DD2451
Parallel and Distributed Computing

FDD3008
Distributed Algorithms

Lecture 6

Linked Lists

Mads Dam
Autumn/Winter 2011

Slides: Much material due to M. Herlihy

Last Lecture

- Spin locks
- Using RMW instructions, CAS etc.
- Memory contention, cache effects
- Exponential backoff
- Queue locks

- But: Scalable locks =/=> scalable objects
  - If anybody thought that ;-)

Today

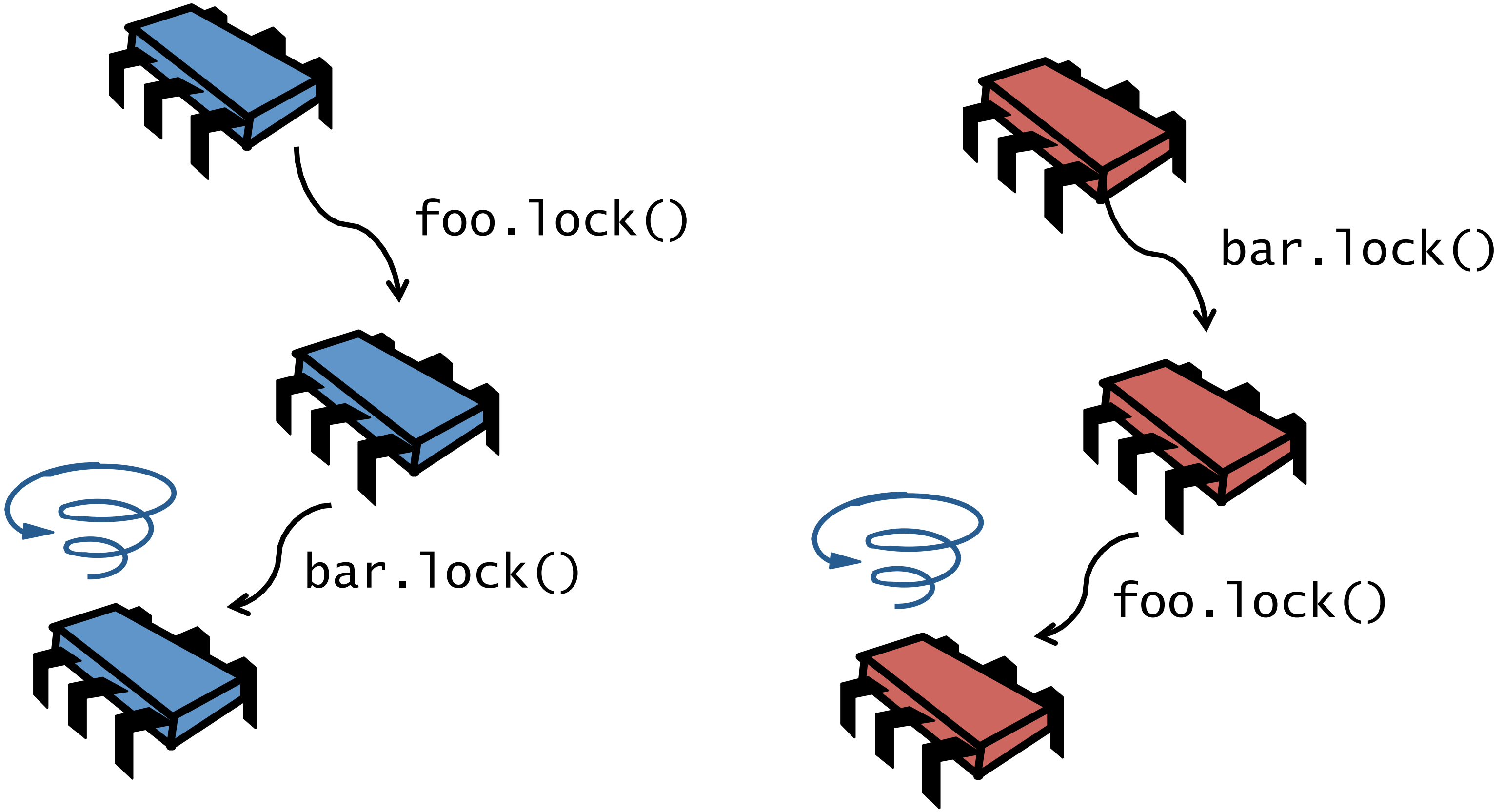
- Start looking at concurrent data structures

- Adding threads should not reduce throughput
  - Contention
  - Mostly fixed by queue locks

- Adding threads should increase throughput
  - Not always possible
  - Surprising things are parallelizable

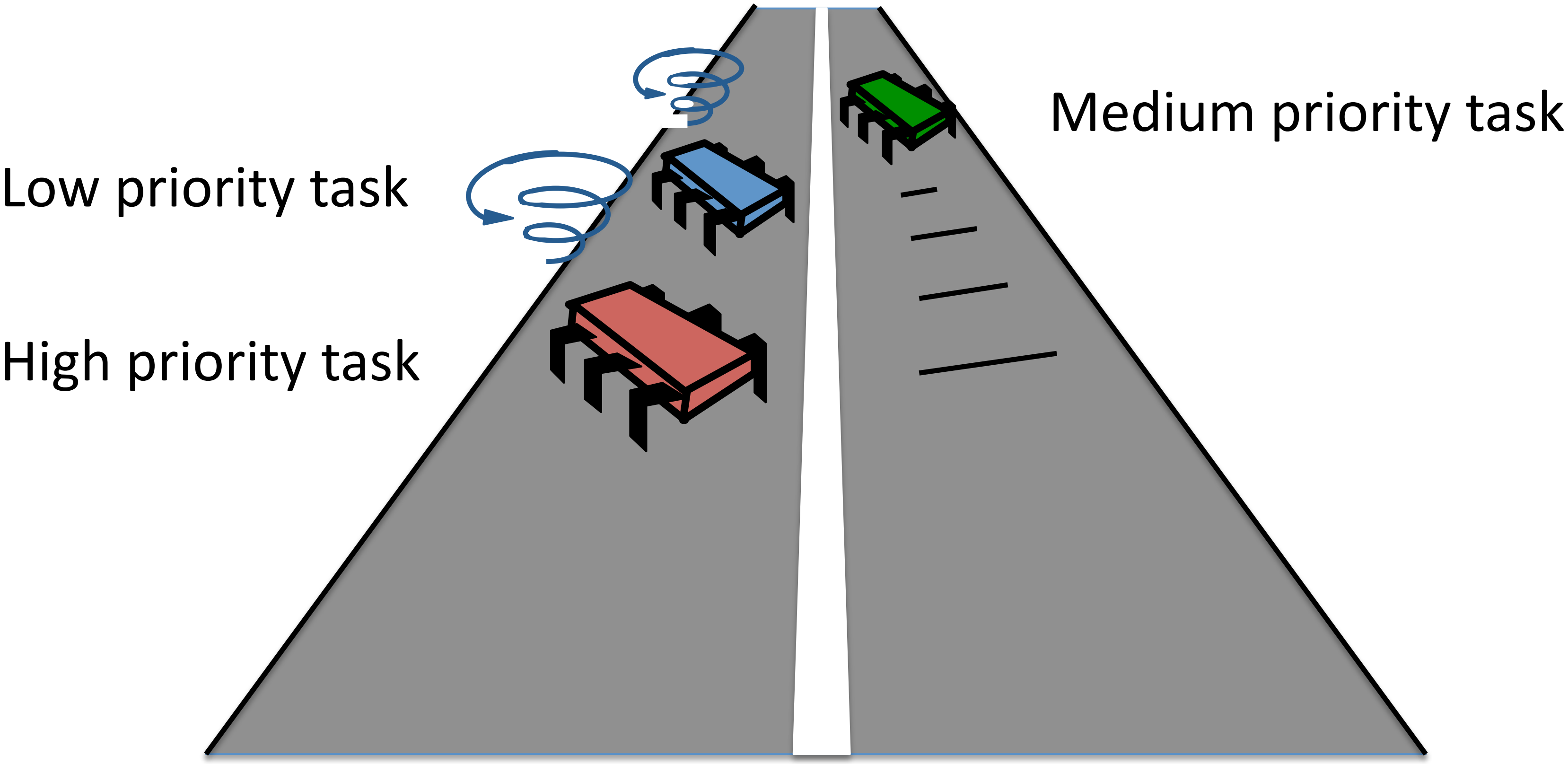
What Is the Problem with Locks?

- Careless use of locks may cause deadlocks



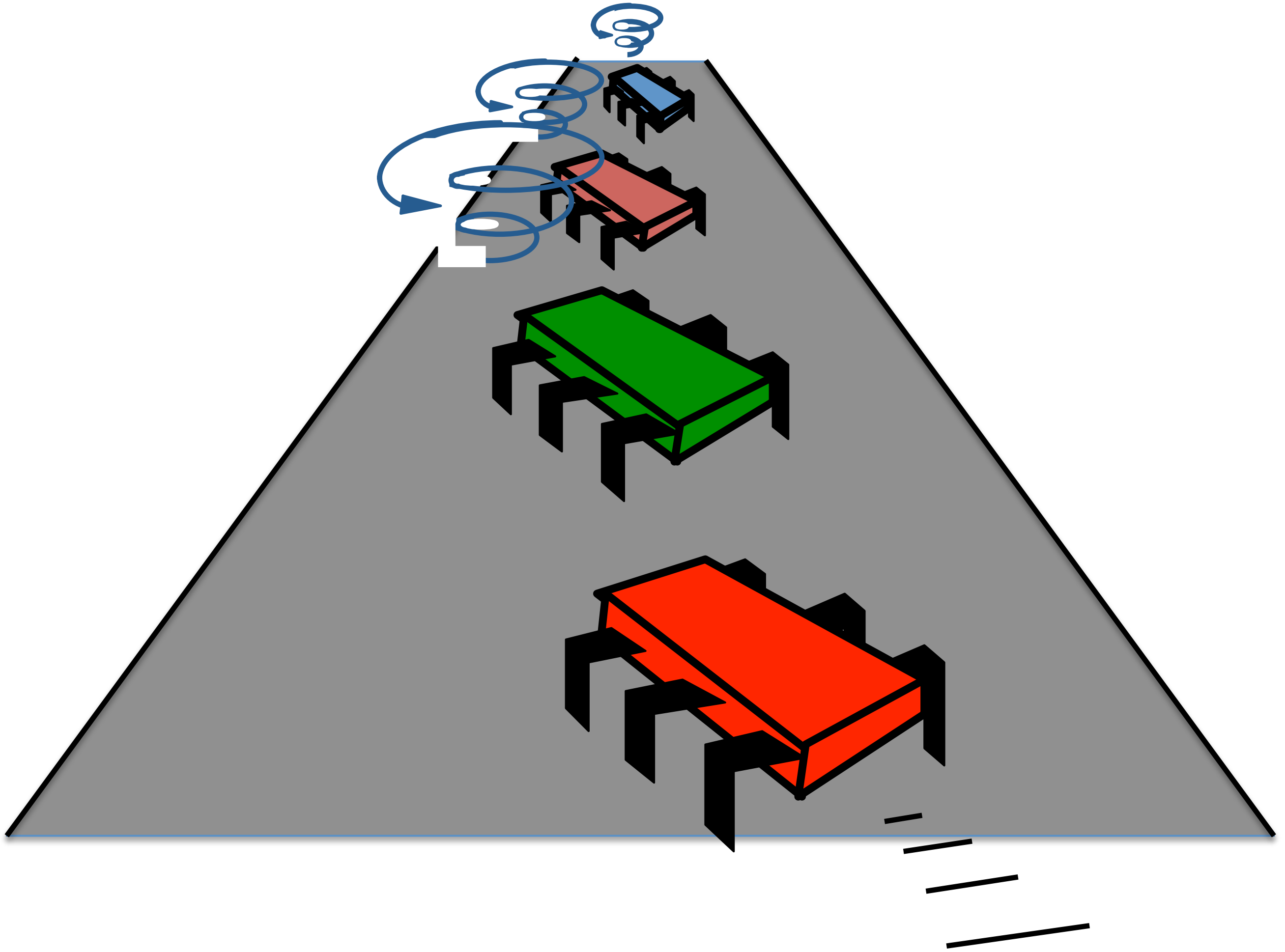
Priority Inversion

- High priority threads pile up behind low priority threads
- Less important threads may get to go first



Convoying

- Fast threads pile up behind slow ones and cause congestion



Coarse-grained Synchronization

- Each method locks the object
  - Avoid contention using queue locks
  - Mostly easy to reason about
  - This is the standard Java model (**synchronized** blocks and methods)

- Problem: Sequential bottleneck
  - Threads "stand in line"
  - Adding more threads does not improve throughput
  - We even struggle to keep it from getting worse…

- So why do we even use a multiprocessor?
  - Well, some applications are inherently parallel…

Exploiting Parallelism

- We will now talk about four "patterns"
  - Bag of tricks …
  - Methods that work more than once …

- For highly-concurrent objects
  - Concurrent access
  - More threads, more throughput

Pattern #1: Fine-Grained Synchronization

- Instead of using a single lock …

- Split object into
  - Independently-synchronized components

- Methods conflict when they access
  - The same component …
  - At the same time

- But one method may still block another
  - Even if they access disjoint parts of the data structure!

Pattern #2: Optimistic Synchronization

- Search without locking …
- If you find it, lock and check …
  - OK: we are done
  - Oops: start over
- Evaluation
  - Usually cheaper than locking, but
  - Mistakes are expensive

Pattern 3: Lazy Synchronization

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

Pattern 4: Lock-free Synchronization

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

Concurrent Linked Lists

- In the following, we will illustrate these patterns using a list-based set
  - Common application
  - Building block for other apps

- A set is an collection of items
  - No duplicates

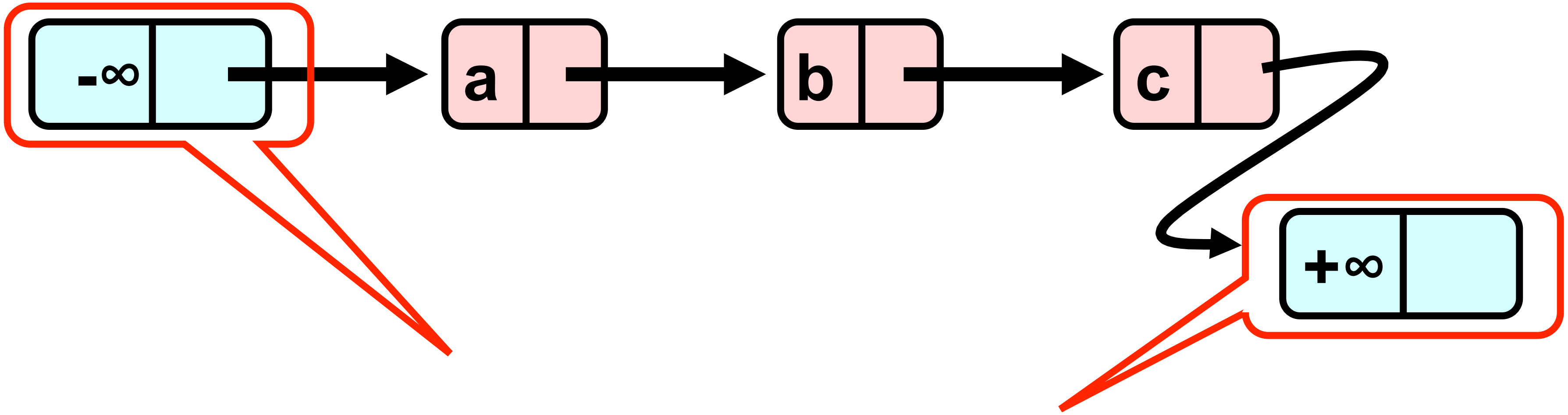
- The operations that we want to allow on the set are
  - `add(x)` puts x into the set
  - `remove(x)` takes x out of the set
  - `contains(x)` tests if x is in the set

Lists and List Nodes

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

List-Based Set



Sorted with Sentinel nodes
(min & max possible keys)

Reasoning about Concurrent Objects

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

- **Not** sufficient to consider only calls and returns!
- Because method bodies may interfere

Specifically …

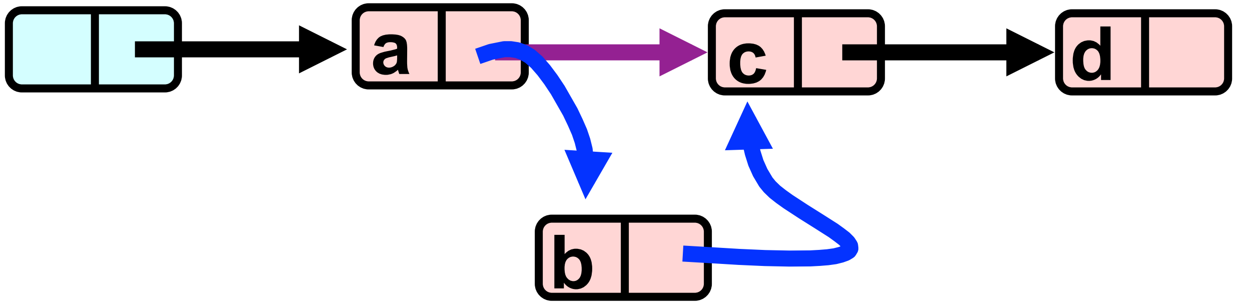
- Invariants preserved by
  - **`add()`**
  - **`remove()`**
  - **`contains()`**
- Most steps are trivial
  - Usually one step tricky
  - Often linearization point
- Representation invariant here:
  - Sentinel nodes:
    - tail reachable from head
  - List is sorted
  - No duplicates

Interference

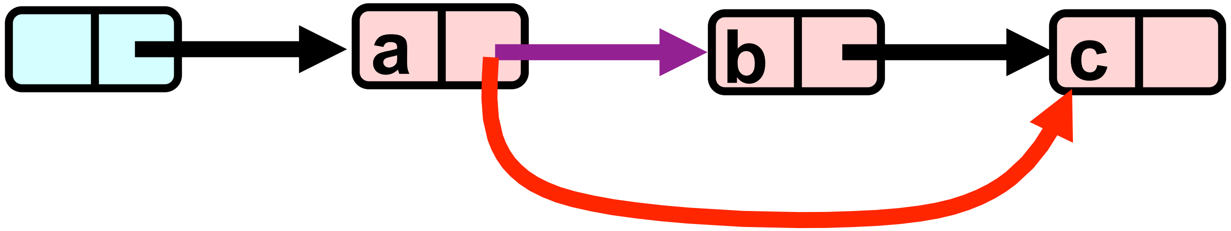
- Invariants make sense only if
  - methods considered
  - are the only modifiers
- Language encapsulation helps
  - List nodes not visible outside class
- Freedom from interference needed even for removed nodes
  - Some algorithms traverse removed nodes
  - Careful with **malloc()** & **free()** !
- Garbage collection helps here

Sequential List-Based Set

- Add:

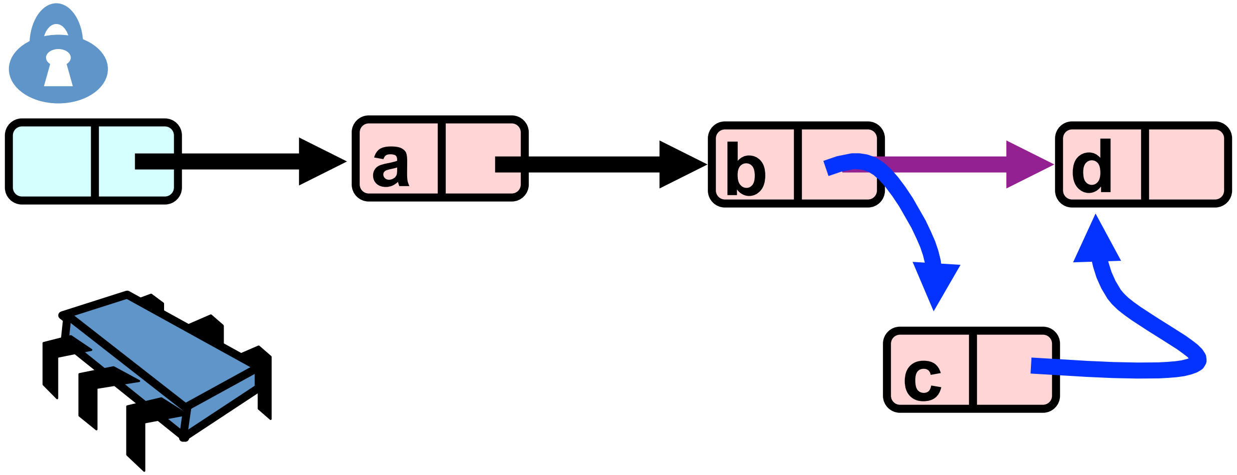


- Remove:



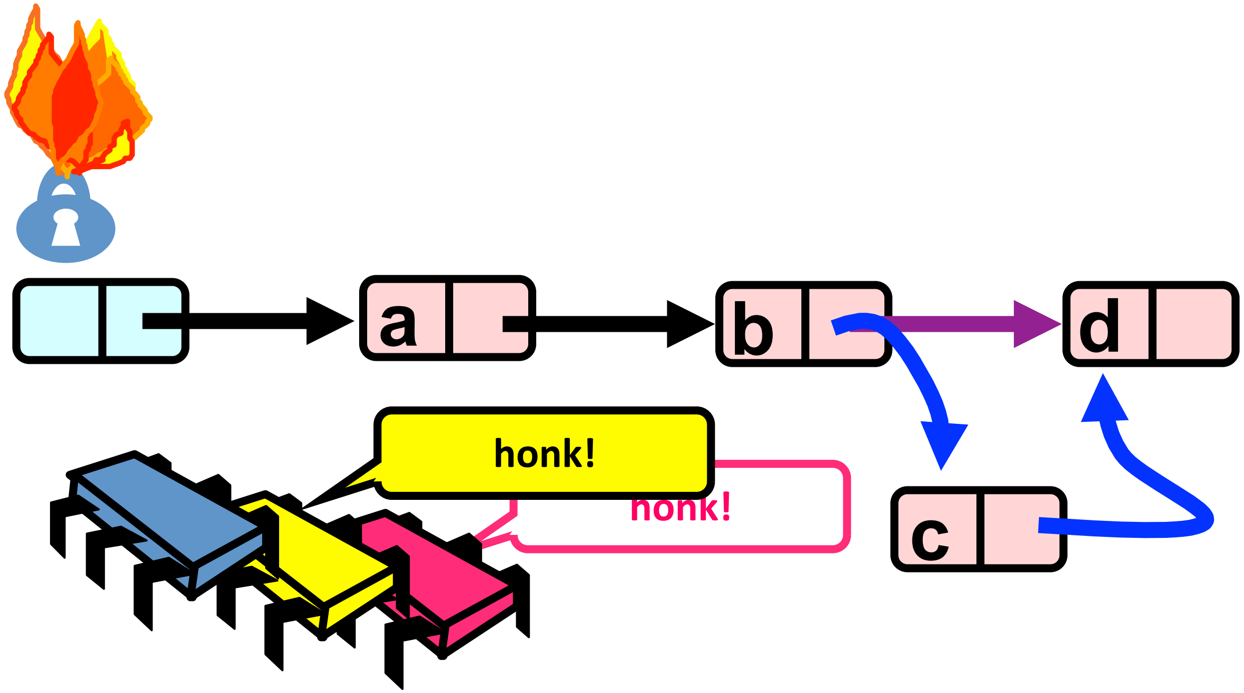
Coarse-Grained Locking

- Lock the sentinel node



- Same as with synchronized methods
- Simple and clearly correct
- Not to be dismissed too lightly

Coarse-Grained Locking

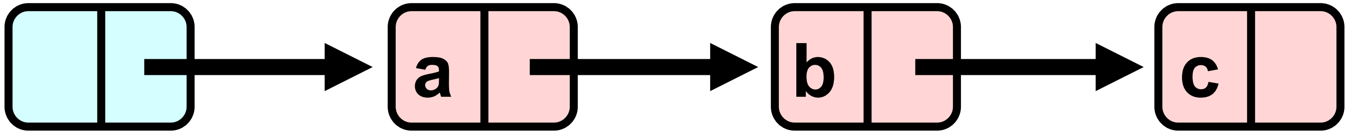


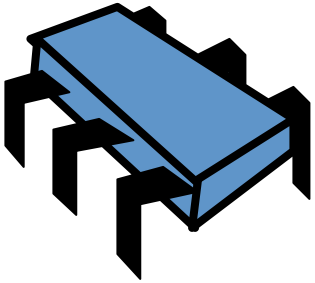
Hotspot + bottleneck

Fine-Grained Locking

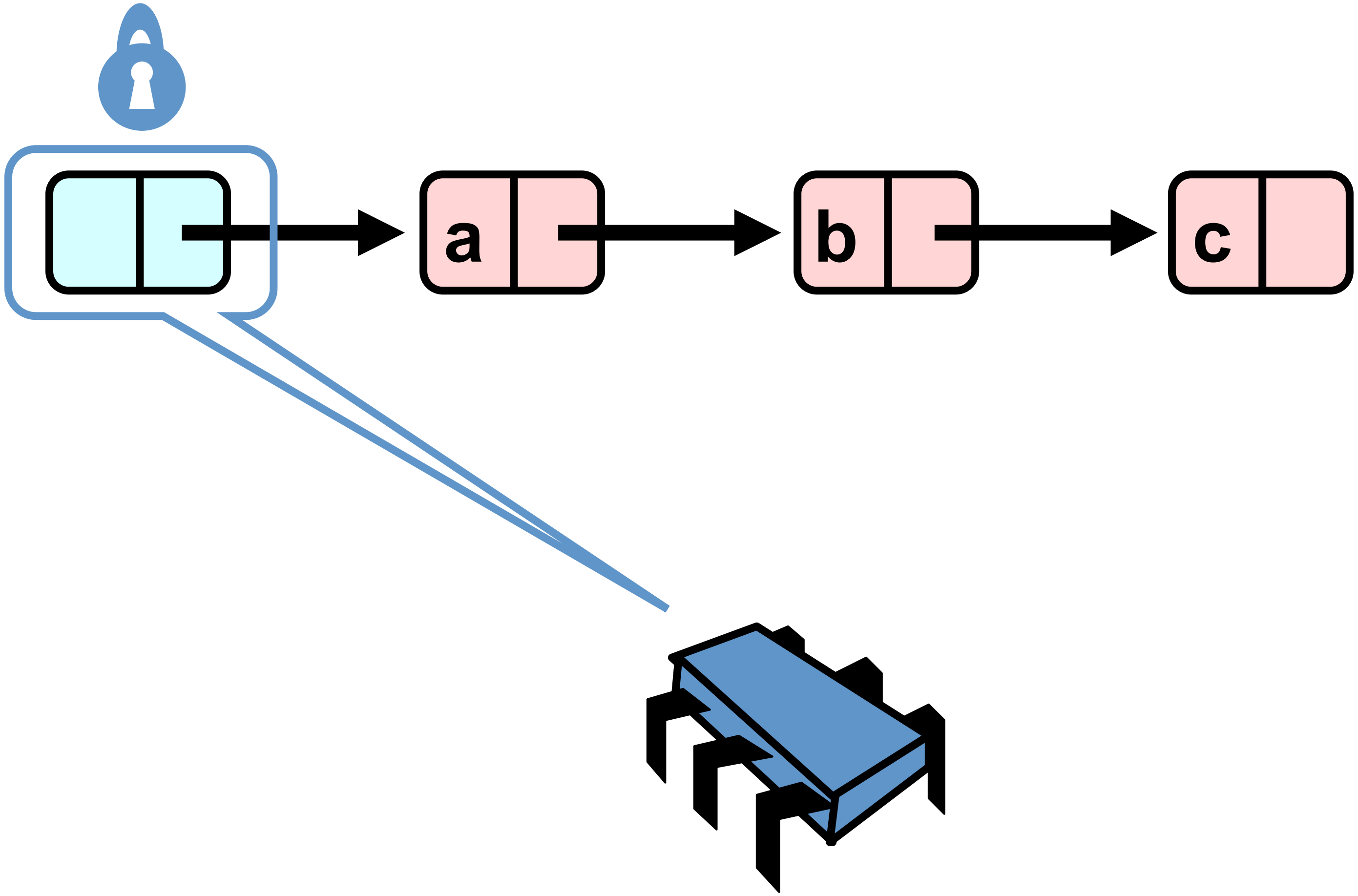
- Requires **careful** thought
  - “Do not meddle in the affairs of wizards, for they are subtle and quick to anger”
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other
- Allows list operations to be pipelined

Hand-over-Hand locking

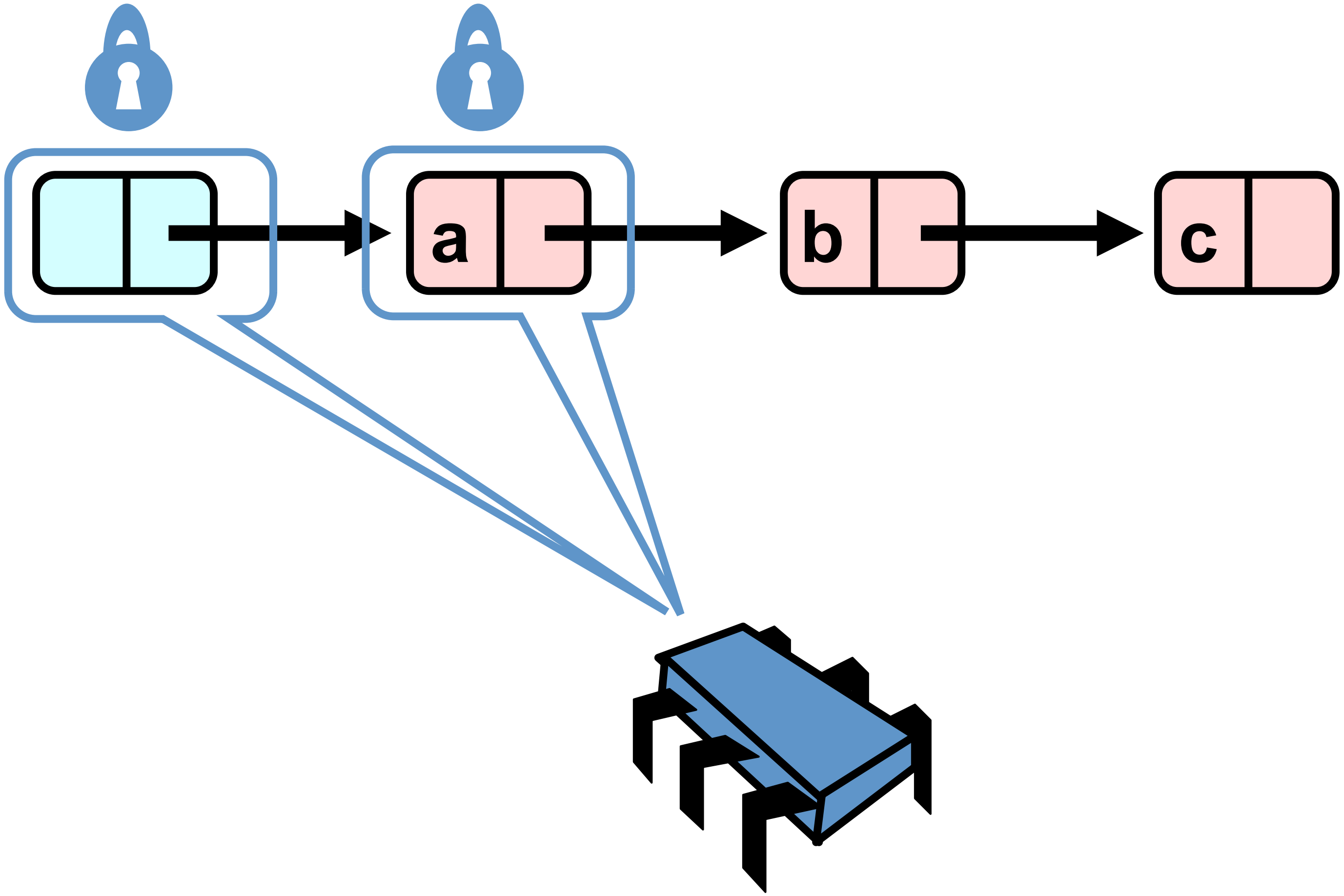




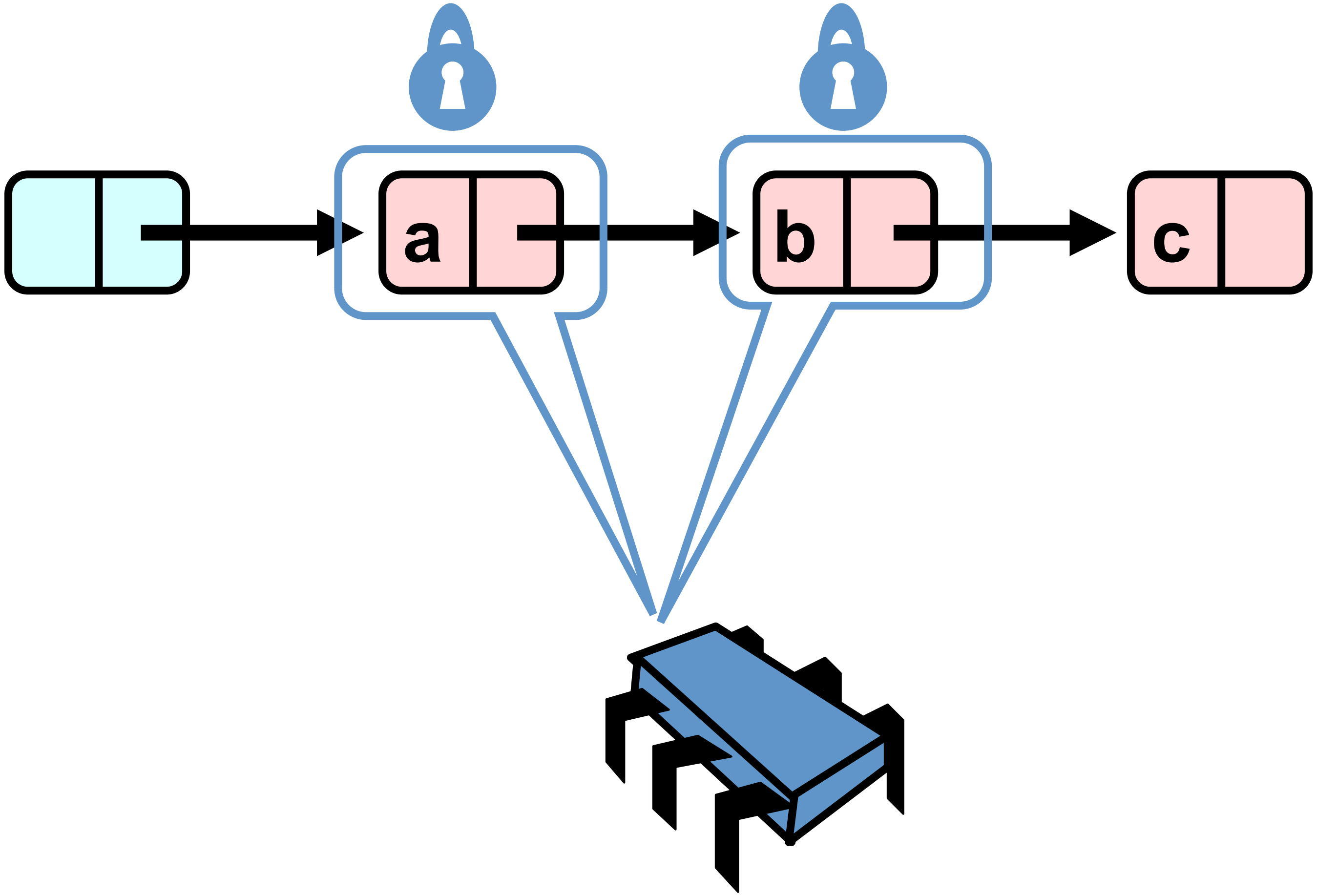
Hand-over-Hand locking



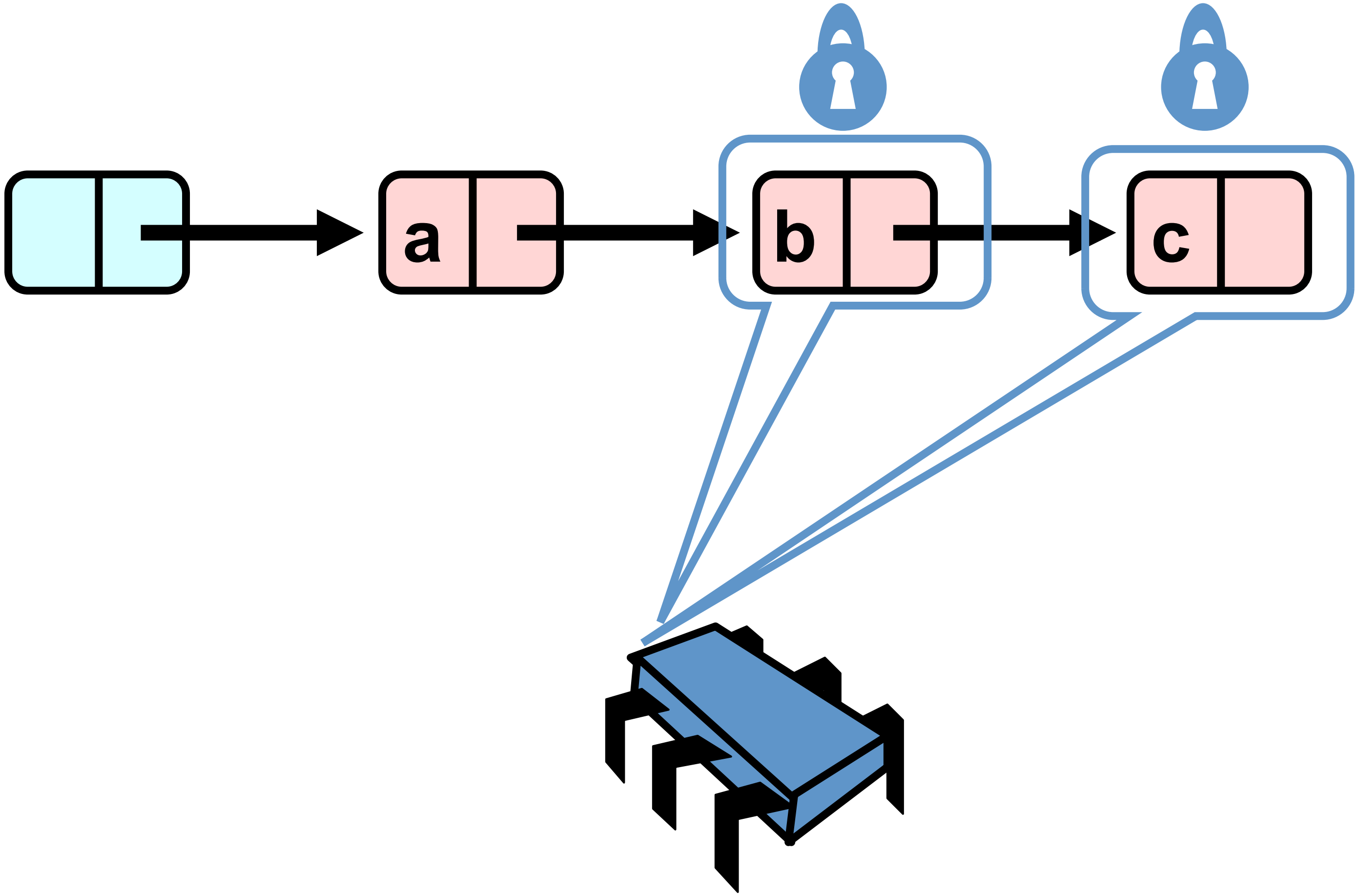
Hand-over-Hand locking



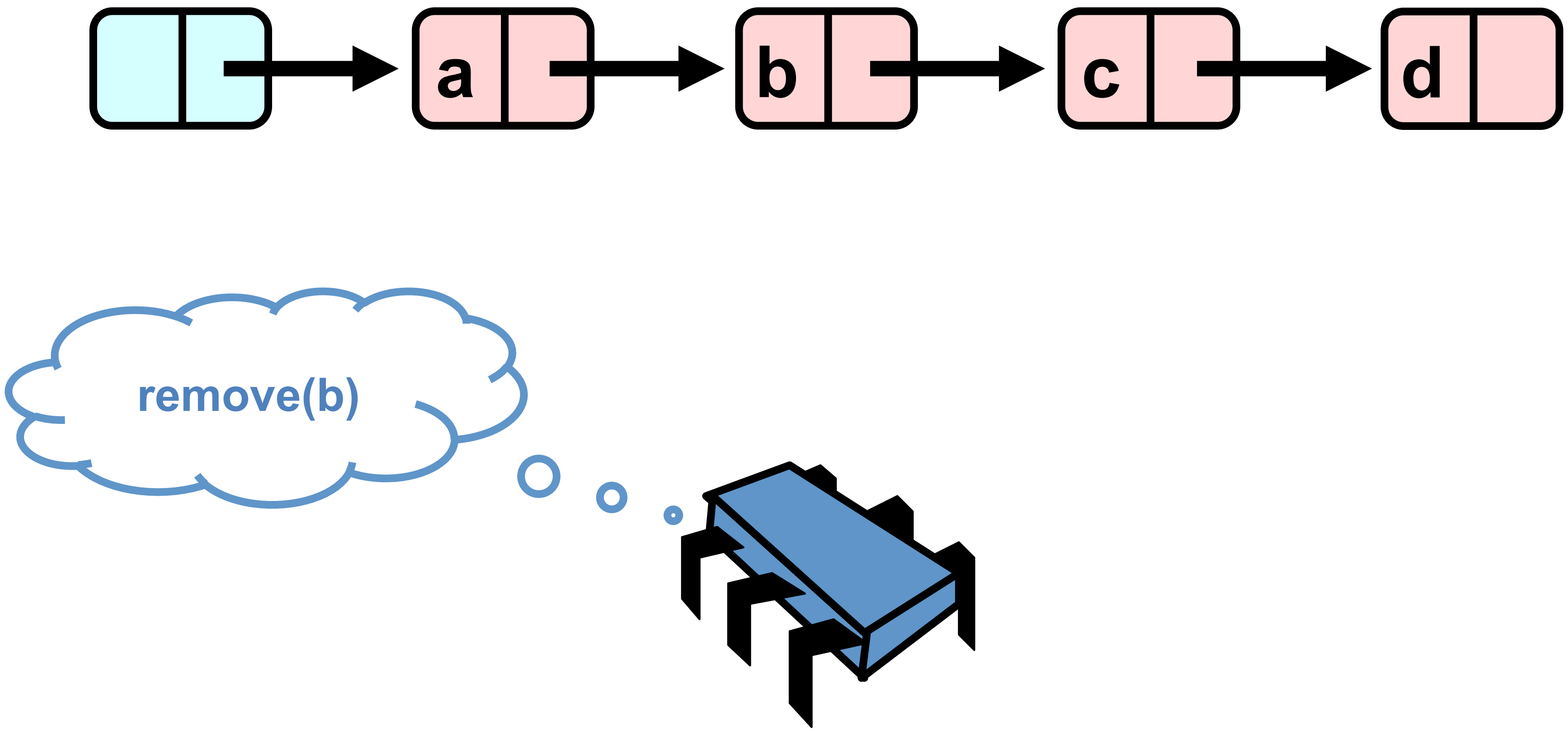
Hand-over-Hand locking



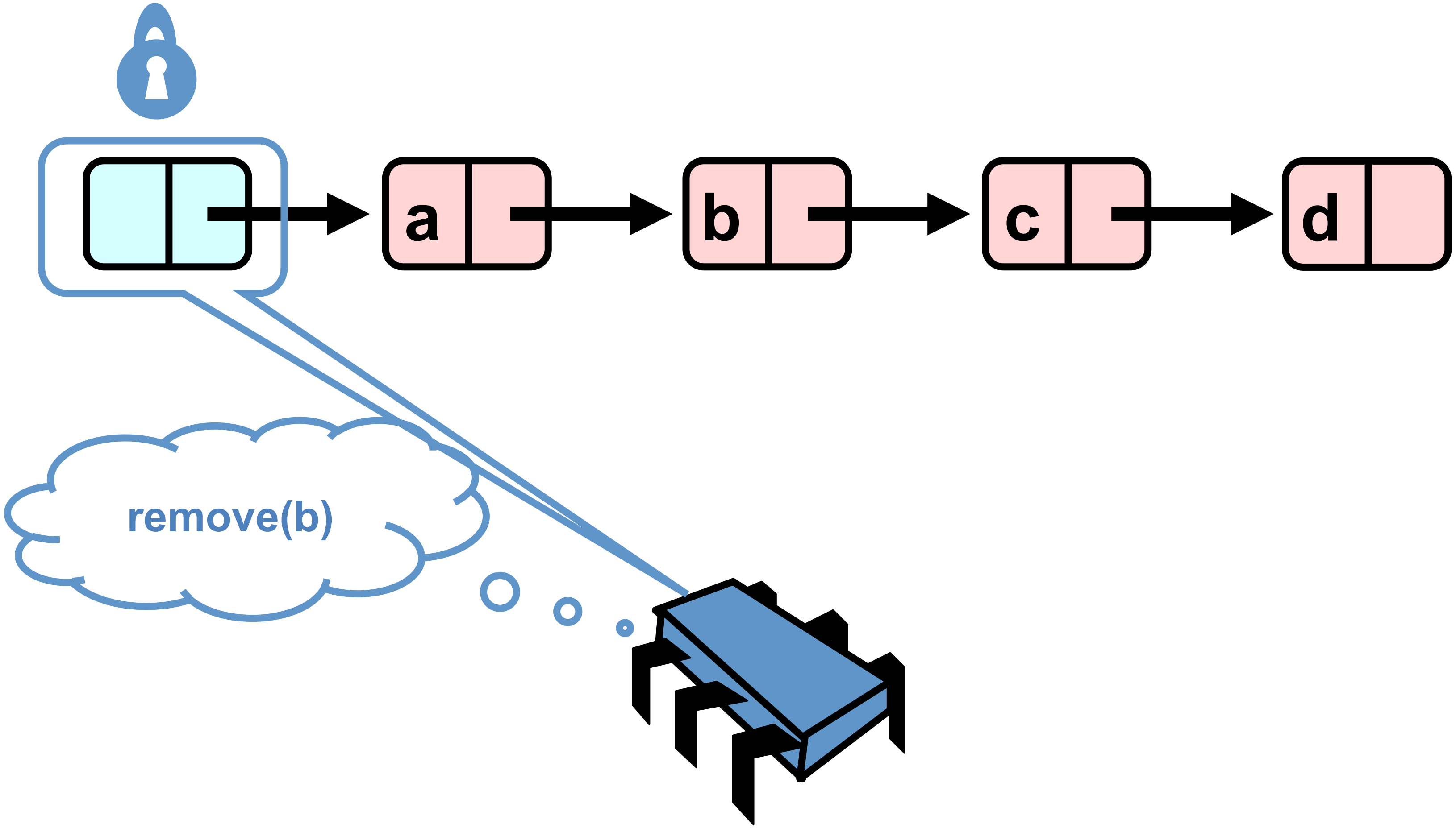
Hand-over-Hand locking



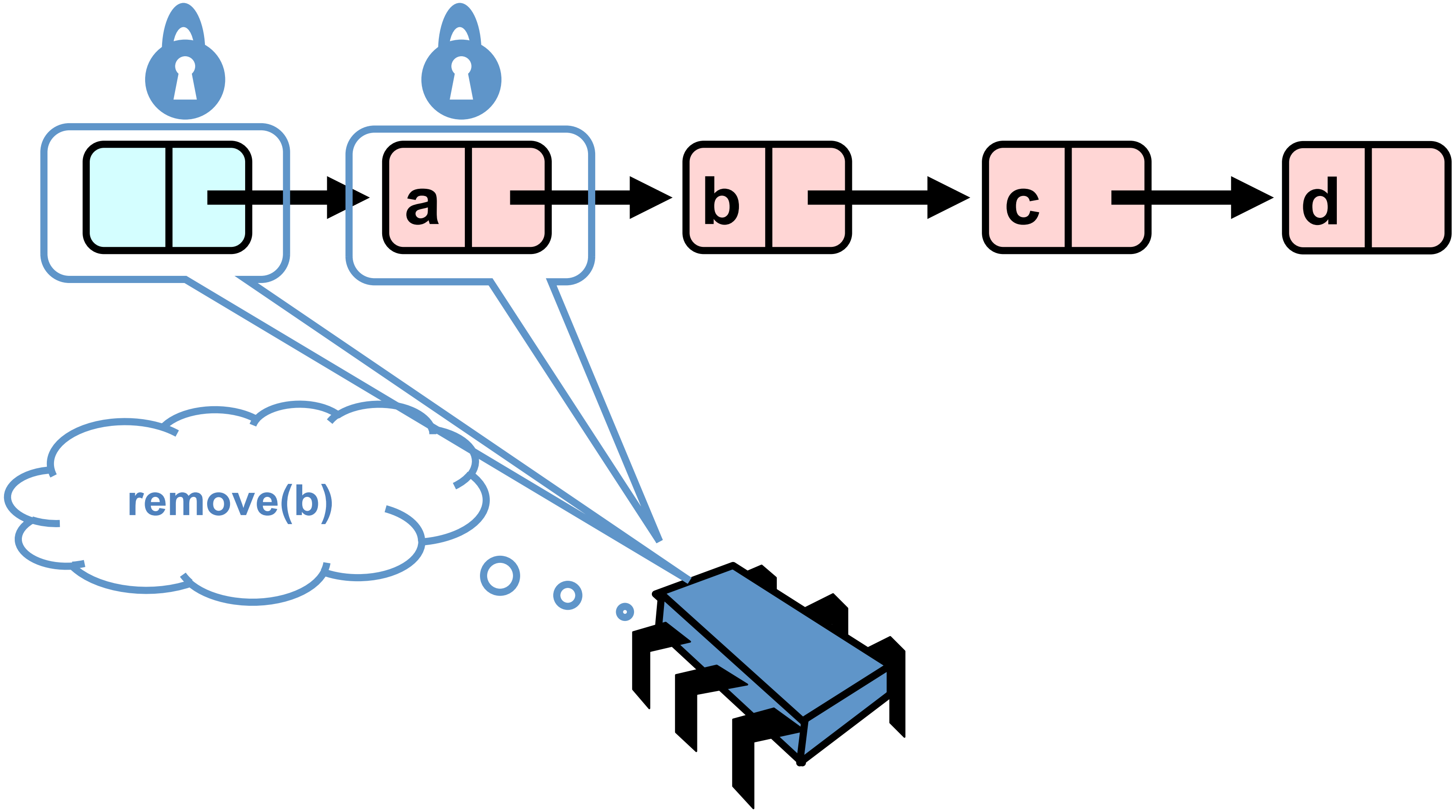
Removing a Node



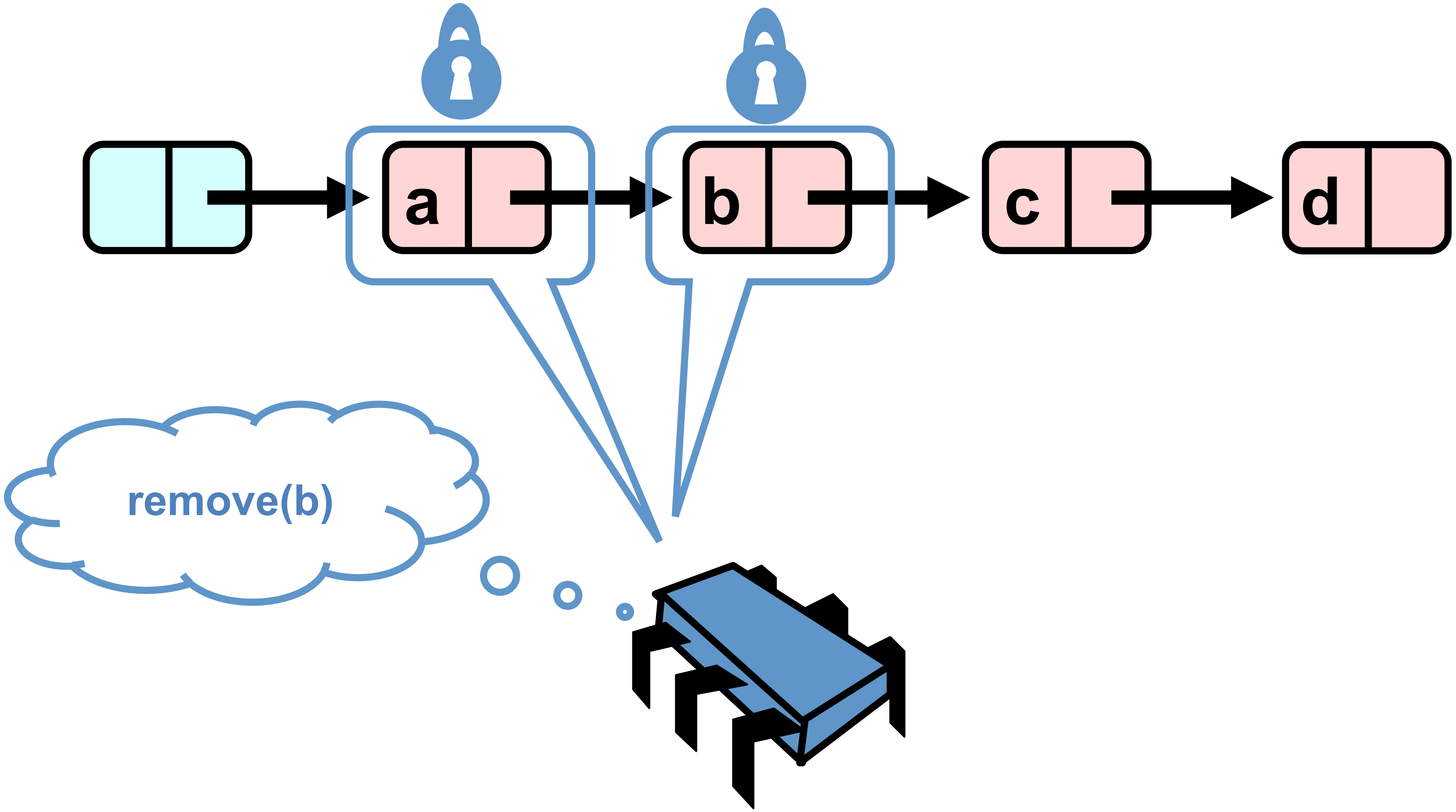
Removing a Node



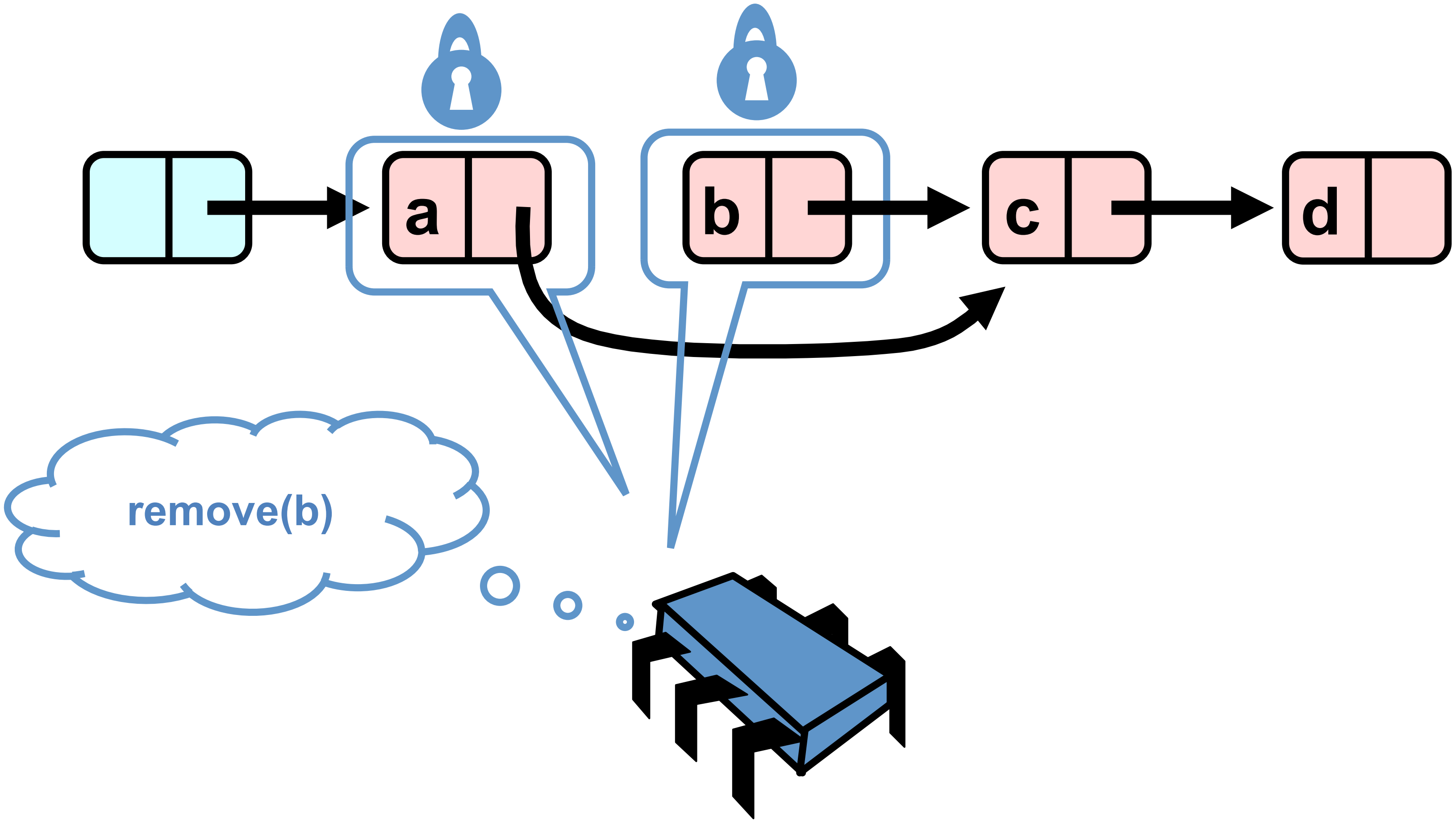
Removing a Node



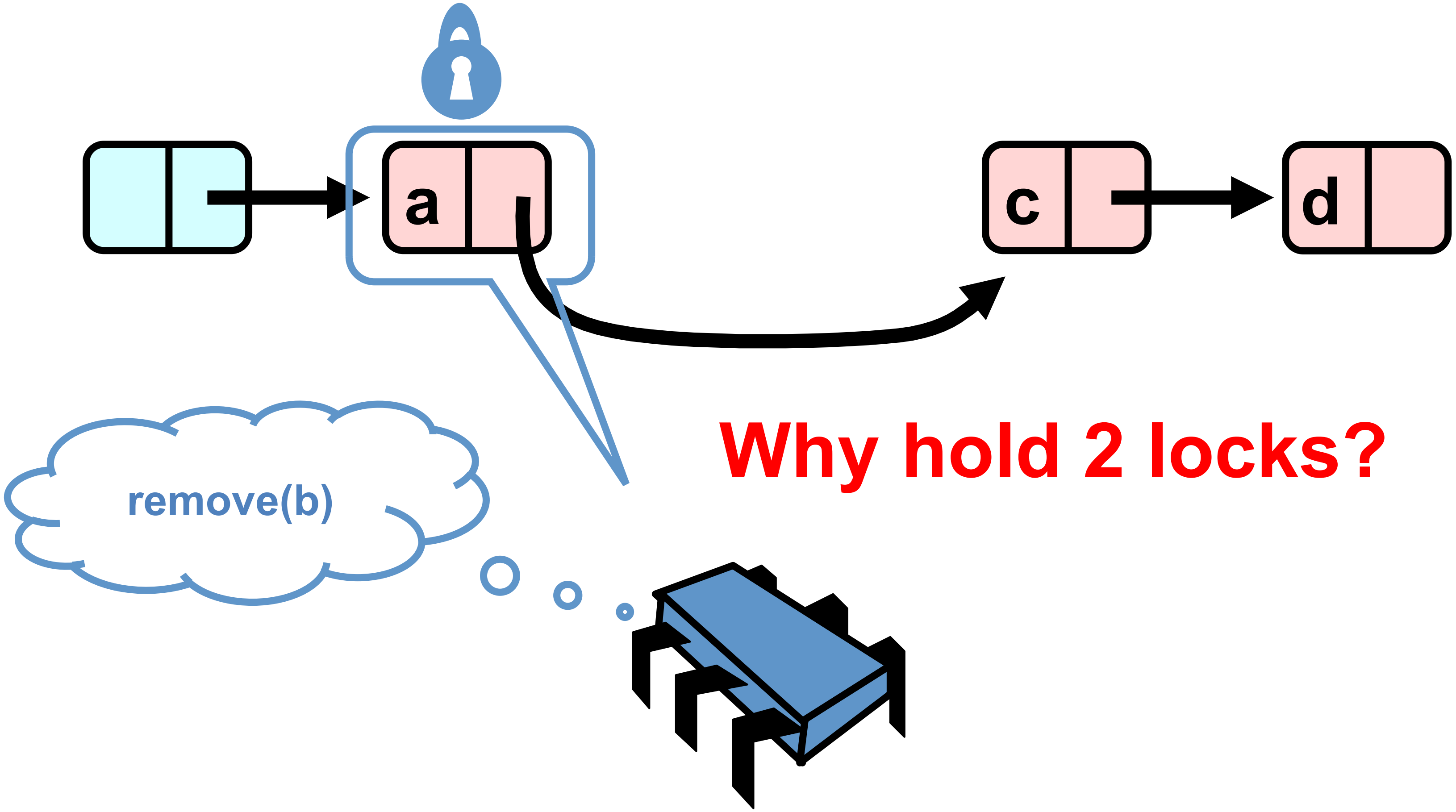
Removing a Node



Removing a Node

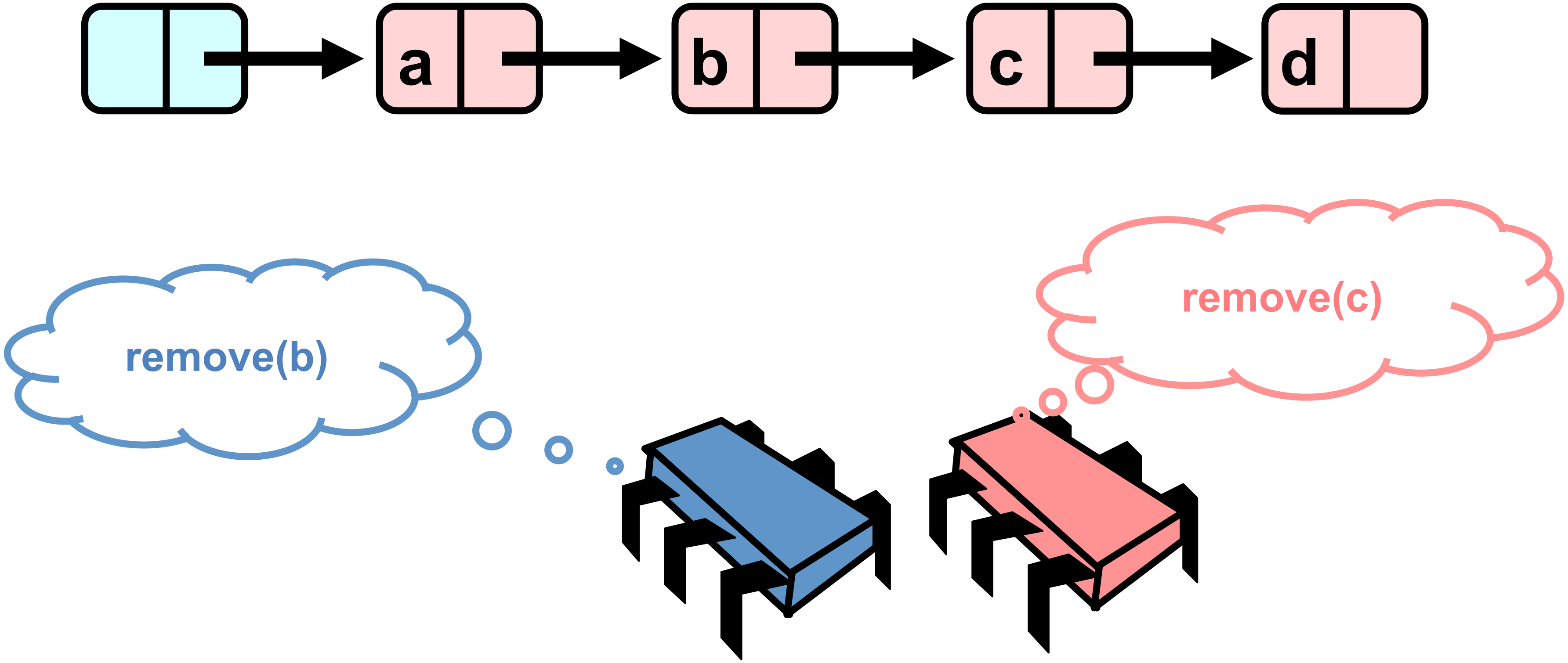


Removing a Node

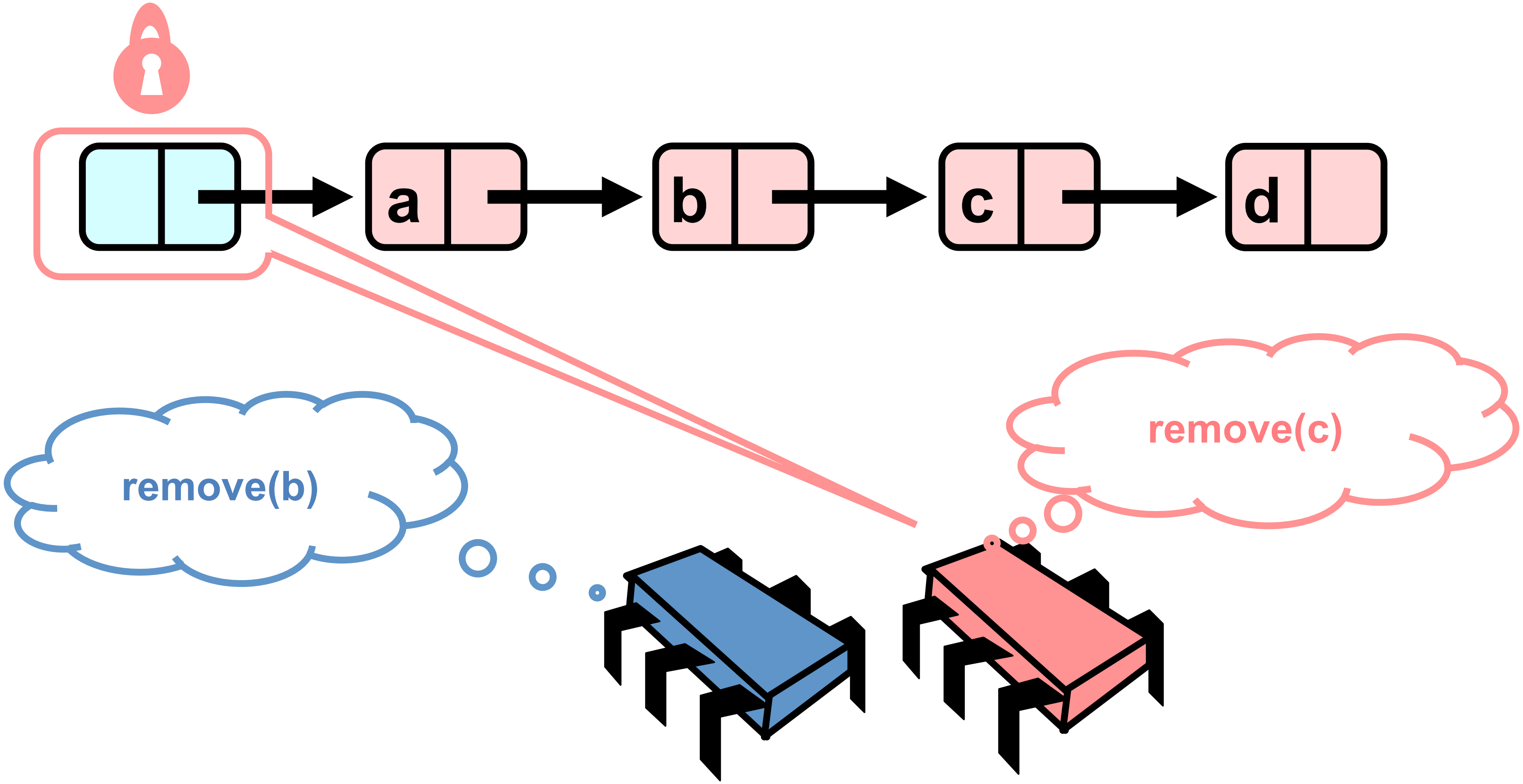


Why hold 2 locks?

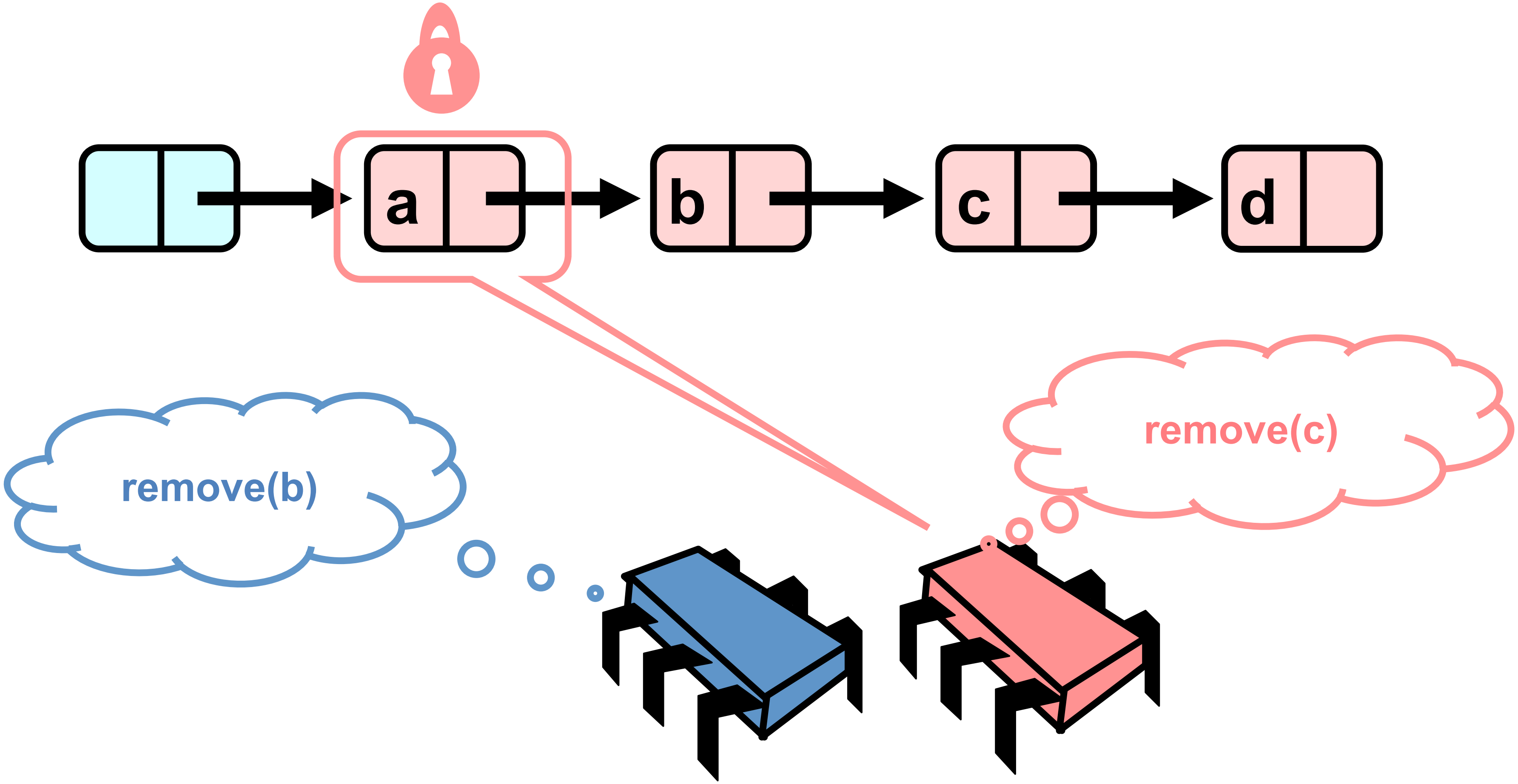
Concurrent Removes



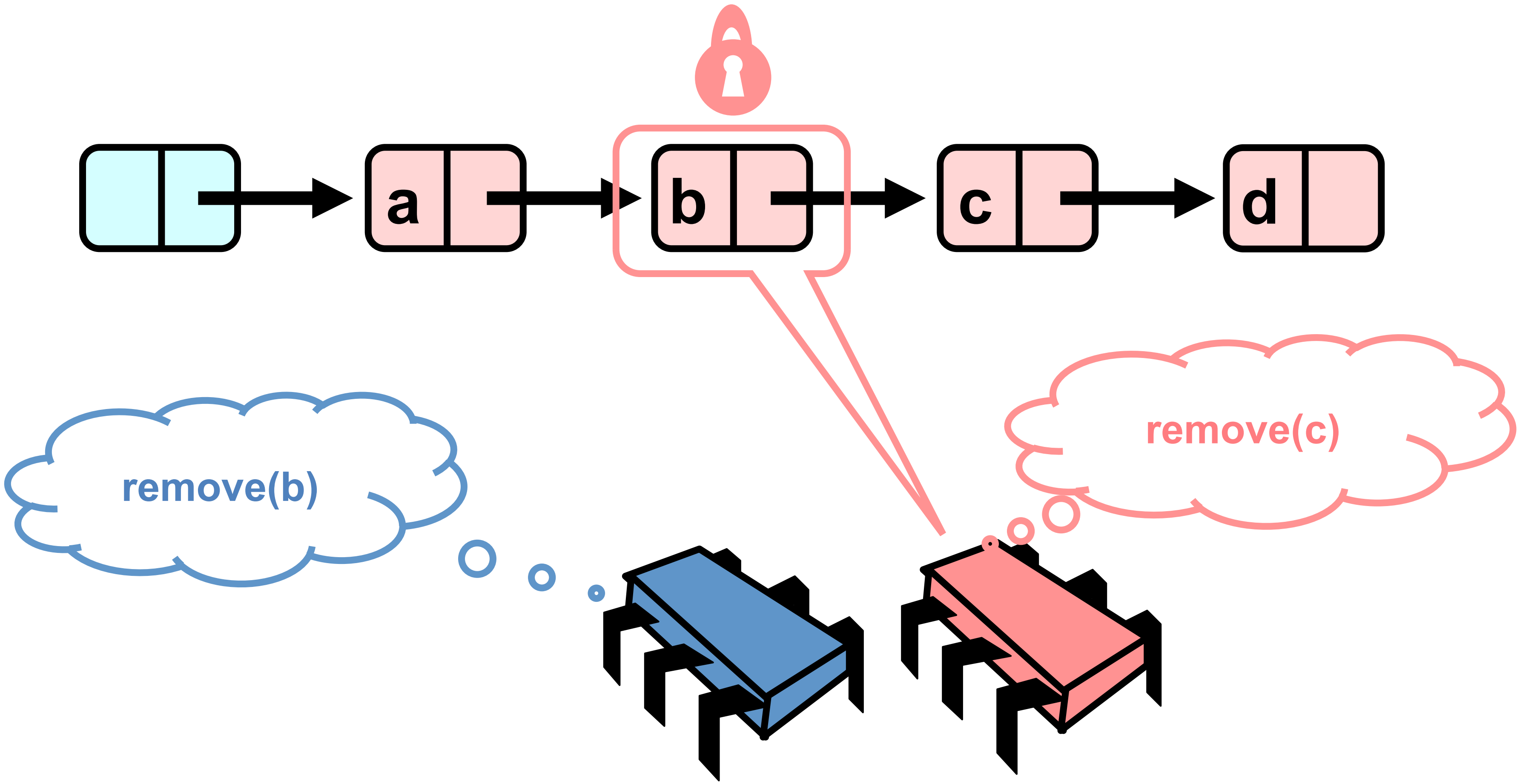
Concurrent Removes



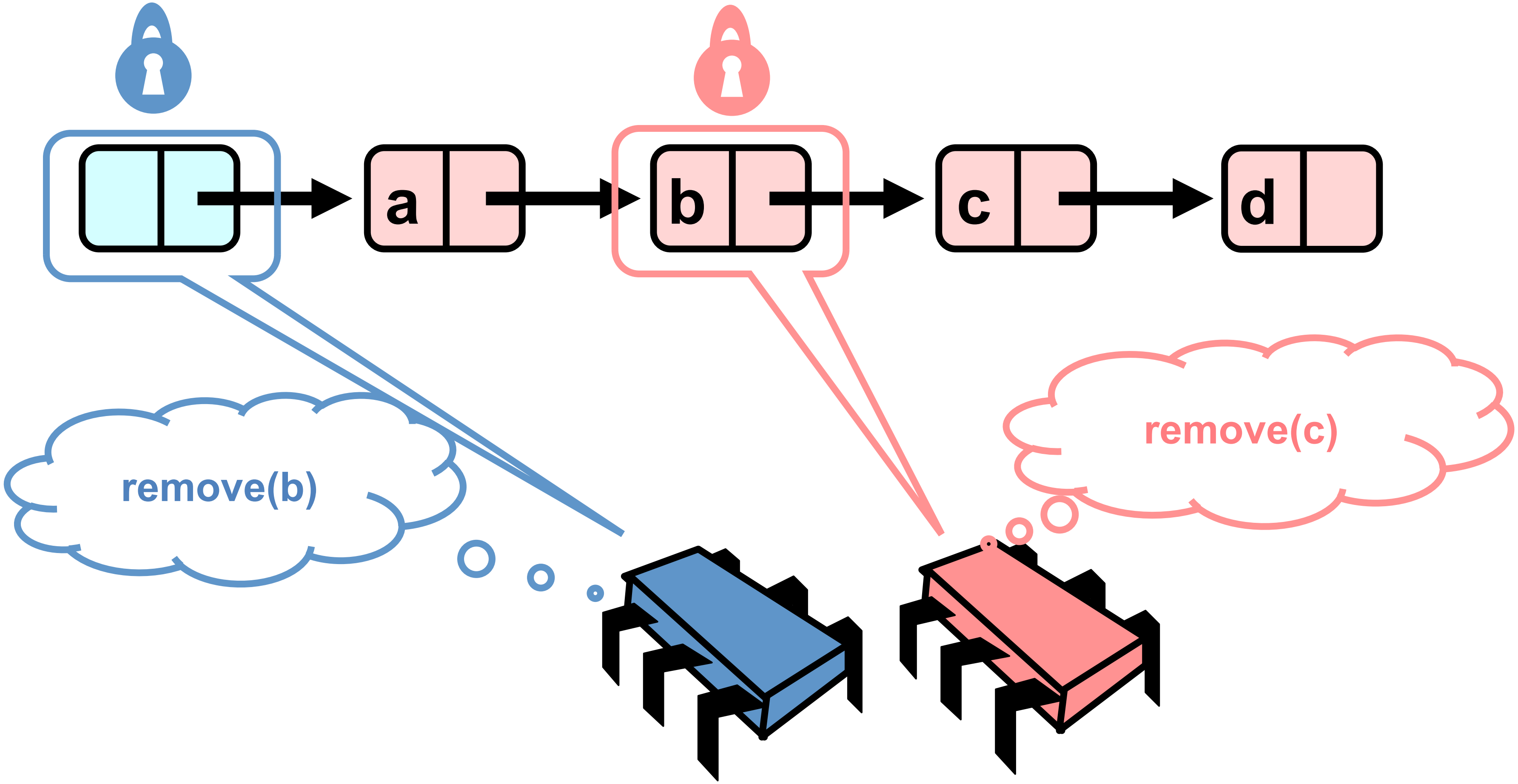
Concurrent Removes



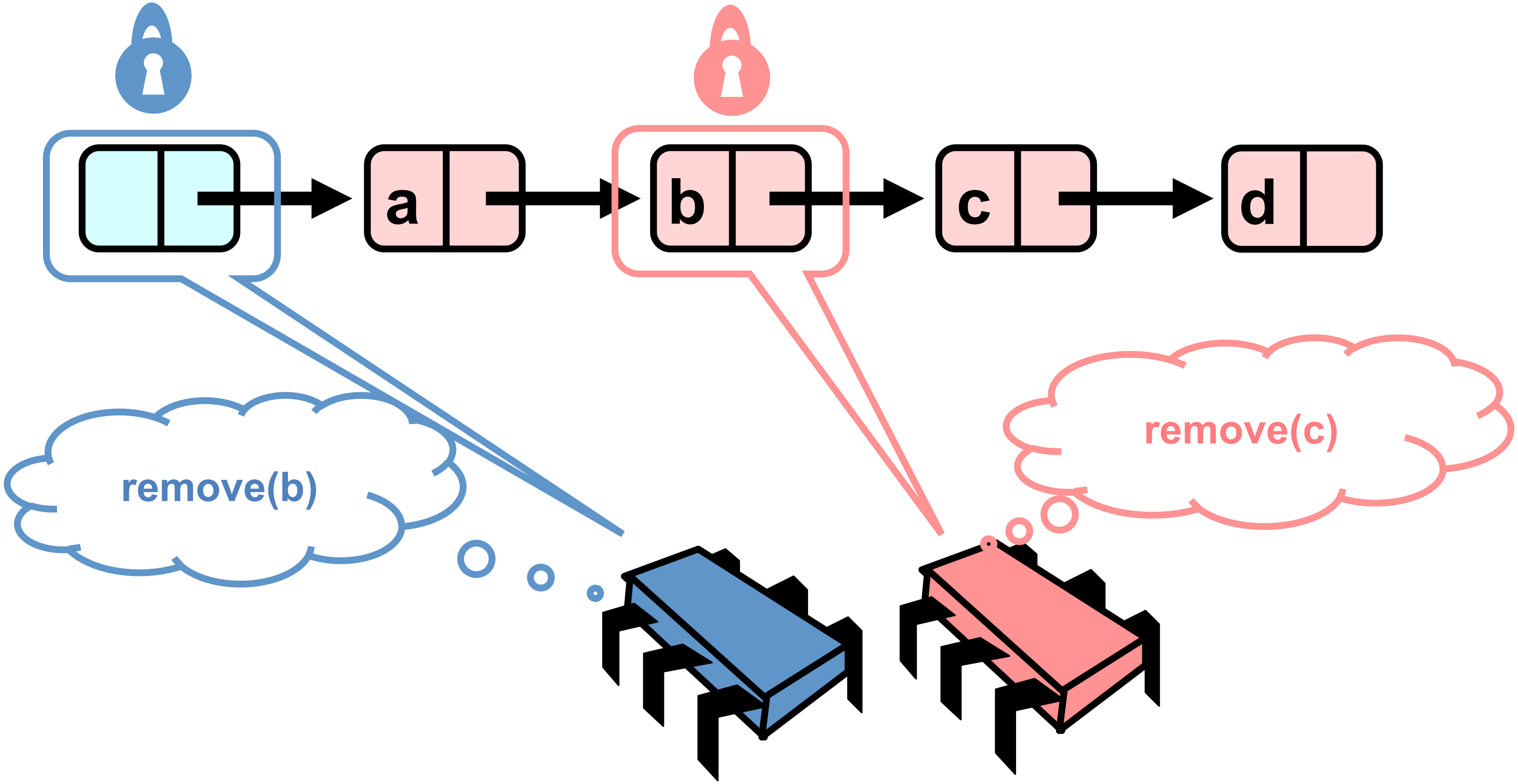
Concurrent Removes



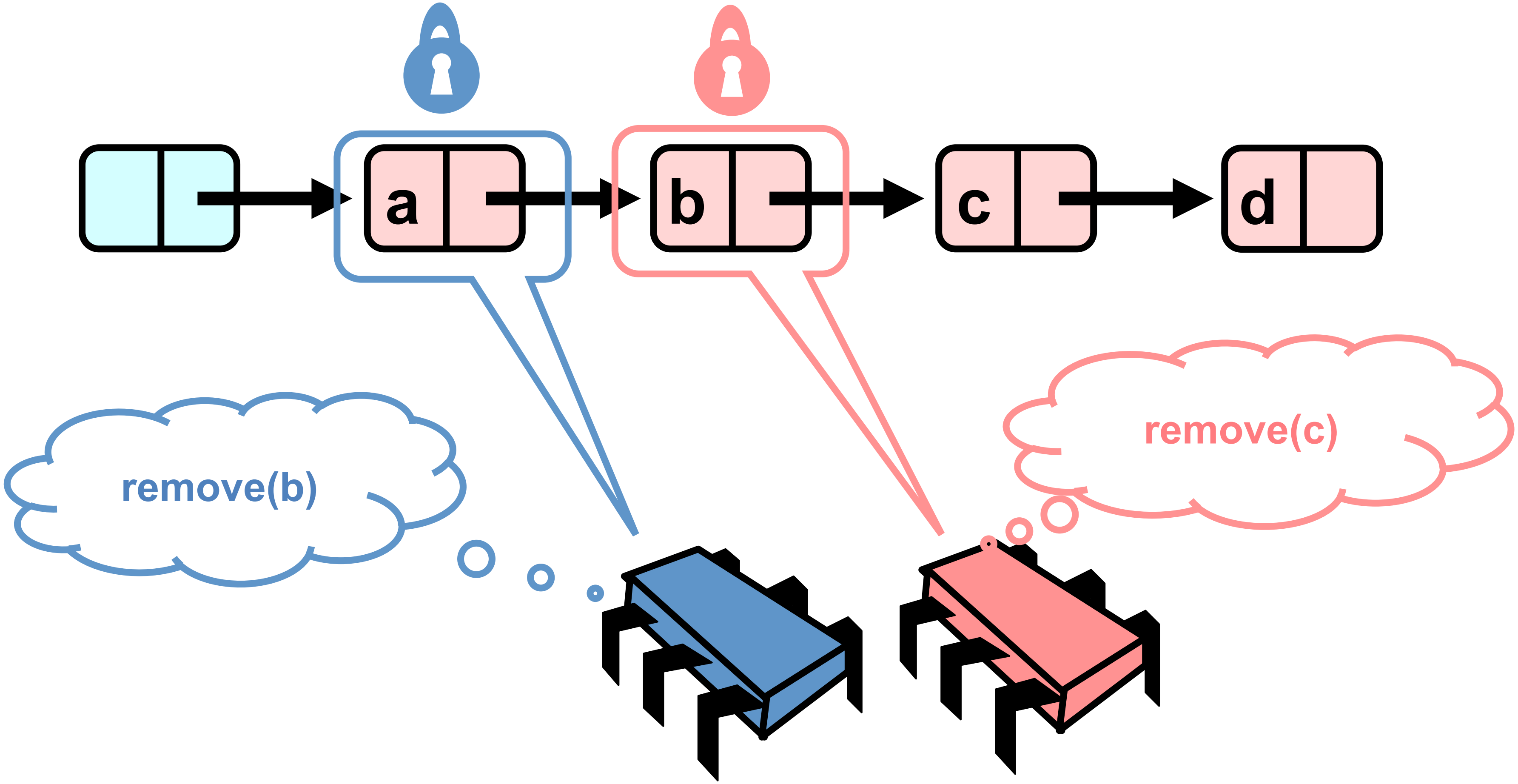
Concurrent Removes



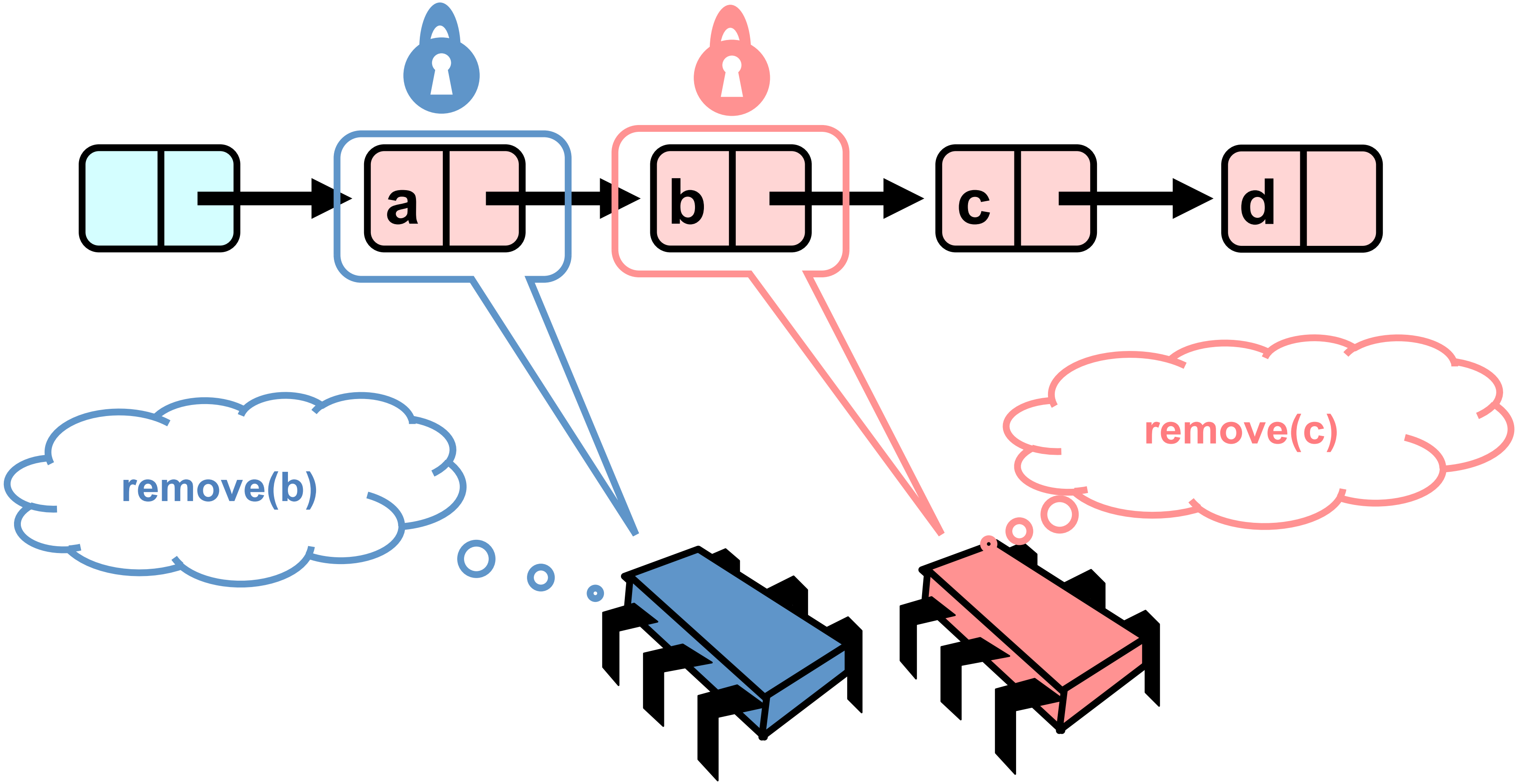
Concurrent Removes



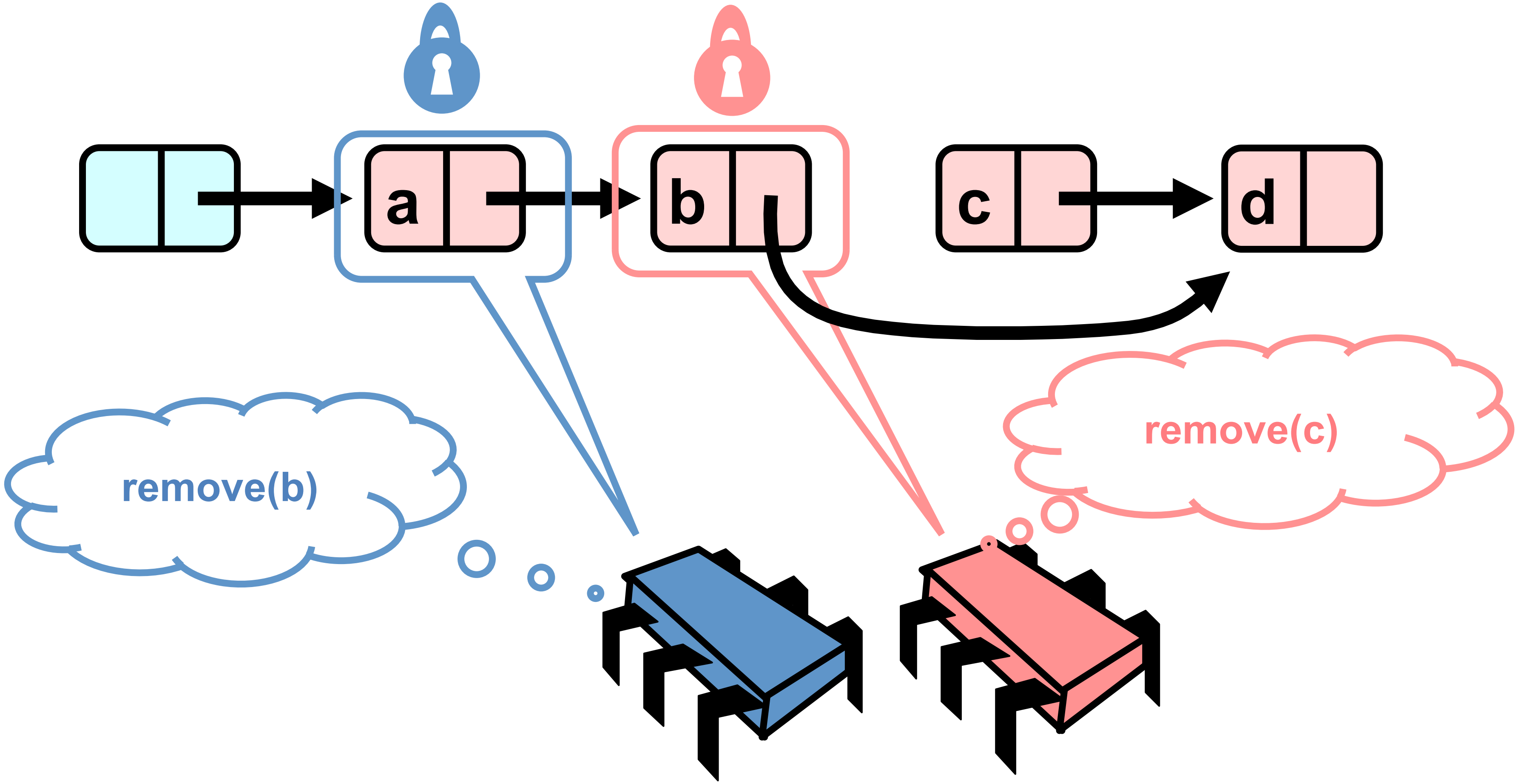
Concurrent Removes



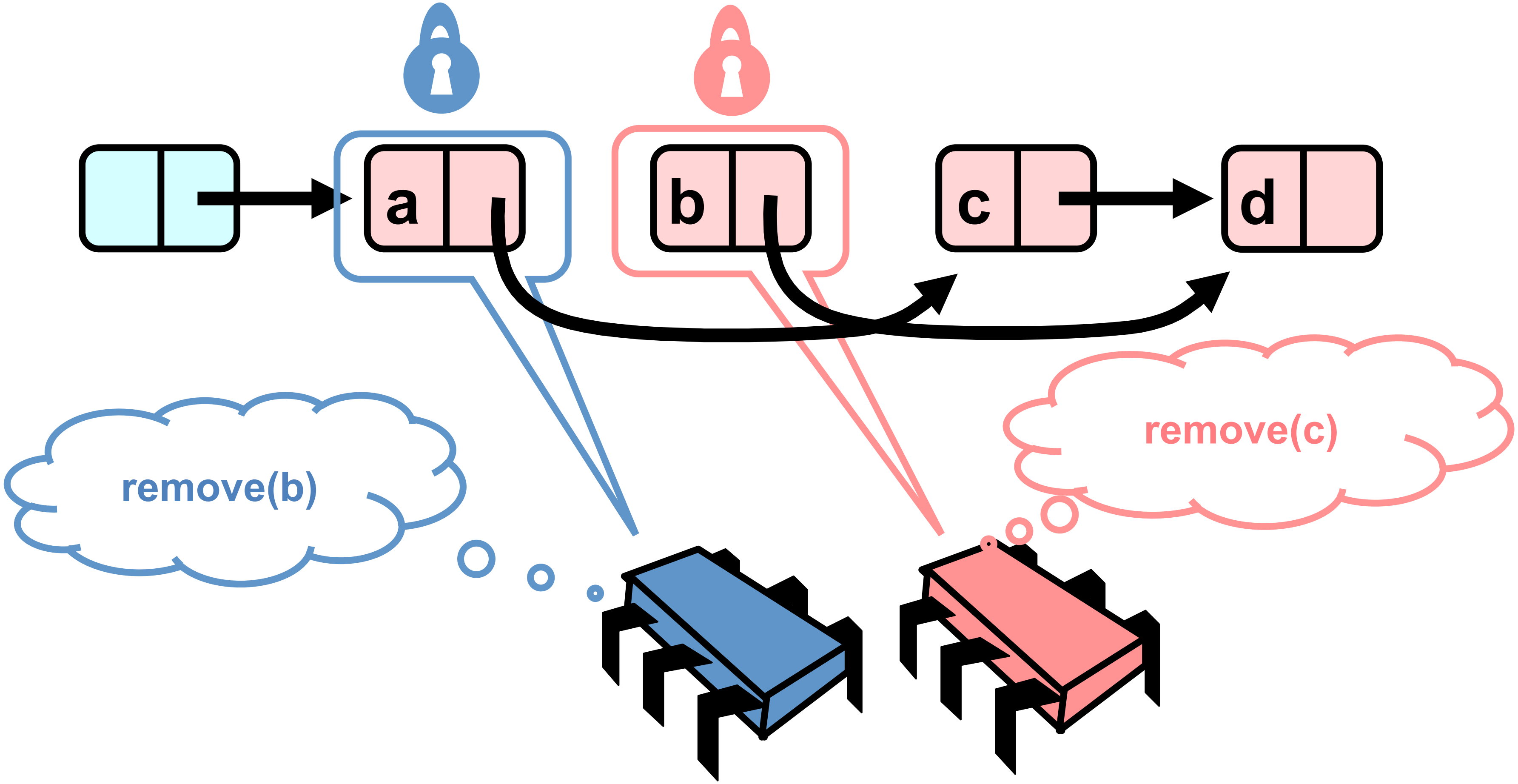
Concurrent Removes



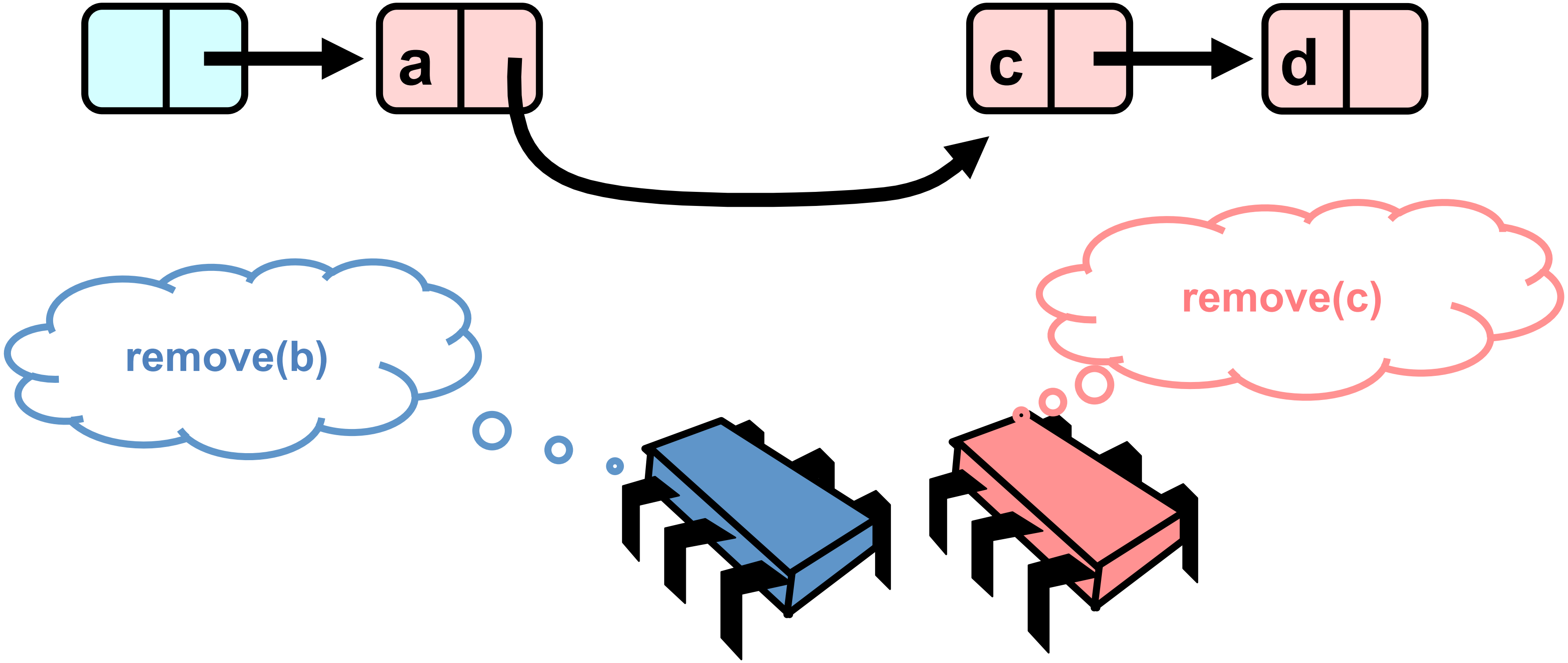
Concurrent Removes



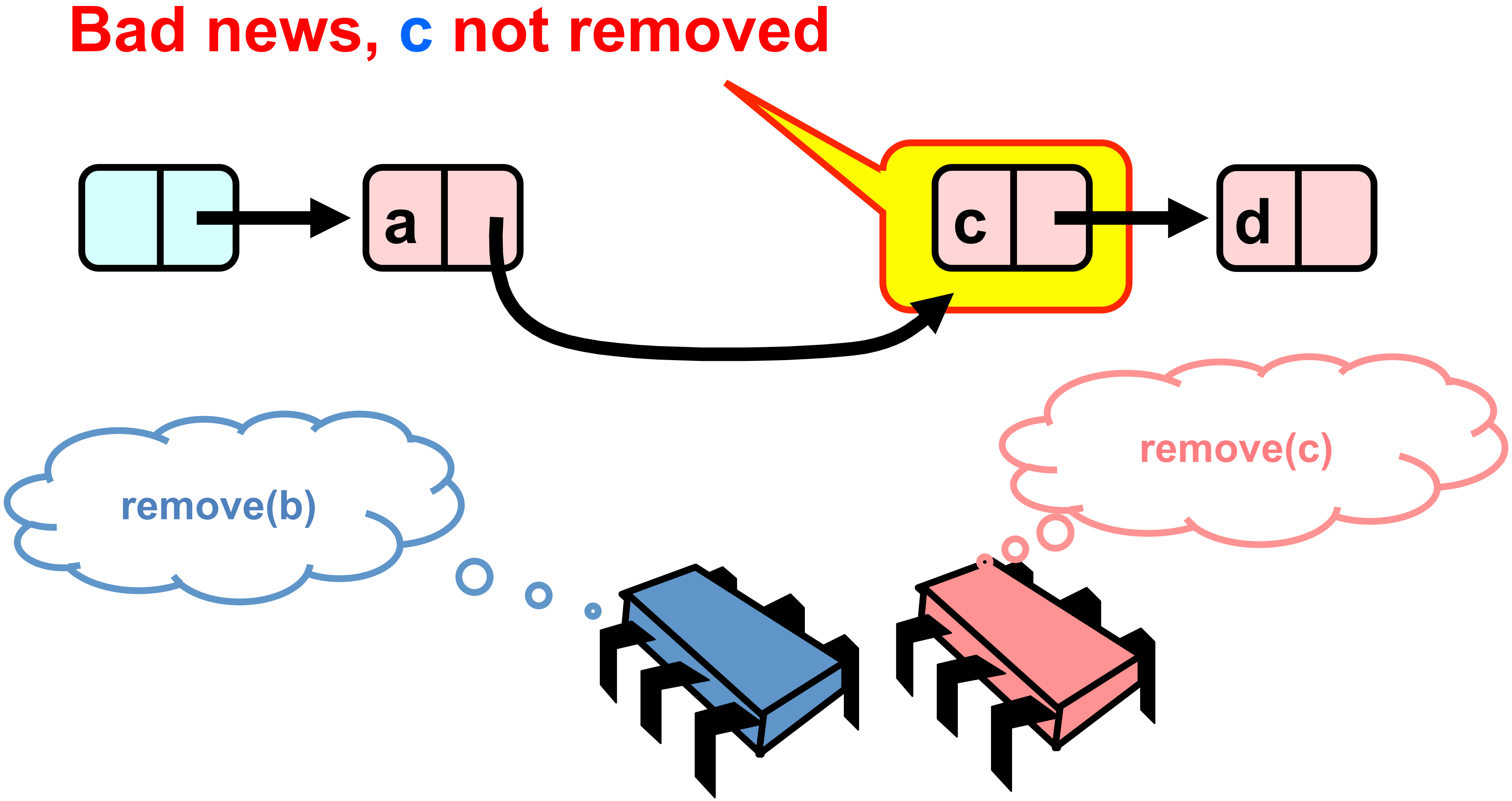
Concurrent Removes



Uh, Oh



Uh, Oh



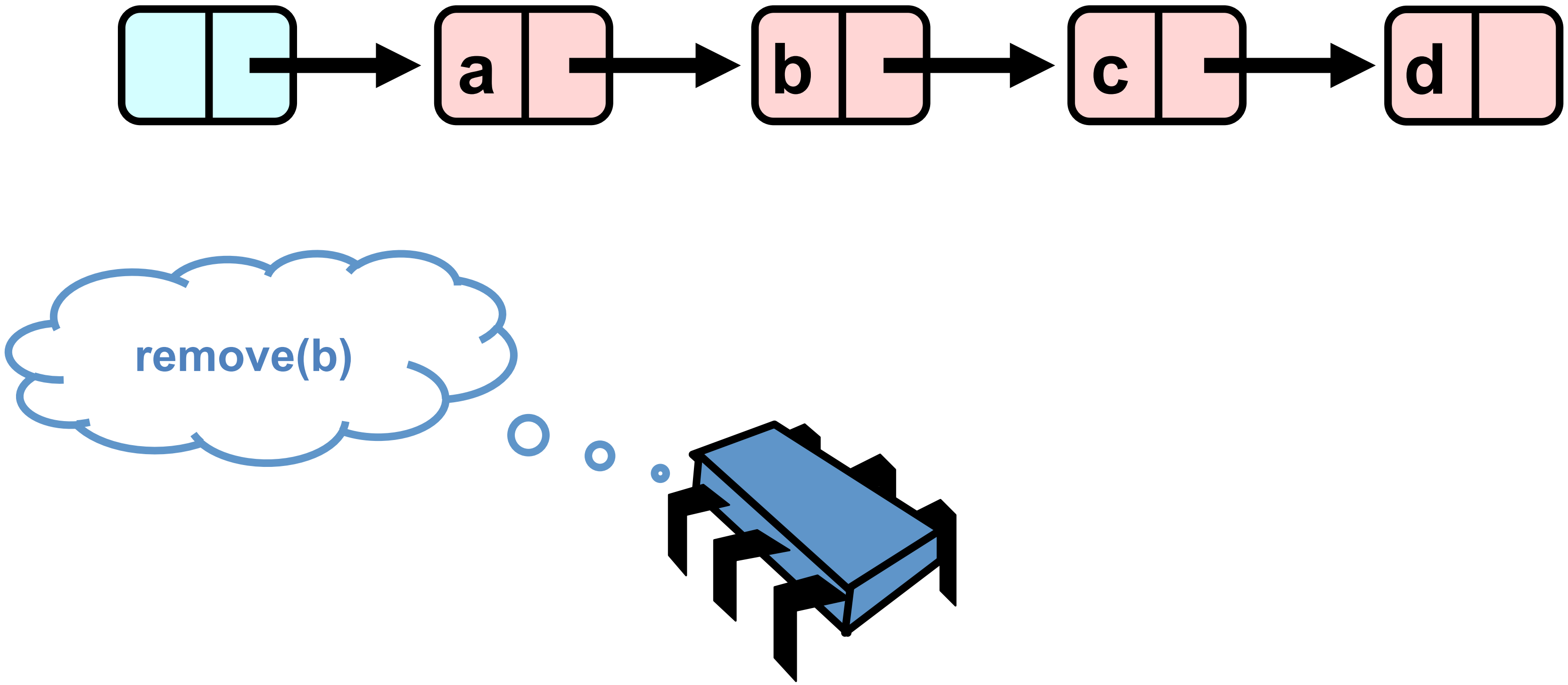
Insight

- If a node is locked, no one can delete the node’s *successor*

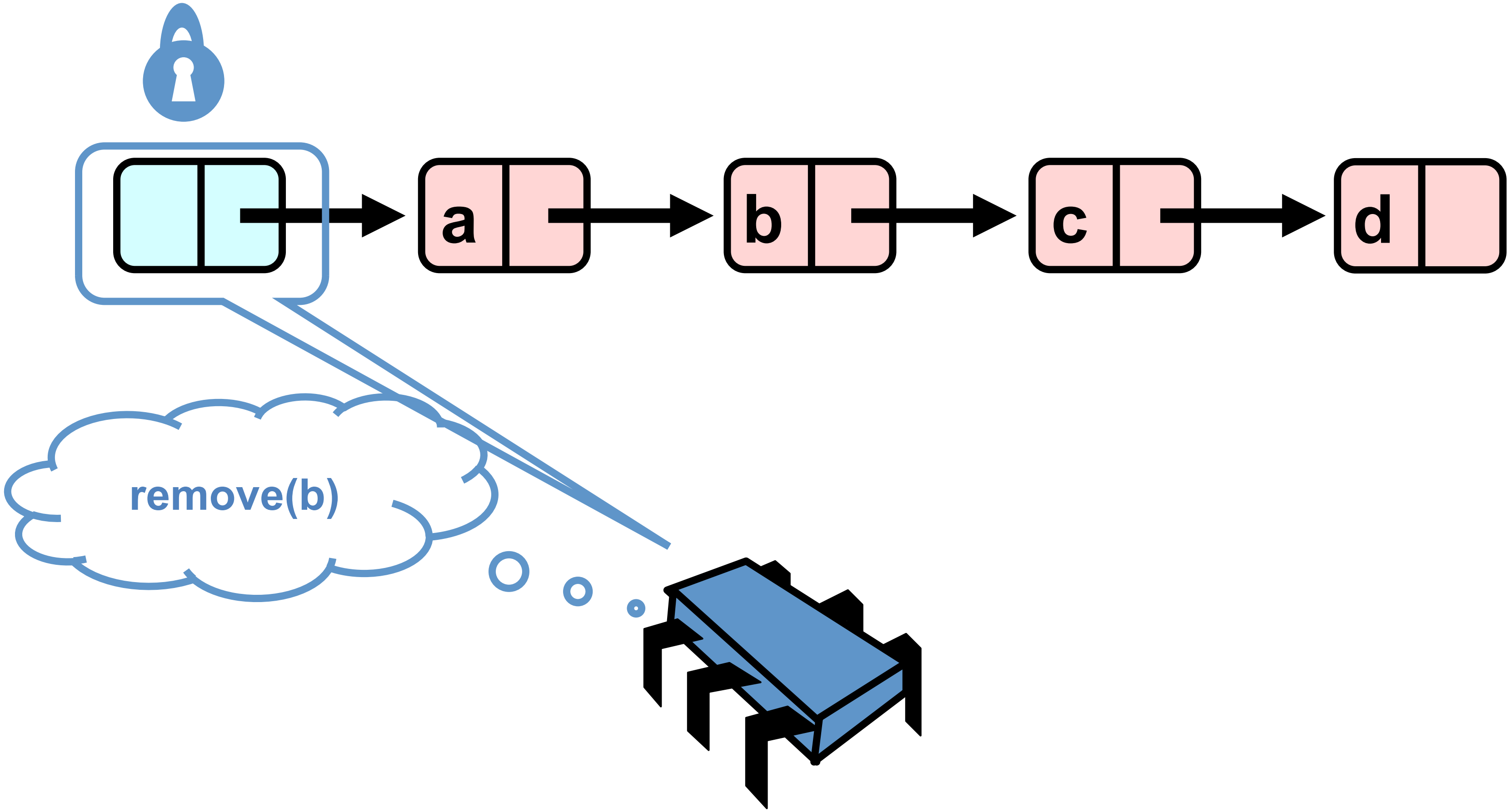
- If a thread locks
  - the node to be deleted
  - and also its predecessor
- then it works!

- That’s why we (have to) use two locks!

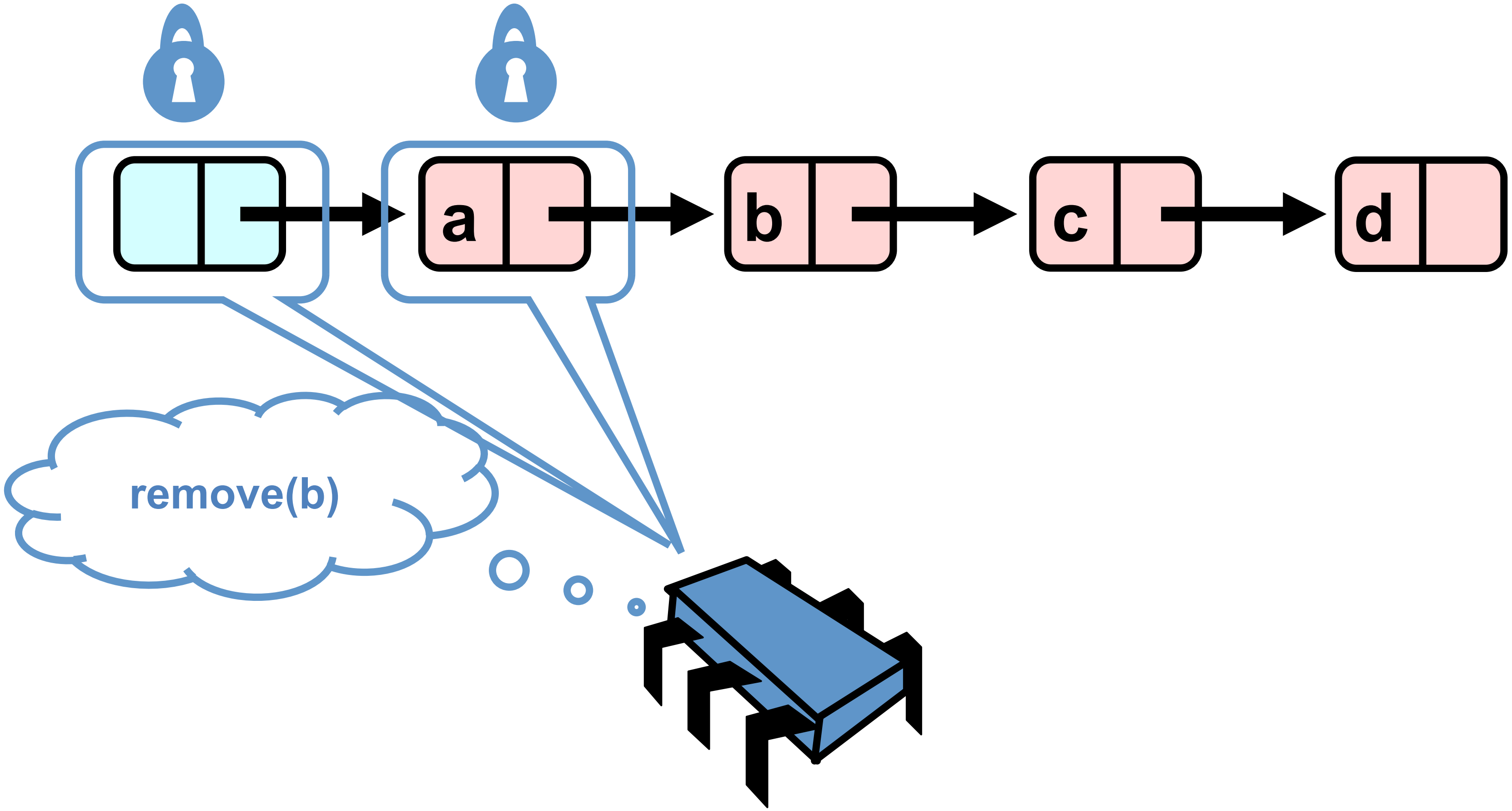
Hand-Over-Hand Again



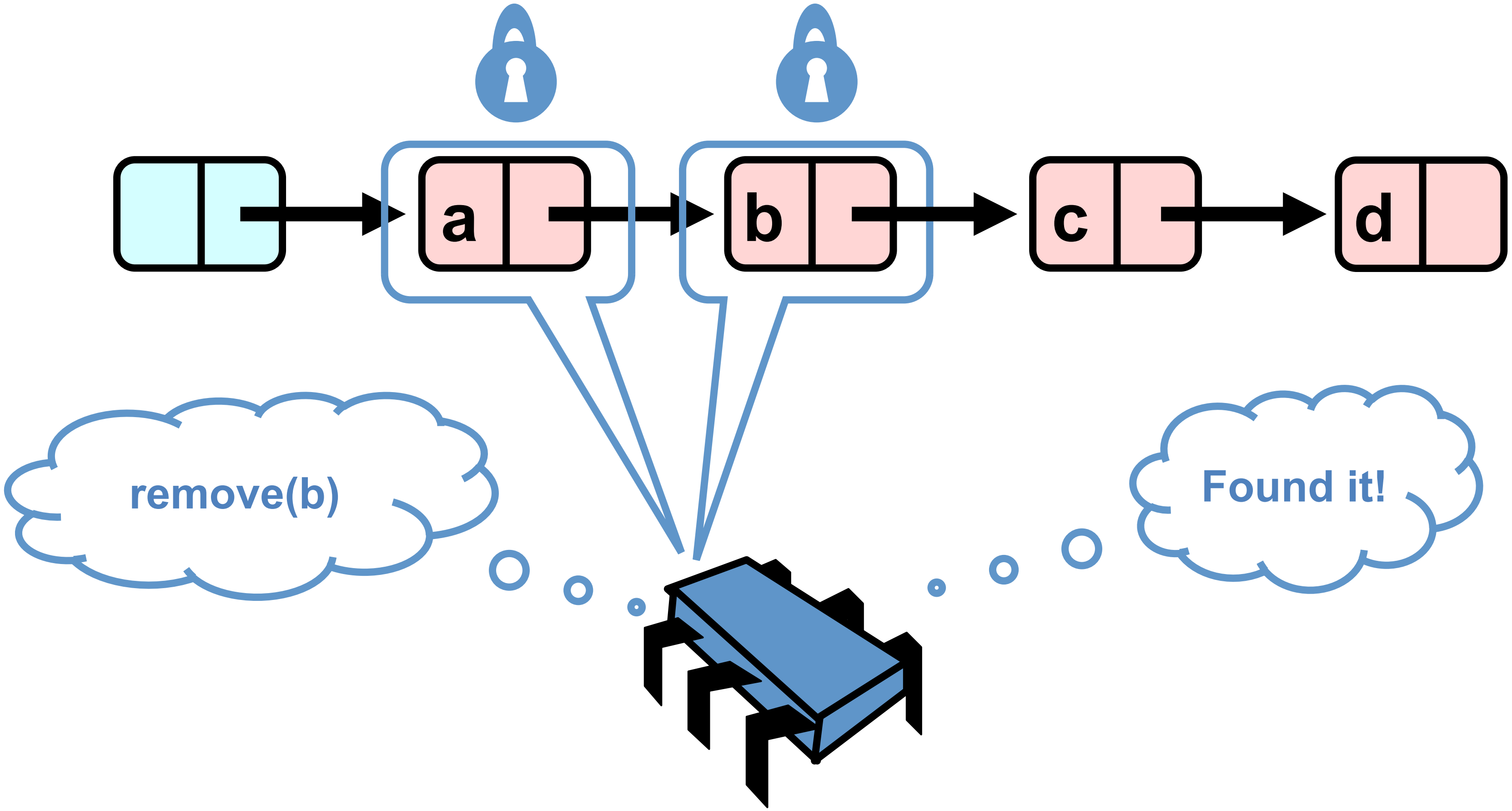
Hand-Over-Hand Again



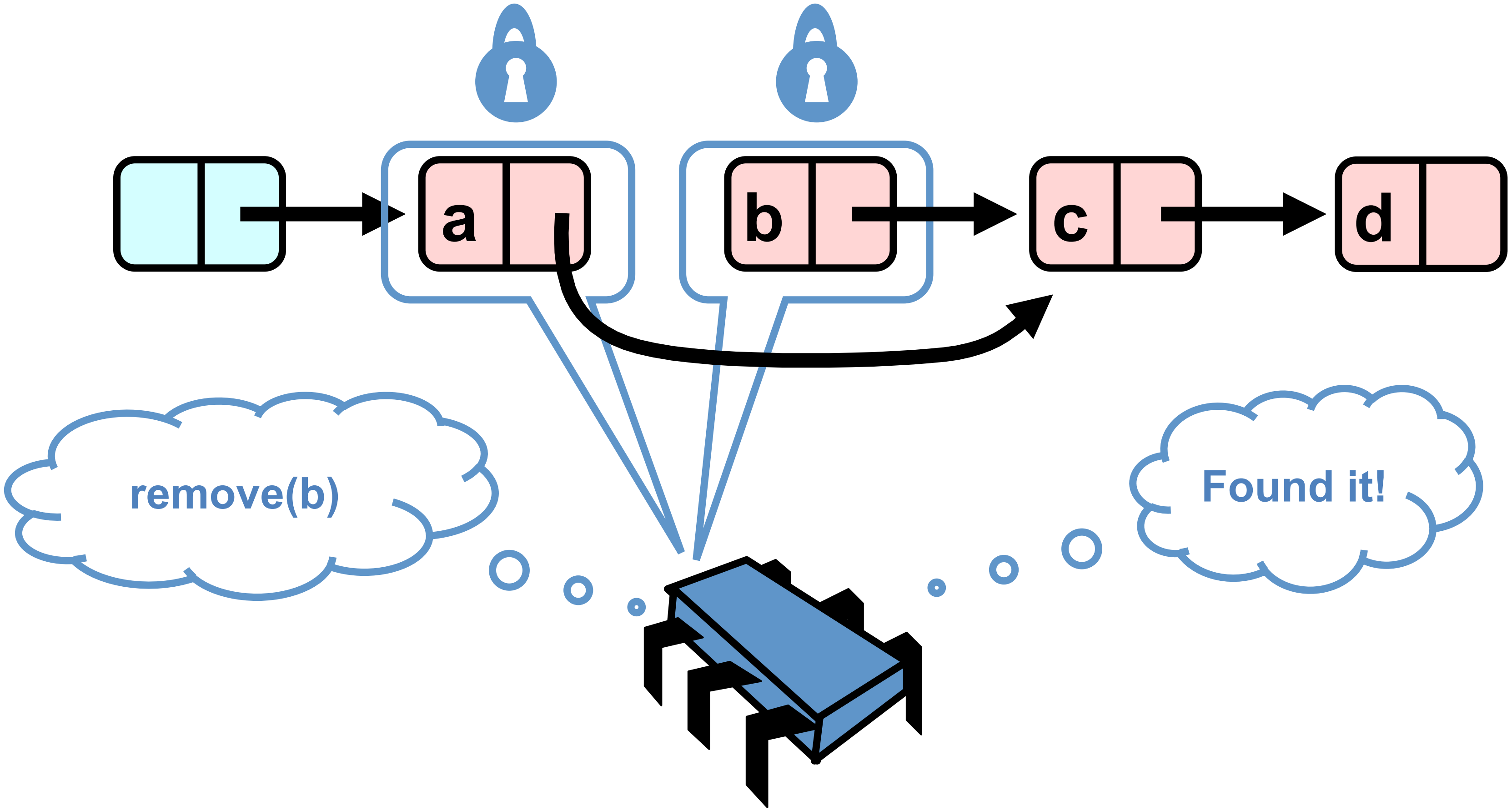
Hand-Over-Hand Again



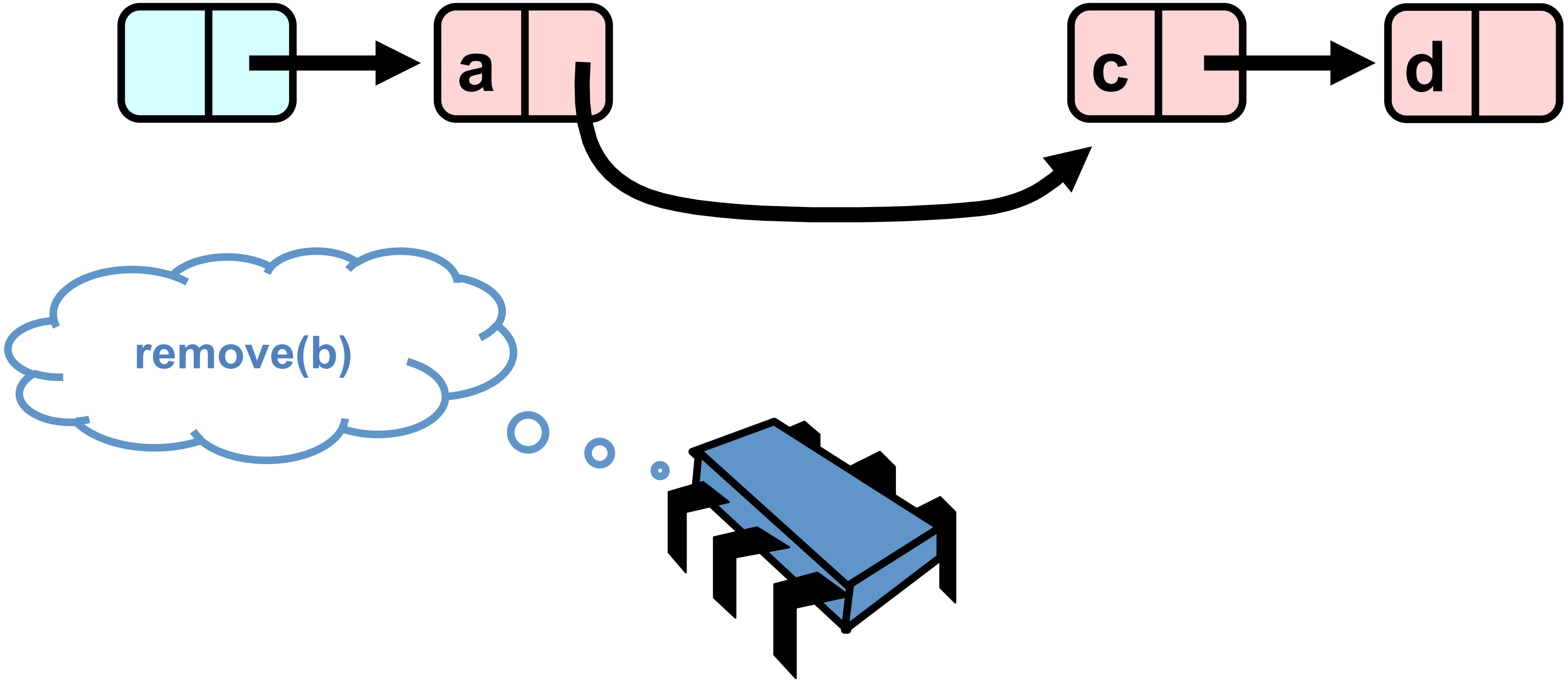
Hand-Over-Hand Again



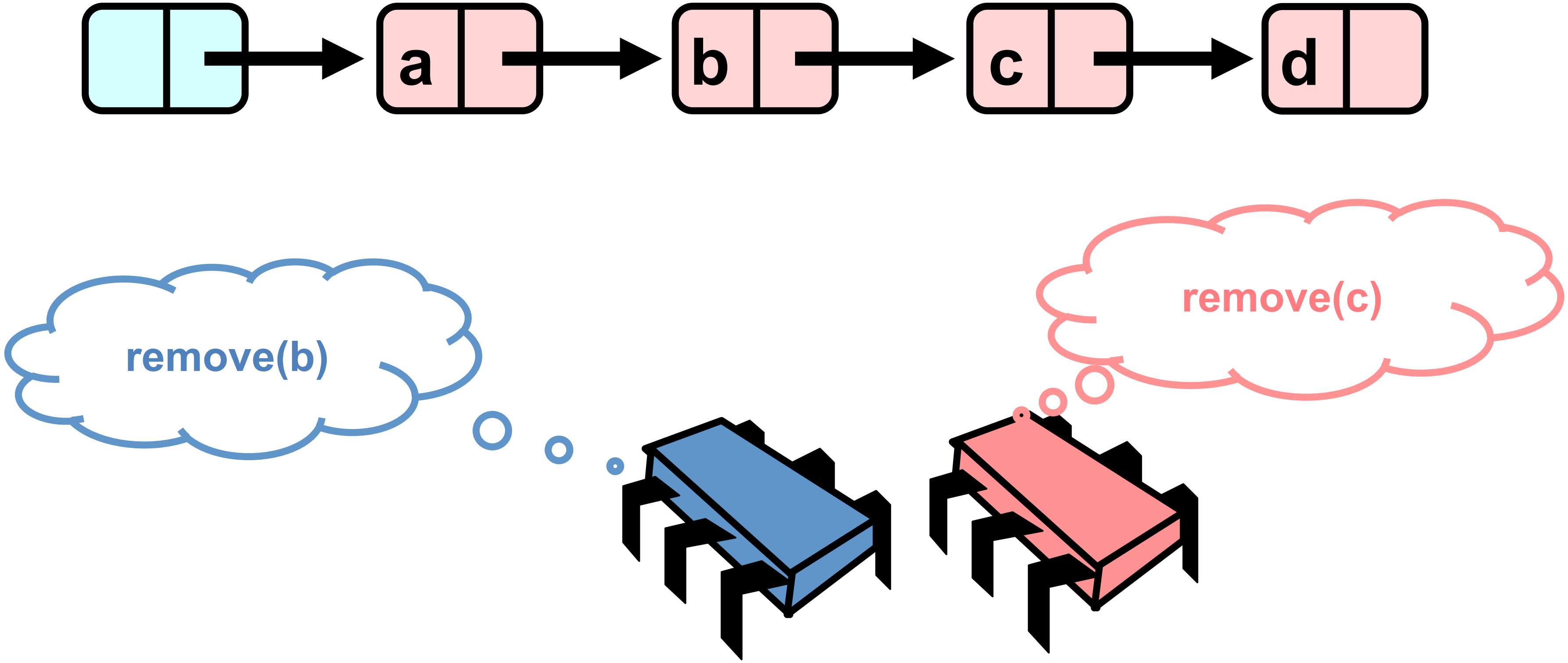
Hand-Over-Hand Again



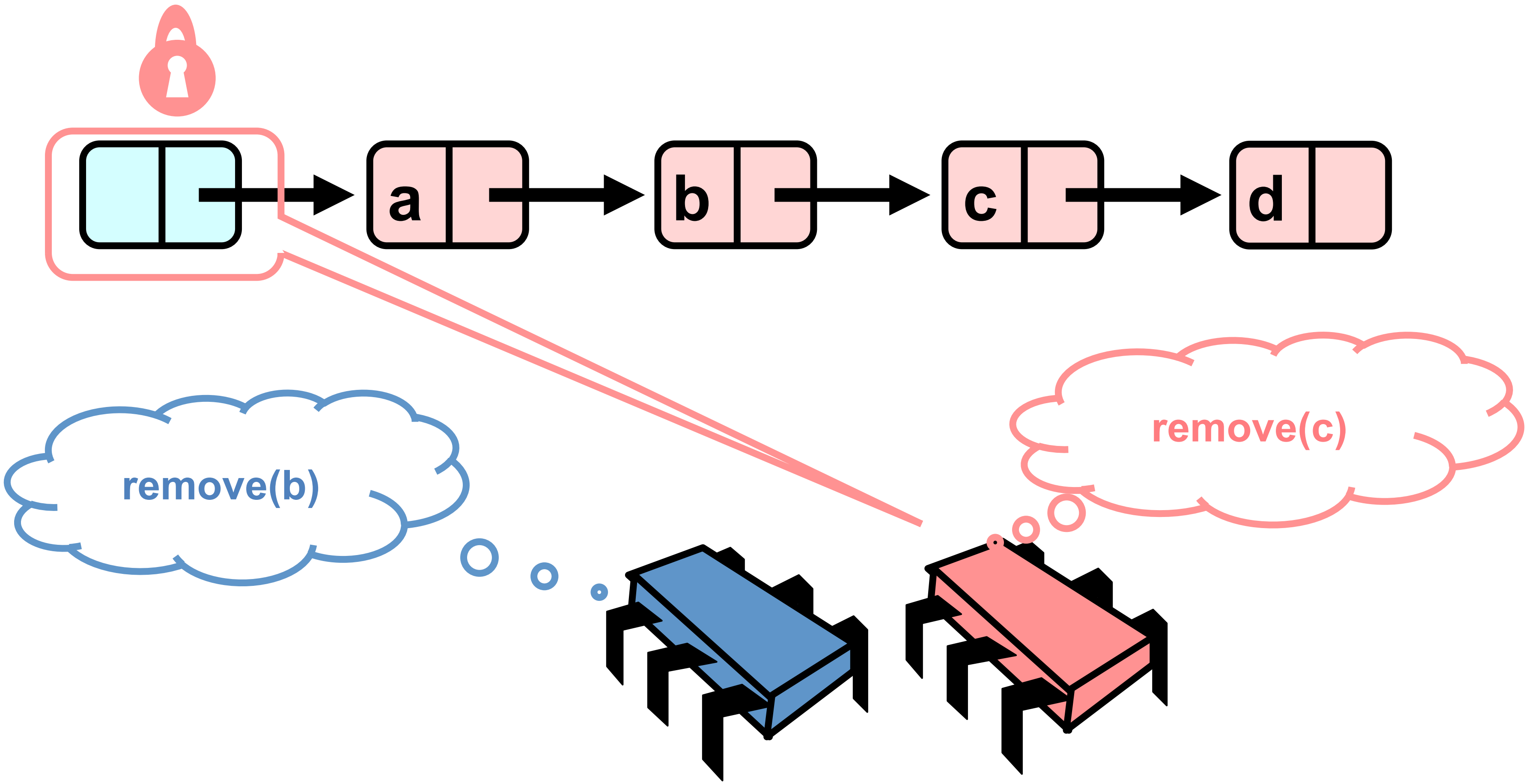
Hand-Over-Hand Again



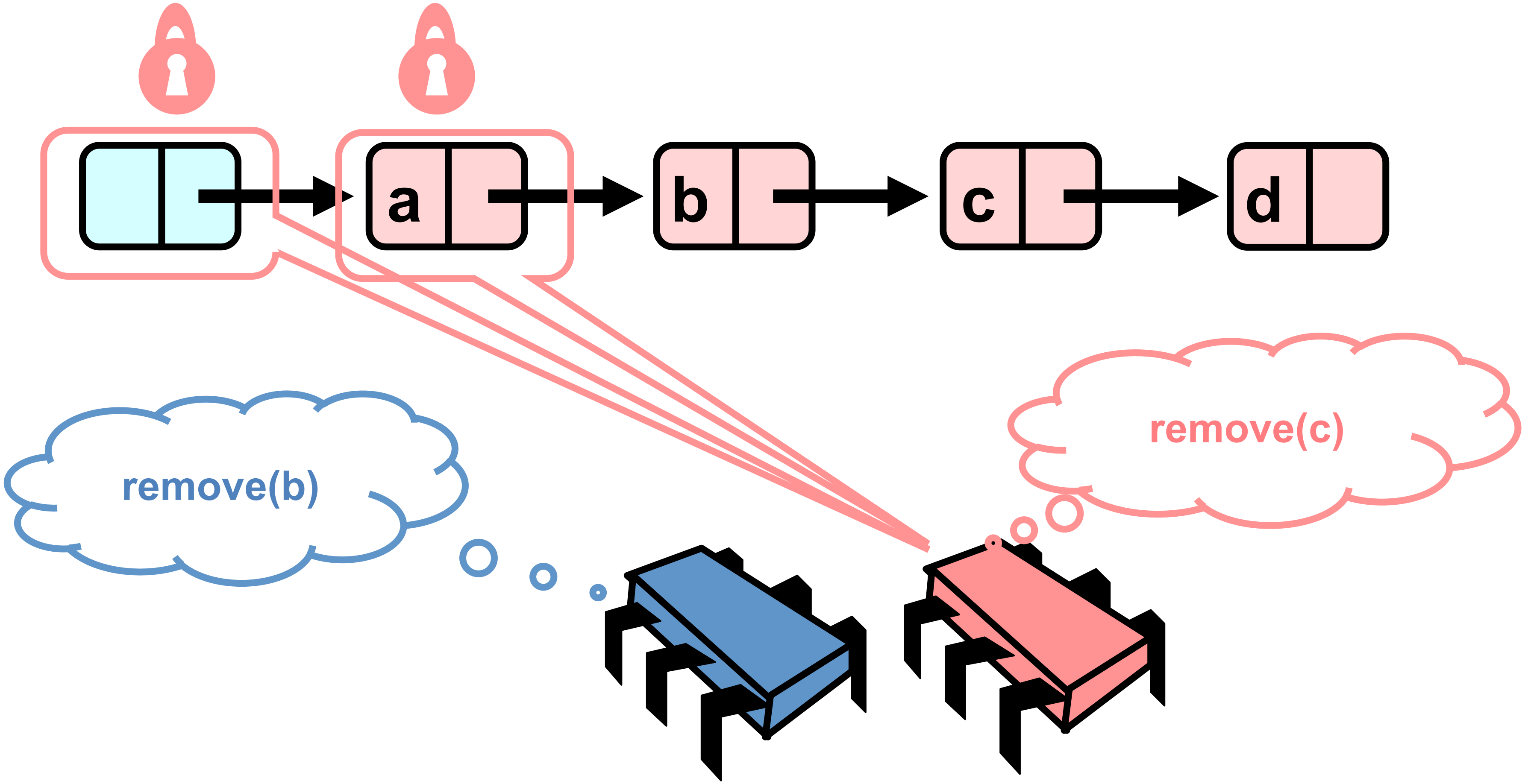
Removing a Node



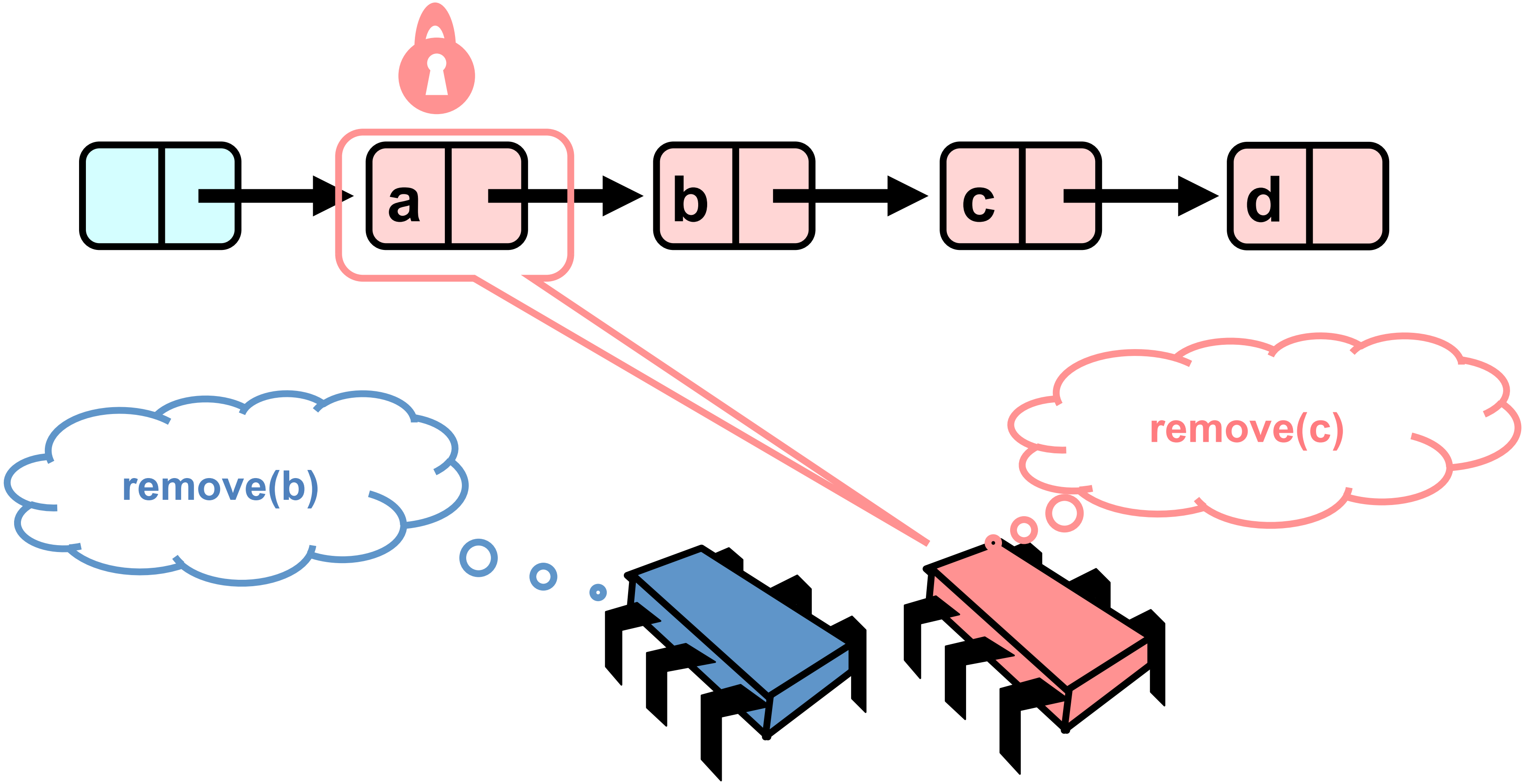
Removing a Node



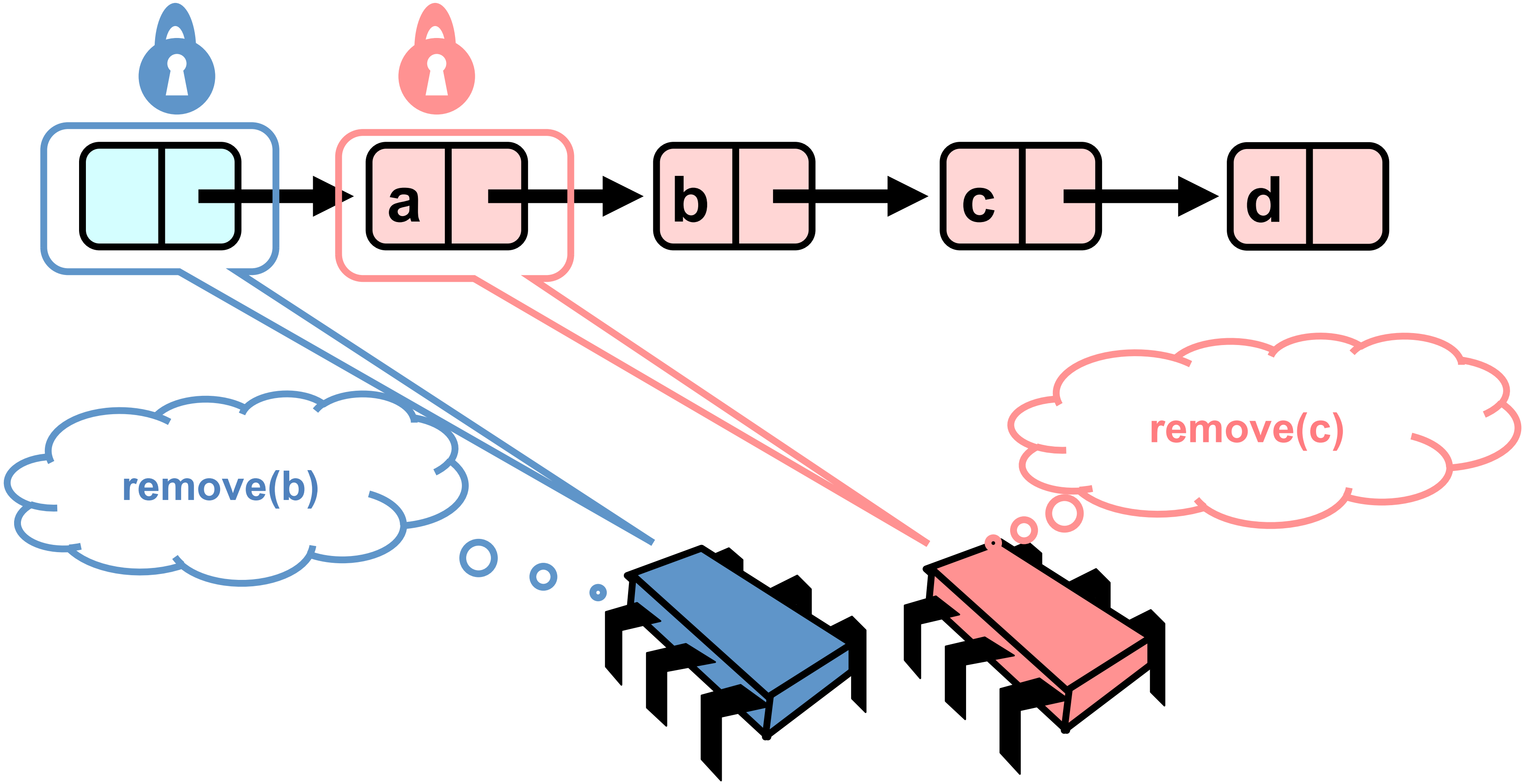
Removing a Node



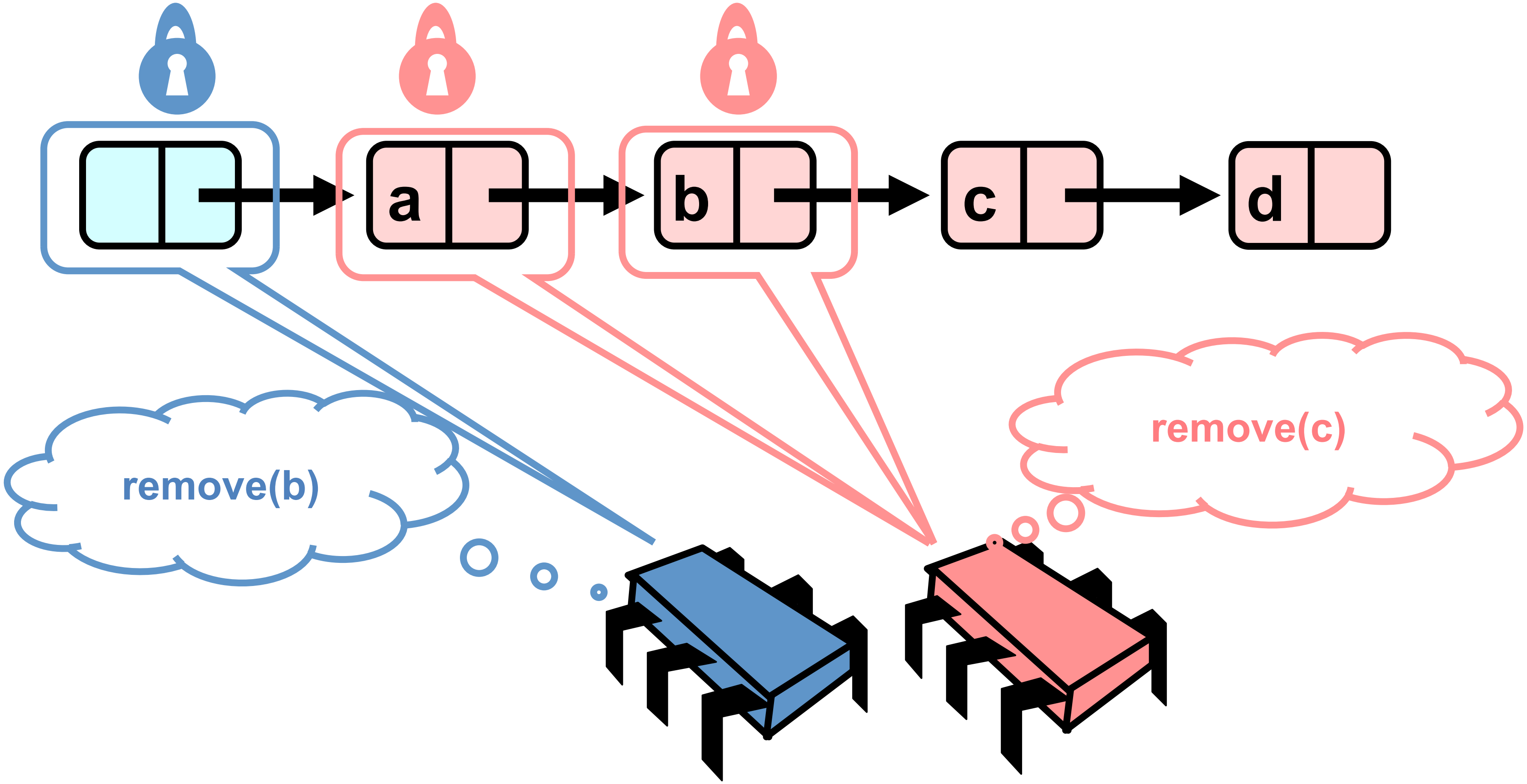
Removing a Node



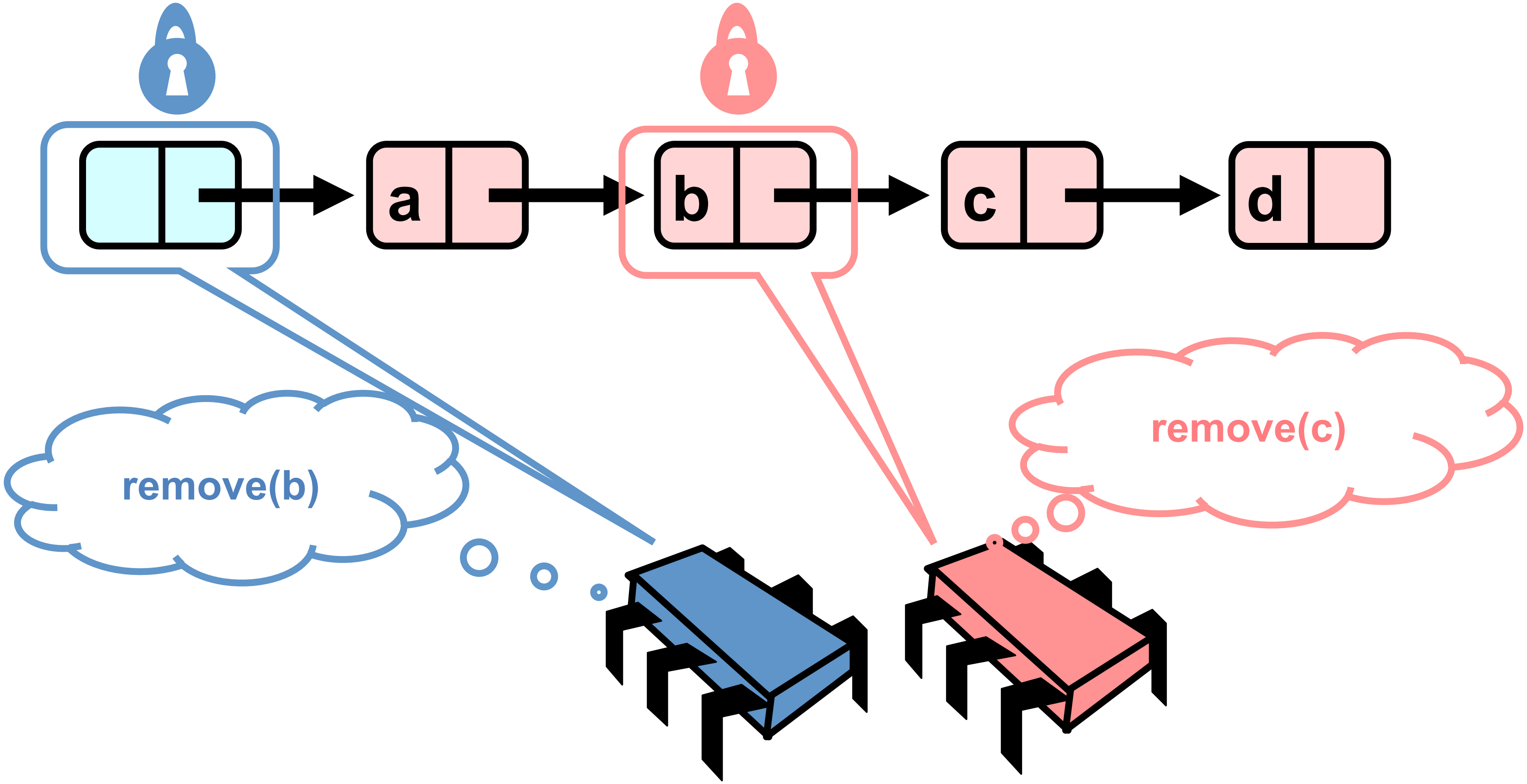
Removing a Node



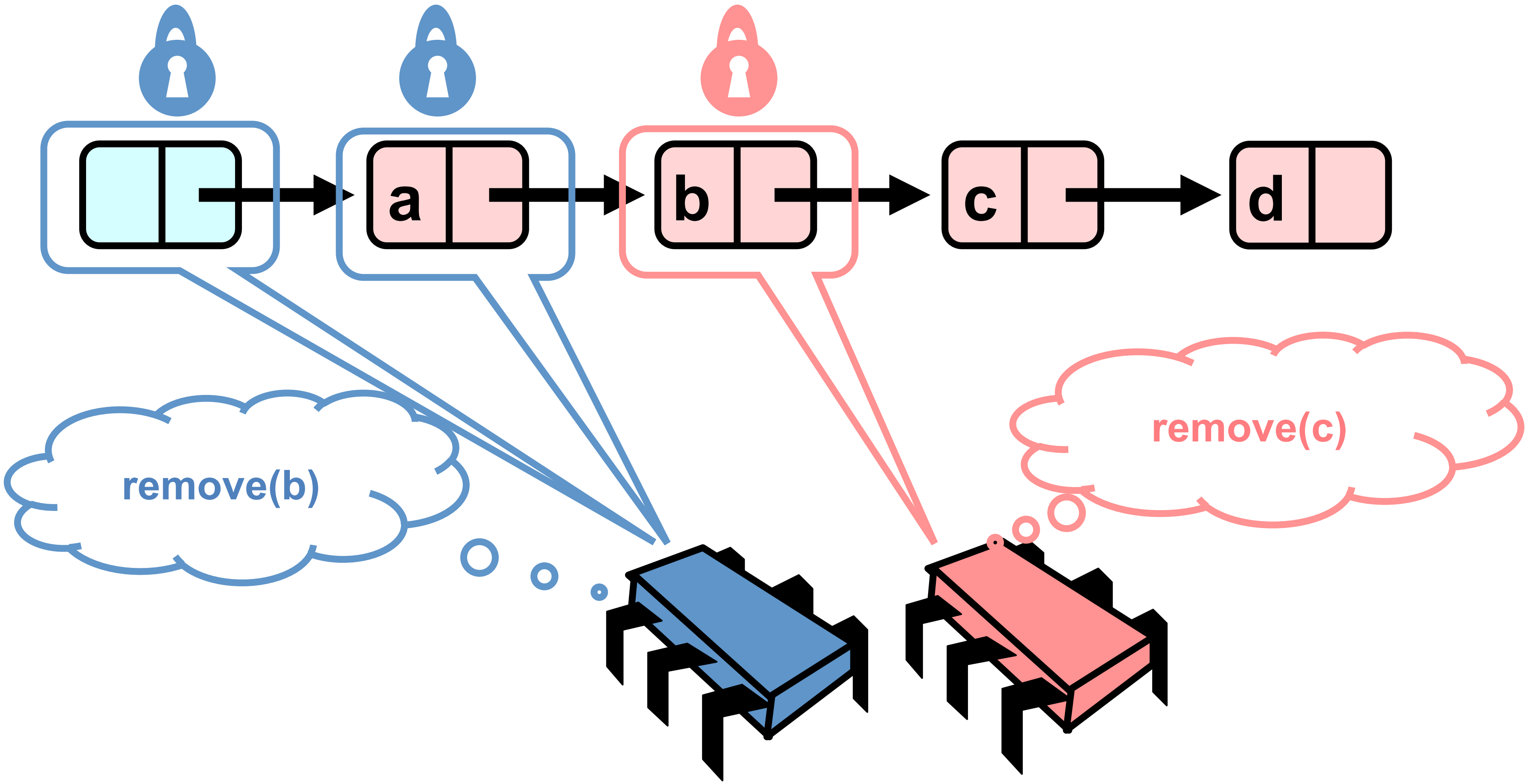
Removing a Node



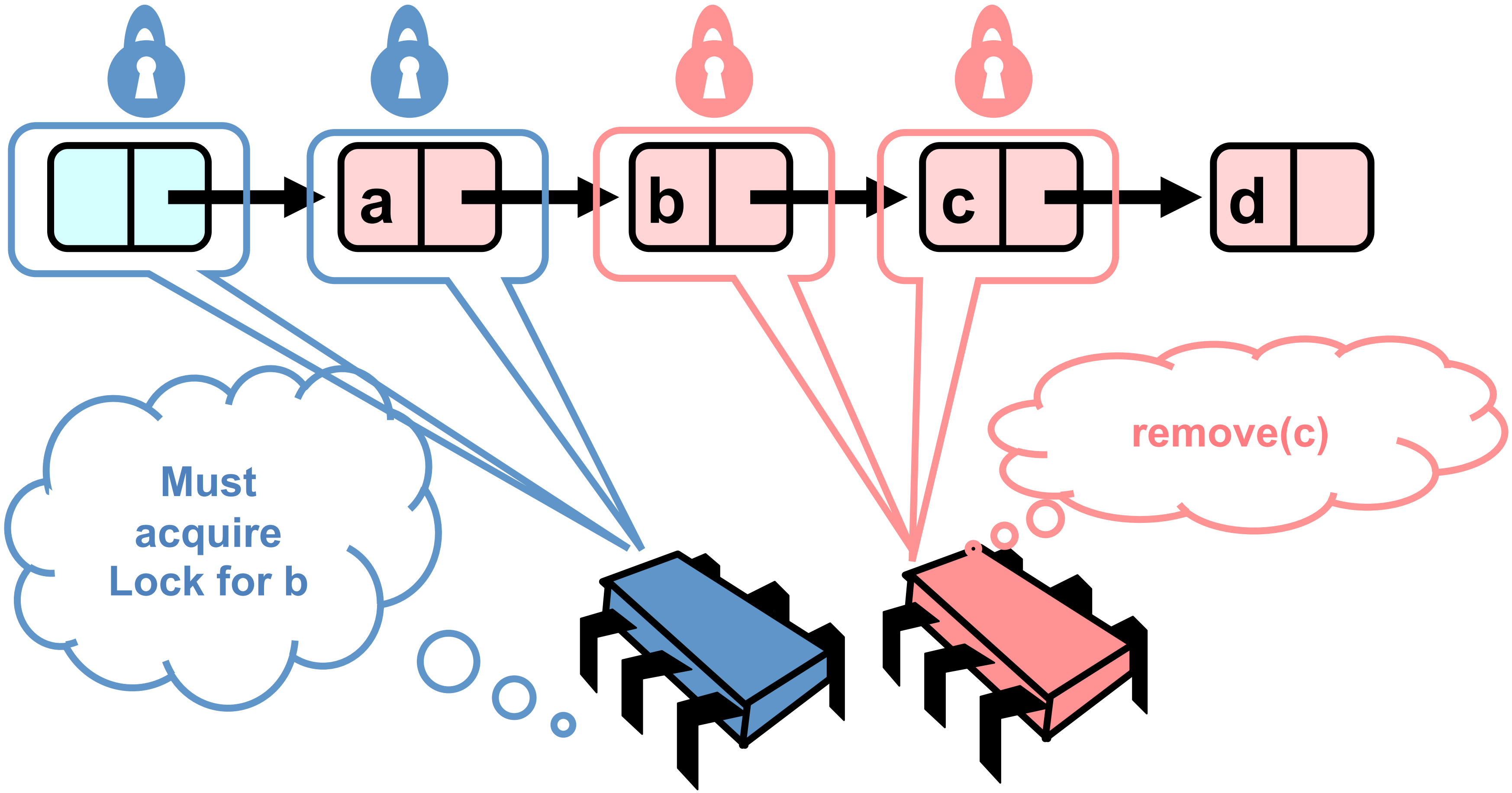
Removing a Node



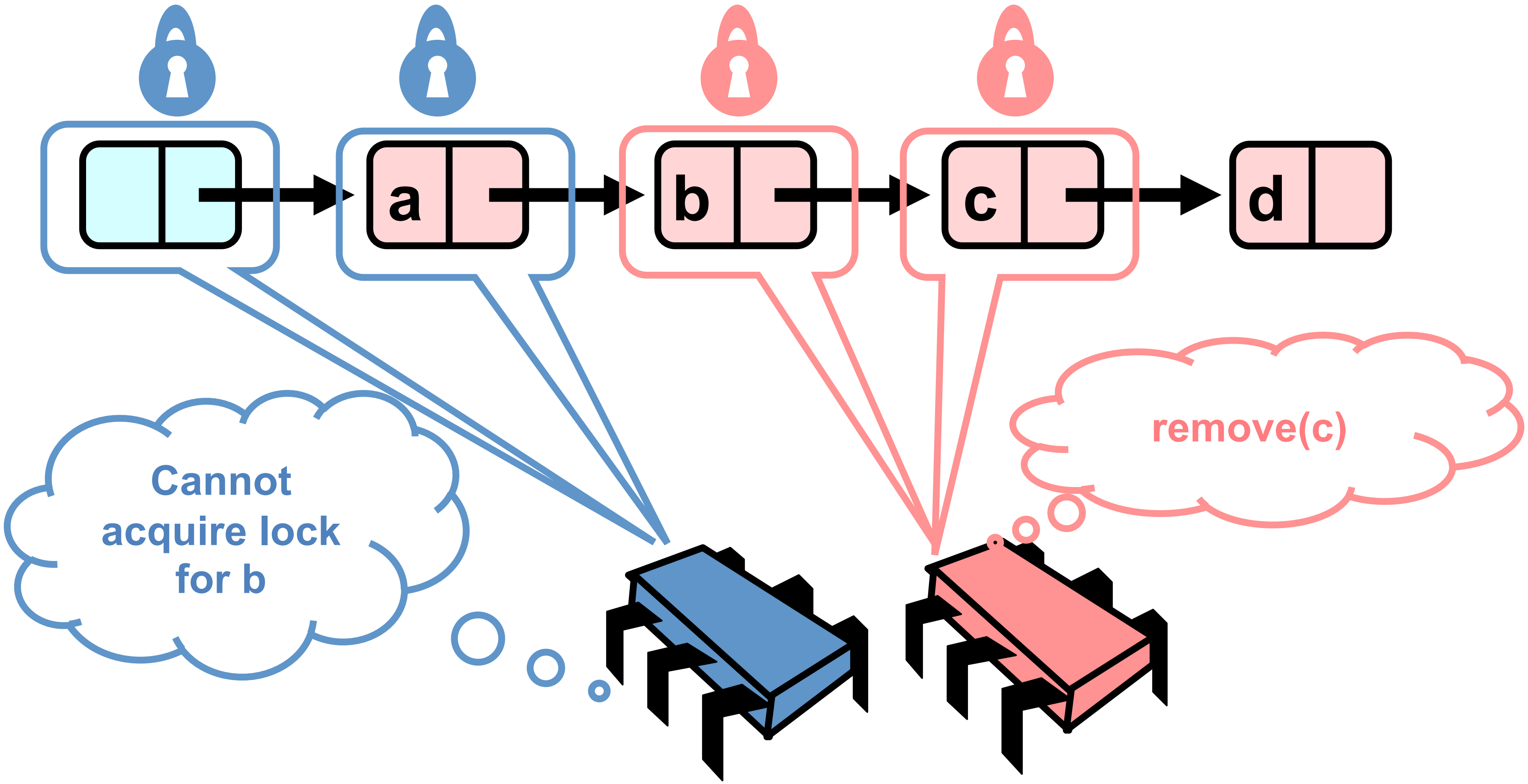
Removing a Node



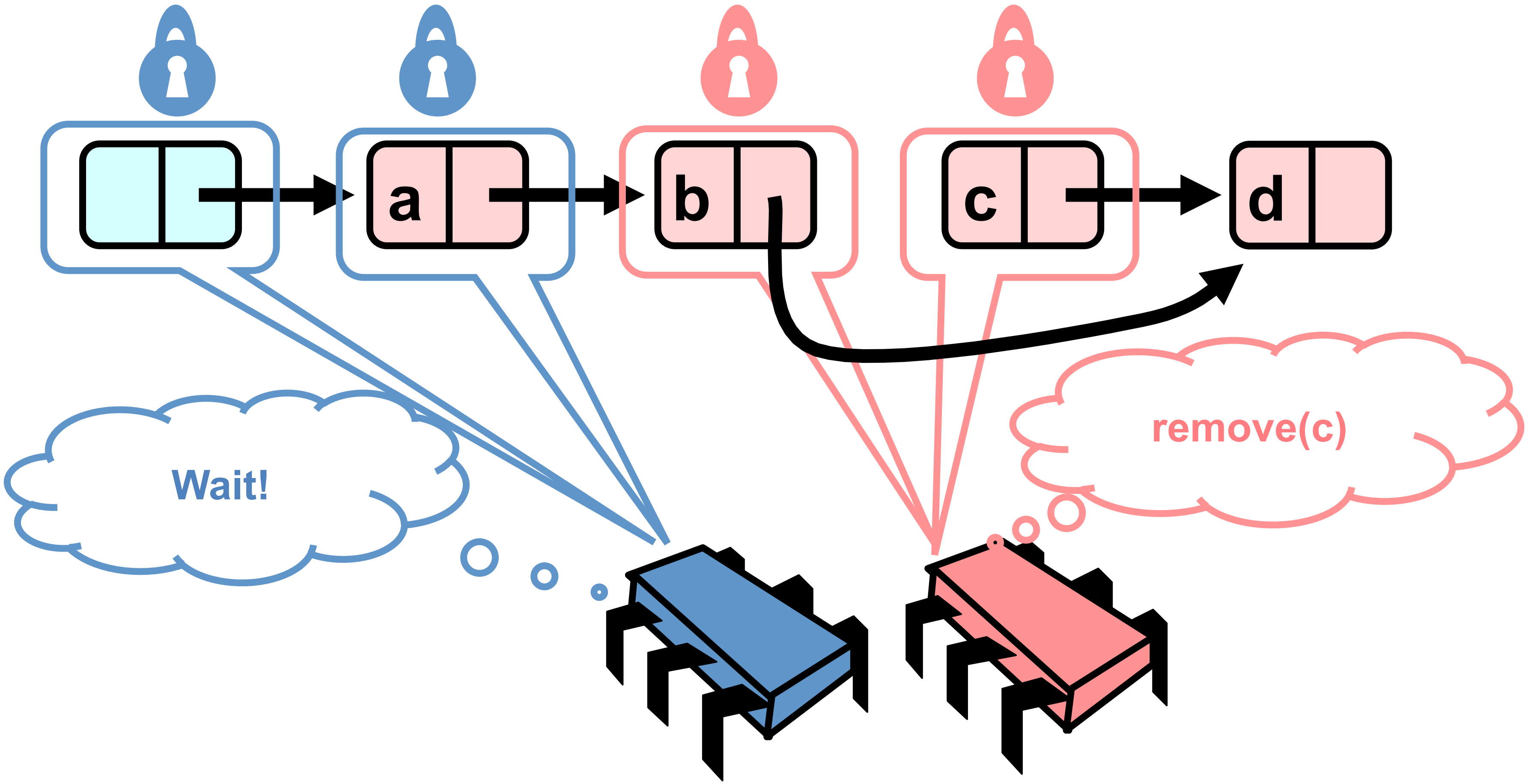
Removing a Node



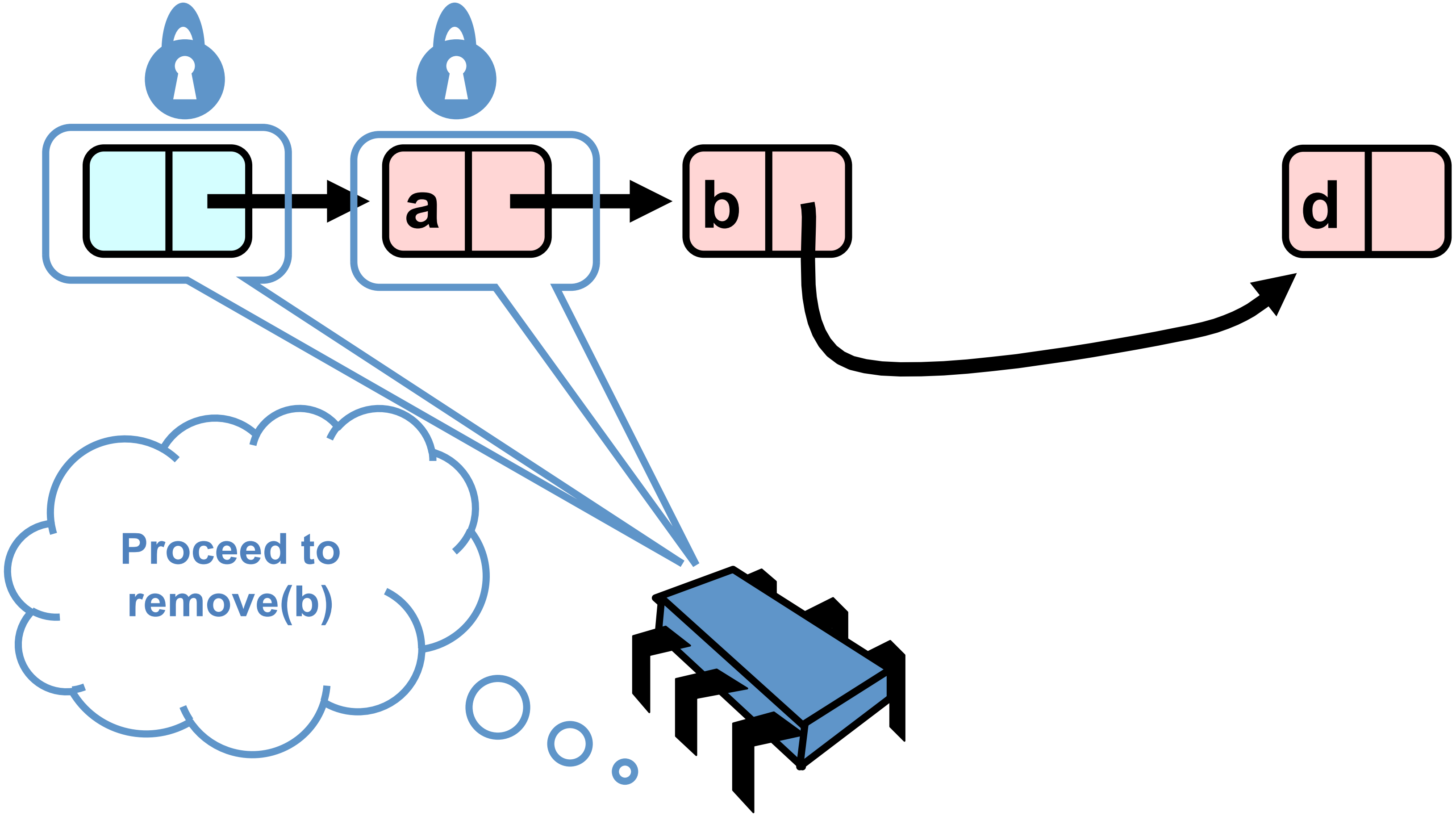
Removing a Node



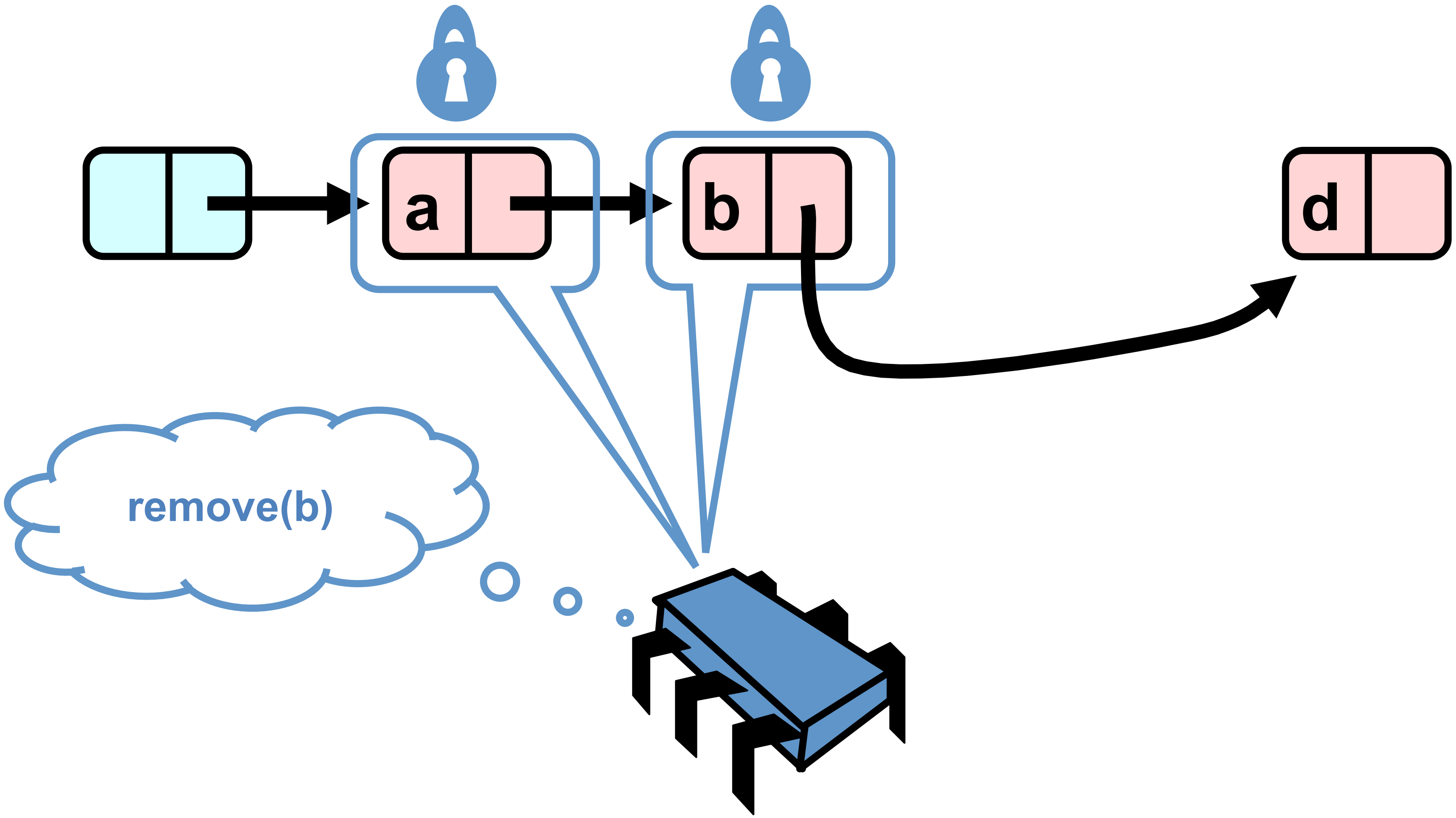
Removing a Node



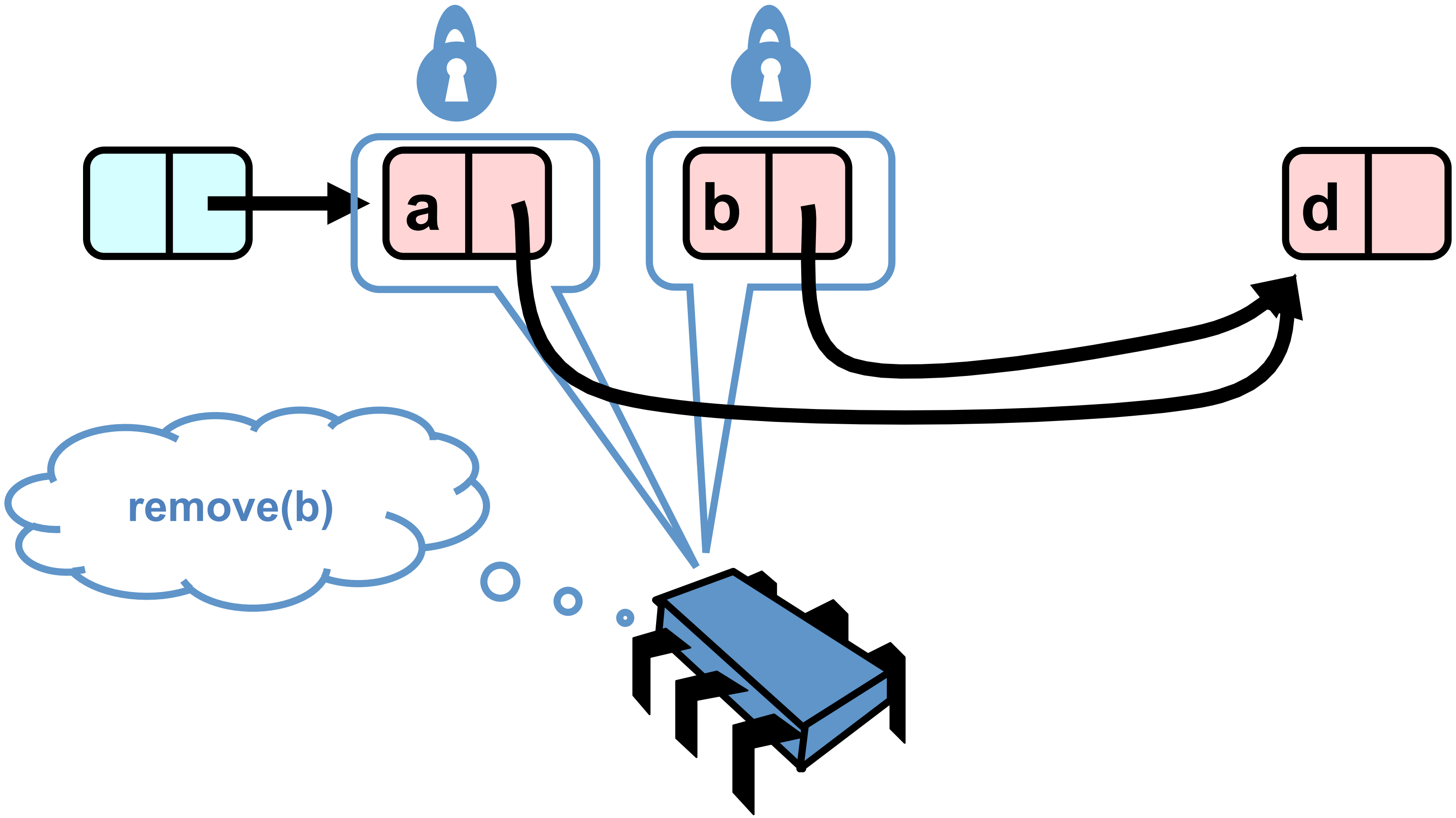
Removing a Node



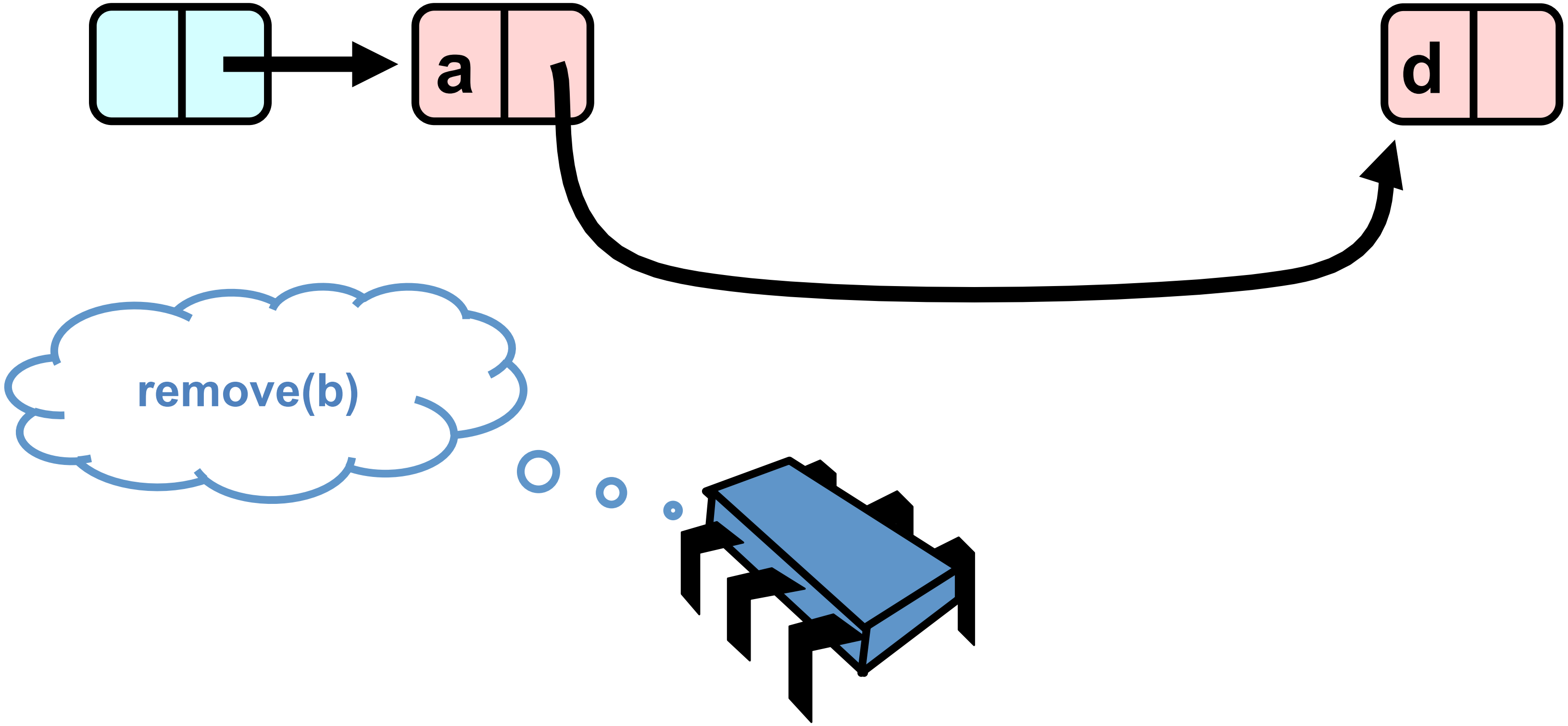
Removing a Node



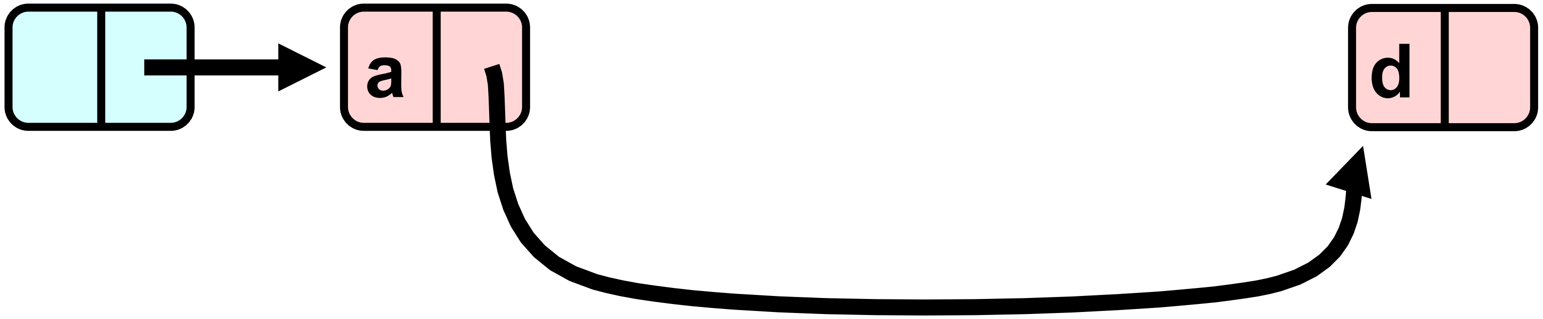
Removing a Node



Removing a Node



Removing a Node



Remove Method

```
public boolean remove(Item item) {
  int key = item.hashCode();
  Node pred, curr;
  try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();

 ...

  } finally {
   curr.unlock();
   pred.unlock();
  }
}
```

Start at the head and lock it

Lock the current node

Traverse the list and remove the item

On the next slide!

Make sure that the locks are released

Remove Method

```
while (curr.key <= key) {
  if (item == curr.item) {
    pred.next = curr.next;
      return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
}
return false;
```

Search key range

If item found, remove the node

Unlock pred and lock the next node

Return false if the element is not present

Linearization Points

```
while (curr.key <= key) {
  if (item == curr.item) {
    pred.next = curr.next;
      return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
}
return false;
```

Linearization point if item is present

Linearization point if item not present

Why Does This Work?

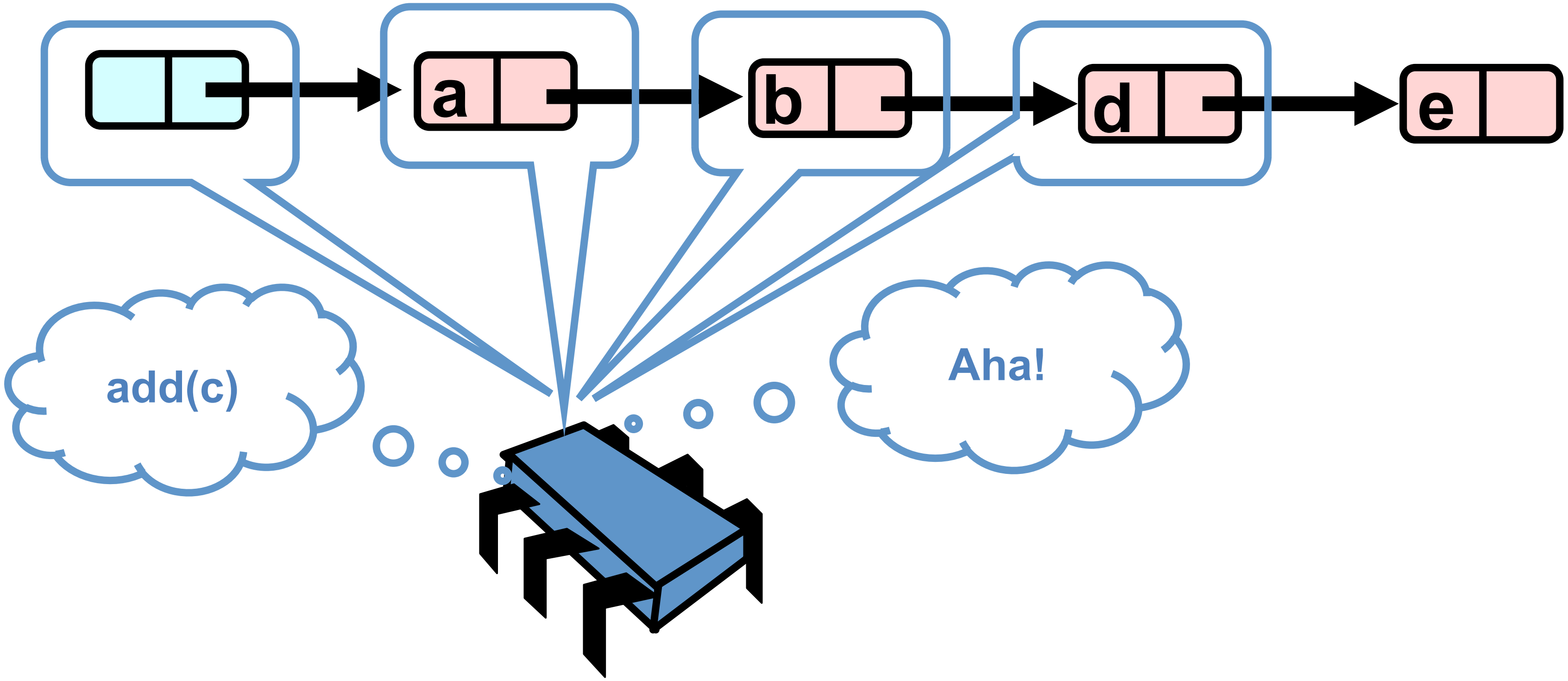
- To remove node *n*
  - Node *n* must be locked
  - Node *n*’s predecessor must be locked
- Therefore, if you lock a node
  - It cannot be removed
  - And neither can its successor

- To add node *n*
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted
  - Is the successor lock actually required?

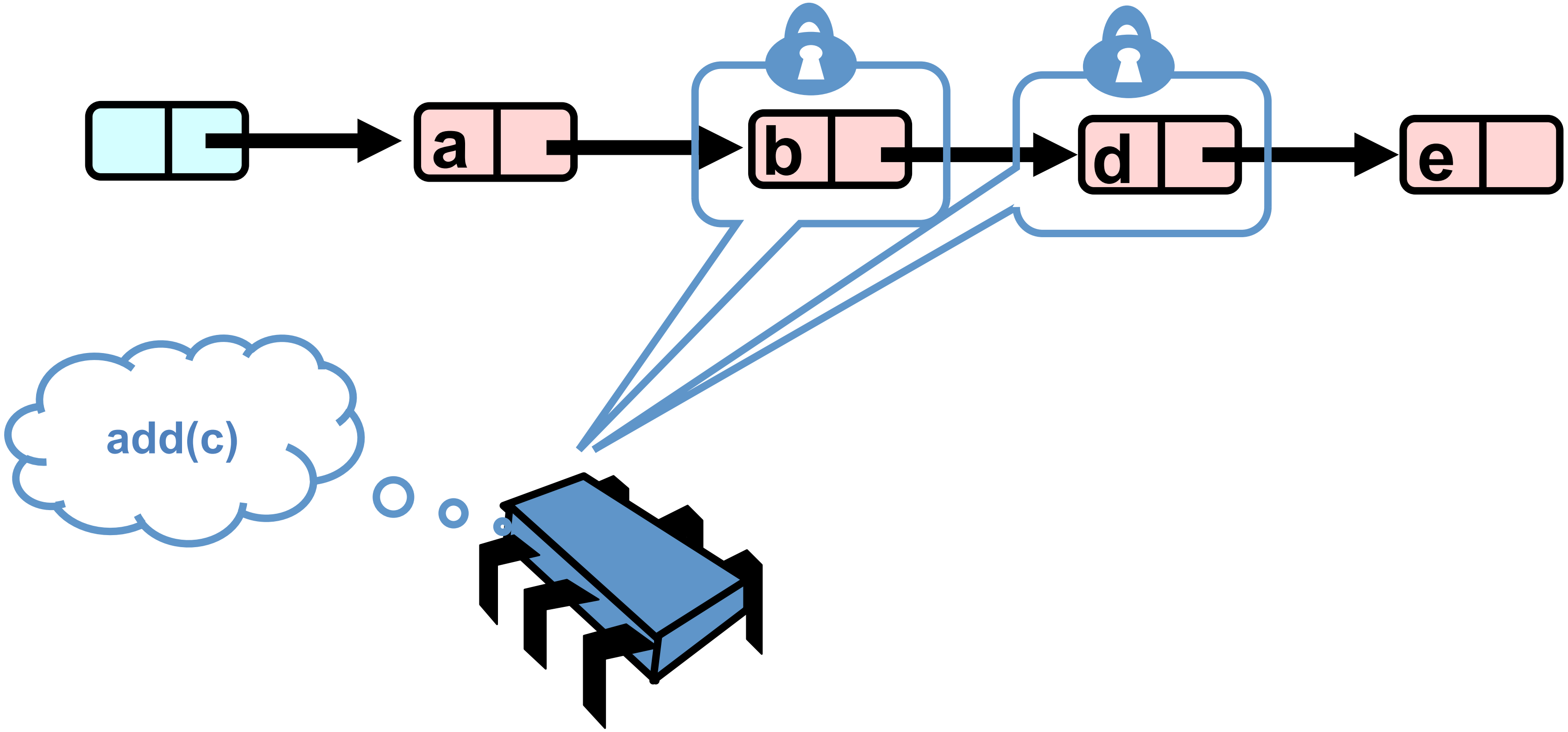
Drawbacks

- Hand-over-hand locking is sometimes better than coarse-grained lock
  - Threads can traverse in parallel
  - Sometimes, it's worse!
- However, it's certainly not ideal
  - Inefficient because many locks must be acquired and released
  - All methods use locks
  - Access to representation is pipelined
- How can we do better?

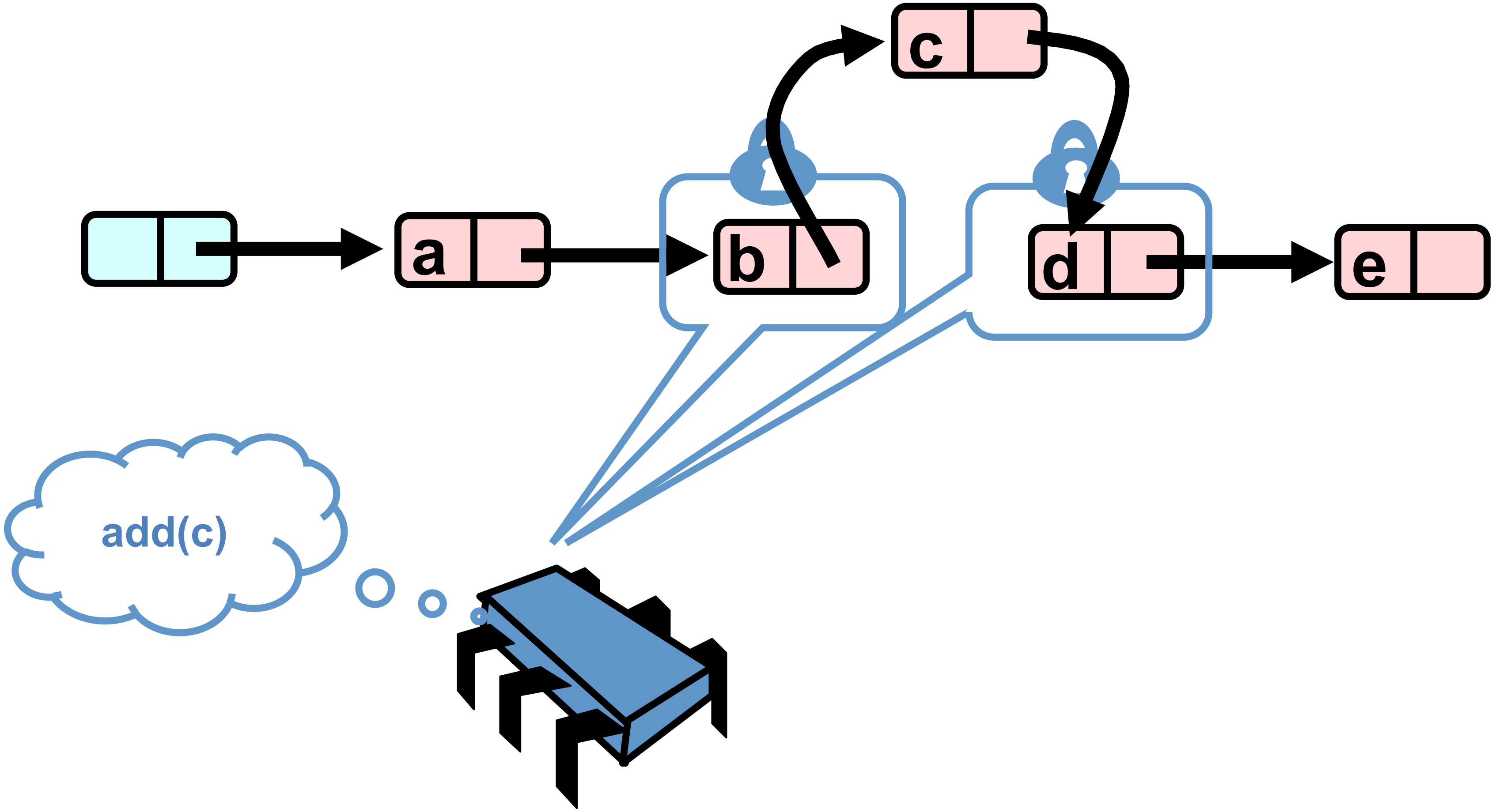
Optimistic: Traverse without Locking



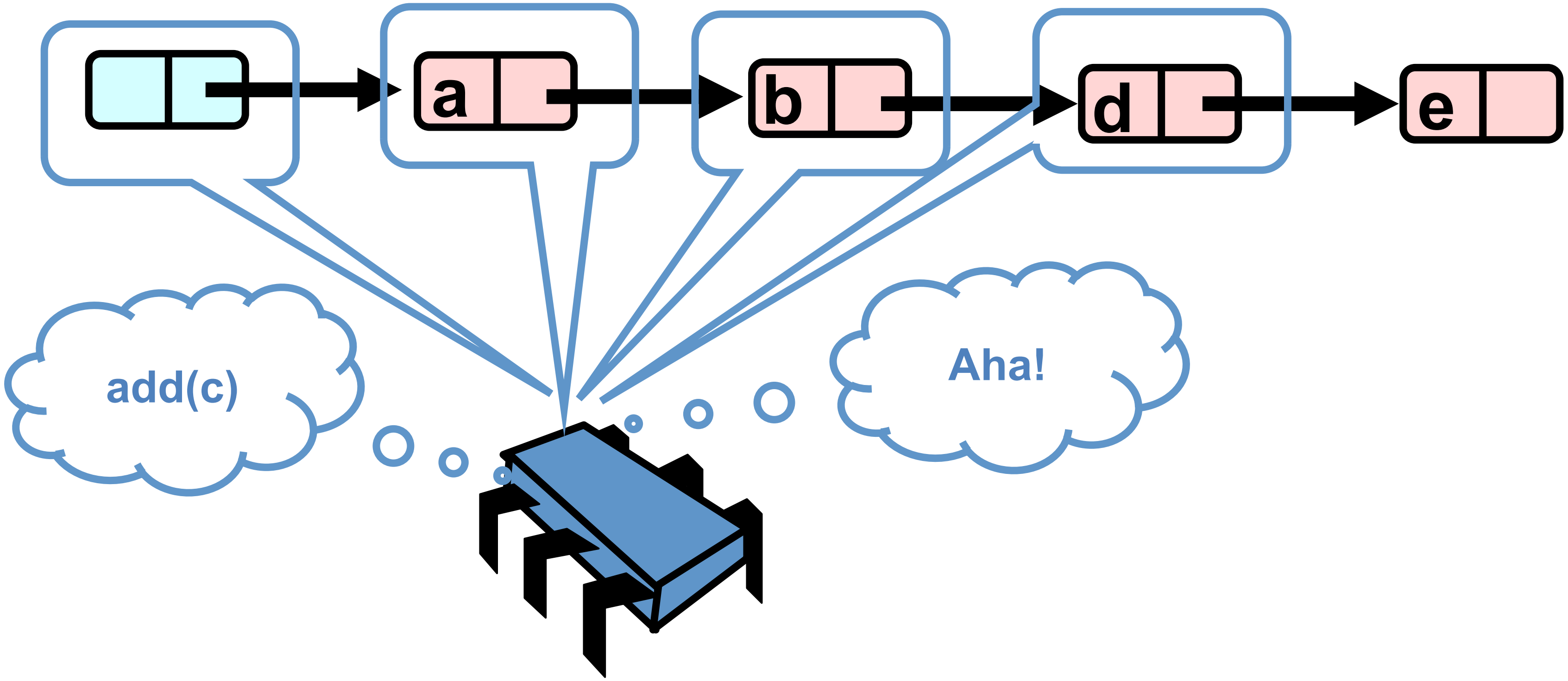
Optimistic: Lock and Load



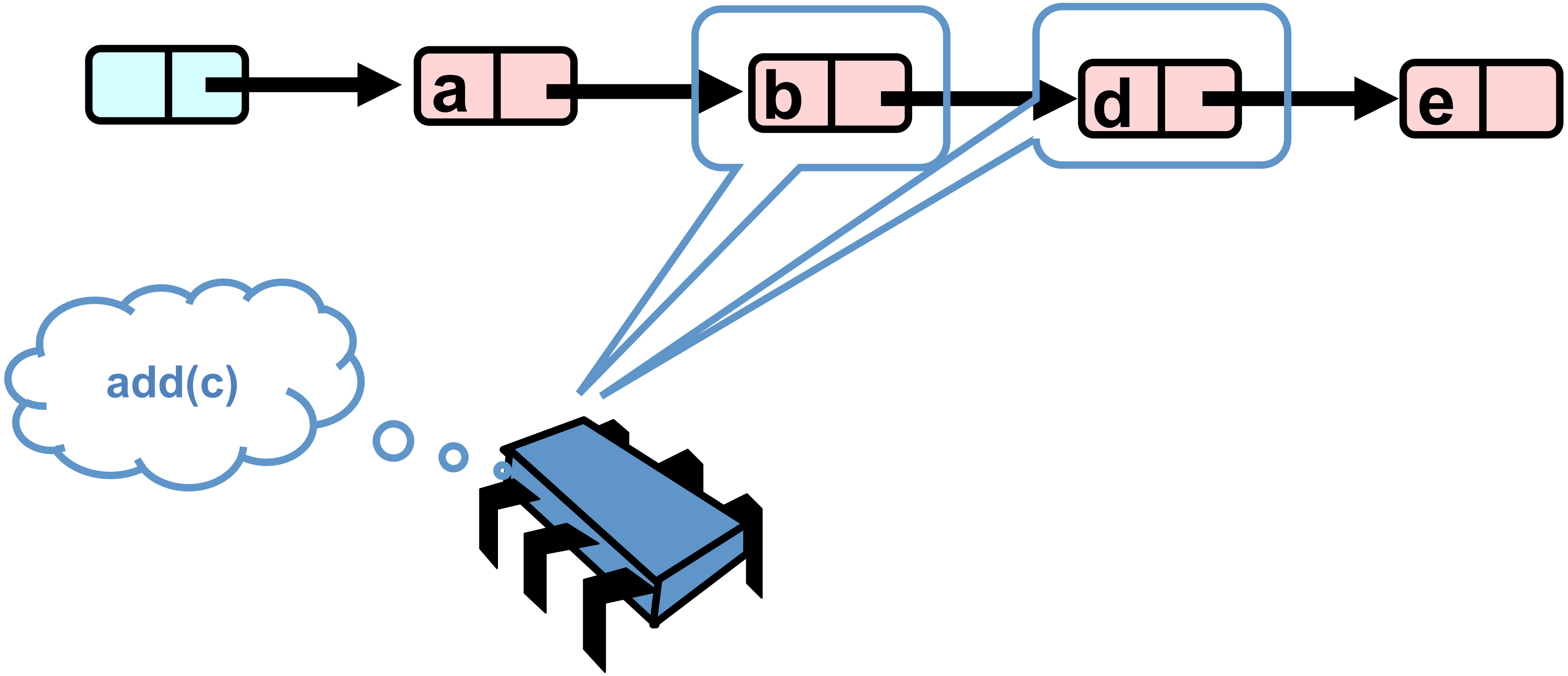
Optimistic: Lock and Load



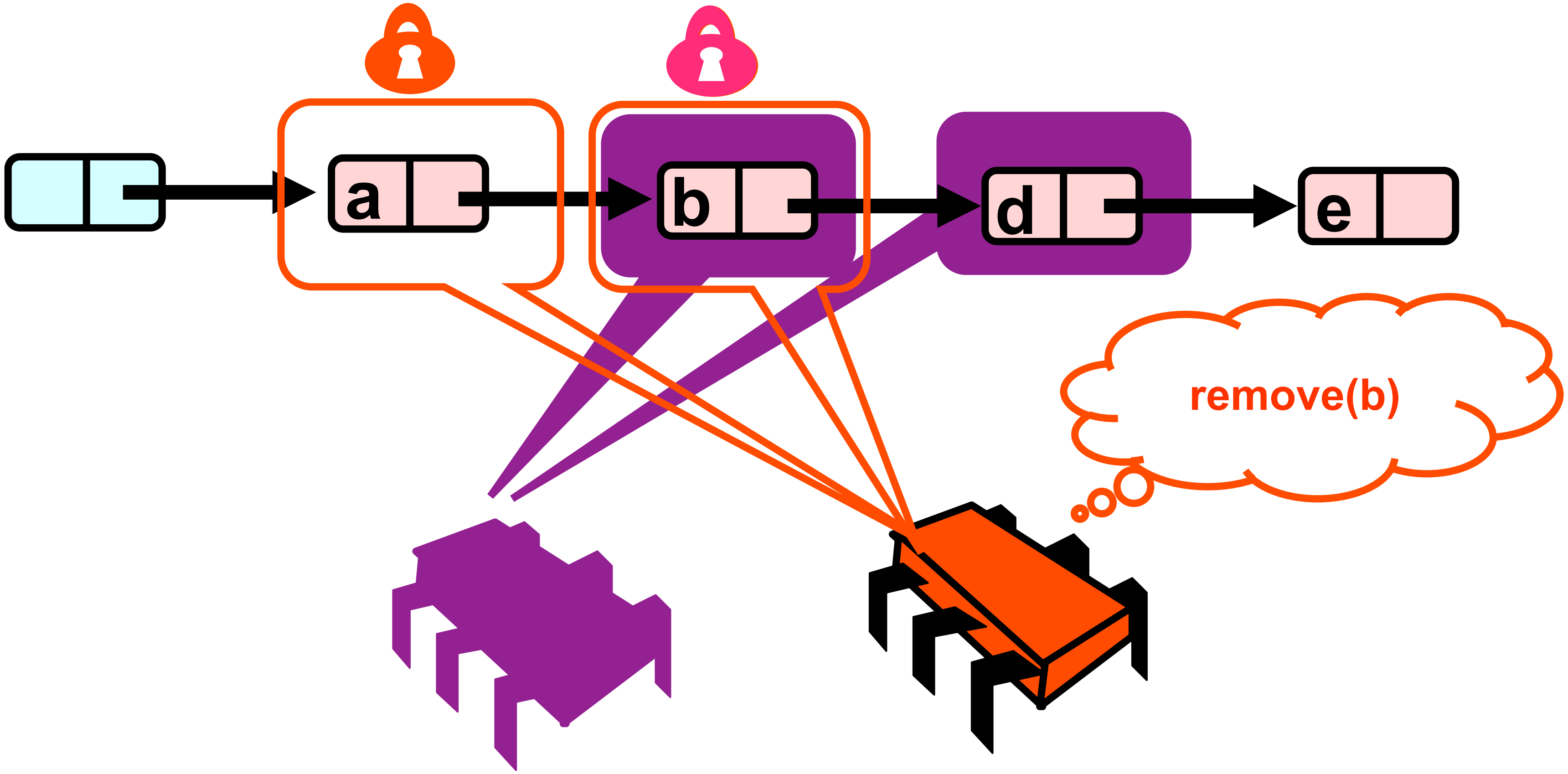
What could go wrong?



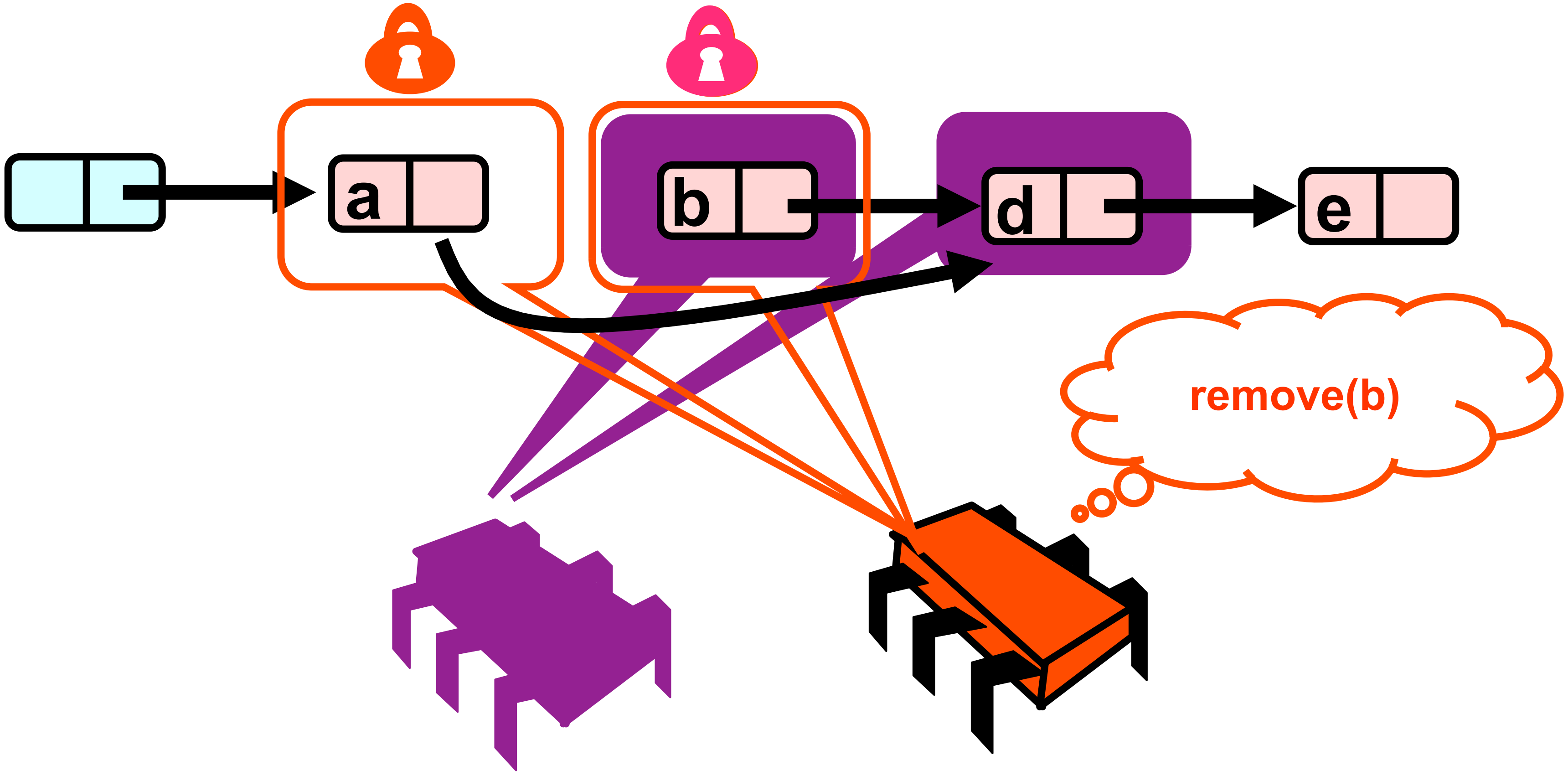
What could go wrong?



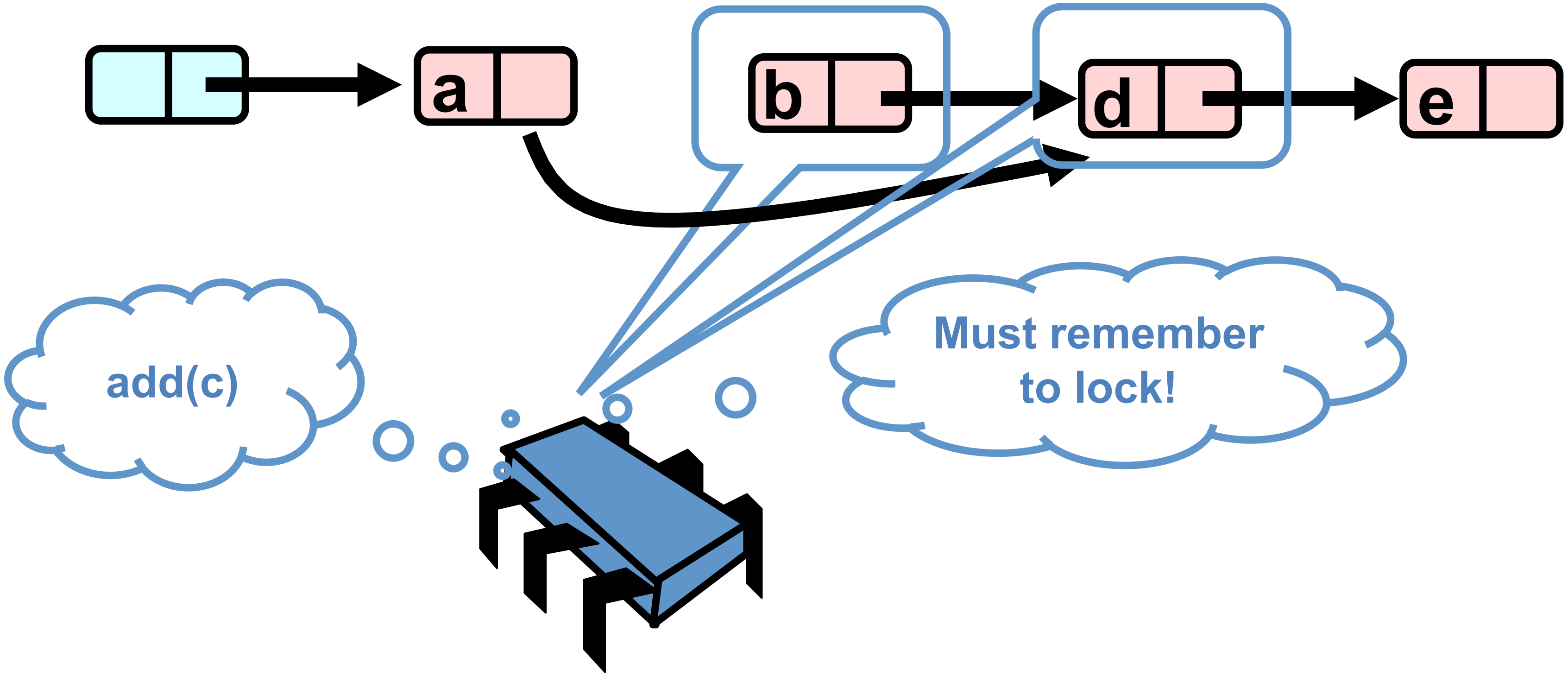
What could go wrong?



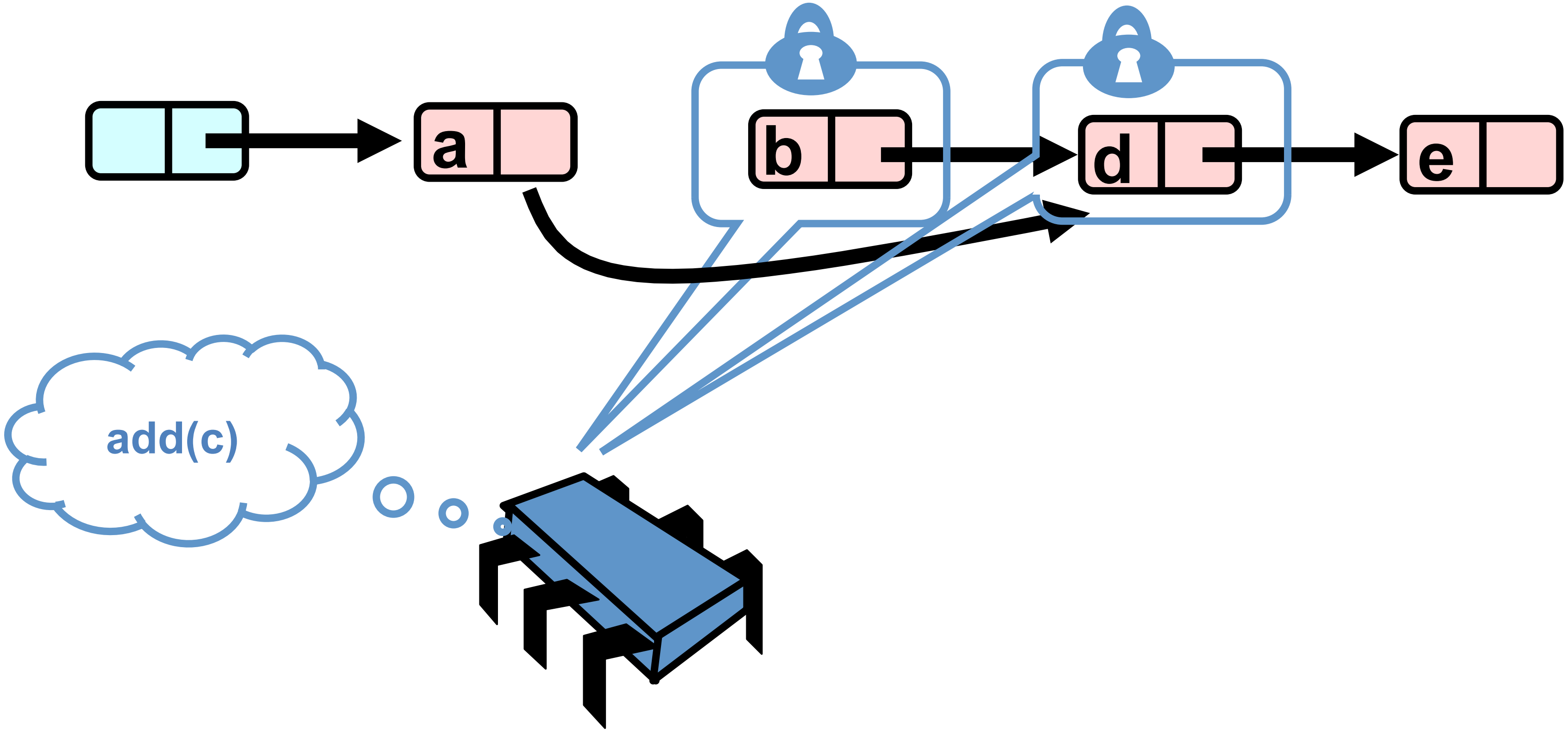
What could go wrong?



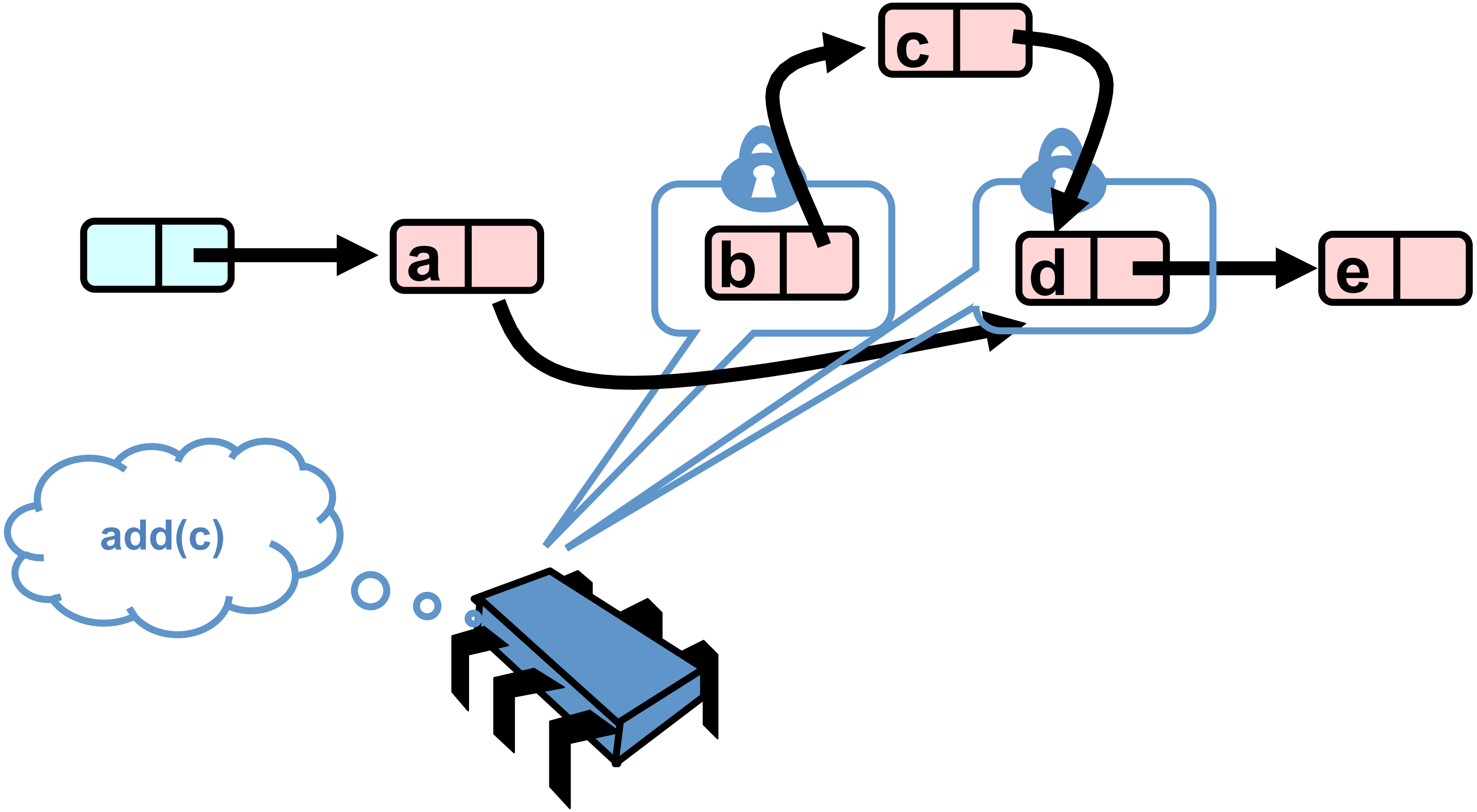
What could go wrong?



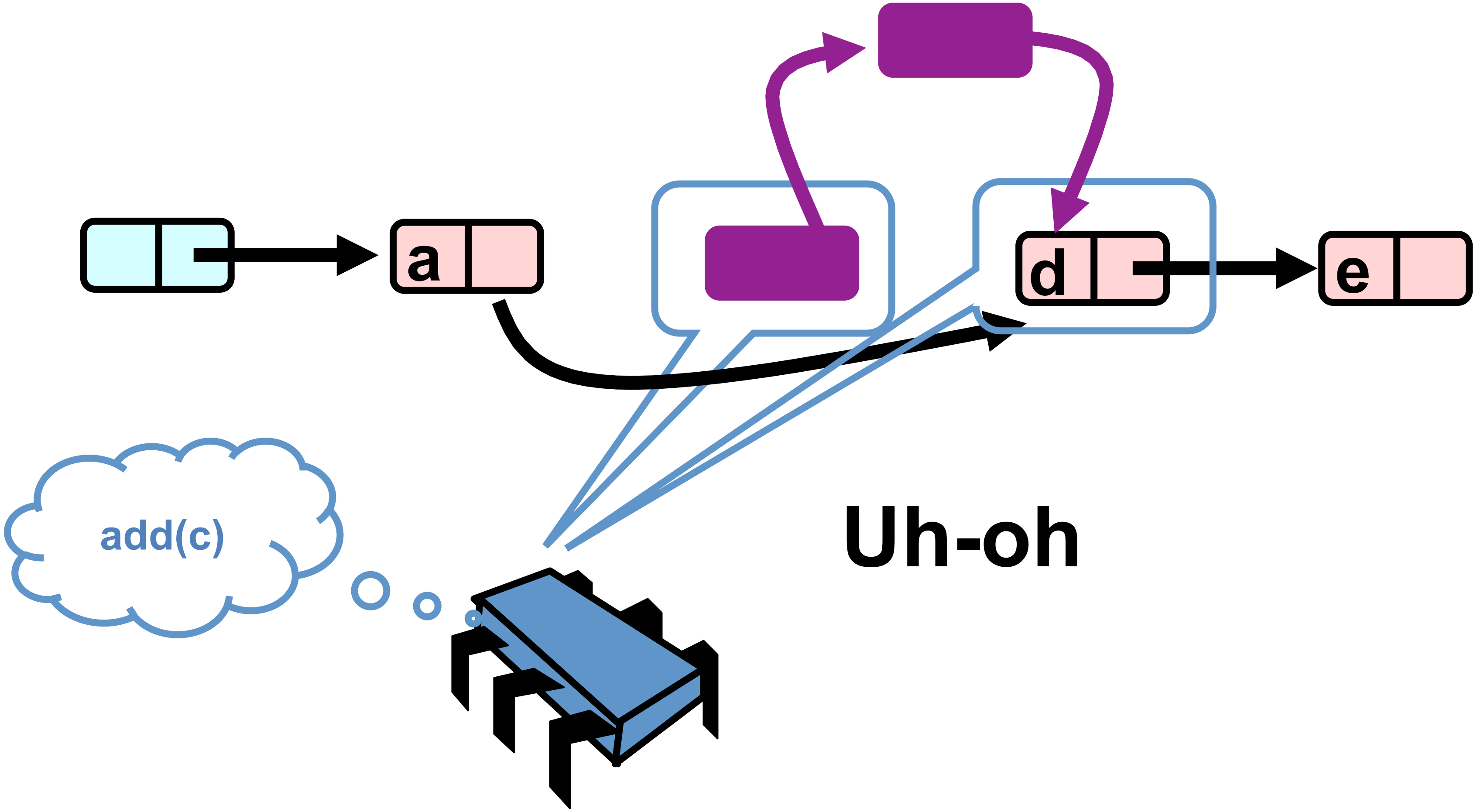
What could go wrong?



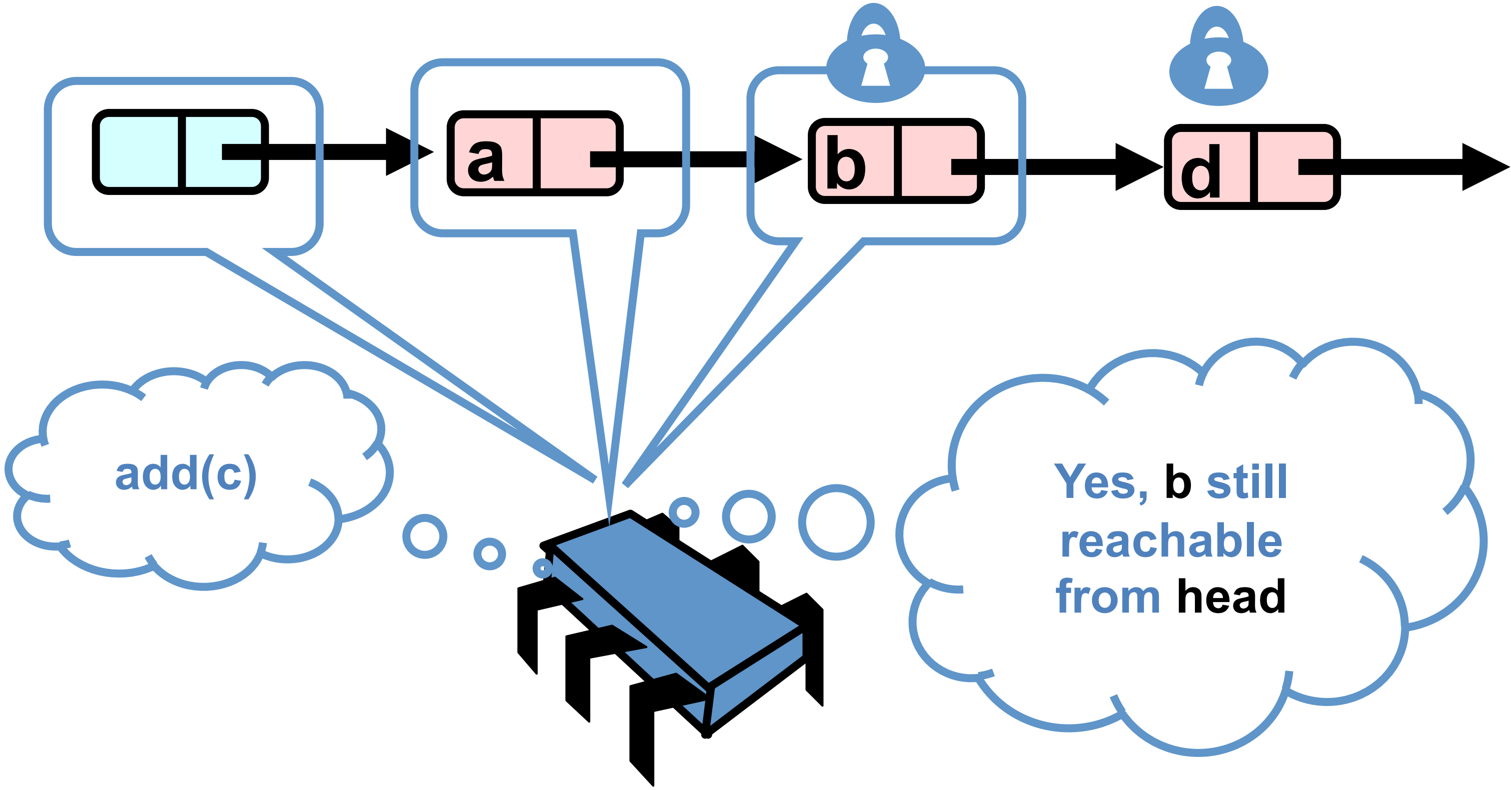
What could go wrong?



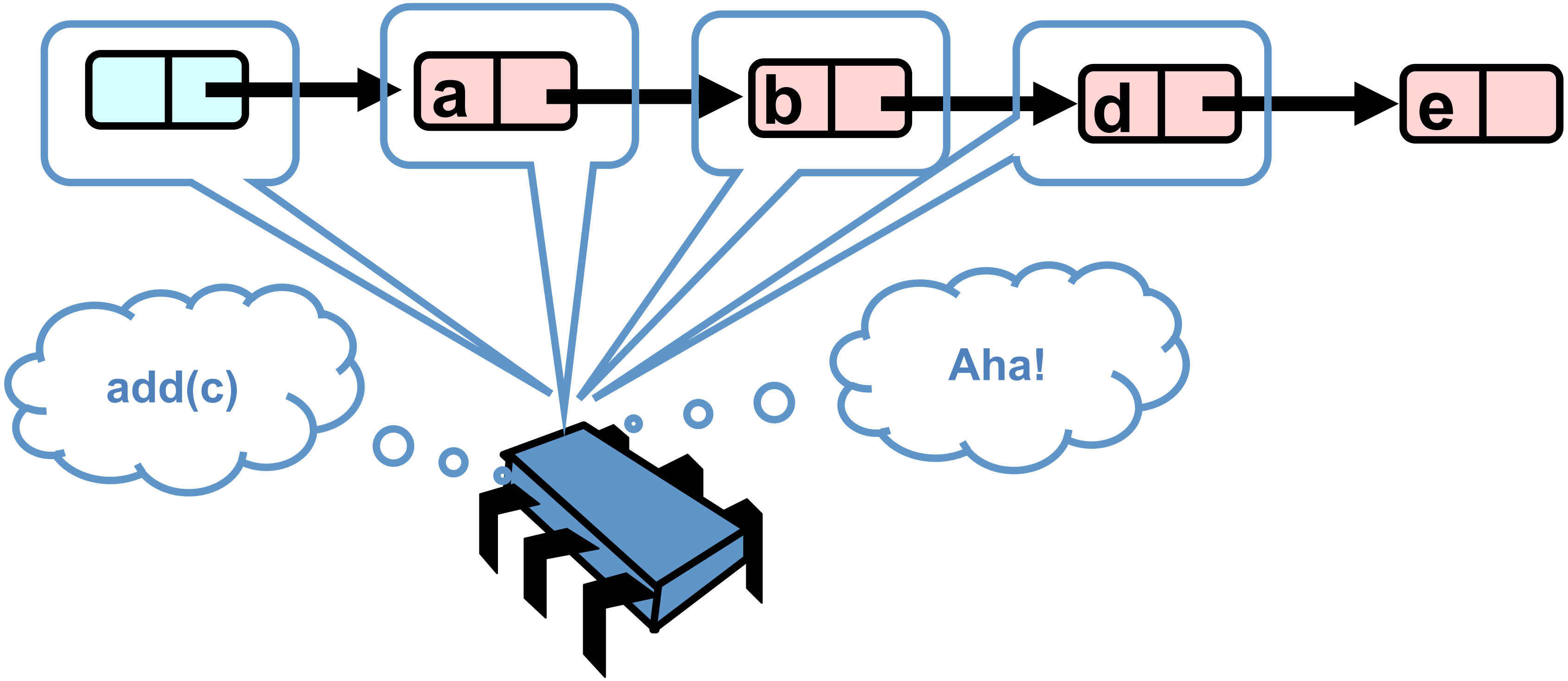
What could go wrong?



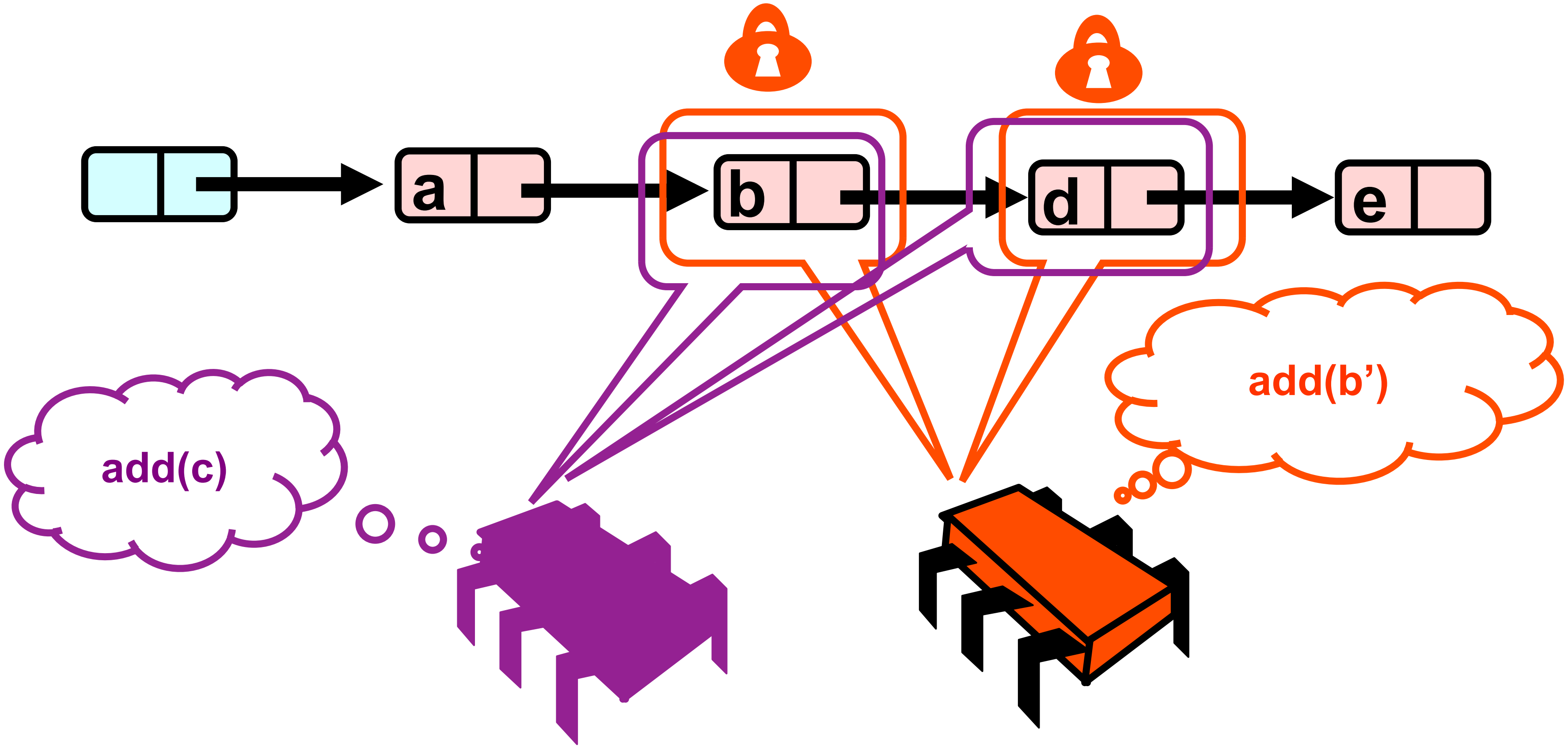
Validate – Part 1



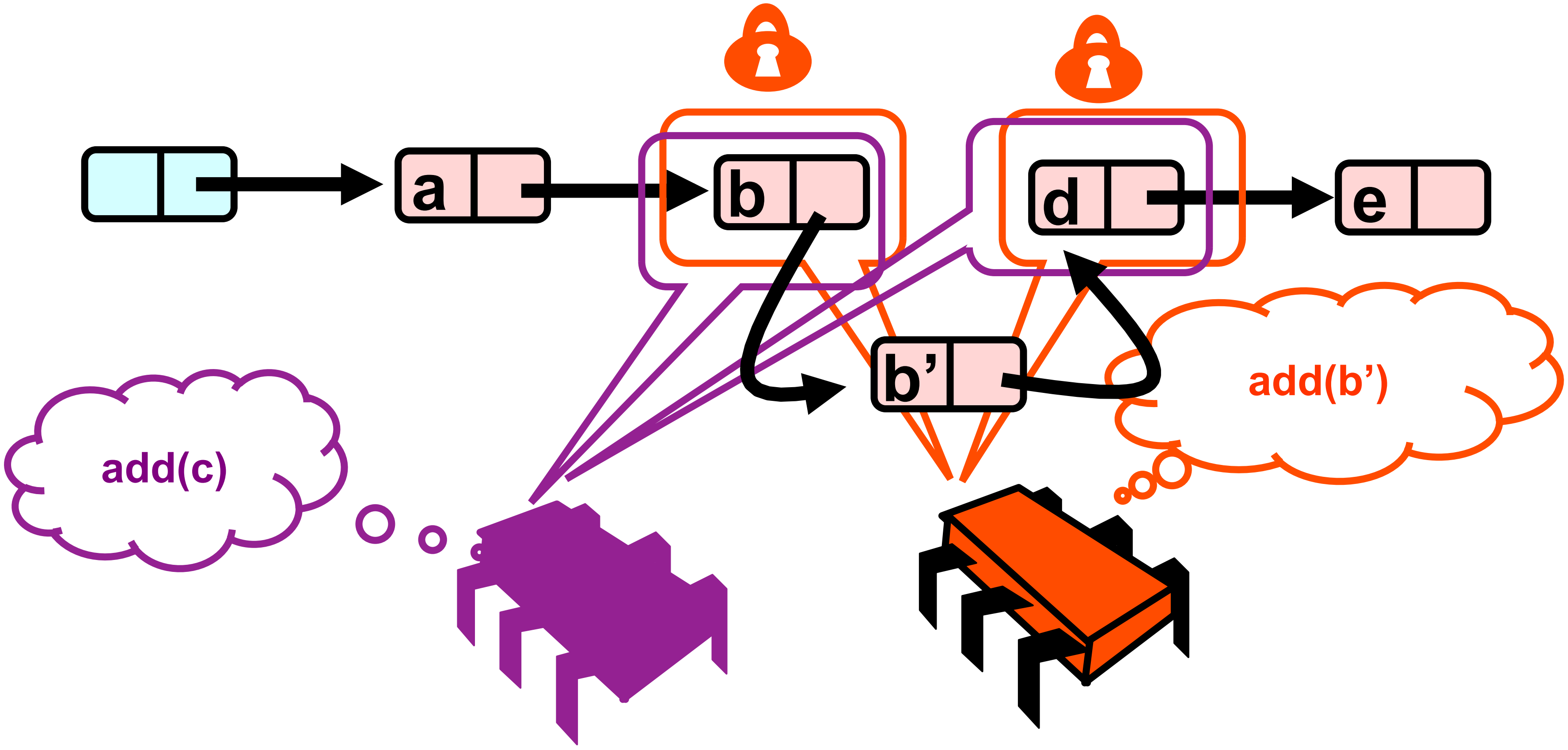
What Else Could Go Wrong?



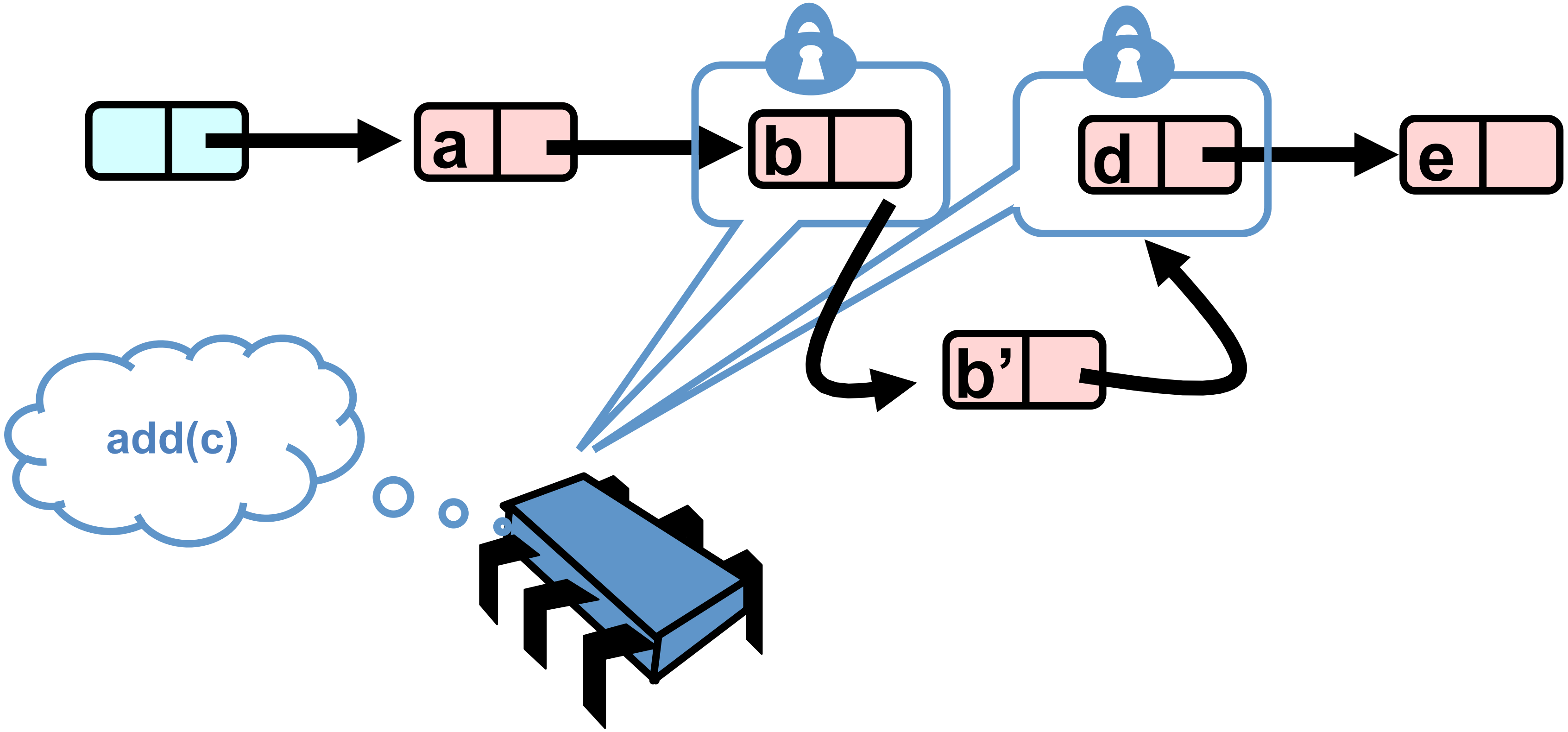
What Else Could Go Wrong?



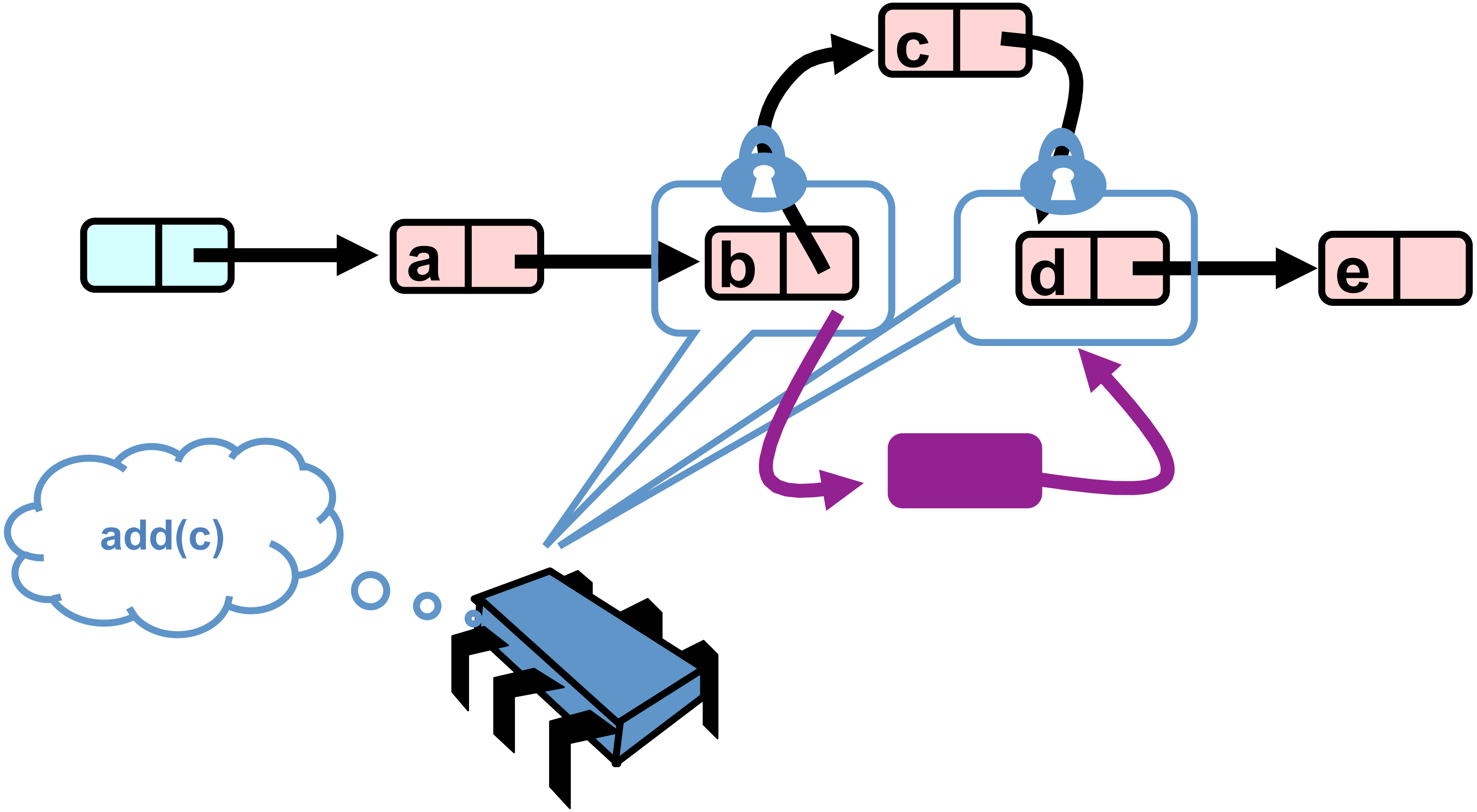
What Else Could Go Wrong?



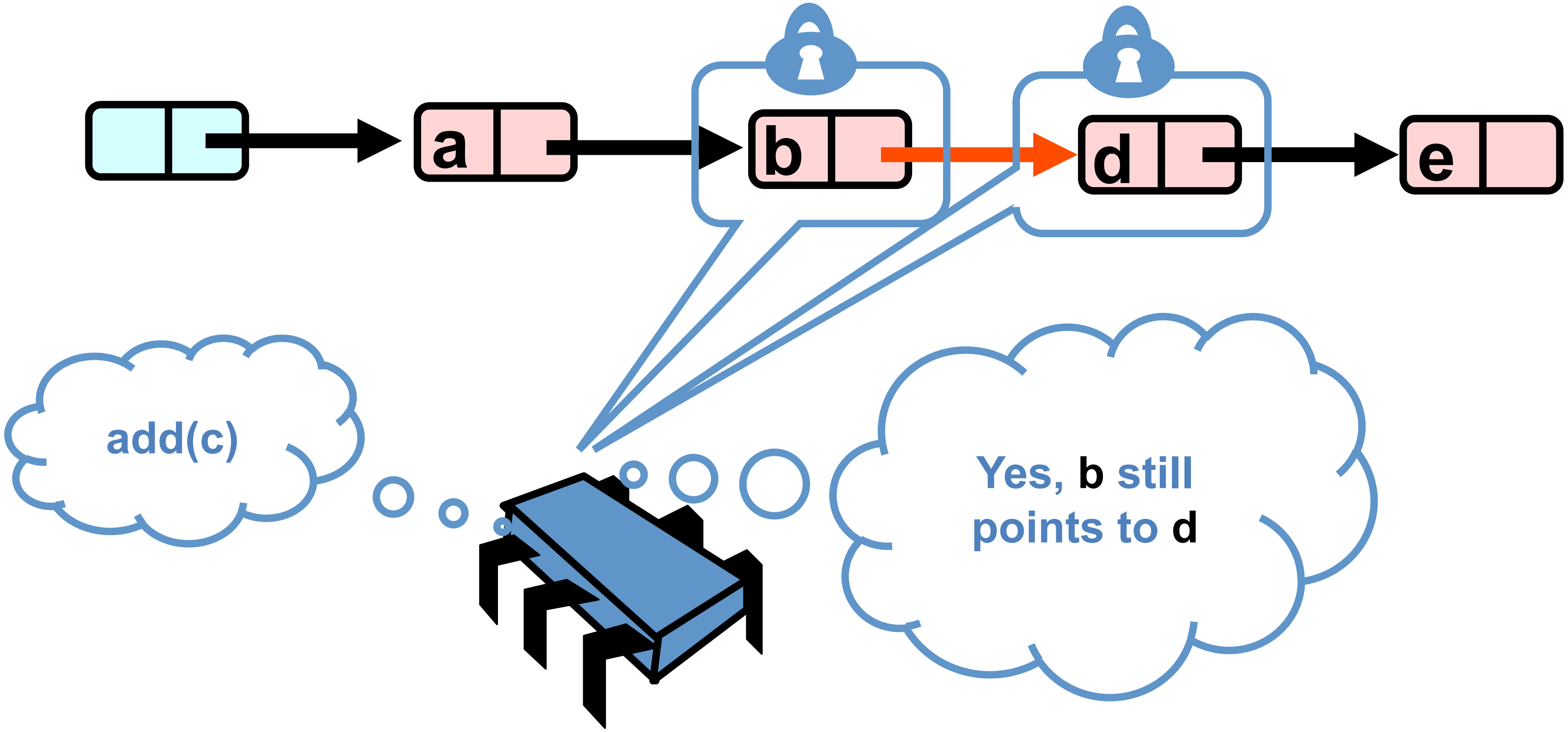
What Else Could Go Wrong?



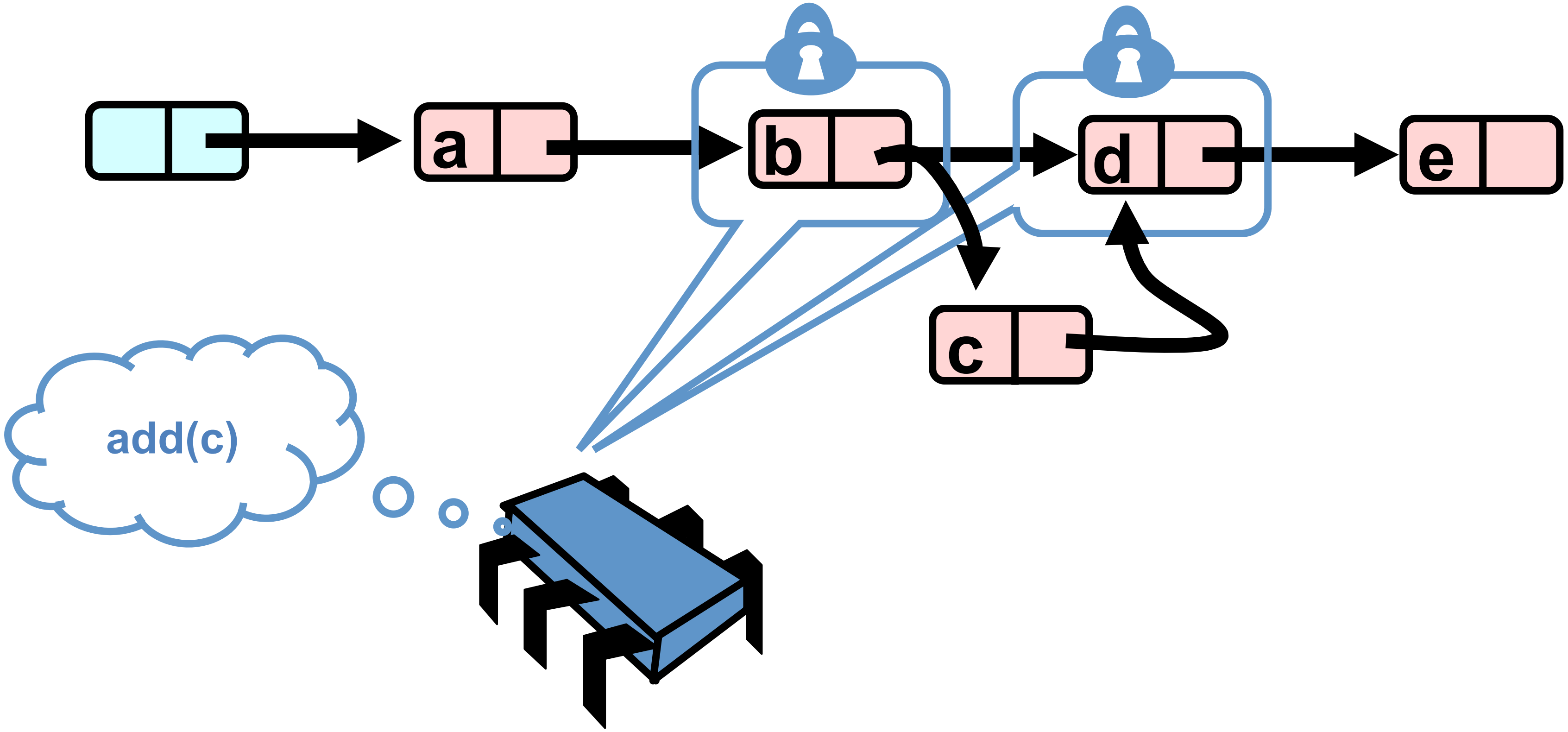
What Else Could Go Wrong?



Validate Part 2 (while holding locks)



Optimistic: Linearization Point



Correctness

- Careful: we may traverse deleted nodes
- But we establish properties by
  - Validation
  - After we lock target nodes
- If
  - Nodes b and c both locked
  - Node b still accessible
  - Node c still successor to b
- Then
  - Neither will be deleted
  - OK to delete and return true

Optimistic Synchronization: Validation

```
private boolean validate(Node pred,Node curr) {
  Node node = head;
  while (node.key <= pred.key) {
    if (node == pred)
    return pred.next == curr;
 node = node.next;
  }
  return false;
}
```

If pred is reached, test if the successor is curr

Predecessor not reachable

Optimistic Synchronization: Remove

```
private boolean remove(Item item) {
  int key = item.hashCode();
  while (true) {
    Node pred = this.head;
    Node curr = pred.next;
    while (curr.key <= key) {
       if (item == curr.item)
       break;
       pred = curr;
       curr = curr.next;
    }
 ...
```

Retry on synchronization conflict – validate fails

Stop if we find the item

On Exit from Loop

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

Optimistic Synchronization: Remove

```
  ...
  try {
    pred.lock(); curr.lock();
    if (validate(pred,curr)) {
      if (curr.item == item) {
        pred.next = curr.next;
        return true;
      } else {
        return false;
      }
    }
  } finally {
    pred.unlock();
    curr.unlock();
  }
 }
}
```

Lock both nodes

Check for synchronization conflicts

Remove node if target found

Always unlock the nodes

Optimistic List

- Limited hot-spots
  - Targets of **`add()`**, **`remove()`**, **`contains()`**
  - No contention on traversals
- Moreover
  - Traversals are wait-free
  - Food for thought …
- Much less lock acquisition/release
  - Performance
  - Concurrency
- Problems
  - Need to traverse list twice
  - **`contains()`** acquires locks

90% of calls in many apps!

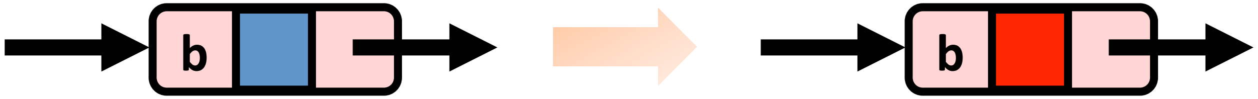
Lazy List

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

Lazy List

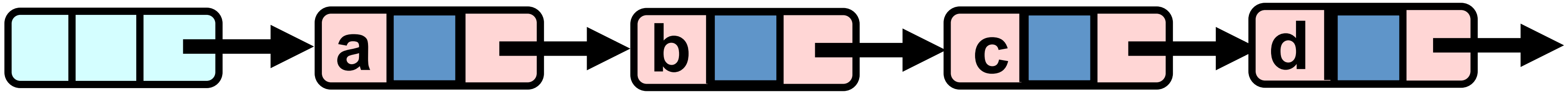
- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”

- How can we remove nodes “lazily”?
  - First perform a logical delete: Mark current node as removed (new!)

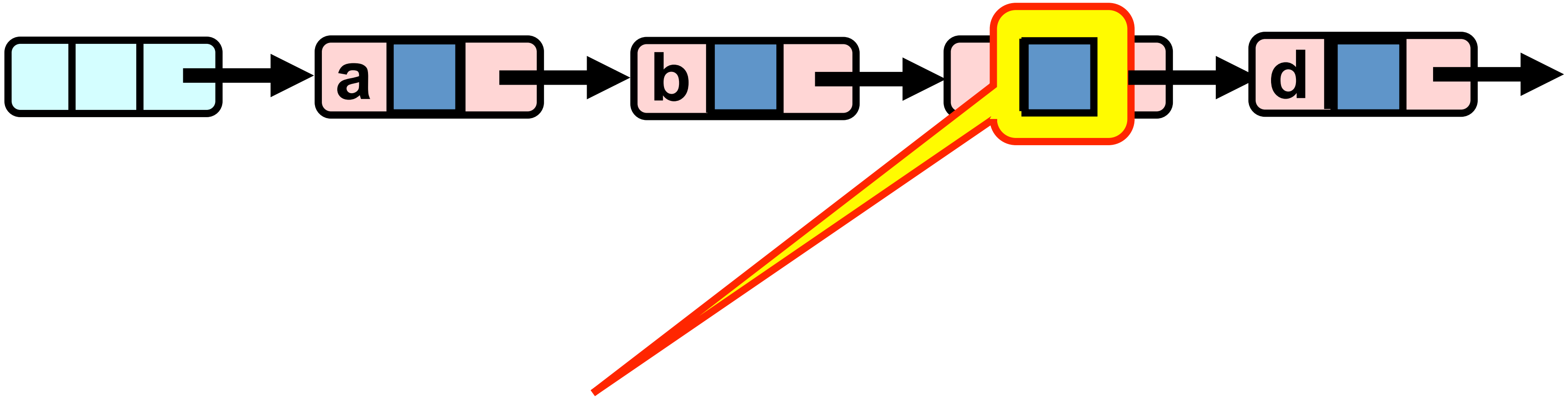


  - Then perform a physical delete: Redirect predecessor’s next (as before)
  - Logically deleted nodes still hang around!

Lazy Removal

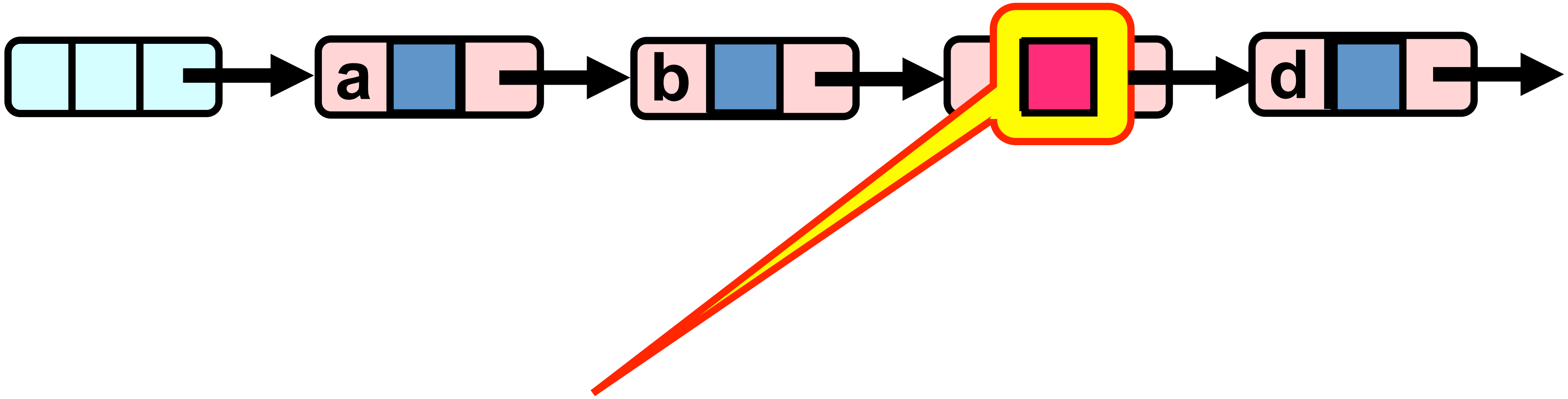


Lazy Removal



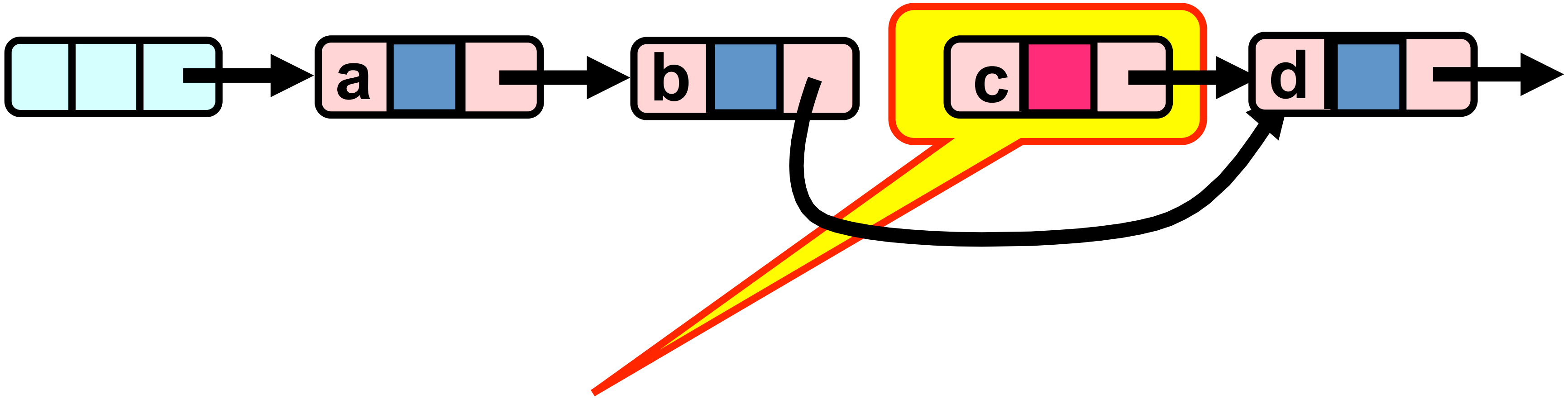
Present in list

Lazy Removal



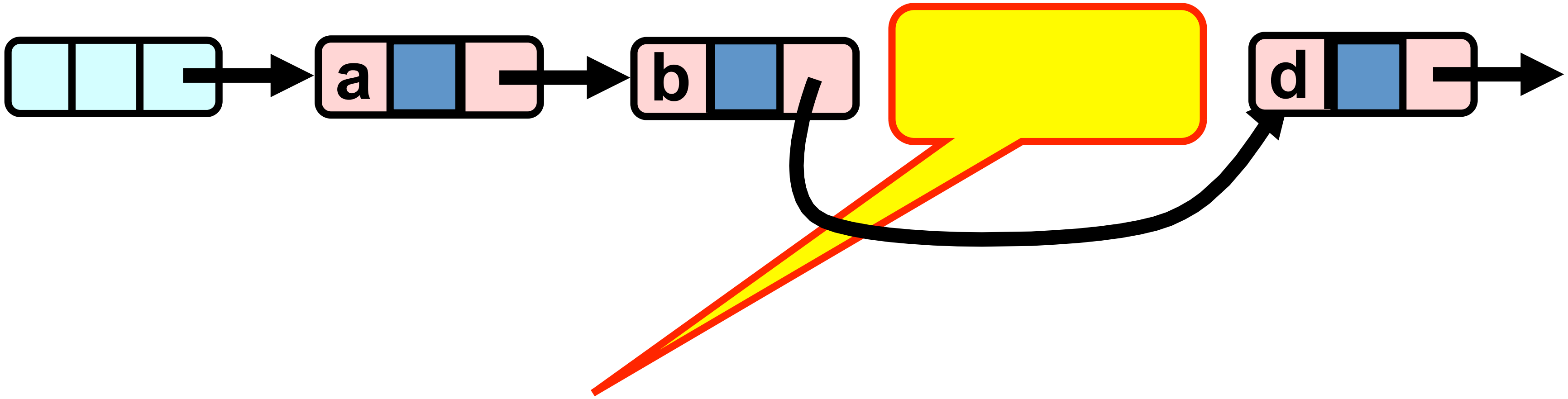
Logically deleted

Lazy Removal



Deleted from list of reachable elements

Lazy Removal

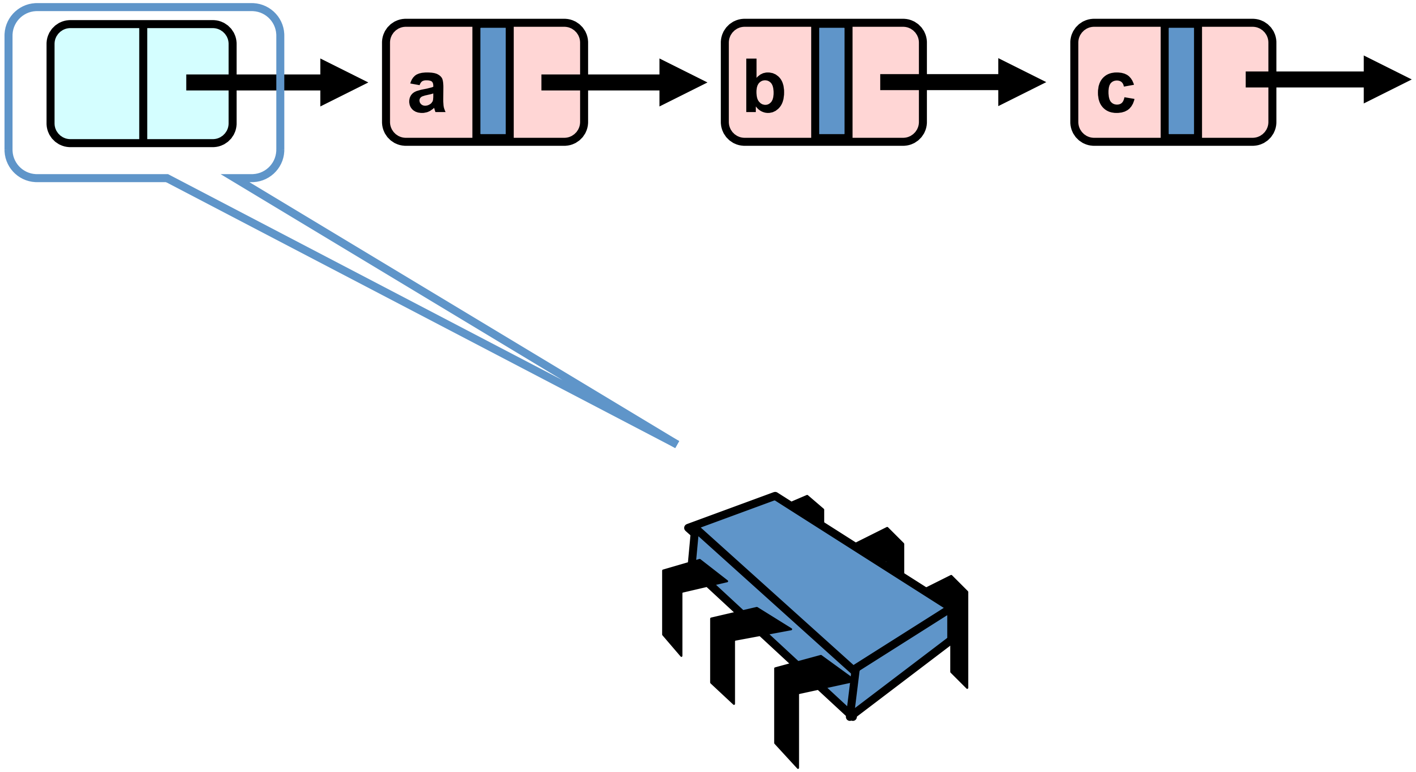


Garbage collected when all references used up

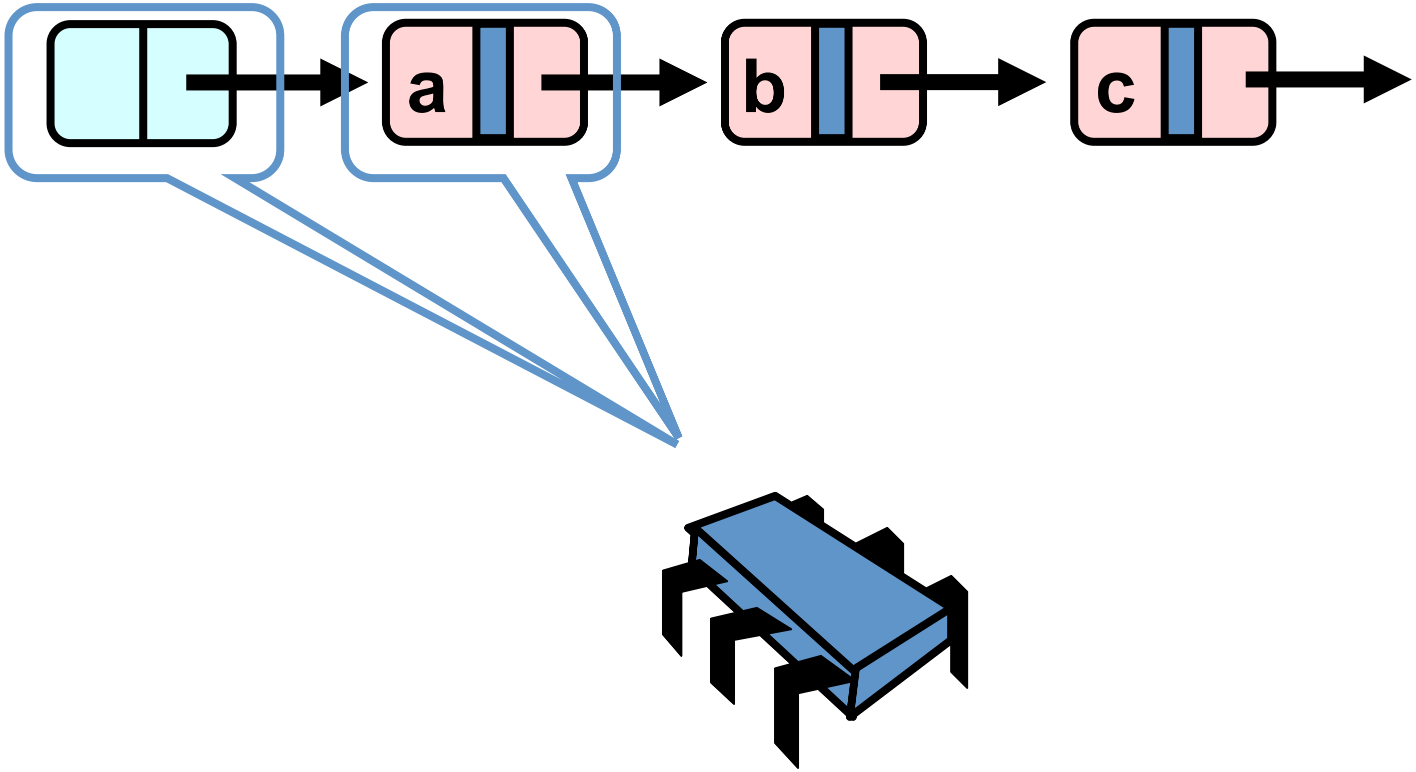
Lazy Synchronization

- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn’t slow down other method calls…
- Note that we must still lock pred and curr nodes!
- Validation:
  - Check that neither pred nor curr are marked
  - Check that pred points to curr

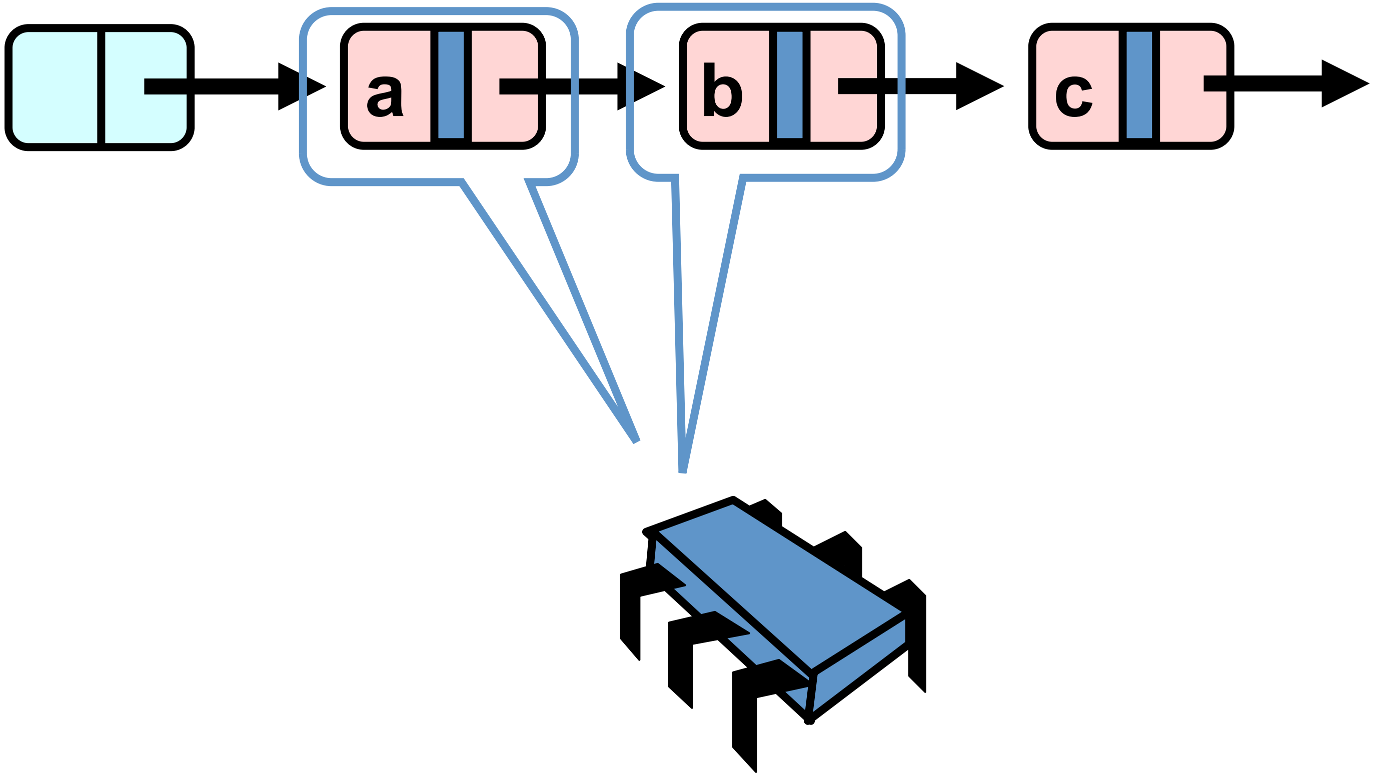
Lazy Removal



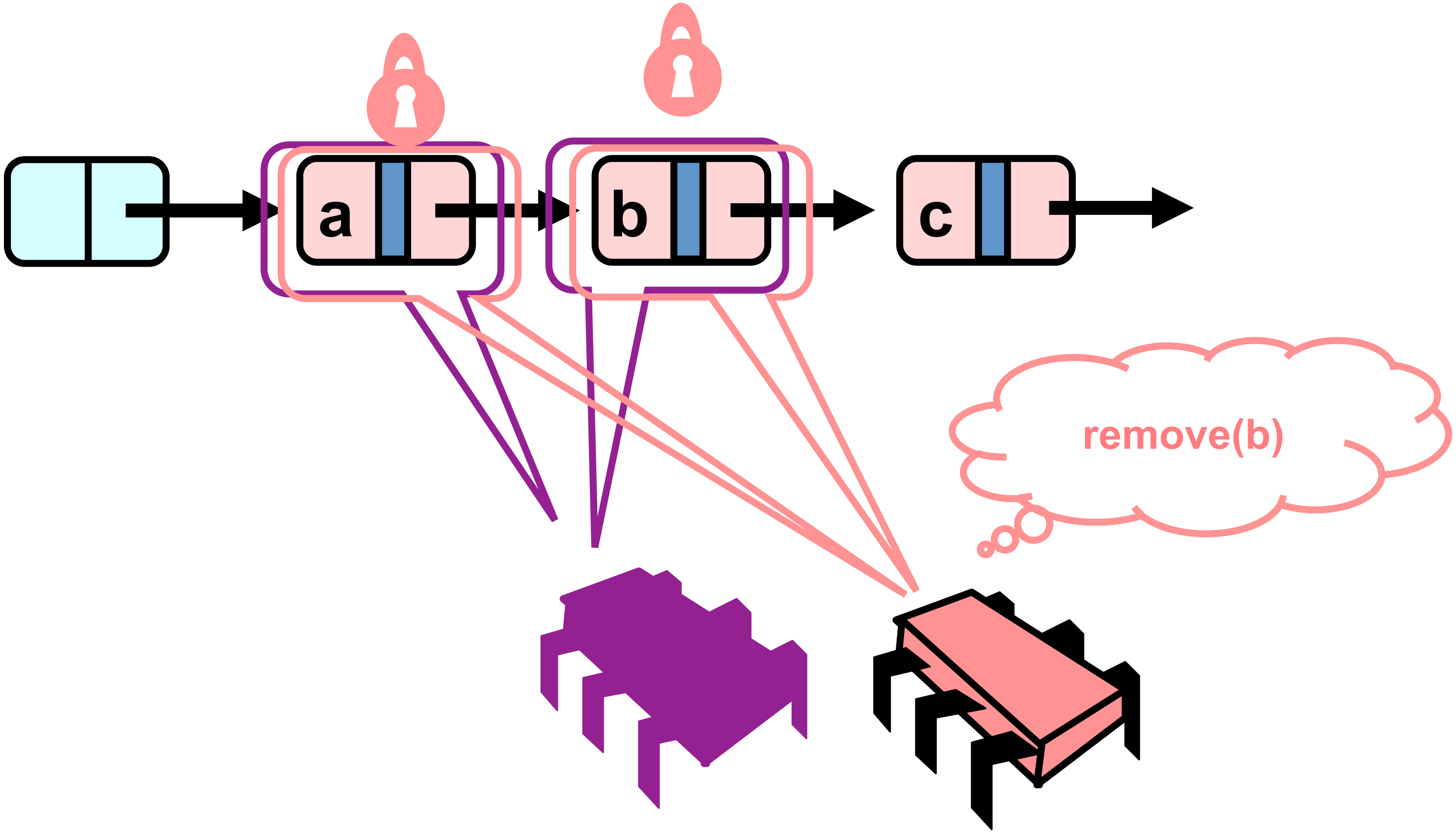
Lazy Removal



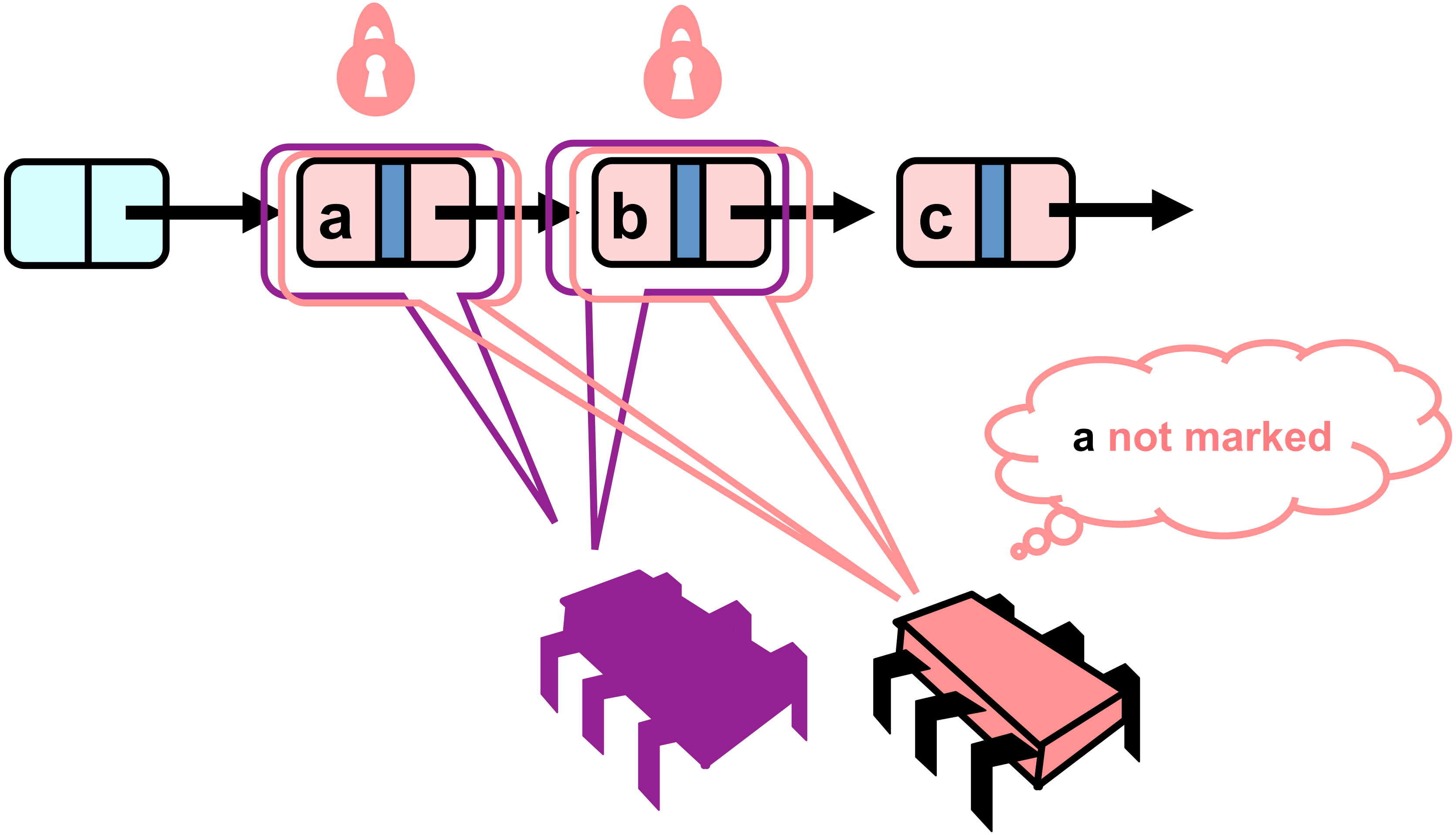
Lazy Removal



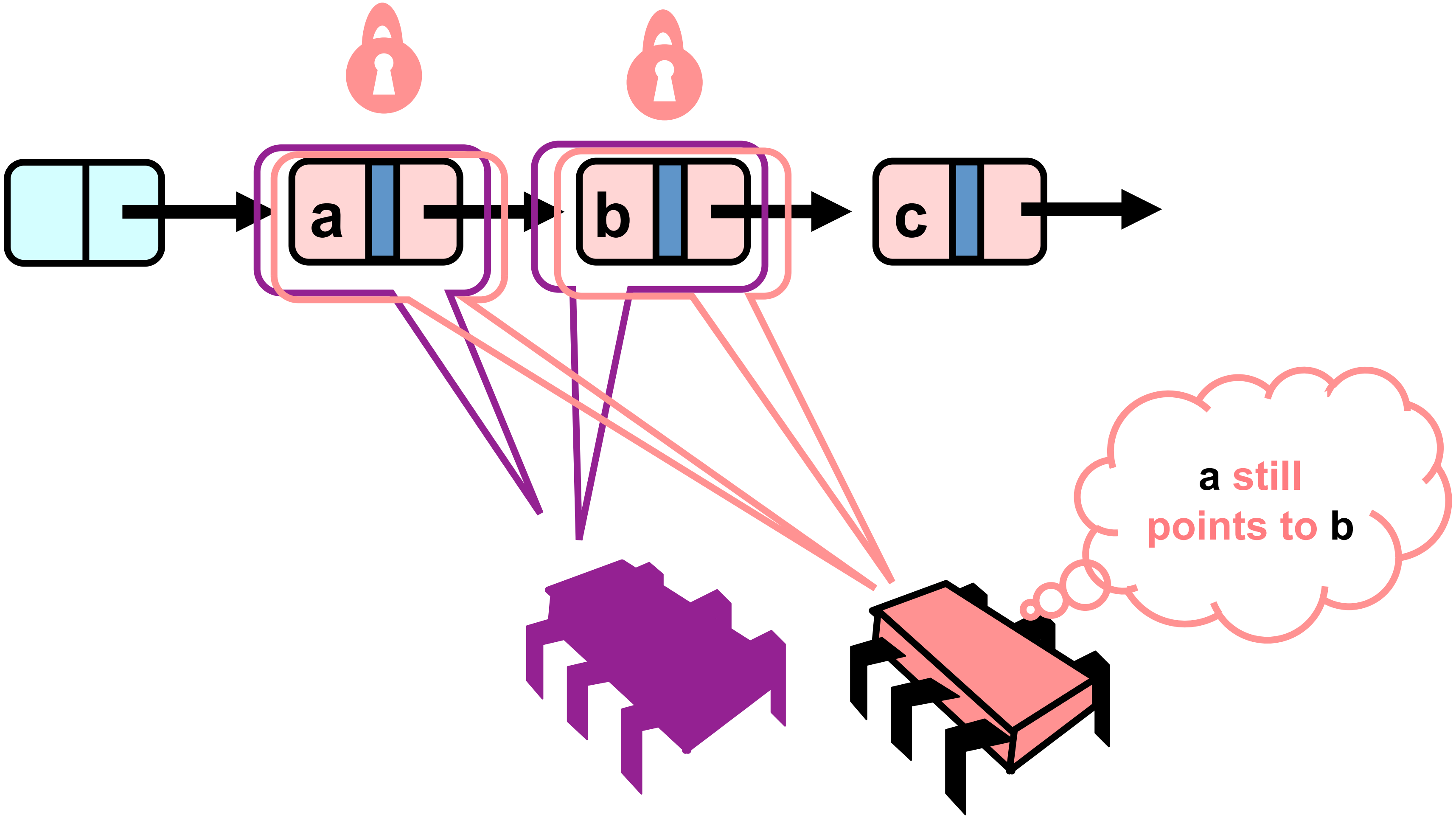
Lazy Removal



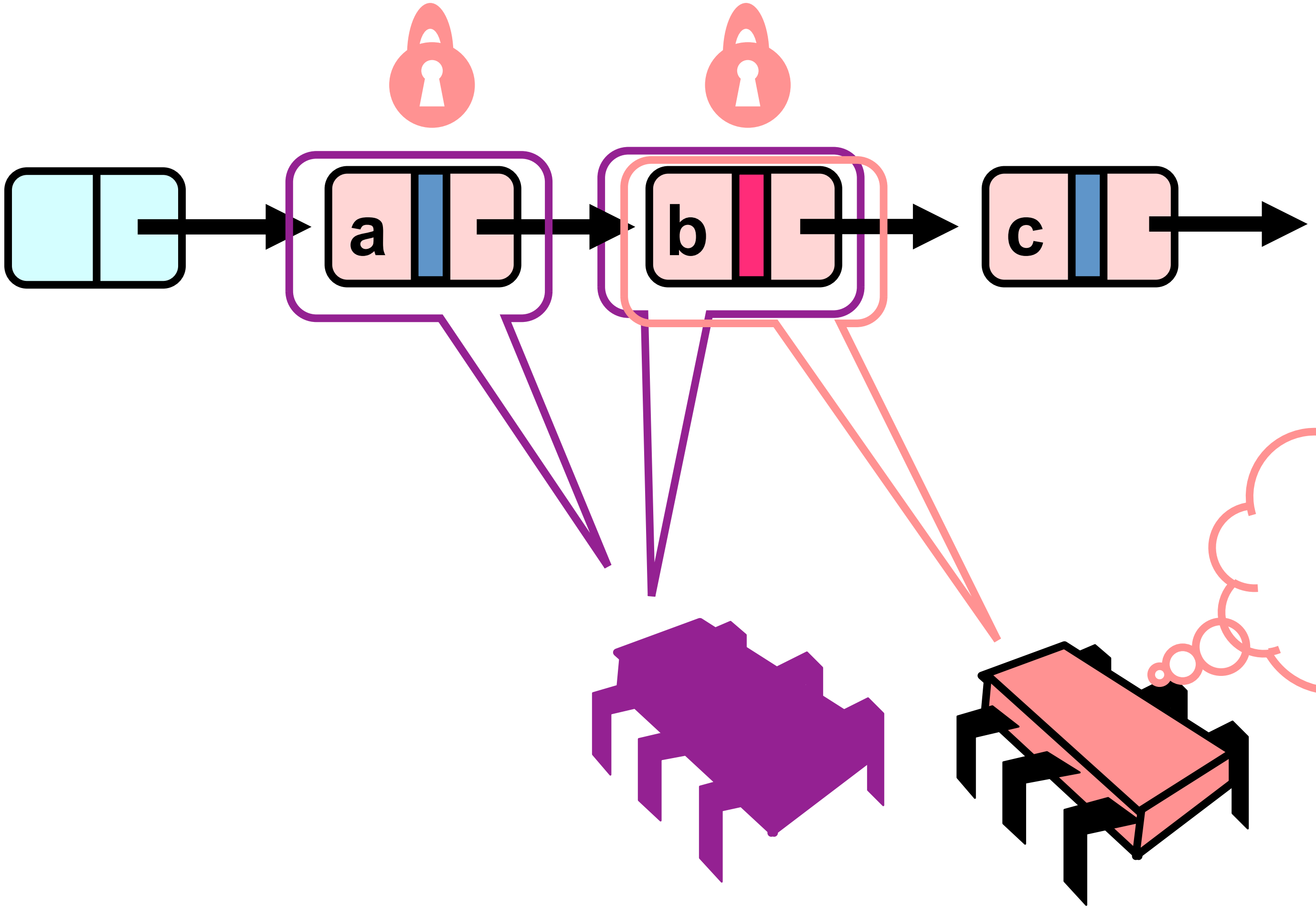
Lazy Removal



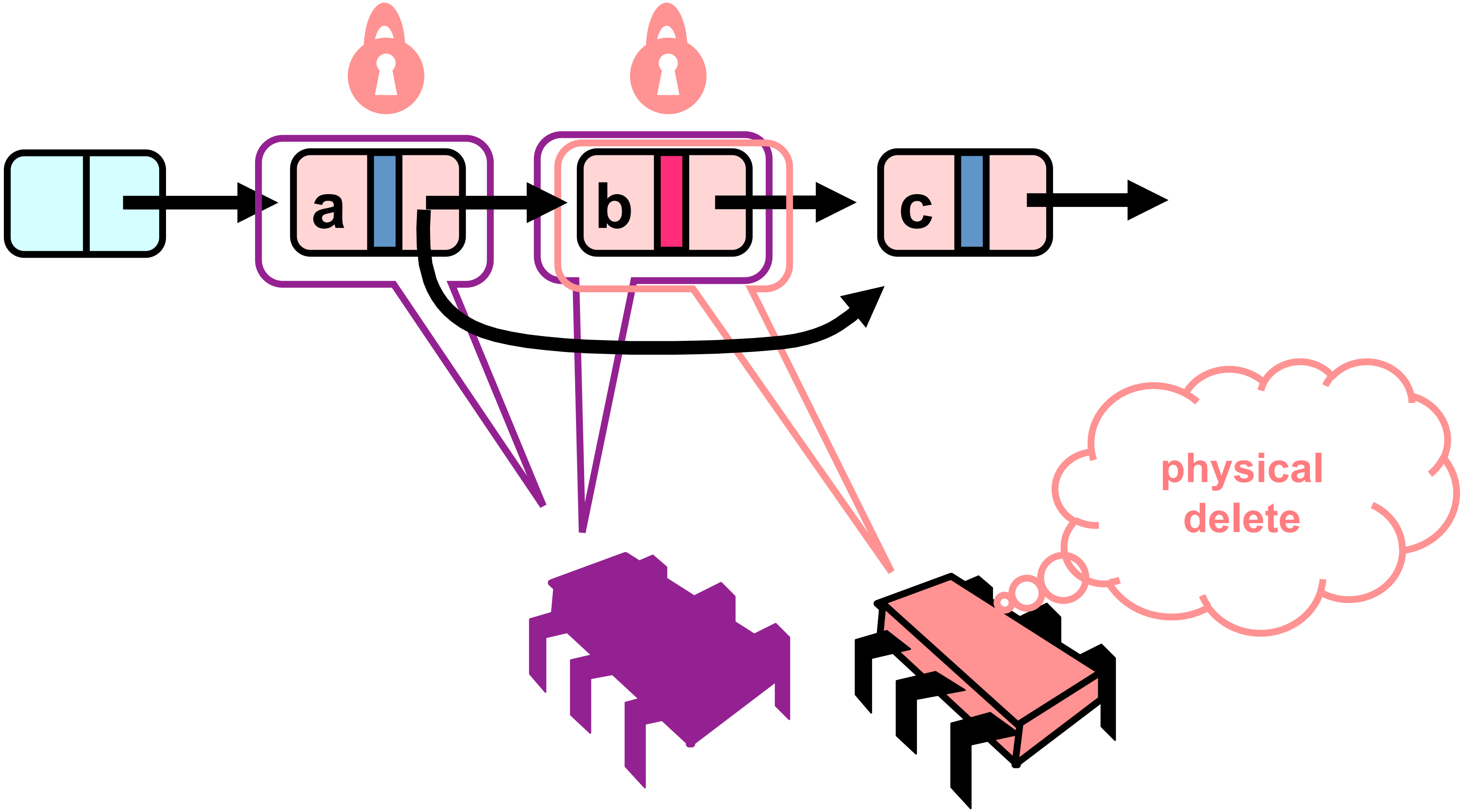
Lazy Removal



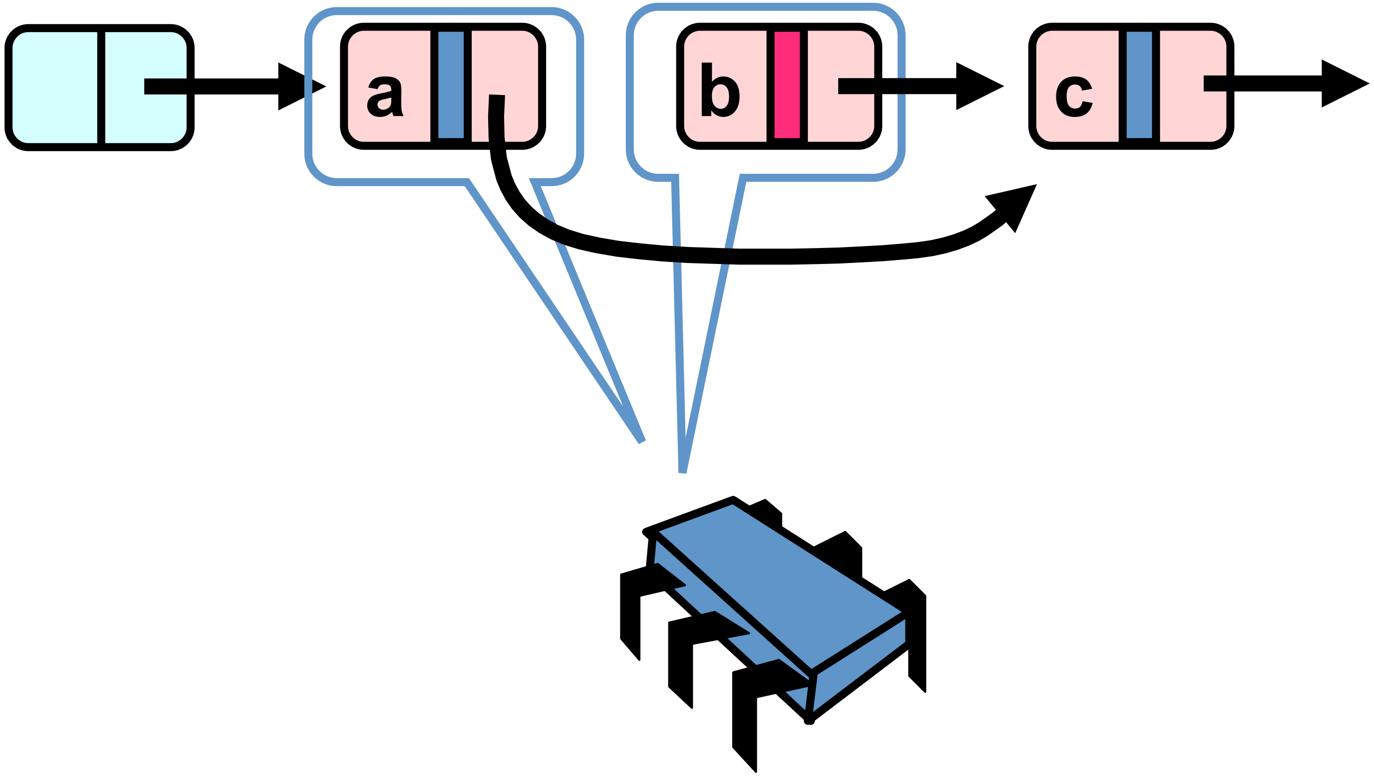
Lazy Removal



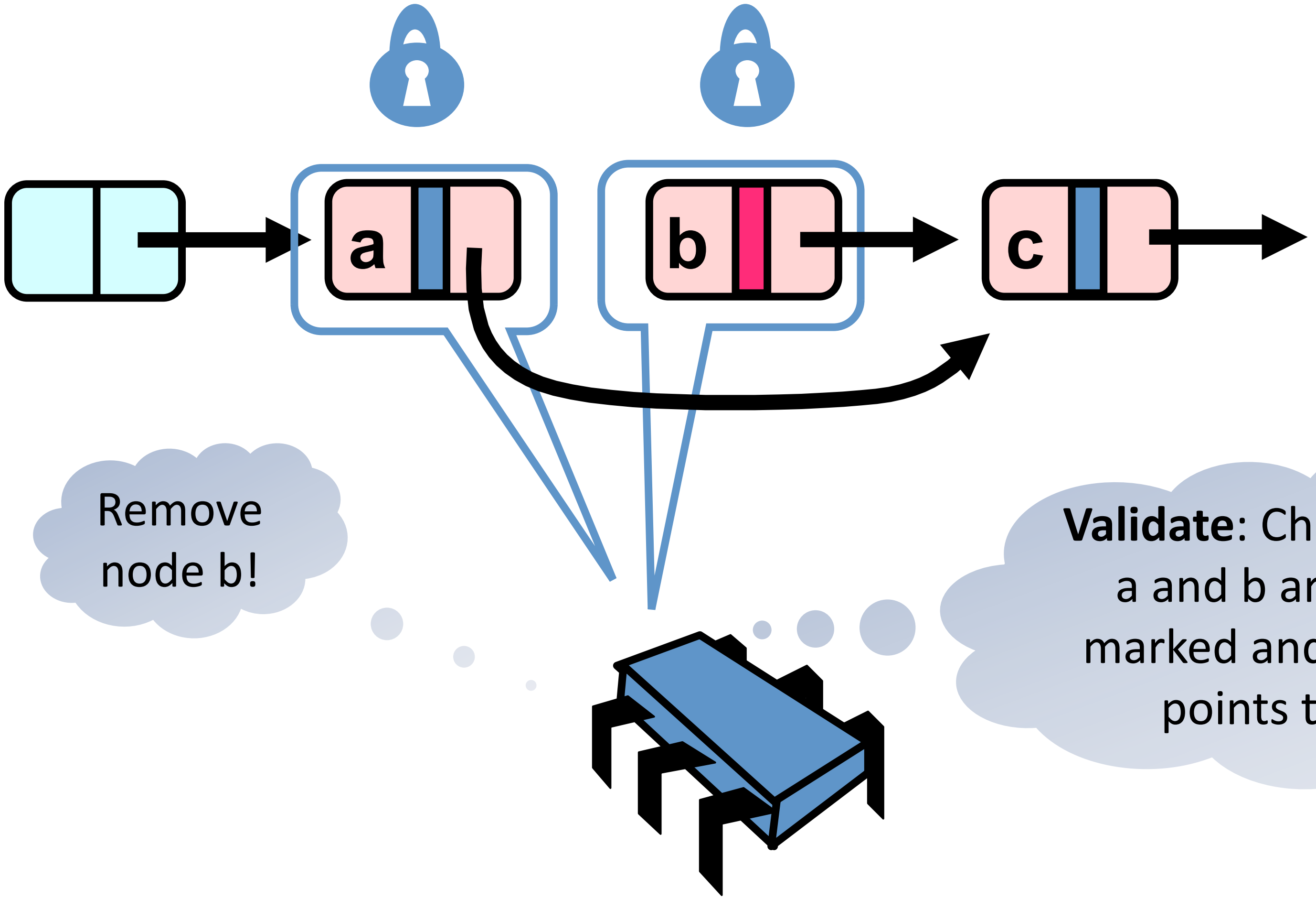
Lazy Removal



Lazy Removal



Lazy Removal



Remove node b!

Validate: Check that a and b are not marked and that a points to b

Lazy Synchronization: Validation

```
private boolean validate(Node pred,Node curr) {
  return !pred.marked && !curr.marked &&
  pred.next == curr);
}
```

Predecessor still points to current

Nodes have not been logically removed

Lazy Synchronization: Remove

```
public boolean remove(Item item) {
  int key = item.hashCode();
  while (true) {
    Node pred = this.head;
    Node curr = pred.next;
    while (curr.key <= key) {
    if (item == curr.item)
     break;
    pred = curr;
    curr = curr.next;
 }
 ...
```

This is the same as before!

Lazy Synchronization: Remove

```
...
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr)) {
    if (curr.item == item) {
      curr.marked = true;
      pred.next = curr.next;
      return true;
    } else {
      return false;
    }}
} finally {
  pred.unlock();
  curr.unlock();
}}}
```

Check for synchronization conflicts

If the target is found, mark the node and remove it

Lazy Synchronization: Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next
  }
  return curr.key == key && !curr.marked;
```

Traverse without locking (nodes may have been removed)

Is the element present and not marked?

Observe: contains() is wait-free !

- Depends on boundedness of keyspace – why?

Evaluation

- Good
  - The list is traversed only once without locking
  - `contains()` is wait-free
  - `contains()` "more common" than `add()` or `remove()`
  - Uncontended calls don't re-traverse
- Bad
  - Contended `add()` and `remove()` calls do re-traverse
  - Traffic jam if one thread delays
- Traffic jam?
  - If one thread gets the lock and experiences a cache miss/ page fault, every other thread that needs the lock is stuck!
  - We need to trust the scheduler….

Reminder: Lock-Free Data Structures

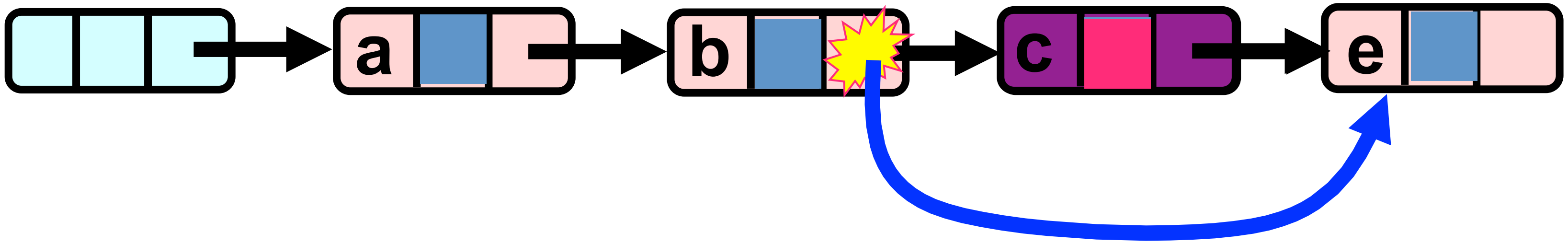
- No matter what …
  - Guarantees minimal progress in any execution
  - i.e. some thread will always complete a method call
  - Even if others halt at malicious times
  - Implies that implementation can't use locks

Lock-Free Lists

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

Lock-Free Lists

Logical Removal

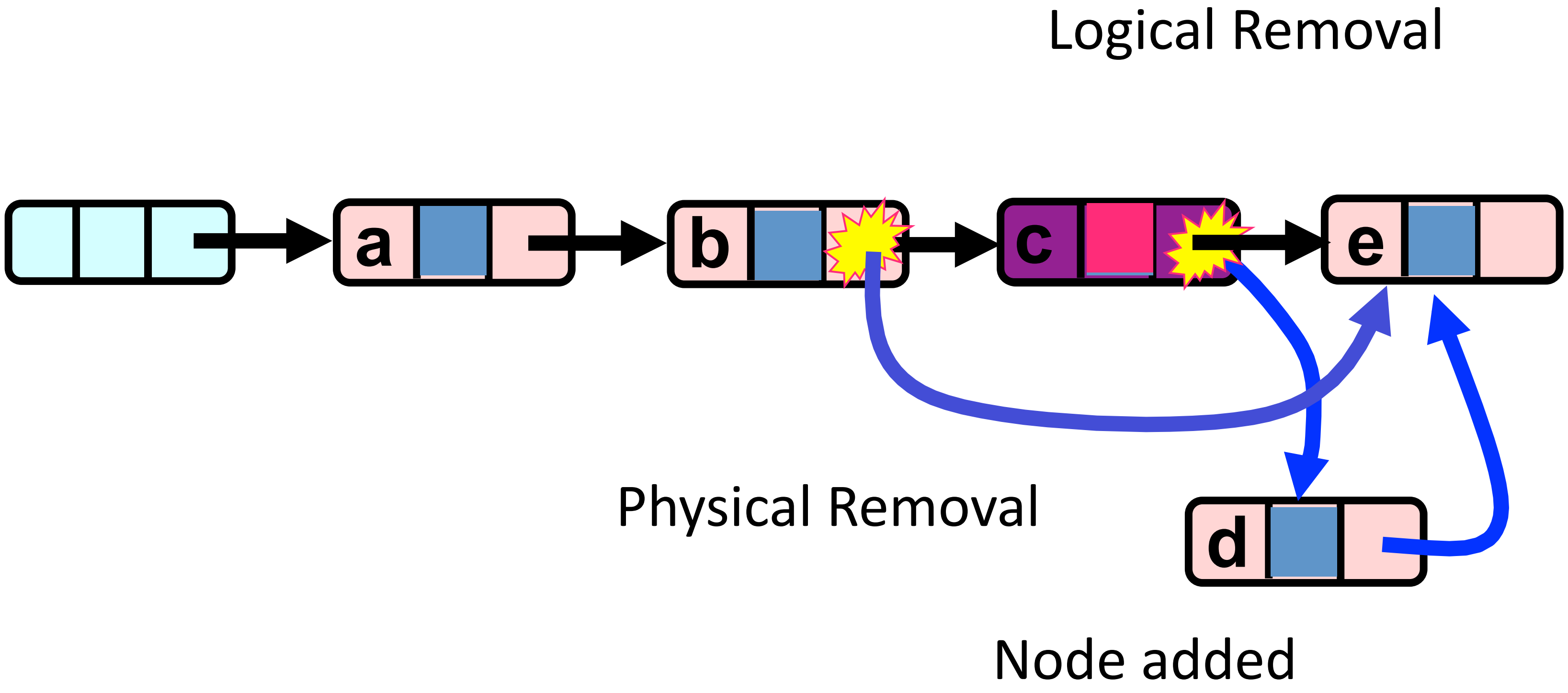


Physical Removal

Use CAS to verify pointer is correct

Not enough!

Problem…

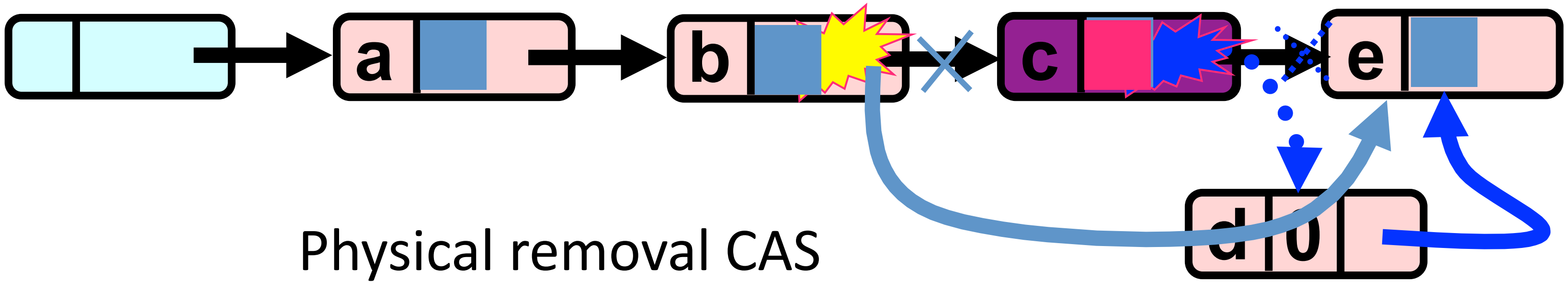


Logical Removal

Physical Removal

Node added

The Solution: Combine Bit and Pointer



Logical Removal = Set Mark Bit

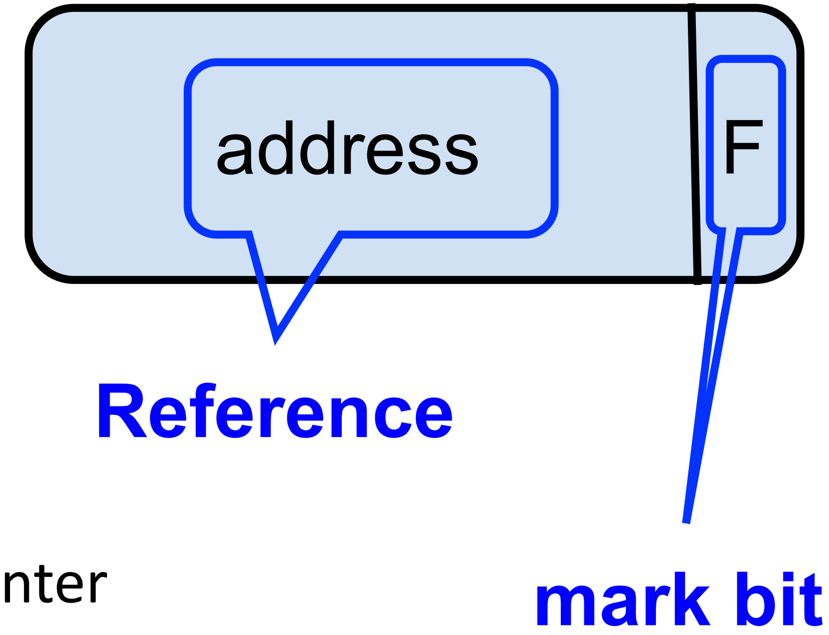
Physical removal CAS

Fail CAS: Node not added after logical removal

Mark-Bit and Pointer are CASed together (AtomicMarkableReference)

Solution

- Use `AtomicMarkableReference`
- Atomically
  - Swing reference and
  - Update flag



- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor’s pointer

- `AtomicMarkableReference` class
  - `java.util.concurrent.atomic` package

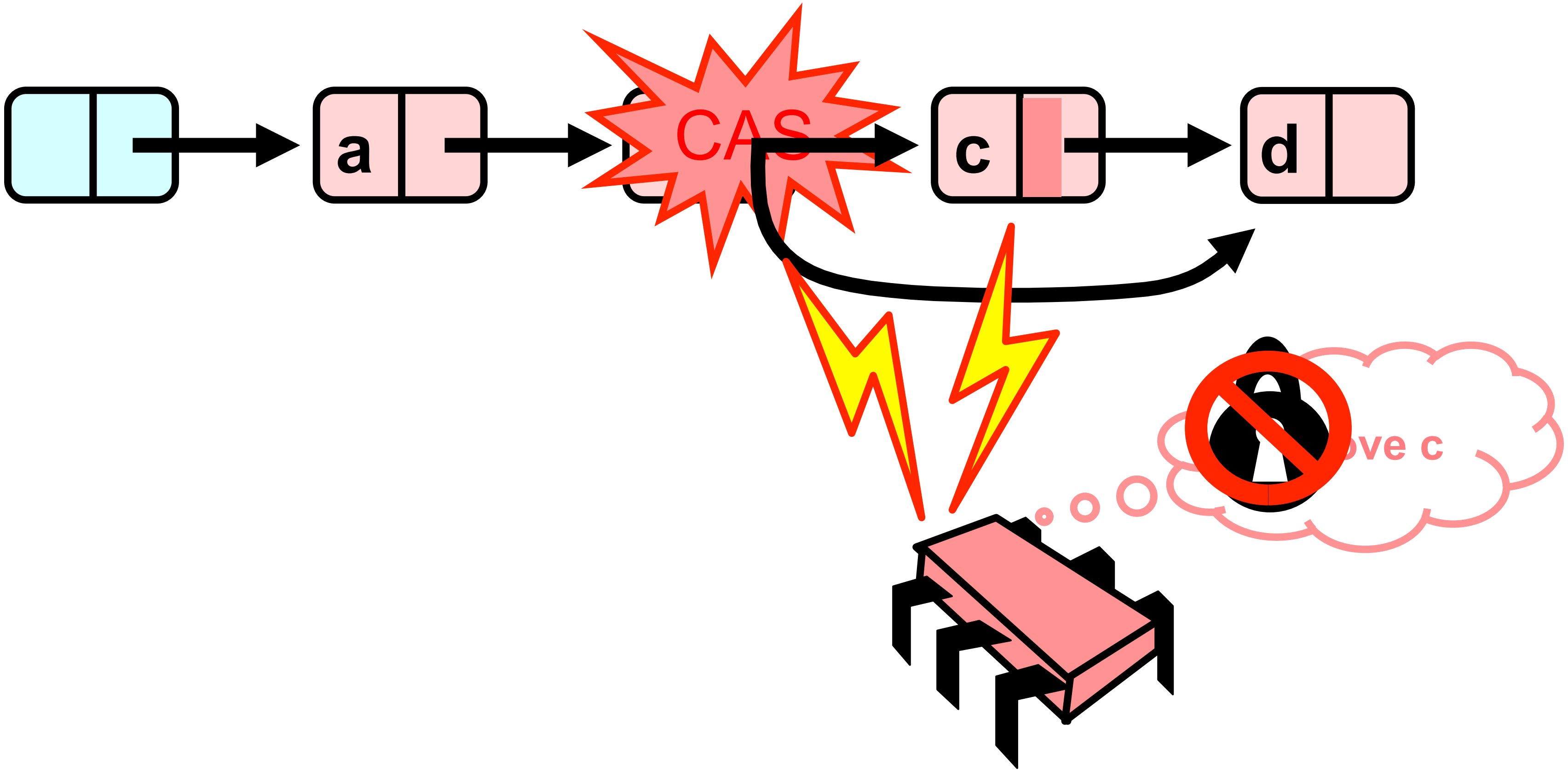
Changing State

```
private Object ref;
private boolean mark;

public synchronized boolean compareAndSet(
Object expectedRef, Object updateRef,
boolean expectedMark, boolean updateMark) {

  if (ref == expectedRef && mark == expectedMark){
    ref = updateRef;
    mark = updateMark;
  }
}
```

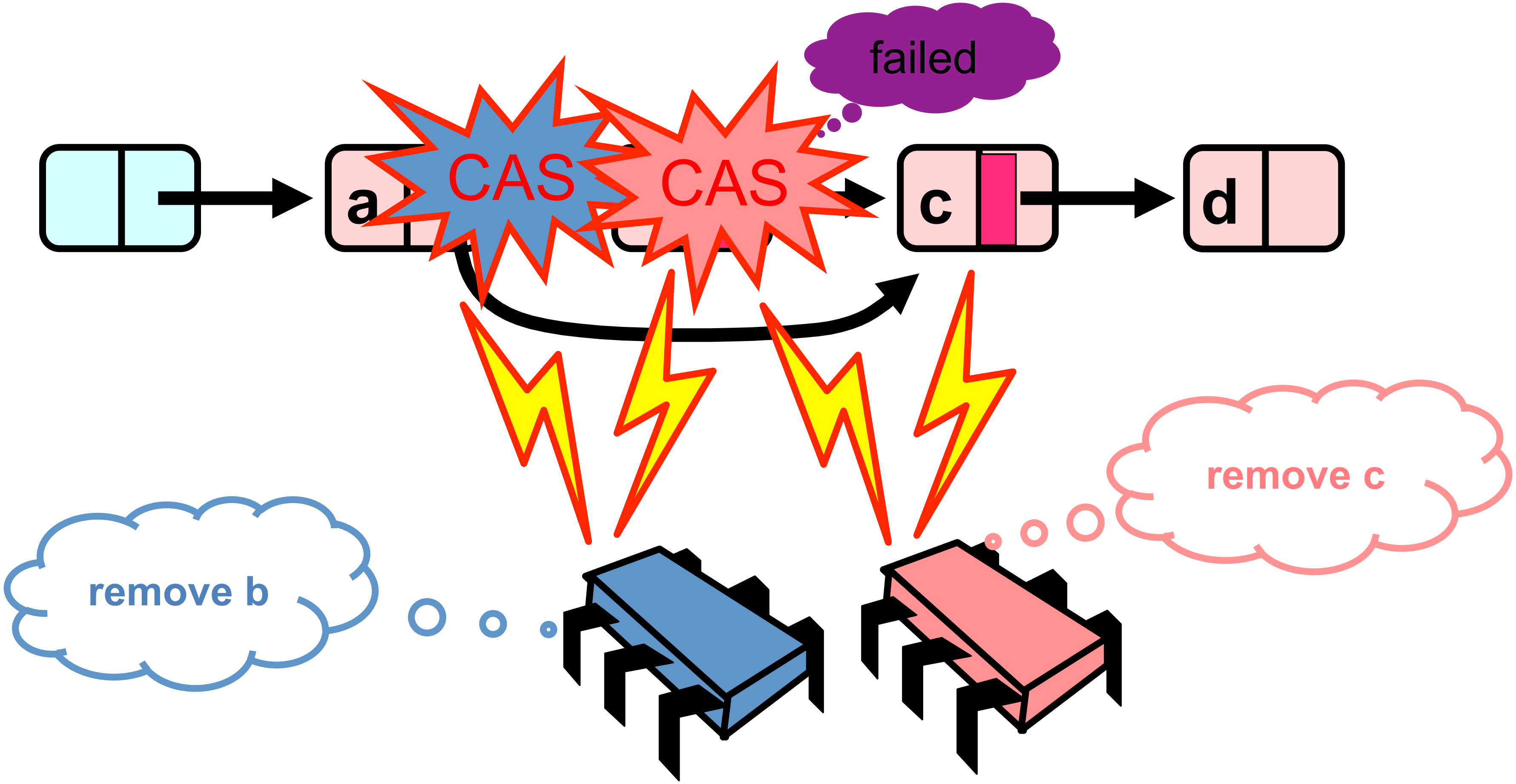
The reference to the next object and the mark bit

If the reference and the mark are as expected, update them atomically

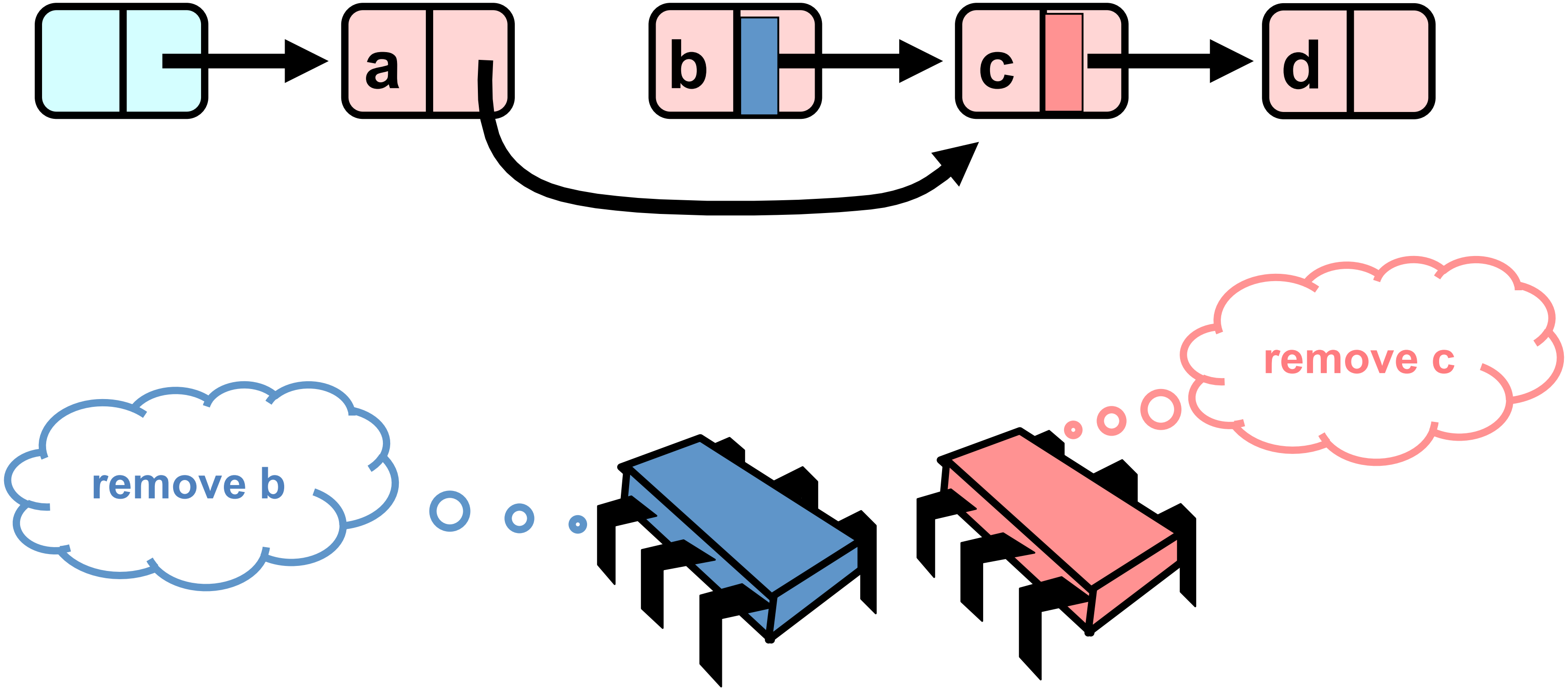
Removing a Node



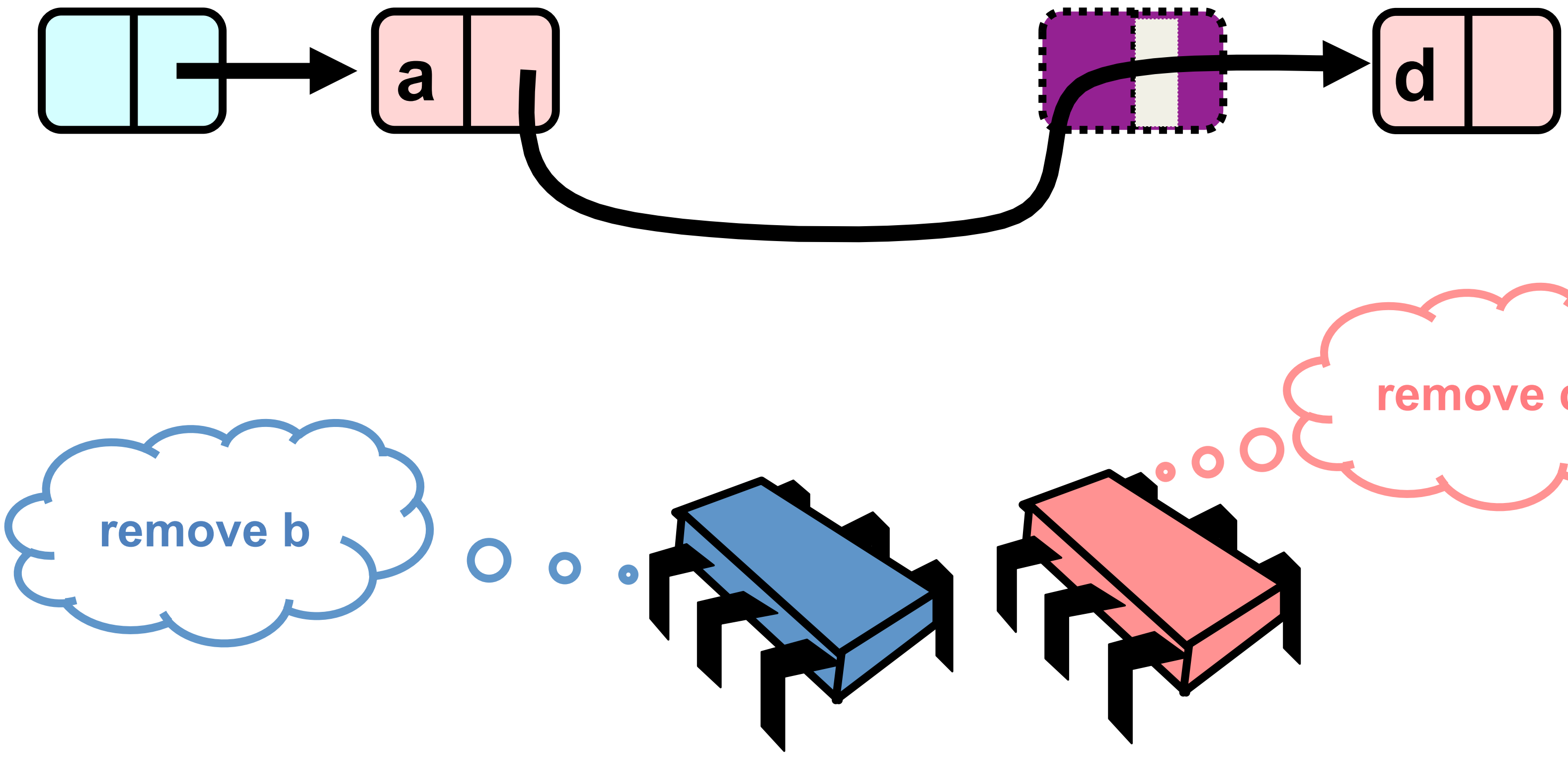
Removing a Node



Removing a Node



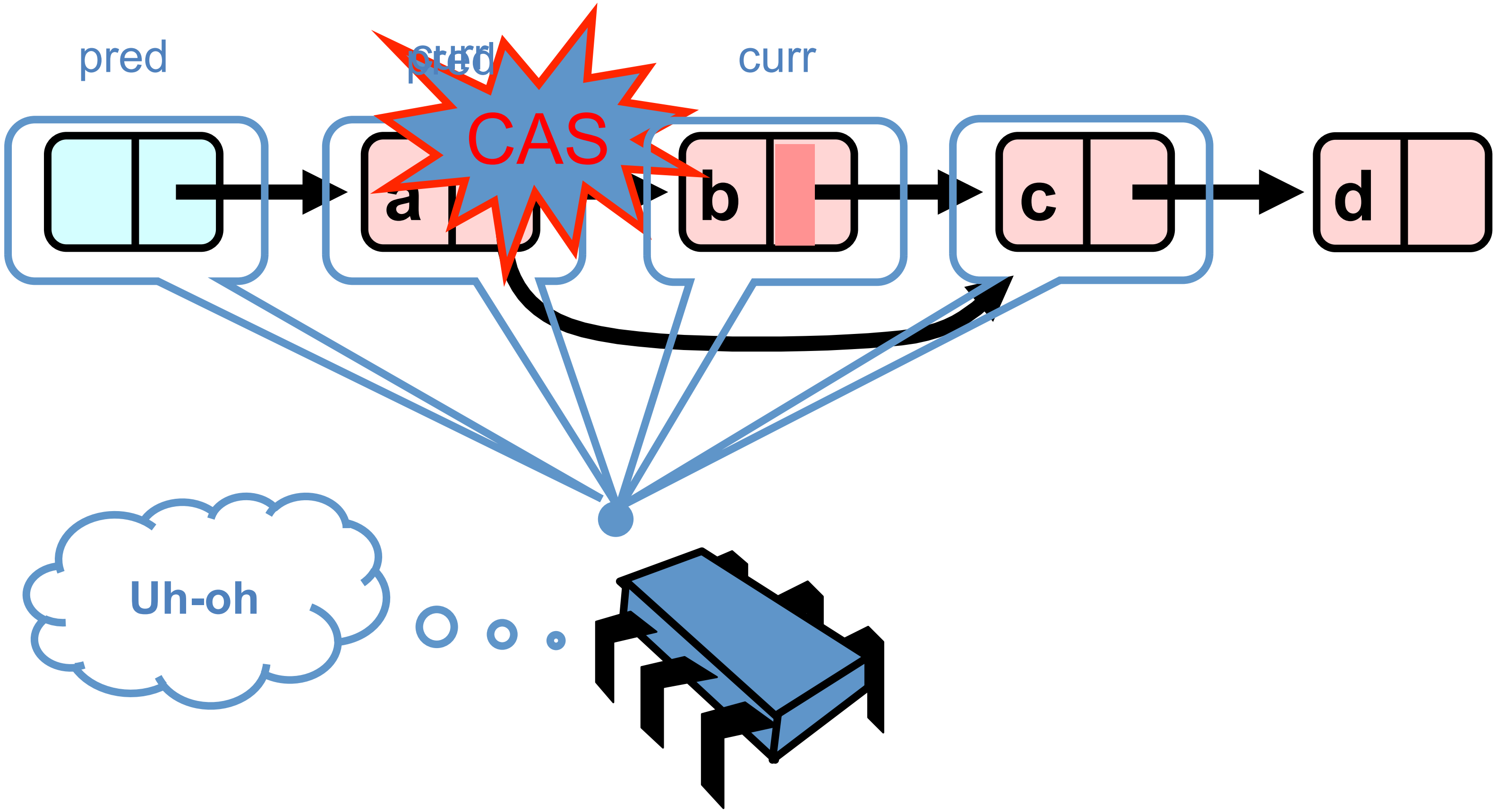
Removing a Node



Traversing the List

- CAS on an `AtomicMarkableReference` marks and swaps
- Marked nodes still hang around
- So: what do you do when you find a marked node in your path?
- Answer: finish the job.
  - CAS the predecessor's next field
  - Proceed (repeat as needed)

Lock-Free Traversal
(only Add and Remove)



The Window Class

- Ancillary class to help with traversal
- Produced by **`find(item)`**
- **`find()`** also removes marked nodes on the fly

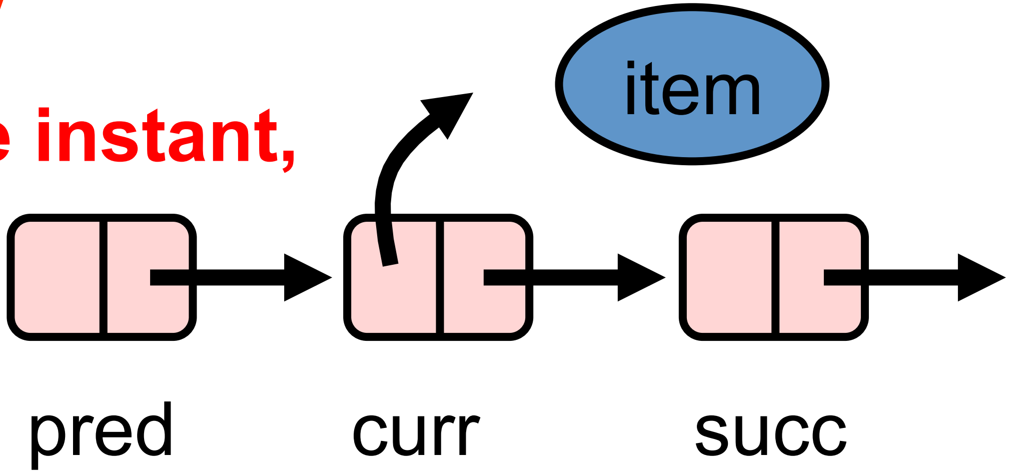
```
class Window {
 public Node pred;
 public Node curr;
 Window(Node pred, Node curr) {
   this.pred = pred; this.curr = curr;
 }
}
```

A container for pred and current values

Using the Find Method

```
Window window = find(item);
```

At some instant,

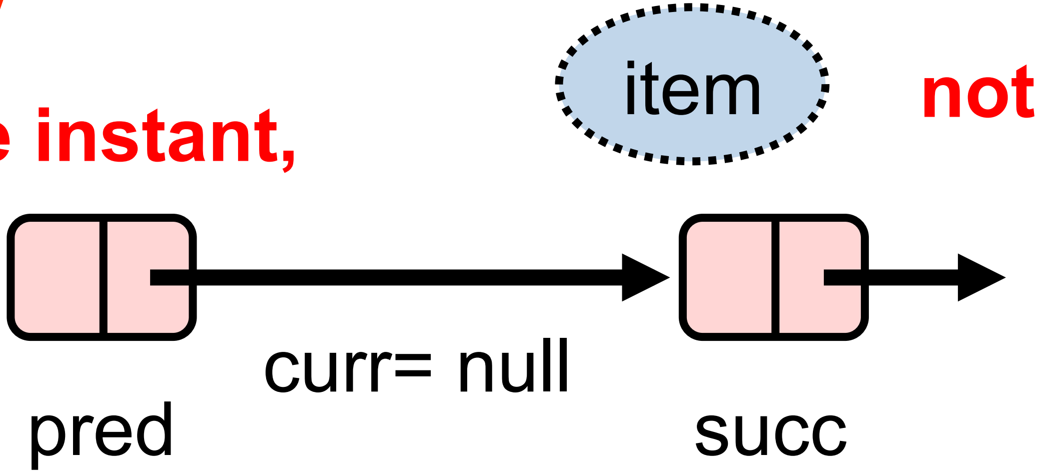


or …

The Find Method

```
Window window = find(item);
```

At some instant,



not in list

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

Remove

Find neighbors

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

She’s not there …

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

Try to mark node as deleted

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

Didn’t work? Retry

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false,
false);
     return true;
}}}
```

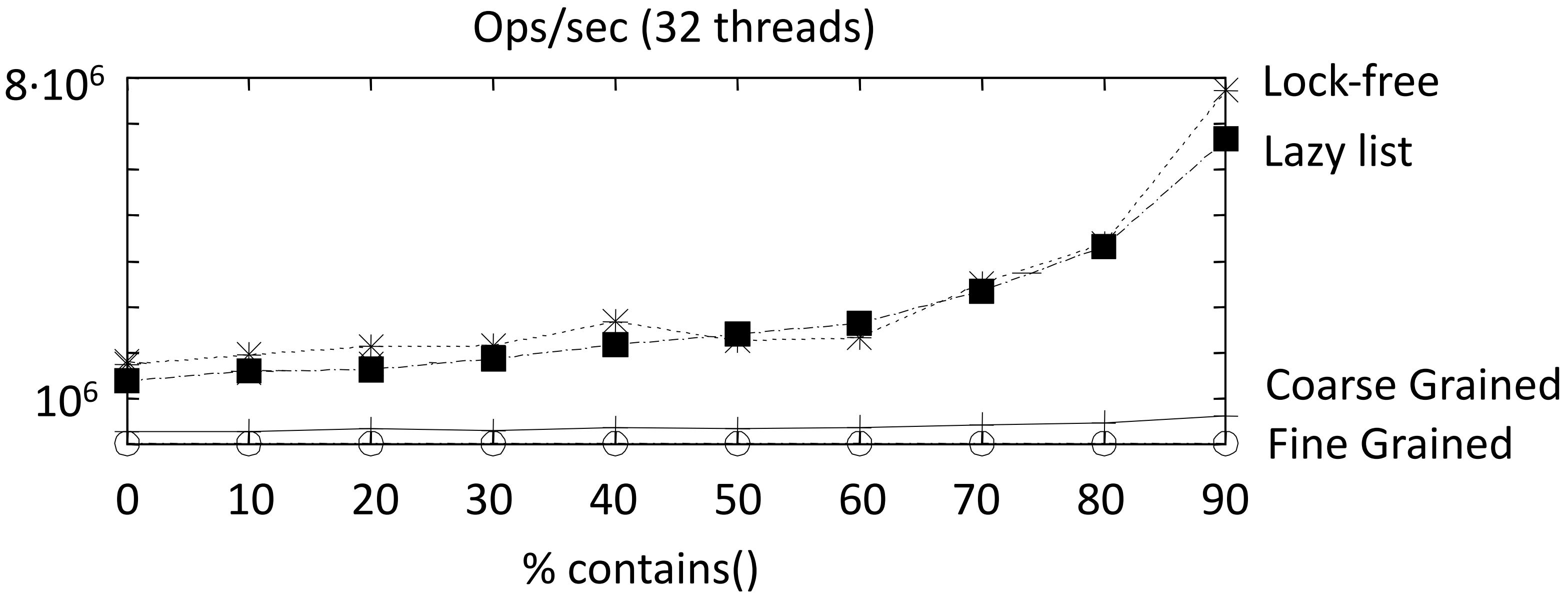
Try to complete the removal – if not successful no matter, someone else will be

Other Methods

- Check out the H&S book for:
- `add(item)`
- Wait-free `contains(item)`
  - Much as the lazy list case
- Lock-free `find(item)`

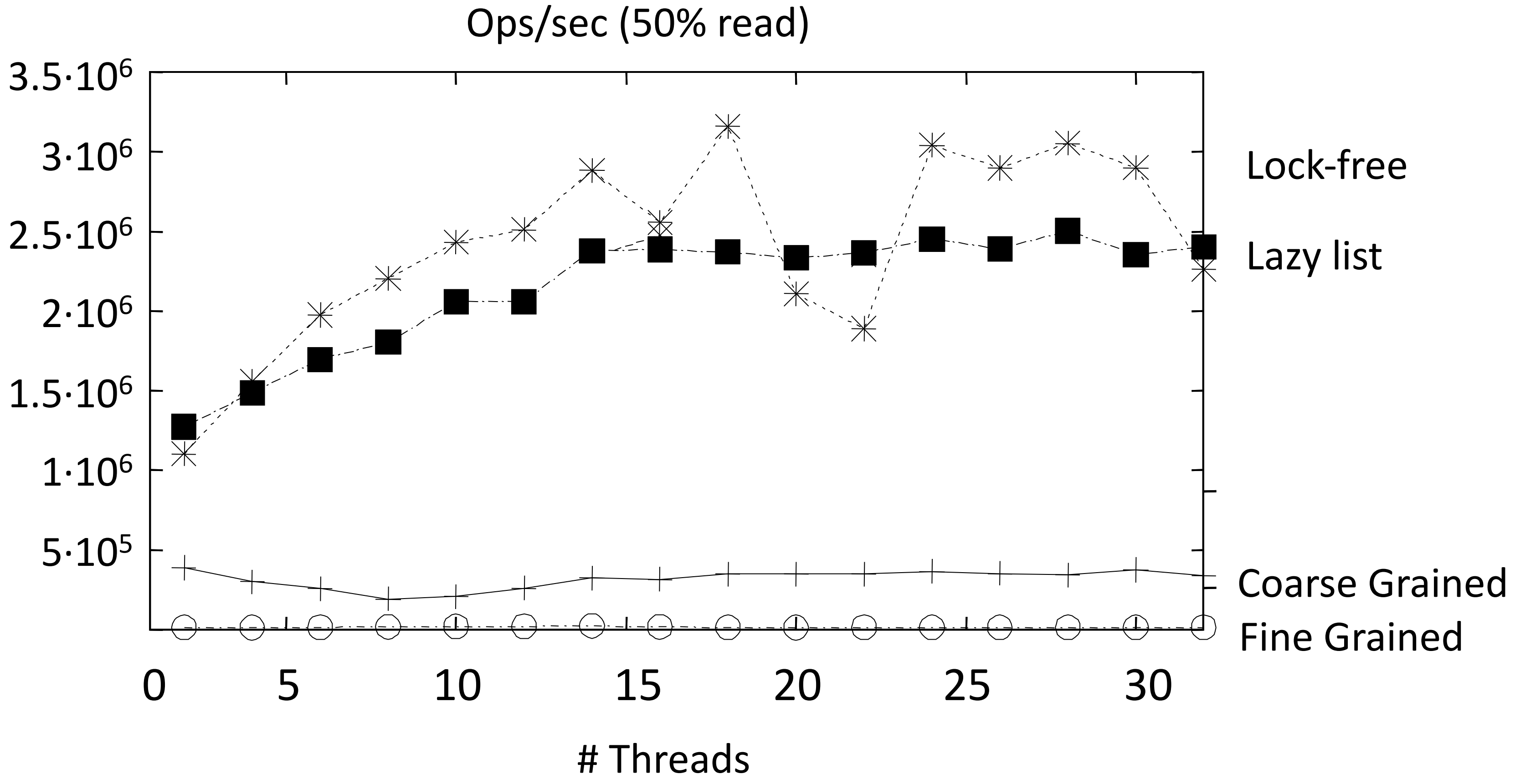
Performance

- The throughput of the presented techniques has been measured for a varying percentage of `contains()` calls
- Benchmarked on a 16 node shared memory machine



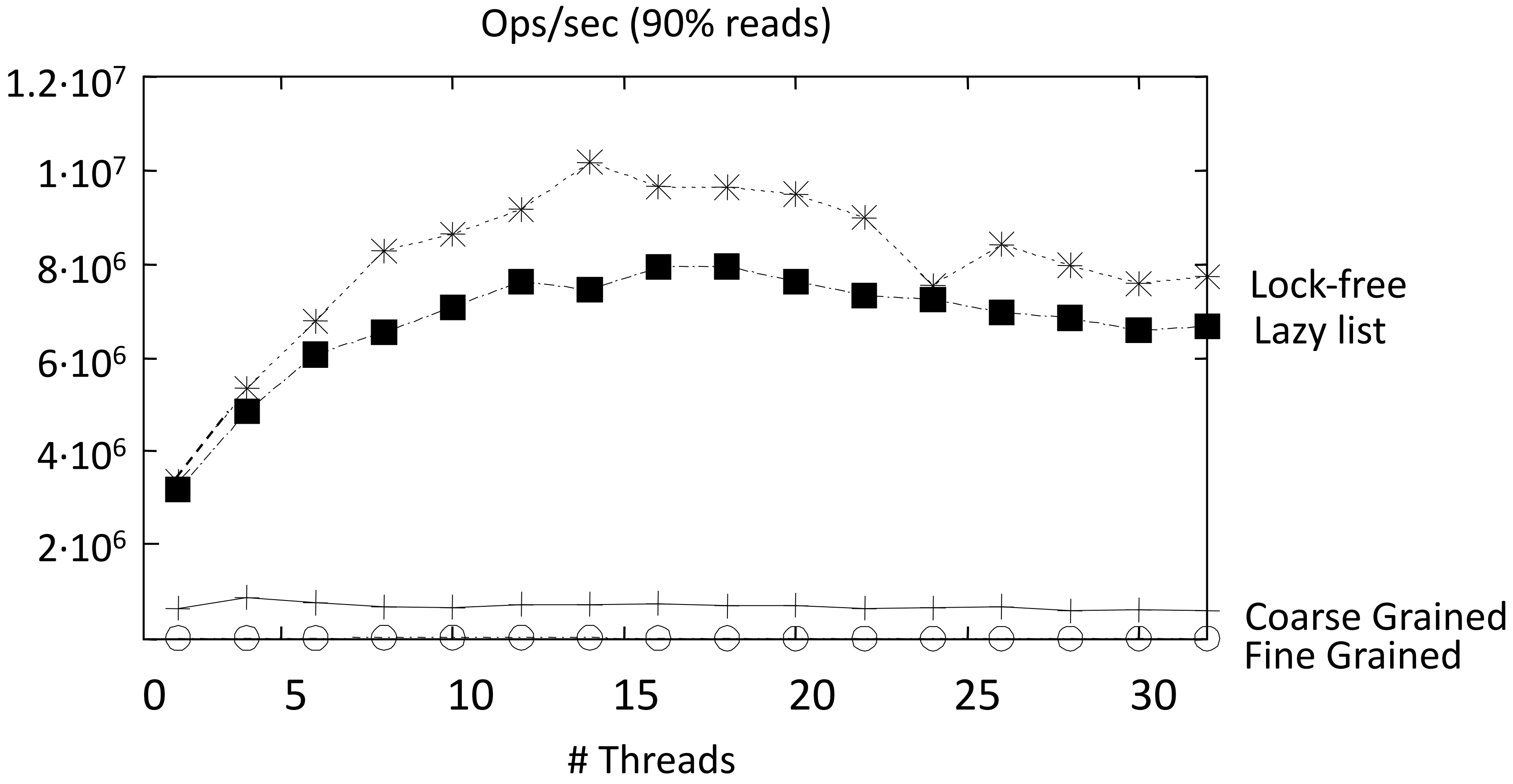
Low Ratio of `contains()`

- The lock-free linked list and the linked list with lazy synchronization perform well even if there are many threads



High Ratio of `contains()`

- Similar picture



Summary

- Concurrent linked list implementations of increasing complexity
- Optimistic – lazy – lock-free: Recurring themes
- Lock-free:
  - Still not ideal
  - Needs atomic updates of reference/mark pairs
  - Traversal more complex
- Next in line:
  - More complex data structures
  - Scheduling, work stealing, barrier synchronization
  - Software transactional memory
- Instead change course to message passing concurrency