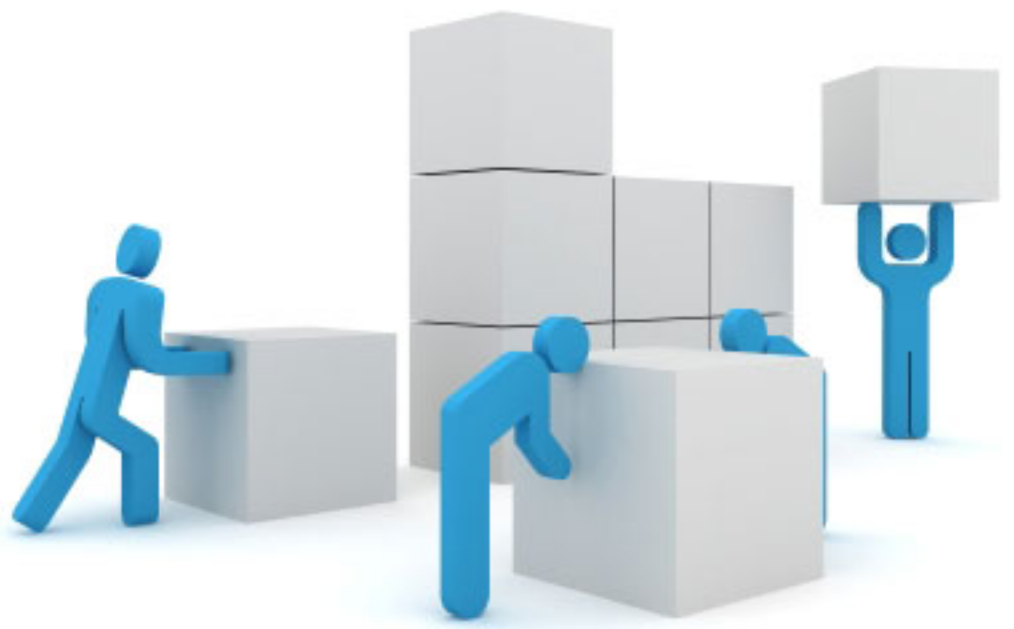
Linked Lists: The Role of Locking

Erez Petrank

Technion

Why Data Structures?

- Concurrent Data Structures are building blocks
  - Used as libraries
  - Construction principles apply broadly

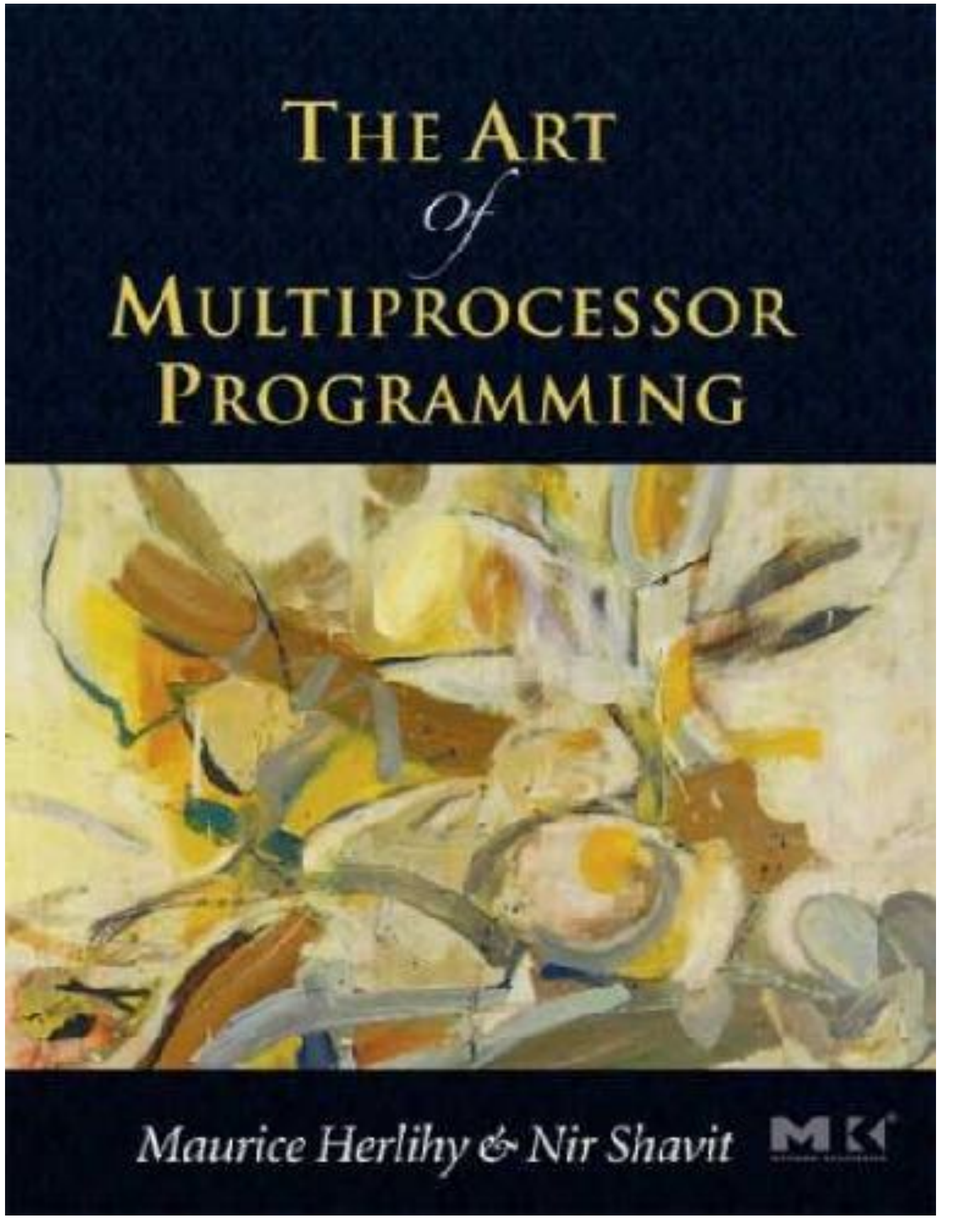


This Lecture

- Designing the first concurrent data structure: the linked-list
  - How do we use locks?
  - How do we achieve progress guarantees

Proper Credit

Several drawings are taken from the book, or from its accompanying slides.



Locking vs. Progress Guarantees

Counter Example

T1	T2
local := counter	local := counter
local++	local++
counter := local	counter := local

concurrent execution is not safe !

Use a Lock

```
Lock (L)
local := counter
local++
counter := local
Unlock(L)
```

Synchronization:
Only one thread can
acquire a lock L

Compare and Swap (CAS)

- CAS (addr, expected, new)

Atomically:

```
If ( MEM[addr] == expected ) {
        MEM[addr] = new
        return (TRUE)
        }
else return (FALSE)
```

Use a Lock or a CAS

```
Lock (L)
local := counter
local++
counter := local
Unlock(L)
```

Synchronization:
Only one thread can acquire a lock L

```
START:
old := counter
new := old ++
if ( ! CAS(&counter,old,new) )
        goto START
```

Synchronization:
Only one thread changes from old in a concurrent CAS.

Use a Lock or a CAS

```
Lock (L)
local := counter
local++
counter := local
Unlock(L)
```

```
START:
old := counter
new := old ++
if ( ! CAS(&counter,old,new) )
        goto START
```

Issues:
Efficiency, scalability, Fairness, progress guarantee, design complexity.

What Does "Lock-Free" Mean?

- First try: "never using a lock".
- Well, is this using a lock? (word is initially zero.)

```
while (!CAS(&word,0,1)) {}
local := counter
local++
counter := local
word := 0
```

- It is not easy to say if something is a "lock".

What Does "Lock-Free" Mean?

- Better: "No matter which interleaving is scheduled, my program will make progress.".
- A.k.a. non-blocking.
- Our second example is lock-free.

```
START:
old := counter
new := old ++
if ( ! CAS(&counter,old,new) )
        goto START
```

Lock-Freedom

Lock-Freedom

If you schedule enough steps across all threads, one of them will make progress.

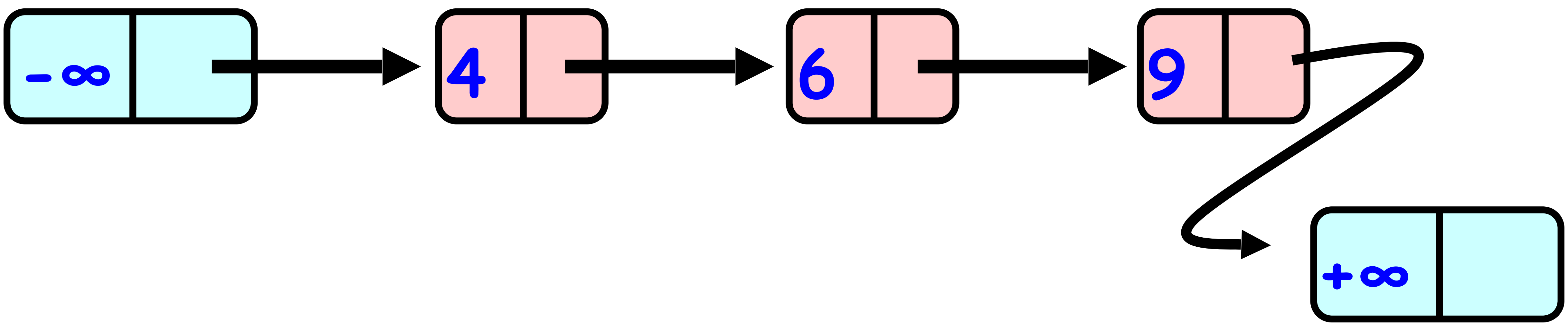
- Realistic (though difficult):
  Various lock-free data structures exist in the literature (stack, queue, hashing, skiplist, trees, etc.).

- Advantages:
  Worst-case responsiveness, scalability, no deadlocks, no livelocks, added robustness to threads fail-stop.

Linked List: Our First Example

Fine-grained locking and lock-freedom

The Linked List

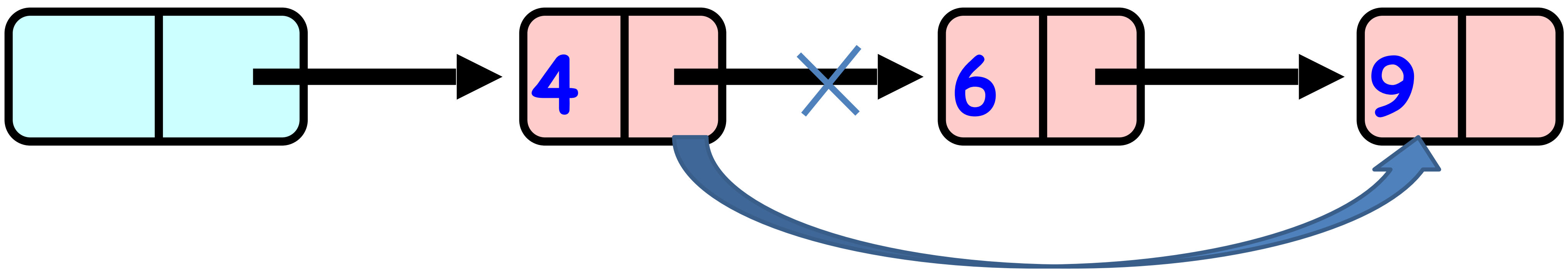


Support: insert, delete, contains.

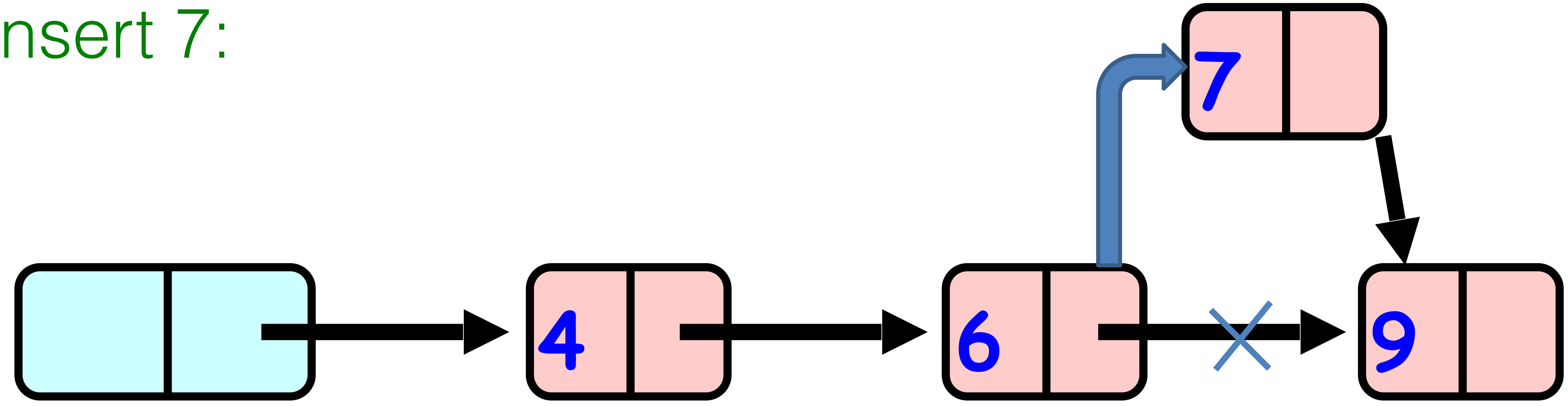
Implements a set: no duplicates, order maintained.

Sequential Implementation

- Delete 6:

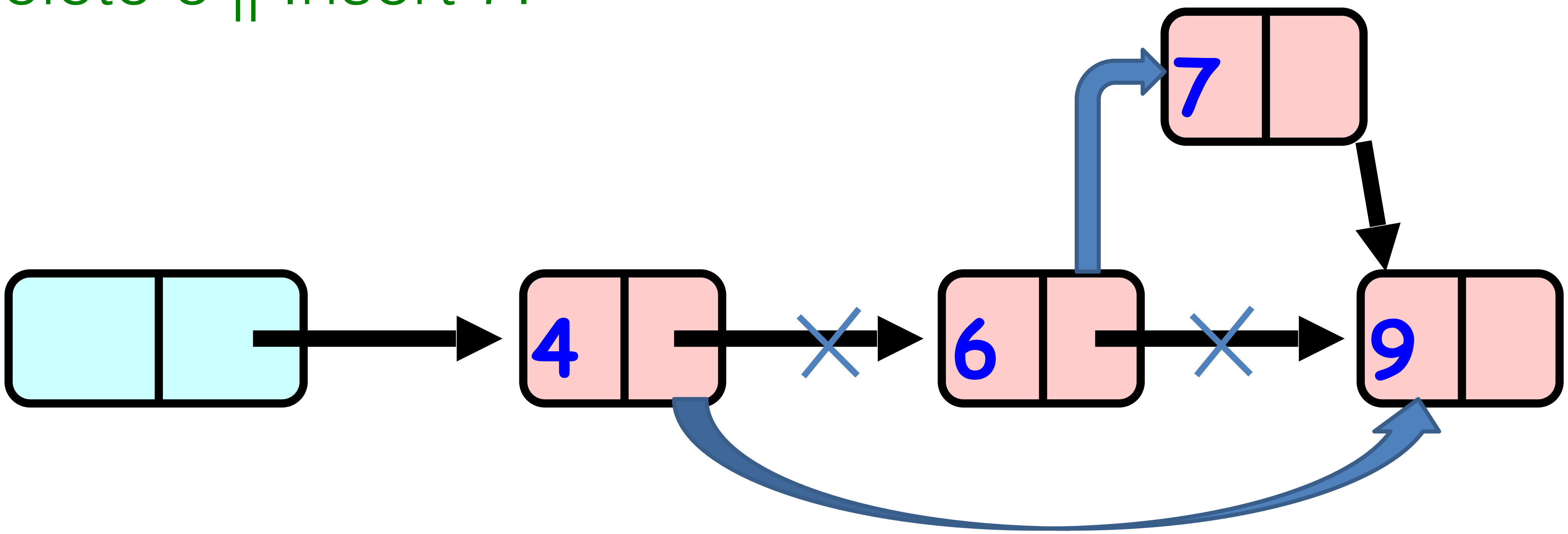


- Insert 7:



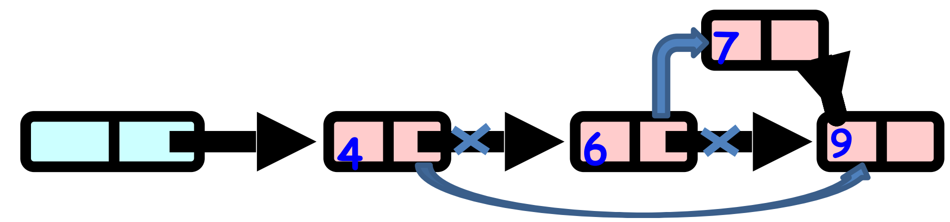
But don’t try this concurrently!

- Delete 6 || Insert 7:

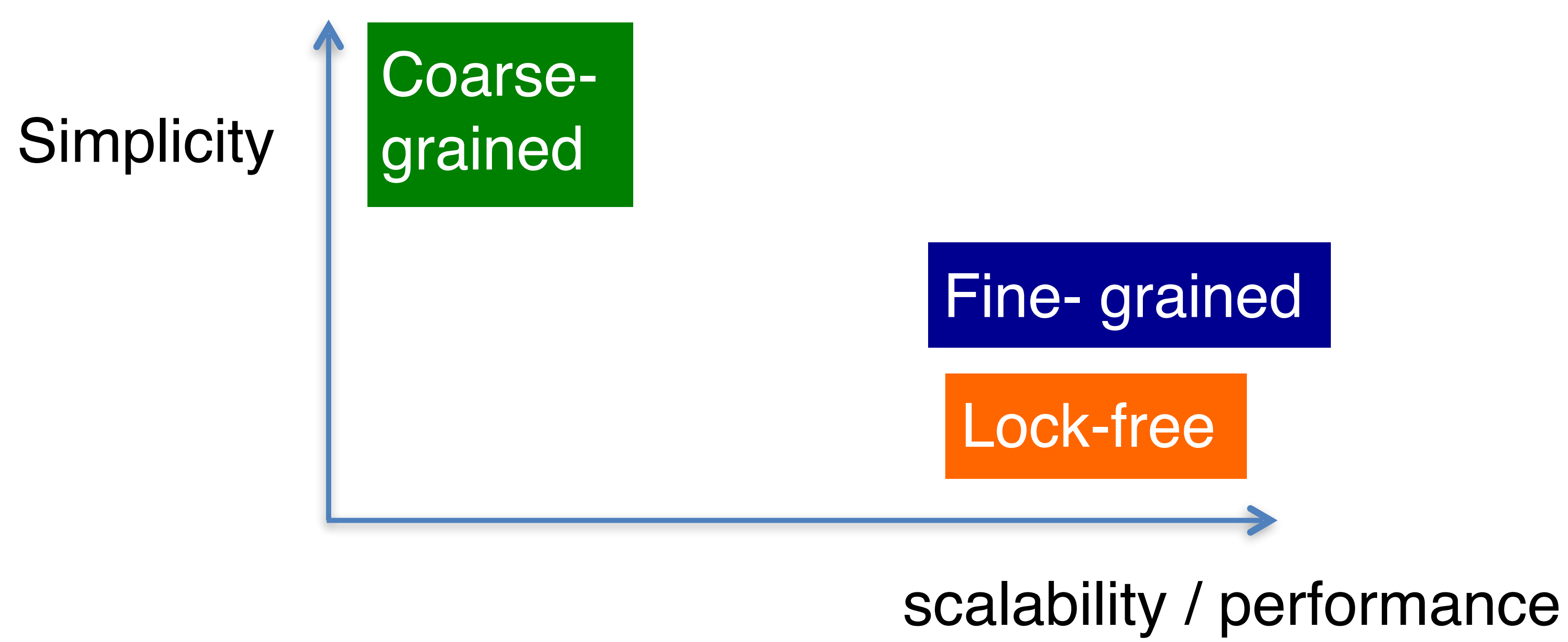


- A similar problem with concurrent deletes.

Solutions



- Coarse-grained locking.
  - Sequential with overhead…
- Fine-grained locking
  - hand-over-hand, optimistic, lazy synchronization.
- Lock-free (or wait-free) implementation.



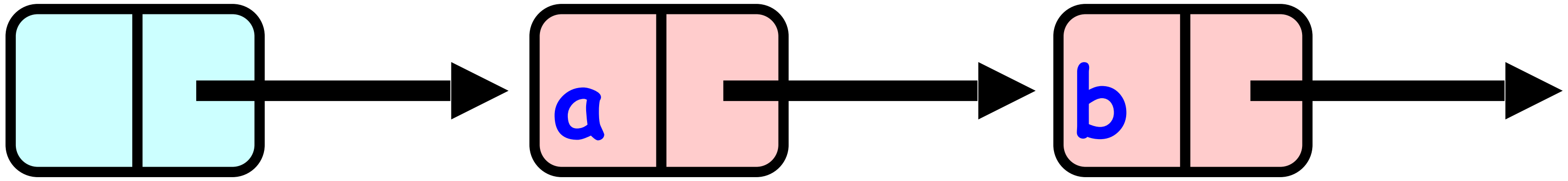
scalability / performance

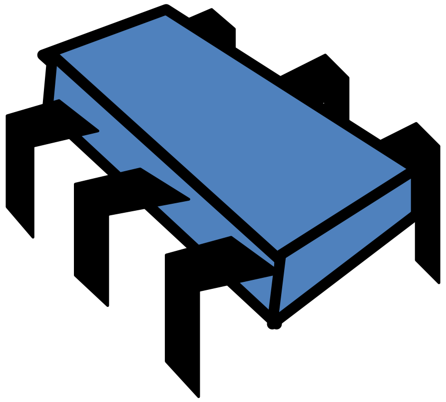
Fine-Grained Locking #1:

Hand-Over-Hand

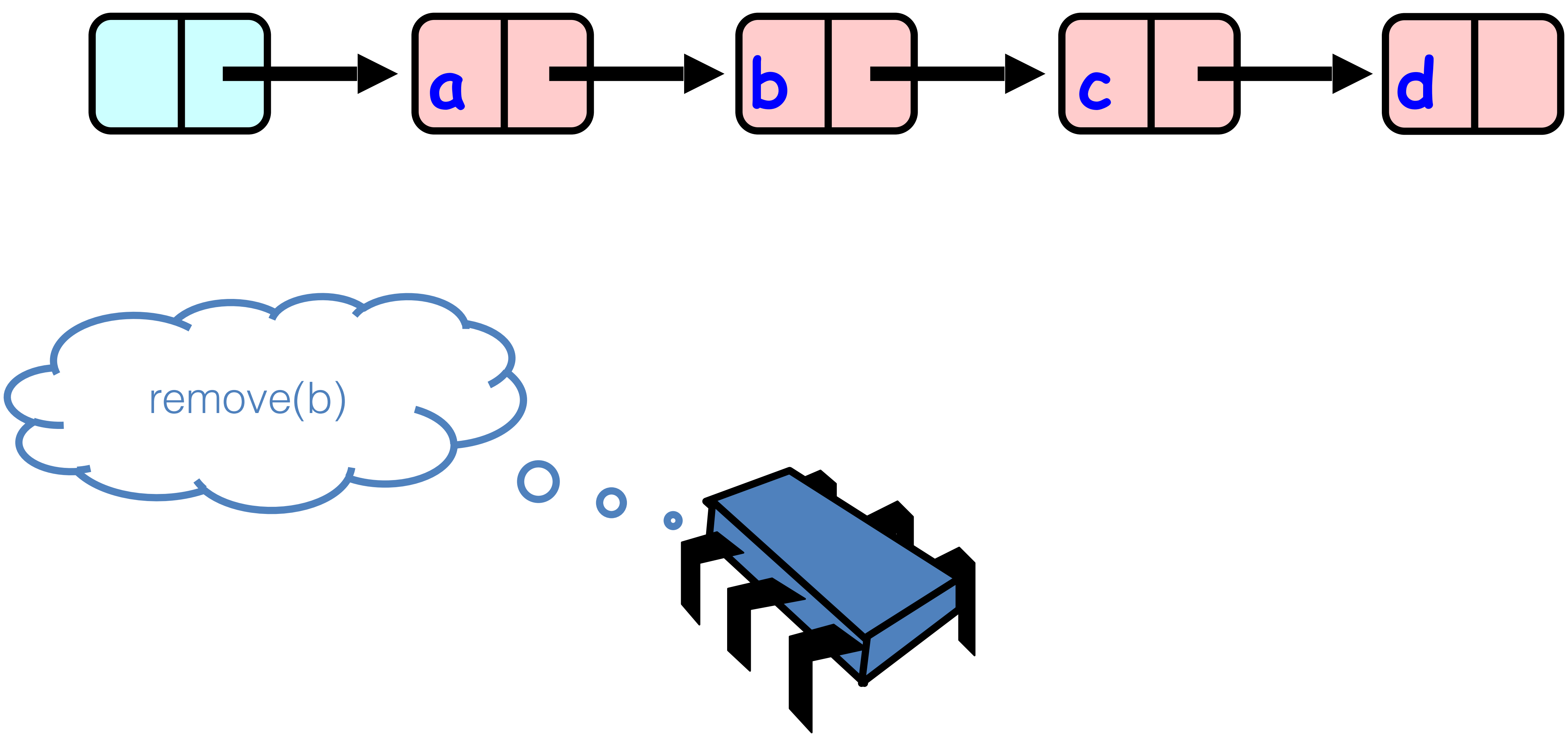
Hand-over-Hand locking

[Bayer-Schkolnik 1977]

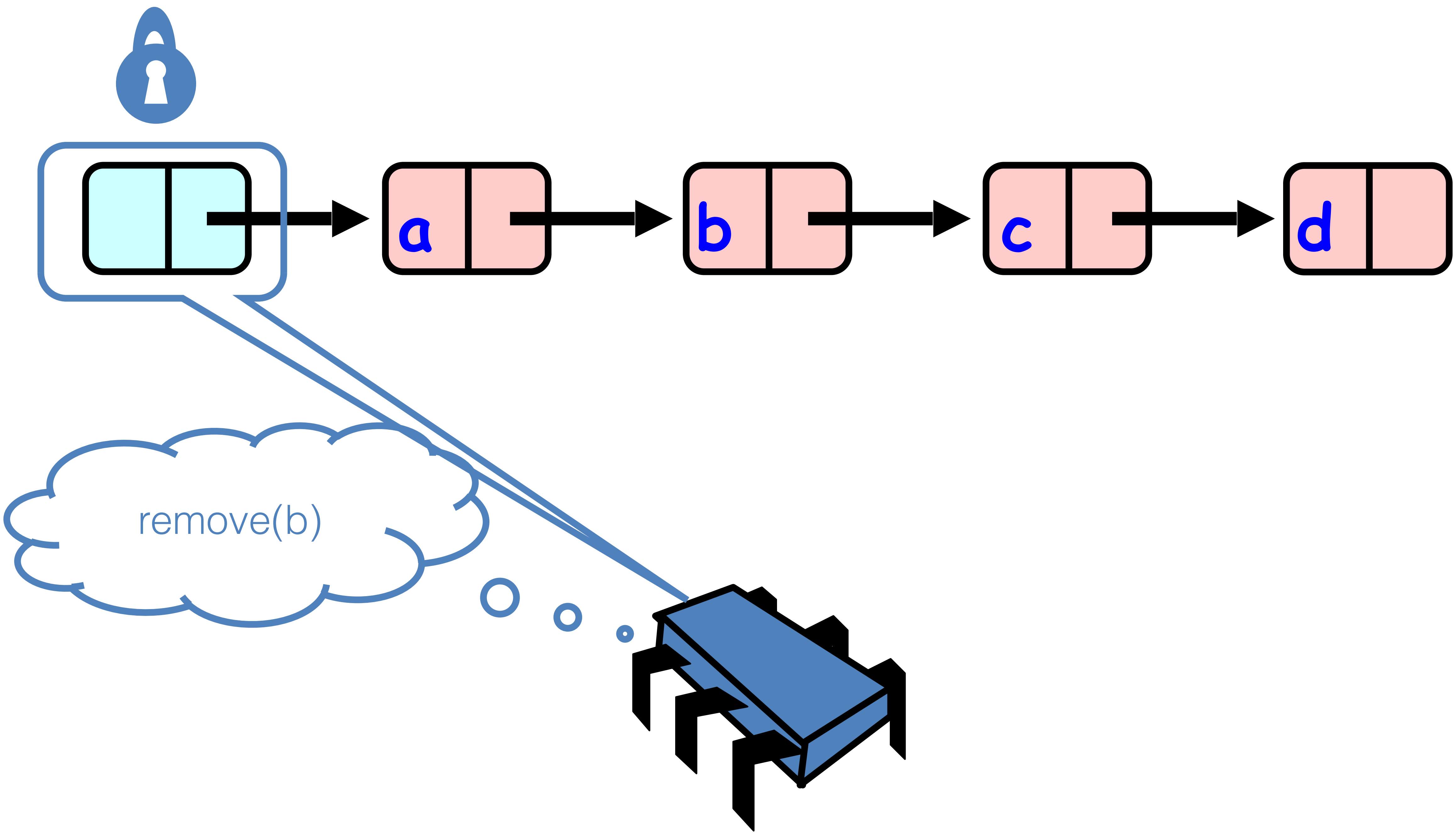




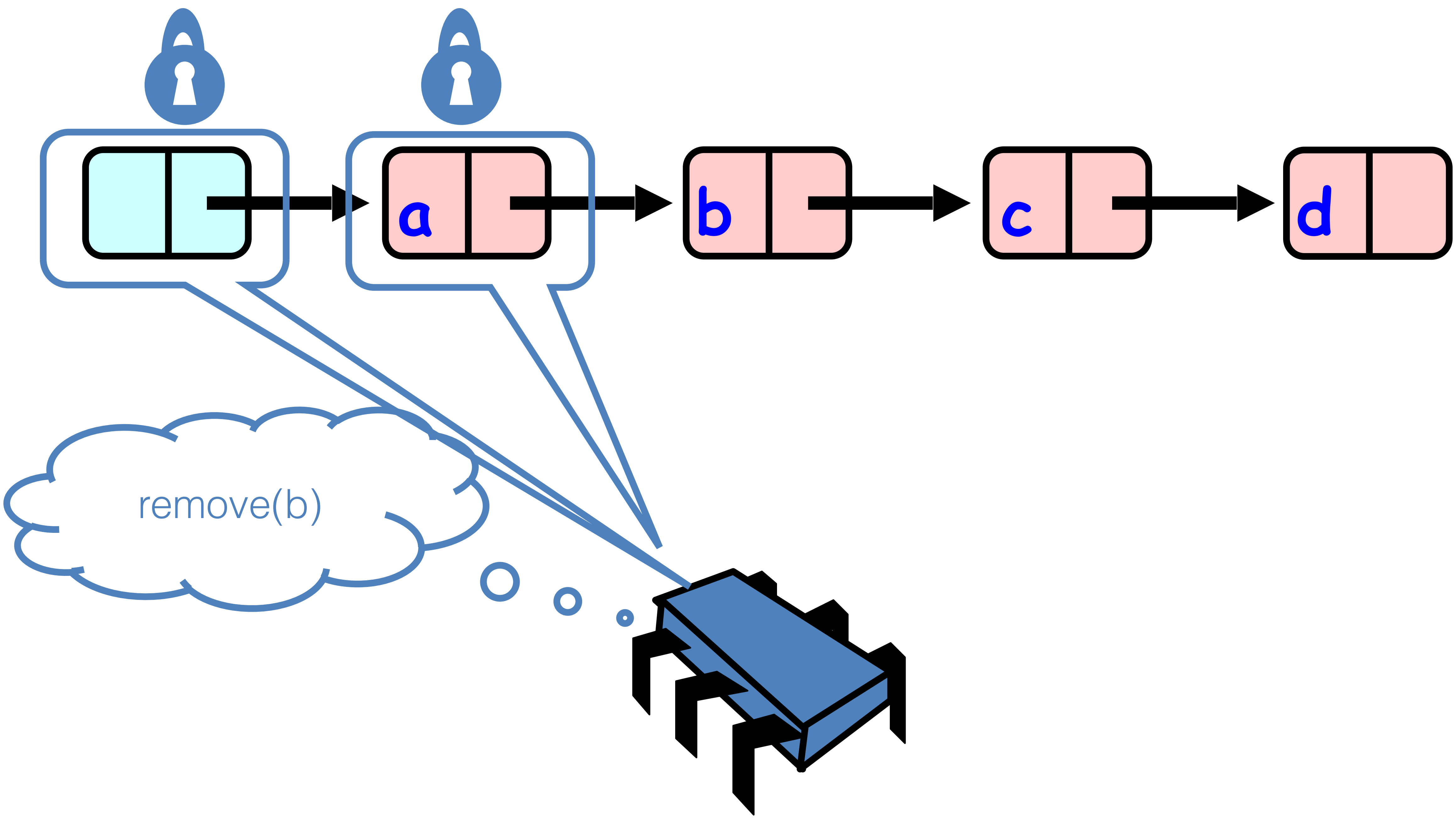
Operation 1: Remove a Node



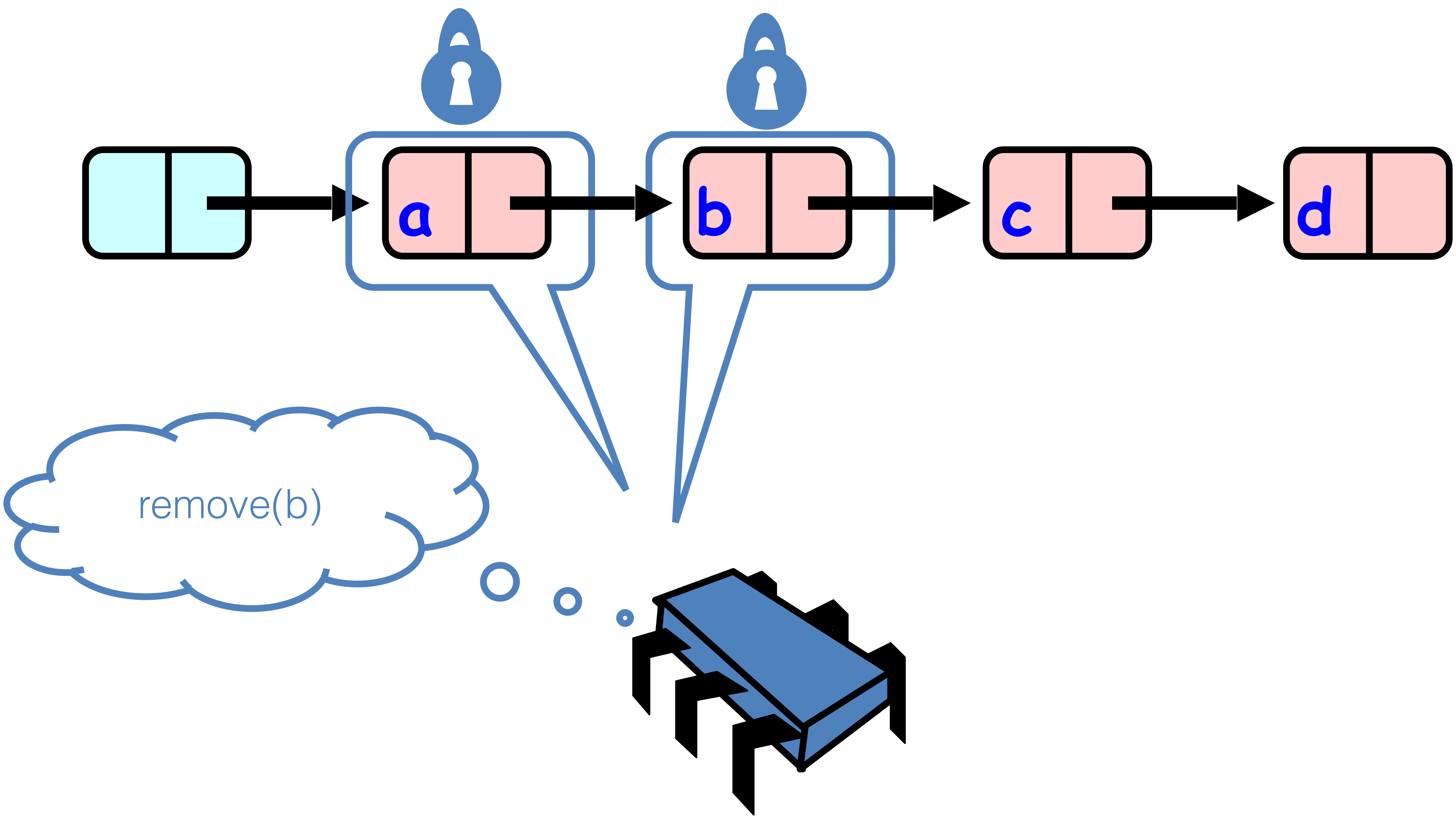
Removing a Node



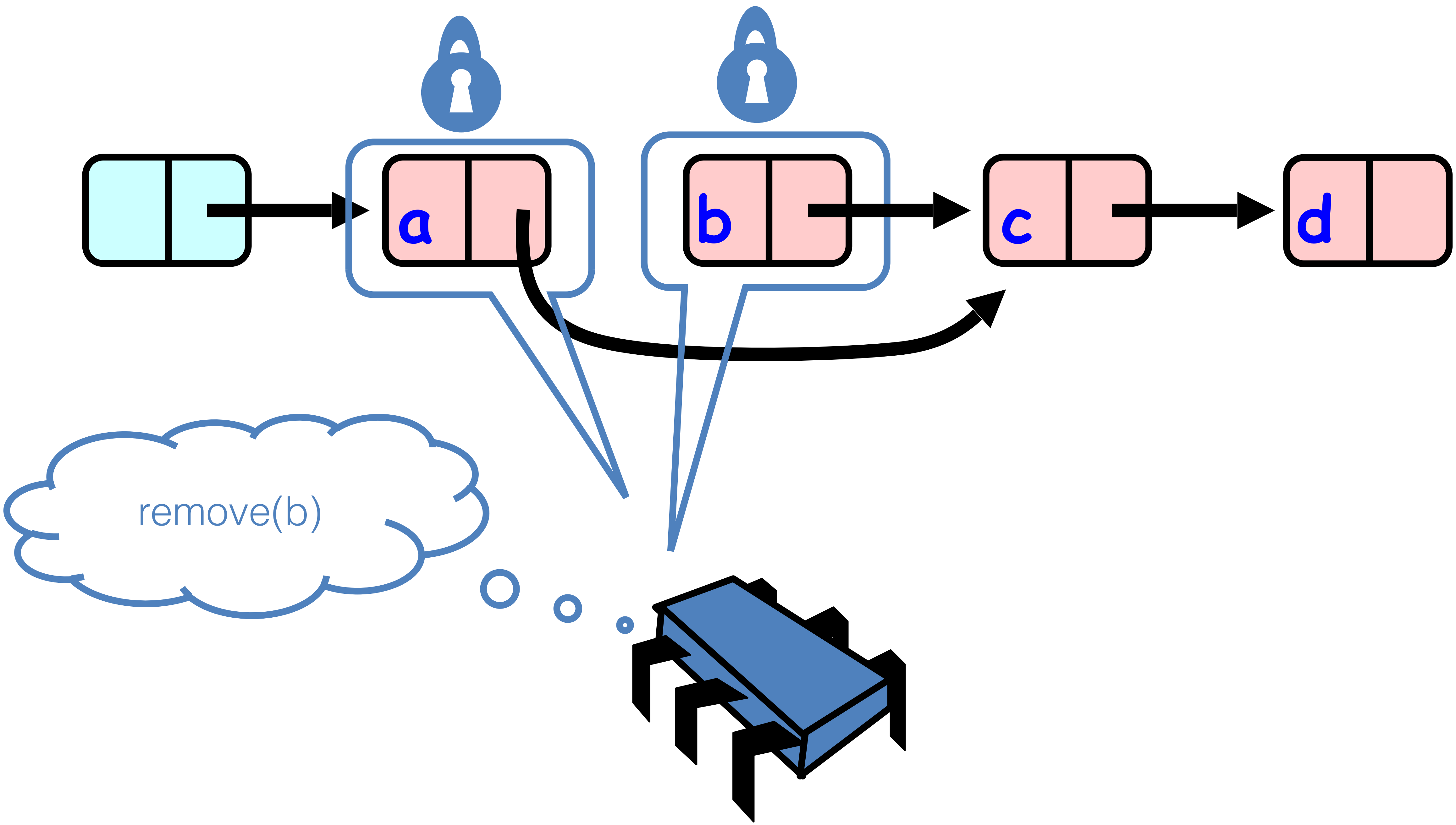
Removing a Node



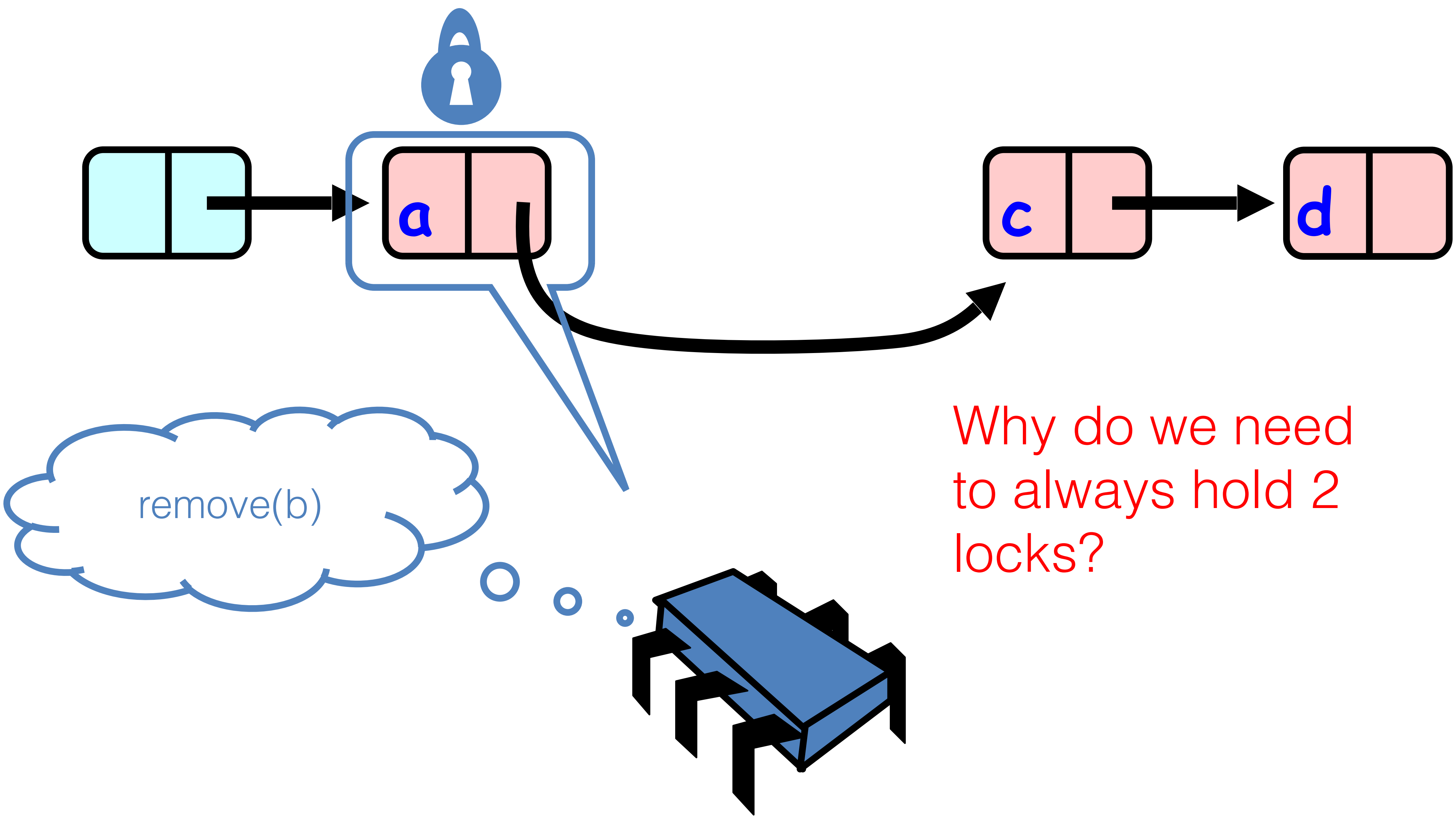
Removing a Node



Removing a Node

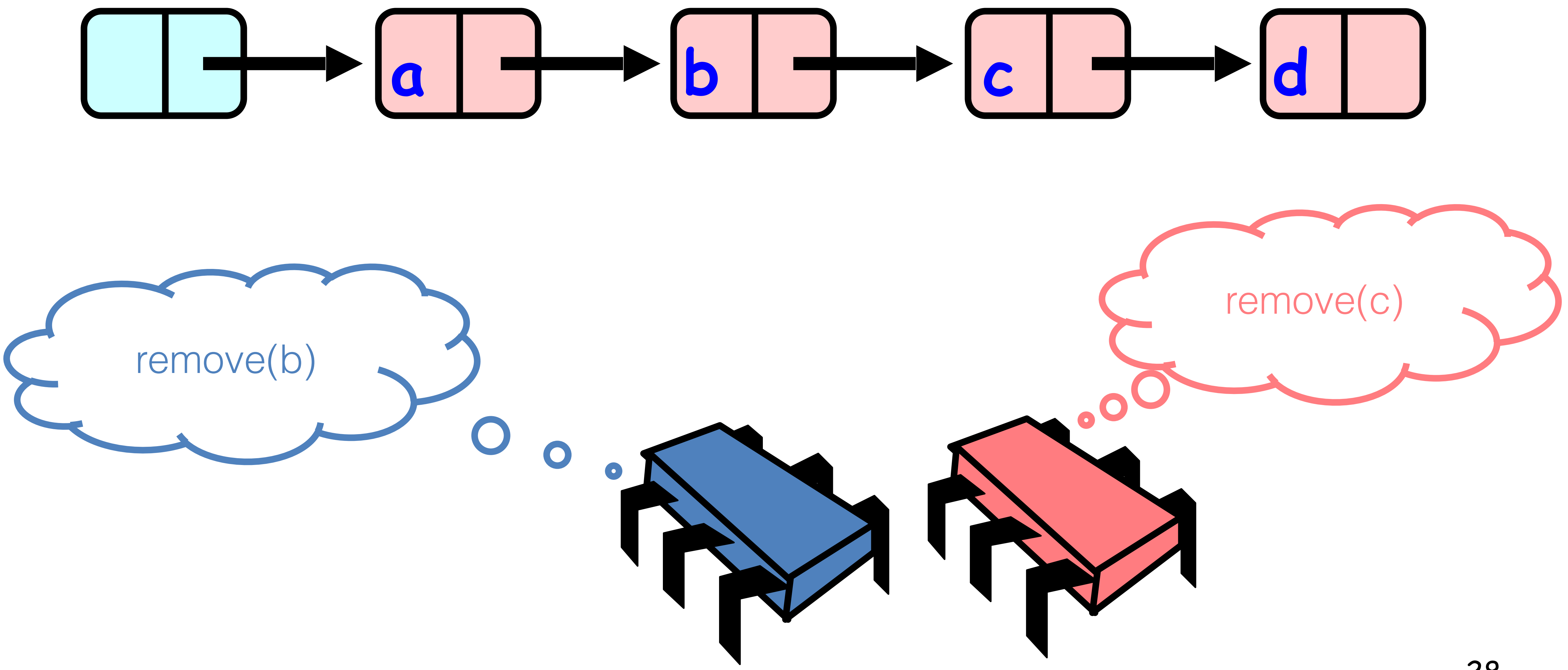


Removing a Node

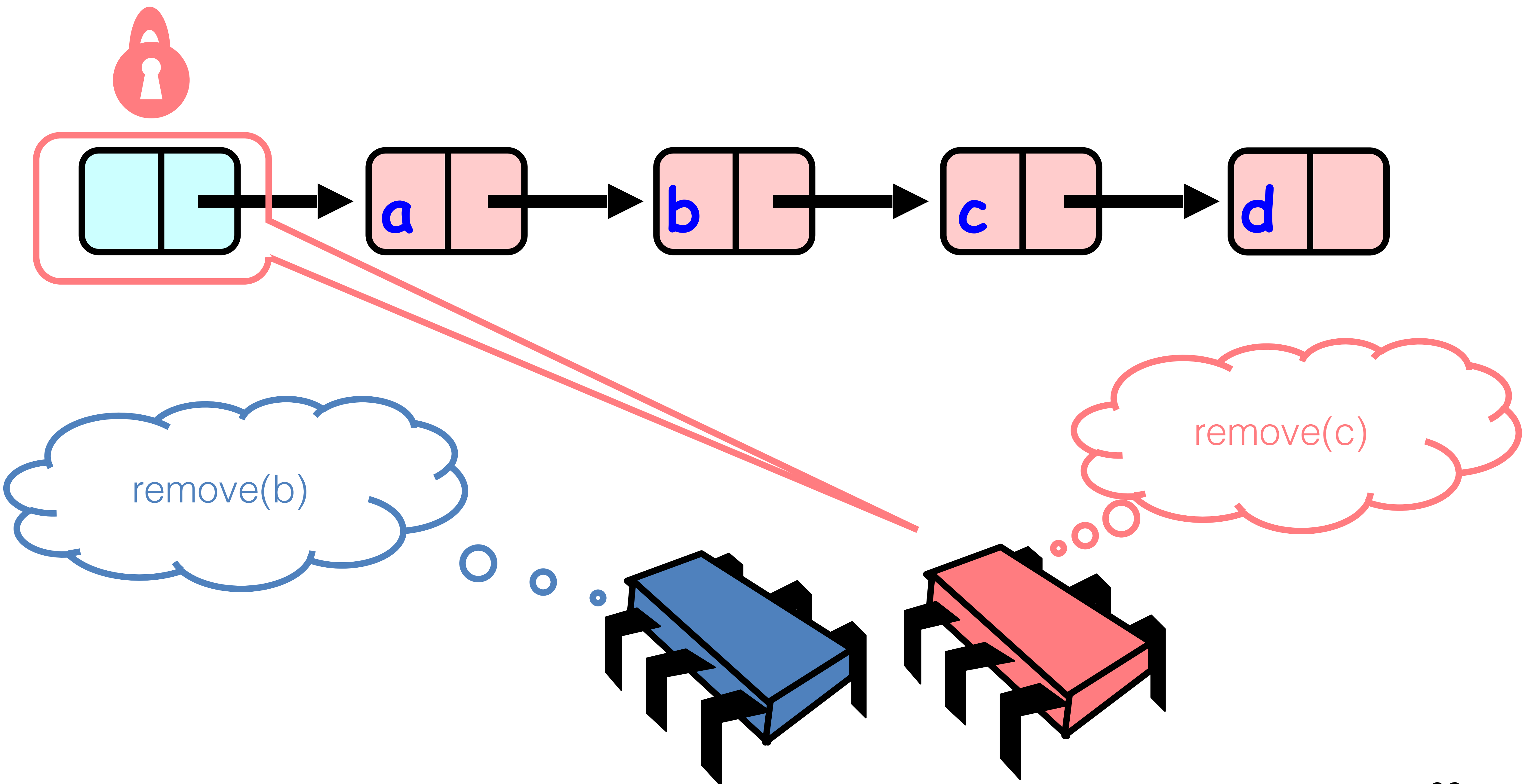


Why do we need to always hold 2 locks?

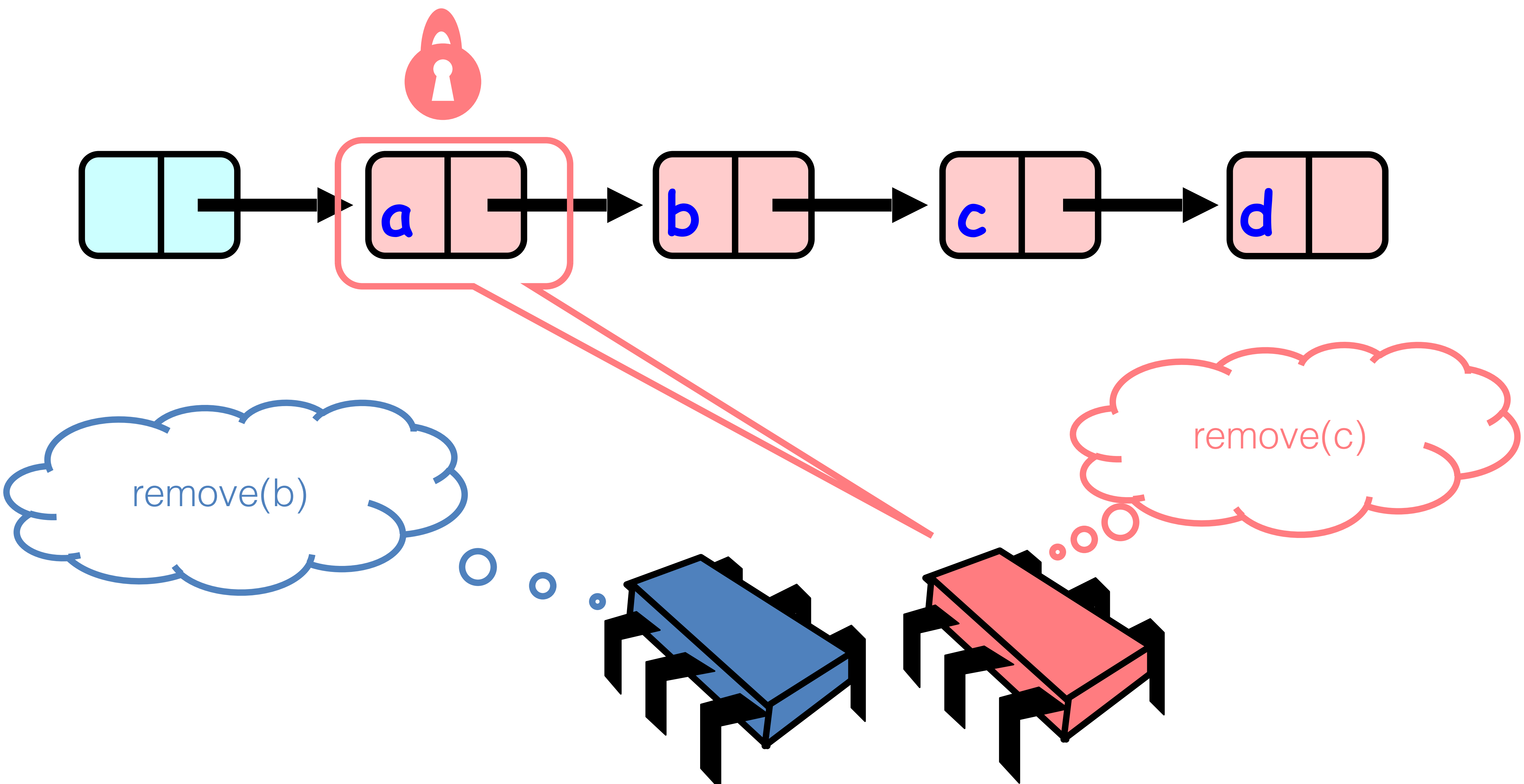
Concurrent Removes



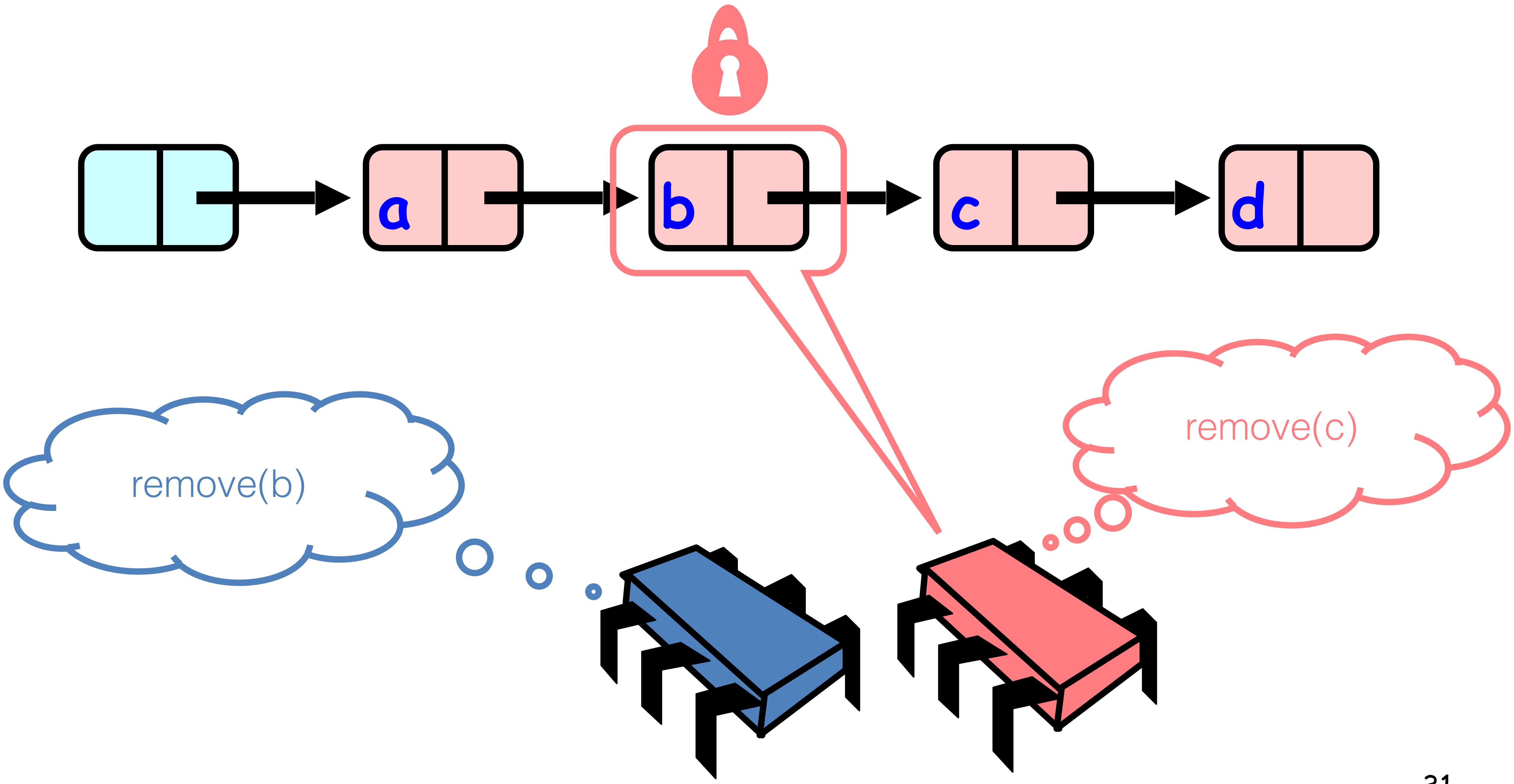
Concurrent Removes



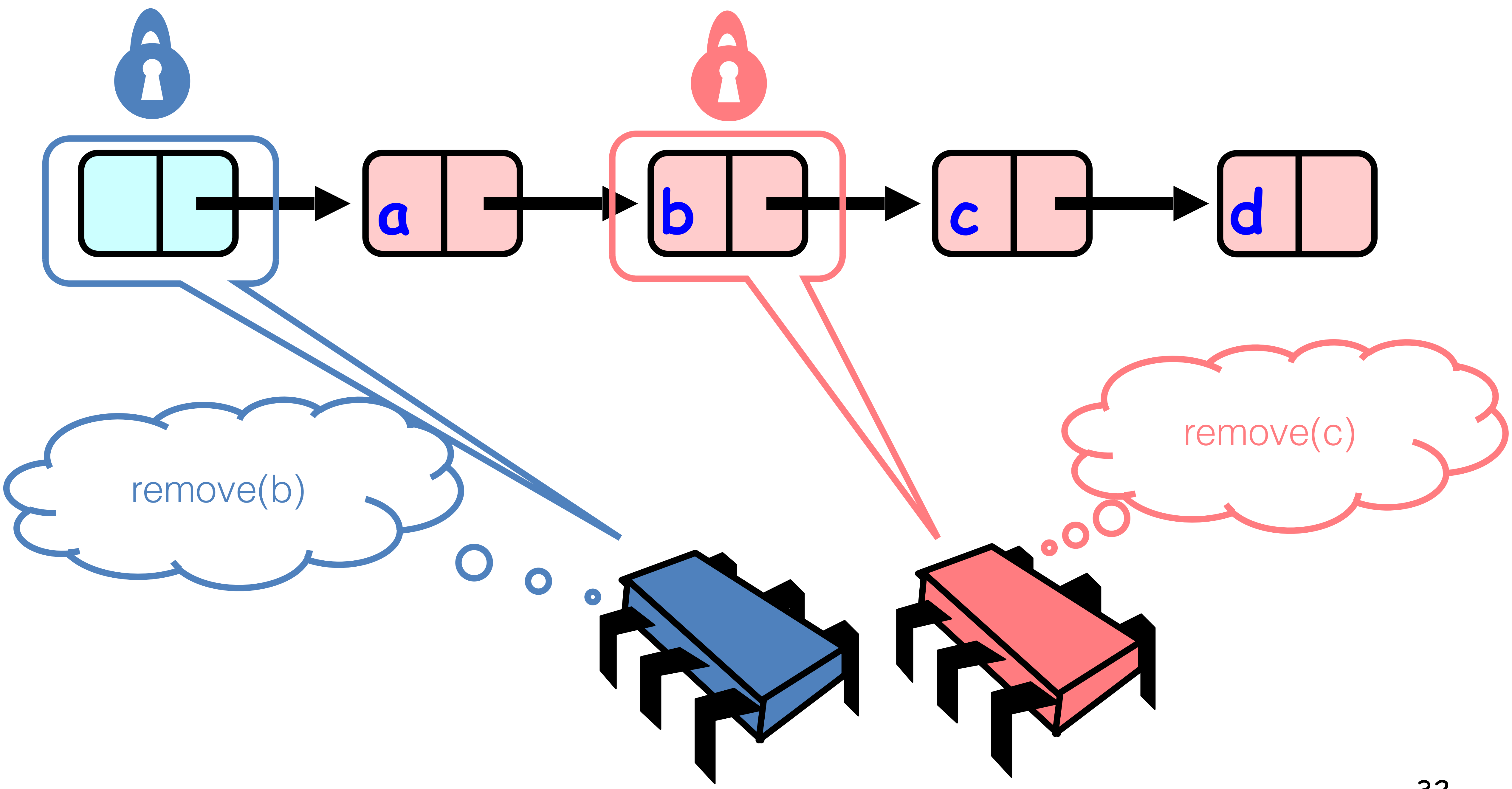
Concurrent Removes



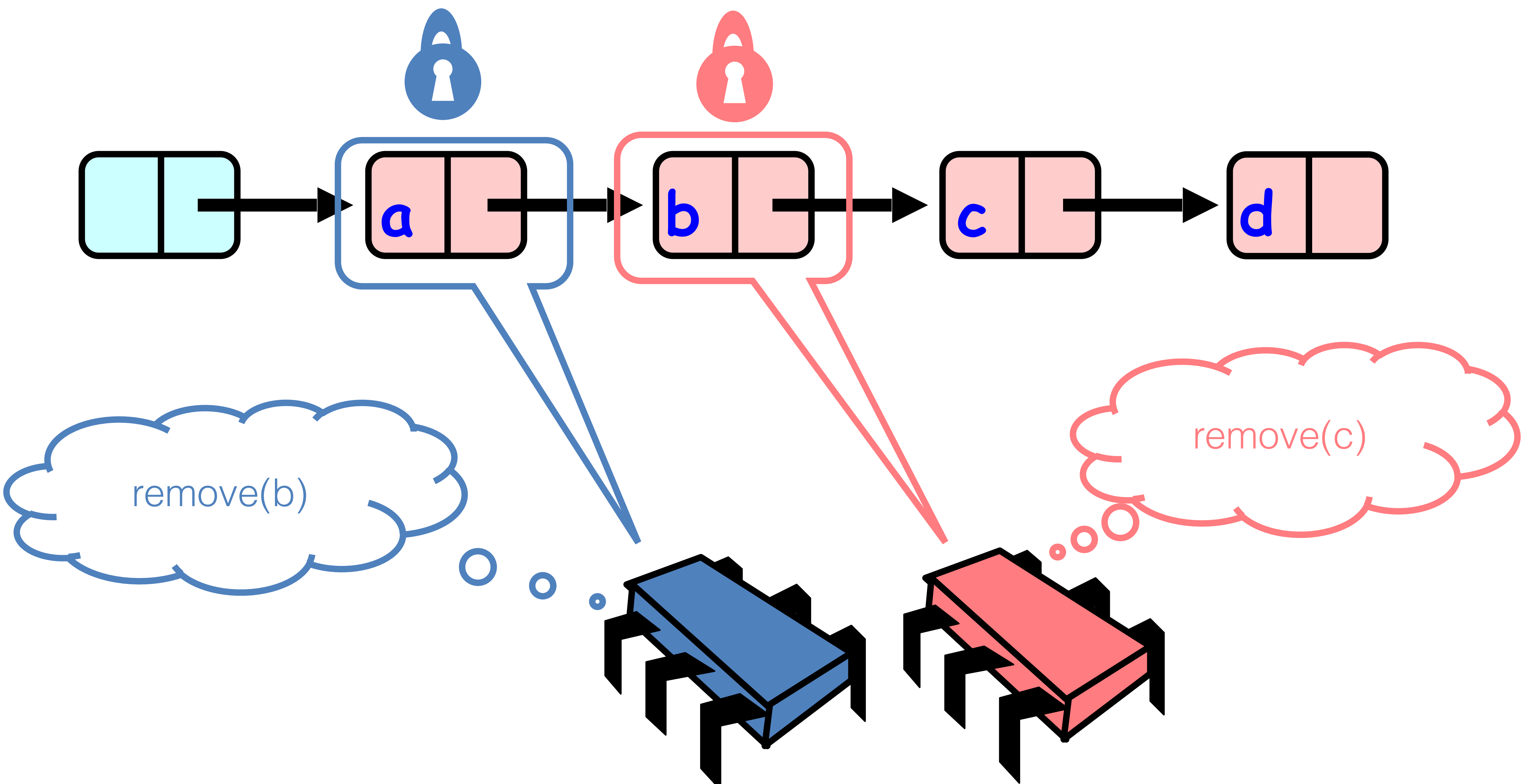
Concurrent Removes



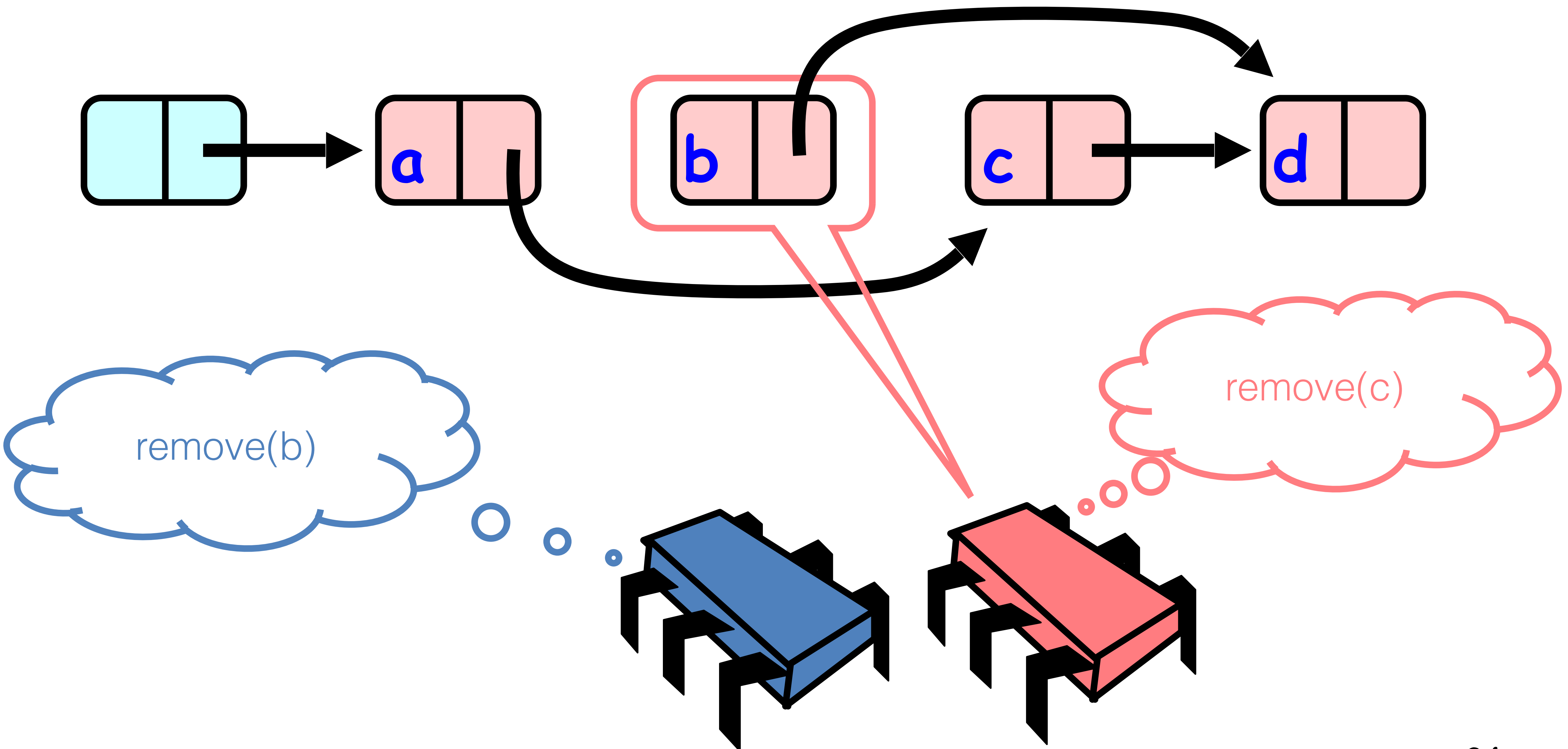
Concurrent Removes



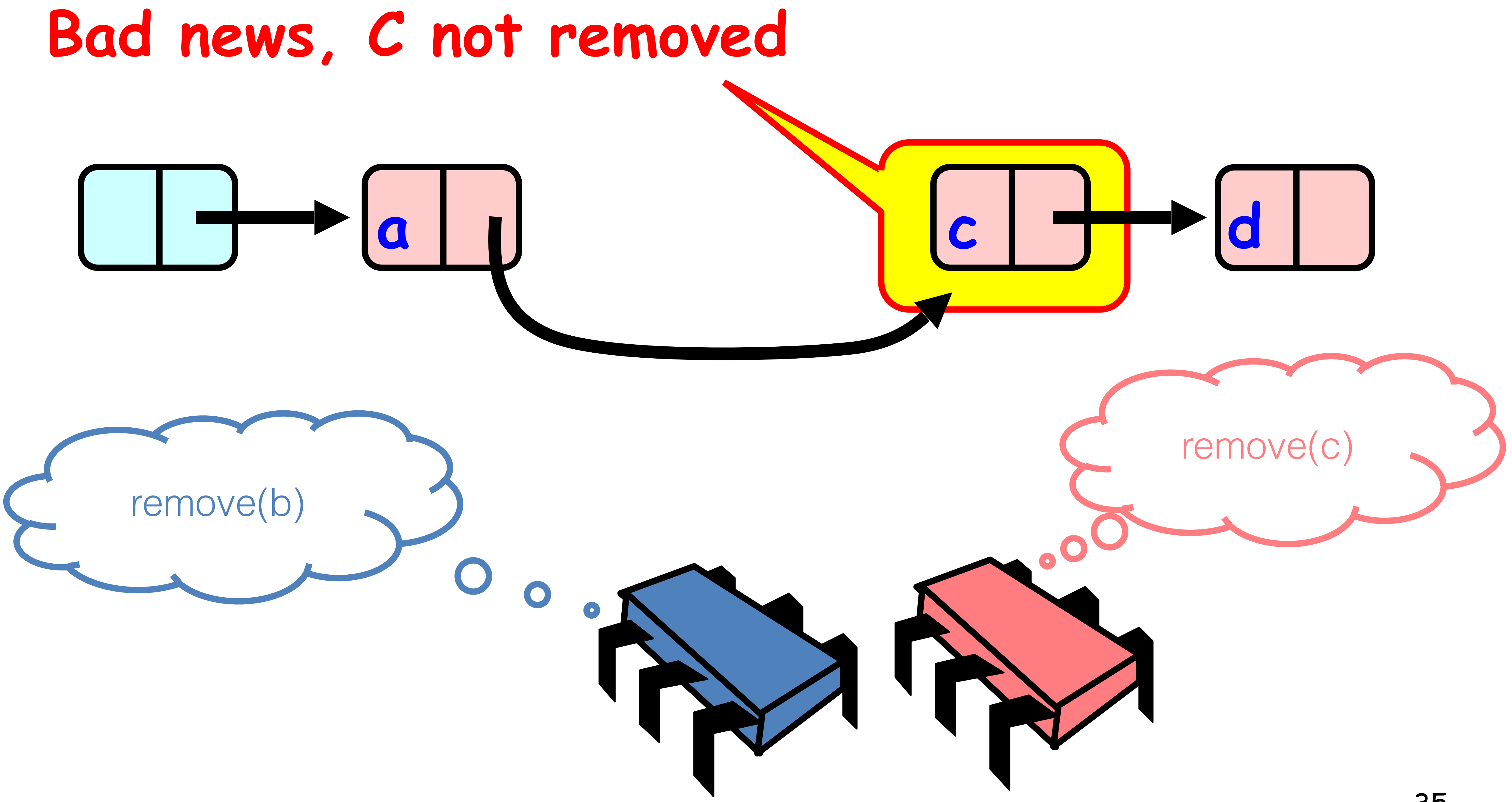
Concurrent Removes



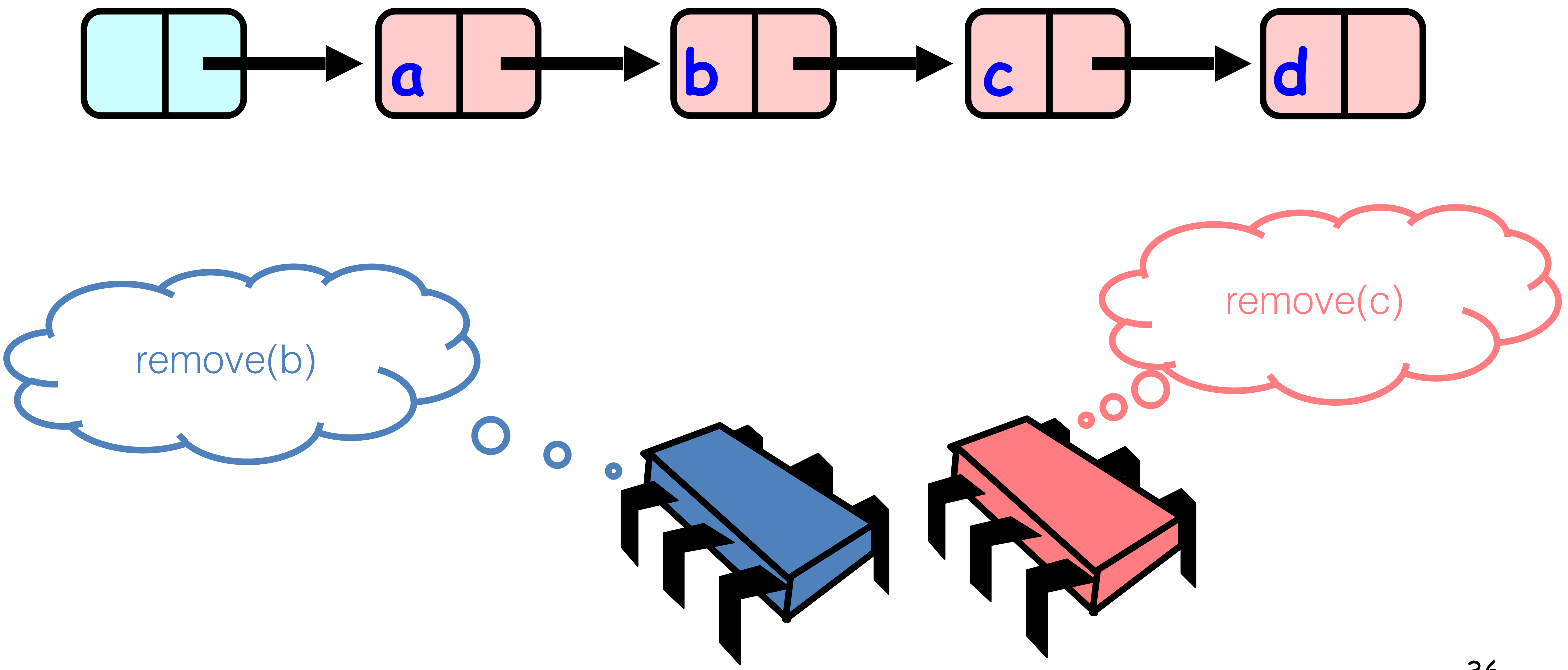
Uh, Oh



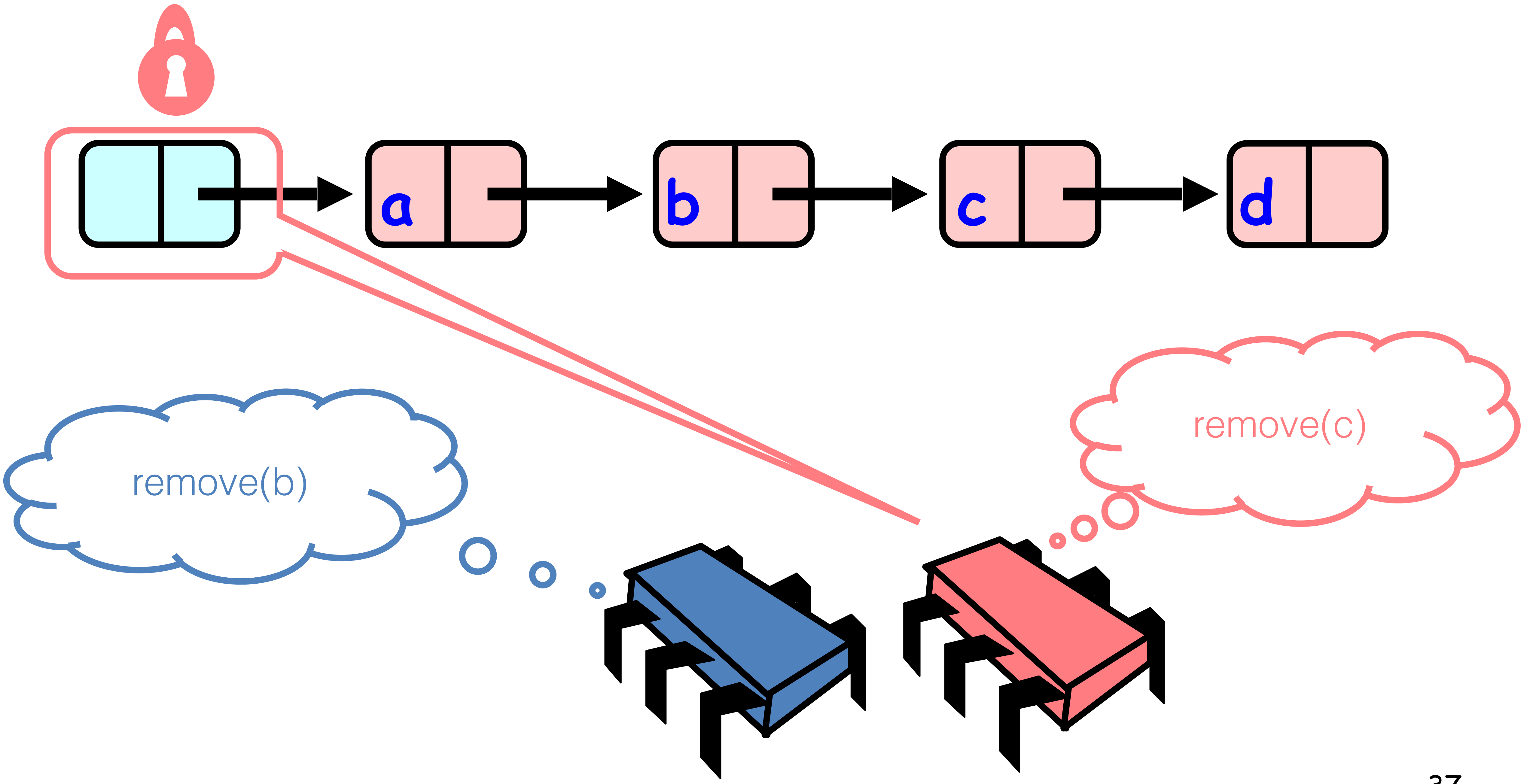
Uh, Oh



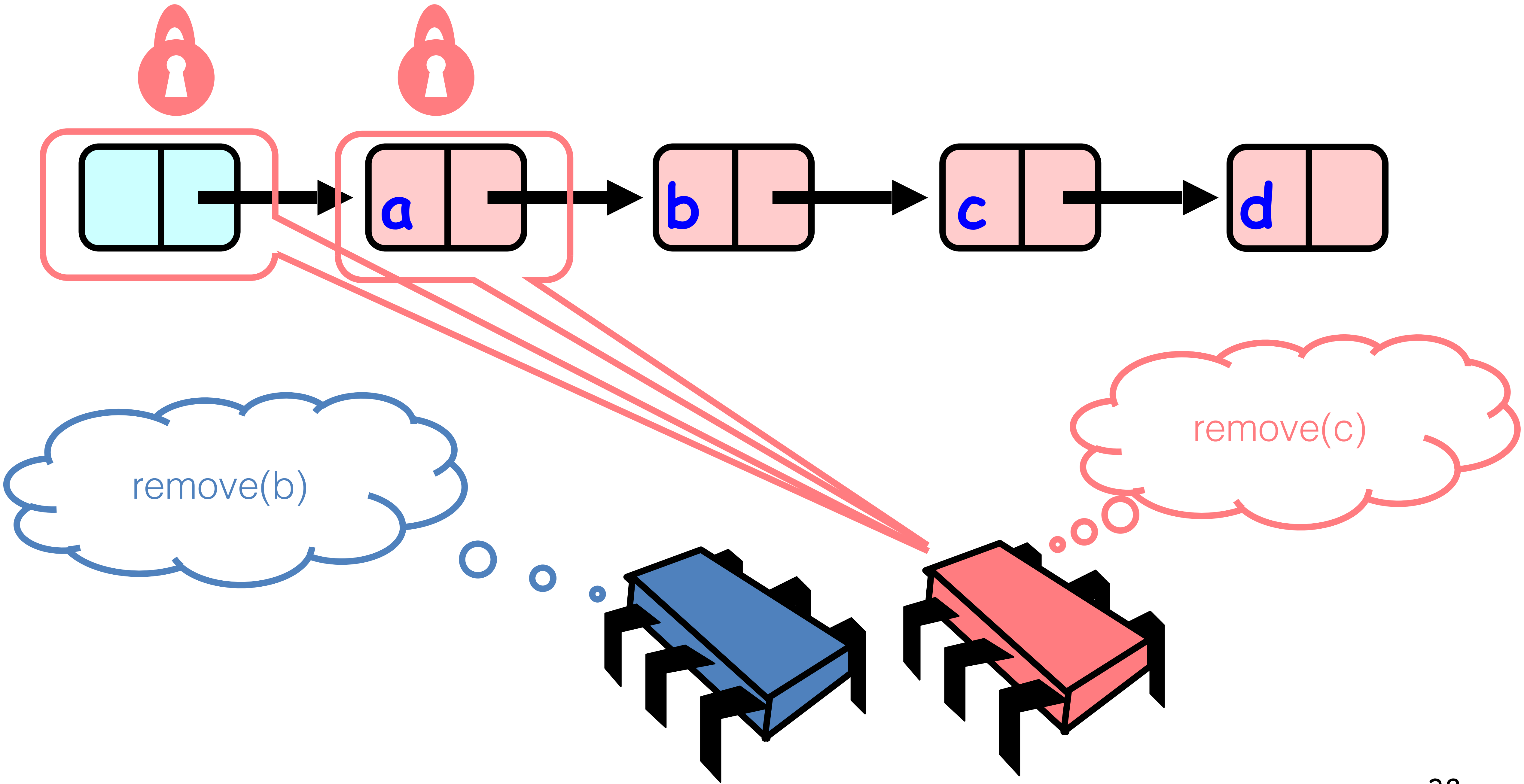
With Two Locks



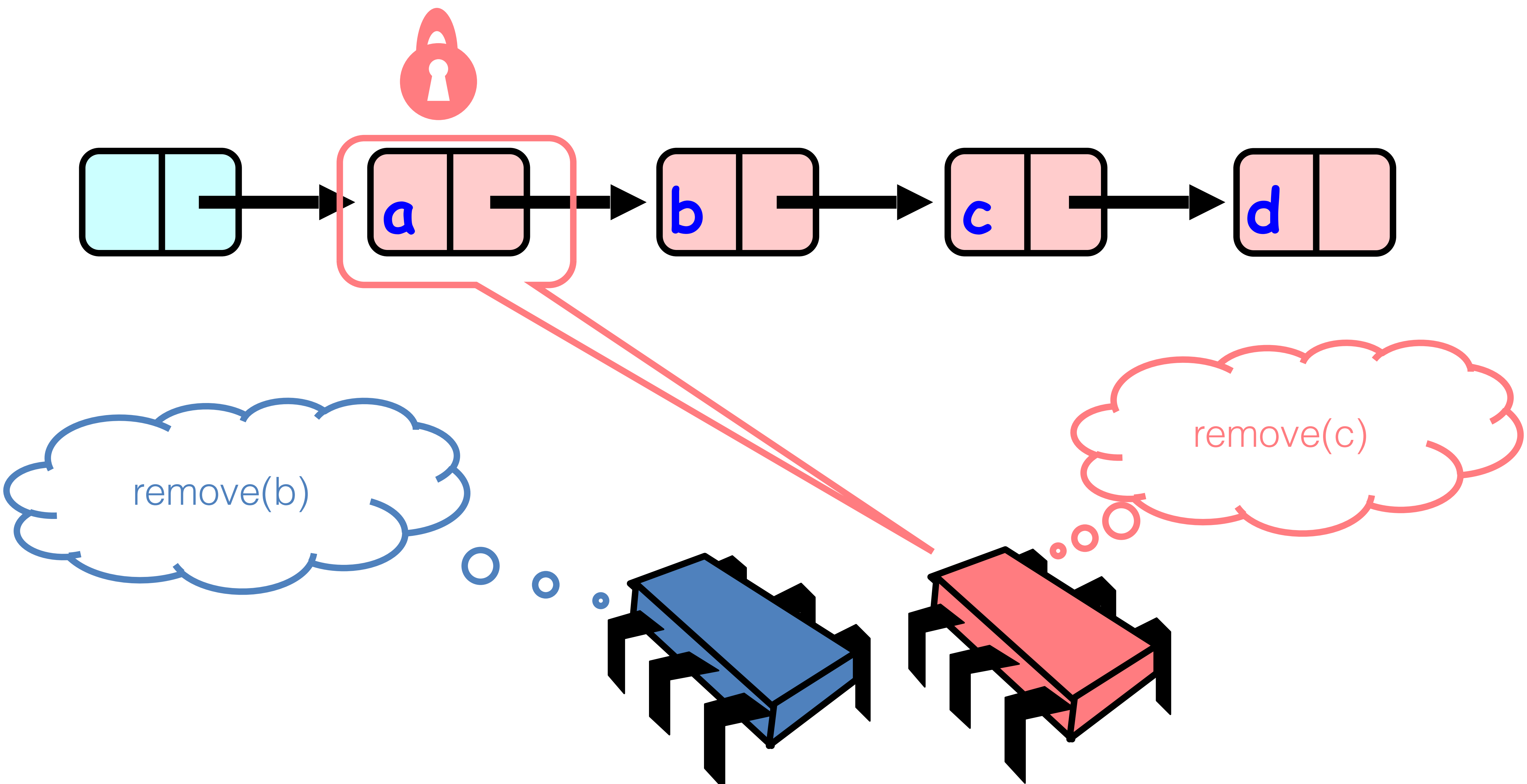
Removing a Node



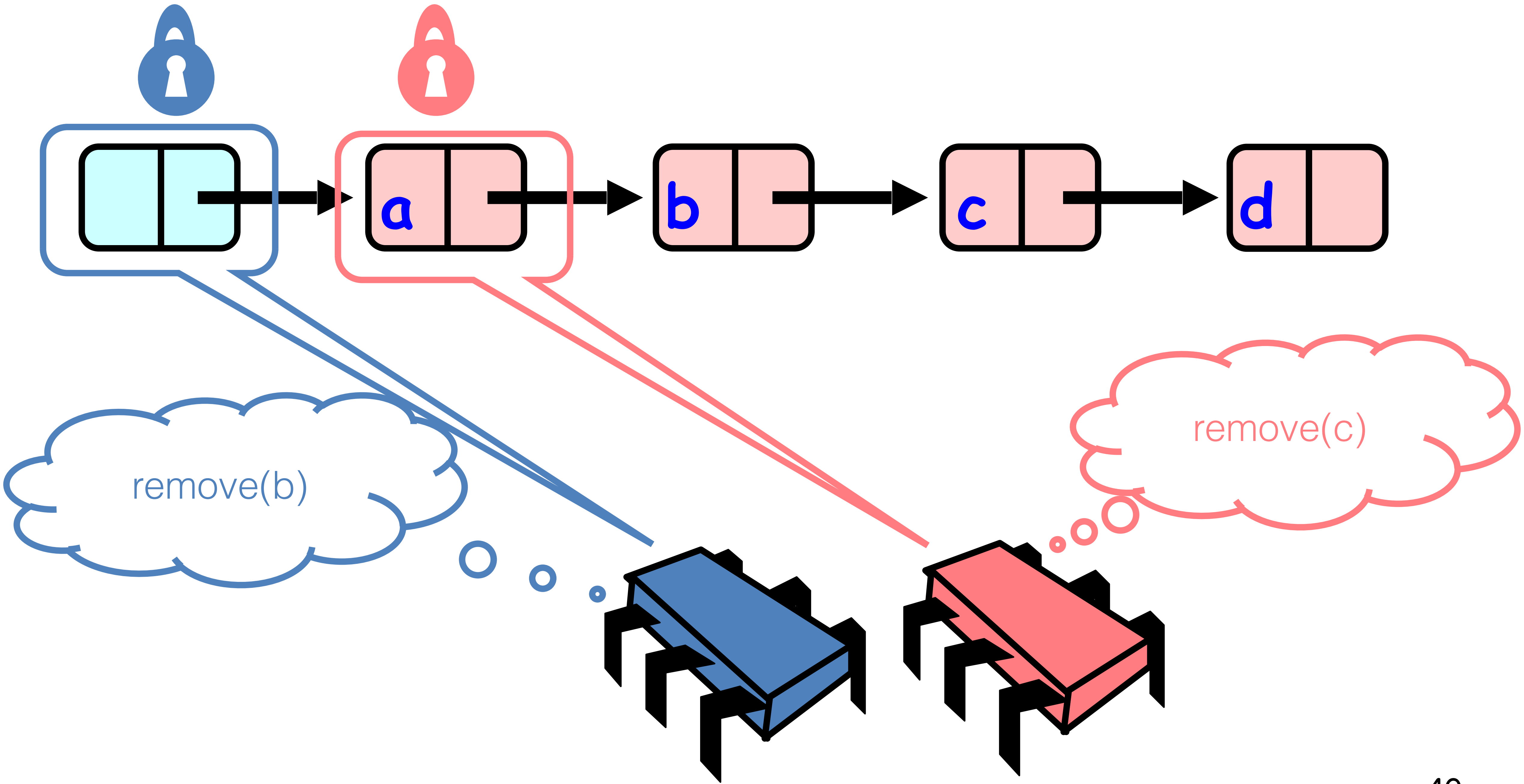
Removing a Node



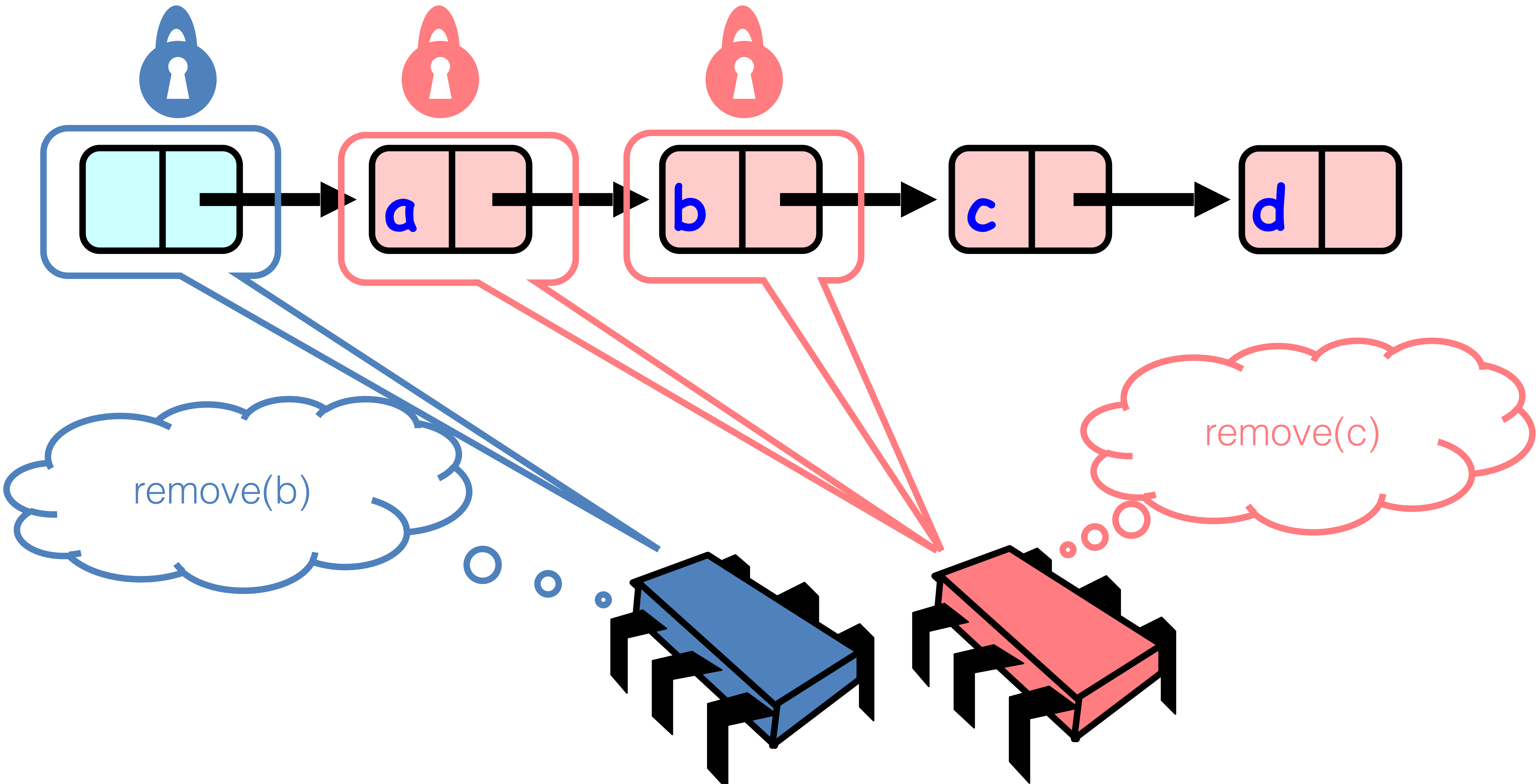
Removing a Node



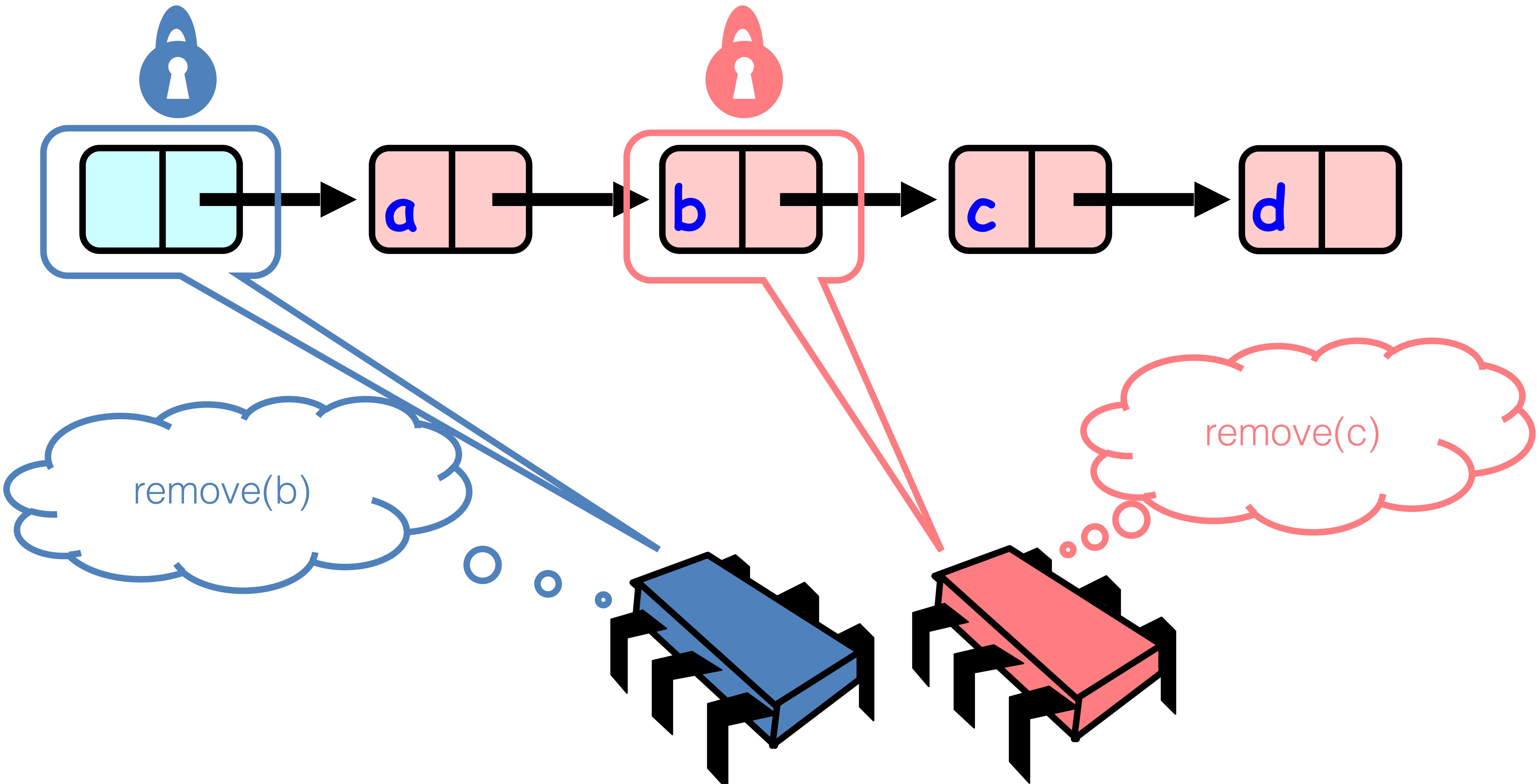
Removing a Node



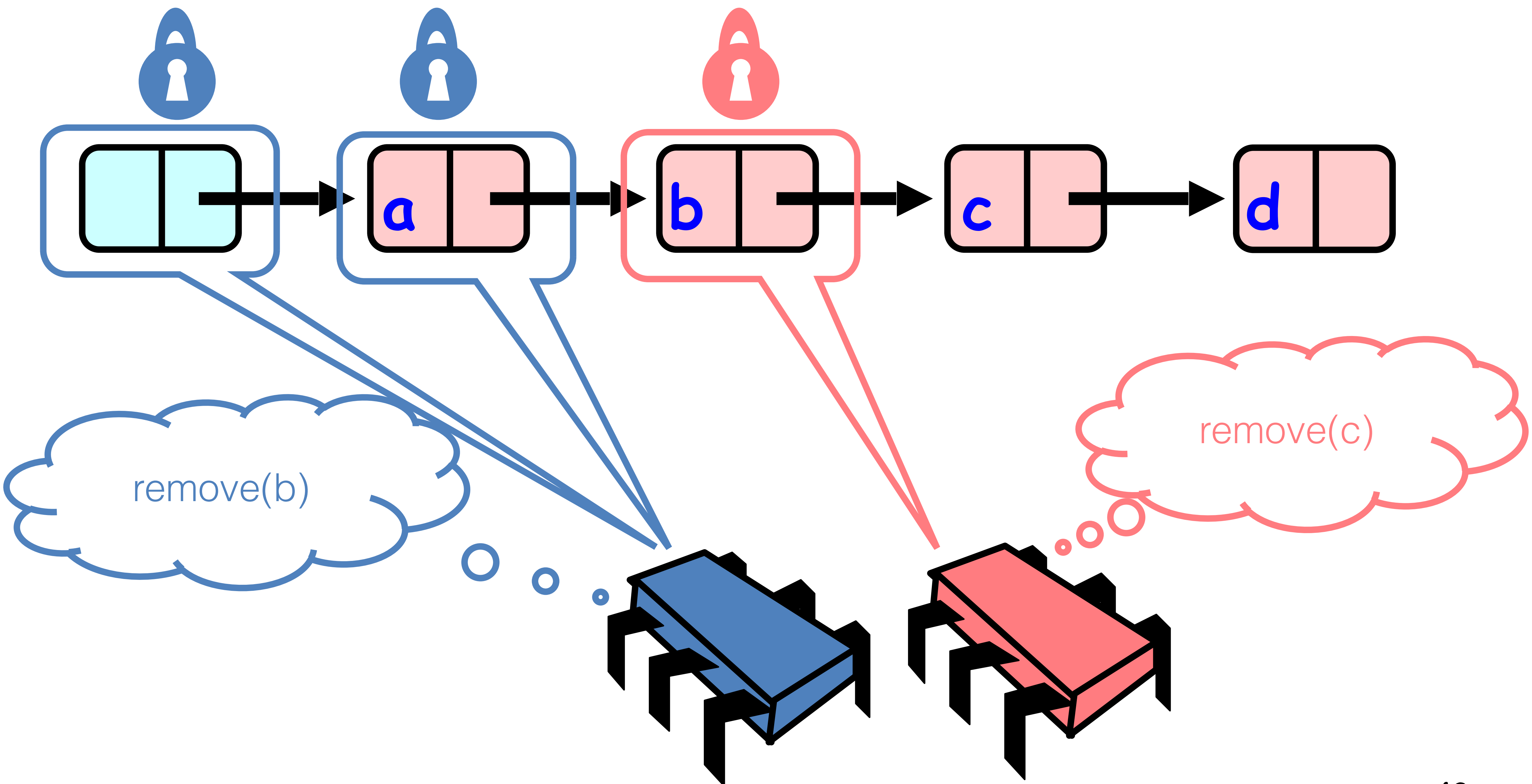
Removing a Node



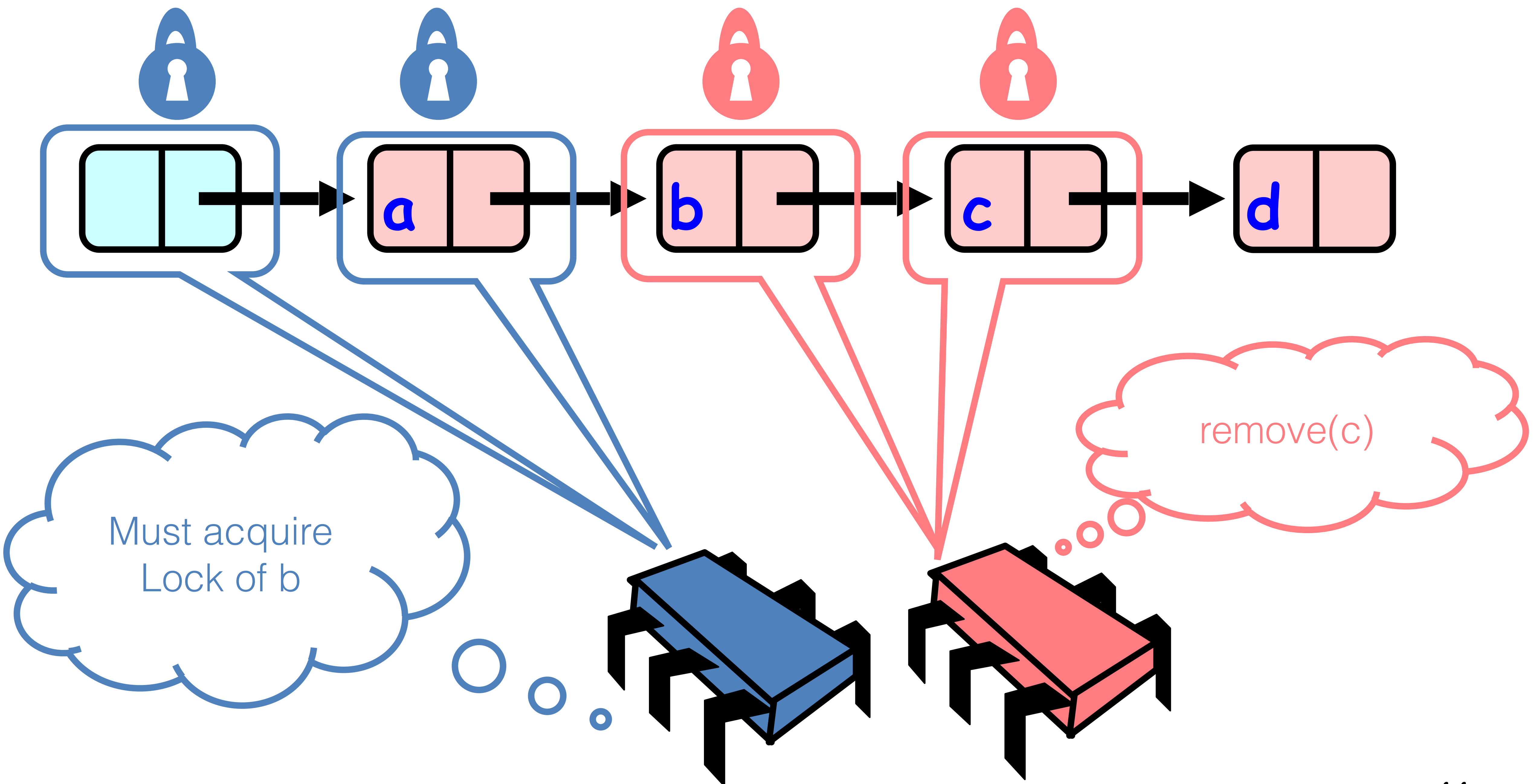
Removing a Node



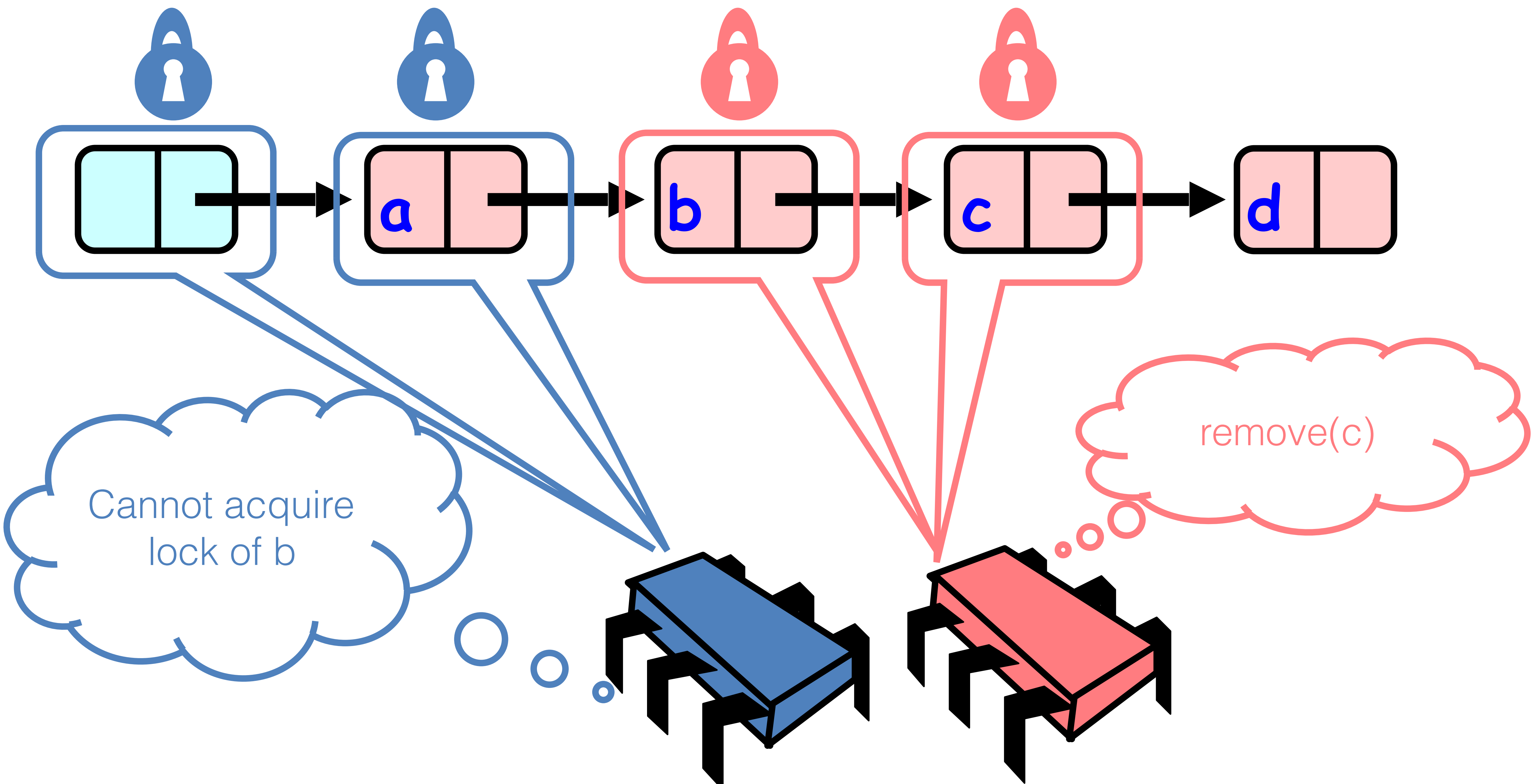
Removing a Node



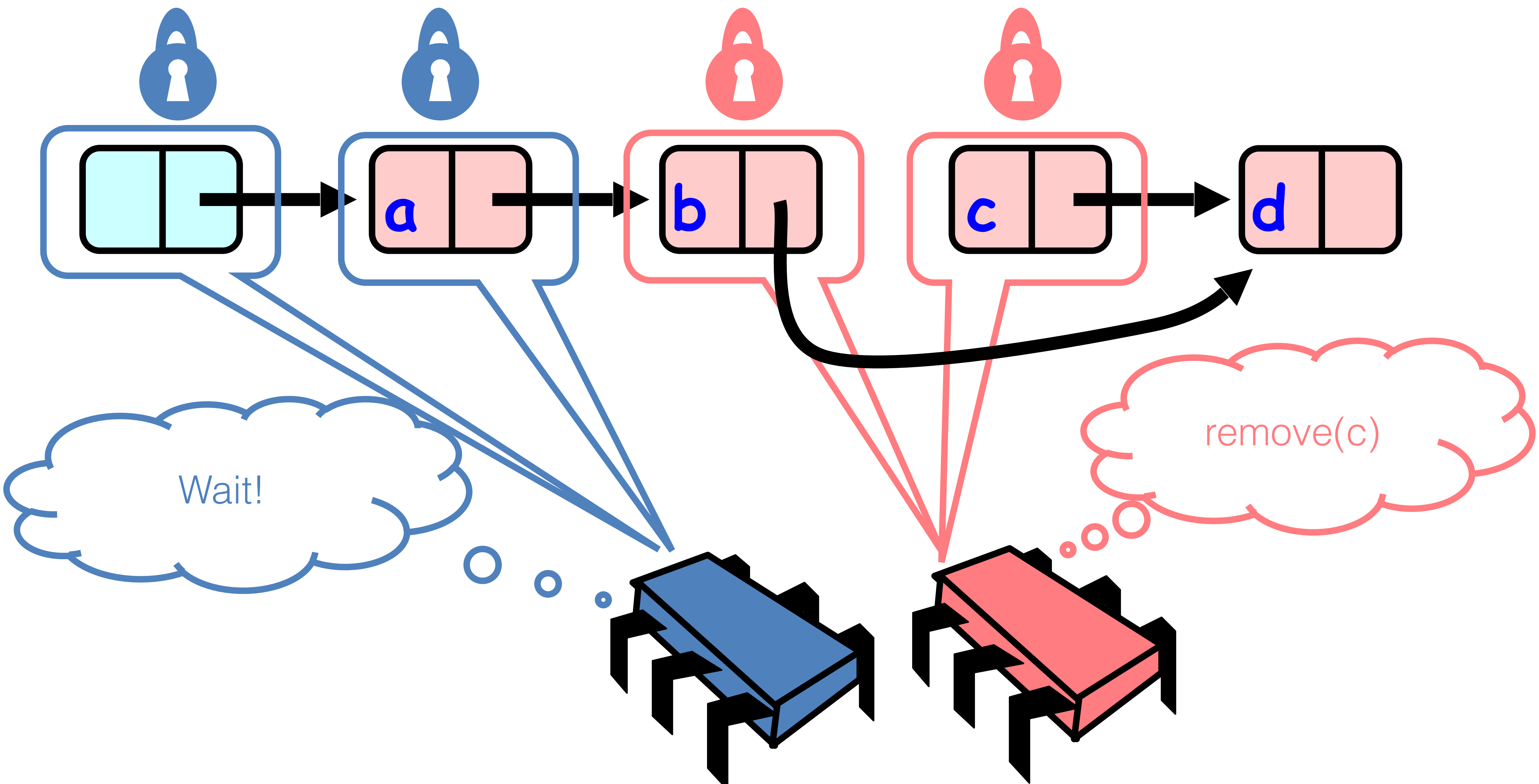
Removing a Node



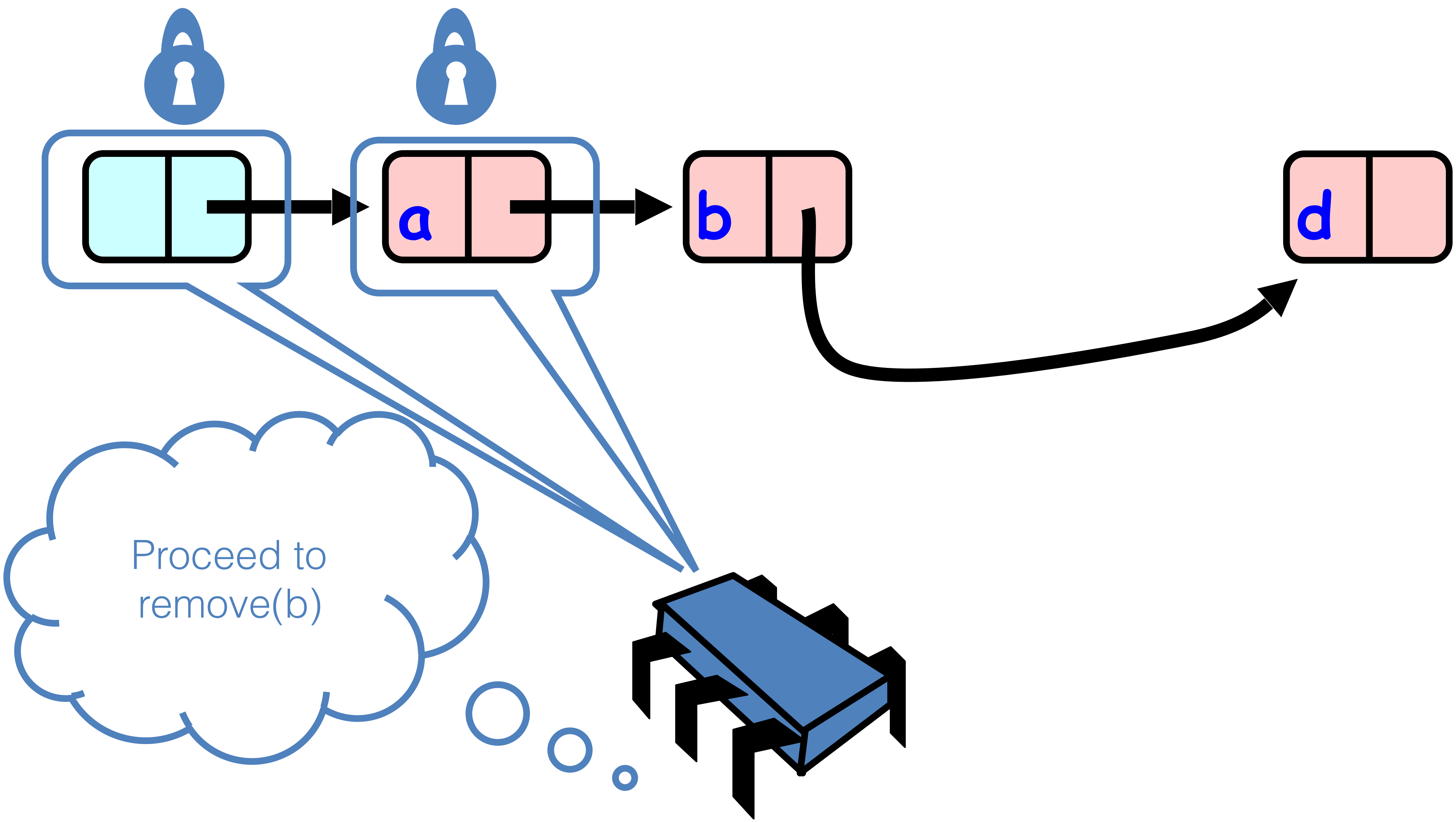
Removing a Node



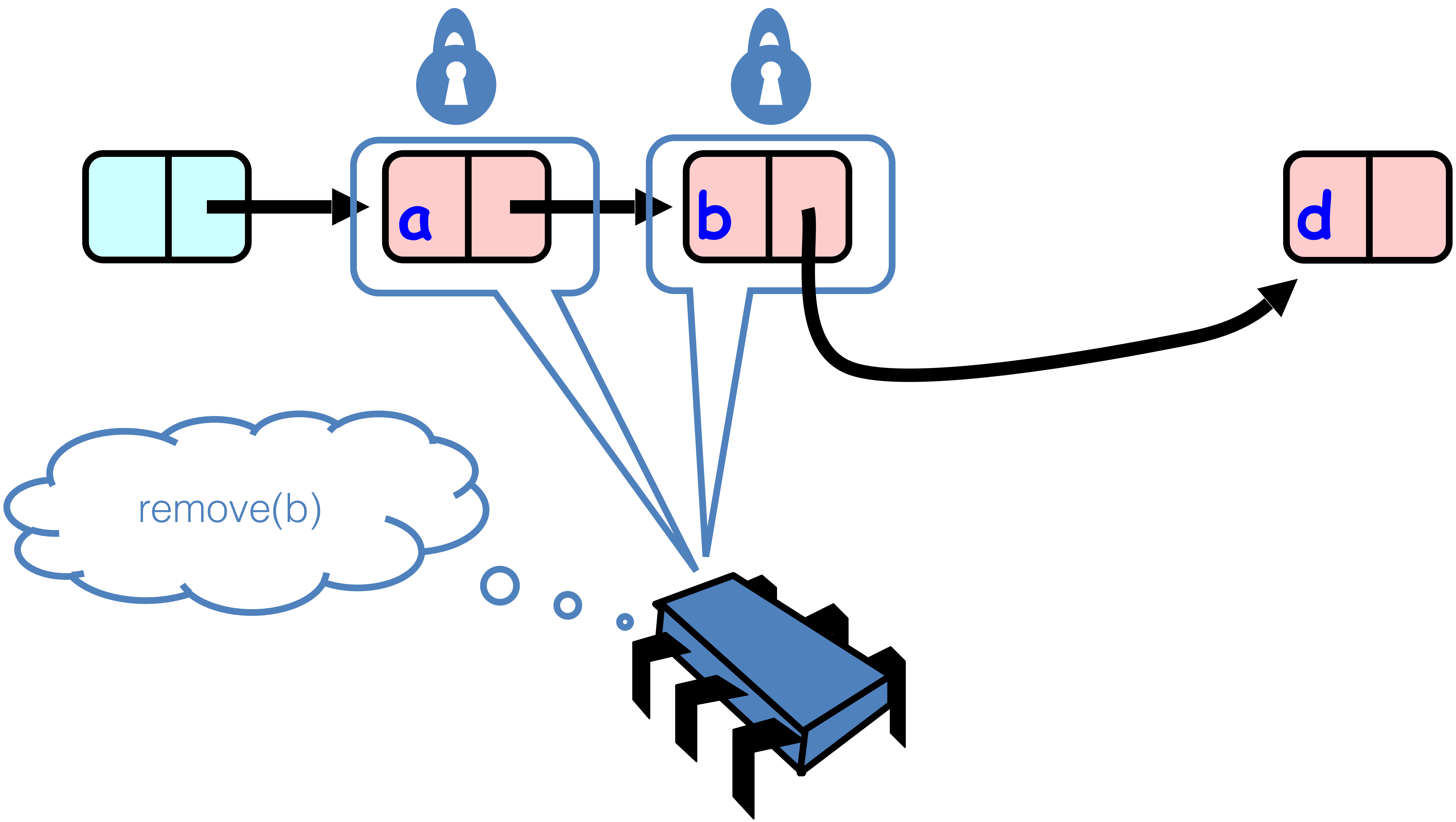
Removing a Node



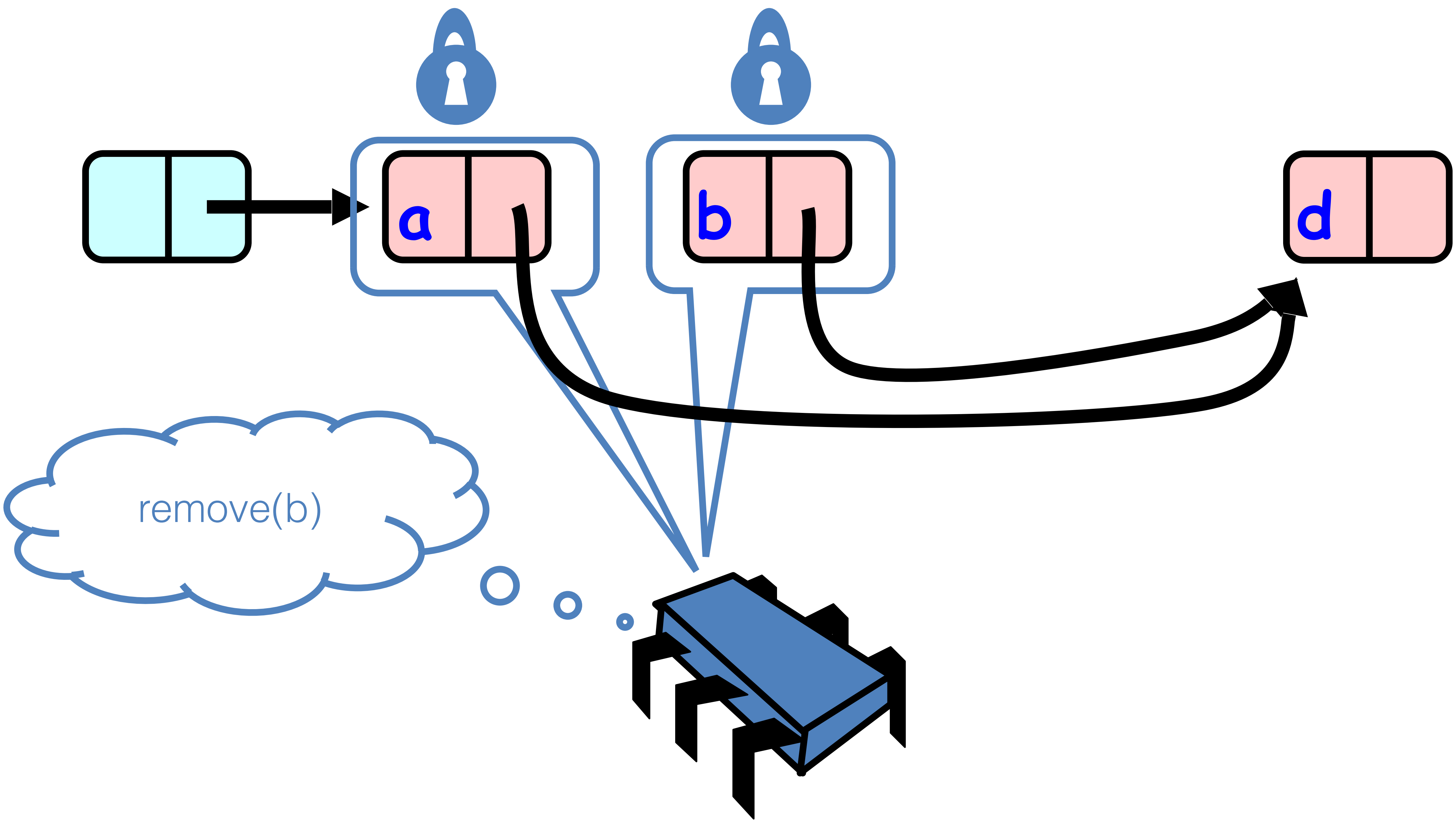
Removing a Node



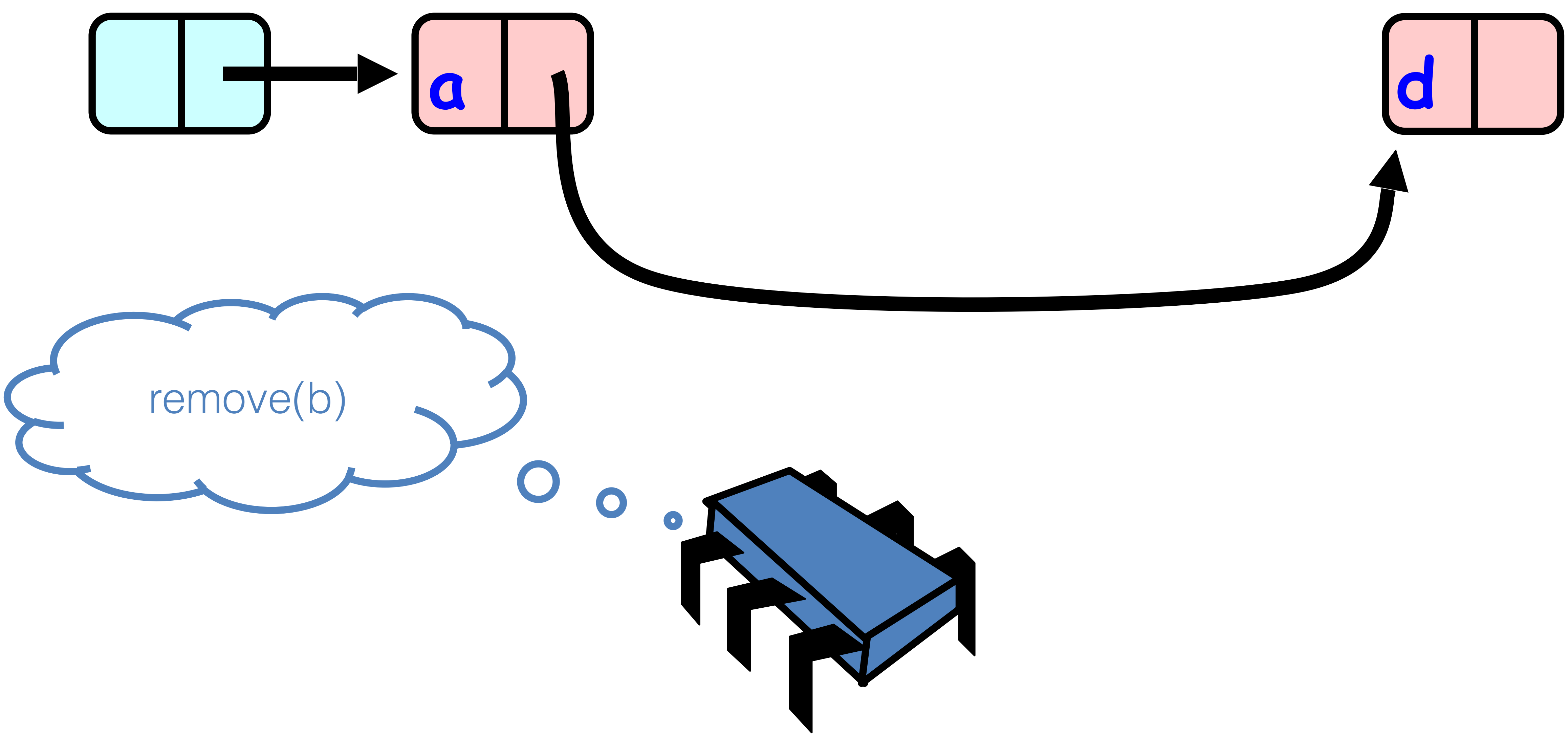
Removing a Node



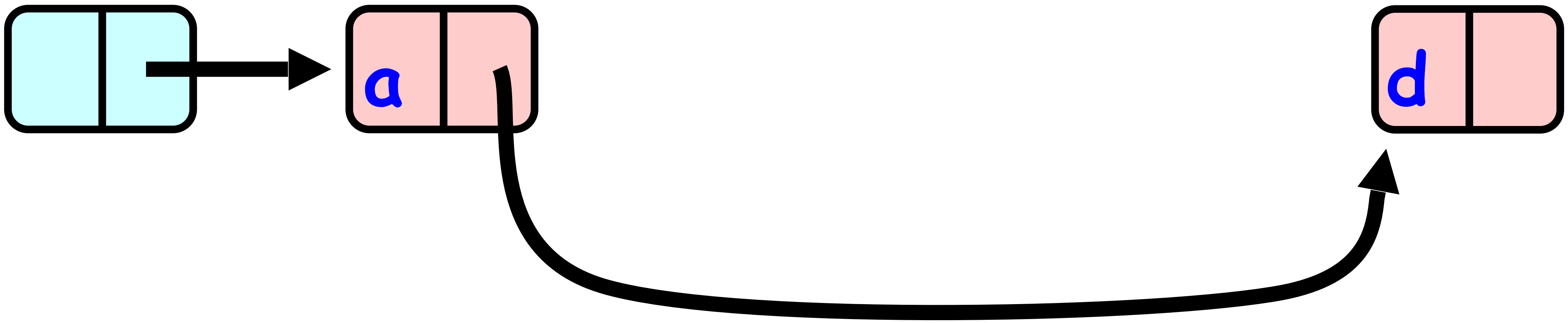
Removing a Node



Removing a Node



Removing a Node



Adding Nodes

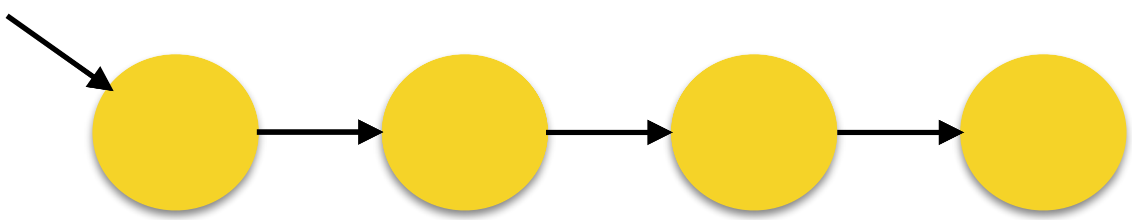
- To add node e
  - Go hand-over-hand
  - Lock predecessor
  - Lock successor
- Neither can be deleted
- Actually it is enough to lock predecessor (for an insert).
  - But must go hand-over-hand.

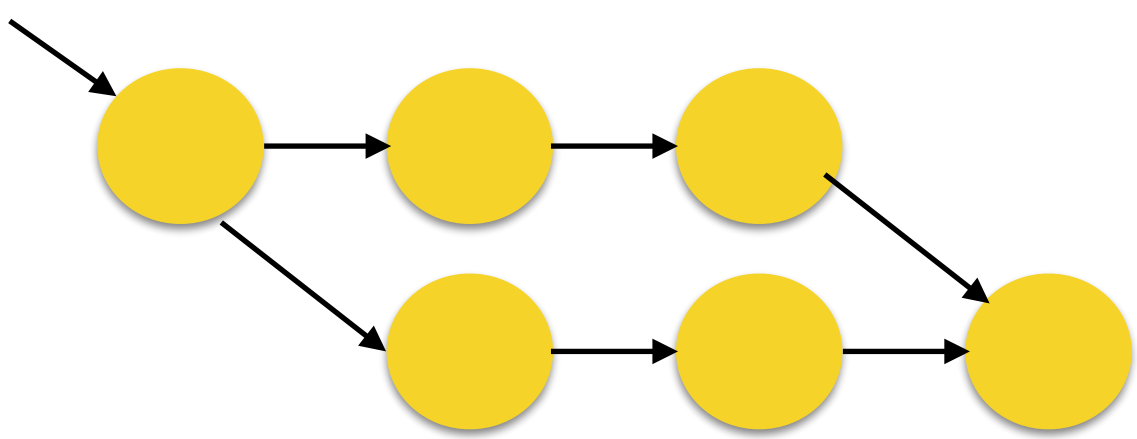
No Deadlock

- In general, no deadlock if locks are always acquired in the same order.

Why Is It Correct

- The idea: snapshot.
- Each thread sees all operations executed “earlier”, and no operation that started “afterwards”.
- Start time: take head’s lock.
- Implications:
  - sequentialization of operations
  - Good only for “hierarchical” data structures.





Properties

- Scalability better than coarse-grained locking.
- But long chains of threads waiting for the first thread to advance.
  - Limited parallelism.
- Excessive locking harms performance.

- Can we obtain more parallelism and better performance?

Second List: Optimistic

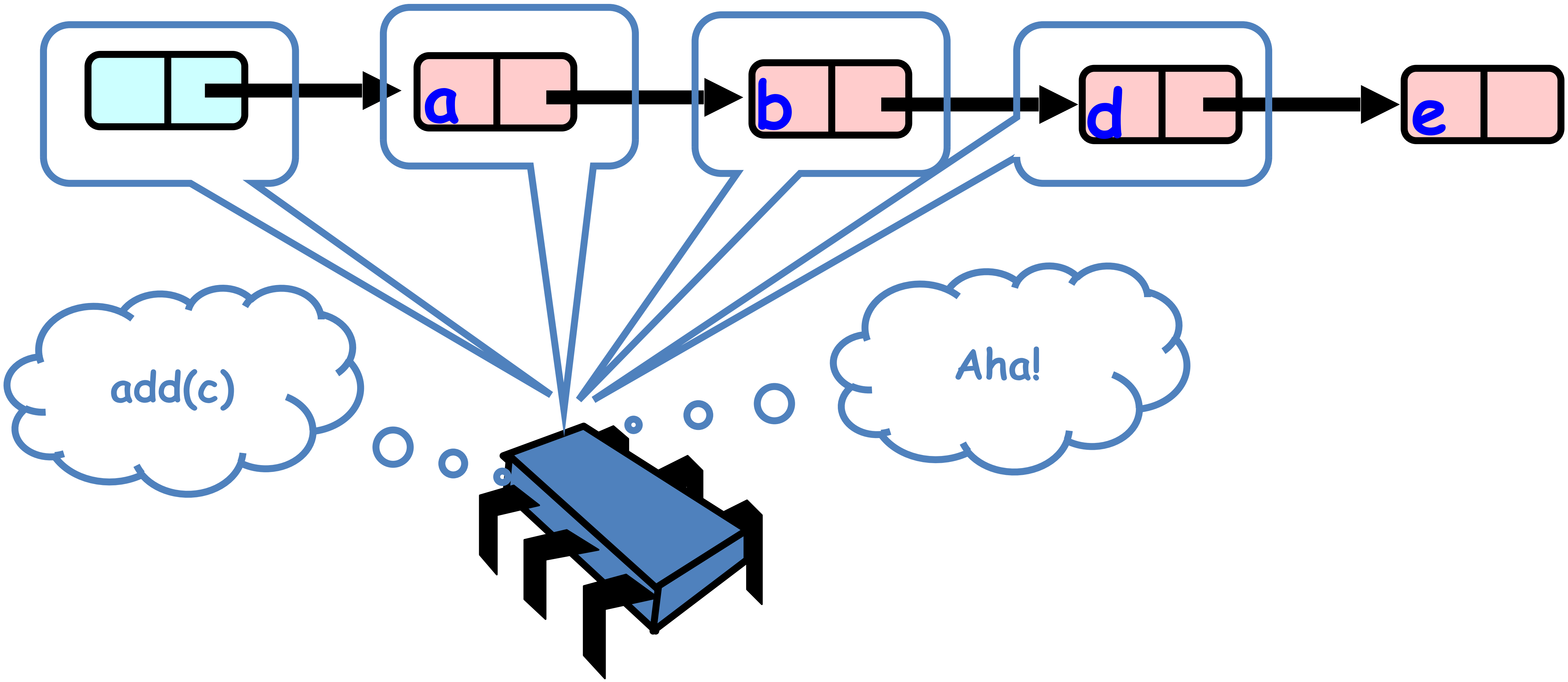
(First was hand-over-hand.)

Optimistic Synchronization

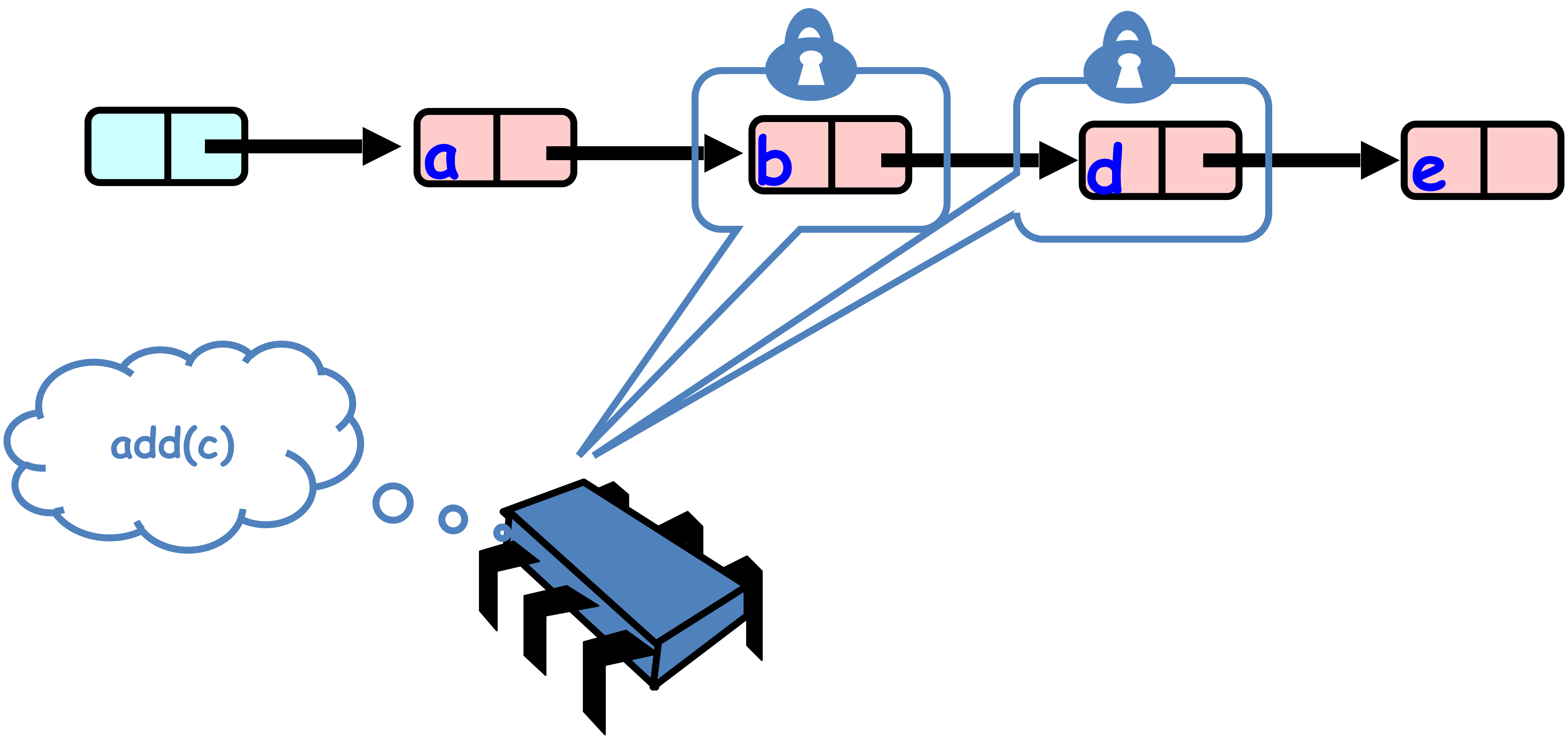
[Herlihy-Shavit 2008]

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

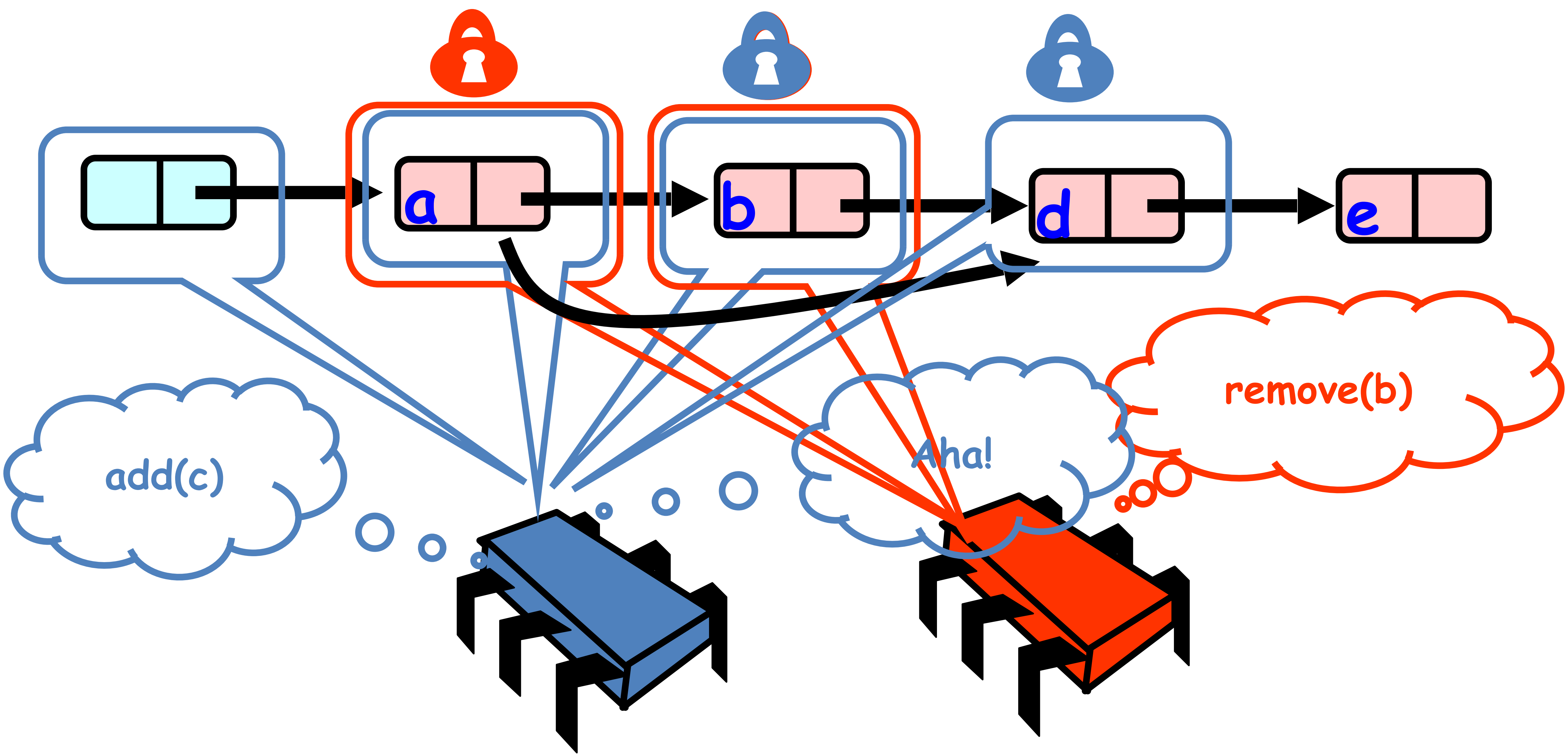
Optimistic: Traverse without Locking



Optimistic: Lock and Load



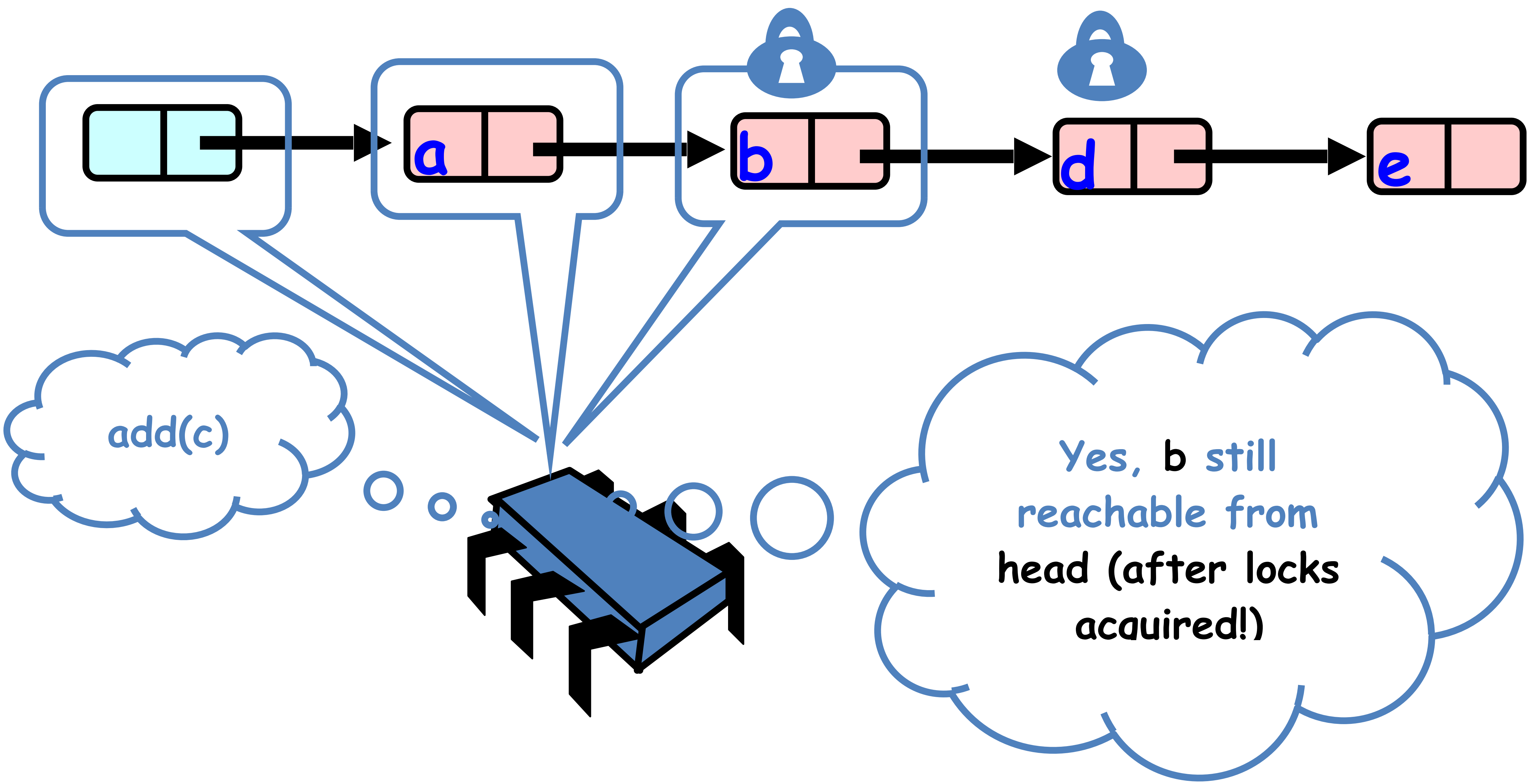
What could go wrong?



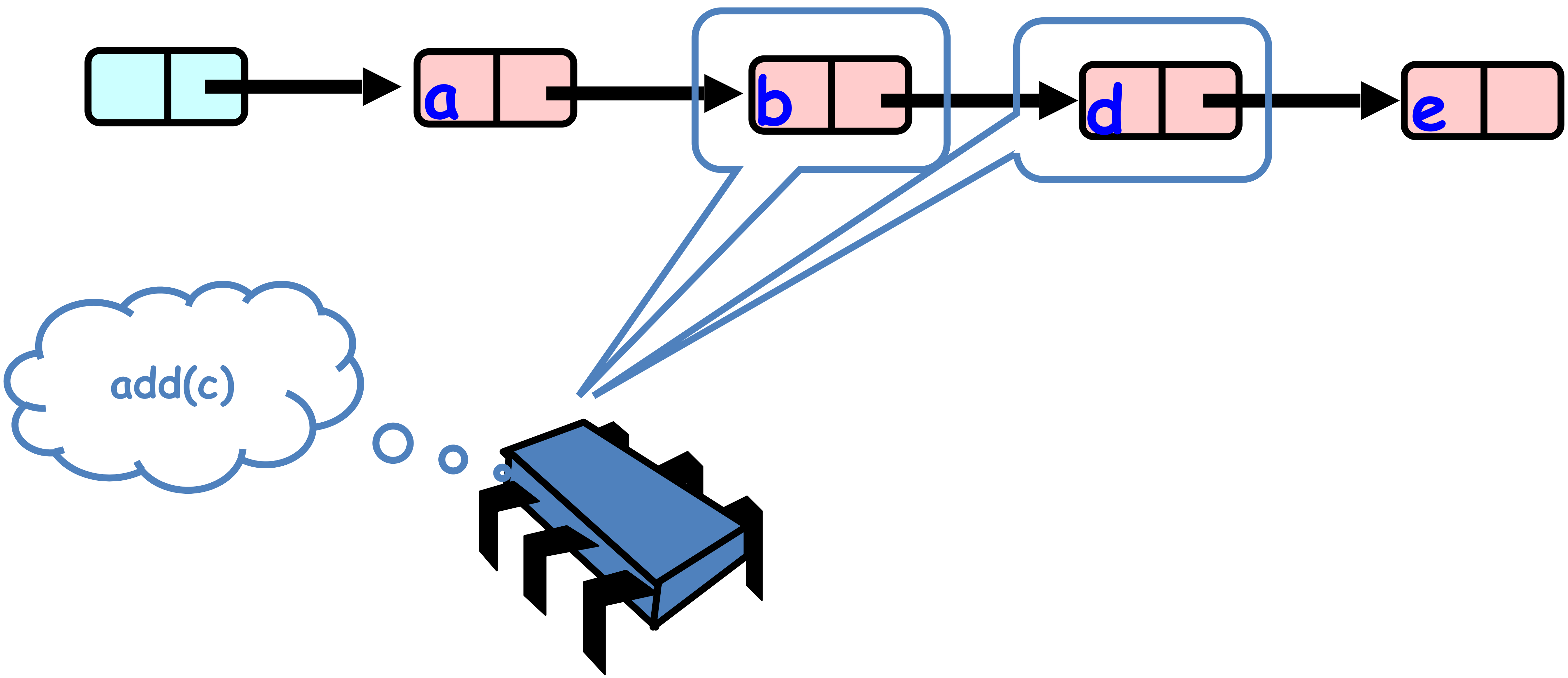
First node must be in the list!

- While holding the lock, check that first node is reachable from the head
- While we hold the lock this node cannot be removed.

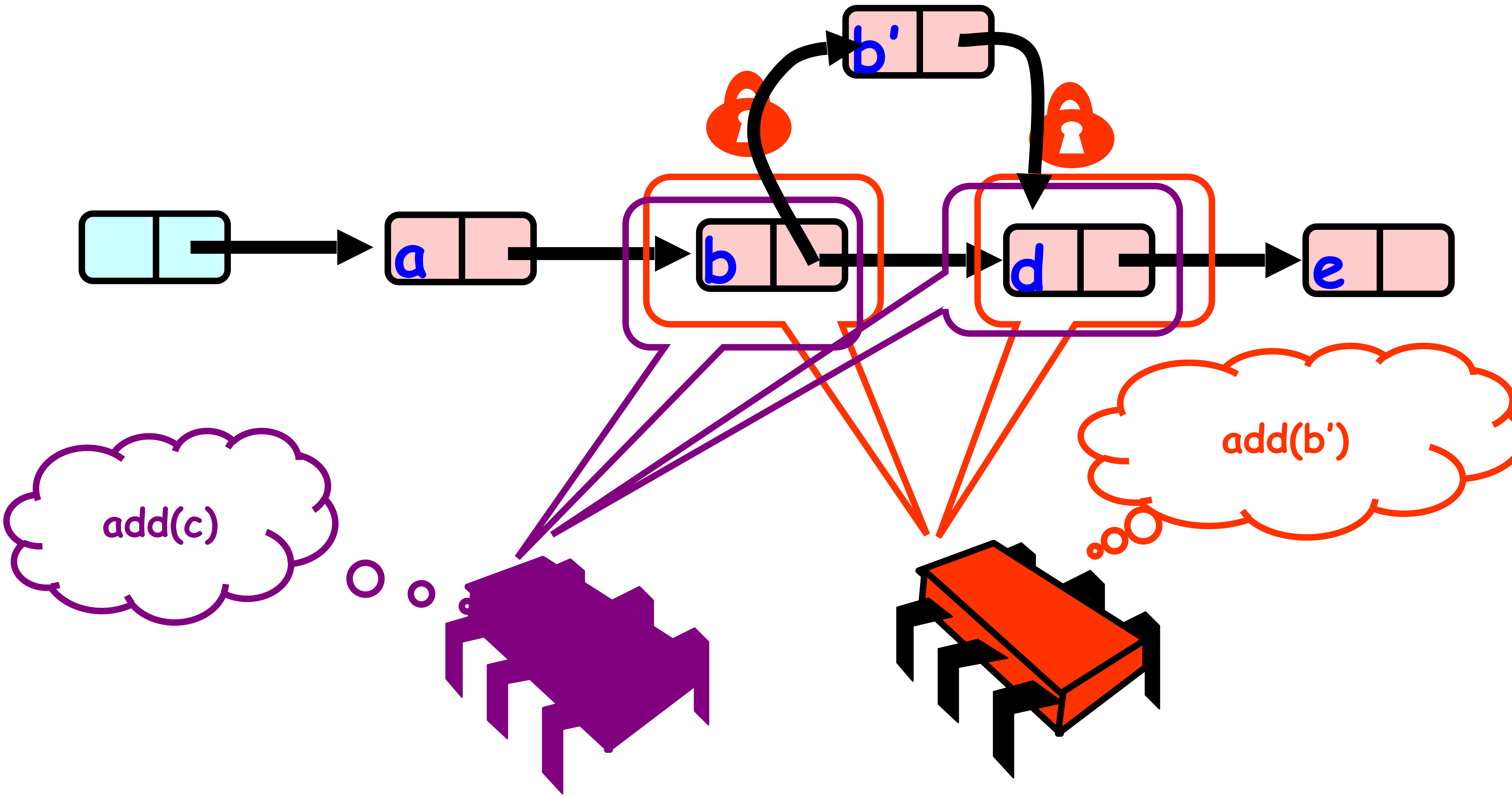
Validate – Part 1 (while holding locks)



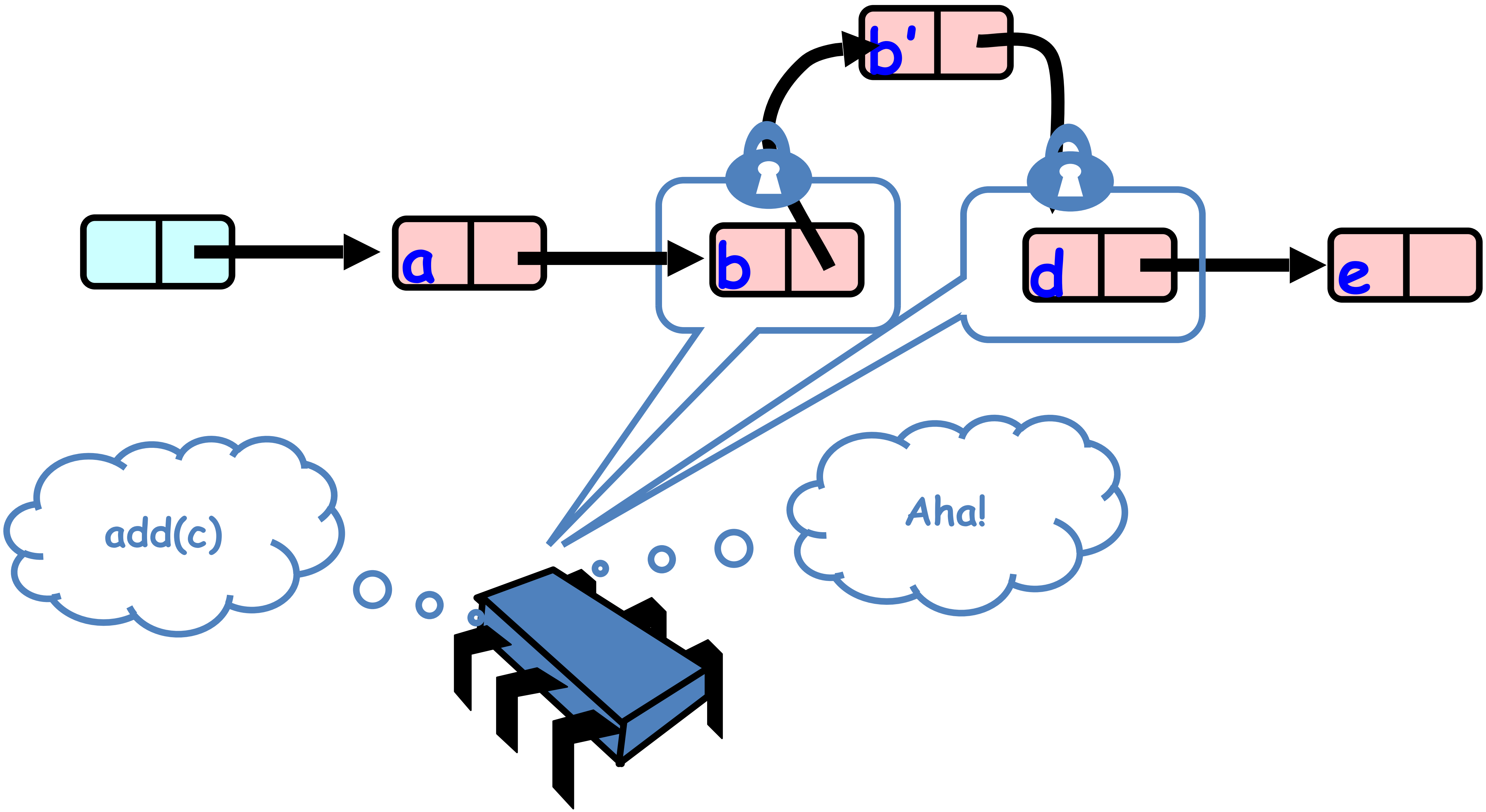
What Else Can Go Wrong?



What Else Can Go Wrong?



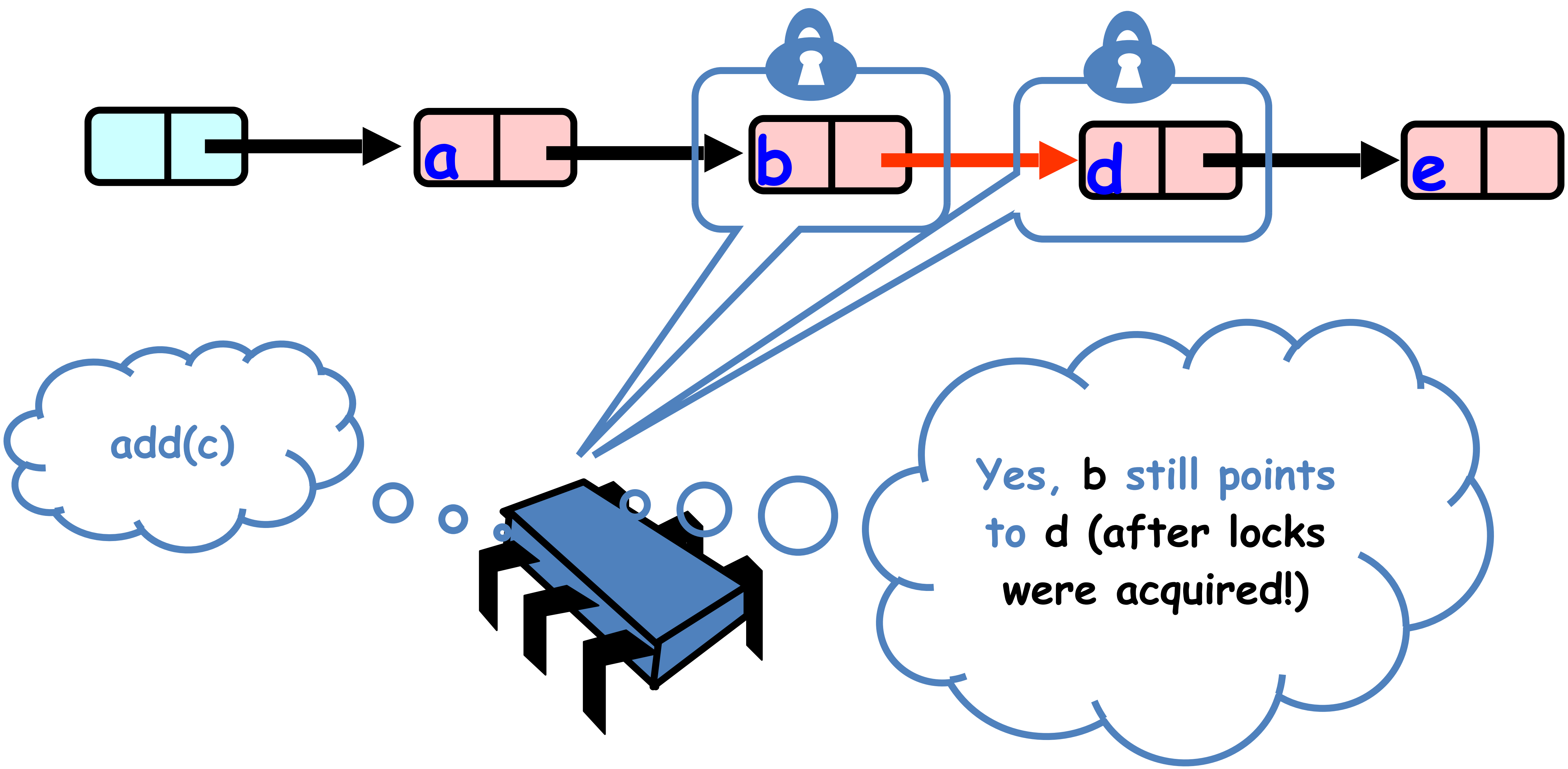
What Else Can Go Wrong?



First node must still point to second!

- Validation 1: first node still reachable.
  - While we hold the lock the node cannot be removed.
- Validation 2: first node pointing to second.
  - While we hold the lock the pointer from the first node cannot be modified (no adding and no removing).

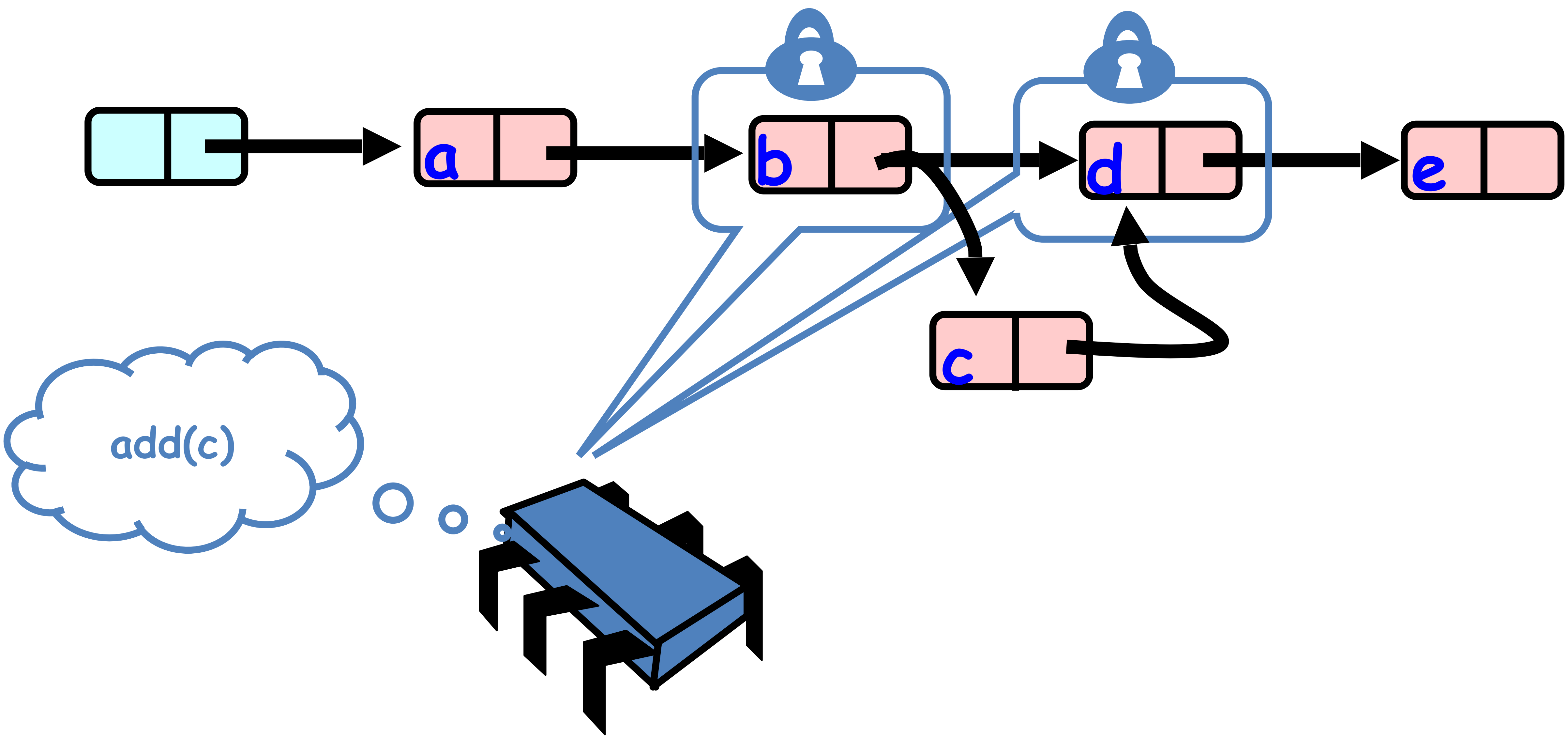
Validate Part 2
(while holding locks)



Validation Failure?

- Upon failure to validate start from scratch.
- Assumed to happen infrequently.

Insert (After Validation)



Optimistic Synchronization

- More parallelism, better scalability.
  - Only lock nodes where actually modifying.
  - Scalability depends on the actual workload.

- Excessive work on validation (double traversal).
  - Less efficient.

- There is another fine-grained locking methodology.
  - But let’s jump to lock-freedom

Third List: Lock-Free

We did hand-over-hand and optimistic

Lock-Freedom

- Don’t use locks.
- And more important: guarantee progress!
  - Complete robustness against worst-case scheduling
  - No swapping problems
  - Even when a thread dies, the other threads will continue to make progress.

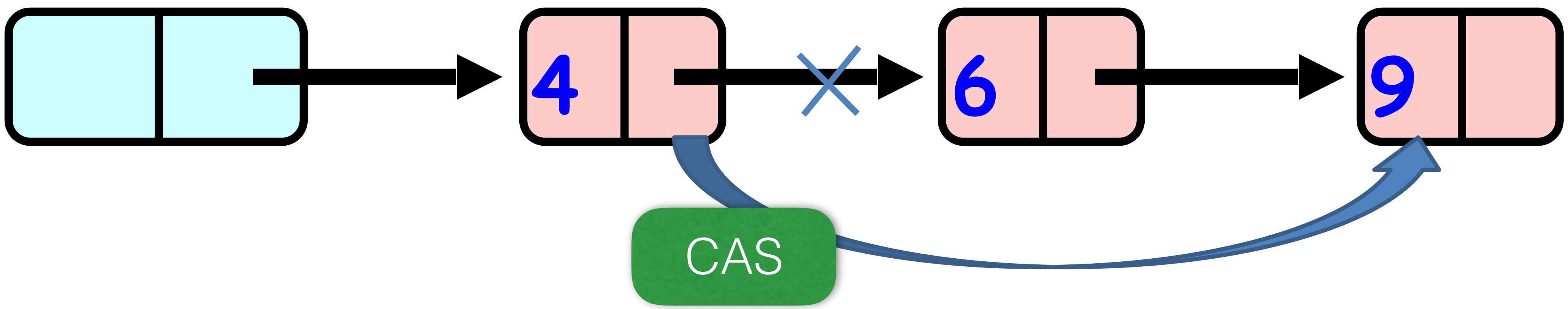
- Design by [Harris 2001], improvement by [Michael 2002].

Lock-Free Linked Lists
[Harris-Michael 2001-2]

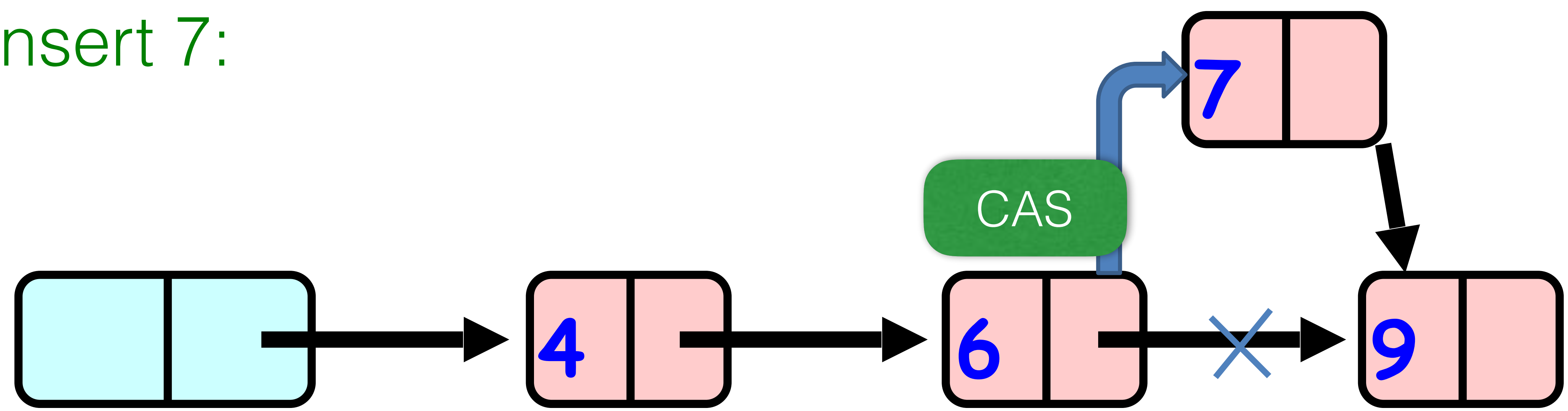
- First attempt: insert/delete using CAS instead of a regular read/write operation.

First Attempt: Use CASes Instead of Locks

- Delete 6:

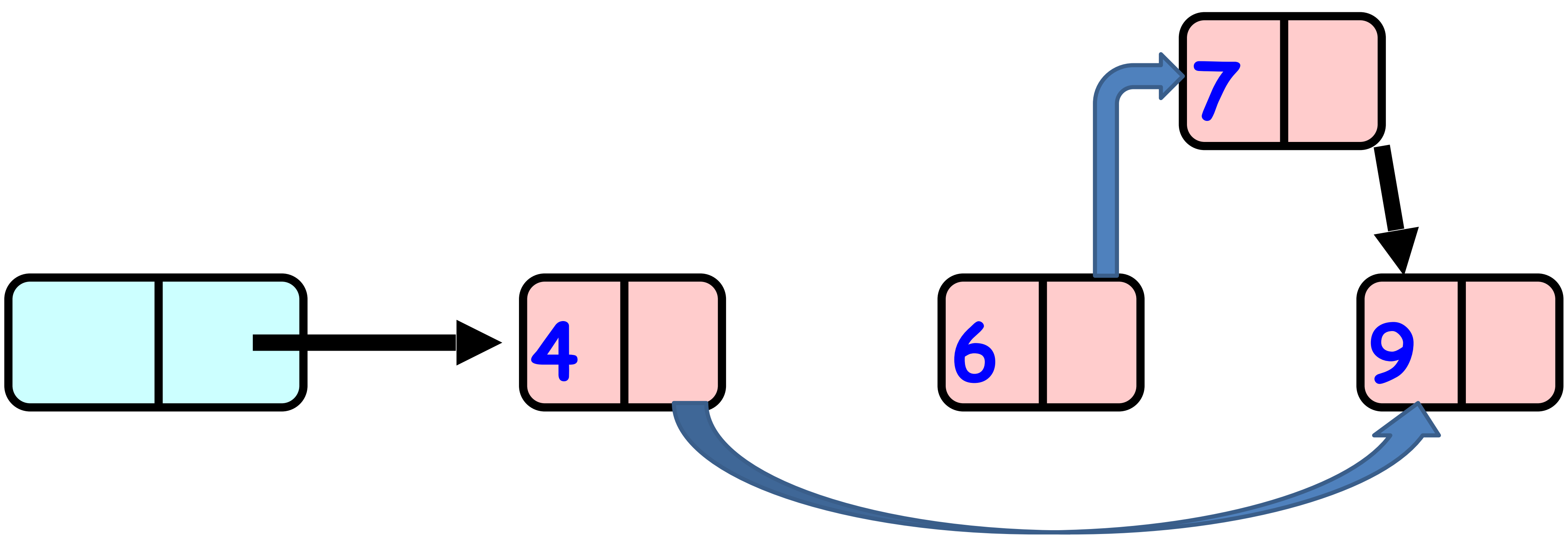


- Insert 7:



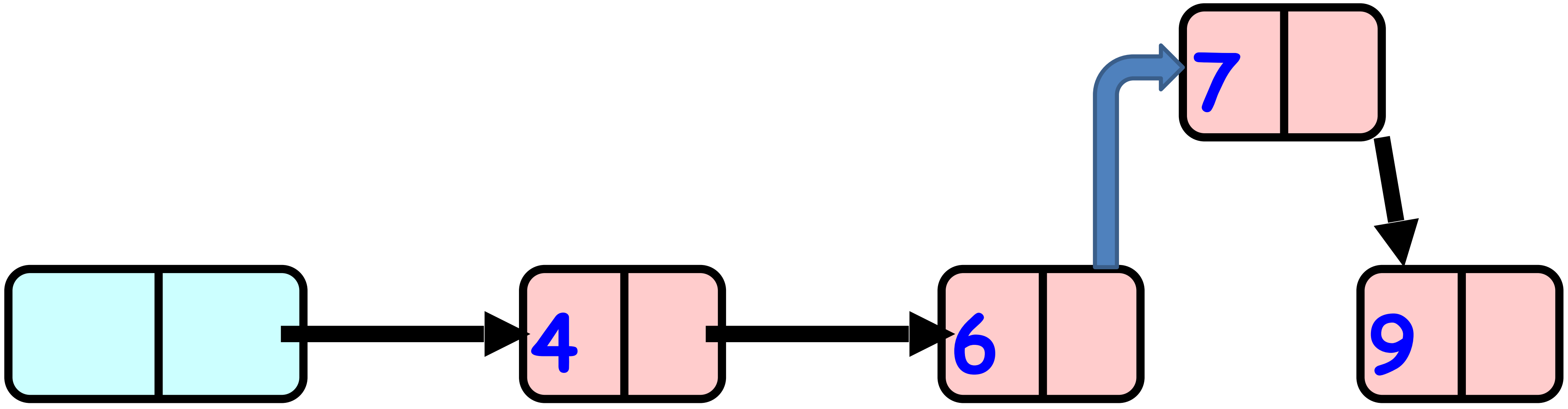
The original problem still exists

- Outcome for deleting 6 and inserting 7 in parallel:



We Keep the Simple Insert

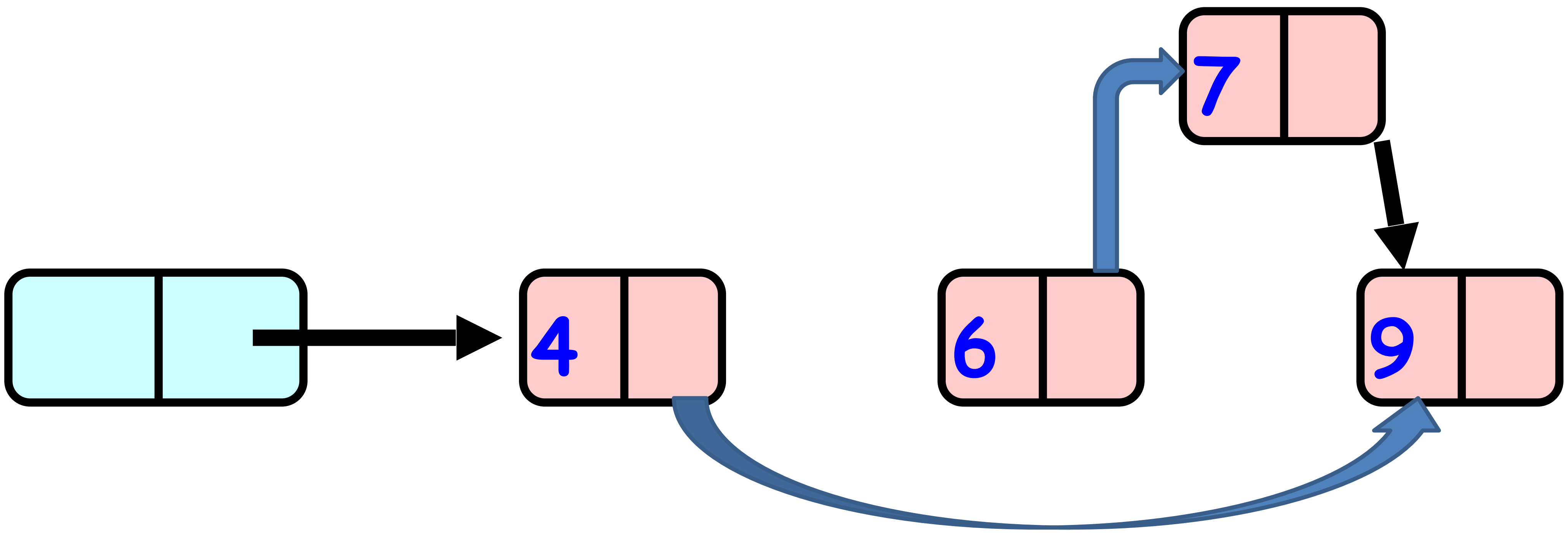
- A single CAS to insert 7, after locally allocating and initializing it.



- But *delete* will be more complicated.

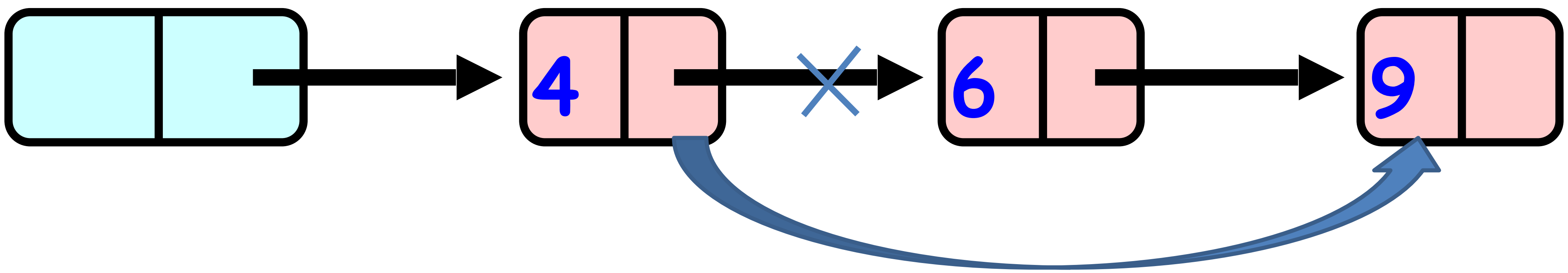
Recall the Problem

- Outcome for deleting 6 and inserting 7 in parallel:



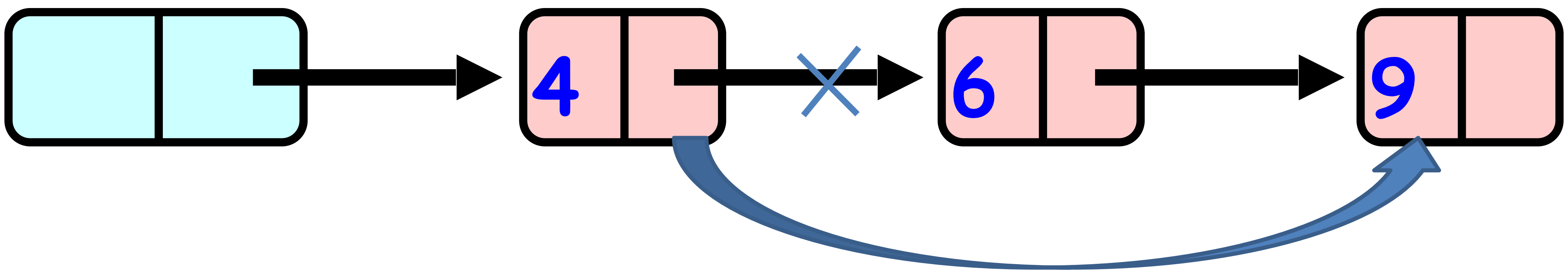
The Crux of the Problem

- When deleting 6, we want to block changes both on the pointer that points at 6, as well as the pointer that points out of 6.



The Crux of the Problem

- When deleting 6, we want to block changes both on the pointer that points at 6, as well as the pointer that points out of 6.

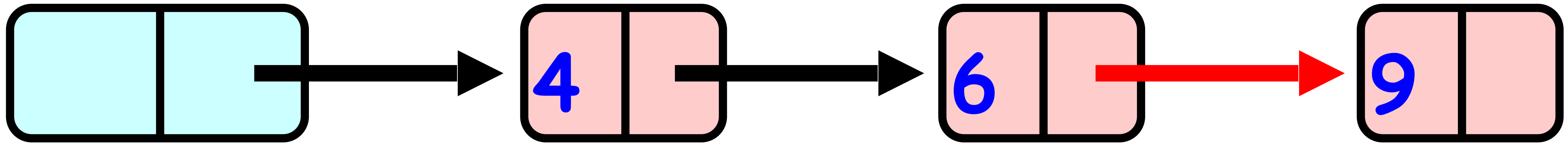


- Harris's idea:

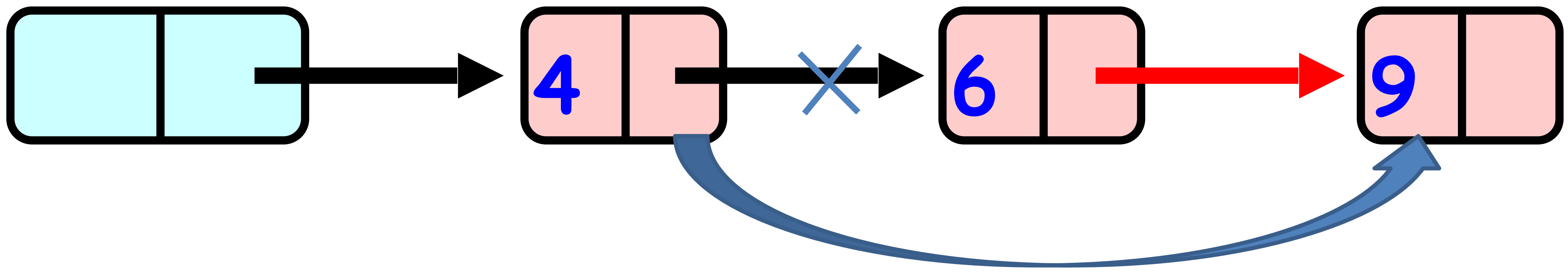
1. "mark" the pointer out of 6, and then
2. "modify" the pointer out of 4.

Solution: Mark & Delete

- Logically delete 6 by marking the outgoing pointer of 6.



- Physically delete 6 by unlinking it from the list.



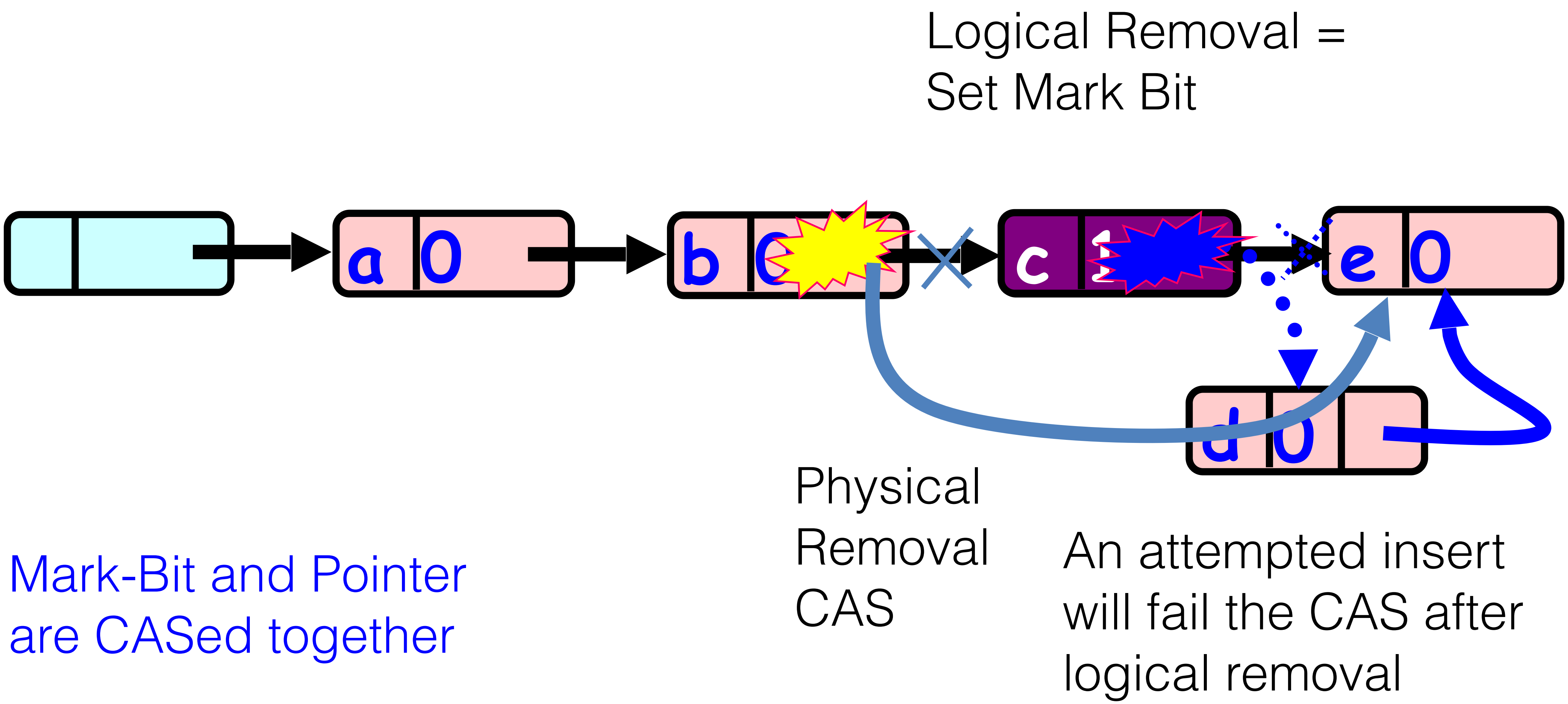
Implementing a “Red Pointer”

- Use least bit.
- Essentially unused with pointers as words are composed of 4 or 8 bytes.

Unmarked: 00010…10100101011100000100

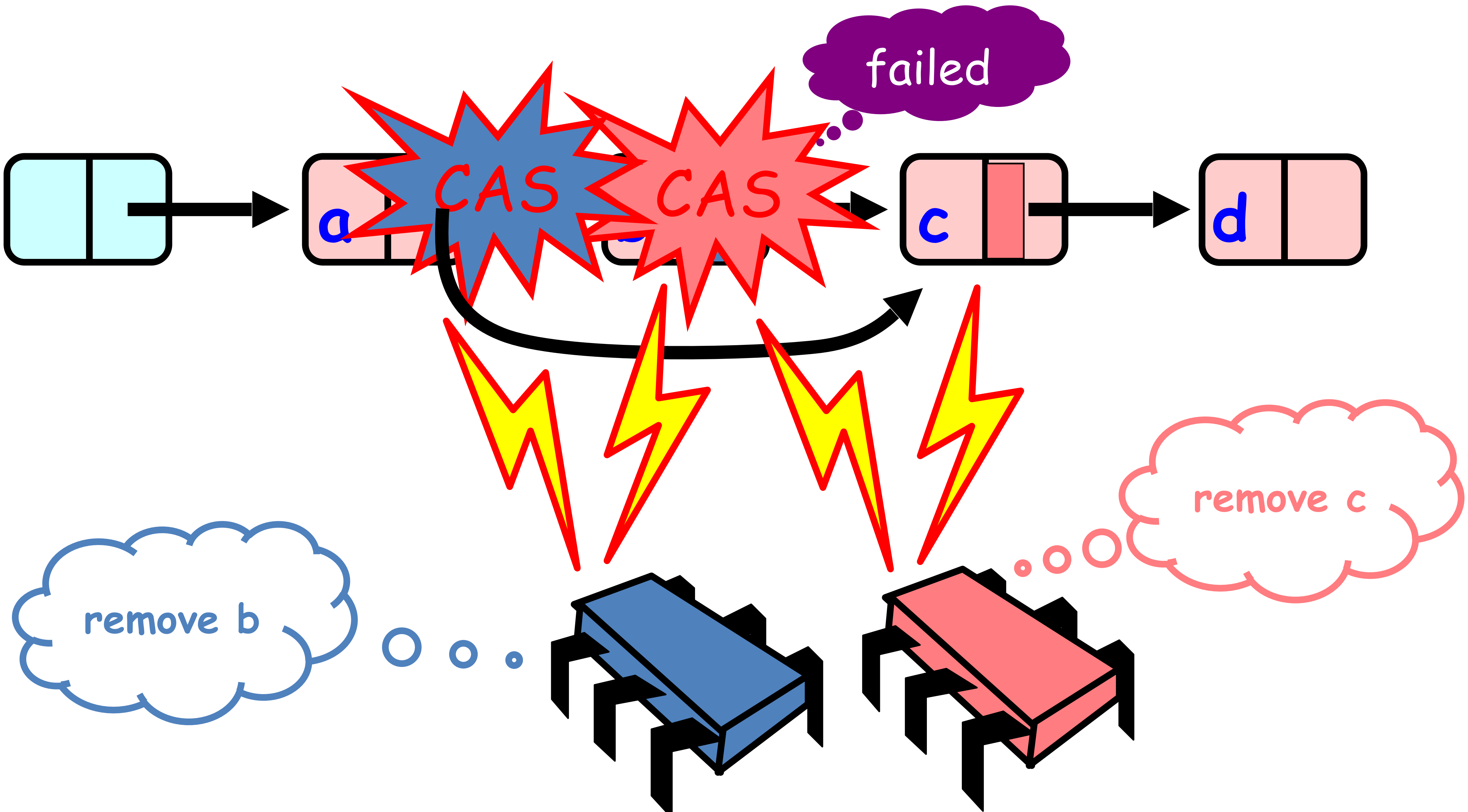
Marked: 00010…10100101011100000101

Logical Deletion

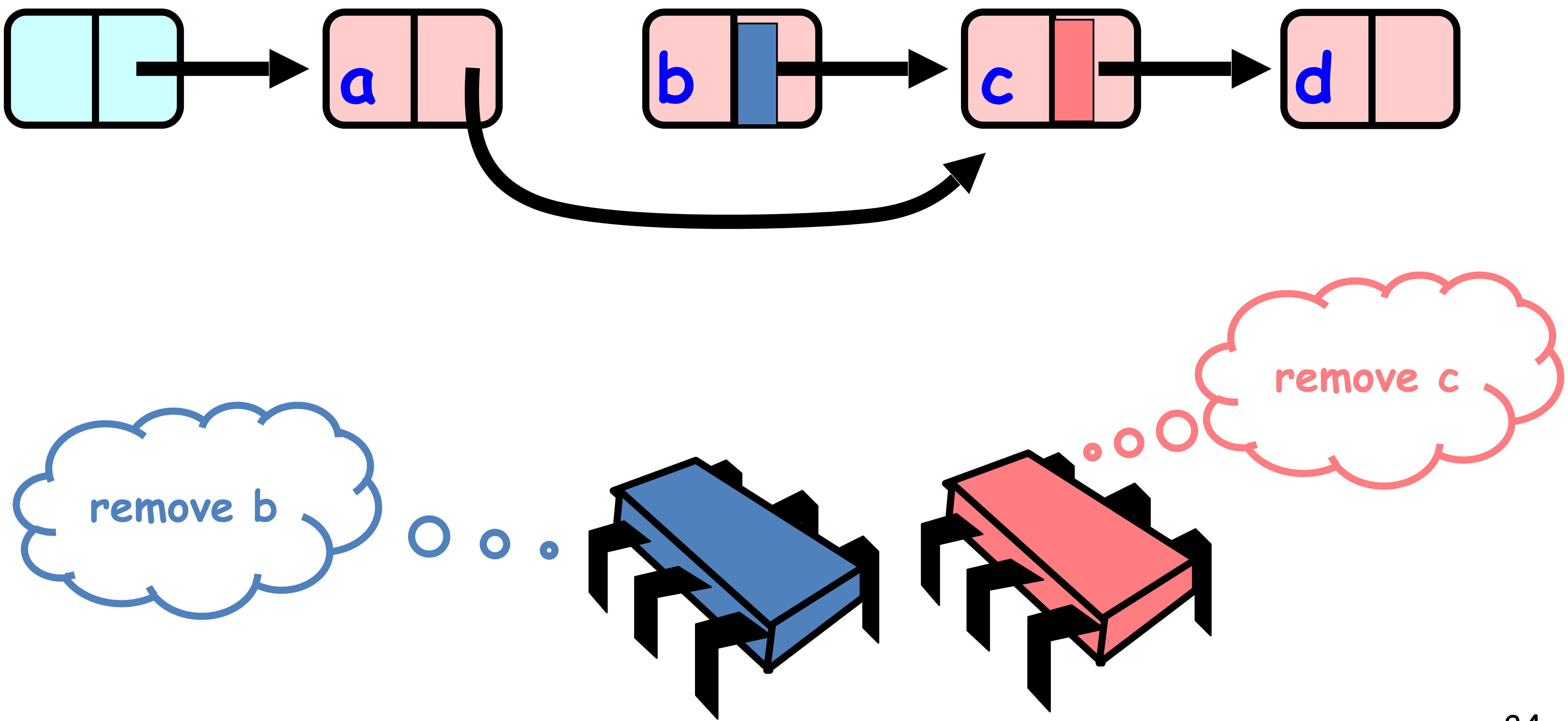


Mark-Bit and Pointer
are CASed together

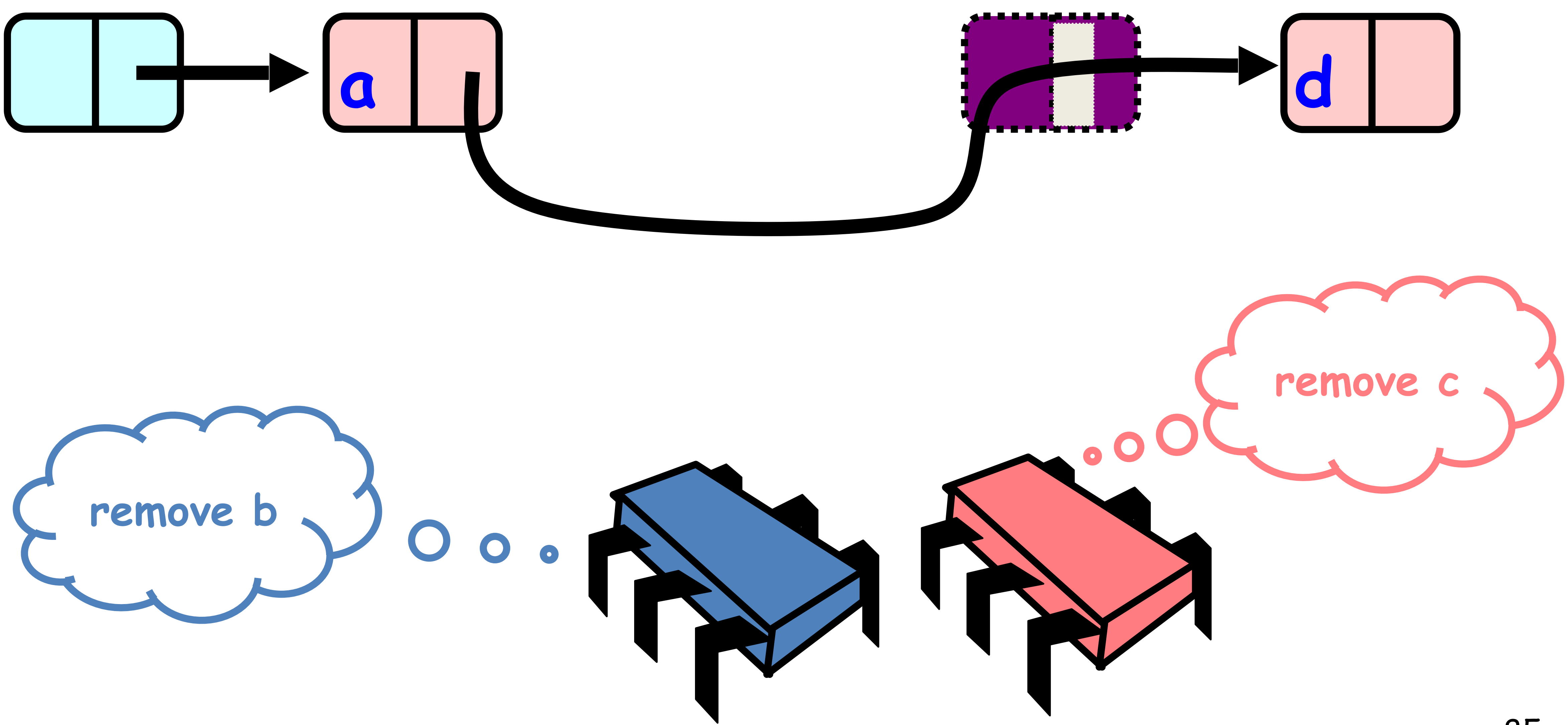
Concurrent Removal



Removing a Node



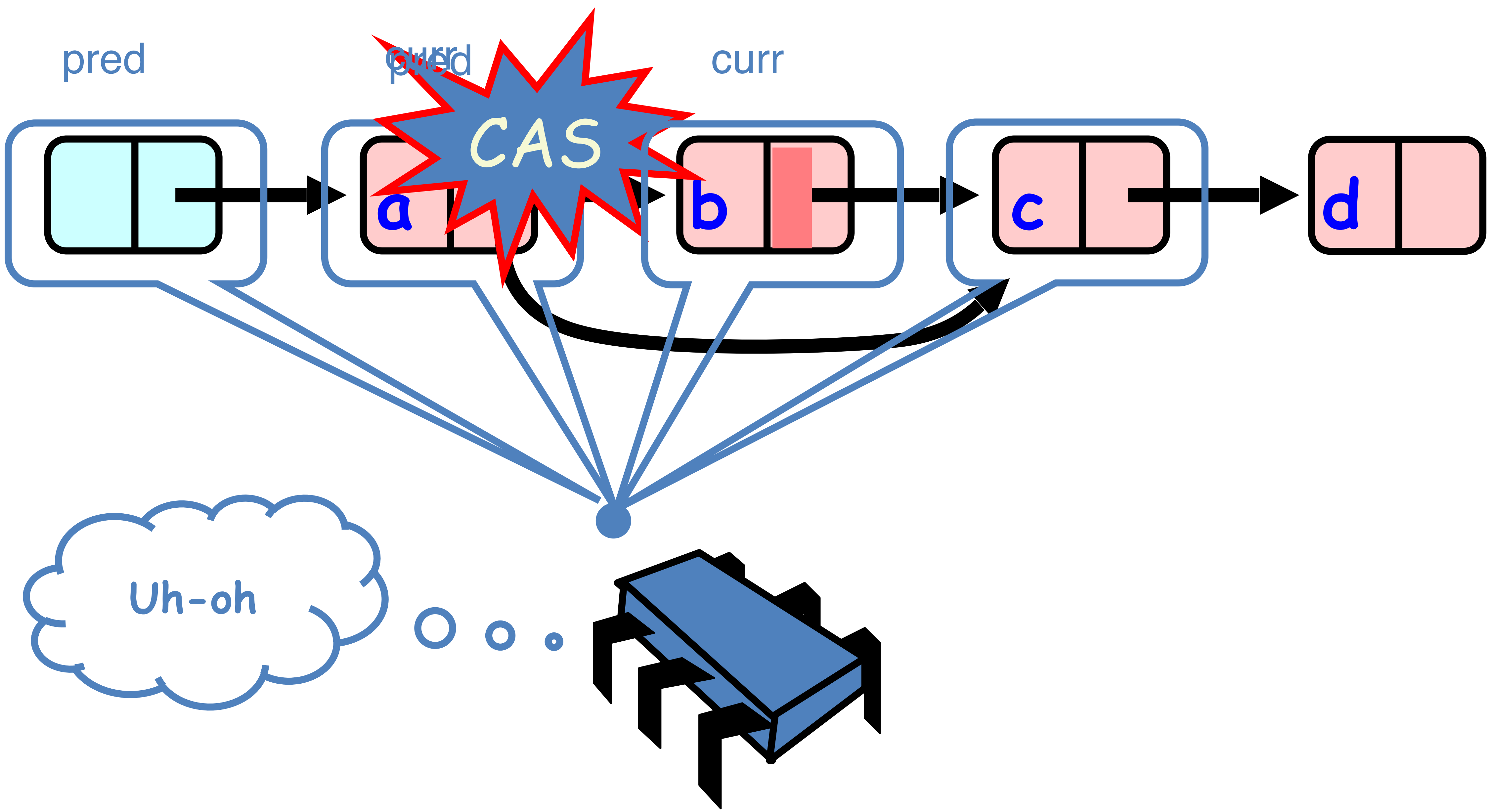
Removing a Node



Traversing the List

- When you find a “logically” deleted node in your path:
  - Finish the job:
    - CAS the predecessor’s next field,
  - Proceed (repeat as needed).

Lock-Free Traversal (only Add and Remove)



CAS Failures

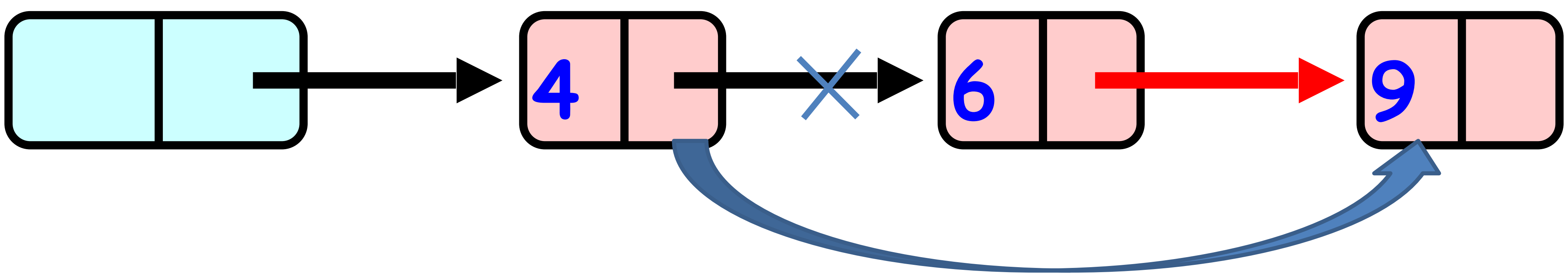
- Node removal:
  - Logical remove fails: start from scratch.
  - Physical remove fails: ignore.
    - Why?
- Node insert:
  - CAS fails: start from scratch.

Why is it Lock-Free?

- Node removal:
  - Logical remove fails:
    either someone else has succeeded to remove this node, or someone else has inserted a node.
  - Logical remove succeeds:
    I succeeded to delete a node (and will finish the operation after trying the physical remove once).
- Node insert:
  - CAS fails: someone else succeeded to insert or delete a node.
  - CAS succeeds: I succeeded in inserting a node.

The Main Intuition

- Logical marking locks the next pointer from being modified.
- But this “lock” can be unlocked by anyone (by trimming the node from the list), so no one is stuck.
- Different from “normal” locking that only the owner can unlock.
- This is the methodology in all lock-free algorithms:
  - Make a change in the data structure, leaving it “unstable”.
  - Anyone can stabilize the data structure and continue to work on it.



Progress Guarantees are good for

- Real-time, OS, interactive systems, service level agreements, etc.
- But it's always good to have.
  - Avoid deadlock, live-lock, convoying, priority inversion, etc.
- Scalability.

Progress Guarantees

Great guarantee!
Until recently considered difficult to achieve and inefficient.

Wait-Freedom

If you schedule enough steps of *any thread*, *it* will make progress.

Lock-Freedom

If you schedule enough steps across *all threads*, one of them will make

Contains is Wait-Free

- Contains(key);
  - curr = head;
  - while (curr.key < key)
    - curr = removeMark (curr.next)
    - succ = removeMark (curr.next)
  - return (curr.key == key && !marked(curr.next))

Contains is important!

Fourth List:

Third (and Best) Fine-Grained Locking:

Lazy List

We discussed hand-over-hand, optimistic, and lock-free.

Lazy Synchronization

[Heller et al. 2005]

- Lock only relevant nodes
- Do not validate reachability
- Instead, leave a mark on deleted nodes, like in lock-free algorithm.

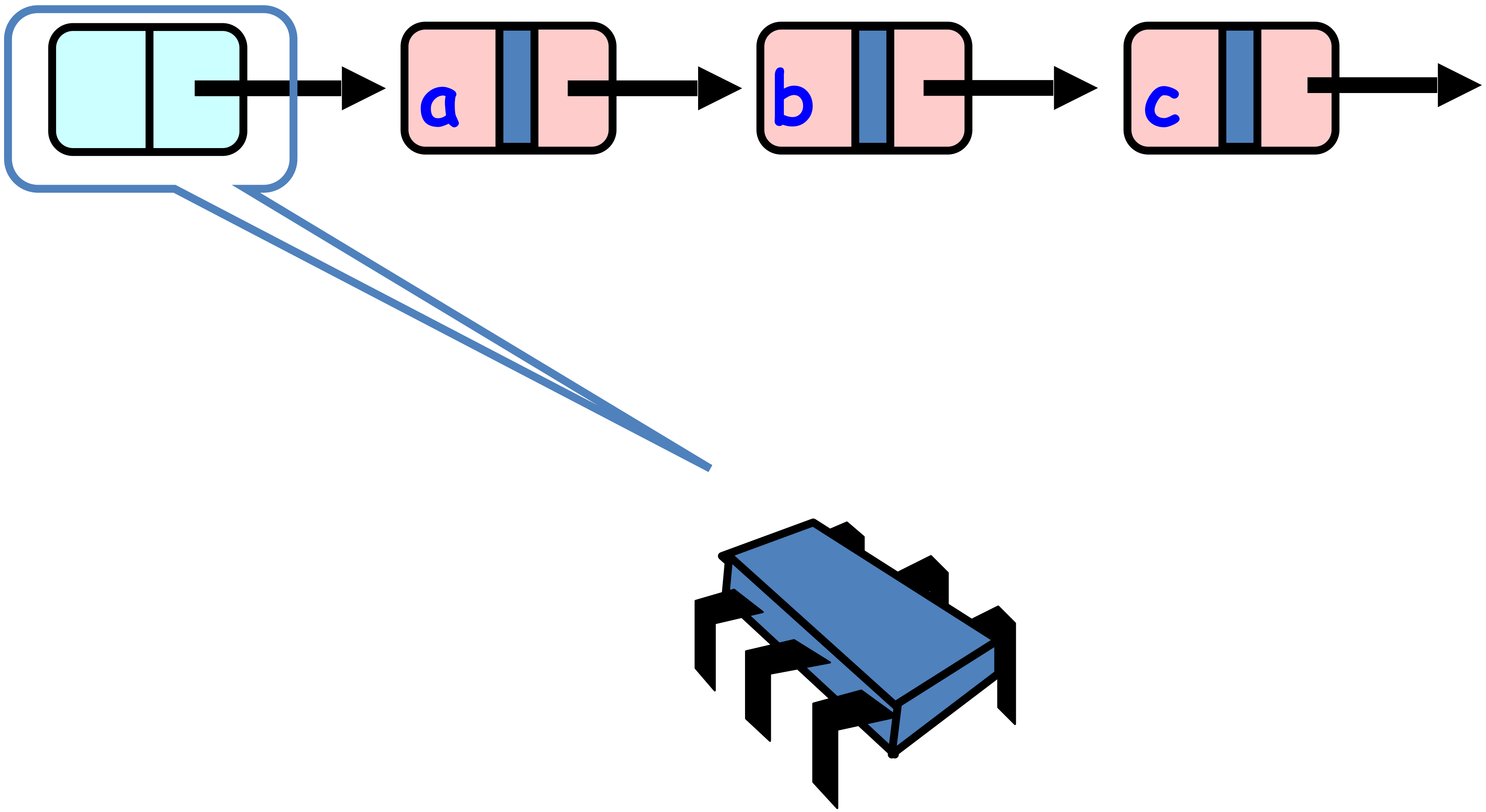
- To remove a node:
  - Logically remove it by marking it “removed”.
  - Physically remove it by unlinking it.

Remove or Add

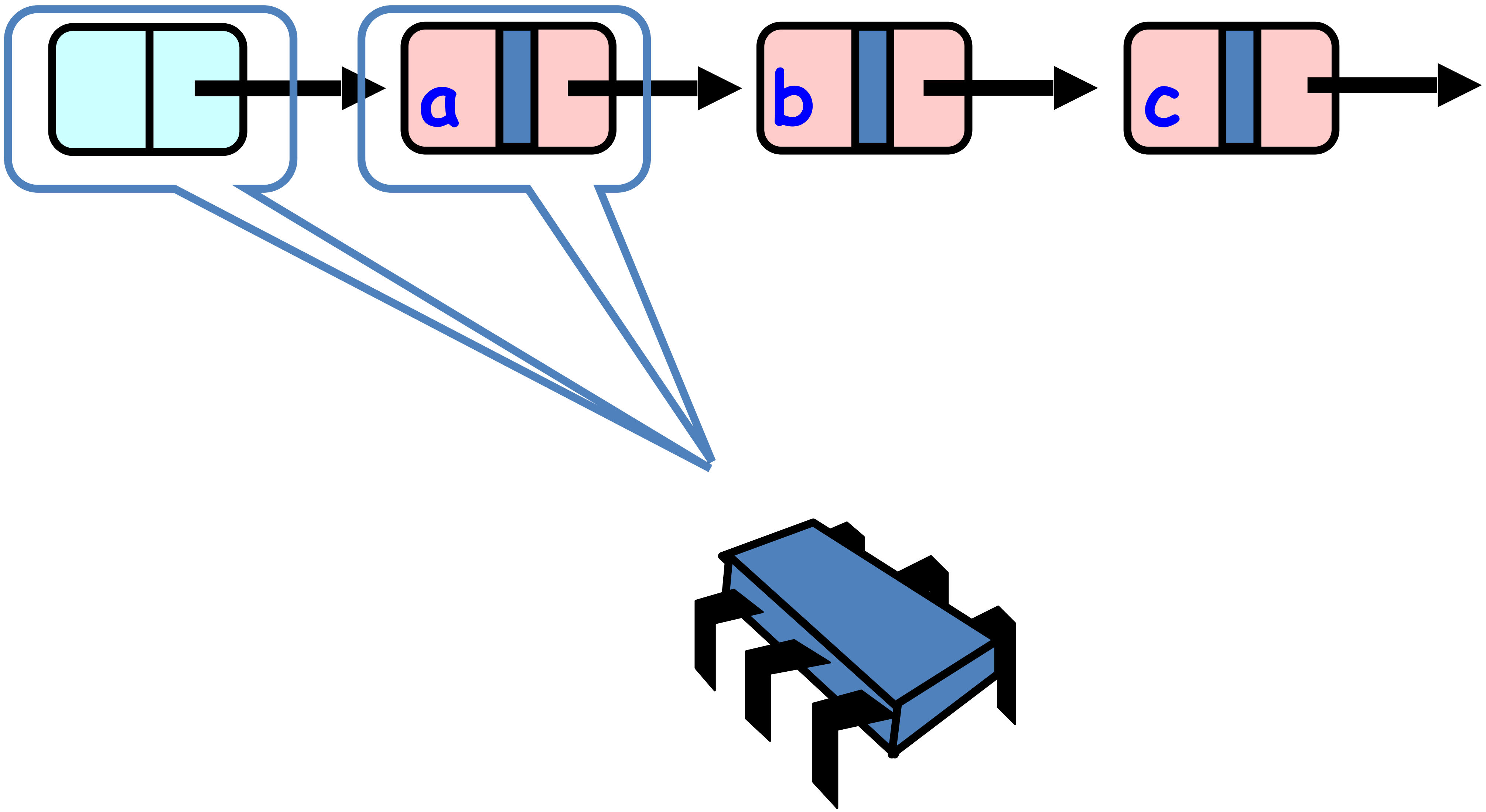
- Scan through the list
- Lock predecessor and current nodes
- validate that
  - both are not deleted, and that
  - predecessor points to current.
- Perform the add or the remove.

- Search can simply traverse the list.

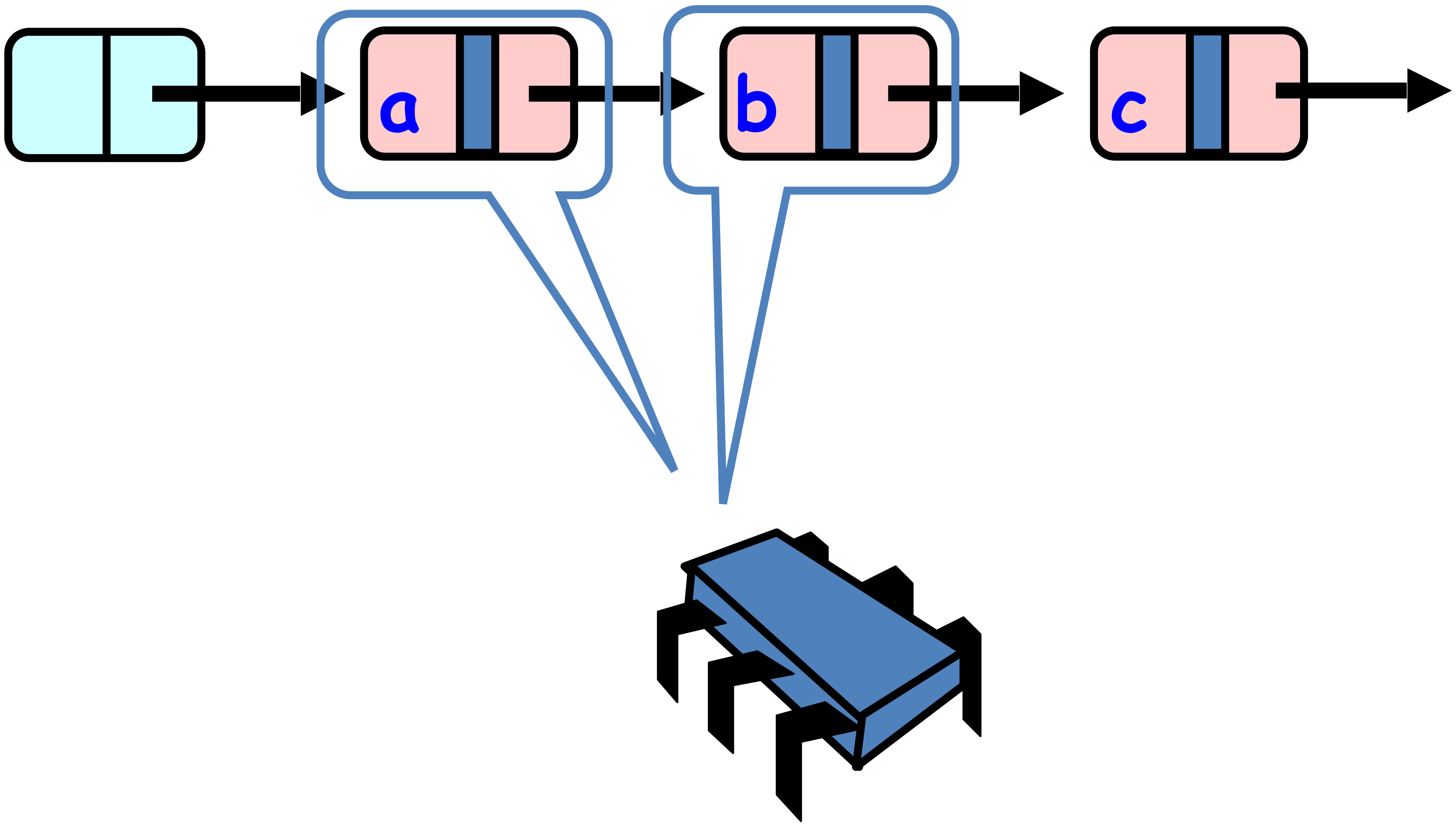
Lazy List



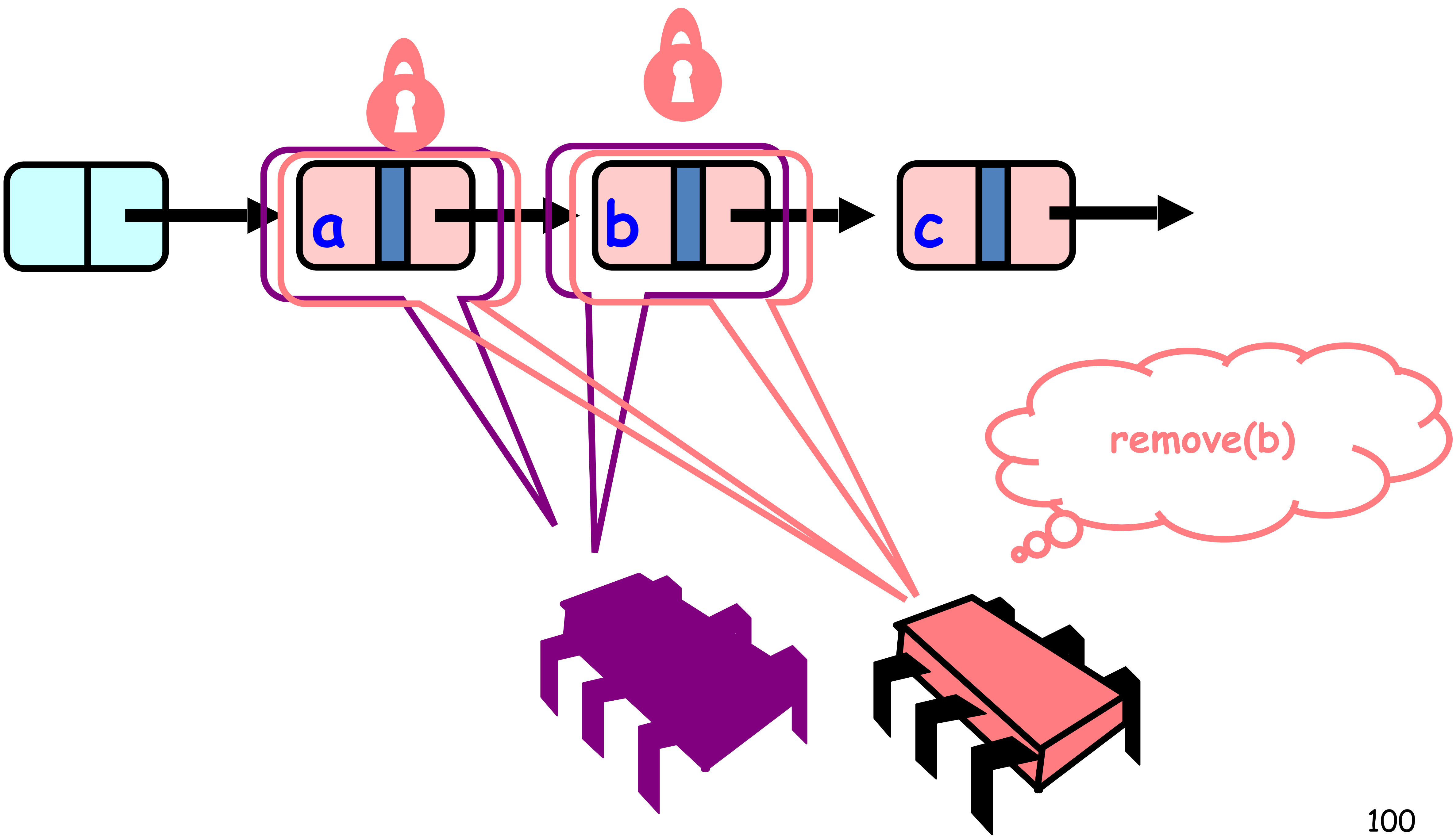
Lazy List



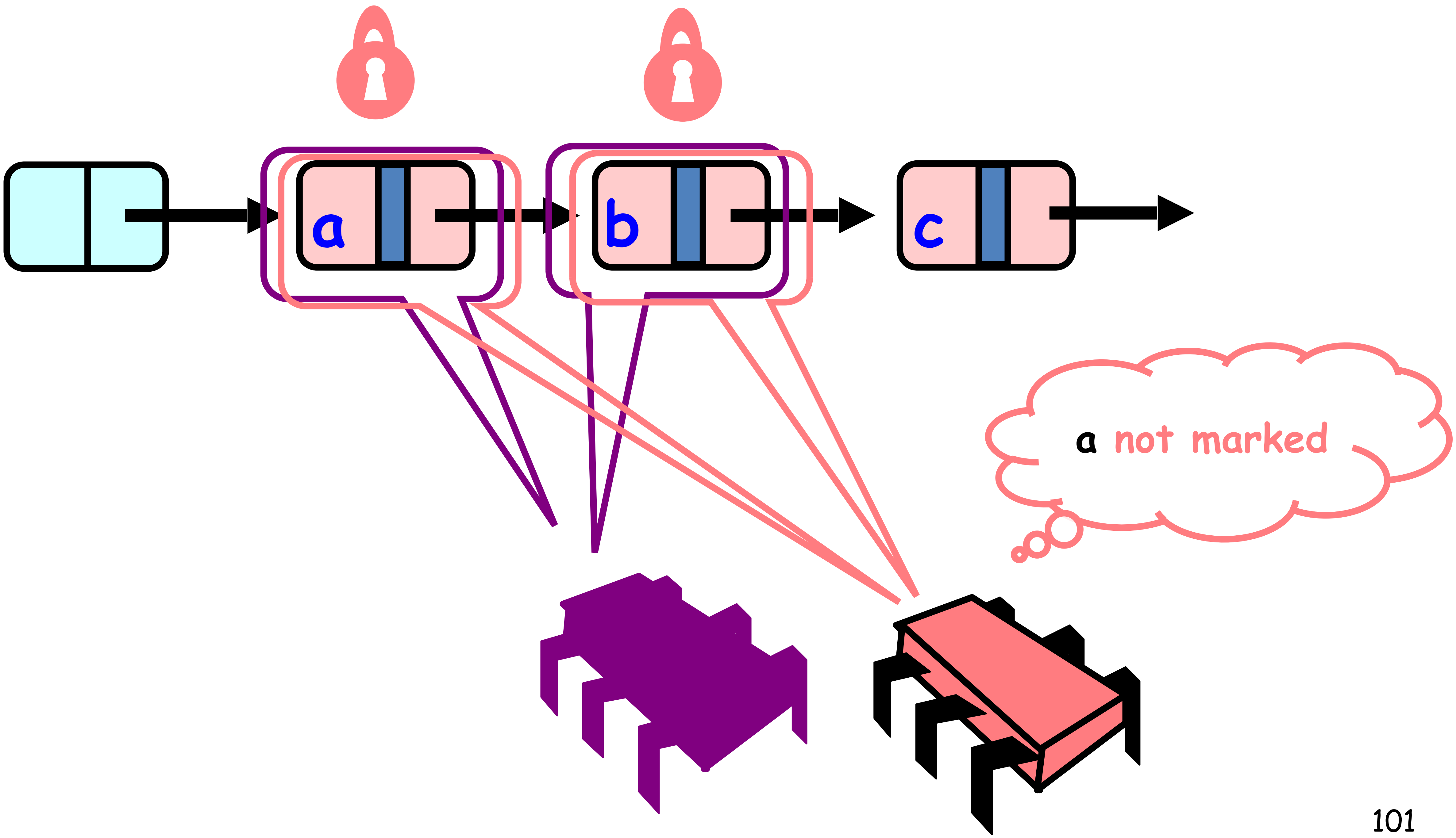
Lazy List



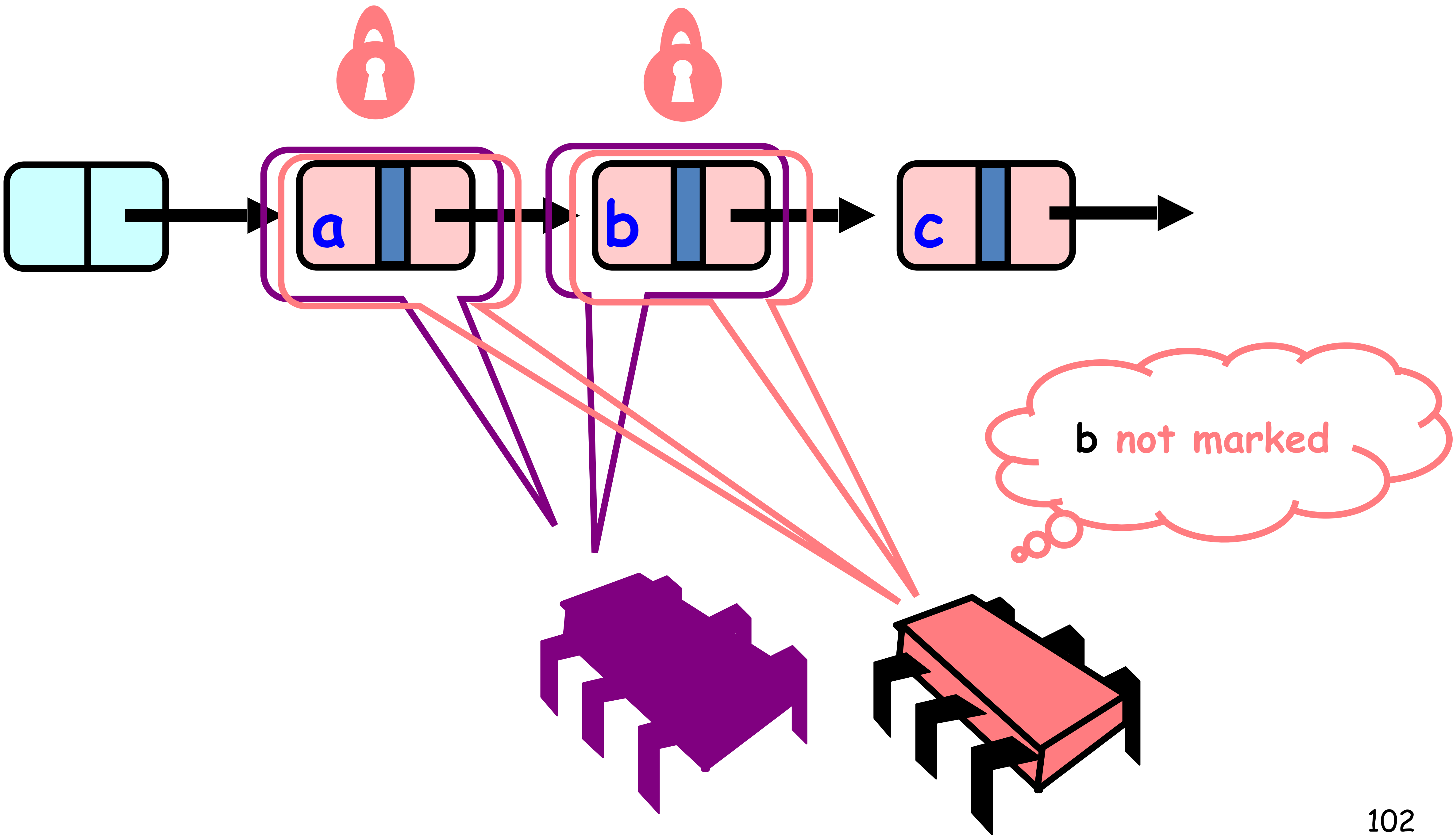
Lazy List



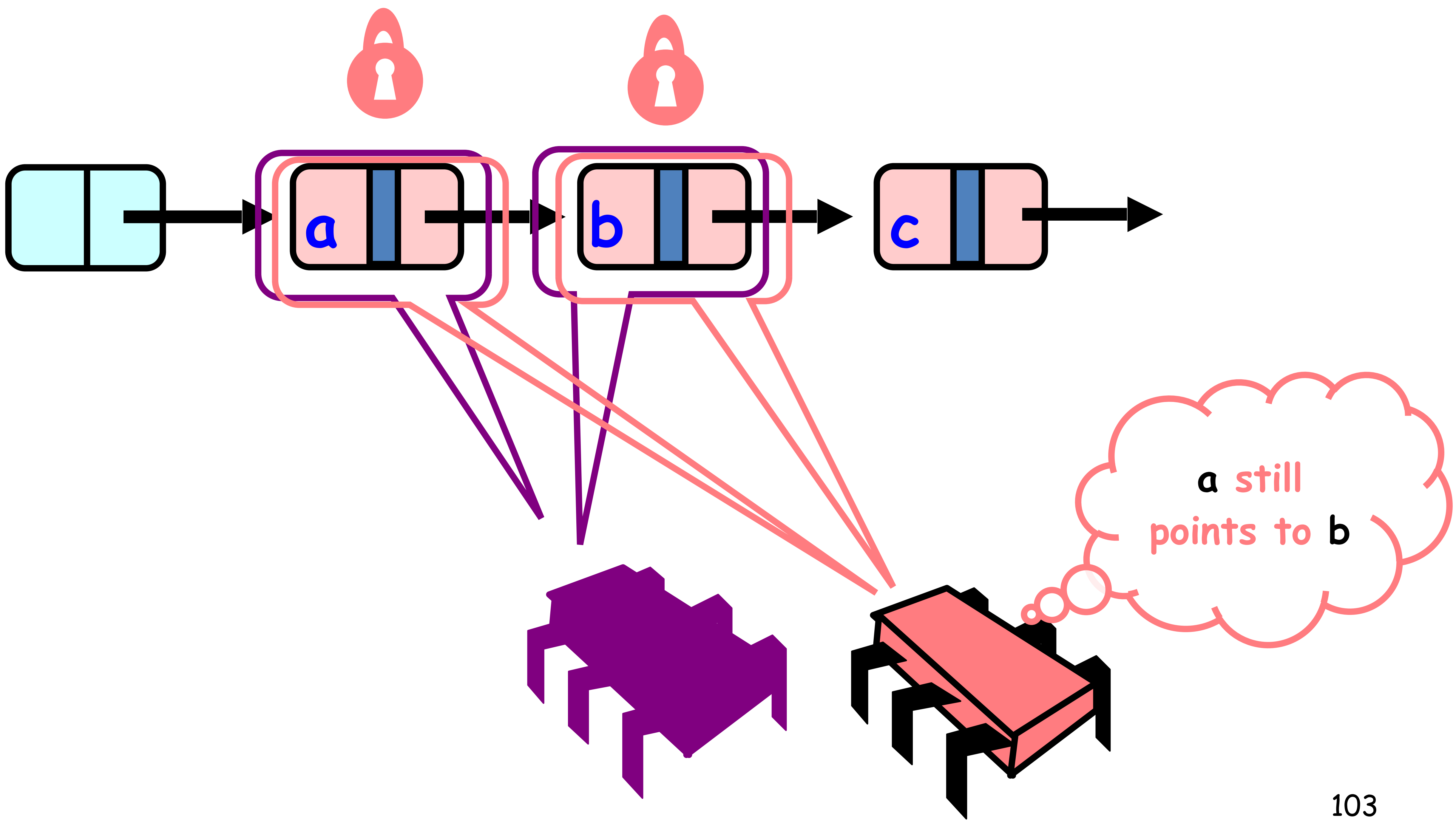
Lazy List



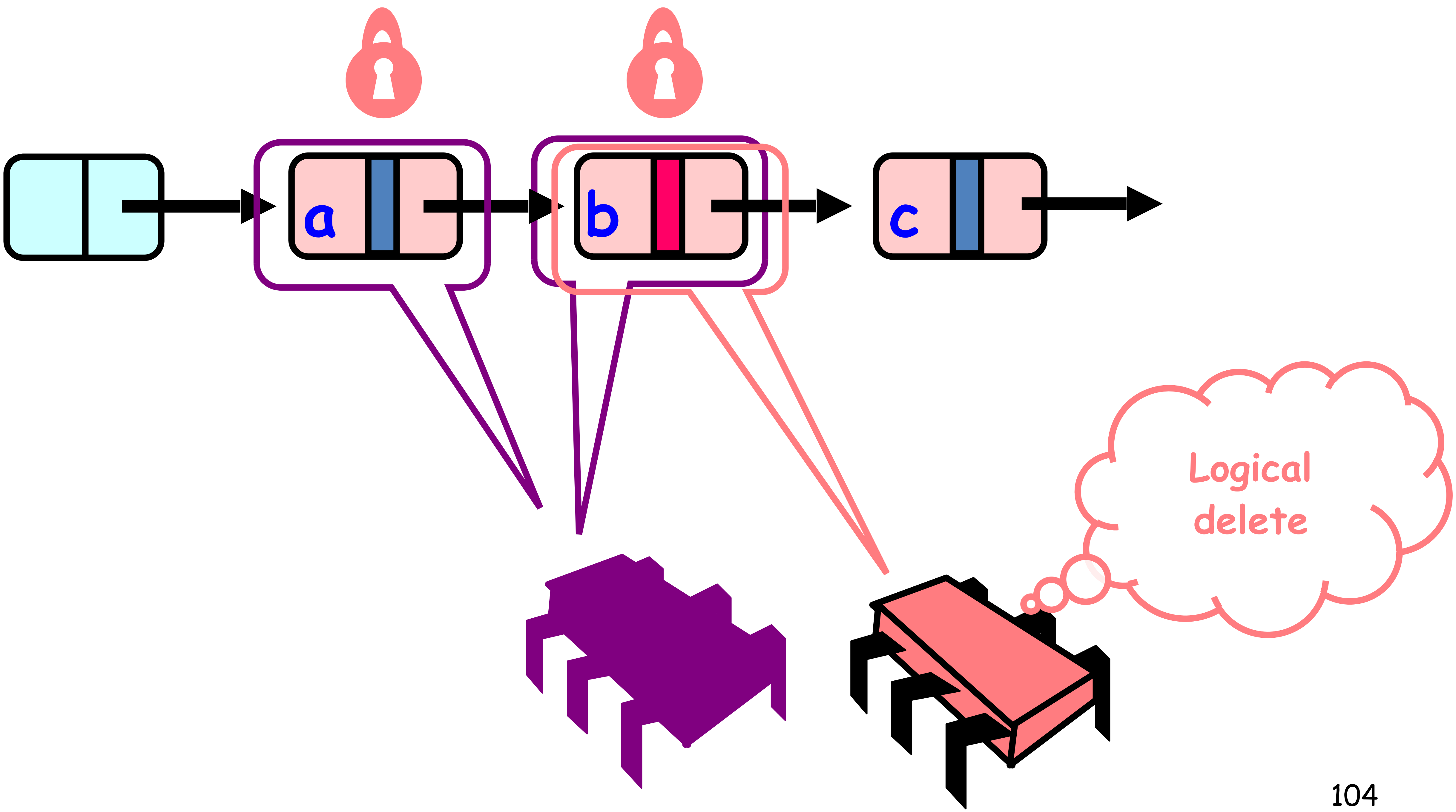
Lazy List



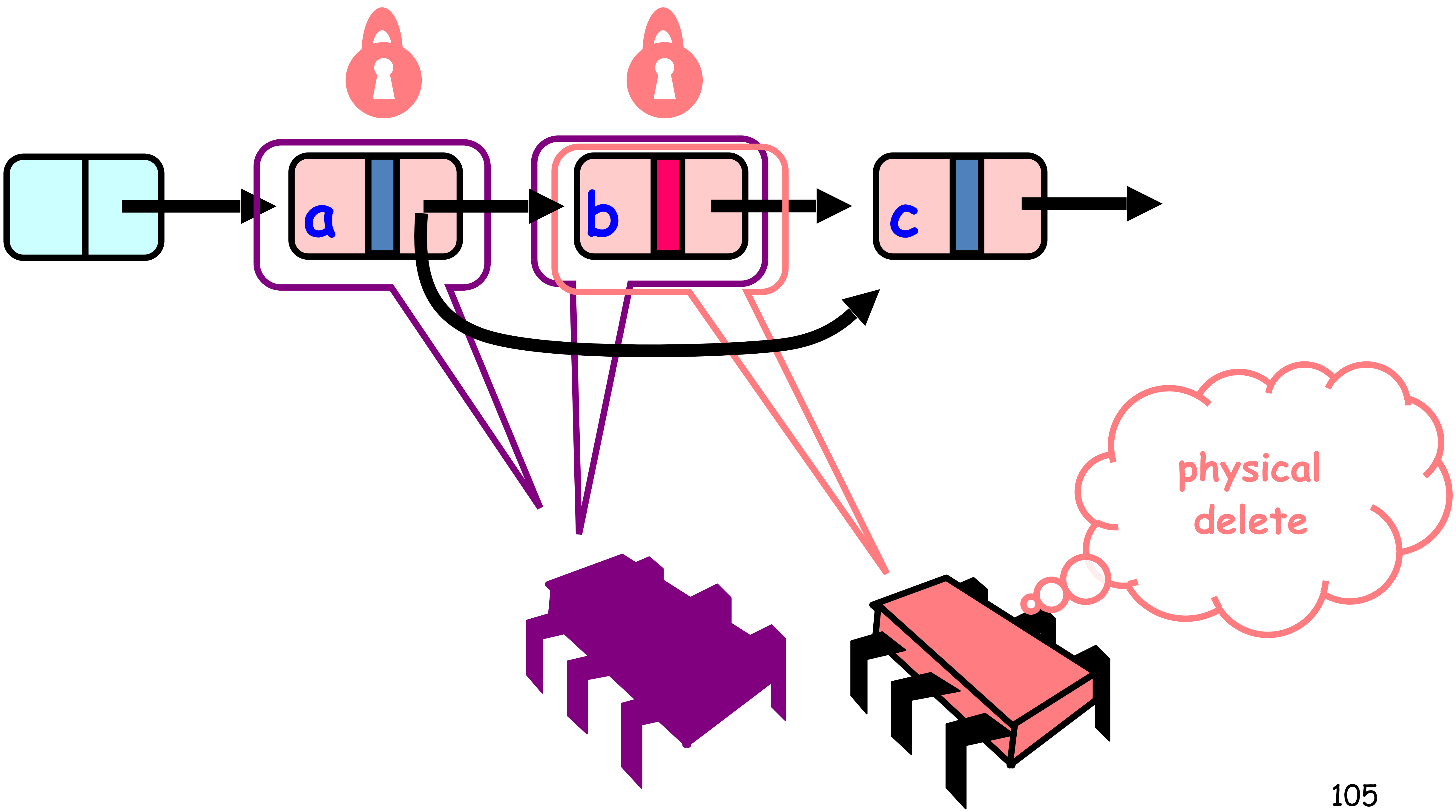
Lazy List



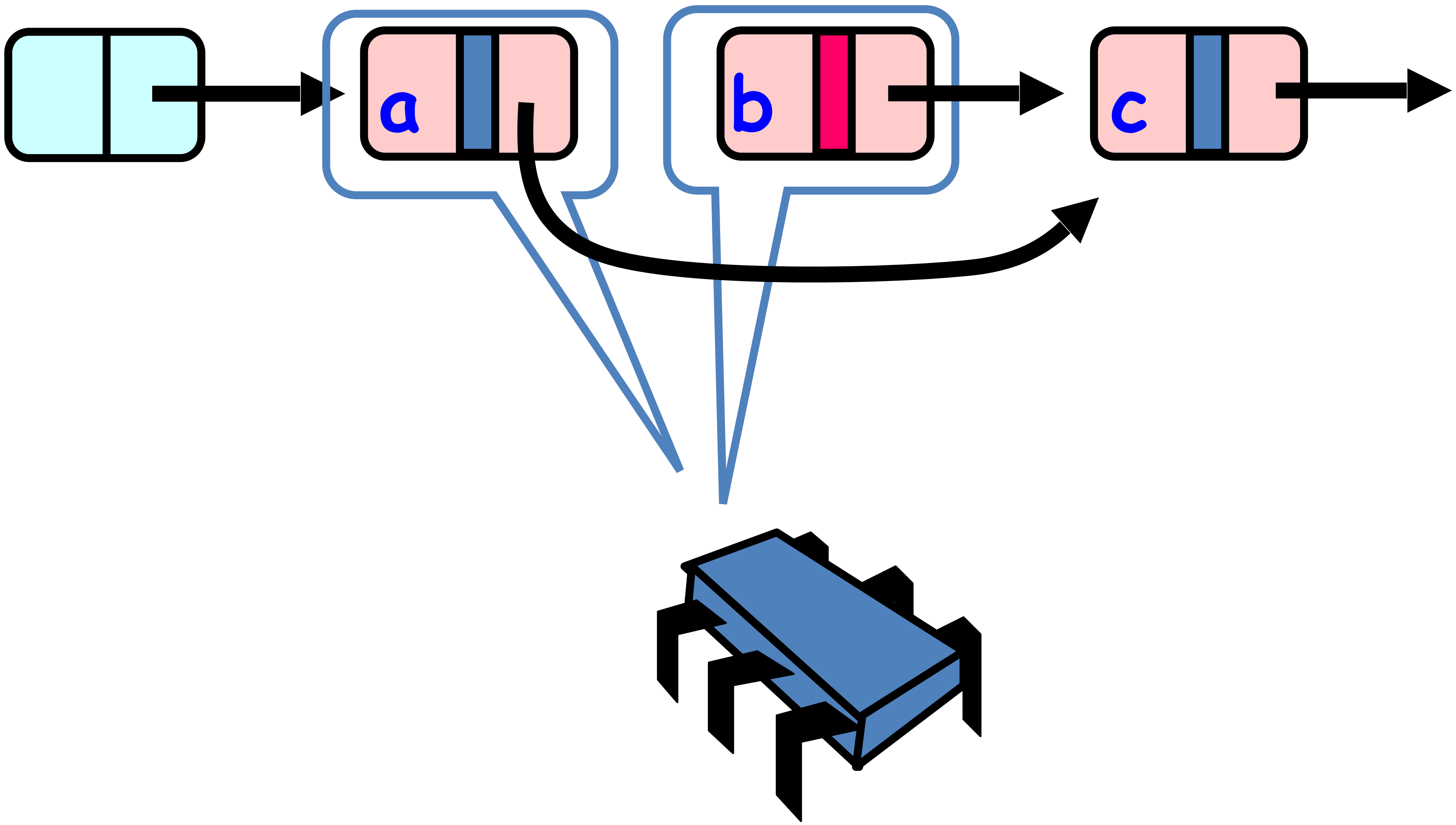
Lazy List



Lazy List



Lazy List



Invariant

- If a node is not marked then its key is in the set
  - and reachable from head

The contains Method

- Simply traverses the list and reports finding.
- Very efficient, progress guaranteed.
  - Wait-free
- Most “popular” method.

Properties of Lazy List

- Good performance
  - no rescanning,
  - a small number of locks,
  - hopefully not too many validation failures
- Good scalability
  - Lock only relevant nodes.
- Still standard locking problems
  - No progress guarantee
  - A thread holding a lock may face a cache-miss, page-fault, swap-out, etc.
  - Worst-case scalability issues, scheduler critical here…

Summary

- Starting data structures
- Locking and Lock-freedom
- Linked list (ordered for sets)
- Parallization problems
- Fine grained locking:
  - Hand-over-hand
  - Optimistic
  - Lazy
- Lock-Free version

Which List Should You Use?

- If little contention: coarse-grained locking.
- To handle contention pretty well: lazy list.
- To handle high contention and provide a progress guarantee: lock-free list.

The End