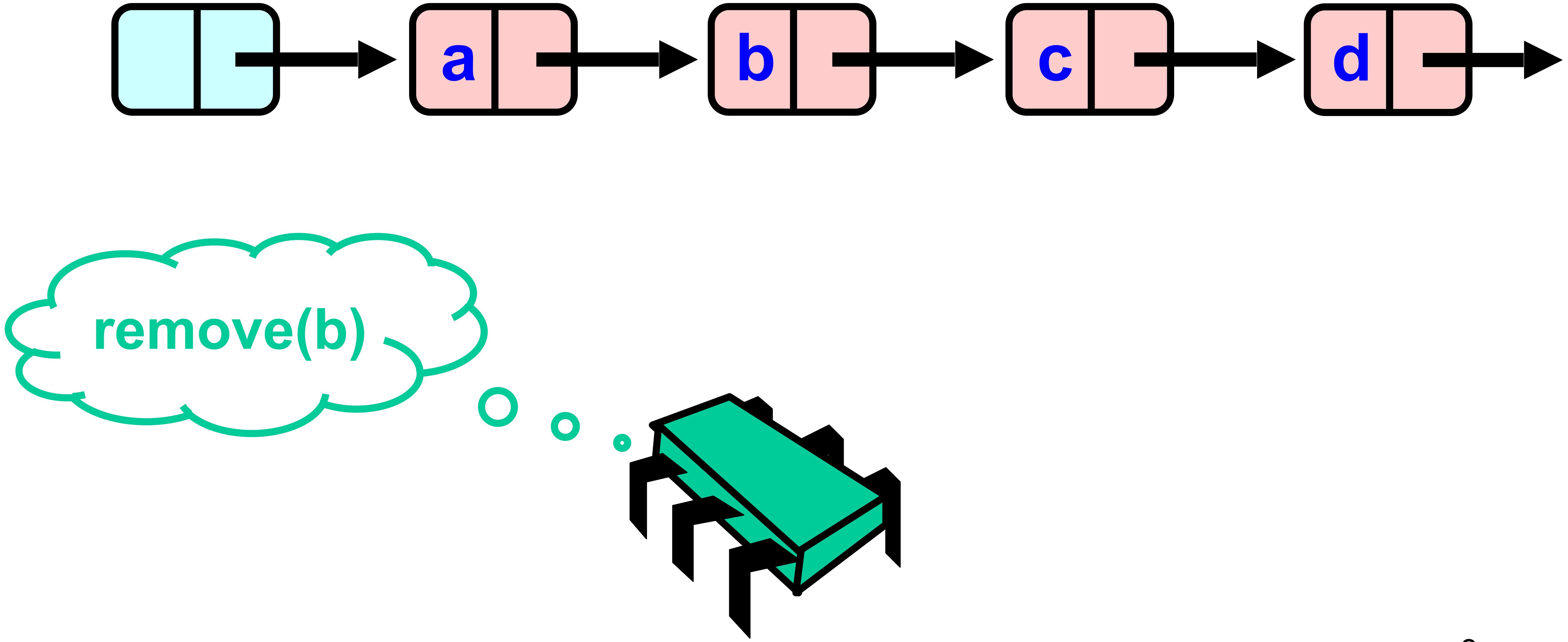
# YSC3248: Parallel, Concurrent and Distributed Programming

Concurrent Linked Lists
Part II

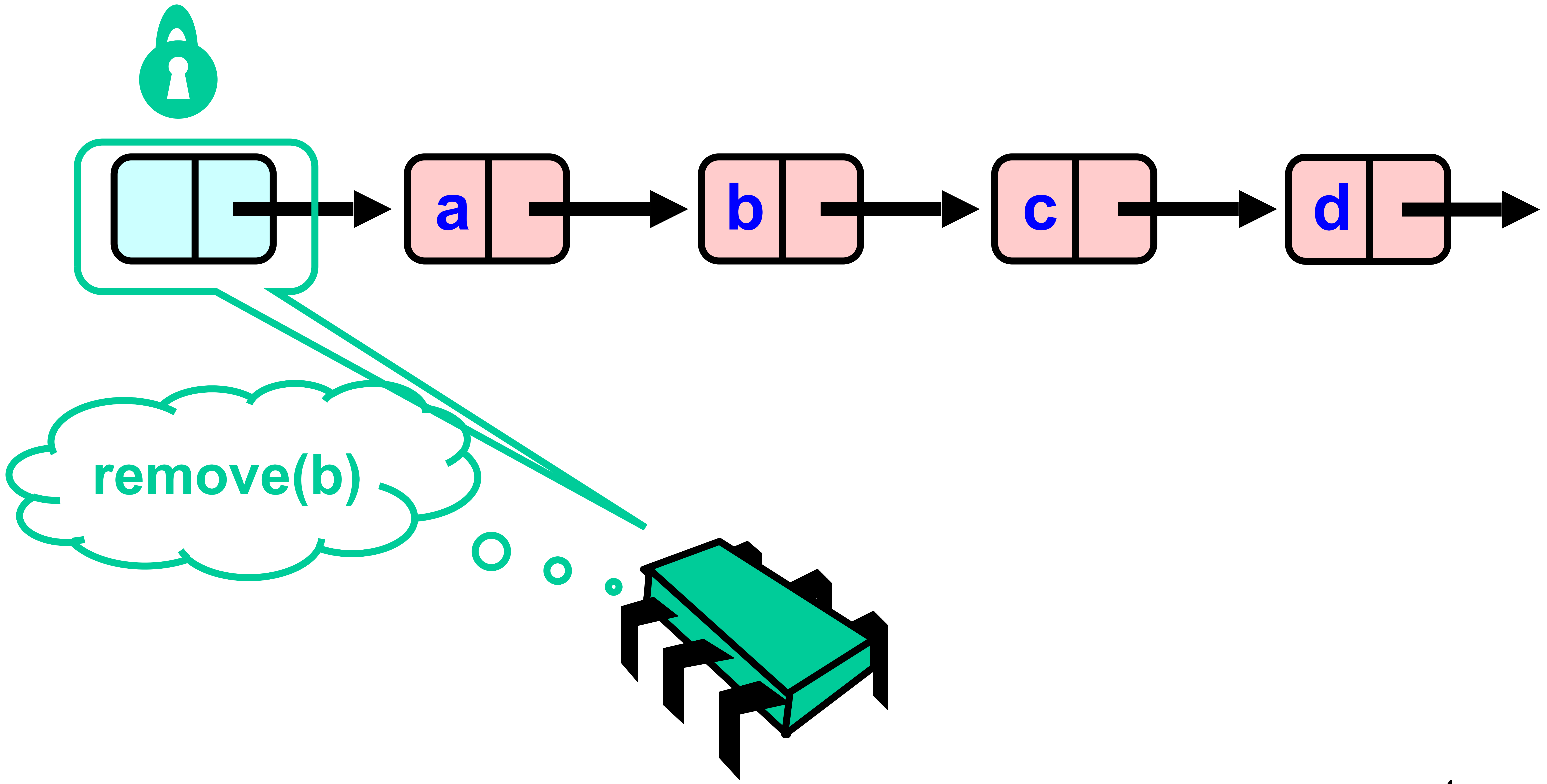
## Last Lecture: Fine-Grained Synchronization

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

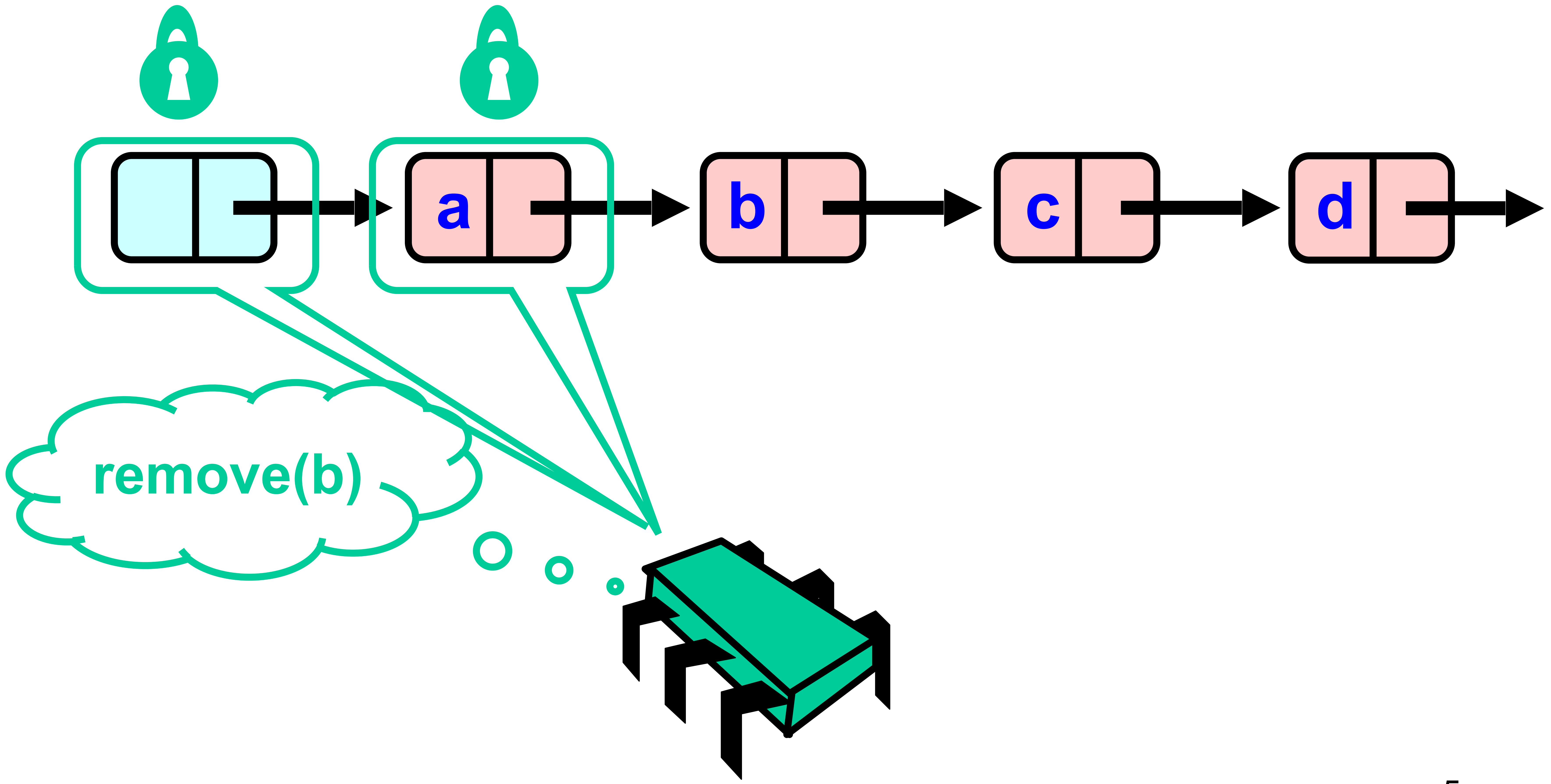
# Hand-Over-Hand Again



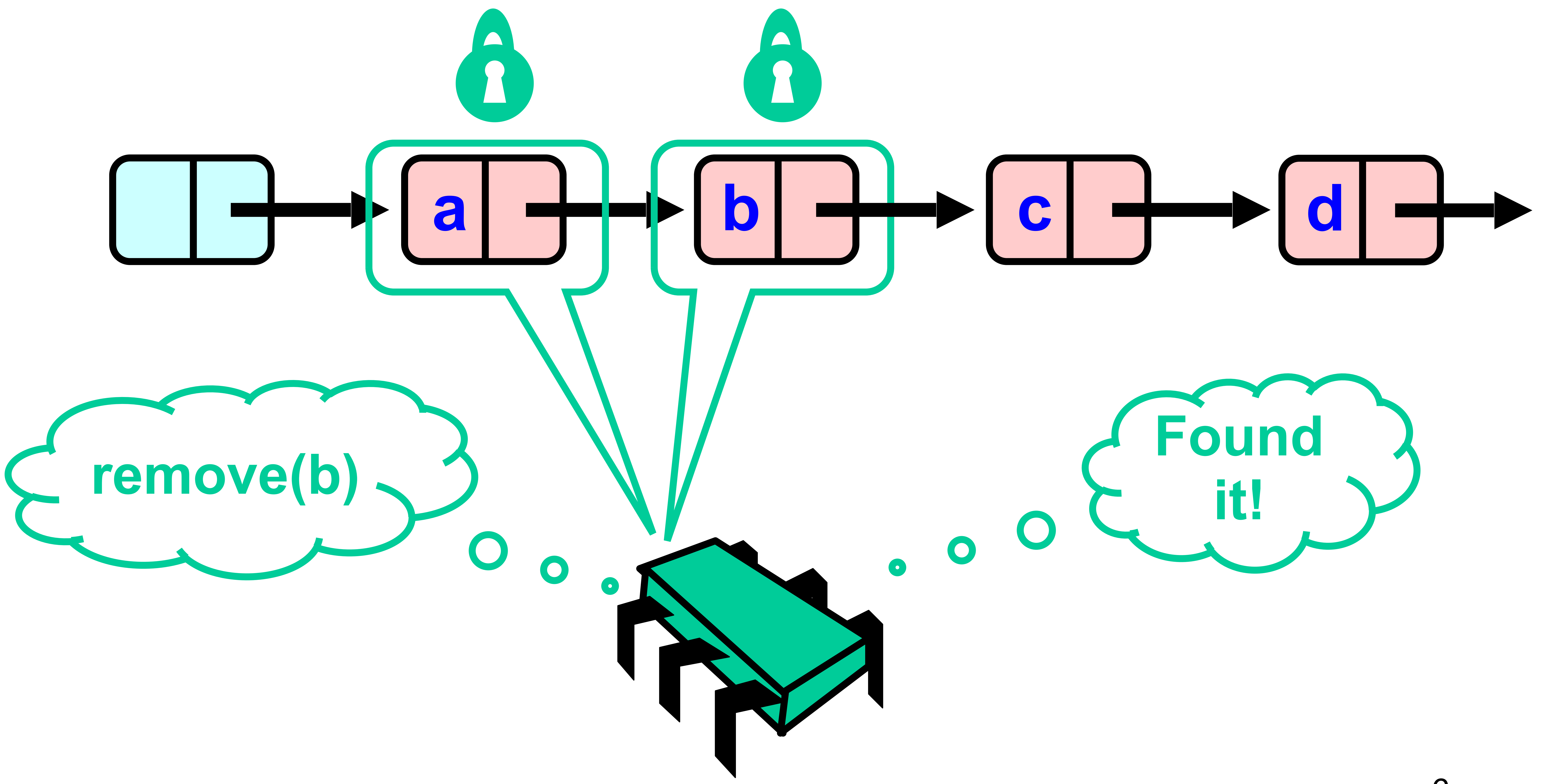
# Hand-Over-Hand Again



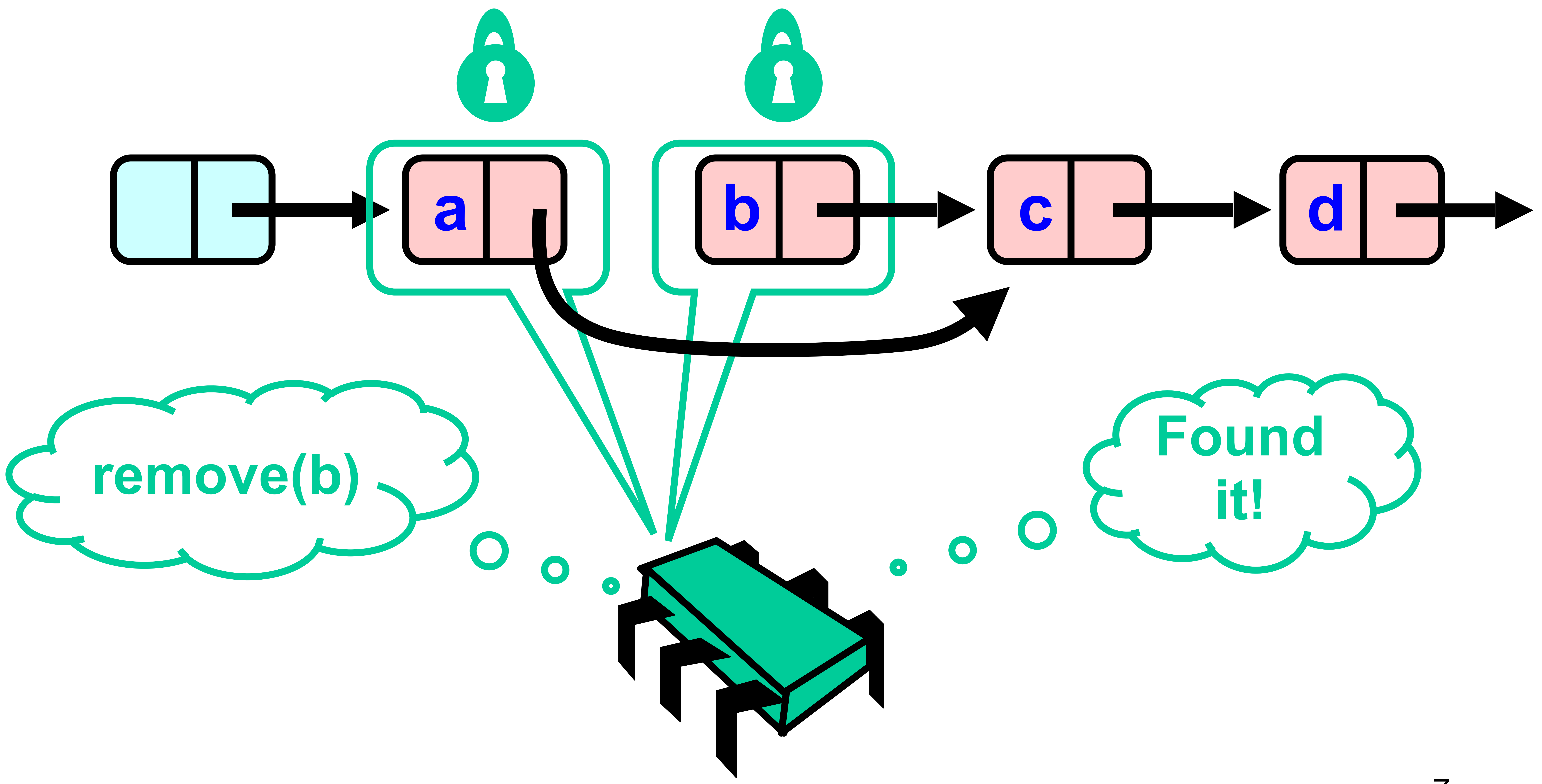
# Hand-Over-Hand Again



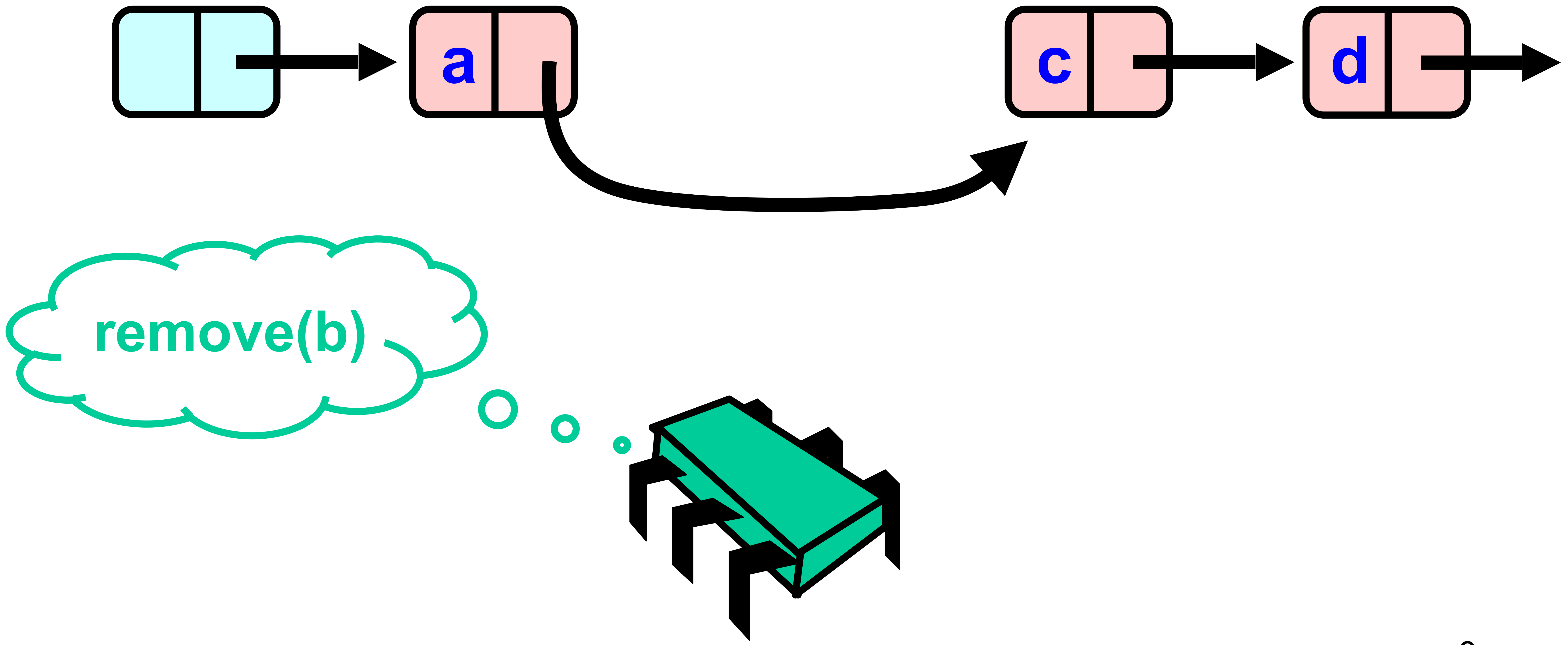
# Hand-Over-Hand Again



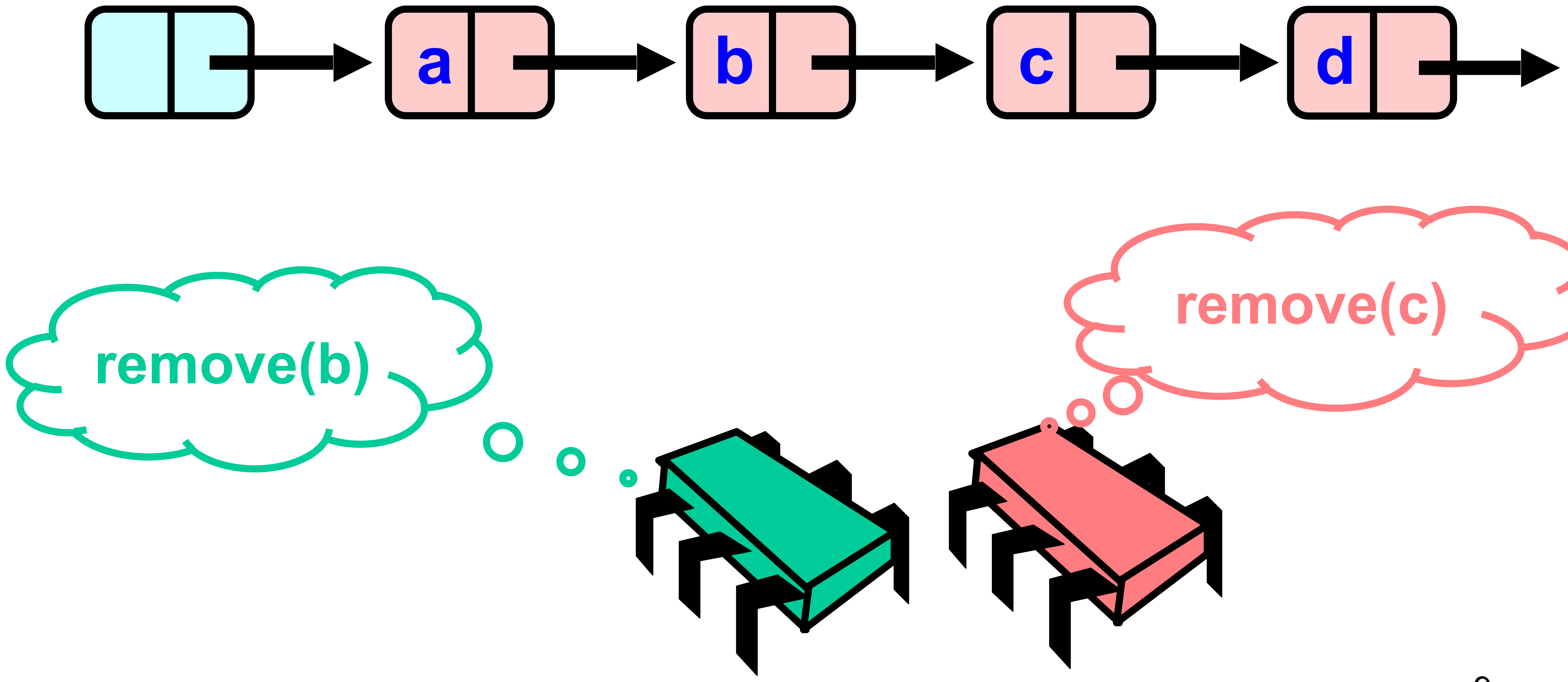
# Hand-Over-Hand Again



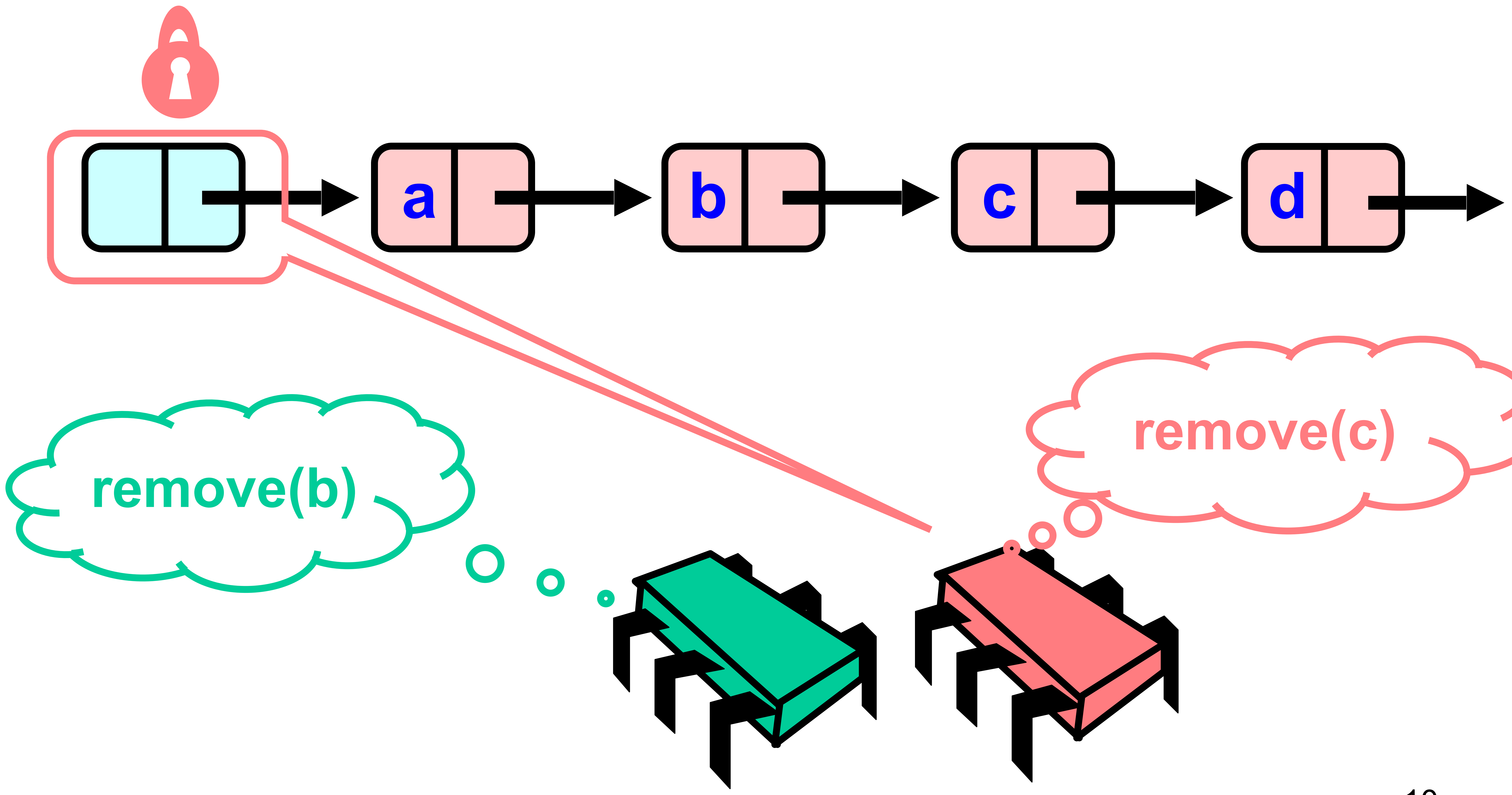
# Hand-Over-Hand Again



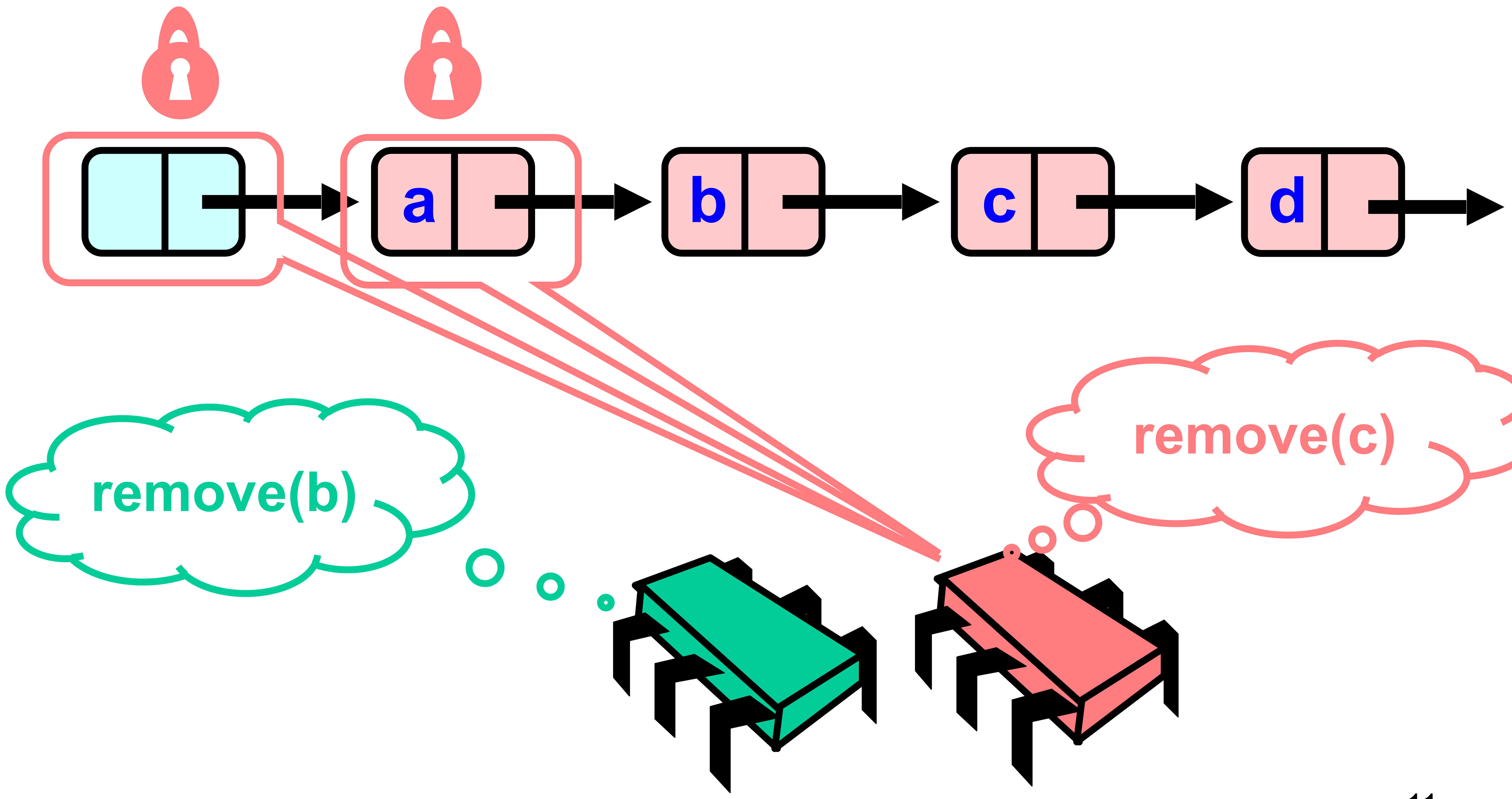
# Removing a Node



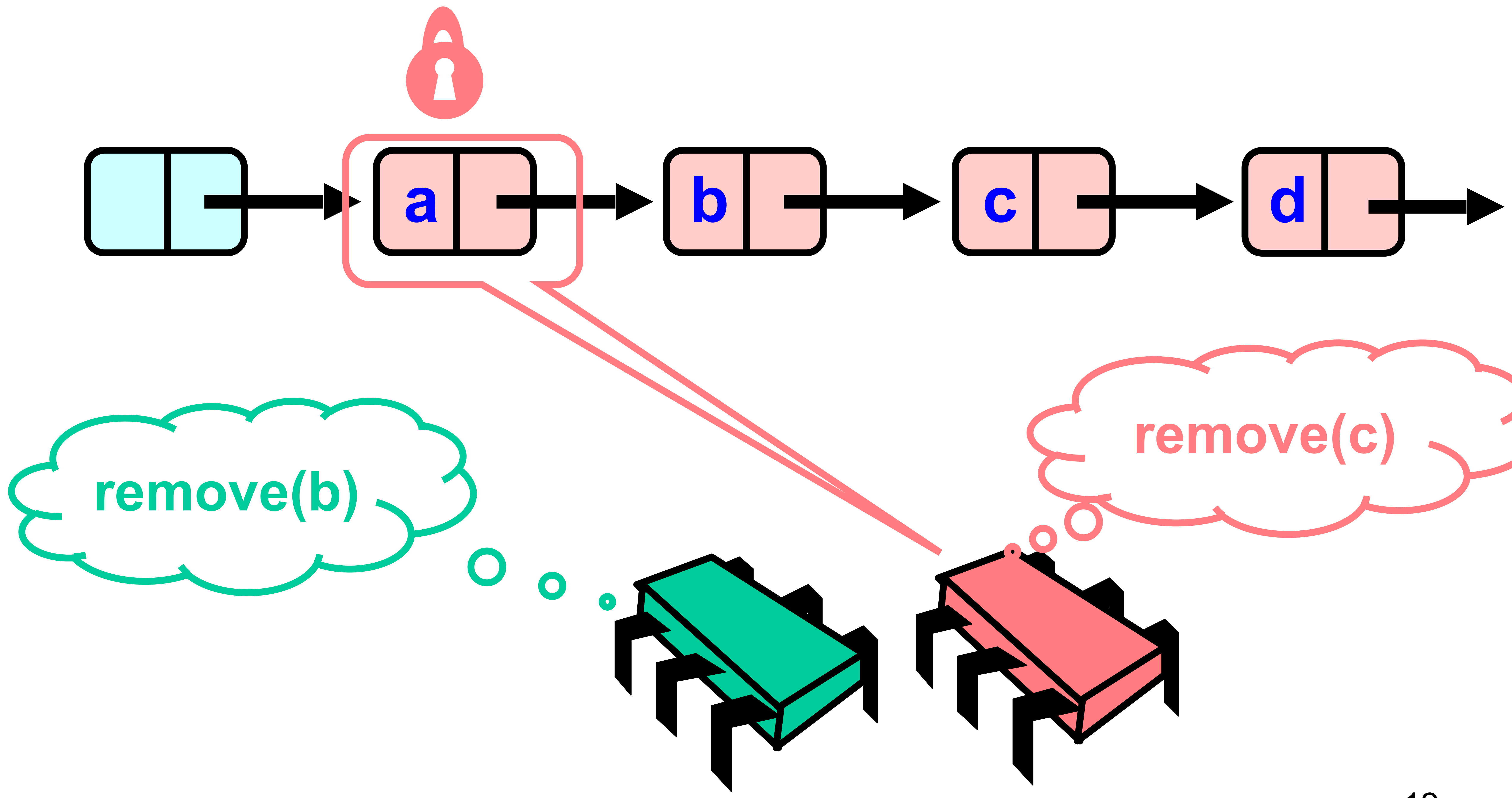
# Removing a Node



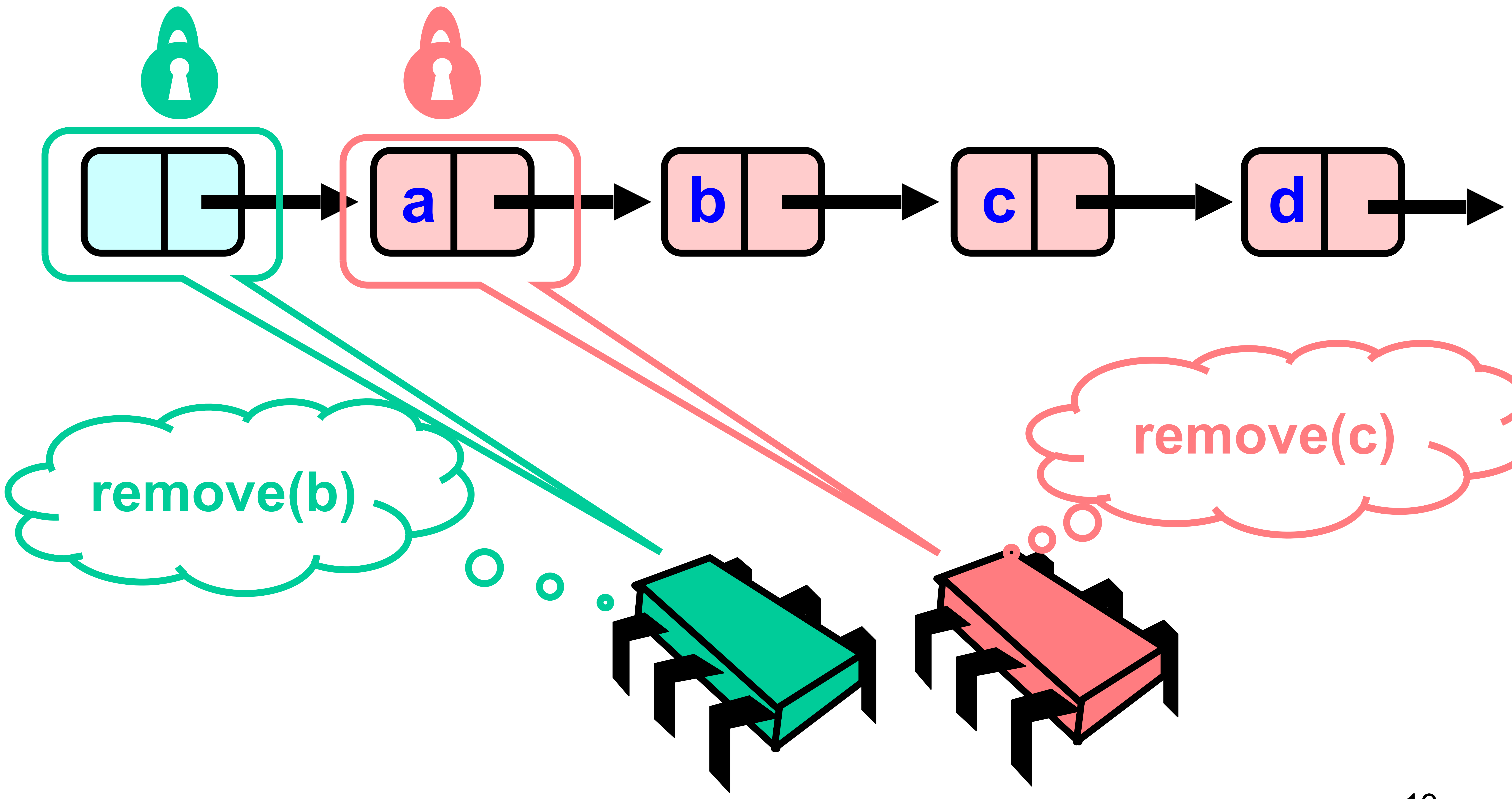
# Removing a Node



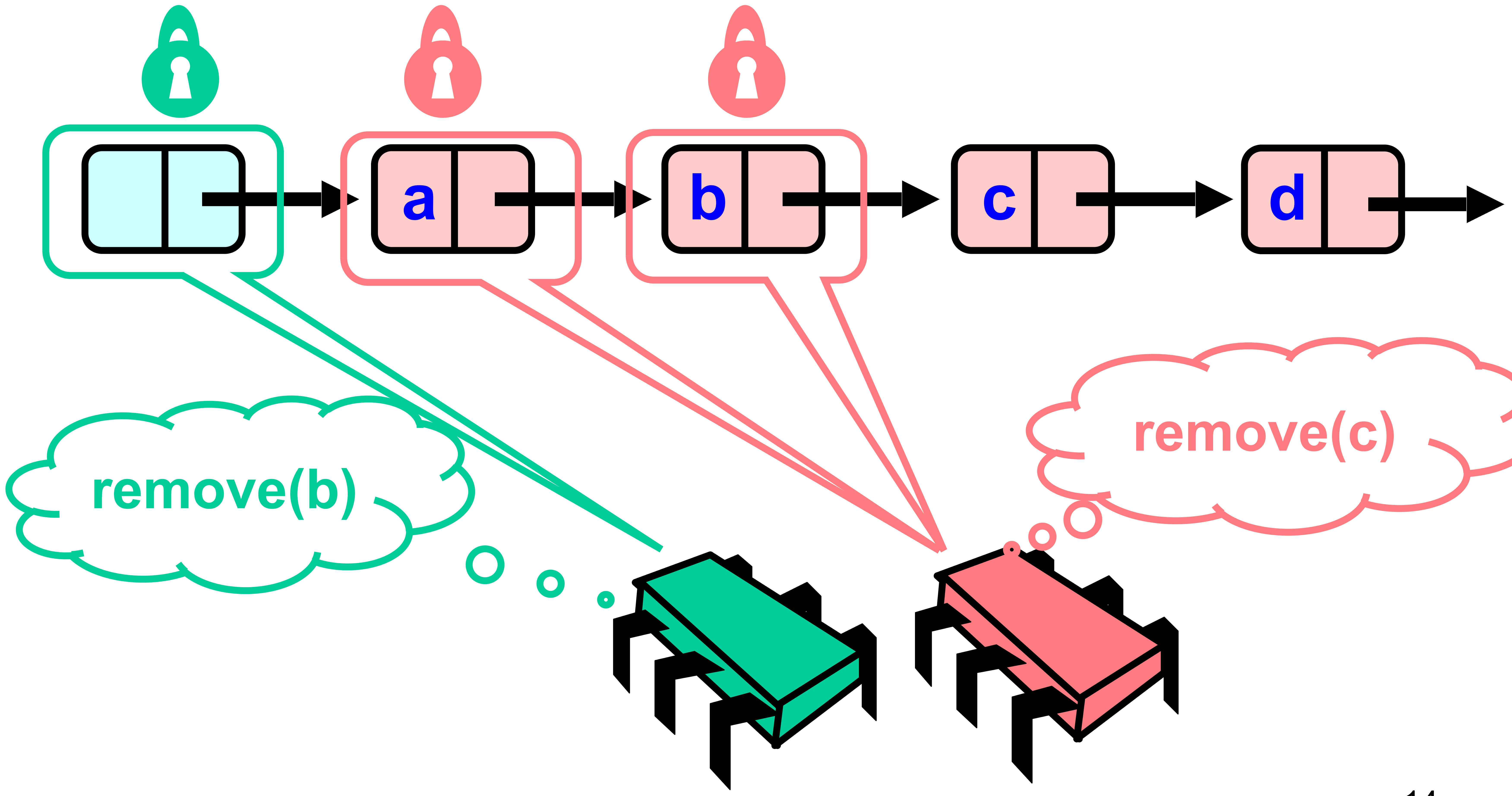
# Removing a Node



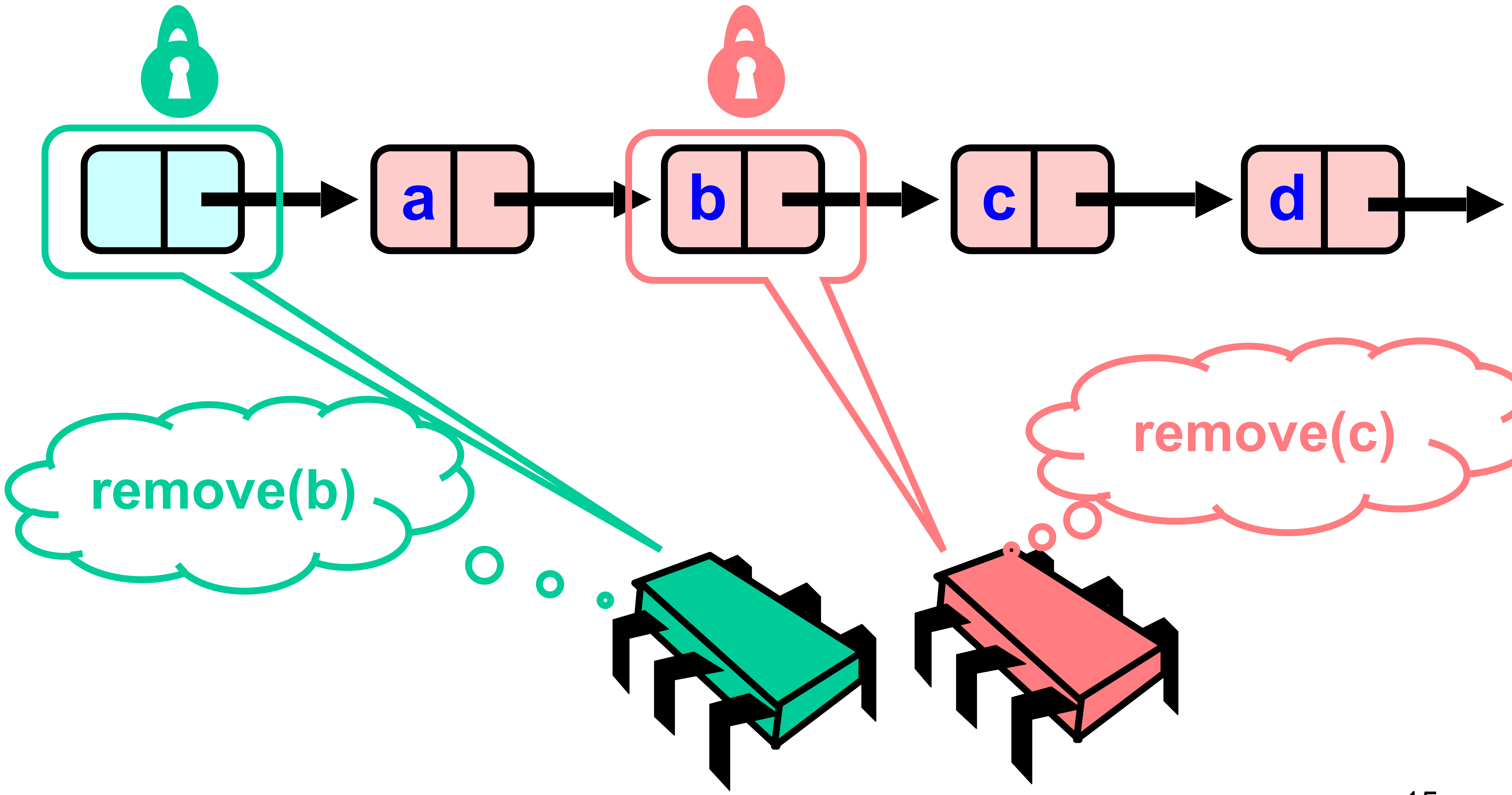
## Removing a Node



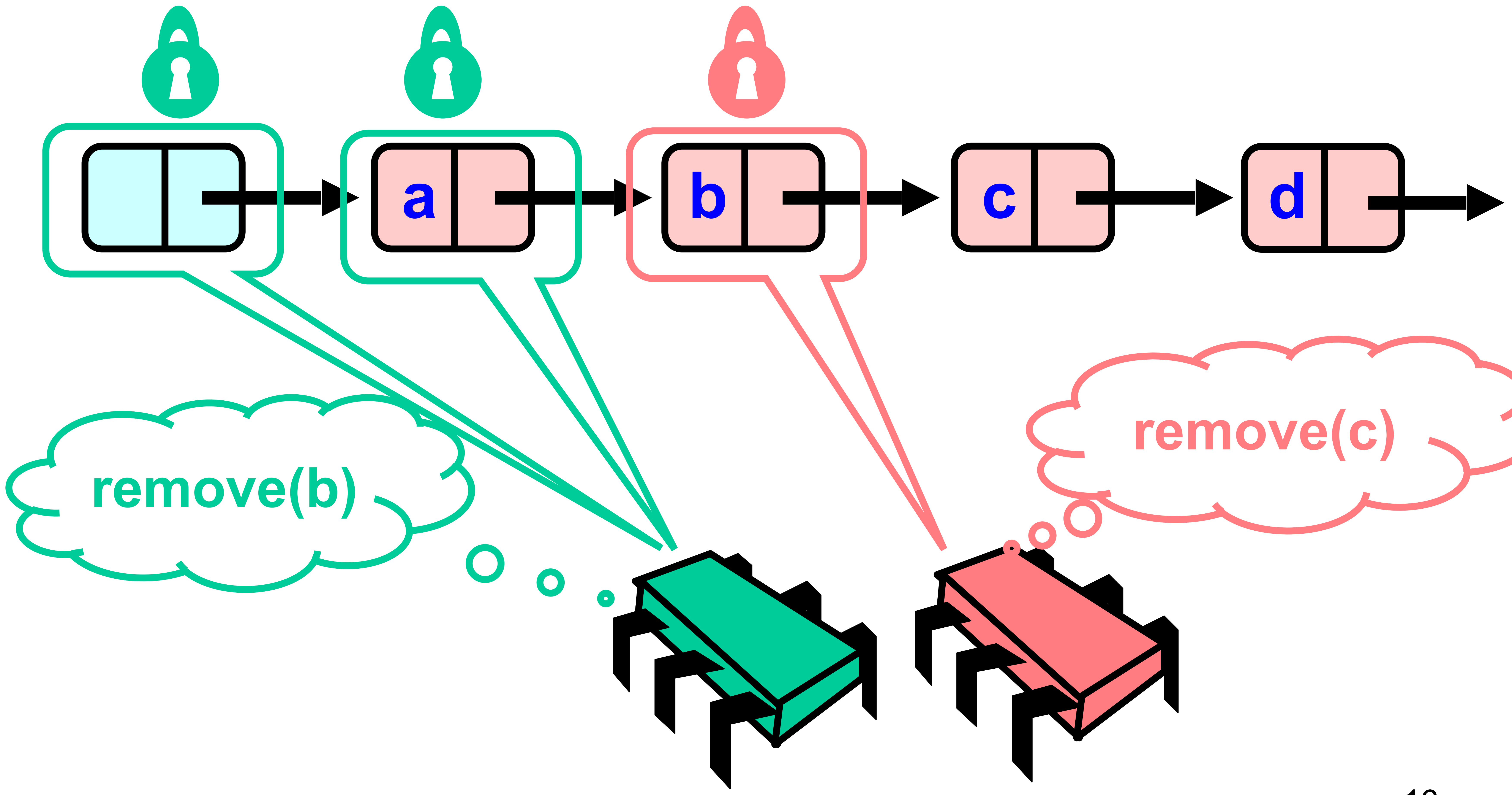
## Removing a Node



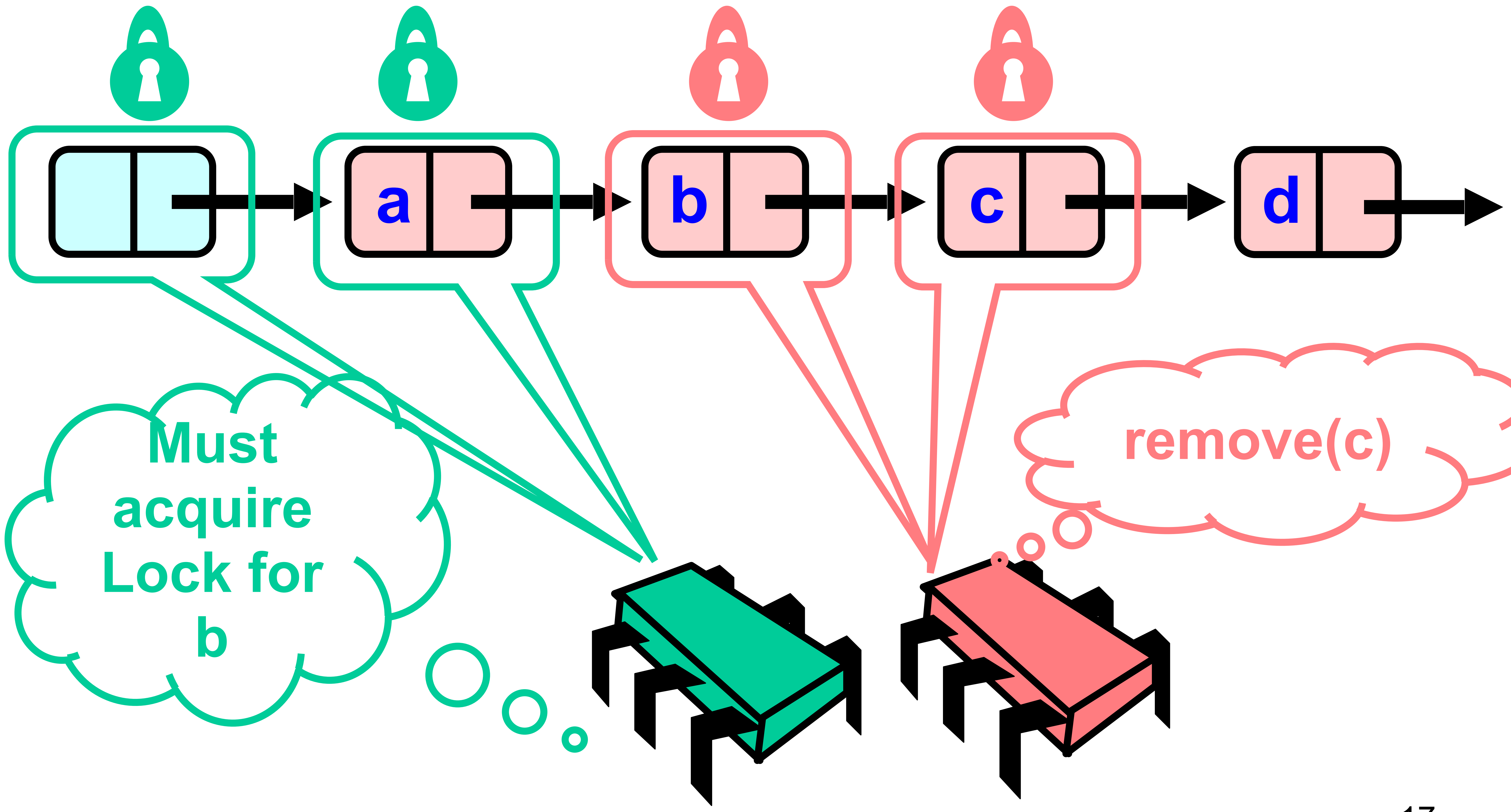
# Removing a Node



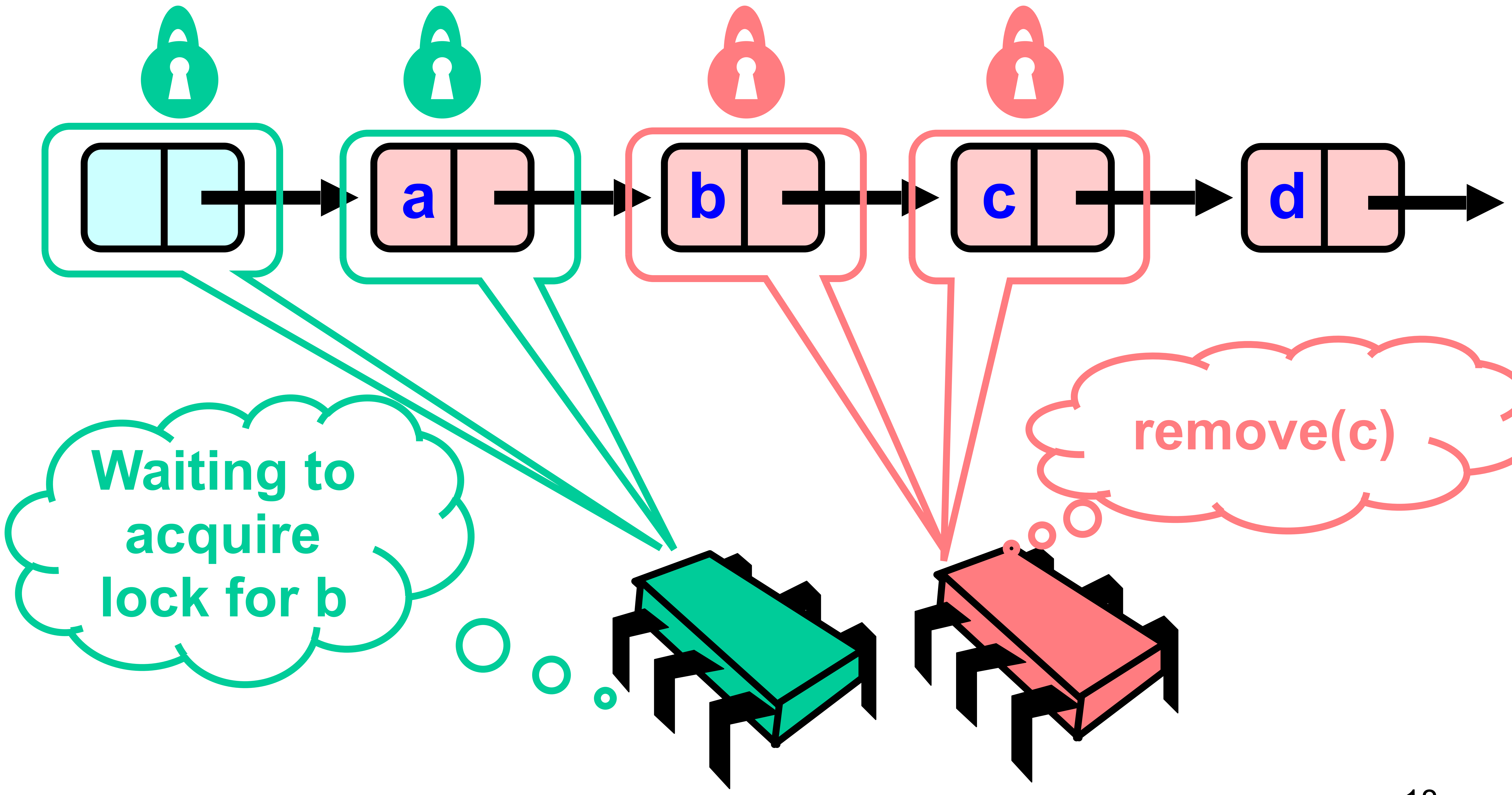
# Removing a Node



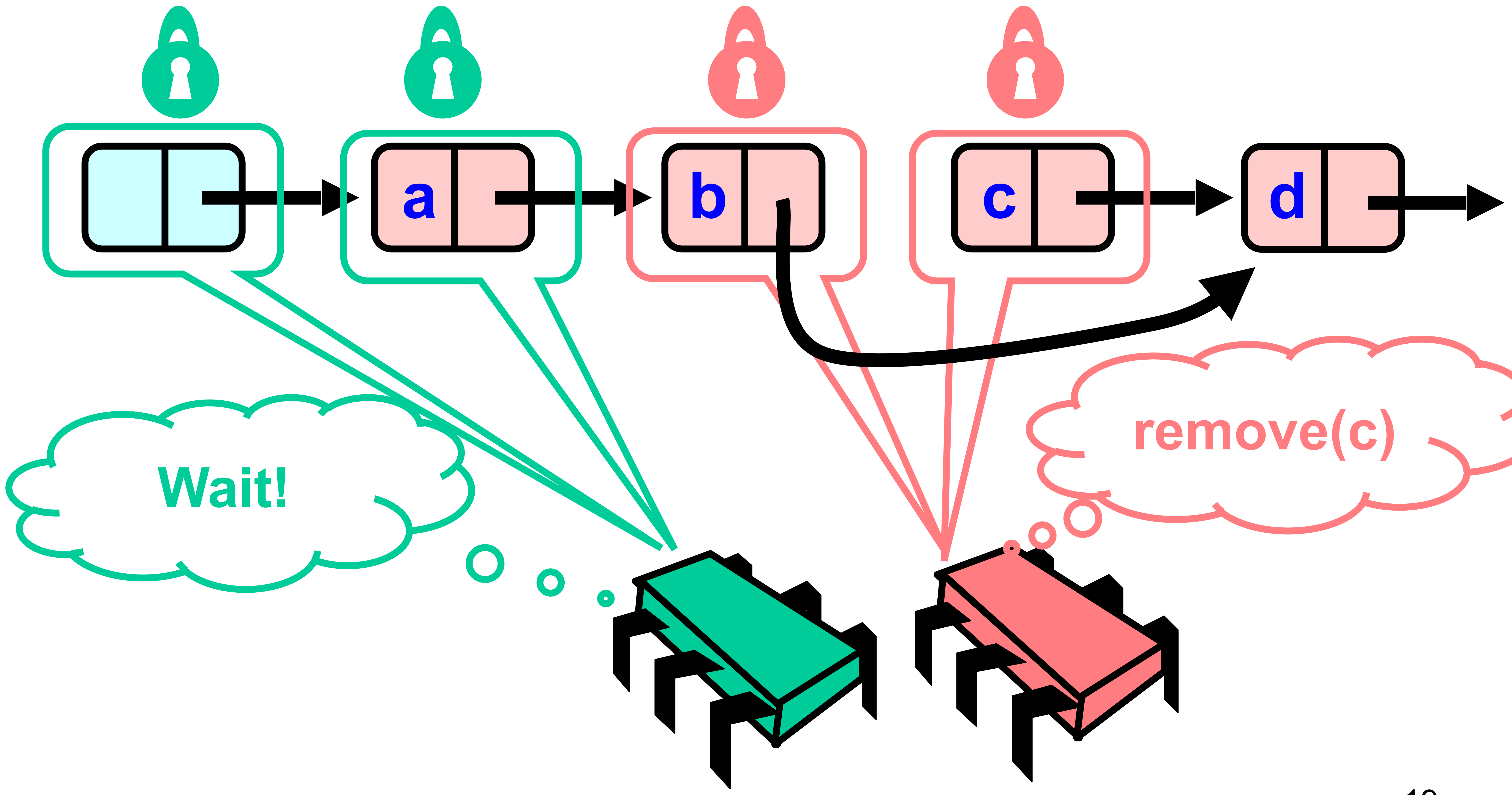
# Removing a Node



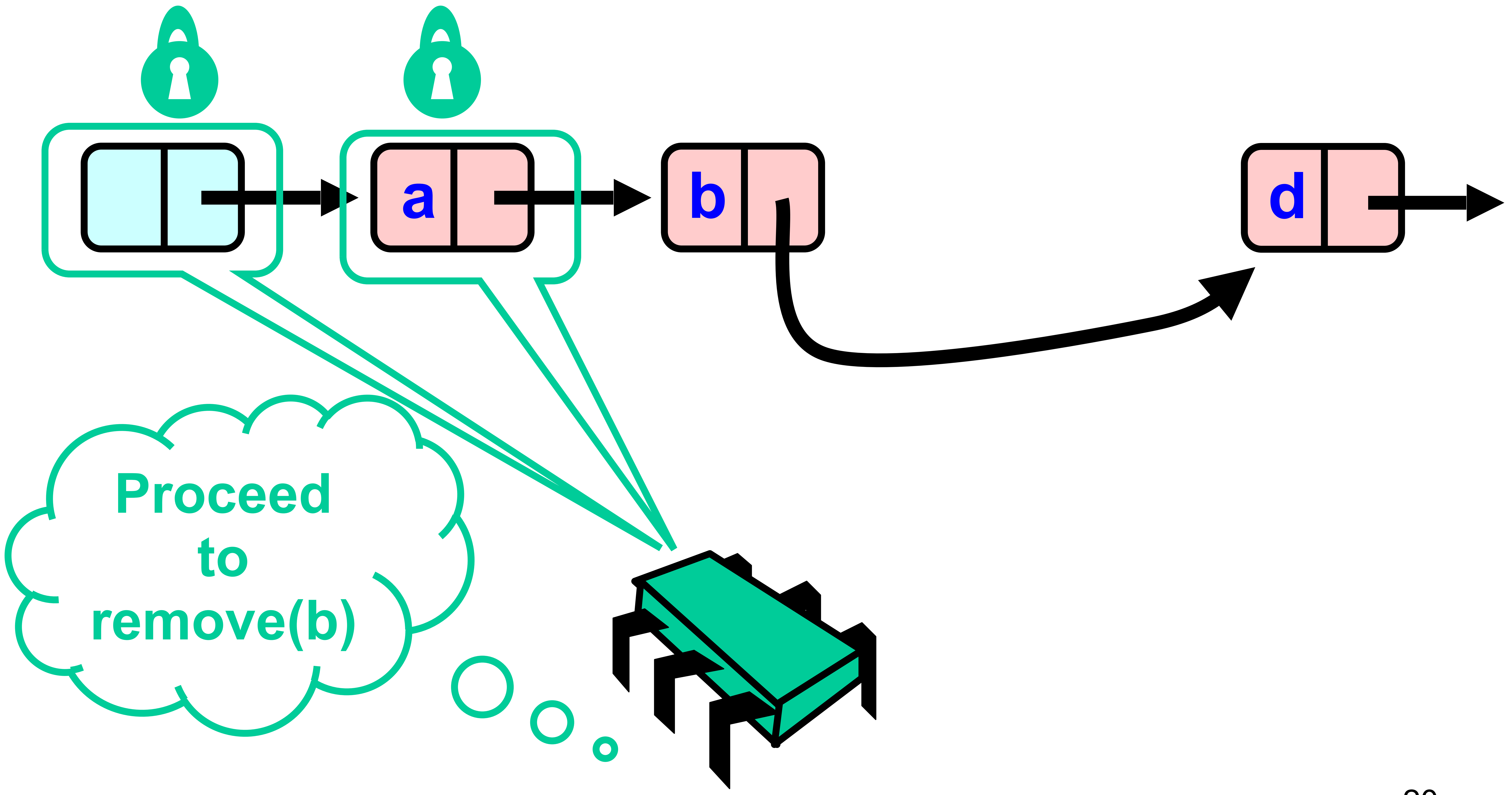
## Removing a Node



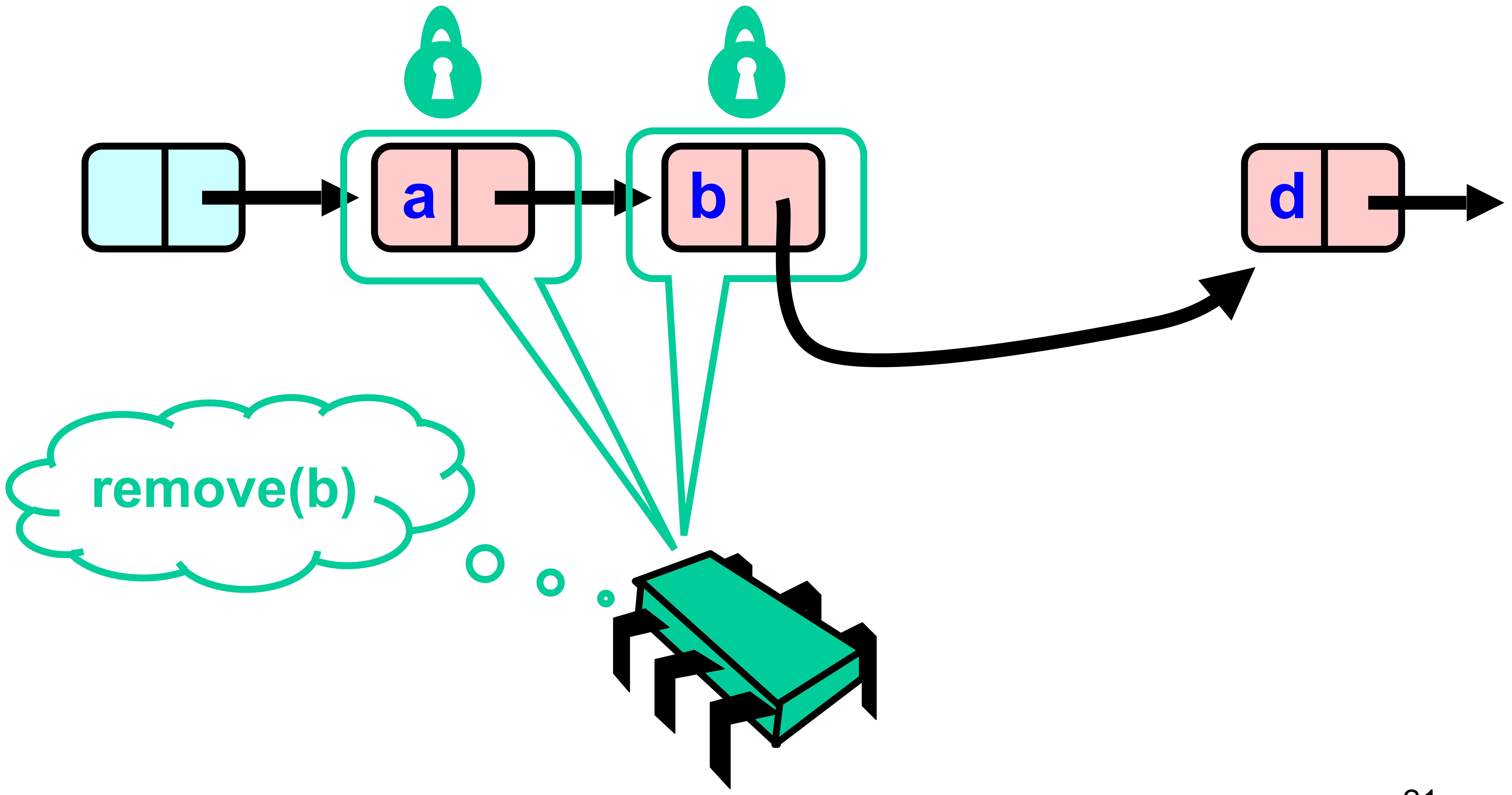
# Removing a Node



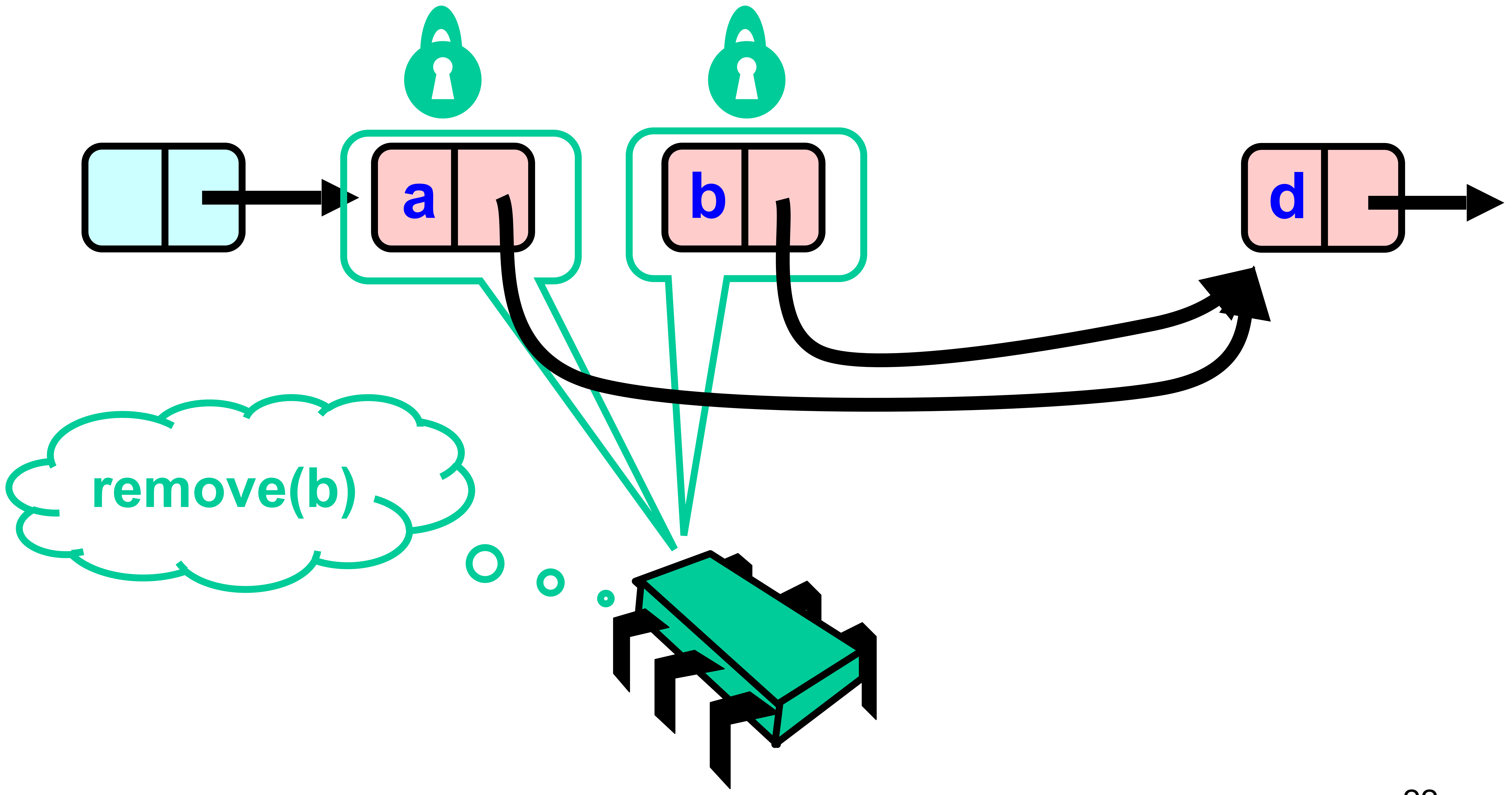
# Removing a Node



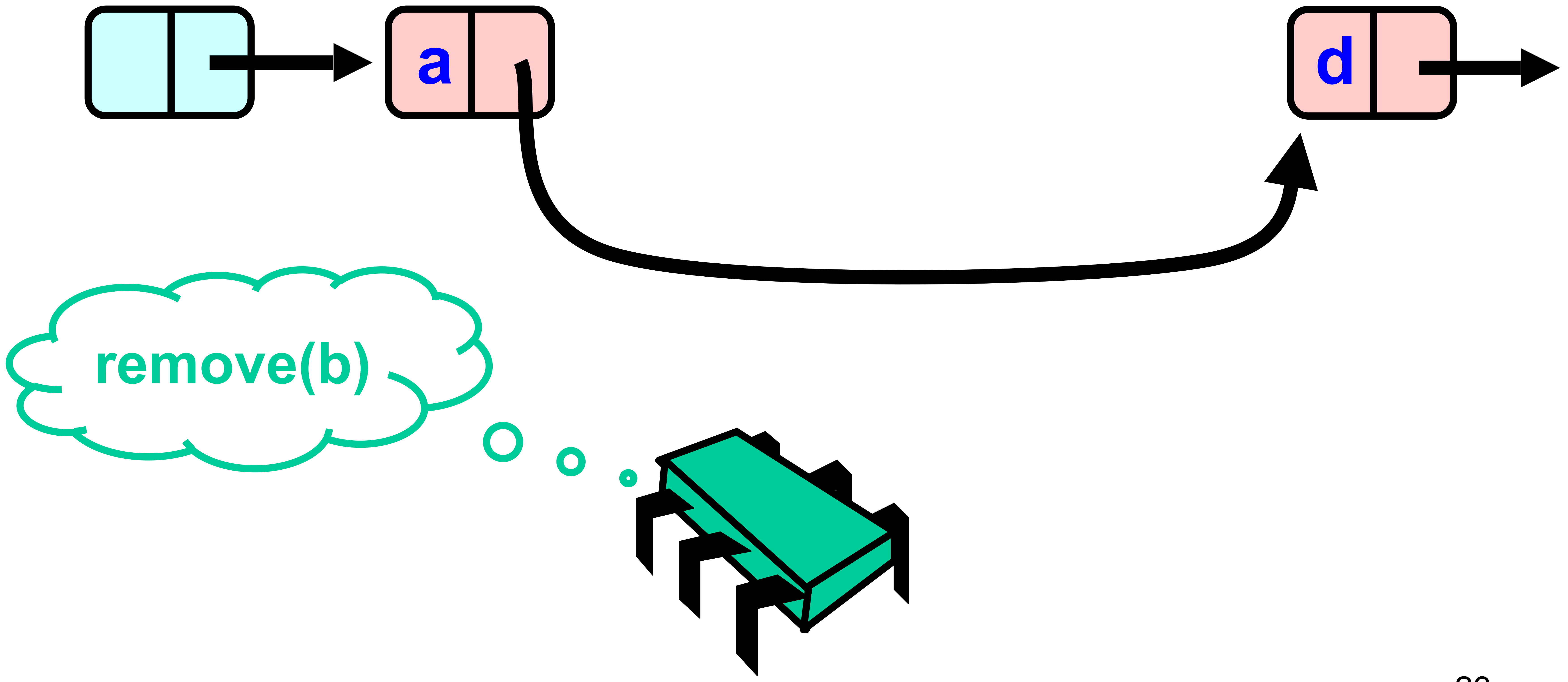
# Removing a Node



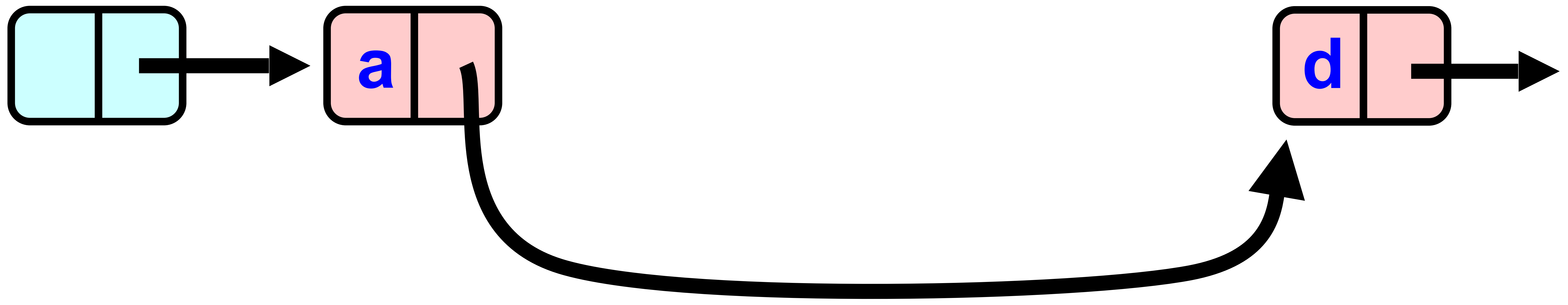
# Removing a Node



# 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

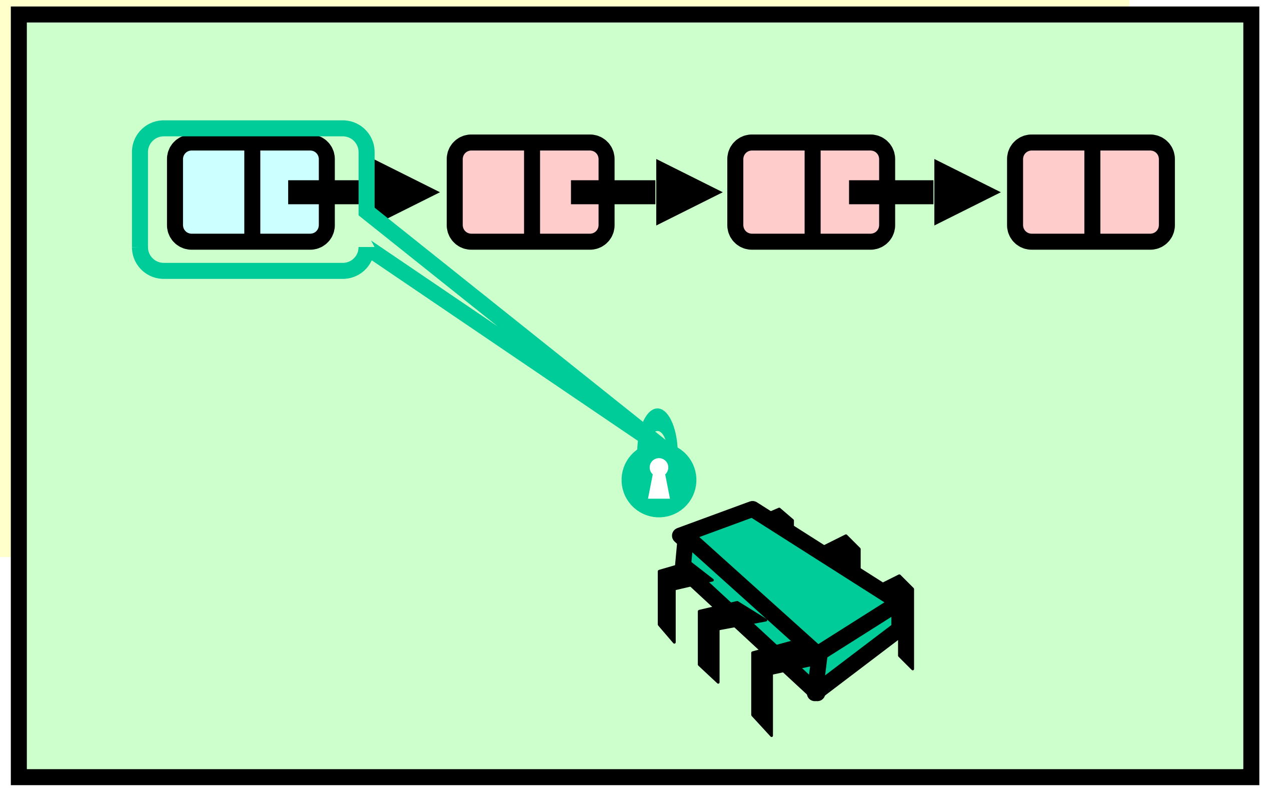
## Remove method

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

## Remove method

lock pred == head

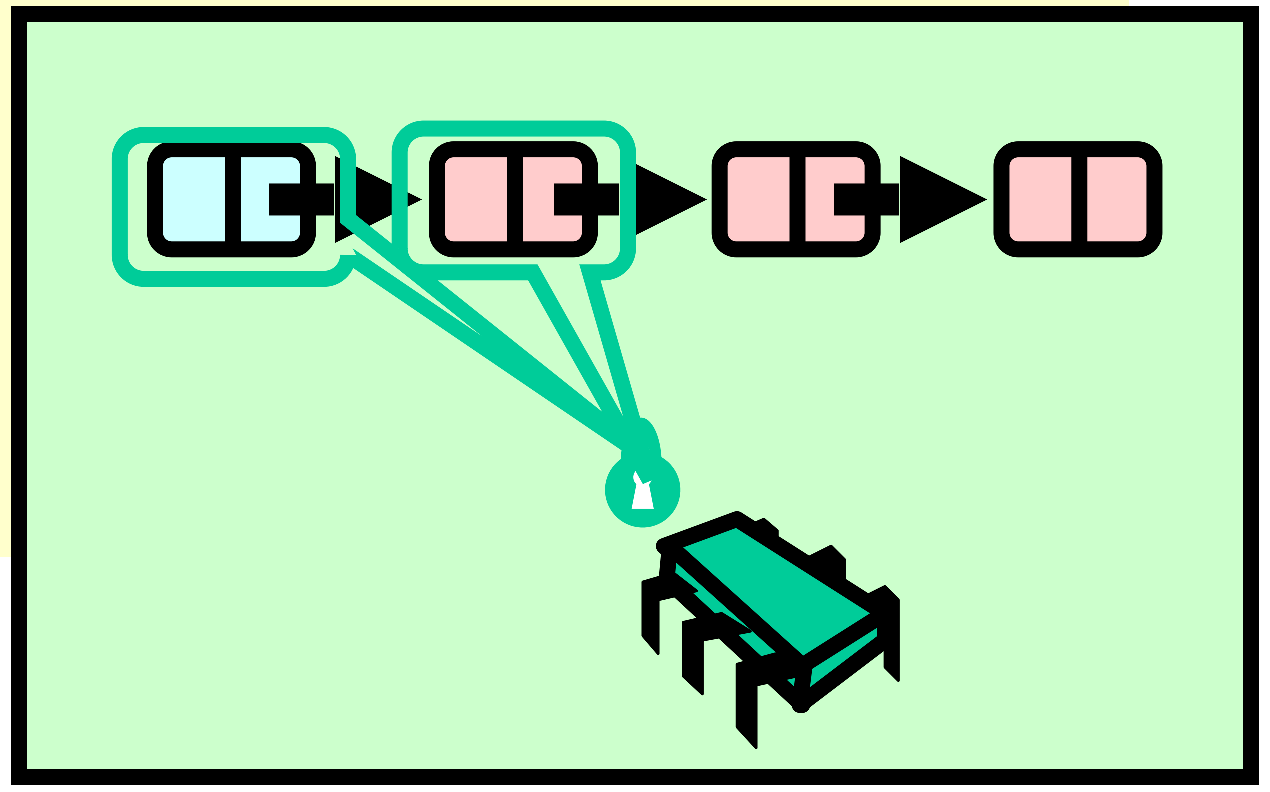
```
try {
 pred = head
 pred.lock()
 curr = pred.next
 curr.lock()
 …
} finally { … }
```



# Remove method

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

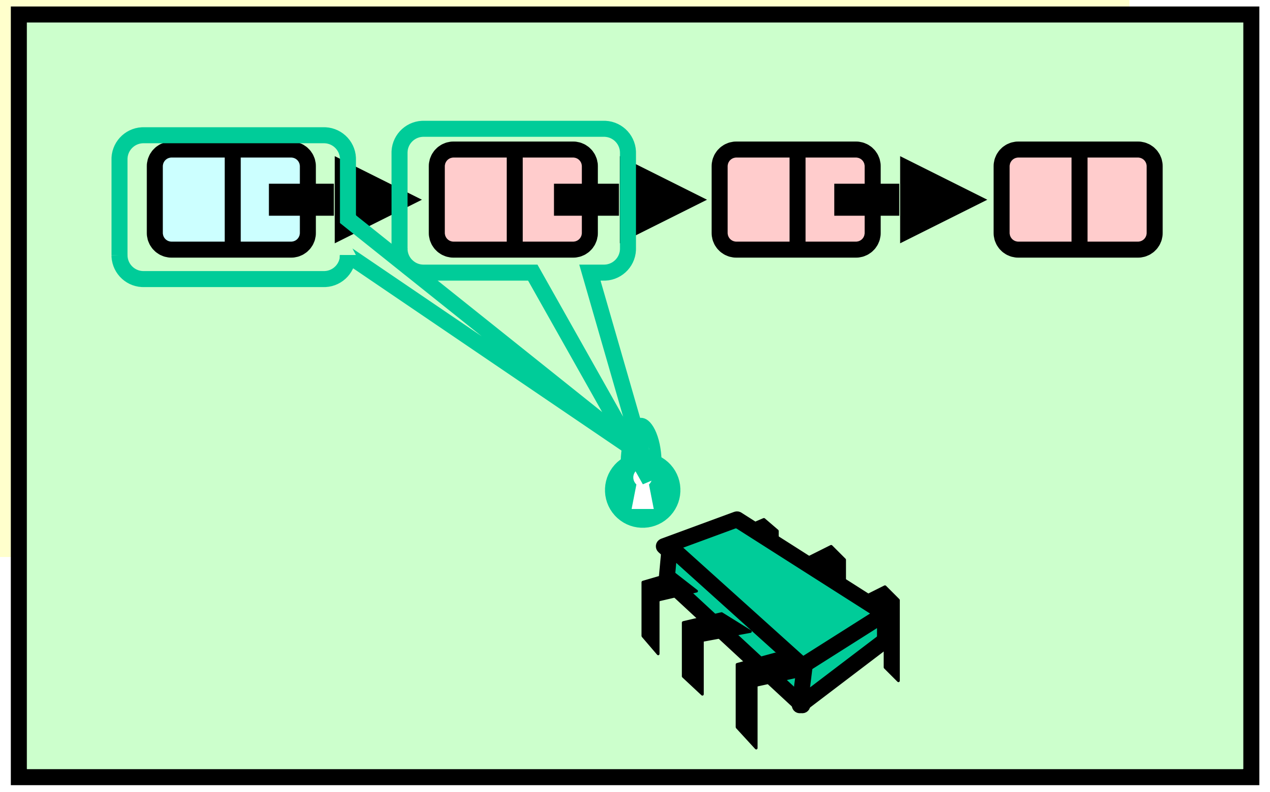
Lock current



## Remove method

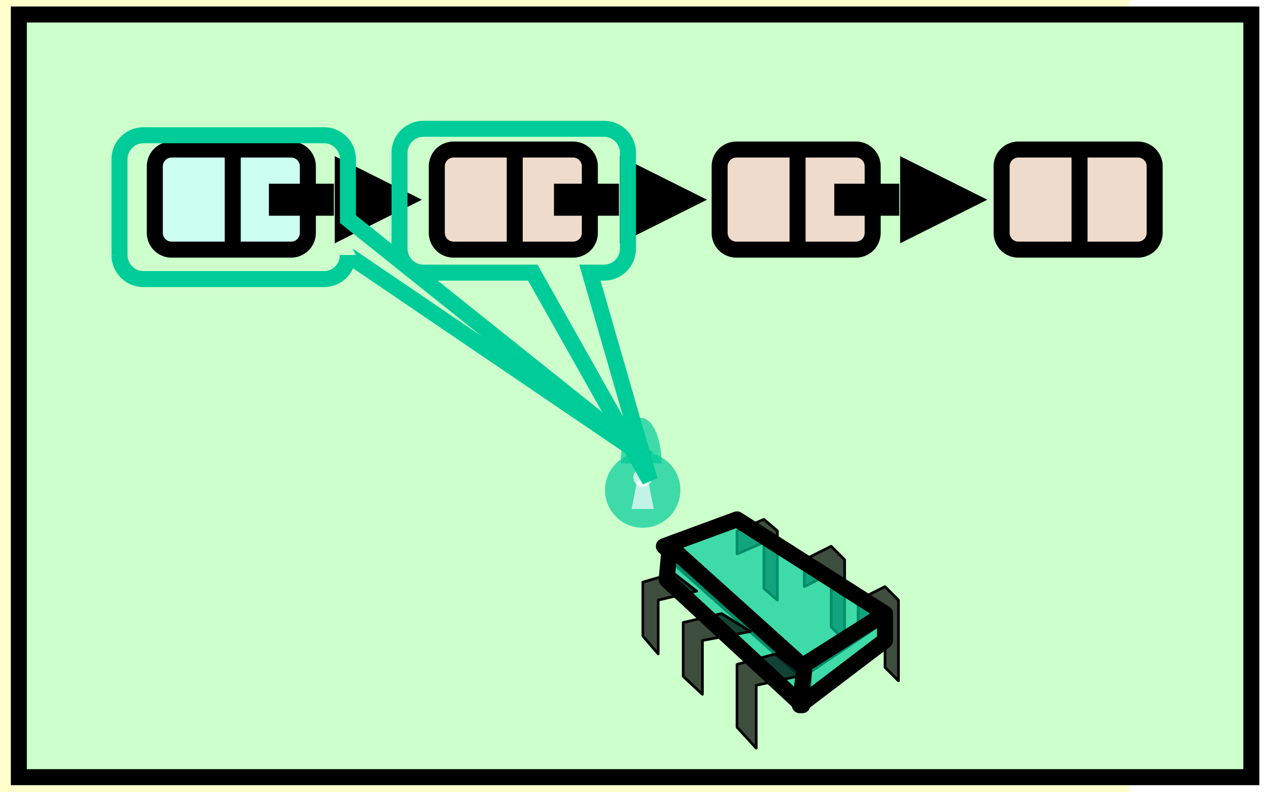
```
try {
 pred = head
 pred.lock()
 curr = pred.next
 curr.lock()
 …
} finally { … }
```

Traversing list



# Remove: searching

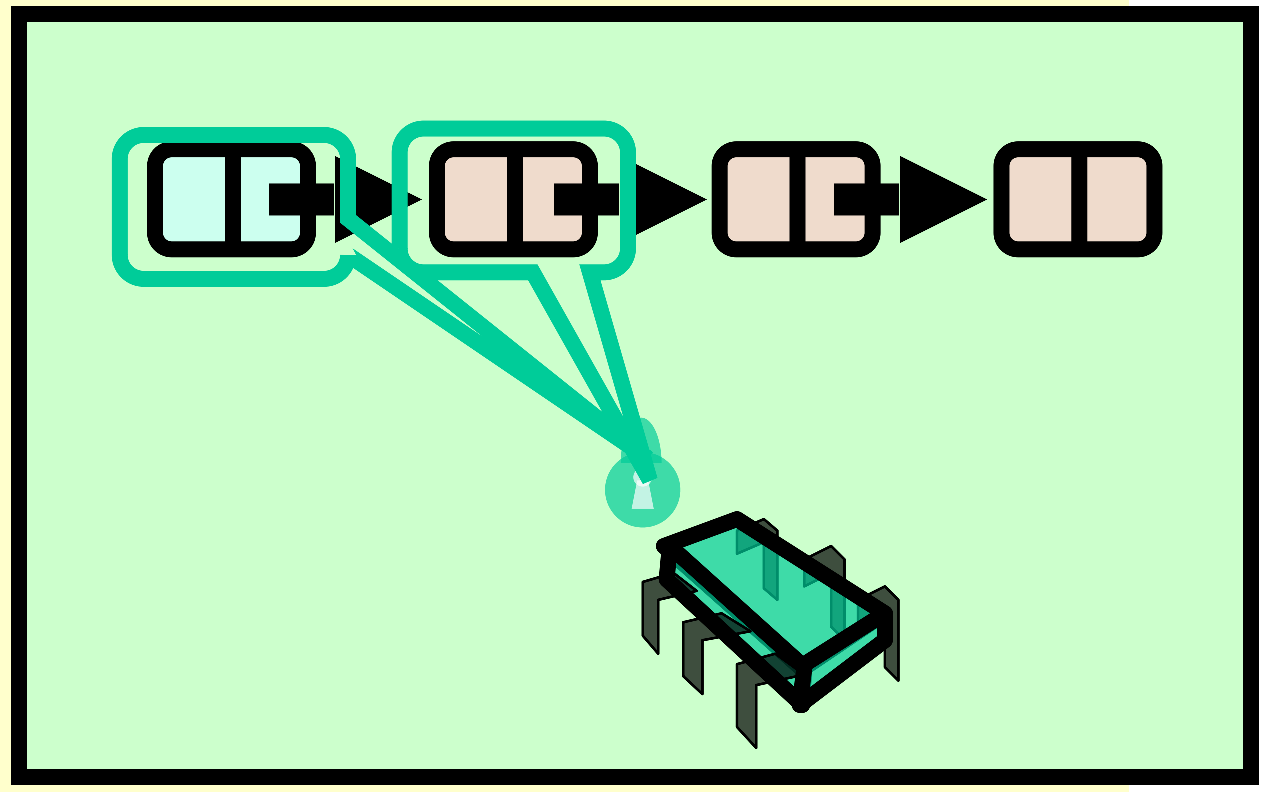
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next
   return true
  }
  pred.unlock()
  pred = curr
  curr = curr.next
  curr.lock()
 }
 return false
```



## 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
```

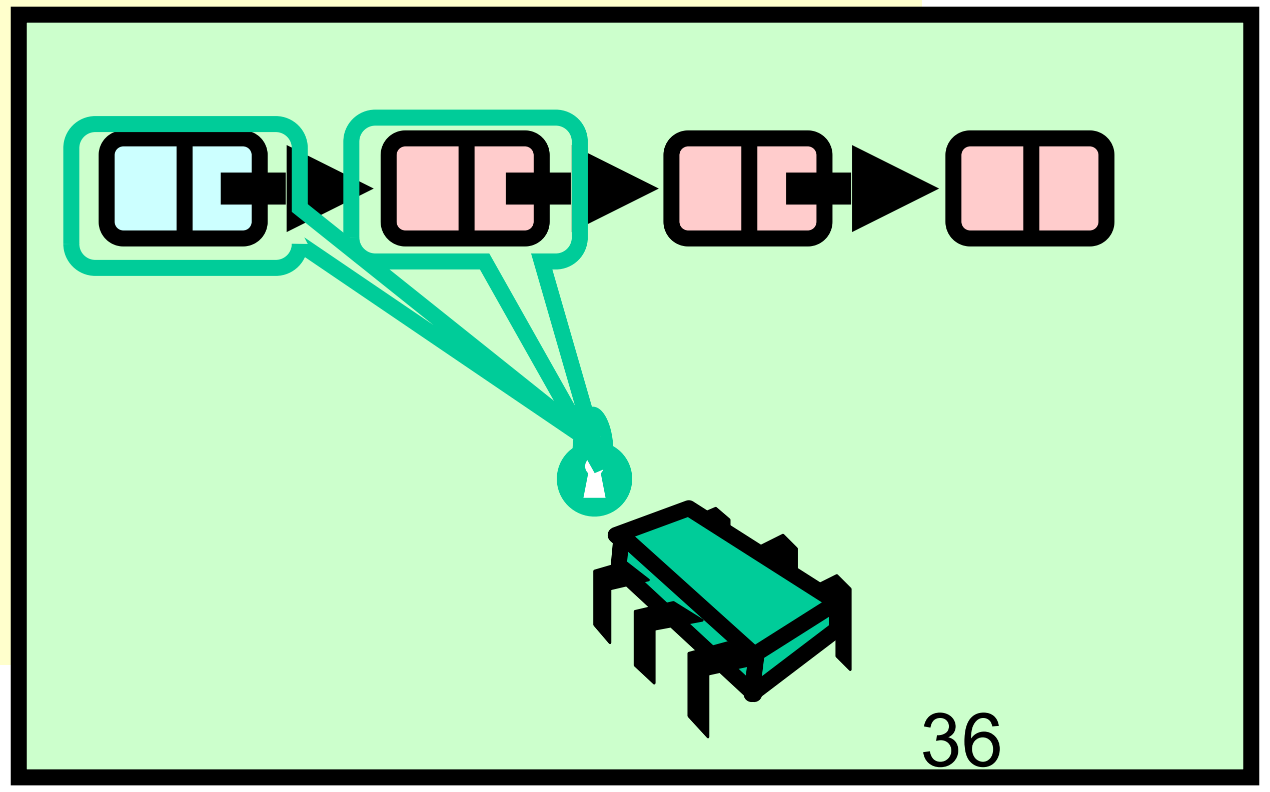
Search key range



# 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
```

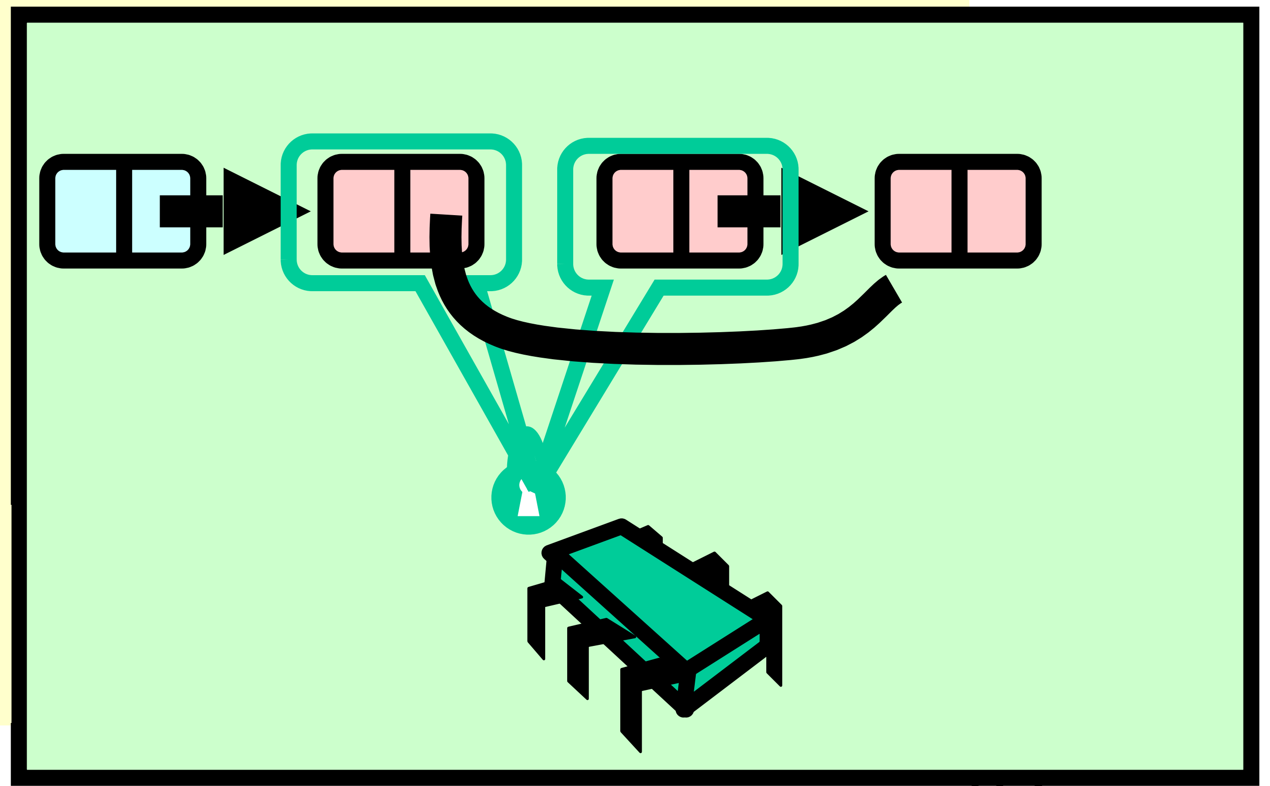
**At start of each loop: curr and pred locked**



# 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
```

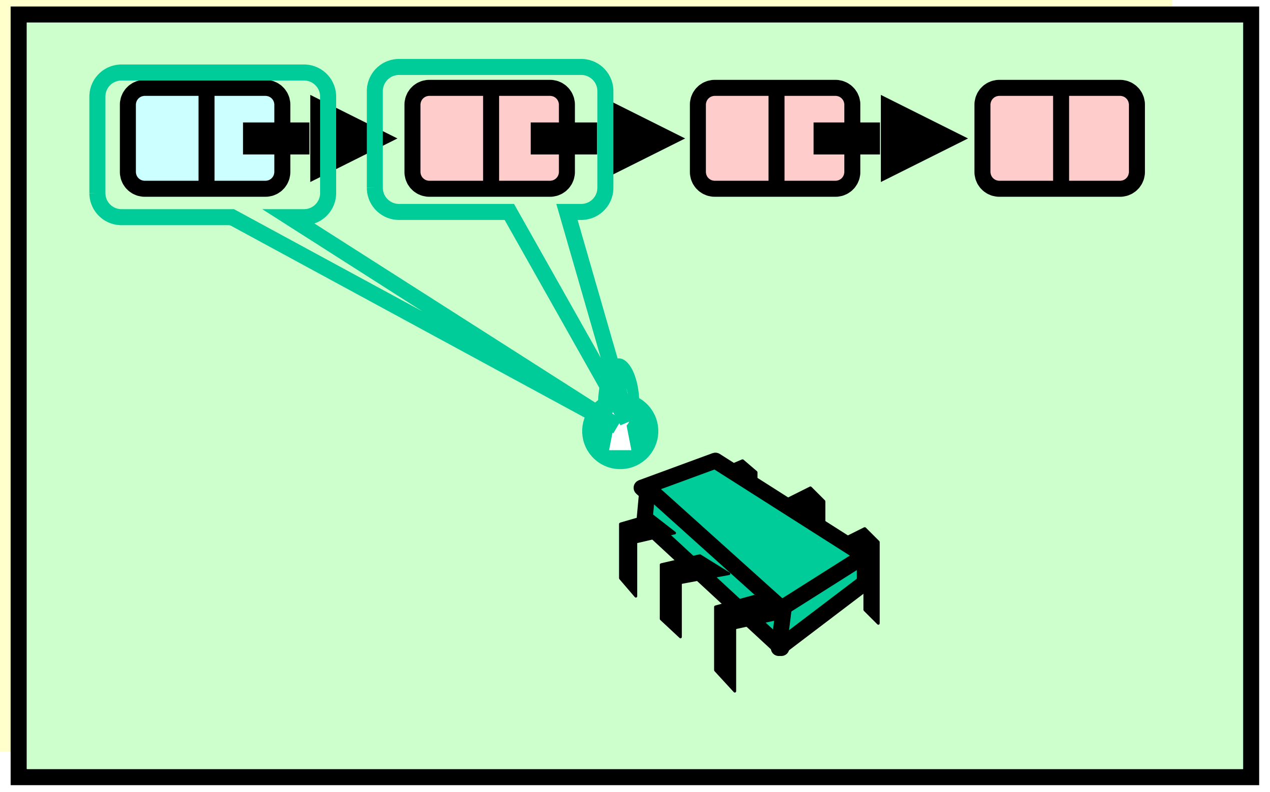
If item found, remove node



# Remove: searching

**Unlock predecessor**

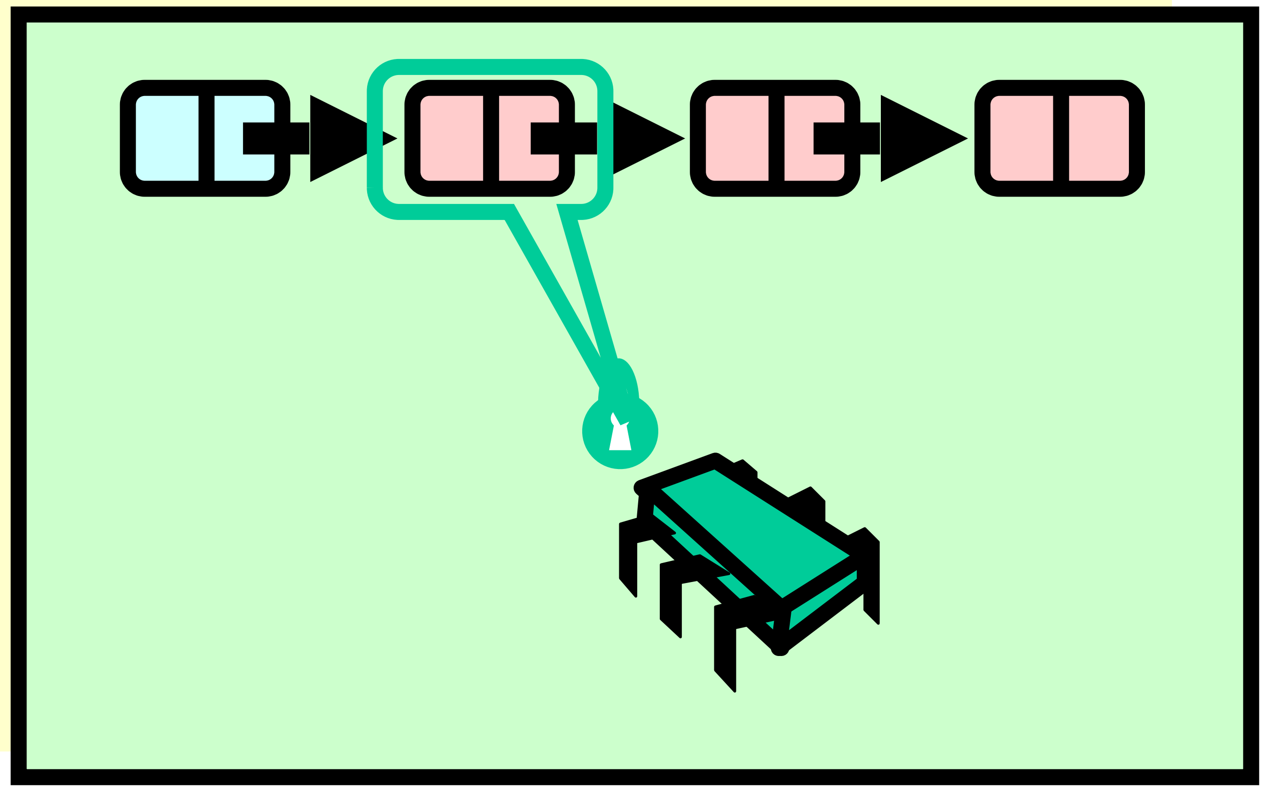
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next
   return true
  }
  pred.unlock()
  pred = curr
  curr = curr.next
  curr.lock()
 }
 return false
```



## Remove: searching

Only one node locked!

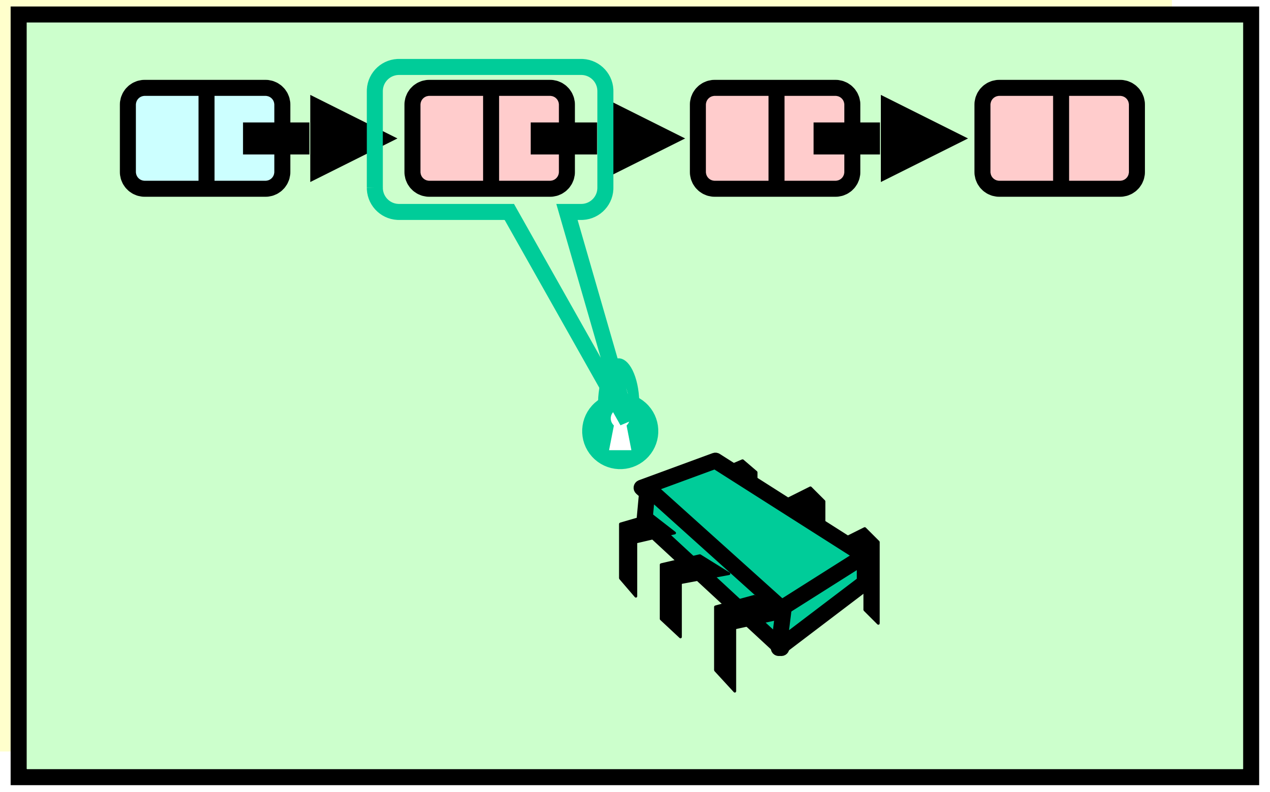
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next
   return true
  }
  pred.unlock()
  pred = curr
  curr = curr.next
  curr.lock()
 }
 return false
```



# Remove: searching

demote current

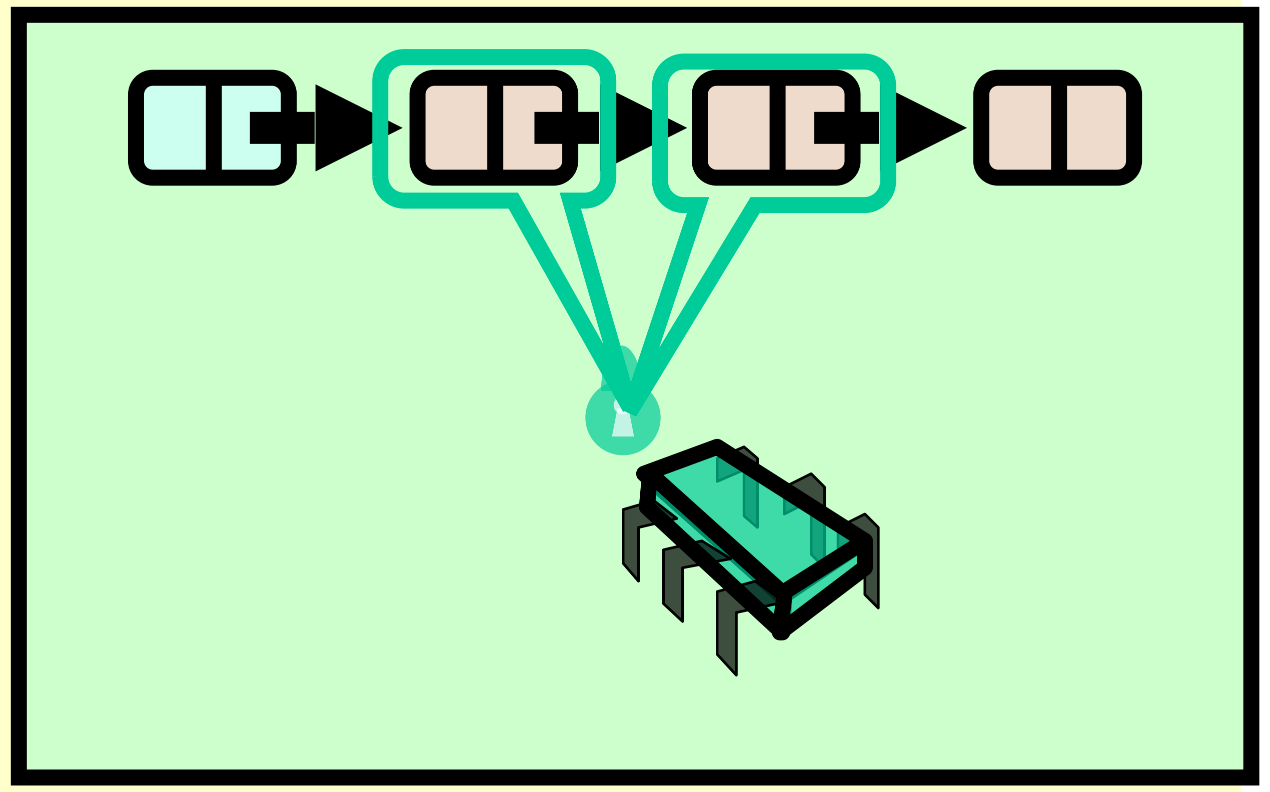
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next
   return true
  }
  pred.unlock()
  pred = curr
  curr = curr.next
  curr.lock()
 }
 return false
```



## Remove: searching

Find and lock new current

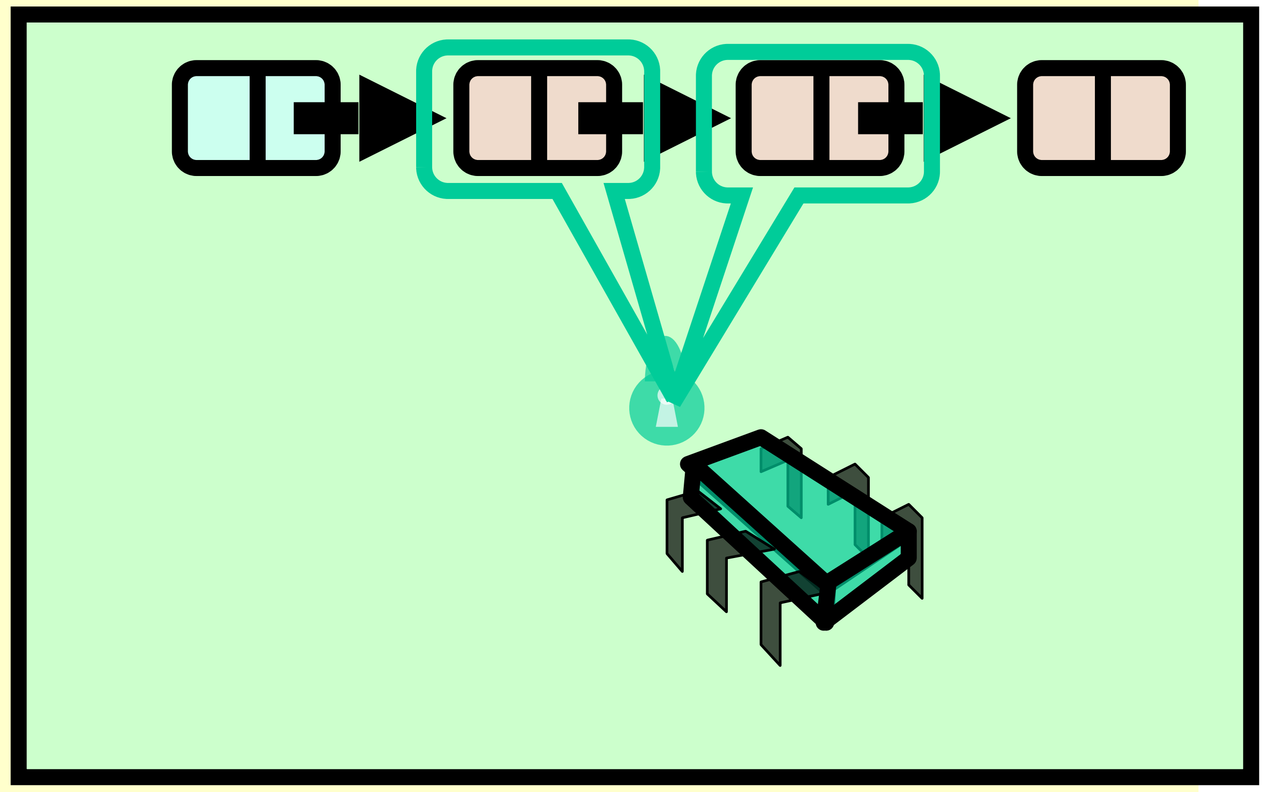
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next
   return true
  }
  pred.unlock()
  pred = currNode
  curr = curr.next
  curr.lock()
 }
 return false
```



# 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()
 }
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

# 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(head) =

  { x | there exists a such that
  - a reachable from head and
  - a.item = x

  }

## Rep Invariant

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

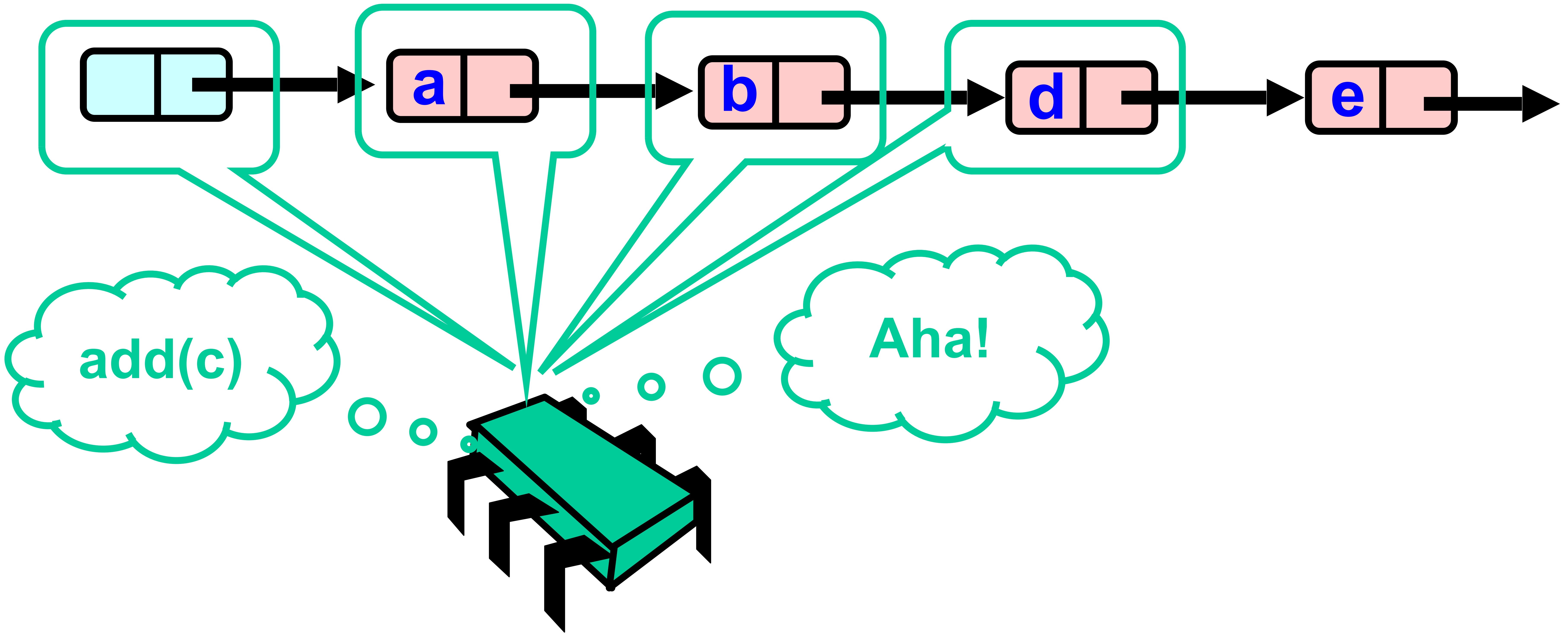
## Drawbacks

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

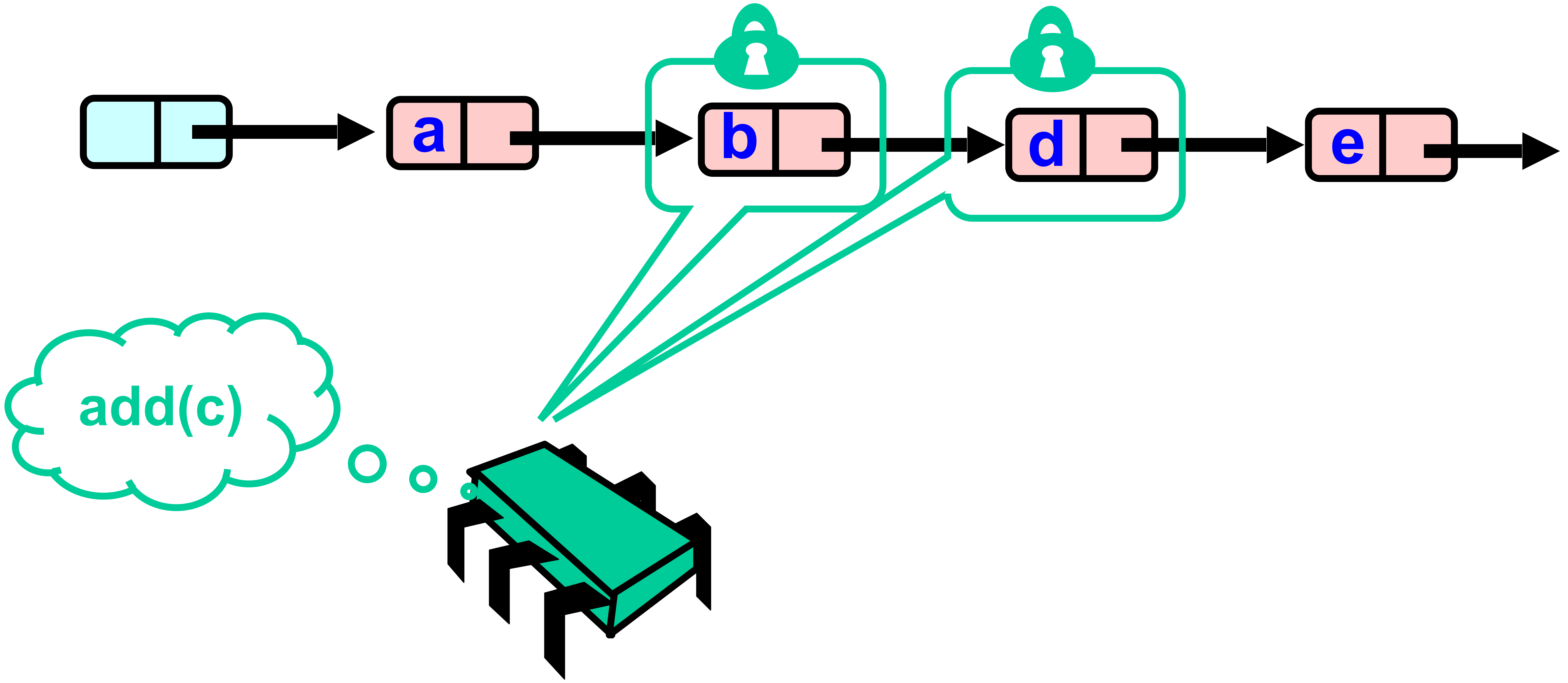
## Optimistic Synchronization

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

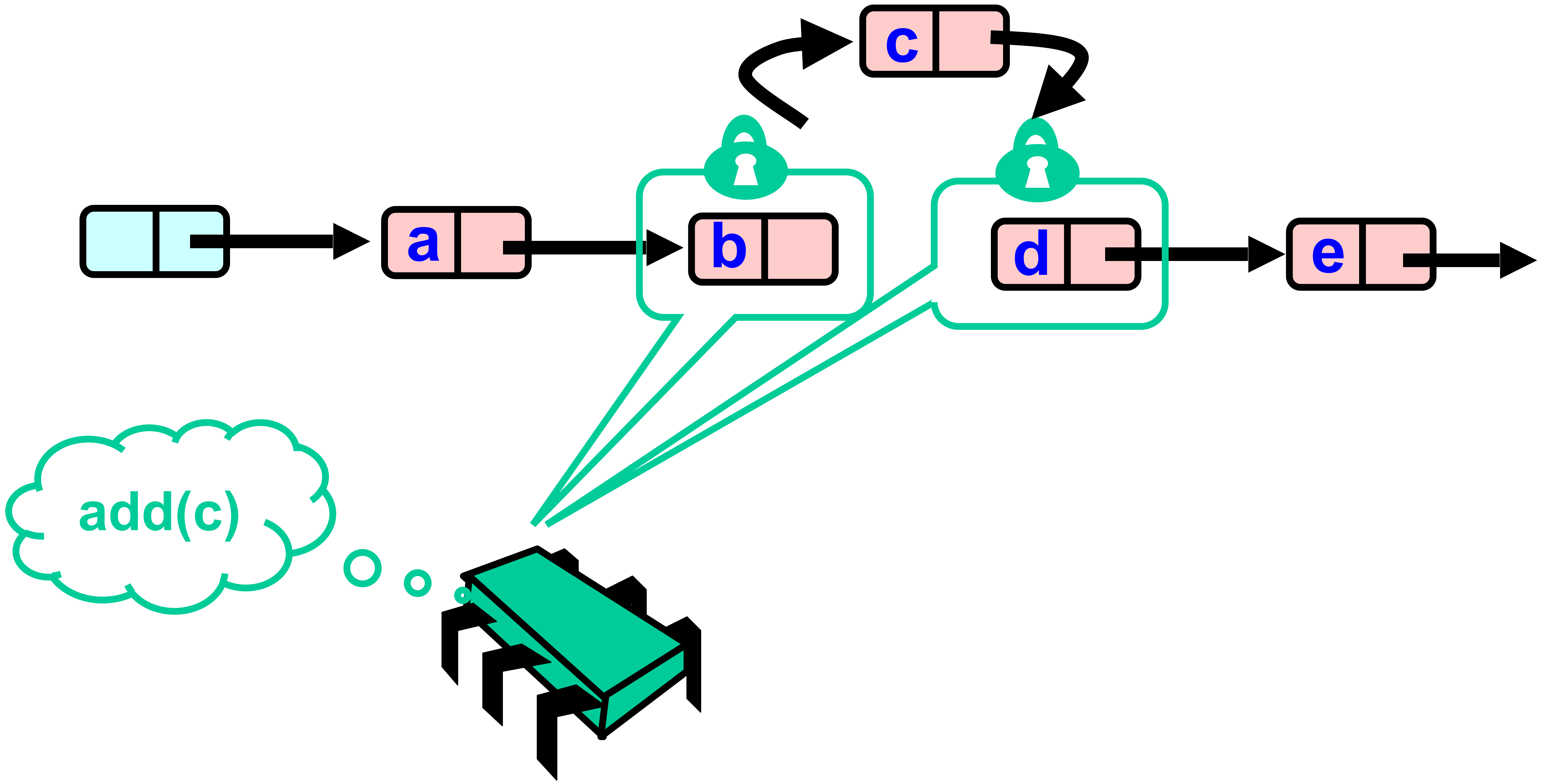
## Optimistic: Traverse without Locking



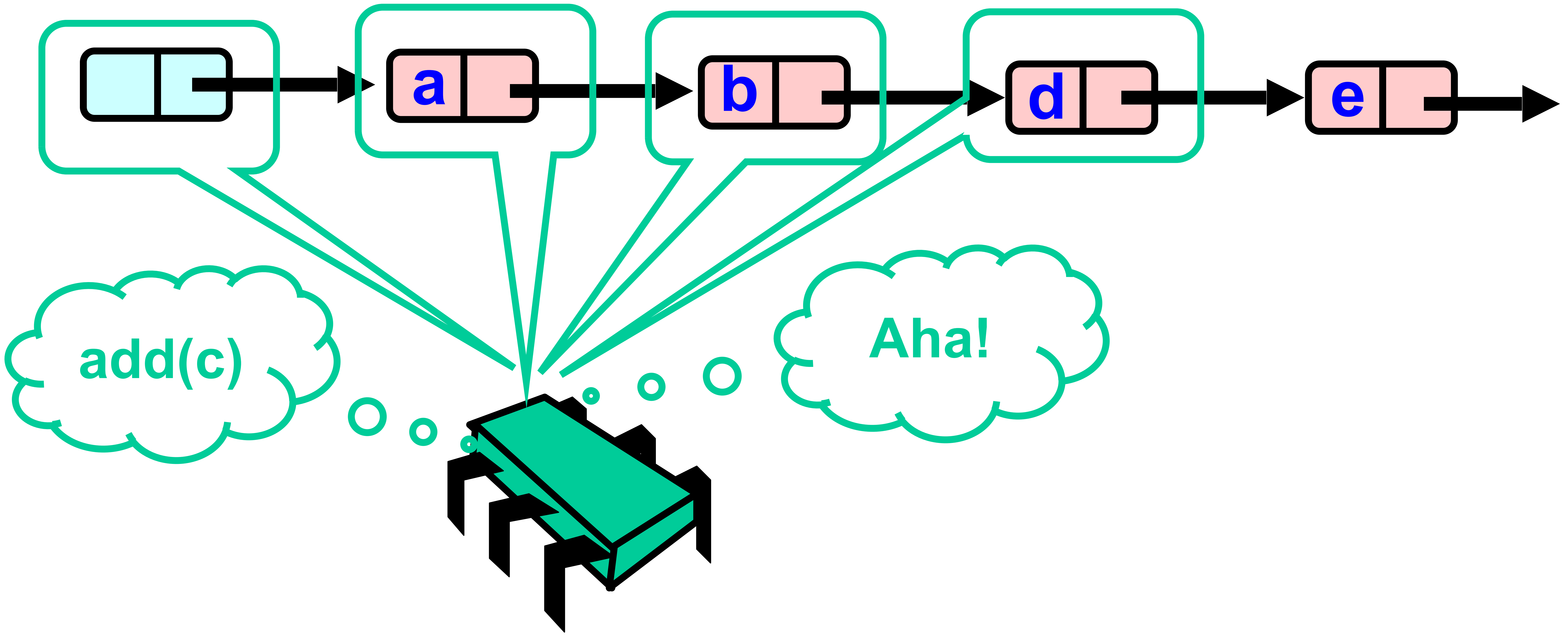
# Optimistic: Lock and Load



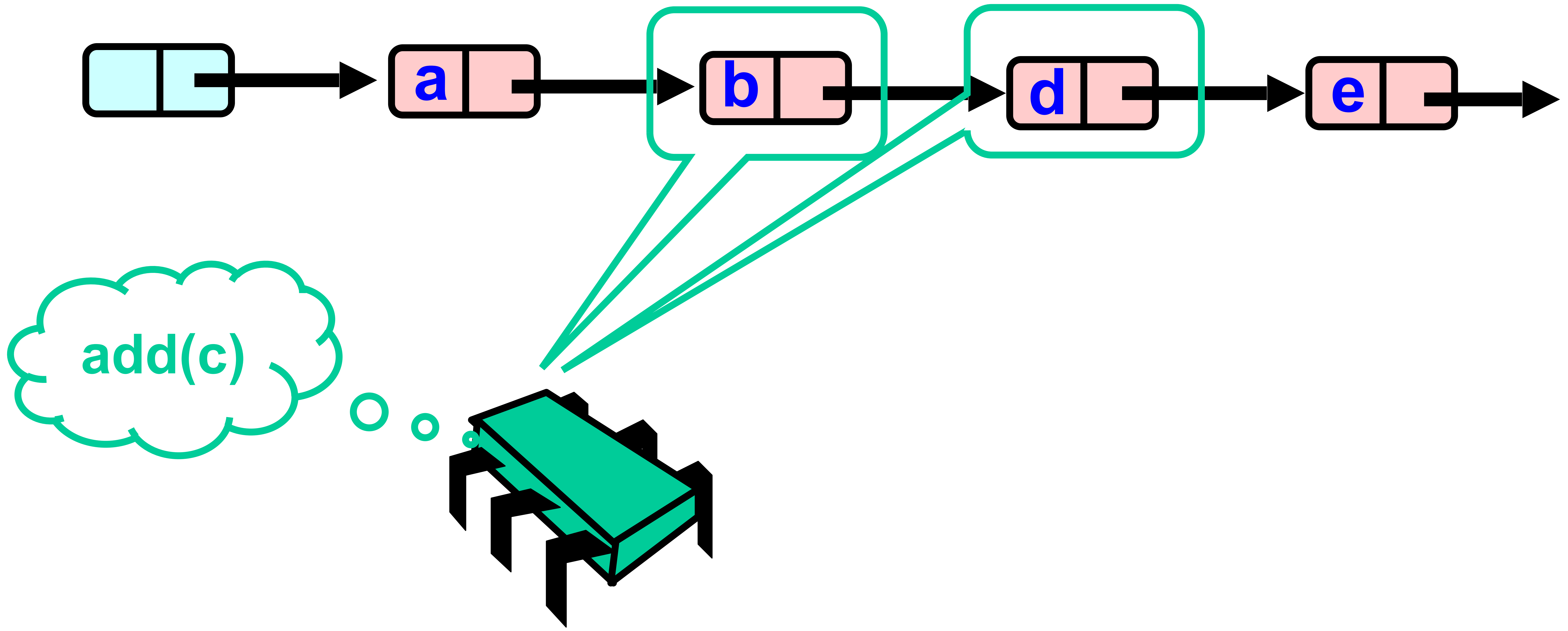
## Optimistic: Lock and Load



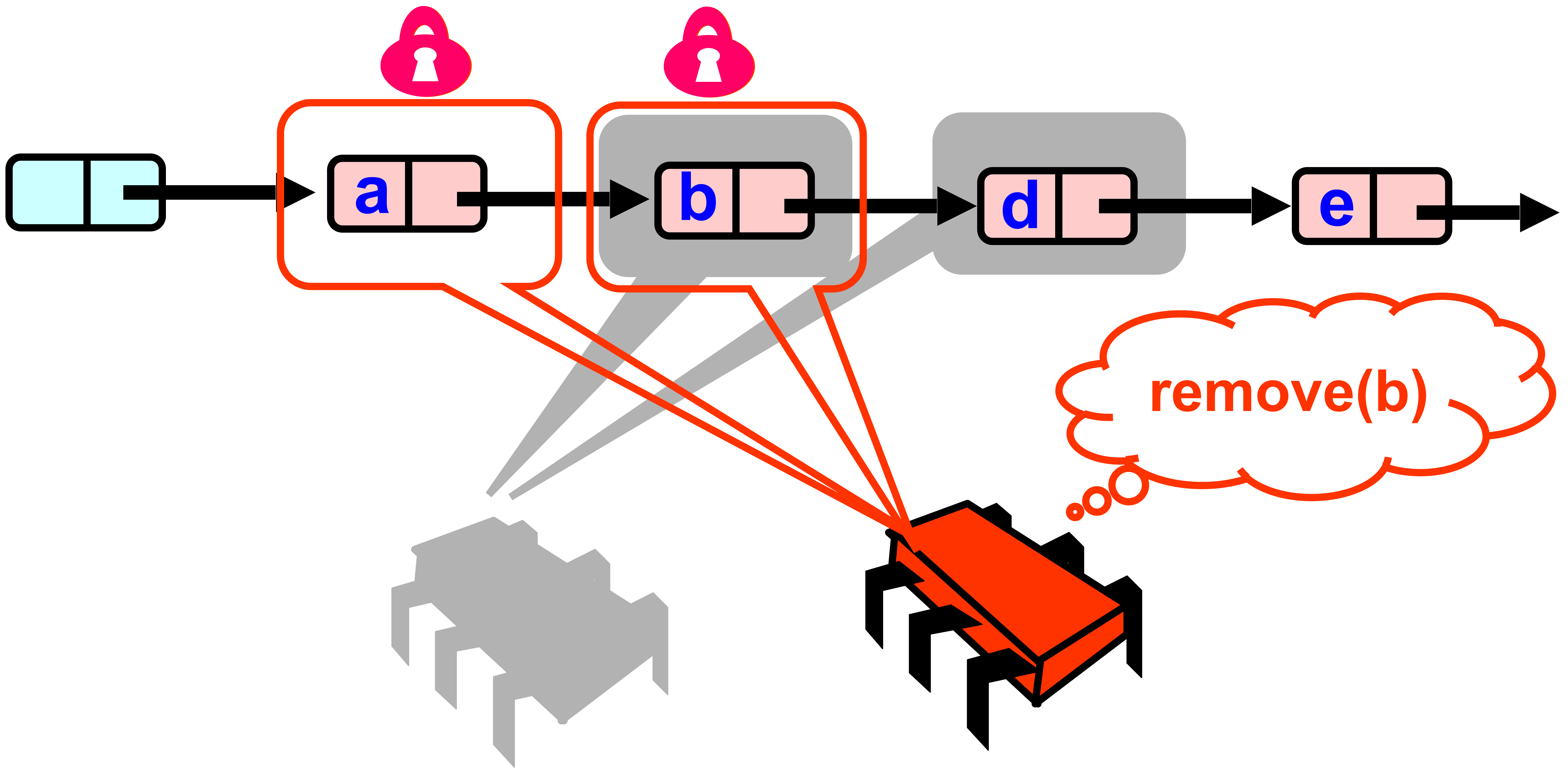
## What could go wrong?



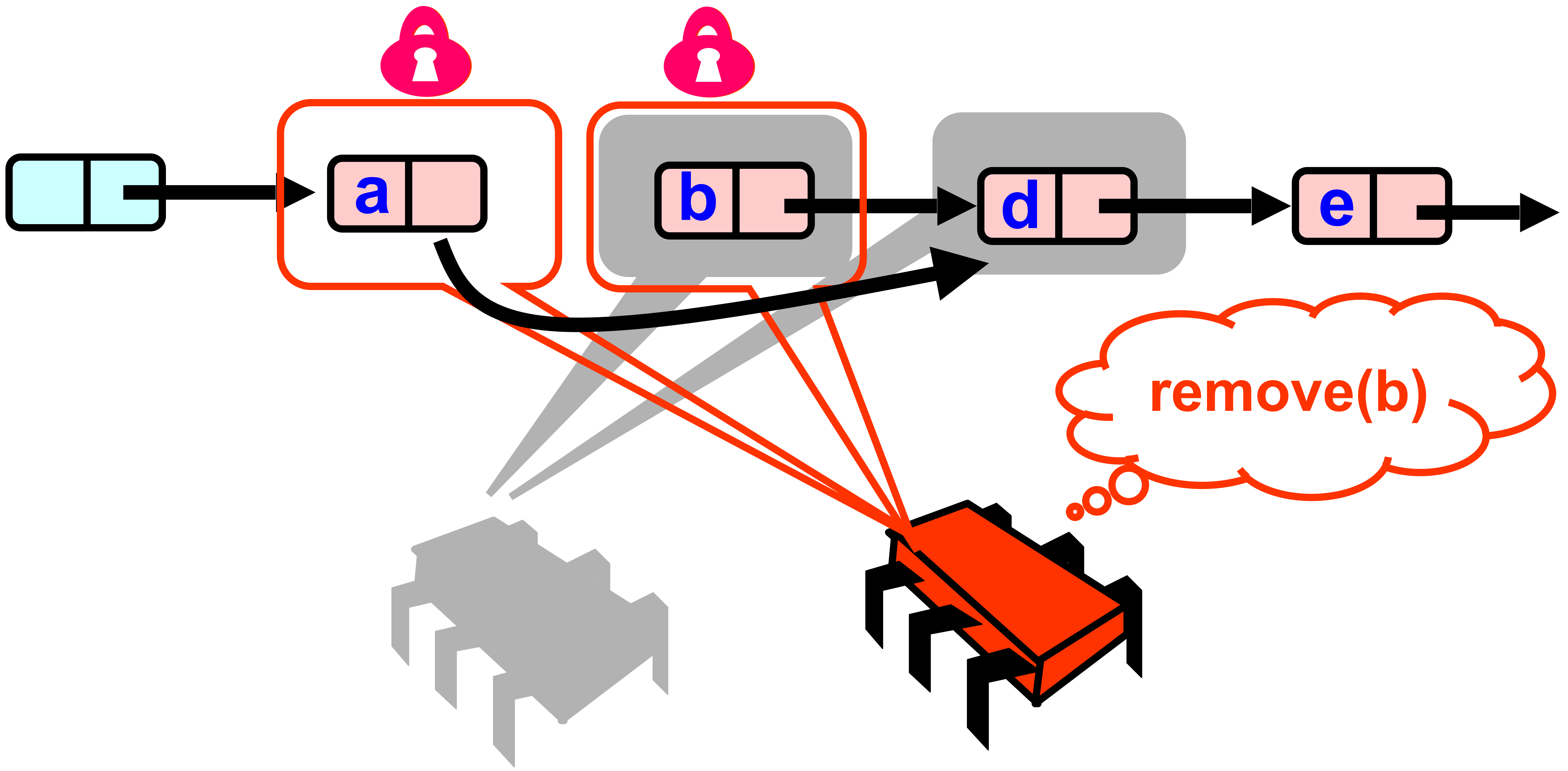
## What could go wrong?



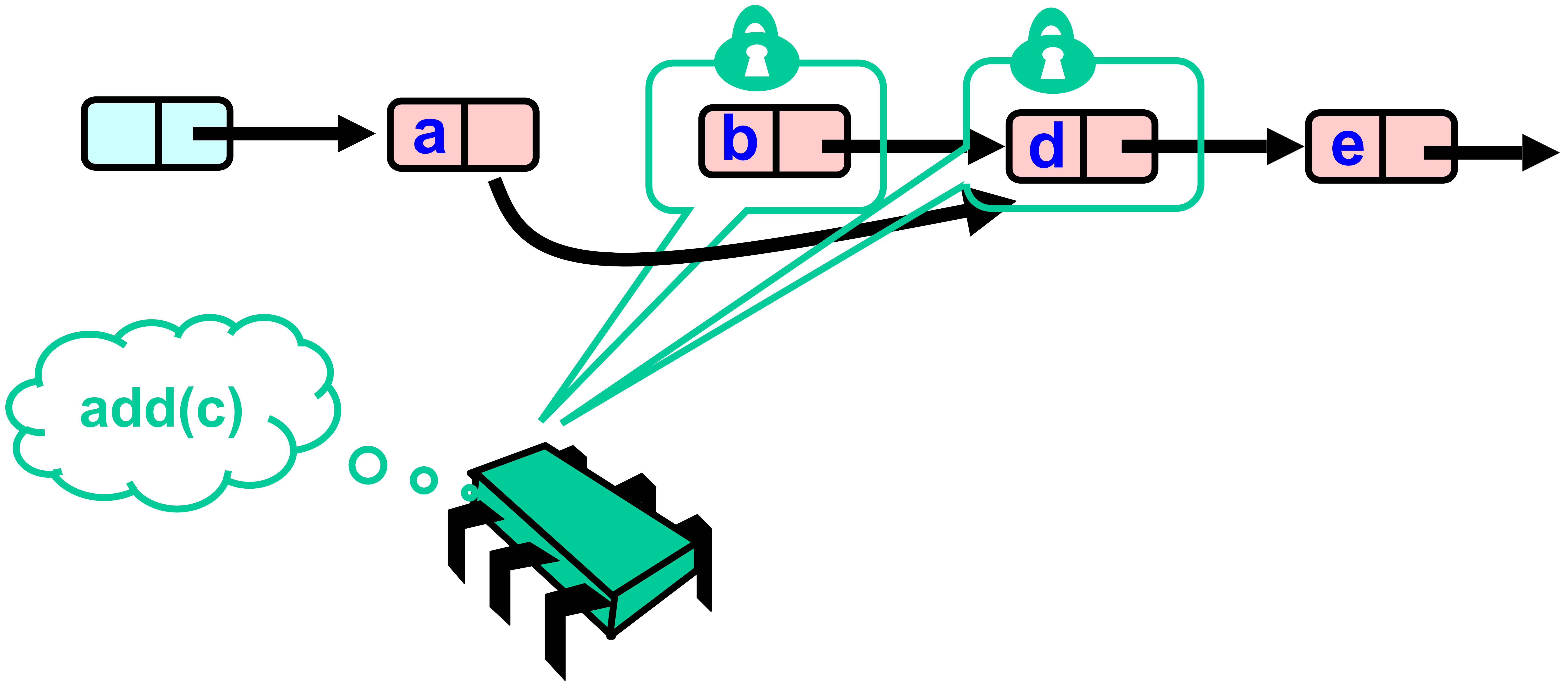
## What could go wrong?



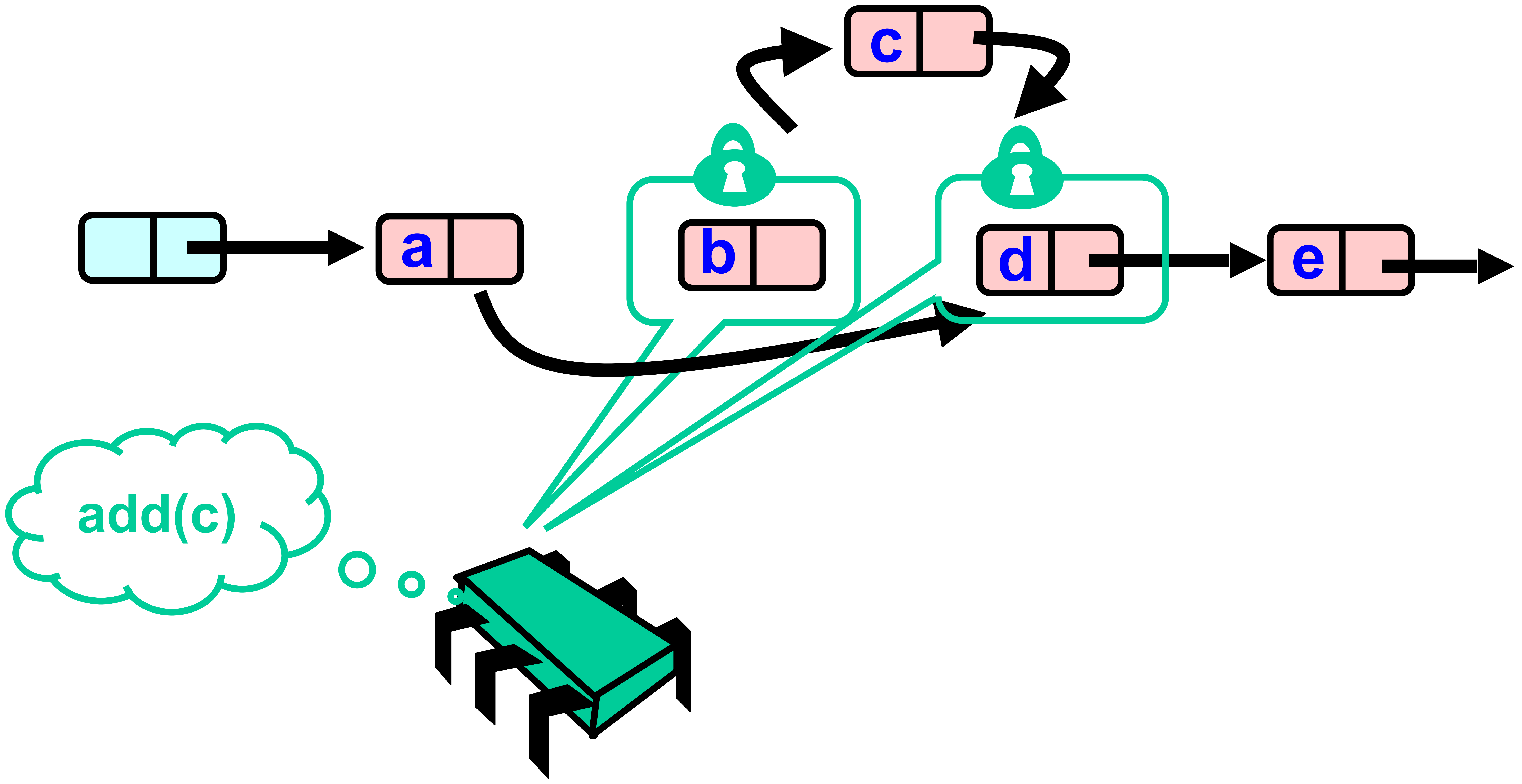
# What could go wrong?



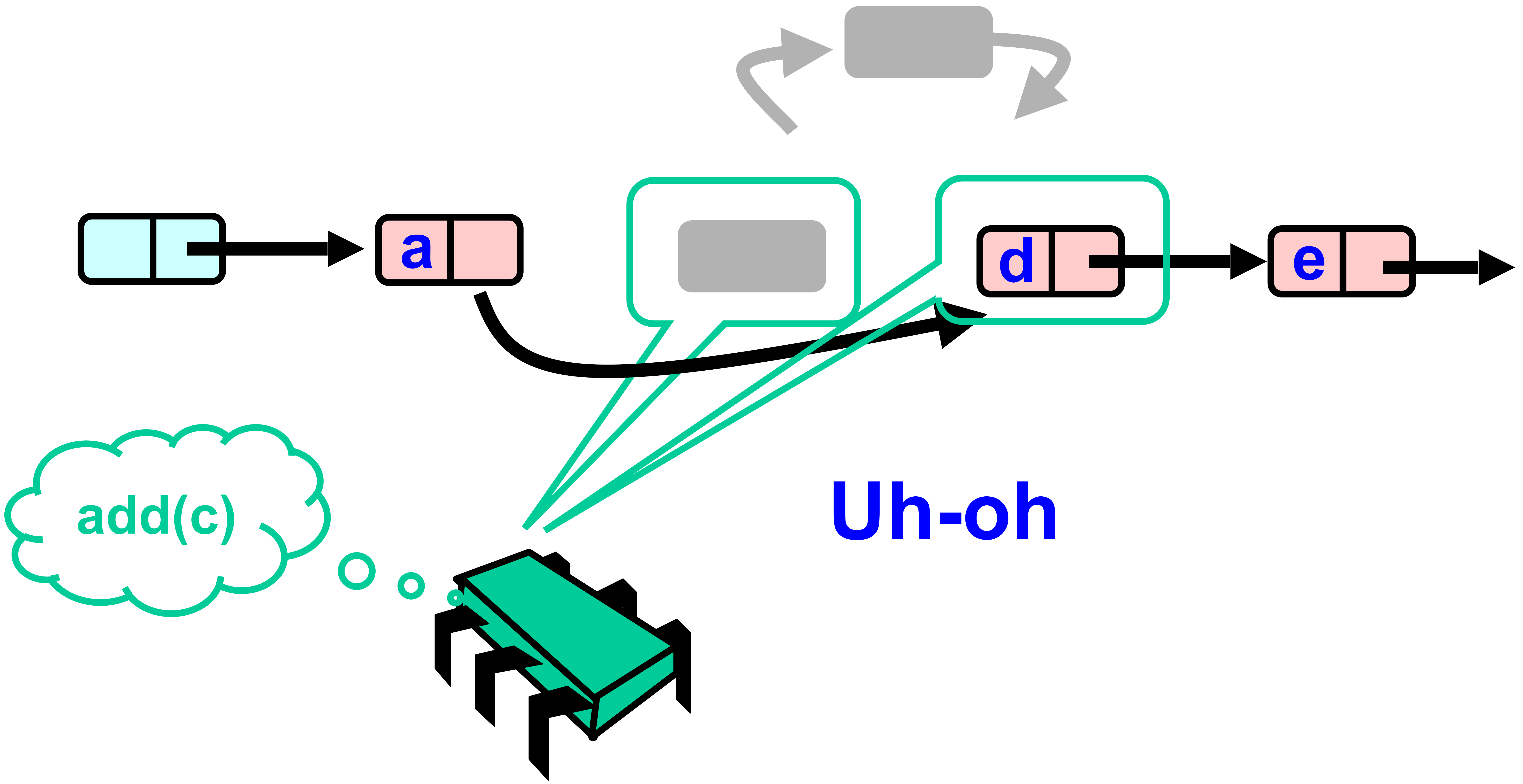
# What could go wrong?



## What could go wrong?

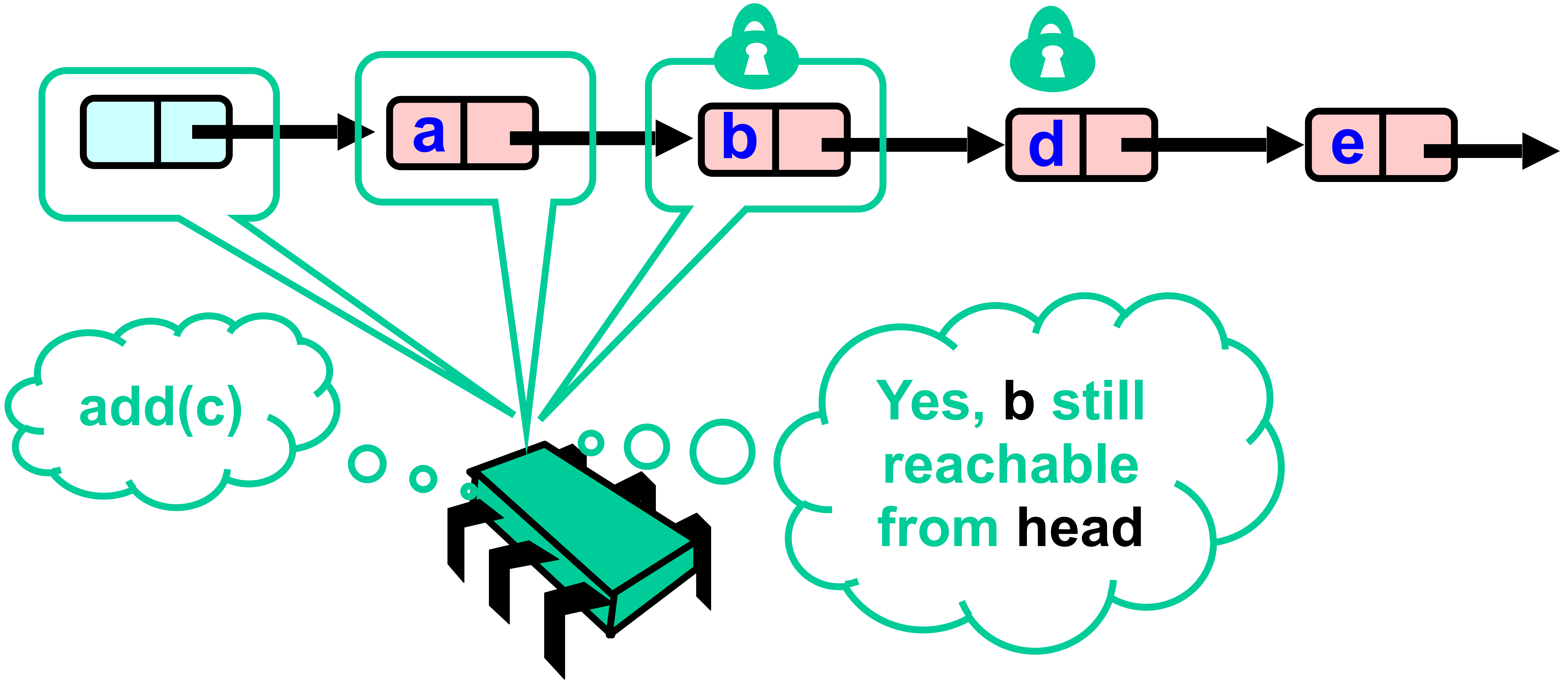


# What could go wrong?

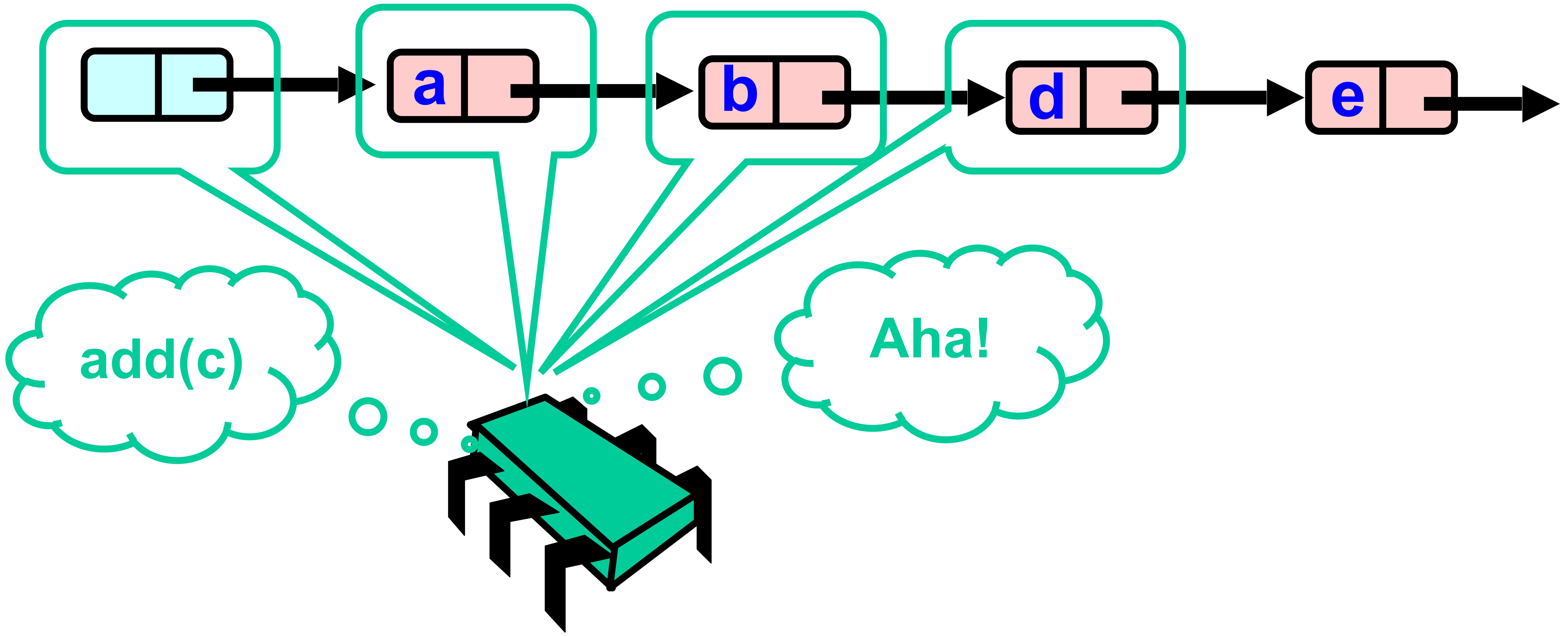


Uh-oh

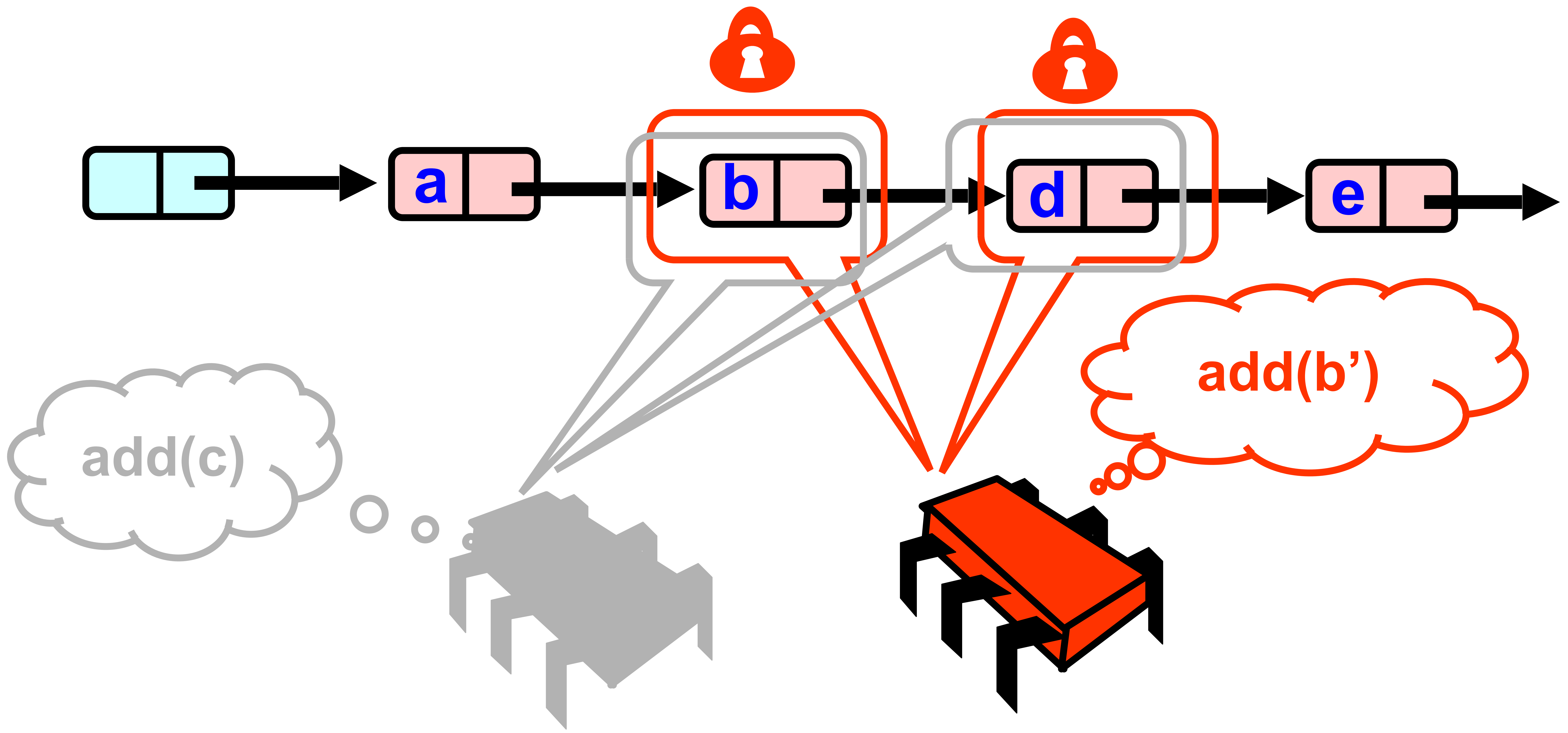
# Validate – Part 1



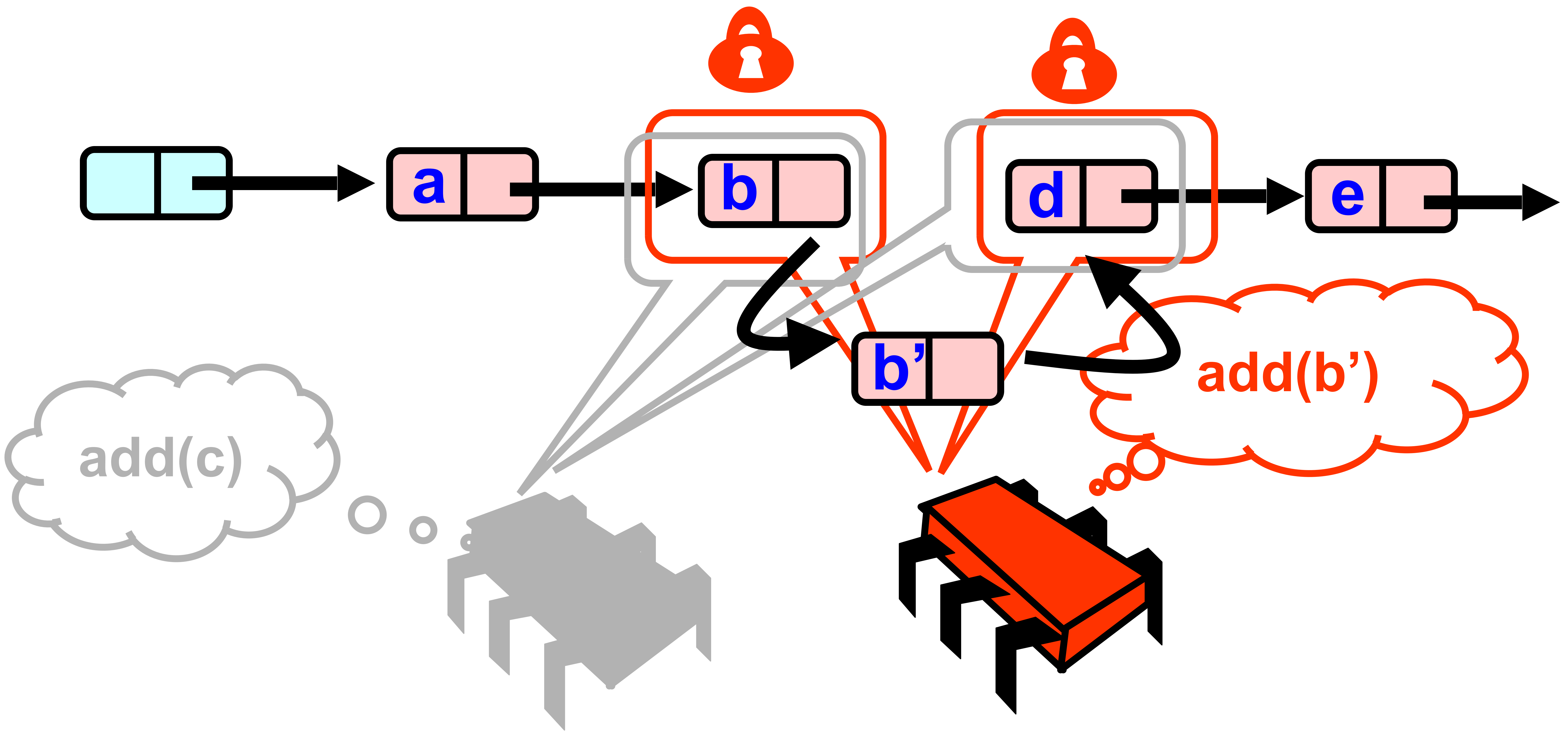
# What Else Could Go Wrong?



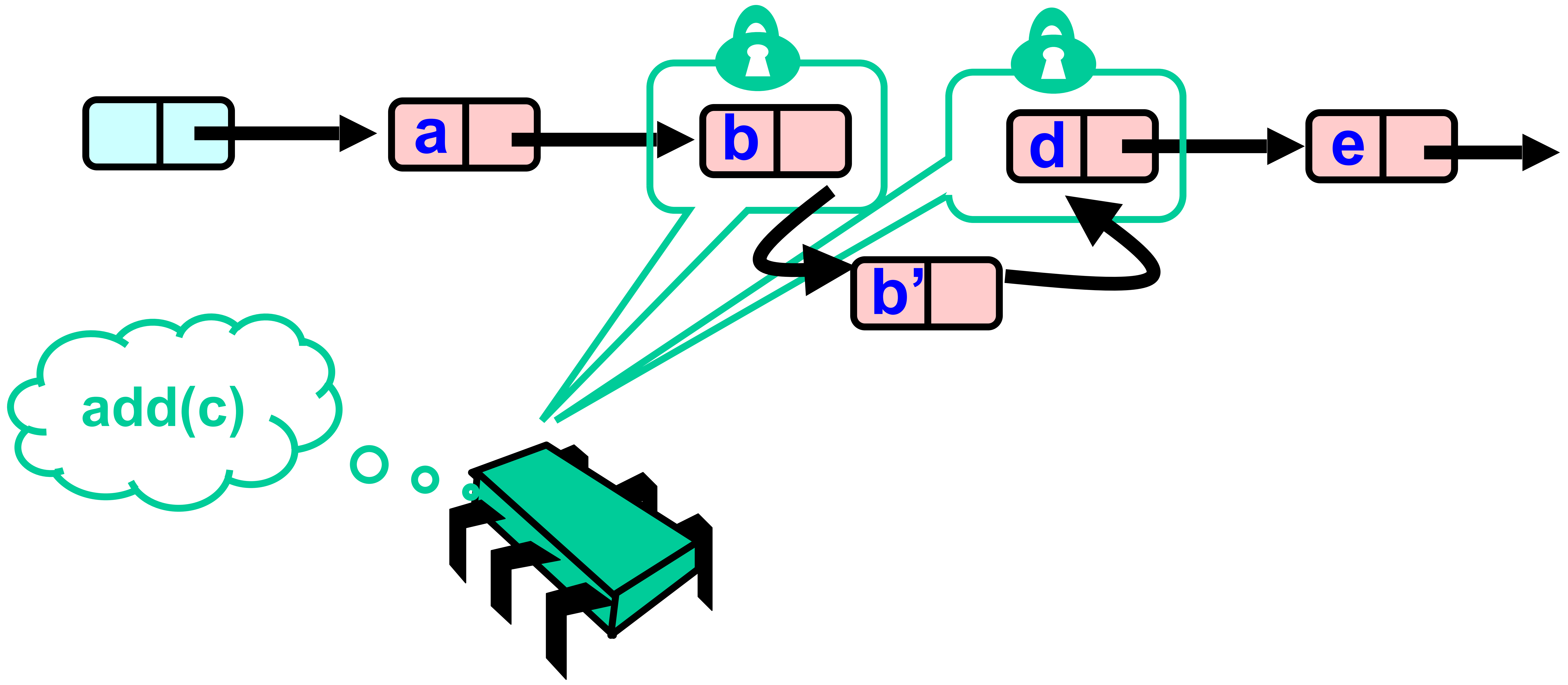
# What Else Could Go Wrong?



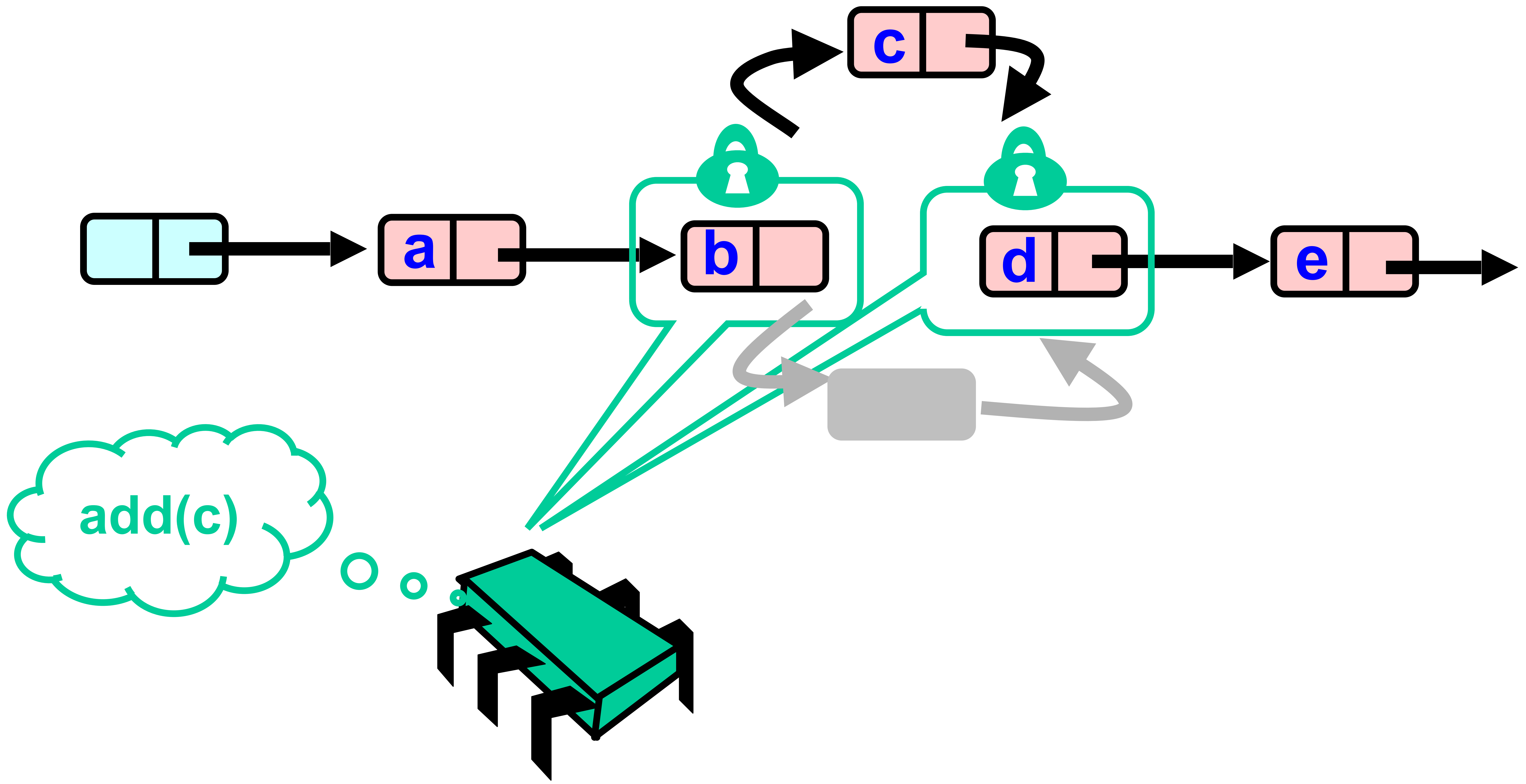
# What Else Could Go Wrong?



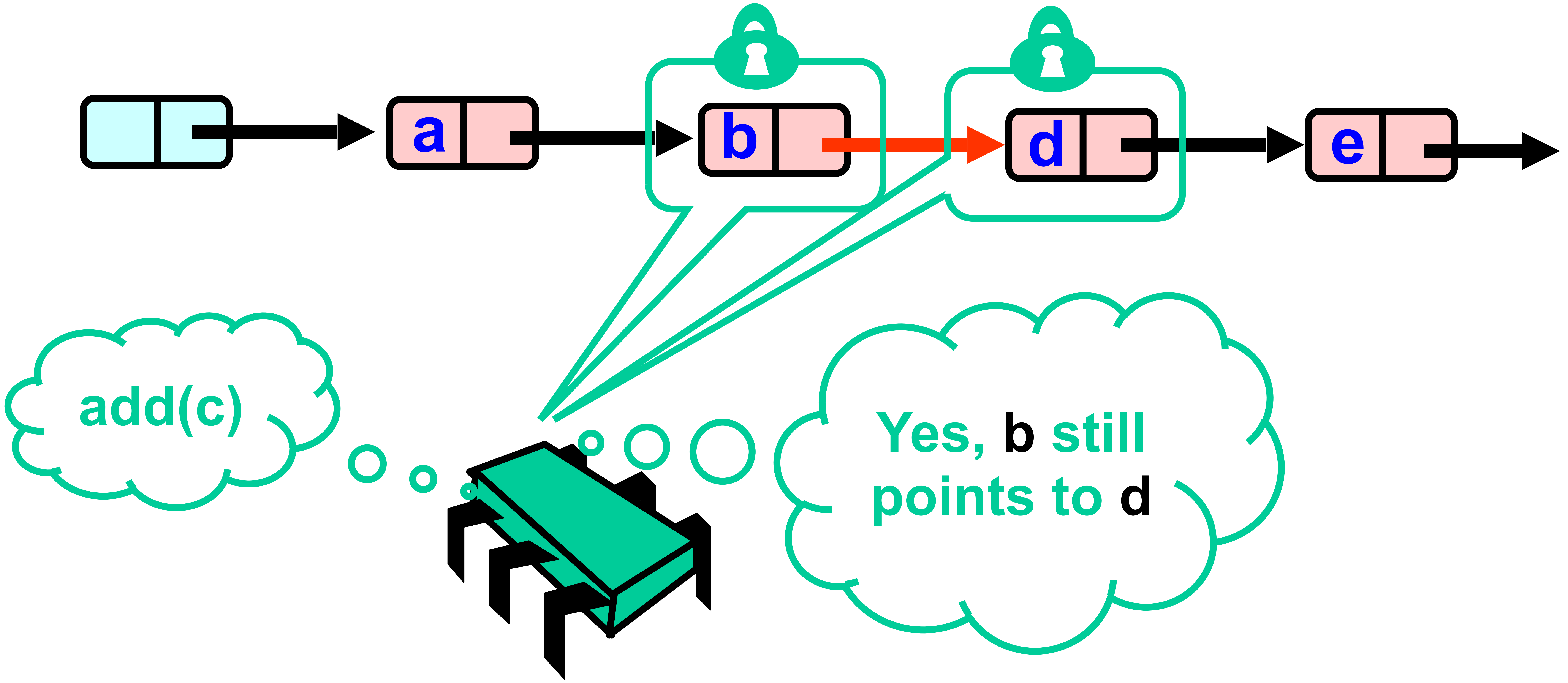
## What Else Could Go Wrong?



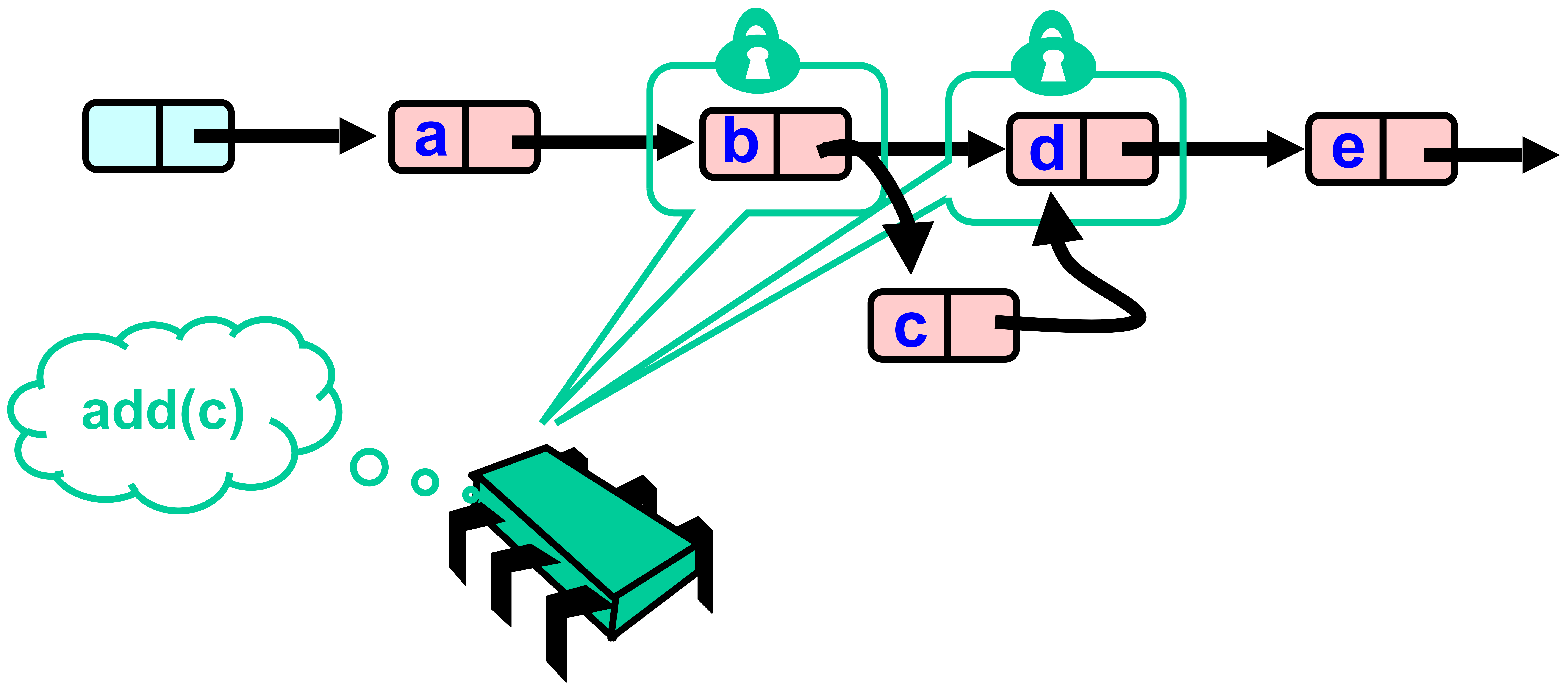
# What Else Could Go Wrong?



# Validate Part 2
(while holding locks)



# Optimistic: Linearization Point



## Same Abstraction Map

- S(head) =

  { x | there exists a such that
  - a reachable from head and
  - a.item = x

  }

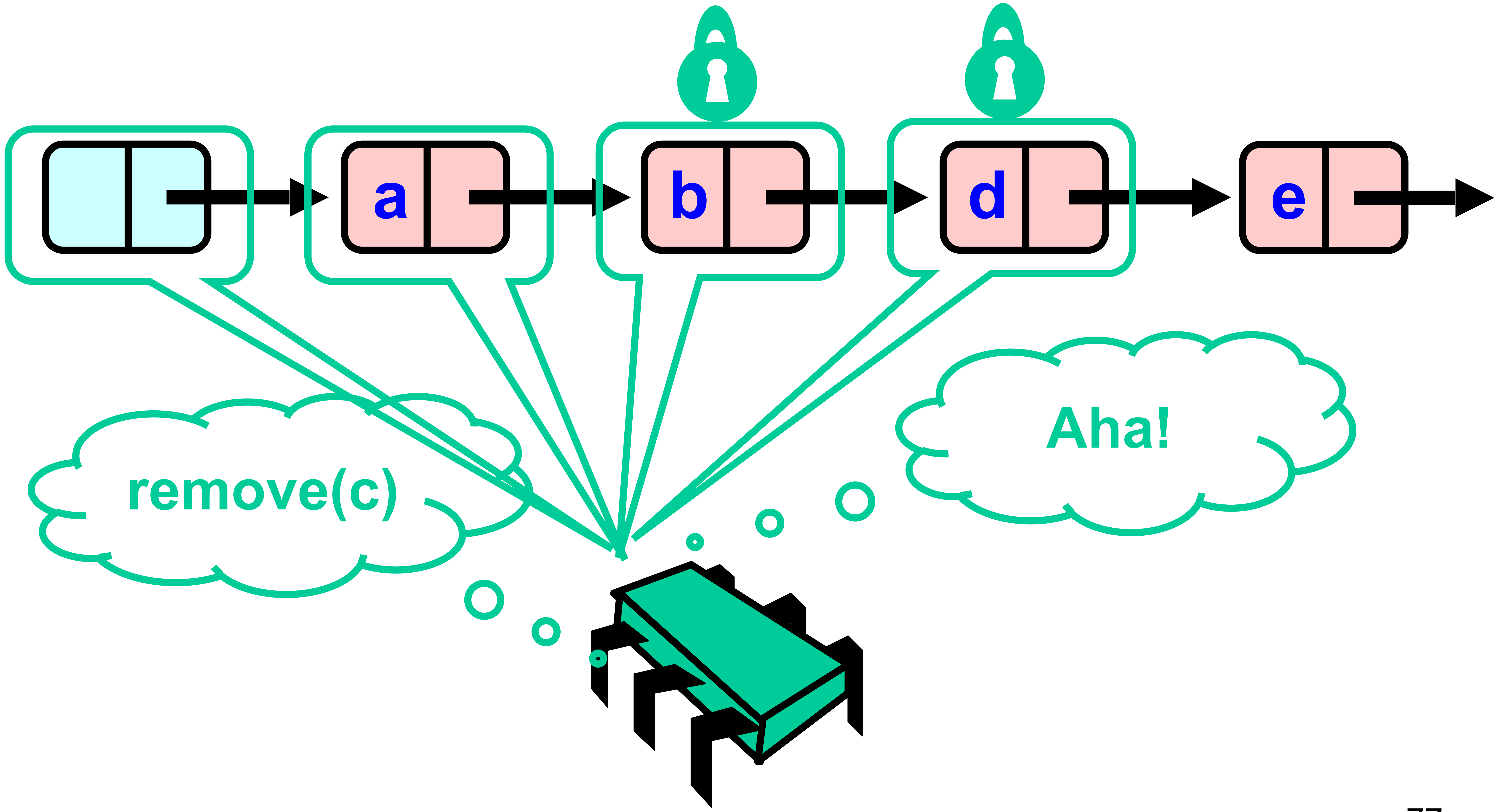
# Invariants

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

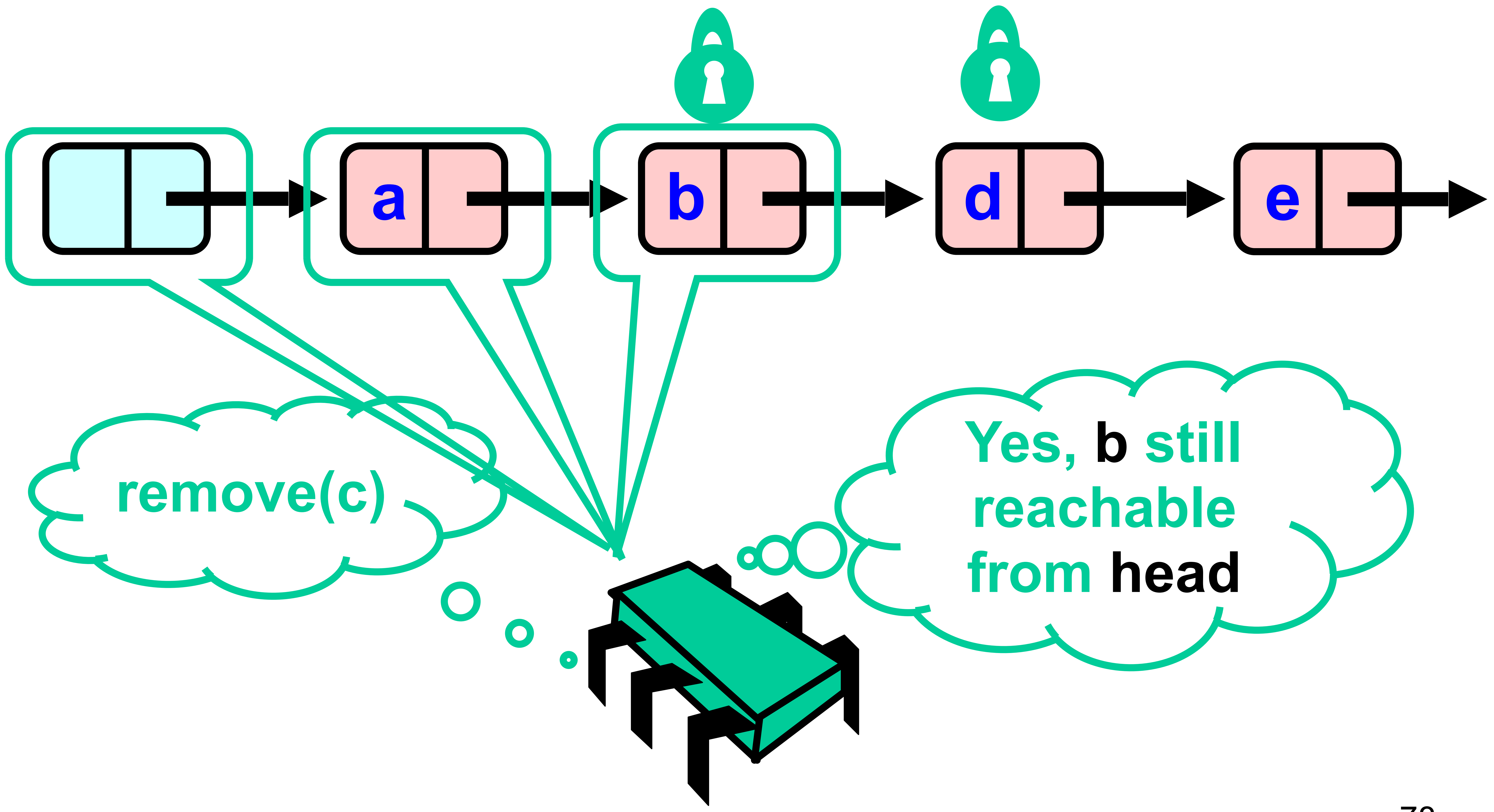
## Removal

- 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

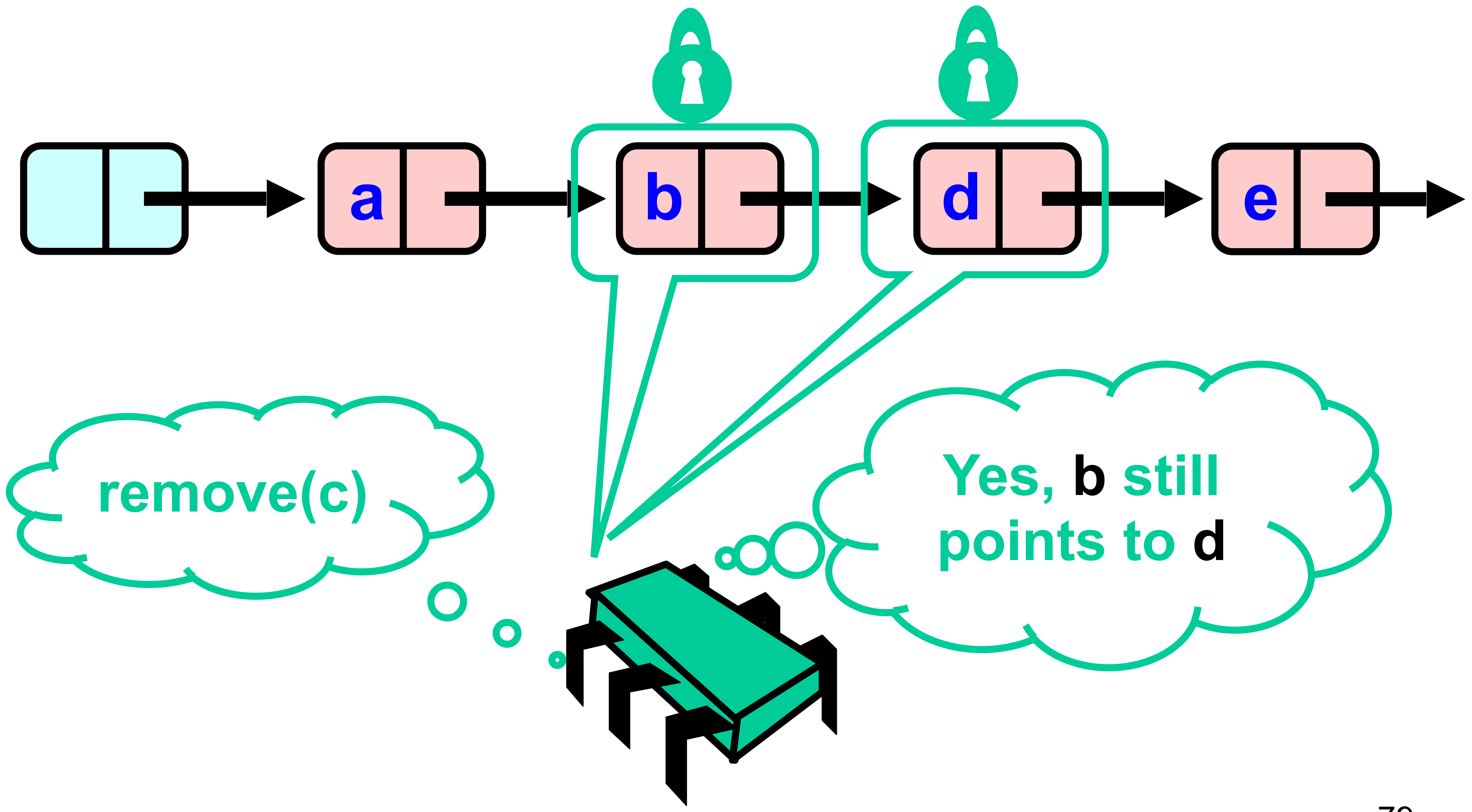
# Unsuccessful Remove



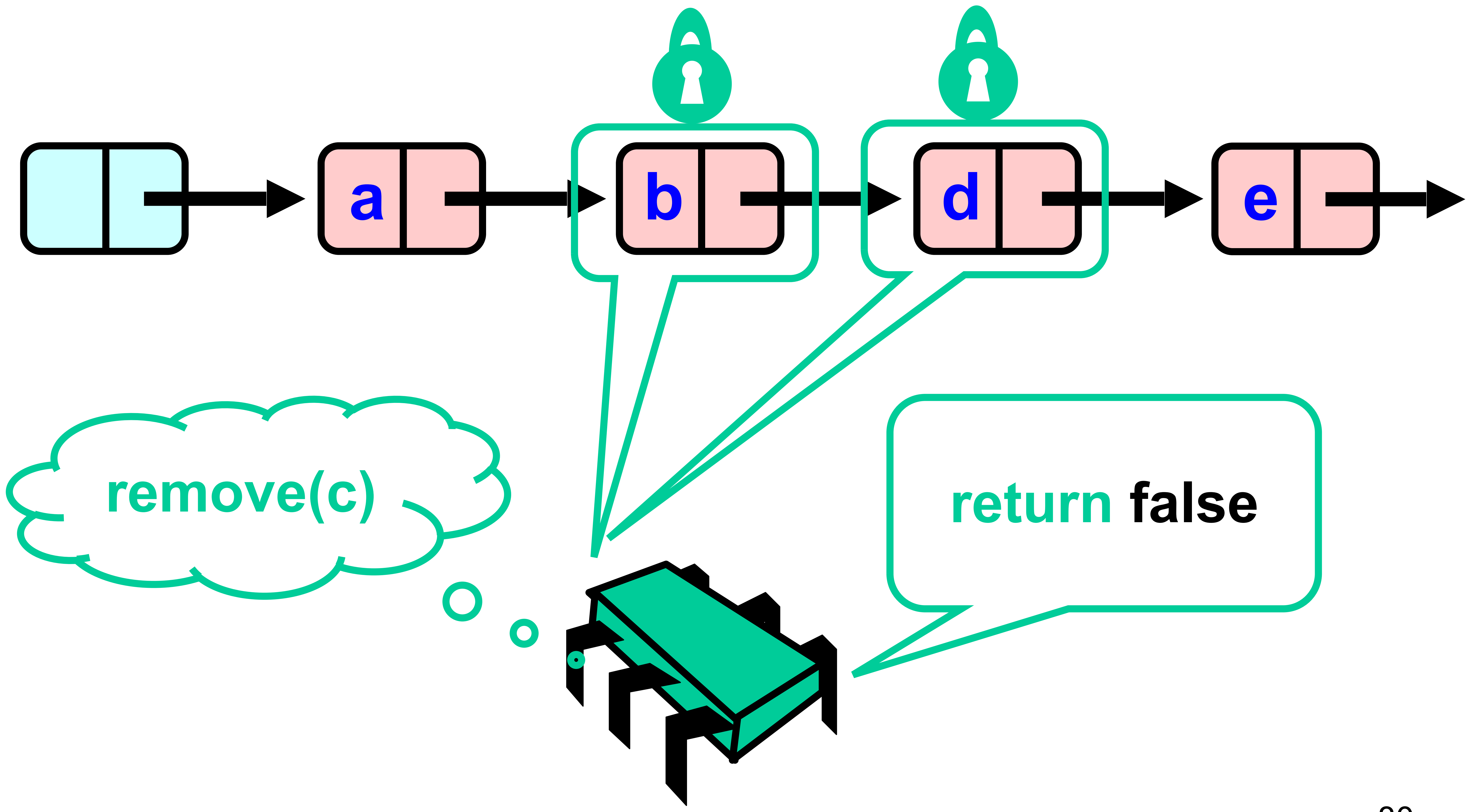
# Validate (1)



# 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

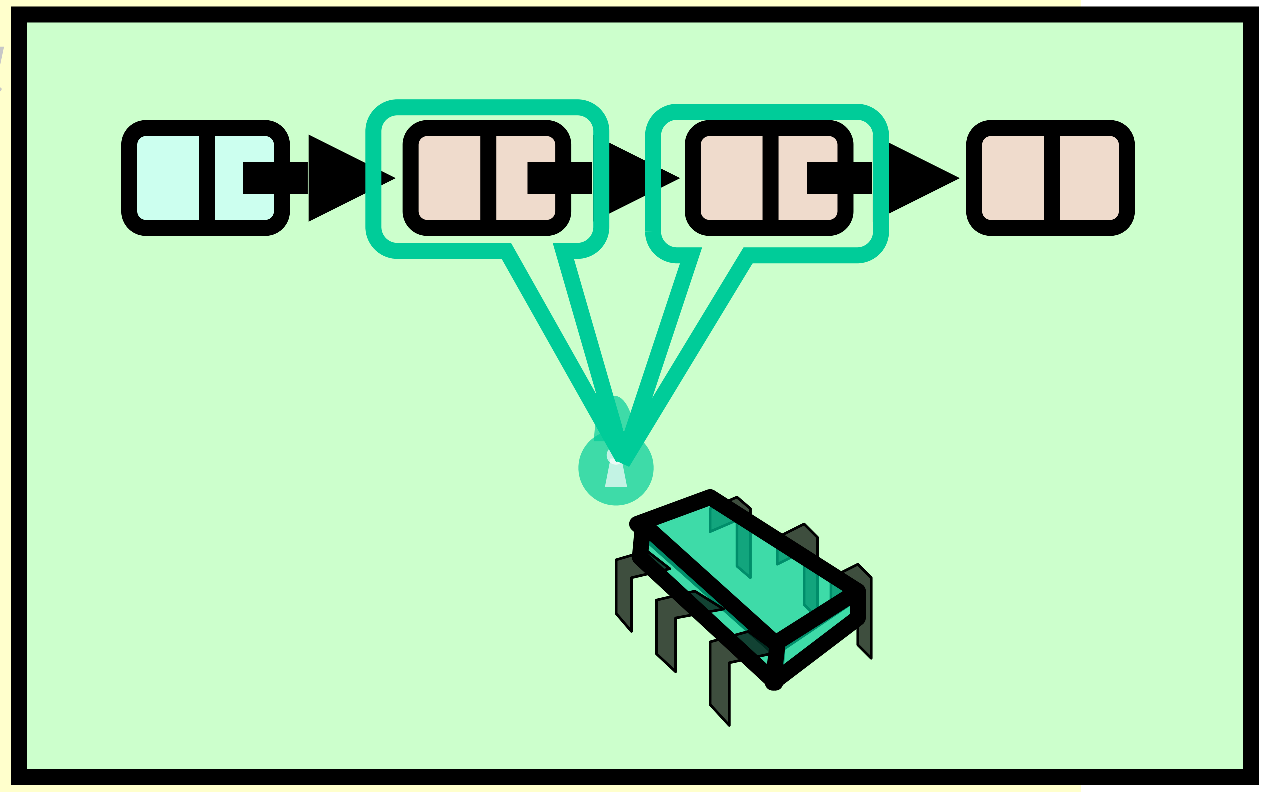
## Validation

```
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 equ
    if (entry eq pred) {
     return pred.next eq curr
    }
    entry = entry.next
  }
  false
}
```

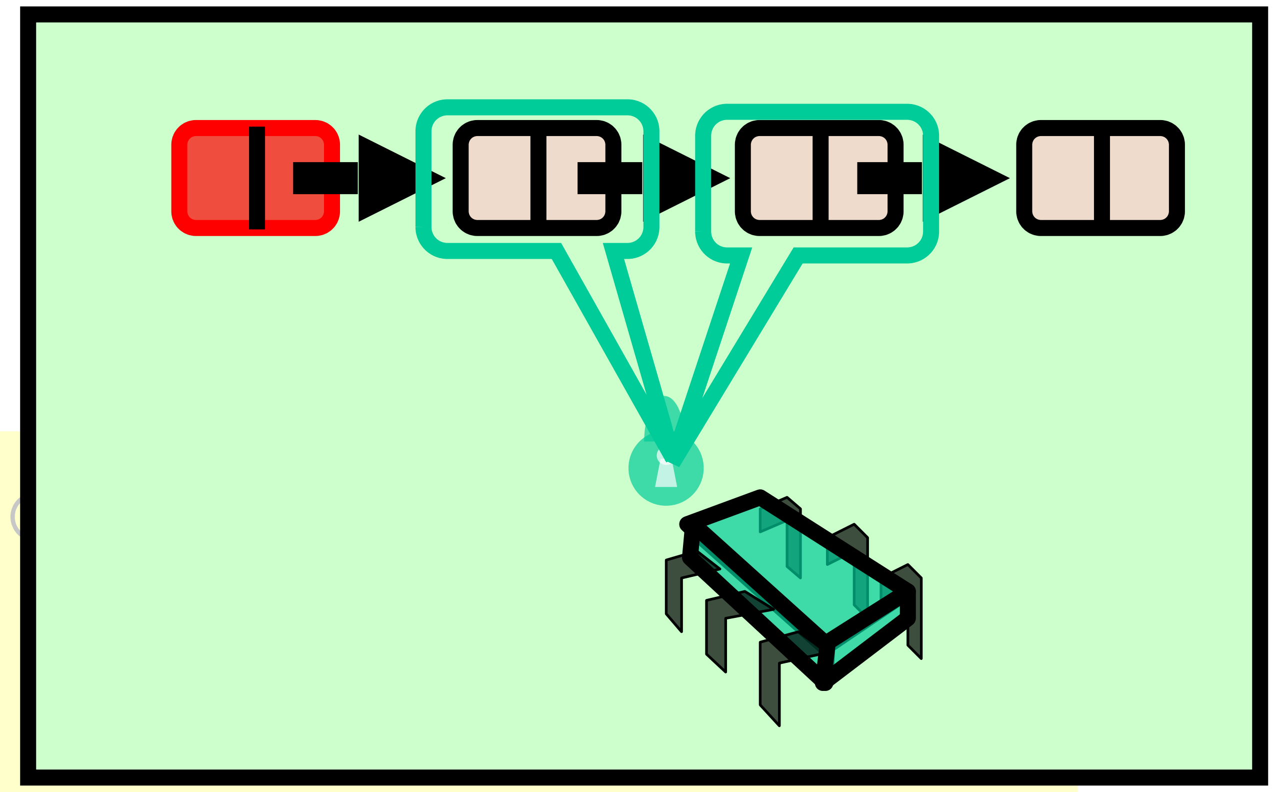
**Predecessor & current nodes**



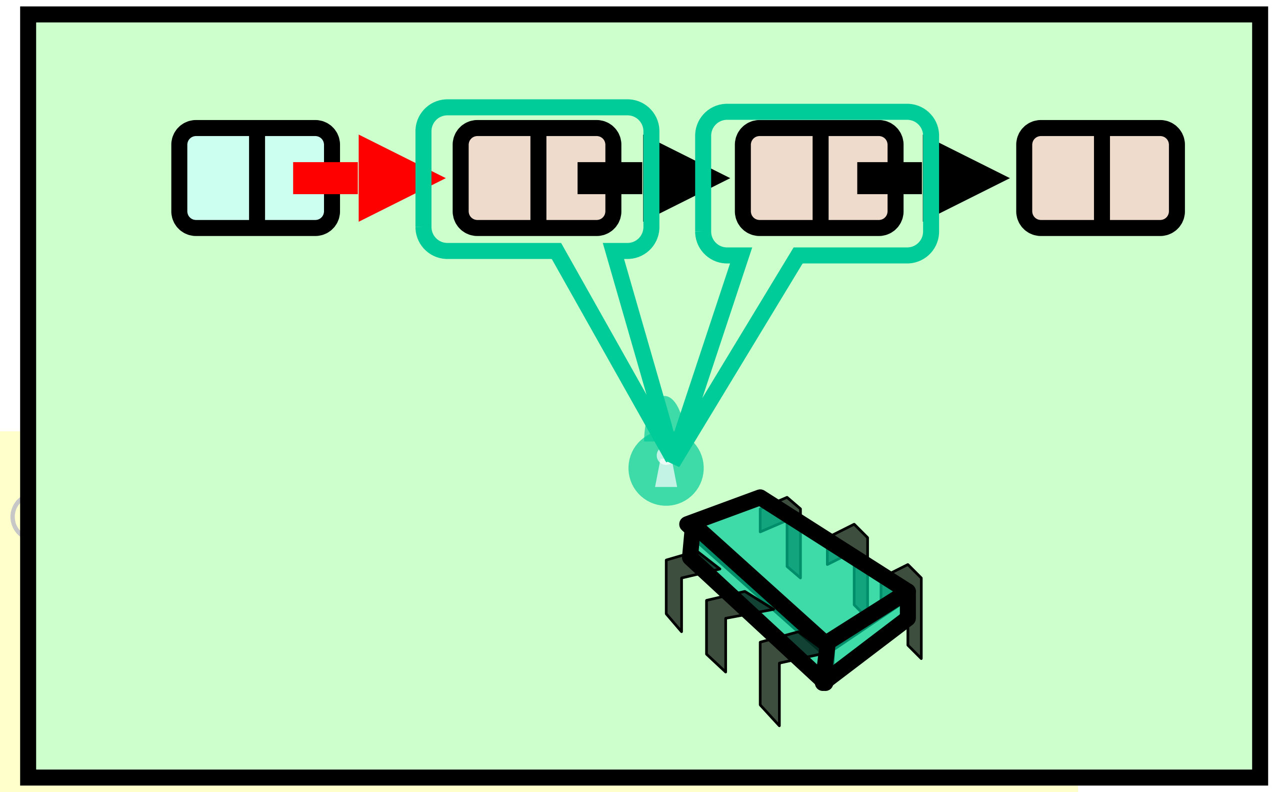
# Validation

```
def validate(pred: Node, curr: Nod
  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



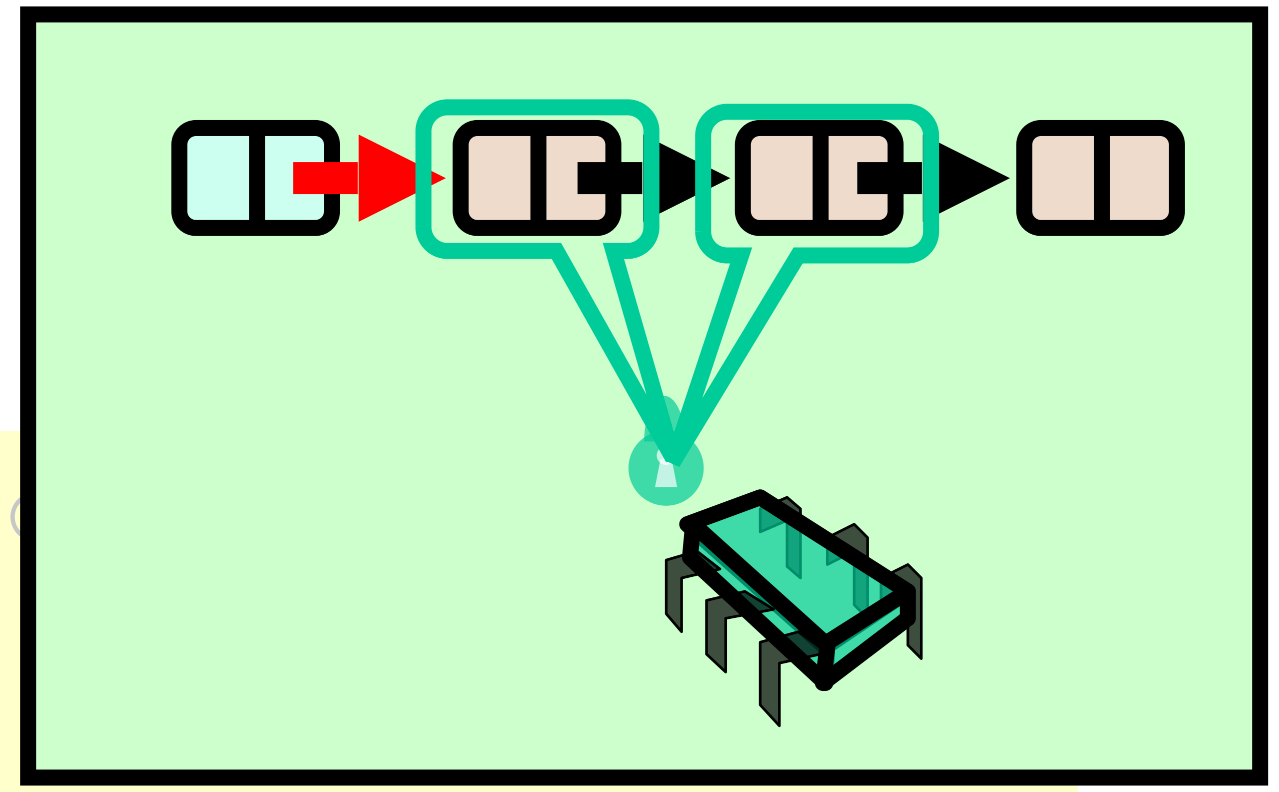
# Validation



```
def validate(pred: Node, curr: Noc
  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**

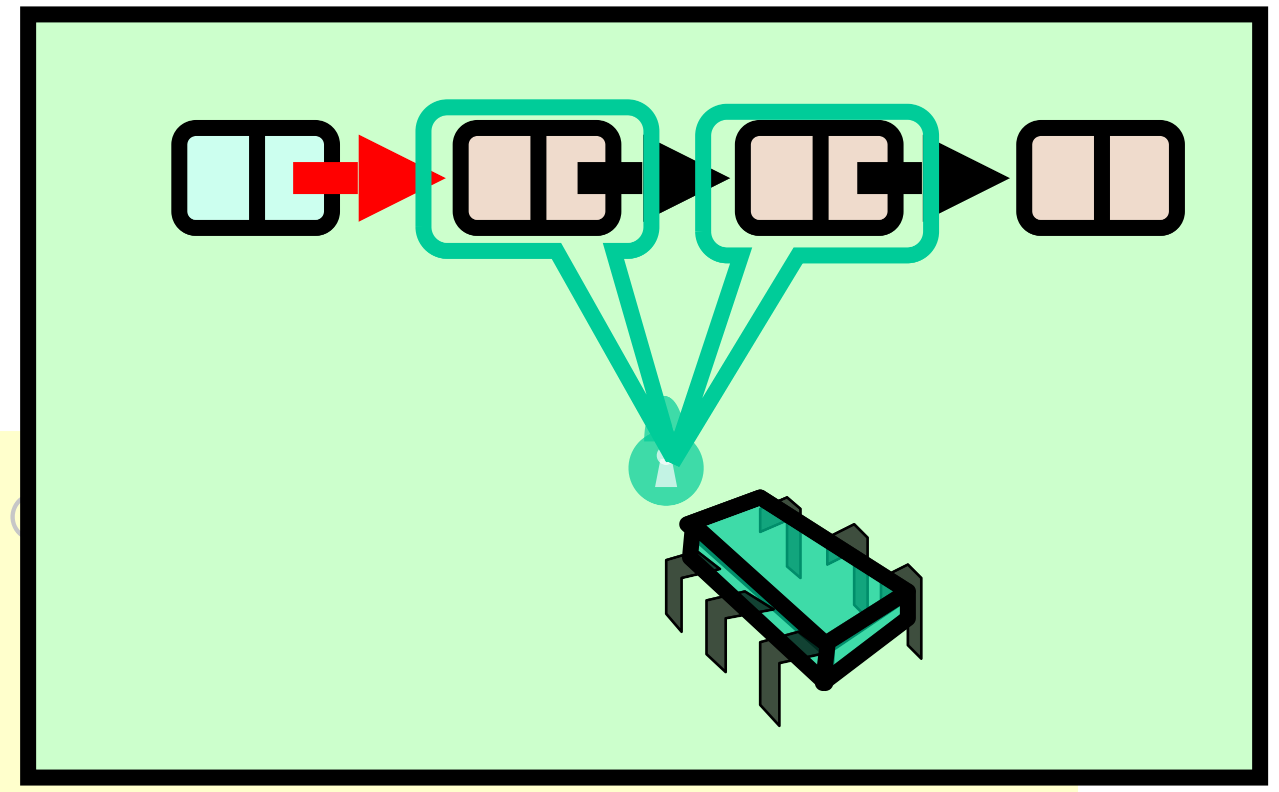
## Validation



```
def validate(pred: Node, curr: No
  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



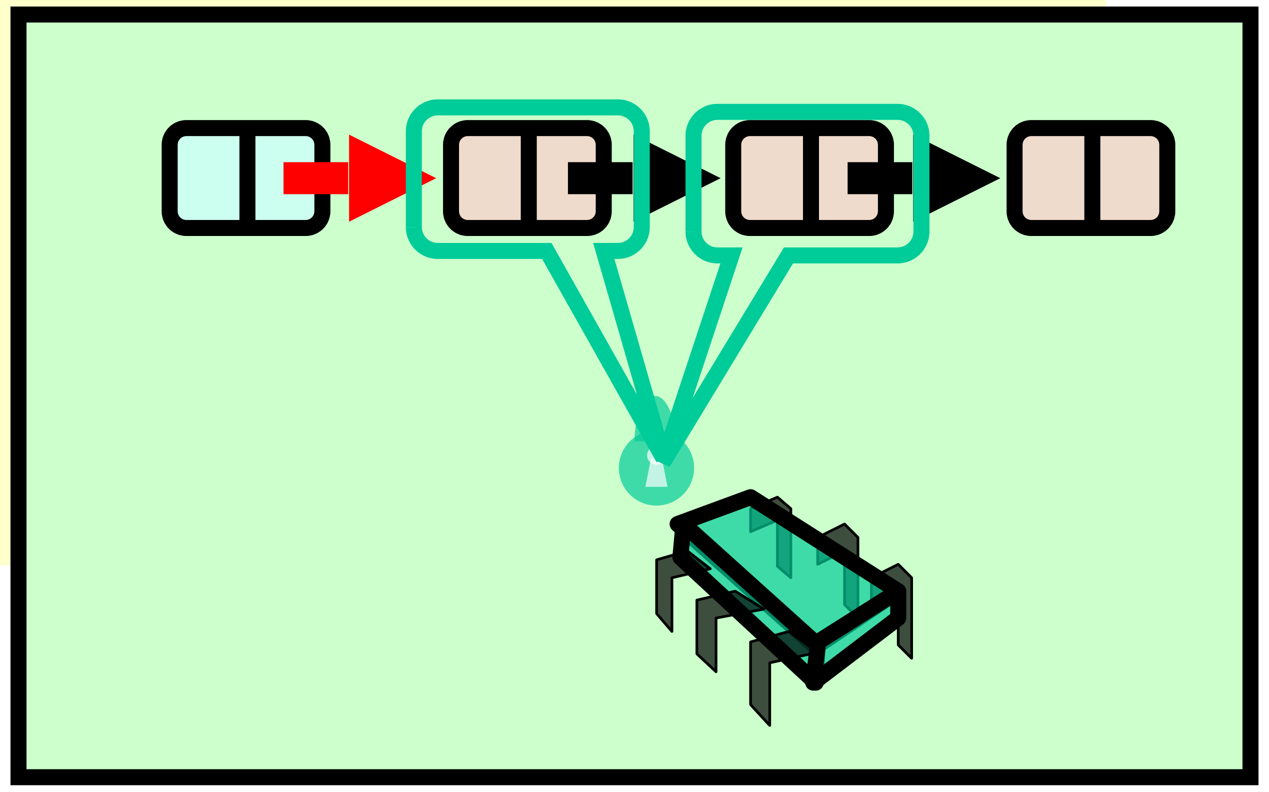
```
def validate(pred: Node, curr: No
  var entry = head
  while (entry.key <= pred.key) {
    // Checking for reference equality
    if (entry eq pred) {
      return pred.next eq curr
    }
    entry = entry.next
  }
  false
}
```

**Is current node next?**

# Validation

Otherwise move on

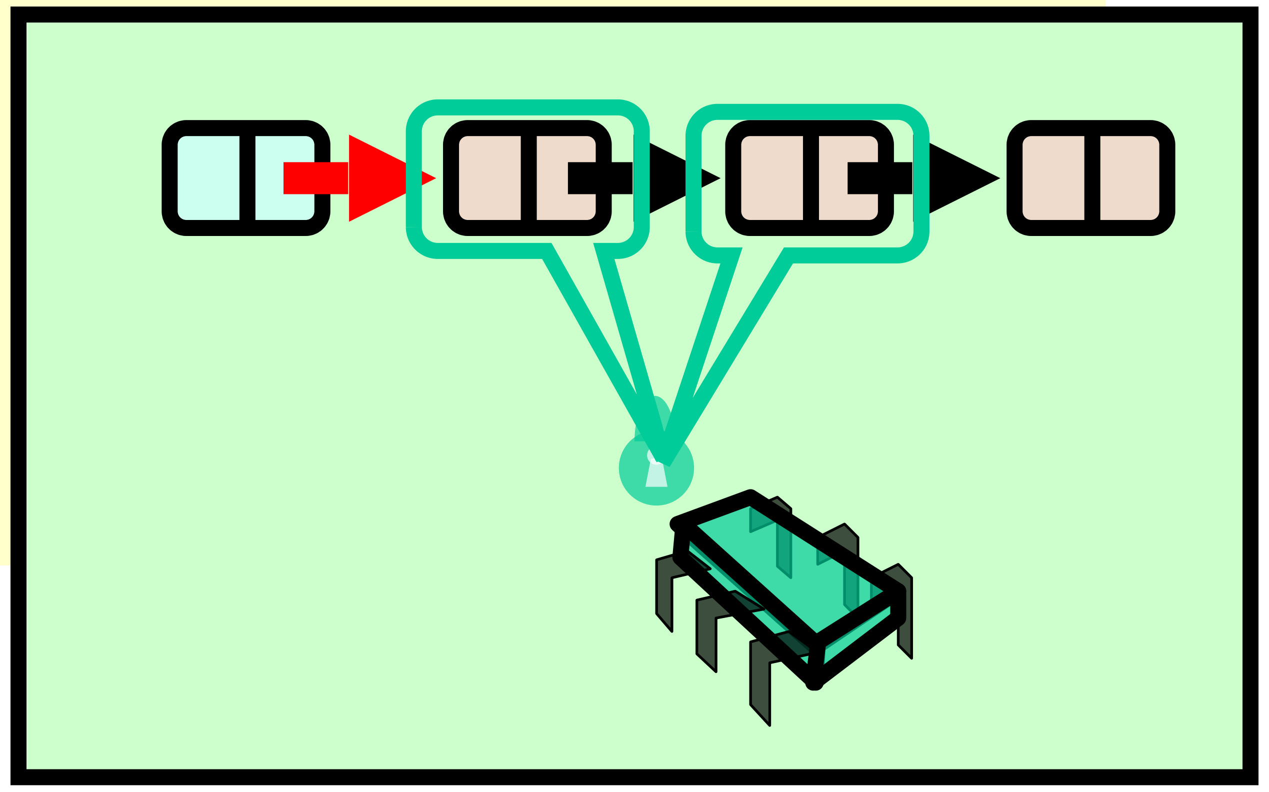
```
def validate(pred: Node, curr: Node): Boolean = {
  var entry = head
  while (entry.key <= pred.key) {
    // Checking for reference equality
    if (entry eq pred) {
     return pred.next eq curr
    }
    entry = entry.next
  }
  false
}
```



## Validation

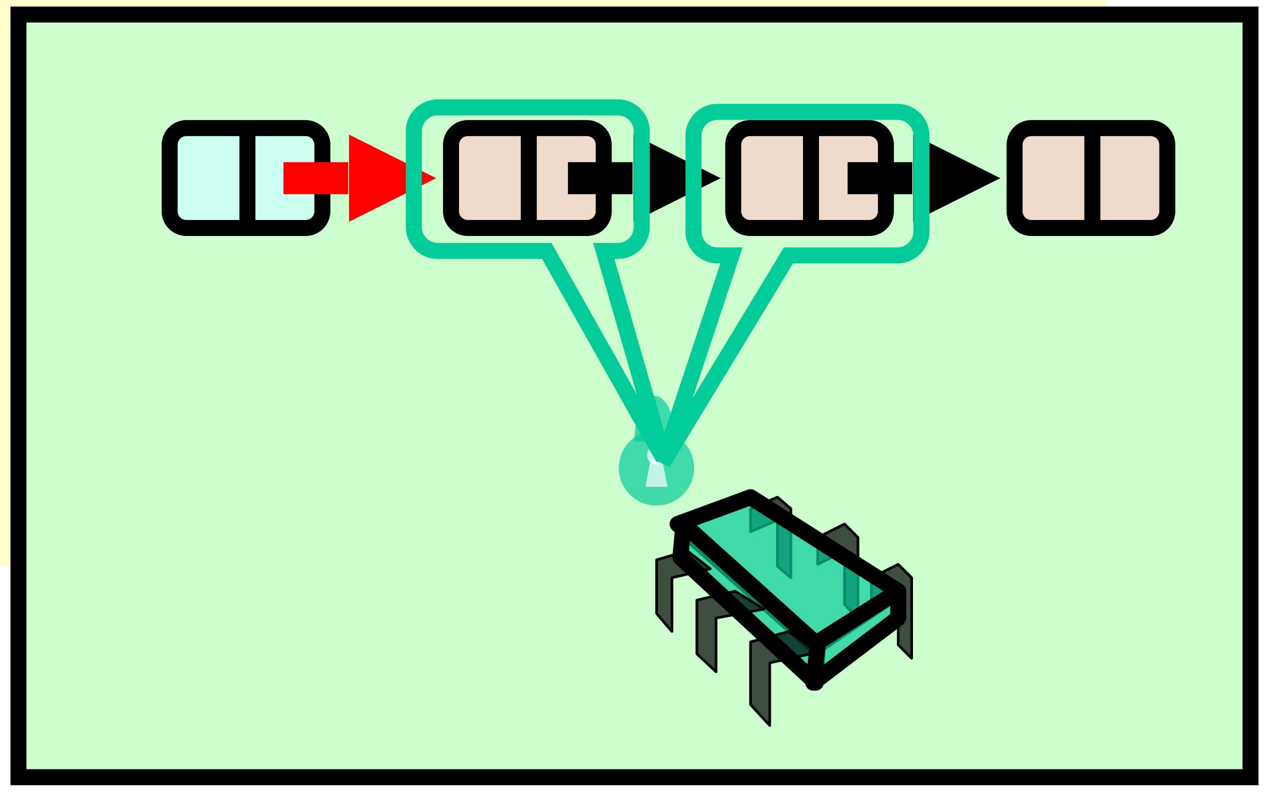
**Predecessor not reachable**

```
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

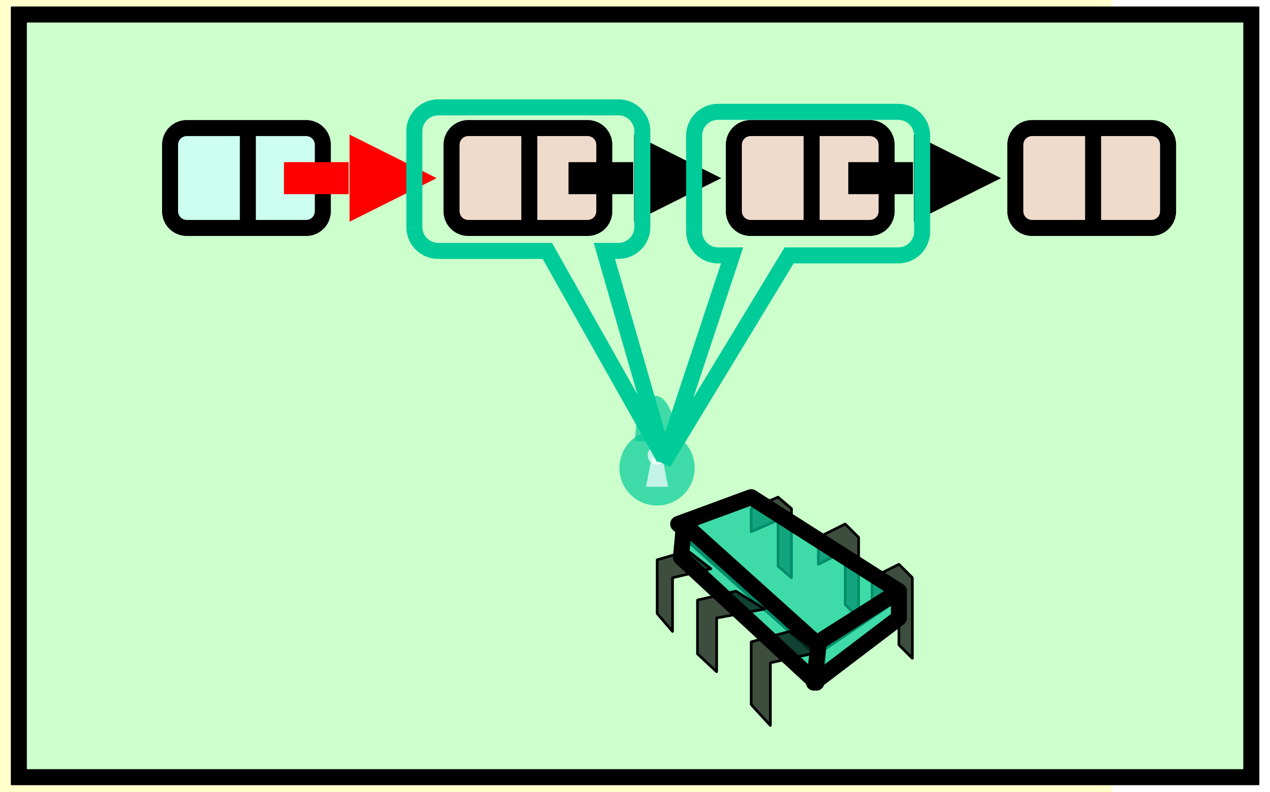
```
def remove(item: T): Boolean = {
  val key = item.hashCode()
  while (true) {
    var pred = this.head
    var curr = pred.next
    while (curr.key < key) {
      pred = curr
      curr = curr.next
    }
  …
}
```



## Remove: searching

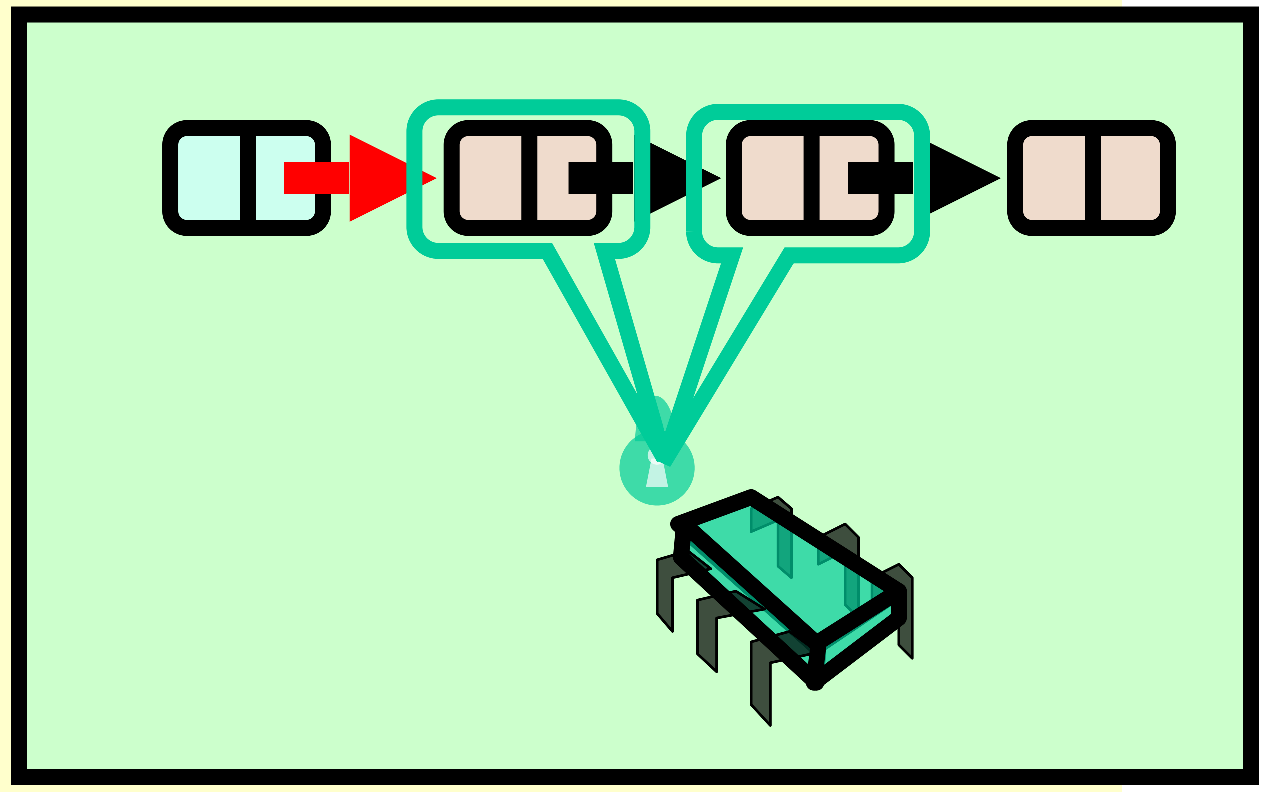
```
def remove(item: T): Boolean = {
  val key = item.hashCode()
  while (true) {
    var pred = this.head
    var curr = pred.next
    while (curr.key < key) {
      pred = curr
      curr = curr.next
    }
  …
}
```

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
    }
  …
}
```

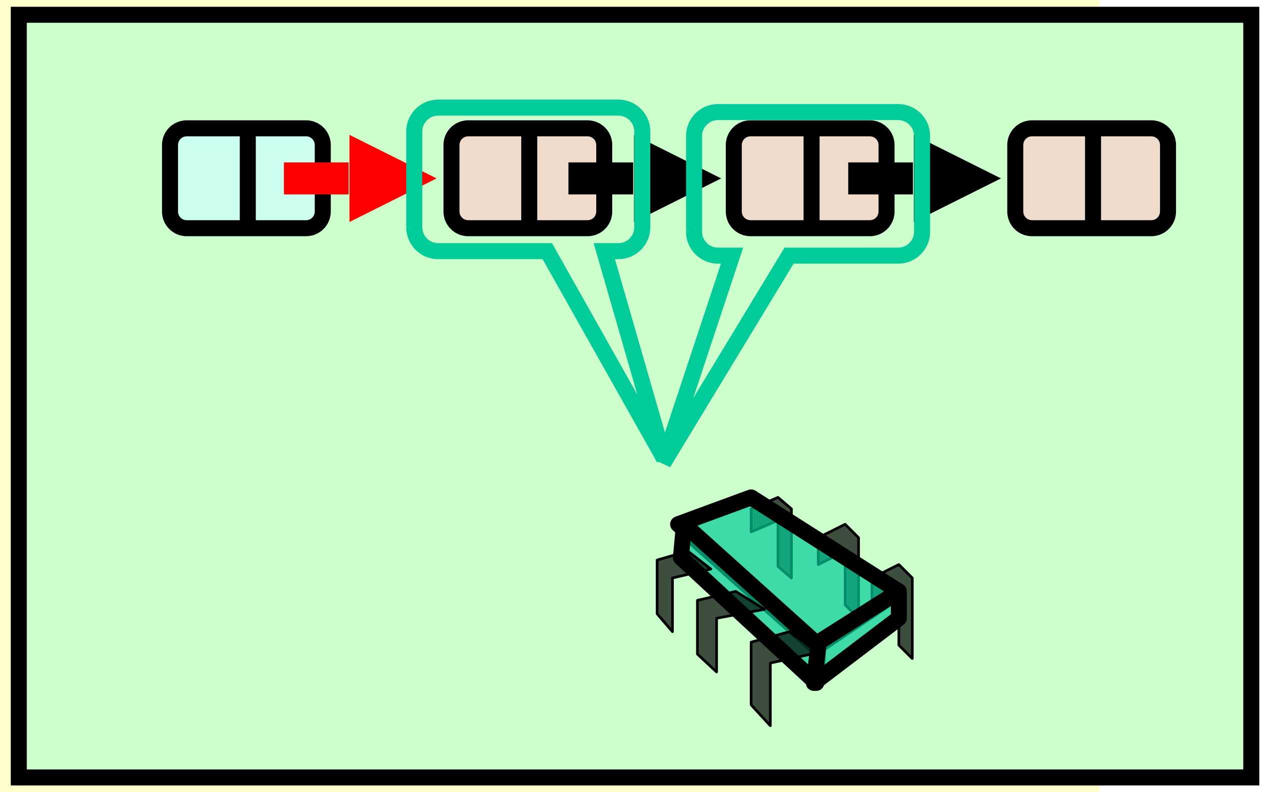


**Loop until no synchronization conflict (see the code further)**

# 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
    }
  …
}
```

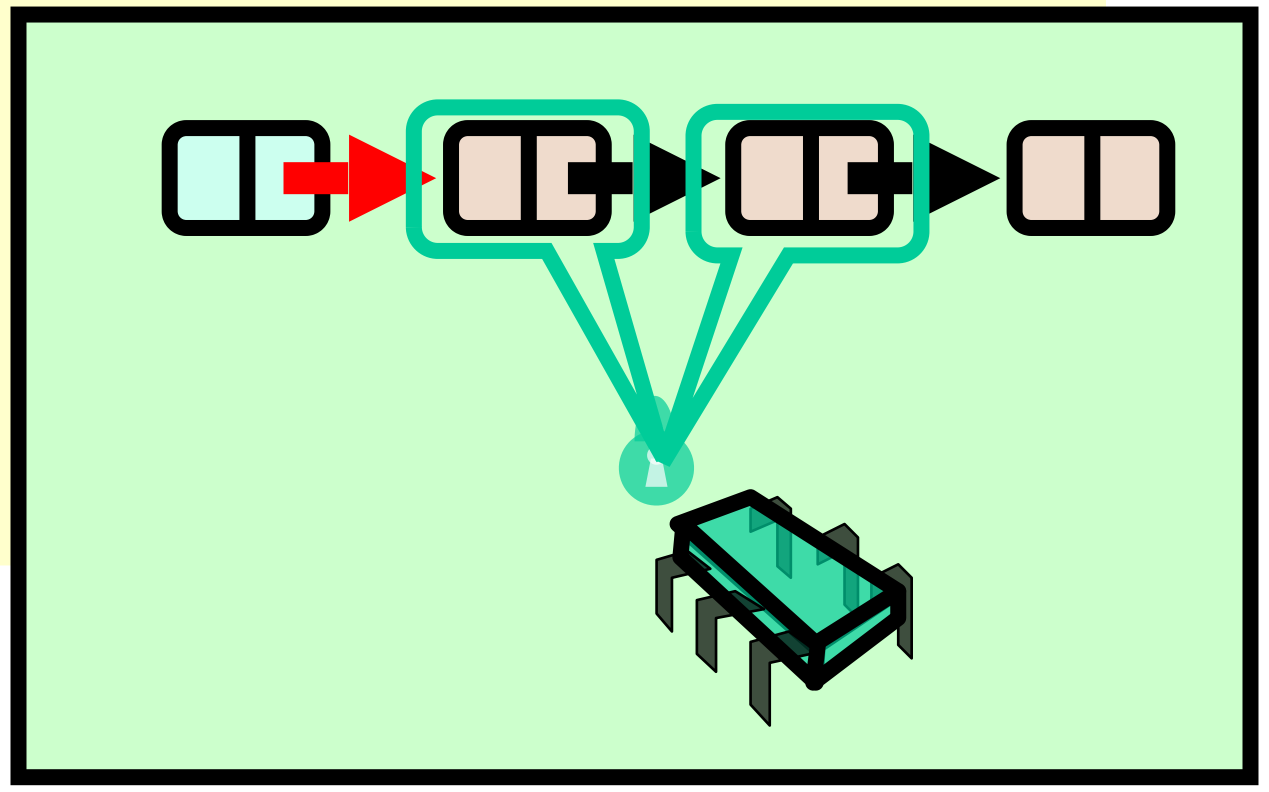
Examine predecessor and current nodes



# 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 Whilte-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 { // 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()
}
```

**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 {
  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

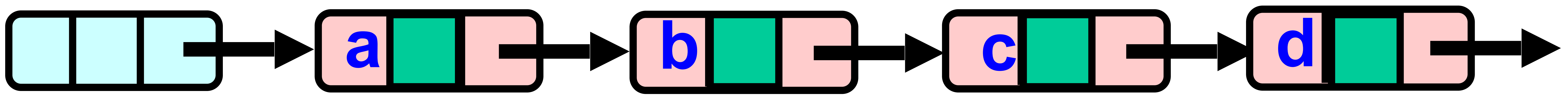
# Lazy List

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

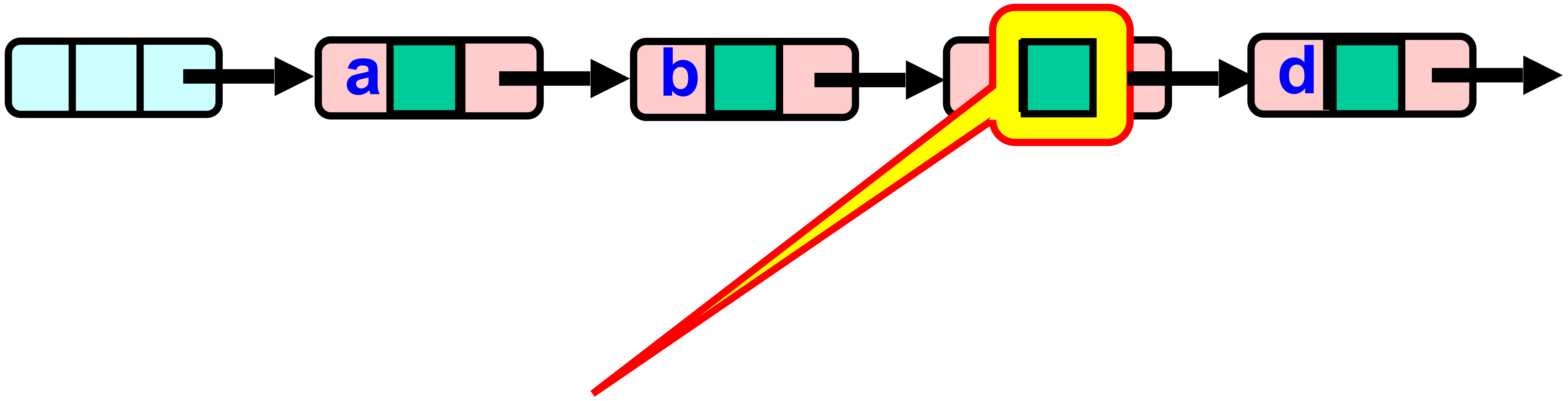
## Lazy List

- **`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

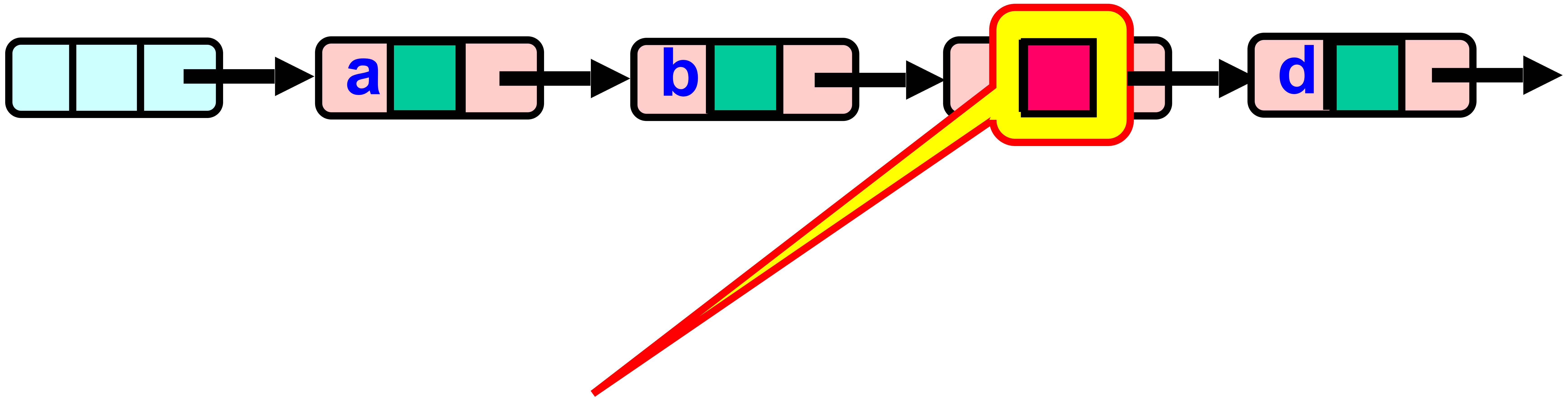


# Lazy Removal



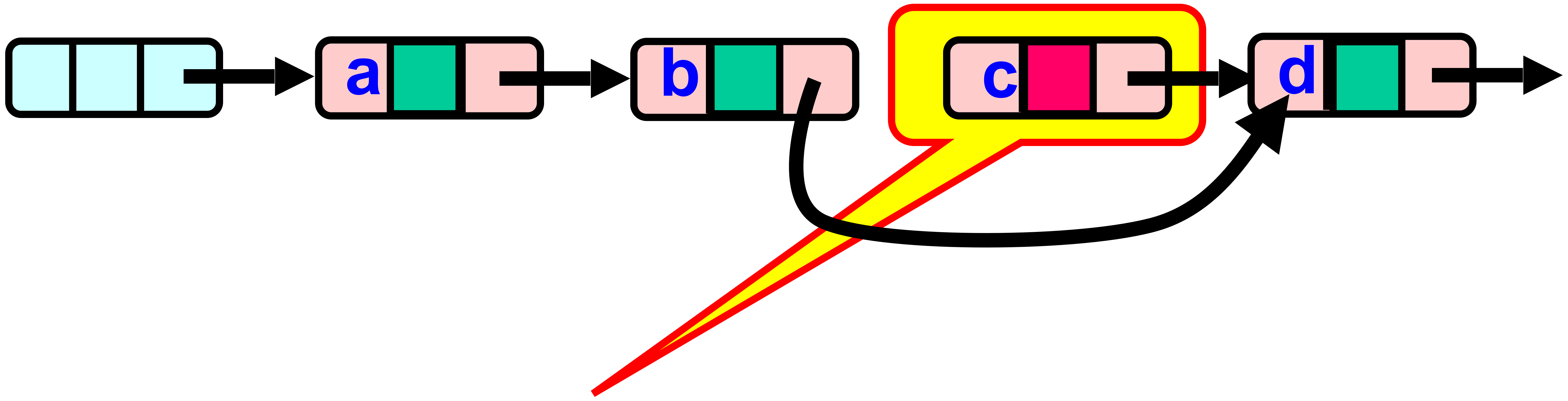
Present in list

# Lazy Removal



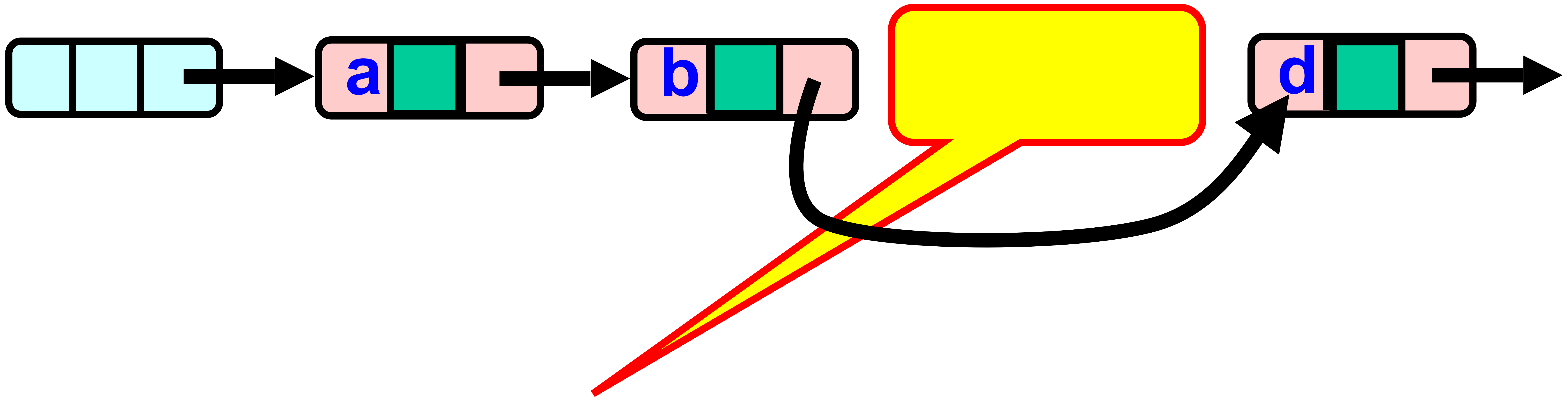
Logically deleted

## Lazy Removal



Physically deleted

# Lazy Removal



Physically deleted

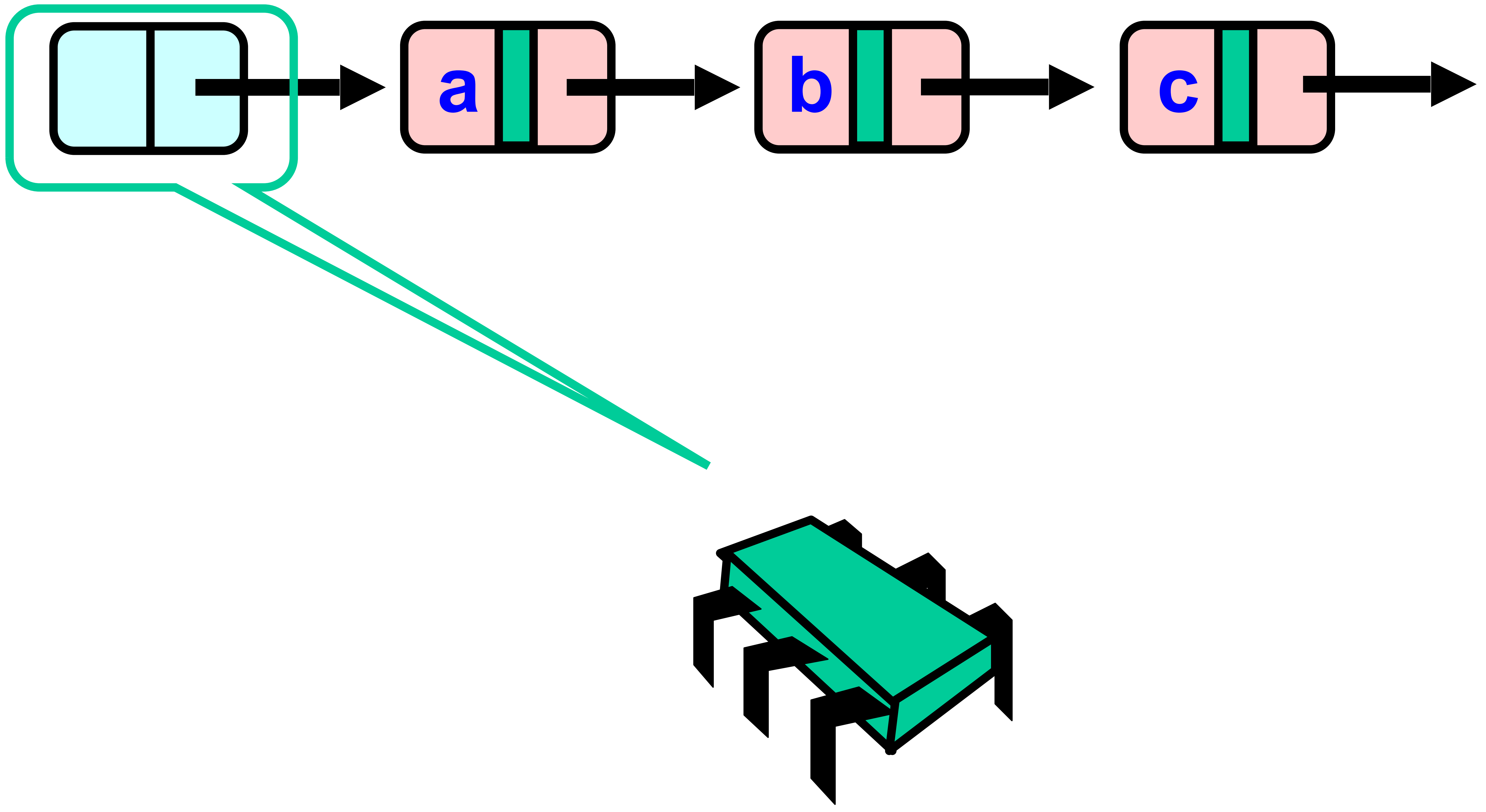
## Lazy List

- 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.

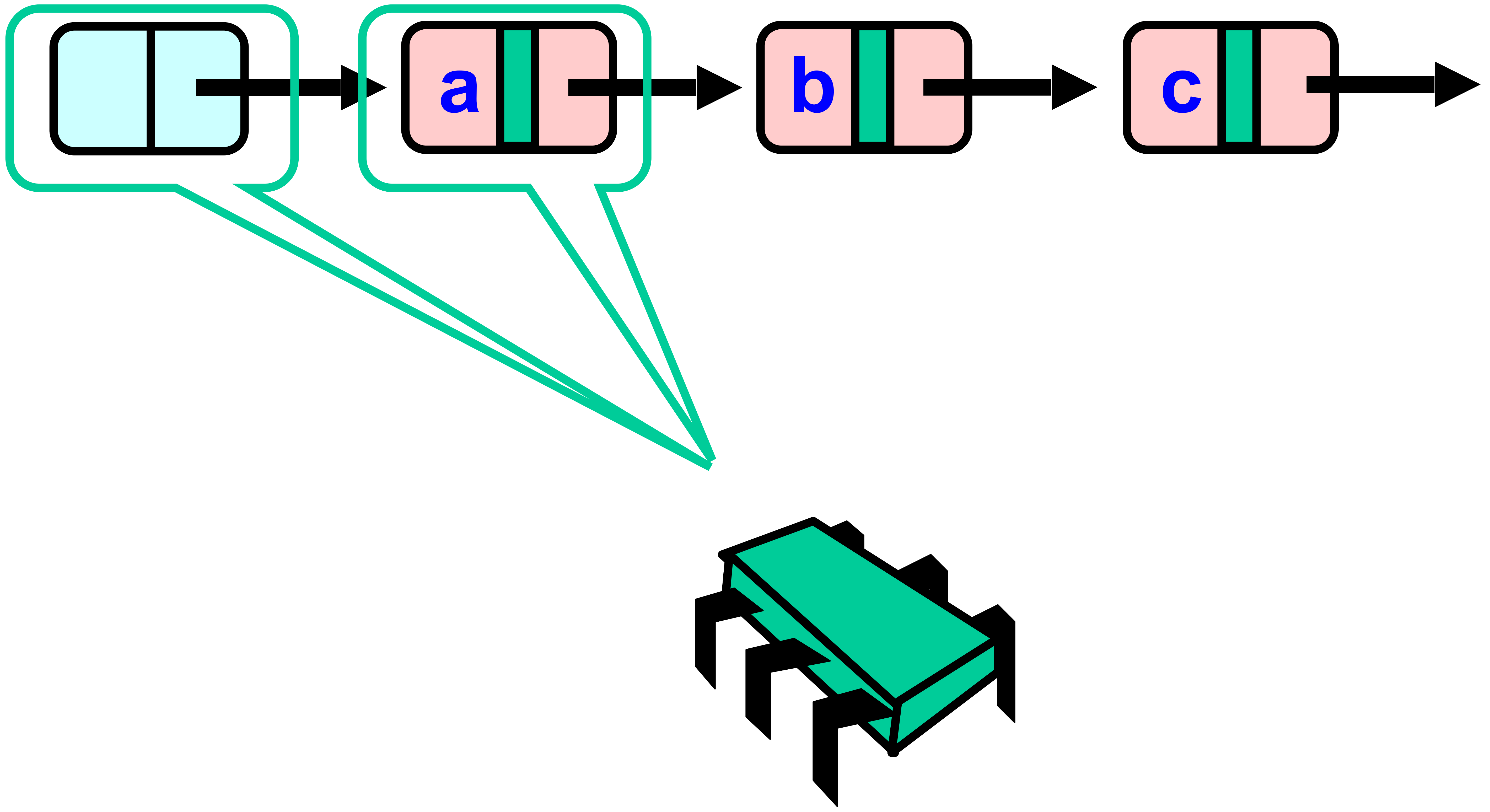
# Validation

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

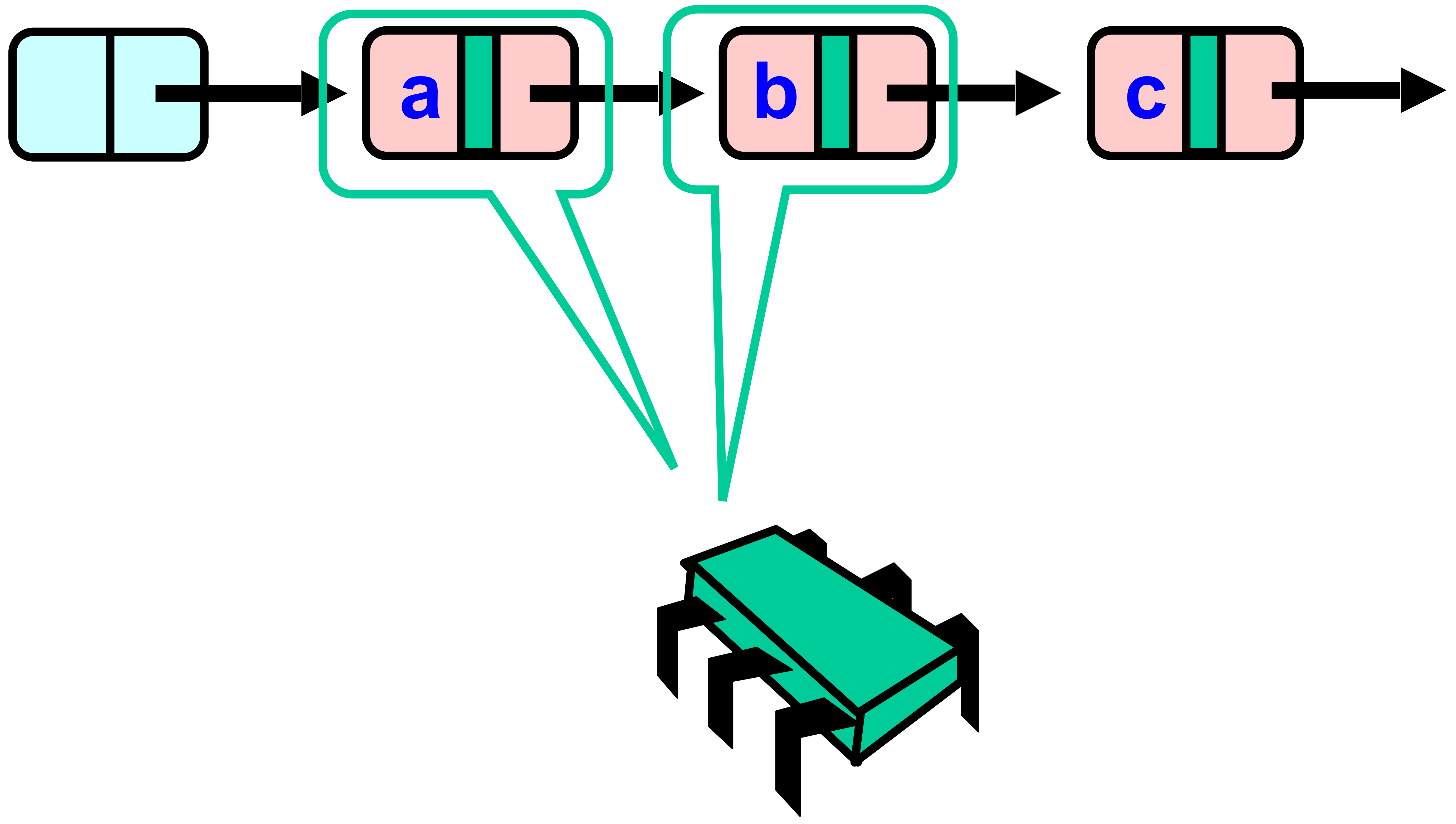
# Business as Usual



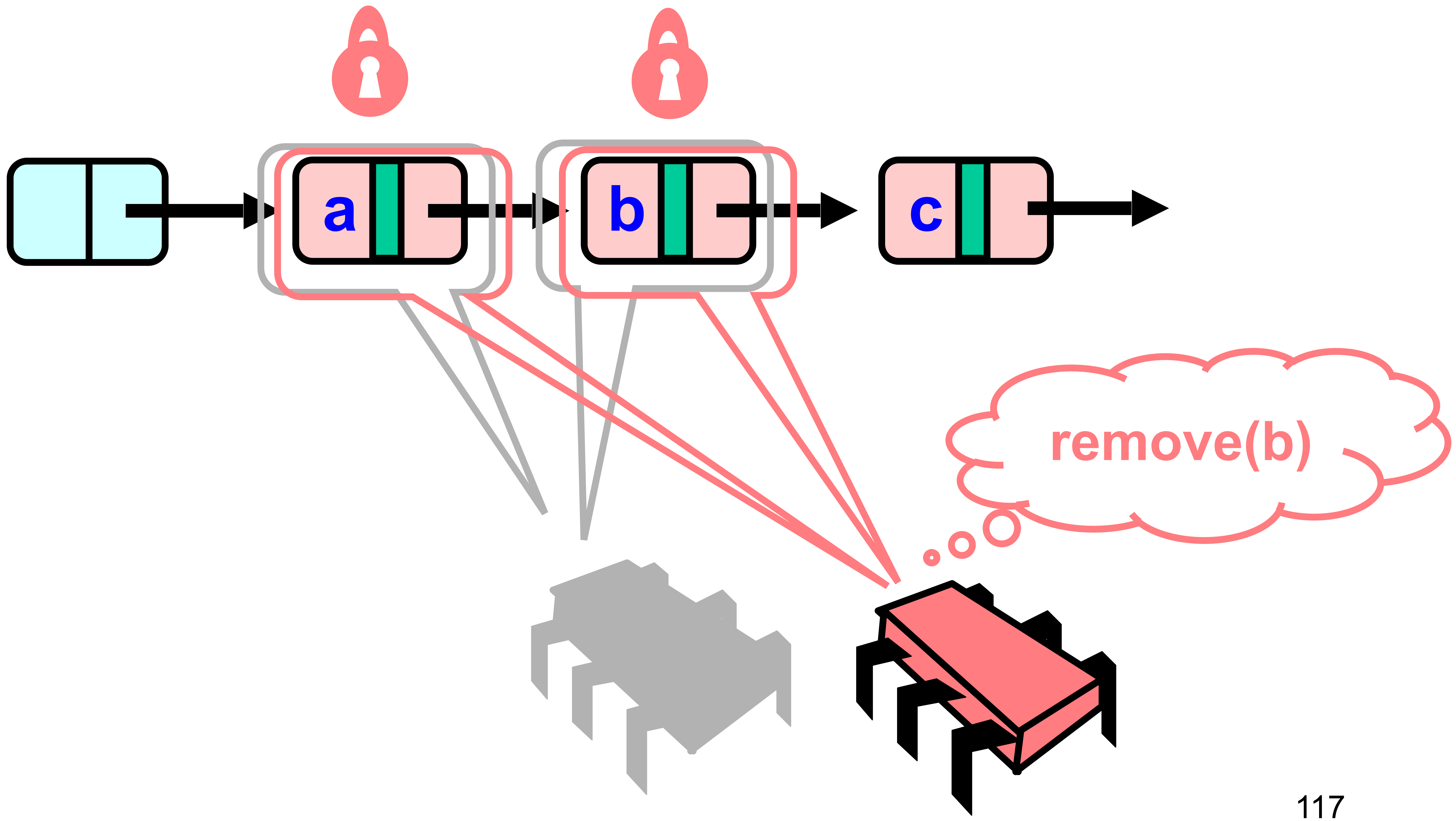
# Business as Usual



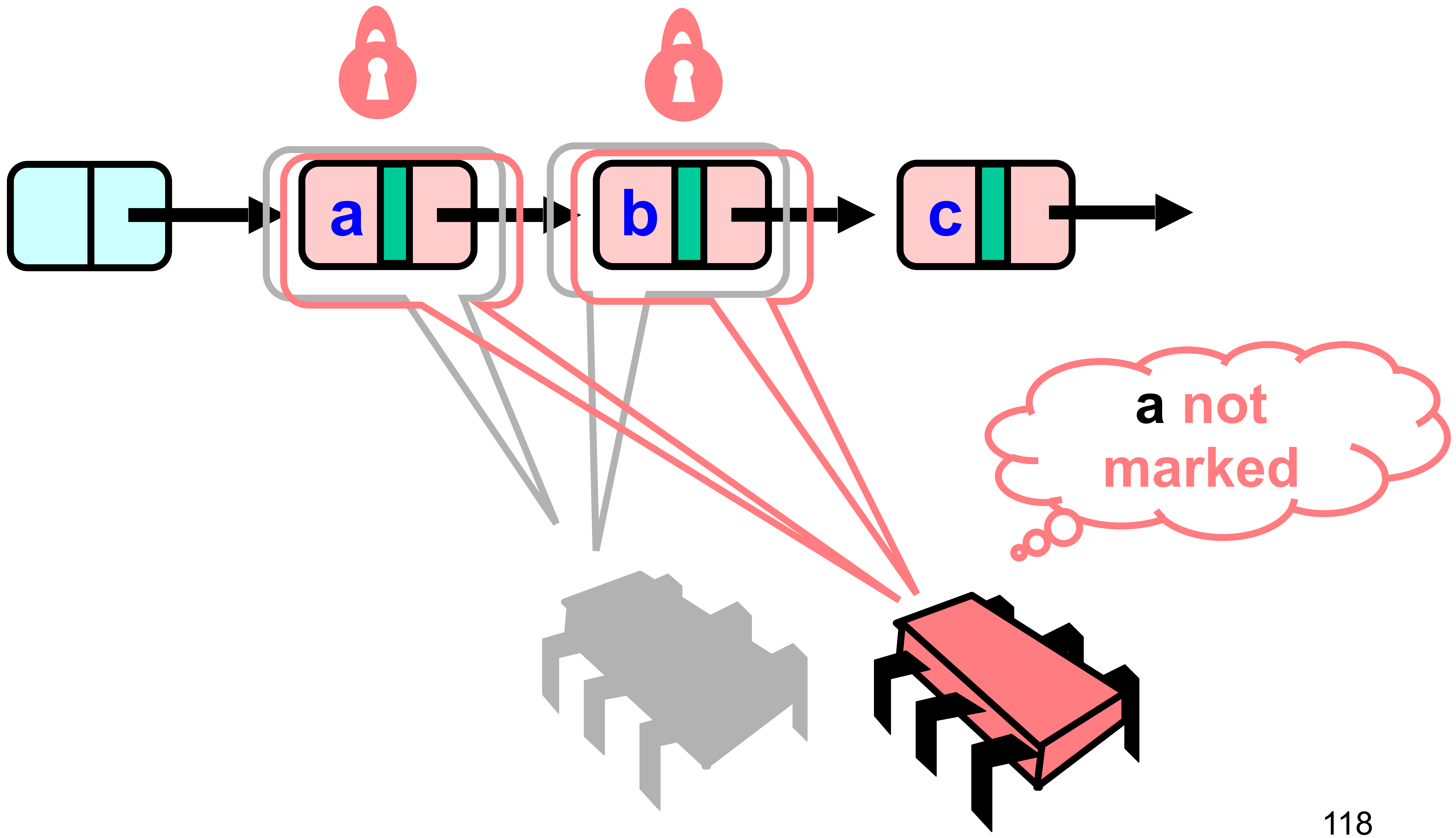
# Business as Usual



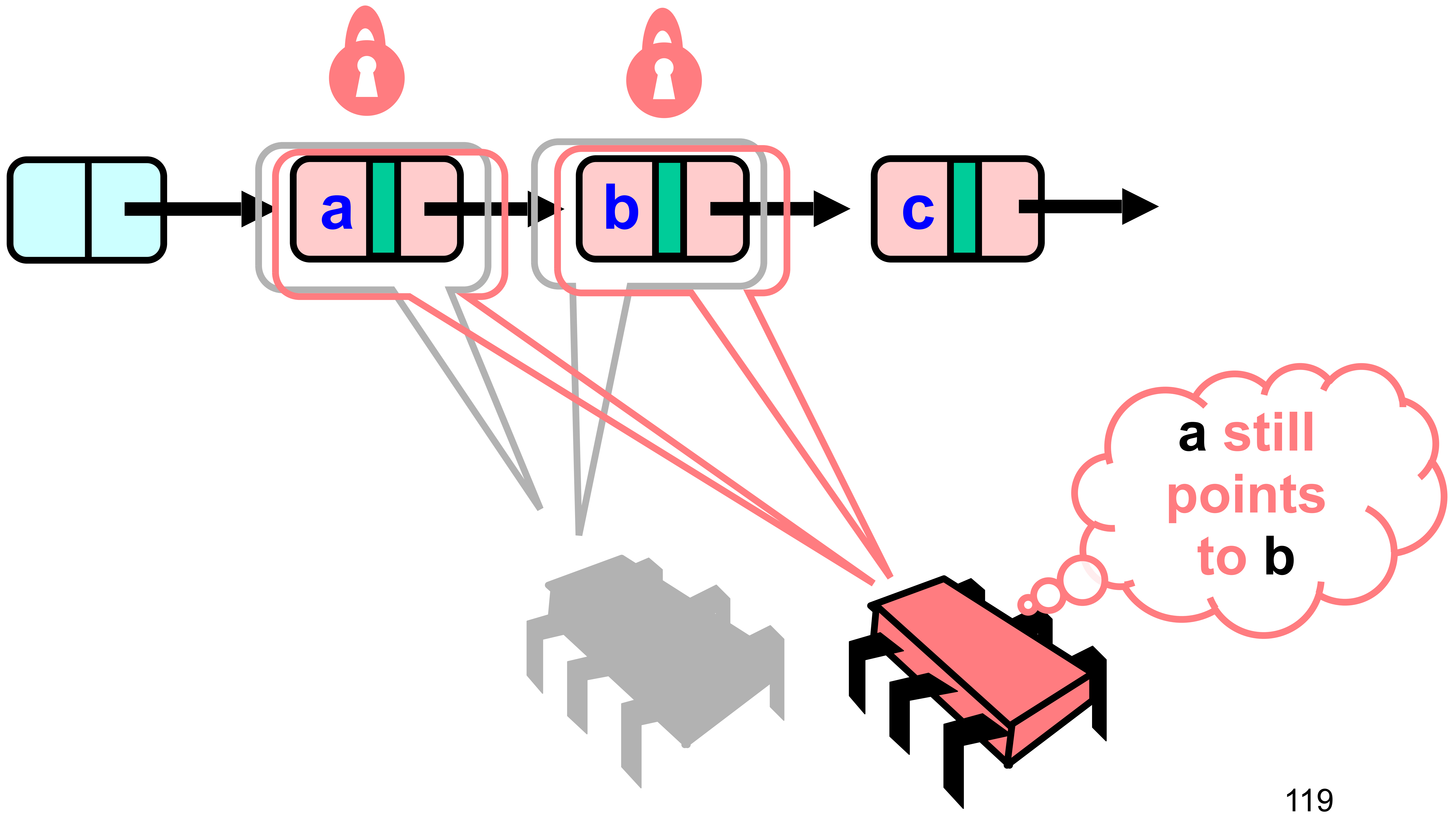
# Business as Usual



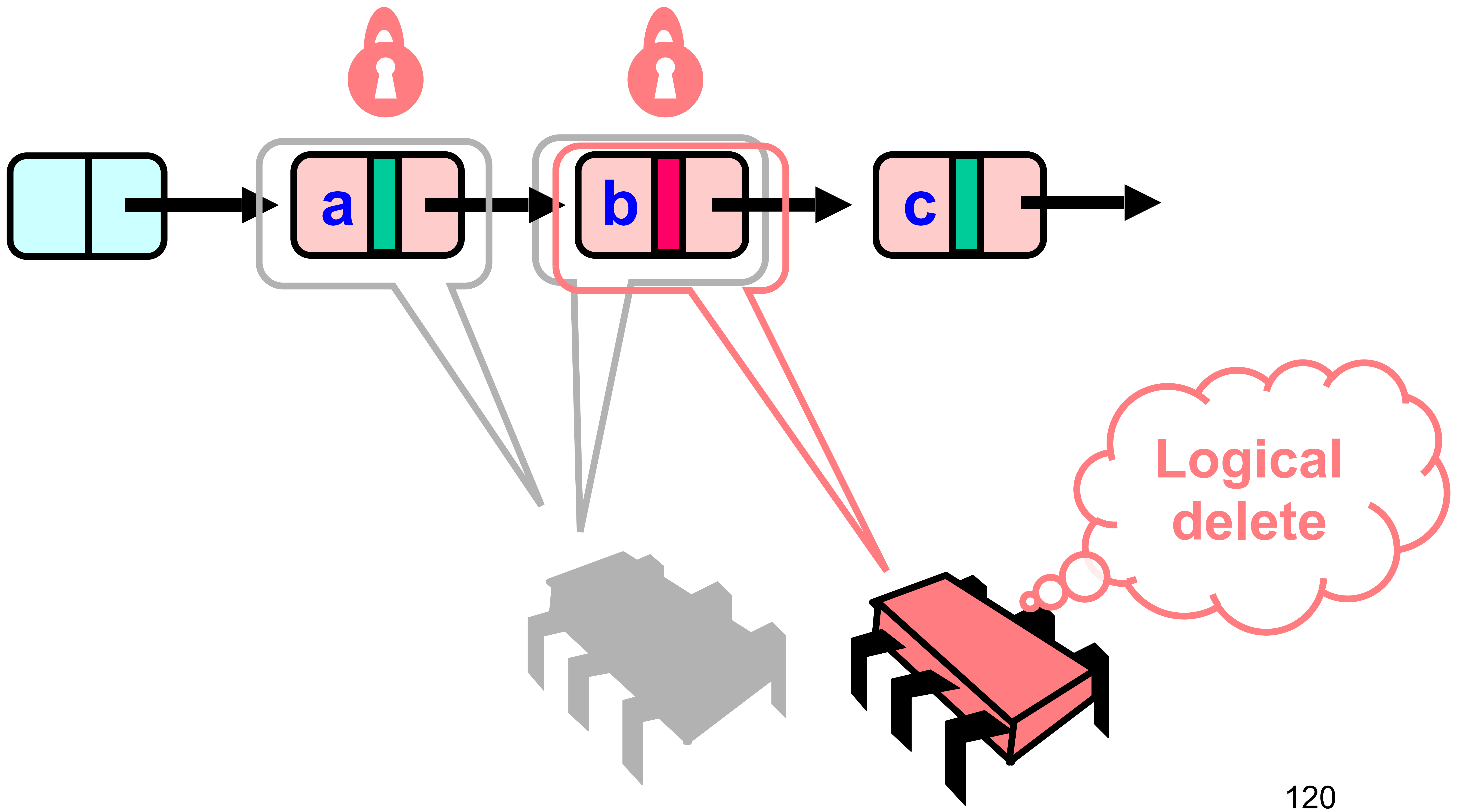
# Business as Usual



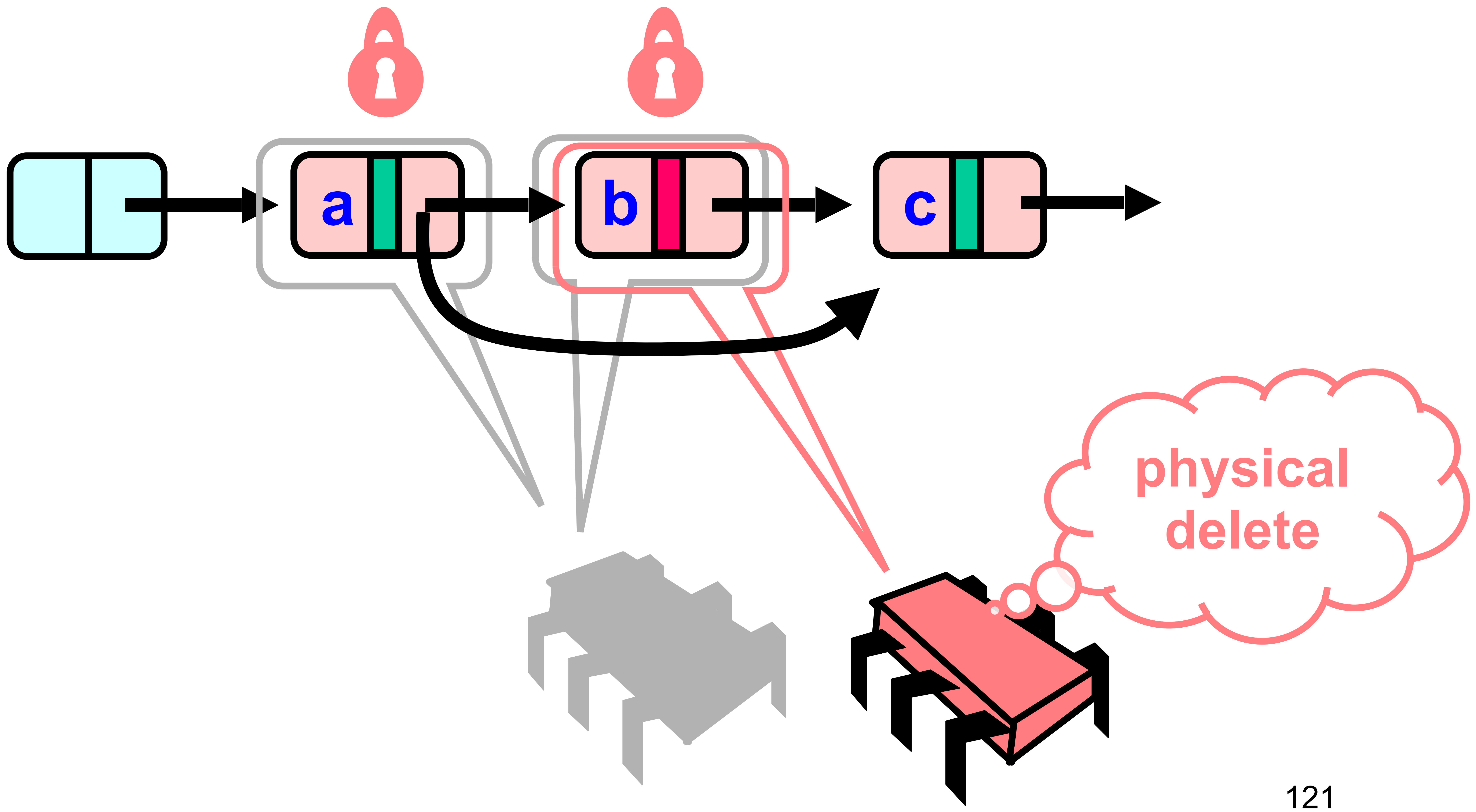
# Business as Usual



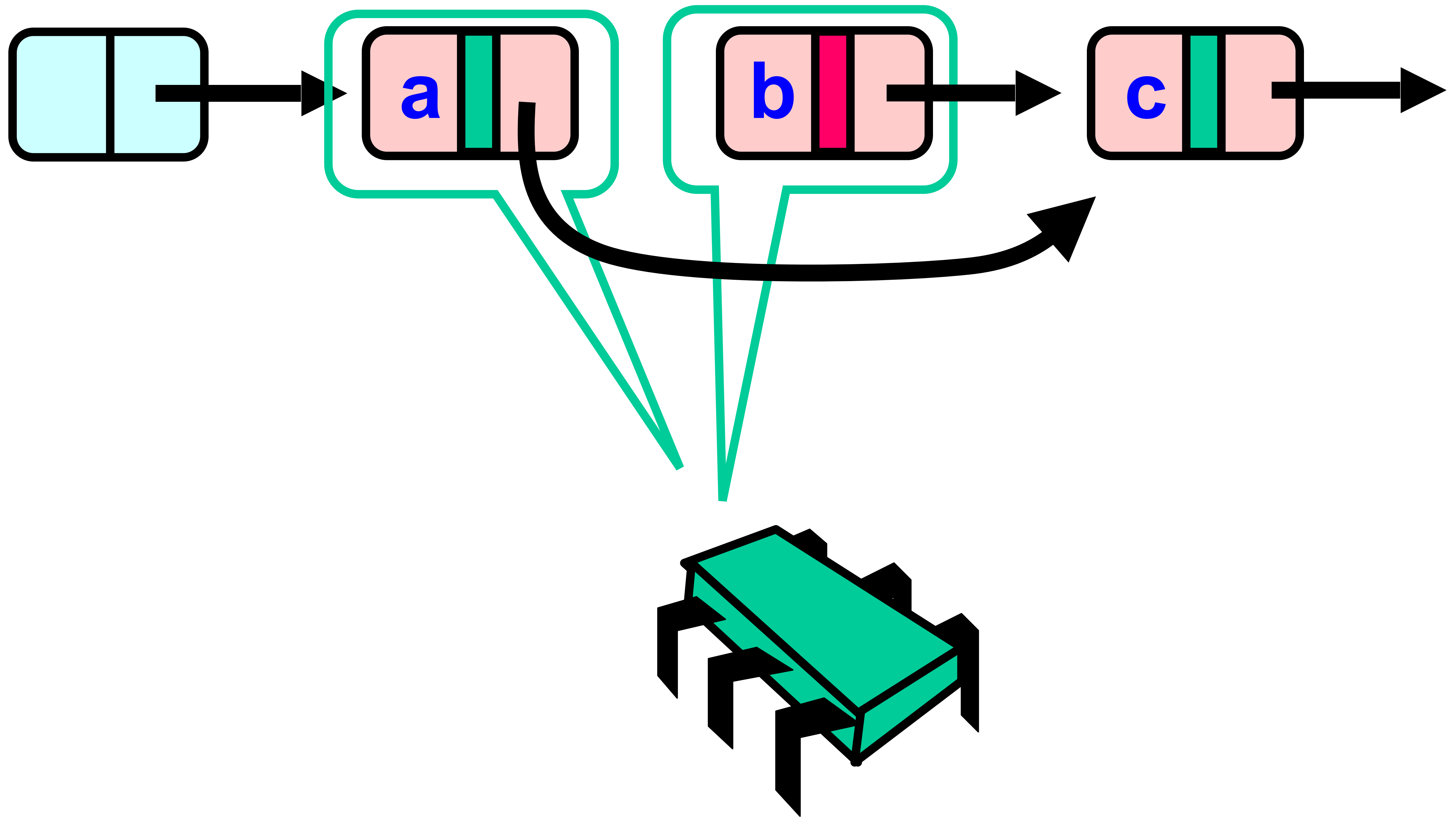
## Business as Usual



# Business as Usual



# Business as Usual



## New Abstraction Map

- S(head) =

  { x | there exists node a such that
  - a reachable from head and
  - a.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
}
```

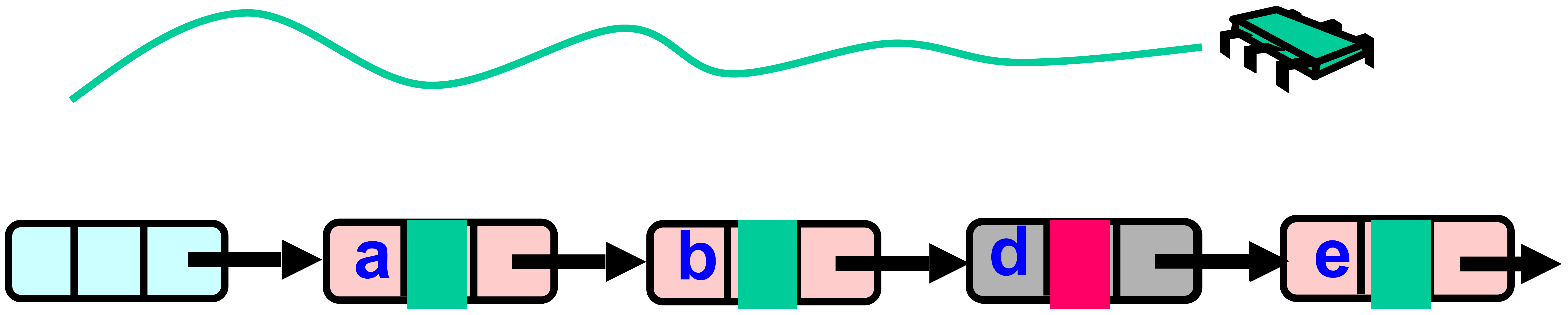
**Traverse *without locking***
**(nodes may have been removed)**

# 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?**

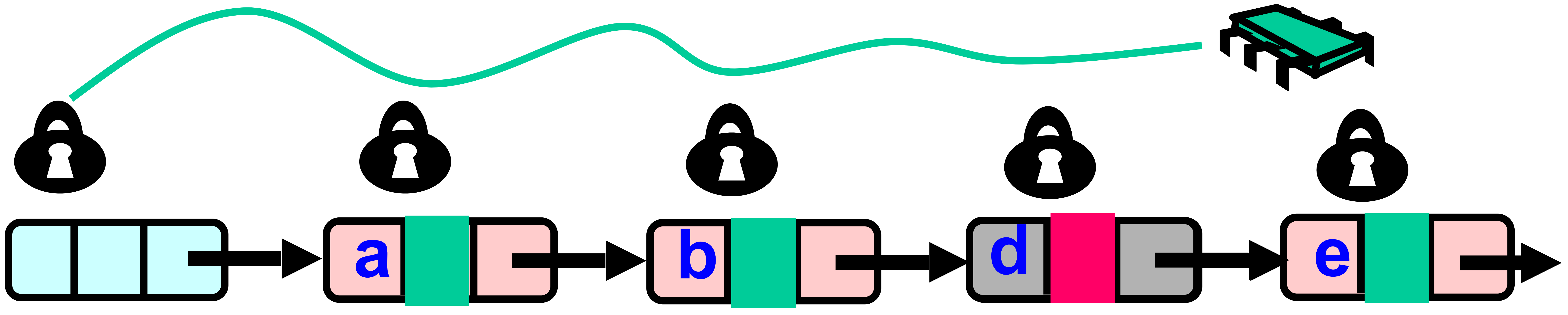
## Summary: Wait-free Contains



Use Mark bit + list ordering

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

# Lazy List



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

## Evaluation

- Good:
  - `contains()` doesn't lock
  - In fact, its 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….

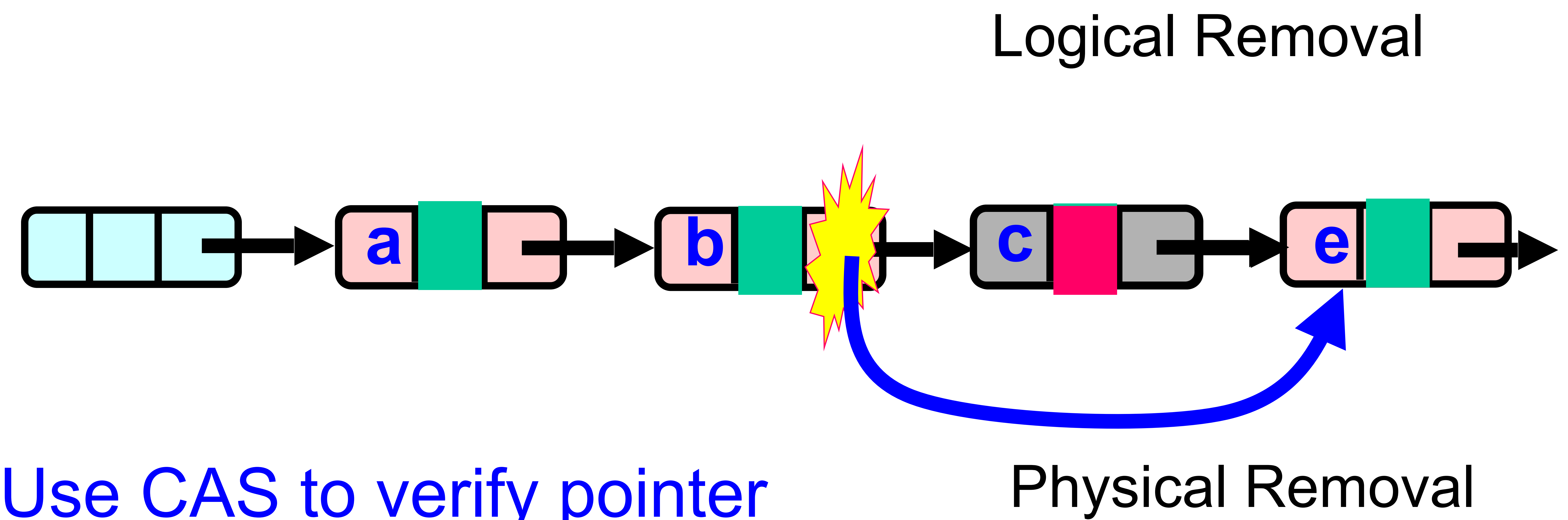
## Reminder: Lock-Free Data Structures

- 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?

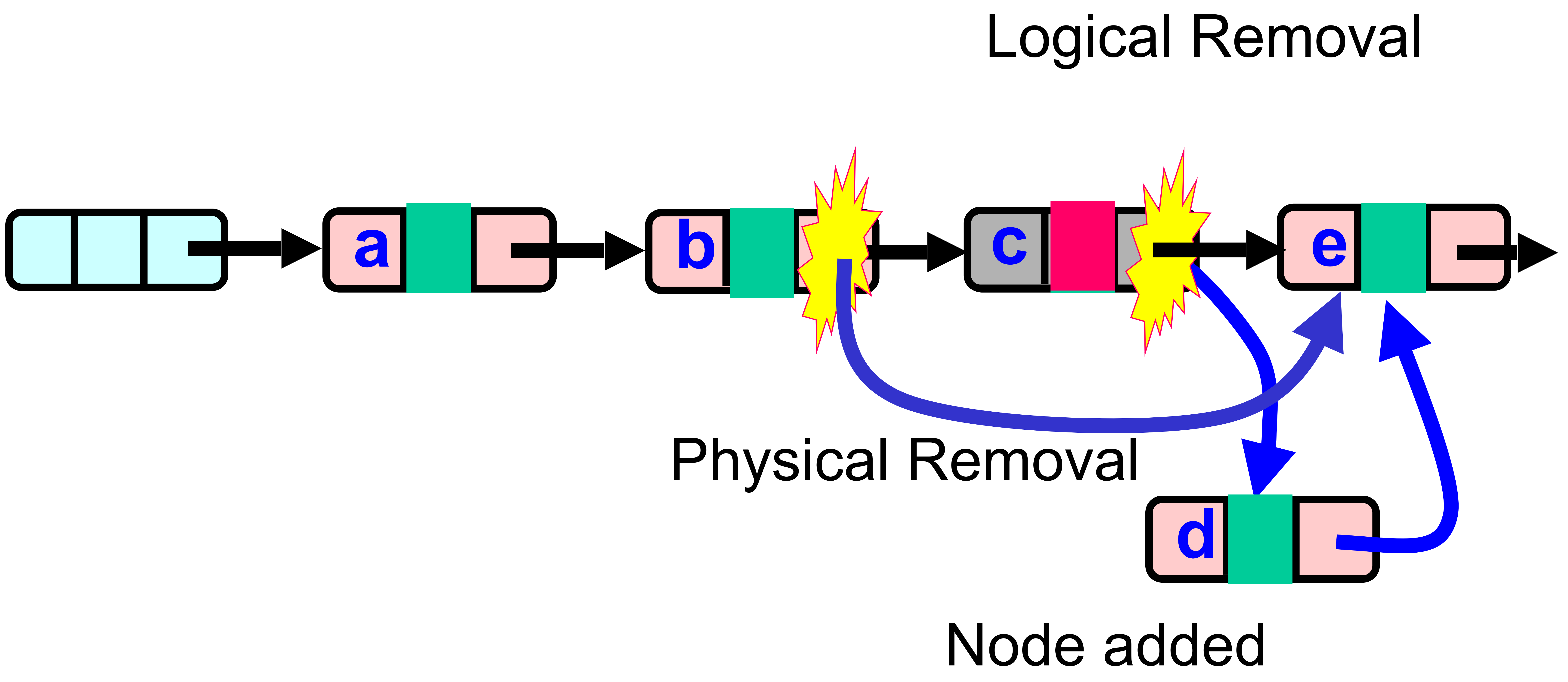
# Lock-free Lists



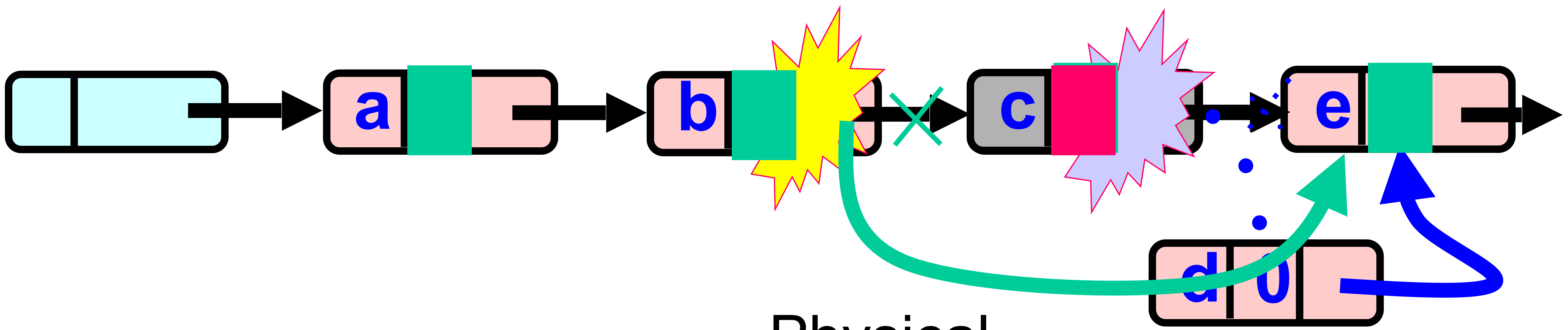
Use CAS to verify pointer is correct

Not enough!

# Problem…



# The Solution: Combine Bit and Pointer



Logical Removal = Set Mark Bit

Physical Removal CAS

Fail CAS: Node not added after logical Removal

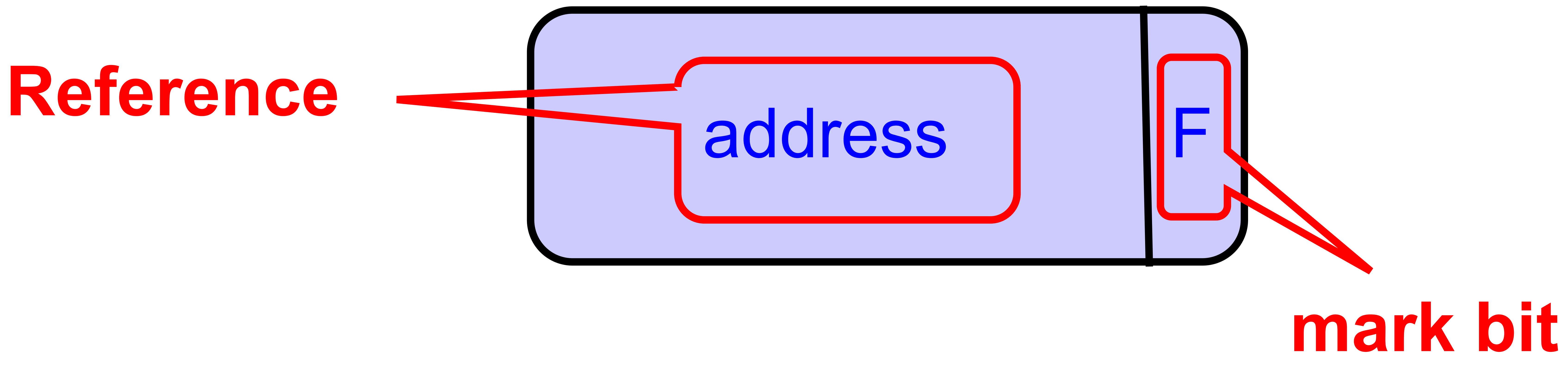
Mark-Bit and Pointer are CASed together (**AtomicMarkableReference**)

## 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

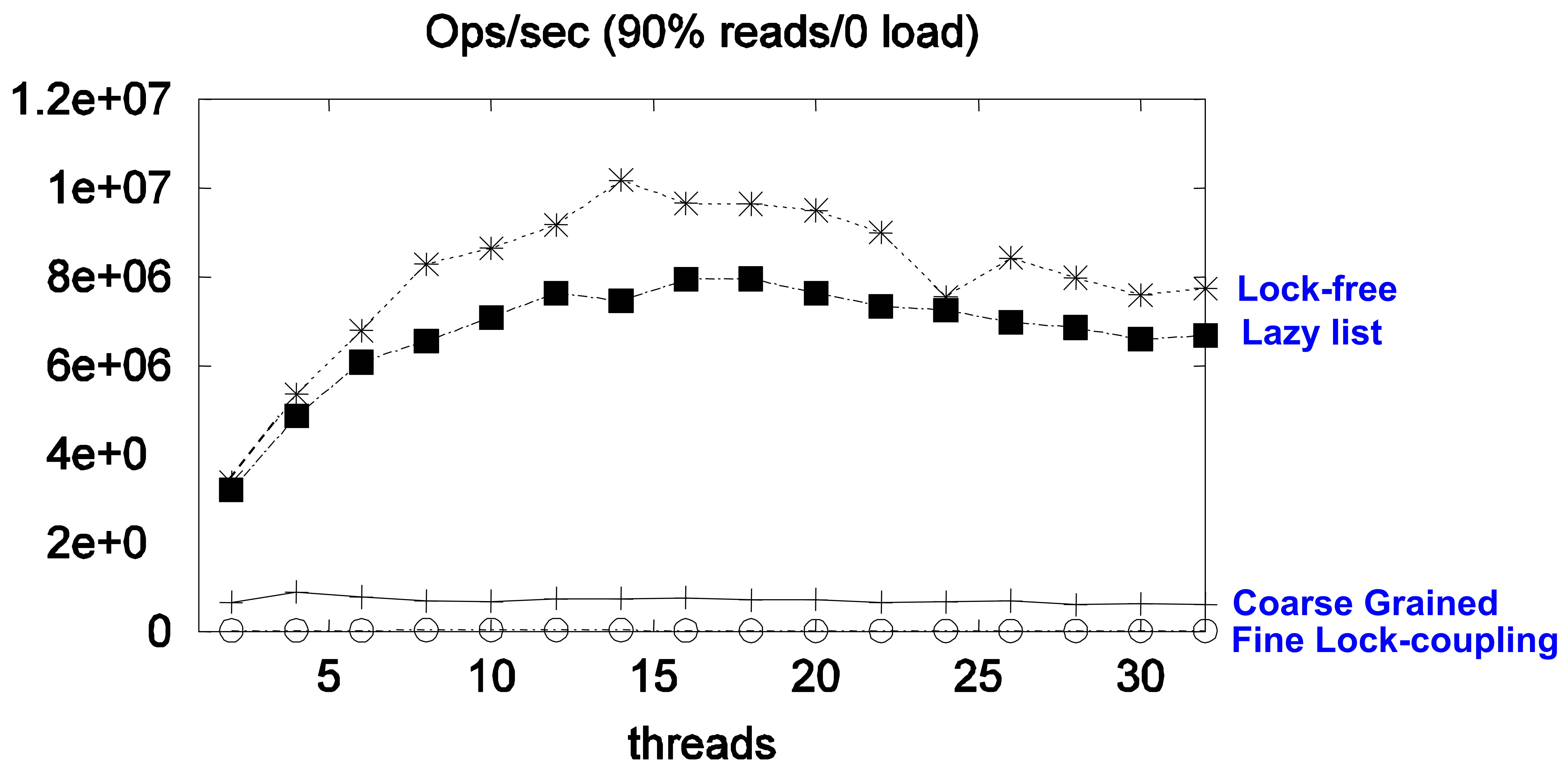
- **AtomicMarkableReference** class
  - Java.util.concurrent.atomic package



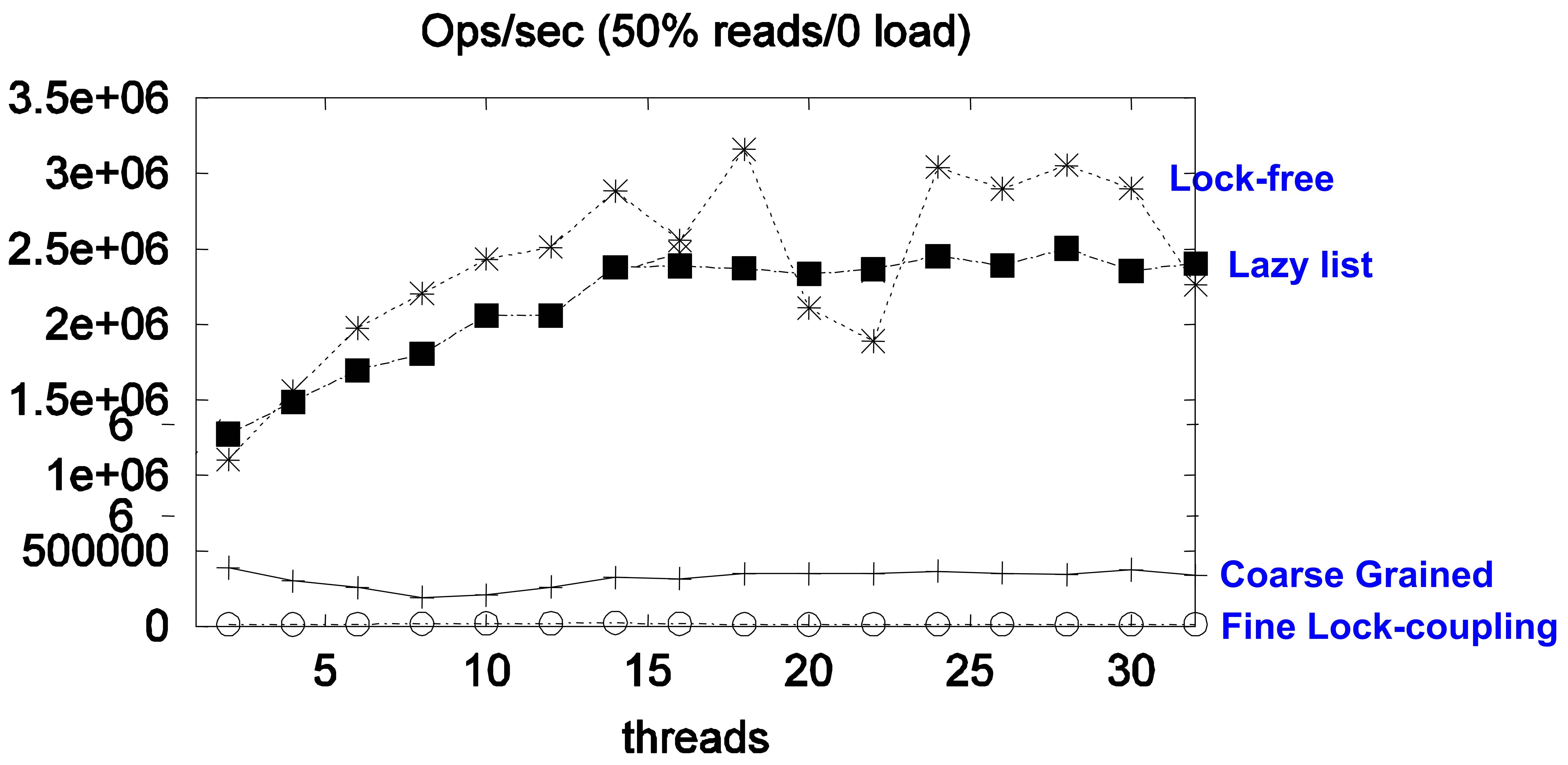
## Performance

- Different list-based set implementaions
- 16-node machine
- Vary  percentage of `contains()` calls

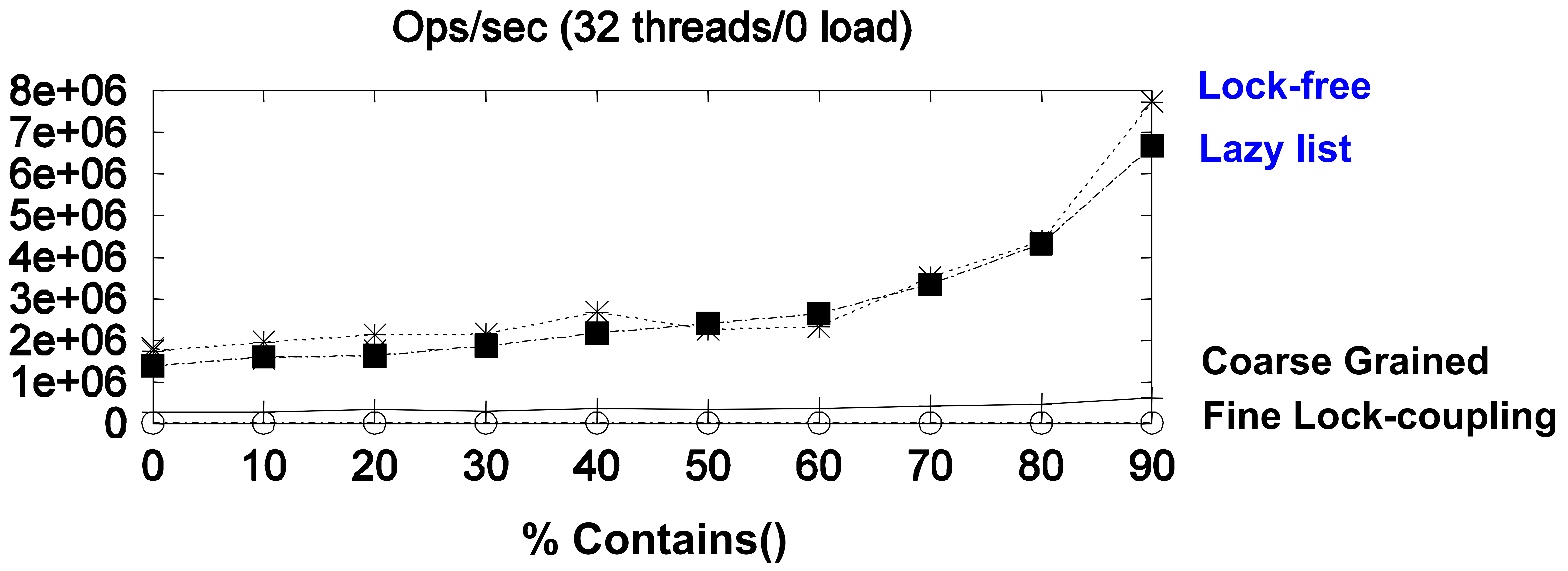
# High Contains Ratio



# 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