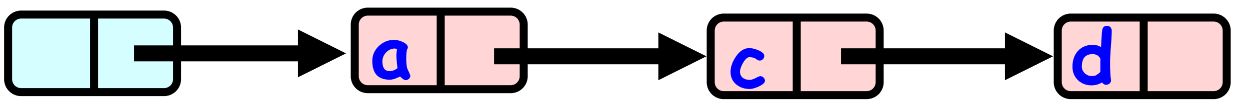
Linked Lists:
Locking vs. Lock-Free

Concurrent Algorithms 2013

Programming Assignment

Linked list

- Data structure with group of nodes
  - representing a sequence



- Operations
  - add()
  - remove()
  - contains()

Task

- Implement 2 versions of a linked list
    - lock-based
    - lock-free
- The algorithms are given
    - design is tough
    - implementation can also be tricky

Deliverables

- An archive with your code
- A short report

- Deadline (strict)

  Monday, December 16th, 23:59

Skeleton Code in C

- Benchmarking code: do NOT change it
- Scripts
  - test correctness
  - execute experiments
  - print graphs
- See README (or ca_prog_assignment.pdf)
- If C is a problem, contact the TAs

Programmer's Toolbox

- Registers:
  - Shared memory locations
- Atomic Operations:
  - Fetch-and-Add
  - Test-and-Set
  - Compare-and-Swap
  - Provided in atomic_ops.h
- Use them to build concurrent objects

Atomic Operations in Practice

- Example: CAS based lock:

```
void lock(lock_t* lock) {
  while (CAS(lock,0,1)==1) {}
}
void unlock(lock_t* lock) {
    *lock = 0;
}
```

Linked Lists: Locking vs. Lock-Free

Original slides
by Maurice Herlihy & Nir Shavit

Outline

- **Lock-free linked list**
- Lock-based linked list

Linked List

- Using a list-based Set
  - Common application
  - Building block for other apps

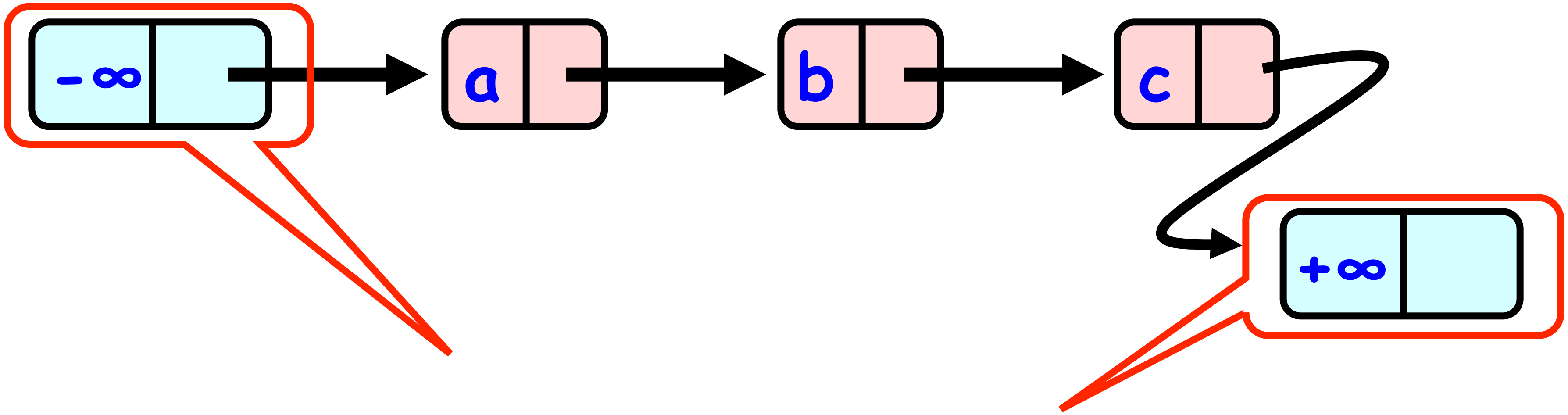
Set Interface

- 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 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)

Reminder: 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

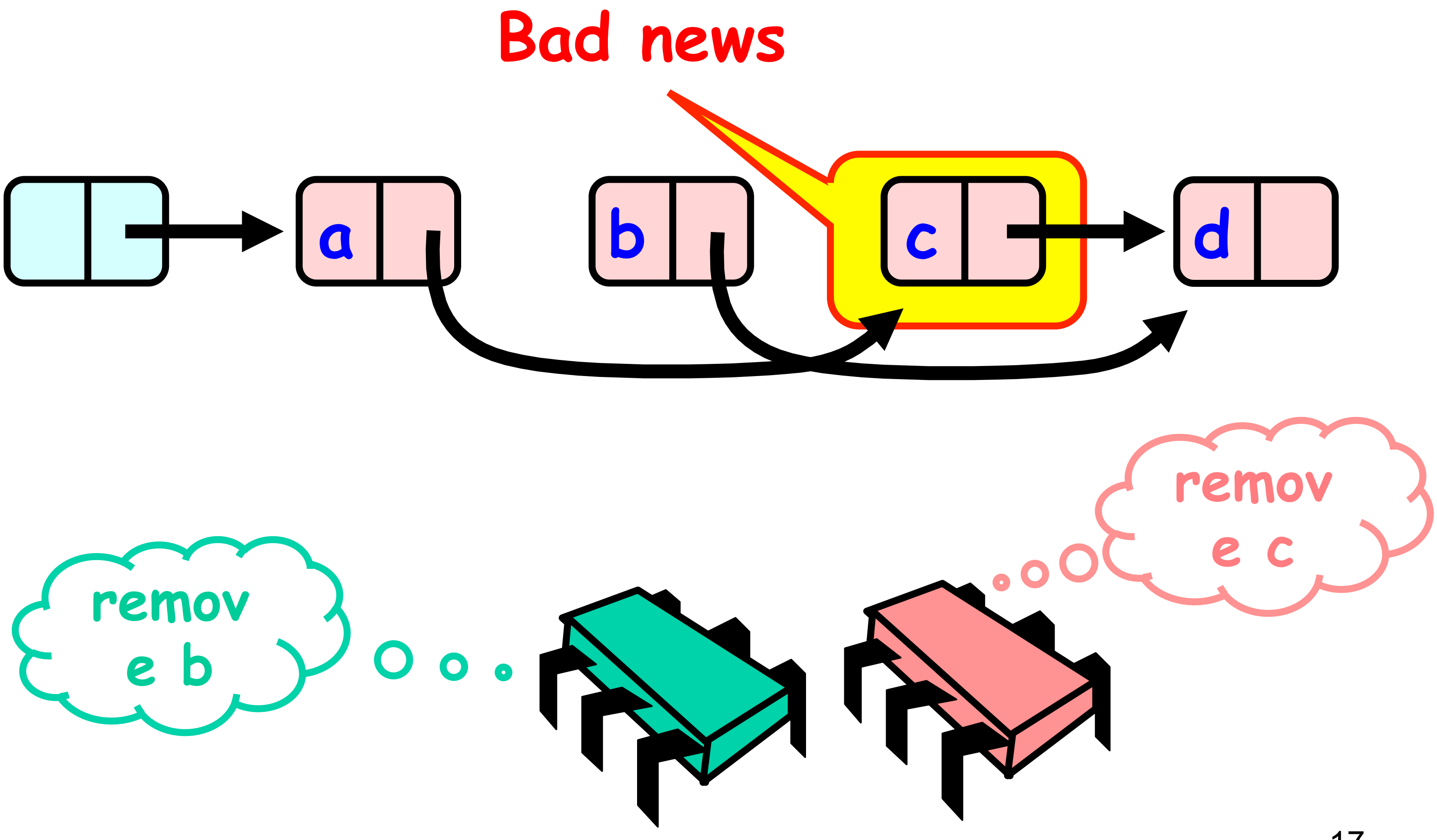
Why lock-free?

- 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 Lists

- Eliminate locking entirely
- contains() wait-free and add() and remove() lock-free
- Use only compareAndSwap()

Problem



Problem

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

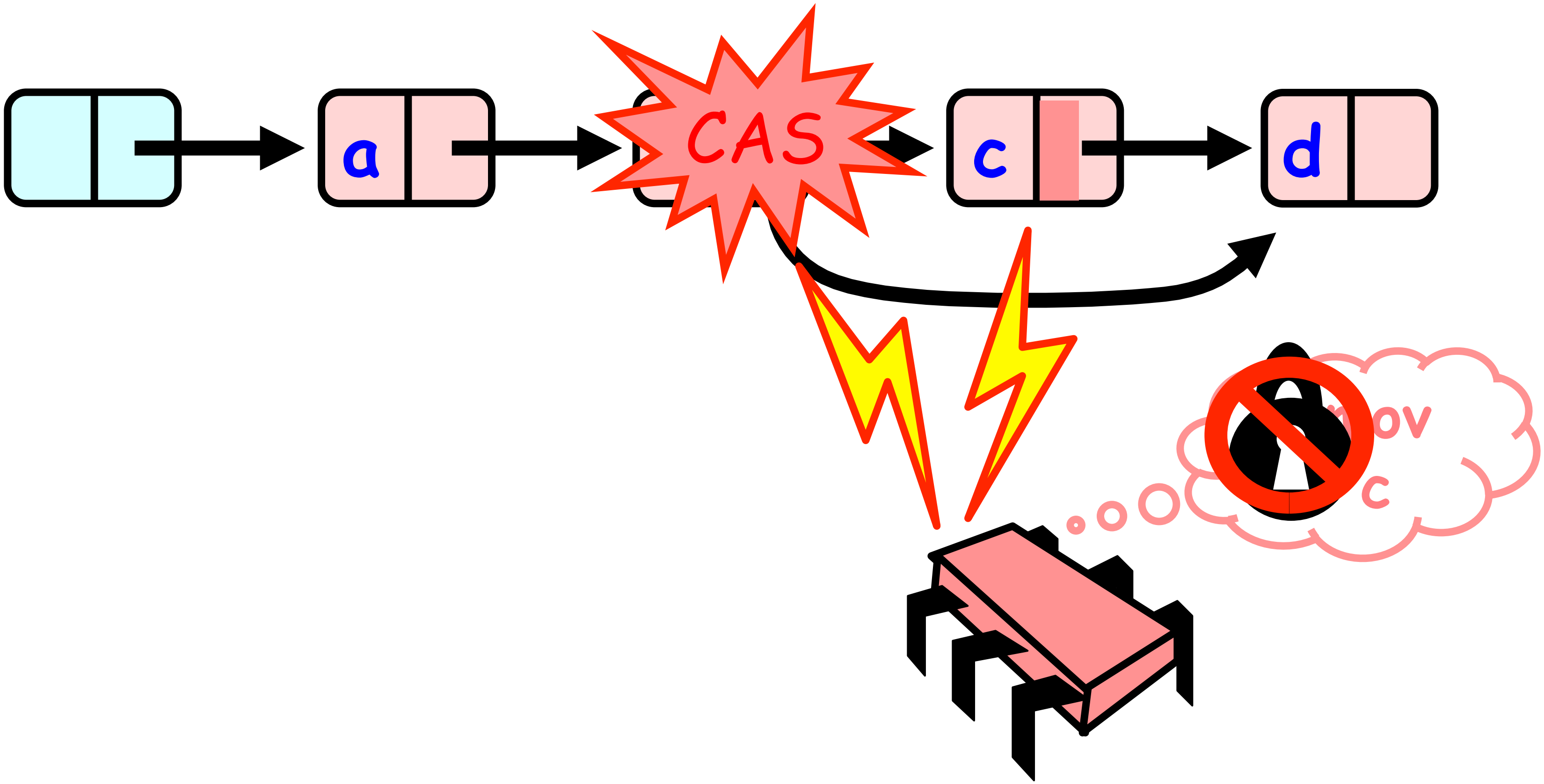
Solution

- Use 1 bit to signify removal
- Atomically
  - Swing reference and
  - Update flag
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor's pointer

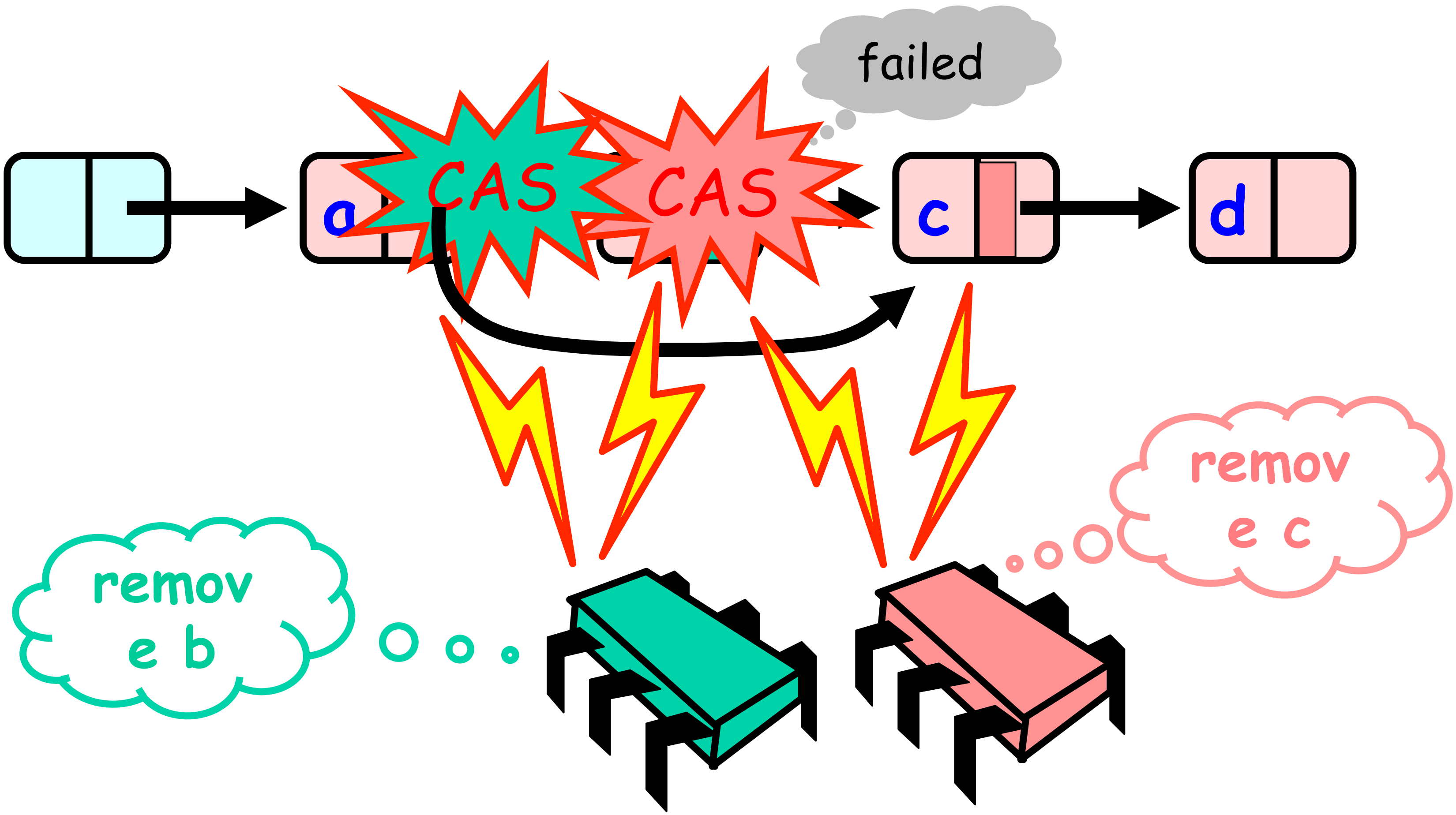
Logical vs. Physical Deletion

- Logical delete
  - Marks current node as removed
- Physical delete
  - Redirects predecessor's next

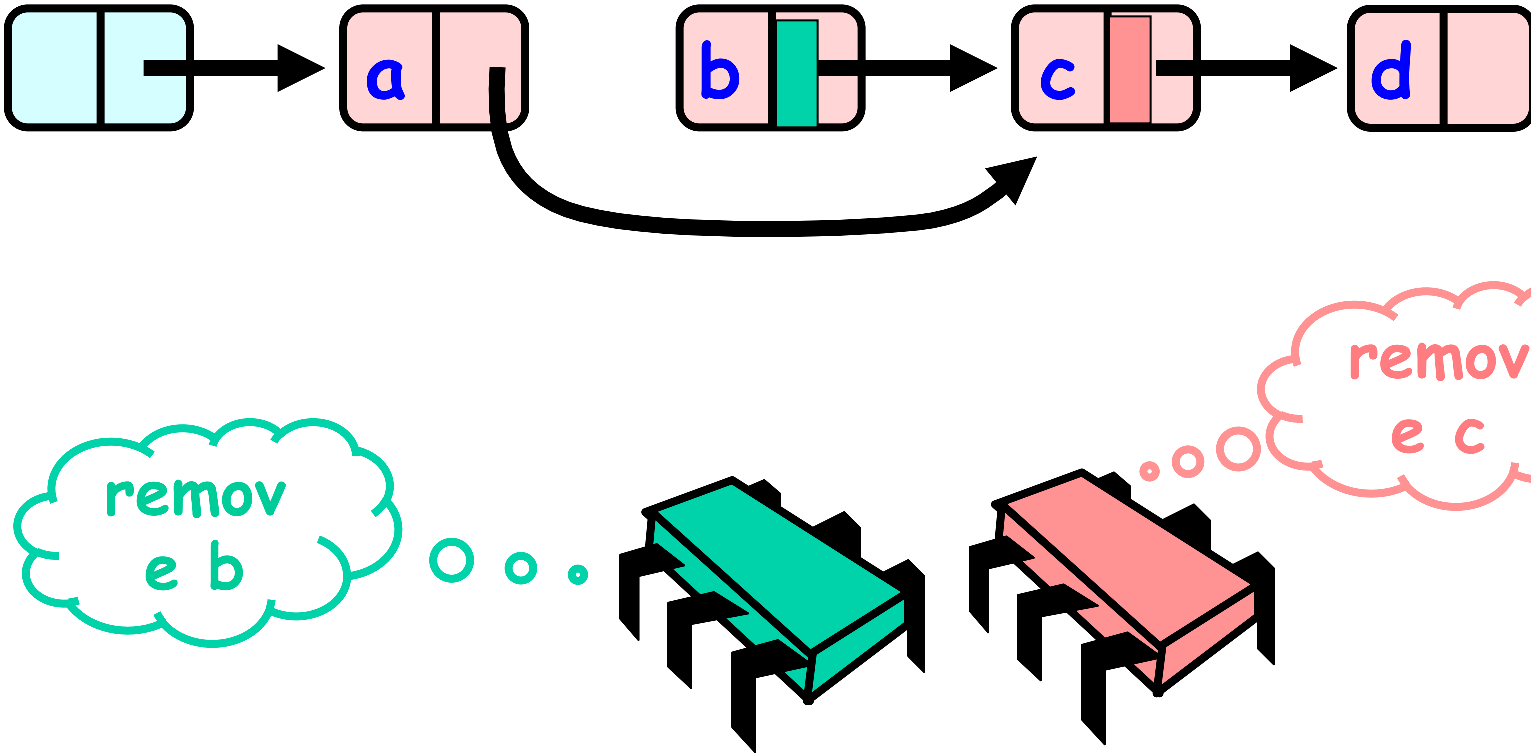
Removing a Node



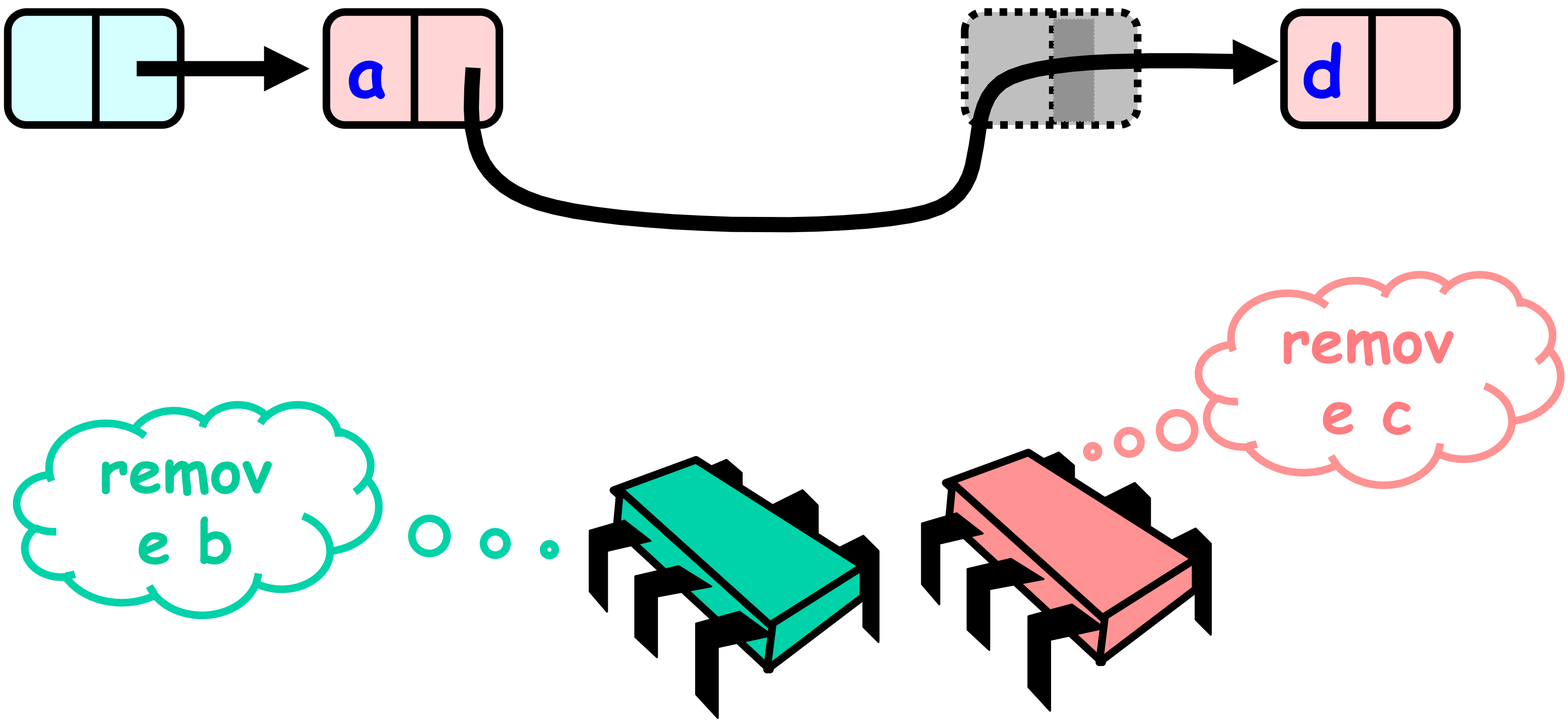
Removing a Node



Removing a Node



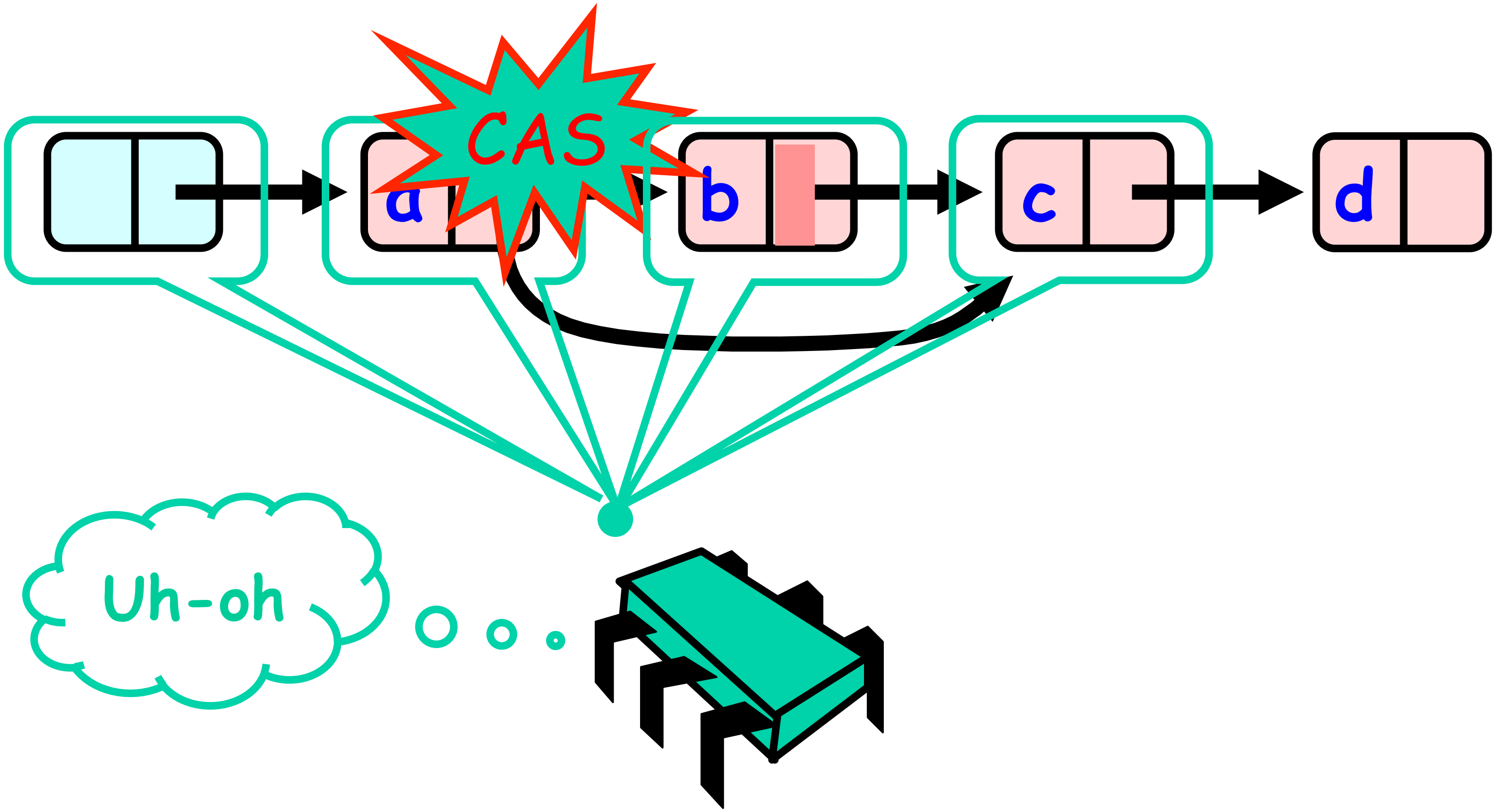
Removing a Node



Traversing the List

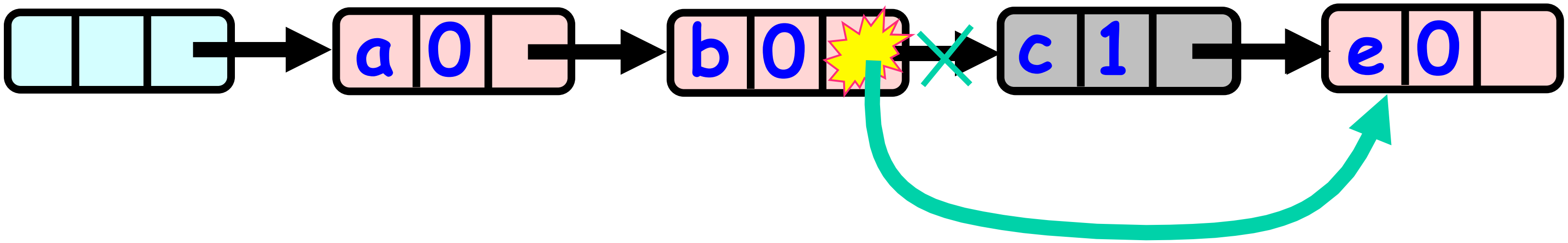
- Q: what do you do when you find a "logically" deleted node in your path?
- A: finish the job.
  - CAS the predecessor's next field
  - Proceed (repeat as needed)

Lock-Free Traversal



Summary: Lock-free Removal

Logical Removal = Set Mark Bit



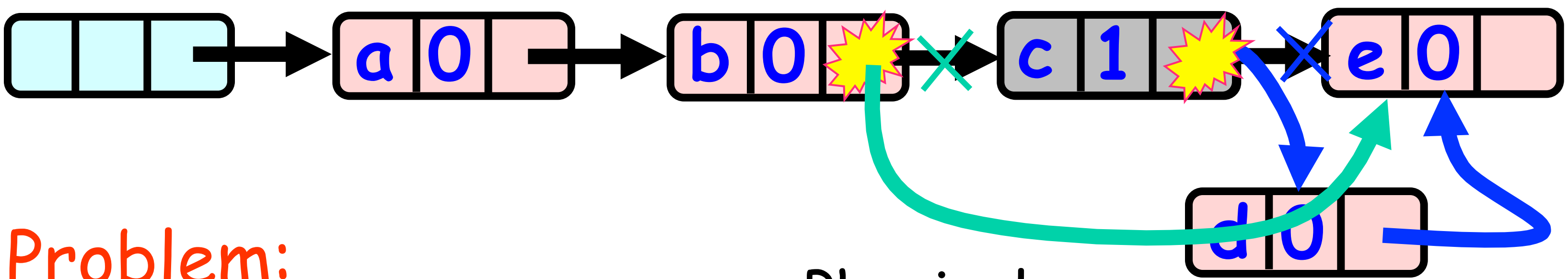
Physical Removal CAS pointer

Use CAS to verify pointer is correct

Not enough!

Lock-free Removal

Logical Removal =
Set Mark Bit



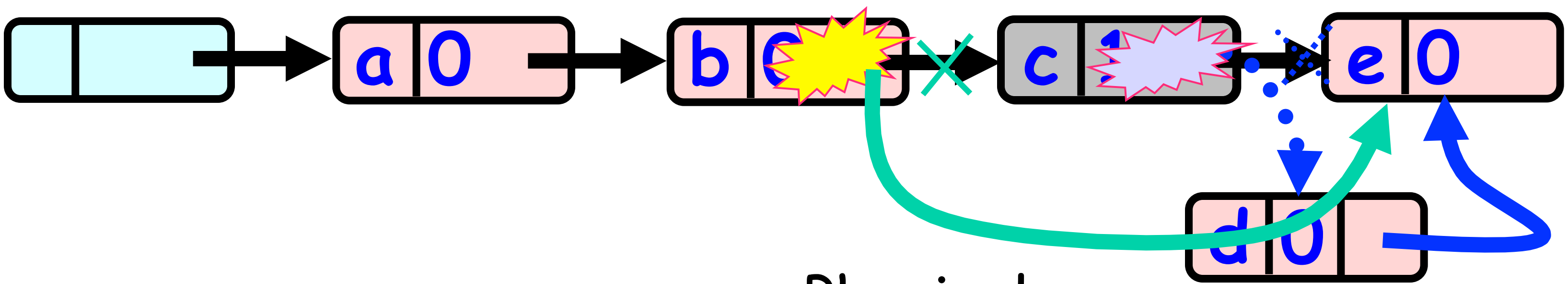
Problem:
d not added to list…
Must Prevent
manipulation of
removed node's pointer

Physical
Removal
CAS

Node added
Before
Physical
Removal CAS

Our Solution: Combine Bit and Pointer

Logical Removal = Set Mark Bit

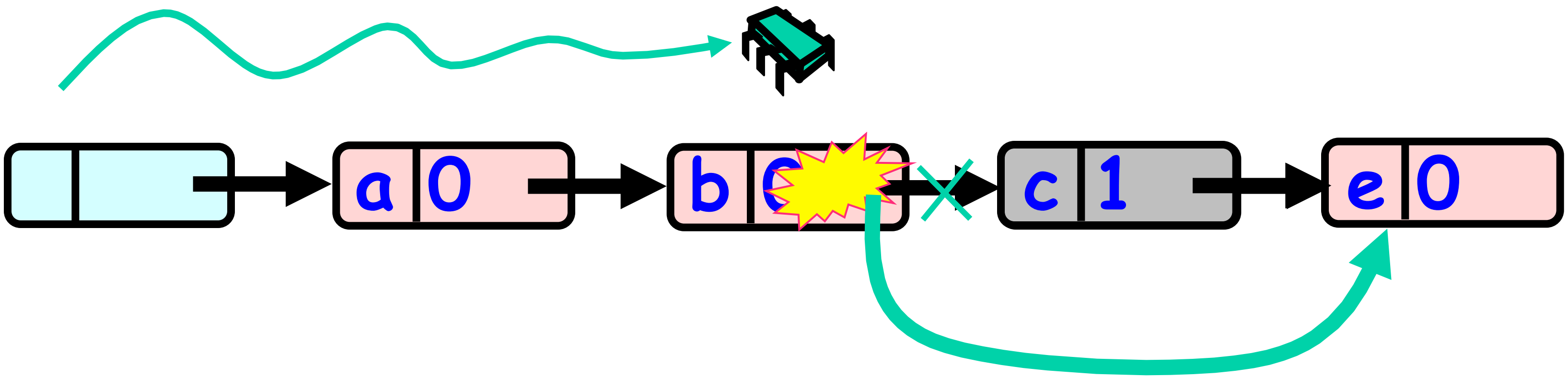


Physical Removal CAS

Fail CAS: Node not added after logical Removal

Mark-Bit and Pointer are CASed together

A Lock-free Algorithm



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

Outline

- Lock-free linked list
- **Lock-based linked list**

Locks

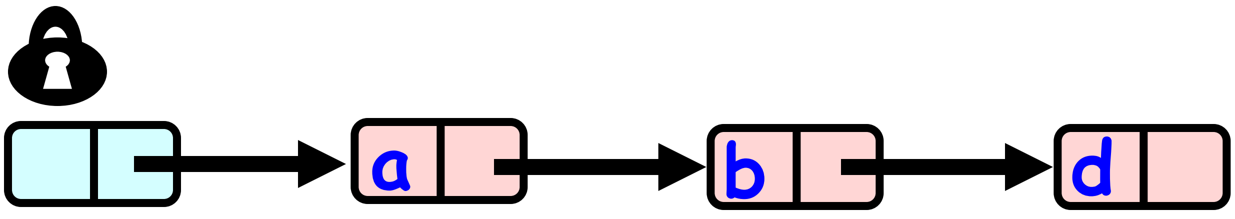
- Used to ensure mutual exclusion in critical sections
- 2 methods:
  - acquire()
  - release()
- Many algorithms to implement locks

What about lock-based algorithms?

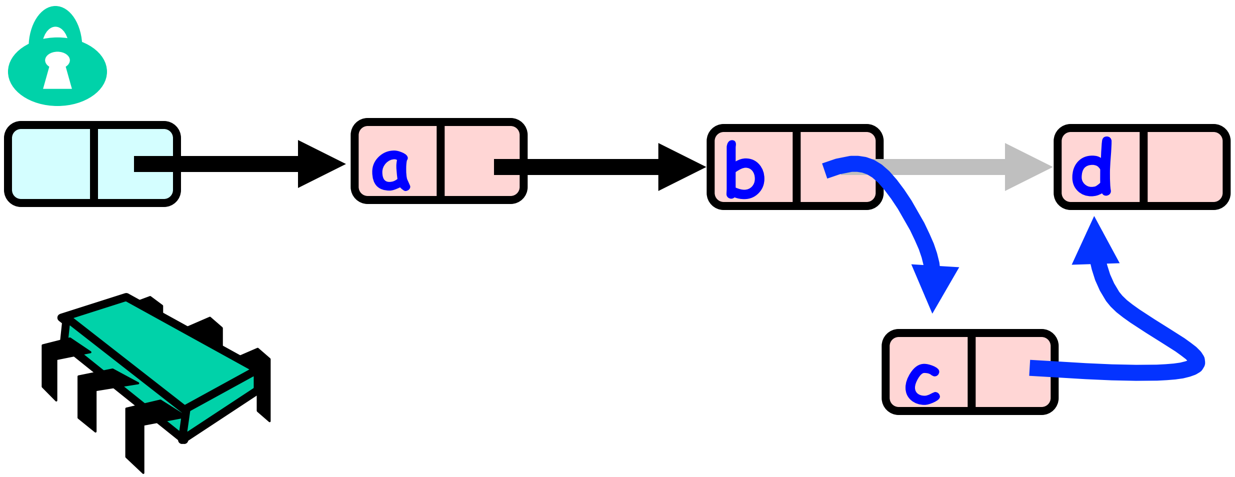
- Generally easier to design
- In many cases simpler code
- May be faster?

- However
  - Deadlocks etc.

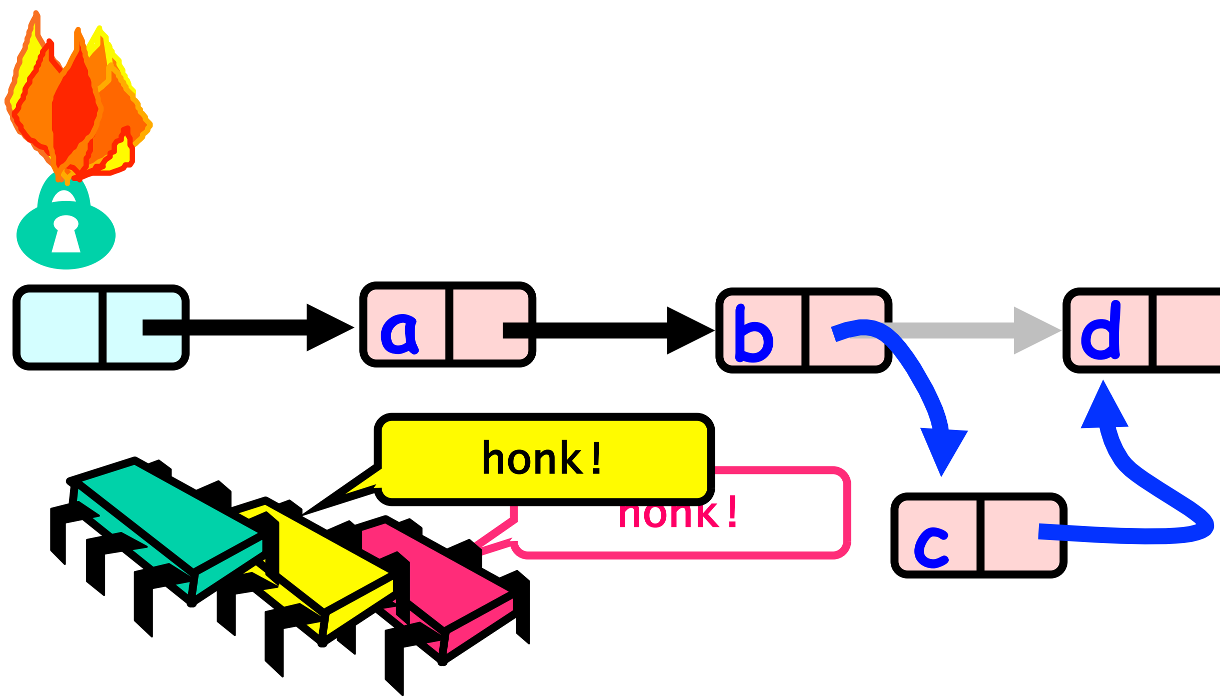
Coarse Grained Locking



Coarse Grained Locking



Coarse Grained Locking



Simple but hotspot + bottleneck

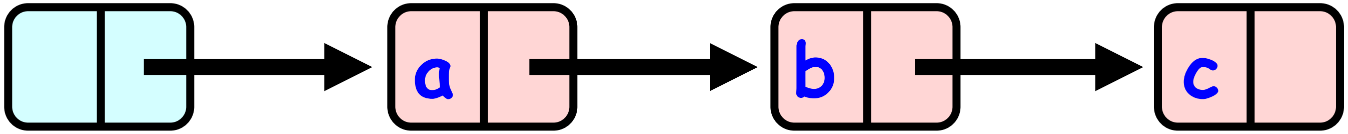
Coarse-Grained Locking

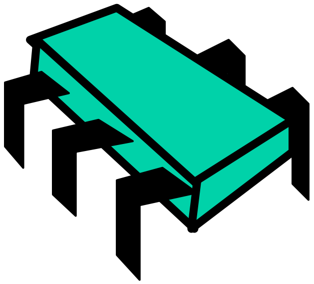
- Easy, same as synchronized methods
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention
  - Queue locks help
  - But bottleneck still an issue

Fine-grained Locking

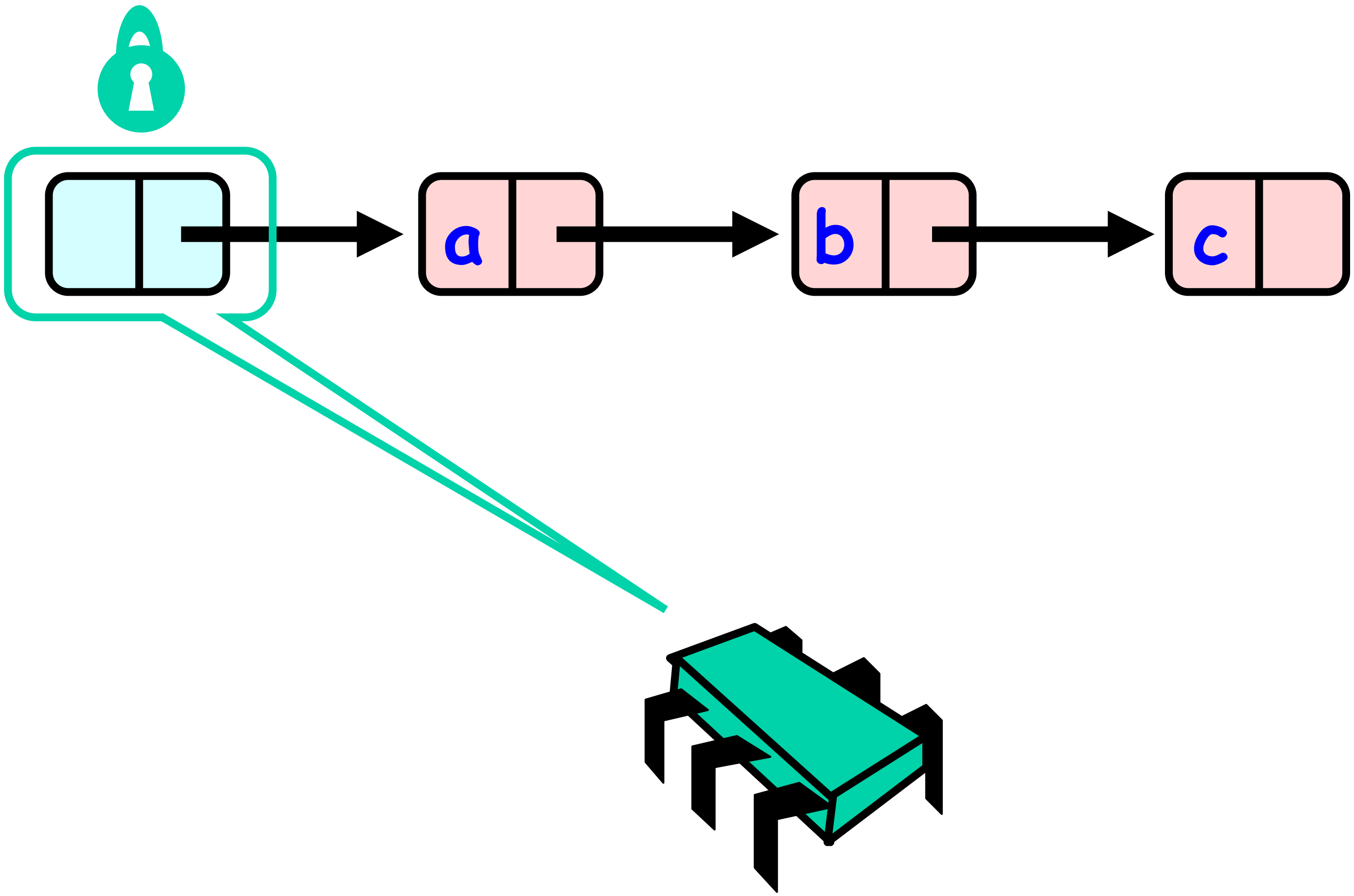
- Requires **careful** thought
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand locking

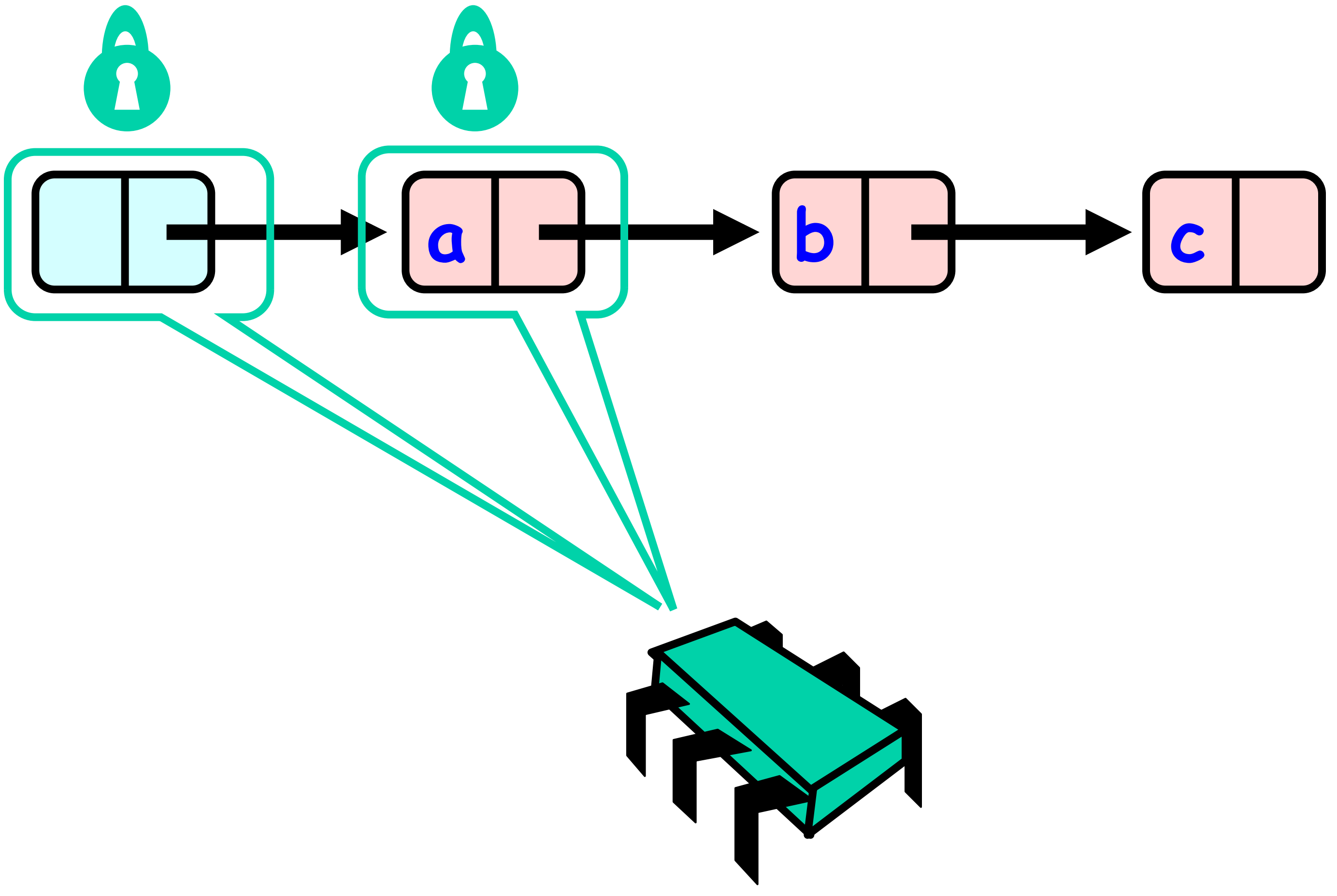




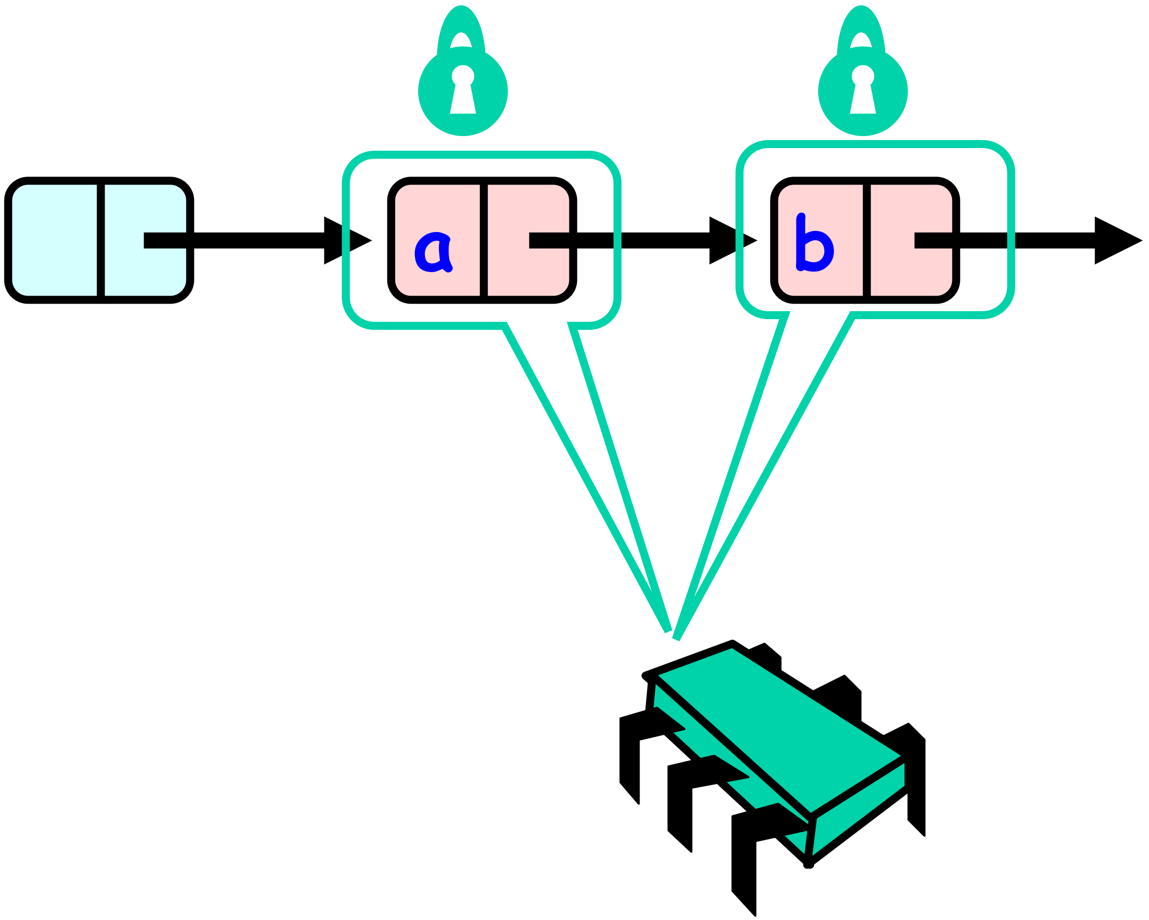
Hand-over-Hand locking



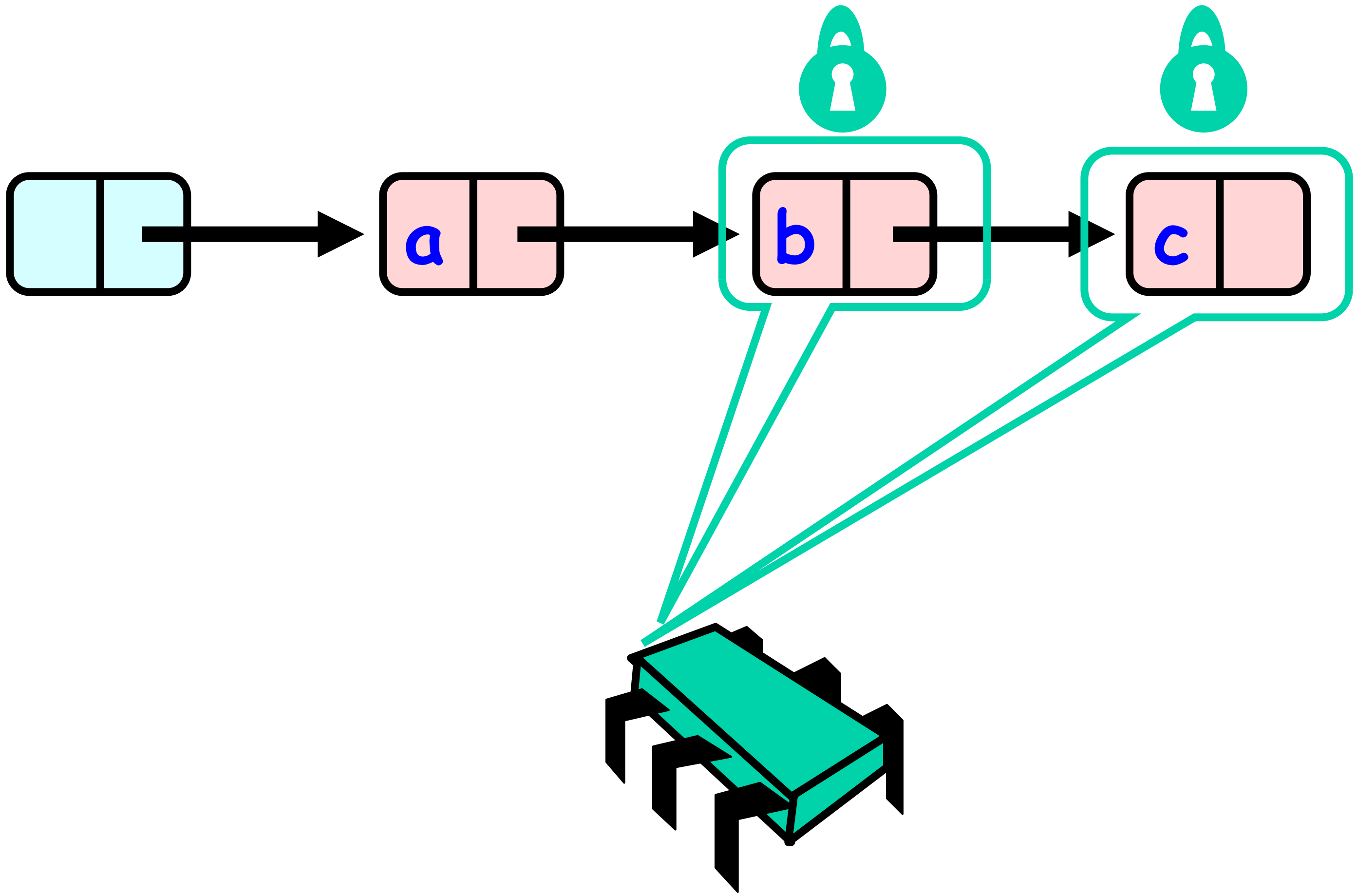
Hand-over-Hand locking



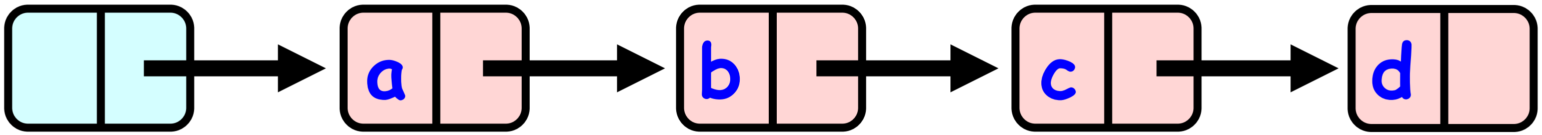
Hand-over-Hand locking

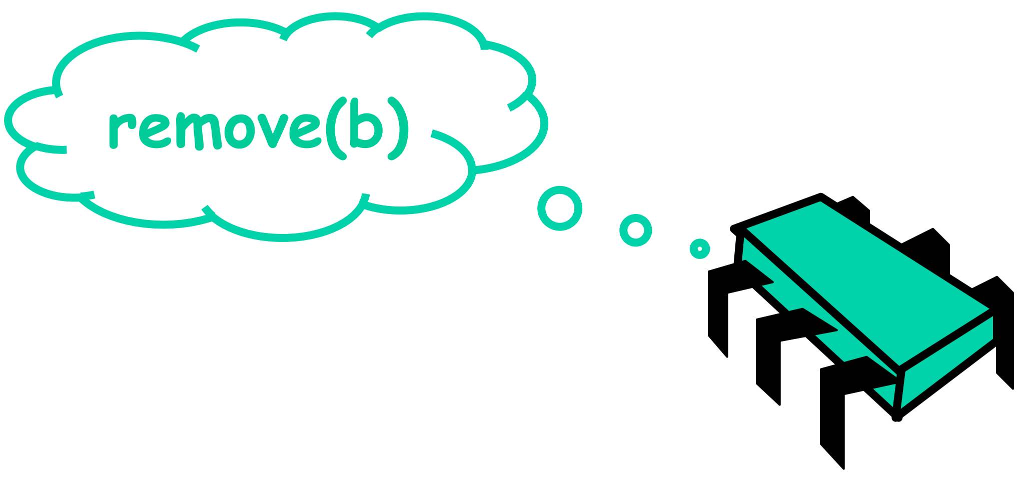


Hand-over-Hand locking

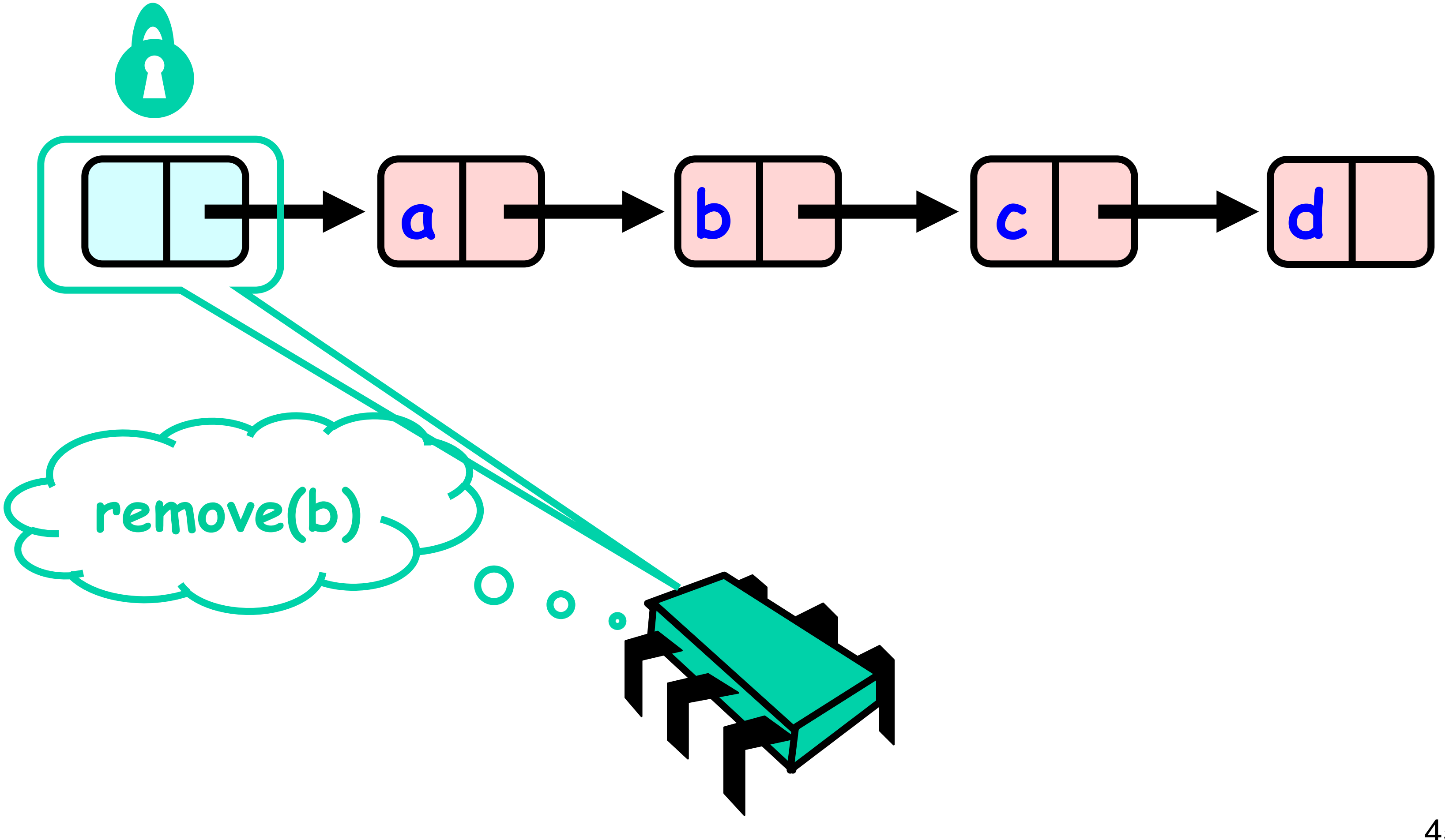


Removing a Node

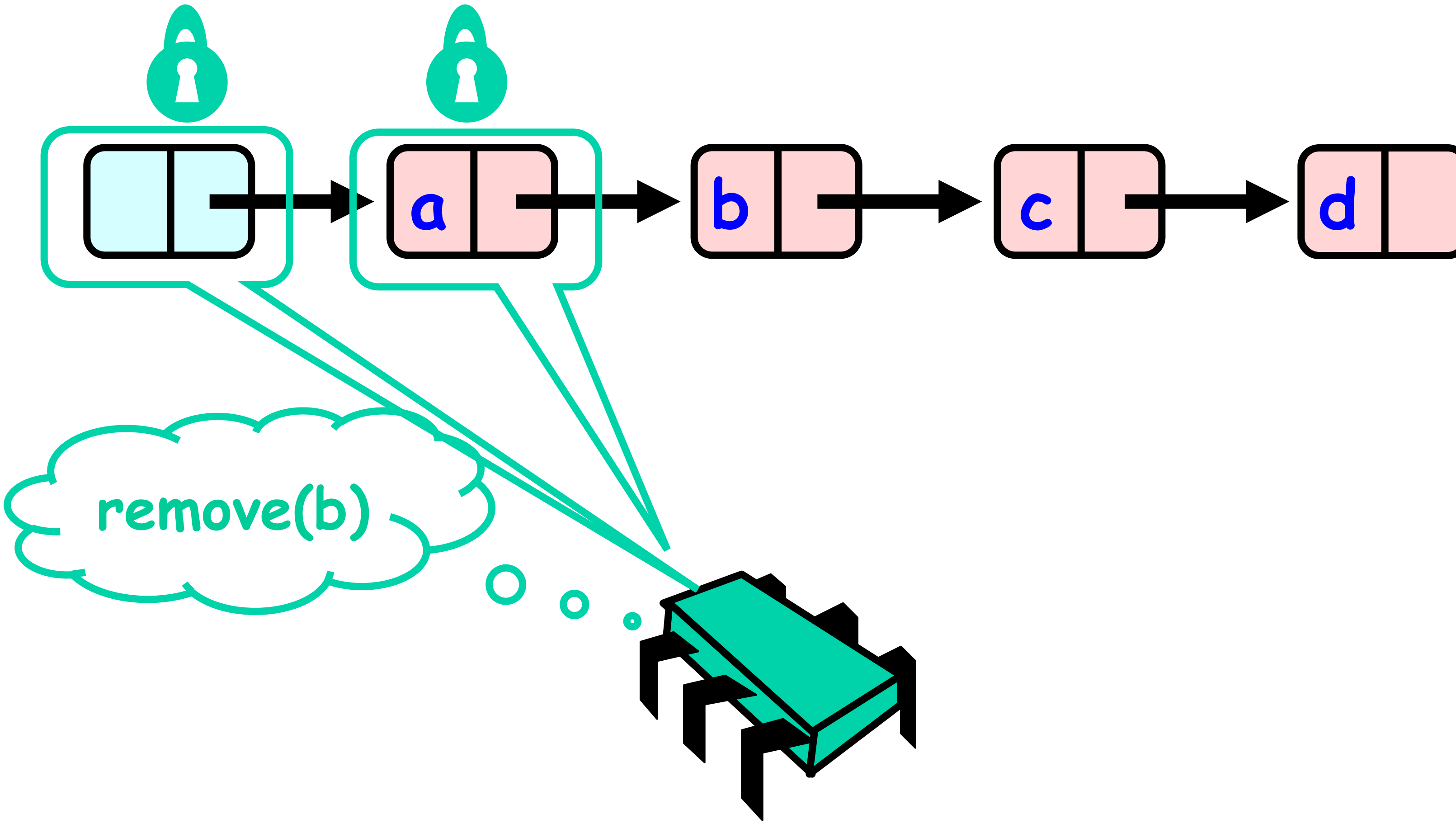




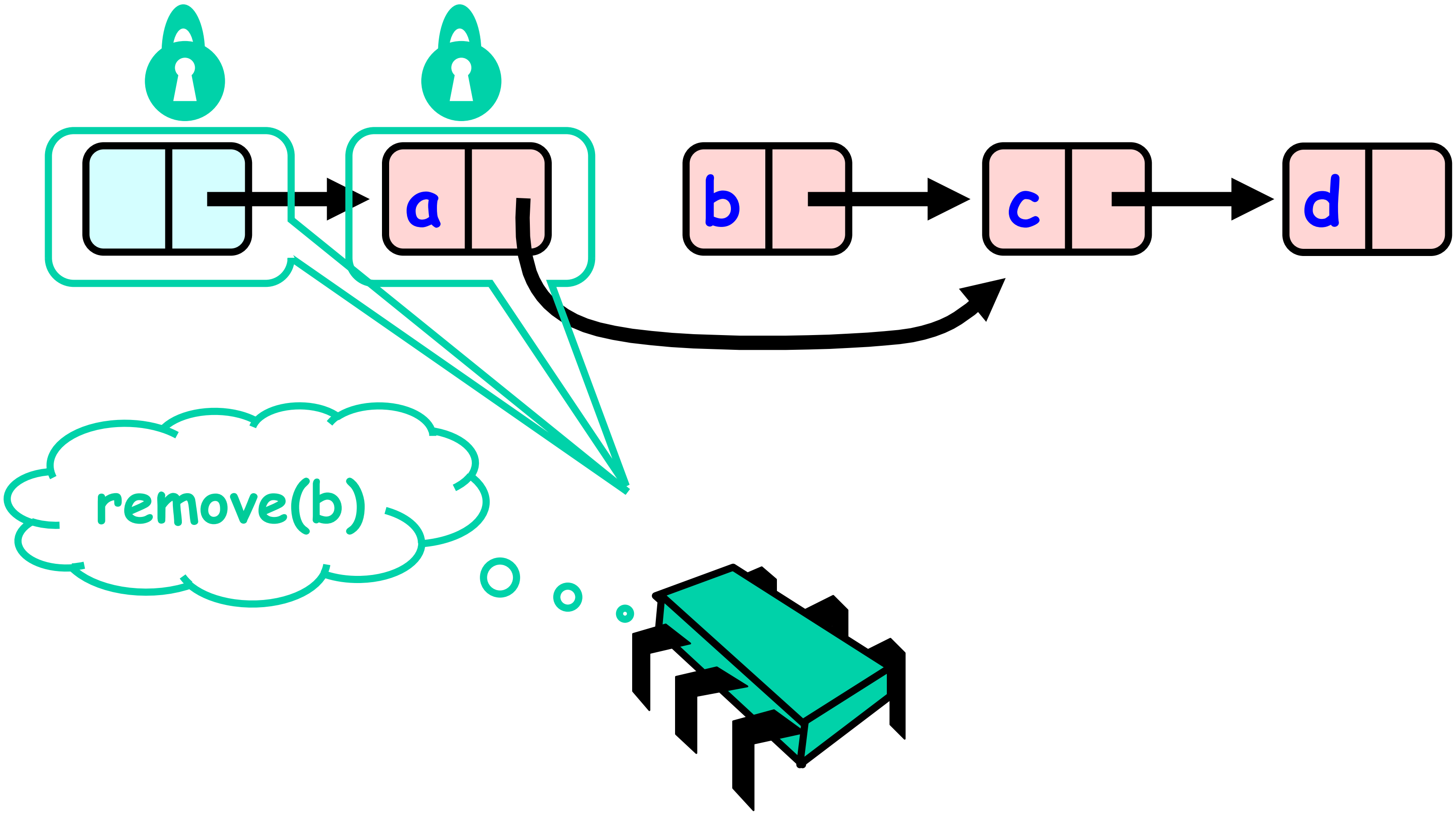
Removing a Node



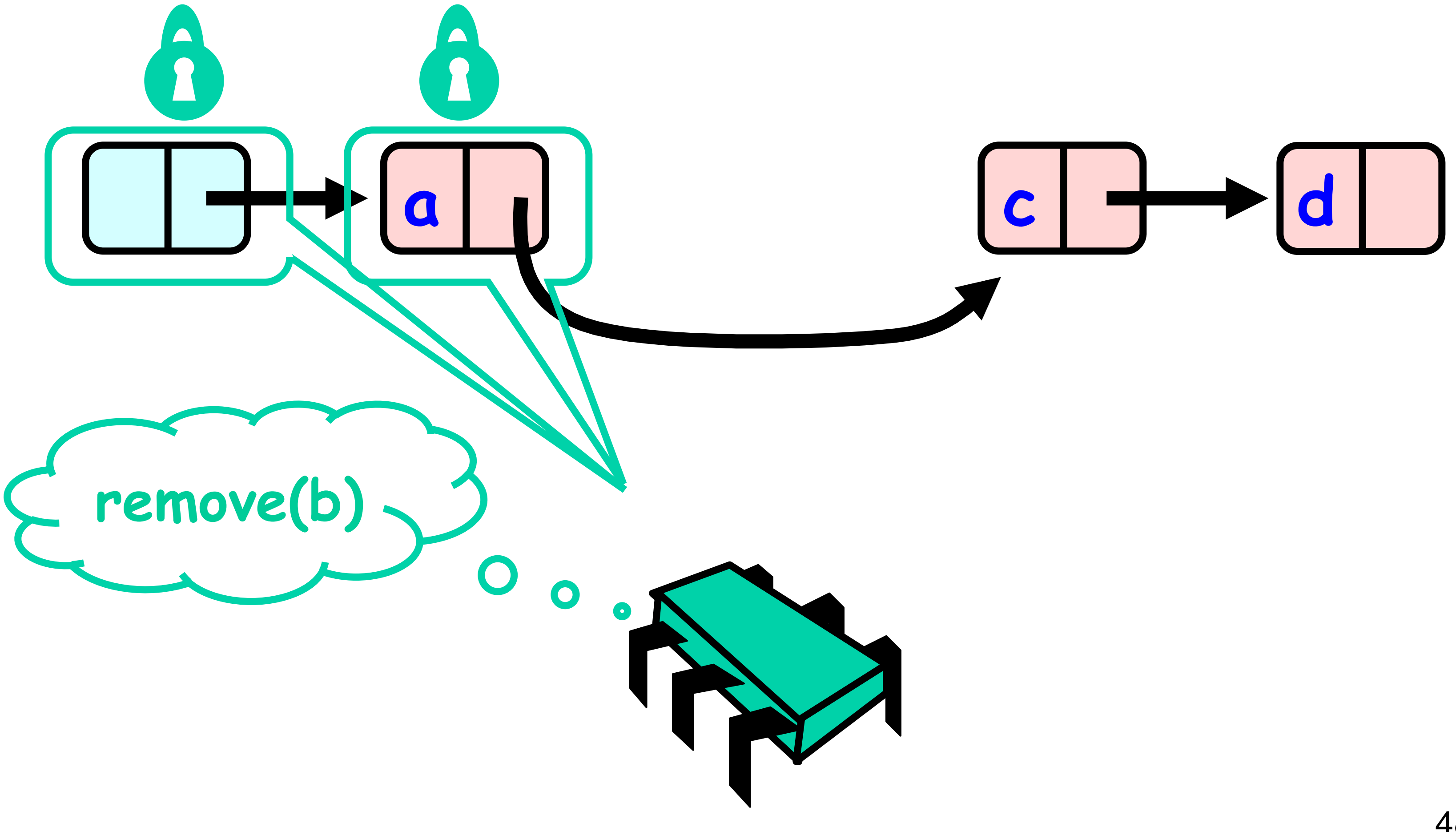
Removing a Node



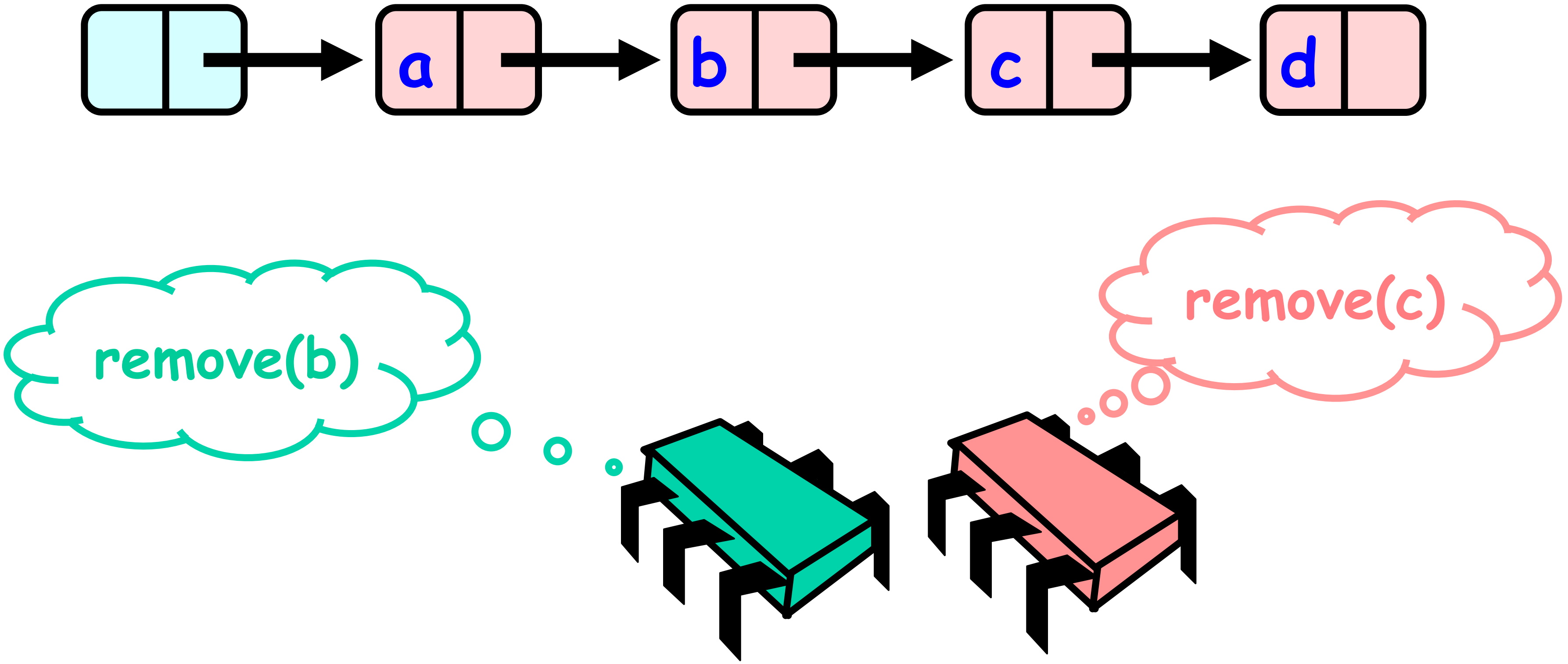
Removing a Node



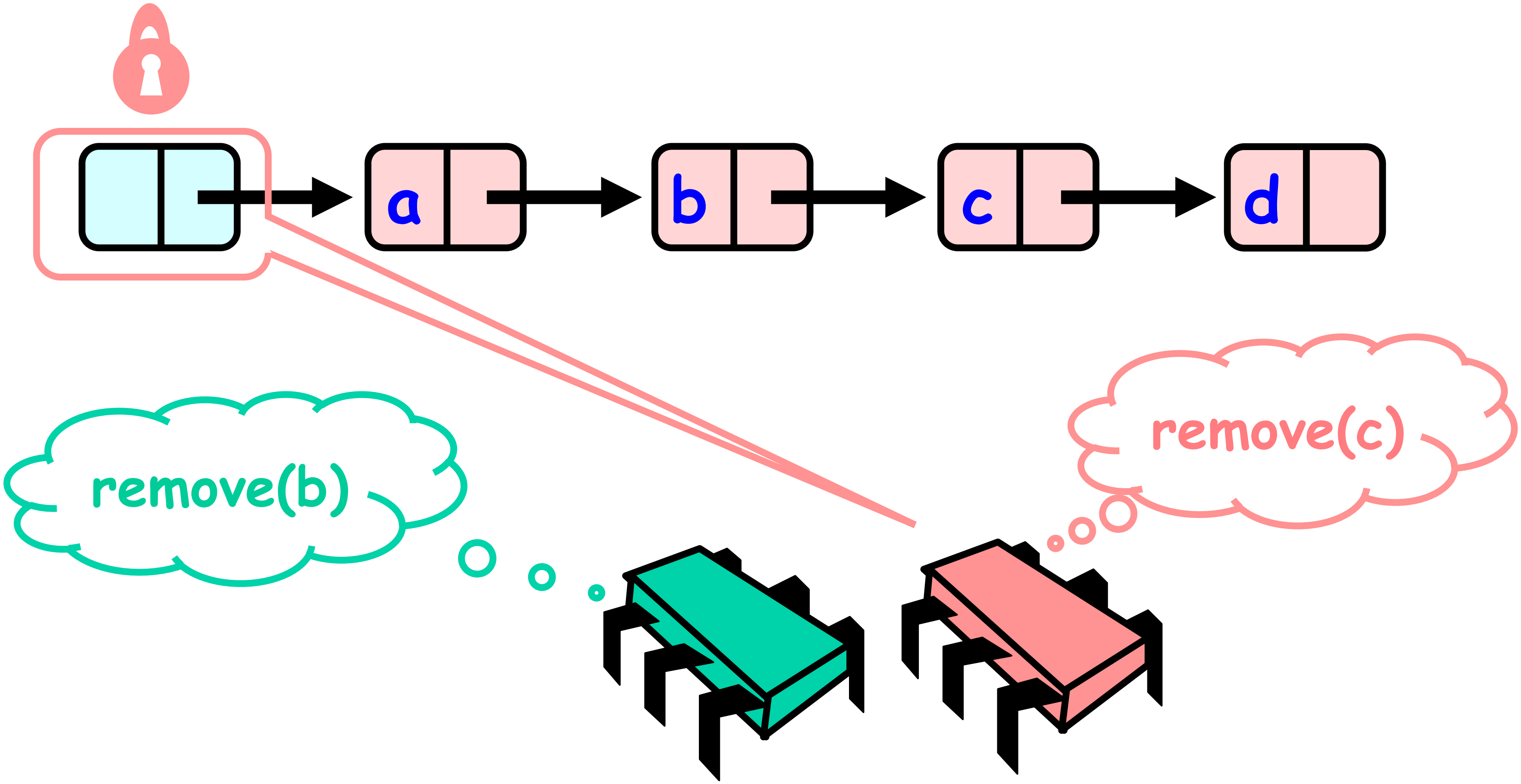
Removing a Node



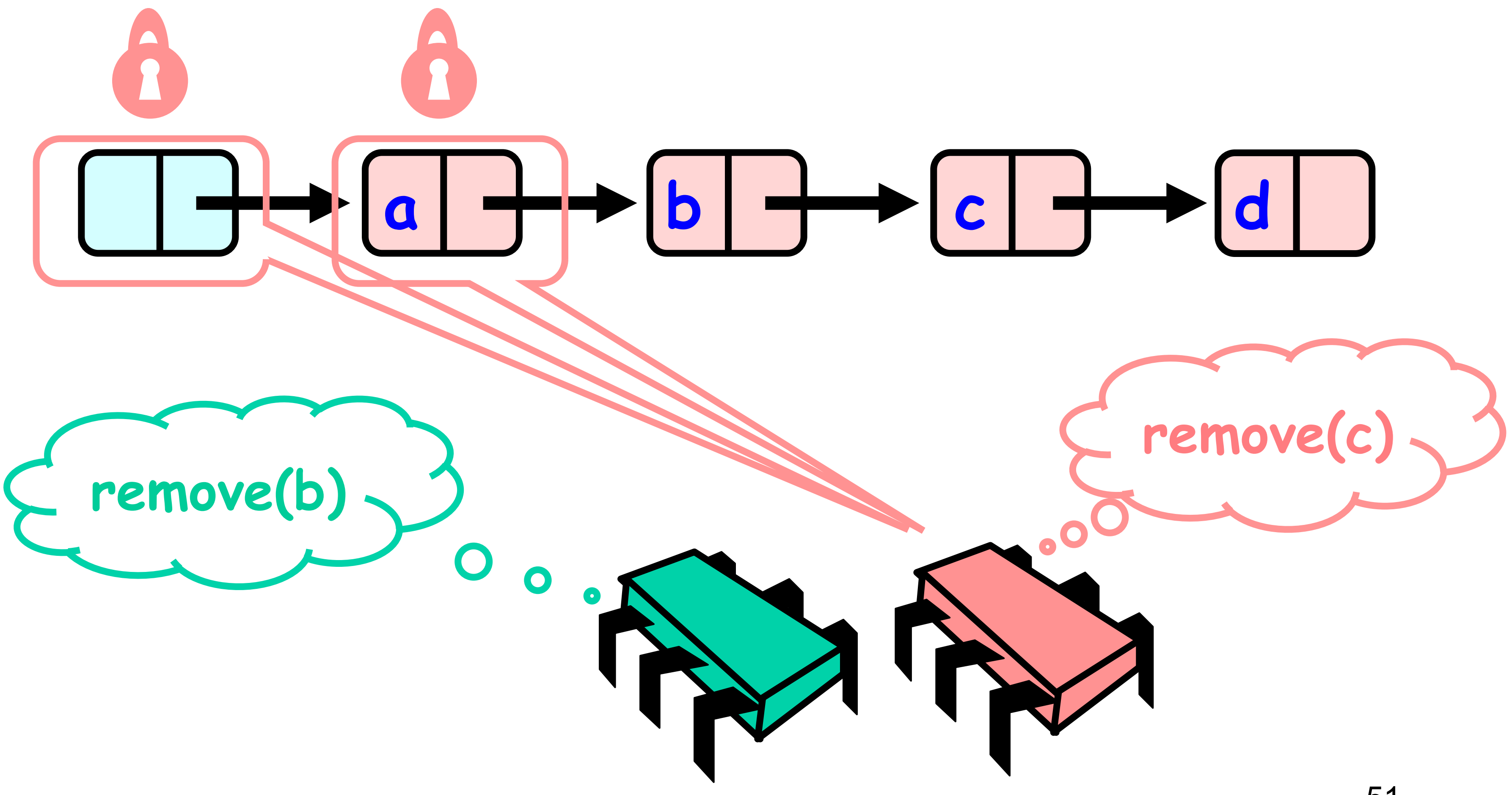
Removing a Node



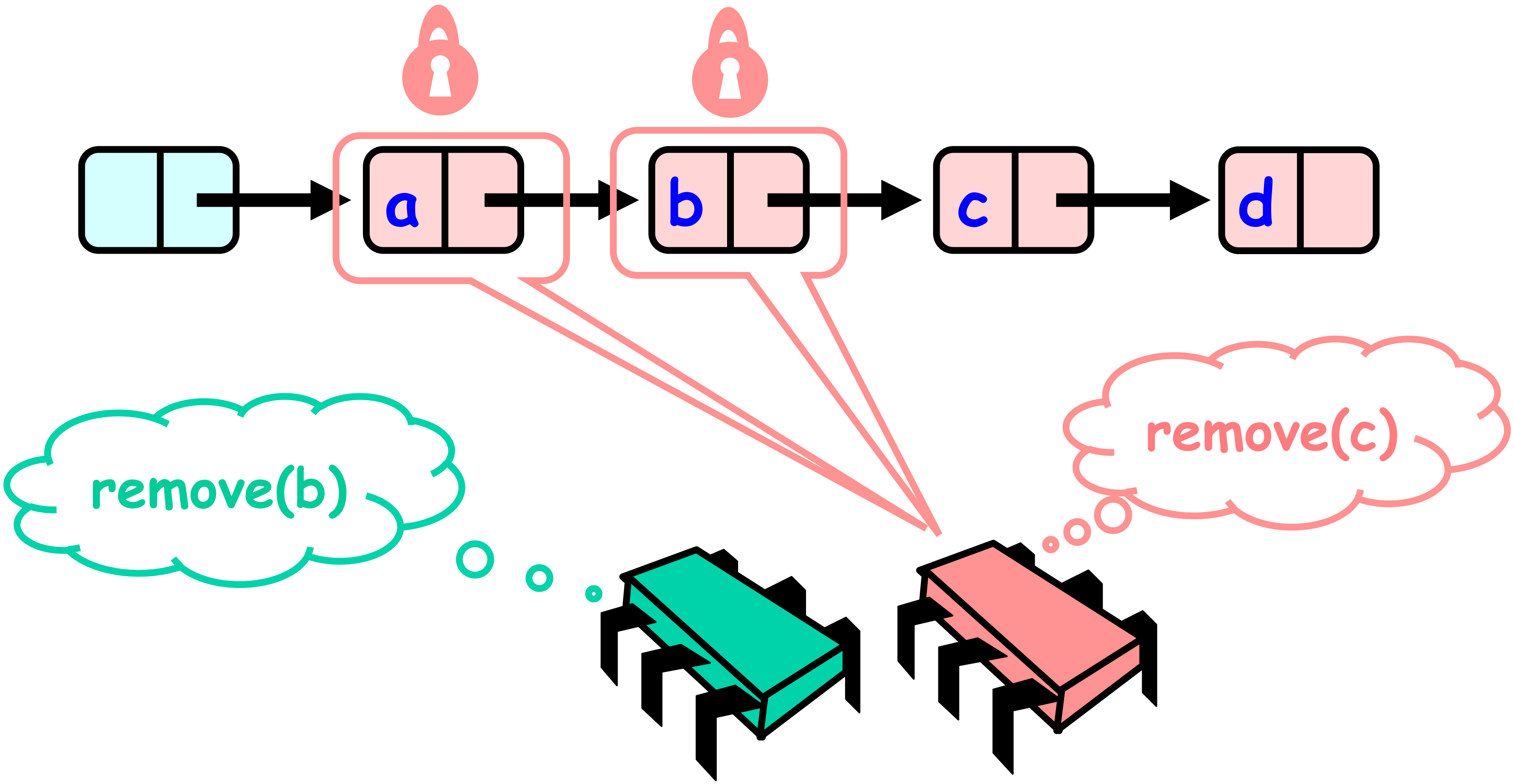
Removing a Node



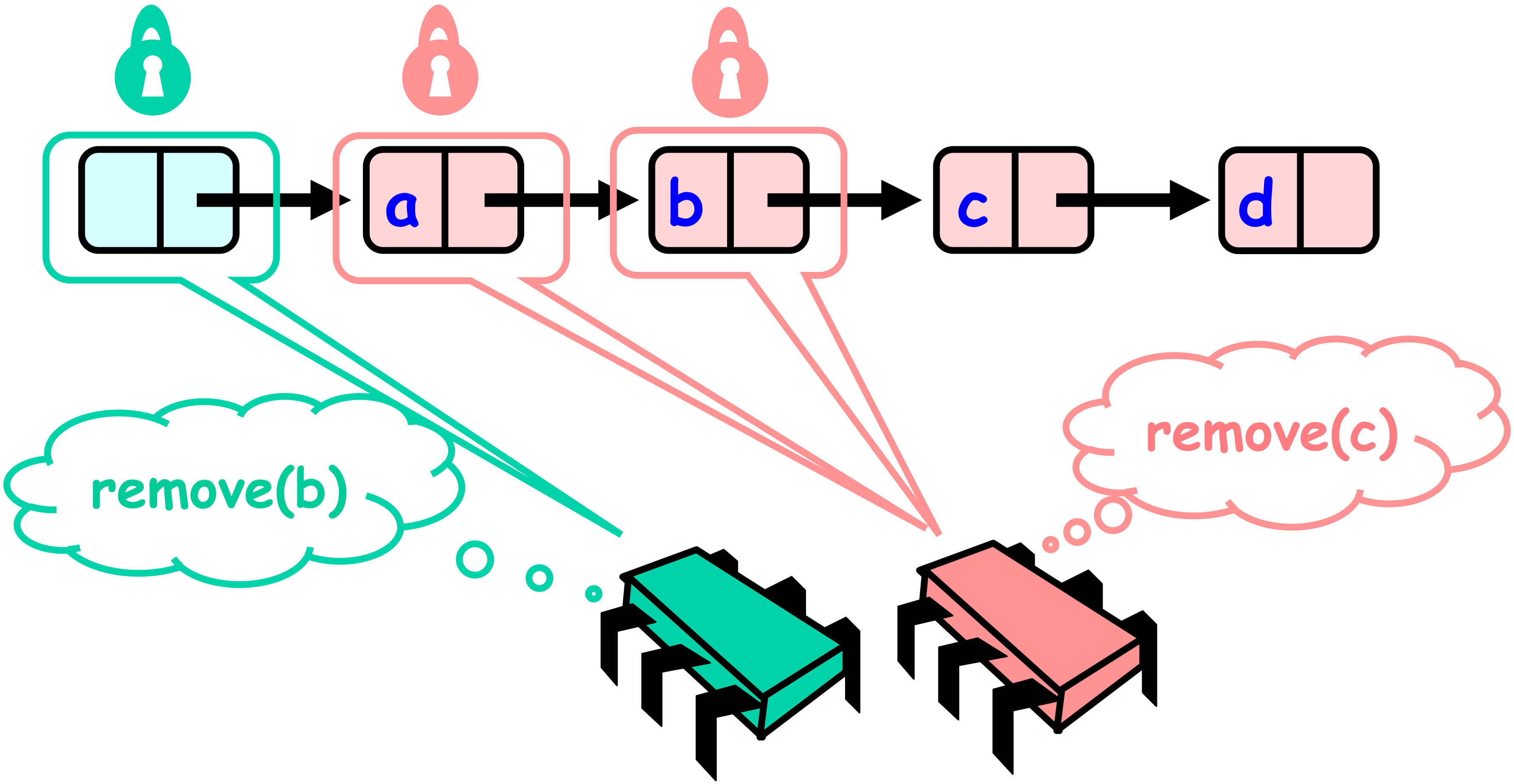
Removing a Node



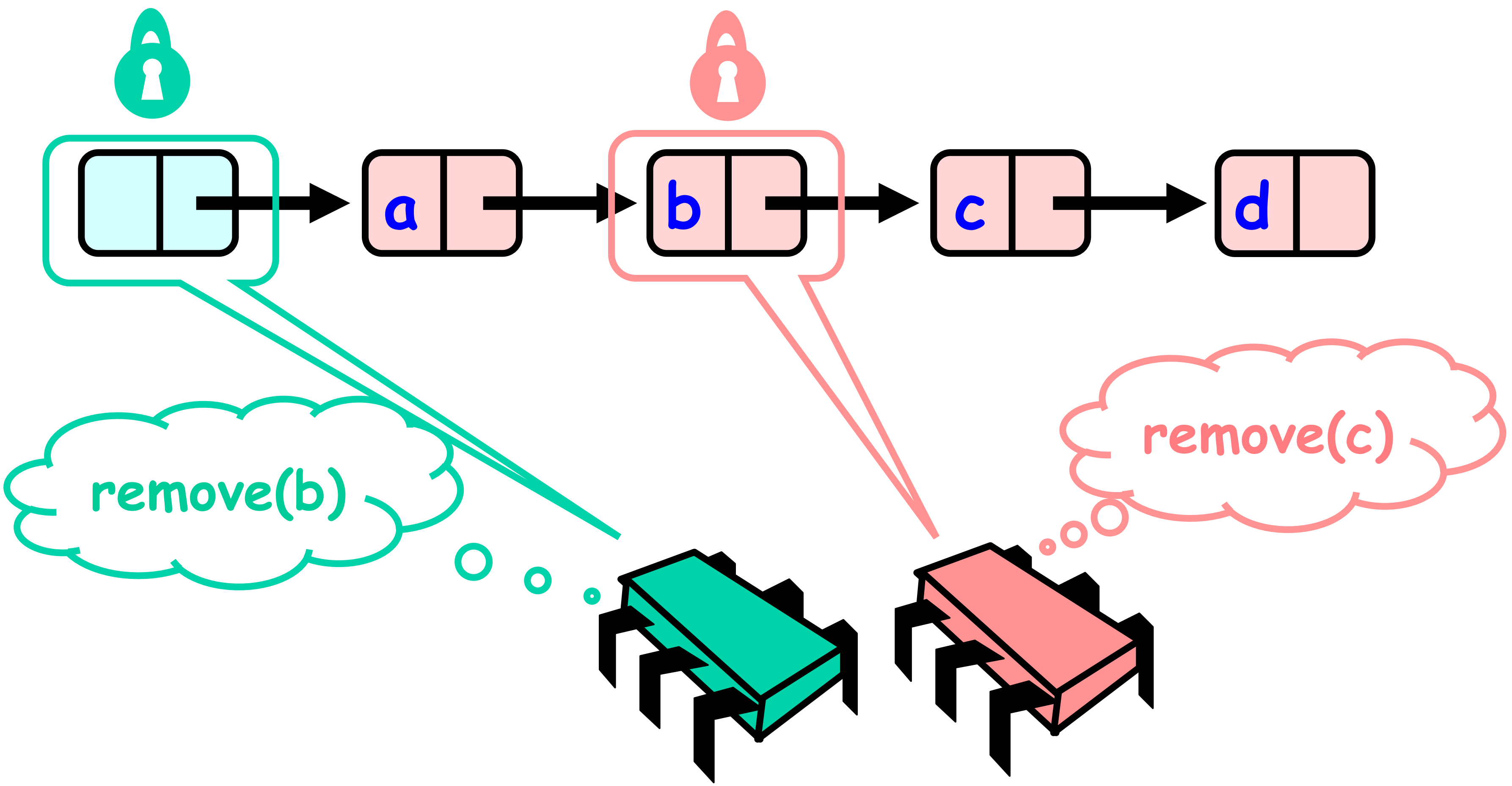
Removing a Node



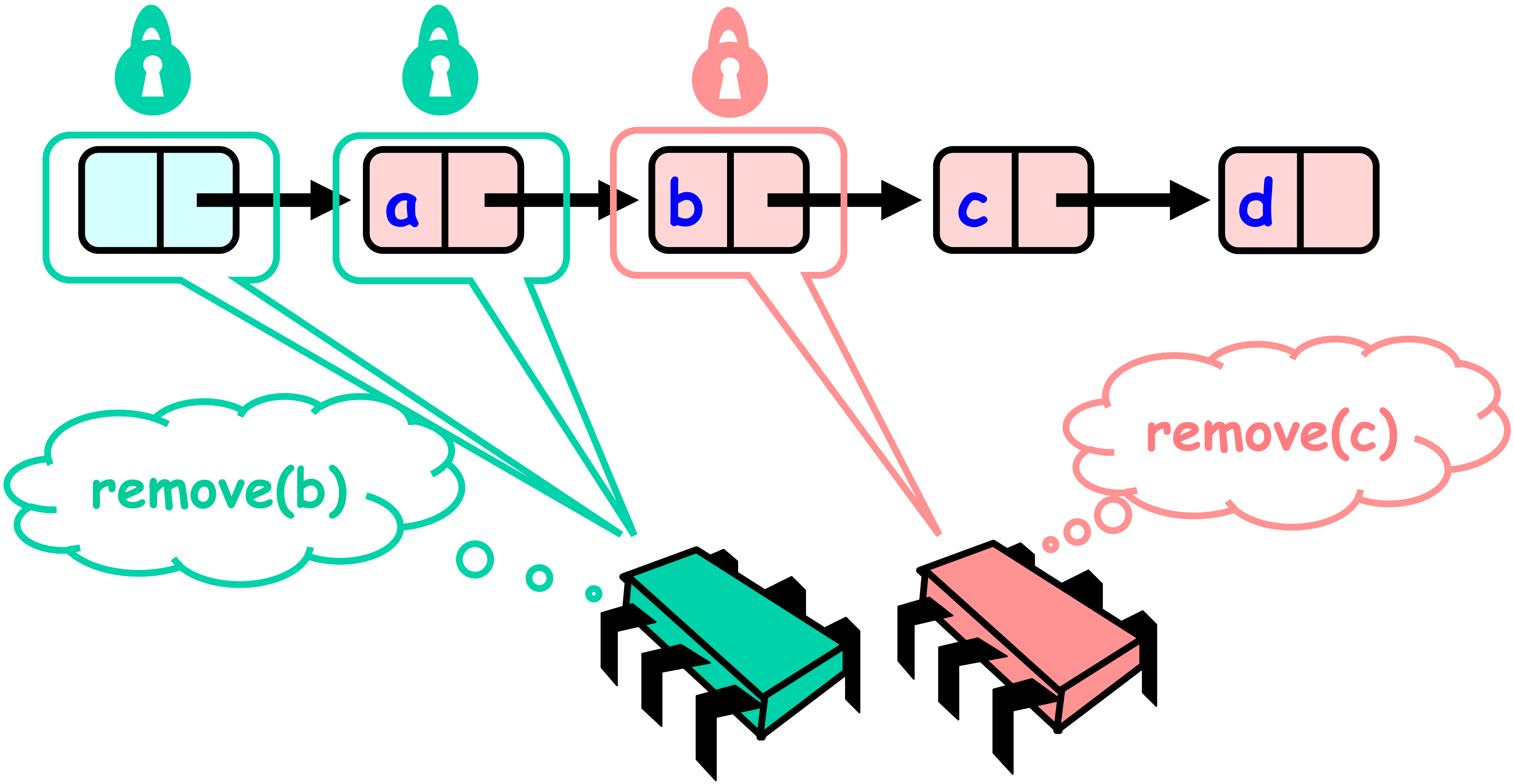
Removing a Node



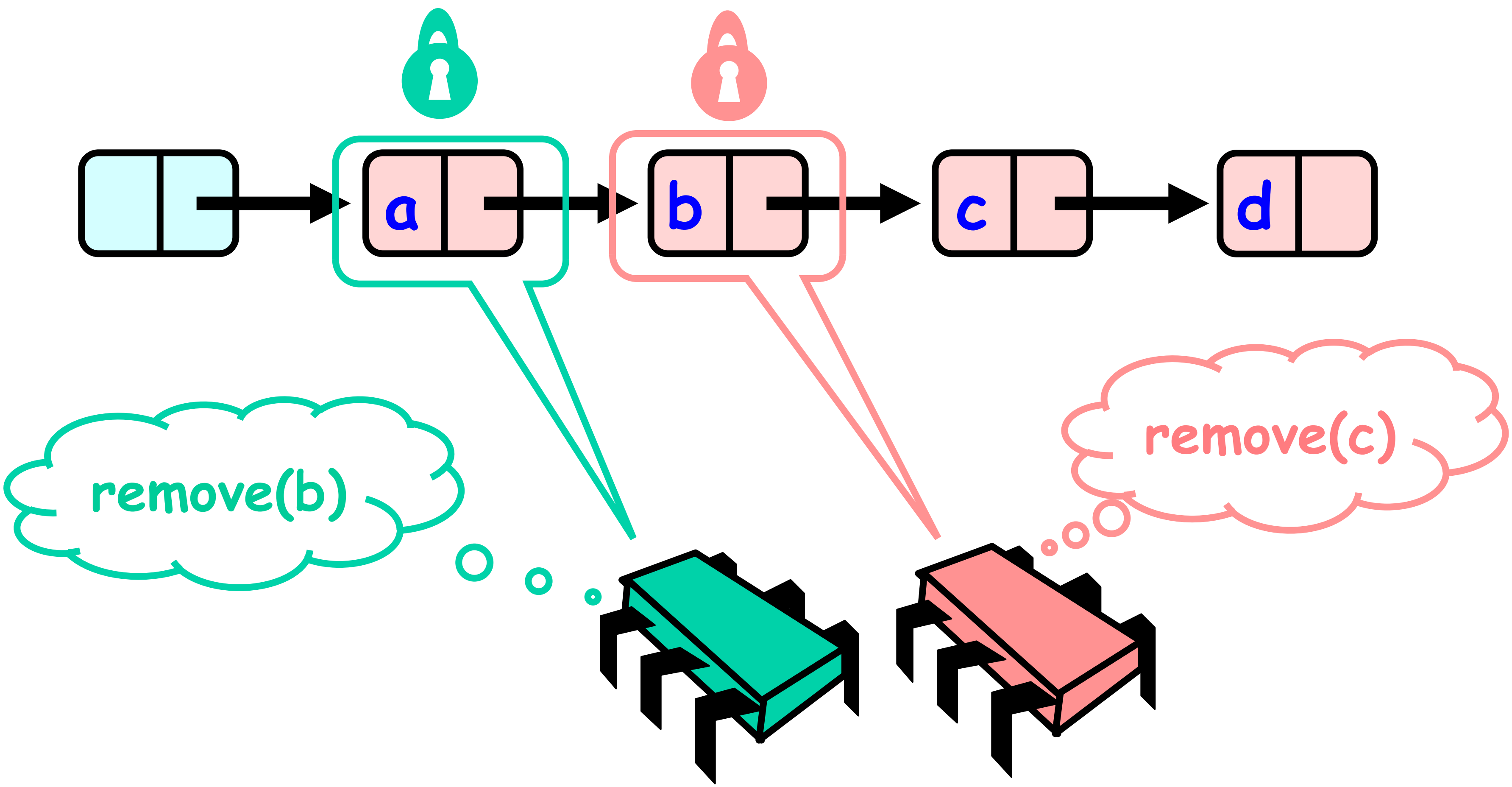
Removing a Node



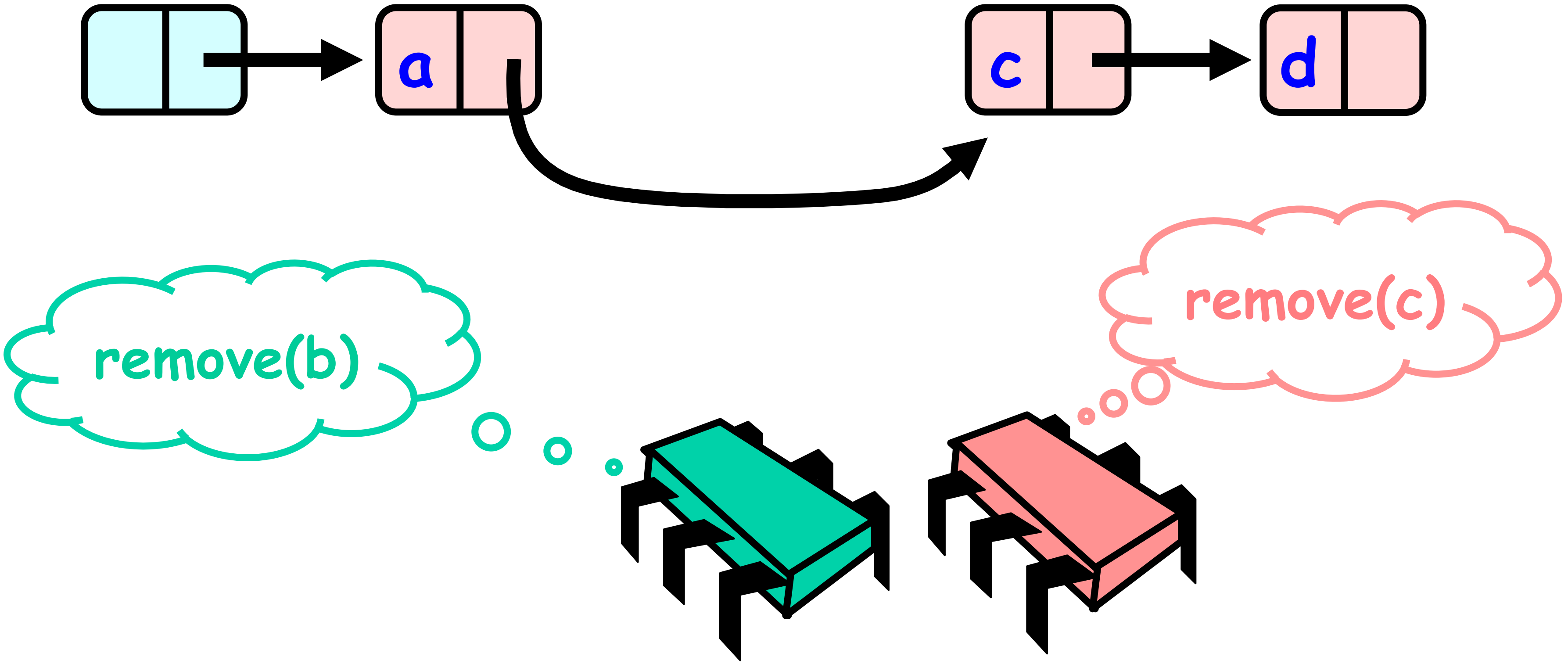
Removing a Node



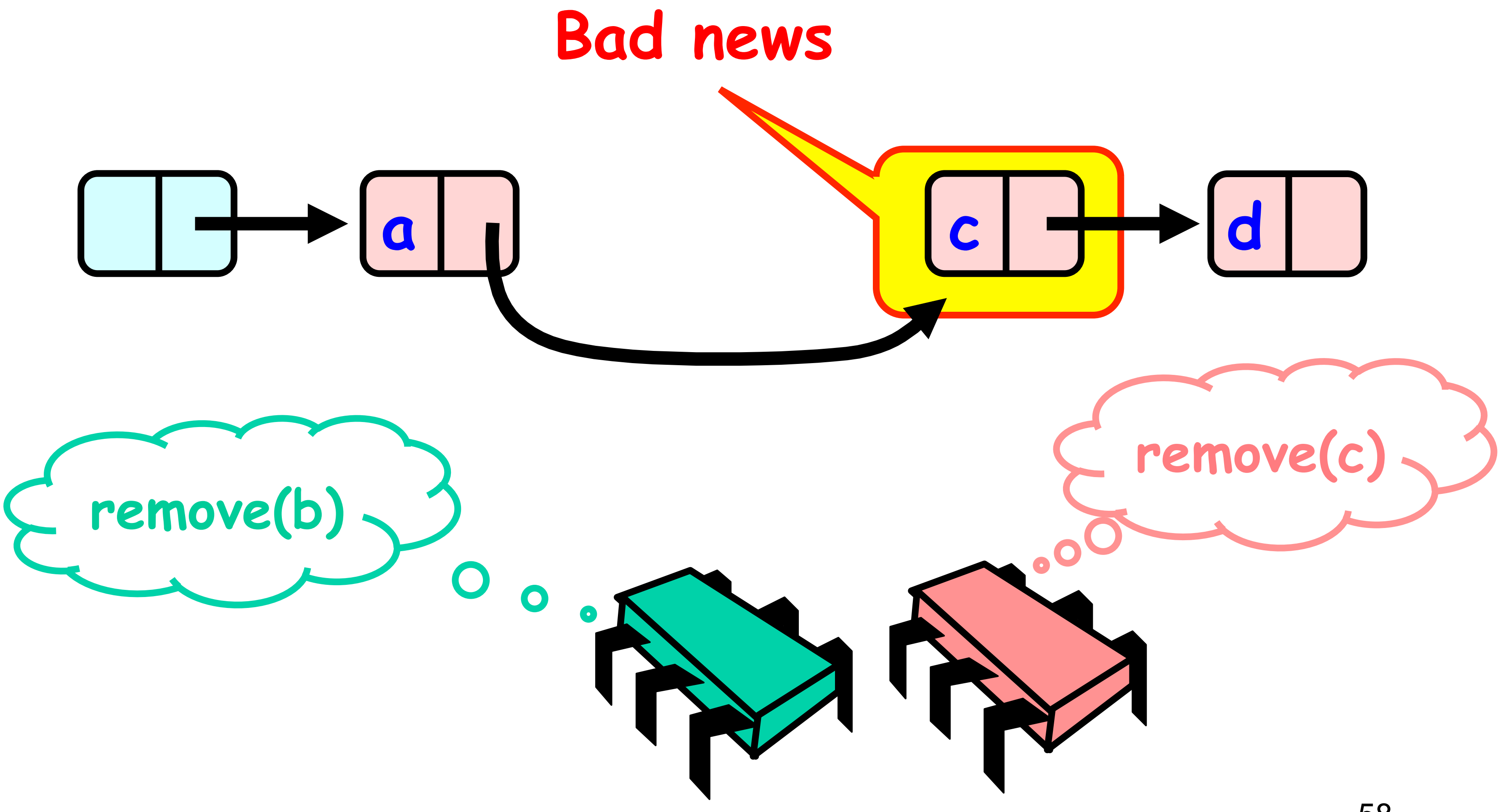
Removing a Node



Uh, Oh

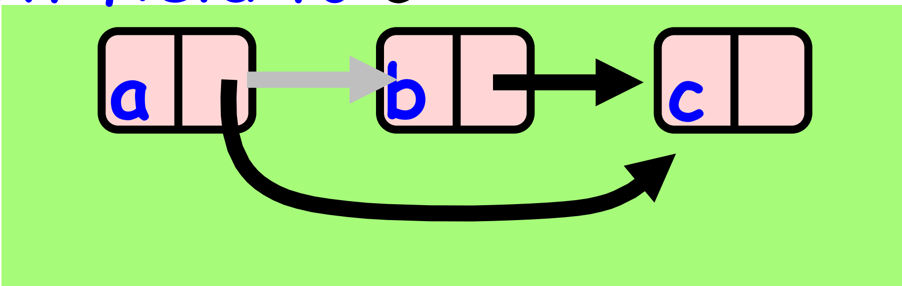


Uh, Oh

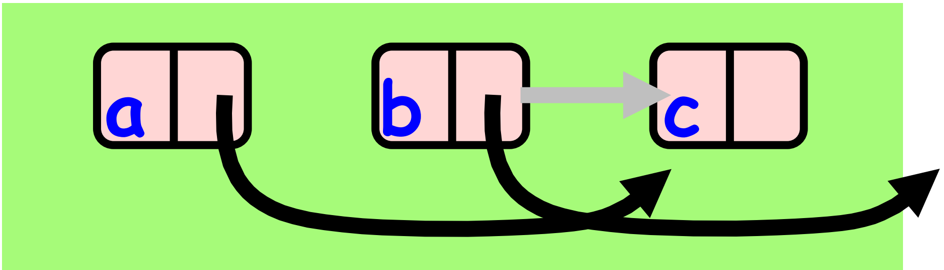


Problem

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



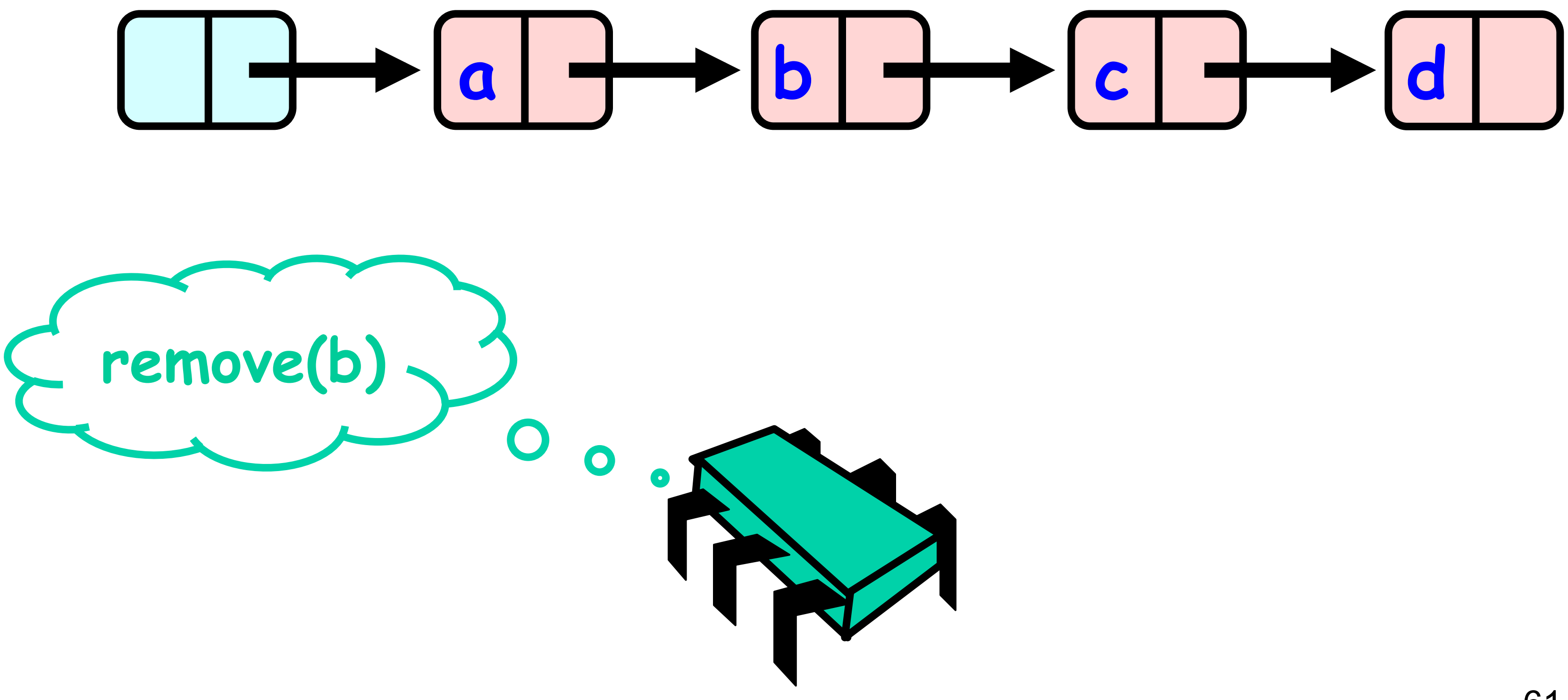
- Problem is,
  - Someone could delete c concurrently



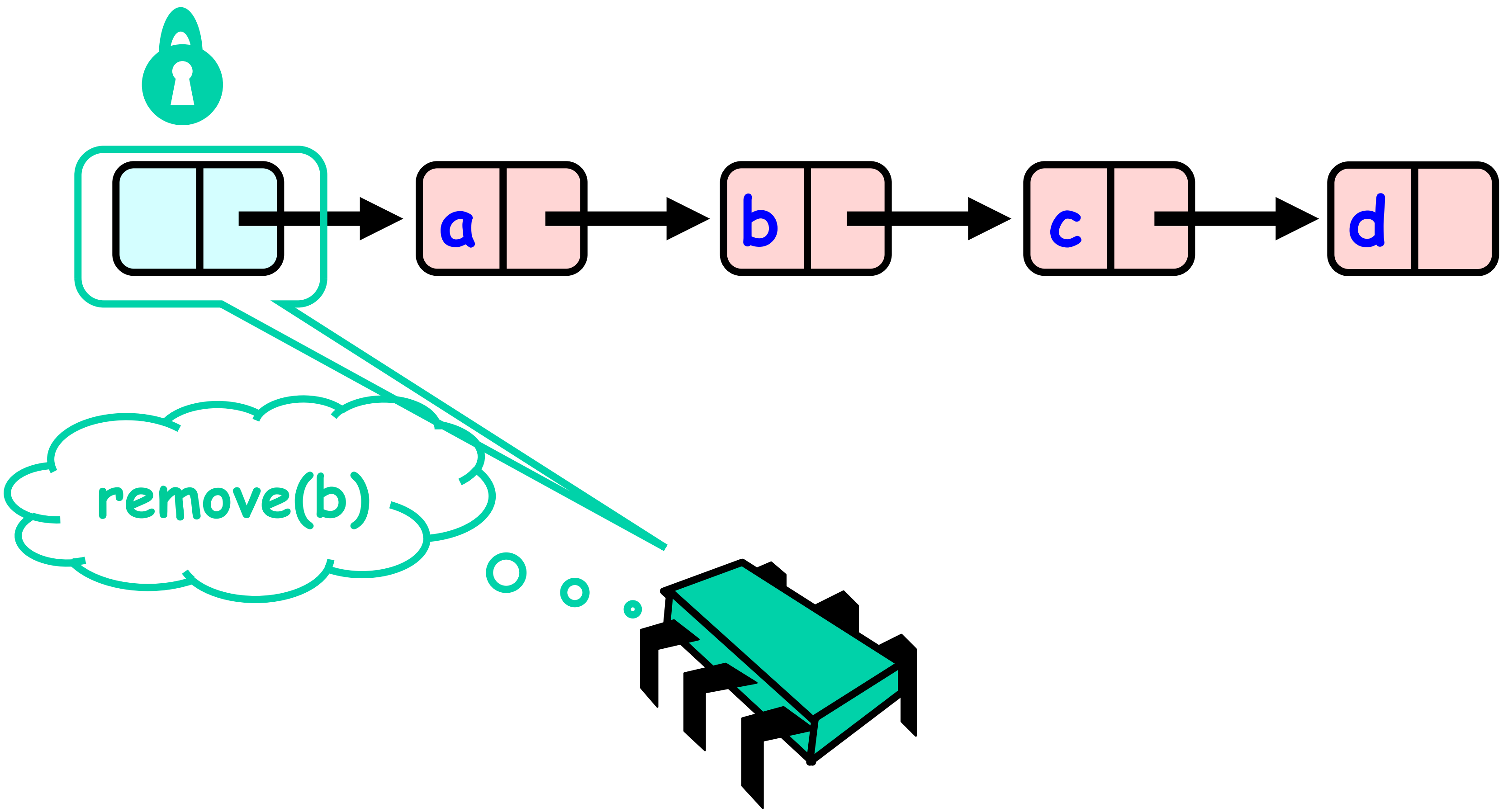
Insight

- 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

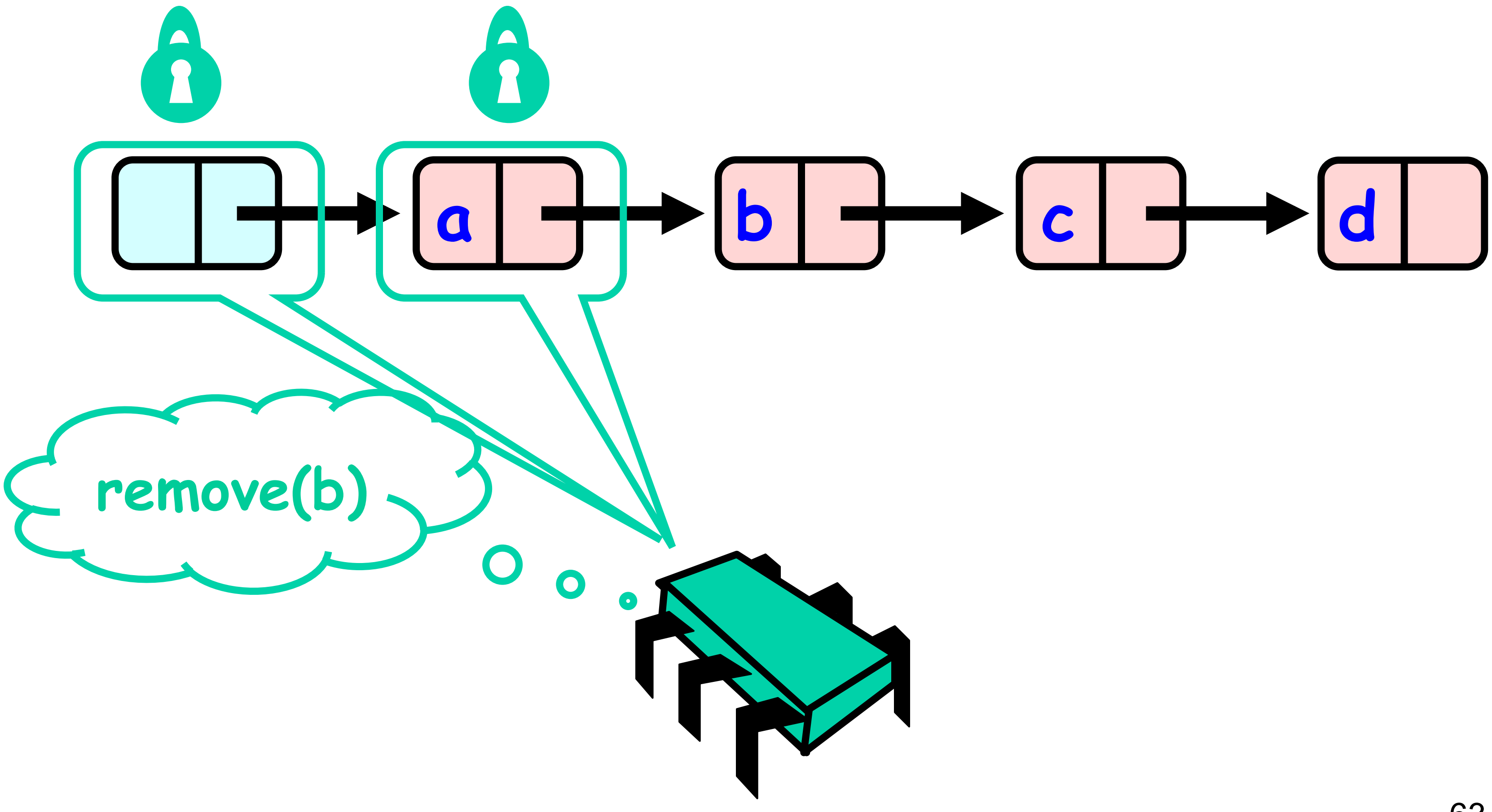
Hand-Over-Hand Again



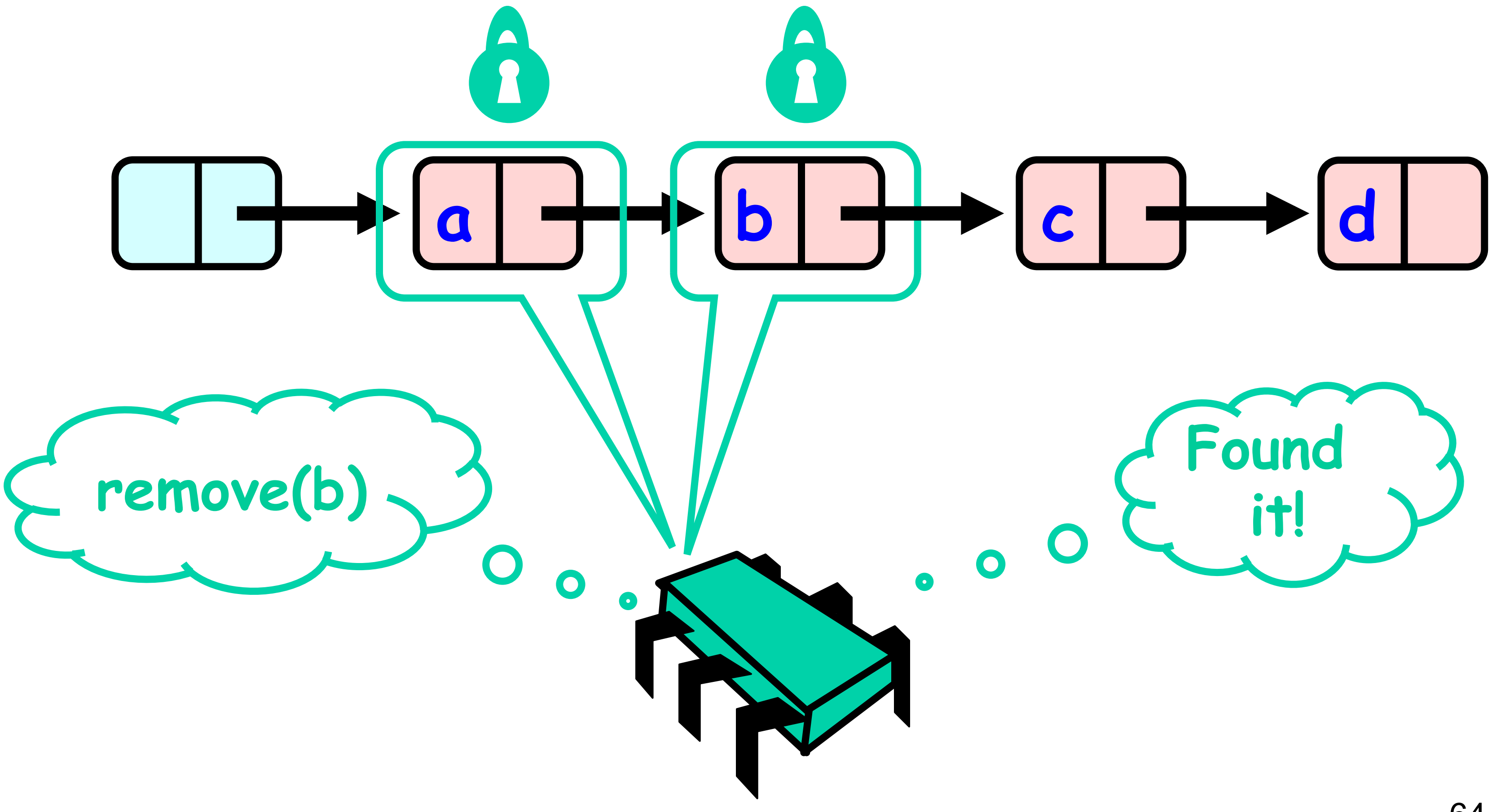
Hand-Over-Hand Again



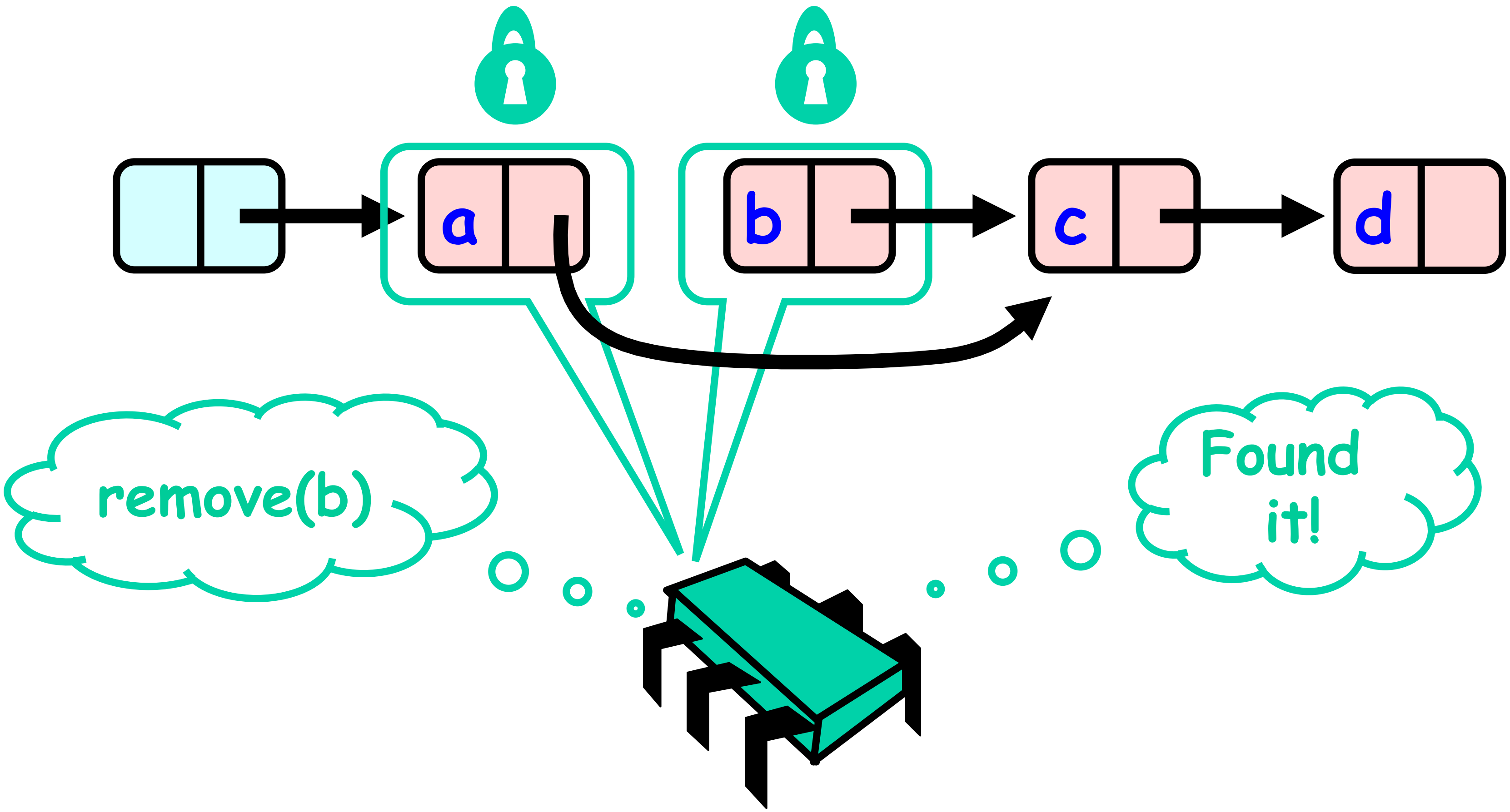
Hand-Over-Hand Again



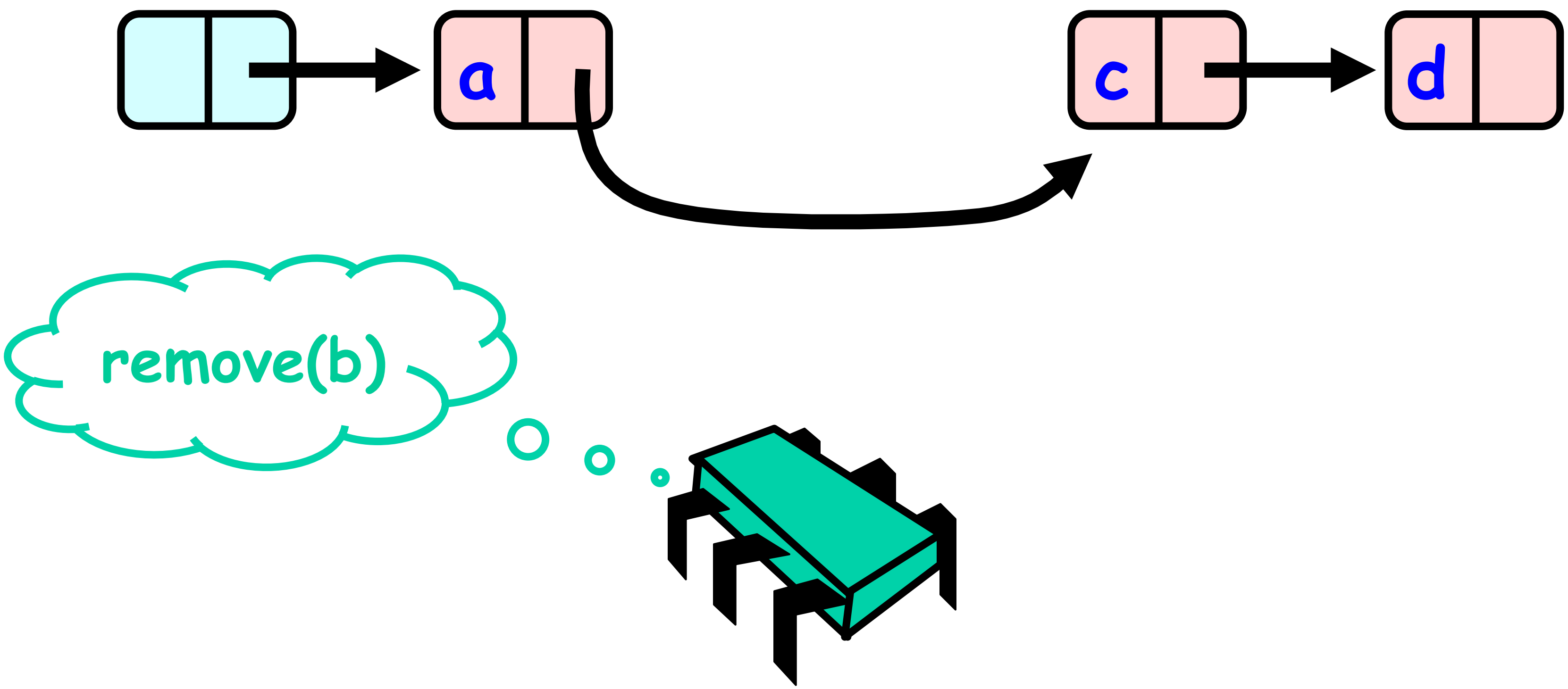
Hand-Over-Hand Again



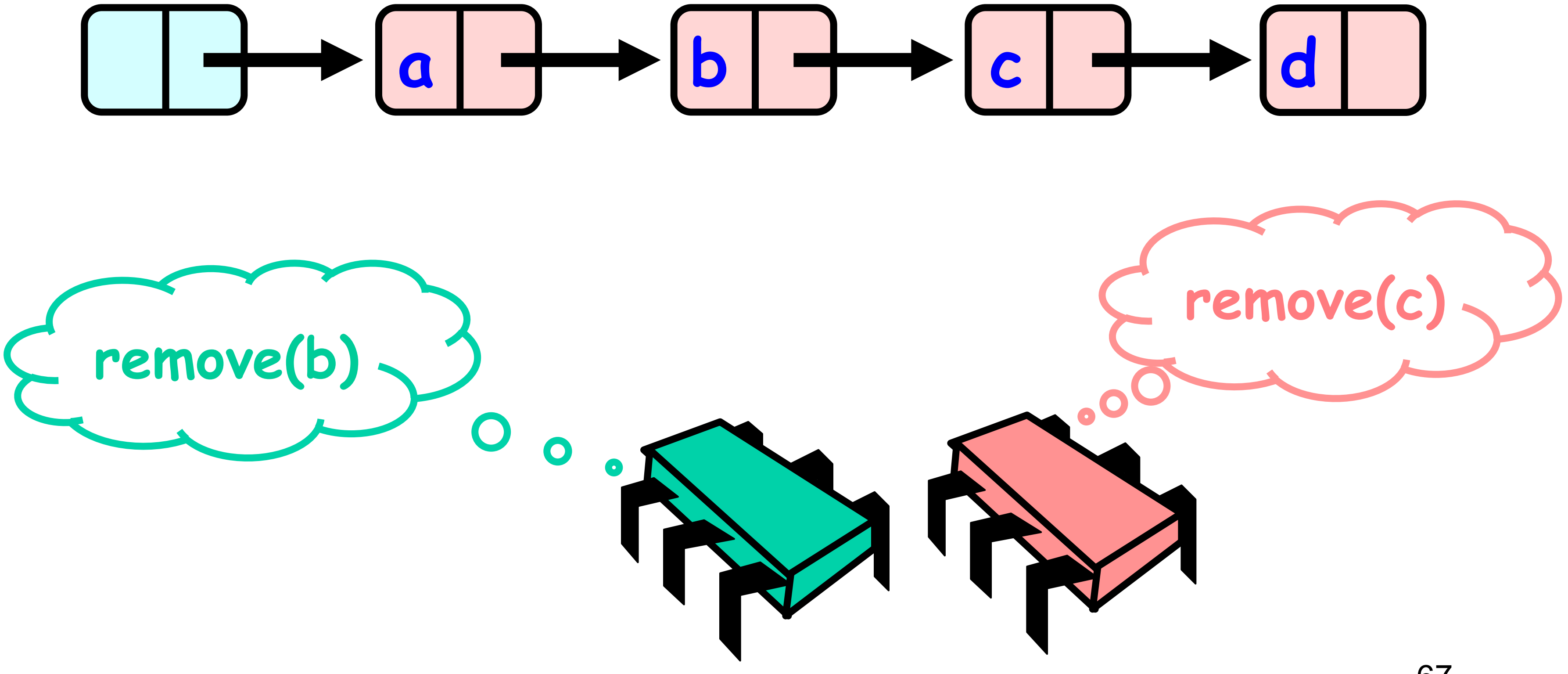
Hand-Over-Hand Again



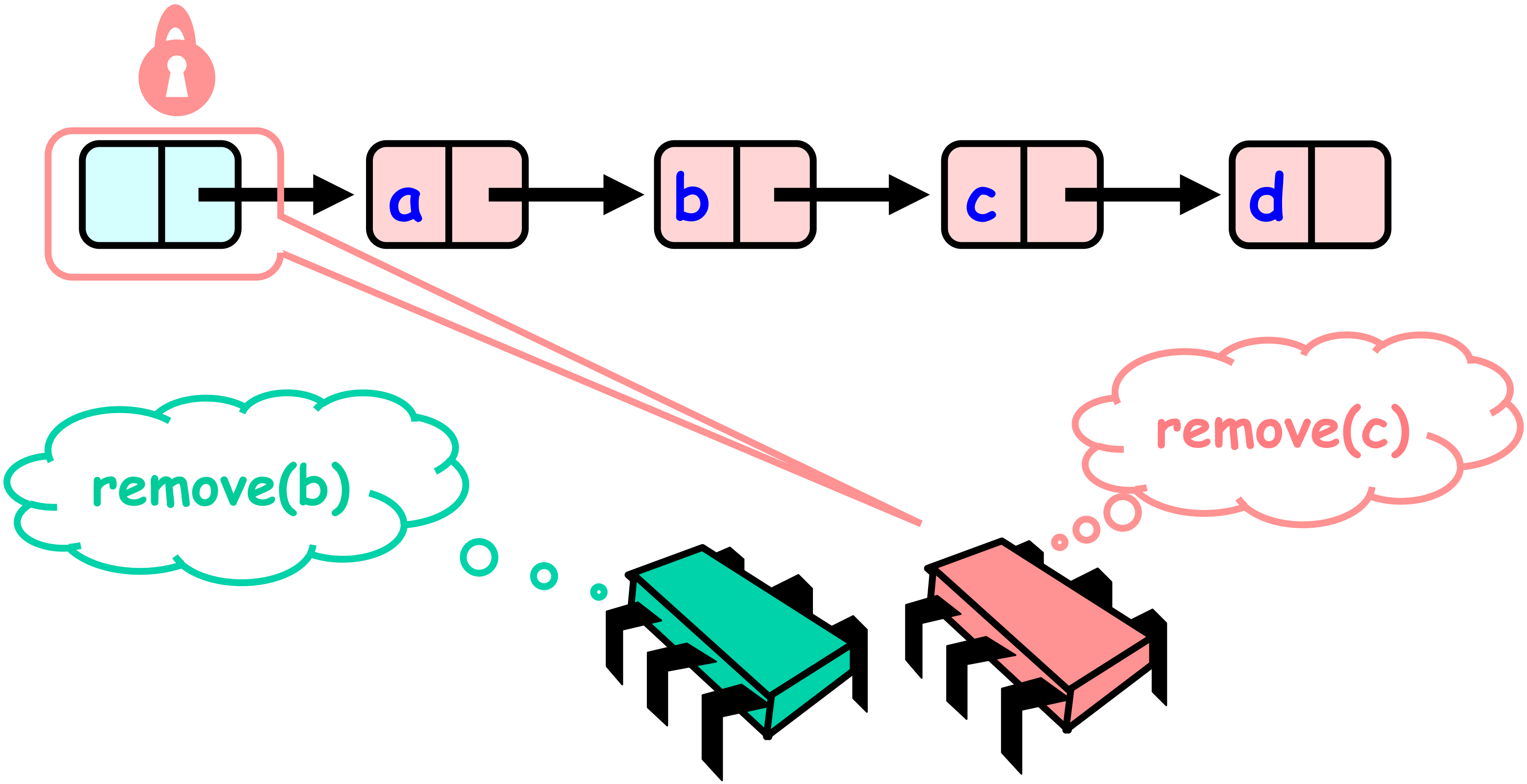
Hand-Over-Hand Again



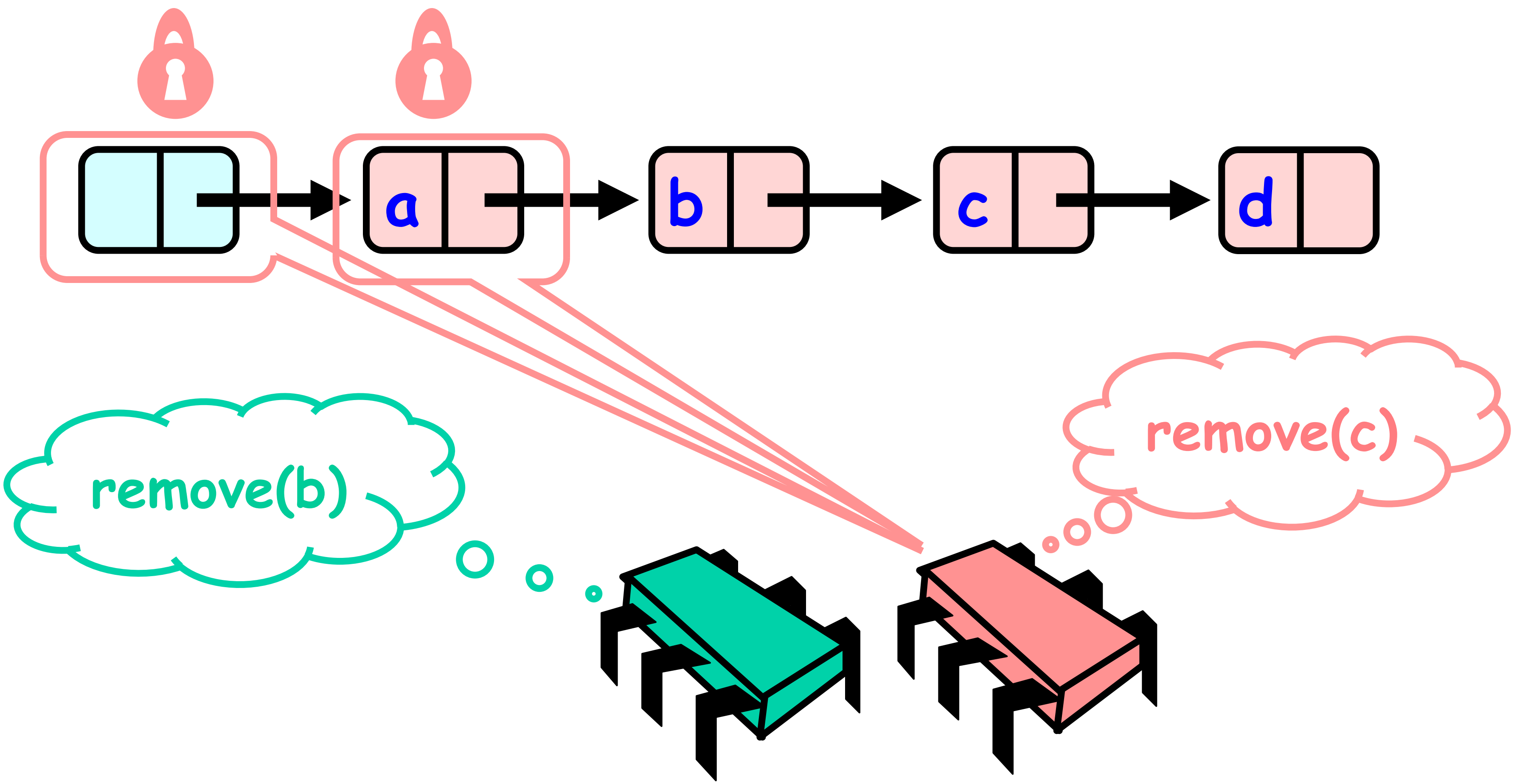
Removing a Node



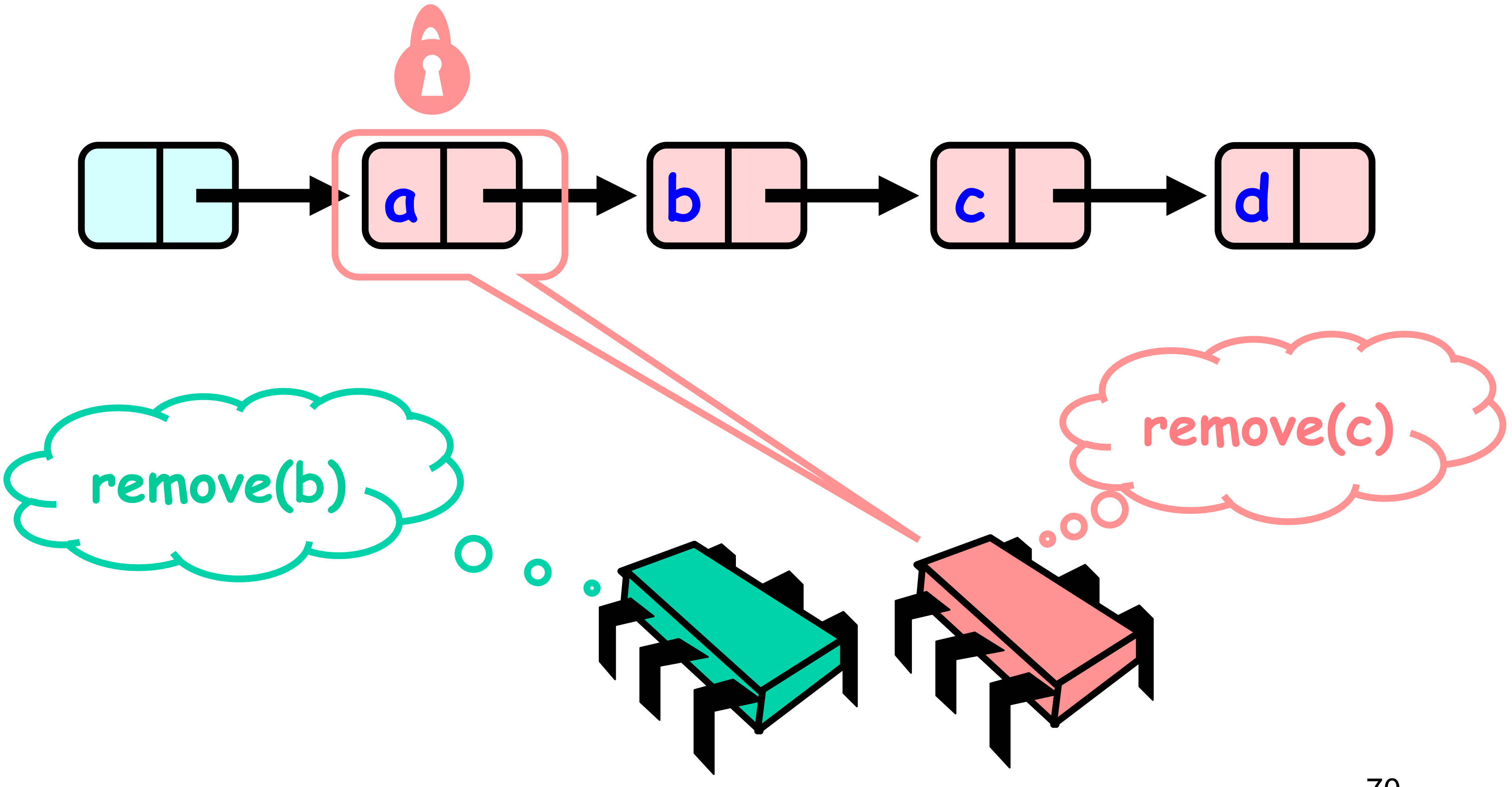
Removing a Node



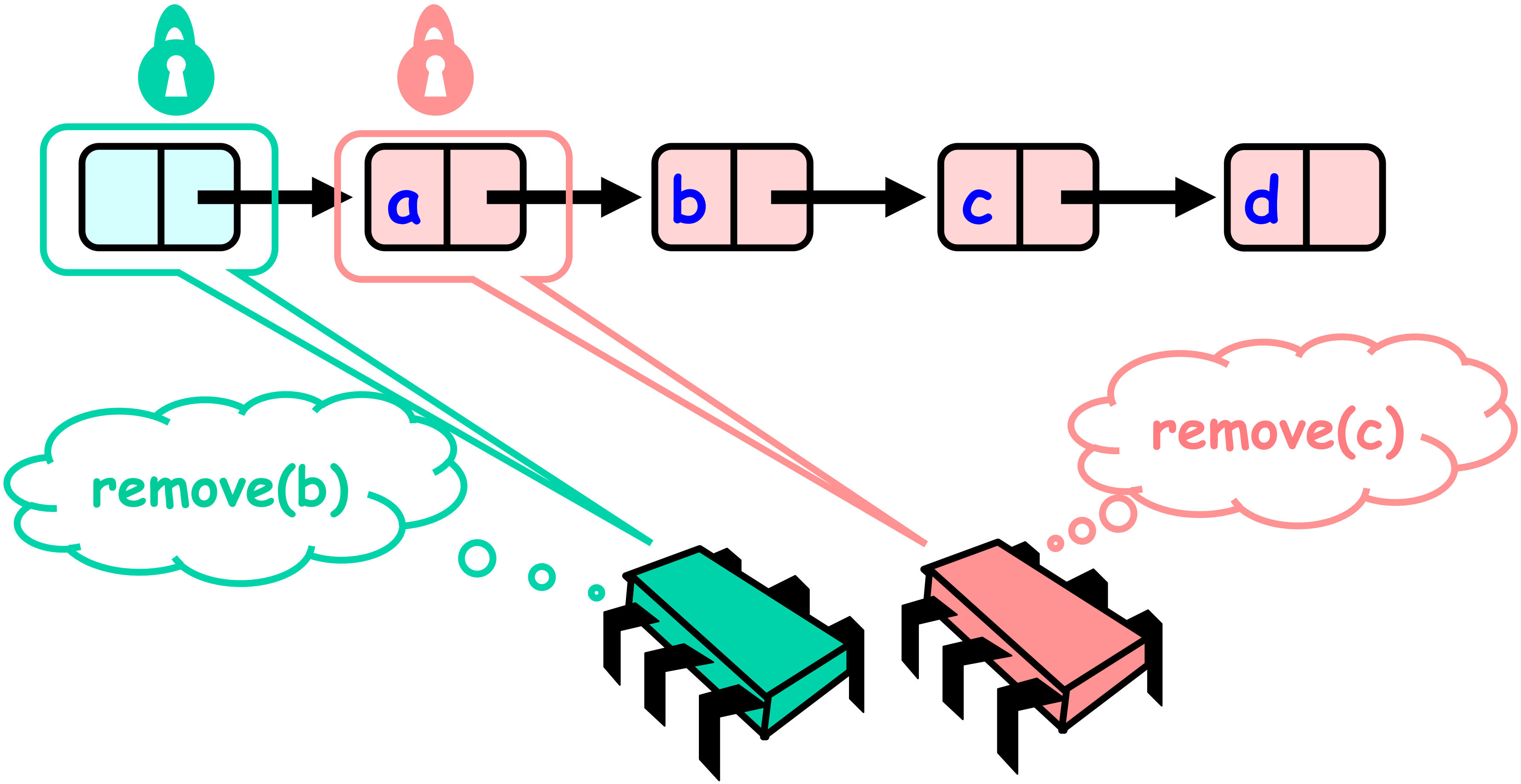
Removing a Node



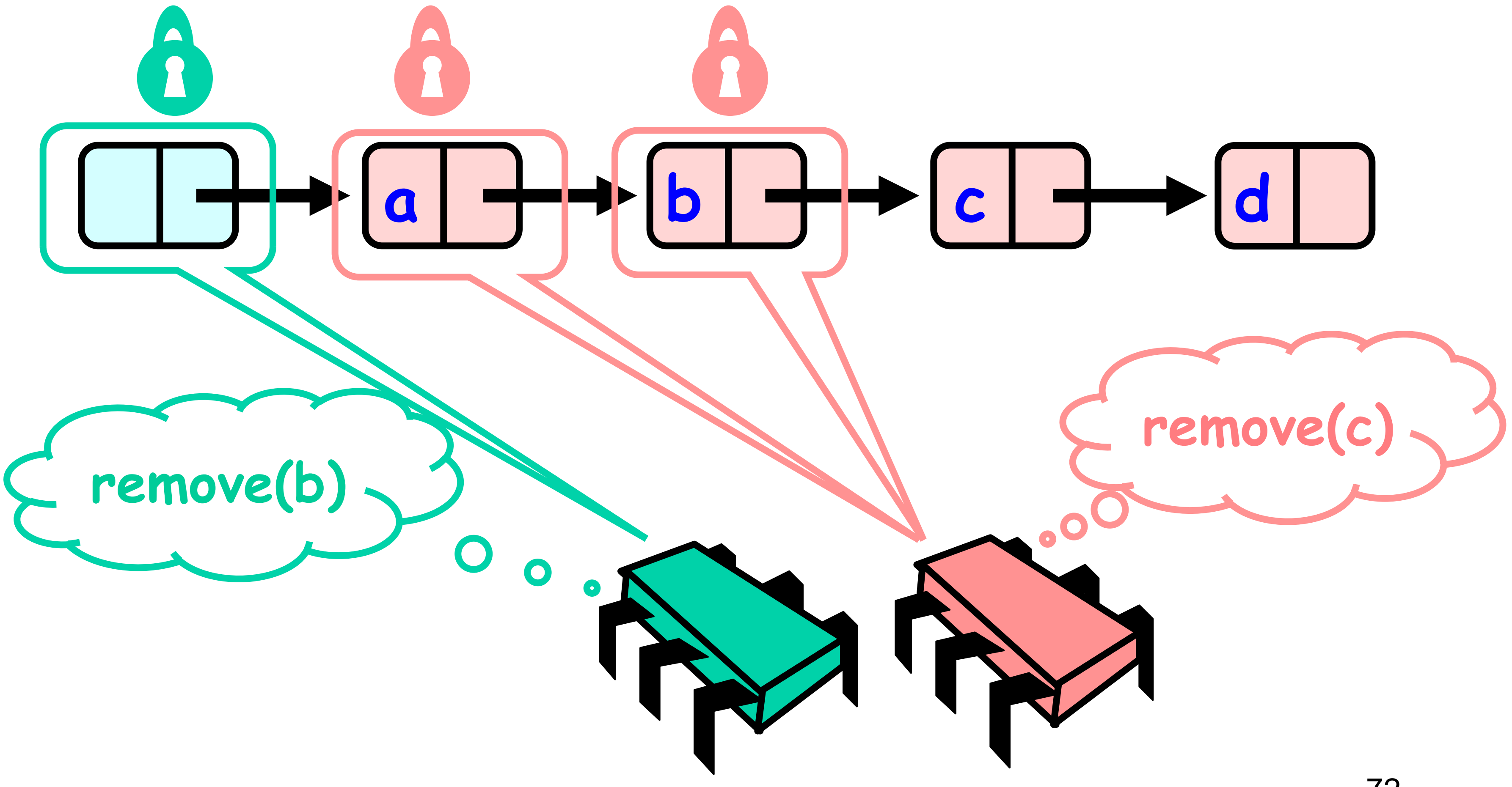
Removing a Node



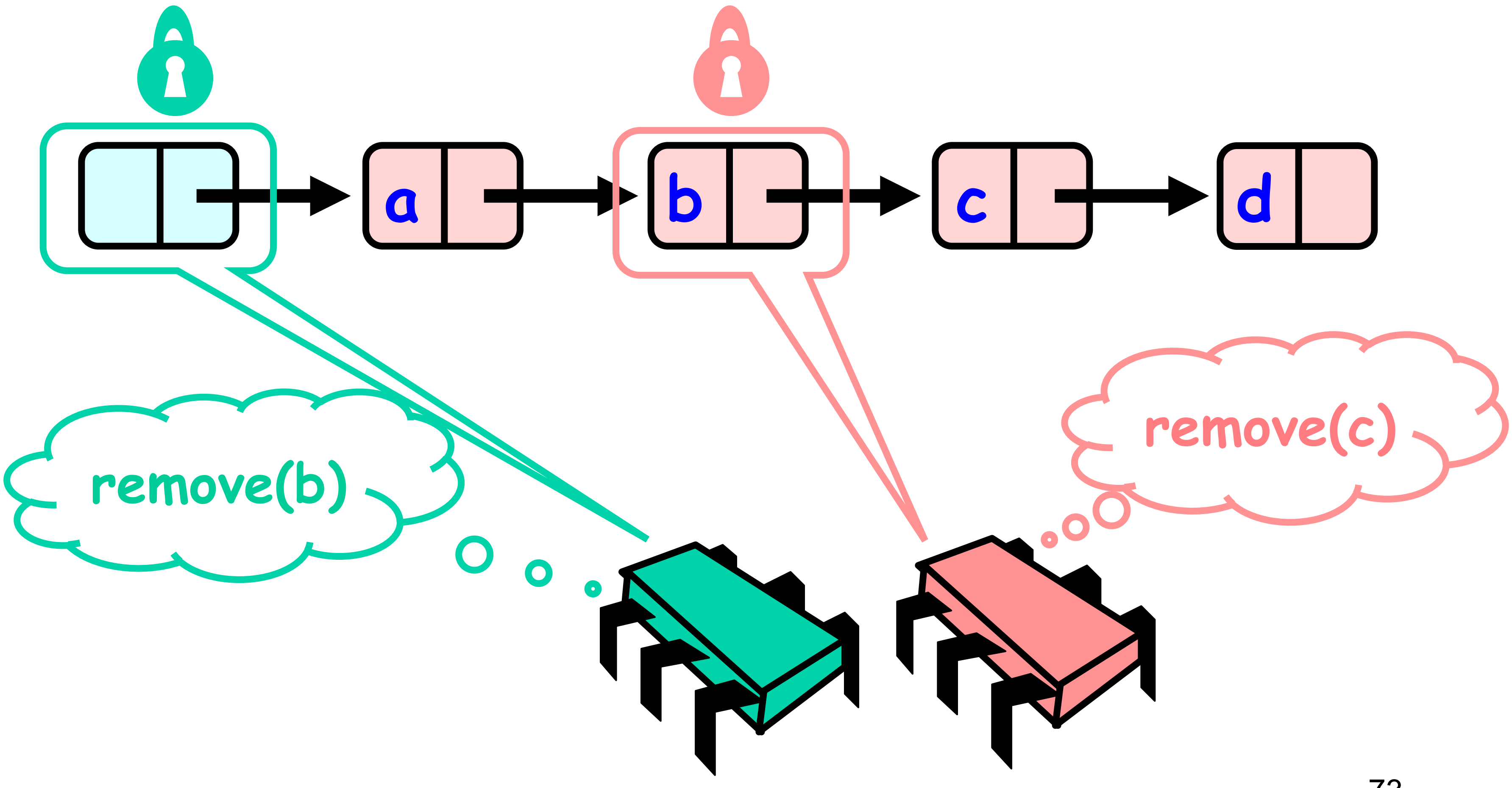
Removing a Node



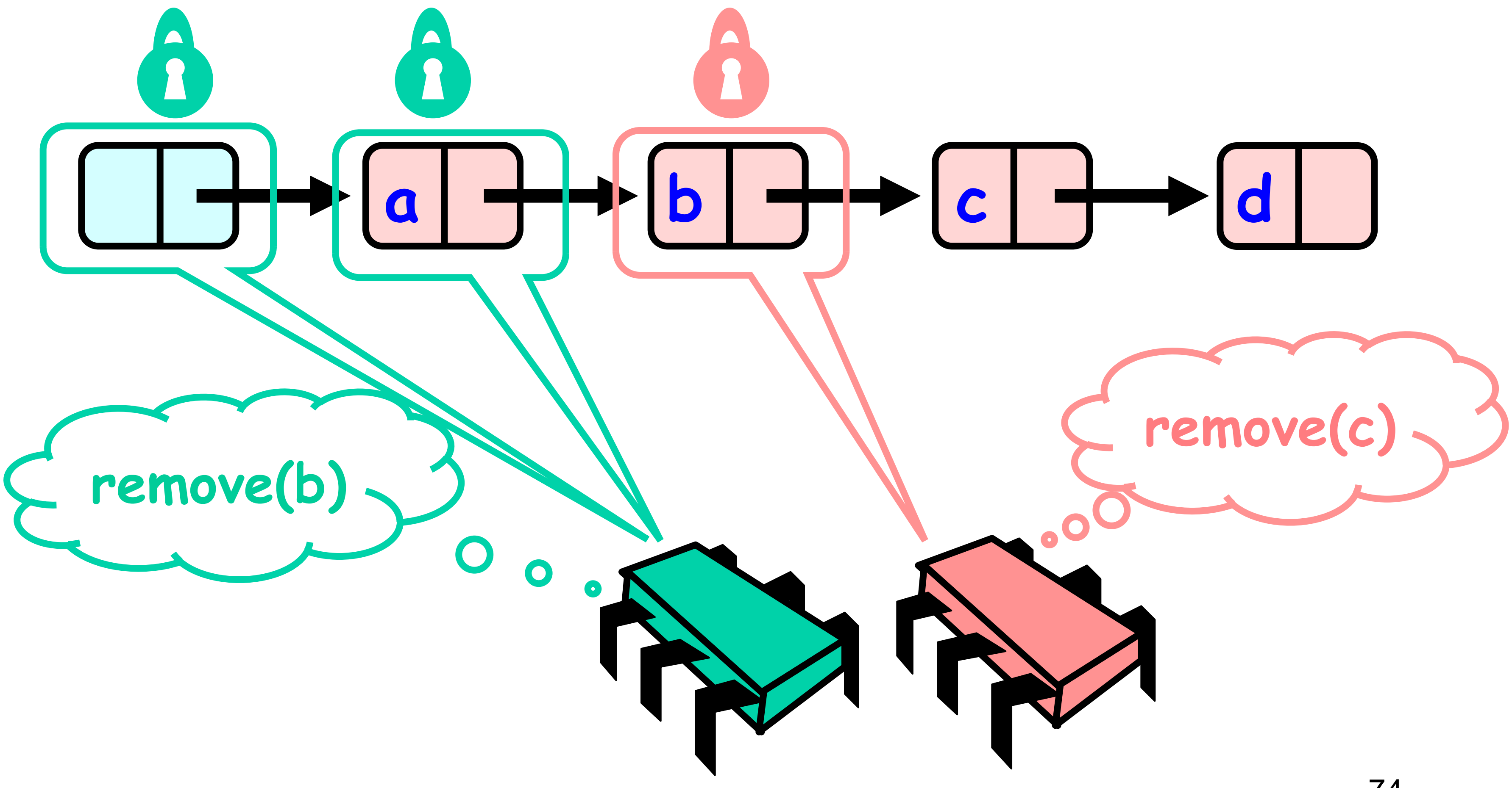
Removing a Node



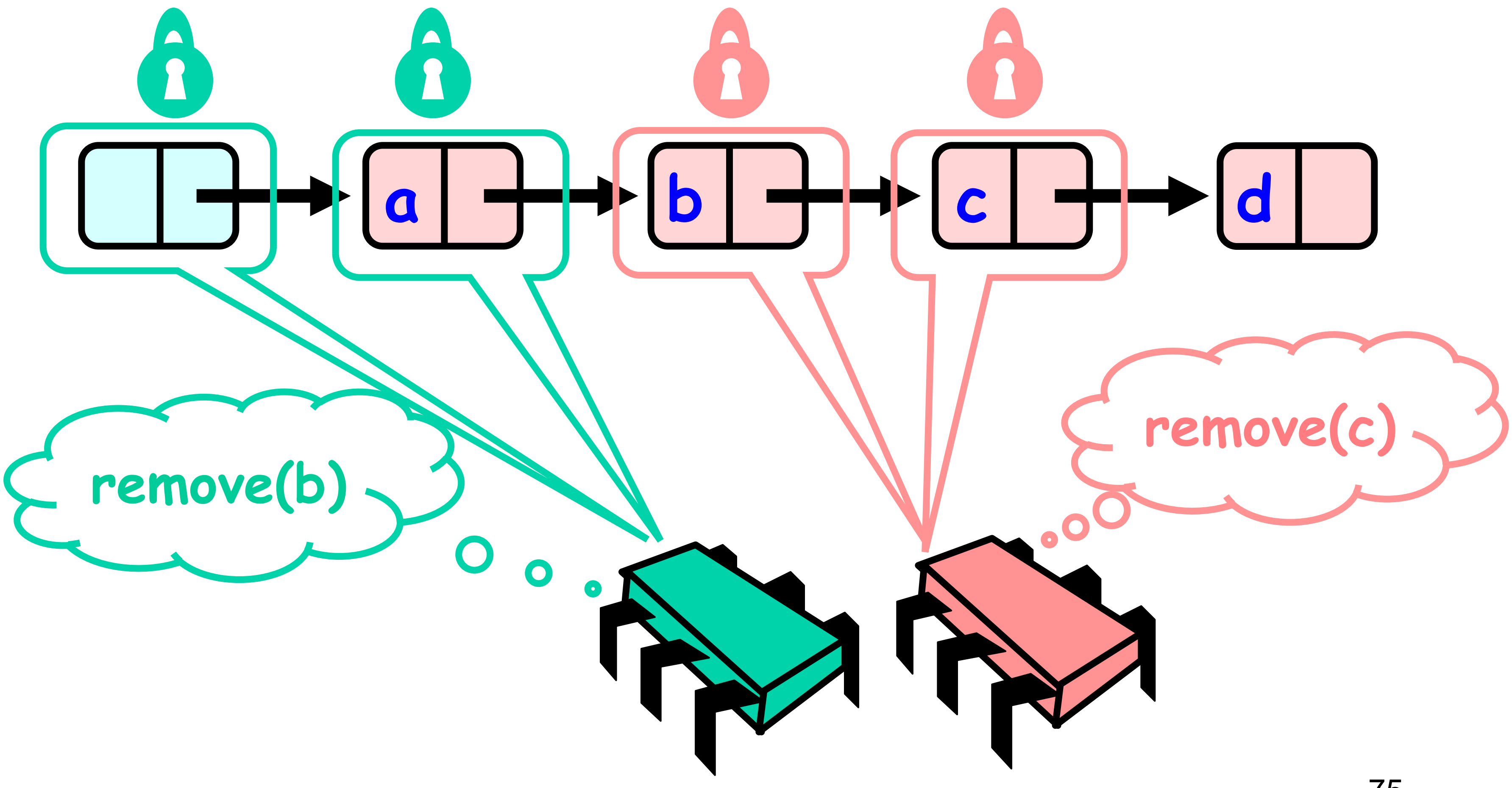
Removing a Node



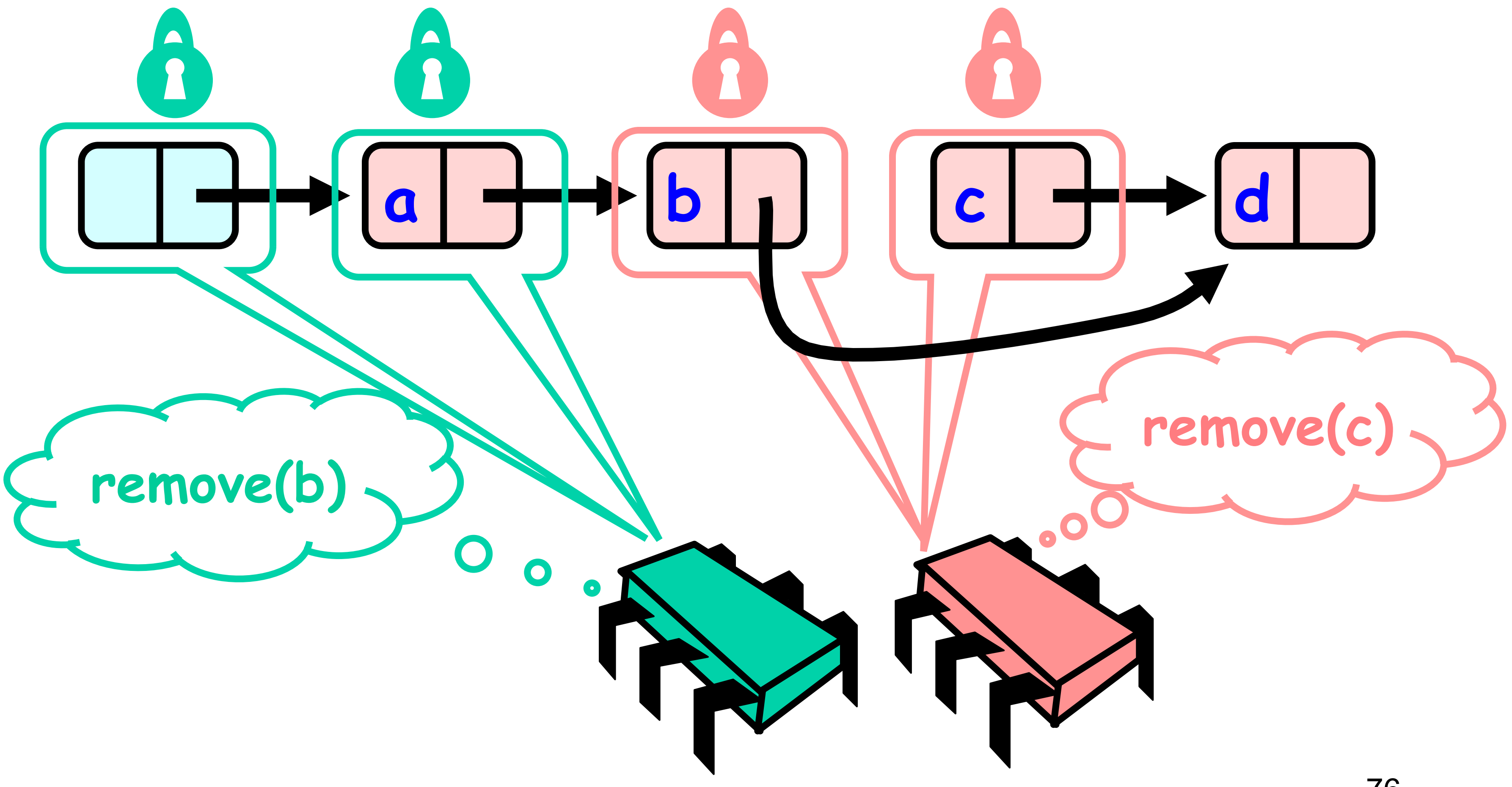
Removing a Node



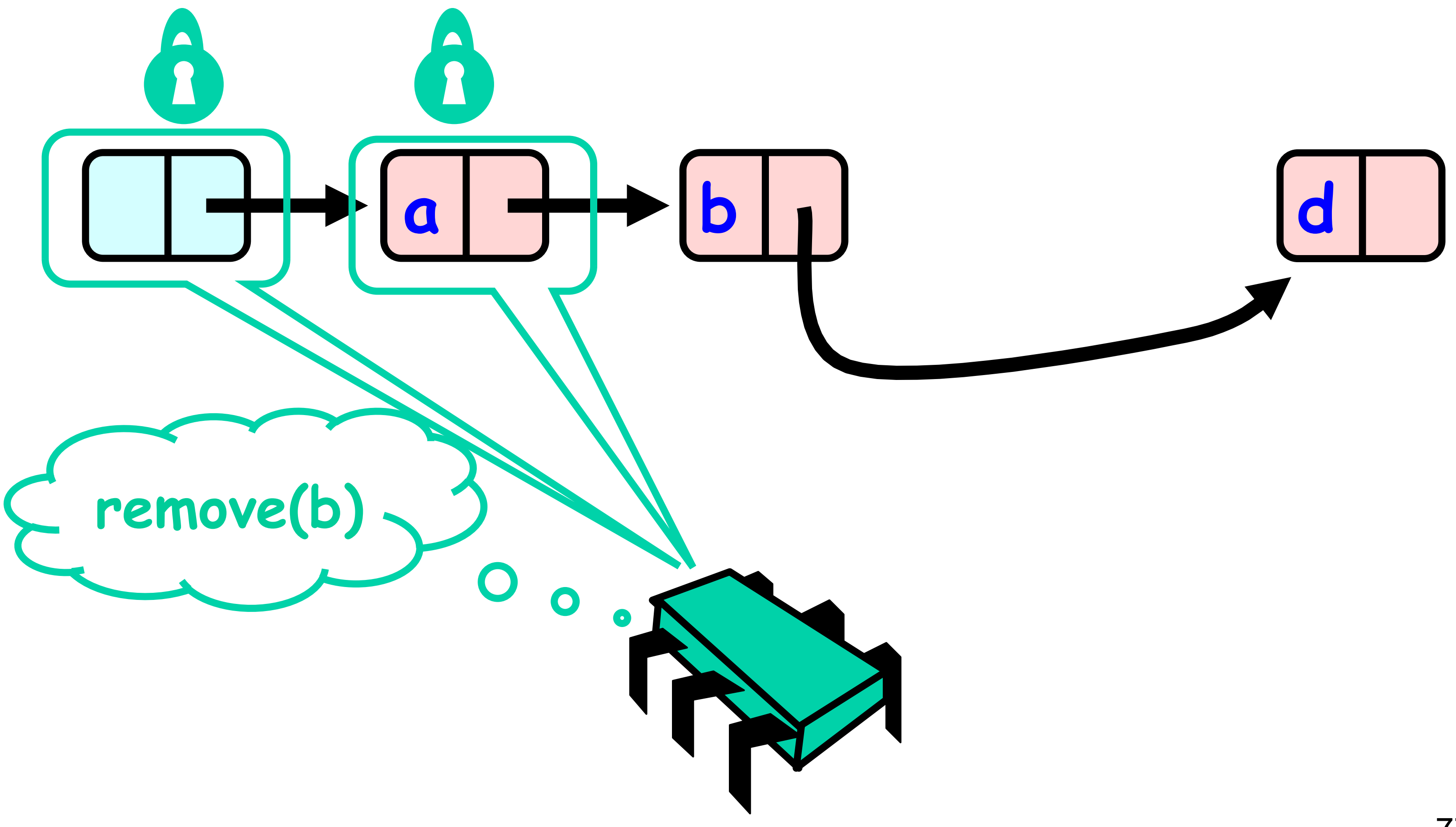
Removing a Node



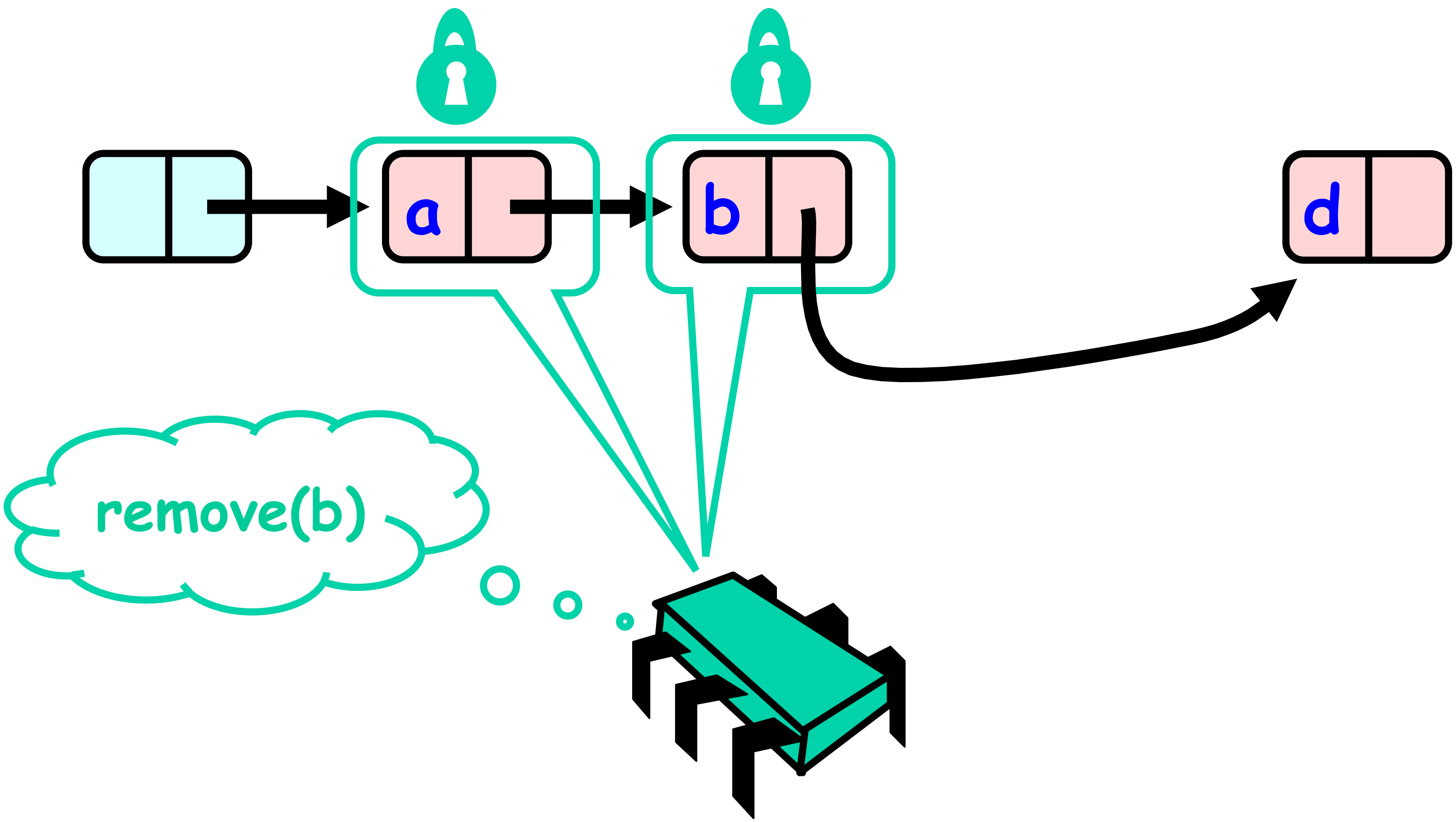
Removing a Node



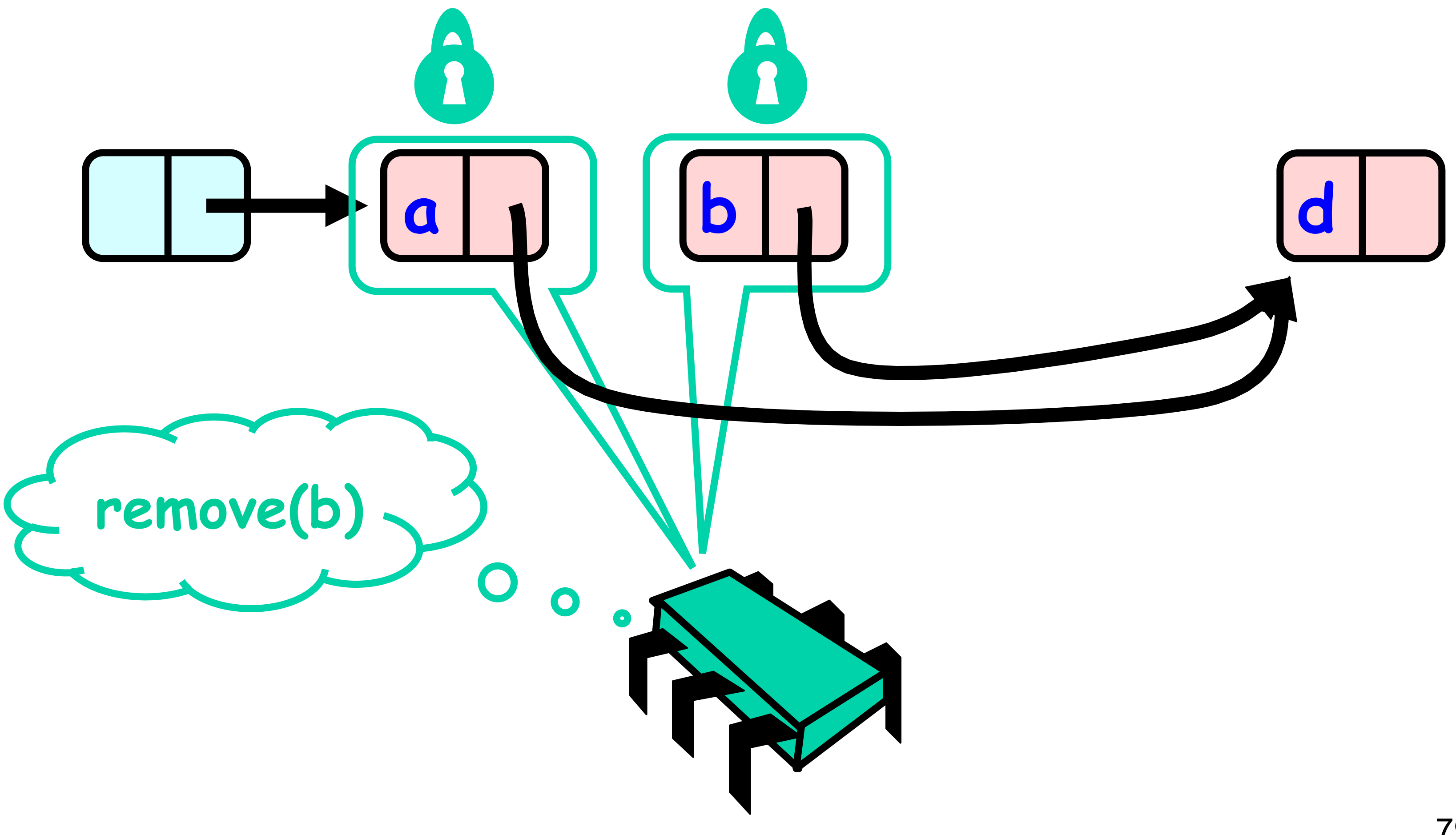
Removing a Node



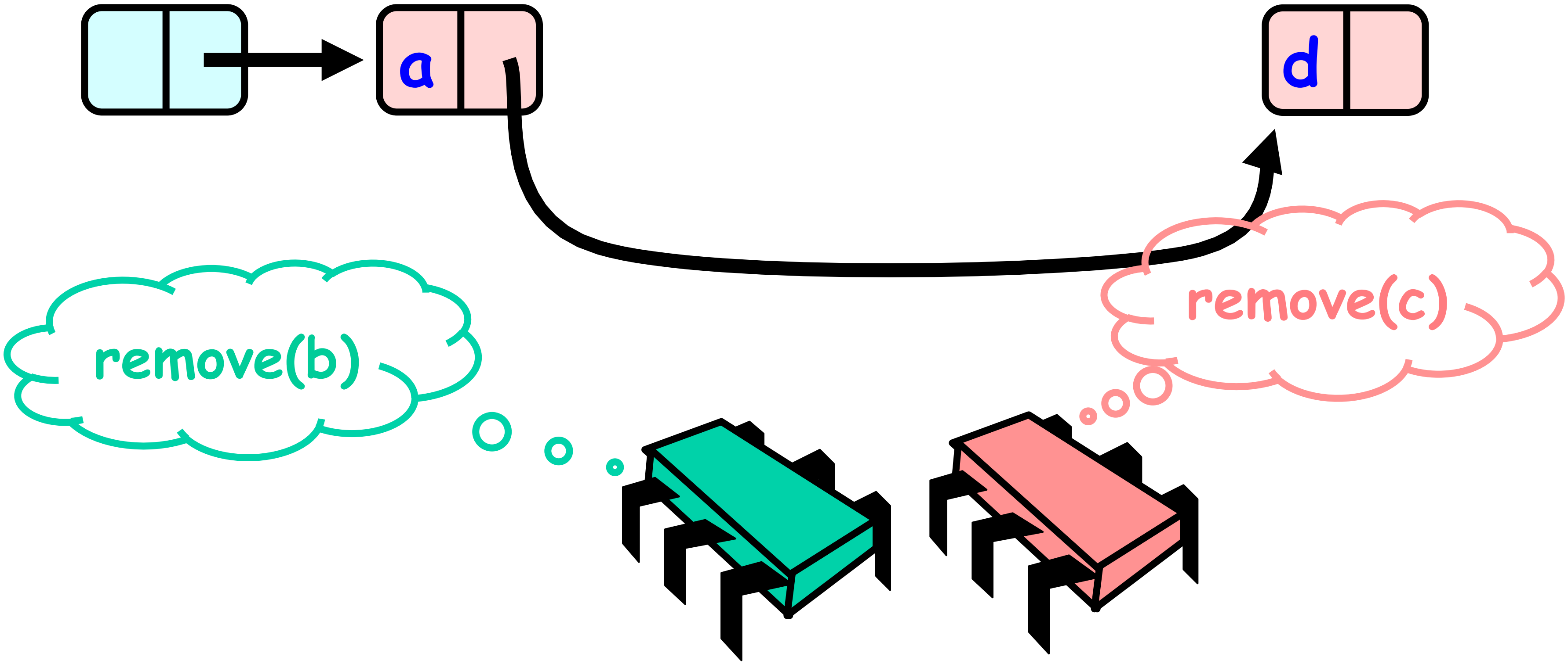
Removing a Node



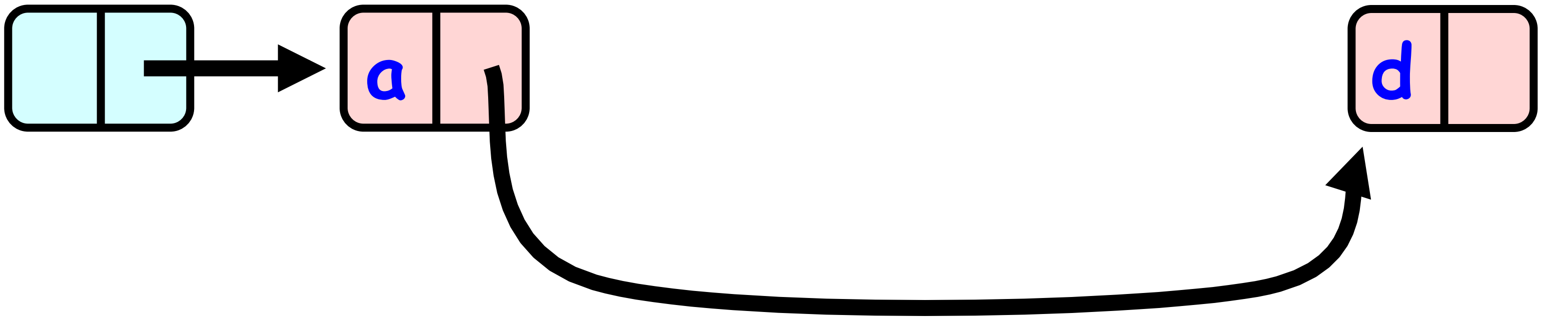
Removing a Node



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?)

Drawbacks

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

"To Lock or Not to Lock"

- Locking vs. Non-blocking: Extremist views on both sides

- Programming assignment:
  - Locking & non-blocking linked list implementations.

Grading (bonus)

- Lock-based: 0.5 points
- Lock-free: 0.5 points
- Fastest implementation
  - Lock-based: 0.5 points
  - Lock-free: 0.5 points
  - A student can get only one bonus bonus
    - If needed: 2nd fastest (lock-based) will get it

Recap

- Implement 2 linked list algorithms
  - A lock-based
  - A lock-free
- Deadline (strict):

  Monday, December 16th, 23:59