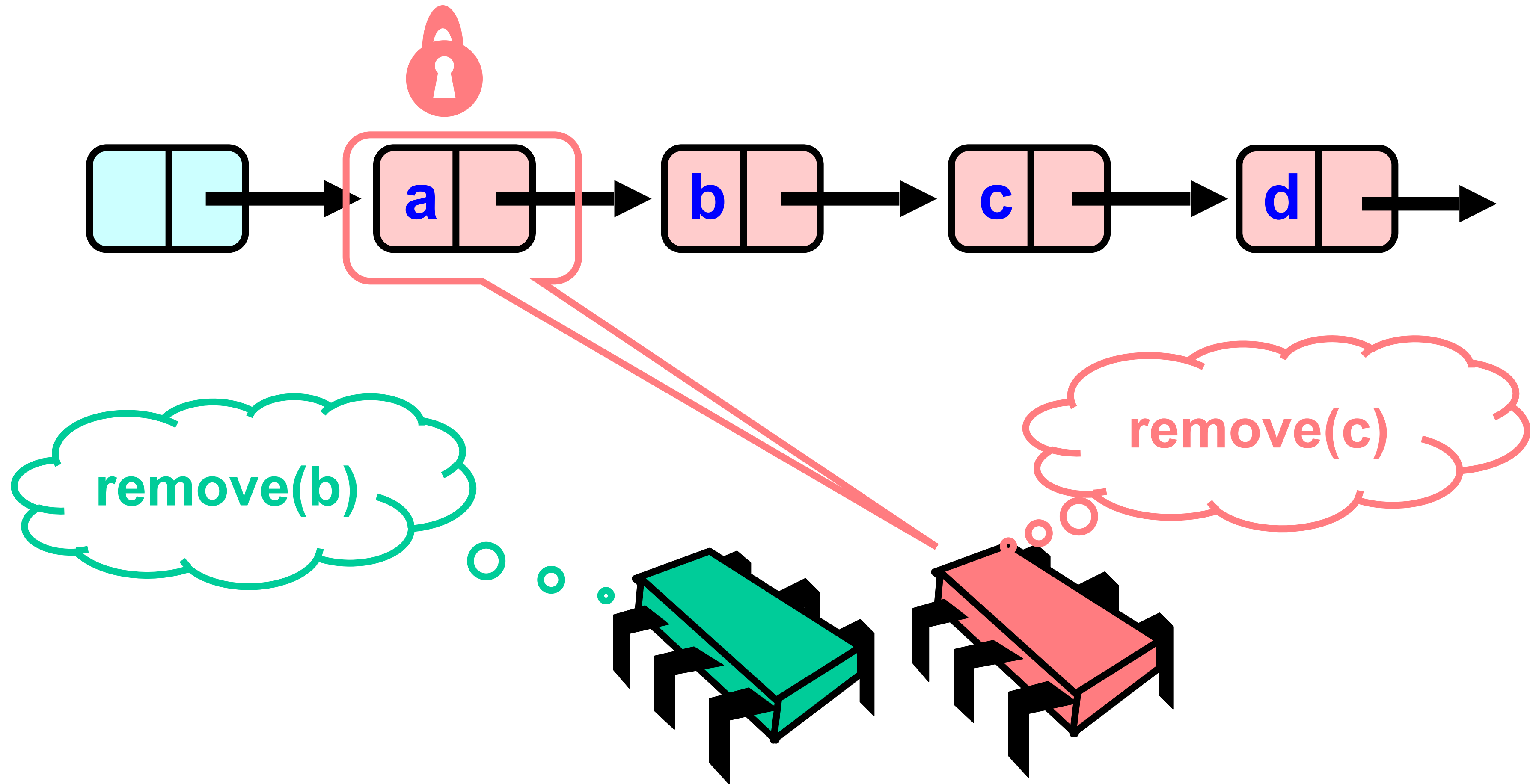
# Removing a Node



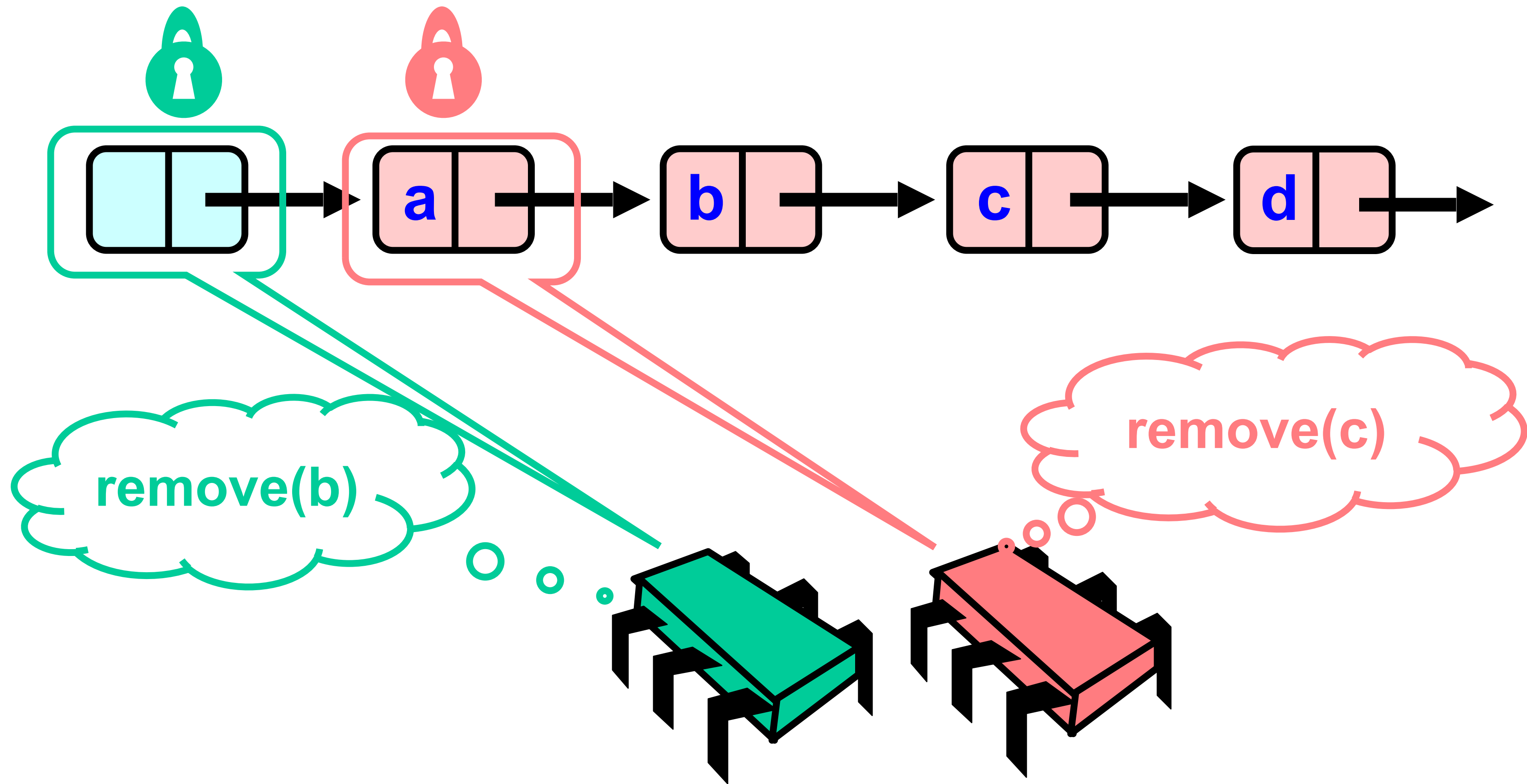
# Removing a Node



# Removing a Node

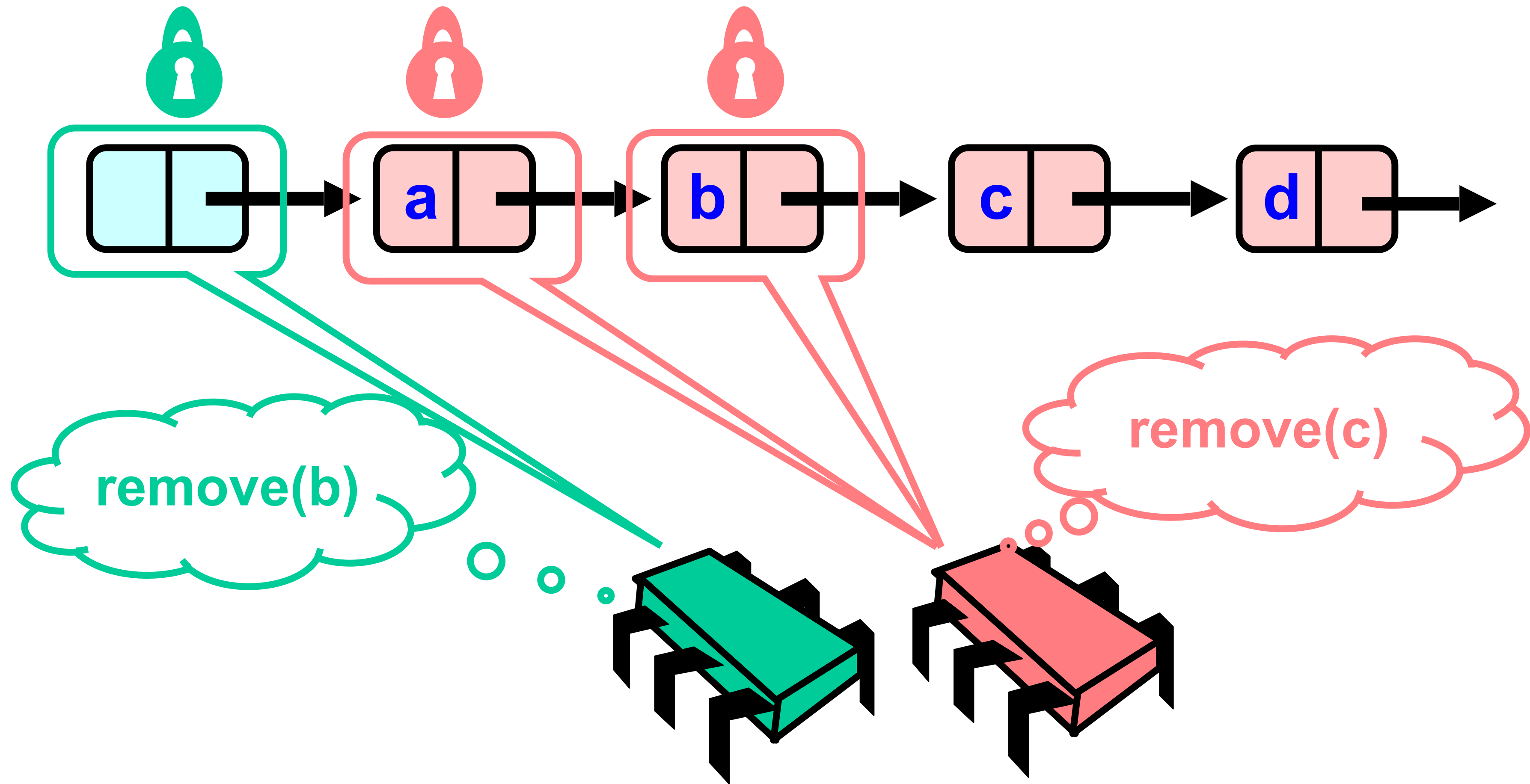


# Removing a Node

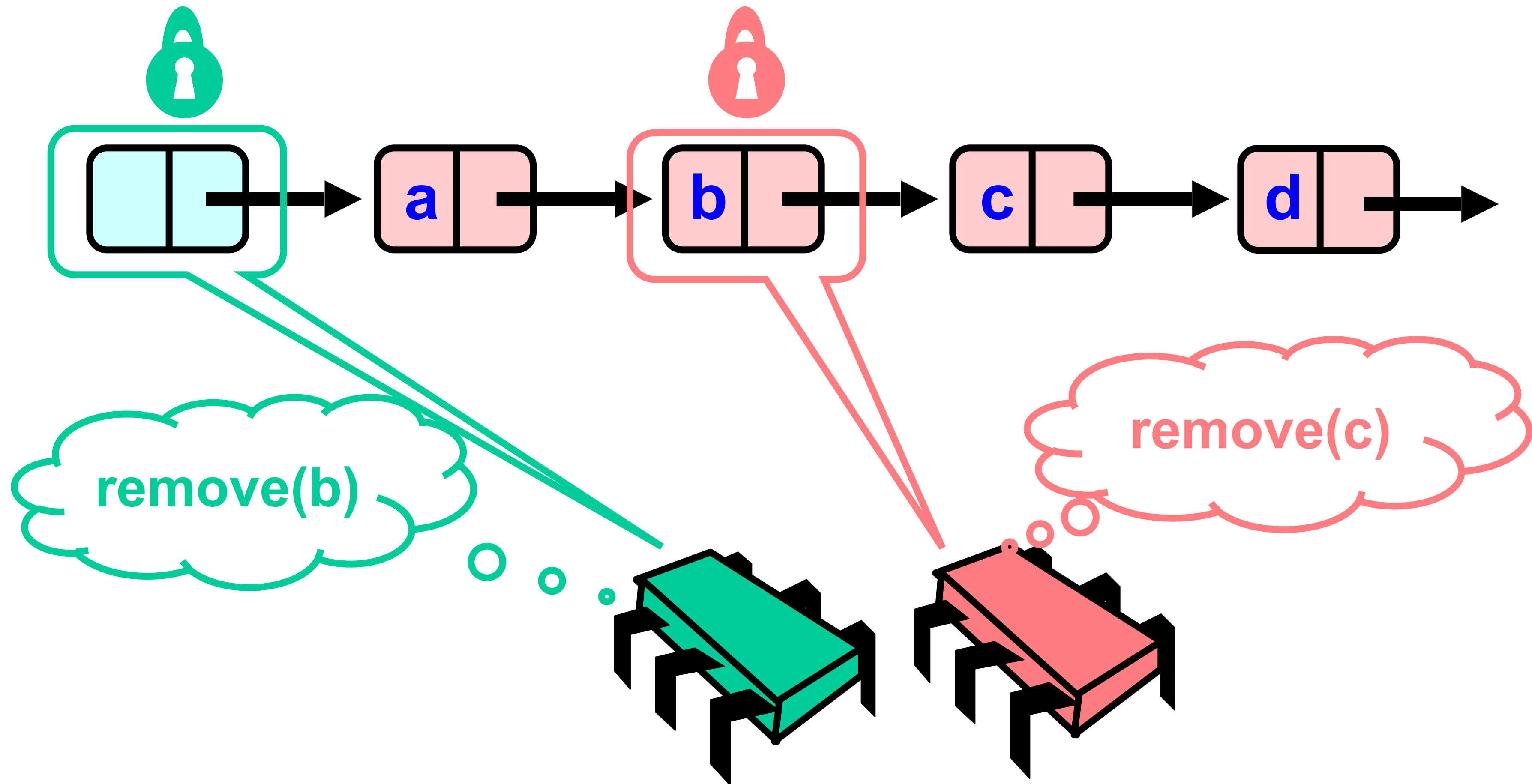




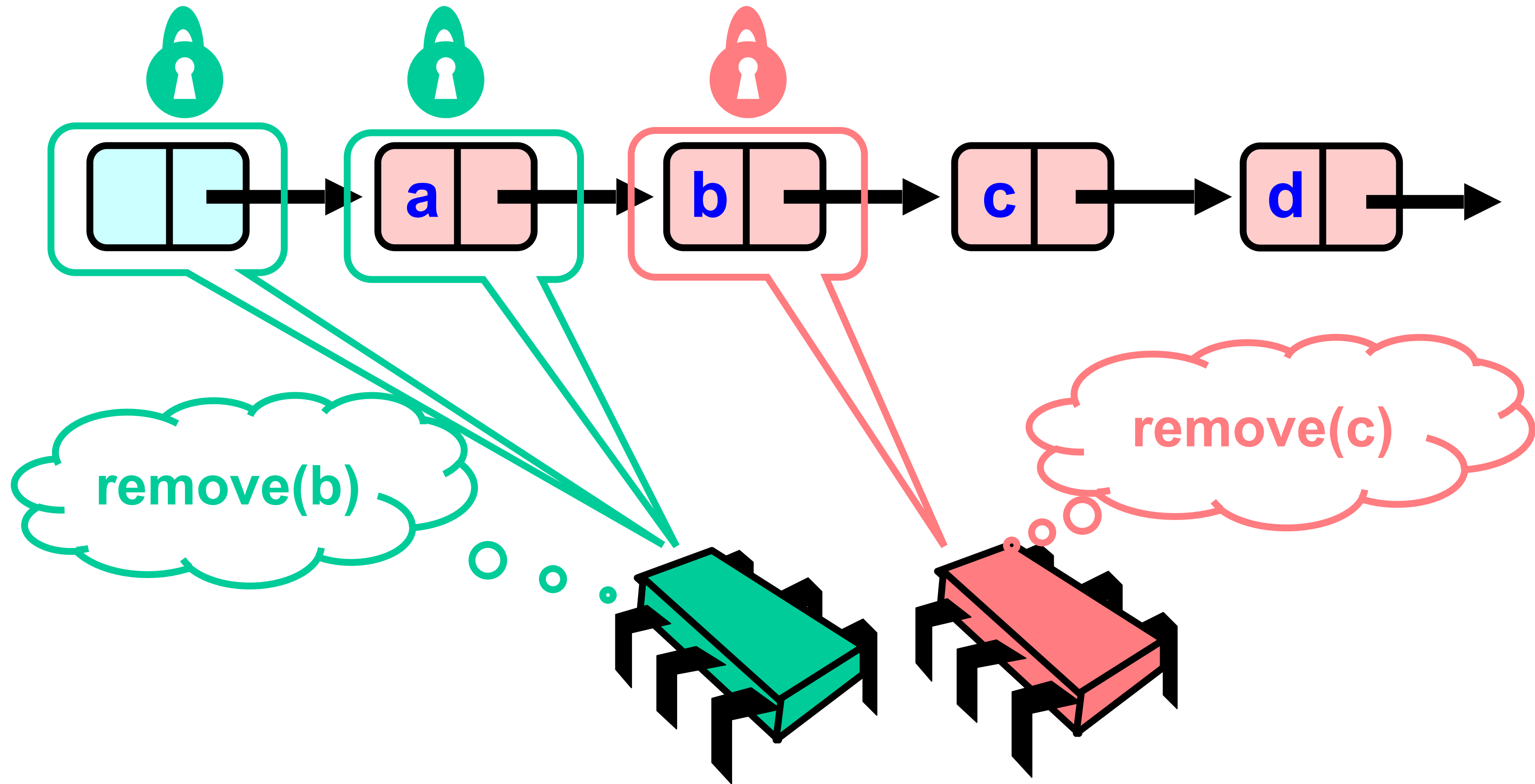
# Removing a Node



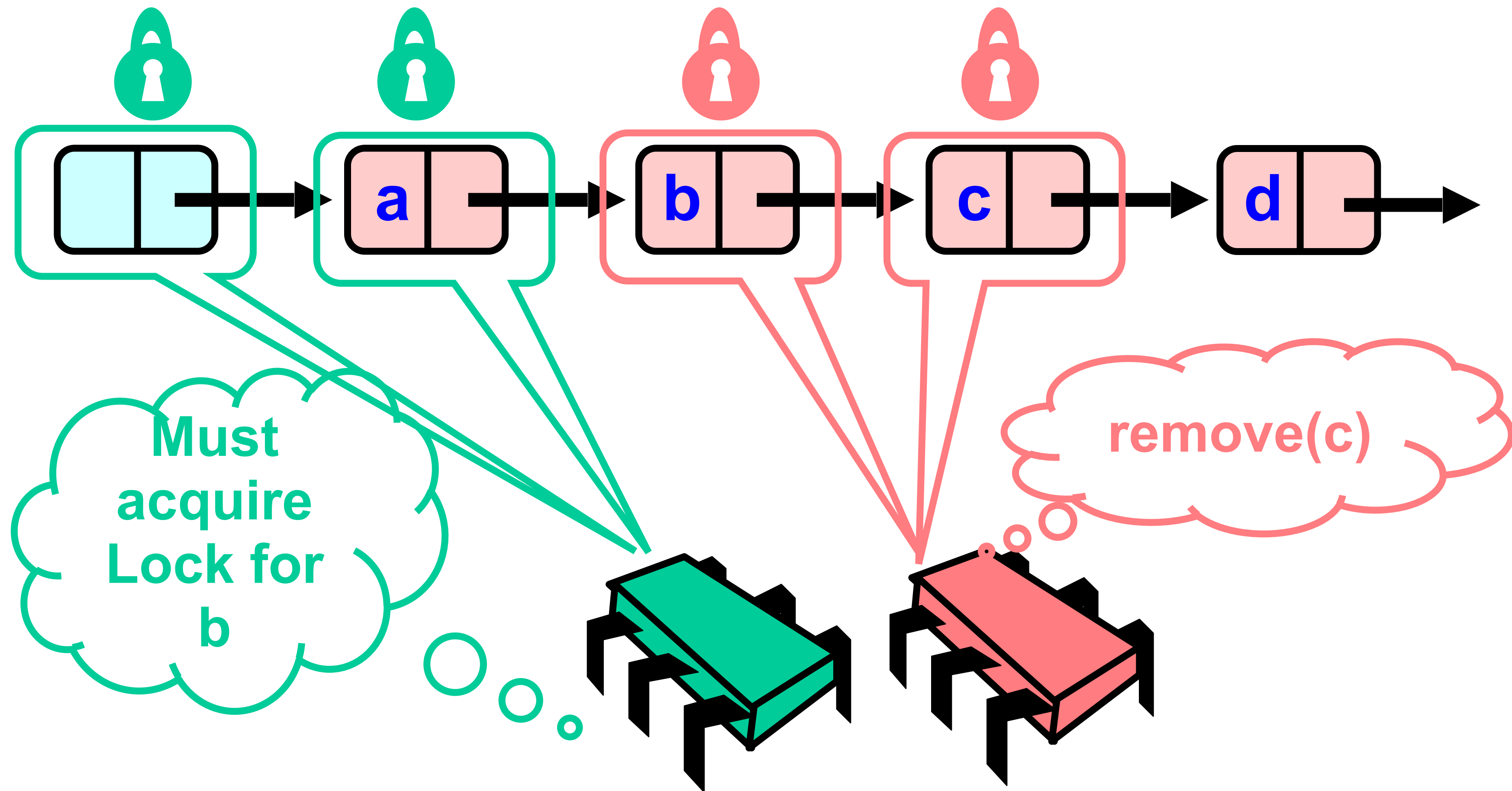
# Removing a Node



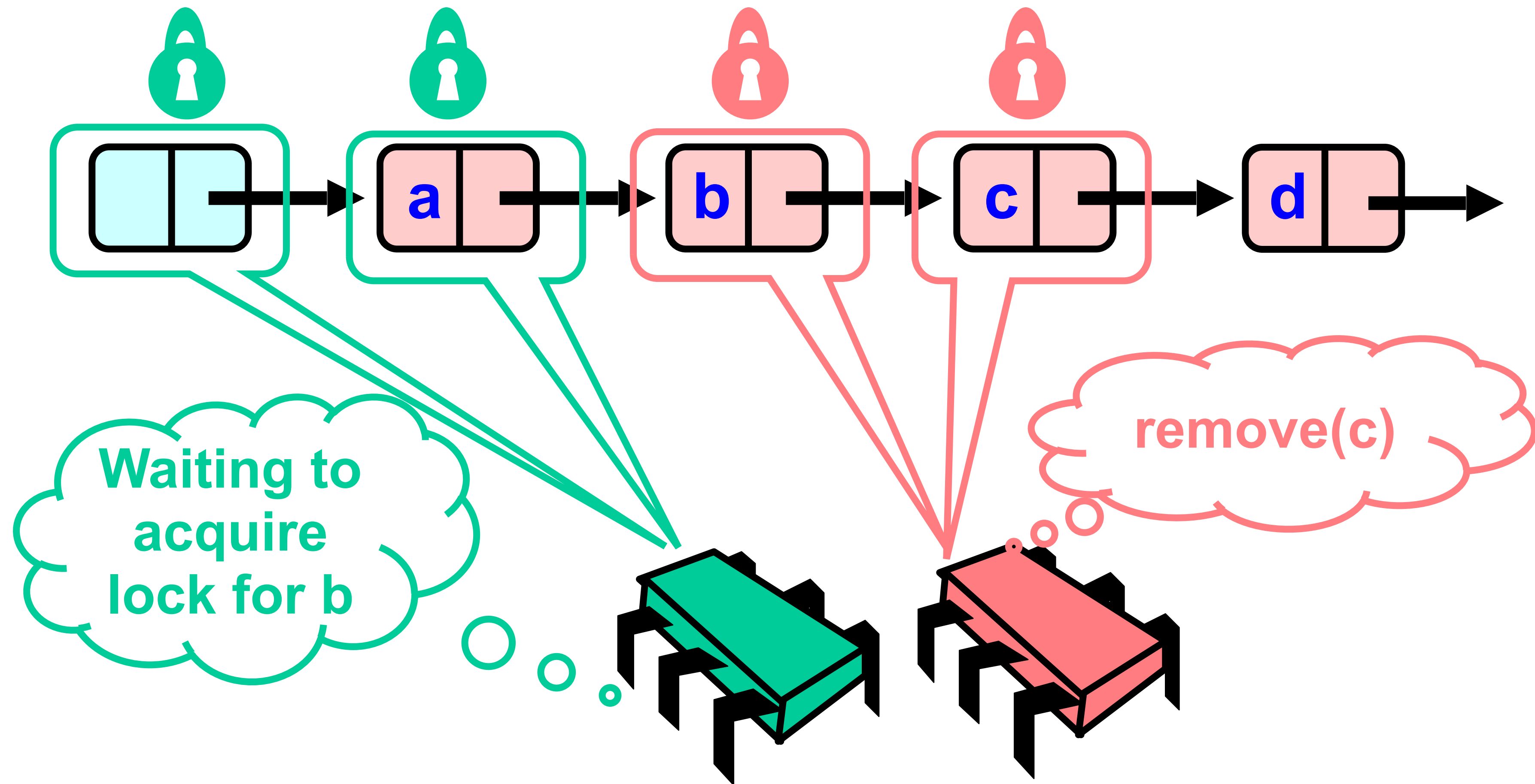
# Removing a Node



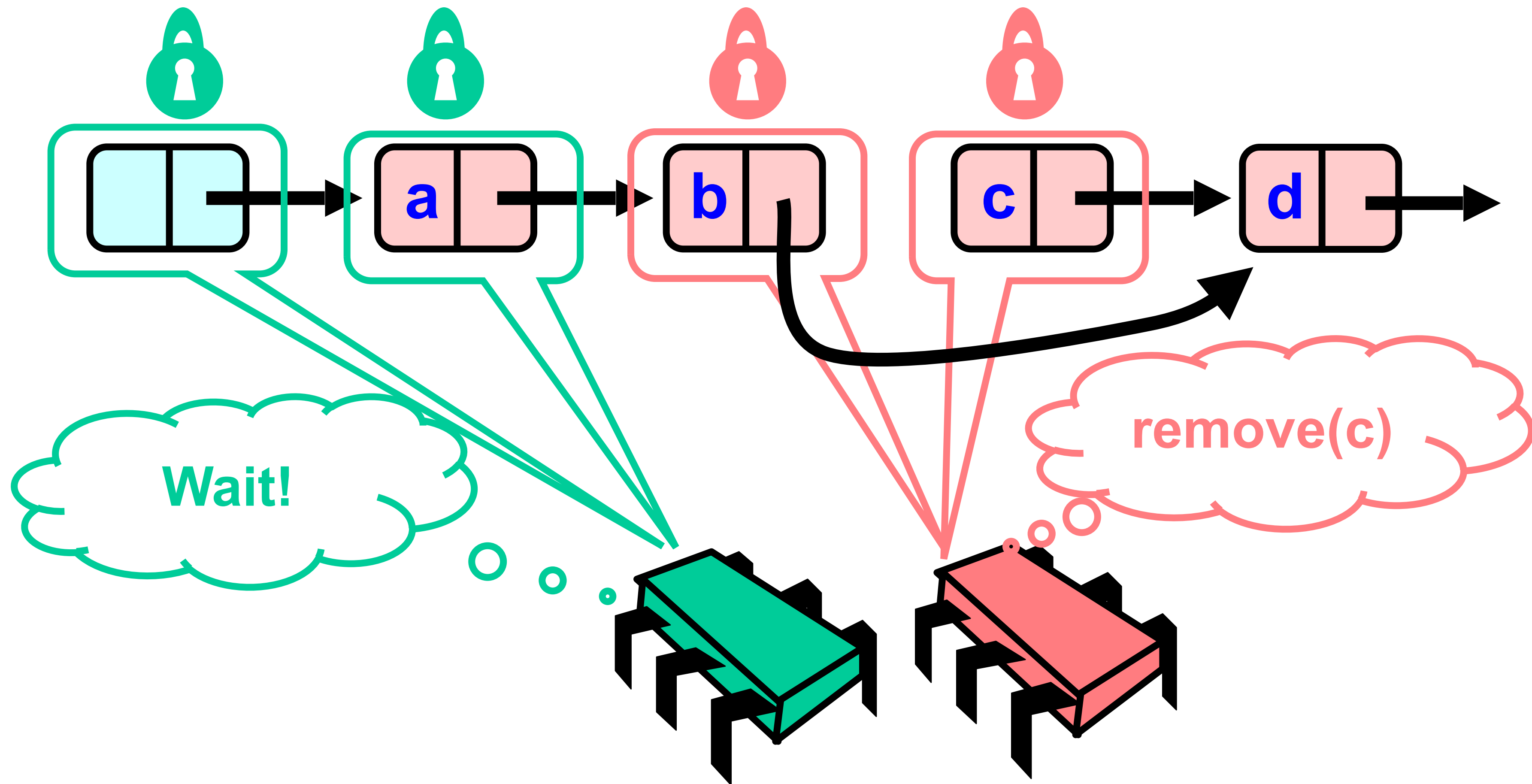
# Removing a Node



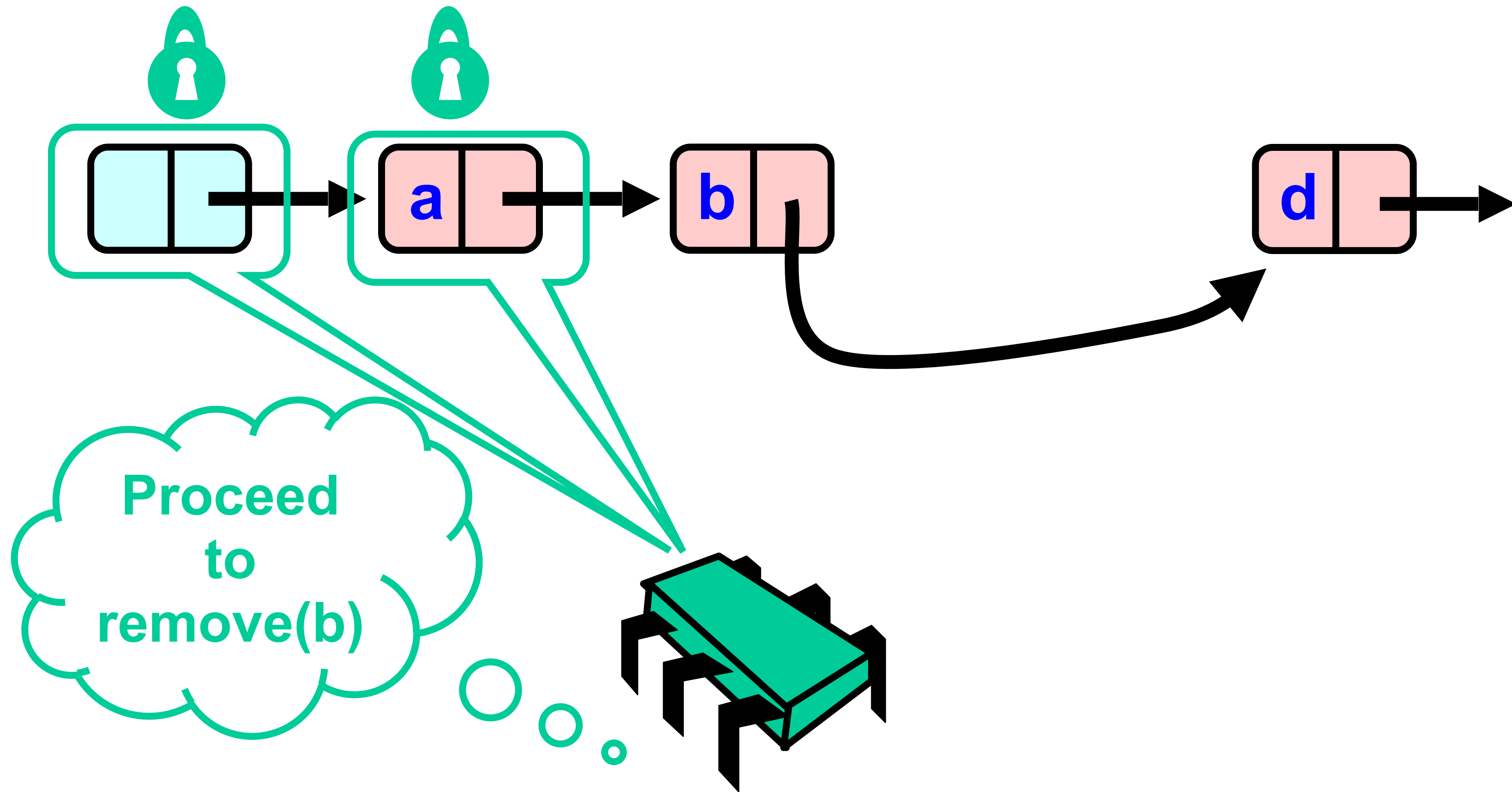
# Removing a Node



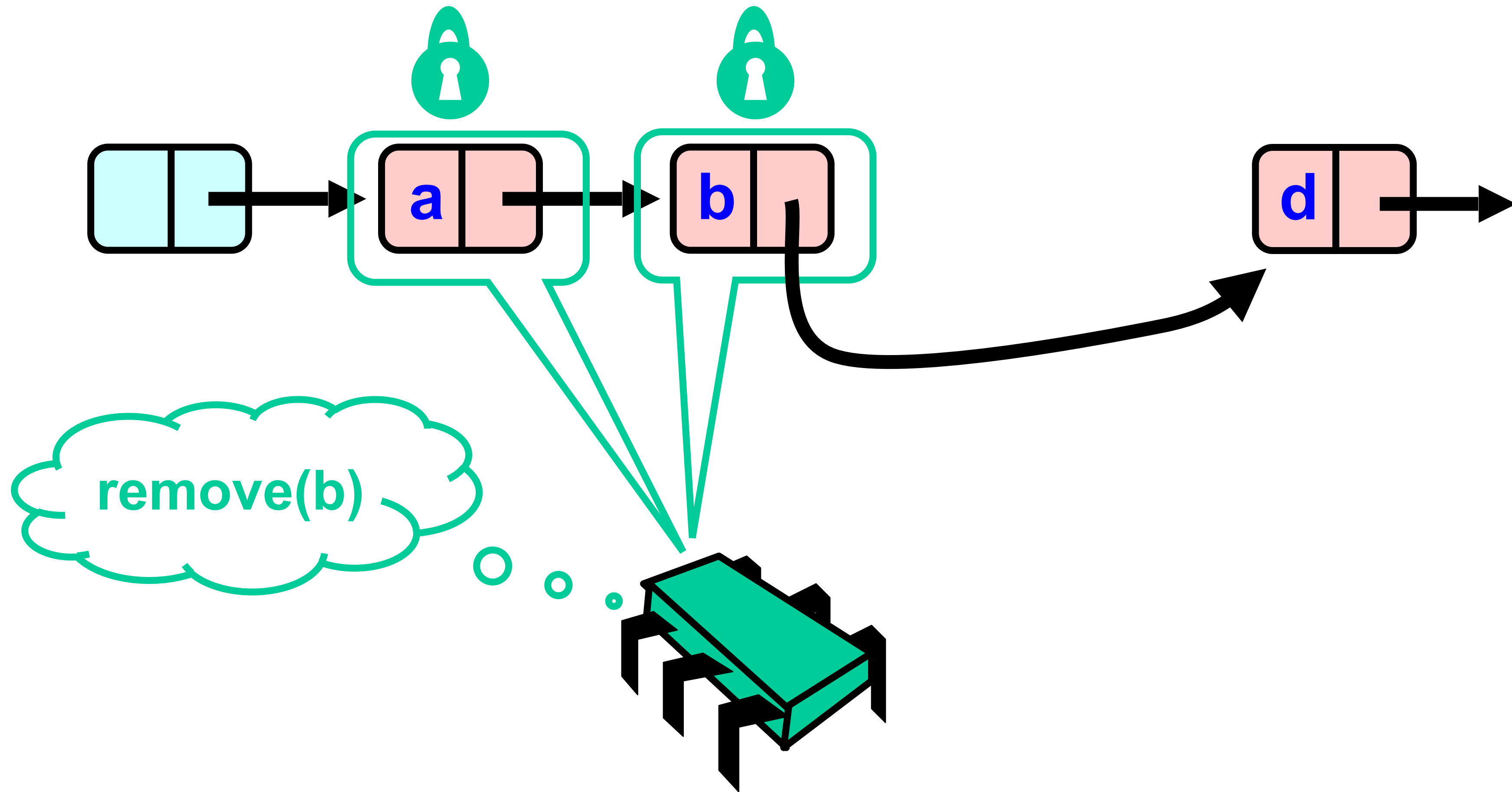
# Removing a Node



# Removing a Node

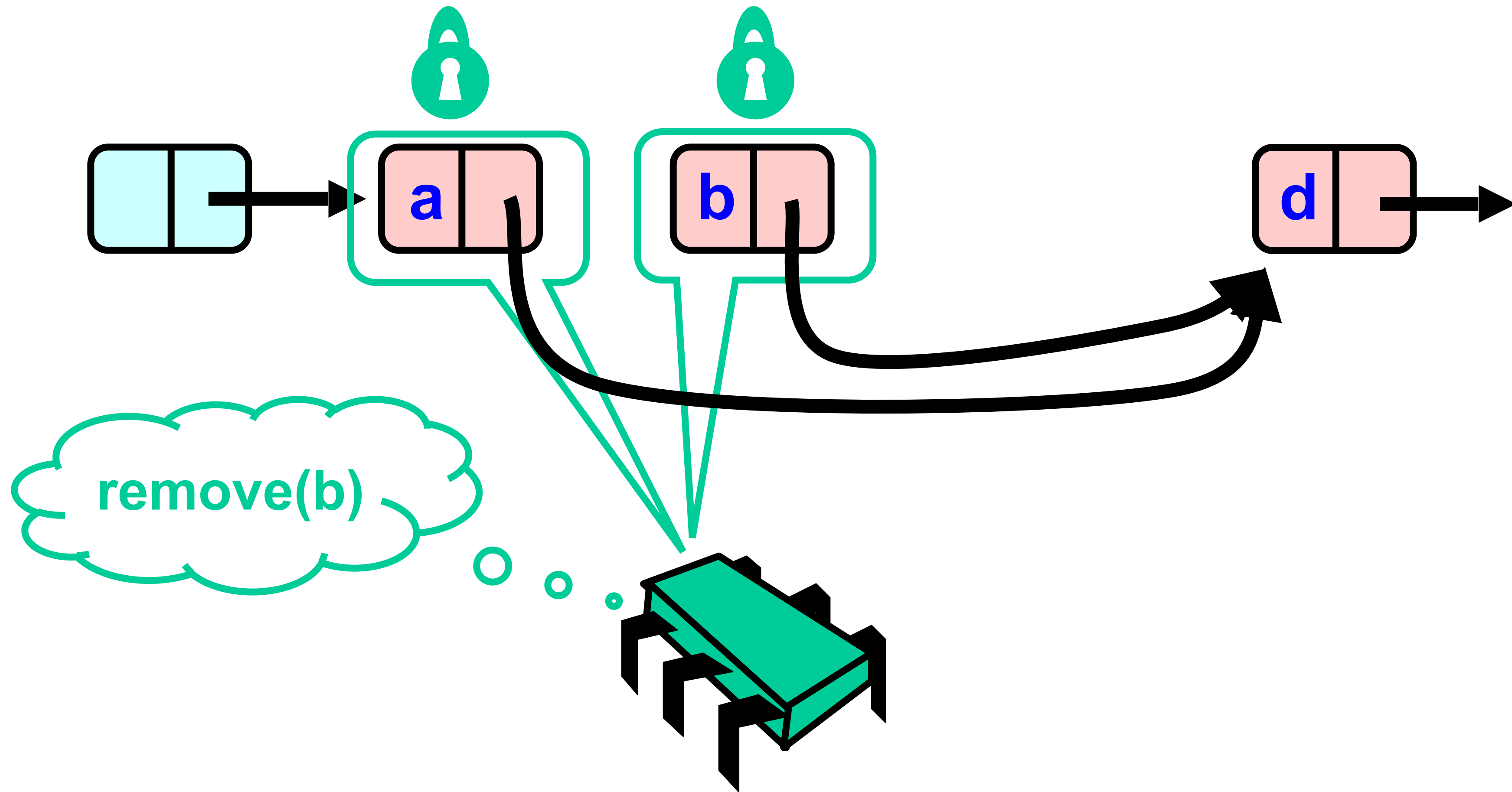


# Removing a Node

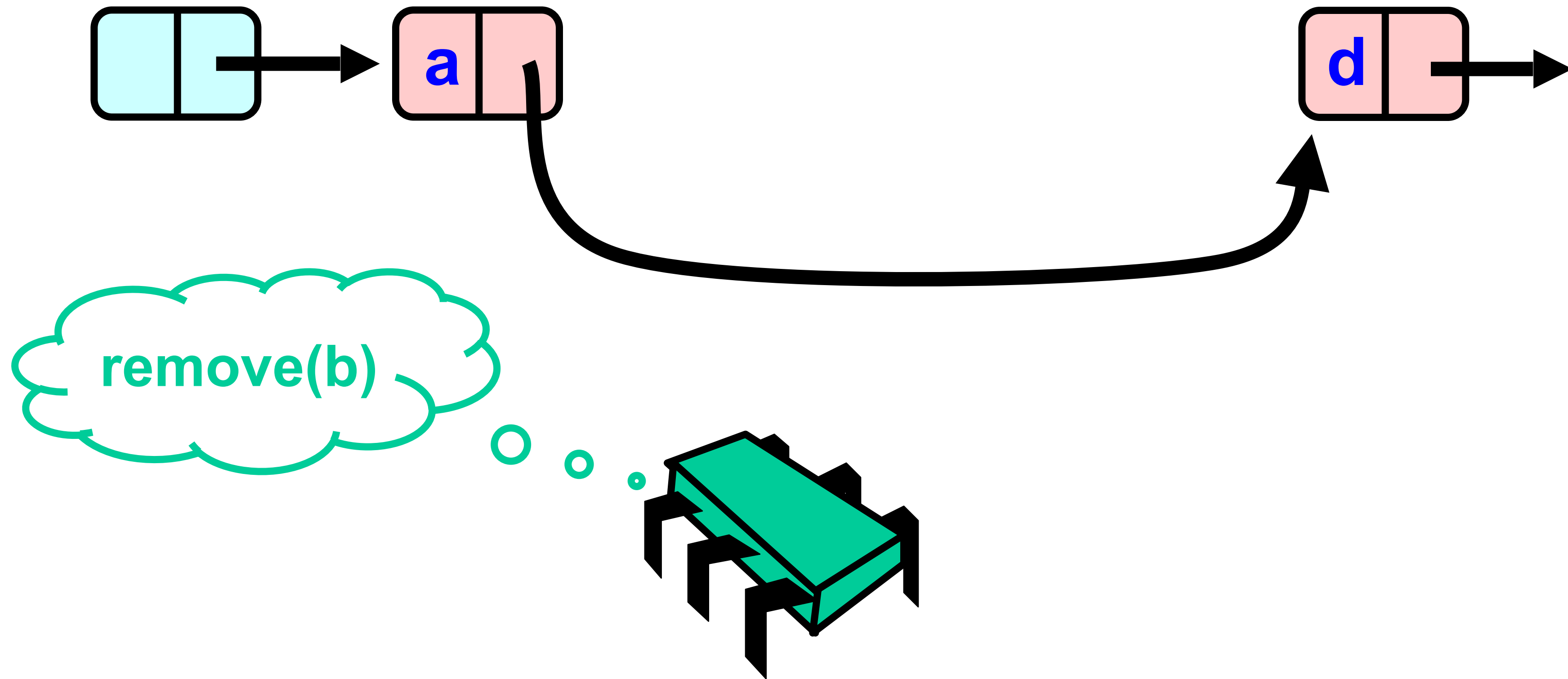




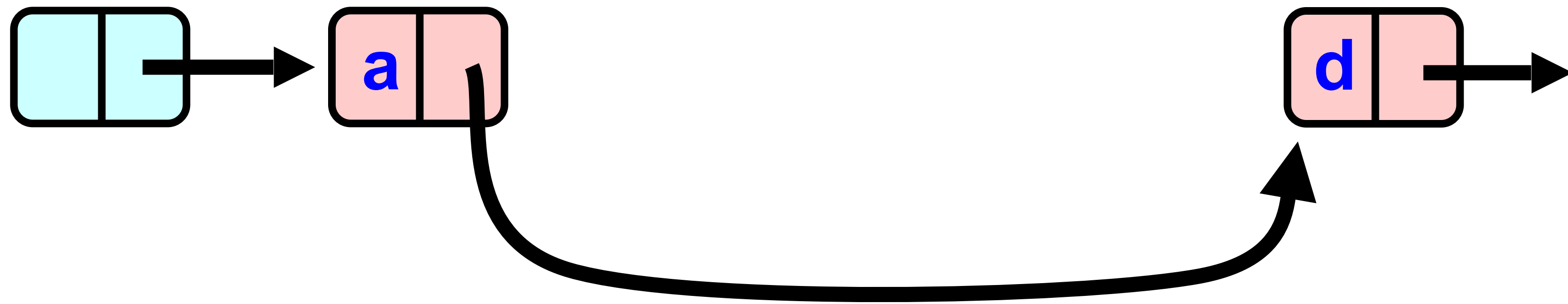
# Removing a Node



# Removing a Node



# Removing a Node



# Remove method

```
def remove(item: T): Boolean = {  
    var pred, curr: Node = null  
    val key = item.hashCode  
  
    try { ... } finally {  
        curr.unlock()  
        pred.unlock()  
    }  
}
```

# Remove method

```
def remove(item: T): Boolean = {  
  var pred, curr: Node = null  
  val key = item.hashCode  
  
  try { ... } finally {  
    curr.unlock()  
    pred.unlock()  
  }  
}
```

**Key used to order node**

# Remove method

```
def remove(item: T): Boolean = {  
  var pred, curr: Node = null  
  val key = item.hashCode  
  
  try { ... } finally {  
    curr.unlock()  
    pred.unlock()  
  }  
}
```

**Predecessor and current nodes**

# Remove method

```
def remove(item: T): Boolean = {  
  var pred, curr: Node = null  
  val key = item.hashCode  
  
  try { ... } finally {  
    curr.unlock()  
    pred.unlock()  
  }  
}
```

**Make sure  
locks released**

# Remove method

```
def remove(item: T): Boolean = {  
  var pred, curr: Node = null  
  val key = item.hashCode  
  
  try { ... } finally {  
    curr.unlock()  
    pred.unlock()  
  }  
}
```

**Everything else**



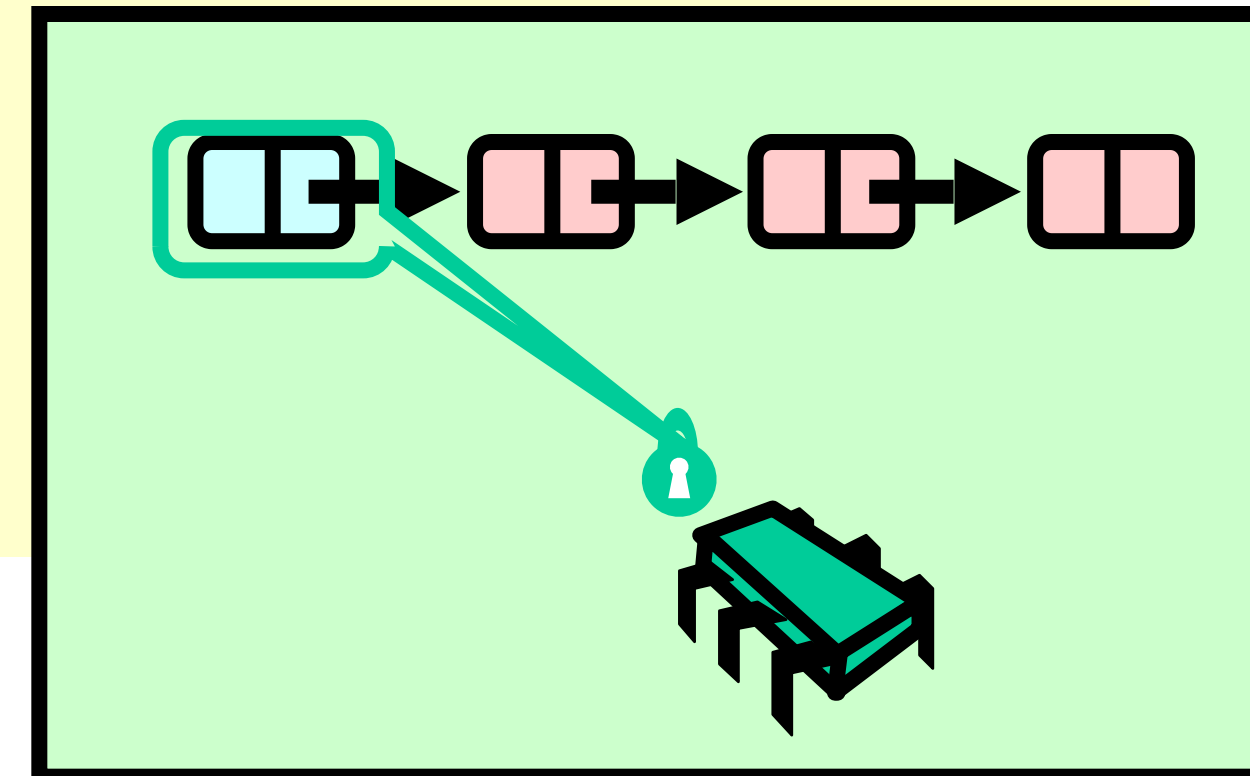
# Remove method

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

# Remove method

**lock pred == head**

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

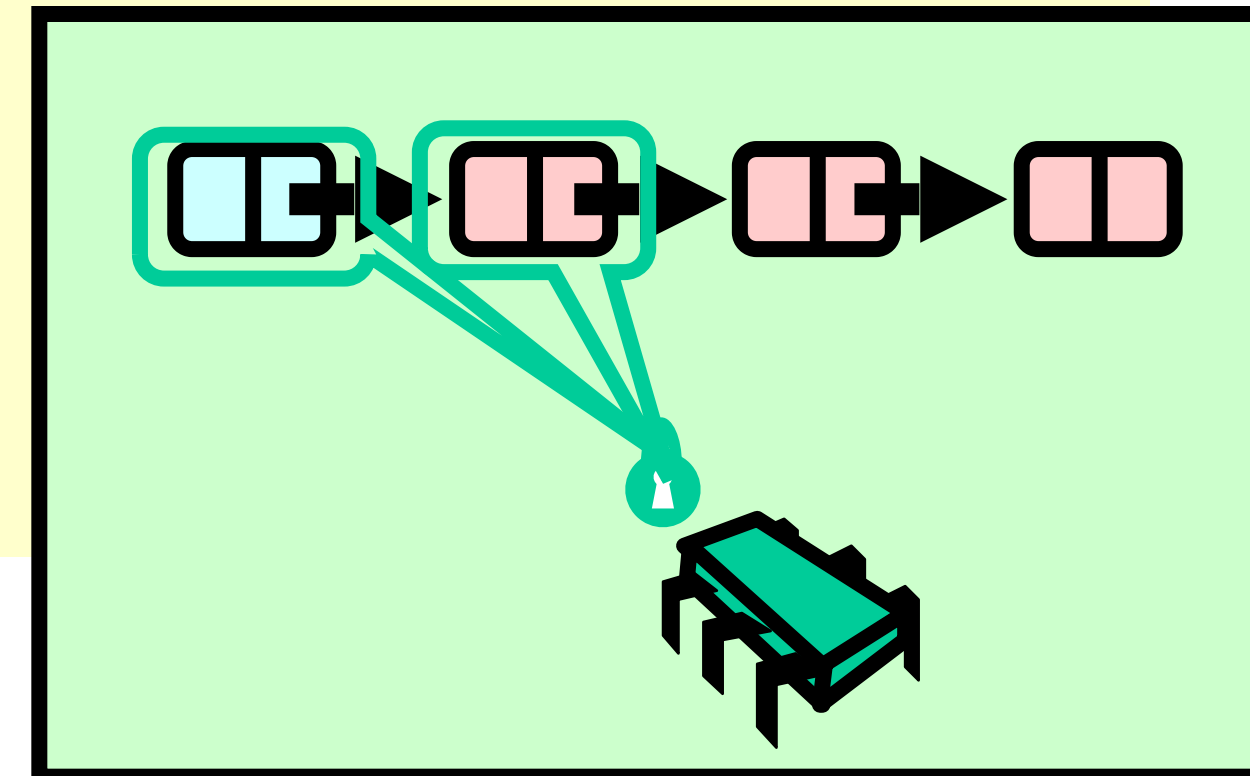


# Remove method

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

**Lock current**

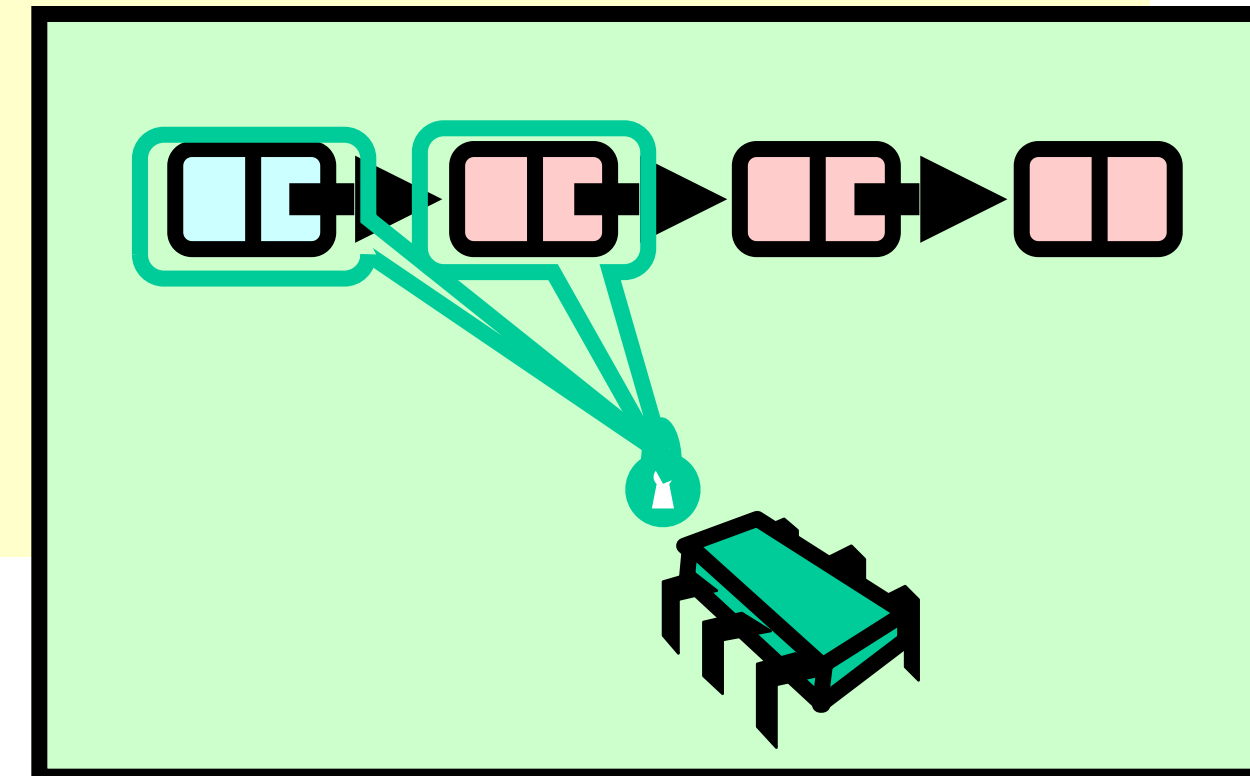
**curr = pred.next  
curr.lock()**



# Remove method

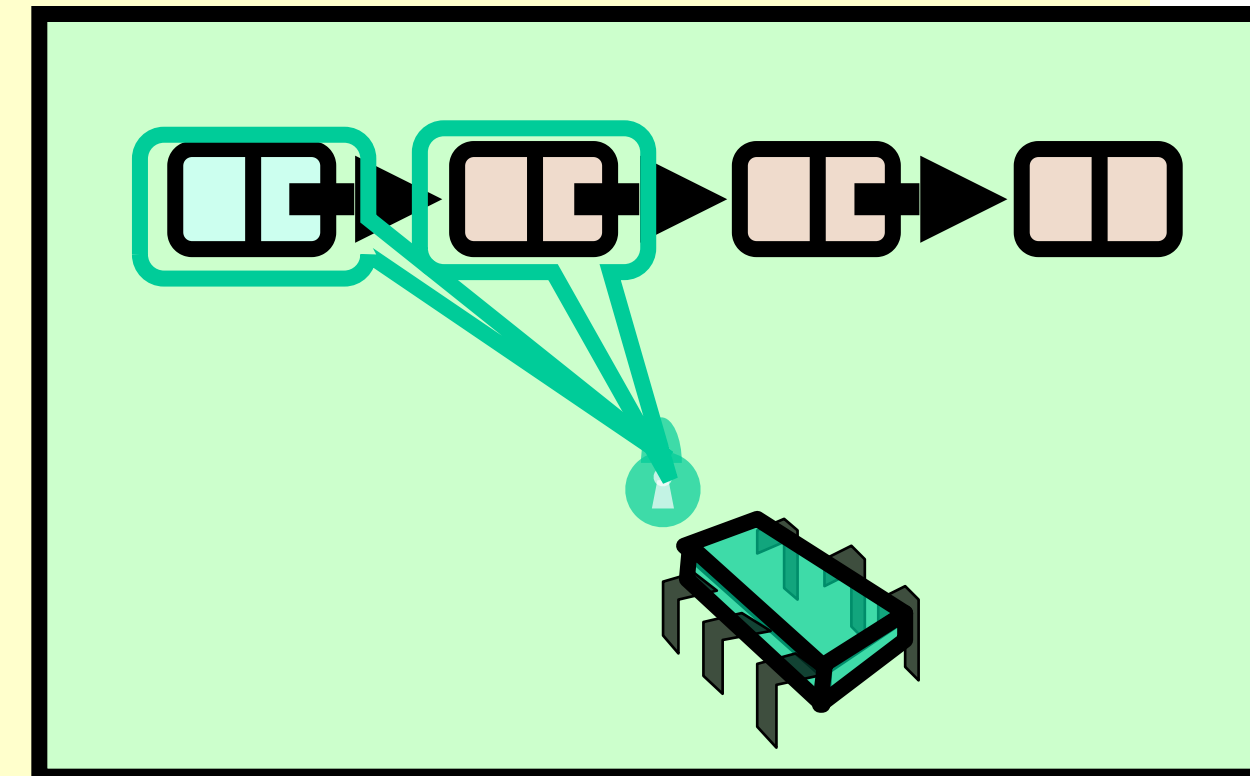
```
try {  
  pred = head  
  pred.lock()  
  curr = pred.next  
  curr.lock()  
  ...  
} finally { ... }
```

**Traversing list**



# Remove: searching

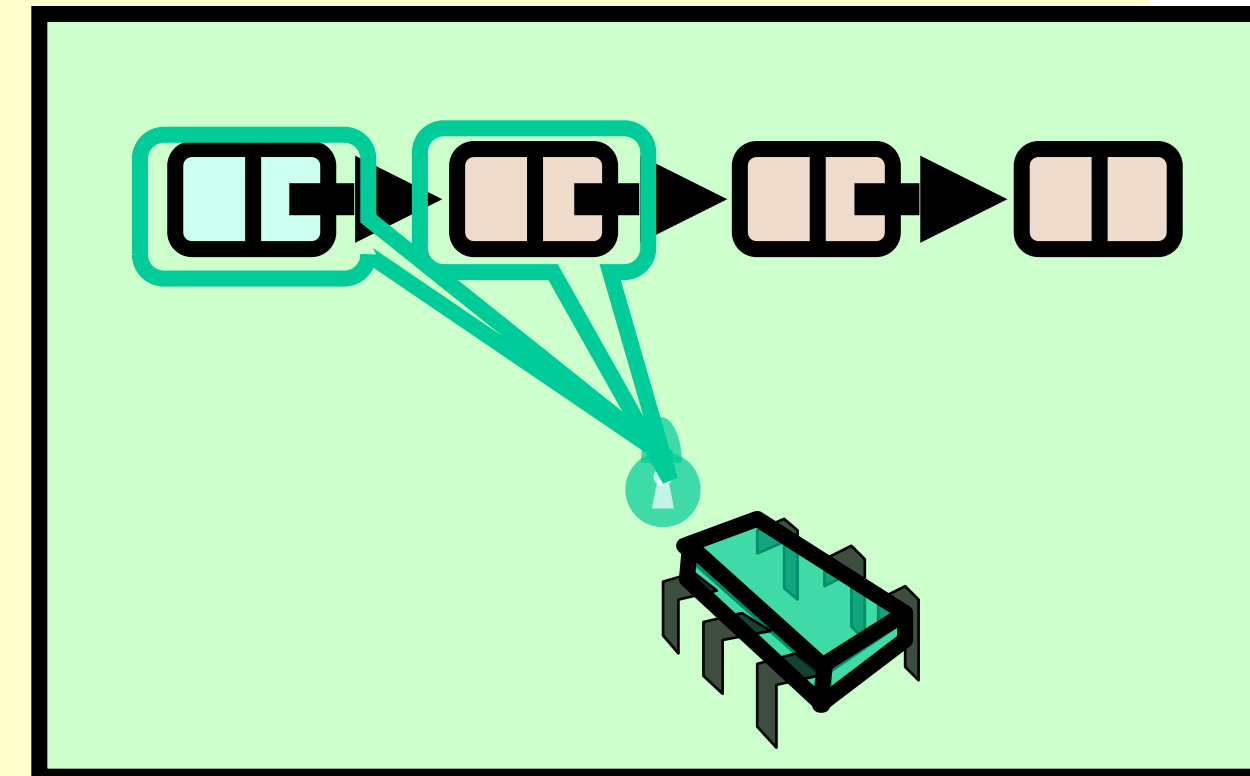
```
while (curr.key <= key) {  
  if (item == curr.item) {  
    pred.next = curr.next  
    return true  
  }  
  pred.unlock()  
  pred = curr  
  curr = curr.next  
  curr.lock()  
}  
return false
```



# 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
```

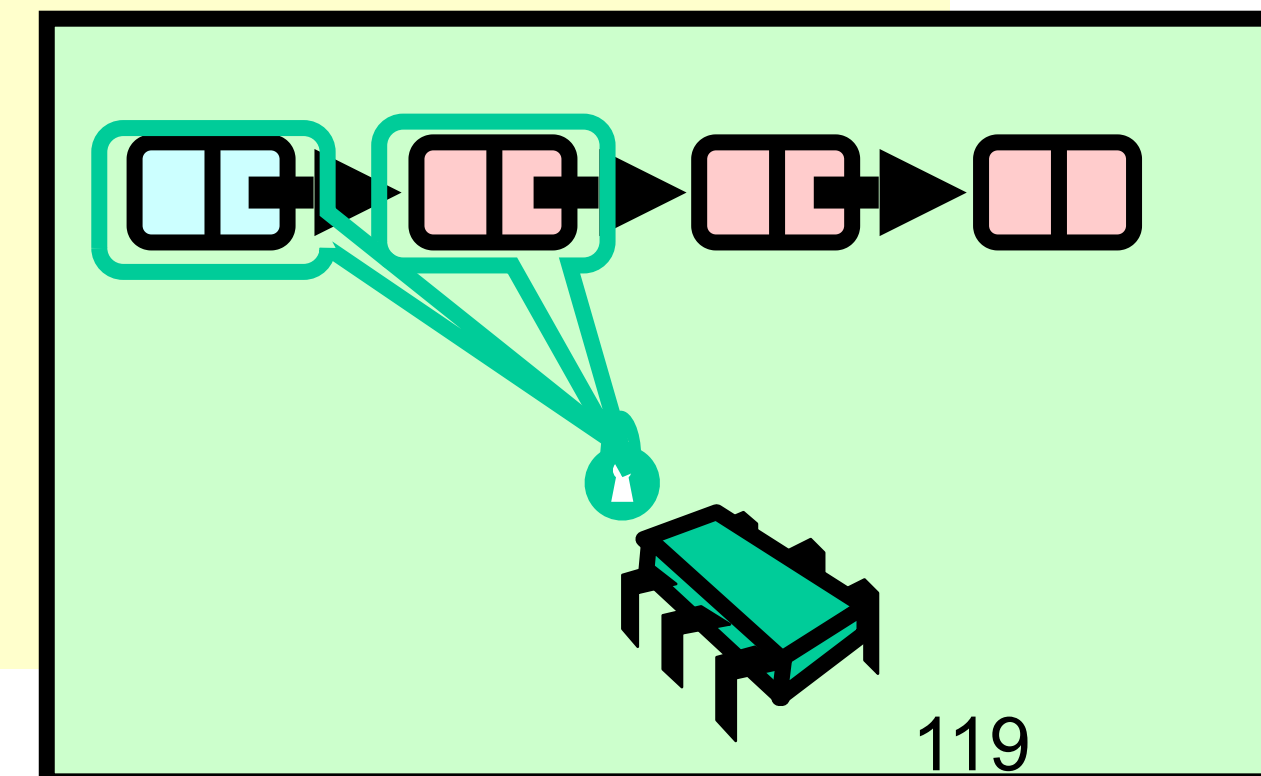
**Search key range**



# 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
```

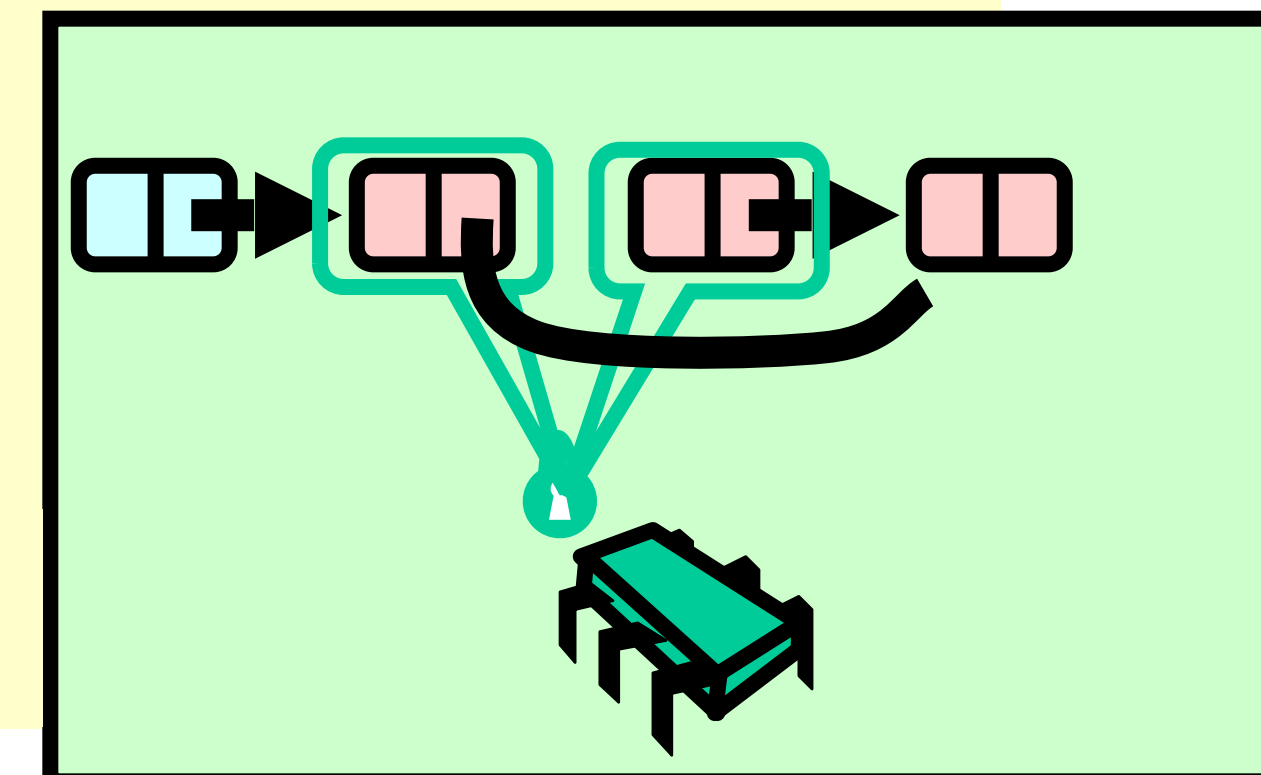
**At start of each loop:  
curr and pred locked**



# 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()  
}  
return false
```

**If item found, remove node**



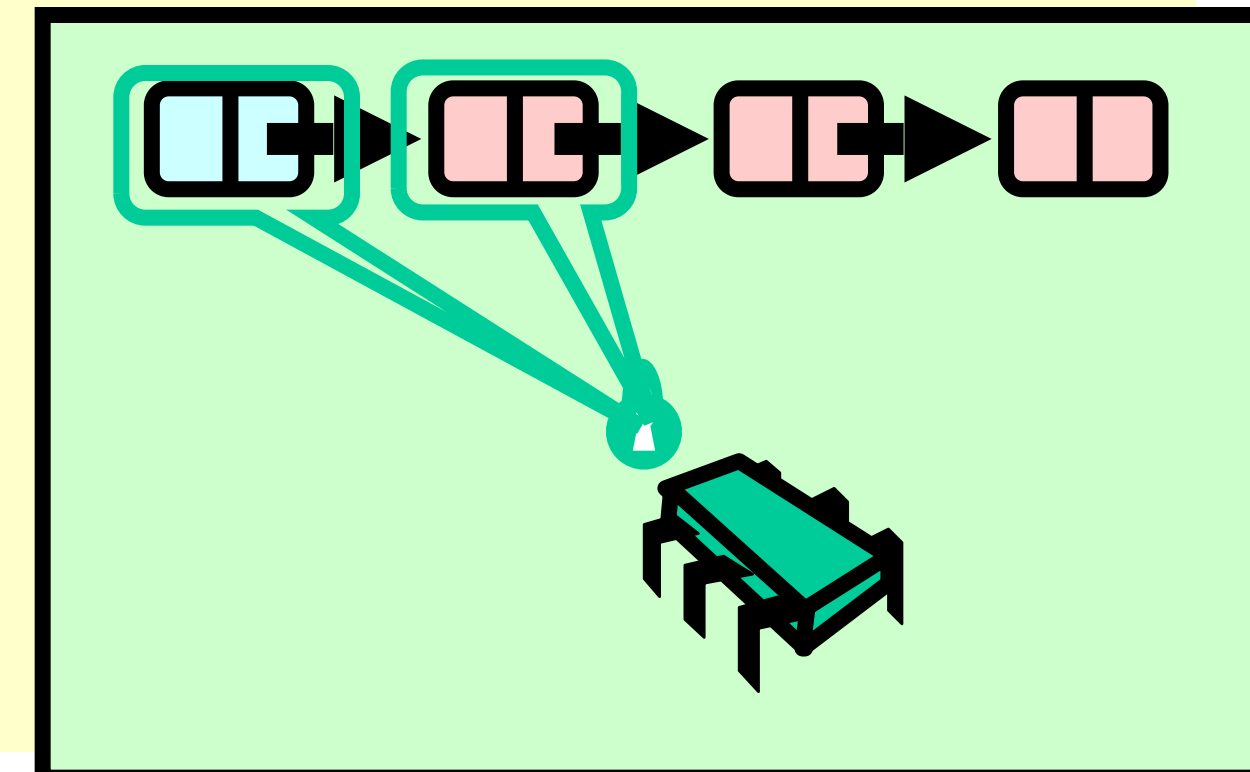


# Remove: searching

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

`pred.unlock()`

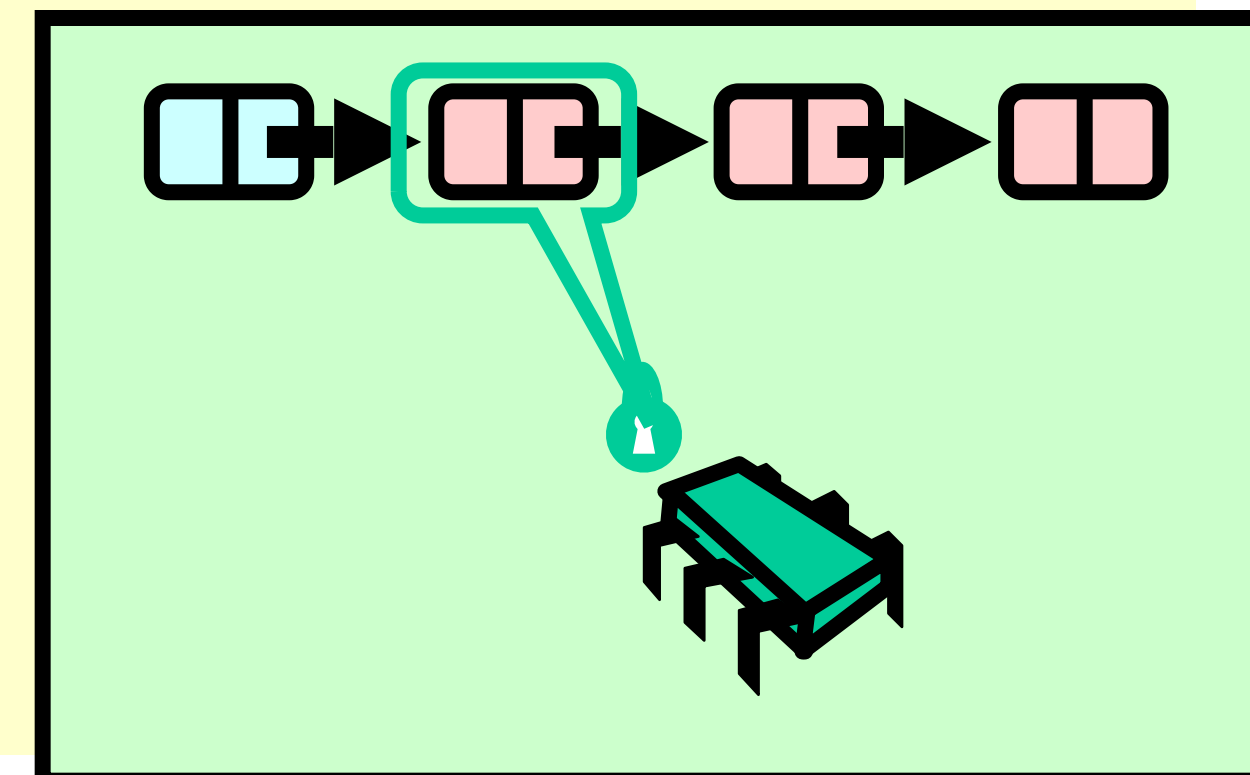
**Unlock predecessor**



# Remove: searching

Only one node locked!

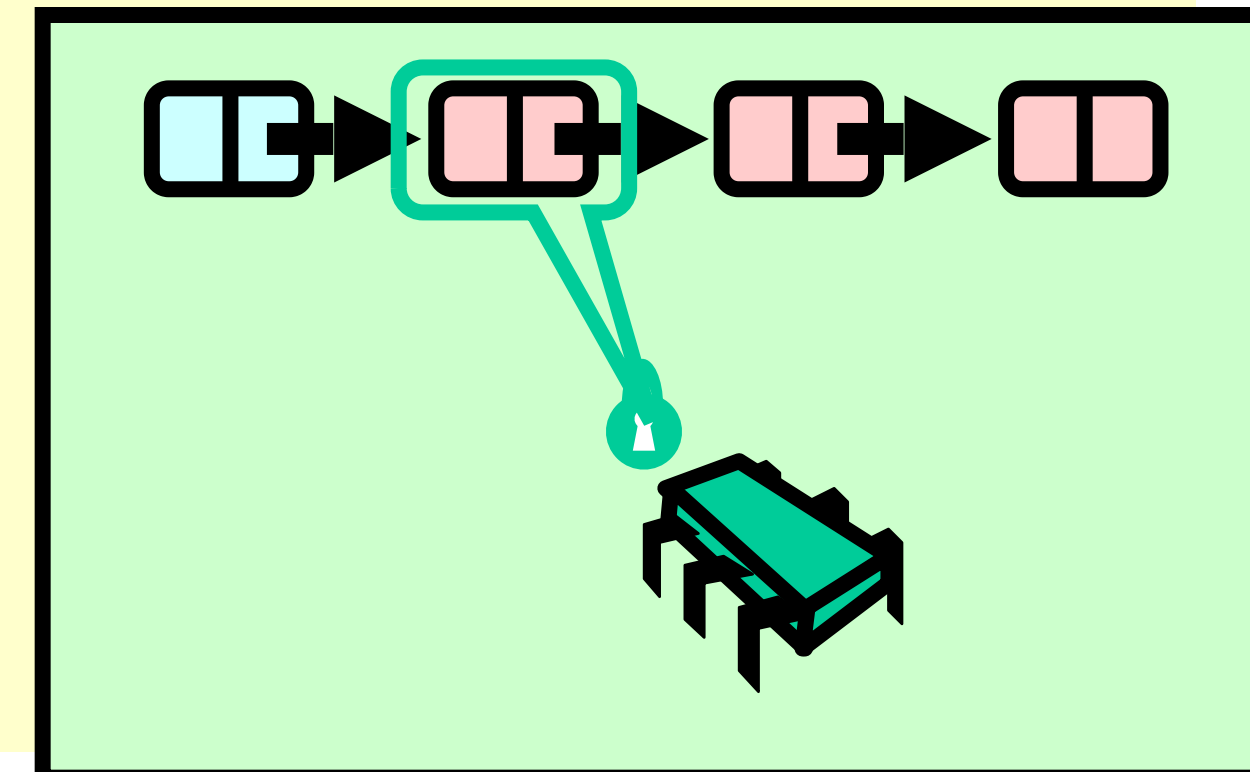
```
while (curr.key <= key) {  
  if (item == curr.item) {  
    pred.next = curr.next  
    return true  
  }  
  pred.unlock()  
  pred = curr  
  curr = curr.next  
  curr.lock()  
}  
return false
```



# Remove: searching

**demote current**

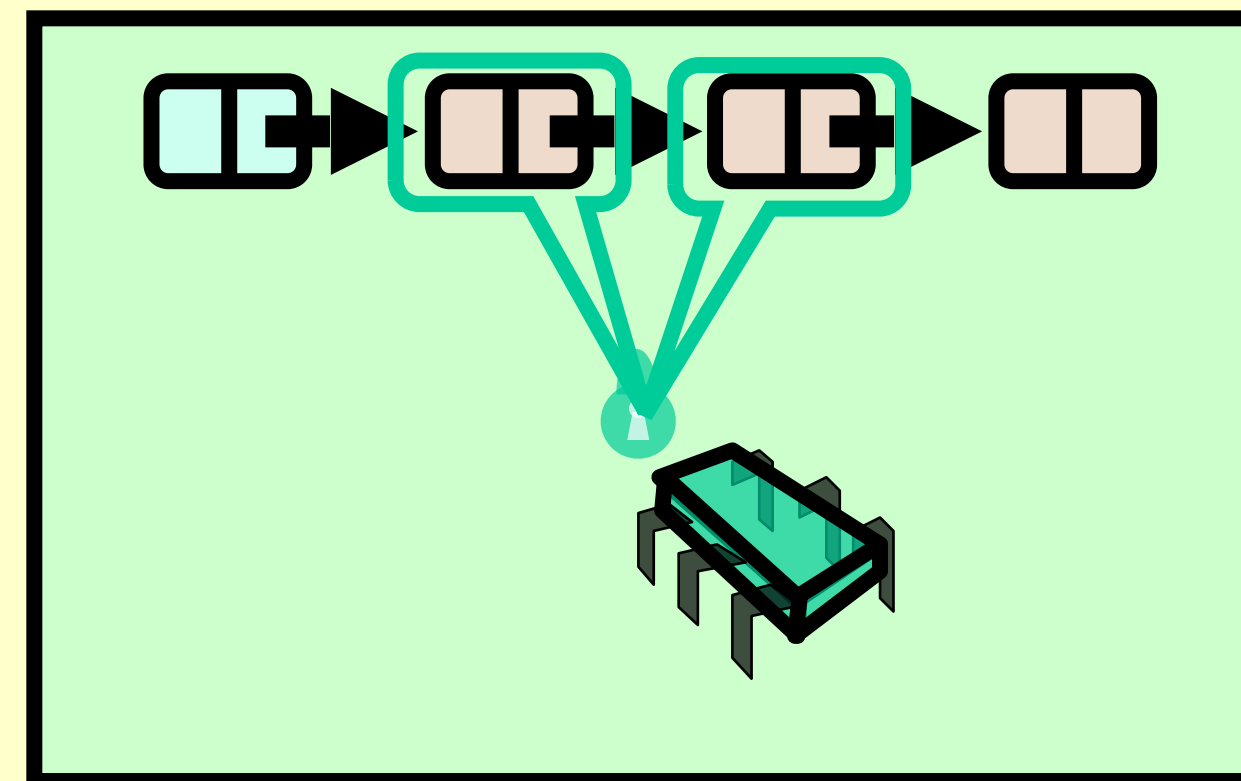
```
while (curr.key <= key) {  
  if (item == curr.item) {  
    pred.next = curr.next  
    return true  
  }  
  pred.unlock()  
  pred = curr  
  curr = curr.next  
  curr.lock()  
}  
return false
```



# Remove: searching

## Find and lock new current

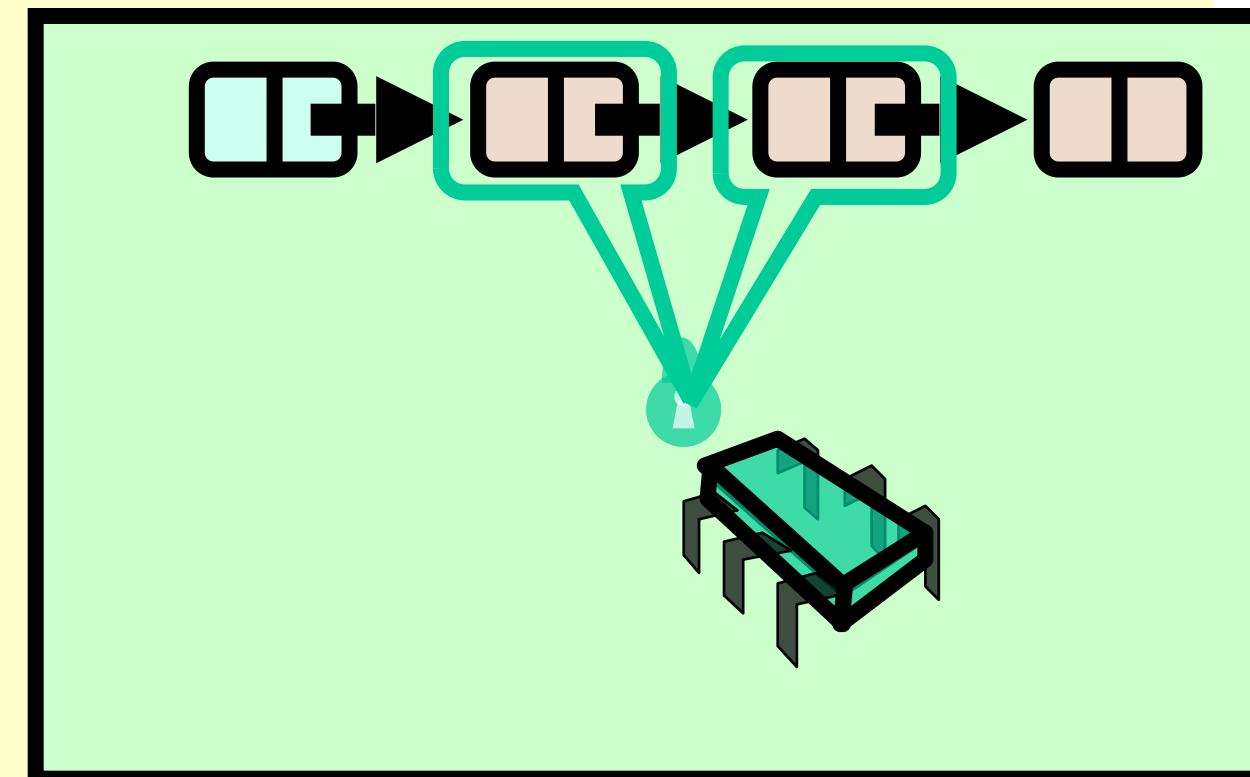
```
while (curr.key <= key) {  
  if (item == curr.item) {  
    pred.next = curr.next  
    return true  
  }  
  pred.unlock()  
  pred = currNode  
  curr = curr.next  
  curr.lock()  
}  
return false
```



# Remove: searching

**Loop 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()  
}
```

**Otherwise, not present**



**return false**

# 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

# 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
```

- **pred** reachable from head
- **curr** is pred.next
- **So curr.item** is in the set



# 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 is present**

# Why remove() is linearizable

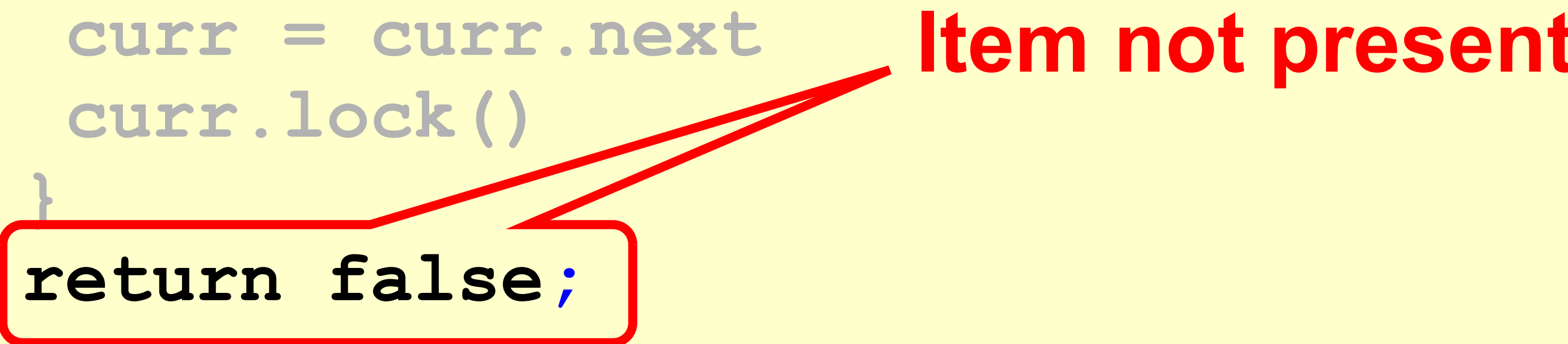
```
while (curr.key <= key) {  
  if (item == curr.item) {  
    pred.next = curr.next  
    return true  
  }  
  pred.unlock()  
  pred = curr  
  curr = curr.next  
  curr.lock()  
}  
return false
```

**Node locked, so no other thread can remove it ....**

# 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;
```

**Item not 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**

- pred reachable from head
- curr is pred.next
- pred.key < key
- key < curr.key

# 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**



# Adding Nodes

- To add node  $e$ 
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted
  - (Is successor lock actually required?)

# Same Abstraction Map

- $S(\text{head}) =$   
 $\{ x \mid \text{there exists } a \text{ such that}$ 
  - $a \text{ reachable from head and}$
  - $a.\text{item} = x$ $\}$

# Rep Invariant

- Easy to check that
  - tail always reachable from head
  - Nodes sorted, no duplicates



# Demo: Benchmarking Fine-Grained Lists

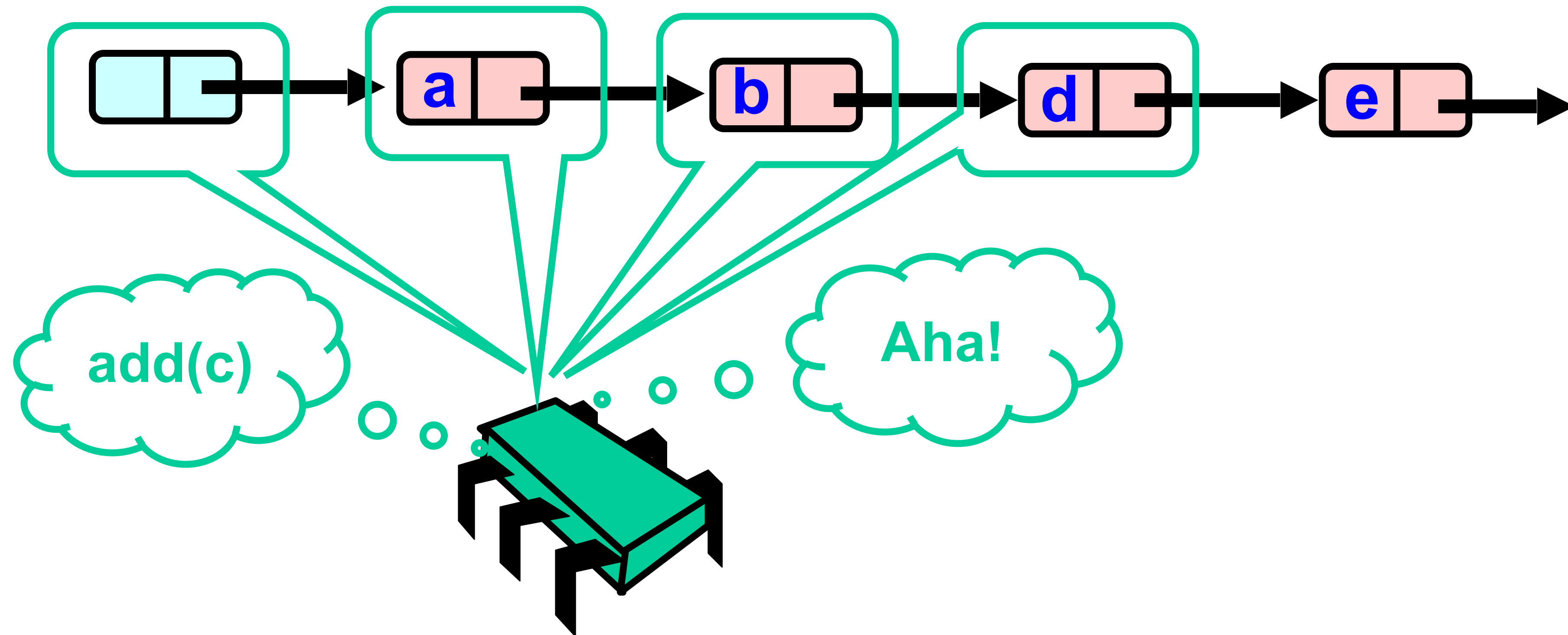
# Drawbacks

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

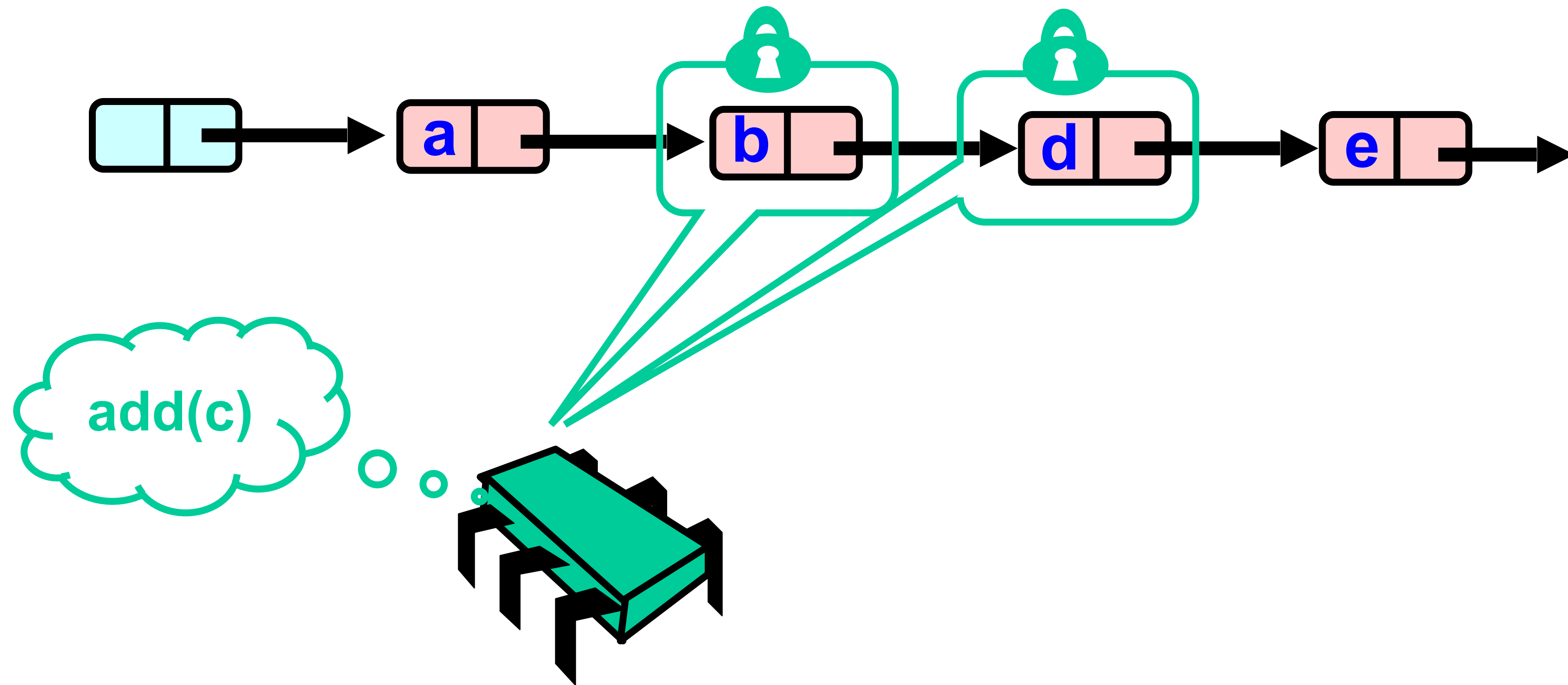
# Optimistic Synchronization

- Find nodes without locking
- Lock nodes
- Check that everything is OK

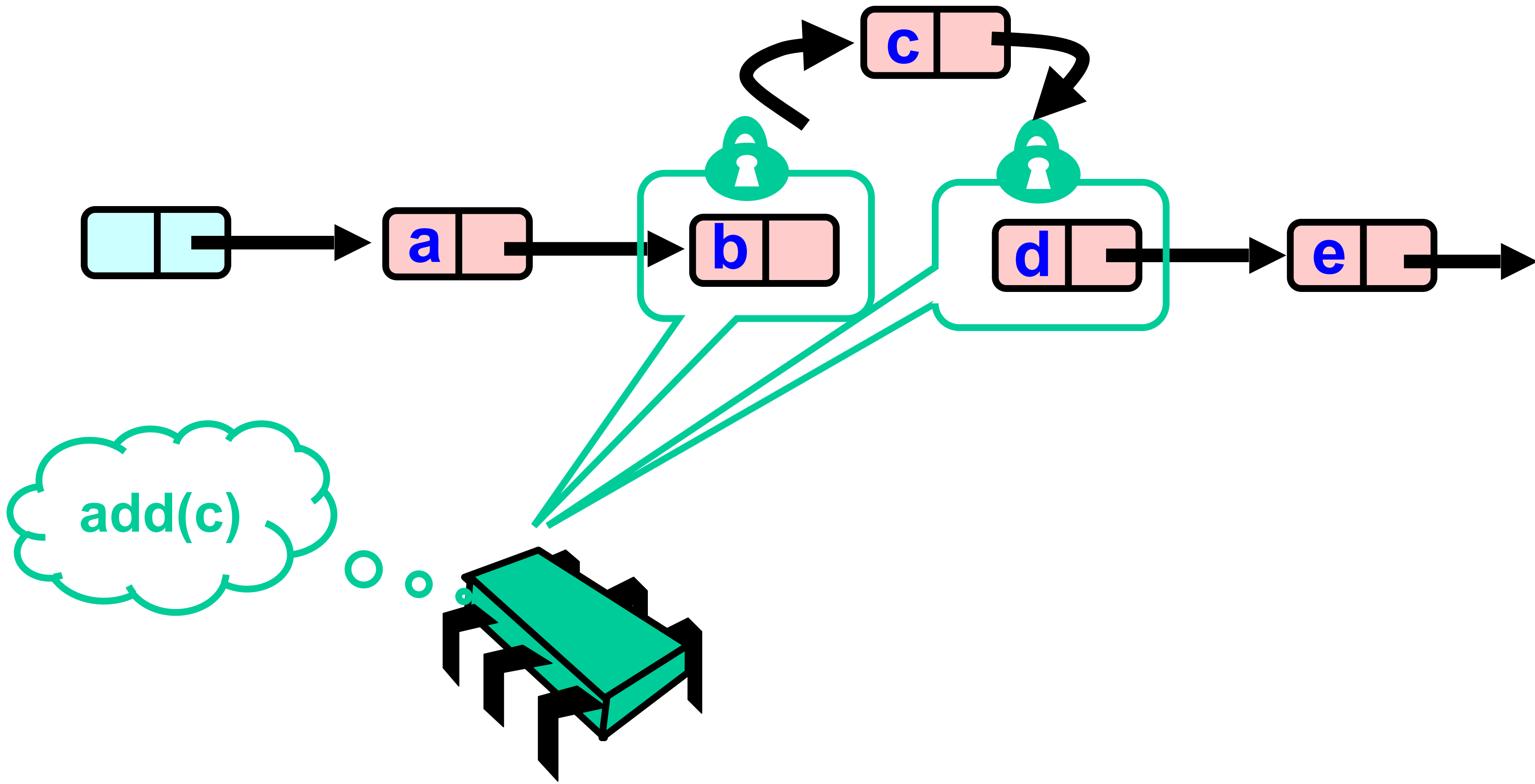
# Optimistic: Traverse without Locking



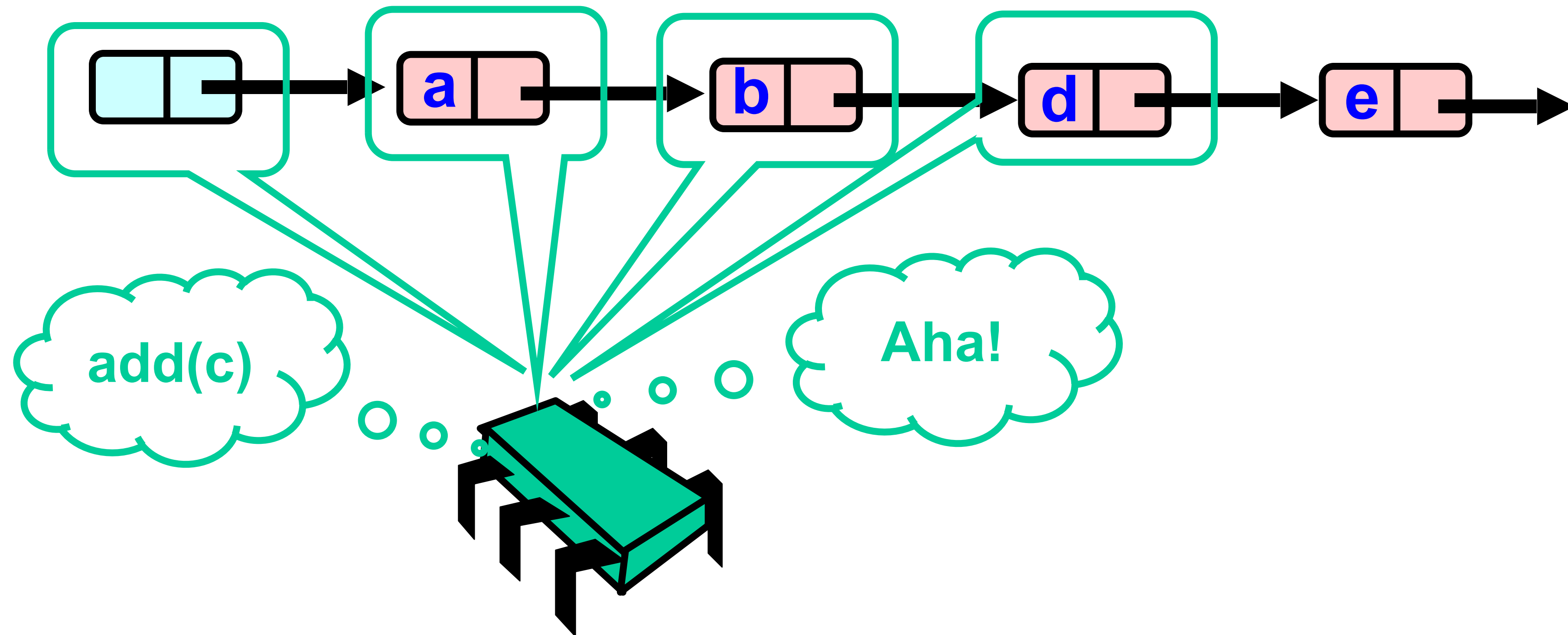
# Optimistic: Lock and Load



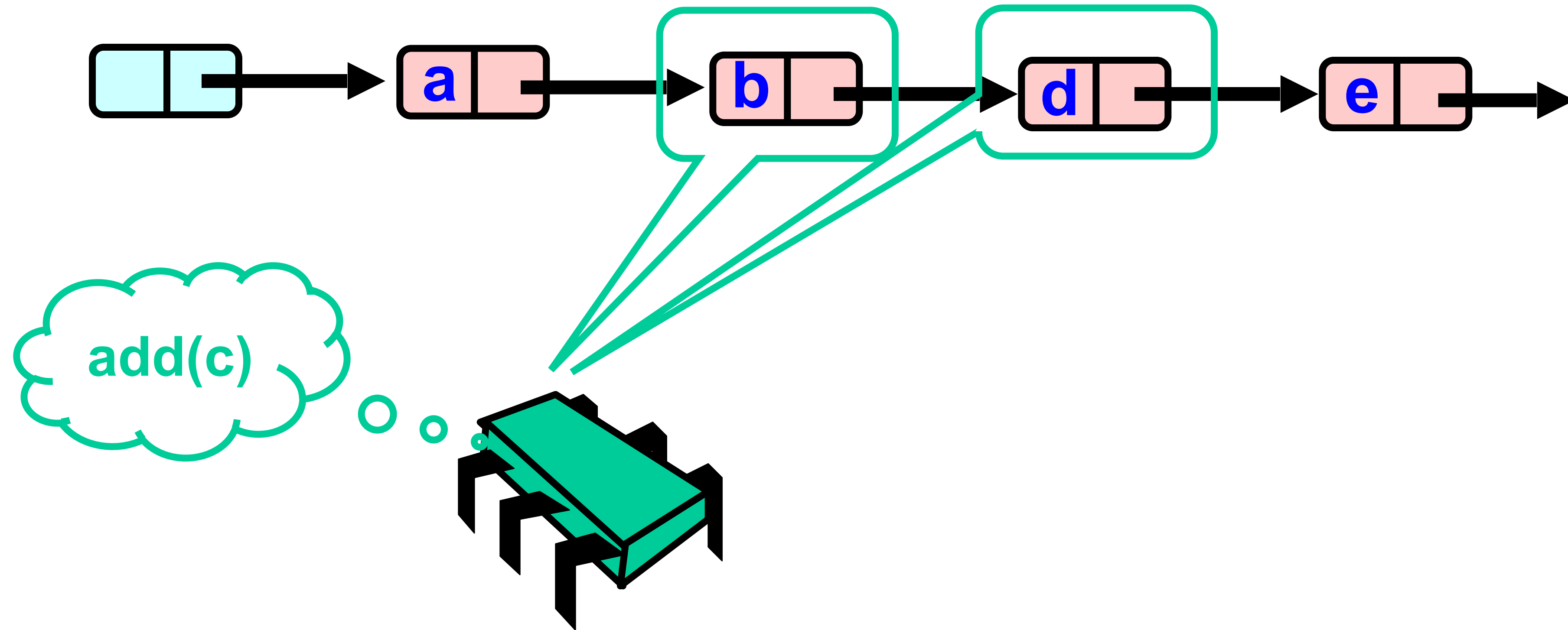
# Optimistic: Lock and Load



# What could go wrong?

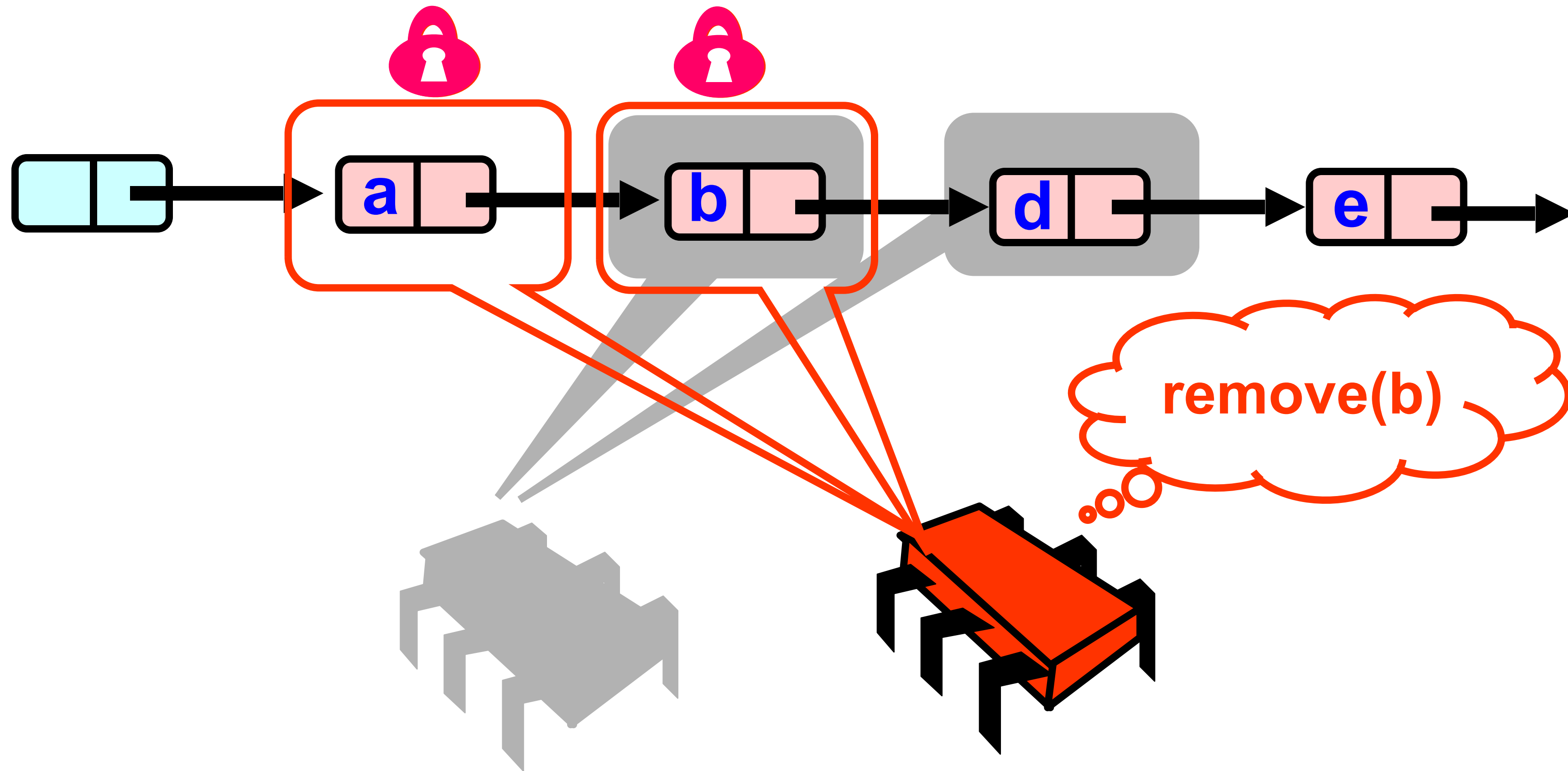


# What could go wrong?

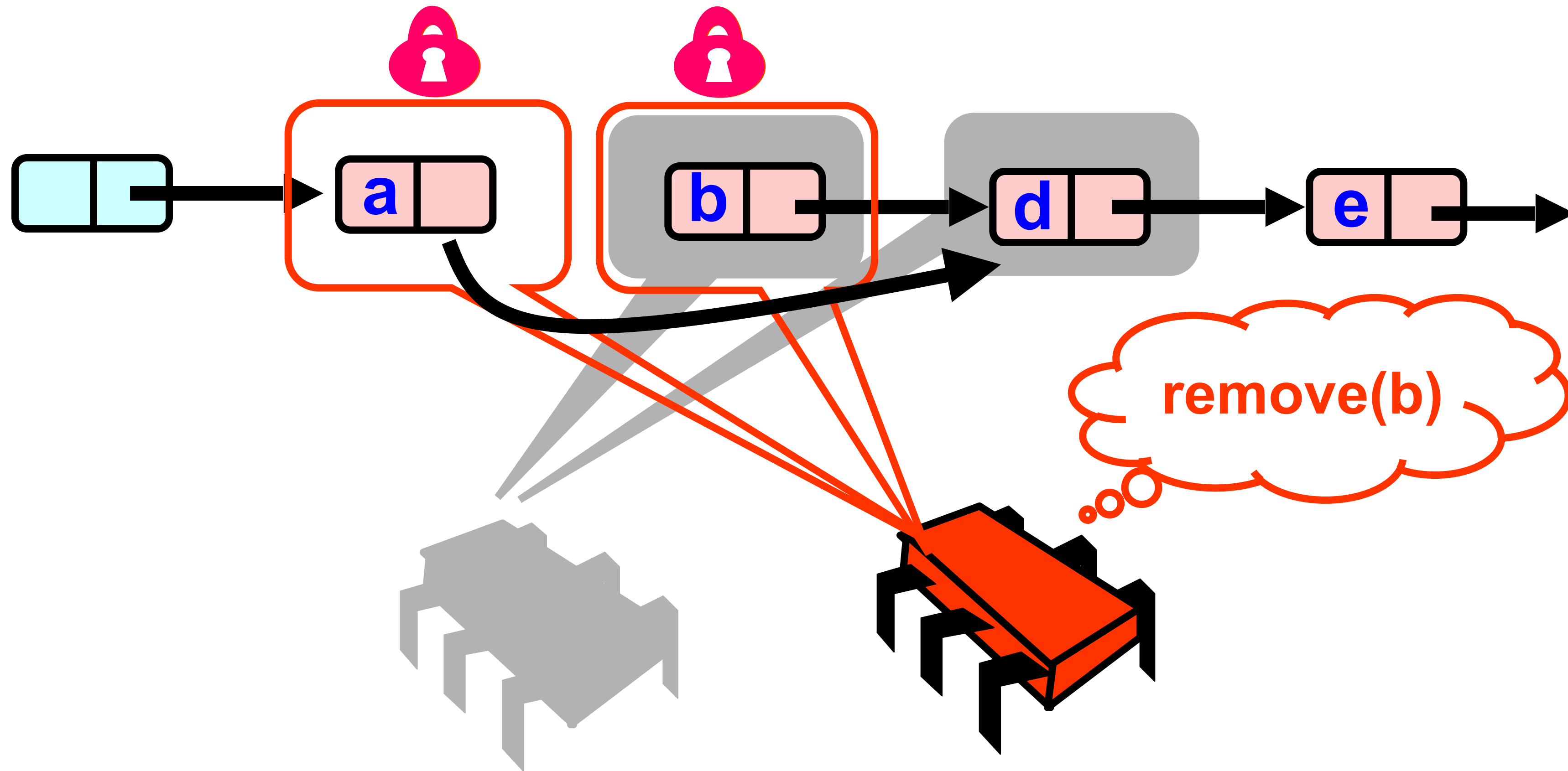




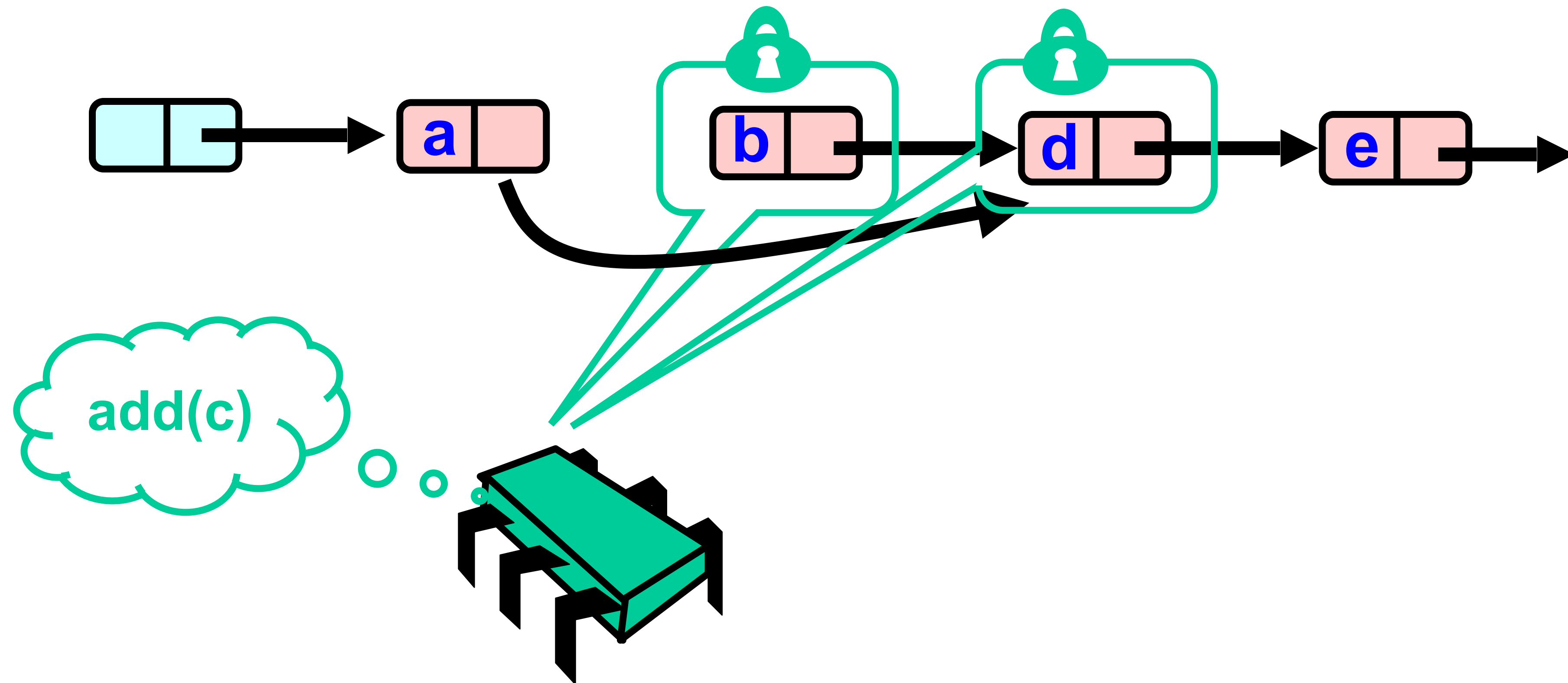
# What could go wrong?



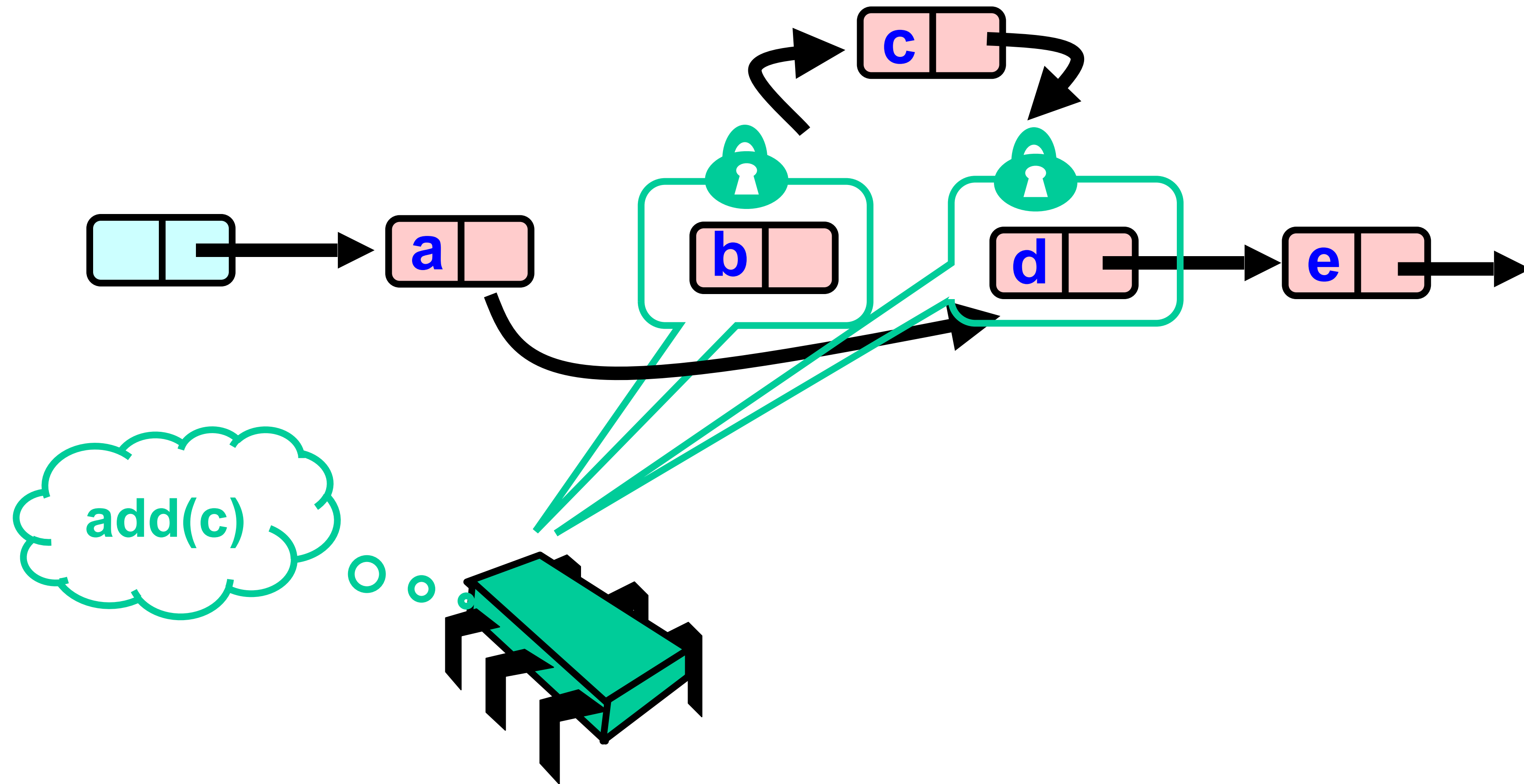
# What could go wrong?



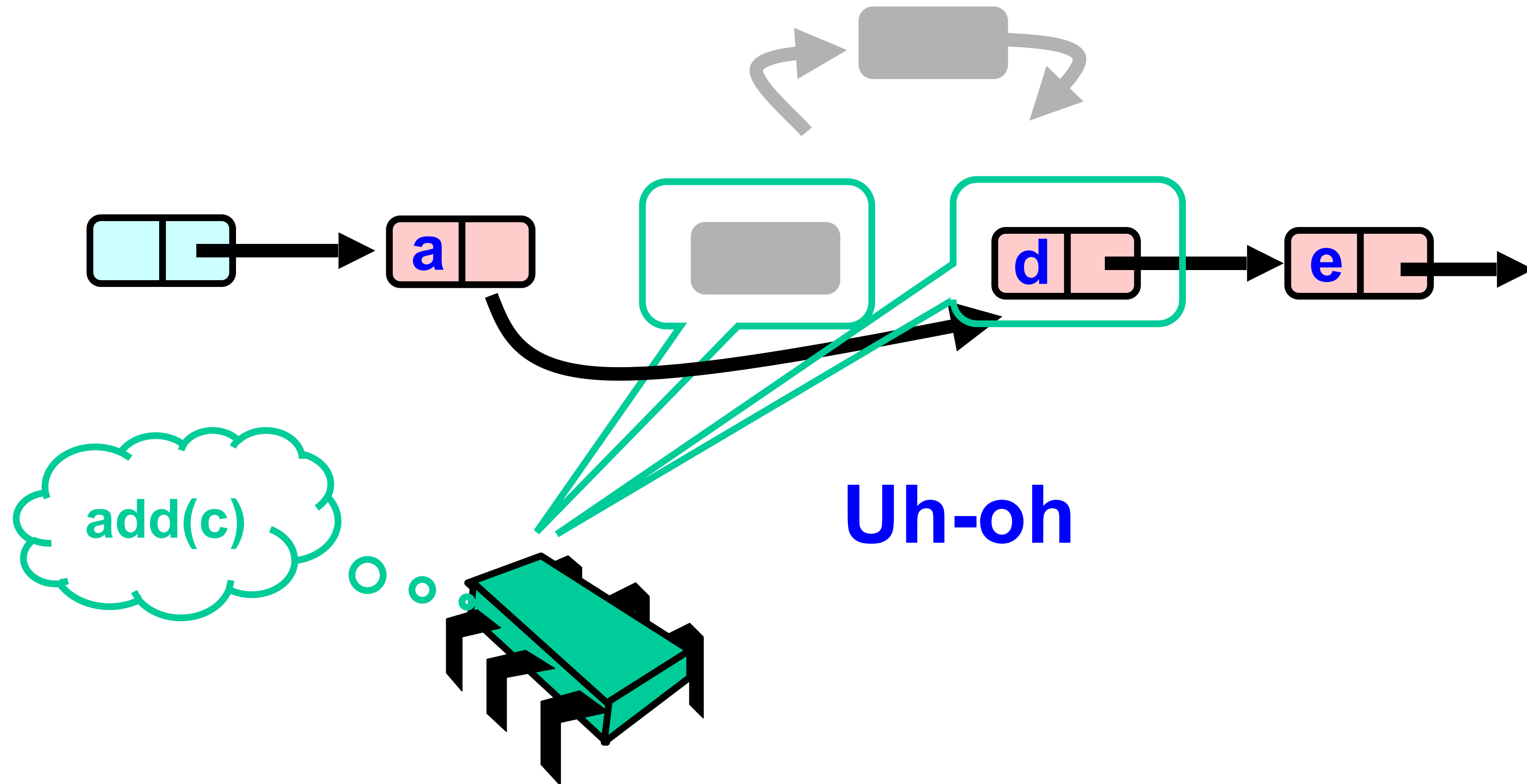
# What could go wrong?



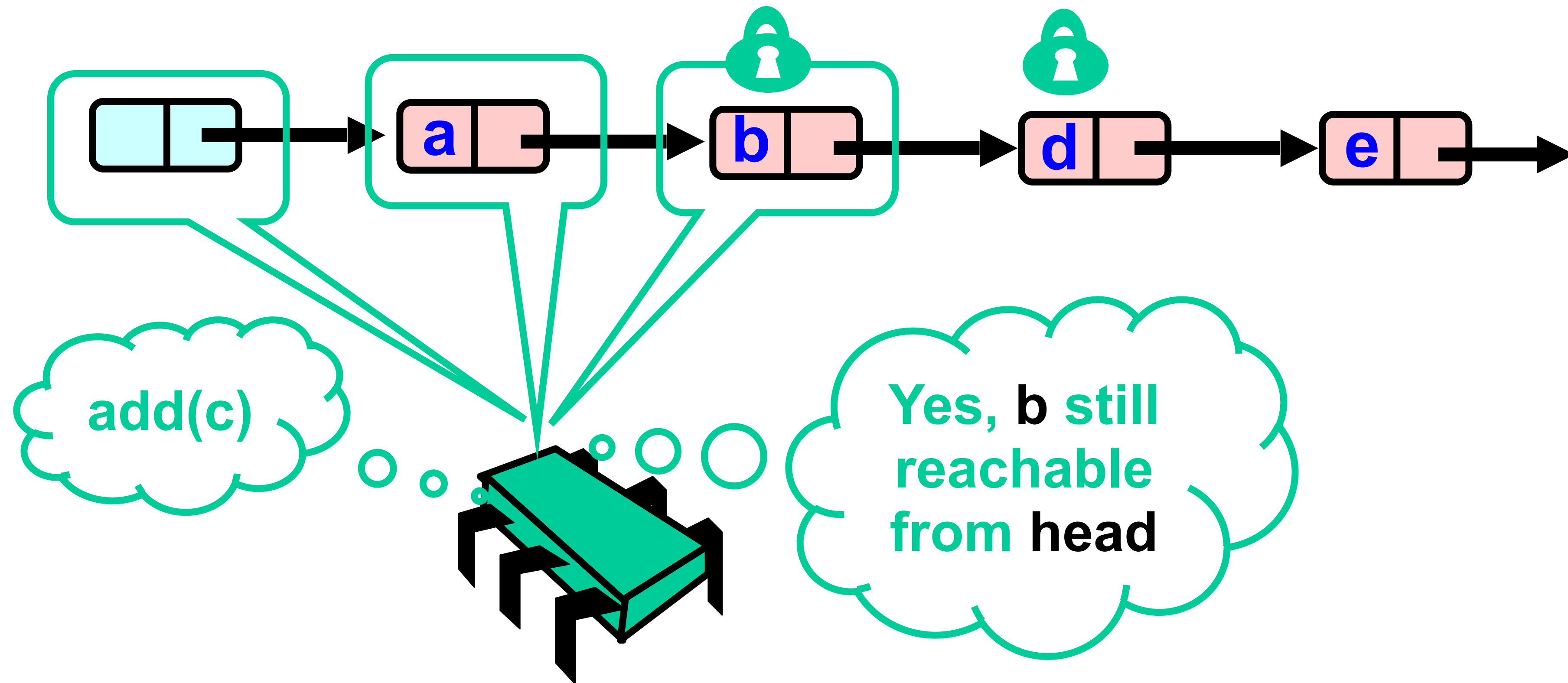
# What could go wrong?



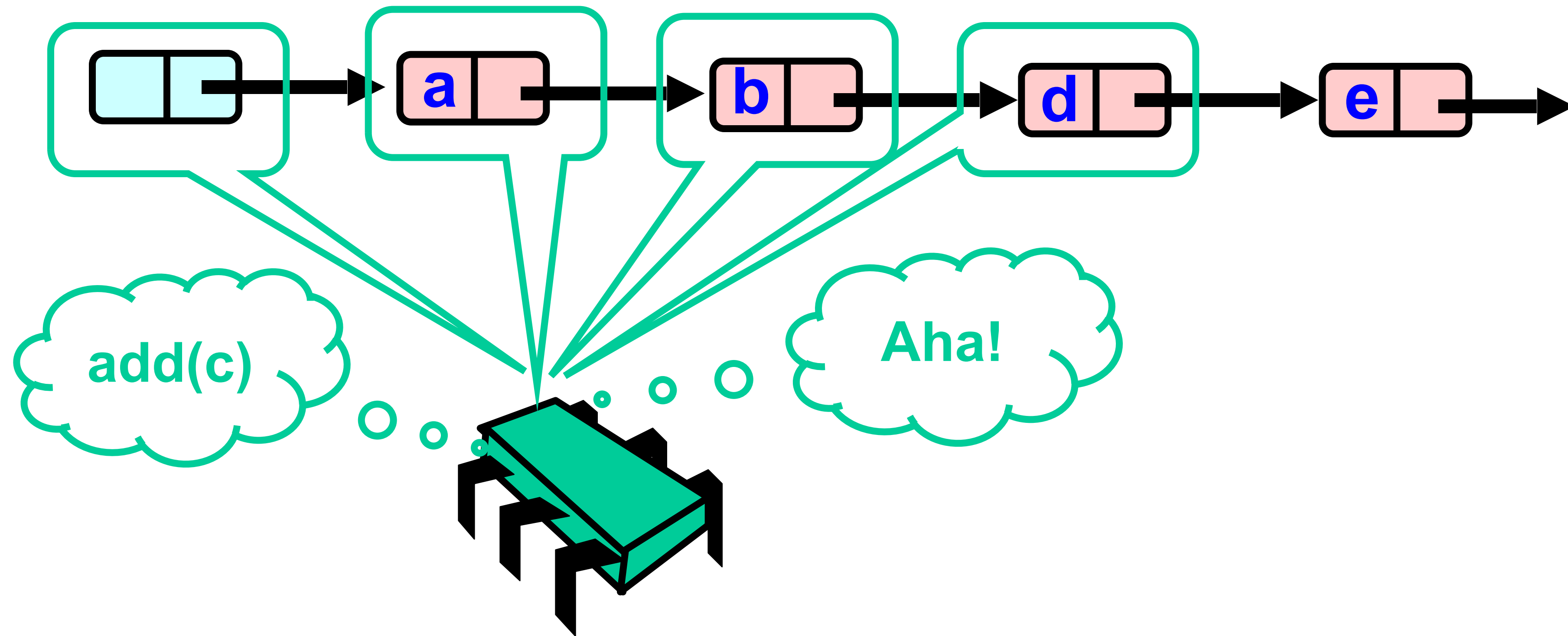
# What could go wrong?



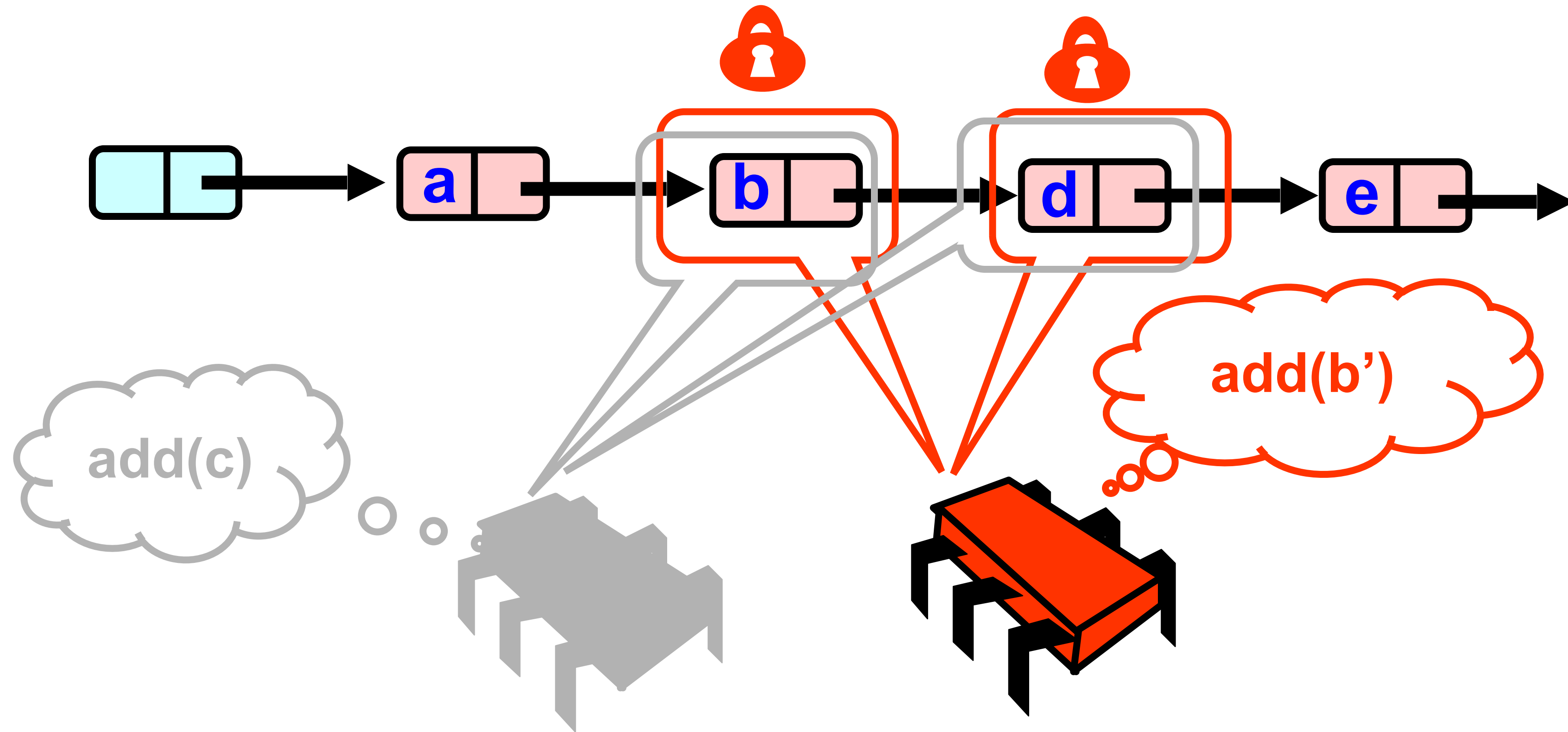
# Validate – Part 1



# What Else Could Go Wrong?

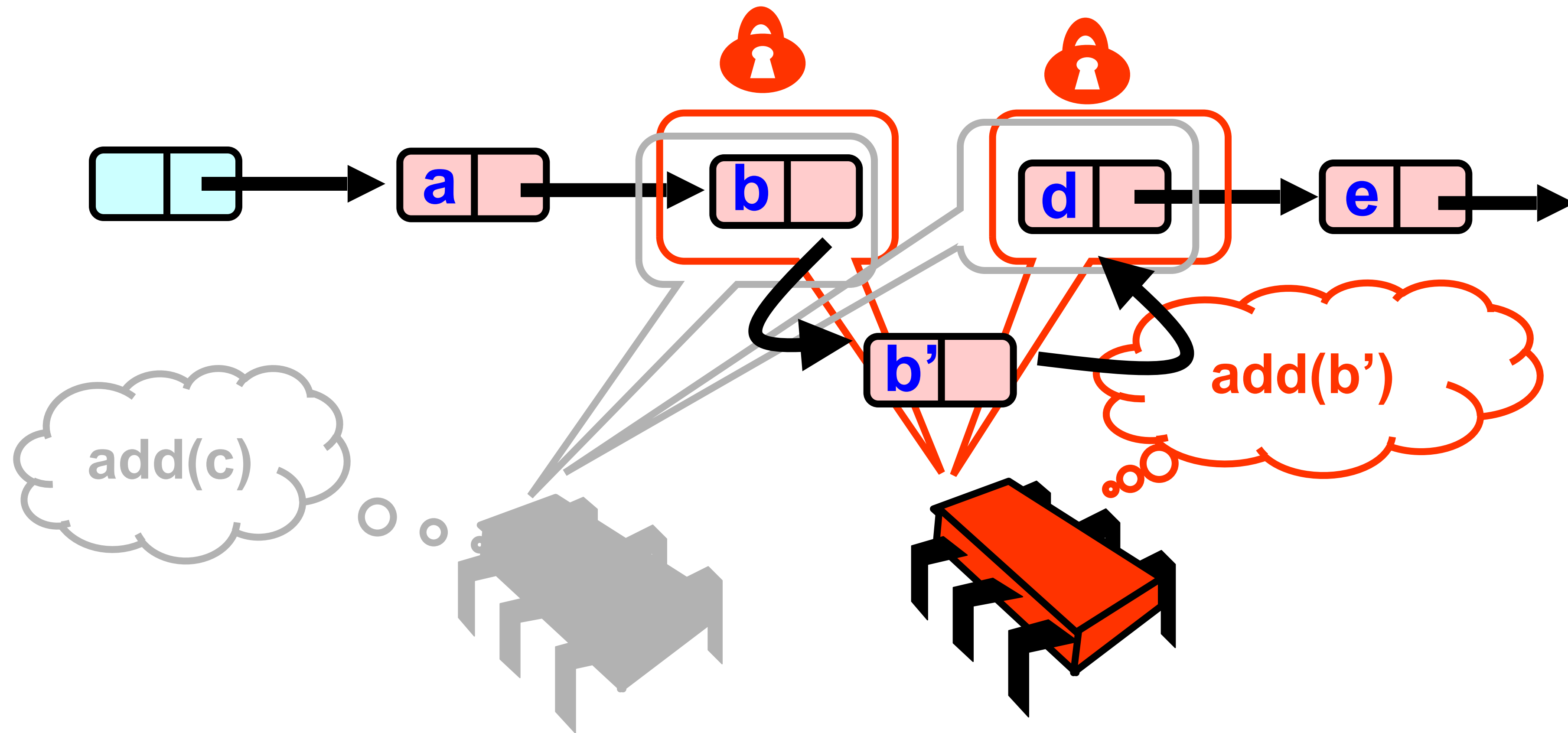


# What Else Could Go Wrong?

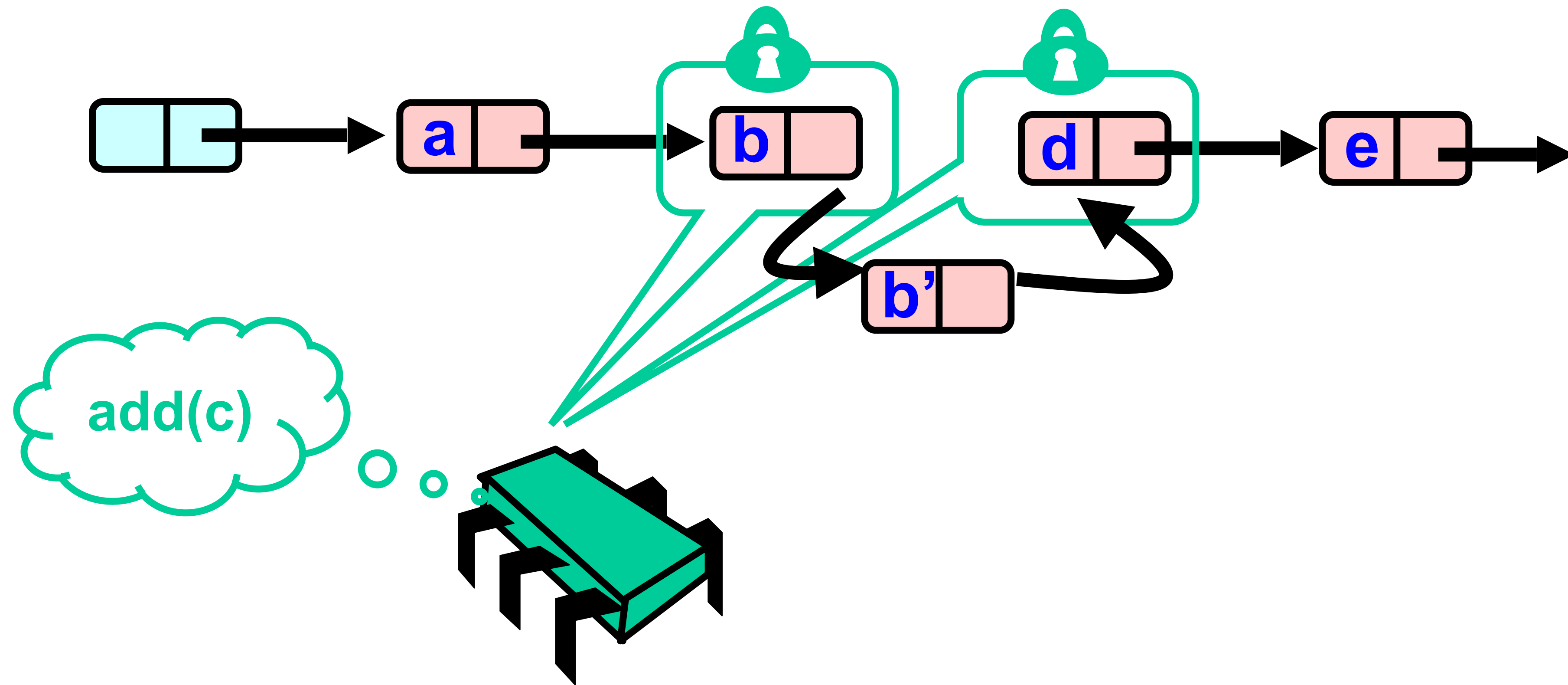




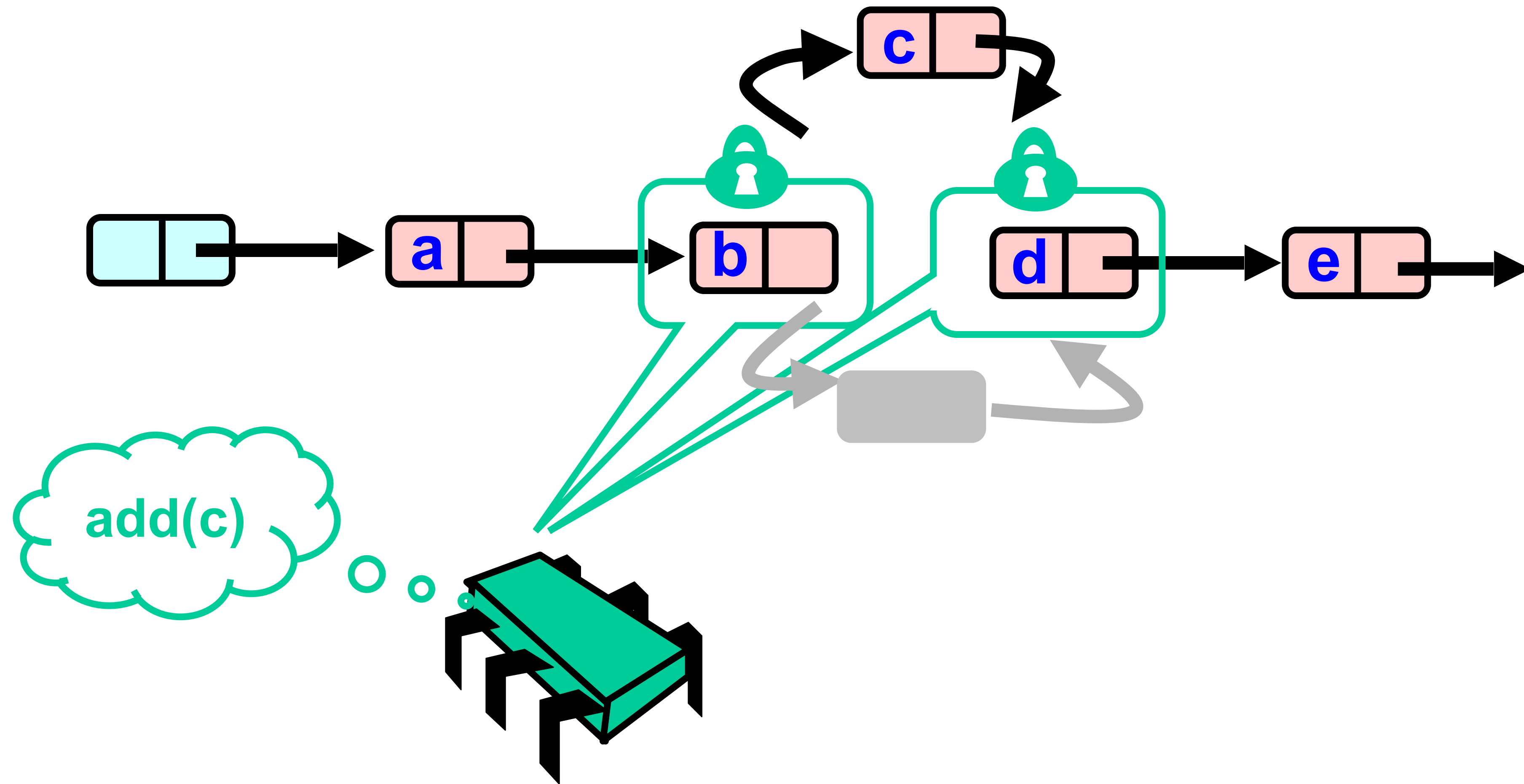
# What Else Could Go Wrong?



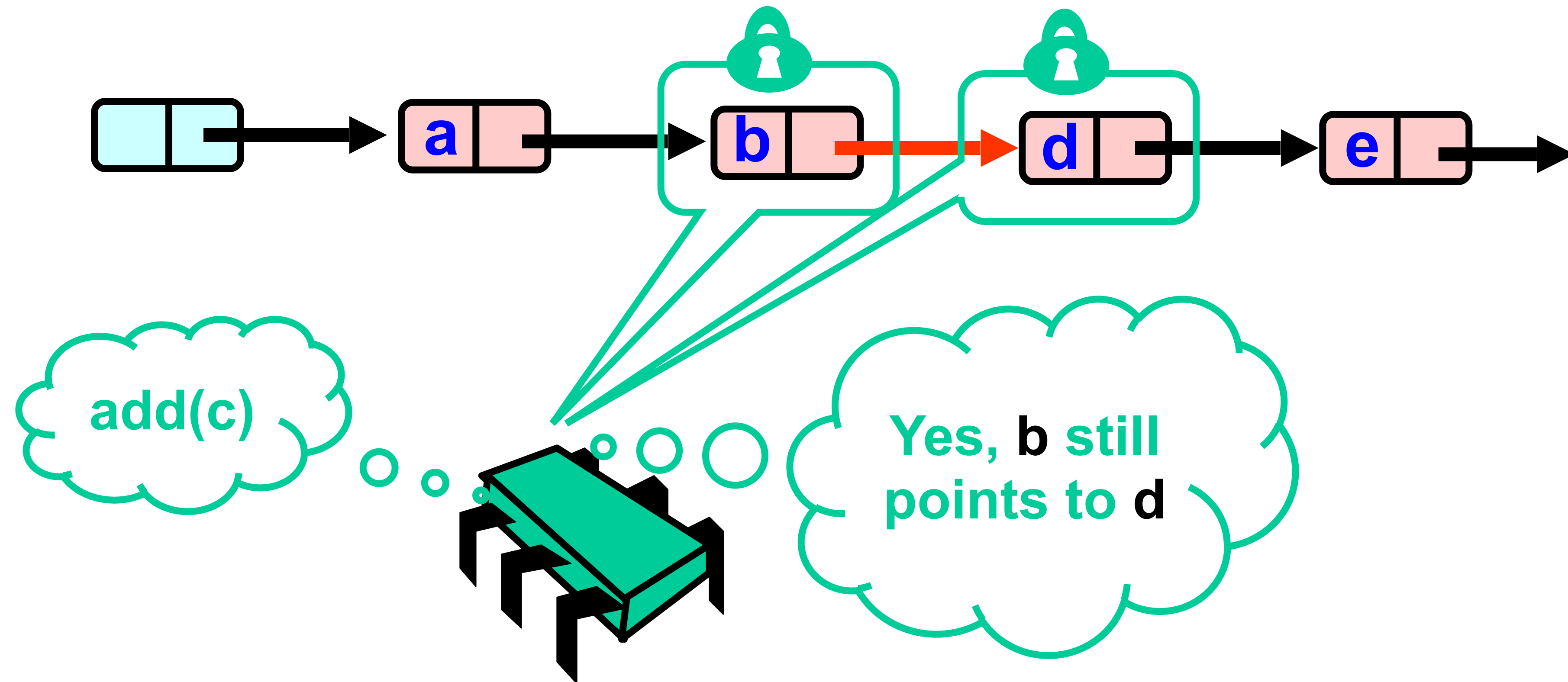
# What Else Could Go Wrong?



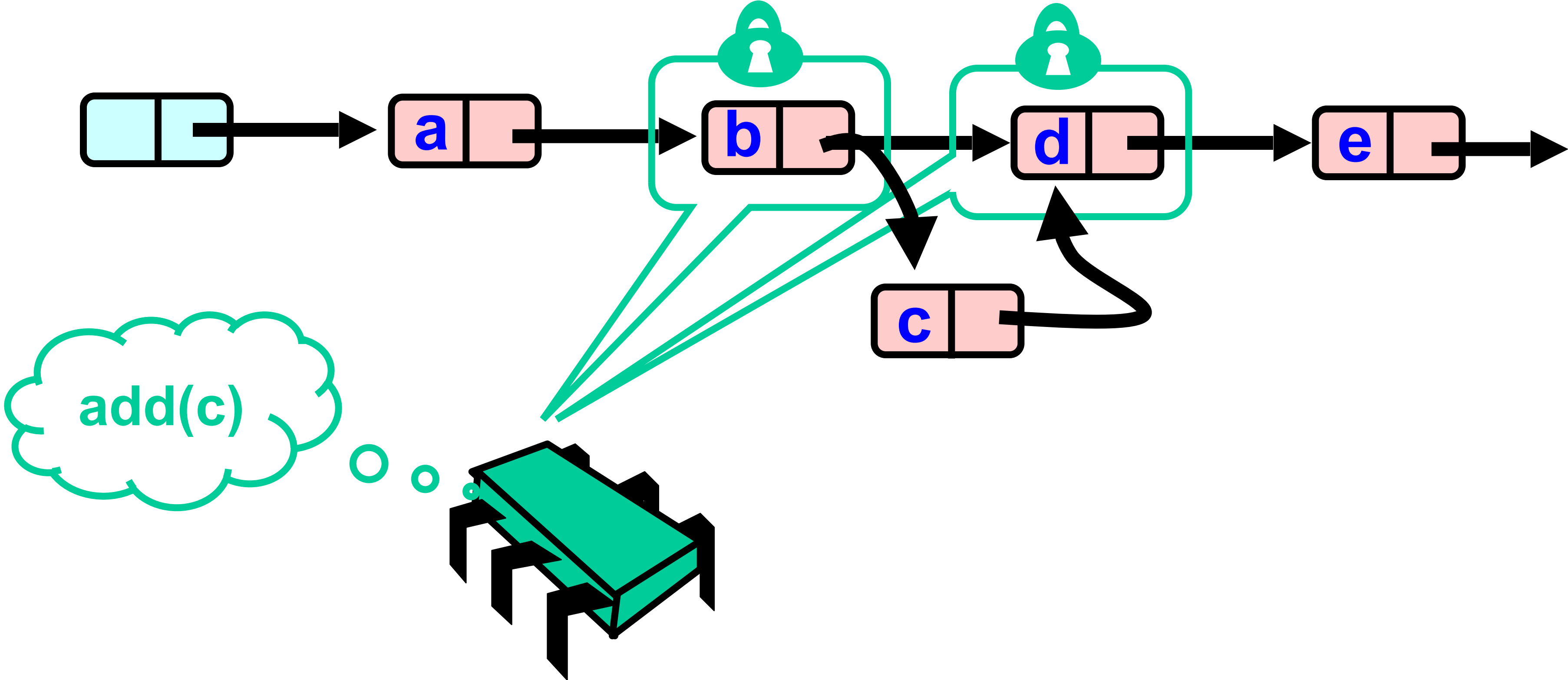
# What Else Could Go Wrong?



# Validate Part 2 (while holding locks)



# Optimistic: Linearization Point



# Same Abstraction Map

- $S(\text{head}) =$   
 $\{ x \mid \text{there exists } a \text{ such that}$ 
  - $a \text{ reachable from head and}$
  - $a.\text{item} = x$ $\}$

# Invariants

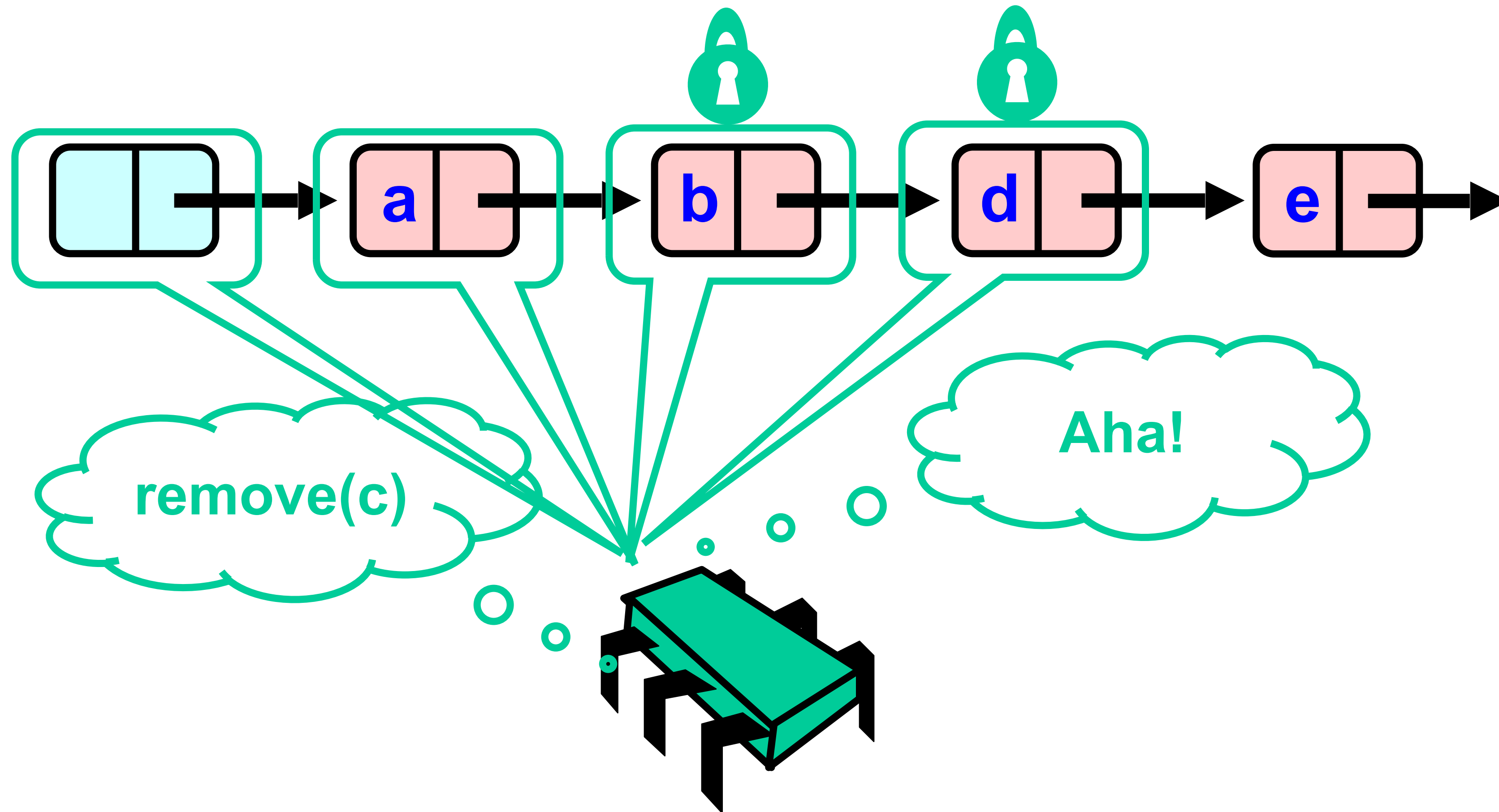
- Careful: we may traverse deleted nodes
- But we establish properties by
  - Validation
  - After we lock target nodes

# Removal of c

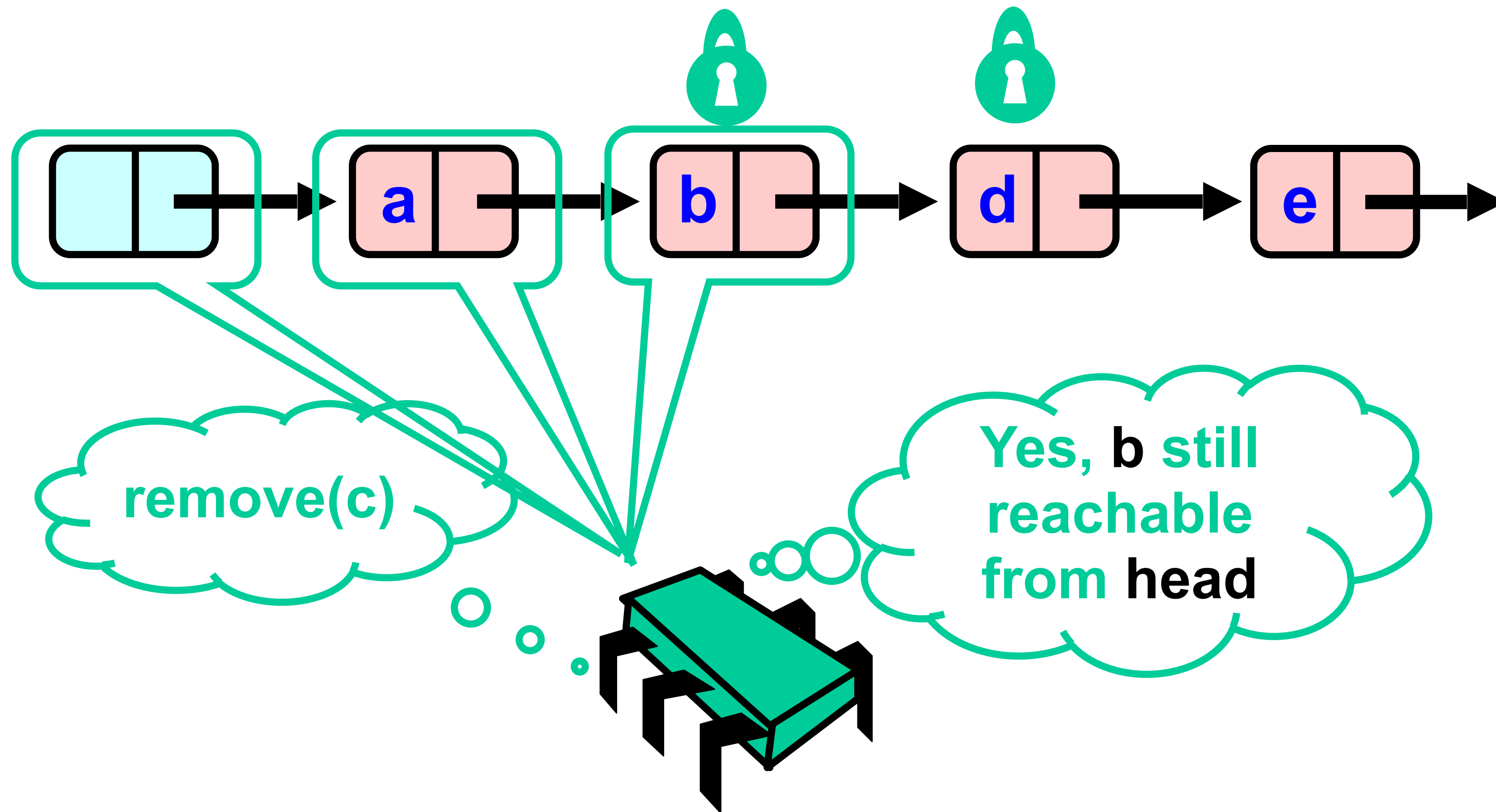
- If
  - Nodes b and c both locked
  - Node b still accessible
  - Node c still successor to b
- Then
  - Neither will be deleted
  - OK to delete and return true



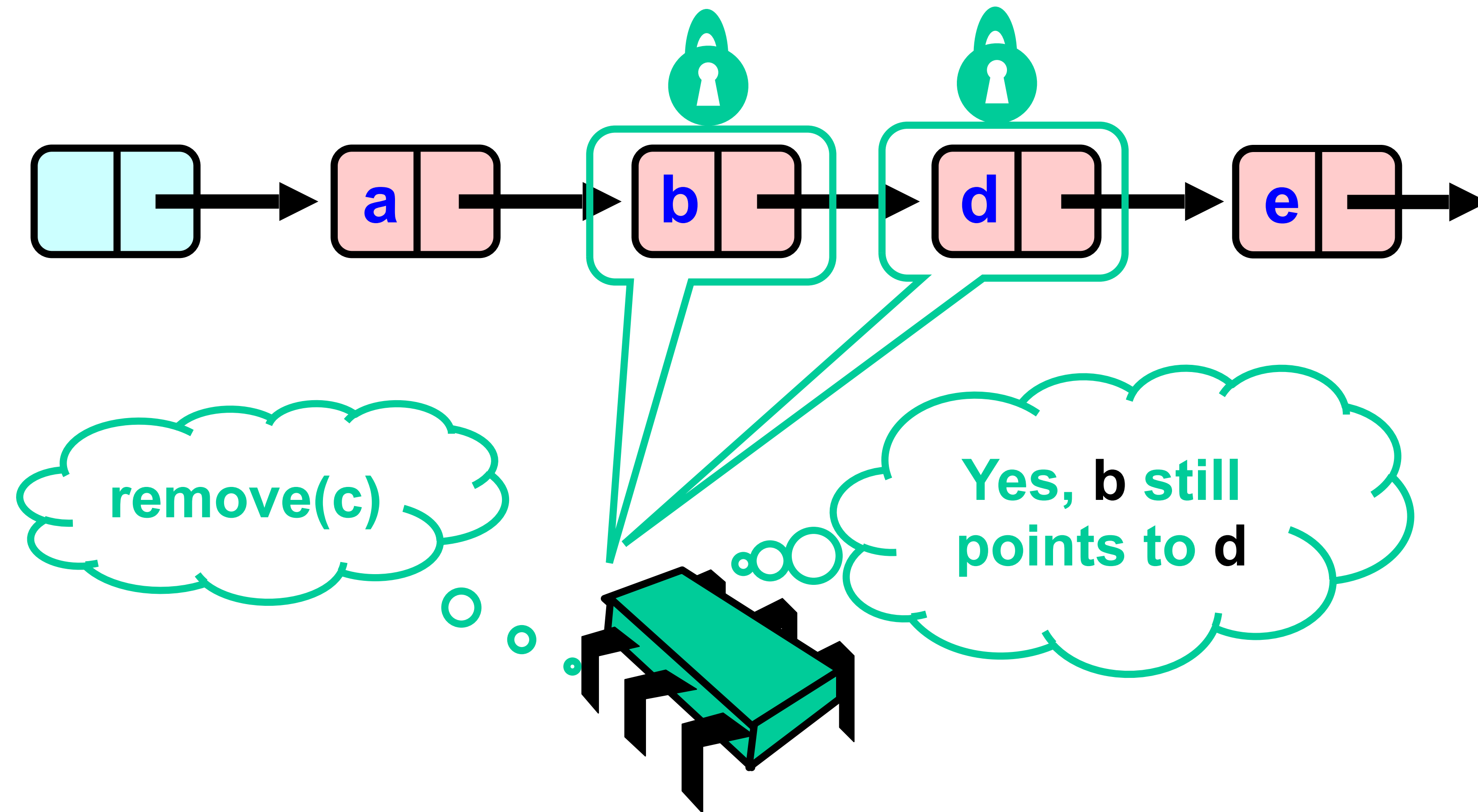
# Unsuccessful Remove



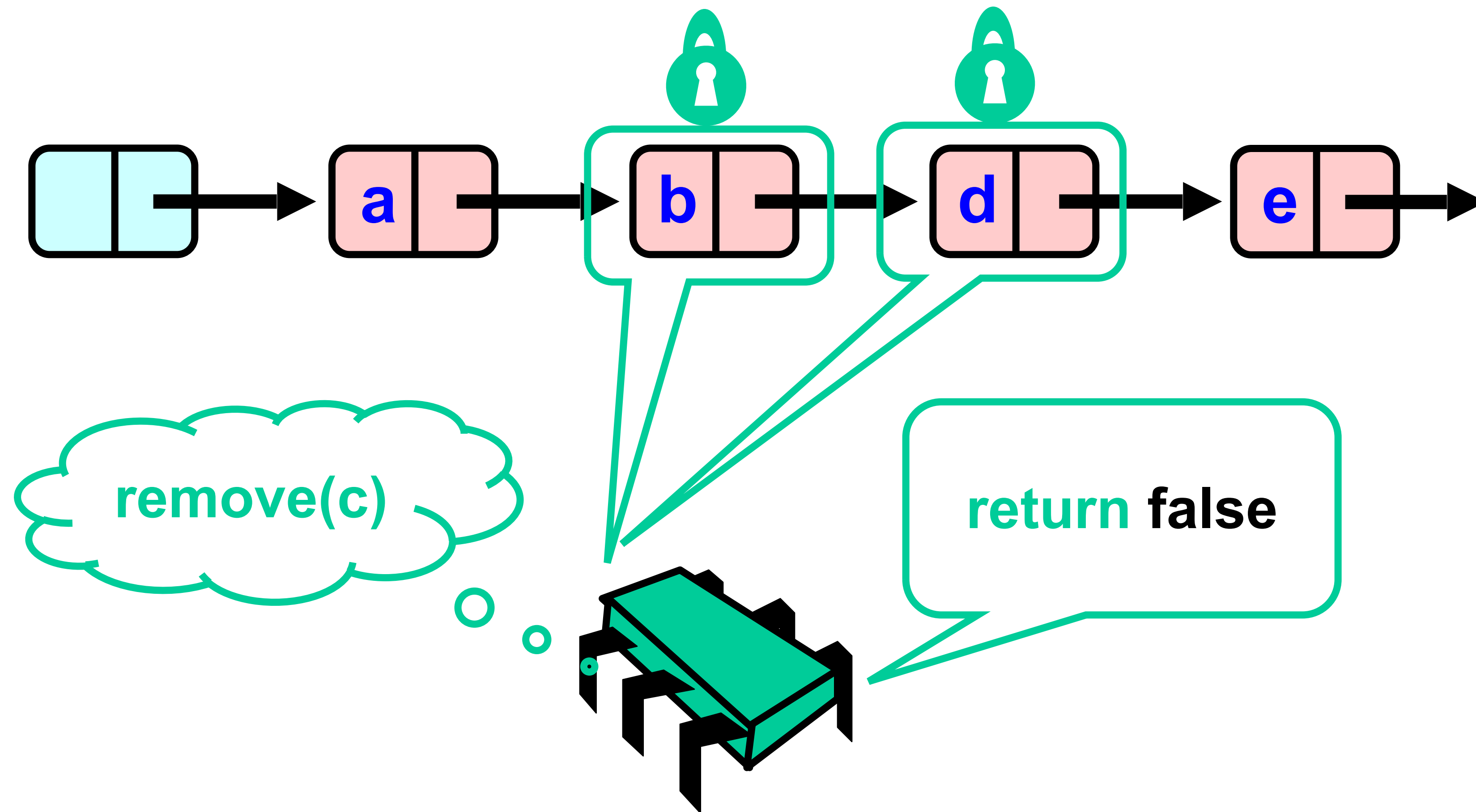
# Validate (1)



# Validate (2)



# OK Computer



# Correctness

- If
  - Nodes b and d both locked
  - Node b still accessible
  - Node d still successor to b
- Then
  - Neither will be deleted
  - No thread can add c after b
  - OK to return false

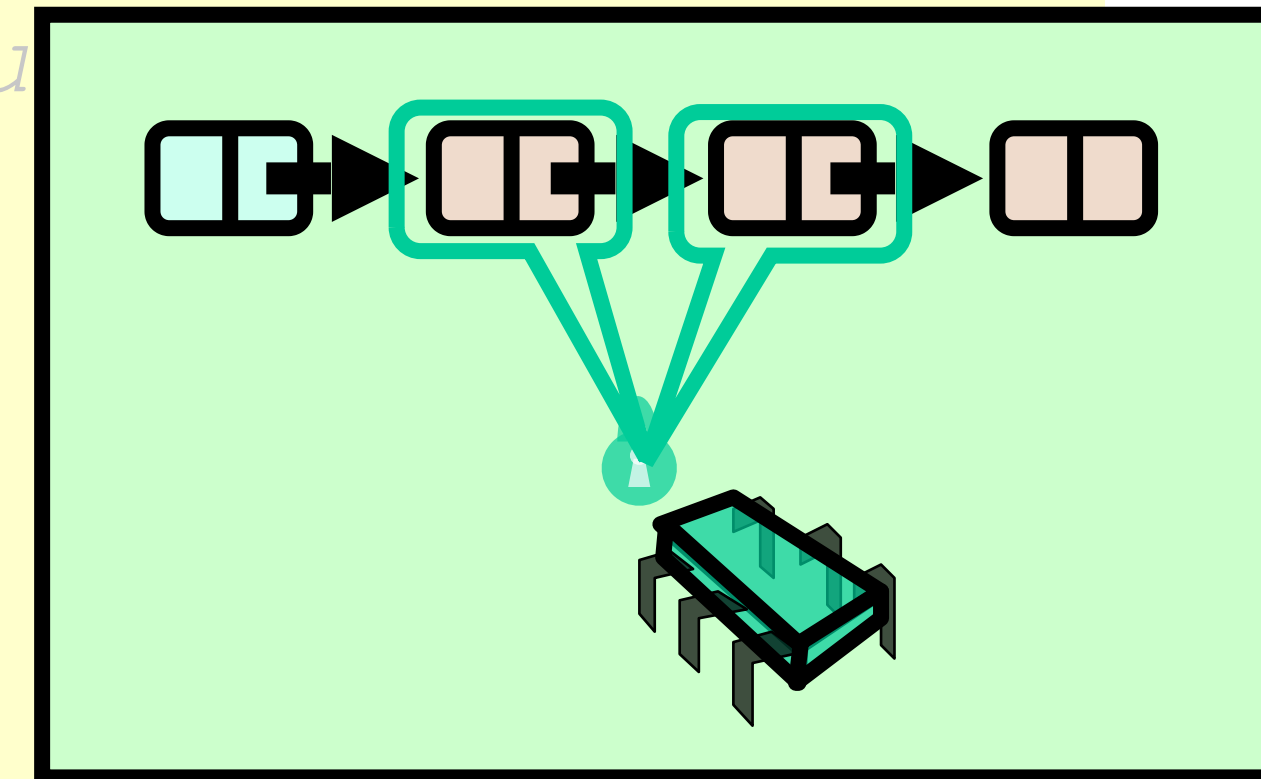
# Validation

```
def validate(pred: Node, curr: Node): Boolean = {  
  var entry = head  
  while (entry.key <= pred.key) {  
    // Checking for reference equality  
    if (entry eq pred) {  
      return pred.next eq curr  
    }  
    entry = entry.next  
  }  
  false  
}
```

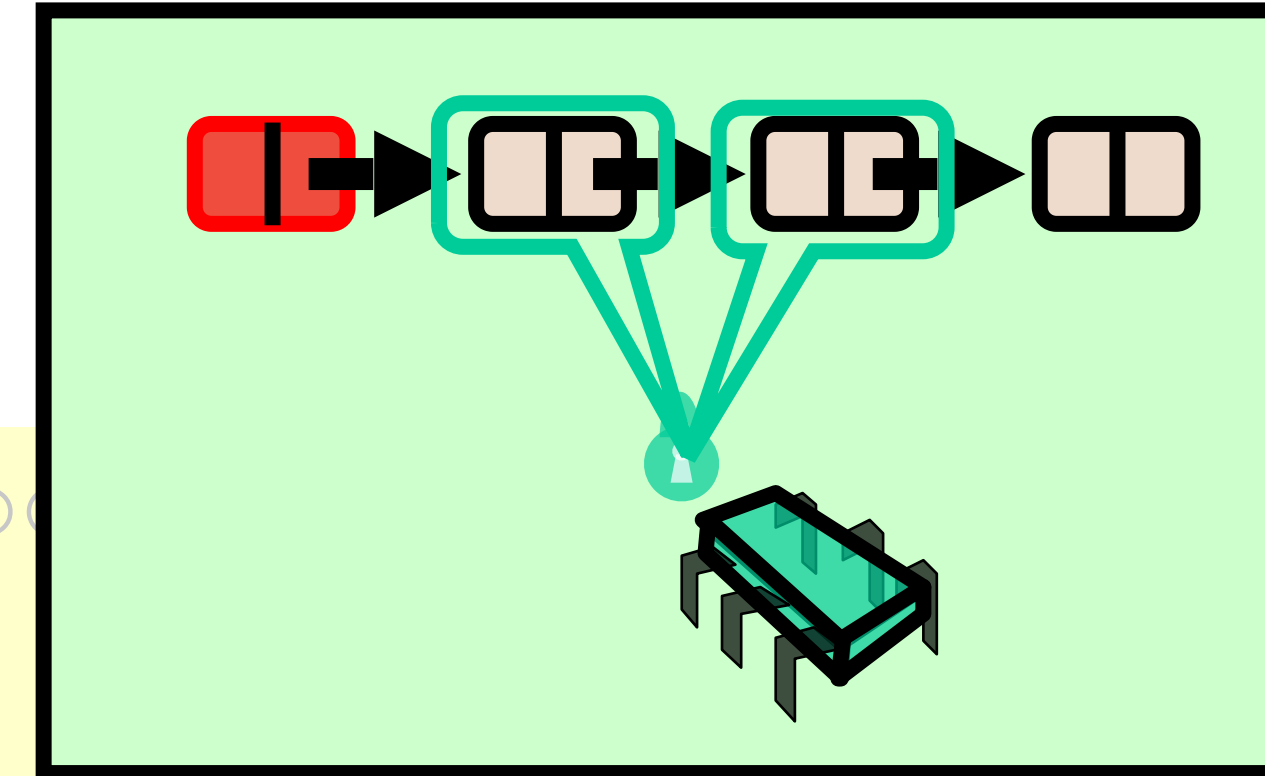
# Validation

```
def validate(pred: Node, curr: Node): Boolean = {  
  var entry = head  
  while (entry.key <= pred.key) {  
    // Checking for reference equality  
    if (entry eq pred) {  
      return pred.next eq curr  
    }  
    entry = entry.next  
  }  
  false  
}
```

**Predecessor &  
current nodes**



# Validation

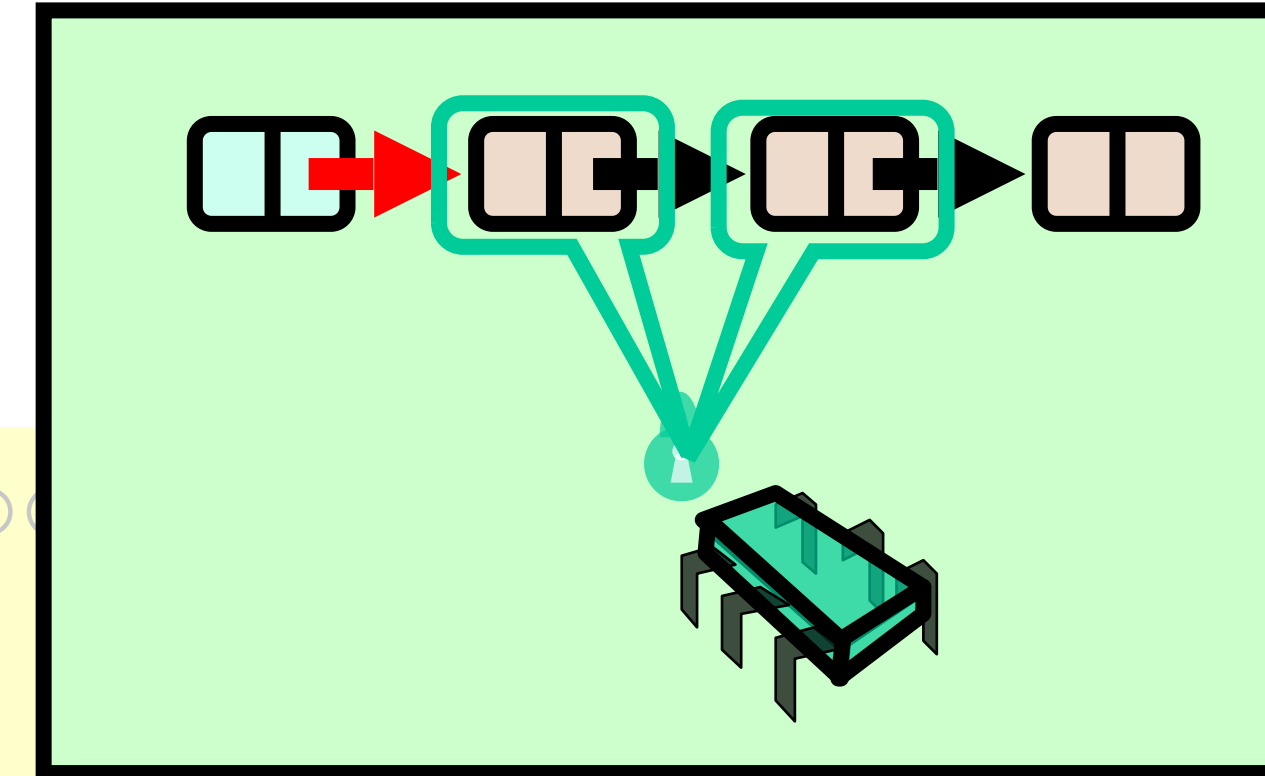


```
def validate(pred: Node, curr: Node): Boolean = {  
  var entry = head  
  while (entry.key <= pred.key) {  
    // Checking for reference equality  
    if (entry eq pred) {  
      return pred.next eq curr  
    }  
    entry = entry.next  
  }  
  false  
}
```

**Start at the  
beginning**



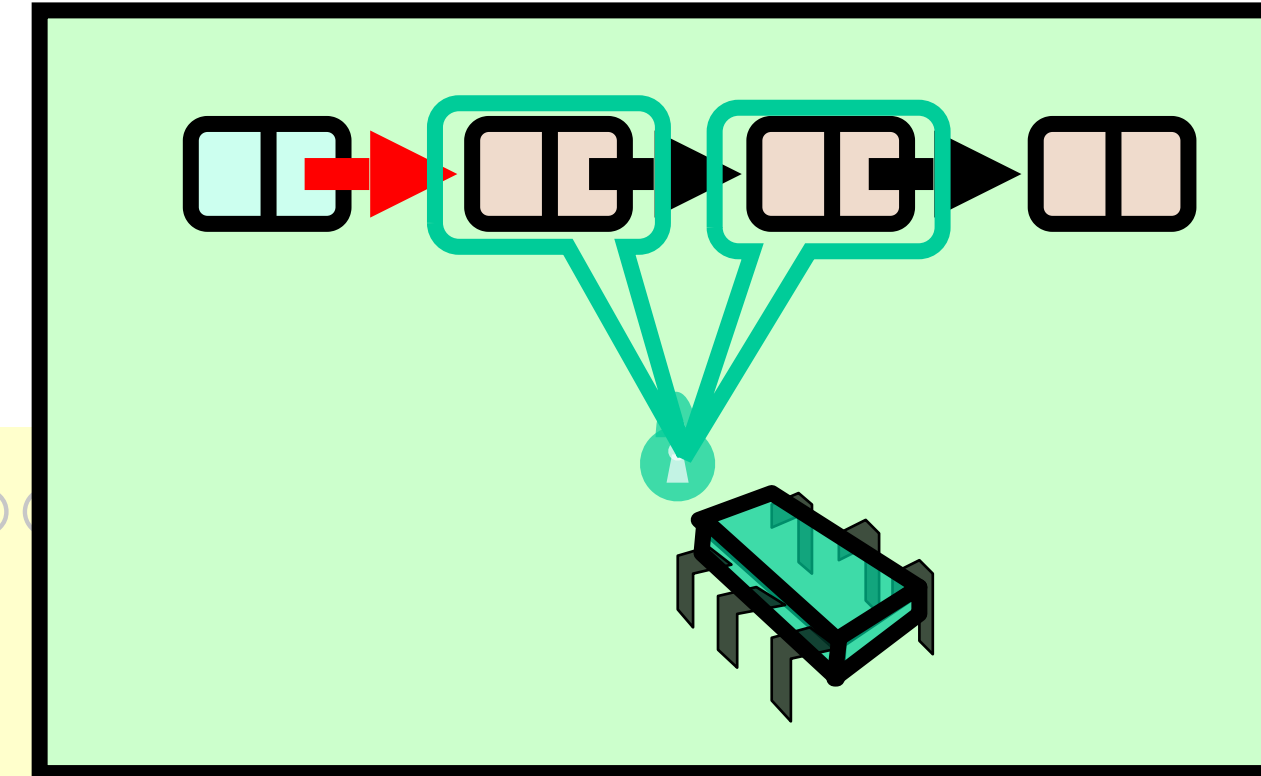
# Validation



```
def validate(pred: Node, curr: Node): Boolean = {  
  var entry = head  
  while (entry.key <= pred.key) {  
    // Checking for reference equality  
    if (entry eq pred) {  
      return pred.next eq curr  
    }  
    entry = entry.next  
  }  
  false  
}
```

**Search range of keys**

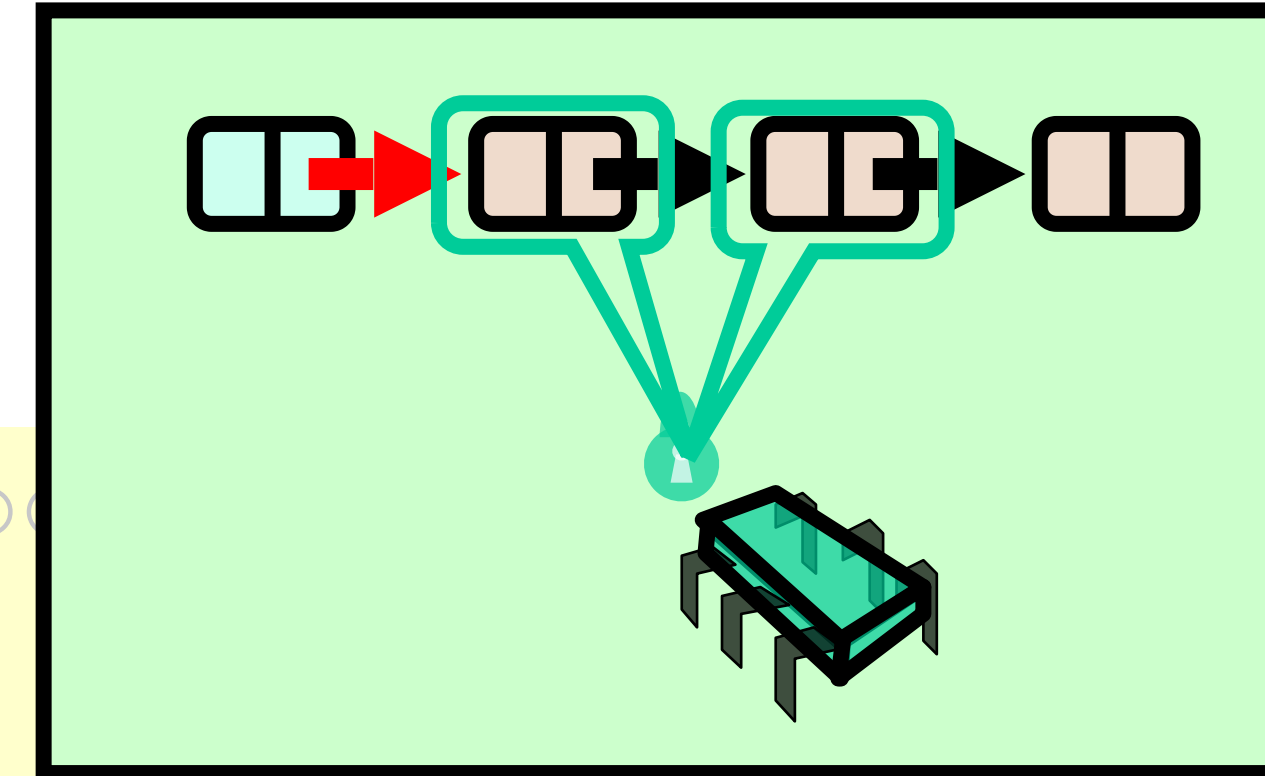
# Validation



```
def validate(pred: Node, curr: Node): Boolean = {  
  var entry = head  
  while (entry.key <= pred.key) {  
    // Checking for reference equality  
    if (entry eq pred) {  
      return pred.next eq curr  
    }  
    entry = entry.next  
  }  
  false  
}
```

**Predecessor reachable**

# Validation



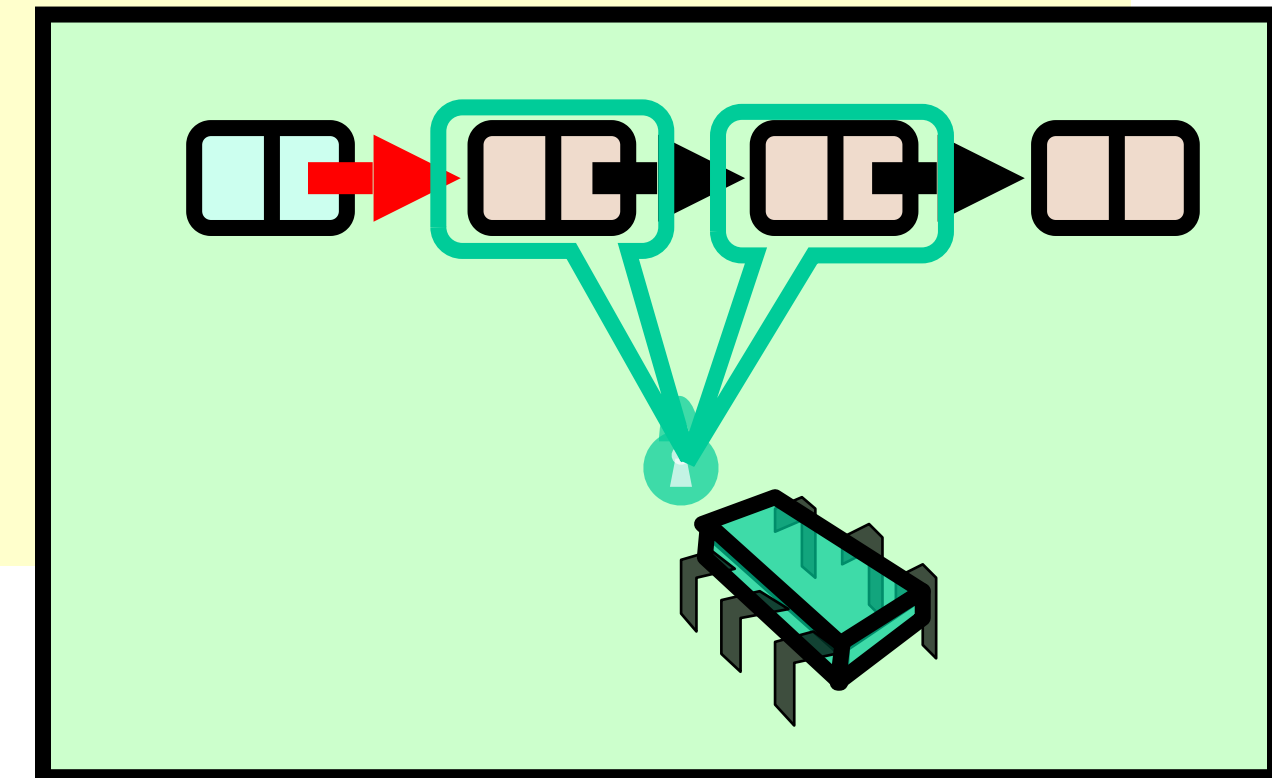
```
def validate(pred: Node, curr: Node): Boolean = {  
  var entry = head  
  while (entry.key <= pred.key) {  
    // Checking for reference equality  
    if (entry eq pred) {  
      return pred.next eq curr  
    }  
    entry = entry.next  
  }  
  false  
}
```

**Is current node next?**

# Validation

Otherwise move on

```
def validate(pred: Node, curr: Node): Boolean = {  
  var entry = head  
  while (entry.key <= pred.key) {  
    // Checking for reference equality  
    if (entry eq pred) {  
      return pred.next eq curr  
    }  
    entry = entry.next  
  }  
  false  
}
```

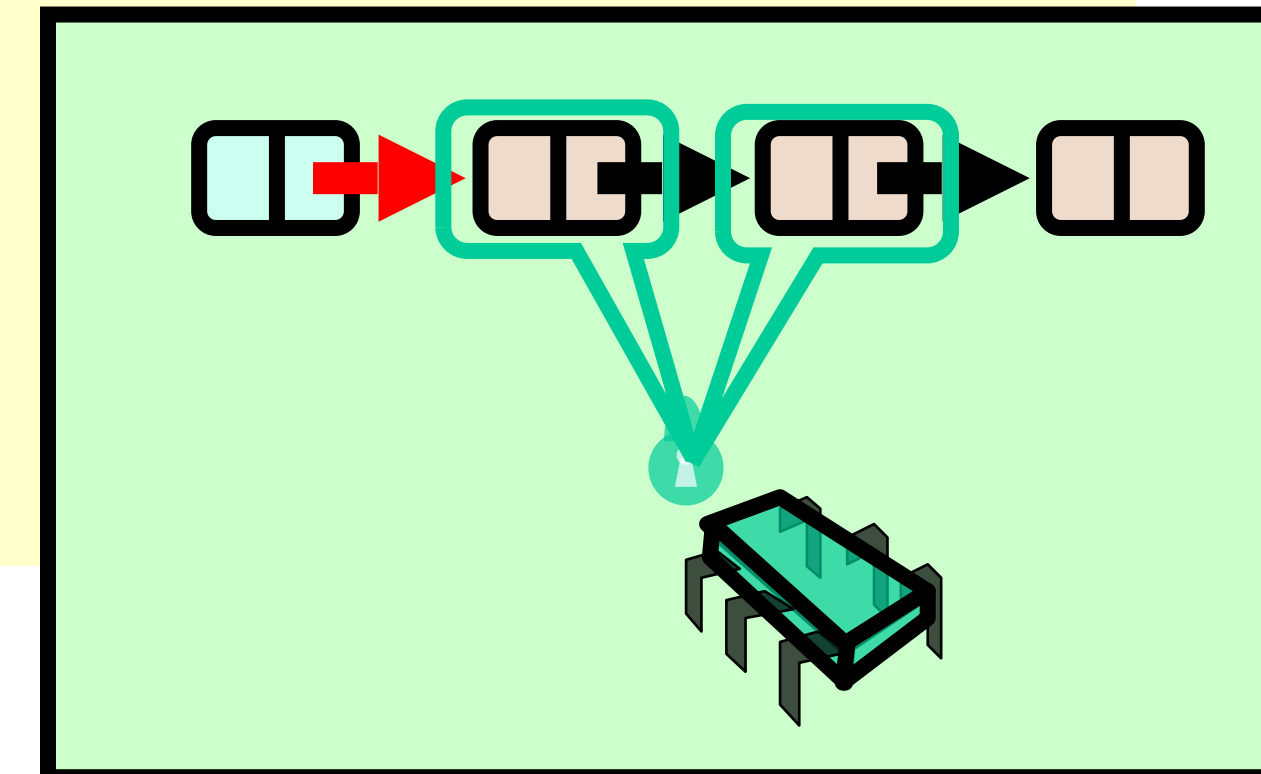


# Validation

## Predecessor not reachable

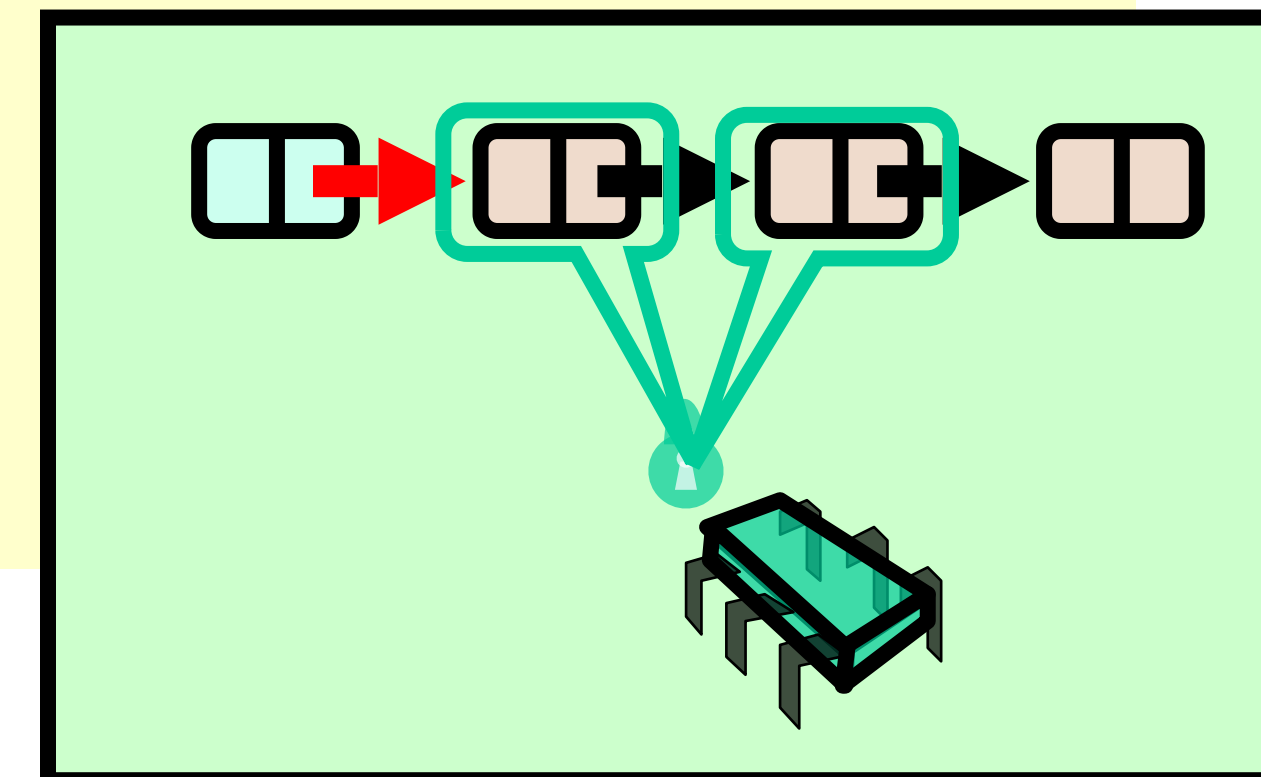
```
def validate(pred: Node, curr: Node): Boolean = {  
  var entry = head  
  while (entry.key <= pred.key) {  
    // Checking for reference equality  
    if (entry eq pred) {  
      return pred.next eq curr  
    }  
    entry = entry.next  
  }  
}
```

**false**



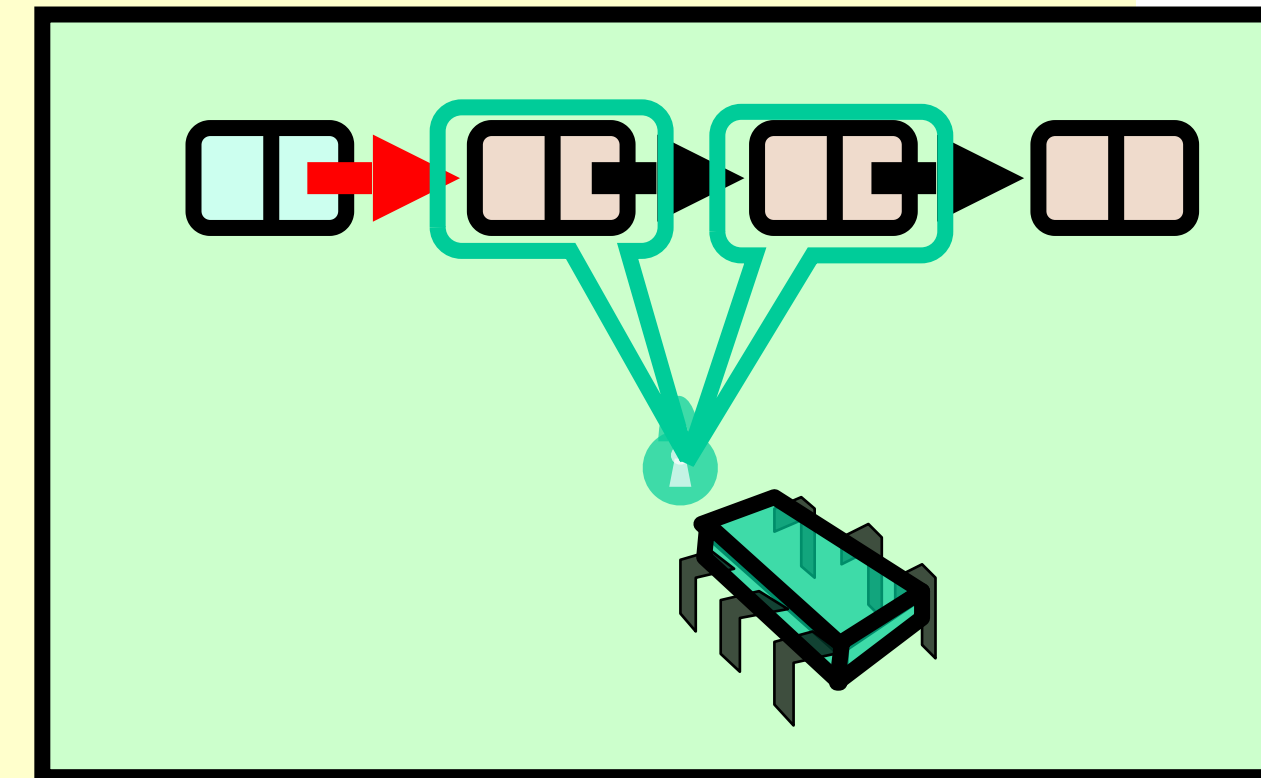
# Remove: searching

```
def remove(item: T): Boolean = {  
  val key = item.hashCode()  
  while (true) {  
    var pred = this.head  
    var curr = pred.next  
    while (curr.key < key) {  
      pred = curr  
      curr = curr.next  
    }  
    ...  
  }  
}
```



# Remove: searching

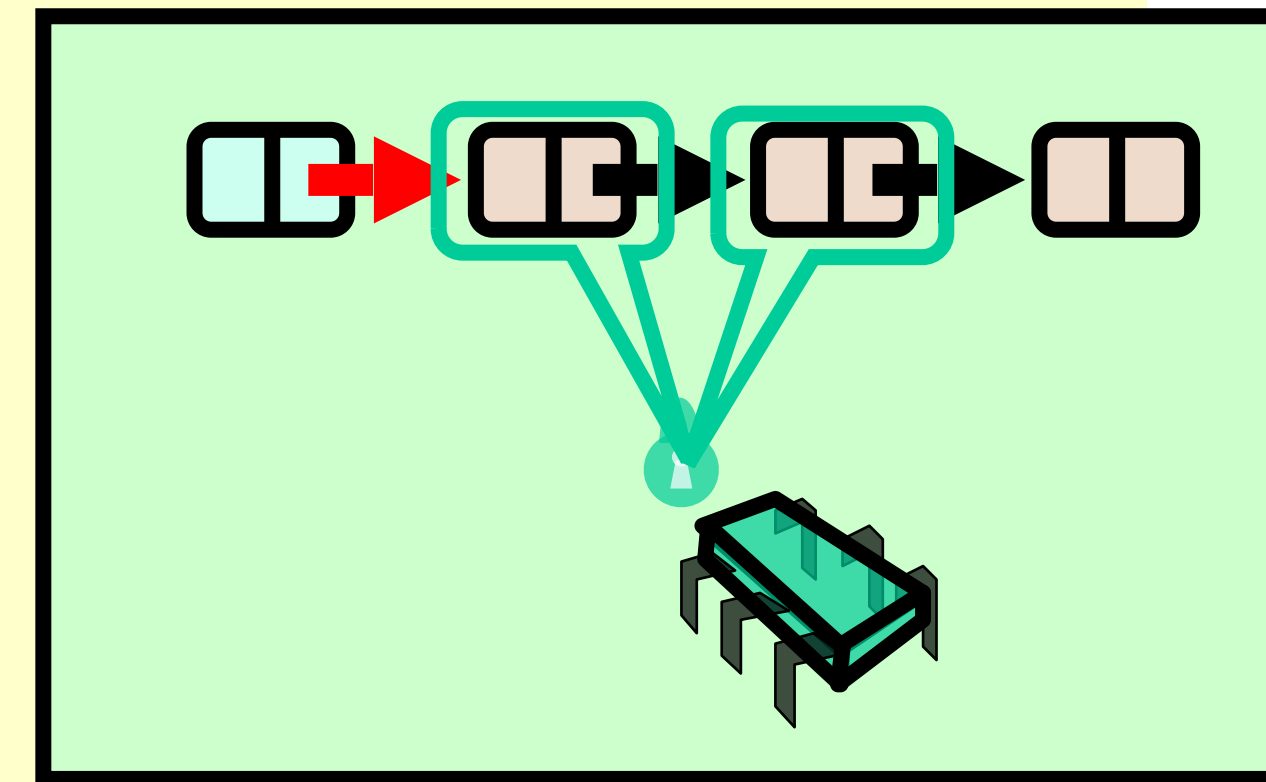
```
def remove(item: T): Boolean = {  
  val key = item.hashCode()  
  while (true) {  
    var pred = this.head  
    var curr = pred.next  
    while (curr.key < key) {  
      pred = curr  
      curr = curr.next  
    }  
    ...  
  }  
}
```



**Search key**

# Remove: searching

```
def remove(item: T): Boolean = {  
  val key = item.hashCode()  
  while (true) {  
    var pred = this.head  
    var curr = pred.next  
    while (curr.key < key) {  
      pred = curr  
      curr = curr.next  
    }  
    ...  
  }  
}
```



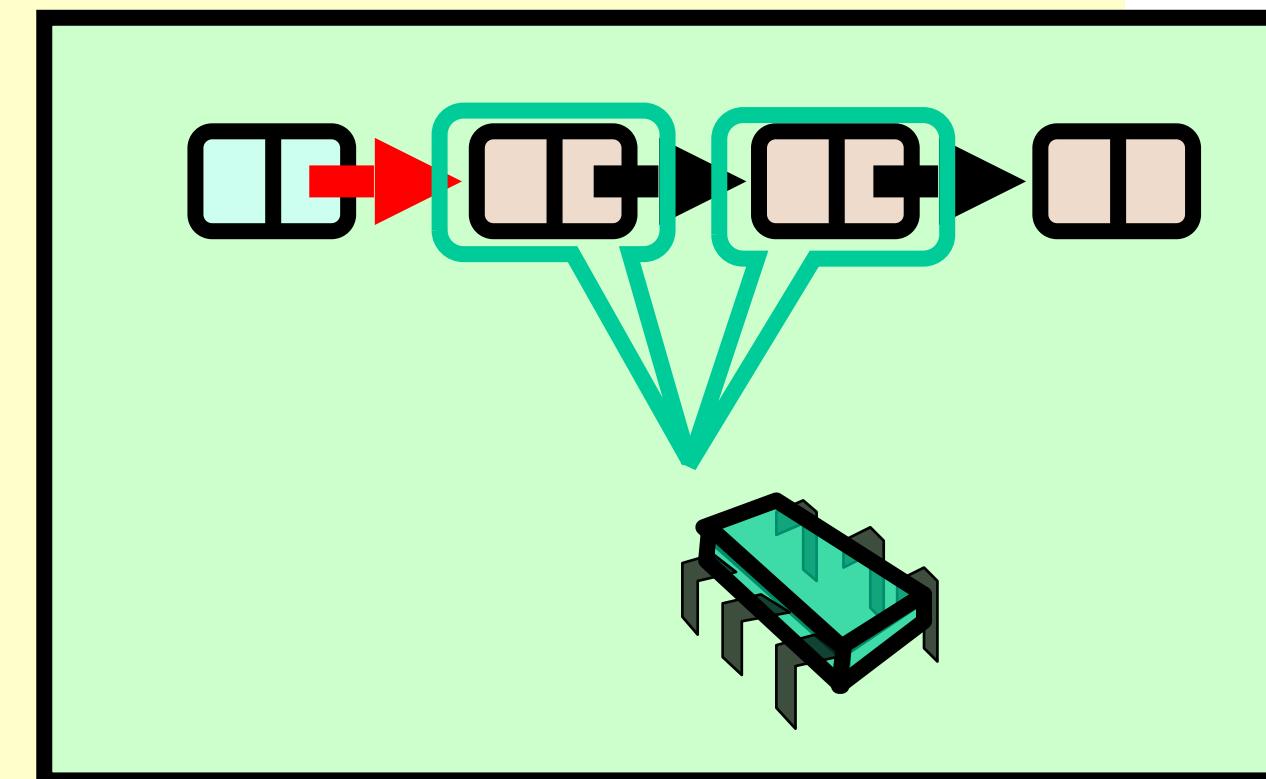
**Loop until no synchronization conflict  
(see the code further)**



# Remove: searching

```
def remove(item: T): Boolean = {  
  val key = item.hashCode()  
  while (true) {  
    var pred = this.head  
    var curr = pred.next  
    while (curr.key < key) {  
      pred = curr  
      curr = curr.next  
    }  
    ...  
  }  
}
```

**var** pred = this.head  
**var** curr = pred.next

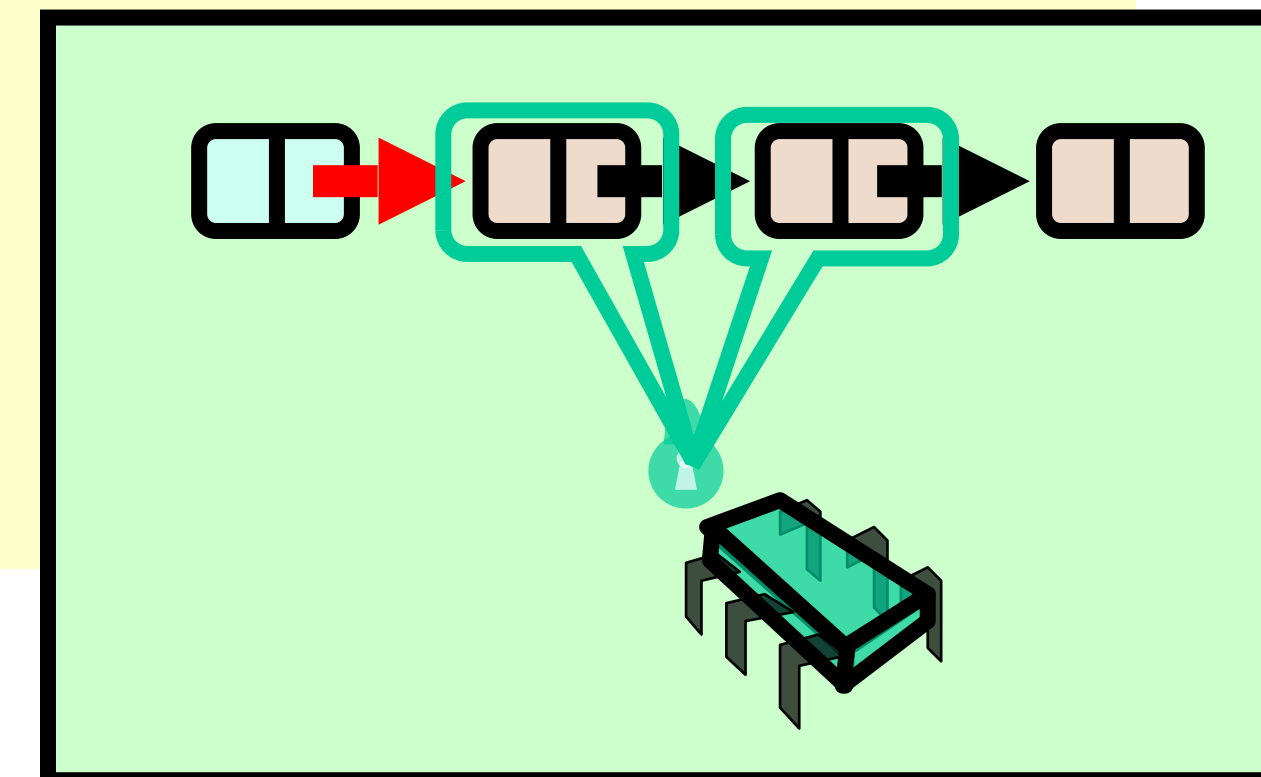


**Examine predecessor and current nodes**

# Remove: searching

```
def remove(item: T): Boolean = {  
  val key = item.hashCode()  
  while (true) {  
    var pred = this.head  
    var curr = pred.next  
    while (curr.key < key) {  
      pred = curr  
      curr = curr.next  
    }  
    ...  
  }  
}
```

**Search by key**



# On Exit from While-True-Loop

- If item is present
  - curr holds item
  - pred just before curr
- If item is absent
  - curr has first higher key
  - pred just before curr
- Assuming no synchronization problems

# Remove Method

```
pred.lock(); curr.lock()
try {
    if (validate(pred, curr)) {
        if (curr.key == key) { // present in list
            pred.next = curr.next
            return true
        } else { // not present in list
            return false
        }
    }
} finally { // always unlock
    pred.unlock(); curr.unlock()
}
```

# Remove Method

```
pred.lock(); curr.lock()
```

```
try {  
    if (validate(pred, curr)) {  
        if (curr.key == key) {  
            pred.next = curr.next  
            return true  
        } else {  
            return false  
        }  
    }  
} finally {  
    pred.unlock(); curr.unlock()  
}
```

**Lock both nodes**

# Remove Method

```
pred.lock(); curr.lock()
try {
    if (!validate(pred, curr)) {
        if (curr.key == key) {
            pred.next = curr.next
            return true
        } else {
            return false
        }
    }
} finally { // always unlock
    pred.unlock(); curr.unlock()
}
```

**Always unlock**

# Remove Method

```
pred.lock(); curr.lock()
try {
    if (validate(pred, curr)) {
        if (curr.key == key) {
            pred.next = curr.next
            return true
        } else {
            return false
        }
    }
} finally {
    pred.unlock(); curr.unlock()
}
```

**Check for synchronization conflicts**

# Remove Method

```
pred.lock(); curr.lock()
try {
    if (validate(pred, curr)) {
        if (curr.key == key) {
            pred.next = curr.next
            return true
        } else {
            return false
        }
    }
} finally {
    pred.unlock(); curr.unlock()
}
```

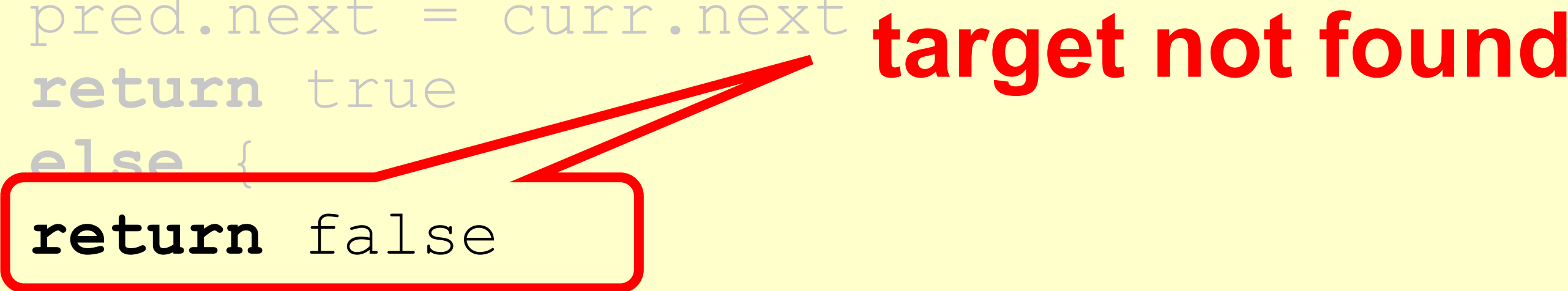
**target found, remove  
node**



# Remove Method

```
pred.lock(); curr.lock()
try {
    if (validate(pred, curr)) {
        if (curr.key == key) {
            pred.next = curr.next
            return true
        } else {
            return false
        }
    }
} finally {
    pred.unlock(); curr.unlock()
}
```

**target not found**



# Optimistic List

- Limited hot-spots
  - Targets of `add()`, `remove()`, `contains()`
  - No contention on traversals
- Moreover
  - Traversals are wait-free
  - Food for thought ...

# So Far, So Good

- Much less lock acquisition/release
  - Performance
  - Concurrency
- Problems
  - Need to traverse list twice
  - `contains ()` method acquires locks

# Evaluation

- Optimistic is effective if
  - cost of scanning twice without locks is less than
  - cost of scanning once with locks
- Drawback
  - `contains ()` acquires locks
  - 90% of calls in many apps

# Demo: Benchmarking Optimistic Lists

<A good place to pause>

# Lazy List

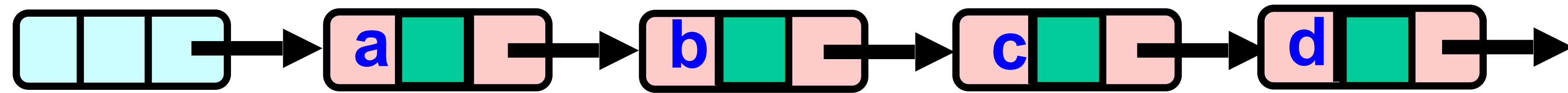
- Like optimistic, except
  - Scan once
  - `contains(x)` never locks ...
- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”

# Lazy List

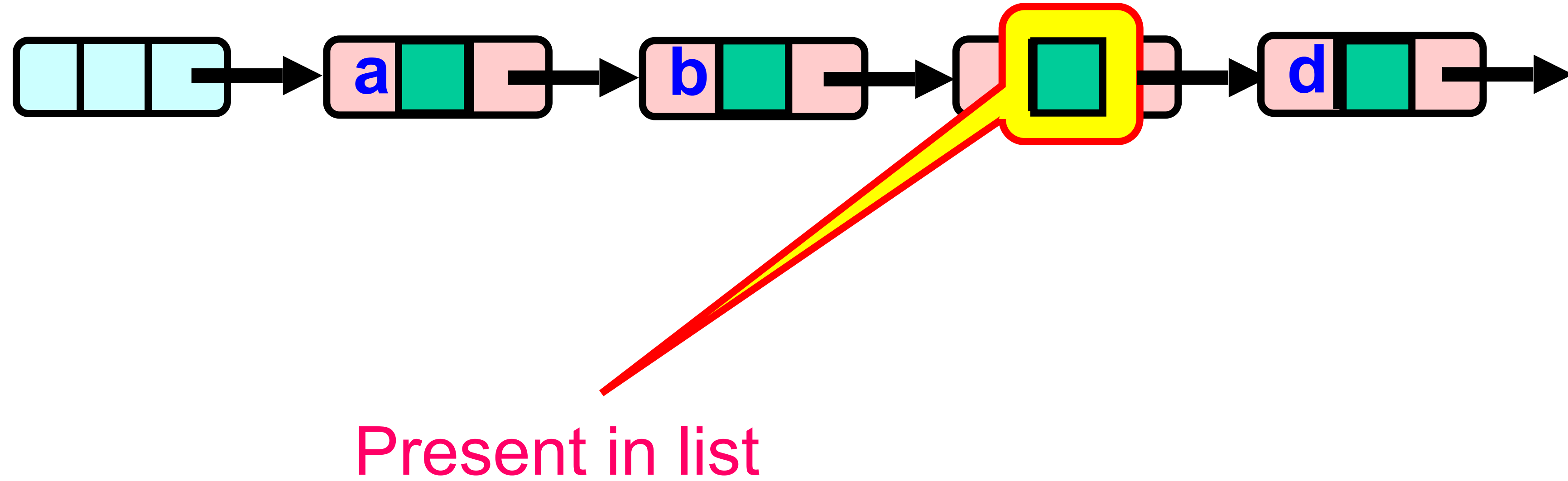
- **remove ()**
  - Scans list (as before)
  - Locks predecessor & current (as before)
- **Logical delete**
  - Marks current node as removed (new!)
- **Physical delete**
  - Redirects predecessor's next (as before)



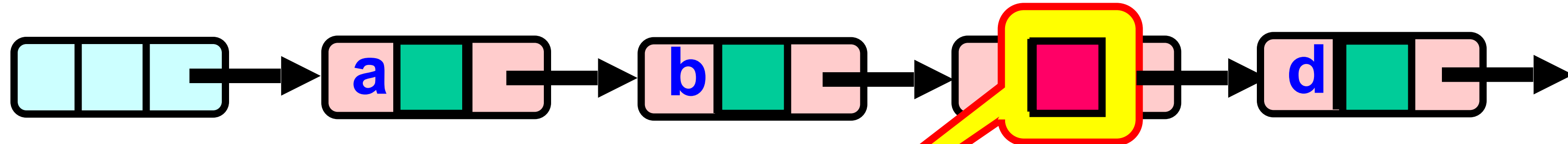
# Lazy Removal



# Lazy Removal

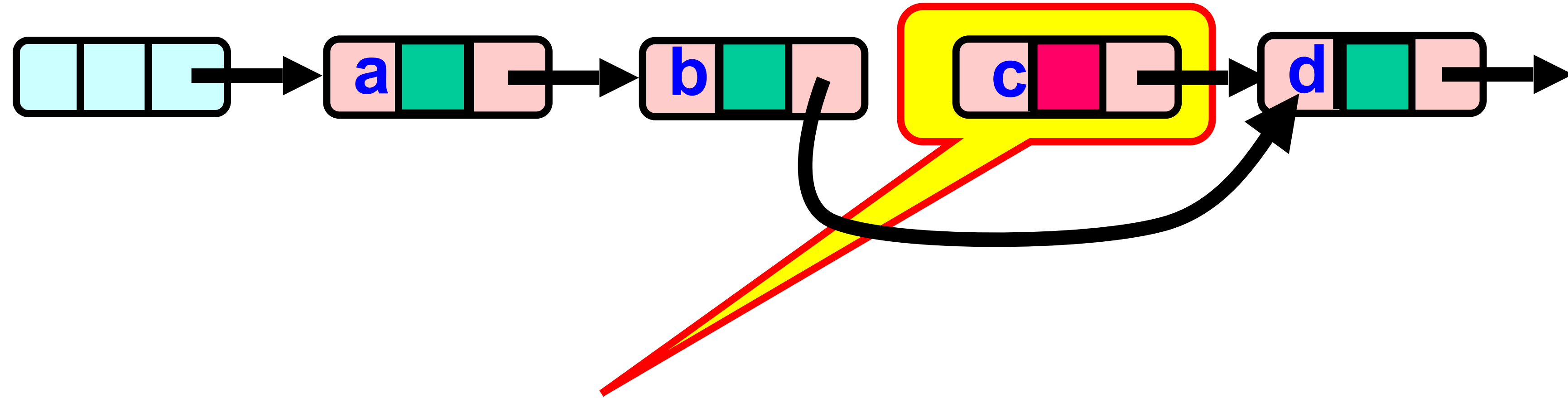


# Lazy Removal



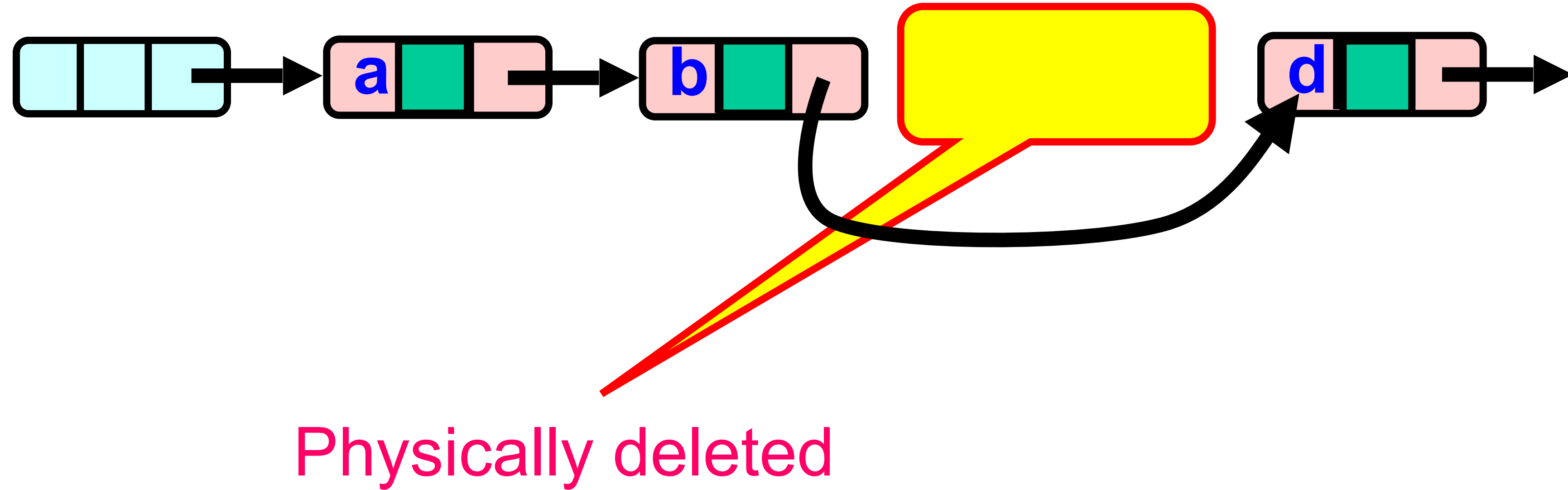
Logically deleted

# Lazy Removal



Physically deleted

# Lazy Removal



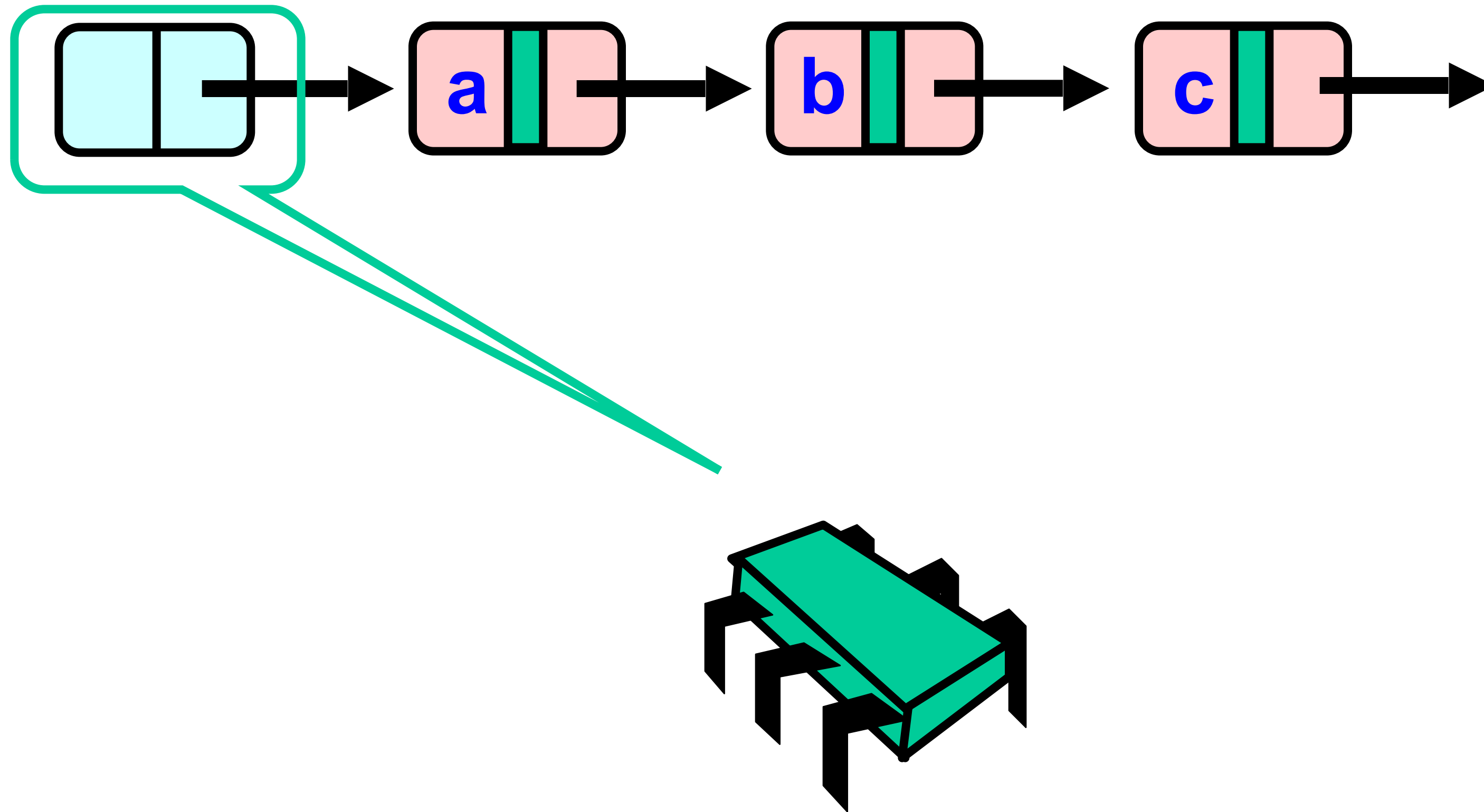
# Lazy List

- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn't slow down other method calls ...
- Must still lock pred and curr nodes.

# Validation

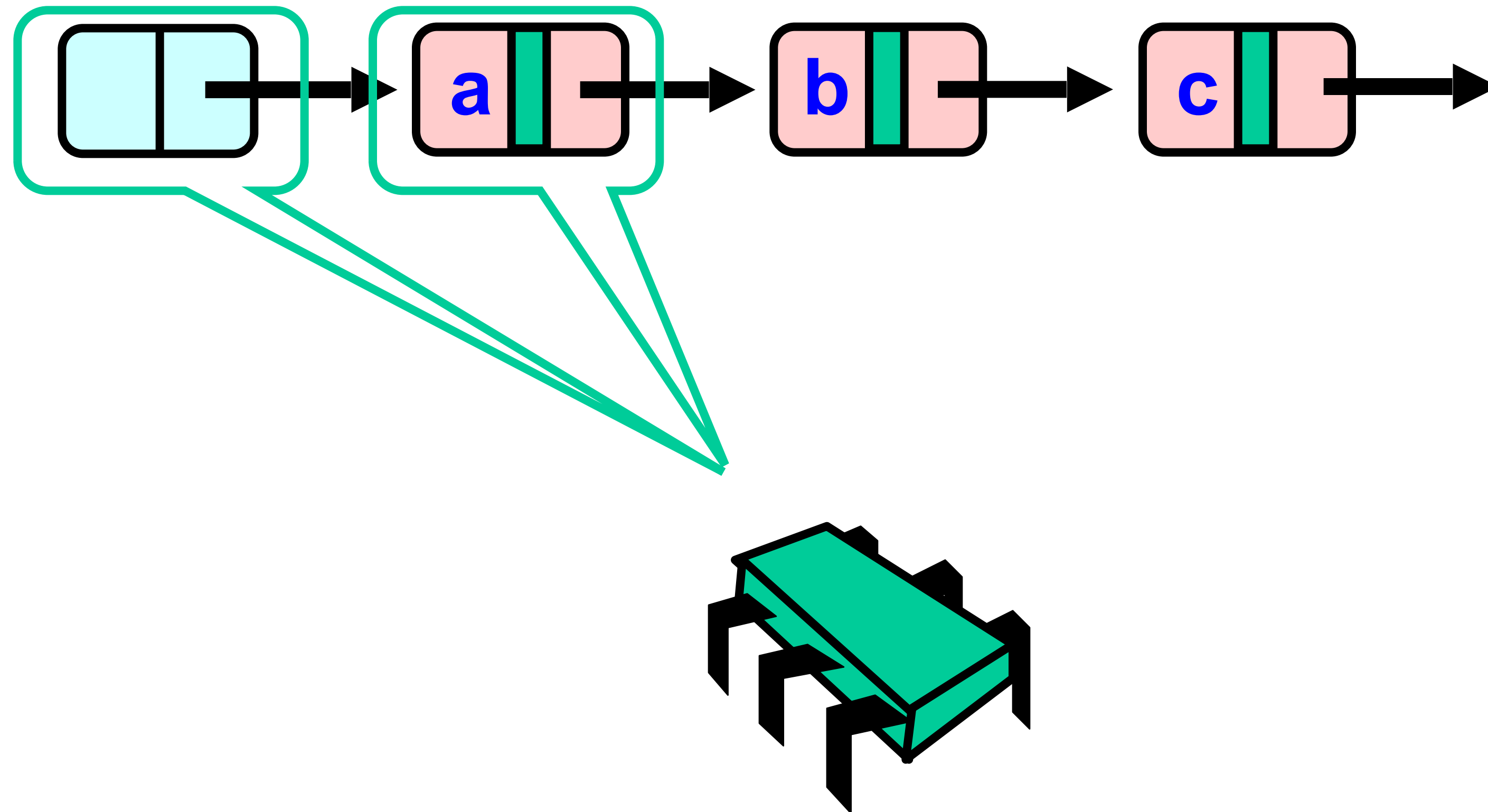
- No need to rescan list!
- Check that pred is not marked
- Check that curr is not marked
- Check that pred points to curr

# Business as Usual

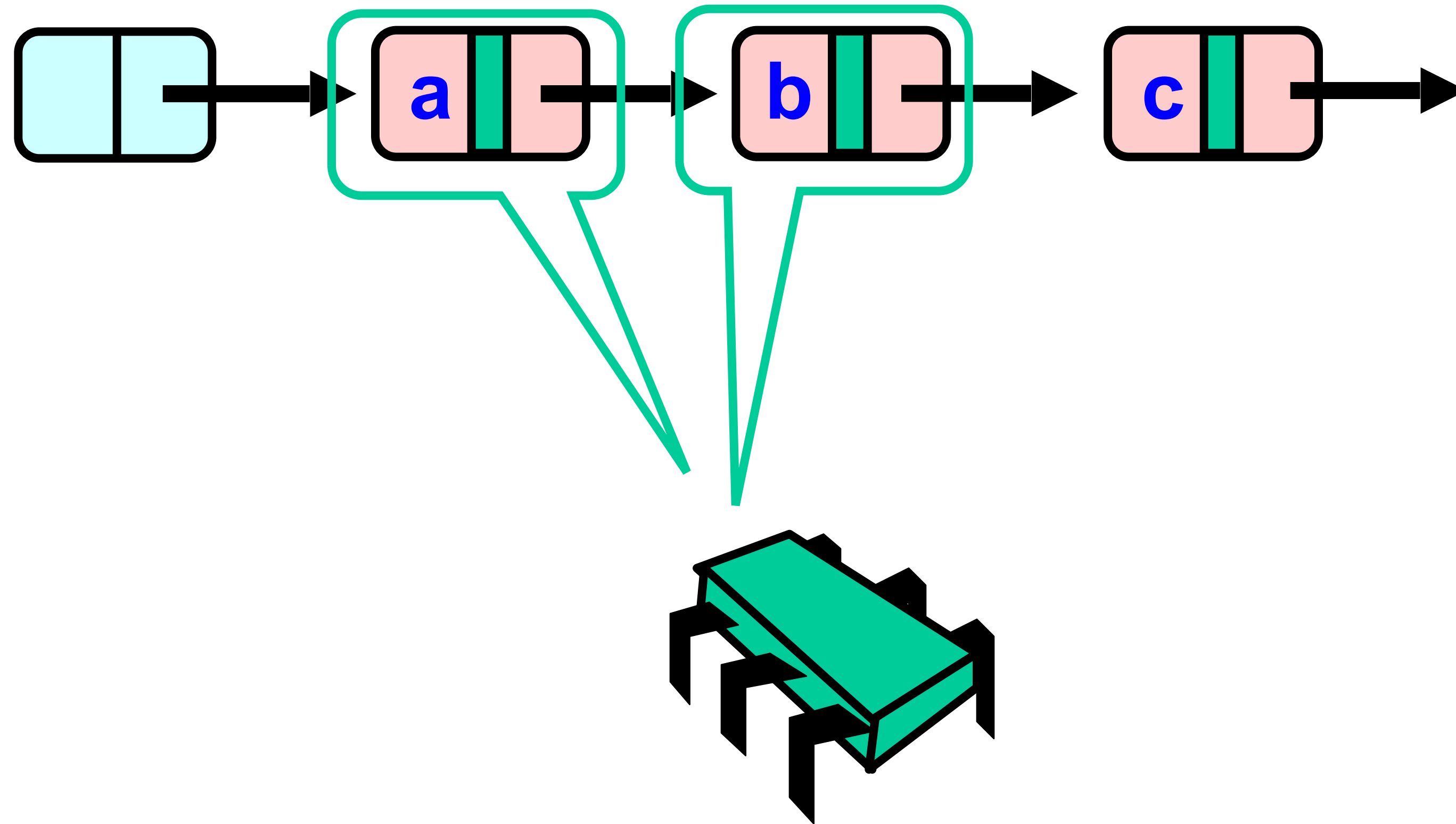




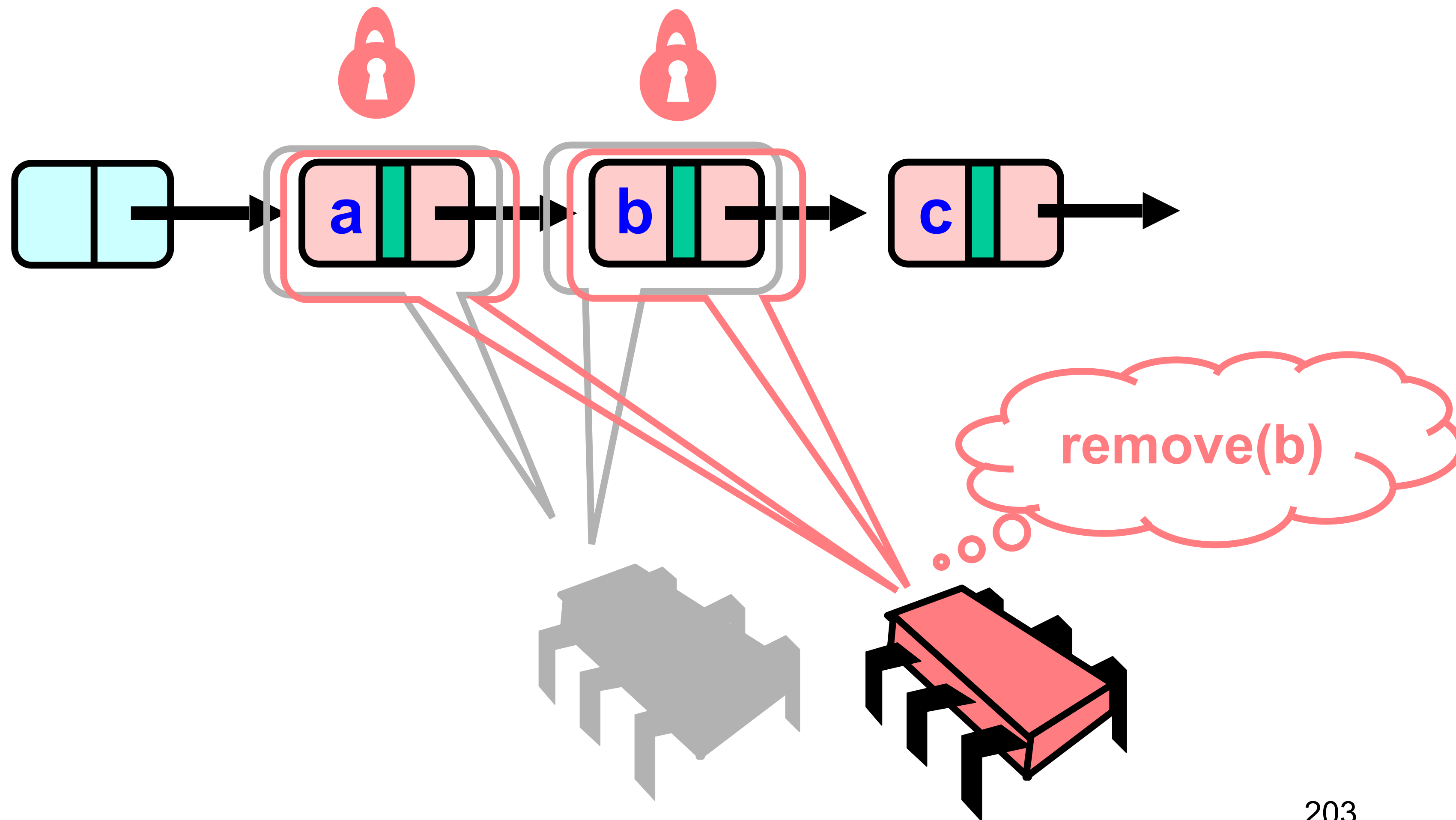
# Business as Usual



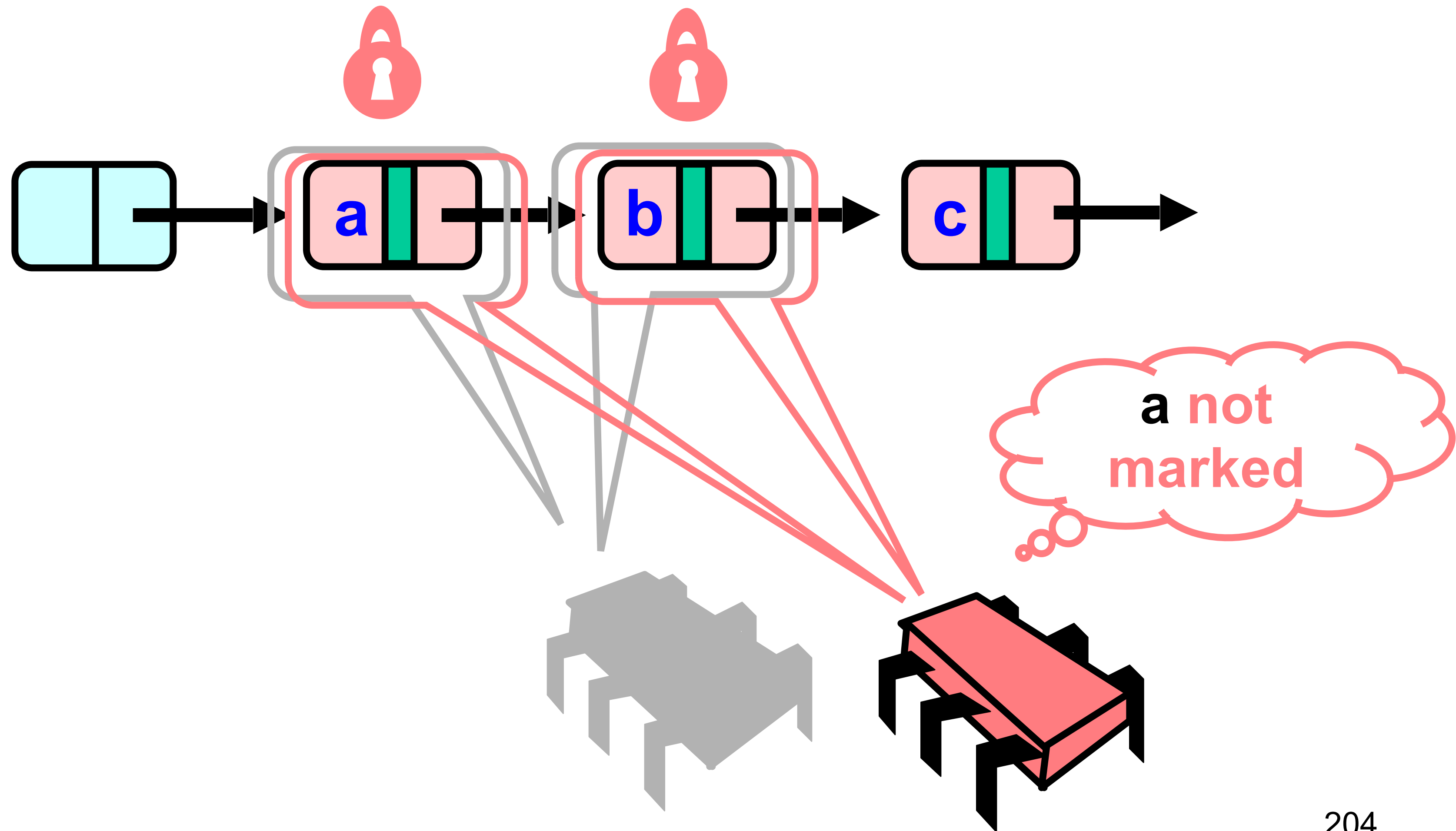
# Business as Usual



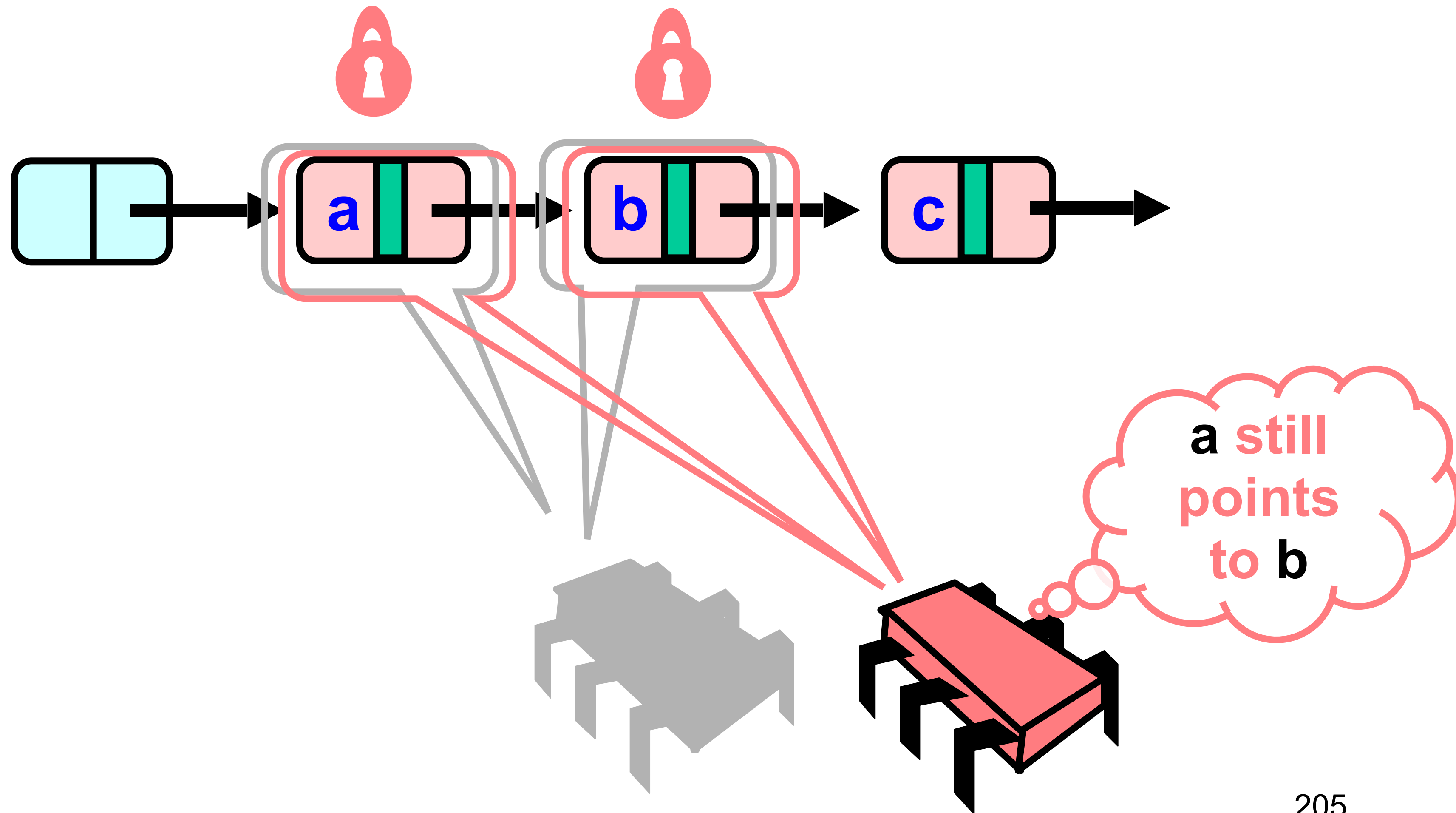
# Business as Usual



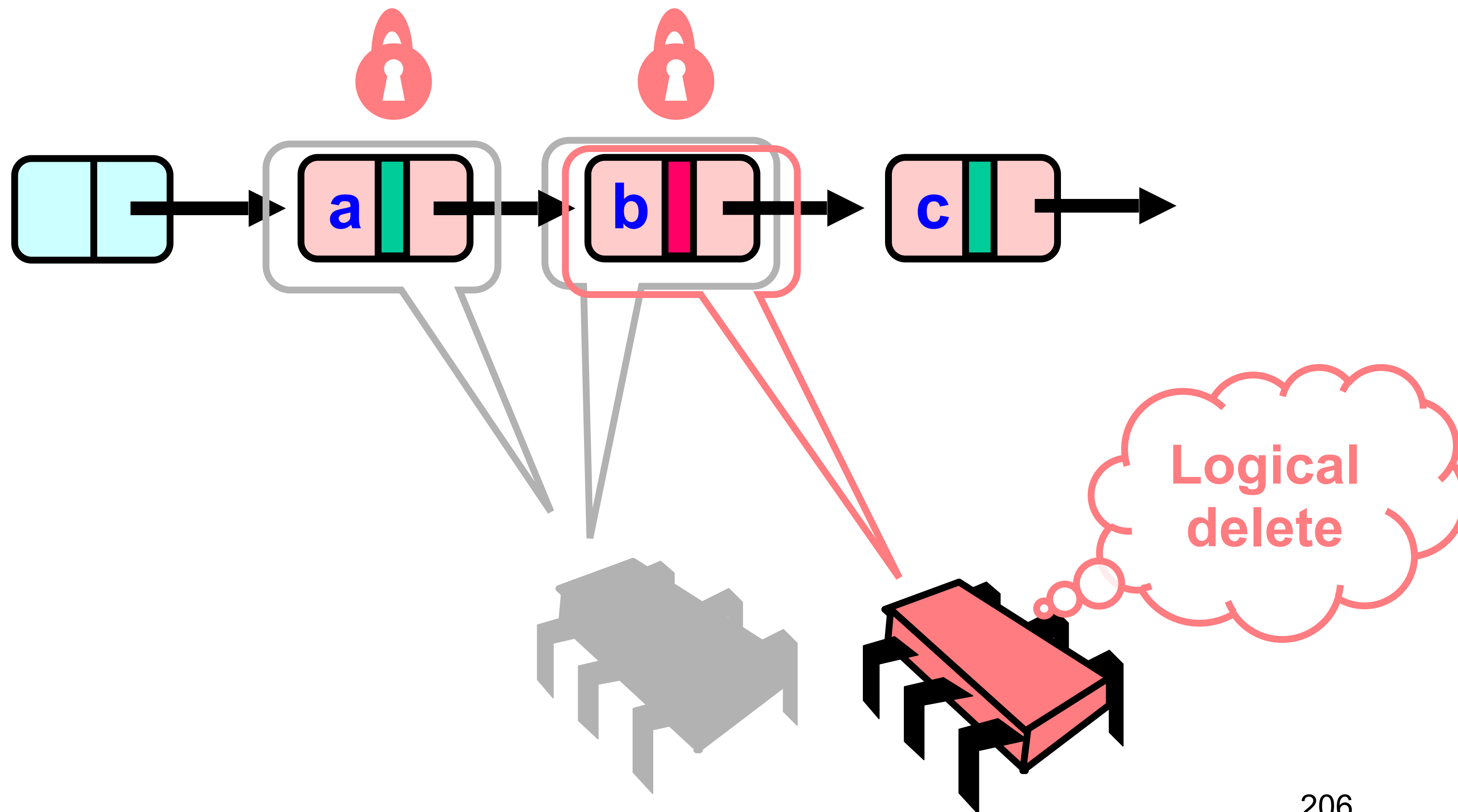
# Business as Usual



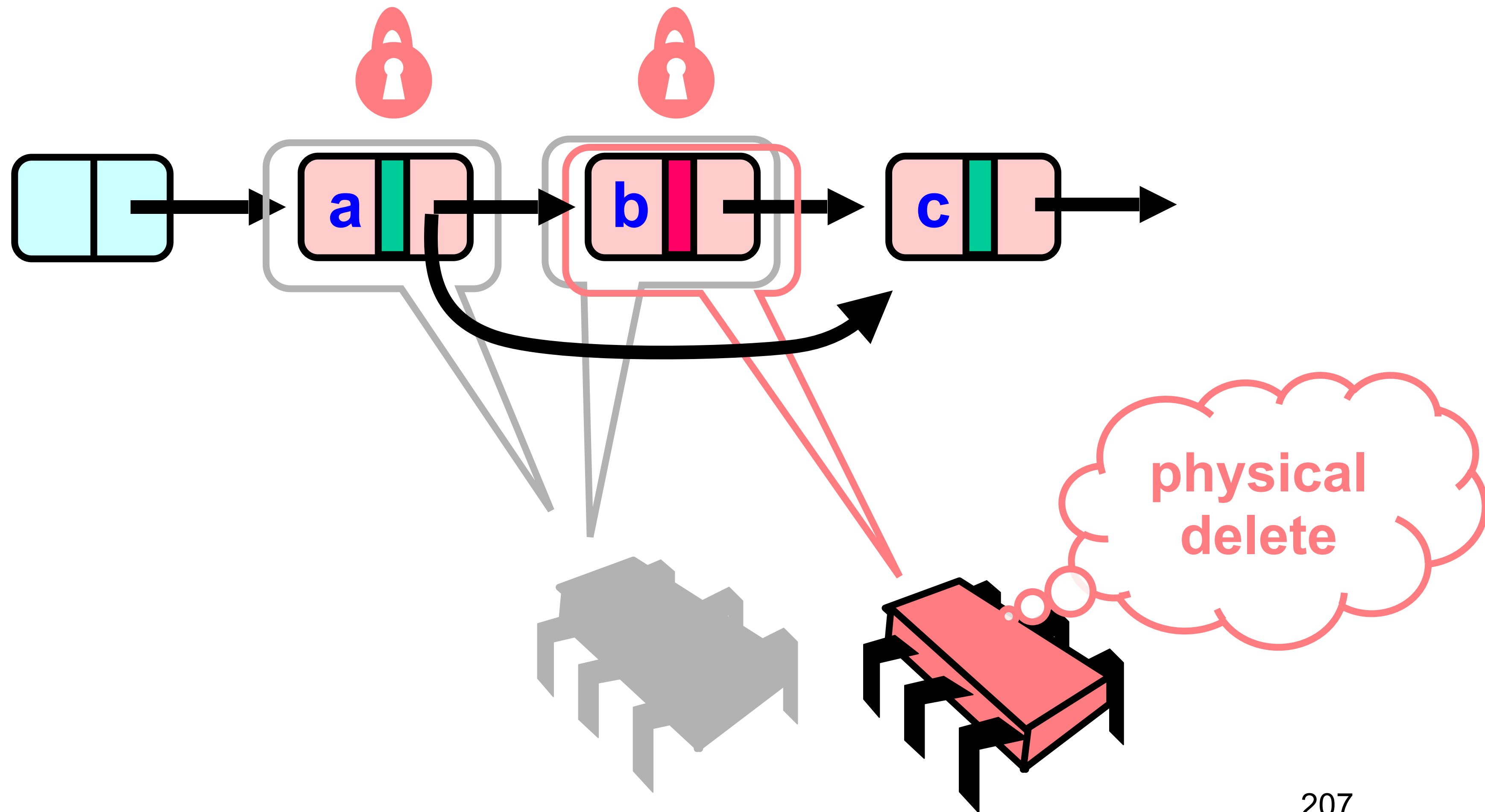
# Business as Usual



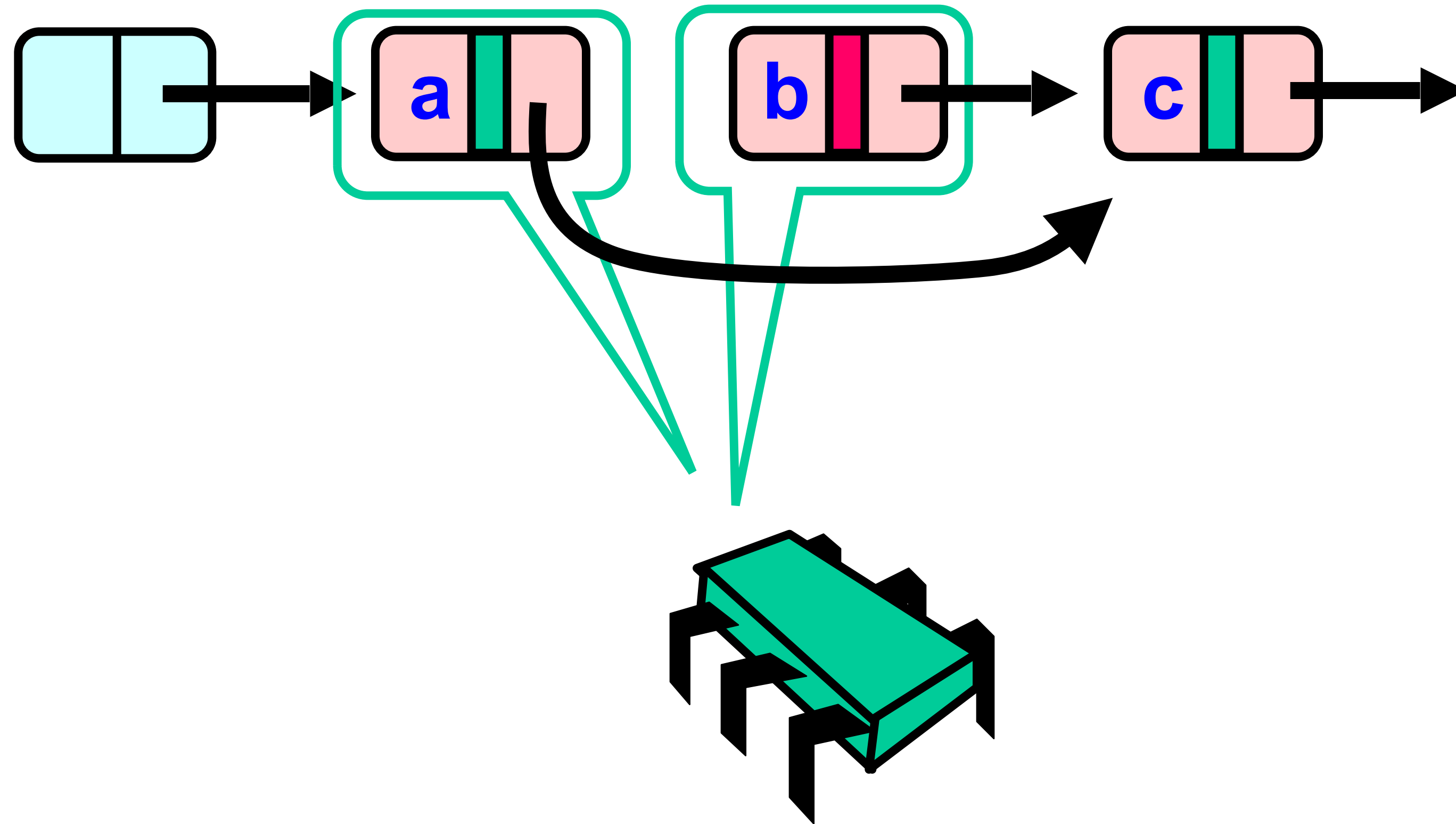
# Business as Usual



# Business as Usual



# Business as Usual





# New Abstraction Map

- $S(\text{head}) =$ 
  - $\{ x \mid$  there exists node  $a$  such that
    - $a$  reachable from head and
    - $a.\text{item} = x$  and
    - $a$  is unmarked
  - $\}$

# Invariant

- If not marked then item in the set
- and is reachable from head
- and if not yet traversed it is reachable from pred

# Validation

```
def validate(pred: Node, curr: Node) =  
    !pred.marked &&  
    !curr.marked &&  
    (pred.next eq curr)
```

# List Validate Method

```
def validate(pred: Node, curr: Node) =  
  !pred.marked &&  
  !curr.marked &&  
  (pred.next eq curr)
```

**Predecessor not  
Logically removed**

# List Validate Method

```
def validate(pred: Node, curr: Node) =  
  !pred.marked &&  
  !curr.marked &&  
  (pred.next eq curr)
```

**Current not  
Logically removed**

# List Validate Method

```
def validate(pred: Node, curr: Node) =  
  !pred.marked &&  
  !curr.marked &&  
  (pred.next eq curr)
```

**Predecessor still  
Points to current**

# Remove

```
try {
    pred.lock(); curr.lock()
    if (validate(pred, curr) {
        if (curr.key == key) {
            curr.marked = true
            pred.next = curr.next
            return true;
        } else {
            return false
        }
    } finally {
        pred.unlock()
        curr.unlock()
    }
}
```

# Remove

```
try {
    pred.lock(); curr.lock()
    if (validate(pred, curr) {
        if (curr.key == key) {
            curr.marked = true
            pred.next = curr.next
            return true
        } else {
            return false
        }
    } finally {
        pred.unlock()
        curr.unlock()
    }
}
```

**Validate as before**



# Remove

```
try {  
    pred.lock(); curr.lock();  
    if (validate(pred, curr) {  
        if (curr.key == key) {  
            curr.marked = true;  
            pred.next = curr.next;  
            return true;  
        } else {  
            return false;  
        }  
    }  
} finally {  
    pred.unlock();  
    curr.unlock();  
}
```

**Key found**

# Remove

```
try {
  pred.lock(); curr.lock()
  if (validate(pred, curr) {
    if (curr.key == key) {
      curr.marked = true;
      pred.next = curr.next
      return true
    } else {
      return false
    }
  } finally {
    pred.unlock()
    curr.unlock()
  }
}
```

**Logical remove**

# Remove

```
try {  
    pred.lock(); curr.lock()  
    if (validate(pred, curr) {  
        if (curr.key == key) {  
            curr.marked = true  
            pred.next = curr.next;  
            return true  
        } else {  
            return false  
        }  
    }  
} finally {  
    pred.unlock()  
    curr.unlock()  
}
```

**physical remove**

# Contains

```
def contains(item: T) = {  
  val key = item.hashCode  
  var curr = this.head  
  while (curr.key < key) curr = curr.next  
  curr.key == key && !curr.marked  
}
```

# Contains

```
def contains(item: T) = {  
  val key = item.hashCode  
  var curr = this.head  
  while (curr.key < key) curr = curr.next  
  curr.key == key && !curr.marked  
}
```

**Start at the head**

# Contains

```
def contains(item: T) = {  
  val key = item.hashCode  
  var curr = this.head  
  while (curr.key < key) curr = curr.next  
  curr.key == key && !curr.marked  
}
```

**Search key range**

# Contains

```
def contains(item: T) = {  
  val key = item.hashCode  
  var curr = this.head  
  while (curr.key < key) curr = curr.next  
  curr.key == key && !curr.marked  
}
```

**Traverse *without locking***  
**(nodes may have been removed)**

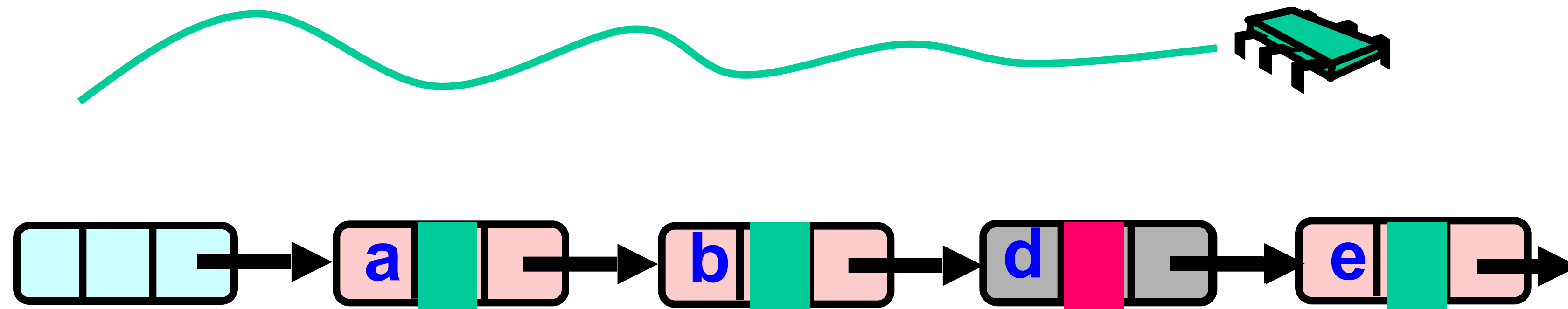
# Contains

```
def contains(item: T) = {  
  val key = item.hashCode  
  var curr = this.head  
  while (curr.key < key) curr = curr.next  
  curr.key == key && !curr.marked  
}
```

**Present and undeleted?**



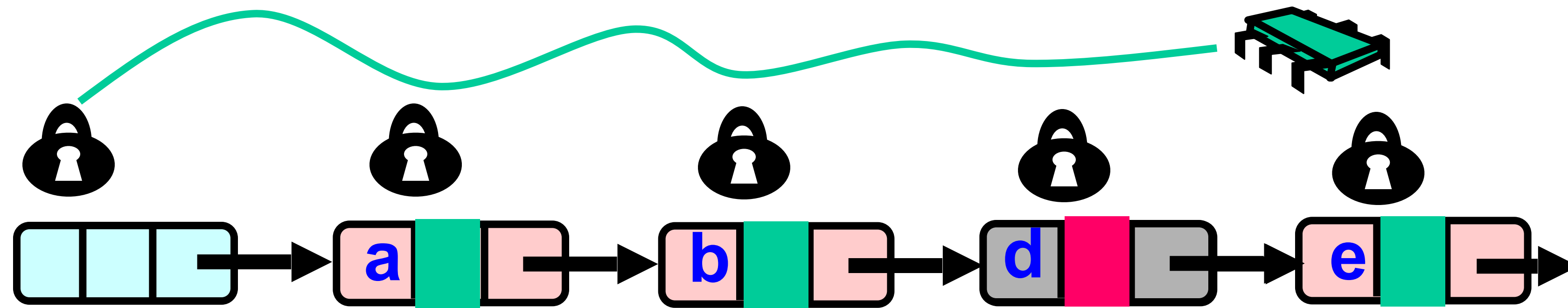
# Summary: Wait-free Contains



Use Mark bit + list ordering

1. Not marked  $\rightarrow$  in the set
2. Marked or missing  $\rightarrow$  not in the set

# Lazy List



Lazy add () and remove () + Wait-free contains ()

# Evaluation

- Good:
  - `contains ()` doesn't lock
  - In fact, it's wait-free!
  - Good because typically high % `contains()`
  - Uncontended calls don't re-traverse
- Bad
  - Contended `add ()` and `remove ()` calls must re-traverse
  - Traffic jam if one thread delays

# Traffic Jam

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

# Reminder: Lock-Free Data Structures



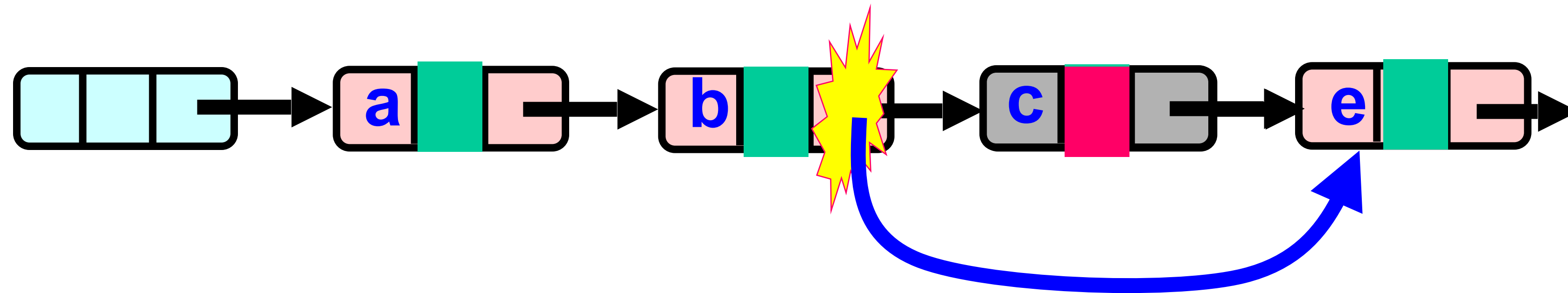
- No matter what ...
  - Guarantees minimal progress in any execution
  - i.e. Some thread will always complete a method call
  - Even if others halt at malicious times
  - Implies that implementation can't use locks

# Lock-free Lists

- Next logical step
  - Wait-free `contains()`
  - lock-free `add()` and `remove()`
- Use only `compareAndSet()`
  - What could go wrong?

# Lock-free Lists

Logical Removal

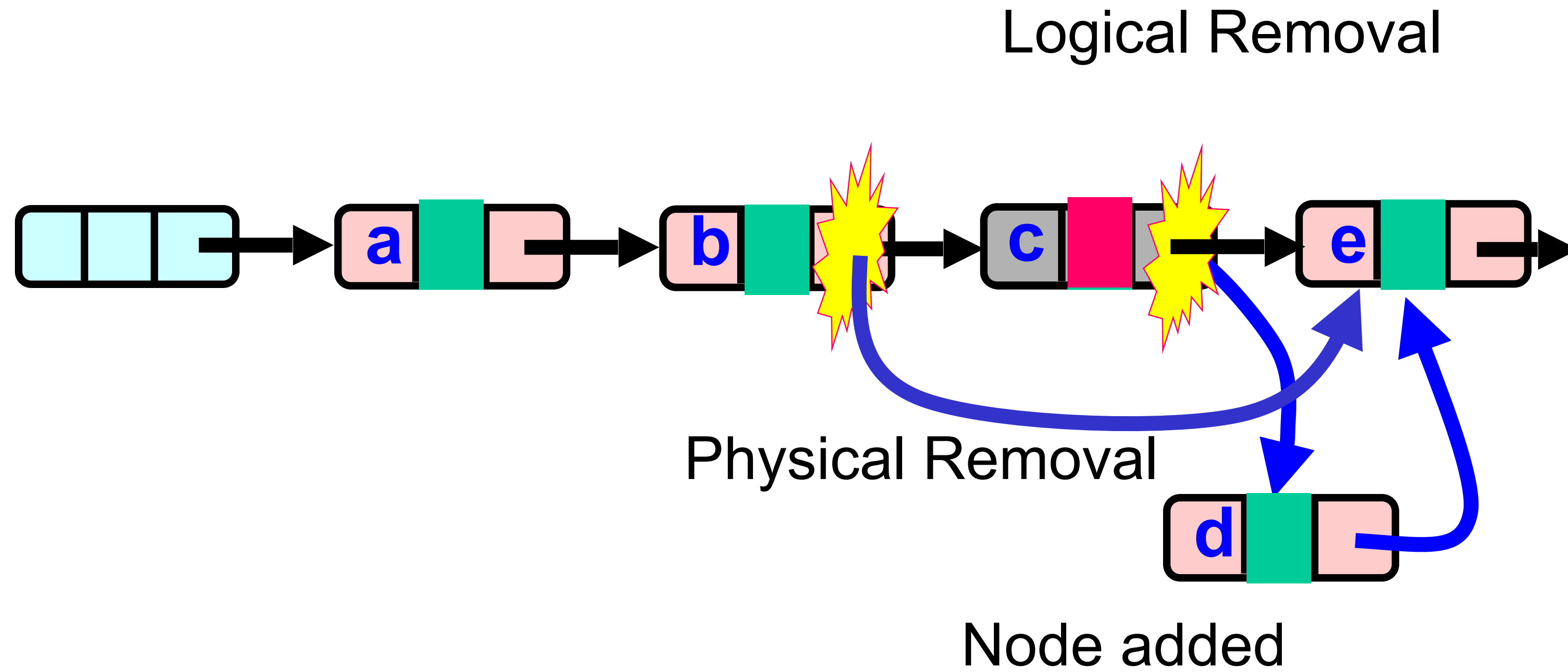


Physical Removal

Use CAS to verify pointer  
is correct

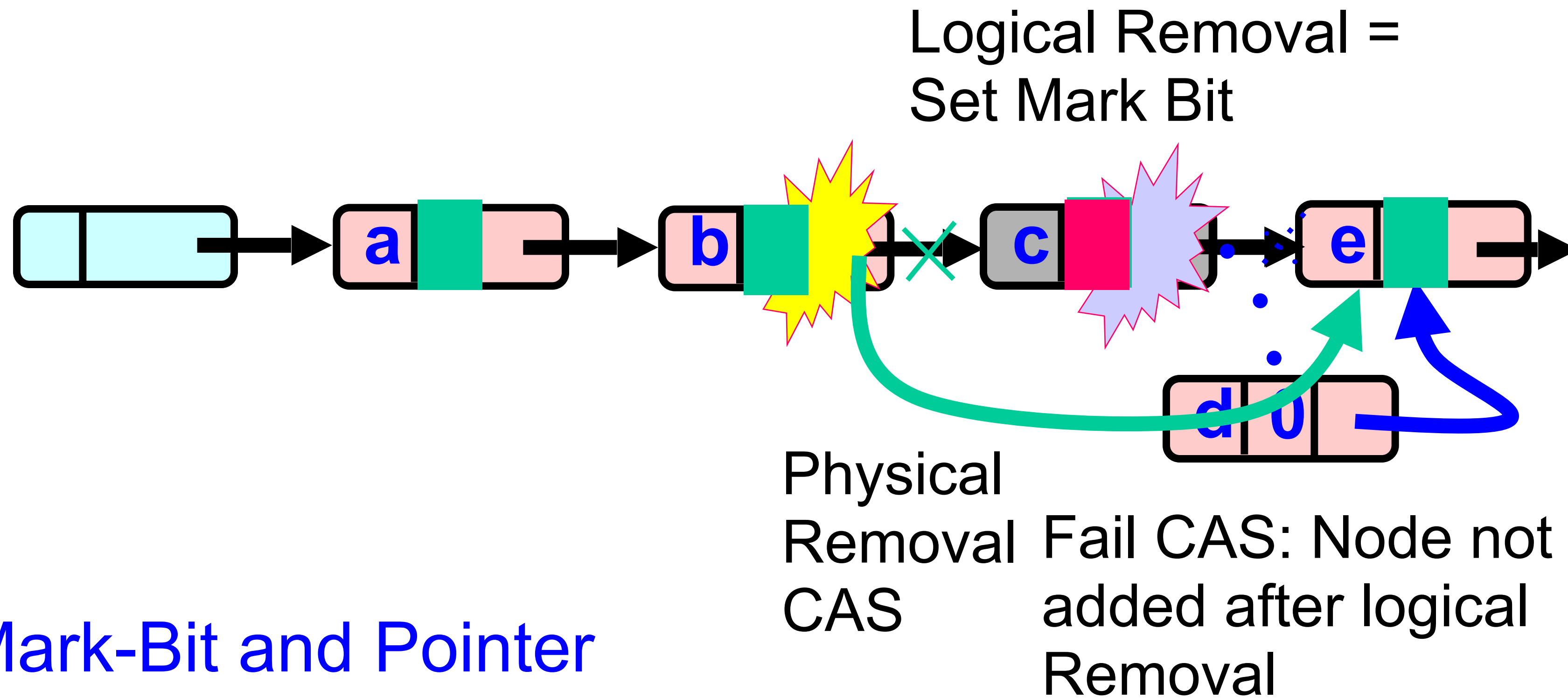
Not enough!

# Problem...





# The Solution: Combine Bit and Pointer



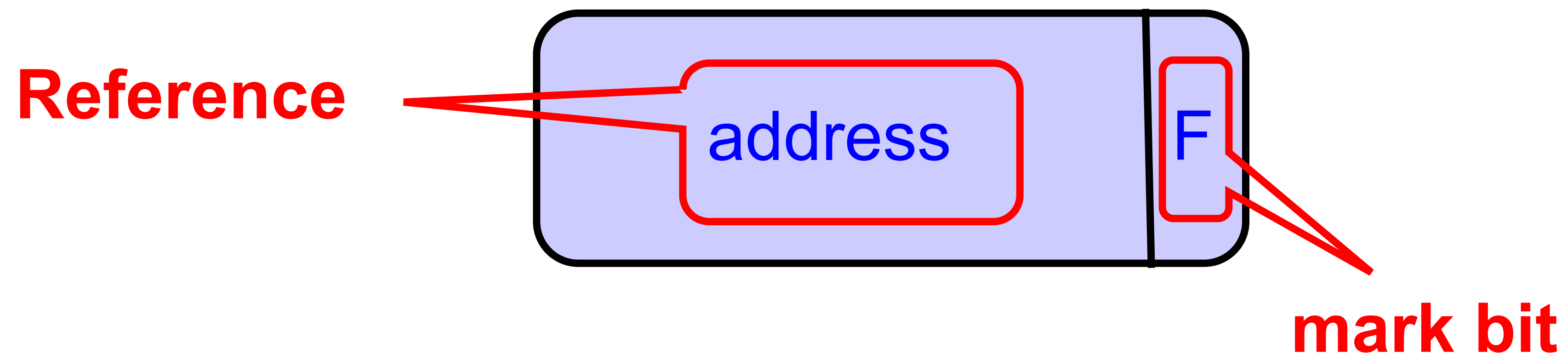
Mark-Bit and Pointer  
are CASed together  
**(AtomicMarkableReference)**

# 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

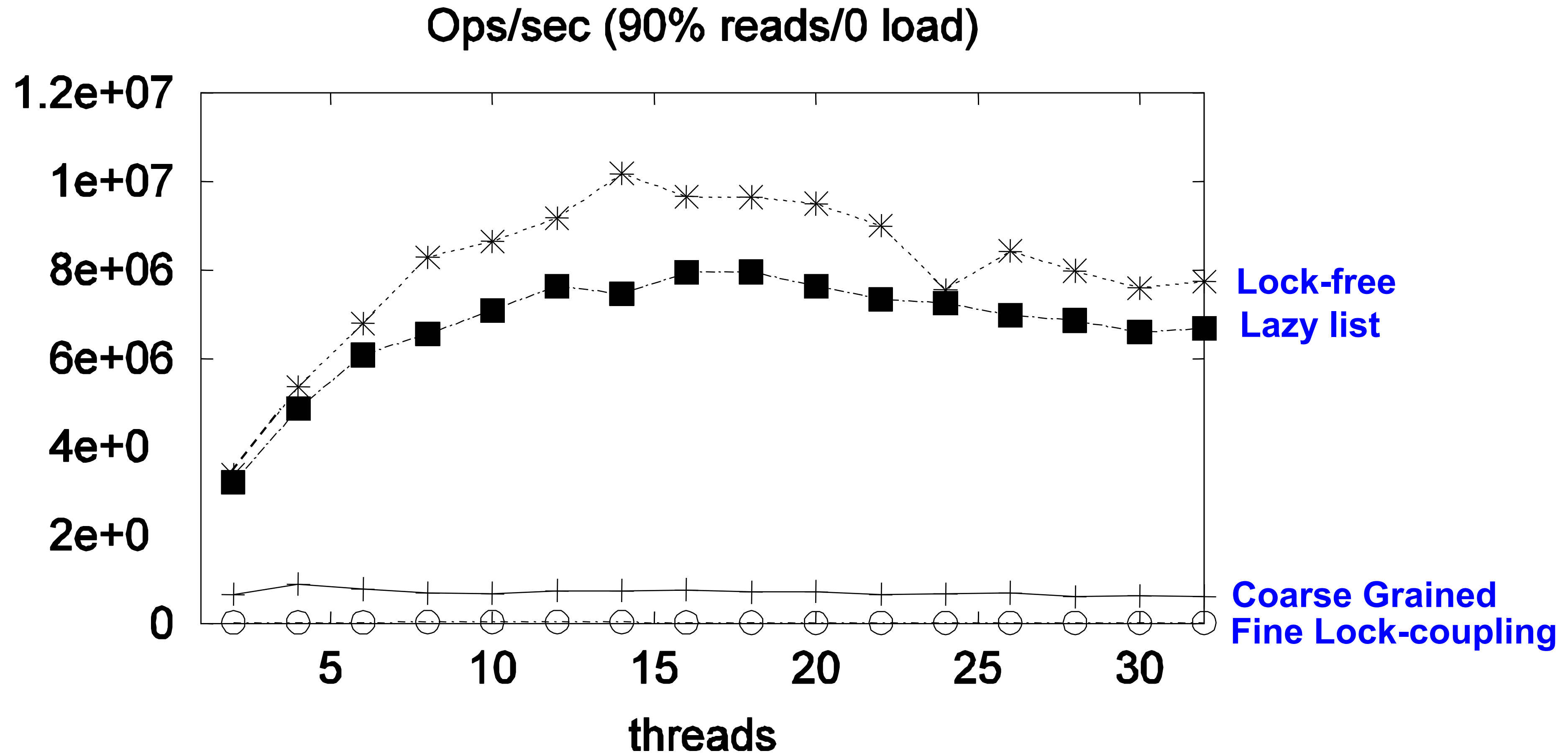
- **AtomicMarkableReference** class
  - Java.util.concurrent.atomic package



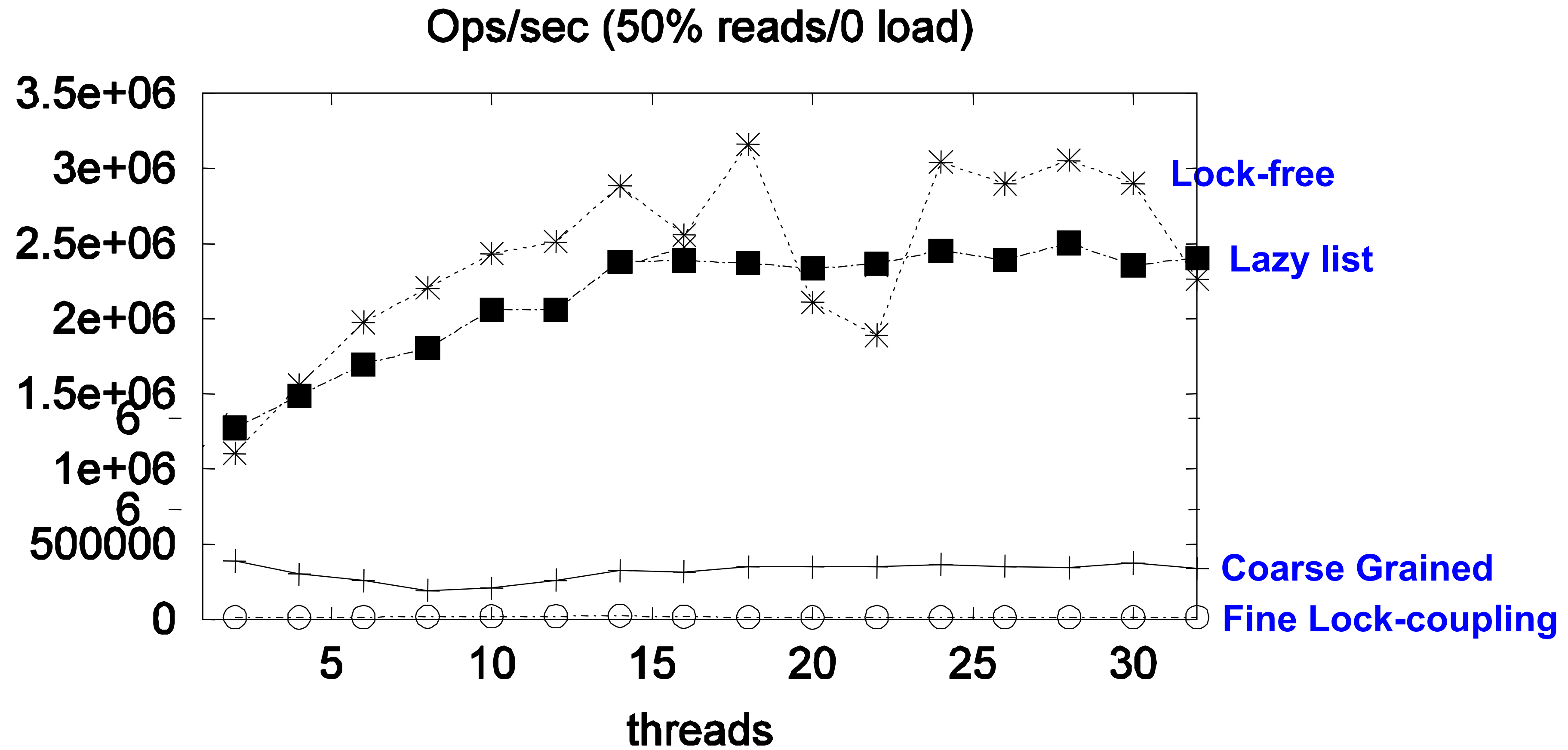
# Performance

- Different list-based set implementations
- 16-node machine
- Vary percentage of **contains ()** calls

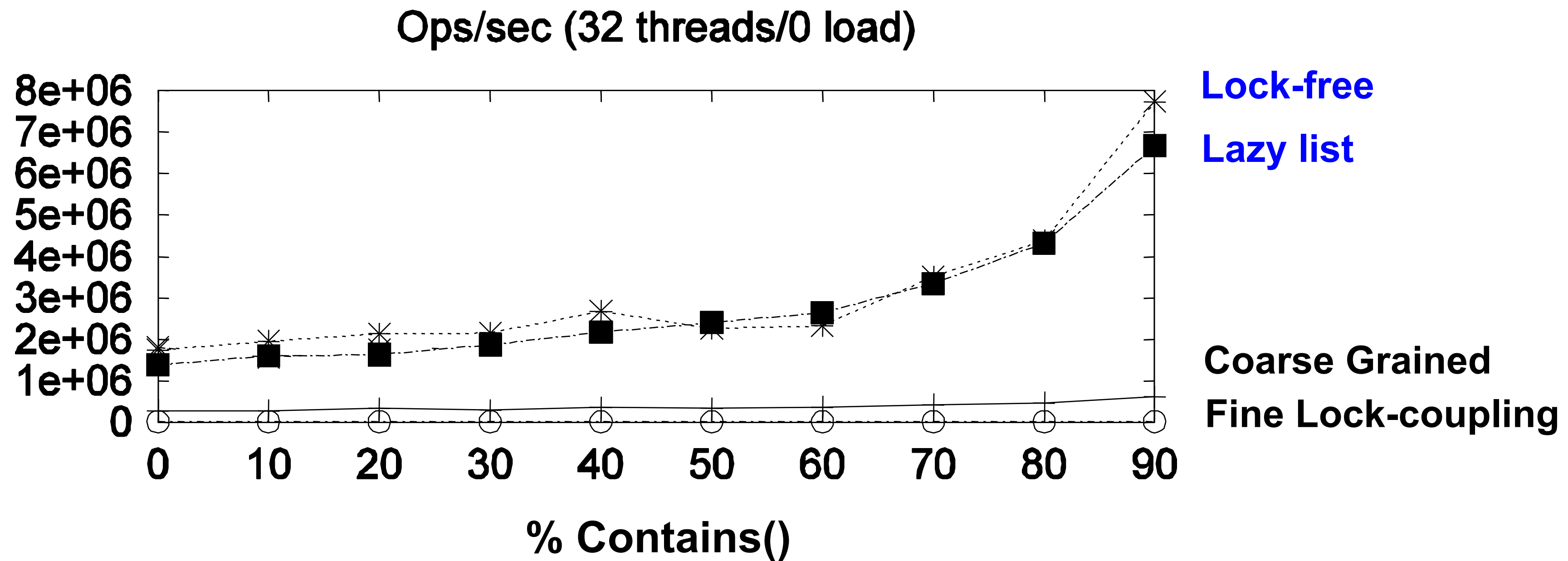
# High Contains Ratio



# Low Contains Ratio



# As Contains Ratio Increases



# Summary

- Coarse-grained locking
- Fine-grained locking (“hand-over-hand”)
- Optimistic synchronization
- Lazy synchronization
- Lock-free synchronization



# “To Lock or Not to Lock”

- Locking vs. Non-blocking:
  - Extremist views on both sides
  - Locking: longs waits
  - Non-blocking: long “clean-ups”
- The answer: nobler to compromise
  - Example: Lazy list combines blocking `add()` and `remove()` and a wait-free `contains()`
  - Remember: Blocking/non-blocking is a property of a method



This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](https://creativecommons.org/licenses/by-sa/2.5/).

- **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.