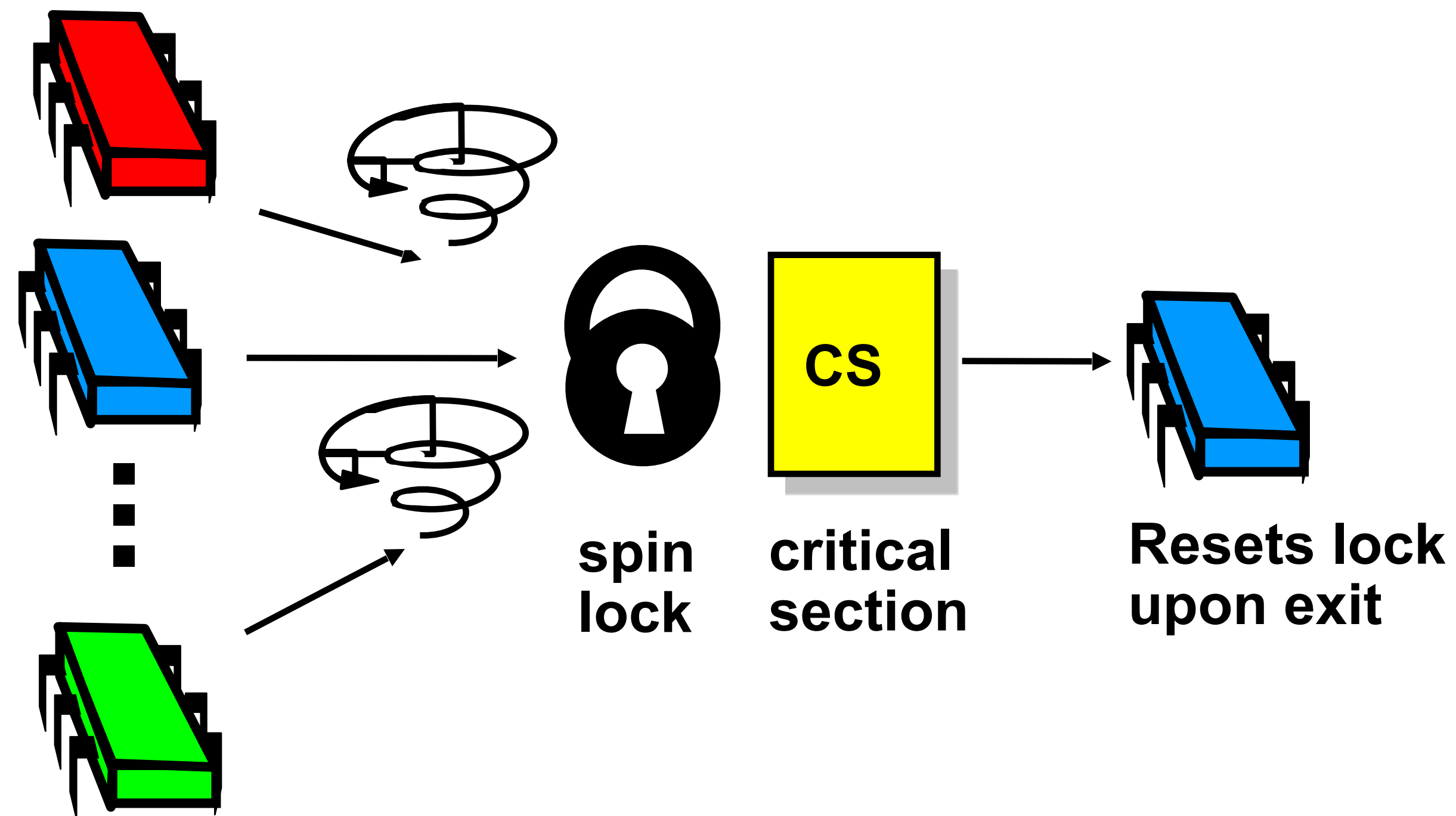


YSC3248: Parallel, Concurrent and Distributed Programming

Concurrent Linked Lists
Part I

Previous Lectures: Spin-Locks



Today: More Concurrent Objects

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

Today: More Concurrent Objects

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

Coarse-Grained Synchronization

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

Coarse-Grained Synchronization

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

Coarse-Grained Synchronization

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

Coarse-Grained Synchronization

- Sequential bottleneck
 - Threads “stand in line”

Coarse-Grained Synchronization

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

Coarse-Grained Synchronization

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

This Lecture

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

This Lecture

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

First:

Fine-Grained Synchronization

- Instead of using a single lock ...

First:

Fine-Grained Synchronization

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

First:

Fine-Grained Synchronization

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

Second: Optimistic Synchronization

- Search without locking ...

Second: Optimistic Synchronization

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

Second: Optimistic Synchronization

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

Third: Lazy Synchronization

- Postpone hard work

Third:

Lazy Synchronization

- Postpone hard work
- Removing components is tricky

Third:

Lazy Synchronization

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

Third:

Lazy Synchronization

- Postpone hard work
- Removing components is tricky
 - Logical removal
 - Mark component to be deleted
 - Physical removal
 - Do what needs to be done

Fourth:

Lock-Free Synchronization

- Don't use locks at all
 - Use `compareAndSet()` & relatives ...

Fourth:

Lock-Free Synchronization

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

Fourth:

Lock-Free Synchronization

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

Linked List

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

Set Interface

- Unordered collection of items

Set Interface

- Unordered collection of items
- No duplicates

Set Interface

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

Warm-up: Testing Concurrent Sets

List-Based Sets

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

List-Based Sets

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

Add item to set

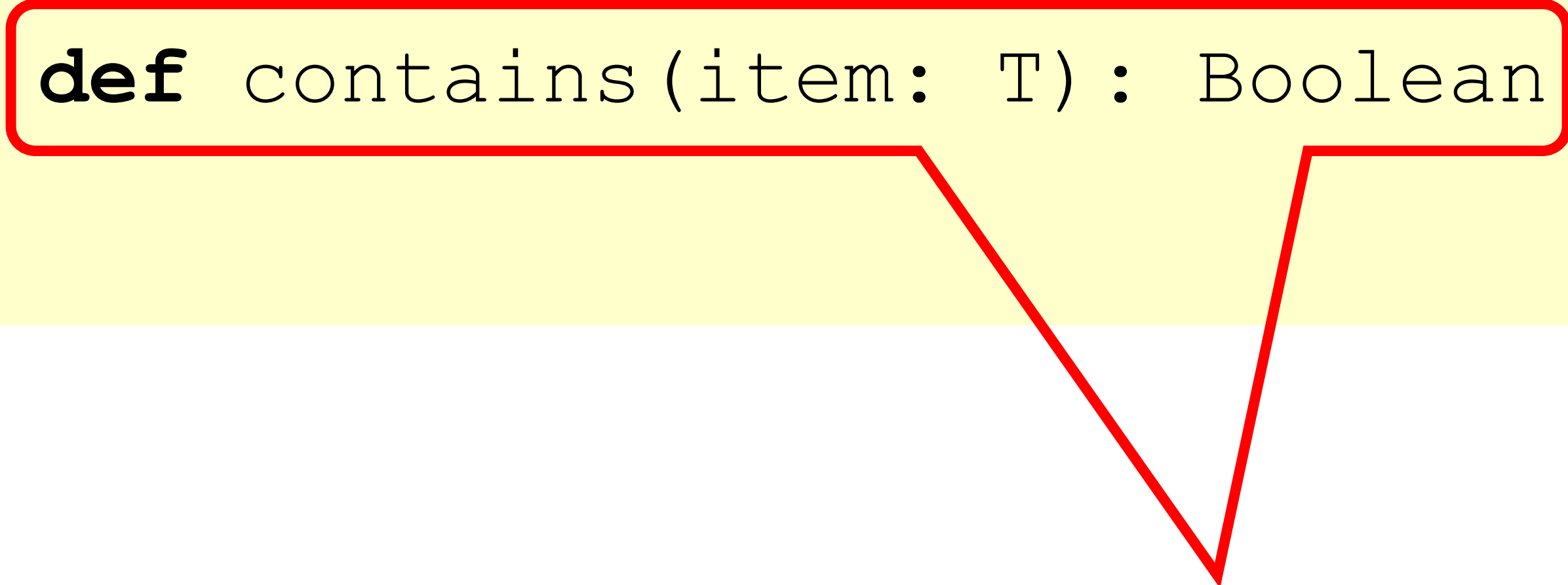
List-Based Sets

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

Remove item from set

List-Based Sets

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



Is item in set?

List Node

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

List Node

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

item of interest

List Node

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

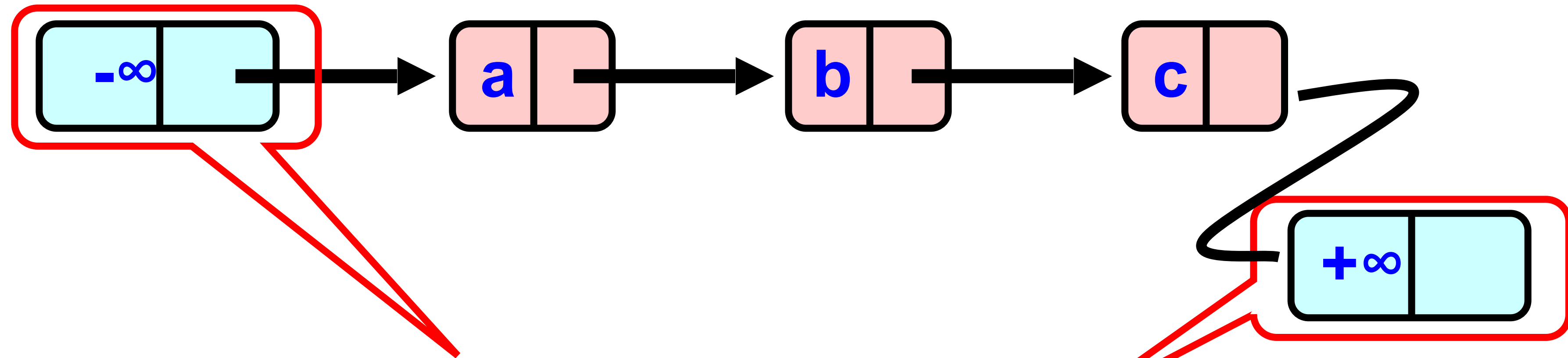
Usually hash code

List Node

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

Reference to next node

The List-Based Set



Sorted with Sentinel nodes
(min & max possible keys)

Reasoning about Concurrent Objects

- Invariant
 - Property that always holds

Reasoning about Concurrent Objects

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

Specifically ...

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

Specifically ...

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

Interference

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

Interference

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

Interference

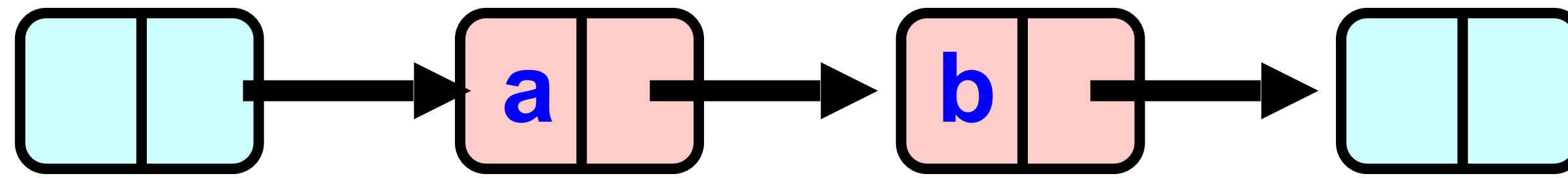
- Invariants make sense only if
 - methods considered
 - are the only modifiers
- Language encapsulation helps
 - List nodes *not visible* outside class
- Similar to *loop invariants*
 - Each method must preserve the invariant (same as each loop iteration)

Interference

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

Recap: Abstract Data Types

- Concrete representation:



- Abstract Type:
 $\{a, b\}$

Abstract Data Types

- Meaning of rep given by abstraction map

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

Representation Invariant

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

Blame Game

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

Blame Game

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

Blame Game

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

Lists' Rep Invariant (partly)

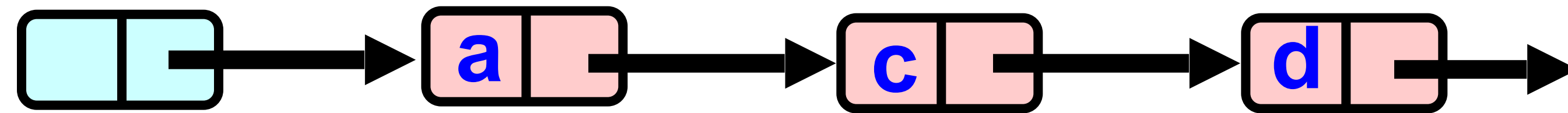
- Sentinel nodes
 - tail reachable from head
- Sorted
- No duplicates

Abstraction Map

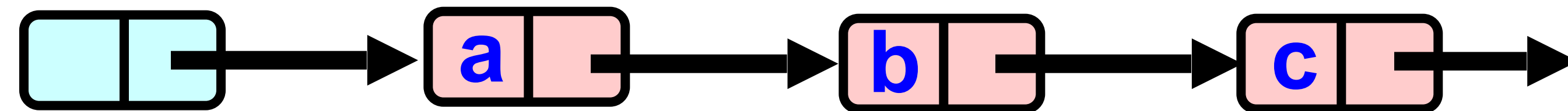
- $S(\text{head}) =$
 $\{ x \mid \text{there exists } a \text{ such that}$
 - $a \text{ reachable from head and}$
 - $a.\text{item} = x$ $\}$

Sequential List Based Set

`add()`

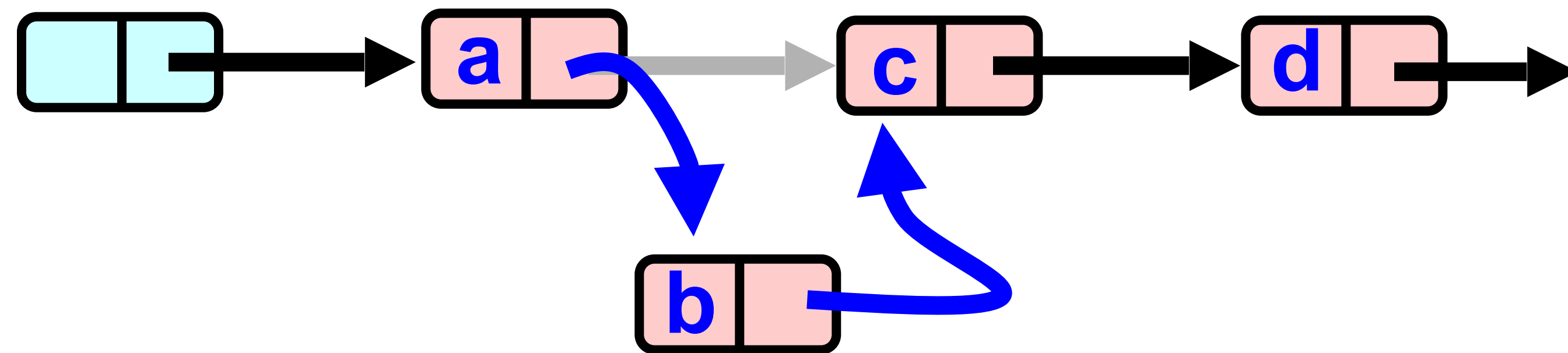


`remove()`

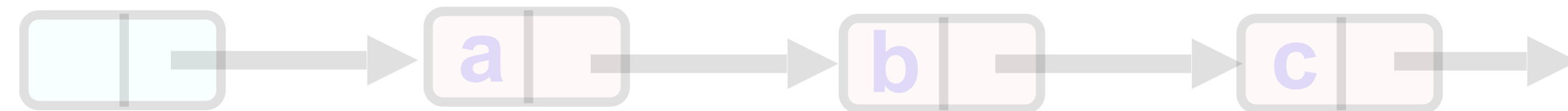


Sequential List Based Set

add()

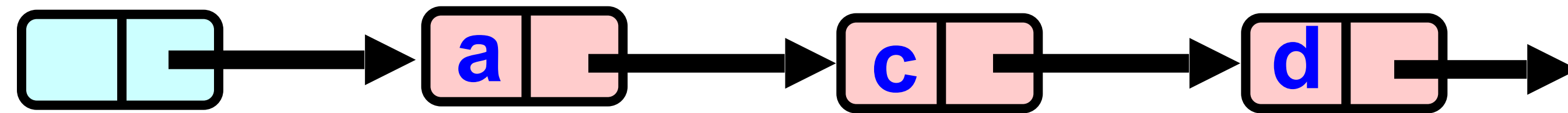


remove()

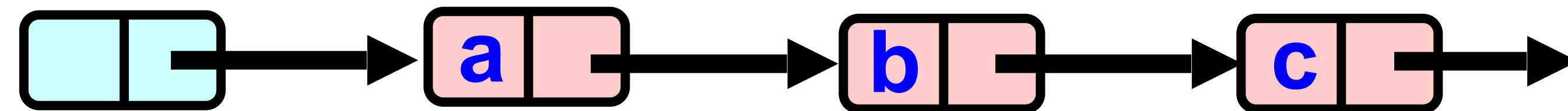


Sequential List Based Set

add()

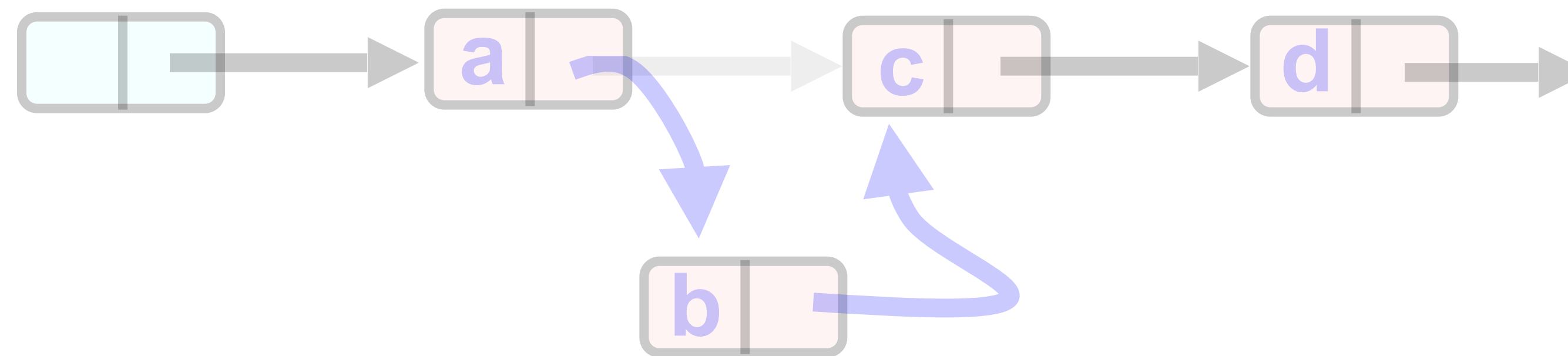


remove()

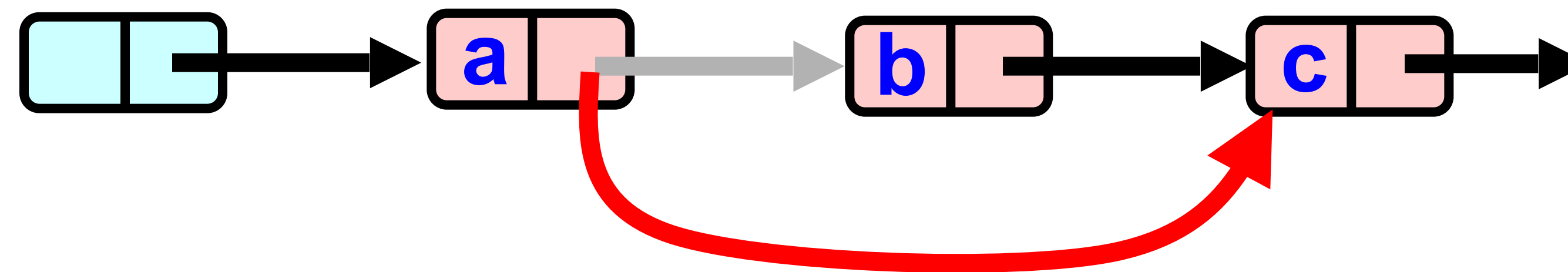


Sequential List Based Set

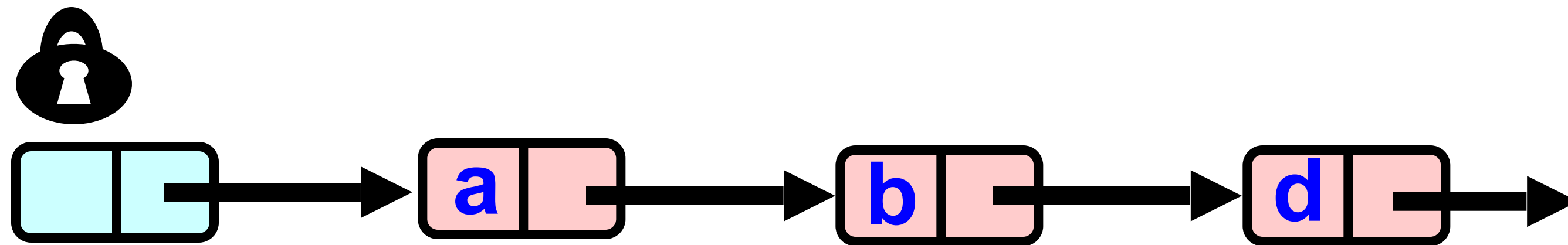
add()



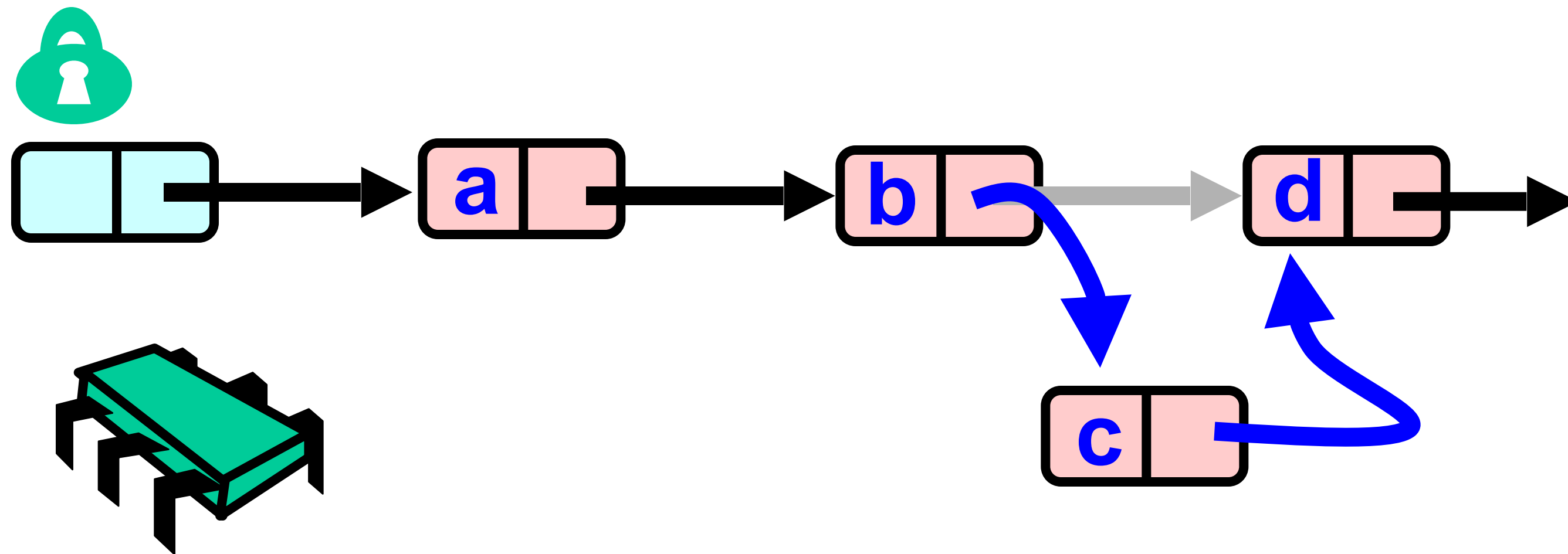
remove()



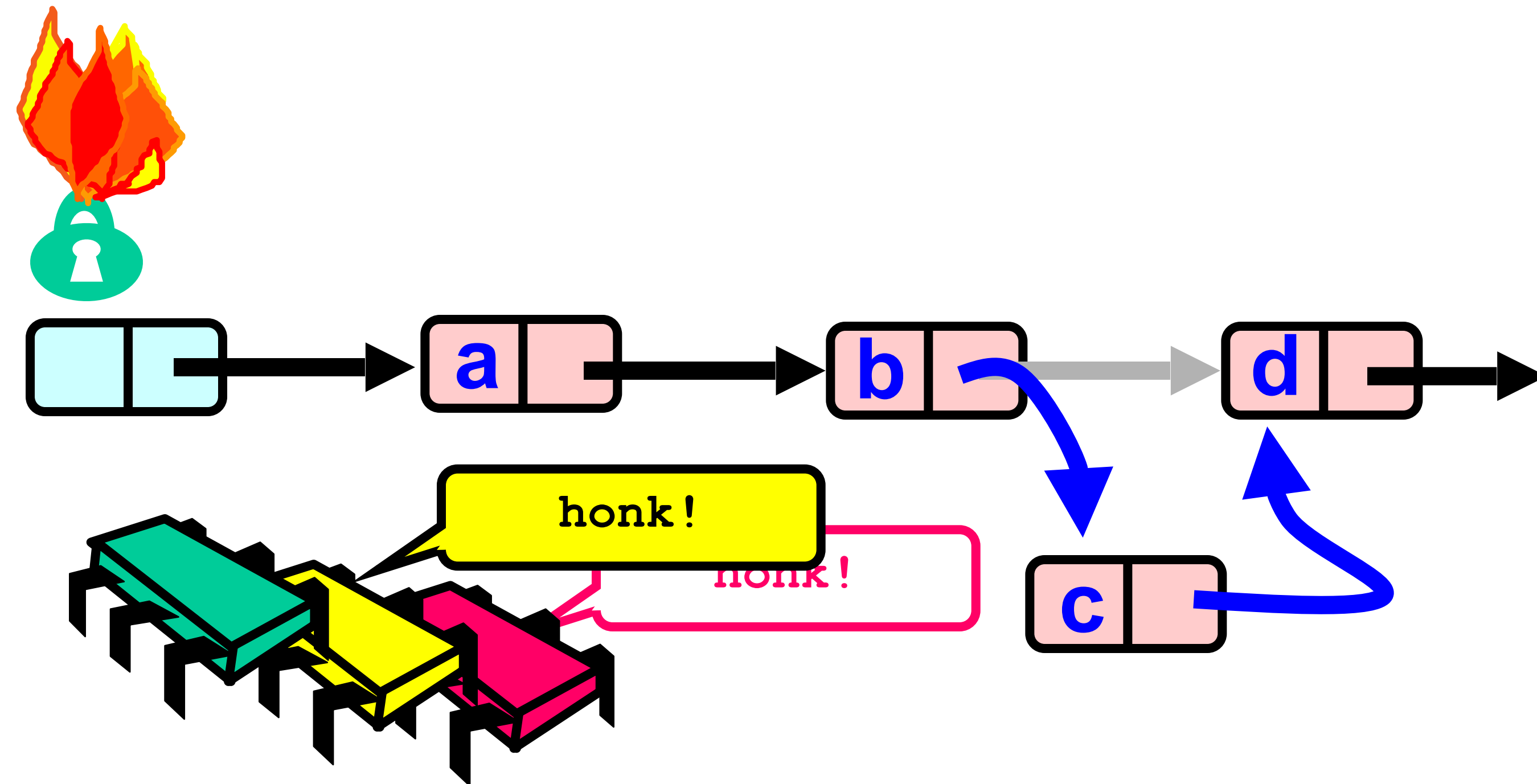
Coarse-Grained Locking



Coarse-Grained Locking



Coarse-Grained Locking



Simple but hotspot + bottleneck

Coarse-Grained Locking

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

Coarse-Grained Locking

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

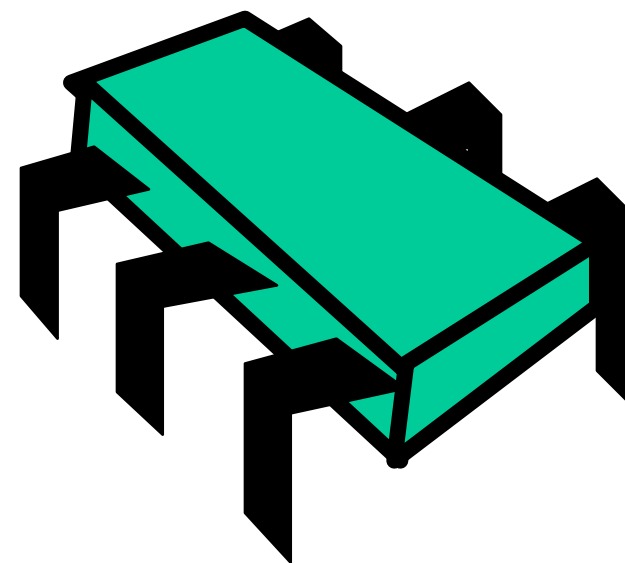
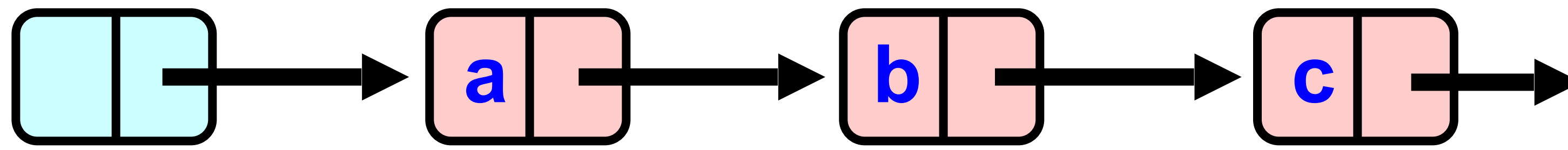
Fine-grained Locking

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

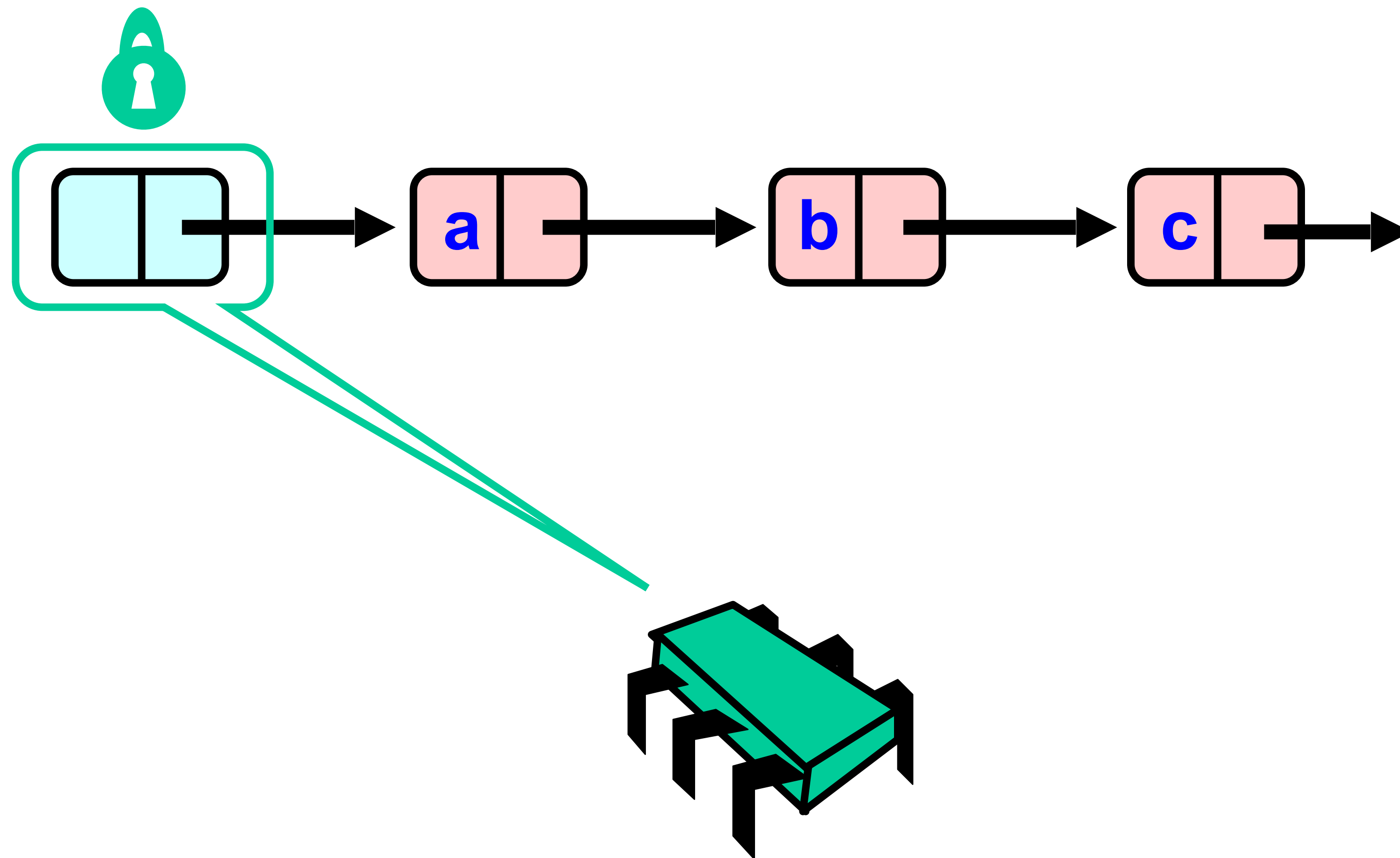
Fine-grained Locking

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

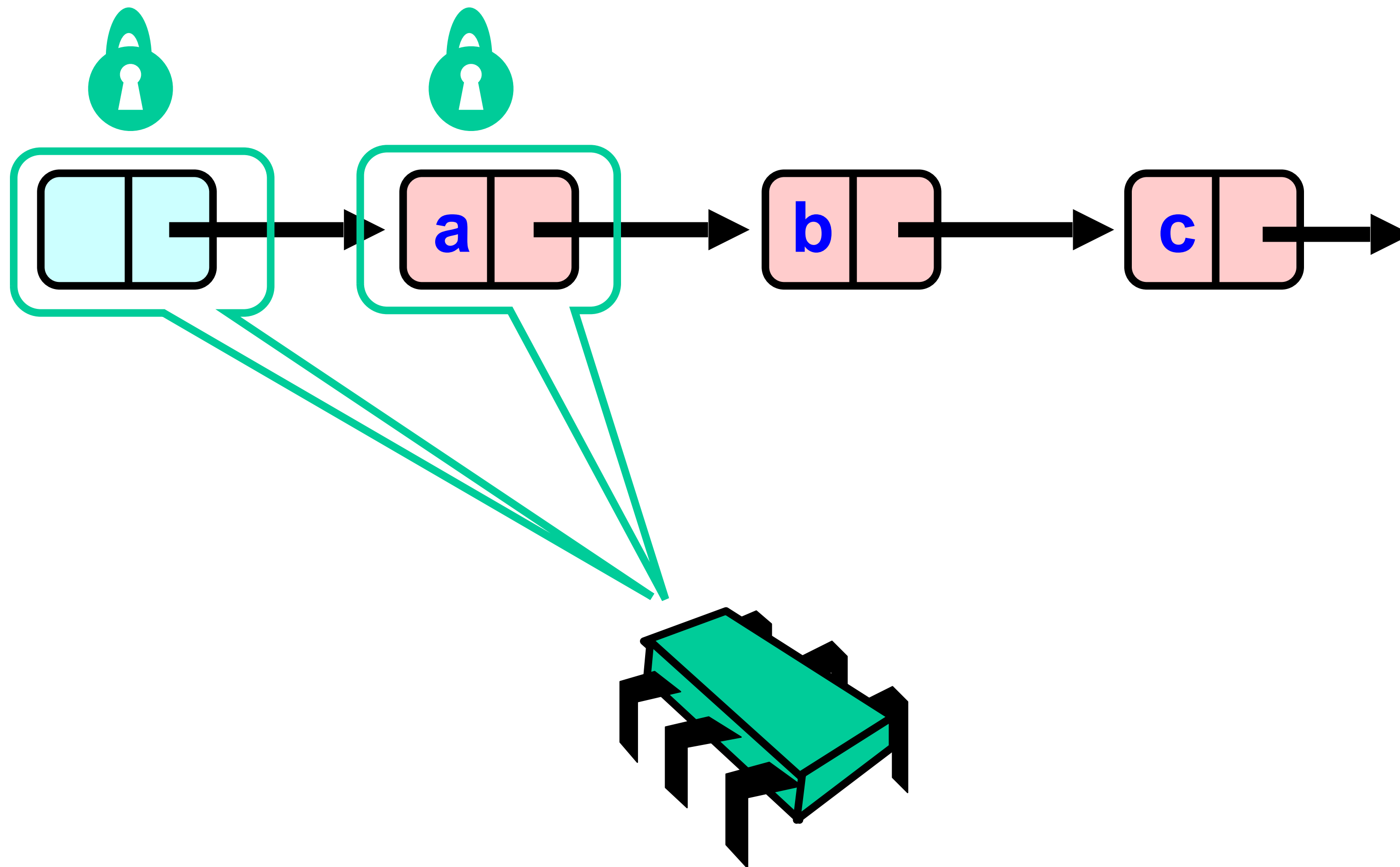
Hand-over-Hand locking



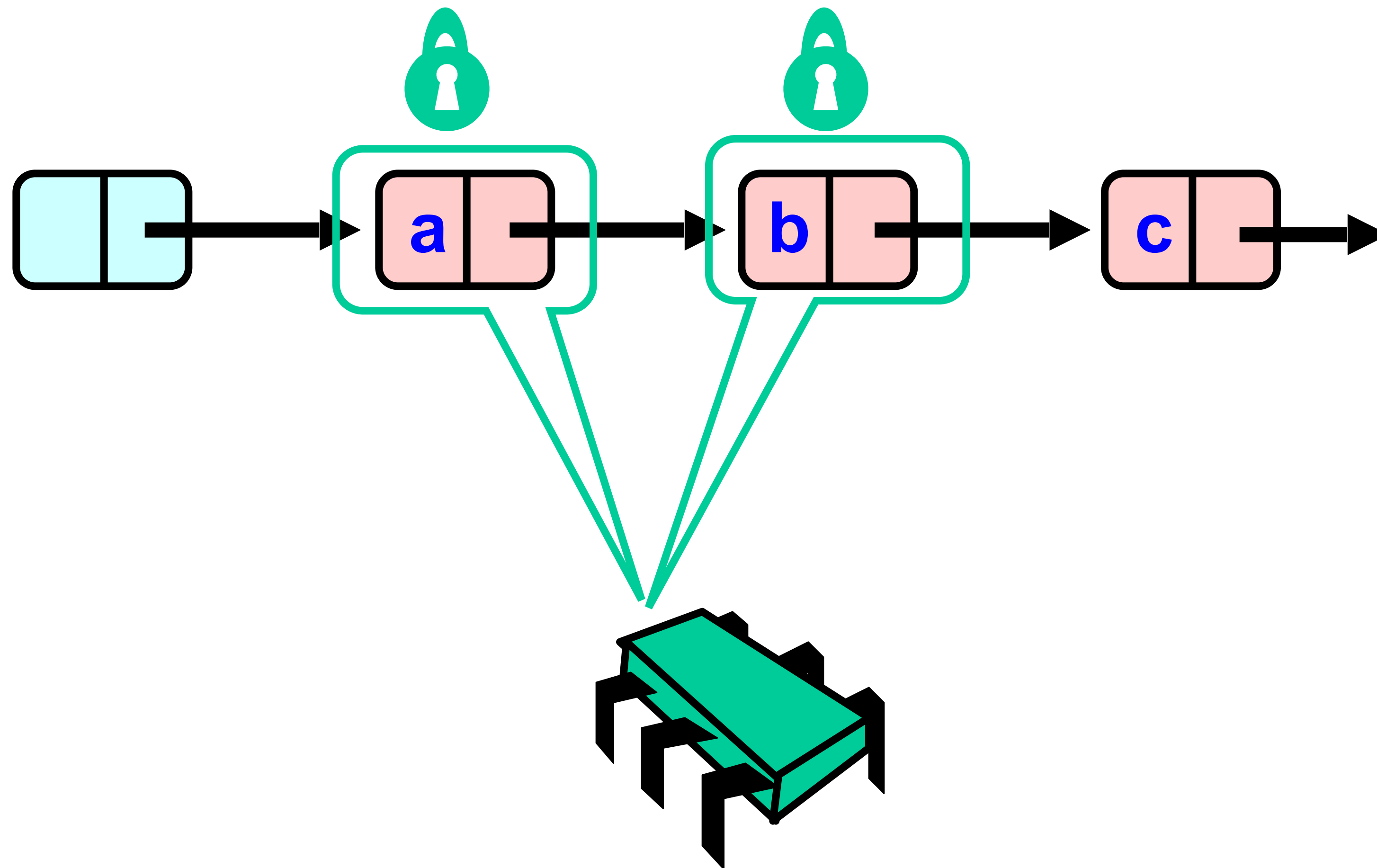
Hand-over-Hand locking



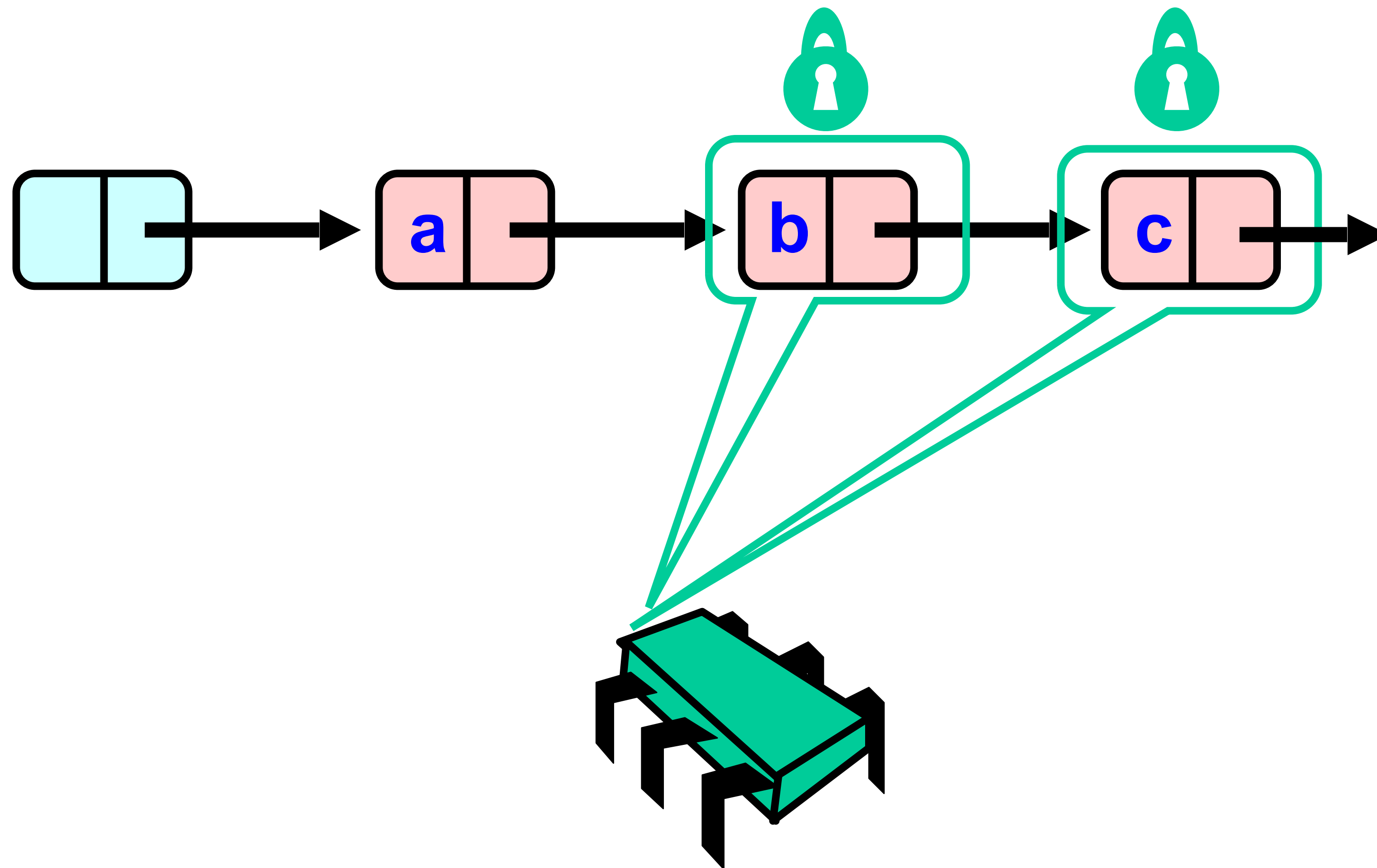
Hand-over-Hand locking



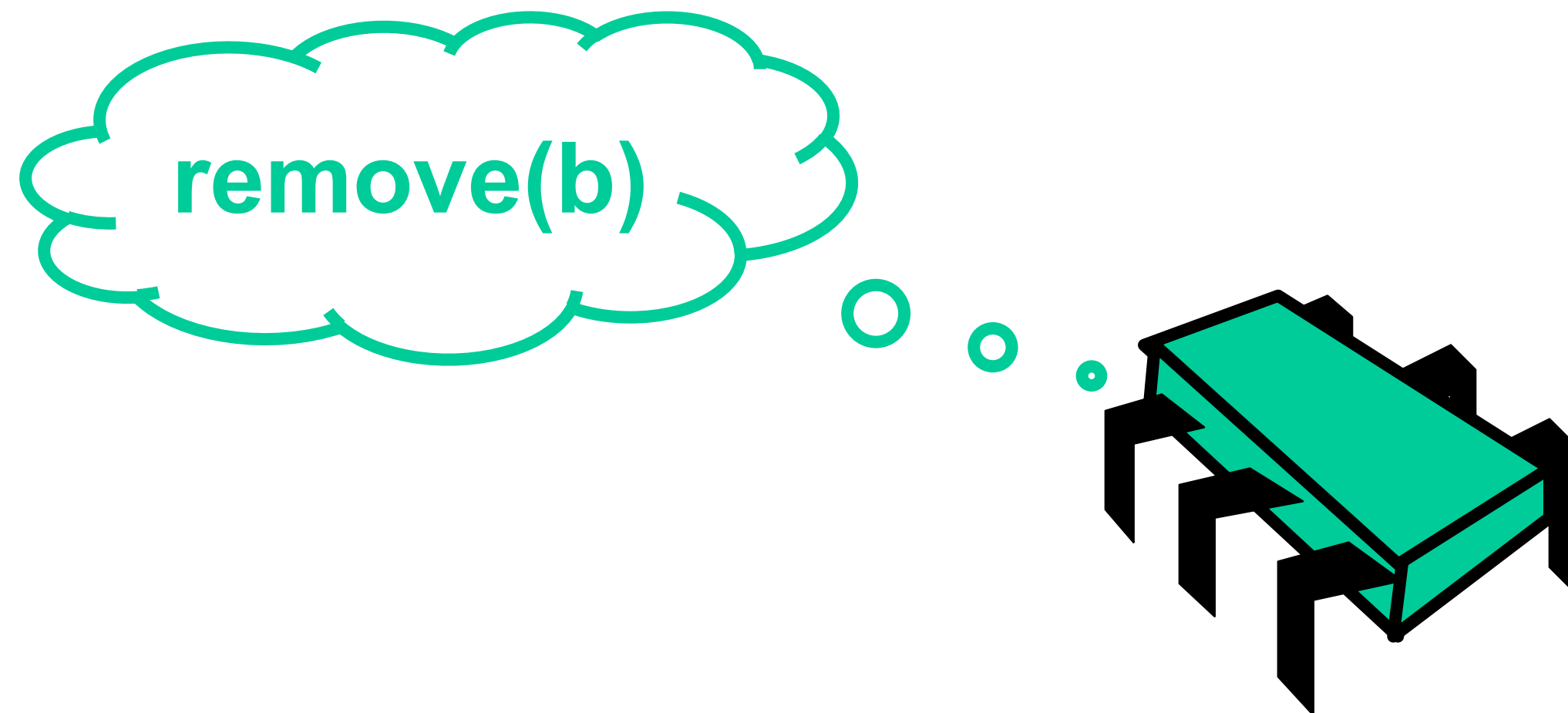
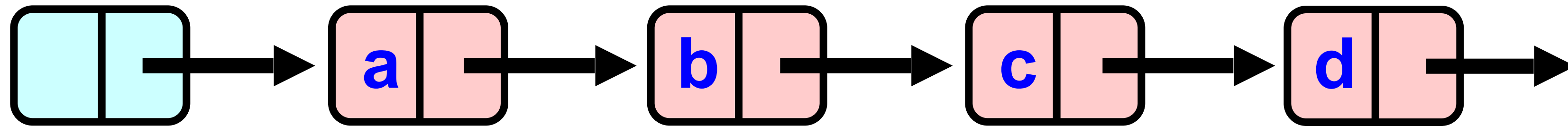
Hand-over-Hand locking



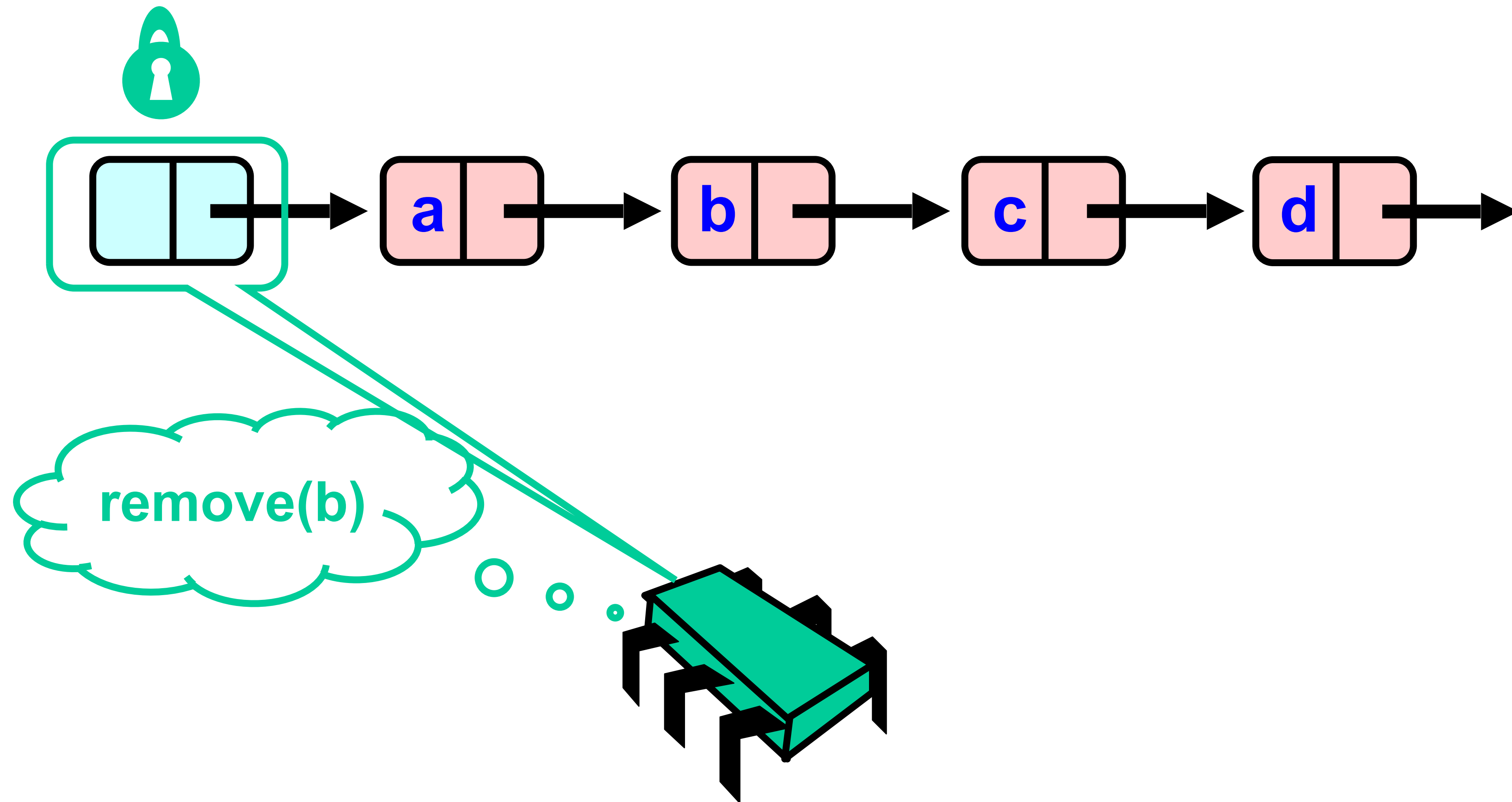
Hand-over-Hand locking



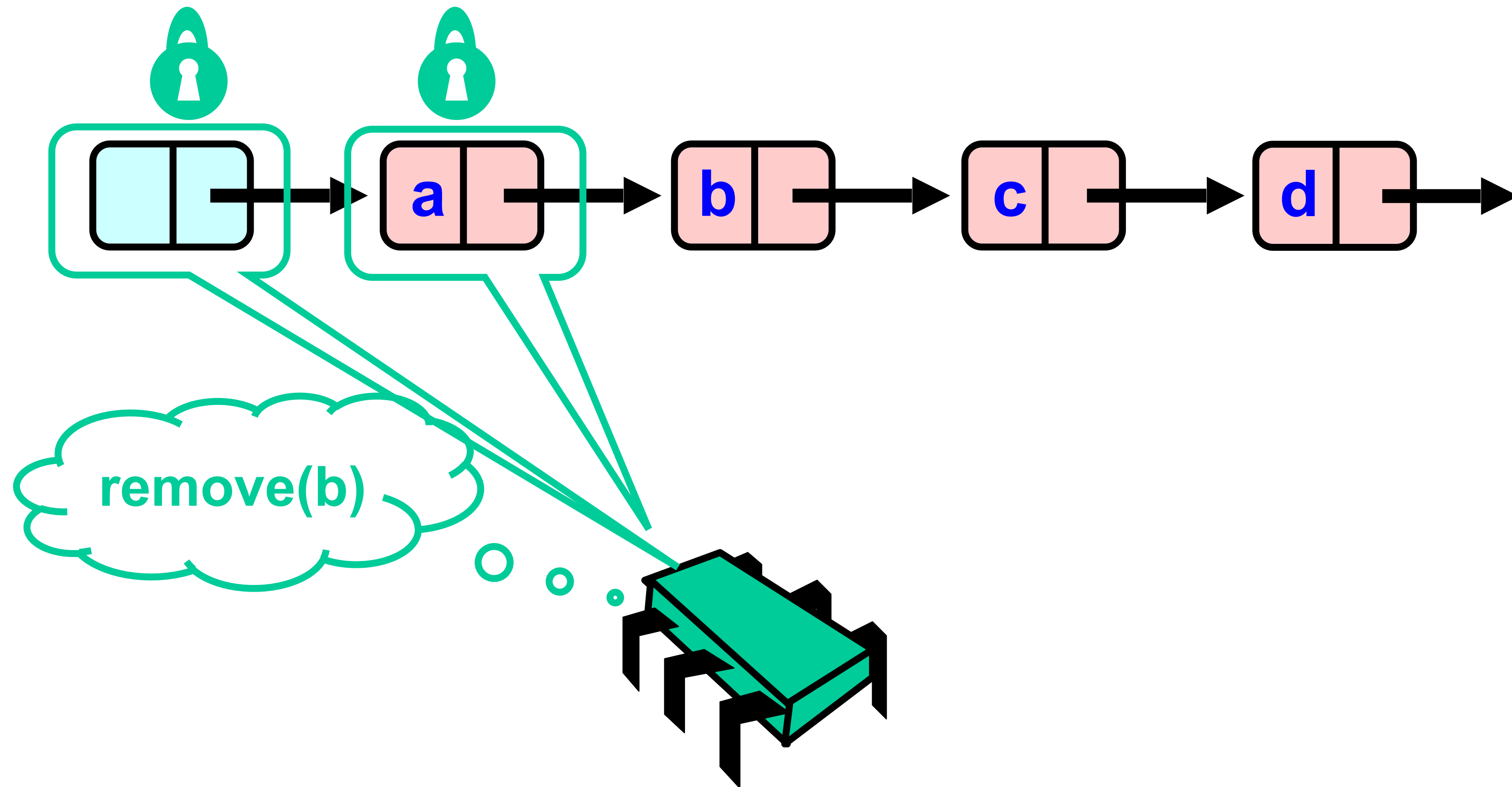
Removing a Node



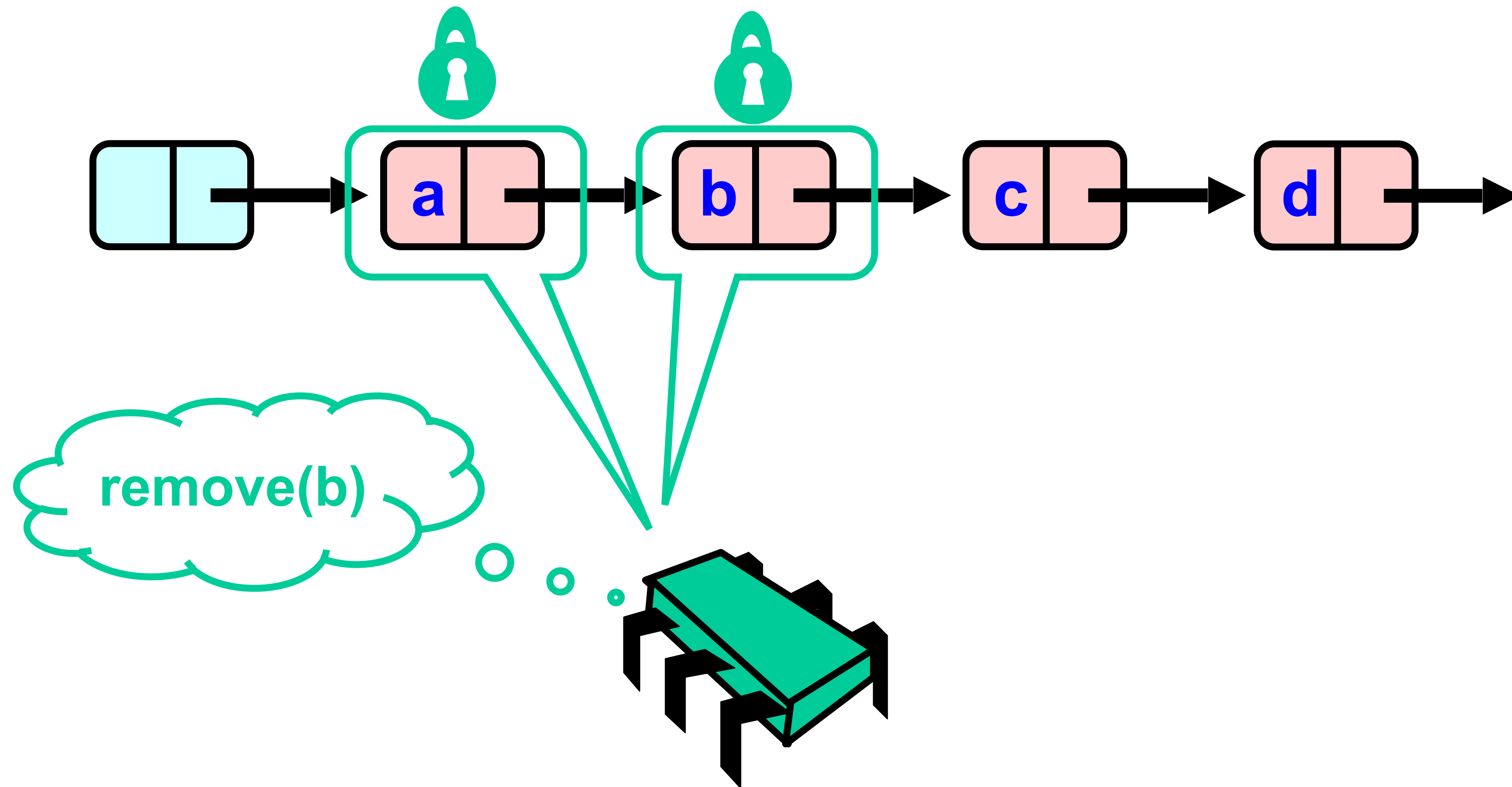
Removing a Node



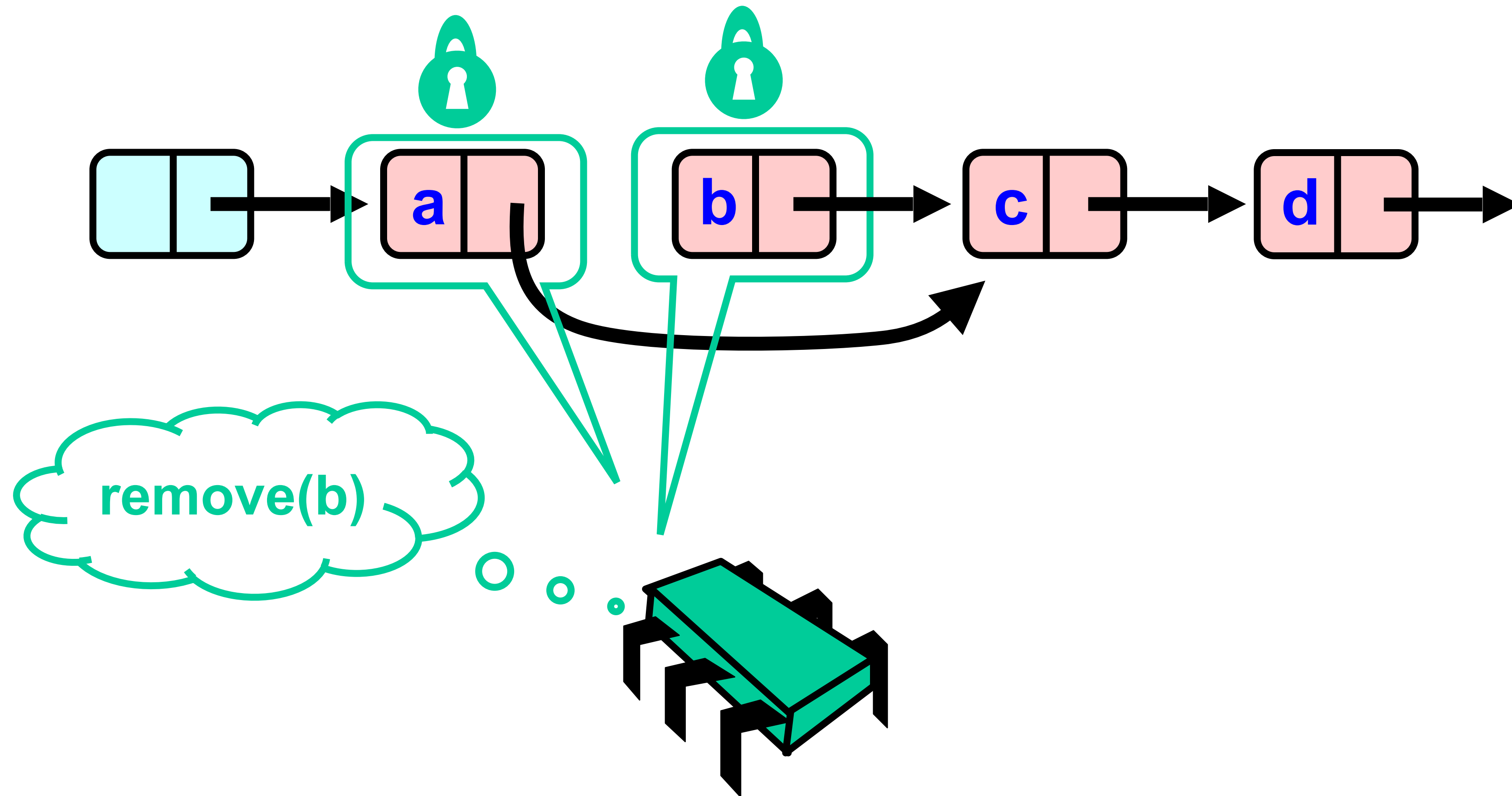
Removing a Node



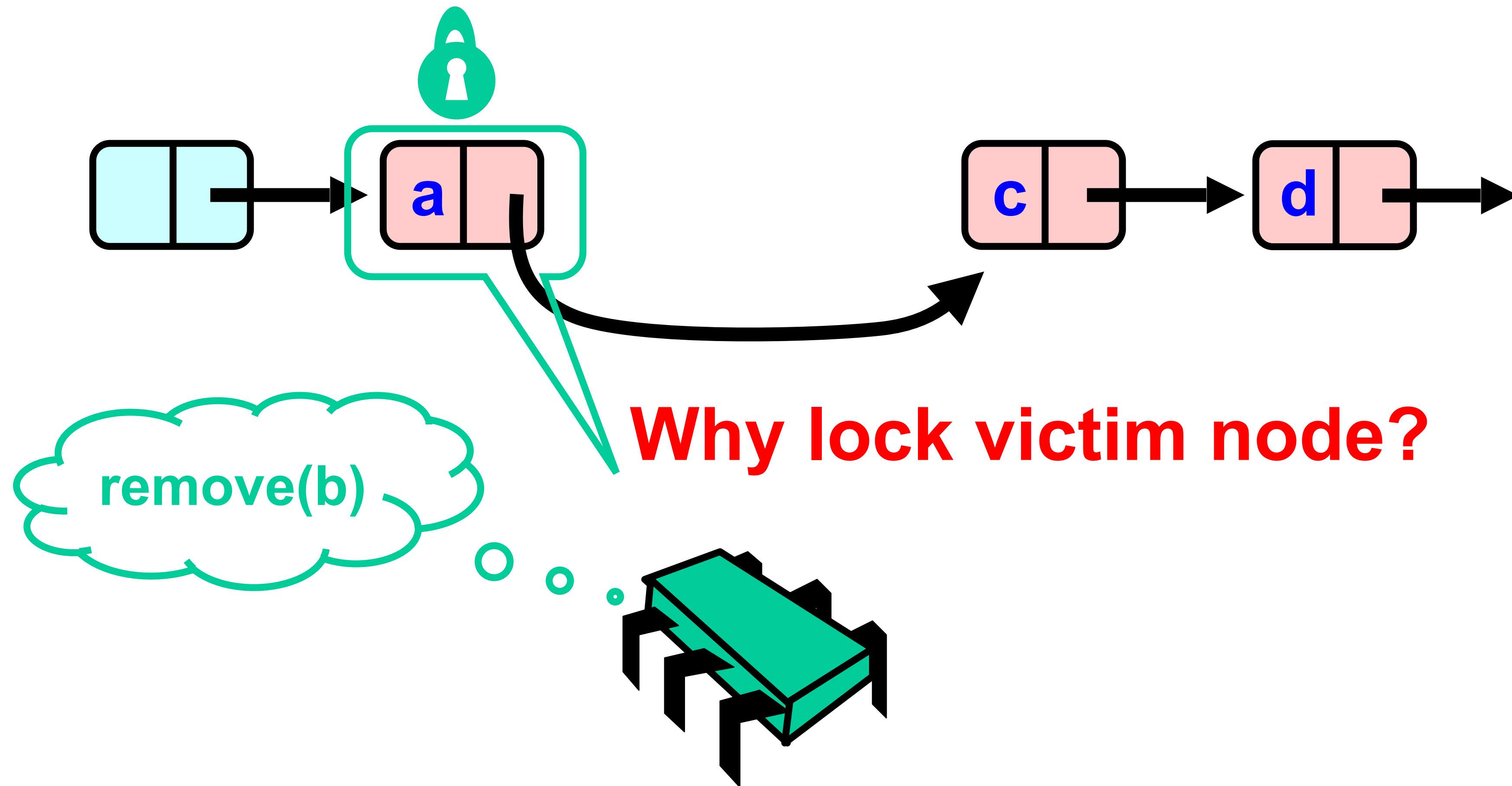
Removing a Node



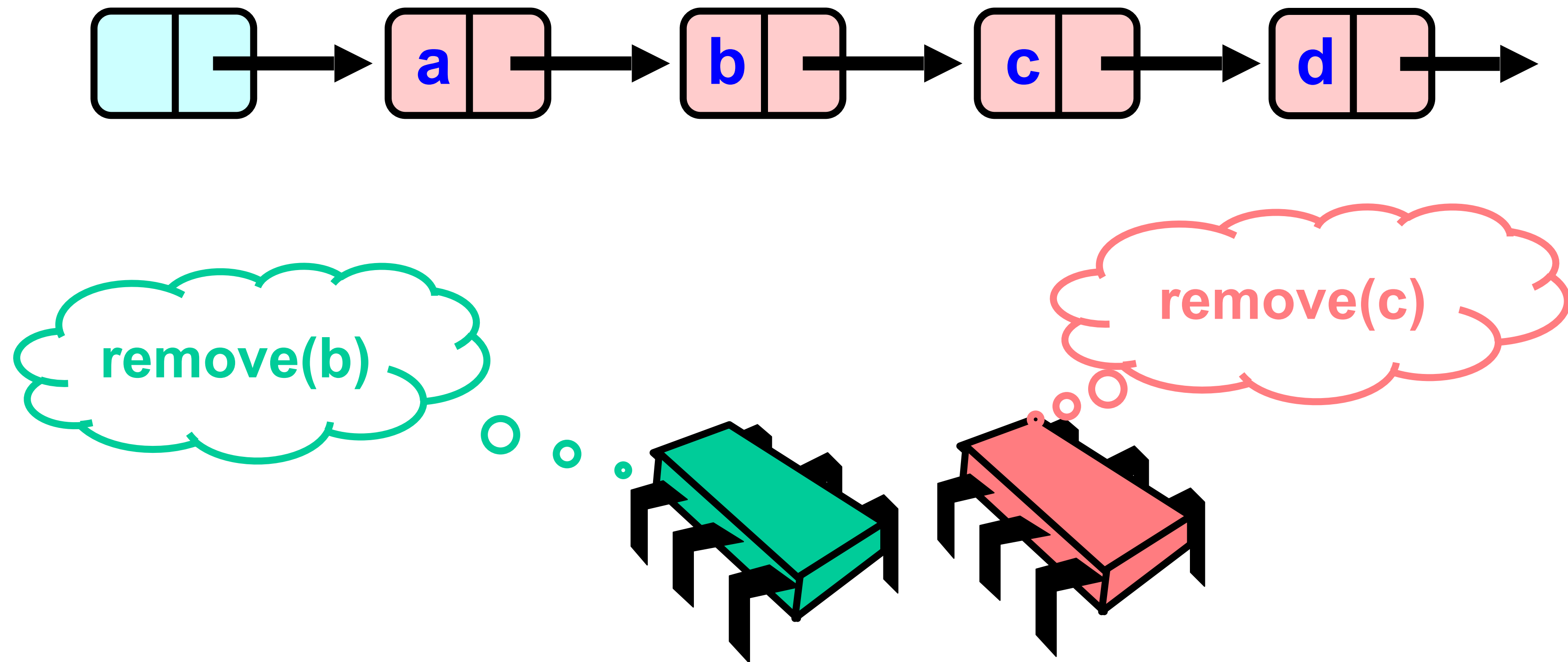
Removing a Node



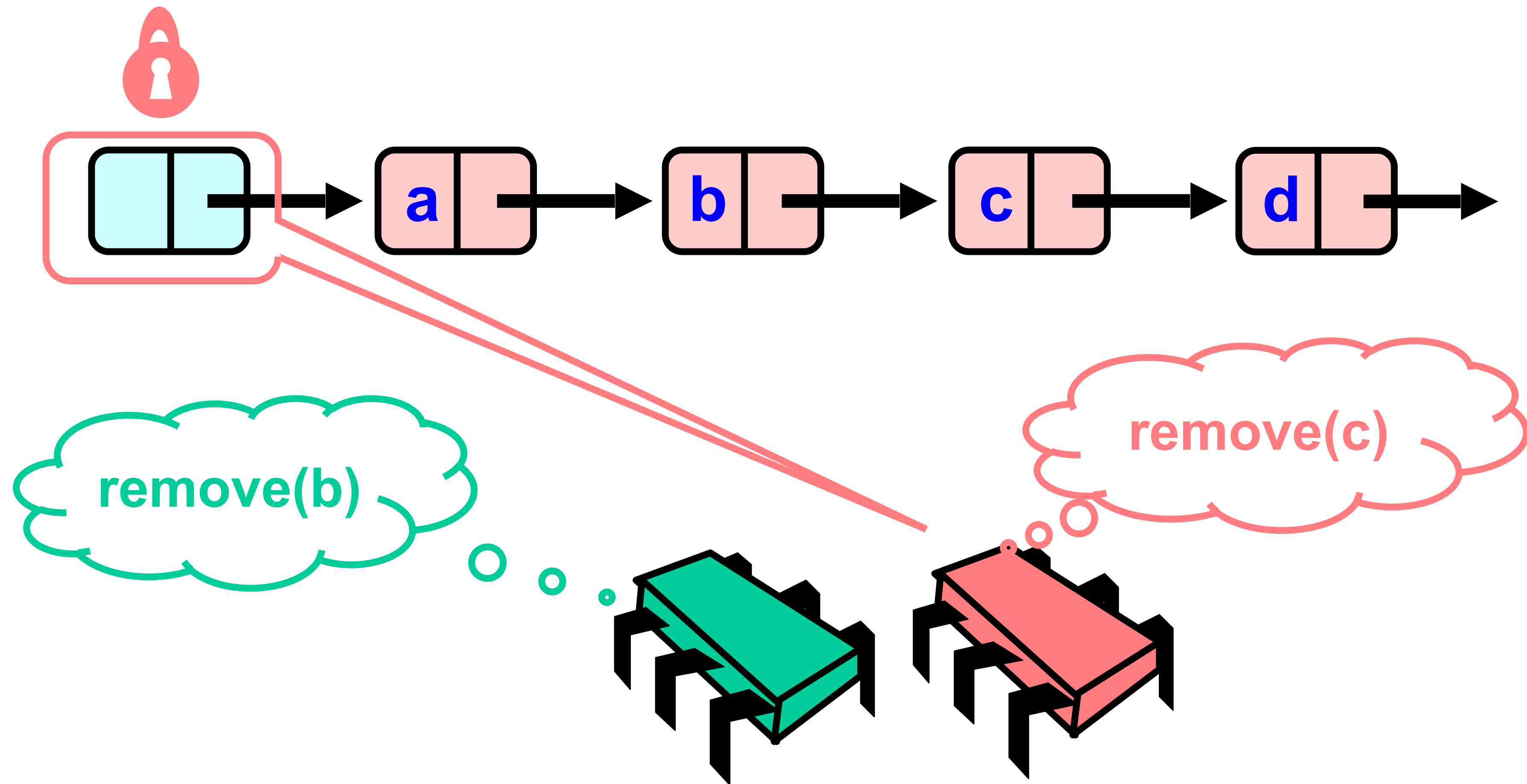
Removing a Node



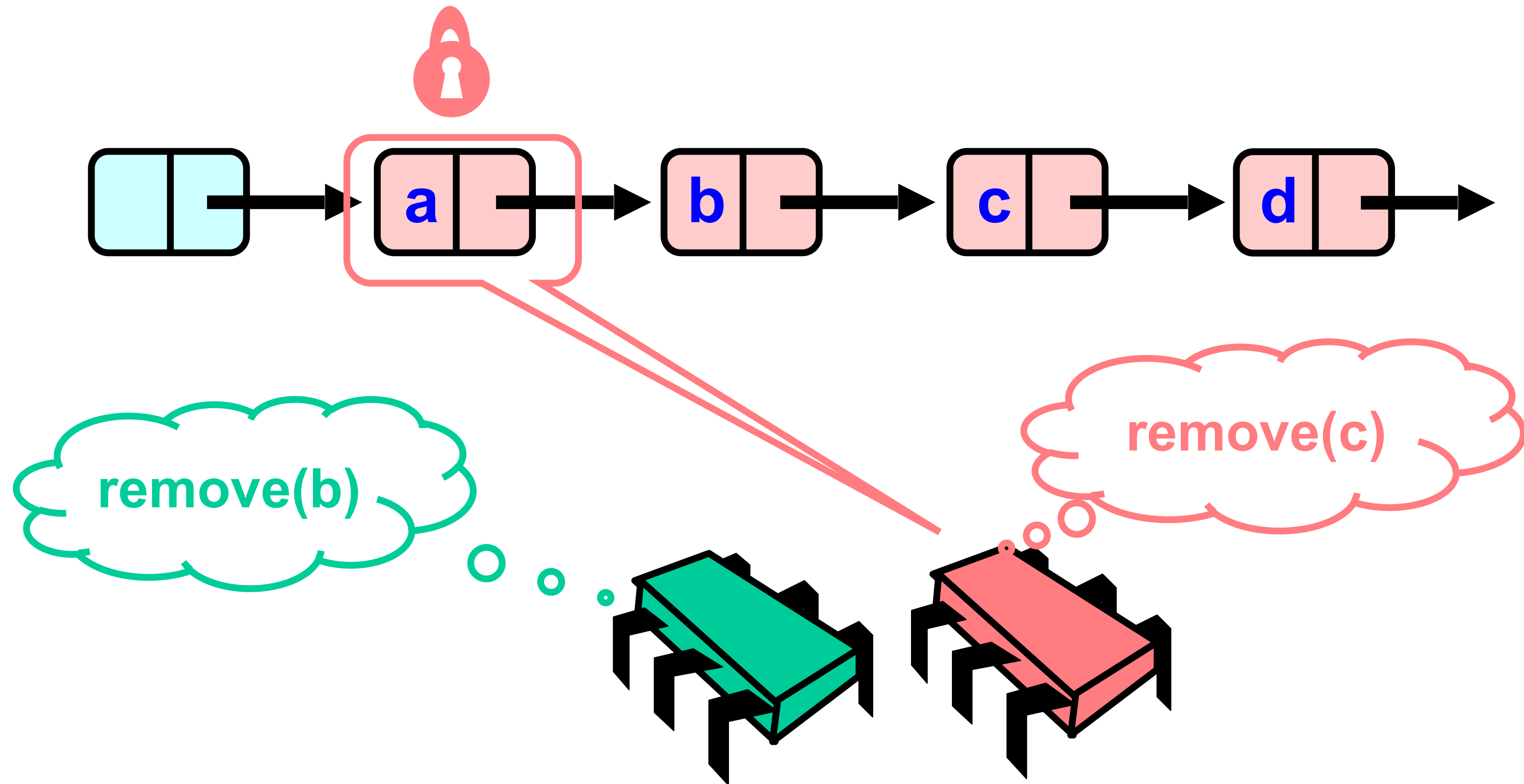
Concurrent Removes



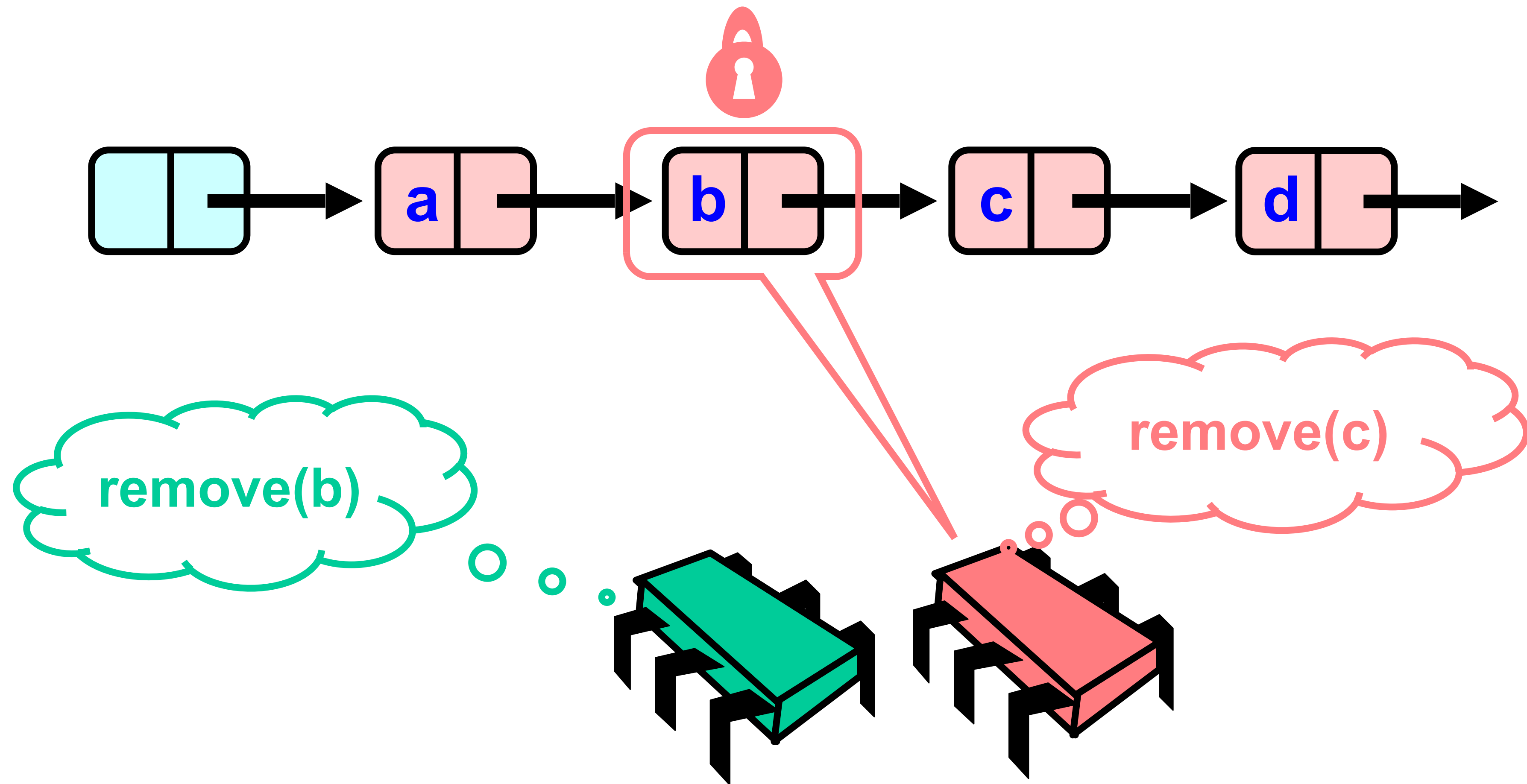
Concurrent Removes



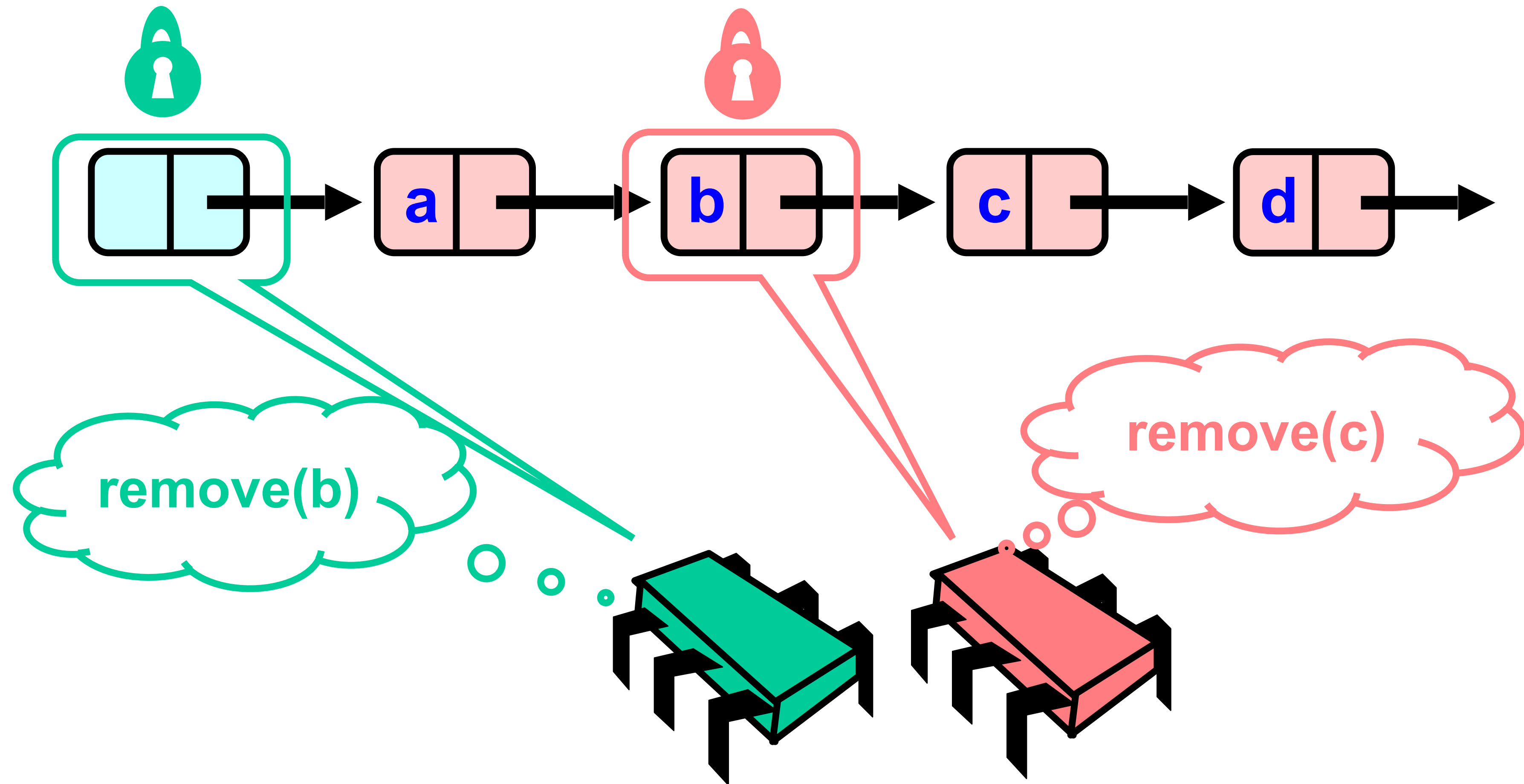
Concurrent Removes



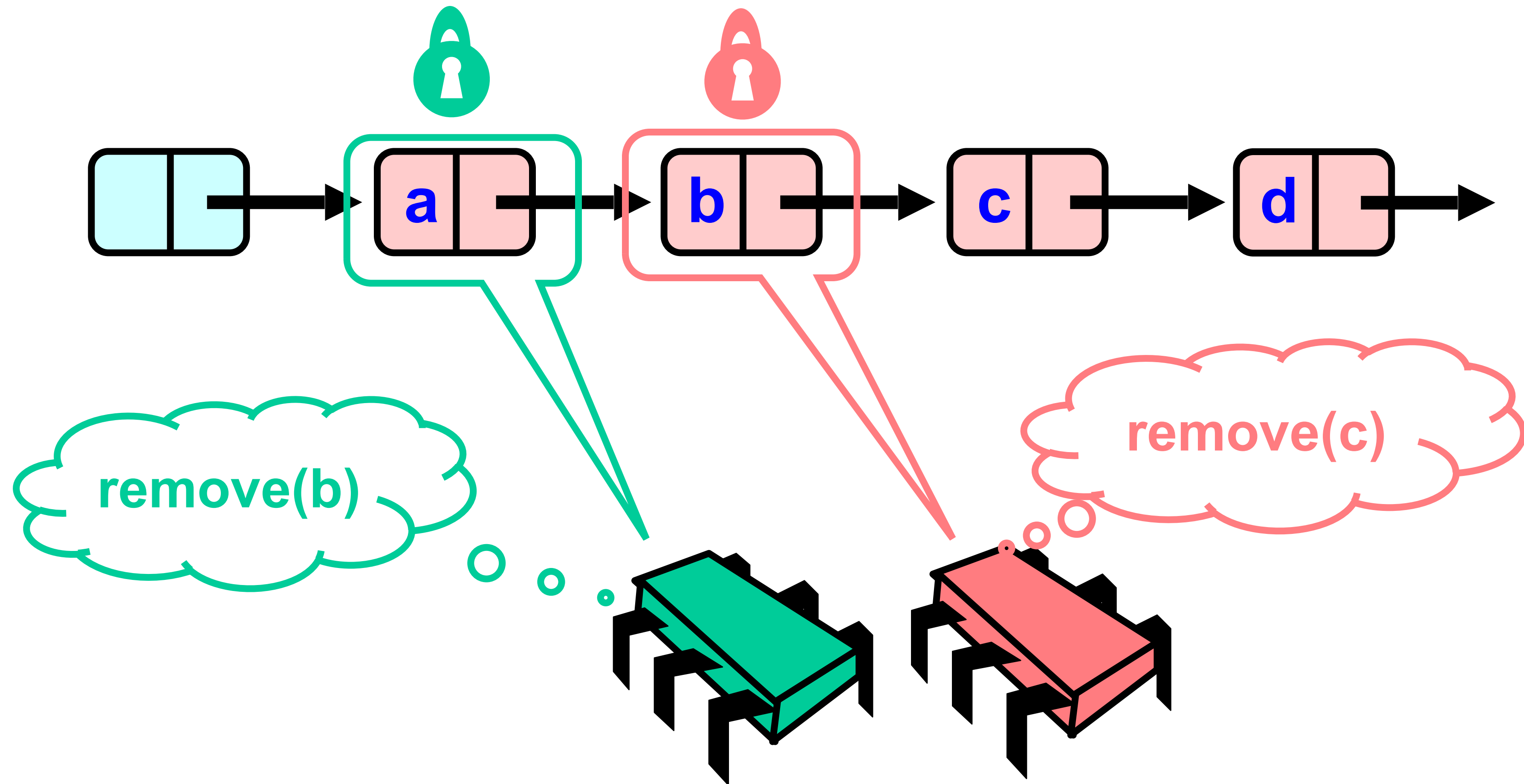
Concurrent Removes



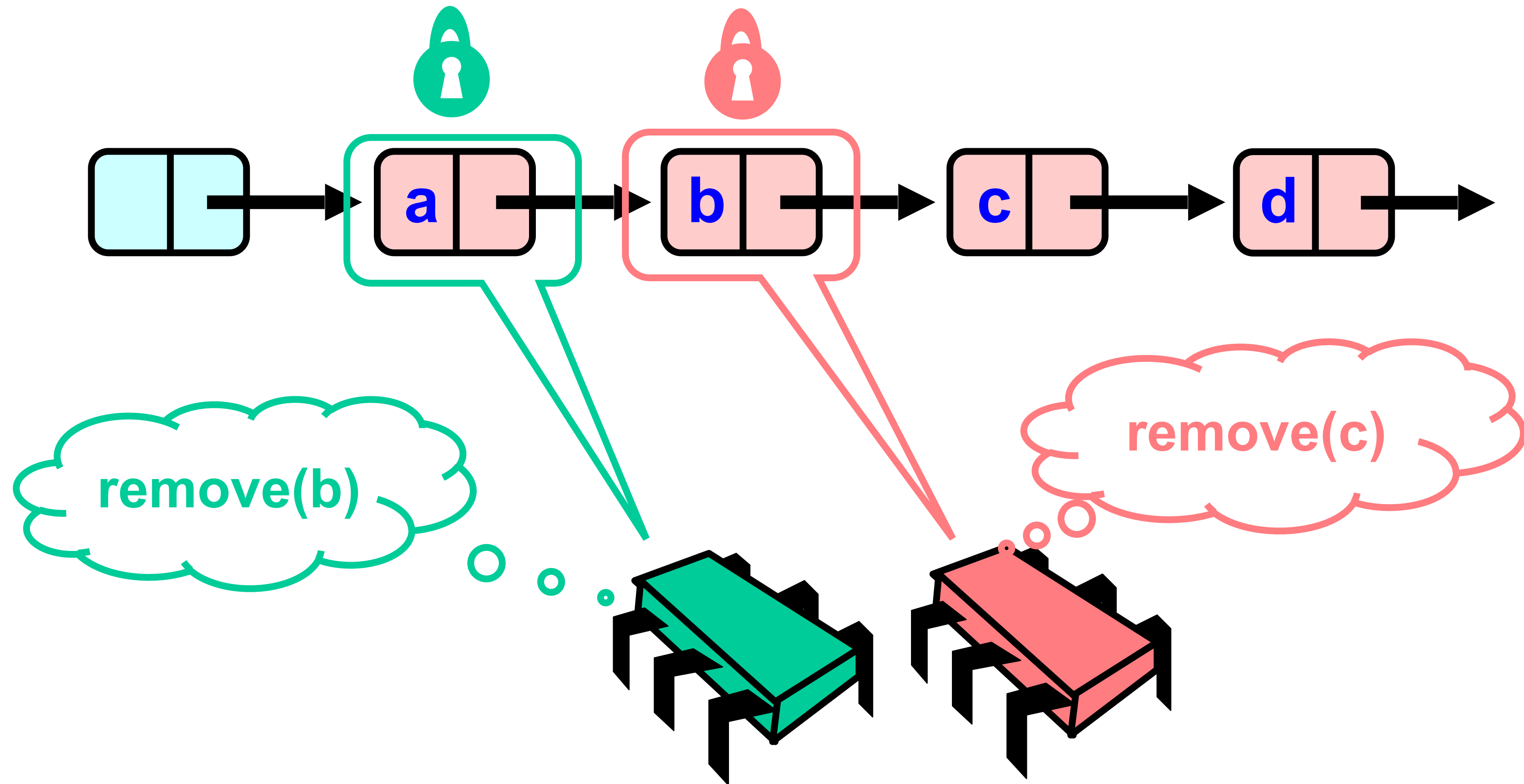
Concurrent Removes



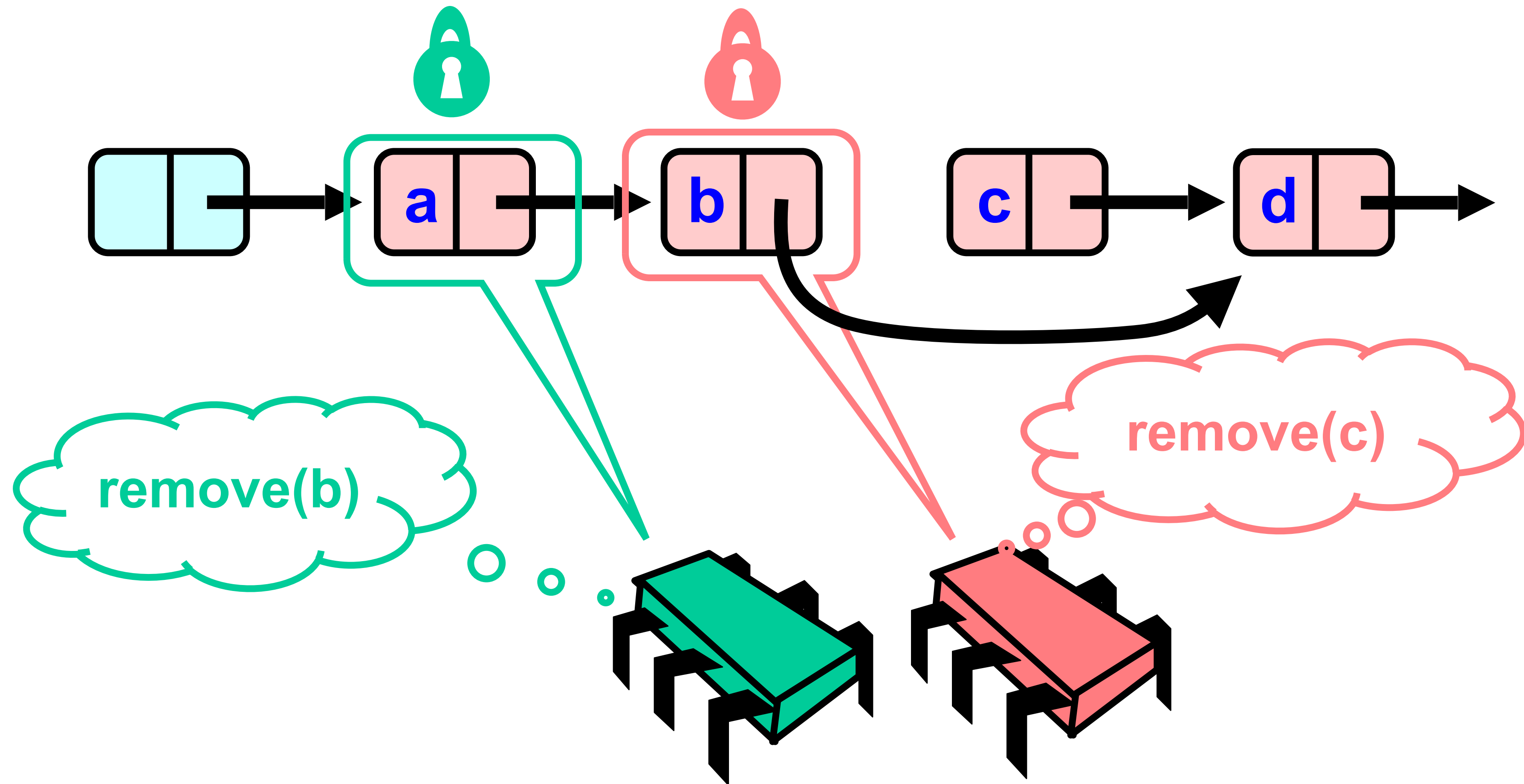
Concurrent Removes



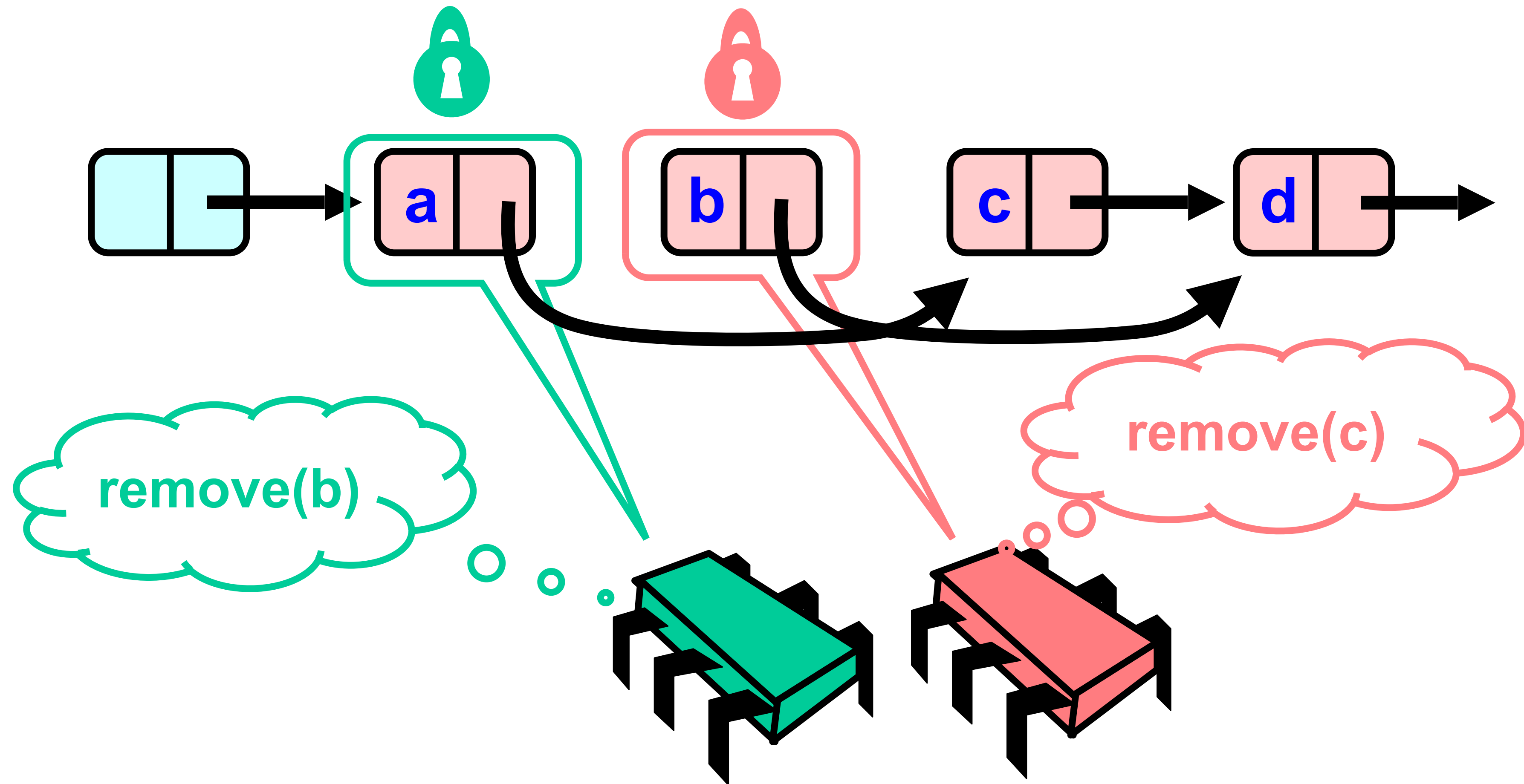
Concurrent Removes



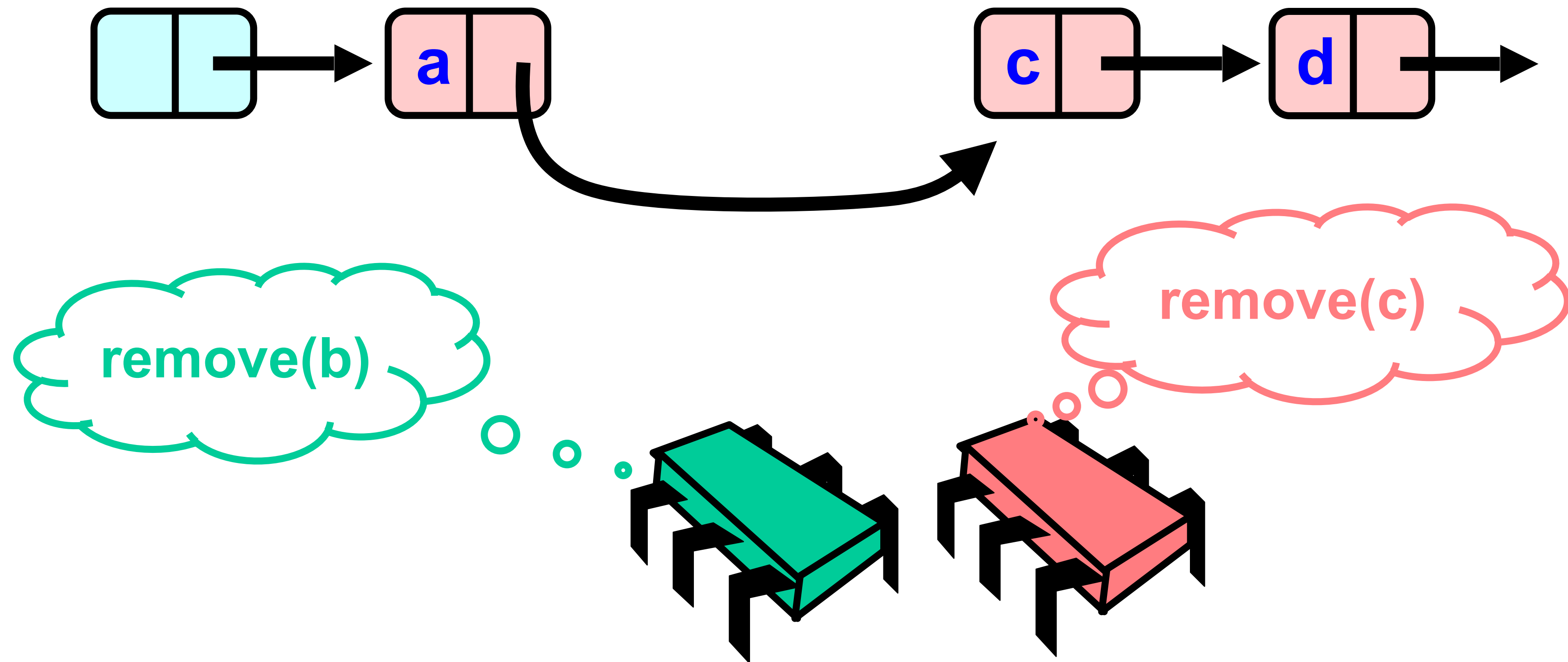
Concurrent Removes



Concurrent Removes

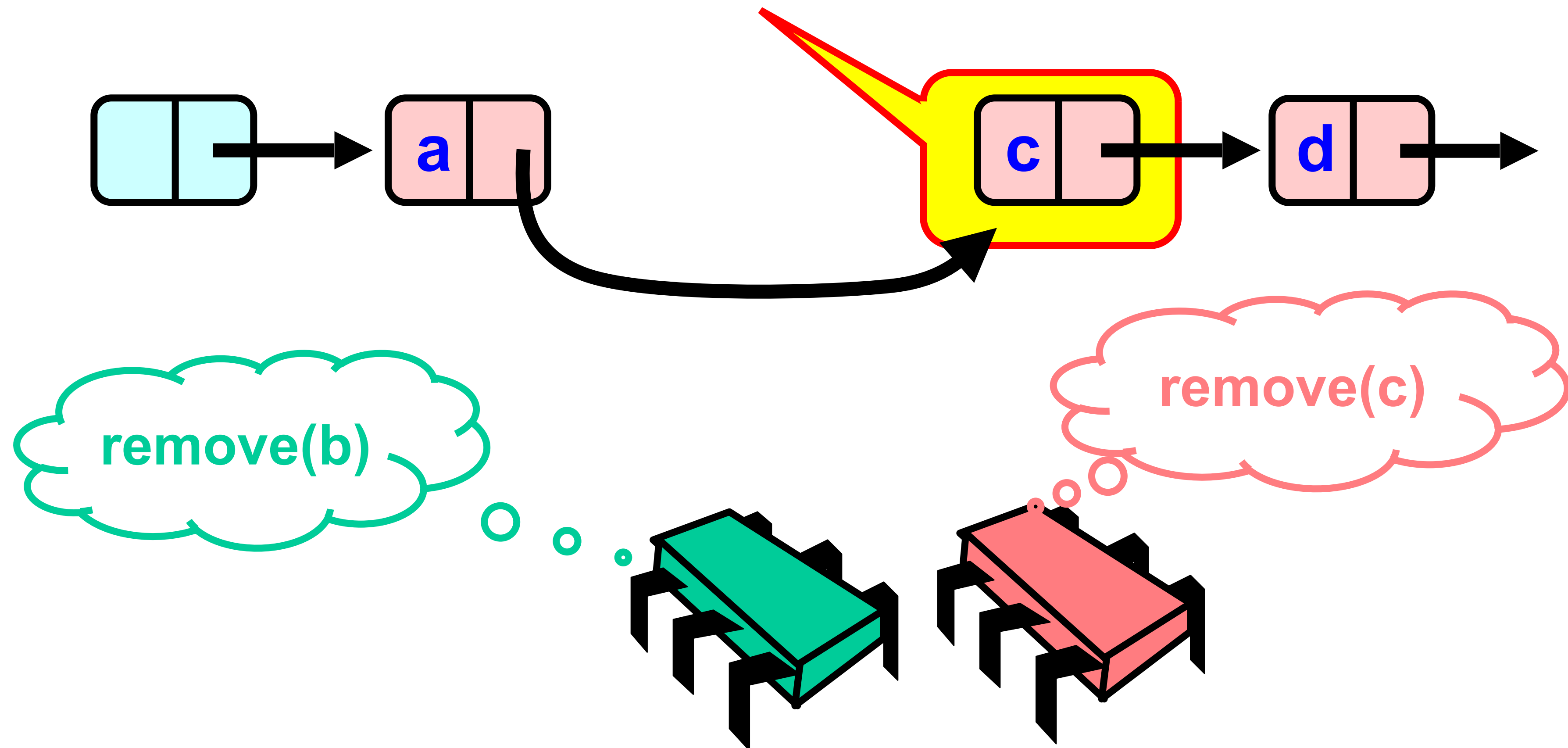


Uh, Oh



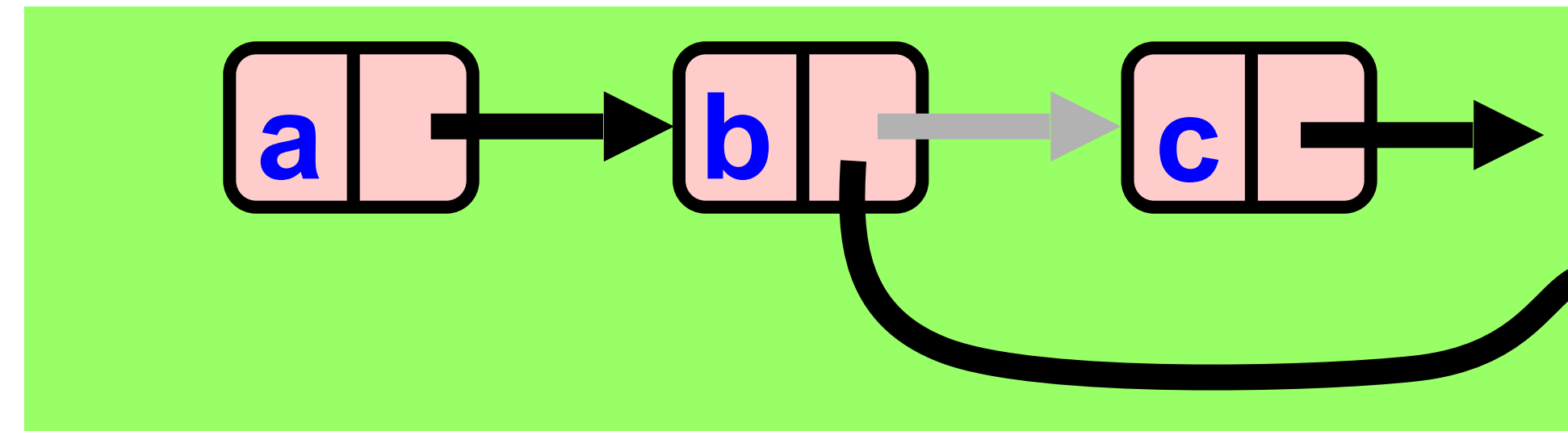
Uh, Oh

Bad news, c not removed

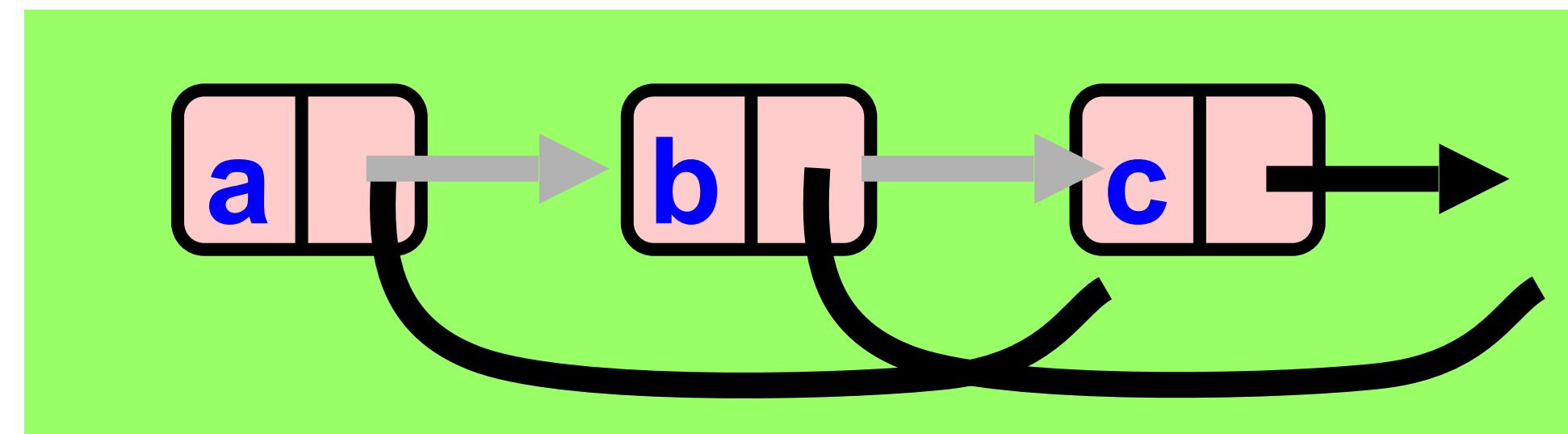


Problem

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



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



Hand-over-Hand Locking: Insight

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

Next Lecture:

Even less locking



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

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