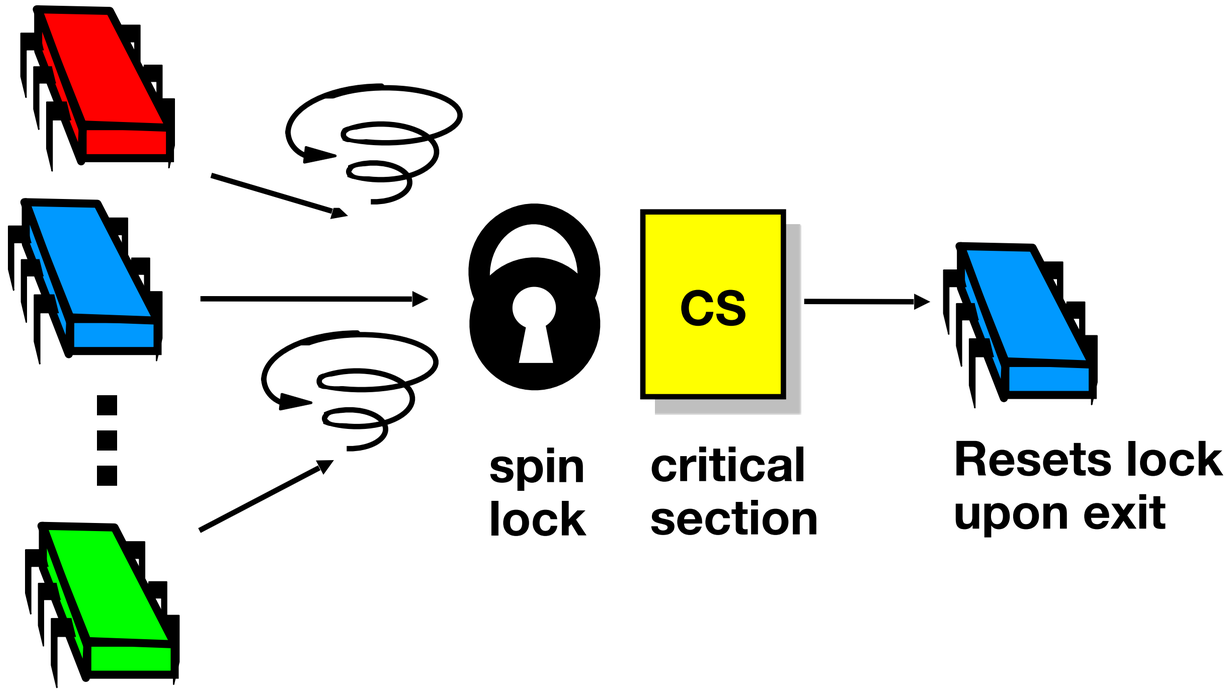
07 Linked Lists — Locking, Lock-free and beyond

CS 6868: Concurrent Programming

KC Sivaramakrishnan

Spring 2026, IIT Madras

In Previous Lectures: Spin Locks



Today: Concurrent Objects

Today: Concurrent Objects

- Adding threads should not **lower** throughput
  - Contention effects
  - Mostly fixed by scalable FIFO locks

Today: Concurrent Objects

- Adding threads should not **lower** throughput
  - Contention effects
  - Mostly fixed by scalable FIFO locks
- Should **increase** throughput
  - Not possible if inherently sequential
  - Surprising things are parallelizable

Coarse-Grained Synchronization

Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using scalable FIFO locks

Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using scalable FIFO locks
- Easy to reason about
  - In Simple Cases

Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using scalable FIFO locks
- Easy to reason about
  - In Simple Cases
- ***So, are we done?***

Coarse-Grained Synchronization

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

This Lecture

- **Introduce four “patterns”**
  - Bag of tricks …
  - Methods that work more than once …

This Lecture

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

(1) Fine-grained synchronization

(1) Fine-grained synchronization

- Instead of using a single lock …

(1) Fine-grained synchronization

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

(1) 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

(1) 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
- See `fine_list.ml`

(2) Optimistic synchronization

(2) Optimistic synchronization

- Search without locking …

(2) Optimistic synchronization

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

(2) 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

(2) 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
- See `optimistic_list.ml`

(3) Lazy synchronization

(3) Lazy synchronization

- Postpone hard work

(3) Lazy synchronization

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

(3) Lazy synchronization

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

(4) Lock-free synchronization

(4) Lock-free synchronization

- Don’t use locks at all
  - Use `compareAndSet()` & relatives …

(4) Lock-free synchronization

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

(4) Lock-free synchronization

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

(4) Lock-free synchronization

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

Linked List

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

Set Interface

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

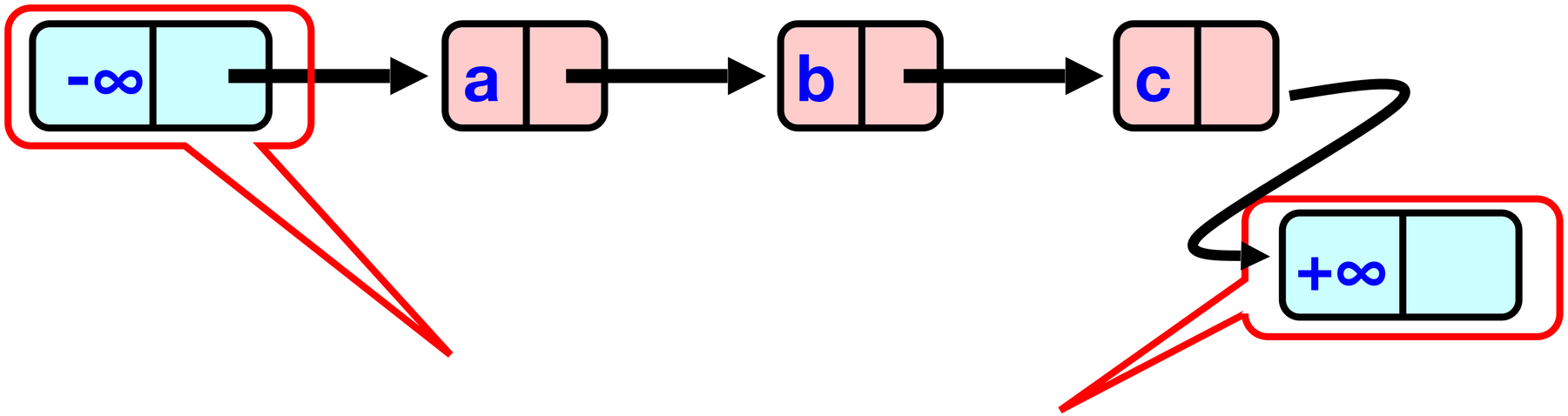
List-based Sets

```
type 'a t
(** The type of a coarse-grained list containing elements of type ['a] *)

val create : unit -> 'a t
(** [create ()] creates a new empty list with sentinel nodes *)

val add : 'a t -> 'a -> bool
(** [add list item] adds [item] to the list. Returns [true] if the element was newly
    added, [false] if it was already present. *)

val remove : 'a t -> 'a -> bool
(** [remove list item] removes [item] from the list. Returns [true] if the element
    was present and removed, [false] if it was not present. *)

val contains : 'a t -> 'a -> bool
(** [contains list item] tests whether [item] is present in the list.
    Returns [true] if present, [false] otherwise. *)
```

List representation

```
(** Internal node representation *)
type 'a node = {
  item : 'a option;          (* None for sentinel nodes *)
  key : int;                 (* hash code for the item, or min_int/max_int for sentinels *)
  mutable next : 'a node;    (* next node in the list, tail points to itself *)
}

(** Seqentual list type *)
type 'a t = {
  head : 'a node;           (* sentinel node at the start *)
}
```

List-based Set



Reasoning about correctness

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

Specifically…

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

Interference

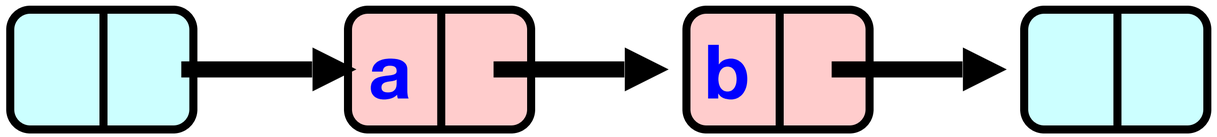
- Invariants make sense only if
  - methods considered
  - are the only modifiers
- Language encapsulation helps
  - List nodes not visible outside the OCaml module
  - Encapsulation may be provided by classes in other langauges

Interference

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

Abstract Data Type

- Concrete representation:

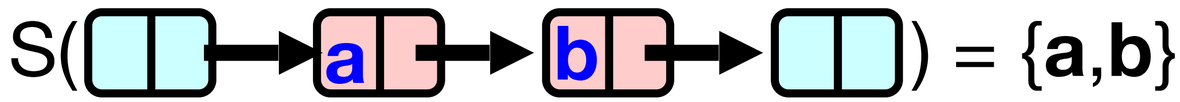


- Abstract Type:

  {**a**, **b**}

Abstract Data Types

- Meaning of rep given by abstraction map



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 is incorrect?
- If rep invariant says *no duplicates*
  - **`add()`** is incorrect
- Otherwise,
  - **`remove()`** is incorrect

Rep Invariant (partly)

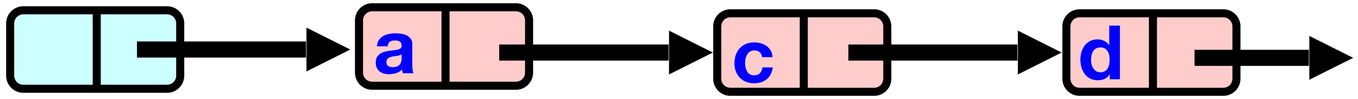
- Sentinel nodes
  - tail reachable from head
- Sorted
- No duplicates

Abstraction Map

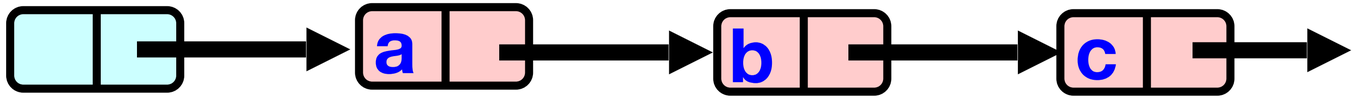
$$S(head) = \{x \mid \exists a \,.\, a \text{ is reachable from } head \wedge a \,.\, item = x\}$$

Sequential List-based Set

`add()`



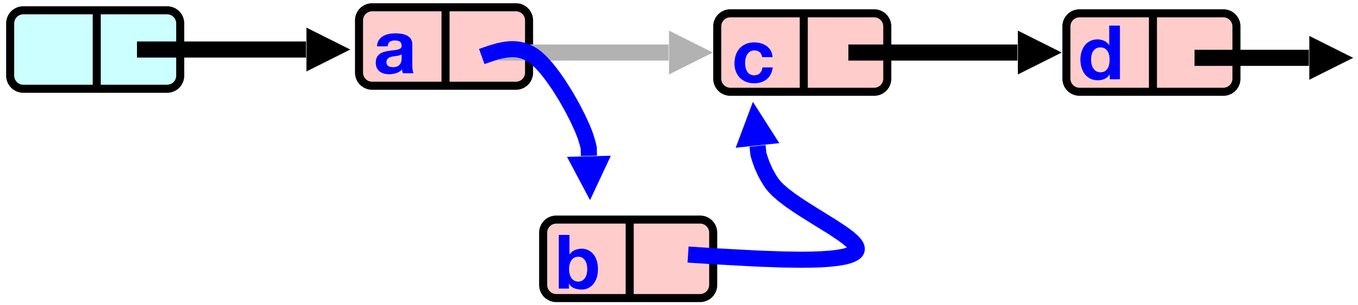
`remove()`



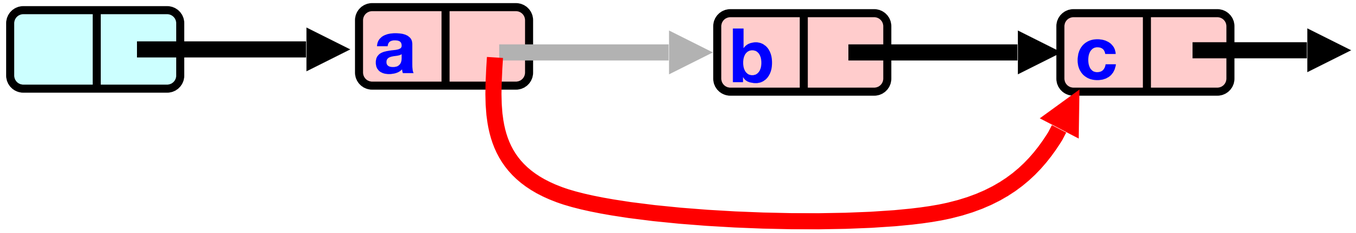
See seq_list.ml

Sequential List-based Set

`add()`

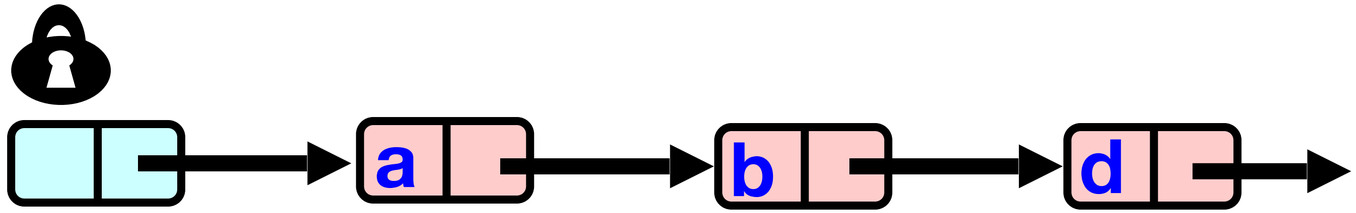


`remove()`

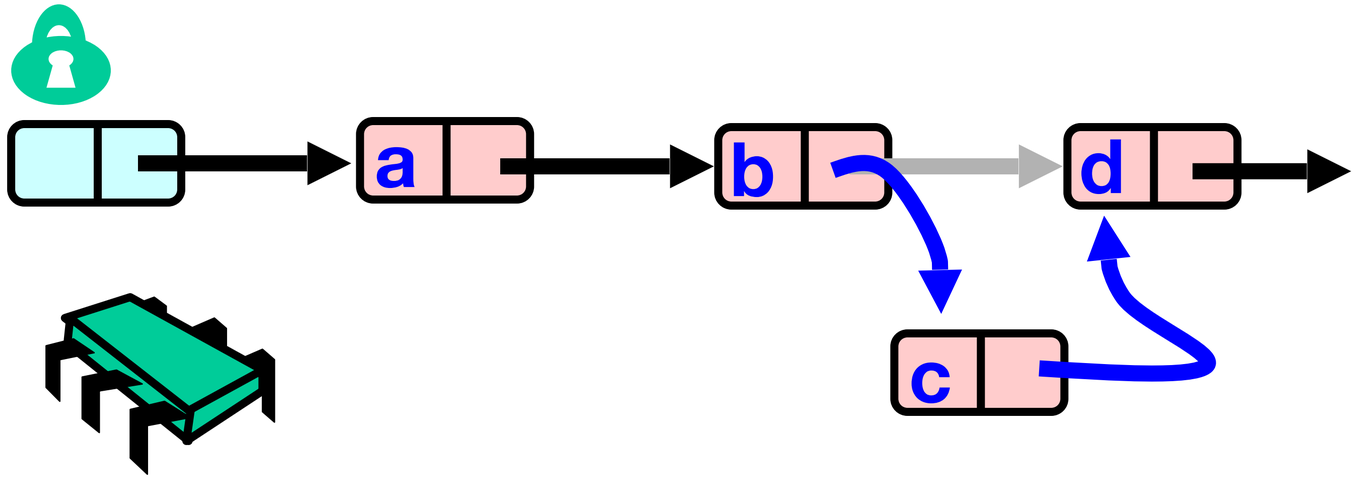


Coarse-grained Locking

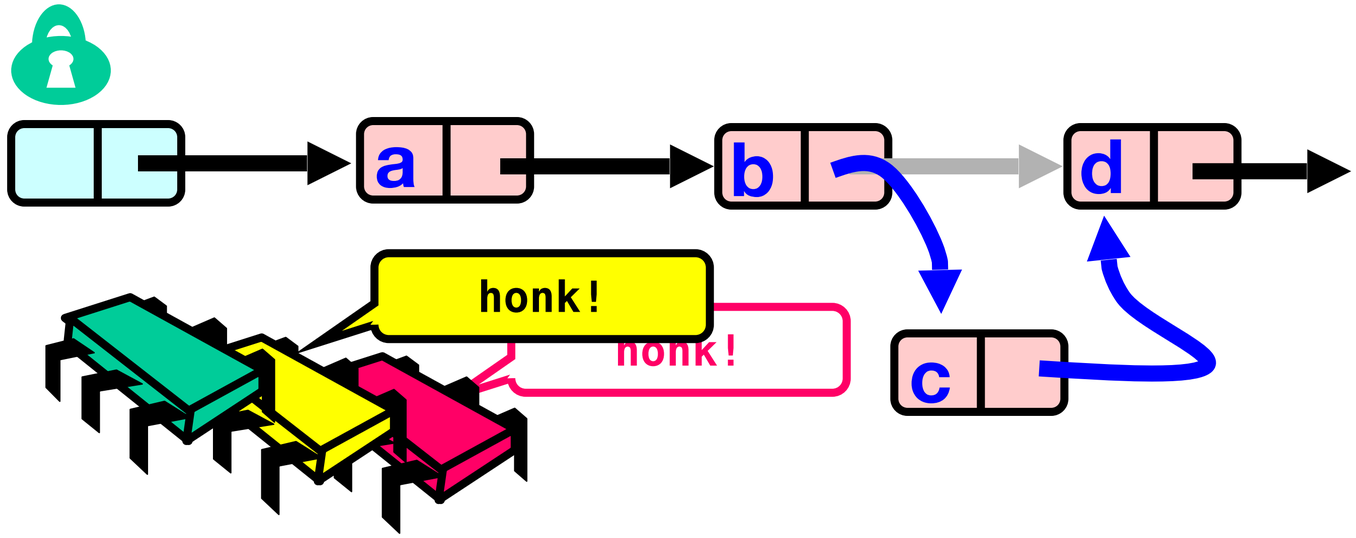
Coarse-grained locking



Coarse-grained locking

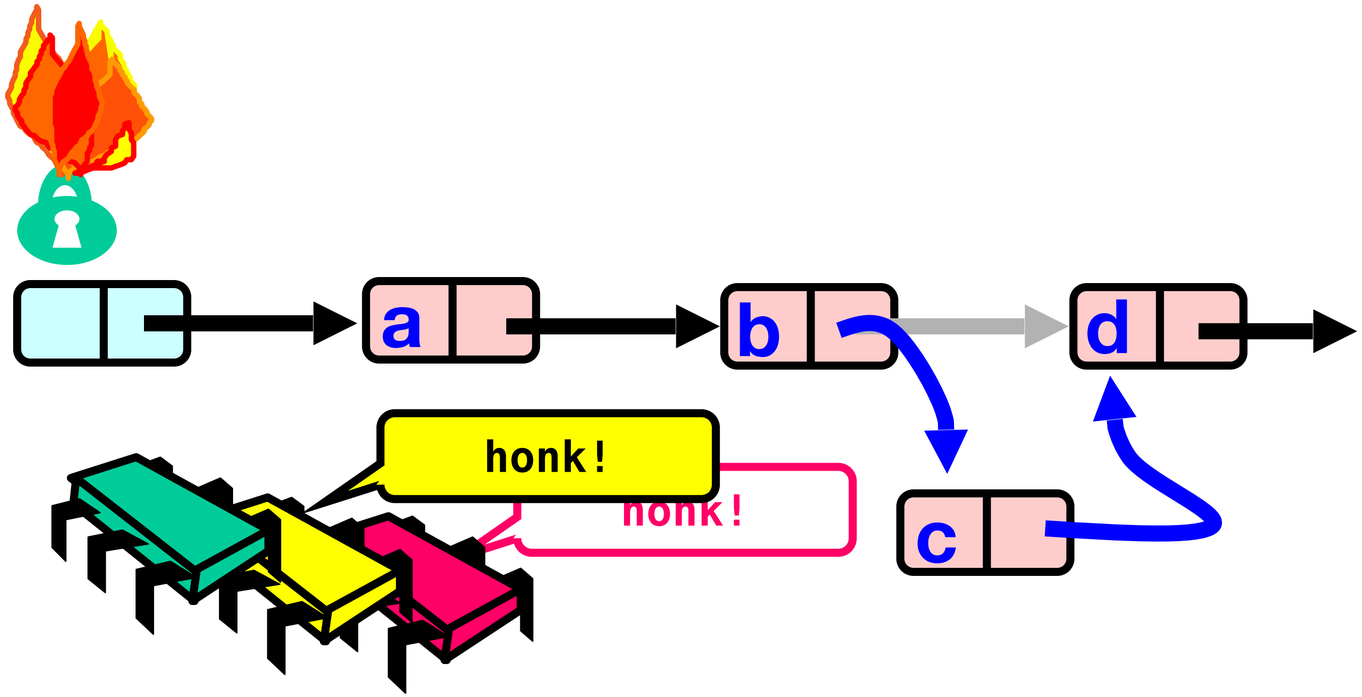


Coarse-grained locking



See coarse_list.ml

Coarse-grained locking



Simple but hotspot + bottleneck

See coarse_list.ml

Coarse-grained locking

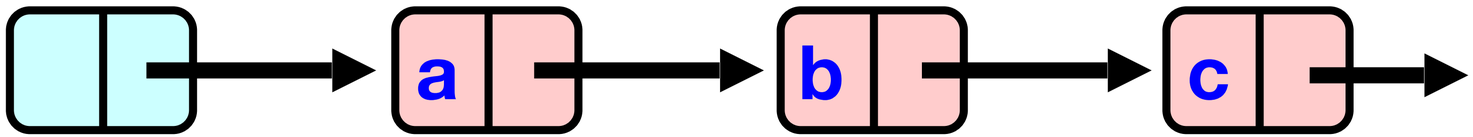
- Easy, same as synchronized methods
  - *“One lock to rule them all …”*
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention
  - Queue locks help
  - But bottleneck still an issue

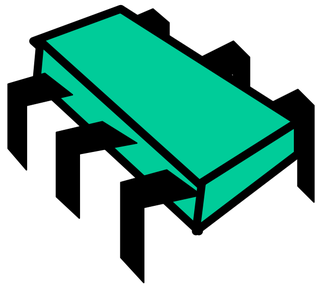
(1) Fine-grained Locking

Fine-grained locking

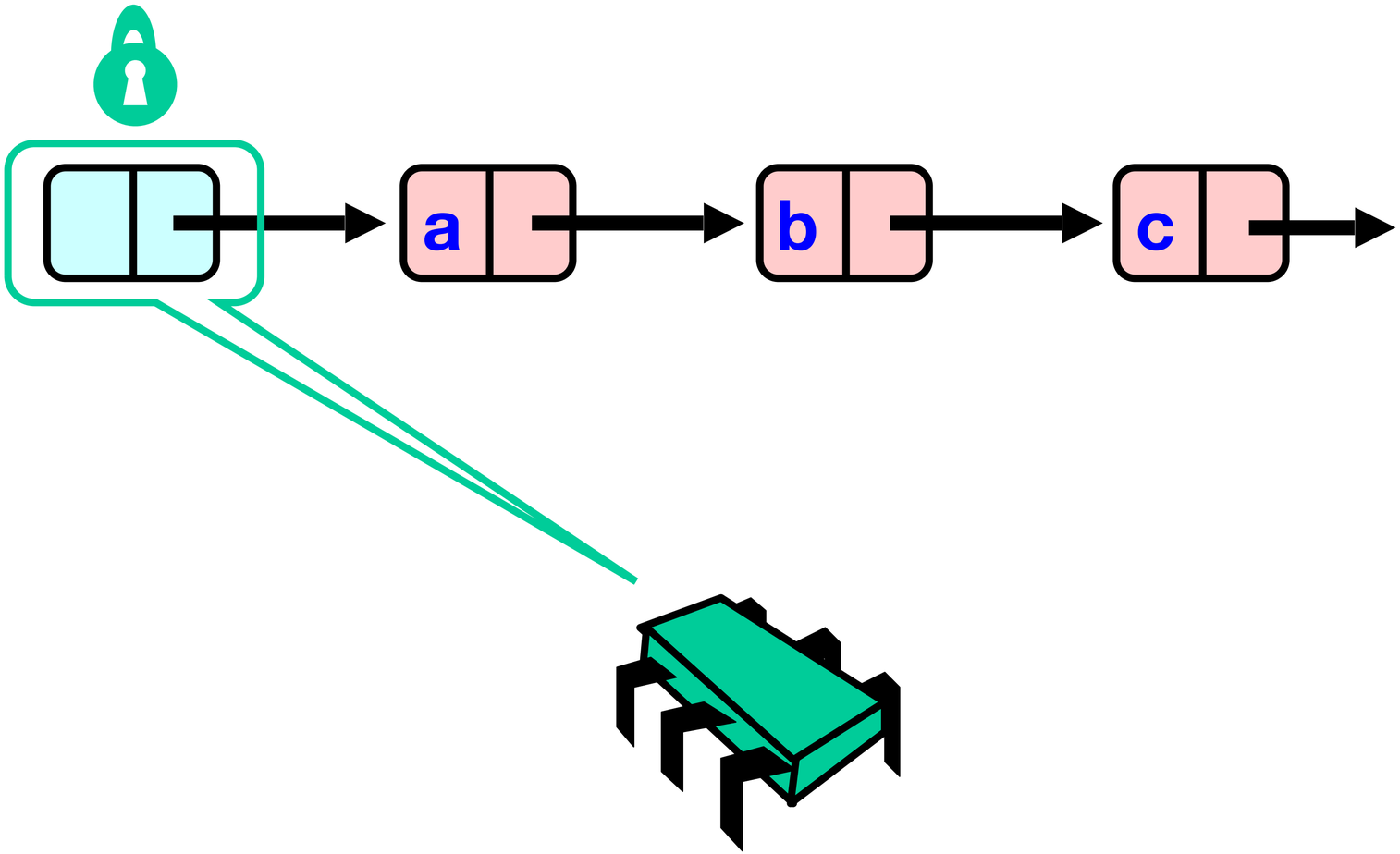
- Requires **careful** thought
  - **“Do not meddle in the affairs of wizards, for they are subtle and quick to anger”**
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Hand-over-hand locking

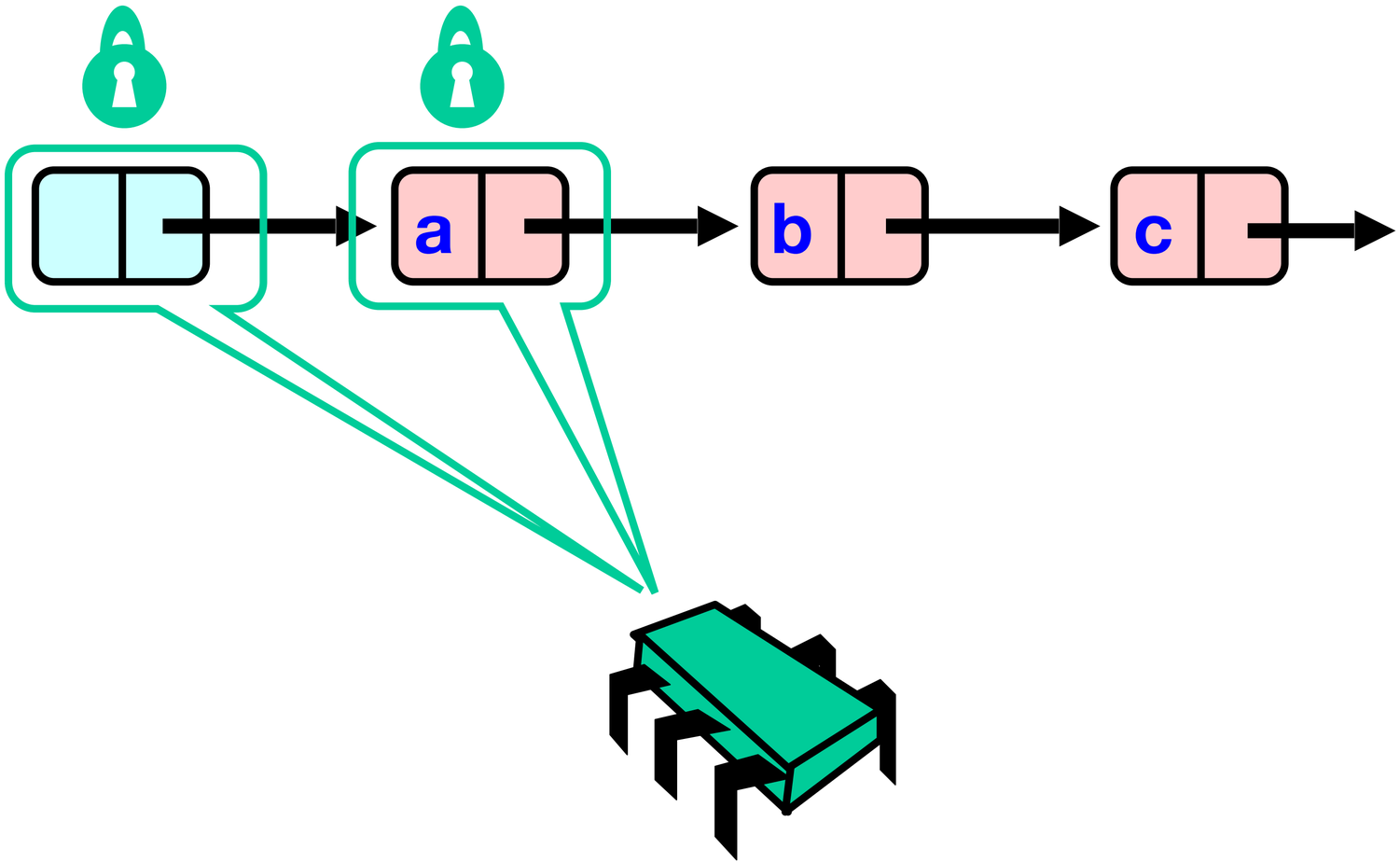




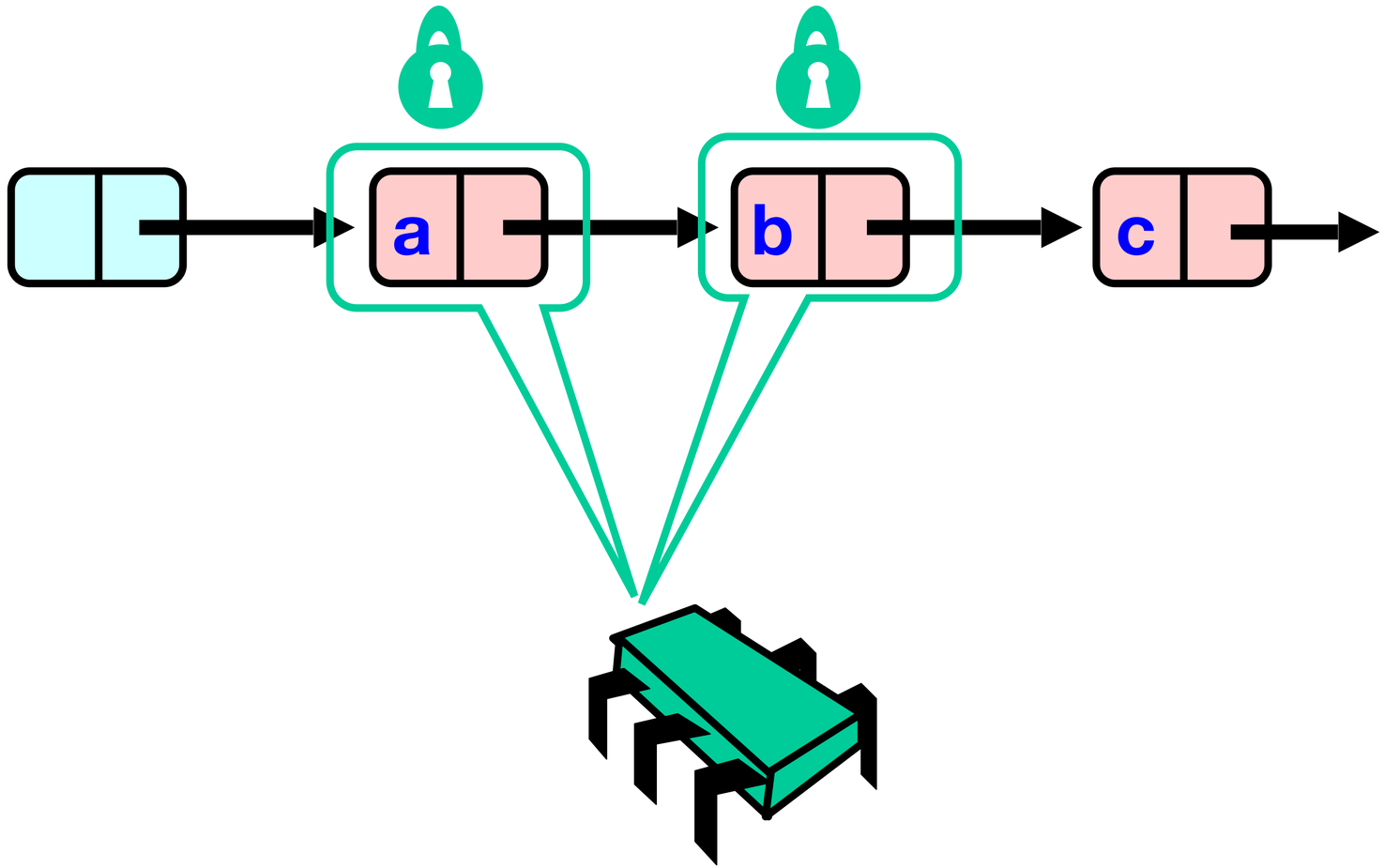
Hand-over-hand locking



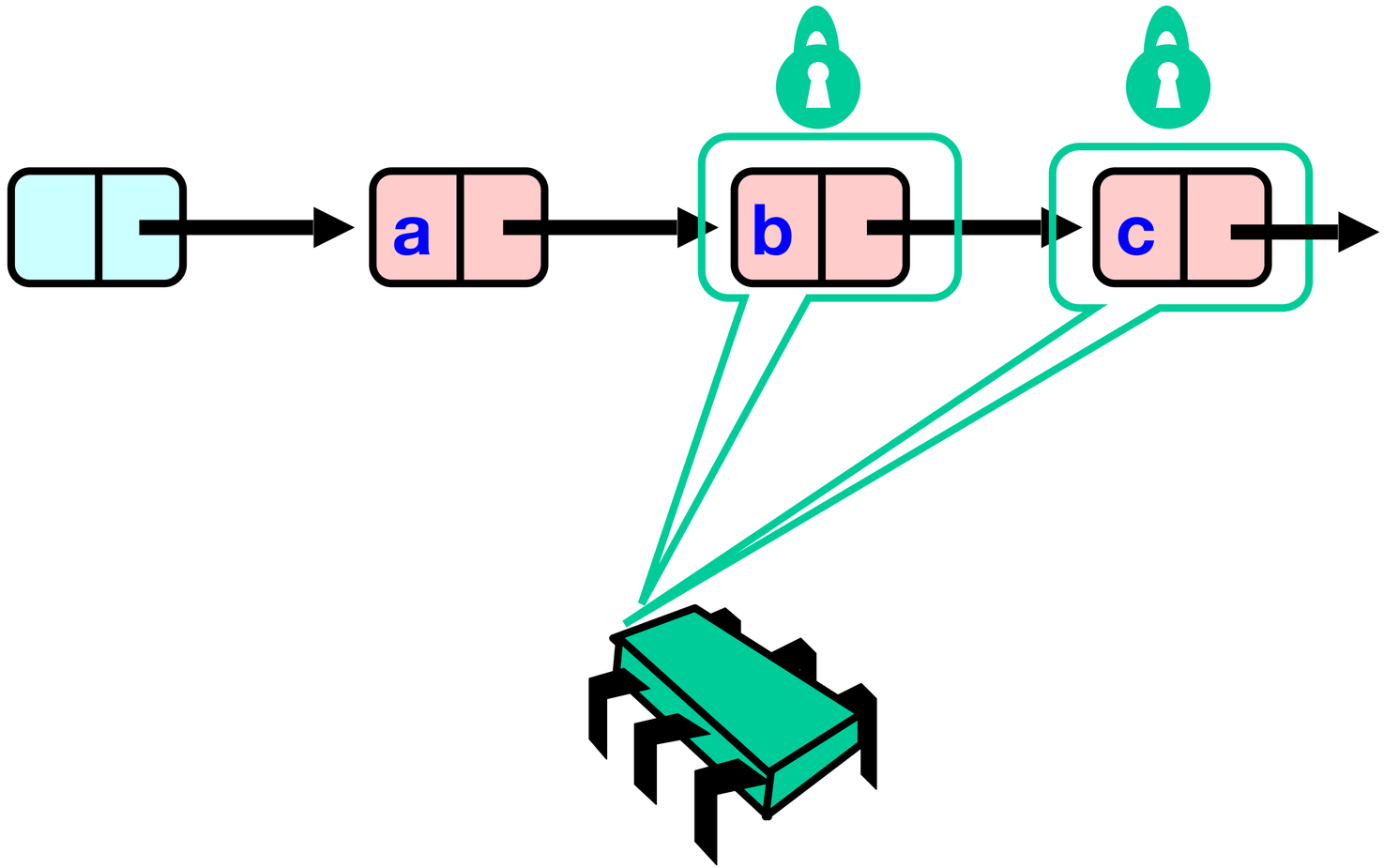
Hand-over-hand locking



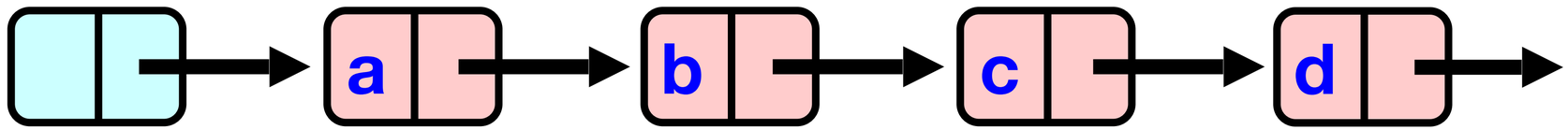
Hand-over-hand locking

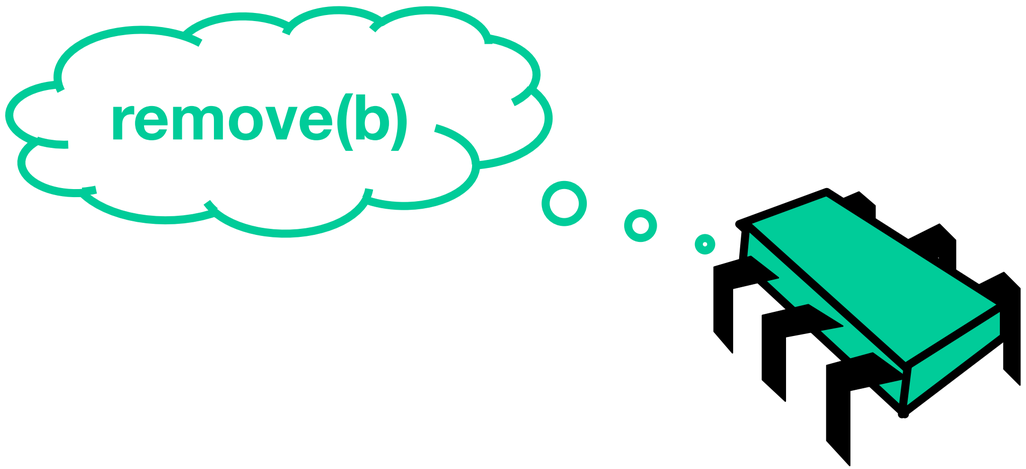


Hand-over-hand locking

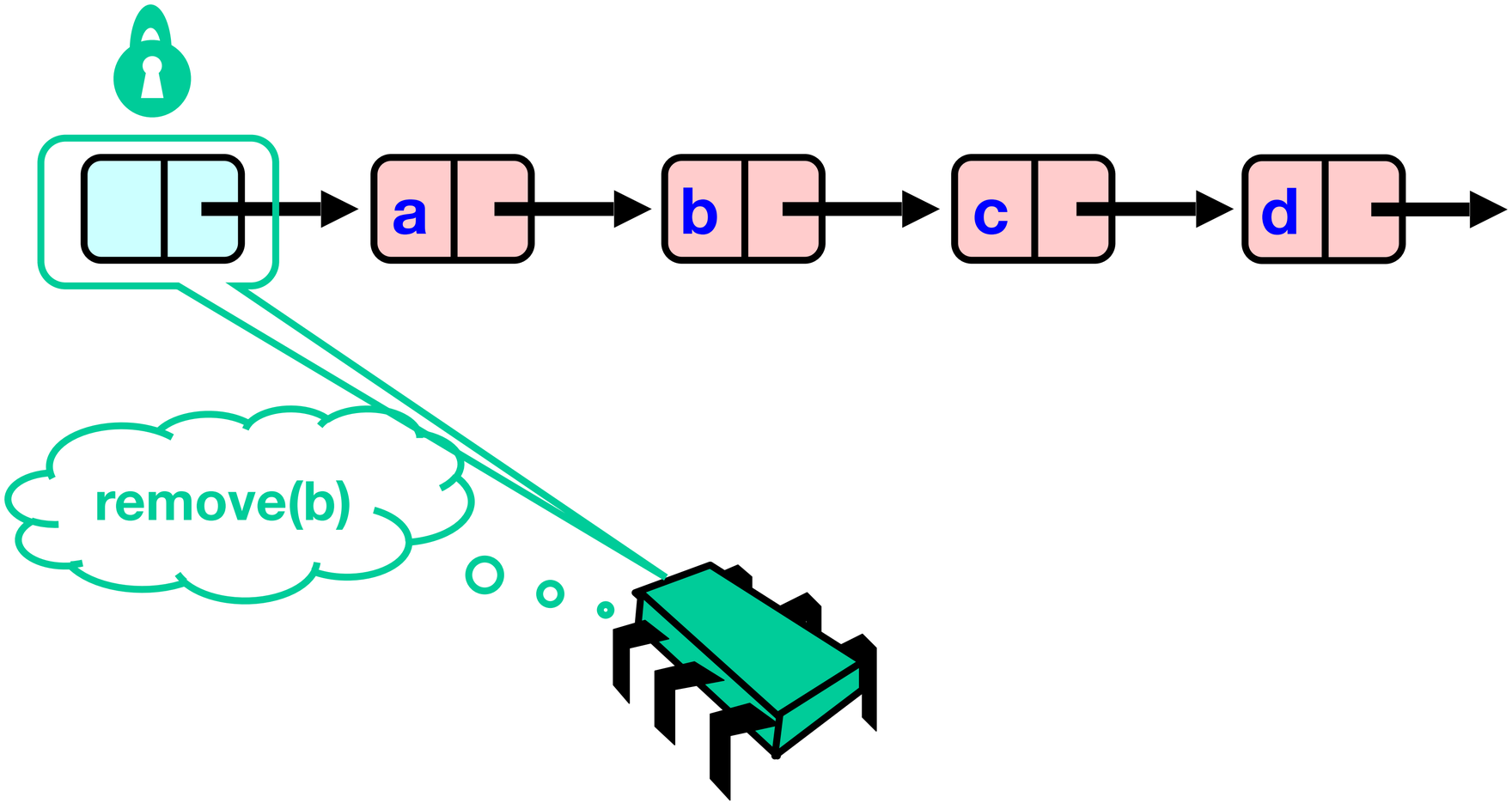


Removing a node

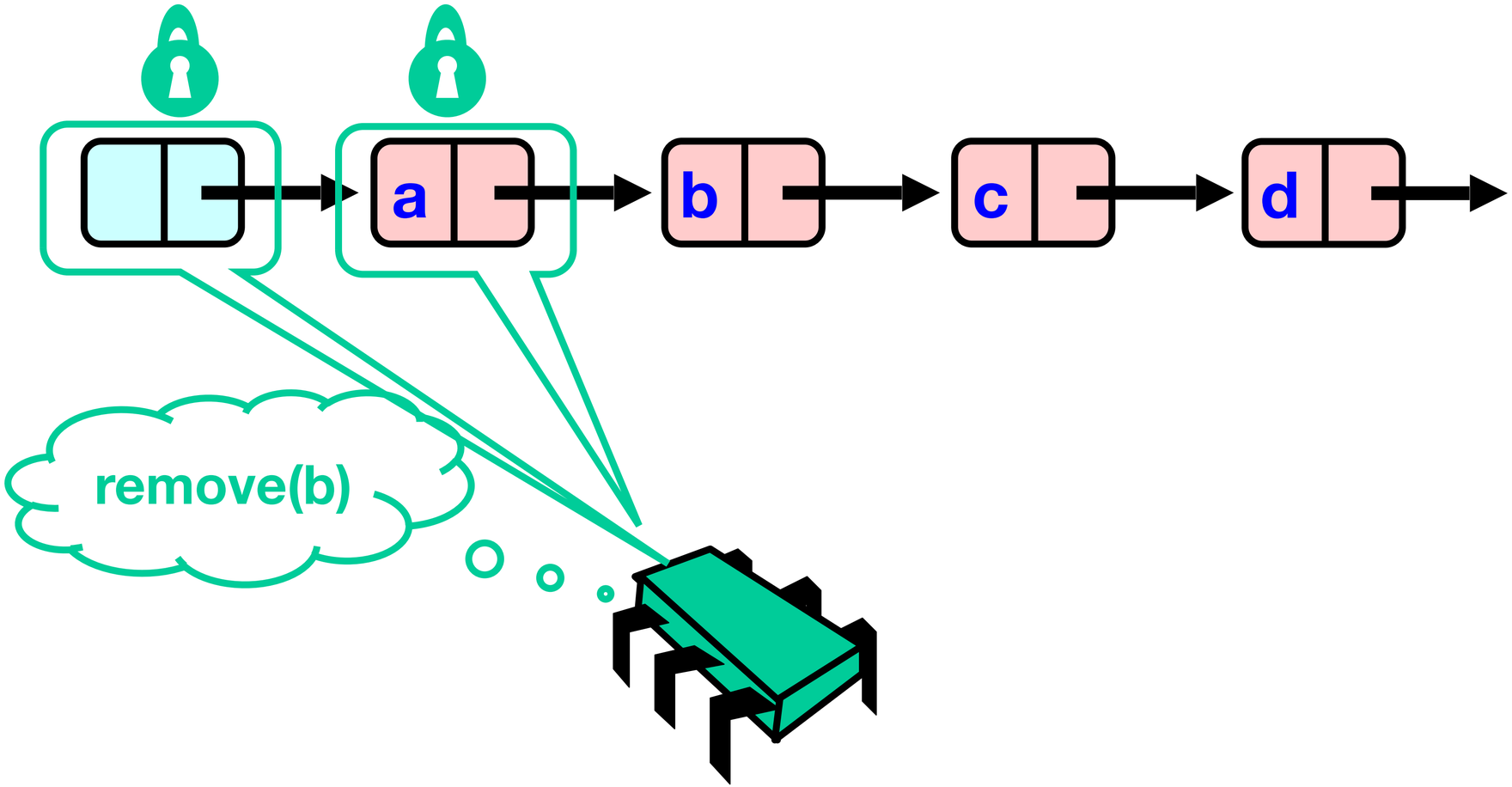




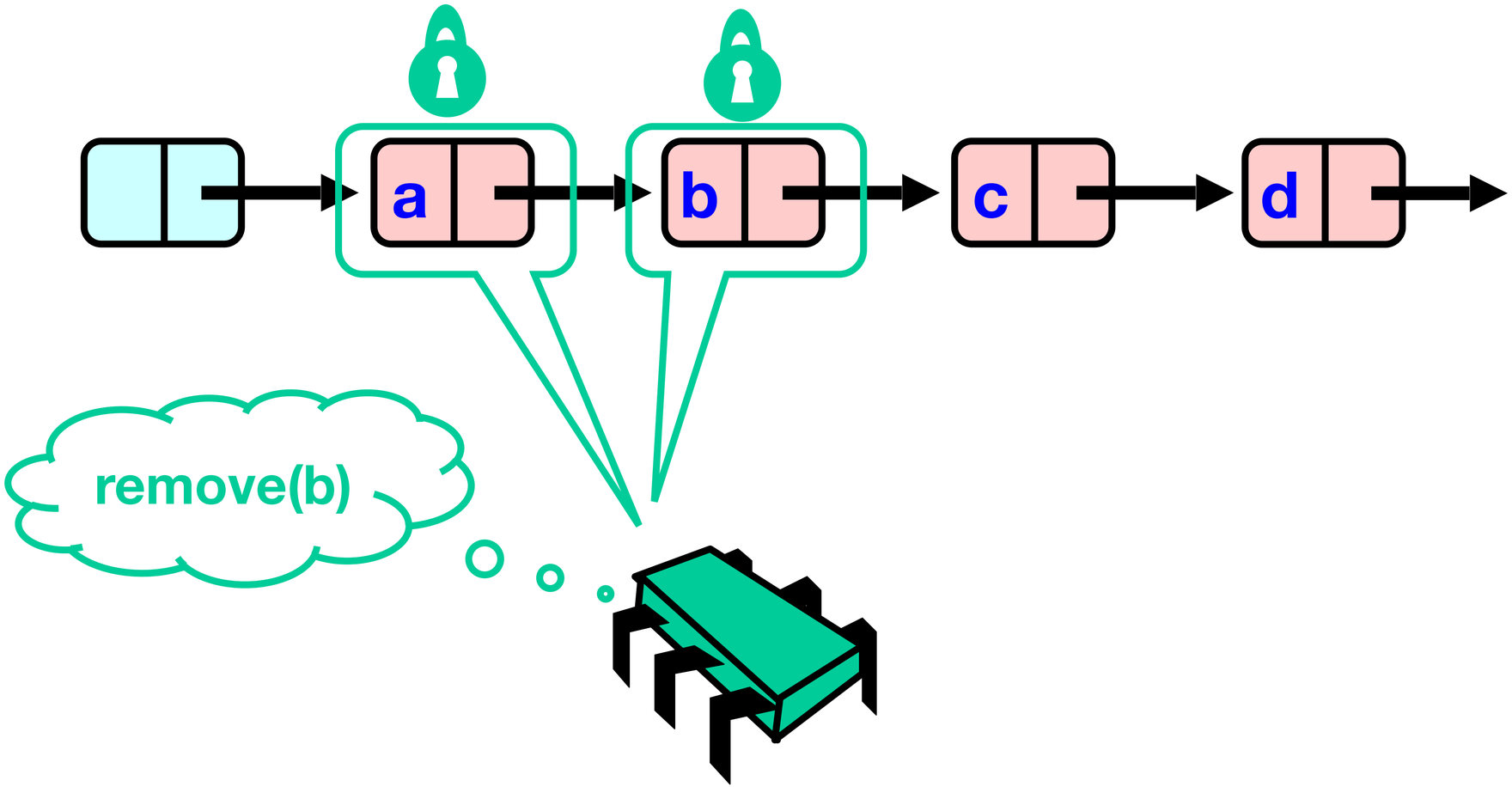
Removing a node



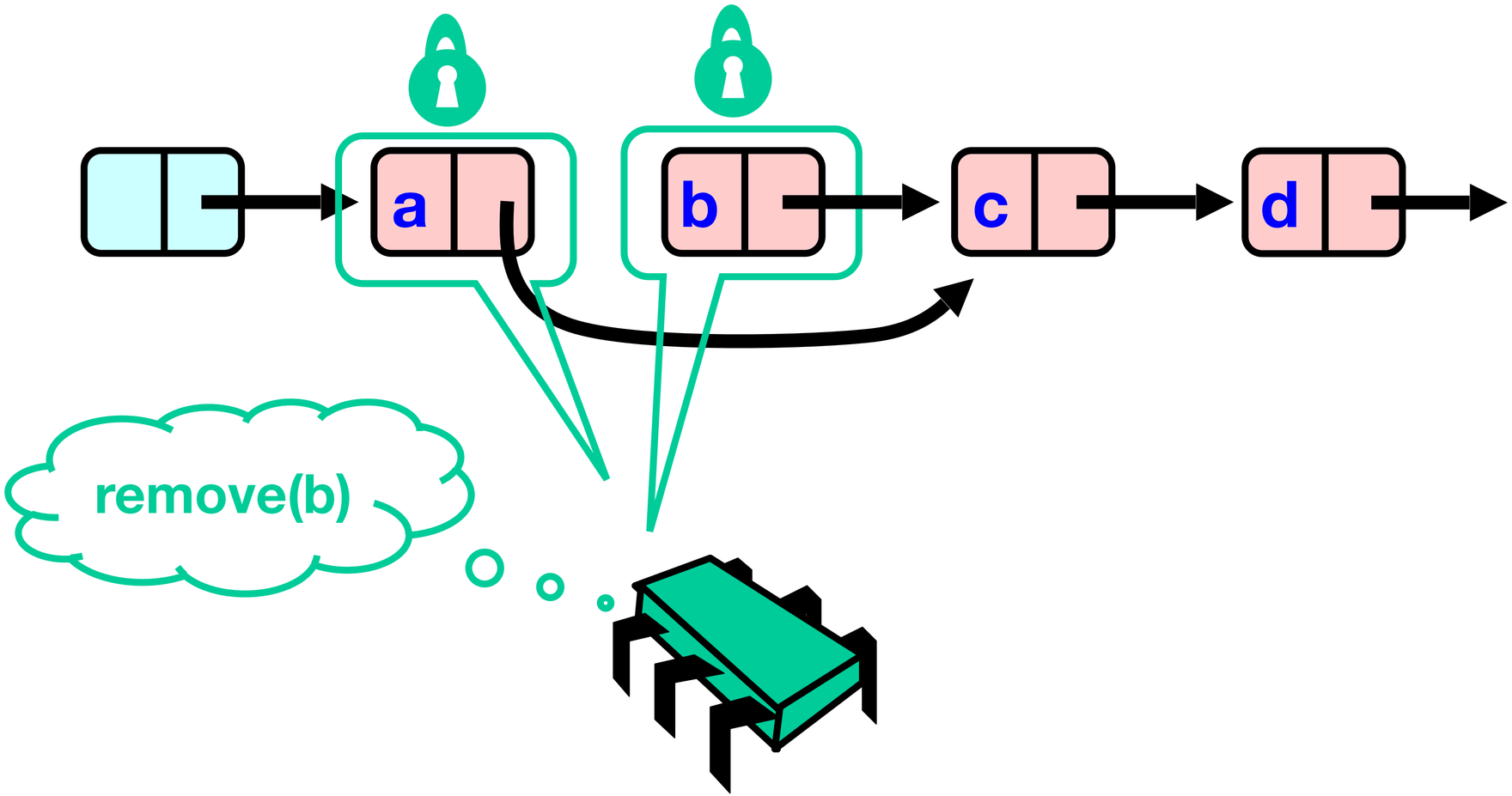
Removing a node



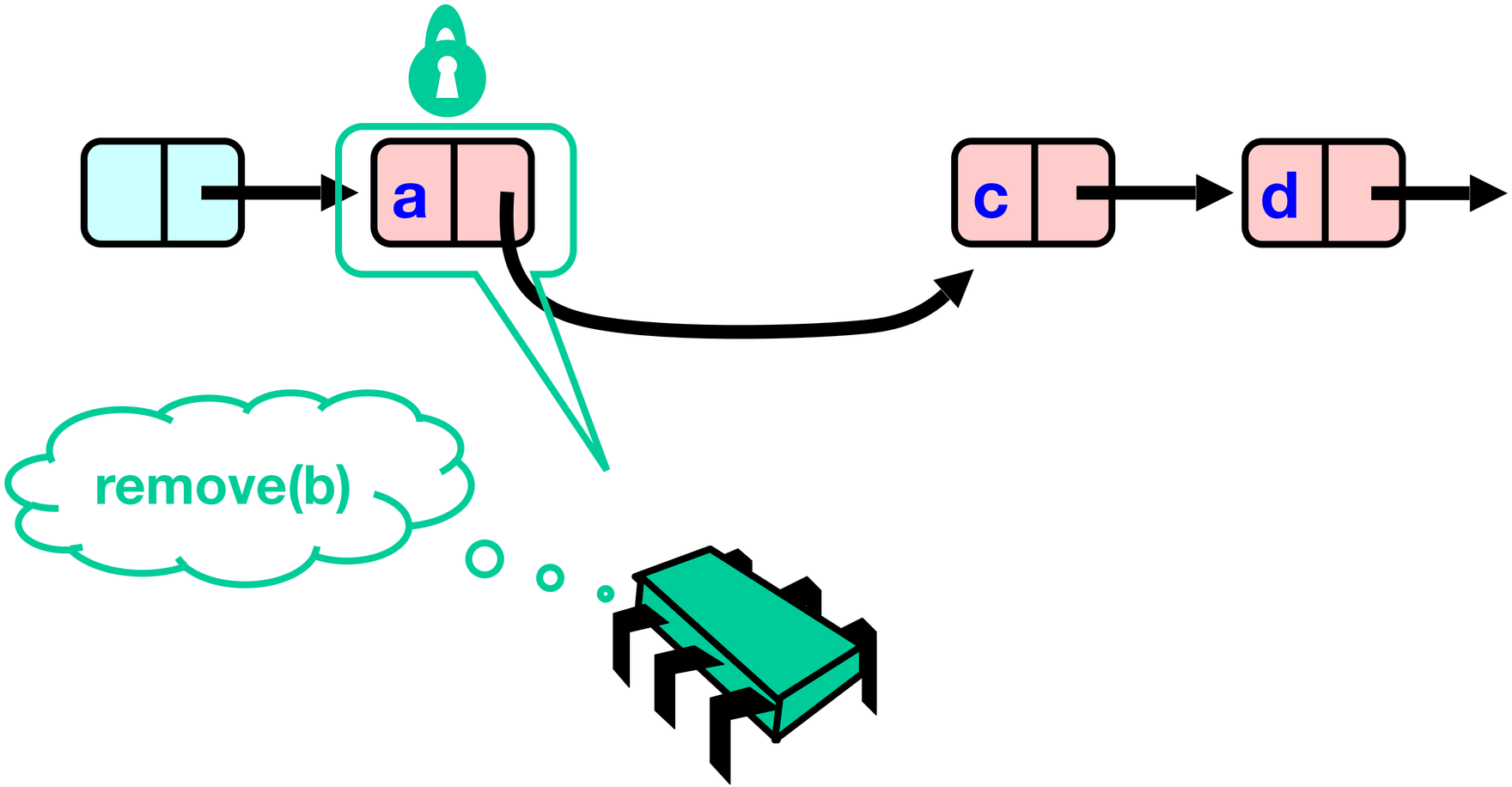
Removing a node



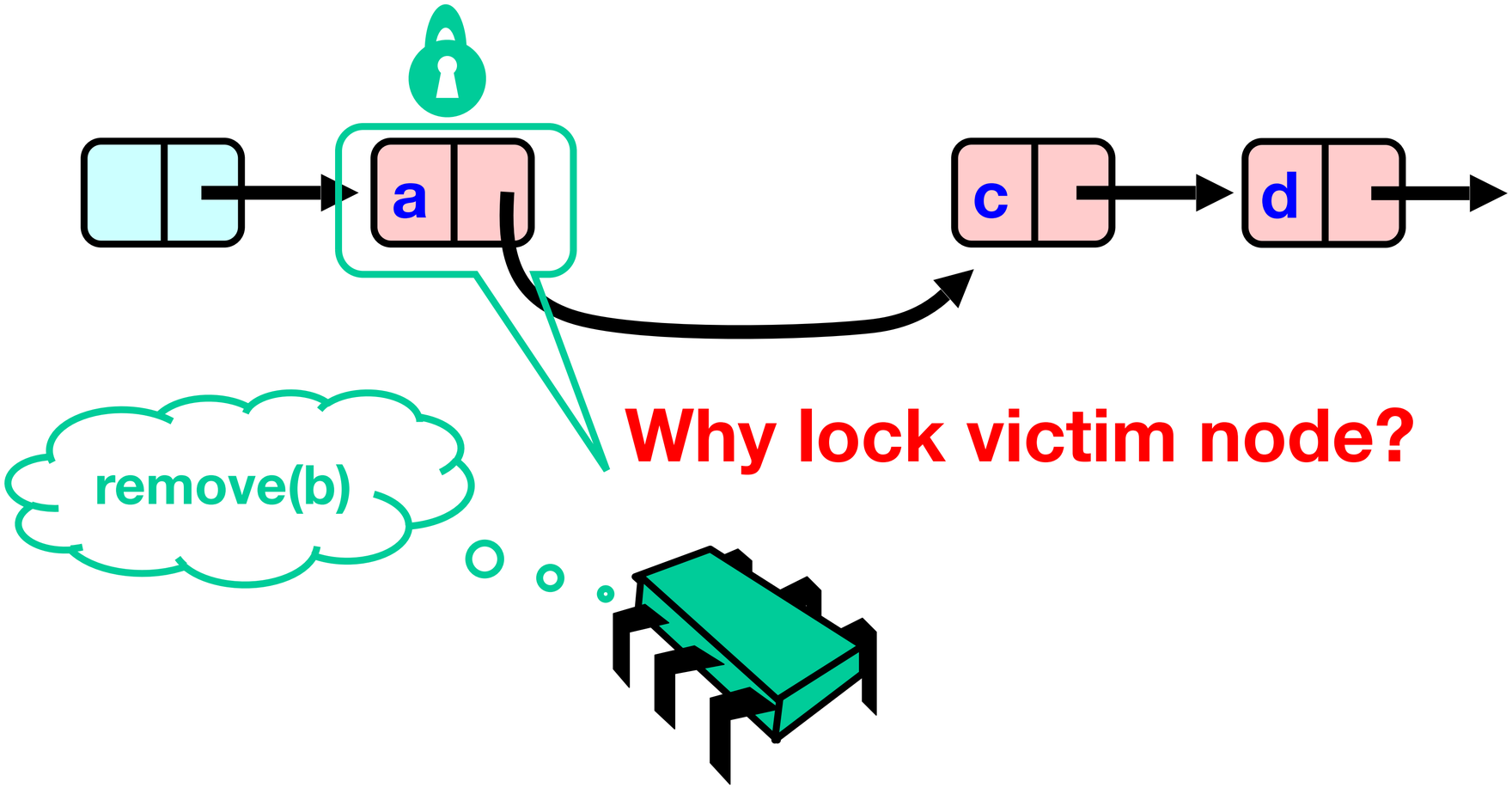
Removing a node



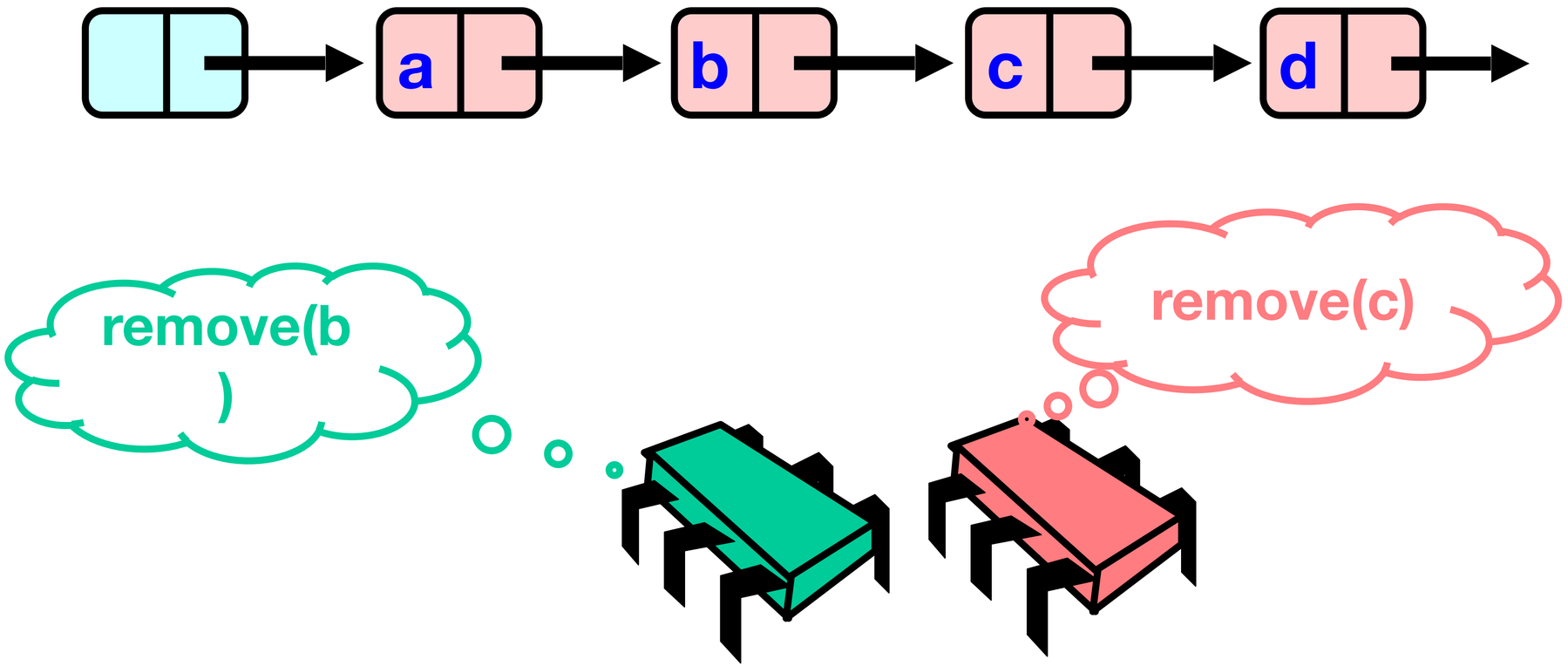
Removing a node



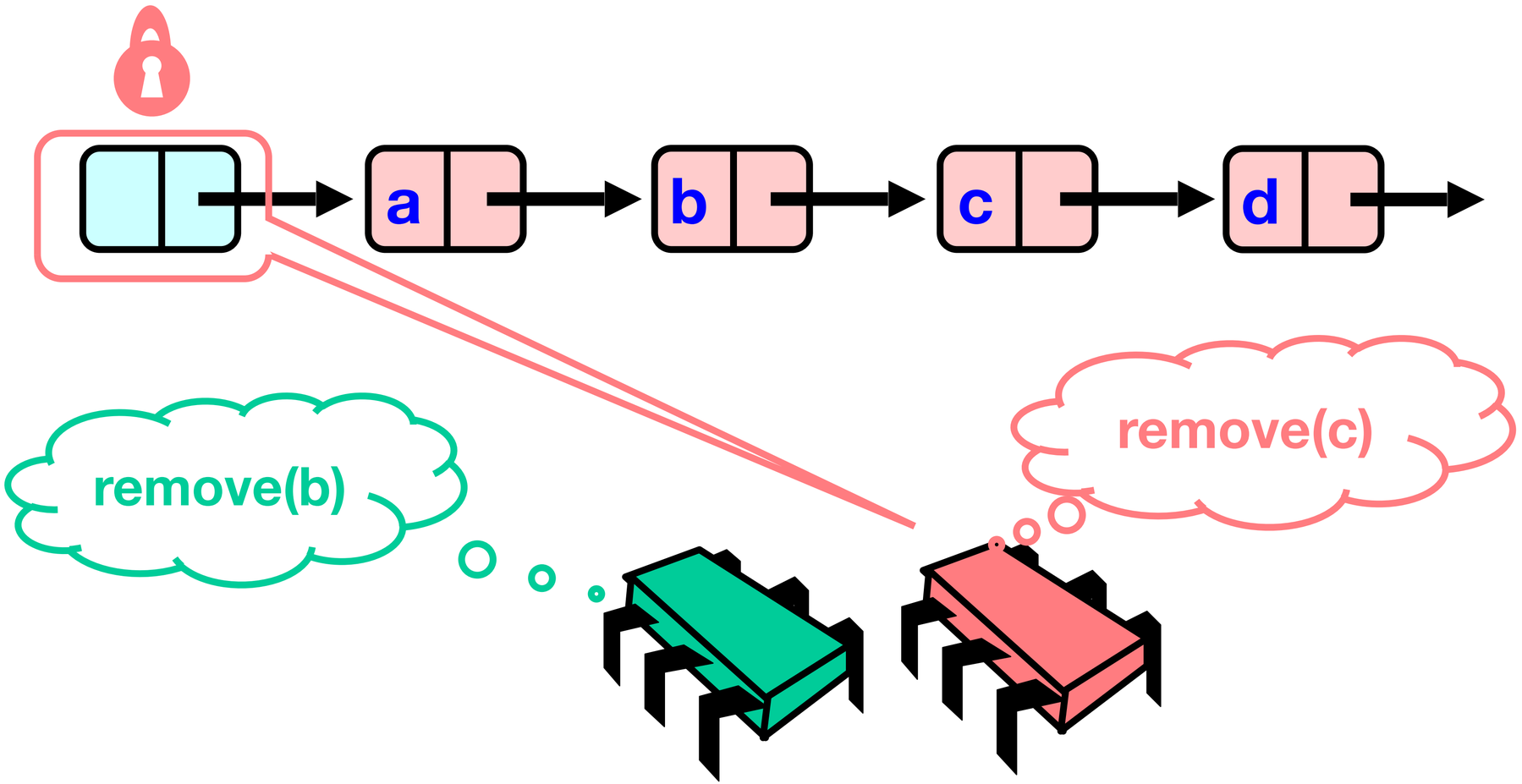
Removing a node



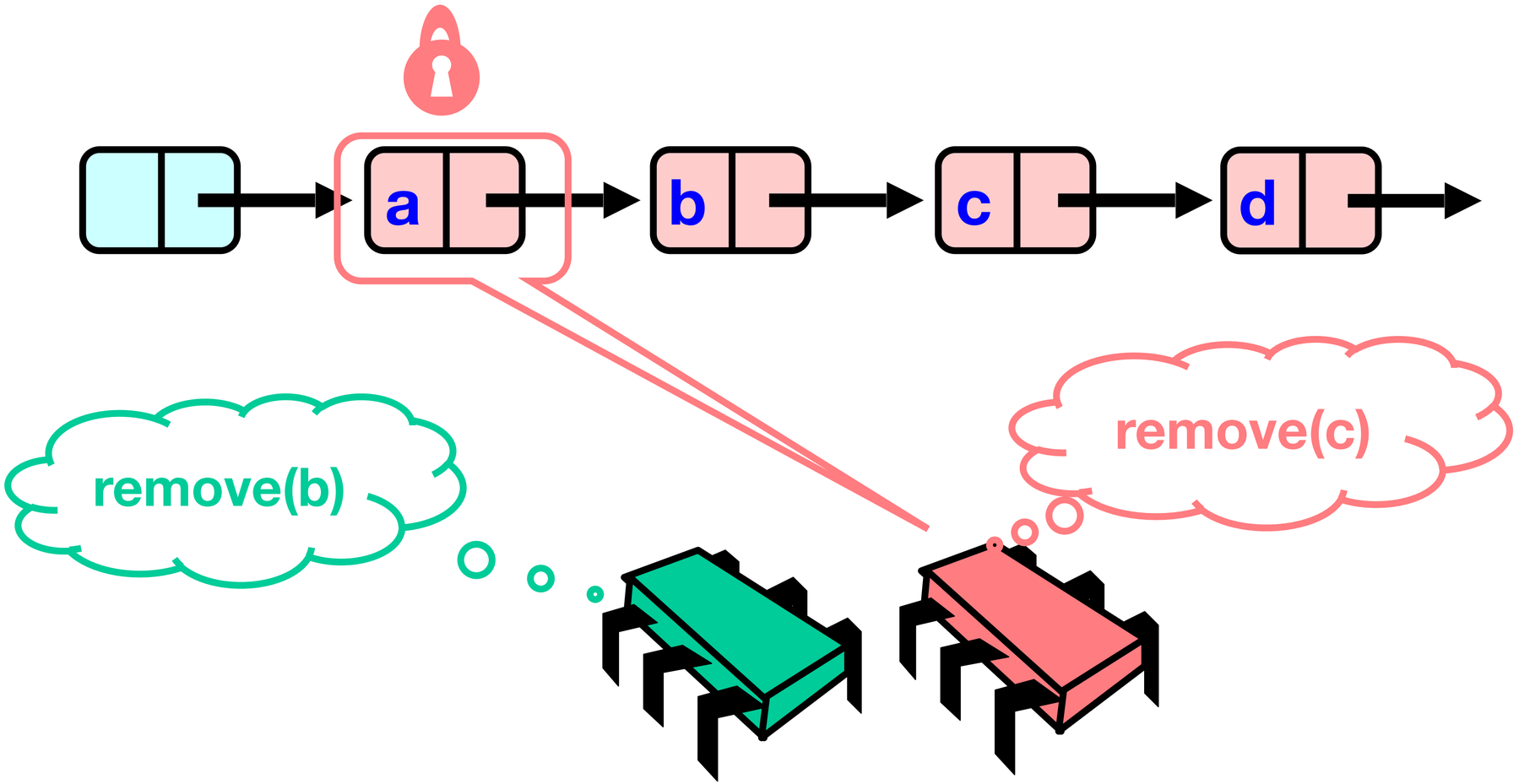
Concurrent Removes



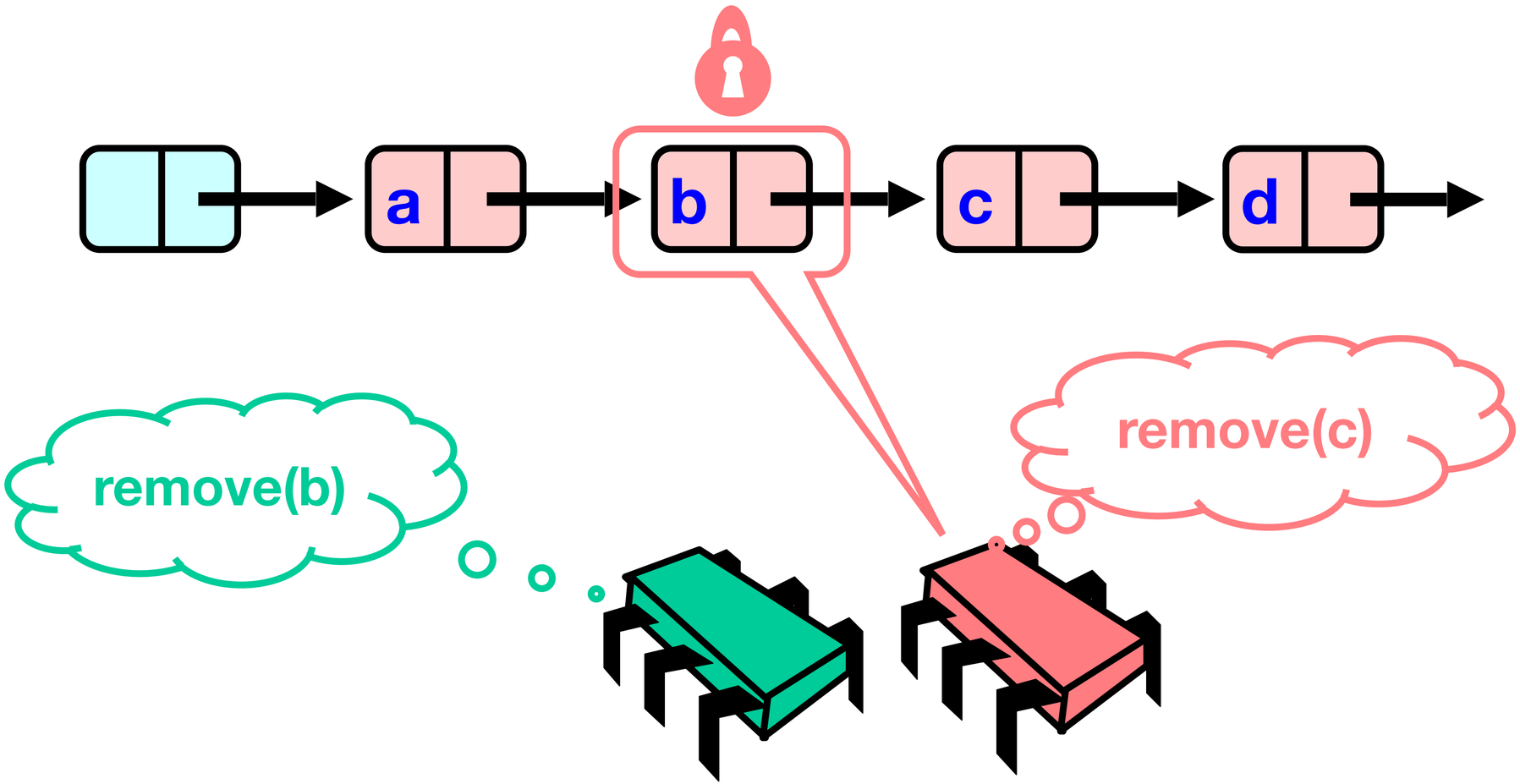
Concurrent Removes



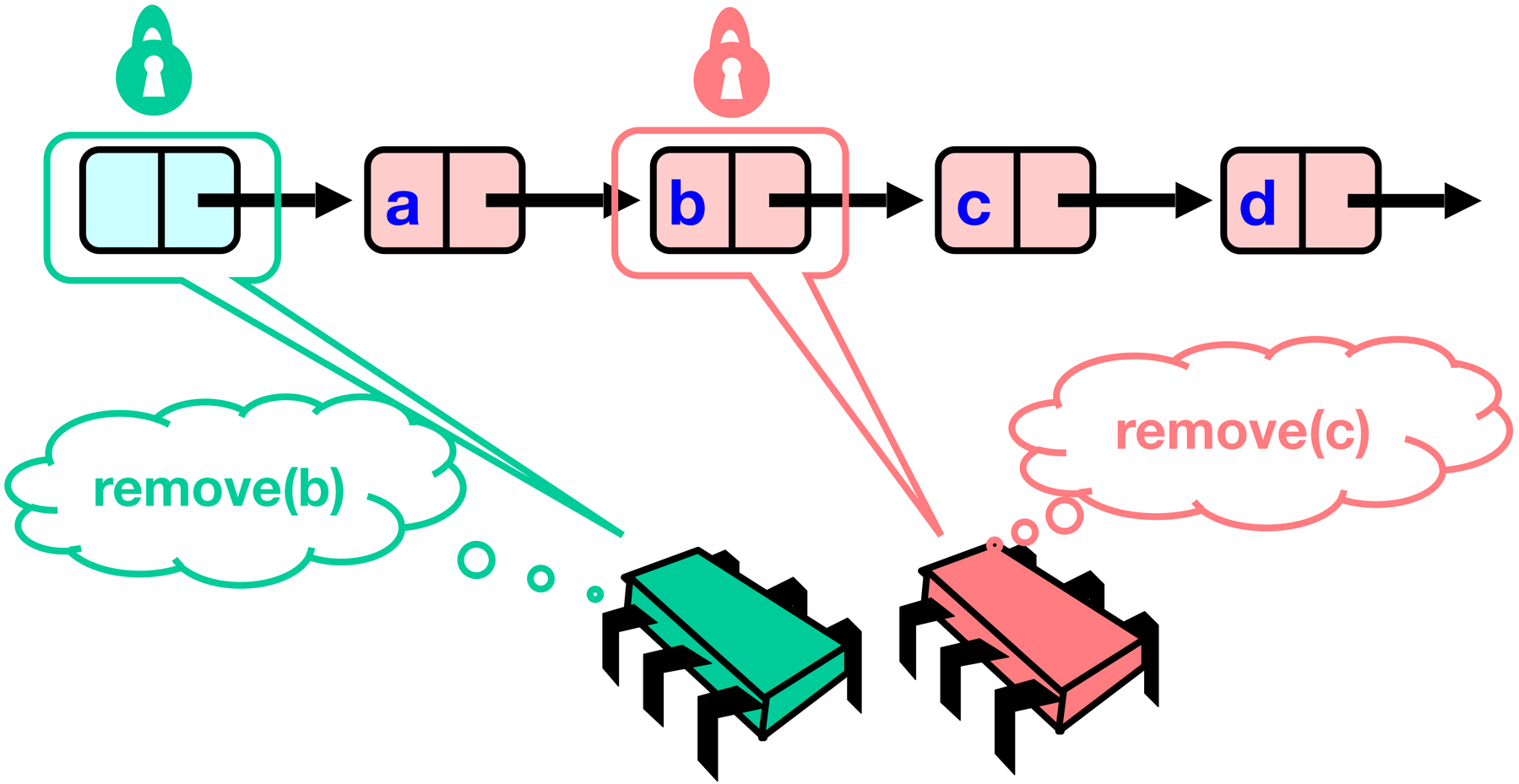
Concurrent Removes



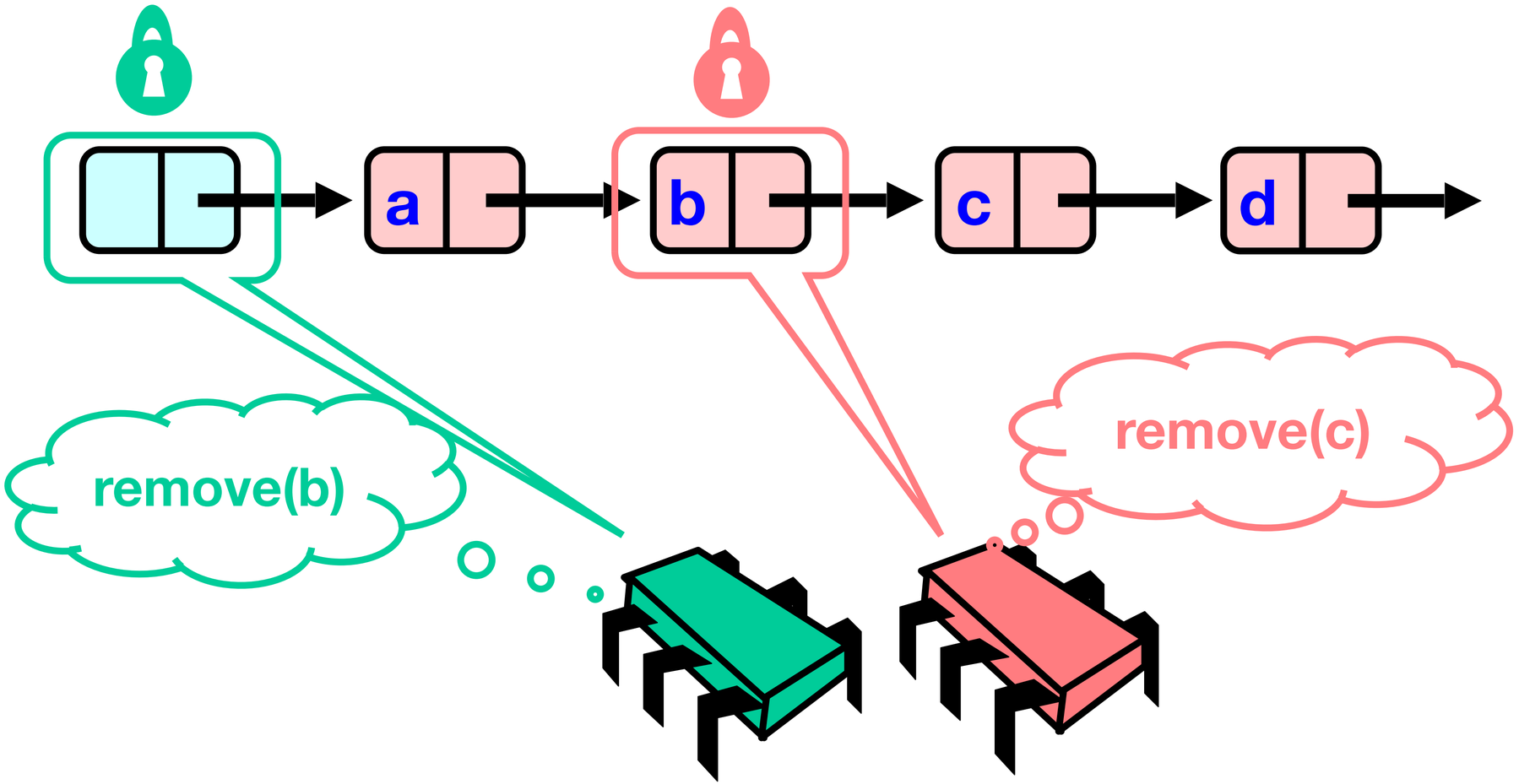
Concurrent Removes



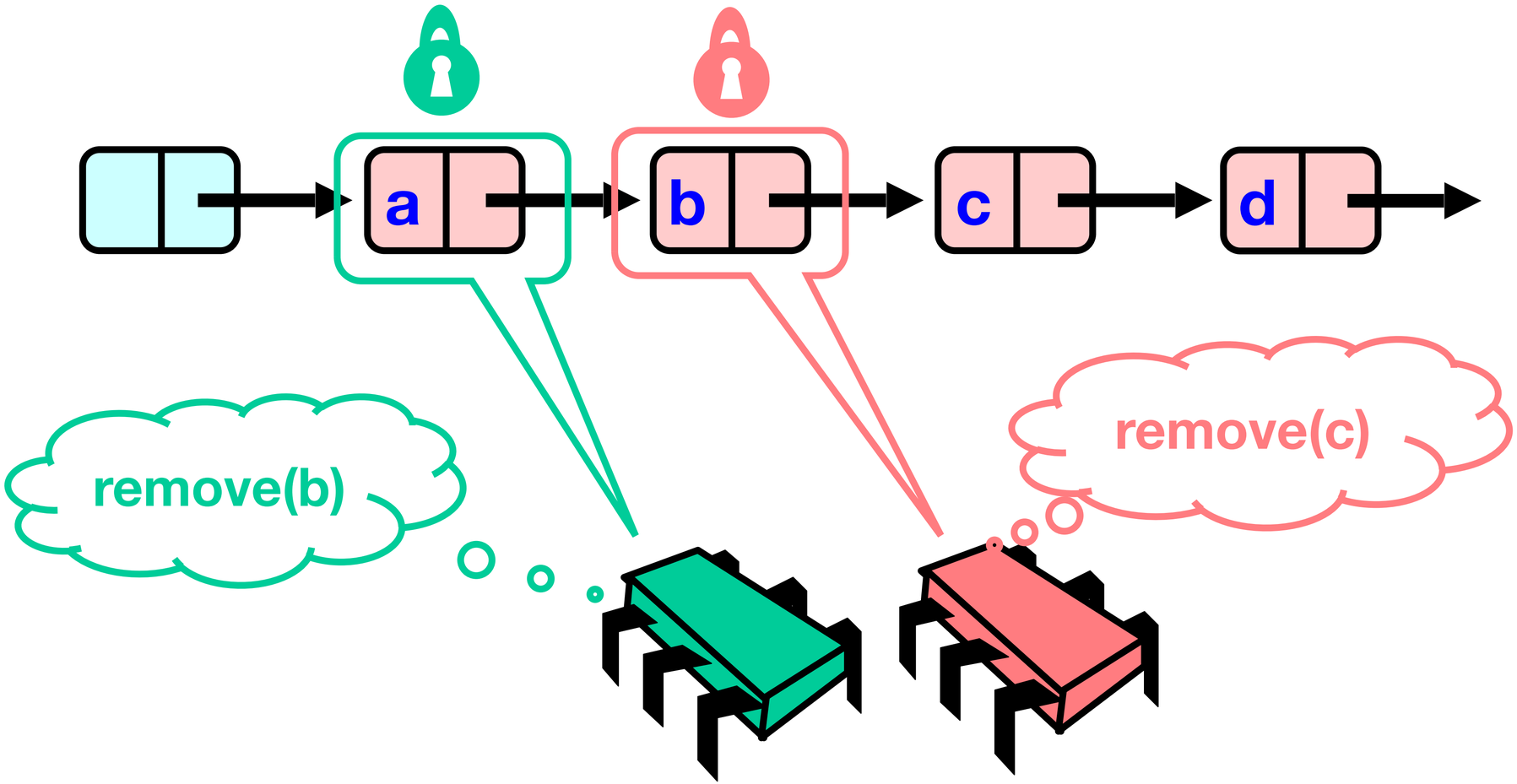
Concurrent Removes



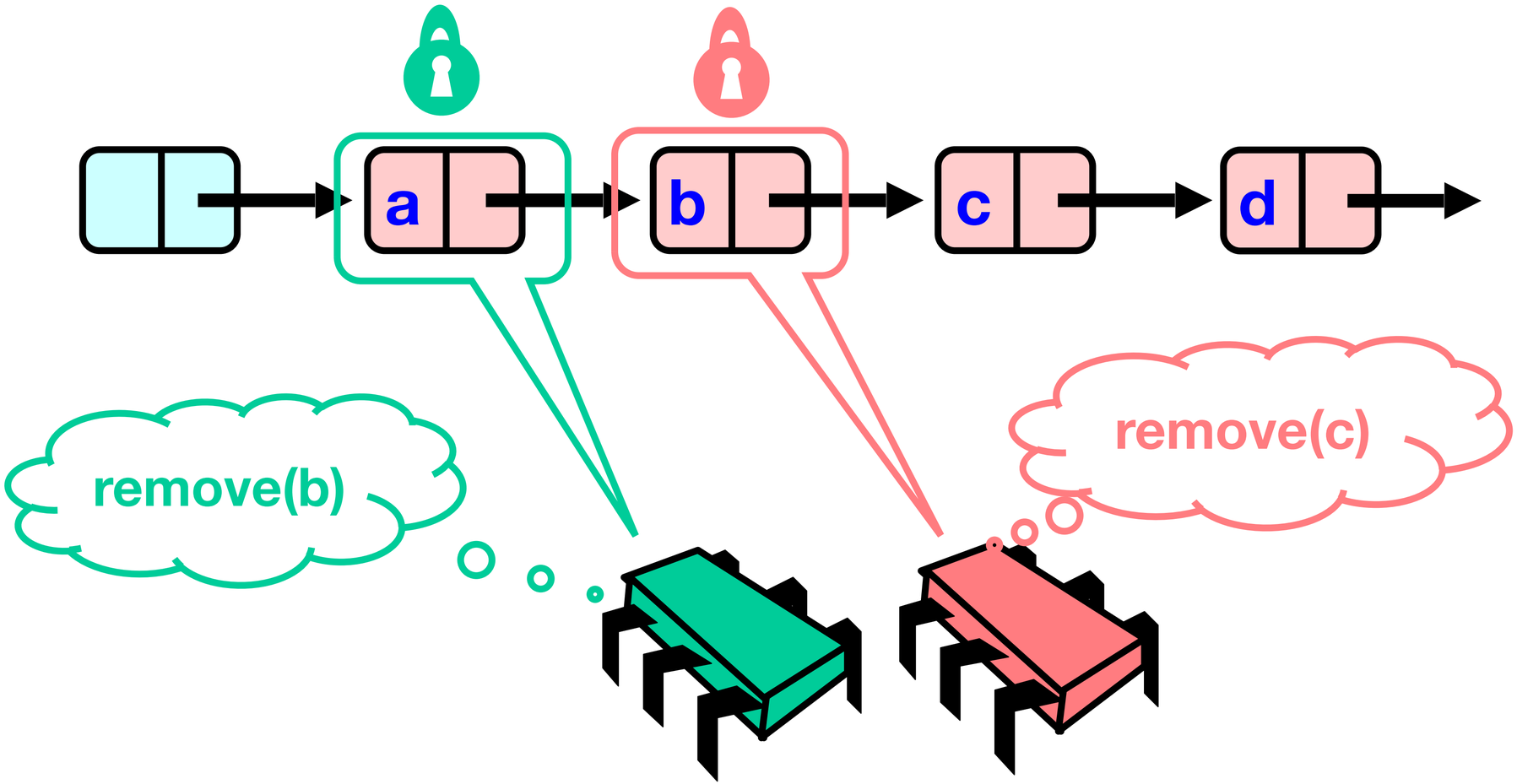
Concurrent Removes



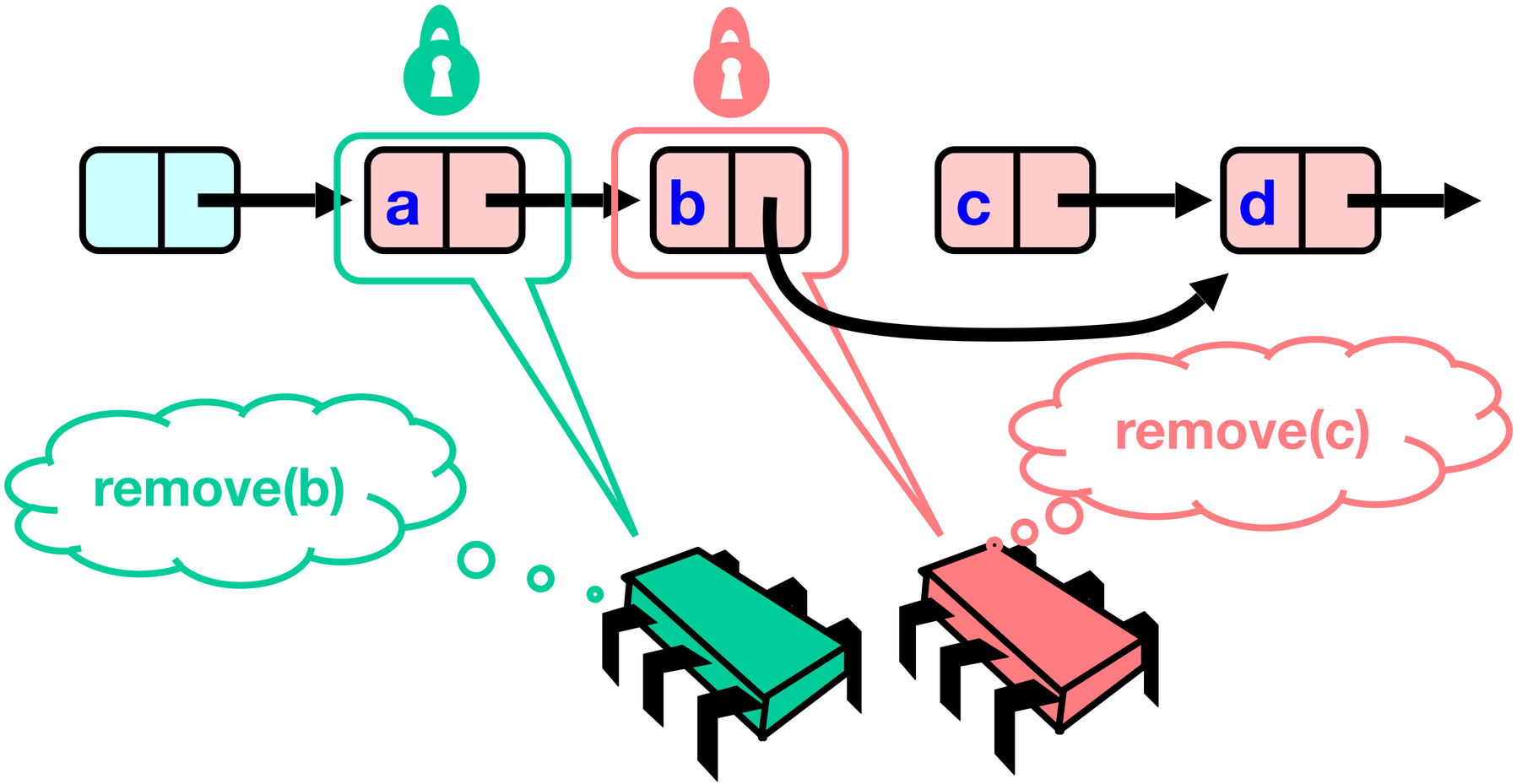
Concurrent Removes



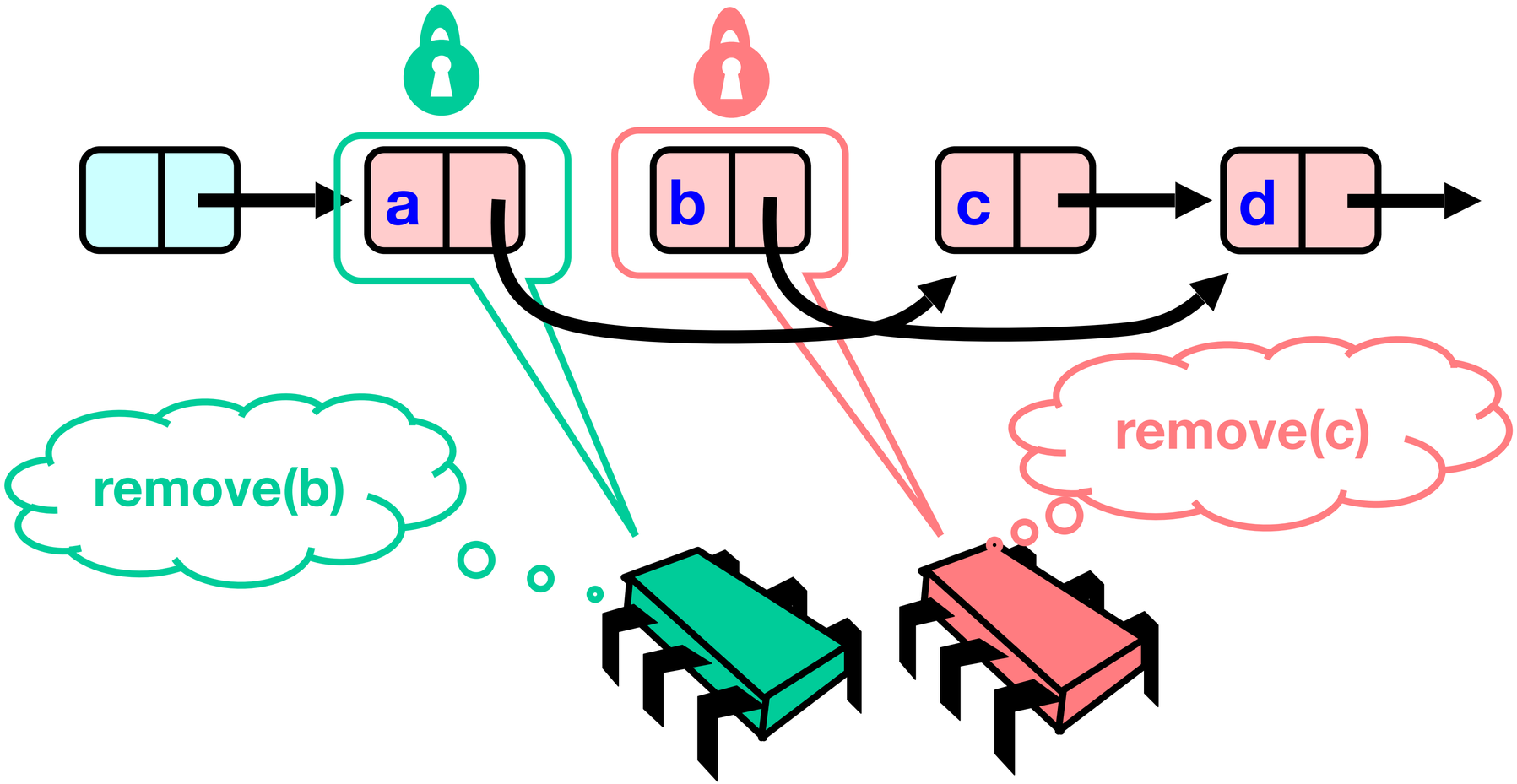
Concurrent Removes



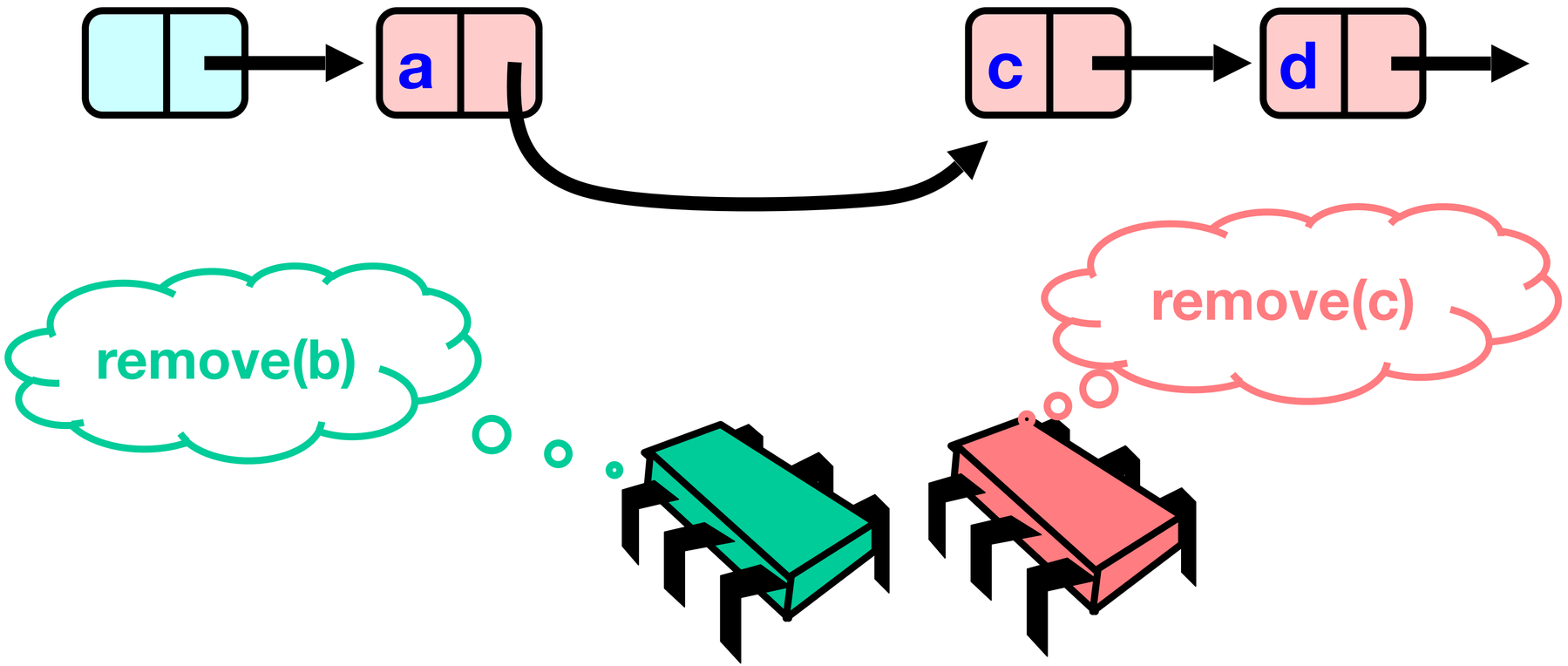
Concurrent Removes



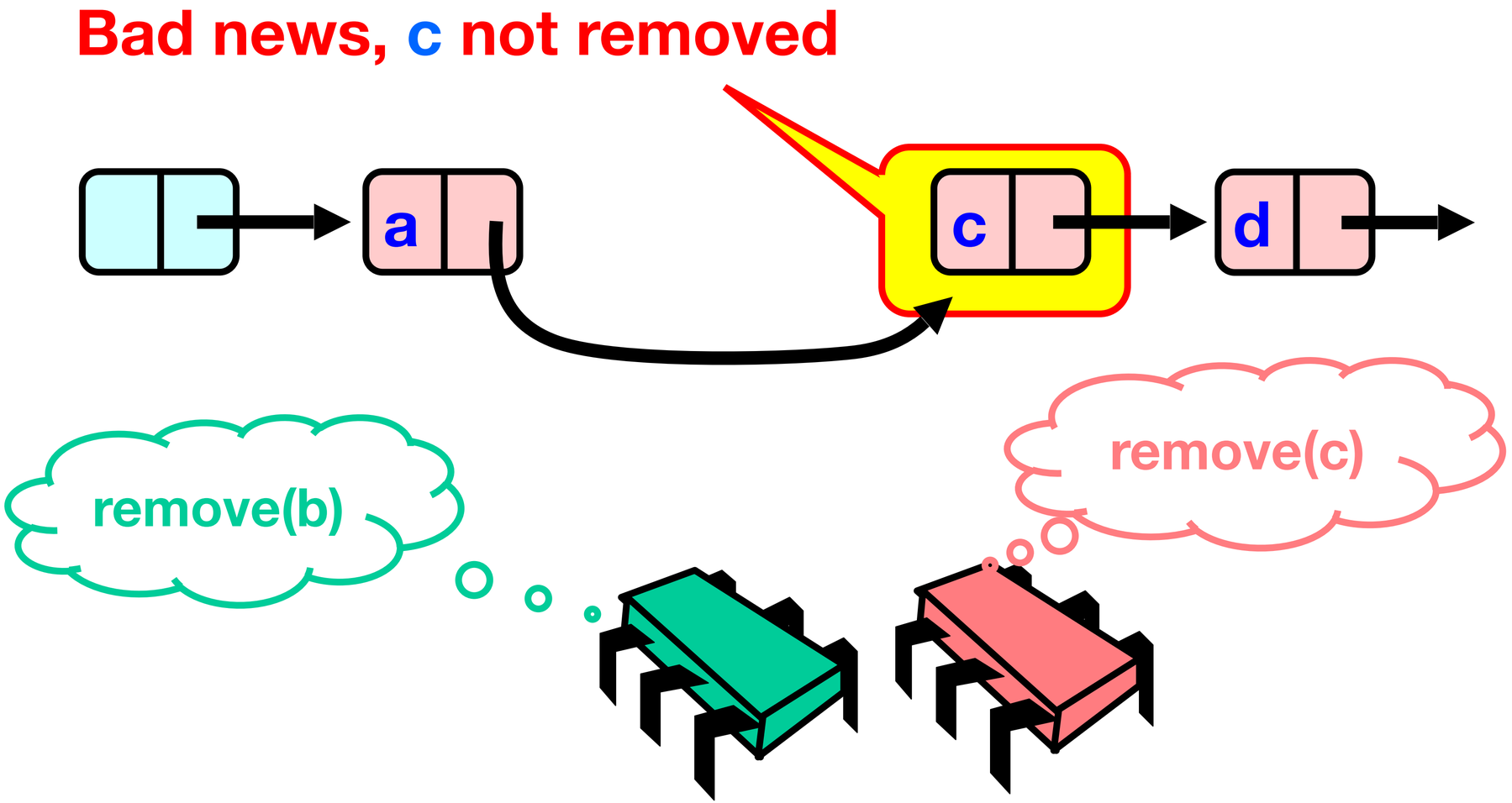
Concurrent Removes



Uh, Oh

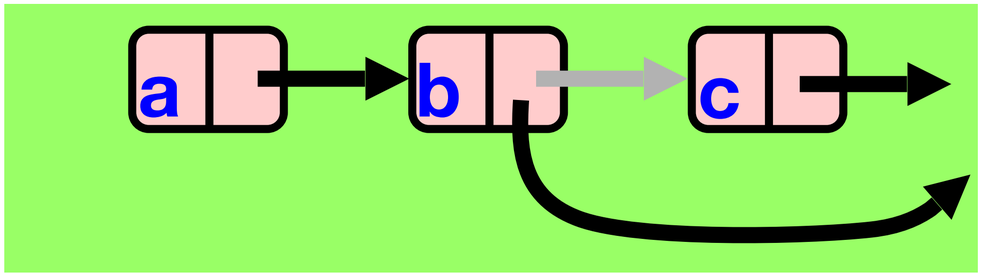


Uh, Oh

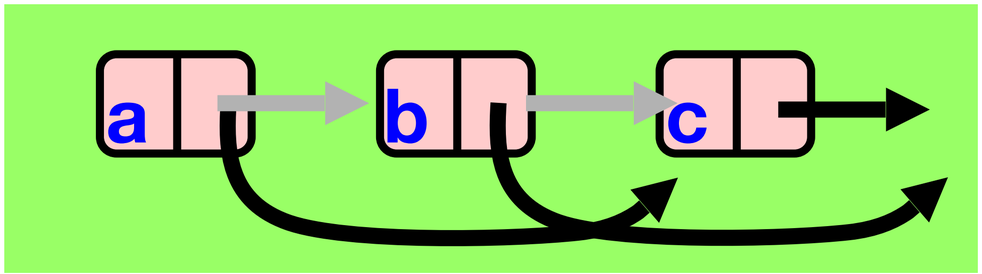


Problem

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



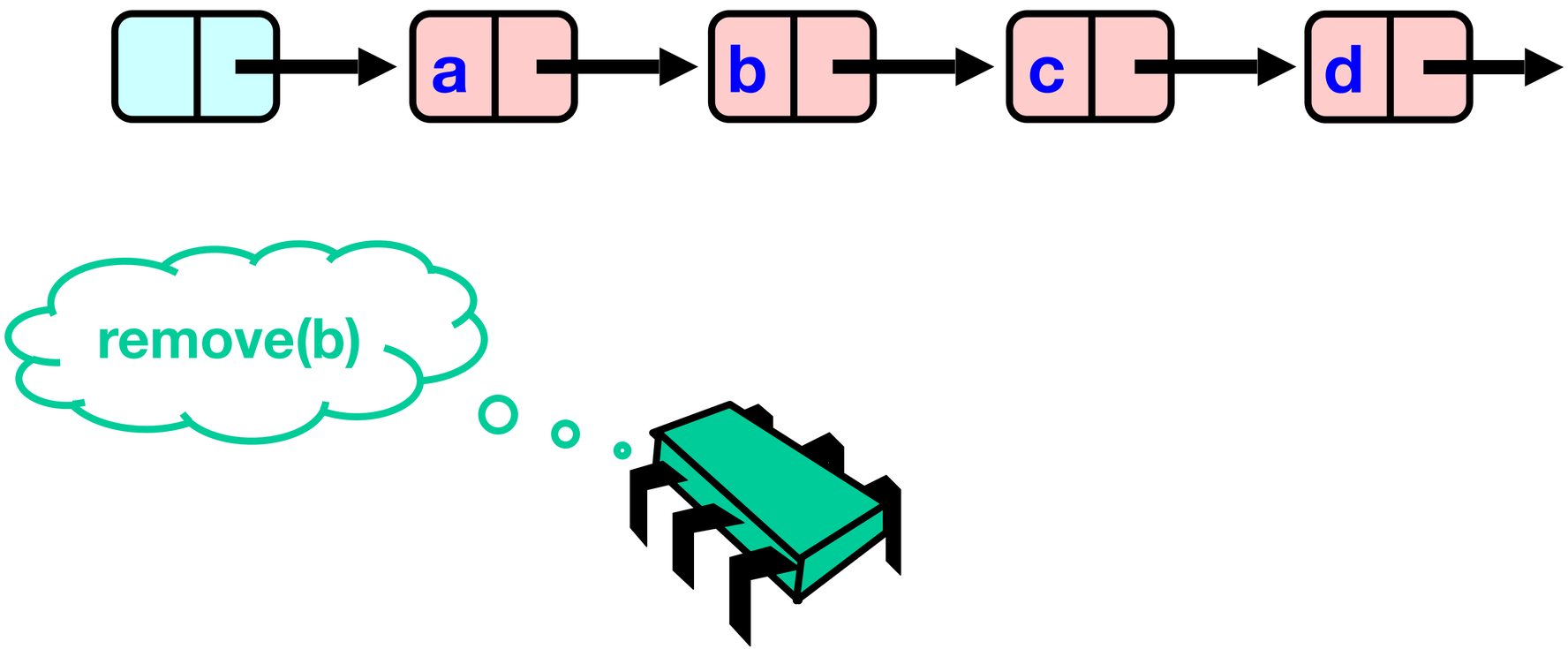
- Problem is,
  - Someone deleting **b** concurrently could direct **a** pointer to **c**



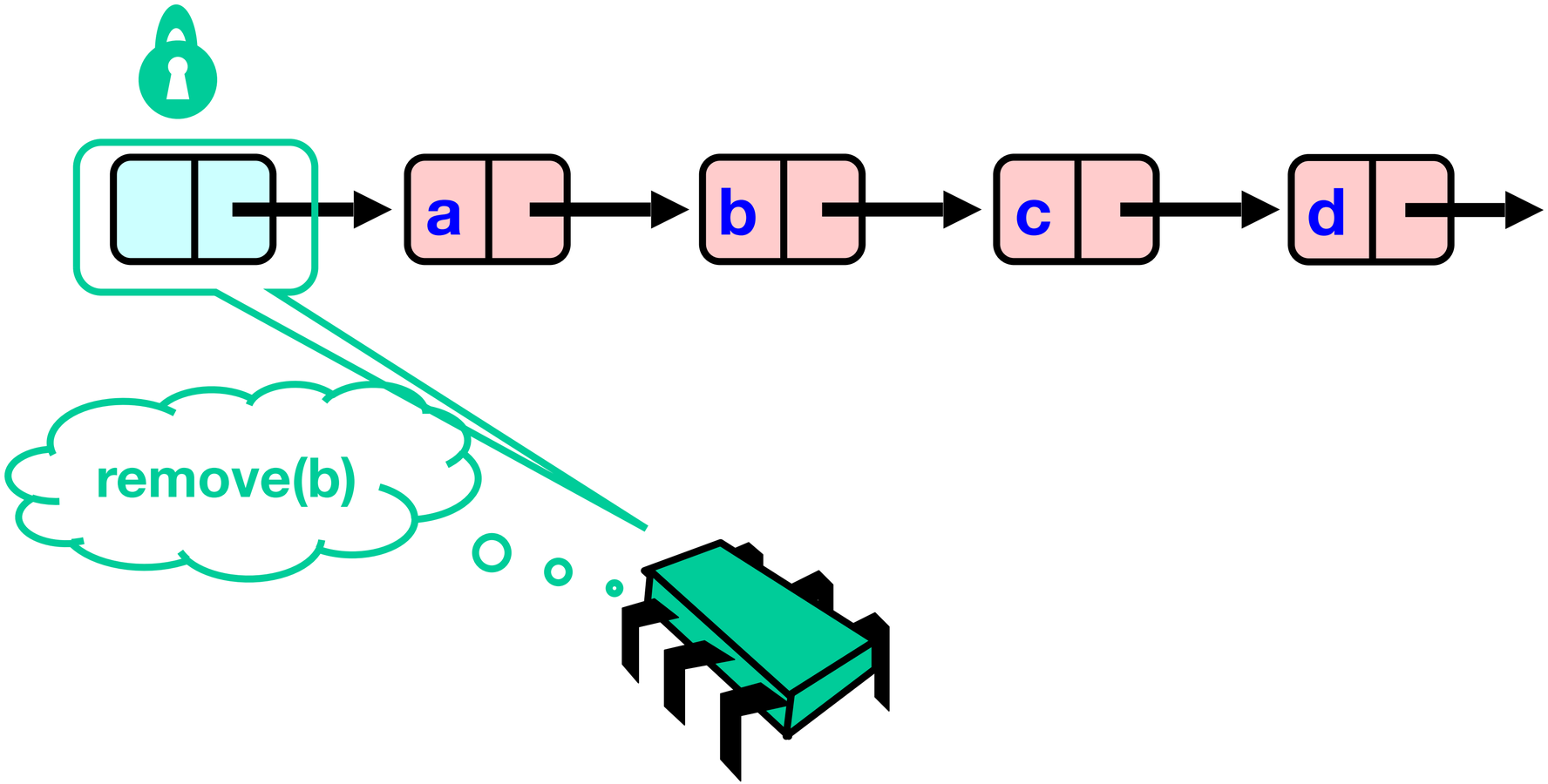
Insight

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

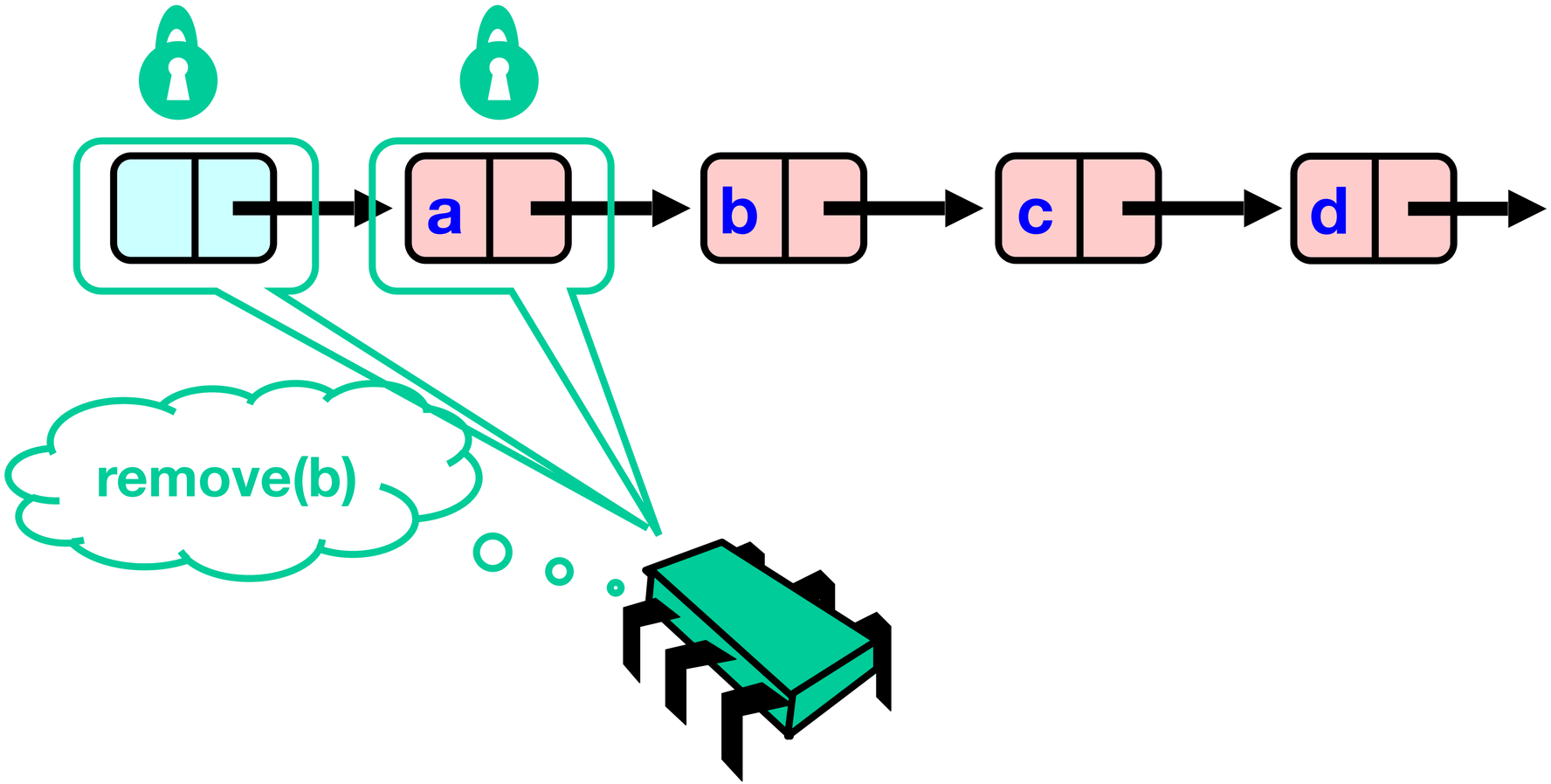
Hand-over-hand again



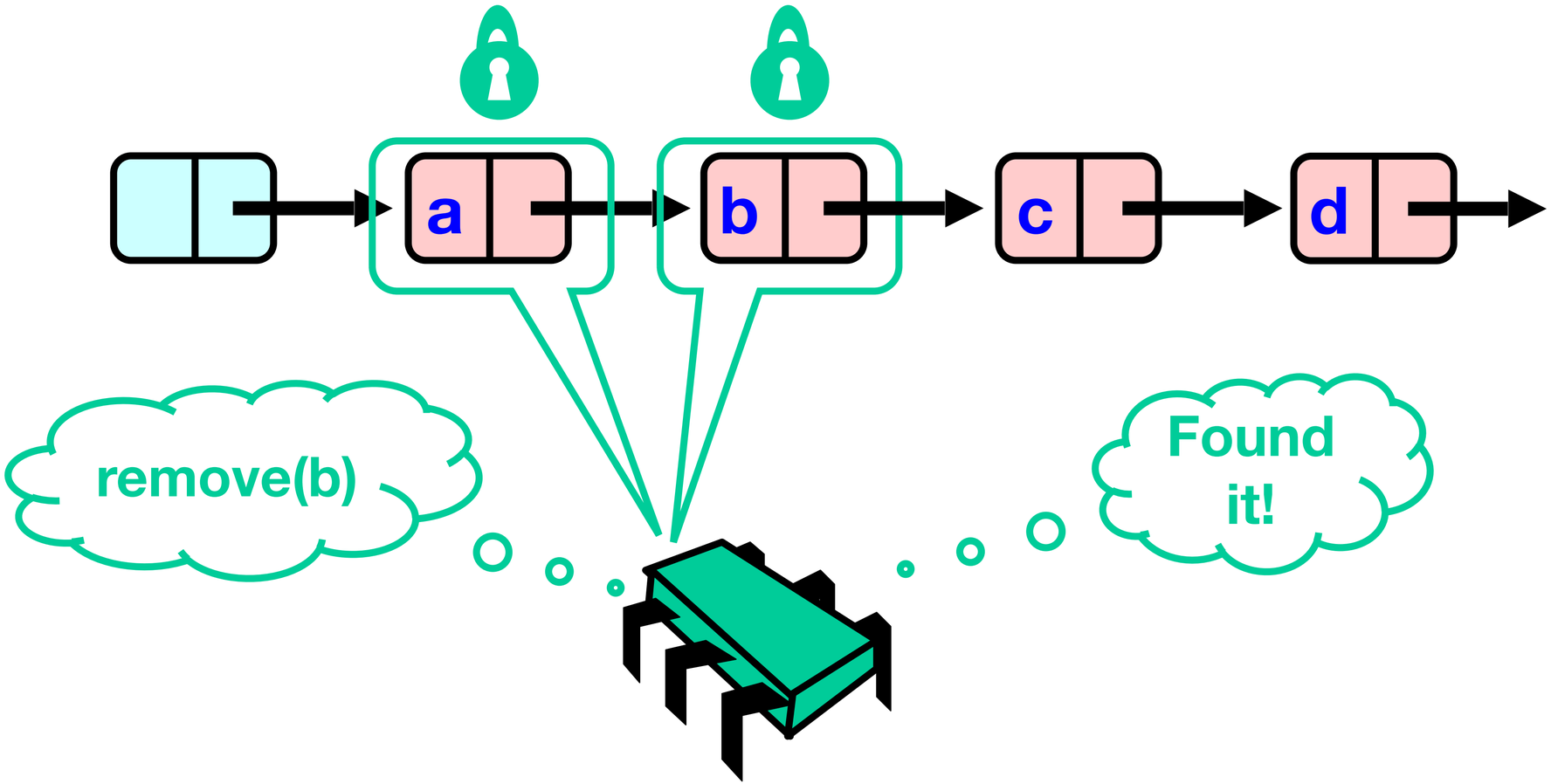
Hand-over-hand again



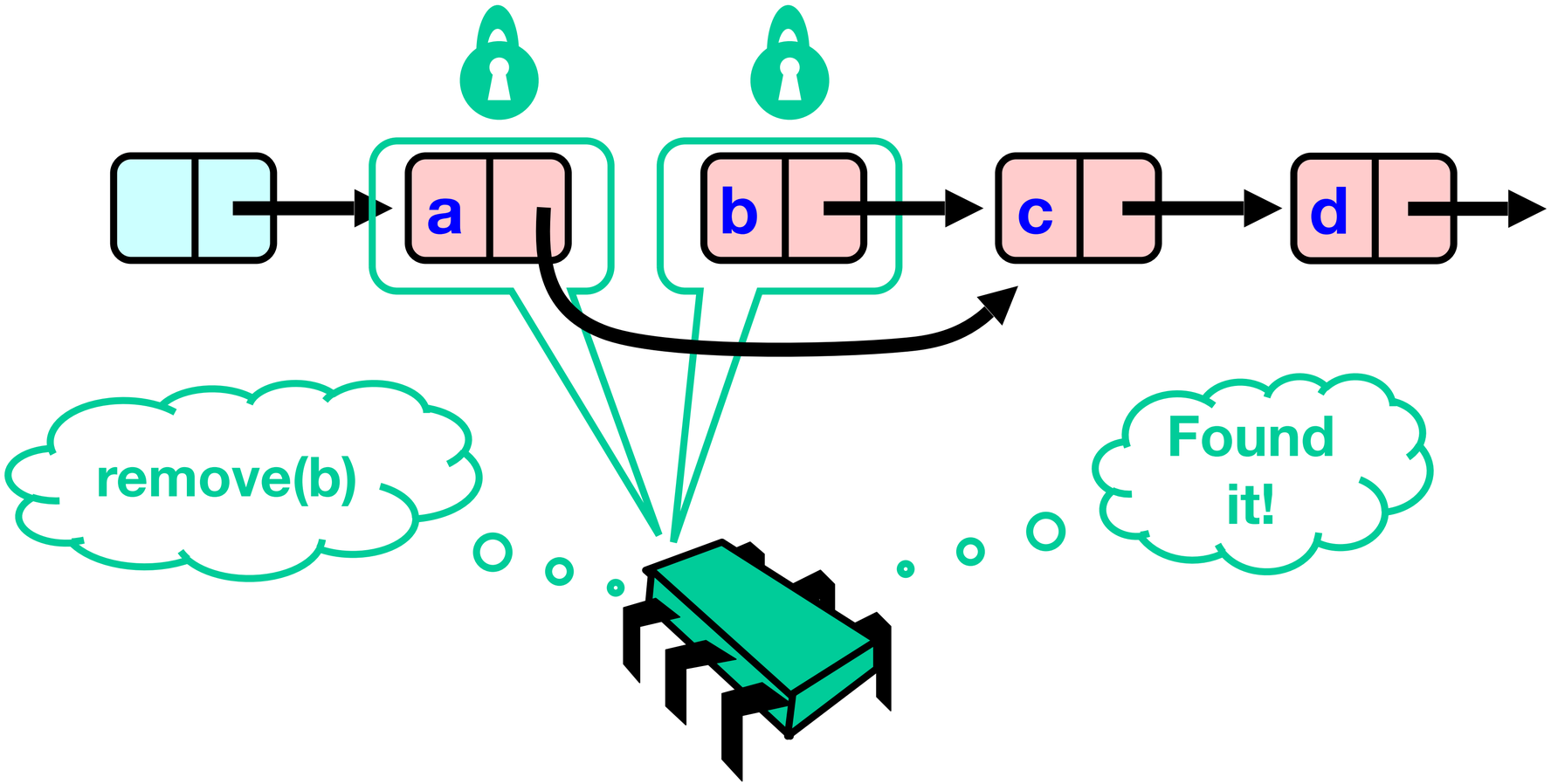
Hand-over-hand again



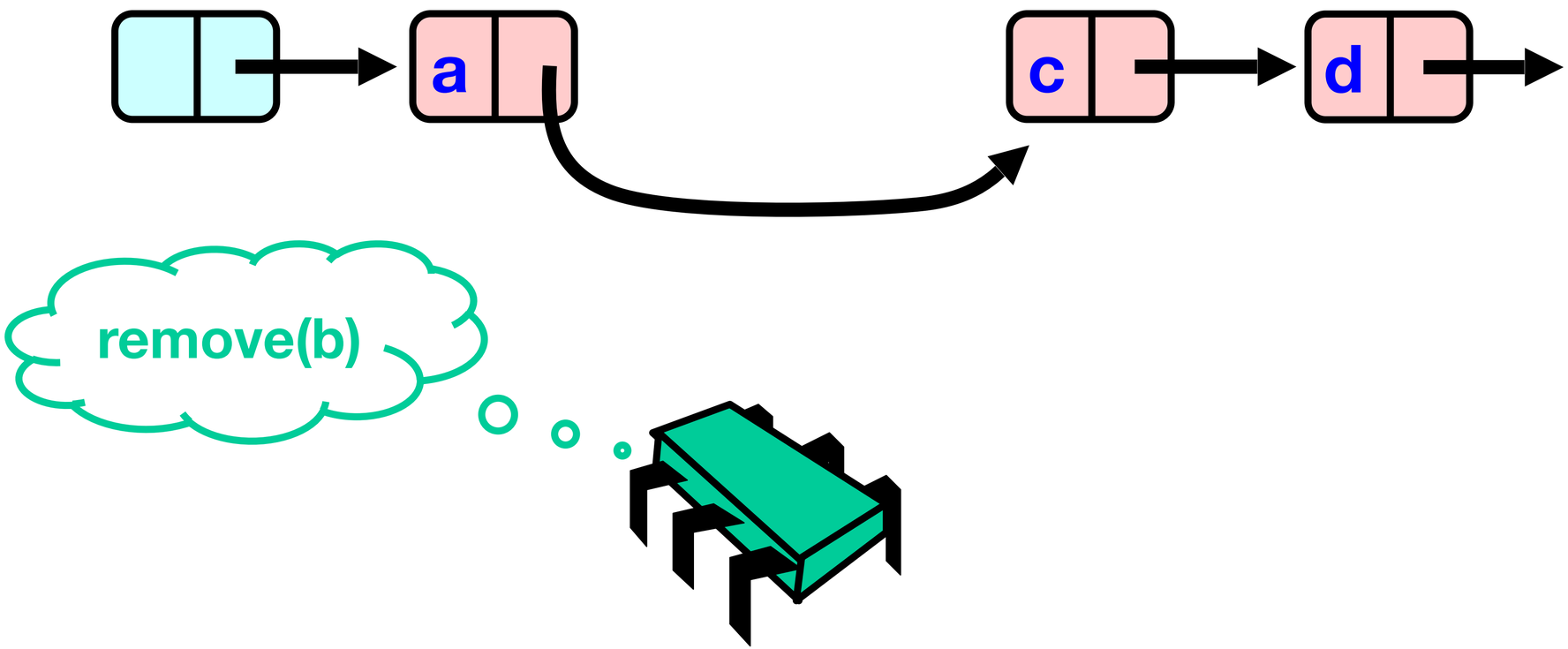
Hand-over-hand again



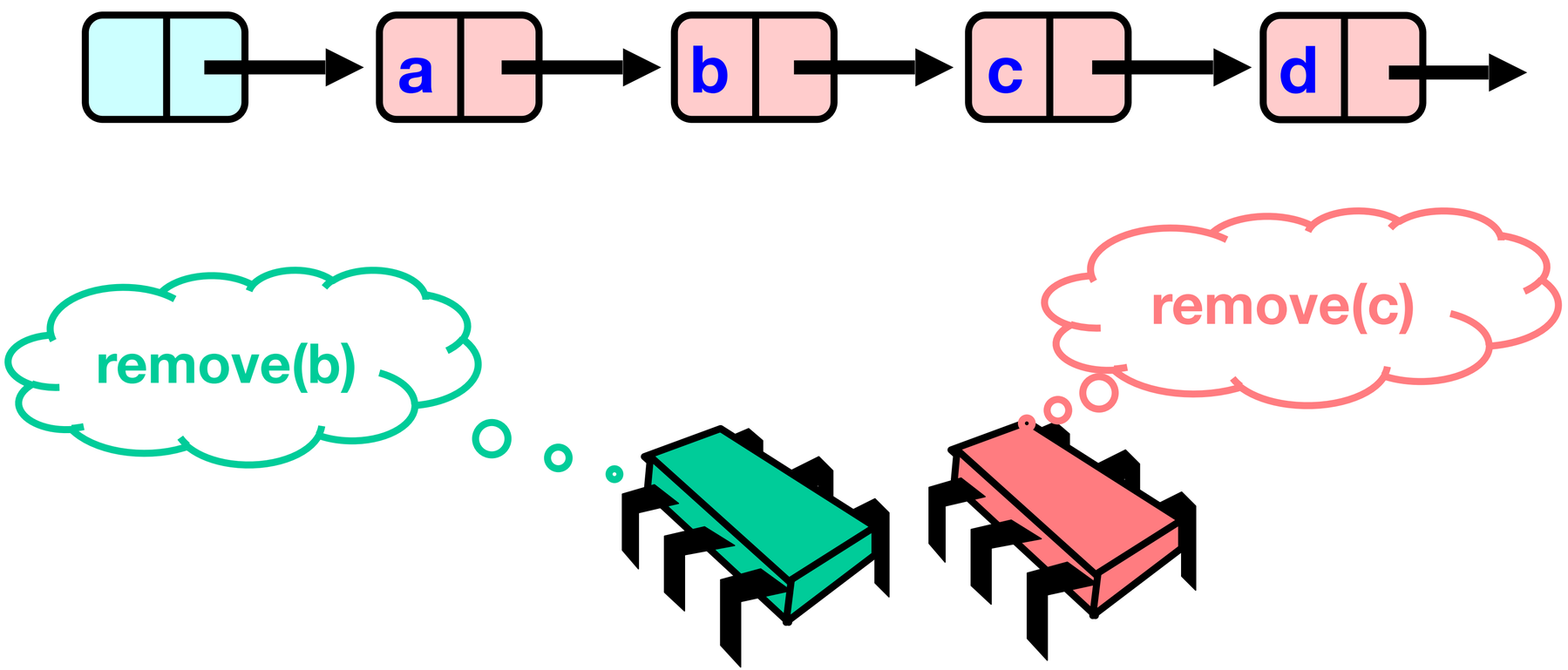
Hand-over-hand again



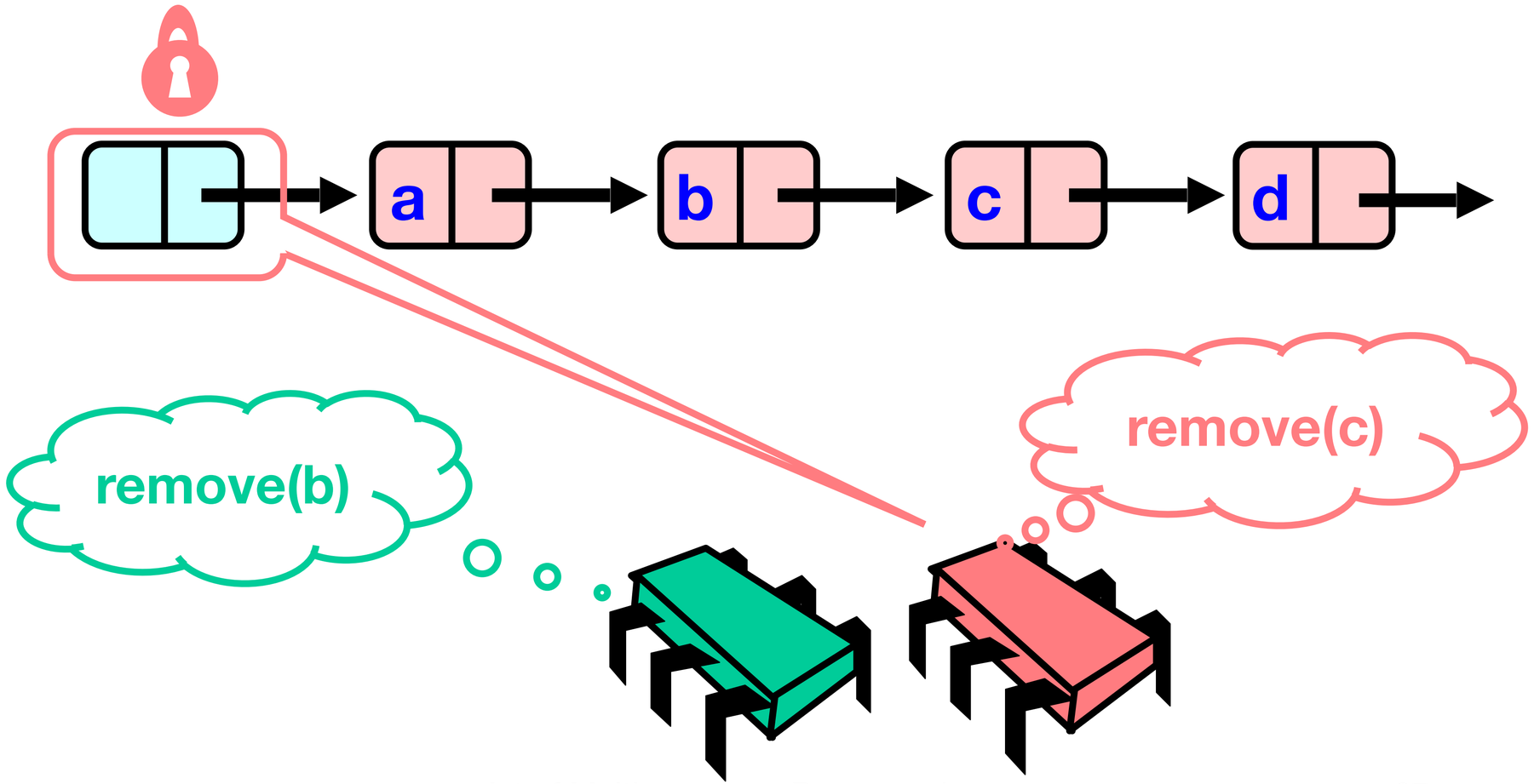
Hand-over-hand again



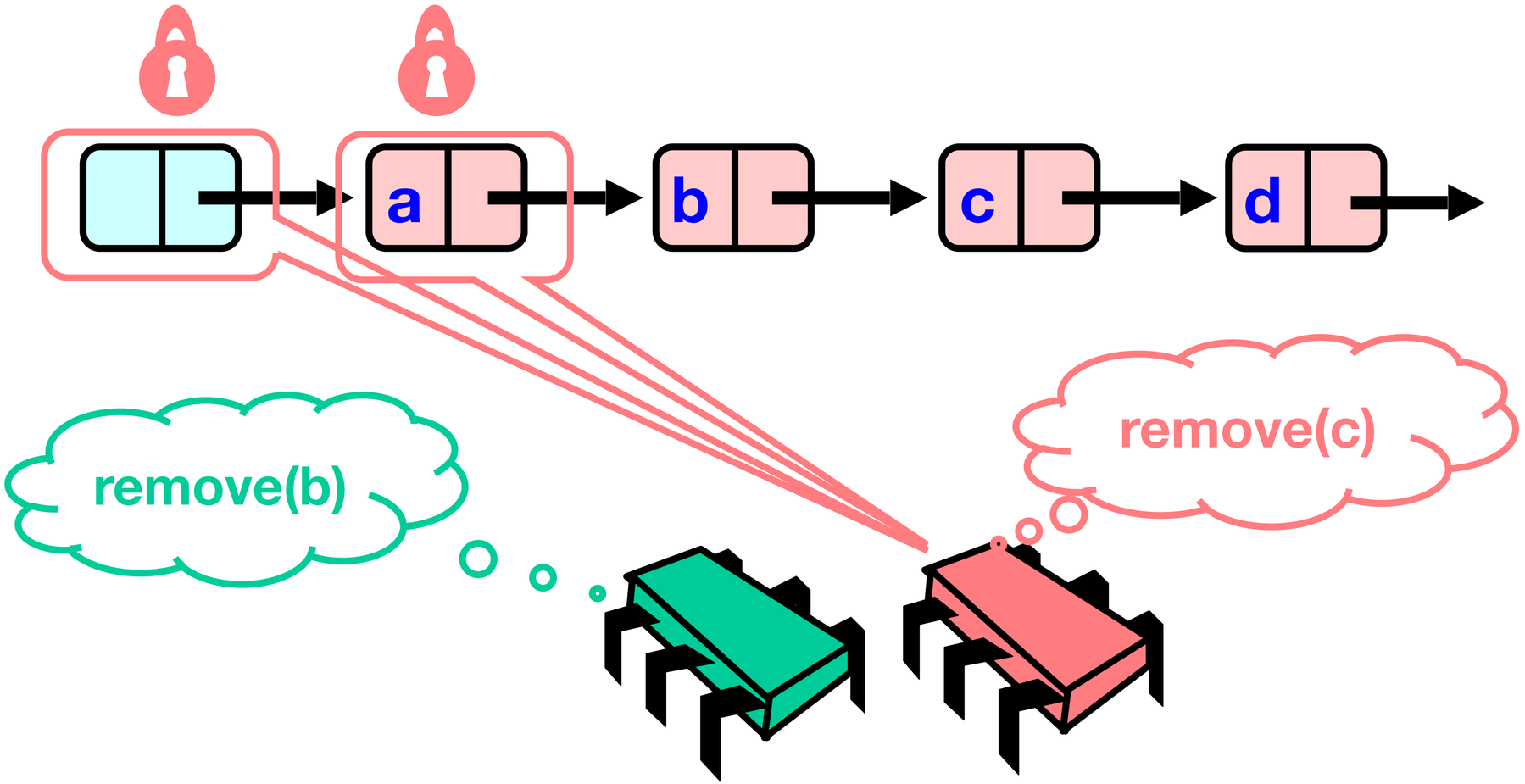
Removing a node



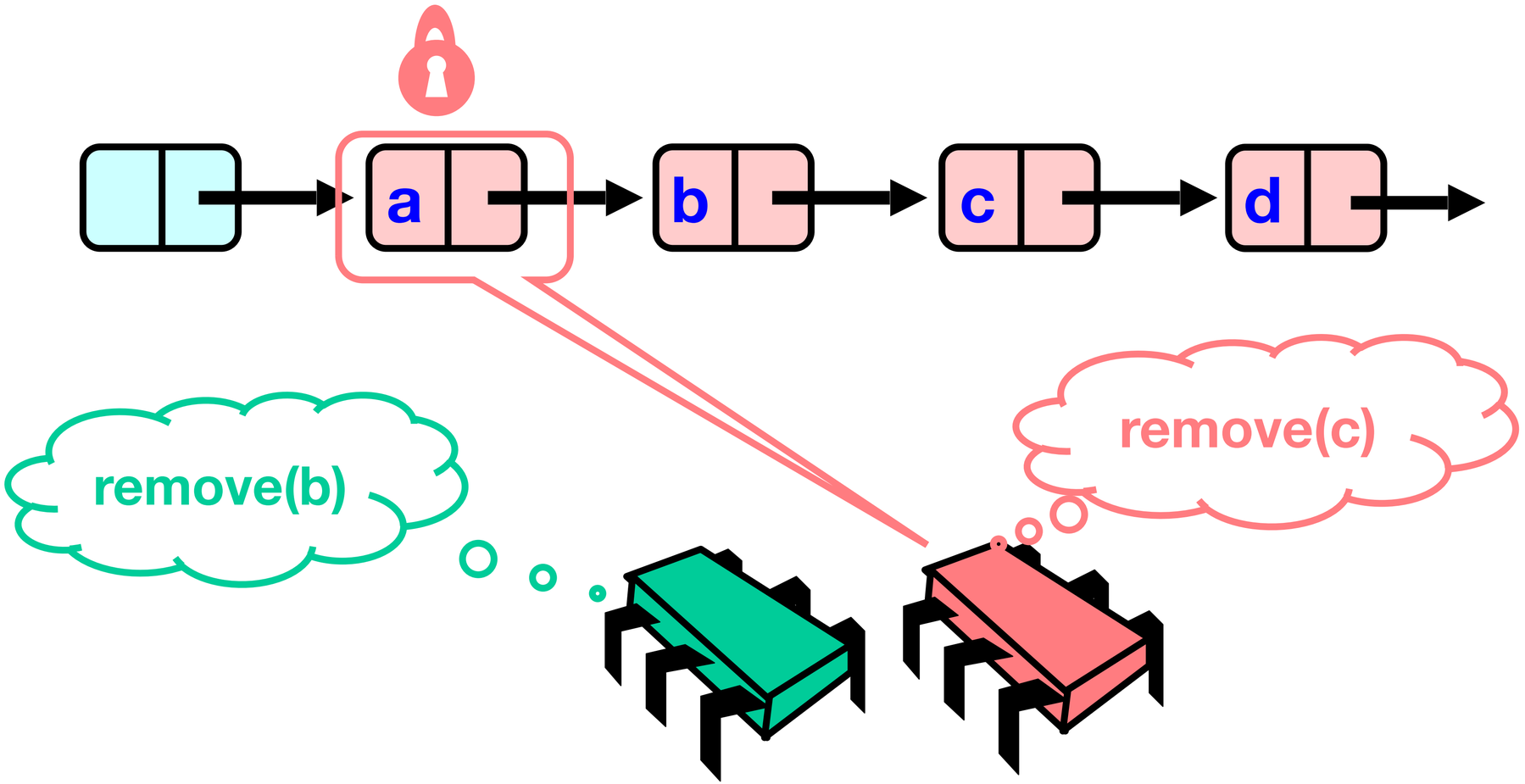
Removing a node



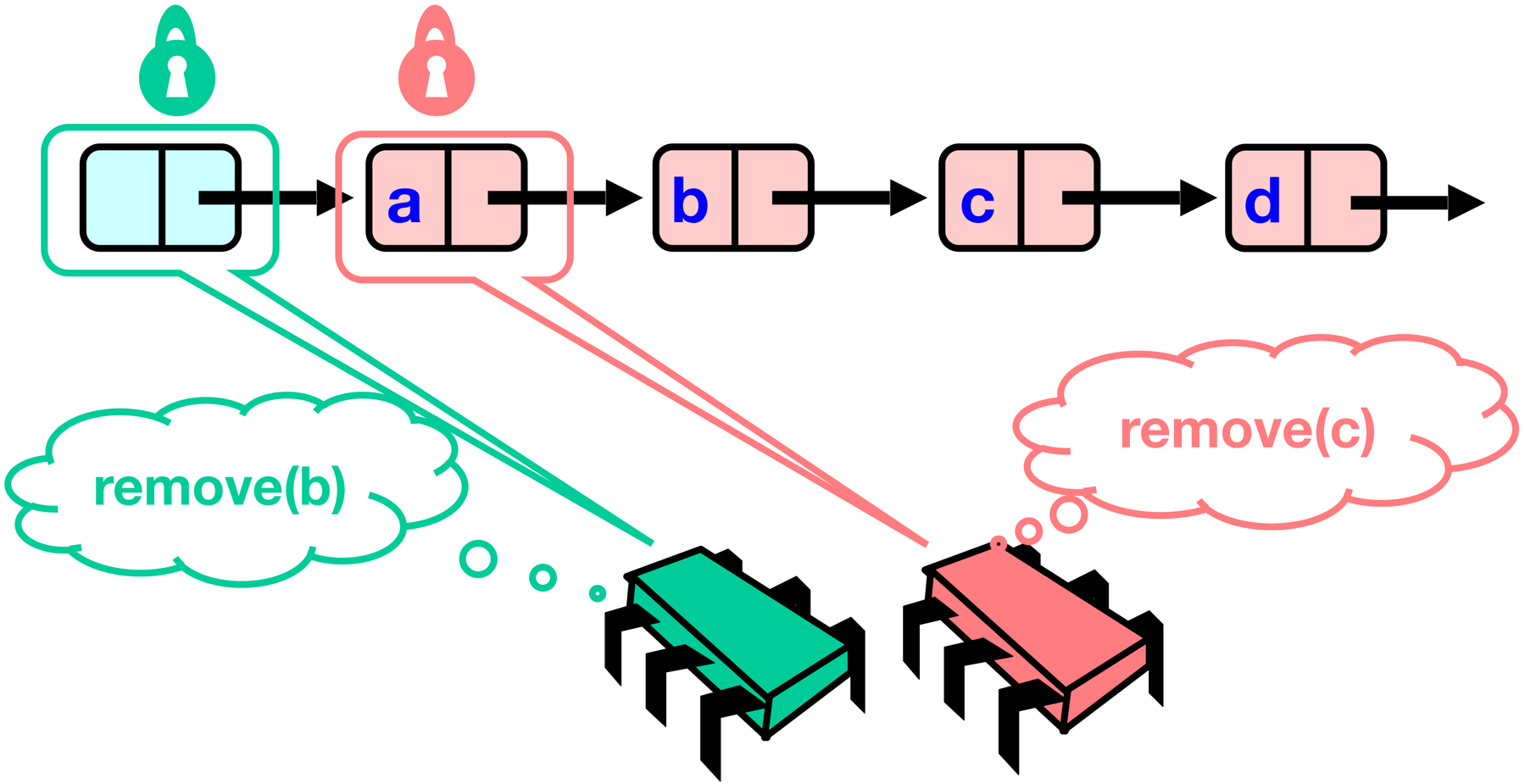
Removing a node



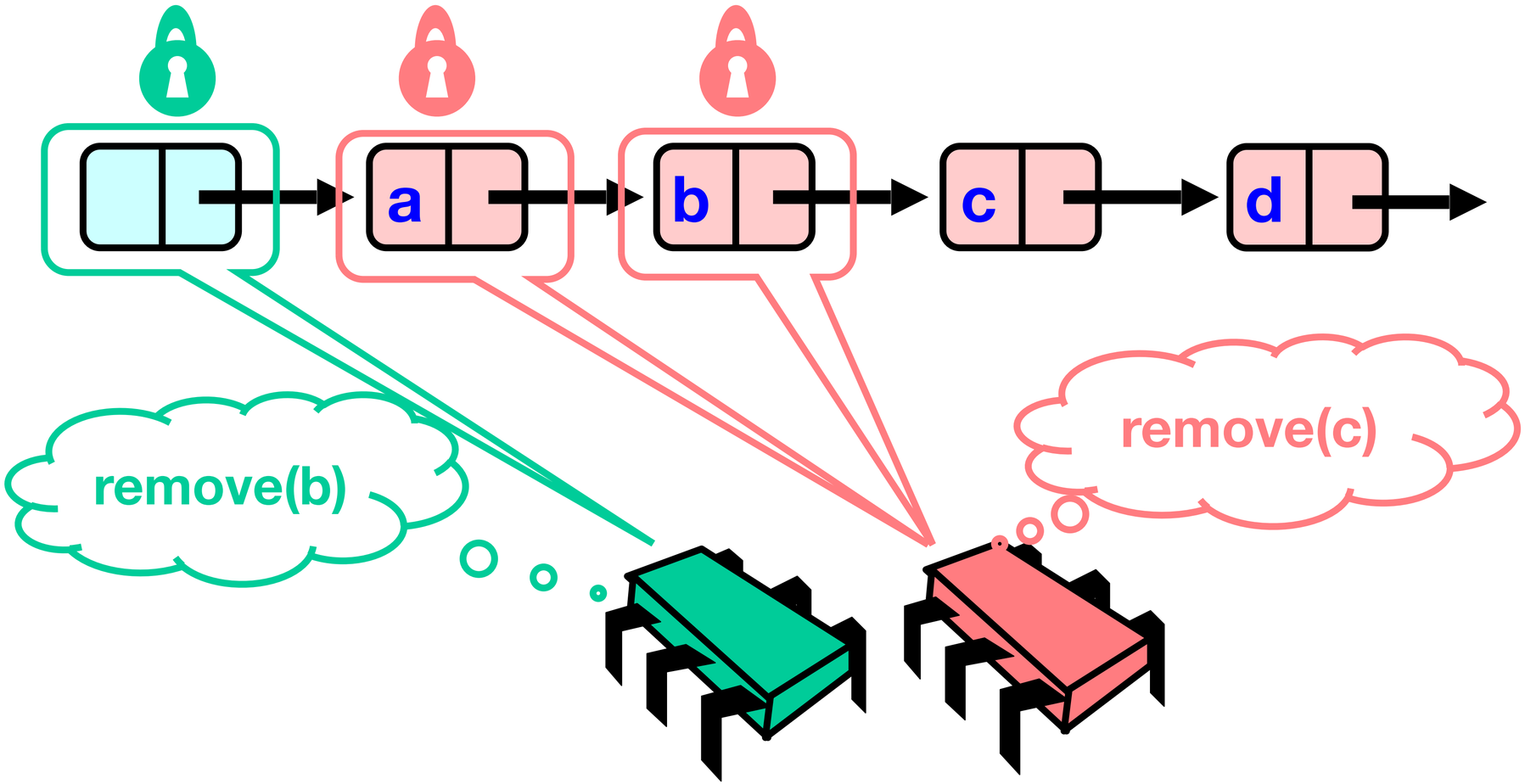
Removing a node



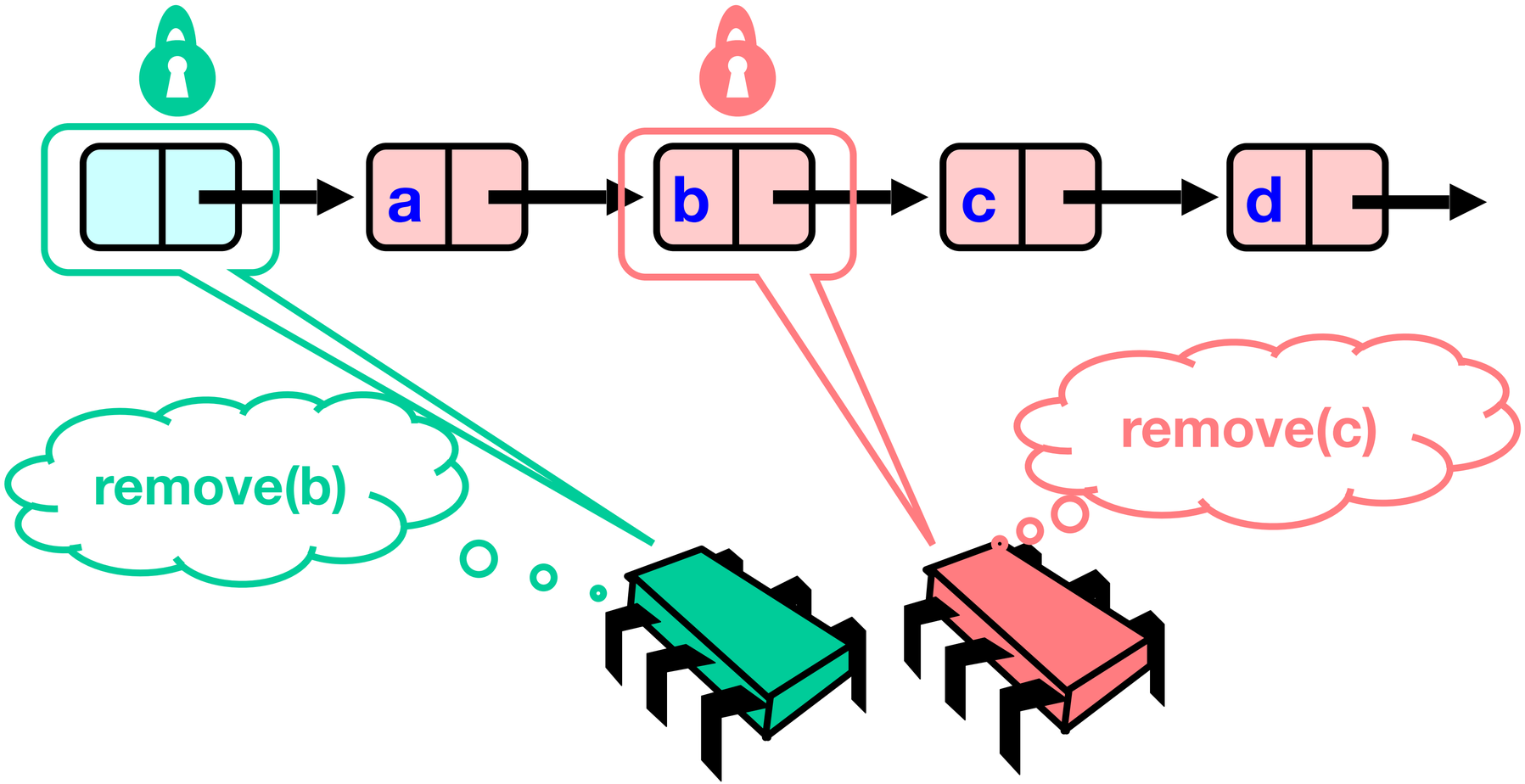
Removing a node



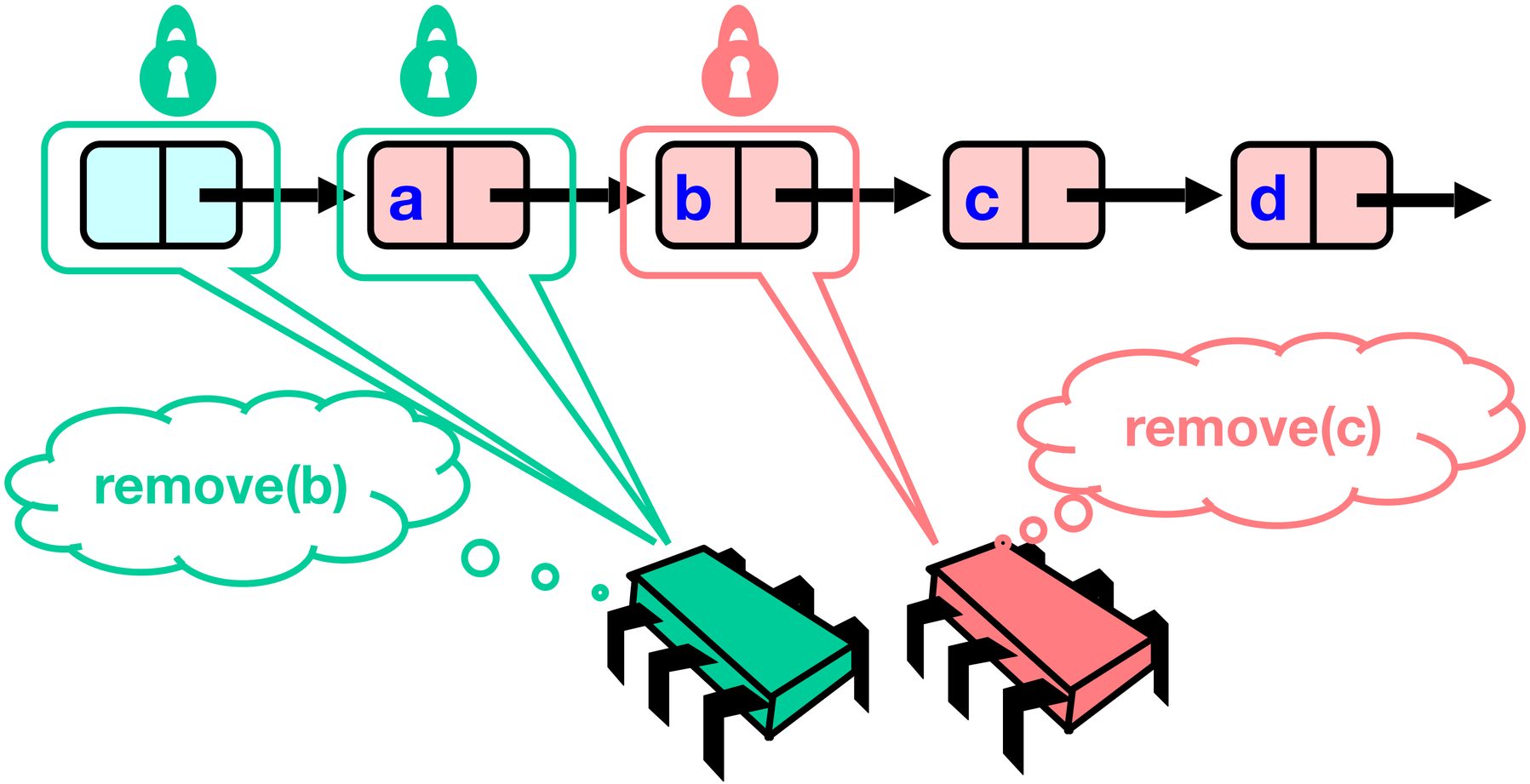
Removing a node



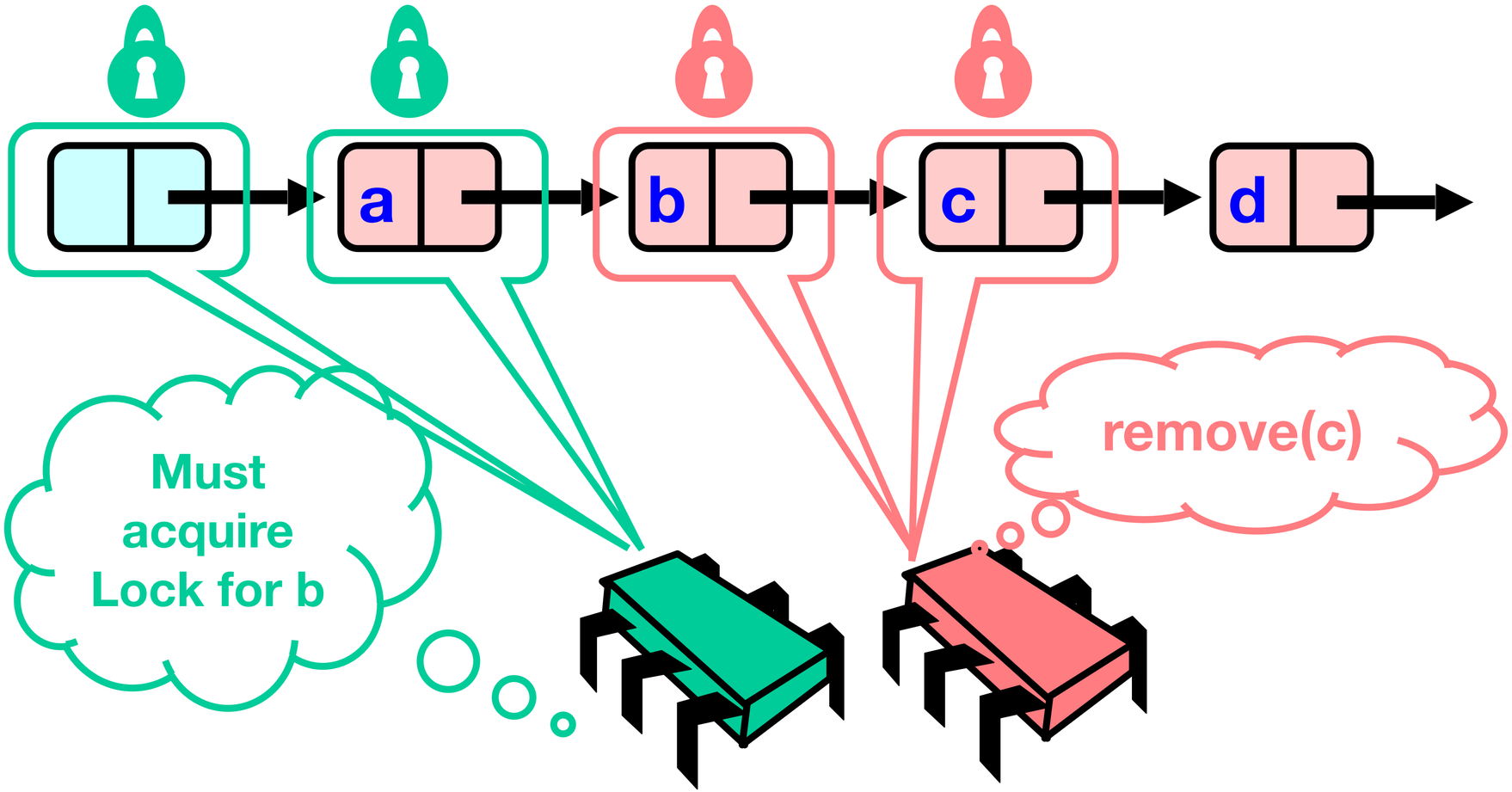
Removing a node



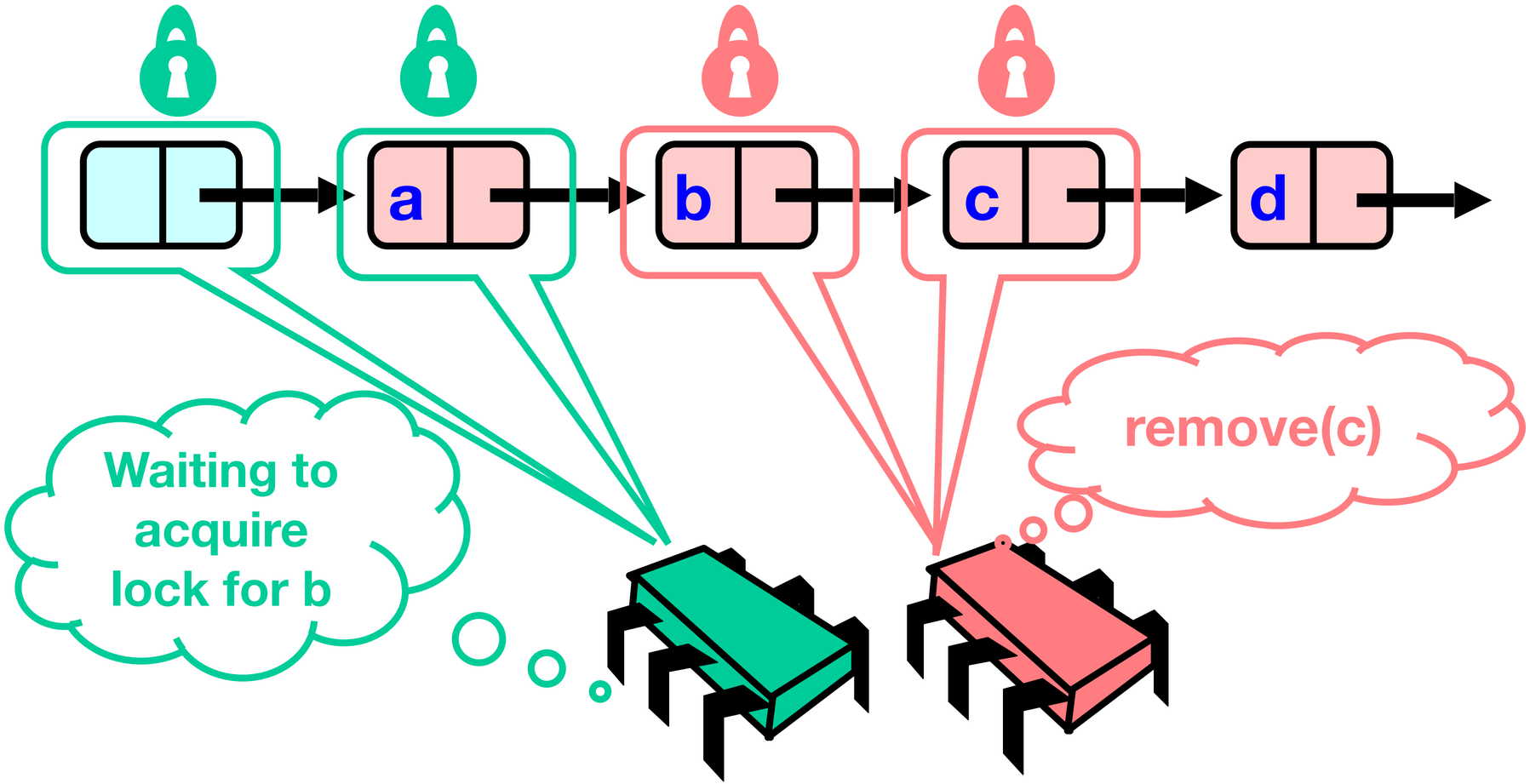
Removing a node



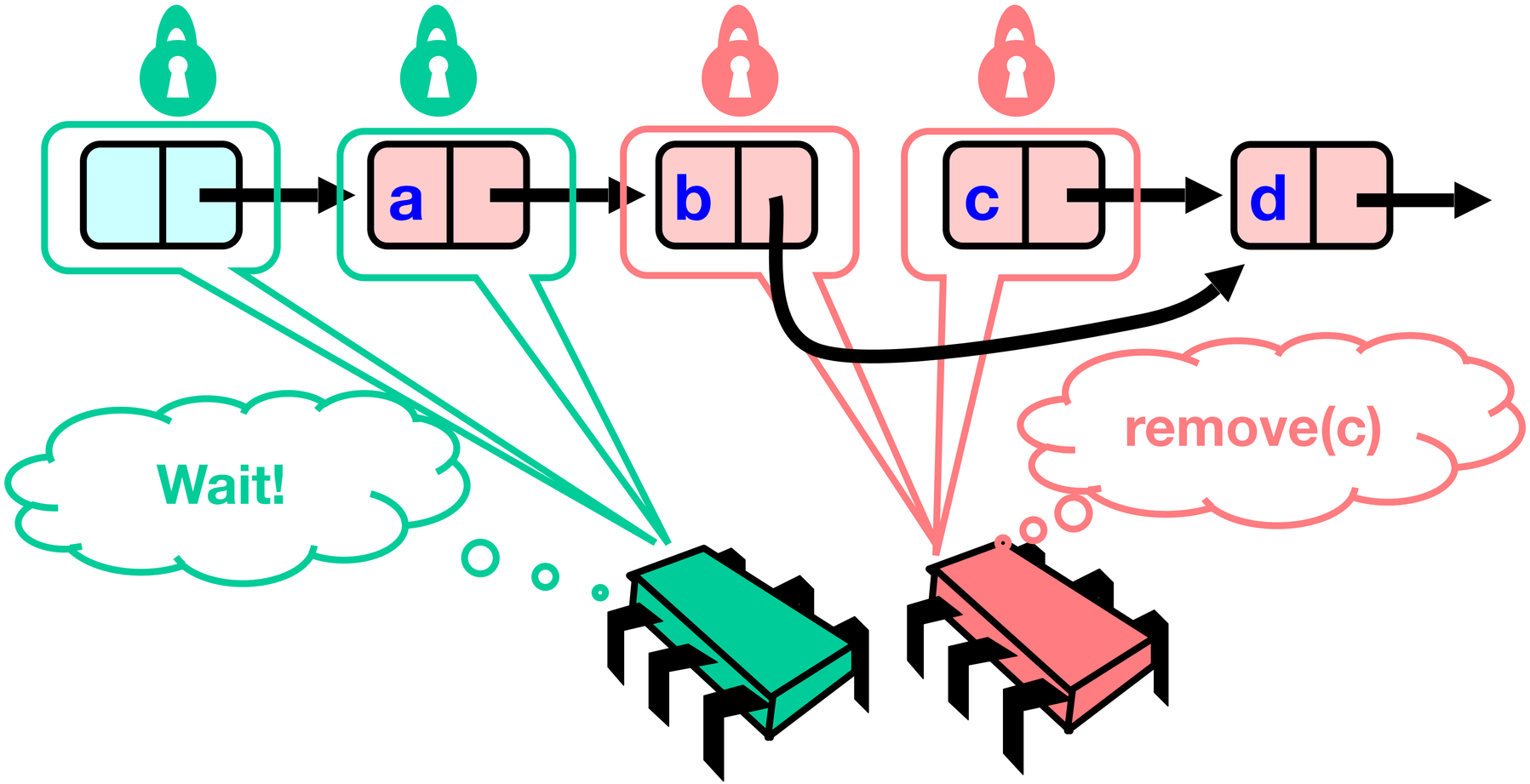
Removing a node



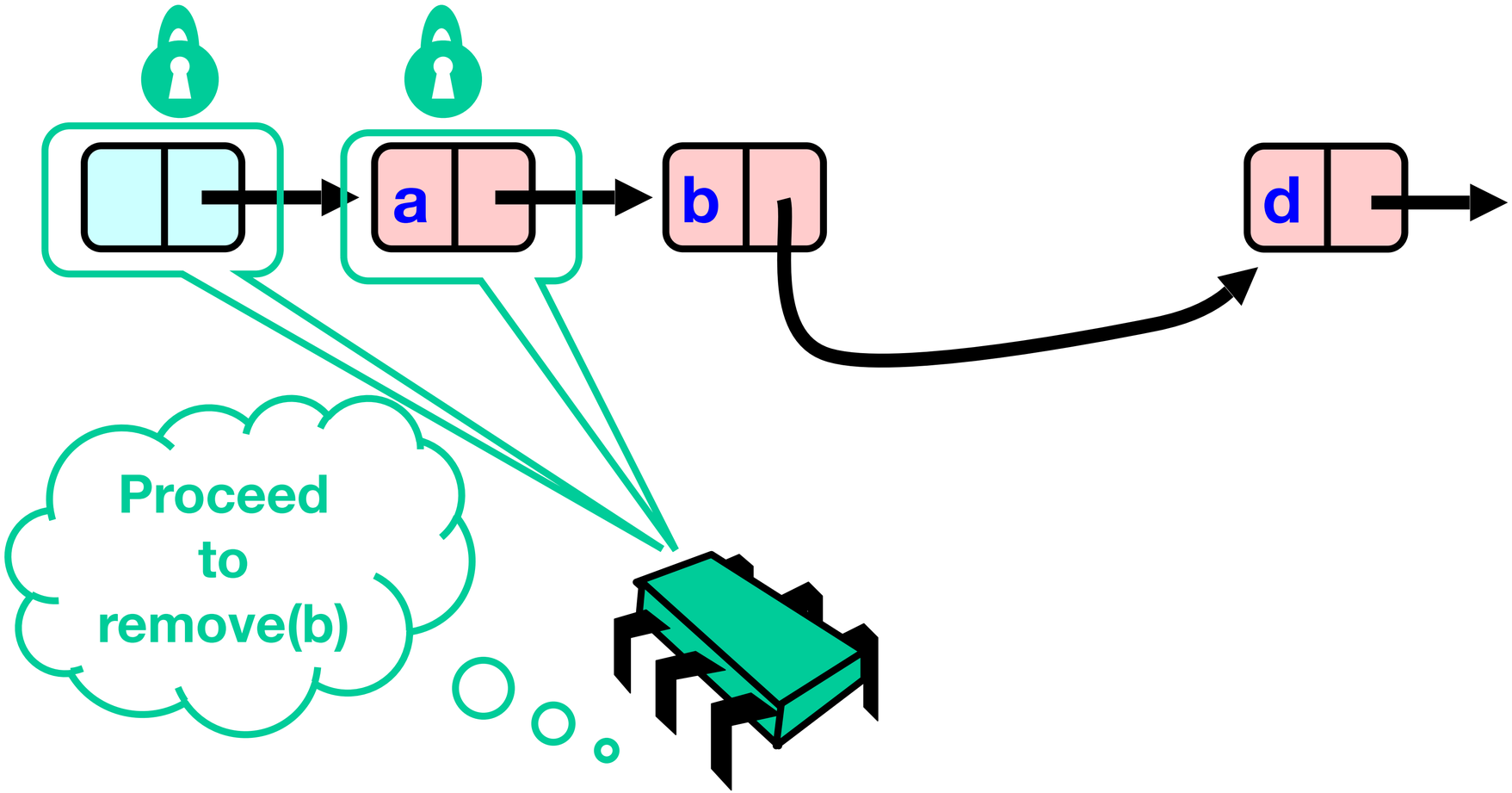
Removing a node



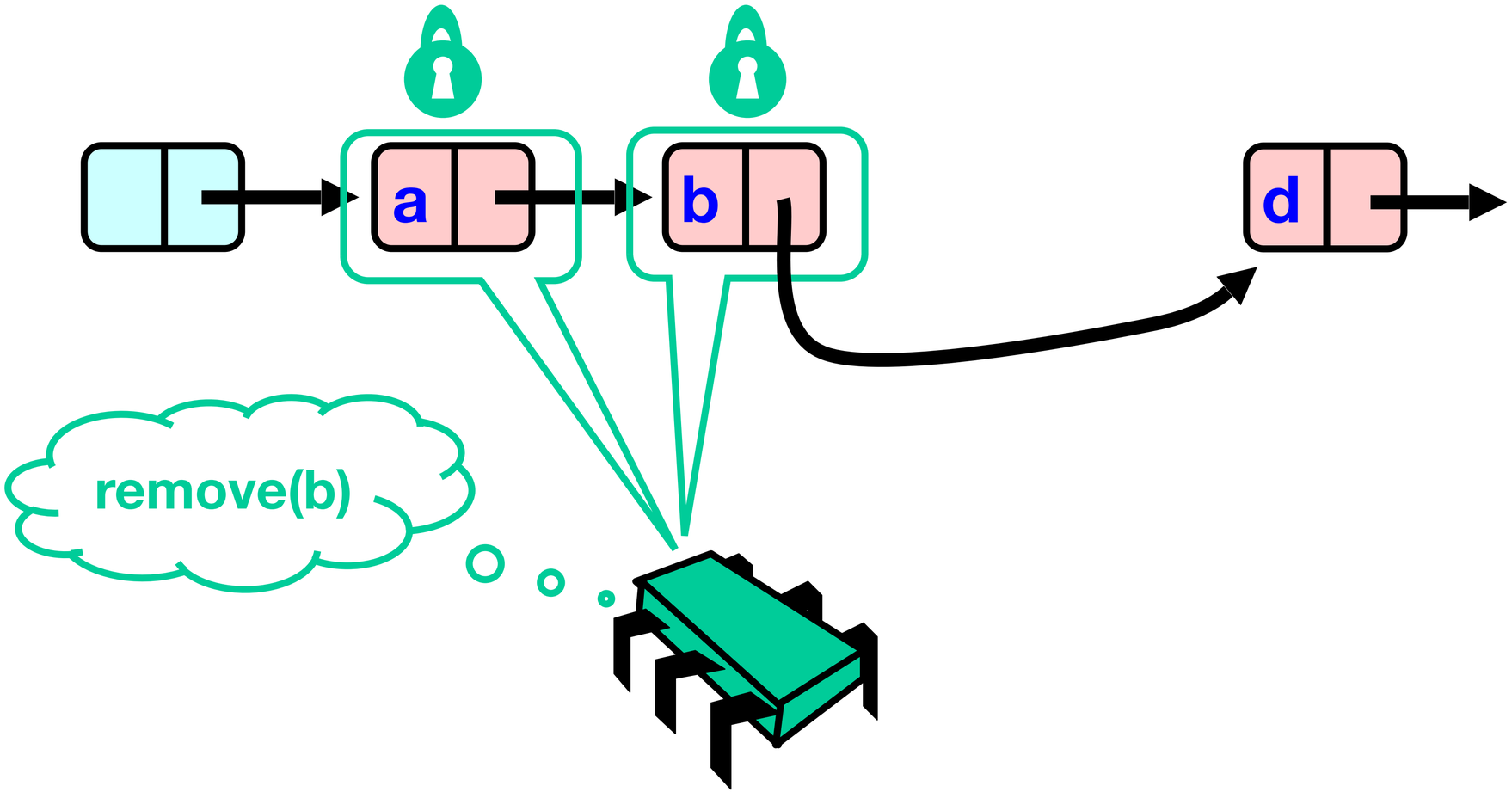
Removing a node



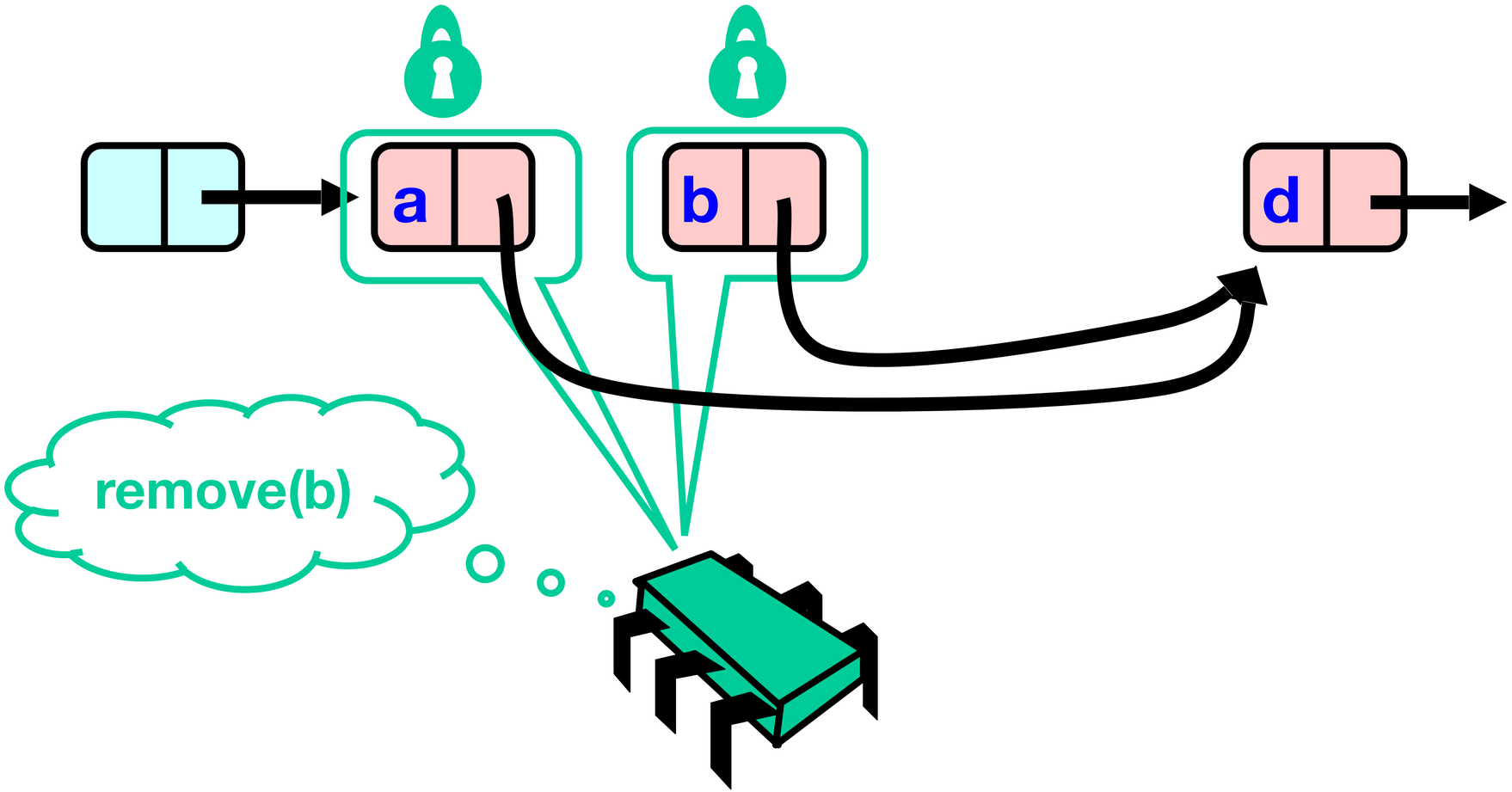
Removing a node



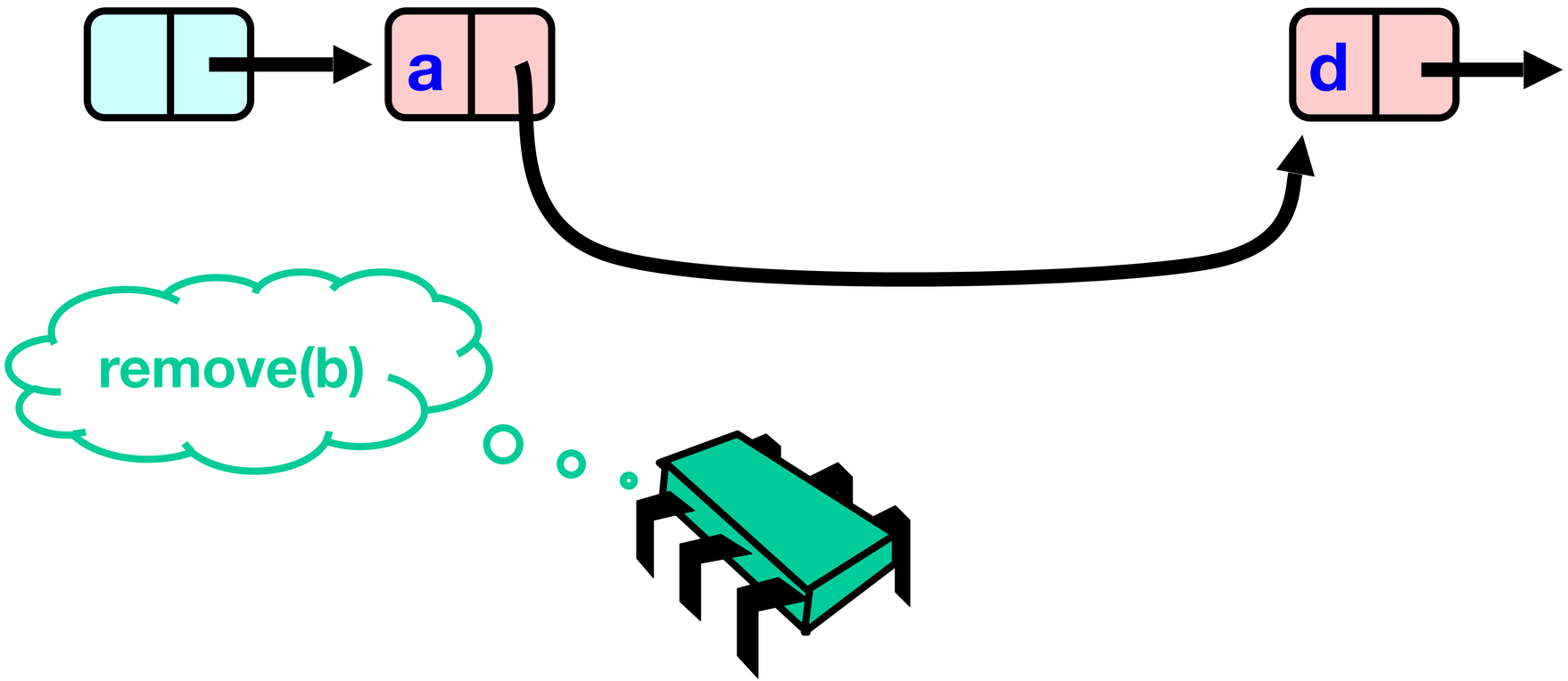
Removing a node



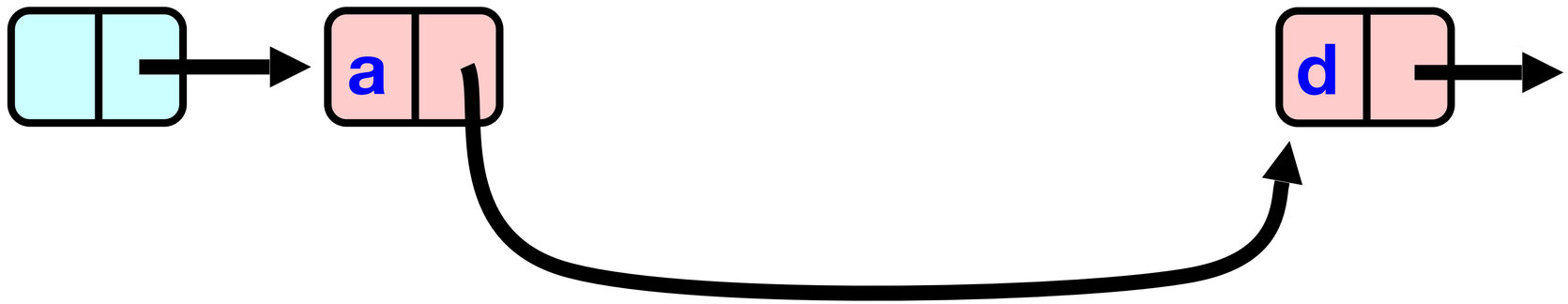
Removing a node



Removing a node



Removing a node



Fine-grained list representation

```
(** Internal node representation *)
type 'a node = {
  item : 'a option;          (* None for sentinel nodes *)
  key : int;                 (* hash code for the item, or min_int/max_int for sentinels *)
  mutable next : 'a node;    (* next node in the list, tail points to itself *)
  lock : Mutex.t;            (* lock for this individual node *)
}

(** The fine-grained list type *)
type 'a t = {
  head : 'a node;           (* sentinel node at the start *)
}
```

See fine_list.ml

Traverse Function

- **Hand-over-hand traversal**
  - find the position where `key` belongs.
- **Pre-condition on `(key, pred, curr)`**
  - `pred.lock` and `curr.lock` are held
  - `pred` and `curr` are adjacent nodes (`pred.next = curr`)
- **Post-condition on returned `(pred, curr)`**
  - `pred.lock` and `curr.lock` are held
  - `pred` and `curr` are adjacent nodes (`pred.next = curr`)
  - `curr` is the first node with `curr.key >= key`
  - All intermediate locks have been released

```
let traverse key pred curr =
  let rec loop pred curr =
    if curr.key < key then begin
      let next = curr.next in
      Mutex.unlock pred.lock;
      Mutex.lock next.lock;
      loop curr next
    end else
      (pred, curr)
  in
  loop pred curr
```

Remove function

curr is the first node with curr.key >= key

```
let remove list item =
  let key = Hashtbl.hash item in
  let head = list.head in
  Mutex.lock head.lock;
  let first = head.next in
  Mutex.lock first.lock;

  let (pred, curr) = traverse key head first in
  let result =
    if curr.key = key then begin
      (* element found, remove it *)
      pred.next <- curr.next;
      true
    end else
      false  (* element not present *)
  in
  Mutex.unlock curr.lock;
  Mutex.unlock pred.lock;
  result
```

Why remove is linearizable

```
let remove list item =
  let key = Hashtbl.hash item in
  let head = list.head in
  Mutex.lock head.lock;
  let first = head.next in
  Mutex.lock first.lock;

  let (pred, curr) = traverse key head first in
  let result =
    if curr.key = key then begin
      (* element found, remove it *)
      pred.next <- curr.next;
      true
    end else
      false  (* element not present *)
  in
  Mutex.unlock curr.lock;
  Mutex.unlock pred.lock;
  result
```

- `pred` reachable from head
- `curr` is `pred.next`
- So `curr.item` is in the list

Why remove is linearizable

```
let remove list item =
  let key = Hashtbl.hash item in
  let head = list.head in
  Mutex.lock head.lock;
  let first = head.next in
  Mutex.lock first.lock;

  let (pred, curr) = traverse key head first in
  let result =
    if curr.key = key then begin
      (* element found, remove it *)
      pred.next <- curr.next;
      true
    end else
      false  (* element not present *)
  in
  Mutex.unlock curr.lock;
  Mutex.unlock pred.lock;
  result
```

Linearization point when item is in the list

- `pred` reachable from `head`
- `curr` is `pred.next`
- So `curr.item` is in the list

Why remove is linearizable

```
let remove list item =
  let key = Hashtbl.hash item in
  let head = list.head in
  Mutex.lock head.lock;
  let first = head.next in
  Mutex.lock first.lock;

  let (pred, curr) = traverse key head first in
  let result =
    if curr.key = key then begin
      (* element found, remove it *)
      pred.next <- curr.next;
      true
    end else
      false  (* element not present *)
  in
  Mutex.unlock curr.lock;
  Mutex.unlock pred.lock;
  result
```

curr is the first node with curr.key >= key

- pred reachable from head
- curr is pred.next
- pred.key < key < curr.key

Why remove is linearizable

```
let remove list item =
  let key = Hashtbl.hash item in
  let head = list.head in
  Mutex.lock head.lock;
  let first = head.next in
  Mutex.lock first.lock;

  let (pred, curr) = traverse key head first in
  let result =
    if curr.key = key then begin
      (* element found, remove it *)
      pred.next <- curr.next;
      true
    end else
      false  (* element not present *)
  in
  Mutex.unlock curr.lock;
  Mutex.unlock pred.lock;
  result
```

curr is the first node with curr.key >= key

Where is the linearization point when the item is not found?

- `pred` reachable from `head`
- `curr` is `pred.next`
- `pred.key < key < curr.key`

Why remove is linearizable

```
let remove list item =
  let key = Hashtbl.hash item in
  let head = list.head in
  Mutex.lock head.lock;
  let first = head.next in
  Mutex.lock first.lock;

  let (pred, curr) = traverse key head first in
  let result =
    if curr.key = key then begin
      (* element found, remove it *)
      pred.next <- curr.next;
      true
    end else
      false  (* element not present *)
  in
  Mutex.unlock curr.lock;
  Mutex.unlock pred.lock;
  result
```

```
let traverse key pred curr =
  let rec loop pred curr =
    if curr.key < key then begin
      let next = curr.next in
      Mutex.unlock pred.lock;
      Mutex.lock next.lock;
      loop curr next
    end else
      (pred, curr)
  in
  loop pred curr
```

Linearization point when item is ***not*** in the list

Where is the linearization point when the item is not found?

- `pred` reachable from `head`
- `curr` is `pred.next`
- `pred.key < key < curr.key`

Adding nodes

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

Same Abstraction Map

$$S(head) = \{x \mid \exists a \,.\, a \text{ is reachable from } head \wedge a \,.\, item = x\}$$

Rep Invariant

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

Drawbacks

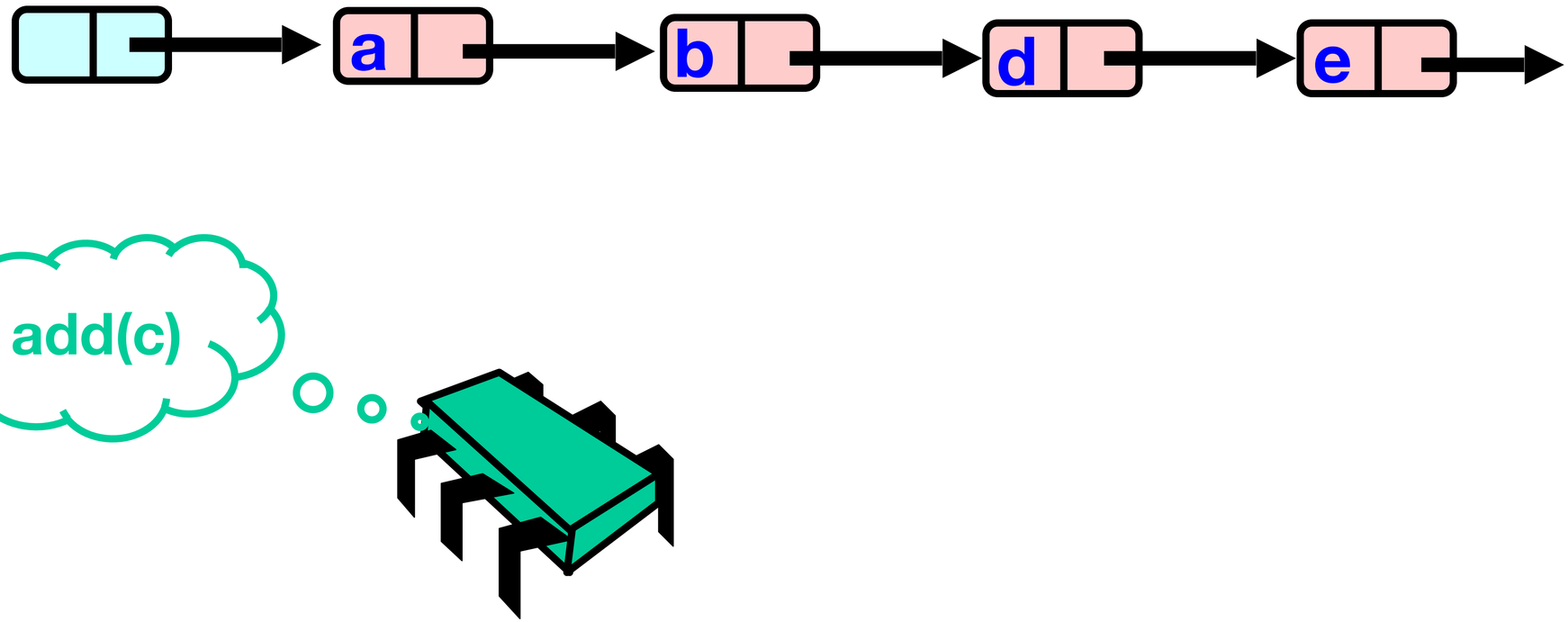
- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient

(2) Optimistic Synchronization

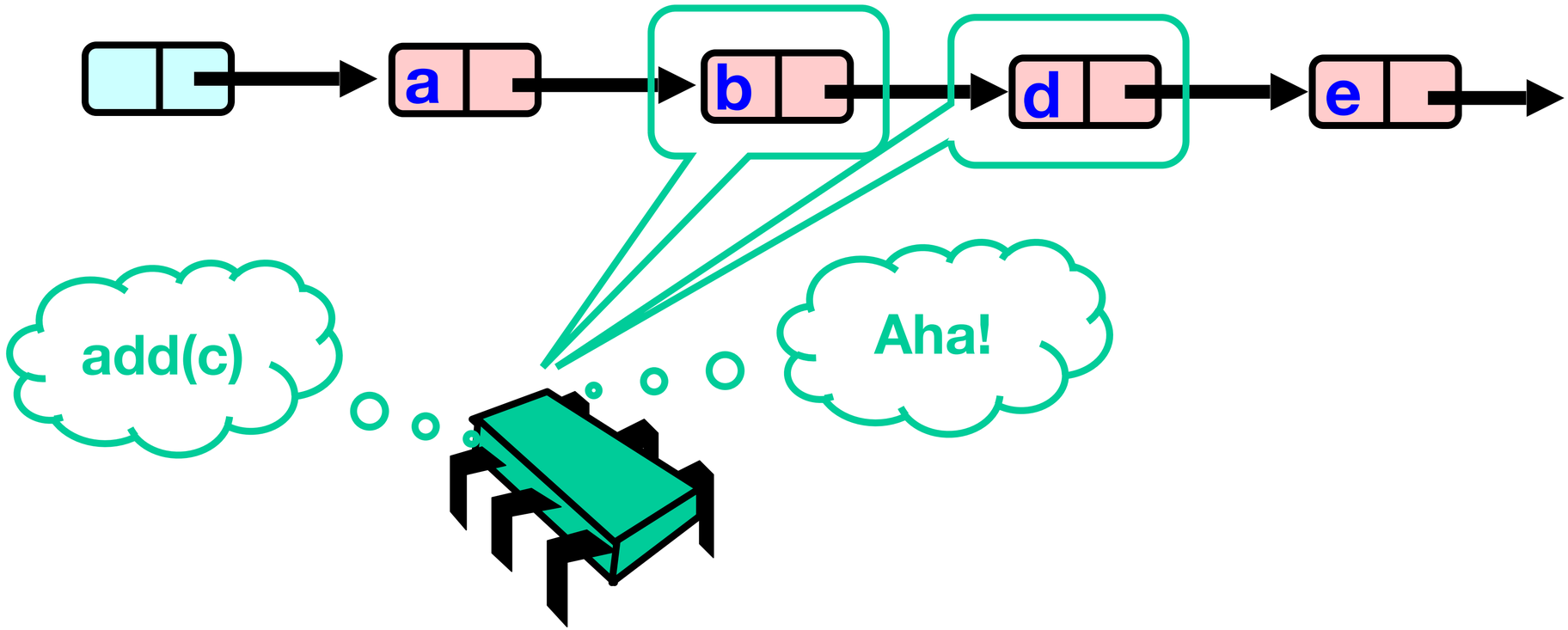
Optimistic Synchronization

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

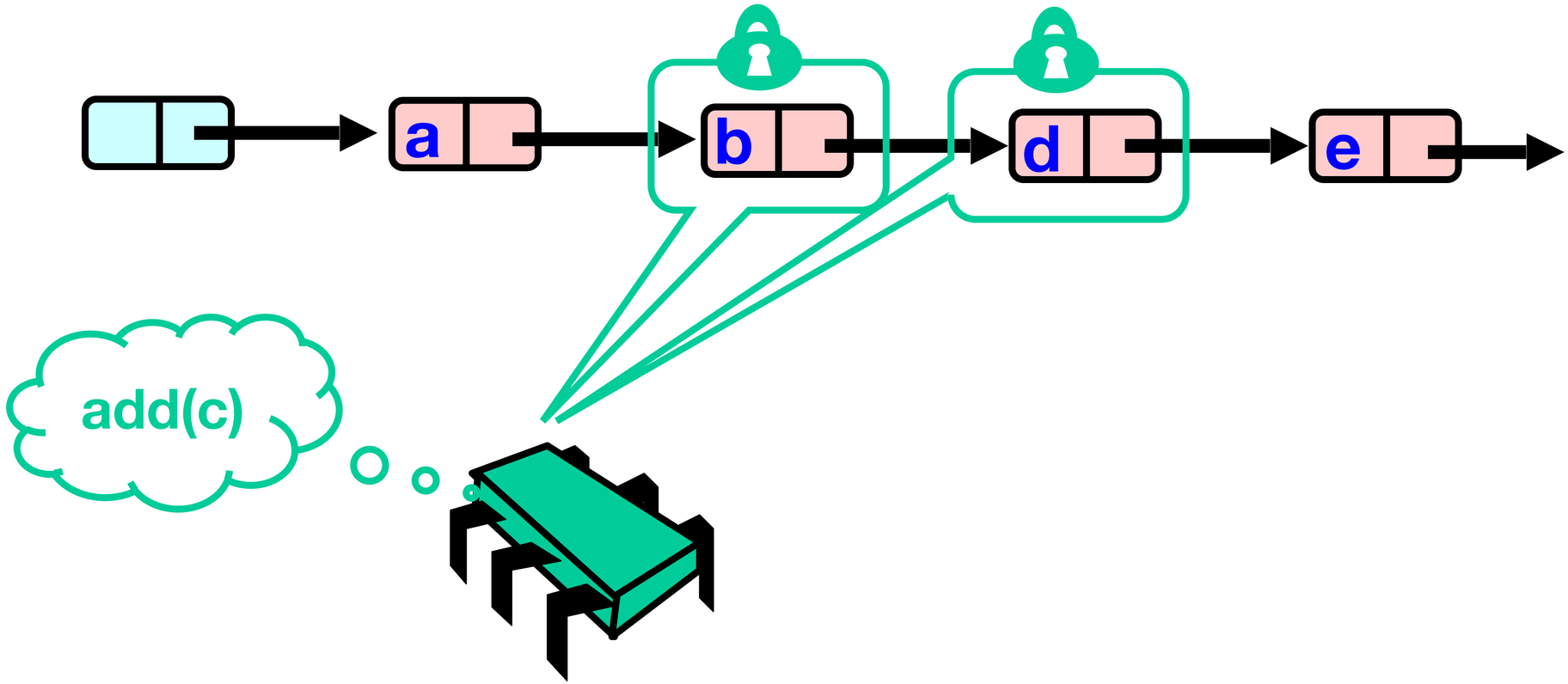
Optimistic: Traverse without locking



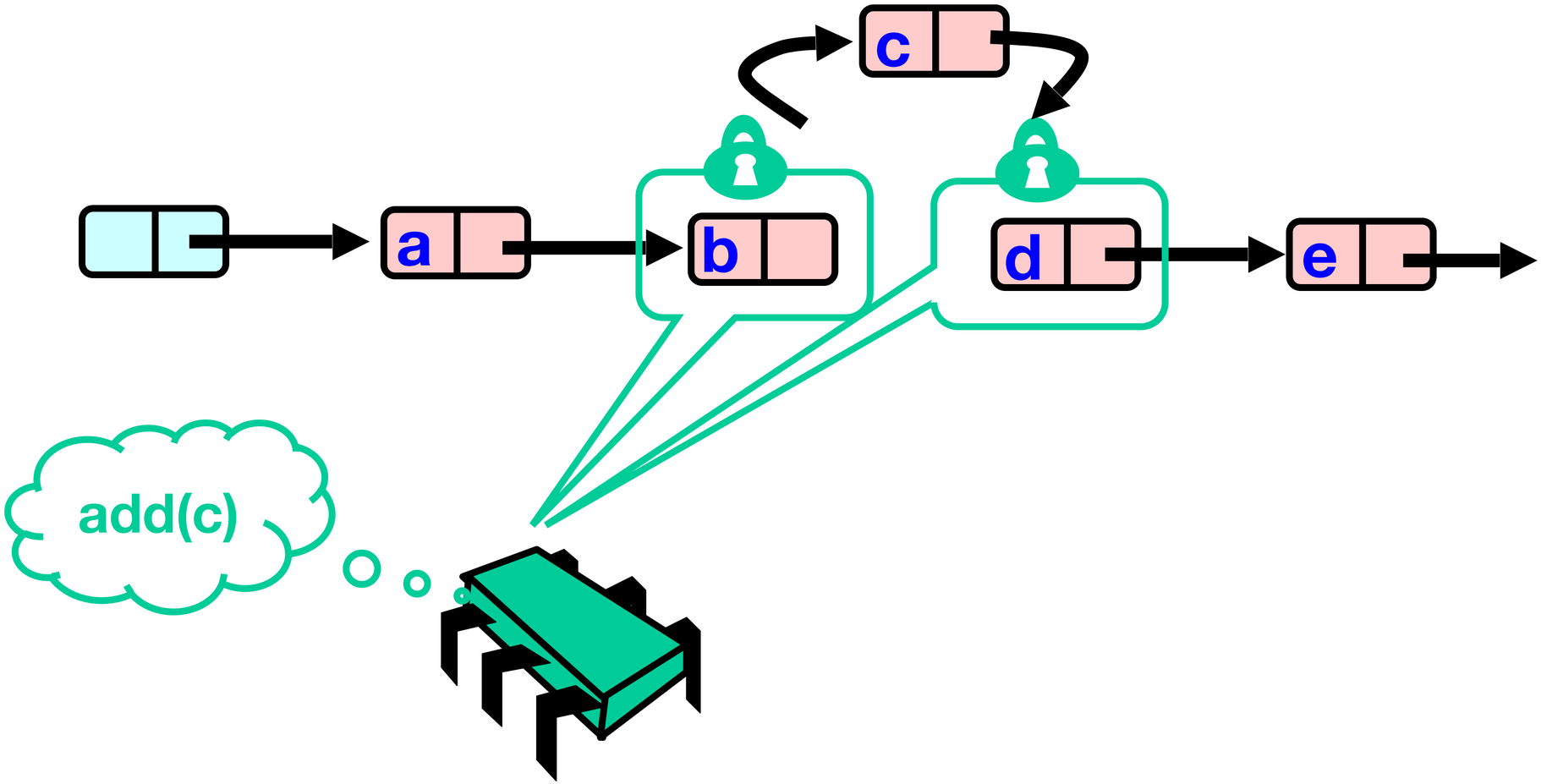
Optimistic: Traverse without locking



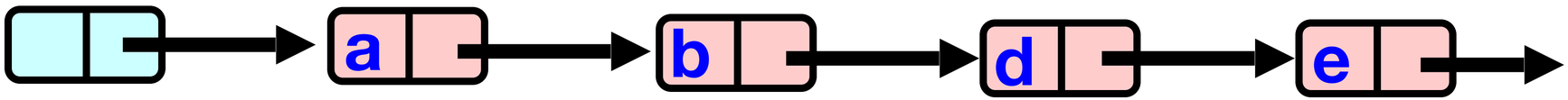
Optimistic: Lock and load

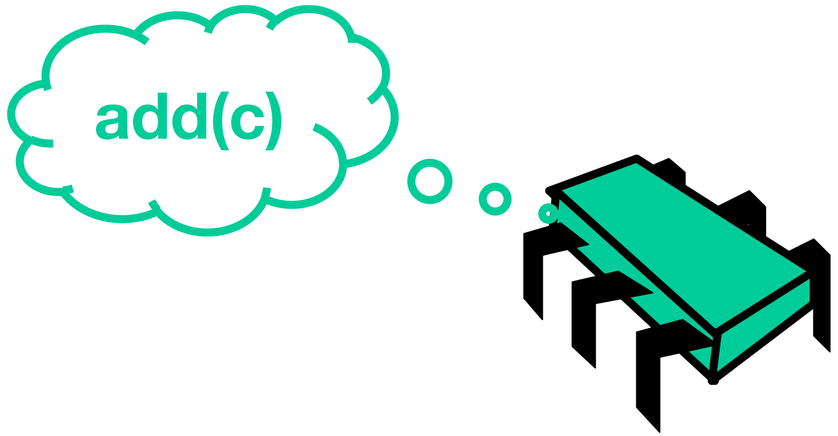


Optimistic: Lock and load

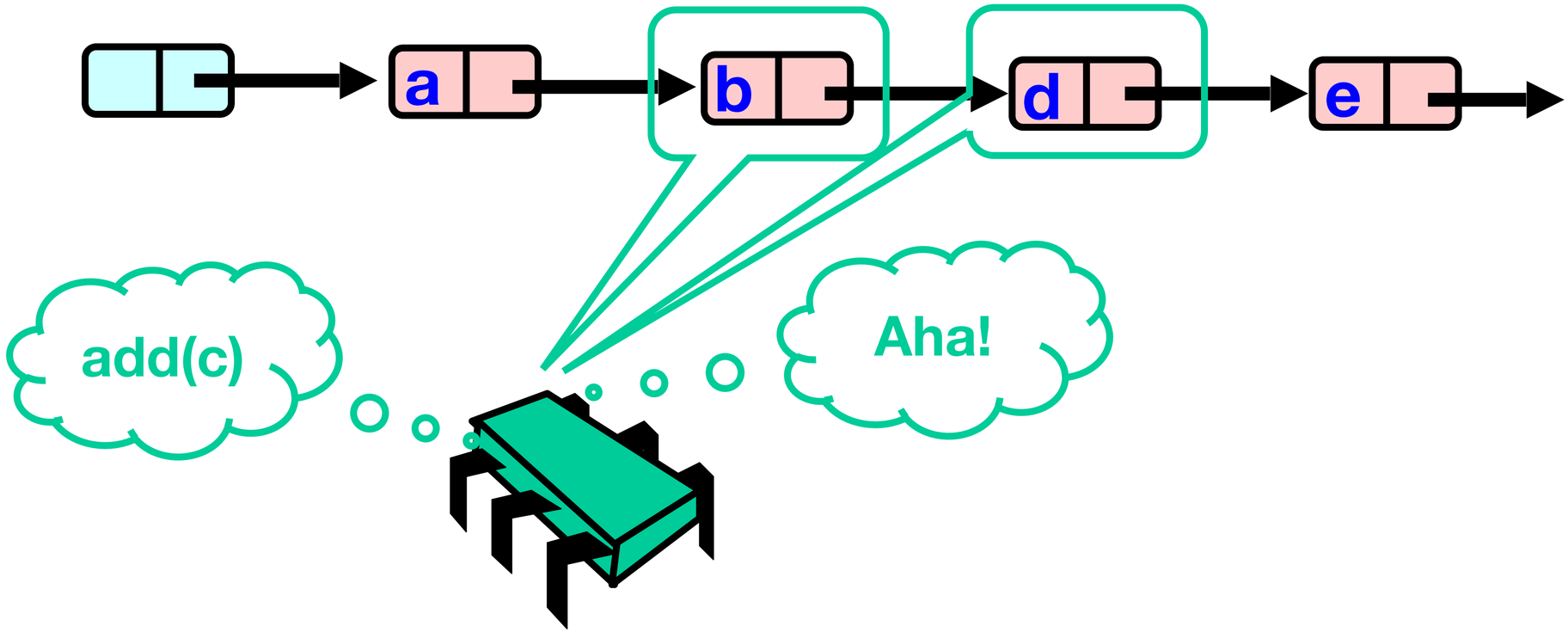


What could go wrong?

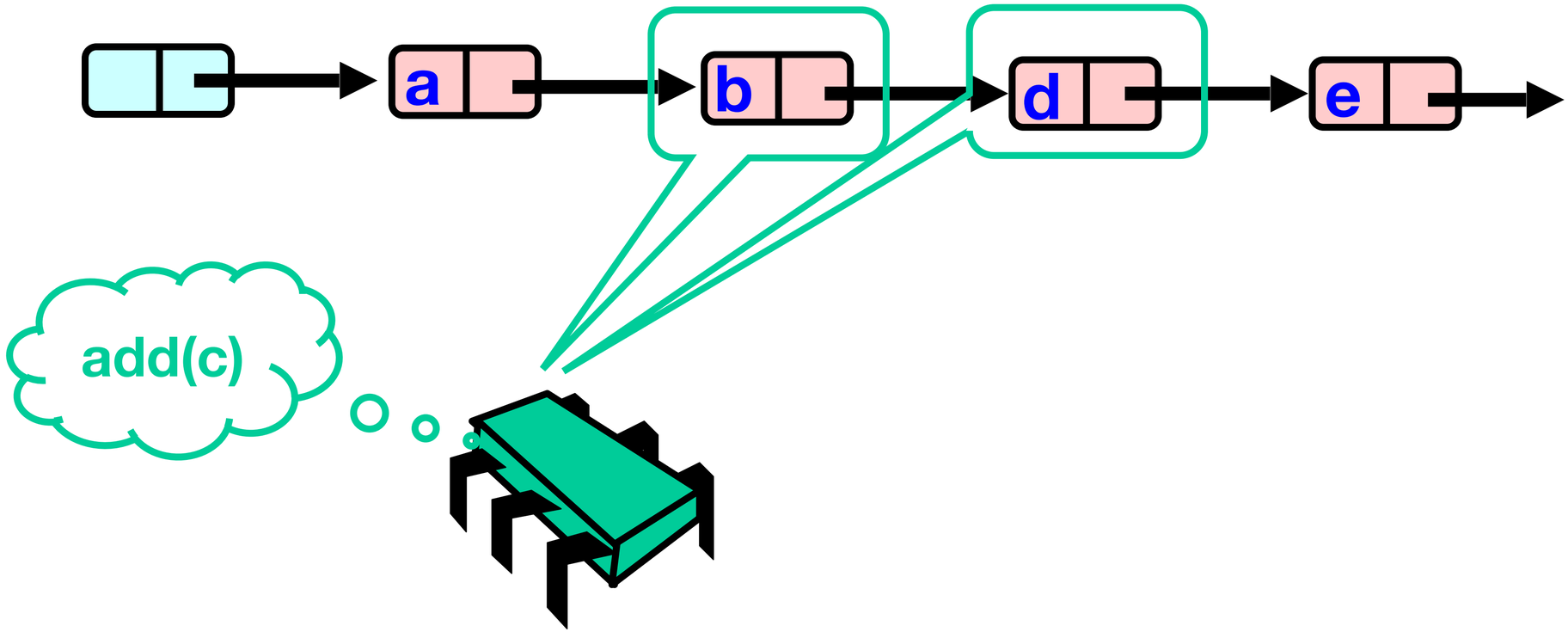




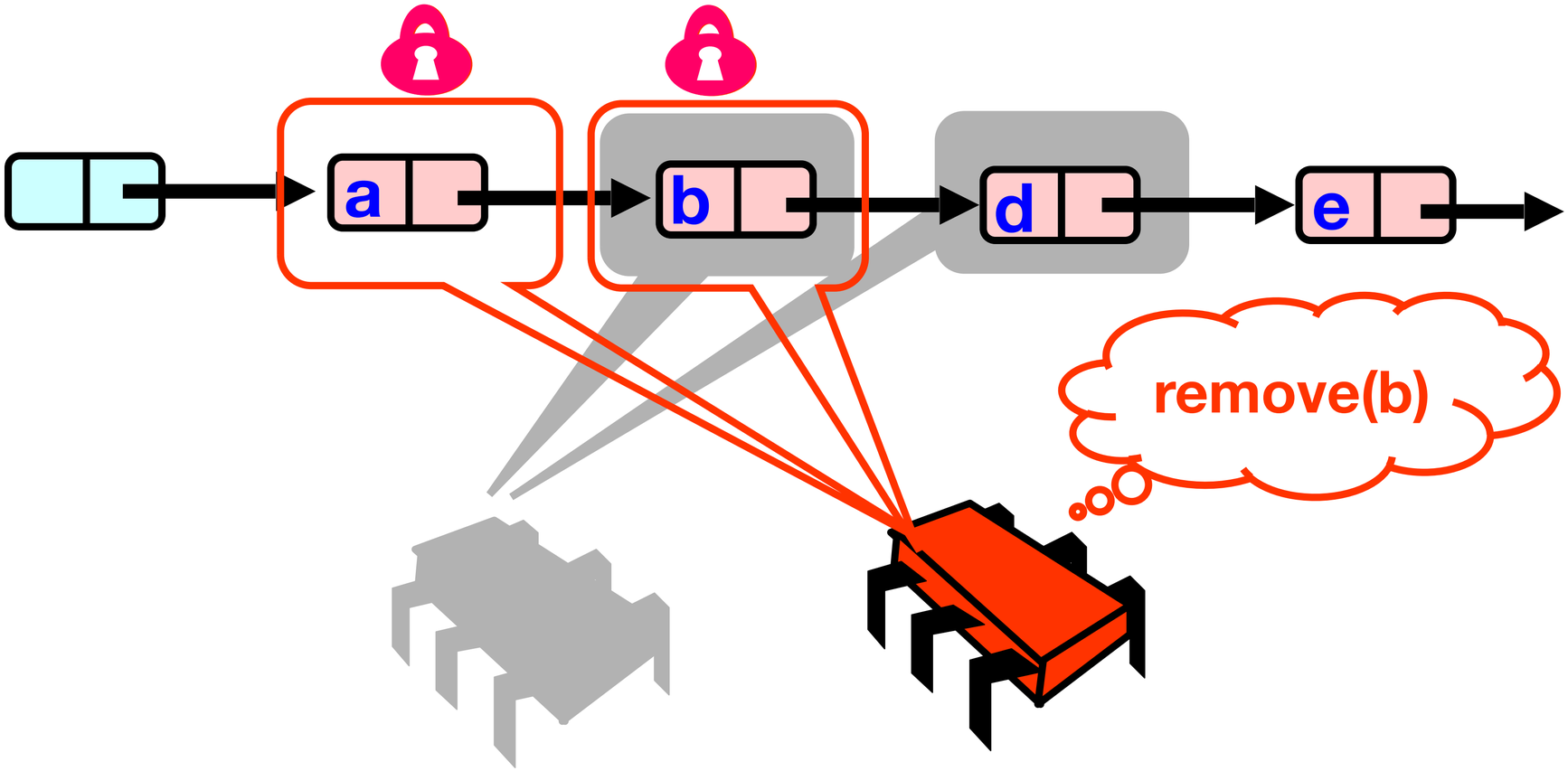
What could go wrong?



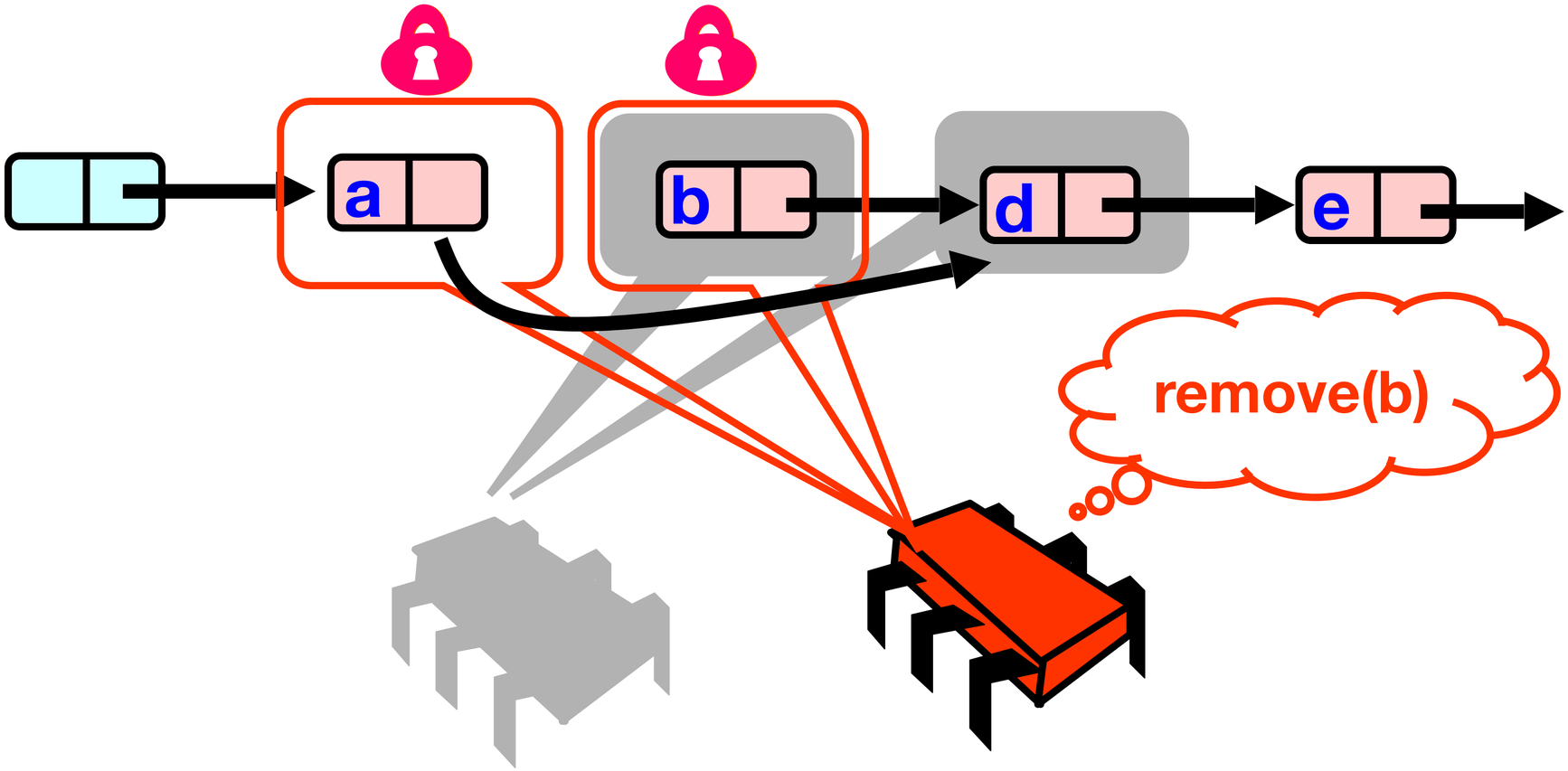
What could go wrong?



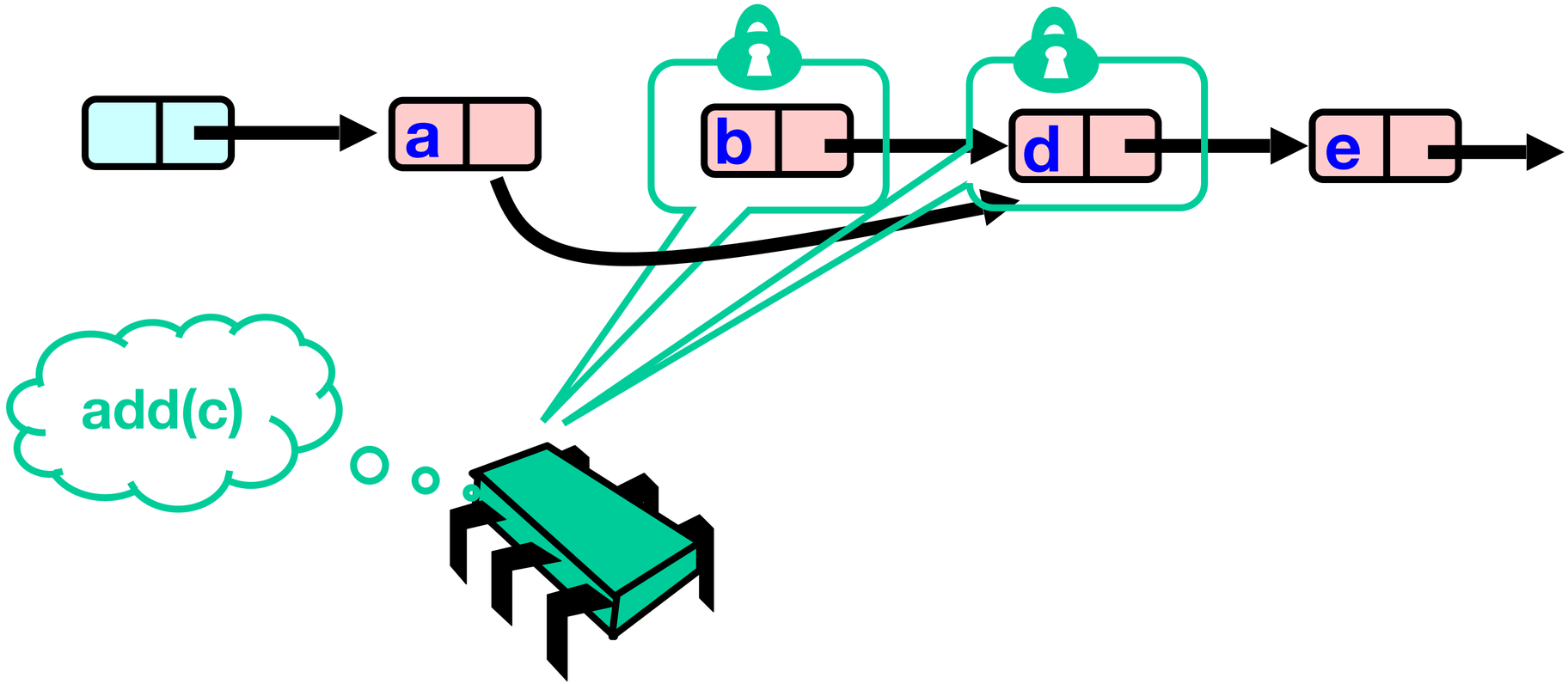
What could go wrong?



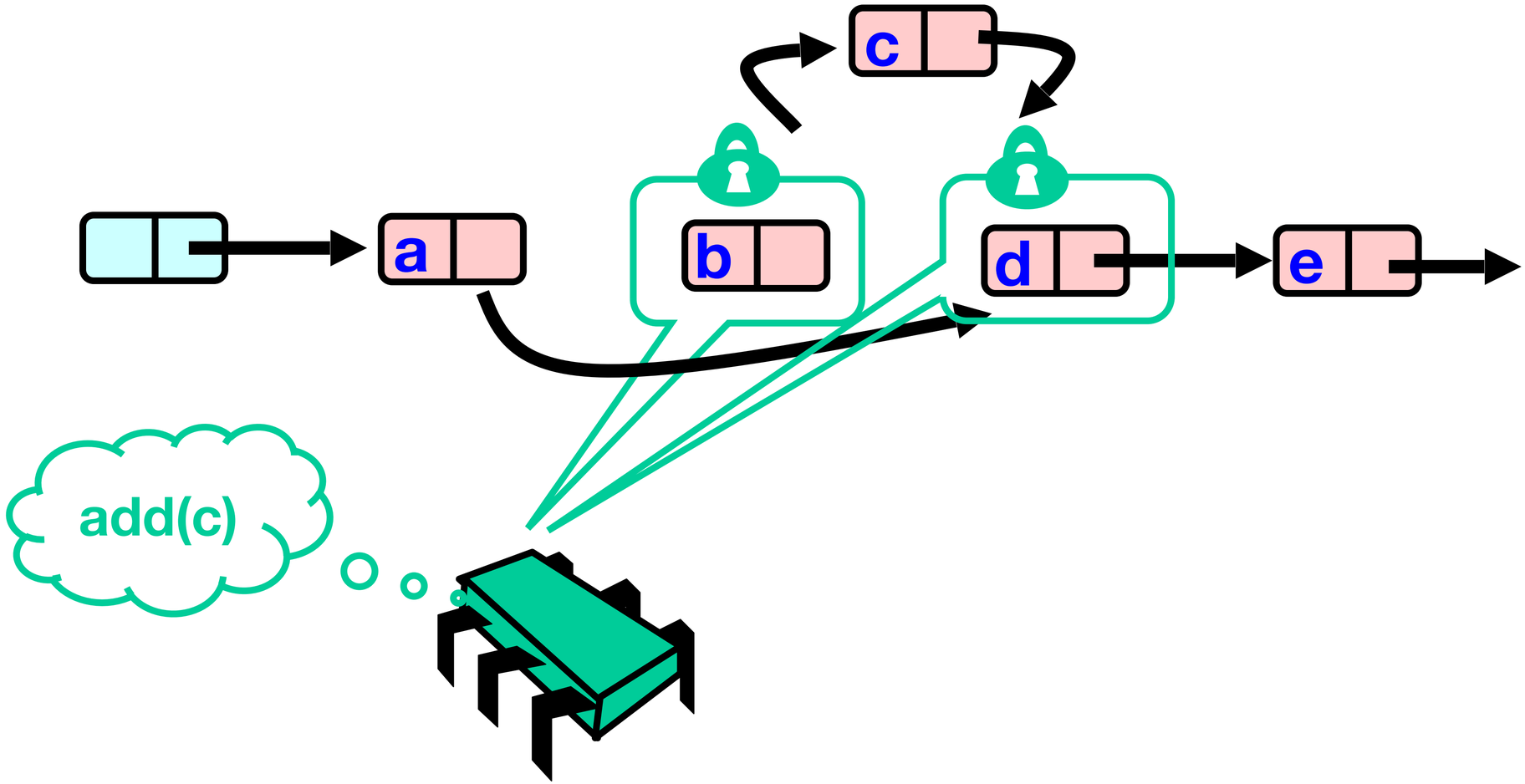
What could go wrong?



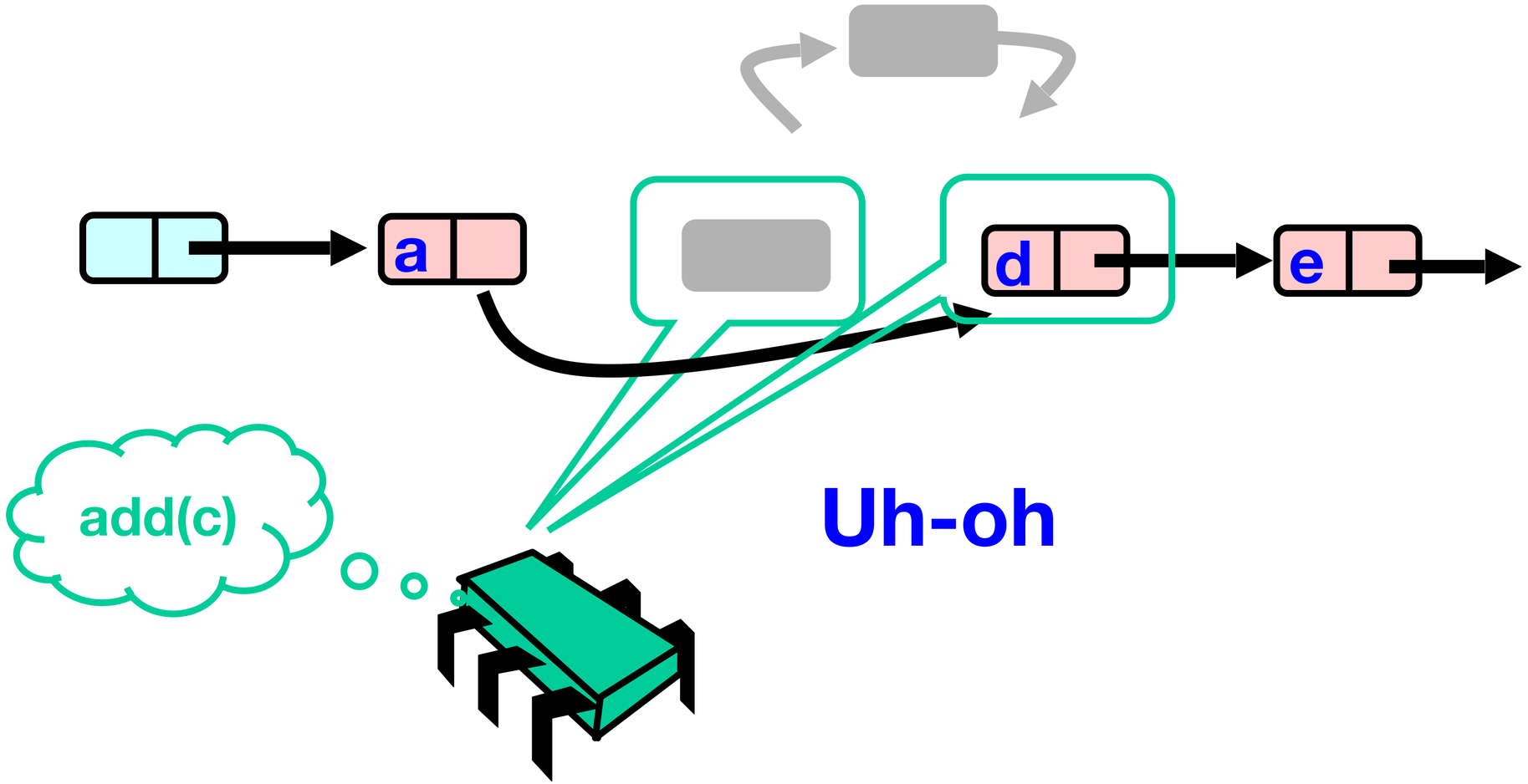
What could go wrong?



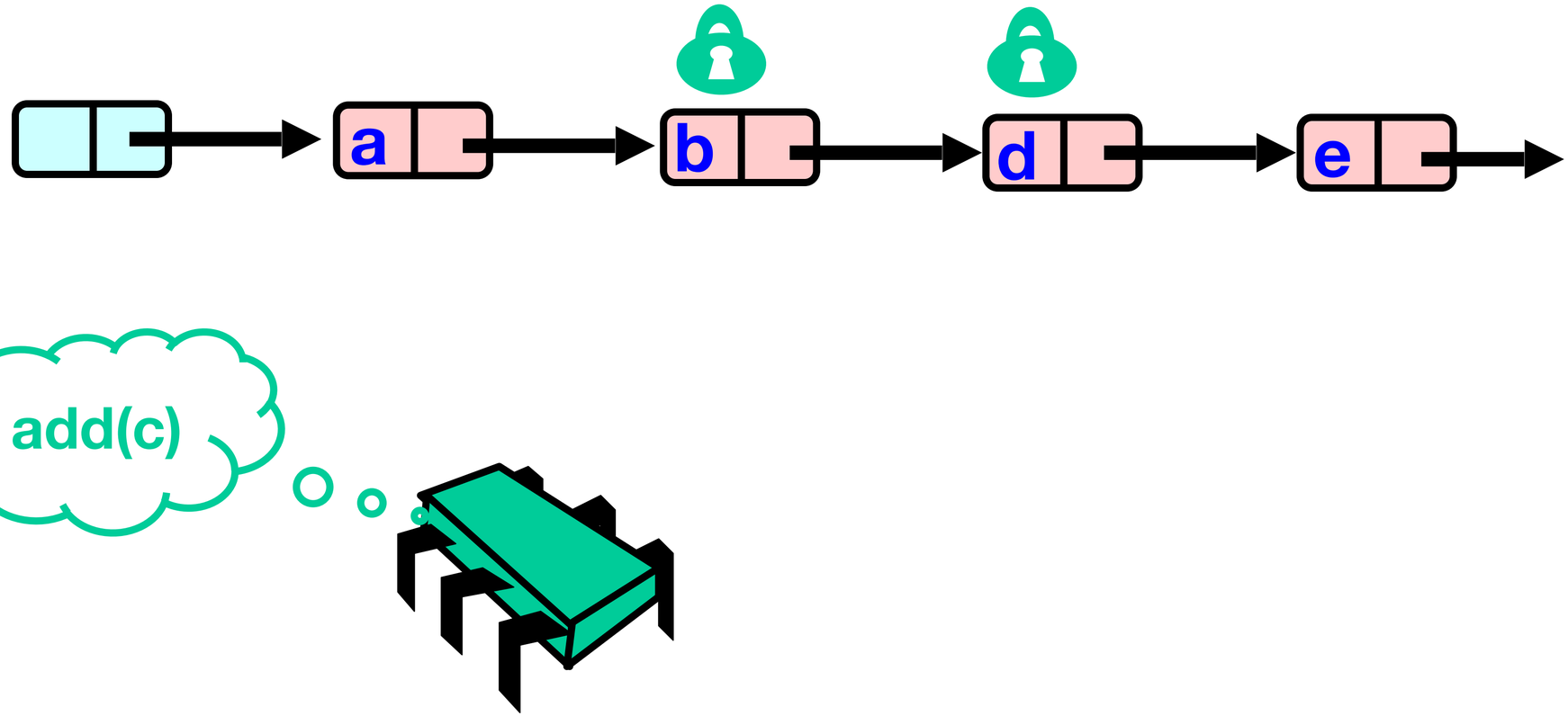
What could go wrong?



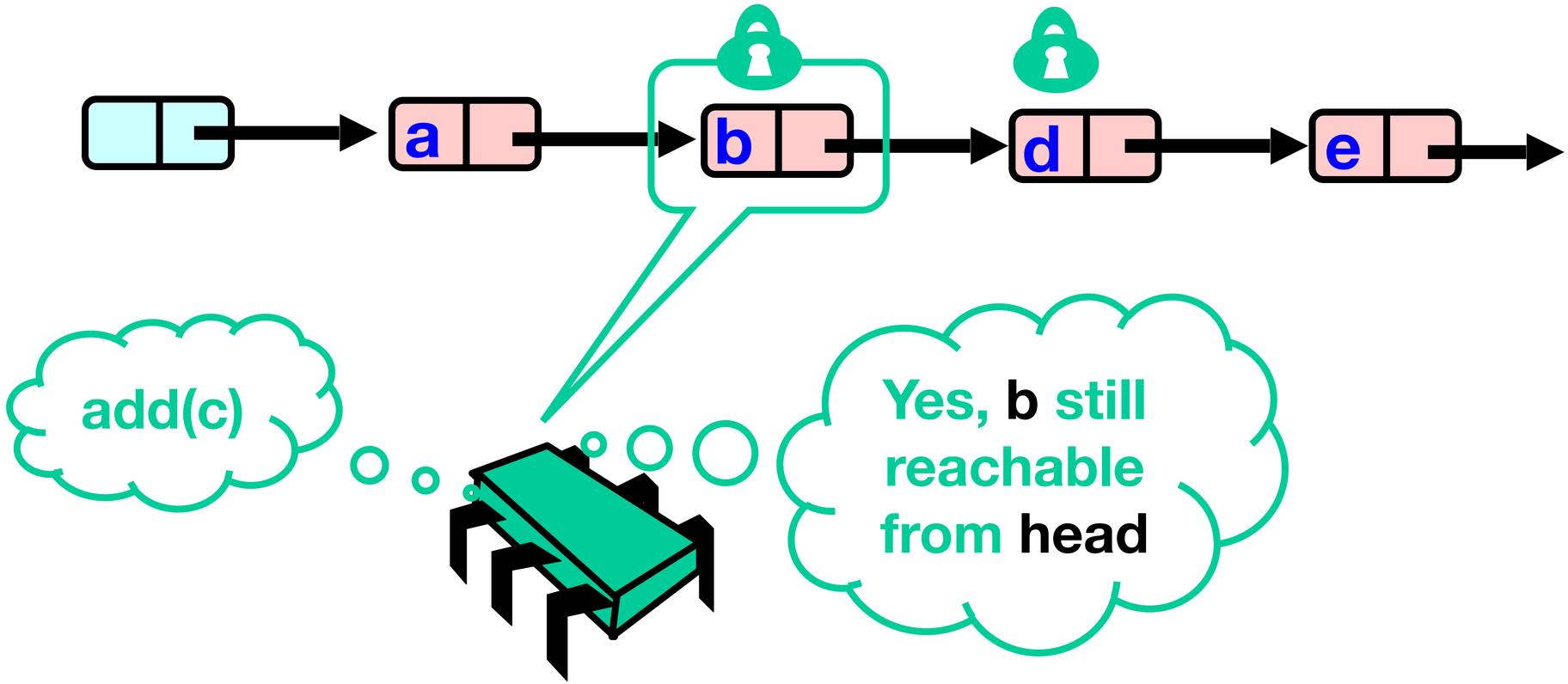
What could go wrong?



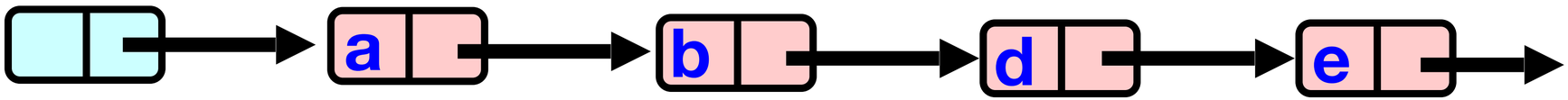
Validate — Part 1

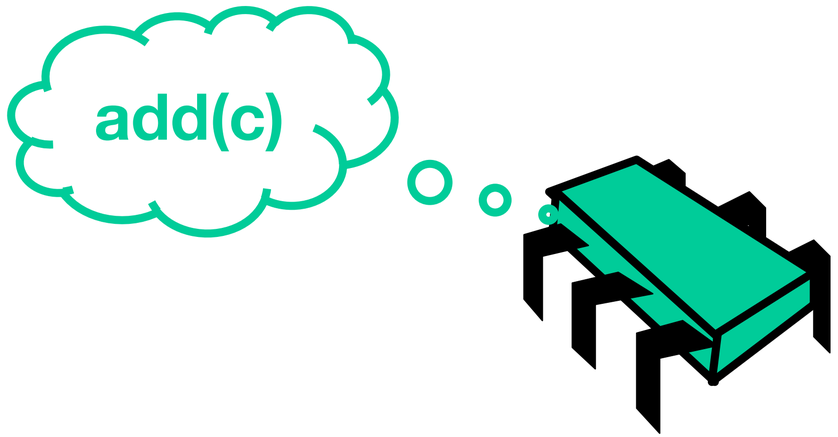


Validate — Part 1

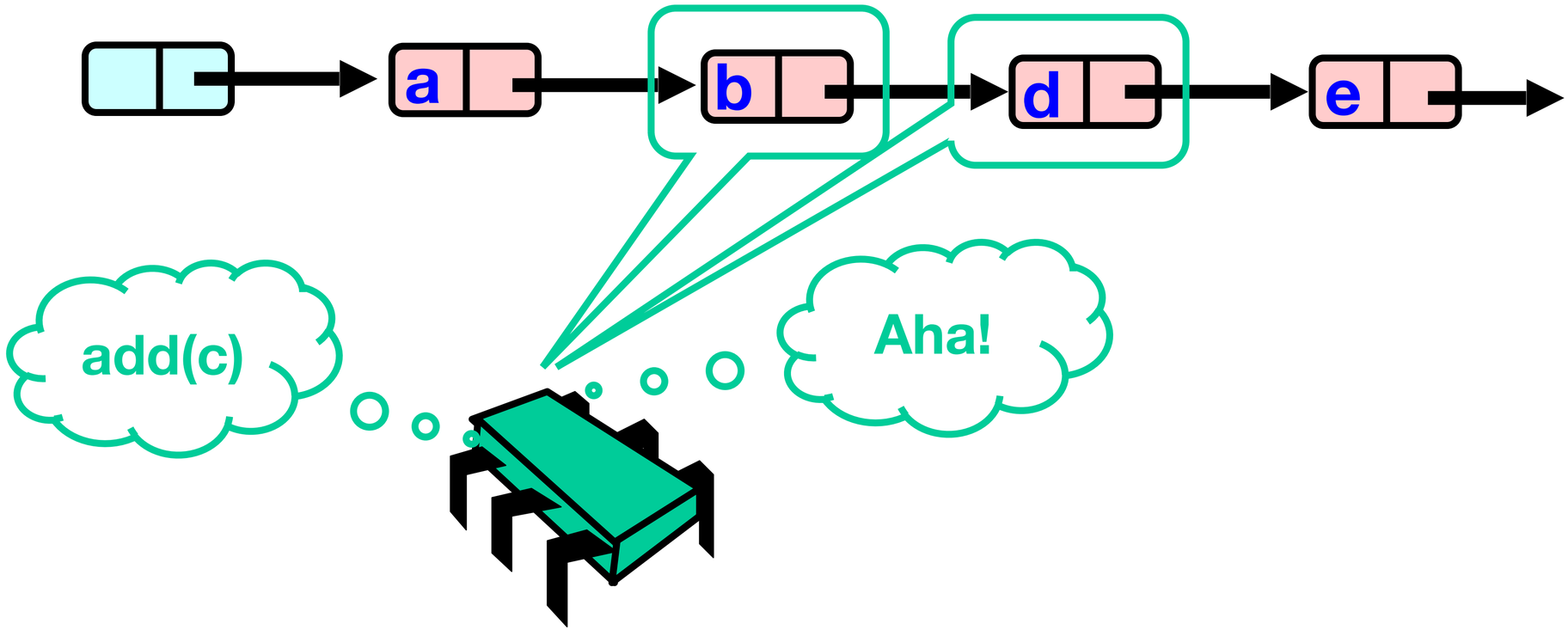


What else could go wrong?

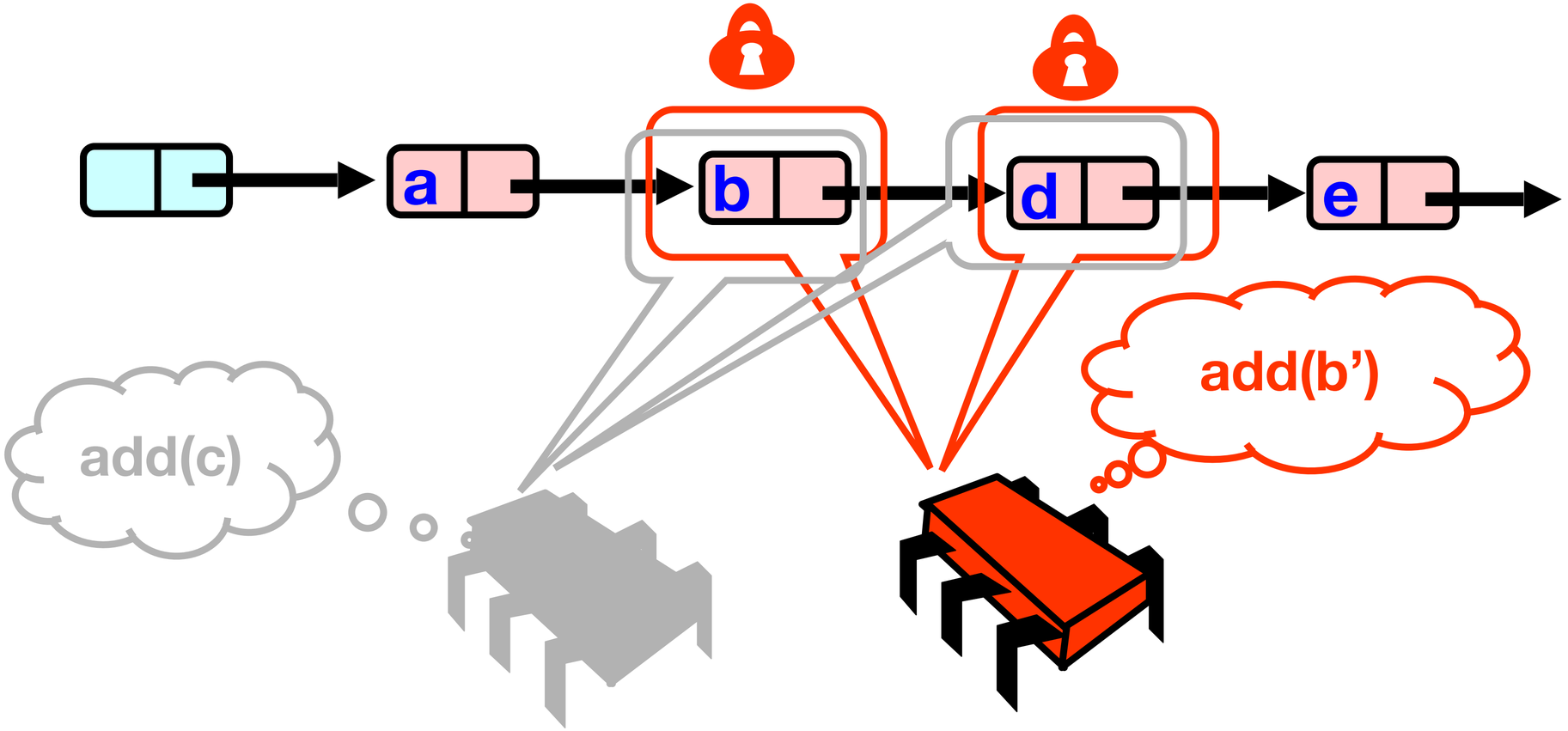




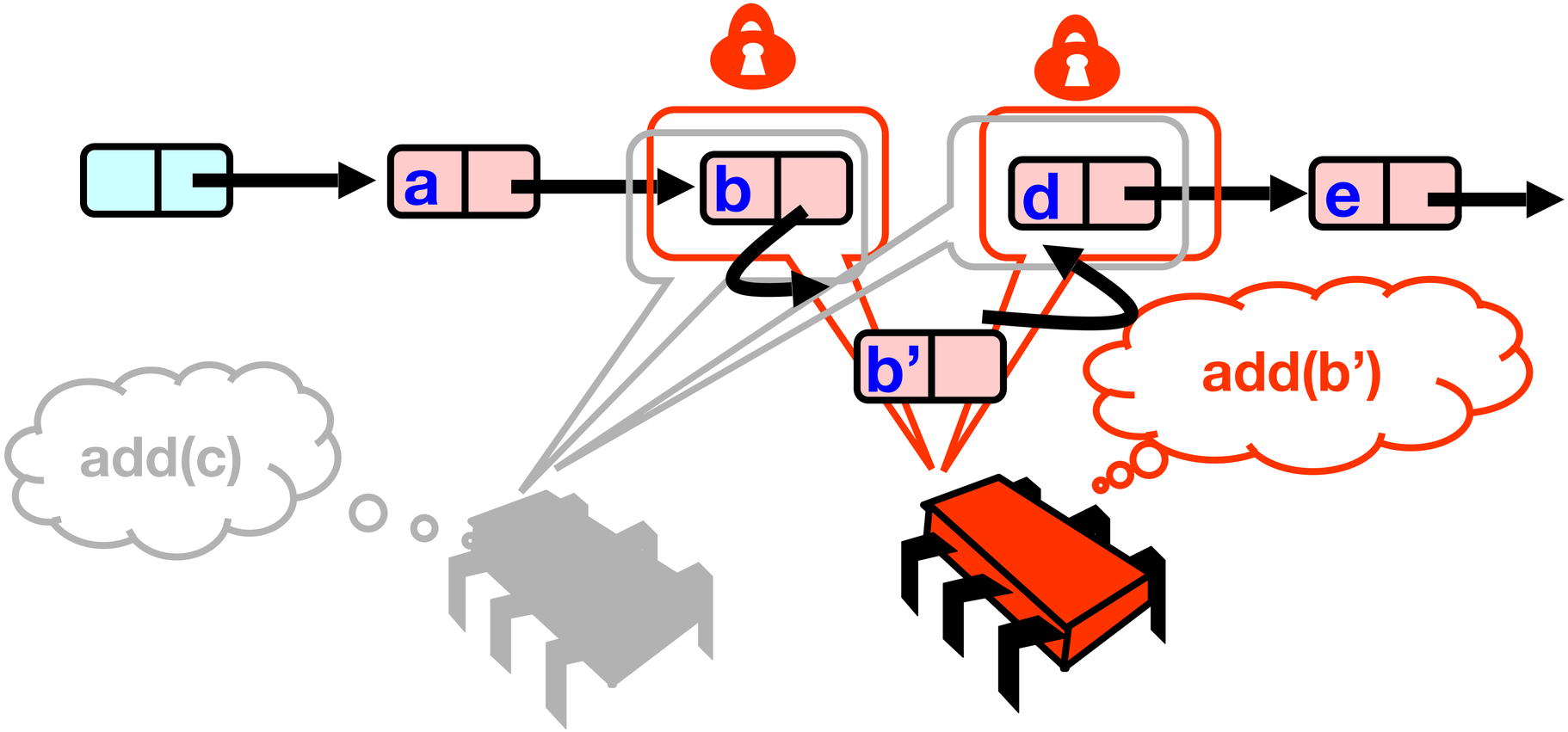
What else could go wrong?



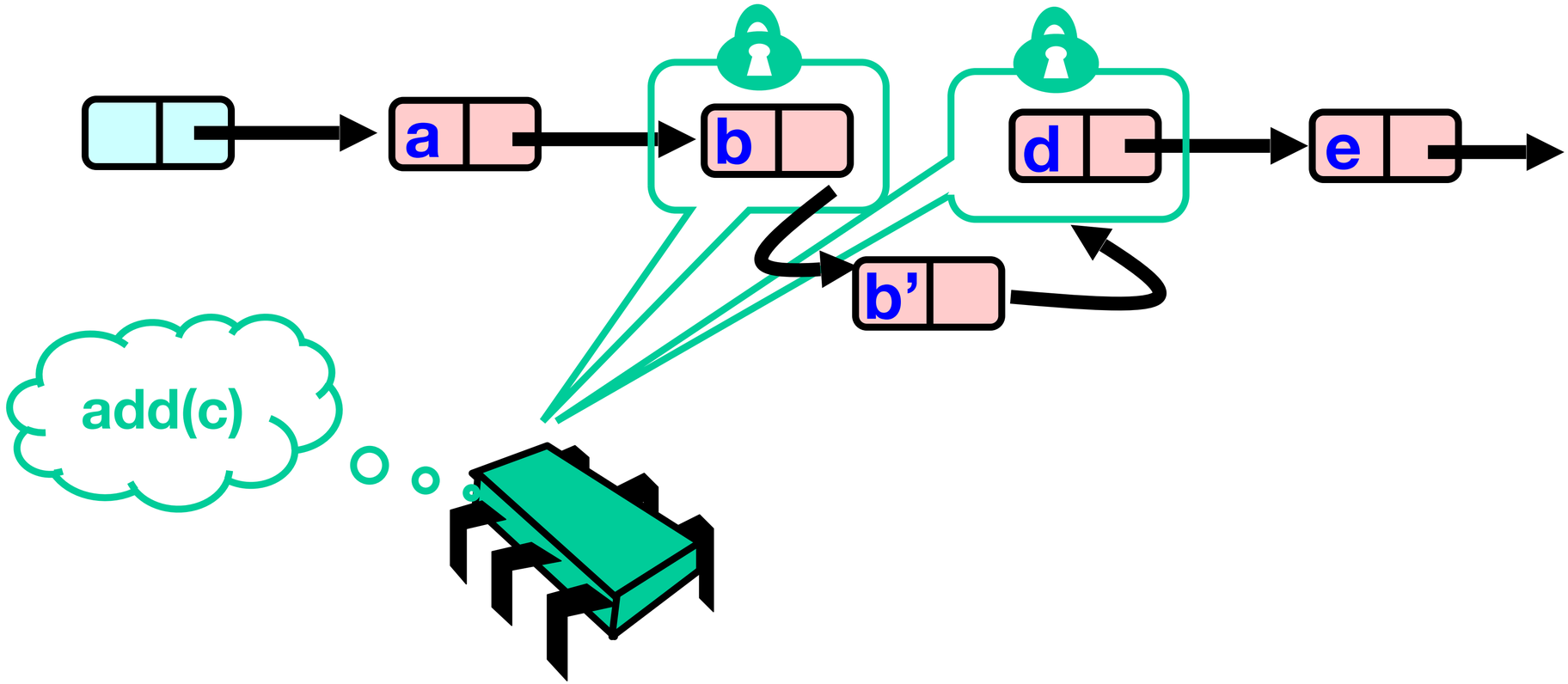
What else could go wrong?



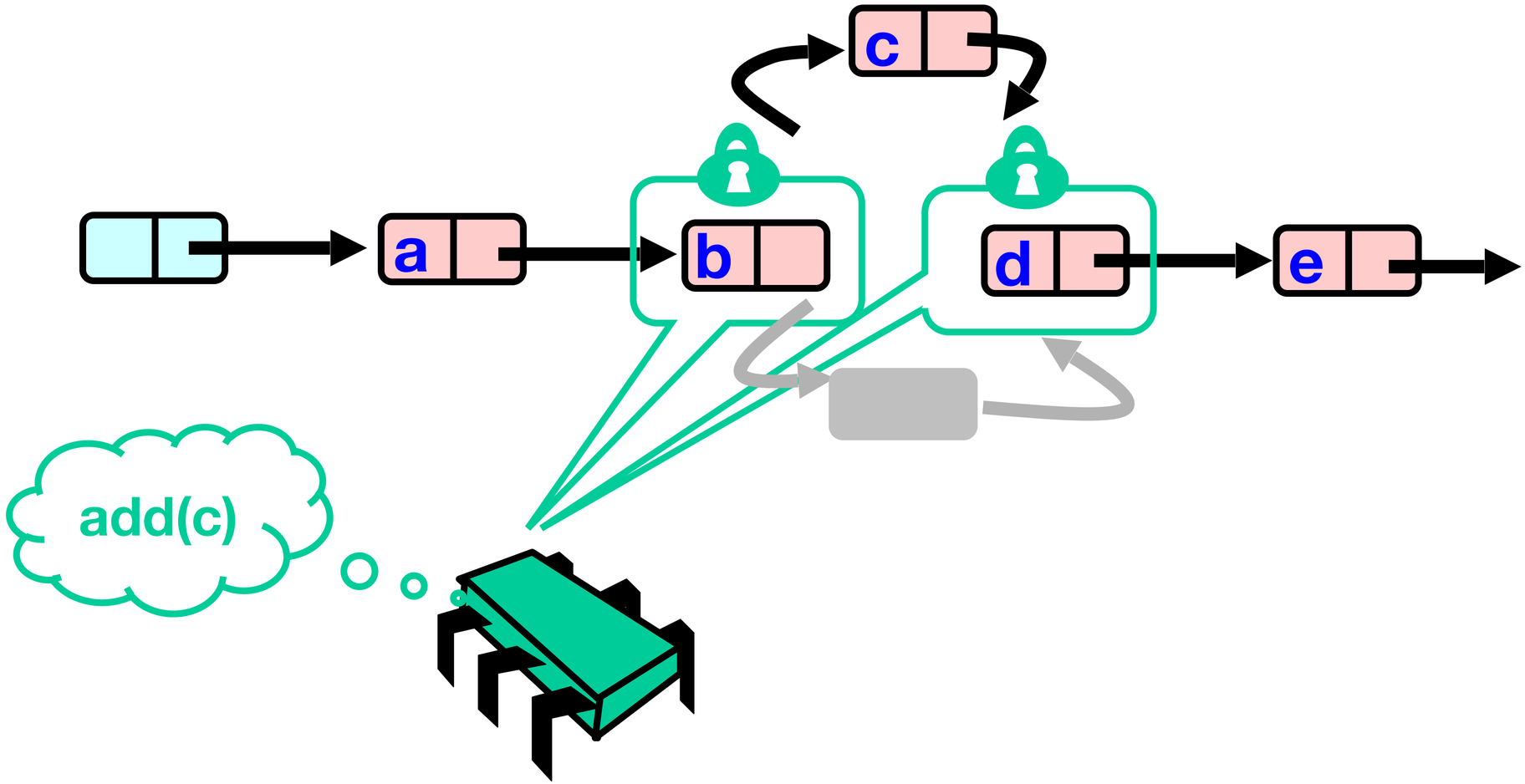
What else could go wrong?



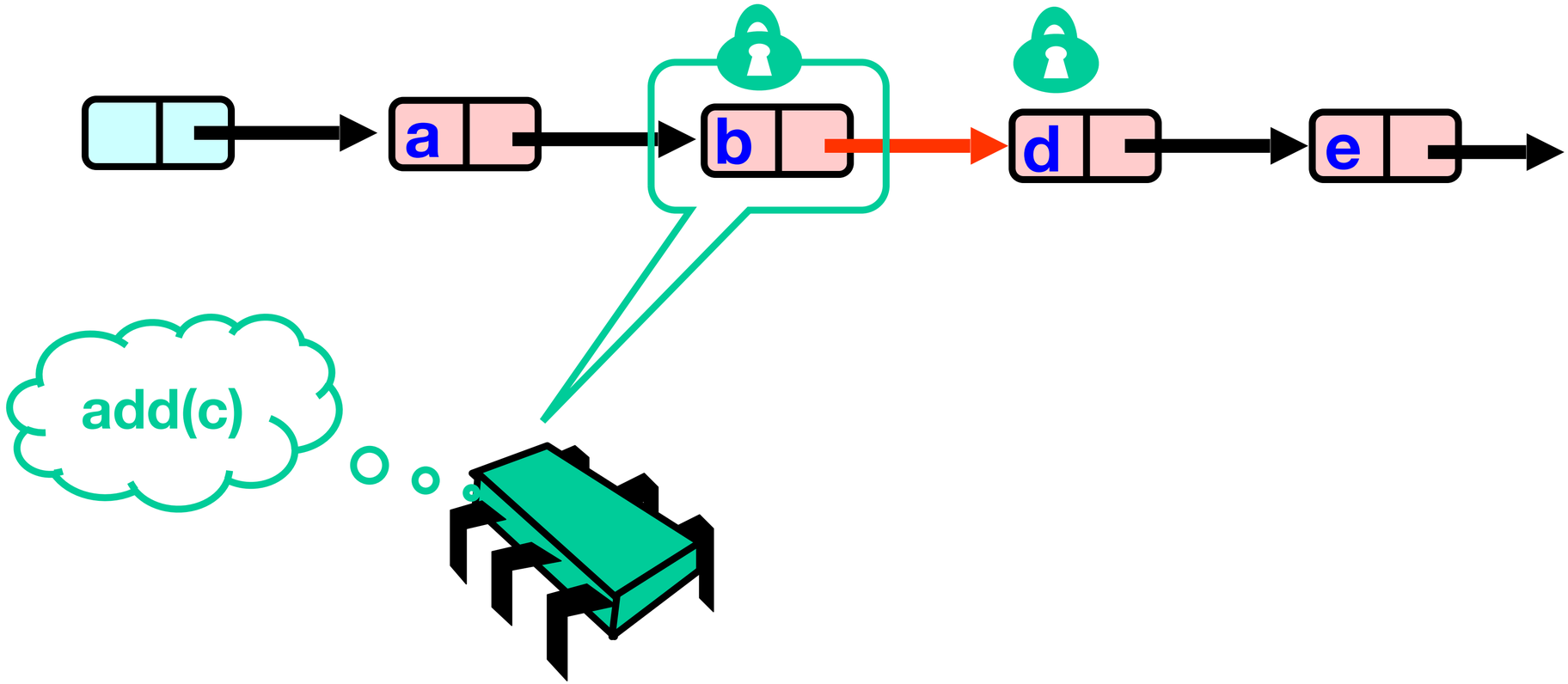
What else could go wrong?



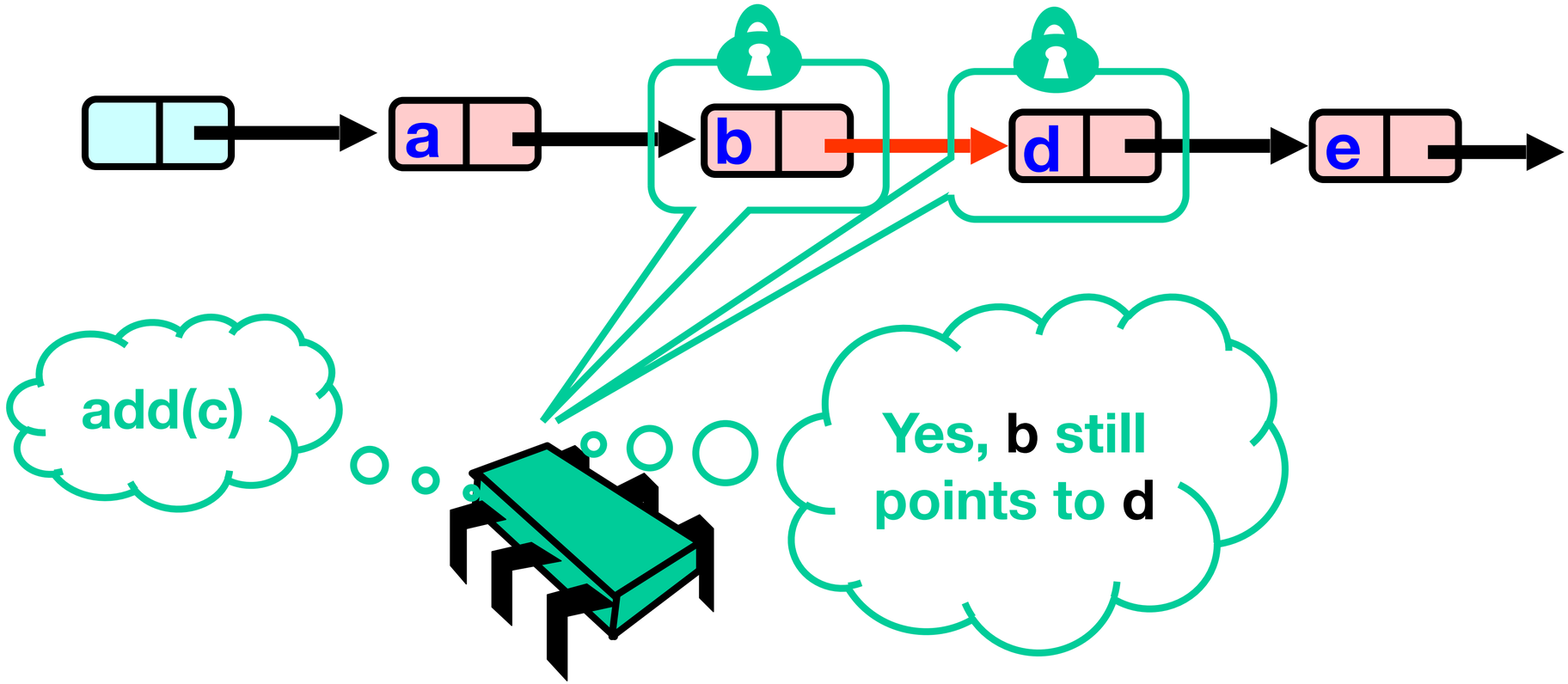
What else could go wrong?



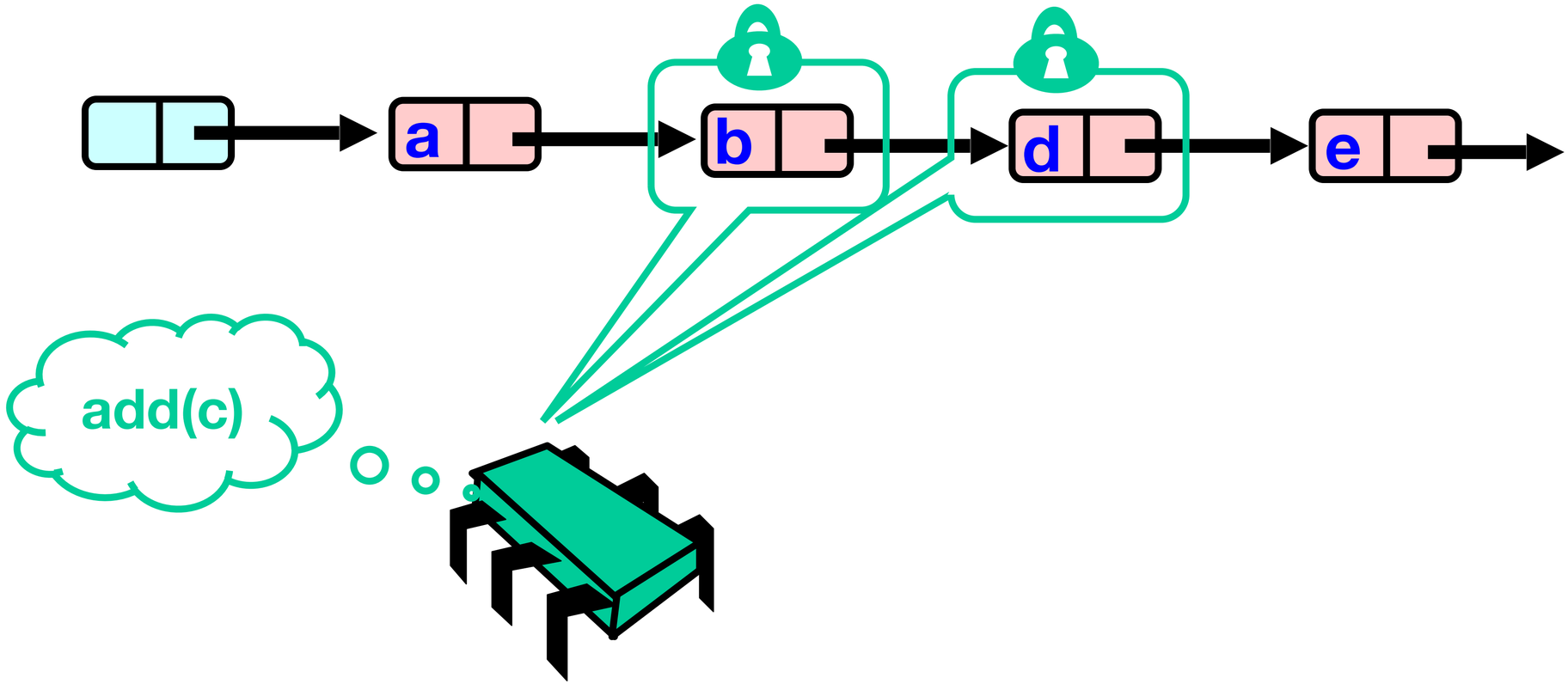
Validate — Part 2



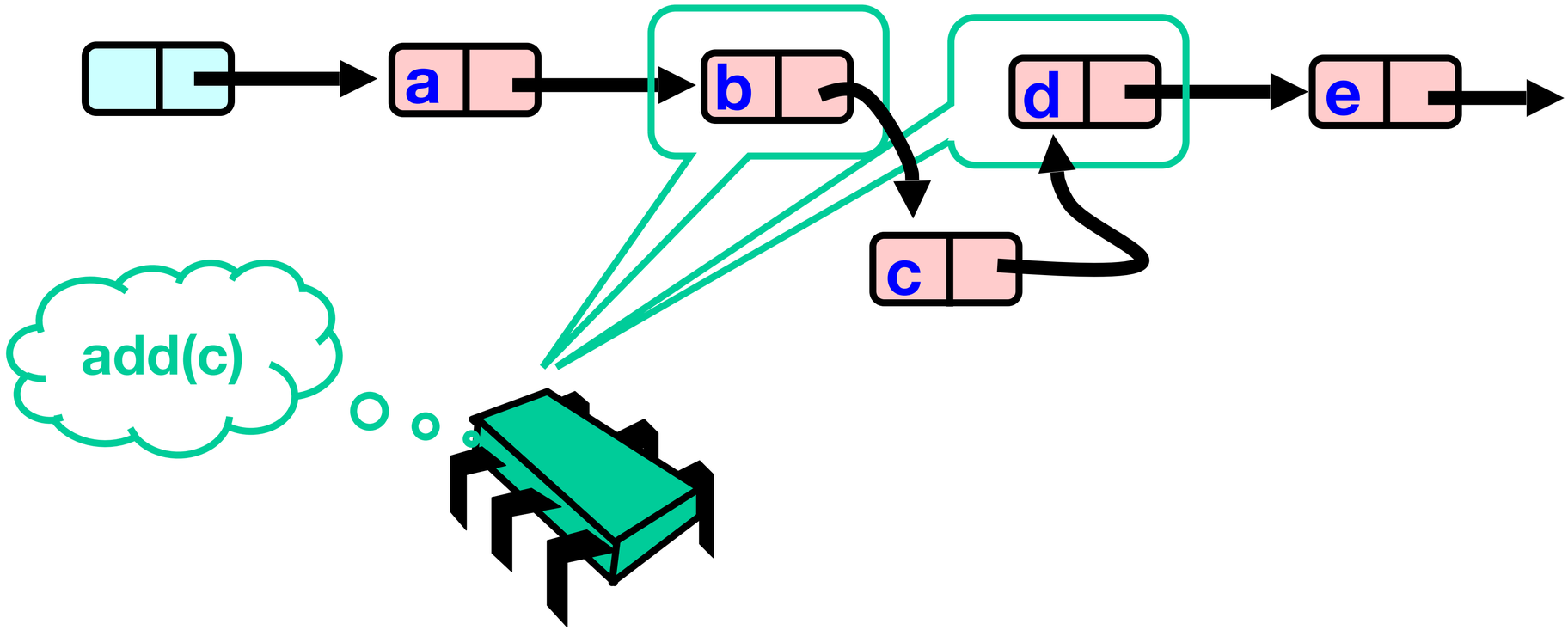
Validate — Part 2



Add after validation



Add after validation



Linearization points

- Same as fine-grained locks
- For successful add and remove
  - When `pred.next` pointer is updated
- For unsuccessful add and remove
  - When the lock on the node with `curr.key >= key` is obtained ***after successful validation***

Same Abstraction Map

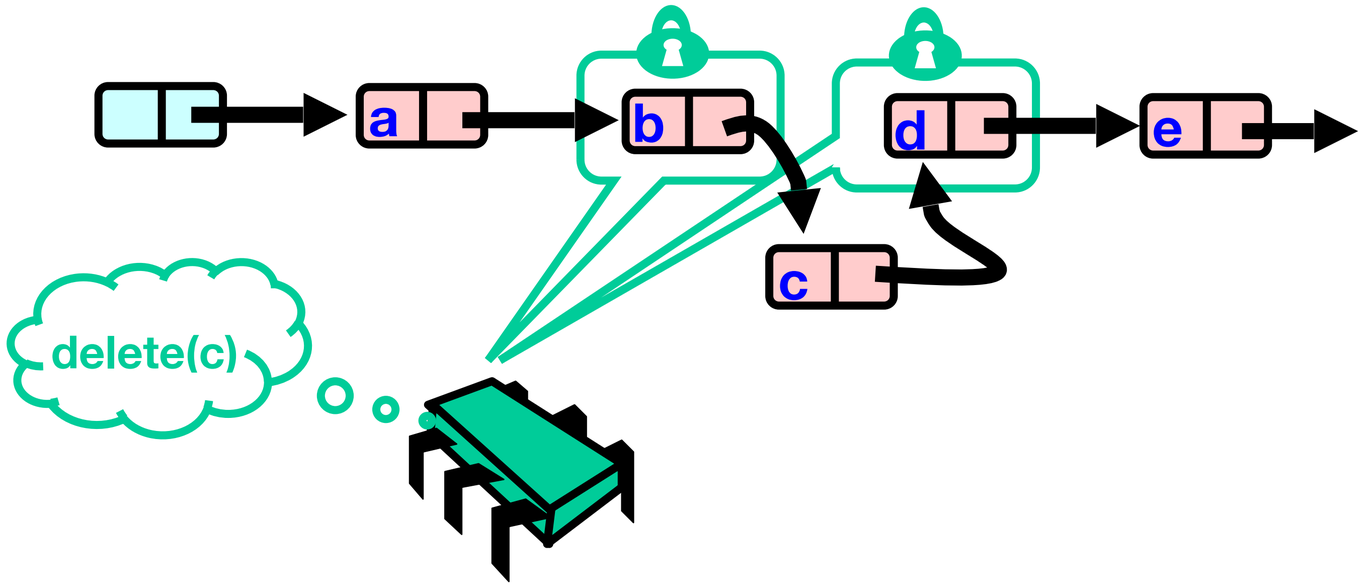
$$S(head) = \{x \mid \exists a \,.\, a \text{ is reachable from } head \wedge a \,.\, item = x\}$$

Invariants

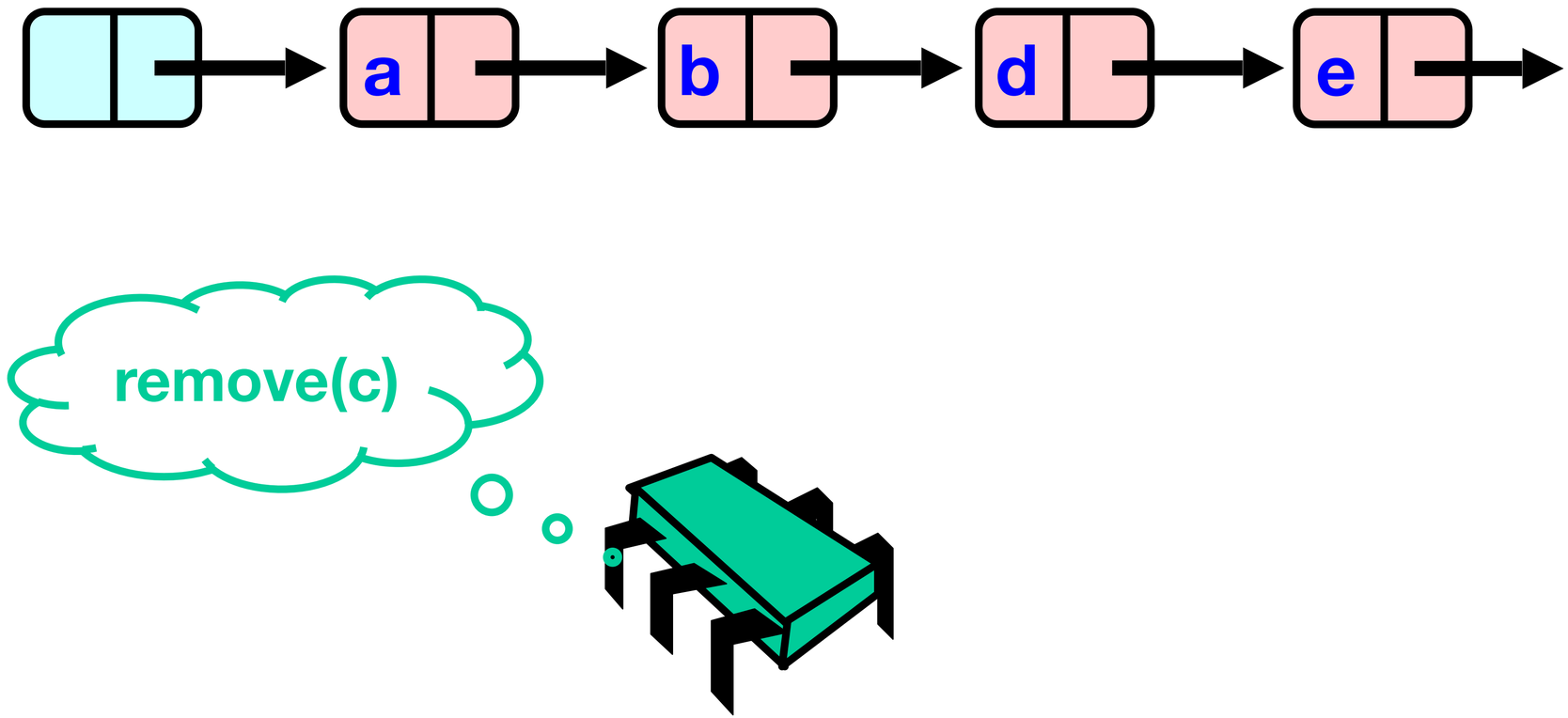
- **Careful:** we may traverse deleted nodes
- But we establish properties by
  - Validation
  - After we lock target nodes

Correctness of delete(c)

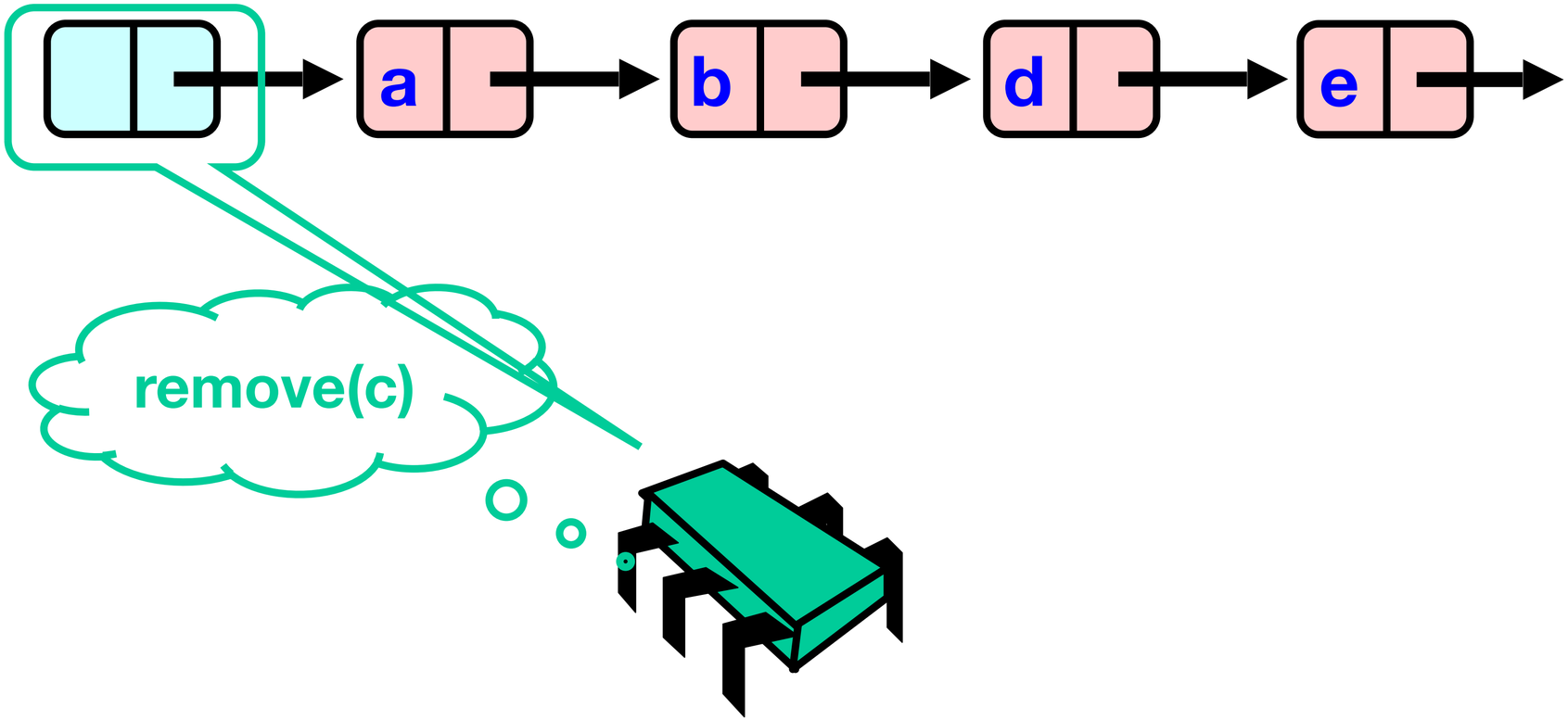
- If
  - Nodes b and c both locked
  - Node b still accessible
  - Node c still successor to b
- Then
  - Neither will be deleted
  - OK to delete and return `true`



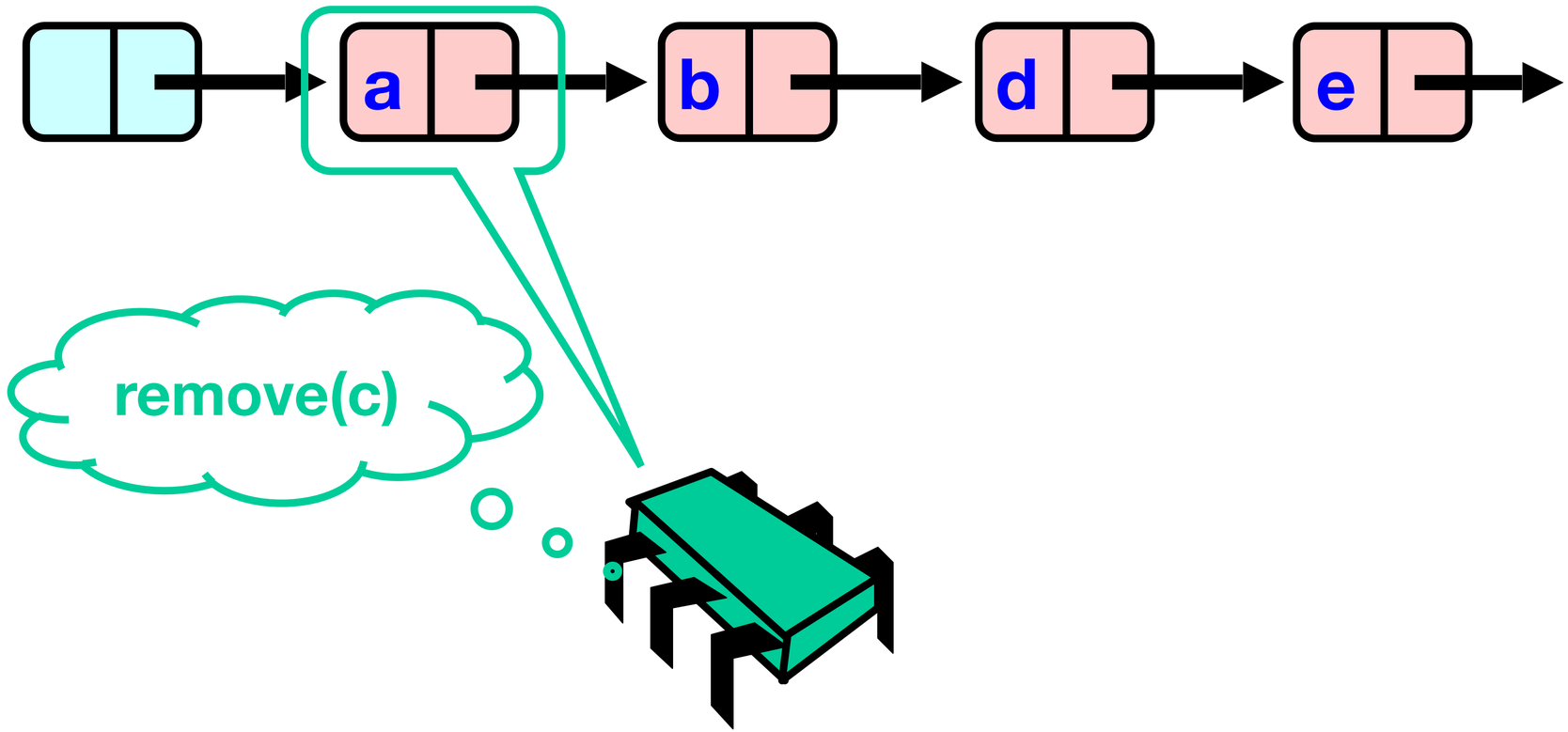
Unsuccessful Remove



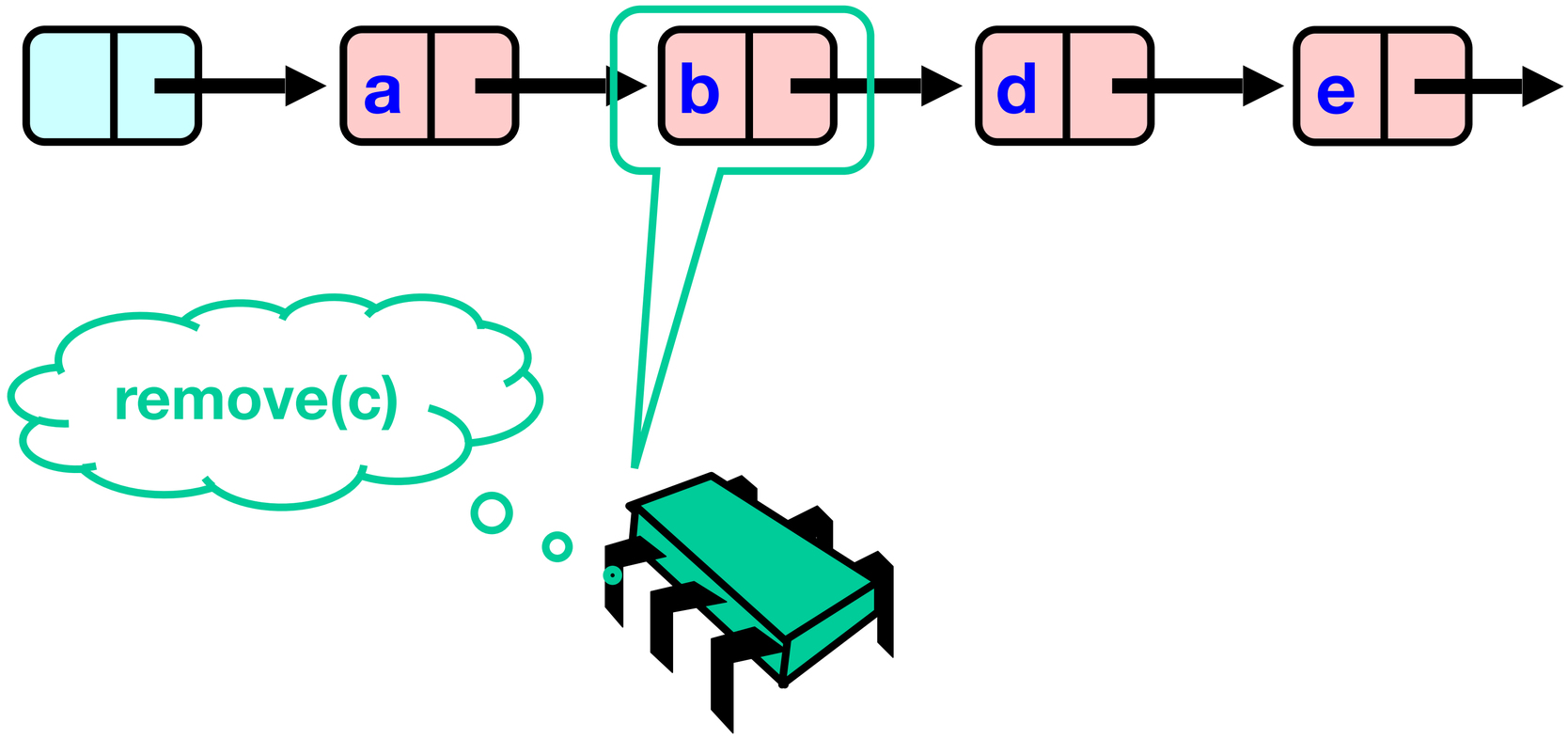
Unsuccessful Remove



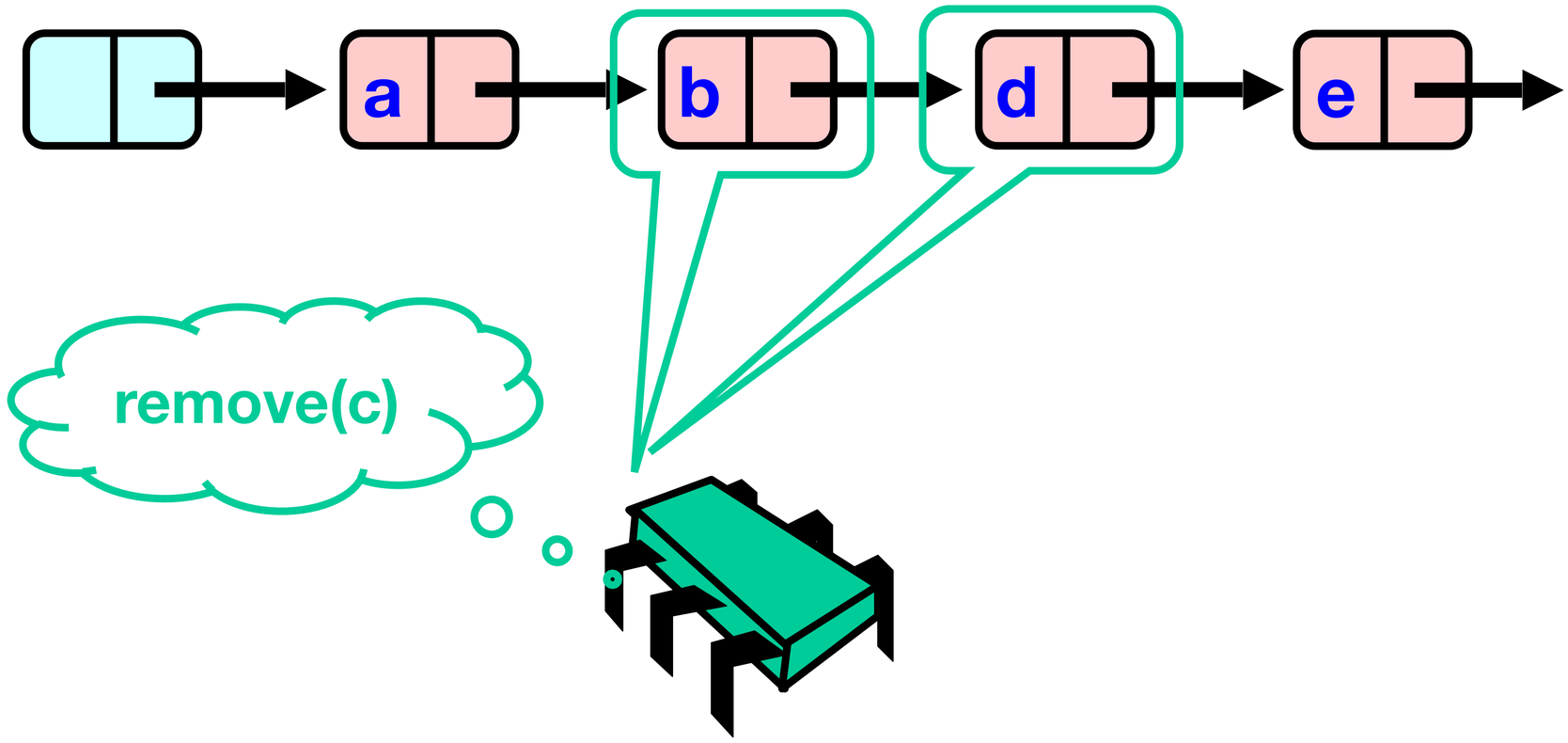
Unsuccessful Remove



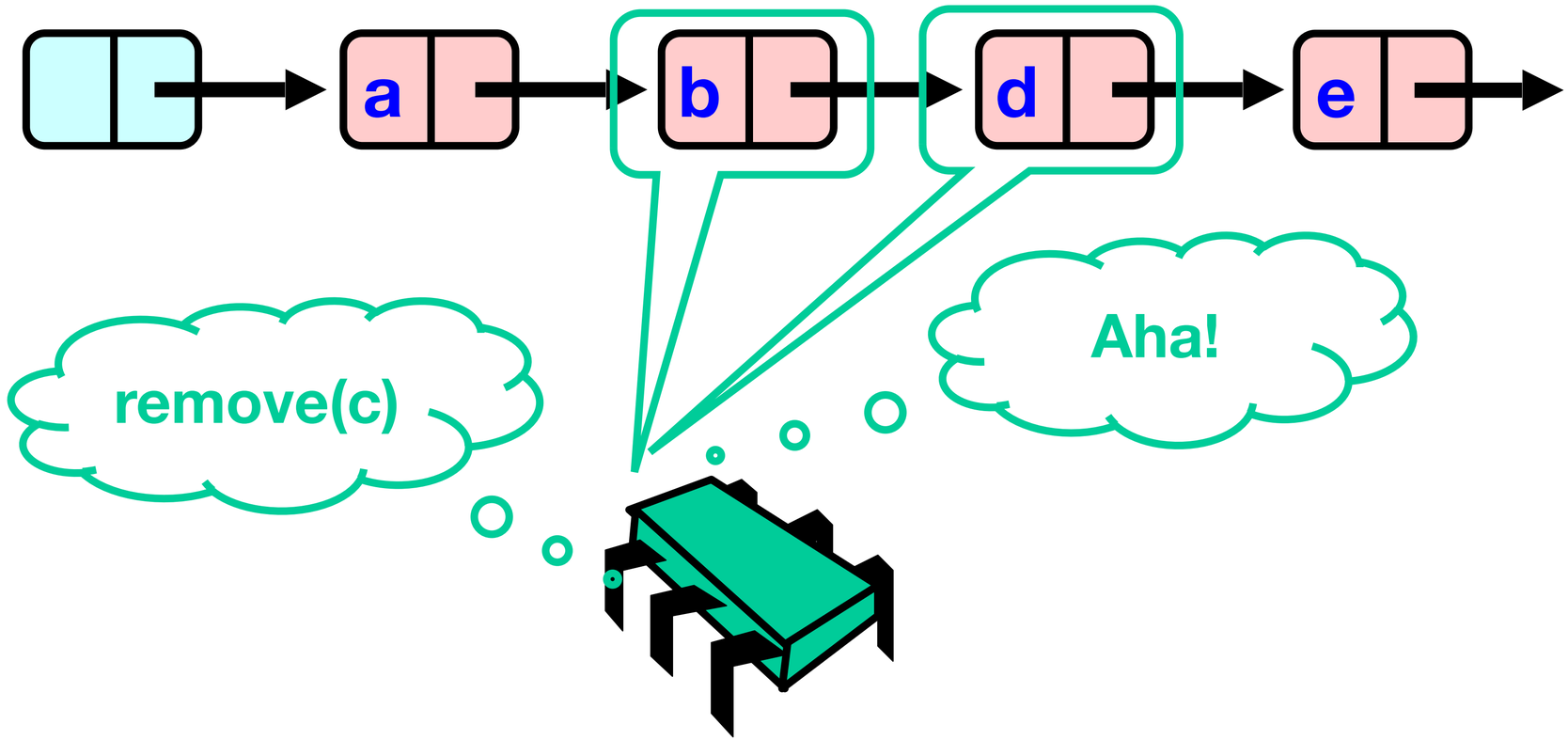
Unsuccessful Remove



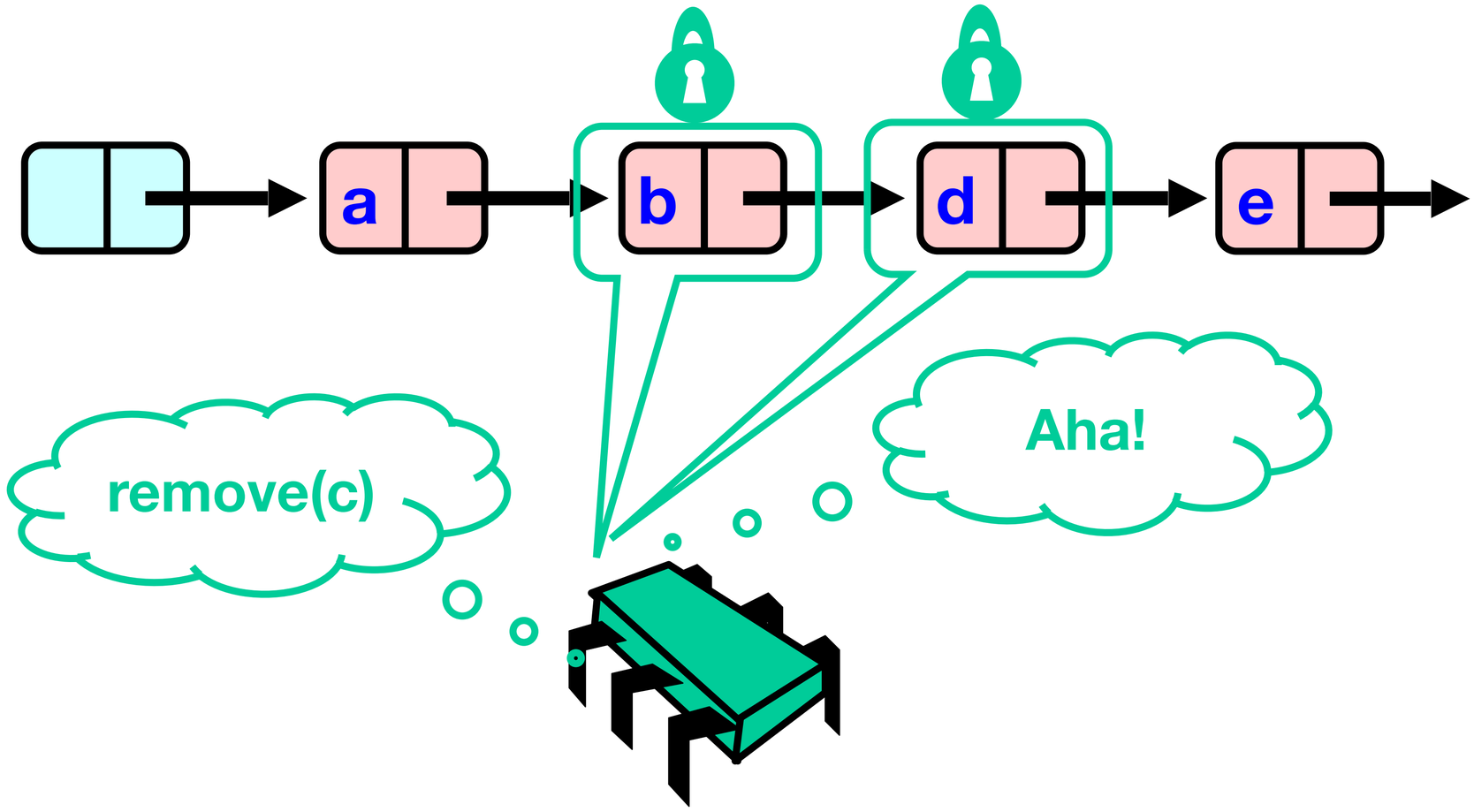
Unsuccessful Remove



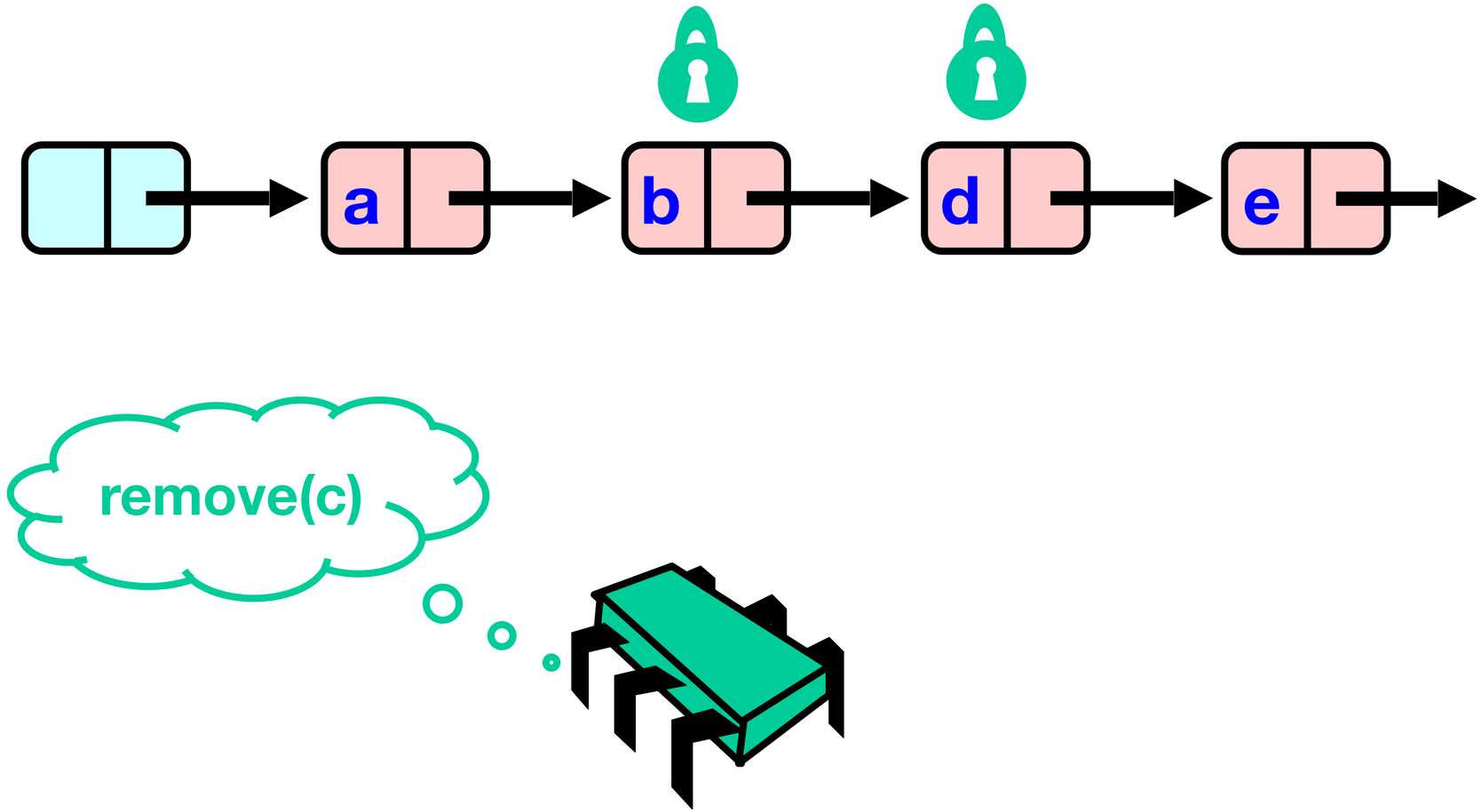
Unsuccessful Remove



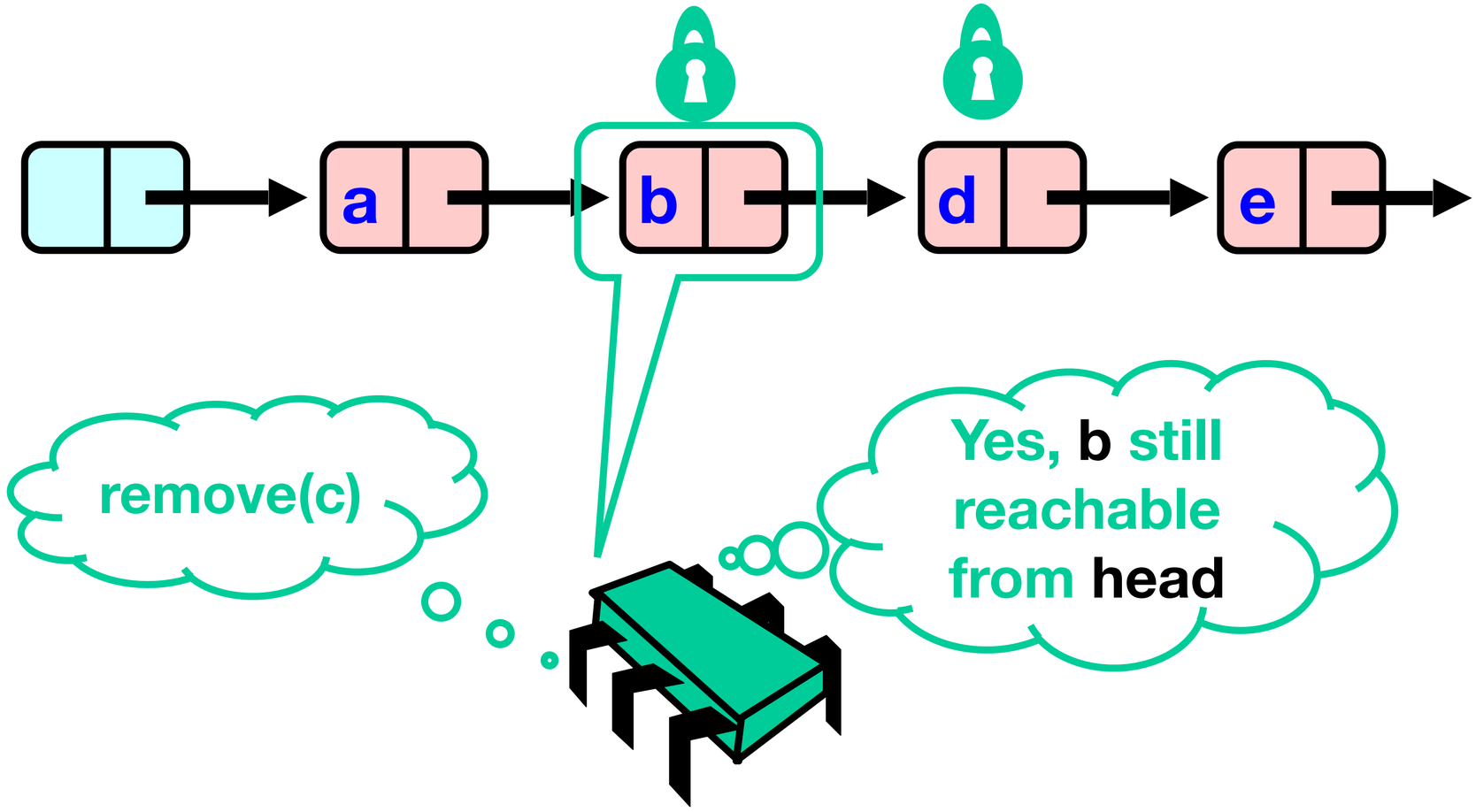
Unsuccessful Remove



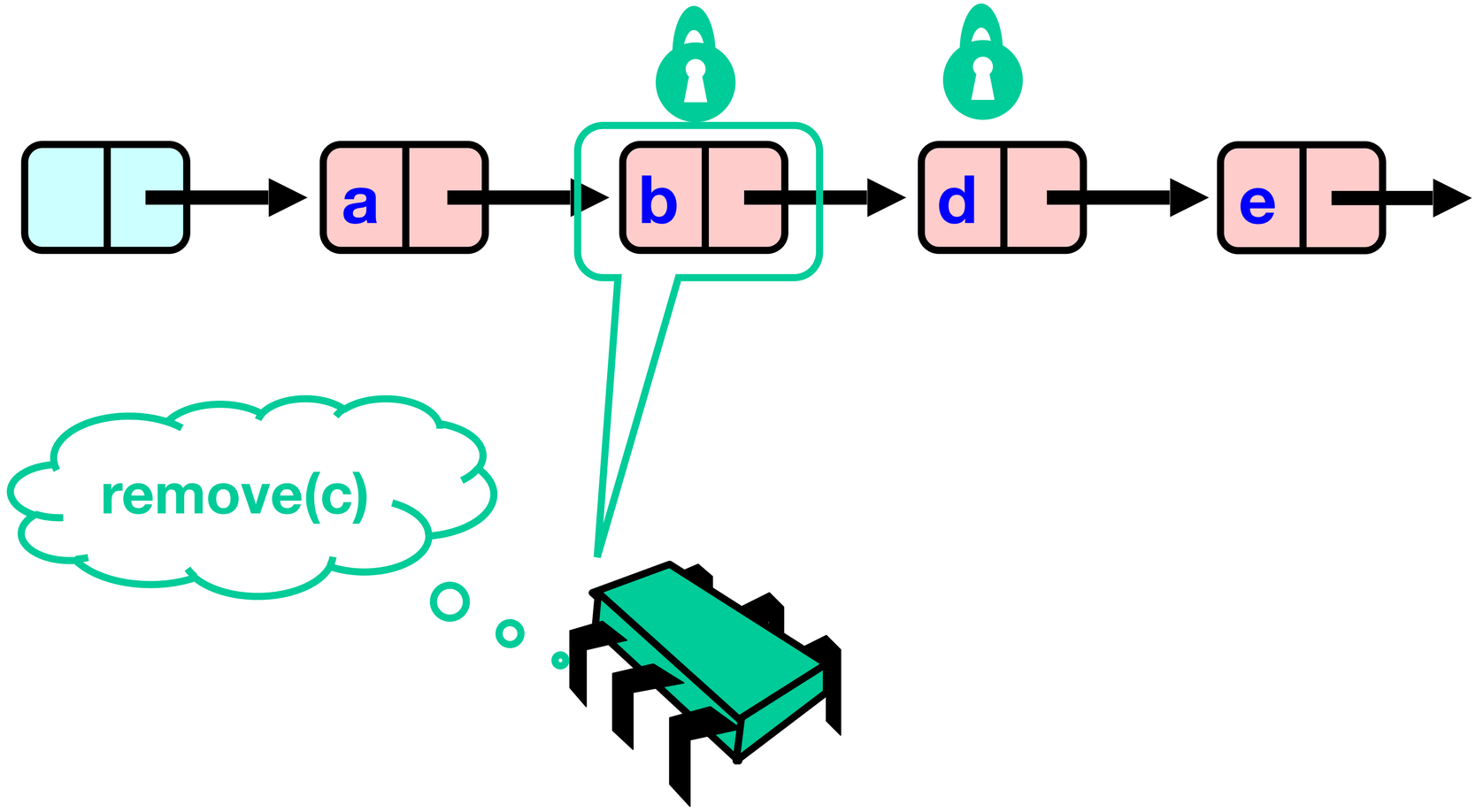
Validate (1)



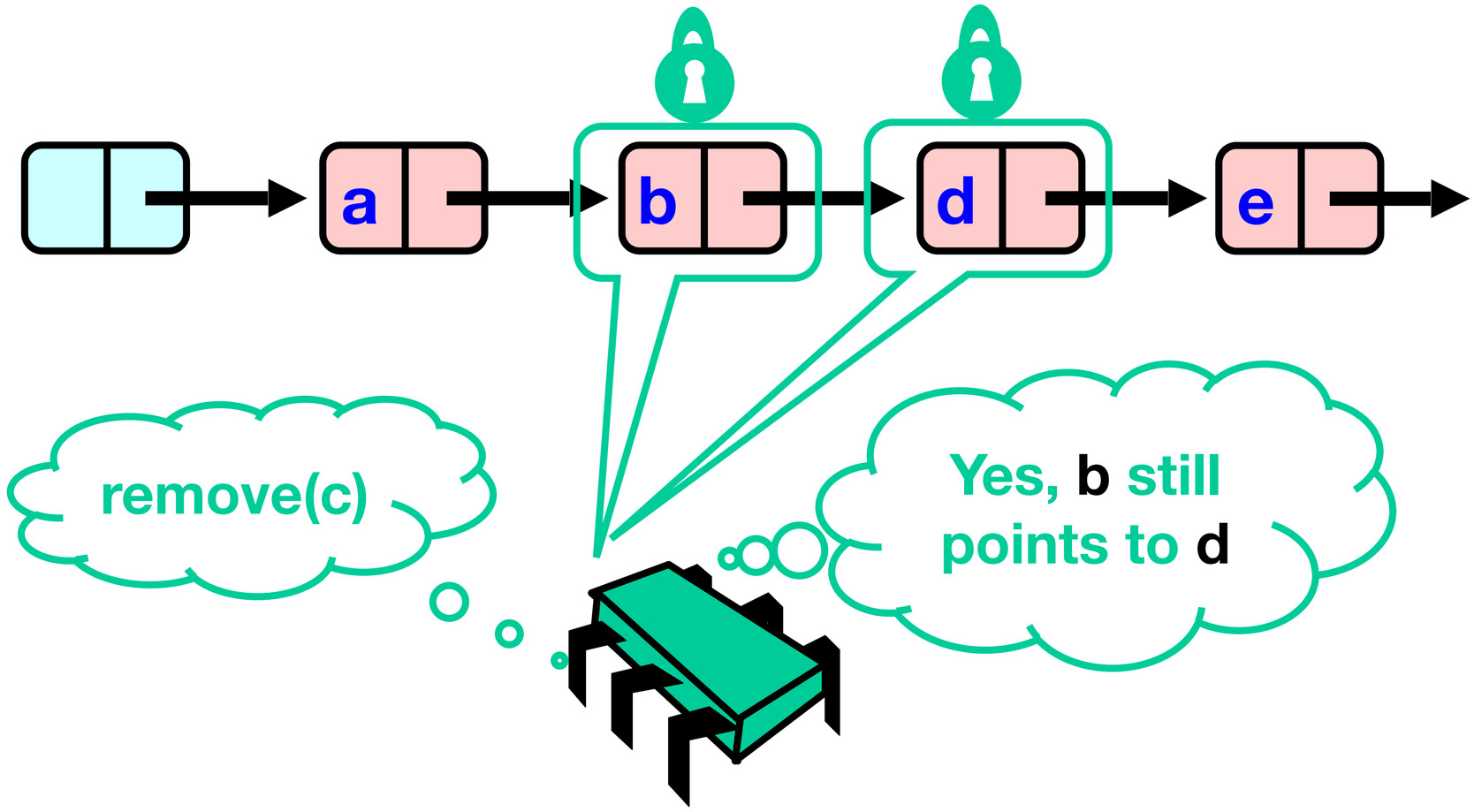
Validate (1)



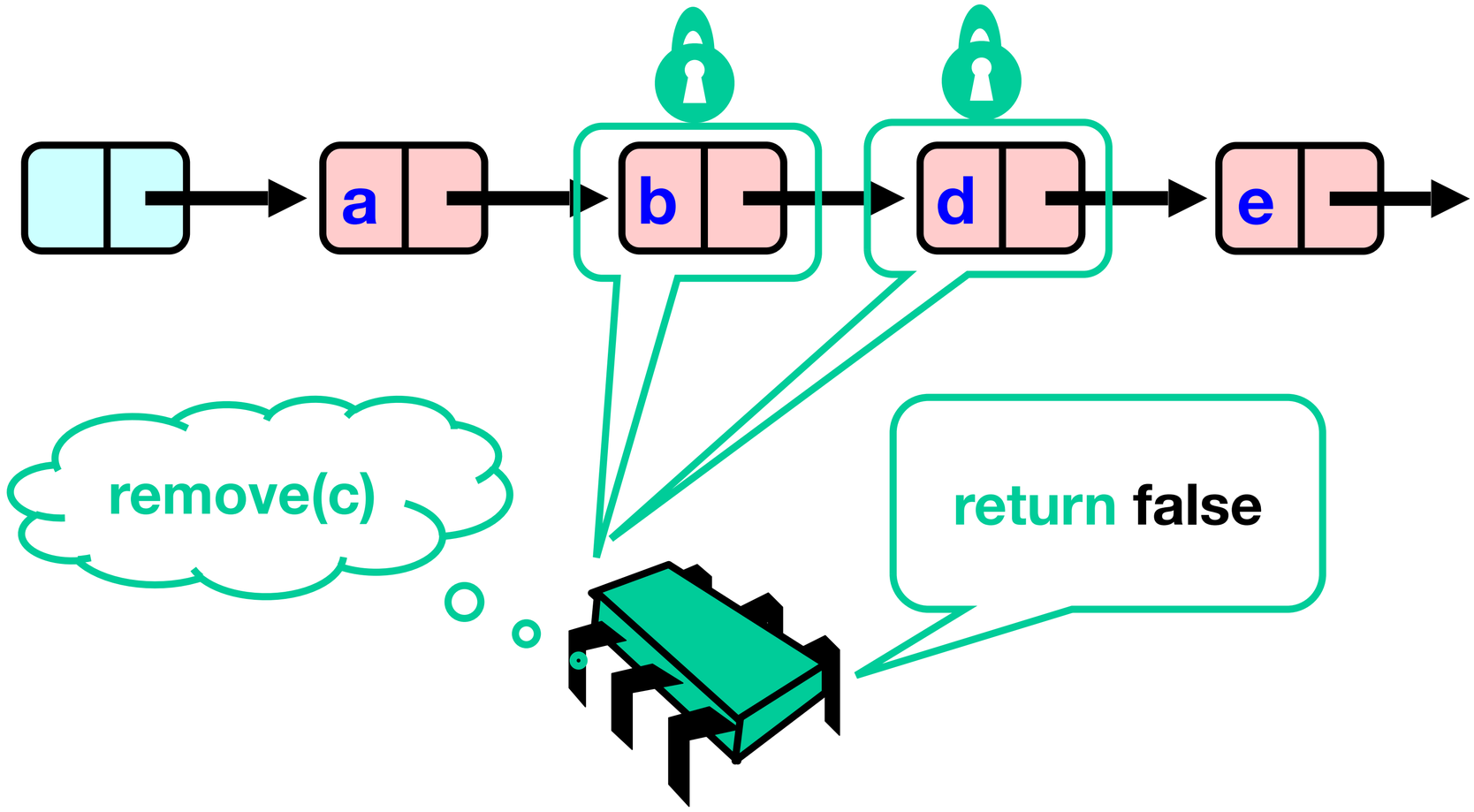
Validate (2)



Validate (2)



OK Computer



Correctness

- If
  - Nodes **b** and **d** both locked
  - Node **b** still accessible
  - Node **d** still successor to **b**
- Then
  - Neither will be deleted
  - No thread can add **c** after **b**
  - OK to return **false**

Locate and validate

- The list representation remains the same (a lock per node)

```
(** Locate position for key without
    locking (optimistic traversal). *)

let locate head key =
  let rec loop pred curr =
    if curr.key < key then
      loop curr curr.next
    else
      (pred, curr)
  in
  loop head head.next
```

```
(** Validate that pred and curr are still adjacent and reachable
    from head. Pre-condition: pred.lock and curr.lock are held.
    Returns true if pred in list /\ pred.next = curr *)
let validate head pred curr =
  let rec loop node =
    if node.key <= pred.key then
      if node == pred then
        pred.next == curr
      else
        loop node.next
    else
      false
  in
  loop head
```

After Locate and validate

- If item is present
  - **curr** holds item
  - **pred** just before **curr**
- If item is absent
  - **curr** has first higher key
  - **pred** just before **curr**
- Assuming no synchronization problems

Remove function

```
let remove list item =
  let key = Hashtbl.hash item in
  let rec attempt () =
    let (pred, curr) = locate list.head key in
    Mutex.lock pred.lock; Mutex.lock curr.lock;
    if validate list.head pred curr then begin
      let result =
        if curr.key = key then begin
          (* element found, remove it *)
          pred.next <- curr.next;
          true
        end else
          false  (* element not present *)
      in
      Mutex.unlock curr.lock; Mutex.unlock pred.lock;
      result
    end else begin
      Mutex.unlock curr.lock; Mutex.unlock pred.lock;
      attempt ()  (* validation failed, retry *)
    end
  in
  attempt ()
```

Optimistic List

- Limited hot-spots
  - Targets of `add()`,`remove()`,`contains()`
  - No contention on traversals
- Moreover
  - Traversals are wait-free
  - Food for thought …

So Far, So Good

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

Evaluation

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

(3) Lazy List

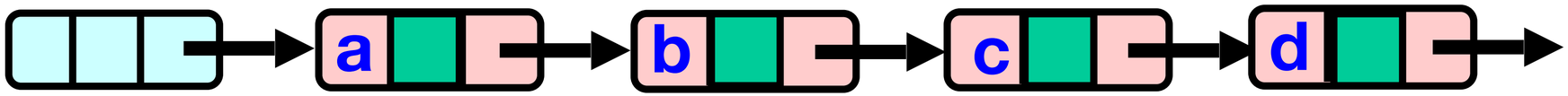
Lazy List

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

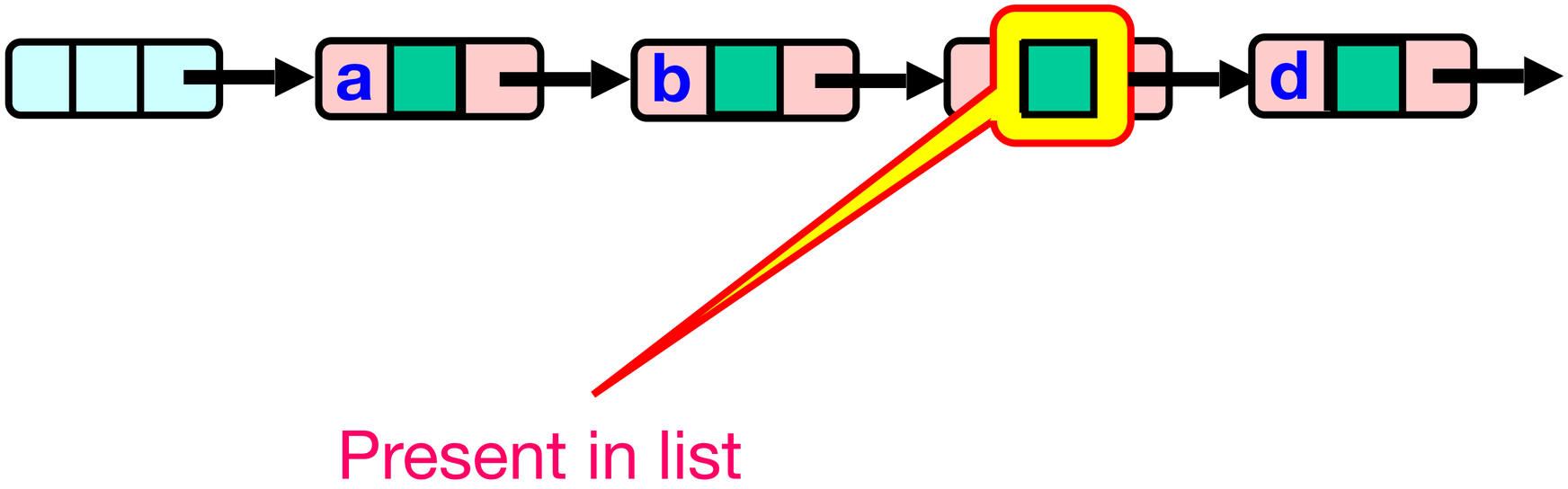
Lazy List

- `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)

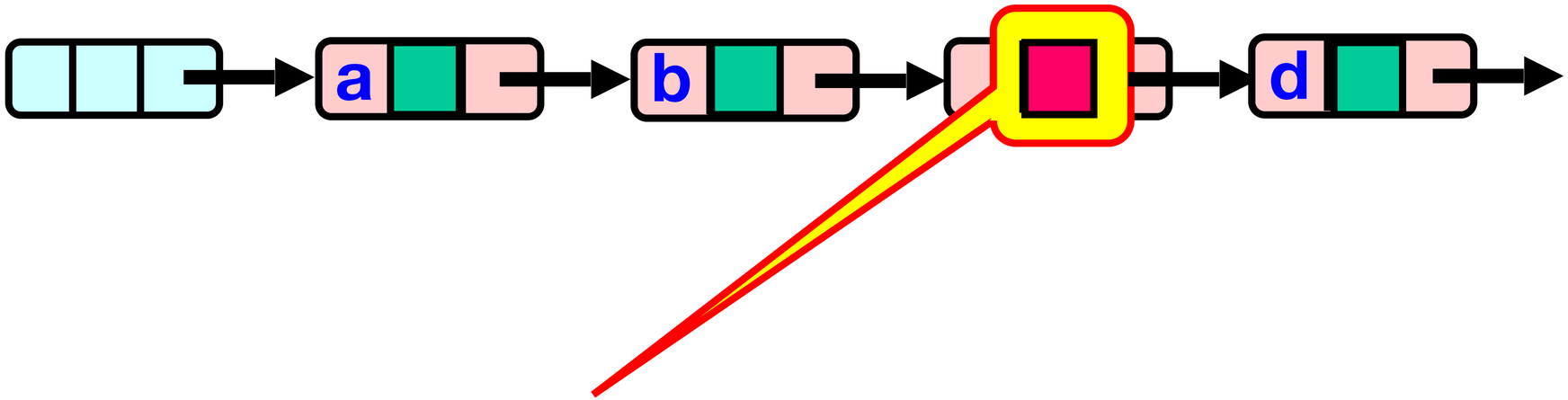
Lazy Removal



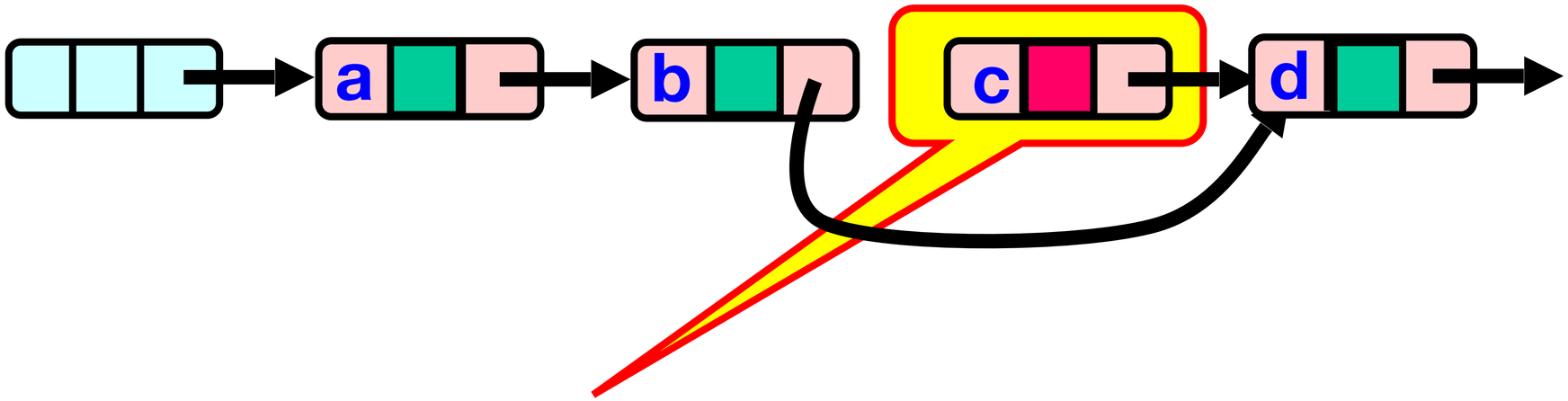
Lazy Removal



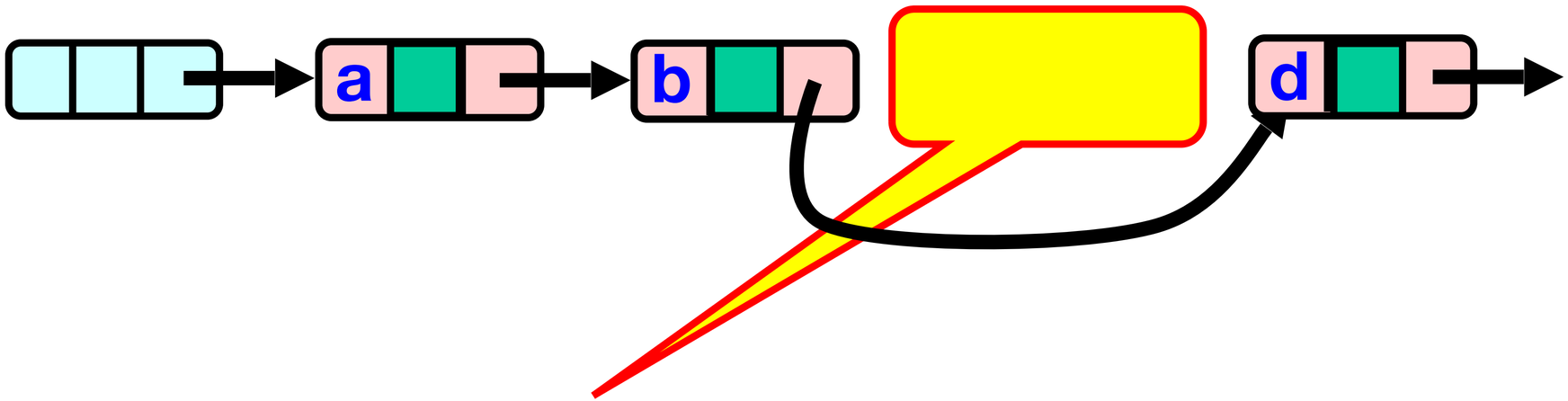
Lazy Removal



Lazy Removal



Lazy Removal



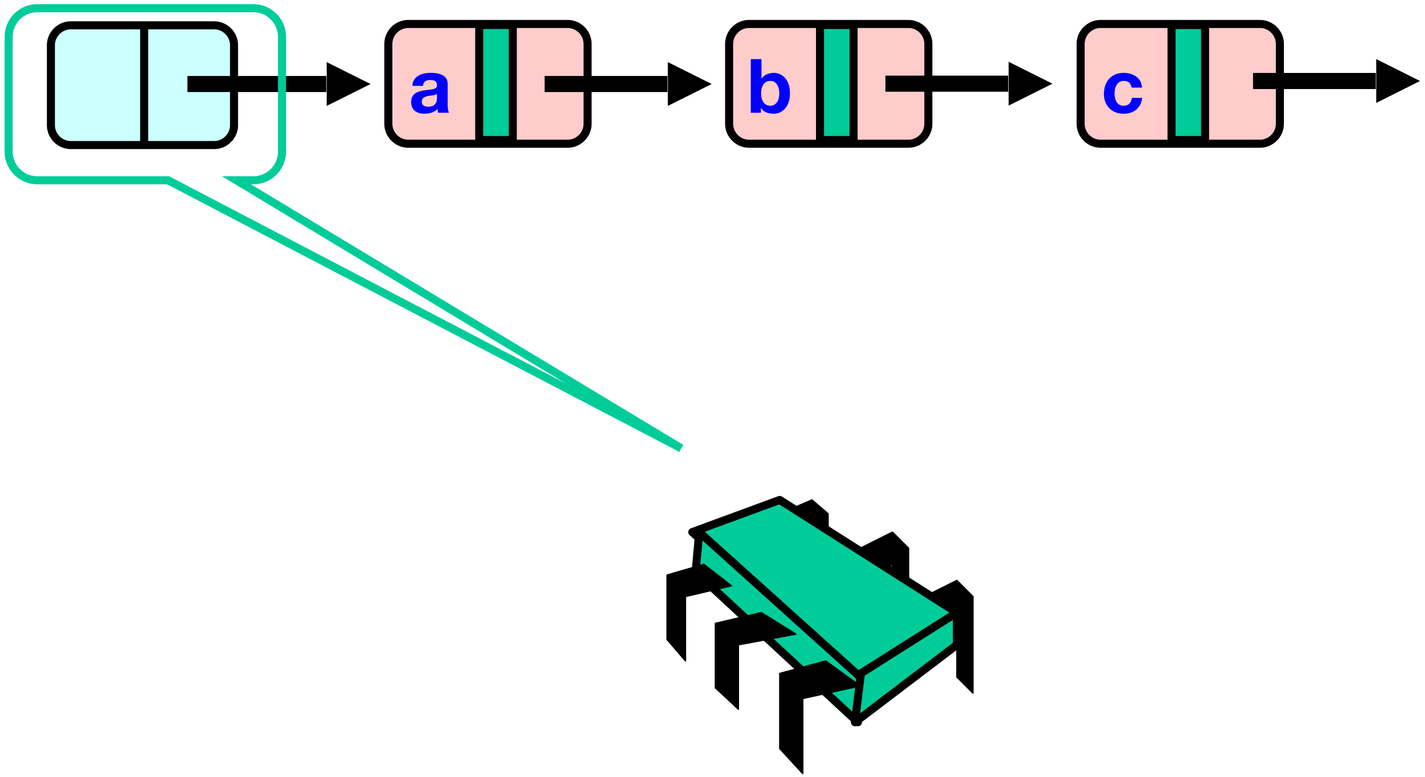
Lazy List

- 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.

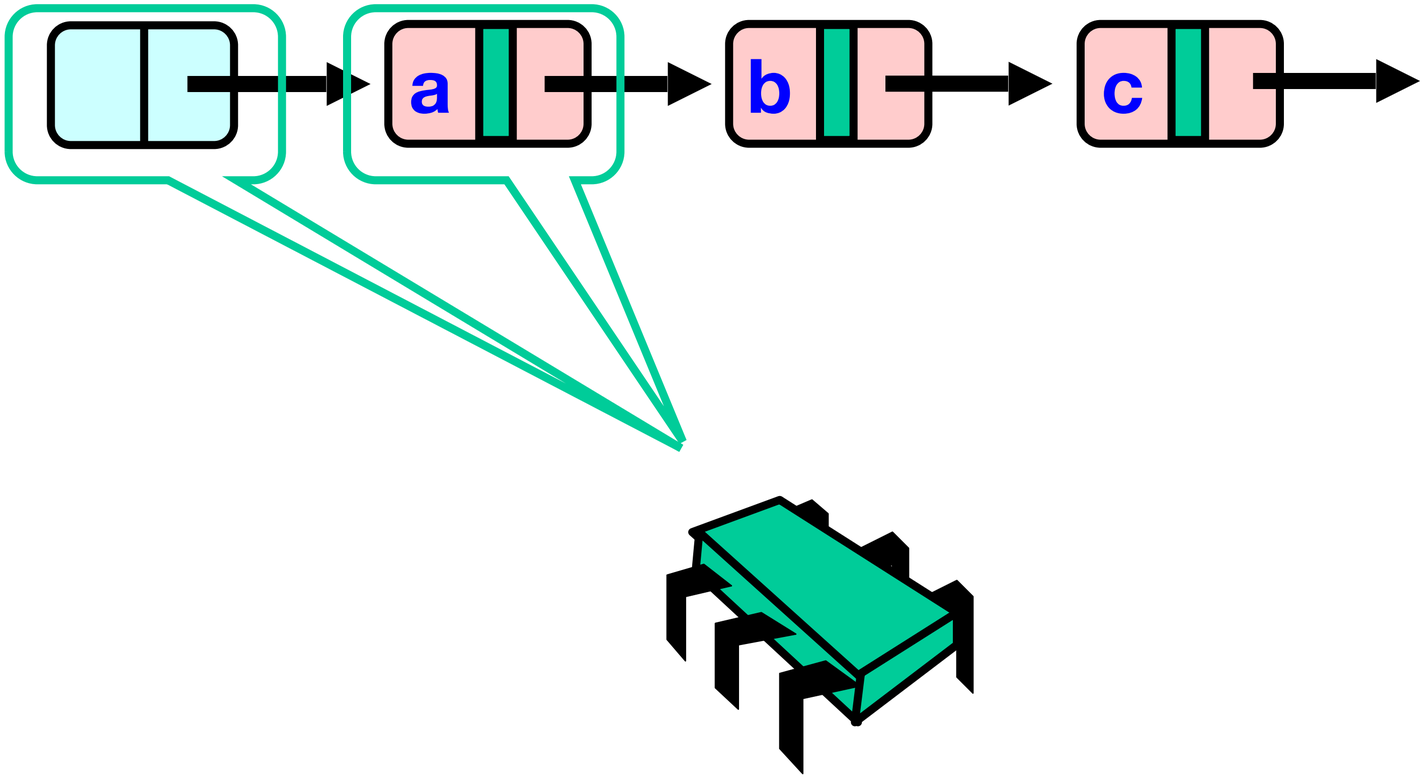
Validation

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

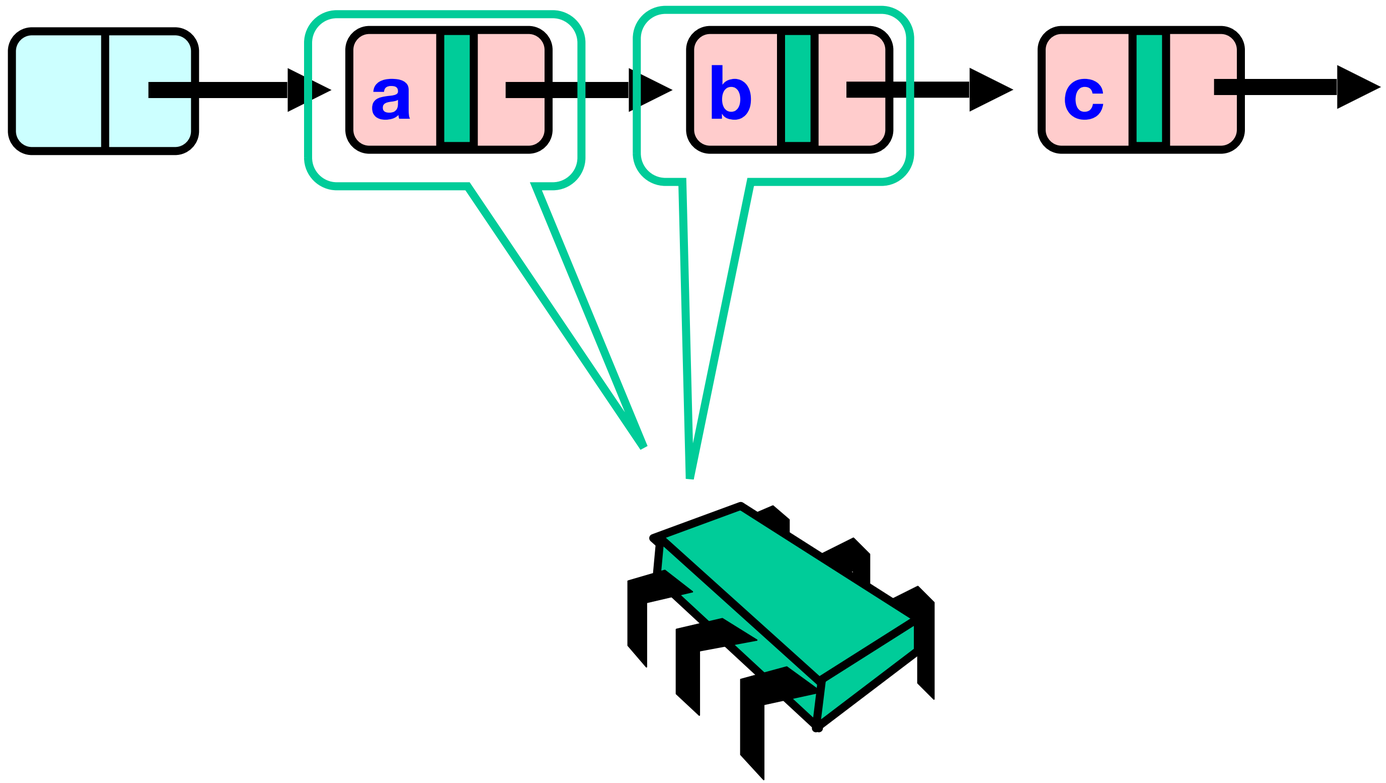
Business as Usual



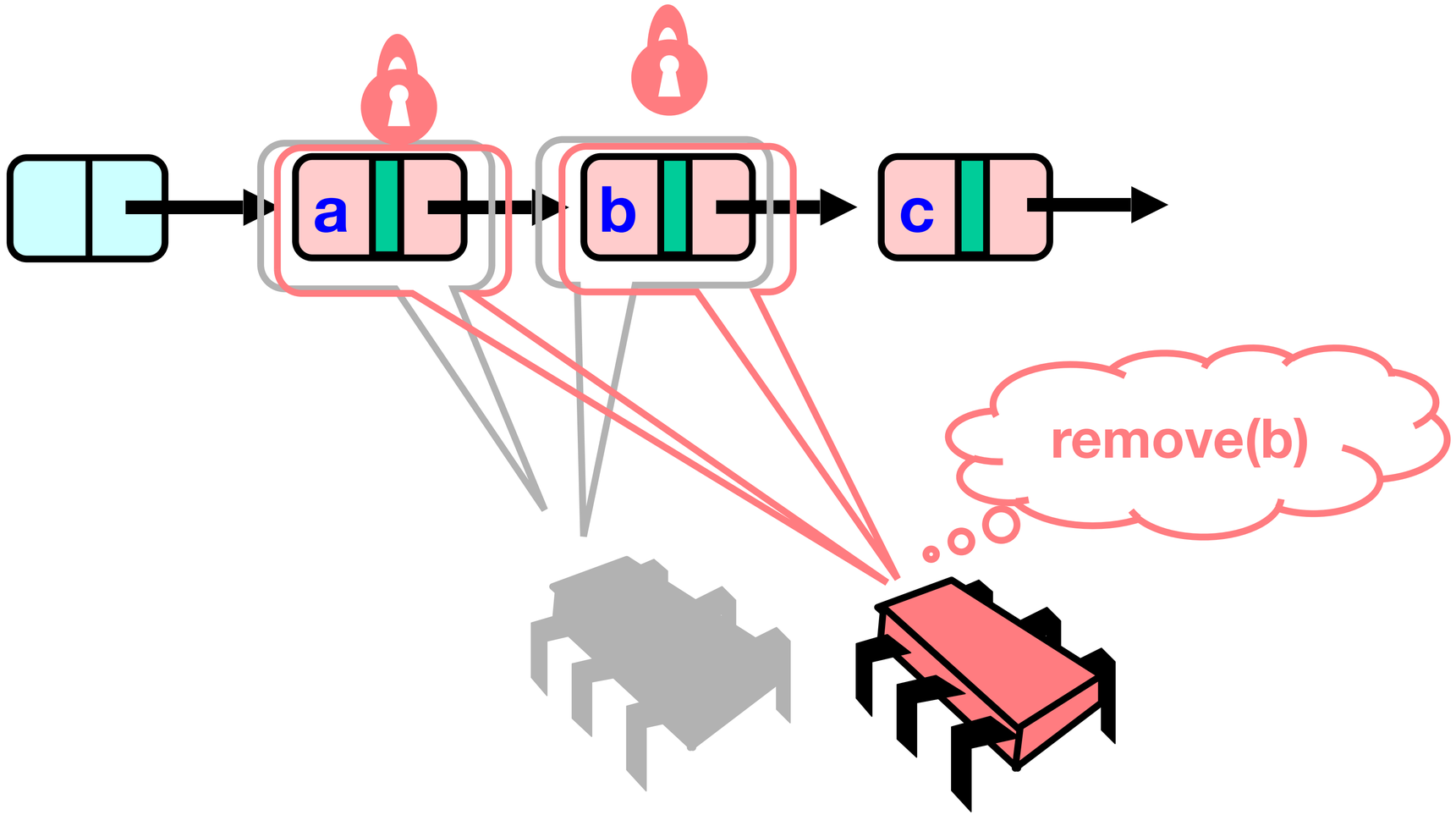
Business as Usual



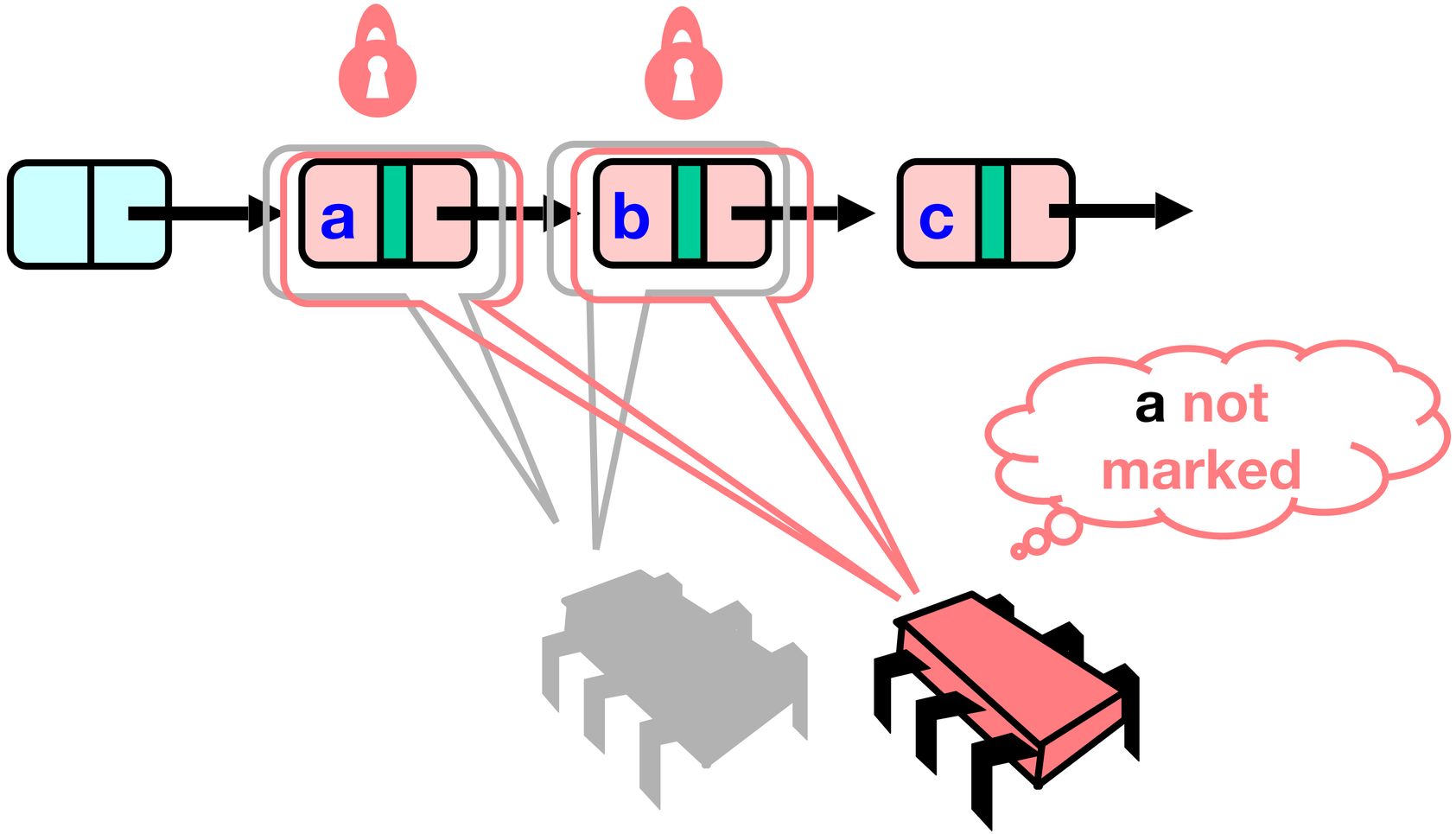
Business as Usual



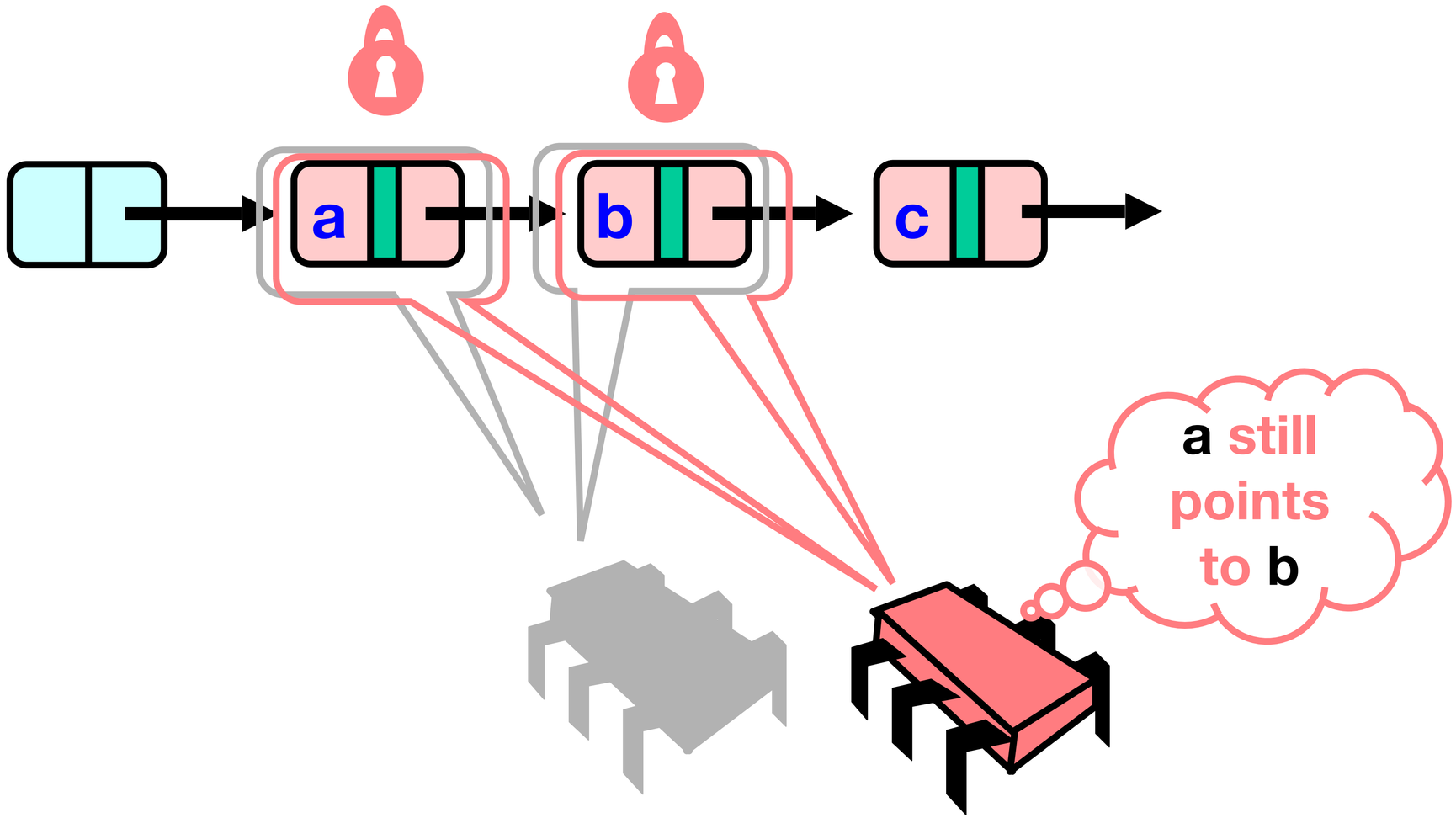
Business as Usual



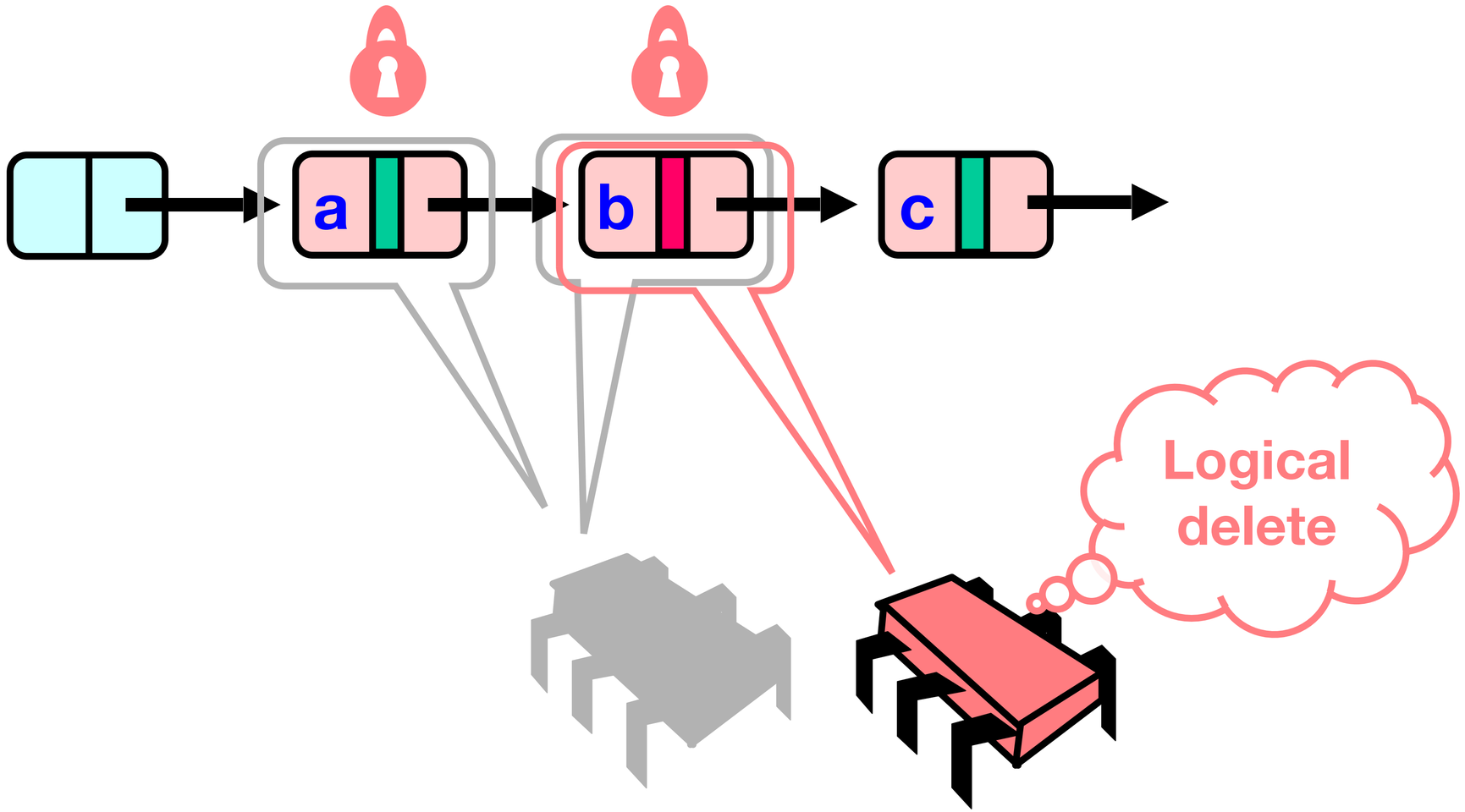
Business as Usual



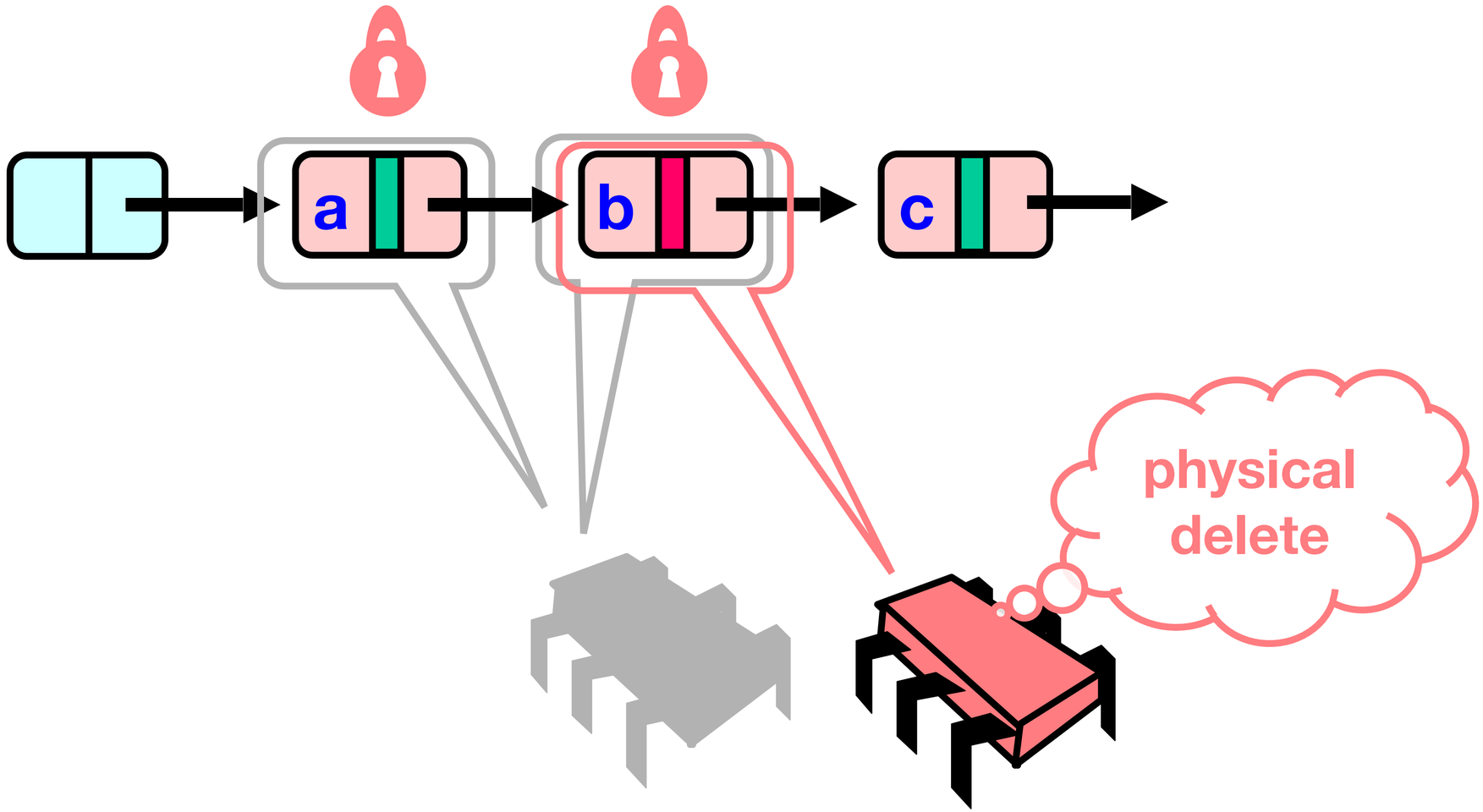
Business as Usual



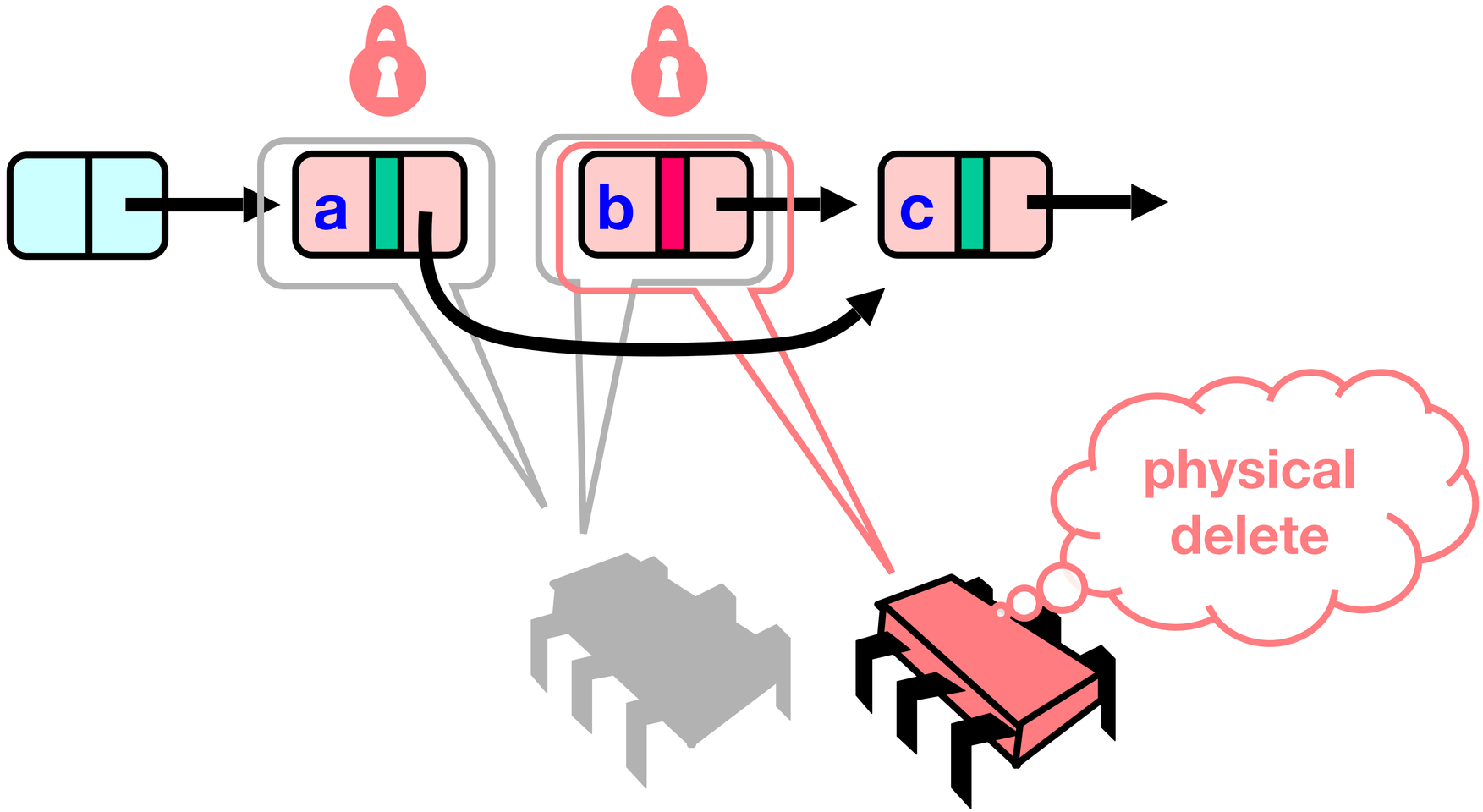
Business as Usual



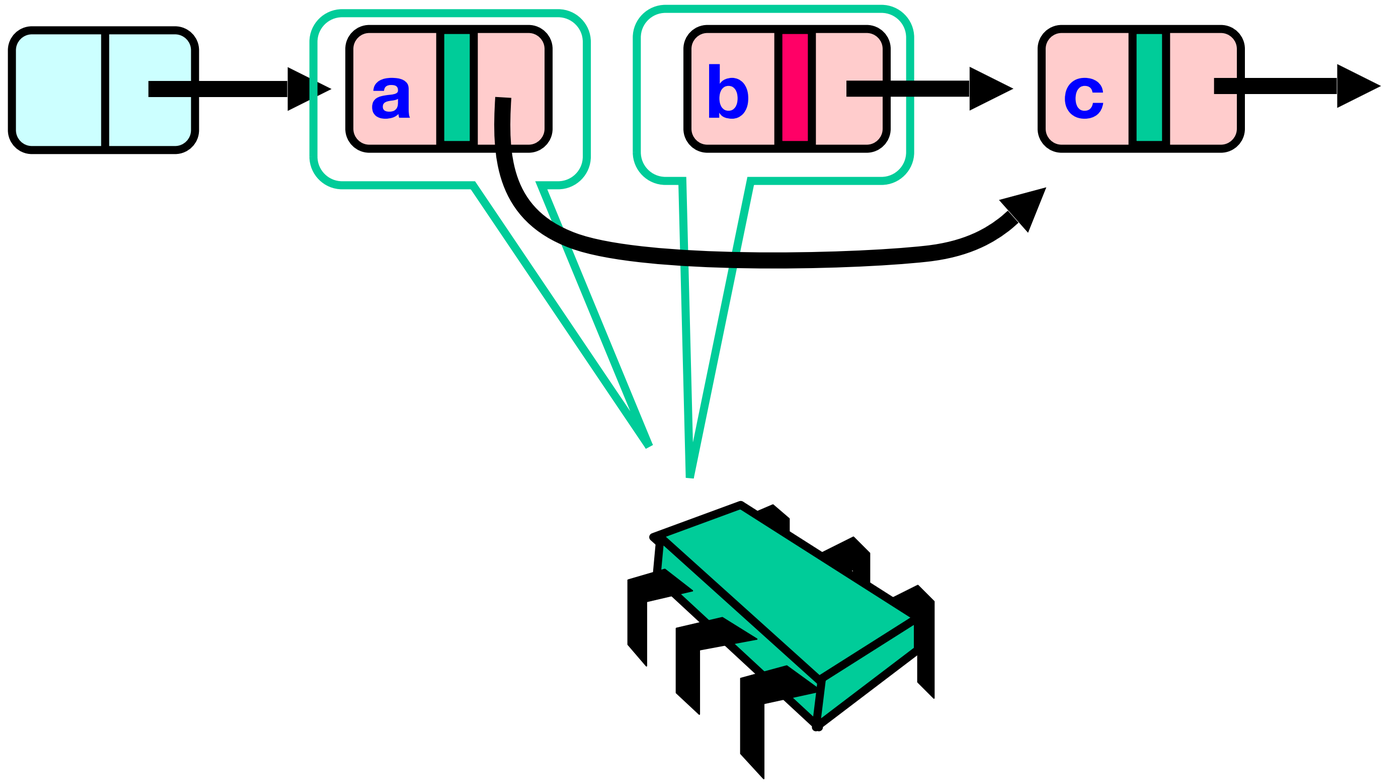
Business as Usual



Business as Usual



Business as Usual



New Abstraction Map

$$S(head) = \left\{ \begin{array}{l} x \mid \exists a \,.\, a \text{ is reachable from } head \\ \qquad \wedge\ a \,.\, item = x \\ \qquad \wedge\ a \text{ is unmarked} \end{array} \right\}$$

Invariant

- If not marked, then the item in the set
  - Logical deletion precedes physical deletion
- Both **pred** and **curr** are locked and verfied to be unmarked
  - pred and curr are in the set!
- Need to also check **pred** still points to **curr**
  - To handle addition

Validation

```
(** Validate that pred and curr are still adjacent and unmarked.

    Pre-condition:
      - pred.lock is held
      - curr.lock is held

    Returns true if both nodes are unmarked and pred.next = curr.
    Much simpler than optimistic list validation - no need to traverse from head!
*)
let validate pred curr =
  not pred.marked && not curr.marked && pred.next == curr
```

Remove

```
let remove list item =
  let key = Hashtbl.hash item in
  let rec attempt () =
    let (pred, curr) = locate list.head key in
    Mutex.lock pred.lock; Mutex.lock curr.lock;
    if validate pred curr then begin
      let result =
        if curr.key = key then begin
          (* element found, mark it then physically remove *)
          curr.marked <- true;      (* logical deletion *)
          pred.next <- curr.next;   (* physical deletion *)
          true
        end else false  (* element not present *)
      in
      Mutex.unlock curr.lock; Mutex.unlock pred.lock;
      result
    end else begin
      Mutex.unlock curr.lock; Mutex.unlock pred.lock;
      attempt ()  (* validation failed, retry *)
    end
  in
  attempt ()
```

Contains

- `Contains()` is wait-free!
  - Completes in a finite number of steps (no retry loops)

```
let contains list item =
  let key = Hashtbl.hash item in
  let rec loop curr =
    if curr.key < key then
      loop curr.next
    else
      curr.key = key && not curr.marked
  in
  loop list.head
```

Contains

- `Contains()` is wait-free!
  - Completes in a finite number of steps (no retry loops)
- No locks needed!

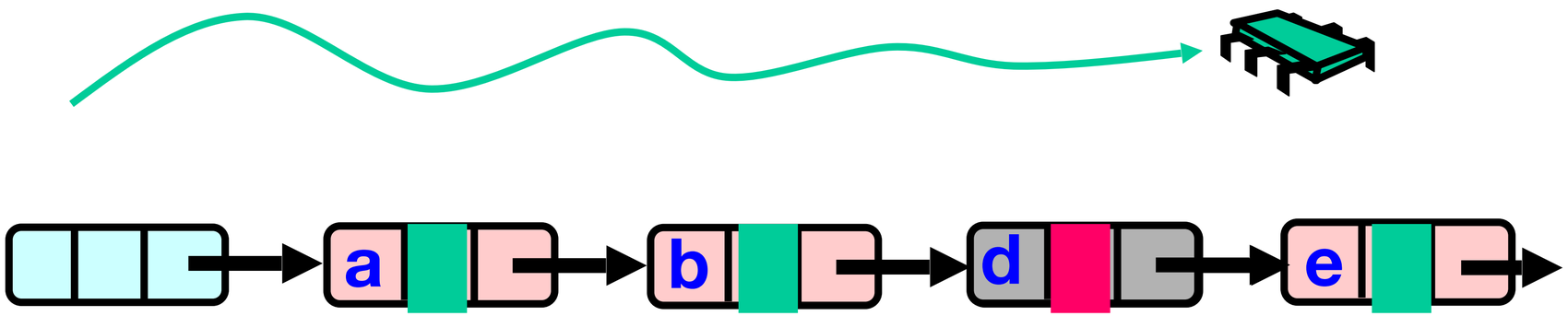
```
let contains list item =
  let key = Hashtbl.hash item in
  let rec loop curr =
    if curr.key < key then
      loop curr.next
    else
      curr.key = key && not curr.marked
  in
  loop list.head
```

Contains

- `Contains()` is wait-free!
  - Completes in a finite number of steps (no retry loops)
- No locks needed!
- We can safely traverse without locking because
  - Marked nodes stay marked (monotonic property)
  - We only report true if key matches AND not marked
  - Even if a node is being concurrently removed, we'll see either
    - the old state (unmarked, report true) or new state (marked, report false),
    - both of which are valid linearization points.

```
let contains list item =
  let key = Hashtbl.hash item in
  let rec loop curr =
    if curr.key < key then
      loop curr.next
    else
      curr.key = key && not curr.marked
  in
  loop list.head
```

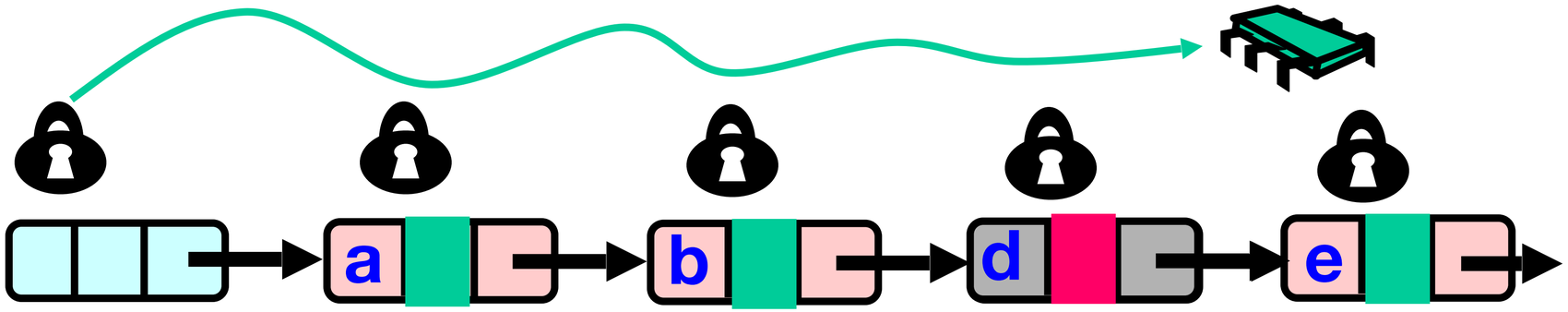
Summary: Wait-free Contains



Use Mark bit + list ordering

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

Lazy List



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

Evaluation

- Good:
  - **`contains()`** doesn't lock
  - In fact, its wait-free!
  - Good because typically high % contains()
  - Uncontended calls don't re-traverse
- Bad
  - Contended **`add()`** and **`remove()`** calls must re-traverse
  - Traffic jam if one thread delays

Traffic Jam

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And “eats the big muffin”
    - Cache miss, page fault, descheduled …
  - Everyone else using that lock is stuck!
  - Need to trust the scheduler….

Lock-free data structures

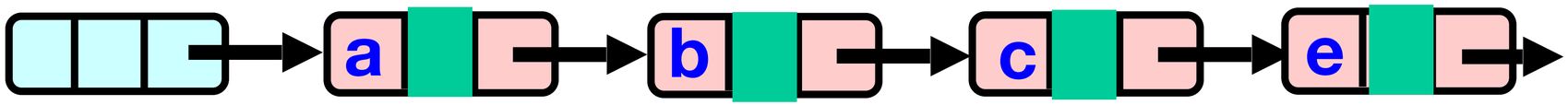
- No matter what …
  - Guarantees minimal progress in any execution
  - i.e. Some thread will always complete a method call
  - Even if others halt at unopportune times
  - Implies that implementation can’t use locks

(4) Lock-fre list

Lock-free Lists

- Next logical step
  - Wait-free **`contains()`**
  - lock-free **`add()`** and **`remove()`**
- Use only **`compareAndSet()`**
  - What could go wrong?

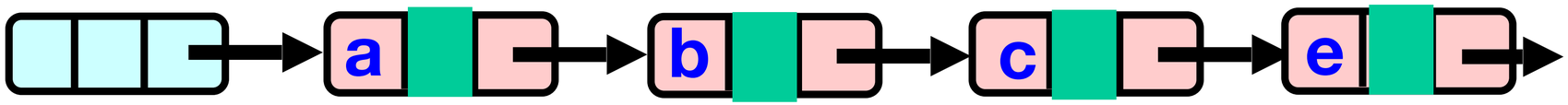
Lock-free Lists



Use CAS to verify pointer
is correct

Lock-free Lists

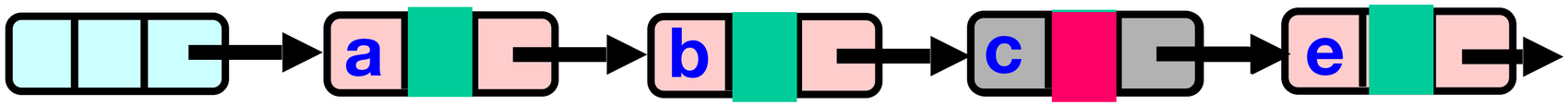
Logical Removal



Use CAS to verify pointer
is correct

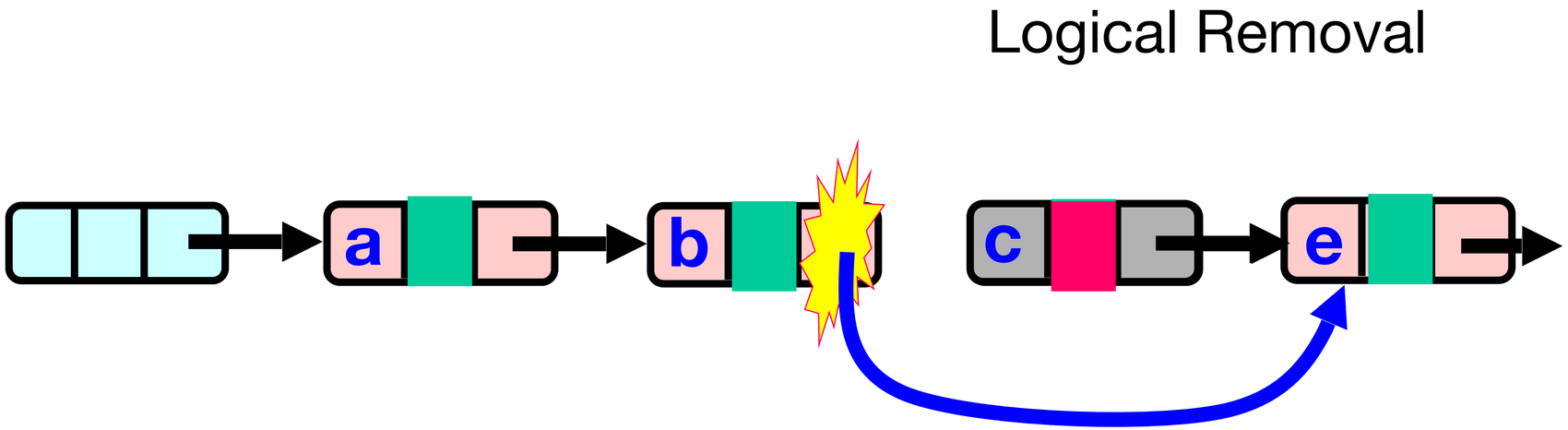
Lock-free Lists

Logical Removal



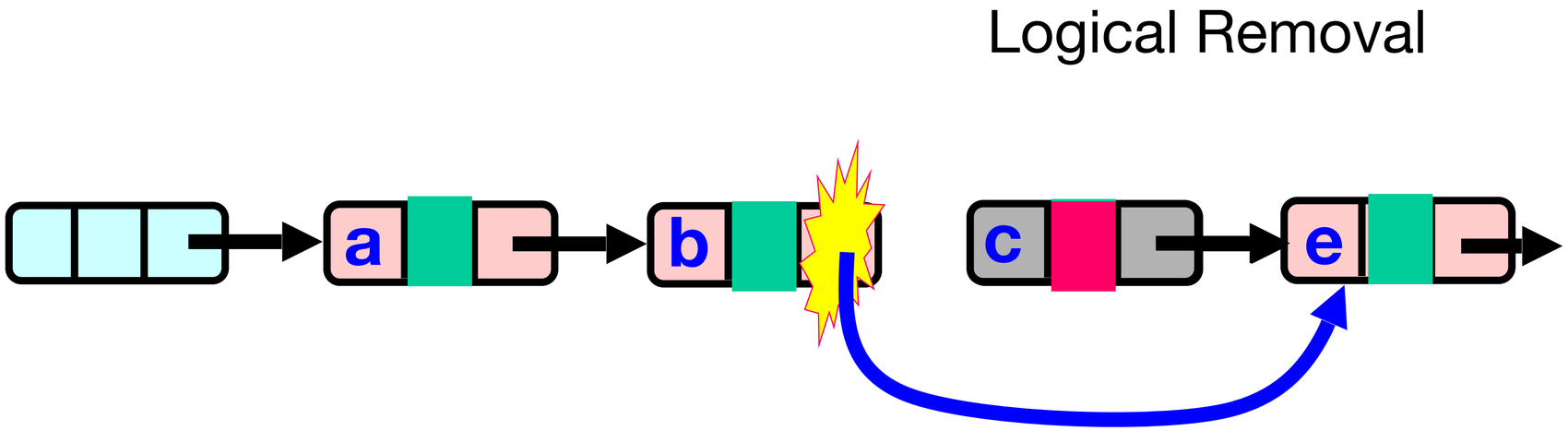
Use CAS to verify pointer
is correct

Lock-free Lists



Use CAS to verify pointer is correct

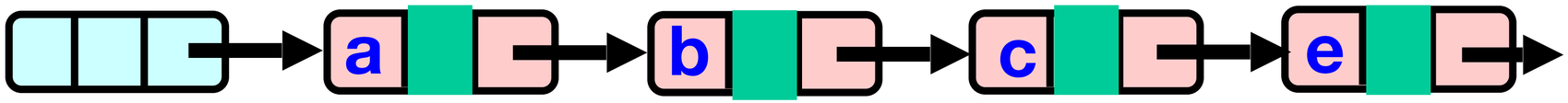
Lock-free Lists



Use CAS to verify pointer is correct

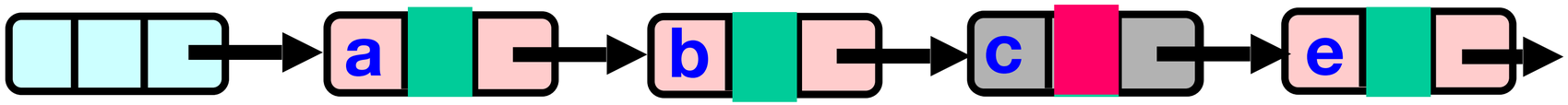
Not enough!

Problem….

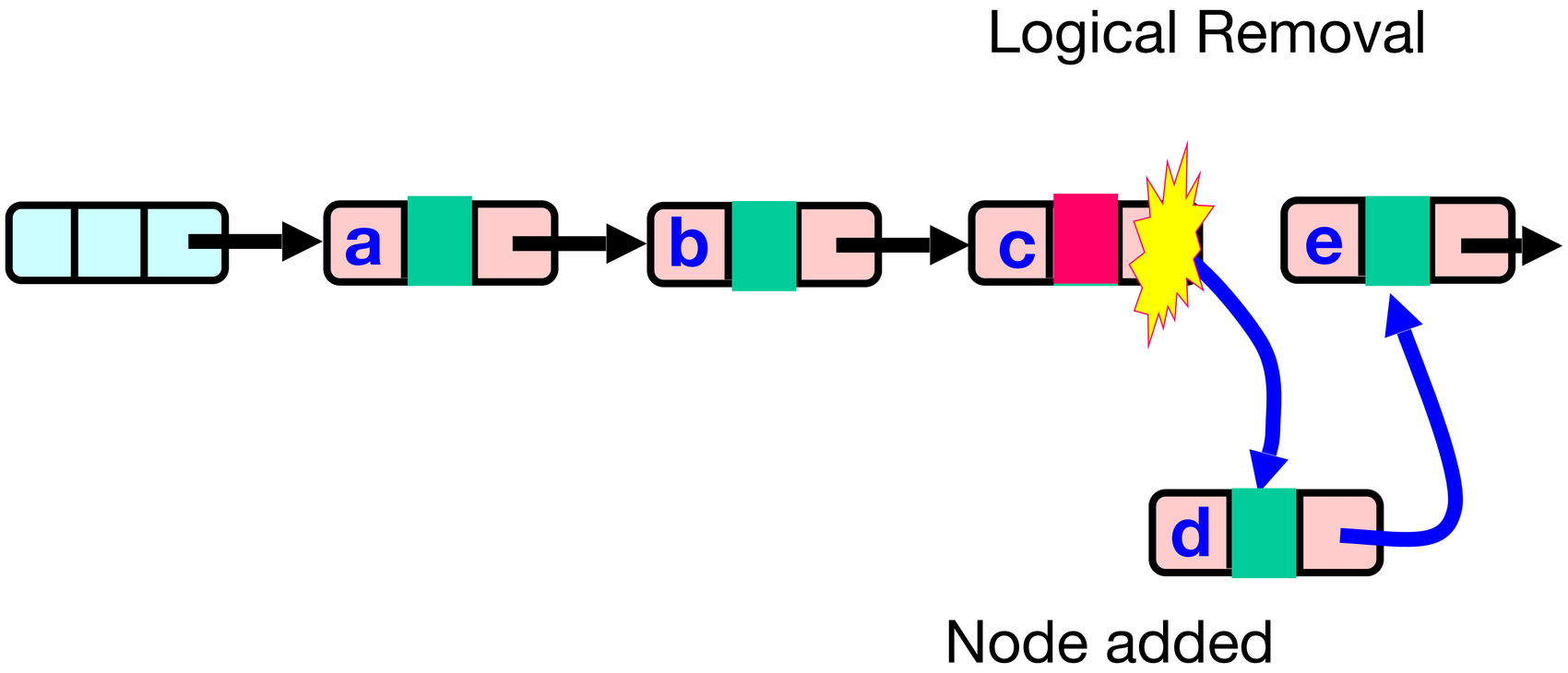


Problem….

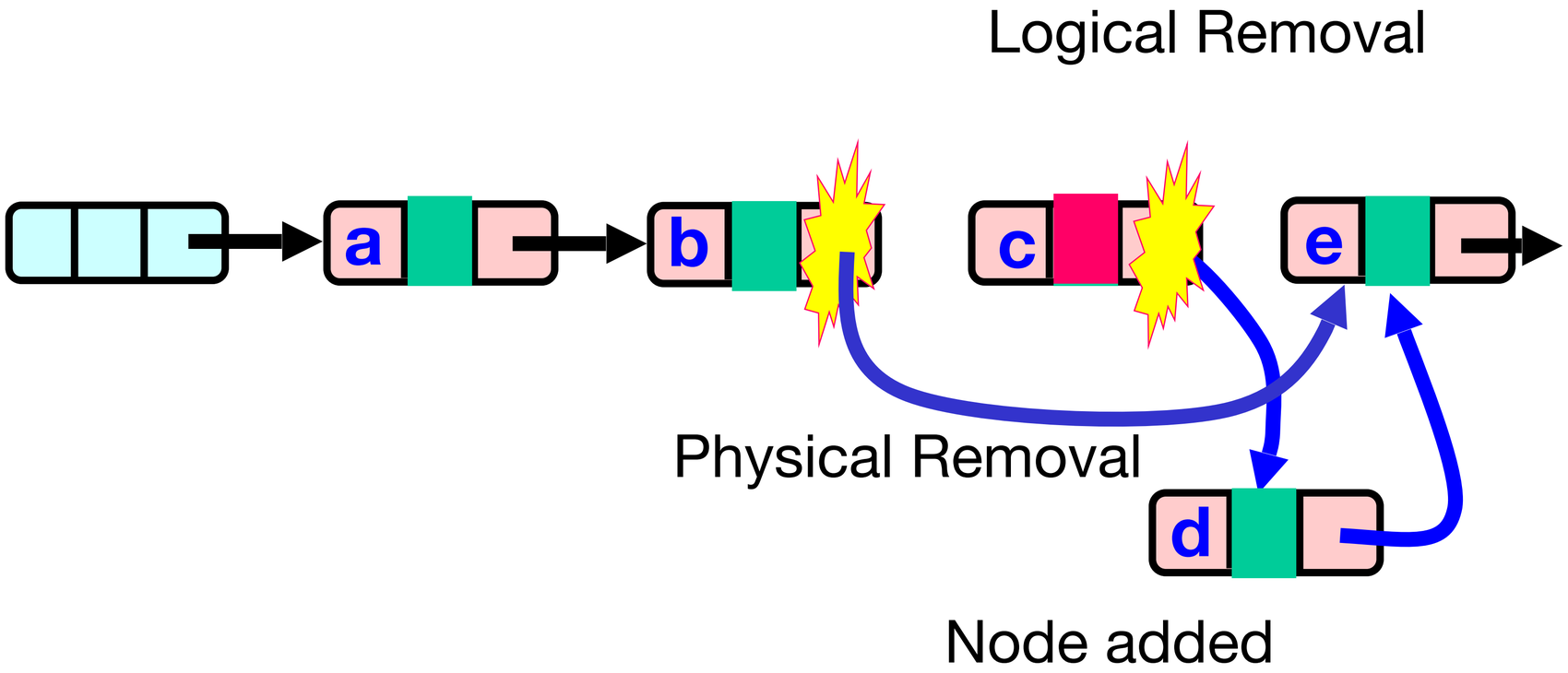
Logical Removal



Problem….



Problem….



Solution: Combine Bit and Pointer

Solution: Combine Bit and Pointer

Mark-Bit and Pointer
are CASed together
(Atomic_markable_ref)

Solution: Combine Bit and Pointer

Mark-Bit and Pointer
are CASed together
(Atomic_markable_ref)

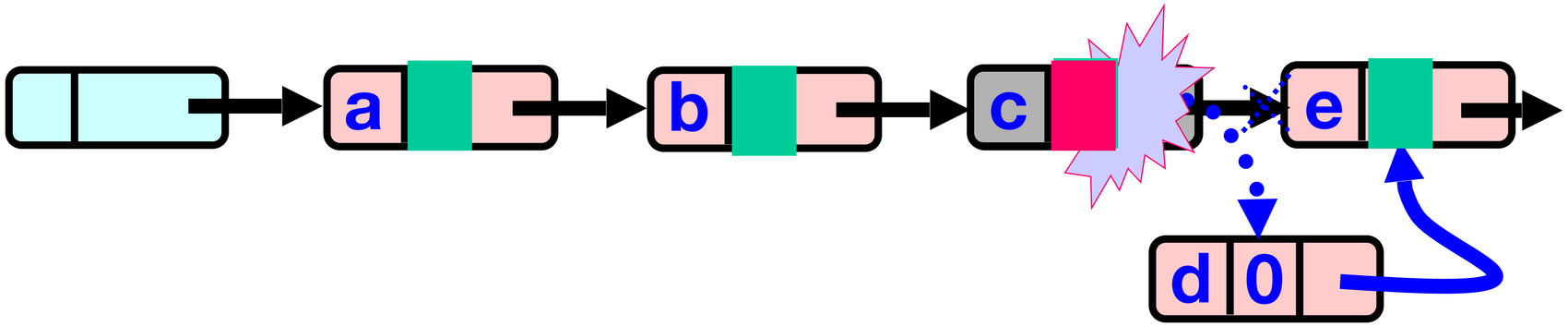
Solution: Combine Bit and Pointer

Logical Removal =
Set Mark Bit

Mark-Bit and Pointer
are CASed together
(Atomic_markable_ref)

Solution: Combine Bit and Pointer

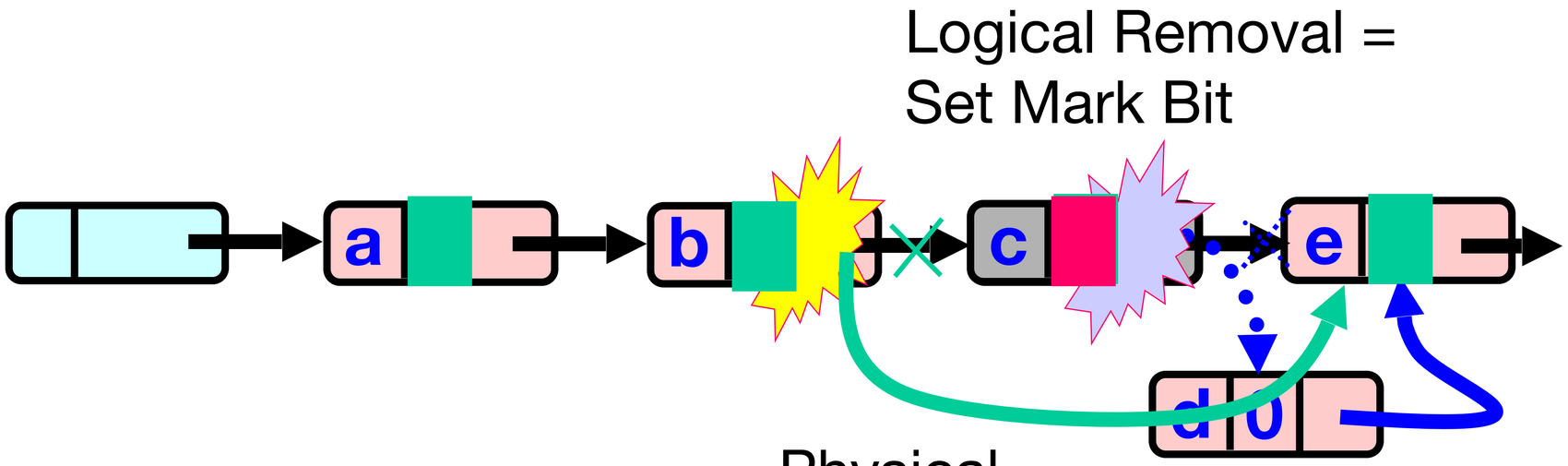
Logical Removal =
Set Mark Bit



Mark-Bit and Pointer are CASed together (Atomic_markable_ref)

Fail CAS: Node not added after logical Removal

Solution: Combine Bit and Pointer



Mark-Bit and Pointer are CASed together (Atomic_markable_ref)

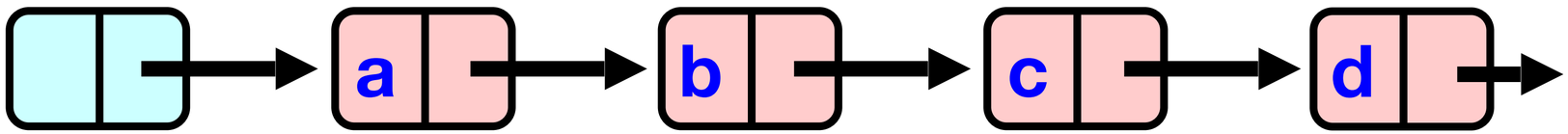
Solution

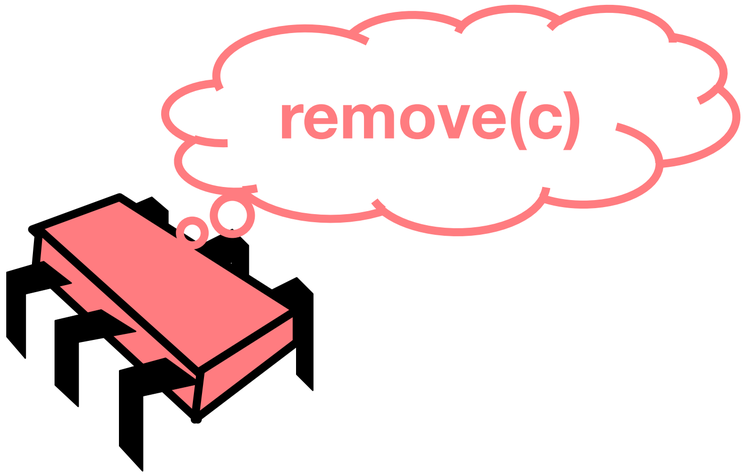
- Use **Atomic_markable_ref**
- Atomically
  - Swing reference and
  - Update flag
- Remove in two steps
  - Set **mark** bit in **next** field
  - Redirect predecessor’s pointer

Atomic_markable_ref

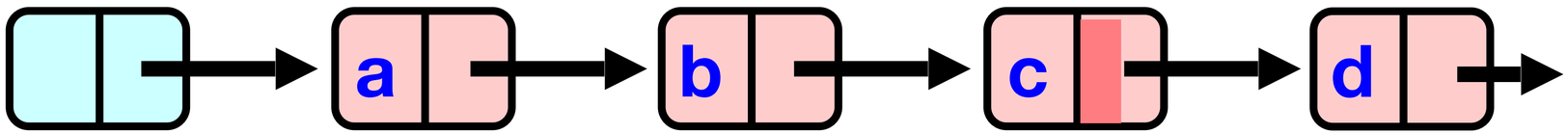
```
type 'a t
(** An atomic reference that maintains a mark bit along with the reference *)

val create : 'a -> bool -> 'a t
(** [create ref mark] creates a new atomic markable reference *)

val get_reference : 'a t -> 'a
(** [get_reference amr] gets the reference value *)

val get_mark : 'a t -> bool
(** [get_mark amr] gets the mark value *)

val get : 'a t -> bool ref -> 'a
(** [get amr marked] atomically reads both fields, stores the mark in [marked],
    and returns the reference. *)

val compare_and_set : 'a t -> expected_ref:'a -> new_ref:'a
                   -> expected_mark:bool -> new_mark:bool -> bool
(** [compare_and_set amr ~expected_ref ~new_ref ~expected_mark ~new_mark]
    atomically sets the value of both the reference and mark to the given
    update values if the current reference is == to the expected reference
    and the current mark is equal to the expected mark.
    Returns true if successful. *)
```

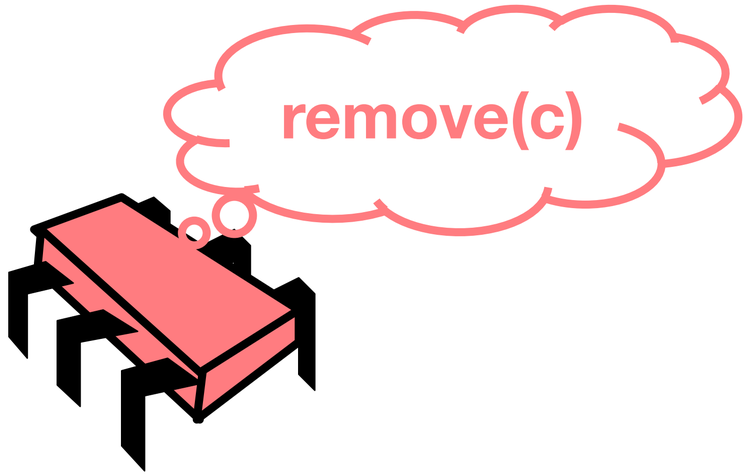
Removing a Node



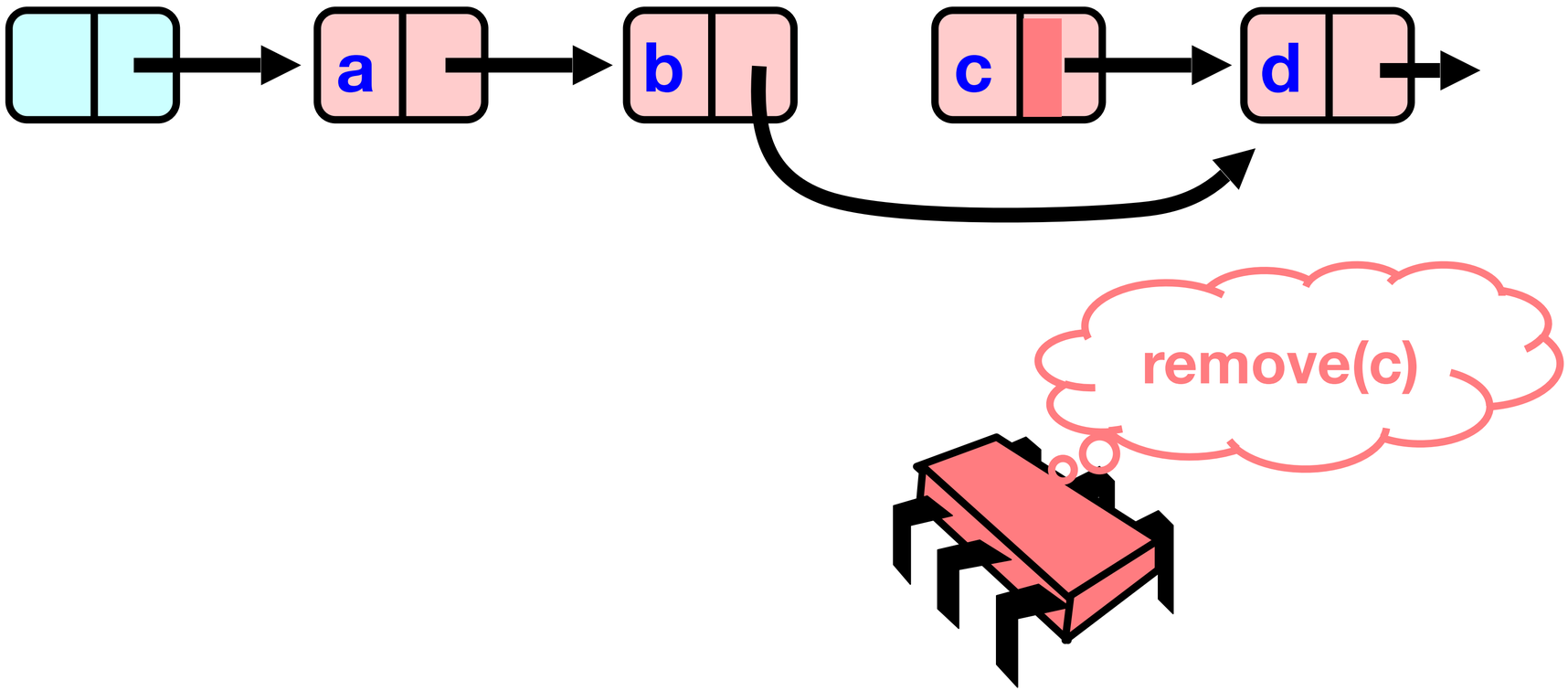


Removing a Node

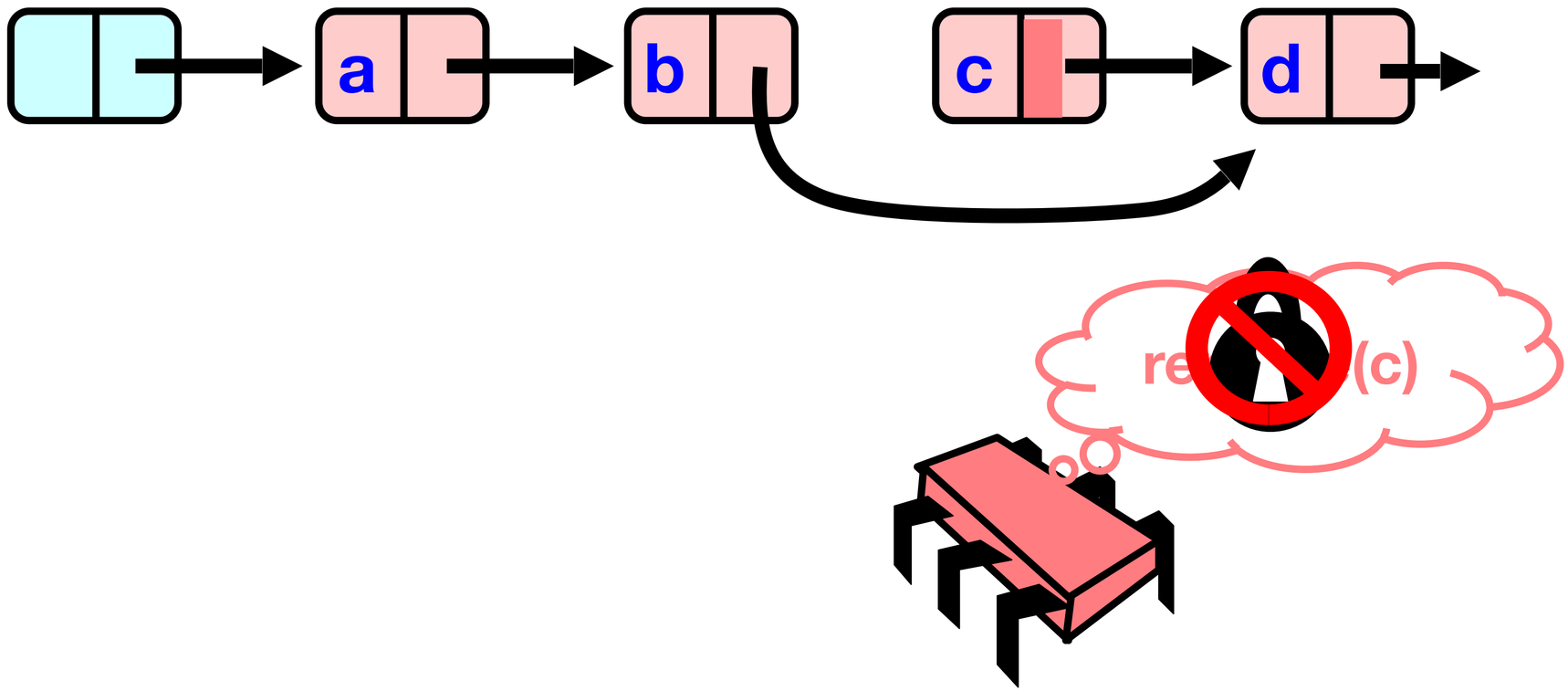




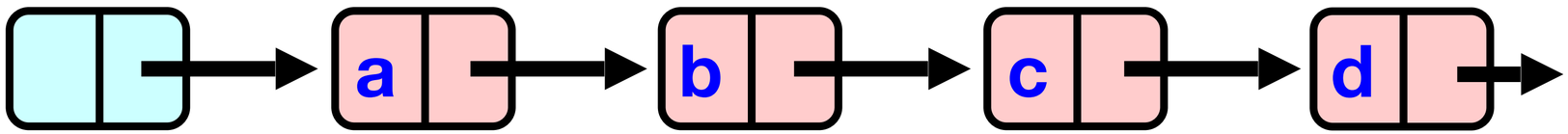
Removing a Node

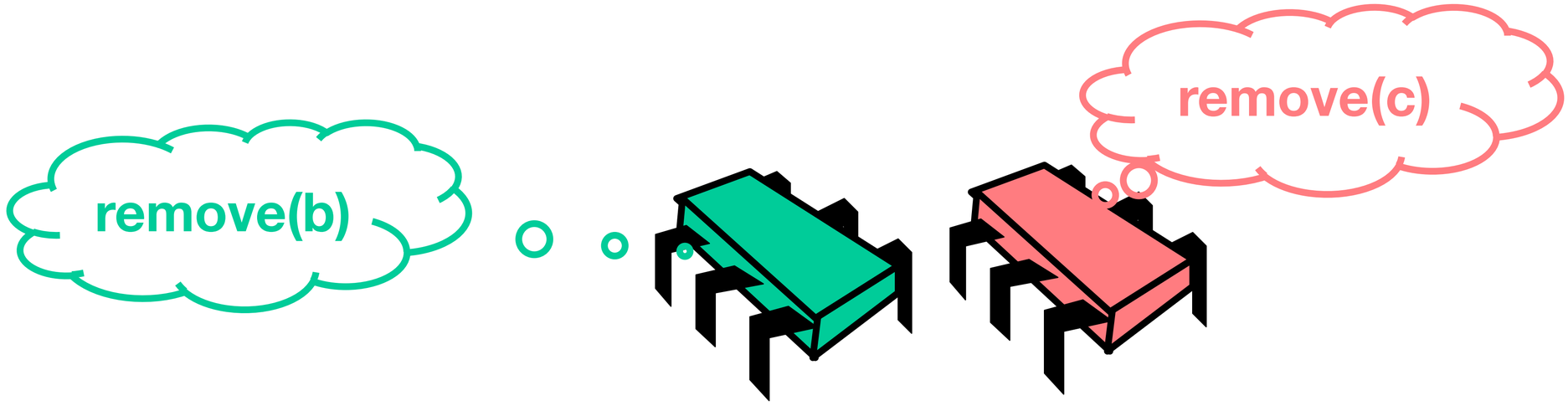


Removing a Node

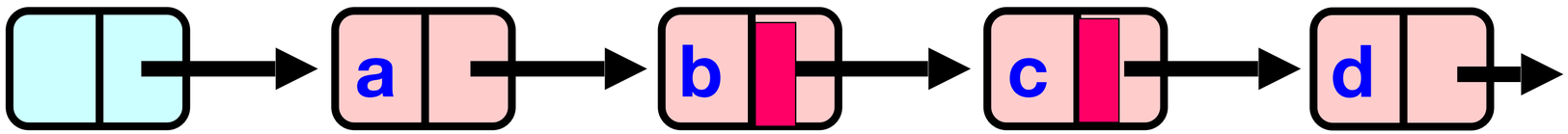


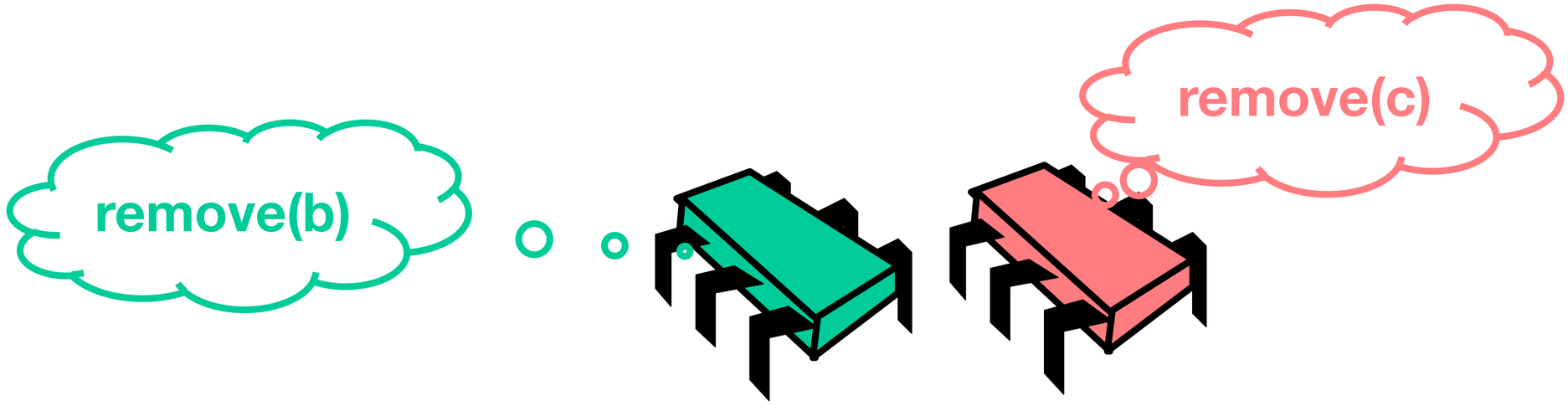
Removing a Node



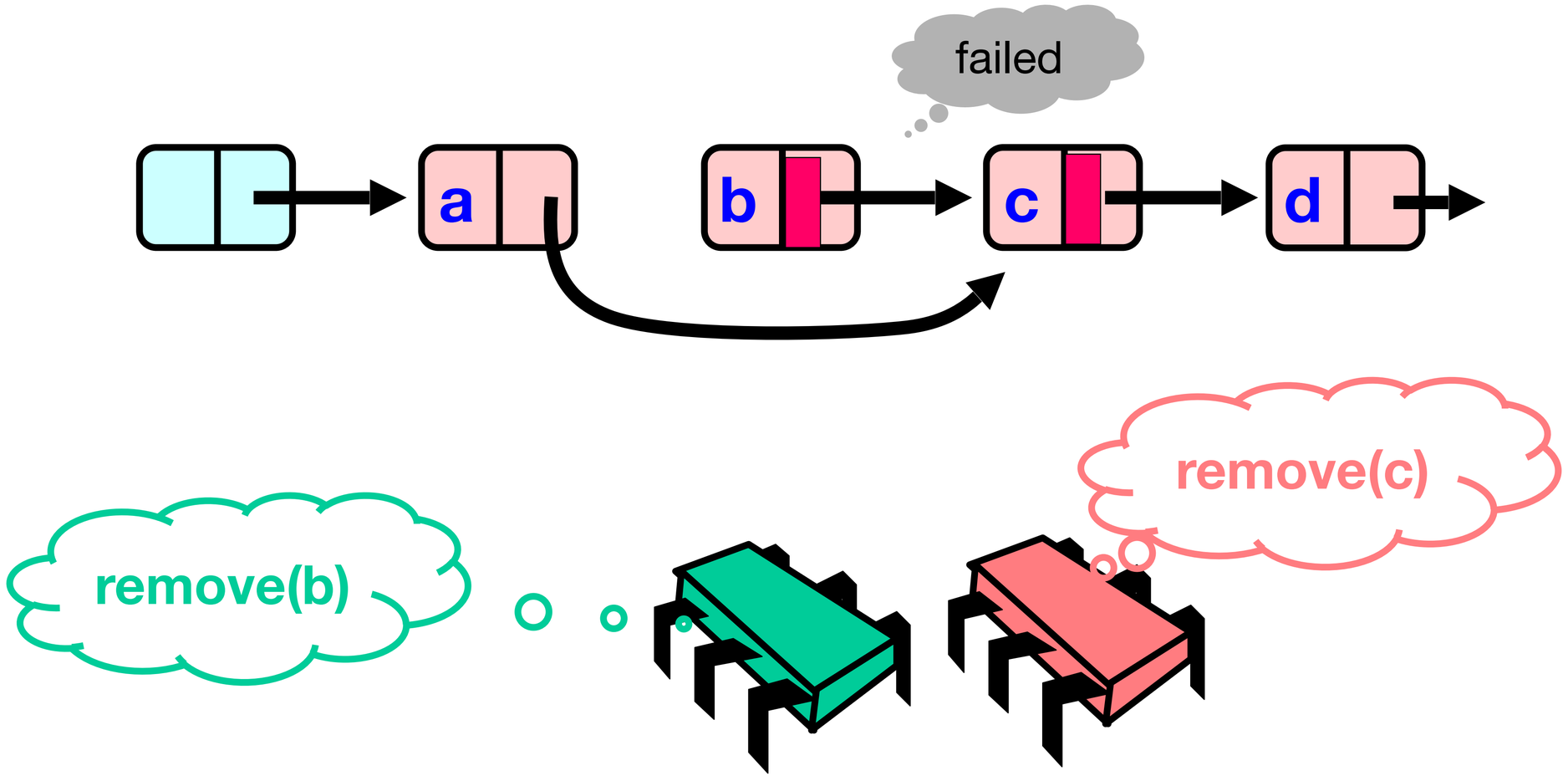


Removing a Node

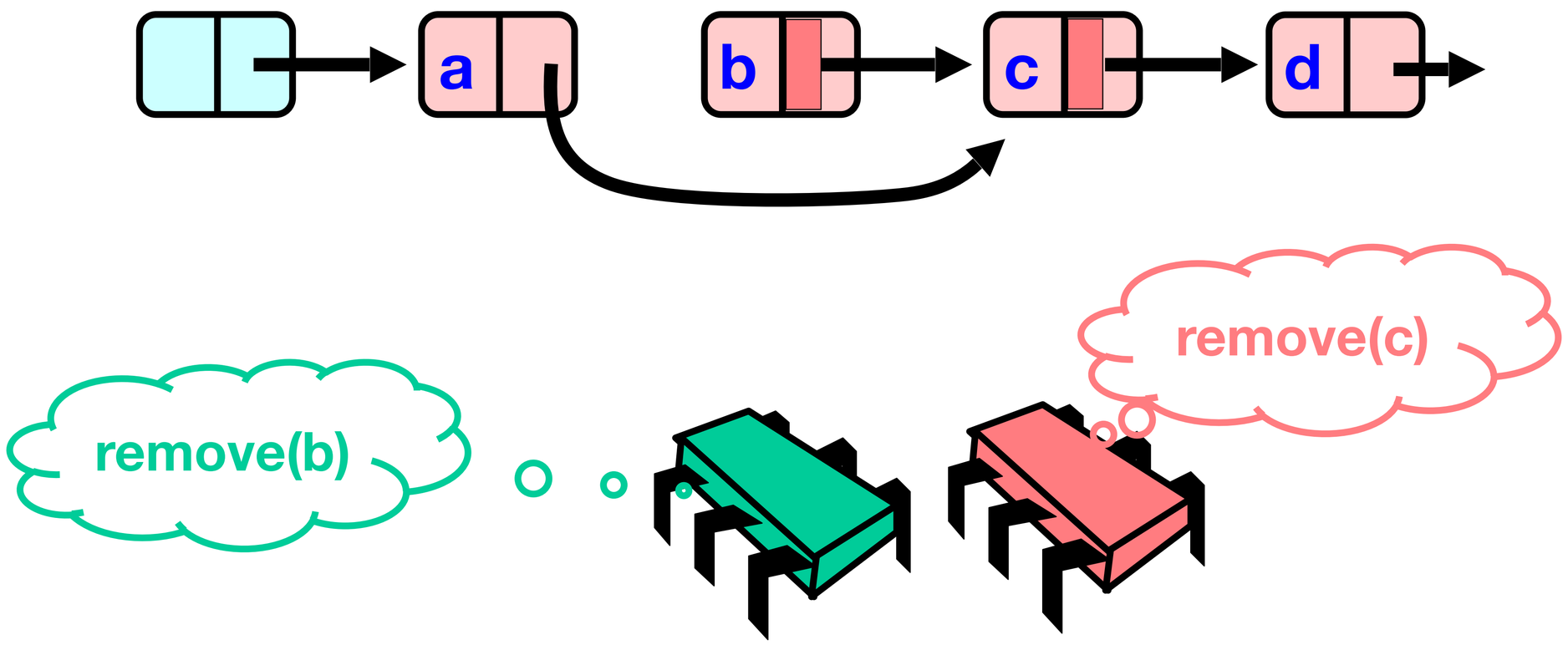




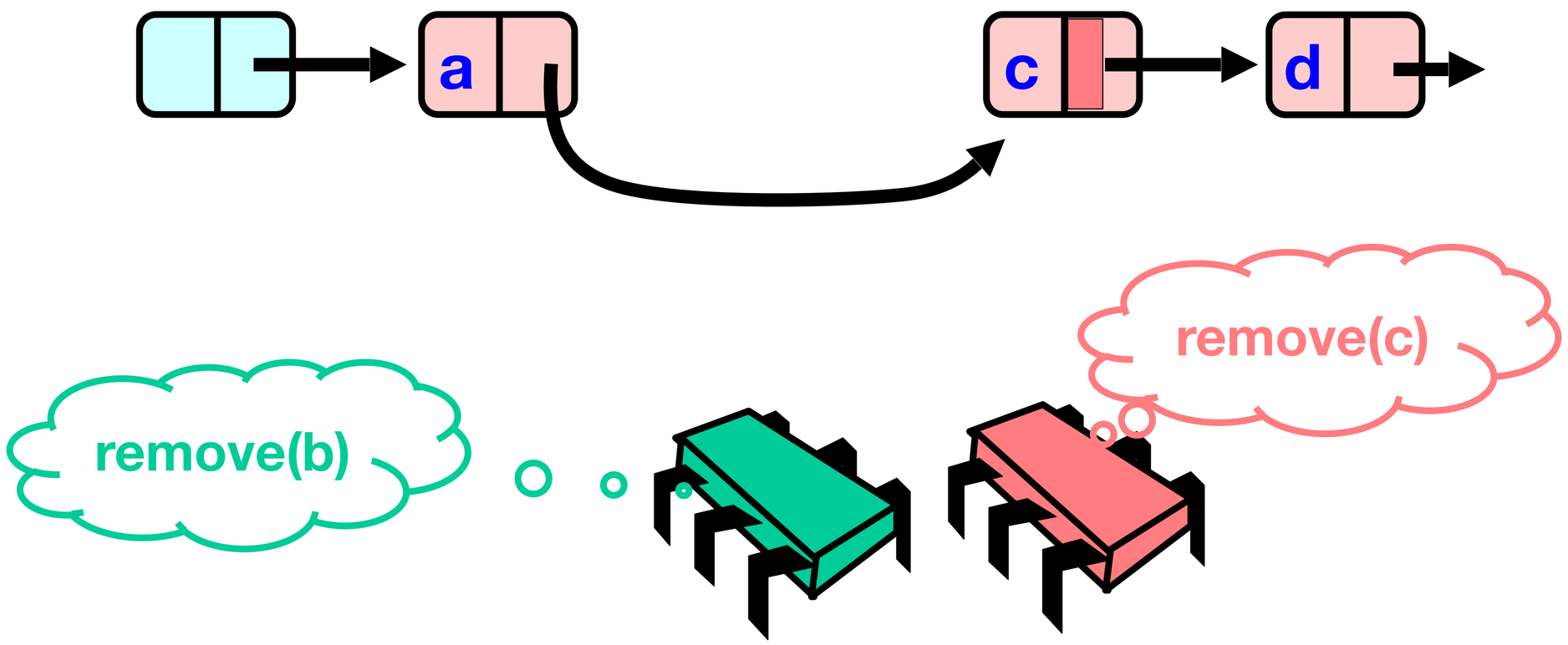
Removing a Node



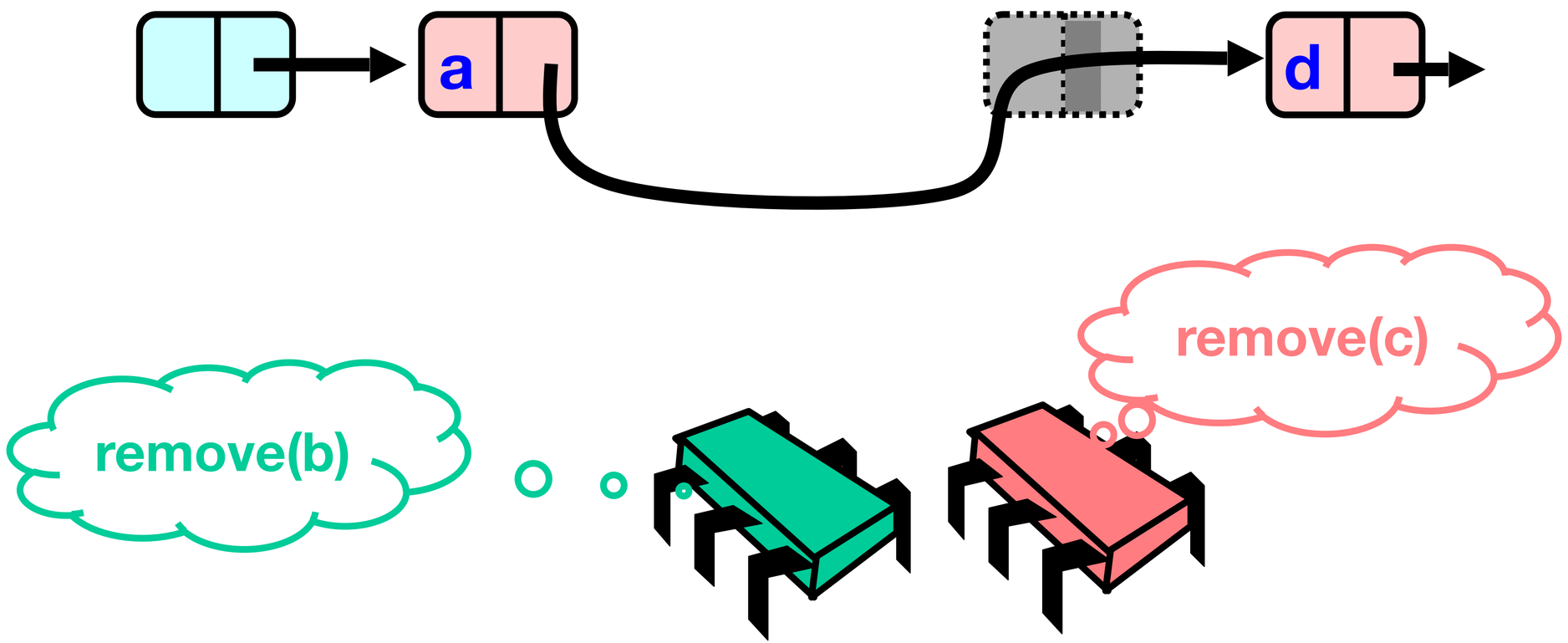
Removing a Node



Removing a Node



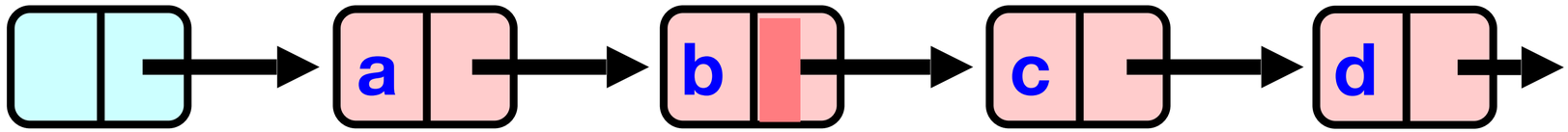
Removing a Node

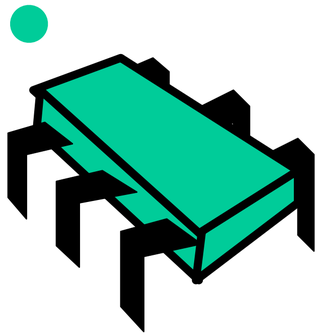


Traversing the List

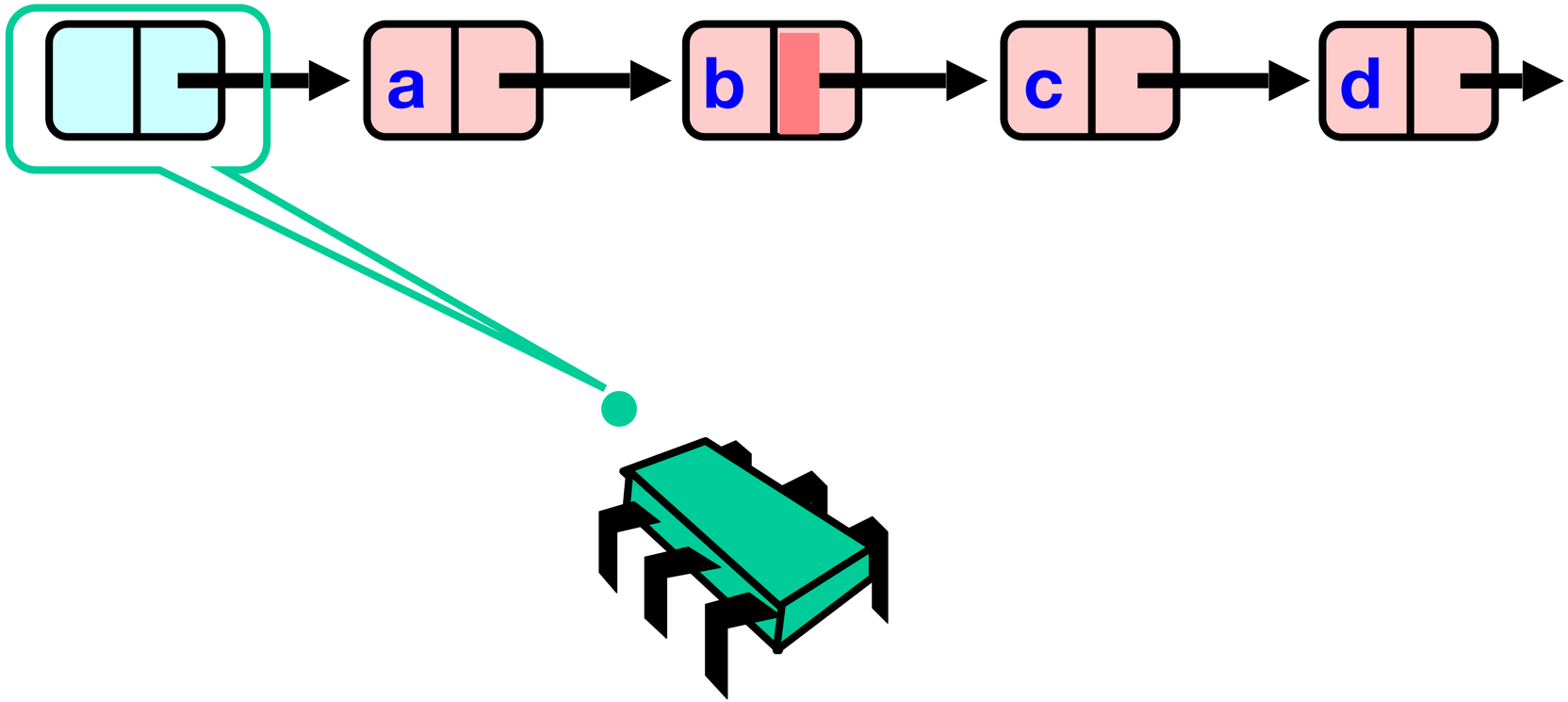
- **Q:** what do you do when you find a “logically” deleted node in your path?
- **A:** finish the job.
  - CAS the predecessor’s next field
  - Proceed (repeat as needed)

Lock-free traversal (only add and remove)

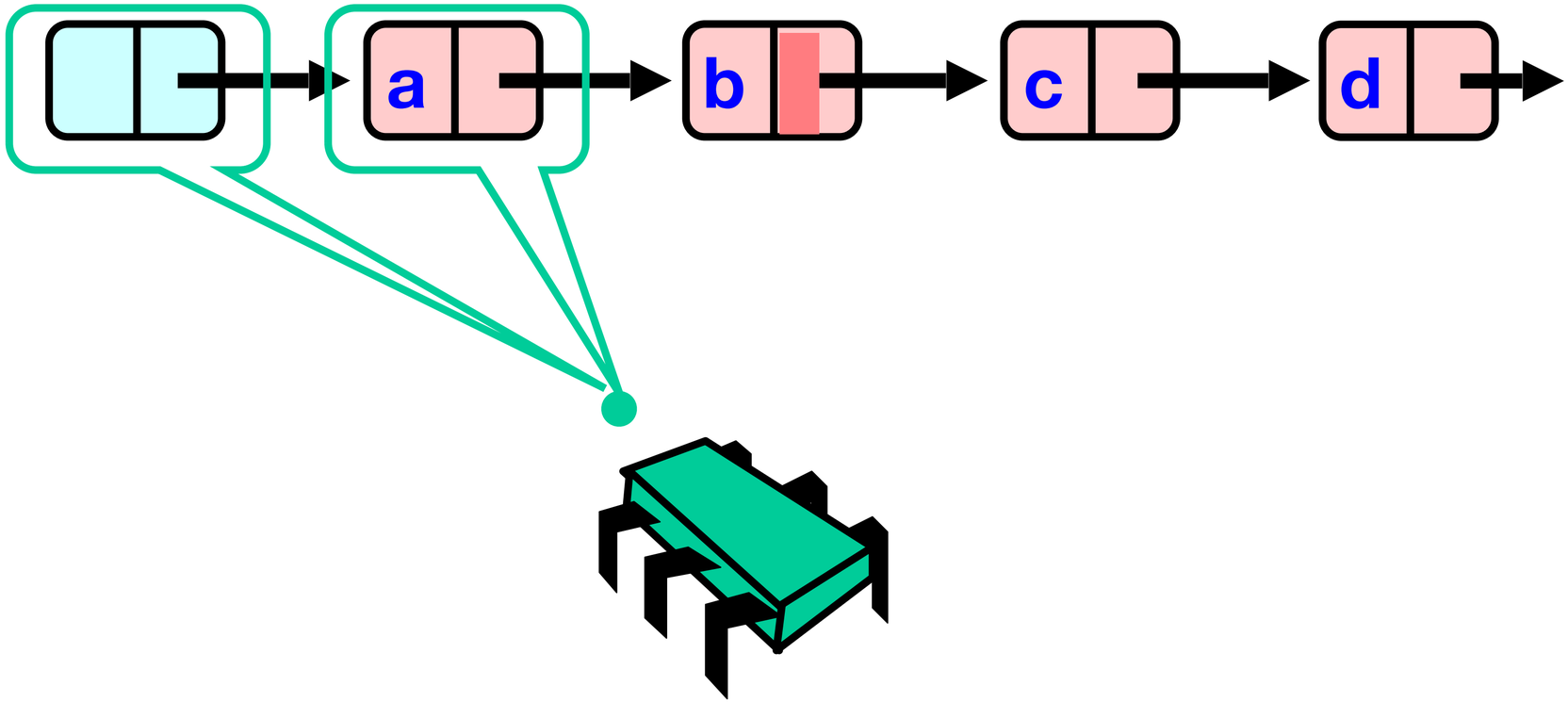




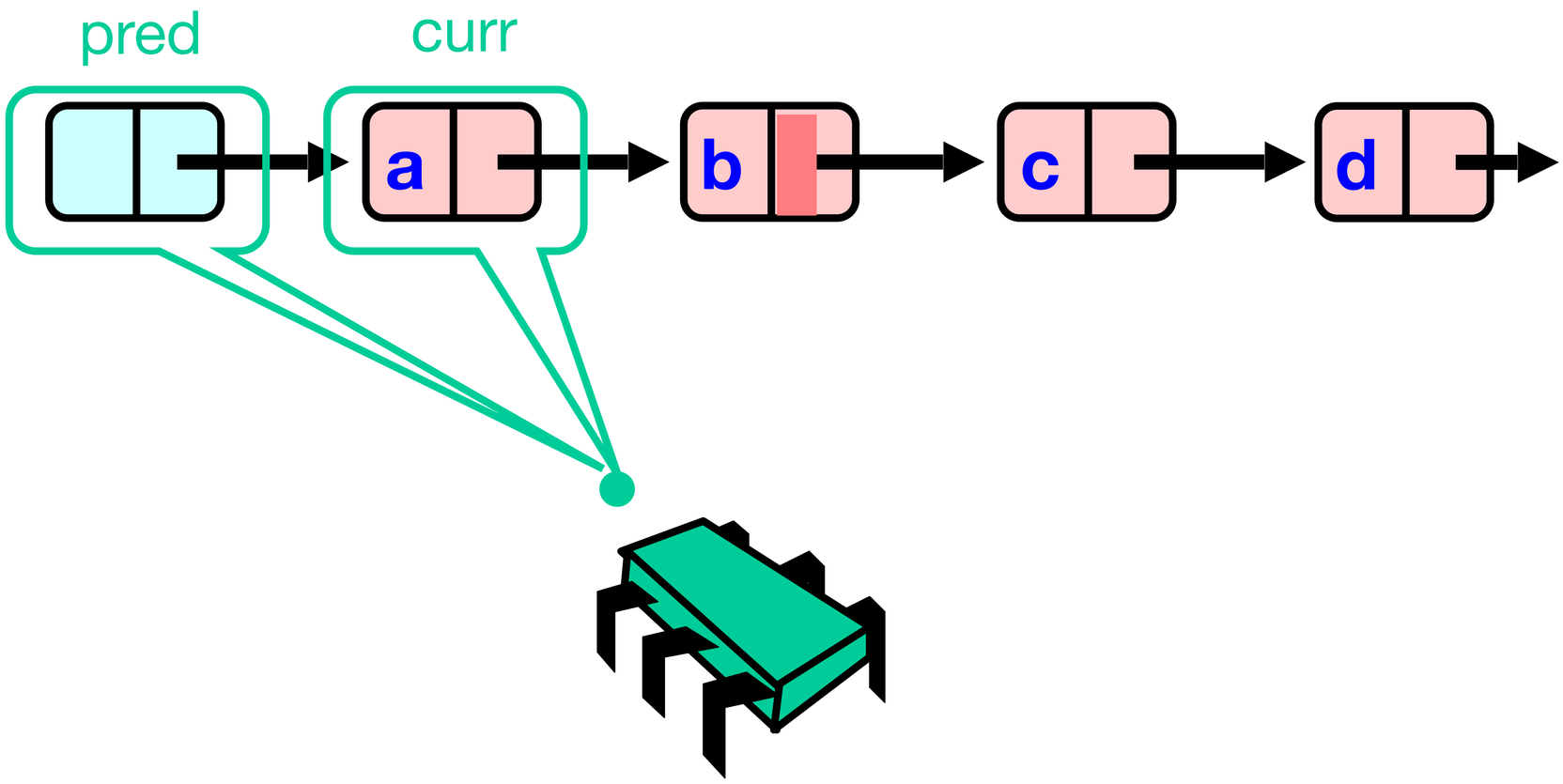
Lock-free traversal (only add and remove)



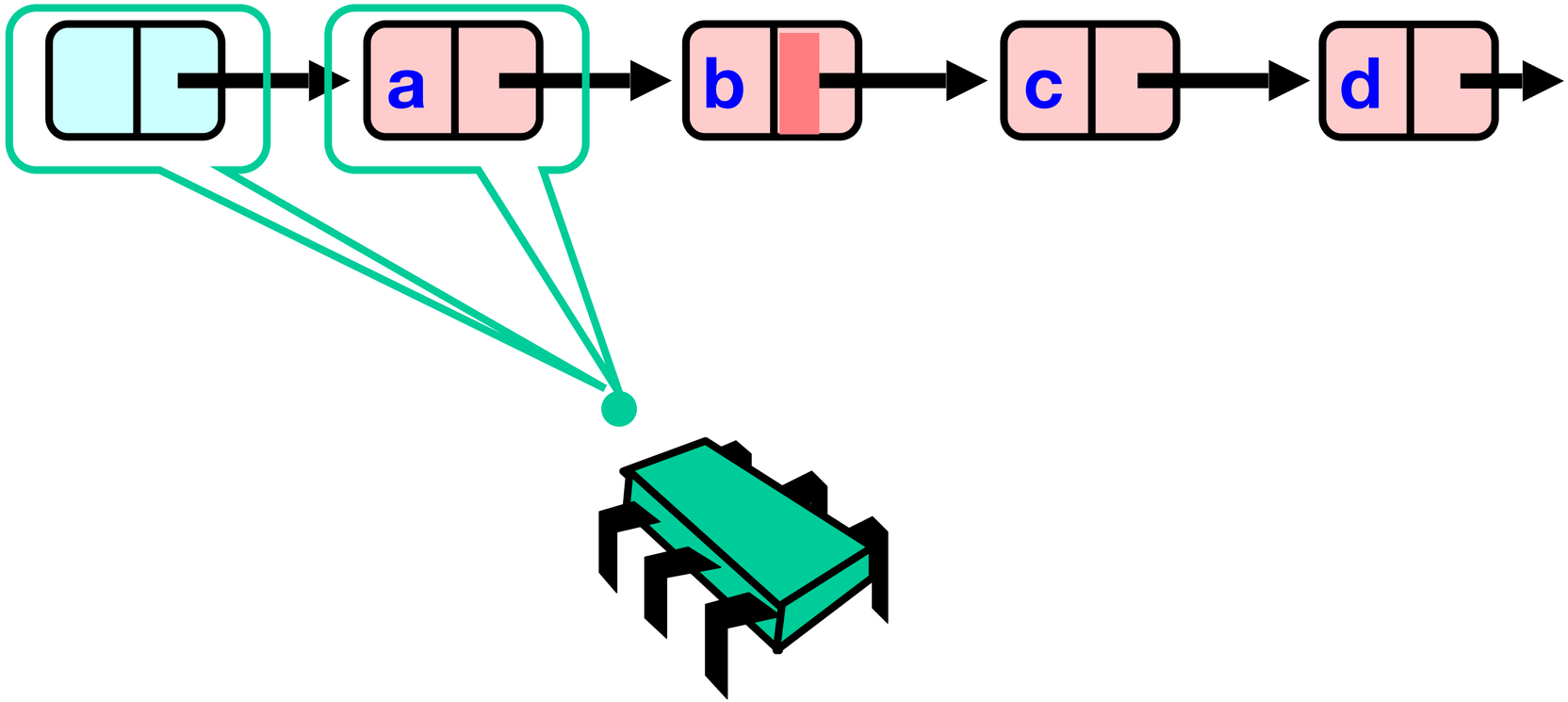
Lock-free traversal (only add and remove)



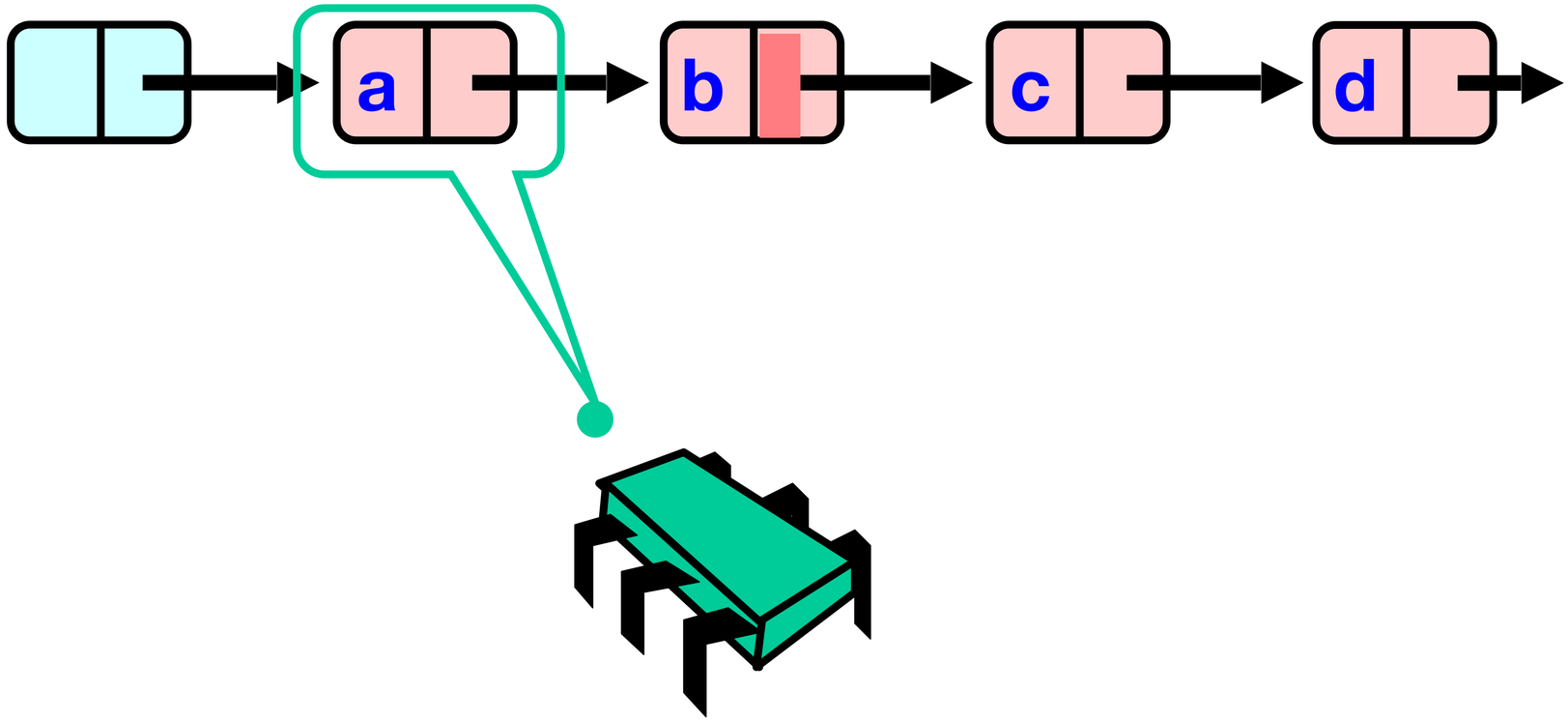
Lock-free traversal (only add and remove)



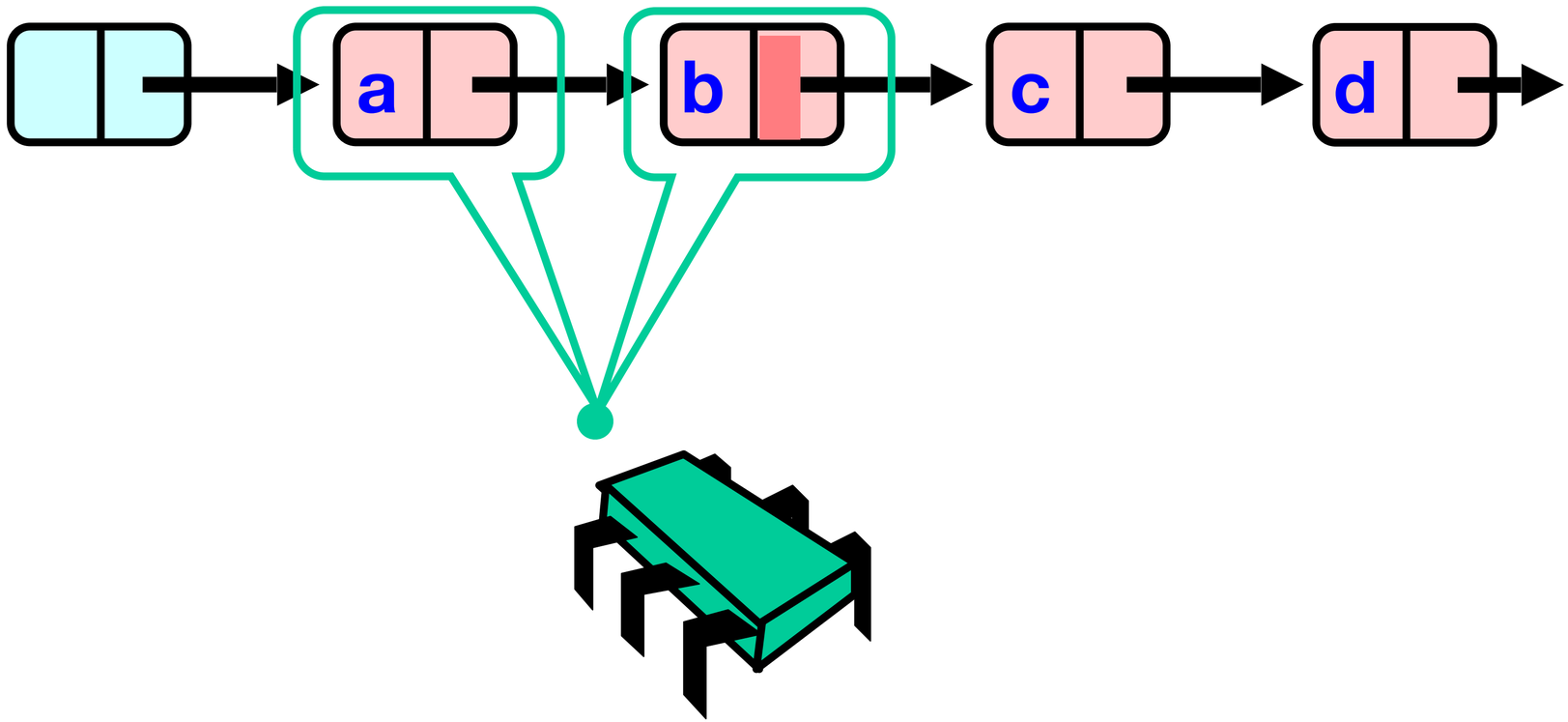
Lock-free traversal (only add and remove)



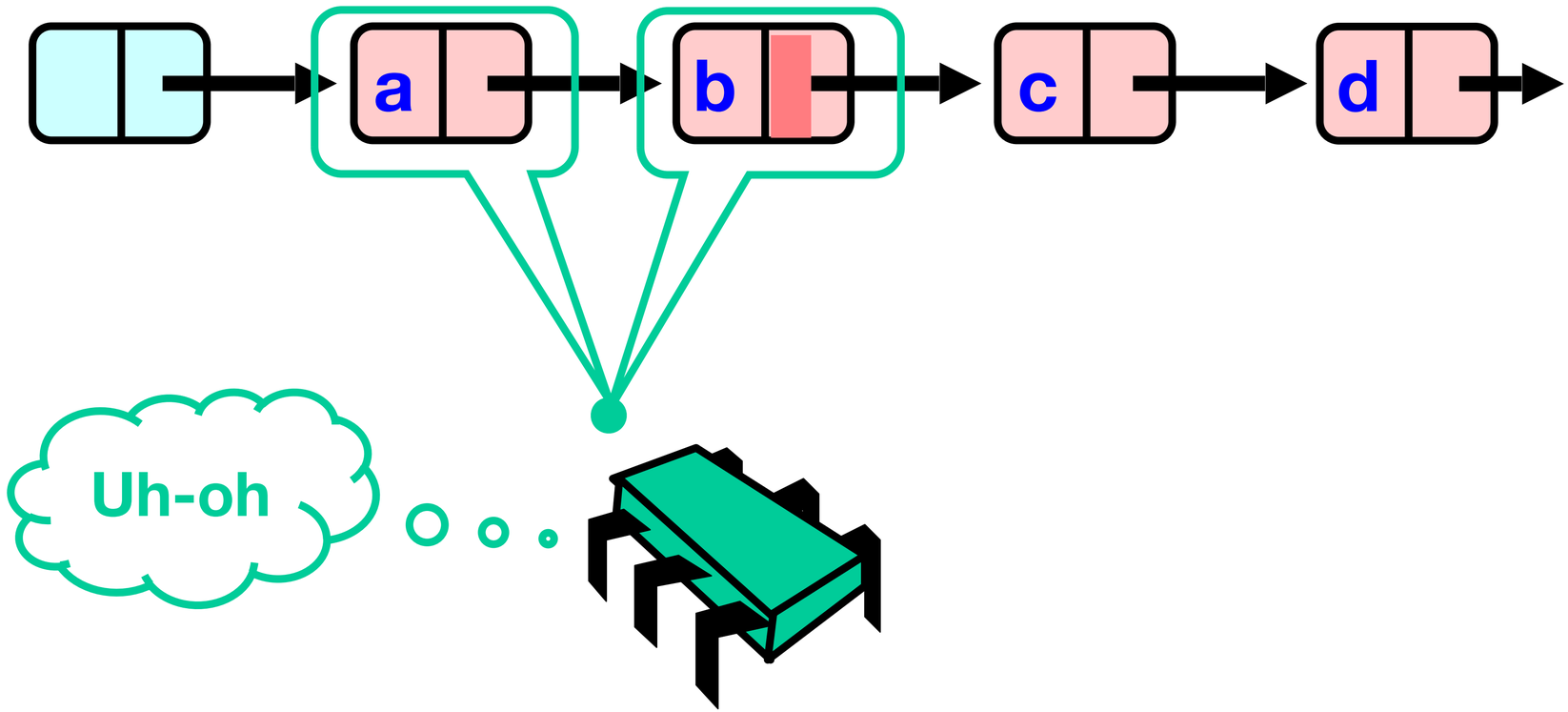
Lock-free traversal (only add and remove)



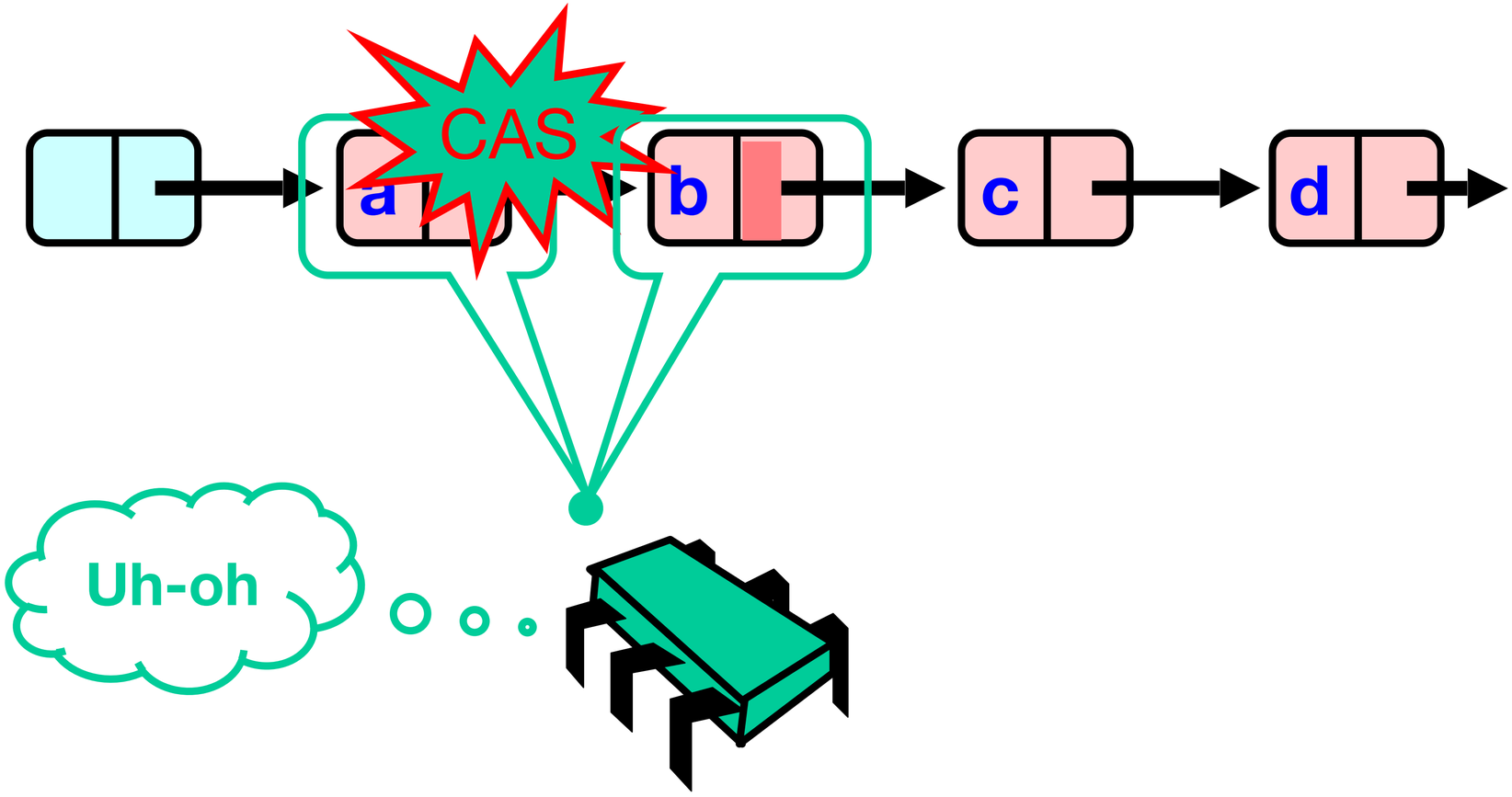
Lock-free traversal (only add and remove)



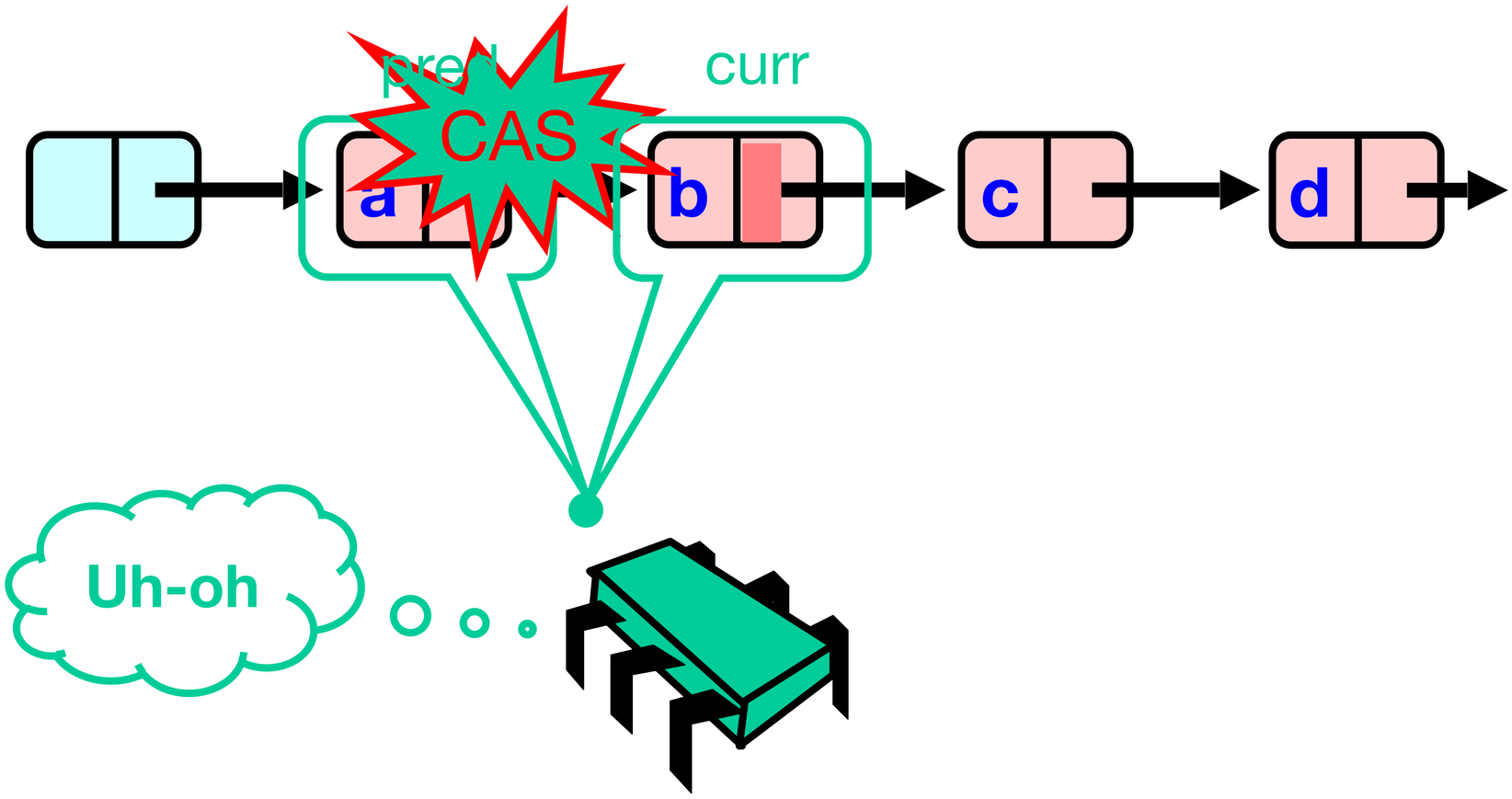
Lock-free traversal (only add and remove)



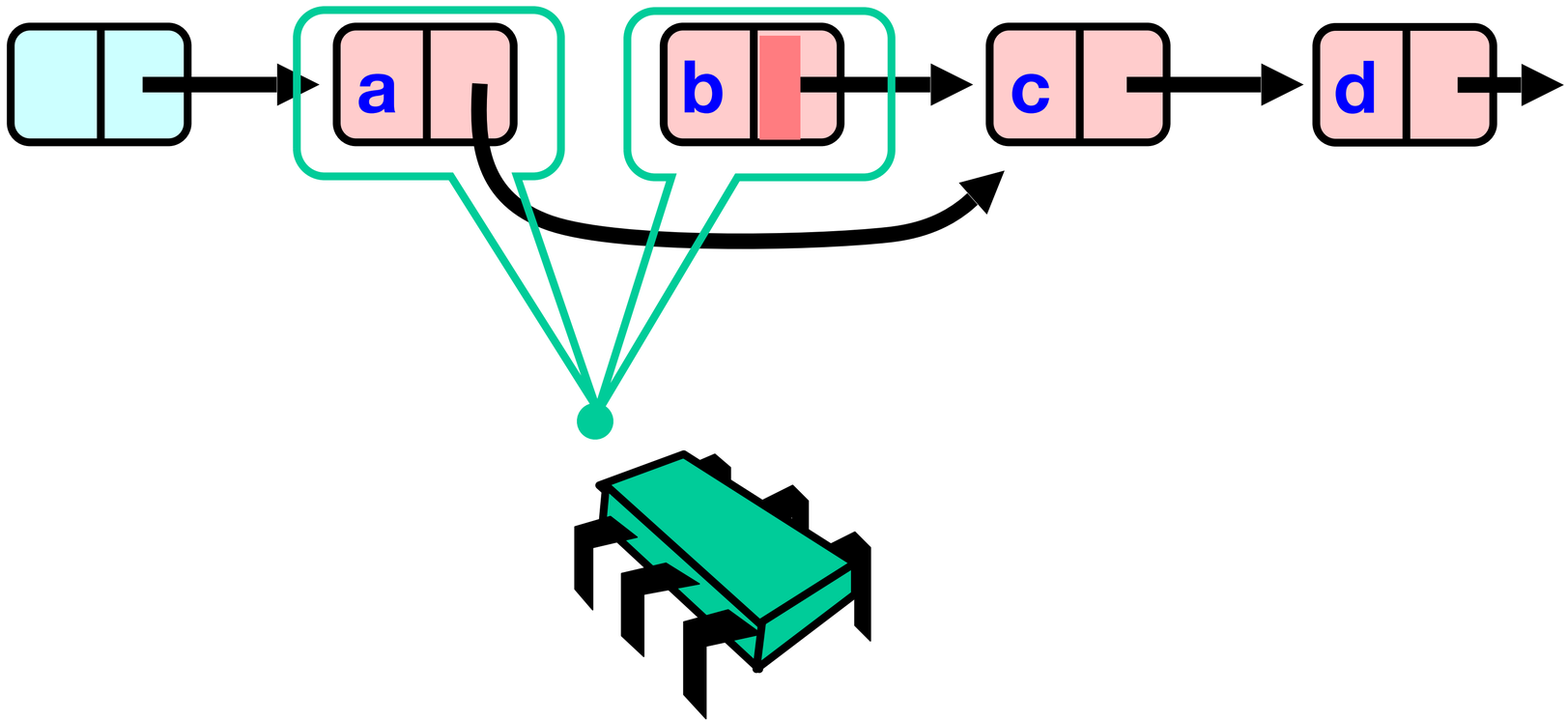
Lock-free traversal (only add and remove)



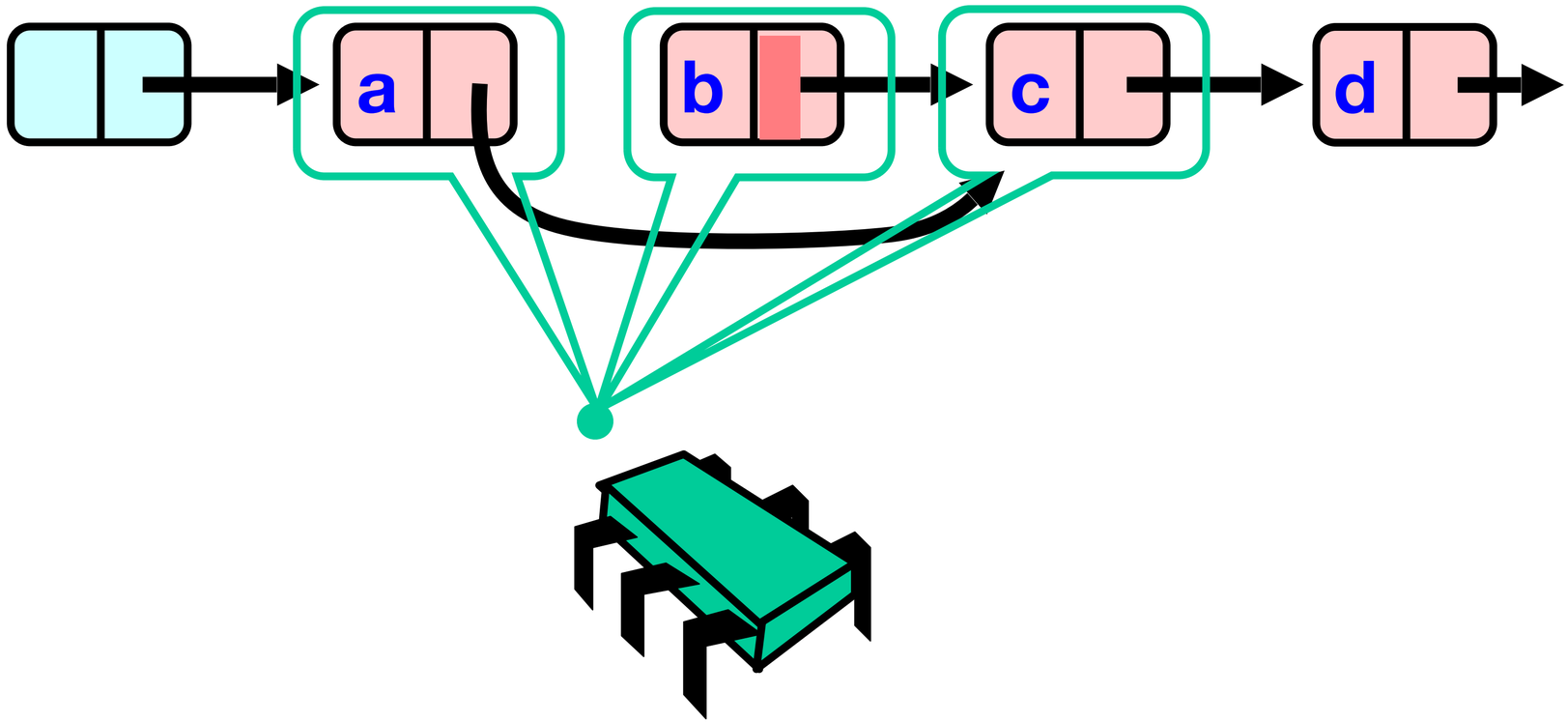
Lock-free traversal (only add and remove)



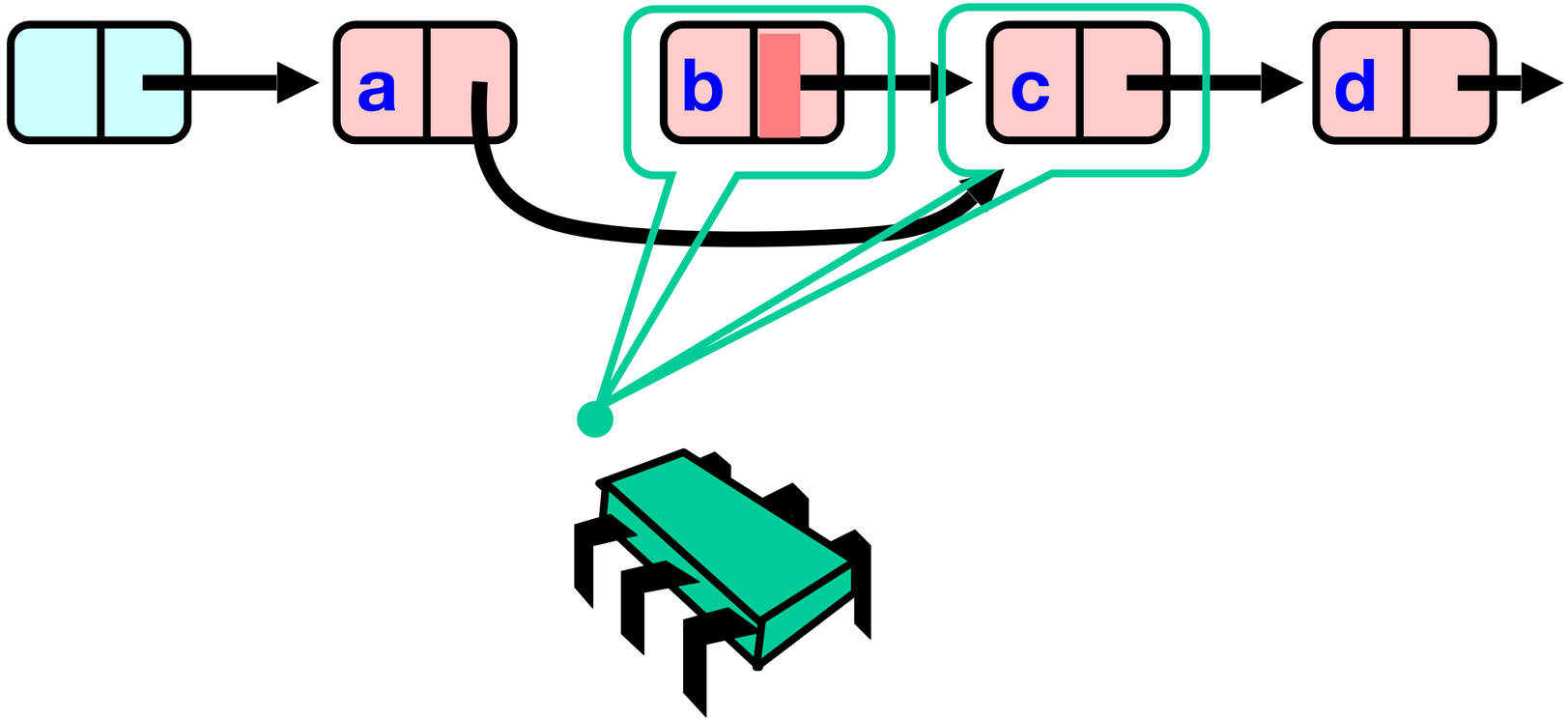
Lock-free traversal (only add and remove)



Lock-free traversal (only add and remove)



Lock-free traversal (only add and remove)



Lock-free Locate function

```
let locate head key =
  let is_marked = ref false in
  let rec retry pred =
    let curr = AMR.get_reference pred.next in
    let rec advance pred curr =
      let succ = AMR.get curr.next is_marked in
      if !is_marked then
        if AMR.compare_and_set pred.next
            ~expected_ref:curr ~new_ref:succ
            ~expected_mark:false ~new_mark:false then
          advance pred succ
        else
          retry head
      else if curr.key >= key then
        (pred, curr)
      else
        advance curr succ
    in
    advance pred curr
  in
  retry head
```

Lock-free Add function

```
let add list item =
  let key = Hashtbl.hash item in
  let rec attempt () =
    let (pred, curr) = locate list.head key in
    if curr.key = key then false
    else
      let node = { item = Some item; key; next = AMR.create curr false } in
      if AMR.compare_and_set pred.next
           ~expected_ref:curr ~new_ref:node
           ~expected_mark:false ~new_mark:false
      then true
      else attempt ()
  in
  attempt ()
```

Lock-free Remove function

```
let remove list item =
  let key = Hashtbl.hash item in
  let rec attempt () =
    let (pred, curr) = locate list.head key in
    if not (curr.key == key) then false
    else
      let succ = AMR.get_reference curr.next in
      if AMR.compare_and_set curr.next
           ~expected_ref:succ ~new_ref:succ
           ~expected_mark:false ~new_mark:true
      then begin
        ignore (AMR.compare_and_set pred.next
                  ~expected_ref:curr ~new_ref:succ
                  ~expected_mark:false ~new_mark:false);
        true
      end else
        attempt ()
  in
  attempt ()
```

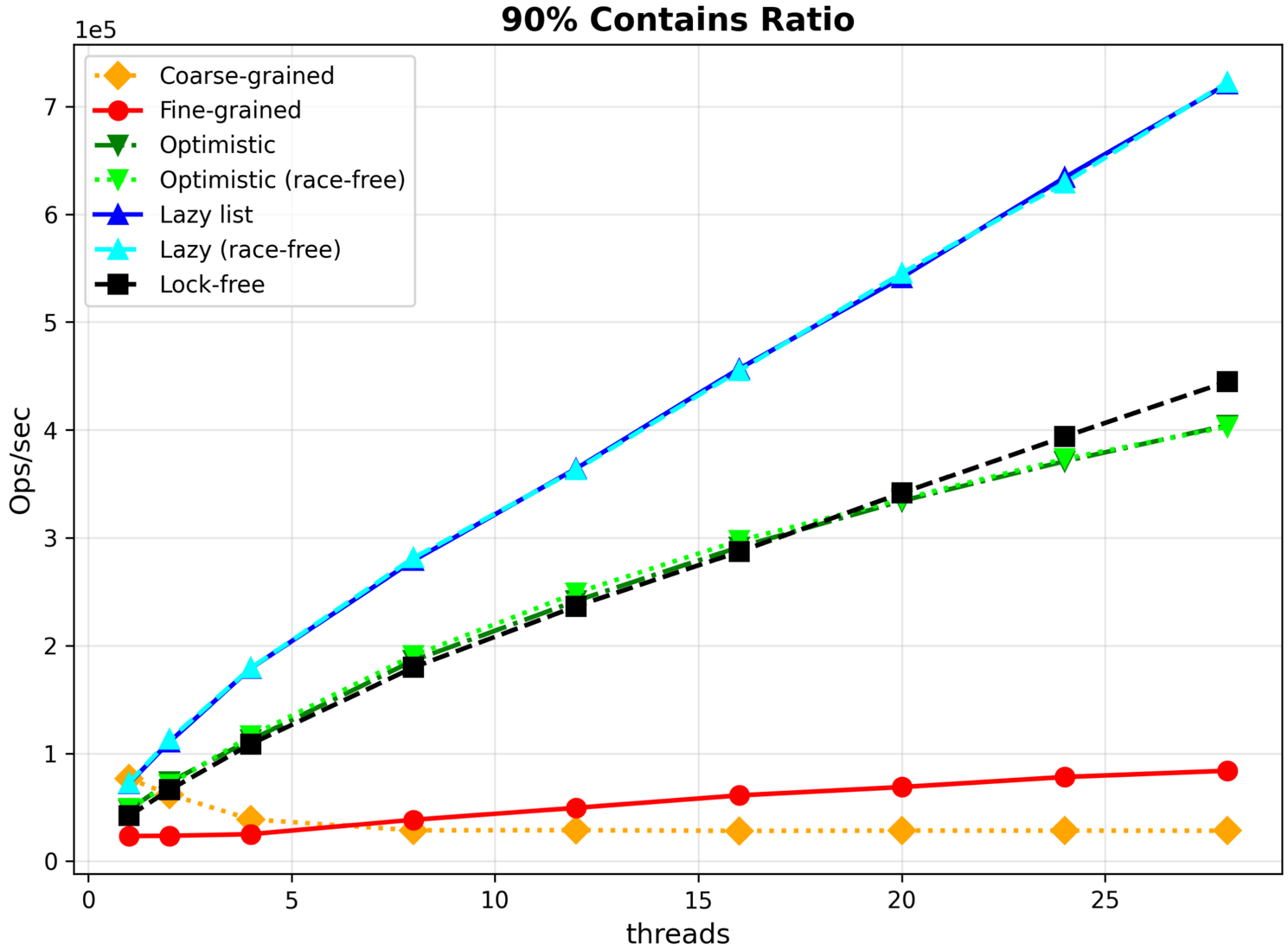
Wait-free Contains function

```
let contains list item =
  let key = Hashtbl.hash item in
  let is_marked = ref false in
  let rec loop curr =
    let next_node = AMR.get curr.next is_marked in
    if !is_marked then
      (* Skip marked nodes *)
      loop next_node
    else if curr.key < key then
      loop next_node
    else
      curr.key = key
  in
  loop list.head
```

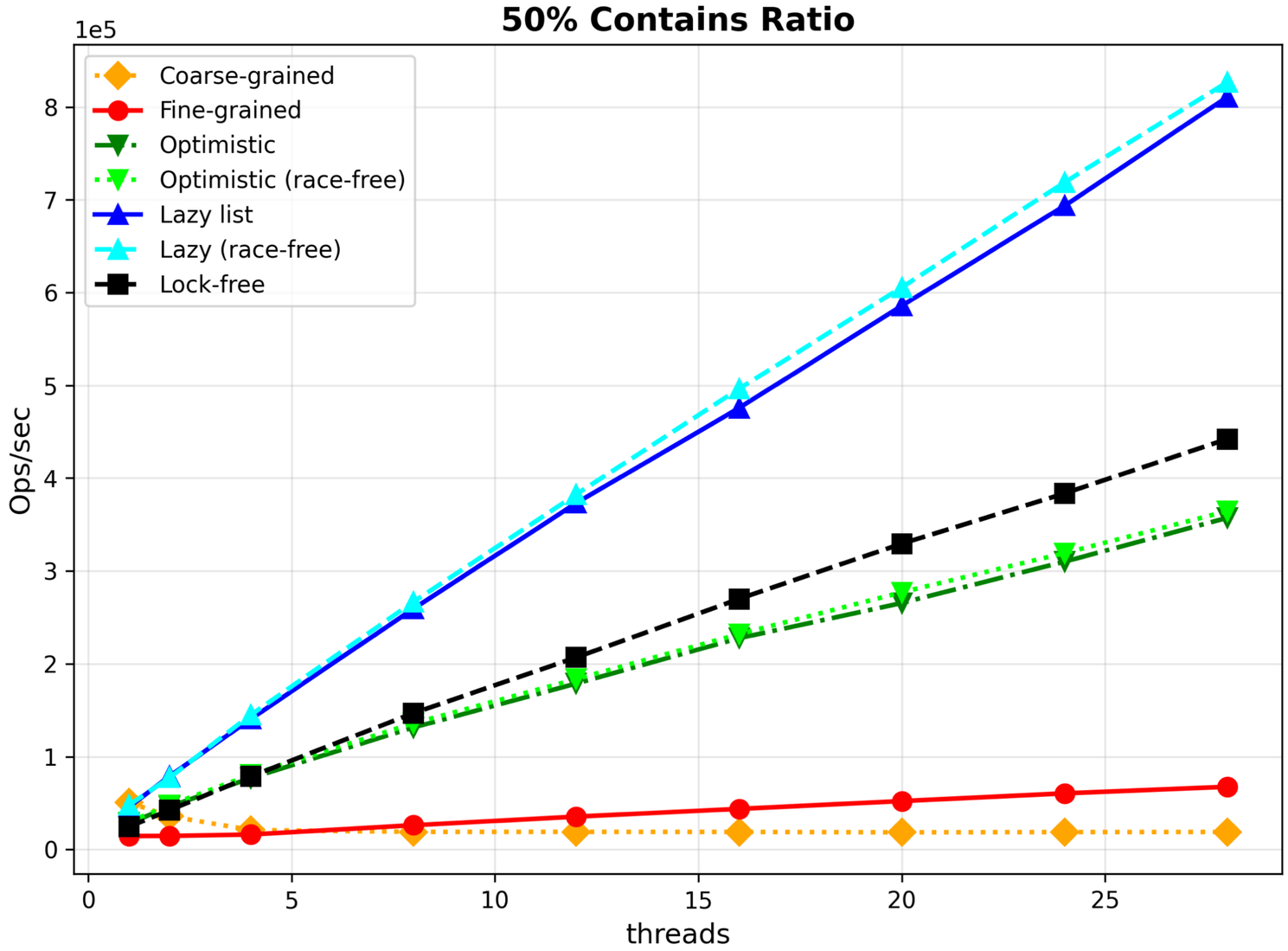
Performance

- Different list-based set implementaions
- 28-core machine
- Vary percentage of **`contains()`** calls

High `contains()` ratio



Low `contains()` ratio



Summary

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

“To Lock or Not to Lock”

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