Design of Parallel and High-Performance Computing

Fall 2013

Lecture: Locks and Lock-Free continued

Instructor: Torsten Hoefler & Markus Püschel

TA: Timo Schneider

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Administrivia

- **Now: Project Presentations!**
  - 10 minutes per team (hard limit)
  - Rough guidelines:

    Present your plan

    Related work (what exists, literature review!)

    Preliminary results (not necessarily)

    Main goal is to gather feedback, so present some details

    Pick one presenter (make sure to switch for other presentations!)

- **Intermediate (very short) presentation: Thursday 11/21 during recitation**
- **Final project presentation: Monday 12/16 during last lecture**

Review of last lecture

- **Hardware operations for concurrency control**
- **More on locks (using advanced operations)**
  - Spin locks
  - Various optimized locks
- **Even more on locks (issues and extended concepts)**
  - Deadlocks, priority inversion, competitive spinning, semaphores

DPHPC Overview

concepts & techniques

DPHPC

locality

parallelism

- caches
- memory hierarchy

vector ISA

shared memory

distributed memory

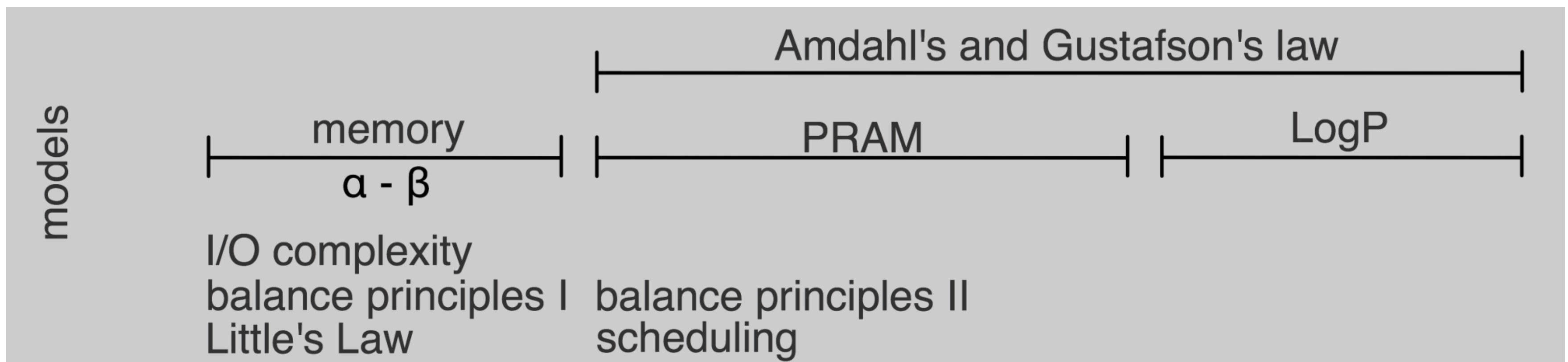
cache coherency

memory models

distributed algorithms

locks
lock free
wait free
linearizability

group commu-
nications



Goals of this lecture

- **Project presentations ☺**

- **Even more on locks (issues and extended concepts)**
  - Deadlocks, priority inversion, competitive spinning, semaphores
- **Case studies**
  - Barrier
- **Locks in practice: a set structure**

When to Spin and When to Block?

- **Spinning consumes CPU cycles but is cheap**
  - "Steals" CPU from other threads
- **Blocking has high one-time cost and is then free**
  - Often hundreds of cycles (trap, save TCB …)
  - Wakeup is also expensive (latency)

    Also cache-pollution
- **Strategy:**
  - Poll for a while and then block

When to Spin and When to Block?

- **What is a "while"?**
- **Optimal time depends on the future**
  - When will the active thread leave the CR?
  - Can compute optimal offline schedule
  - Actual problem is an online problem
- **Competitive algorithms**
  - An algorithm is c-competitive if for a sequence of actions x and a constant a holds:

    $C(x) \leq c*C_{opt}(x) + a$
  - What would a good spinning algorithm look like and what is the competitiveness?

Competitive Spinning

- **If T is the overhead to process a wait, then a locking algorithm that spins for time T before it blocks is 2-competitive!**
  - Karlin, Manasse, McGeoch, Owicki: "Competitive Randomized Algorithms for Non-Uniform Problems", SODA 1989
- **If randomized algorithms are used, then e/(e-1)-competitiveness (~1.58) can be achieved**

Generalized Locks: Semaphores

- **Controlling access to more than one resource**
  - Described by Dijkstra 1965
- **Internal state is an atomic counter C**
- **Two operations:**
  - P() – block until C>0; decrement C (atomically)
  - V() – signal and increment C
- **Binary or 0/1 semaphore equivalent to lock**
  - C is always 0 or 1, i.e., V() will not increase it further
- **Trivia:**
  - If you're lucky (aehem, speak Dutch), mnemonics:

    ***V**erhogen (increment) and **P**rolaag (probeer te verlagen = try to reduce)*

Semaphore Implementation

- **Can be implemented with mutual exclusion!**
  - And can be used to implement mutual exclusion ☺
- **… or with test and set and many others!**
- **Also has fairness concepts:**
  - Order of granting access to waiting (queued) threads
  - strictly fair (starvation impossible, e.g., FIFO)
  - weakly fair (starvation possible, e.g., random)

Case Study 1: Barrier

- **Barrier semantics:**
  - No process proceeds before all processes reached barrier
  - Similar to mutual exclusion but not exclusive, rather “synchronized”
- **Often needed in parallel high-performance programming**
  - Especially in SPMD programming style
- **Parallel programming “frameworks” offer barrier semantics (pthread, OpenMP, MPI)**
  - MPI_Barrier() (process-based)
  - pthread_barrier
  - #pragma omp barrier
  - lock xadd + spin

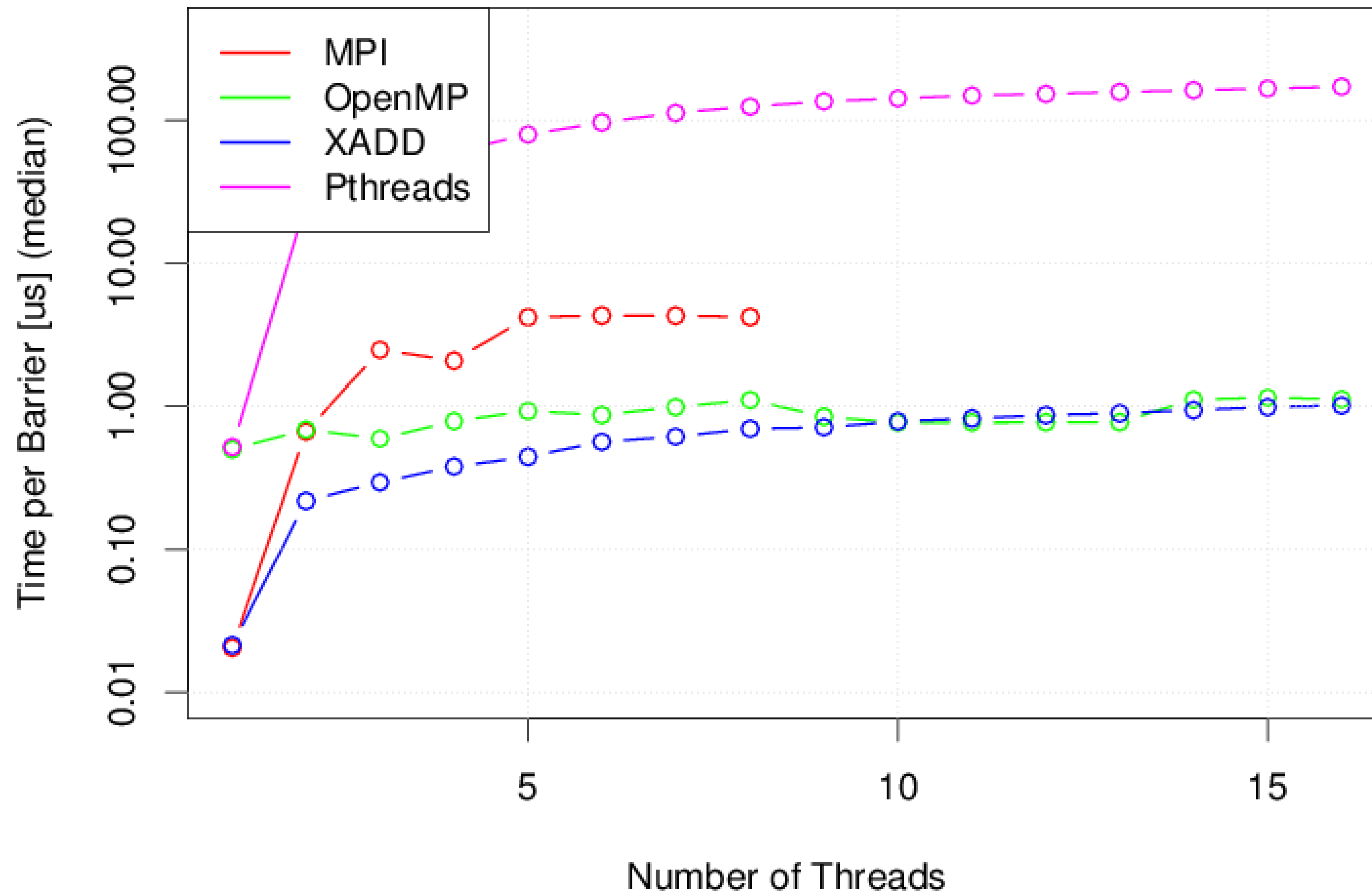
    Problem: when to re-use the counter?

    Cannot just set it to 0 ☹

    Trick: “lock xadd -1” when done ☺

[cf. http://www.spiral.net/software/barrier.html]

Barrier Performance



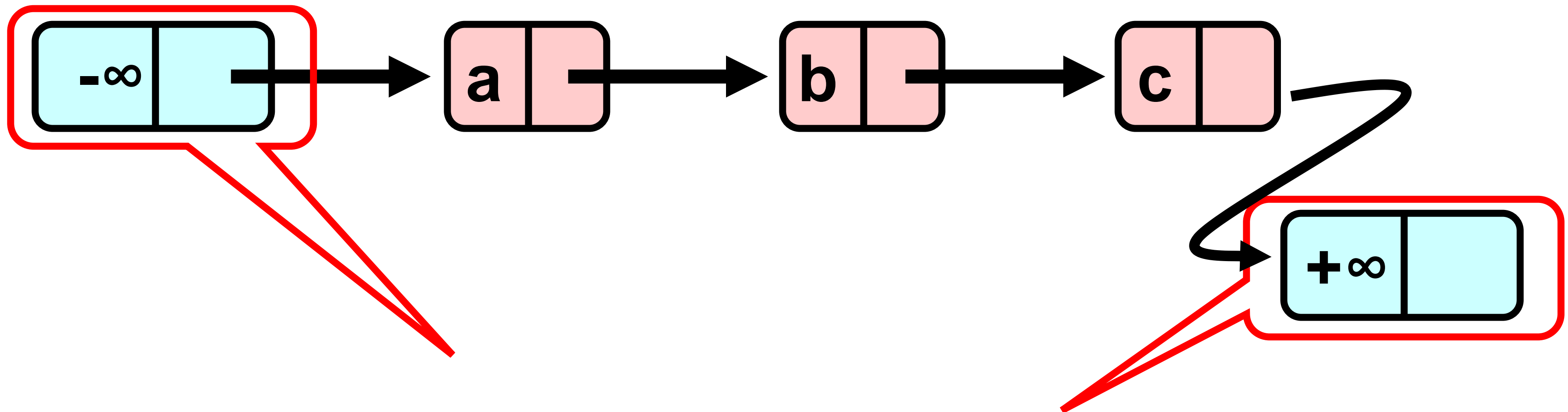
Locks in Practice

- **Running example: List-based set of integers**
  - S.insert(v) – return true if v was inserted
  - S.remove(v) – return true if v was removed
  - S.contains(v) – return true iff v in S
- **Simple ordered linked list**
  - Do not use this at home (poor performance)
  - Good to demonstrate locking techniques

    E.g., skip list would be faster but more complex

Set Structure in Memory

- This and many of the following illustrations are provided by Maurice Herlihy in conjunction with the book "The Art of Multiprocessor Programming"



Sequential Set

```
boolean add(S, x) {
  node *pred = S.head;
  node *curr = pred.next;
  while(curr.key < x) {
    pred = curr;
    curr = pred.next;
  }
  if(curr.key == x)
    return false;
  else {
    node n = new node();
    n.key = x;
    n.next = curr;
    pred.next = n;
  }
  return true;
}
```

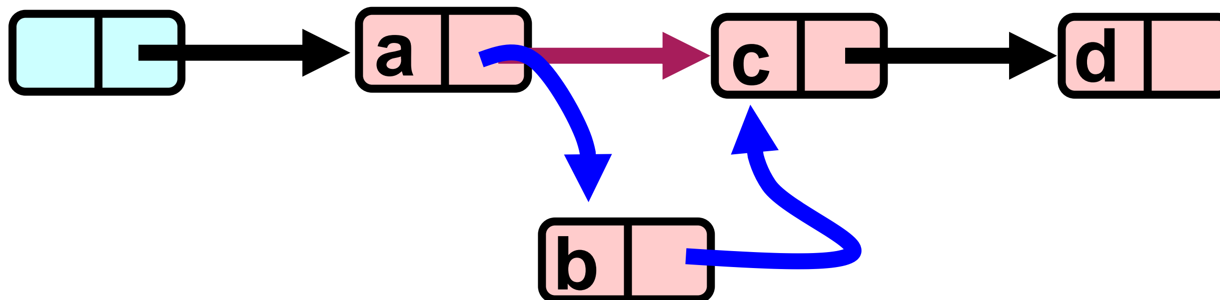
```
boolean remove(S, x) {
  node *pred = S.head;
  node *curr = pred.next;
  while(curr.key < x) {
    pred = curr;
    curr = pred.next;
  }
  if(curr.key == x) {
    pred.next = curr.next;
    free(curr);
    return true;
  }
  return false;
}
```

```
boolean contains(S, x) {
  int *curr = S.head;
  while(curr.key < x)
    curr = curr.next;
  if(curr.key == x)
    return true;
  return false;
}
```

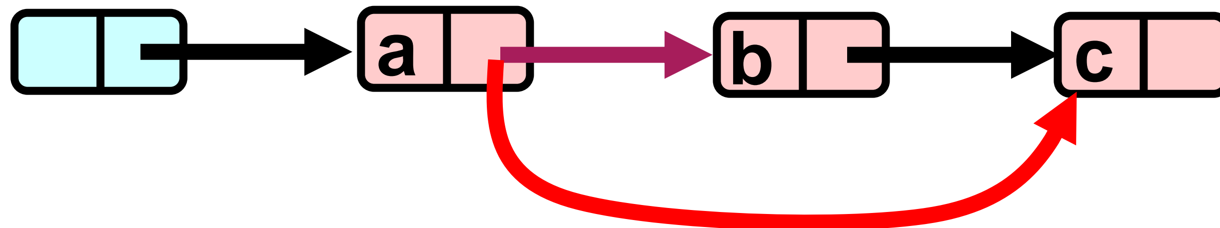
```
typedef struct {
  int key;
  node *next;
} node;
```

Sequential Operations

add ( )



remove ( )



Concurrent Sets

- **What can happen if multiple threads call set operations at the "same time"?**
  - Operations can conflict!
- **Which operations conflict?**
  - (add, remove), (add, add), (remove, remove), (remove, contains) will conflict
  - (add, contains) may miss update (which is fine)
  - (contains, contains) does not conflict
- **How can we fix it?**

Coarse-grained Locking

```
boolean add(S, x) {
  lock(S);
  node *pred = S.head;
  node *curr = pred.next;
  while(curr.key < x) {
    pred = curr;
    curr = pred.next;
  }
  if(curr.key == x)
    unlock(S);
    return false;
  else {
    node node = malloc();
    node.key = x;
    node.next = curr;
    pred.next = node;
  }
  unlock(S);
  return true;
}
```

```
boolean remove(S, x) {
  lock(S);
  node *pred = S.head;
  node *curr = pred.next;
  while(curr.key < x) {
    pred = curr;
    curr = pred.next;
  }
  if(curr.key == x) {
    pred.next = curr.next;
    unlock(S);
    free(curr);
    return true;
  }
  unlock(S);
  return false;
}
```

```
boolean contains(S, x) {
  lock(S);
  int *curr = S.head;
  while(curr.key < x)
    curr = curr.next;
  if(curr.key == x)  {
    unlock(S);
    return true;
  }
  unlock(S);
  return false;
}
```

Coarse-grained Locking

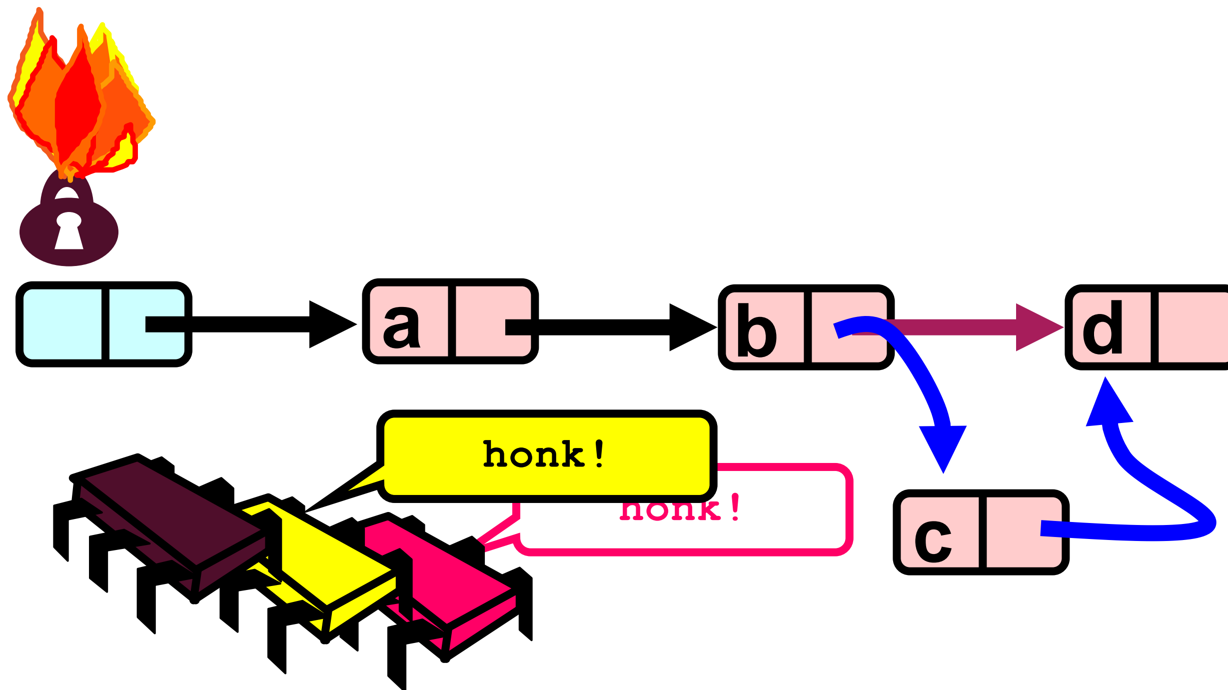
- **Correctness proof?**
  - Assume sequential version is correct

    Alternative: define set of invariants and proof that initial condition as well as all transformations adhere
  - Proof that all accesses to shared data are in CRs

    This may prevent some optimizations
- **Is the algorithm deadlock-free? Why?**
  - Locks are acquired in the same order (only one lock)
- **Is the algorithm starvation-free and/or fair? Why?**
  - It depends on the properties of the used locks!

Coarse-grained Locking

- Is the algorithm performing well with many concurrent threads accessing it?



Simple but hotspot + bottleneck

Coarse-grained Locking

- **Is the algorithm performing well with many concurrent threads accessing it?**
  - No, access to the whole list is serialized
- **BUT: it’s easy to implement and proof correct**
  - Those benefits should **never** be underestimated
  - May be just good enough
  - *“We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified”* — Donald Knuth (in *Structured Programming with Goto Statements*)

How to Improve?

- **Will present some “tricks”**
  - Apply to the list example
  - But often generalize to other algorithms
  - Remember the trick, not the example!
- **See them as “concurrent programming patterns” (not literally)**
  - Good toolbox for development of concurrent programs
  - They become successively more complex

Tricks Overview

1. **Fine-grained locking**
   - Split object into “lockable components”
   - Guarantee mutual exclusion for conflicting accesses to same component
2. **Reader/writer locking**
3. **Optimistic synchronization**
4. **Lazy locking**
5. **Lock-free**

Tricks Overview

1. **Fine-grained locking**
2. **Reader/writer locking**
   - Multiple readers hold lock (traversal)
   - contains() only needs read lock
   - Locks may be upgraded during operation

   Must ensure starvation-freedom for writer locks!
3. **Optimistic synchronization**
4. **Lazy locking**
5. **Lock-free**

Tricks Overview

1. **Fine-grained locking**
2. **Reader/writer locking**
3. **Optimistic synchronization**
   - Traverse without locking

     Need to make sure that this is correct!
   - Acquire lock if update necessary

     May need re-start from beginning, tricky
4. **Lazy locking**
5. **Lock-free**

Tricks Overview

1. **Fine-grained locking**
2. **Reader/writer locking**
3. **Optimistic synchronization**
4. **Lazy locking**
   - Postpone hard work to idle periods
   - Mark node deleted
     Delete it physically later
5. **Lock-free**

Tricks Overview

1. **Fine-grained locking**
2. **Reader/writer locking**
3. **Optimistic synchronization**
4. **Lazy locking**
5. **Lock-free**
   - Completely avoid locks
   - Enables wait-freedom
   - Will need atomics (see later why!)
   - Often very complex, sometimes higher overhead

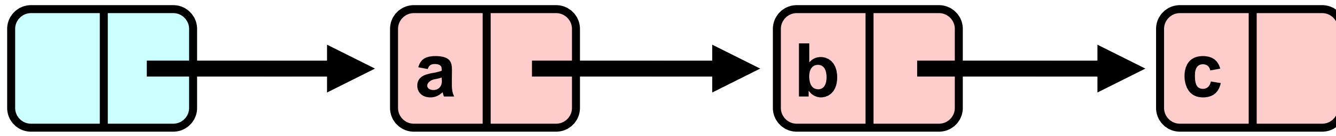
Trick 1: Fine-grained Locking

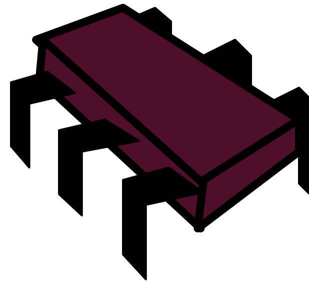
- **Each element can be locked**
  - High memory overhead
  - Threads can traverse list concurrently like a pipeline
- **Tricky to prove correctness**
  - And deadlock-freedom
  - Two-phase locking (acquire, release) often helps
- **Hand-over-hand (coupled locking)**
  - Not safe to release x's lock before acquiring x.next's lock

    will see why in a minute
  - Important to acquire locks in the same order

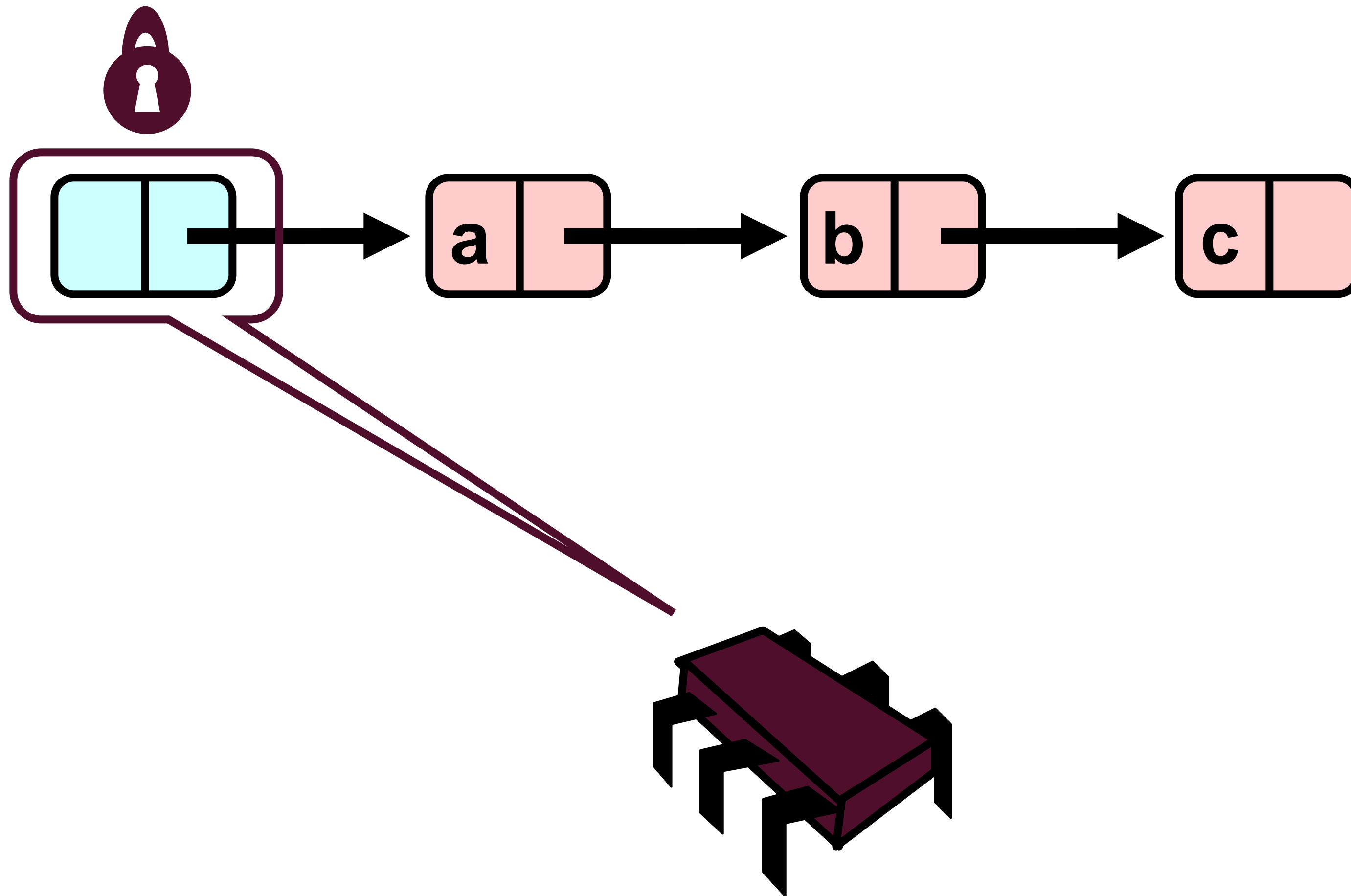
```
typedef struct {
  int key;
  node *next;
  lock_t lock;
} node;
```

Hand-over-Hand (fine-grained) locking

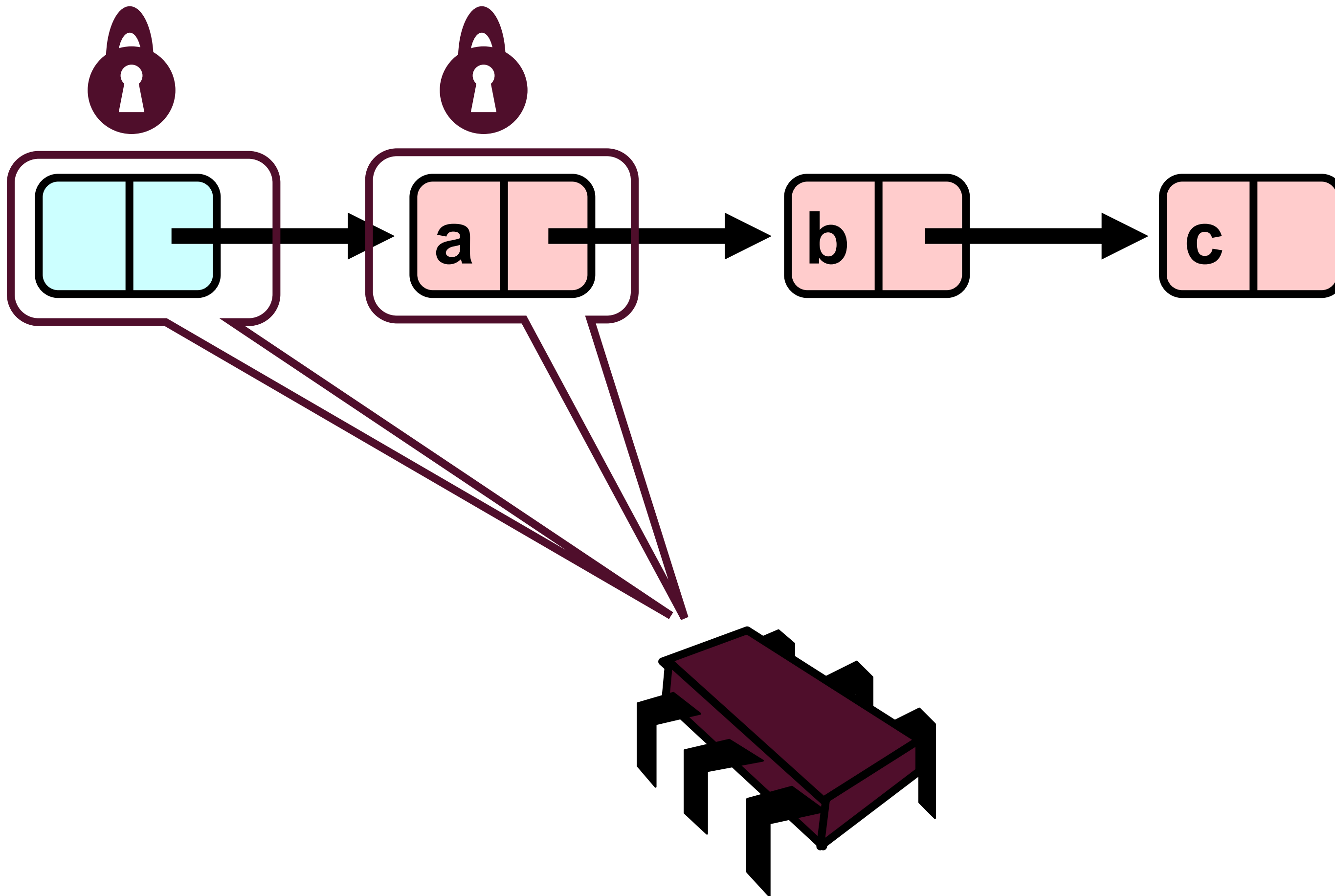




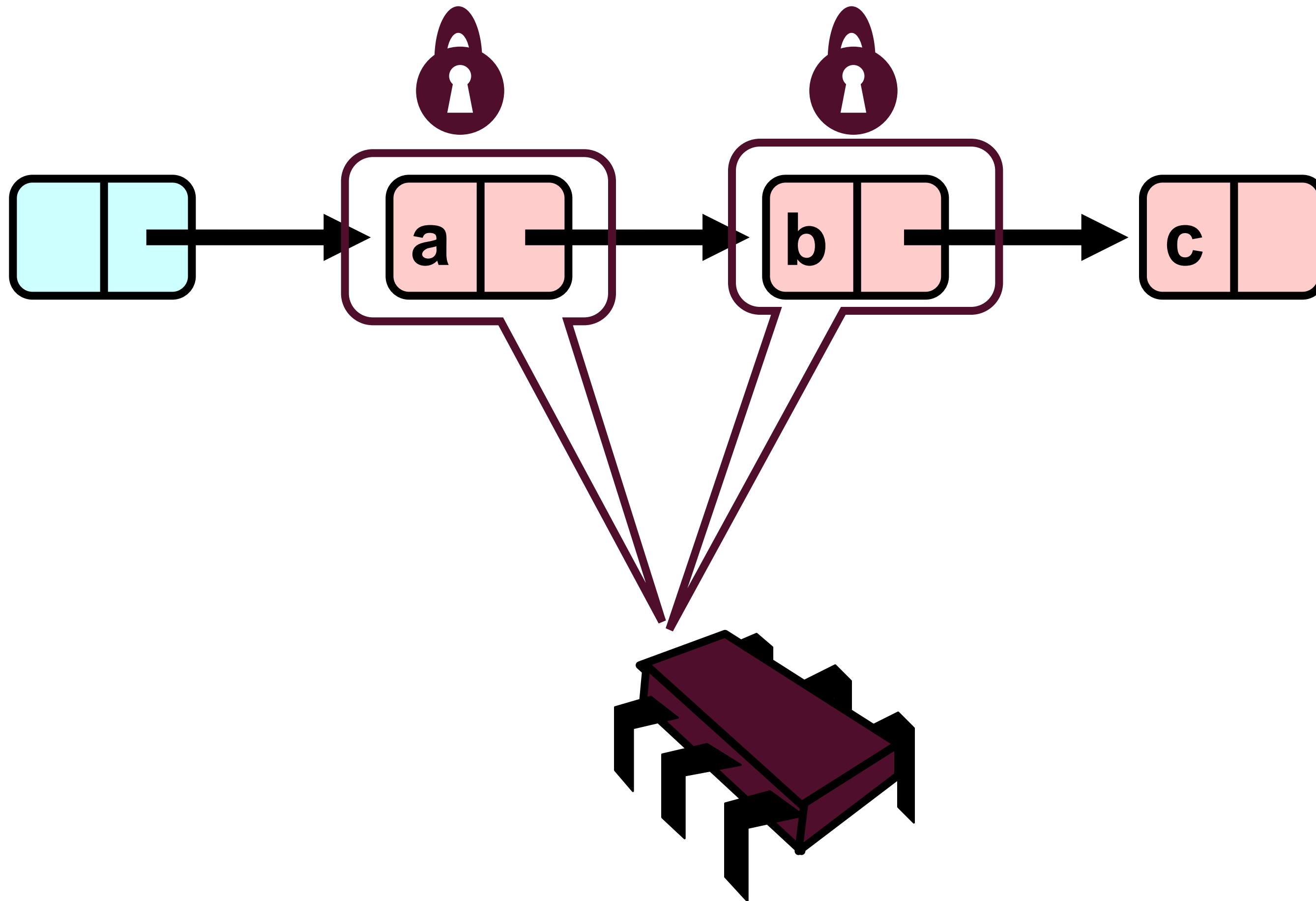
Hand-over-Hand (fine-grained) locking



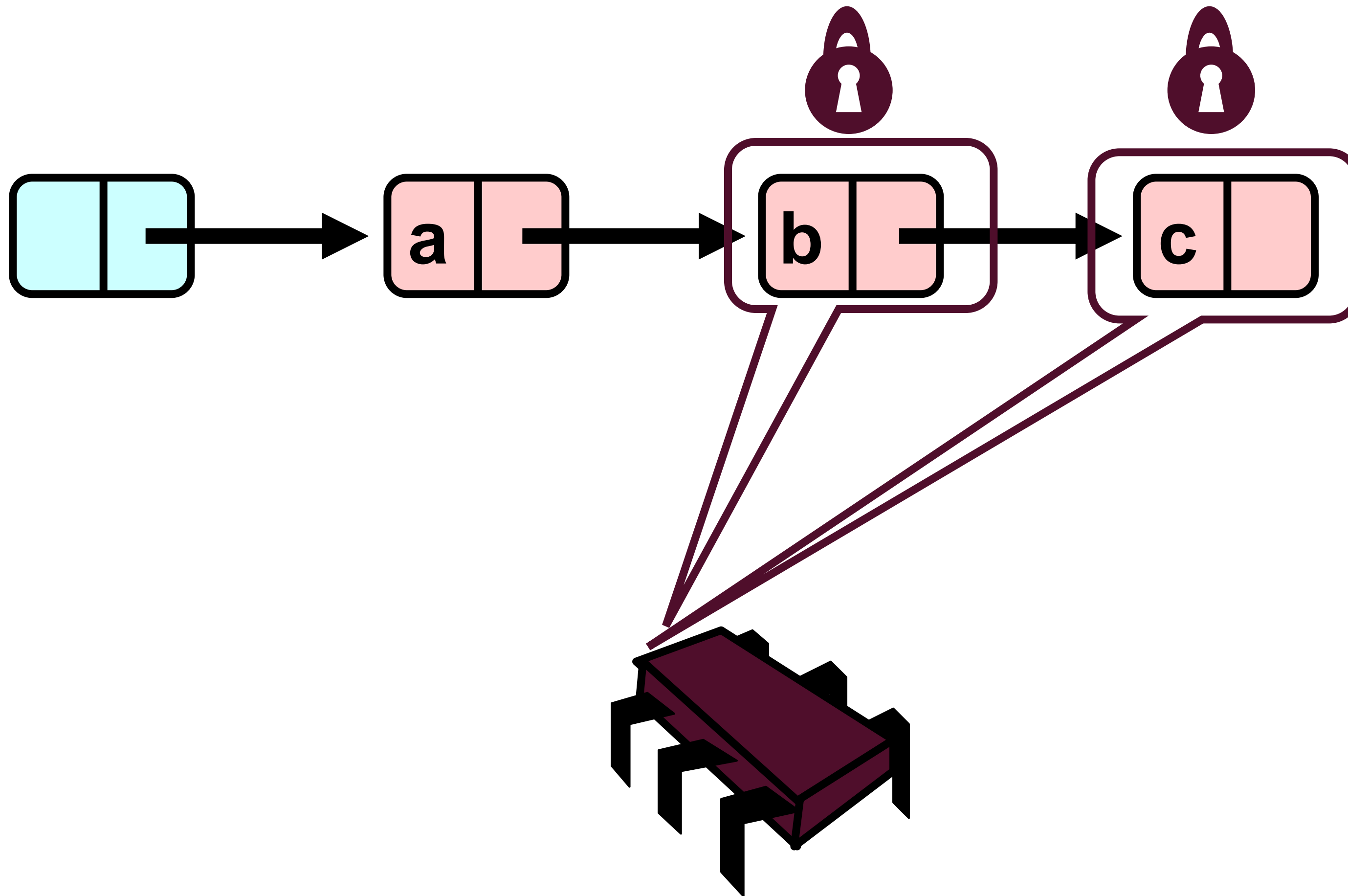
Hand-over-Hand (fine-grained) locking



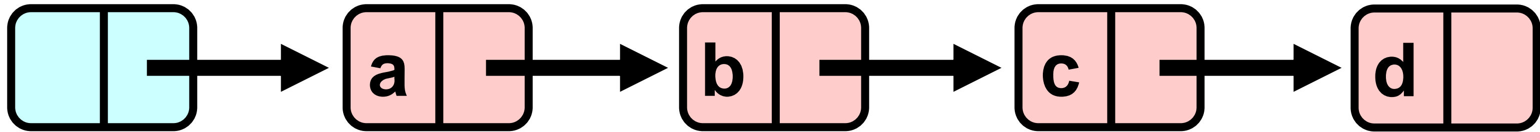
Hand-over-Hand (fine-grained) locking



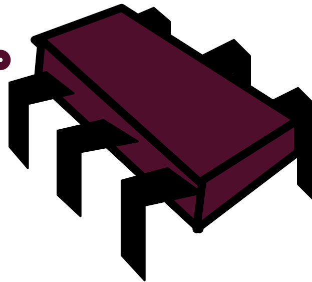
Hand-over-Hand (fine-grained) locking



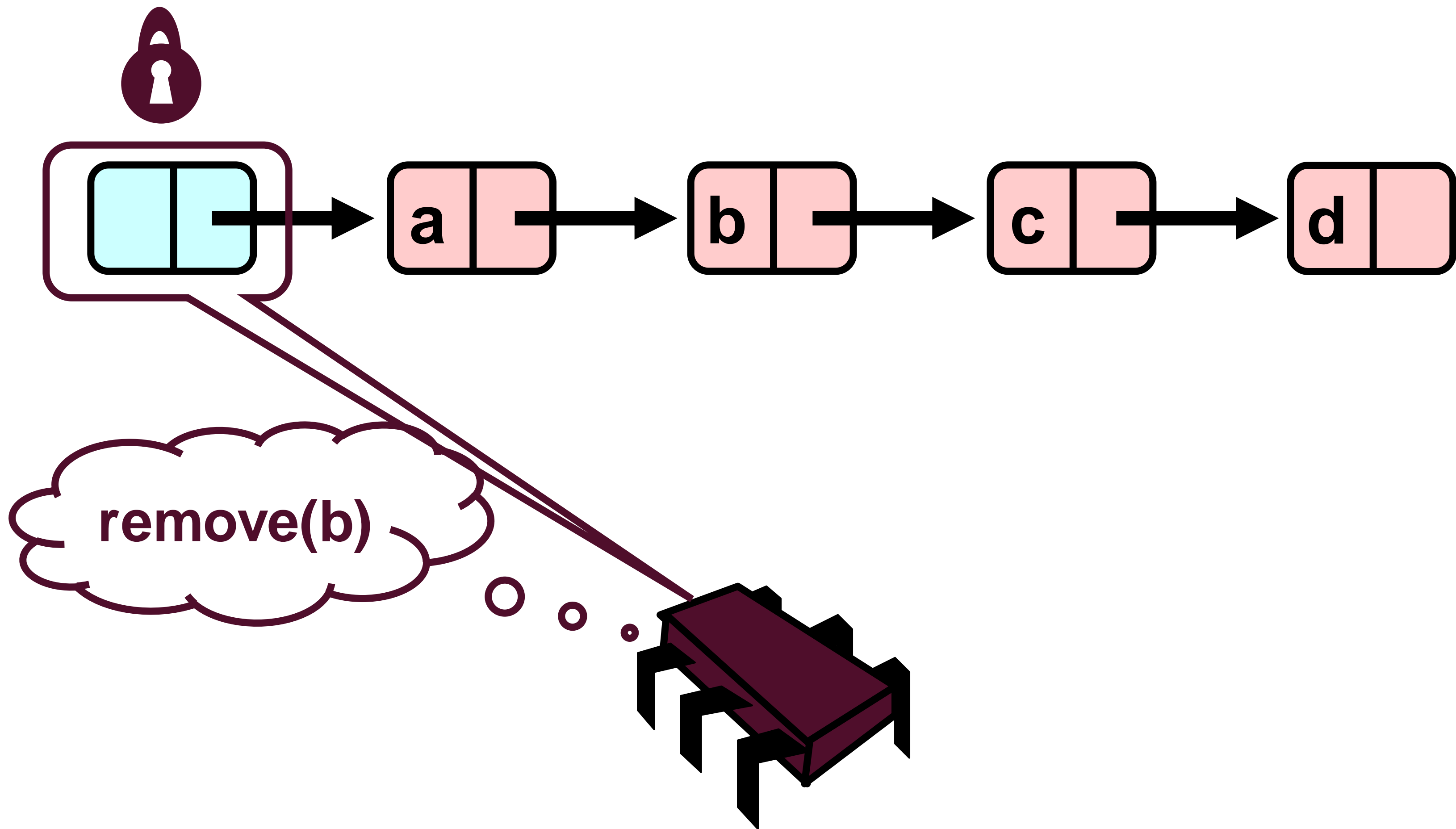
Removing a Node



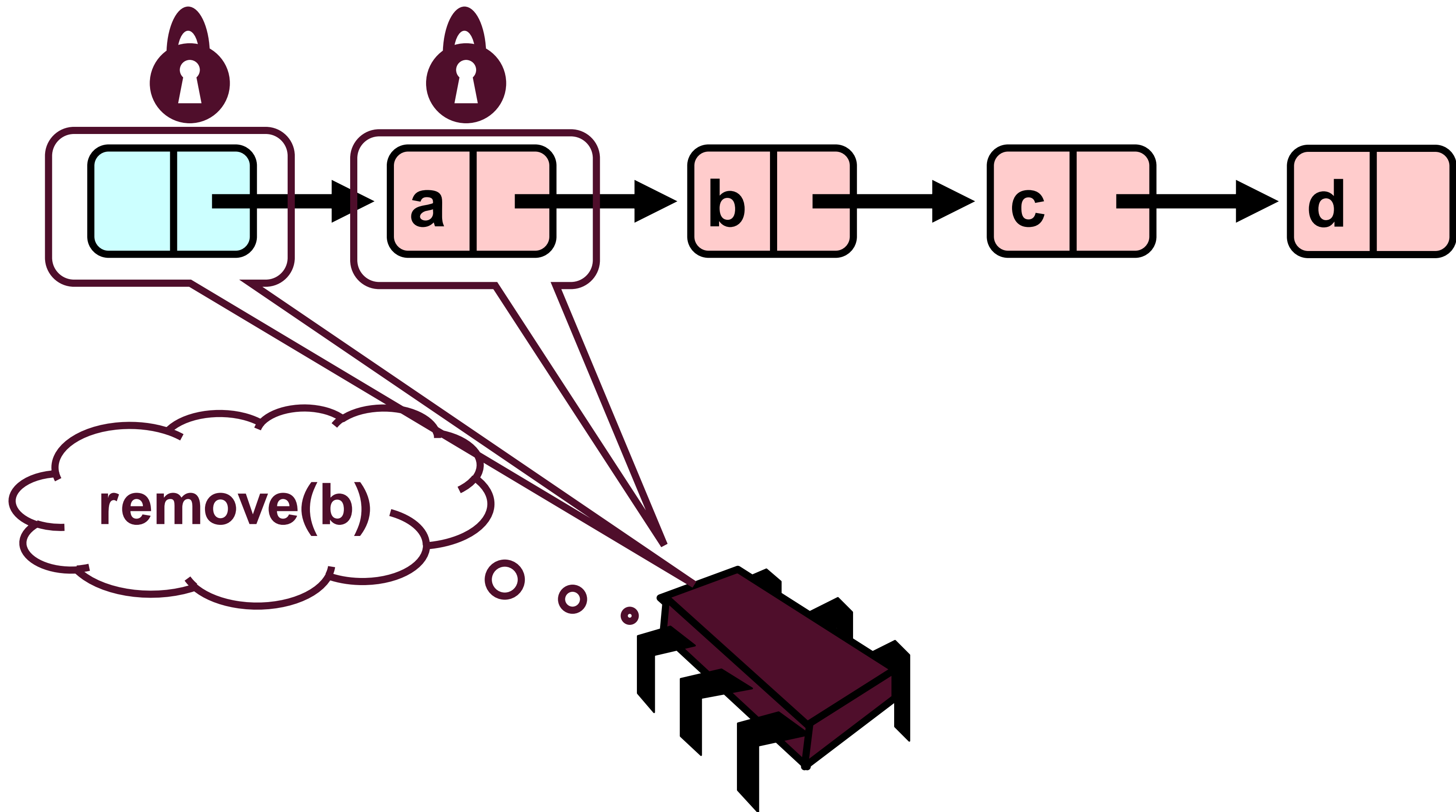
remove(b)



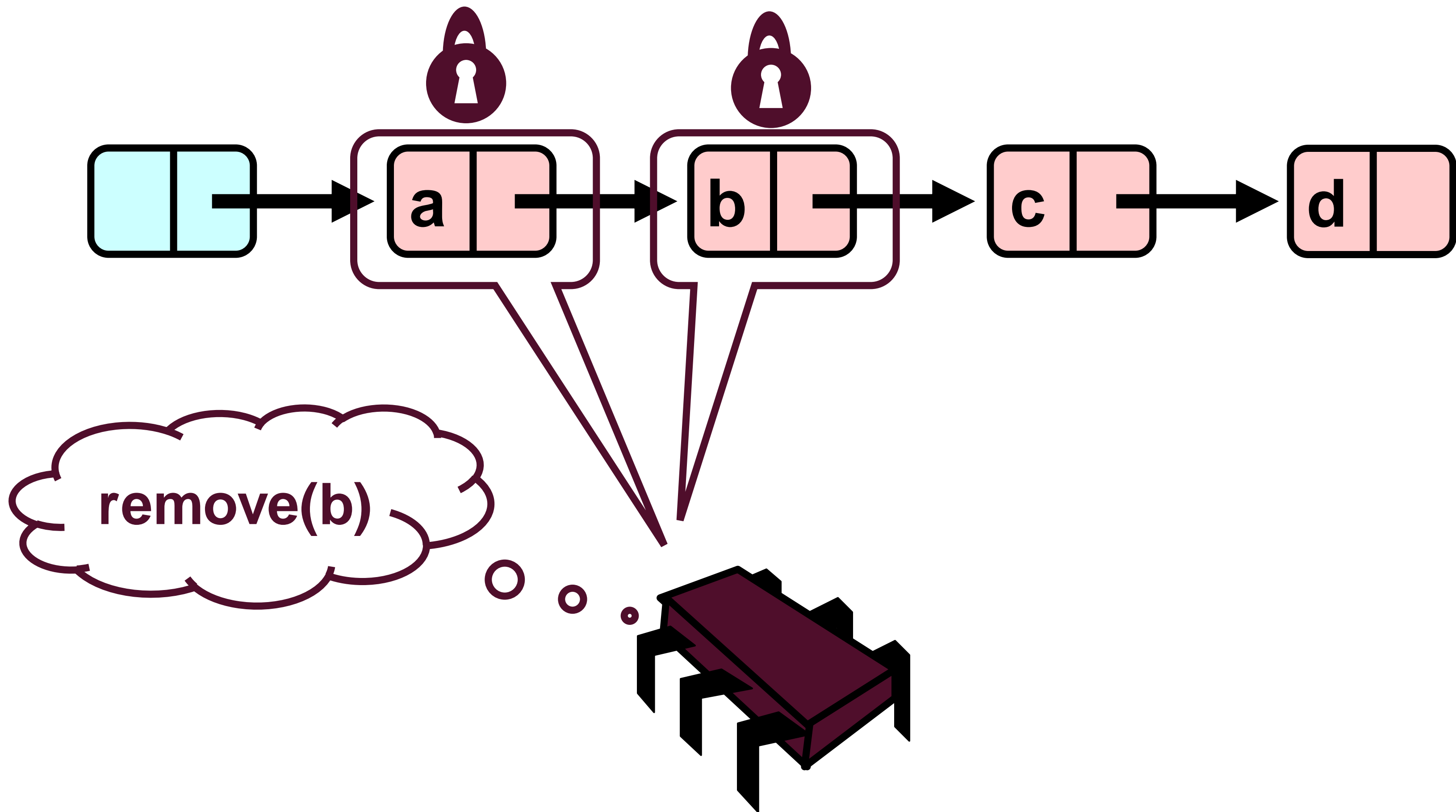
Removing a Node



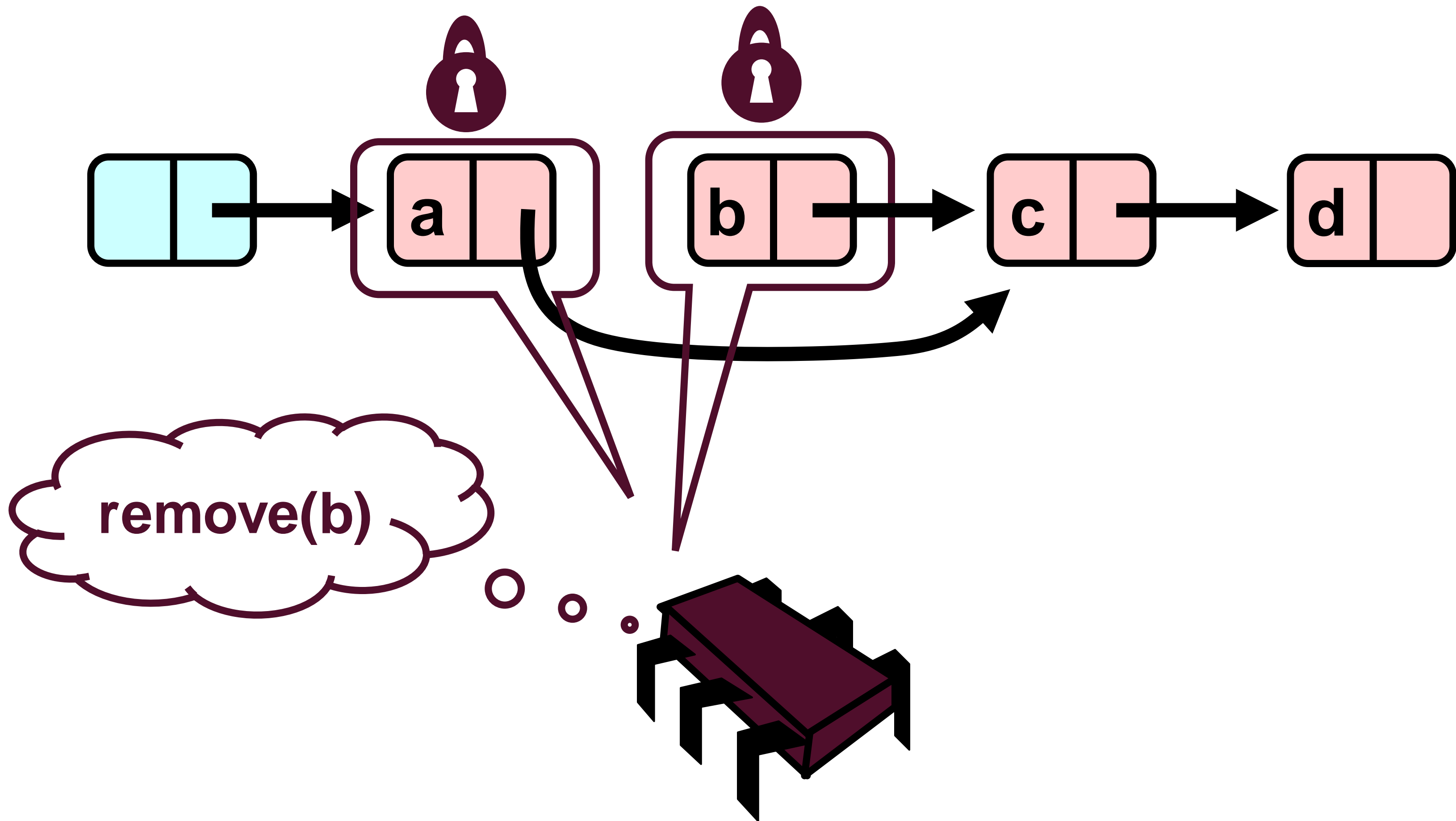
Removing a Node



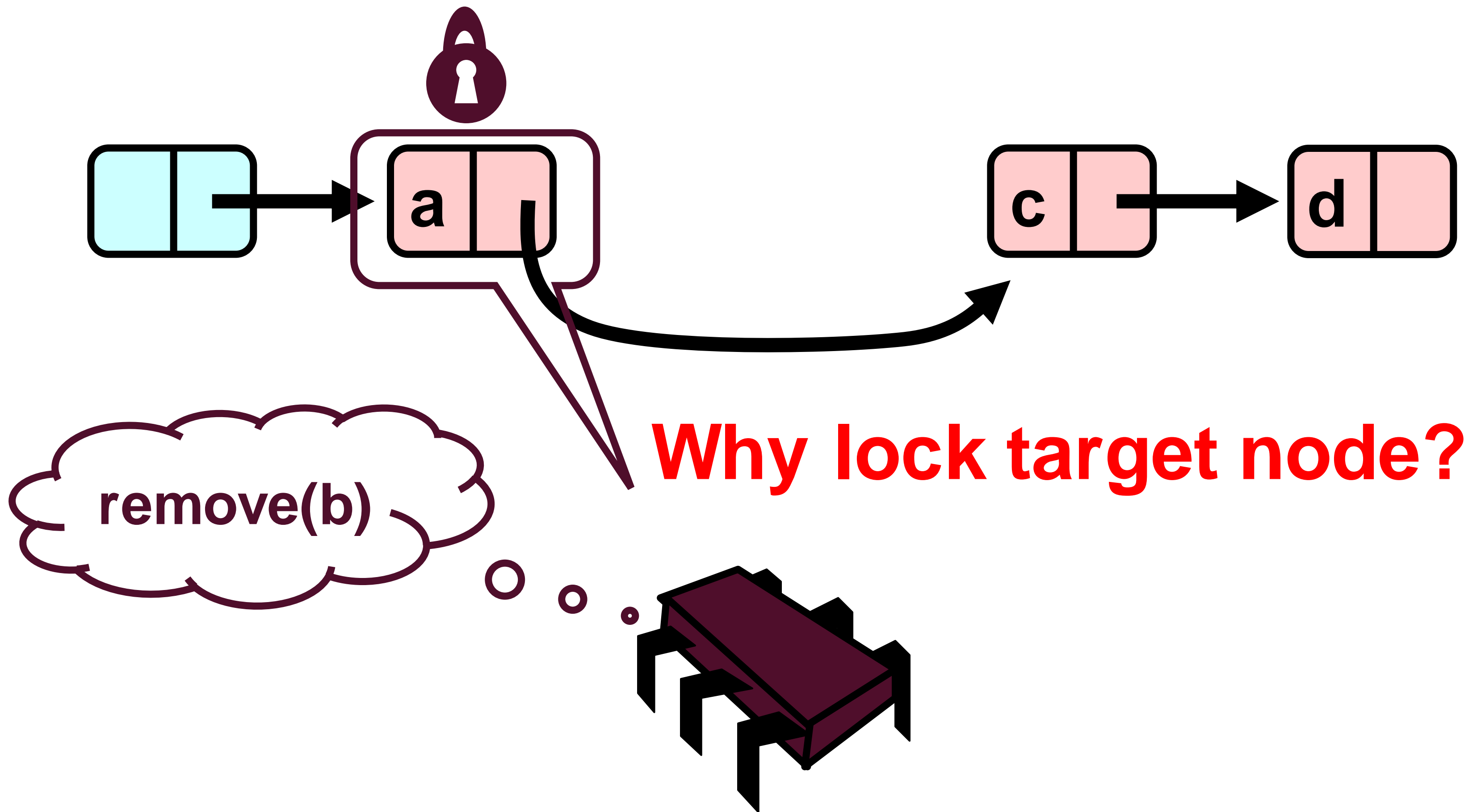
Removing a Node



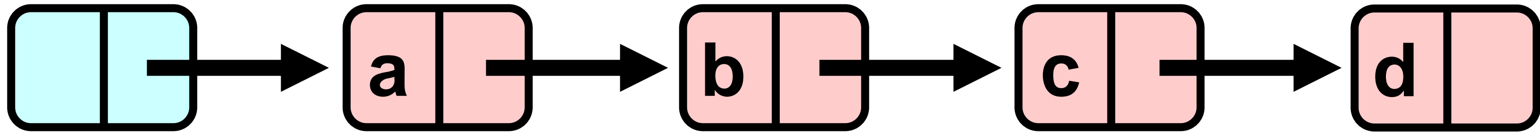
Removing a Node



Removing a Node

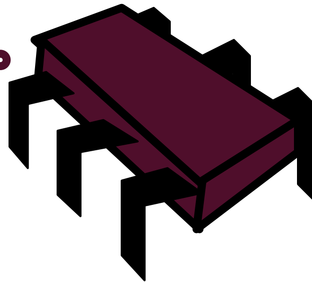


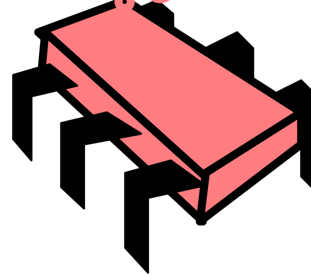
Concurrent Removes



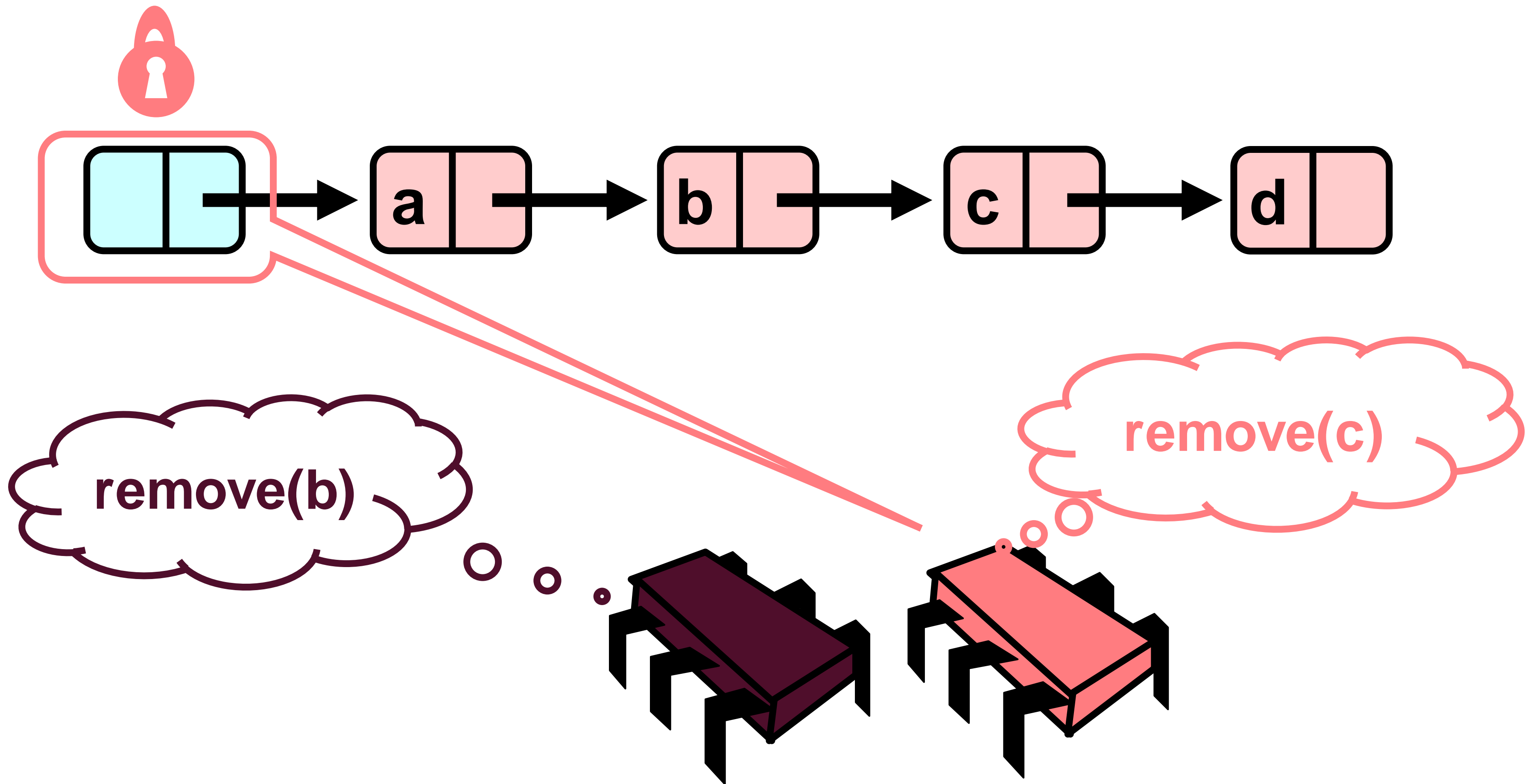
remove(b)

remove(c)

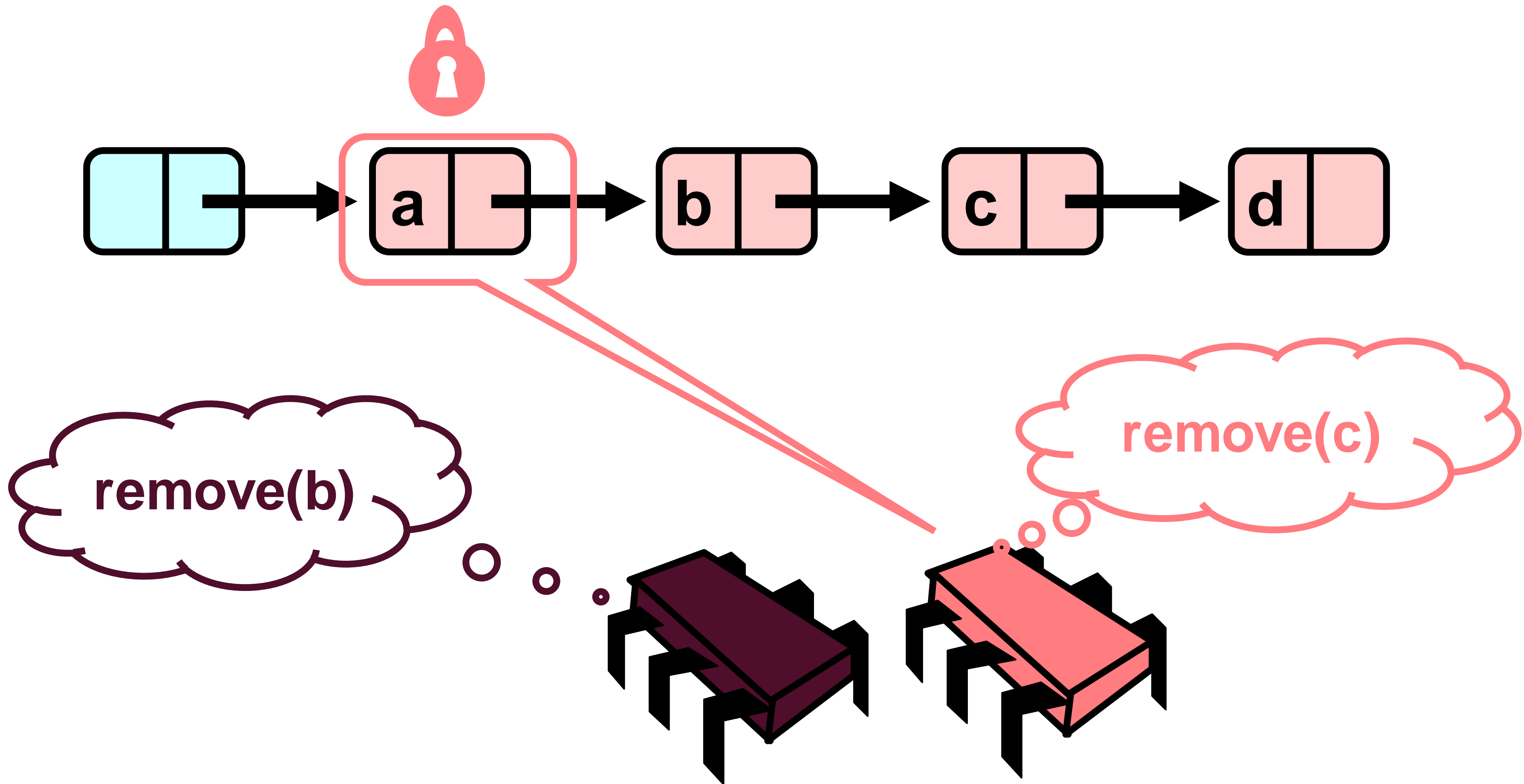




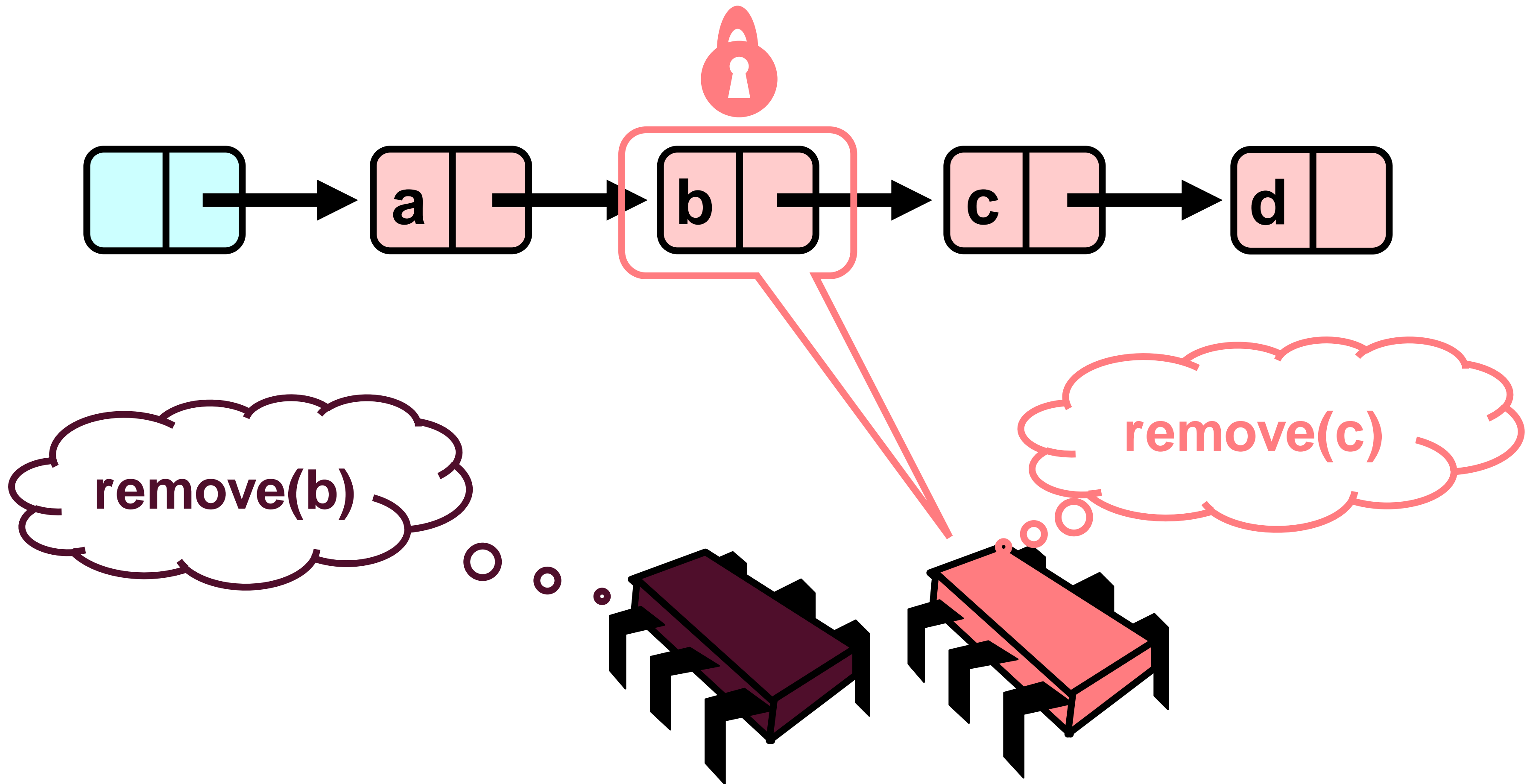
Concurrent Removes



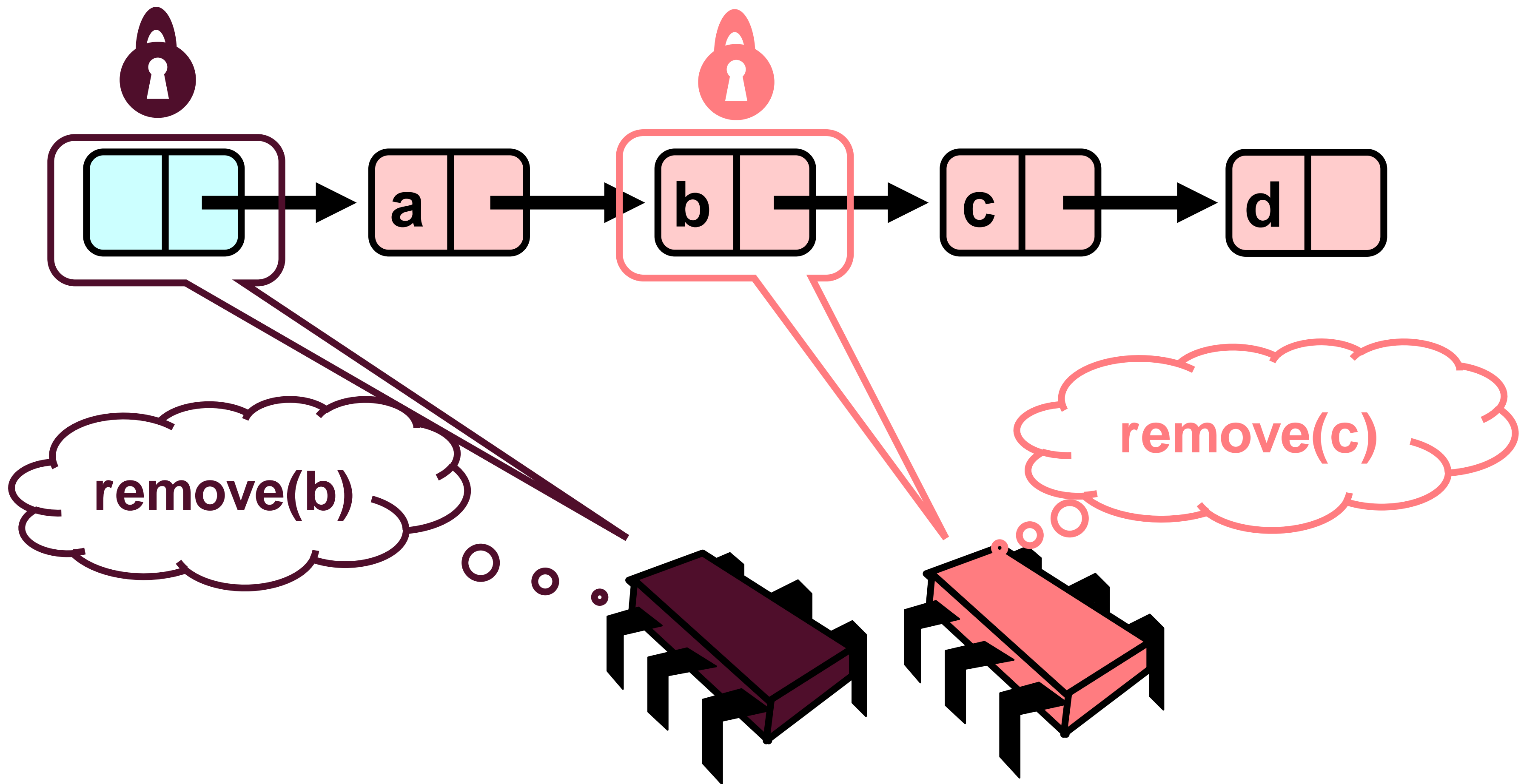
Concurrent Removes



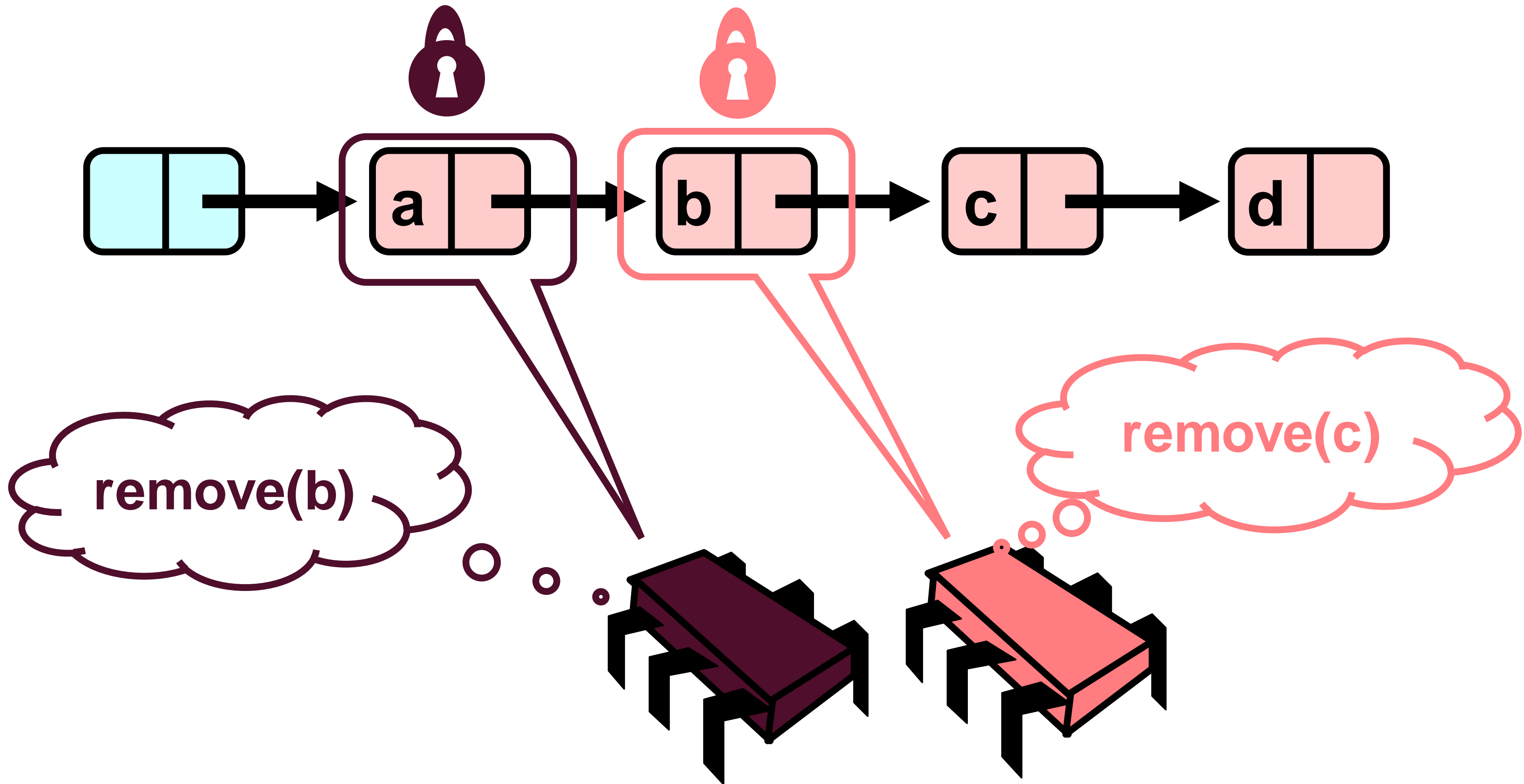
Concurrent Removes



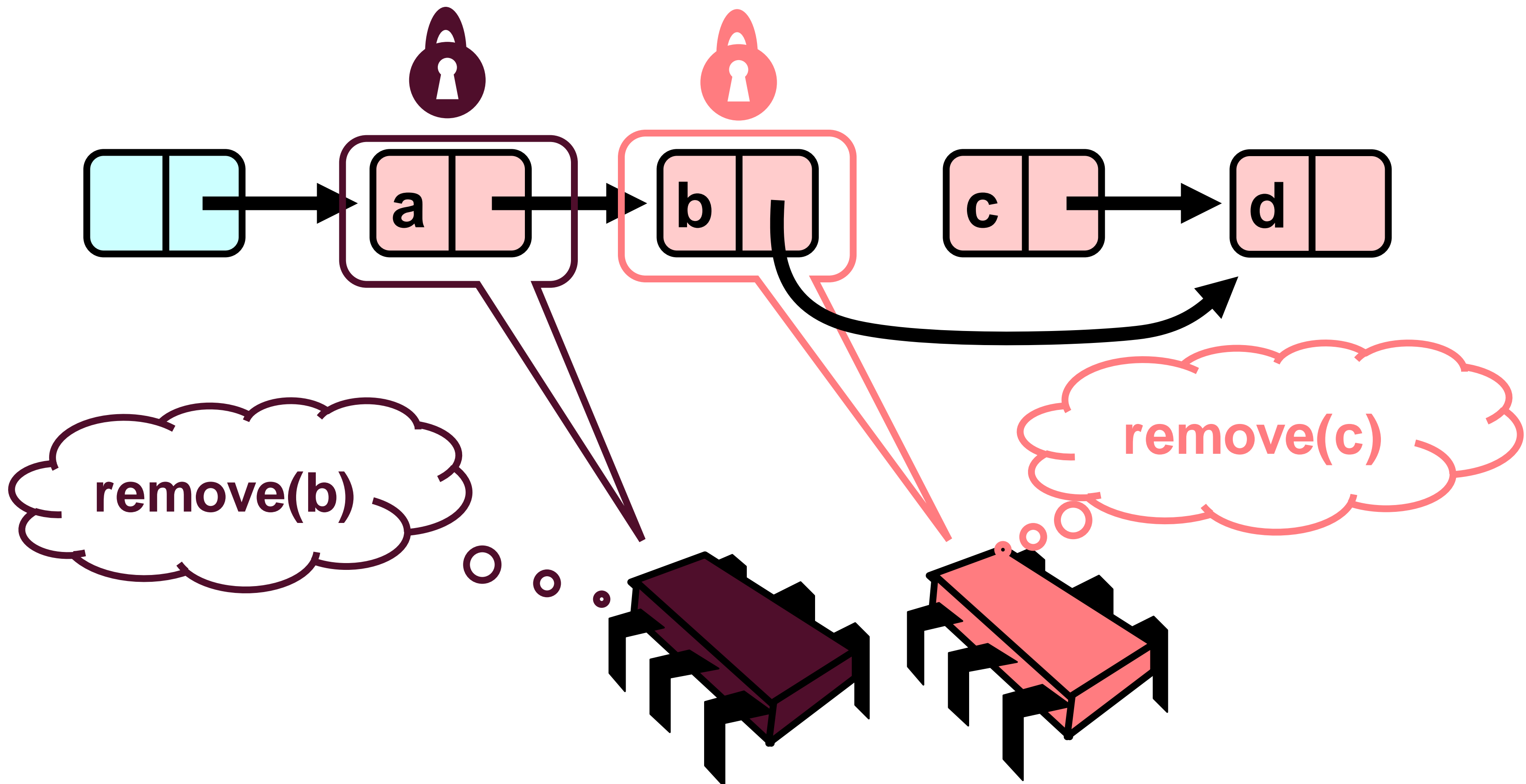
Concurrent Removes



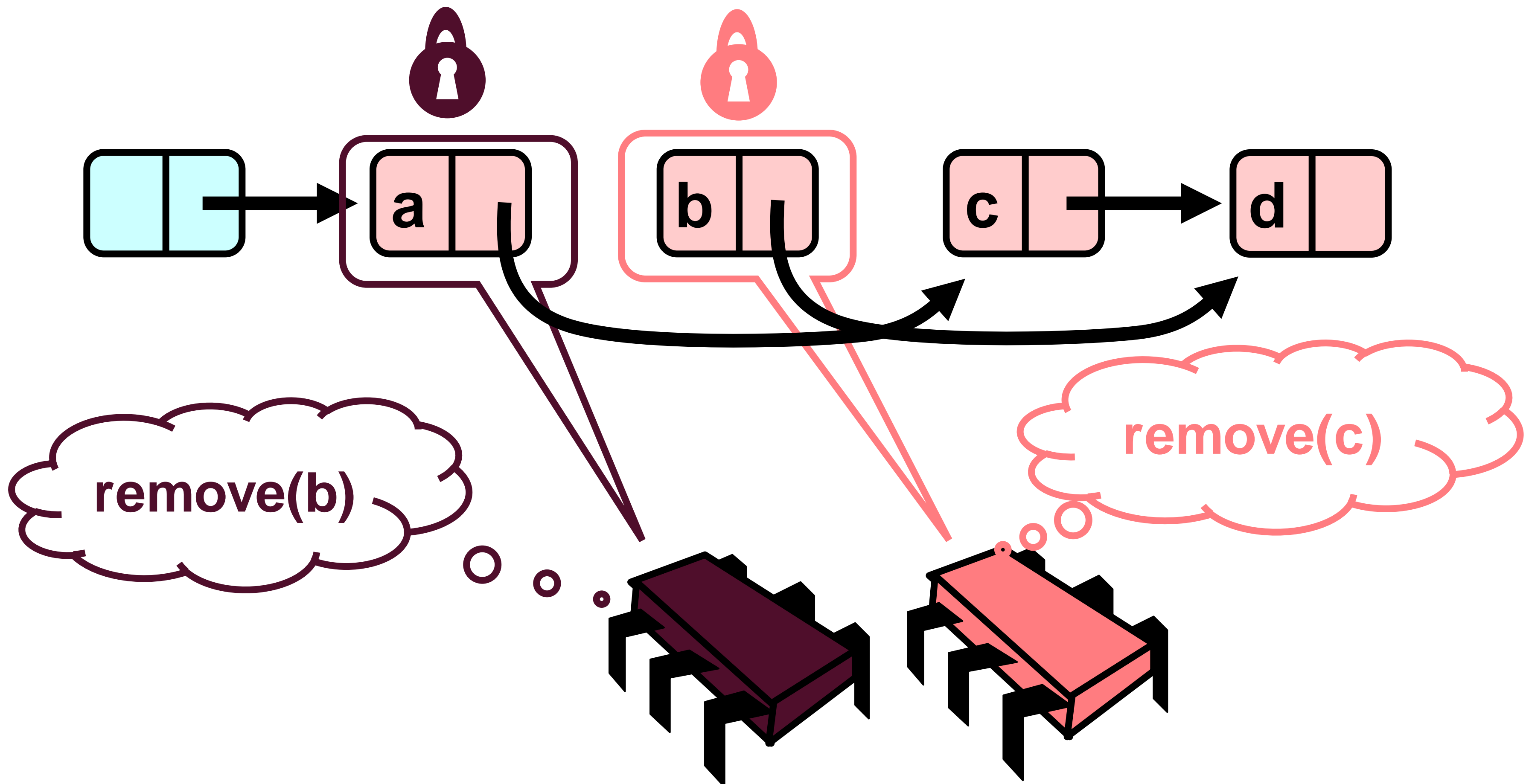
Concurrent Removes



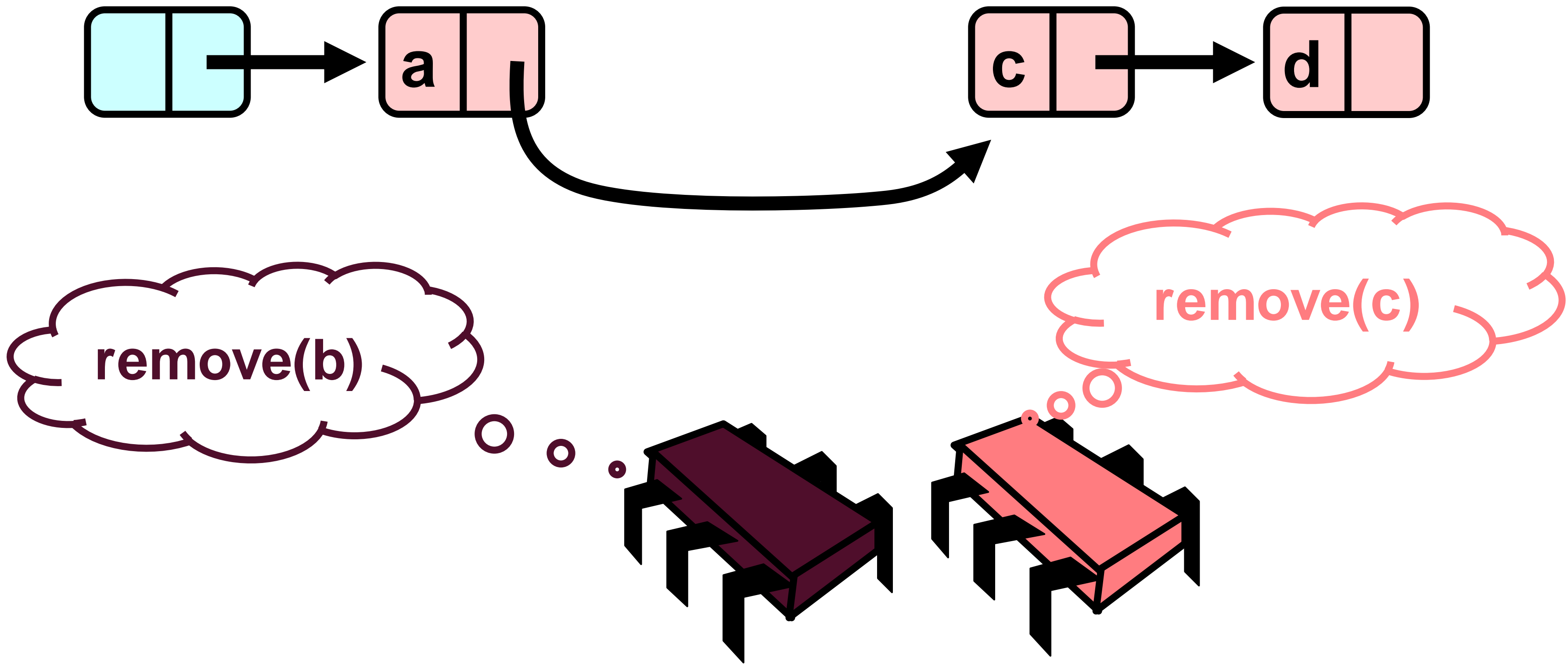
Concurrent Removes



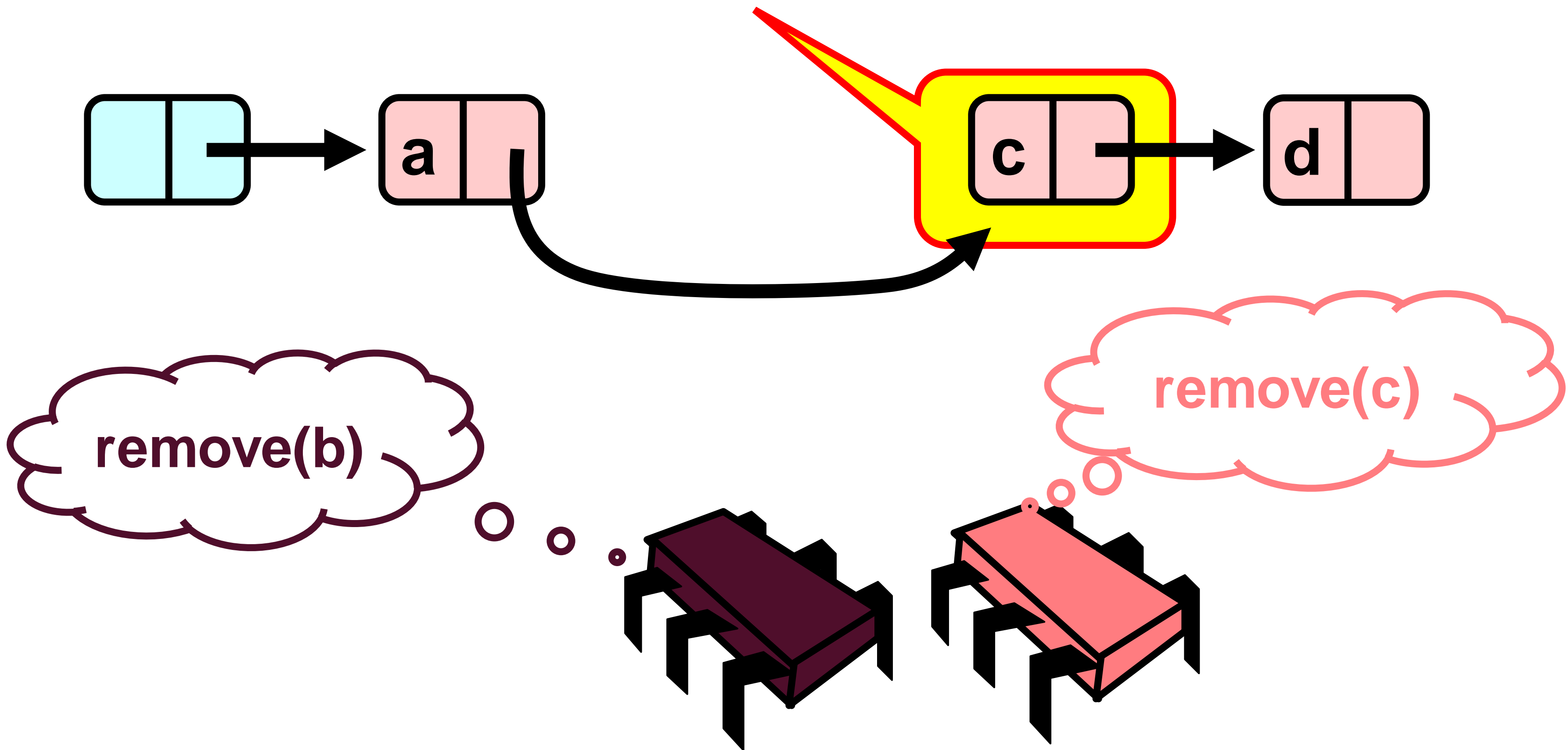
Concurrent Removes



Uh, Oh



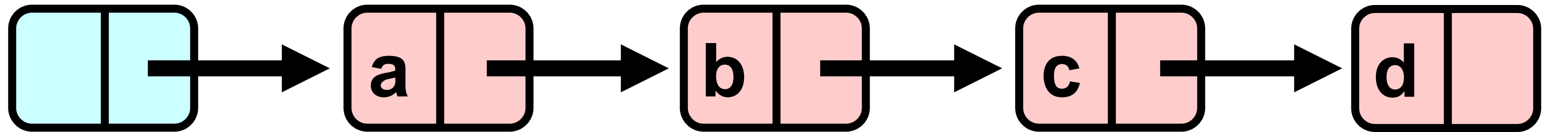
Uh, Oh



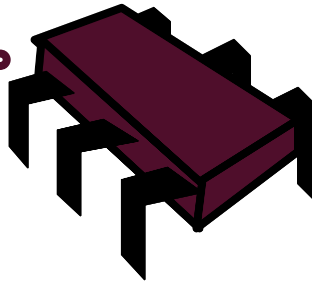
Insight

- **If a node x is locked**
  - Successor of x cannot be deleted!
- **Thus, safe locking is**
  - Lock node to be deleted
  - And its predecessor!
  - → hand-over-hand locking

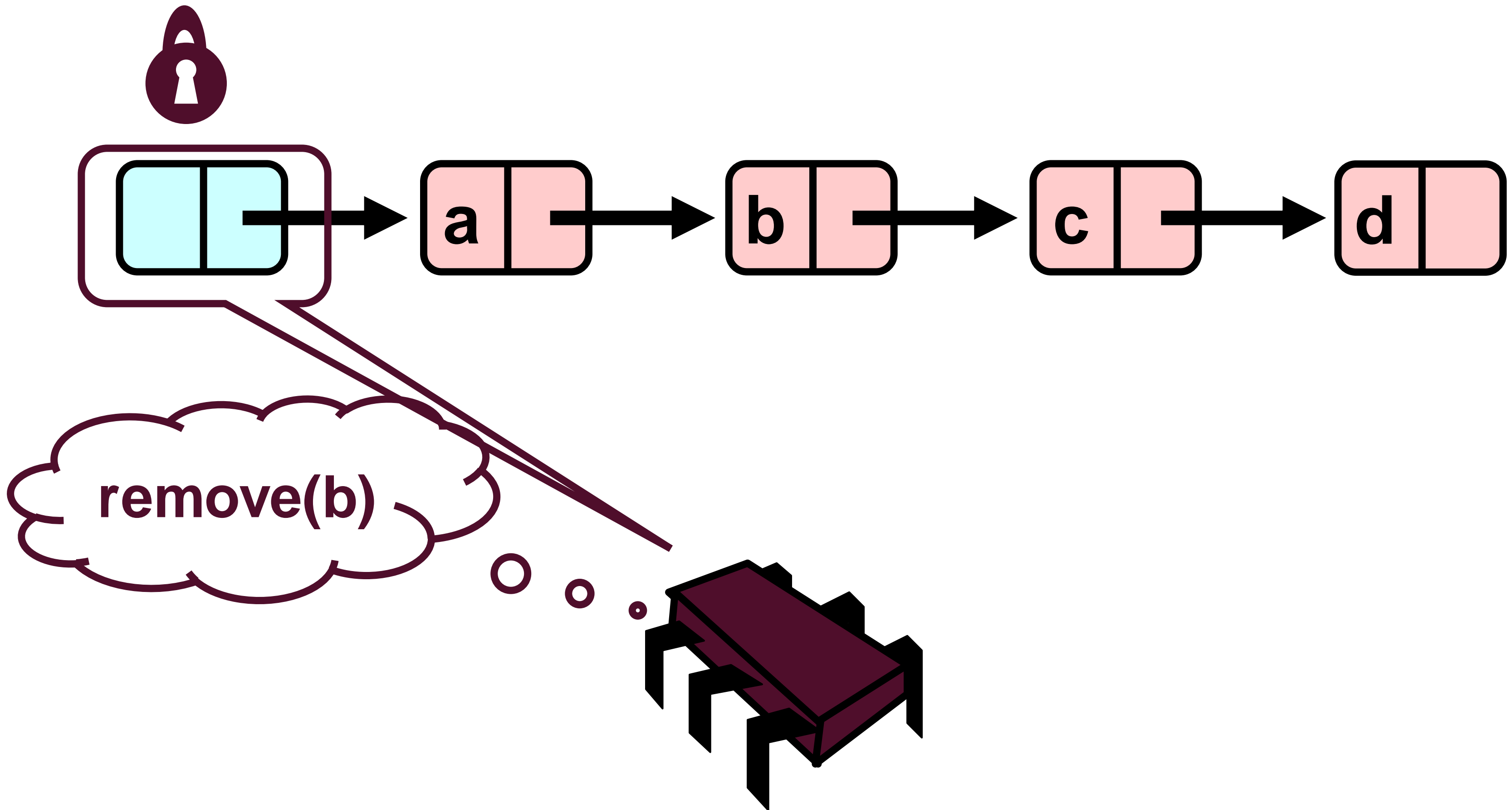
Hand-Over-Hand Again



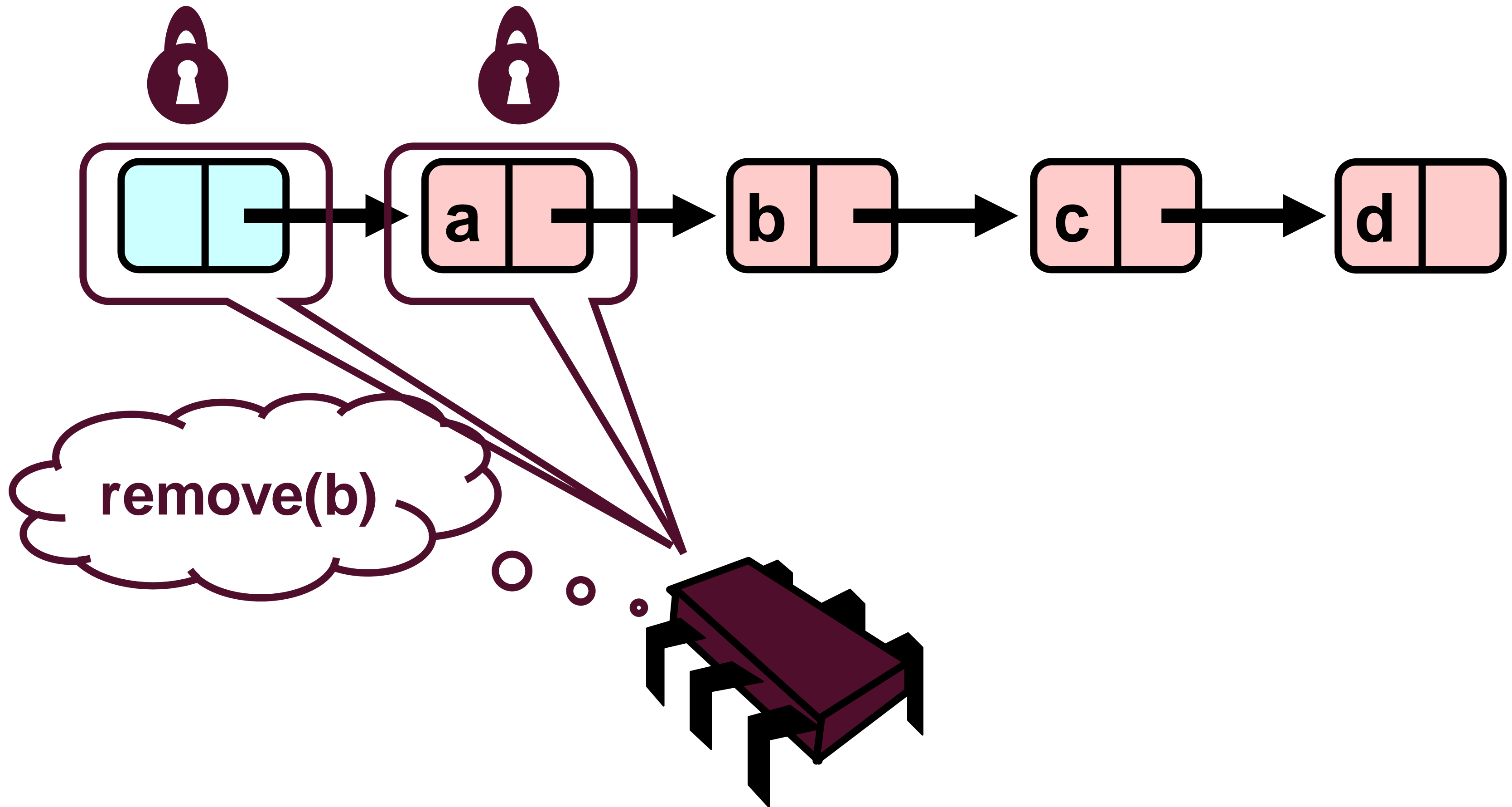
remove(b)



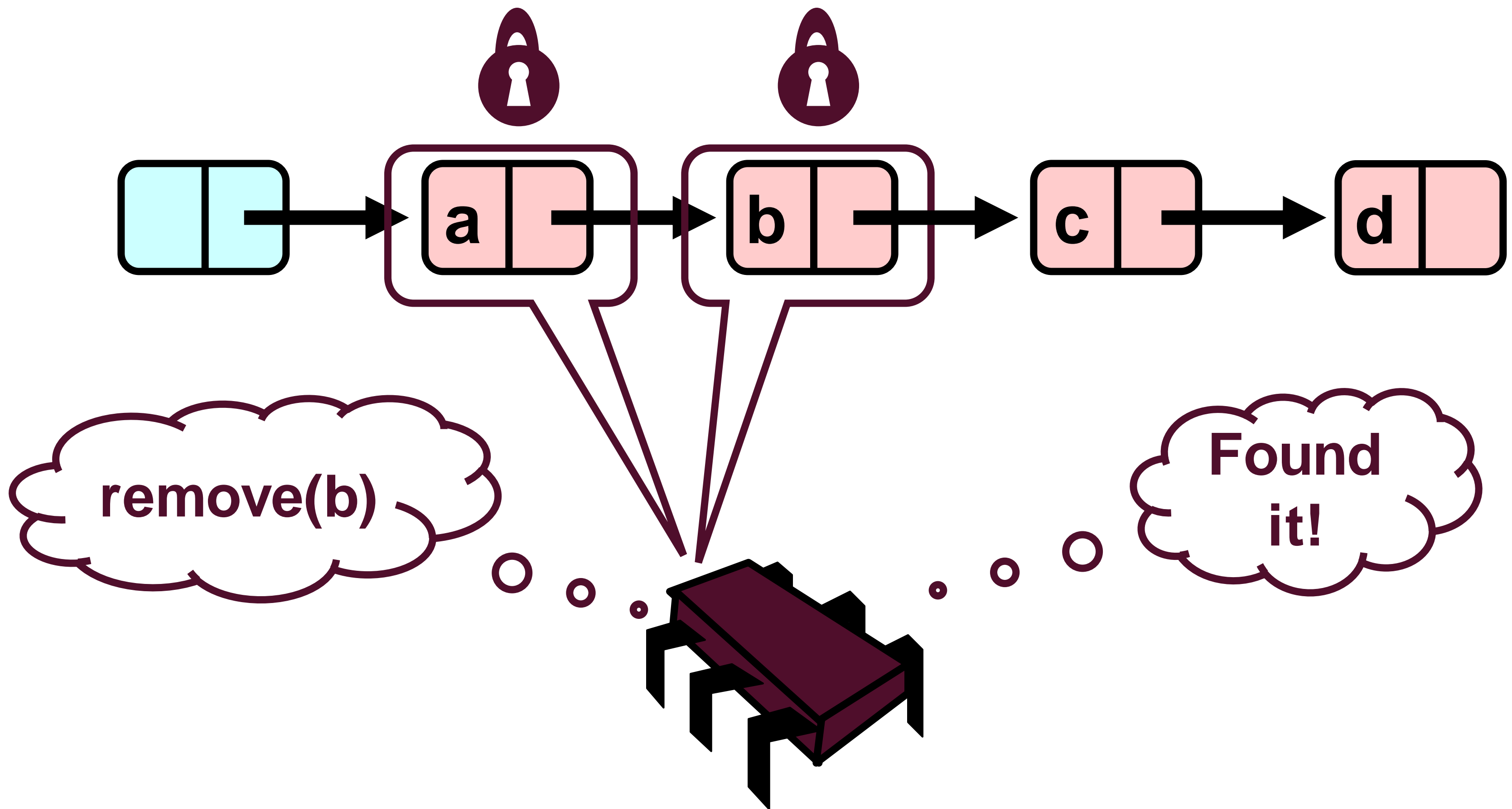
Hand-Over-Hand Again



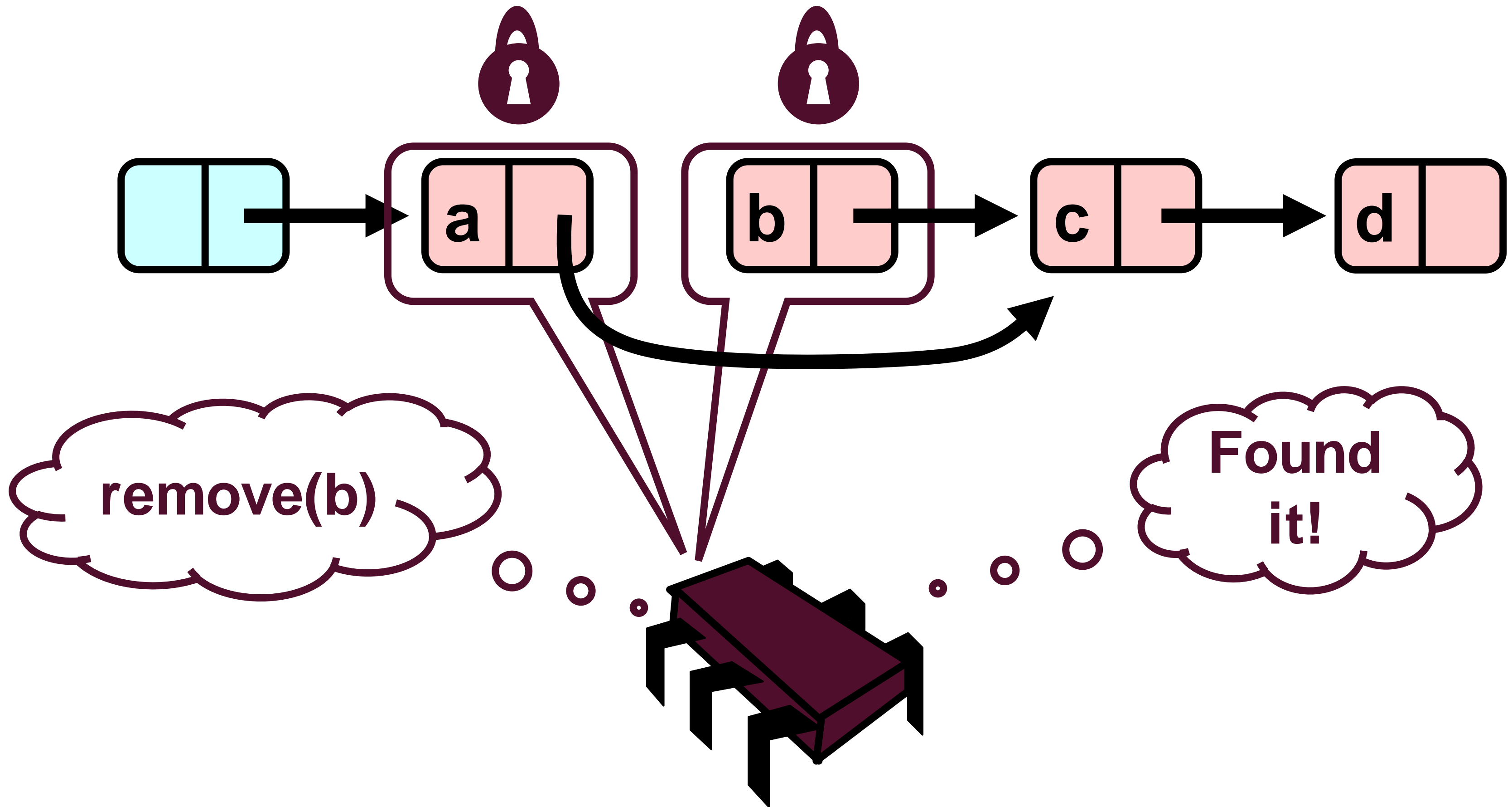
Hand-Over-Hand Again



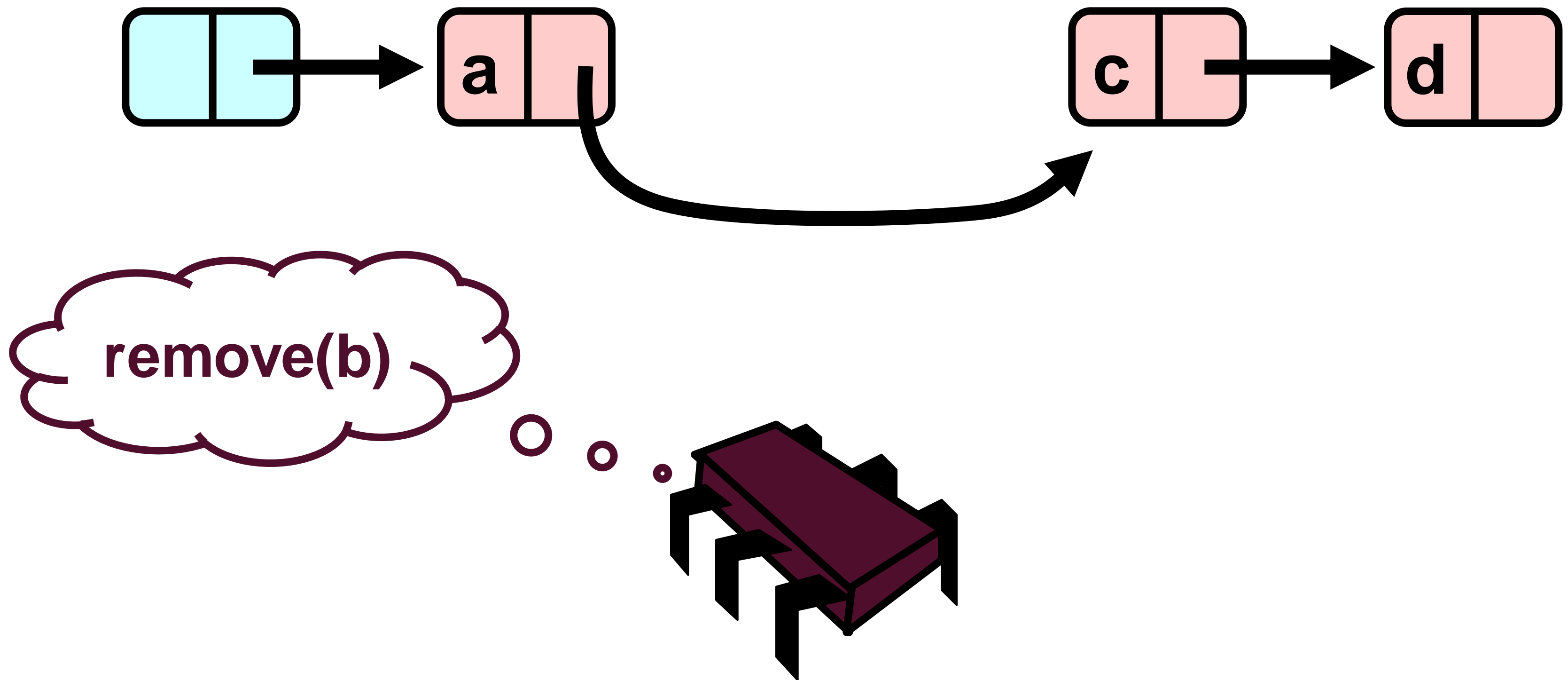
Hand-Over-Hand Again



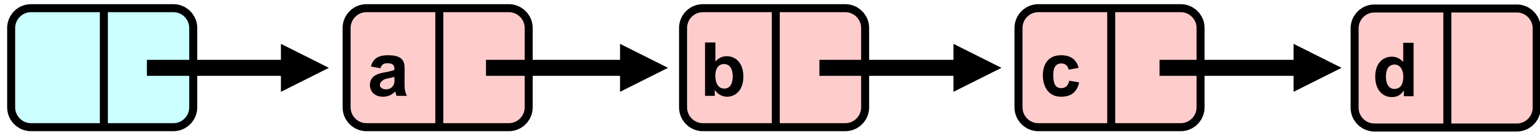
Hand-Over-Hand Again



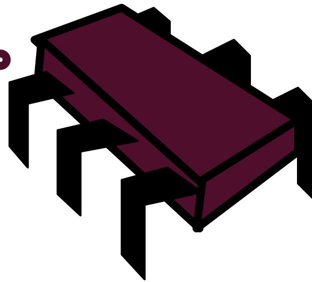
Hand-Over-Hand Again



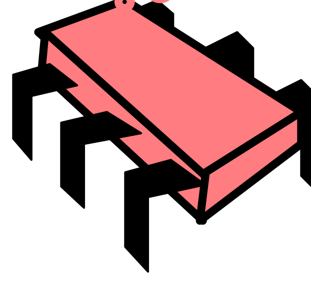
Removing a Node



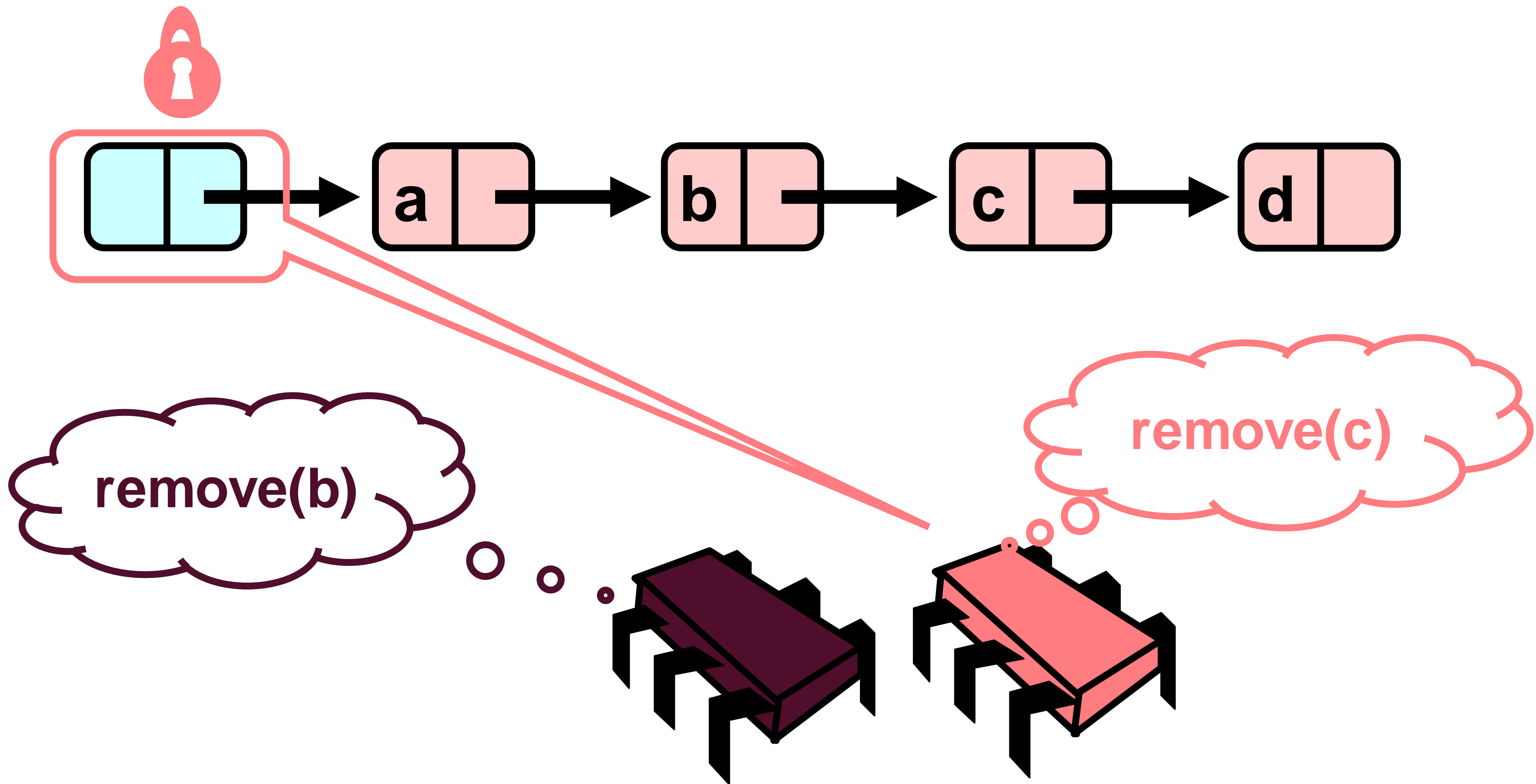
remove(b)



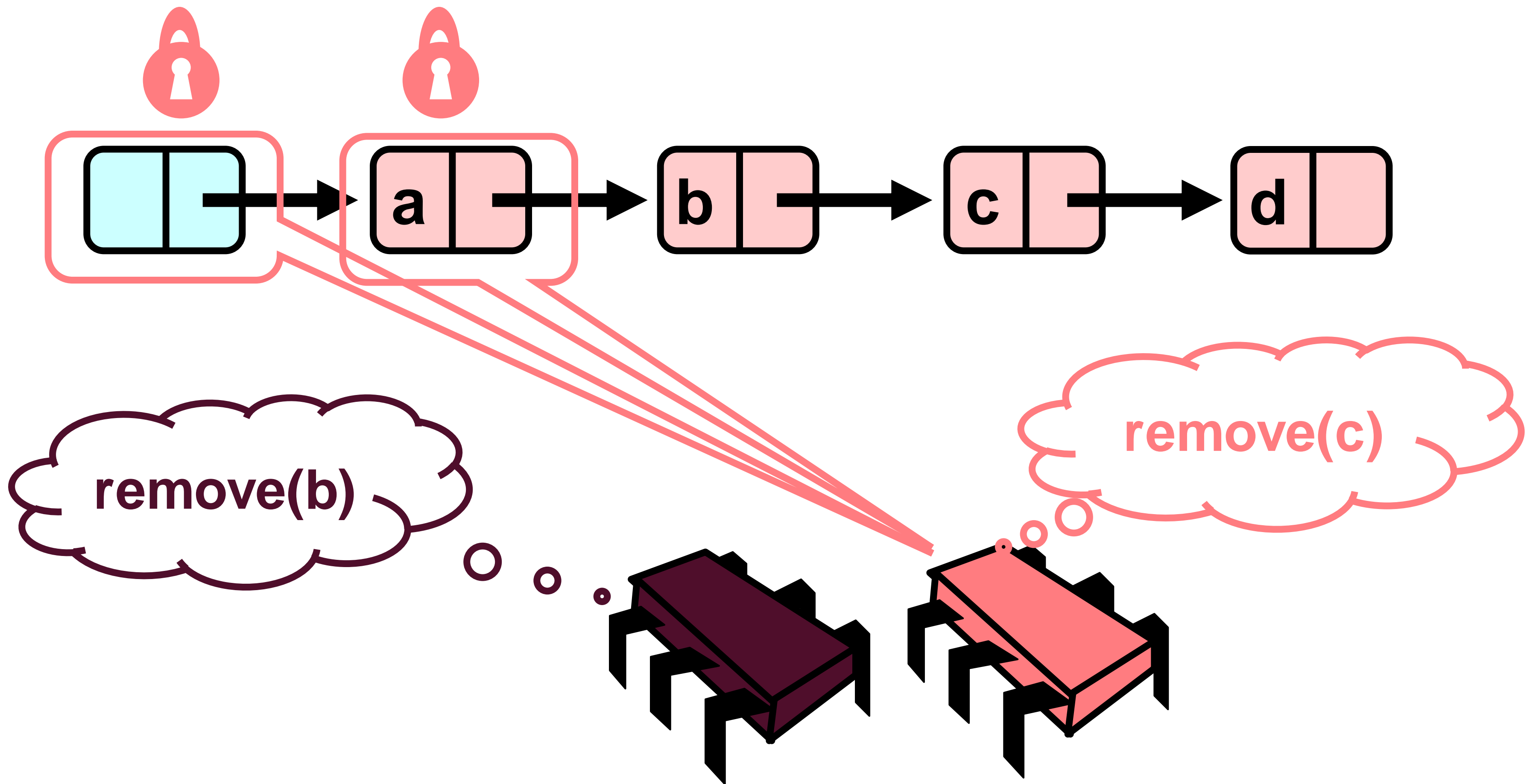
remove(c)



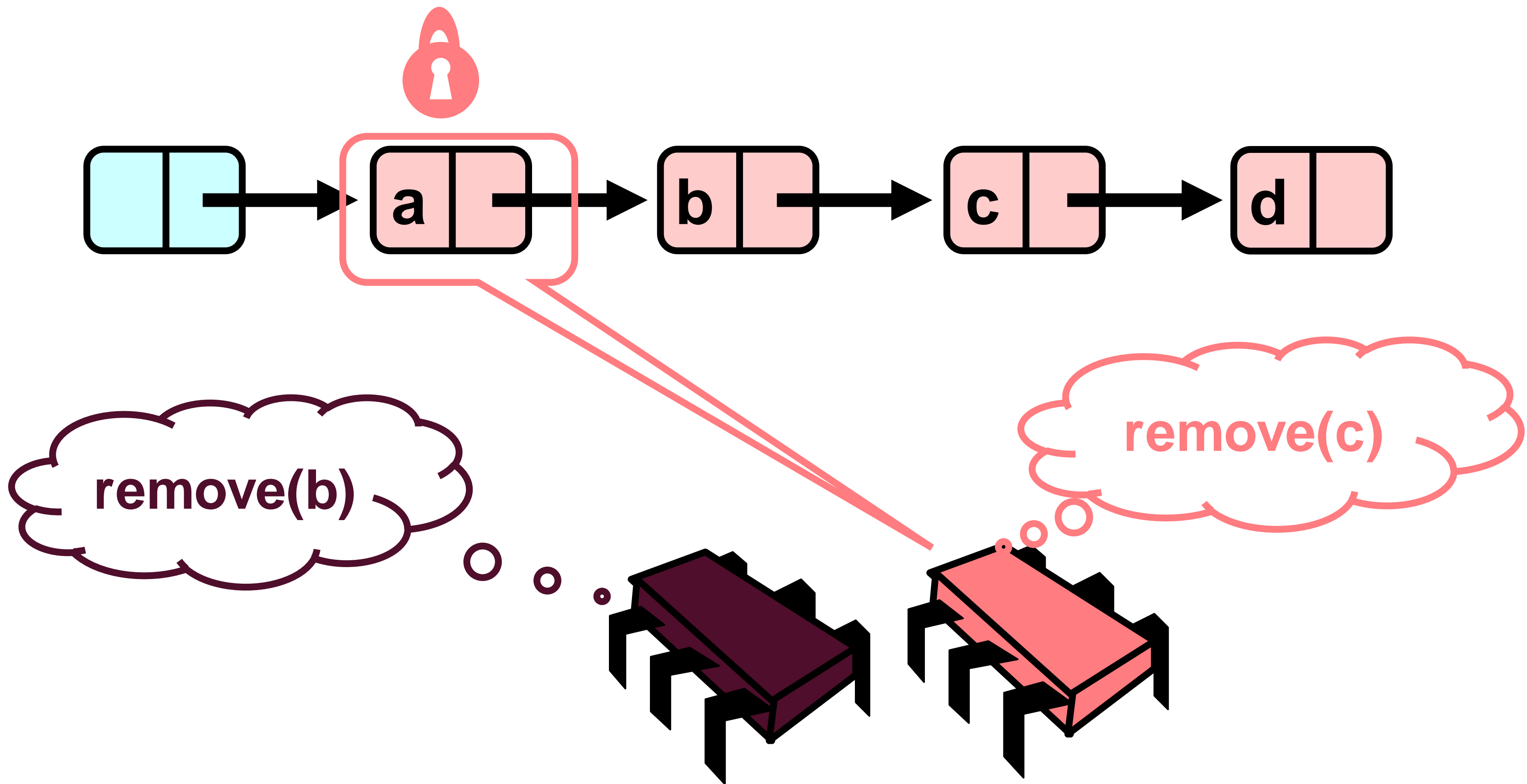
Removing a Node



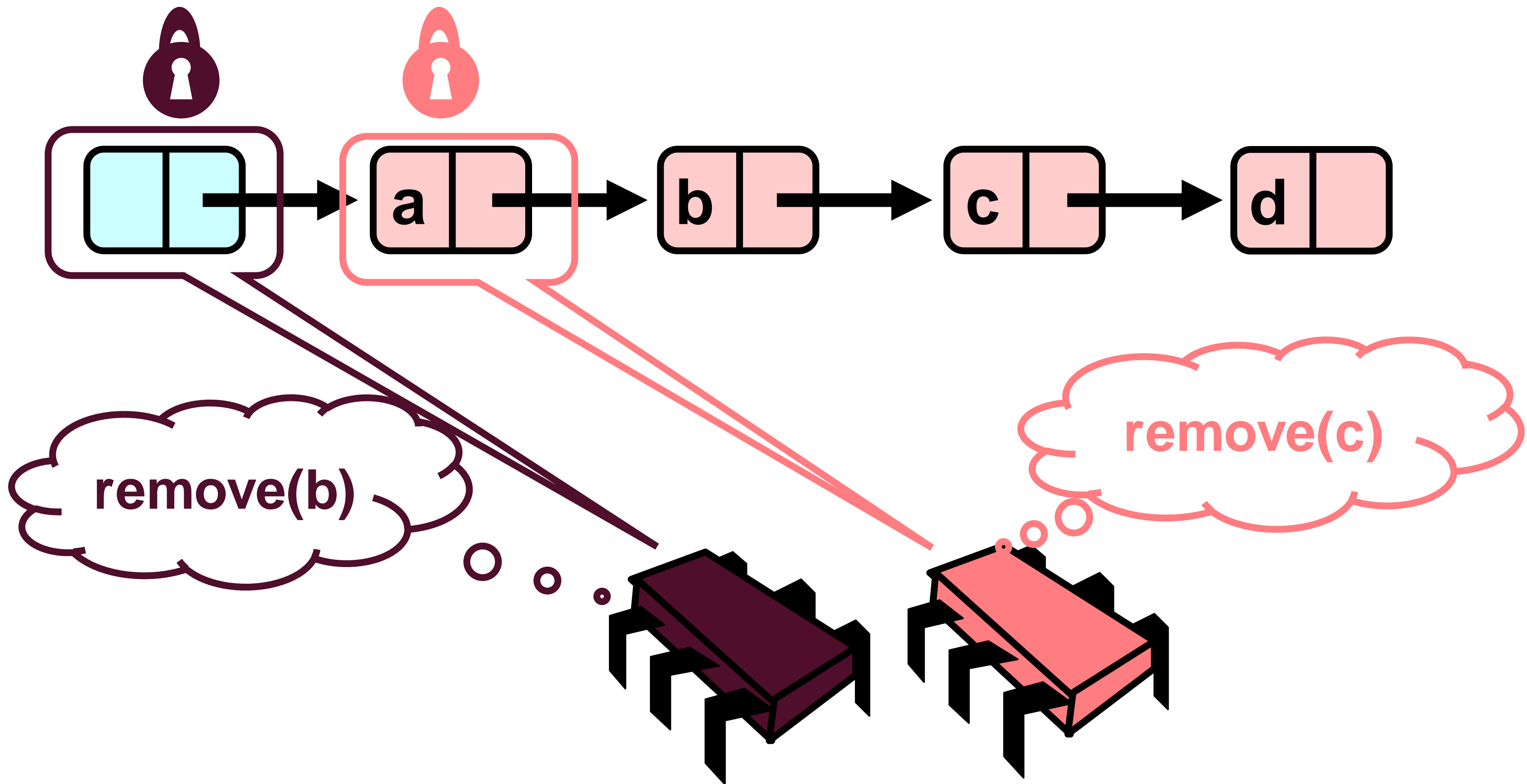
Removing a Node



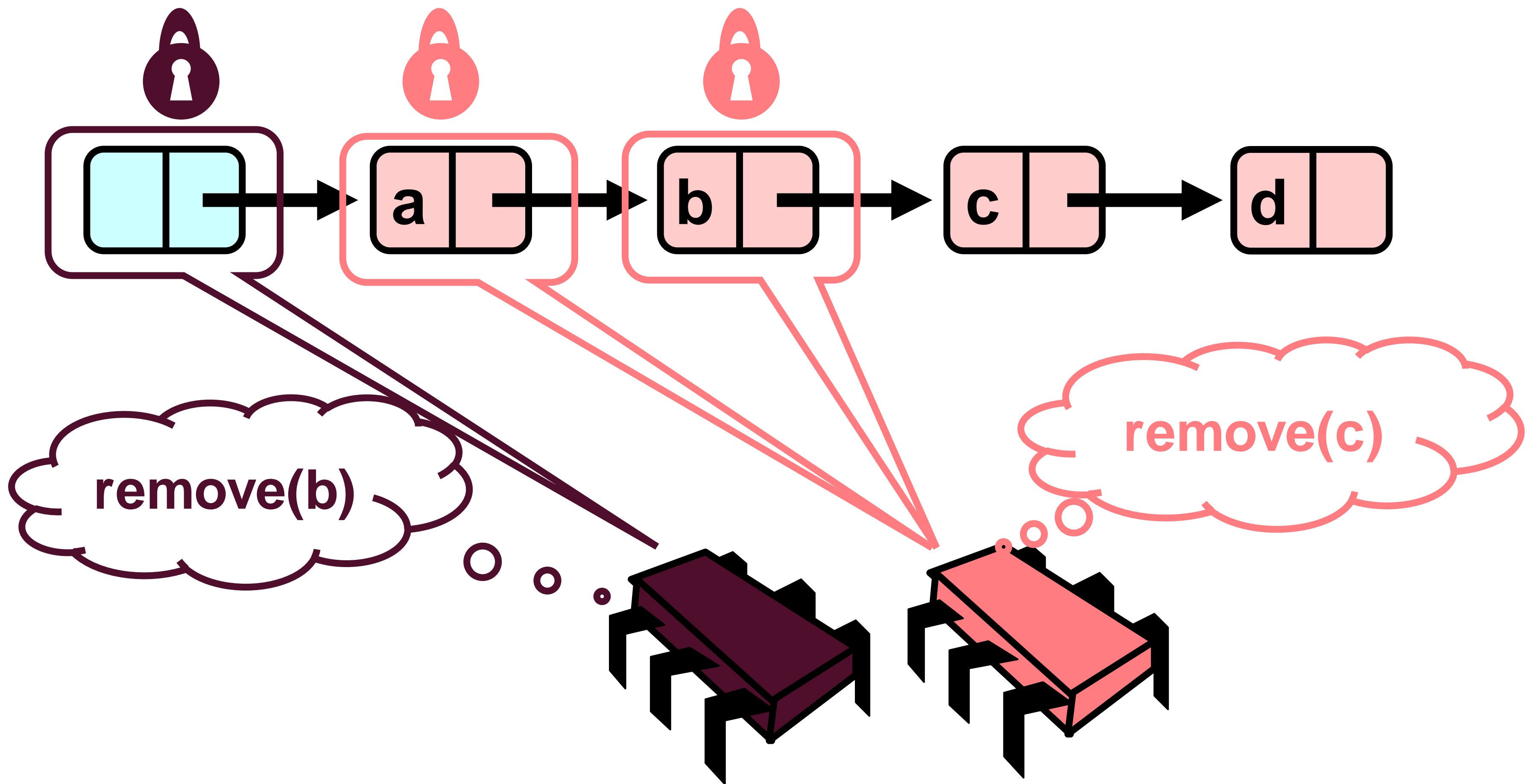
Removing a Node



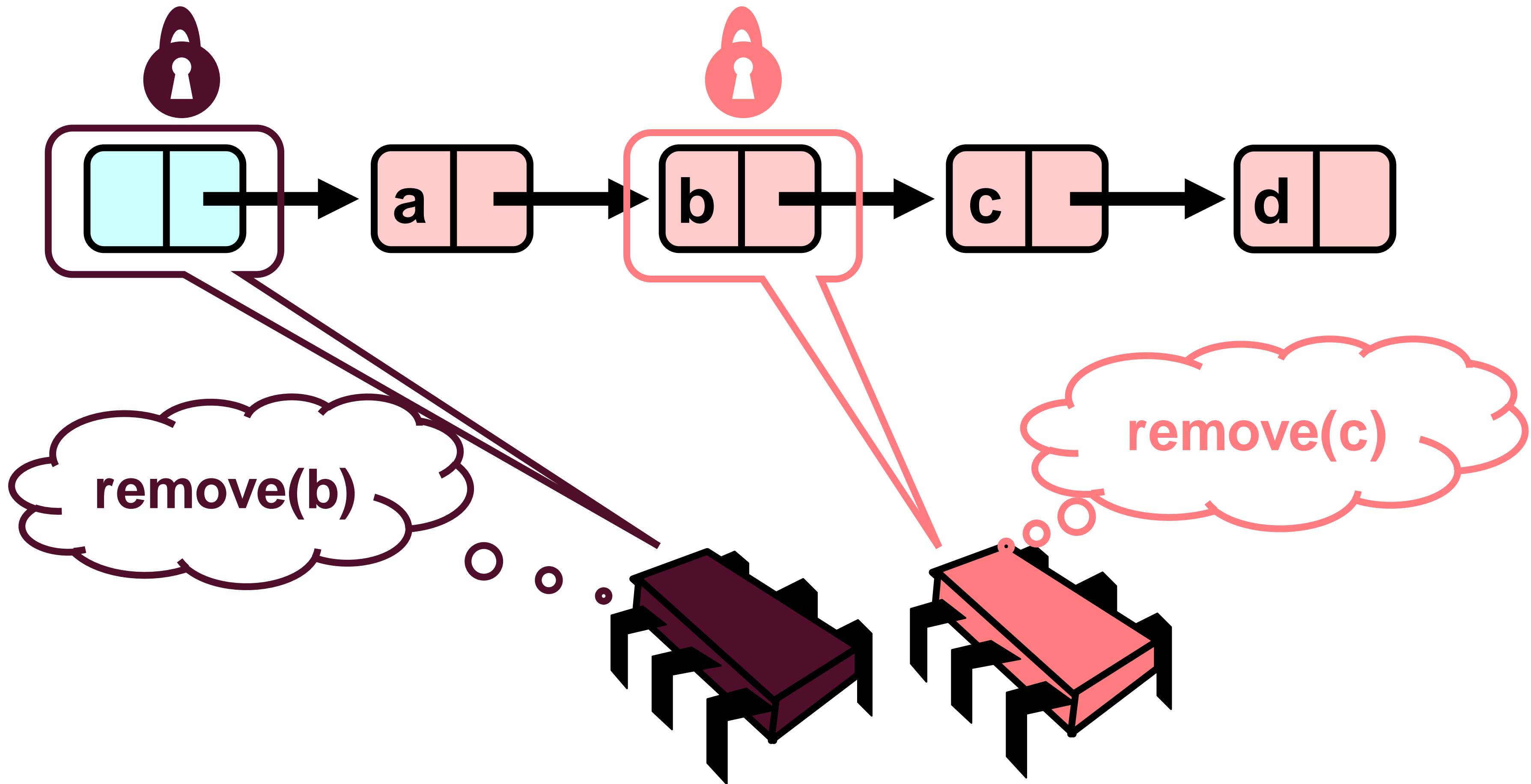
Removing a Node



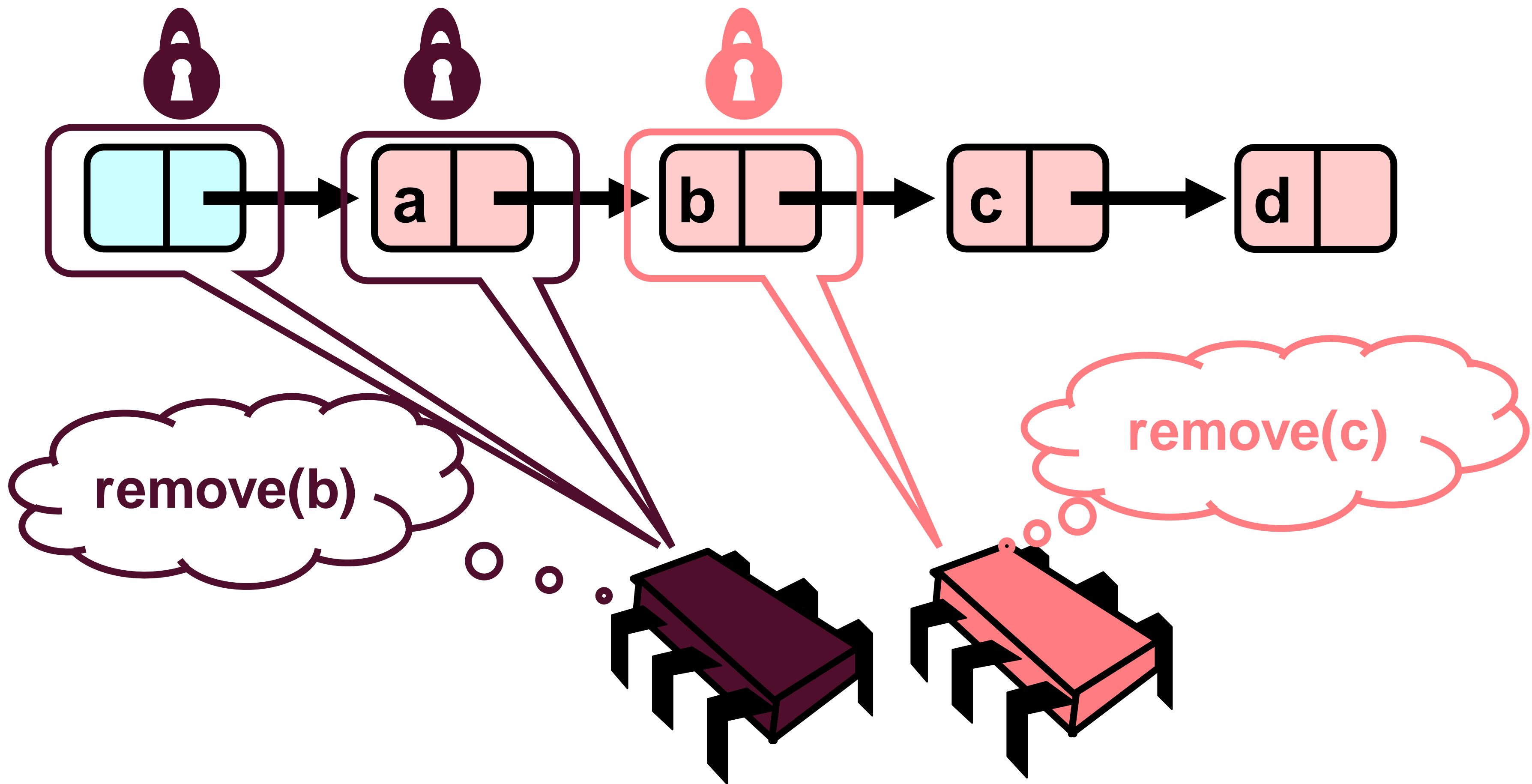
Removing a Node



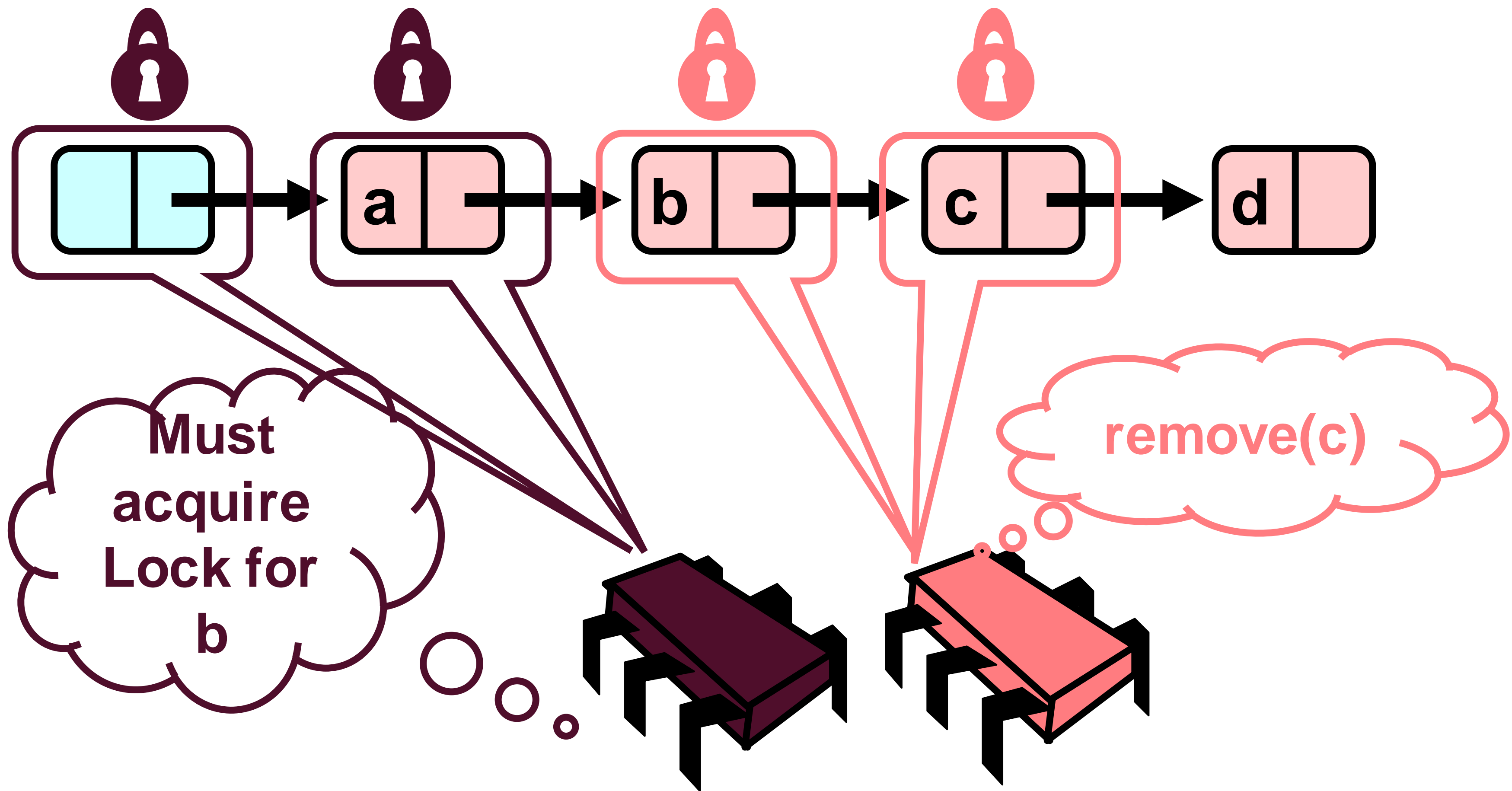
Removing a Node



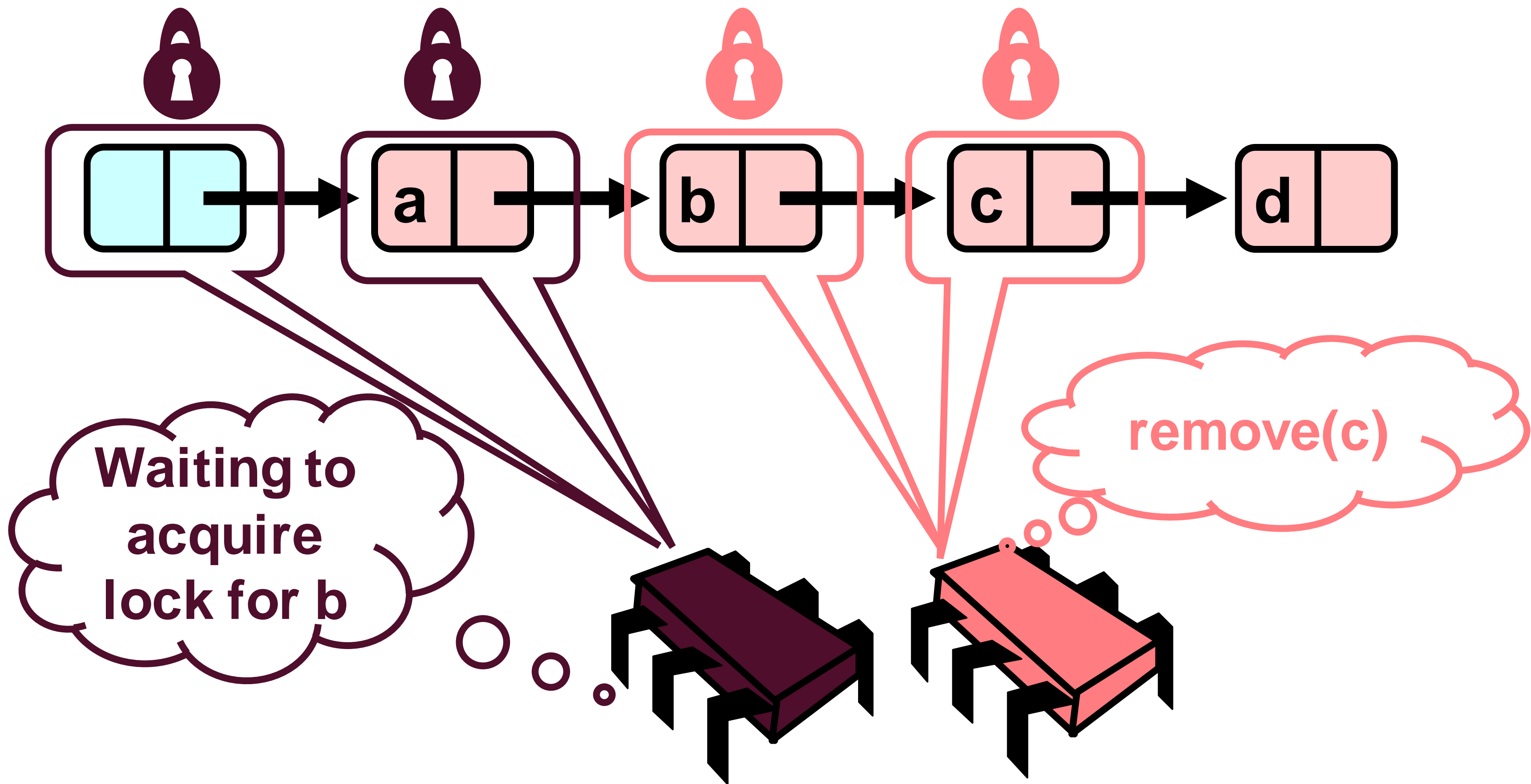
Removing a Node



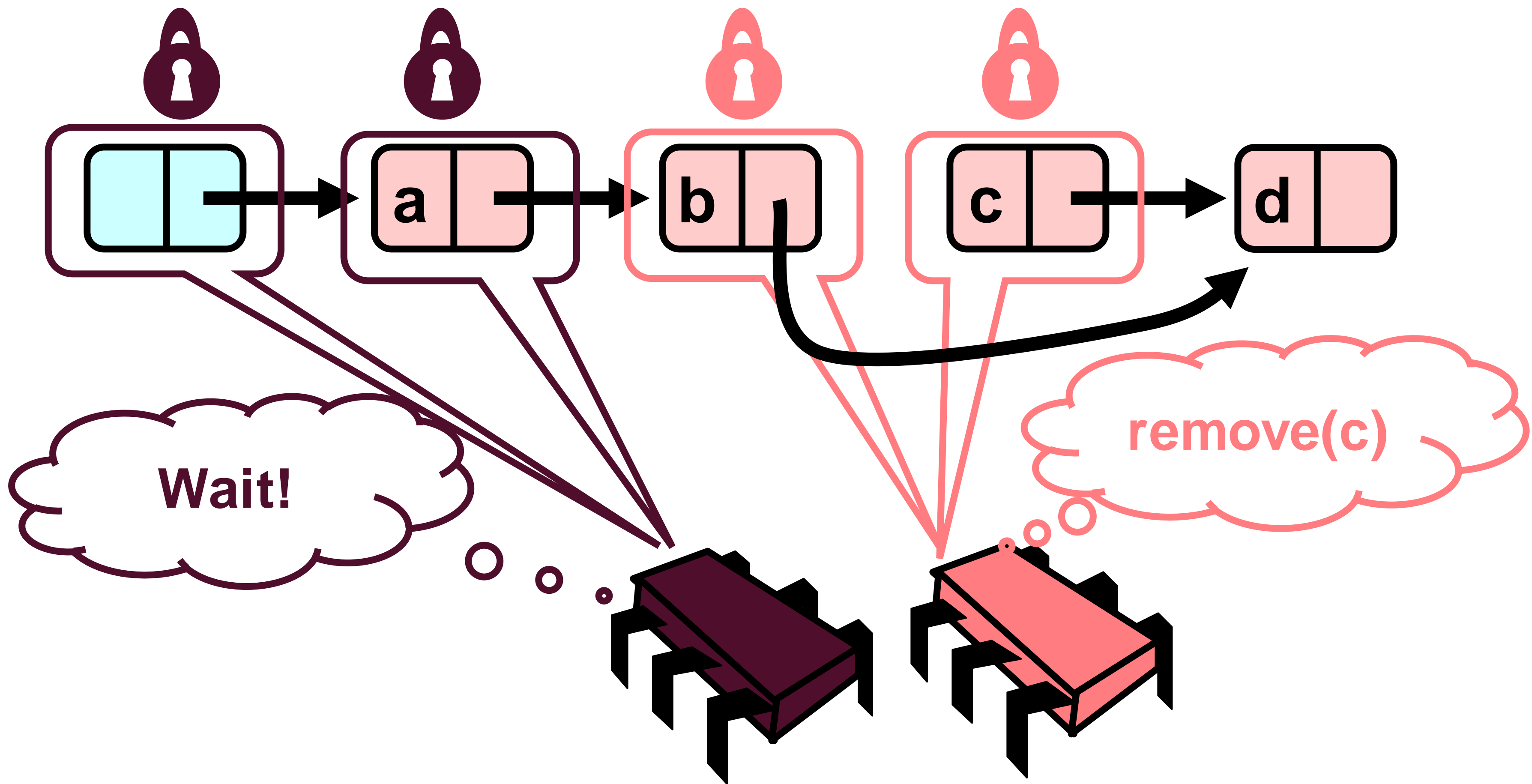
Removing a Node



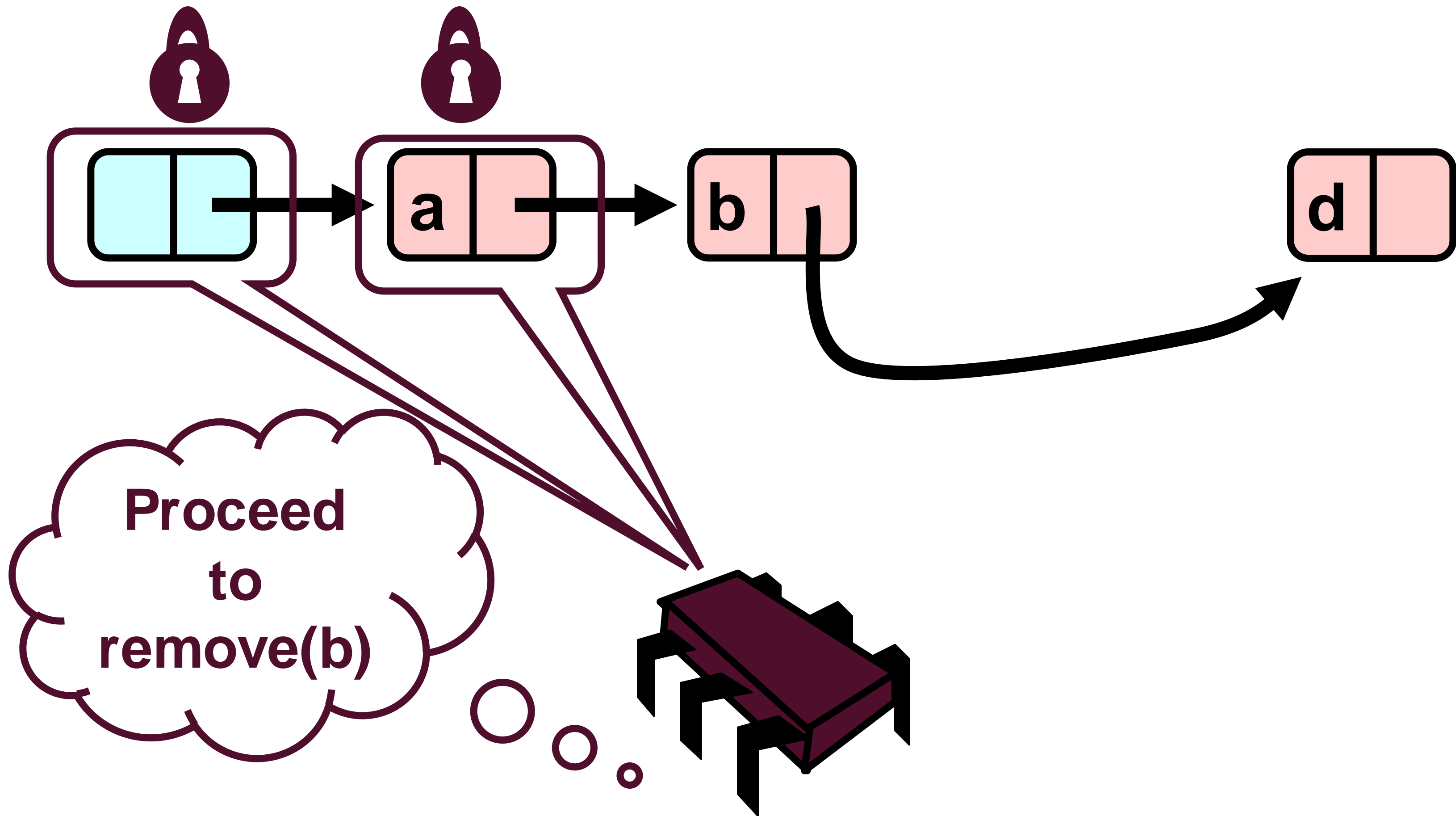
Removing a Node



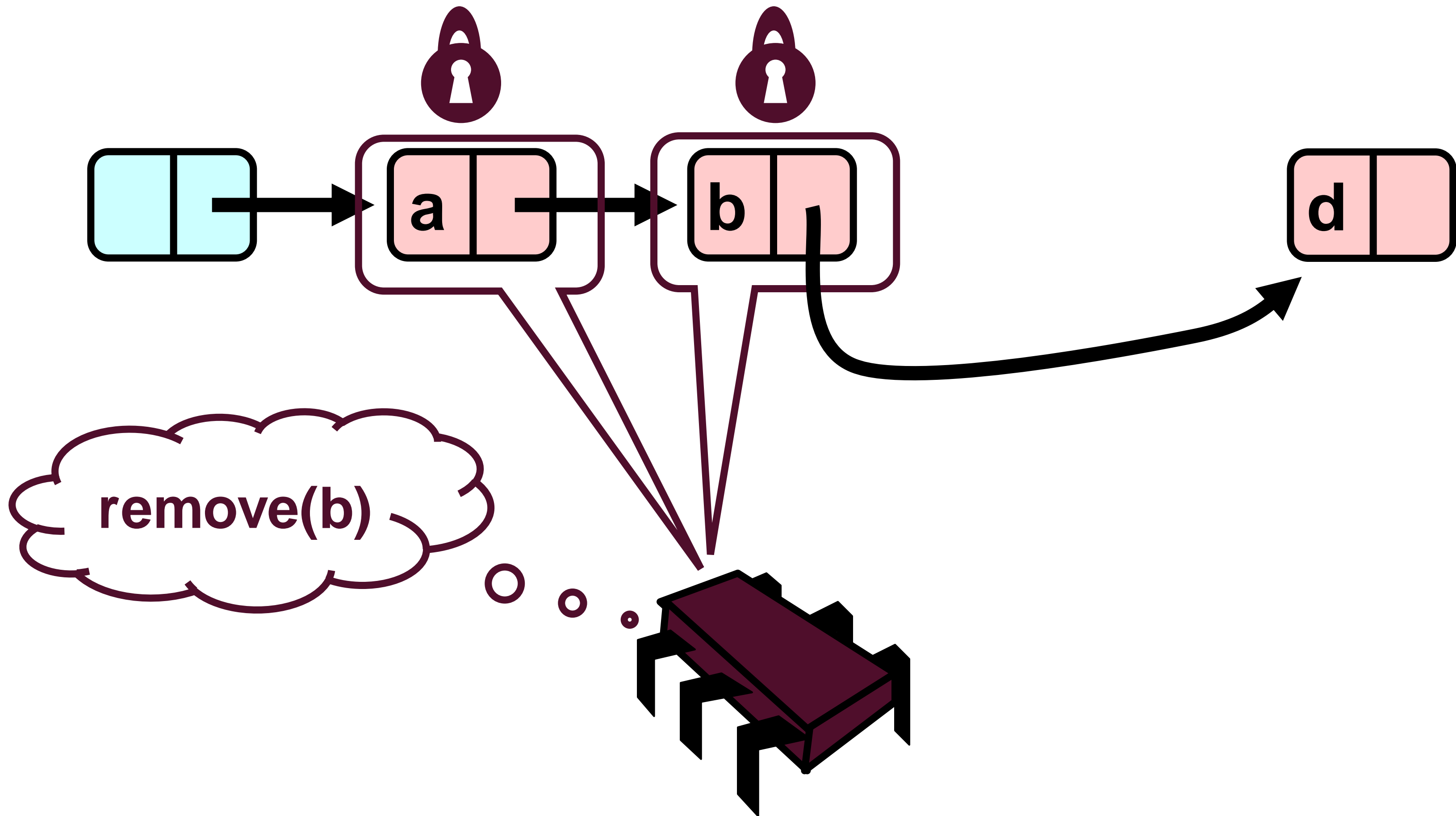
Removing a Node



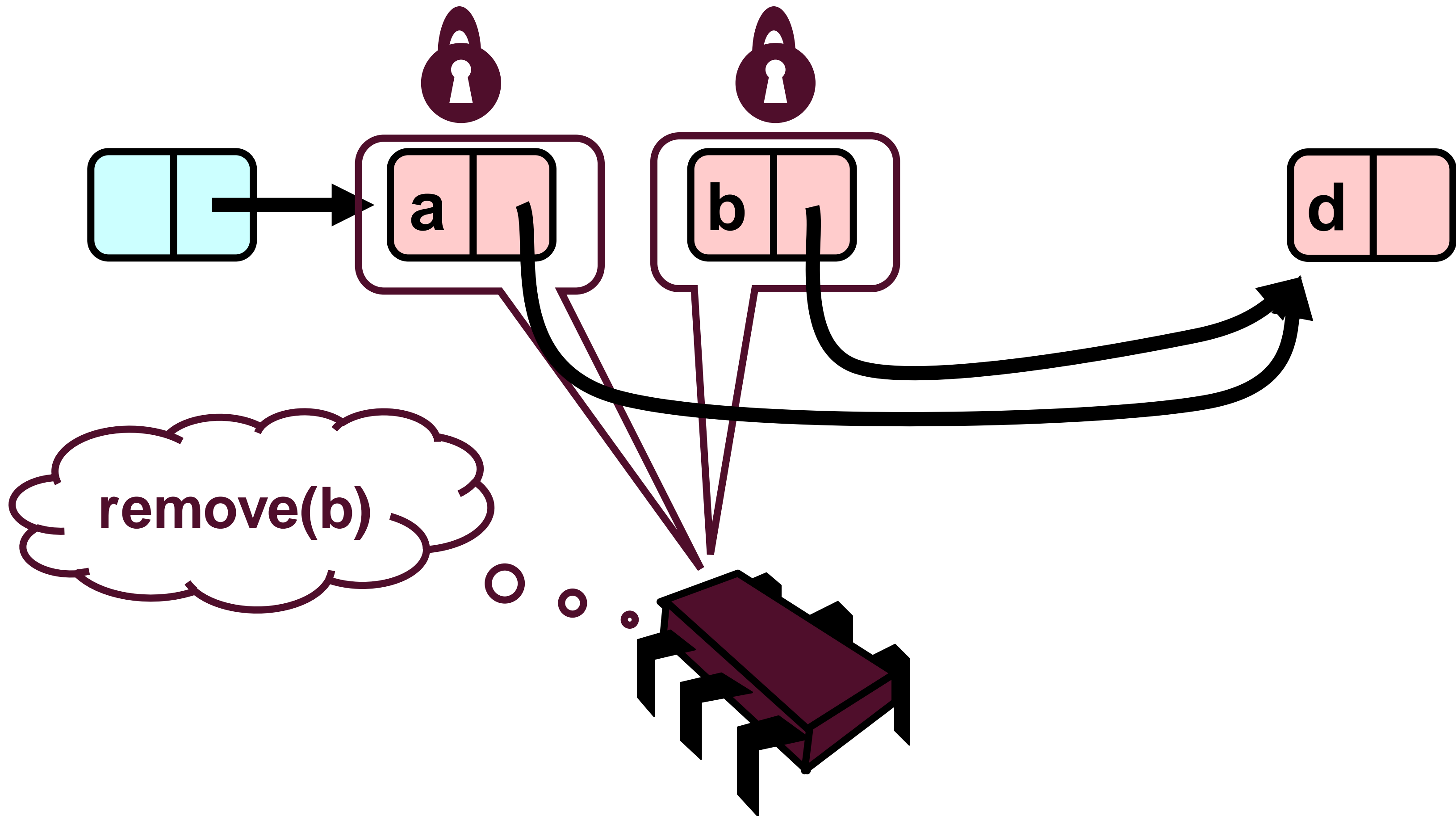
Removing a Node



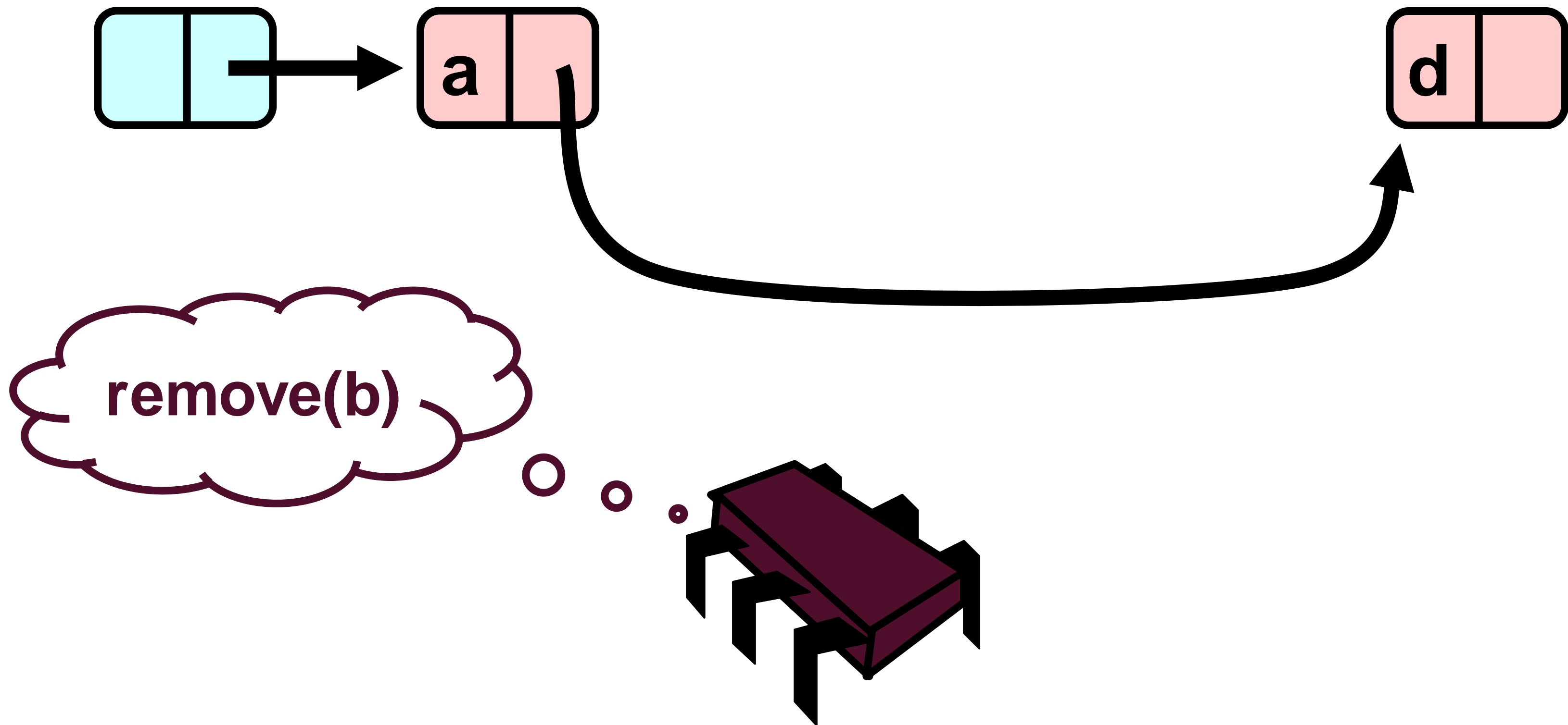
Removing a Node



Removing a Node



Removing a Node



What are the Issues?

- **We have fine-grained locking, will there be contention?**
  - Yes, the list can only be traversed sequentially, a remove of the 3rd item will block all other threads!
  - This is essentially still serialized if the list is short (since threads can only pipeline on list elements)
- **Other problems, ignoring contention?**
  - Must acquire O(|S|) locks

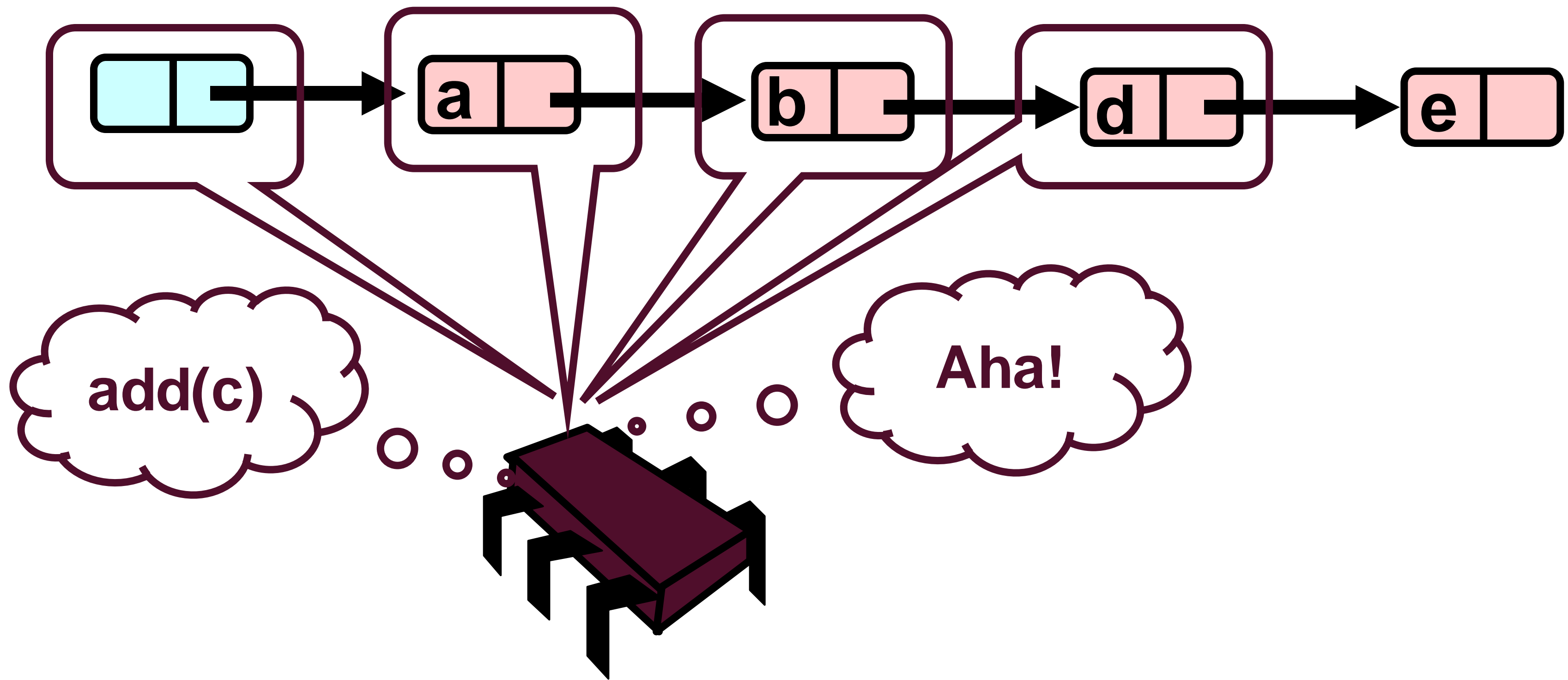
Trick 2: Reader/Writer Locking

- **Same hand-over-hand locking**
  - Traversal uses reader locks
  - Once add finds position or remove finds target node, upgrade **both** locks to writer locks
  - Need to guarantee deadlock and starvation freedom!
- **Allows truly concurrent traversals**
  - Still blocks behind writing threads
  - Still O(|S|) lock/unlock operations

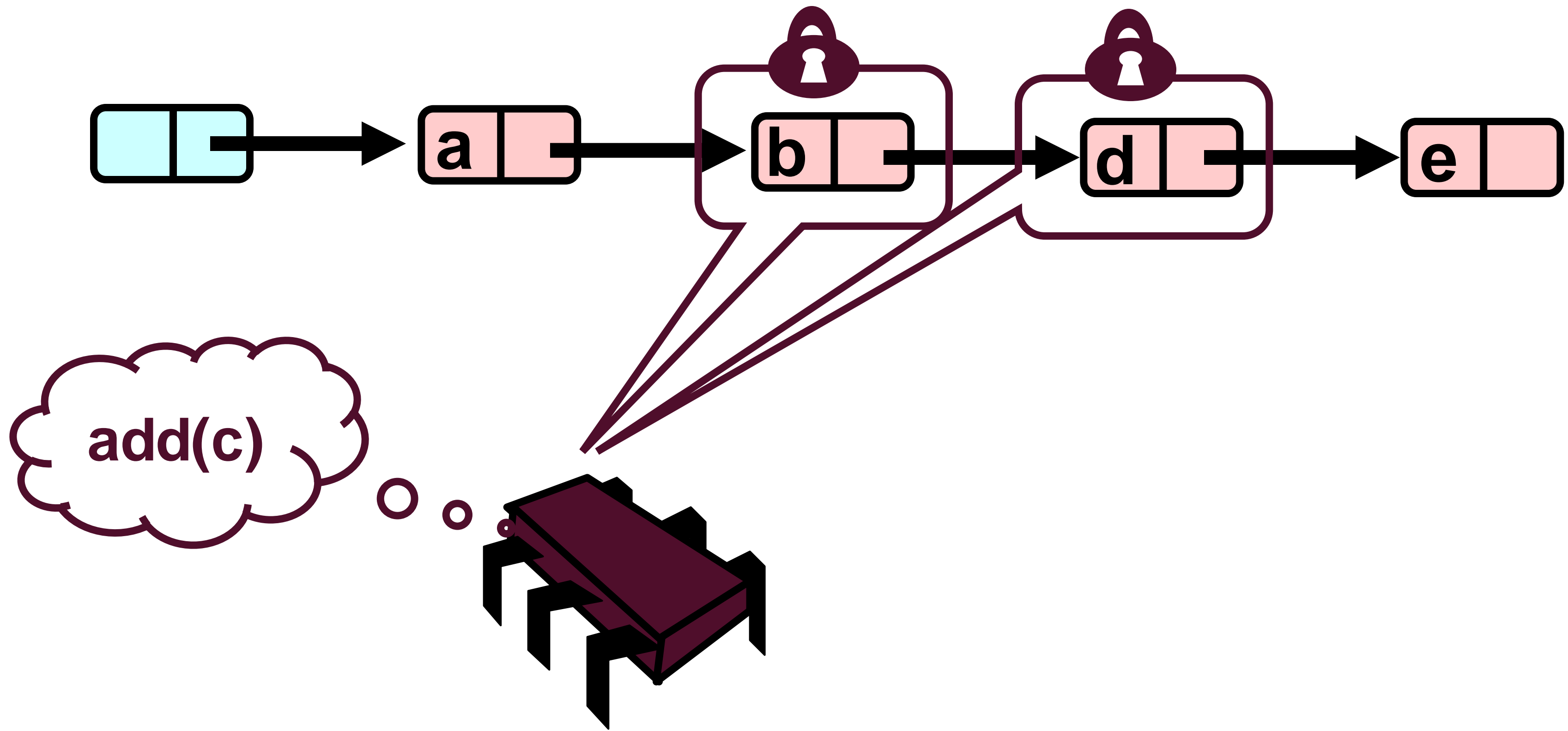
Trick 3: Optimistic synchronization

- **Similar to reader/writer locking but traverse list without locks**
  - Dangerous! Requires additional checks.
- **Harder to proof correct**

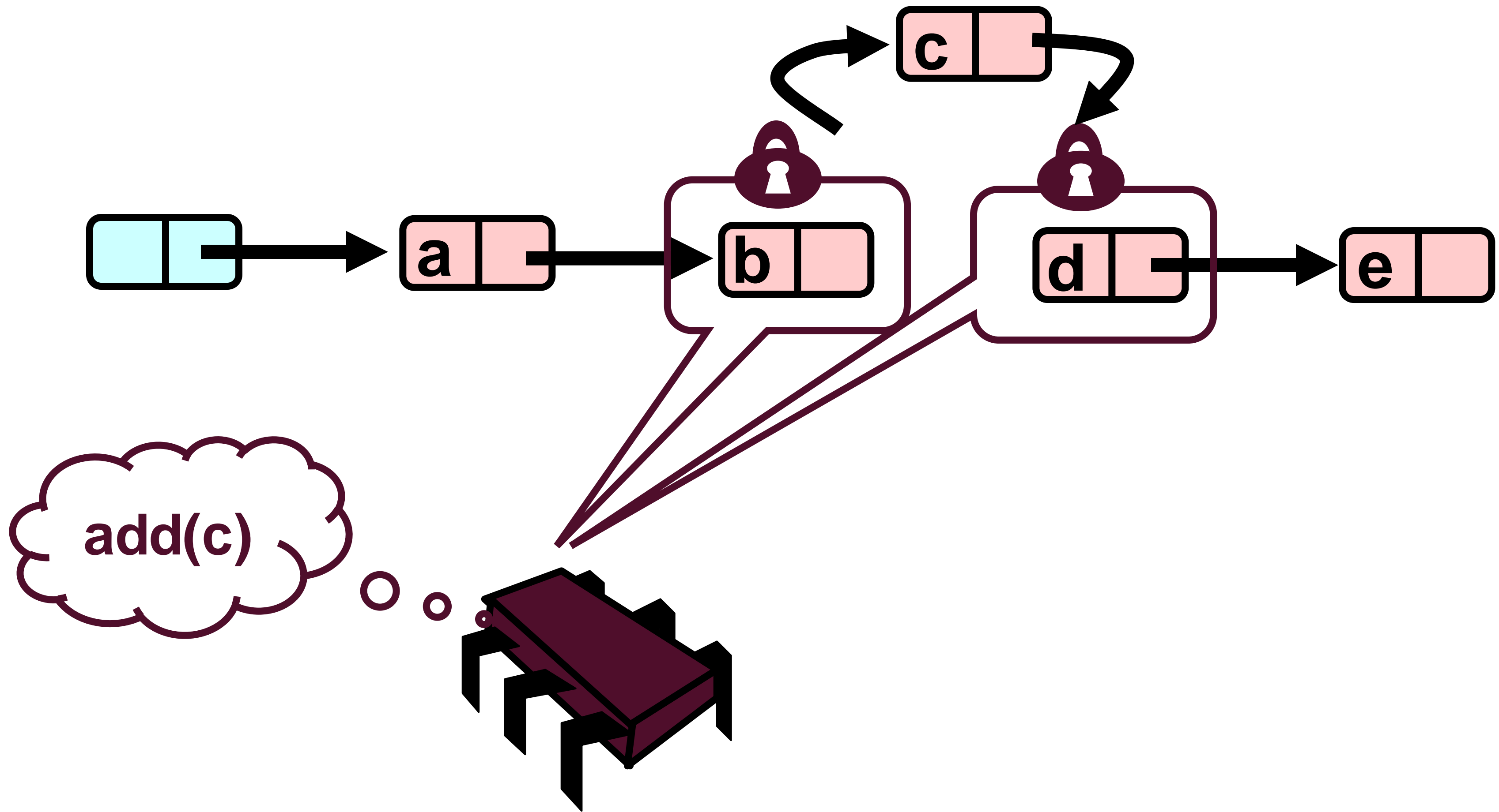
Optimistic: Traverse without Locking



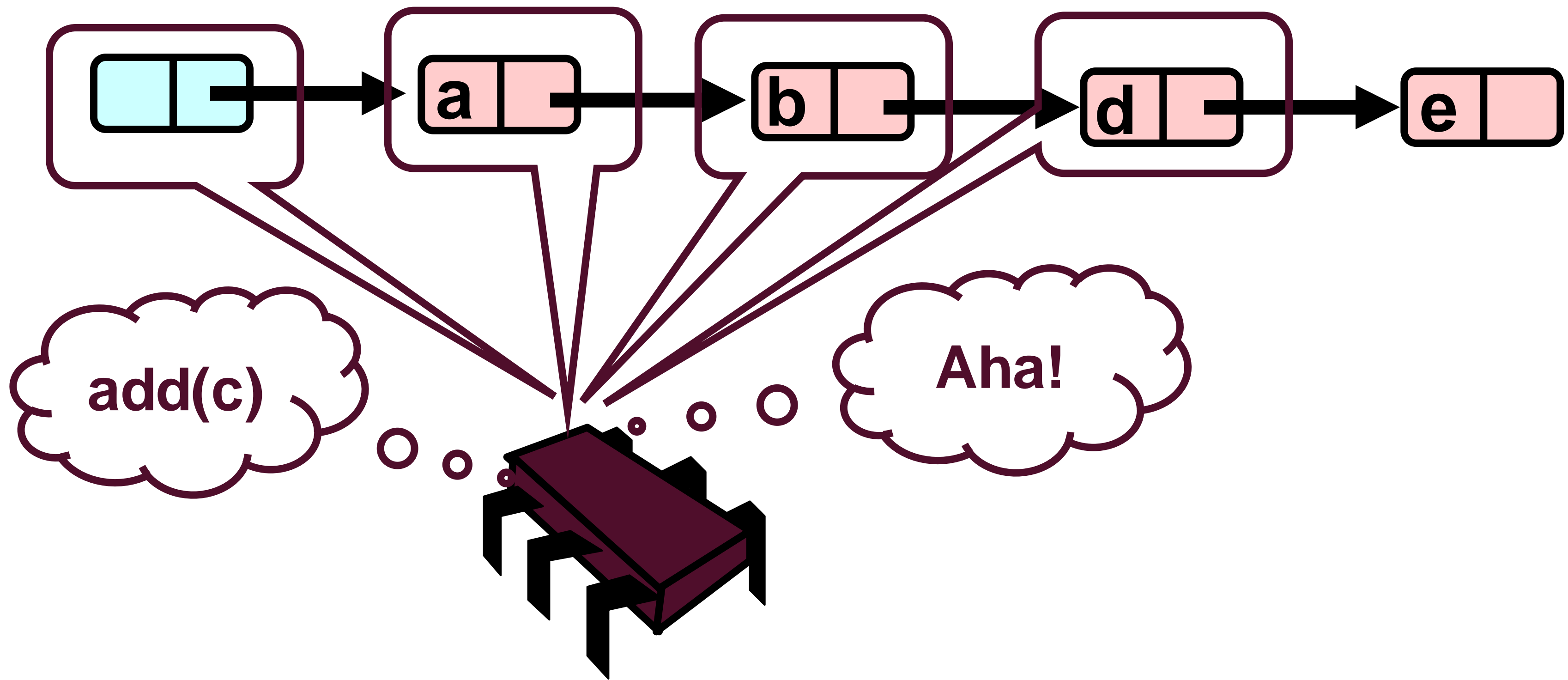
Optimistic: Lock and Load



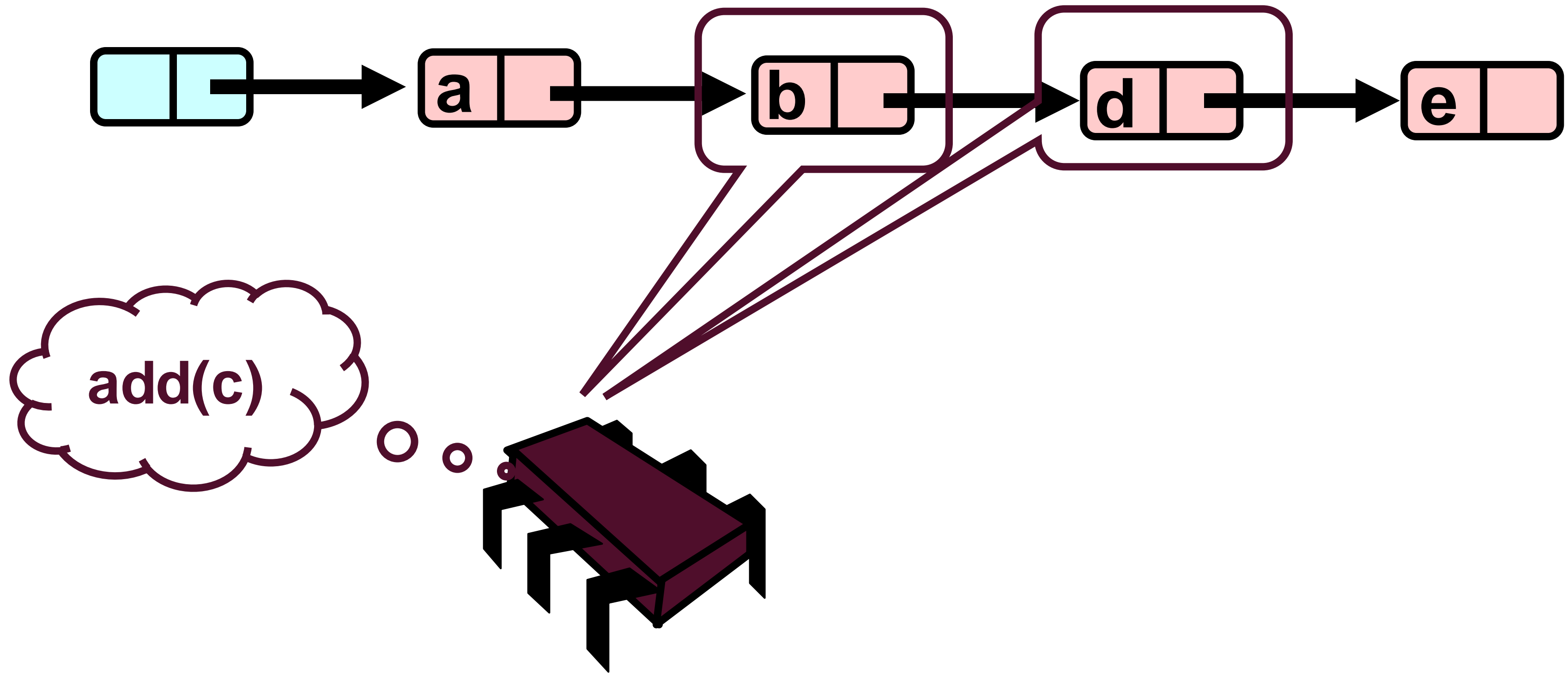
Optimistic: Lock and Load



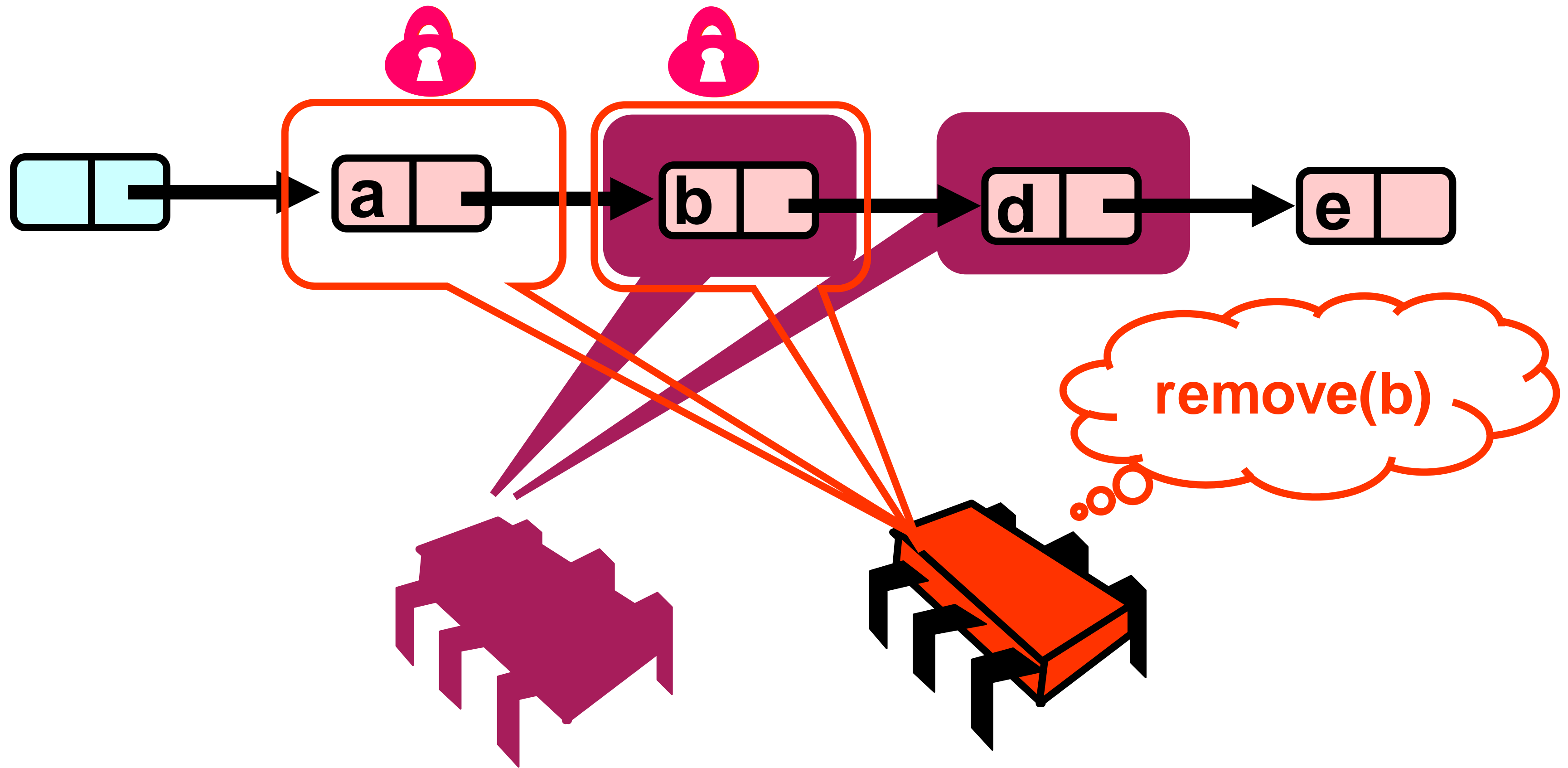
What could go wrong?



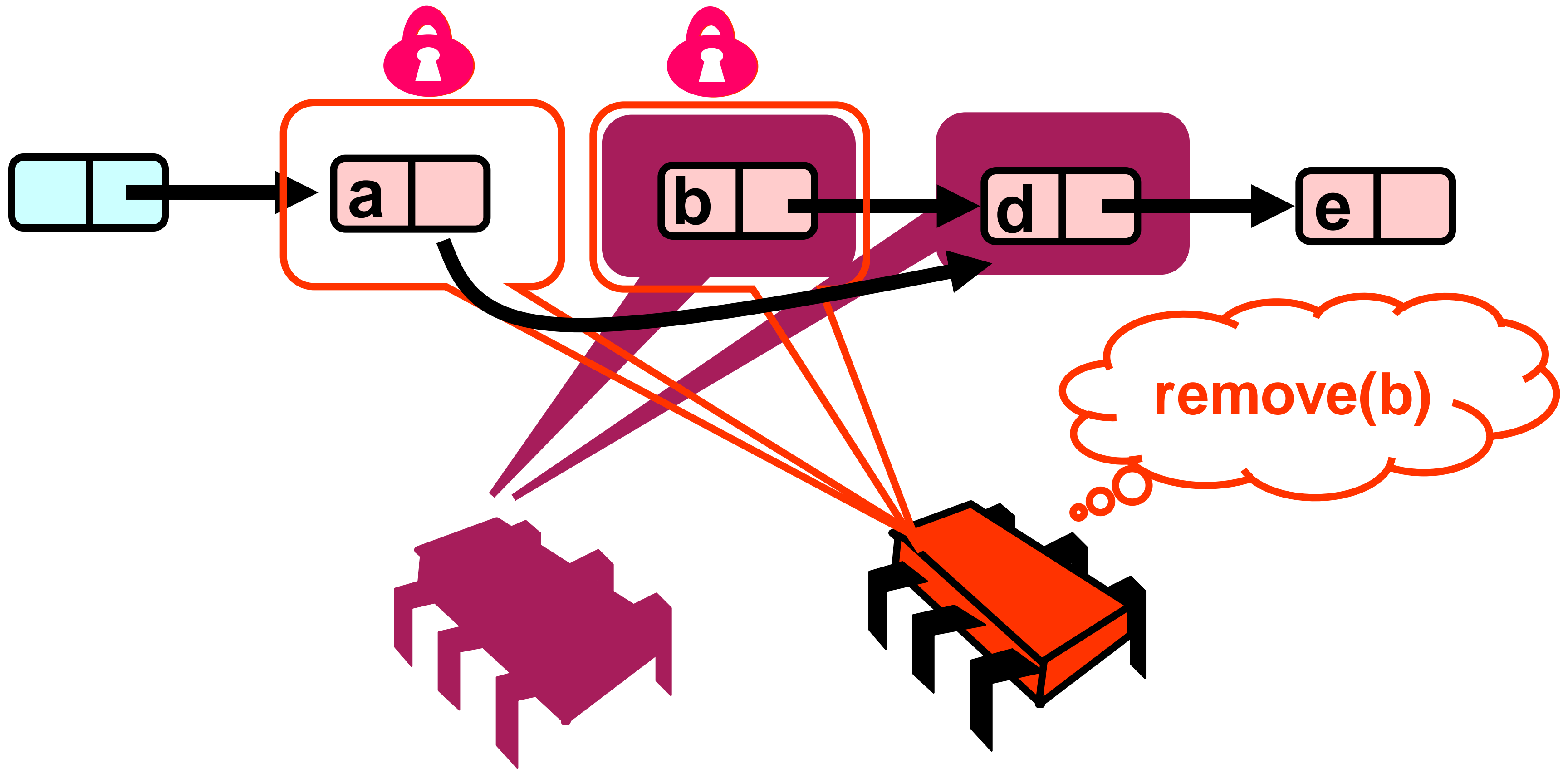
What could go wrong?



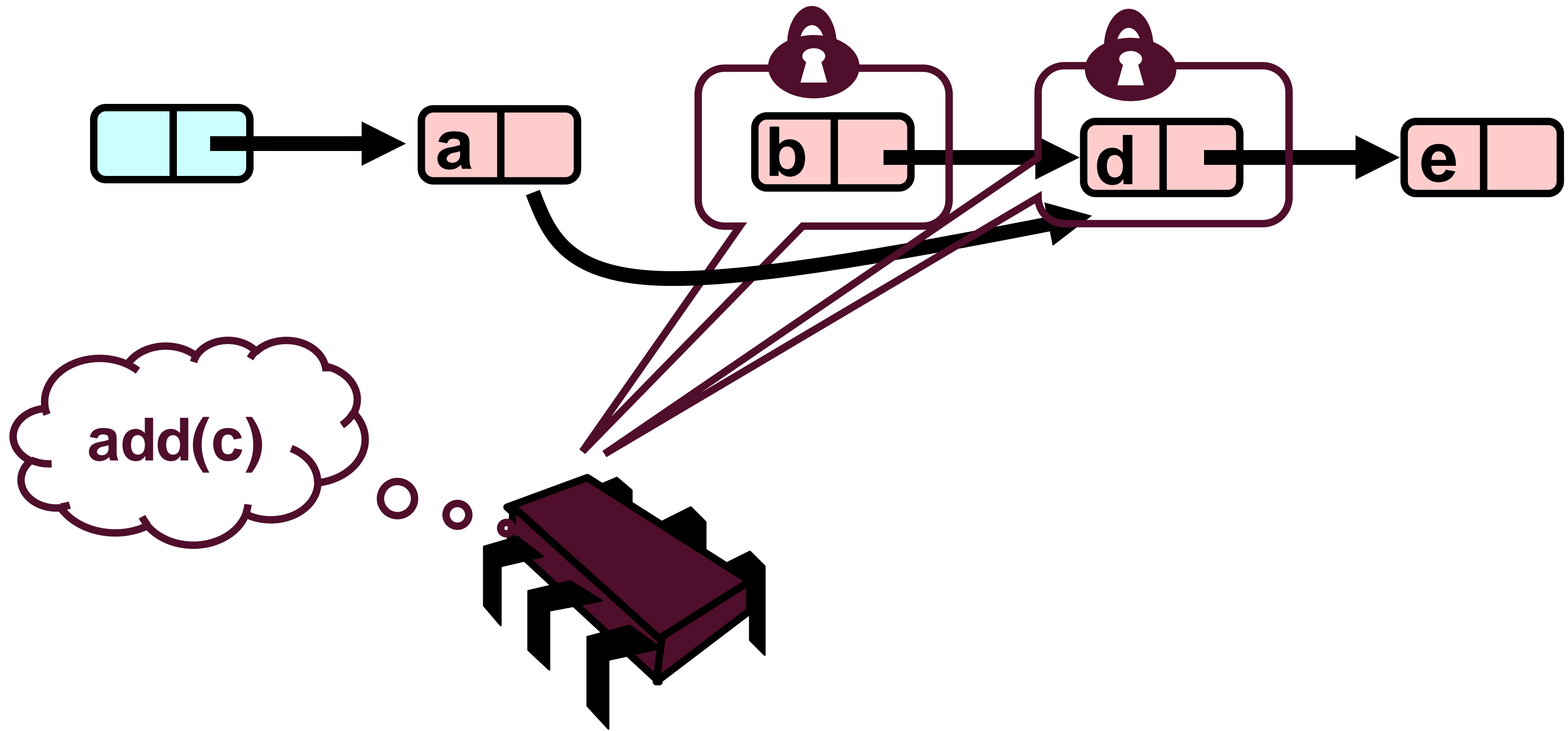
What could go wrong?



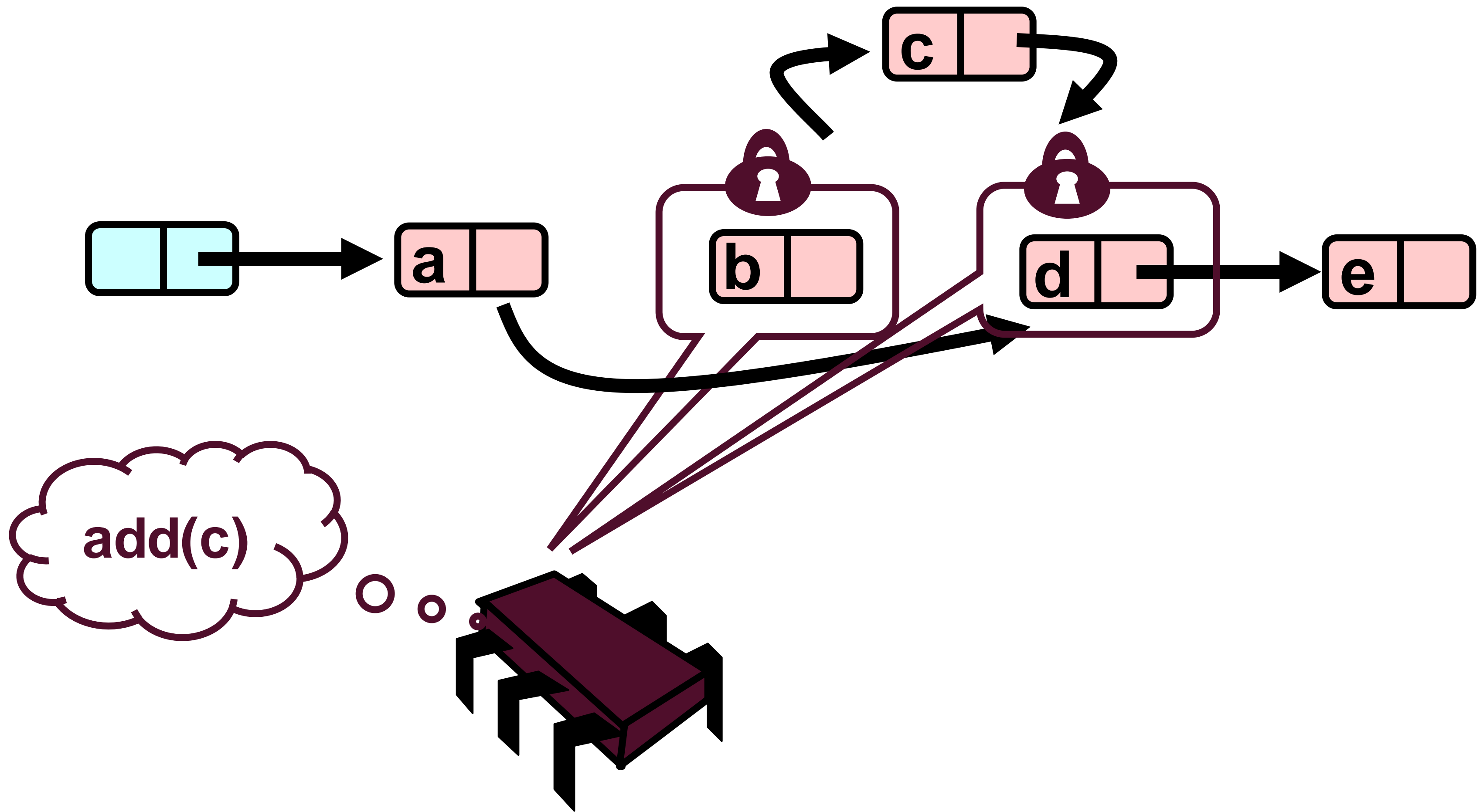
What could go wrong?



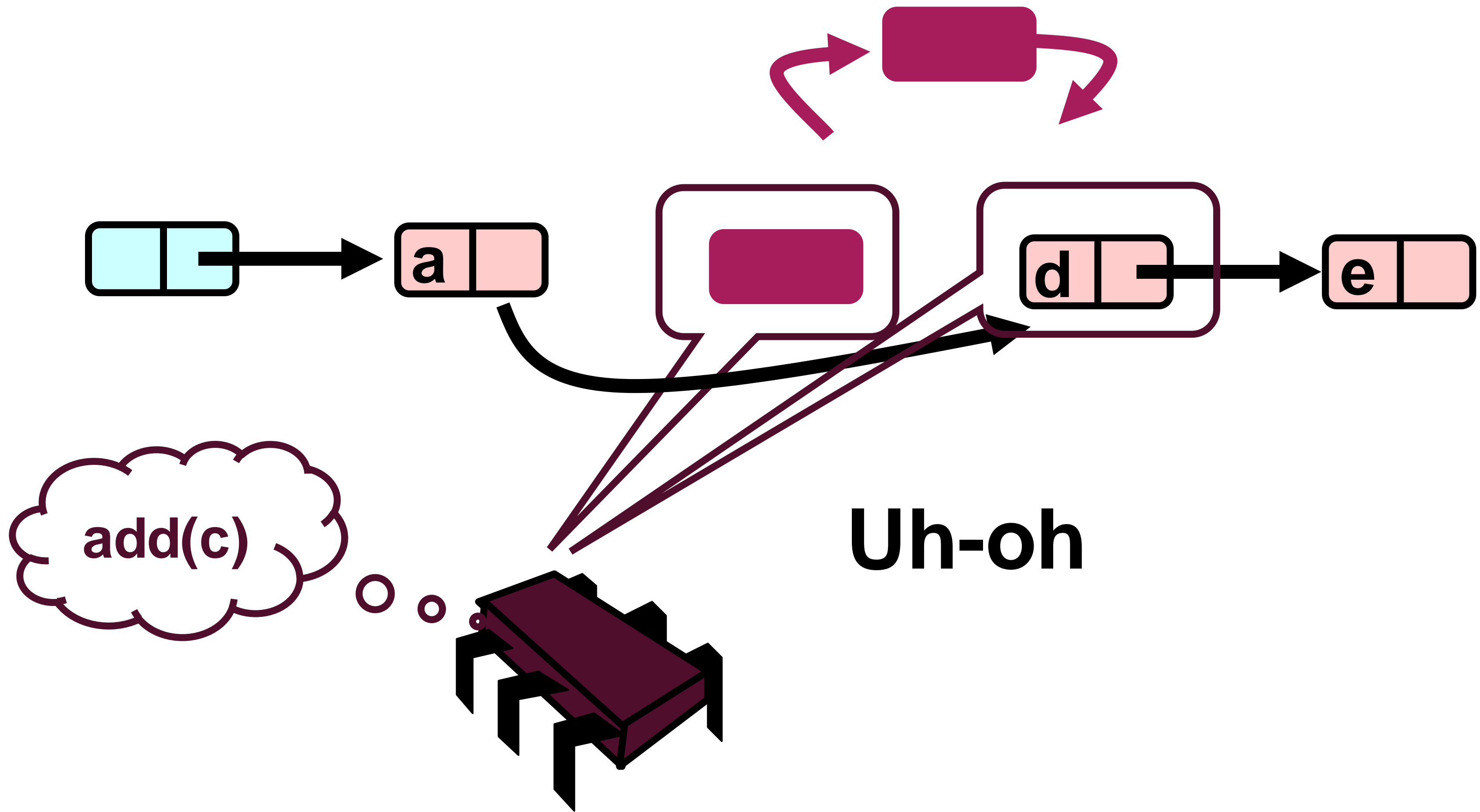
What could go wrong?



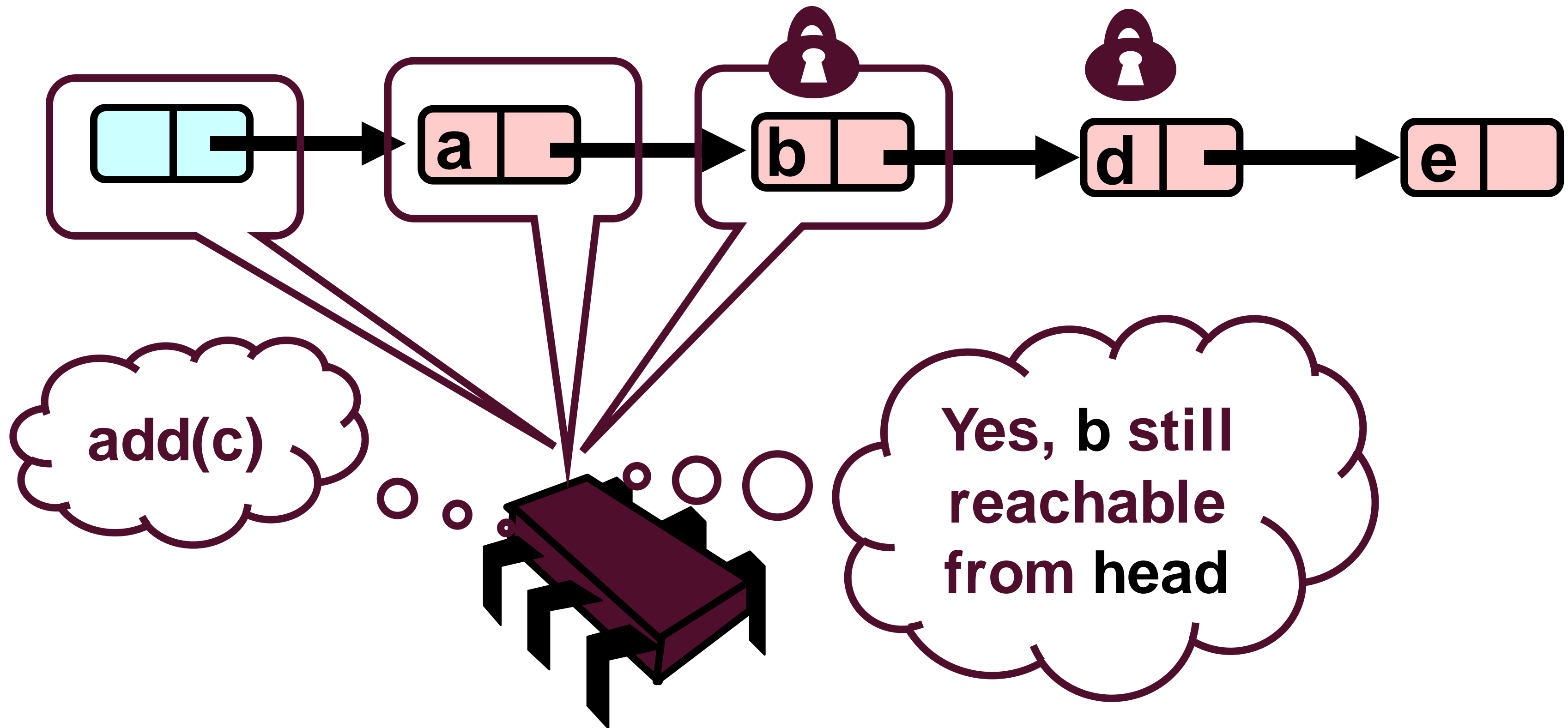
What could go wrong?



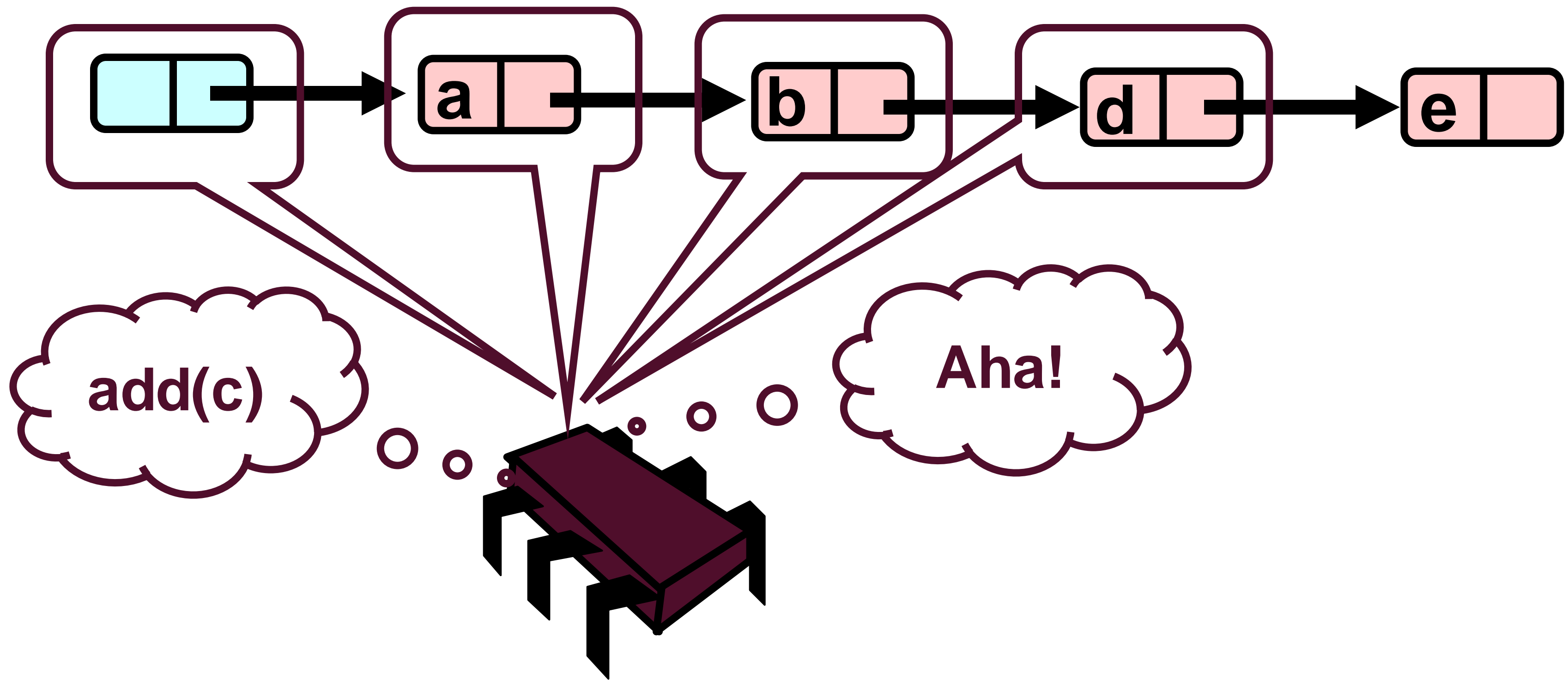
What could go wrong?



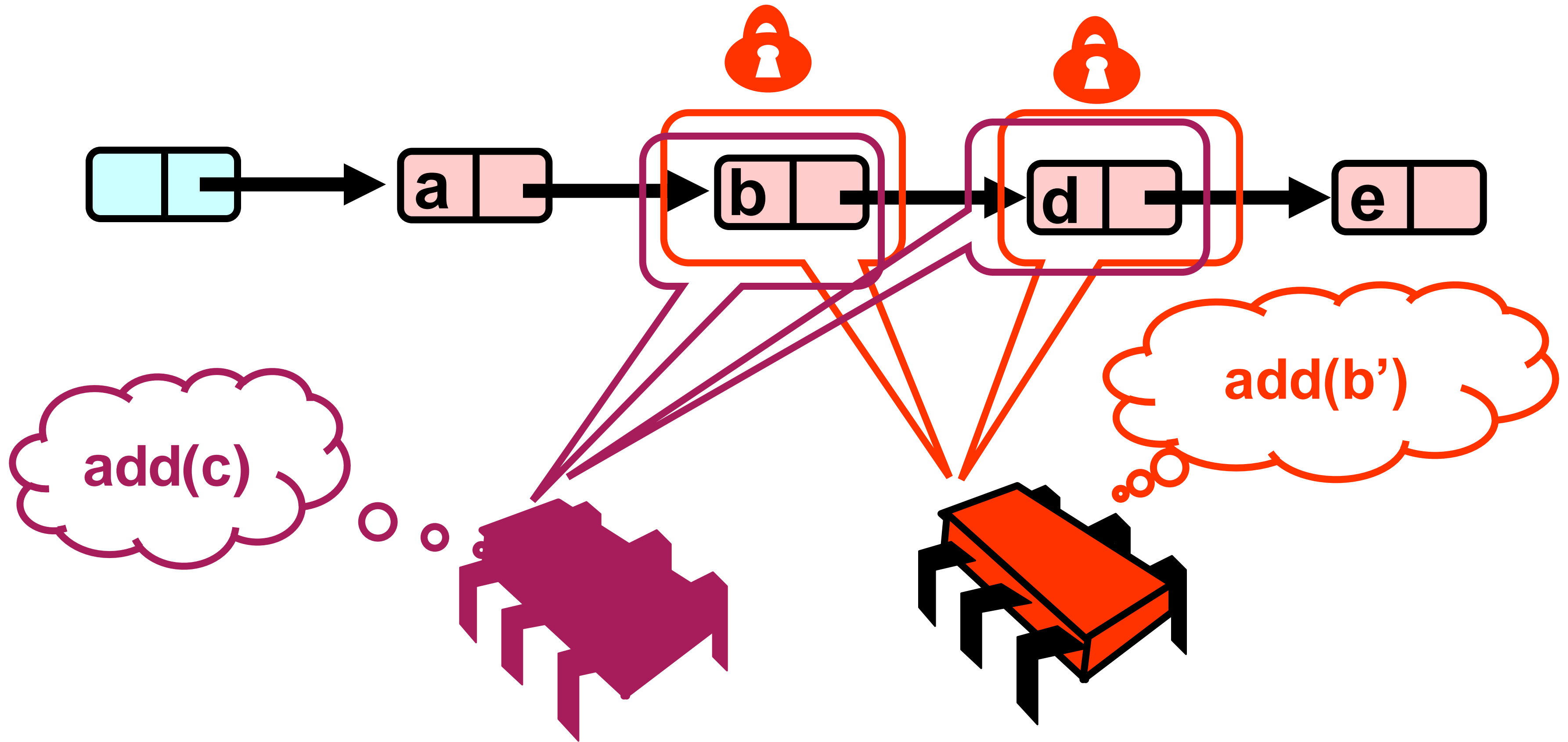
Validate – Part 1



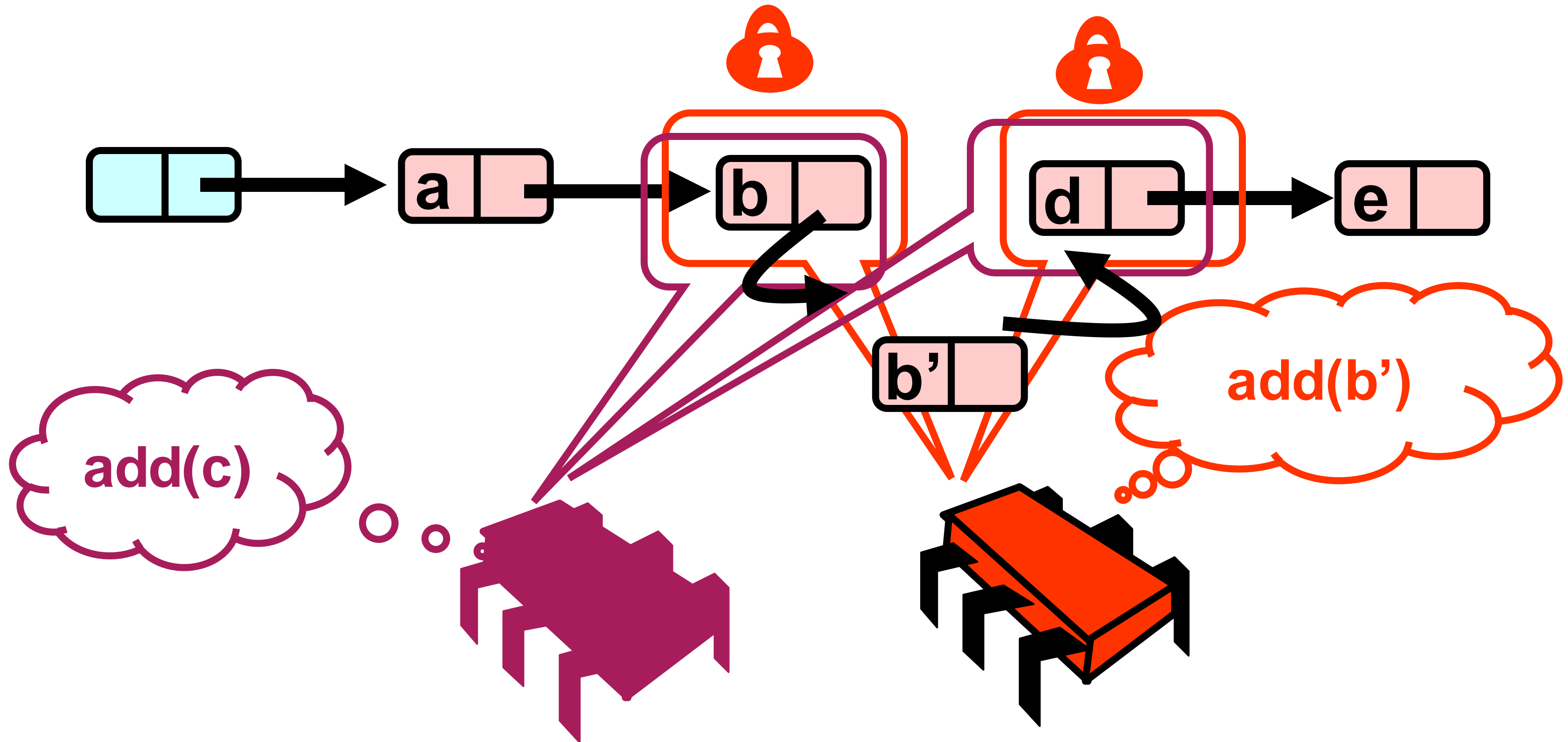
What Else Could Go Wrong?



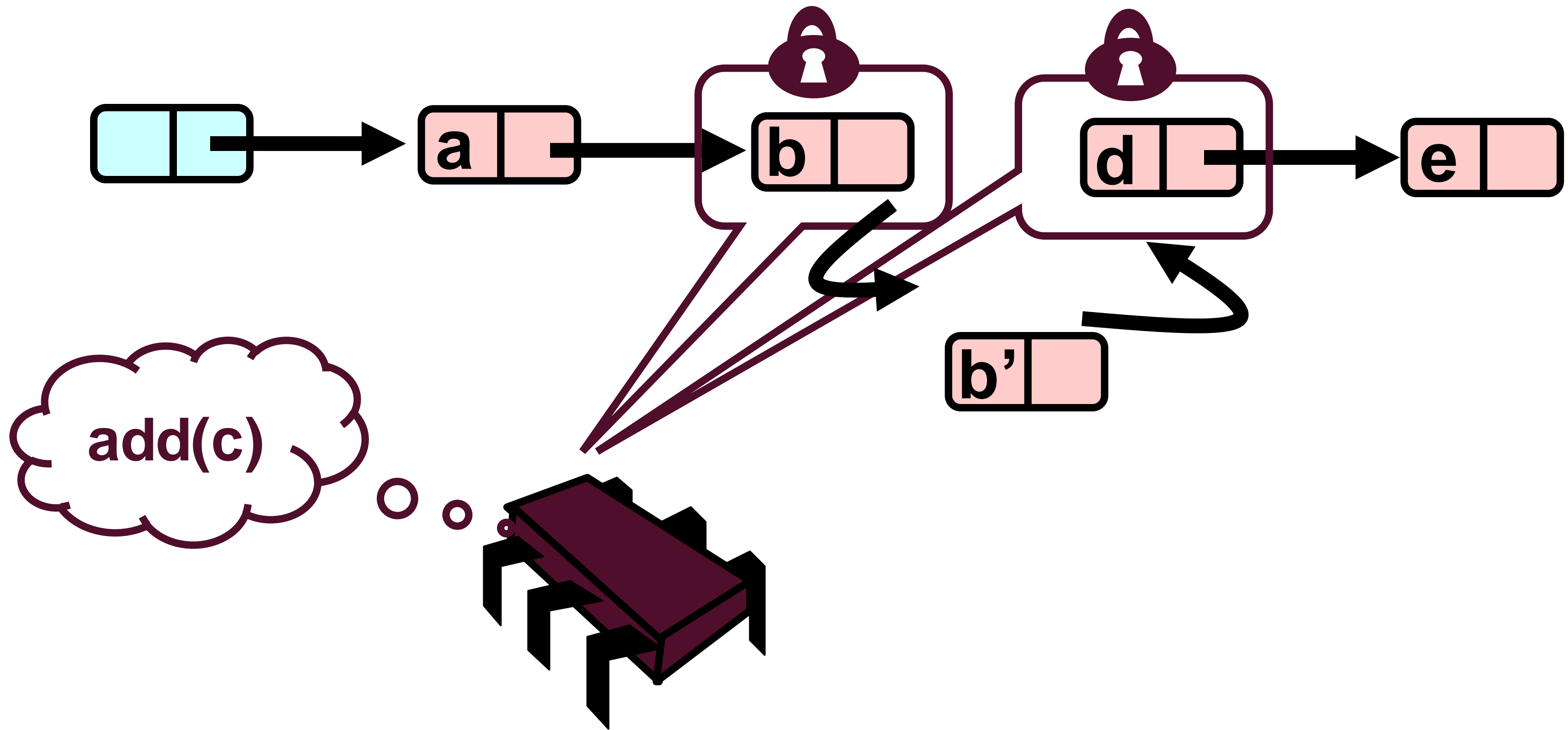
What Else Could Go Wrong?



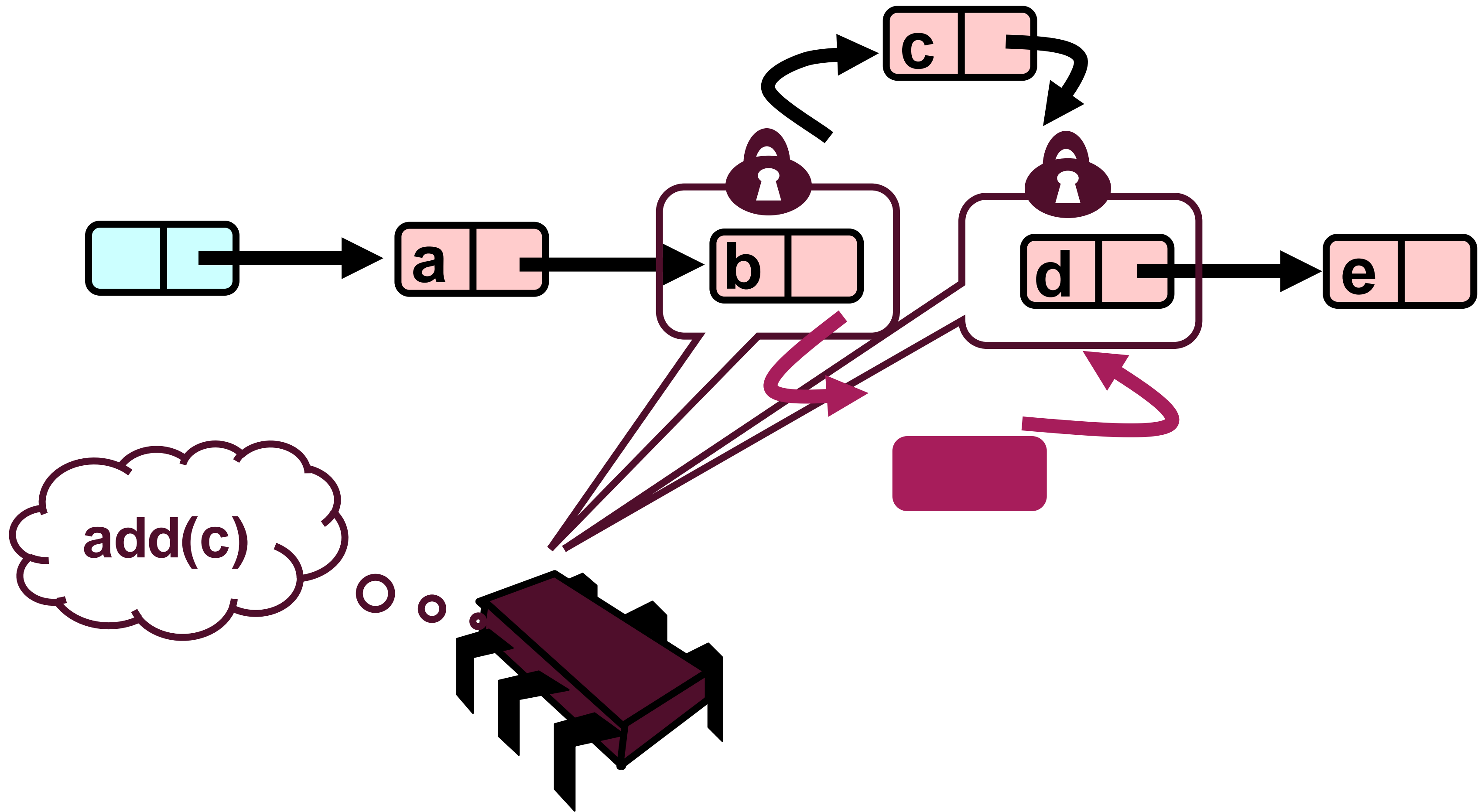
What Else Could Go Wrong?



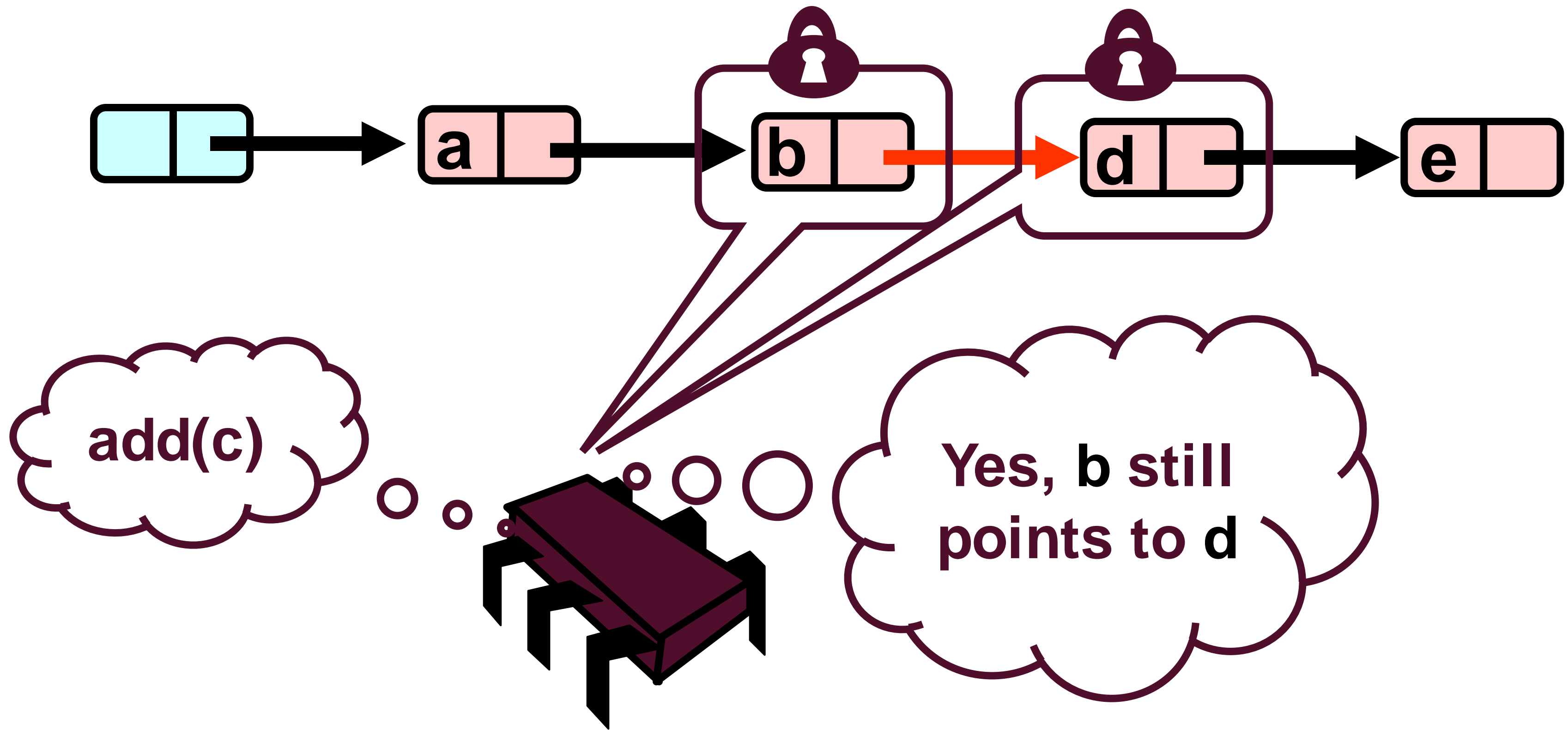
What Else Could Go Wrong?



What Else Could Go Wrong?



Validate Part 2 (while holding locks)



Optimistic synchronization

- **One MUST validate AFTER locking**
  1. Check if the path how we got there is still valid!
  2. Check if locked nodes are still connected
  - If any of those checks fail?

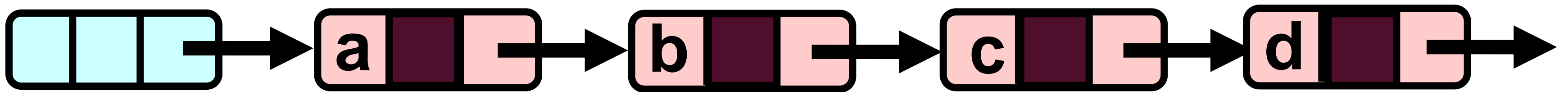
    Start over from the beginning (hopefully rare)
- **Not starvation-free**
  - A thread may need to abort forever if nodes are added/removed
  - Should be rare in practice!
- **Other disadvantages?**
  - All operations requires two traversals of the list!
  - Even contains() needs to check if node is still in the list!

Trick 4: Lazy synchronization

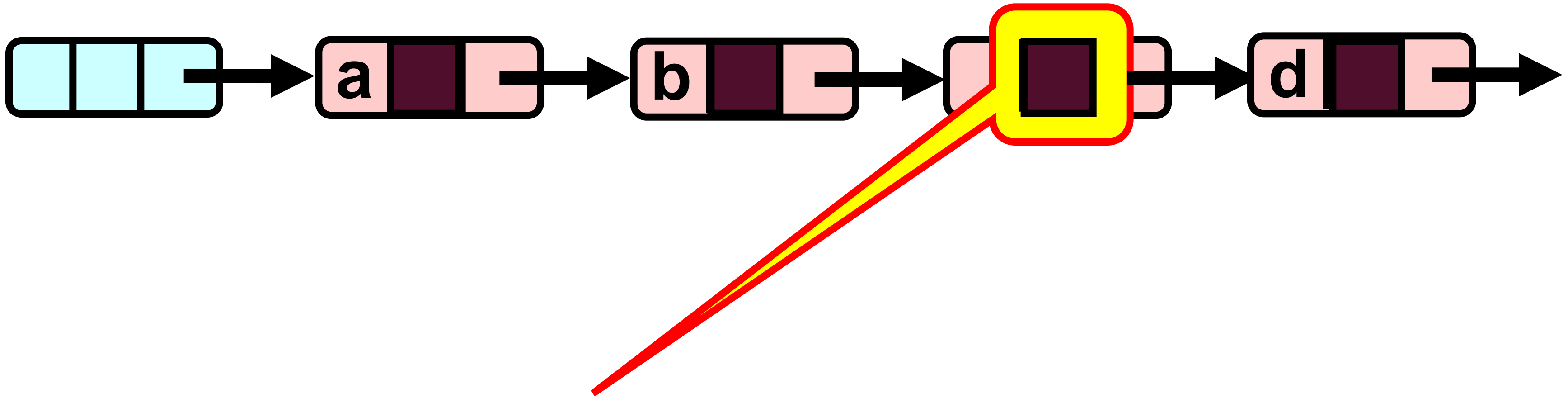
- **We really want one list traversal**
- **Also, contains() should be wait-free**
  - Is probably the most-used operation
- **Lazy locking is similar to optimistic**
  - Key insight: removing is problematic
  - Perform it "lazily"
- **Add a new "valid" field**
  - Indicates if node is still in the set
  - Can remove it without changing list structure!
  - Scan once, contains() never locks!

```
typedef struct {
  int key;
  node *next;
  lock_t lock;
  boolean valid;
} node;
```

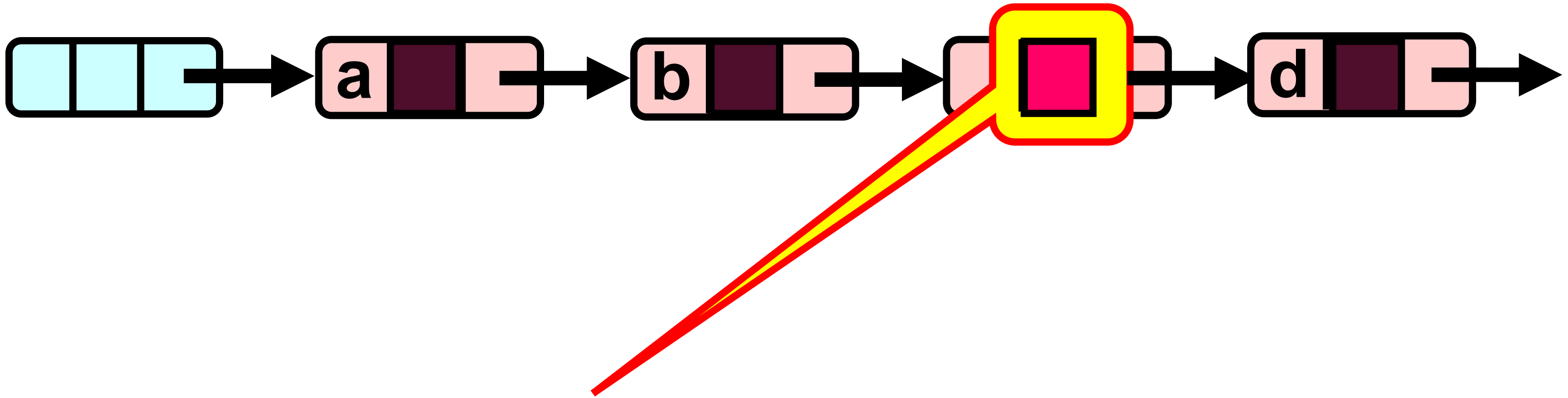
Lazy Removal



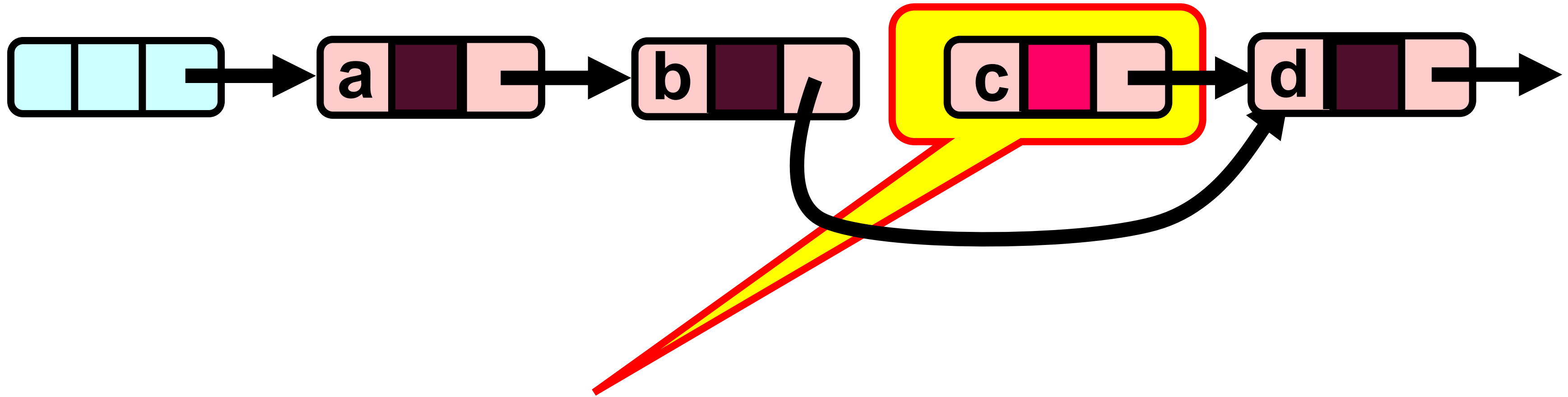
Lazy Removal



Lazy Removal

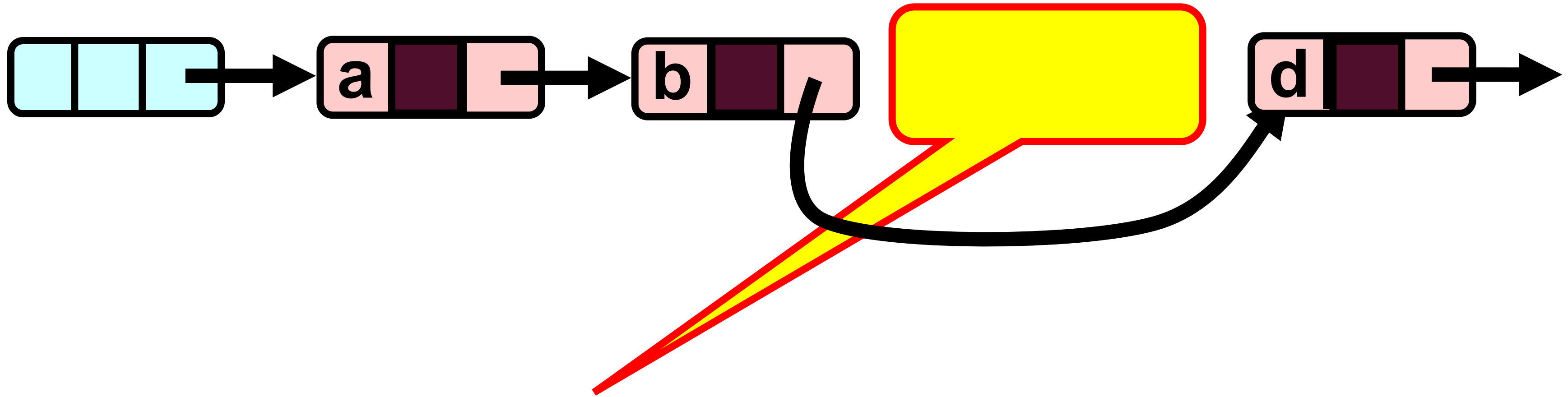


Lazy Removal



Physically deleted

Lazy Removal

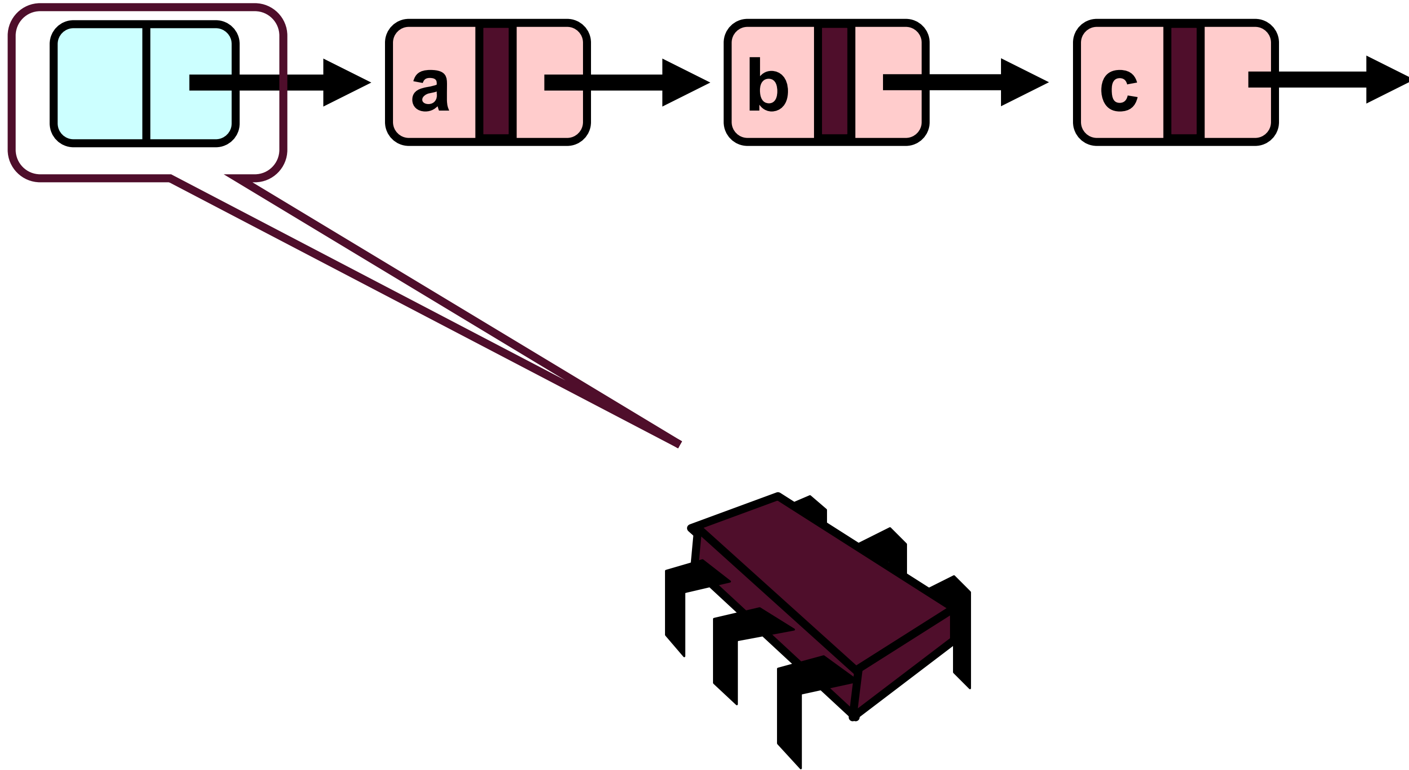


How does it work?

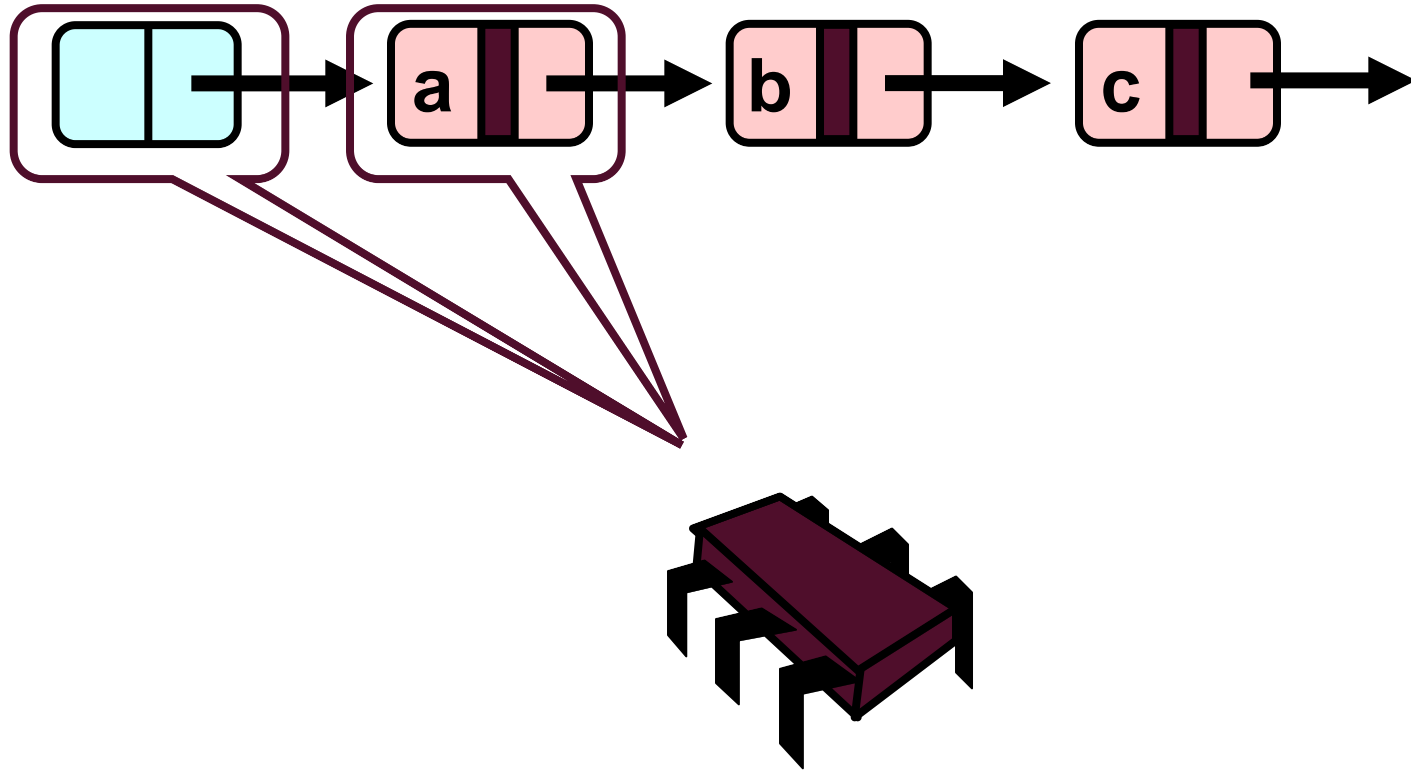
- **Eliminates need to re-scan list for reachability**
  - Maintains invariant that every **unmarked** node is reachable!
- **Contains can now simply traverse the list**
  - Just check marks, not reachability, no locks
- **Remove/Add**
  - Scan through locked and marked nodes
  - Removing does not delay others
  - Must only lock when list structure is updated

    Check if neither pred nor curr are marked, pred.next == curr

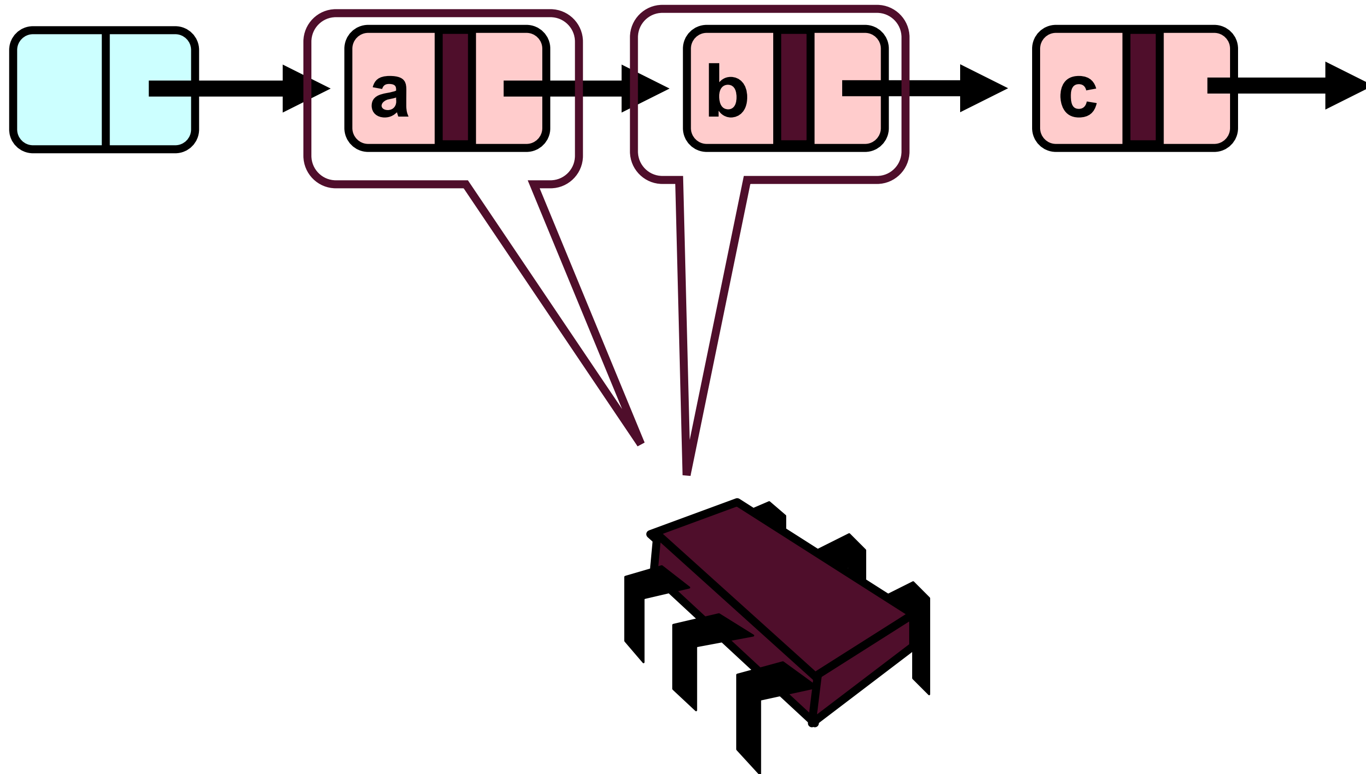
Business as Usual



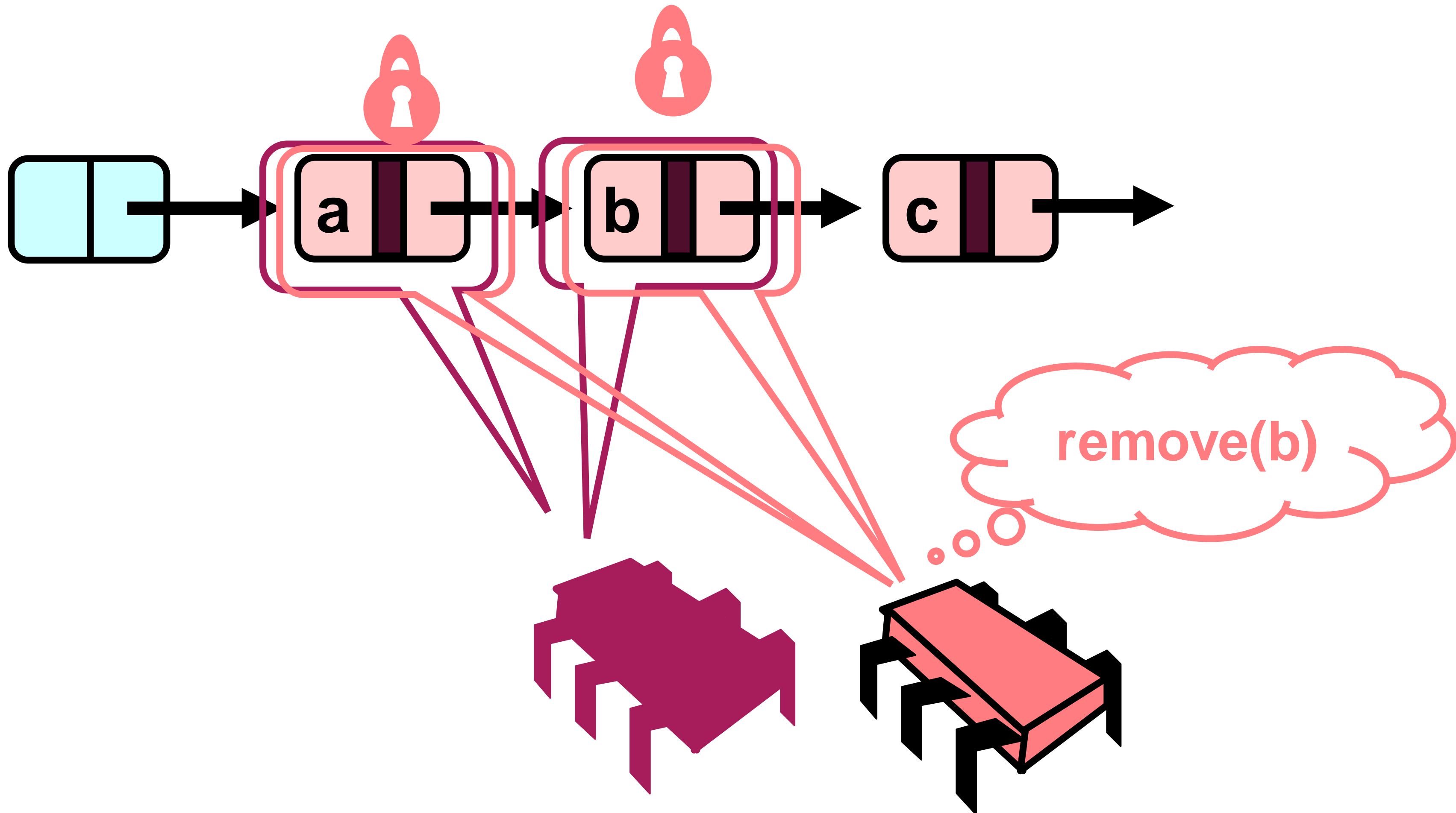
Business as Usual



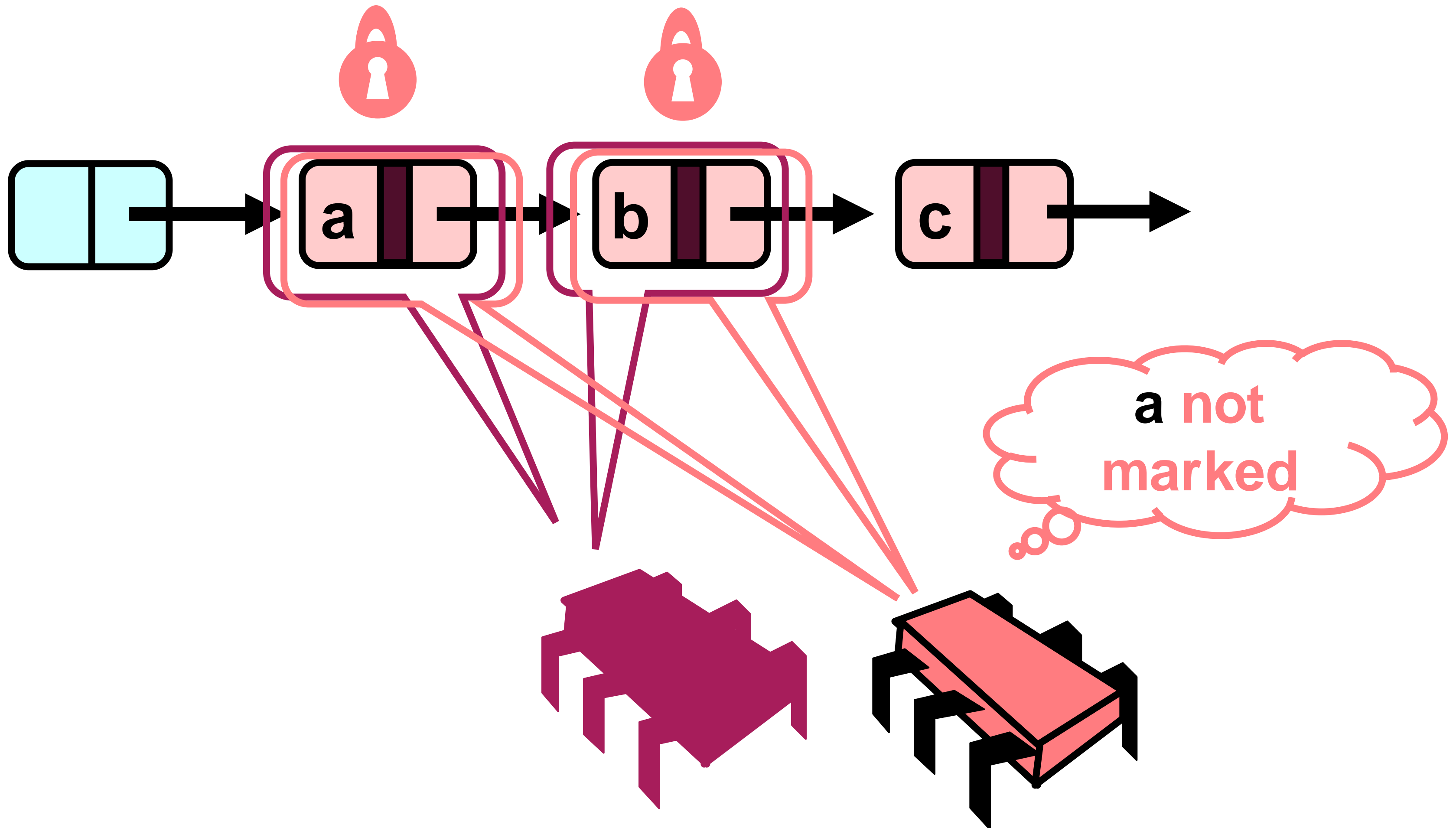
Business as Usual



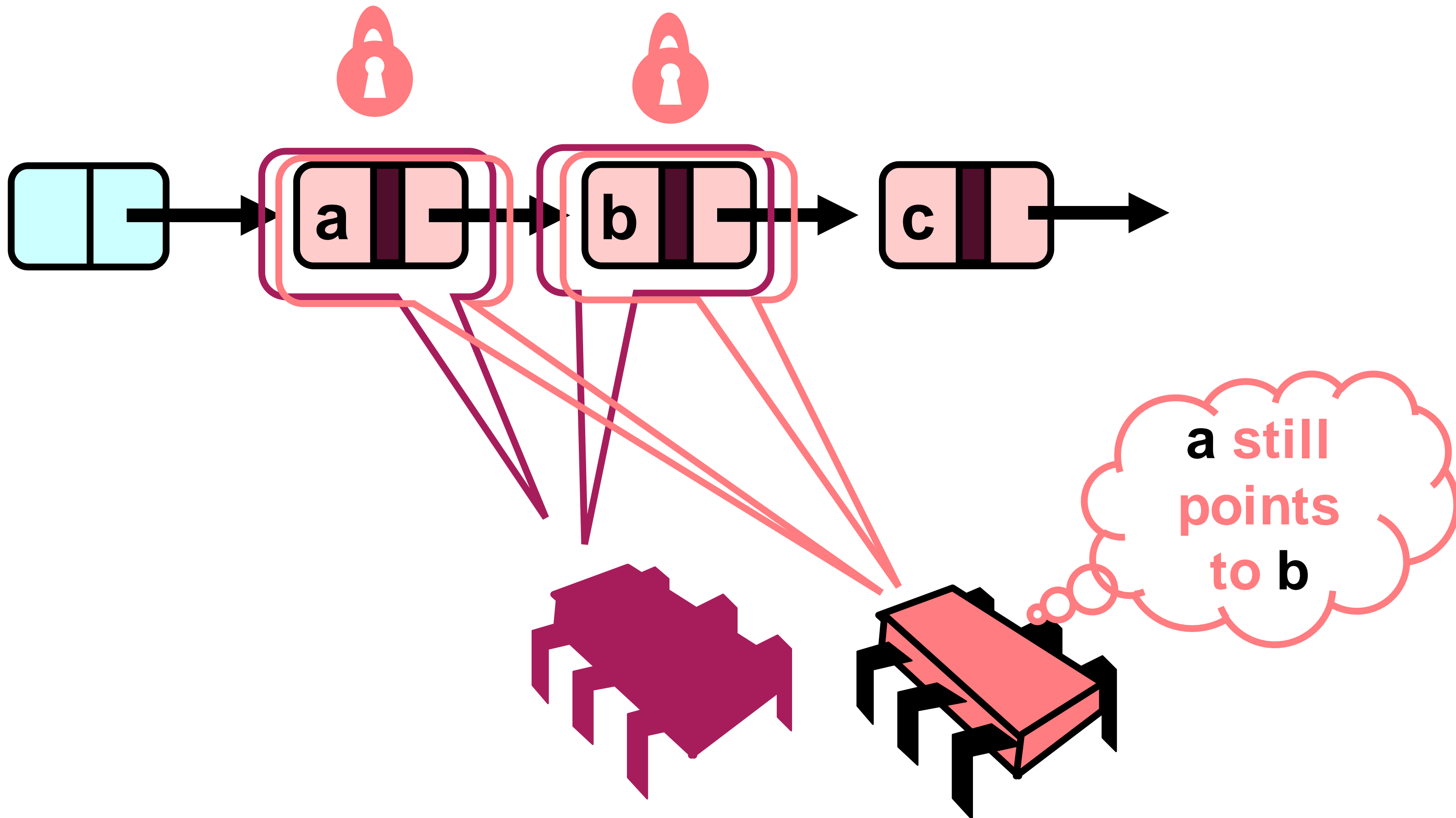
Business as Usual



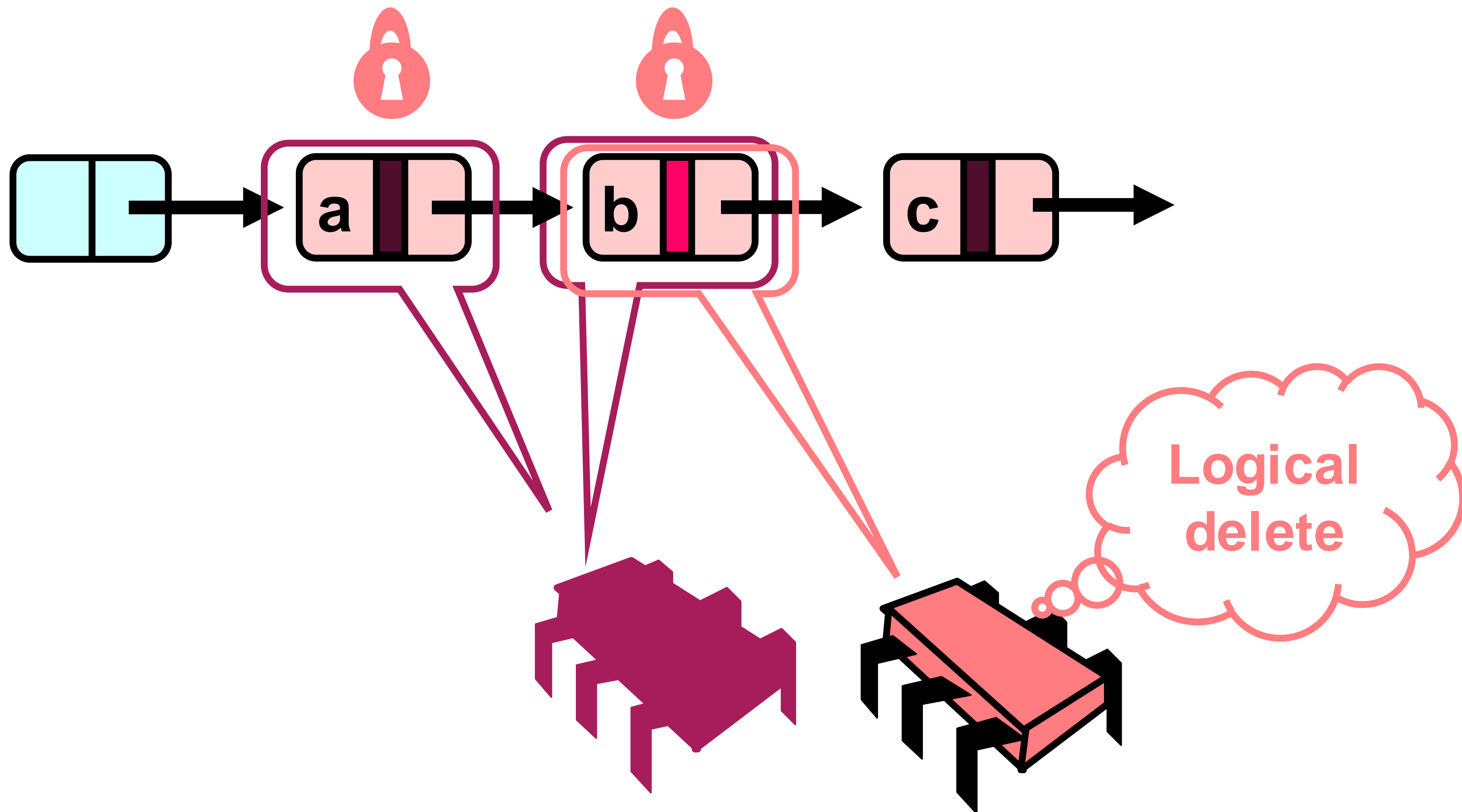
Business as Usual



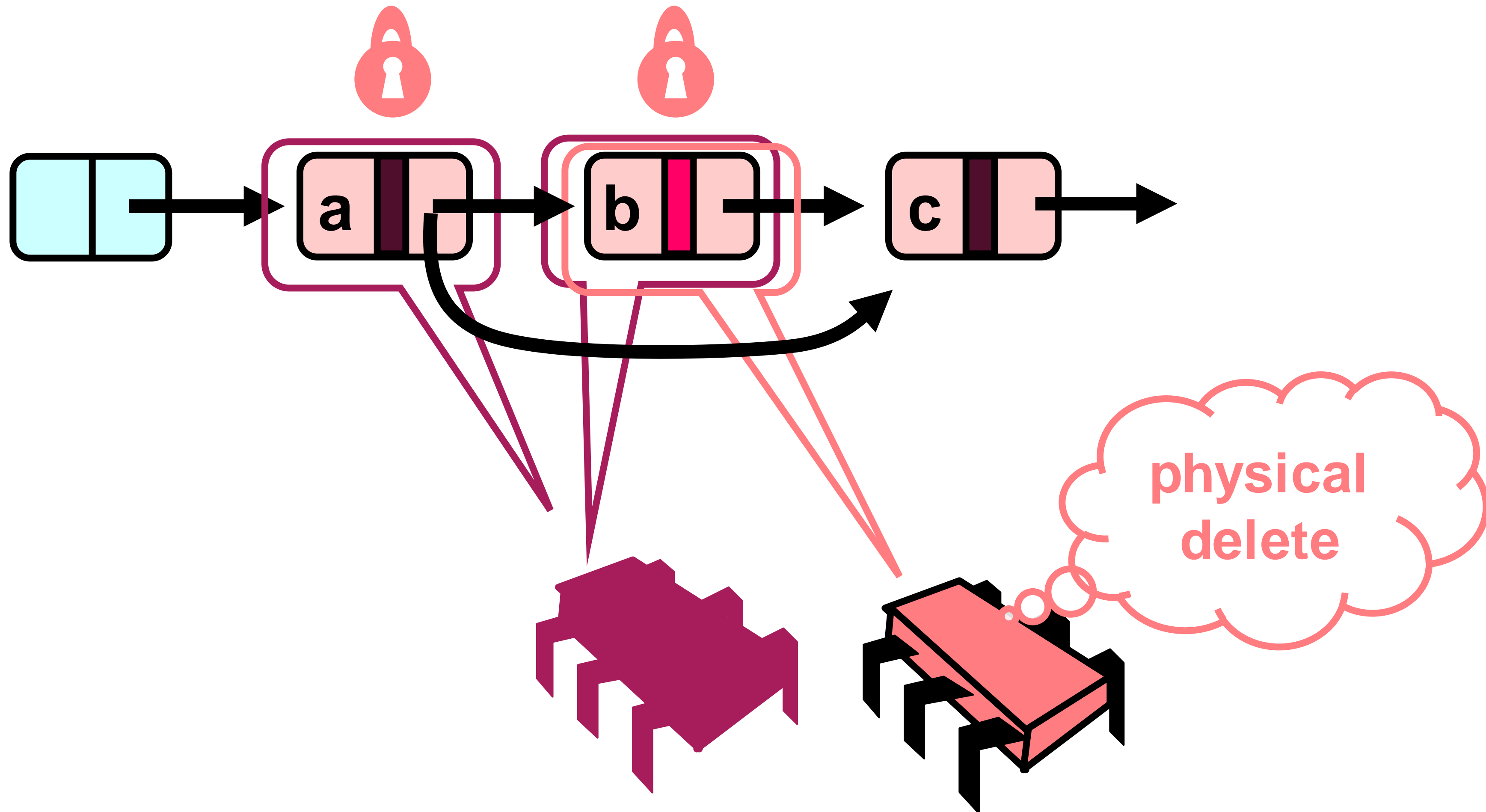
Business as Usual



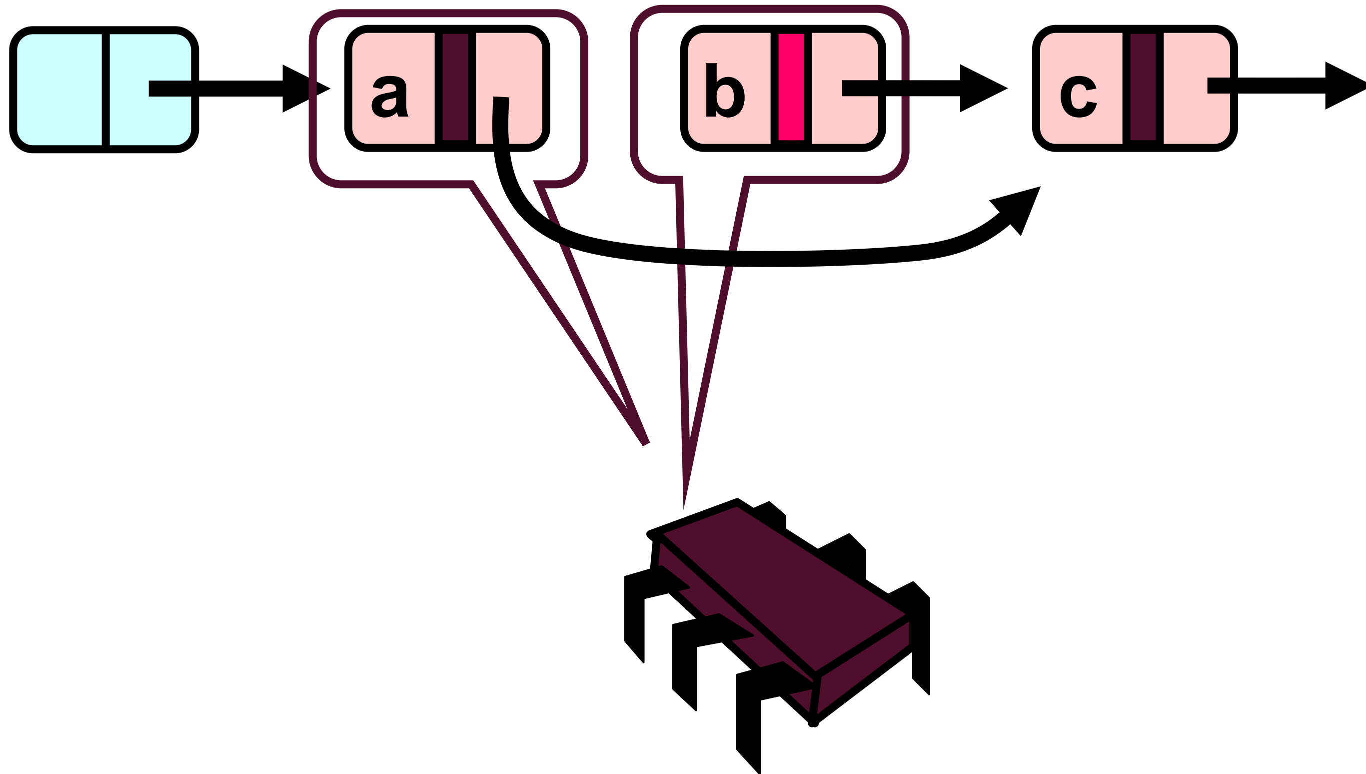
Business as Usual



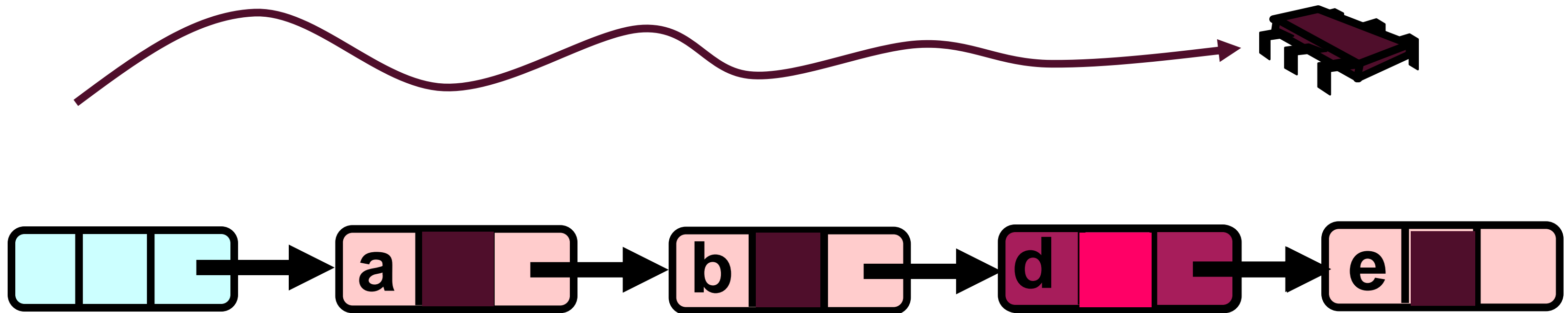
Business as Usual



Business as Usual



Summary: Wait-free Contains



Use Mark bit + list ordering

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

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

Problems with Locks

- **What are the fundamental problems with locks?**
- **Blocking**
  - Threads wait, fault tolerance
  - Especially when things like page faults occur in CR
- **Overheads**
  - Even when not contended
  - Also memory/state overhead
- **Synchronization is tricky**
  - Deadlock, other effects are hard to debug
- **Not easily composable**

Lock-free Methods

- **No matter what:**
  - Guarantee minimal progress

    I.e., some thread will advance
  - Threads may halt at bad times (no CRs! No exclusion!)

    I.e., cannot use locks!
  - Needs other forms of synchronization

    E.g., atomics (discussed before for the implementation of locks)

    Techniques are astonishingly similar to guaranteeing mutual exclusion

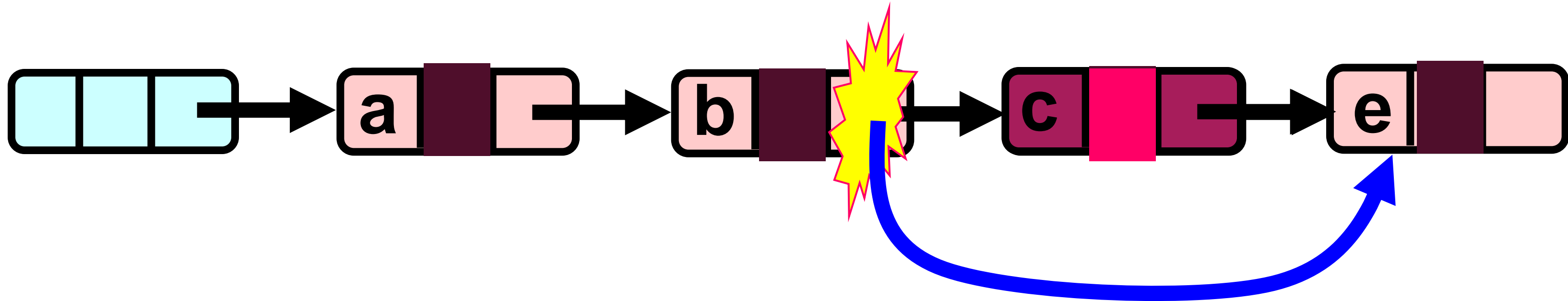
Trick 5: No Locking

- **Make list lock-free**
- **Logical succession**
  - We have wait-free contains
  - Make add() and remove() lock-free!

    Keep logical vs. physical removal
- **Simple idea:**
  - Use CAS to verify that pointer is correct before moving it

Lock-free Lists

(1) Logical Removal

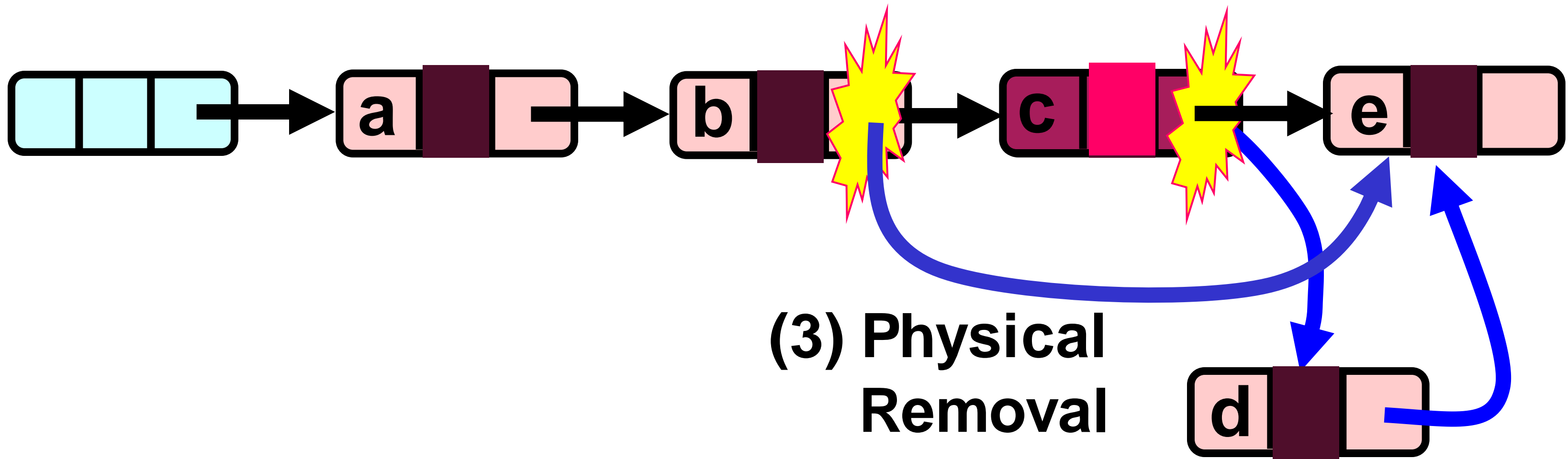


Use CAS to verify pointer is correct

(2) Physical Removal

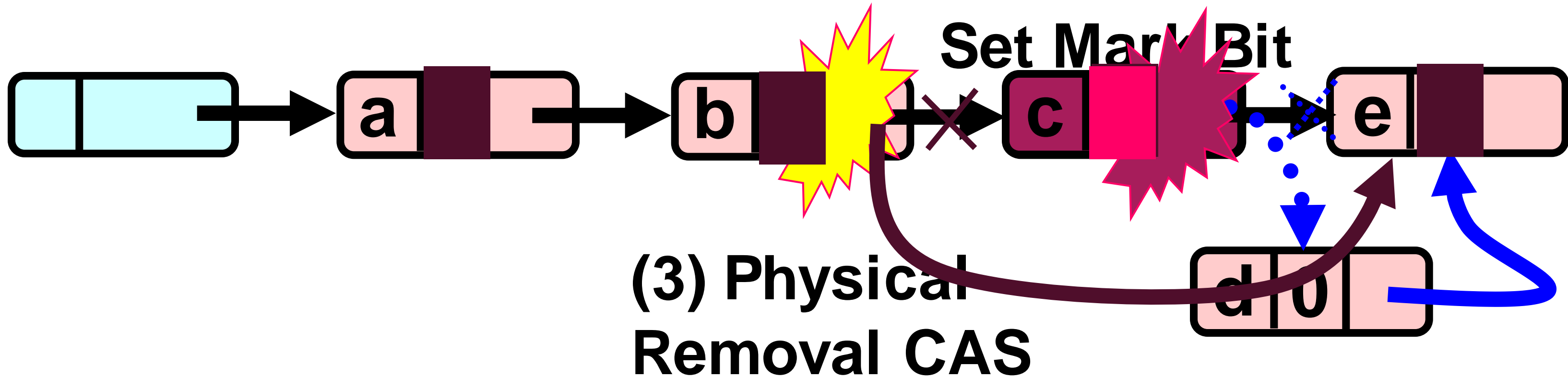
Not enough! Why?

Problem…



The Solution: Combine Mark and Pointer

(1) Logical Removal
=



(2) Fail CAS: Node not added after logical Removal

Mark-Bit and Pointer are CASed together!

Practical Solution(s)

- **Option 1:**
  - Introduce "atomic markable reference" type
  - "Steal" a bit from a pointer
  - Rather complex and OS specific ☹
- **Option 2:**
  - Use Double CAS (or CAS2) ☺

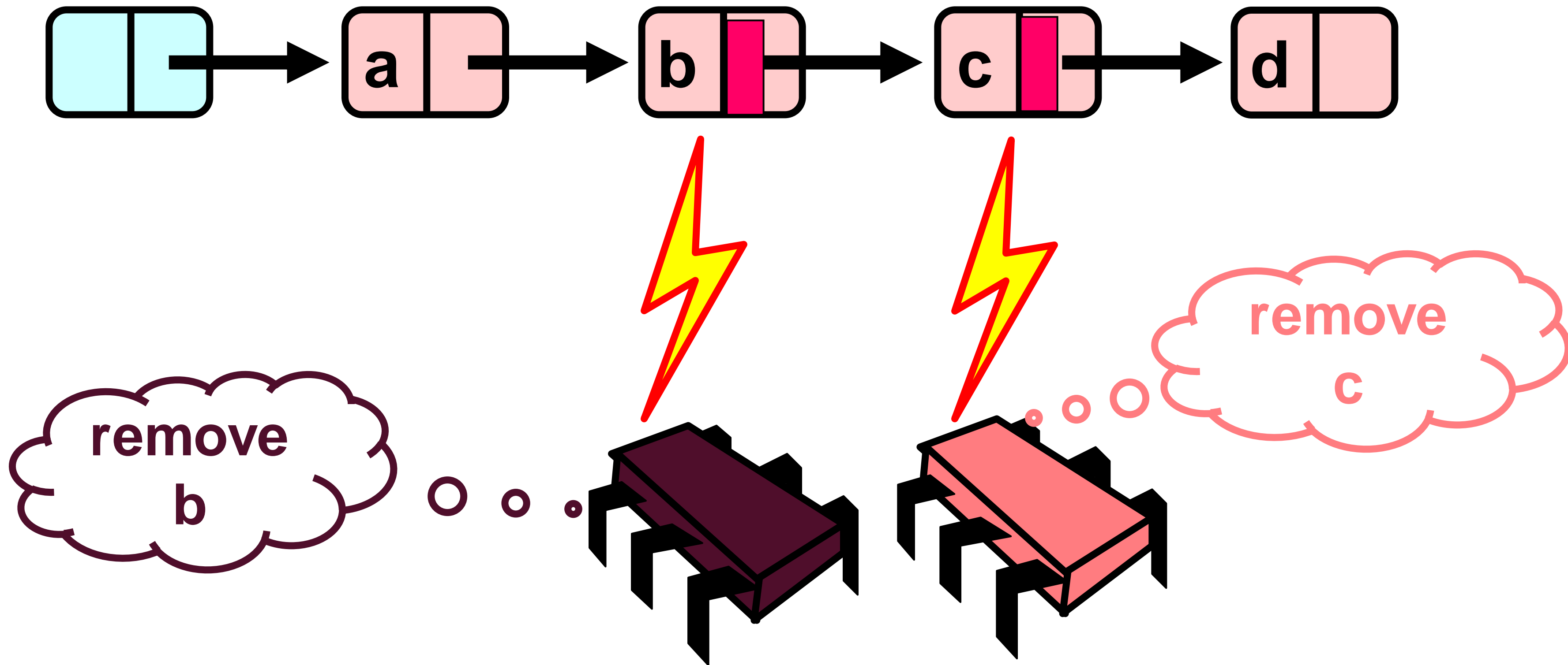
    CAS of two noncontiguous locations
  - Well, not many machines support it ☹

    Any still alive?
- **Option 3:**
  - Our favorite ISA (x86) offers double-width CAS

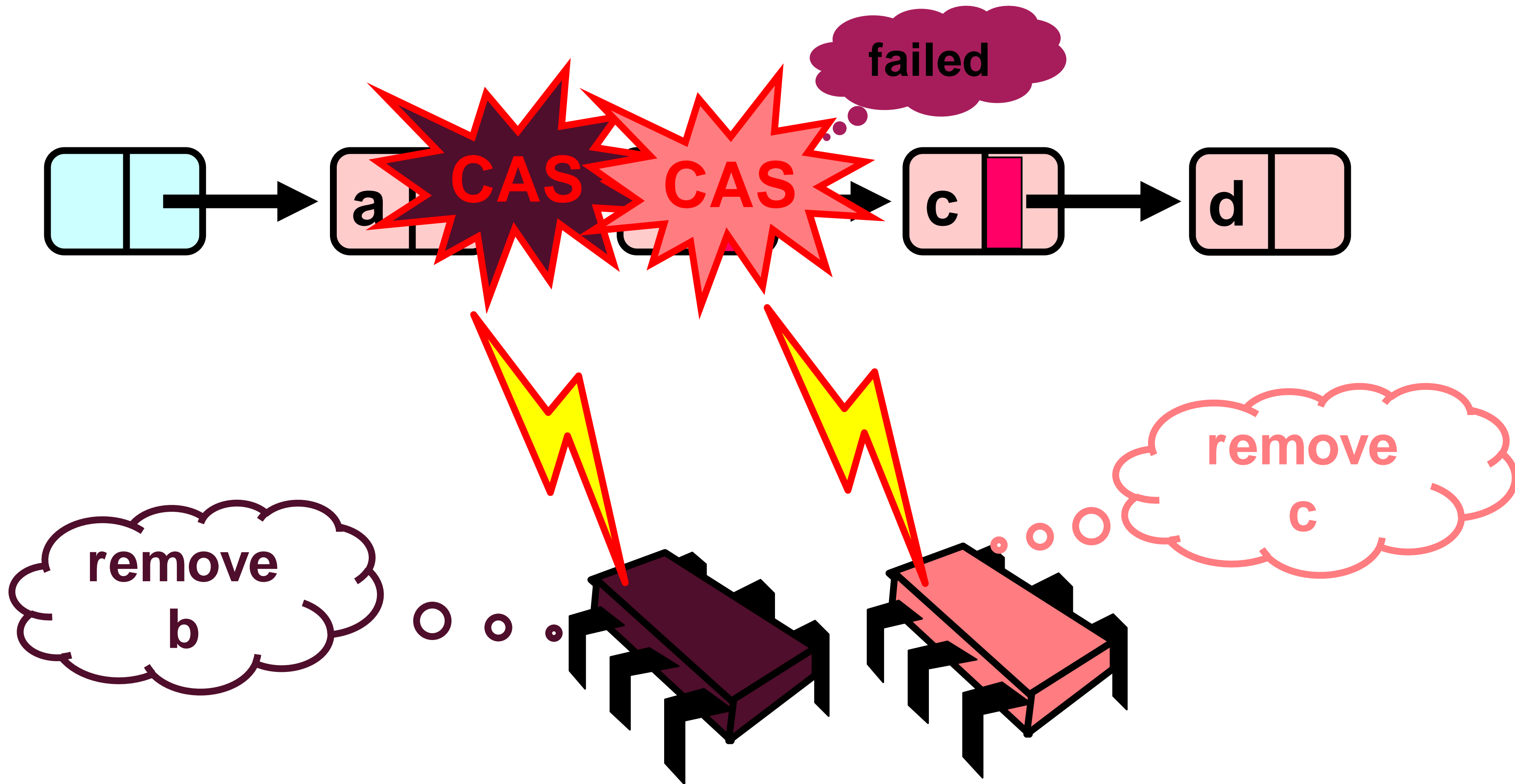
    Contiguous, e.g., lock cmpxchg16b (on 64 bit systems)
- **Option 4:**
  - TM!

    E.g., Intel's TSX (essentially a cmpxchg64b (operates on a cache line))

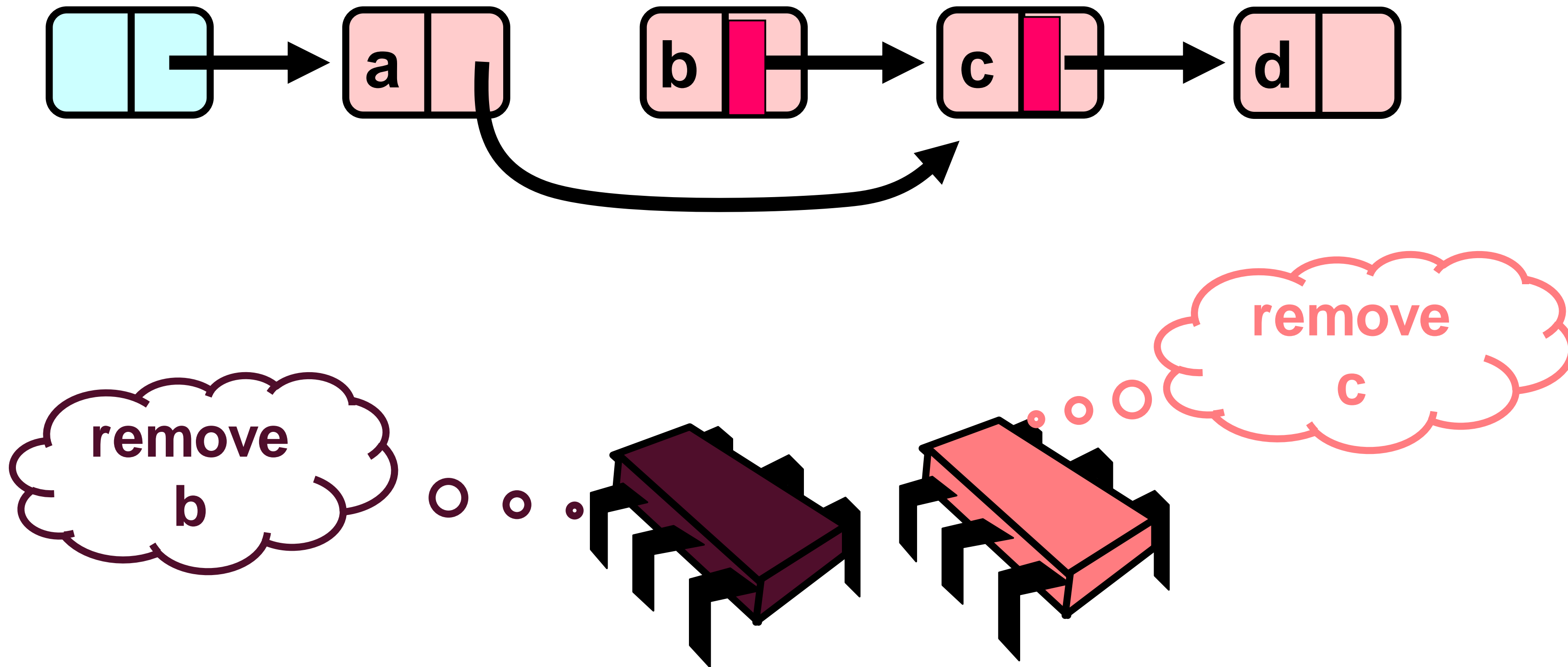
Removing a Node



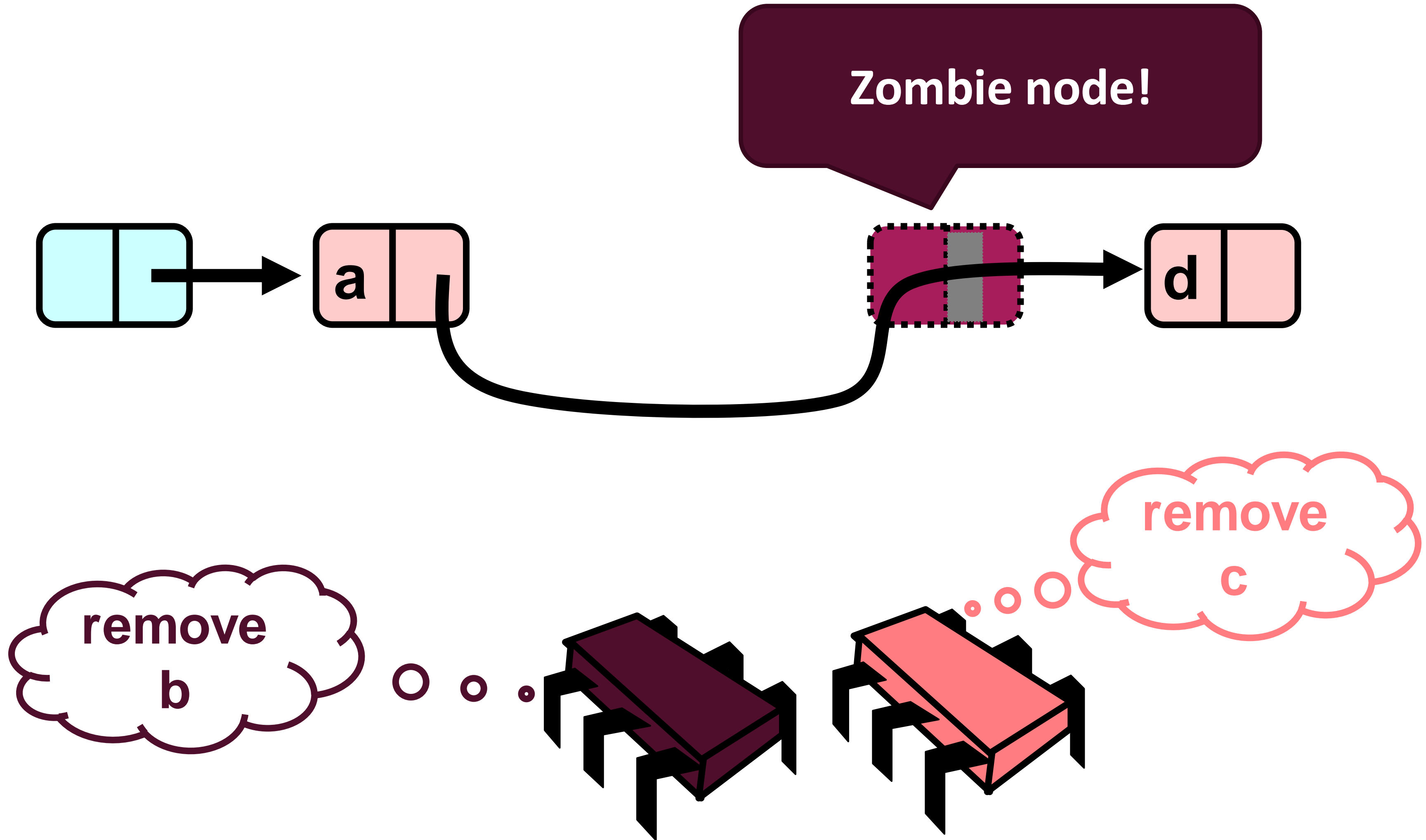
Removing a Node



Removing a Node



Uh oh – node marked but not removed!



Dealing With Zombie Nodes

- **Add() and remove() "help to clean up"**
  - Physically remove any marked nodes on their path
  - I.e., if curr is marked: CAS (pred.next, mark) to (curr.next, false) and remove curr

    If CAS fails, restart from beginning!
- **"Helping" is often needed in wait-free algs**
- **This fixes all the issues and makes the algorithm correct!**

Comments

- **Atomically updating two variables (CAS2 etc.) has a non-trivial cost**
- **If CAS fails, routine needs to re-traverse list**
  - Necessary cleanup may lead to unnecessary contention at marked nodes
- **More complex data structures and correctness proofs than for locked versions**
  - But guarantees progress, fault-tolerant and maybe even faster (that really depends)

More Comments

- **Correctness proof techniques**
  - Establish invariants for initial state and transformations

    E.g., head and tail are never removed, every node in the set has to be reachable from head, …
  - Proofs are similar to those we discussed for locks

    Very much the same techniques (just trickier)

    Using sequential consistency (or consistency model of your choice ☺)

    Lock-free gets somewhat tricky
- **Source-codes can be found in Chapter 9 of “The Art of Multiprocessor Programming”**