Design of Parallel and High-Performance Computing

Fall 2016

Lecture: Lock-Free and distributed memory

Motivational video: https://www.youtube.com/watch?v=PuCx50FdSic

Instructor: Torsten Hoefler & Markus Püschel

TA: Salvatore Di Girolamo

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Administrivia

- **Final project presentation: Monday 12/19 (two weeks)**
  - Should have (pretty much) final results
  - Show us how great your project is
  - Some more ideas what to talk about:

    Which architecture(s) did you test on?

    How did you verify correctness of the parallelization?

    Use bounds models for comparisons!

    (Somewhat) realistic use-cases and input sets?

    Emphasize on the key concepts (may relate to theory of lecture)!

    What are remaining issues/limitations?

- **Report will be due in January!**
  - Still, starting to write early is very helpful --- write – rewrite – rewrite (no joke!)
  - Last unit today: Entertainment with bogus results!

Review of last lecture

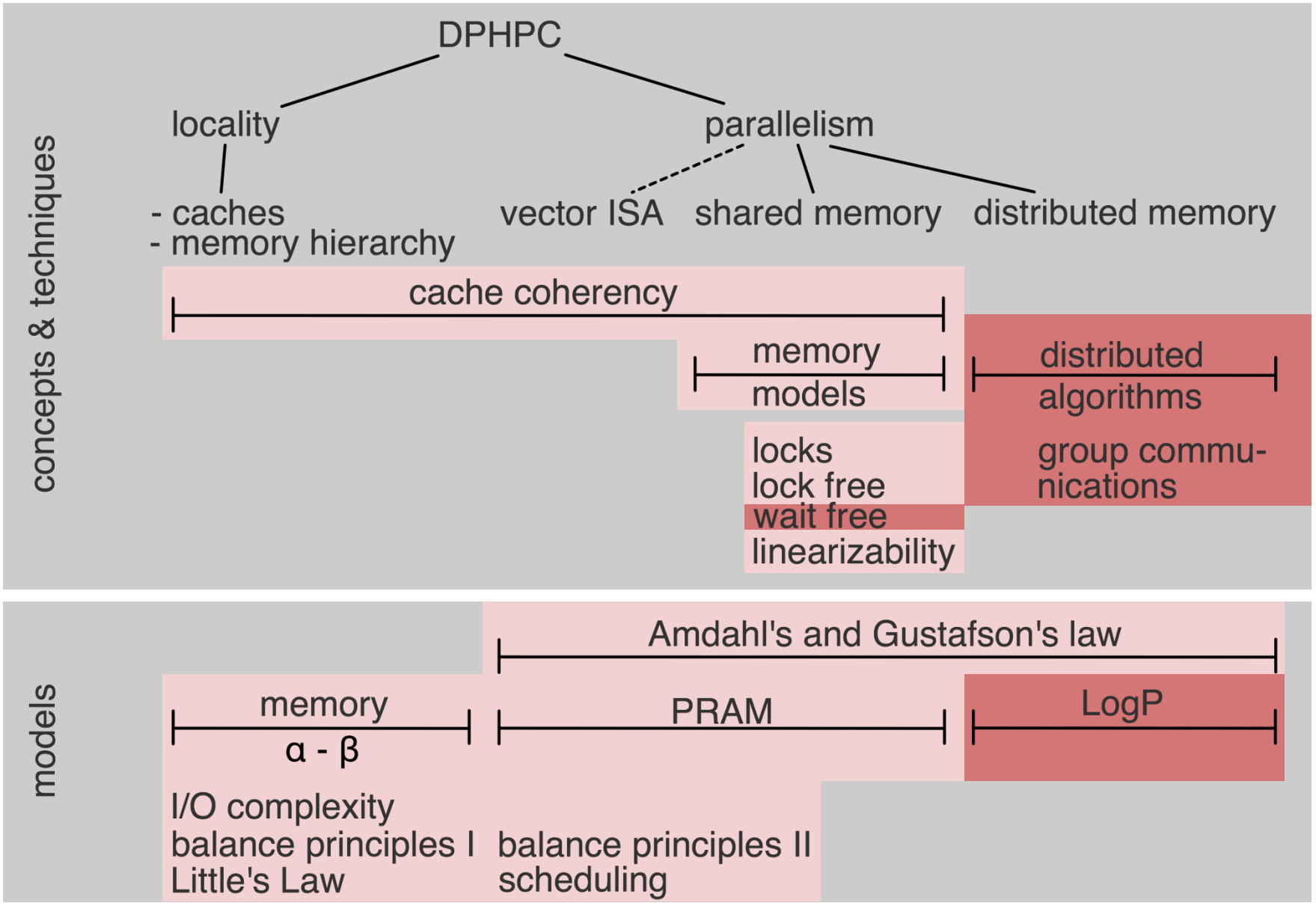
- **Various multi-process locks**
  - Bakery
  - Spinning locks

    Contention issues etc.
  - Queue-based locks

    CLH, MCS

- **MCS – do not forget ☺**
  - RW locks
  - Lock properties/issues (deadlock, priority inversion, blocking vs. spinning)
  - Competitive spinning

DPHPC Overview



Goals of this lecture

- **Locked and Lock-free tricks**
  - (coarse-grained locking)
  - Fine-grained locking
  - RW locking
  - Optimistic synchronization
  - Lazy locking
  - Lock-free (& wait-free)
- **Finish wait-free/lock-free**
  - Consensus hierarchy
  - The promised proof!
- **Maybe: Scientific benchmarking!**
  - Common mistakes!
  - How to improve current practice
  - Important for your project

    Brush up your statistics

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

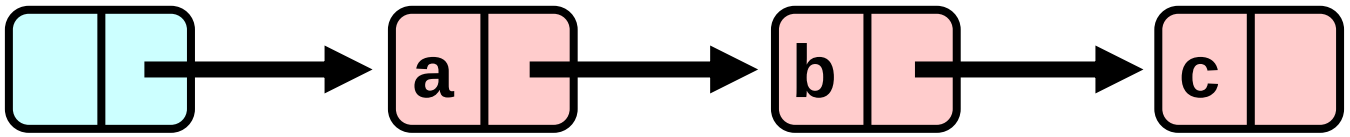
Trick 1: Fine-grained Locking

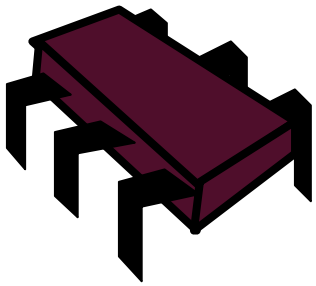
- **Each element can be locked**
  - High memory overhead
  - Threads can traverse list concurrently like a pipeline
- **Tricky to prove correctness**
  - And deadlock-freedom
  - Two-phase locking (acquire, release) often helps
- **Hand-over-hand (coupled locking)**
  - Not safe to release x's lock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

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

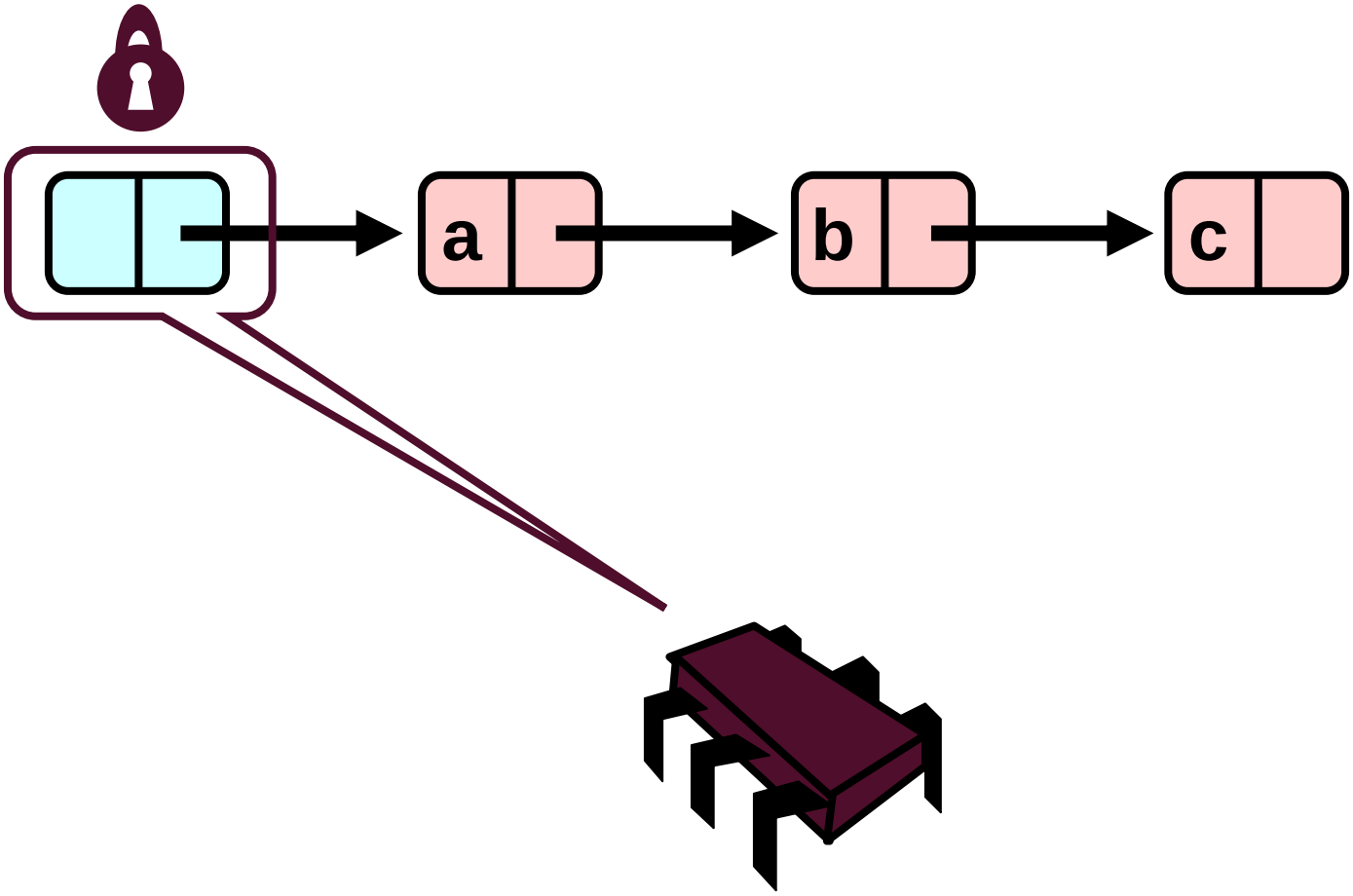
```
typedef struct {
  int key;
  node *next;
  lock_t lock;
} node;
```

Hand-over-Hand (fine-grained) locking

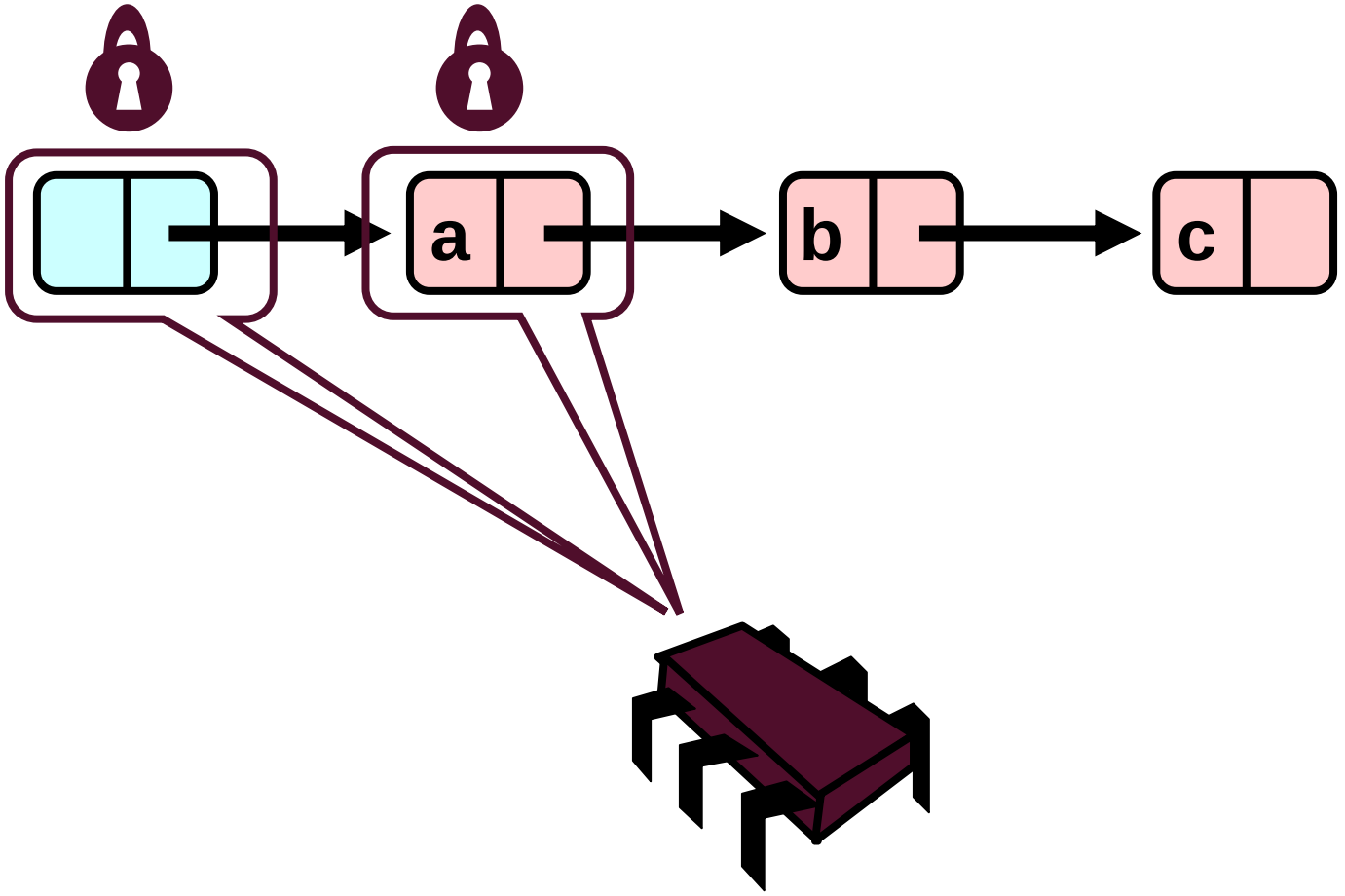




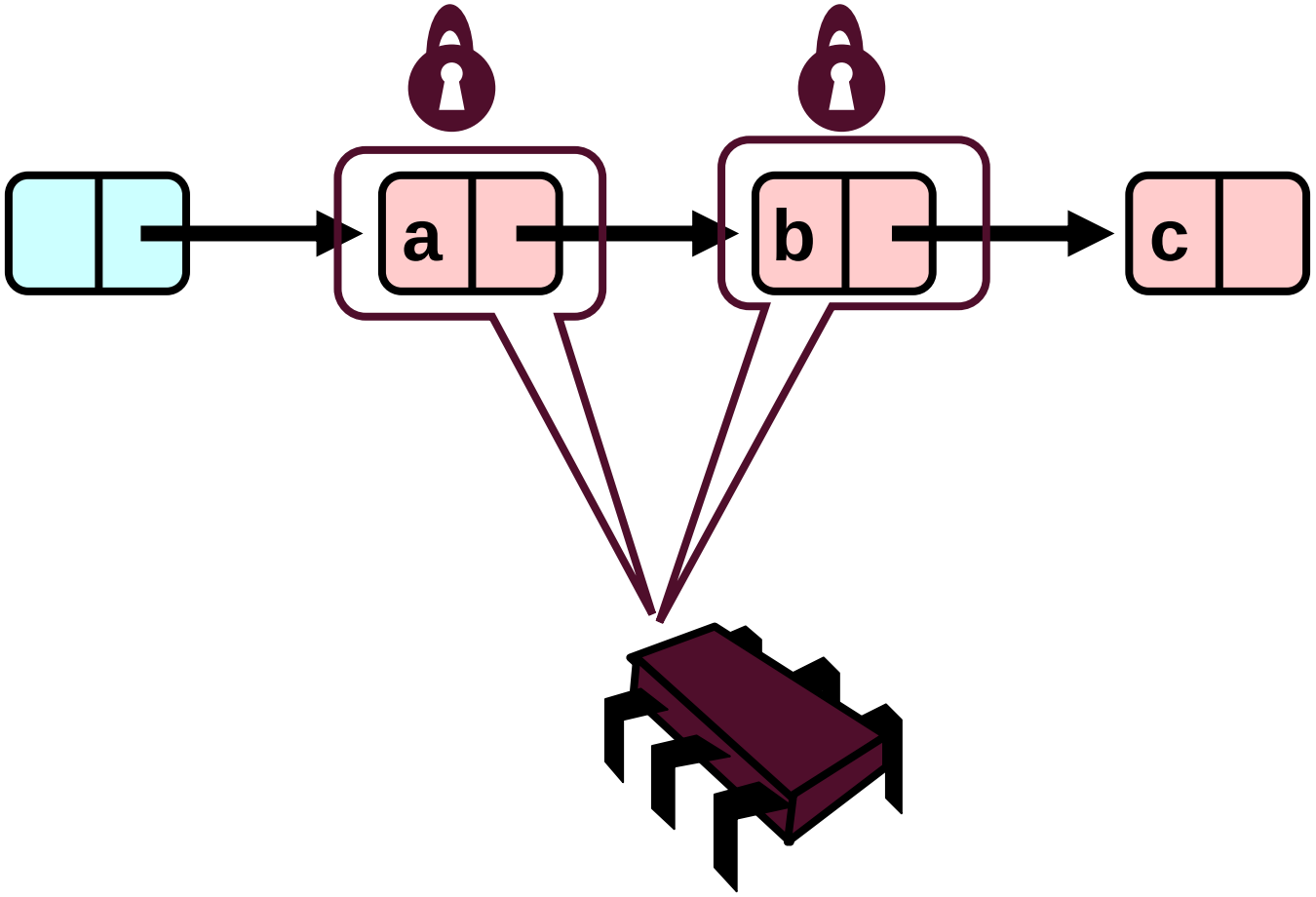
Hand-over-Hand (fine-grained) locking



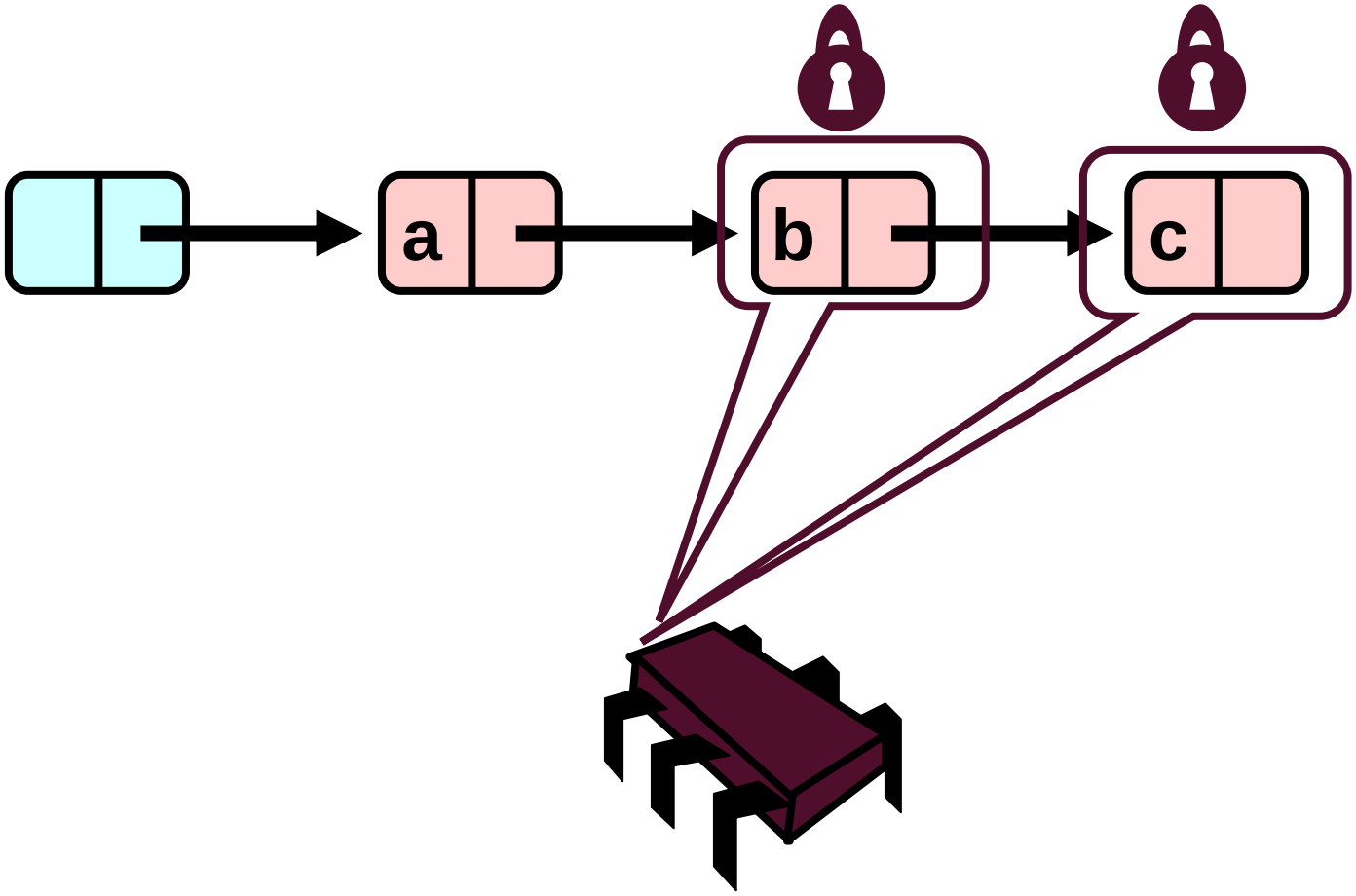
Hand-over-Hand (fine-grained) locking



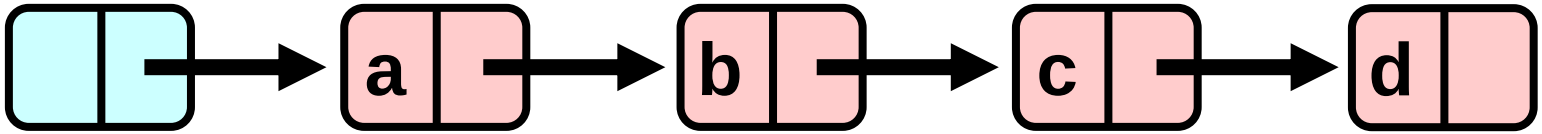
Hand-over-Hand (fine-grained) locking

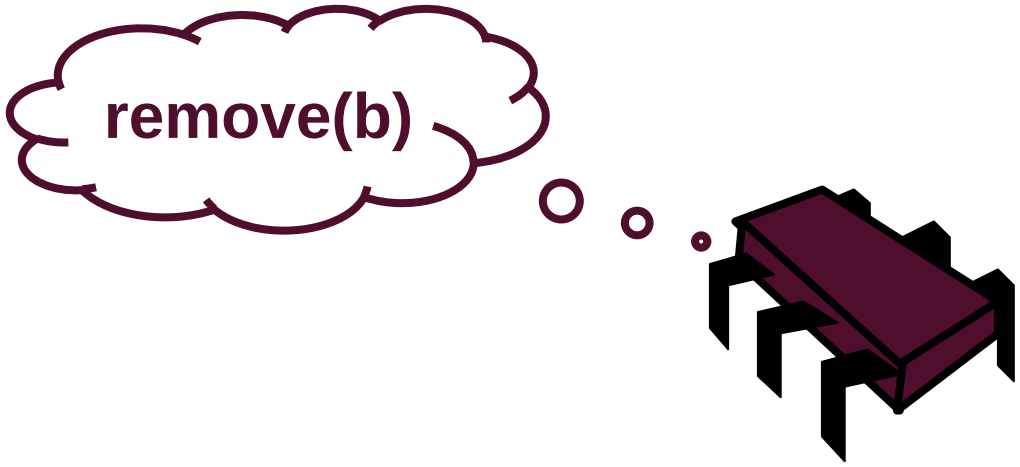


Hand-over-Hand (fine-grained) locking

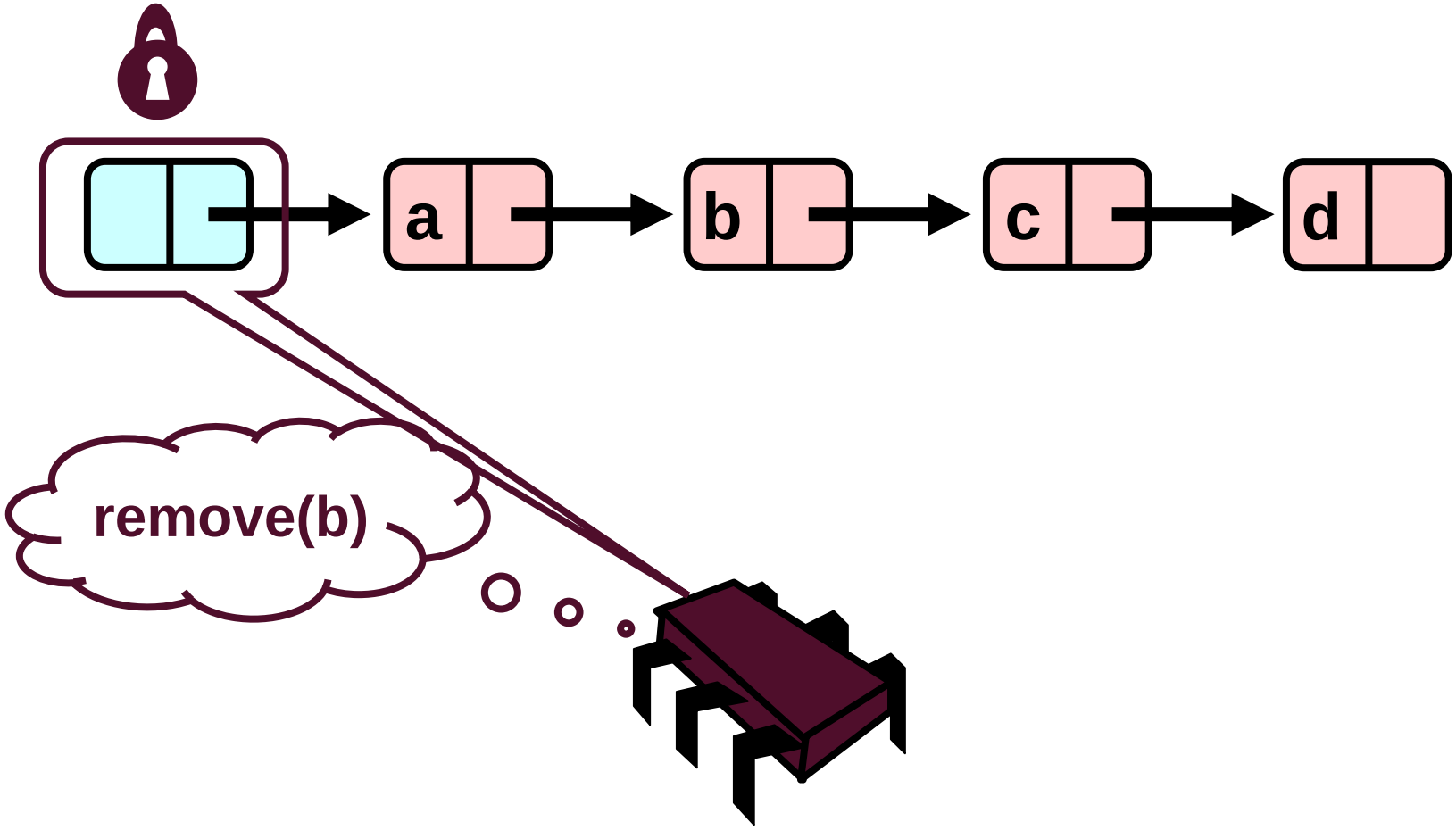


Removing a Node

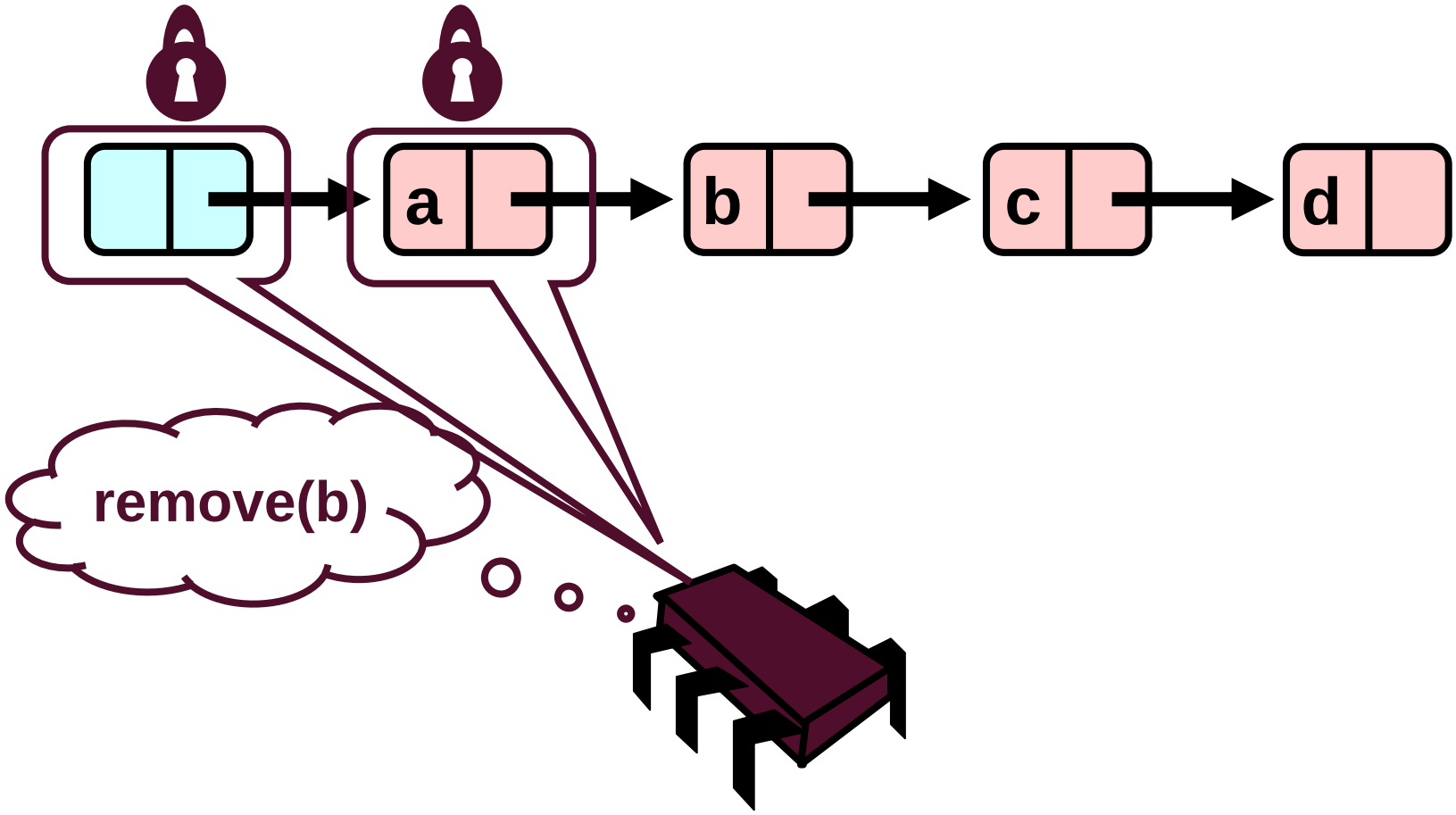




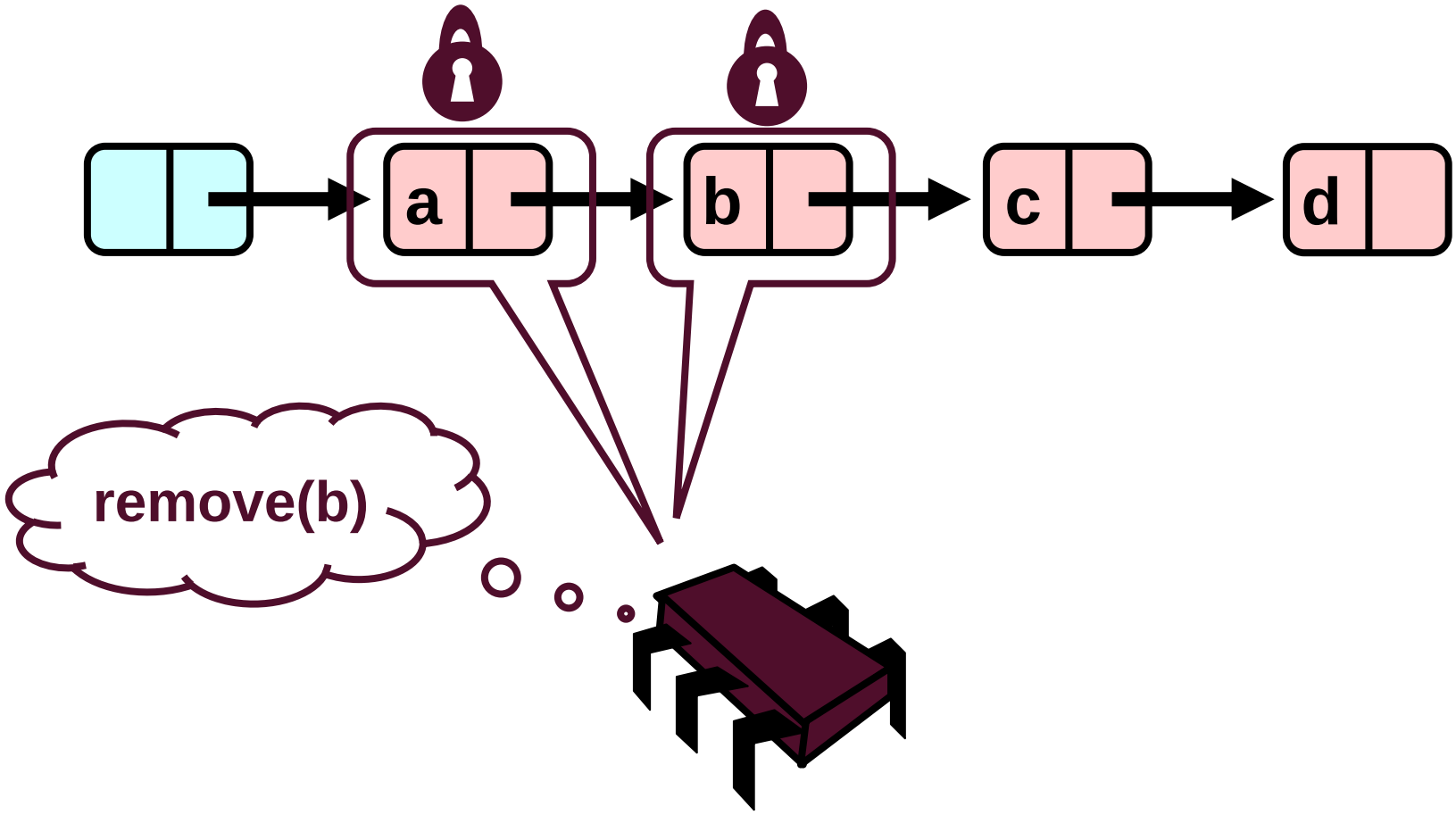
Removing a Node



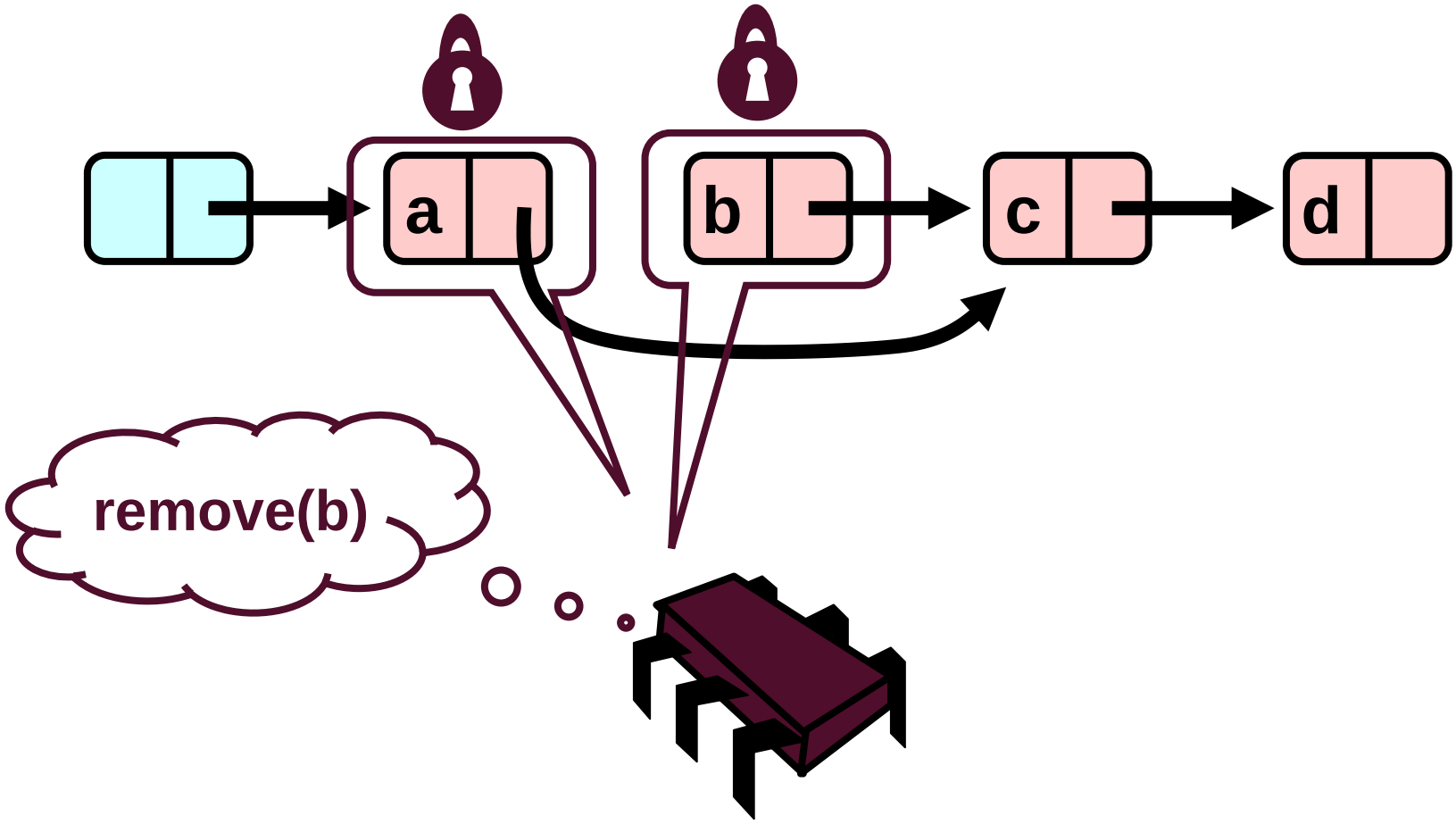
Removing a Node



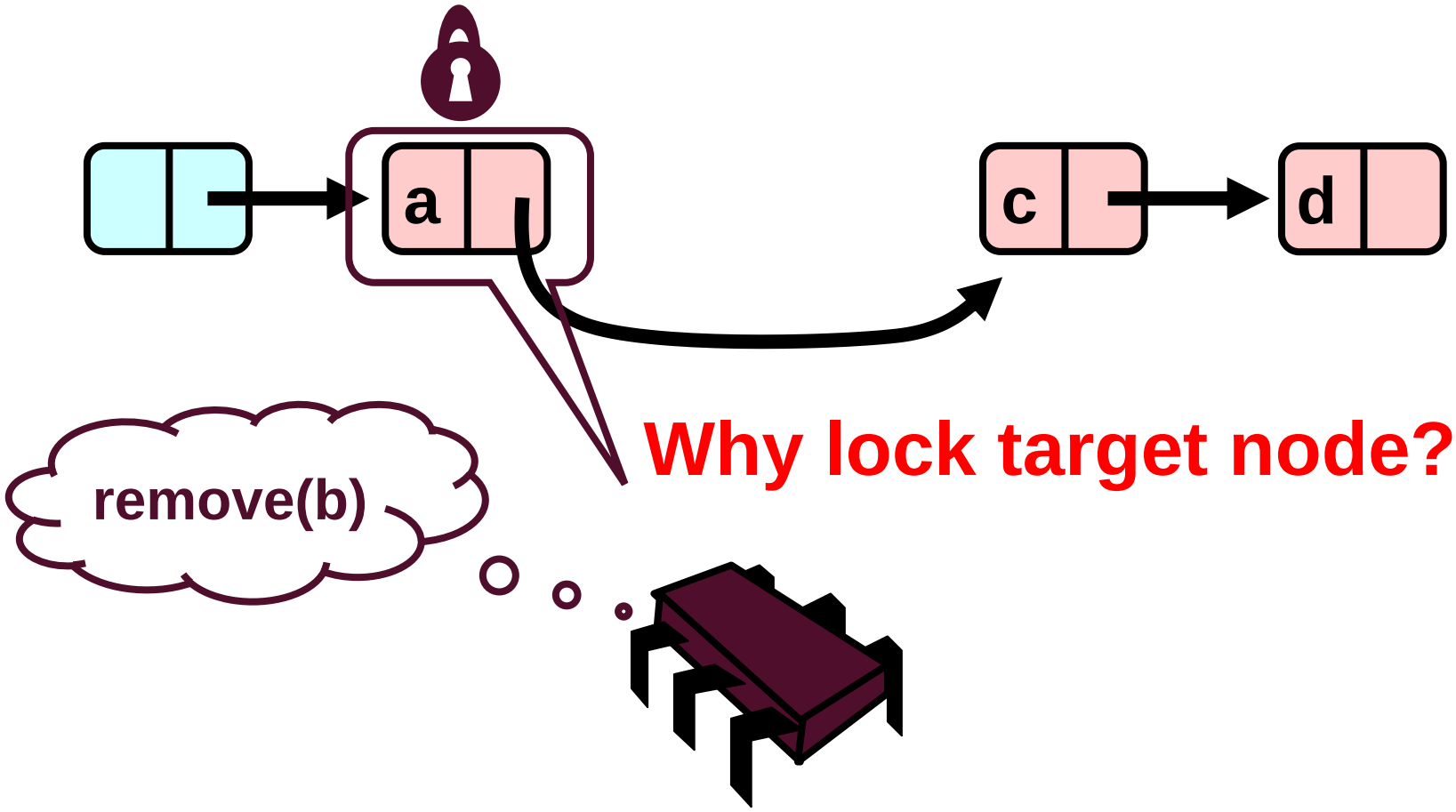
Removing a Node



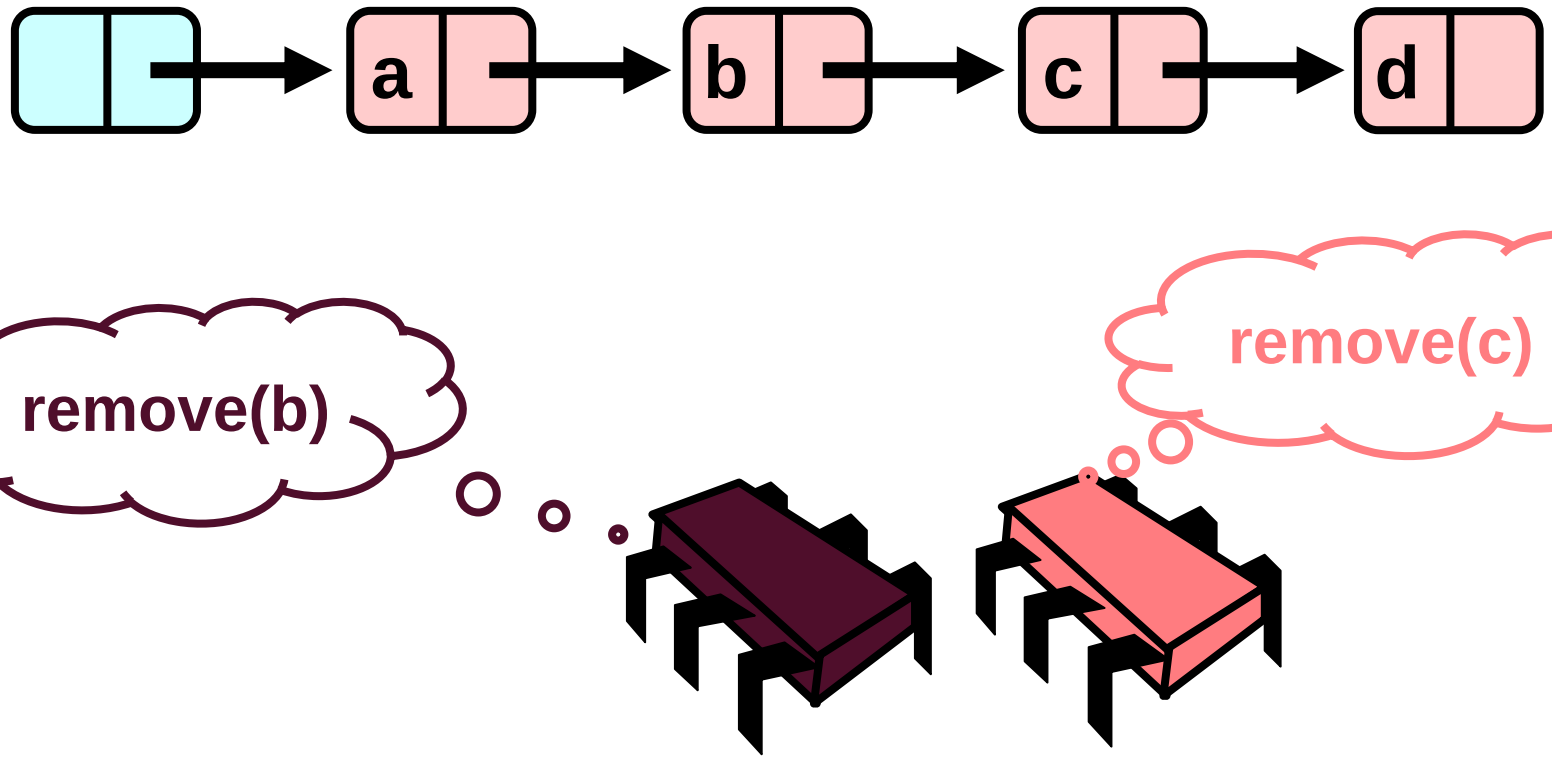
Removing a Node



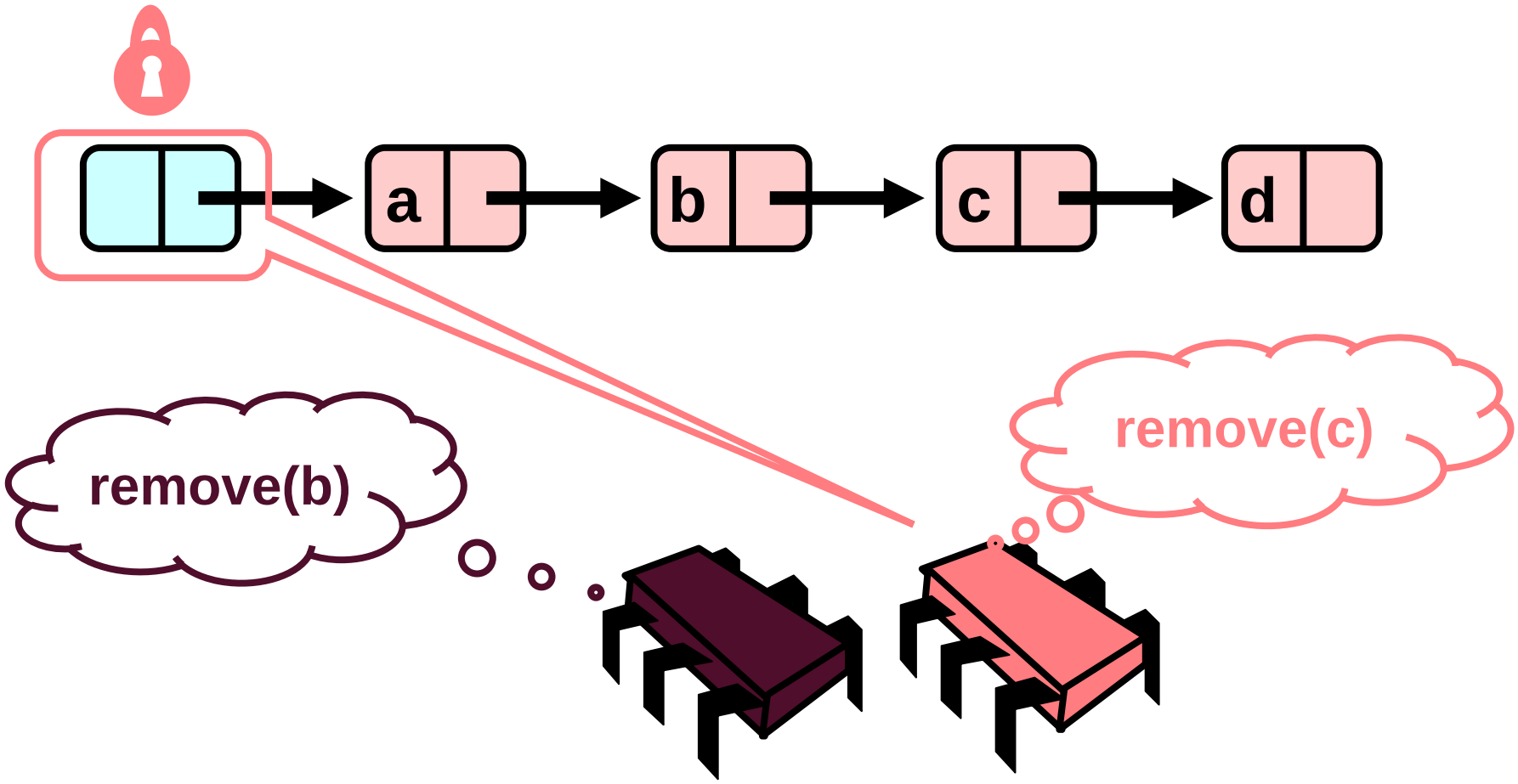
Removing a Node



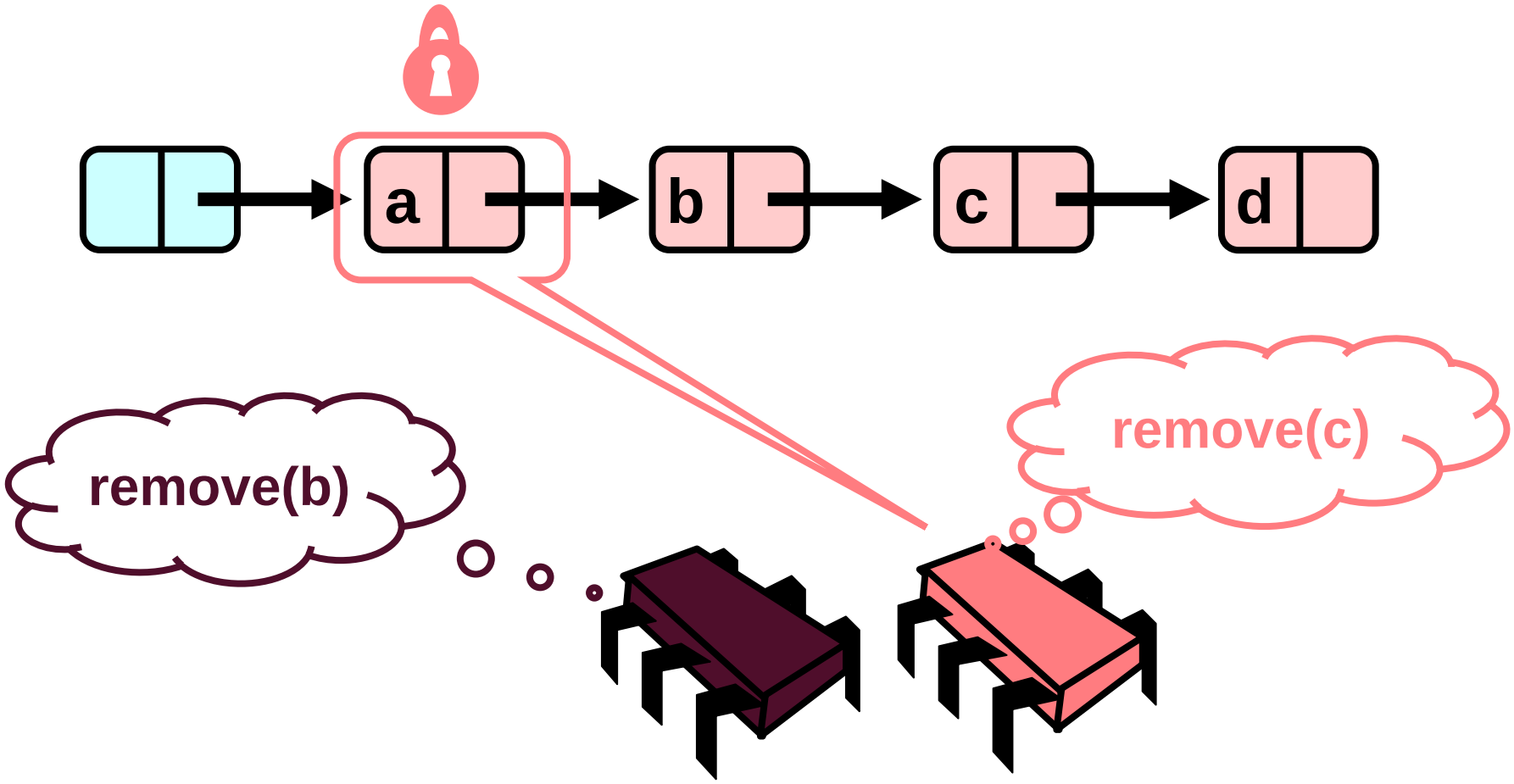
Concurrent Removes



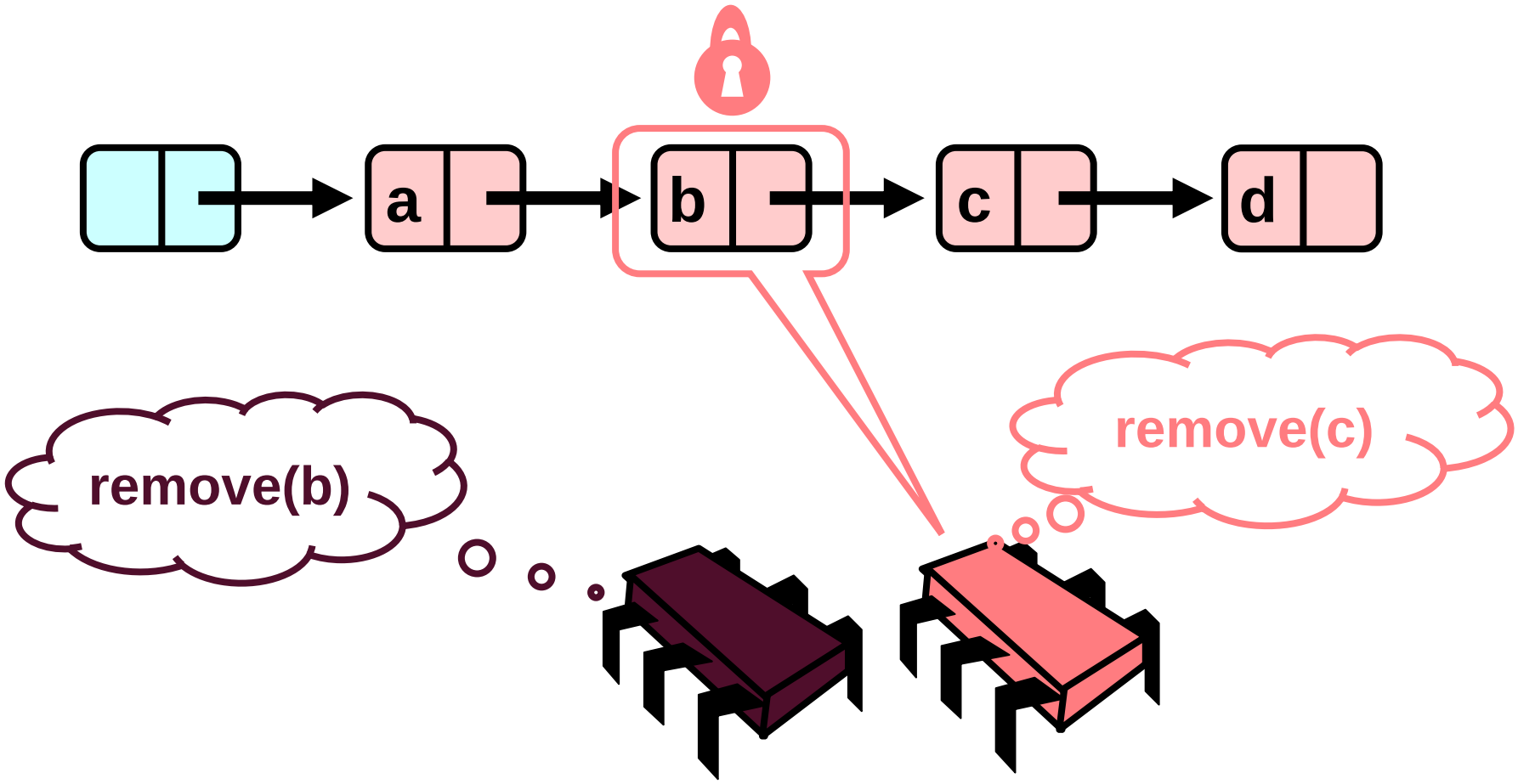
Concurrent Removes



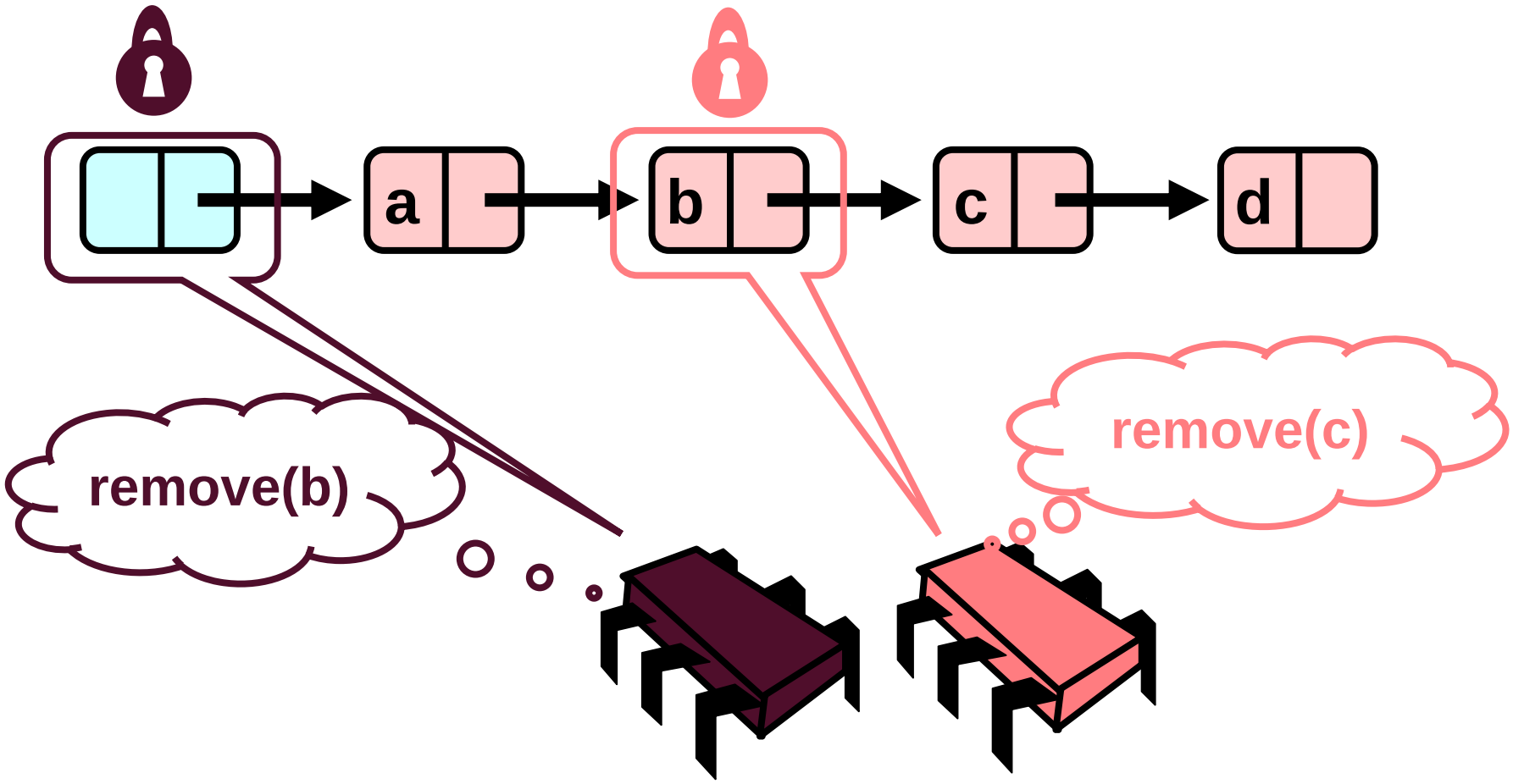
Concurrent Removes



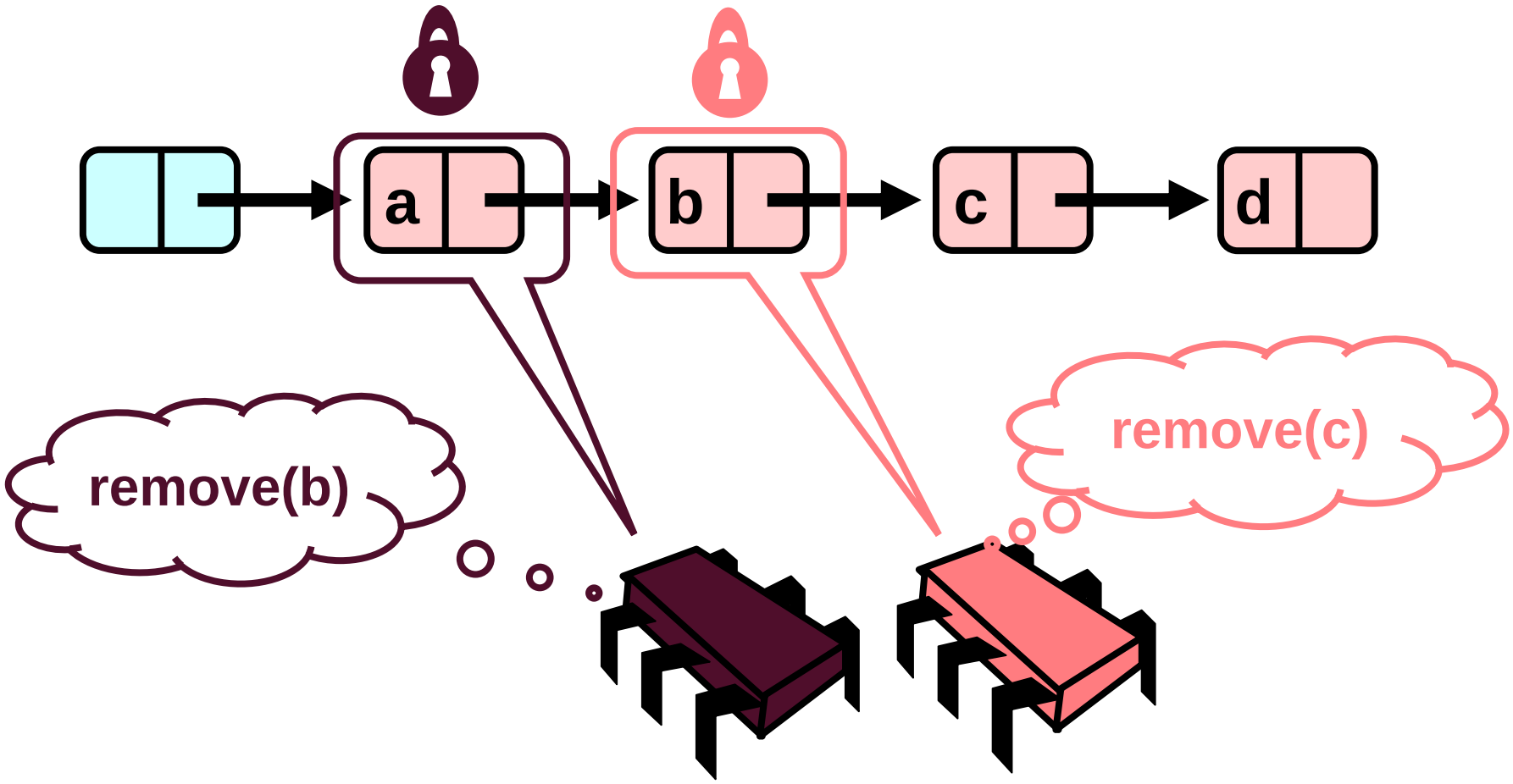
Concurrent Removes



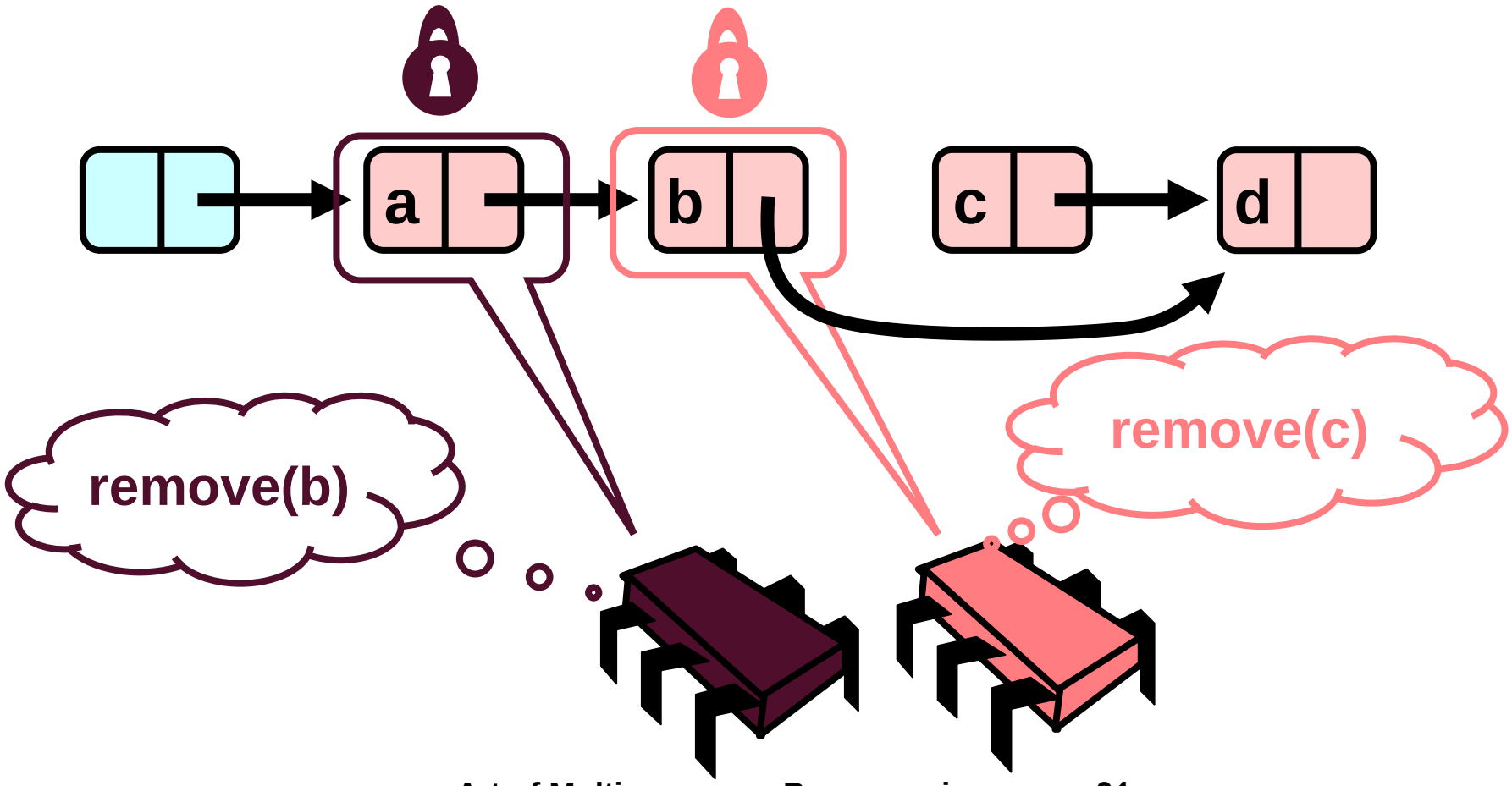
Concurrent Removes



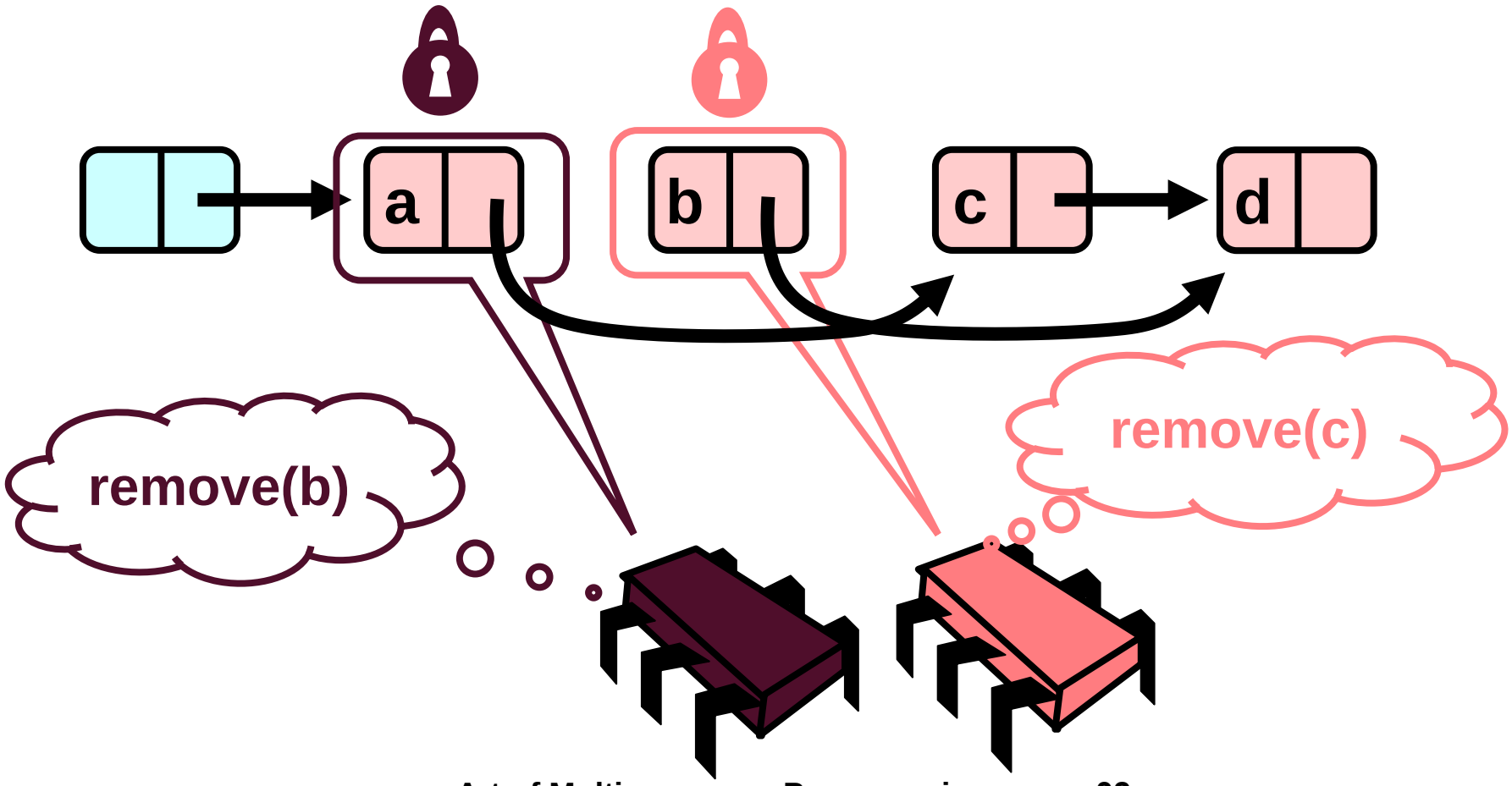
Concurrent Removes



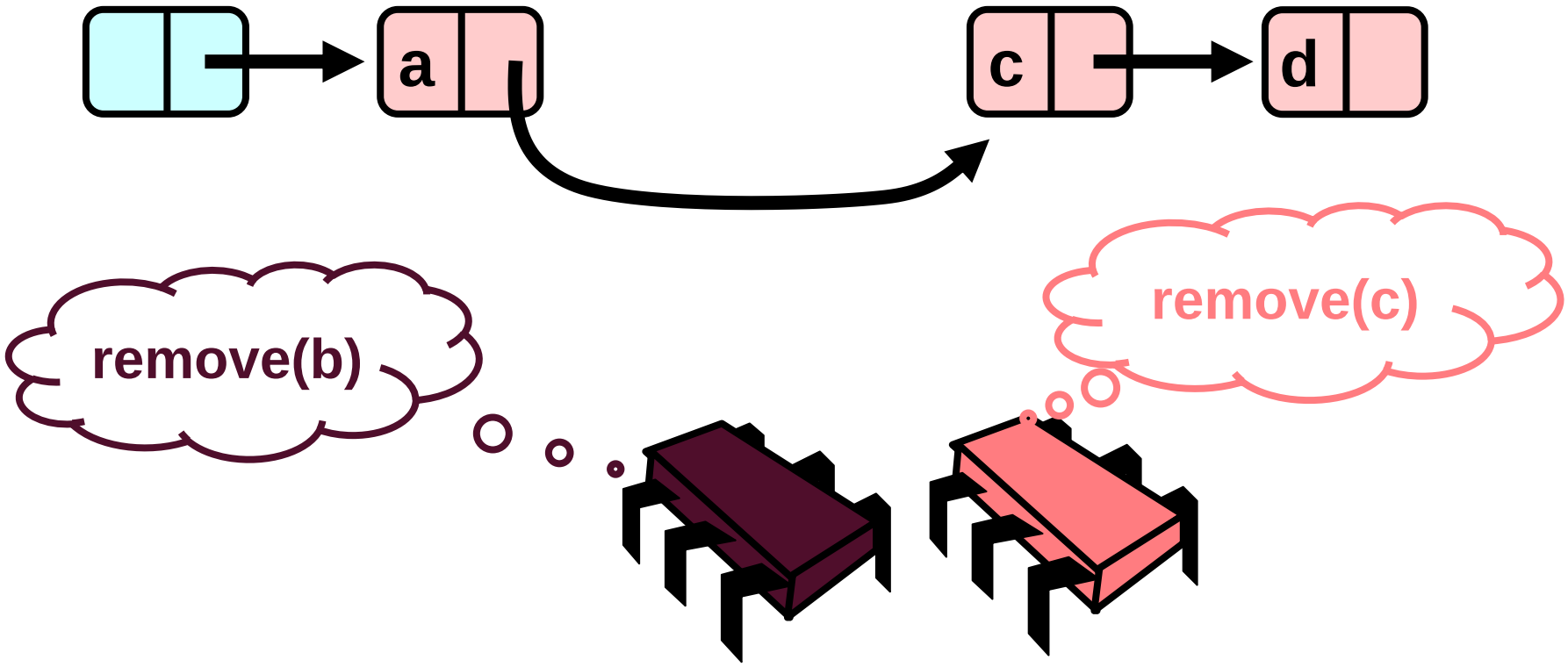
Concurrent Removes



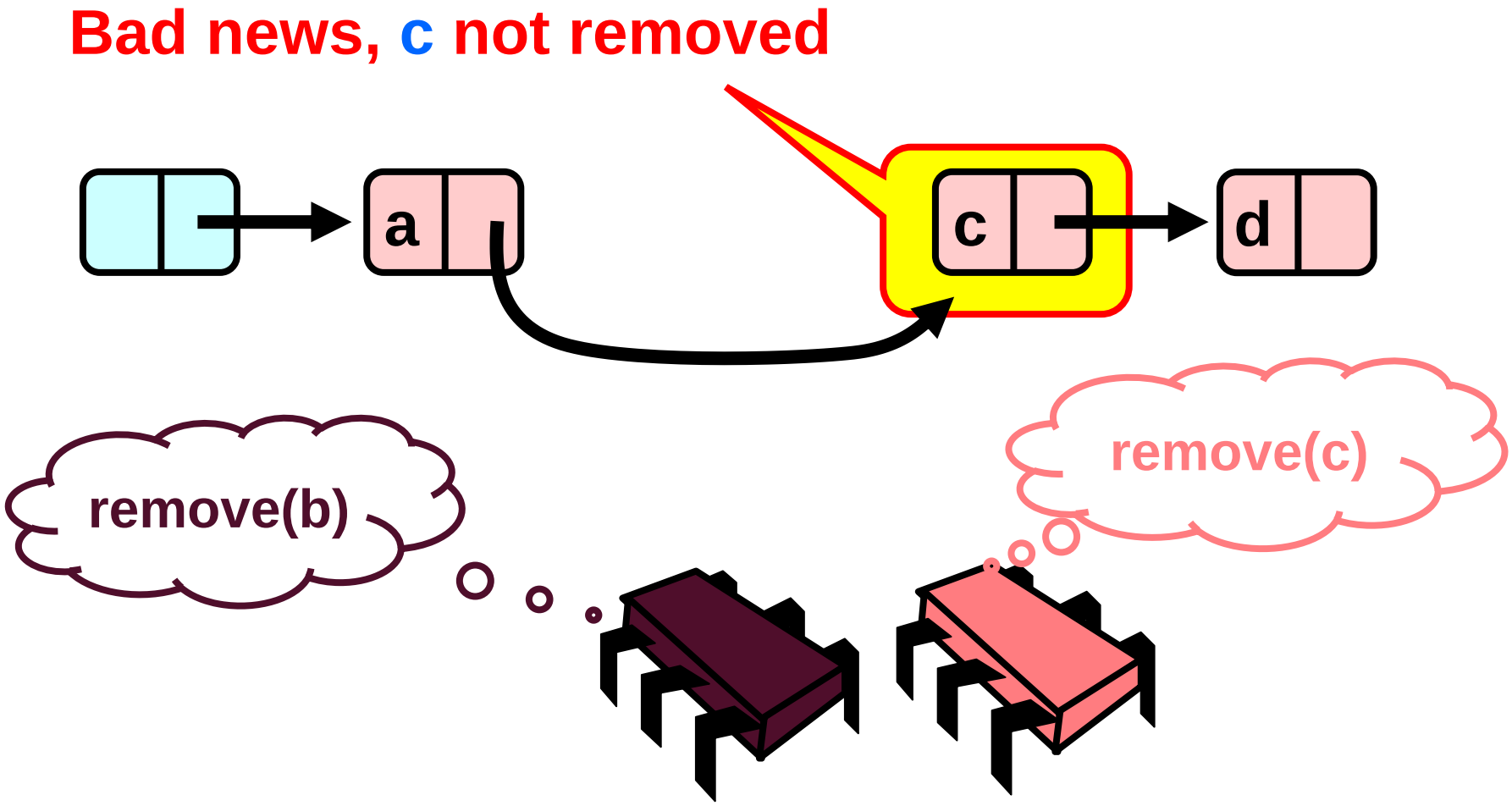
Concurrent Removes



Uh, Oh



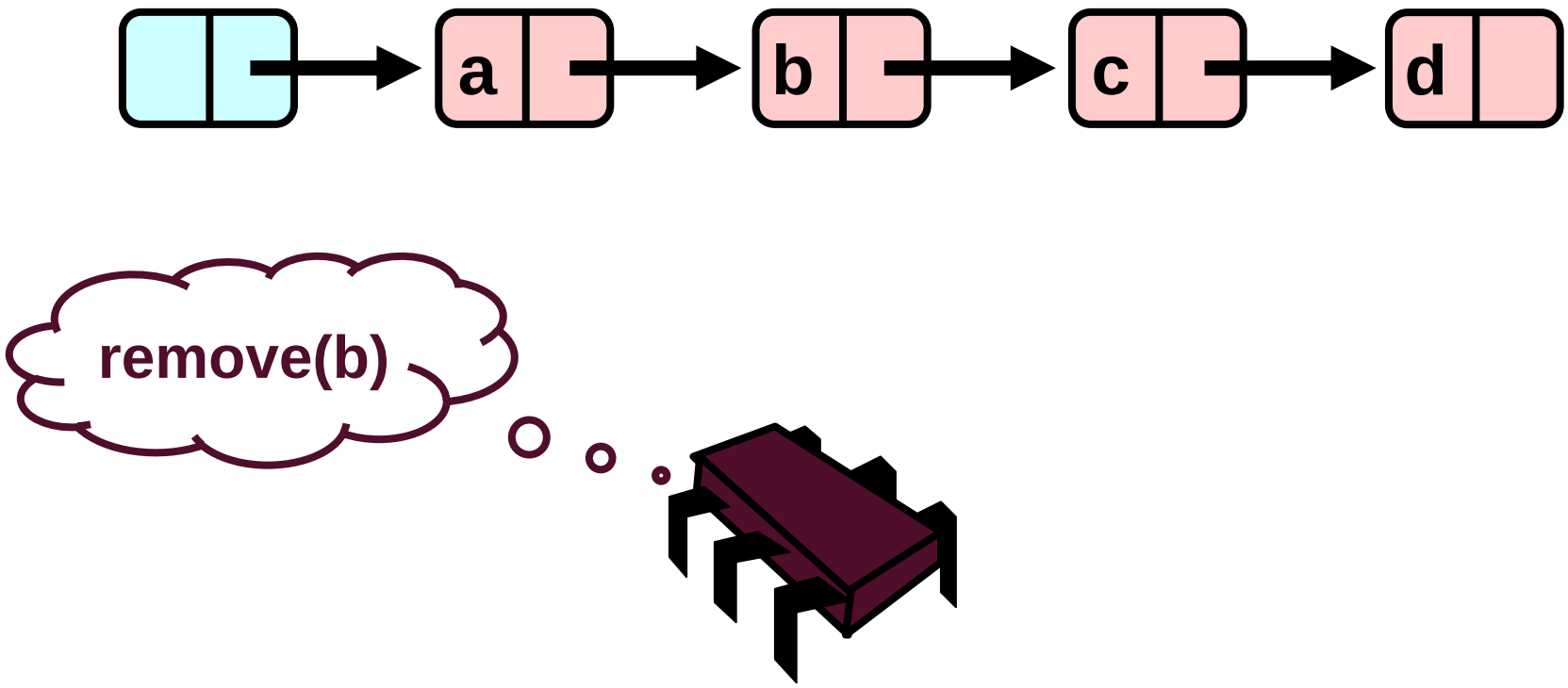
Uh, Oh



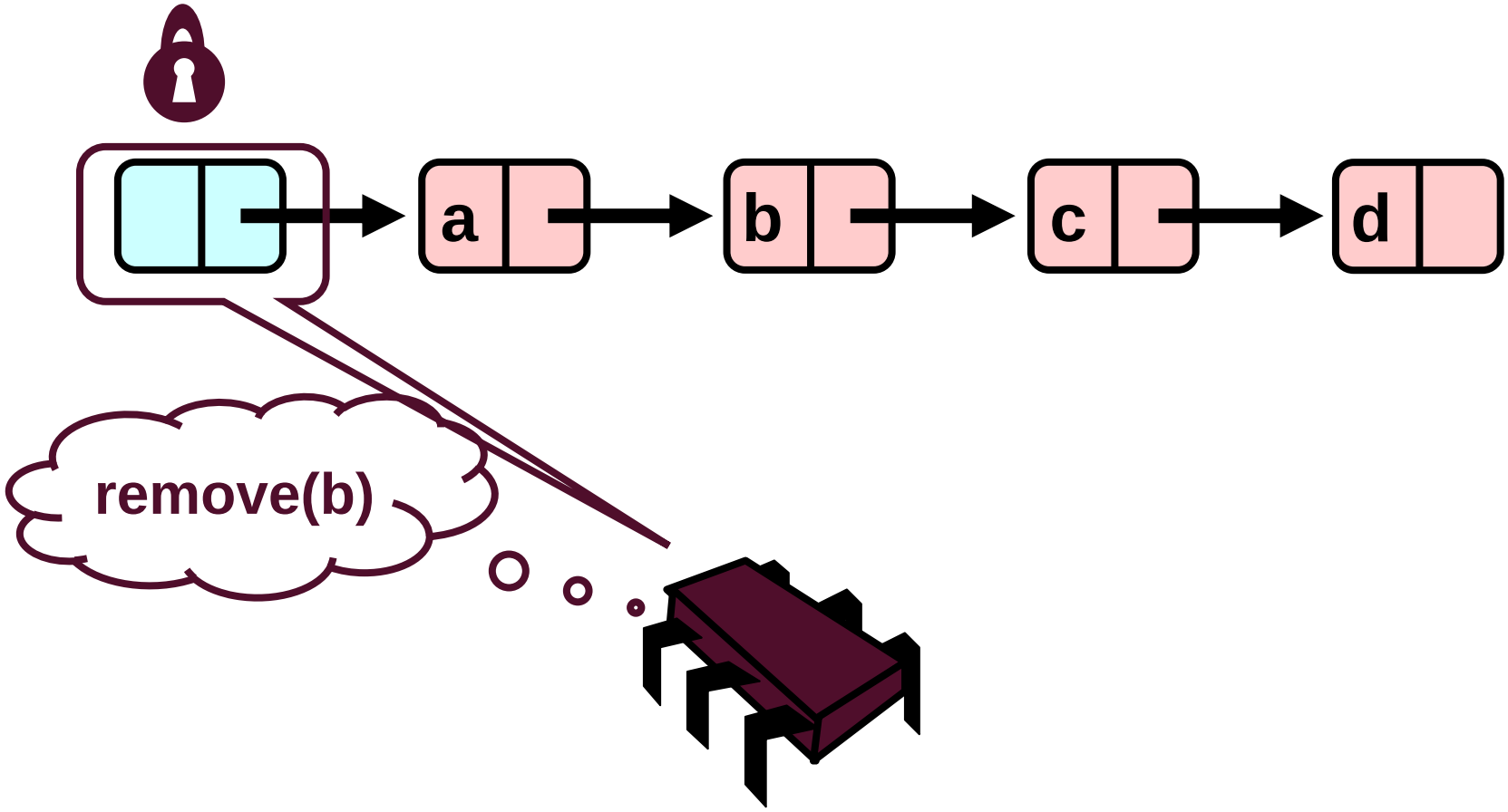
Insight

- **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

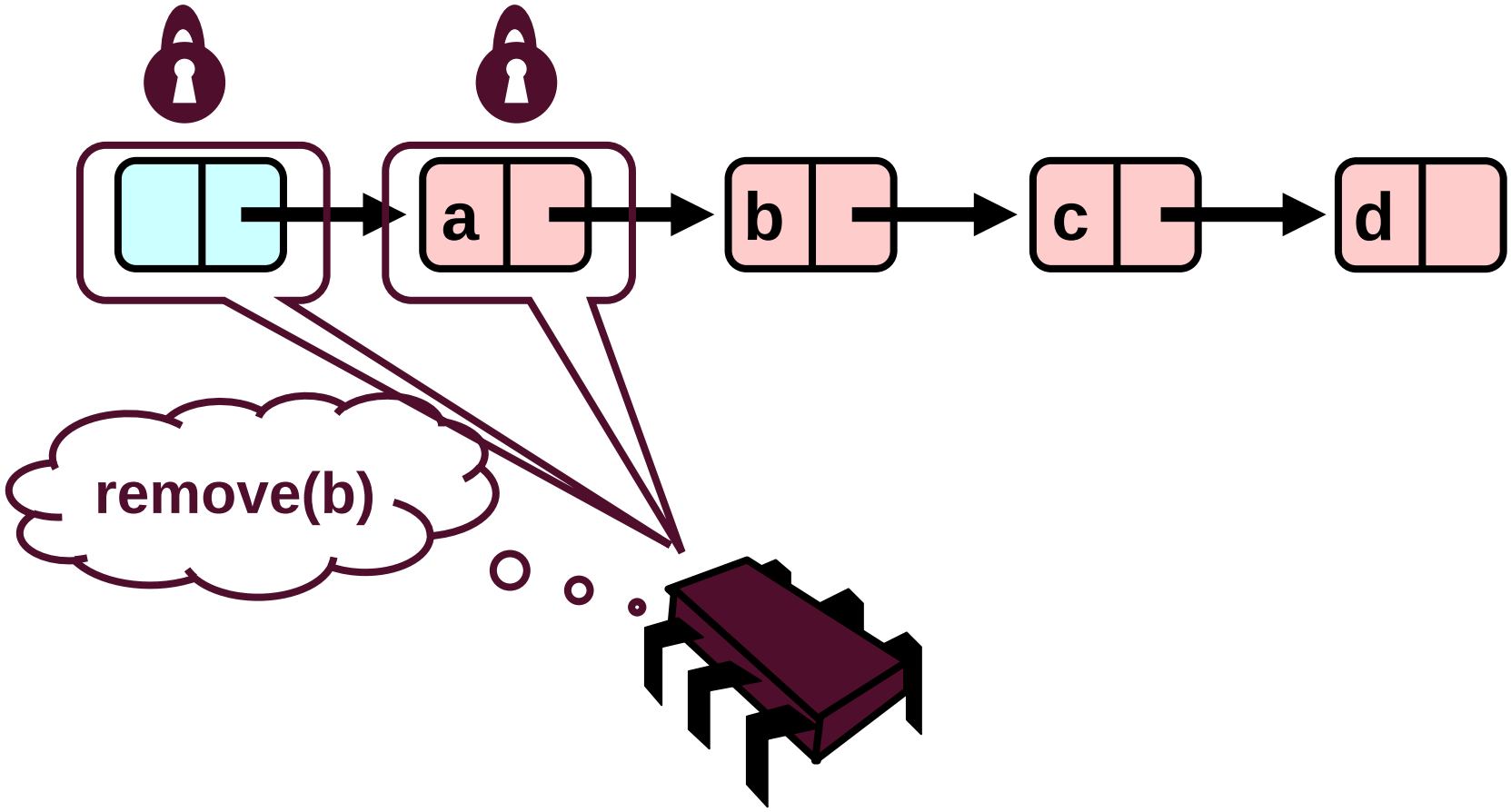
Hand-Over-Hand Again



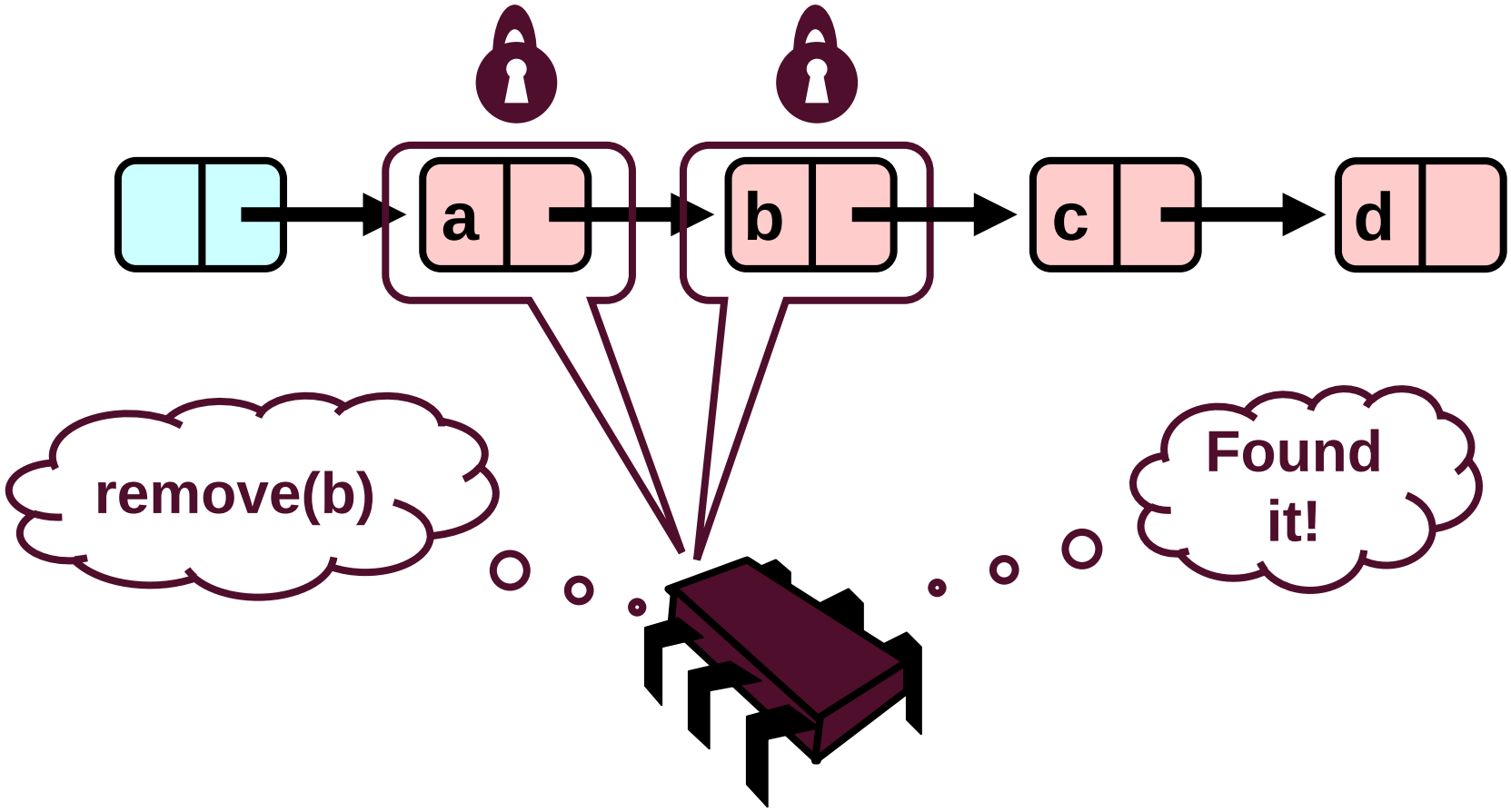
Hand-Over-Hand Again



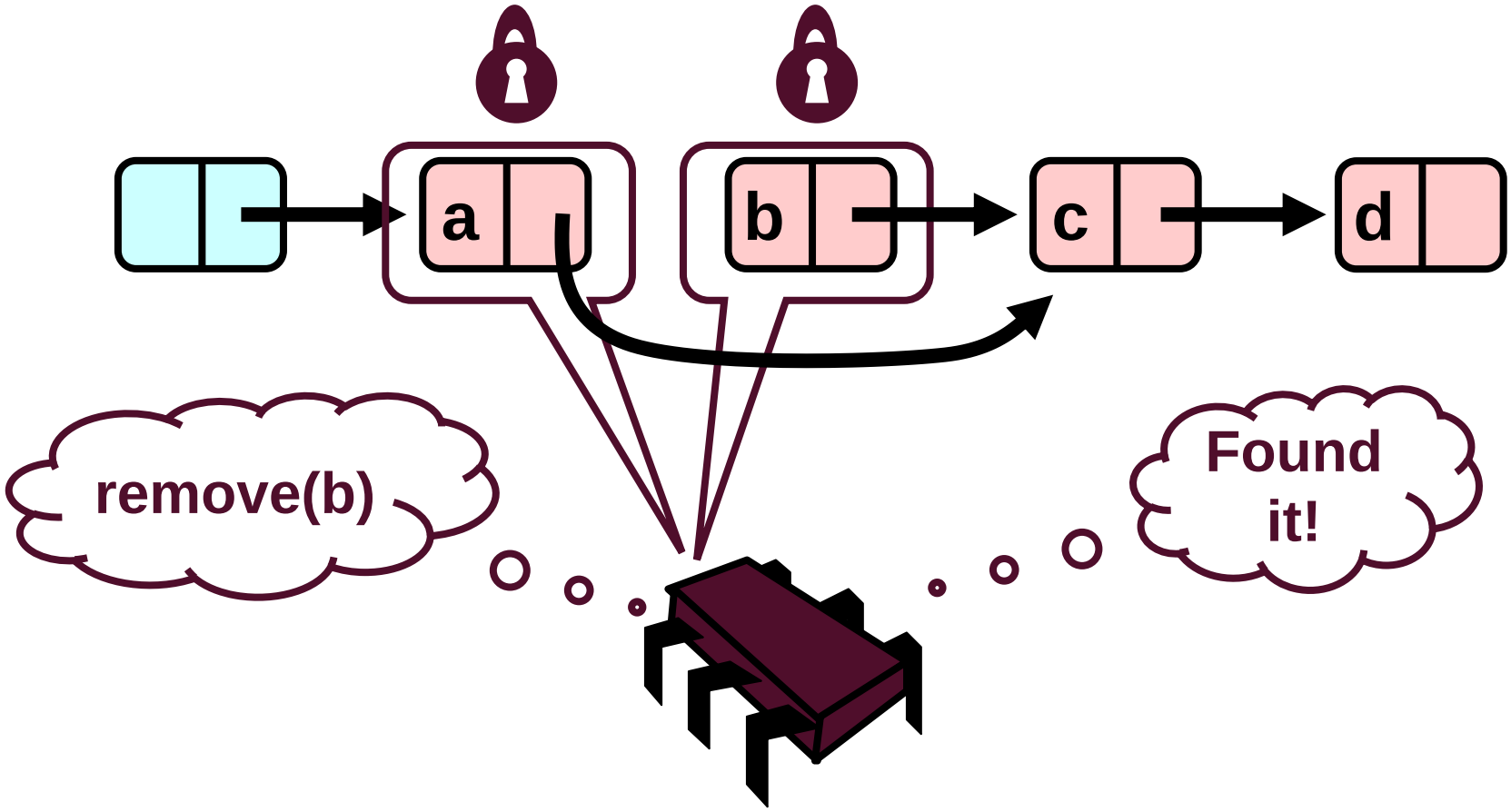
Hand-Over-Hand Again



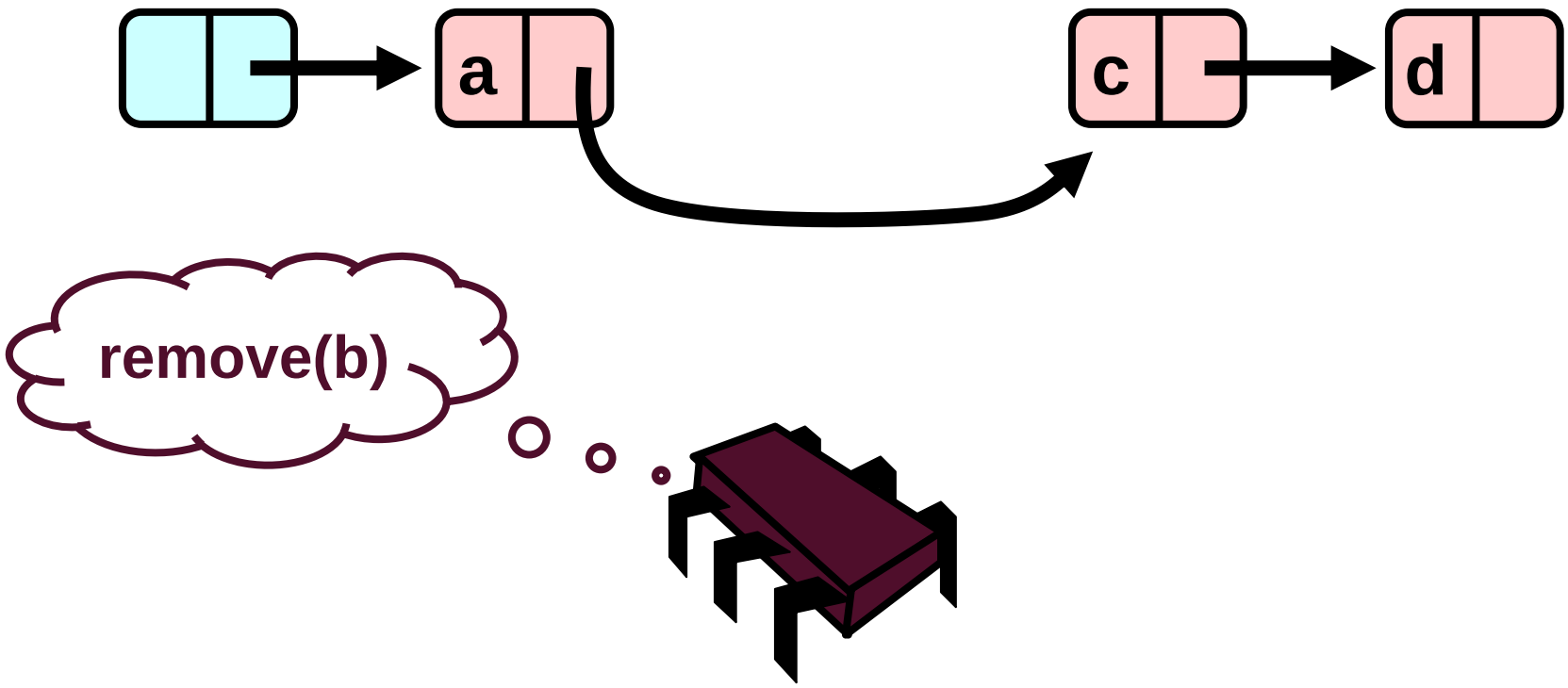
Hand-Over-Hand Again



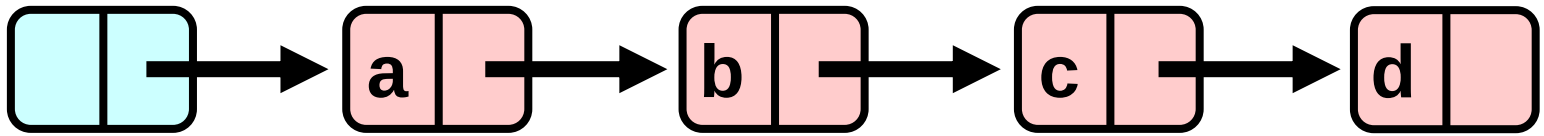
Hand-Over-Hand Again

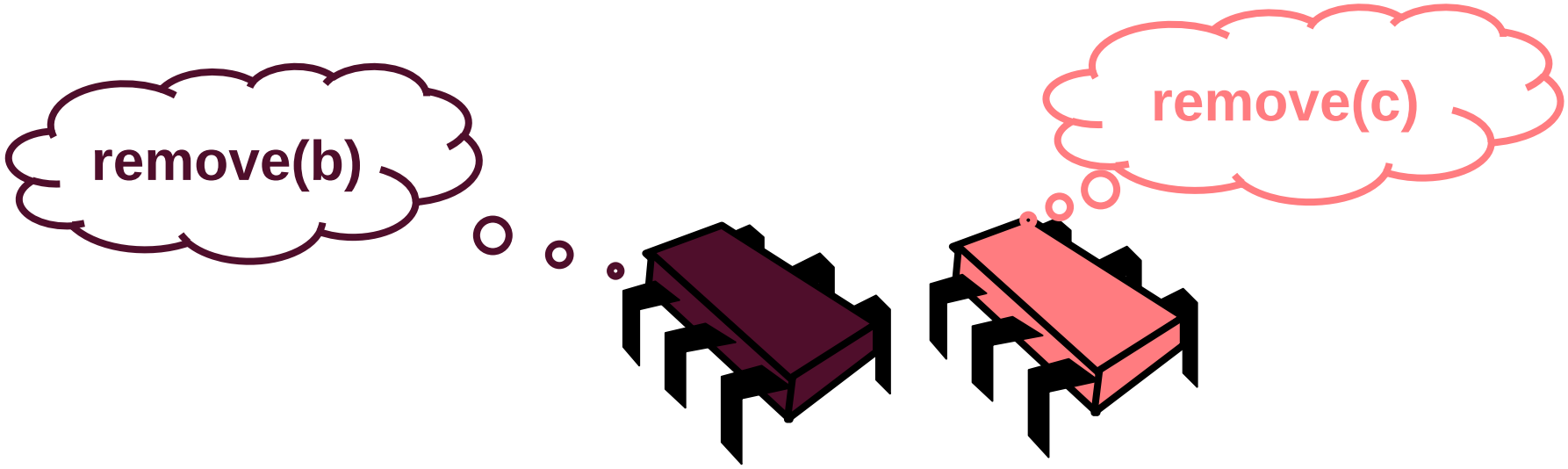


Hand-Over-Hand Again

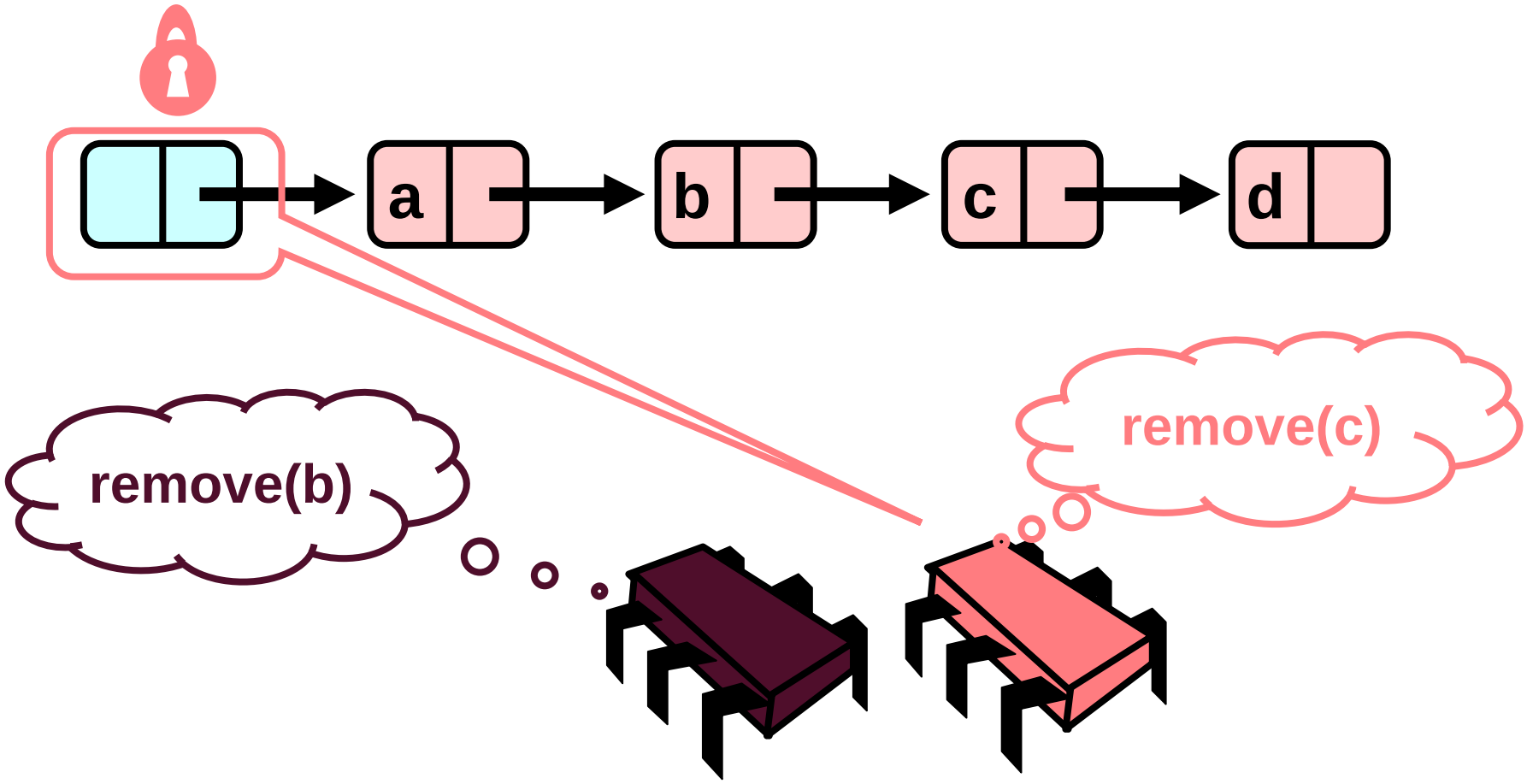


Removing a Node

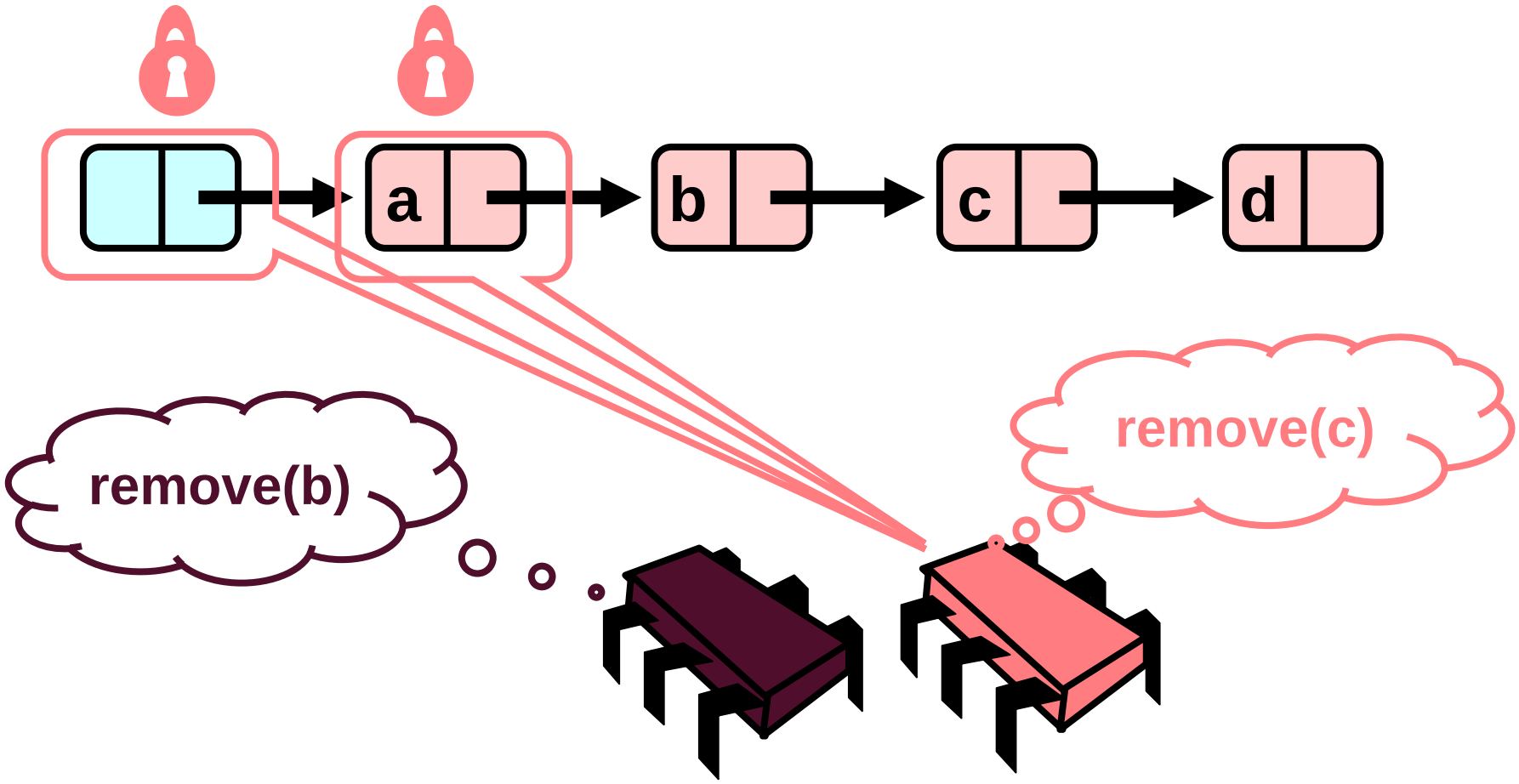




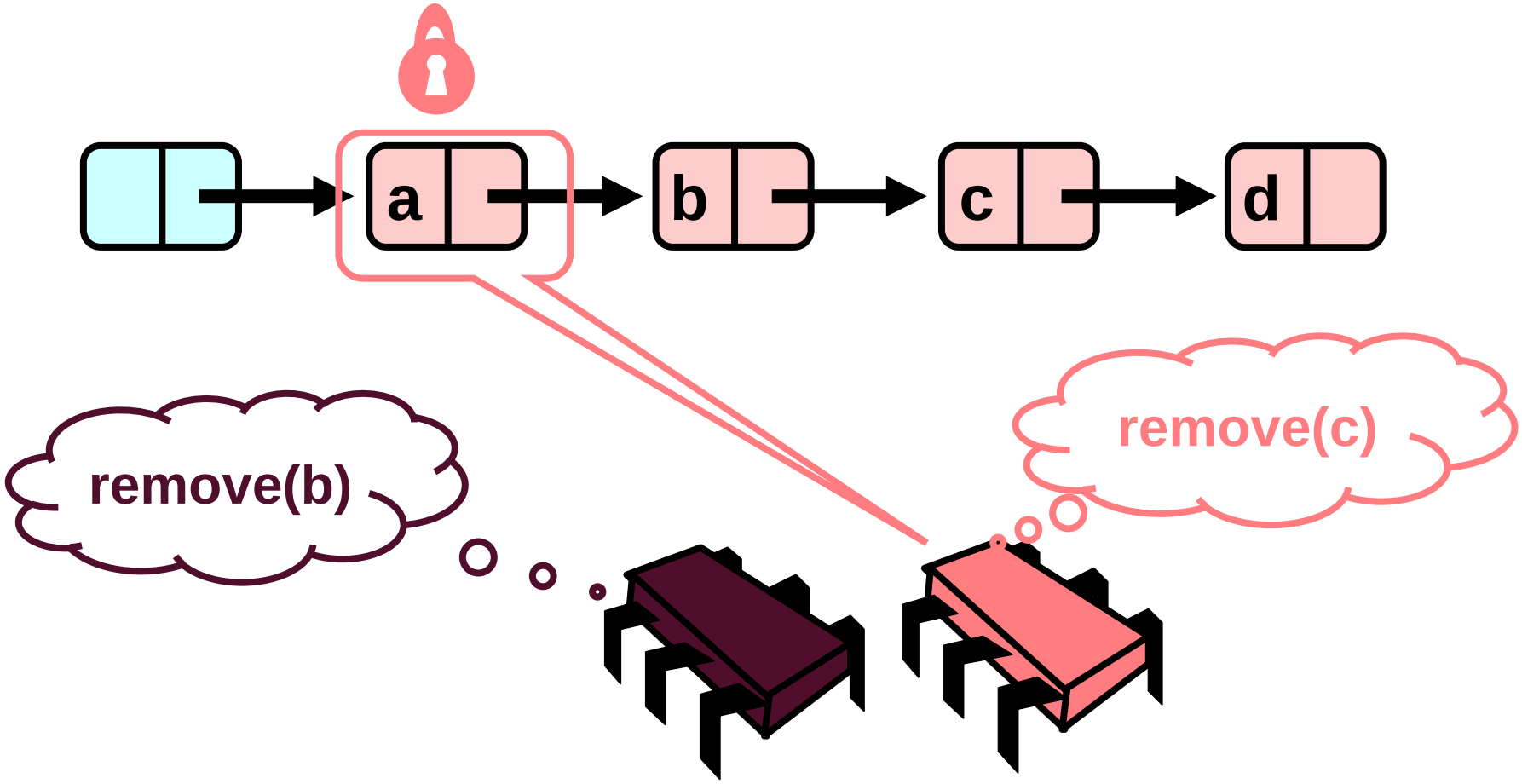
Removing a Node



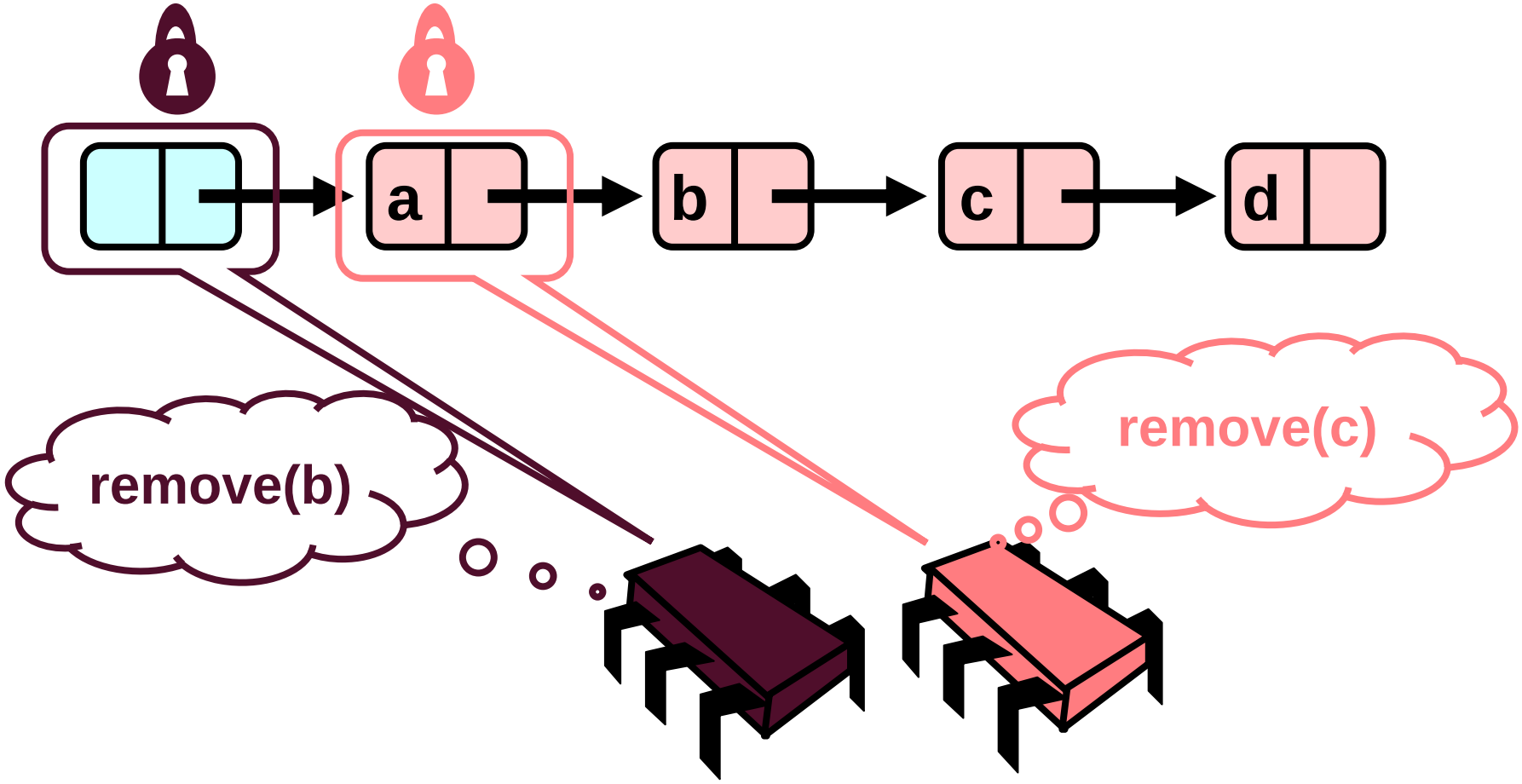
Removing a Node



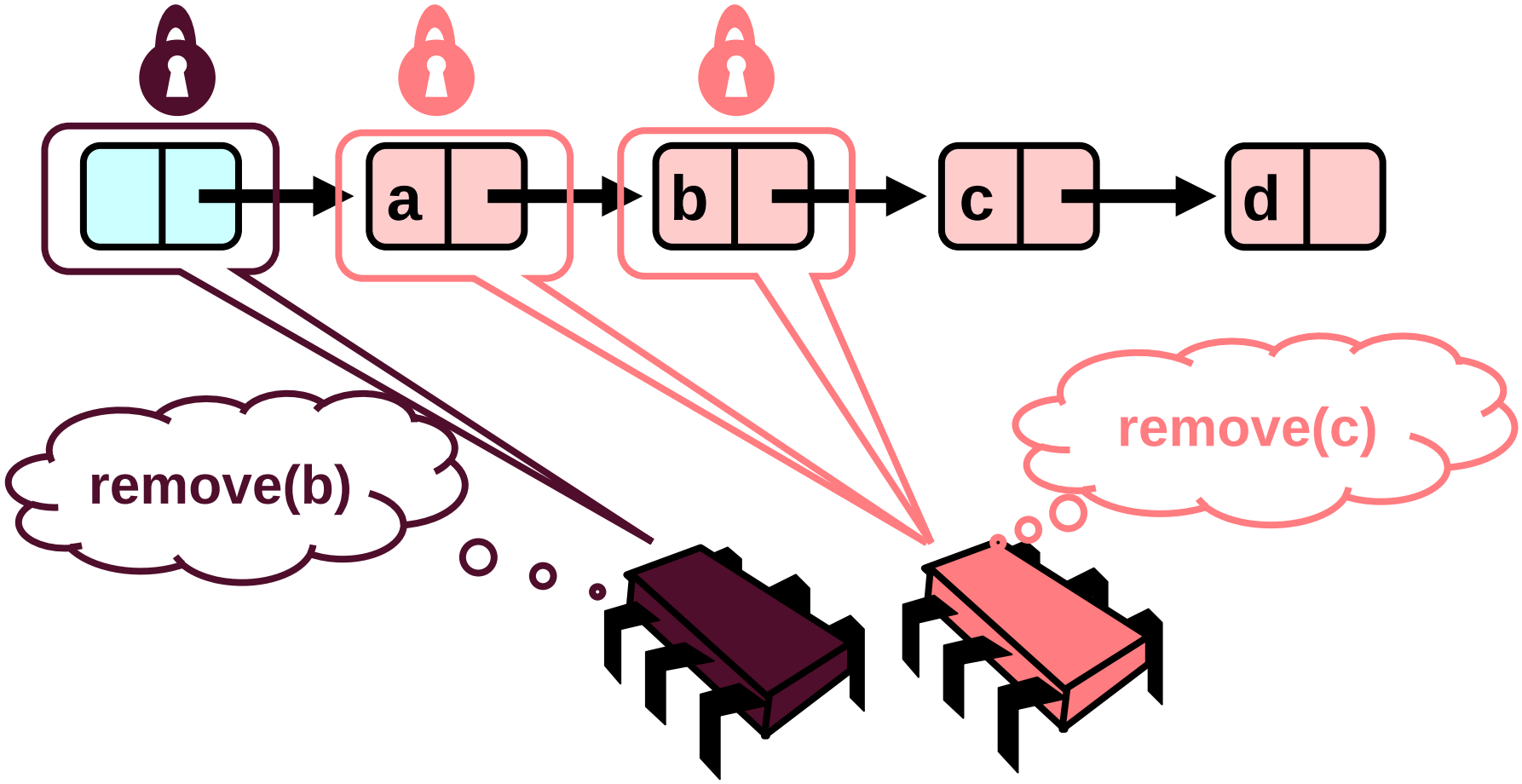
Removing a Node



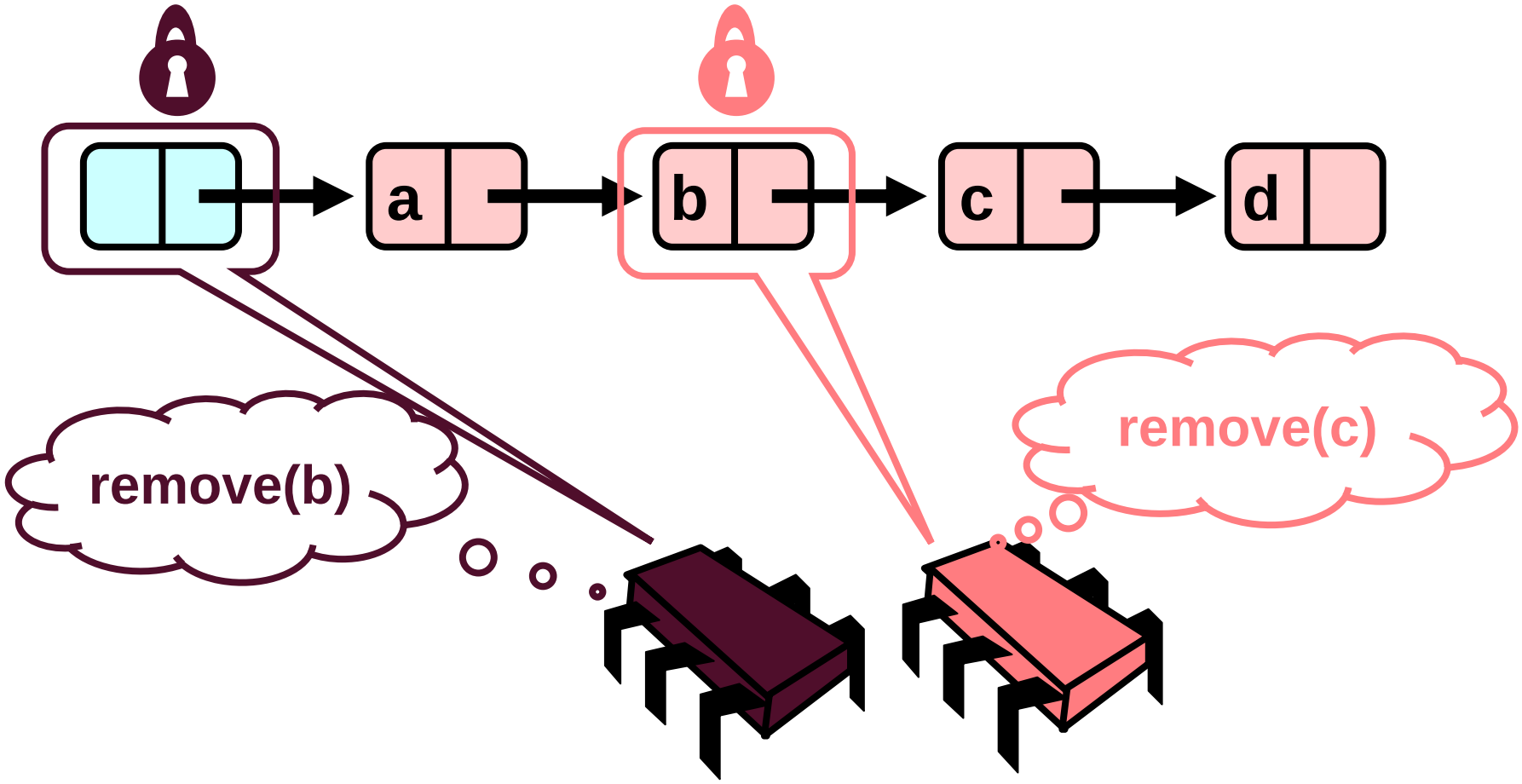
Removing a Node



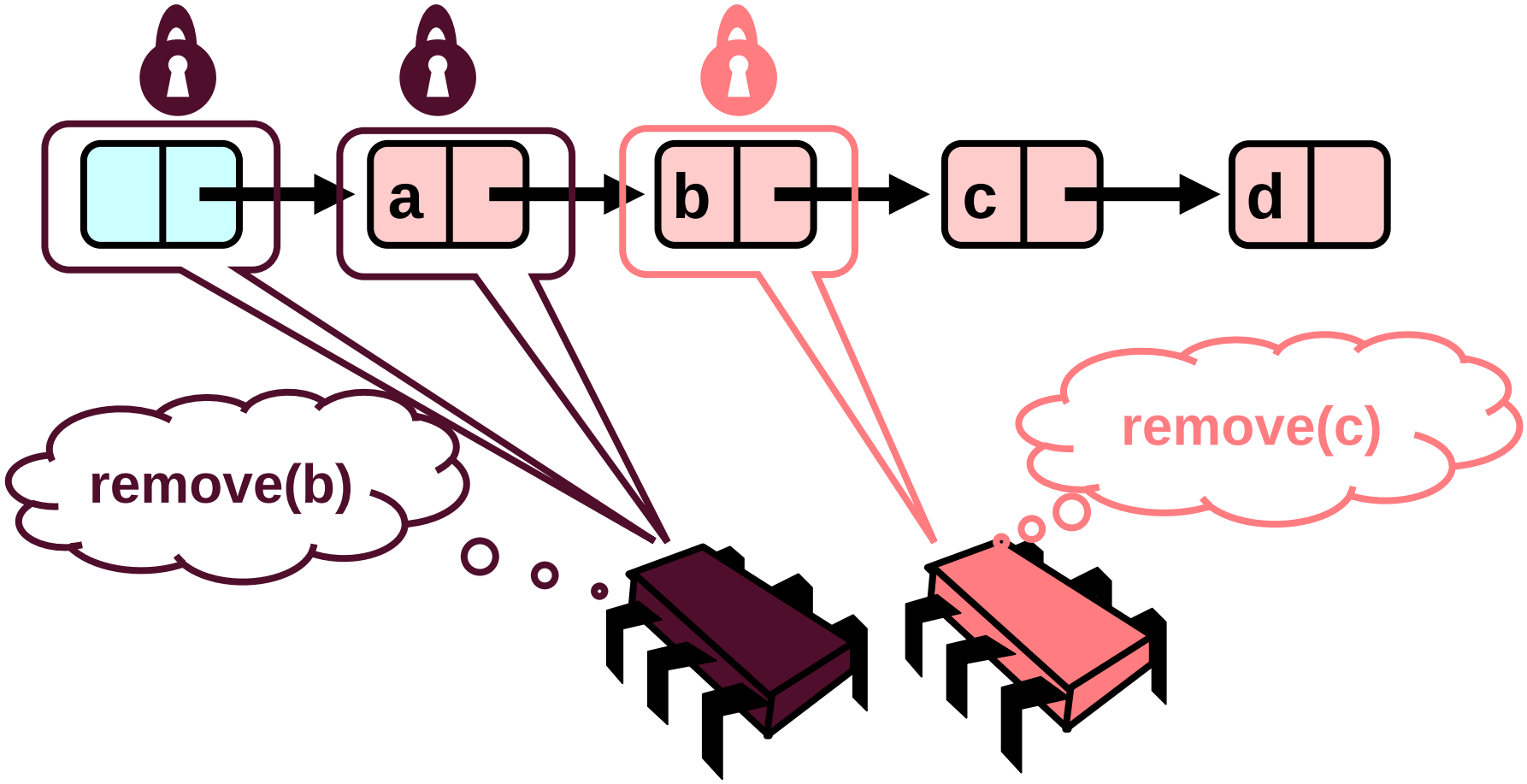
Removing a Node



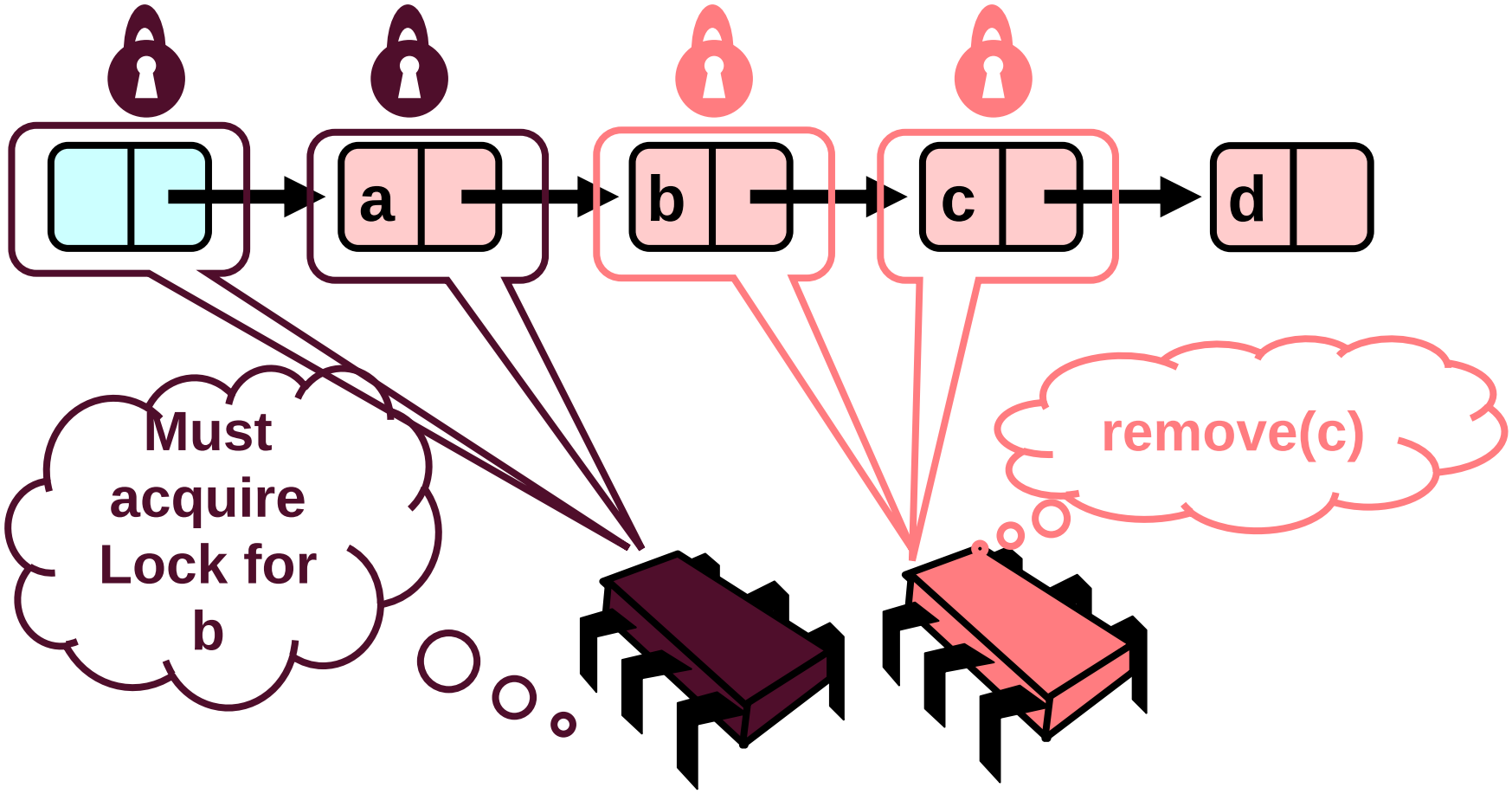
Removing a Node



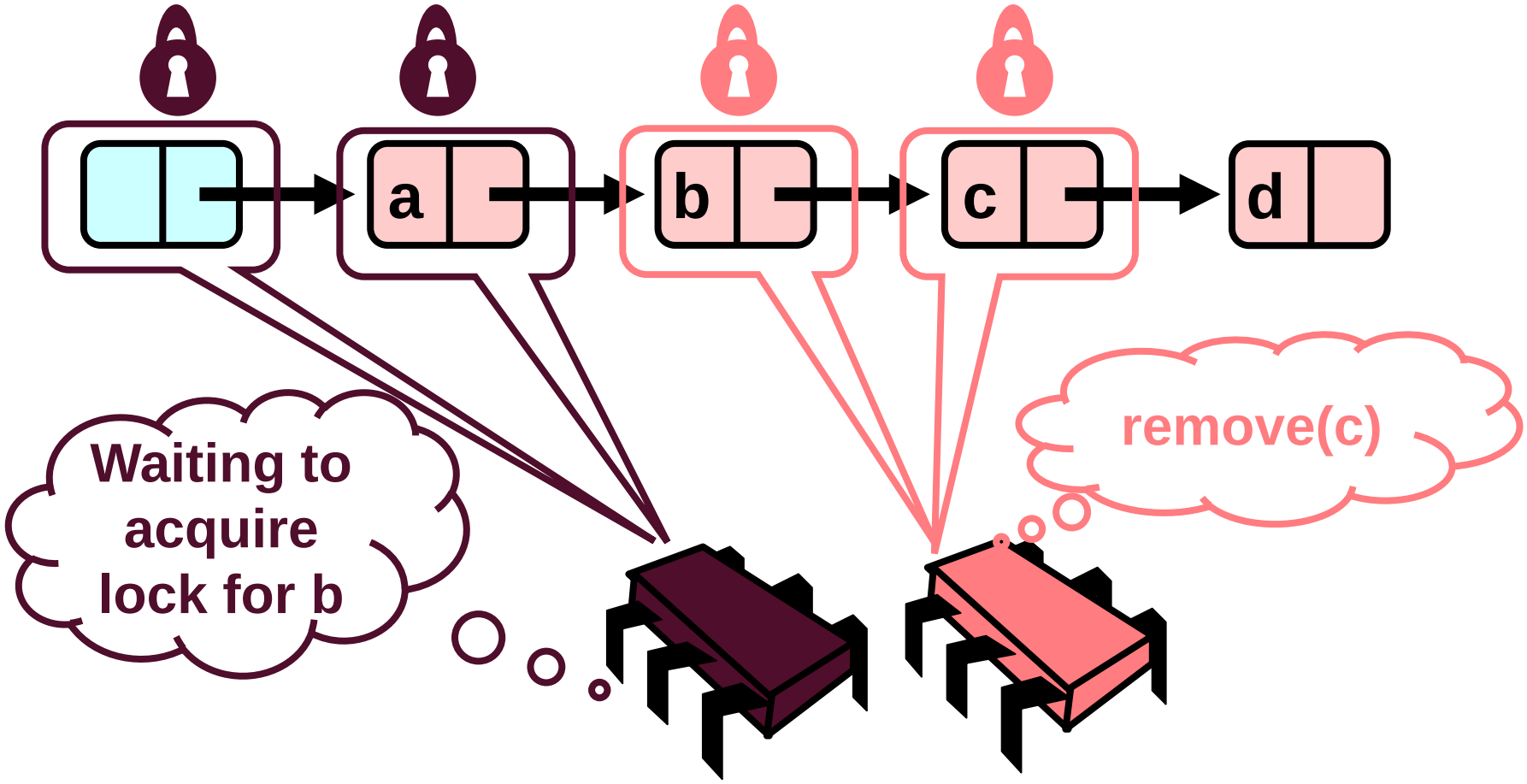
Removing a Node



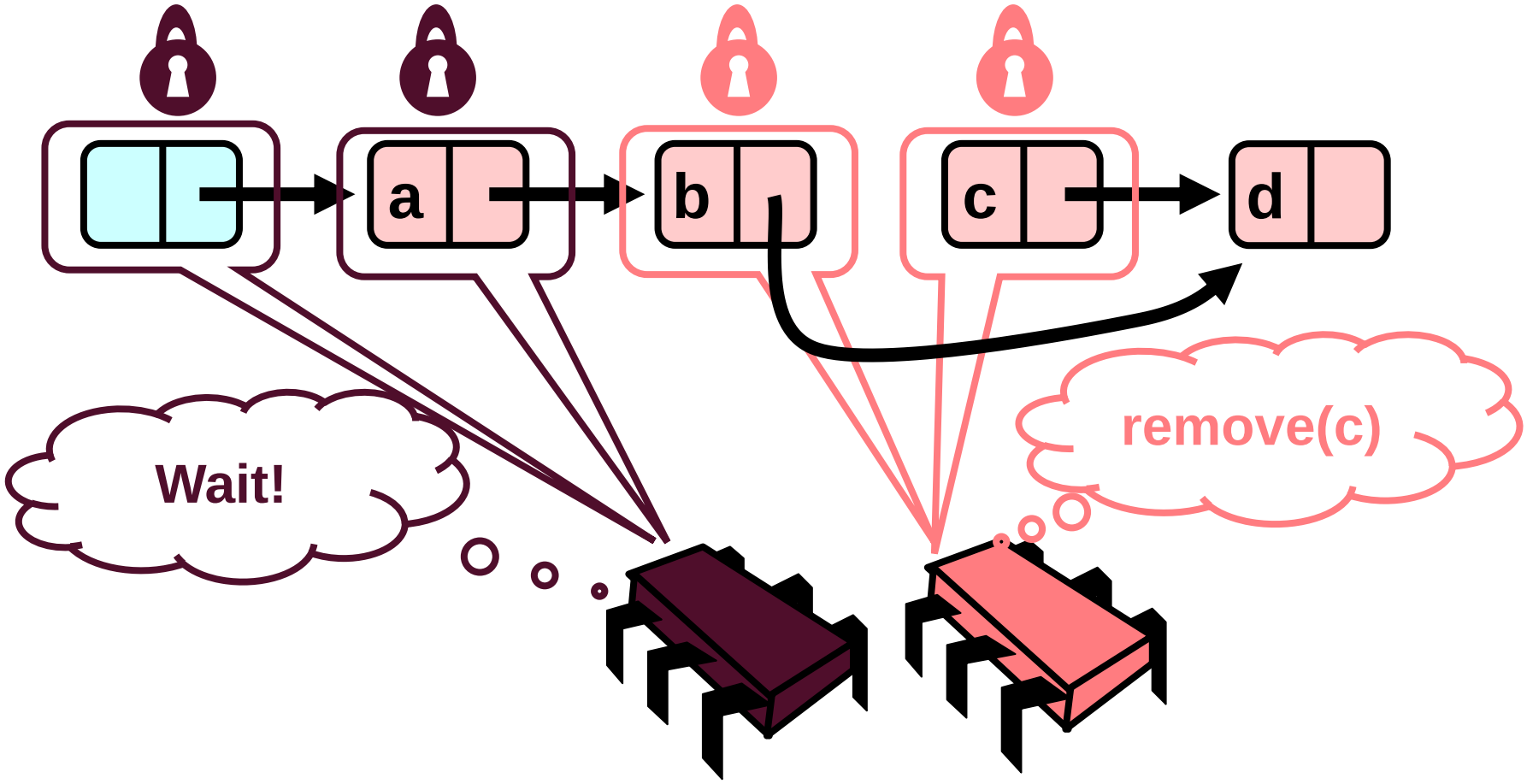
Removing a Node



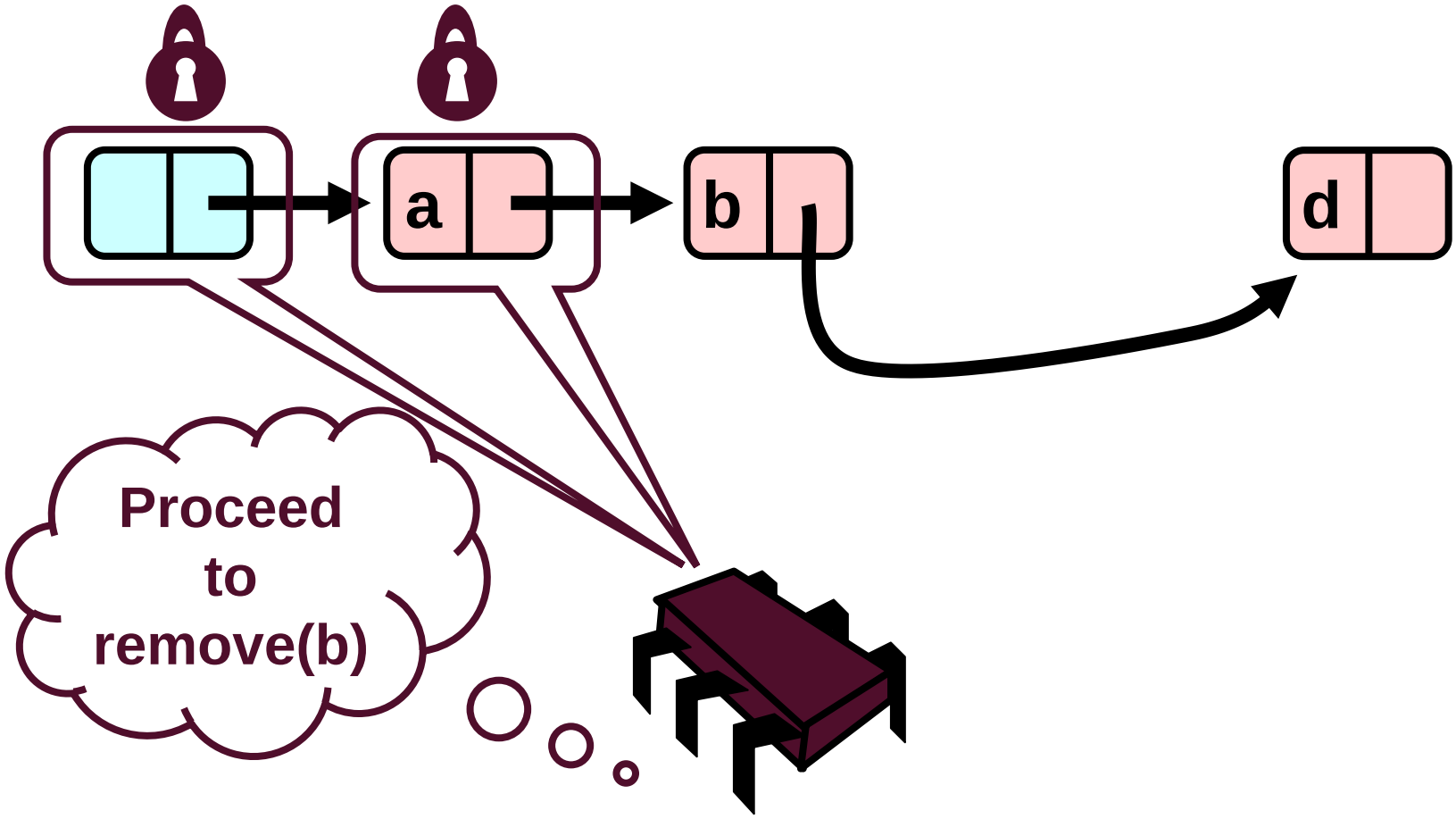
Removing a Node



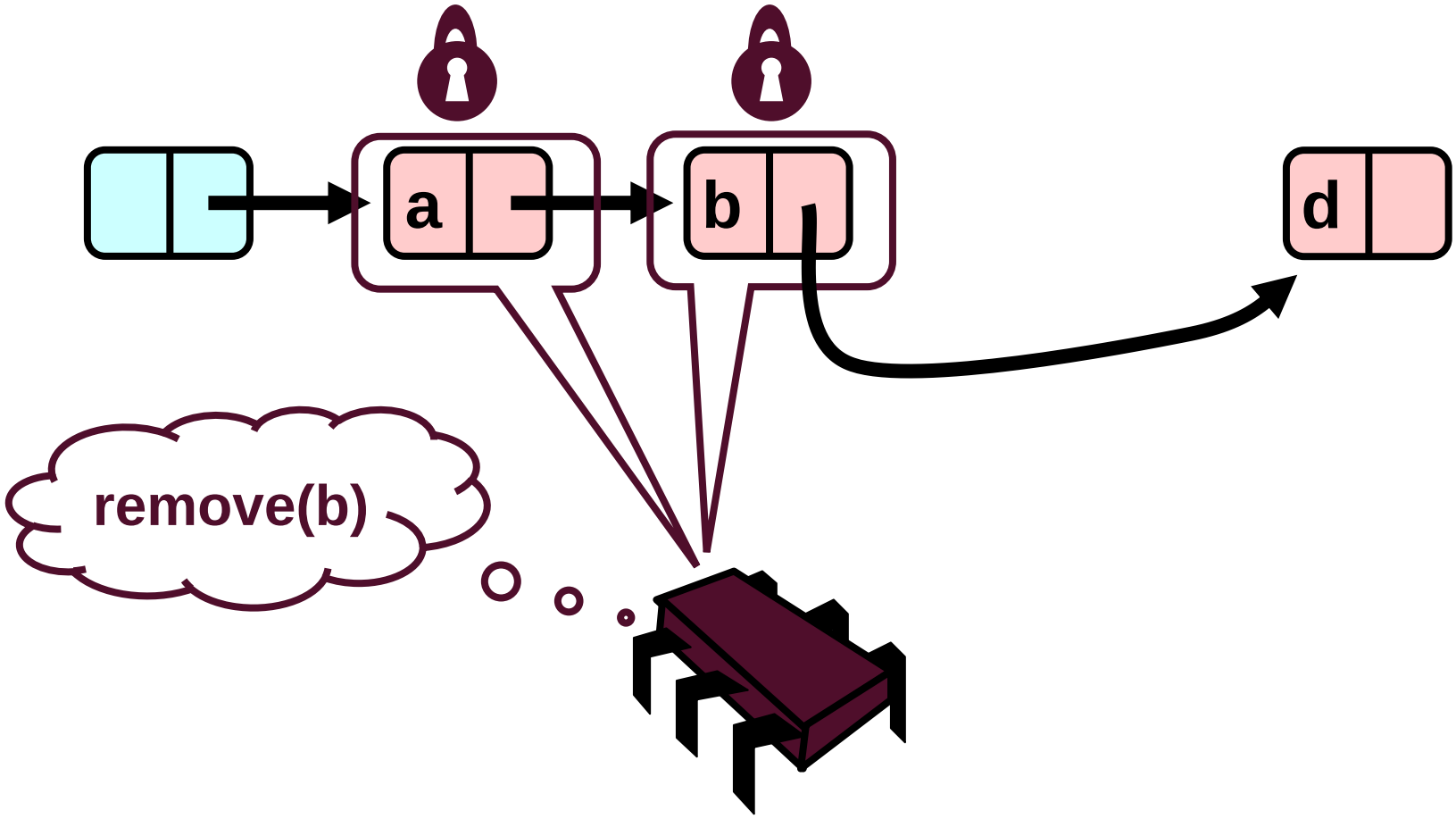
Removing a Node



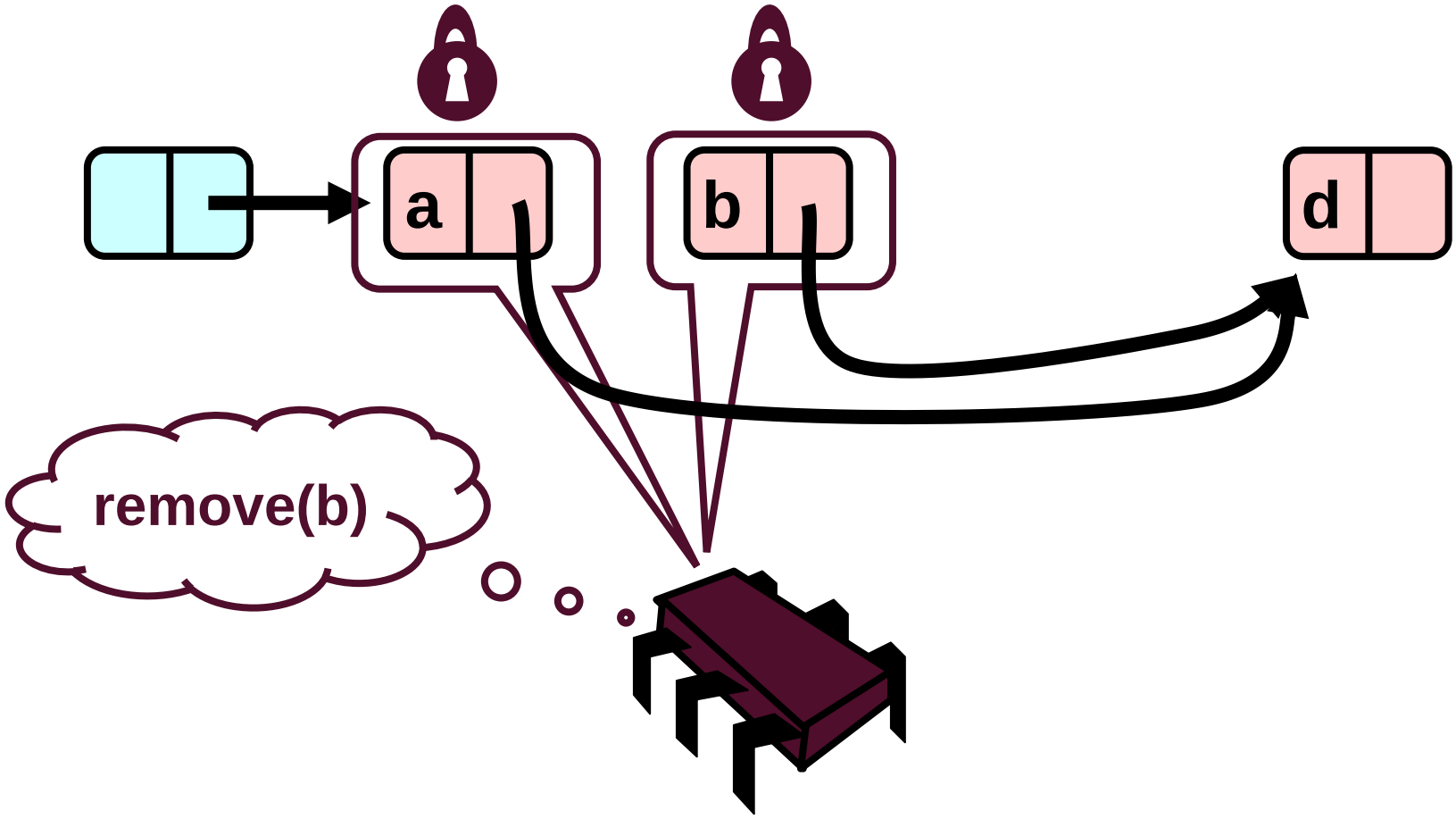
Removing a Node



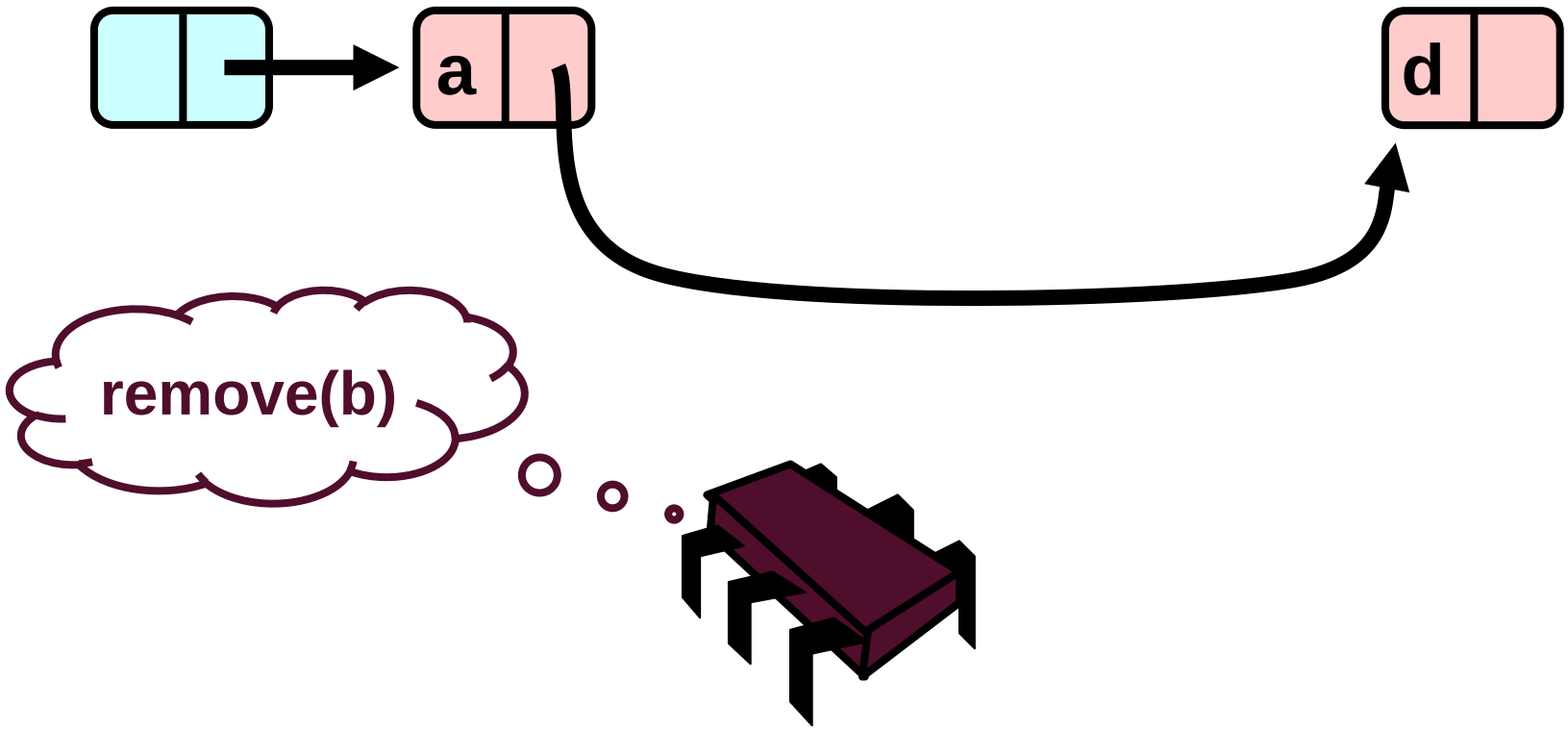
Removing a Node



Removing a Node



Removing a Node



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

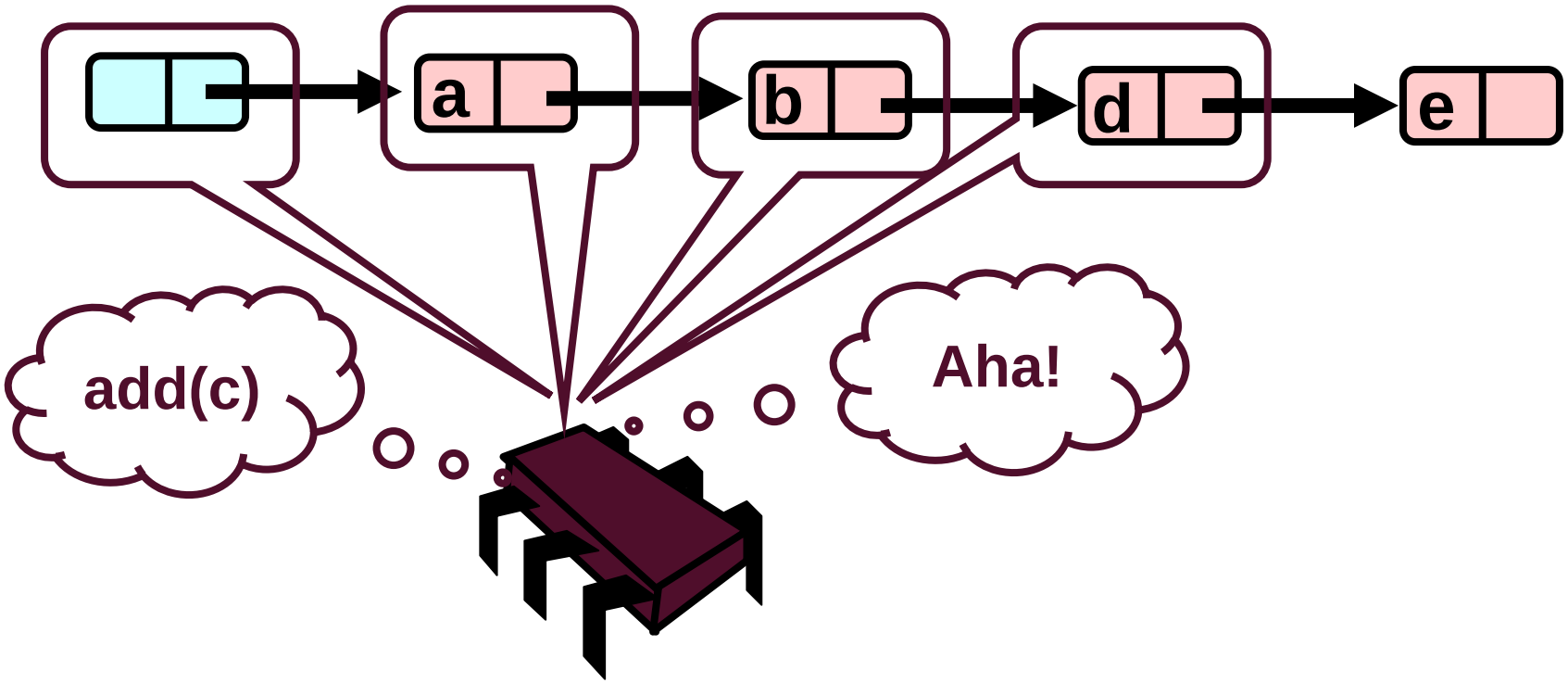
Trick 2: Reader/Writer Locking

- **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

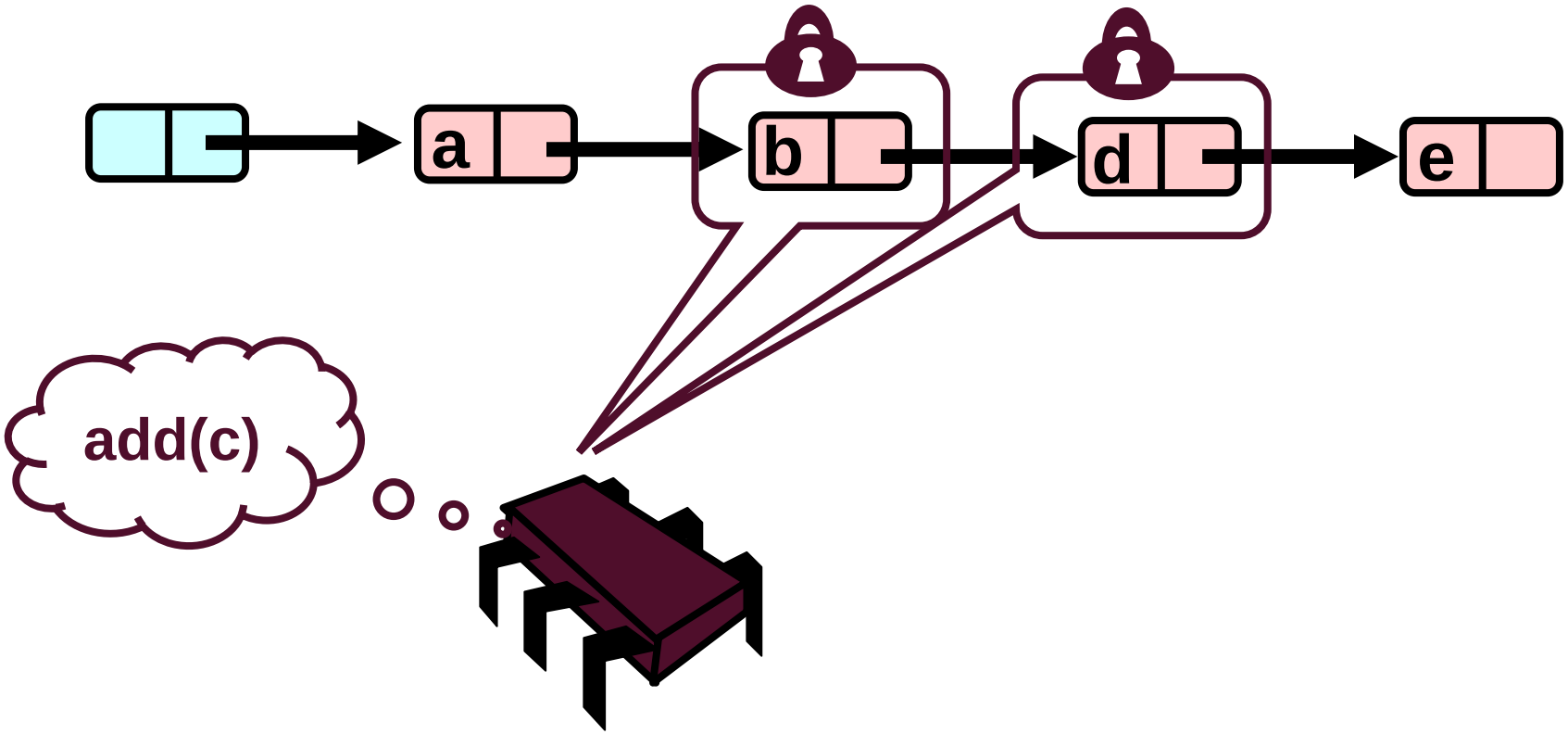
Trick 3: Optimistic synchronization

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

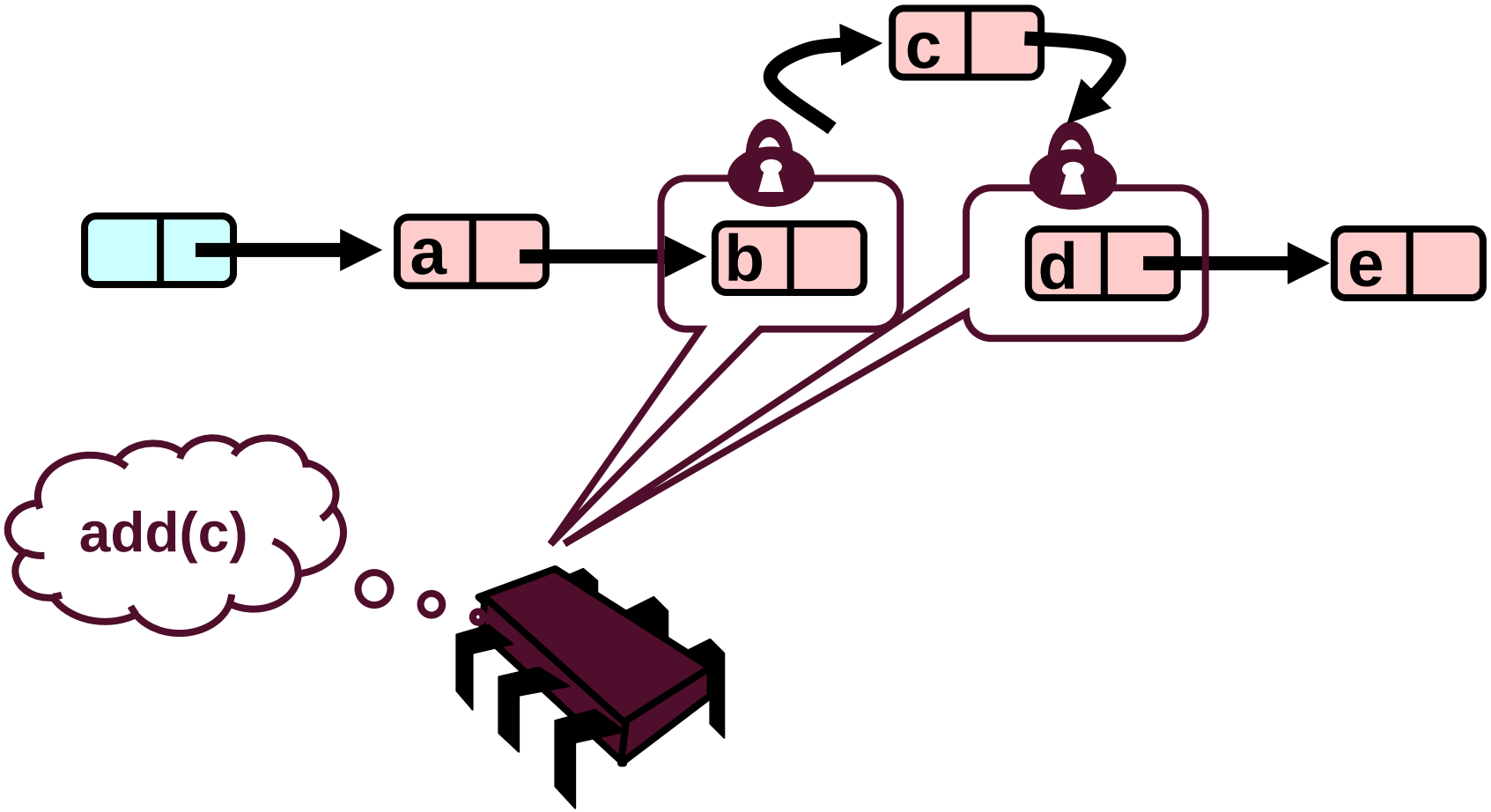
Optimistic: Traverse without Locking



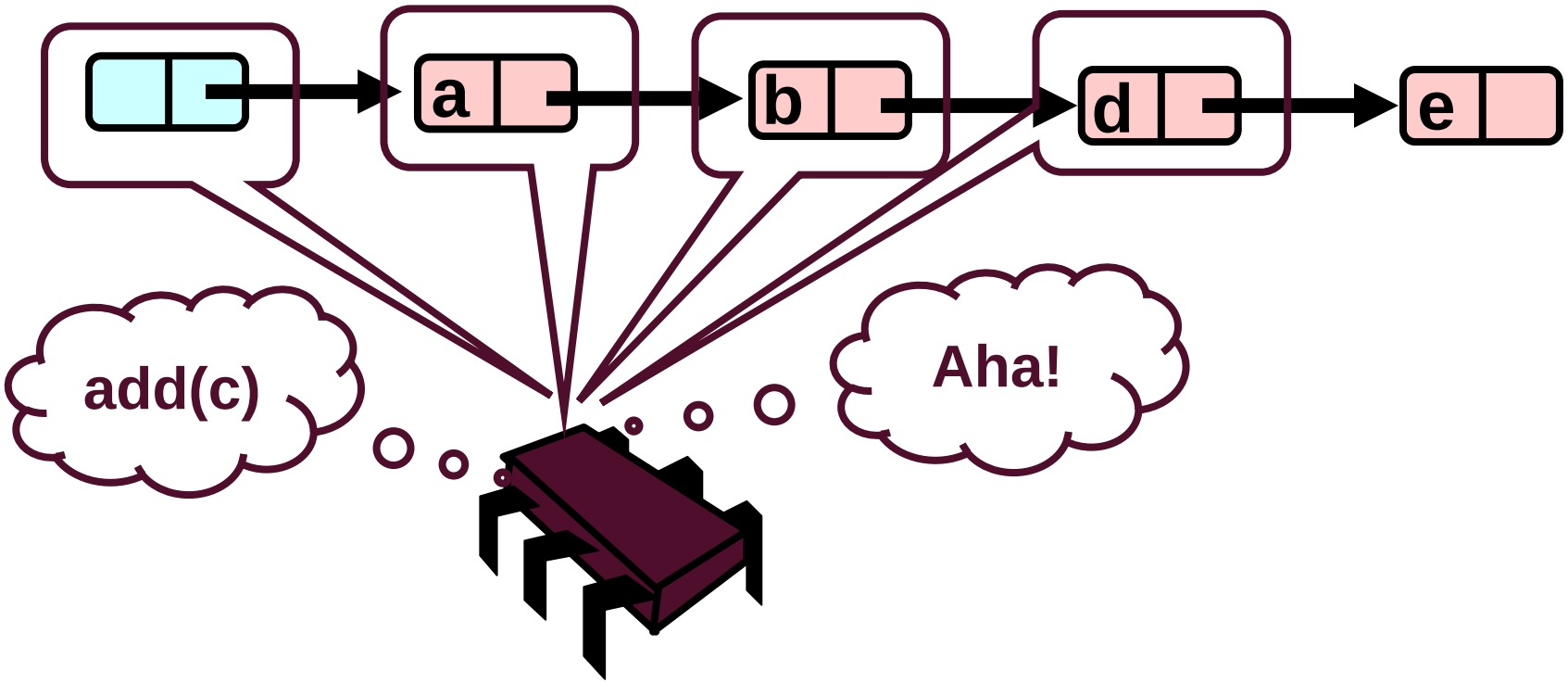
Optimistic: Lock and Load



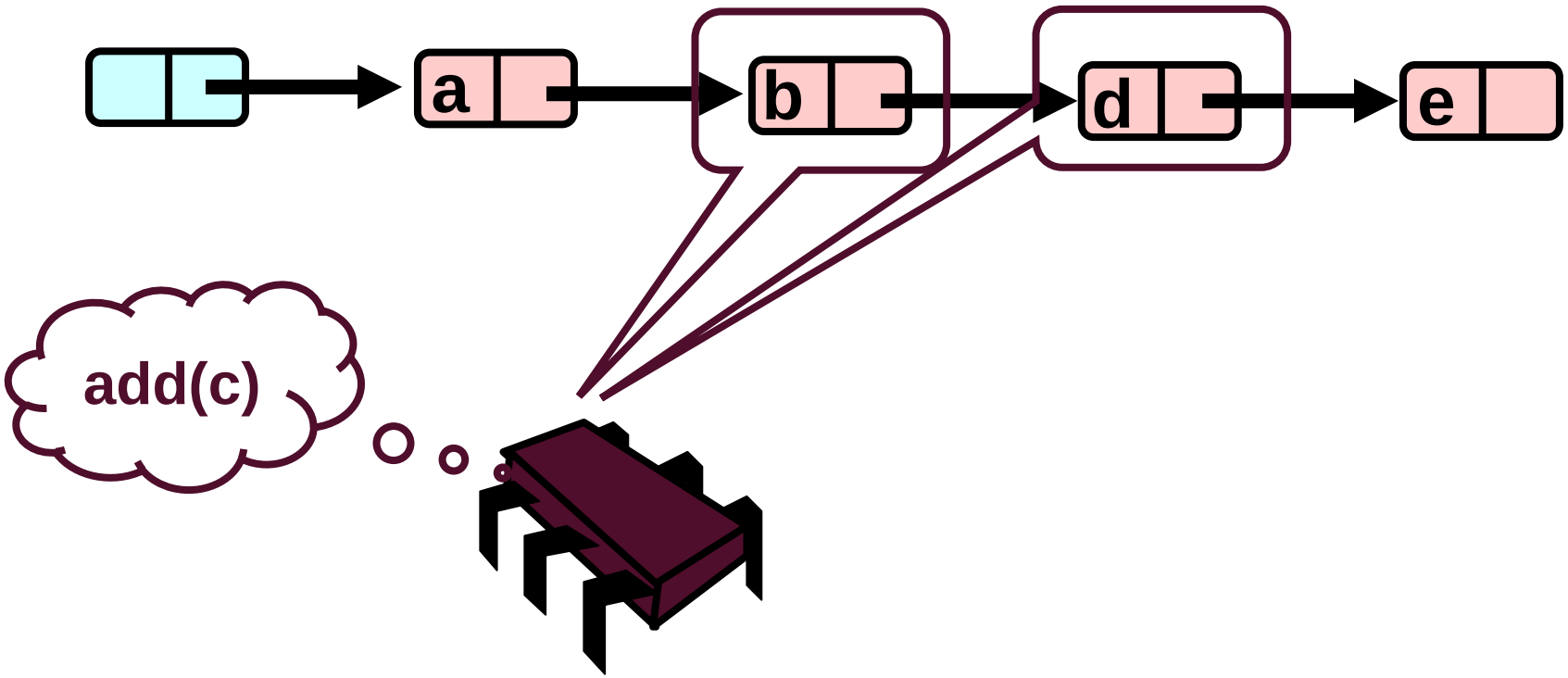
Optimistic: Lock and Load



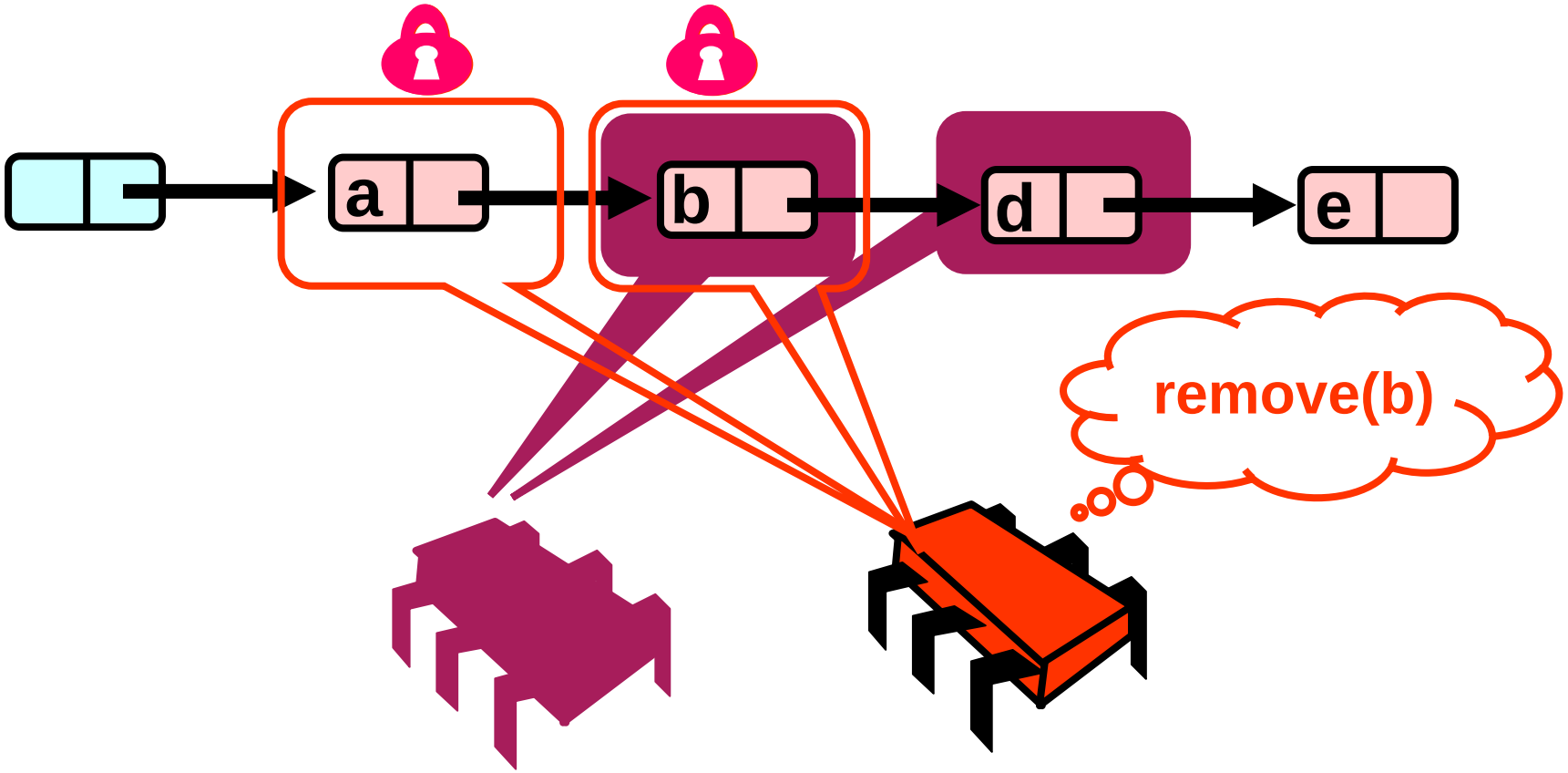
What could go wrong?



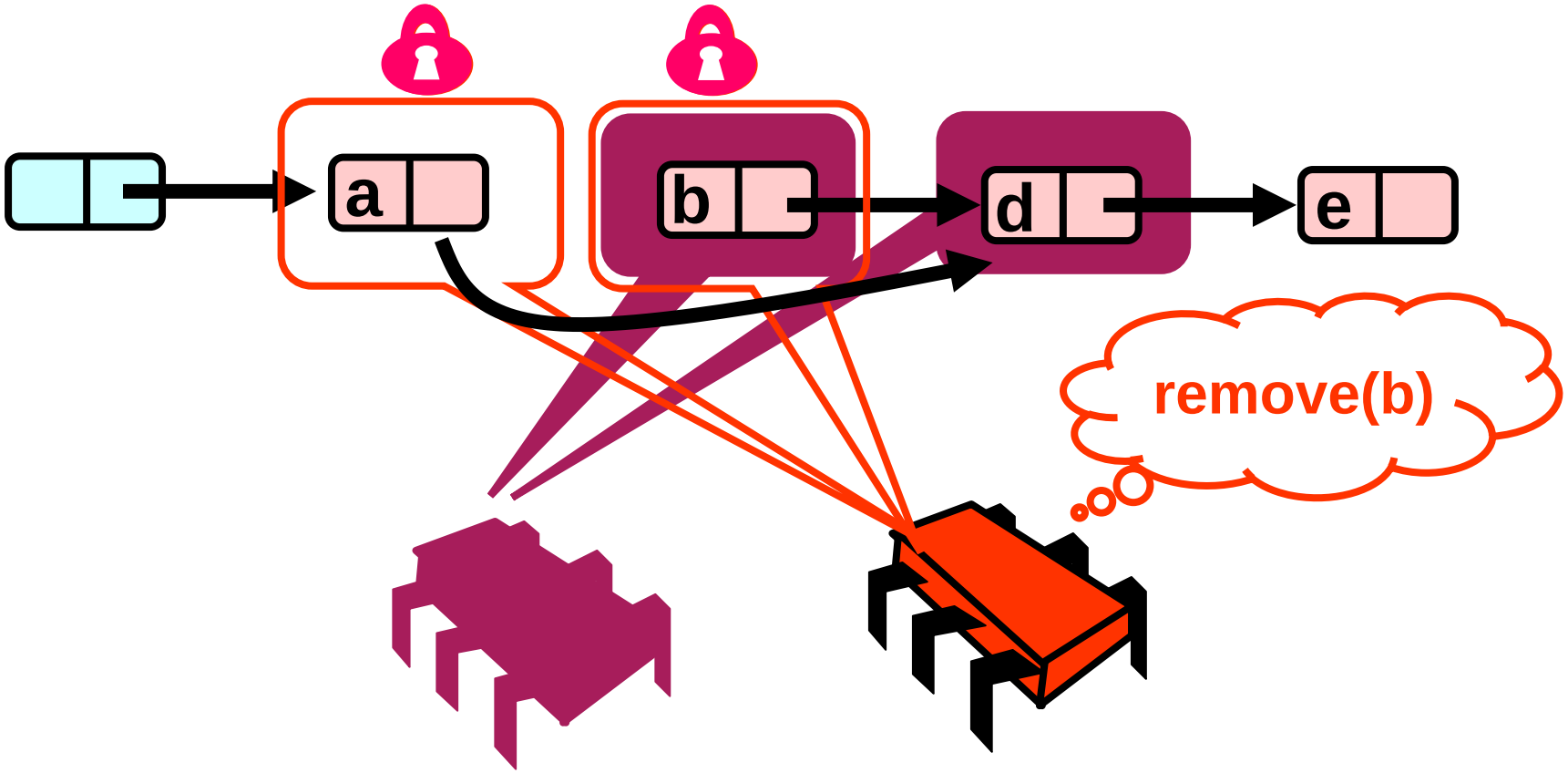
What could go wrong?



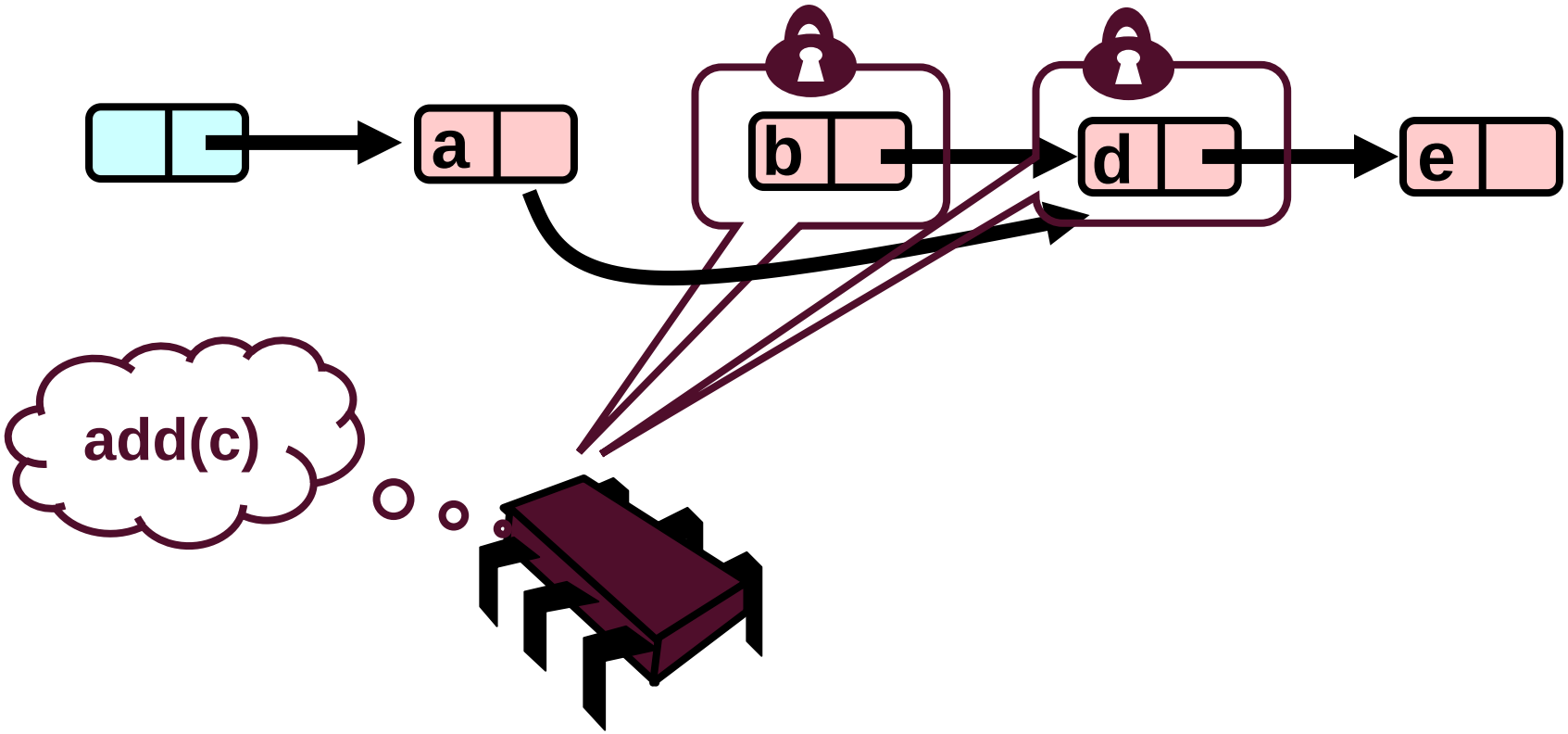
What could go wrong?



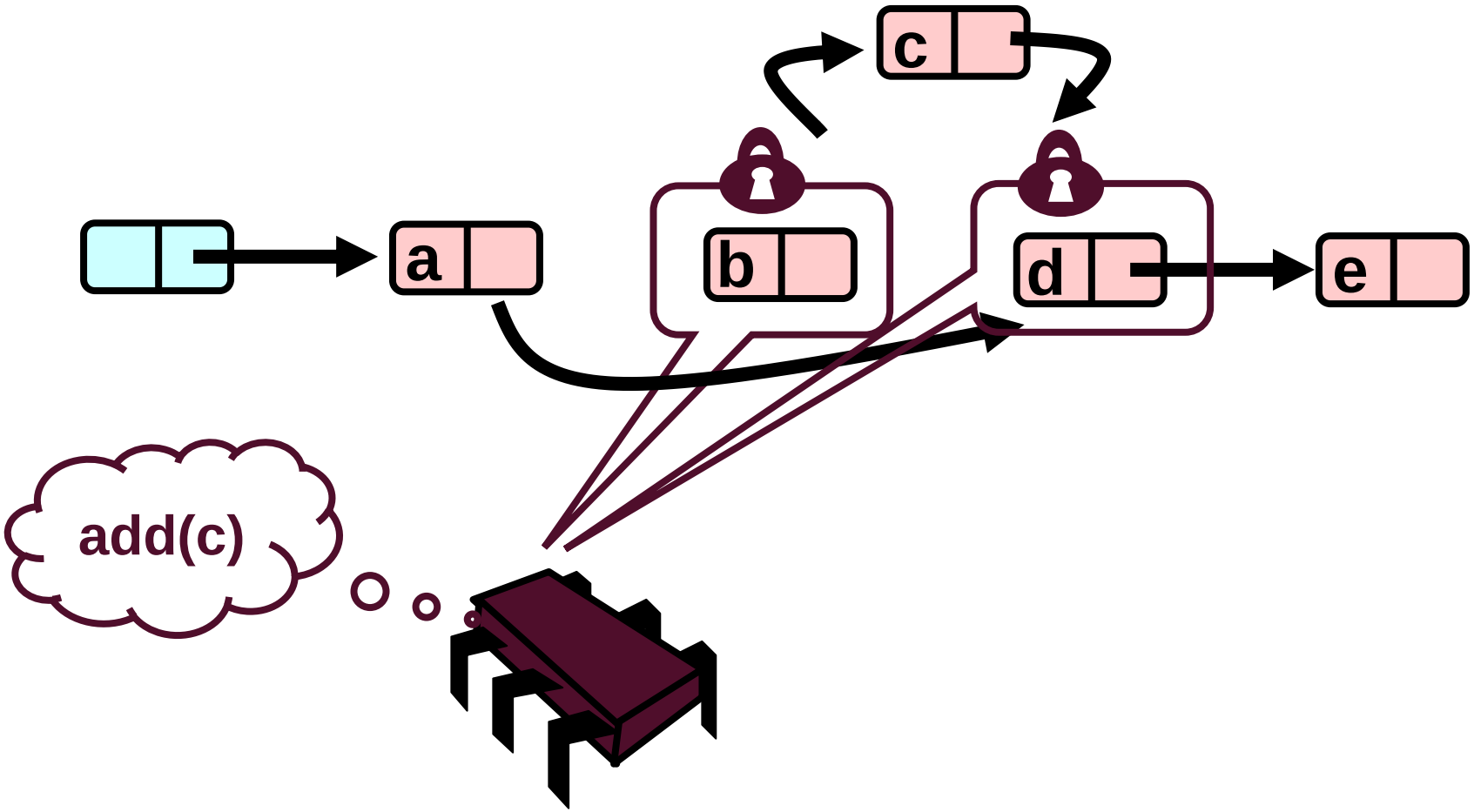
What could go wrong?



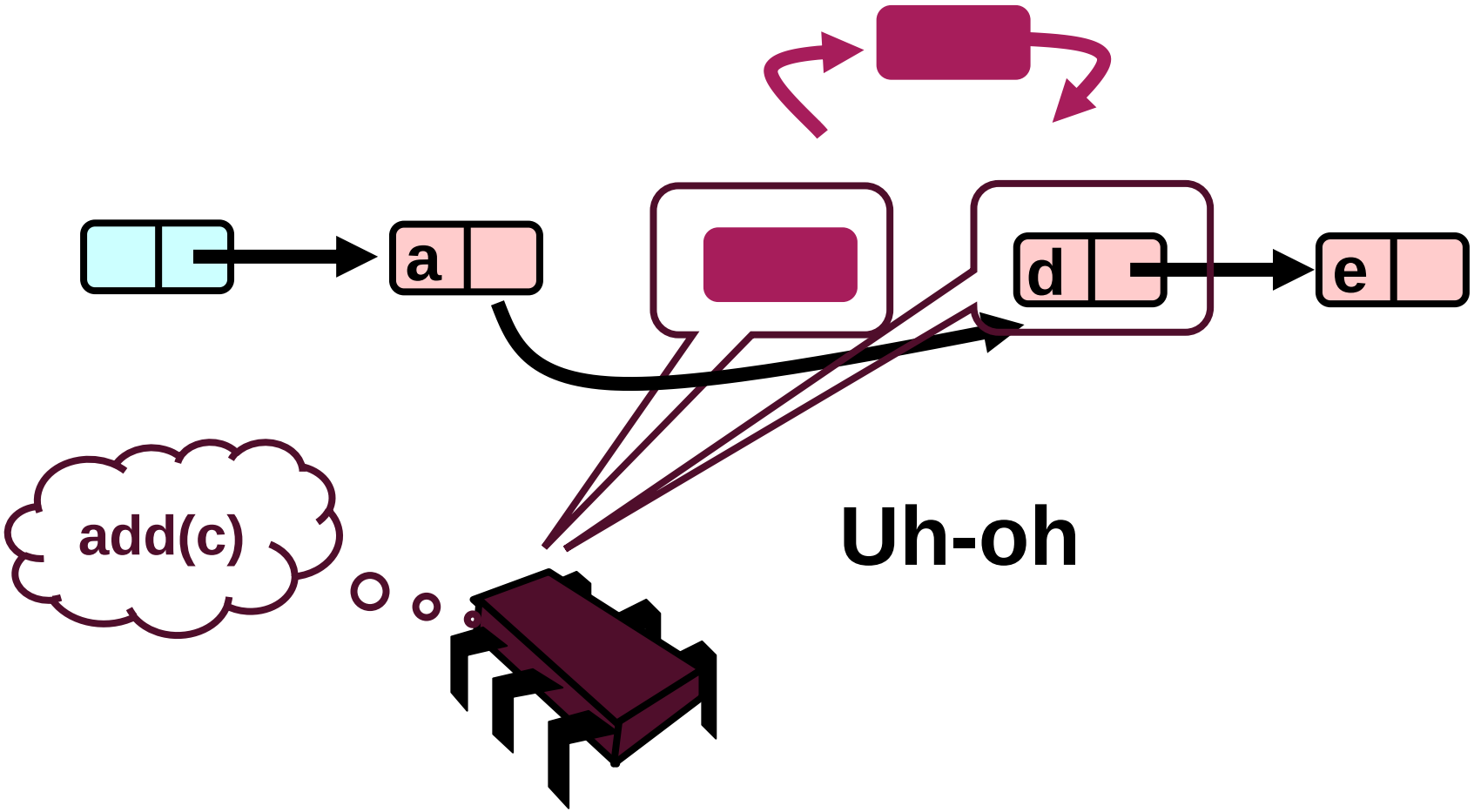
What could go wrong?



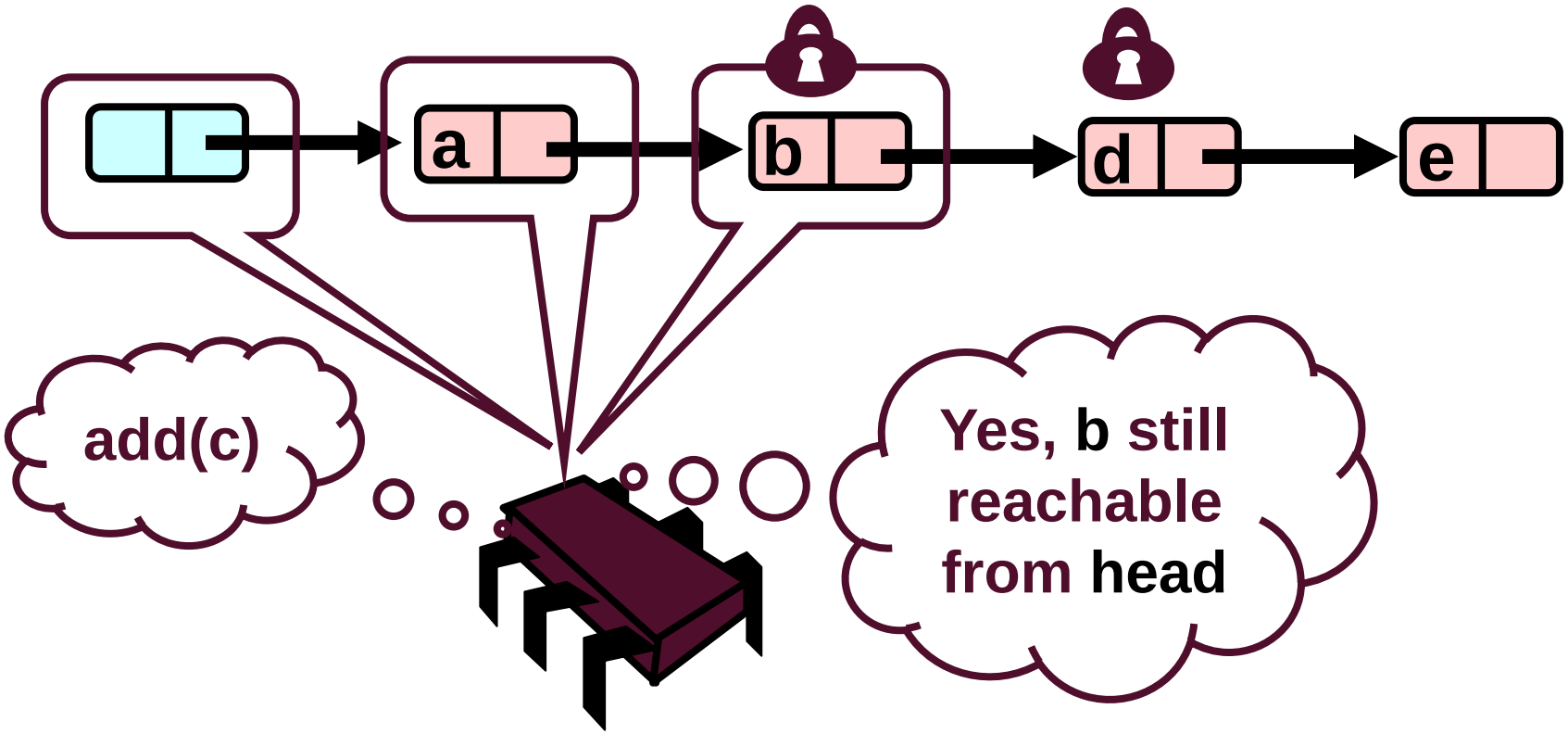
What could go wrong?



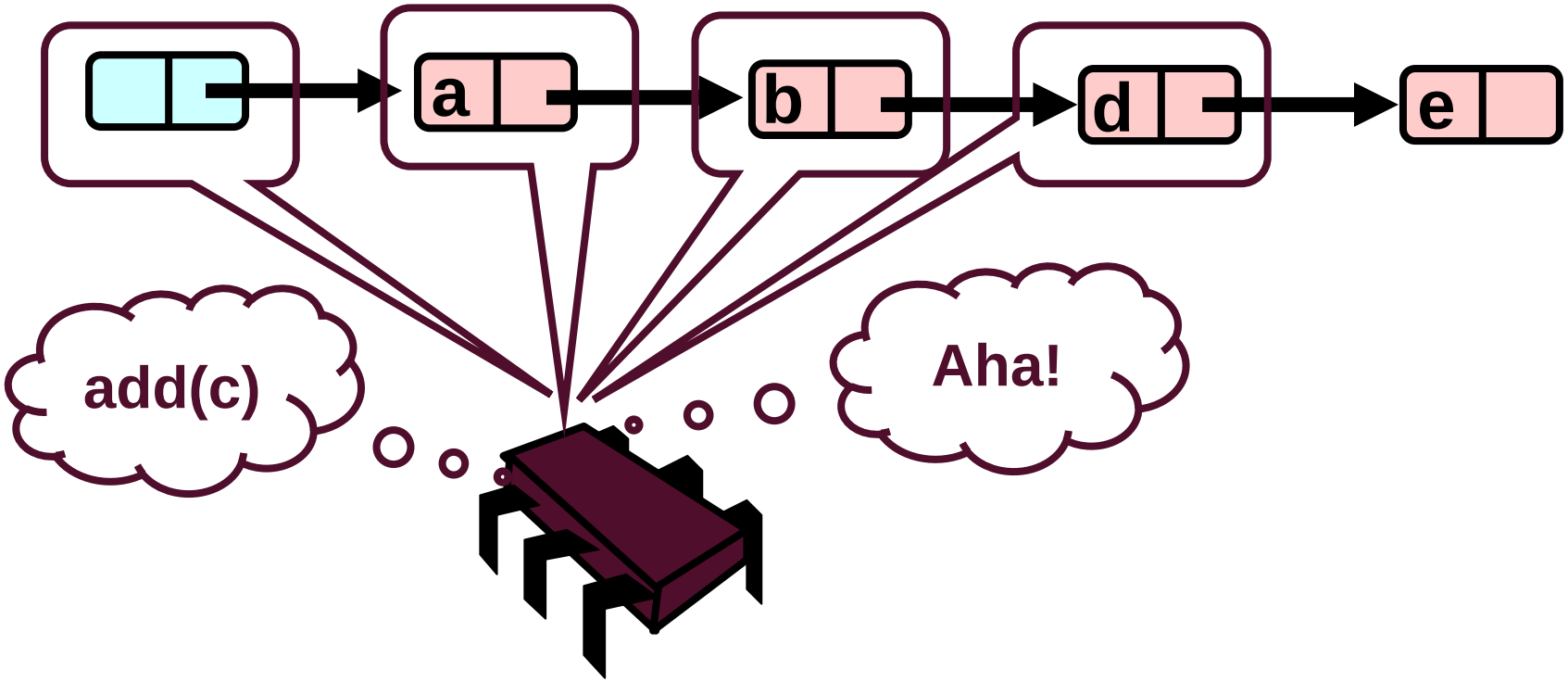
What could go wrong?



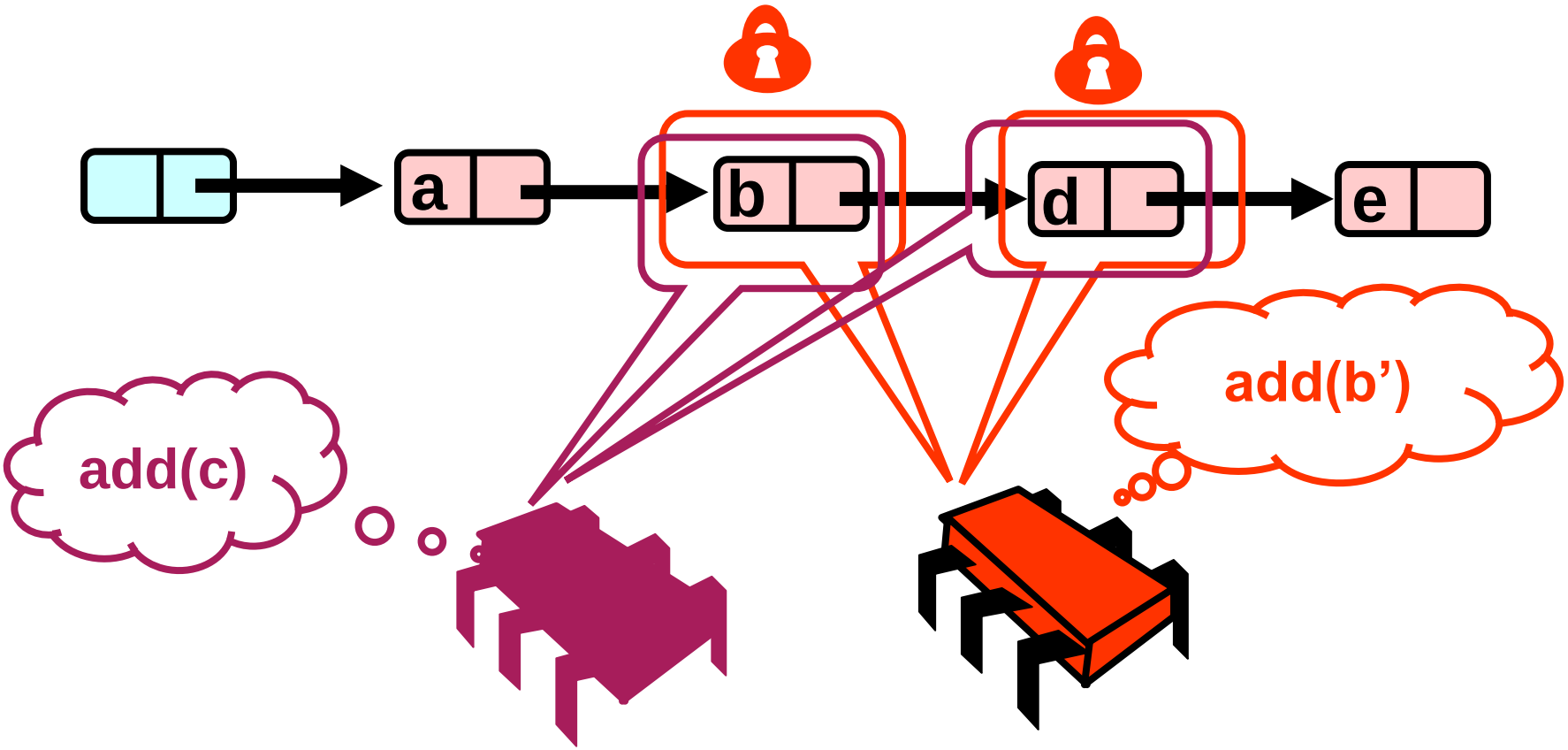
Validate – Part 1



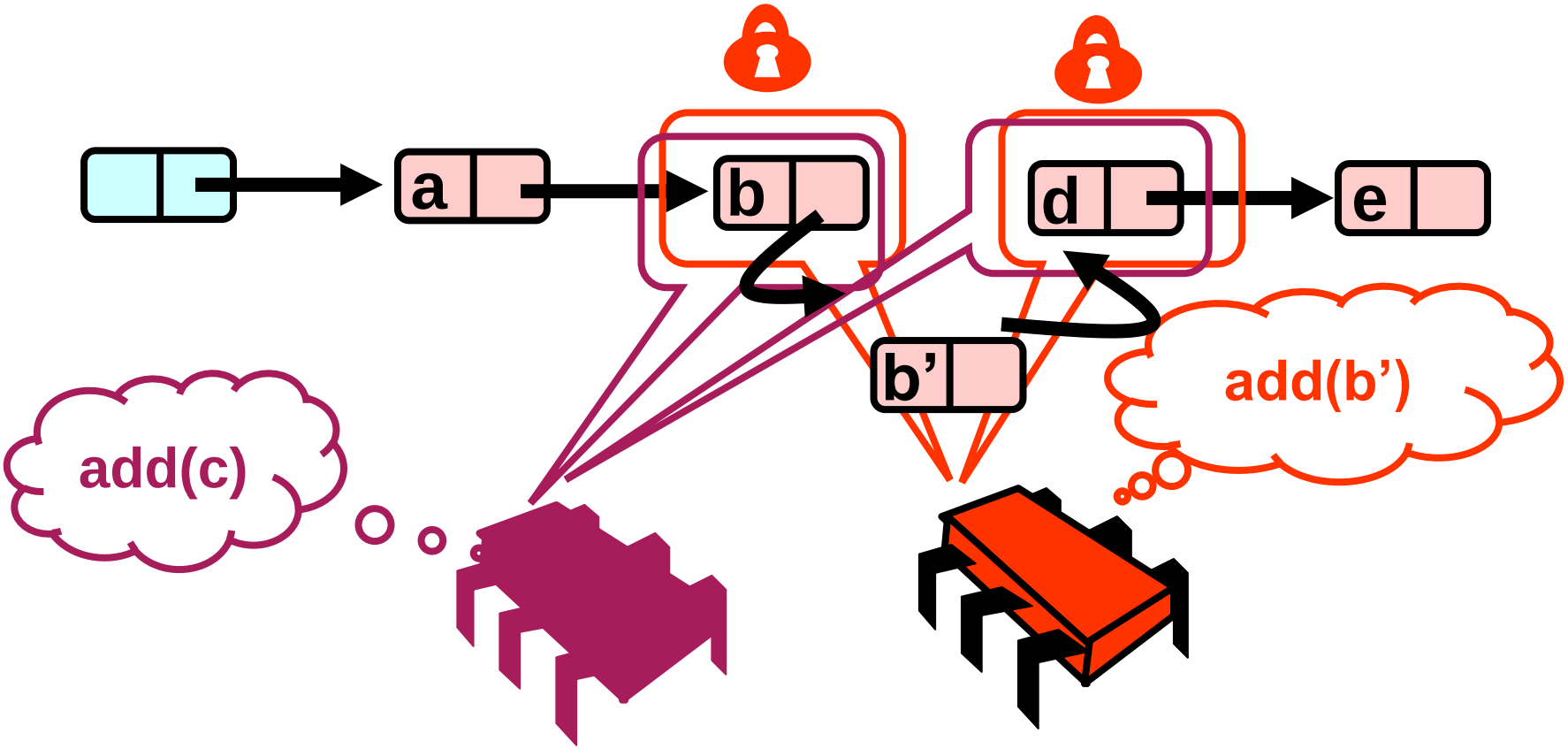
What Else Could Go Wrong?



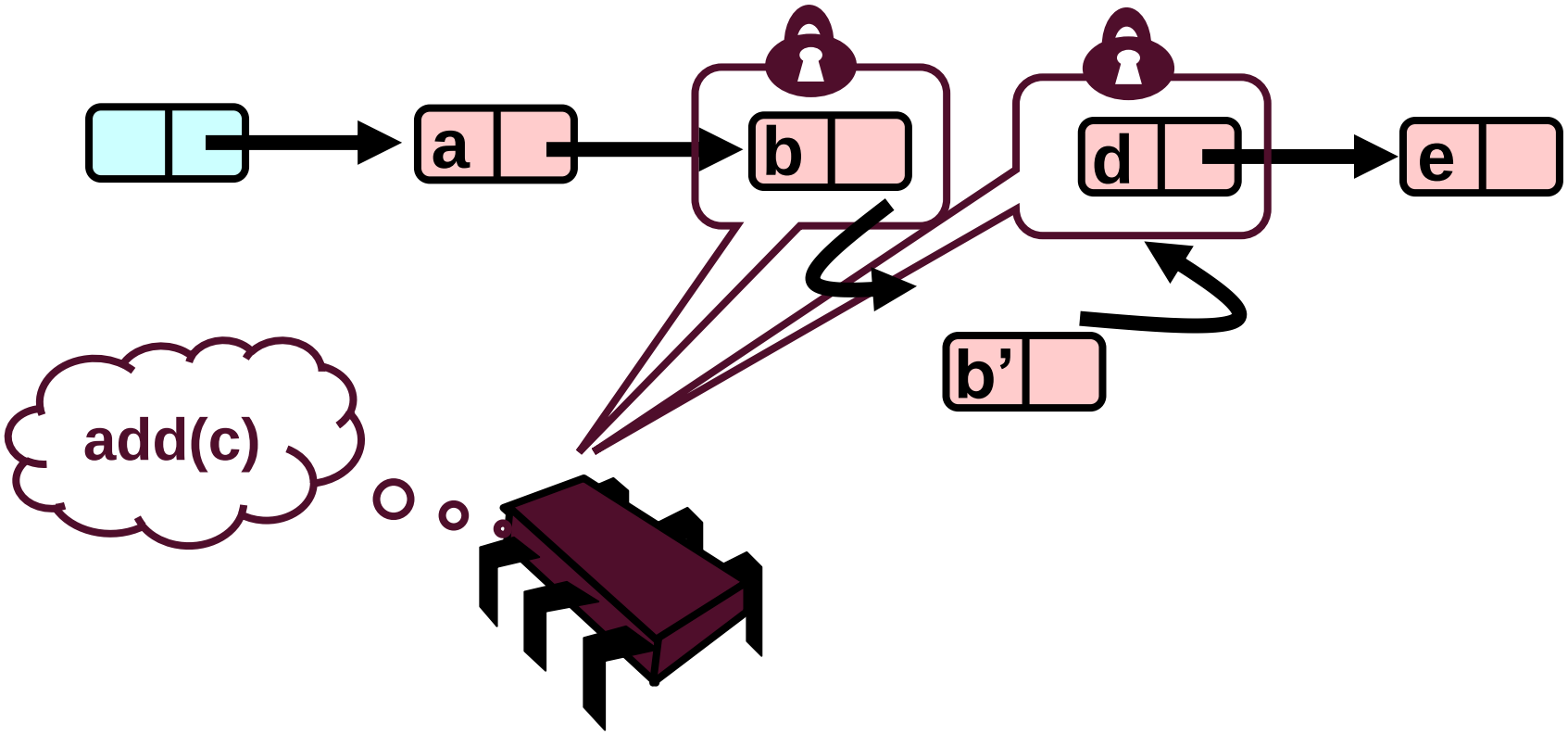
What Else Could Go Wrong?



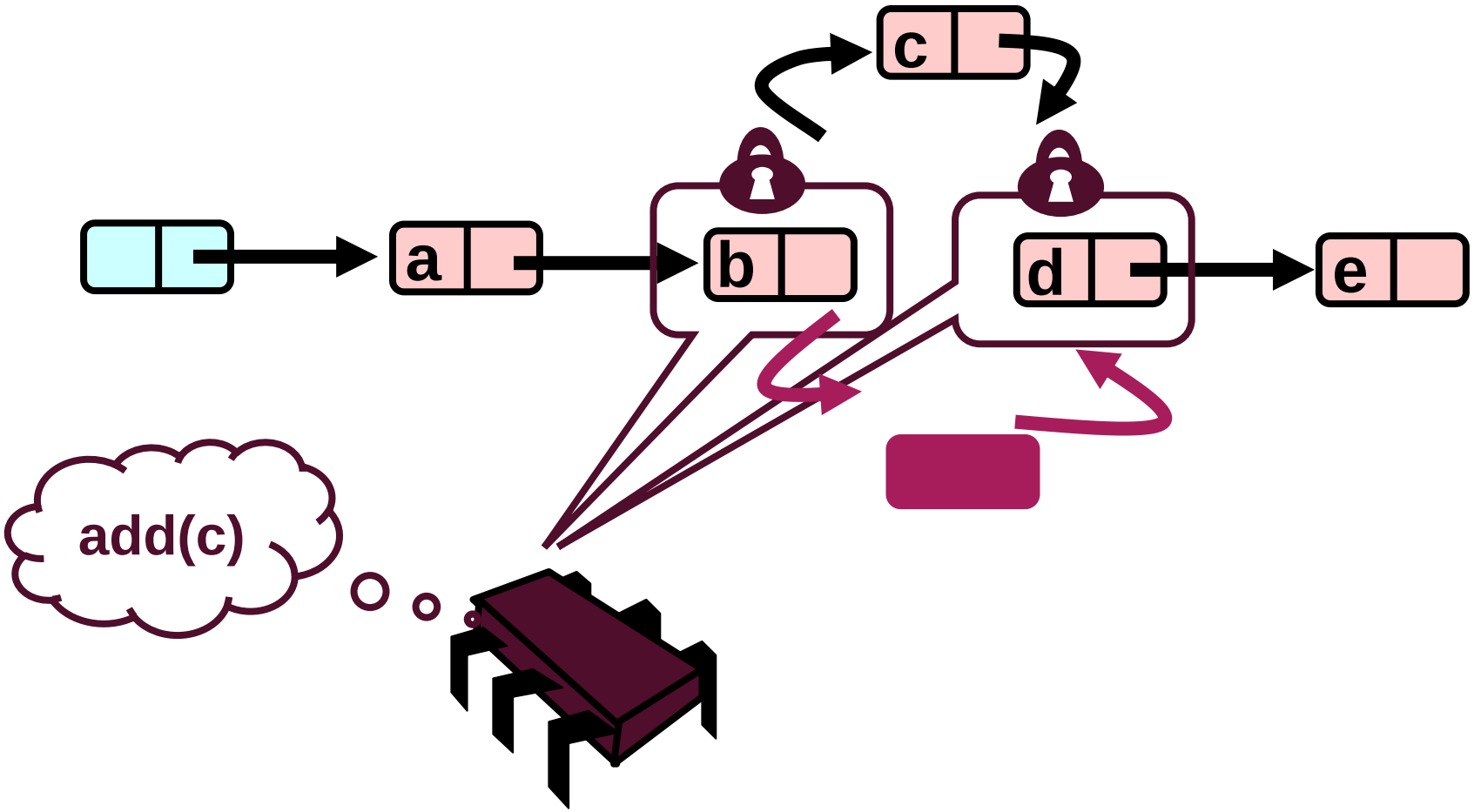
What Else Could Go Wrong?



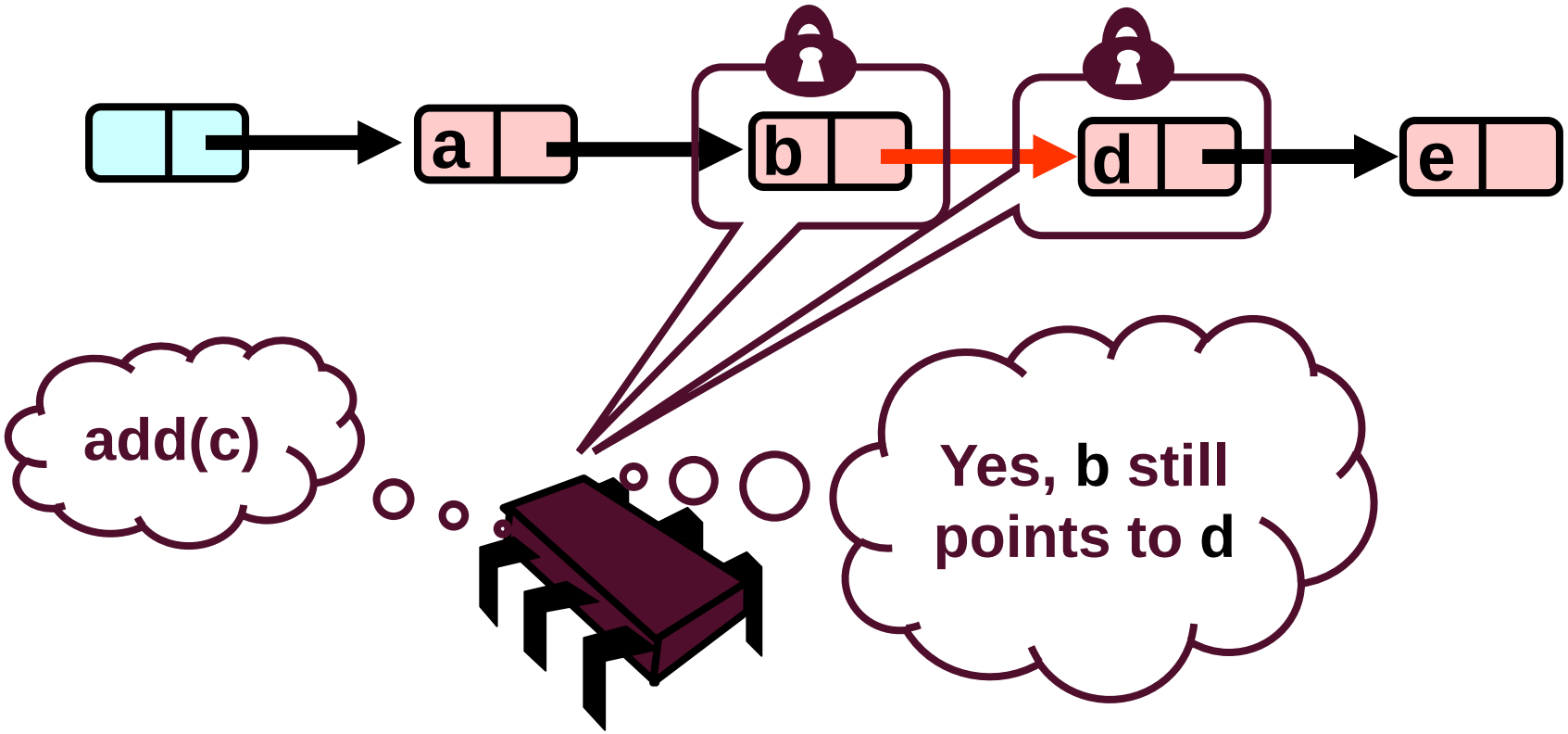
What Else Could Go Wrong?



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?

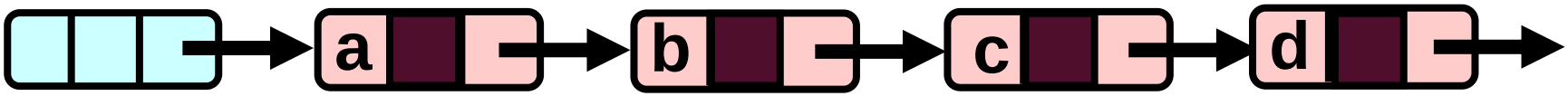
    Start over from the beginning (hopefully rare)
- **Not starvation-free**
  - A thread may need to abort forever if nodes are added/removed
  - Should be rare in practice!
- **Other disadvantages?**
  - All operations require two traversals of the list!
  - Even contains() needs to check if node is still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

Trick 4: Lazy synchronization

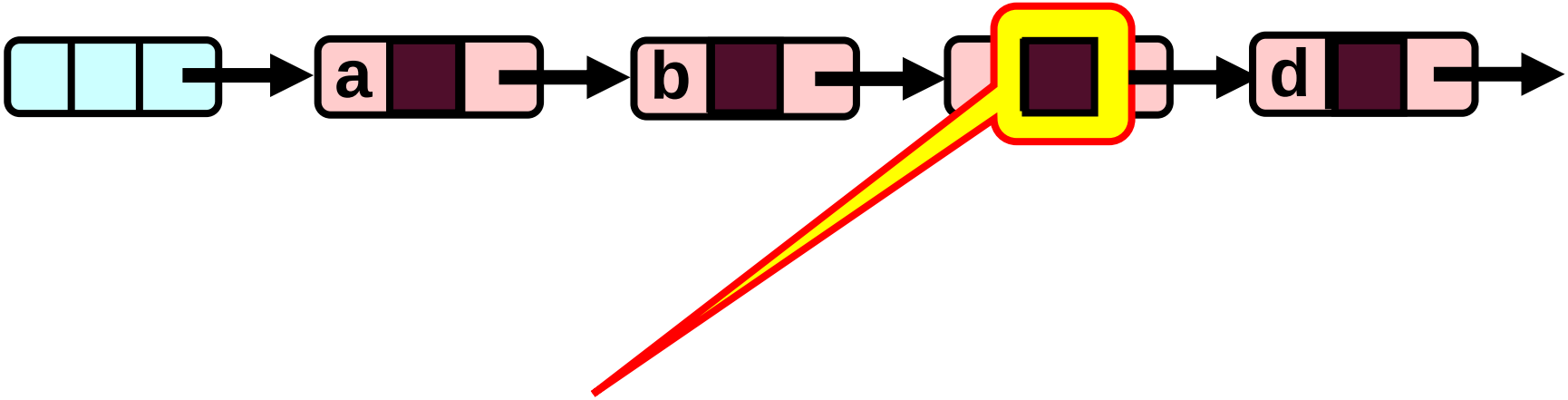
- **We really want one list traversal**
- **Also, contains() should be wait-free**
  - Is probably the most-used operation
- **Lazy locking is similar to optimistic**
  - Key insight: removing is problematic
  - Perform it "lazily"
- **Add a new "valid" field**
  - Indicates if node is still in the set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

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

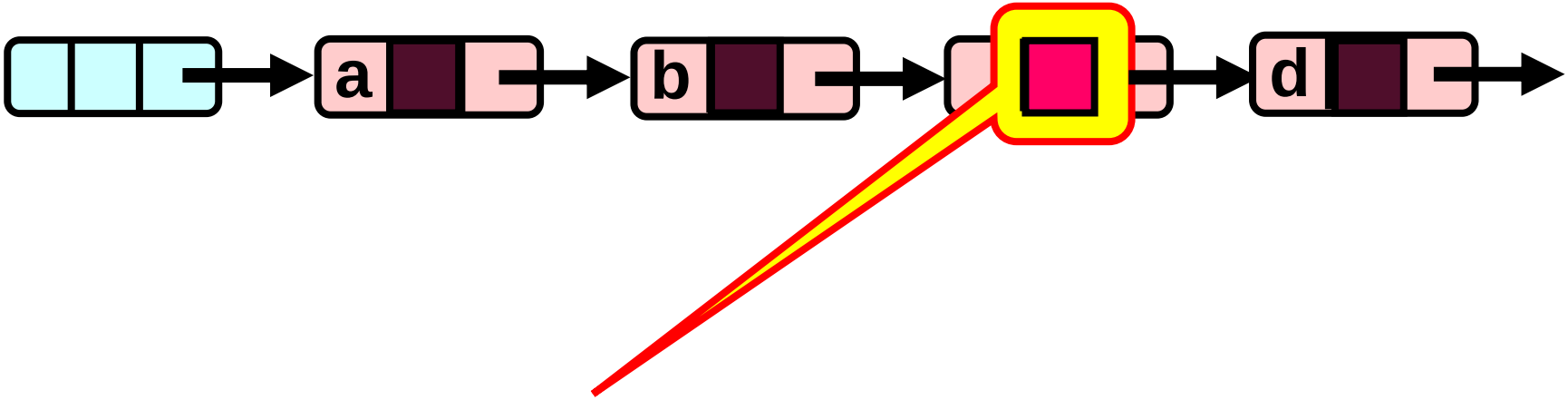
Lazy Removal



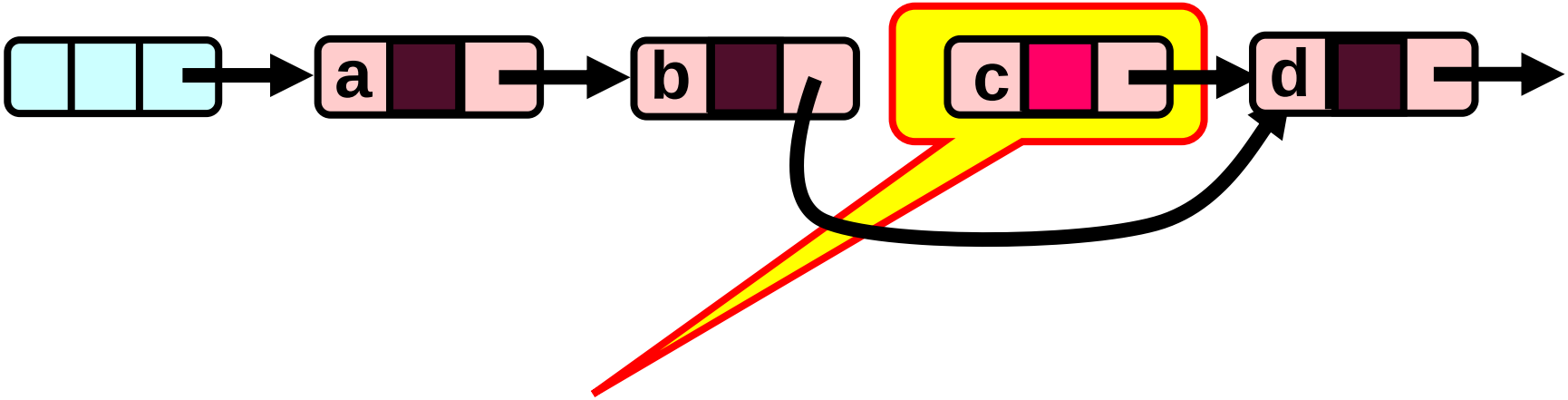
Lazy Removal



Lazy Removal

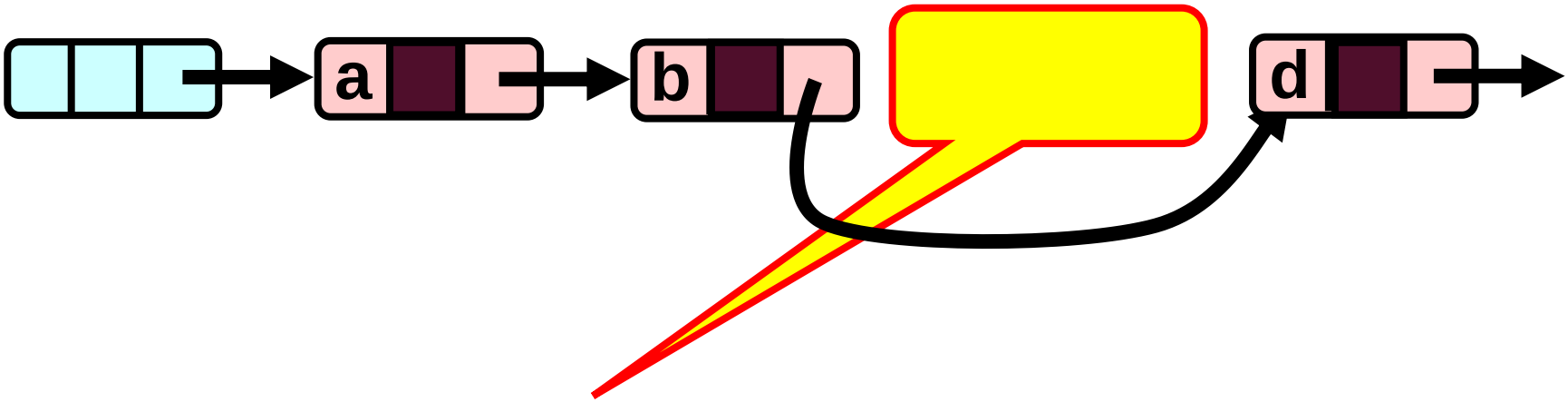


Lazy Removal



Physically deleted

Lazy Removal

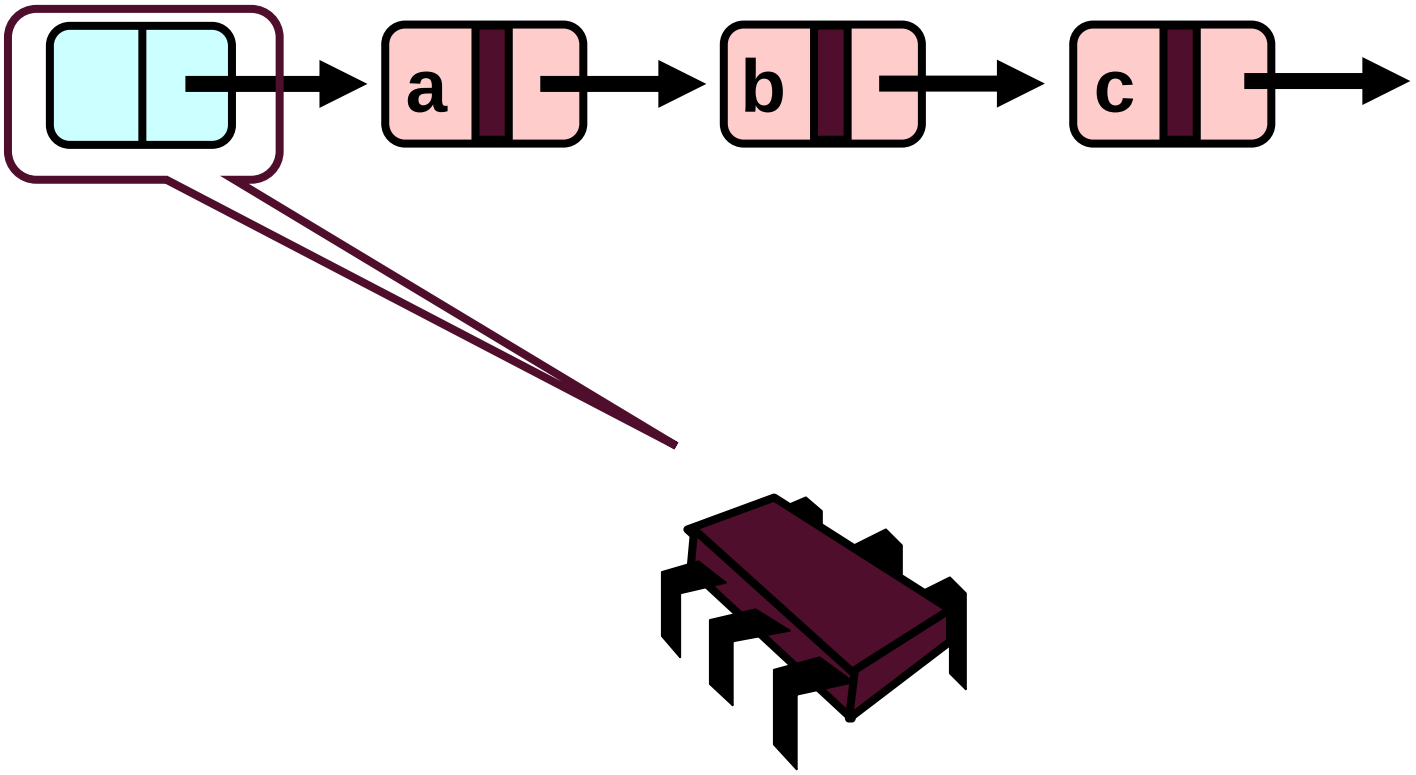


How does it work?

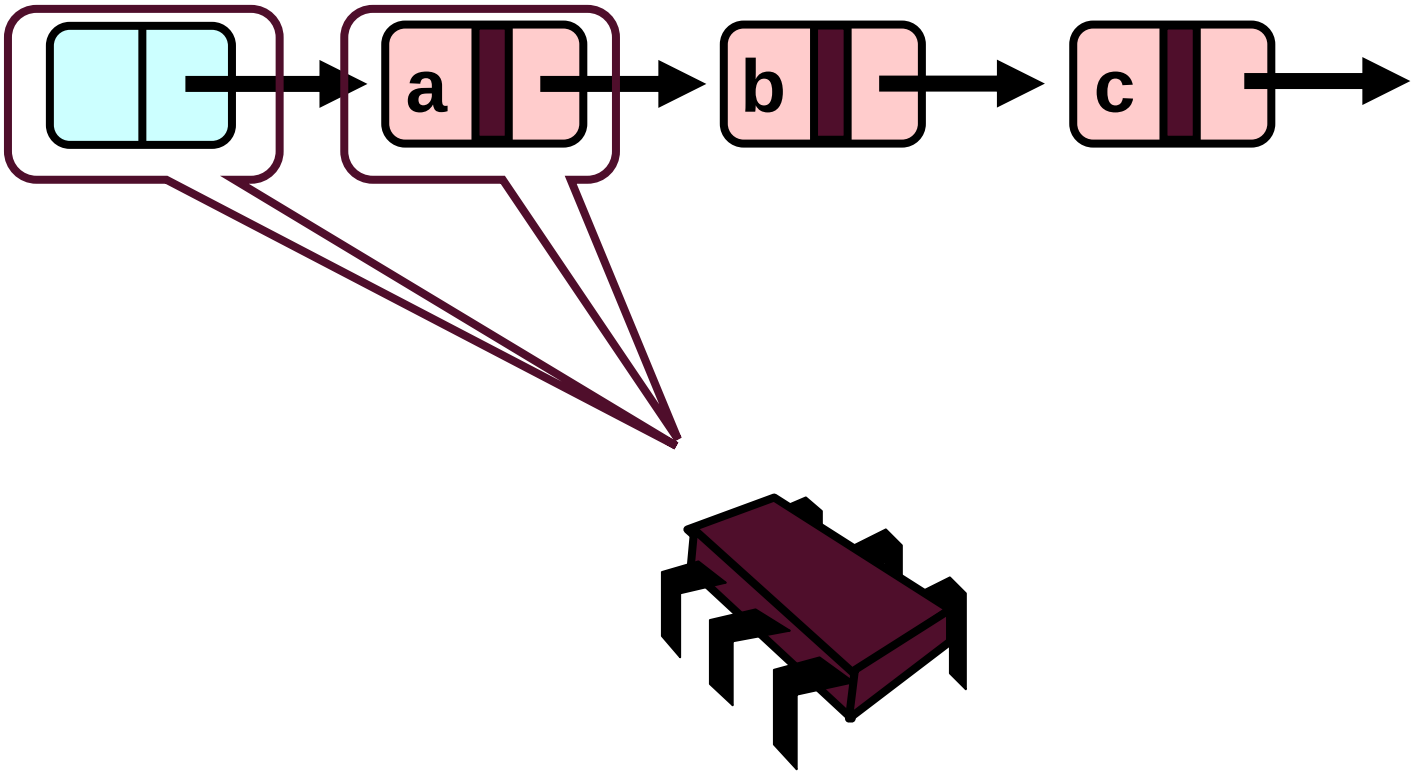
- **Eliminates need to re-scan list for reachability**
  - Maintains invariant that every **unmarked** node is reachable!
- **Contains can now simply traverse the list**
  - Just check marks, not reachability, no locks
- **Remove/Add**
  - Scan through locked and marked nodes
  - Removing does not delay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

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

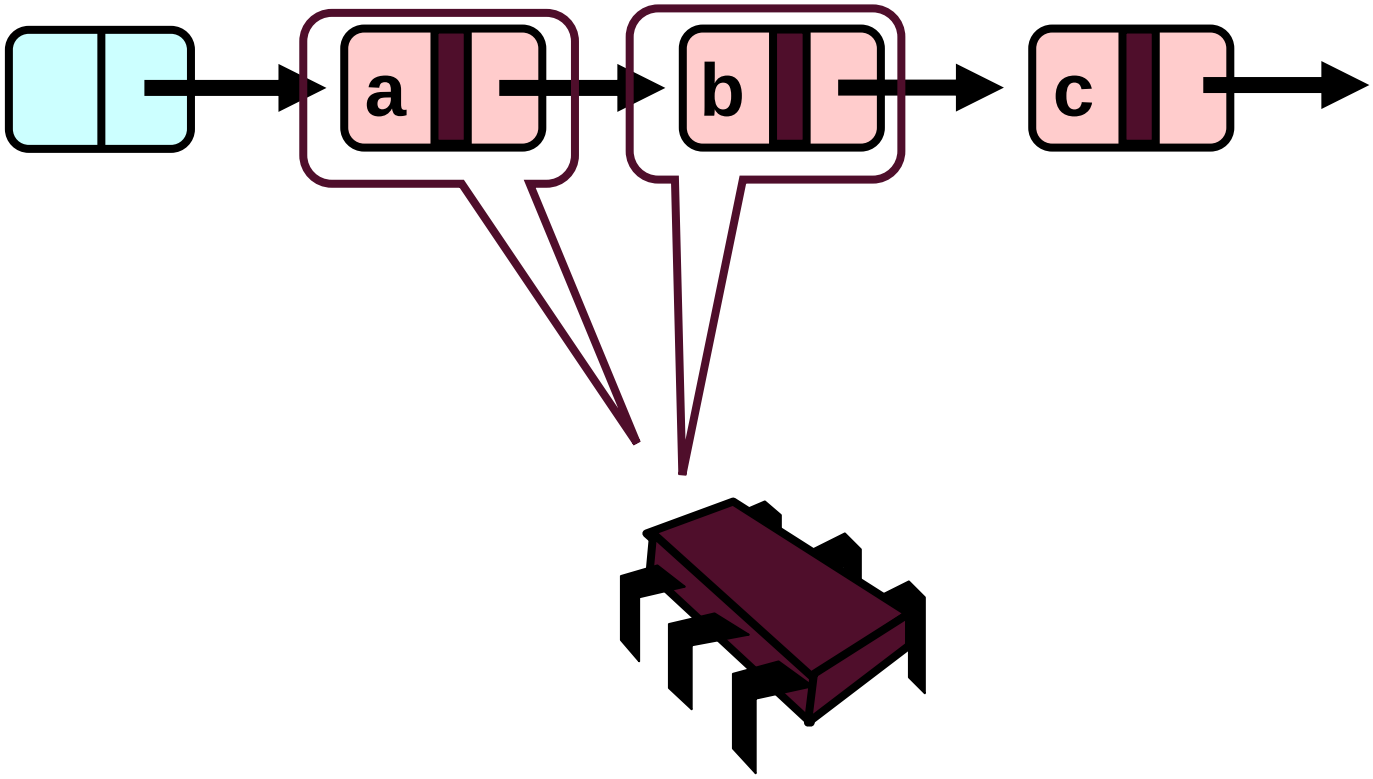
Business as Usual



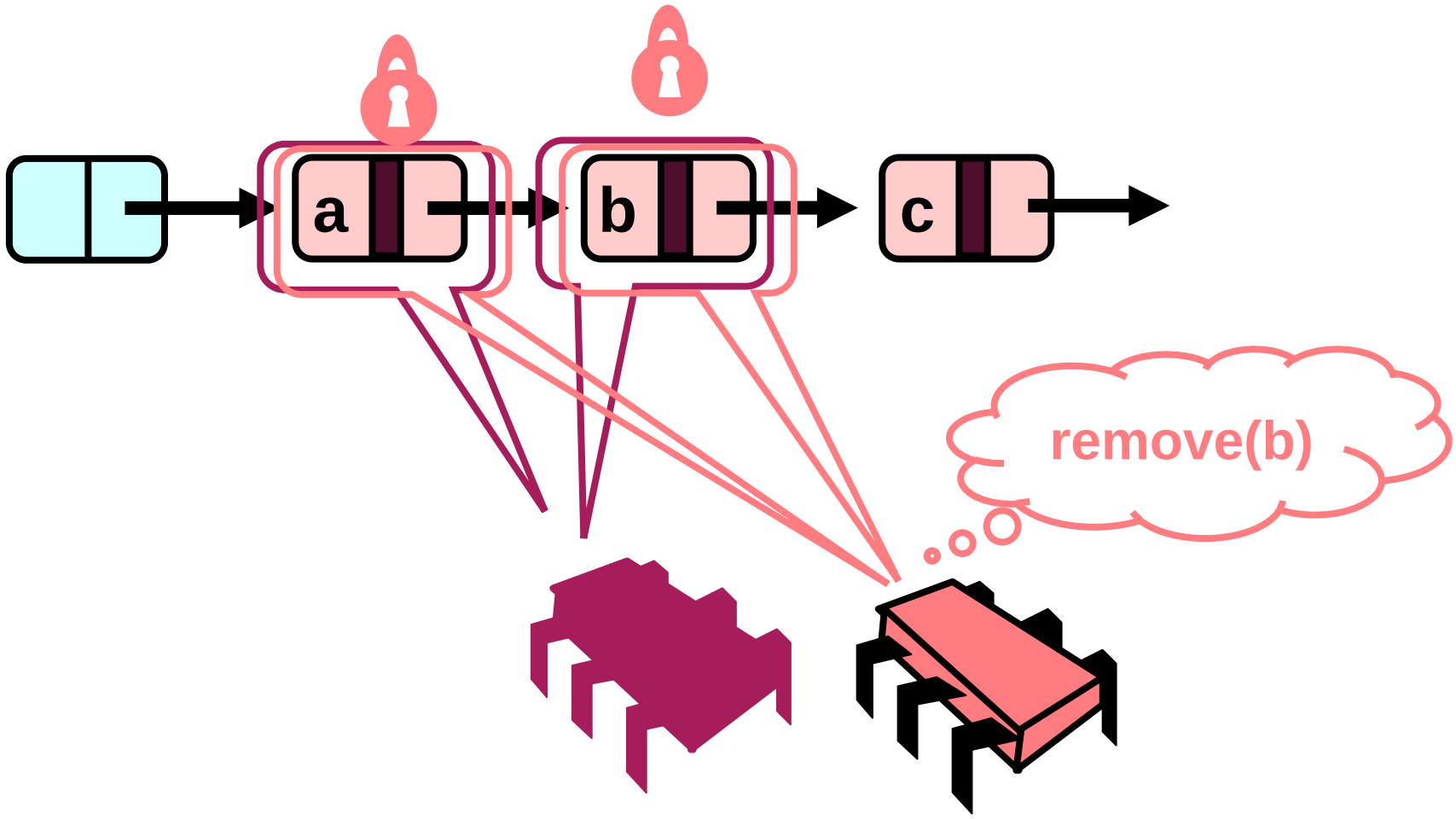
Business as Usual



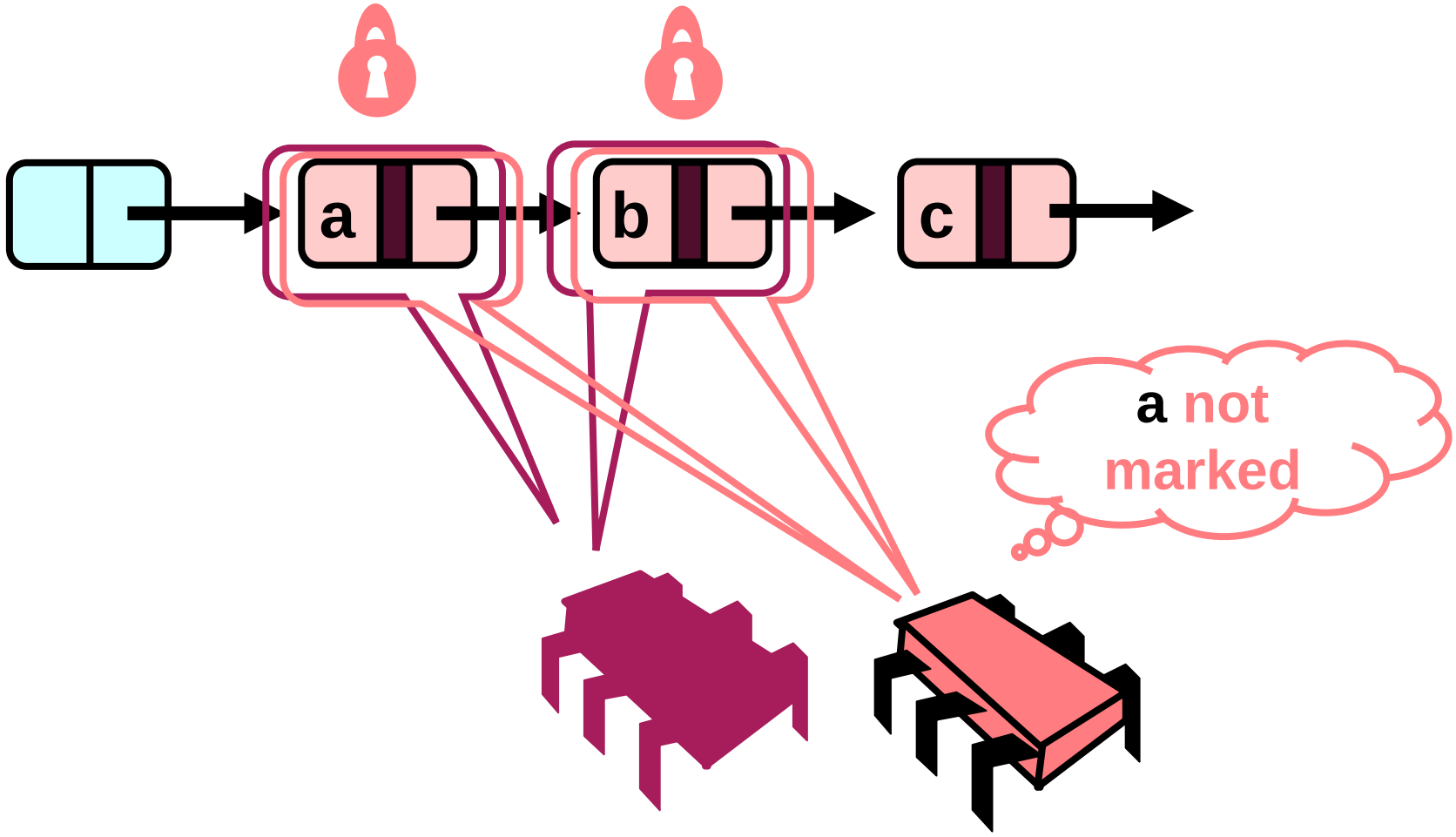
Business as Usual



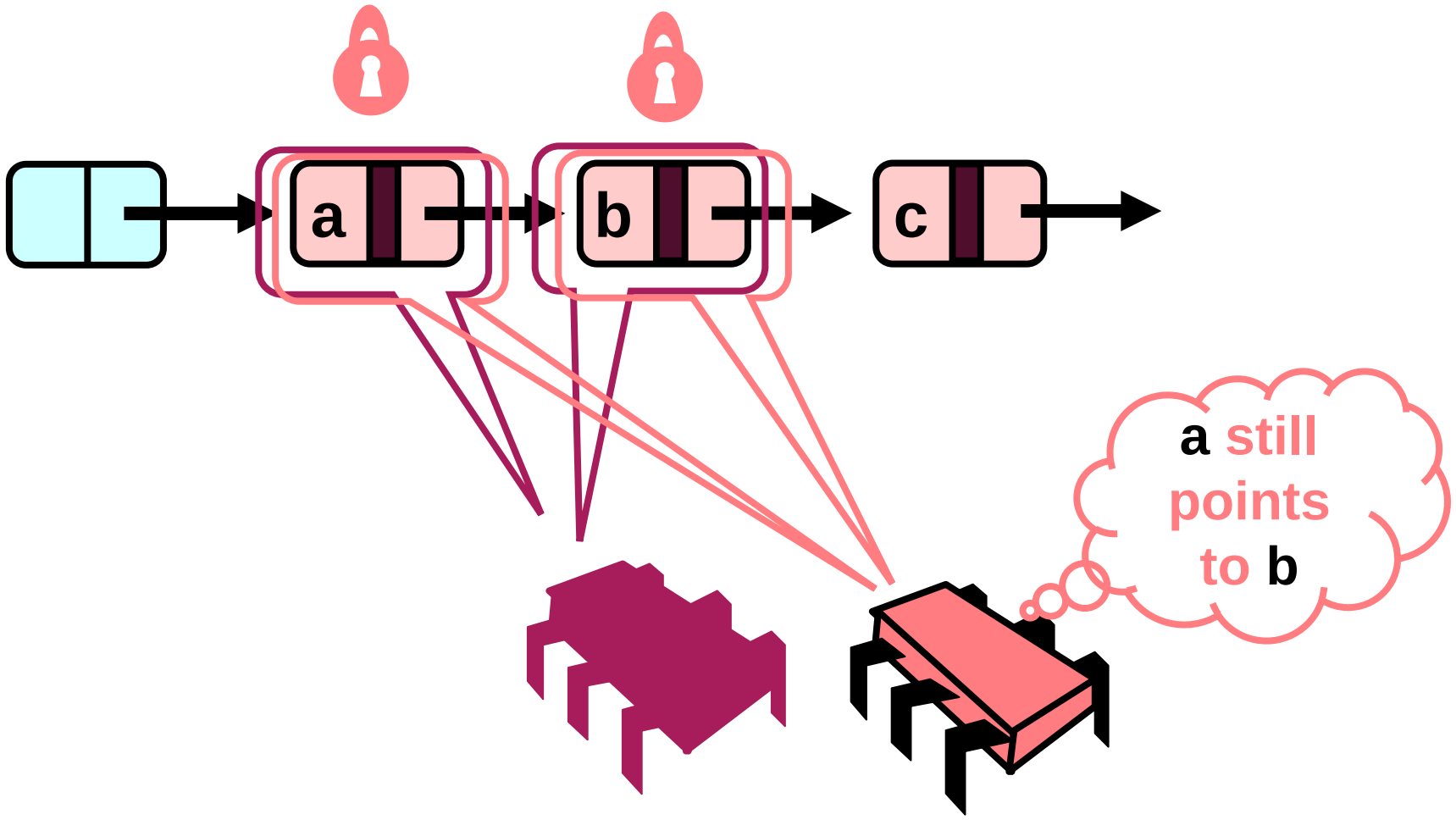
Business as Usual



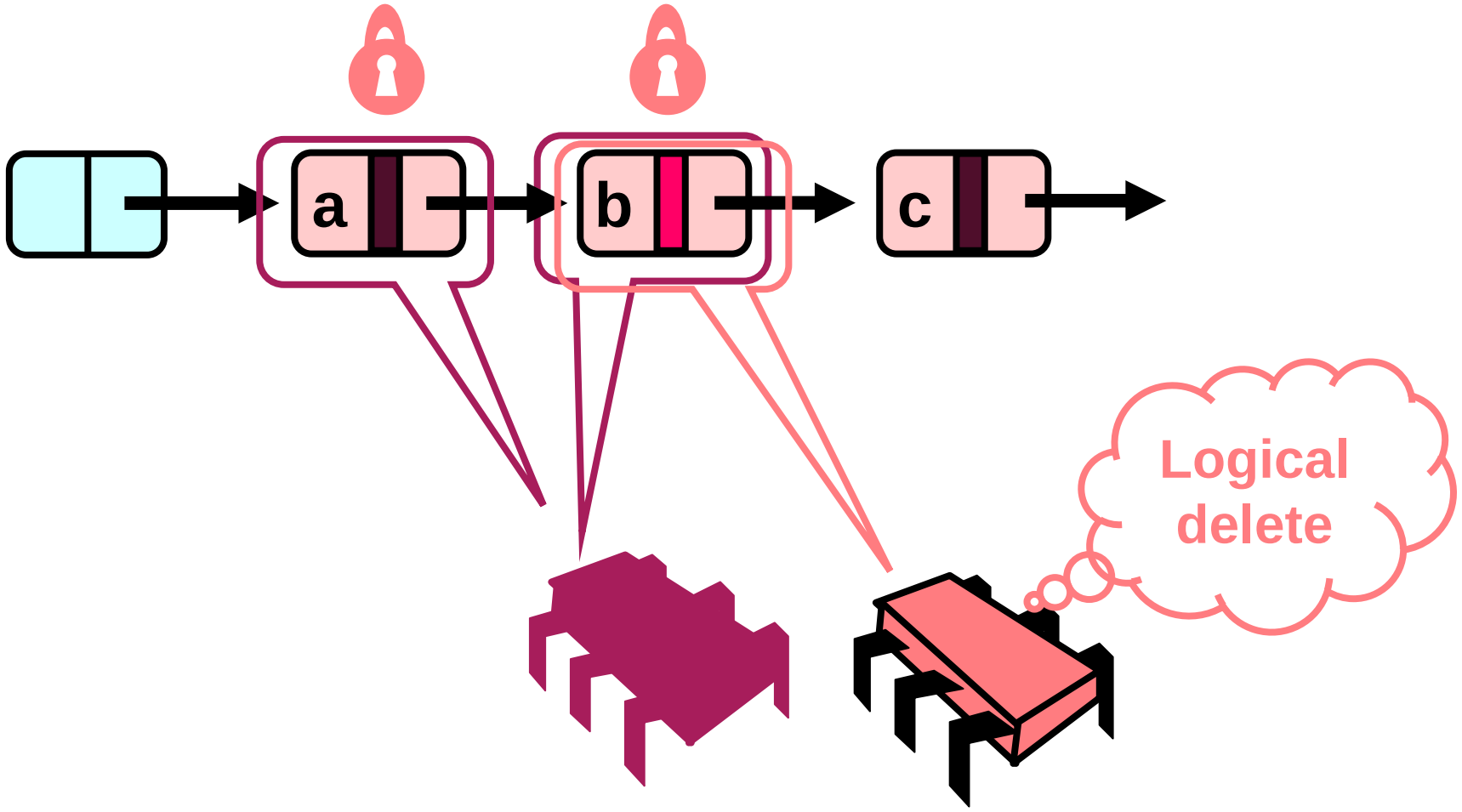
Business as Usual



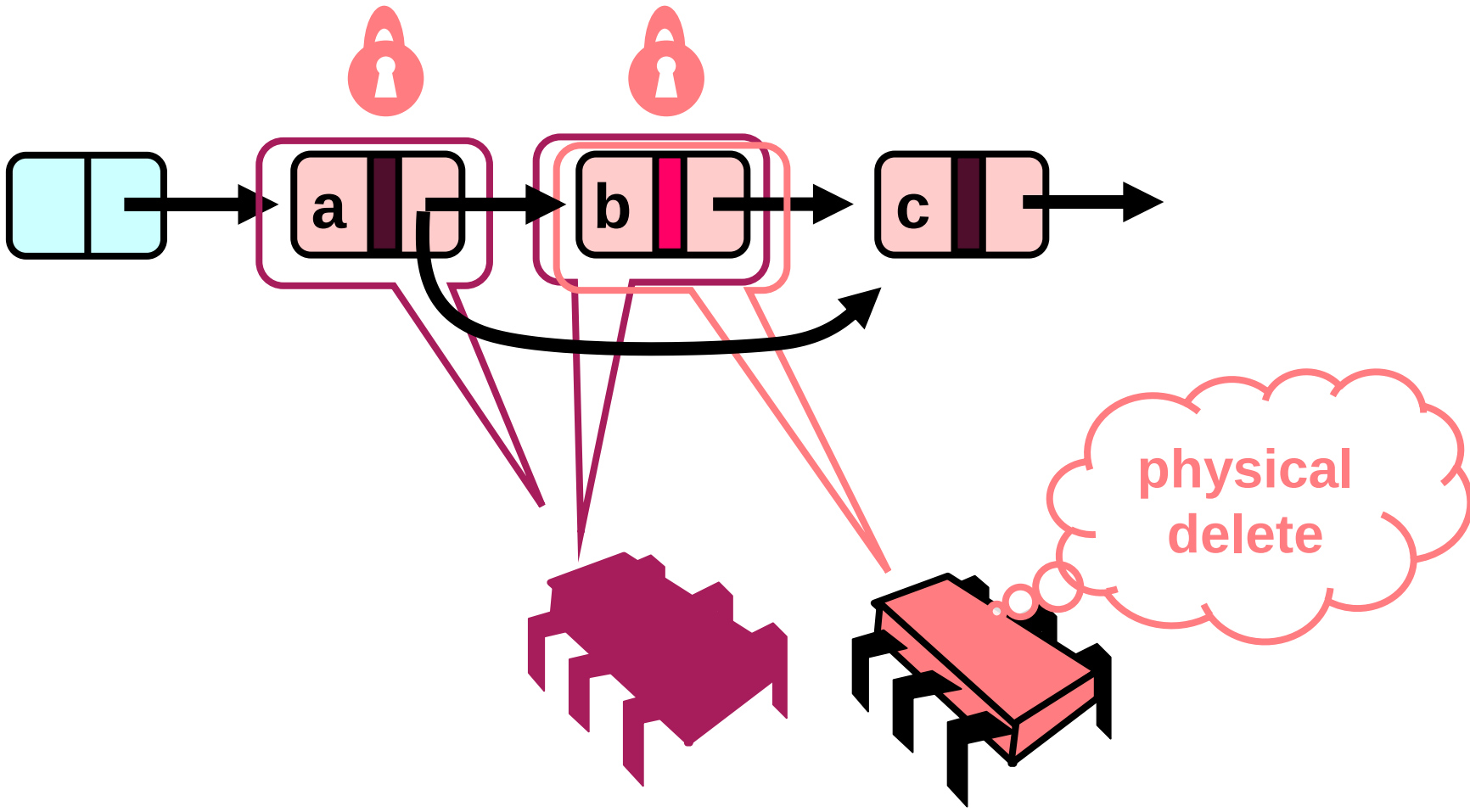
Business as Usual



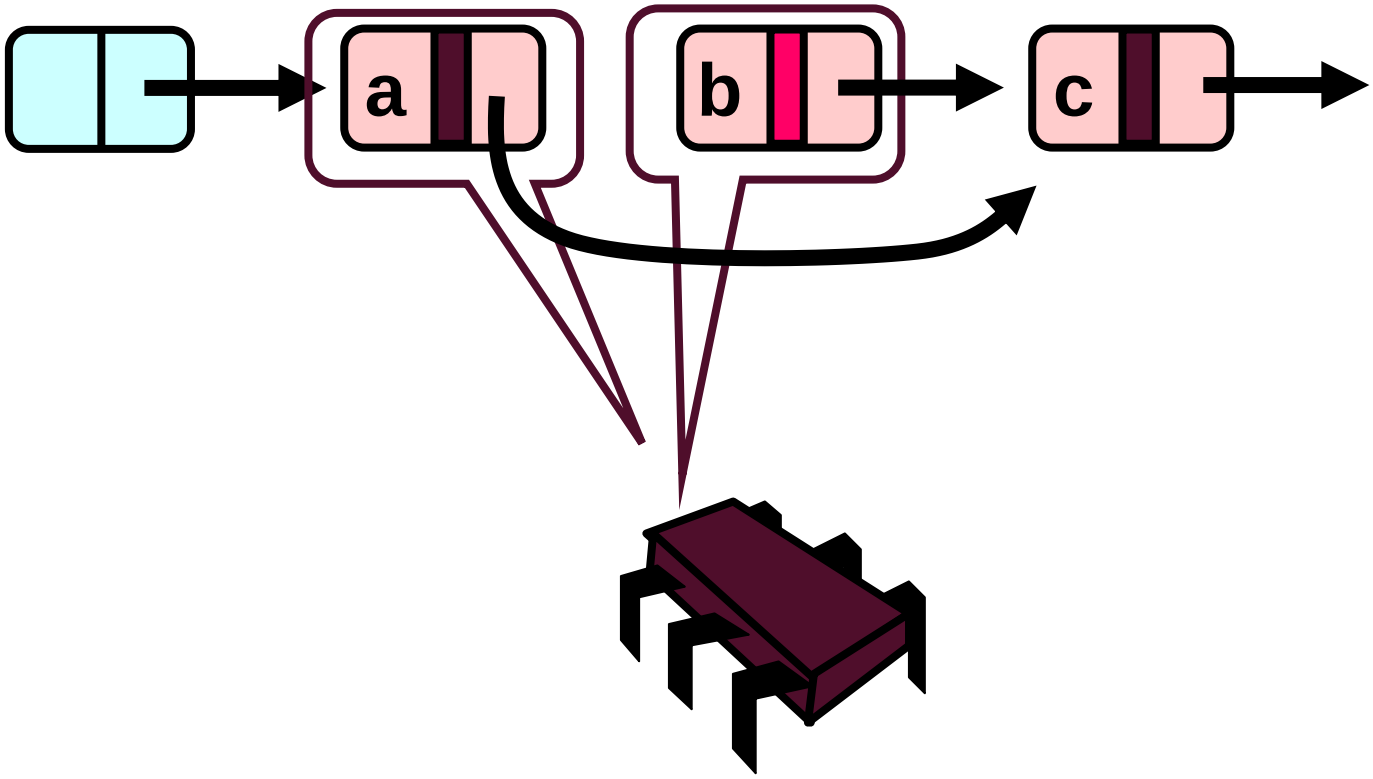
Business as Usual



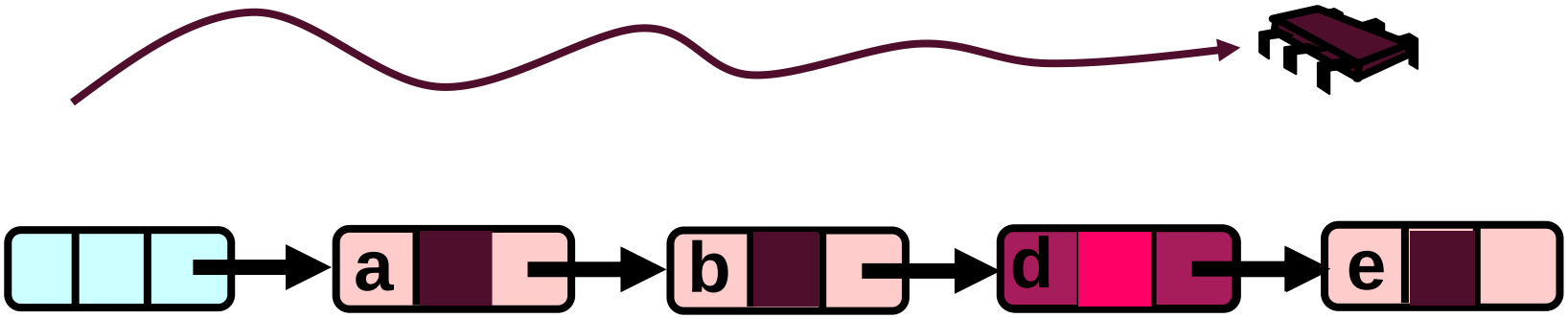
Business as Usual



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

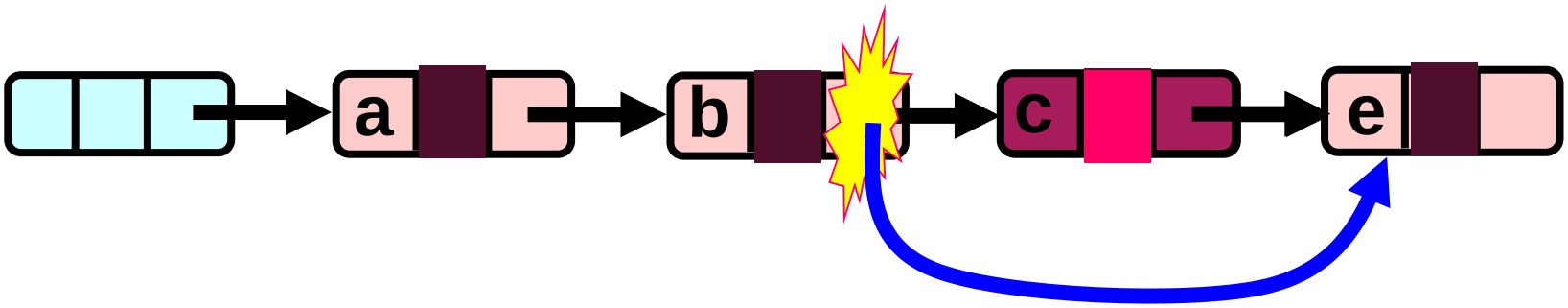
Trick 5: No Locking

- **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

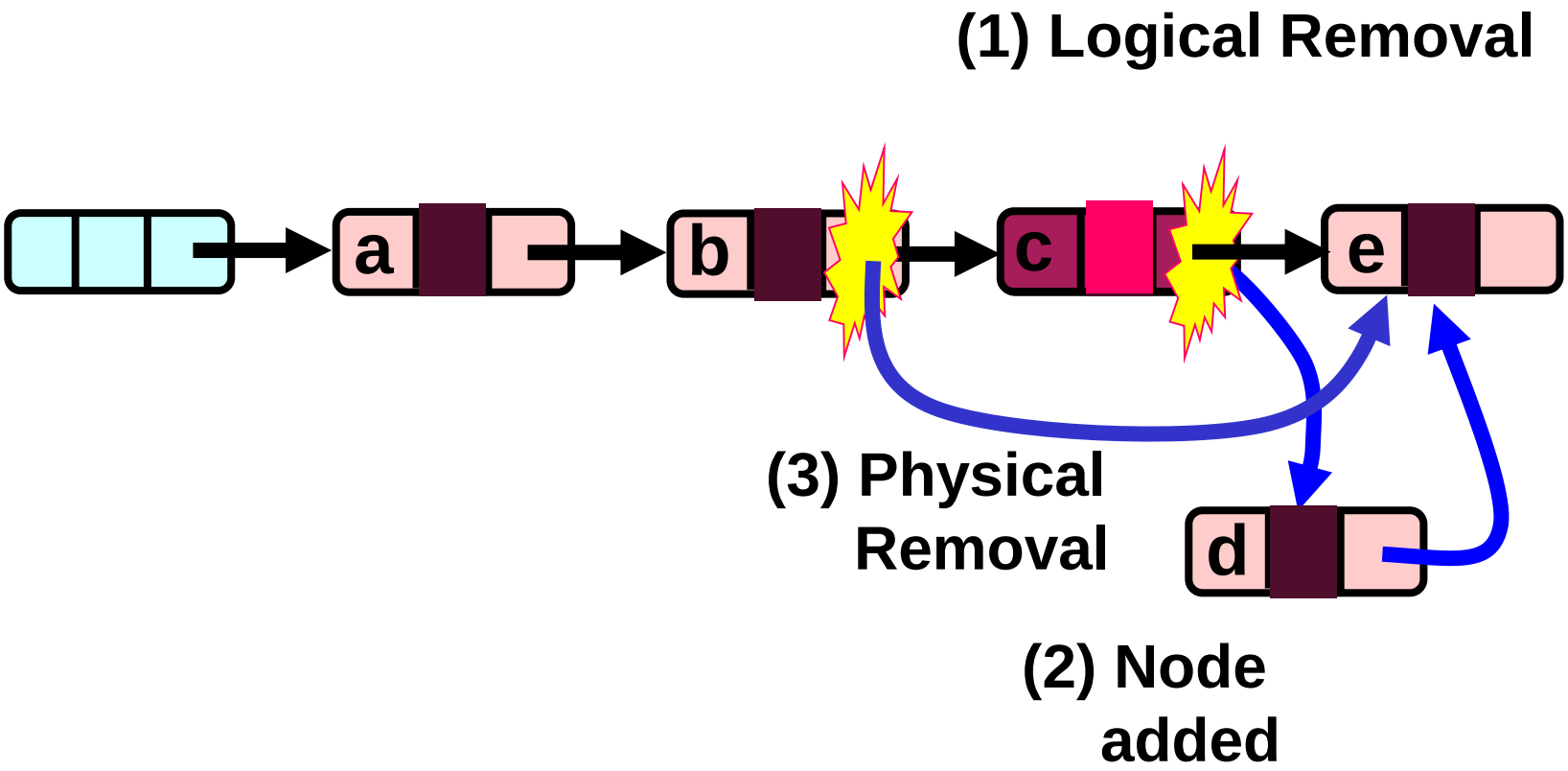


Use CAS to verify pointer is correct

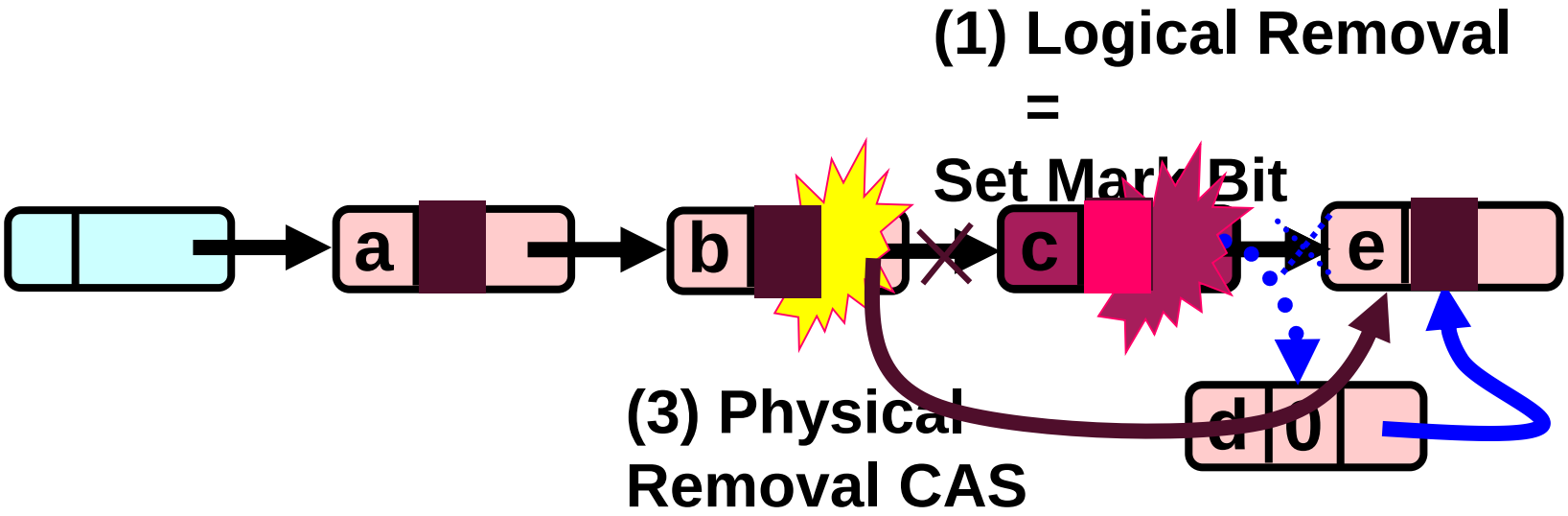
(2) Physical Removal

Not enough! Why?

Problem…



The Solution: Combine Mark and Pointer



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) ☺

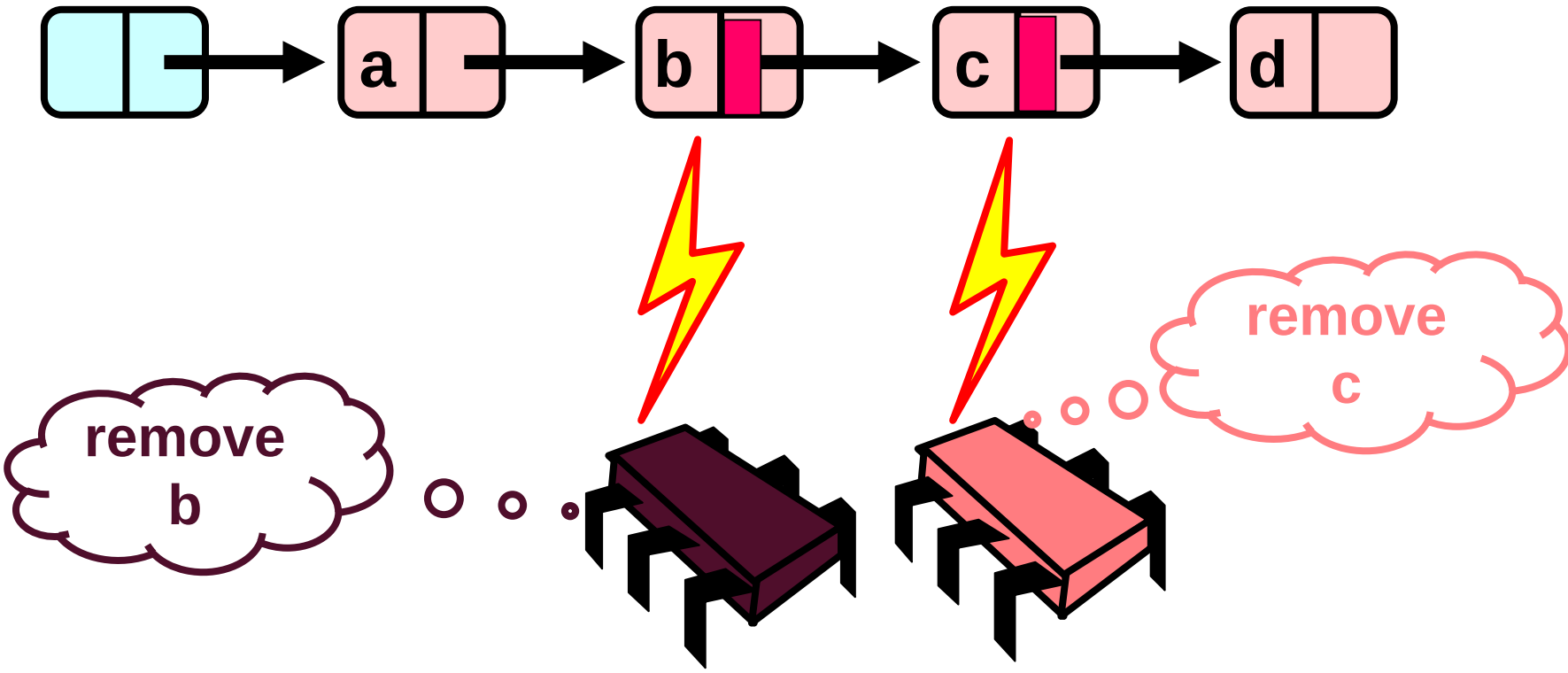
    CAS of two noncontiguous locations
  - Well, not many machines support it ☹

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

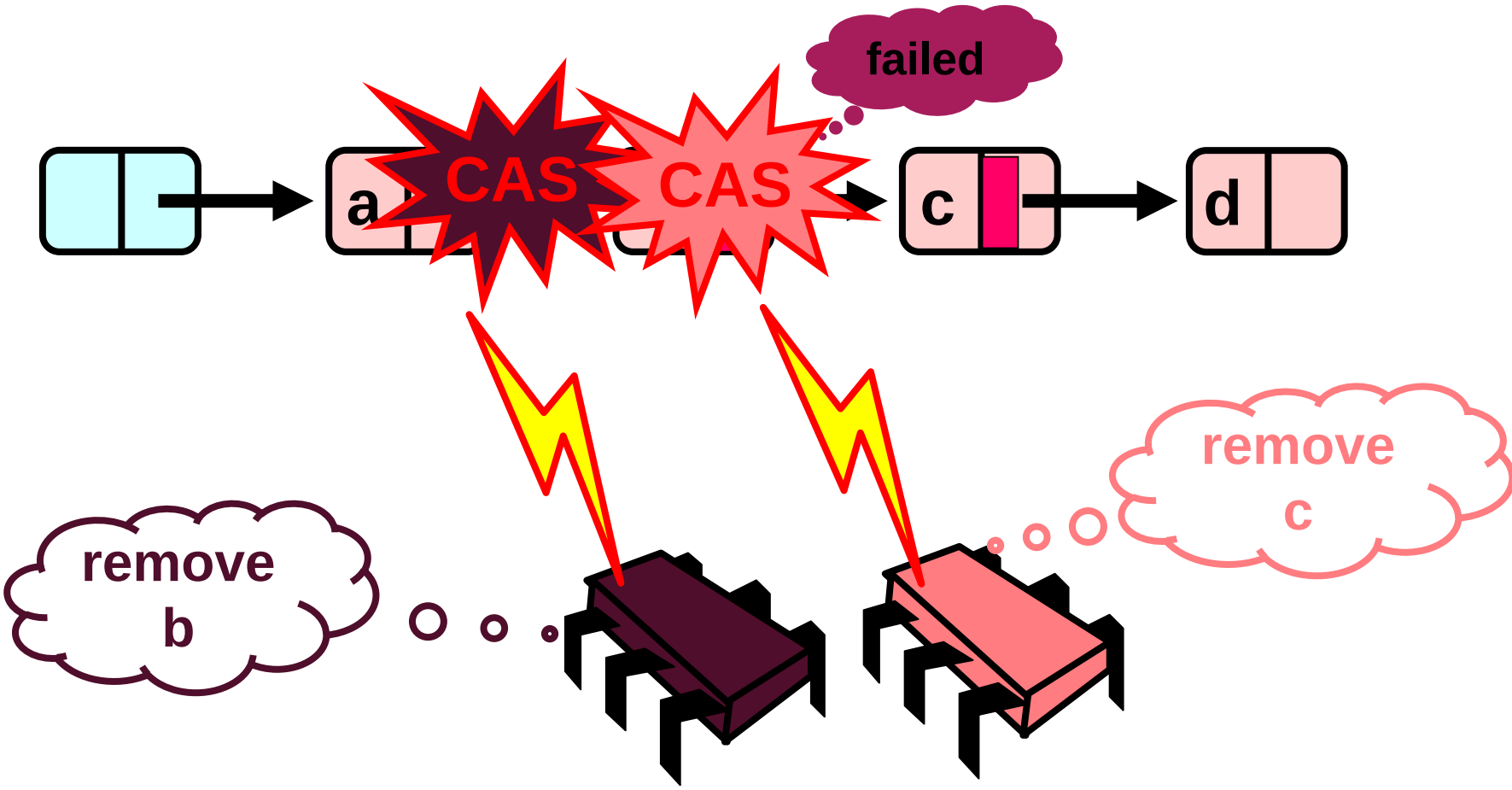
    Contiguous, e.g., lock cmpxchg16b (on 64 bit systems)
- **Option 4:**
  - TM!

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

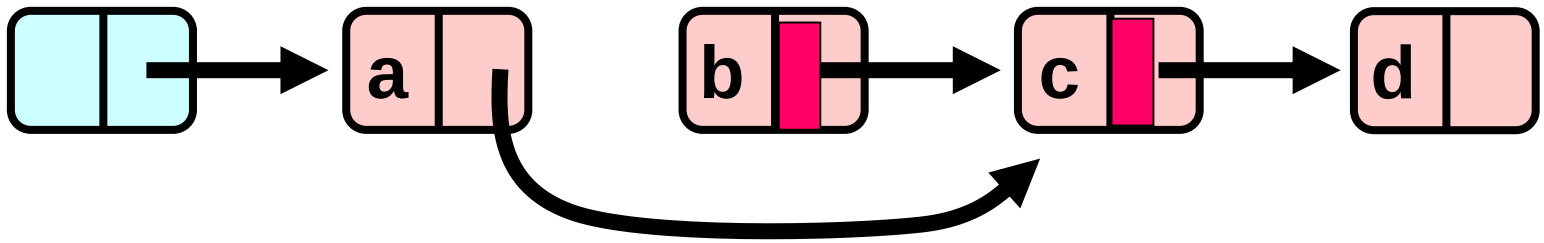
Removing a Node

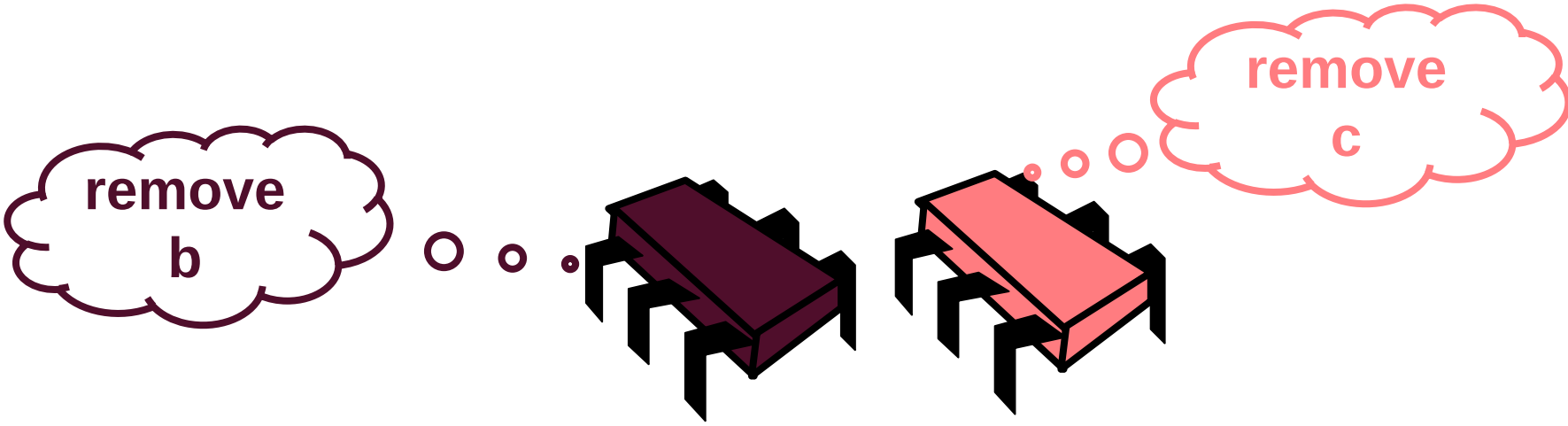


Removing a Node

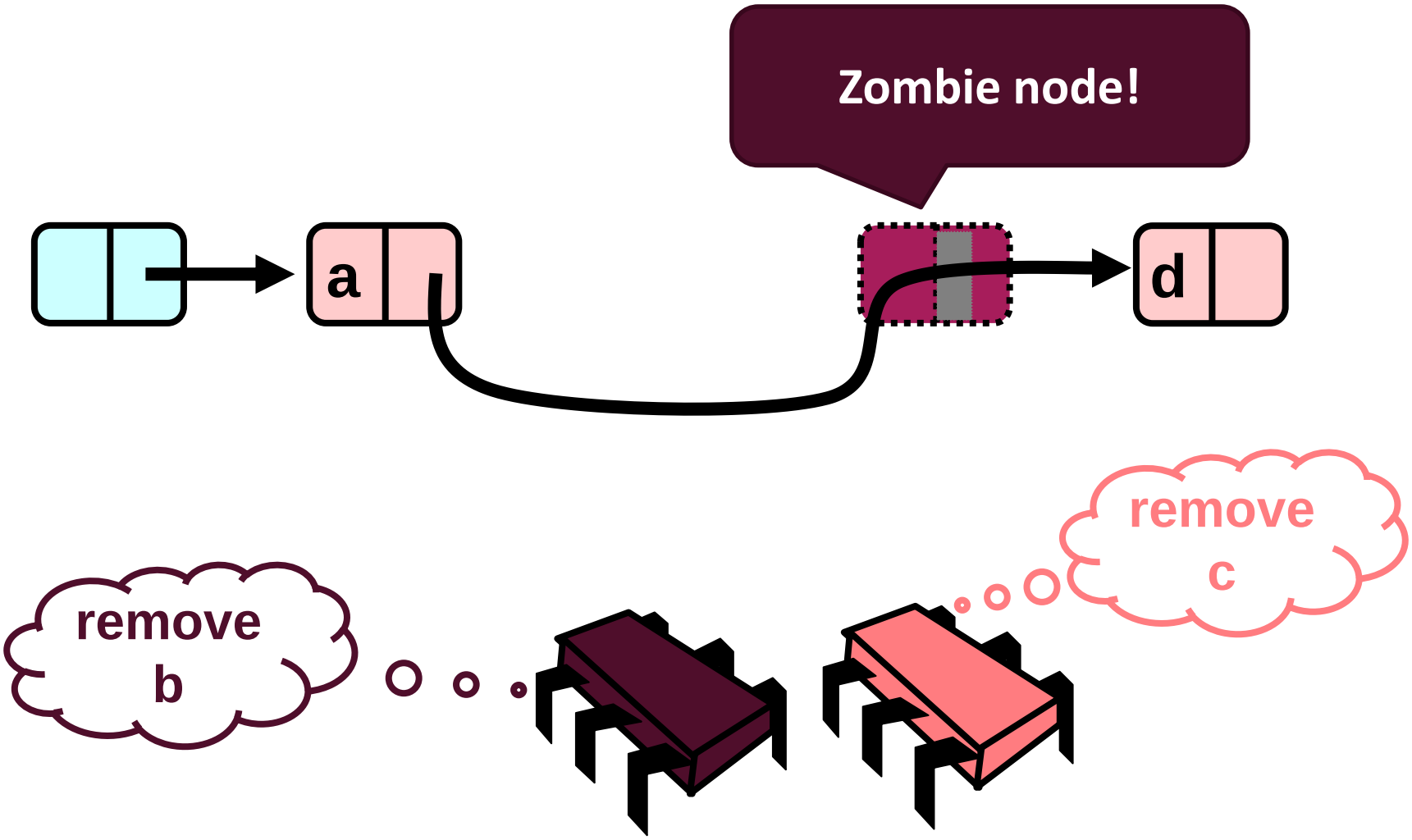


Removing a Node





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"**

Lock-free and wait-free

- **A lock-free method**
  - guarantees that infinitely often **some** method call finishes in a finite number of steps
- **A wait-free method**
  - guarantees that **each** method call finishes in a finite number of steps (implies lock-free)

- **Synchronization instructions are not equally powerful!**
  - Indeed, they form an infinite hierarchy; no instruction (primitive) in level x can be used for lock-/wait-free implementations of primitives in level z>x.

Concept: Consensus Number

- **Each level of the hierarchy has a "consensus number" assigned.**
  - Is the maximum number of threads for which primitives in level x can solve the consensus problem
- **The consensus problem:**
  - Has single function: decide(v)
  - Each thread calls it at most once, the function returns a value that meets two conditions:

    consistency: all threads get the same value

    valid: the value is some thread's input
  - Simplification: binary consensus (inputs in {0,1})

Understanding Consensus

- **Can a particular class solve n-thread consensus wait-free?**
  - A class C solves n-thread consensus if there exists a consensus protocol using **any number** of objects of class C and **any number** of atomic registers
  - The protocol has to be wait-free (bounded number of steps per thread)
  - The consensus number of a class C is the largest n for which that class solves n-thread consensus (may be infinite)
  - Assume we have a class D whose objects can be constructed from objects out of class C. If class C has consensus number n, what does class D have?

Starting simple …

- **Binary consensus with two threads (A, B)!**
  - Each thread moves until it decides on a value
  - May update shared objects
  - Protocol state = state of threads + state of shared objects
  - Initial state = state before any thread moved
  - Final state = state after all threads finished
  - States form a tree, wait-free property guarantees a finite tree

    Example with two threads and two moves each!

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
  - Really?
- **Proof outline:**
  - Assume arbitrary consensus protocol, thread A, B
  - Run until it reaches critical state where next action determines outcome (show that it must have a critical state first)
  - Show all options using atomic registers and show that they cannot be used to determine one outcome for all possible executions!
    1) *Any thread reads (other thread runs solo until end)*
    2) *Threads write to different registers (order doesn't matter)*
    3) *Threads write to same register (solo thread can start after each write)*

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
- **Corollary: It is impossible to construct a wait-free implementation of any object with consensus number of >1 using atomic registers**
  - "perhaps one of the most striking impossibility results in Computer Science" (Herlihy, Shavit)
  - → We need hardware atomics or TM!
- **Proof technique borrowed from:**

Impossibility of **distributed consensus** with one faulty process
MJ Fischer, NA Lynch, MS Paterson - Journal of the ACM (JACM), 1985 - dl.acm.org
Abstract The **consensus** problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of **...**
Cited by 3180 Related articles All 164 versions

- **Very influential paper, always worth a read!**
  - Nicely shows proof techniques that are central to parallel and distributed computing!

Other Atomic Operations

- **Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!**
  - Similar proof technique (bivalence argument)
- **CAS and TM have consensus number ∞**
  - Constructive proof!

Compare and Set/Swap Consensus

```
const int first = -1
volatile int thread = -1;
int proposed[n];

int decide(v) {
  proposed[tid] = v;
  if(CAS(thread, first, tid))
    return  v; // I won!
  else
     return proposed[thread]; // thread won
}
```

- **CAS provides an infinite consensus number**
  - Machines providing CAS are **asynchronous** computation equivalents of the Turing Machine
  - I.e., any concurrent object can be implemented in a wait-free manner (not necessarily fast!)

Now you know everything ☺

- **Not really … ;-)**
  - We'll argue about **performance** now!
- **But you have all the tools for:**
  - Efficient locks
  - Efficient lock-based algorithms
  - Efficient lock-free algorithms (or even wait-free)
  - Reasoning about parallelism!
- **What now?**
  - A different class of problems

    Impact on wait-free/lock-free on actual performance is not well understood
  - Relevant to HPC, applies to shared and distributed memory

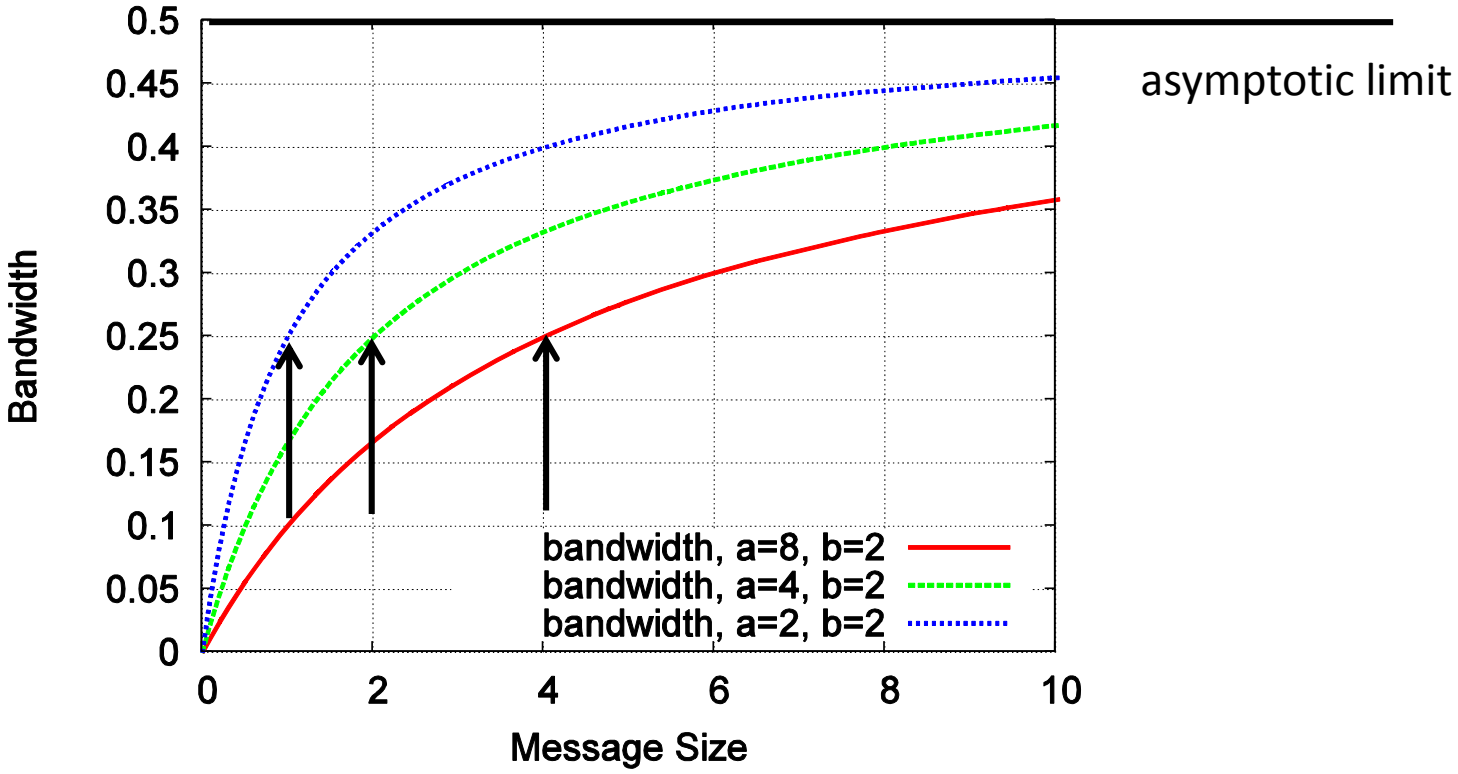
    → *Group communications*

Remember: A Simple Model for Communication

- **Transfer time T(s) = α+βs**
  - α = startup time (latency)
  - β = cost per byte (bandwidth=1/β)
- **As s increases, bandwidth approaches 1/β asymptotically**
  - Convergence rate depends on α
  - $s_{1/2} = \alpha/\beta$
- **Assuming no pipelining (new messages can only be issued from a process after all arrived)**

Bandwidth vs. Latency

- **$s_{1/2} = \alpha/\beta$ often used to distinguish bandwidth- and latency-bound messages**
  - $s_{1/2}$ is in the order of kilobytes on real systems



Quick Example

- **Simplest linear broadcast**
  - One process has a data item to be distributed to all processes
- **Broadcasting s bytes among P processes:**
  - T(s) = (P-1) * (α+βs) = $\mathcal{O}(P)$
- **Class question: Do you know a faster method to accomplish the same?**

k-ary Tree Broadcast

- **Origin process is the root of the tree, passes messages to k neighbors which pass them on**
  - k=2 -> binary tree
- **Class Question: What is the broadcast time in the simple latency/bandwidth model?**
  - $T(s) \approx \lceil log_k(P) \rceil \cdot k \cdot (\alpha + \beta \cdot s) = \mathcal{O}(log(P))$ (for fixed k)
- **Class Question: What is the optimal k?**
  - $0 = \frac{ln(P) \cdot k}{ln(k)} \frac{d}{dk} = \frac{ln(P)ln(k) - ln(P)}{ln^2(k)} \rightarrow k = e = 2.71...$
  - Independent of P, α, βs? Really?

Faster Trees?

- **Class Question: Can we broadcast faster than in a ternary tree?**
  - Yes because each respective root is idle after sending three messages!
  - Those roots could keep sending!
  - Result is a k-nomial tree

    For k=2, it's a binomial tree
- **Class Question: What about the runtime?**
  - $T(s) = \lceil log_k(P) \rceil \cdot (k-1) \cdot (\alpha + \beta \cdot s) = \mathcal{O}(log(P))$
- **Class Question: What is the optimal k here?**
  - T(s) d/dk is monotonically increasing for k>1, thus k_{opt}=2
- **Class Question: Can we broadcast faster than in a k-nomial tree?**
  - $\mathcal{O}(log(P))$ is asymptotically optimal for s=1!
  - But what about large s?

Open Problems

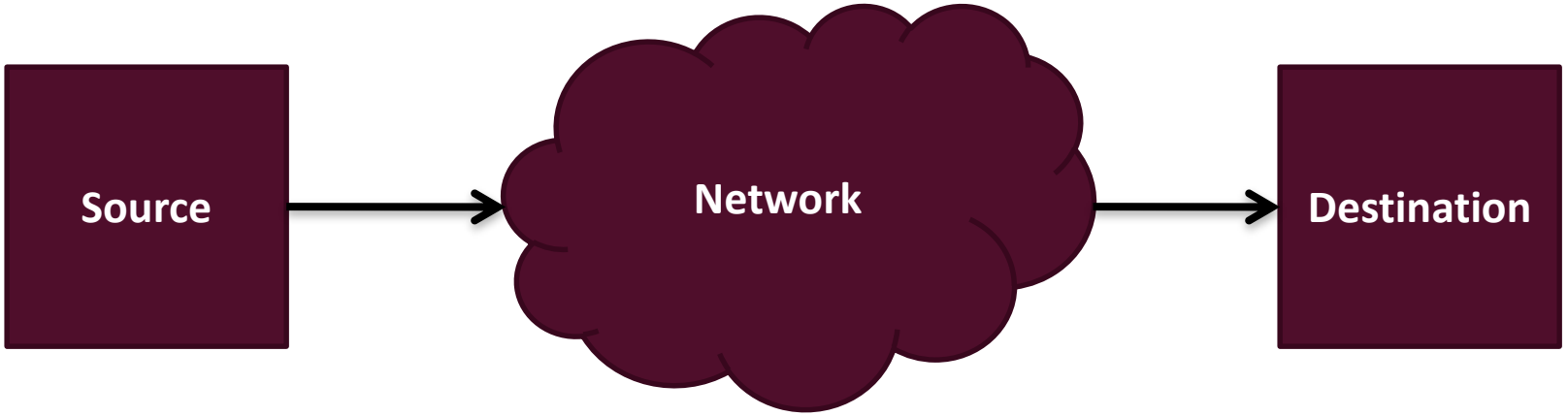
- **Look for optimal parallel algorithms (even in simple models!)**
  - And then check the more realistic models
  - Useful optimization targets are MPI collective operations

    Broadcast/Reduce, Scatter/Gather, Alltoall, Allreduce, Allgather, Scan/Exscan, …
  - Implementations of those (check current MPI libraries ☺)
  - Useful also in scientific computations

    Barnes Hut, linear algebra, FFT, …
- **Lots of work to do!**
  - Contact me for thesis ideas (or check SPCL) if you like this topic
  - Usually involve optimization (ILP/LP) and clever algorithms (algebra) combined with practical experiments on large-scale machines (10,000+ processors)

HPC Networking Basics

- **Familiar (non-HPC) network: Internet TCP/IP**
  - Common model:

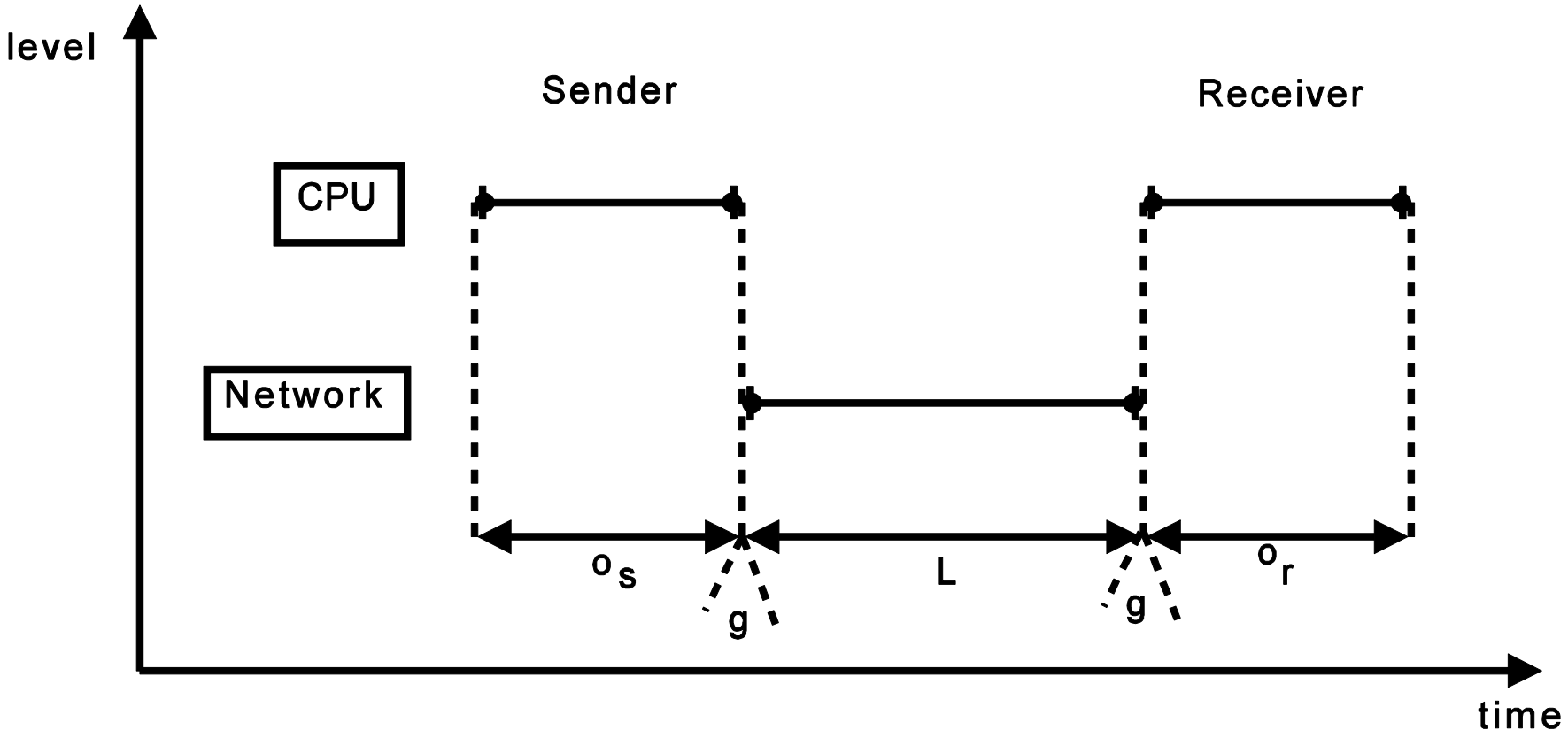


- **Class Question: What parameters are needed to model the performance (including pipelining)?**
  - Latency, Bandwidth, Injection Rate, Host Overhead

The LogP Model

- **Defined by four parameters:**
  - L: an upper bound on the latency, or delay, incurred in communicating a message containing a word (or small number of words) from its source module to its target module.
  - o: the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.
  - g: the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth.
  - P: the number of processor/memory modules. We assume unit time for local operations and call it a cycle.

The LogP Model



Simple Examples

- **Sending a single message**
  - $T = 2o+L$
- **Ping-Pong Round-Trip**
  - $T_{RTT} = 4o+2L$
- **Transmitting n messages**
  - $T(n) = L+(n-1)*\max(g, o) + 2o$

Simplifications

- **o is bigger than g on some machines**
  - g can be ignored (eliminates max() terms)
  - be careful with multicore!
- **Offloading networks might have very low o**
  - Can be ignored (not yet but hopefully soon)
- **L might be ignored for long message streams**
  - If they are pipelined
- **Account g also for the first message**
  - Eliminates “-1”

Benefits over Latency/Bandwidth Model

- **Models pipelining**
  - L/g messages can be “in flight”
  - Captures state of the art (cf. TCP windows)
- **Models computation/communication overlap**
  - Asynchronous algorithms
- **Models endpoint congestion/overload**
  - Benefits balanced algorithms

Example: Broadcasts

- **Class Question: What is the LogP running time for a linear broadcast of a single packet?**
  - $T_{lin} = L + (P-2) * \max(o,g) + 2o$
- **Class Question: Approximate the LogP runtime for a binary-tree broadcast of a single packet?**
  - $T_{bin} \leq \log_2 P * (L + \max(o,g) + 2o)$
- **Class Question: Approximate the LogP runtime for an k-ary-tree broadcast of a single packet?**
  - $T_{k\text{-}n} \leq \log_k P * (L + (k-1)\max(o,g) + 2o)$

Example: Broadcasts

- **Class Question: Approximate the LogP runtime for a binomial tree broadcast of a single packet (assume L > g!)?**
  - $T_{bin} \leq \log_2 P * (L + 2o)$
- **Class Question: Approximate the LogP runtime for a k-nomial tree broadcast of a single packet?**
  - $T_{k\text{-}n} \leq \log_k P * (L + (k\text{-}2)\max(o,g) + 2o)$
- **Class Question: What is the optimal k (assume o>g)?**
  - Derive by k: $0 = o * \ln(k_{opt}) - L/k_{opt} + o$ (solve numerically)

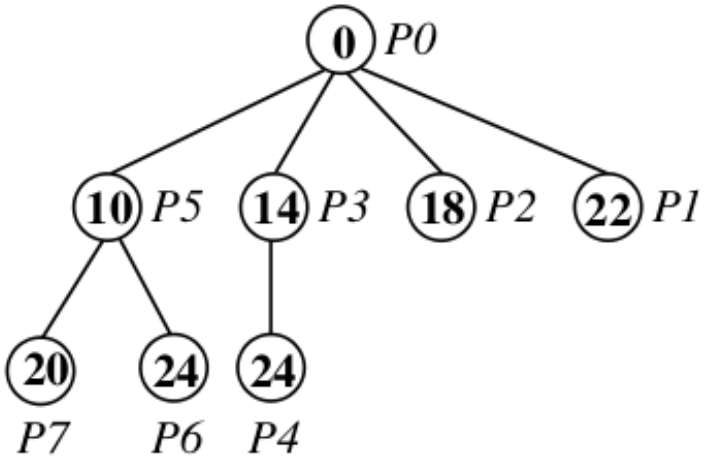
    For larger L, k grows and for larger o, k shrinks
  - Models pipelining capability better than simple model!

Example: Broadcasts

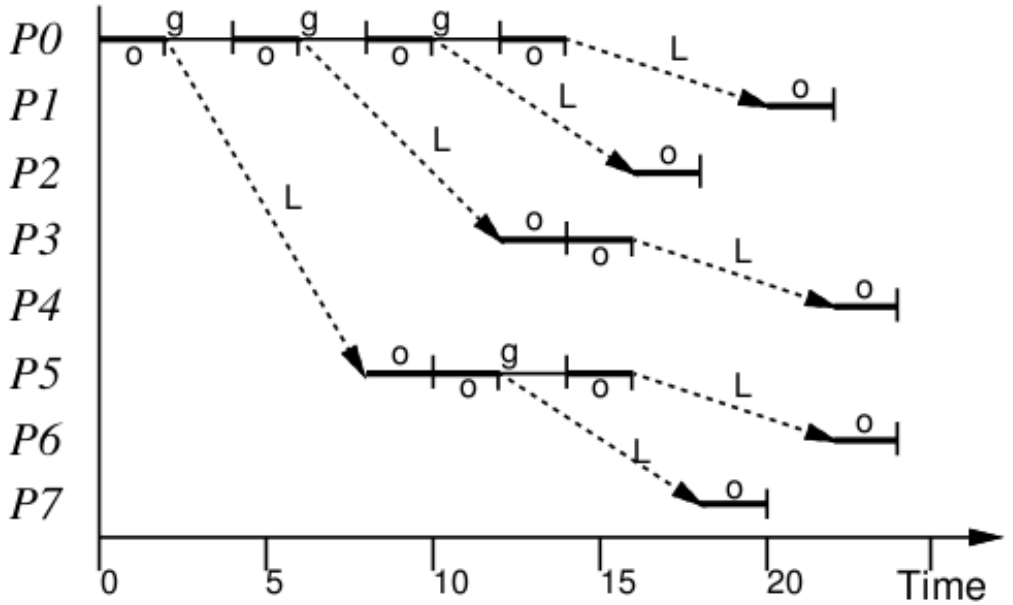
- **Class Question: Can we do better than k_{opt}-ary binomial broadcast?**
  - Problem: fixed k in all stages might not be optimal
  - We can construct a schedule for the optimal broadcast in practical settings
  - First proposed by Karp et al. in “Optimal Broadcast and Summation in the LogP Model”

Example: Optimal Broadcast

- **Broadcast to P-1 processes**
  - Each process who received the value sends it on; each process receives exactly once



P=8, L=6, g=4, o=2



Optimal Broadcast Runtime

- **This determines the maximum number of PEs (P(t)) that can be reached in time t**
- **P(t) can be computed with a generalized Fibonacci recurrence (assuming o>g):**

$$P(t) = \begin{cases} 1: & t < 2o + L \\ P(t-o) + P(t-L-2o): & \text{otherwise.} \end{cases} \quad (1)$$

- **Which can be bounded by (see [1]):** $2^{\left\lfloor \frac{t}{L+2o} \right\rfloor} \leq P(t) \leq 2^{\left\lfloor \frac{t}{o} \right\rfloor}$
  - A closed solution is an interesting open problem!

[1]: Hoefler et al.: "Scalable Communication Protocols for Dynamic Sparse Data Exchange" (Lemma 1)

The Bigger Picture

- **We learned how to program shared memory systems**
  - Coherency & memory models & linearizability
  - Locks as examples for reasoning about correctness and performance
  - List-based sets as examples for lock-free and wait-free algorithms
  - Consensus number
- **We learned about general performance properties and parallelism**
  - Amdahl's and Gustafson's laws
  - Little's law, Work-span, …
  - Balance principles & scheduling
- **We learned how to perform model-based optimizations**
  - Distributed memory broadcast example with two models
- **What next? MPI? OpenMP? UPC?**
  - Next-generation machines "merge" shared and distributed memory concepts → Partitioned Global Address Space (PGAS)