Linked Lists: Locking, Lock-Free, and Beyond …

Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

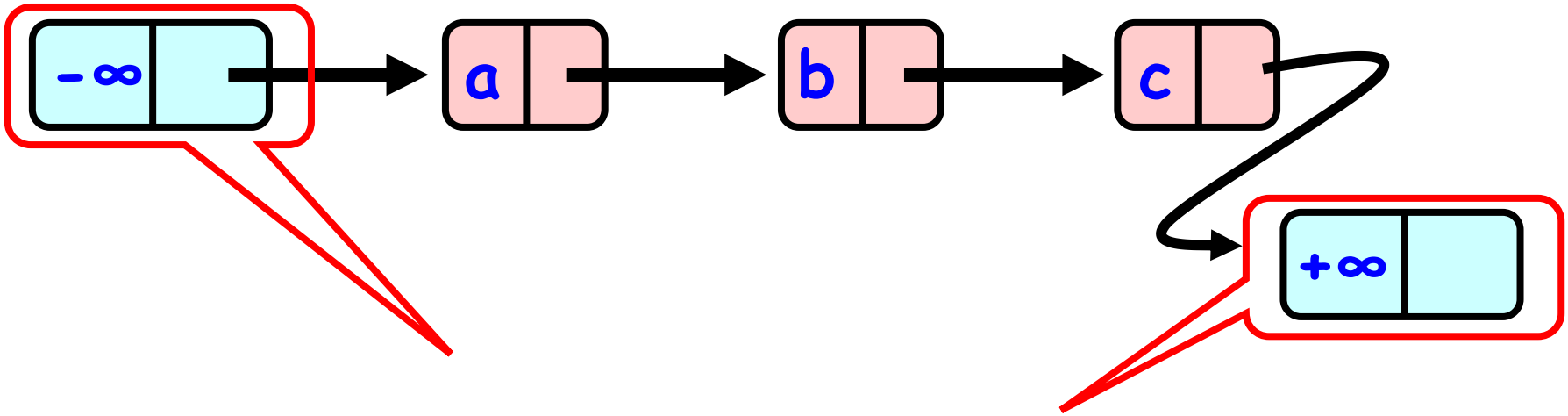
Set Interface

- 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 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 by
  - 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

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

Rep Invariant

- Which concrete values meaningful?
  - Sorted?
  - Duplicates?
- Rep invariant
  - Characterizes legal concrete reps
  - 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 one is incorrect?

Blame Game

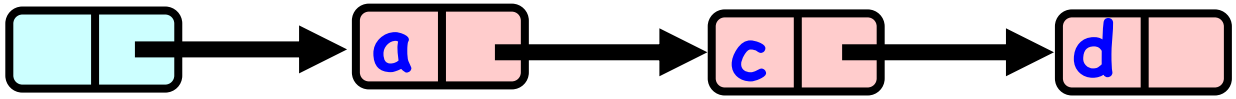
- Suppose
  - **add()** leaves behind 2 copies of x
  - **remove()** removes only 1
- Which one is incorrect?
  - If rep invariant says *no duplicates*
    - **add()** is incorrect
  - Otherwise
    - **remove()** is incorrect

Rep Invariant (partly)

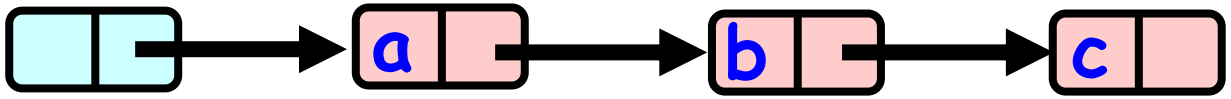
- Sentinel nodes
  - tail reachable from head
- Sorted
- No duplicates

Sequential List Based Set

Add()

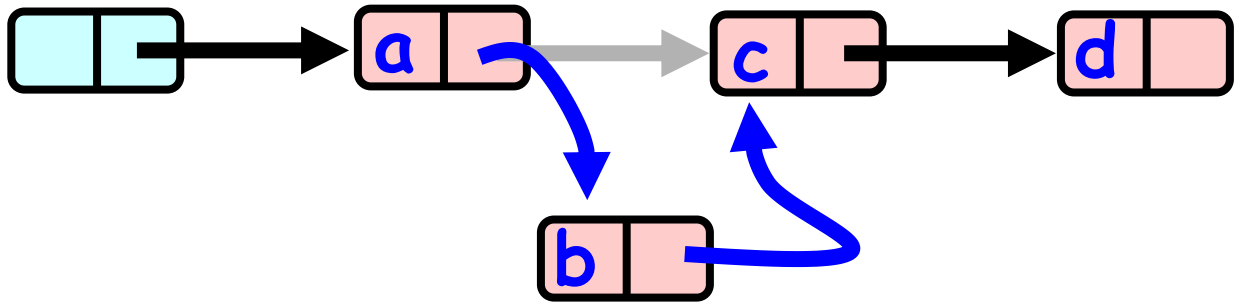


Remove()

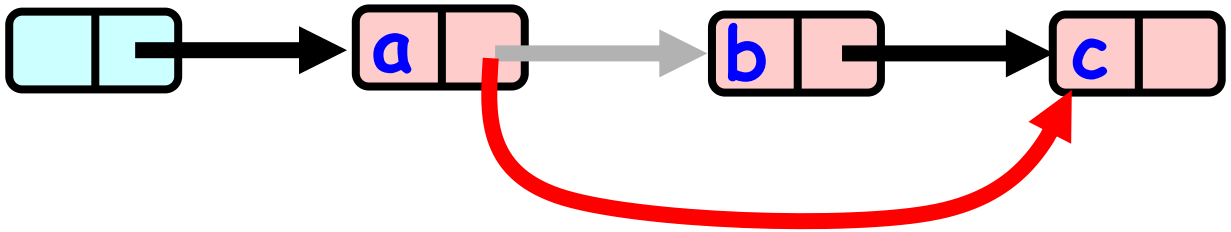


Sequential List Based Set

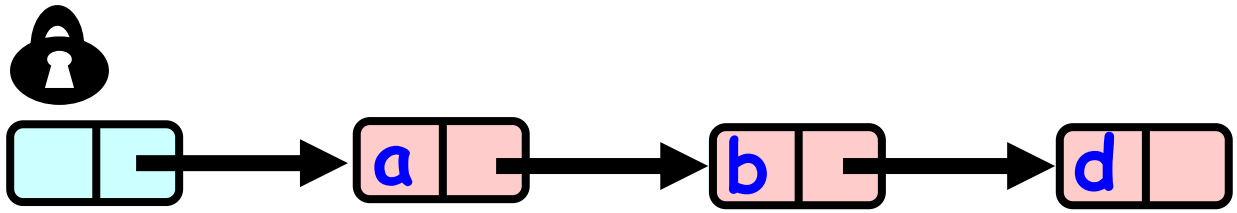
Add()



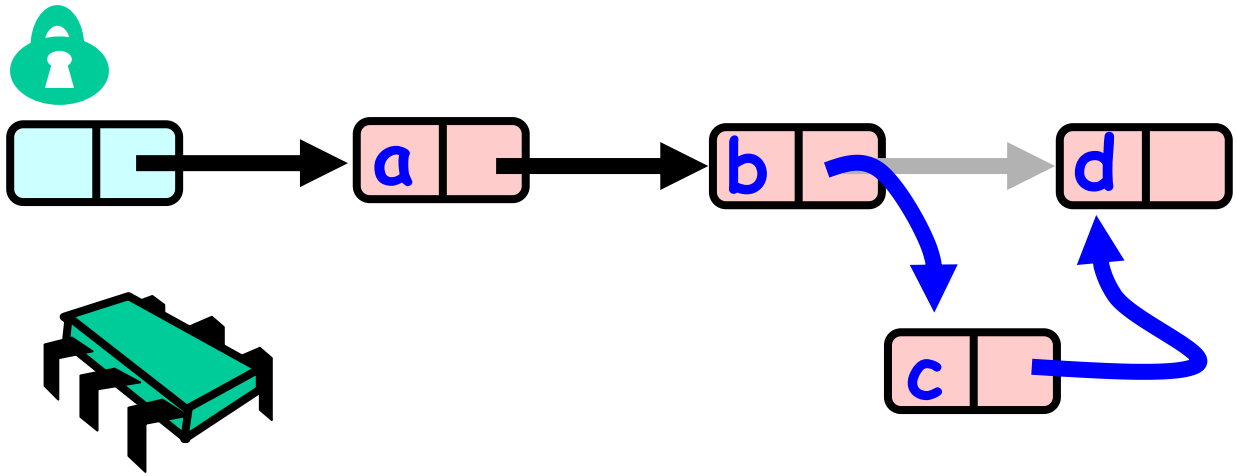
Remove()



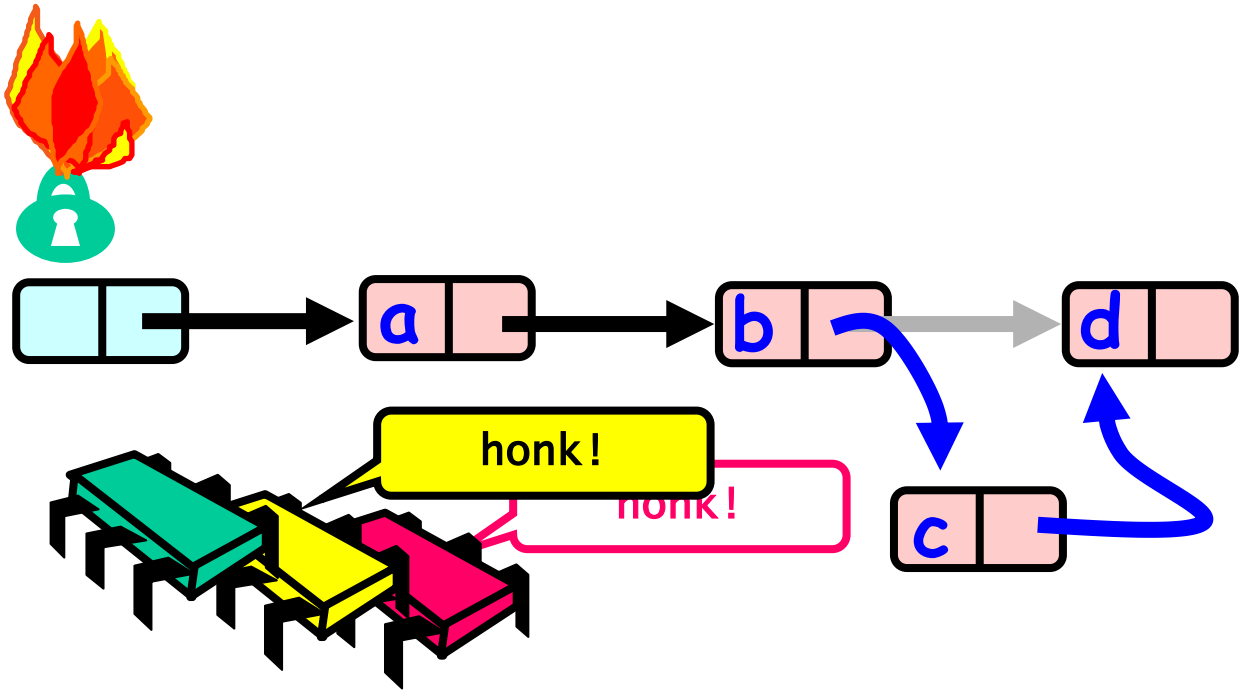
Coarse Grained Locking



Coarse Grained Locking



Coarse Grained Locking



Simple but hotspot + bottleneck

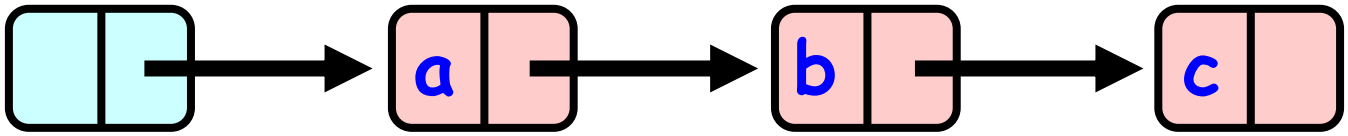
Coarse-Grained Locking

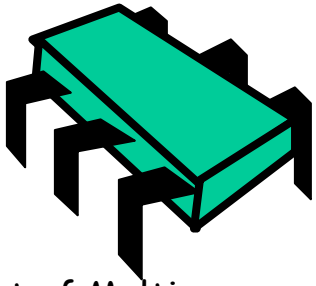
- Easy, same as synchronized methods
  - "One lock to rule them all …"
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention
  - Queue locks help
  - But bottleneck still an issue

Fine-grained Locking

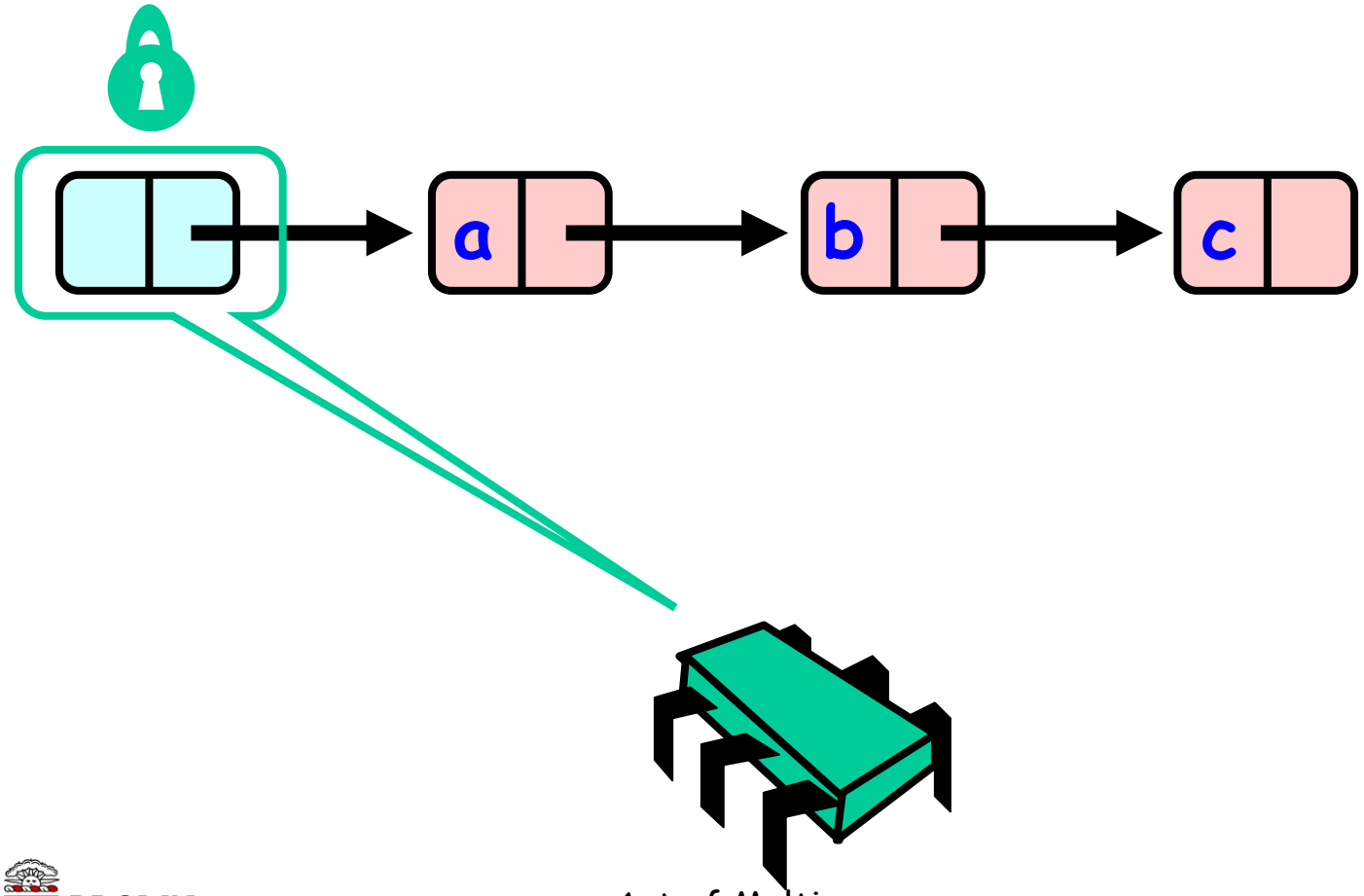
- Requires **careful** thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand locking

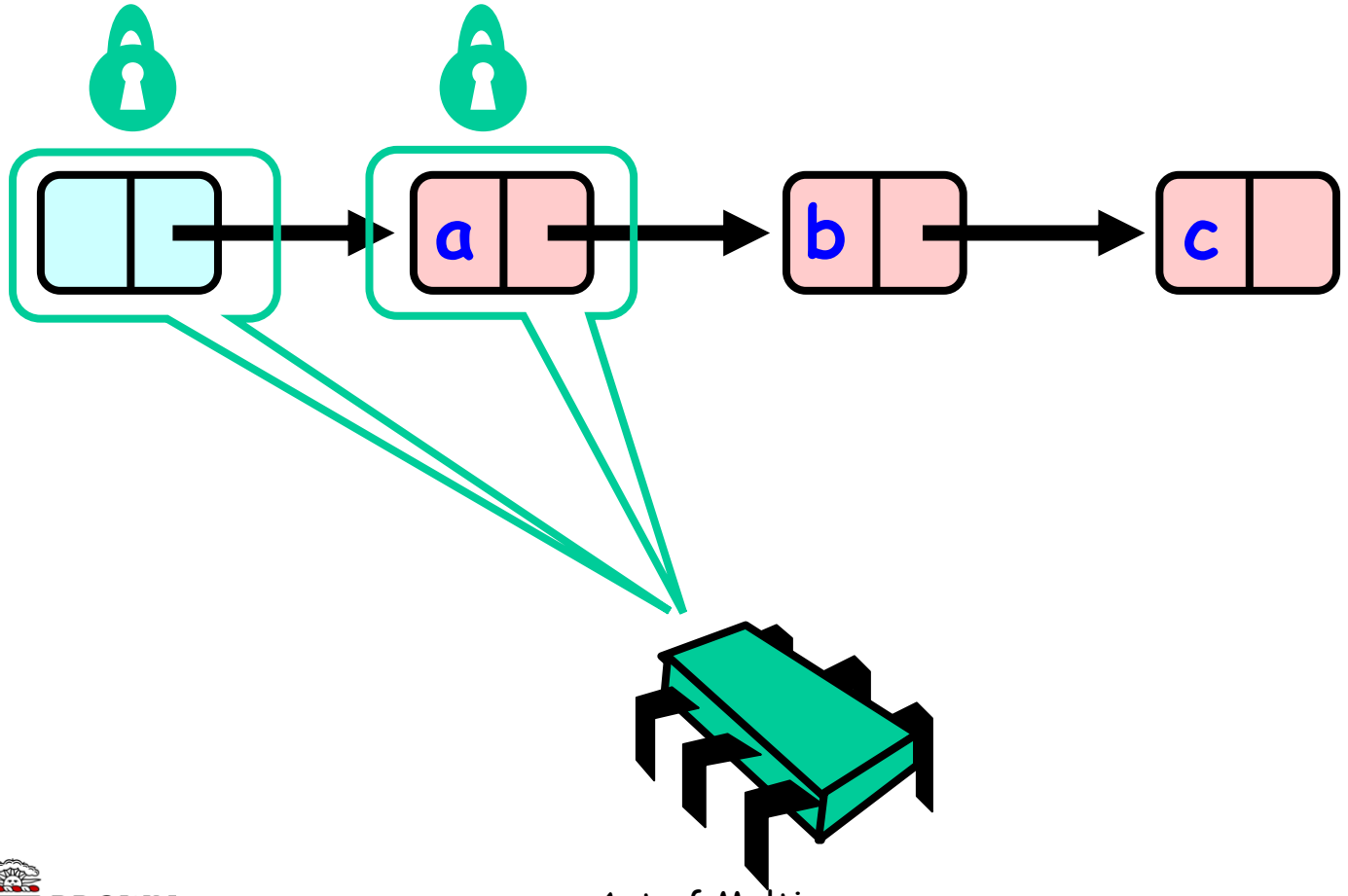




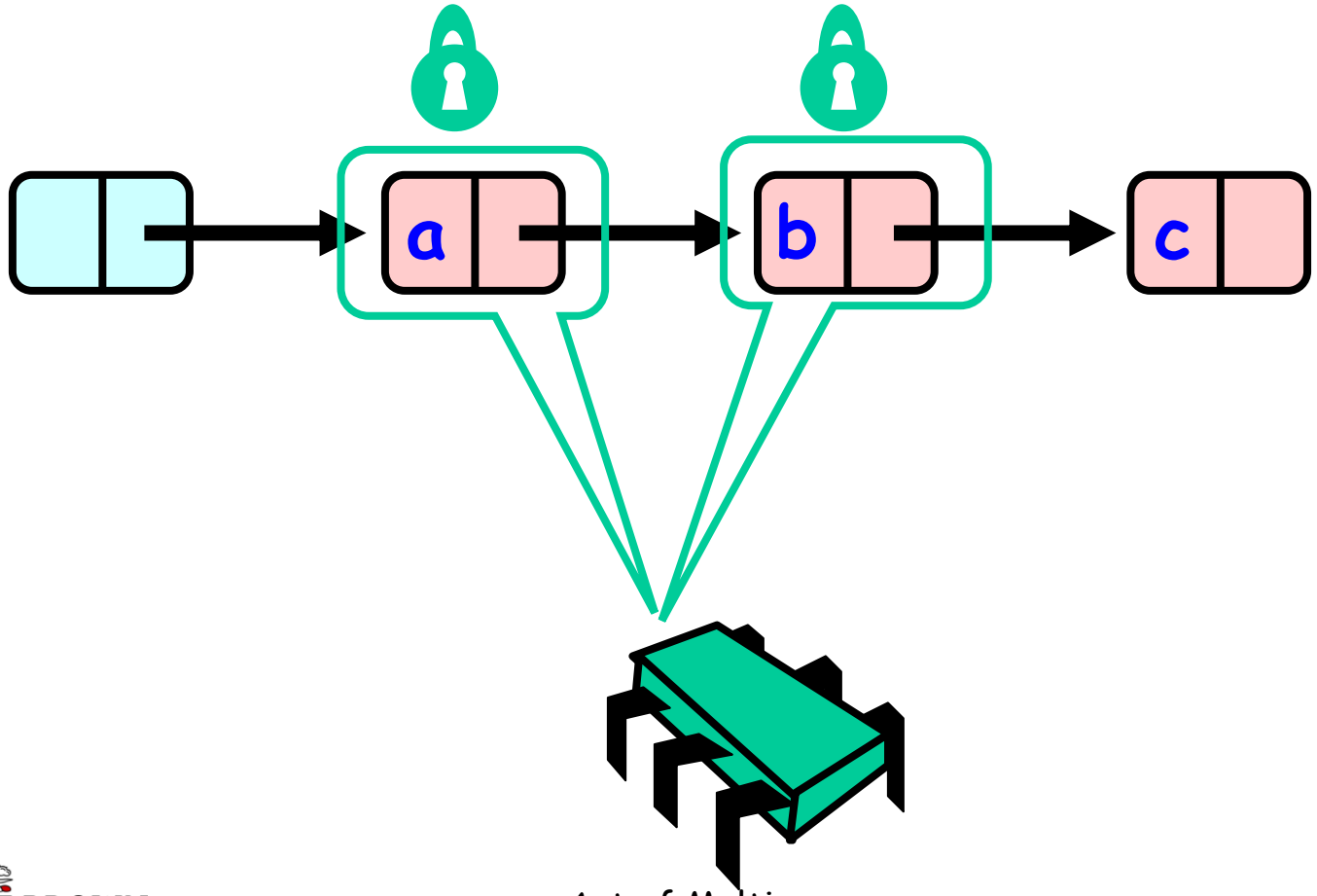
Hand-over-Hand locking



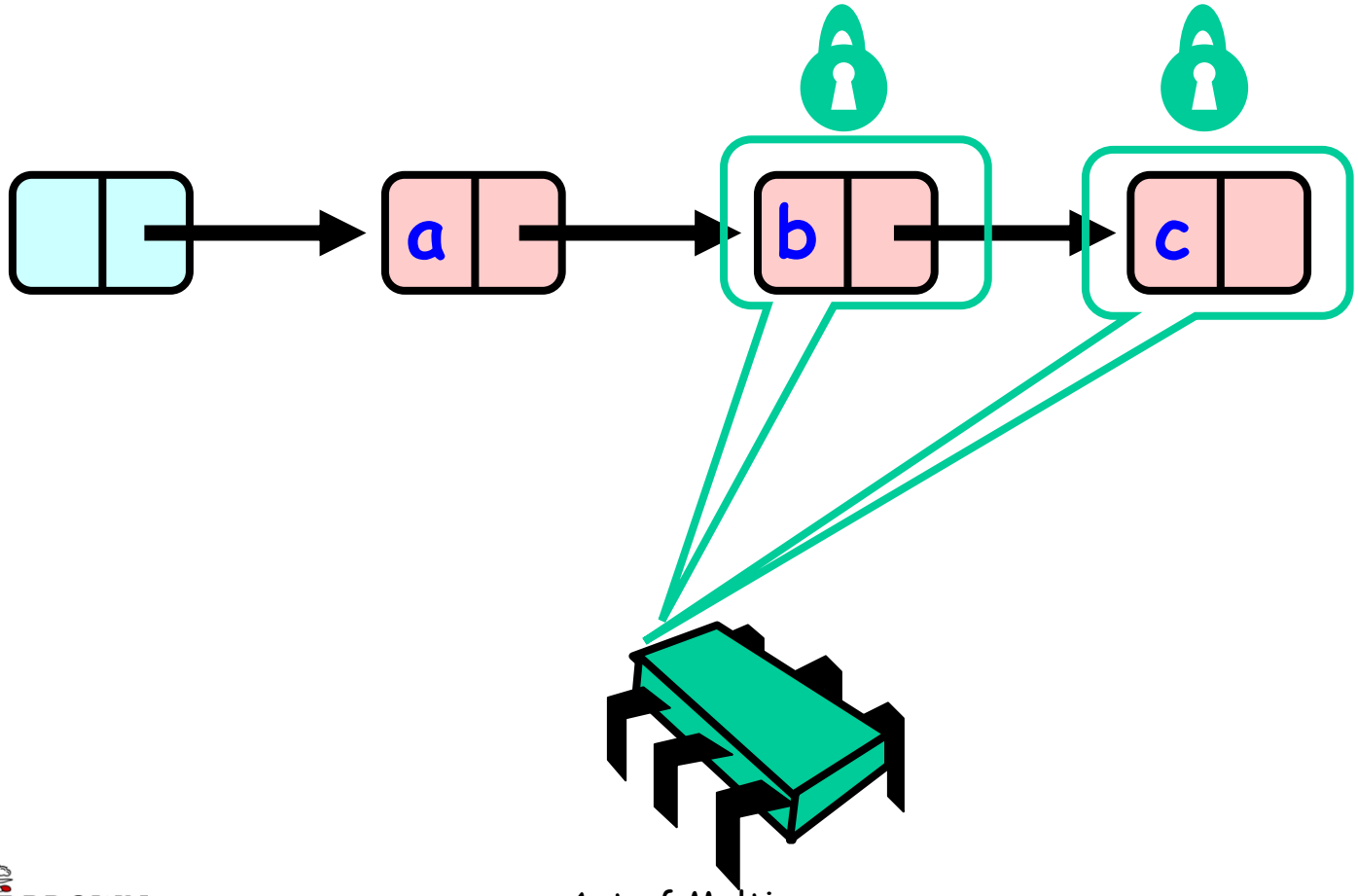
Hand-over-Hand locking



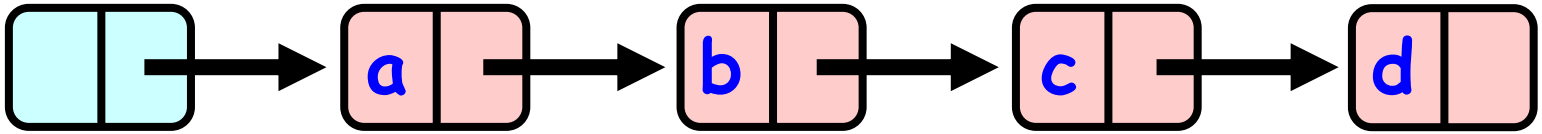
Hand-over-Hand locking

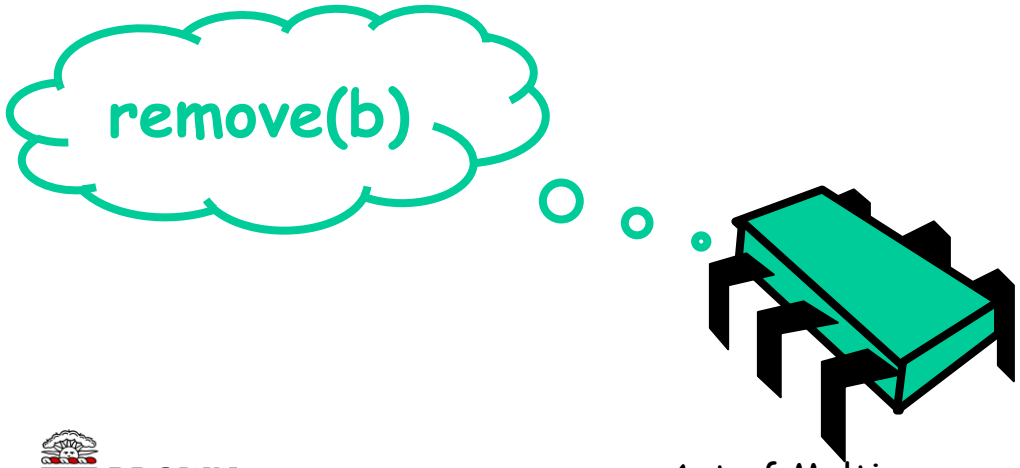


Hand-over-Hand locking

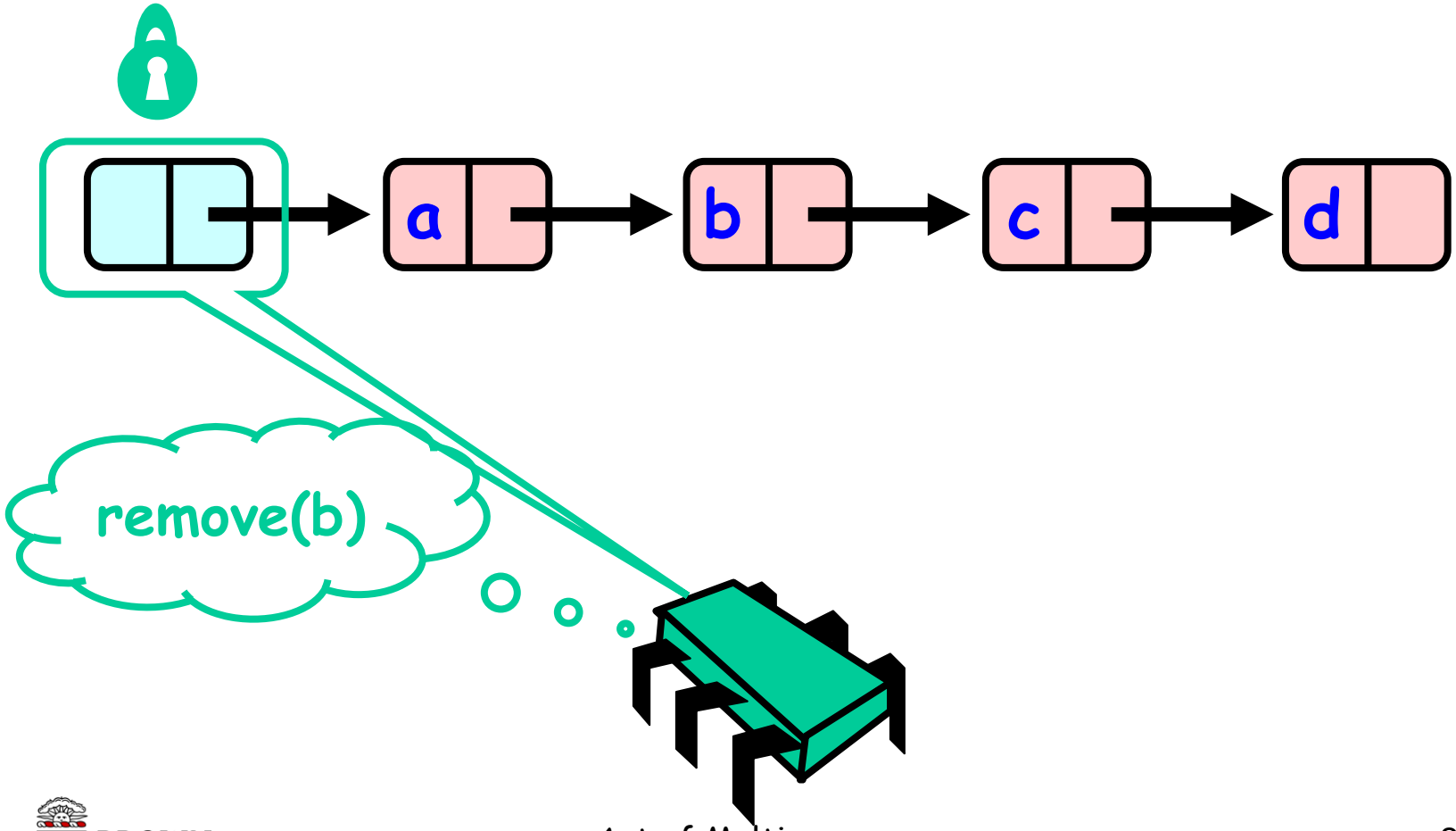


Removing a Node

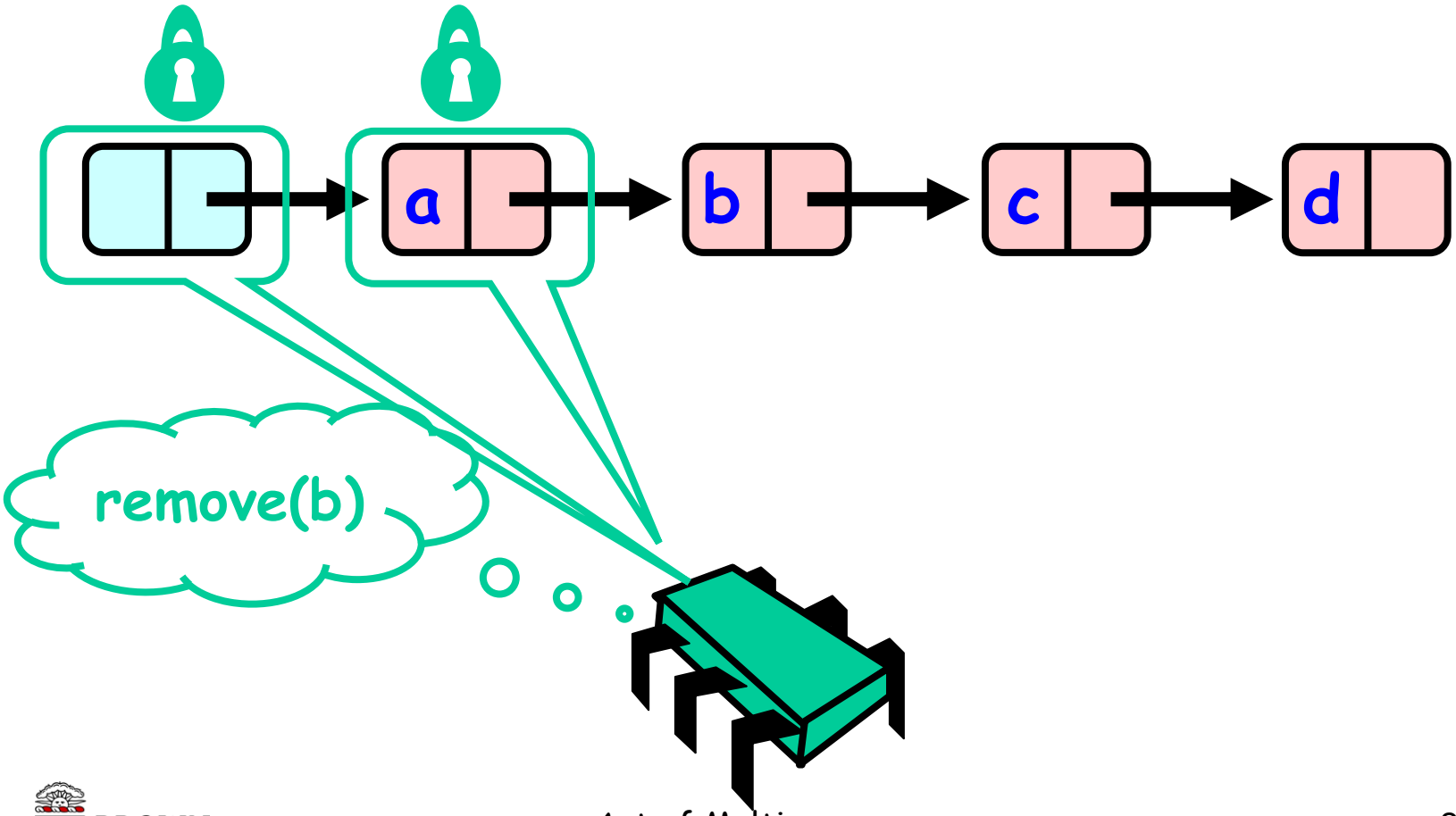




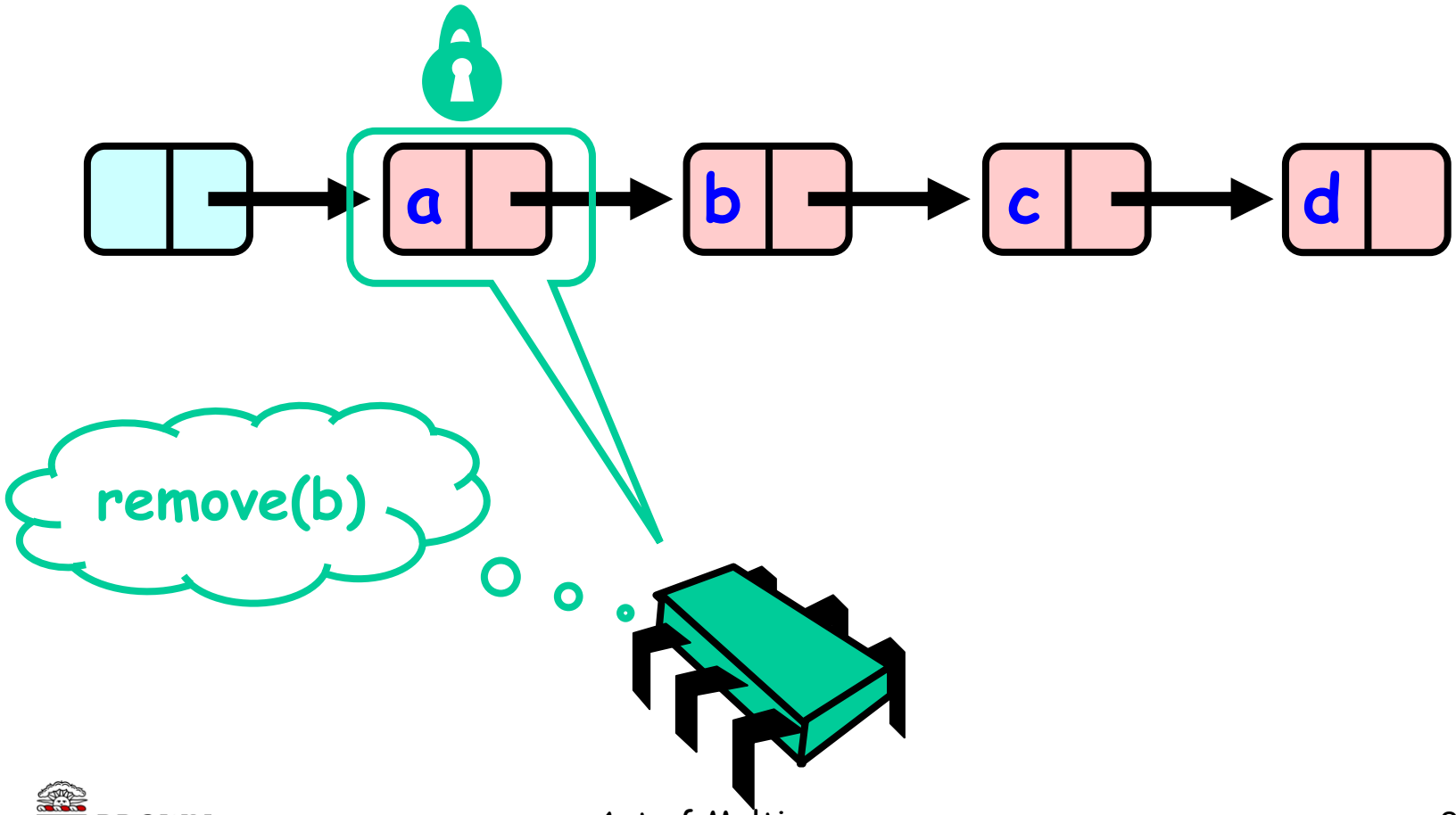
Removing a Node



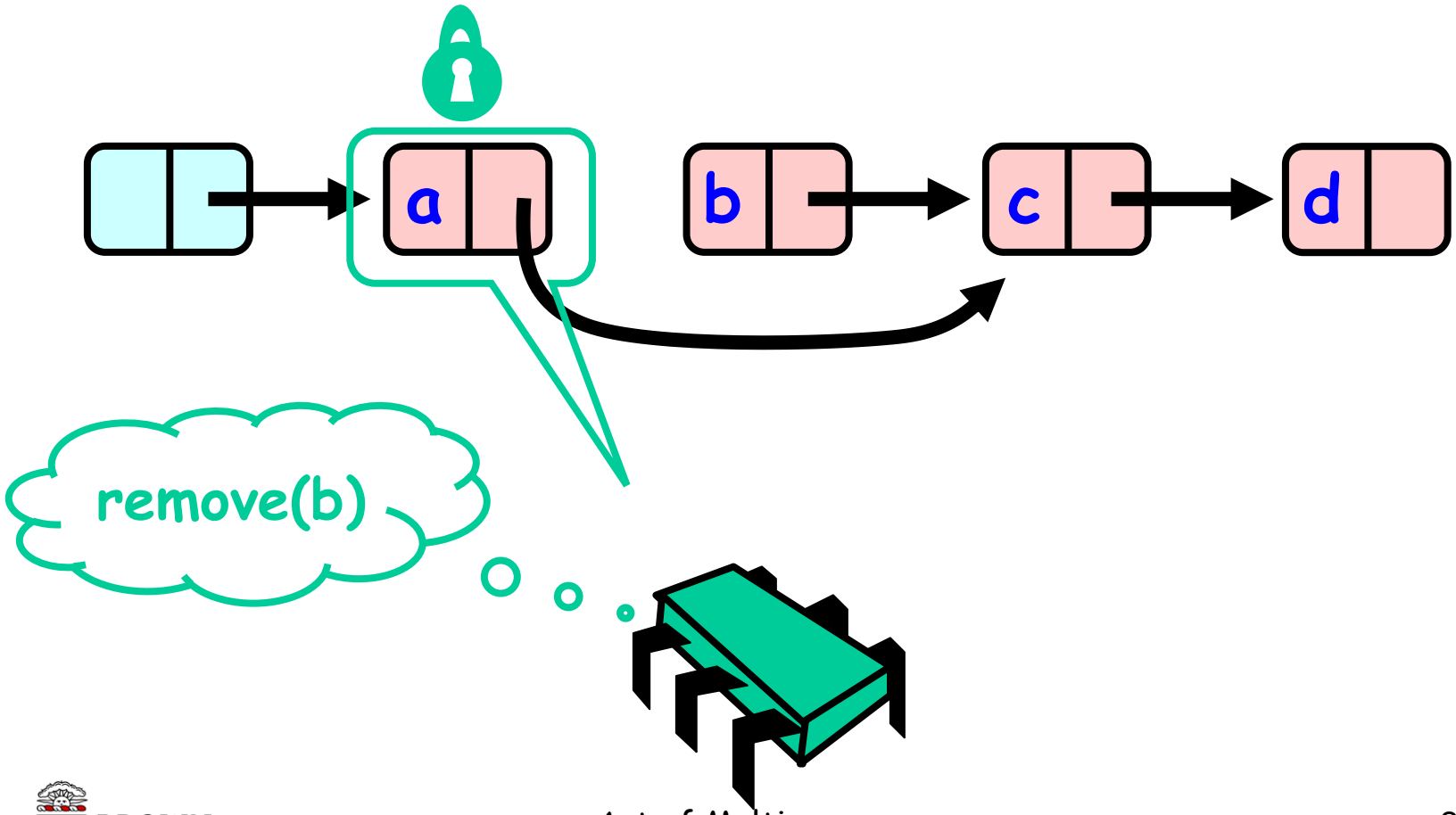
Removing a Node



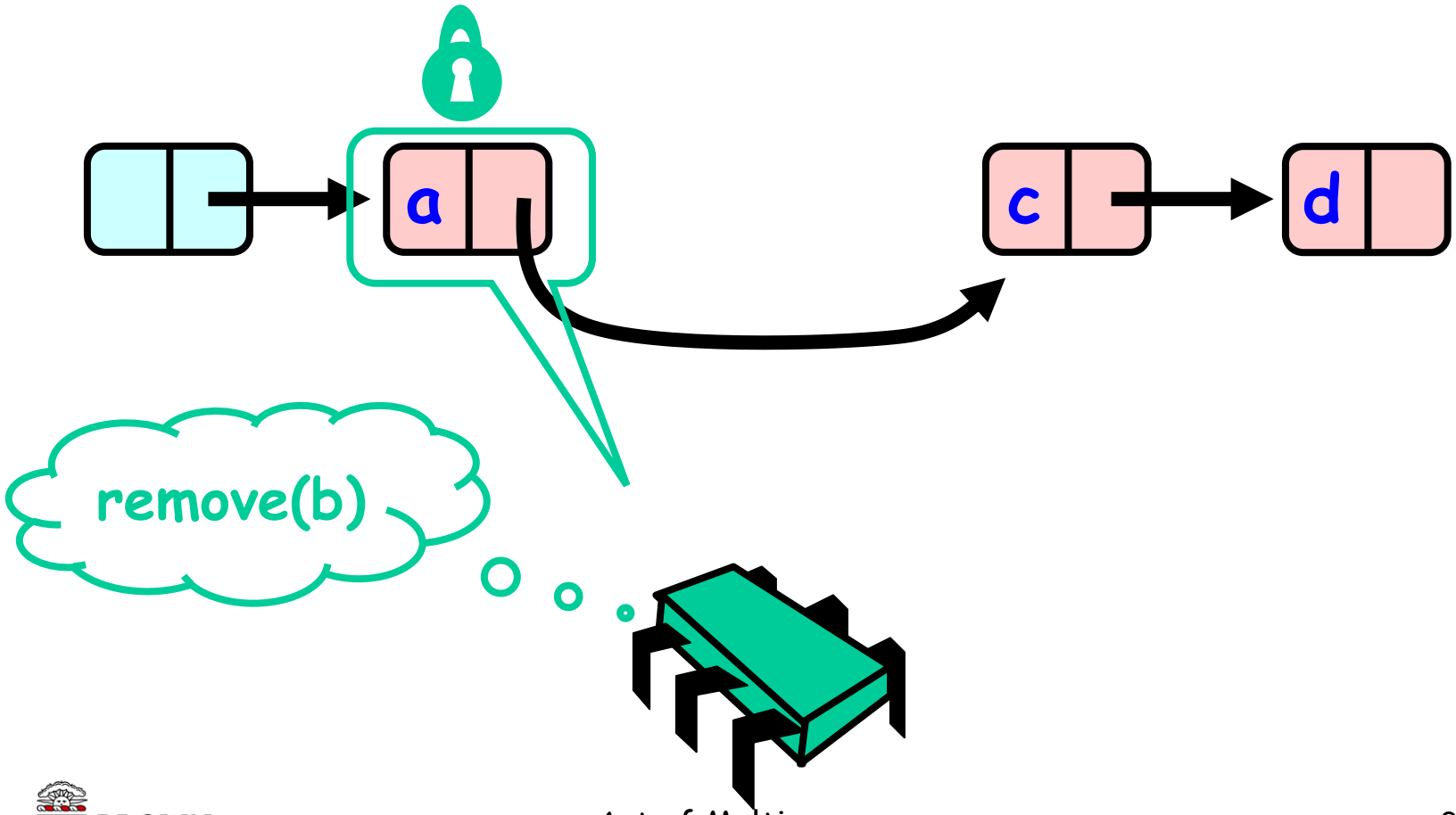
Removing a Node



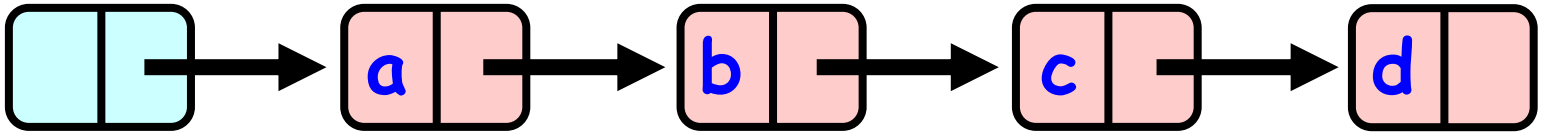
Removing a Node

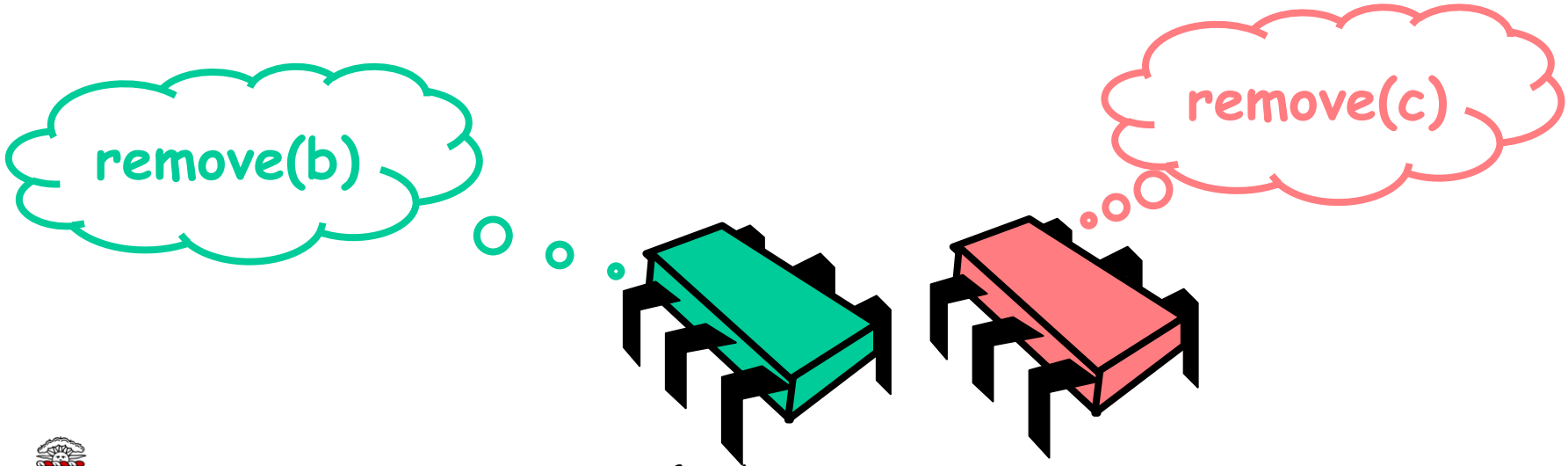


Removing a Node

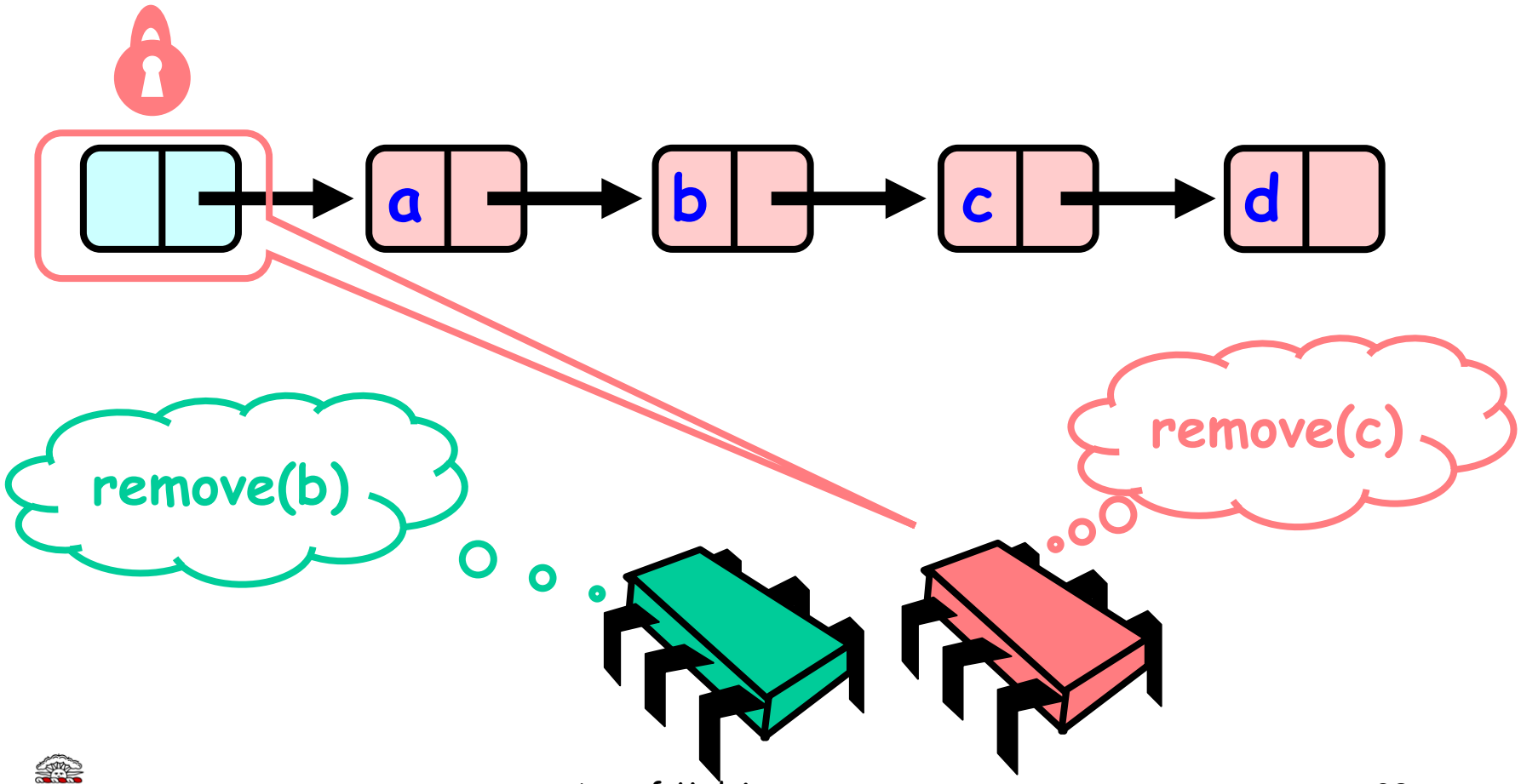


Removing a Node

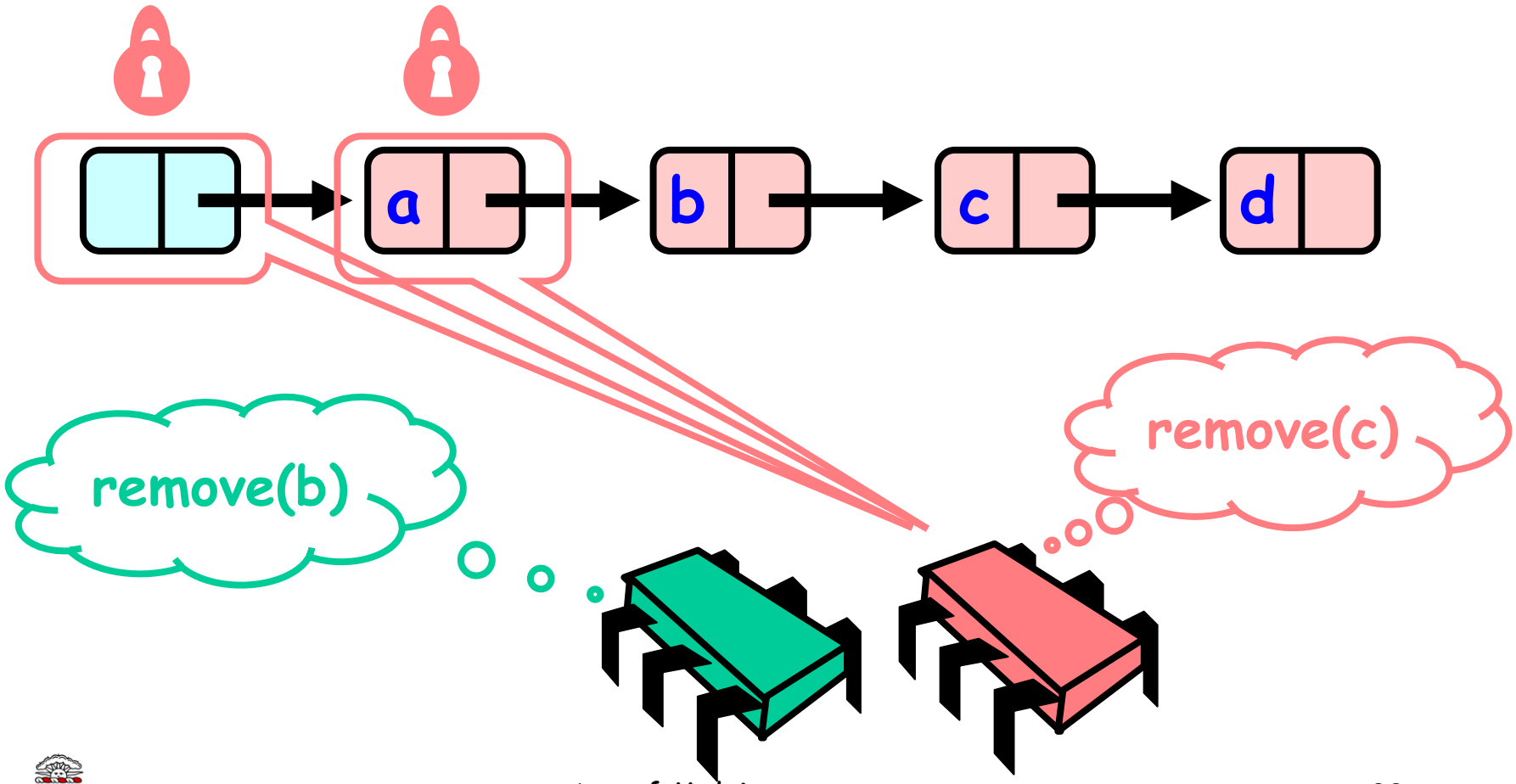




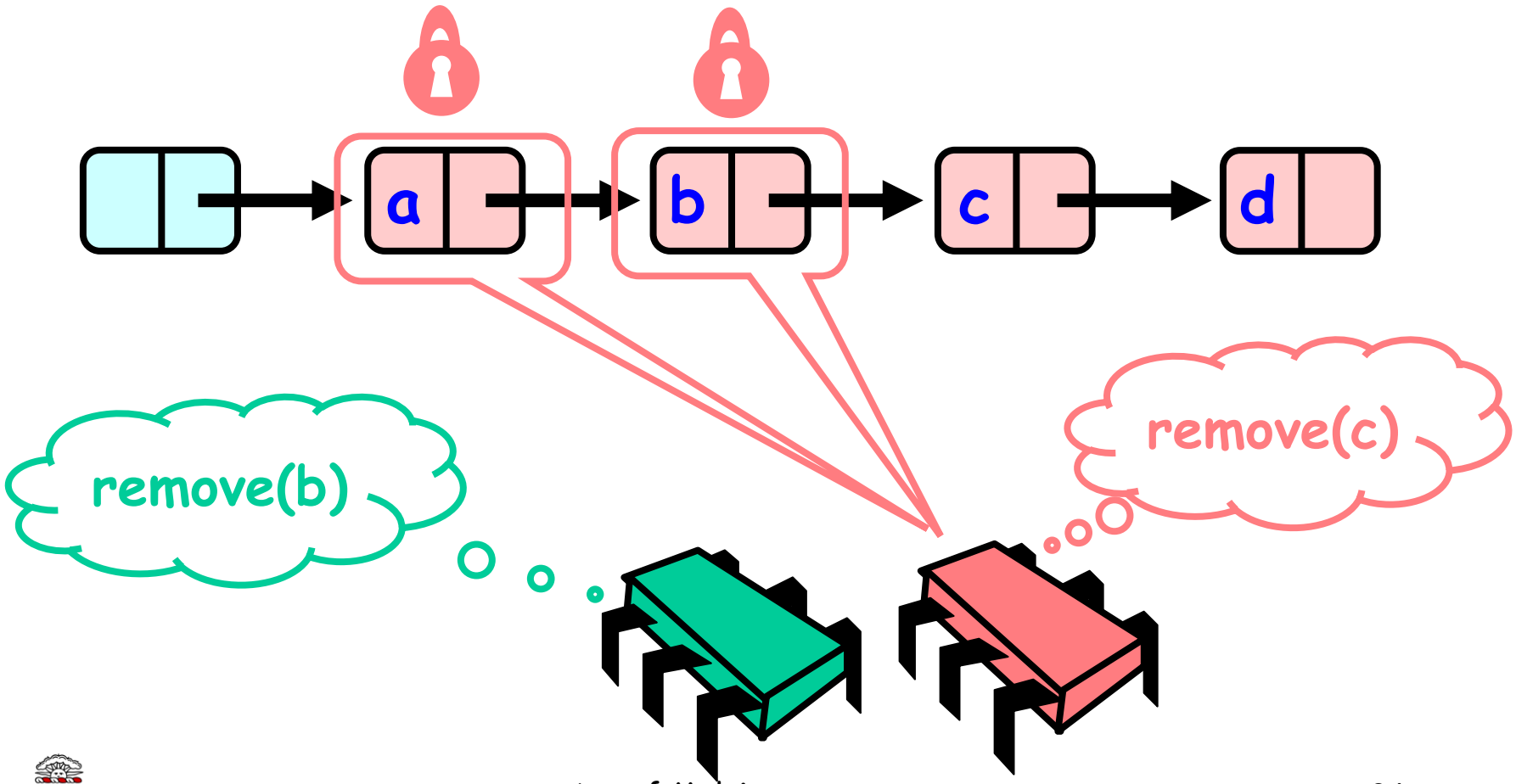
Removing a Node



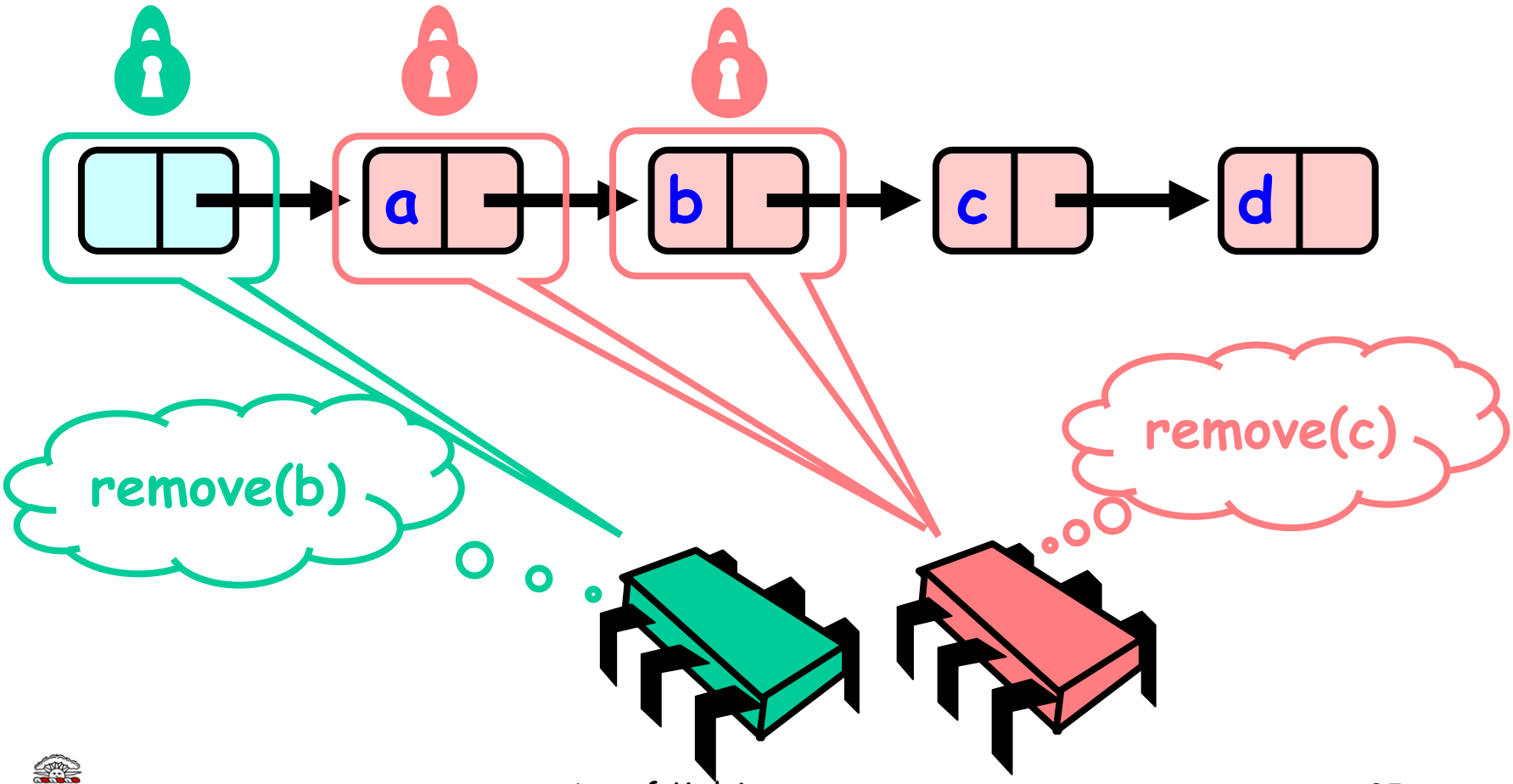
Removing a Node



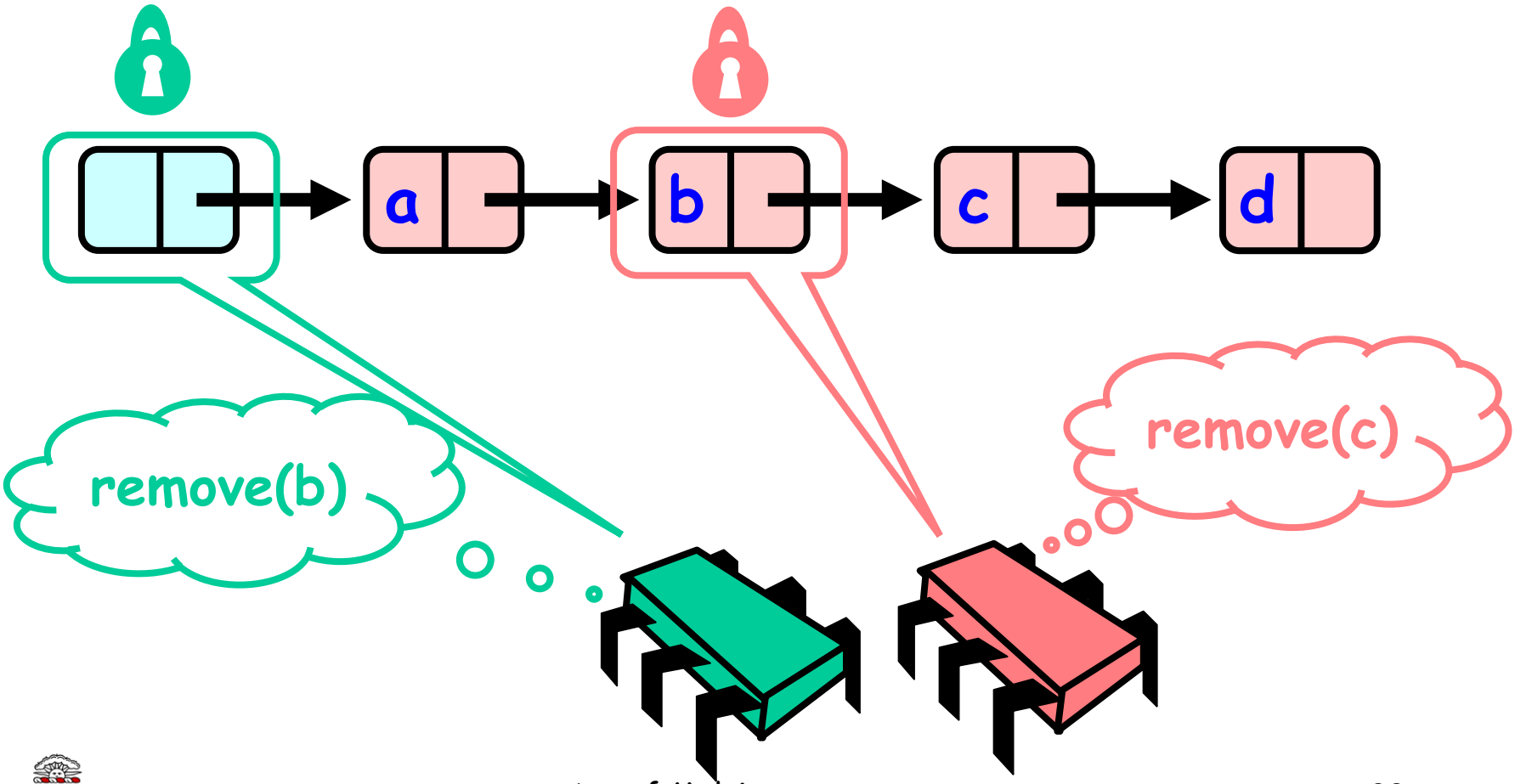
Removing a Node



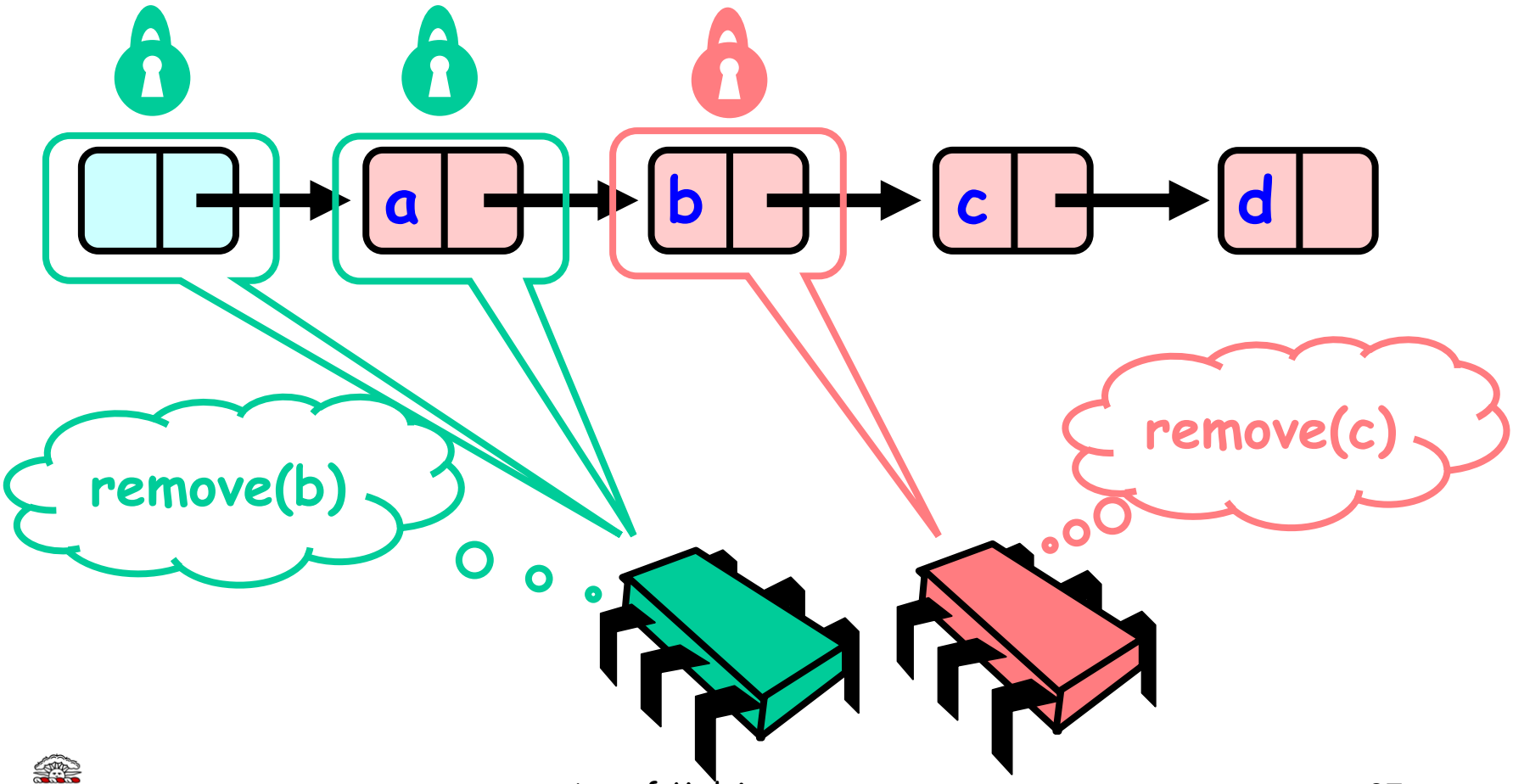
Removing a Node



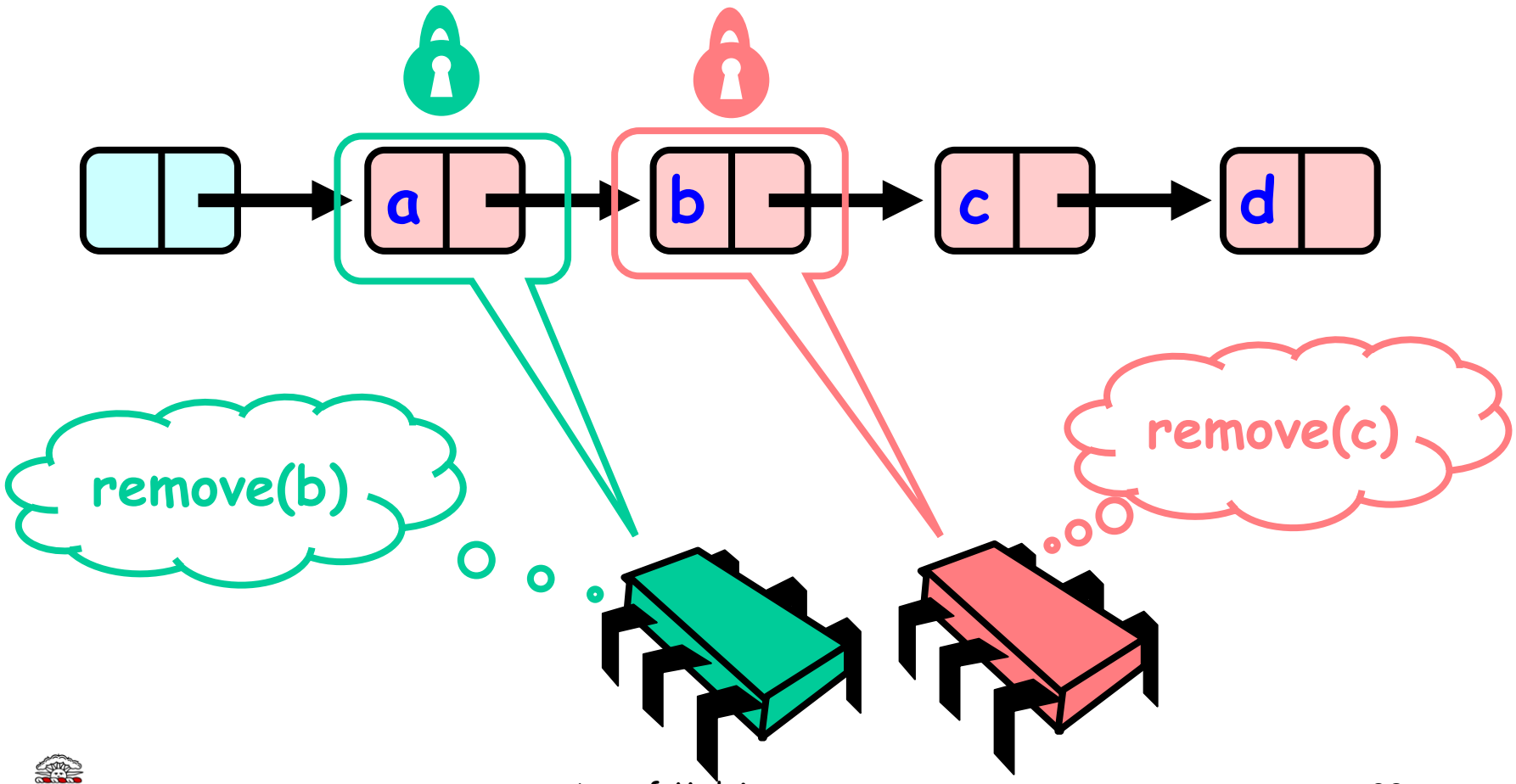
Removing a Node



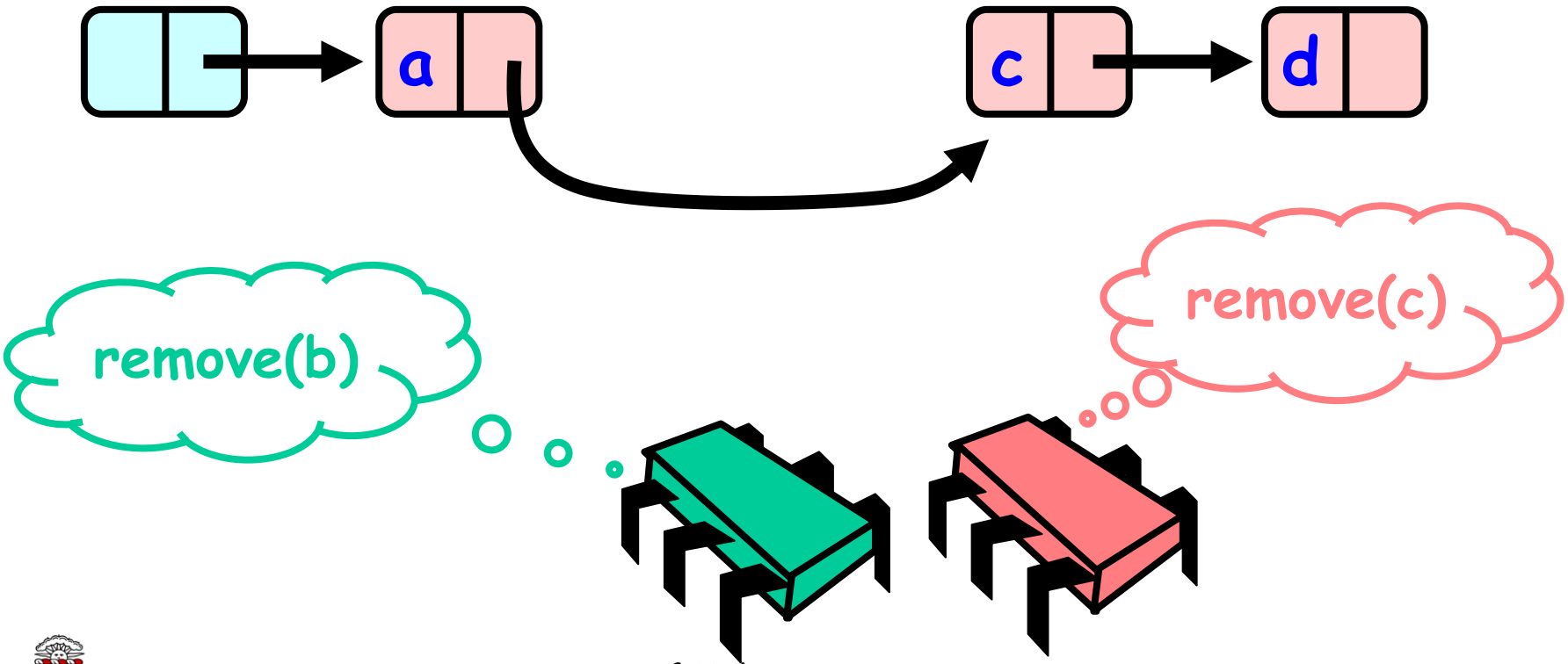
Removing a Node



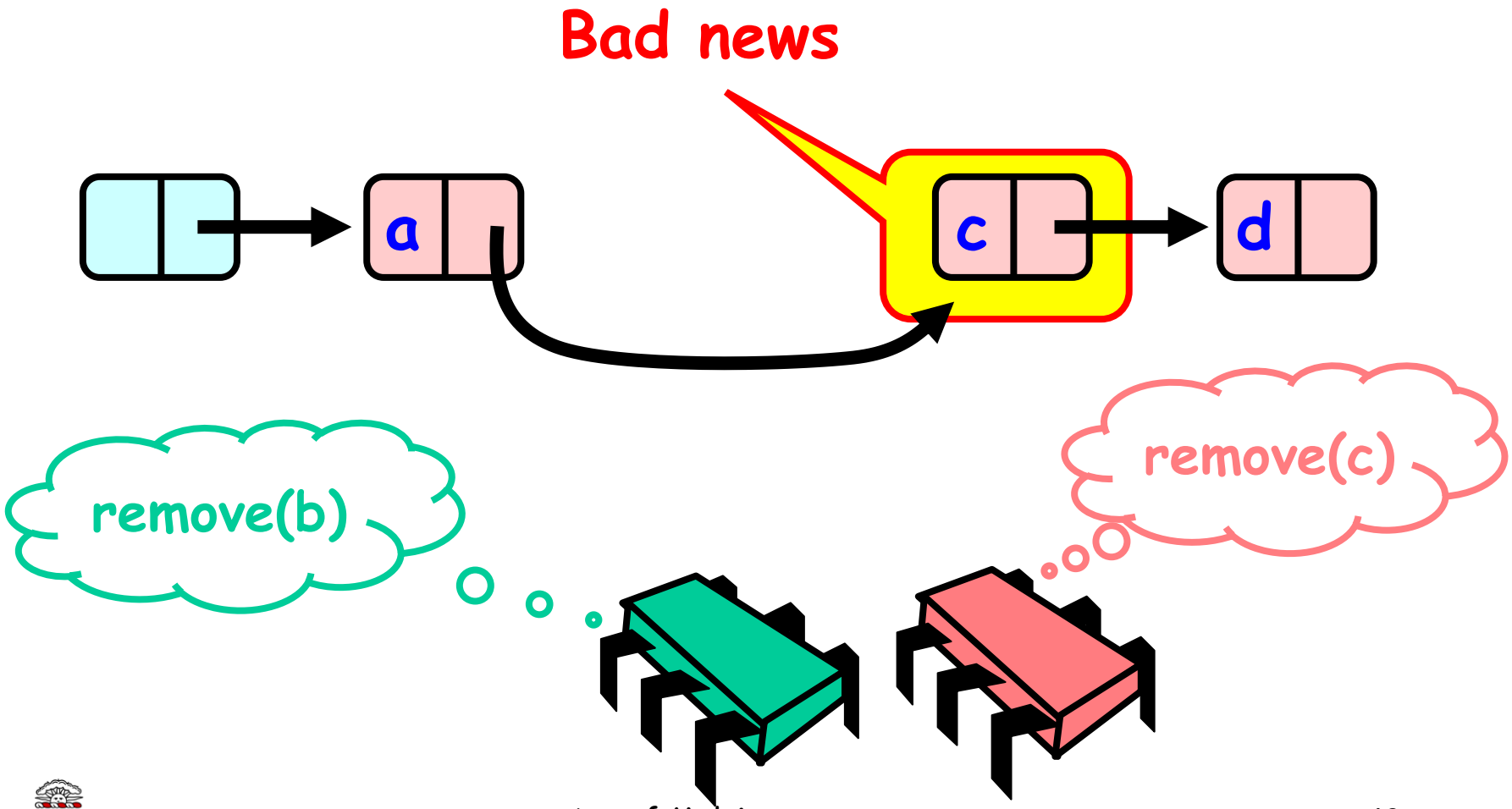
Removing a Node



Uh, Oh

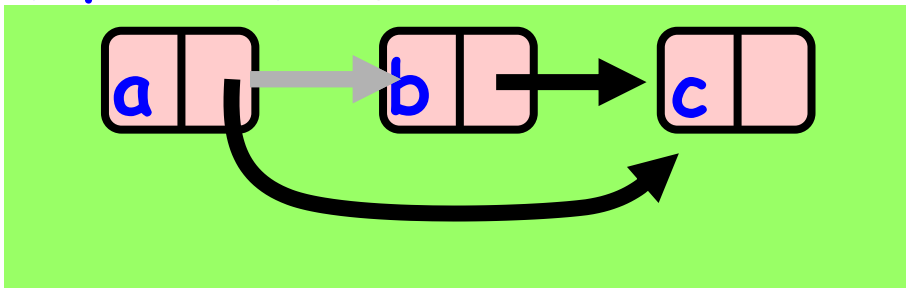


Uh, Oh

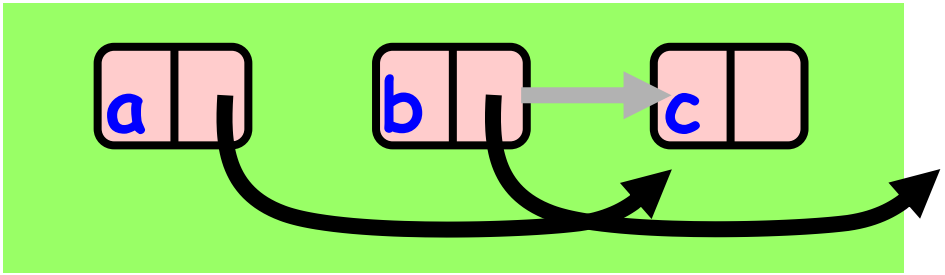


Problem

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



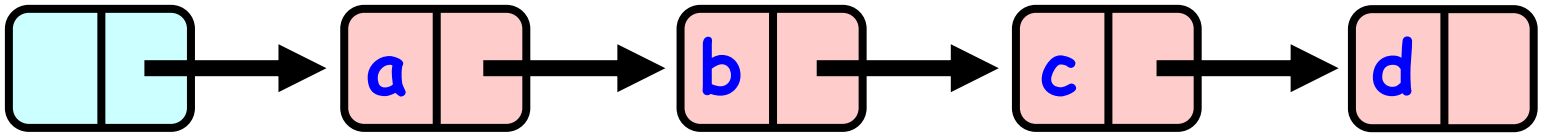
- Problem is,
  - Someone could delete c concurrently

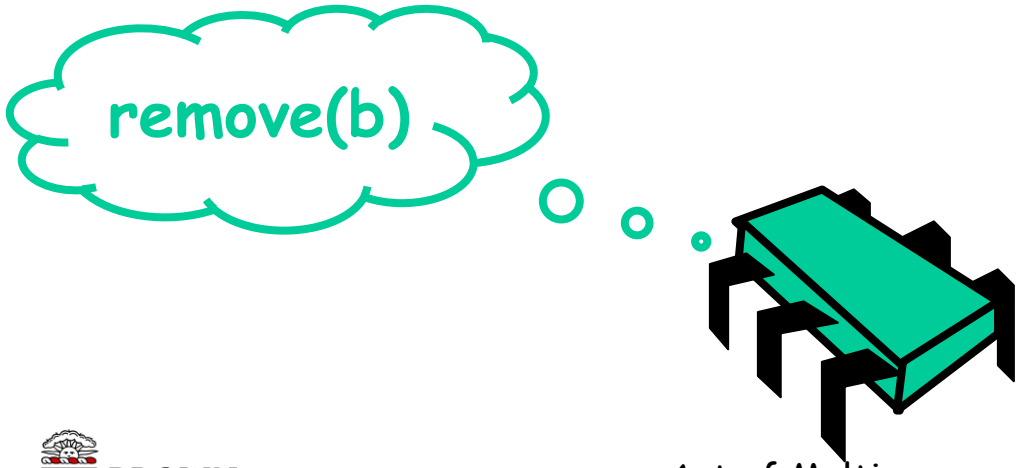


Insight

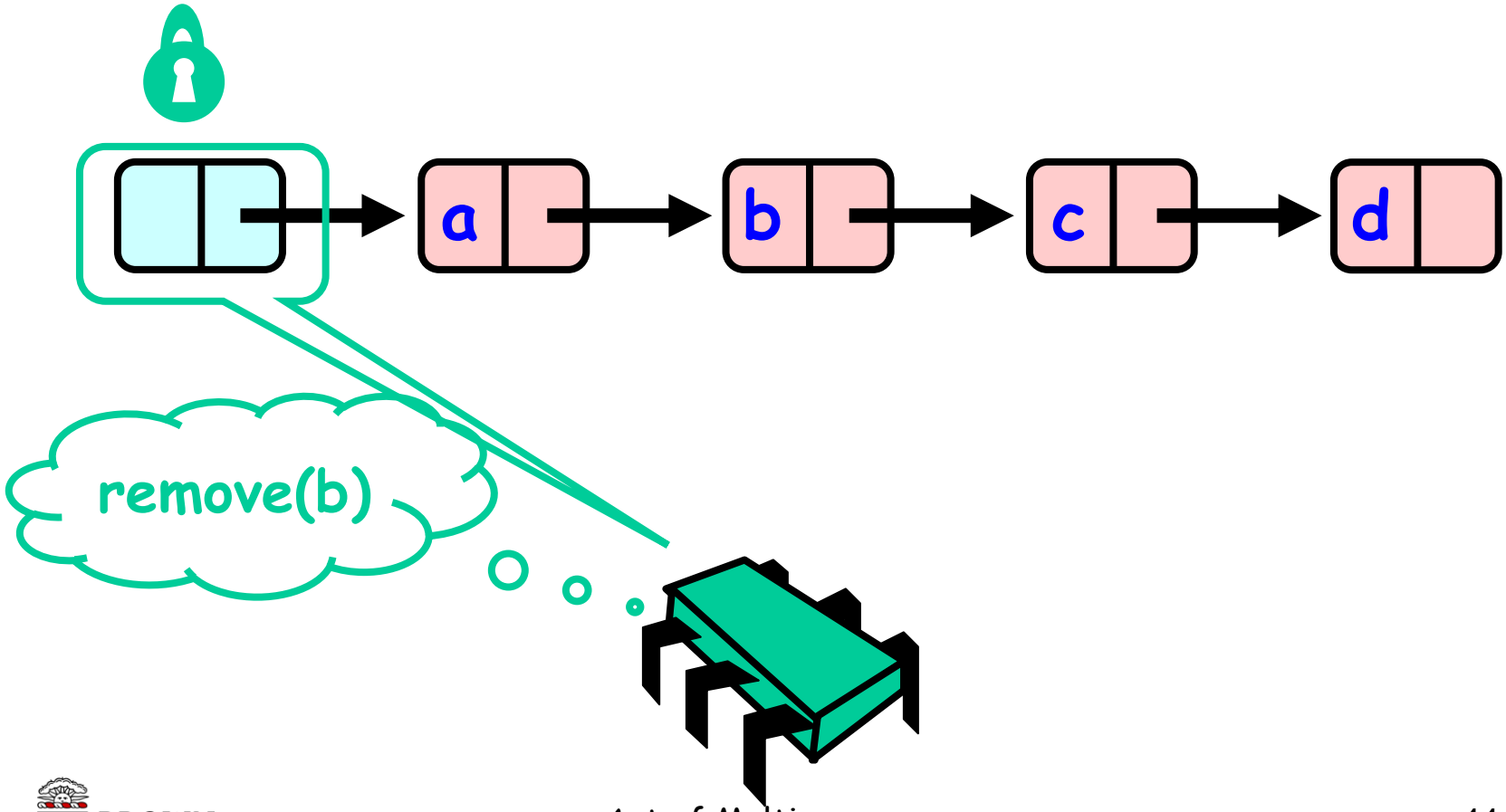
- If a node is locked
  - No one can delete node's *successor*
- If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works

Hand-Over-Hand Again

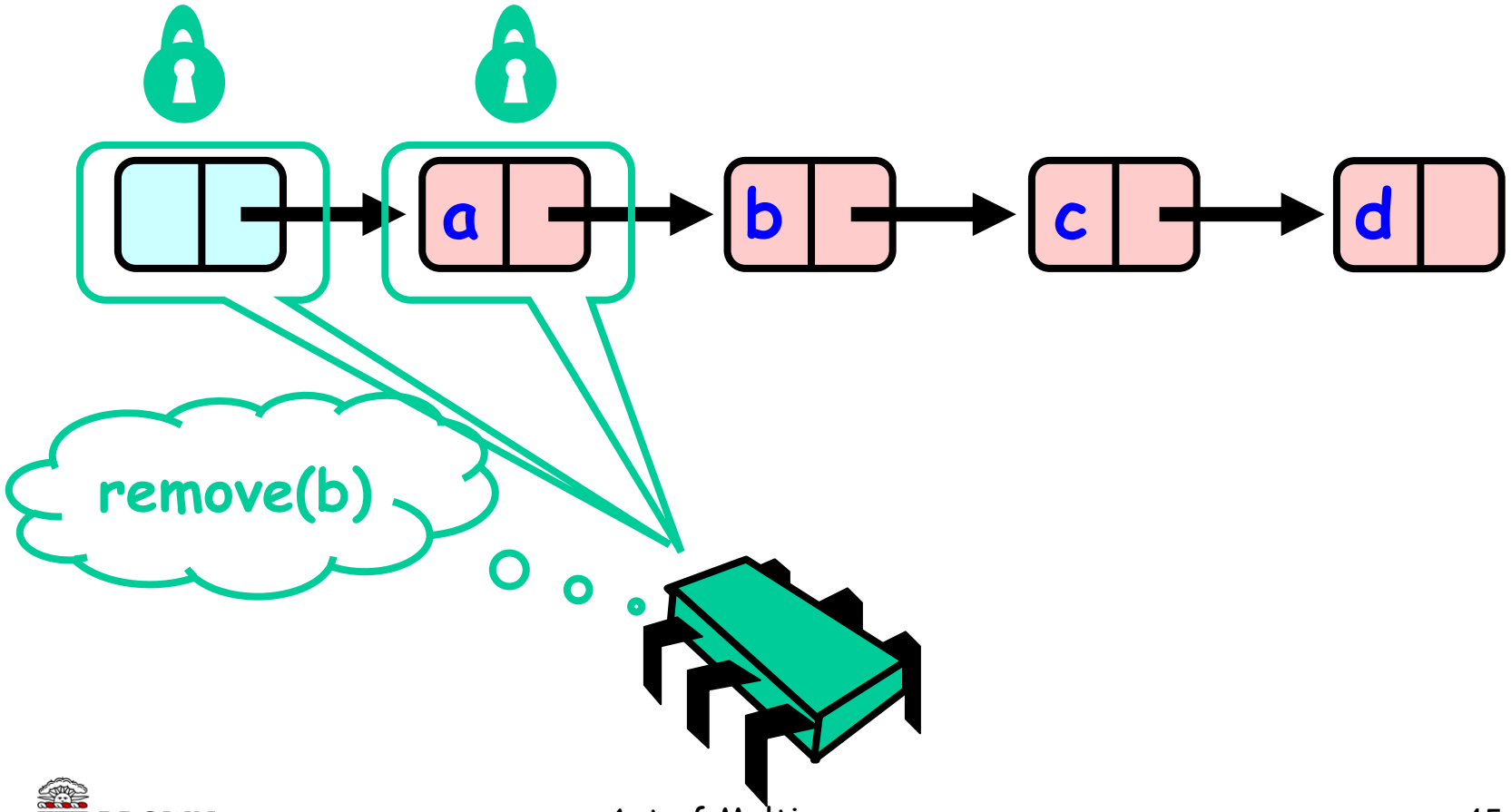




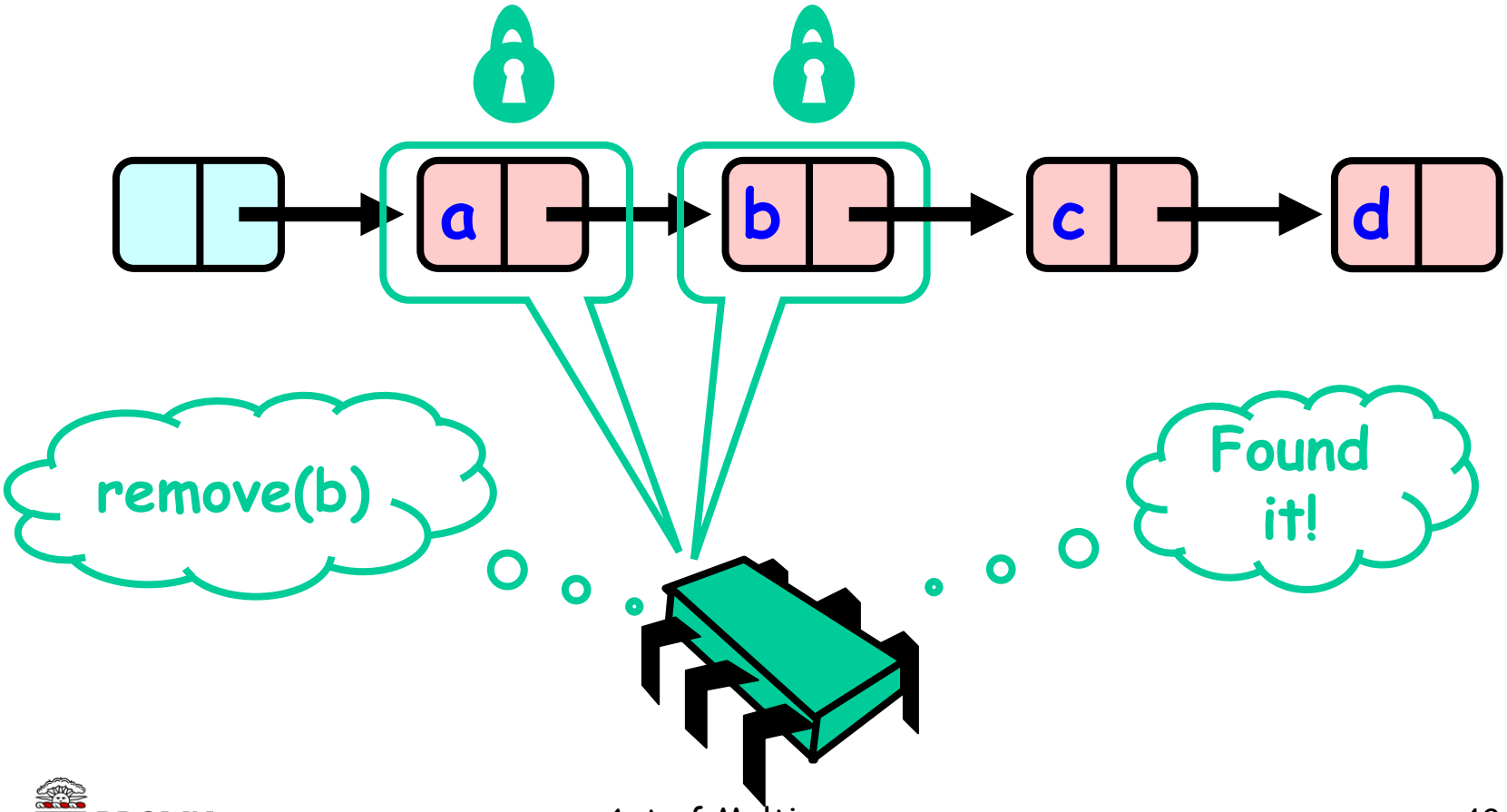
Hand-Over-Hand Again



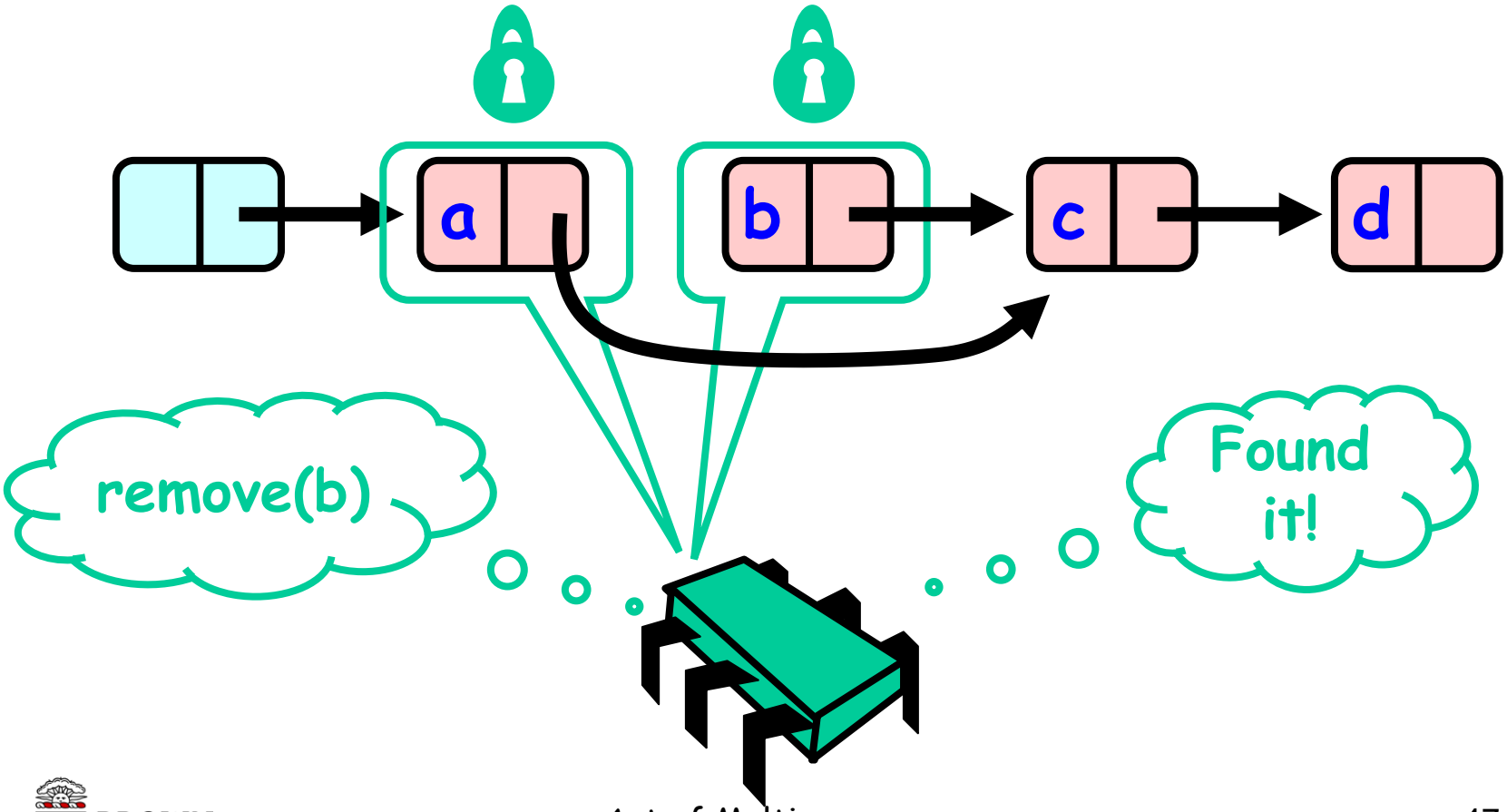
Hand-Over-Hand Again



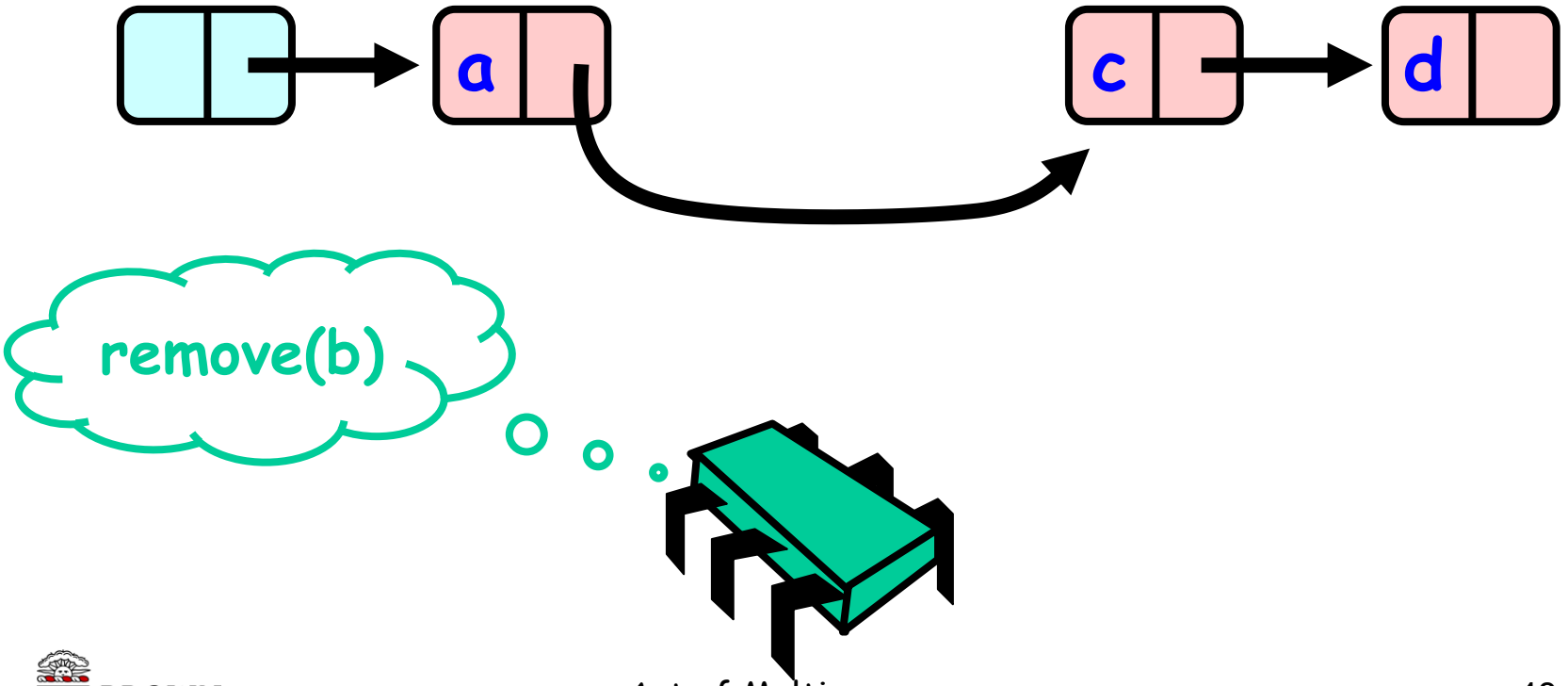
Hand-Over-Hand Again



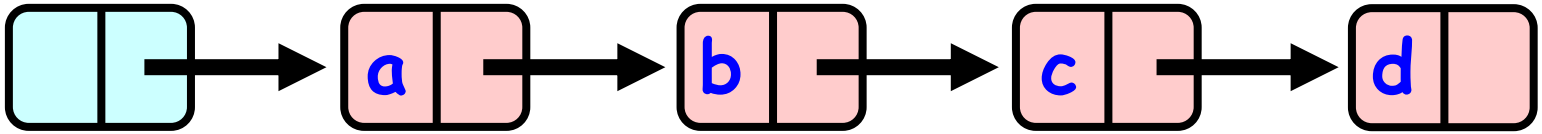
Hand-Over-Hand Again

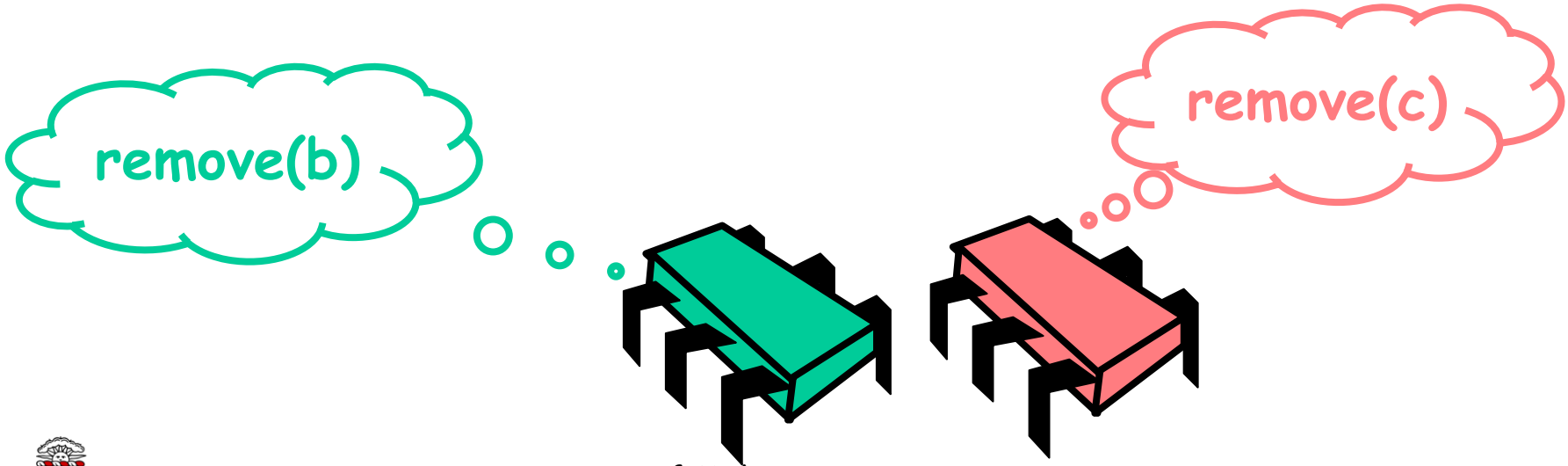


Hand-Over-Hand Again

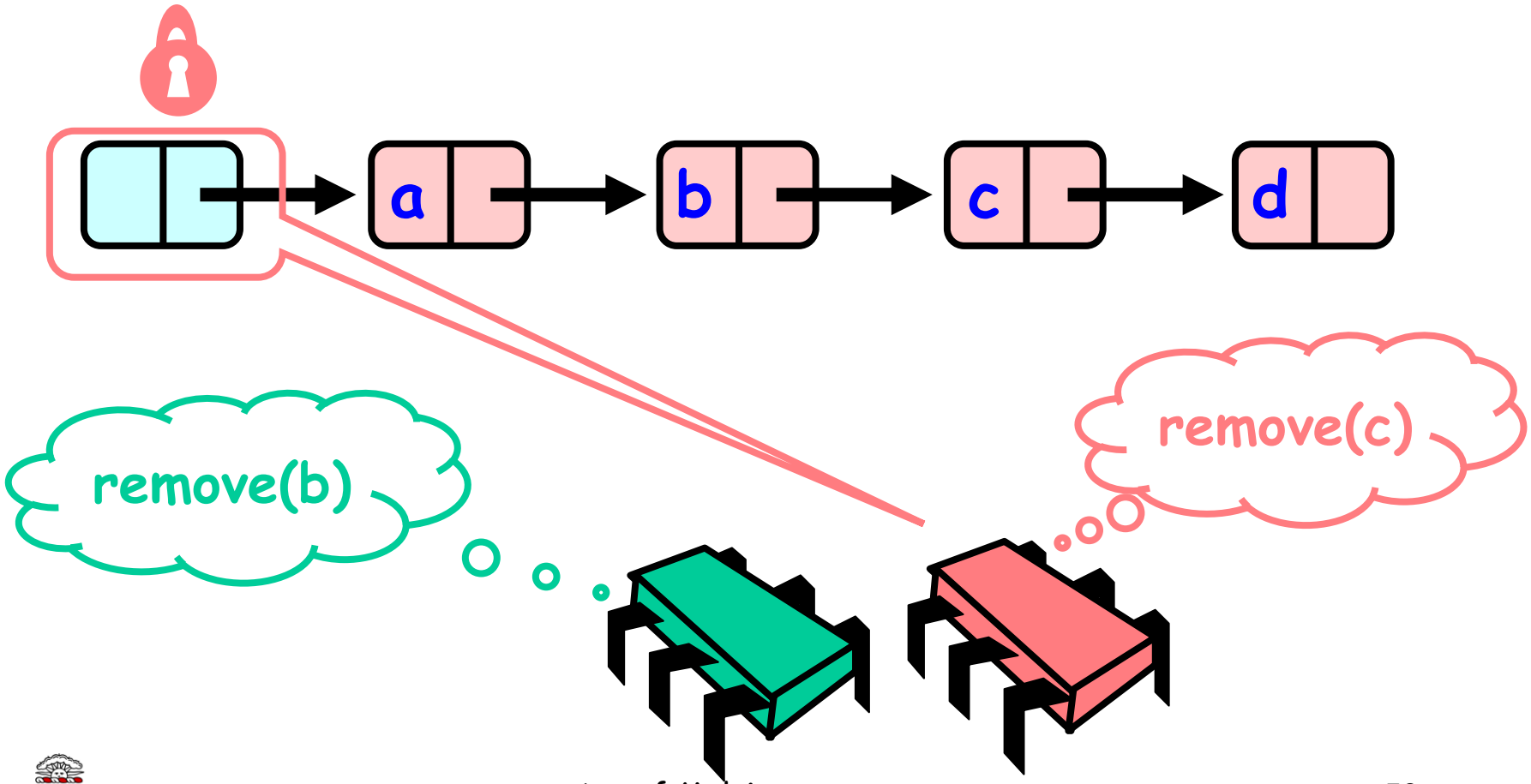


Removing a Node

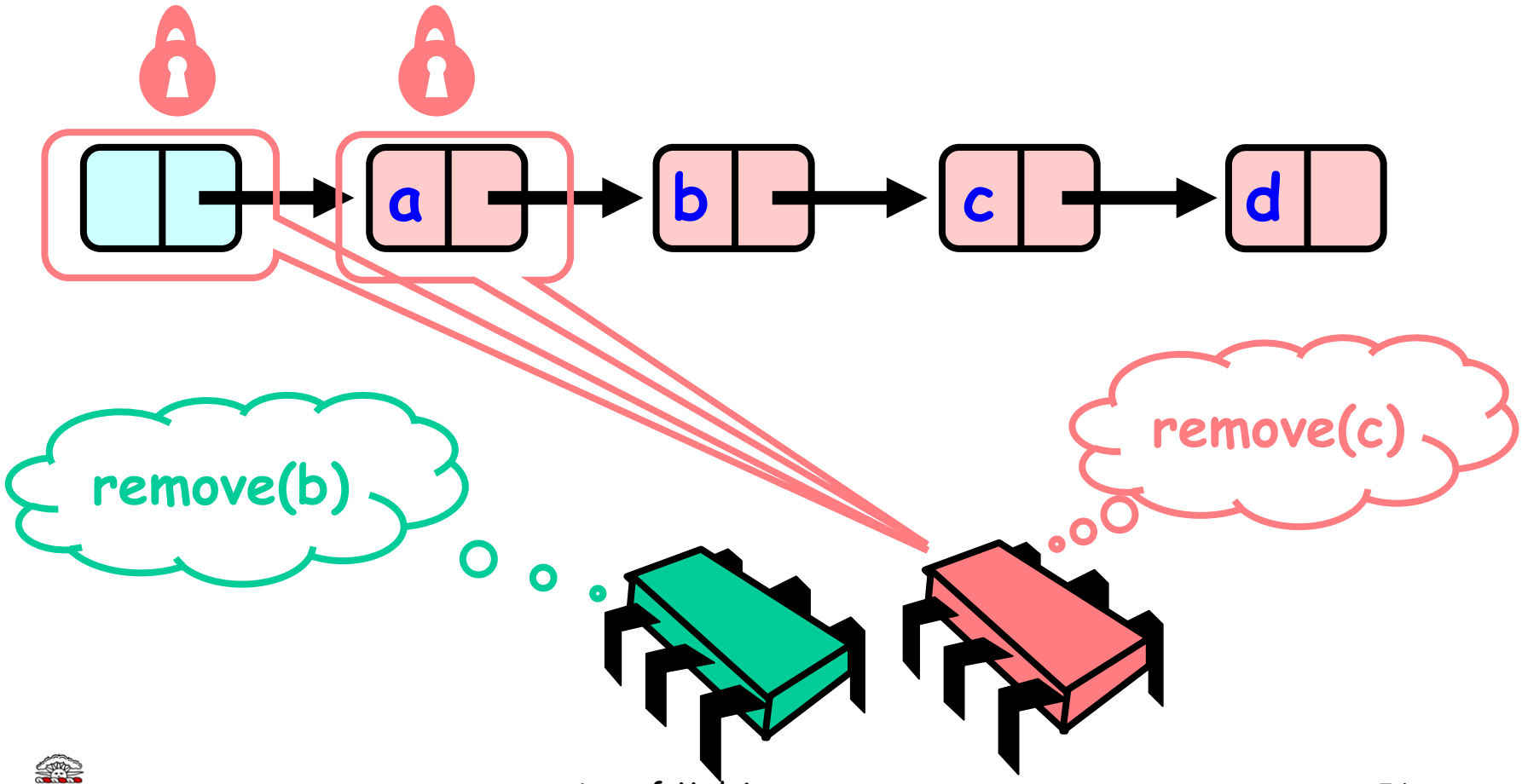




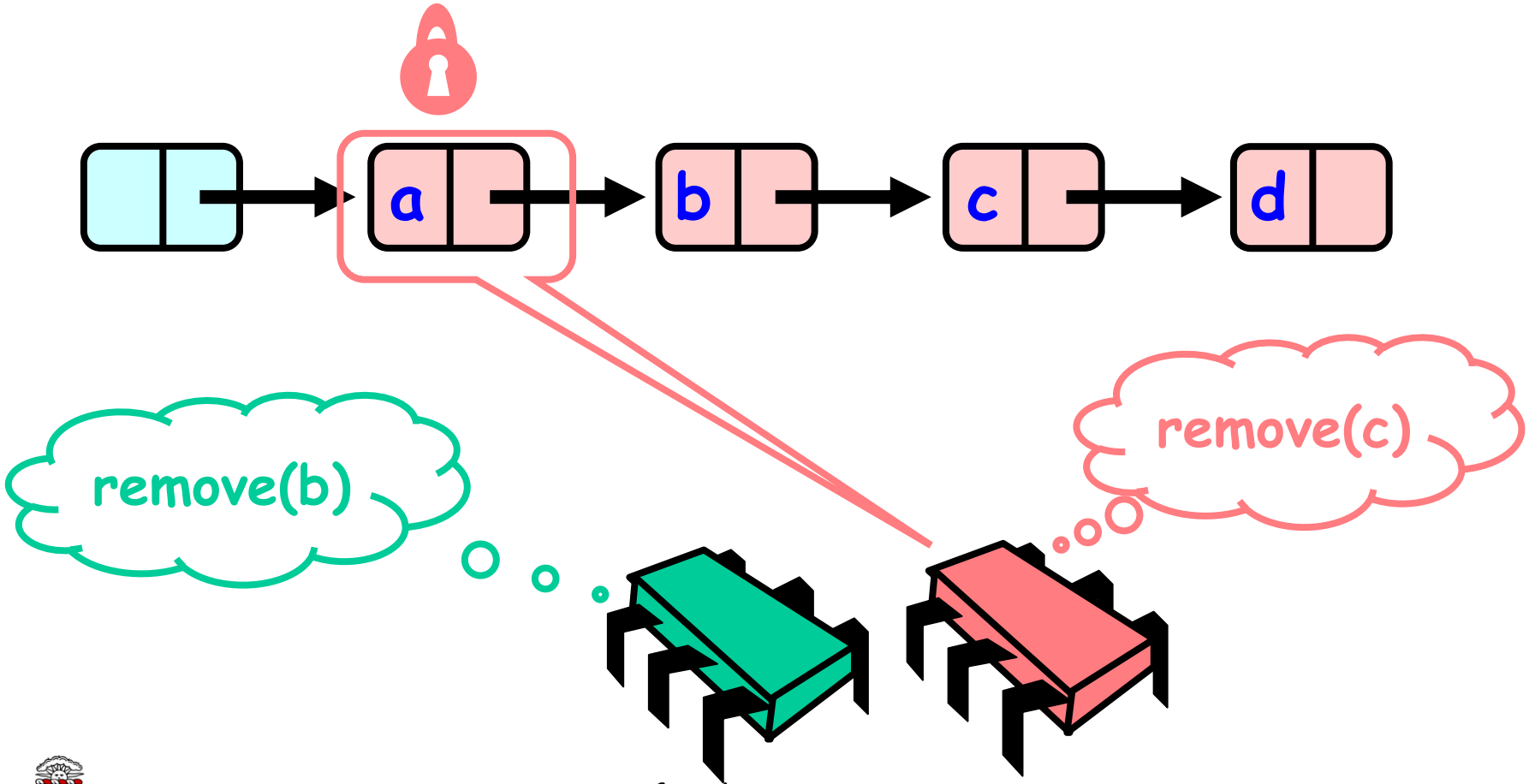
Removing a Node



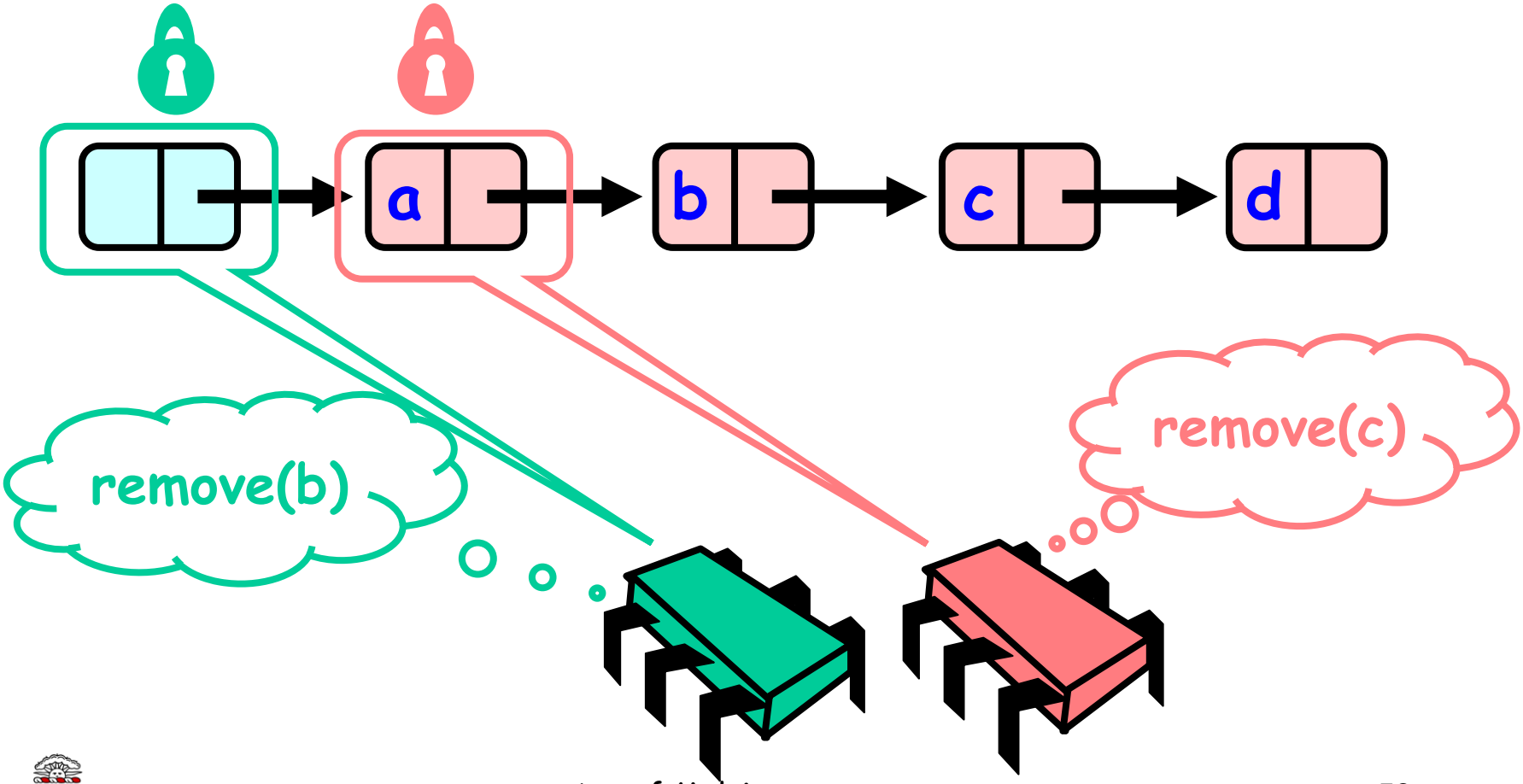
Removing a Node



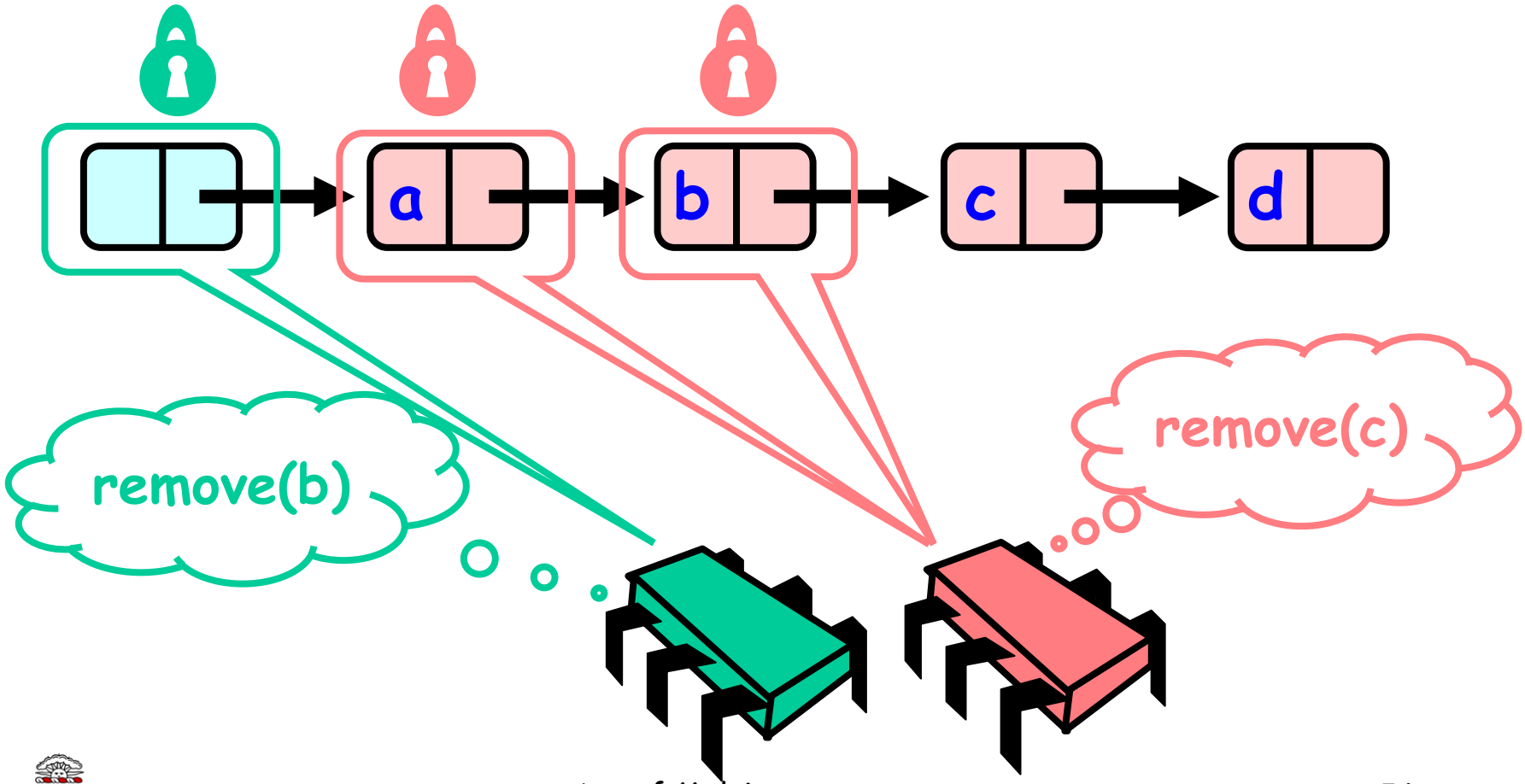
Removing a Node



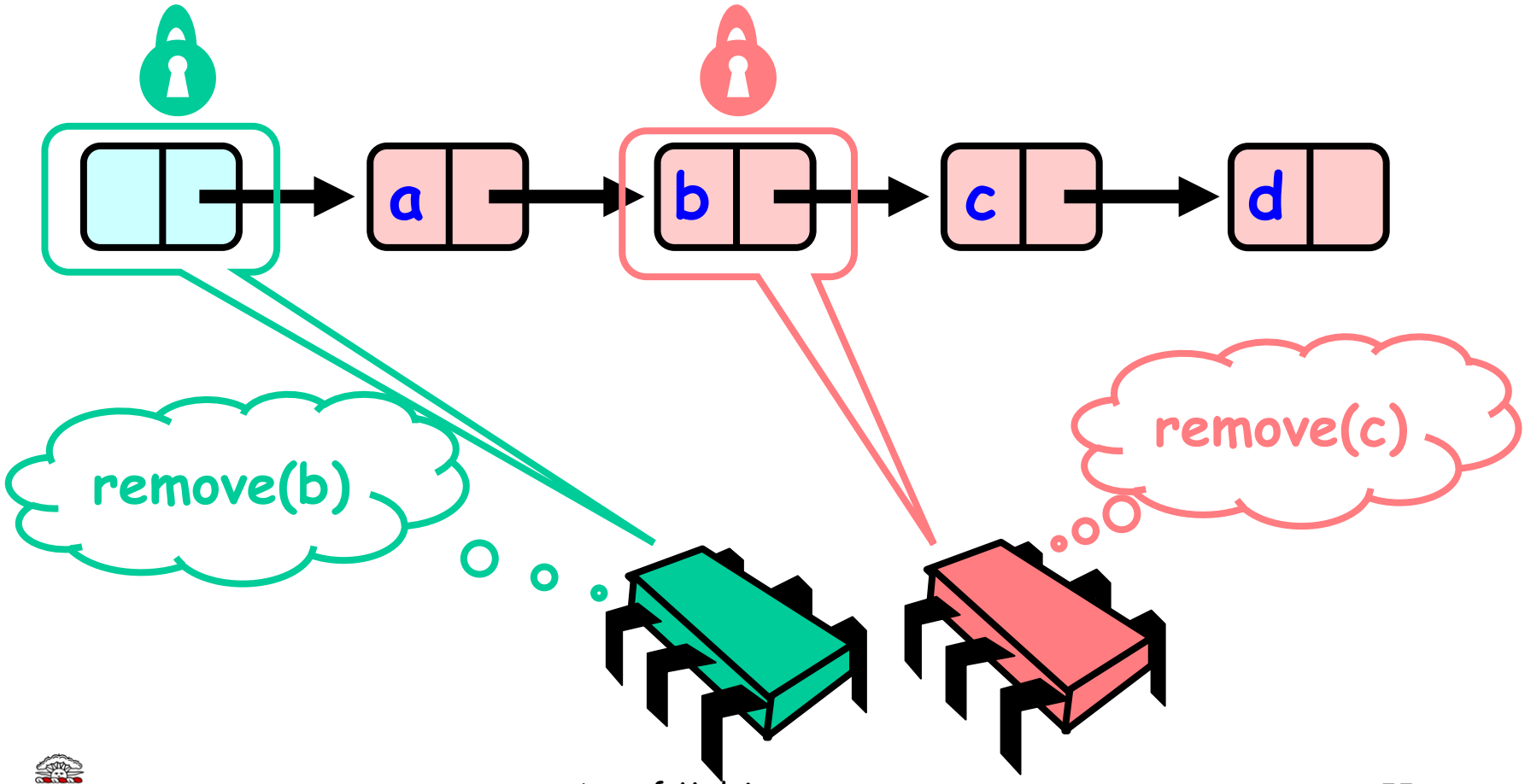
Removing a Node



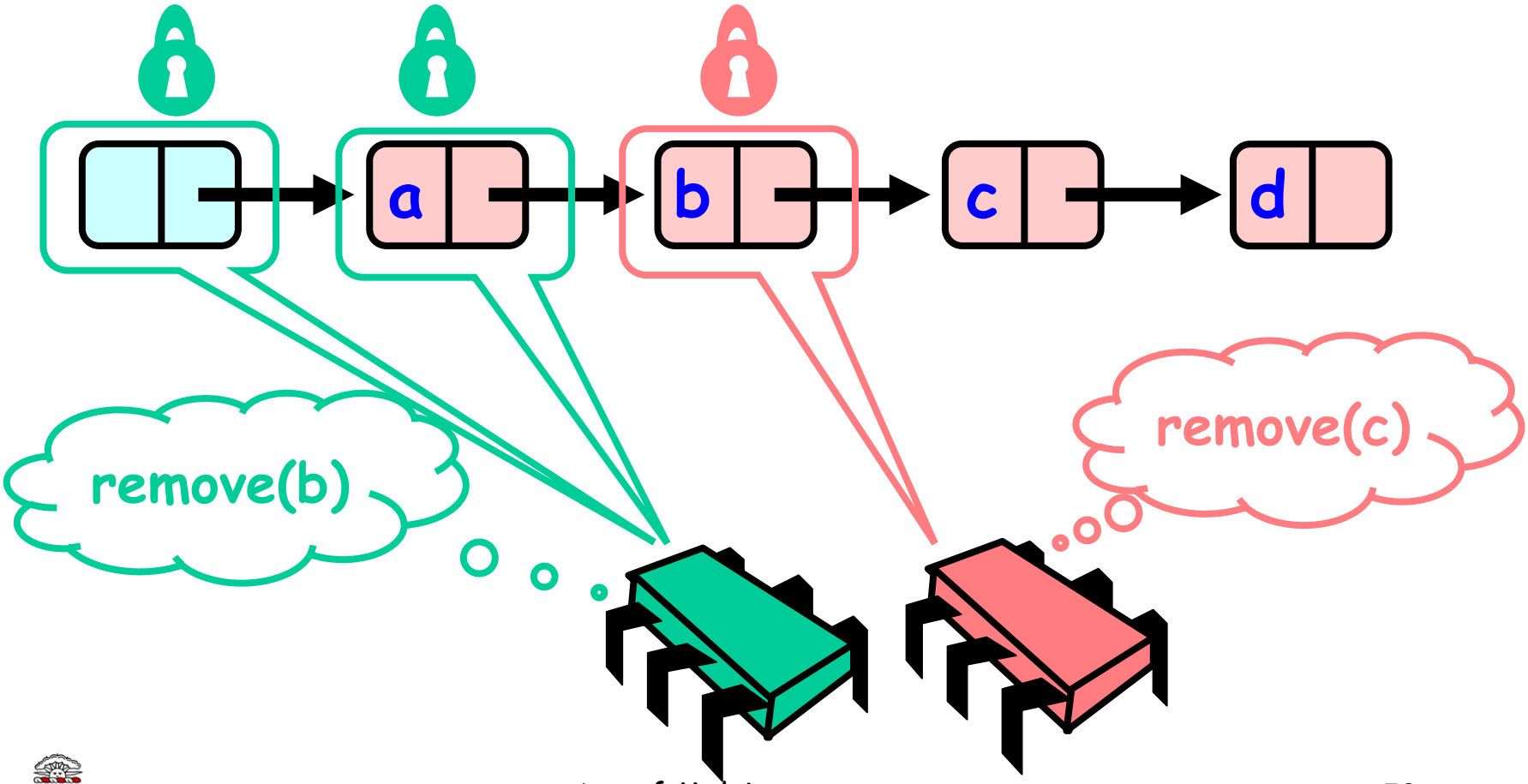
Removing a Node



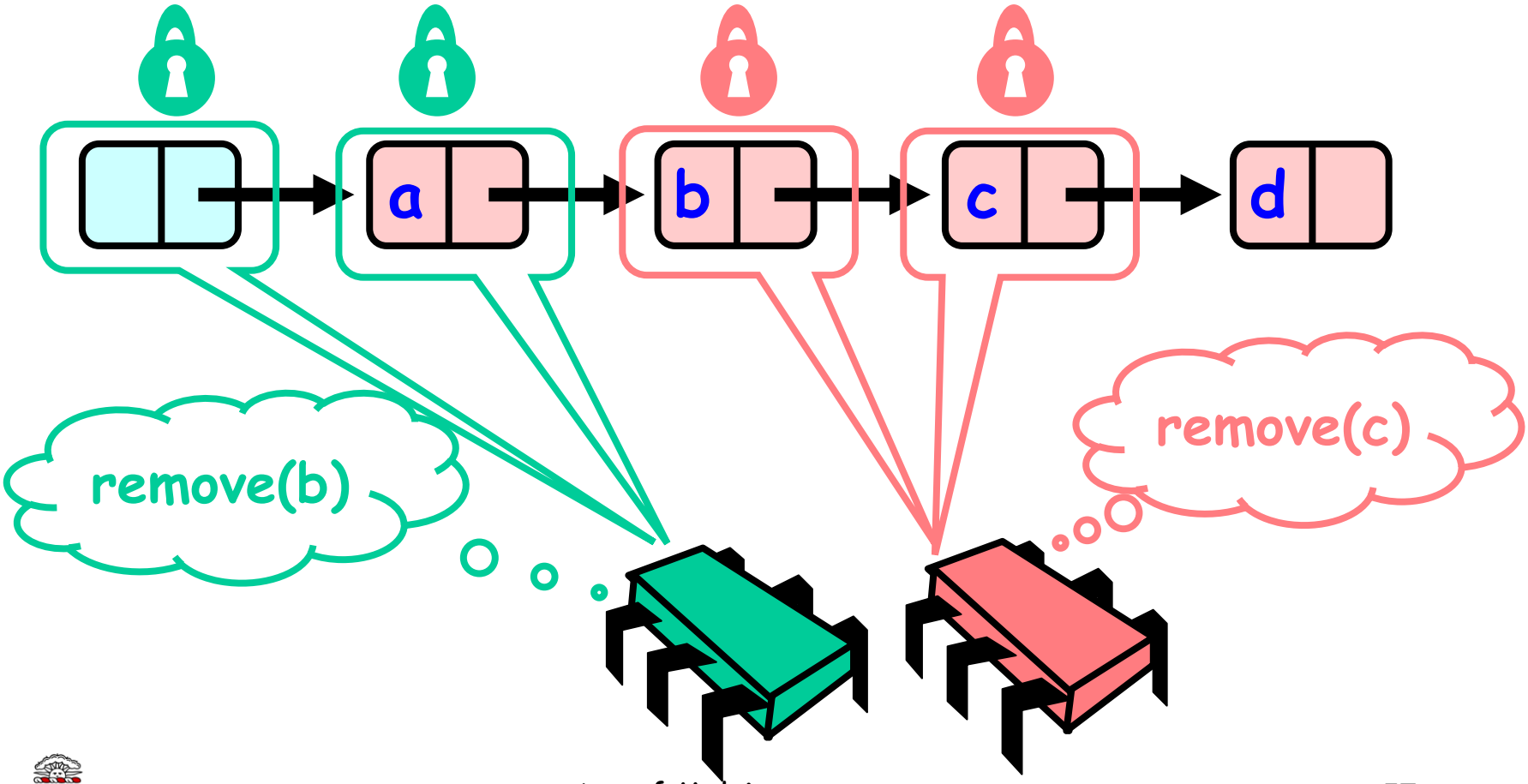
Removing a Node



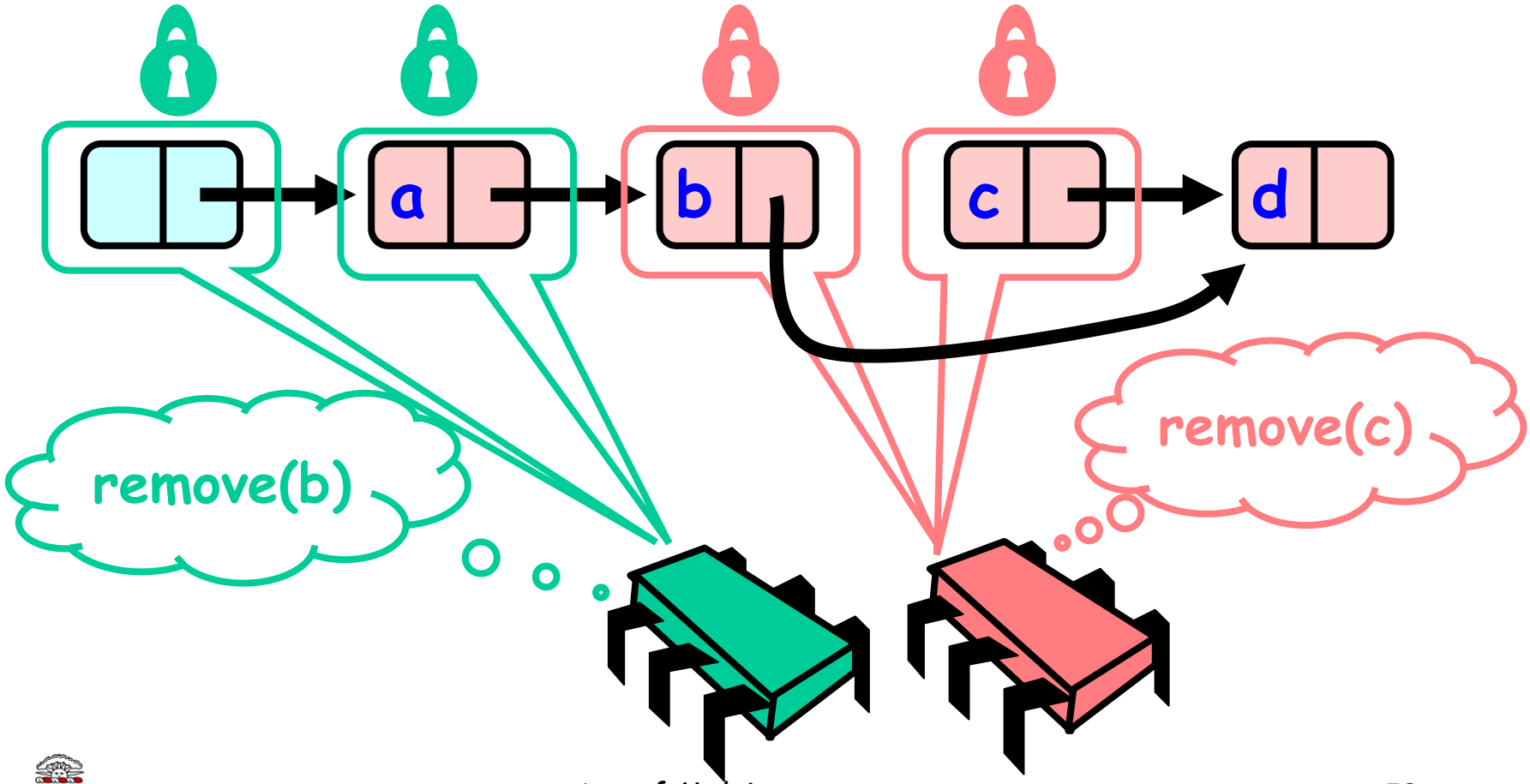
Removing a Node



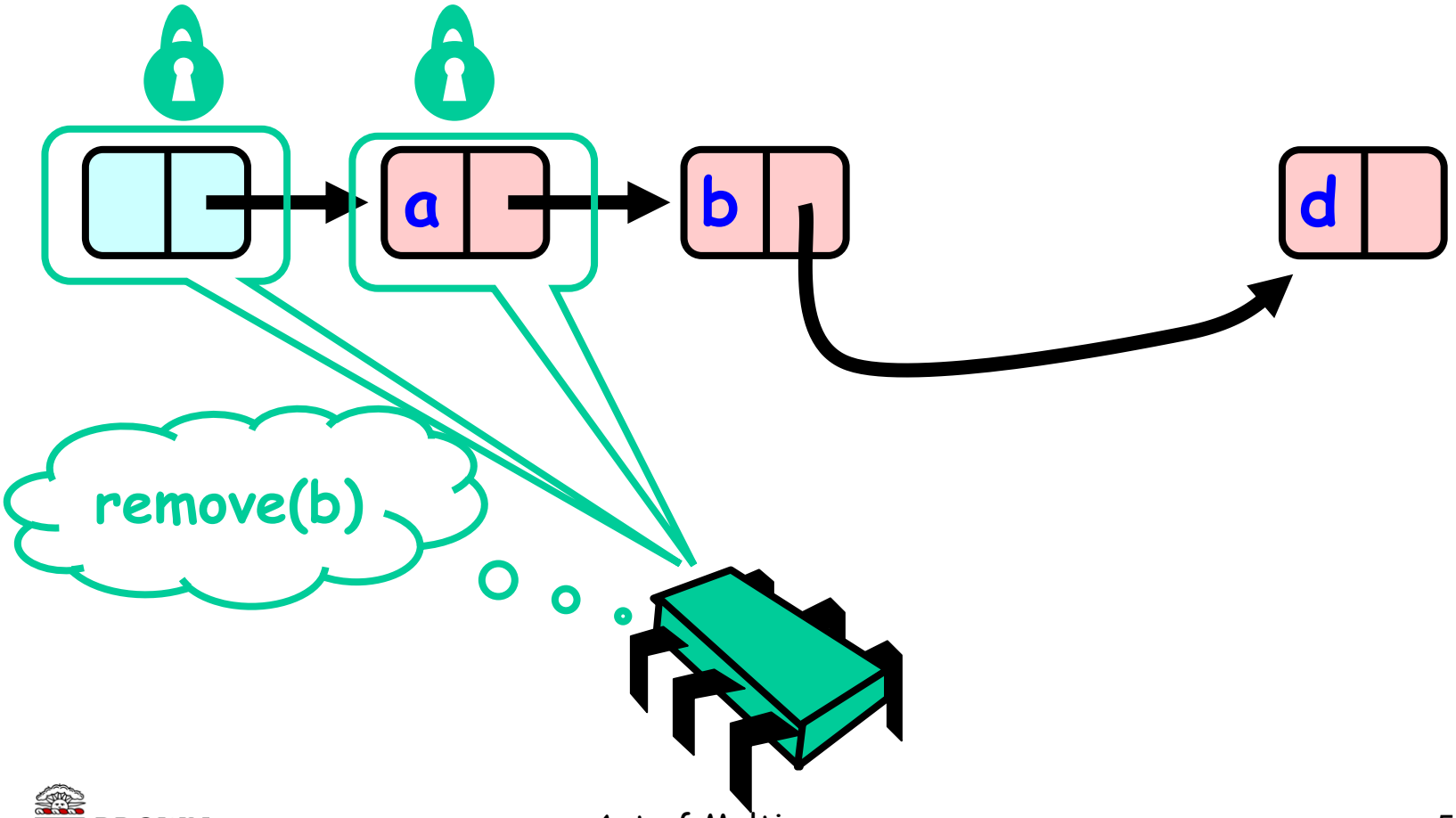
Removing a Node



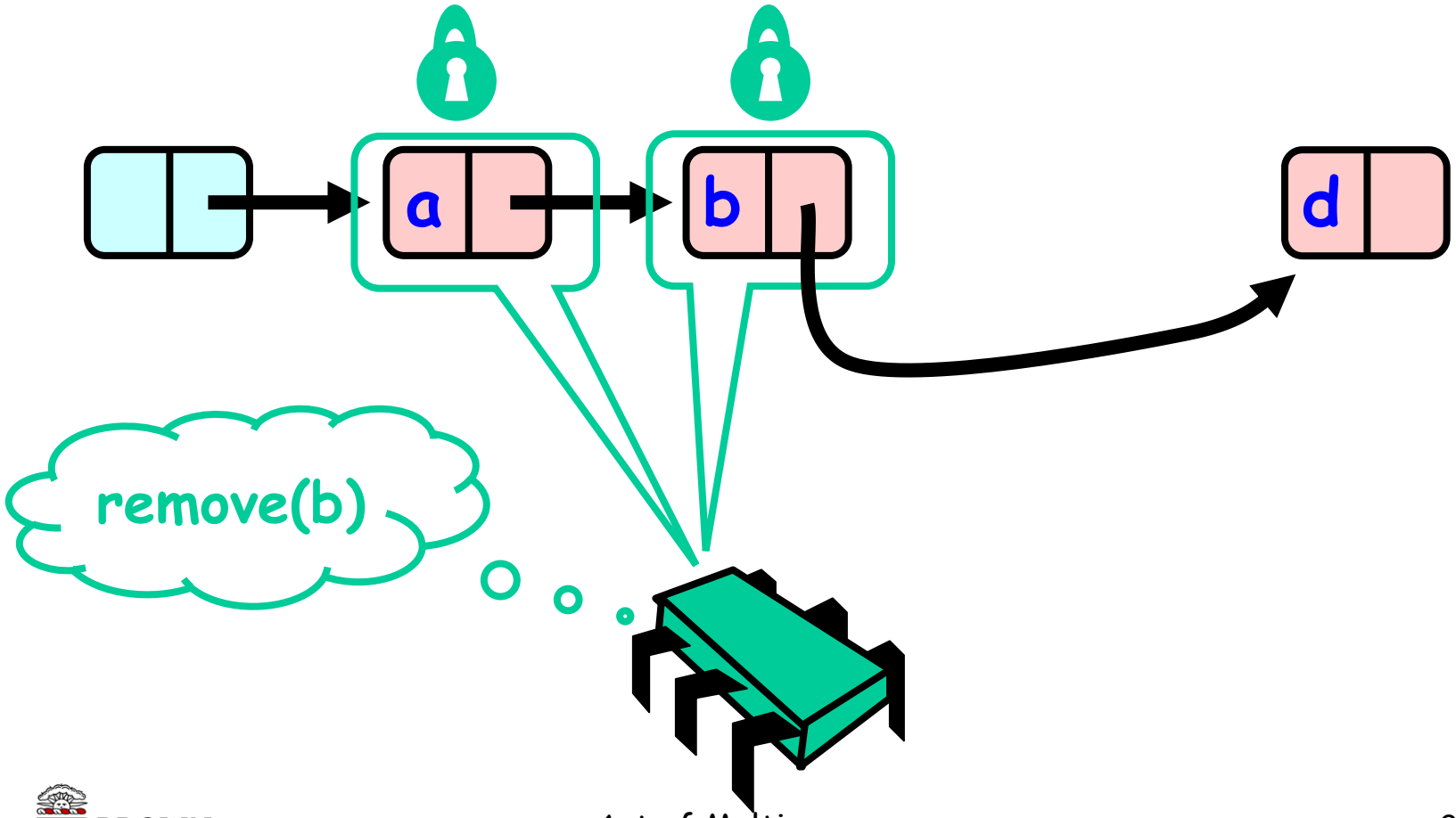
Removing a Node



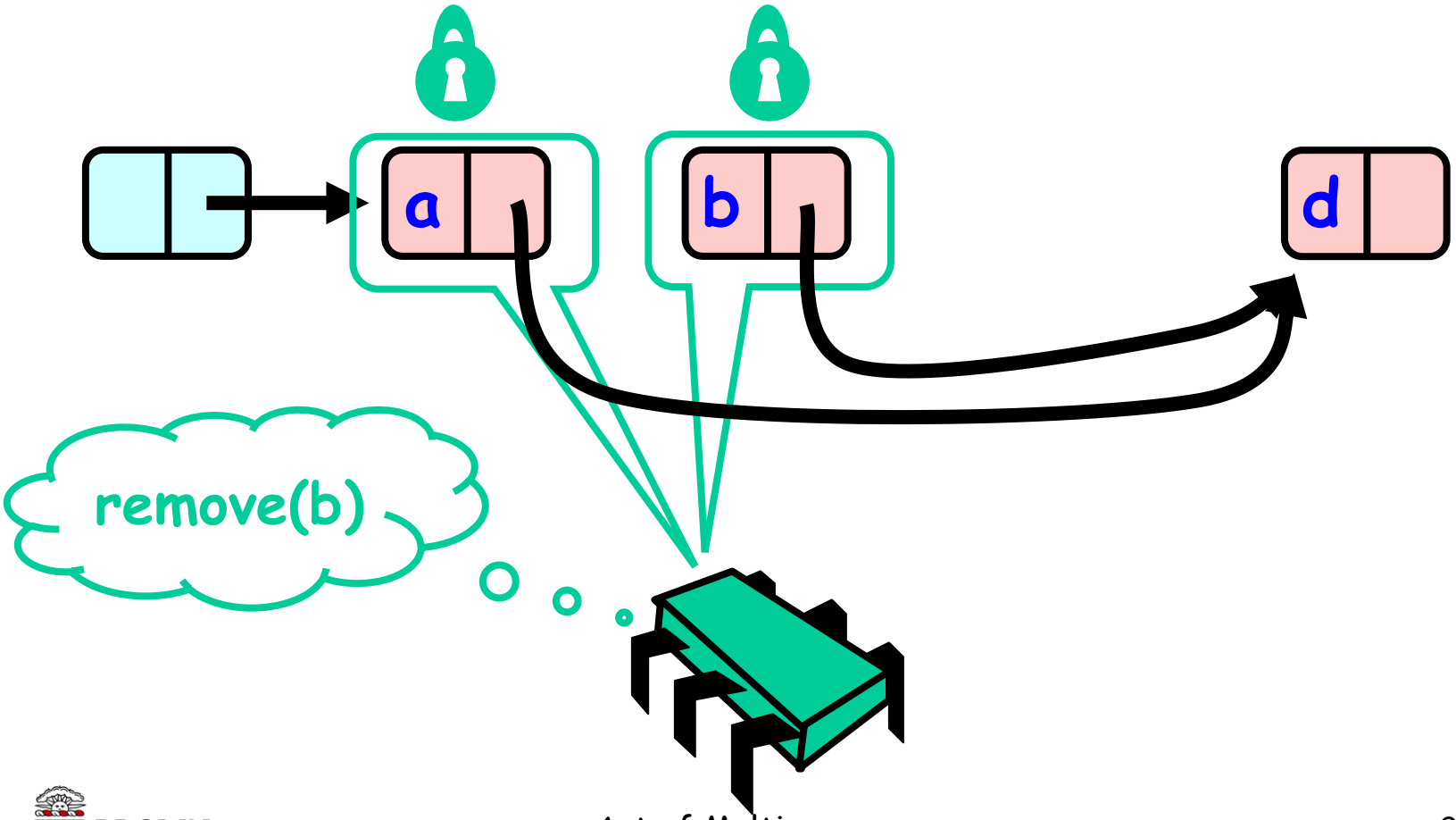
Removing a Node



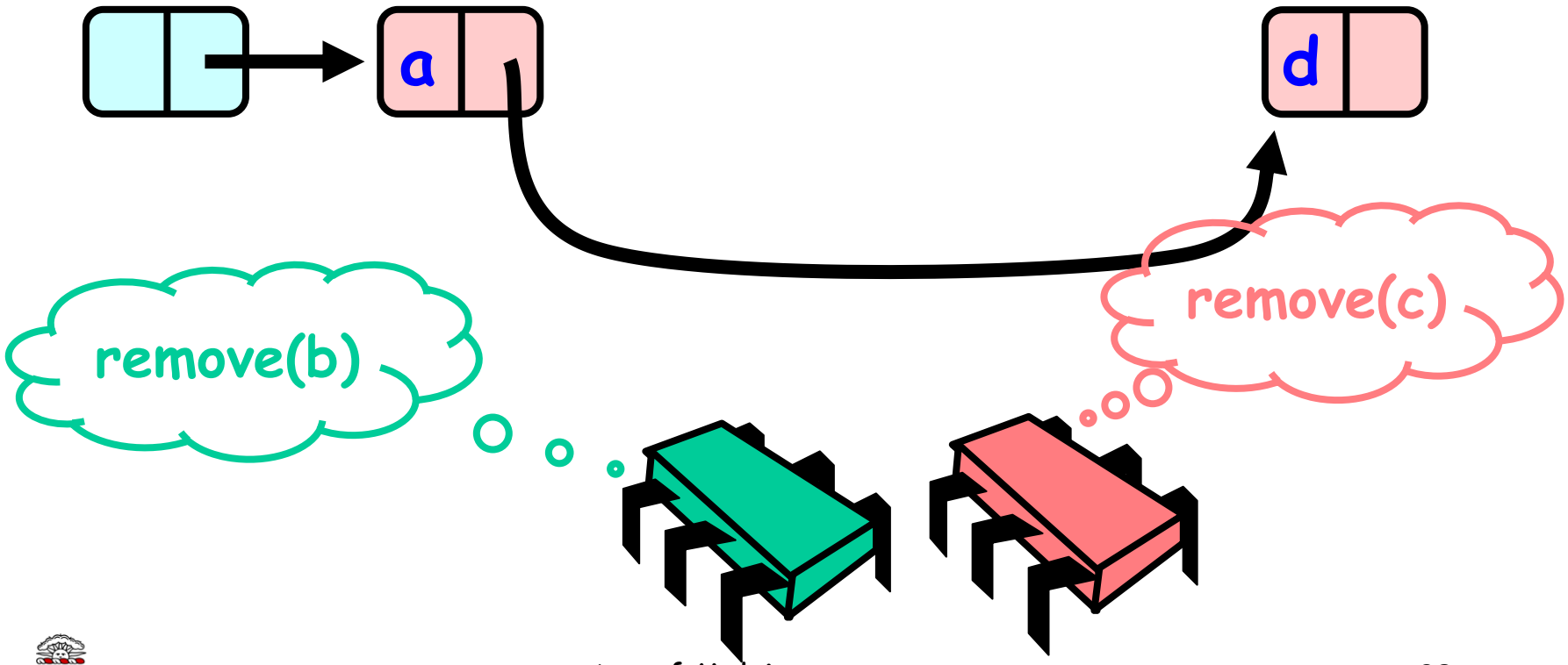
Removing a Node



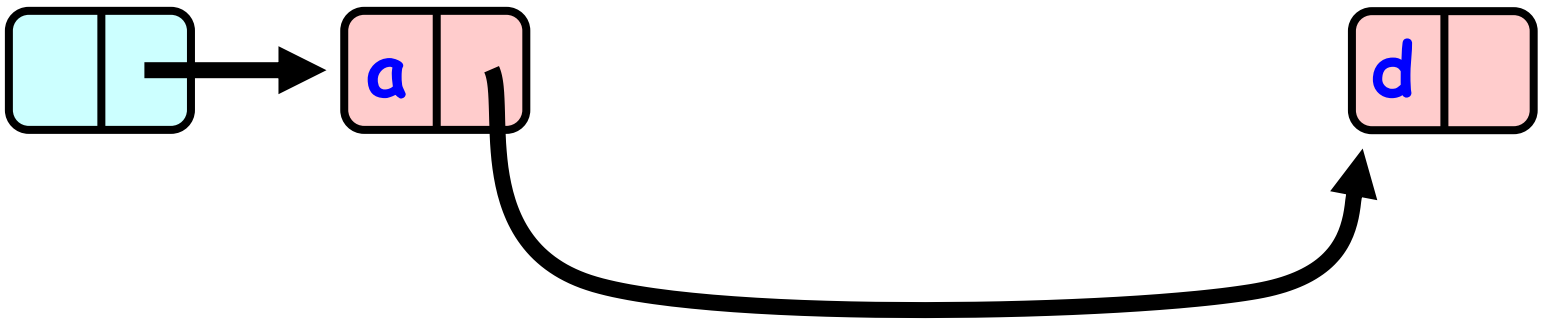
Removing a Node



Removing a Node



Removing a Node



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

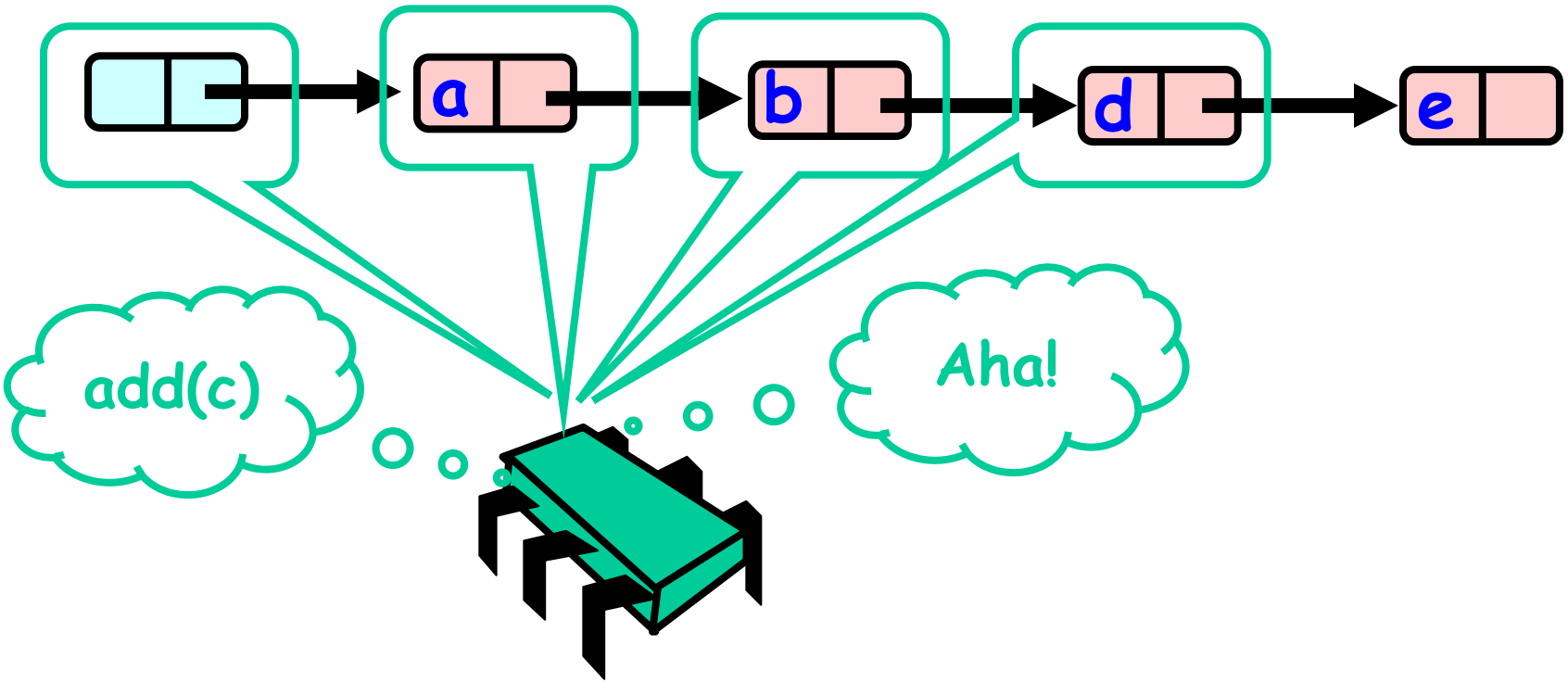
Drawbacks

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

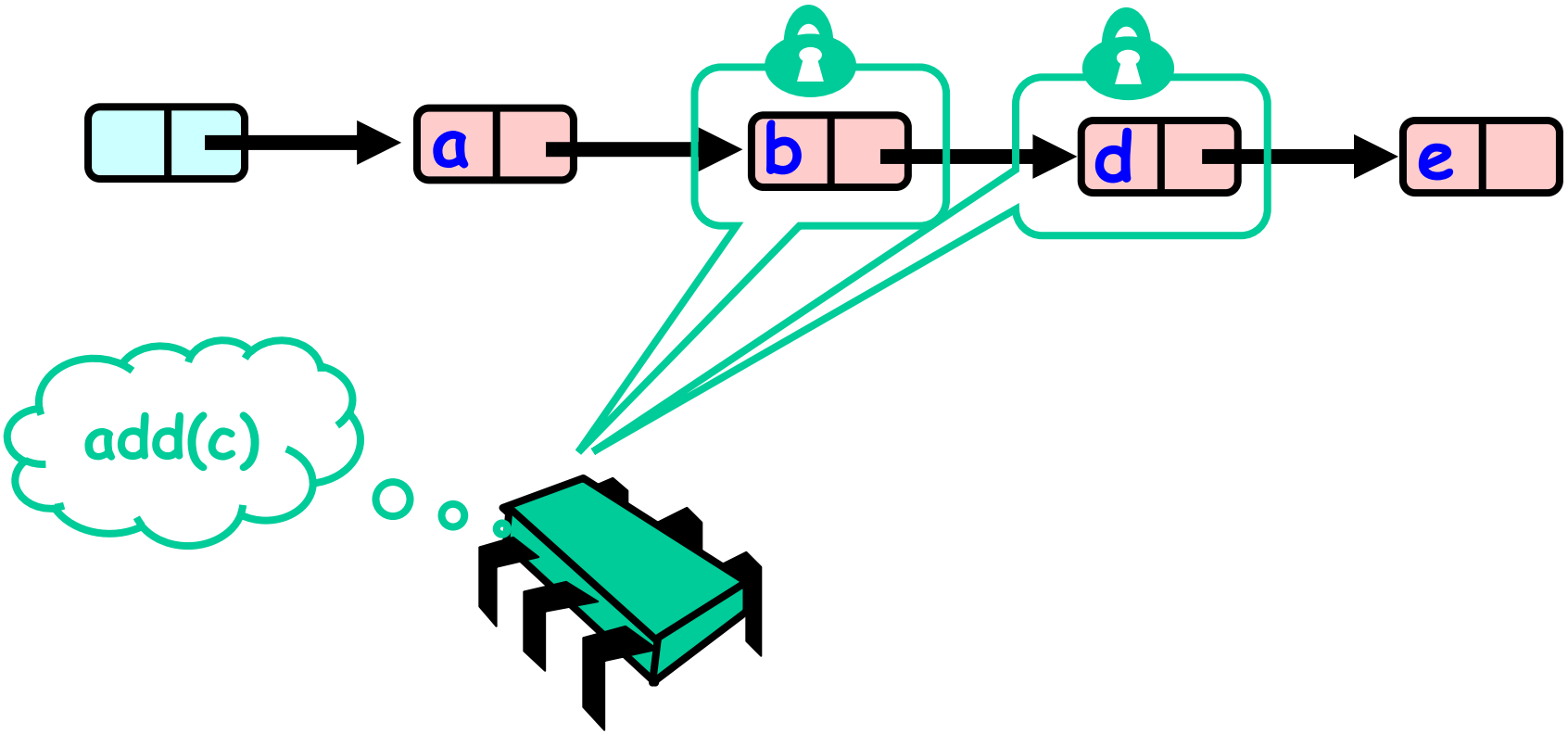
Optimistic Synchronization

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

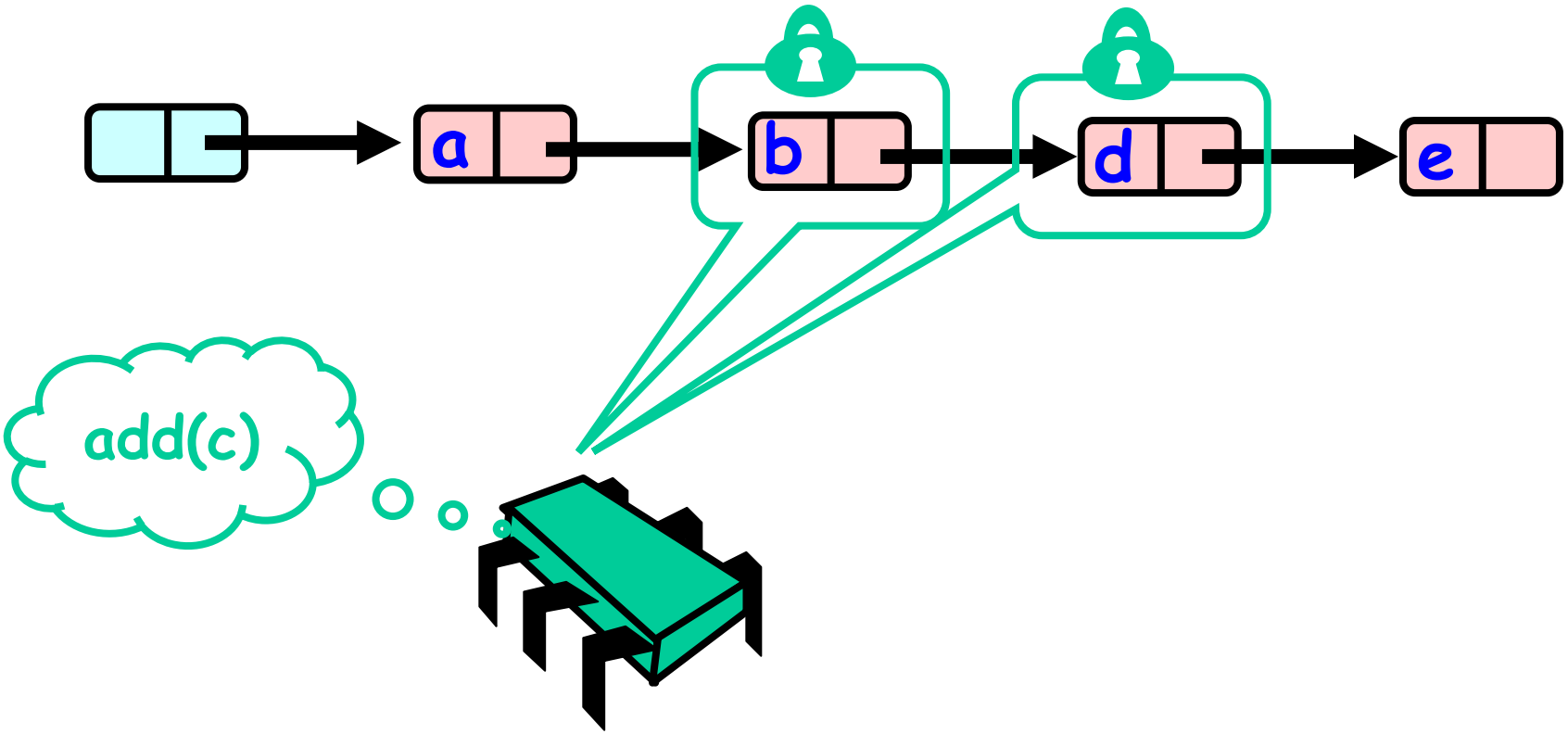
Optimistic: Traverse without Locking



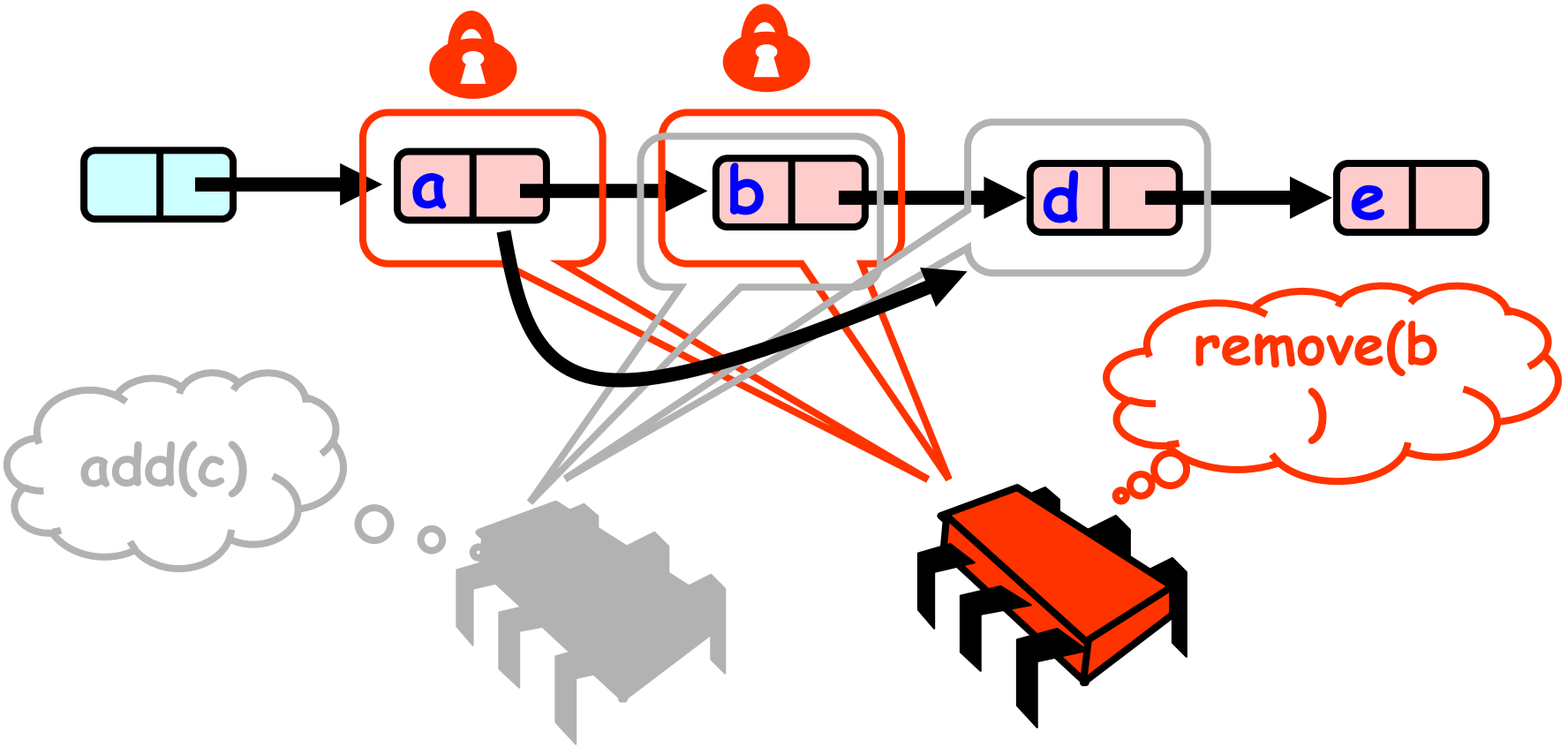
Optimistic: Lock and Load



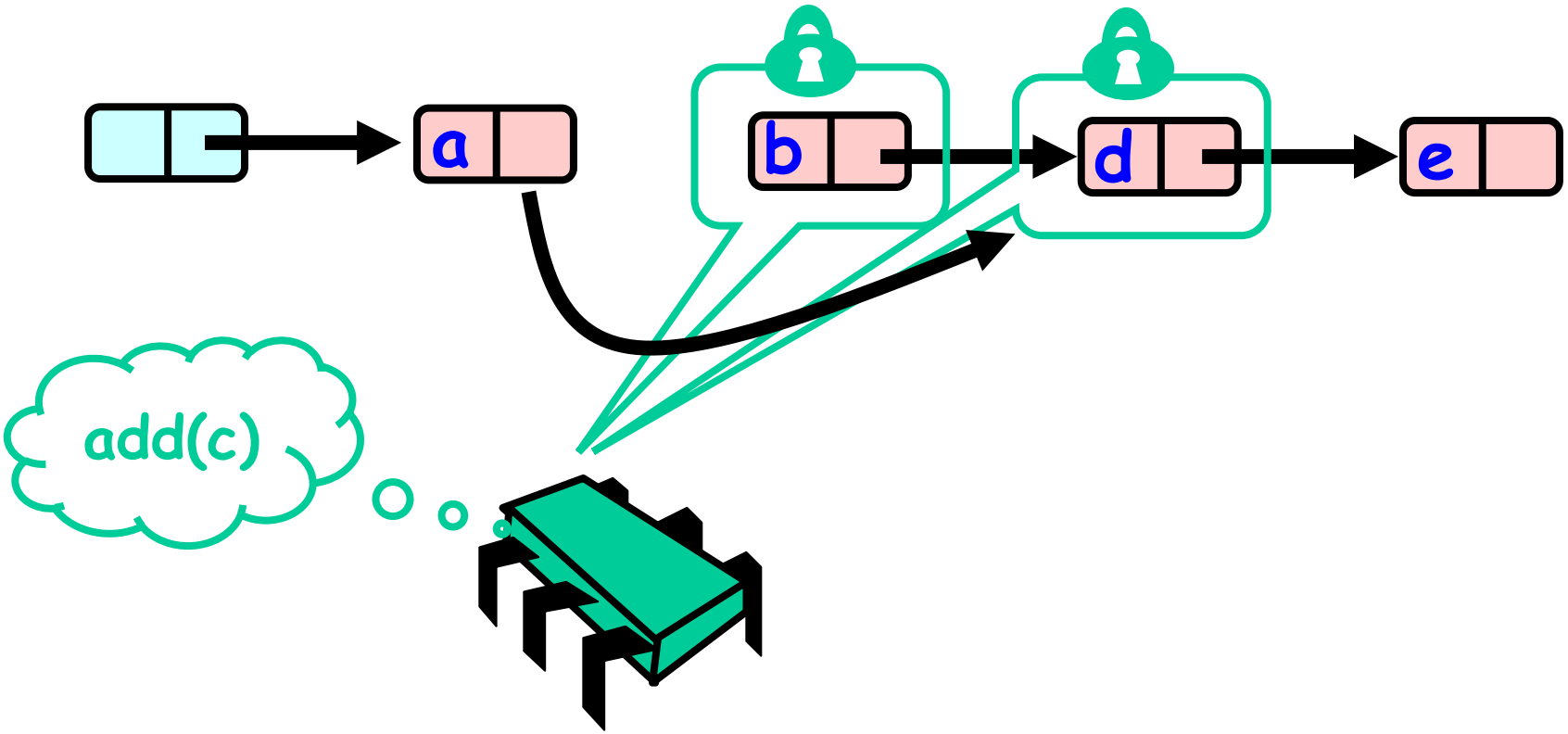
What Can Possibly Go Wrong?



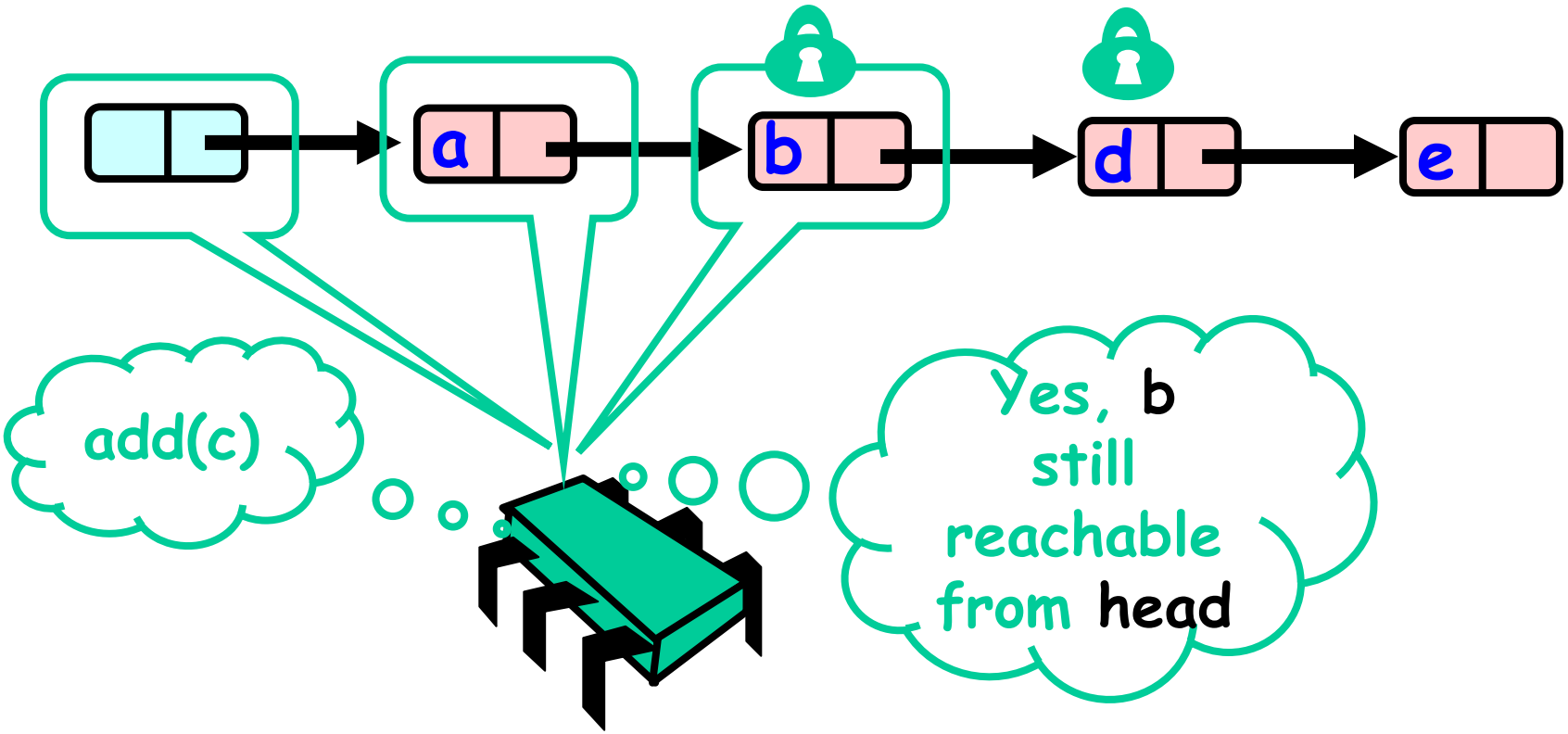
What Can Possibly Go Wrong?



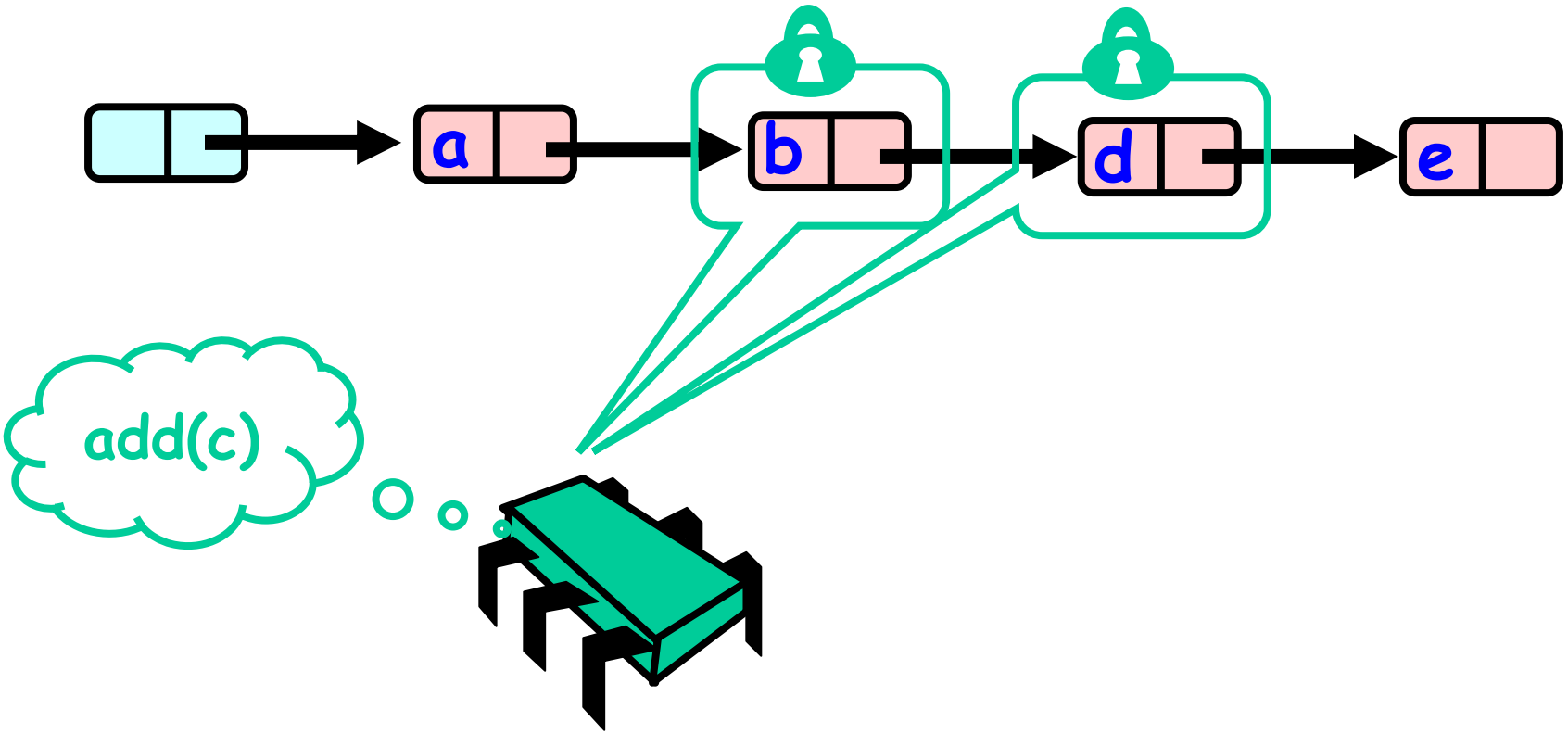
What Can Possibly Go Wrong?



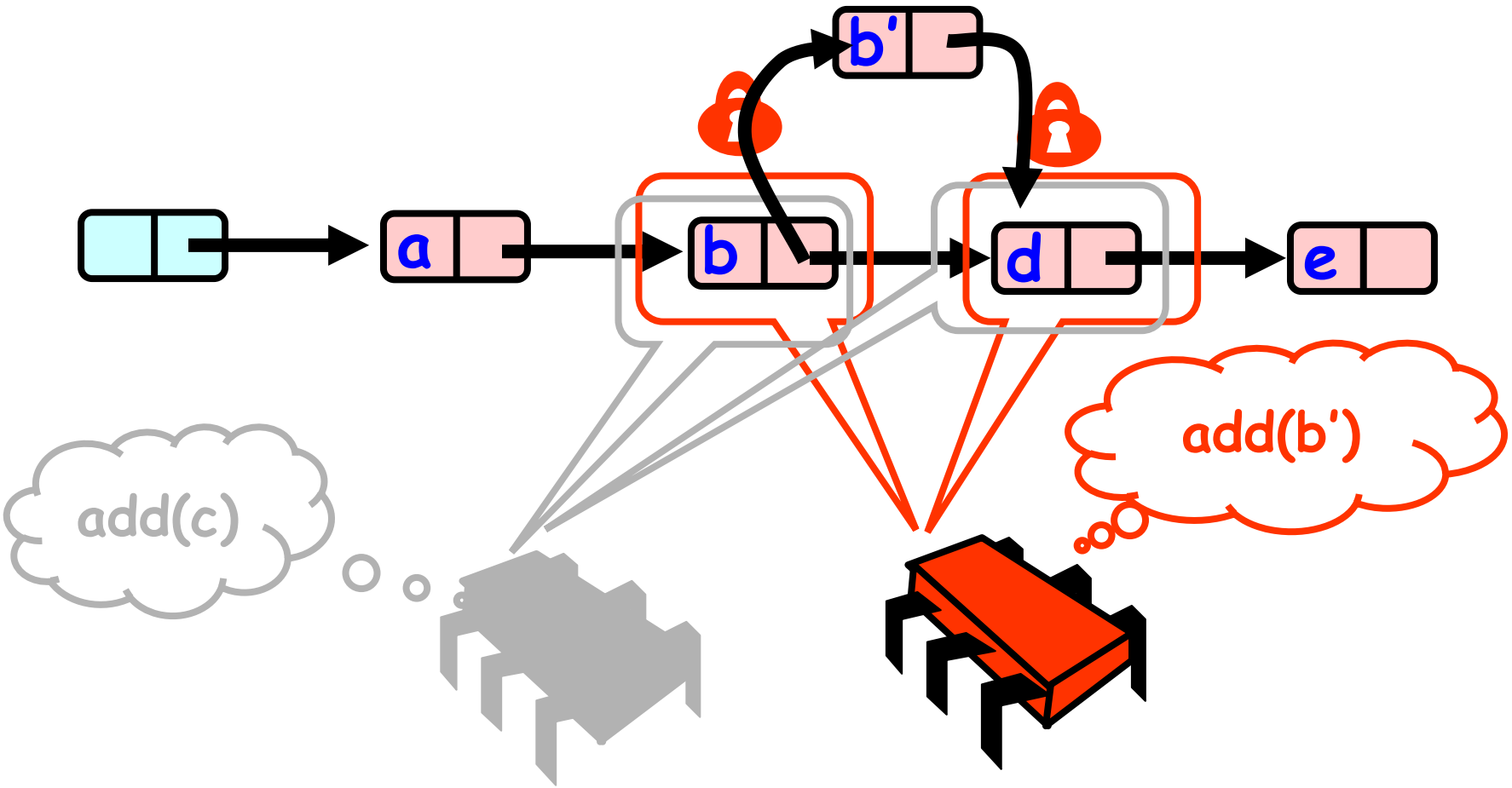
Validate (1)



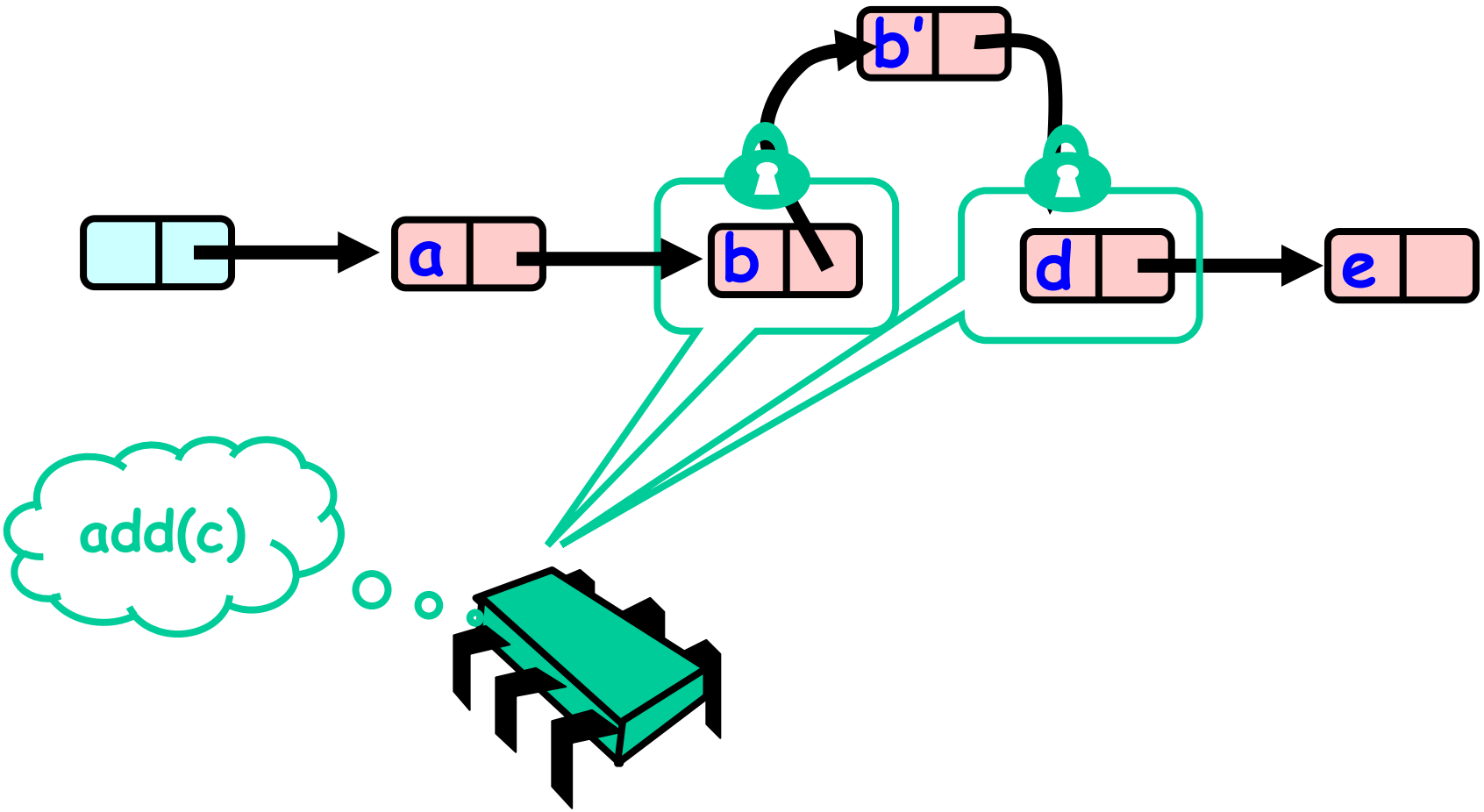
What Else Can Go Wrong?



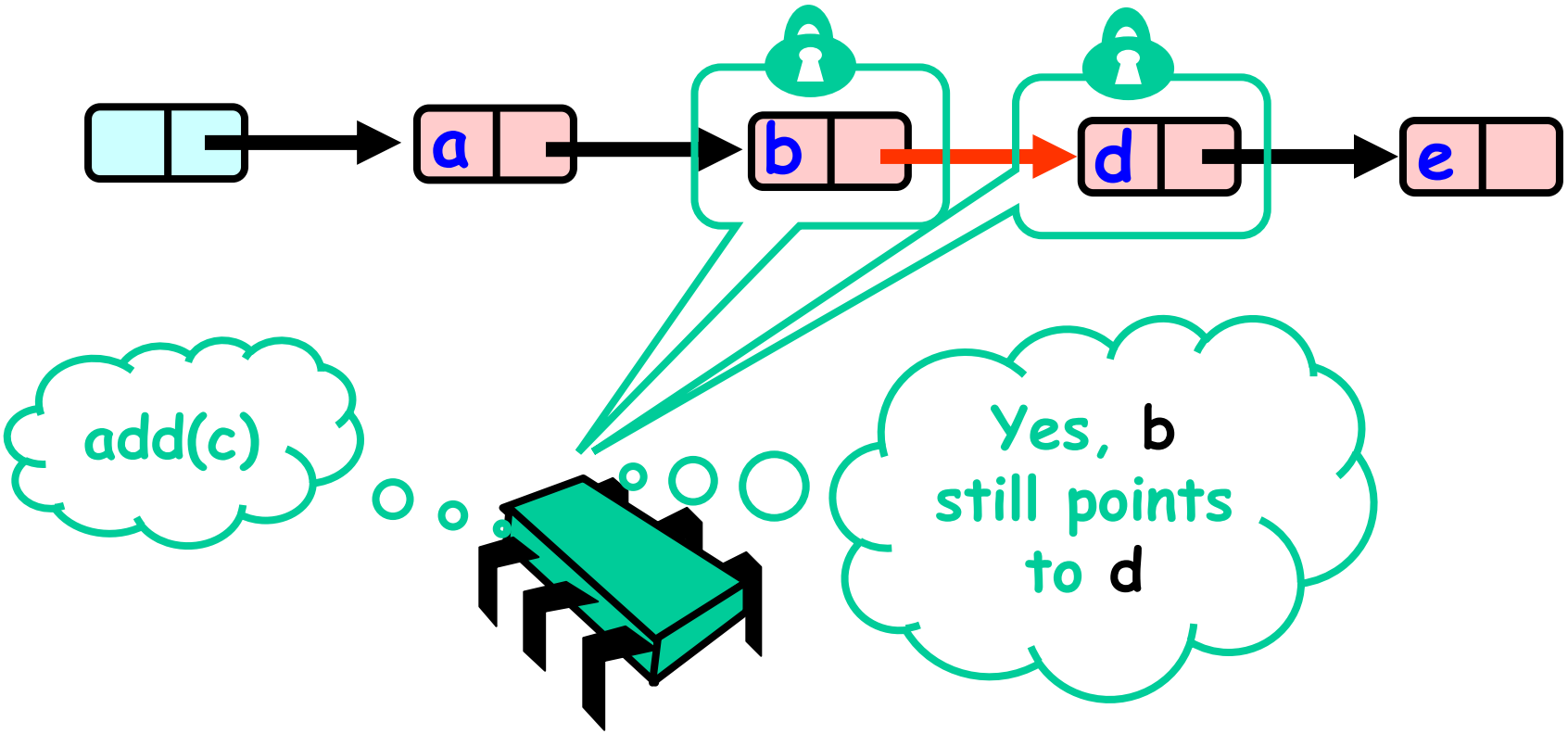
What Else Can Go Wrong?



What Else Can Go Wrong?



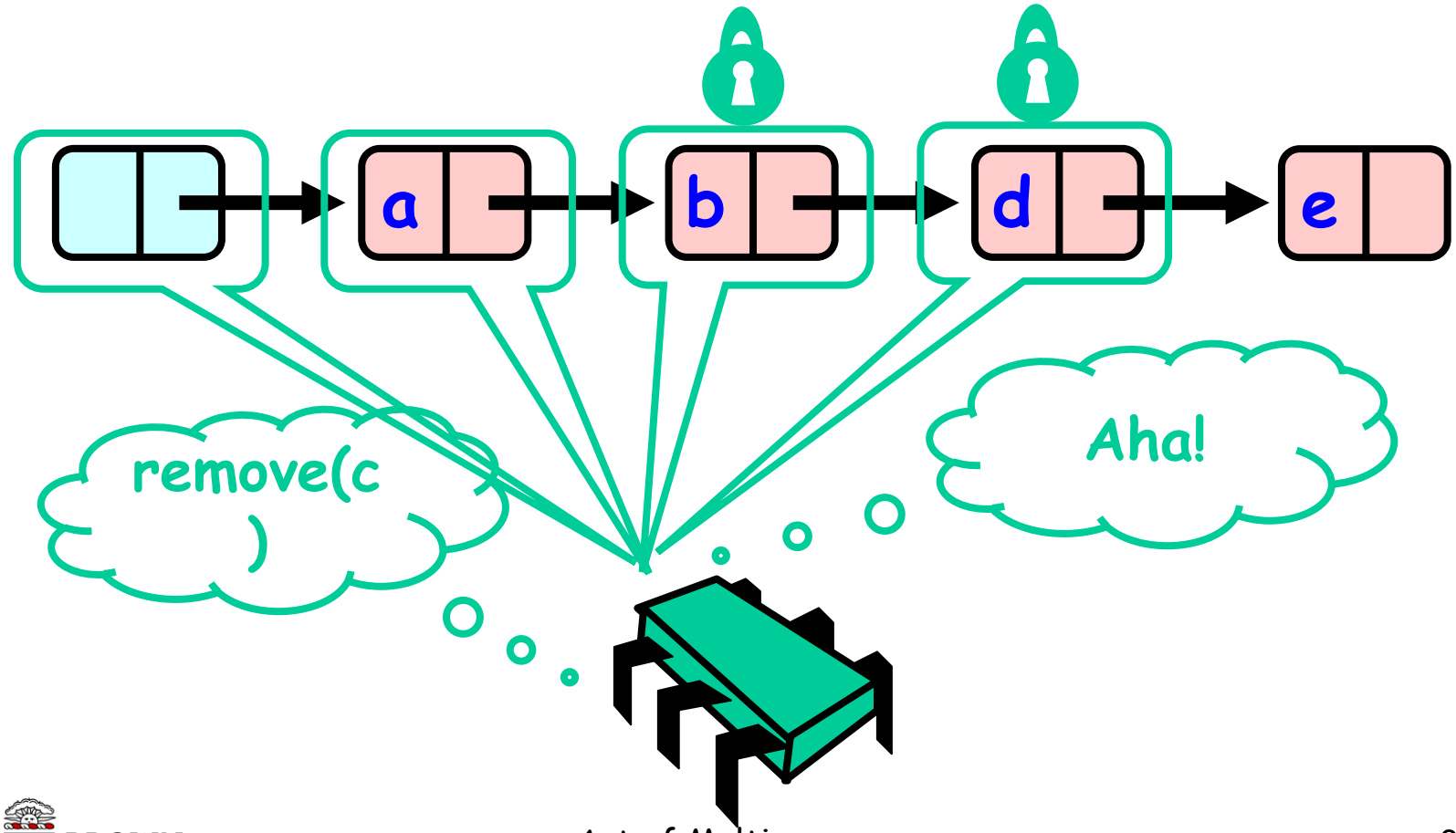
Optimistic: Validate(2)



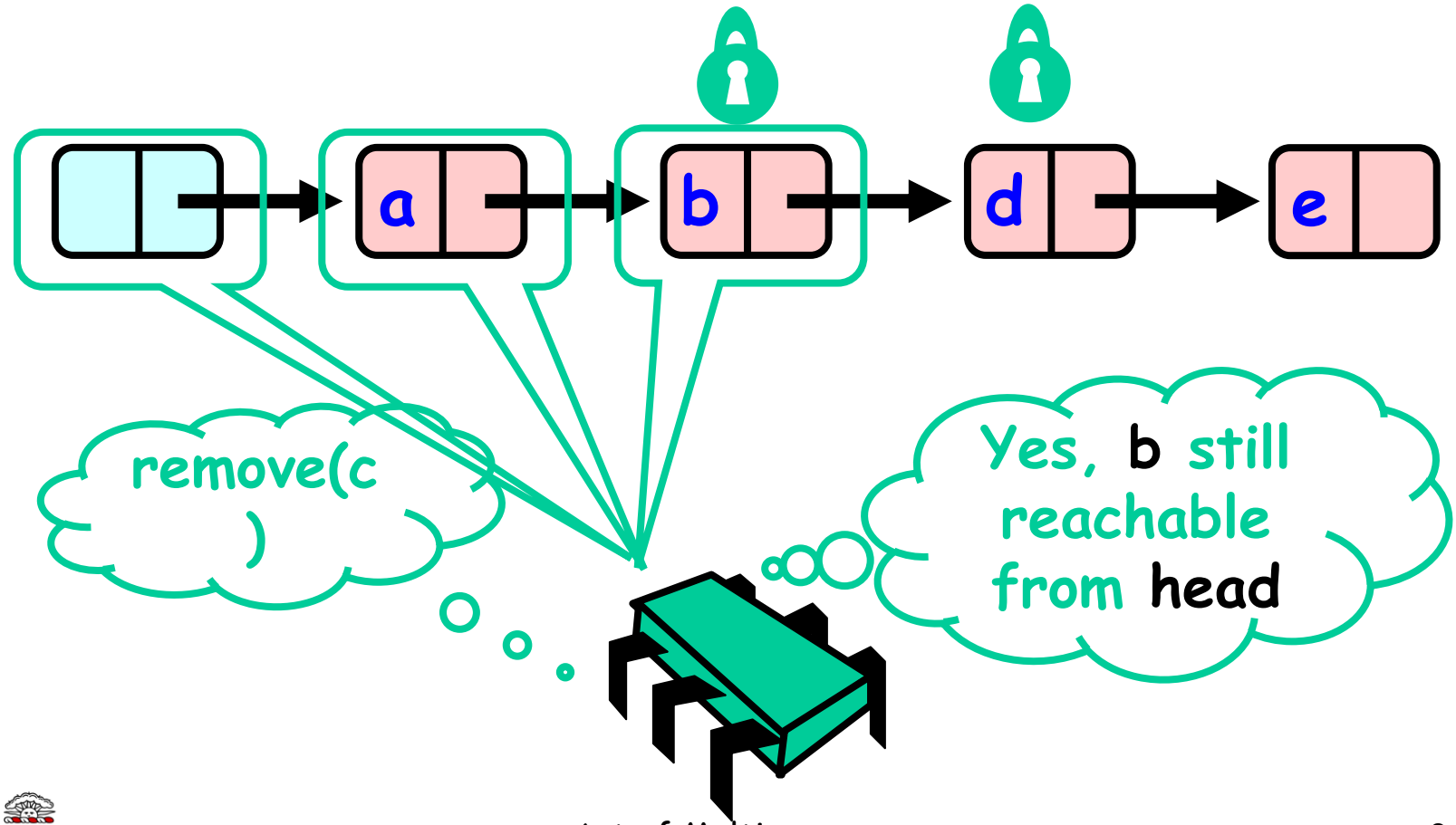
Invariants

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

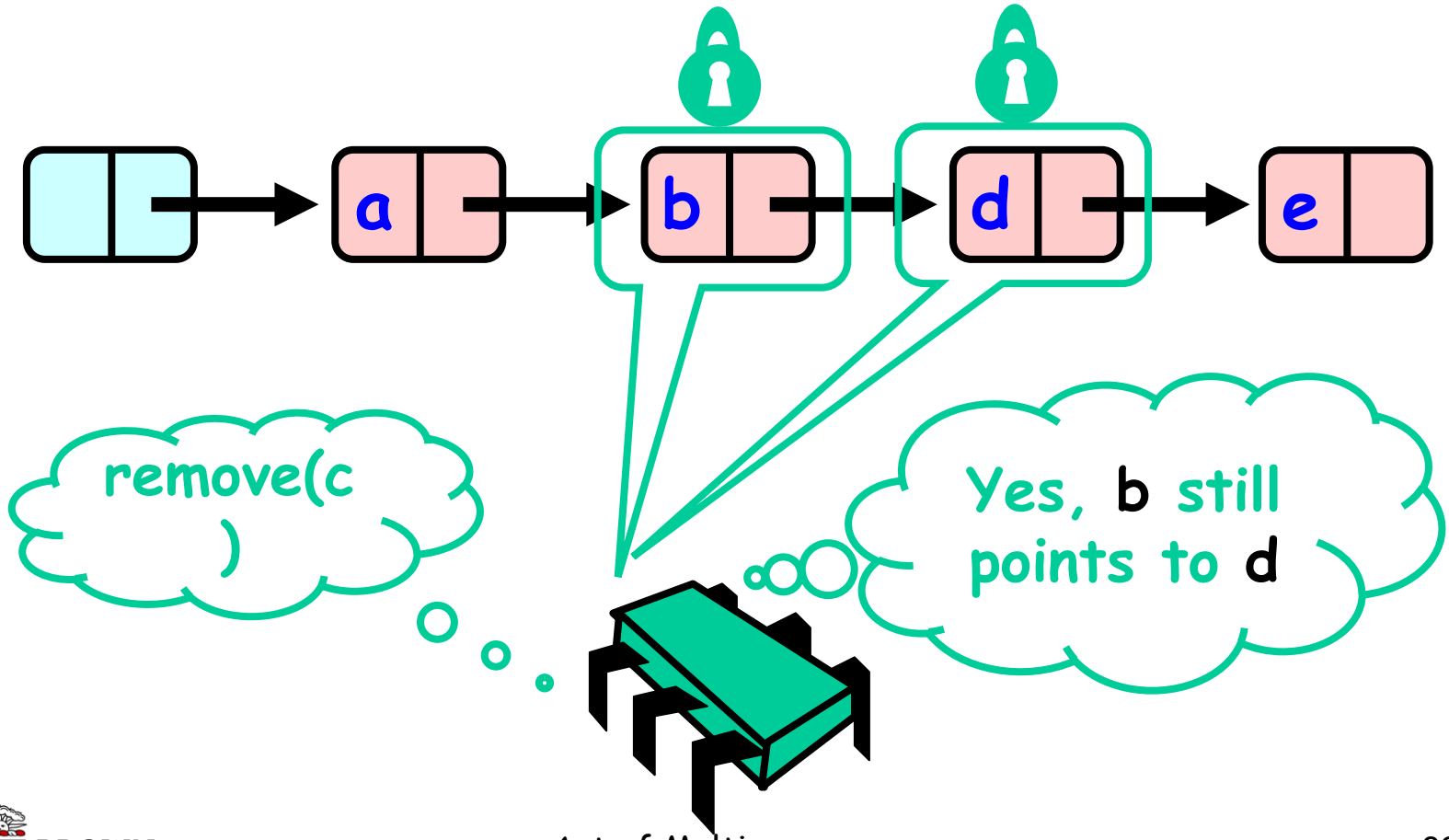
Removing an Absent Node



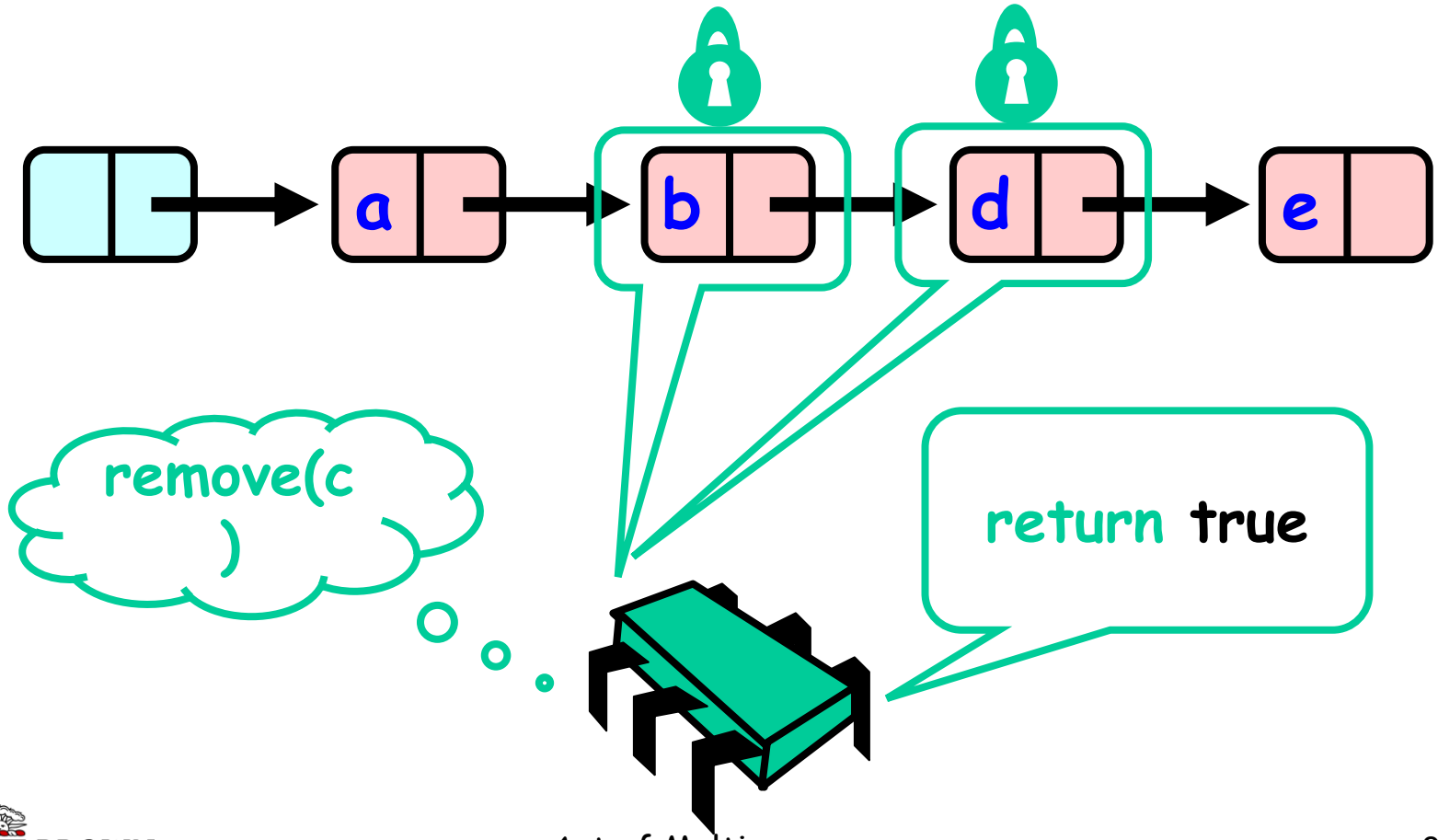
Validate (1)



Validate (2)



OK Computer



Optimistic List

- Limited hot-spots
  - Targets of add(), remove(), contains()
  - No contention on traversals
    - Traversals are **wait-free**

So Far, So Good

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

Evaluation

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

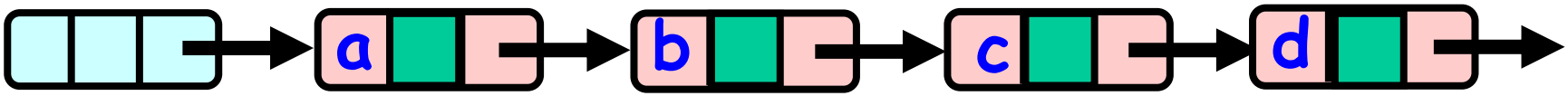
Lazy List

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

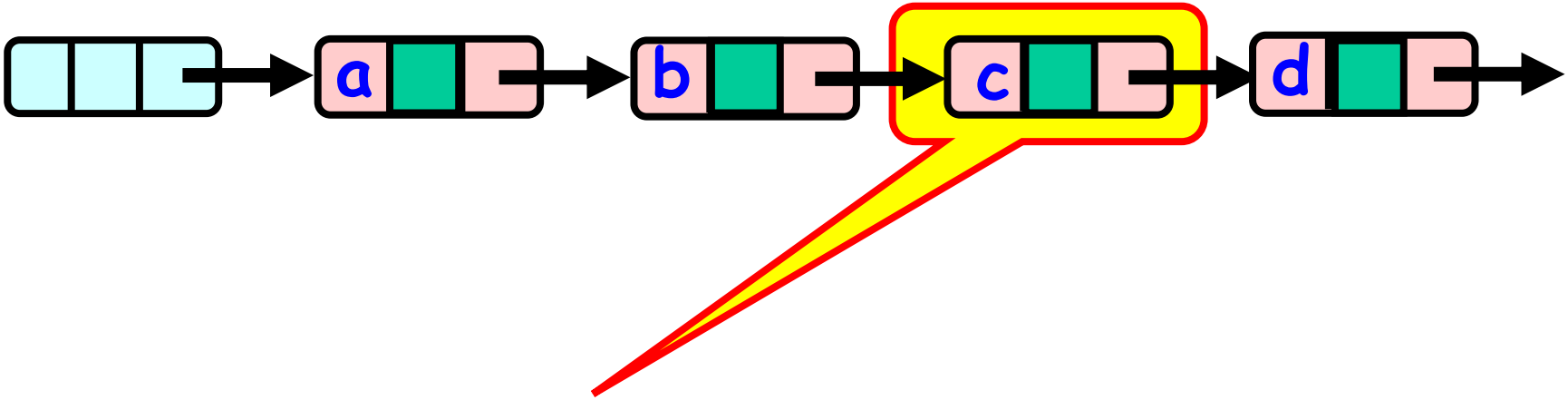
Lazy List

- **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)

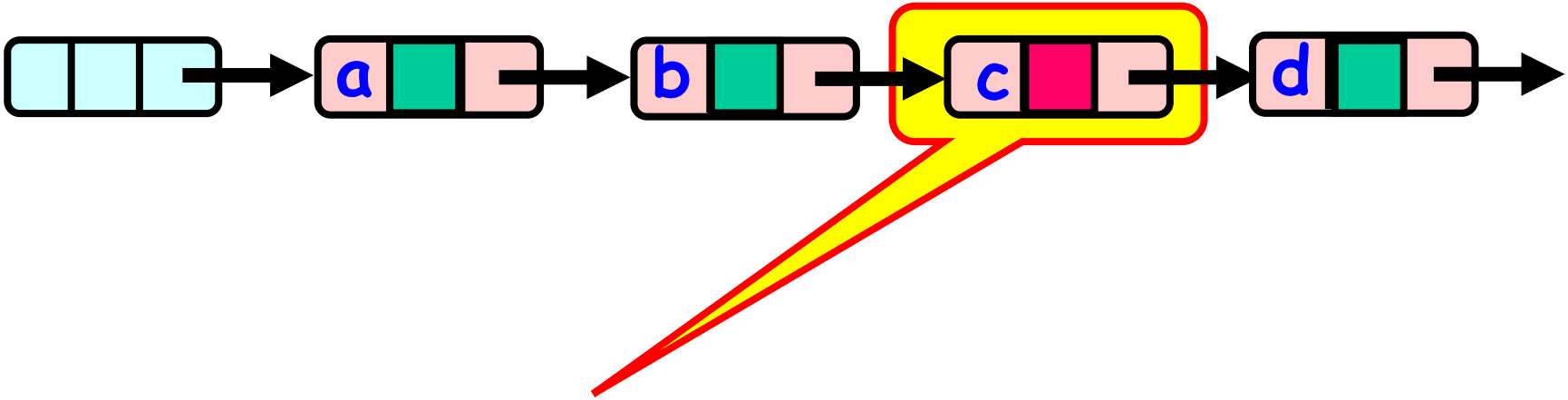
Lazy Removal



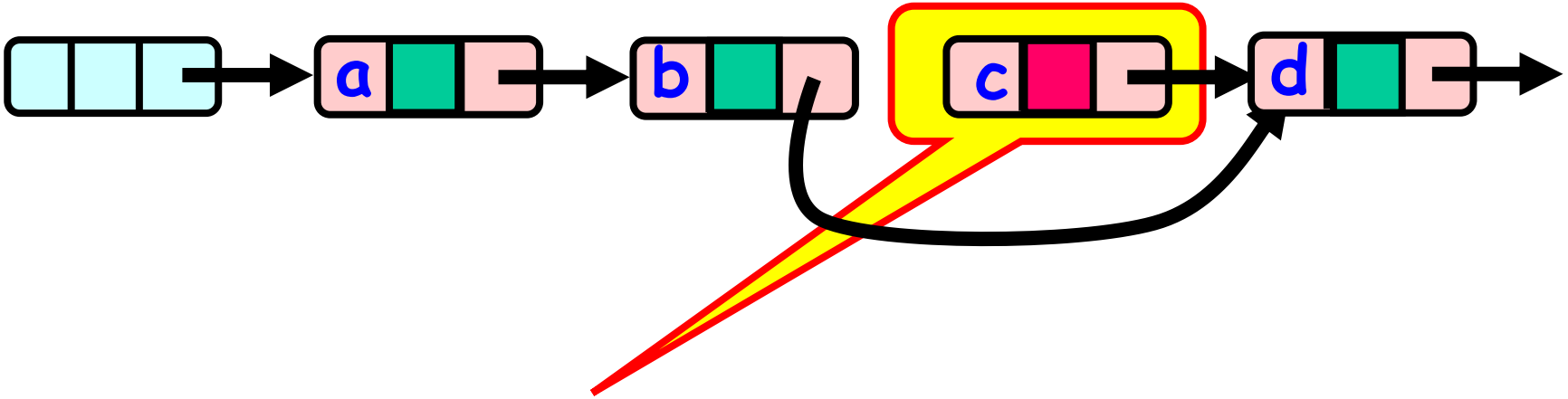
Lazy Removal



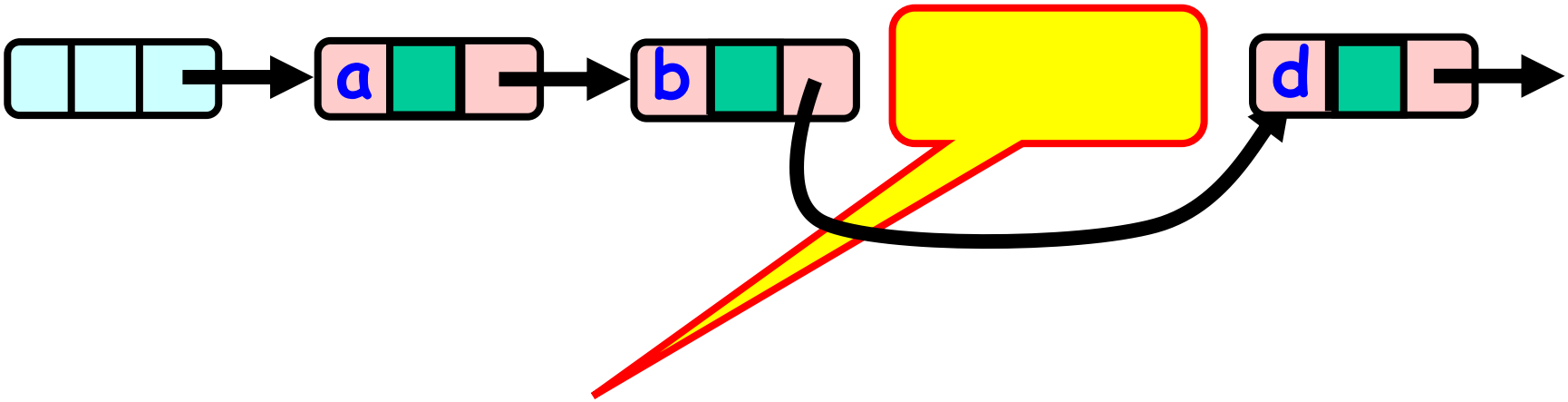
Lazy Removal



Lazy Removal



Lazy Removal



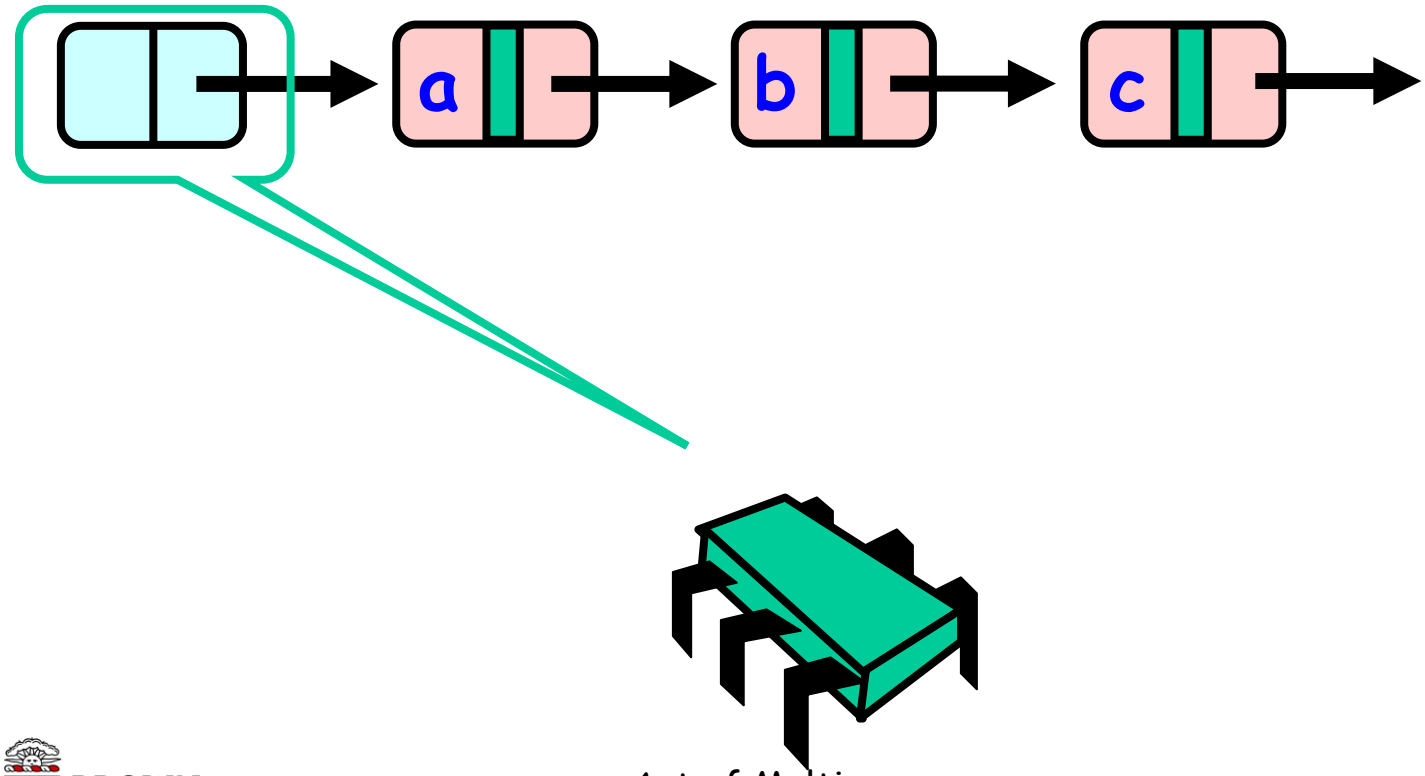
Lazy List

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

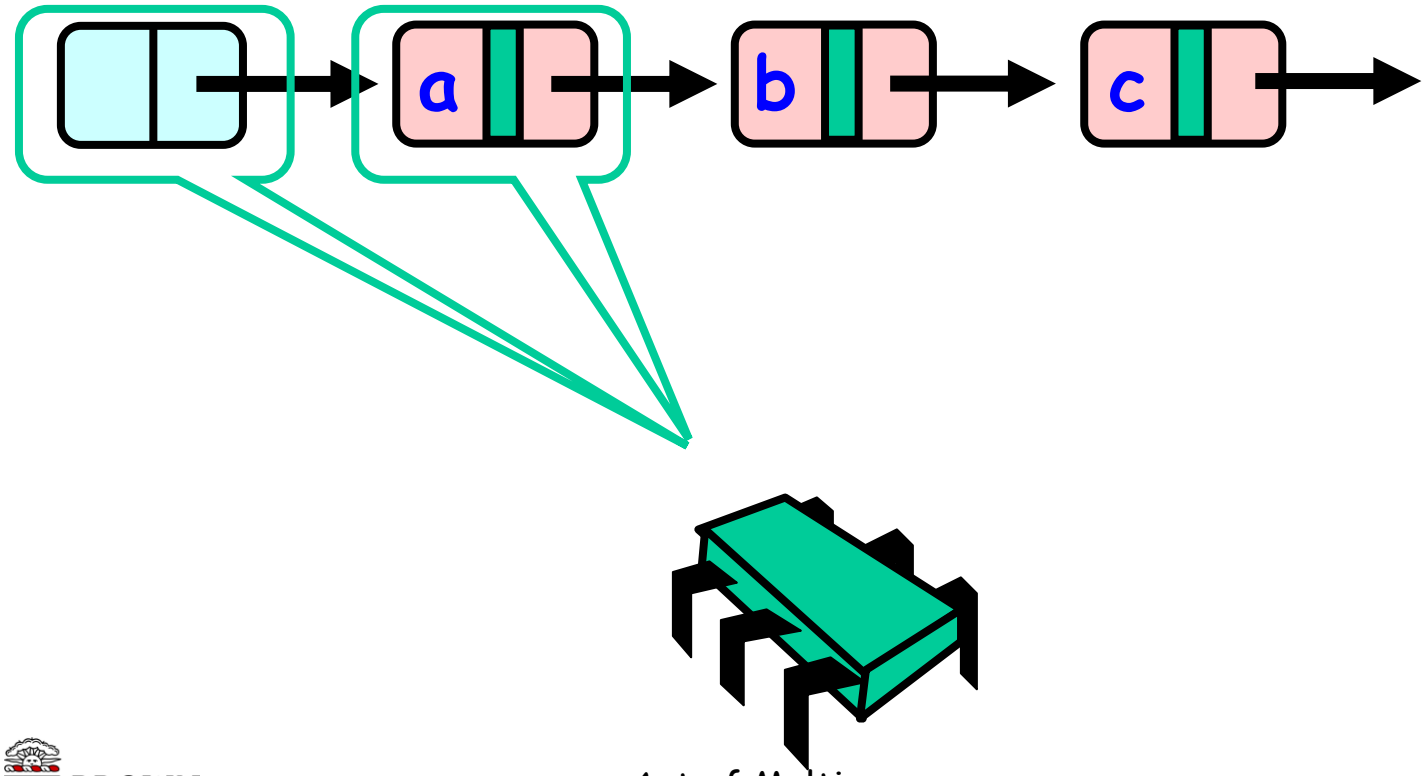
Validation

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

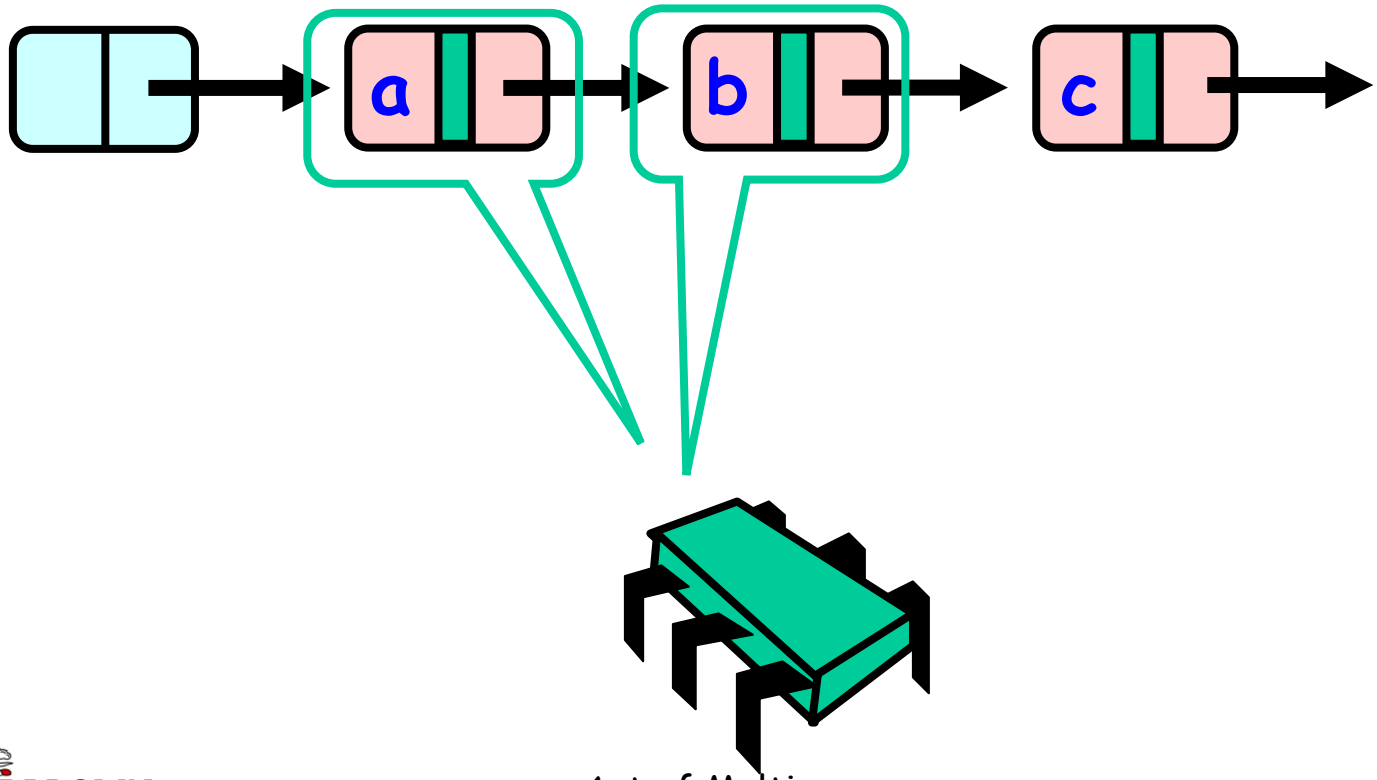
Business as Usual



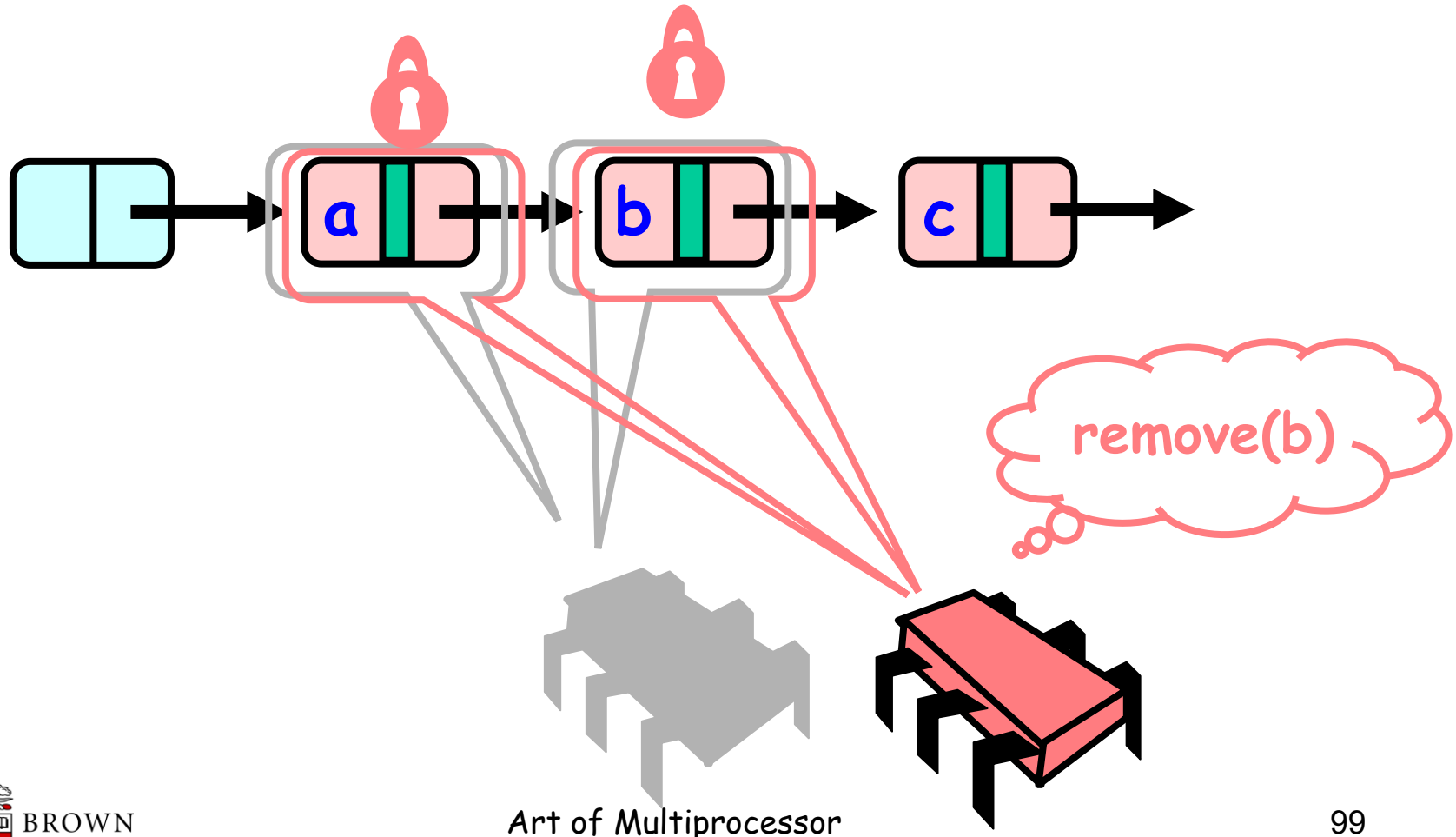
Business as Usual



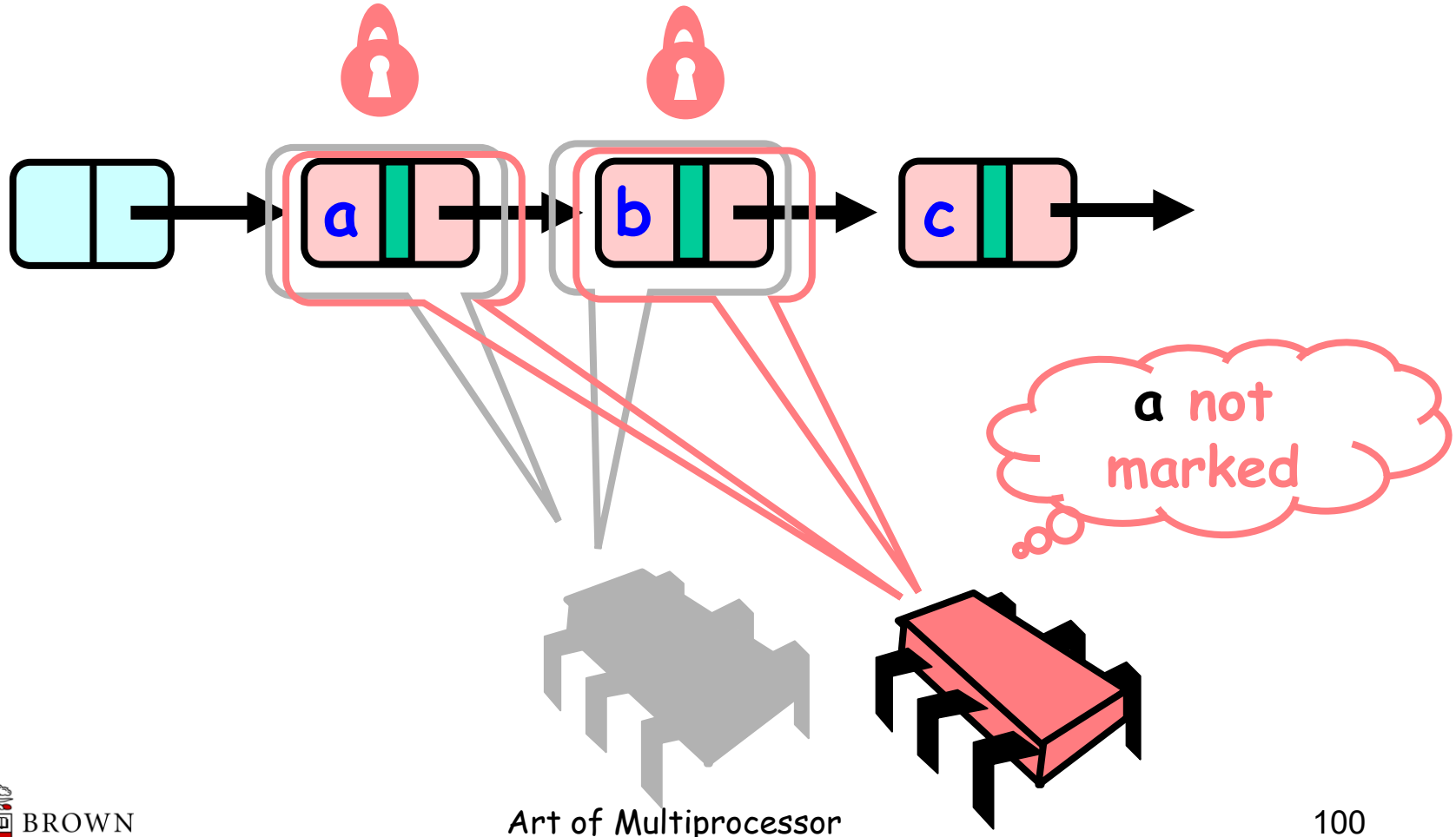
Business as Usual



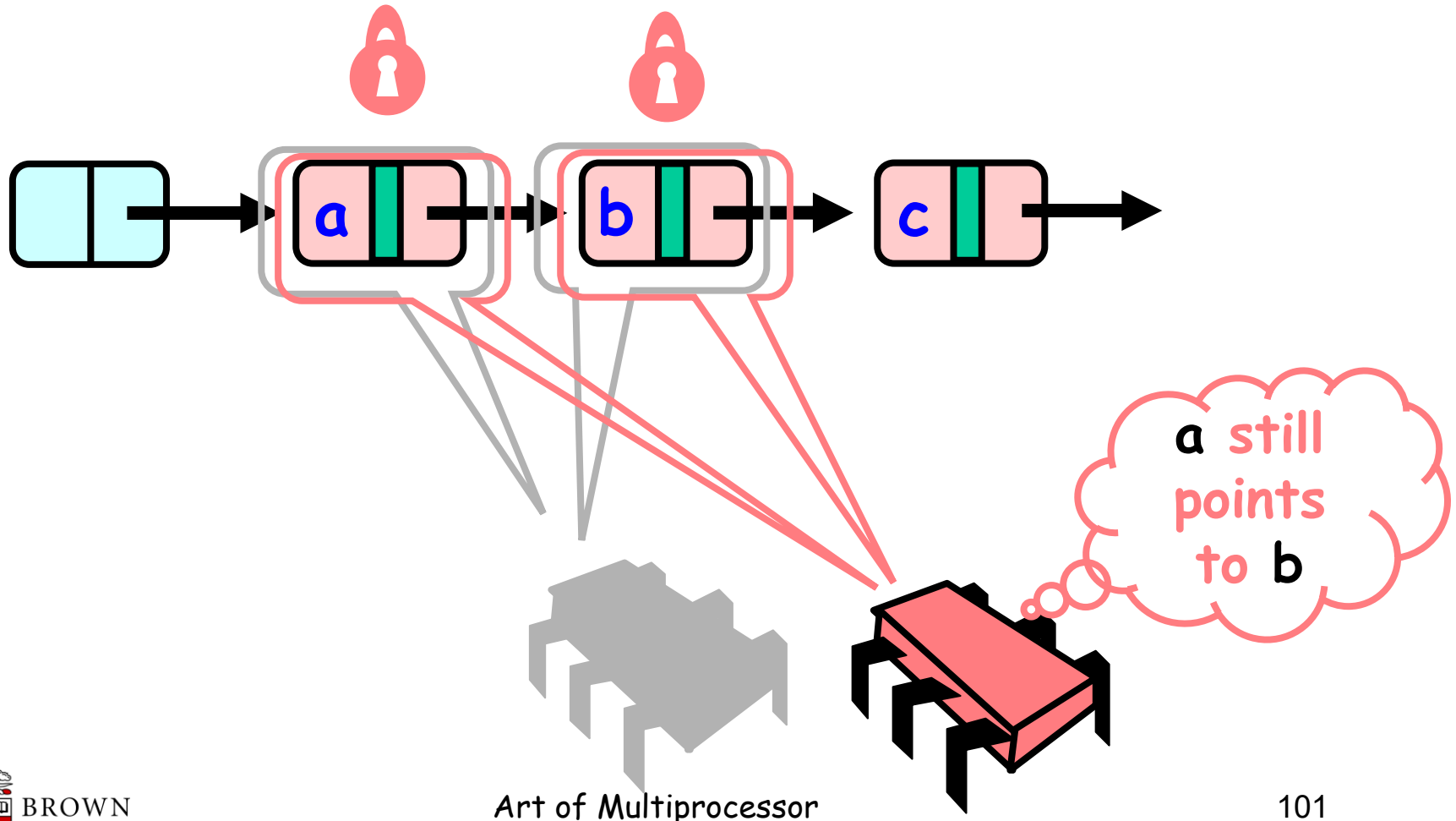
Business as Usual



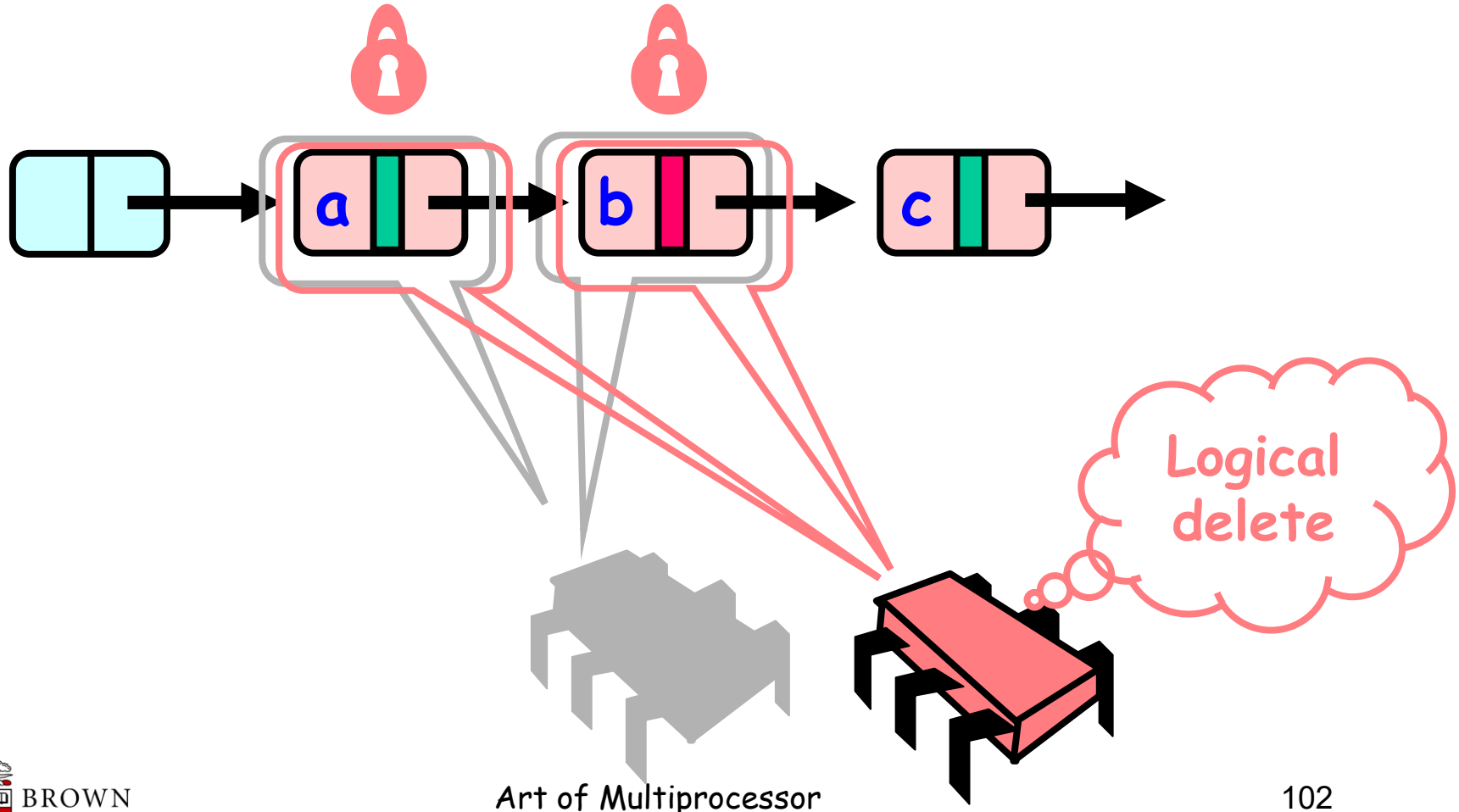
Business as Usual



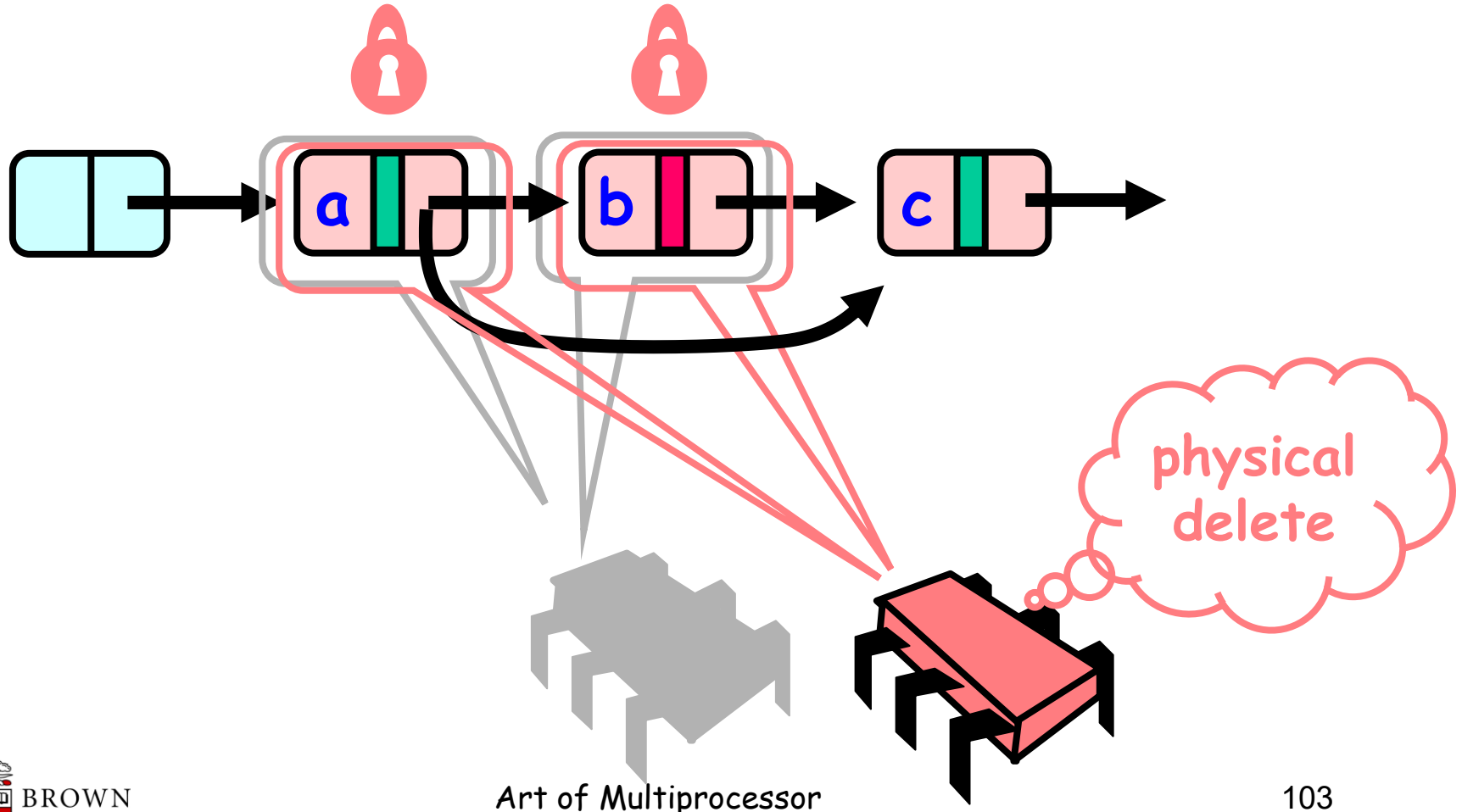
Business as Usual



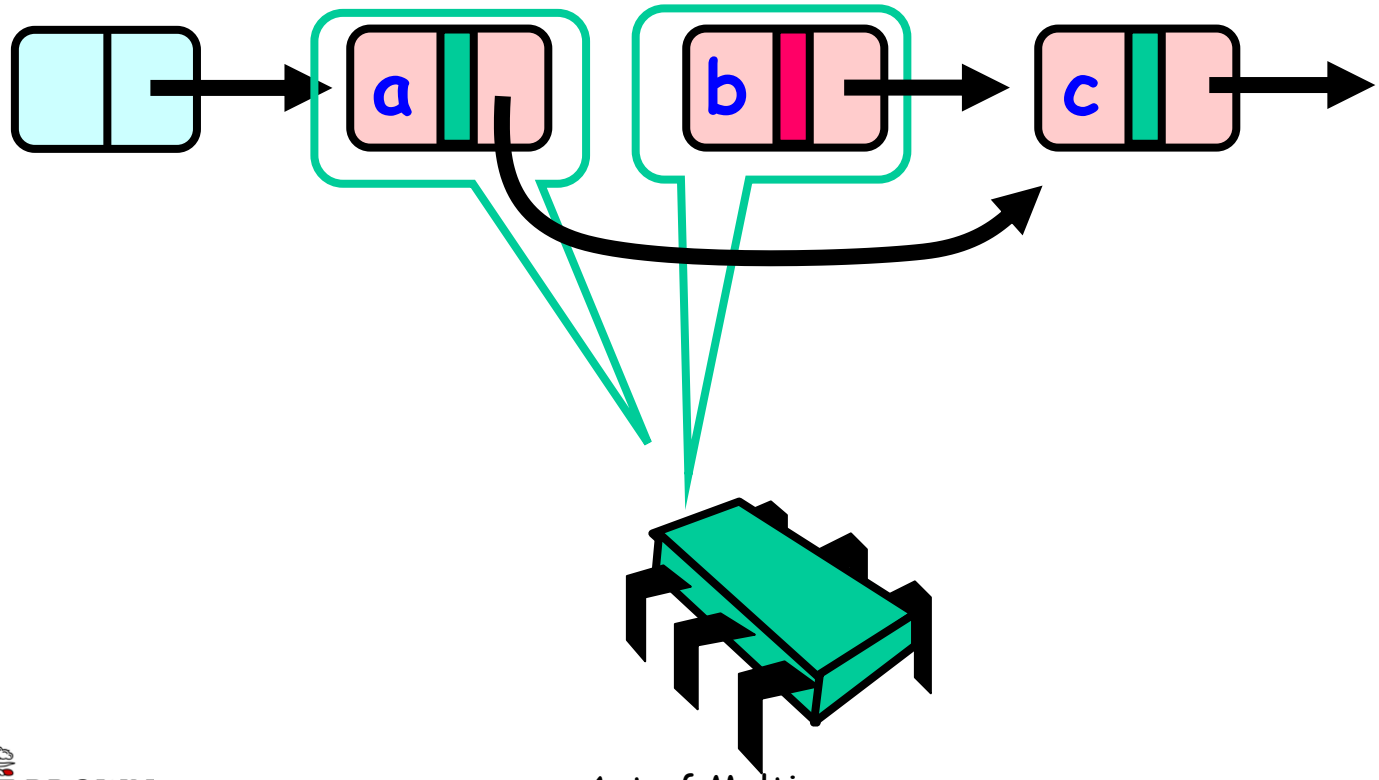
Business as Usual



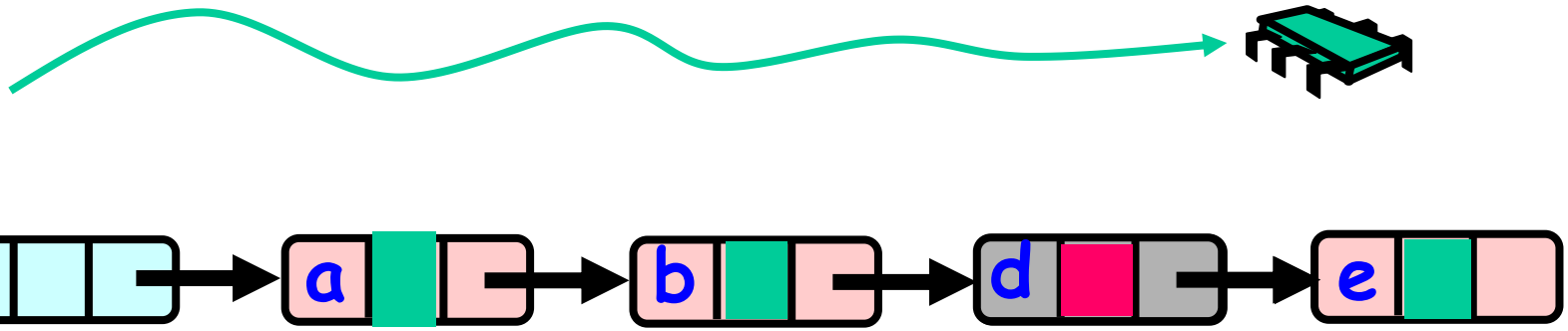
Business as Usual



Business as Usual



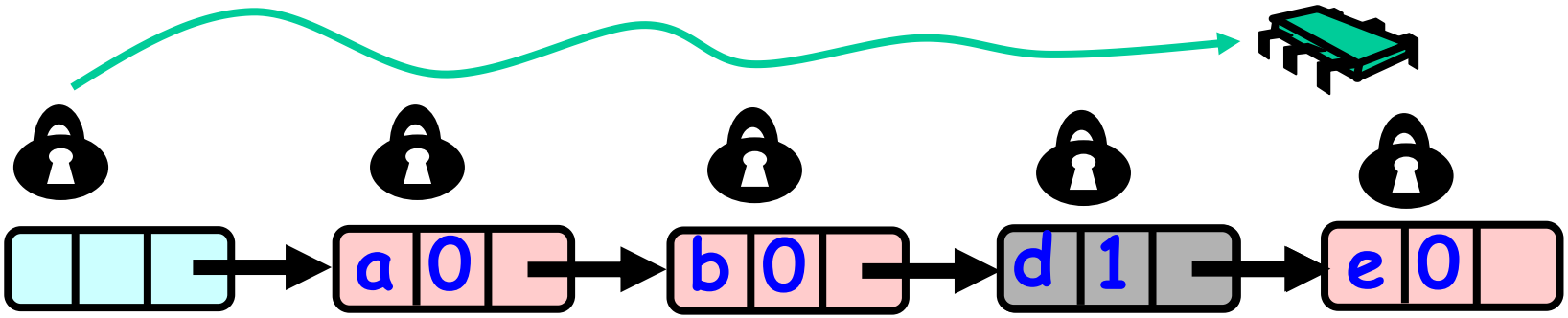
Wait-free Contains – No need to lock in Contains



Use Mark bit + Fact that List is ordered

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

Lazy List



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

Evaluation

- Good:
  - contains() doesn't lock
  - In fact, its wait-free!
  - Good because typically high % contains()
  - Uncontended calls don't re-traverse
- Bad
  - Contended 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 …
    - Software error, …
  - Everyone else using that lock is stuck!

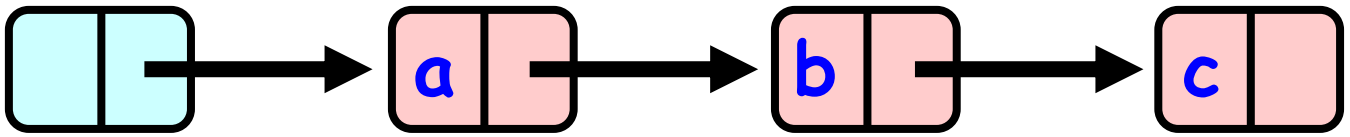
Lock-Free Data Structures

- No matter what …
  - Some thread will complete method call
  - Even if others halt at malicious times
  - Weaker than wait-free, yet
- Implies that
  - You can't use locks (why?)
  - Um, that's why they call it lock-free

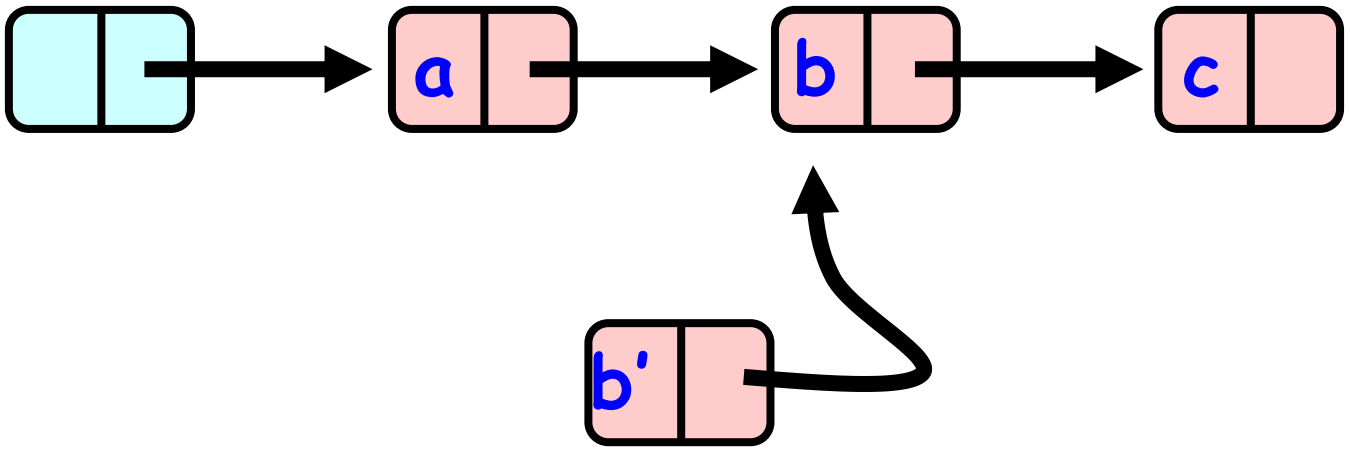
Lock-free Lists

- Next logical step
- Eliminate locking entirely
- contains() wait-free and add() and remove() lock-free
- Use only compareAndSet()
- What could go wrong?

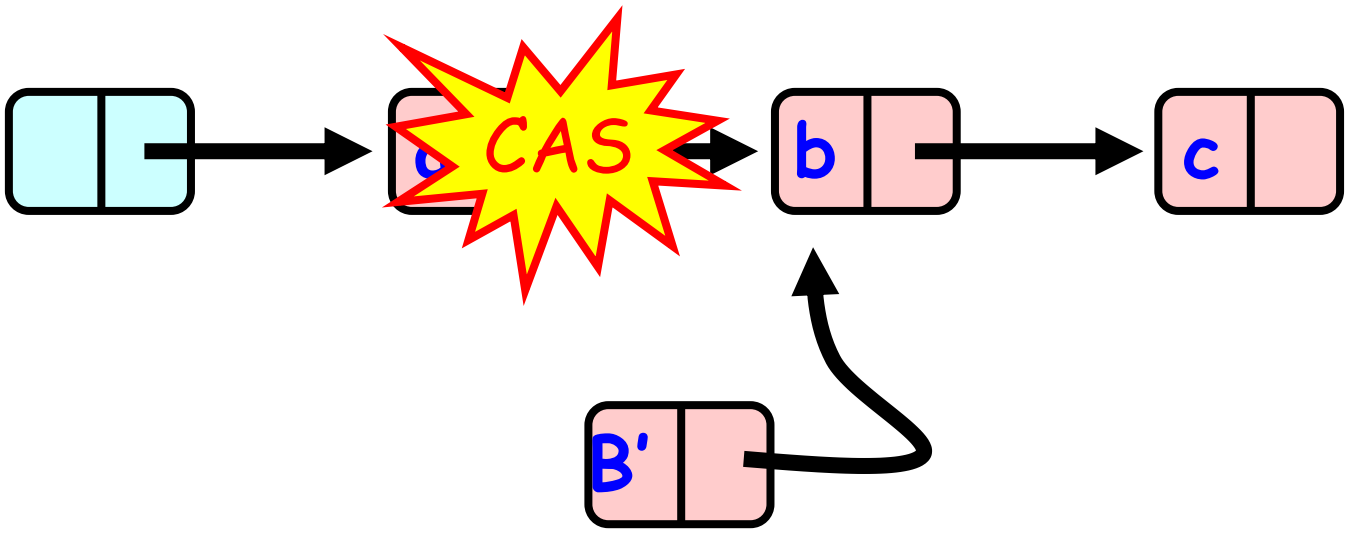
Adding a Node



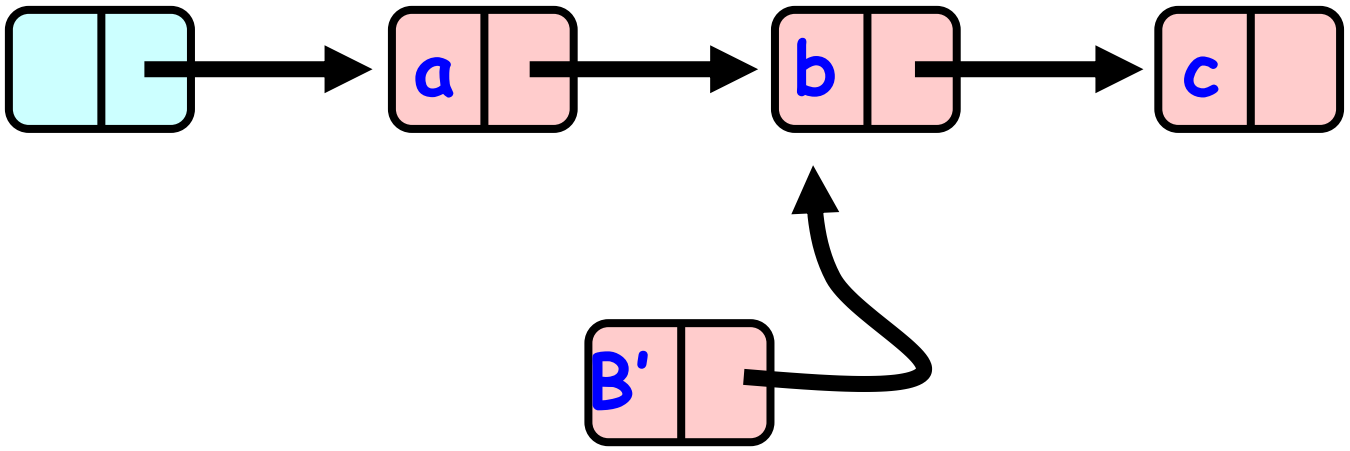
Adding a Node



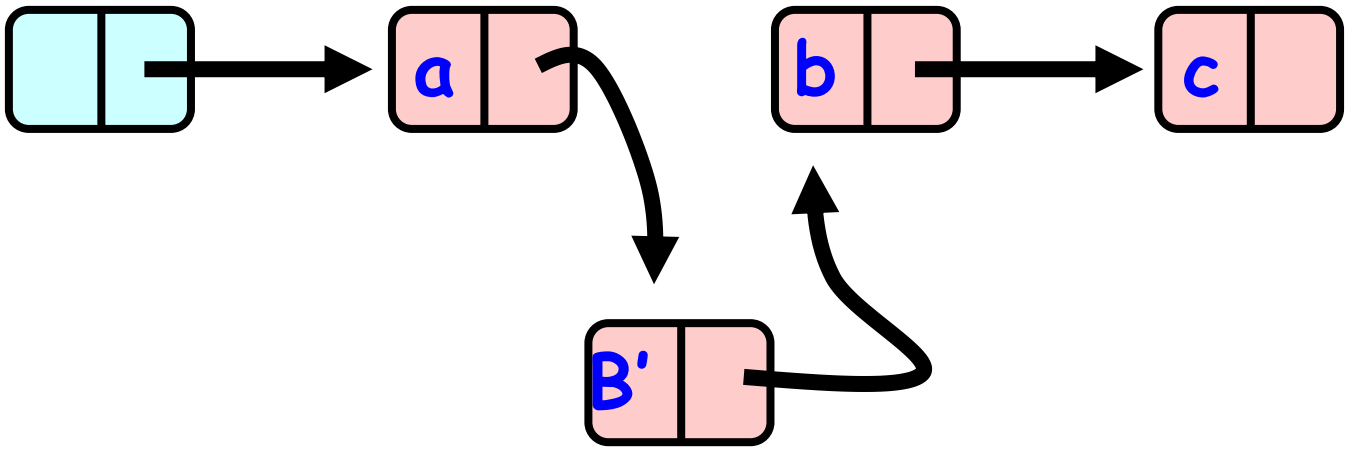
Adding a Node



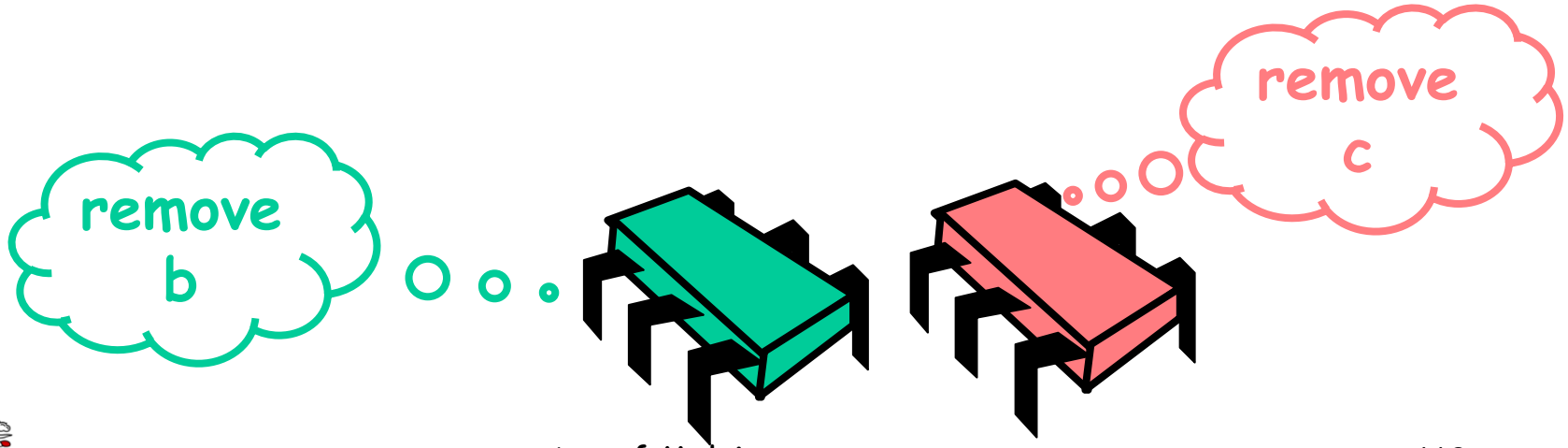
Adding a Node



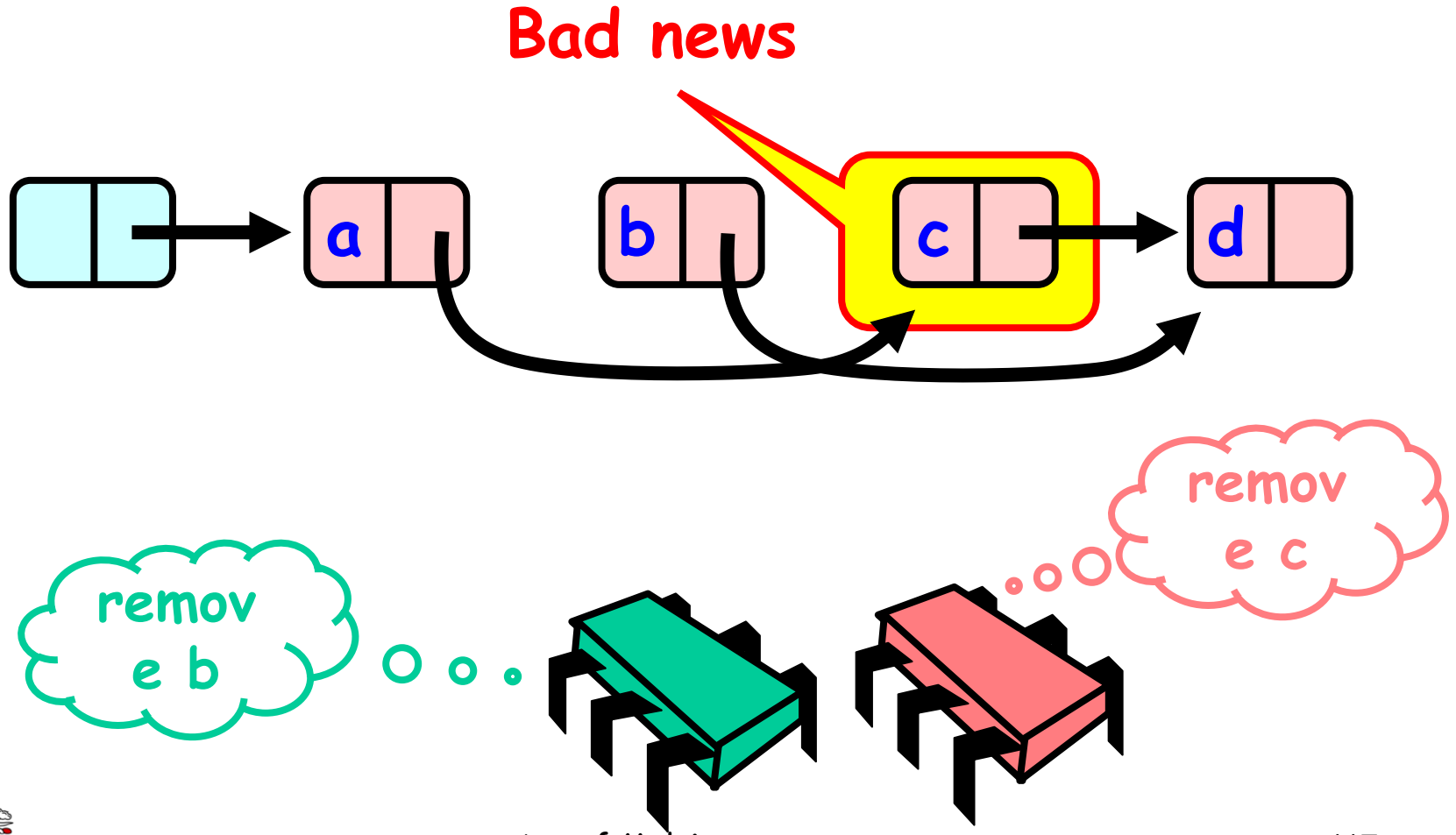
Adding a Node



Removing a Node



Look Familiar?



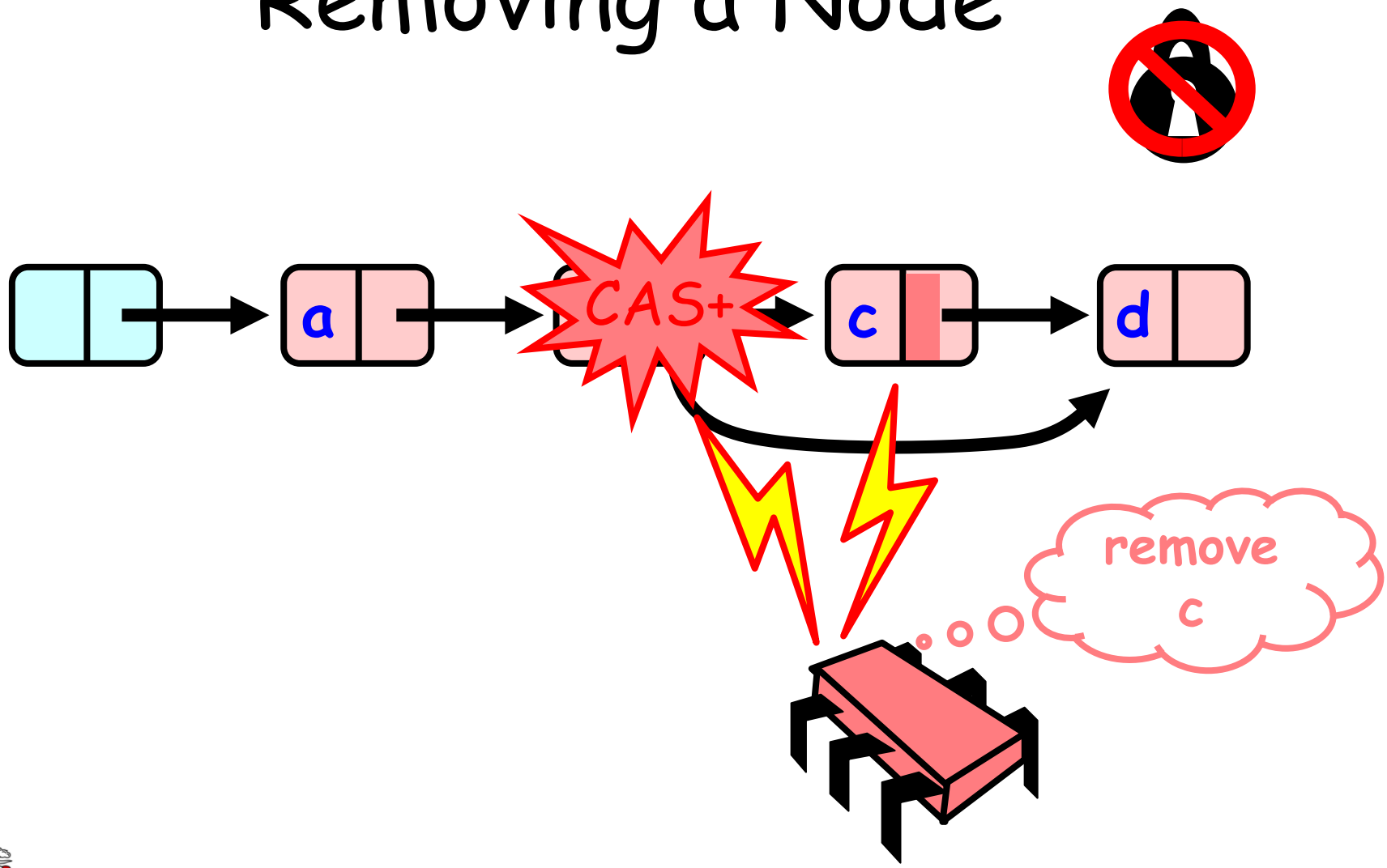
Problem

- Method updates node's next field
- After node has been removed

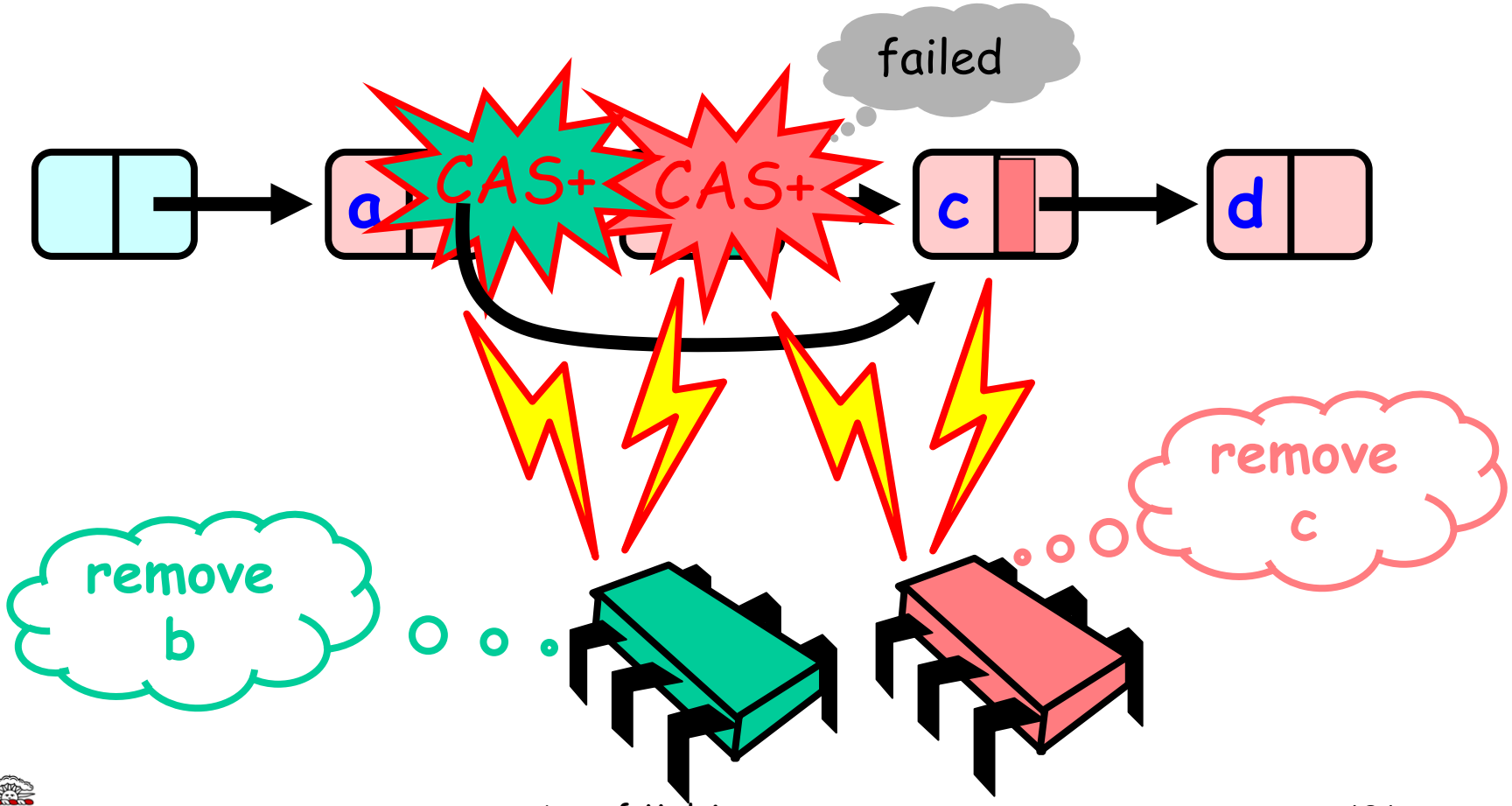
Solution

- Use AtomicMarkableReference (CAS+)
- CAS+: Atomically
  - Check mark bit in next field is unset
  - Check reference unchanged
  - If both true: Swing reference
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor's pointer using CAS+

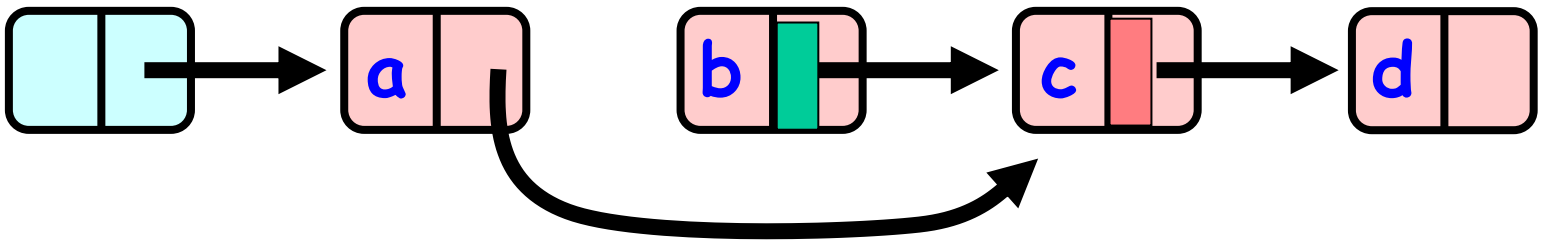
Removing a Node

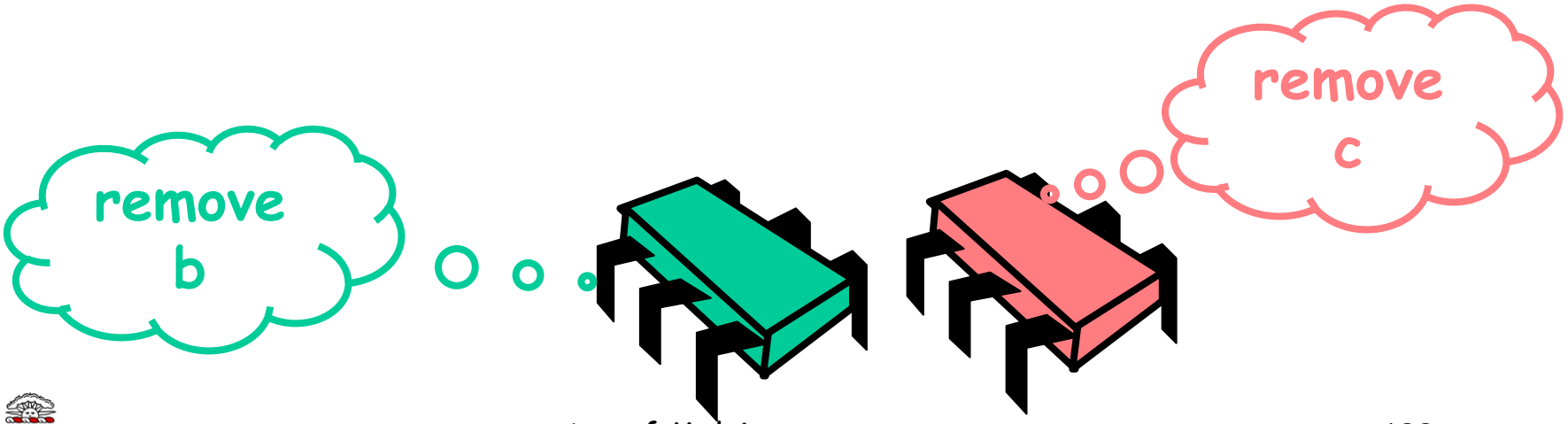


Removing a Node

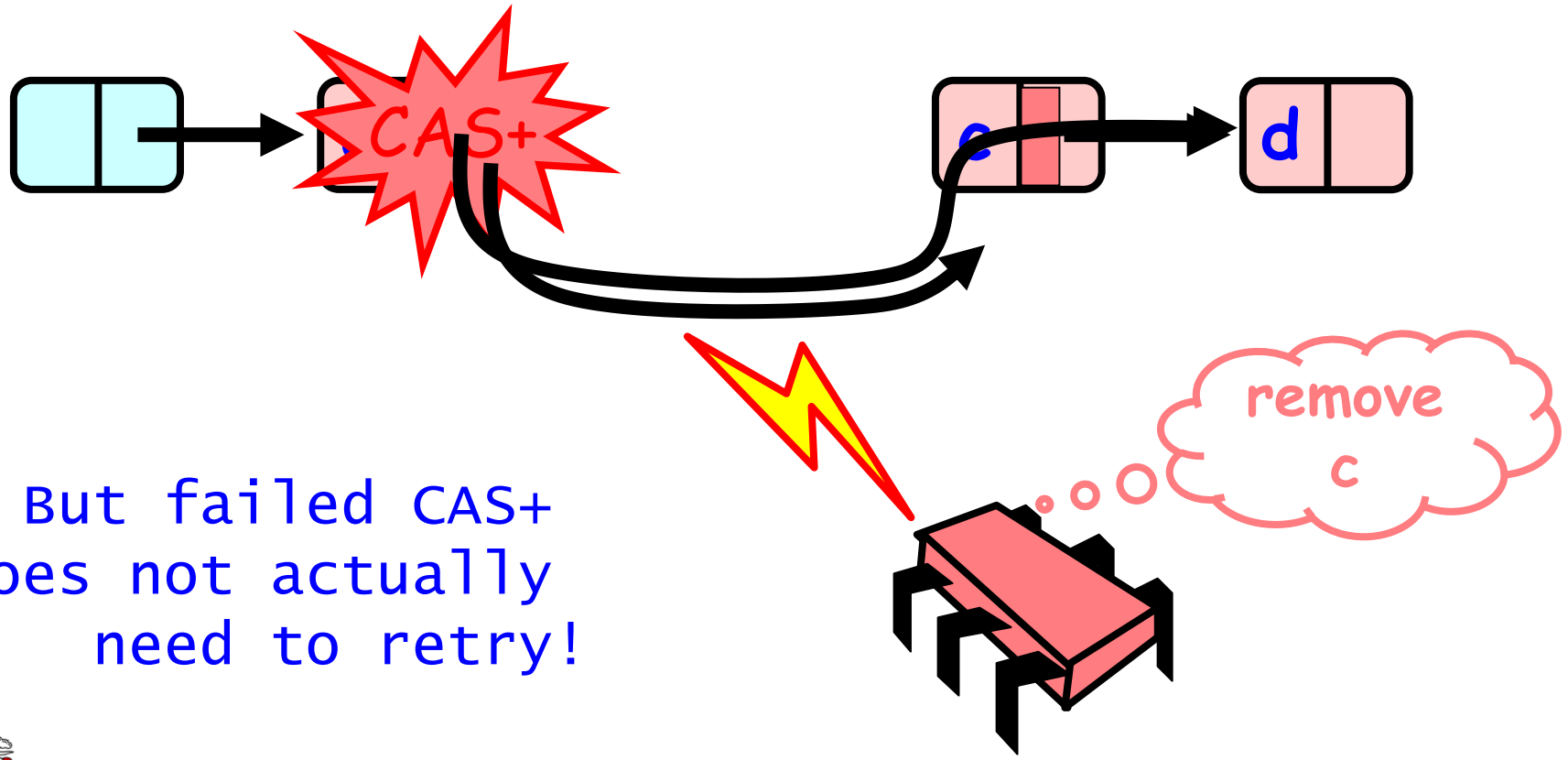


Removing a Node





Removing a Node

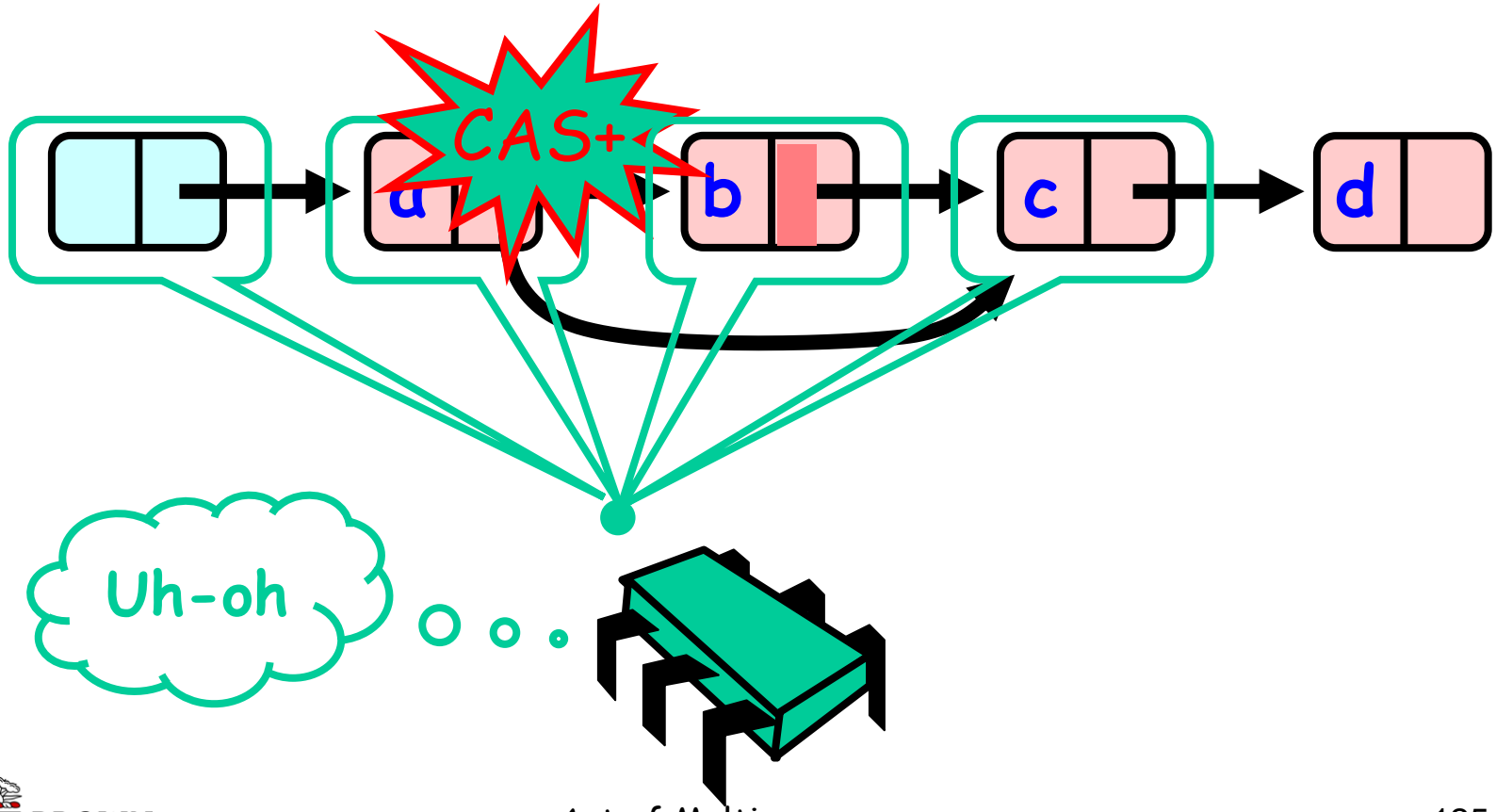


But failed CAS+ does not actually need to retry!

Traversing the List

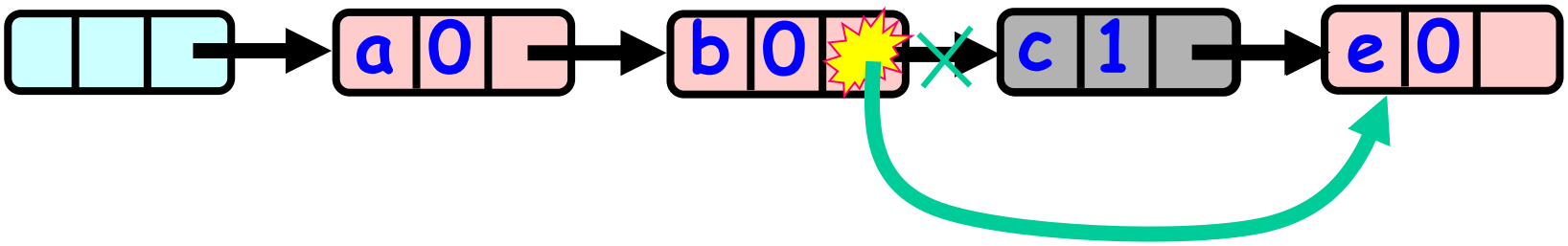
- Q: what do you do when you find a "logically" deleted node in your path?
- A: finish the job.
  - CAS+ the predecessor's next field
  - Proceed (repeat as needed)

Lock-Free Traversal



Lock-free Removal

Logical Removal =
Set Mark Bit

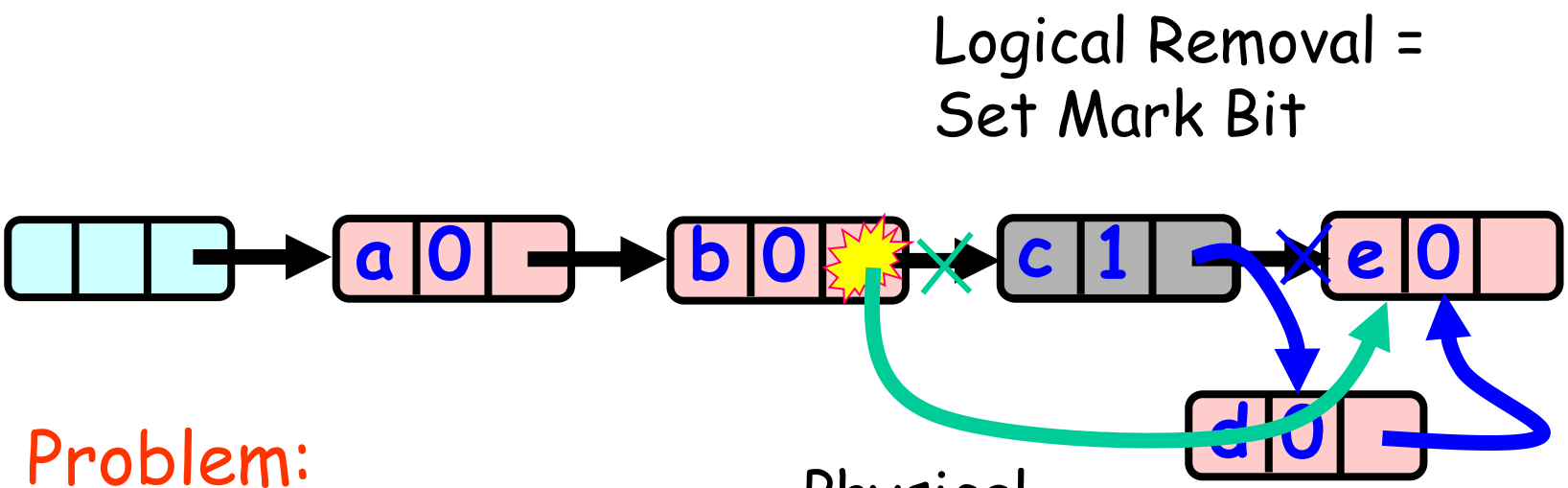


Use CAS+ to verify pointer is correct

Is it enough?

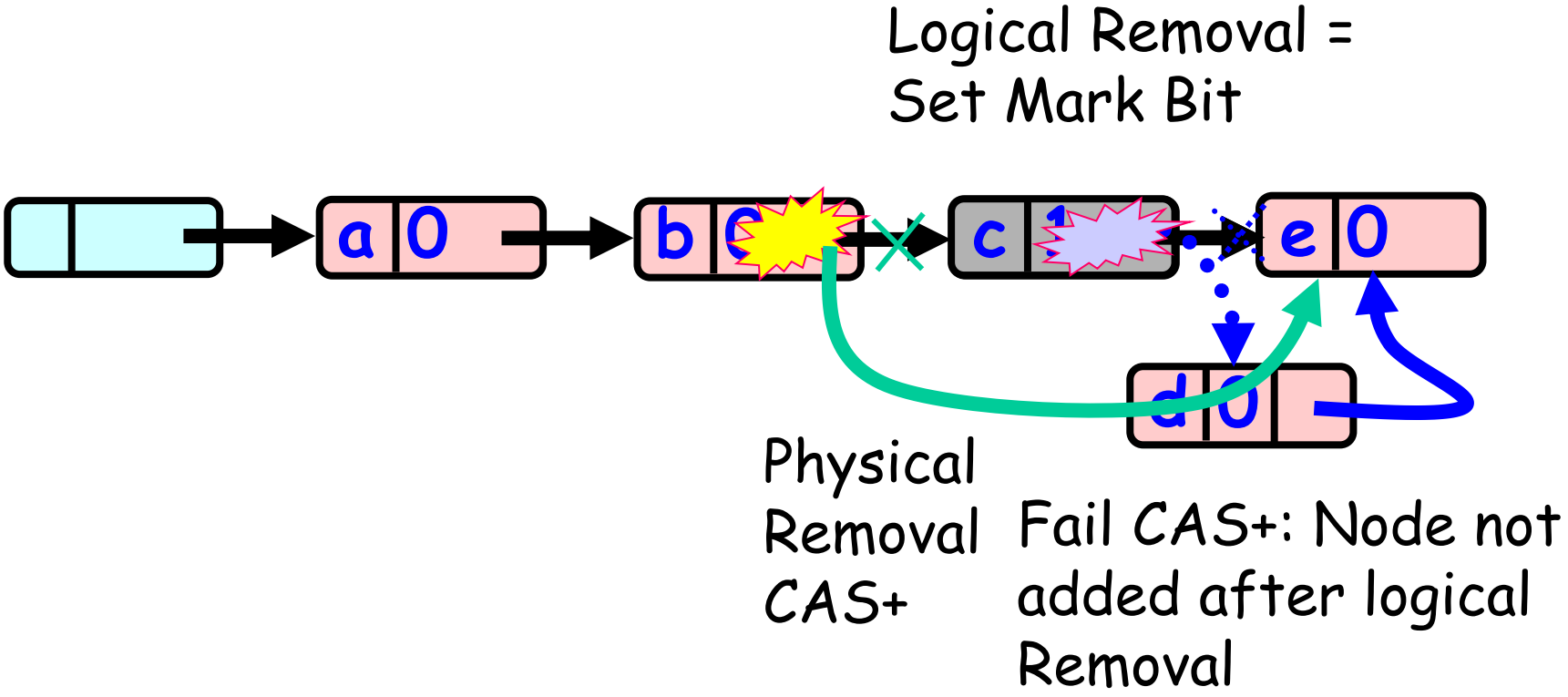
Physical Removal
CAS+ pointer

Lock-free Removal with add

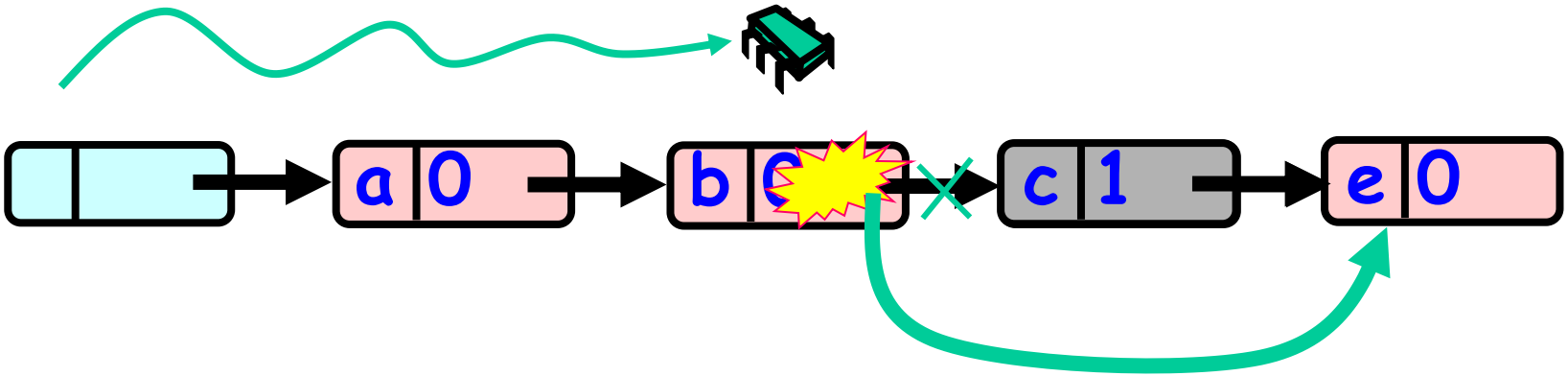


Problem:
d not added to list…
Must Prevent
manipulation of
removed node's pointer

Solution: use CAS+ in ADD()



Summary: A Lock-free Algorithm



1. add() and remove() physically remove marked nodes
2. Wait-free contains() traverses both marked and removed nodes