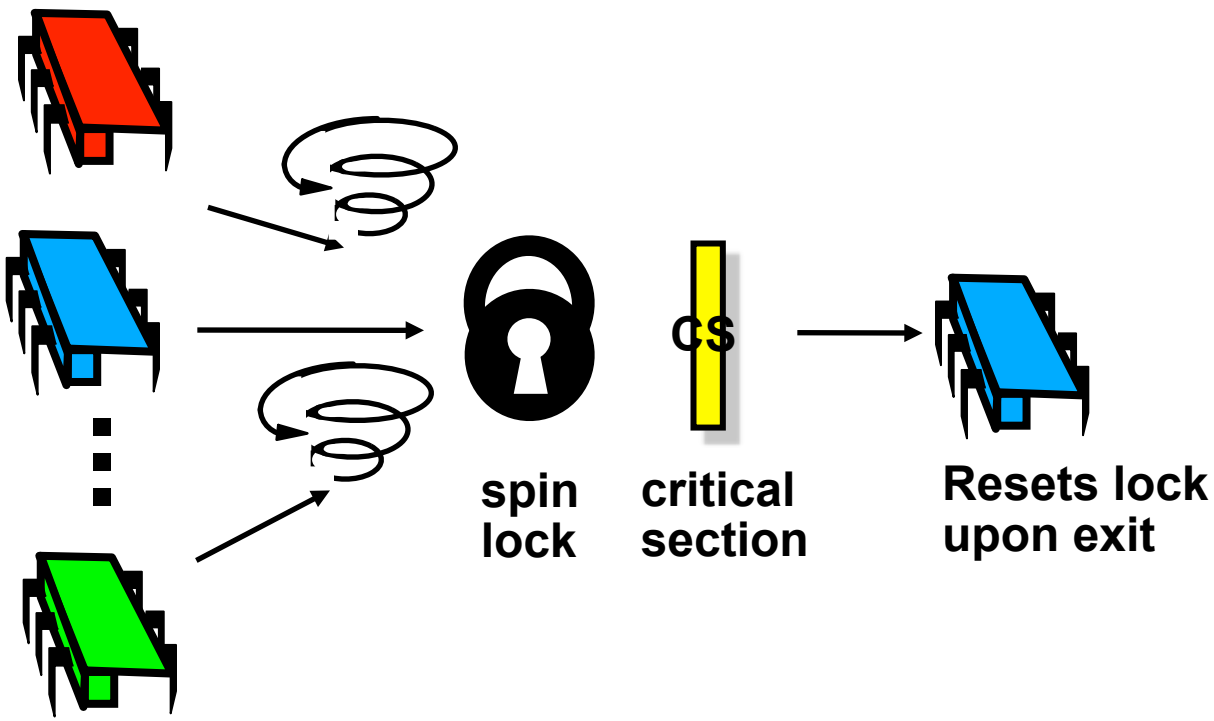
Principles of Concurrency and Parallelism

Lecture 8: Lock-free Data Structures

3/19/12

slides adapted from The Art of Multiprocessor Programming, Herlihy and Shavit

Last Lecture: Spin-Locks



Today: Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by Queue locks

Today: Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by Queue locks
- 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
  - Easy to reason about
    - In simple cases
- So, are we done?

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 …

Fine-Grained Synchronization

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

Second: Optimistic Synchronization

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

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 …
- 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
- No duplicates
- Methods
  - **add(x)** put **x** in set
  - **remove(x)** take **x** out of set
  - **contains(x)** tests if **x** in set

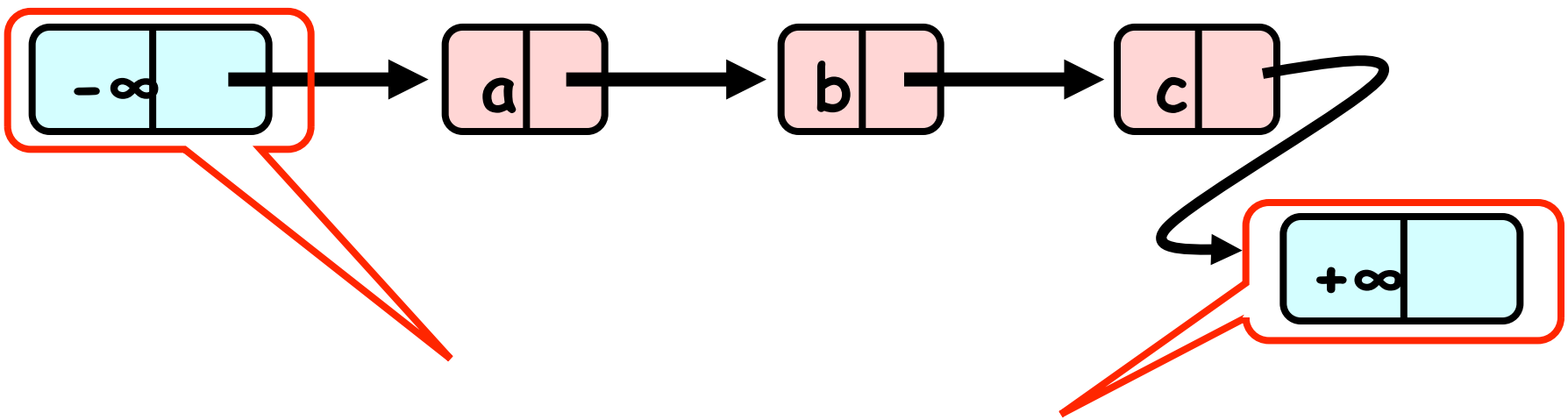
List-Based Sets

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

List Node

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

The List-Based Set



Sorted with Sentinel nodes
(min & max possible keys)

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()**
- Most steps are trivial
  - Usually one step tricky
  - Often linearization point
    - point at which method's effects can be safely made visible

Interference

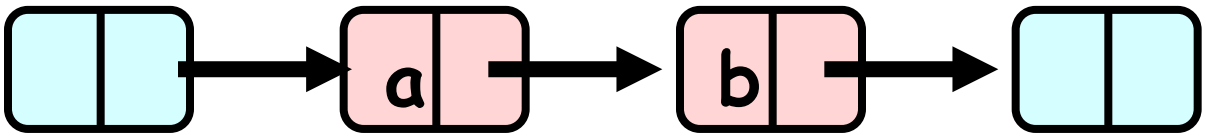
- Invariants make sense only if
  - methods considered
  - are the only modifiers
- Language encapsulation helps
  - List nodes not visible outside class

Interference

- Freedom from interference needed even for removed nodes
  - Some algorithms traverse removed nodes
  - Careful with **malloc()** & **free()!**
- Garbage collection helps here

Abstract Data Types

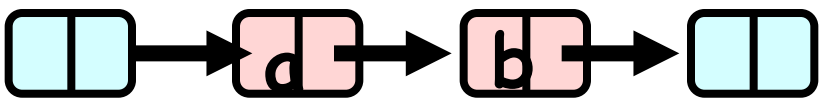
- Concrete representation:



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

Abstract Data Types

- Meaning of rep given by abstraction map
  - S( ) = {**a**,**b**}



Rep Invariant

- Which concrete values meaningful?
  - Sorted?
  - Duplicates?
- Rep invariant
  - Characterizes legal concrete reps
  - Preserved by methods
  - Relied on by methods

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

Rep Invariant (partly)

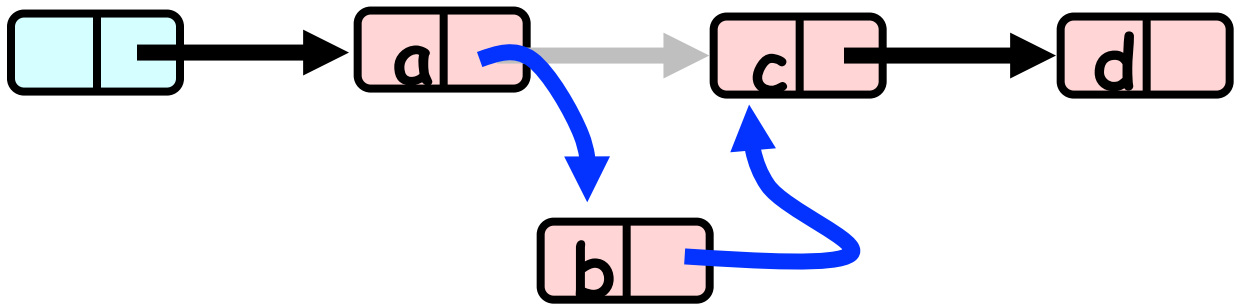
- Sentinel nodes
  - tail reachable from head
- Sorted
- No duplicates

Abstraction Map

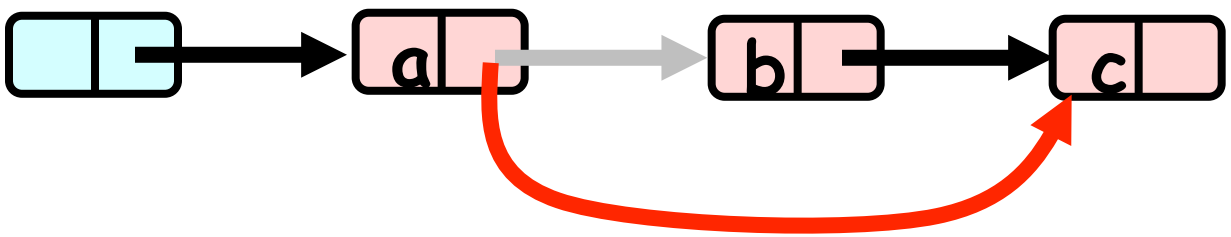
- S(head) =
  - { x | there exists a such that
    - a reachable from head and
    - a.item = x
  - }

Sequential List Based Set

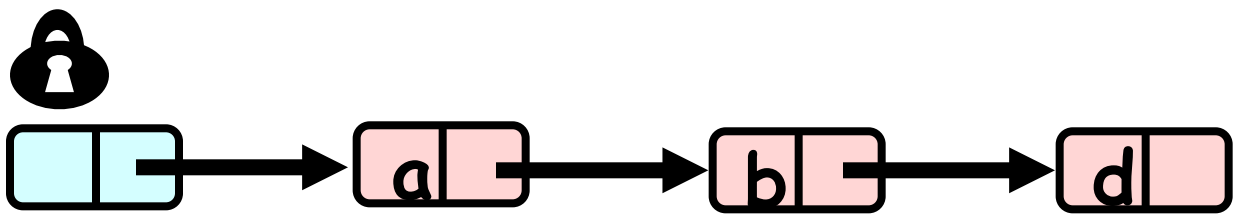
Add()



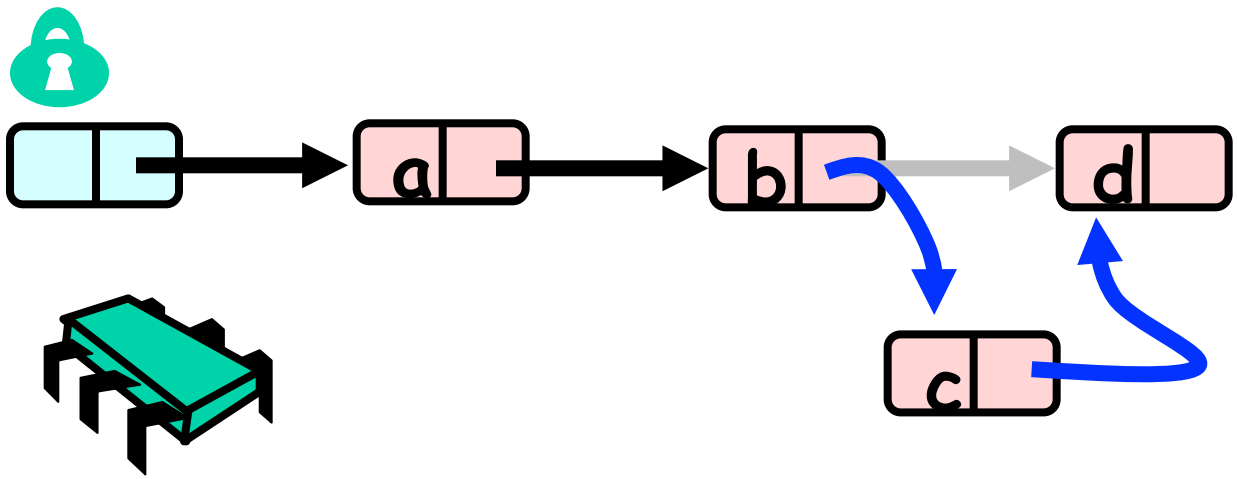
Remove()



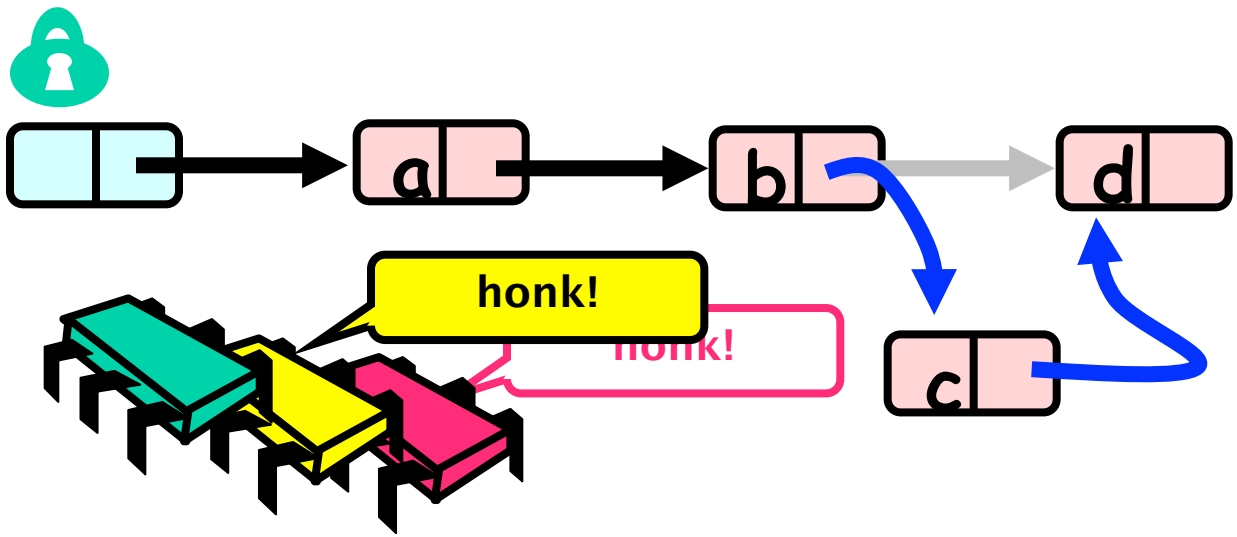
Coarse-Grained Locking



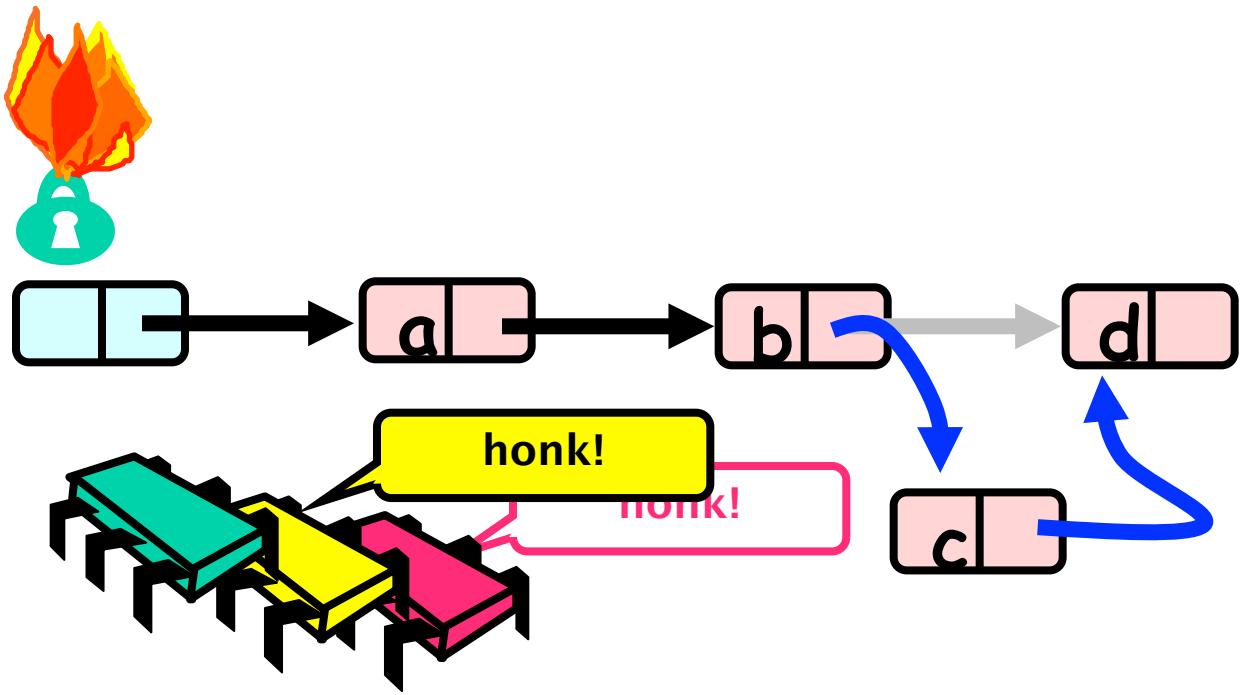
Coarse-Grained Locking



Coarse-Grained Locking



Coarse-Grained Locking



Simple but hotspot + bottleneck

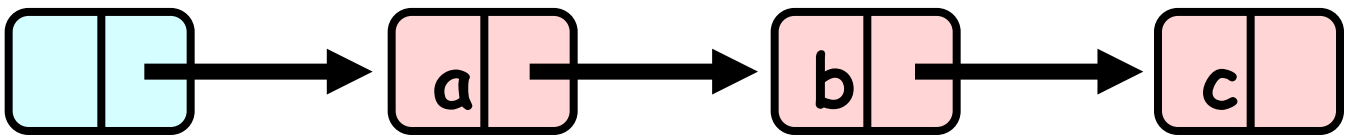
Coarse-Grained Locking

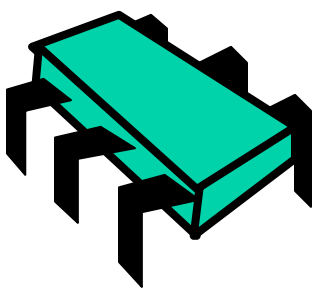
- Easy, same as synchronized methods
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention
  - Queue locks help
  - But bottleneck still an issue

Fine-grained Locking

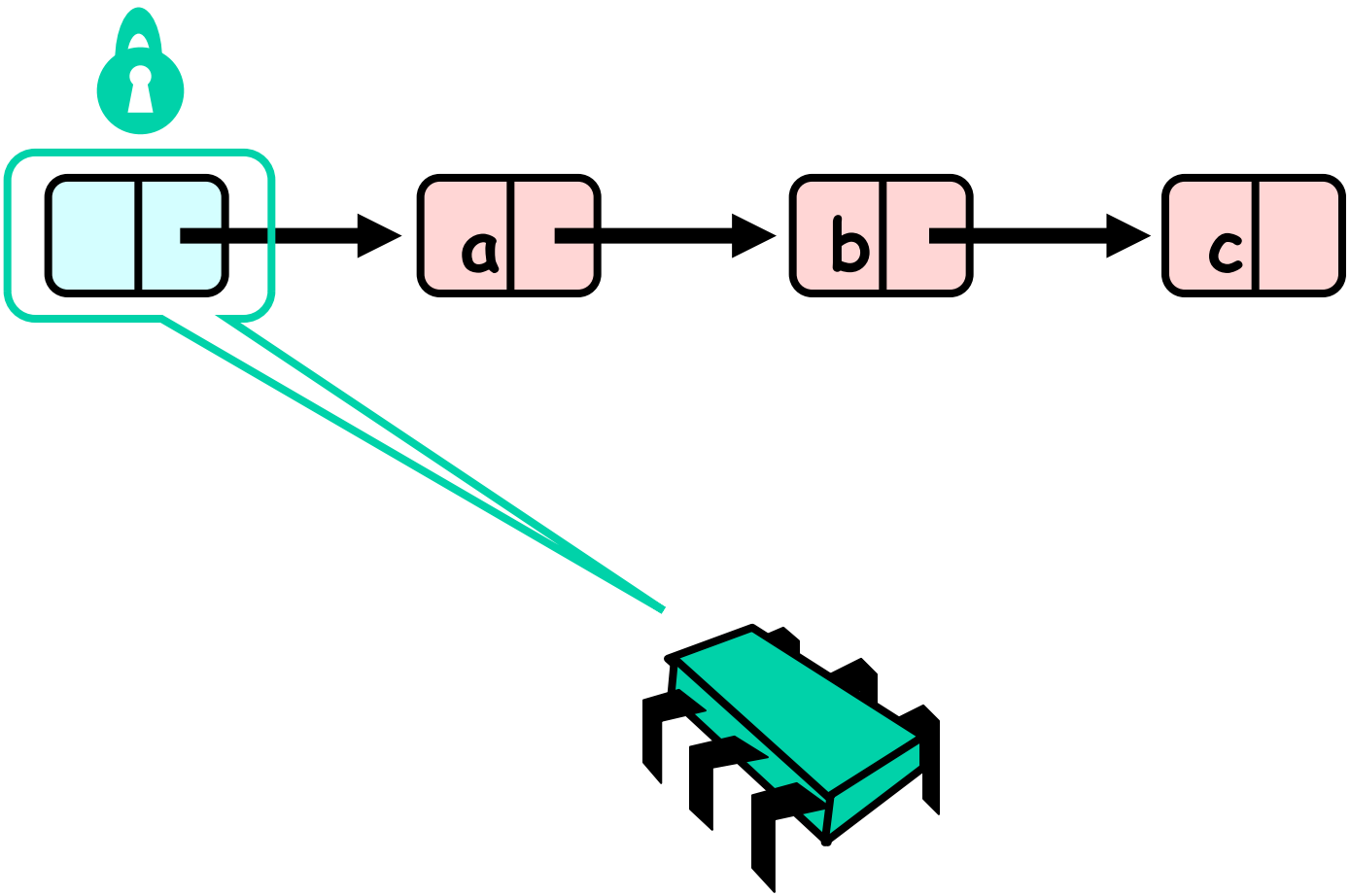
- Requires **careful** thought
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand locking

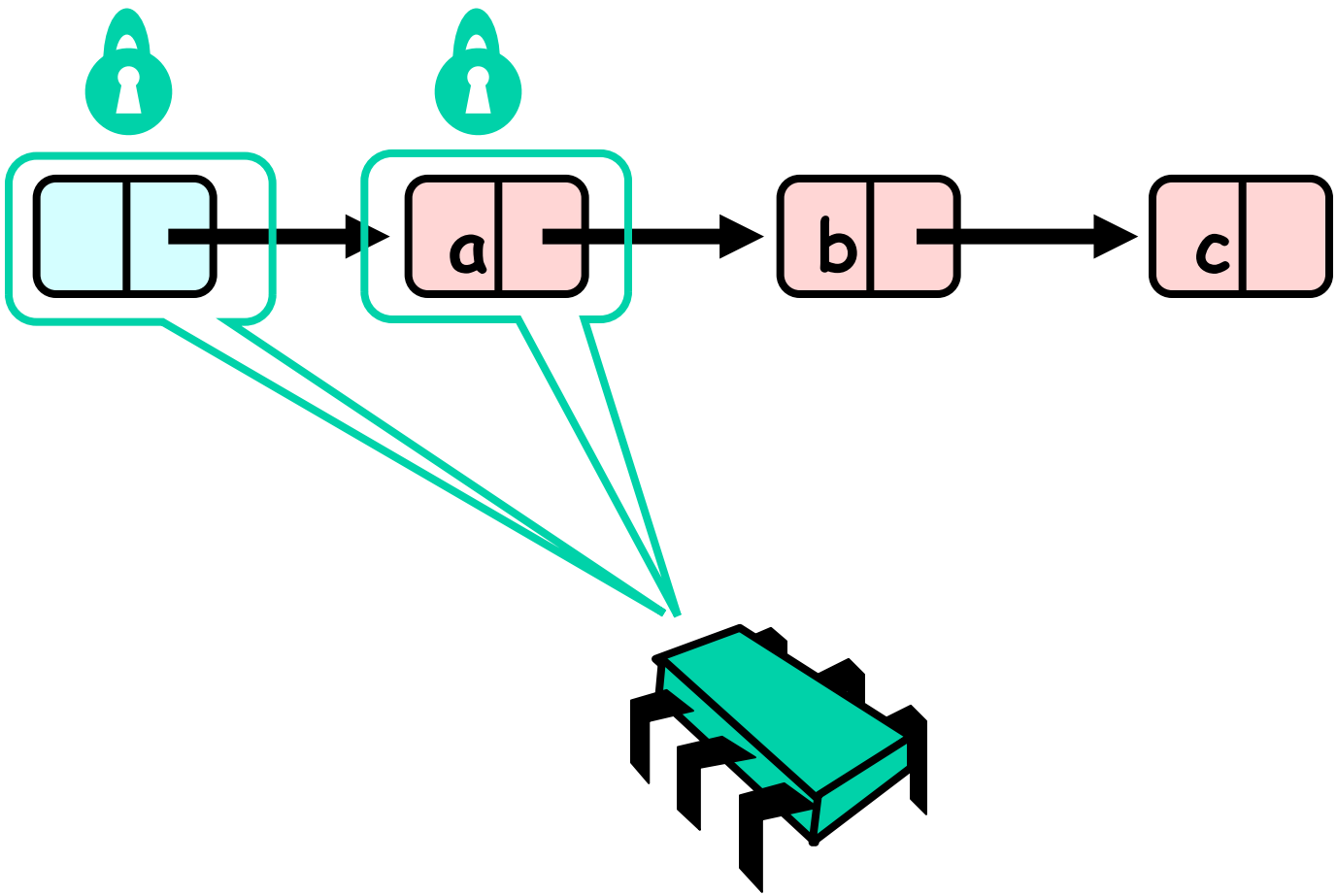




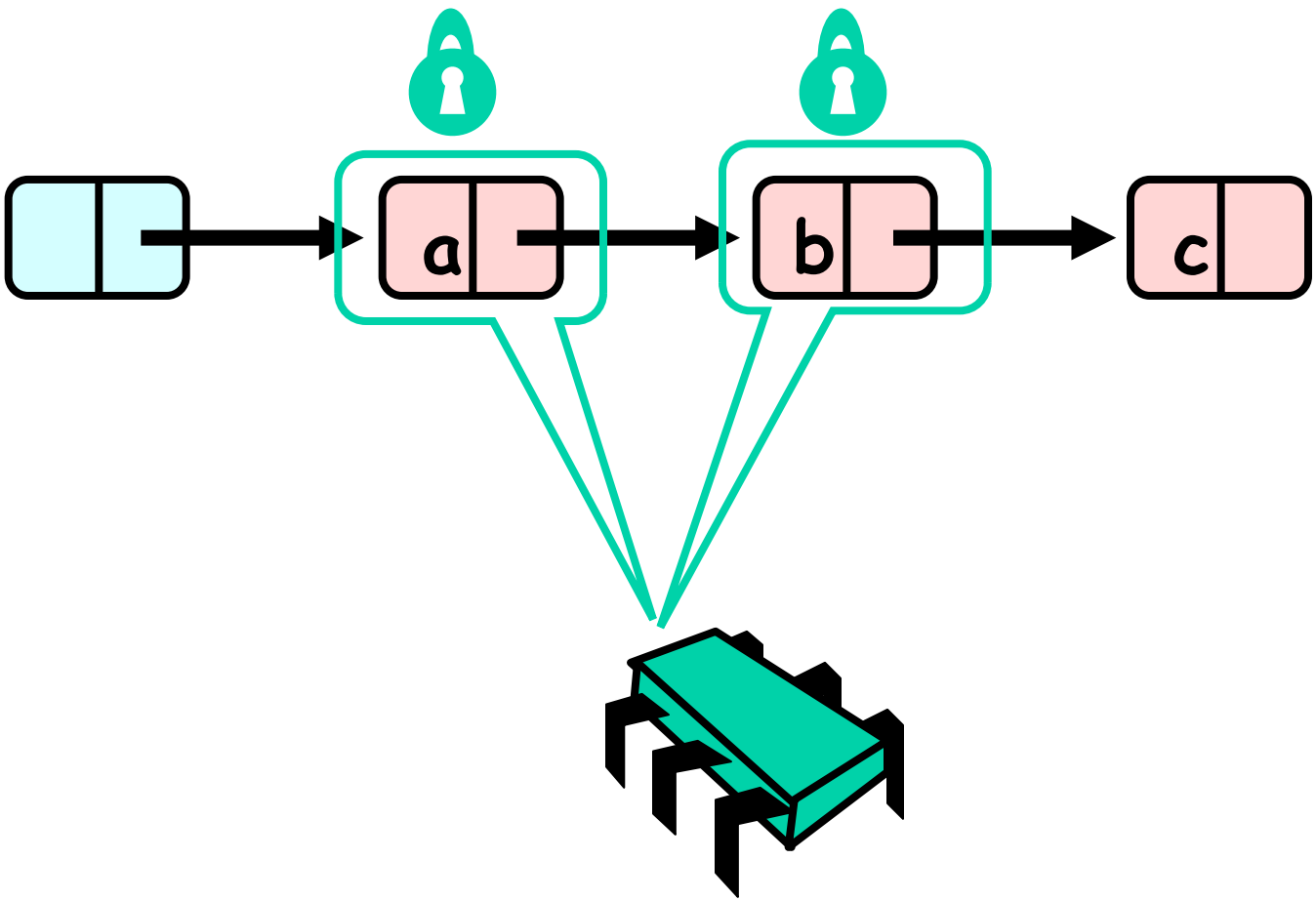
Hand-over-Hand locking



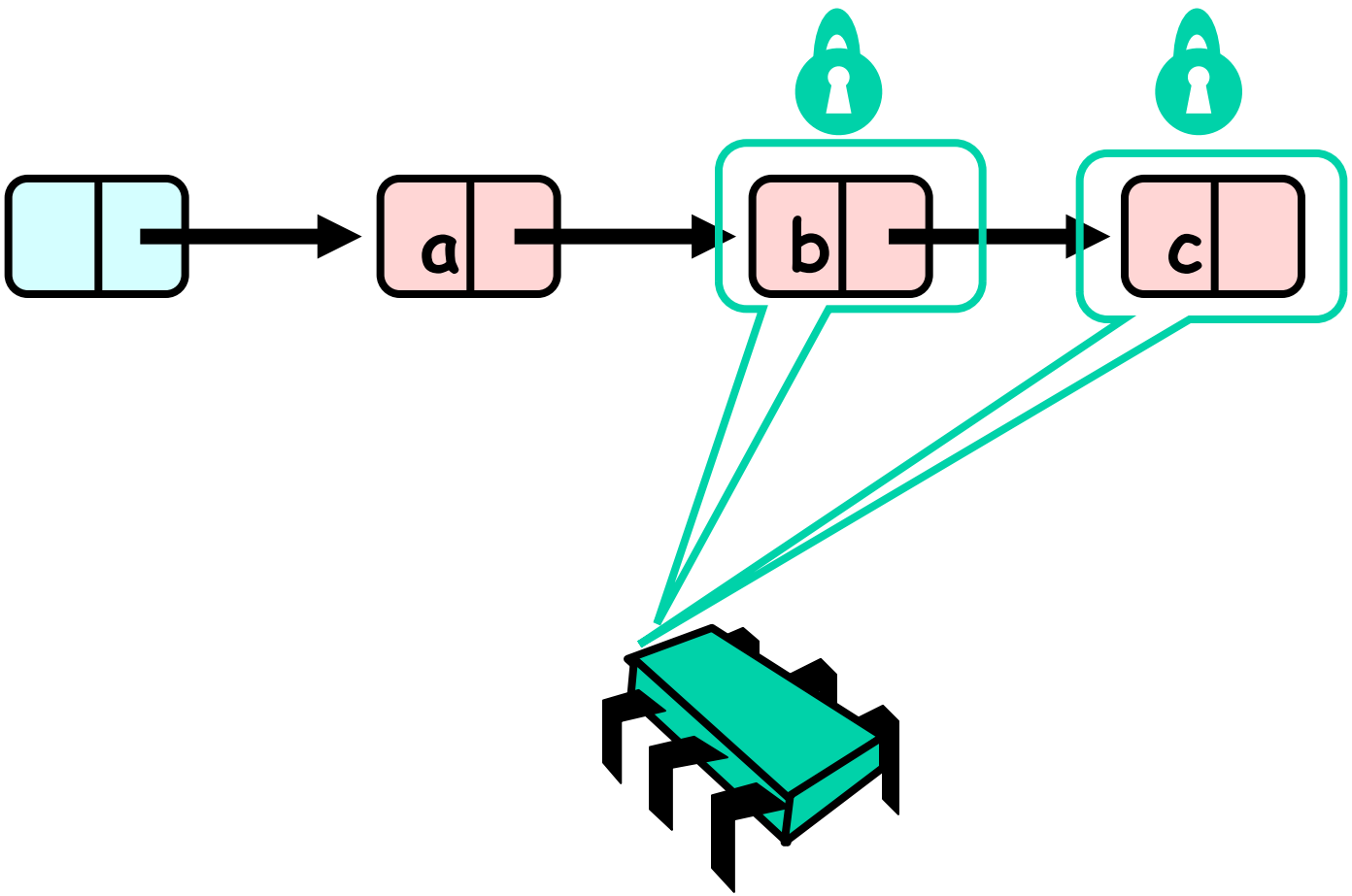
Hand-over-Hand locking



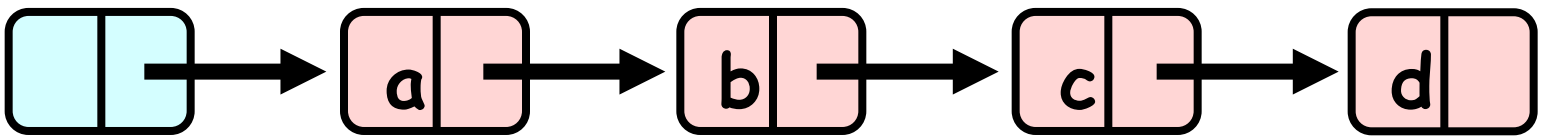
Hand-over-Hand locking

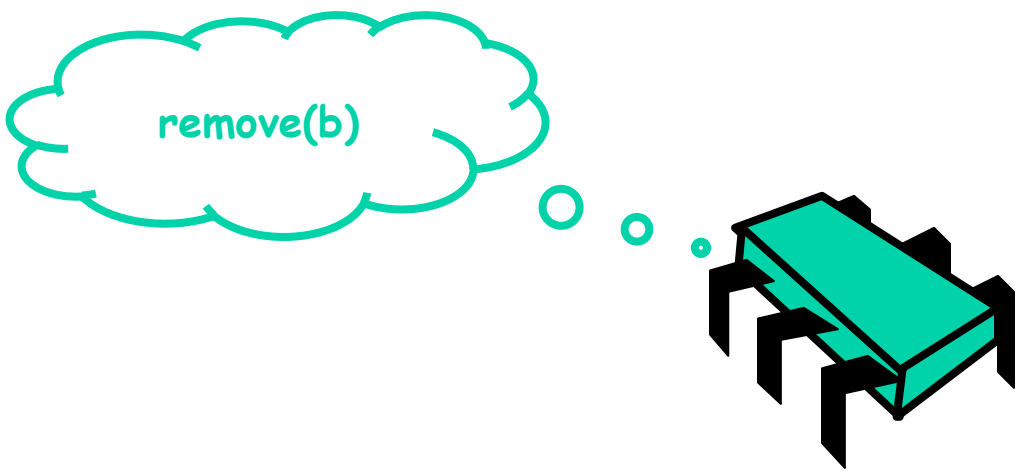


Hand-over-Hand locking

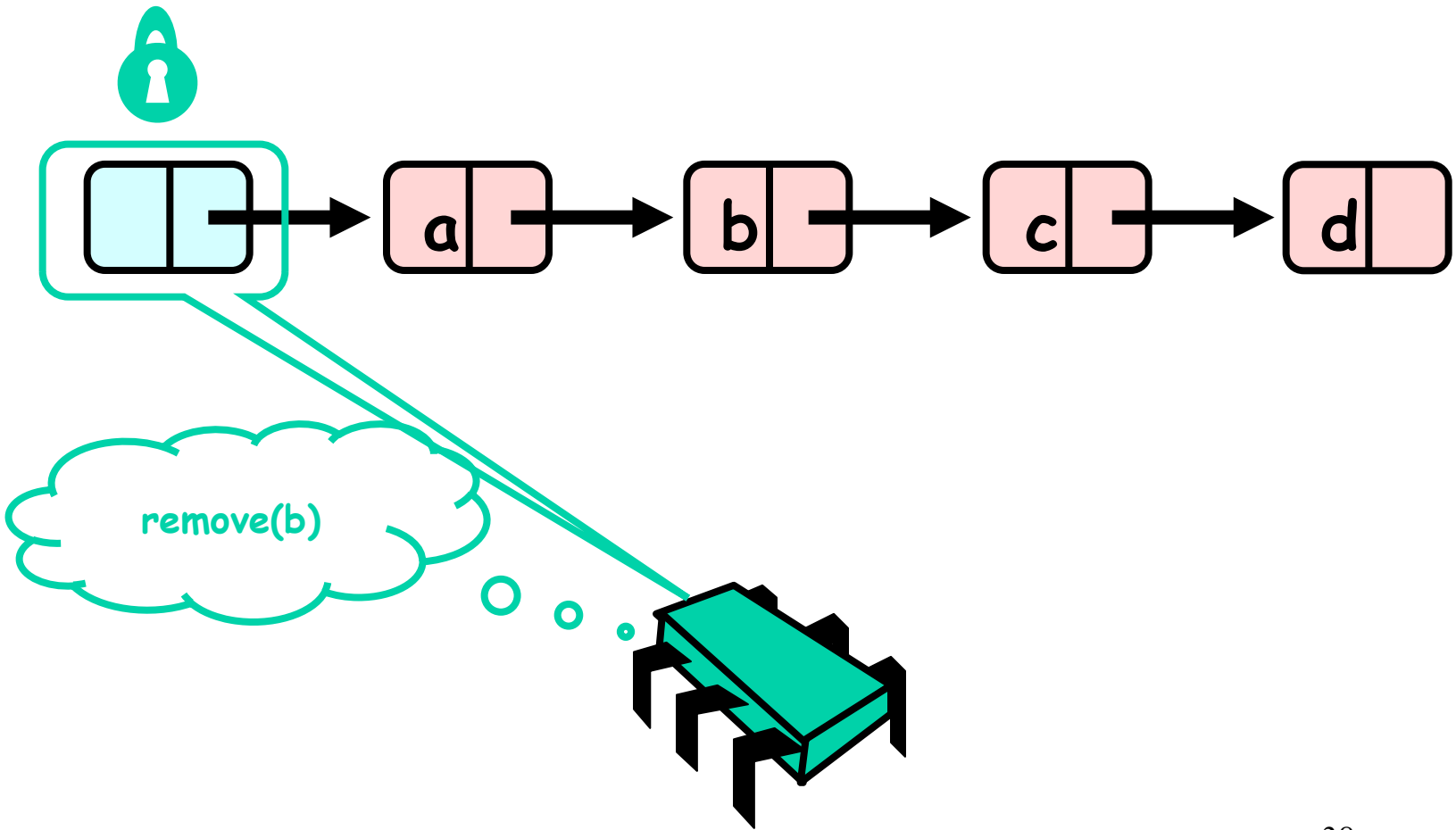


Removing a Node

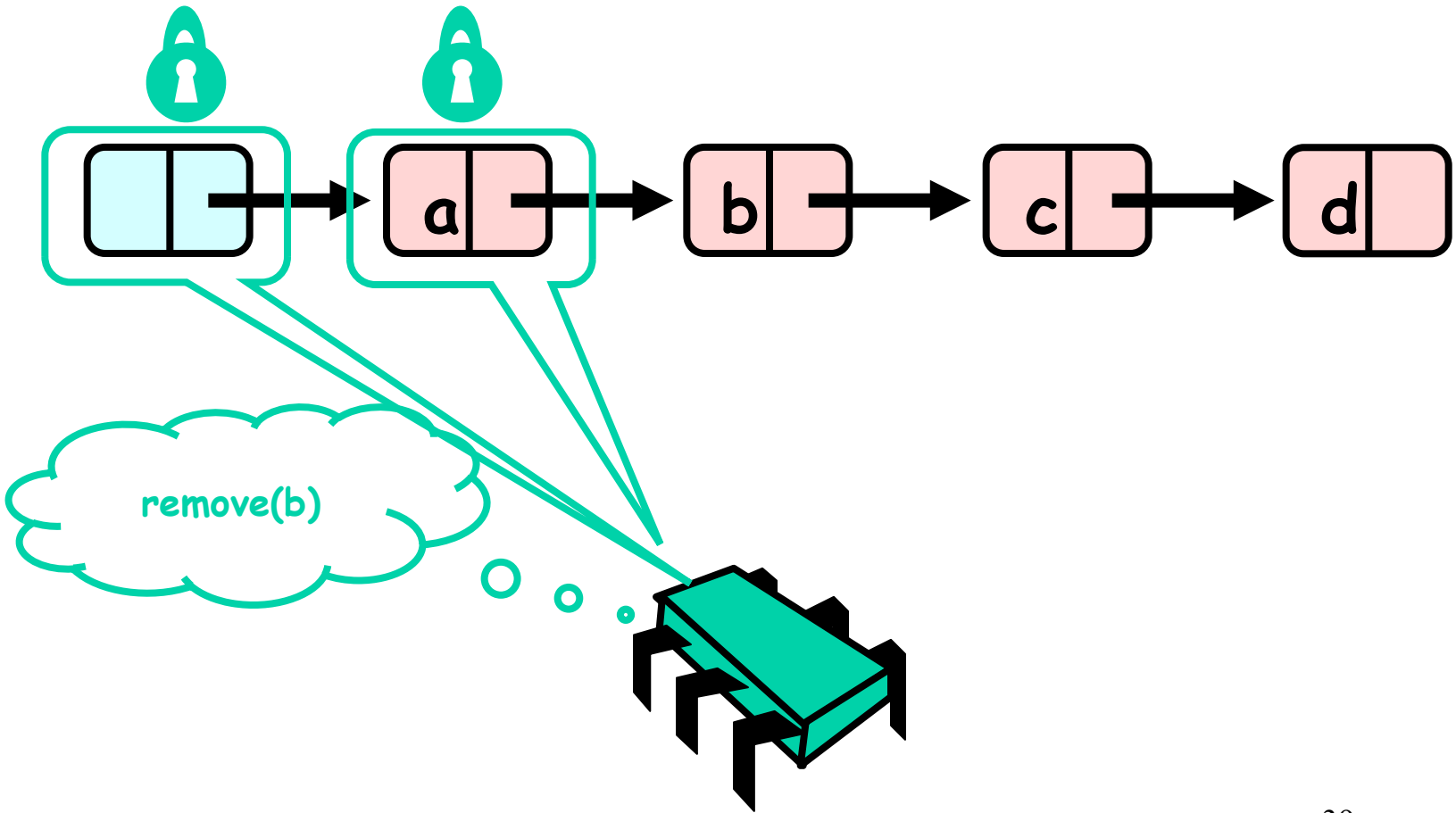




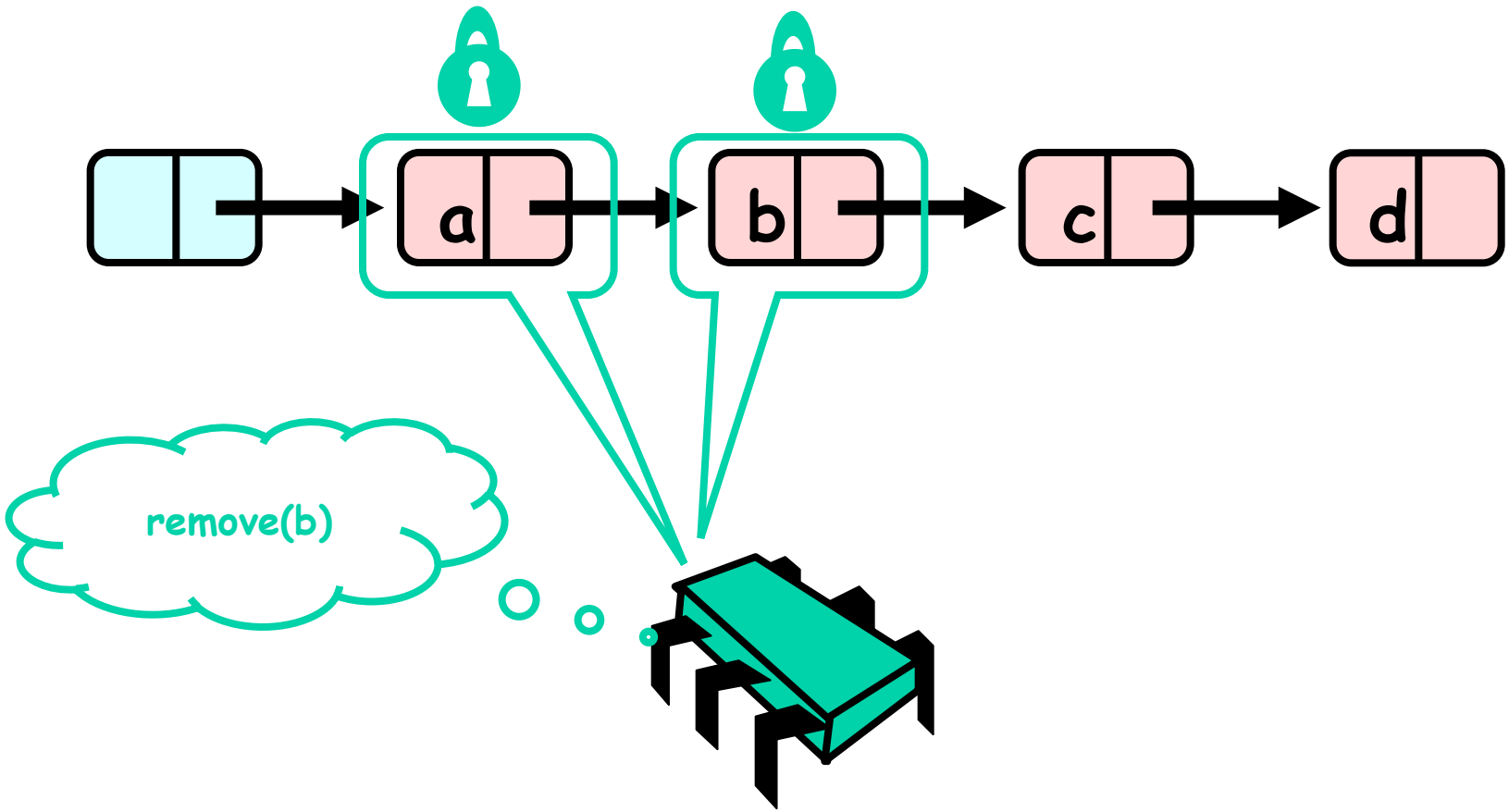
Removing a Node



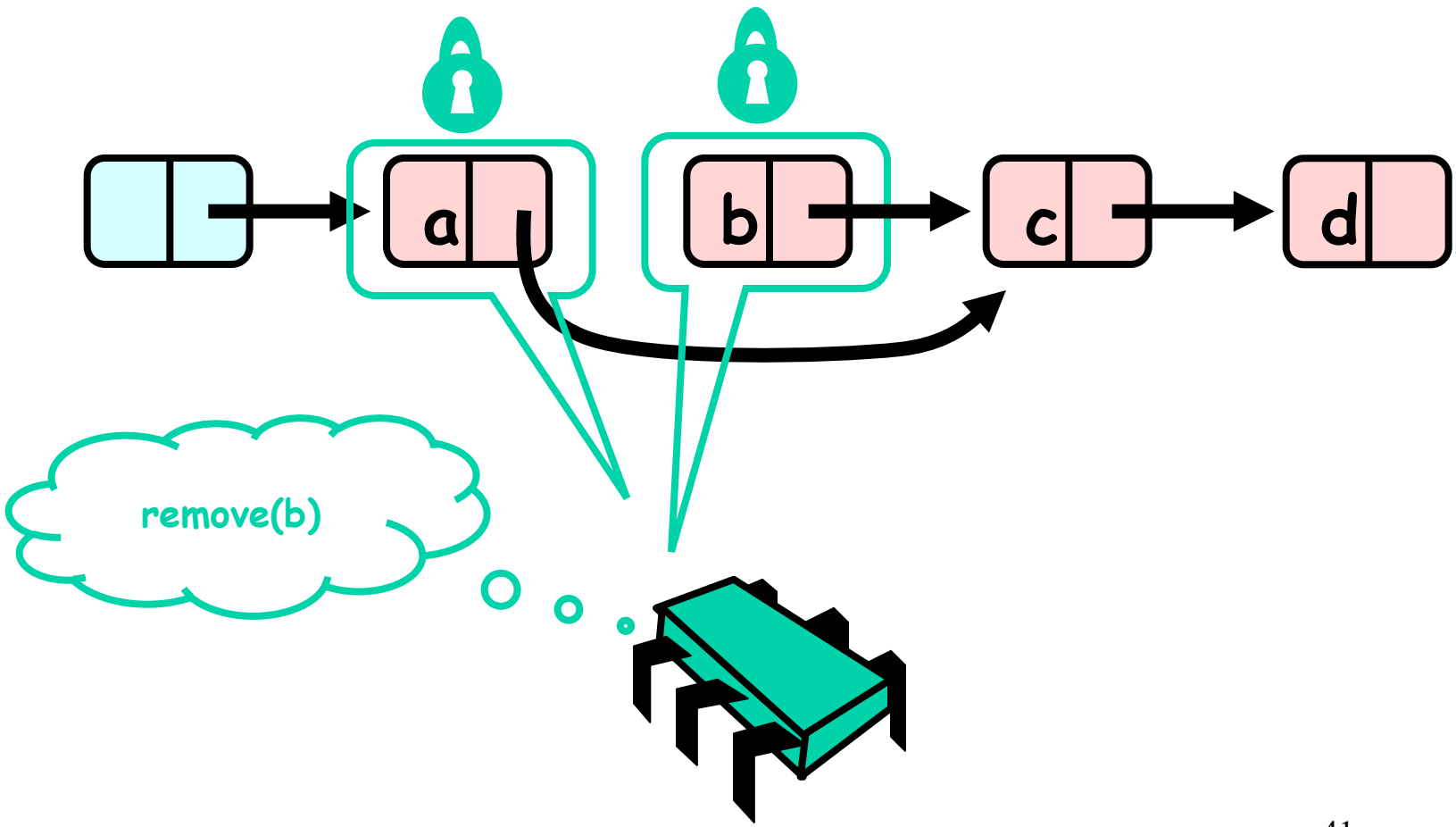
Removing a Node



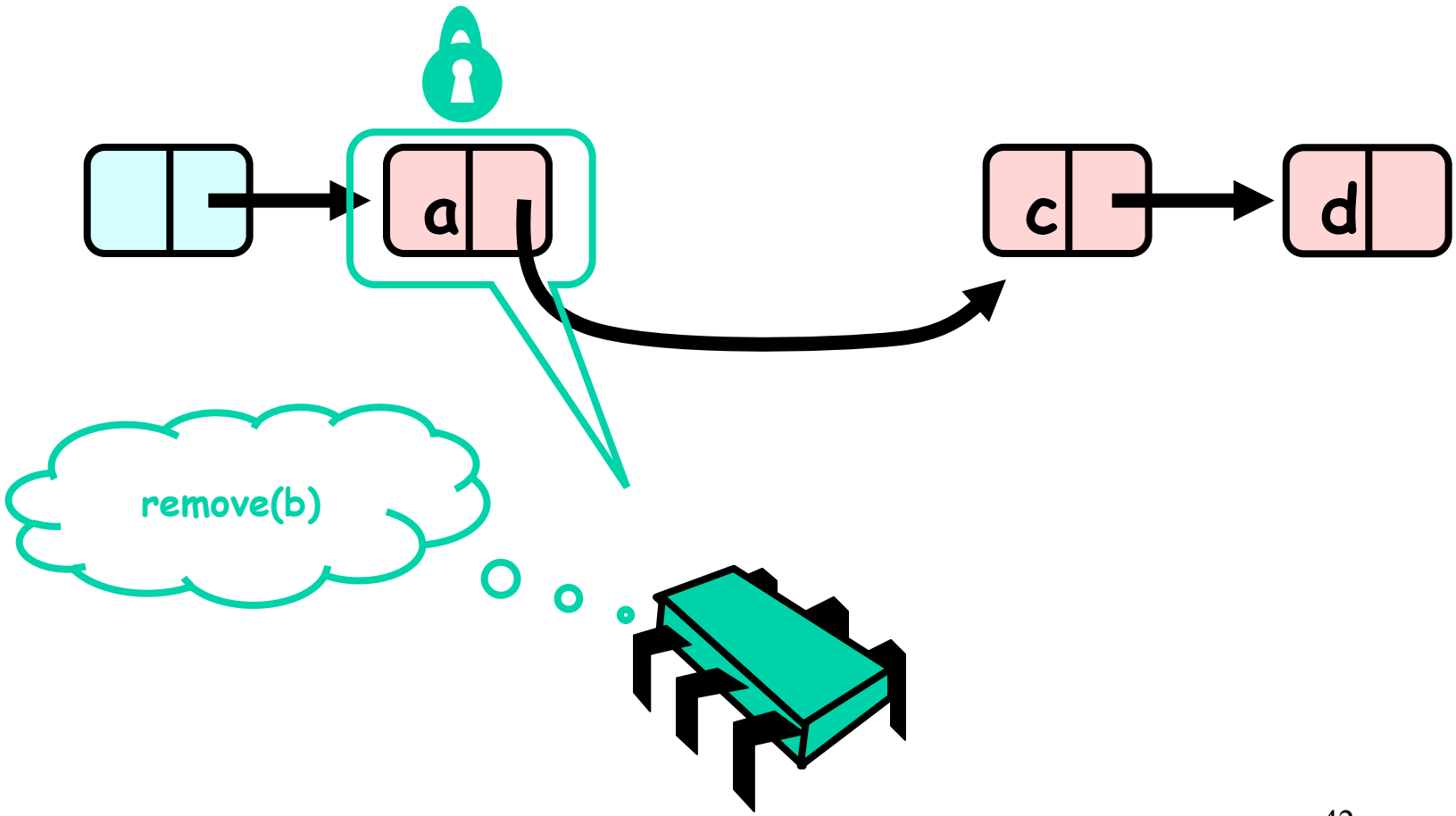
Removing a Node



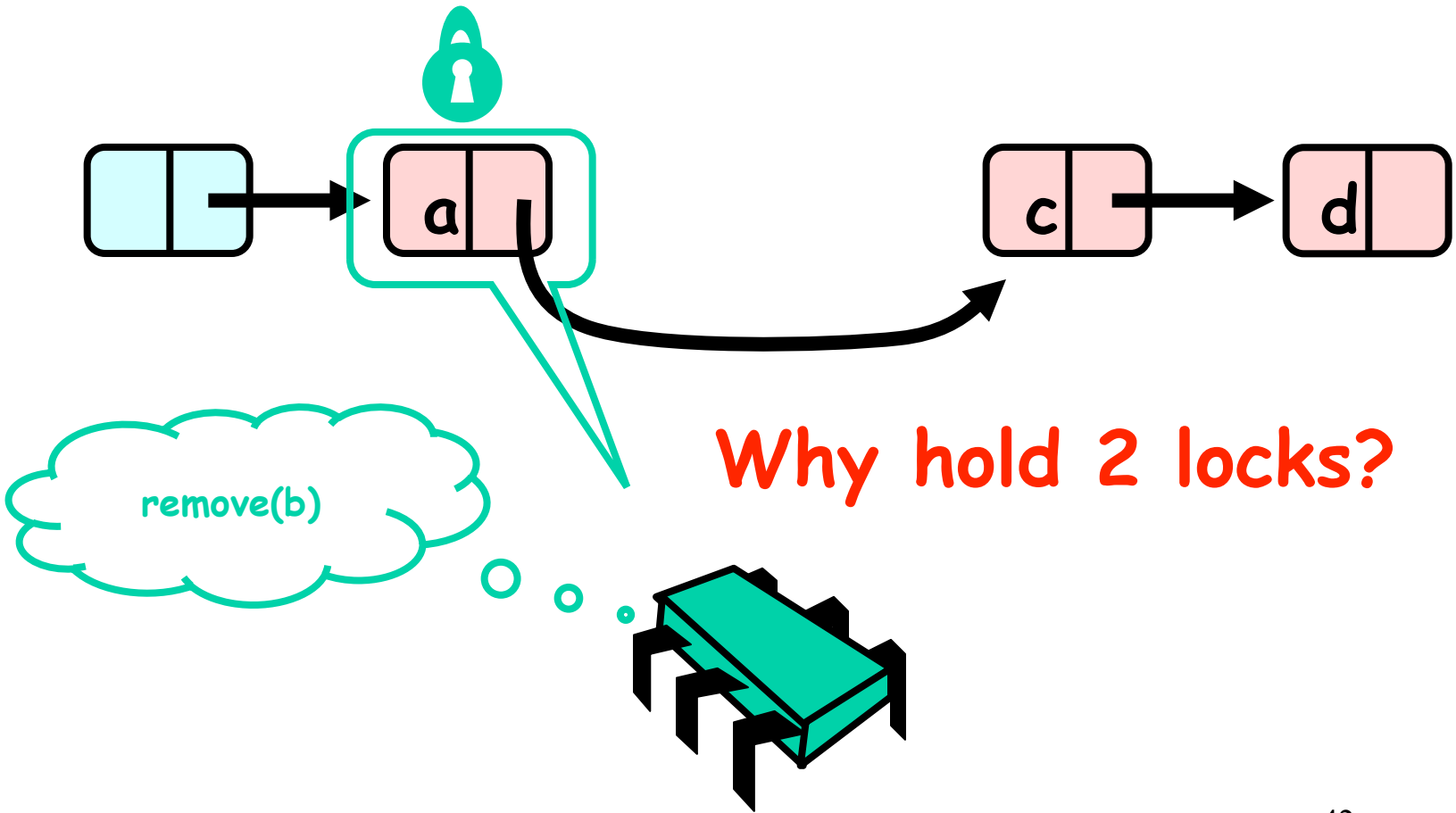
Removing a Node



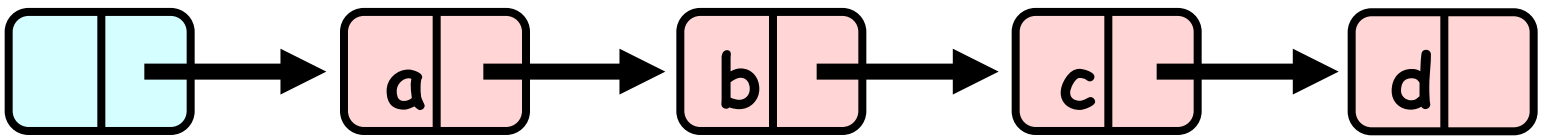
Removing a Node

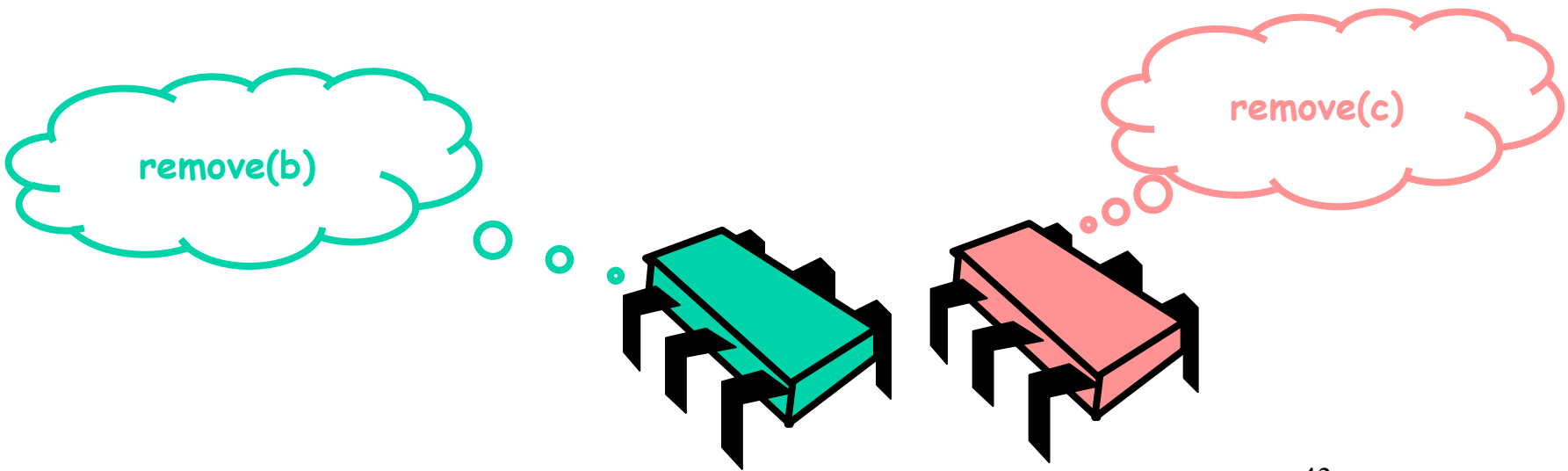


Removing a Node

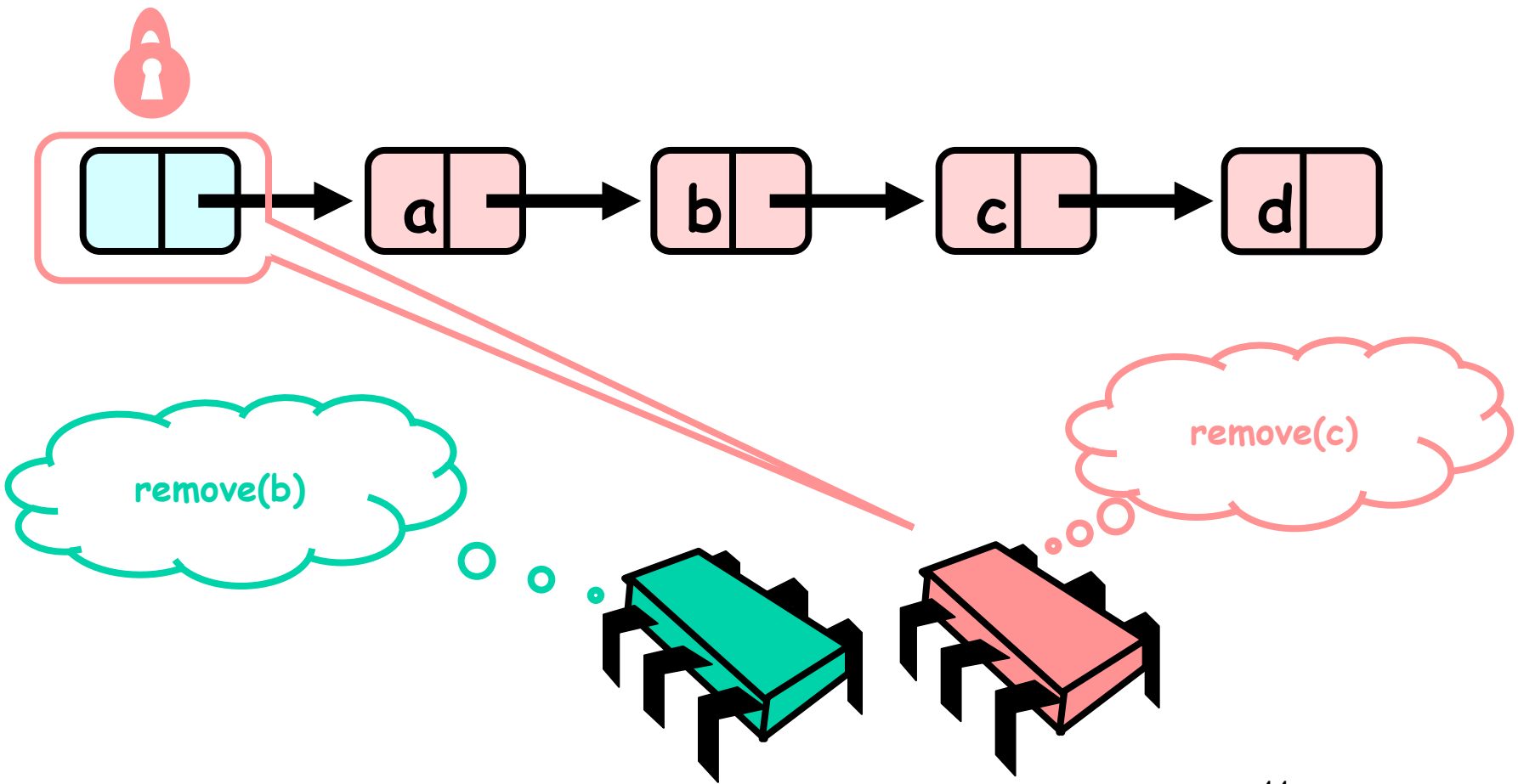


Concurrent Removes

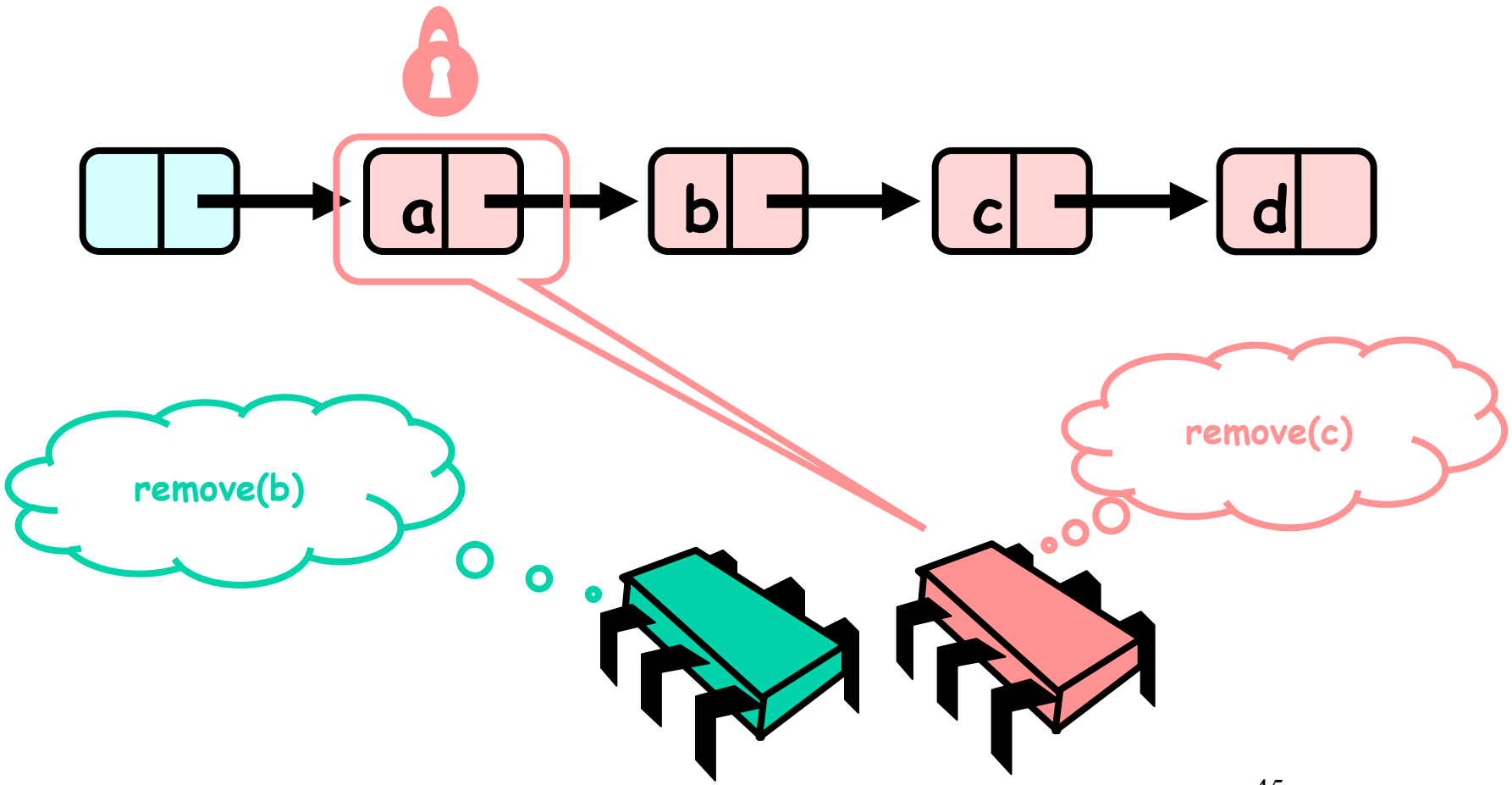




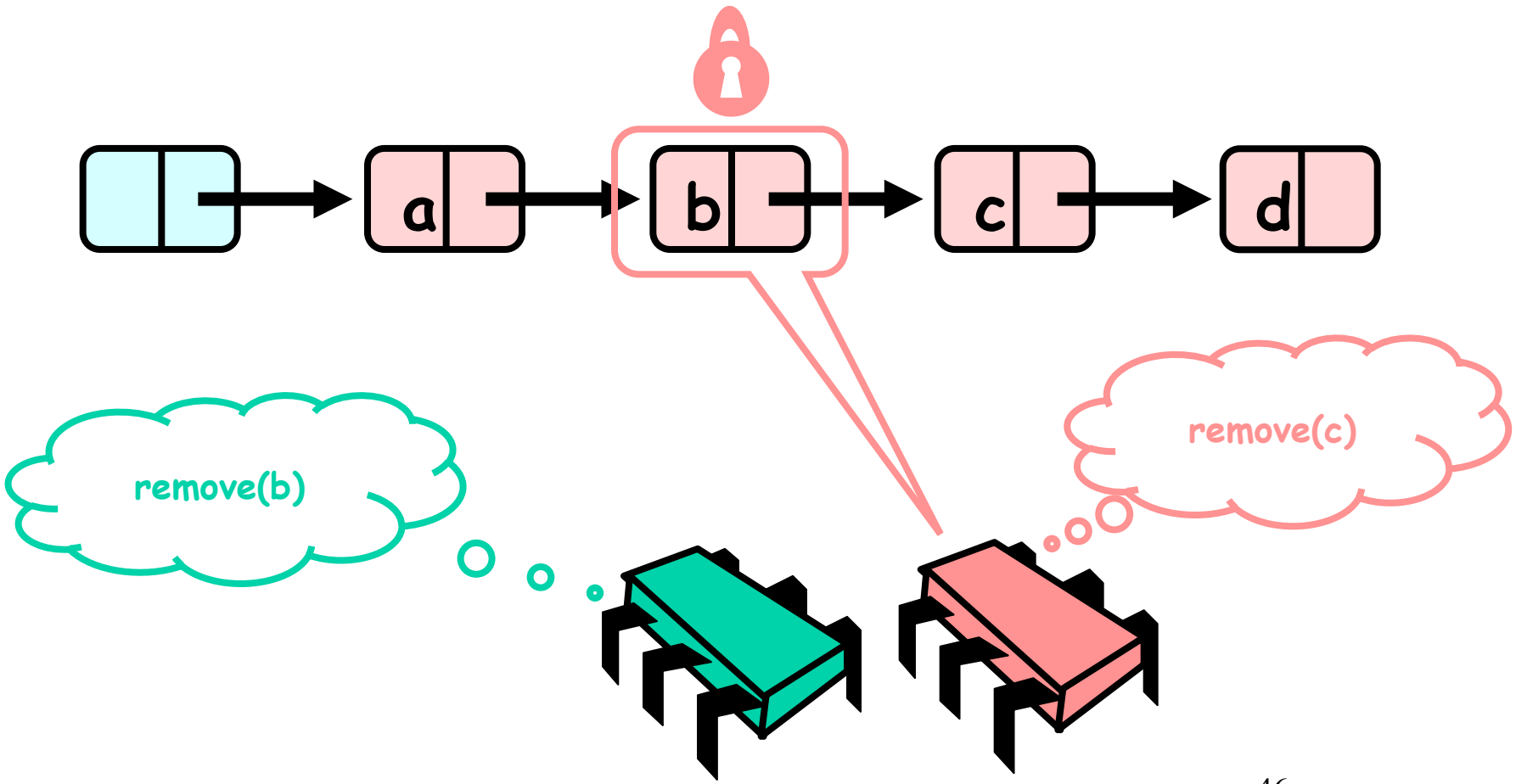
Concurrent Removes



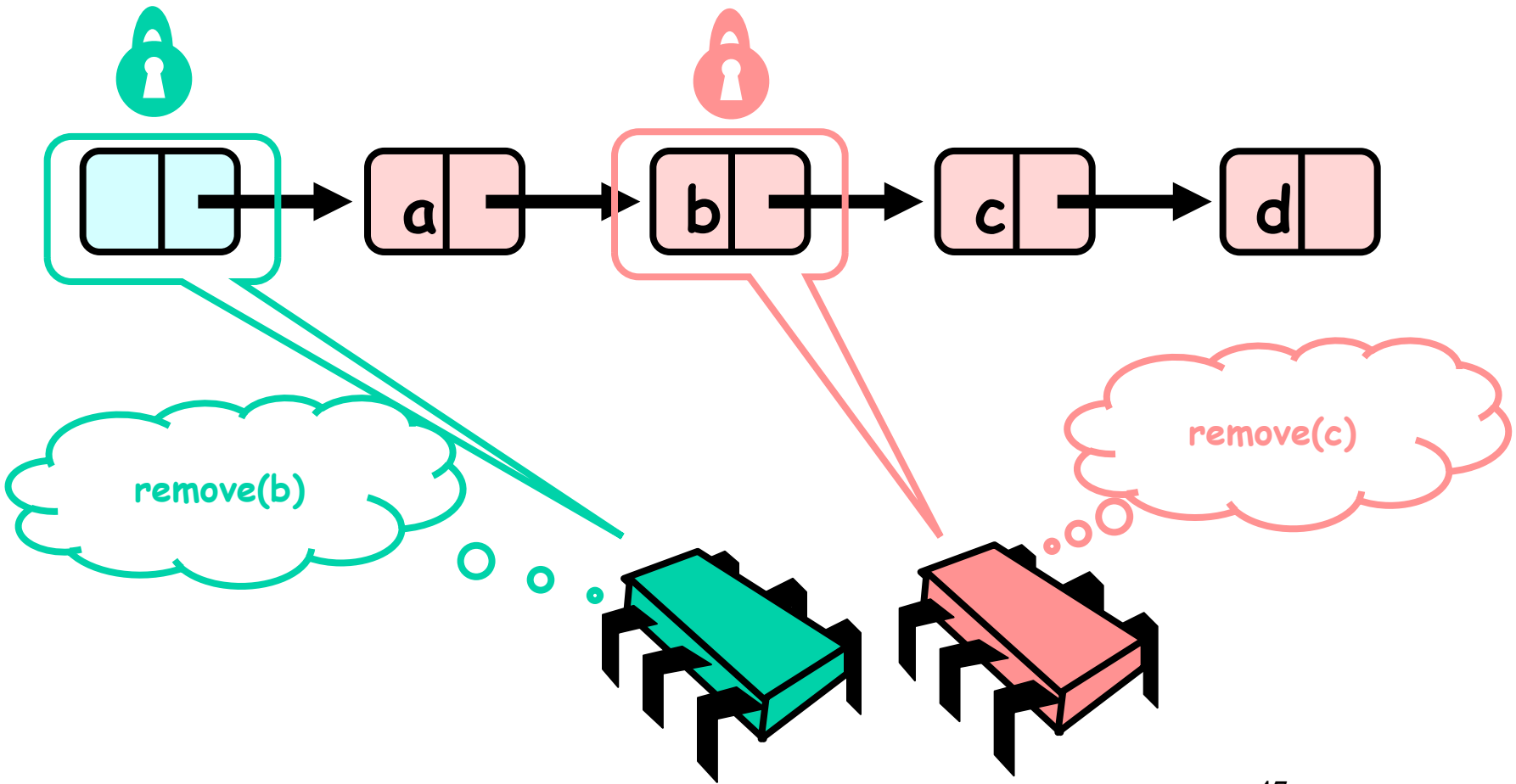
Concurrent Removes



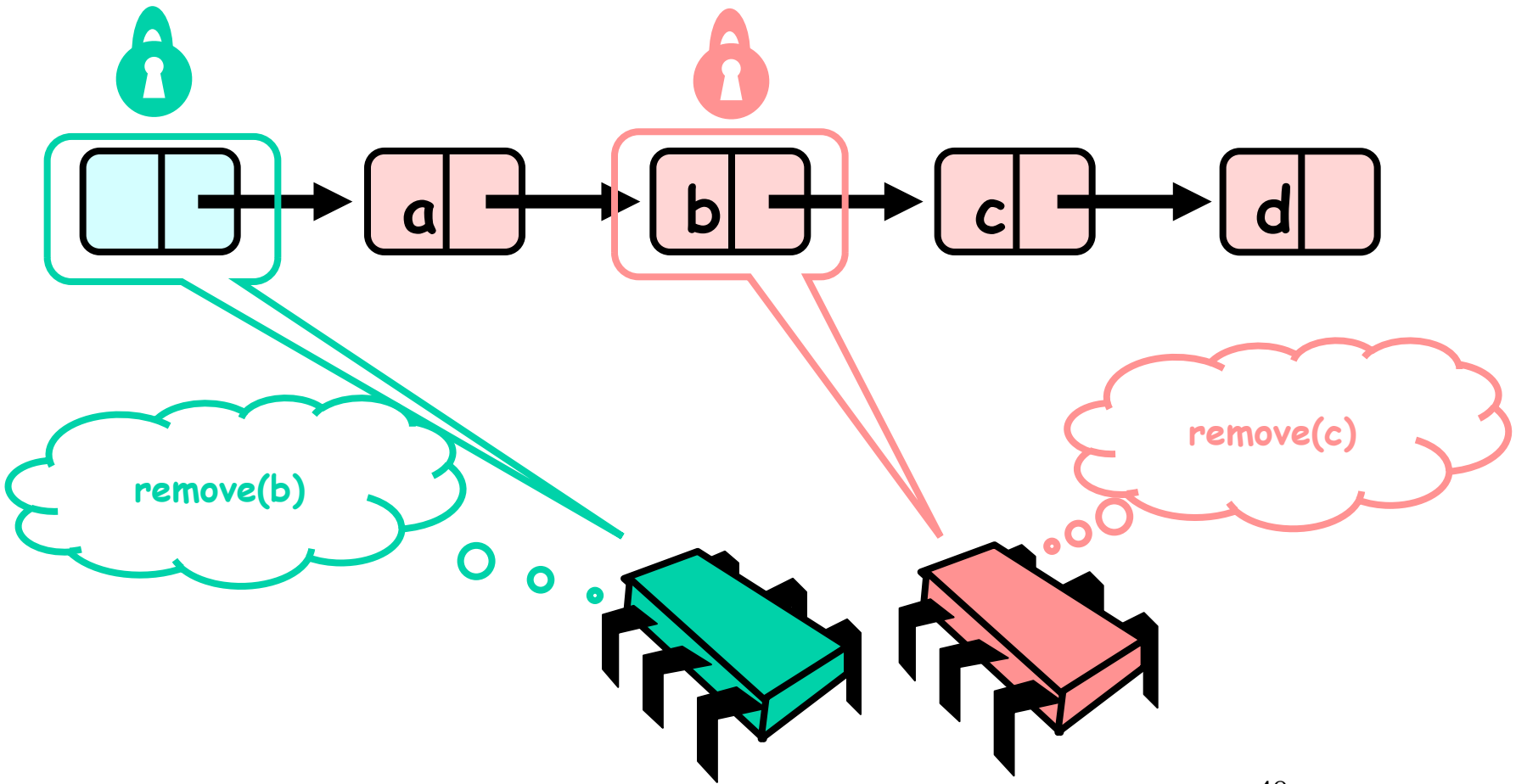
Concurrent Removes



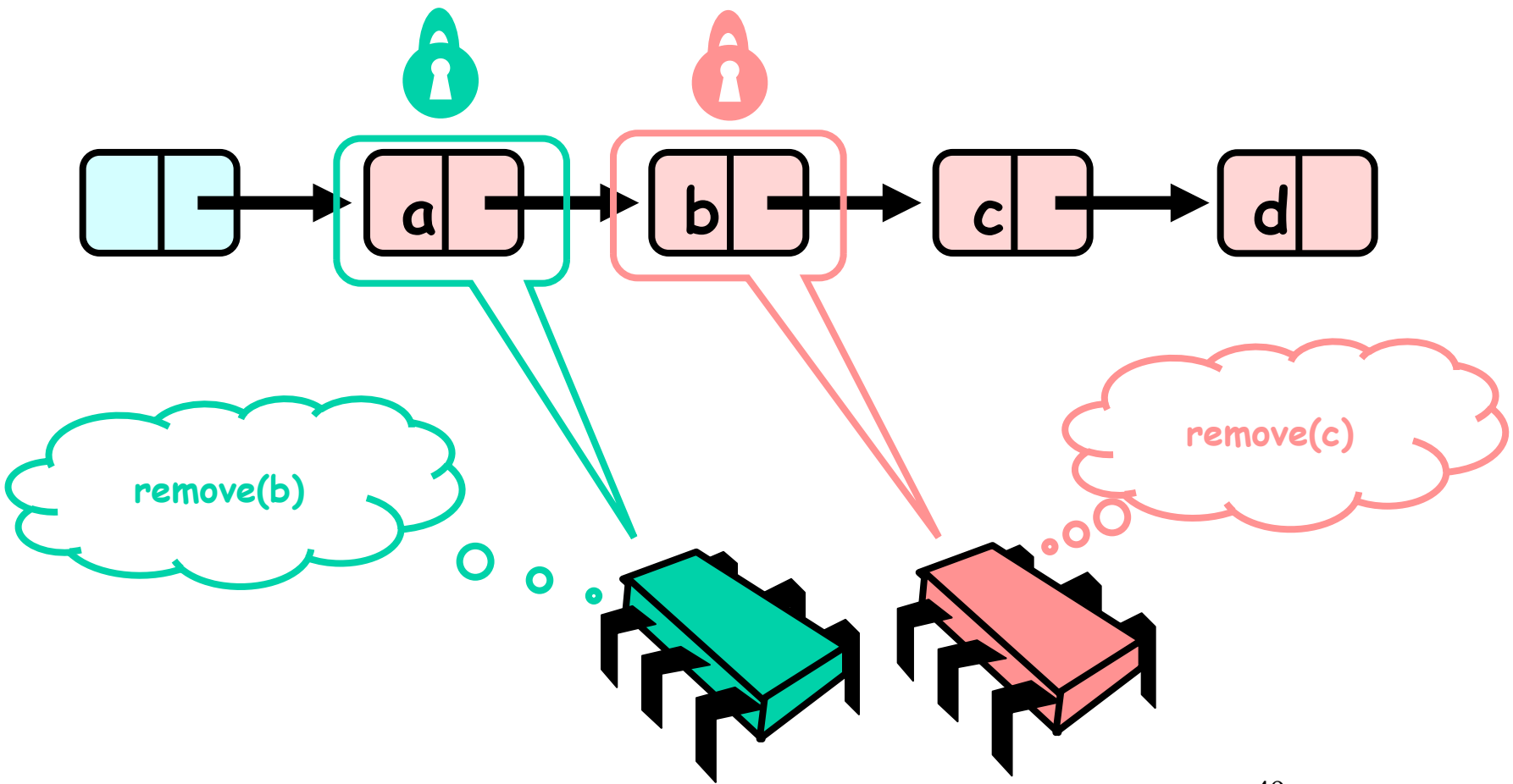
Concurrent Removes



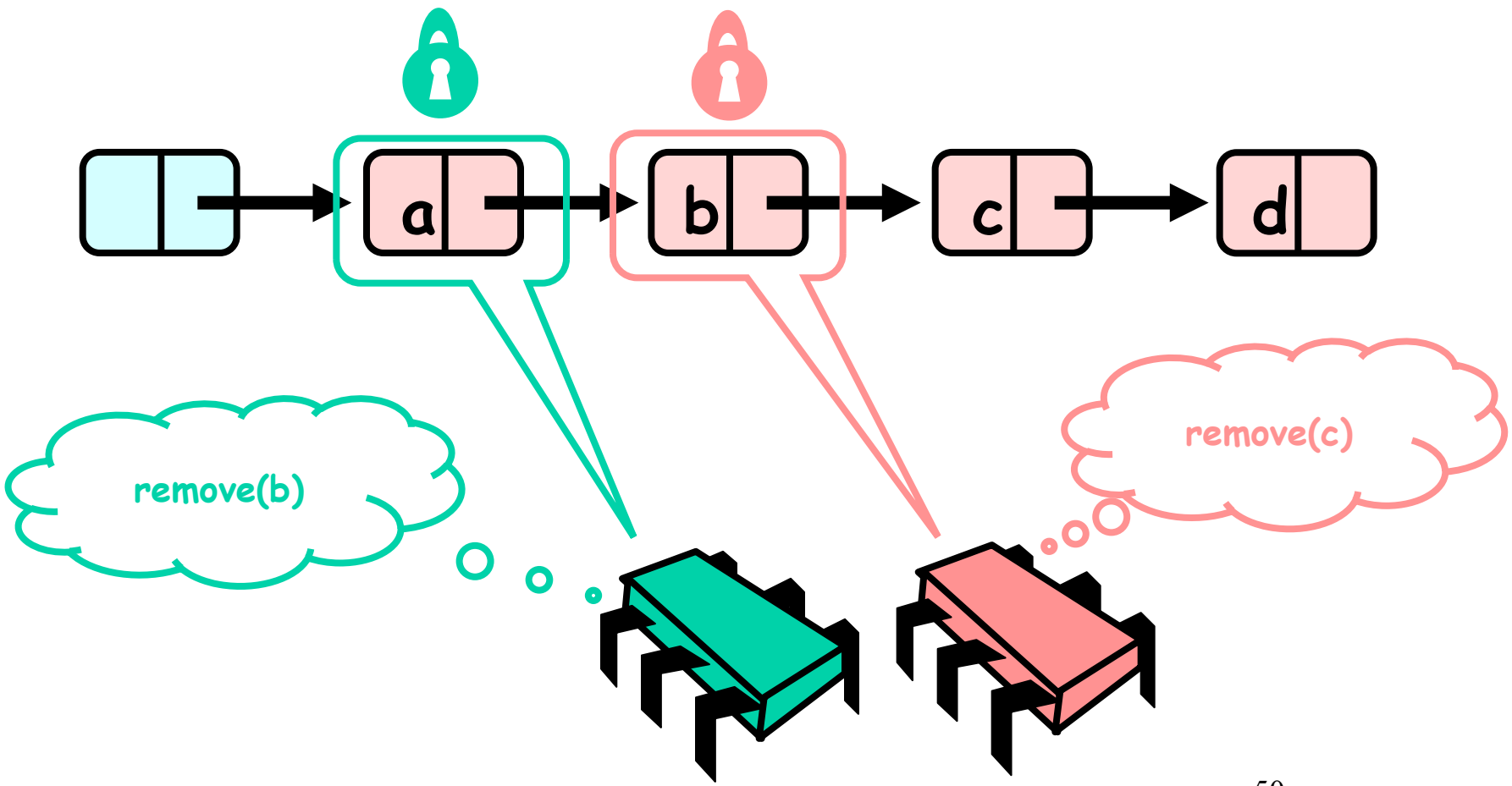
Concurrent Removes



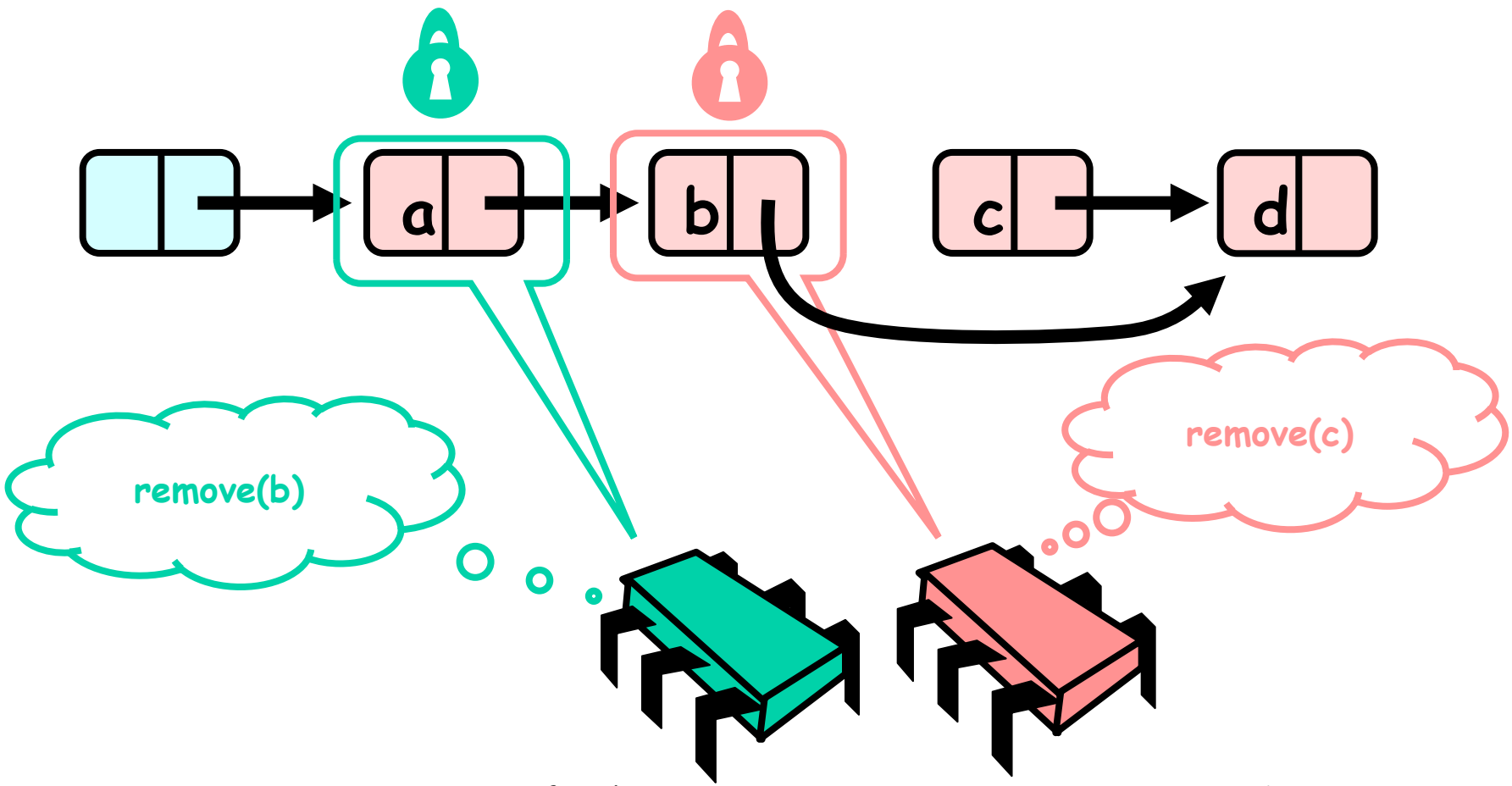
Concurrent Removes



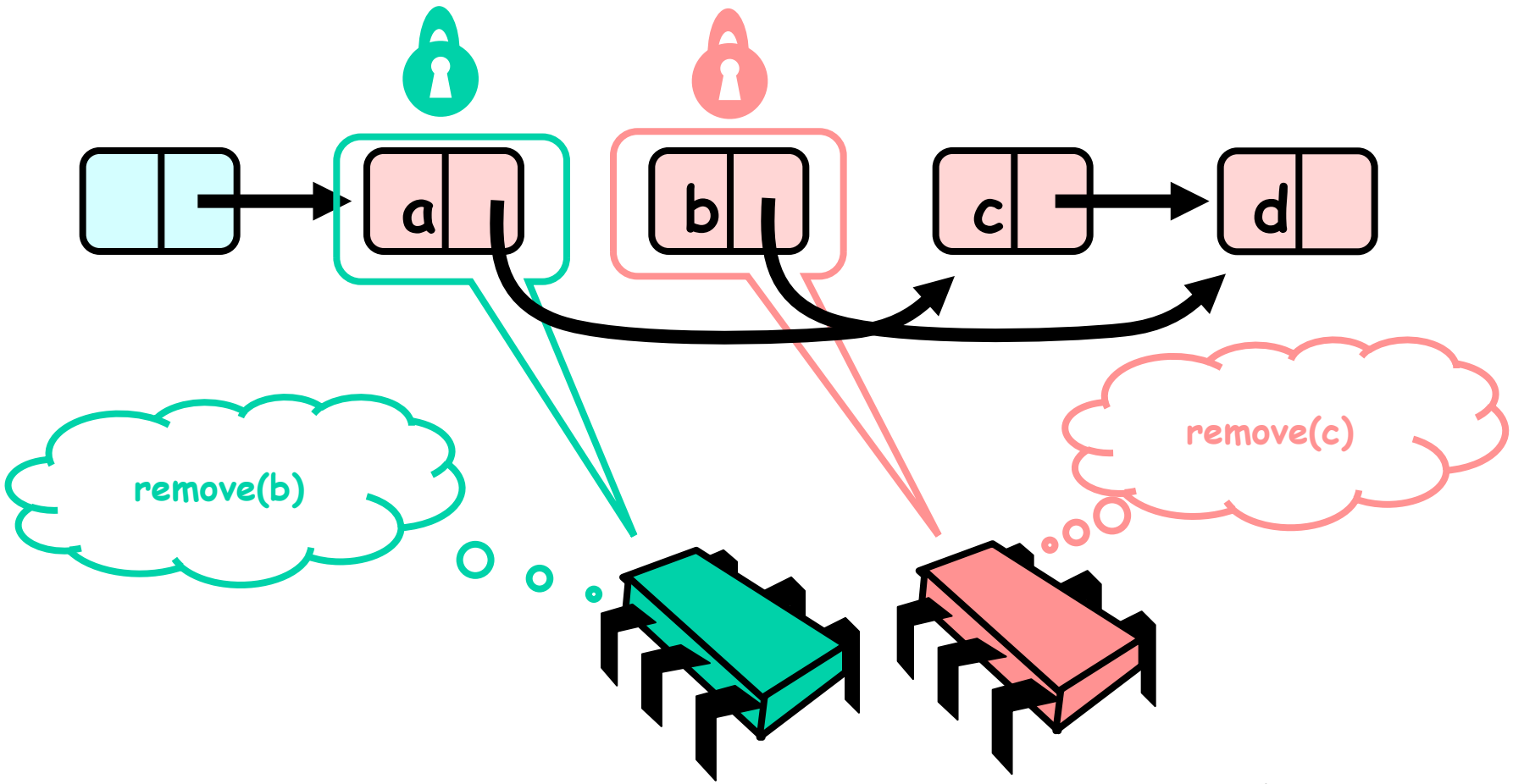
Concurrent Removes



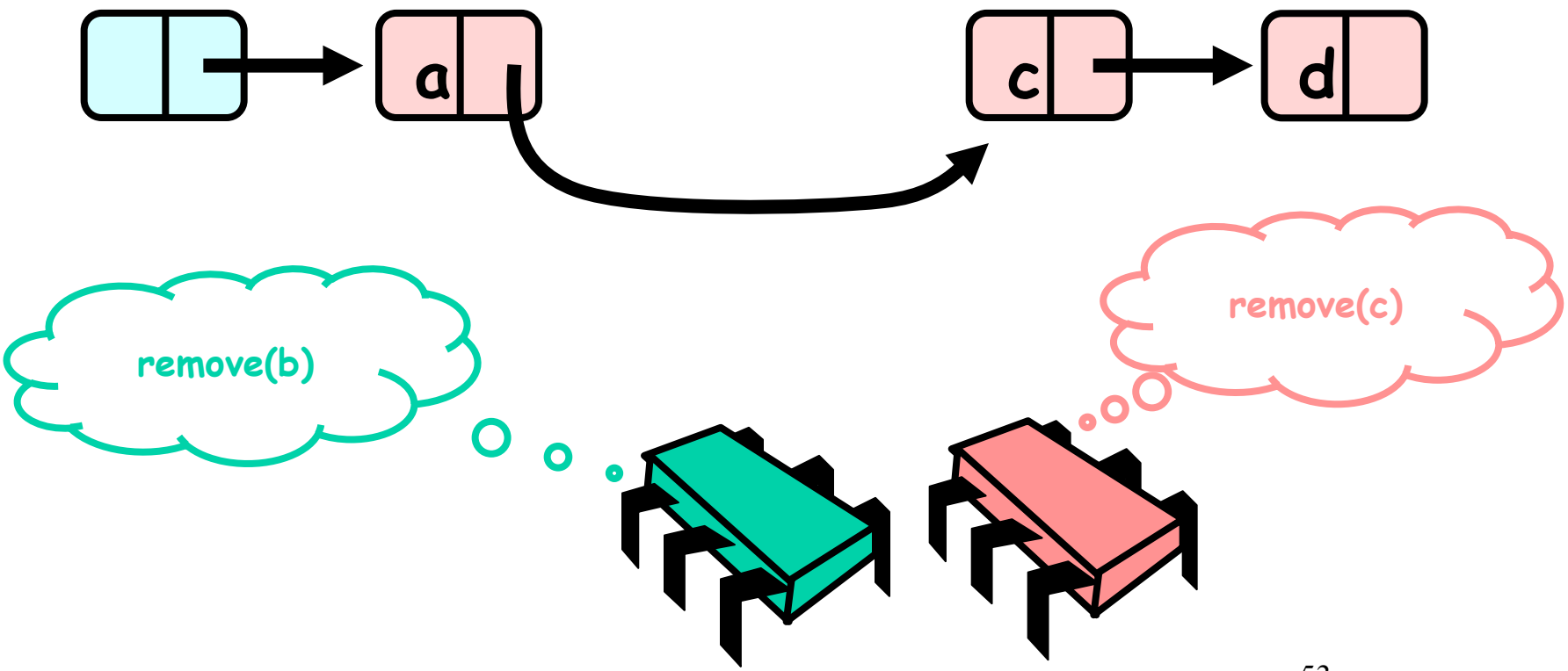
Concurrent Removes



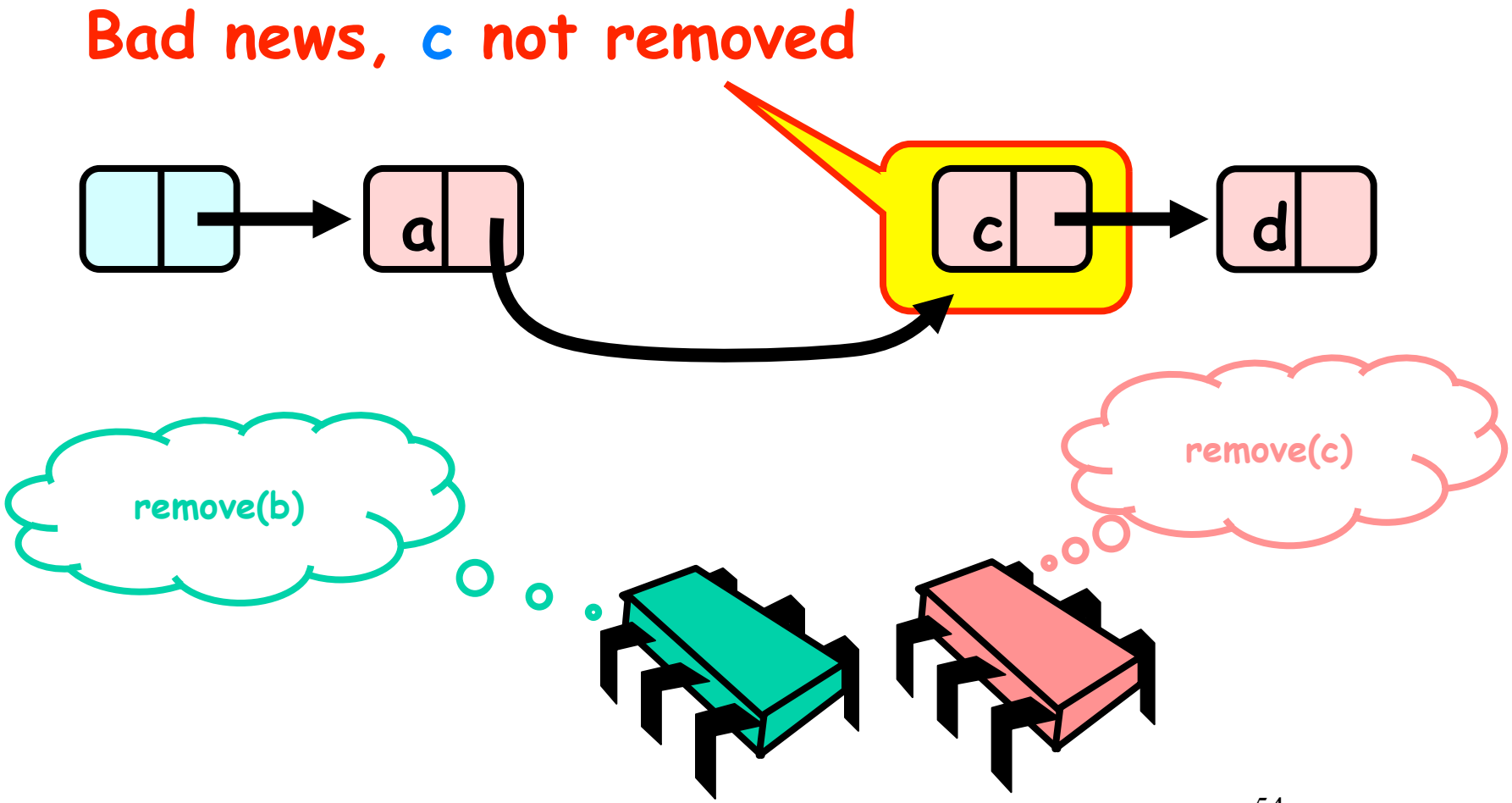
Concurrent Removes



Uh, Oh

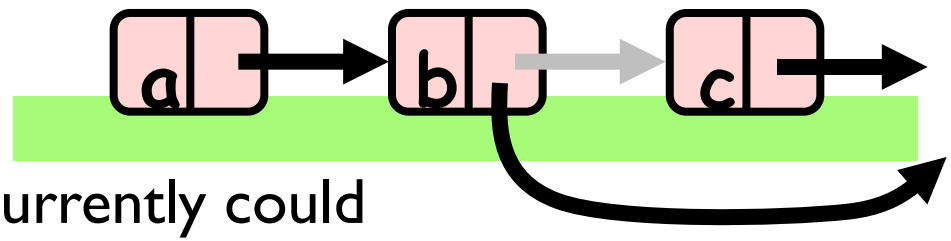


Uh, Oh

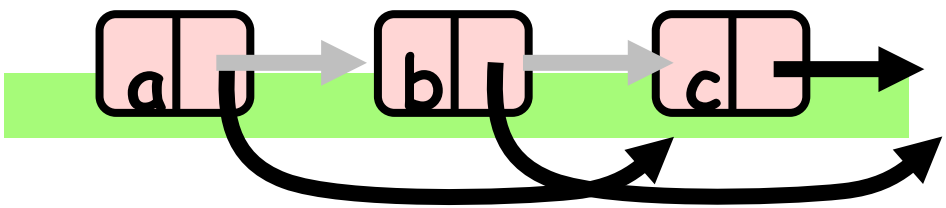


Problem

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



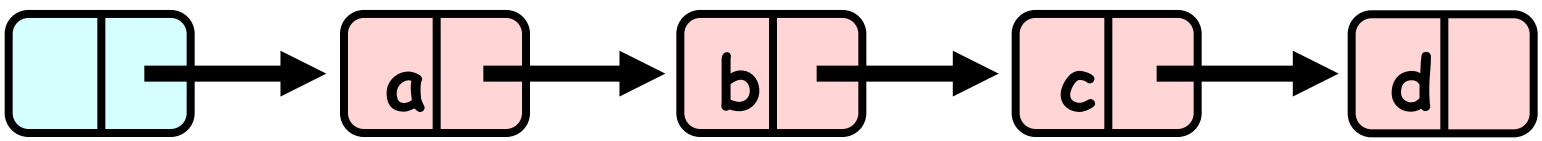
- Problem is,
  - Someone deleting b concurrently could direct a pointer to C

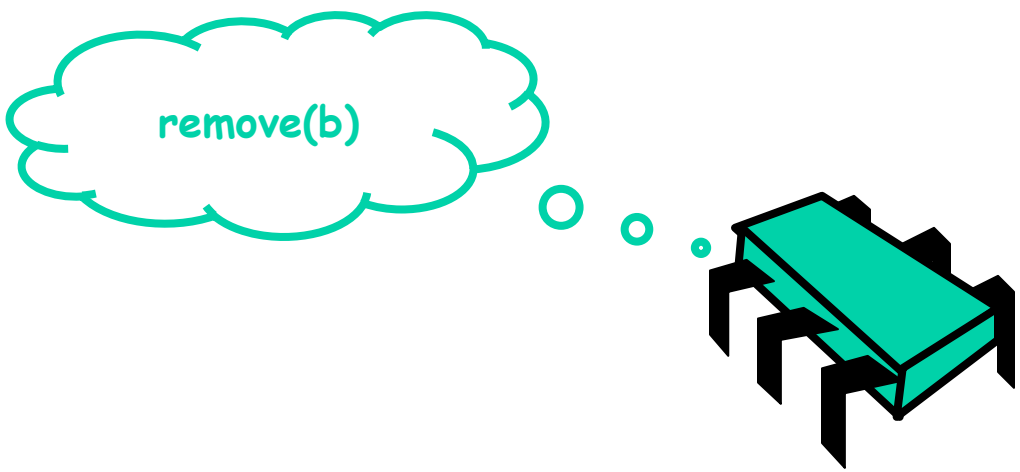


Insight

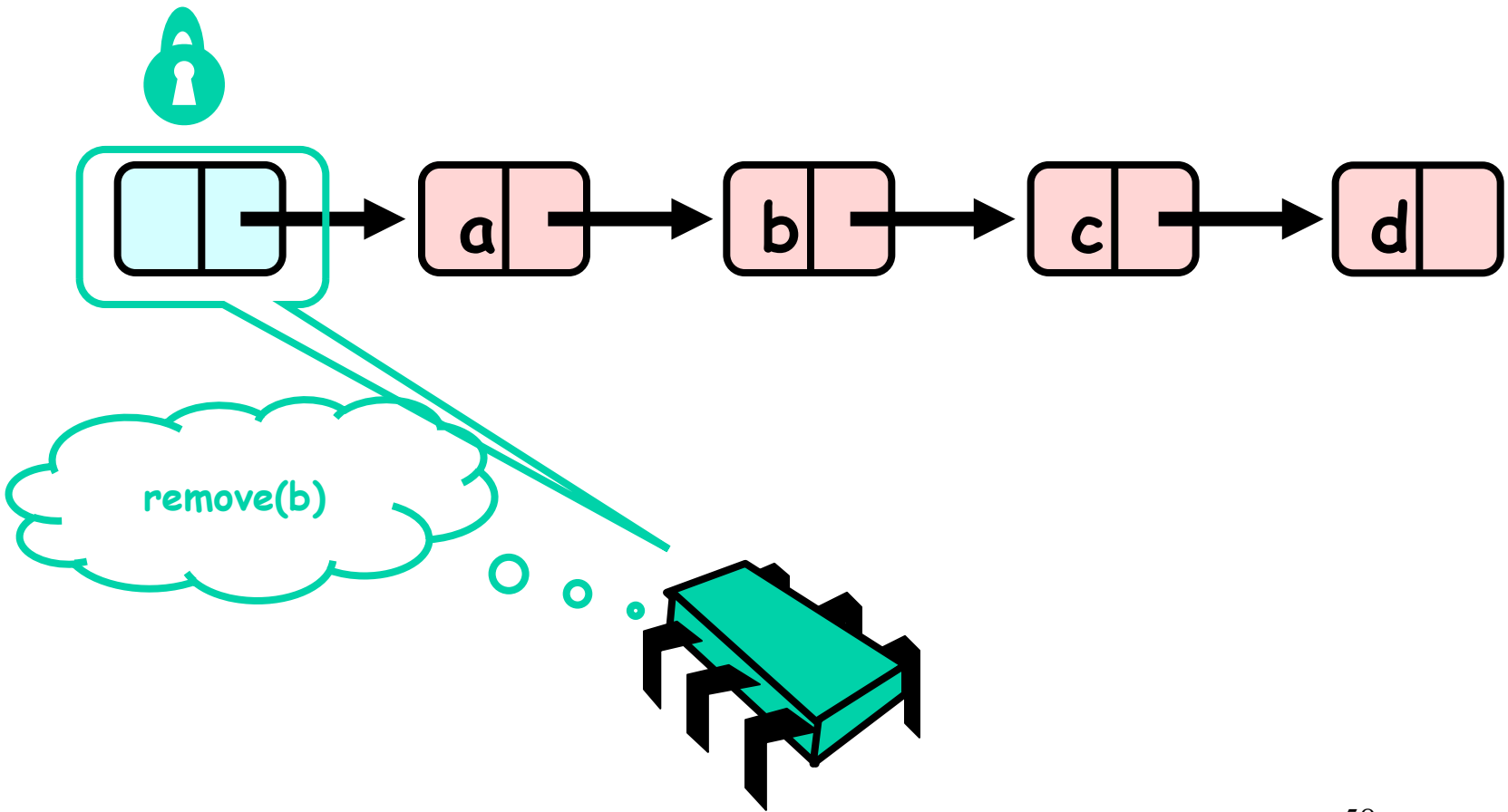
- If a node is locked
  - No one can delete node's successor
- If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works

Hand-Over-Hand Again

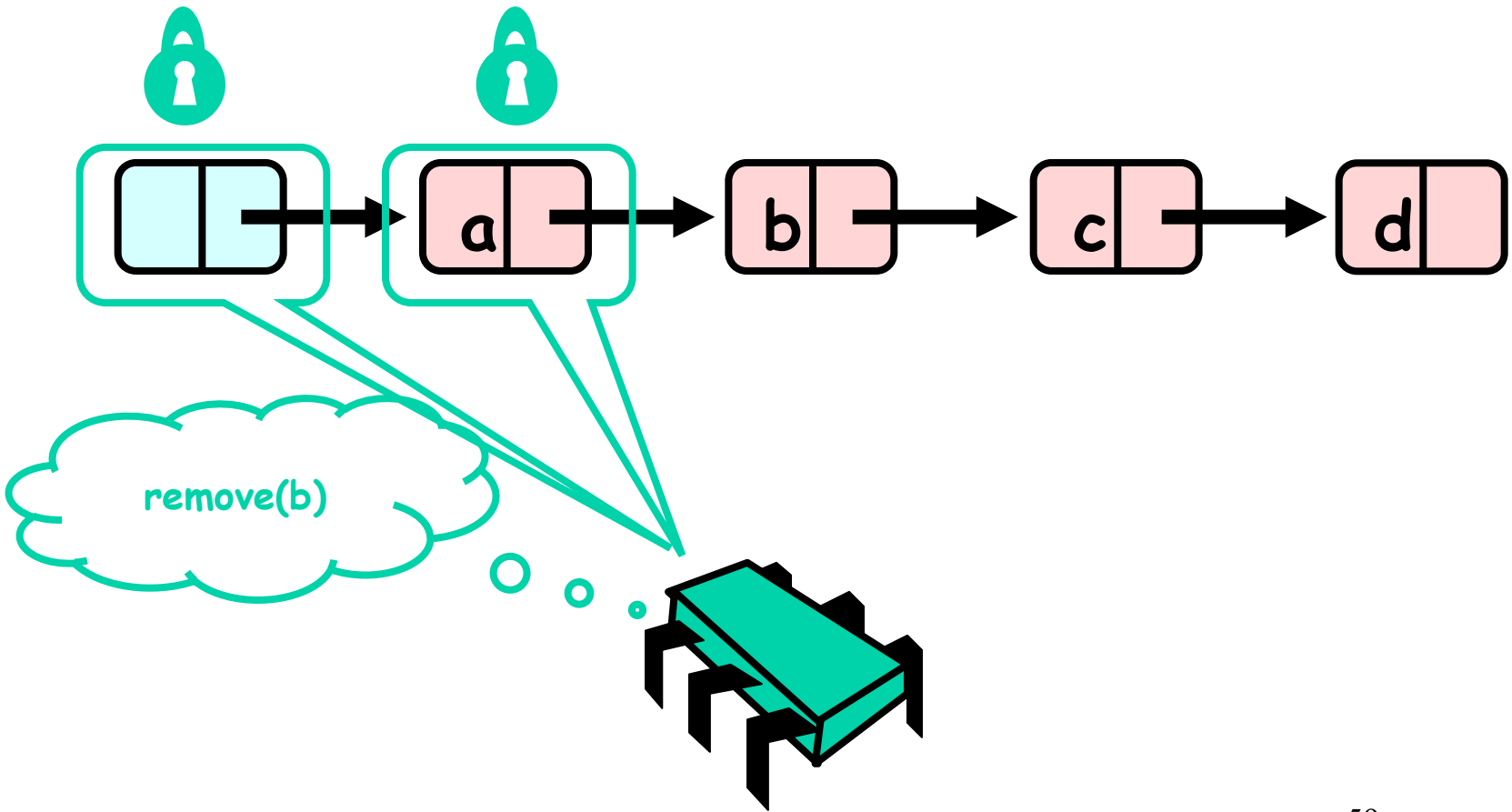




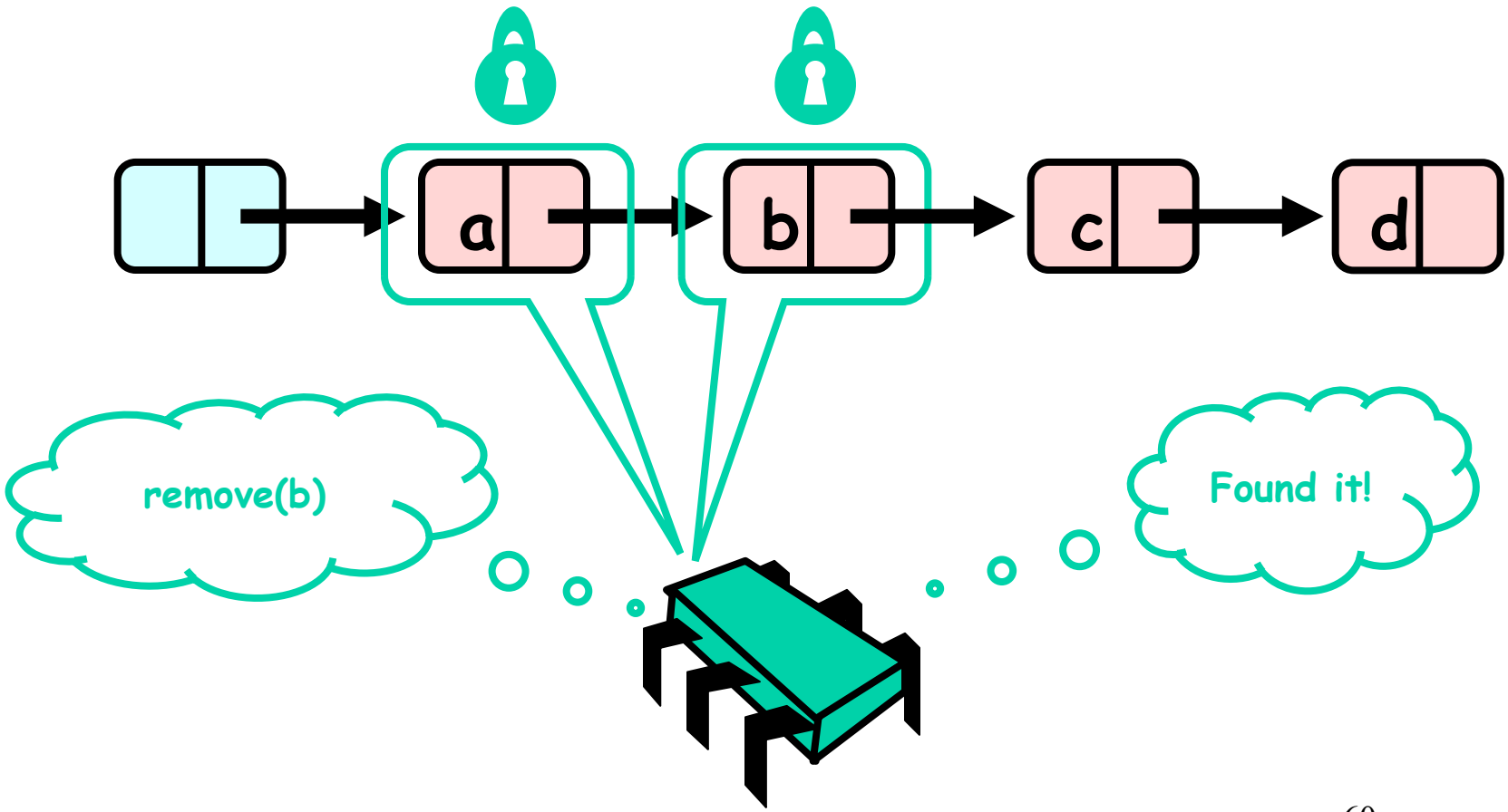
Hand-Over-Hand Again



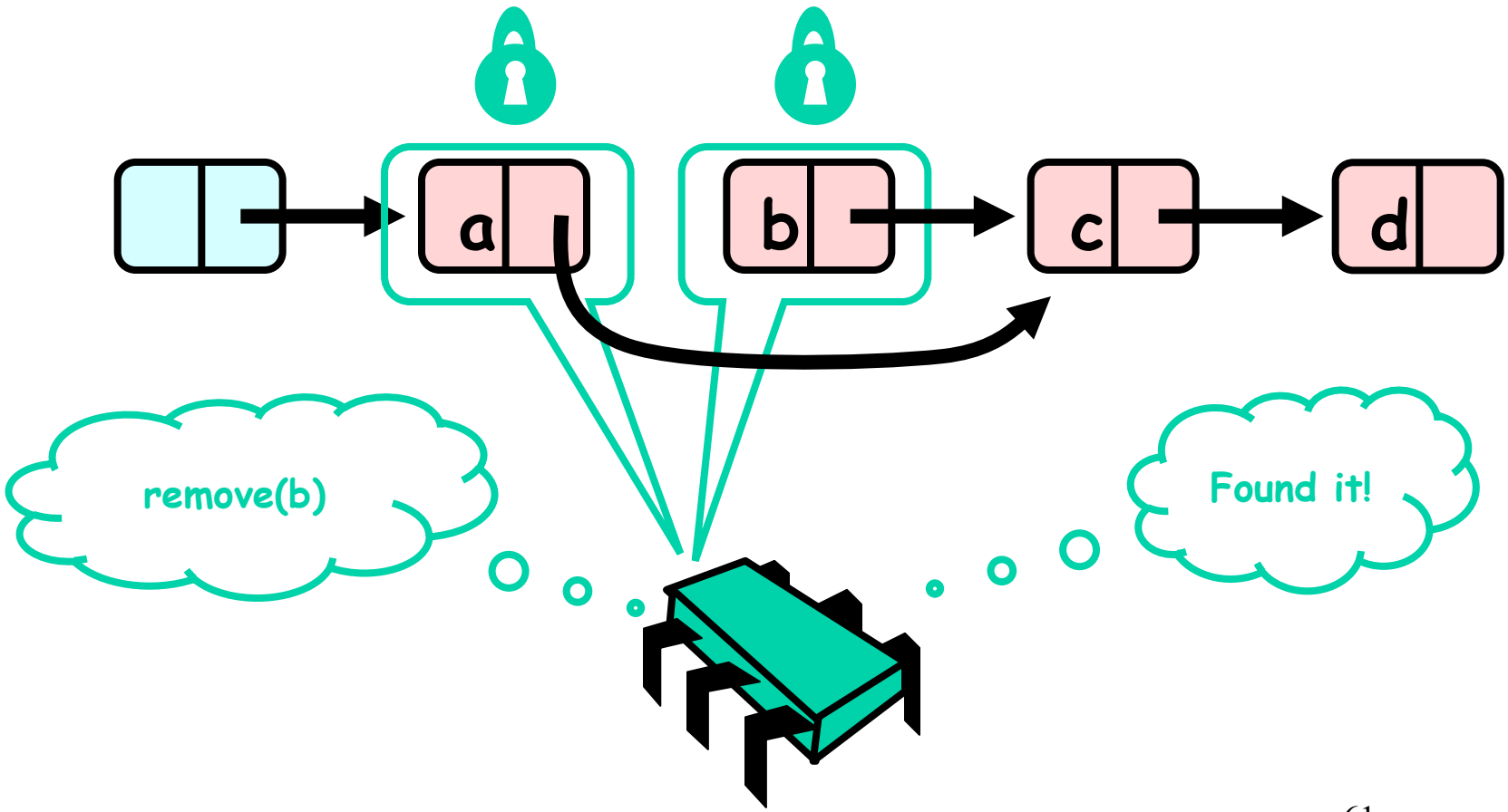
Hand-Over-Hand Again



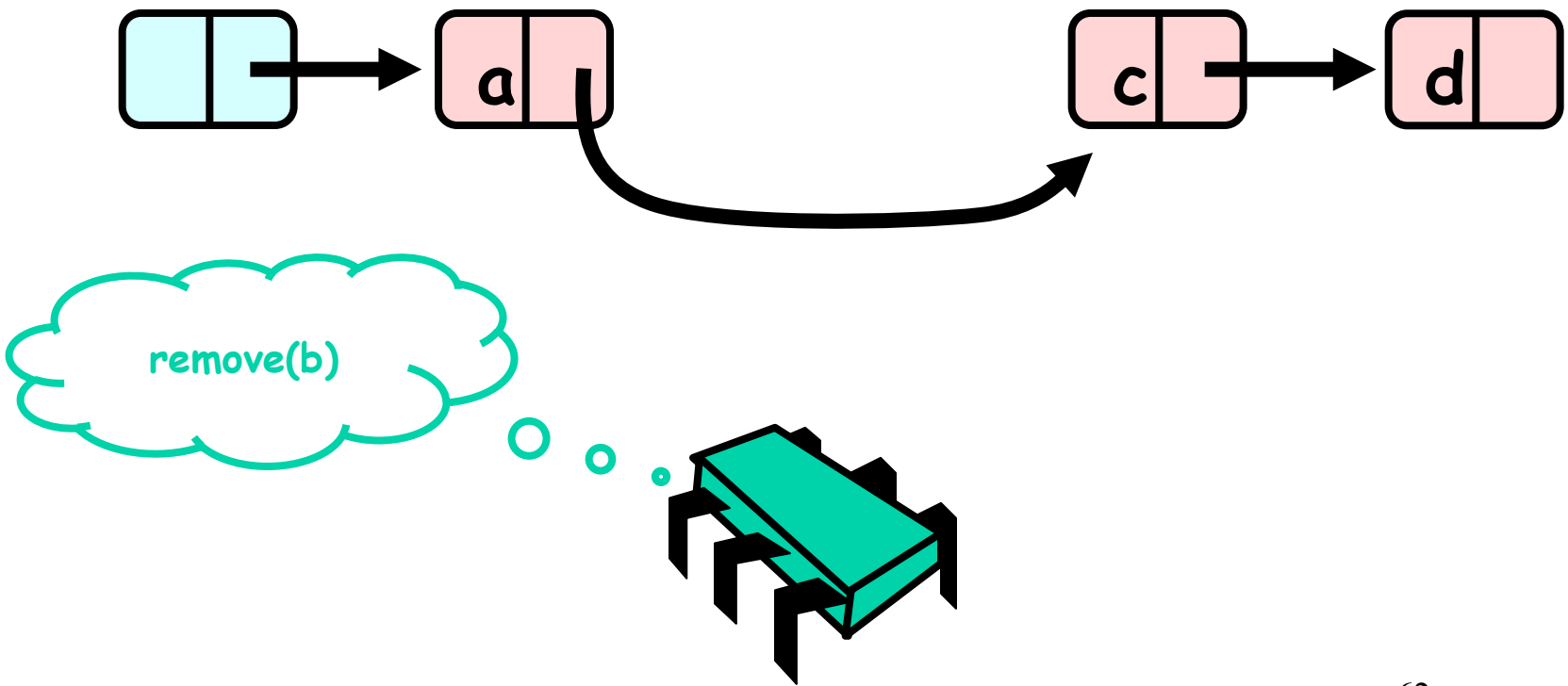
Hand-Over-Hand Again



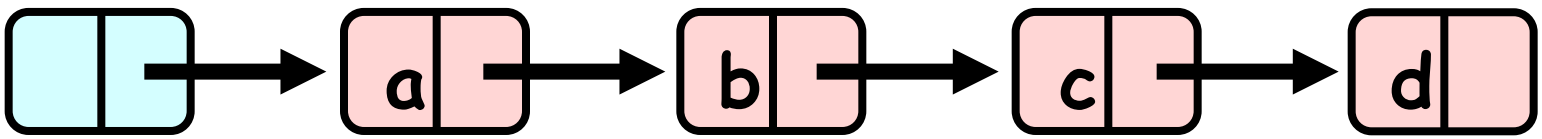
Hand-Over-Hand Again

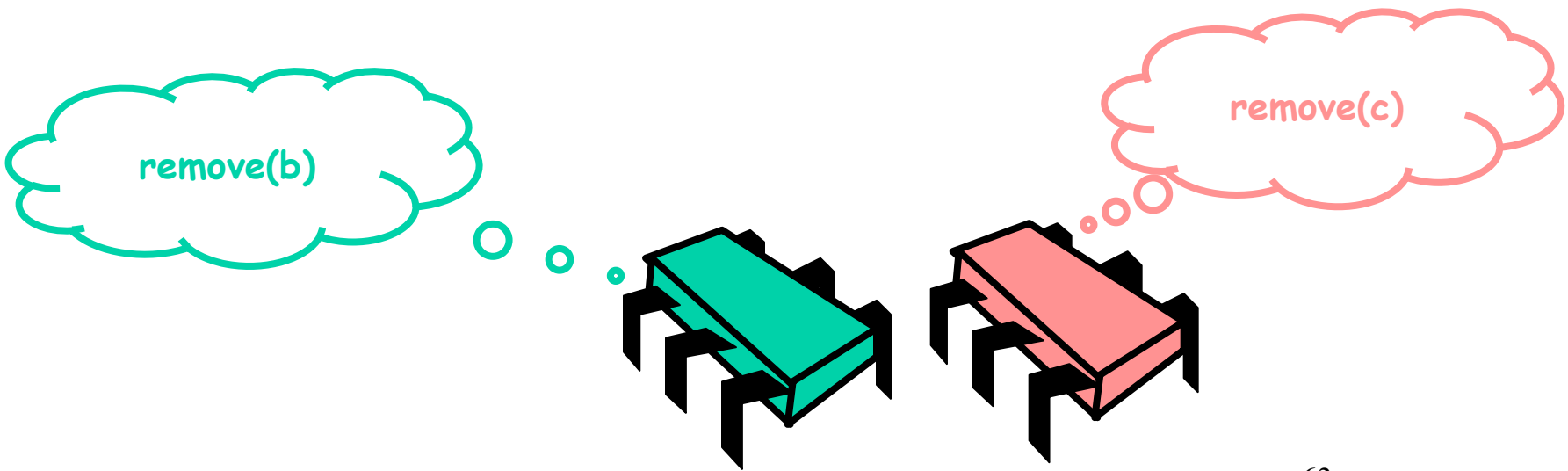


Hand-Over-Hand Again

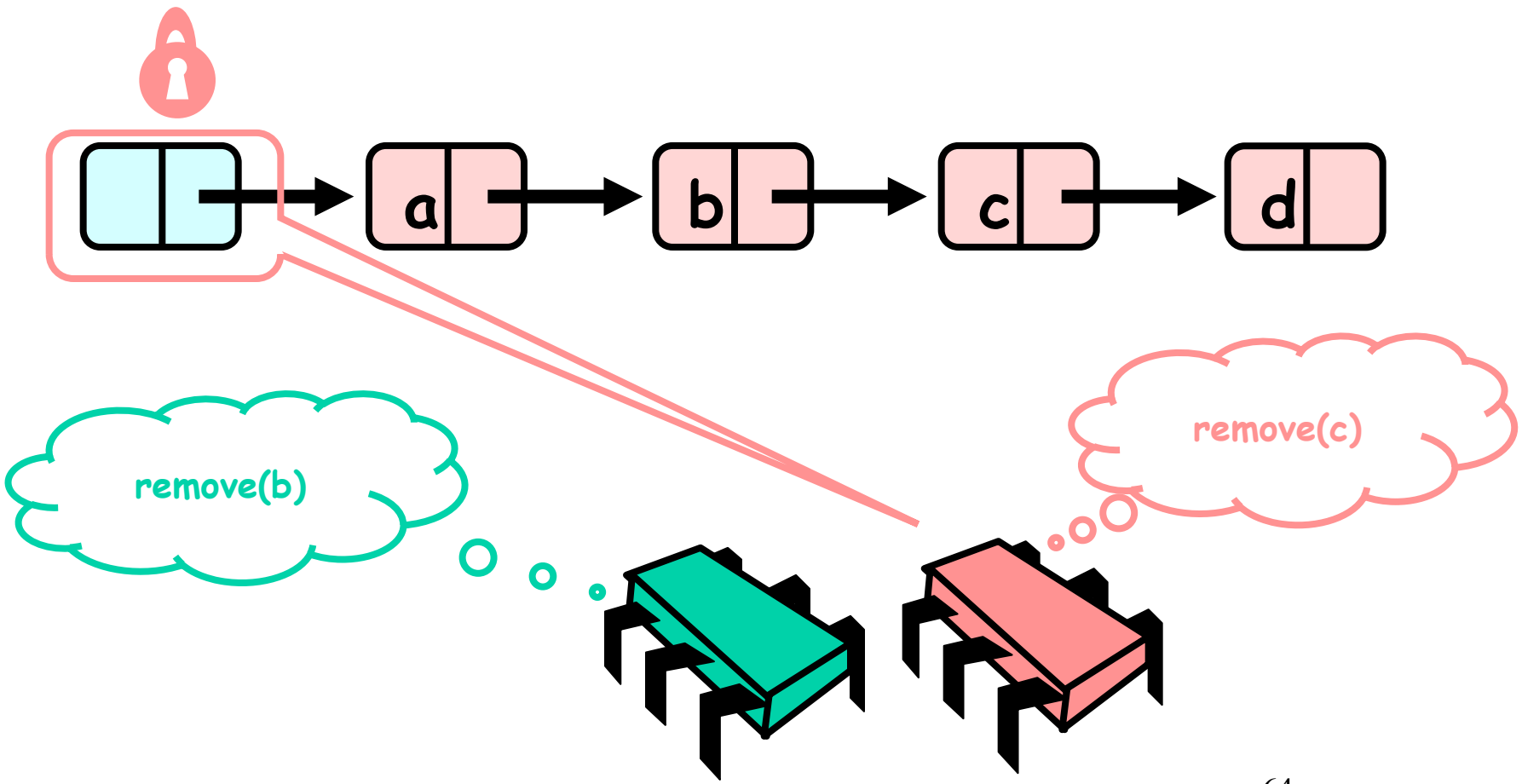


Removing a Node

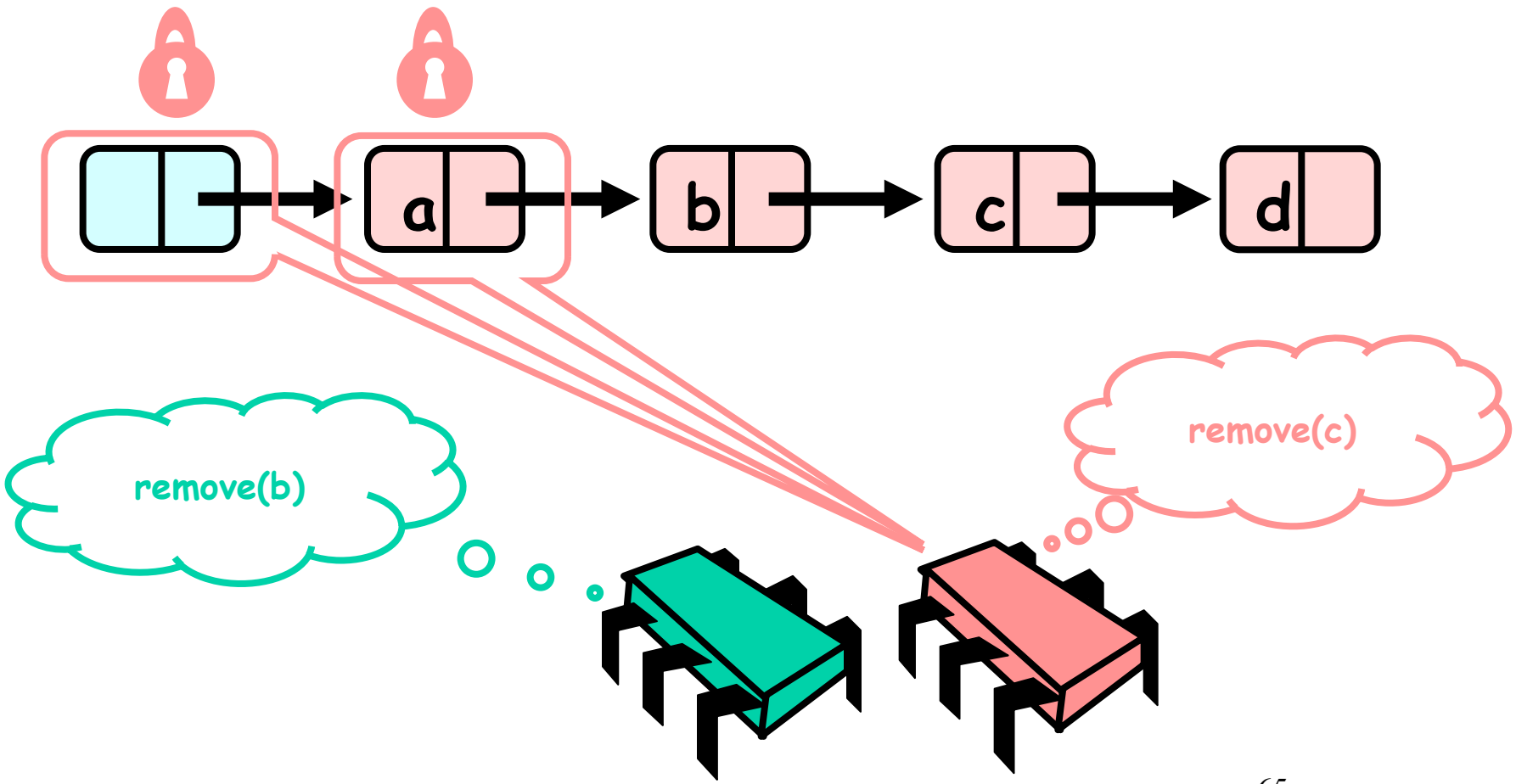




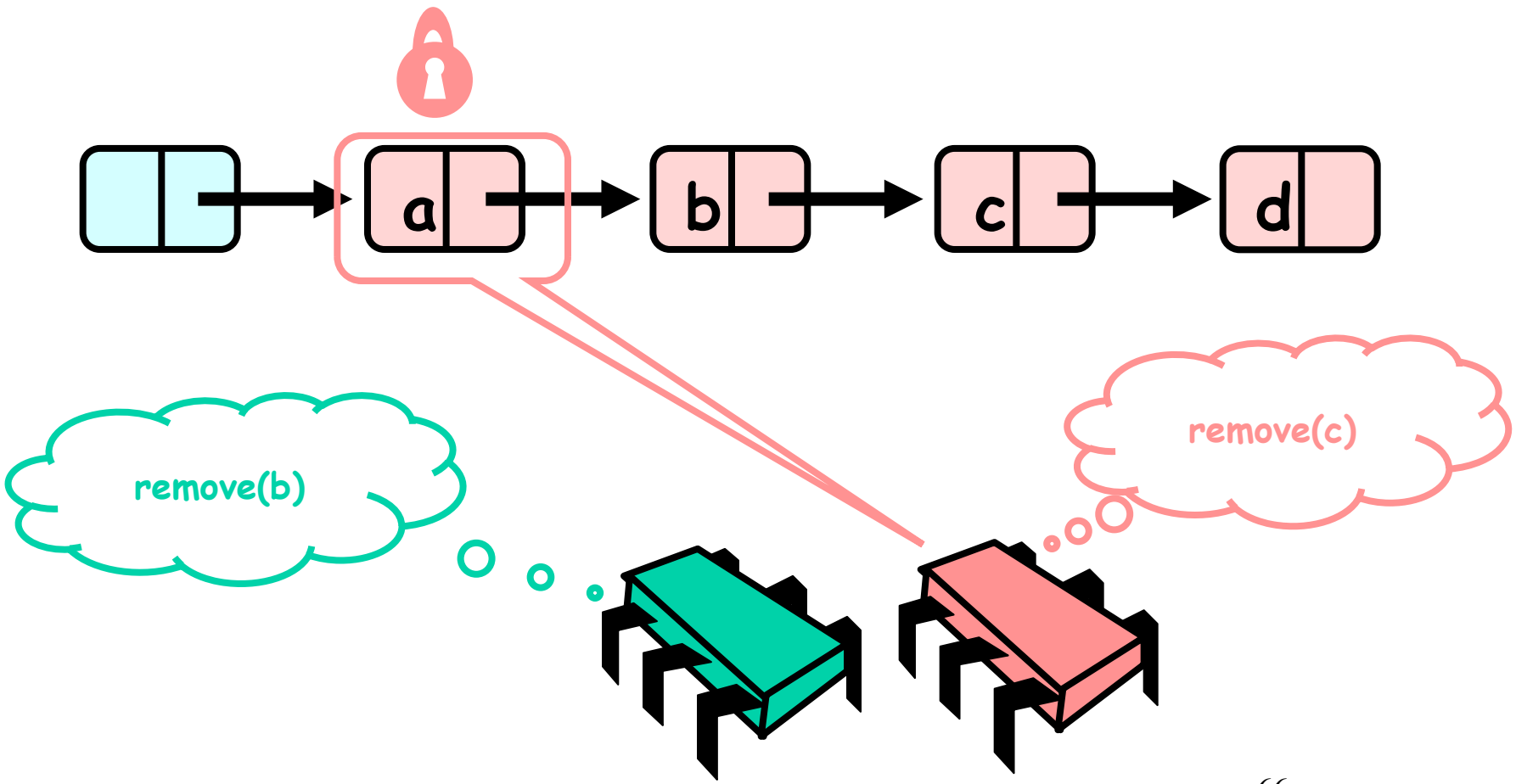
Removing a Node



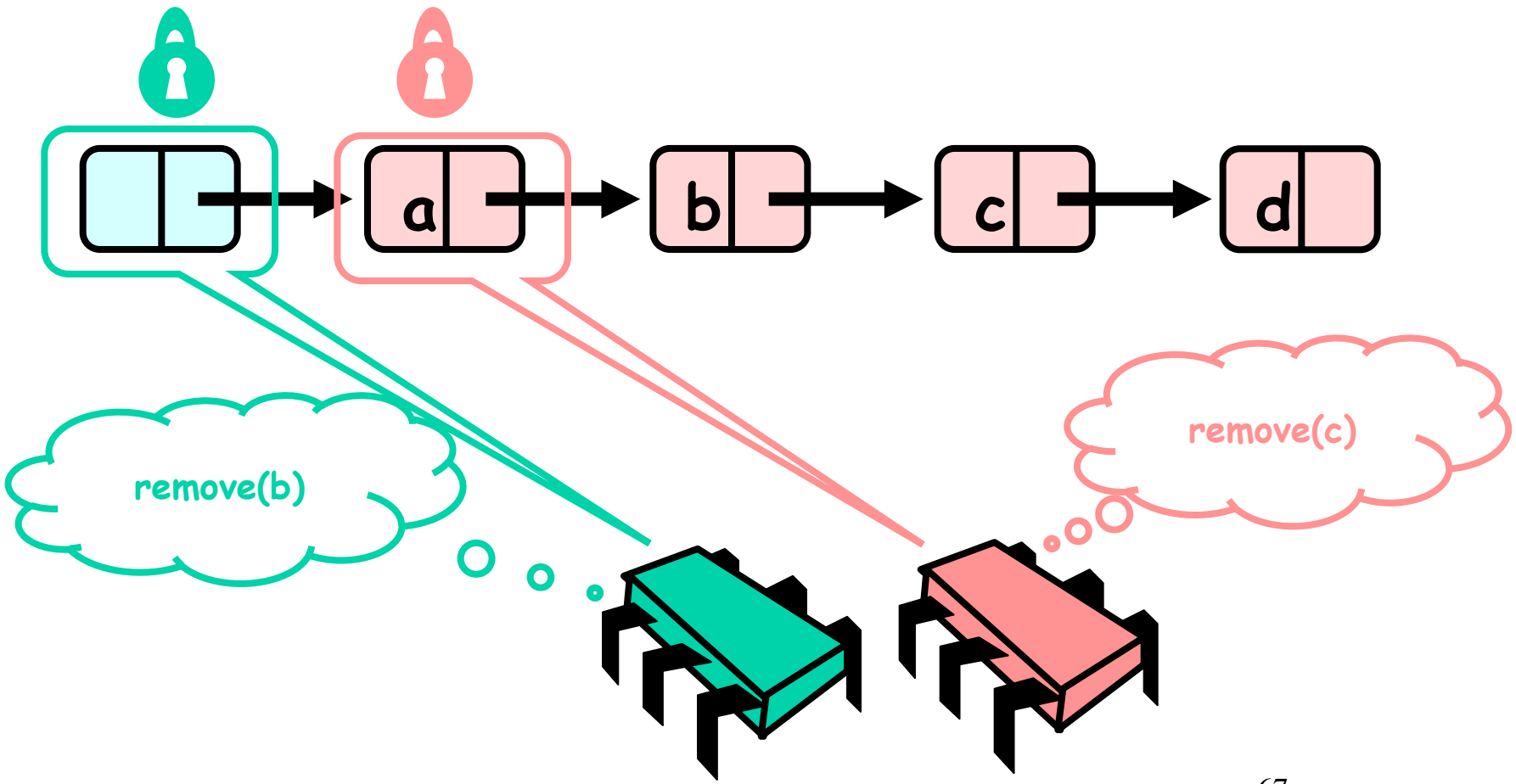
Removing a Node



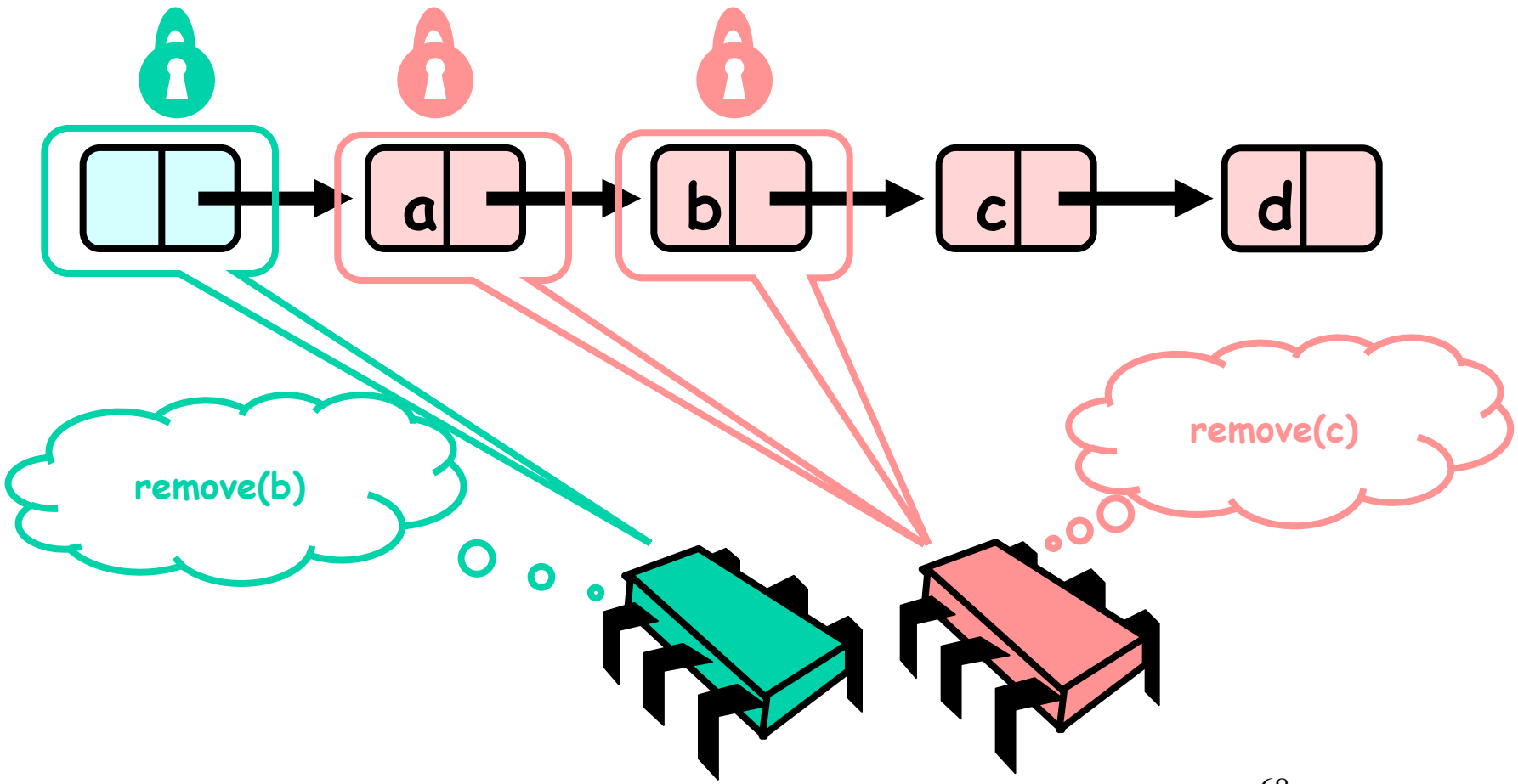
Removing a Node



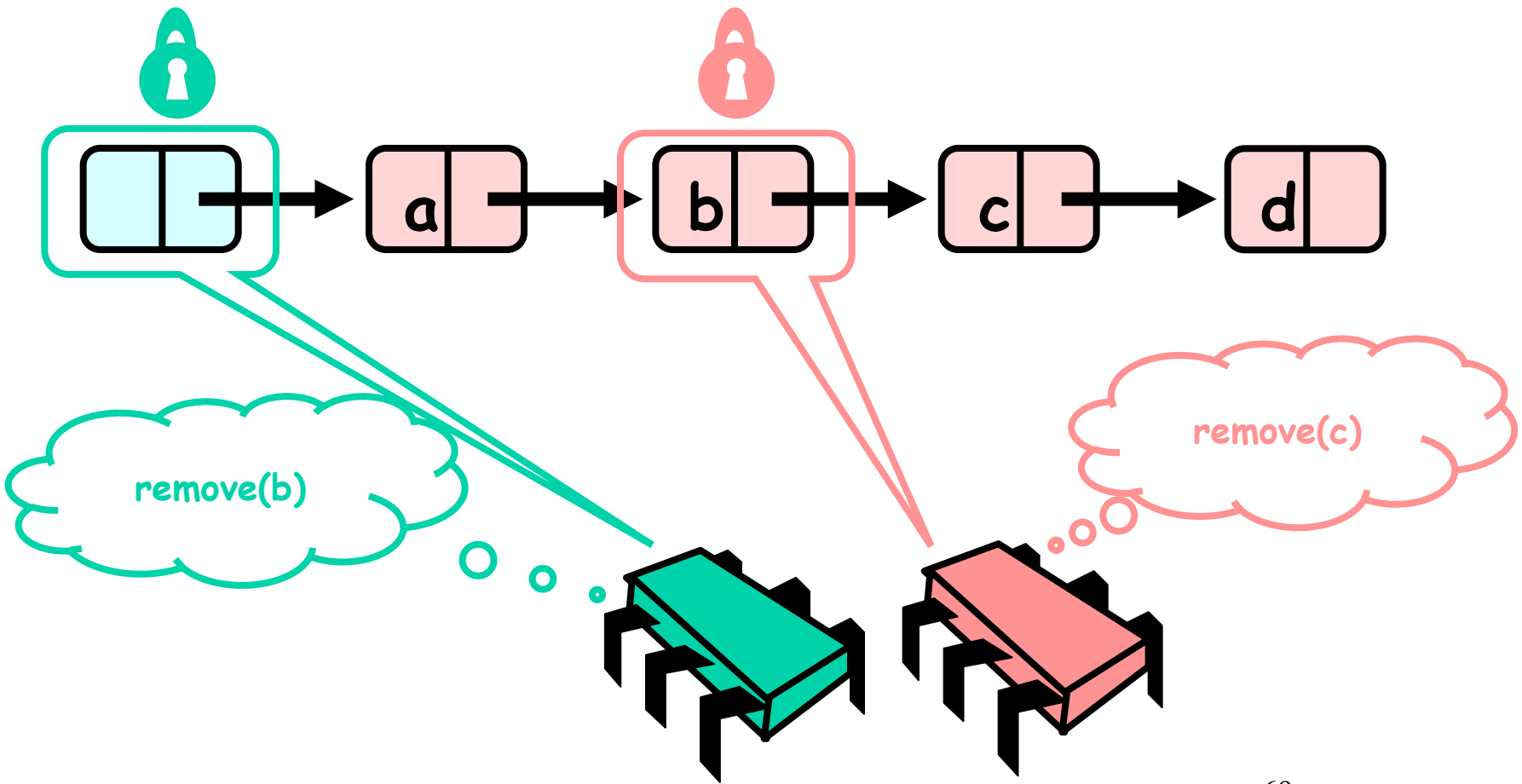
Removing a Node



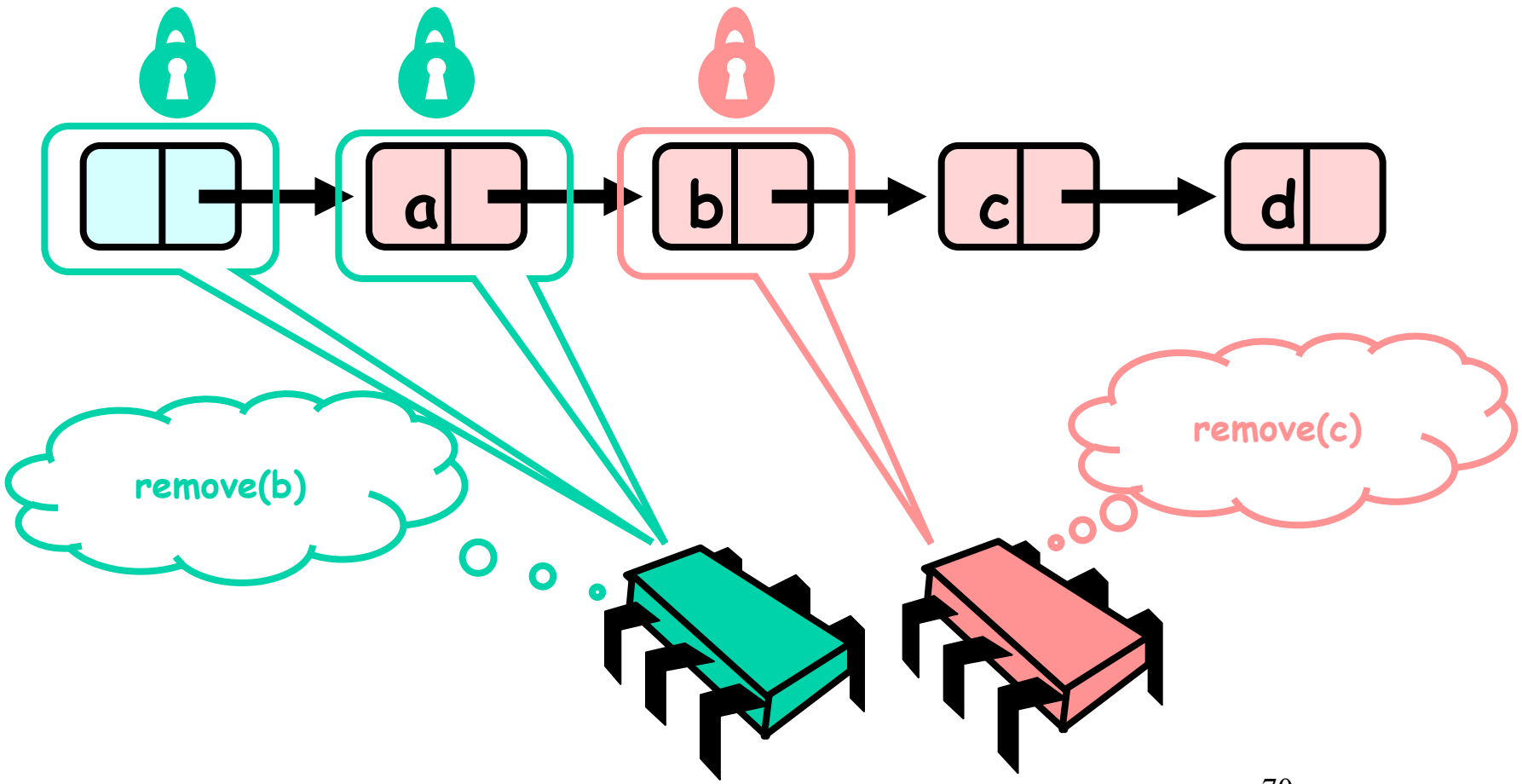
Removing a Node



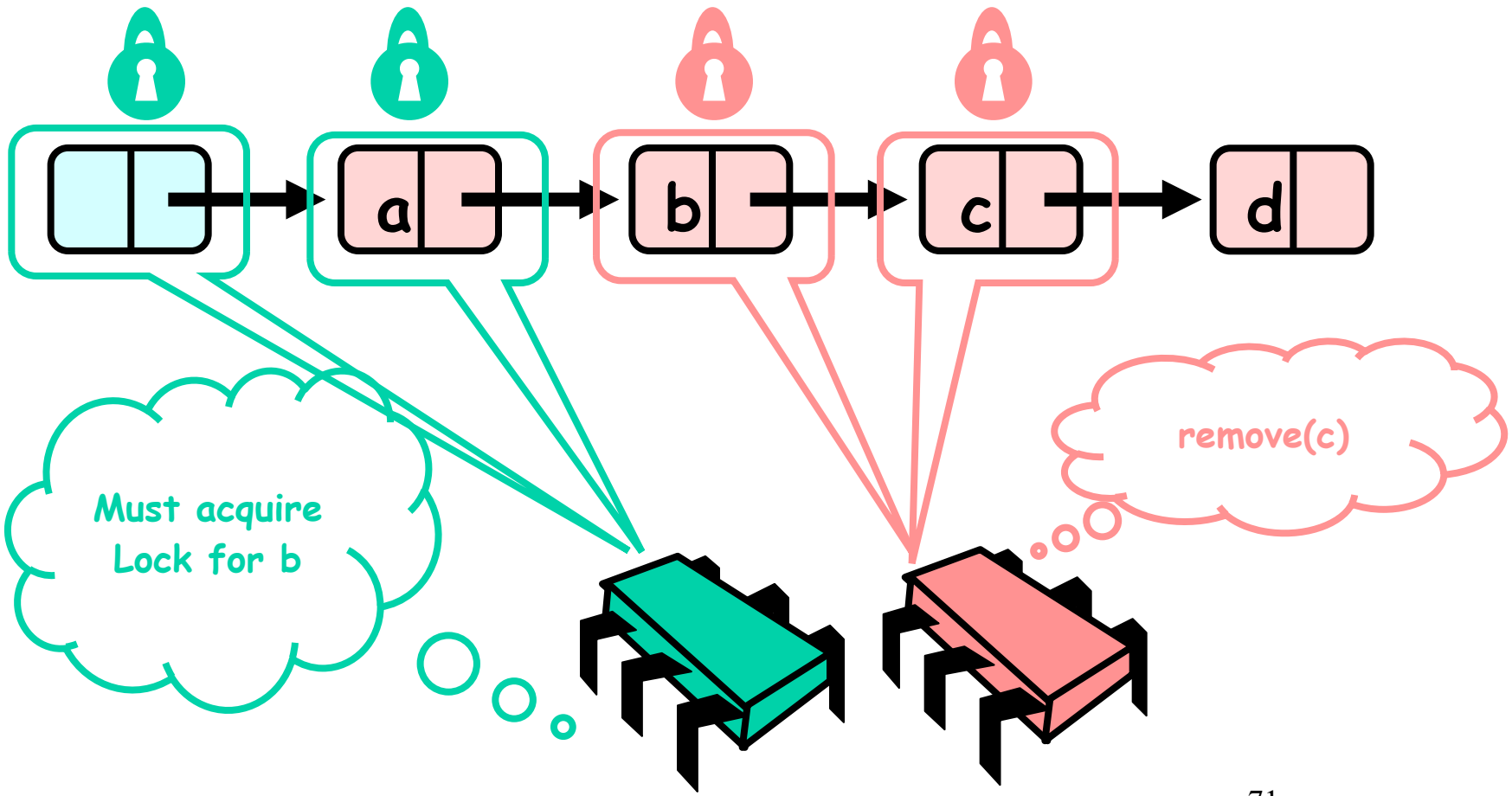
Removing a Node



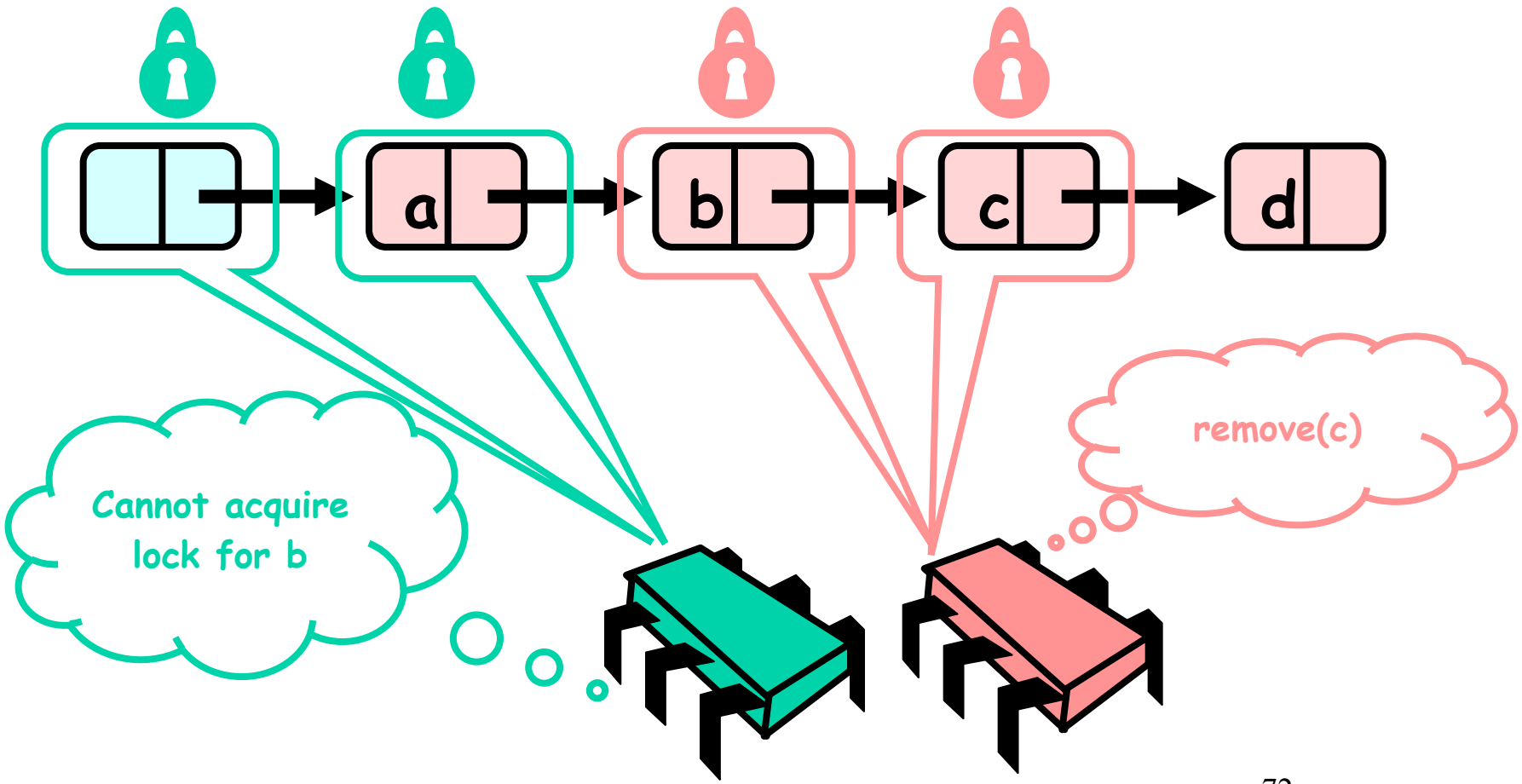
Removing a Node



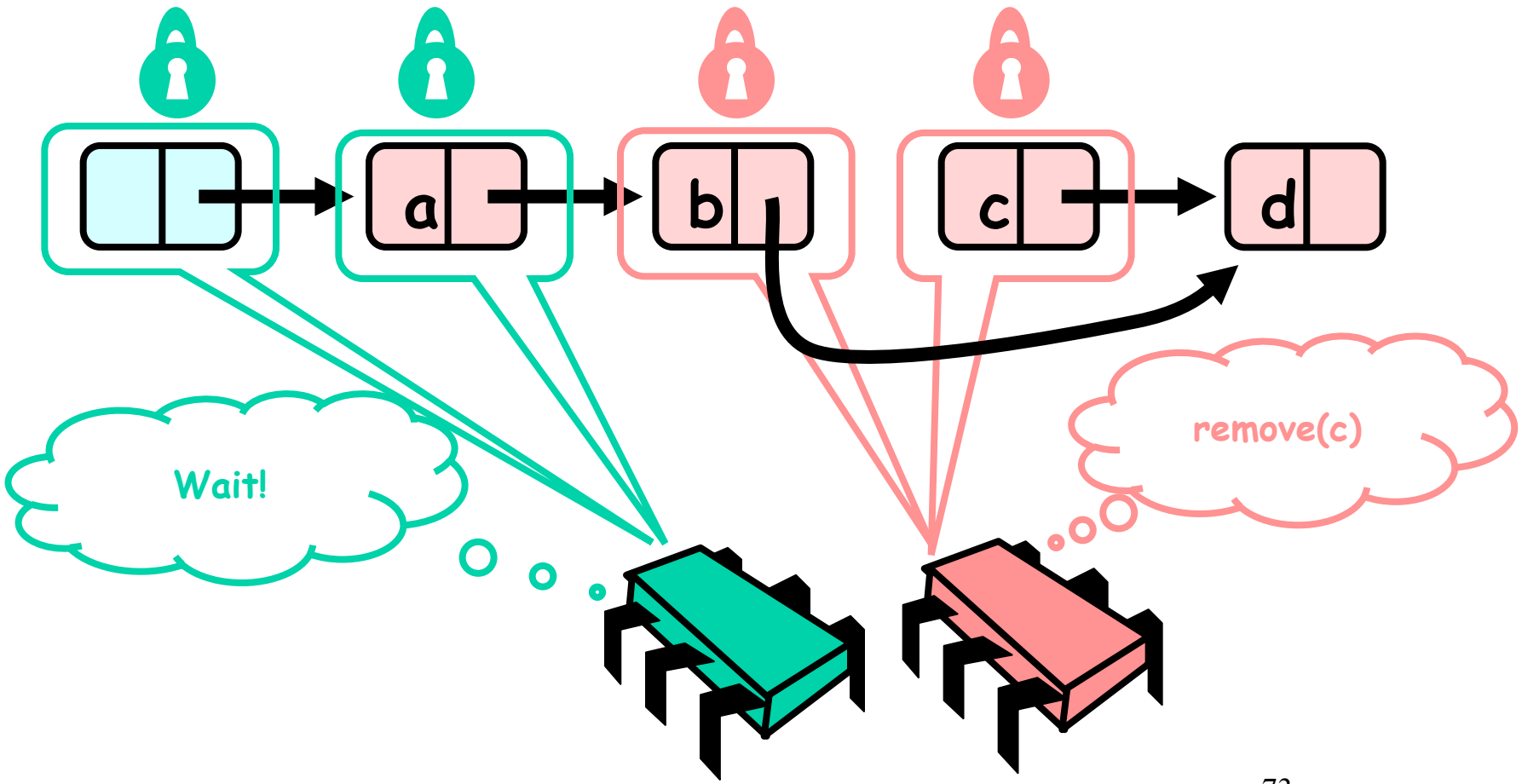
Removing a Node



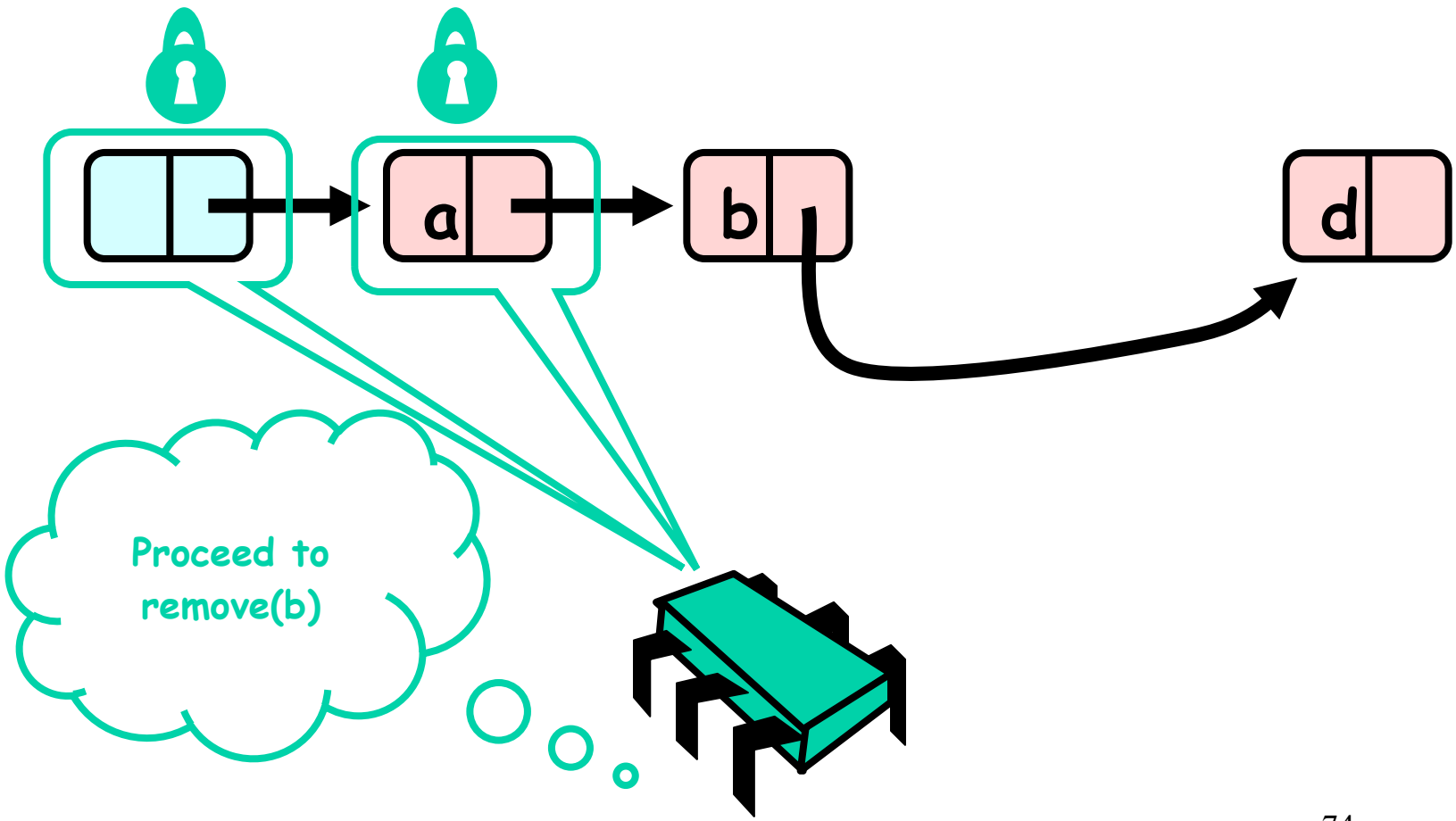
Removing a Node



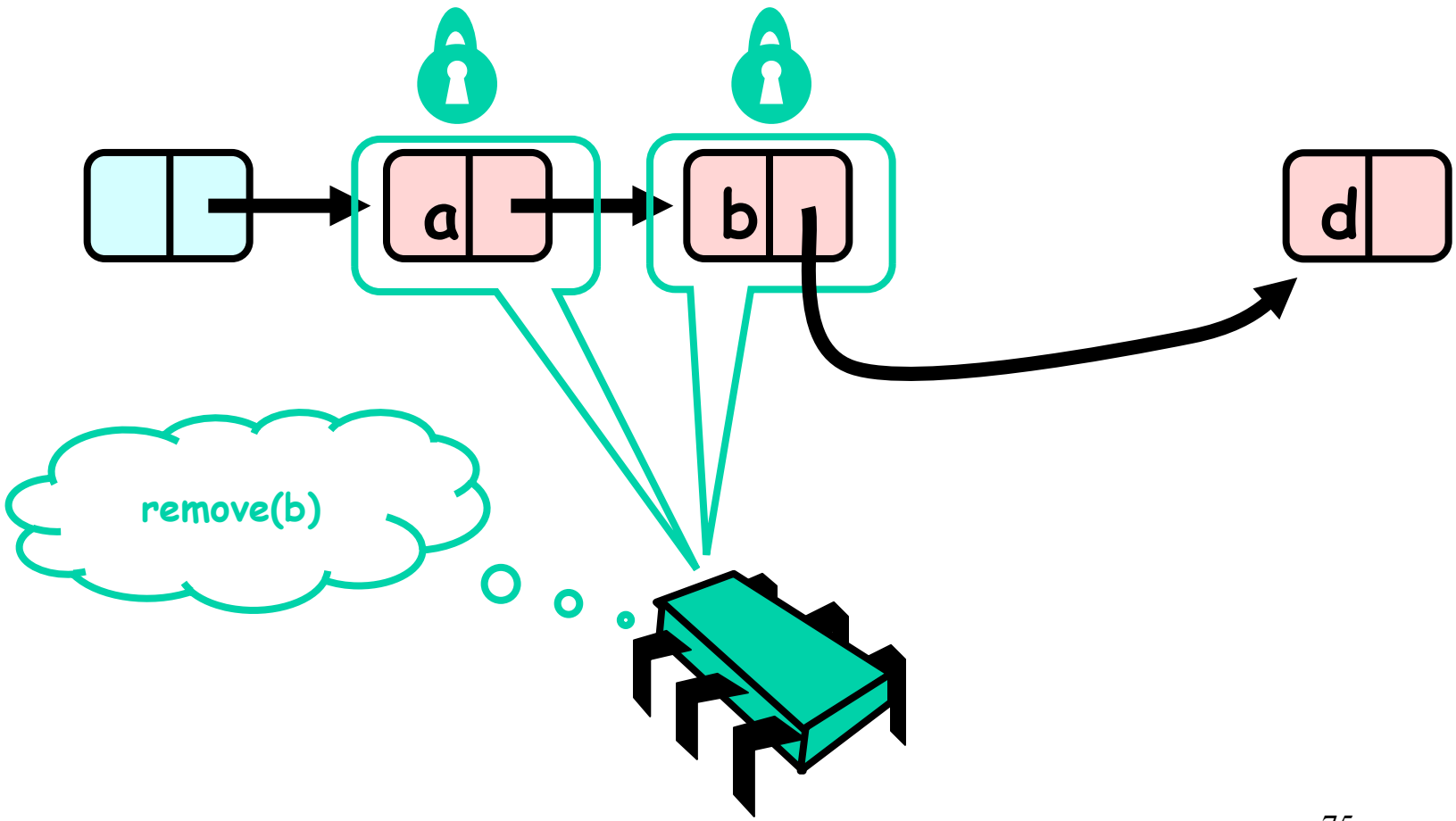
Removing a Node



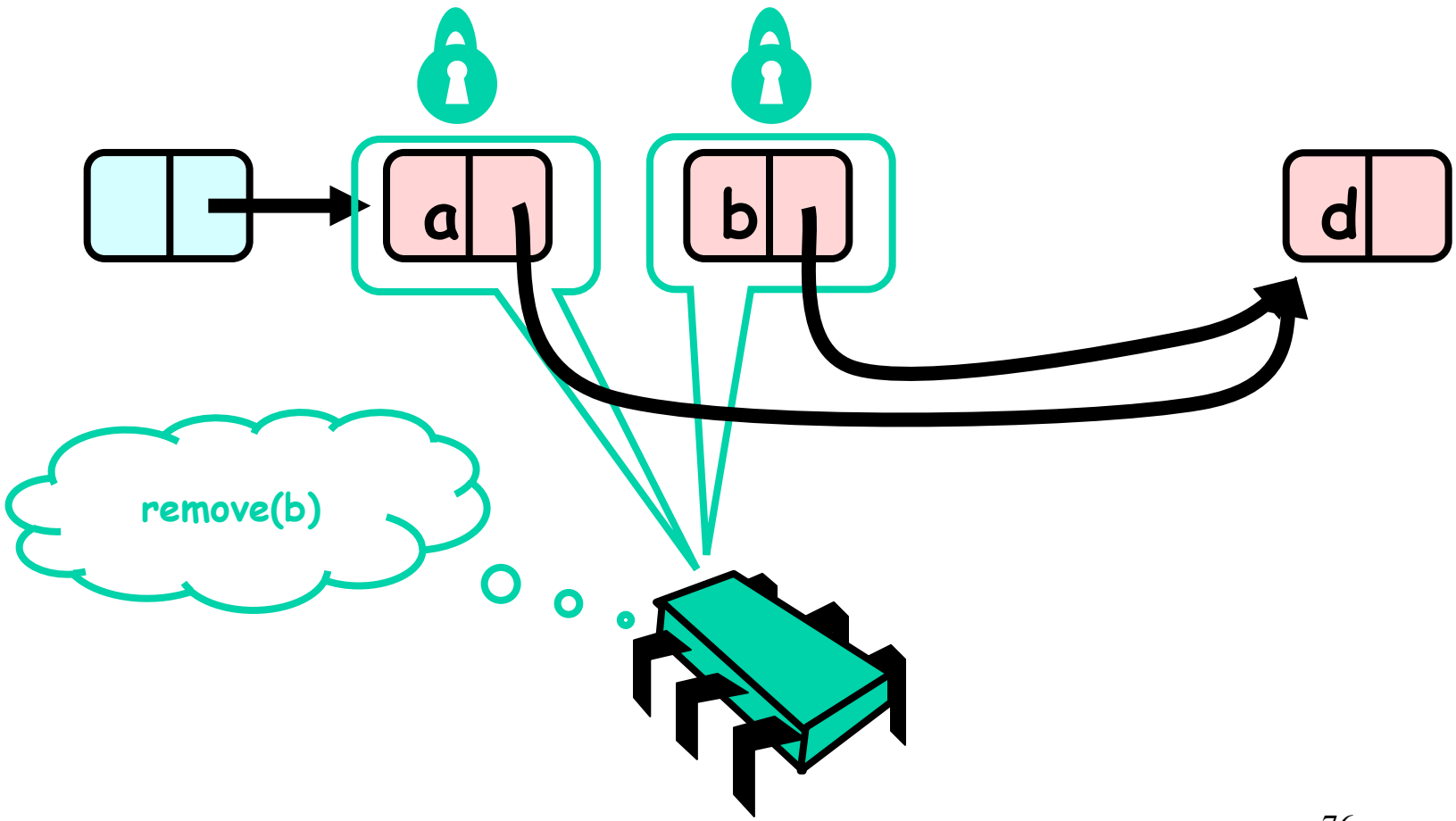
Removing a Node



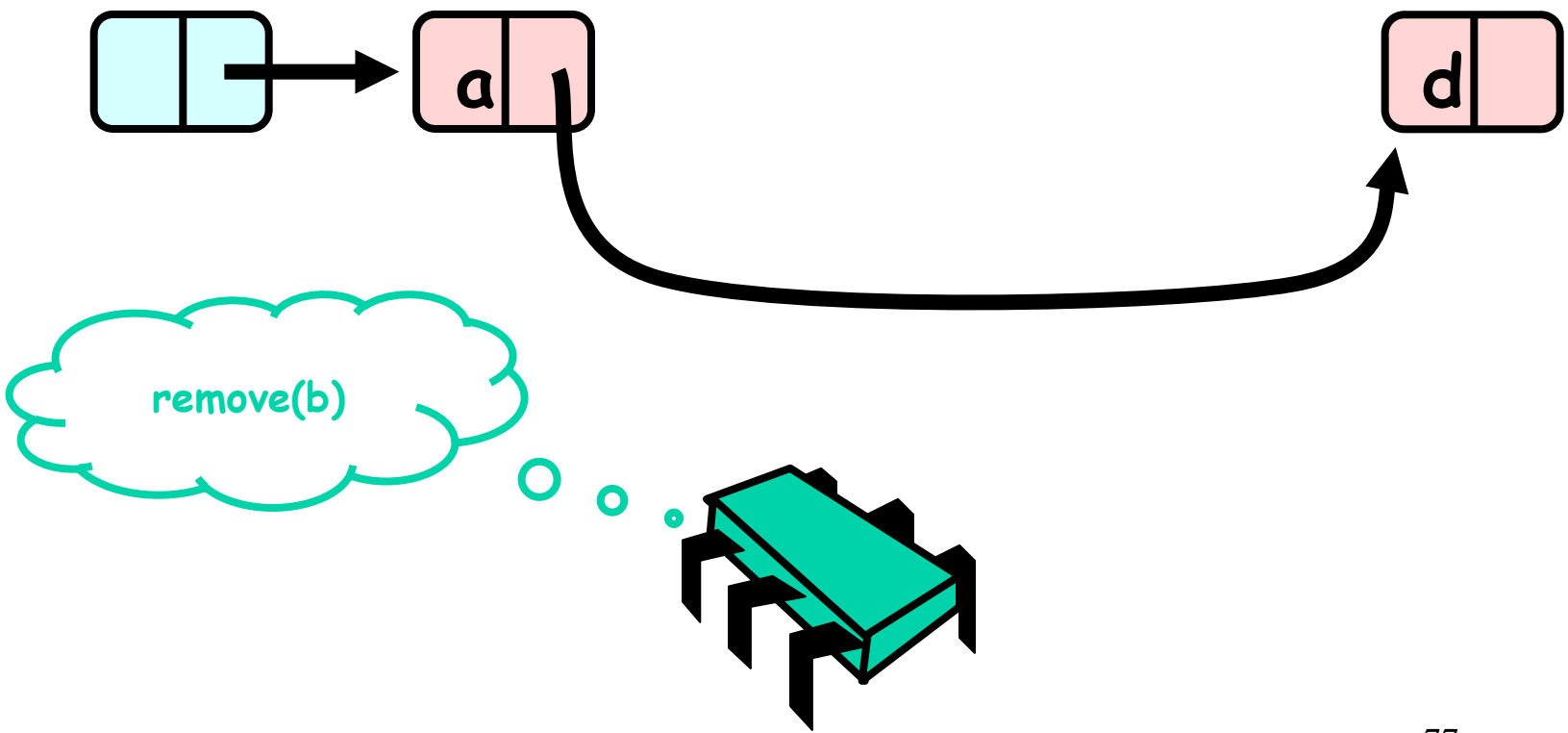
Removing a Node



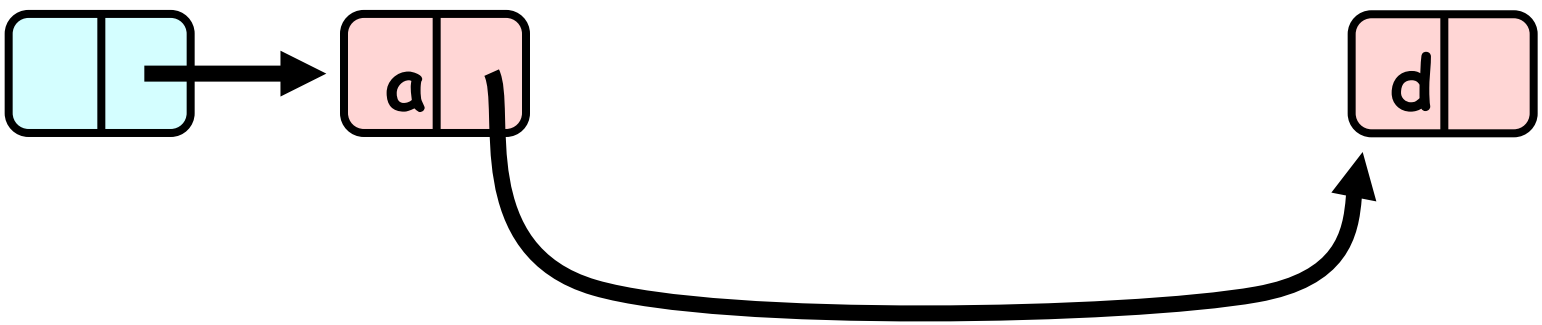
Removing a Node



Removing a Node



Removing a Node



Remove method

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

Remove method

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

Adding Nodes

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

Rep Invariant

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

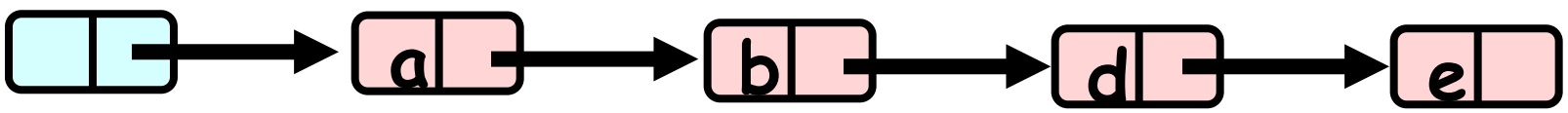
Drawbacks

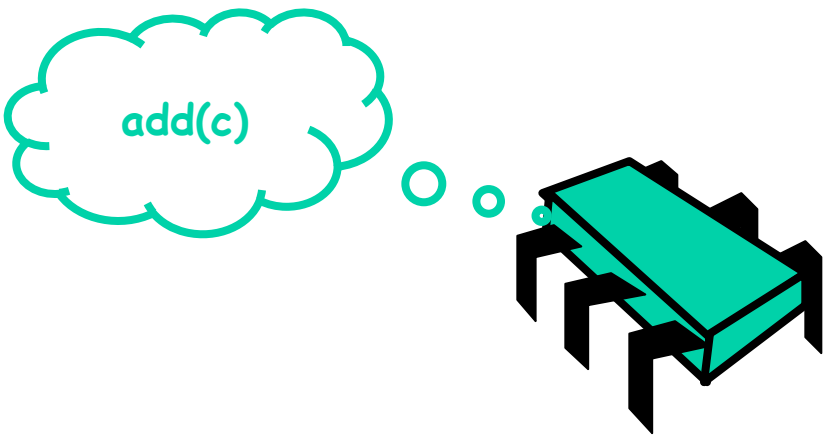
- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient

Optimistic Synchronization

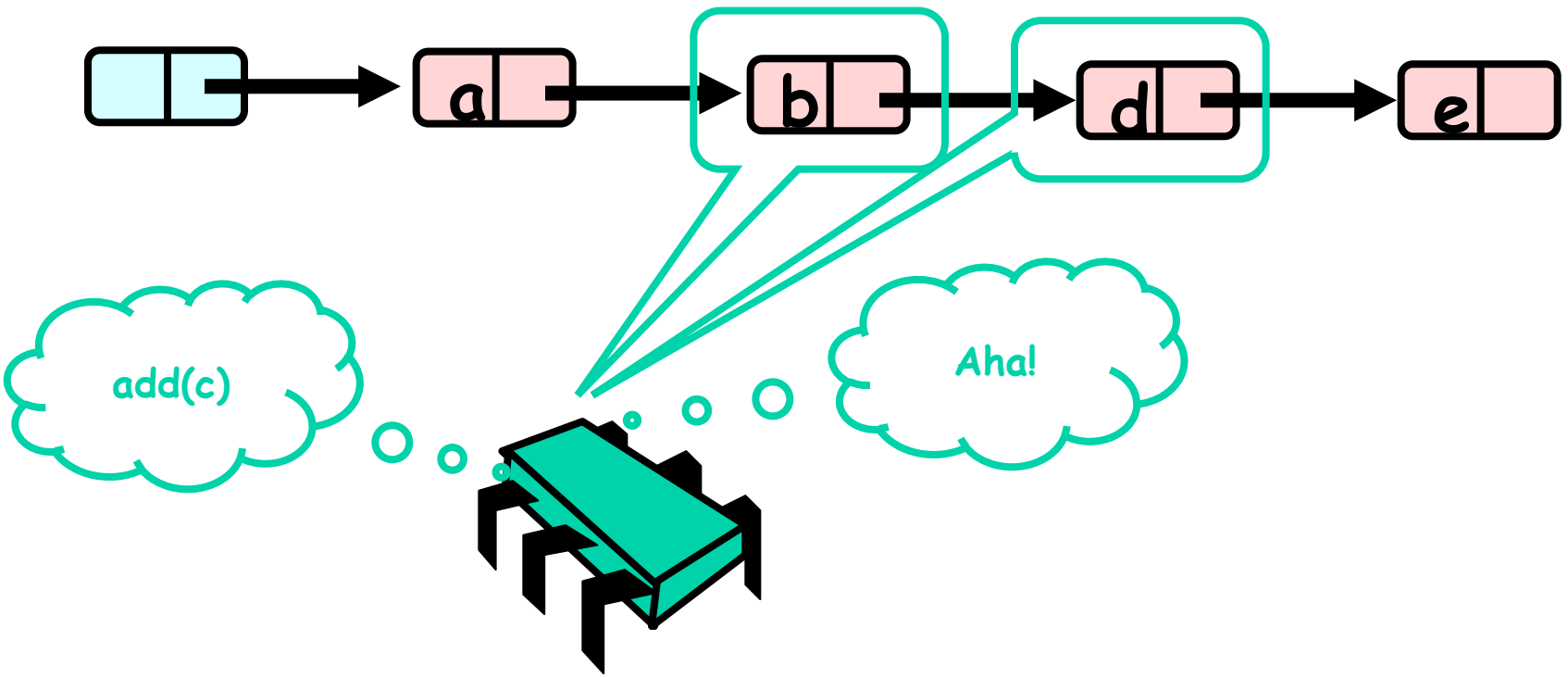
- Find nodes without locking
- Lock nodes
- Check that everything is OK

Optimistic: Traverse without Locking

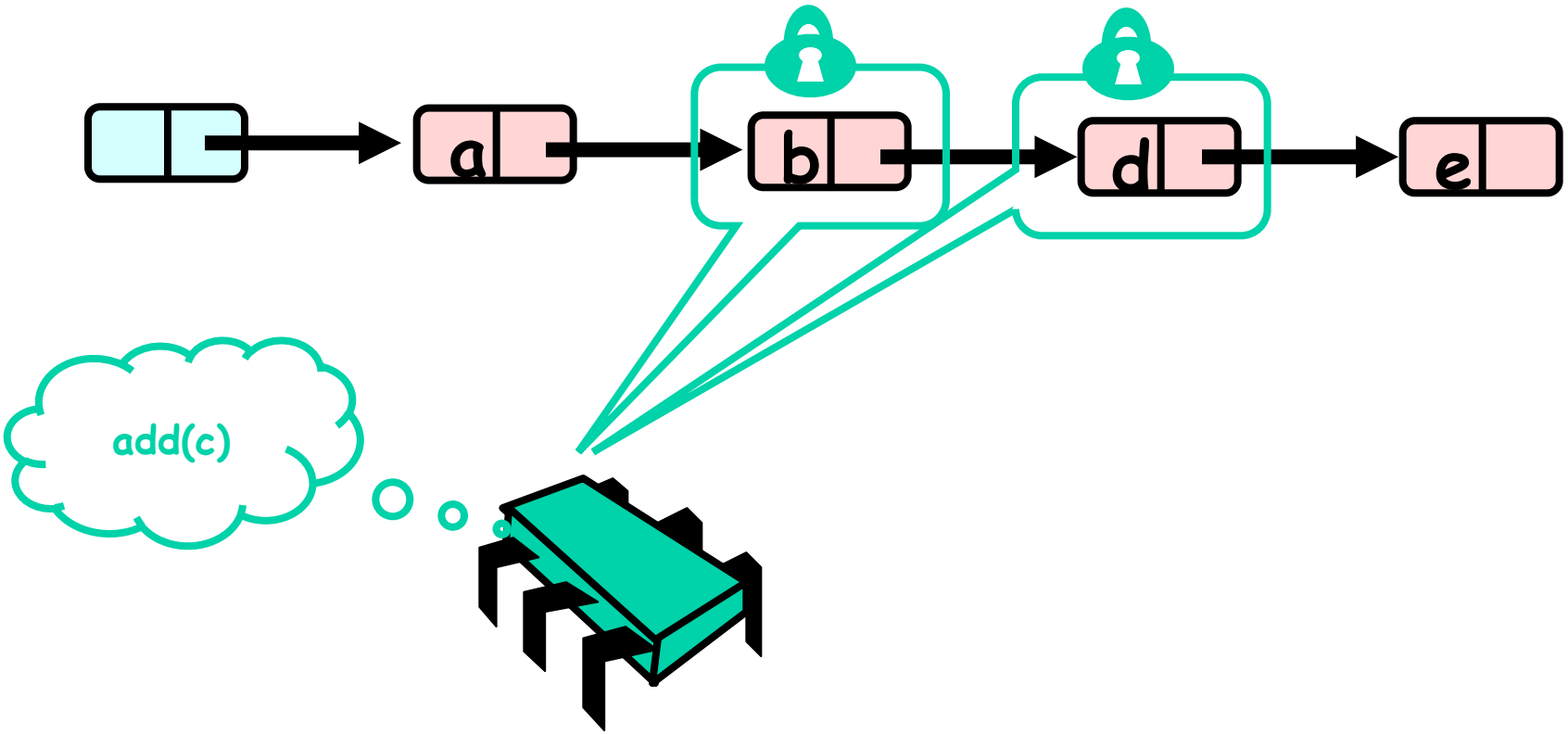




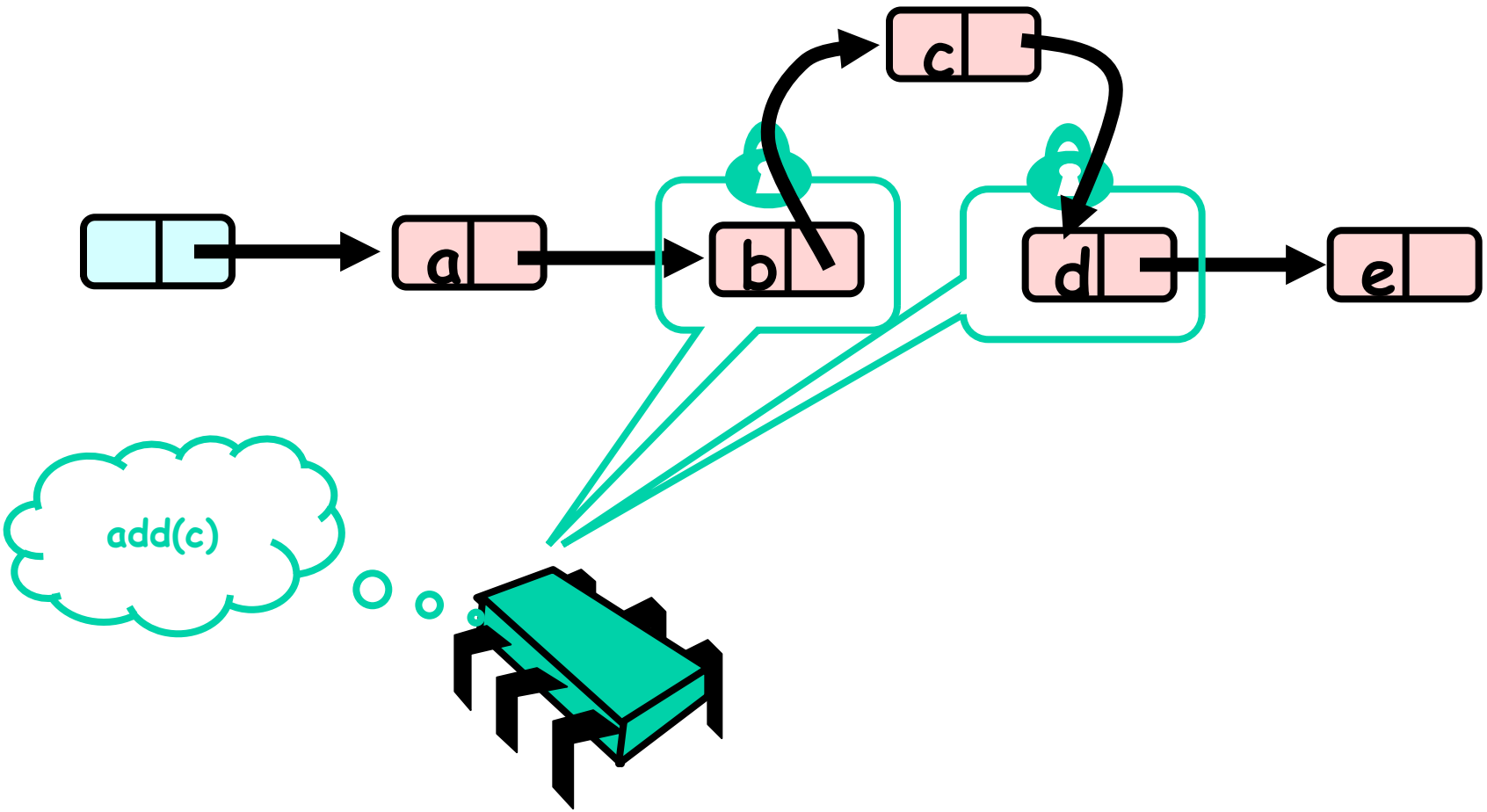
Optimistic: Traverse without Locking



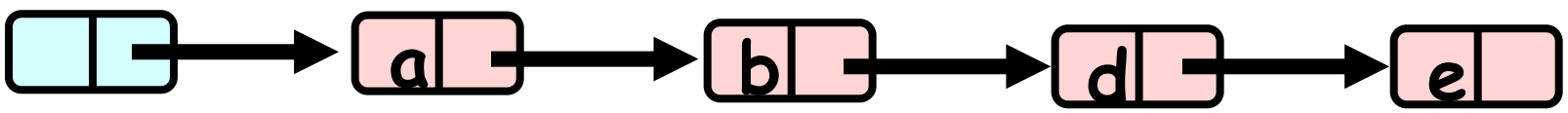
Optimistic: Lock and Load

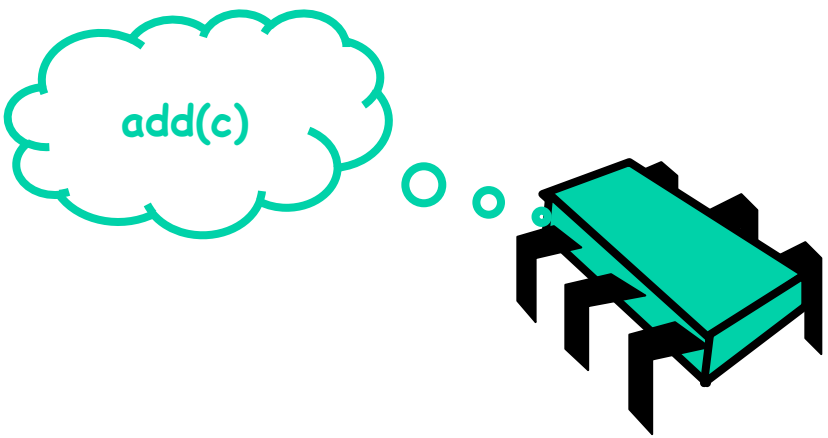


Optimistic: Lock and Load

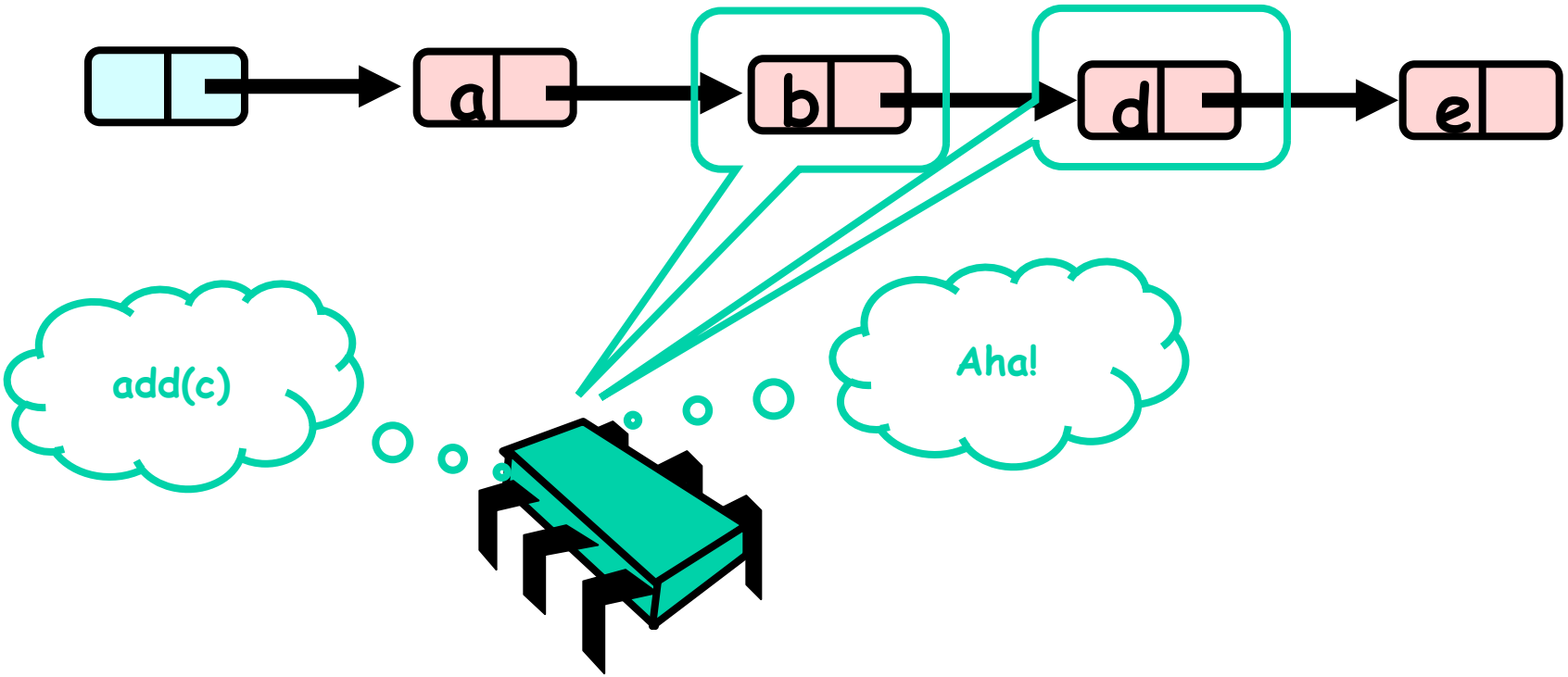


What could go wrong?

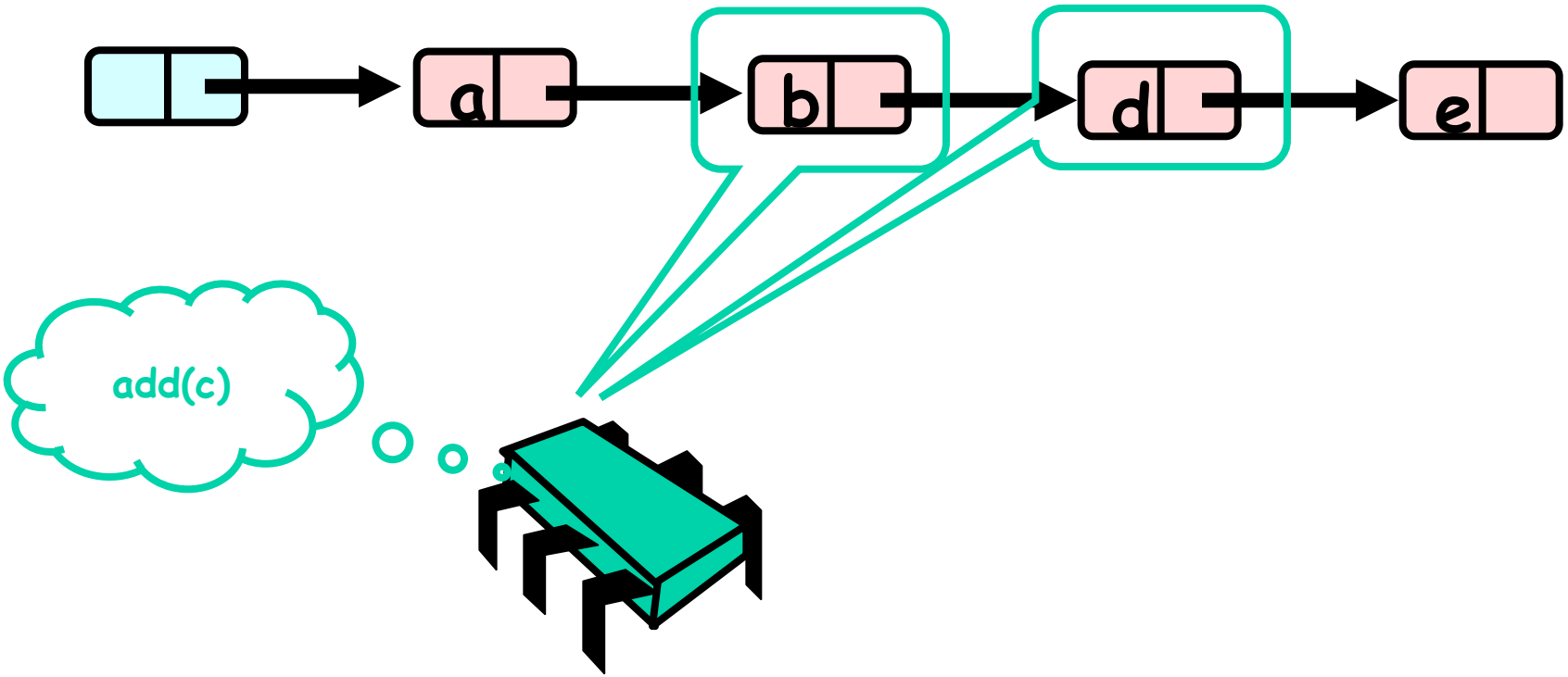




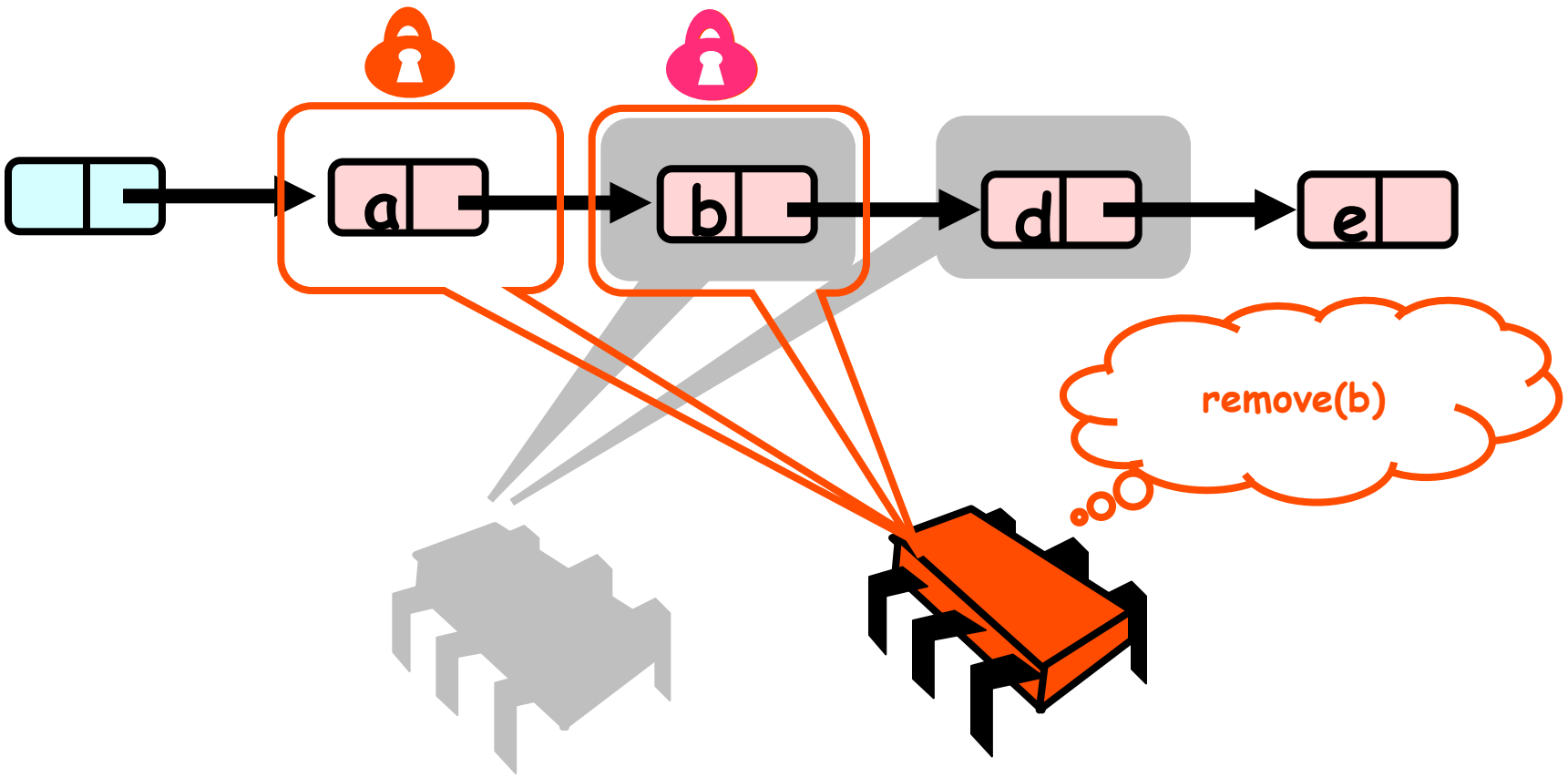
What could go wrong?



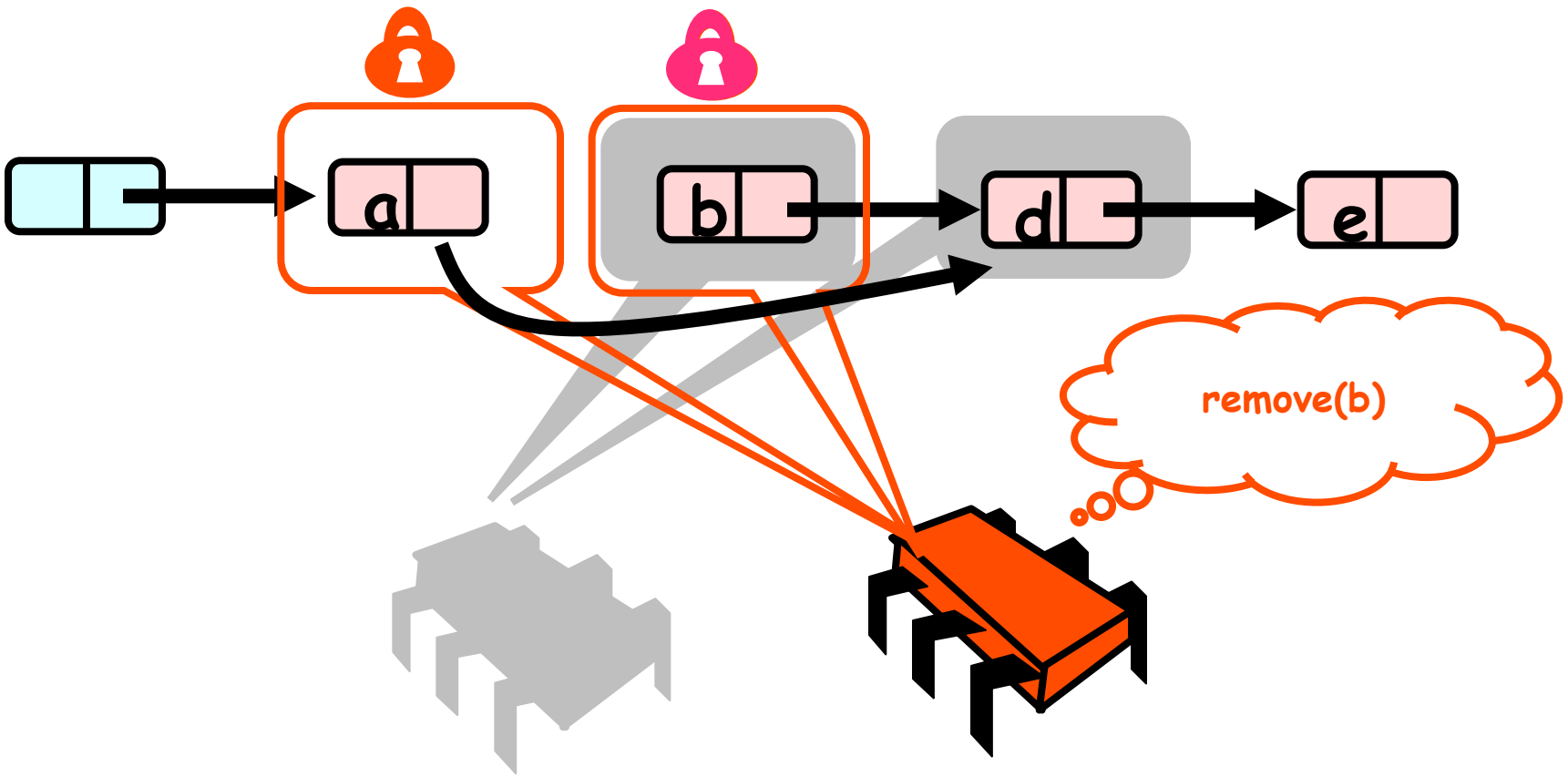
What could go wrong?



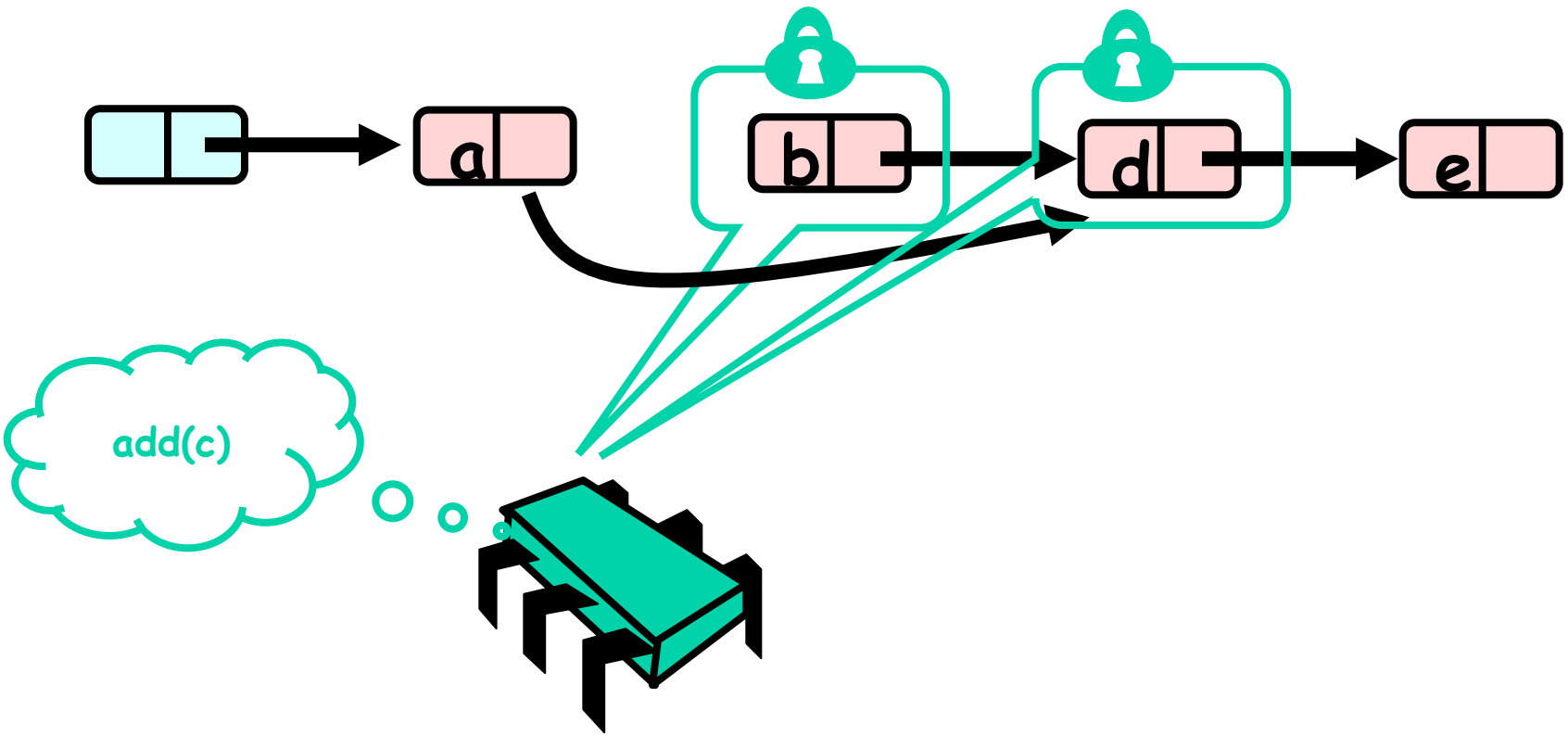
What could go wrong?



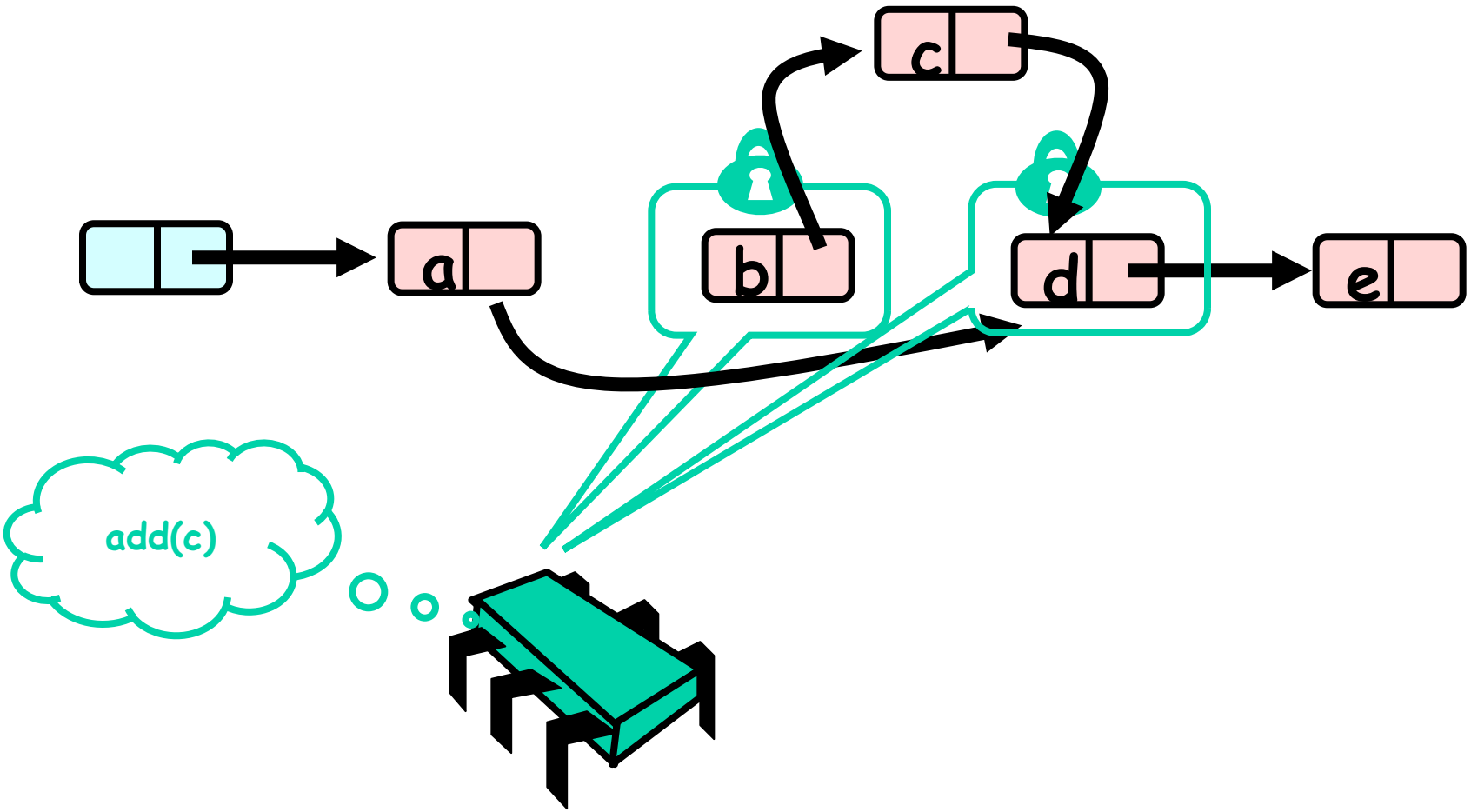
What could go wrong?



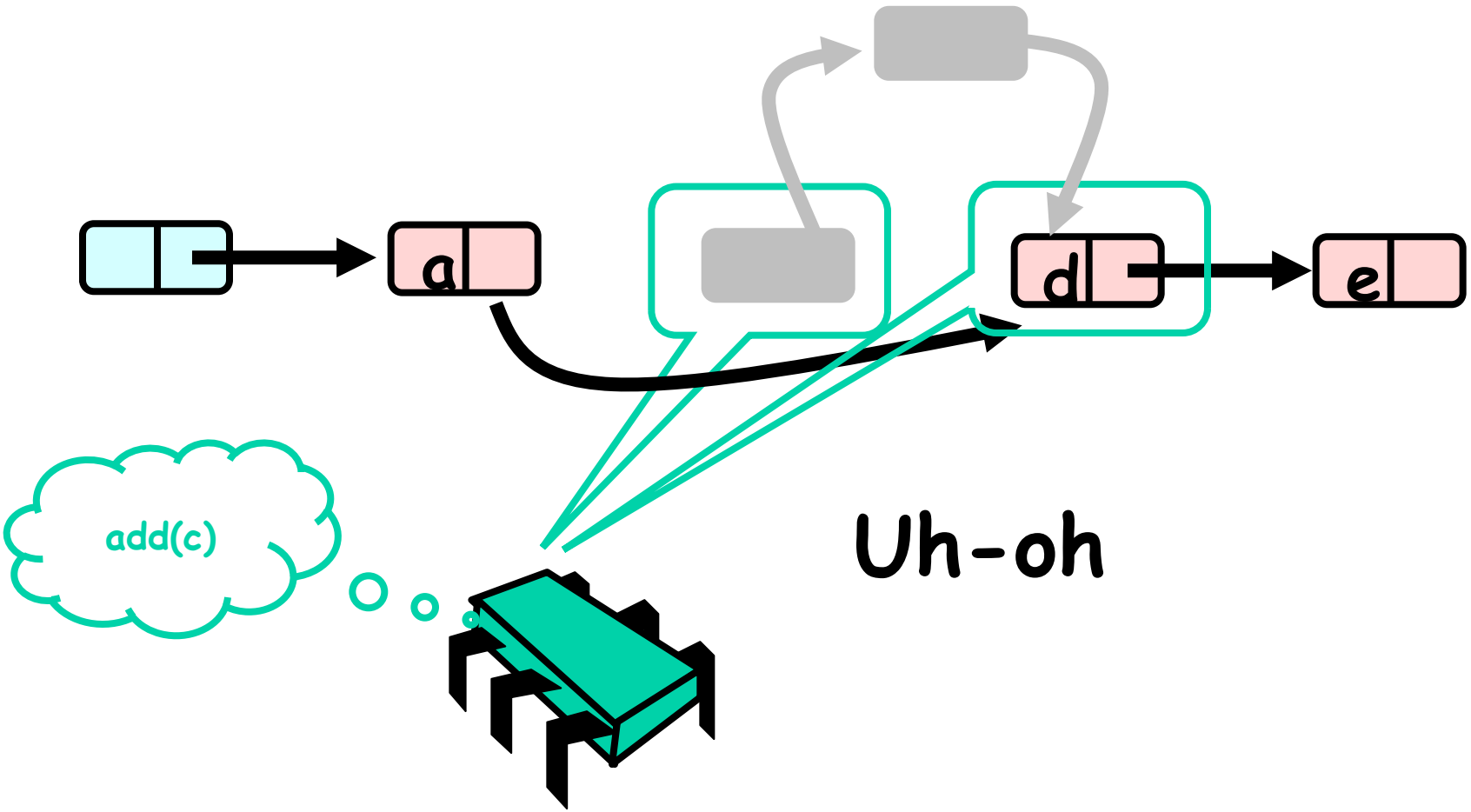
What could go wrong?



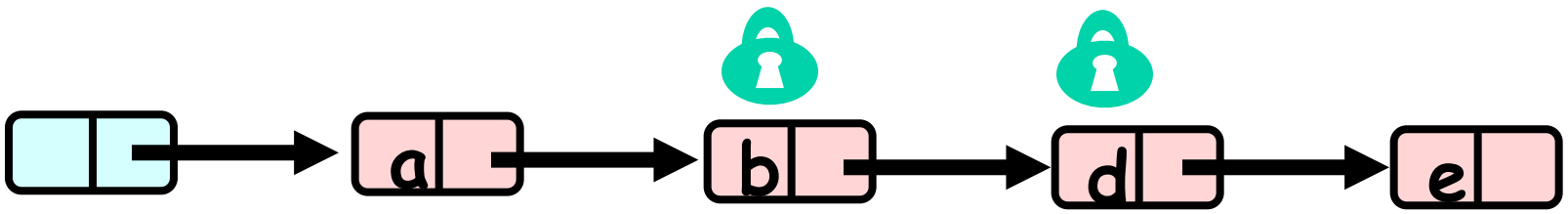
What could go wrong?

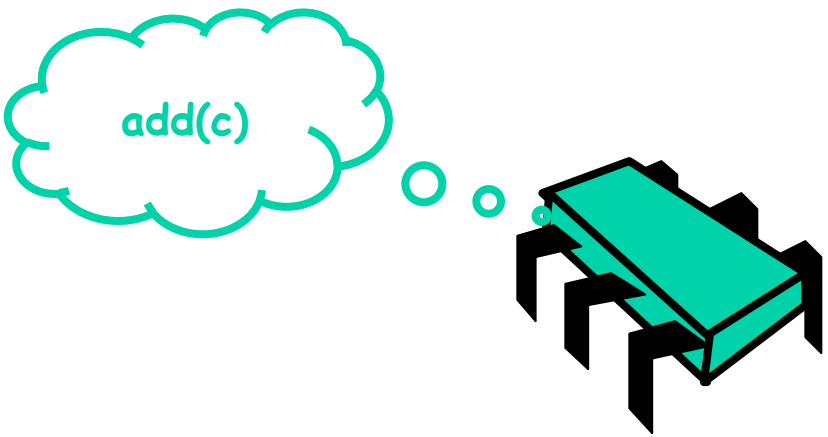


What could go wrong?

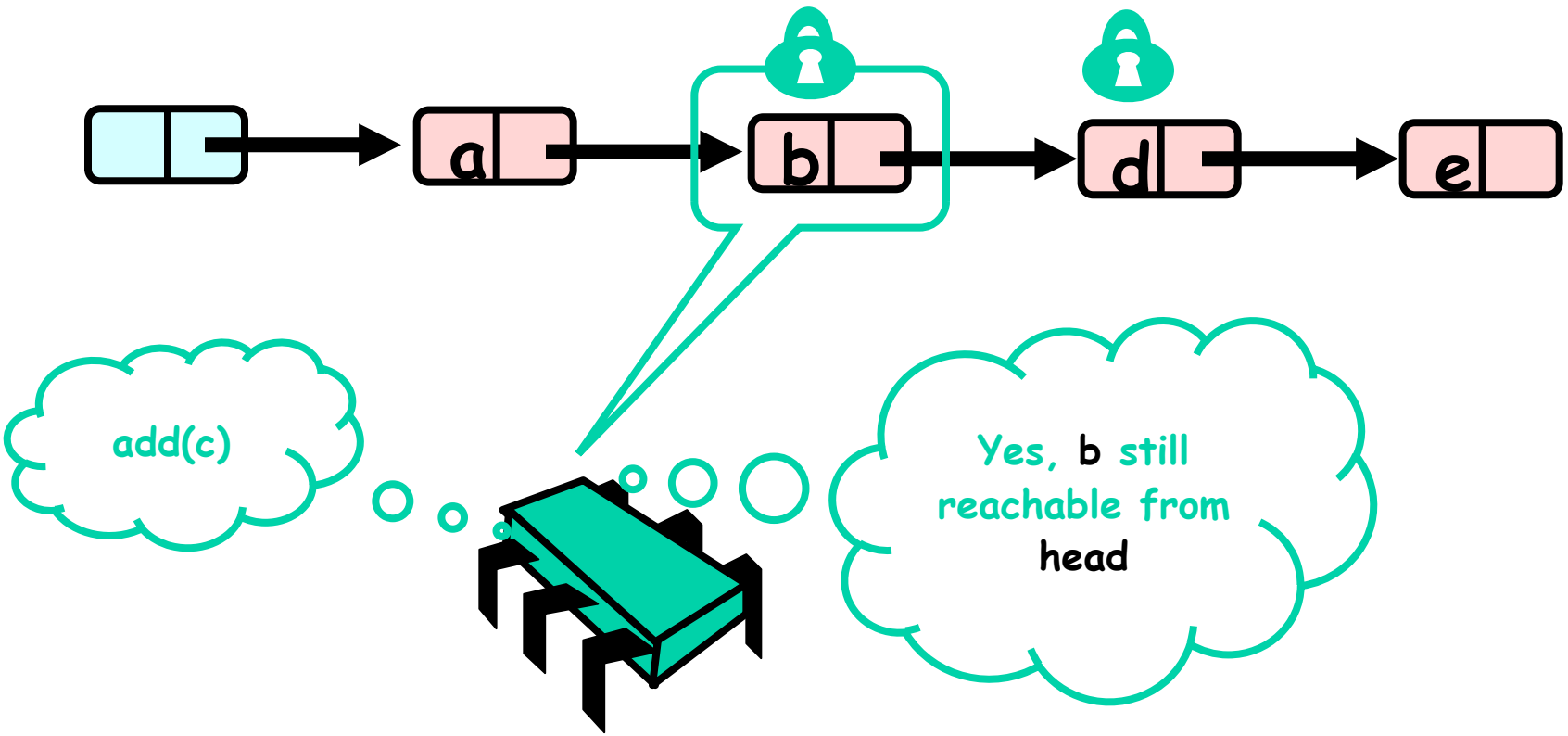


Validate - Part 1

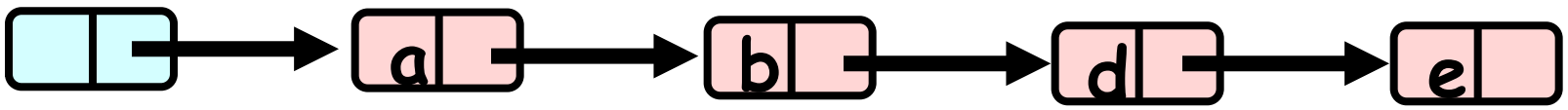


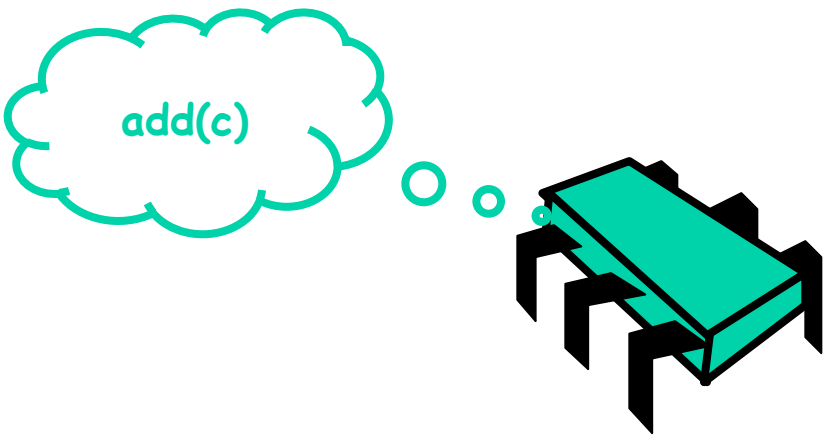


Validate – Part 1

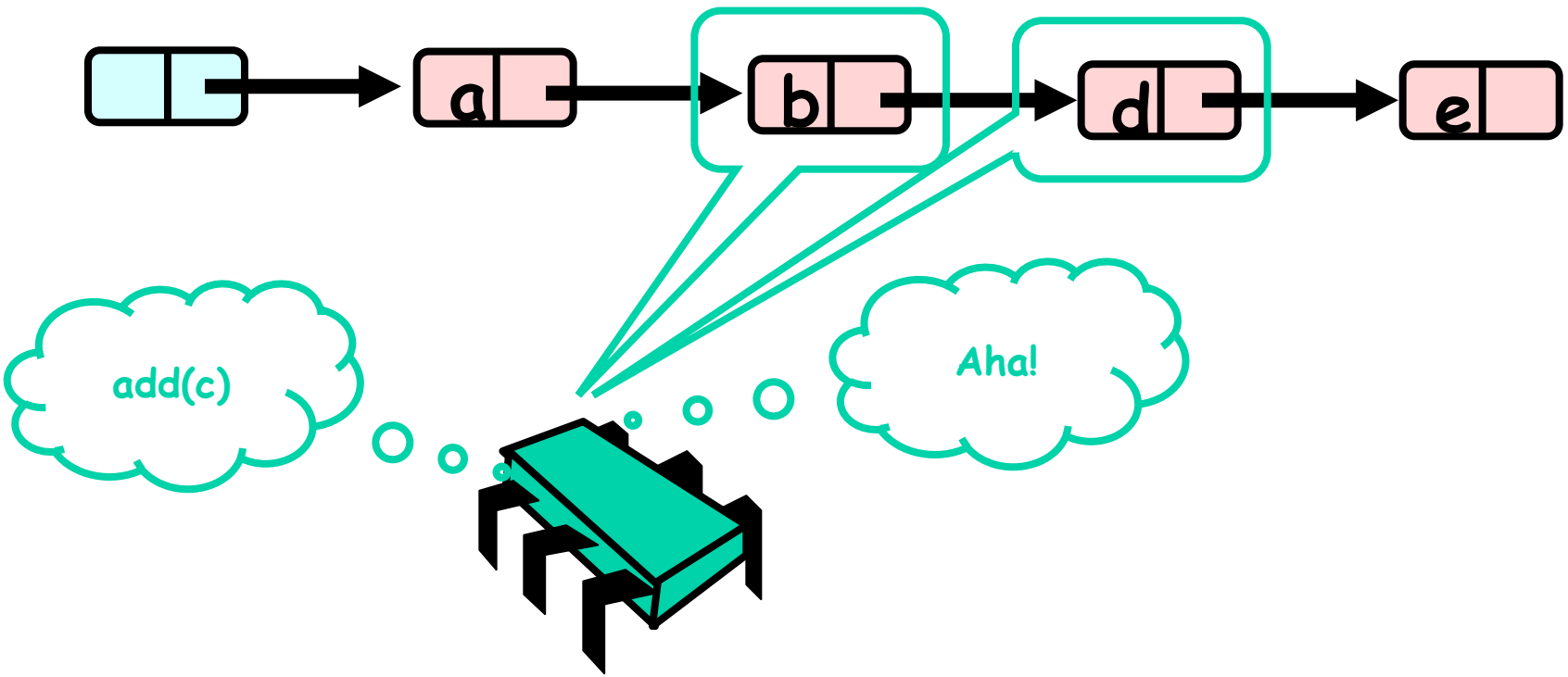


What Else Could Go Wrong?

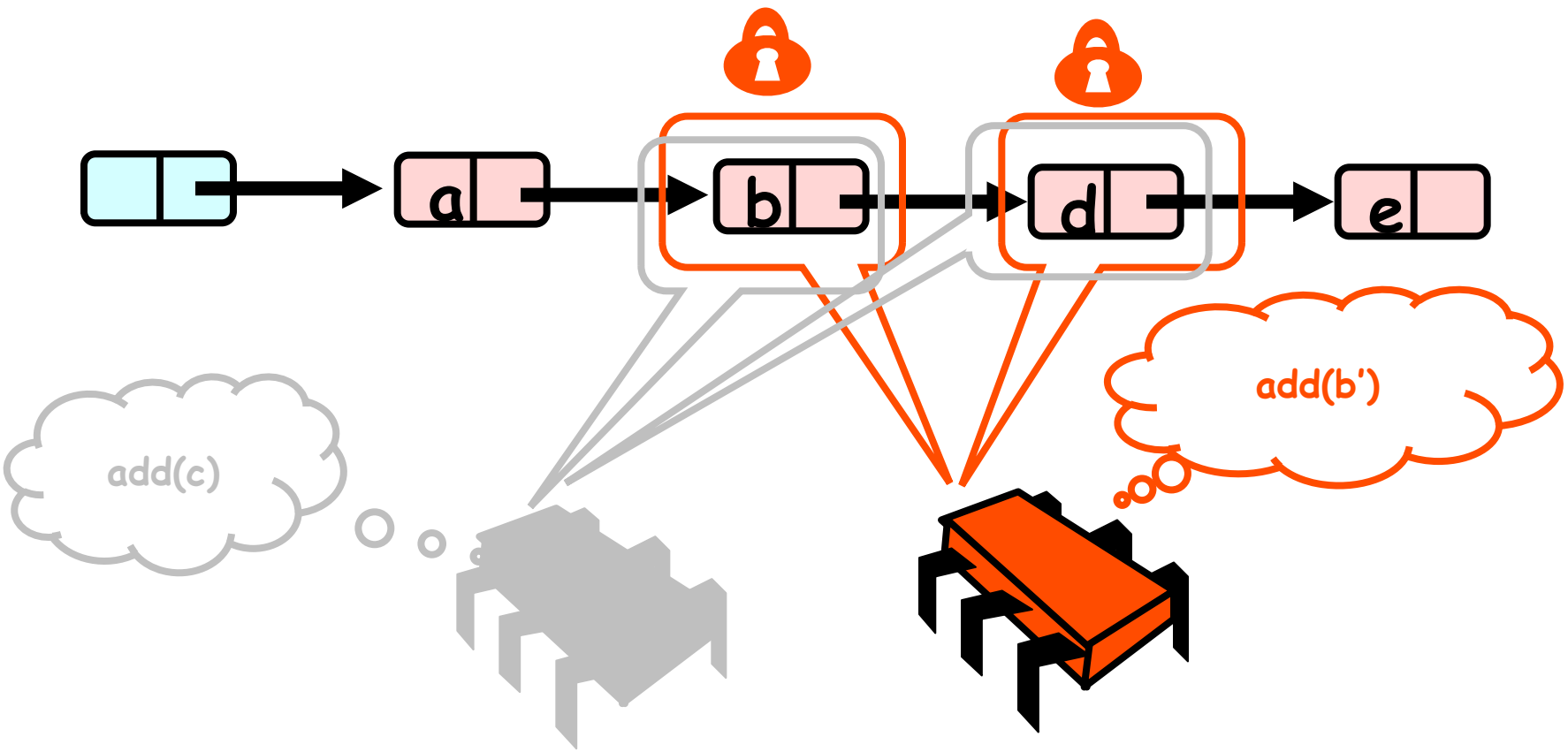




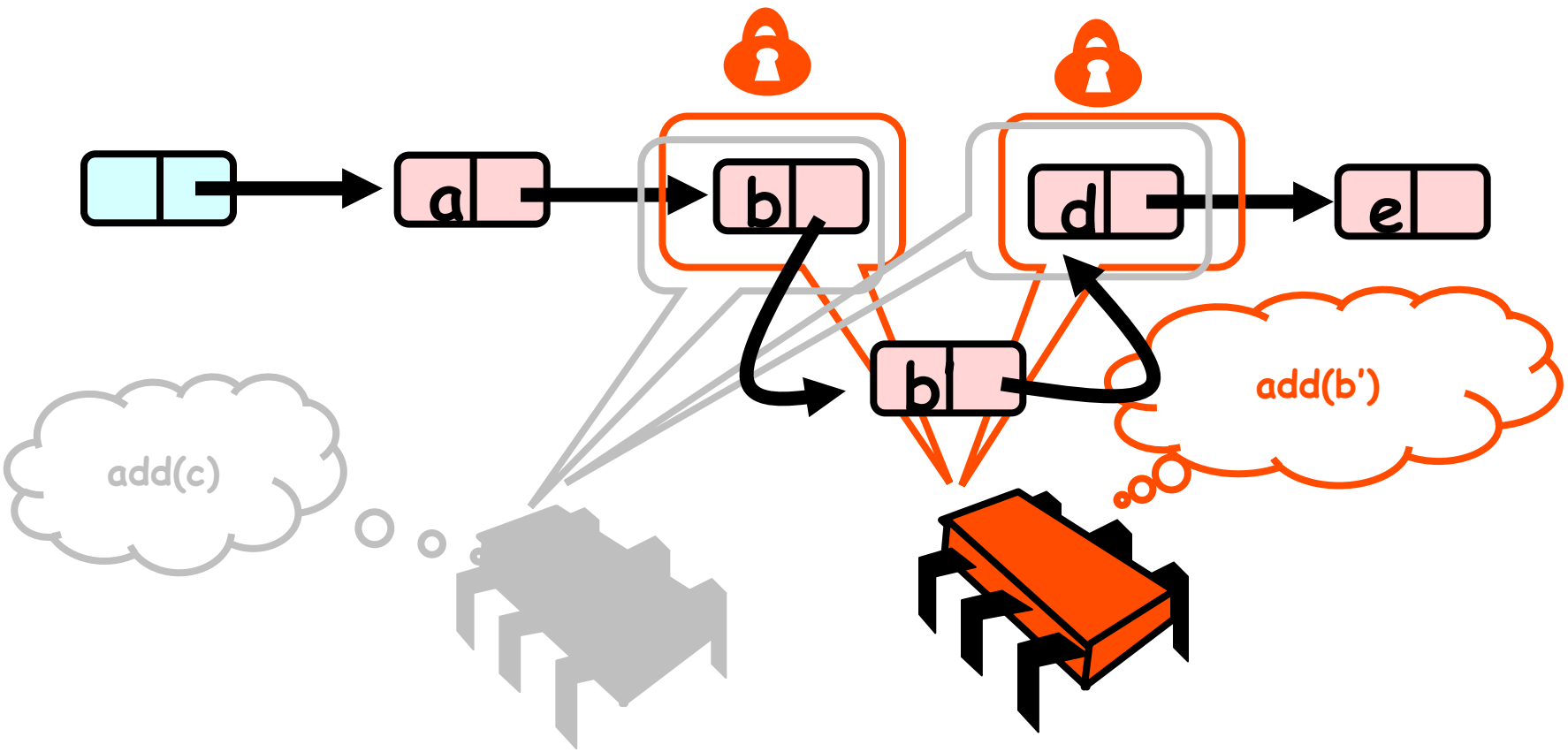
What Else Could Go Wrong?



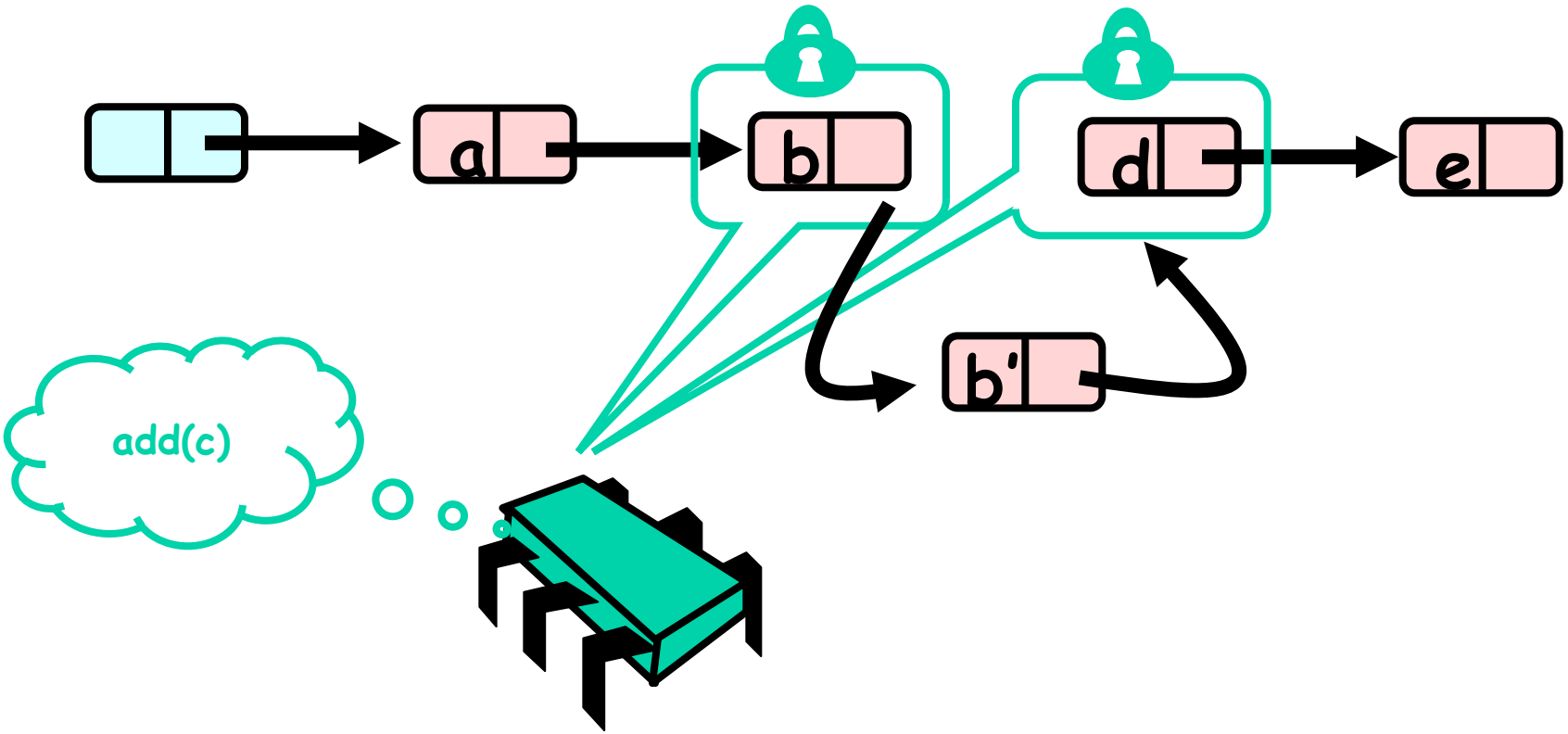
What Else Coould Go Wrong?



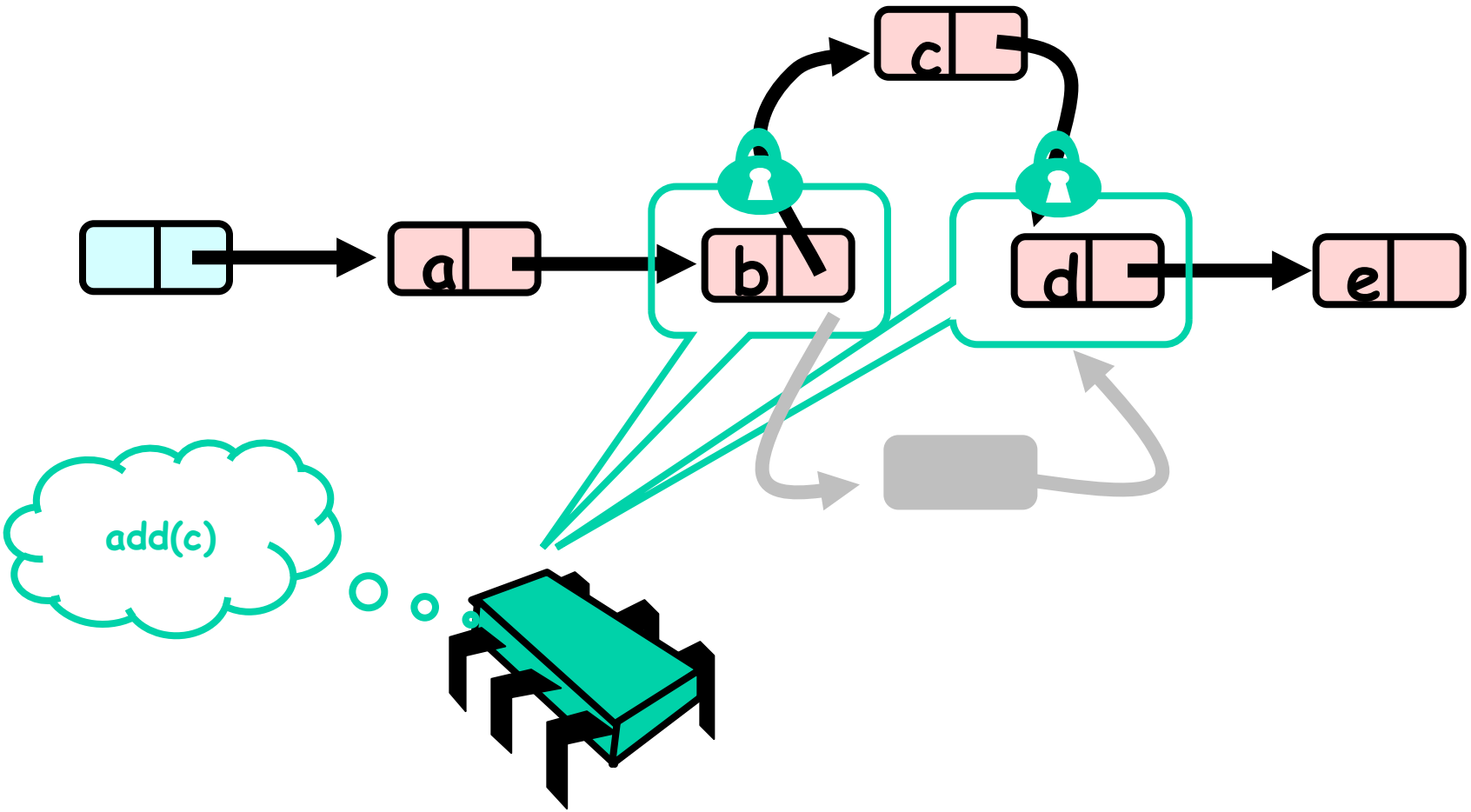
What Else Coould Go Wrong?



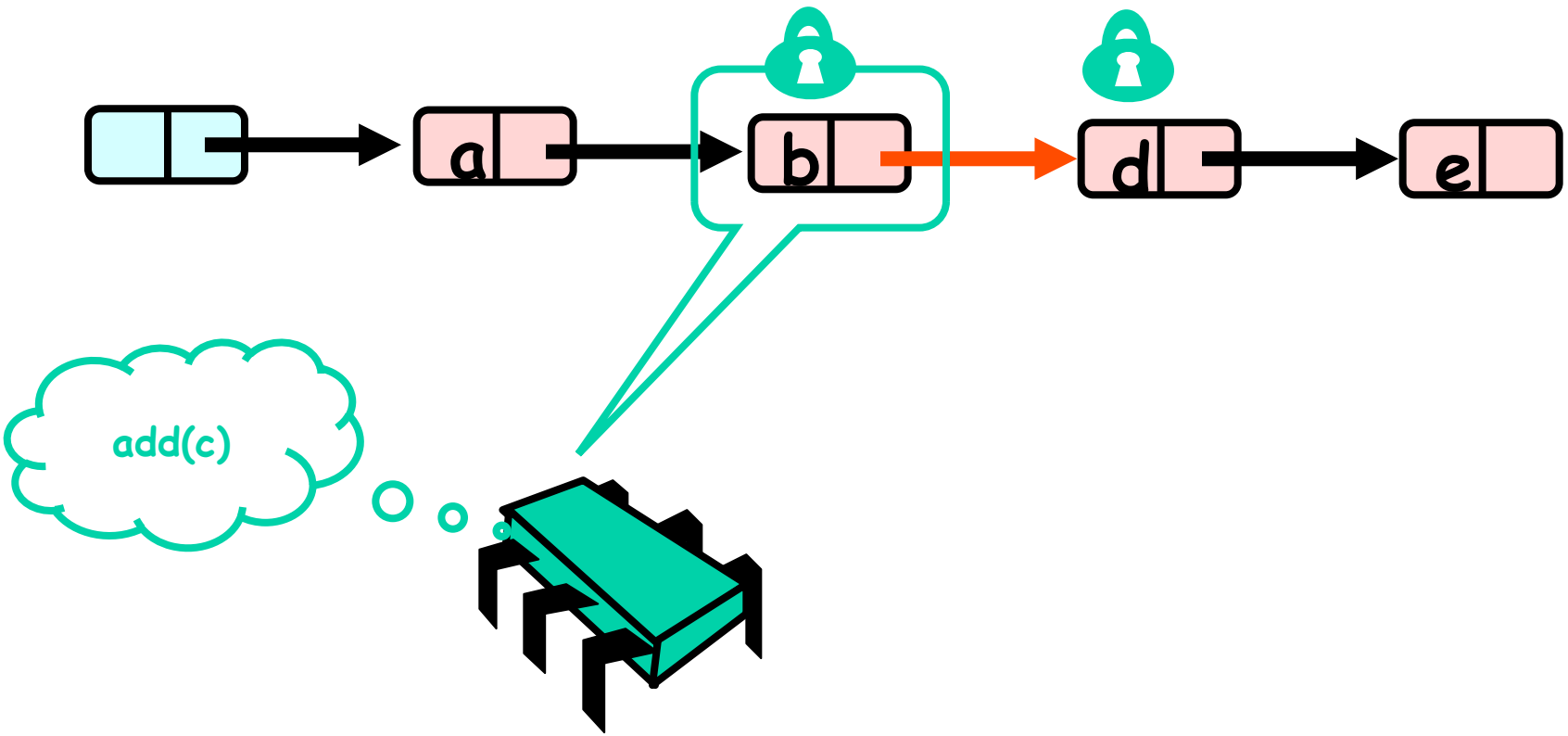
What Else Could Go Wrong?



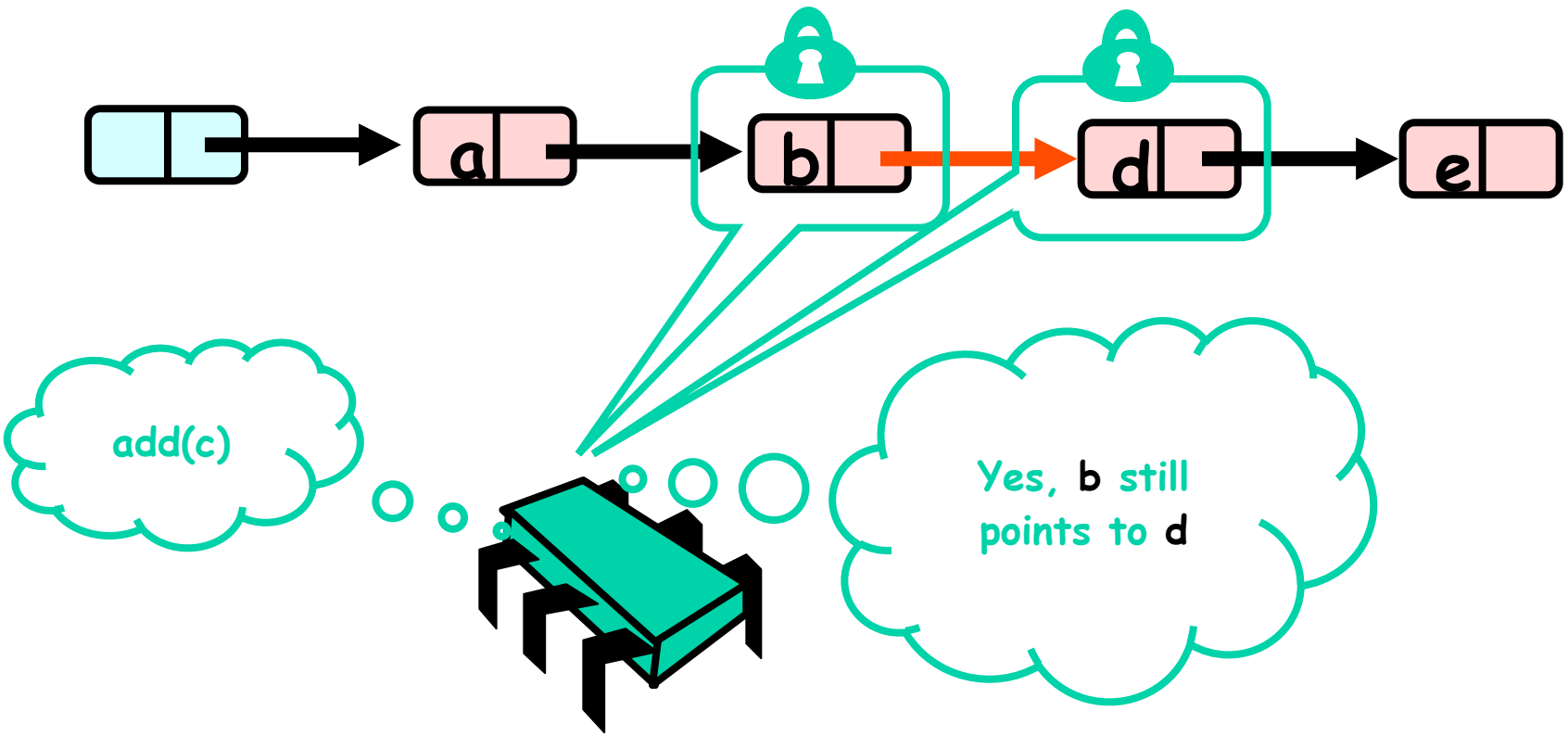
What Else Could Go Wrong?



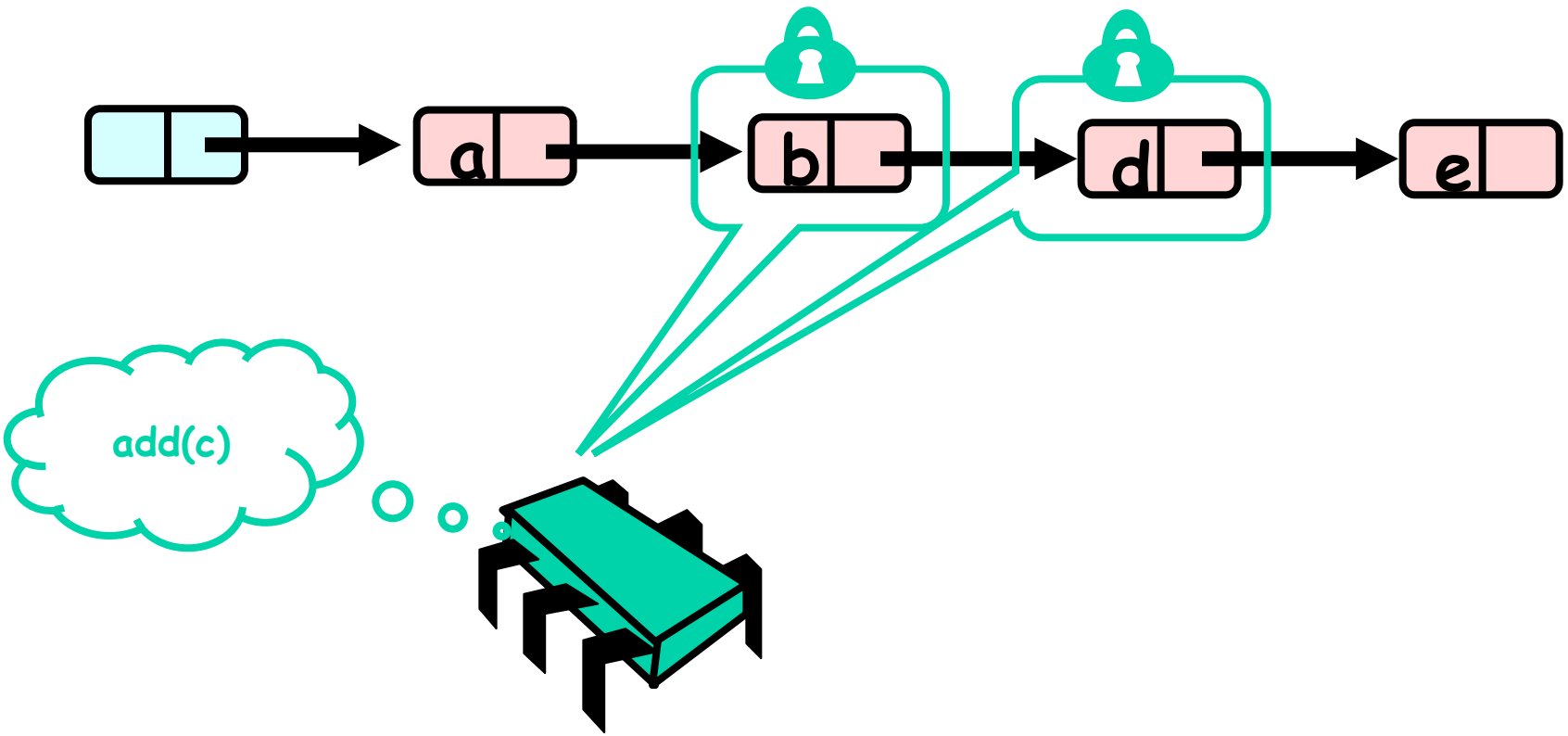
Validate Part 2 (while holding locks)



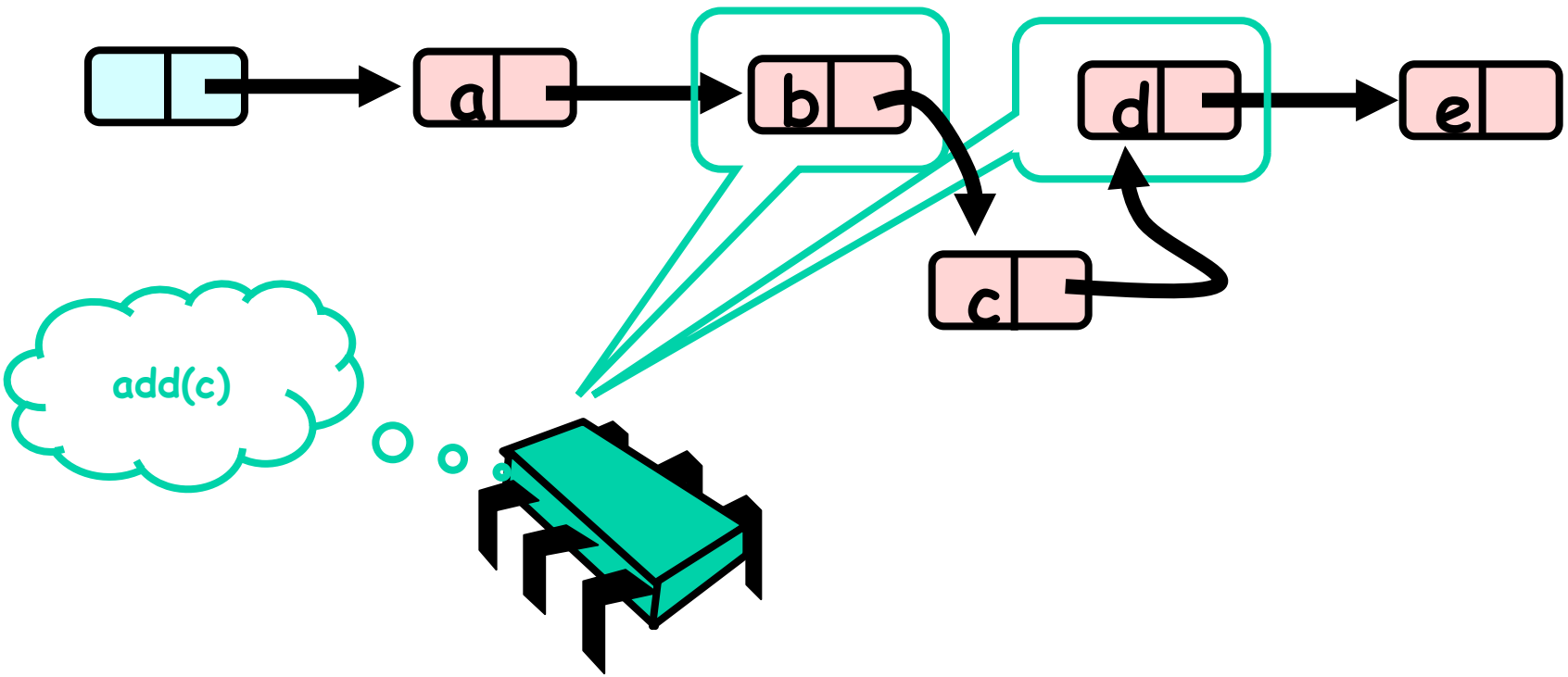
Validate Part 2 (while holding locks)



Optimistic: Linearization Point



Optimistic: Linearization Point



Same Abstraction Map

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

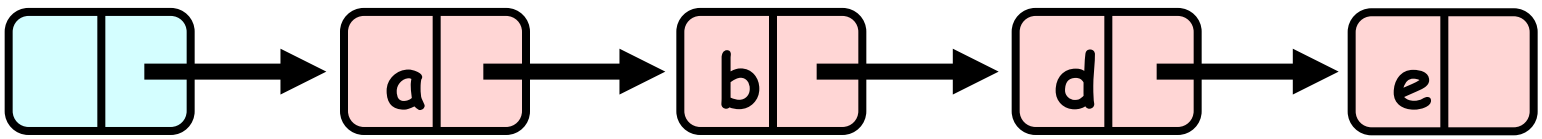
Invariants

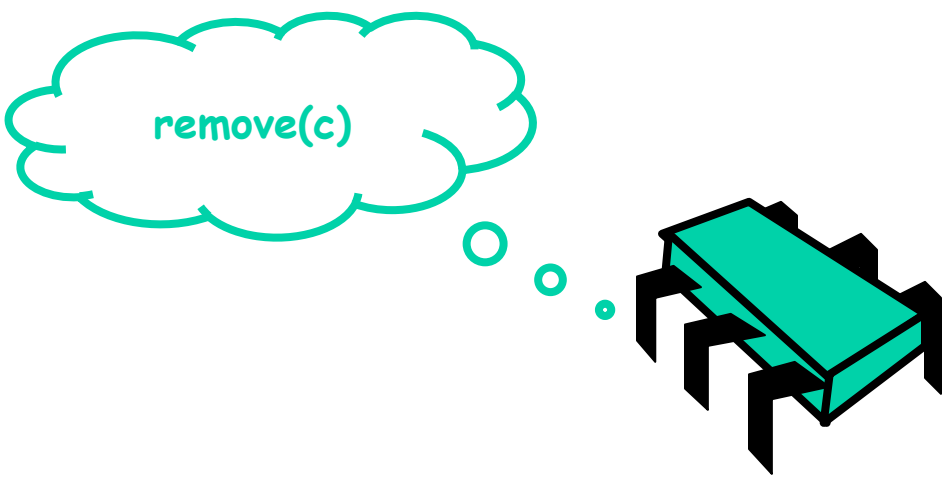
- Careful: we may traverse deleted nodes
- But we establish properties by
  - Validation
  - After we lock target nodes

Correctness

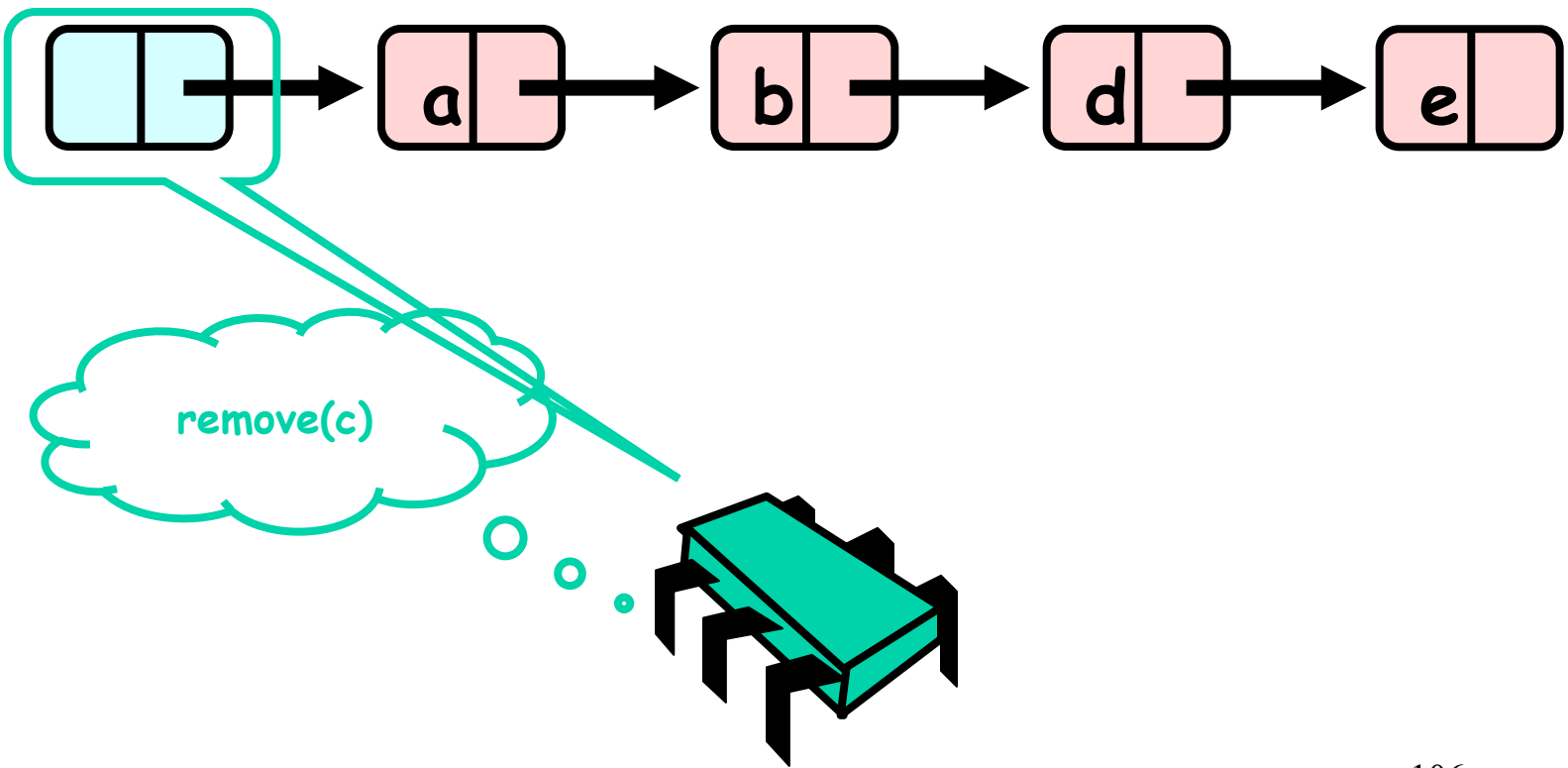
- If
  - Nodes b and c both locked
  - Node b still accessible
  - Node c still successor to b
- Then
  - Neither will be deleted
  - OK to delete and return true

Unsuccessful Remove

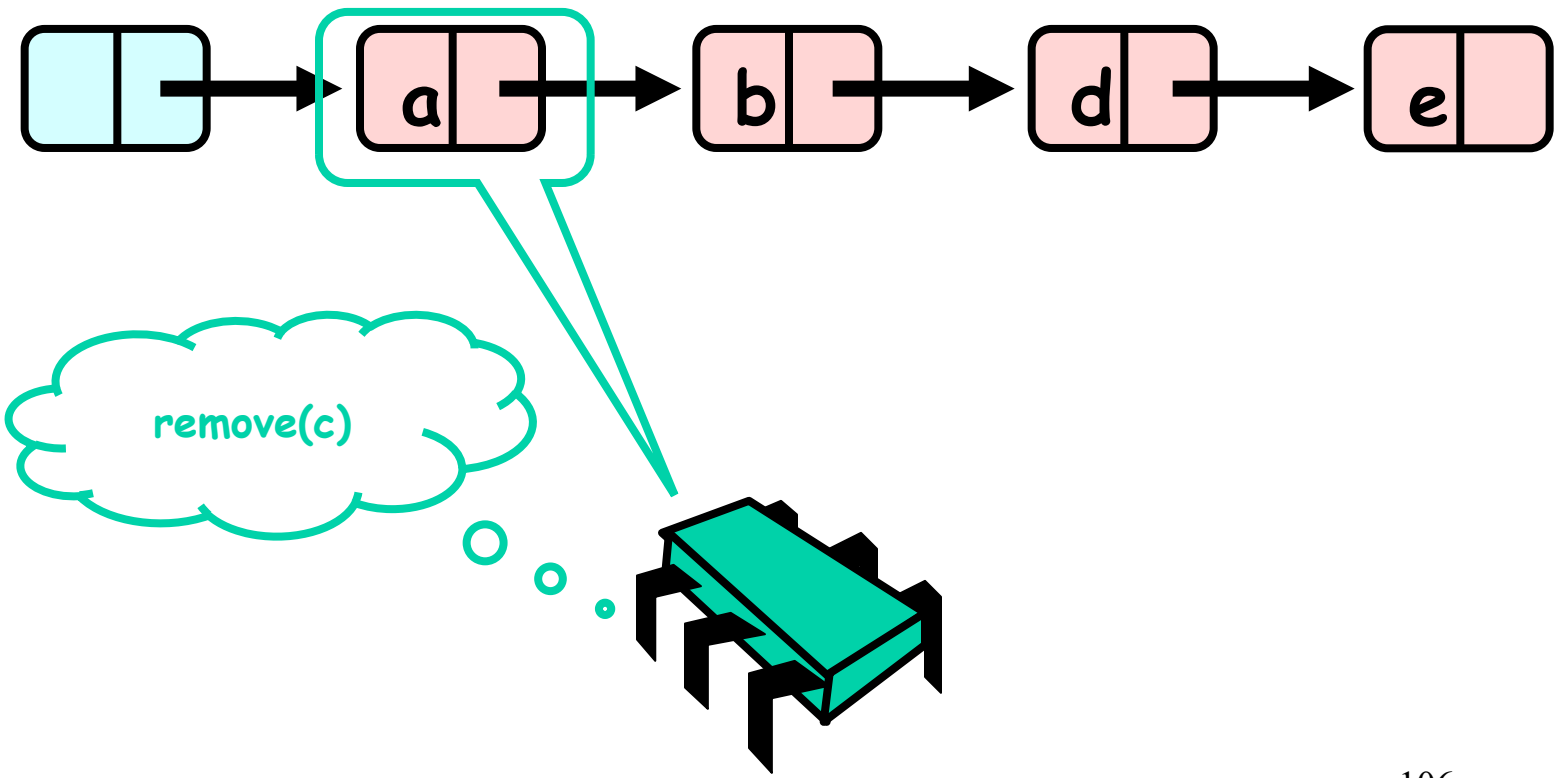




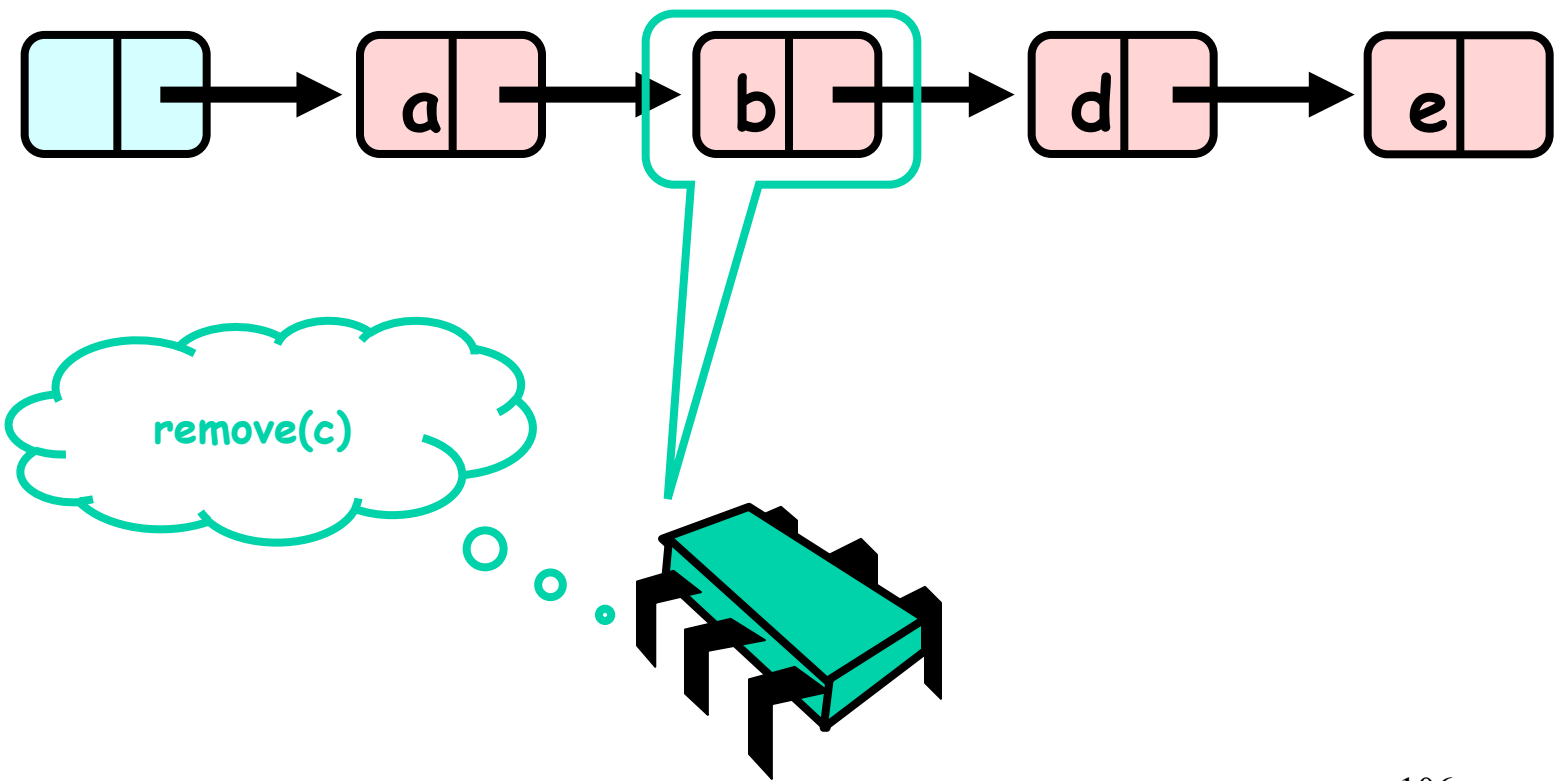
Unsuccessful Remove



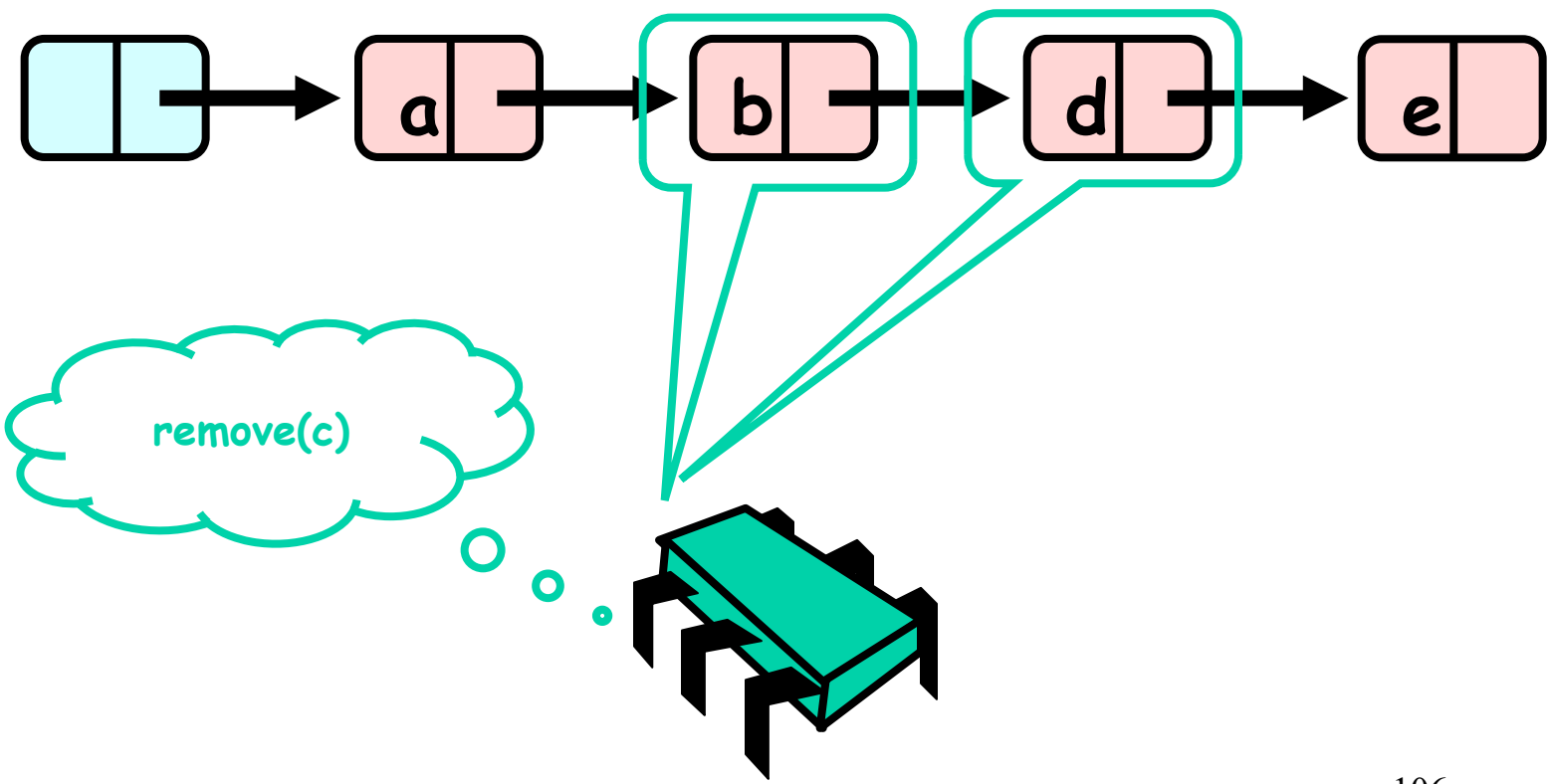
Unsuccessful Remove



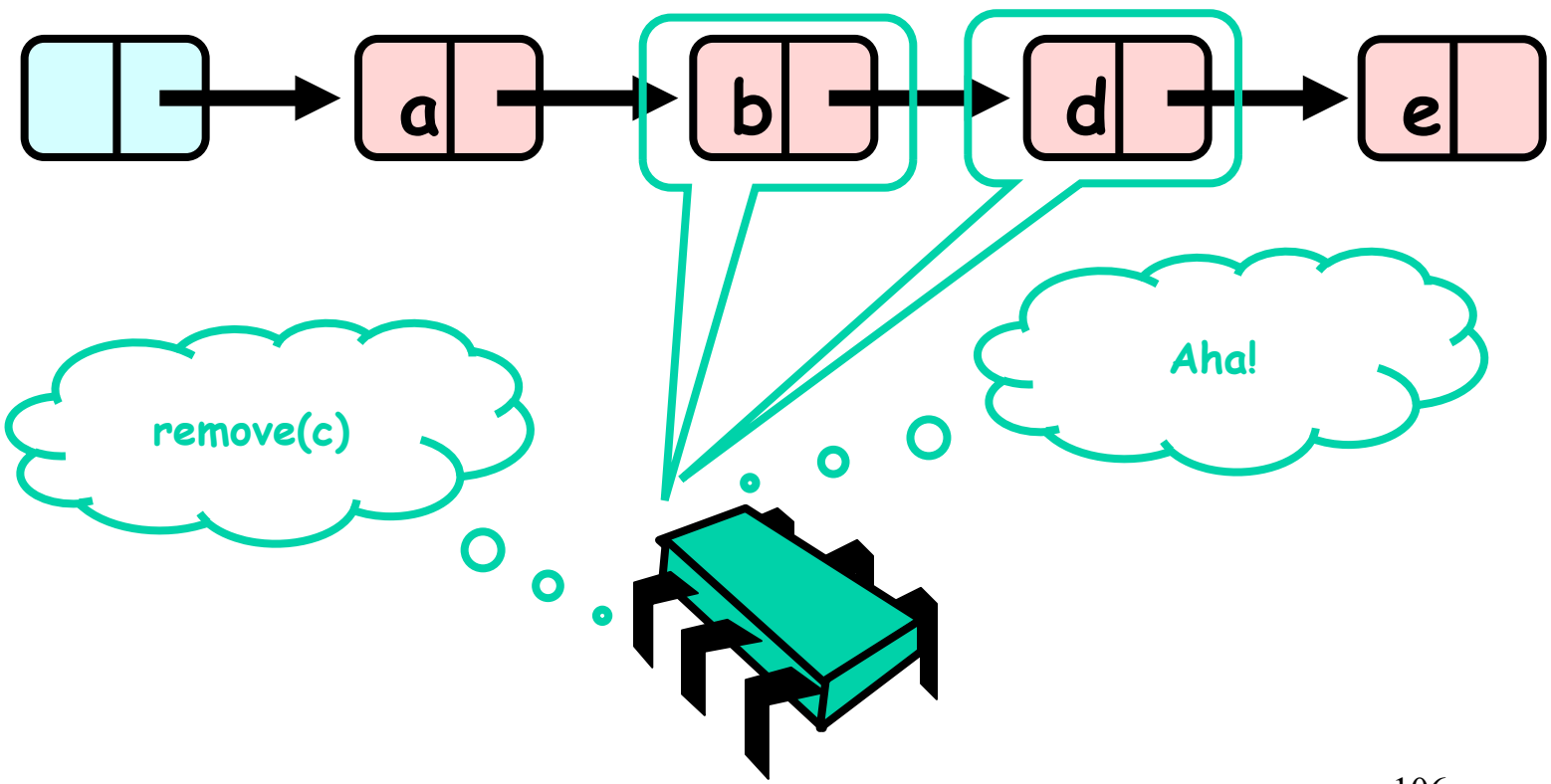
Unsuccessful Remove



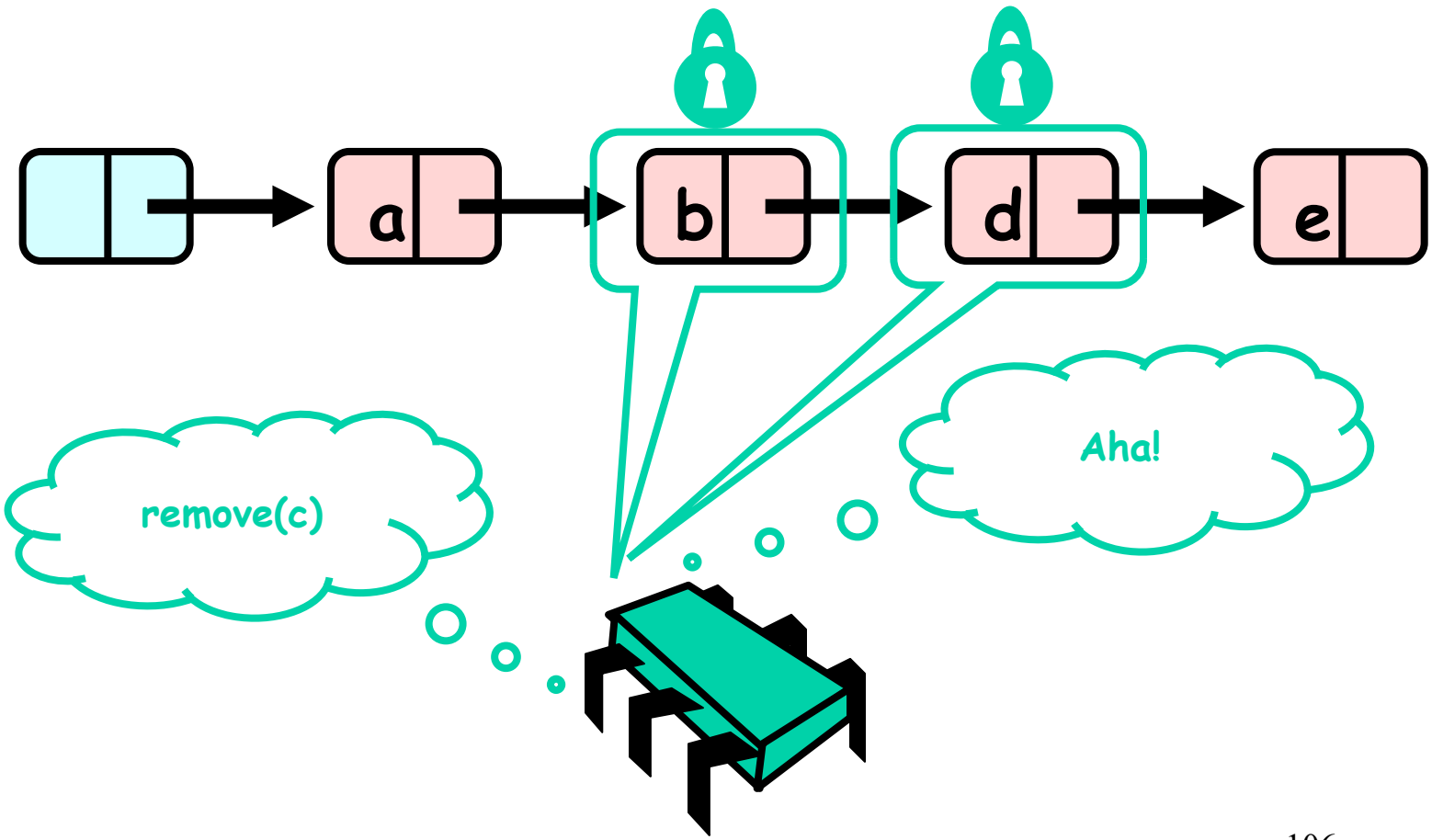
Unsuccessful Remove



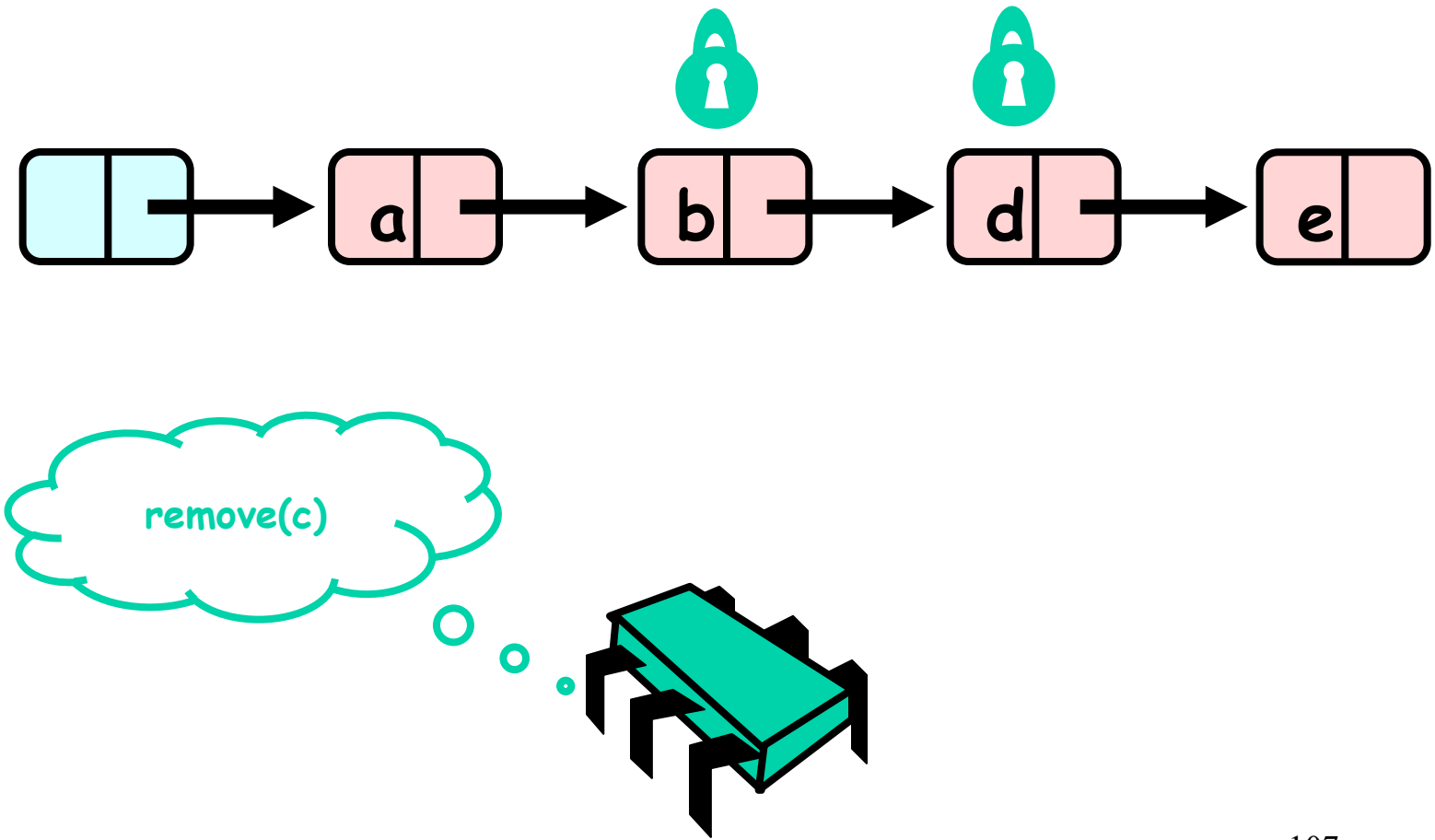
Unsuccessful Remove



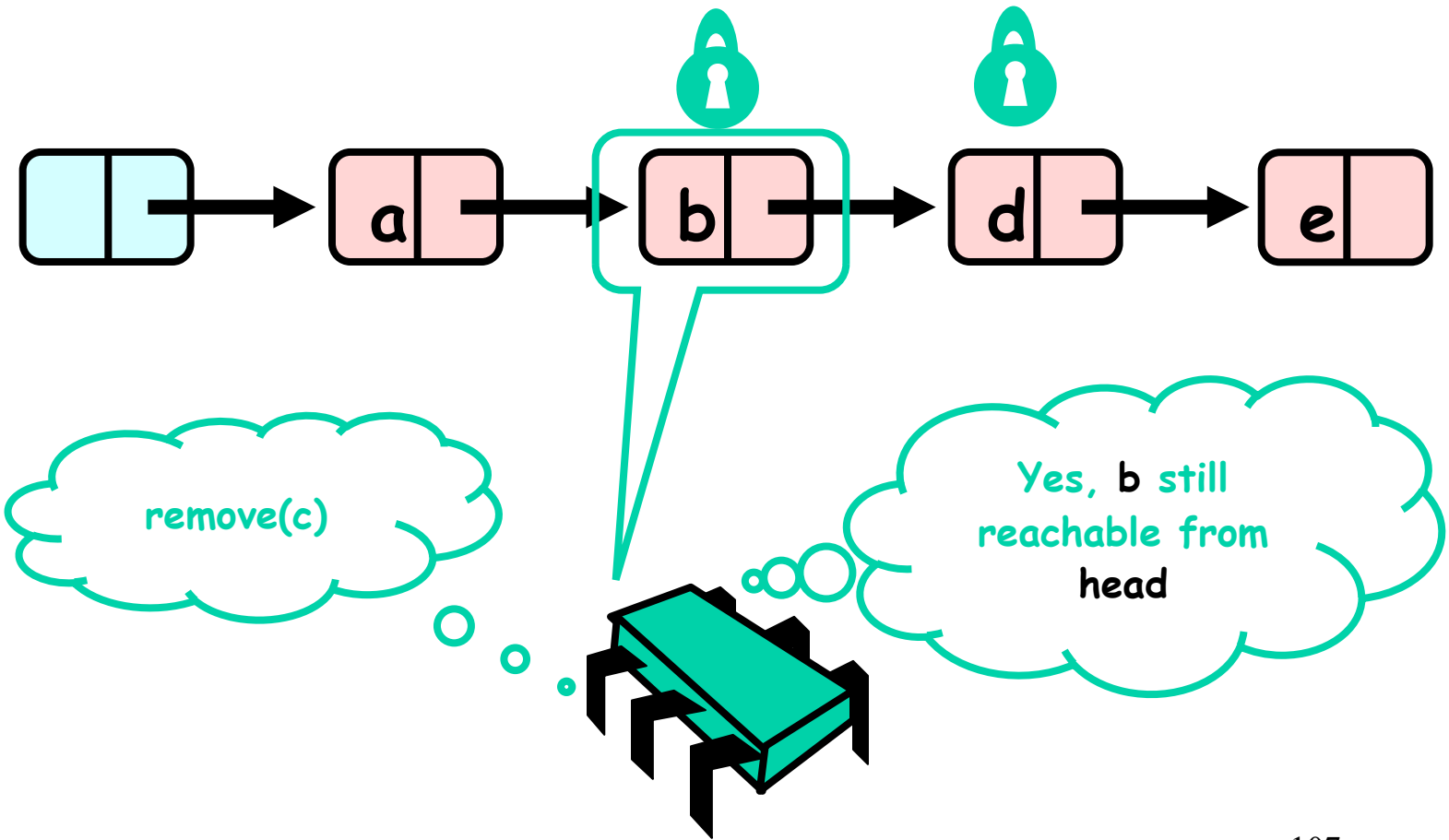
Unsuccessful Remove



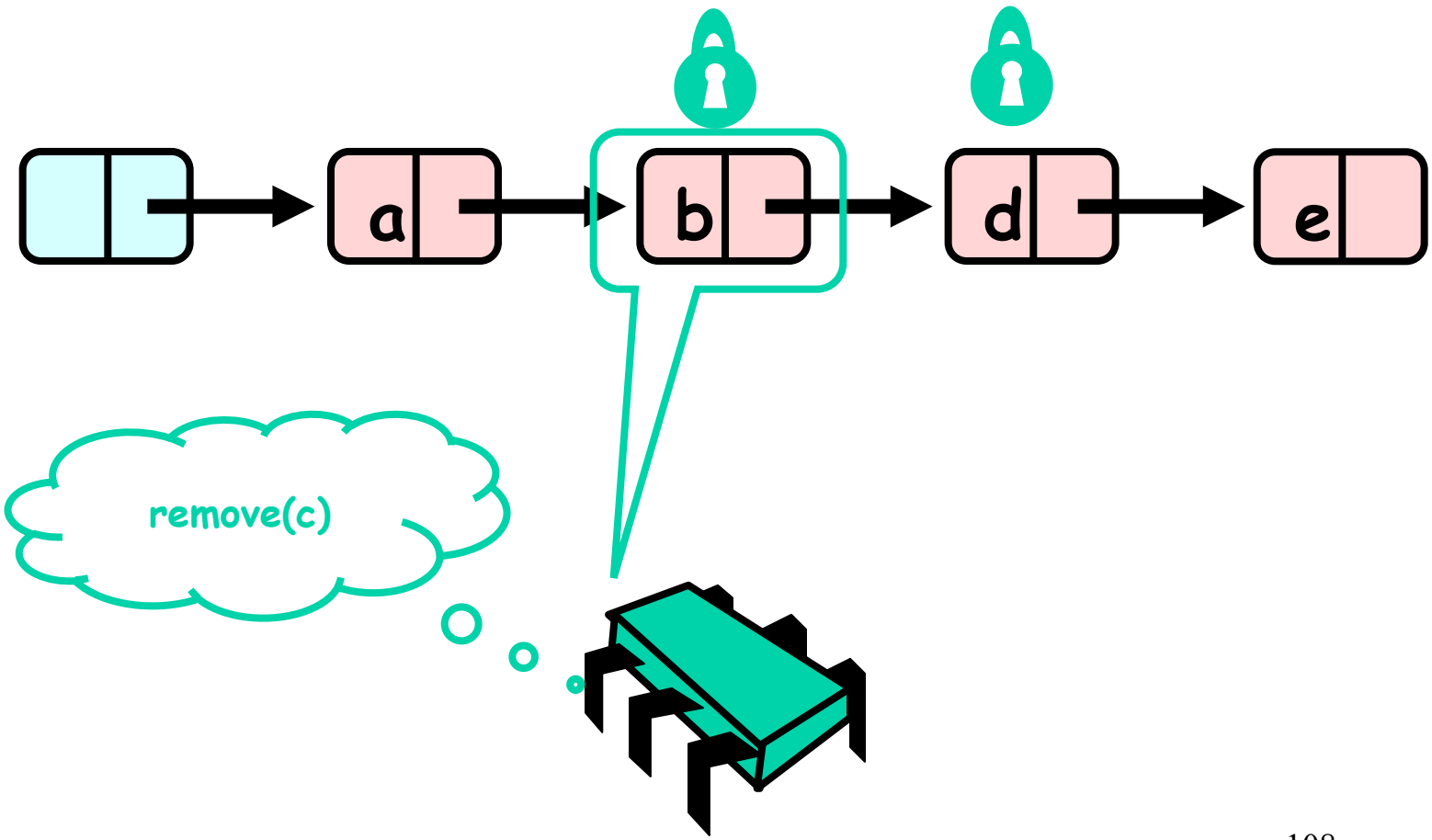
Validate (1)



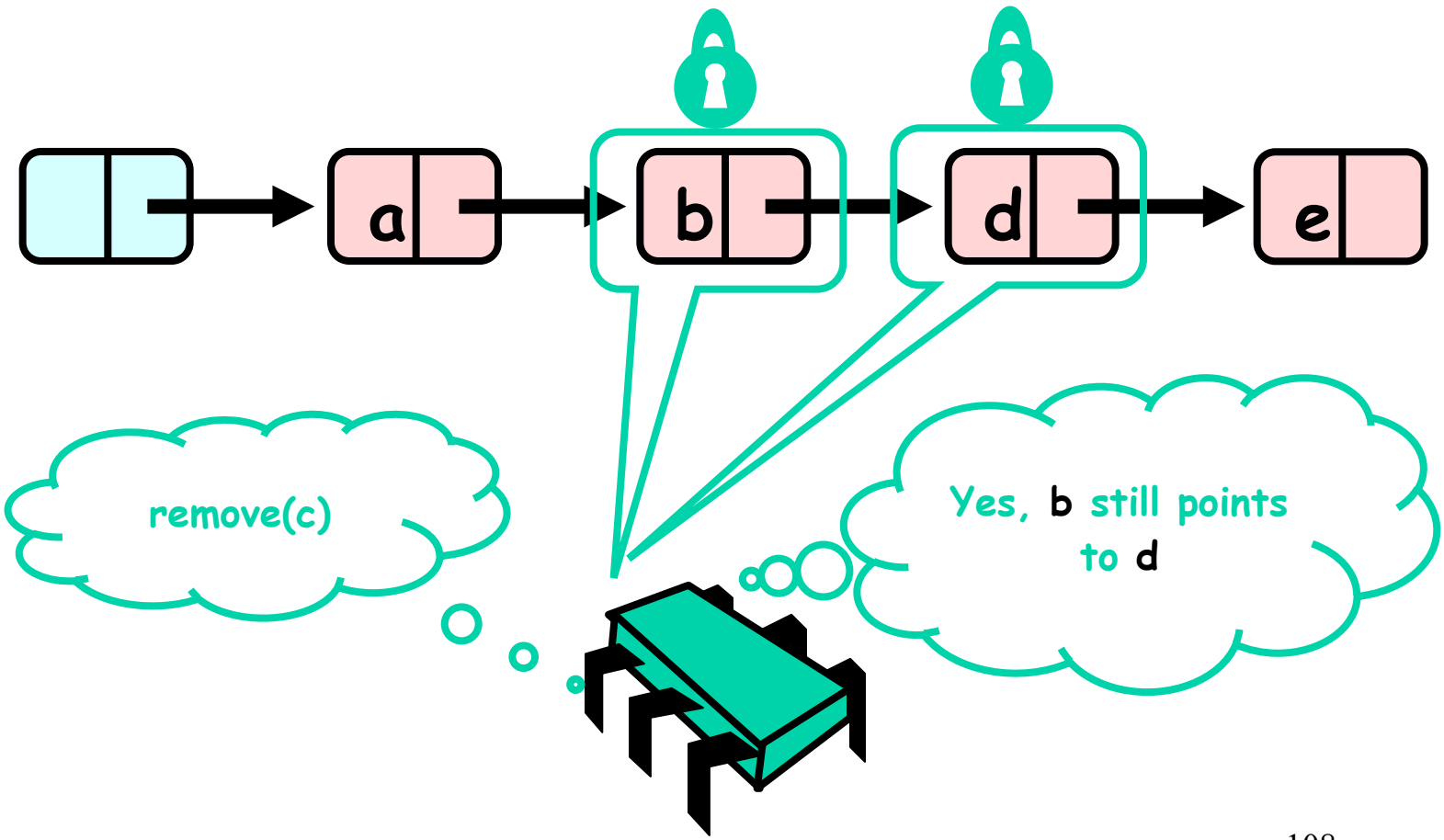
Validate (1)



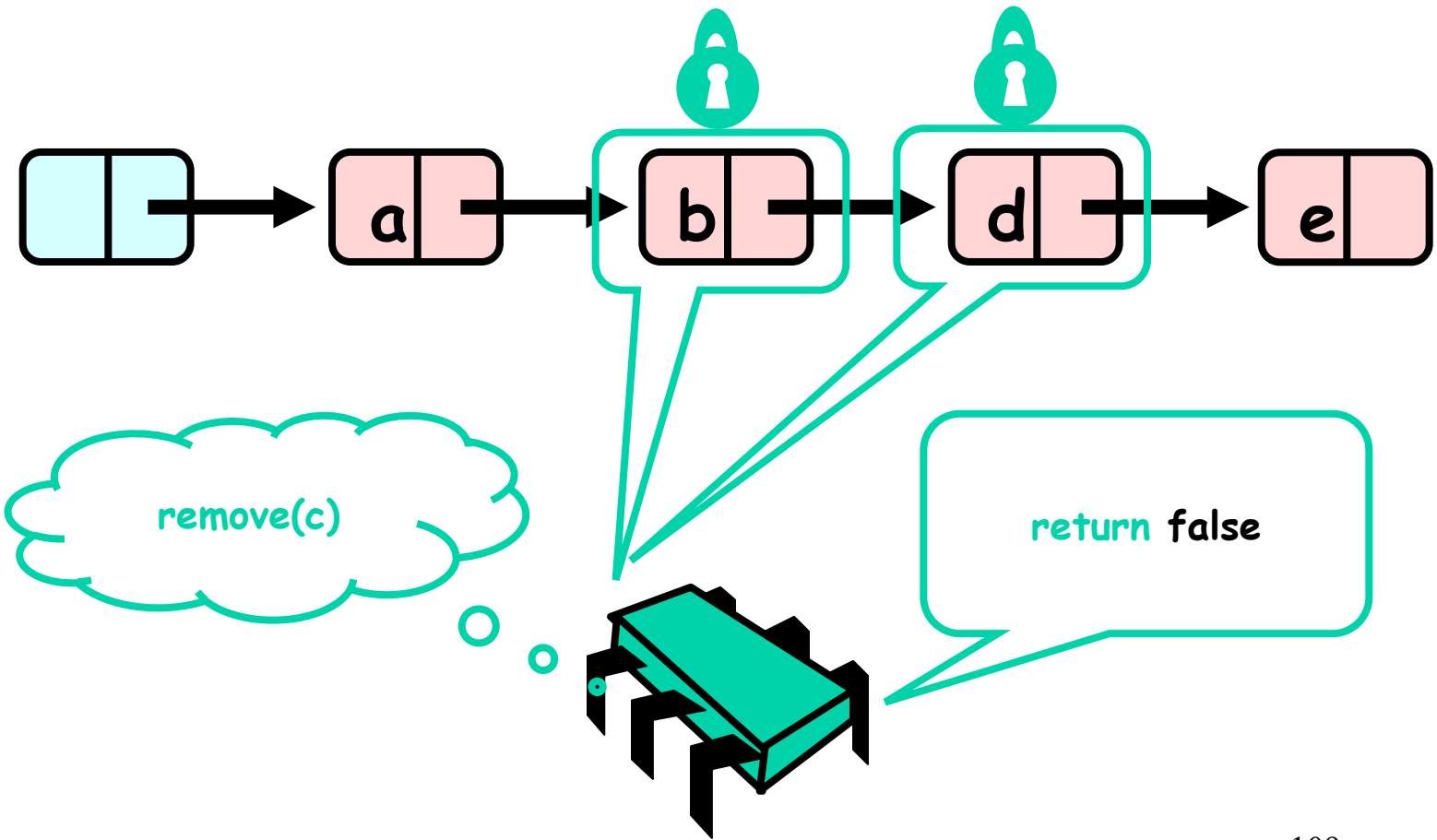
Validate (2)



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

```
private boolean
 validate(Node pred, Node curry) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Remove: searching

```
public boolean remove(Item item) {
 int key = item.hashCode();
 retry: while (true) {
   Node pred = this.head;
   Node curr = pred.next;
   while (curr.key <= key) {
    if (item == curr.item)
      break;
    pred = curr;
    curr = curr.next;
   } …
```

On Exit from 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

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

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

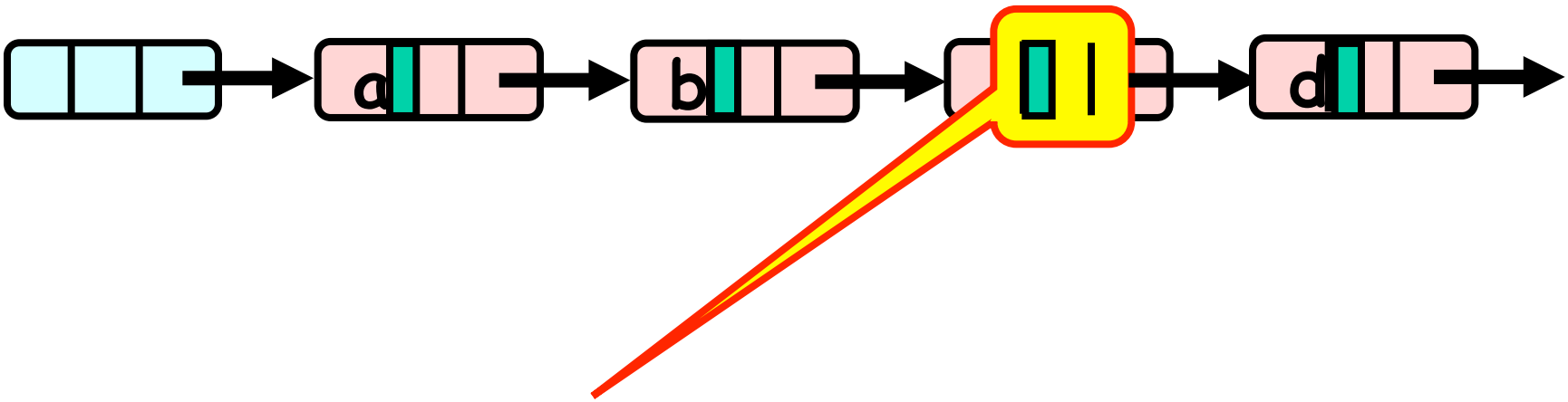
- Like optimistic, except
  - Scan once
  - **contains(x)** never locks …
- Key insight
  - Removing nodes causes trouble
  - Do it "lazily"

Lazy List

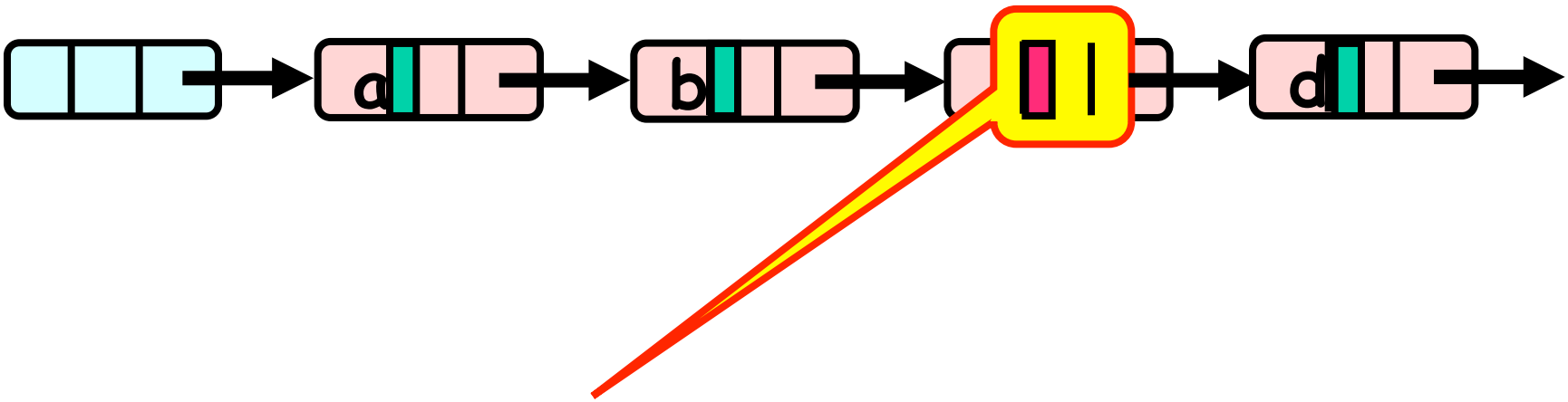
- **remove()**
  - Scans list (as before)
  - Locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

Lazy Removal

Lazy Removal

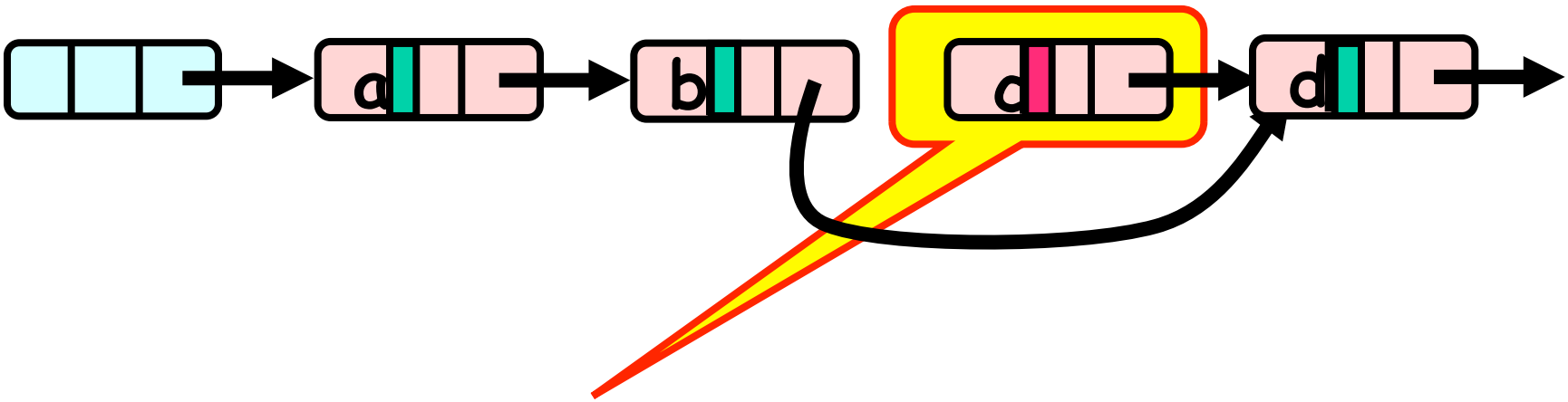


Lazy Removal



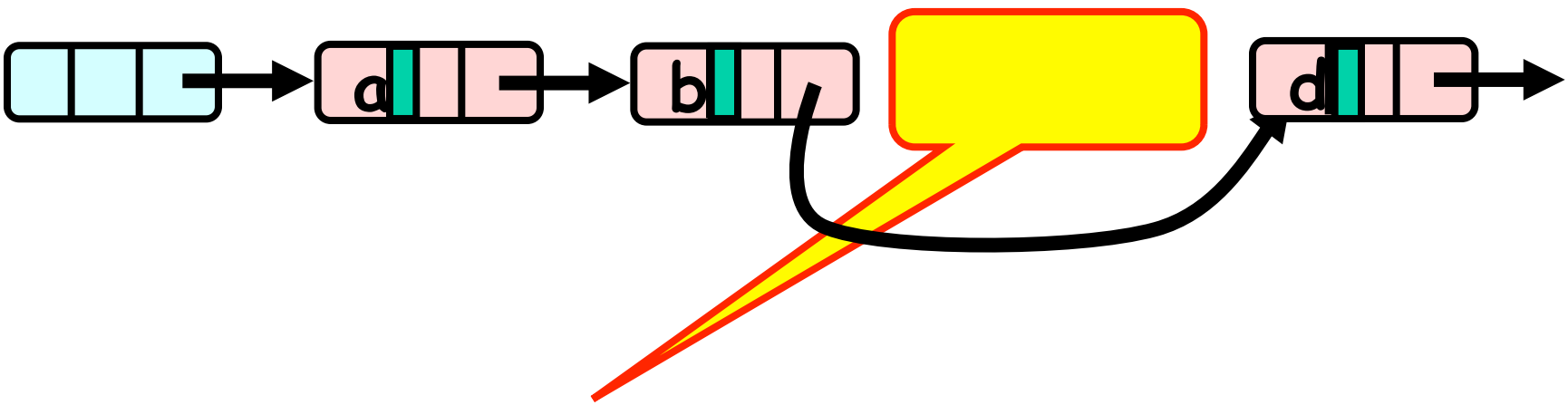
Logically deleted

Lazy Removal



Physically deleted

Lazy Removal



Physically deleted

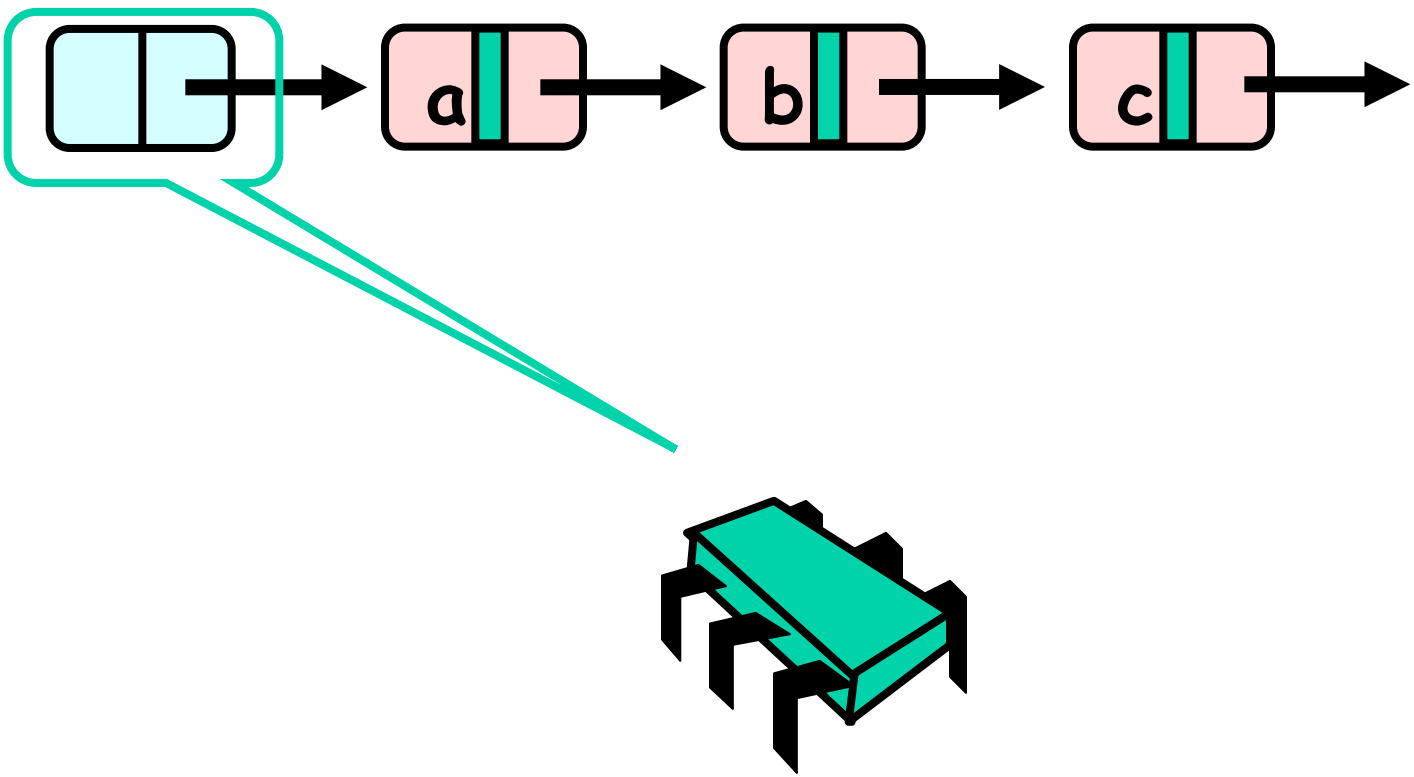
Lazy List

- 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.

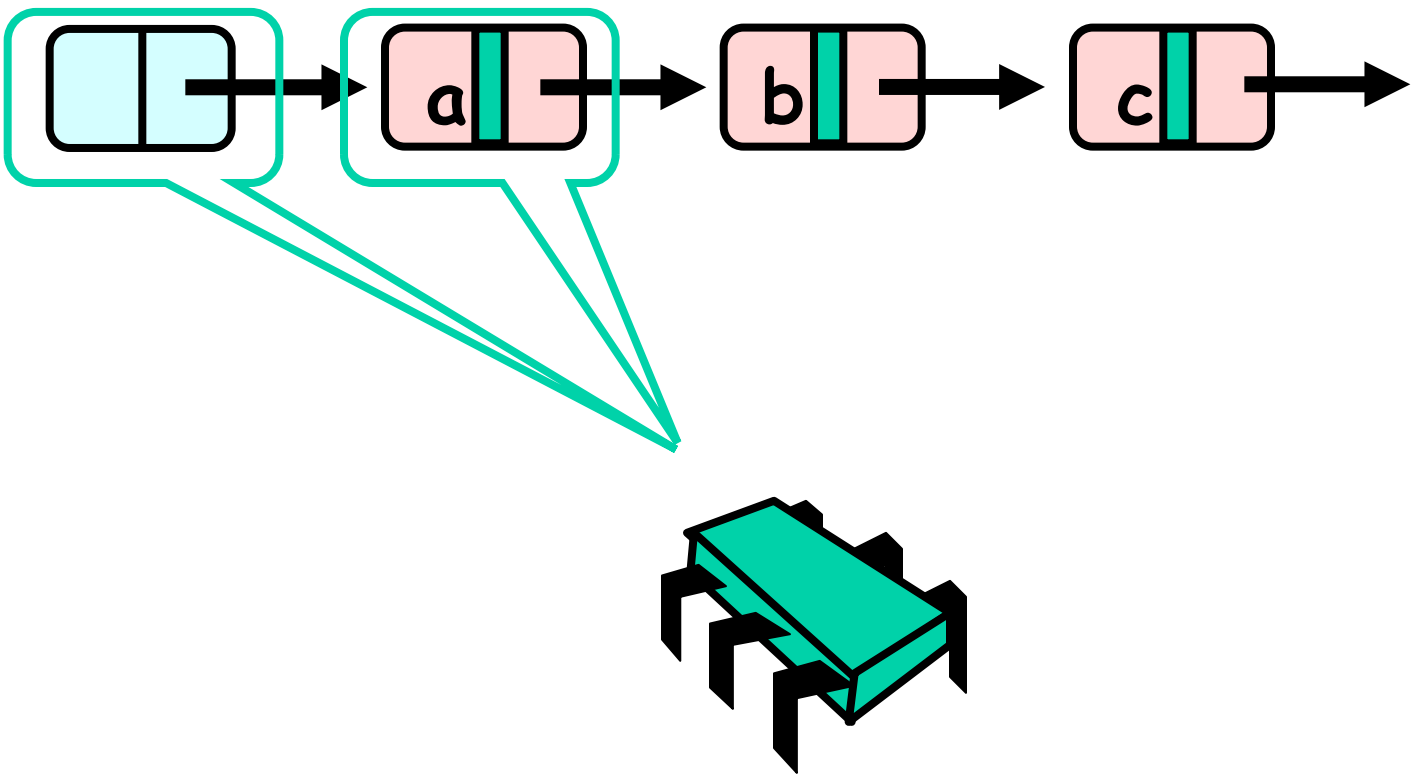
Validation

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

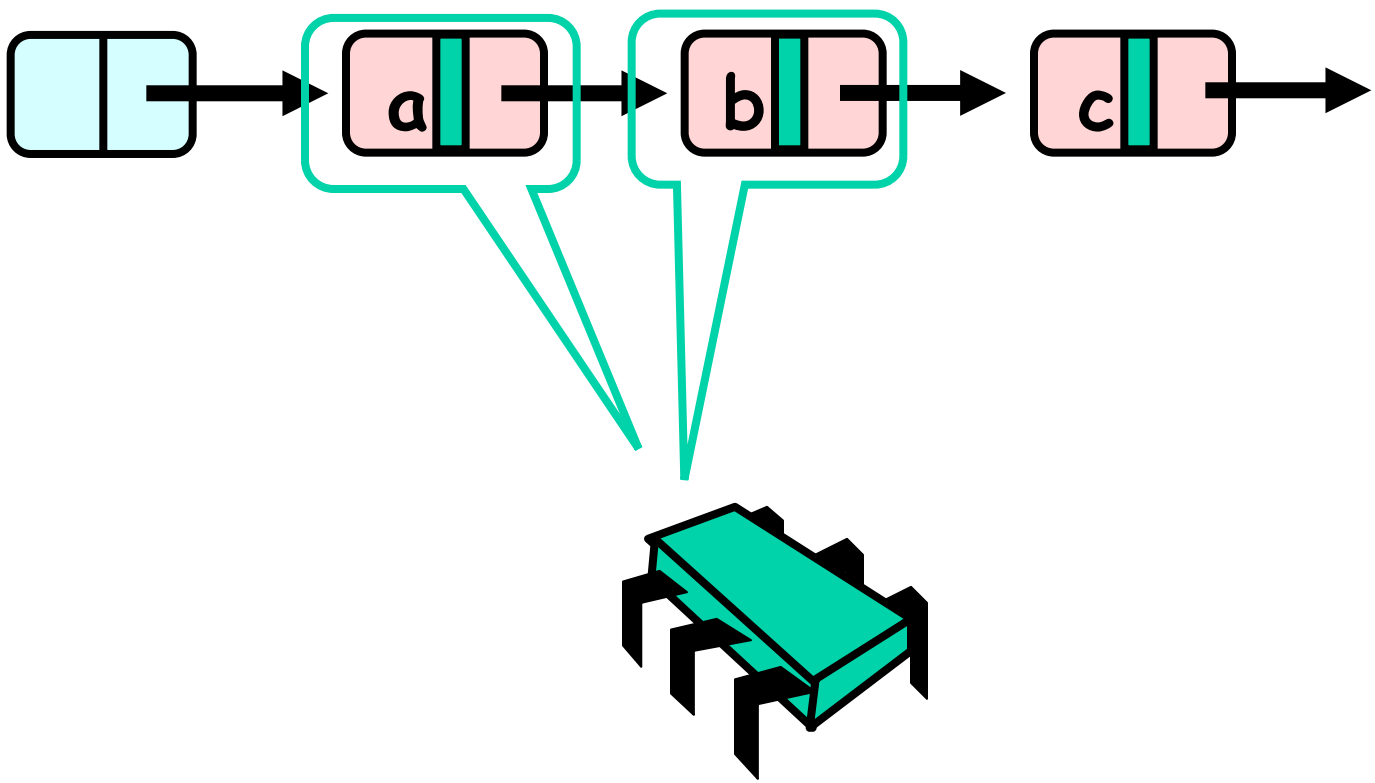
Business as Usual



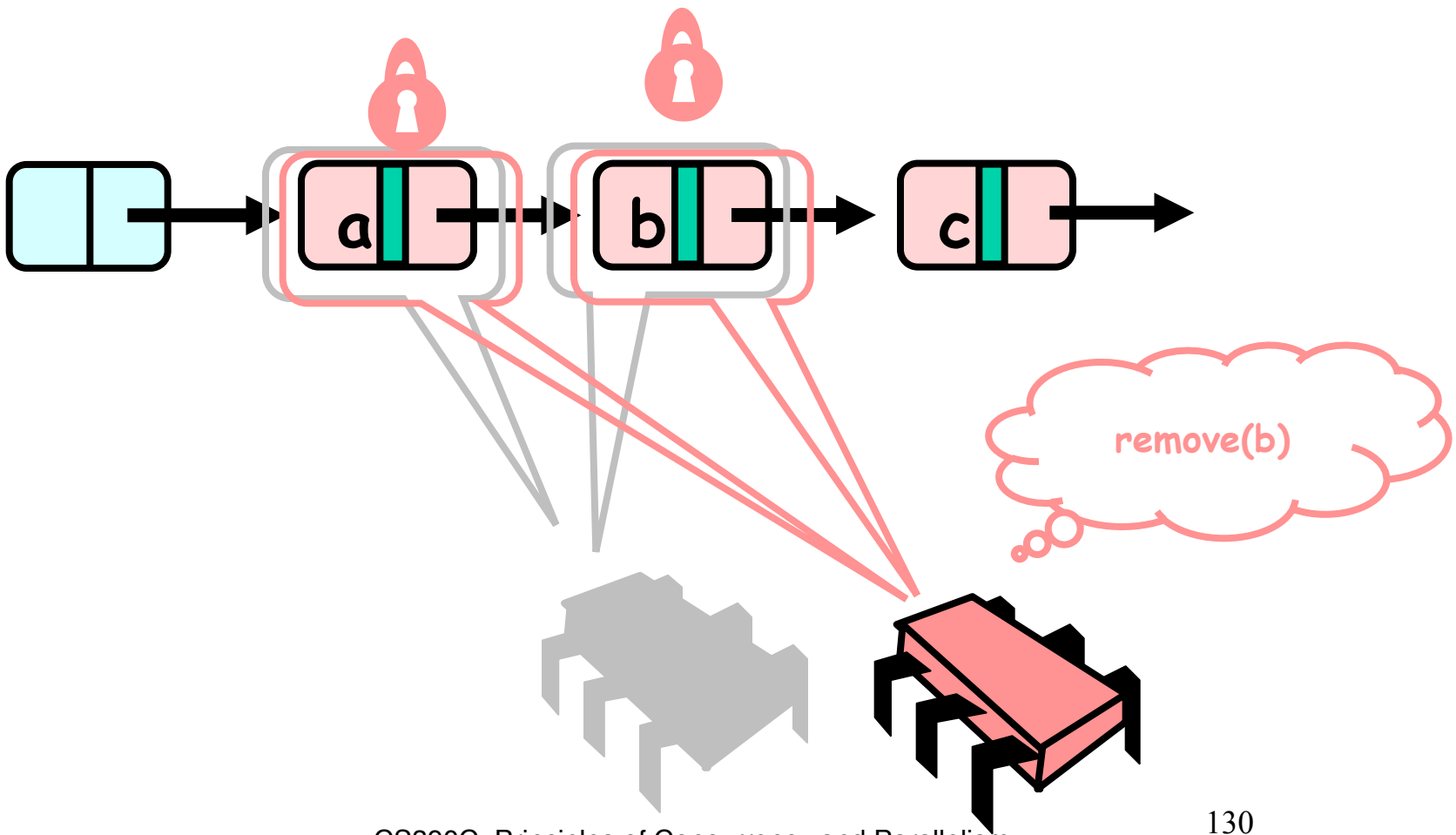
Business as Usual



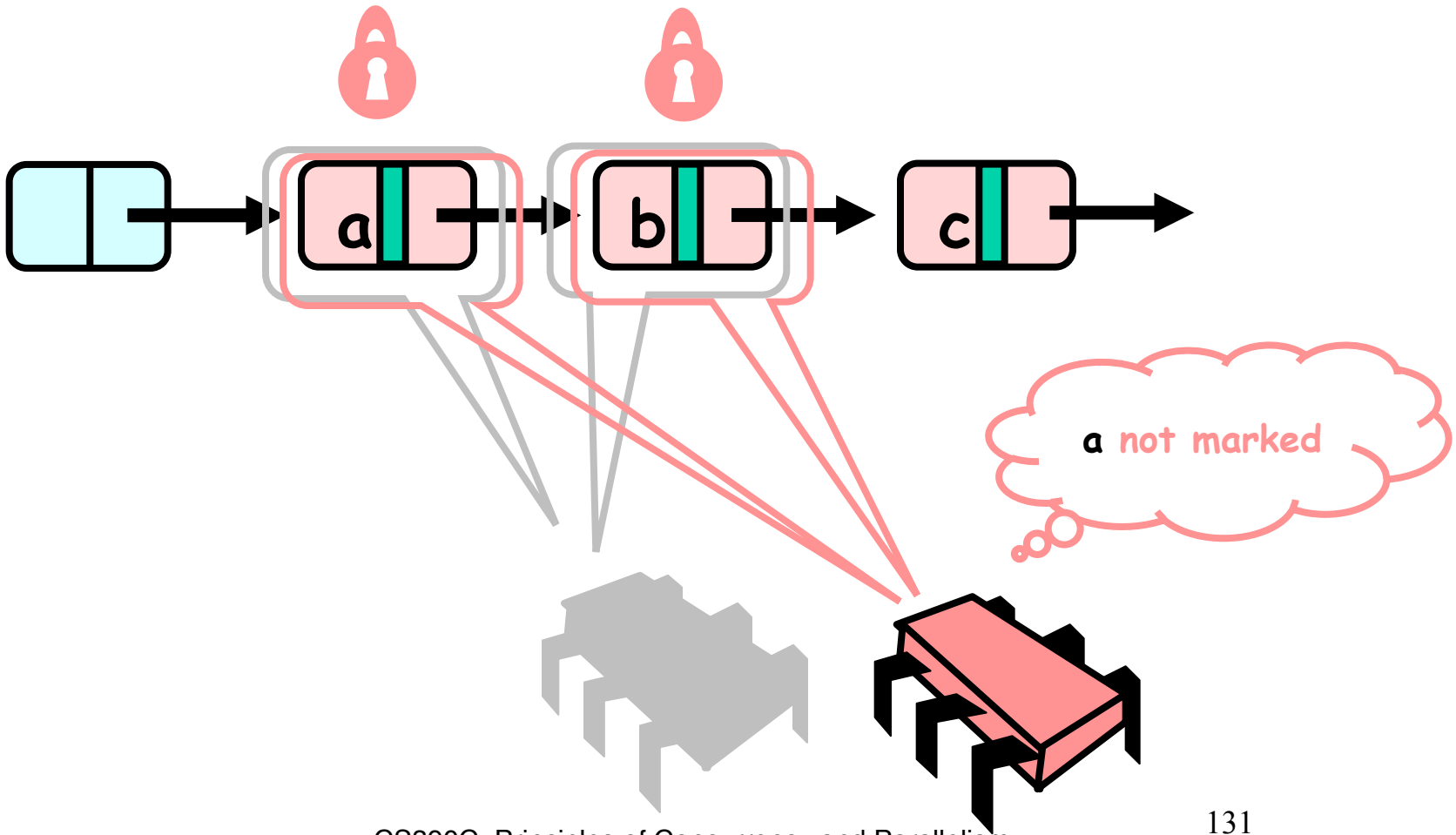
Business as Usual



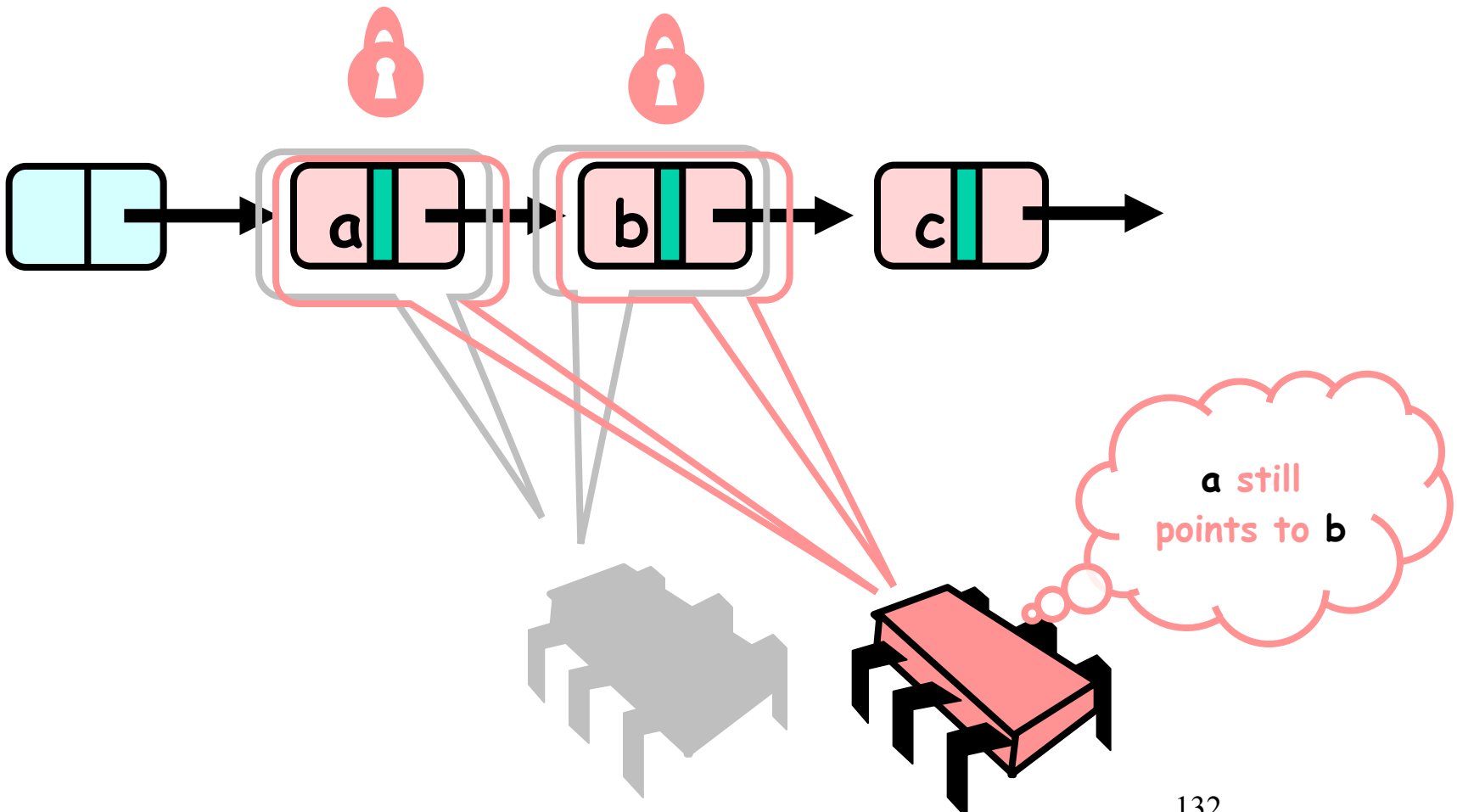
Business as Usual



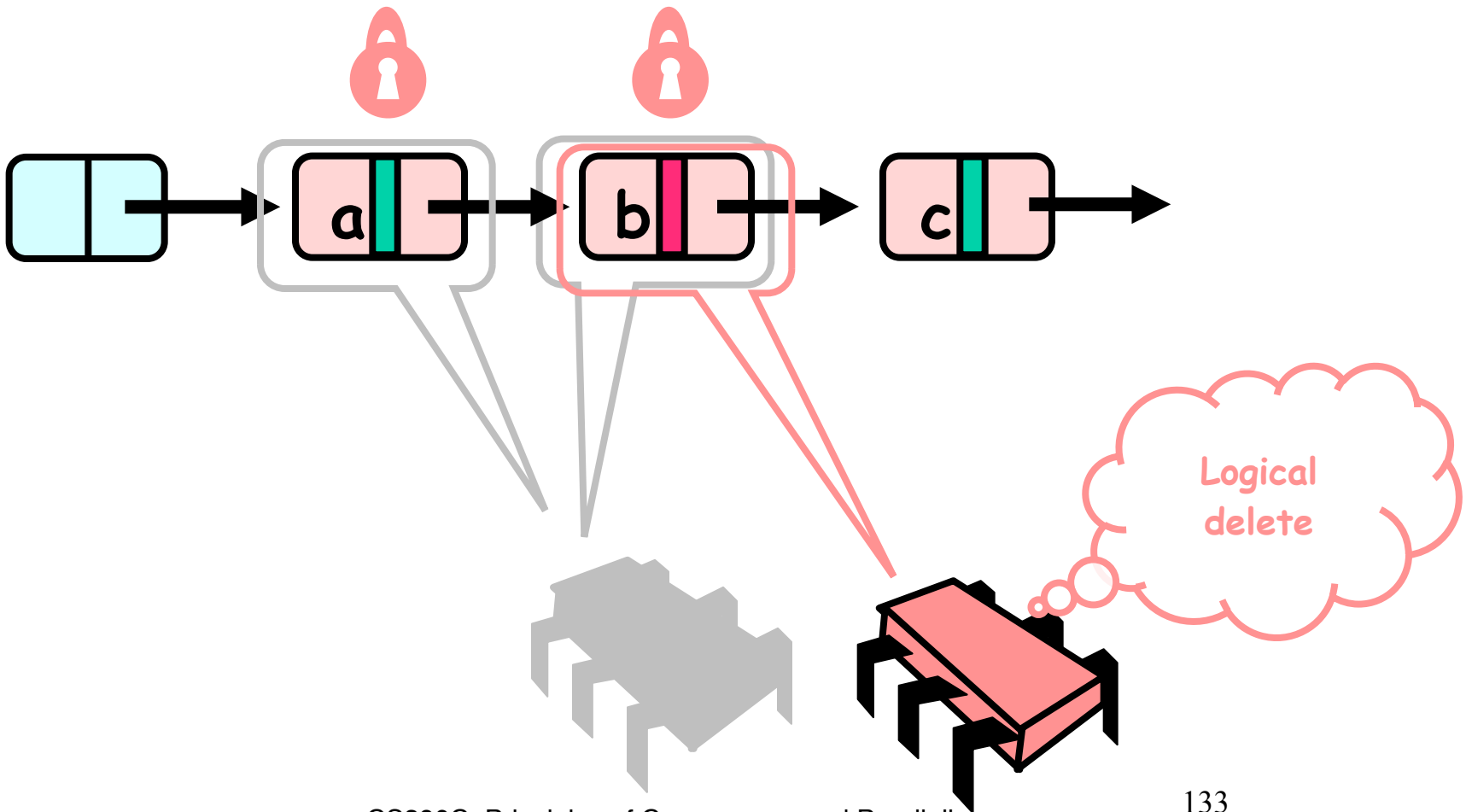
Business as Usual



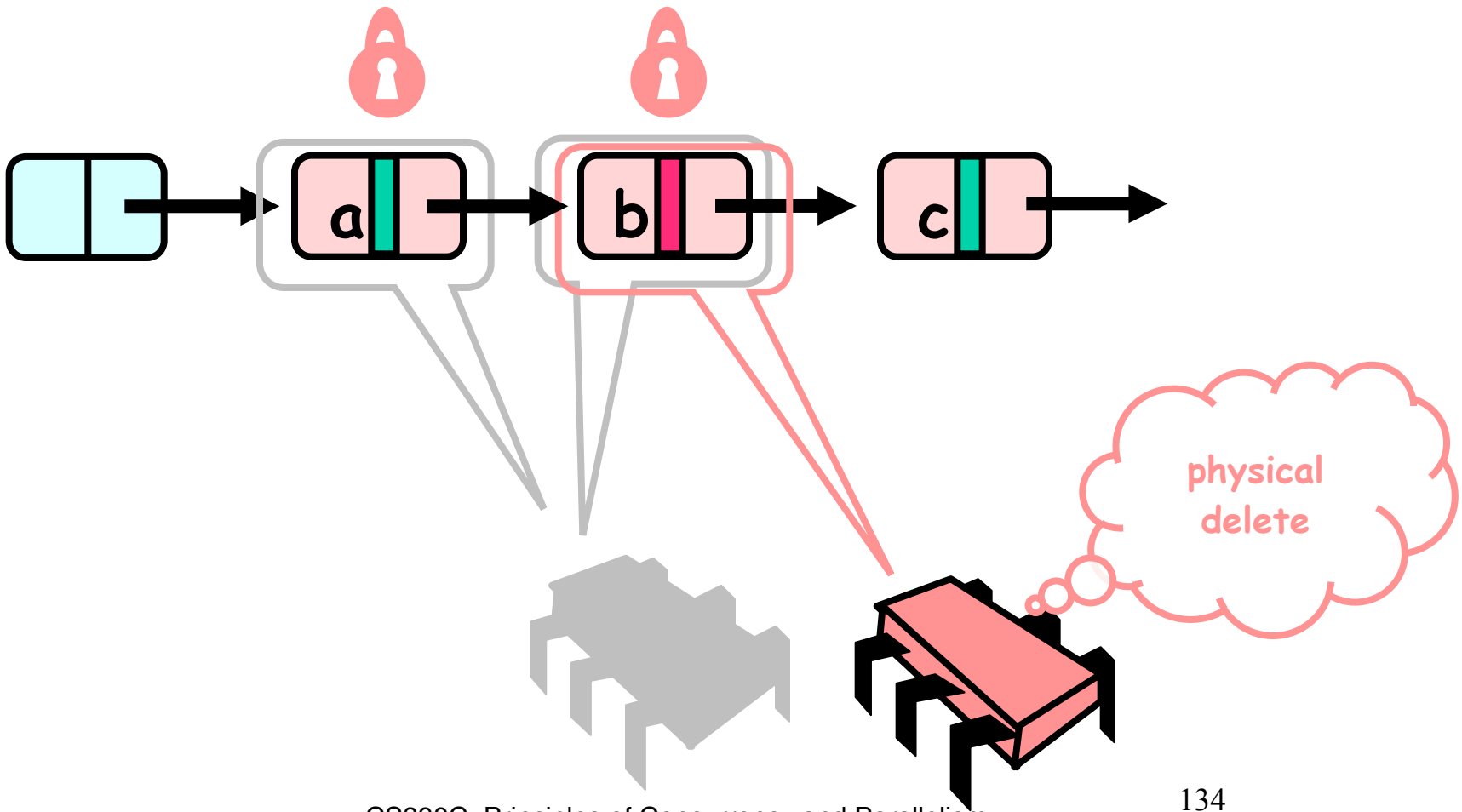
Business as Usual



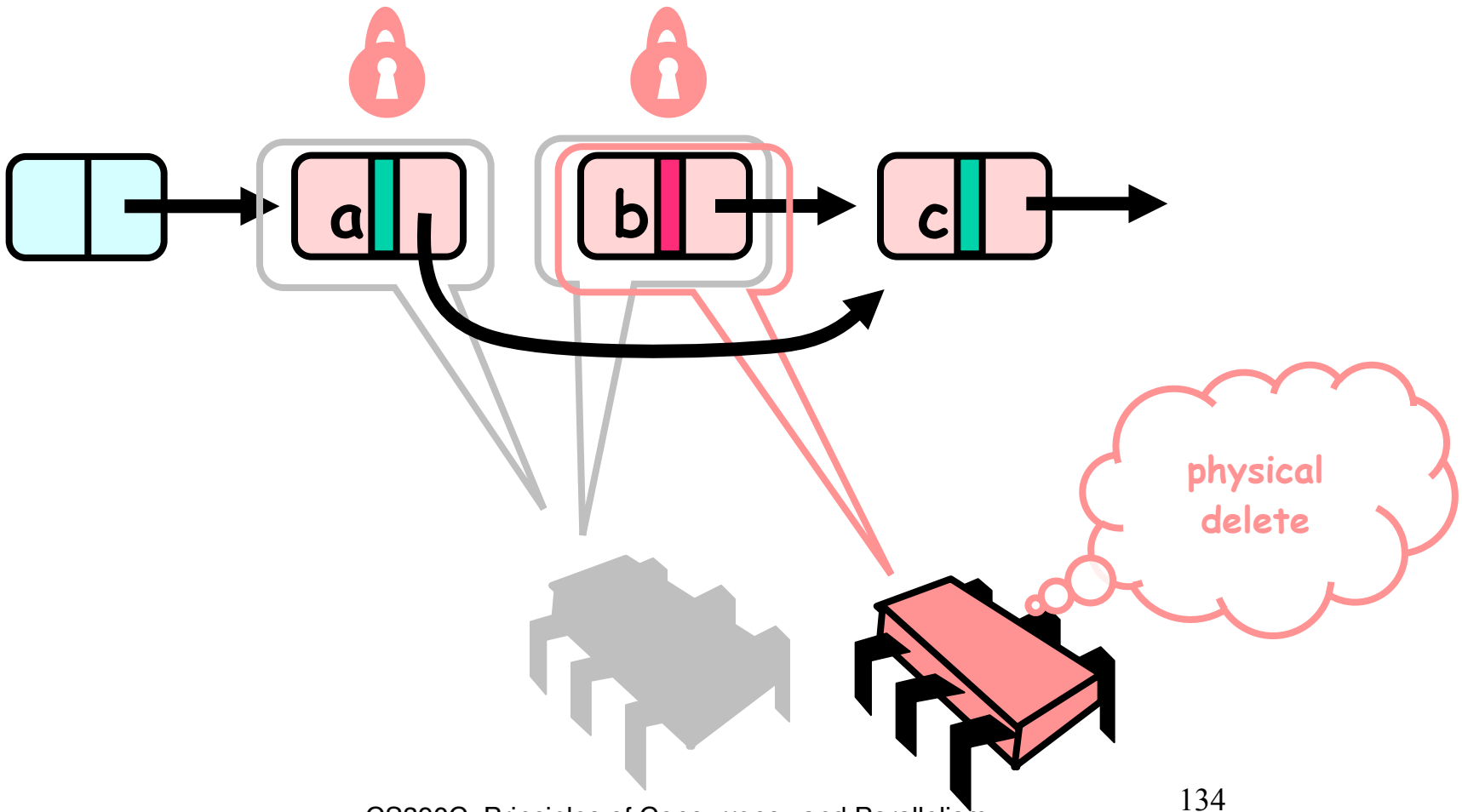
Business as Usual



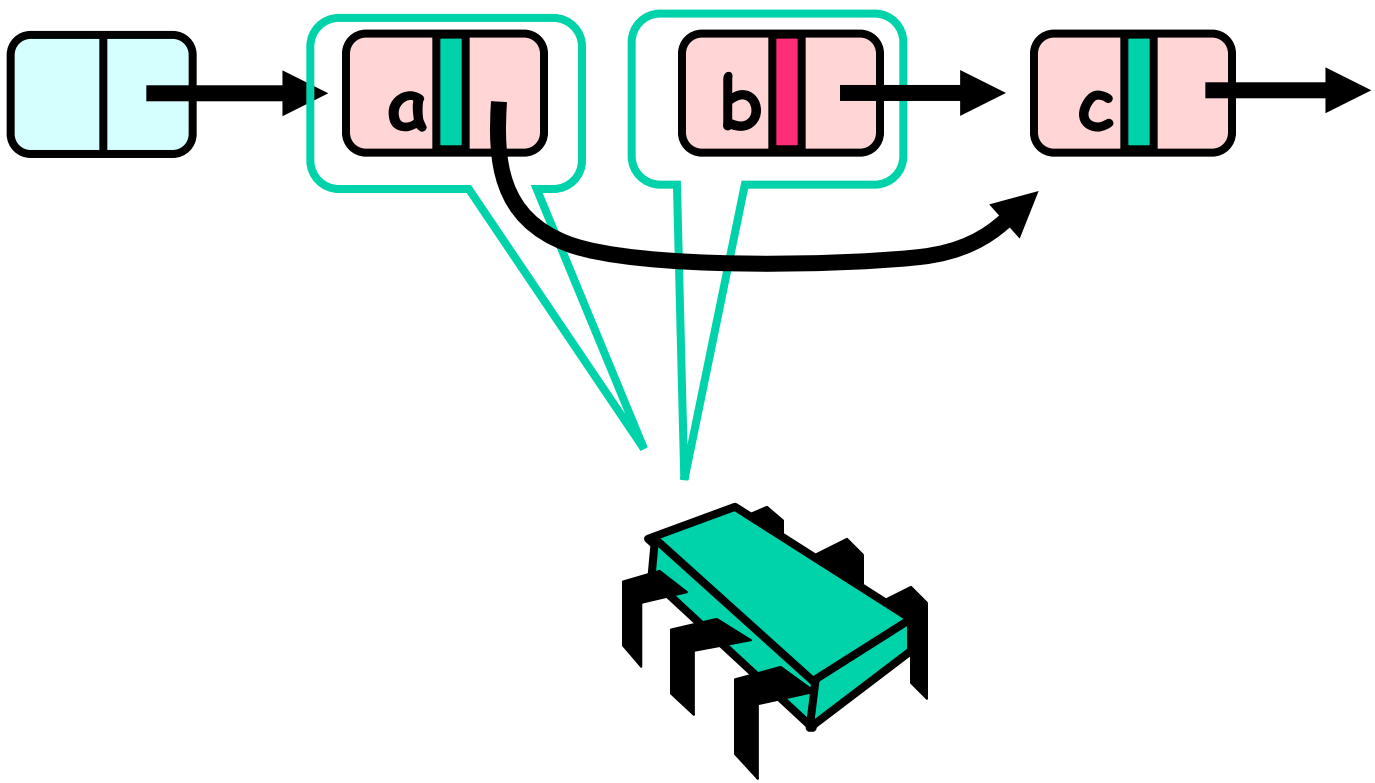
Business as Usual



Business as Usual



Business as Usual

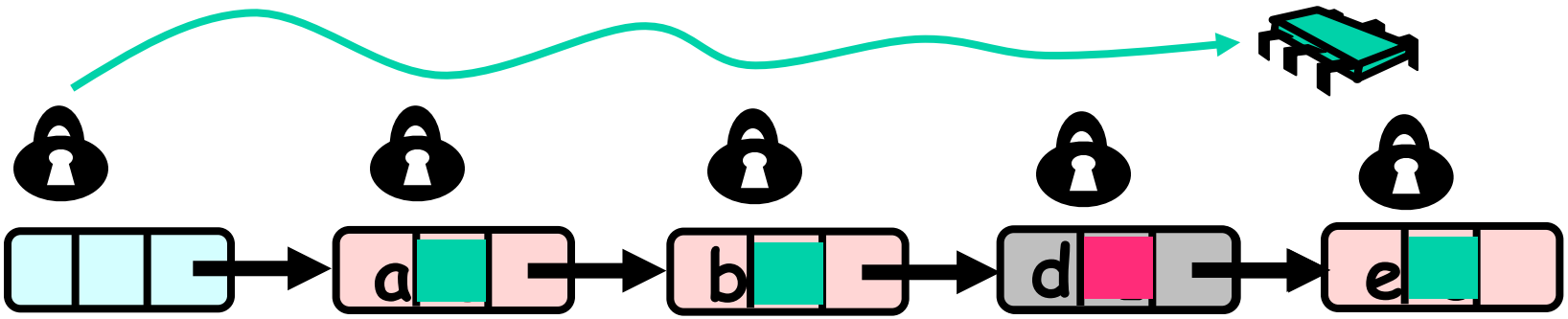


Summary: Wait-free Contains

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()

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 reachable from head
- and if not yet traversed it is reachable from pred

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 do 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

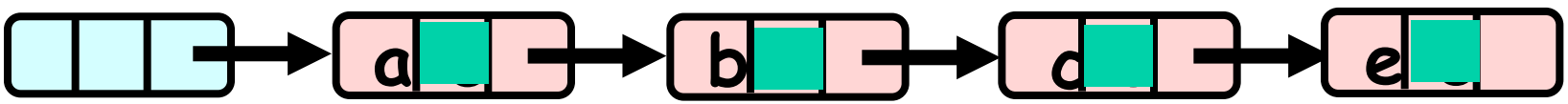
Reminder: Lock-Free Data Structures

- 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?

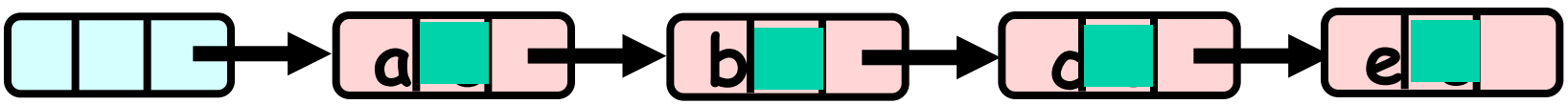
Lock-free Lists



Use CAS to verify pointer is correct

Lock-free Lists

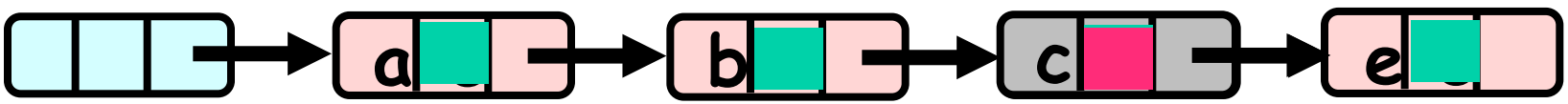
Logical Removal



Use CAS to verify pointer is correct

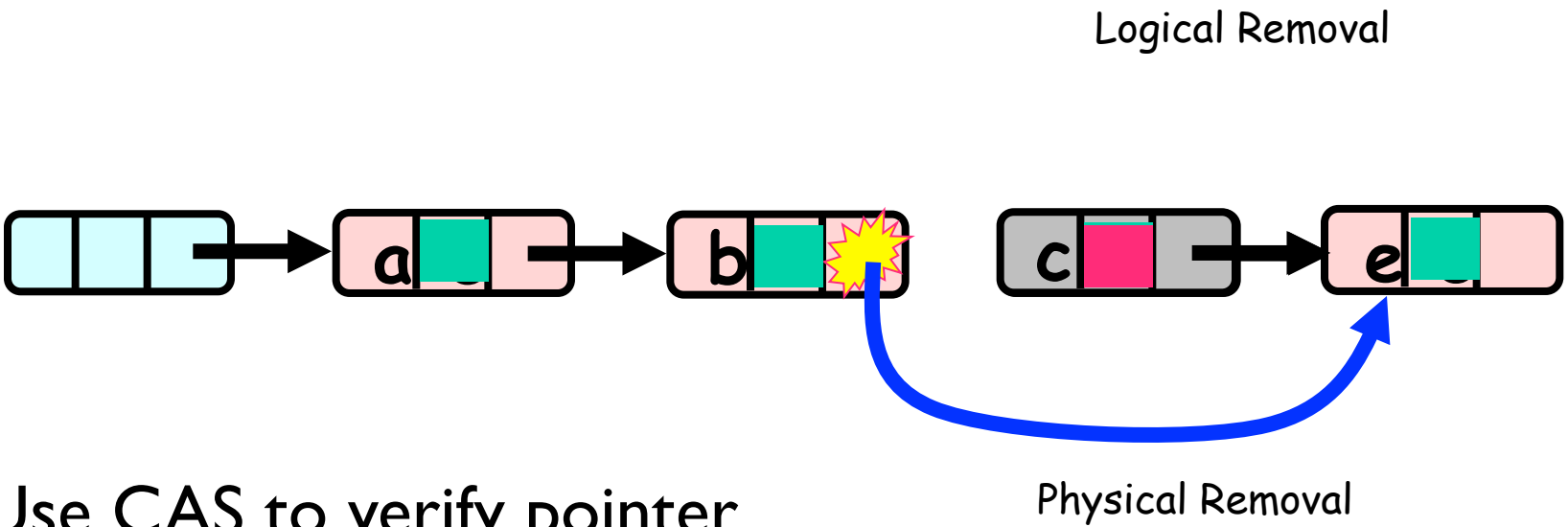
Lock-free Lists

Logical Removal



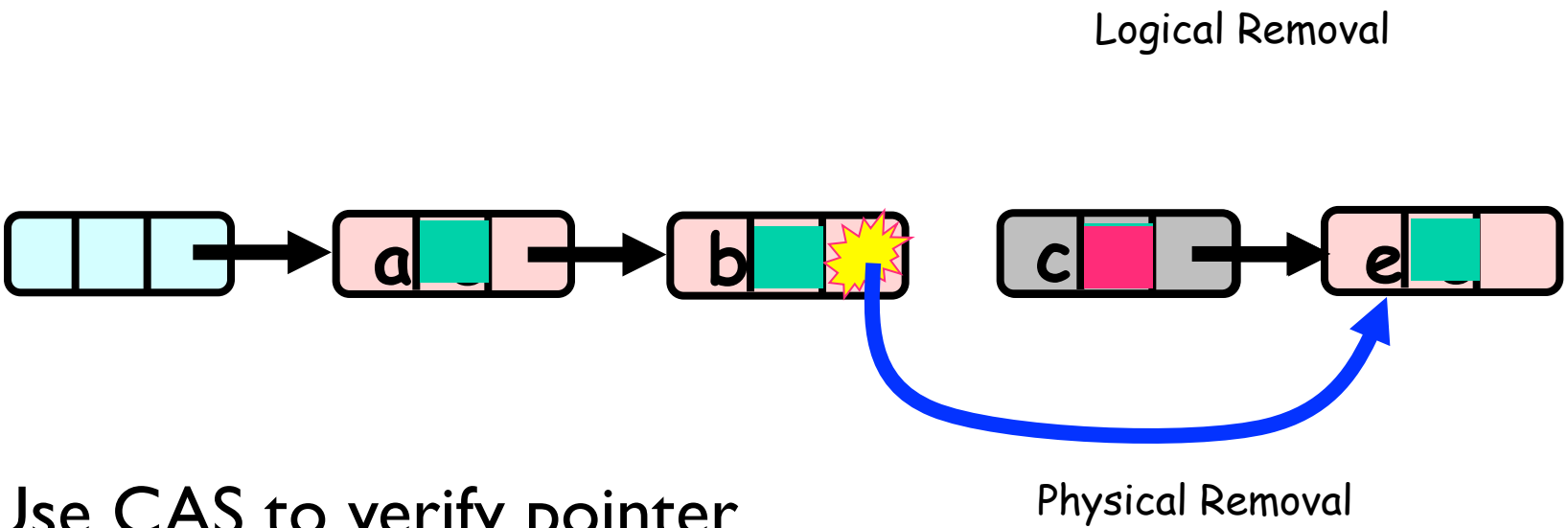
Use CAS to verify pointer is correct

Lock-free Lists



Use CAS to verify pointer is correct

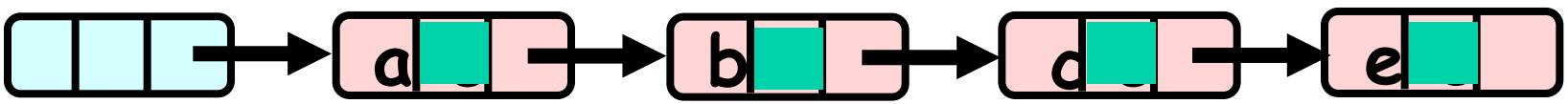
Lock-free Lists



Use CAS to verify pointer is correct

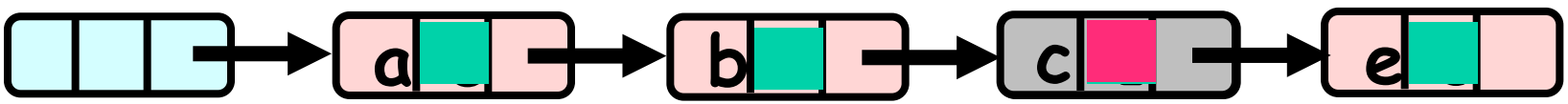
Not enough!

Problem…

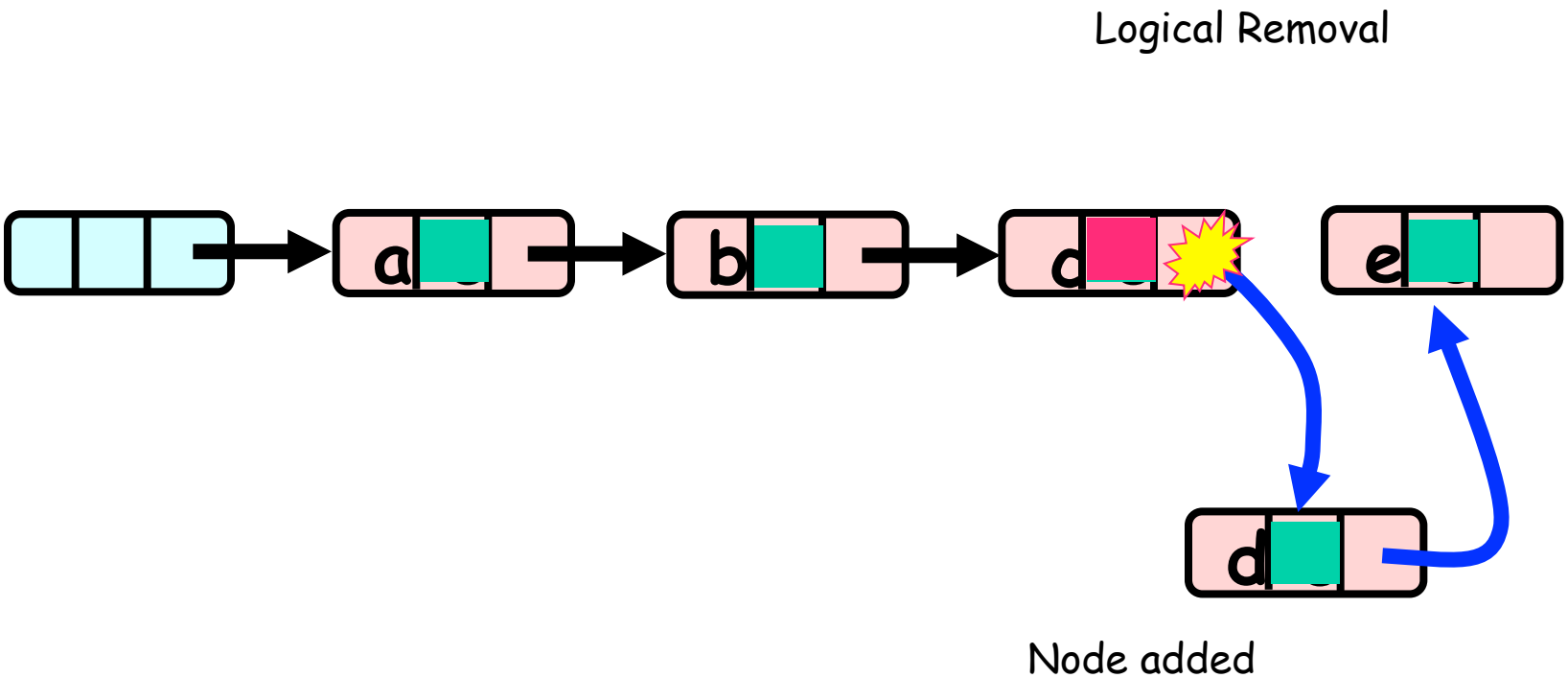


Problem…

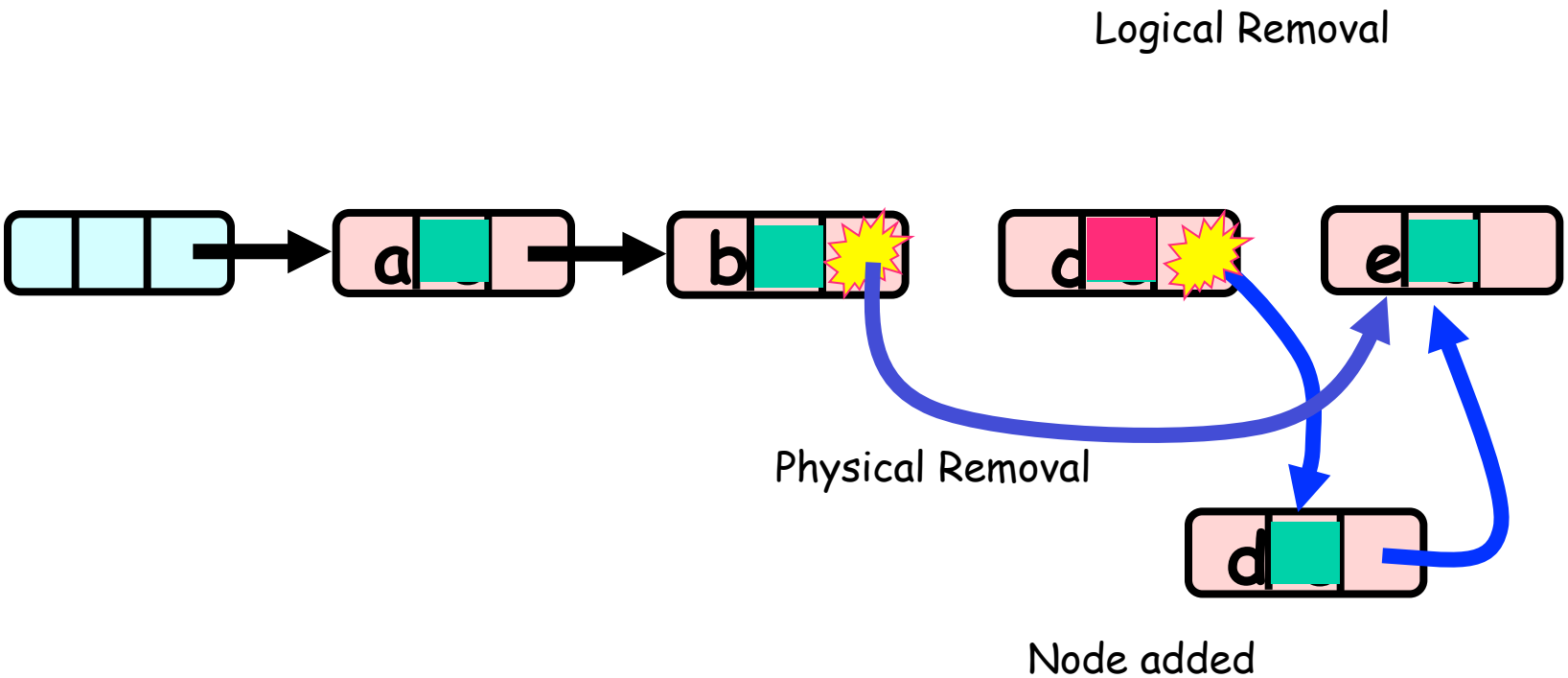
Logical Removal



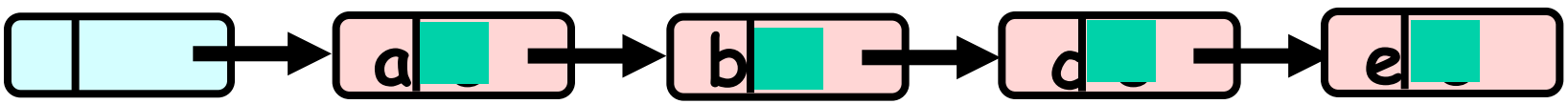
Problem…



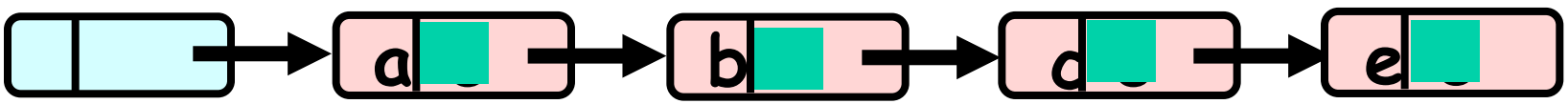
Problem…



The Solution: Combine Bit and Pointer

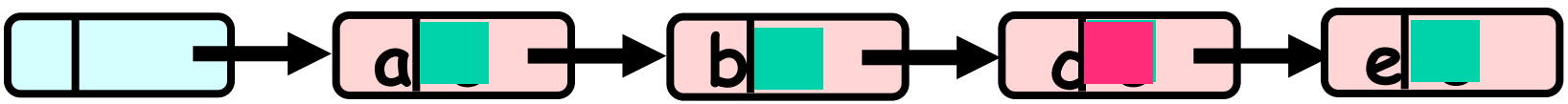


The Solution: Combine Bit and Pointer



Mark-Bit and Pointer
are CASed together
(AtomicMarkableReference)

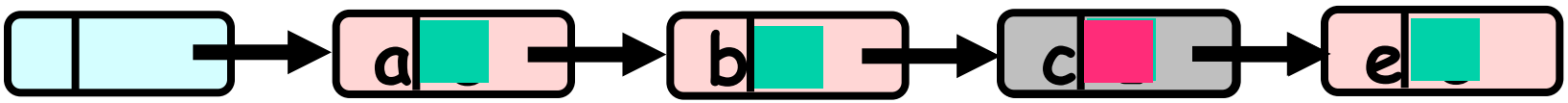
The Solution: Combine Bit and Pointer



Mark-Bit and Pointer
are CASed together
(AtomicMarkableReference)

The Solution: Combine Bit and Pointer

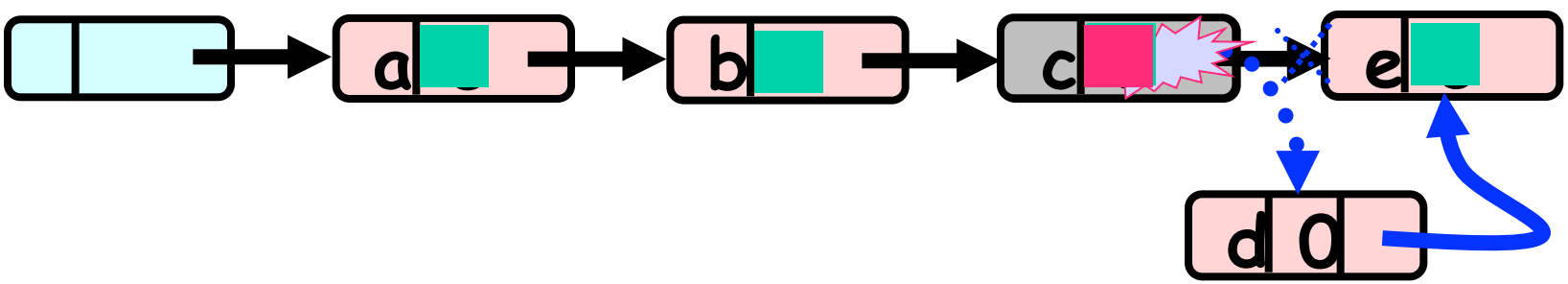
Logical Removal =
Set Mark Bit



Mark-Bit and Pointer
are CASed together
(AtomicMarkableReference)

The Solution: Combine Bit and Pointer

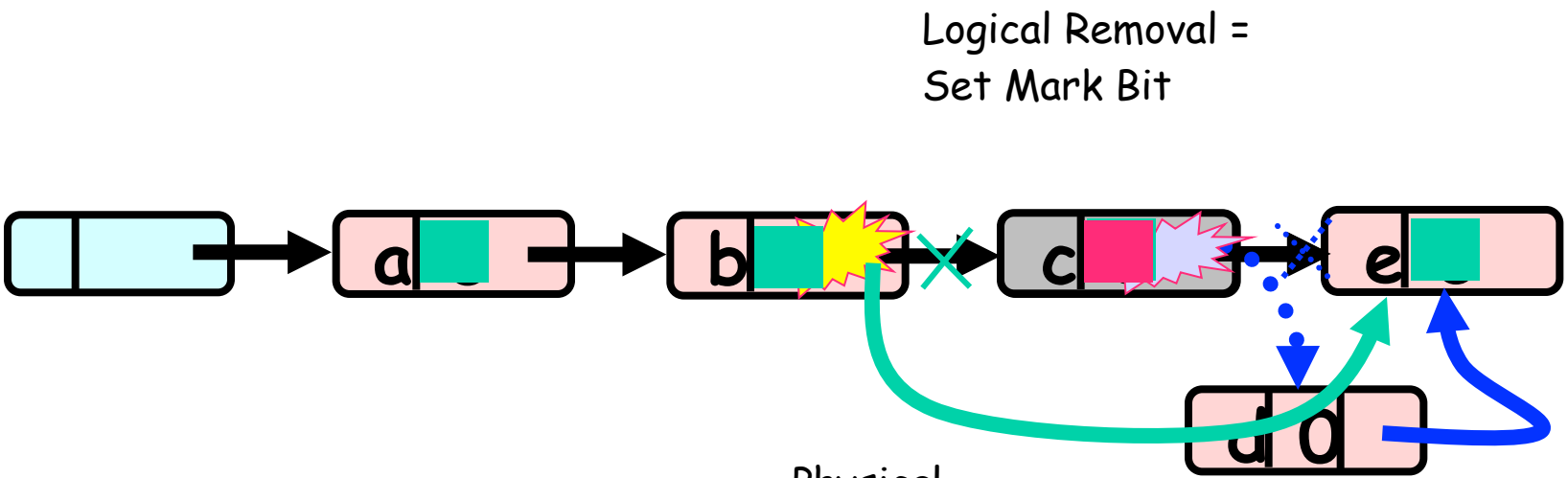
Logical Removal =
Set Mark Bit



Mark-Bit and Pointer are CASed together (AtomicMarkableReference)

Fail CAS: Node not added after logical Removal

The Solution: Combine Bit and Pointer



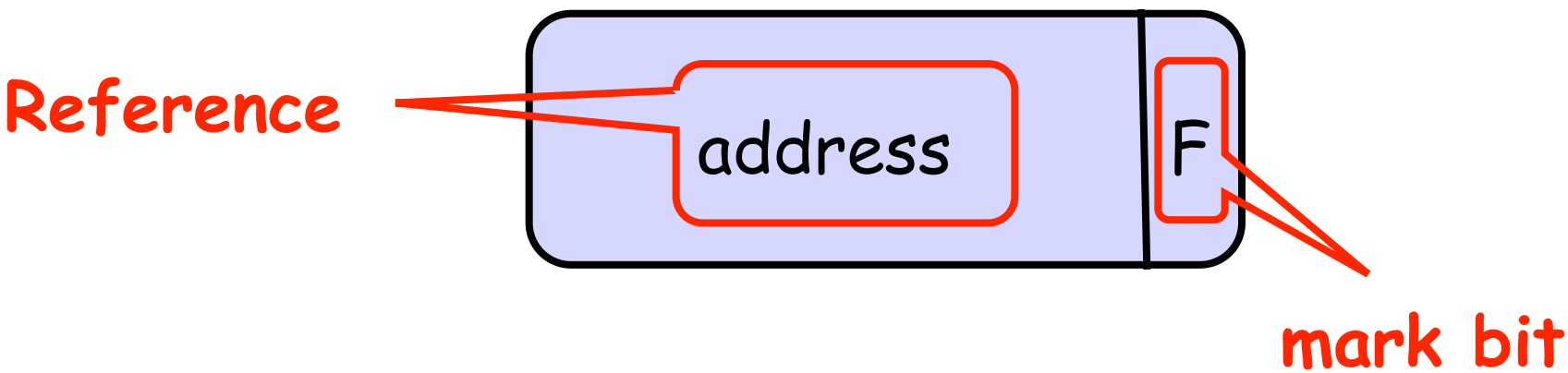
Mark-Bit and Pointer are CASed together (AtomicMarkableReference)

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



Extracting Reference & Mark

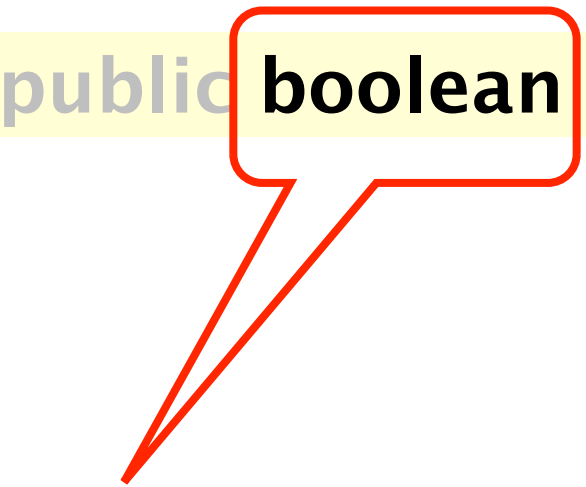
```
Public Object get(boolean[] marked);
```

Returns reference

Returns mark at array index 0!

Extracting Reference Only

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



Changing State

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

Changing State

If this is the current reference …

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

And this is the current mark …

Changing State

…then change to this new reference …

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

… and this new mark

Changing State

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

Changing State

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

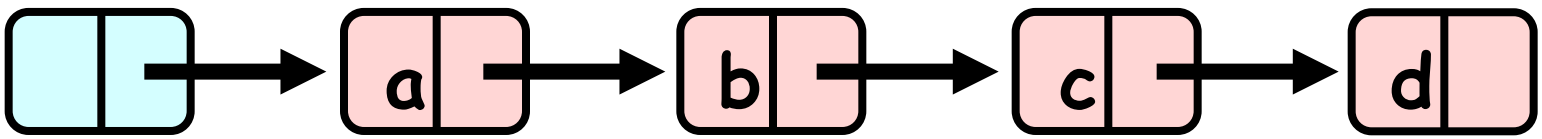
If this is the current reference …

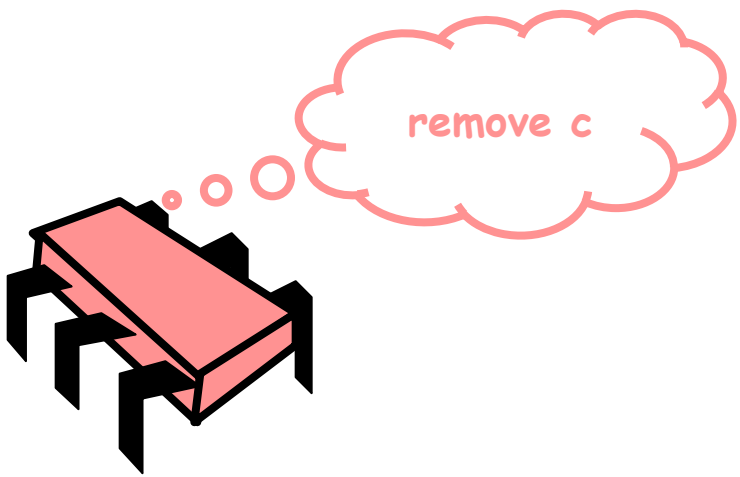
Changing State

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

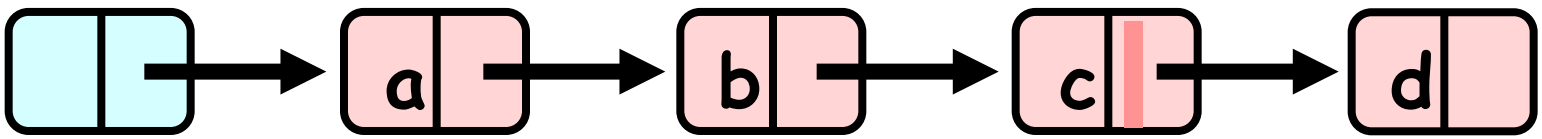
.. then change to this new mark.

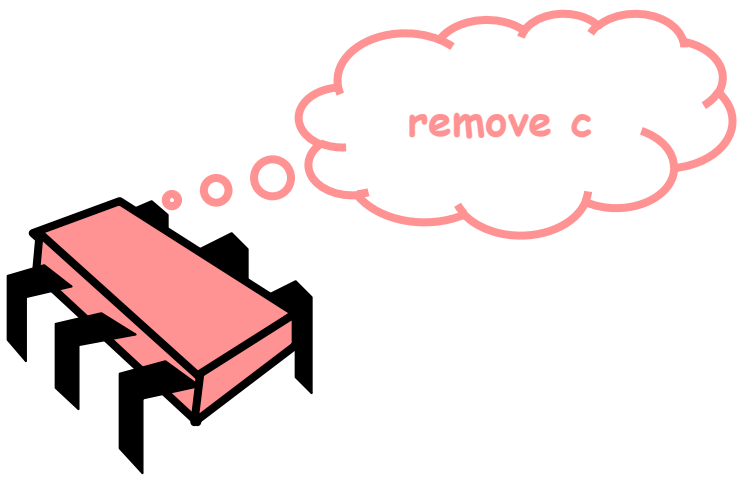
Removing a Node



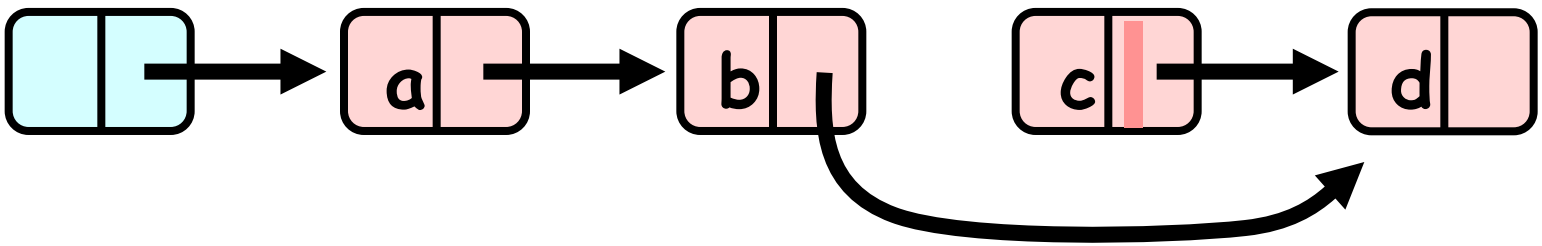


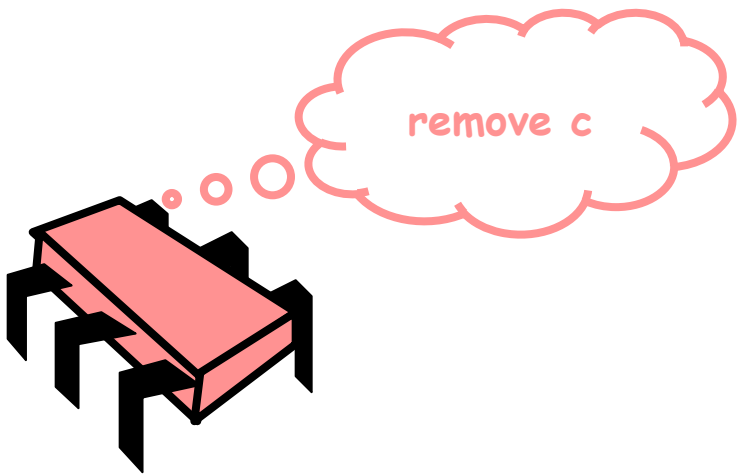
Removing a Node



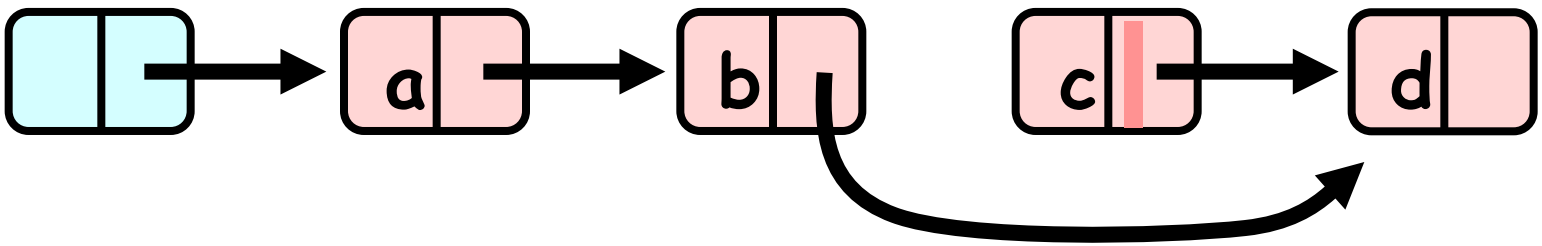


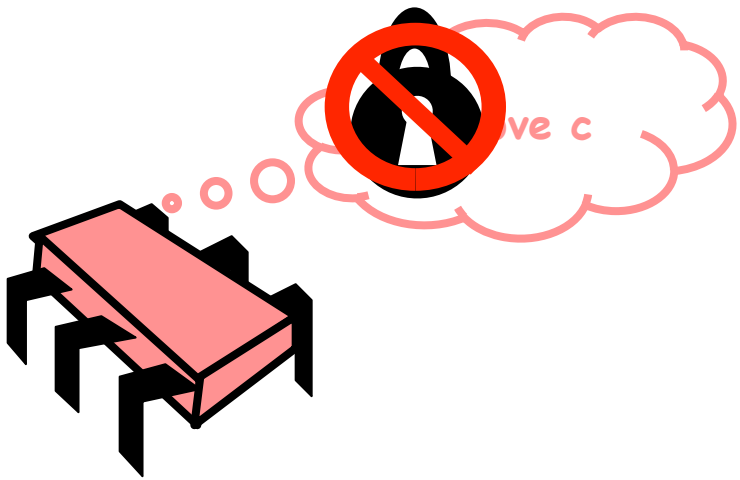
Removing a Node



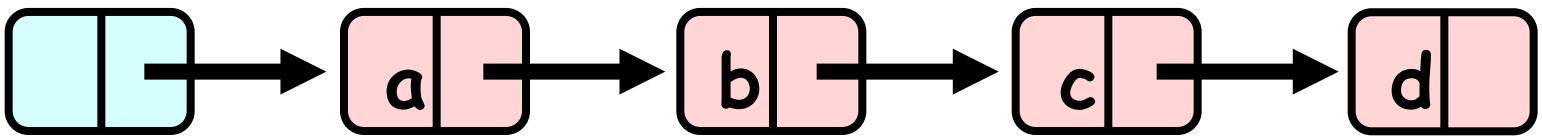


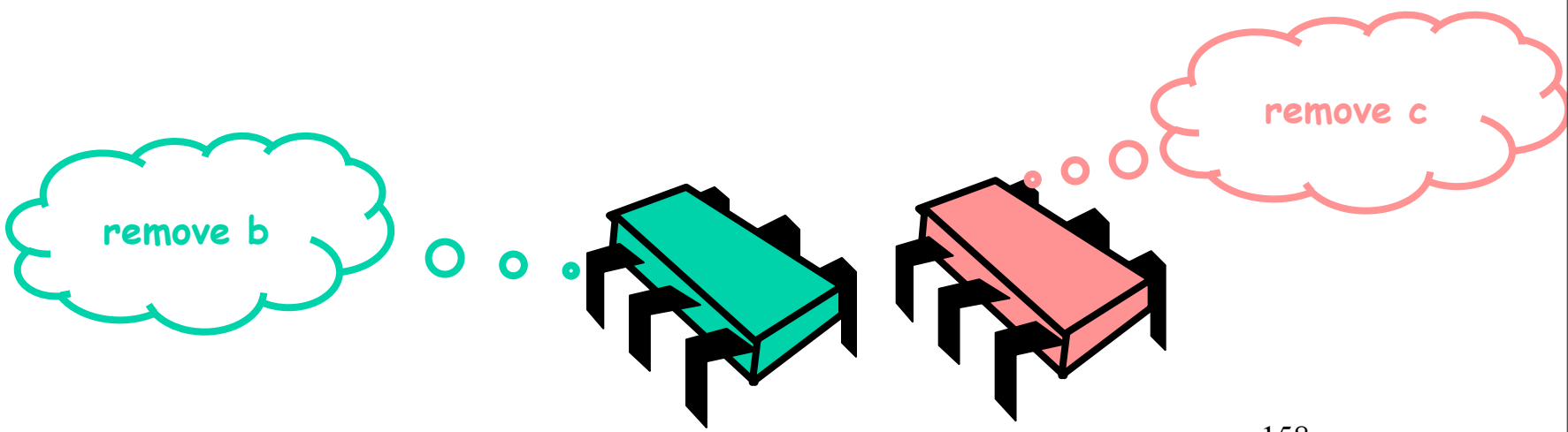
Removing a Node



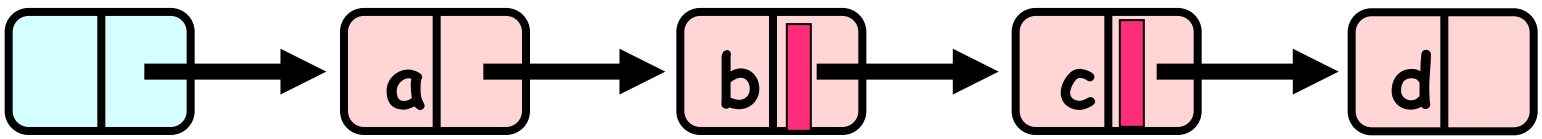


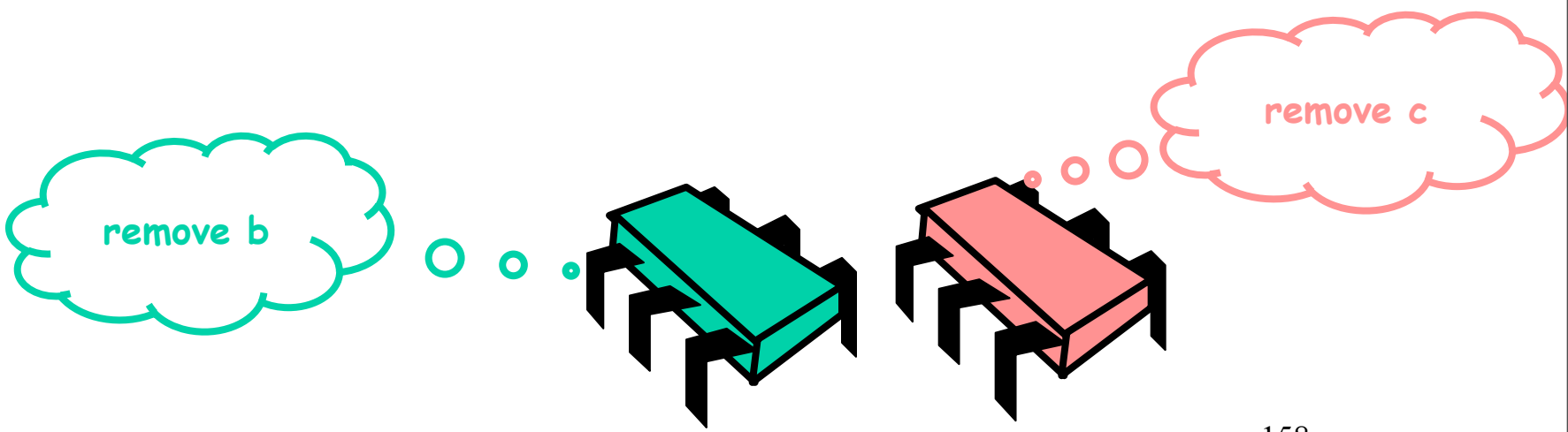
Removing a Node



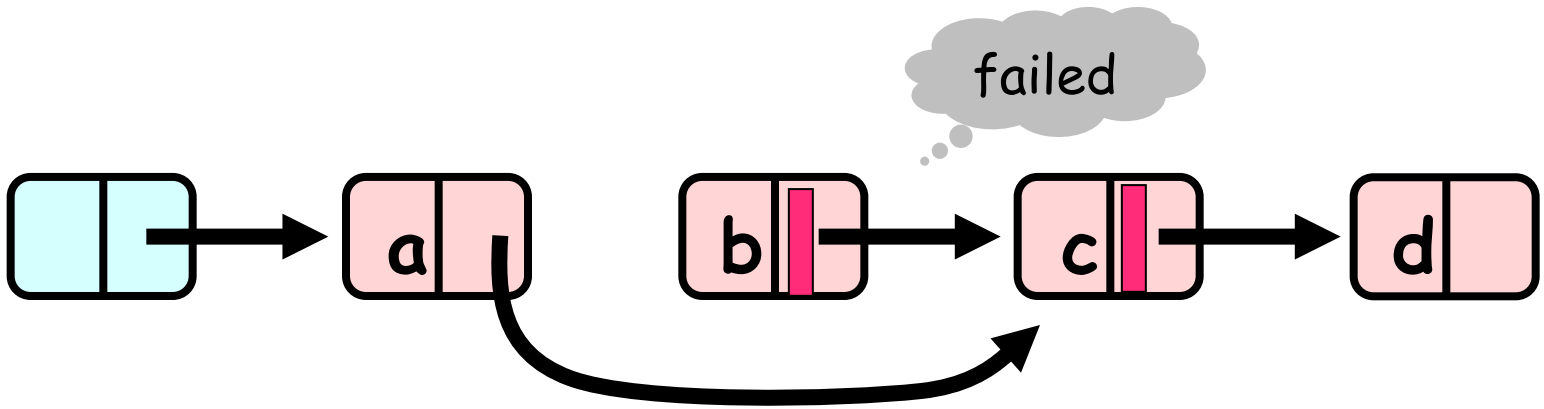


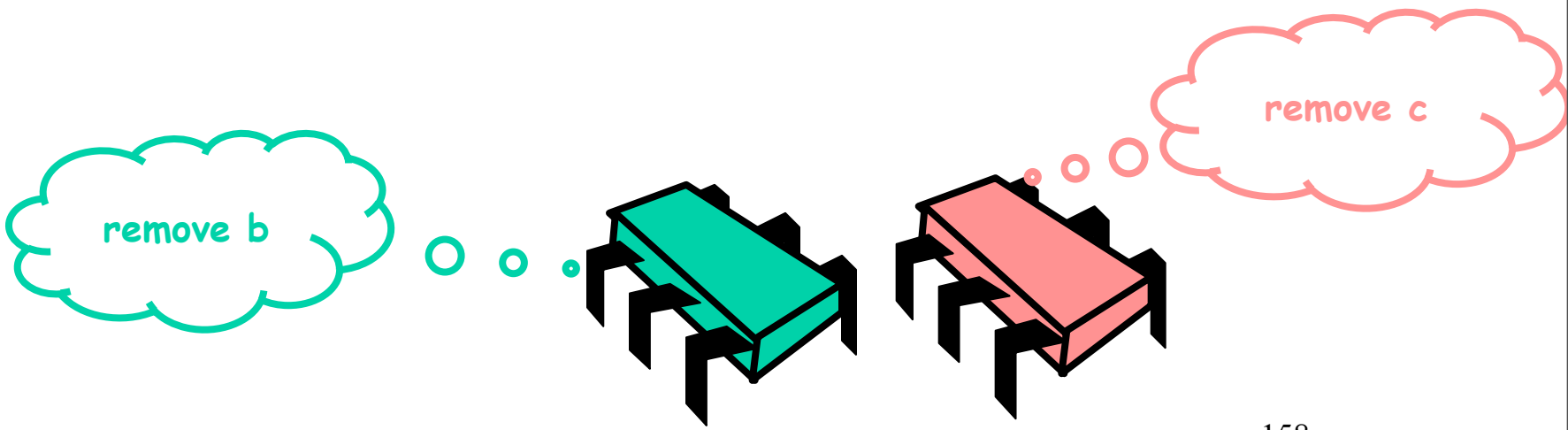
Removing a Node



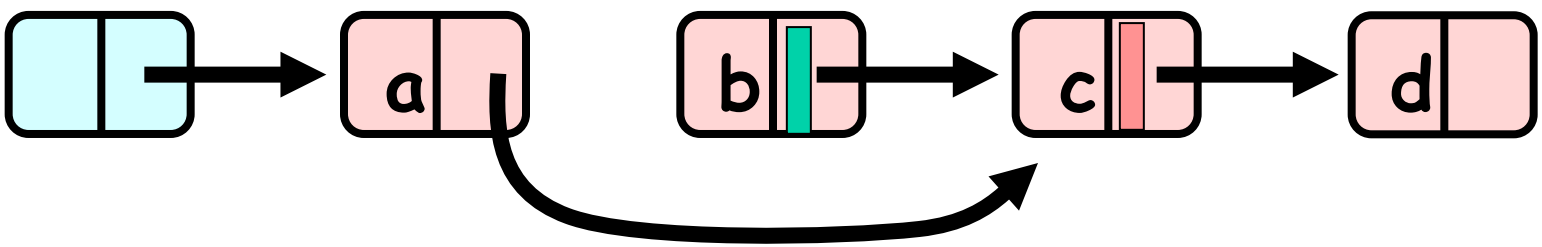


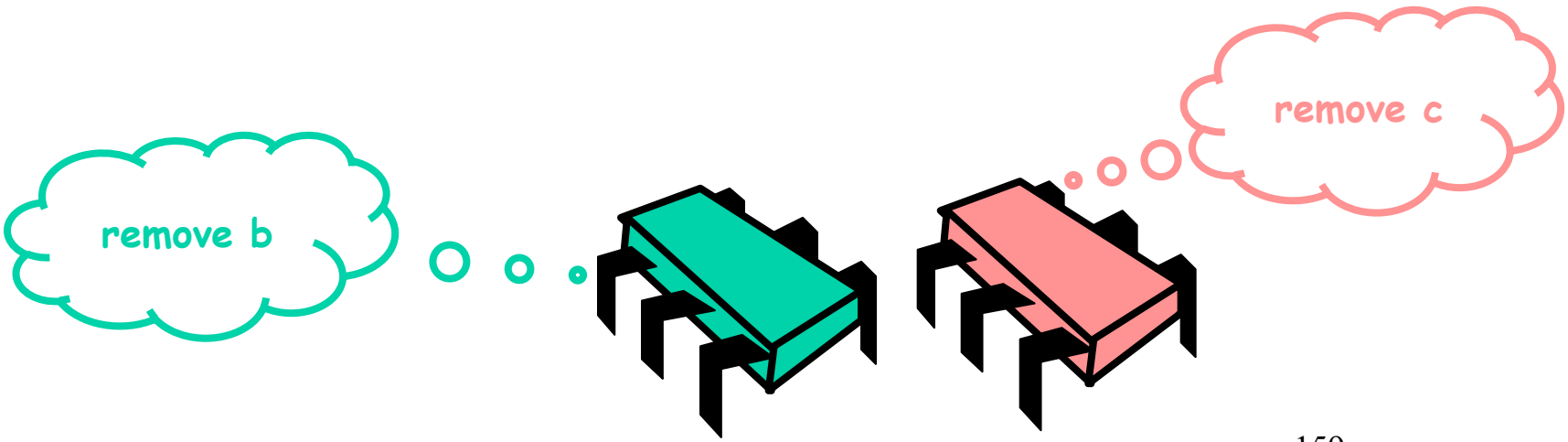
Removing a Node



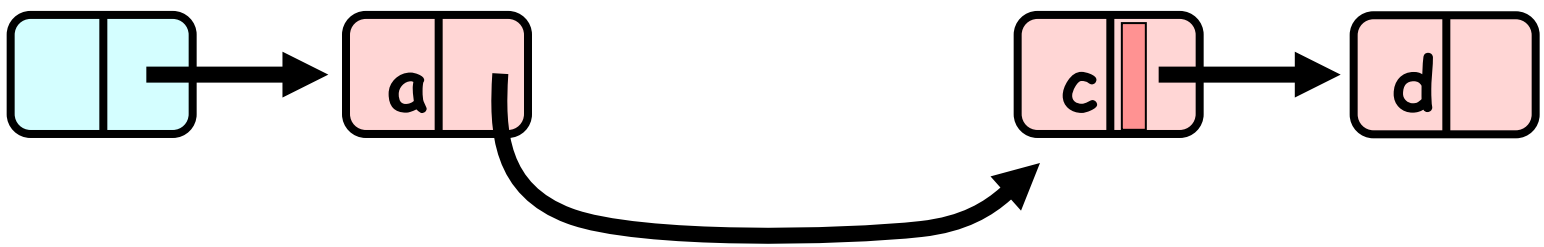


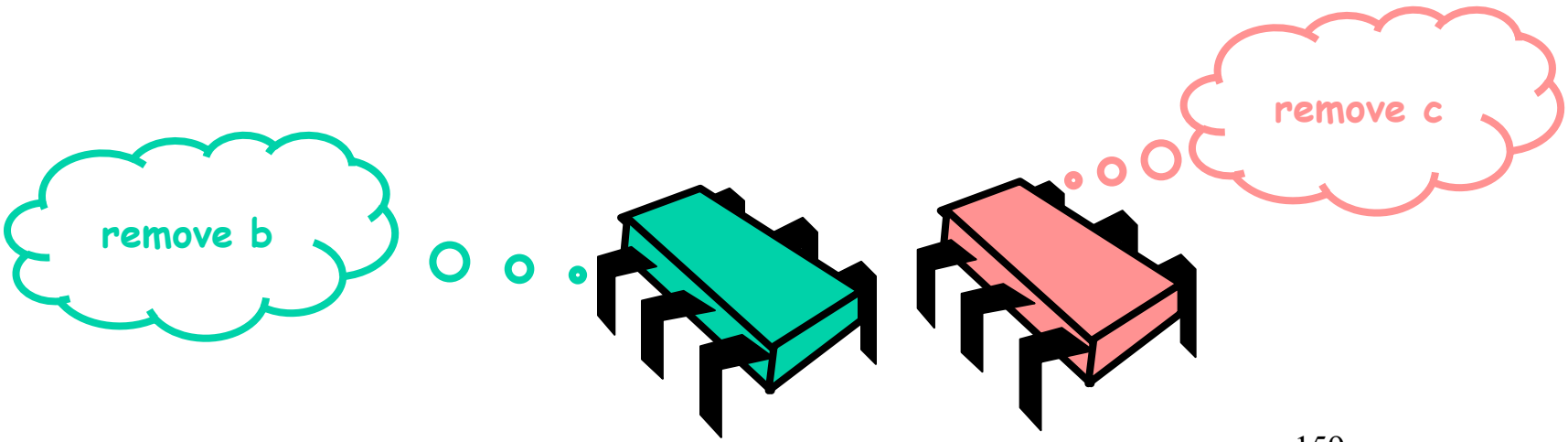
Removing a Node



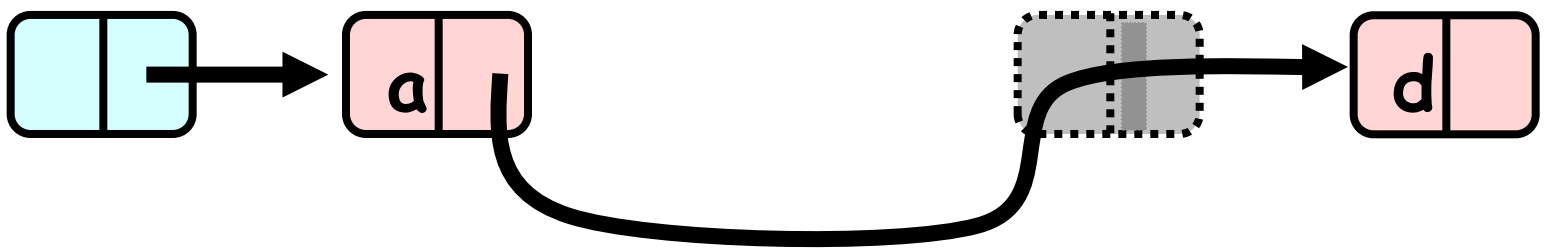


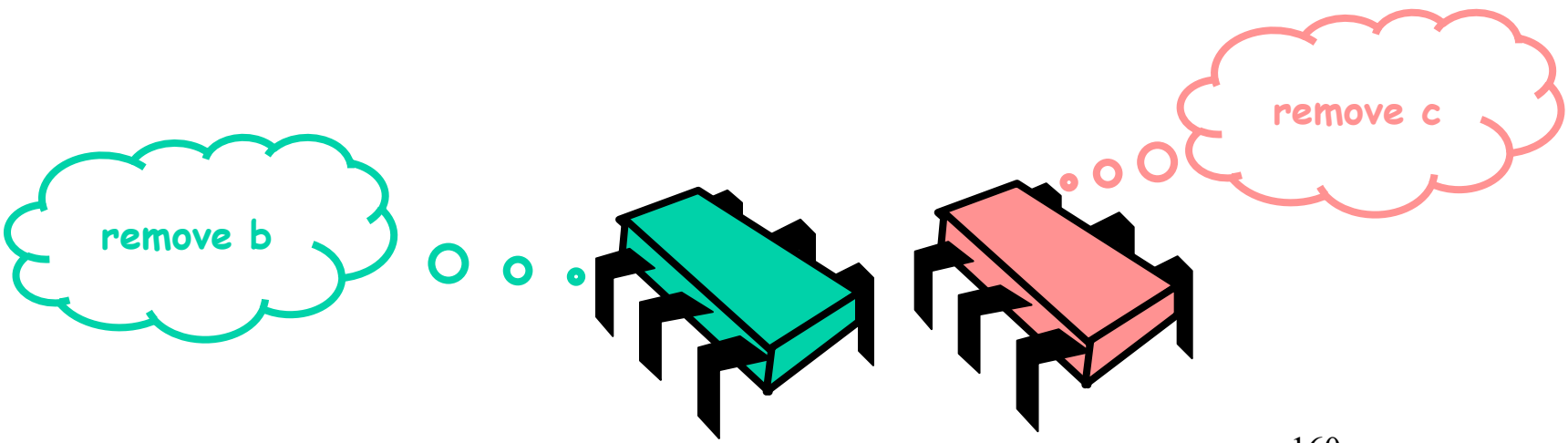
Removing a Node





Removing a Node

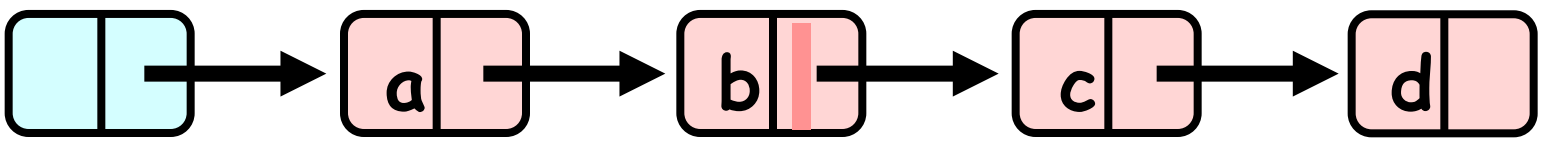


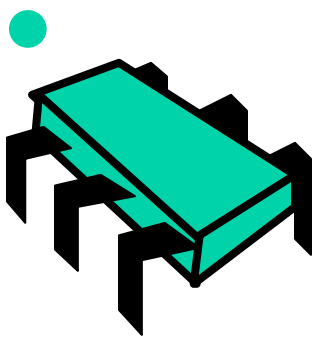


Traversing the List

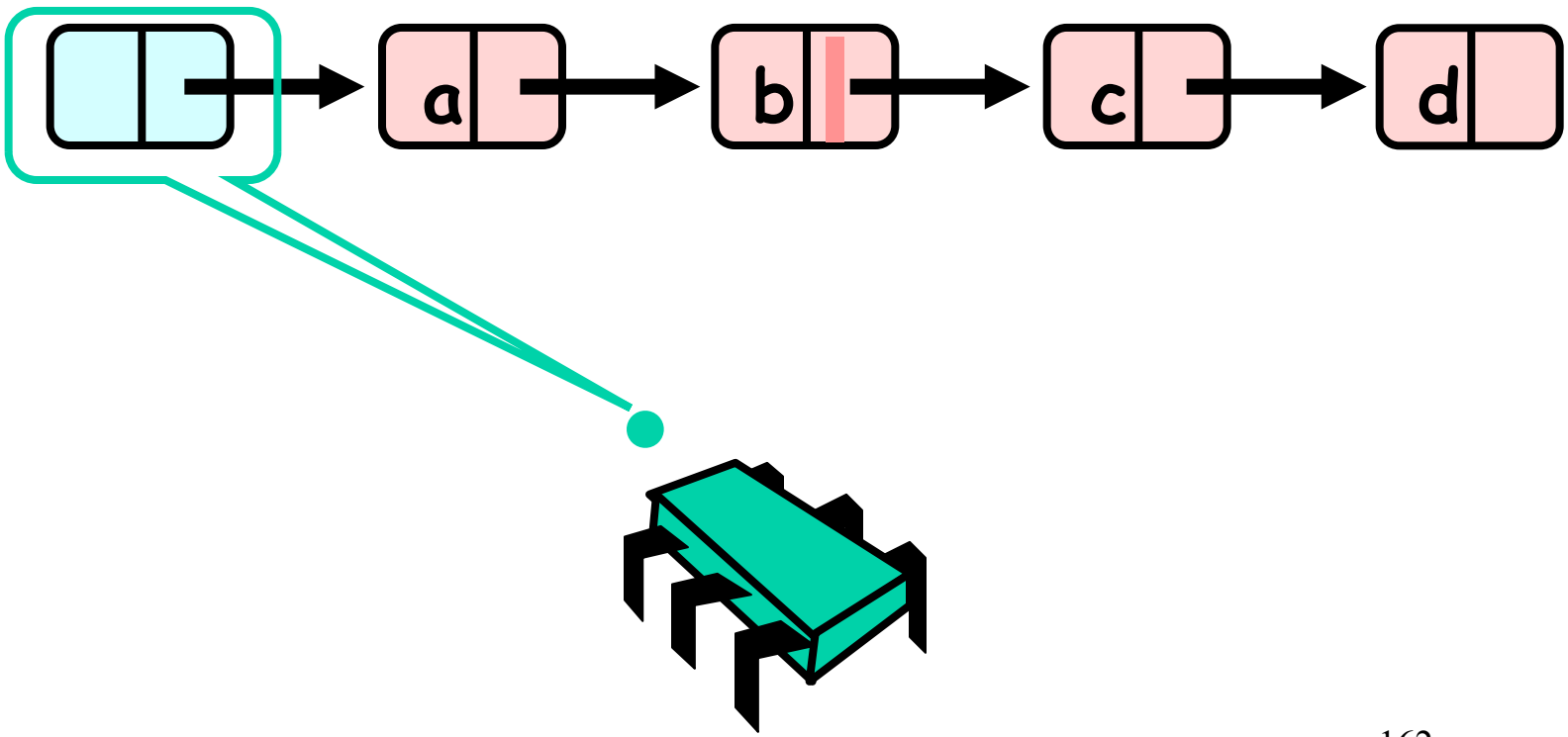
- Q: what do you do when you find a "logically" deleted node in your path?
- A: finish the job.
  - CAS the predecessor's next field
  - Proceed (repeat as needed)

Lock-Free Traversal (only Add and Remove)

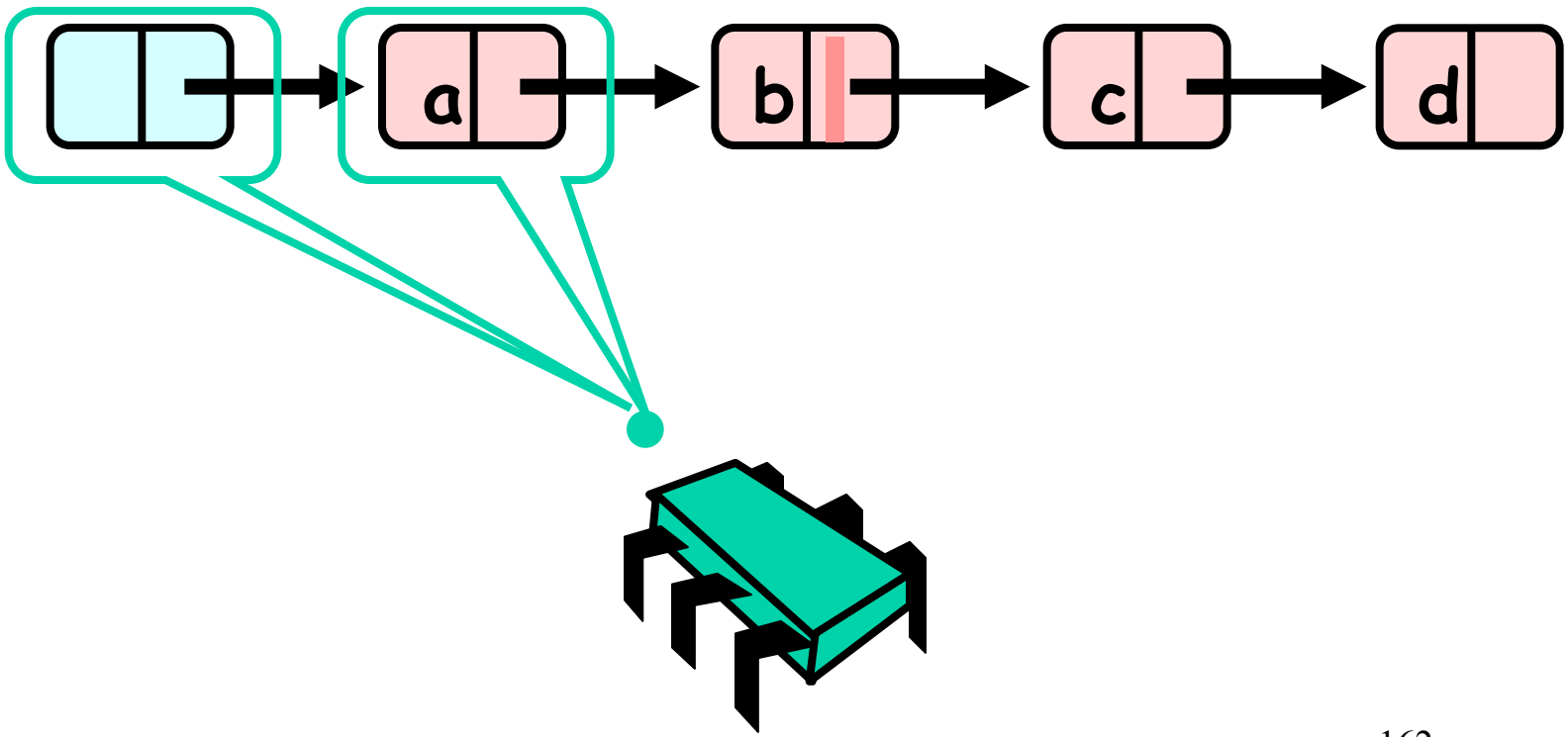




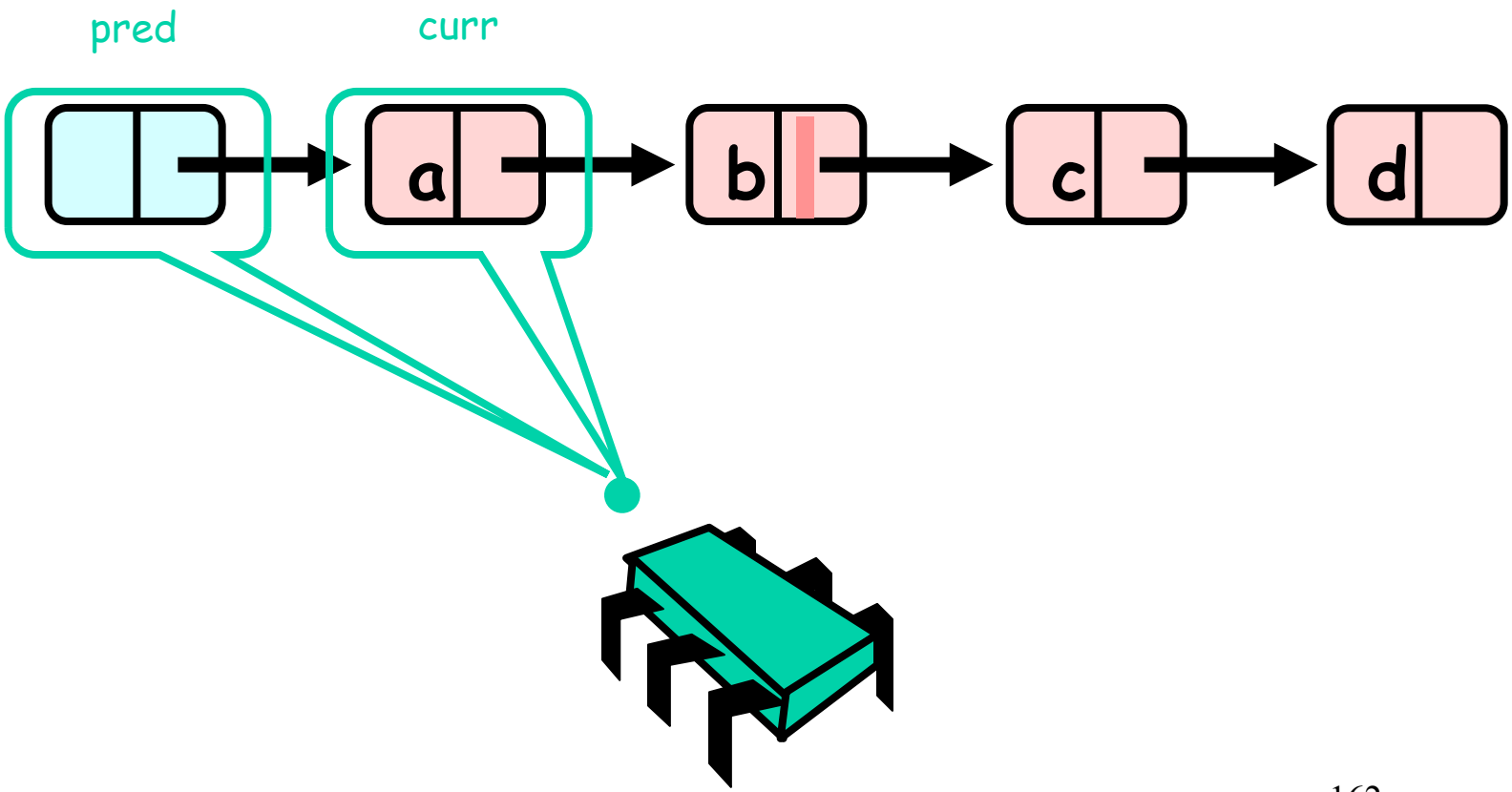
Lock-Free Traversal (only Add and Remove)



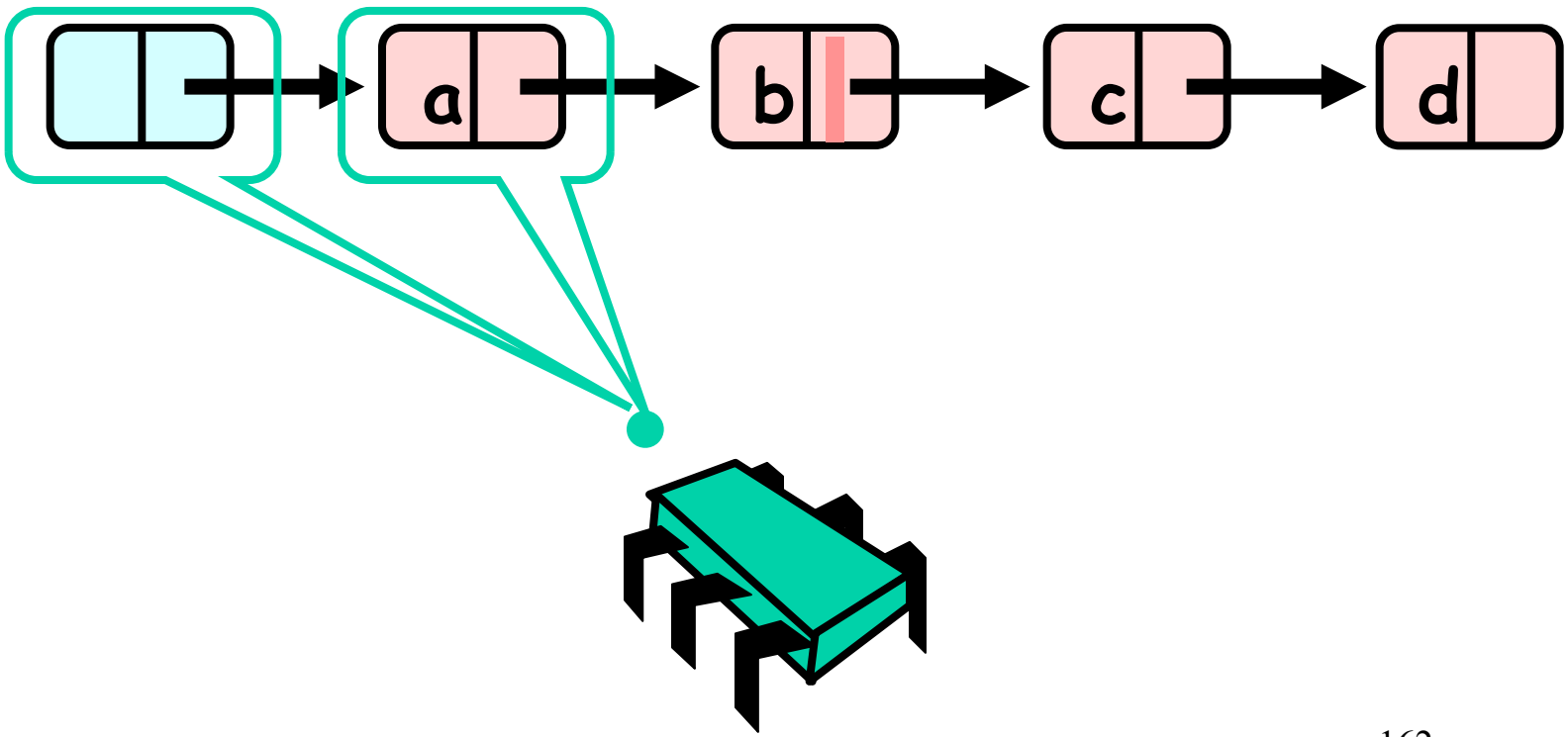
Lock-Free Traversal (only Add and Remove)



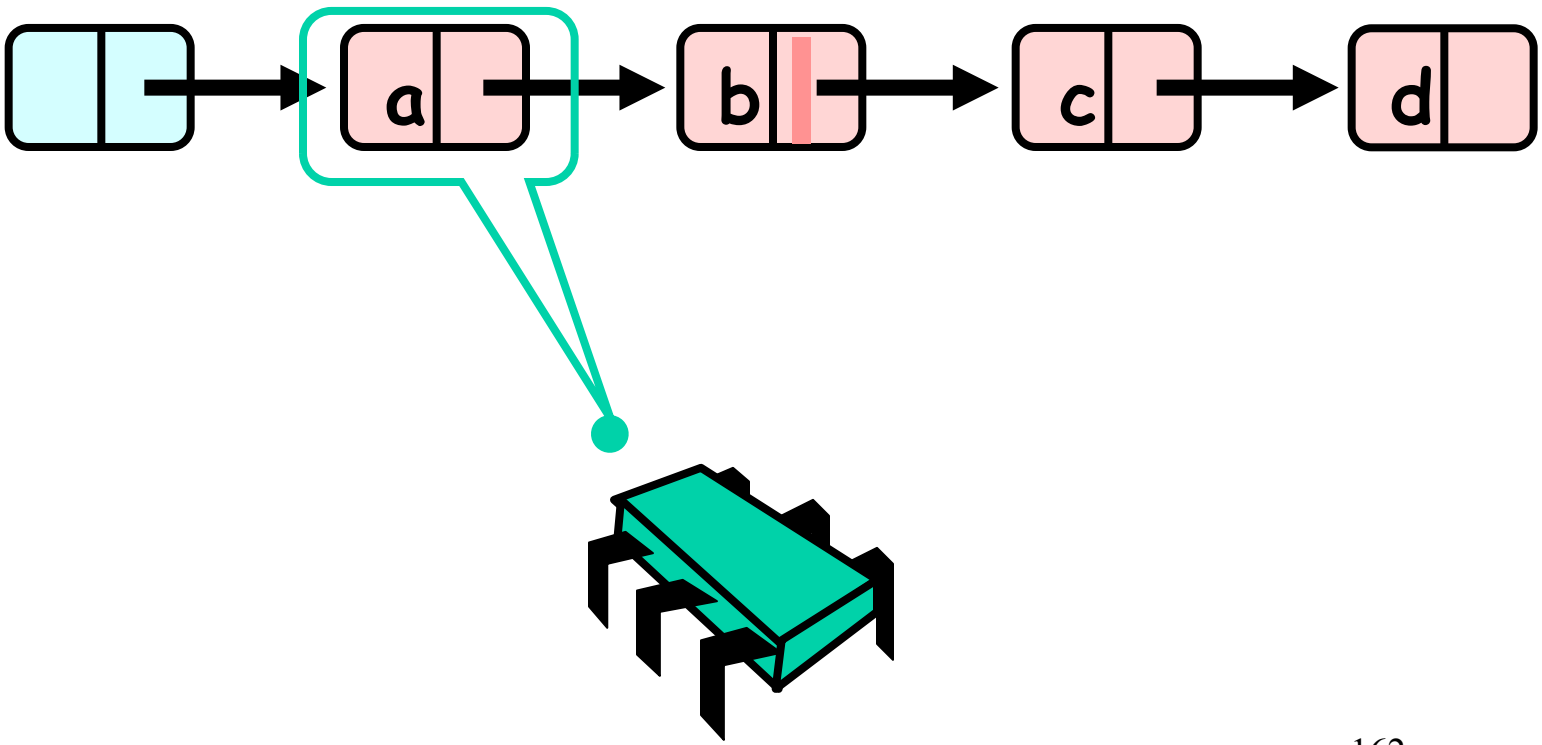
Lock-Free Traversal (only Add and Remove)



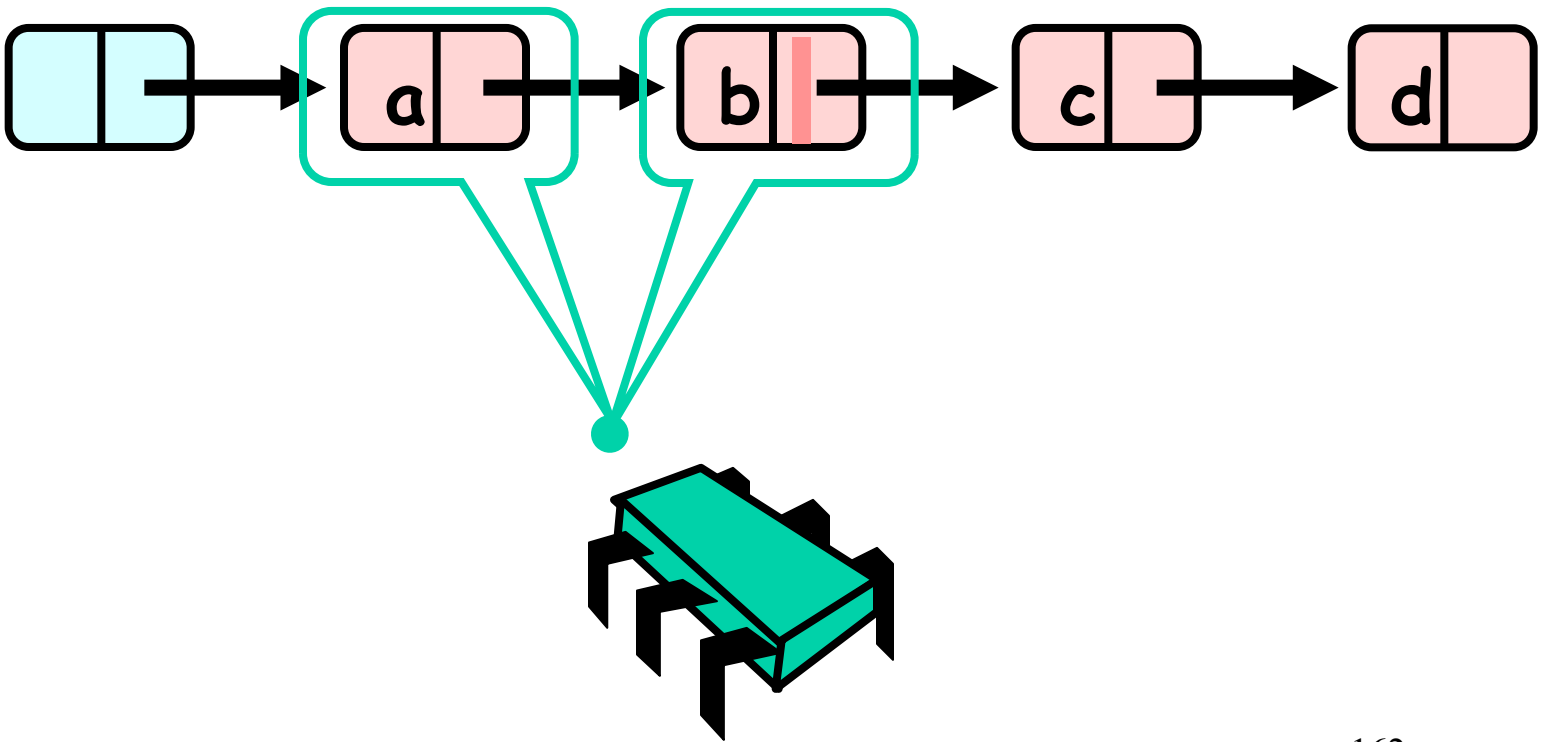
Lock-Free Traversal (only Add and Remove)



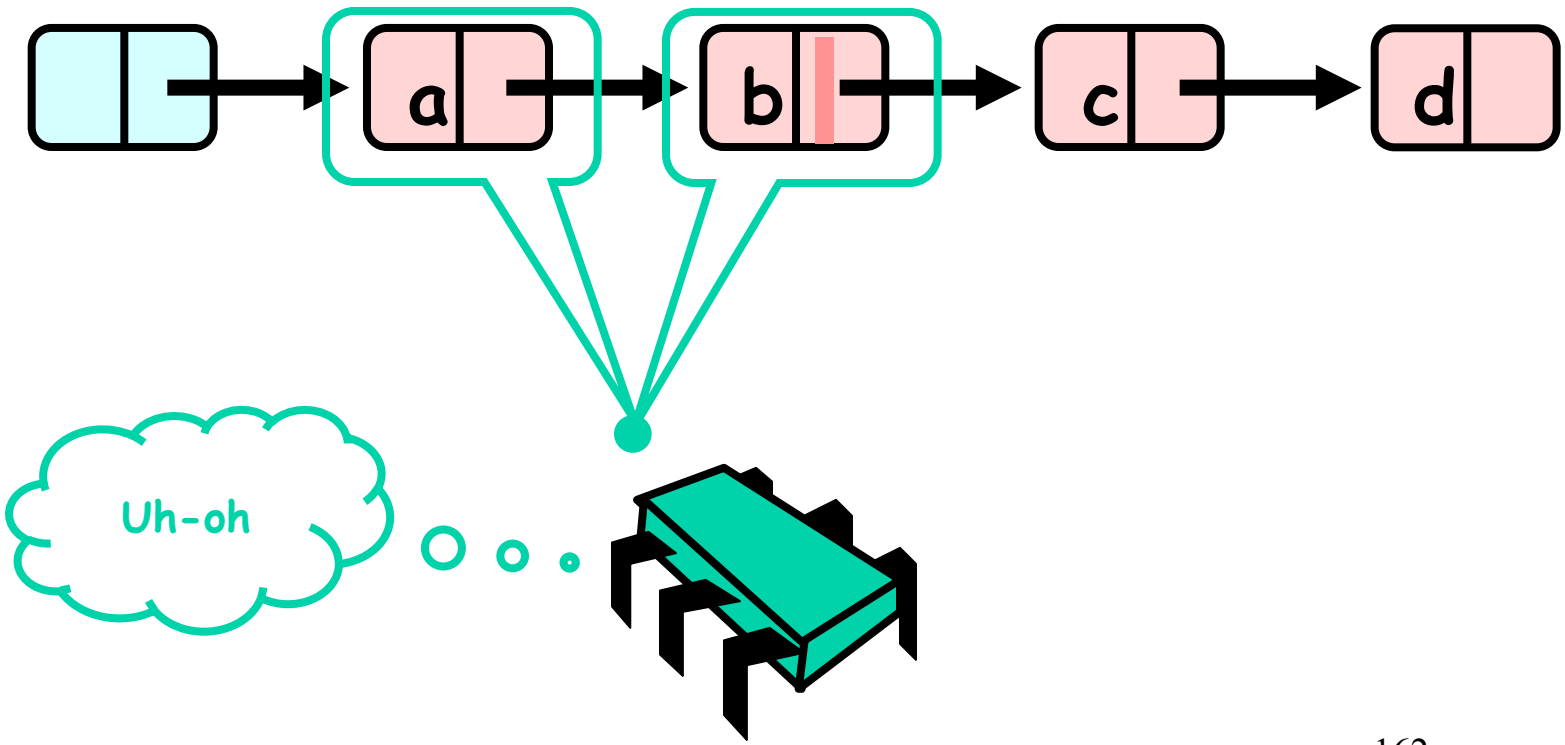
Lock-Free Traversal (only Add and Remove)



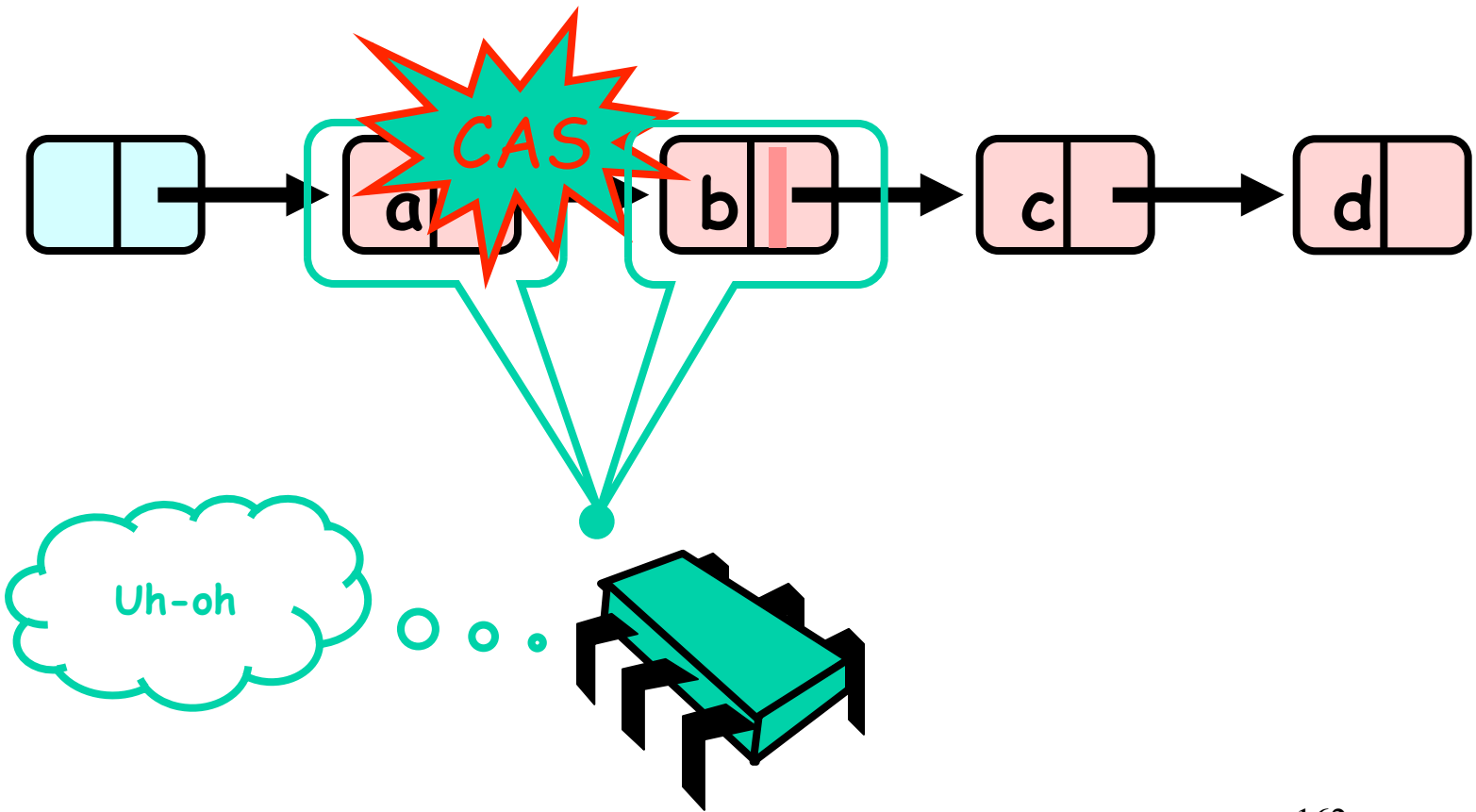
Lock-Free Traversal (only Add and Remove)



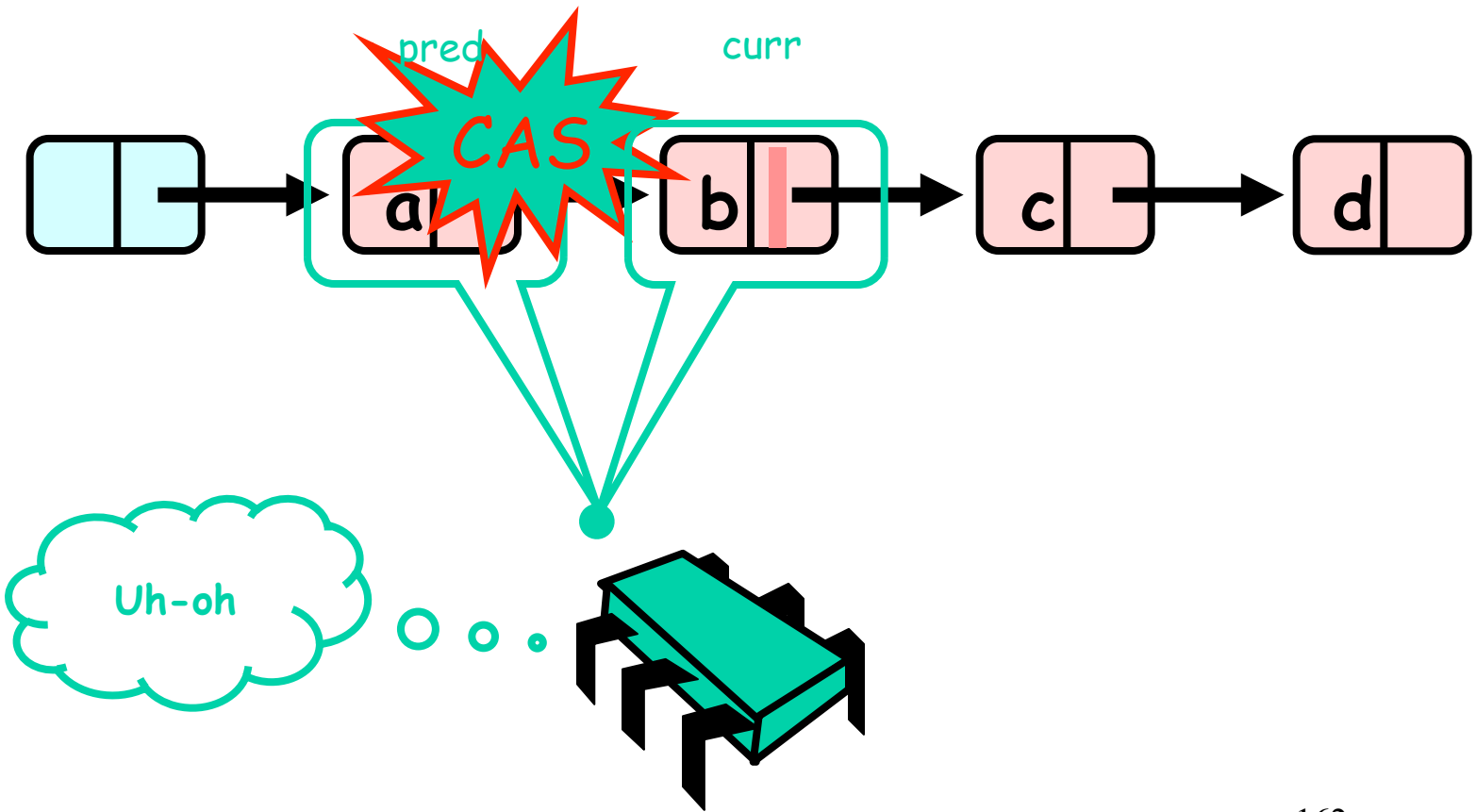
Lock-Free Traversal (only Add and Remove)



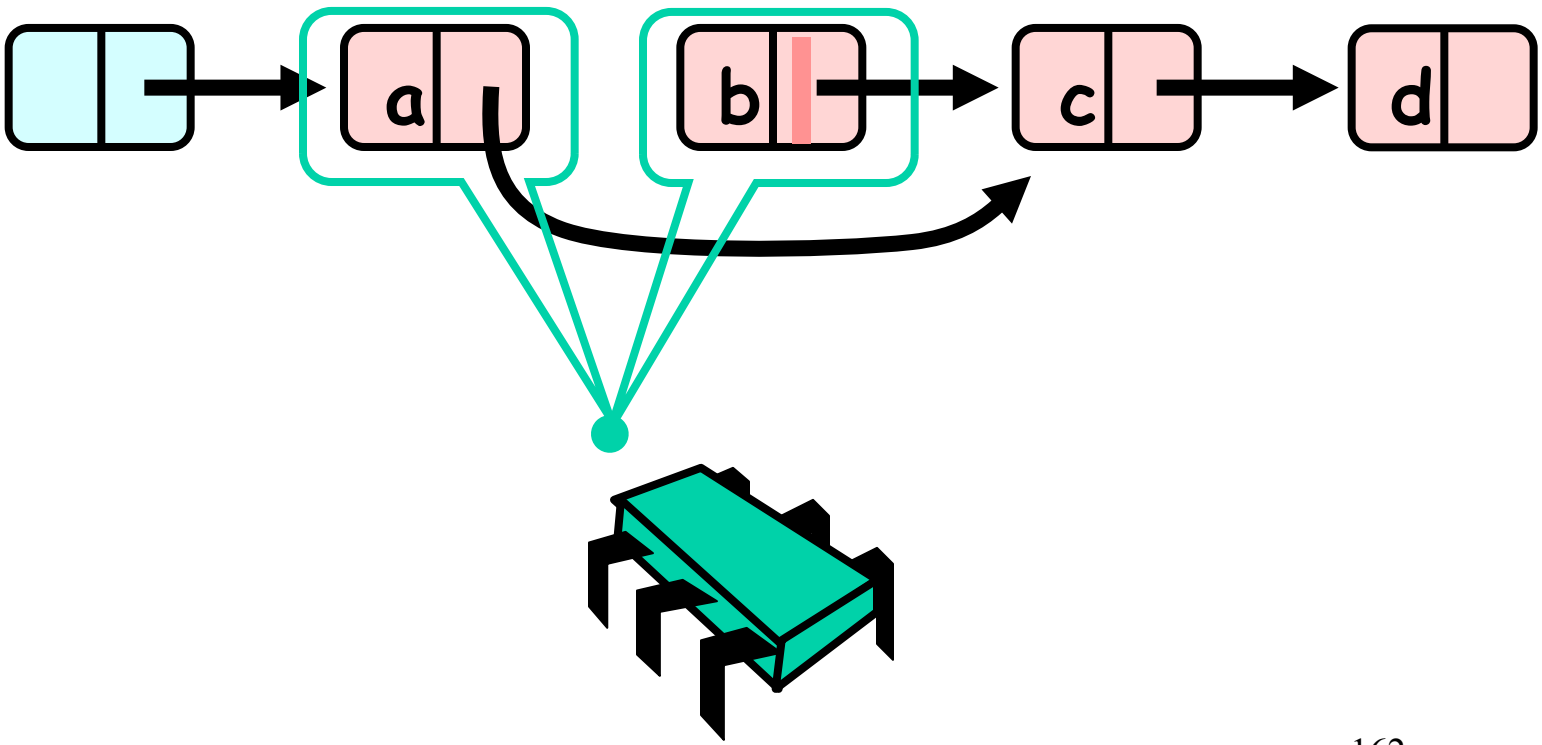
Lock-Free Traversal (only Add and Remove)



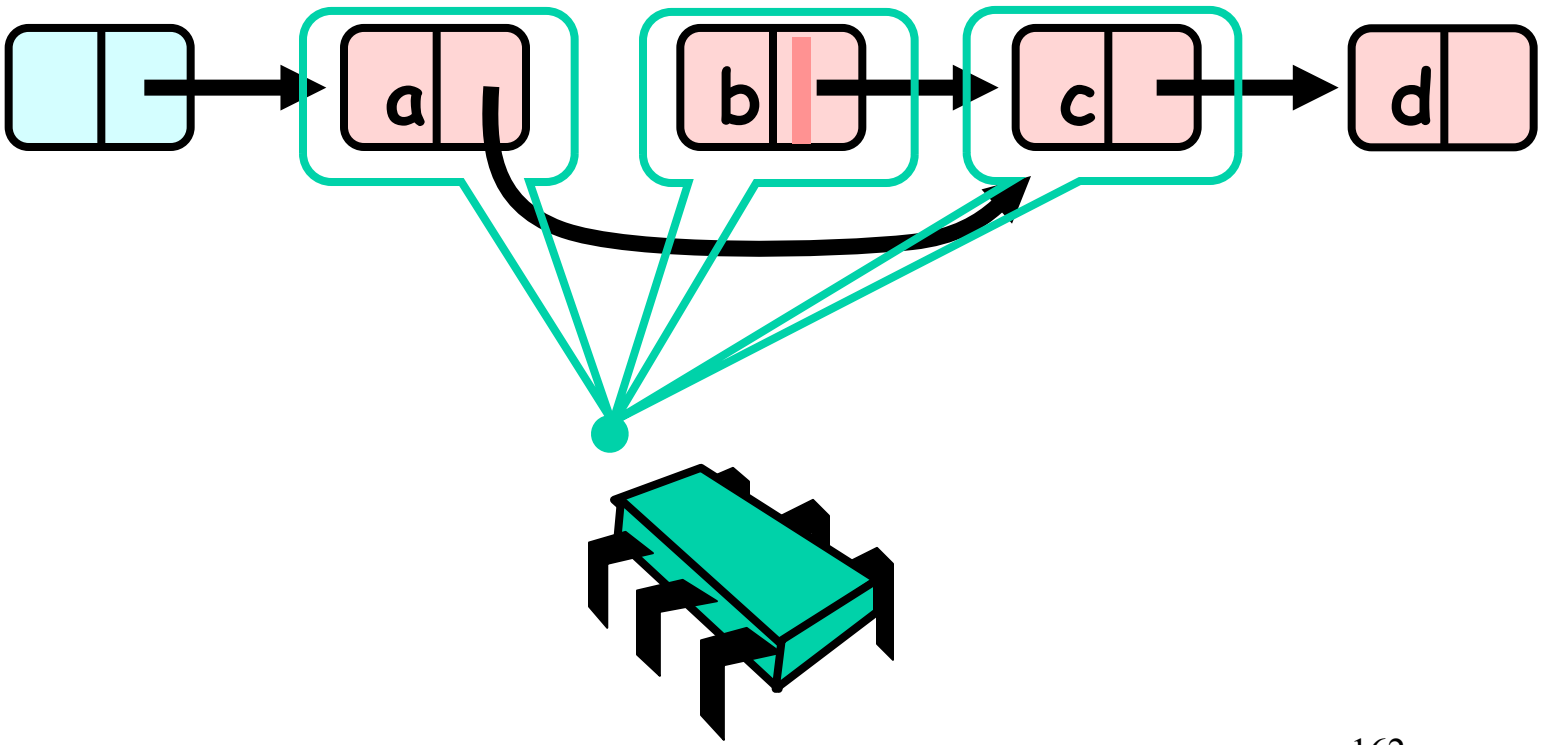
Lock-Free Traversal (only Add and Remove)



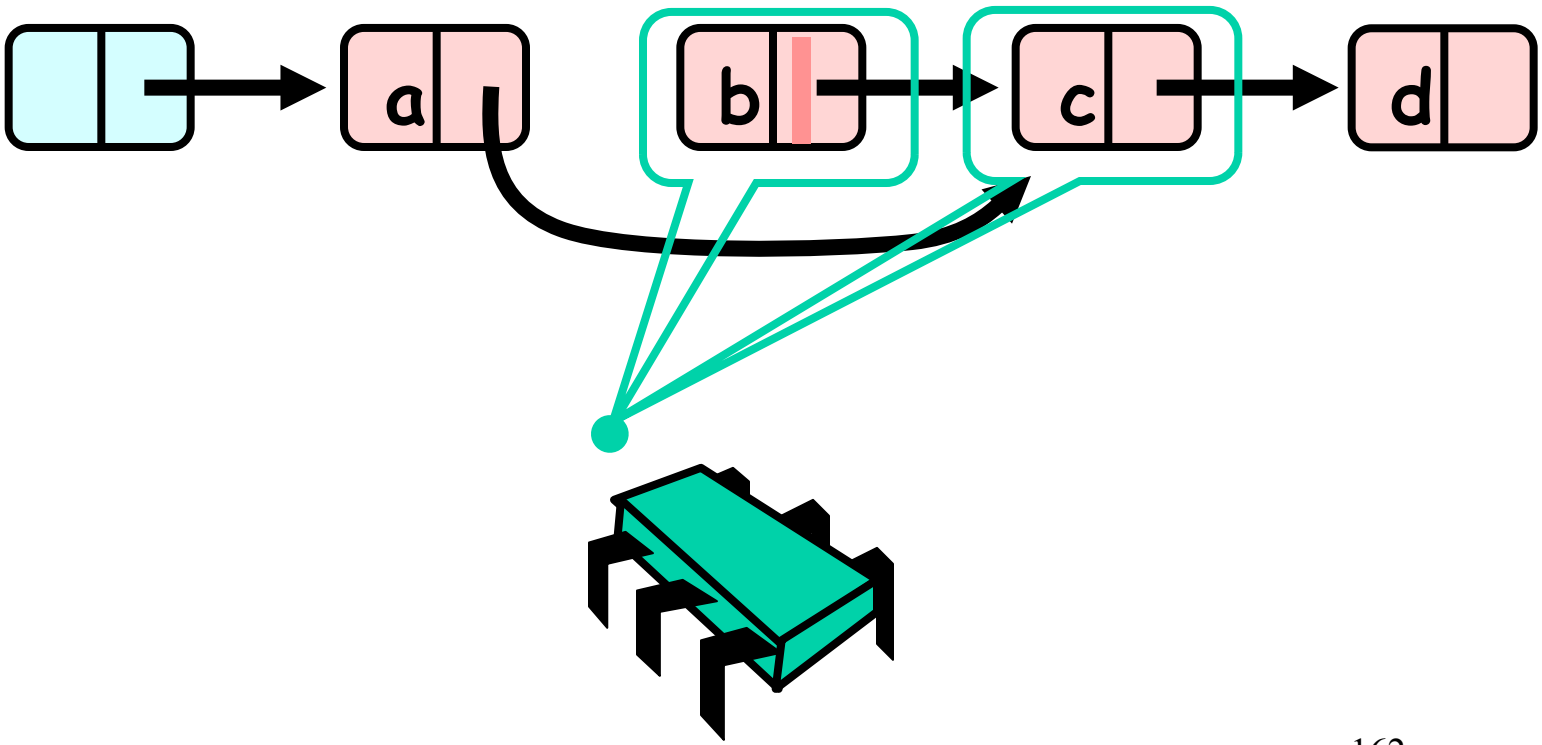
Lock-Free Traversal (only Add and Remove)



Lock-Free Traversal (only Add and Remove)



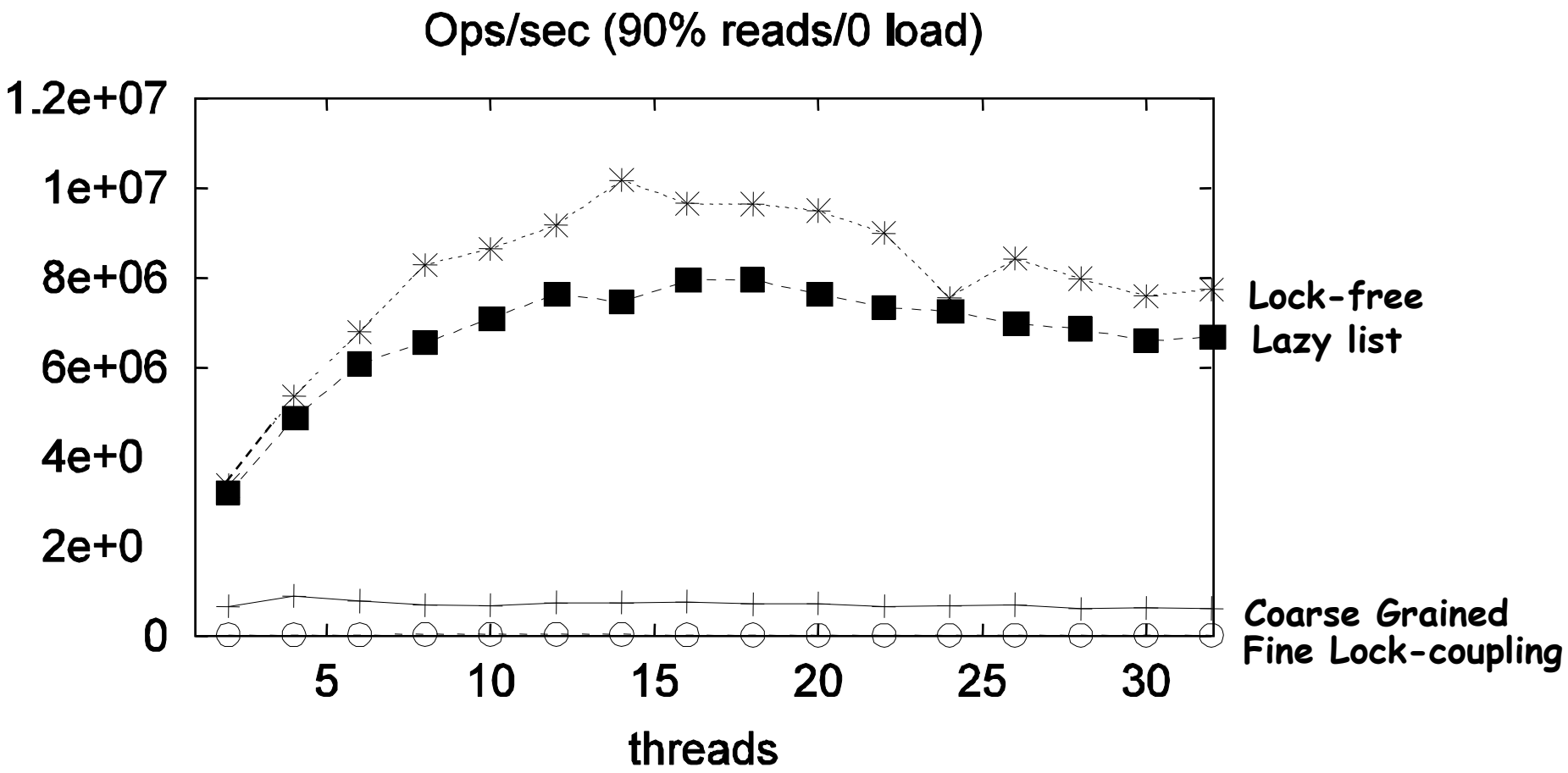
Lock-Free Traversal (only Add and Remove)



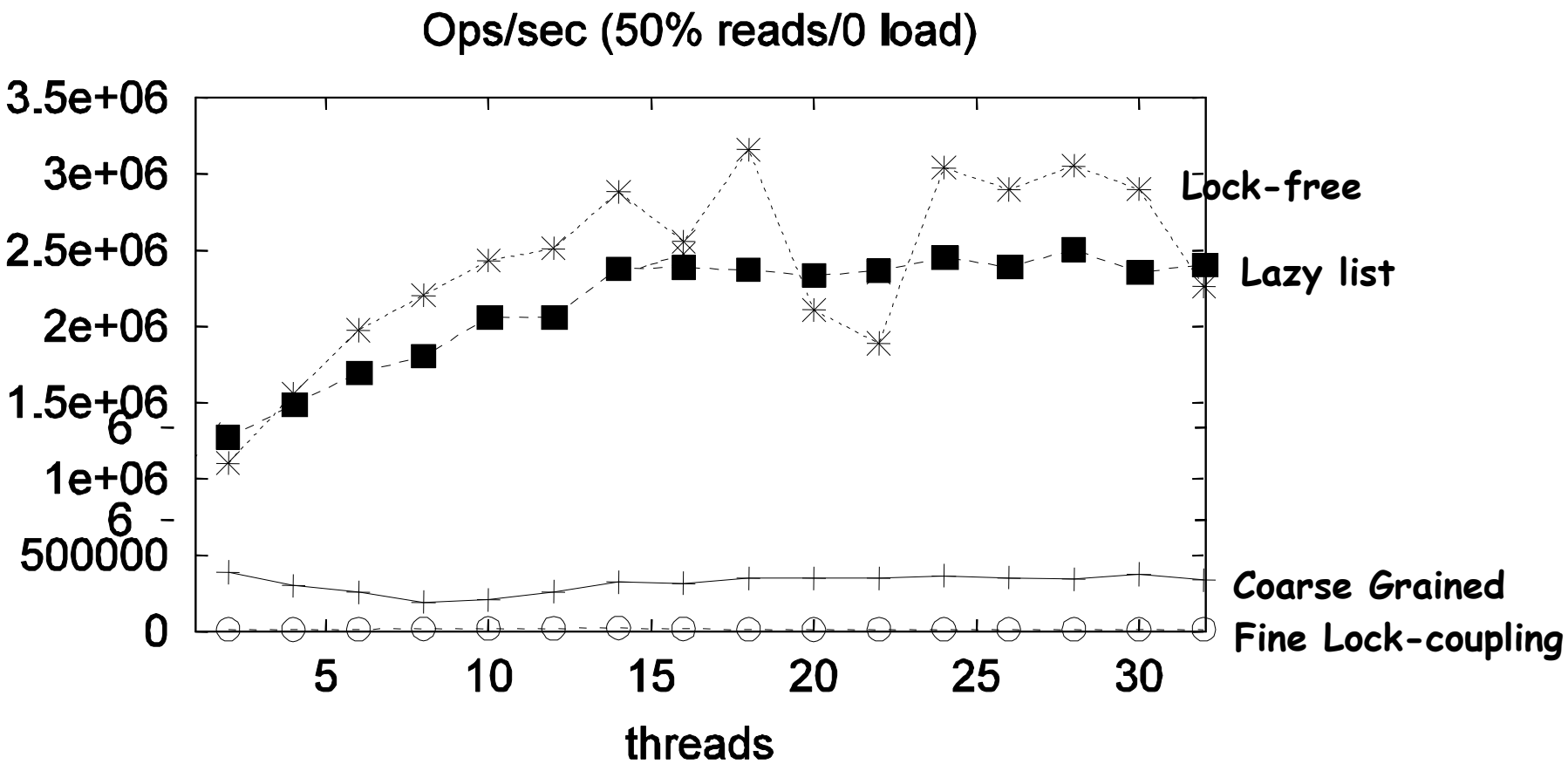
Performance

On 16 node shared memory machine
Benchmark throughput of Java List-based Set algs. Vary % of Contains() method Calls.

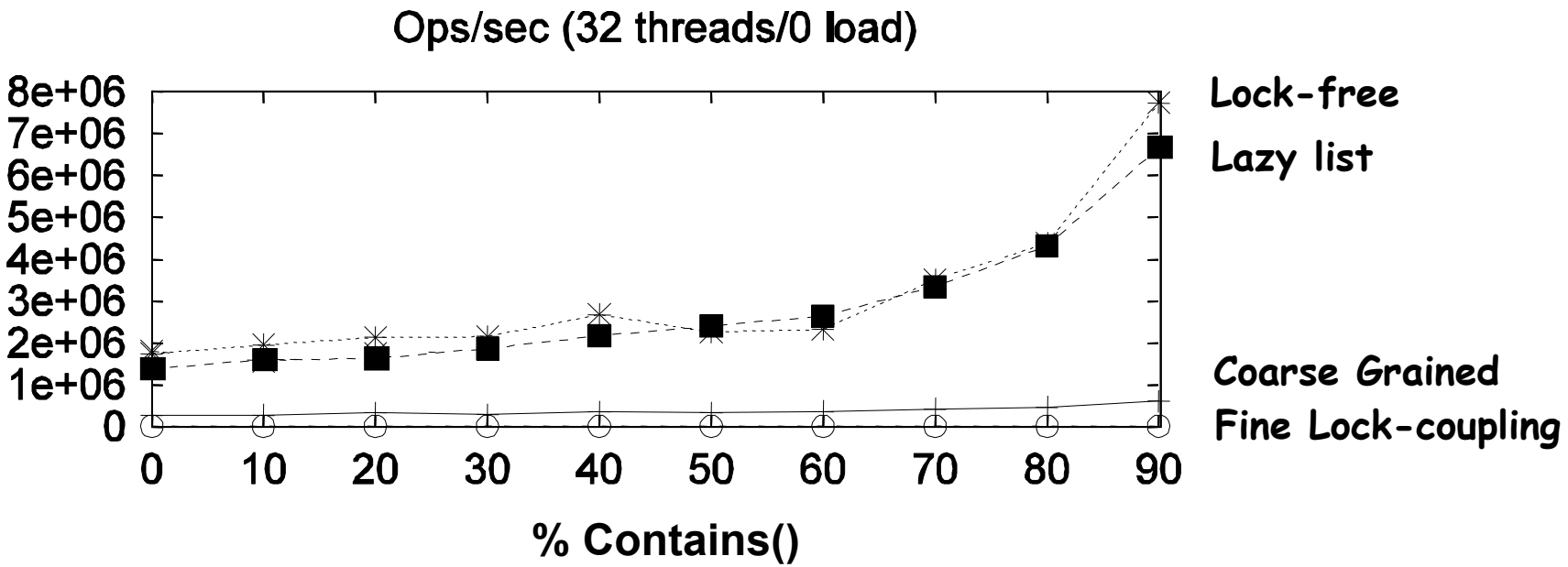
High Contains Ratio



Low Contains Ratio



As Contains Ratio Increases



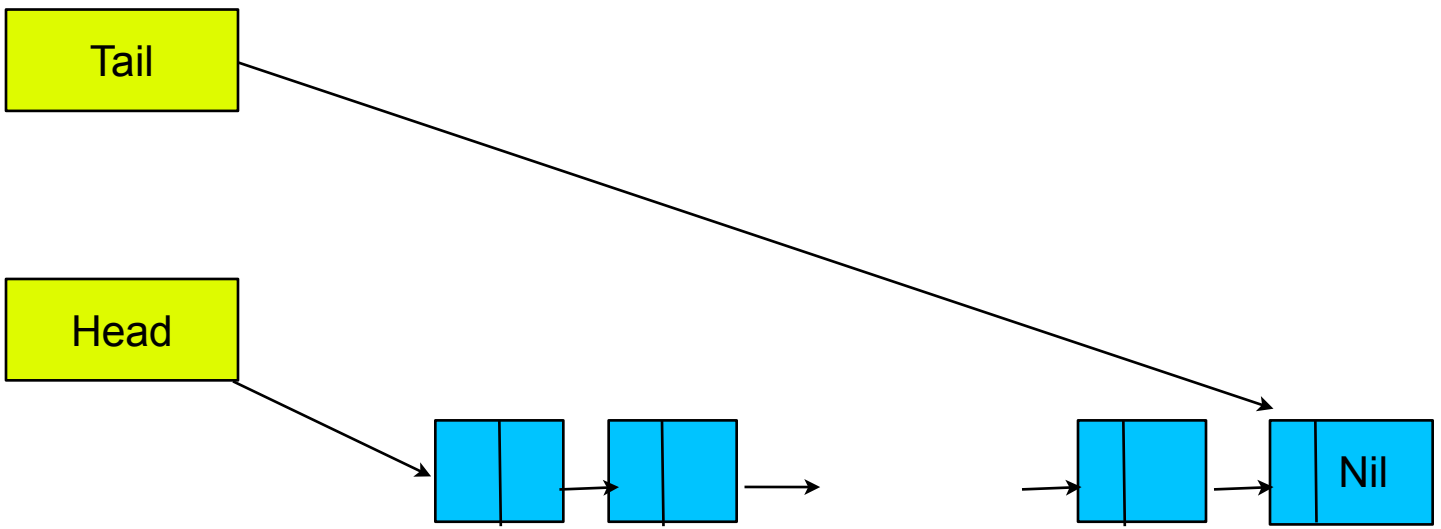
Summary

- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization
- Lock-free synchronization

"To Lock or Not to Lock"

- Locking vs. Non-blocking: Extremist views on both sides
- The answer: nobler to compromise, combine locking and non-blocking
  - Example: Lazy list combines blocking add() and remove() and a wait-free contains()
  - Remember: Blocking/non-blocking is a property of a method

Example: Lock-free queue



Inserting into the tail of the queue requires two updates:
- updating the tail pointer
- updating the next pointer in the current last element

Example

- Separate CAS operations won't work because other threads can interfere in between them
- Idea:
  - Ensure data structure is always in a consistent state even in the middle of a multi-update step
  - Failure should not prevent other threads from making progress
    - if thread B arrives to find structure is currently being modified by A, it will finish the operation on behalf of A
      - this allows B to make progress without waiting for A to finish
      - when A continues, it will realize B already did the job for it.

Code

```
@ThreadSafe
public class LinkedQueue <E> {
   private static class Node <E> {
      final E item;
      final AtomicReference<Node<E>> next;

      public Node(E item, Node<E> next) {
         this.item = item;
         this.next = new AtomicReference <Node<E>> (next);
  }
}

private final Node<E> dummy = new Node<E>(null, null);
private final AtomicReference<Node<E> head =
   new AtomicReference<Node<E>>(dummy);
private final AtomicReference<Node<E>> tail =
   new AtomicReference<Node<E>>(dummy);
```

Code

```
public boolean put(E item) {
  Node<E> newNode = new Node<E>(item, null);
  while (true) {
    Node<E> curTail = tail.get();
    Node<E> tailNext = curTail.next.get();
    if (curTail == tail.get()) {
      if (tailNext != null) {          A
        // Queue in intermediate state, advance tail
        tail.compareAndSet(curTail, tailNext);   B
      } else {
        // In quiescent state, try inserting new node
        if (curTail.next.compareAndSet(null, newNode)) {   C
            // Insertion succeeded, try advancing tail
            tail.compareAndSet(curTail, newNode);   D
            return true;
          }
       }
   }
}
```

Taken from Goetz, Java Concurrency in Practice, 2006, Addison-Wesley