6.852 Lecture 21

- Techniques for highly concurrent objects
  - coarse-grained mutual exclusion
  - read/write locking
  - fine-grained locking (mutex and read/write)
  - optimistic locking
  - lock-free/nonblocking algorithms
  - “lazy” synchronization
  - illustrate on list-based sets, apply to other data structures
- Reading:
  - Herlihy-Shavit Chapter 8 (Chapter 9 in draft version)

Shared-memory algorithms

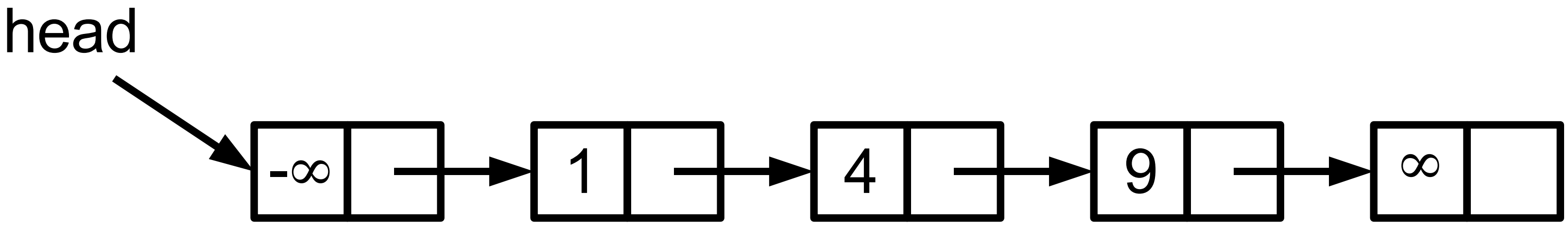
- Object-oriented pseudocode
  - at most one memory access per atomic step
  - memory management: allocation and garbage collection
- Synchronization primitives
  - compare-and-swap (CAS)
  - load-linked/store-conditional (LL/SC)
  - assume lock and unlock methods for every object

Shared-memory algorithms

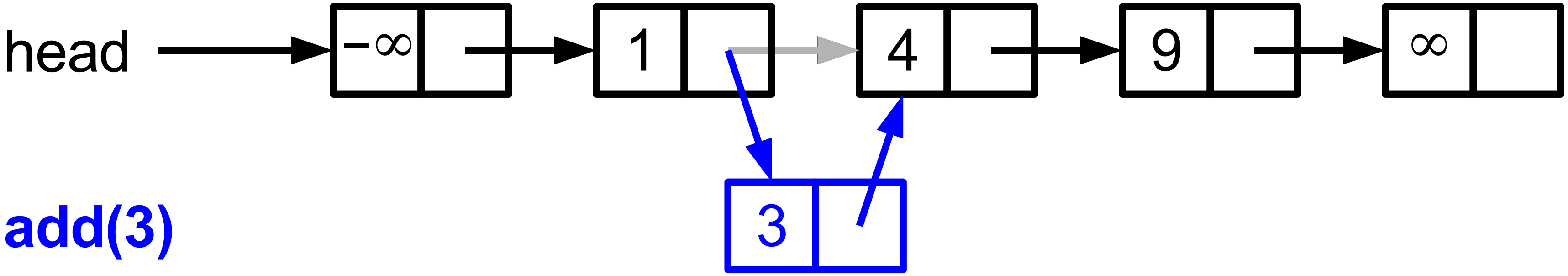
- Atomic (aka linearizable) objects
- Dominant technique: lock-based implementations
- No fault-tolerance (i.e., assume no failures)
  - not even always guaranteed failure-free termination
- Progress properties
  - deadlock-freedom, lockout-freedom (aka starvation-freedom)
  - nonblocking conditions: lock-freedom, wait-freedom
- Performance
  - worst-case (time bounds) vs. average case (throughput)
  - no good formal models

List-based sets

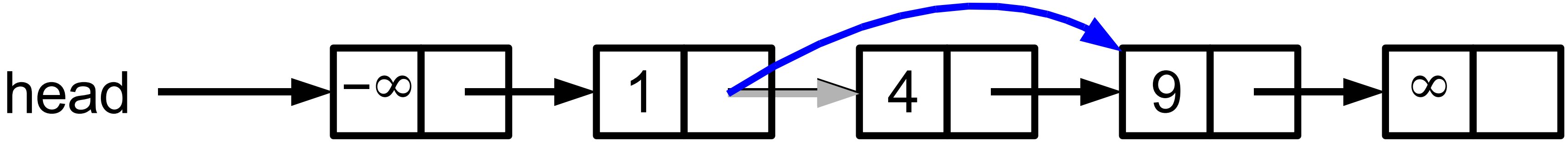
- Data type: set of integers (no duplicates)
  - S.add(x): Boolean: S := S ∪ {x}; return true iff x not already in S
  - S.remove(x): Boolean: S := S \ {x}; return true iff x in S initially
  - S.contains(x): Boolean: return true iff x in S (no change to S)
- Simple ordered linked-list-based implementation
  - illustrate techniques useful for pointer-based data structures
    - poor data structure for this specific data type



Sequential list-based set



add(3)



remove(4)

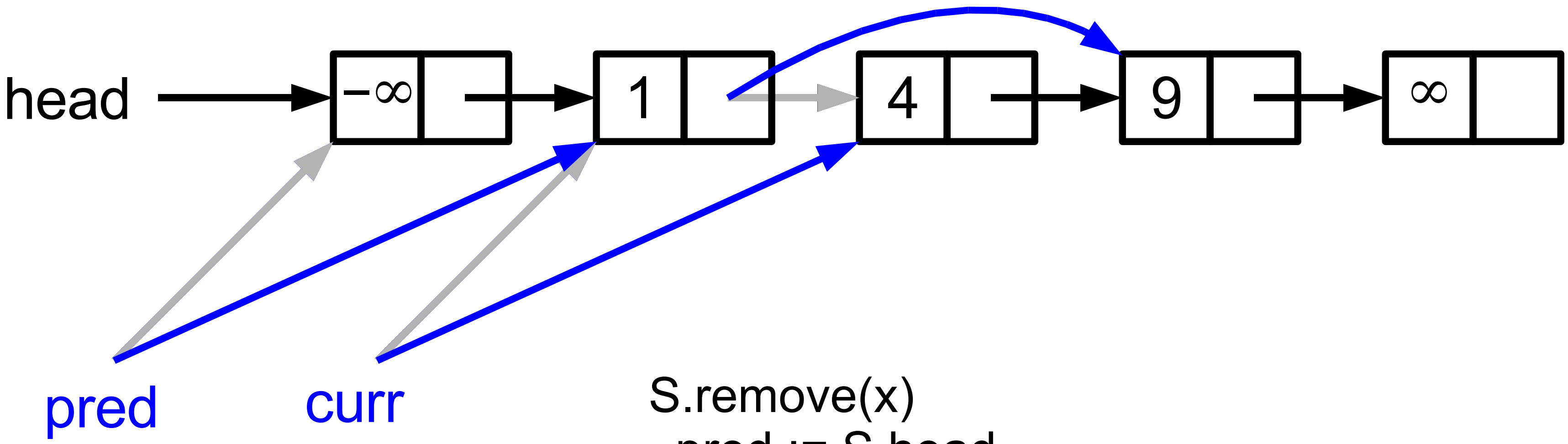
Sequential list-based set

```
S.add(x)
  pred := S.head
  curr := pred.next
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    return false
  else
    node = new Node(x)
    node.next = curr
    pred.next = node
    return true
```

```
S.remove(x)
  pred := S.head
  curr := pred.next
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    pred.next = curr.next
    return true
  else
    return false
```

```
S.contains(x)
  curr := S.head
  while (curr.key < x)
     curr := curr.next
  if curr.key = x then
    return true
  else
    return false
```

Sequential list-based set



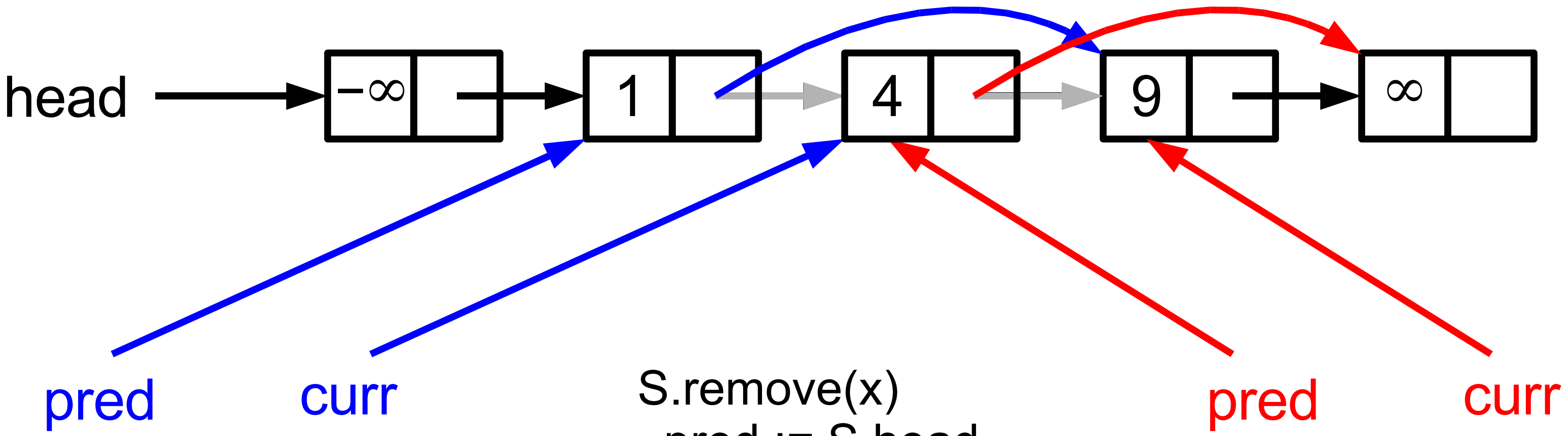
remove(4)

```
S.remove(x)
  pred := S.head
  curr := pred.next
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    pred.next = curr.next
    return true
  else
    return false
```

Allowing concurrent access

- Is this algorithm “thread-safe”?
- What can go wrong?
- Can we “fix” it?
- How?

Concurrent operations (bad)



remove(4)

remove(9)

```
S.remove(x)
  pred := S.head
  curr := pred.next
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    pred.next = curr.next
    return true
  else
    return false
```

Coarse-grained locking

```
S.add(x)
  S.lock()
  pred := S.head
  curr := pred.next
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    S.unlock()
    return false
  else
    node = new Node(x)
    node.next = curr
    pred.next = node
    S.unlock()
  return true
```

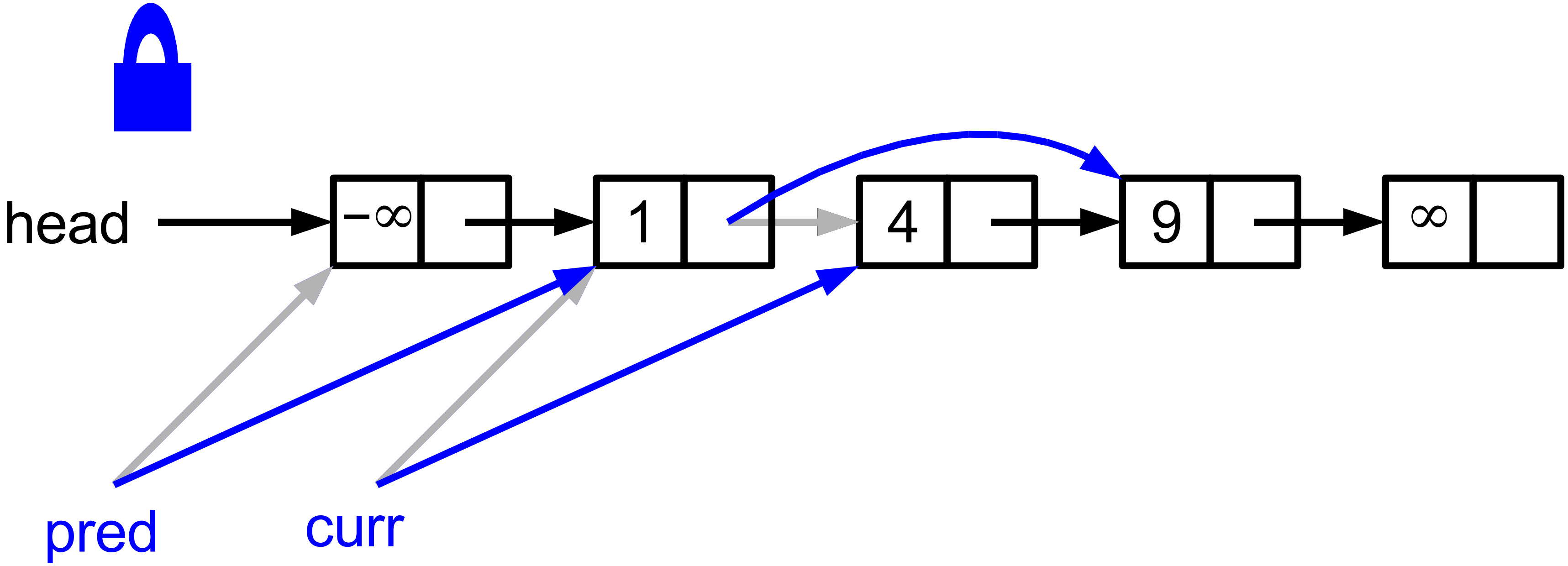
Why can we unlock early here?

```
  pred := S.
  curr := pred.nex
  while (curr.key < x)
    pred := curr
    curr := pred.next
  if curr.key = x then
    pred.next = curr.next
    S.unlock()
    return true
  else
    S.unlock()
    return false
```

```
S.contains(x)
  S.lock()
  curr := S.head
  while (curr.key < x)
     curr := curr.next
  S.unlock()
  if curr.key = x then
    return true
  else
    return false
```

Why does this work? (cf. RMWfromRW algorithm)
What progress guarantees do we get?

Coarse-grained locking



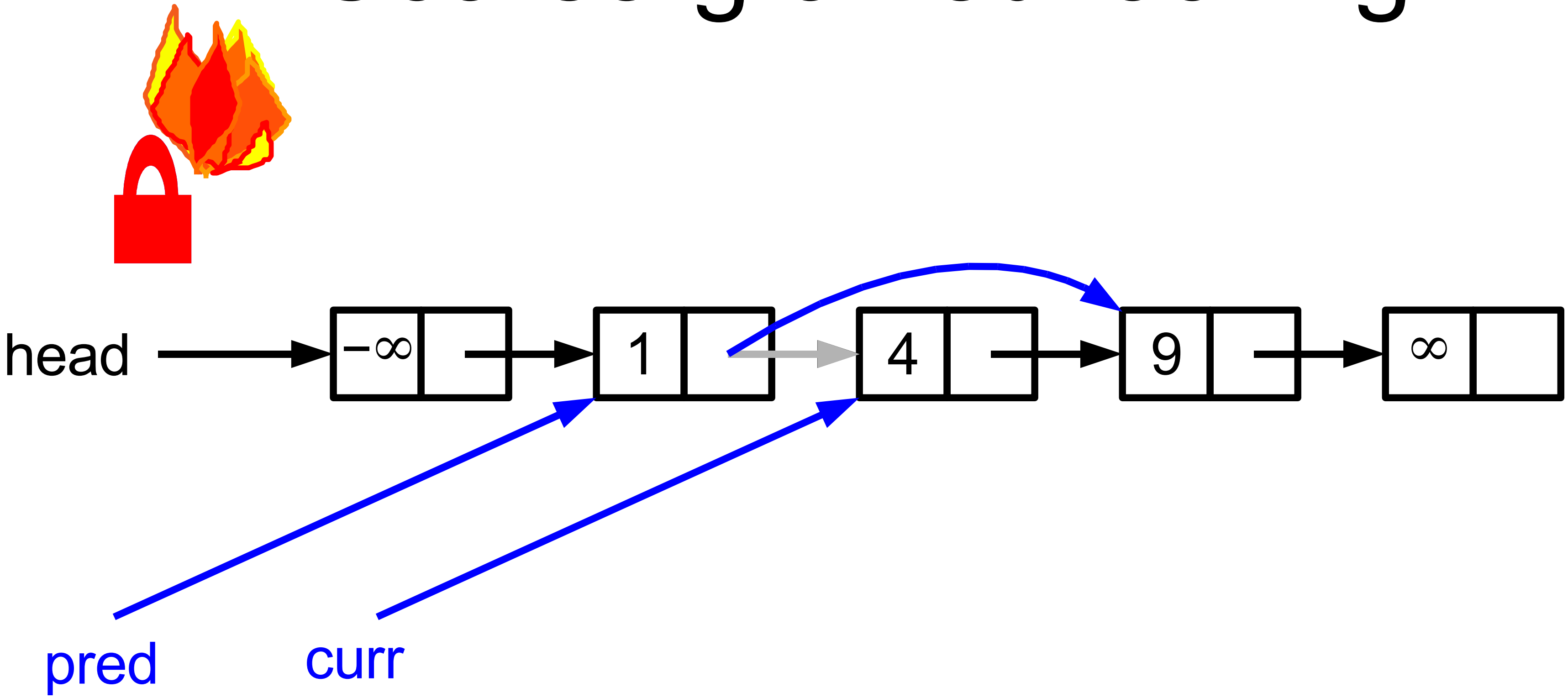
remove(4)

Coarse-grained locking

For many applications, this is the best solution!
(Don't underrate simplicity.)

- Easy
  - to write
  - to prove correct
- No fault-tolerance
  - but it is deadlock-free!
  - if we use queue locks, it's lockout-free
- Poor performance when contention is high
  - essentially no concurrent access
  - but often good enough for low contention

Coarse-grained locking



remove(4)

remove(9)

add(6)

contains(4)

add(3)

Improving coarse-grained locking

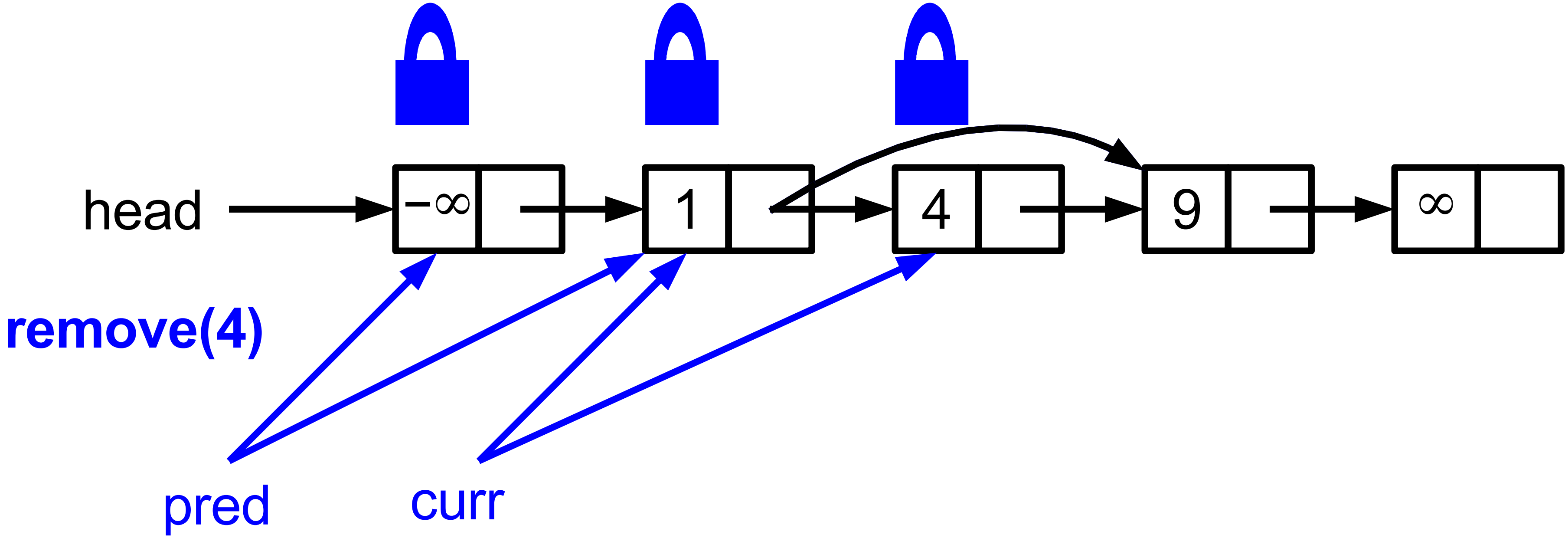
- Reader/writer locks
  - allow multiple readers to hold lock simultaneously
  - writers can easily starve
    - introduce “waiting” bit to avoid this
  - contains takes only read lock
    - can be big win if contains is the most common operation
  - what about add or remove that returns false?
    - upgrading

Fine-grained locking

- associate locks with smaller pieces of data
  - methods that work on disjoint pieces can proceed concurrently
- simple to prove atomicity if locking is “two-phase”
  - first acquire locks, then release (no acquire after any release)
    - typically release at the end of operation: strict two-phase locking
- can be expensive to acquire all the locks
- must be careful to avoid deadlock
  - typically acquire locks in some predetermined order
- naive two-phase application doesn't help (why not?)
  - it does with reader/writer locks, but tricky to avoid deadlock

Hand-over-hand locking

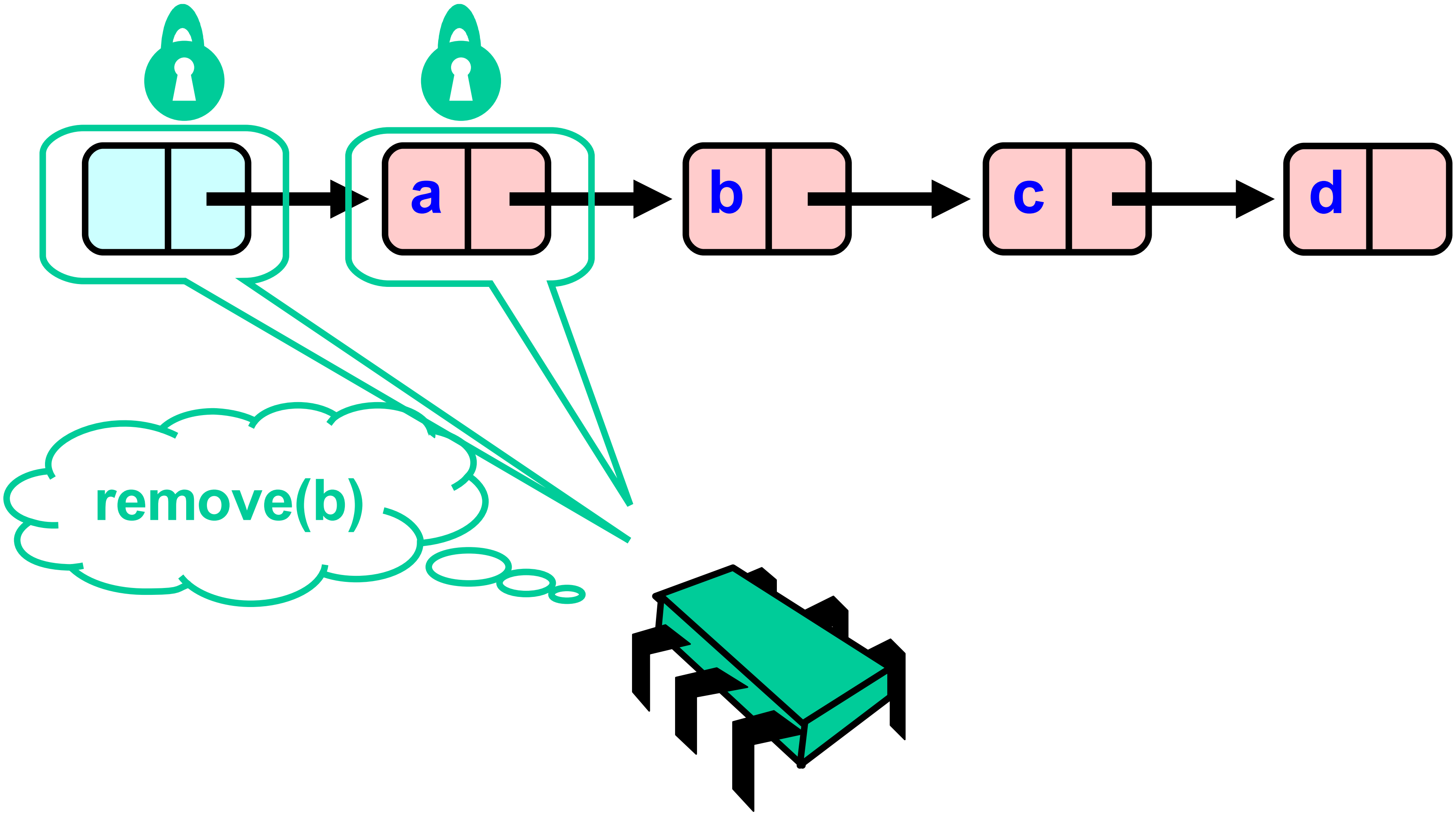
- Fine-grained locking, but not “two-phase”
  - atomicity doesn't follow from general rule; a bit tricky to prove
- Hold at most two locks at a time
  - acquire lock for successor before releasing lock for predecessor



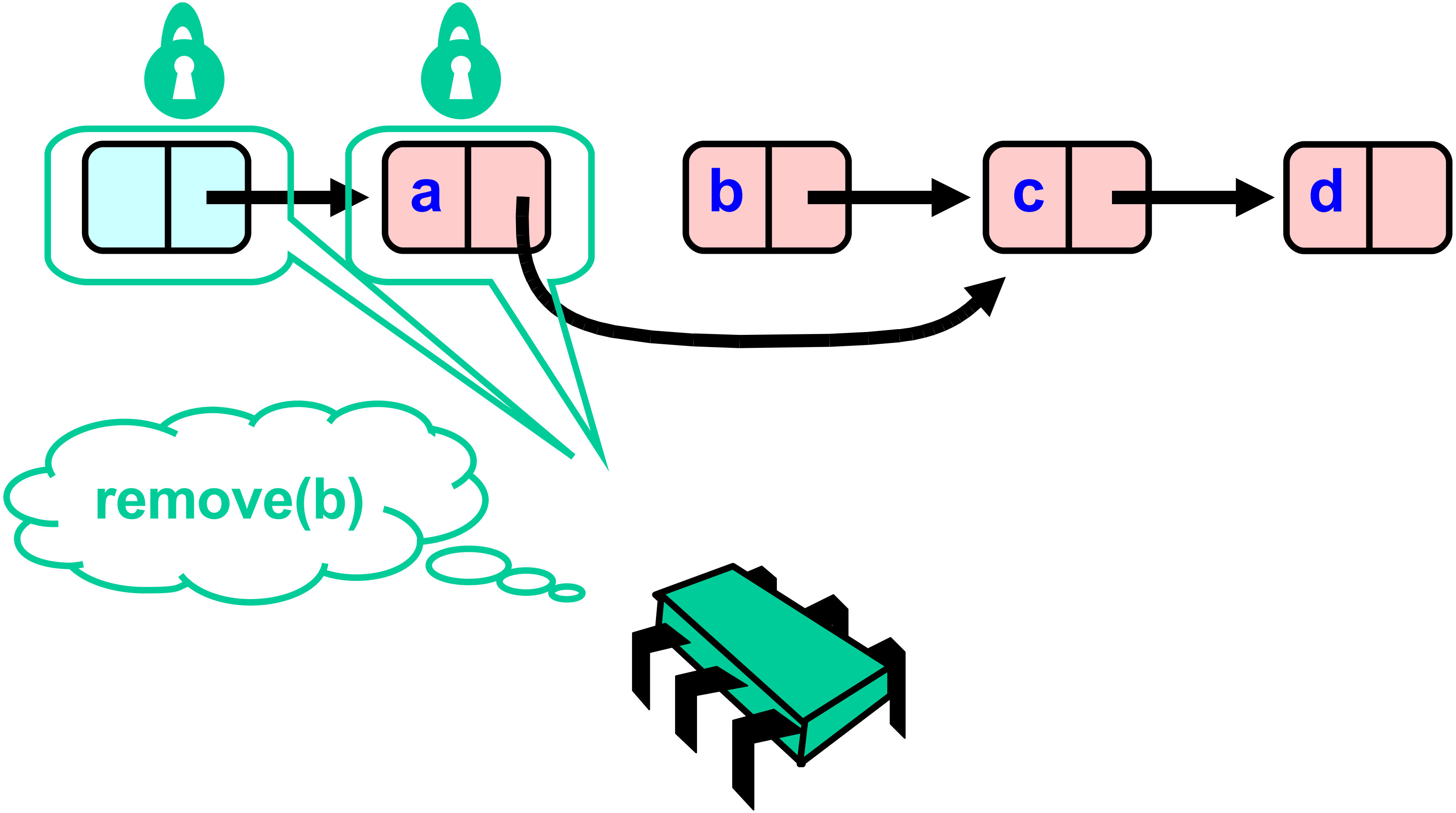
Hand-over-hand locking

- Must we lock the successor of a node we are trying to add?
  - we don't need to lock to read the key (why not?)
- Must we lock a node we are trying to remove?

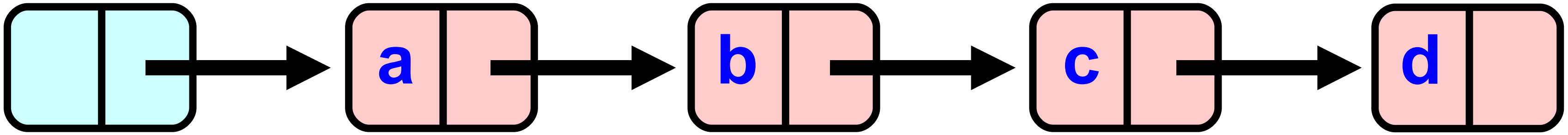
Removing a Node

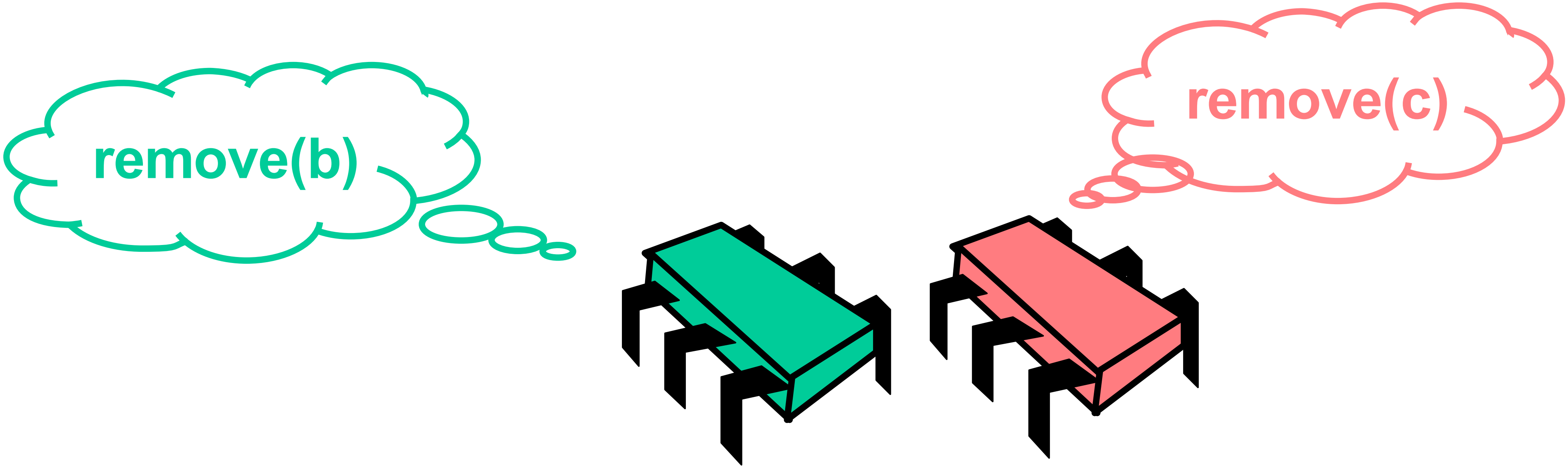


Removing a Node

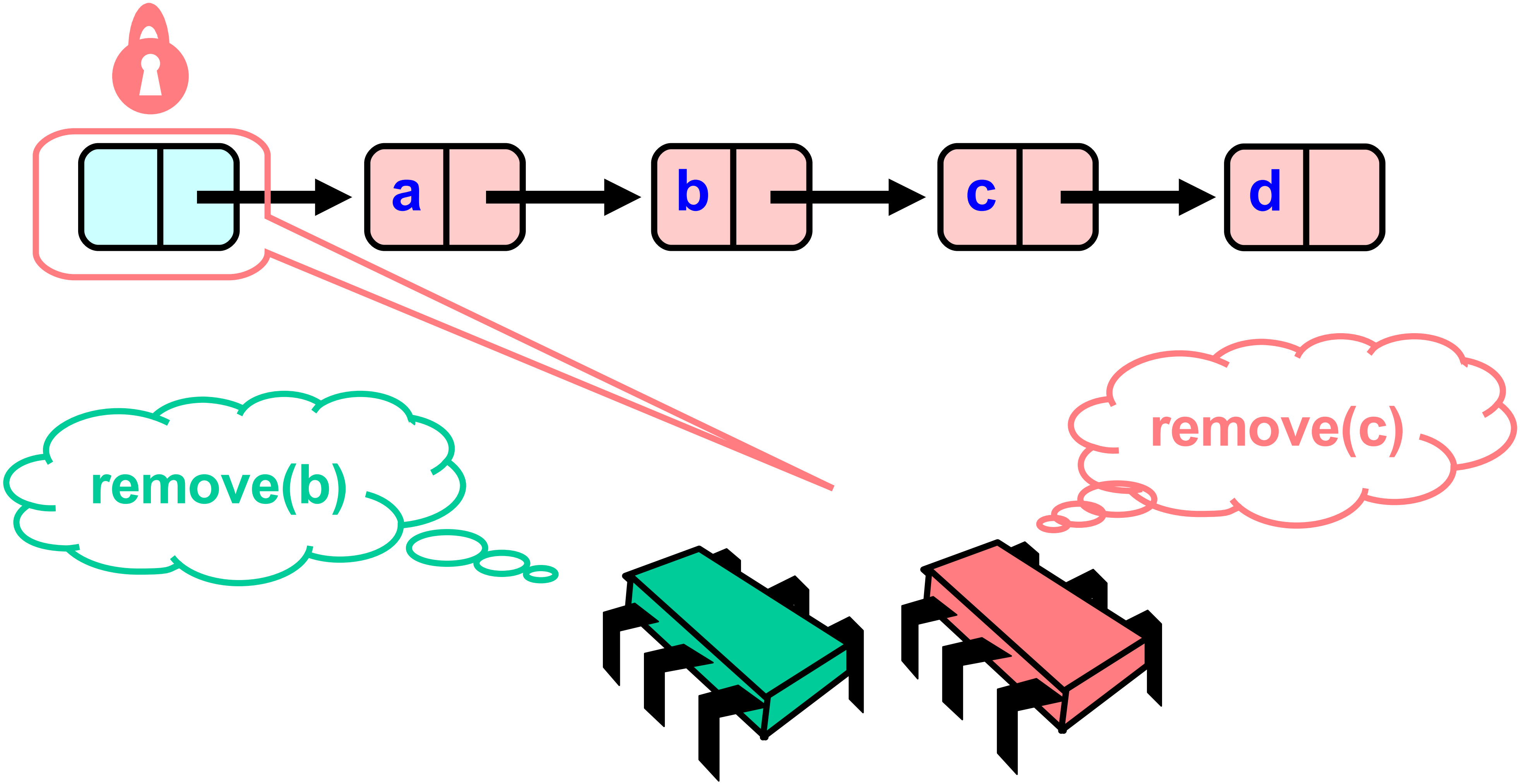


Removing a Node

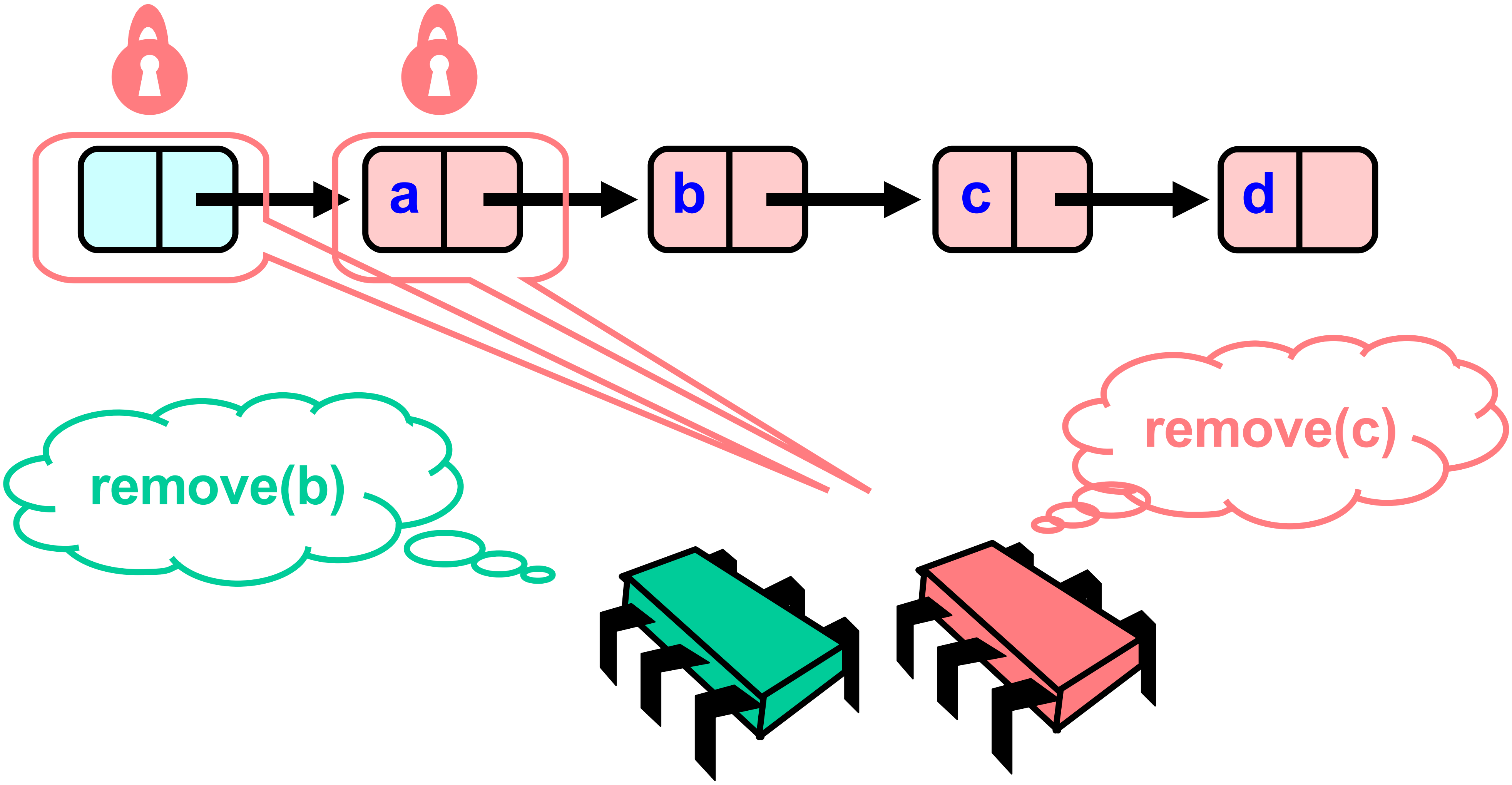




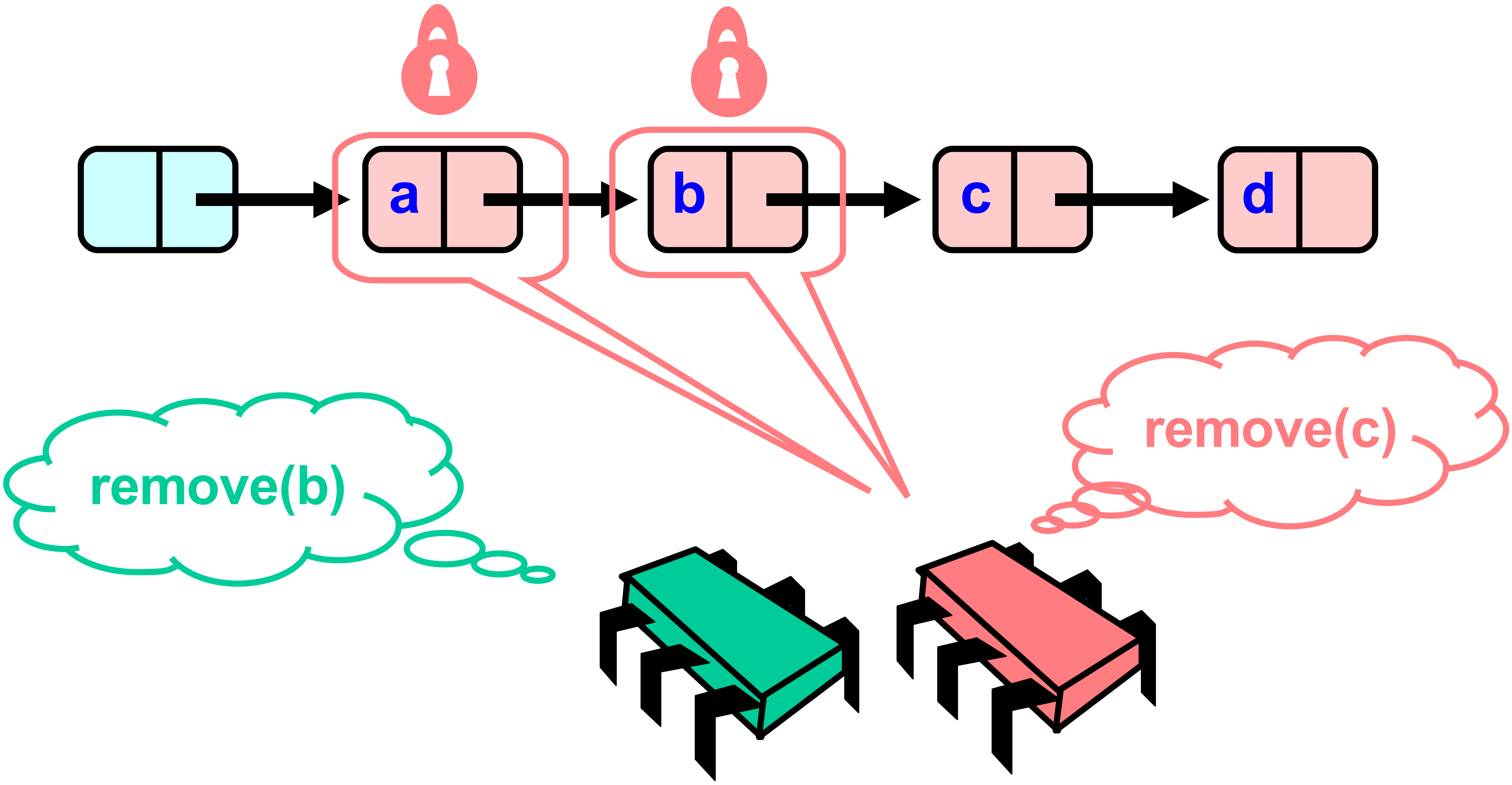
Removing a Node



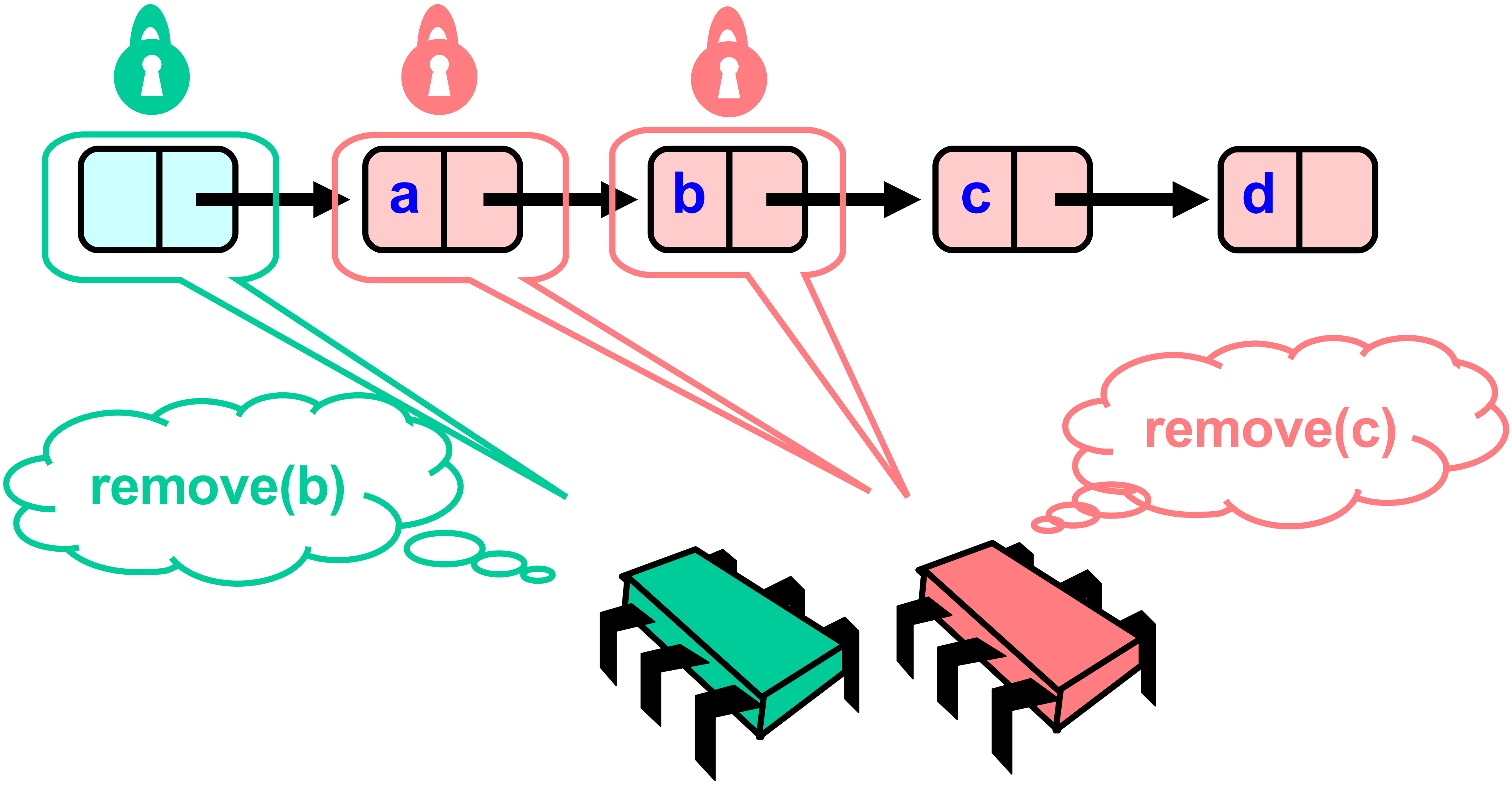
Removing a Node



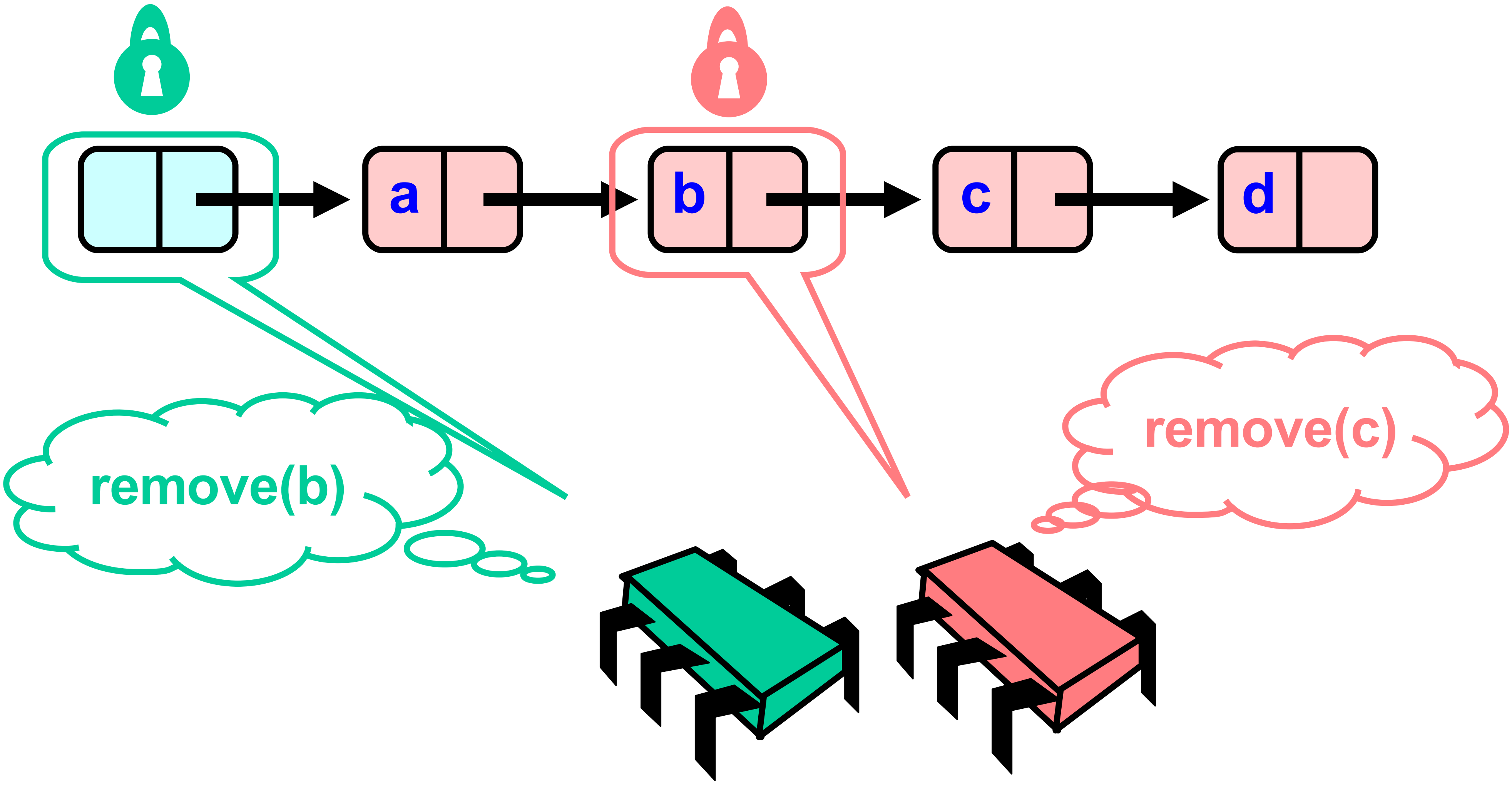
Removing a Node



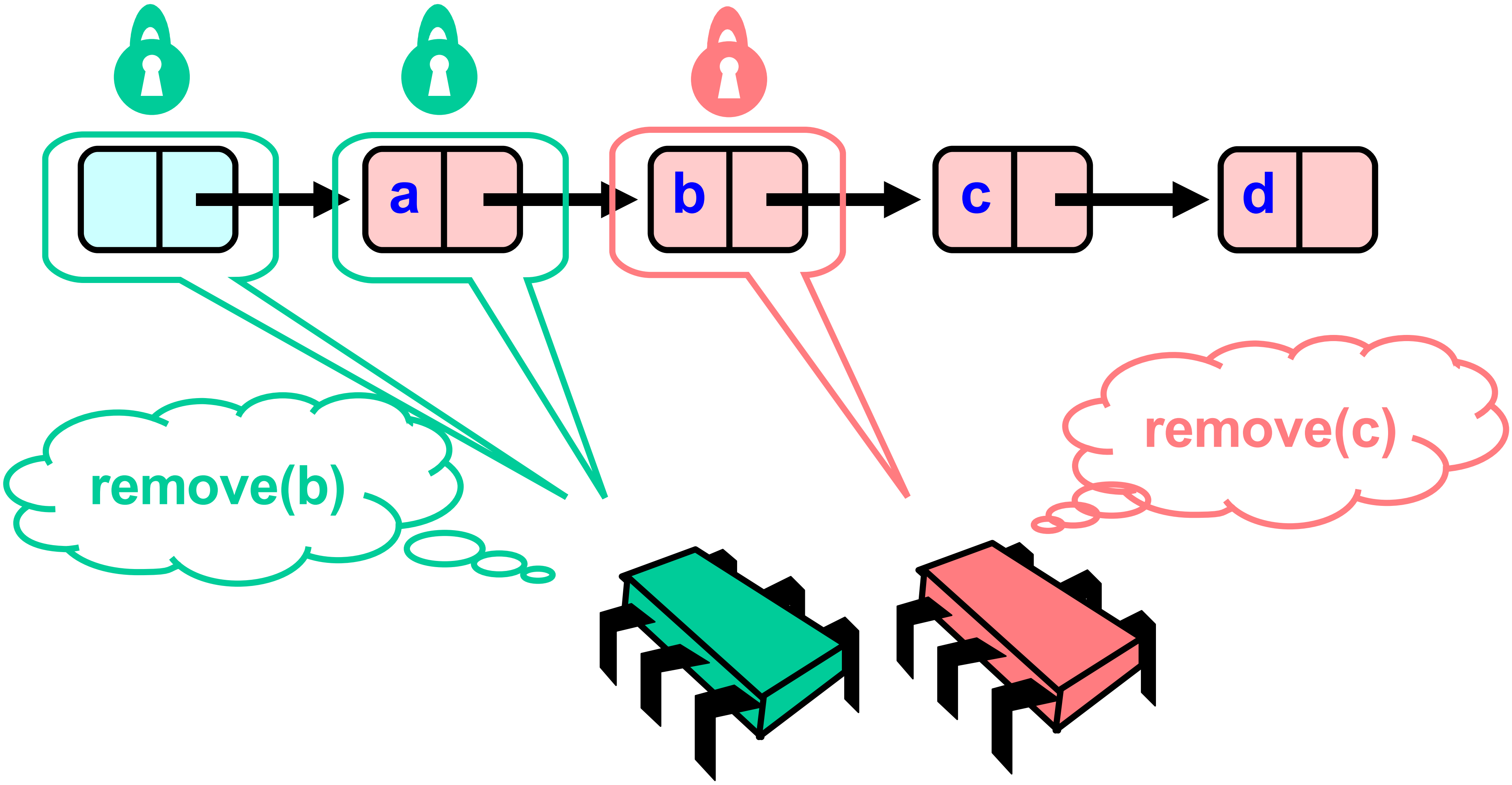
Removing a Node



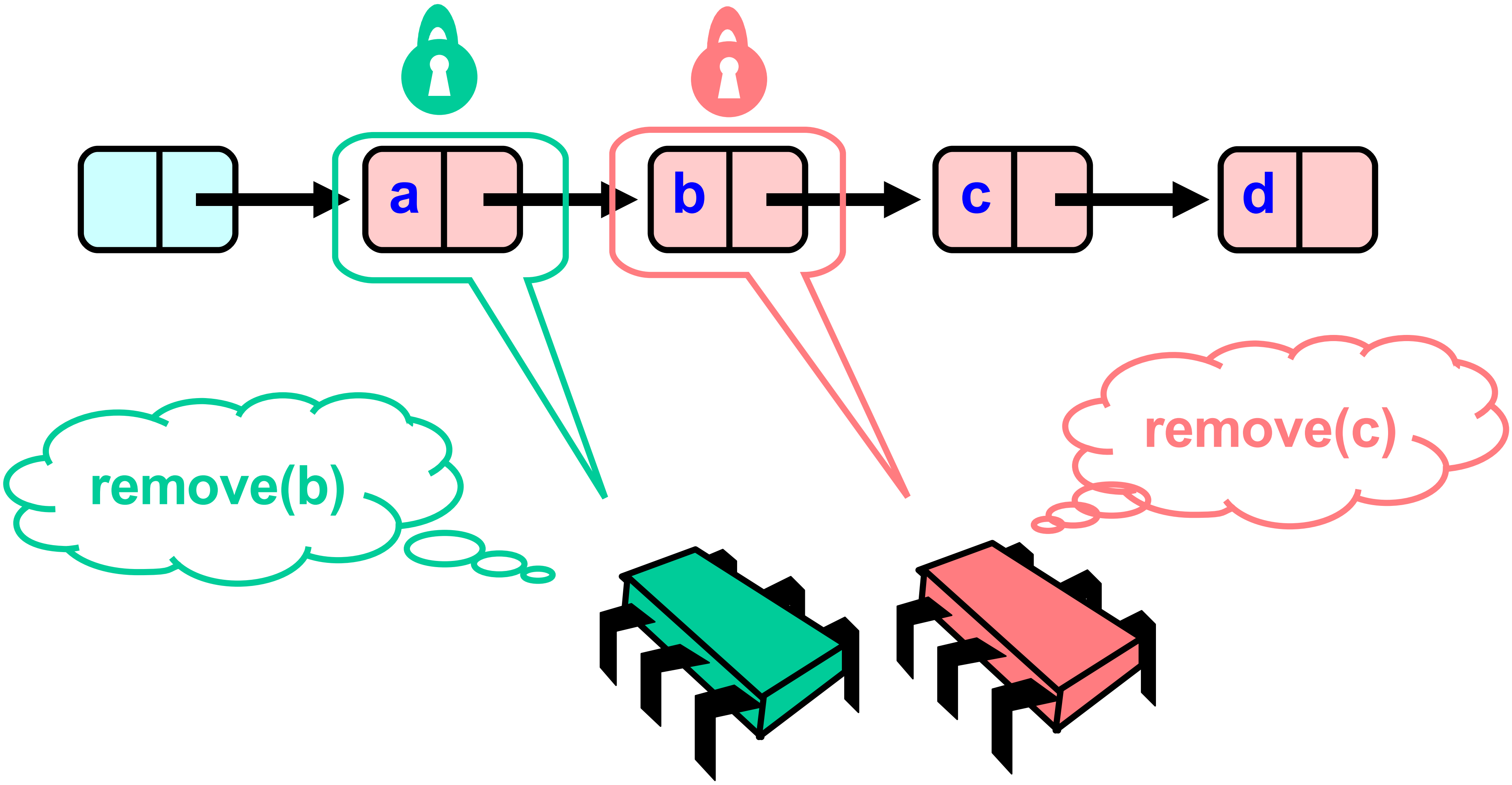
Removing a Node



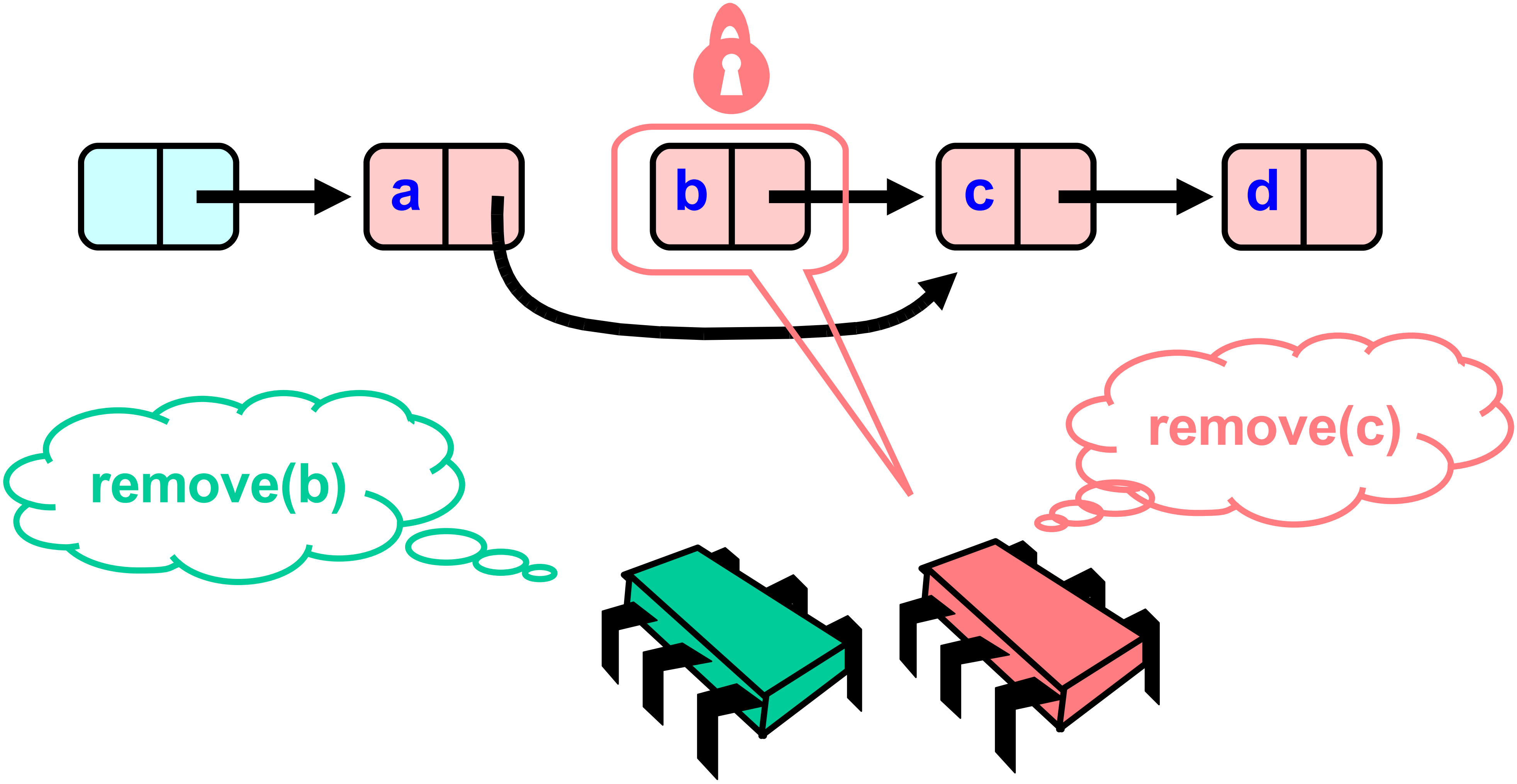
Removing a Node



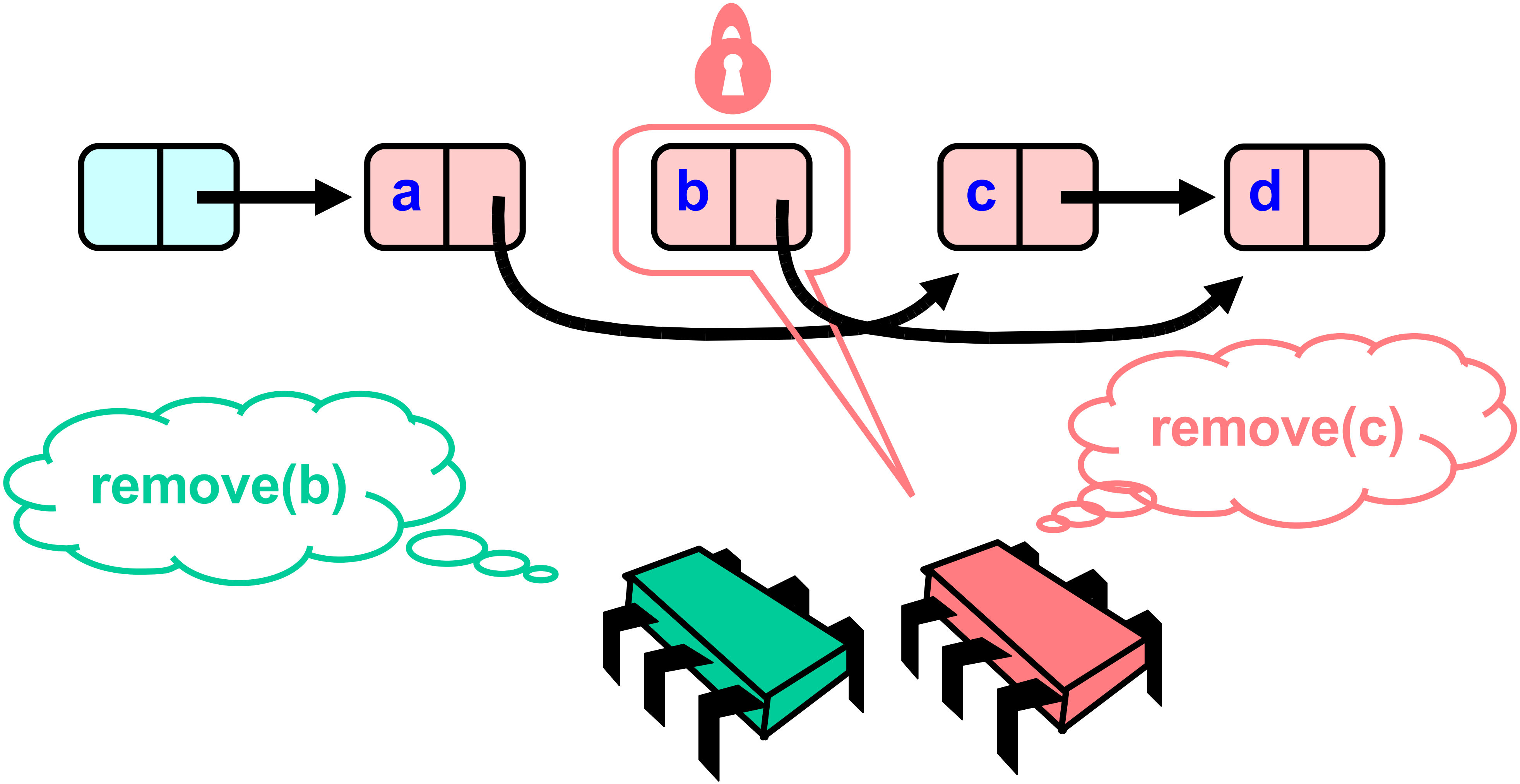
Removing a Node



Uh, Oh



Uh, Oh



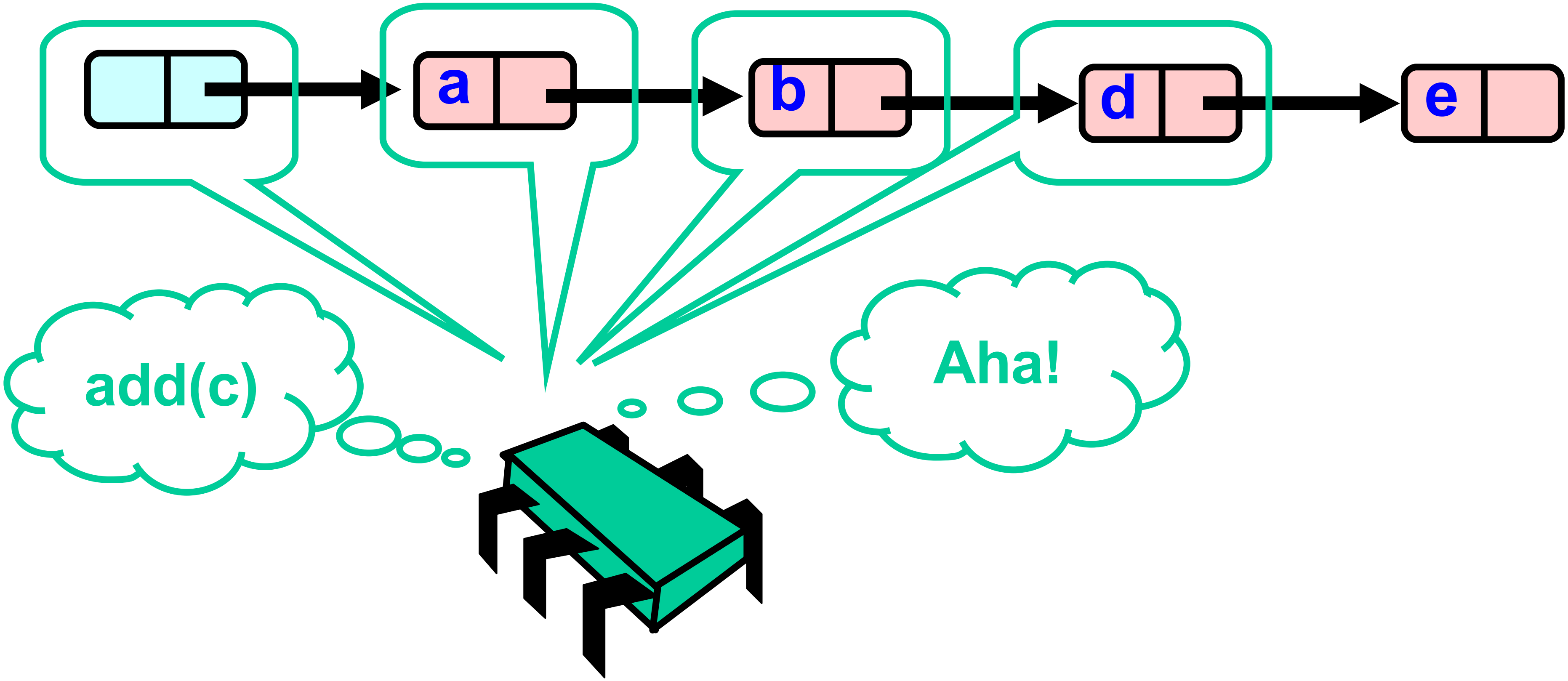
Hand-over-hand locking

- Problems
  - must acquire O(k) locks, where k = |S|
  - threads can get stuck behind a slow thread
    - can avoid this by using reader/writer locks, but then must do something to avoid deadlock
- Idea: What if we find the nodes first without locking, and then lock only the nodes we need?
  - must ensure that the node we modify is still in list
  - optimistic locking

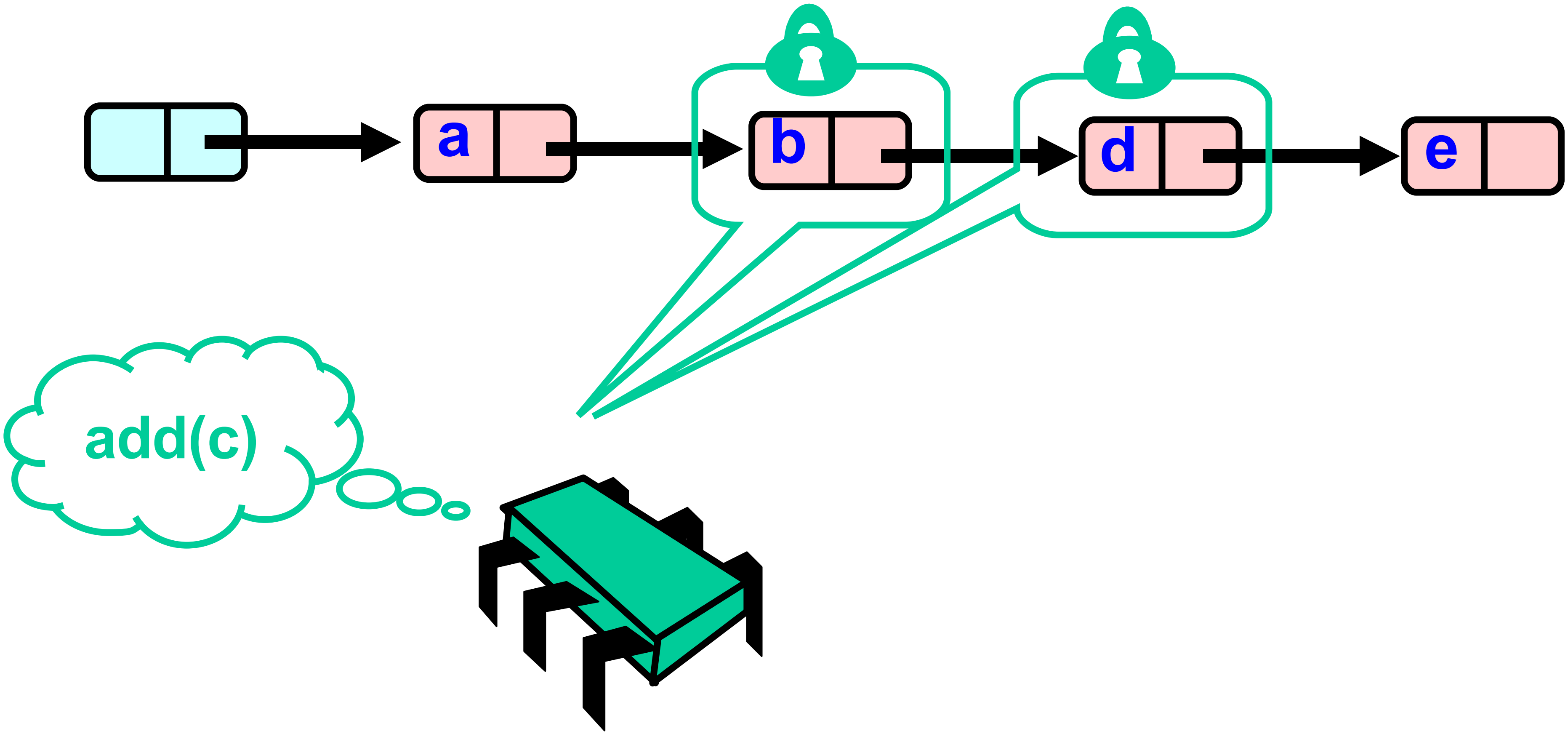
Optimistic locking

- Search down list without locking
- Find and lock appropriate nodes
- Verify that nodes are still adjacent and in list (validation)
  - we can do this by traversing list again (provided that nodes are not removed from list while they are locked)
- Better than hand-over-hand if
  - traversing twice without locking is cheaper than once with locking
    - traversal is wait-free! (we'll come back to this)
  - validation typically succeeds

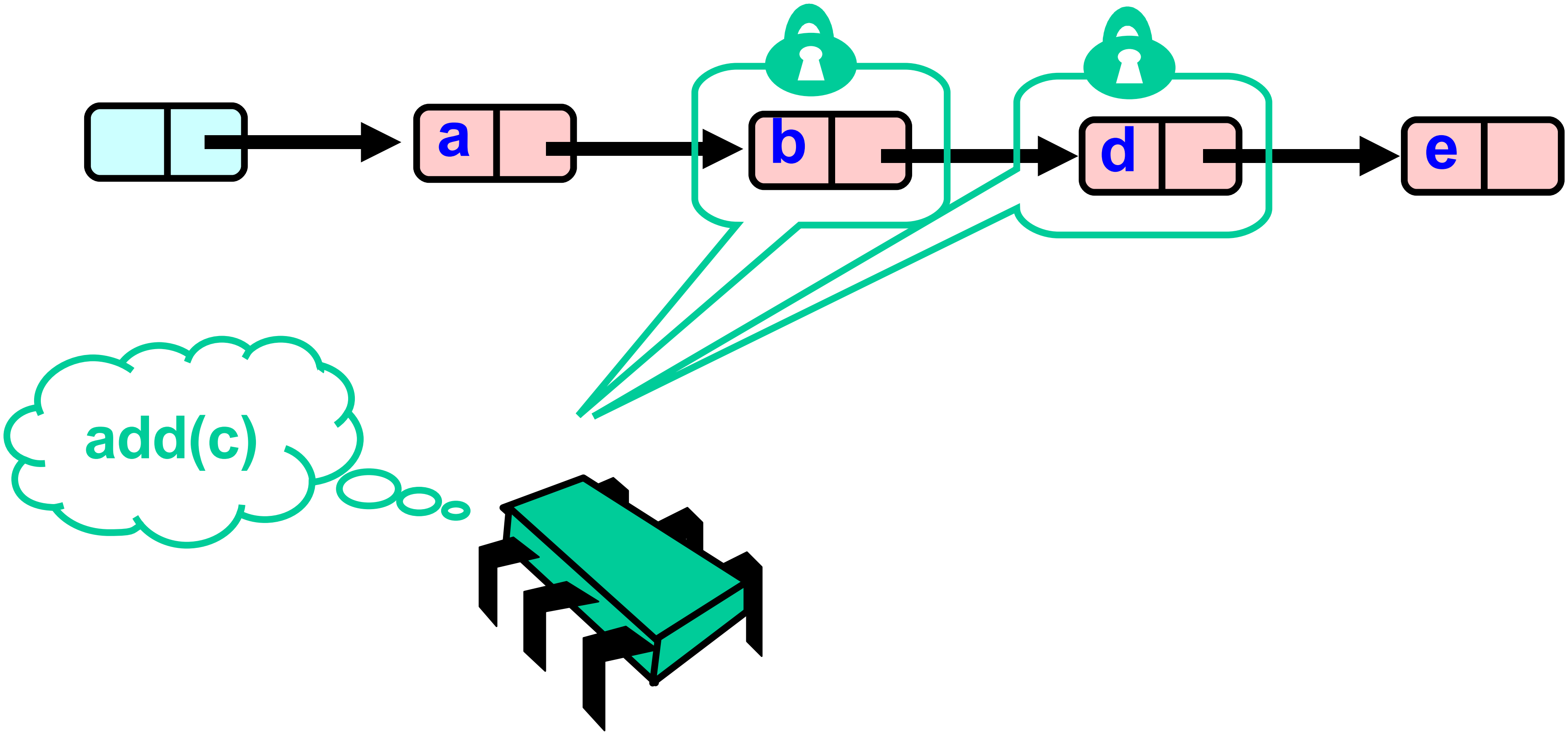
Optimistic locking



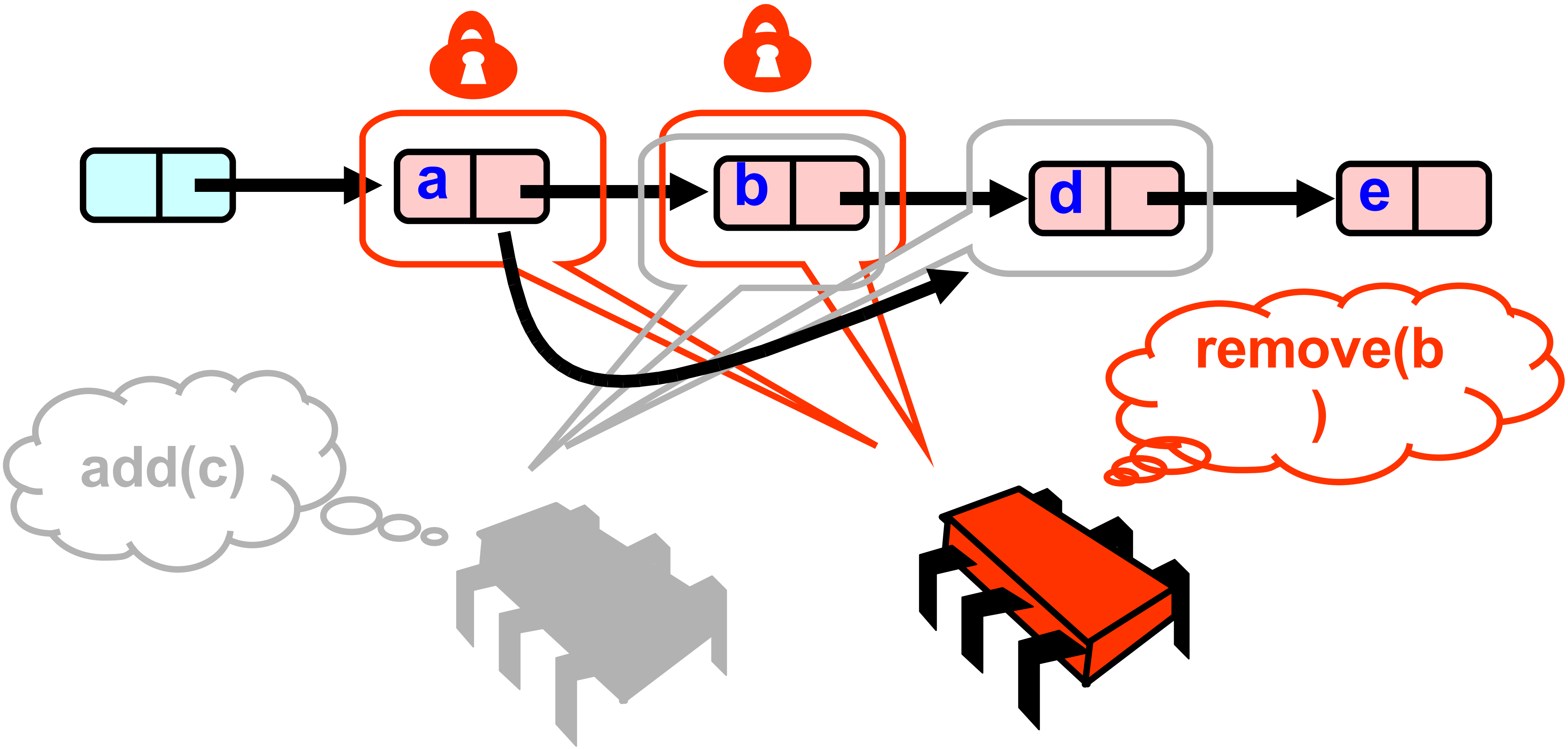
Optimistic locking



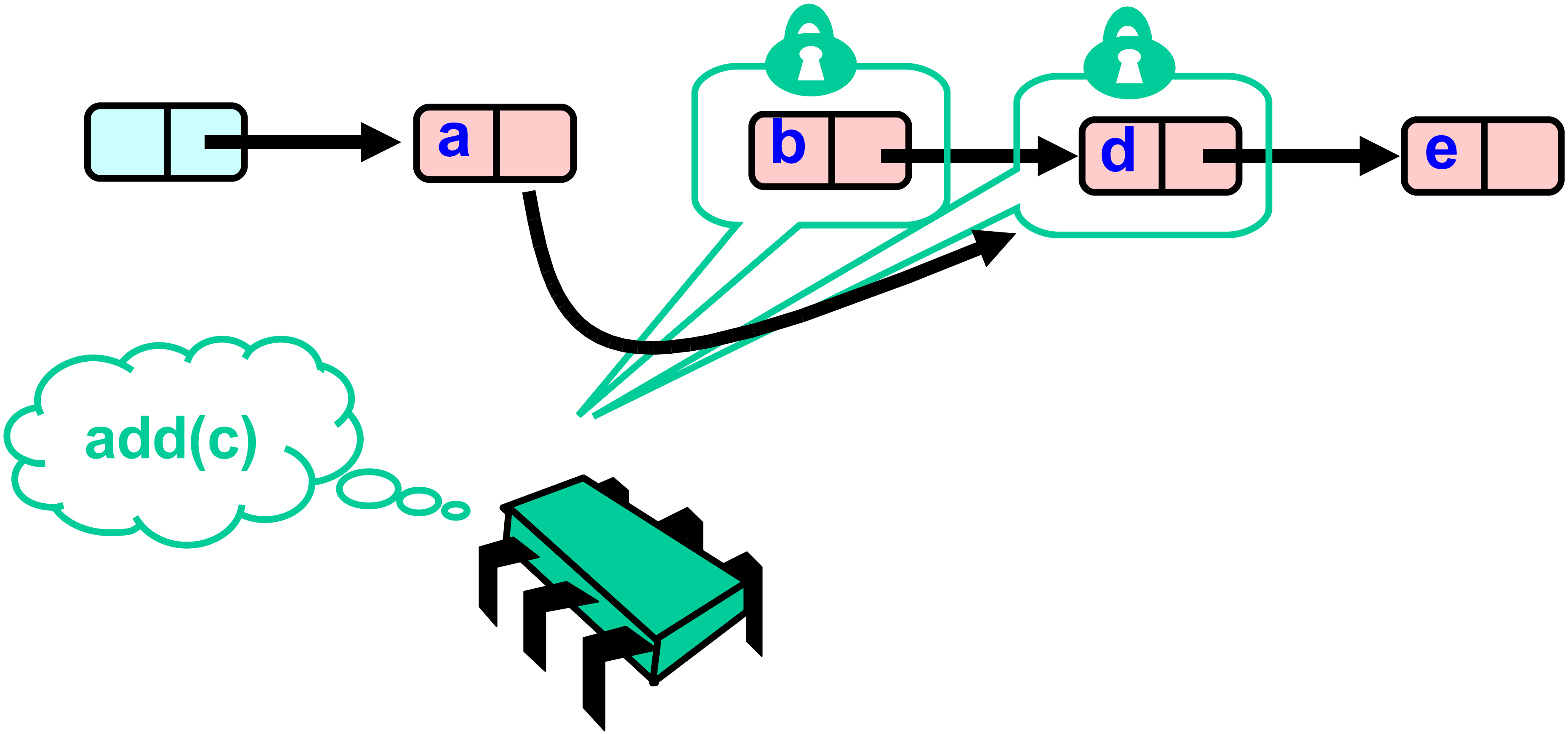
What can go wrong? (part 1)



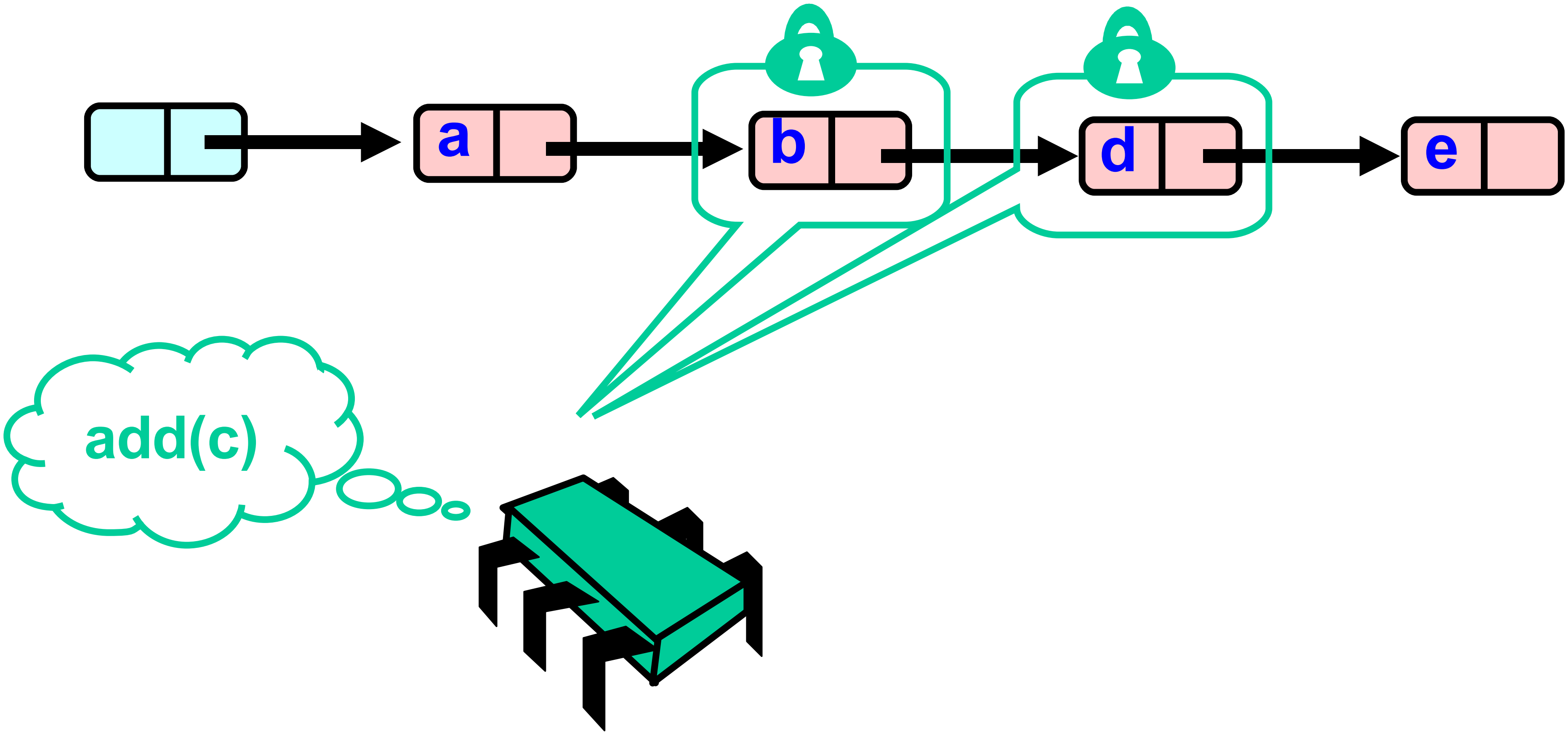
What can go wrong? (part 1)



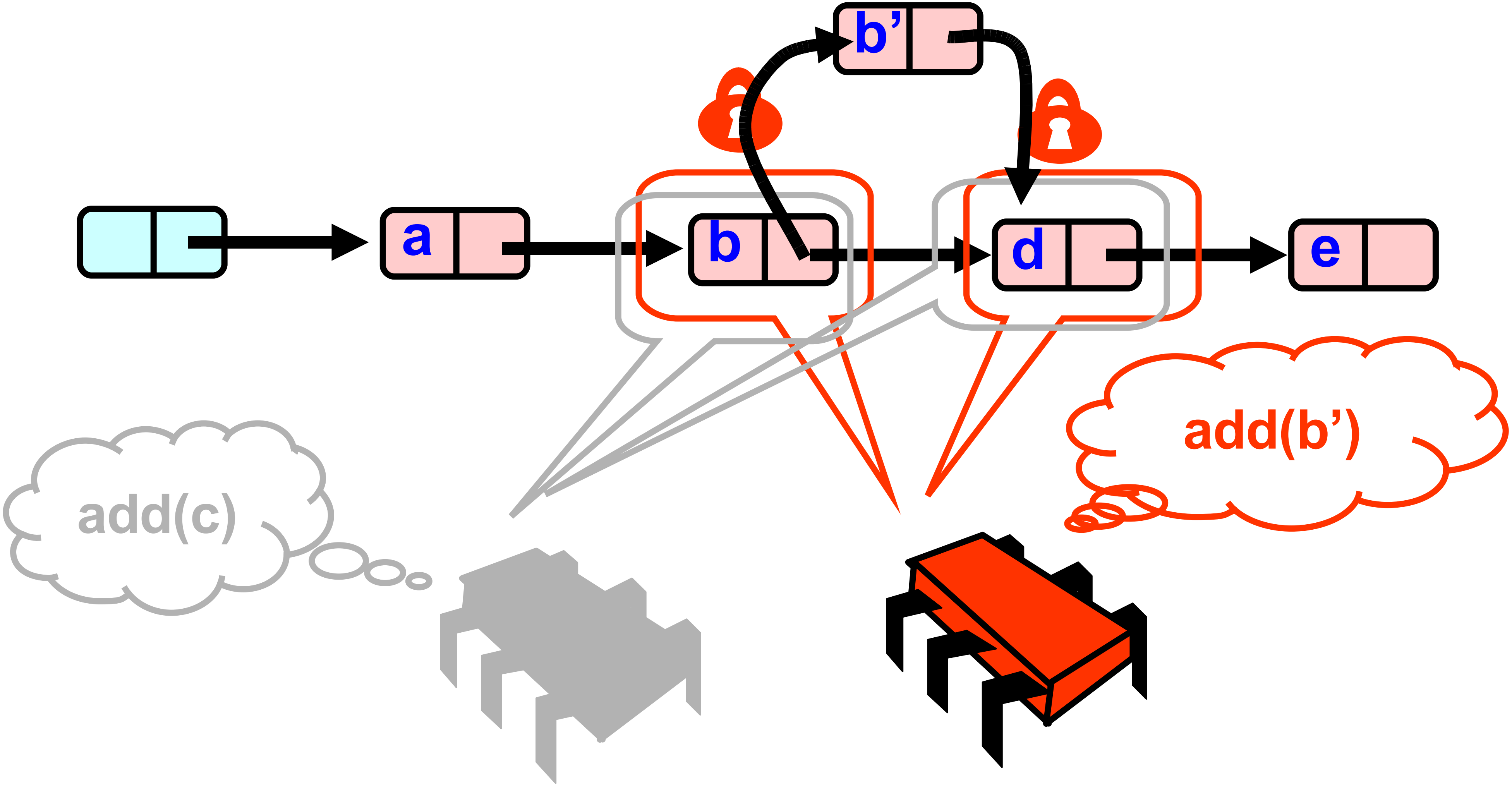
What can go wrong? (part 1)



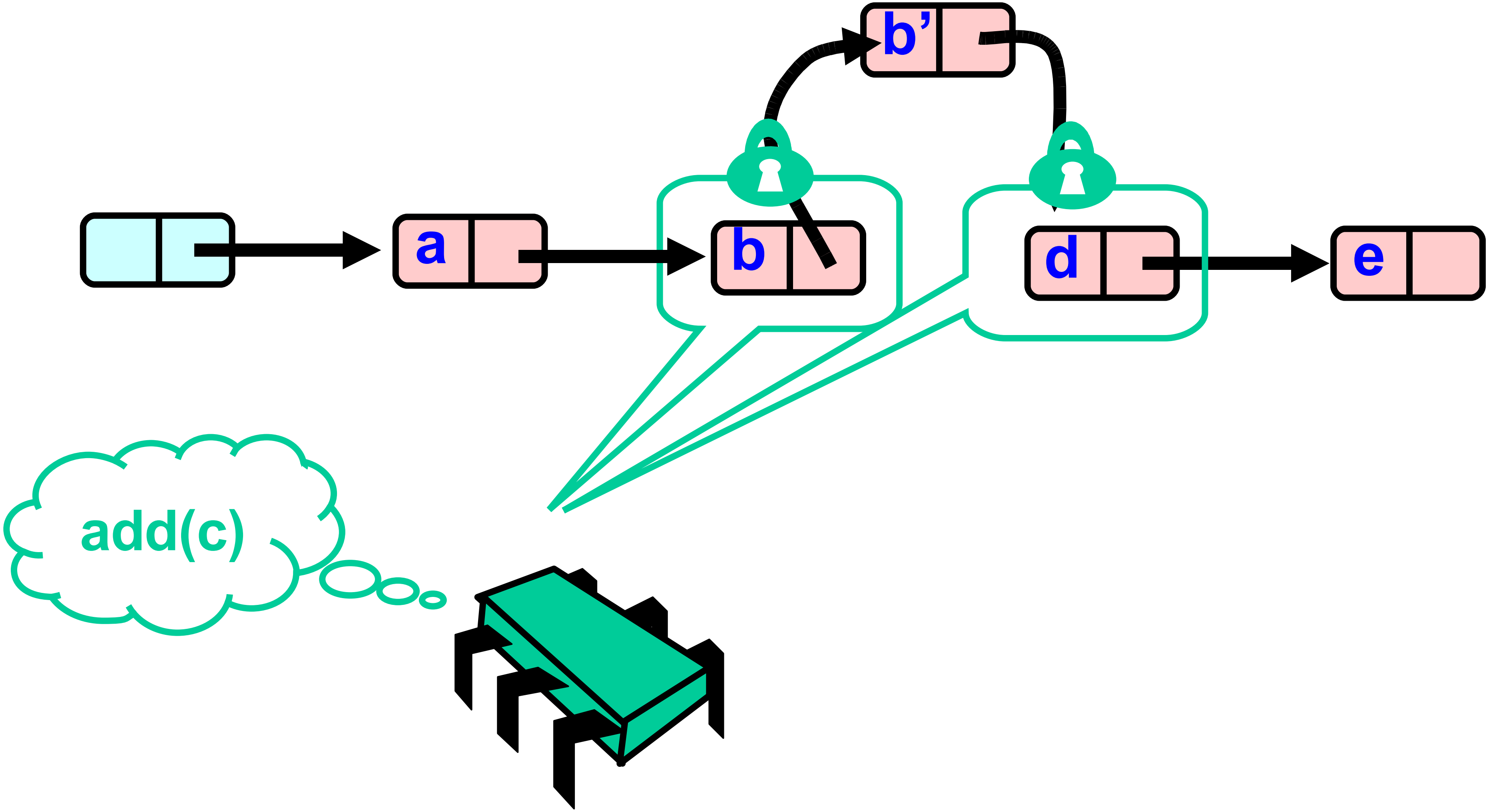
What can go wrong? (part 2)



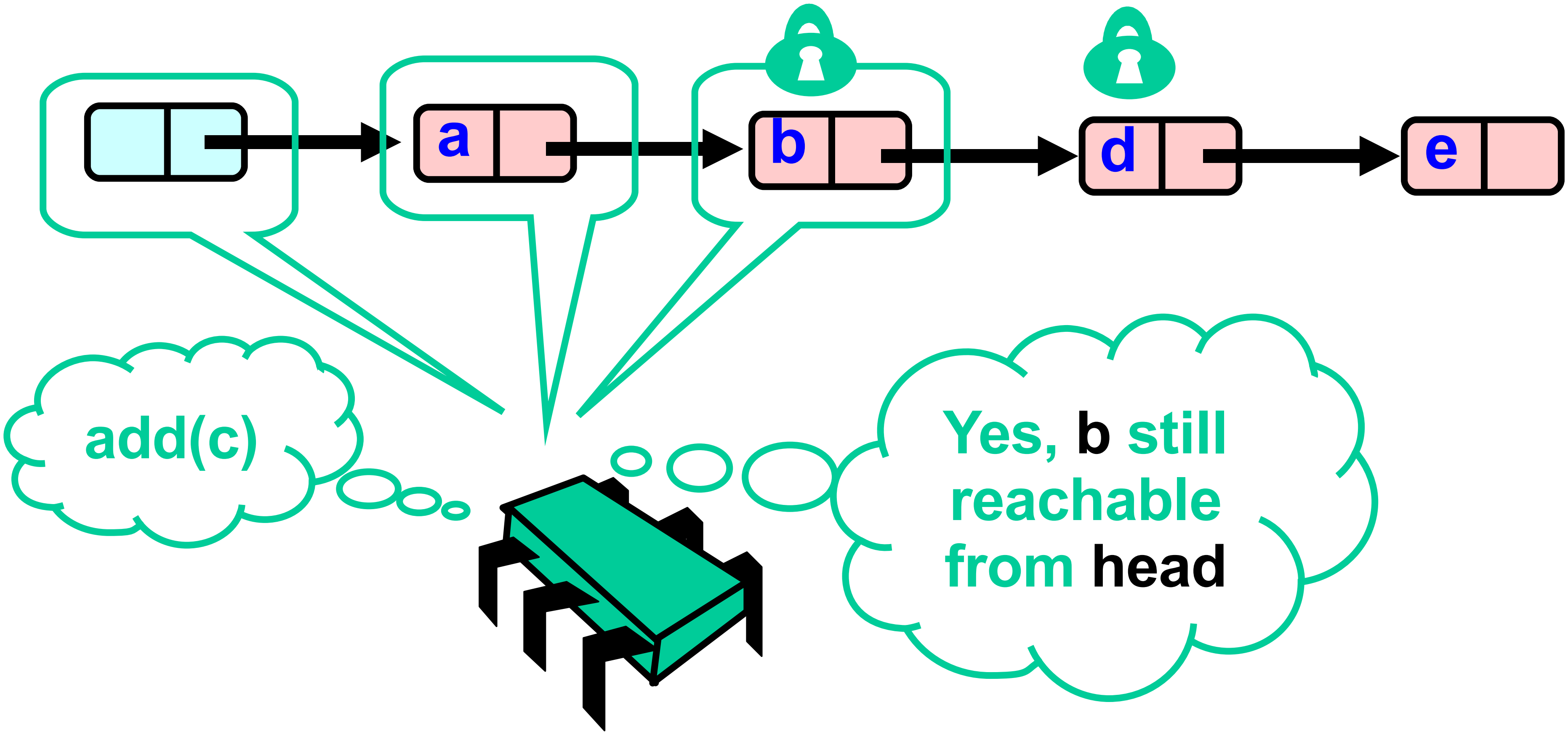
What can go wrong? (part 2)



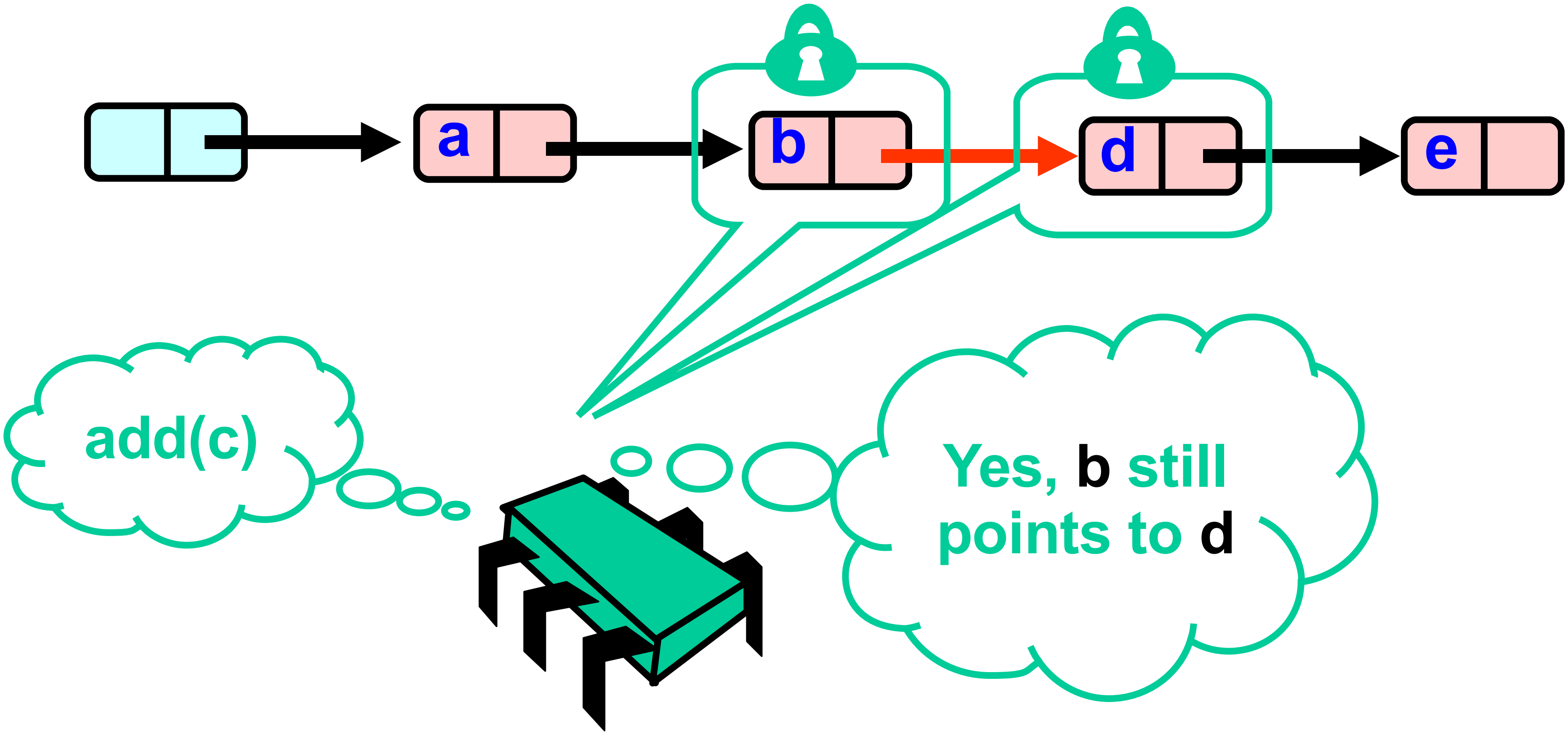
What can go wrong? (part 2)



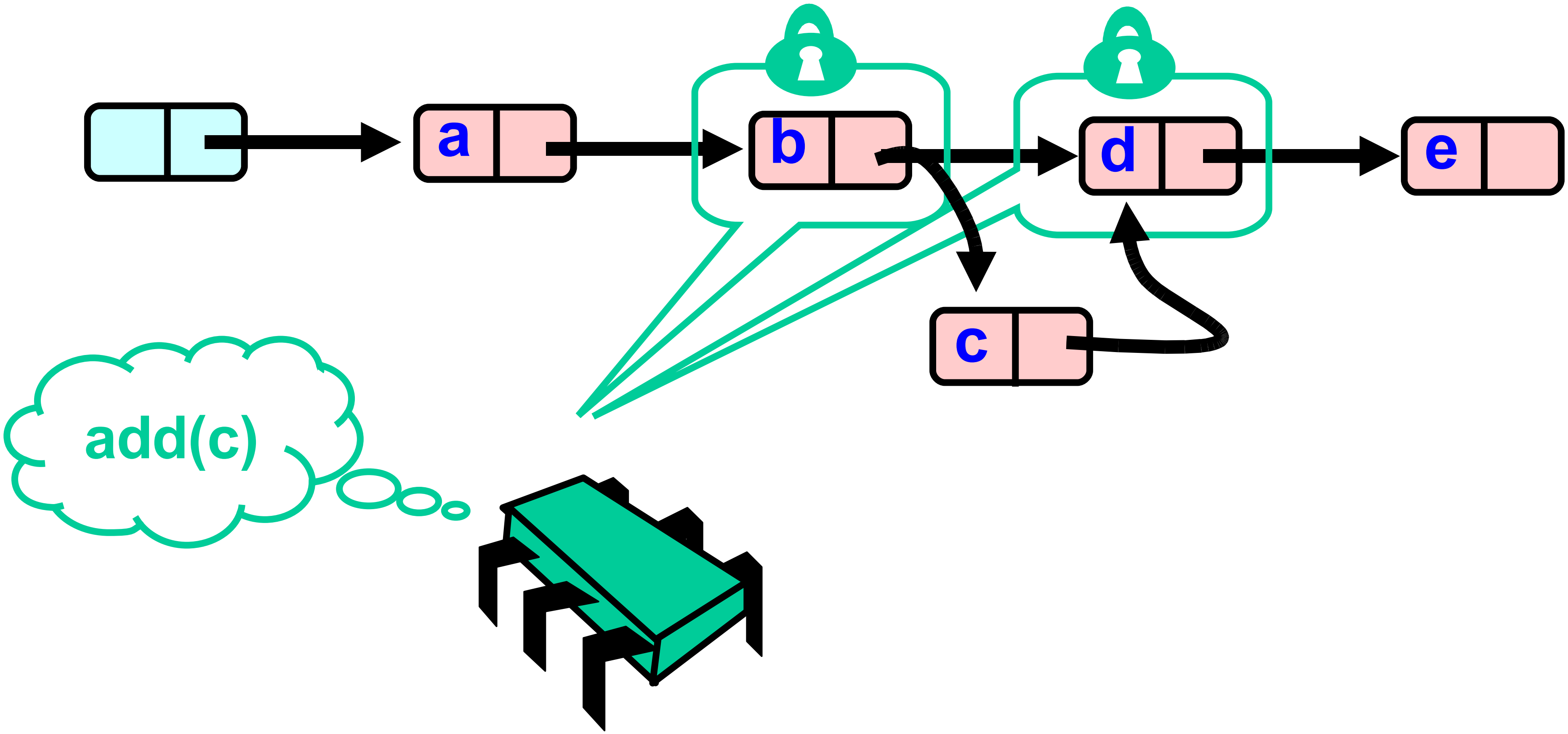
Validate (part 1)



Validate (part 2)



Optimistic locking



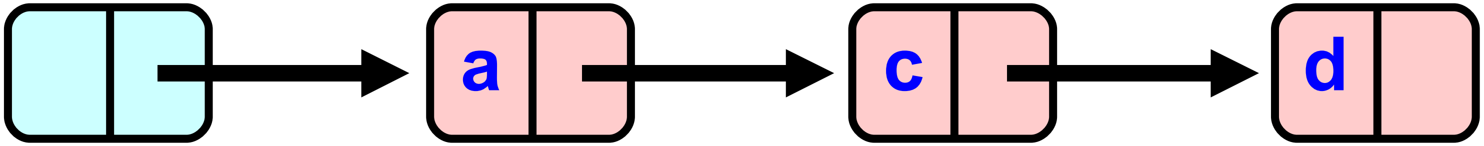
Lock-freedom

- Even without failures, locks can cause problems:
  - some operations take 1000x (or more) longer than others, nondeterministically due to page faults, descheduling, etc.
  - if this happens to anyone in their critical section, everyone else who wants to access that lock must wait
- What about **lock-free** algorithms?
  - if any thread executing a method does not fail then some method completes.
  - weaker than wait-free: starvation is possible
  - but rules out a delayed thread from blocking other threads indefinitely, and thus, no locks

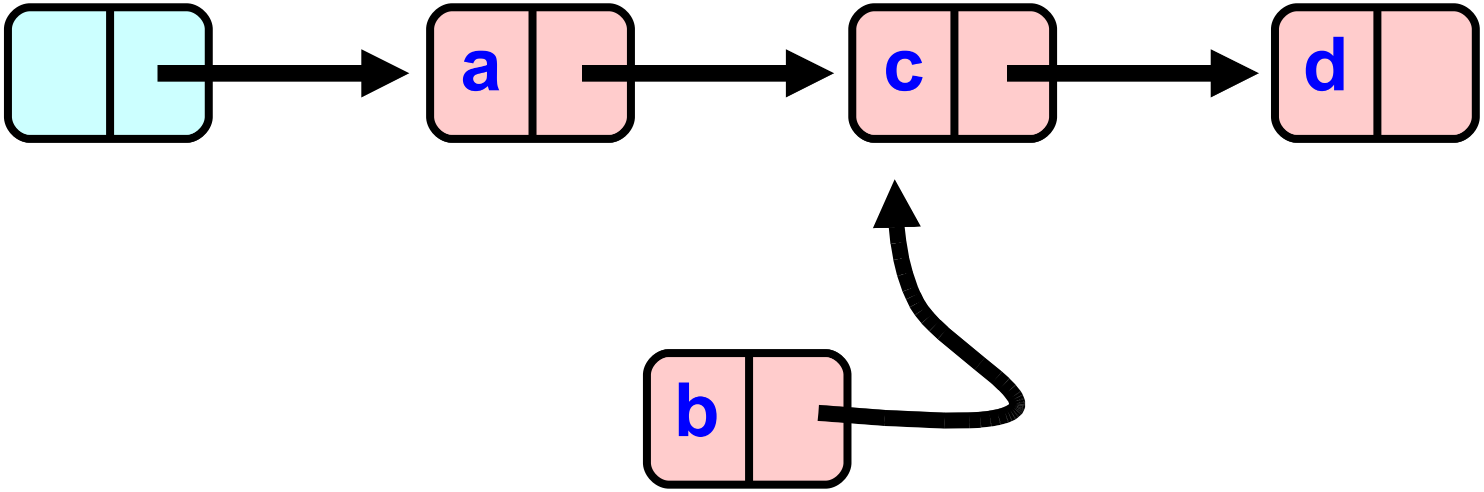
Lock-free list-based set

- Idea: Use CAS to change next pointer
  - make sure next pointer hasn't changed since you read it
    - assumes nodes aren't reused
  - possible because operations only change one pointer
  - but still nontrivial

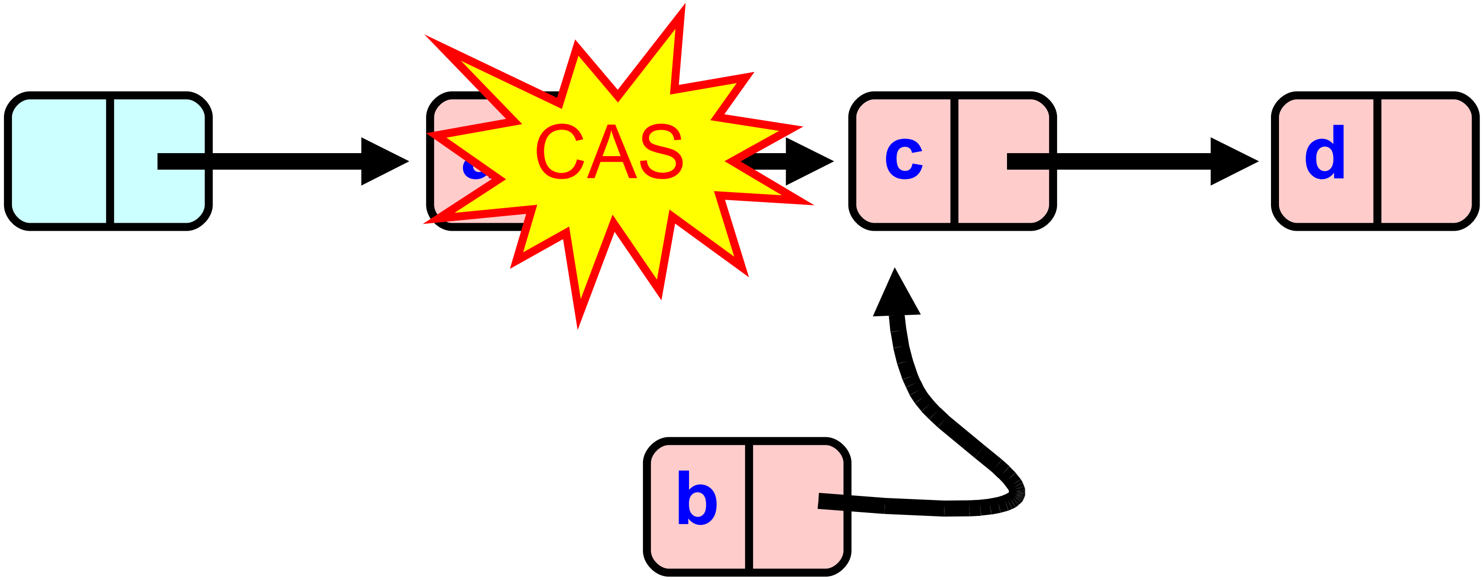
Adding a Node



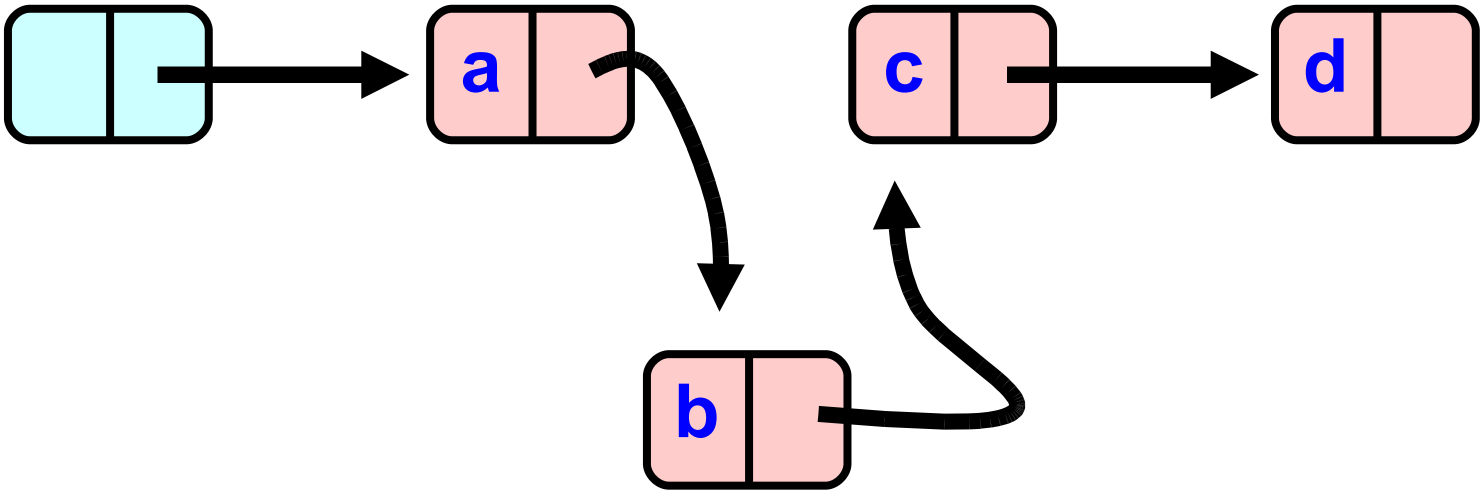
Adding a Node



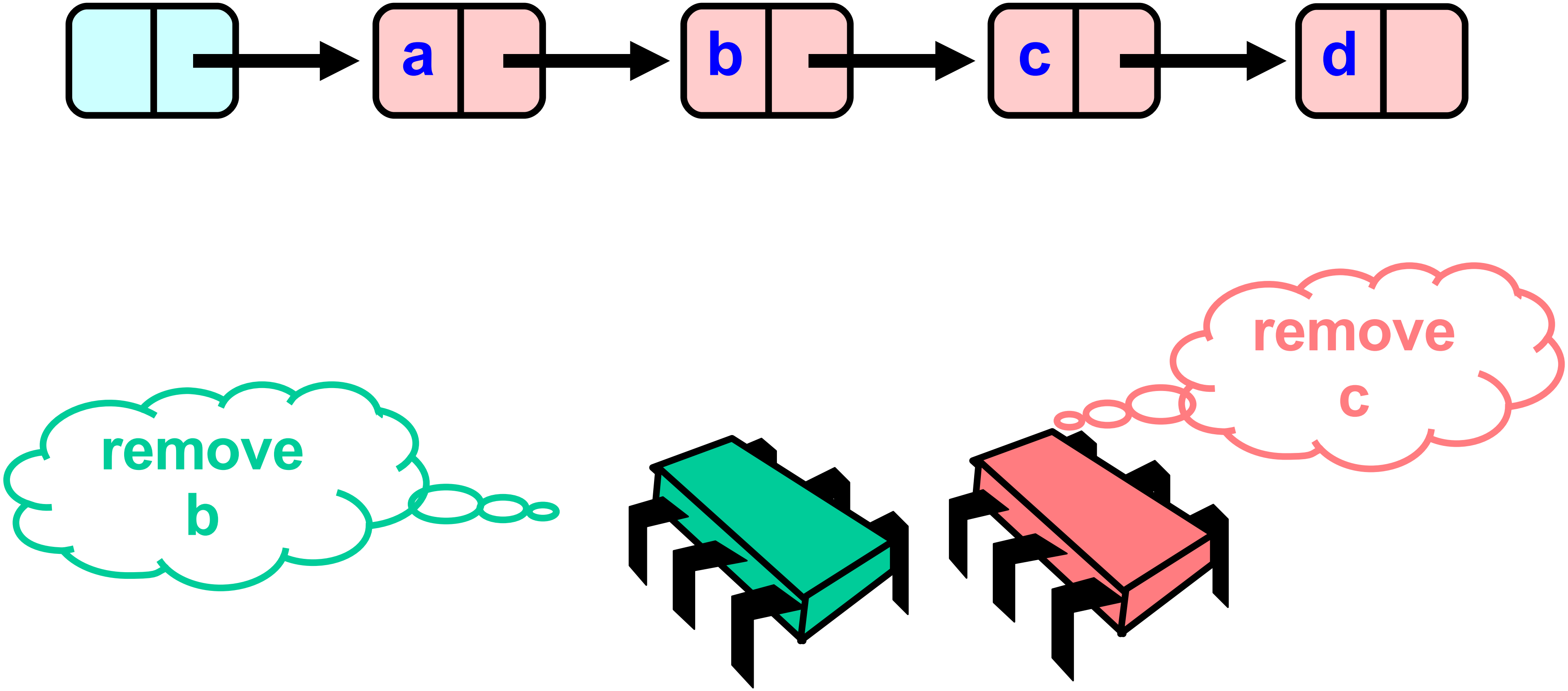
Adding a Node



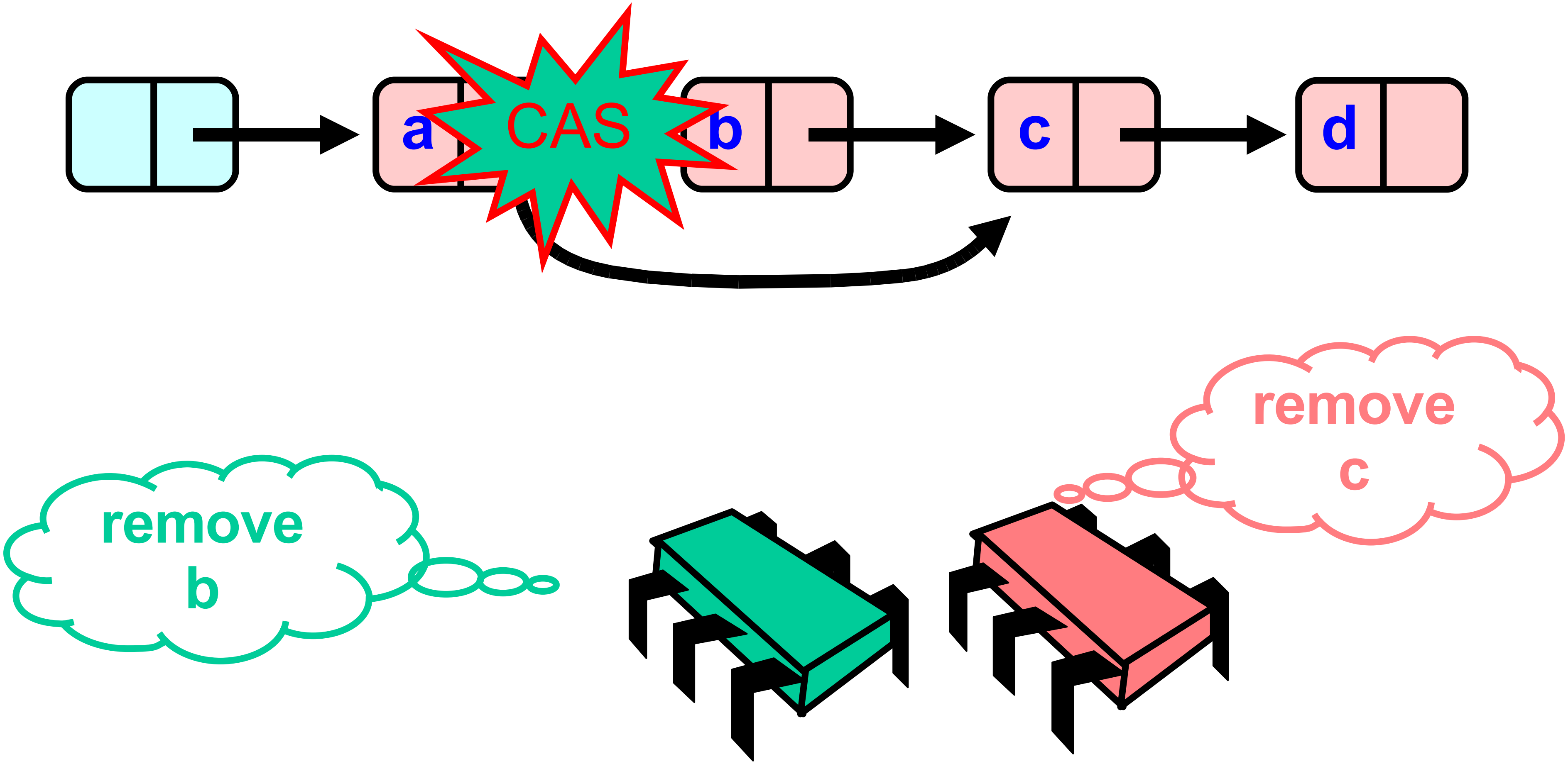
Adding a Node



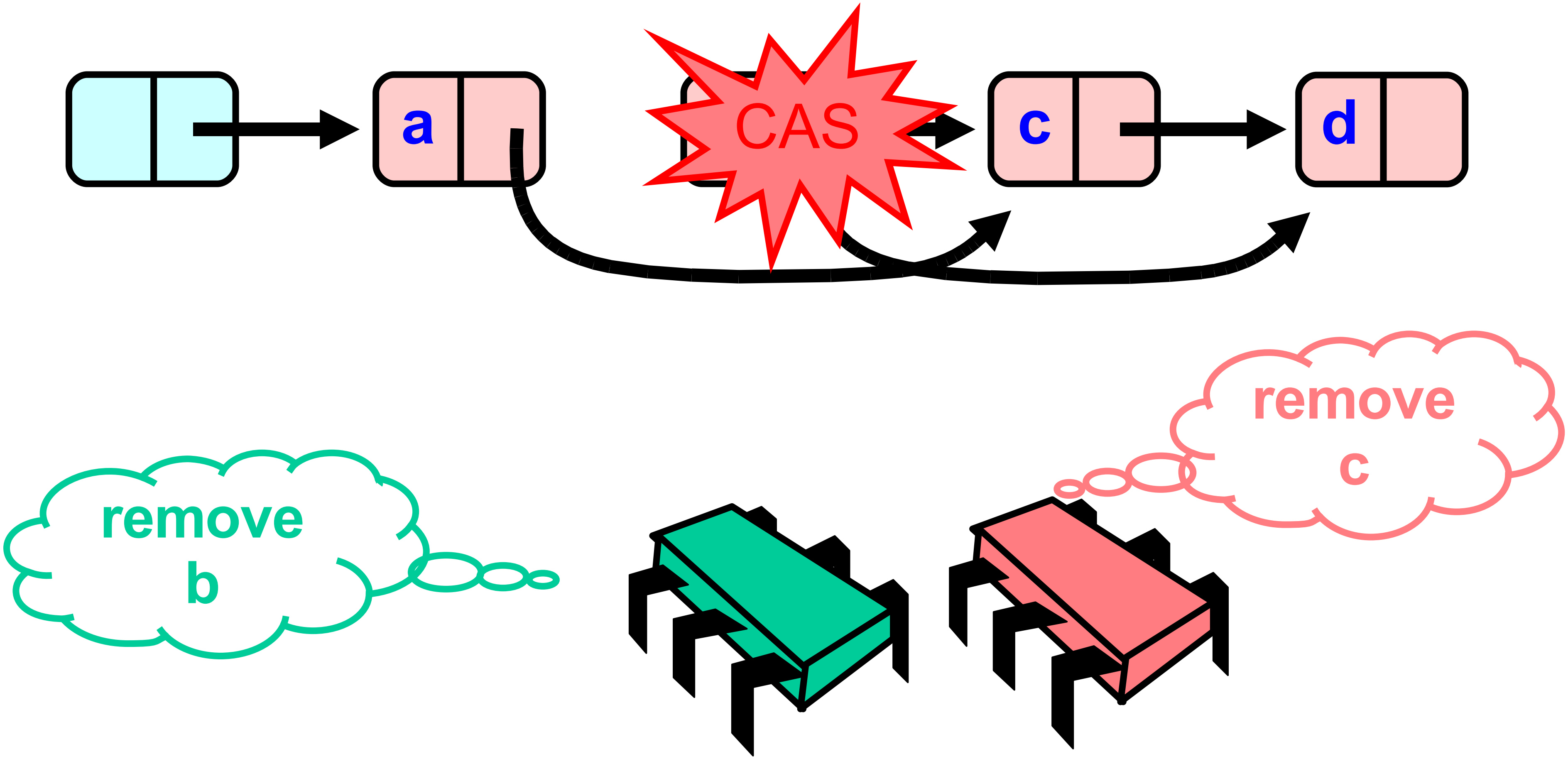
Removing a Node



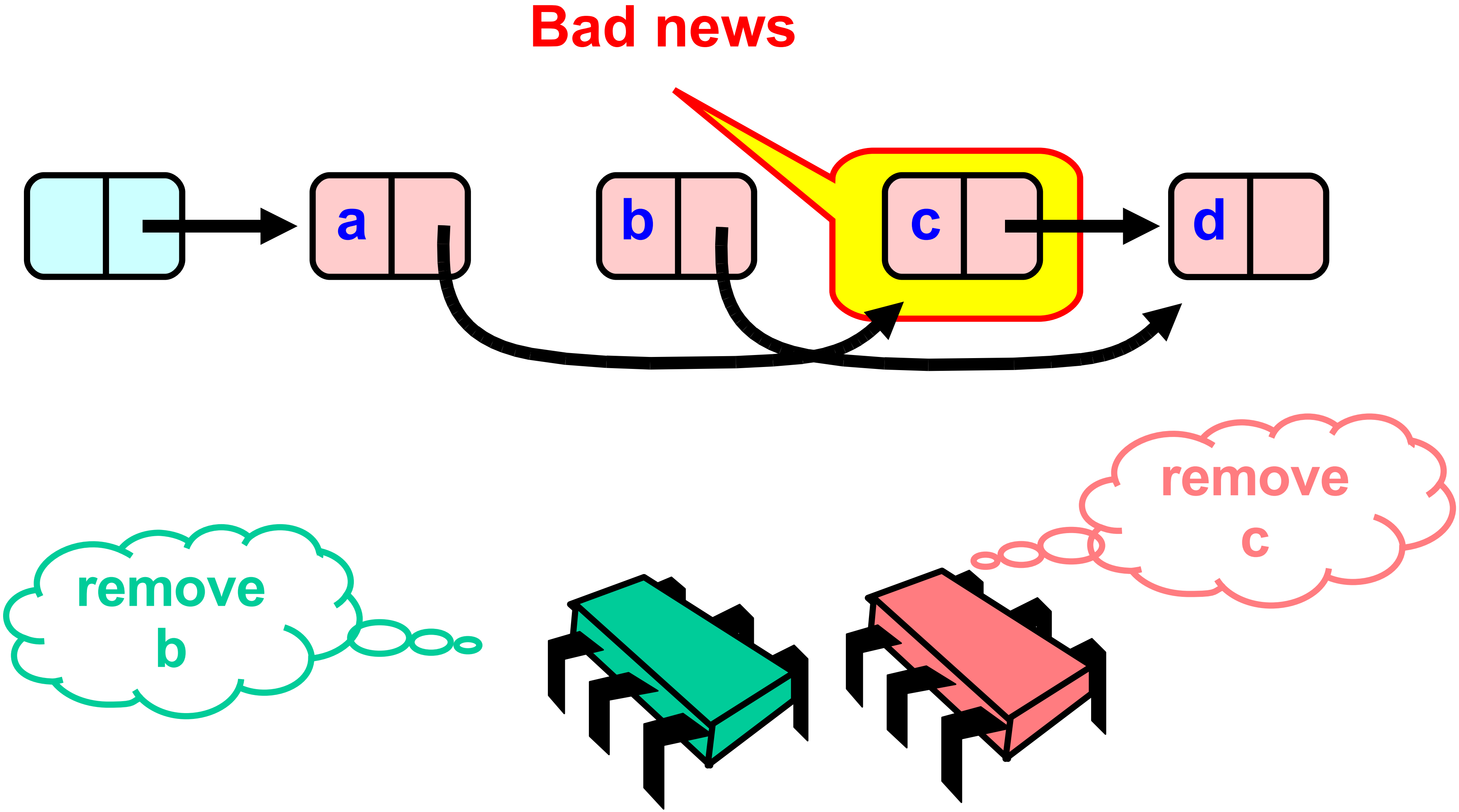
Removing a Node



Removing a Node



Look Familiar?



Lock-free list-based set

- Idea: Add “mark” to a node to indicate whether its key been removed from the set.
  - set mark before removing node from list
    - thus, if mark is not set, node is in the list
  - setting the mark removes key from the set
    - it is the serialization point of a successful remove operation
  - don't change next pointer of a marked node
    - mark and next pointer must be in the same word
      - “steal” a low-order bit from pointers
      - Java provides special class: AtomicMarkableReference

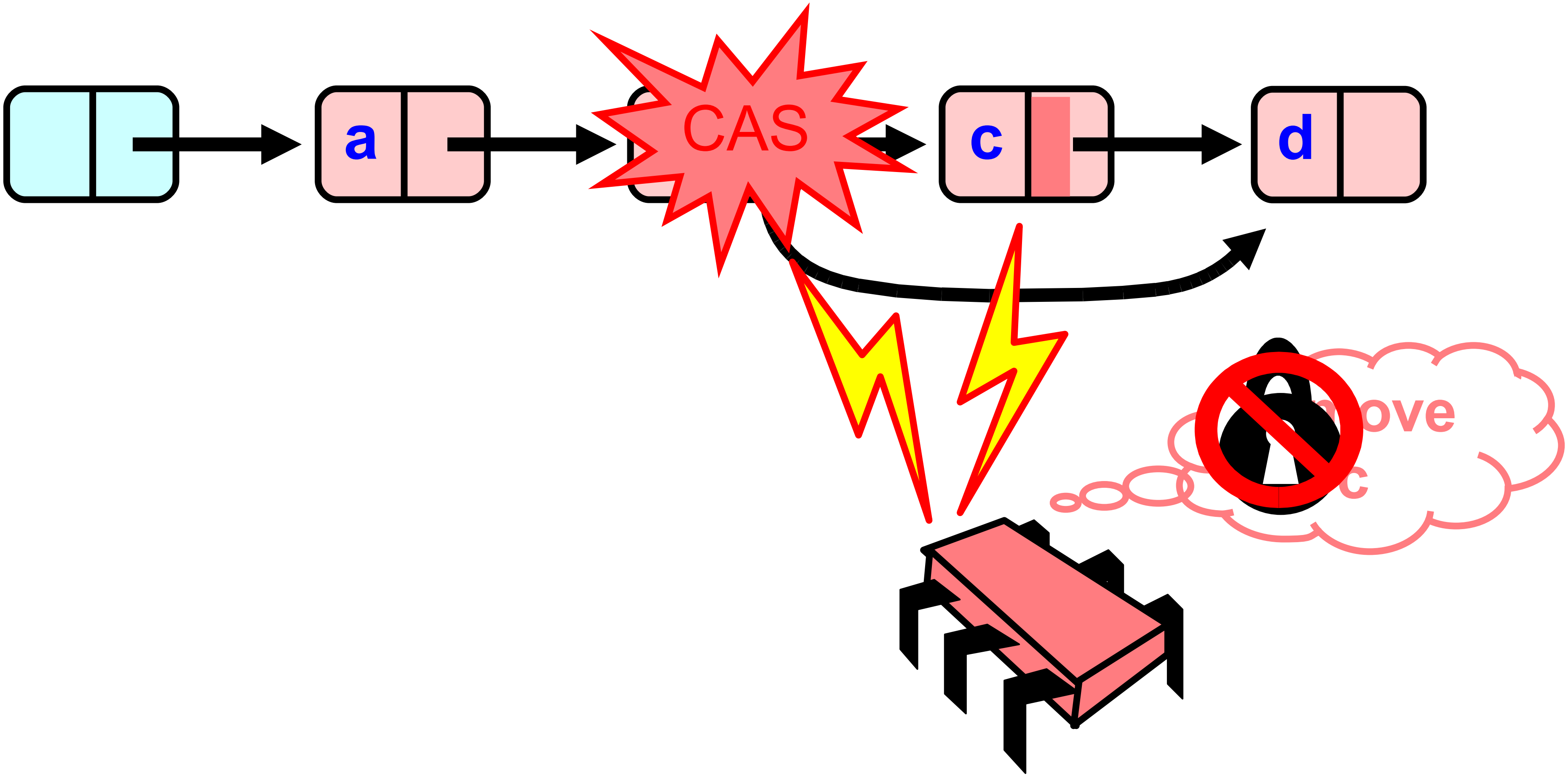
Lock-free list-based set

- Traverse the list to find appropriate nodes
  - what if we encounter marked nodes?
- If nodes are unmarked then operate as follows:
  - for contains(x) or unsuccessful add/remove(x), return appropriate value based on whether curr.key = x
  - for successful add(x), CAS pred.next+mark to (node, false)
  - for successful remove(x),
    - CAS curr.next+mark to (curr.next, true) [logical removal]
    - CAS pred.next+mark to (curr.next, false) [“physical” removal]
  - if (first) CAS fails, retry operation

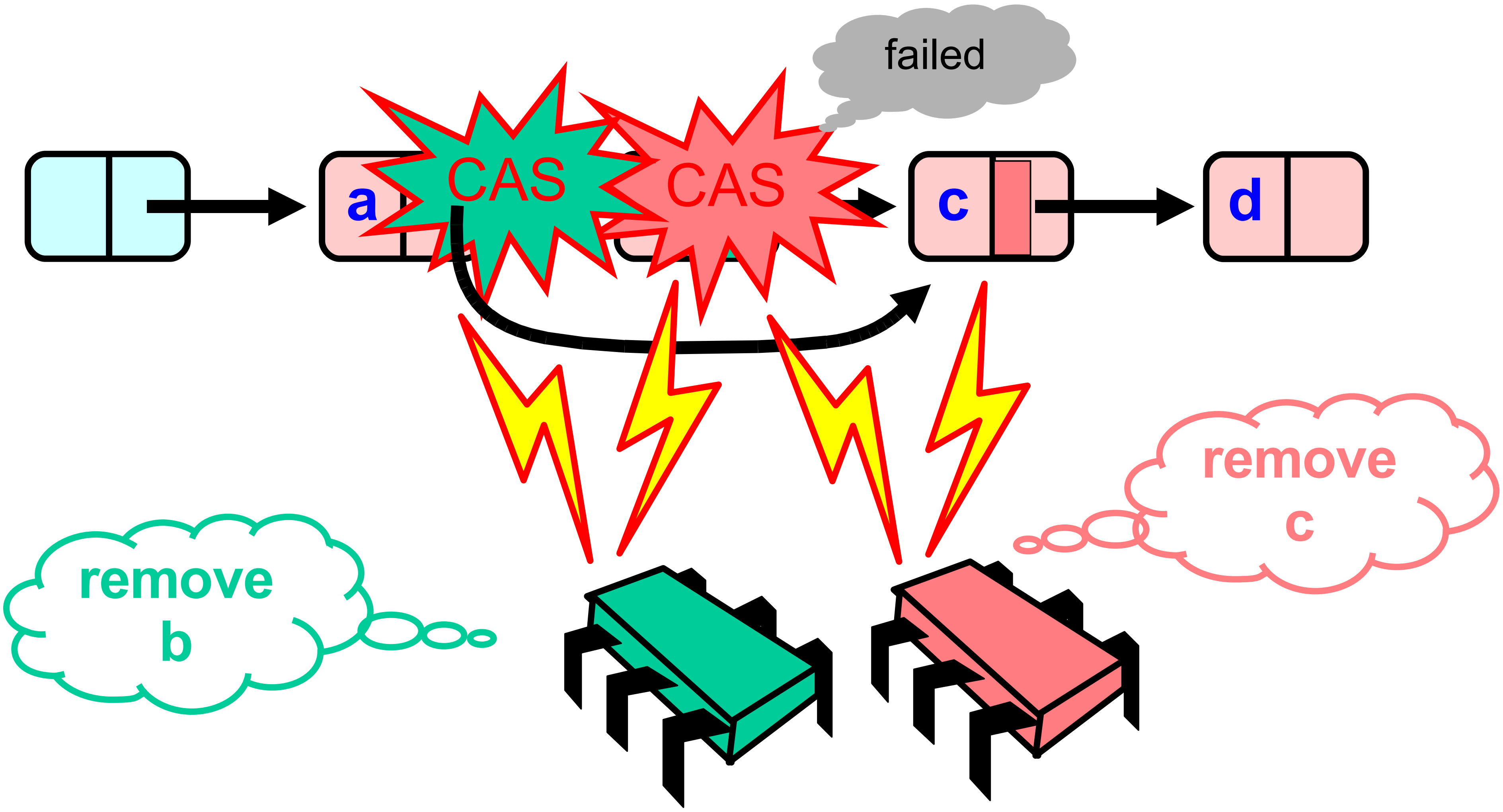
Lock-free list-based set

- What if we encounter marked nodes?
  - HELP!
  - if curr is marked, CAS pred.next+mark to (curr.next, false)
  - if CAS fails, retry operation
- This kind of helping is characteristic of lock-free and wait-free algorithms (not all have it, but most do).
  - next lecture, we'll see **obstruction-freedom**, a weaker condition that doesn't typically require helping.

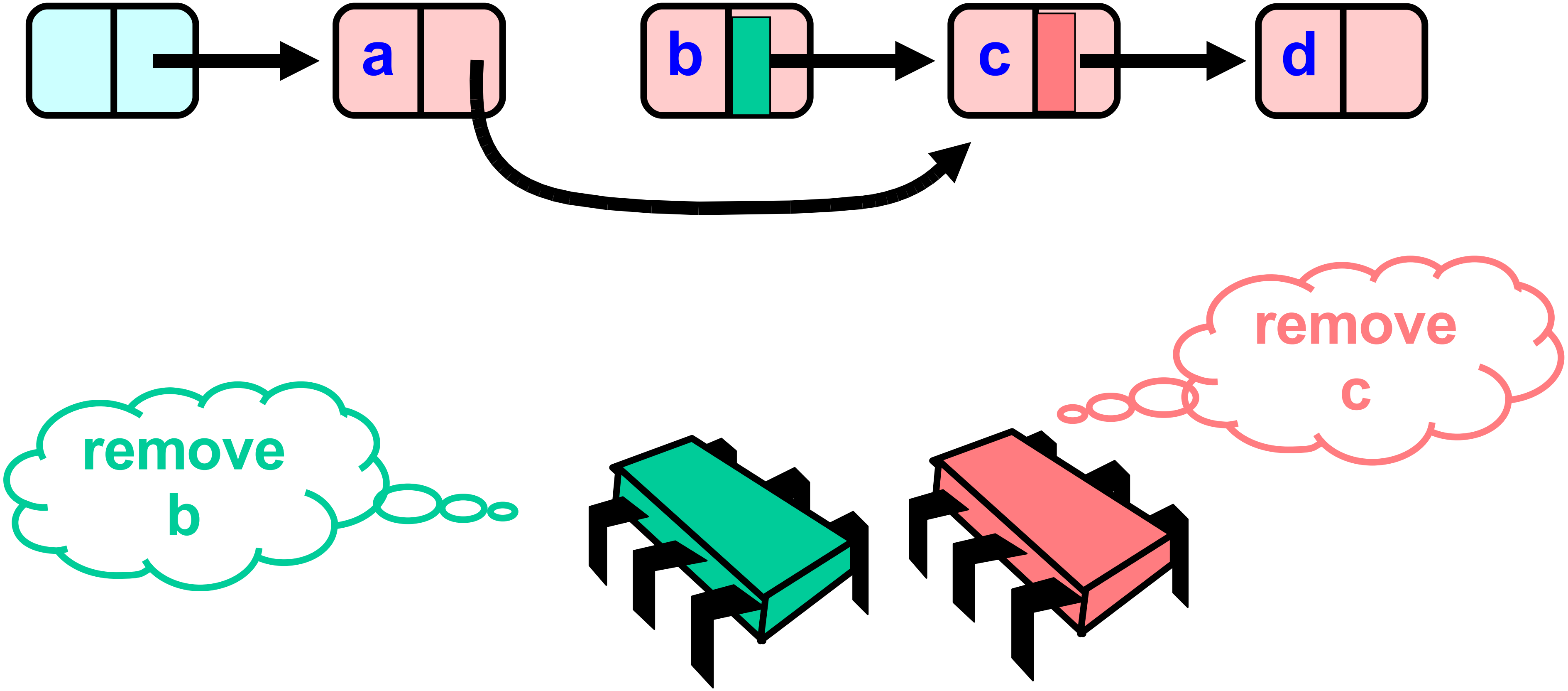
Removing a Node



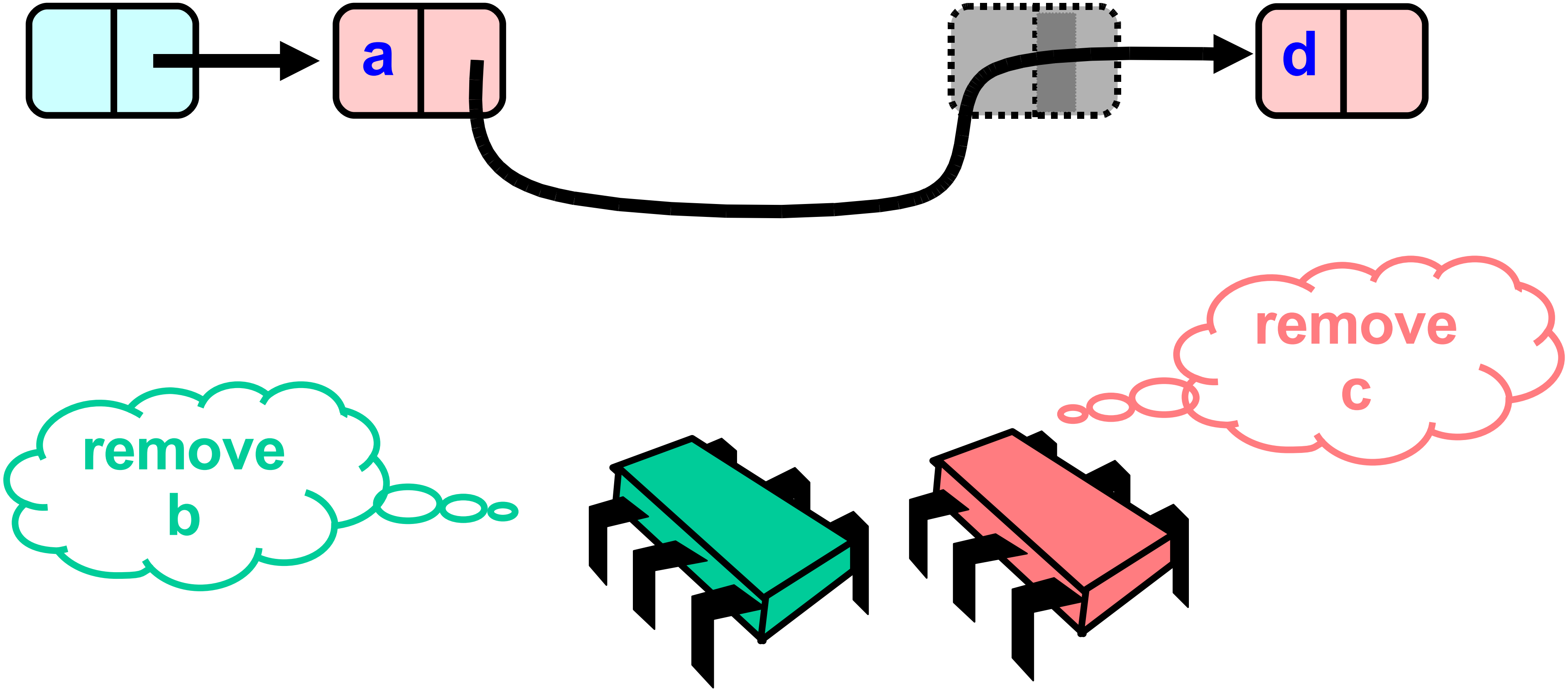
Removing a Node



Removing a Node



Removing a Node



Next time

- Transactional memory
- Reading:
  - Herlihy, Luchangco, Moir, Scherer paper
  - Dice, Shalev, Shavit paper