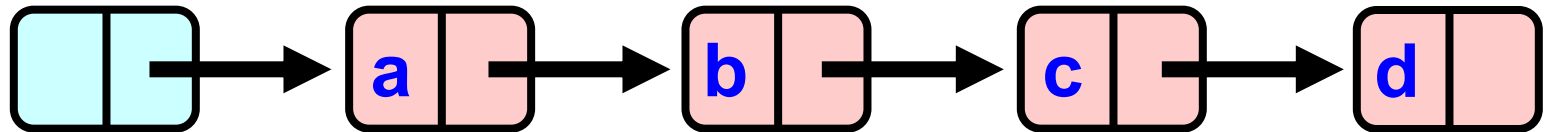
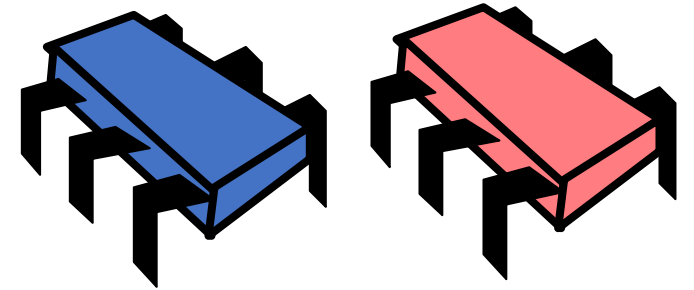


# CSE113: Parallel Programming

- **Topics:**

- Concurrent general set



# Announcements

- HW 4 submission by tomorrow.

# Announcements

- HW 5 will be released on this week.

# Announcements

SETs are out, please do them! It helps us out a lot.

# Previous quiz + Review

# Previous quiz + Review

The C++ relaxed memory order provides

- ☐ no orderings at all
- ☐ orderings only between accesses of the same address
- ☐ TSO memory behaviors when run on an x86 system
- ☐ an easy way to accidentally introduce horrible bugs into your program

# Relaxed memory order

language  
C++11 (sequential consistency)

	L	S
L	NO	NO
S	NO	NO

language  
C++11 (memory\_order\_relaxed)

	L	S
L	different address	different address
S	different address	different address

basically no orderings except for accesses to  
the same address

# Previous quiz + Review

The C++ relaxed memory order provides

- ☐ no orderings at all
- > ☐ orderings only between accesses of the same address
- ☐ TSO memory behaviors when run on an x86 system
- ☐ an easy way to accidentally introduce horrible bugs into your program



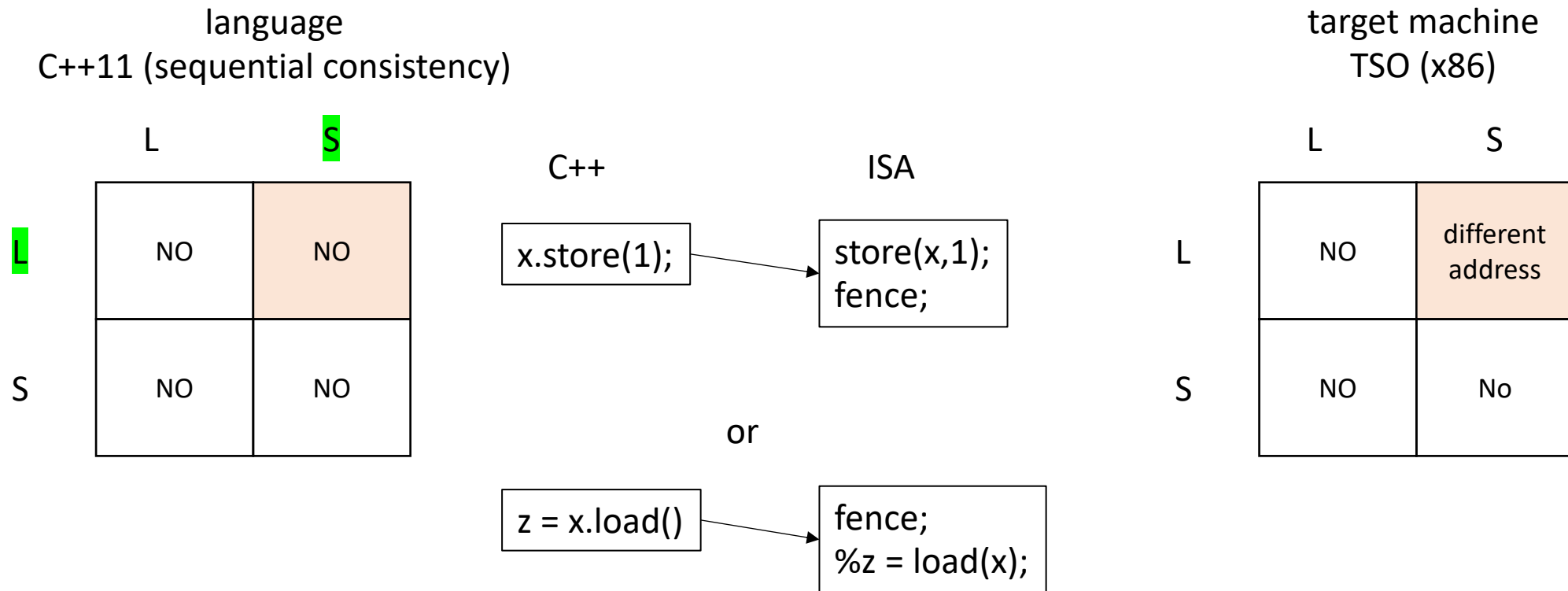
# Previous quiz + Review

In terms of memory models, the compiler needs to ensure the following property:

- ☐ Any weak behavior allowed in the language is also allowed in the ISA
- ☐ Any weak behaviors that are disallowed in the language need to be disallowed in the ISA
- ☐ The compilation ensures that the program has sequentially consistent behavior at the ISA level
- ☐ The compiler does not need to reason about relaxed memory

# C++11 atomic operation compilation

start with both both of the grids for the two different memory models



# Compiling memory order relaxed

language  
C++11 (memory\_order\_relaxed)

	L	S
L	different address	different address
S	different address	different address

lots of mismatches!

But language is more relaxed than machine

*so no fences are needed*

target machine  
TSO (x86)

	L	S
L	NO	different address
S	NO	No

# Previous quiz + Review

In terms of memory models, the compiler needs to ensure the following property:

- ☐ Any weak behavior allowed in the language is also allowed in the ISA
- > ☐ Any weak behaviors that are disallowed in the language need to be disallowed in the ISA
- ☐ The compilation ensures that the program has sequentially consistent behavior at the ISA level
- ☐ The compiler does not need to reason about relaxed memory

# Previous quiz + Review

A program that uses mutexes and has no data conflicts does not have weak memory behaviors for which of the following reasons?

- 
- ☐ Mutexes prevent memory accesses from happening close enough in time for weak behaviors to occur
- 
- ☐ The OS has built in support for Mutexes that disable architecture features, such as the store buffer
- 
- ☐ A correct mutex implementation uses fences in lock and unlock to disallow weak behaviors

# Previous quiz + Review

A program that uses mutexes and has no data conflicts does not have weak memory behaviors for which of the following reasons?

- ☐ Mutexes prevent memory accesses from happening close enough in time for weak behaviors to occur
- ☐ The OS has built in support for Mutexes that disable architecture features, such as the store buffer
- > ☐ A correct mutex implementation uses fences in lock and unlock to disallow weak behaviors

# Previous quiz + Review

Assuming you had a sequentially consistent processor, any C/++ program you ran on it would also be sequentially consistent, regardless of if there are data-conflicts or not.

---

☐ True

---

☐ False

# Previous quiz + Review

Assuming you had a sequentially consistent processor, any C/++ program you ran on it would also be sequentially consistent, regardless of if there are data-conflicts or not.

---

☐ True

-> ☐ False

In addition to processor, the compiler can reorder instruction as well.



# Previous quiz + Review

If you put a fence after every memory instruction, would that be sufficient to disallow all weak behaviors on a weak architecture? Please write a few sentences explaining your answer.

# Previous quiz + Review

If you put a fence after every memory instruction, would that be sufficient to disallow all weak behaviors on a weak architecture? Please write a few sentences explaining your answer.

It is sufficient but makes execution slower.

# General Concurrent Set

# Set Interface

- Unordered collection of items
- No duplicates
- We will implement set as a sorted linked list.

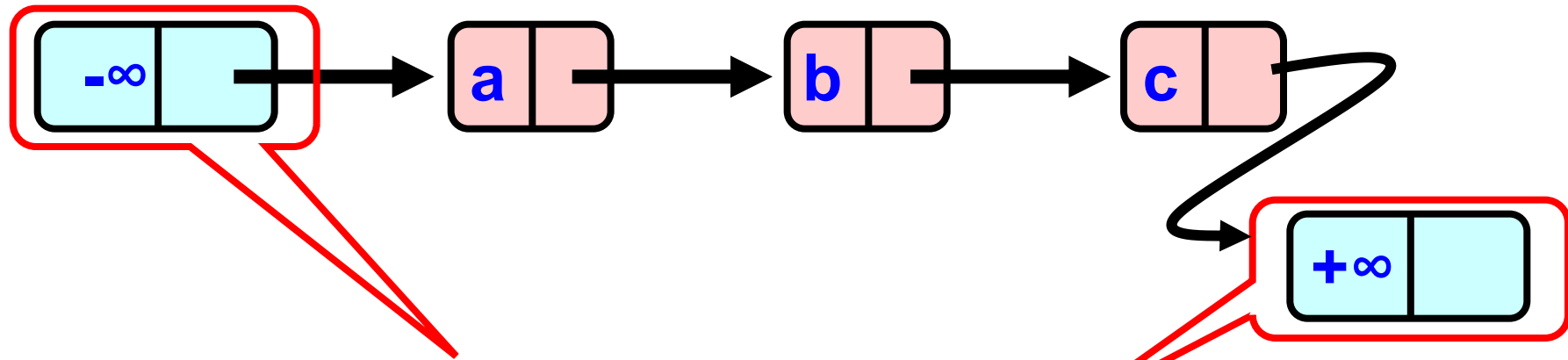
# Set Interface

- Methods
  - **add(x)** put **x** in set
  - **remove(x)** take **x** out of set
  - **contains(x)** tests if **x** in set

# List Node

```
class Node {  
    public:  
        Value v;  
        int key;  
        Node *next;  
}
```

# The List-Based Set



Sorted with Sentinel nodes  
(min & max possible keys)

# Sequential List Based Set

**add(b)**



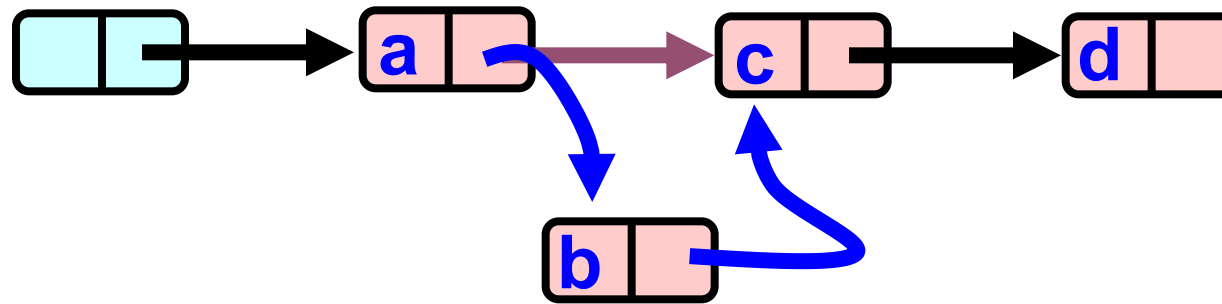
**remove(b)**



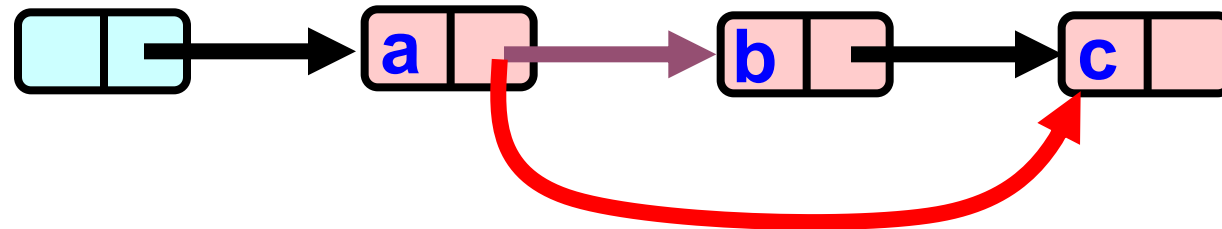


# Sequential List Based Set

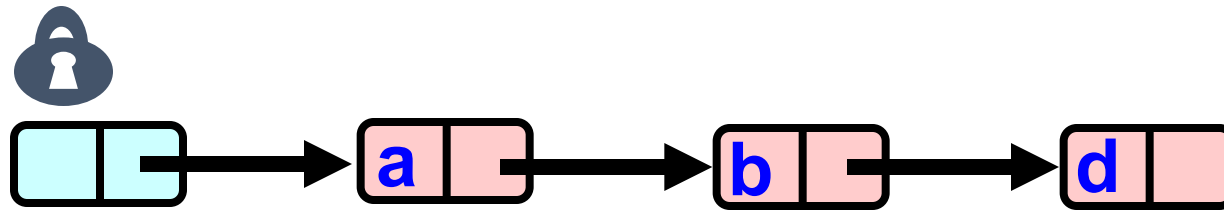
**add(b)**



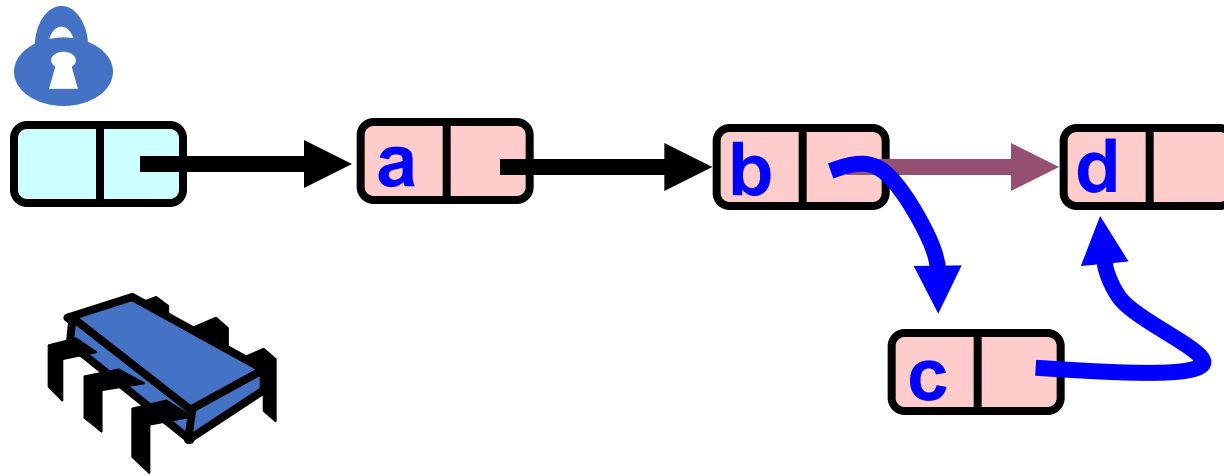
**remove(b)**



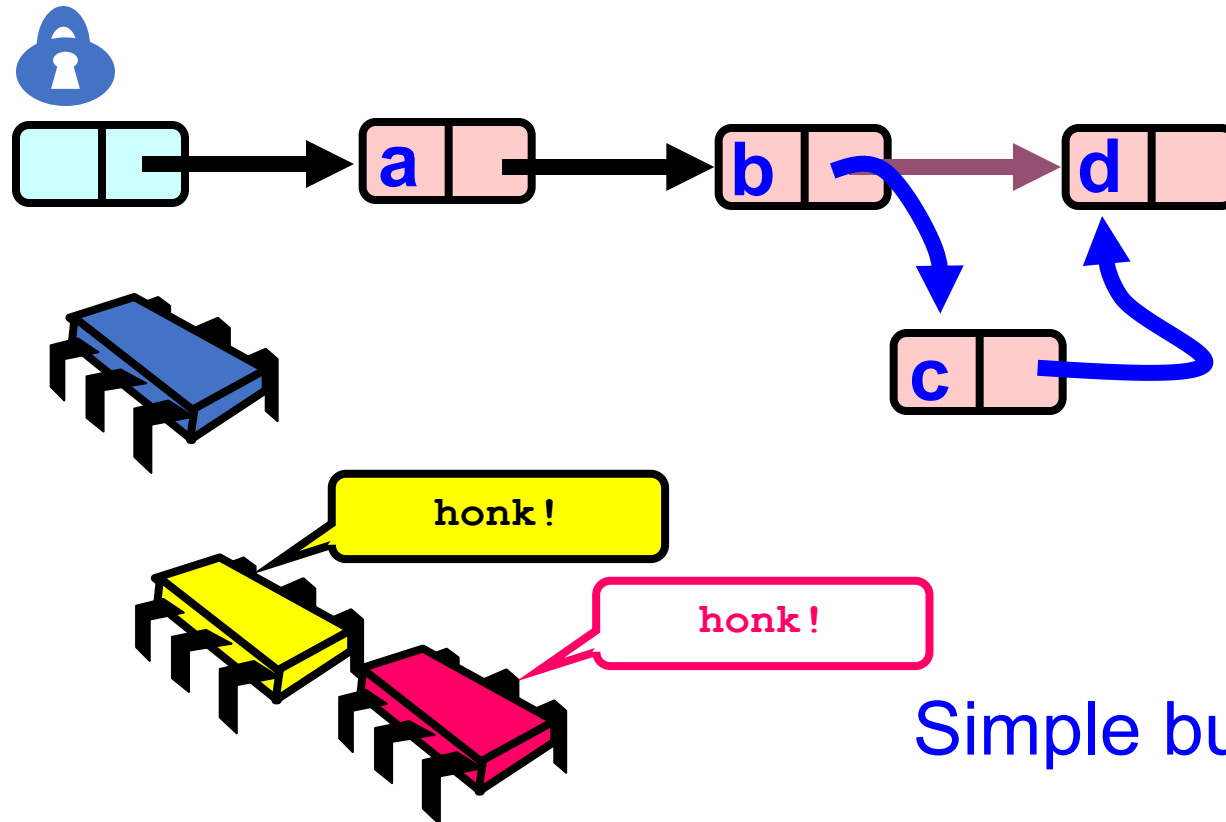
# Coarse-Grained Locking



# Coarse-Grained Locking



# Coarse-Grained Locking



Simple but inefficient!

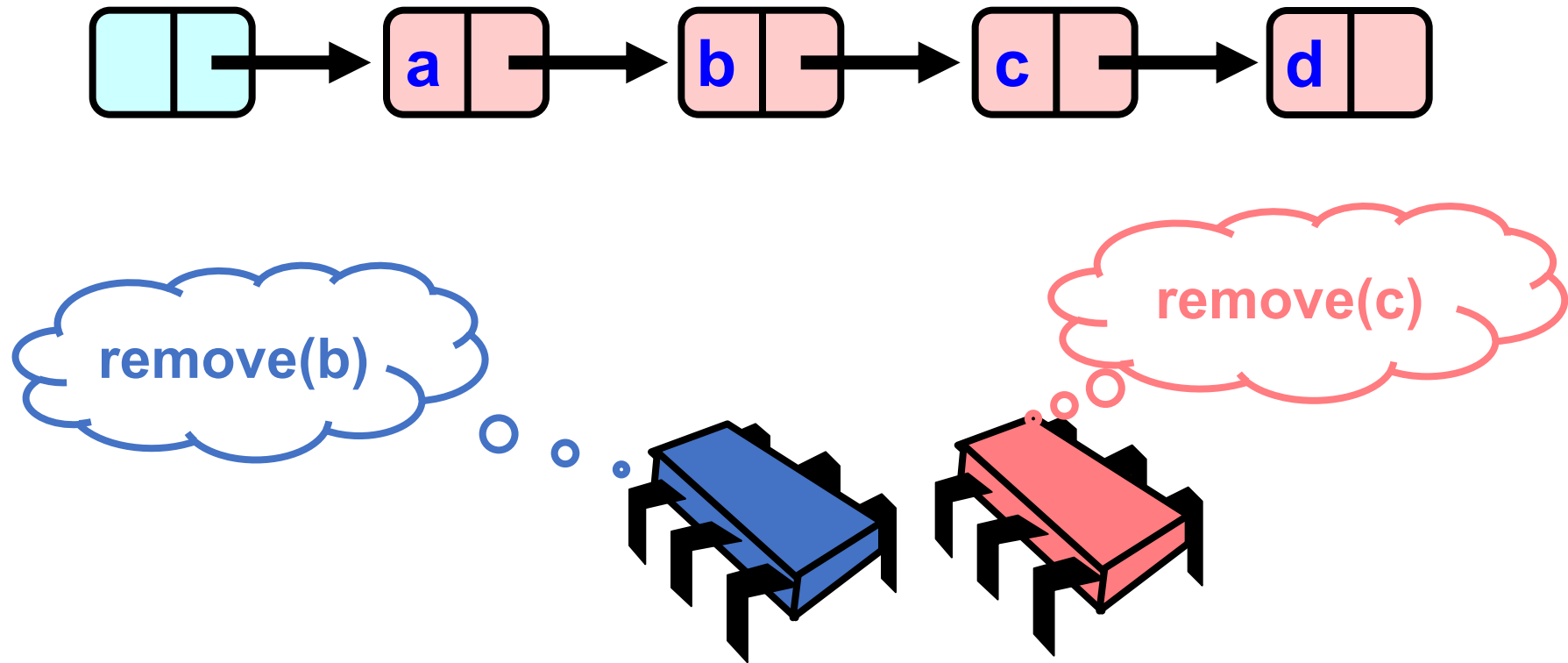
# Schedule

- Concurrent set
  - Coarse-grained lock
  - **fine-grained lock**
  - optimistic locking

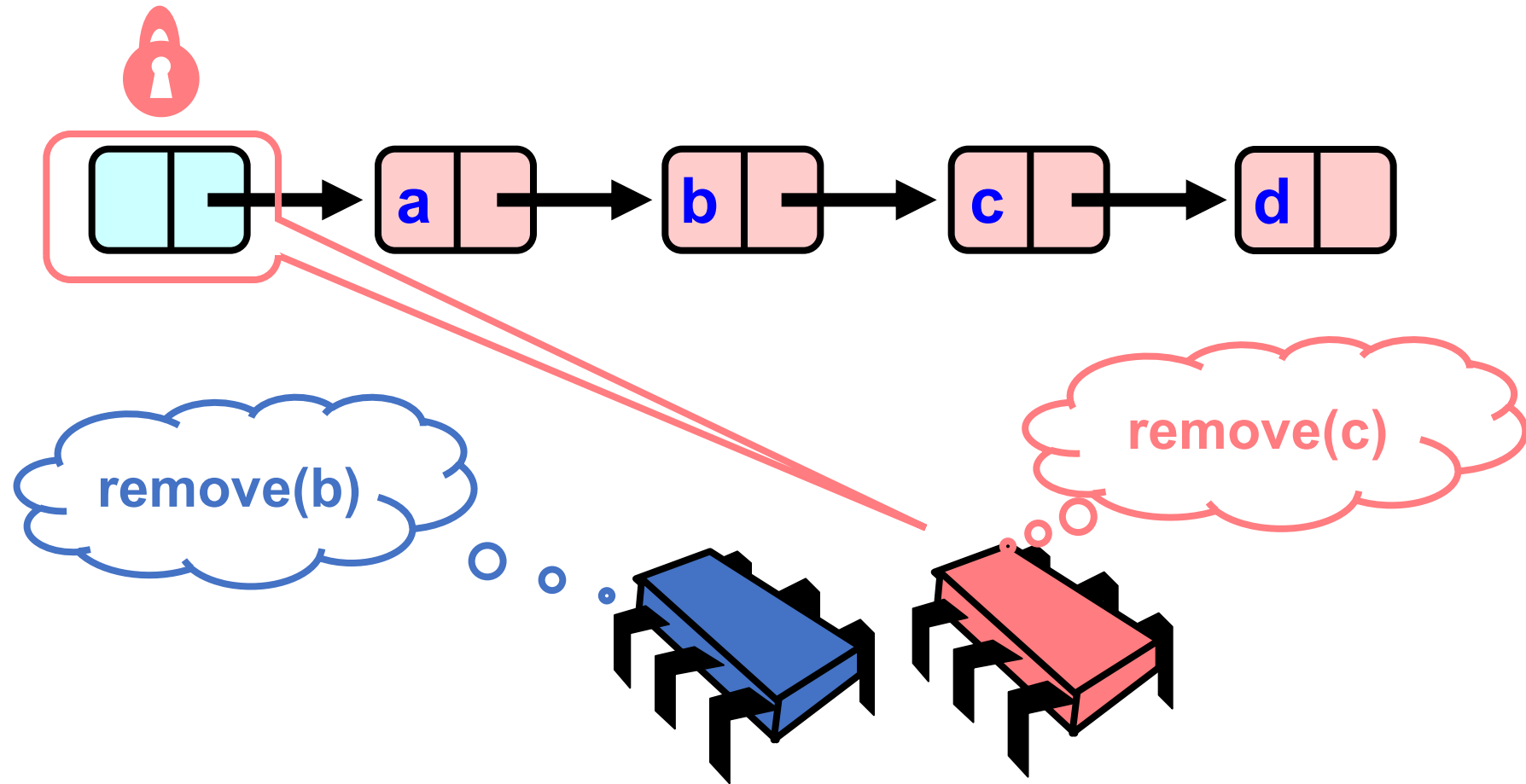
# Fine-grained Locking

- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other
- Requires **careful** thought
  - Acquire all required locks
  - Acquire them in the same order

# Concurrent Removes

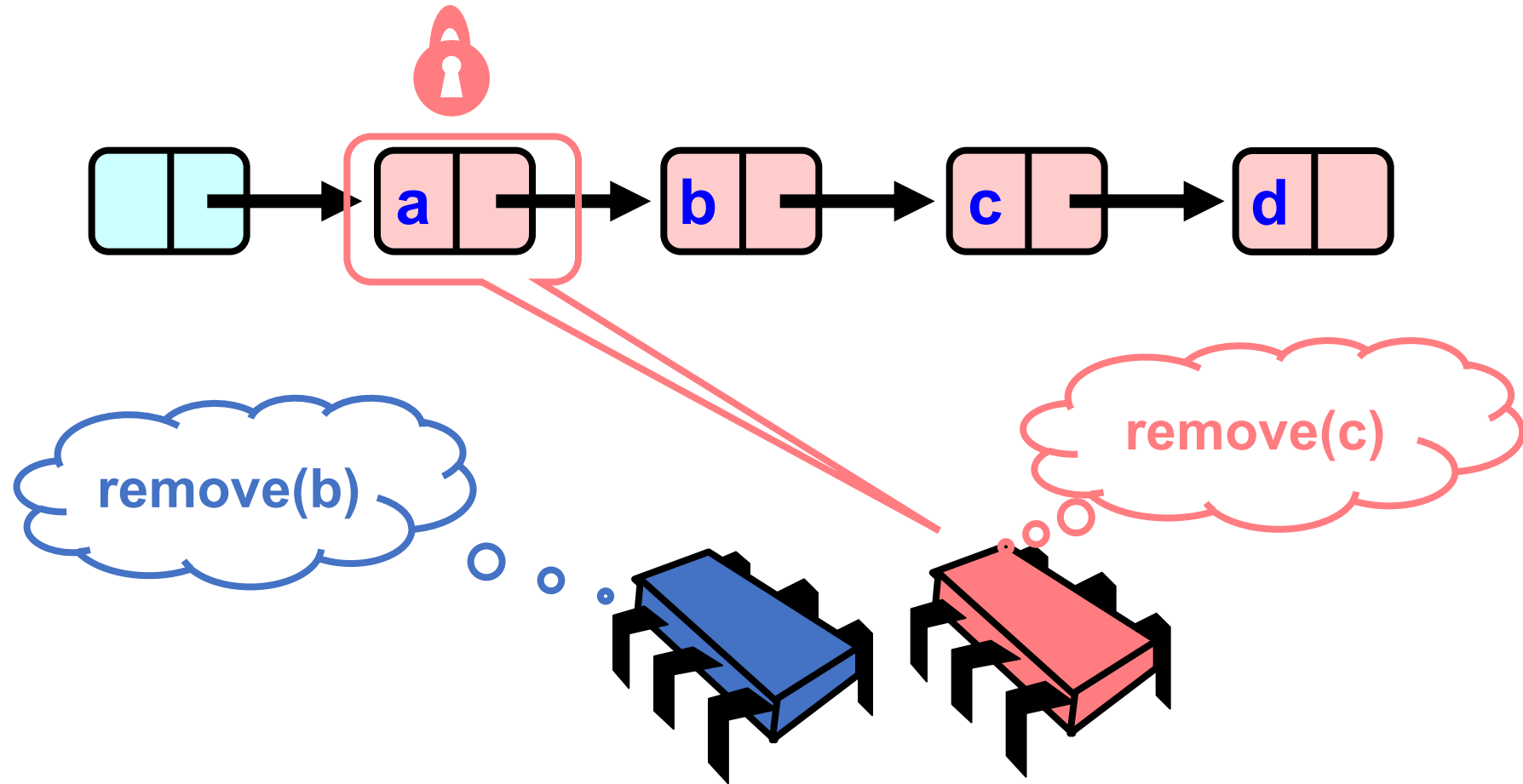


# Concurrent Removes

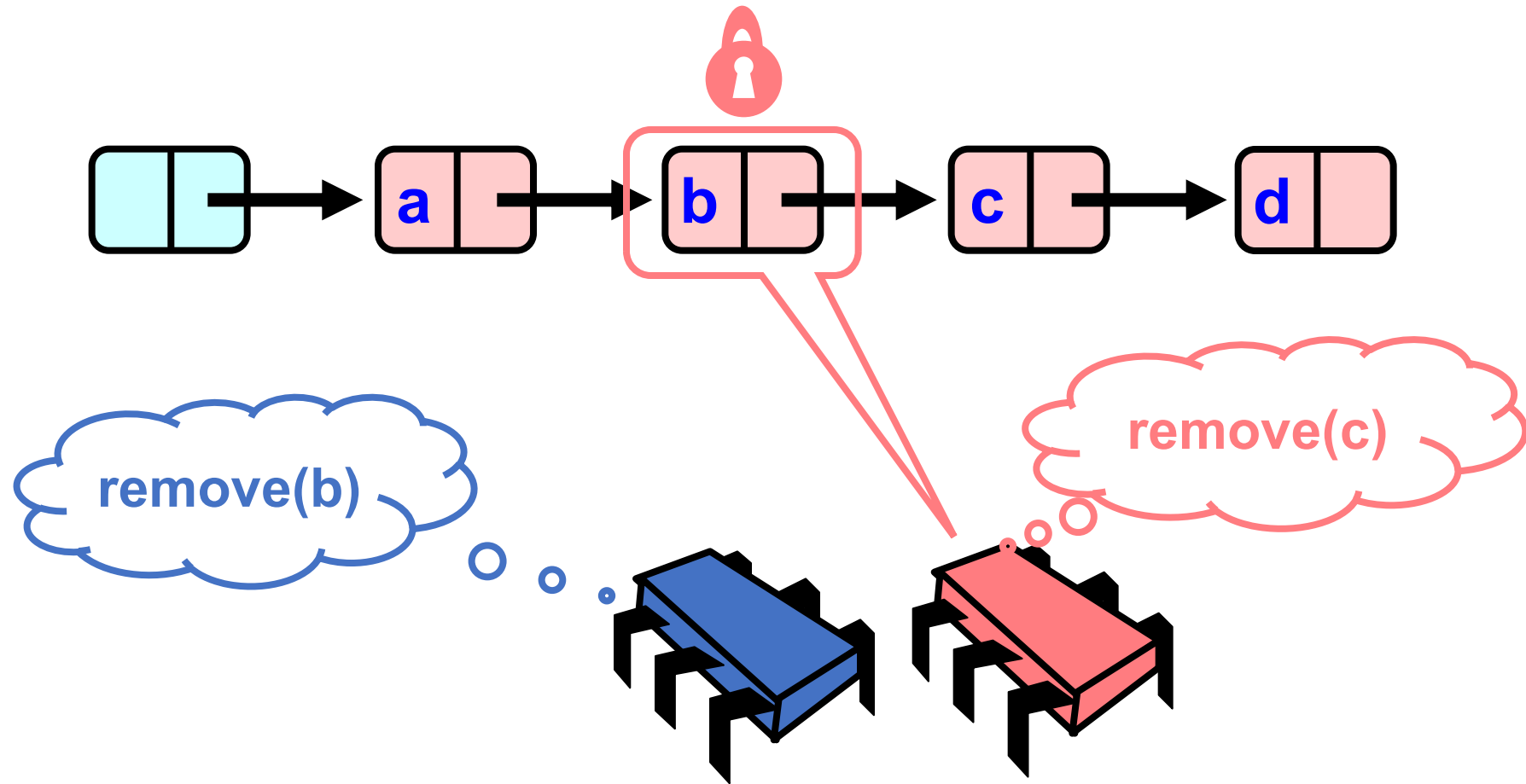




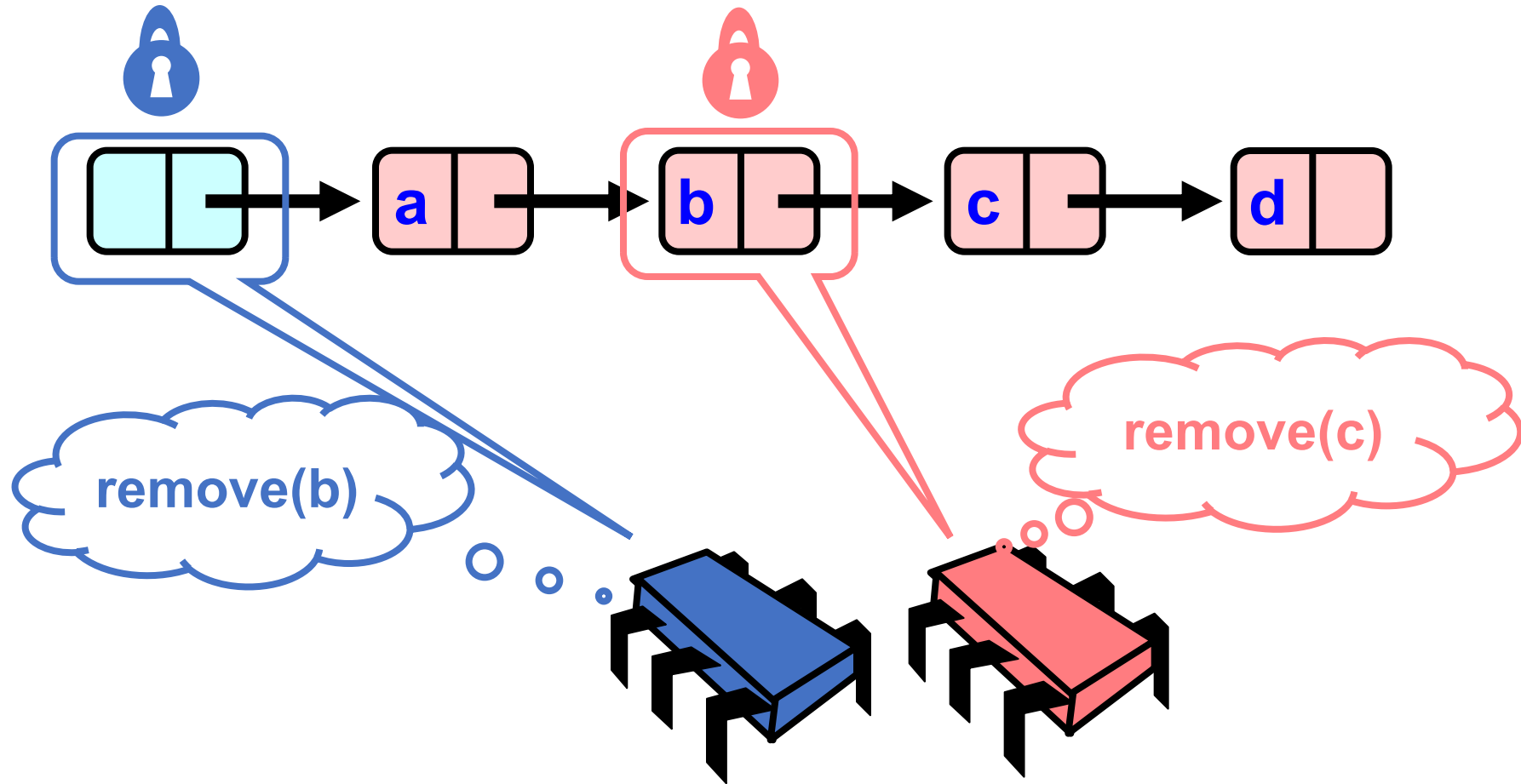
# Concurrent Removes



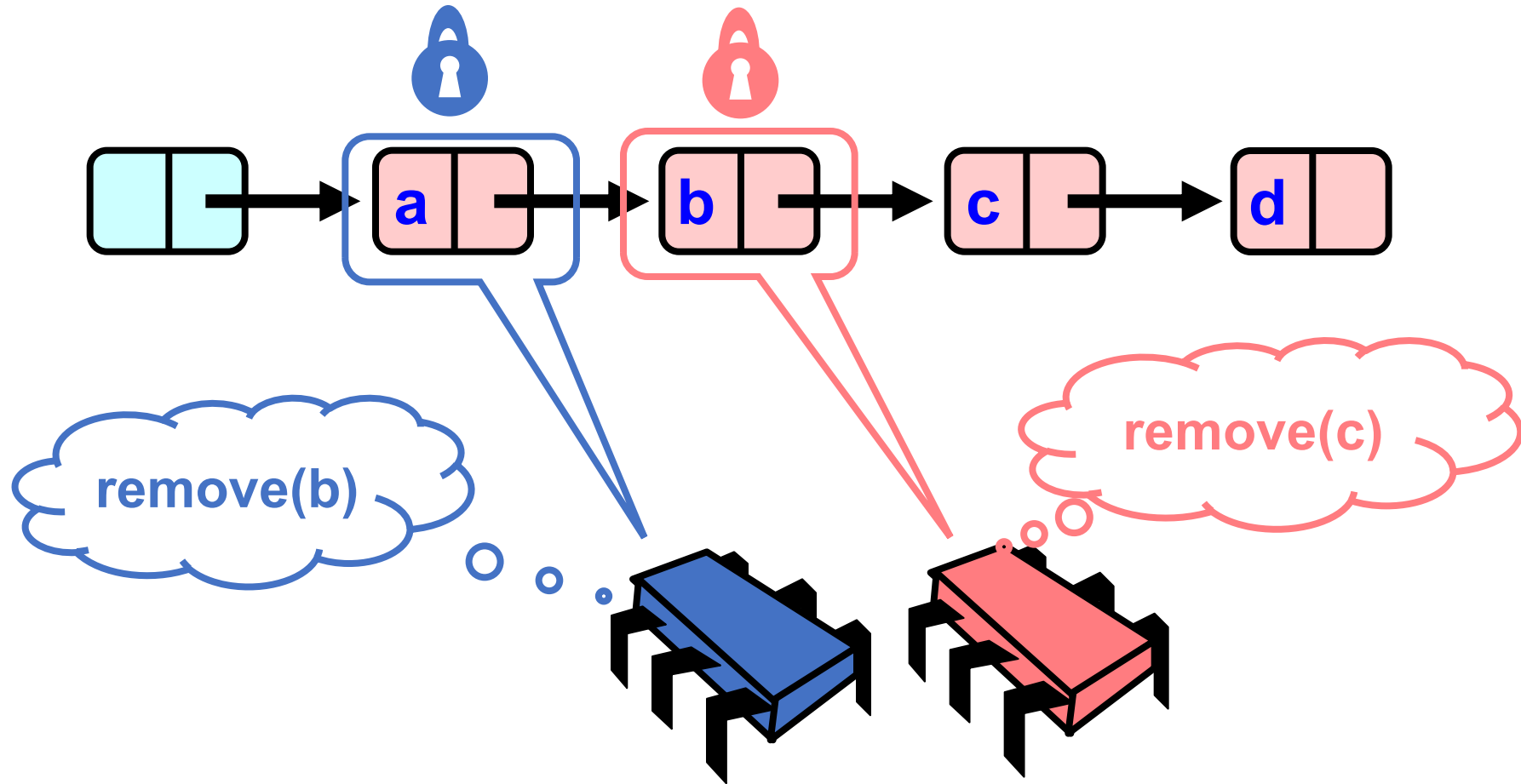
# Concurrent Removes



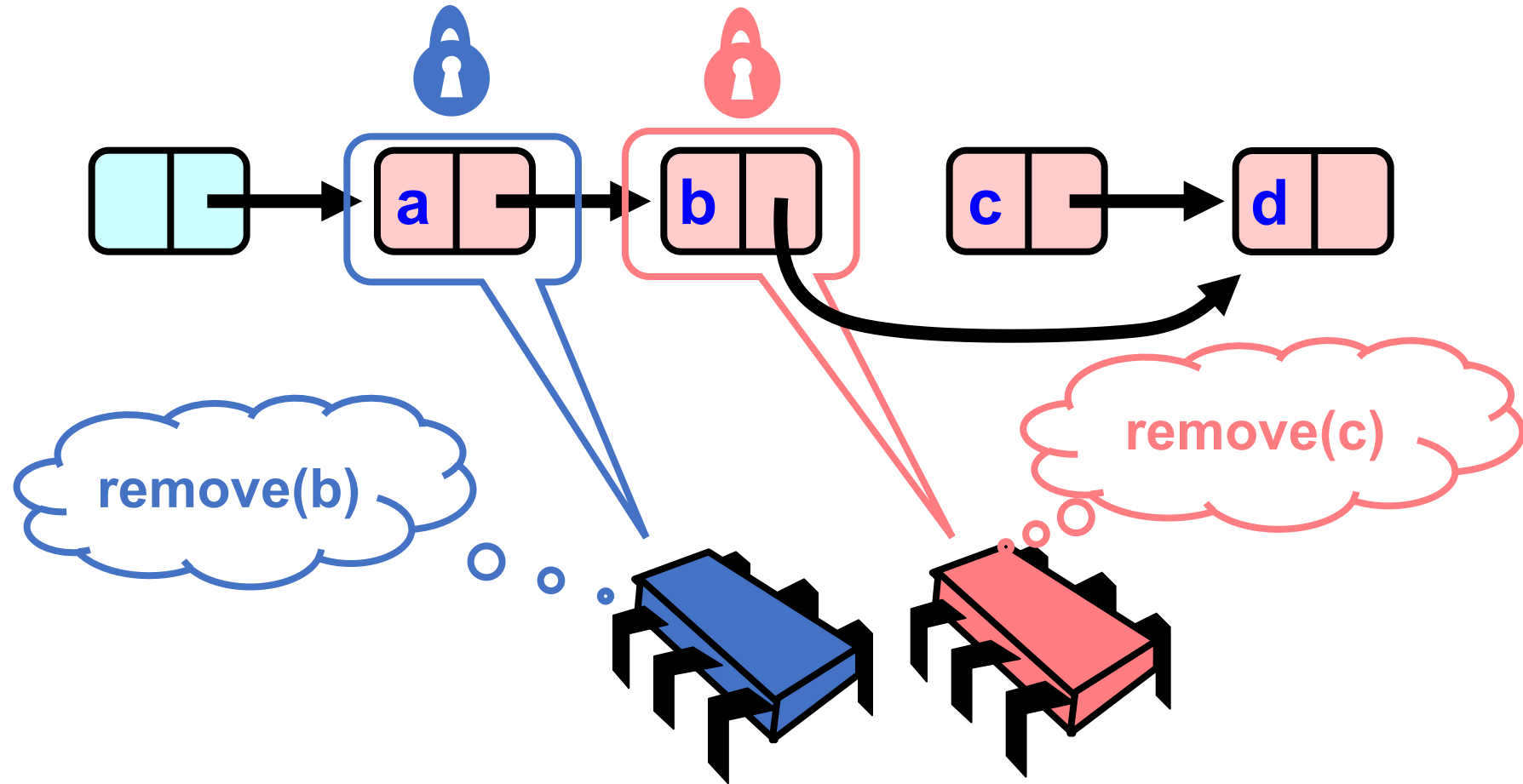
# Concurrent Removes



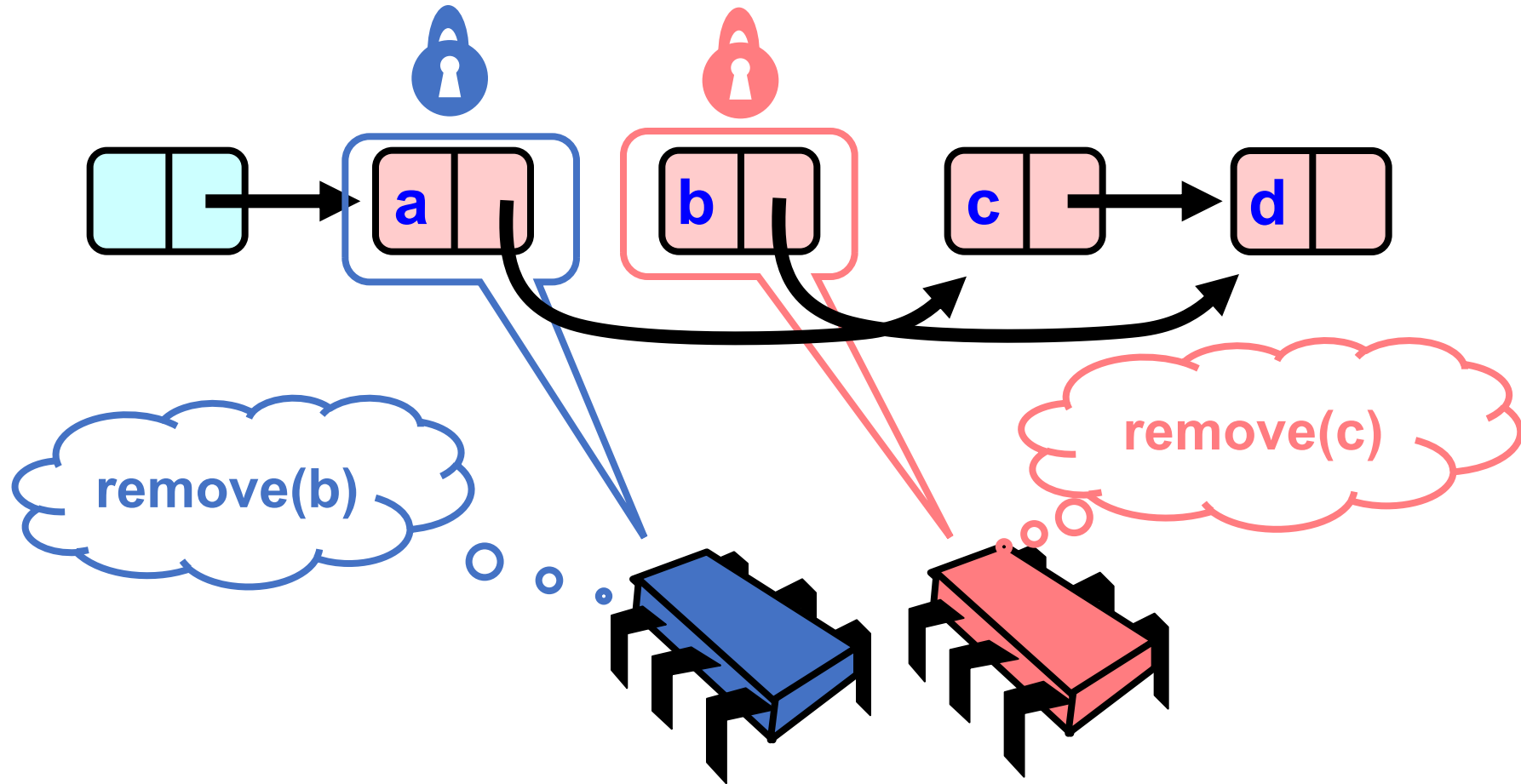
# Concurrent Removes



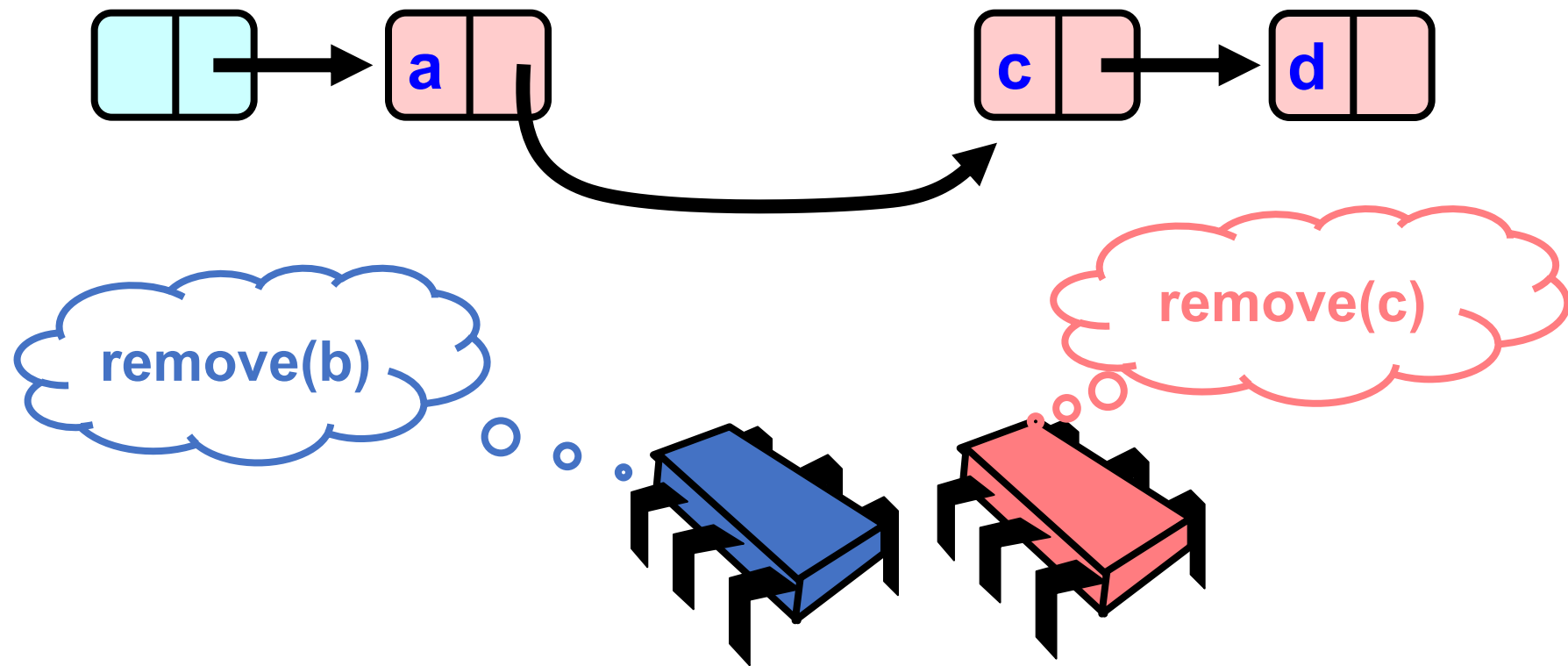
# Concurrent Removes



# Concurrent Removes

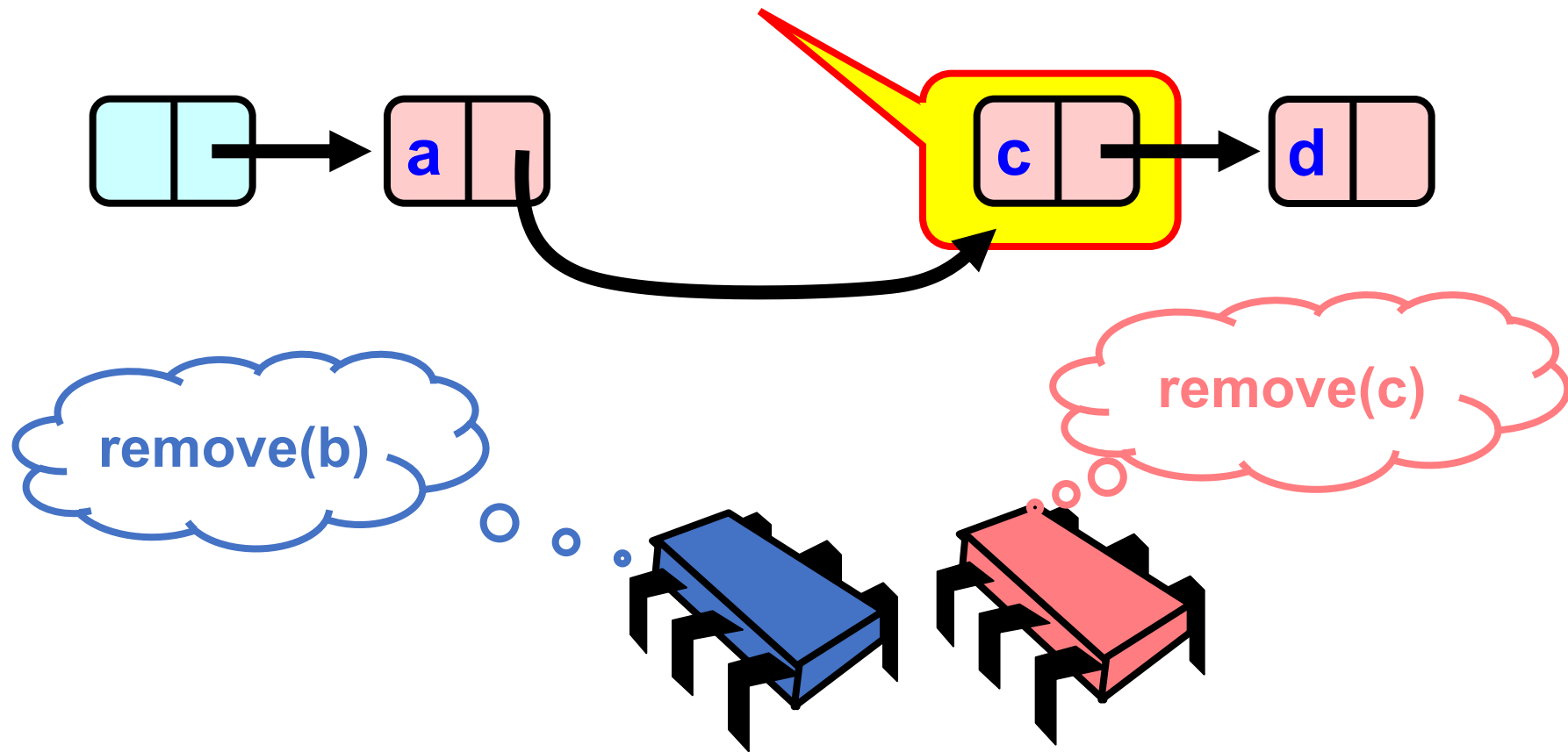


Uh, Oh



Uh, Oh

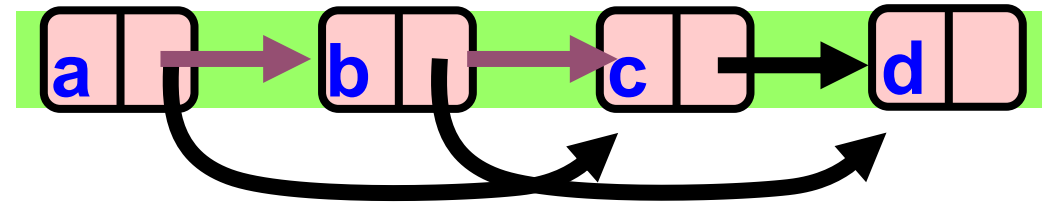
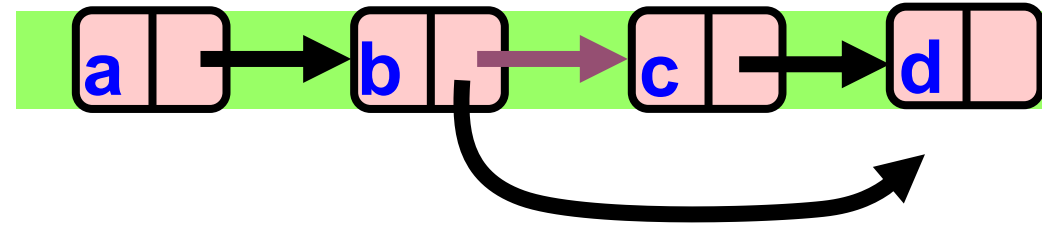
**Bad news, c not removed**



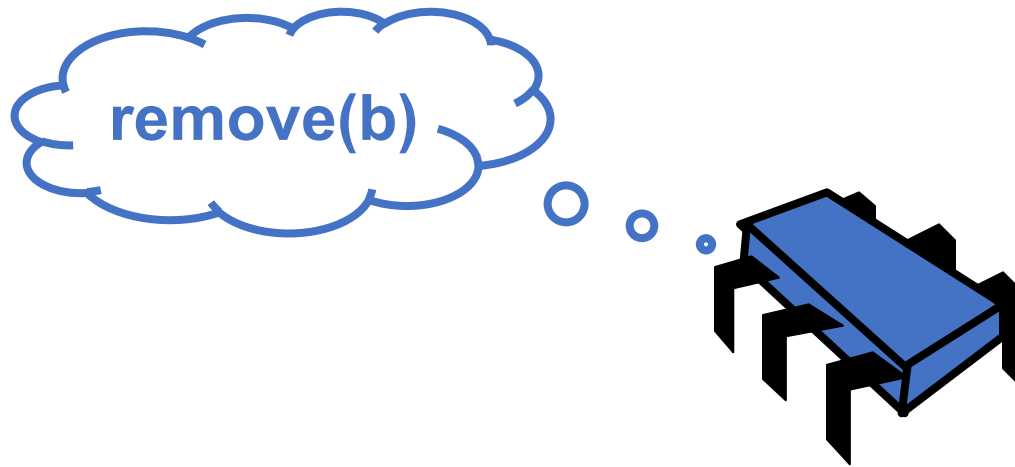
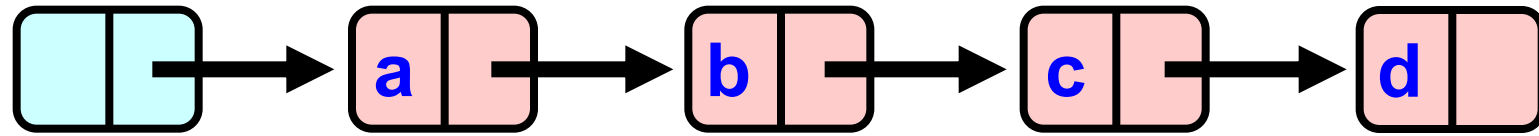


# Problem

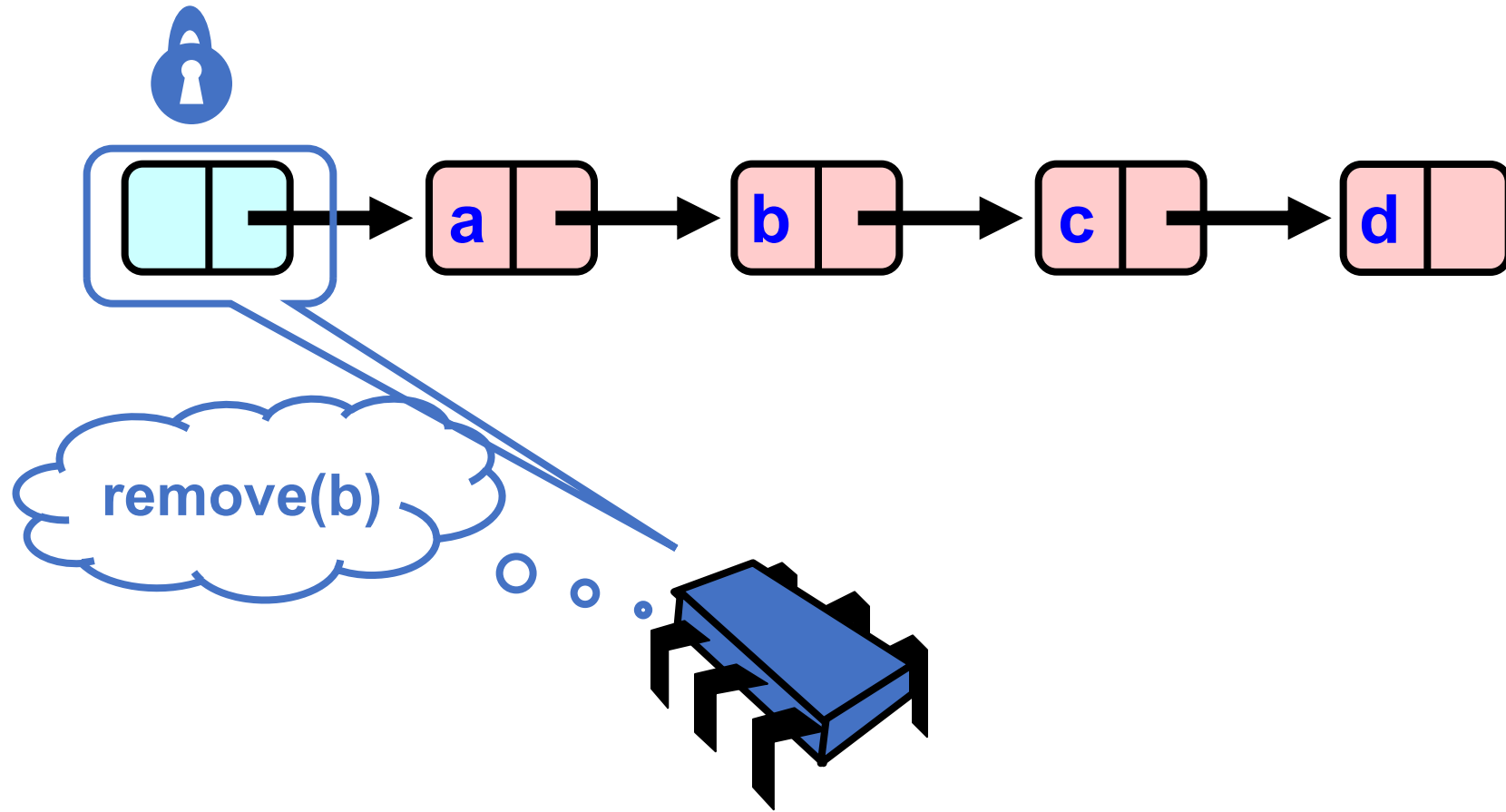
- To delete node c
  - Swing node b's next field to d
- Problem is,
  - Someone deleting b concurrently could direct a pointer to C



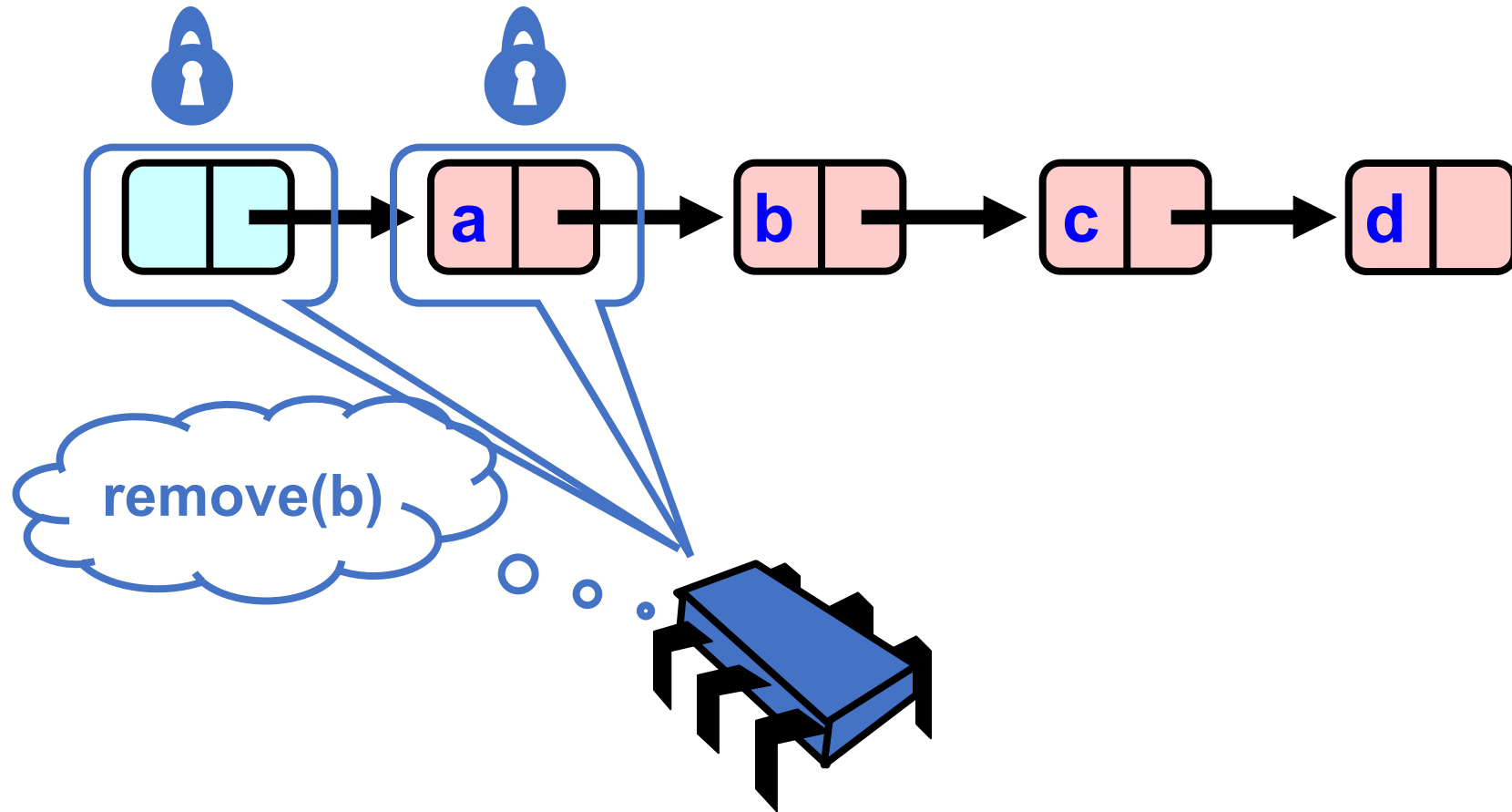
# Hand-Over-Hand Again



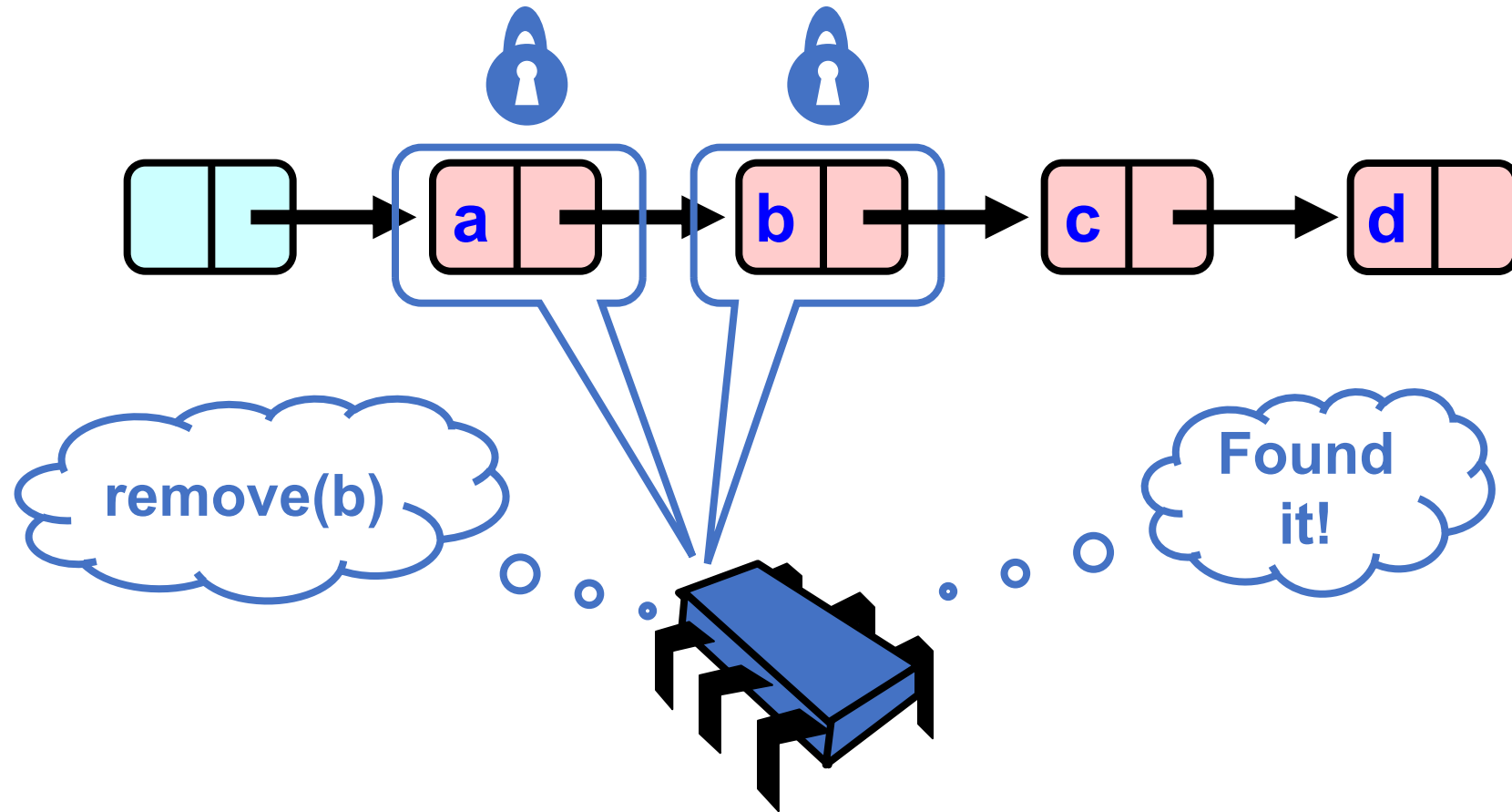
# Hand-Over-Hand Again



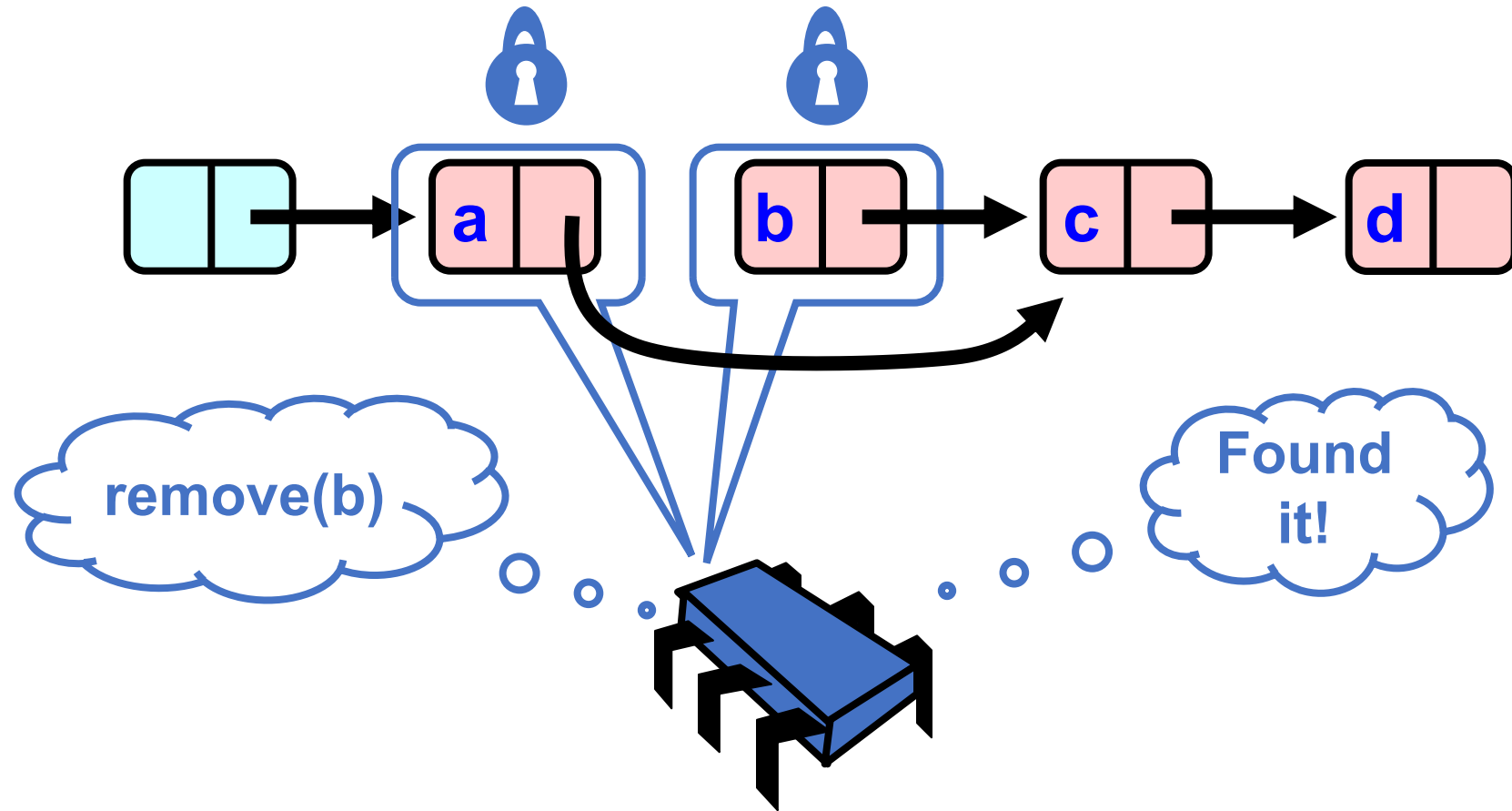
# Hand-Over-Hand Again



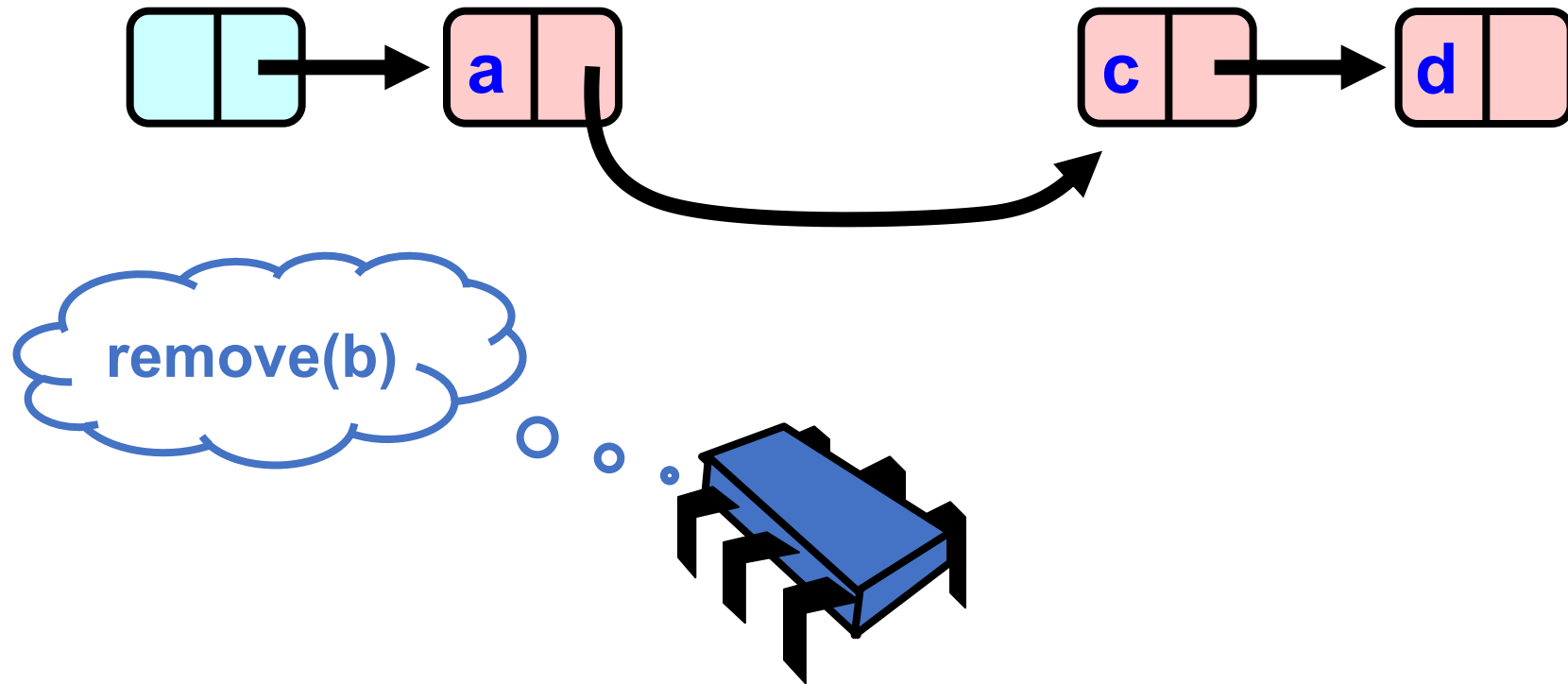
# Hand-Over-Hand Again



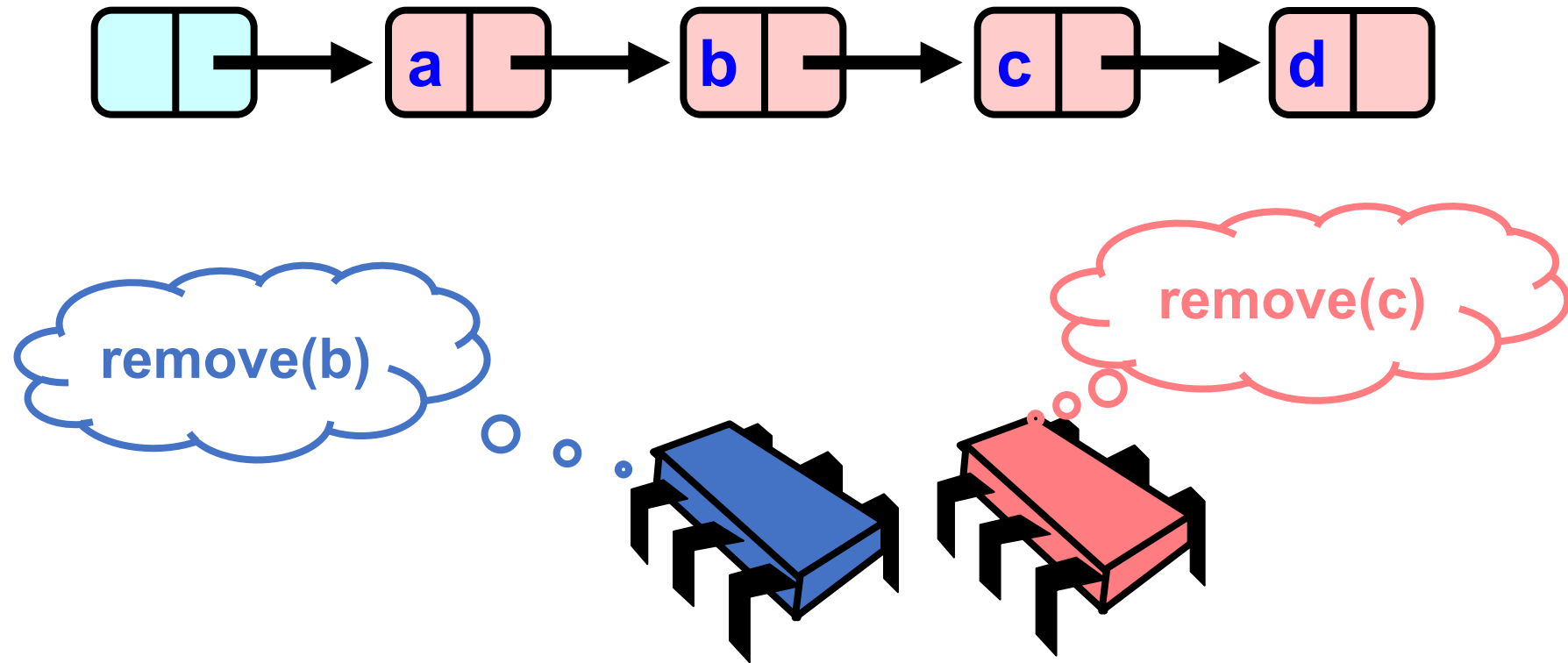
# Hand-Over-Hand Again



# Hand-Over-Hand Again

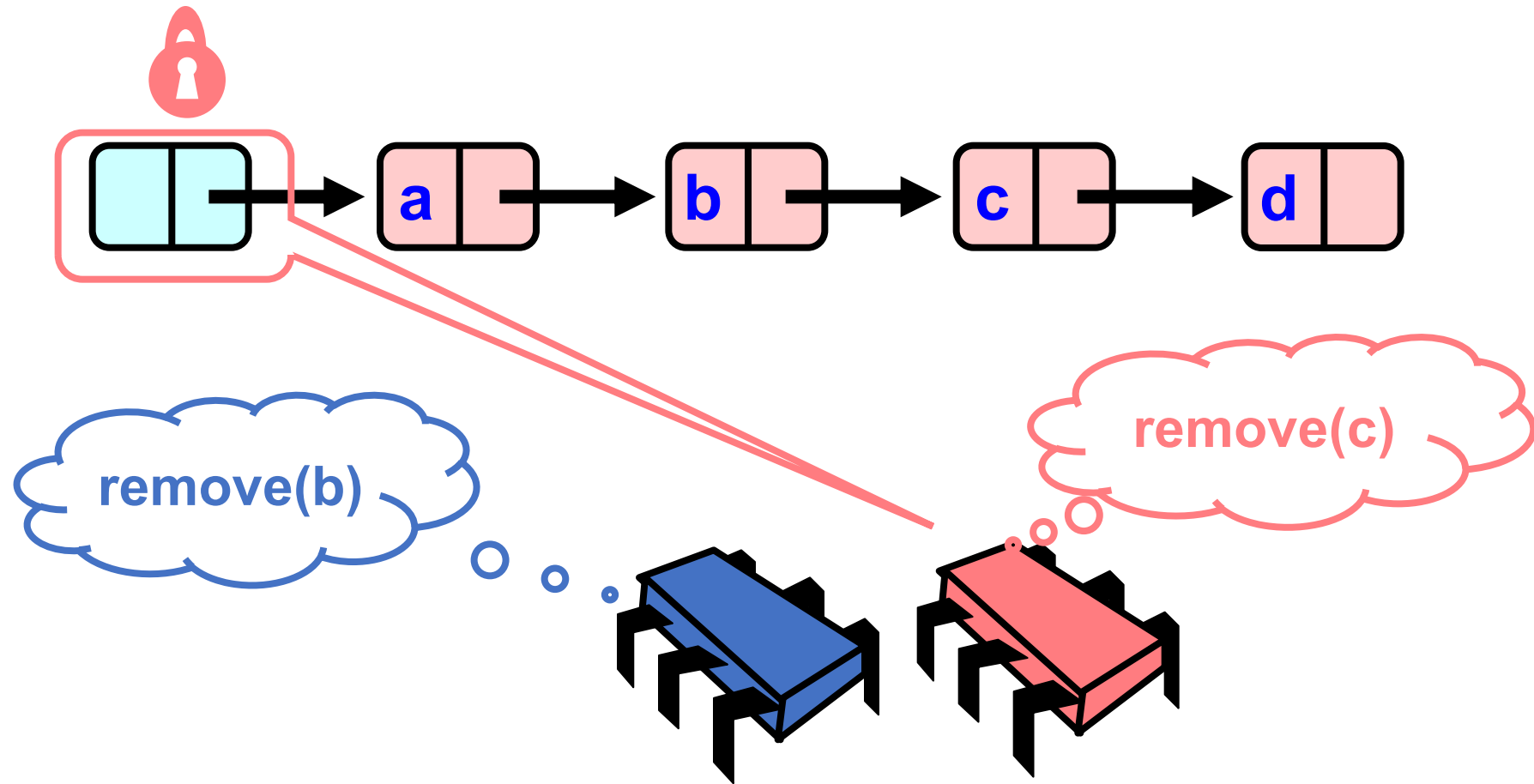


# Removing a Node

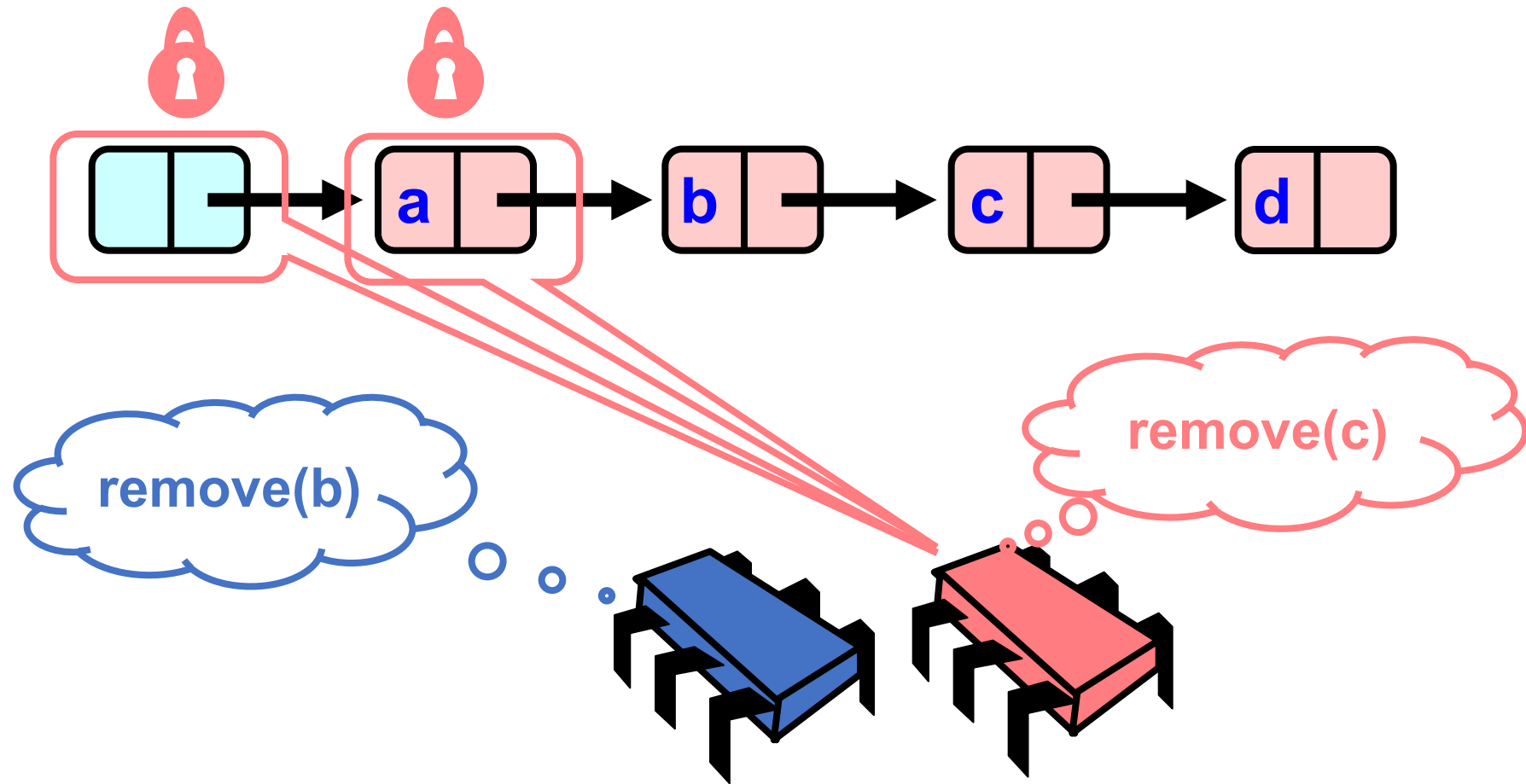




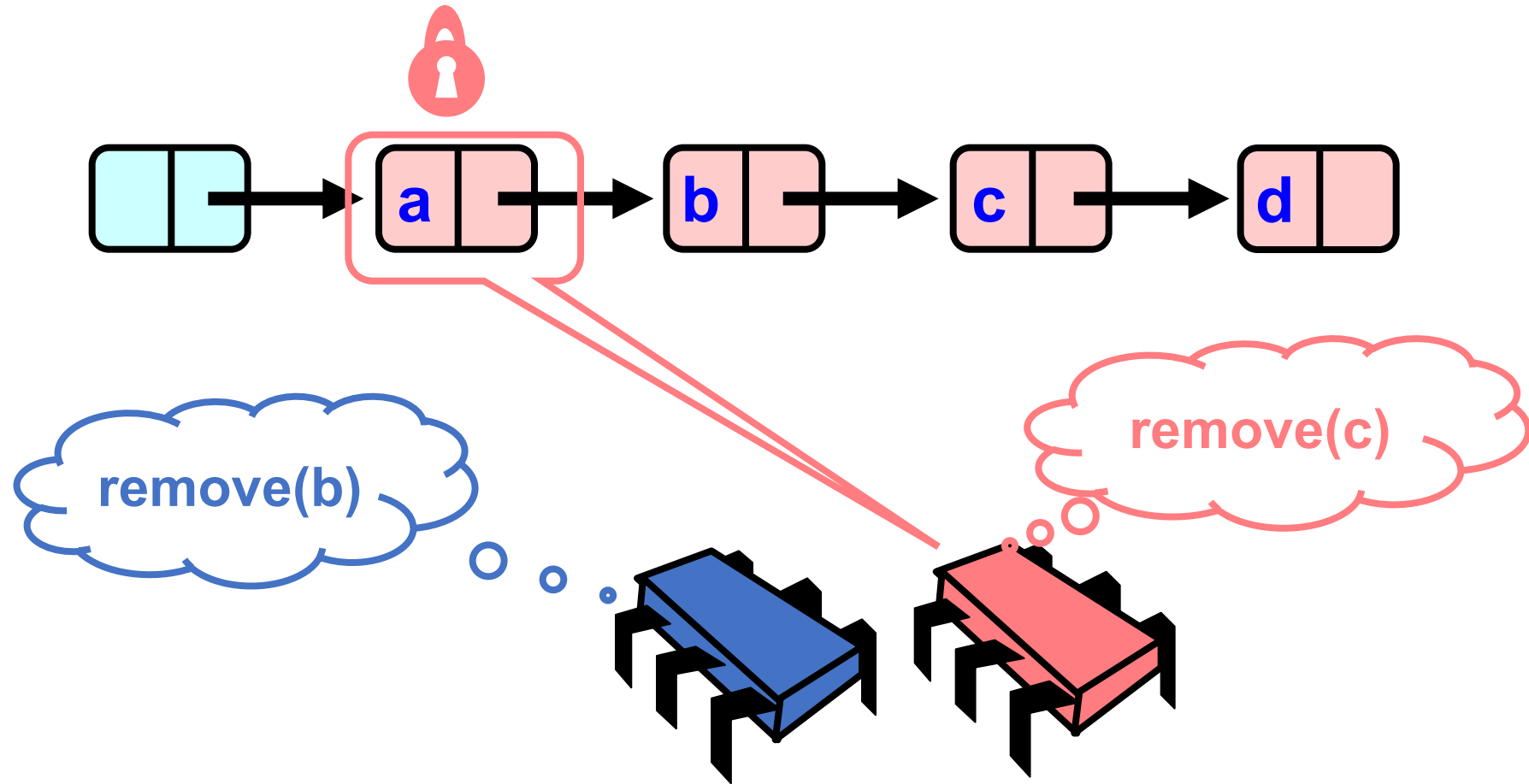
# Removing a Node



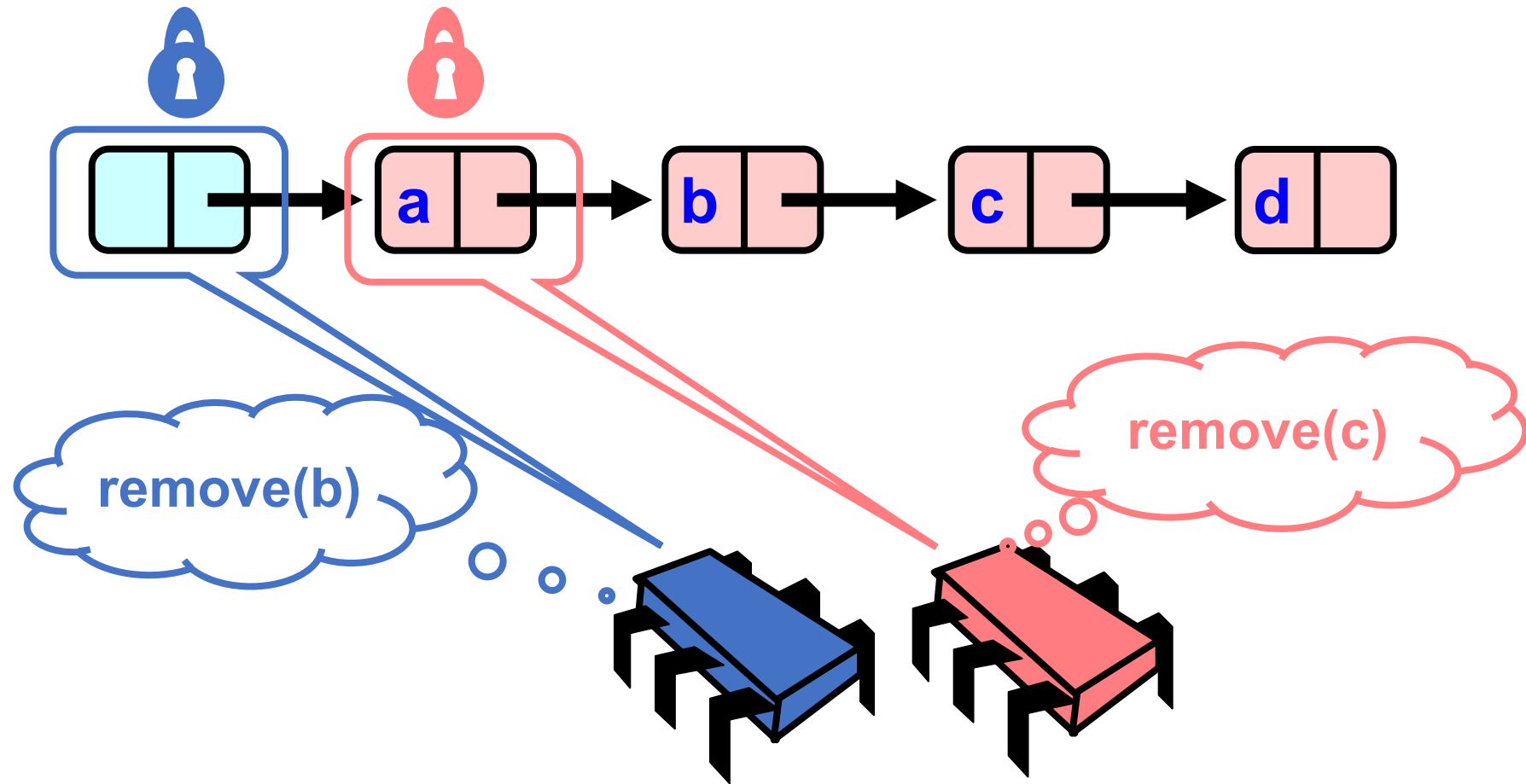
# Removing a Node



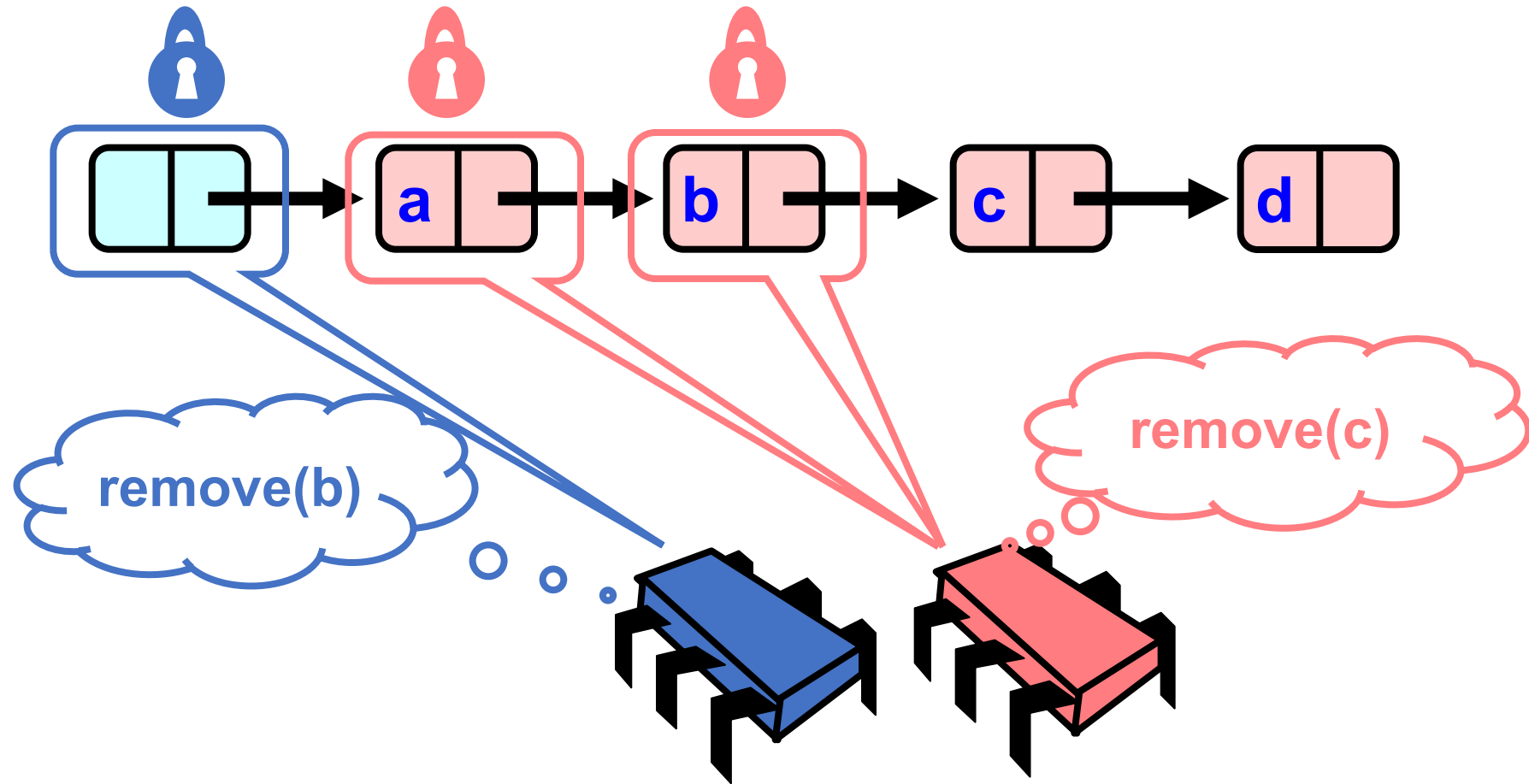
# Removing a Node



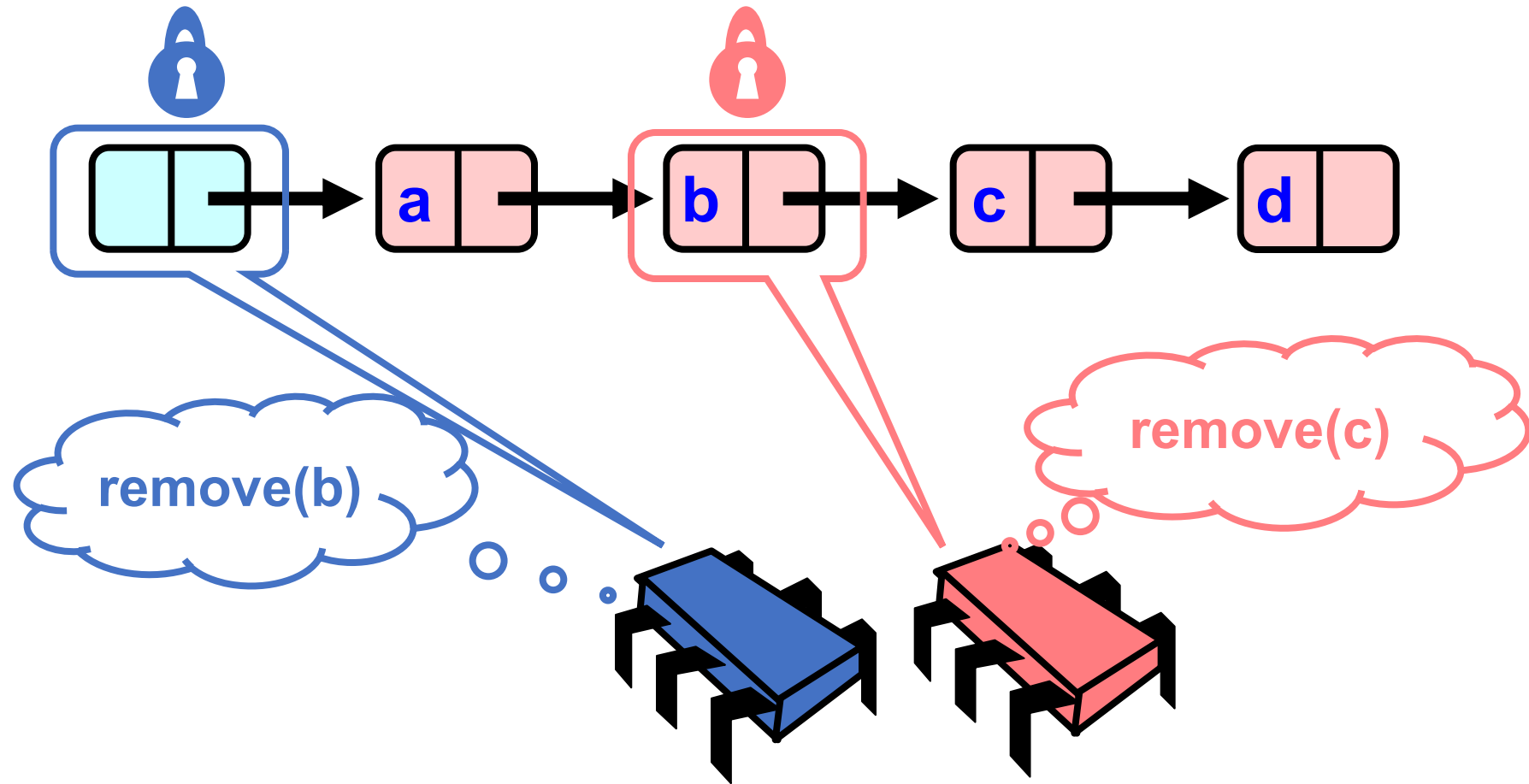
# Removing a Node



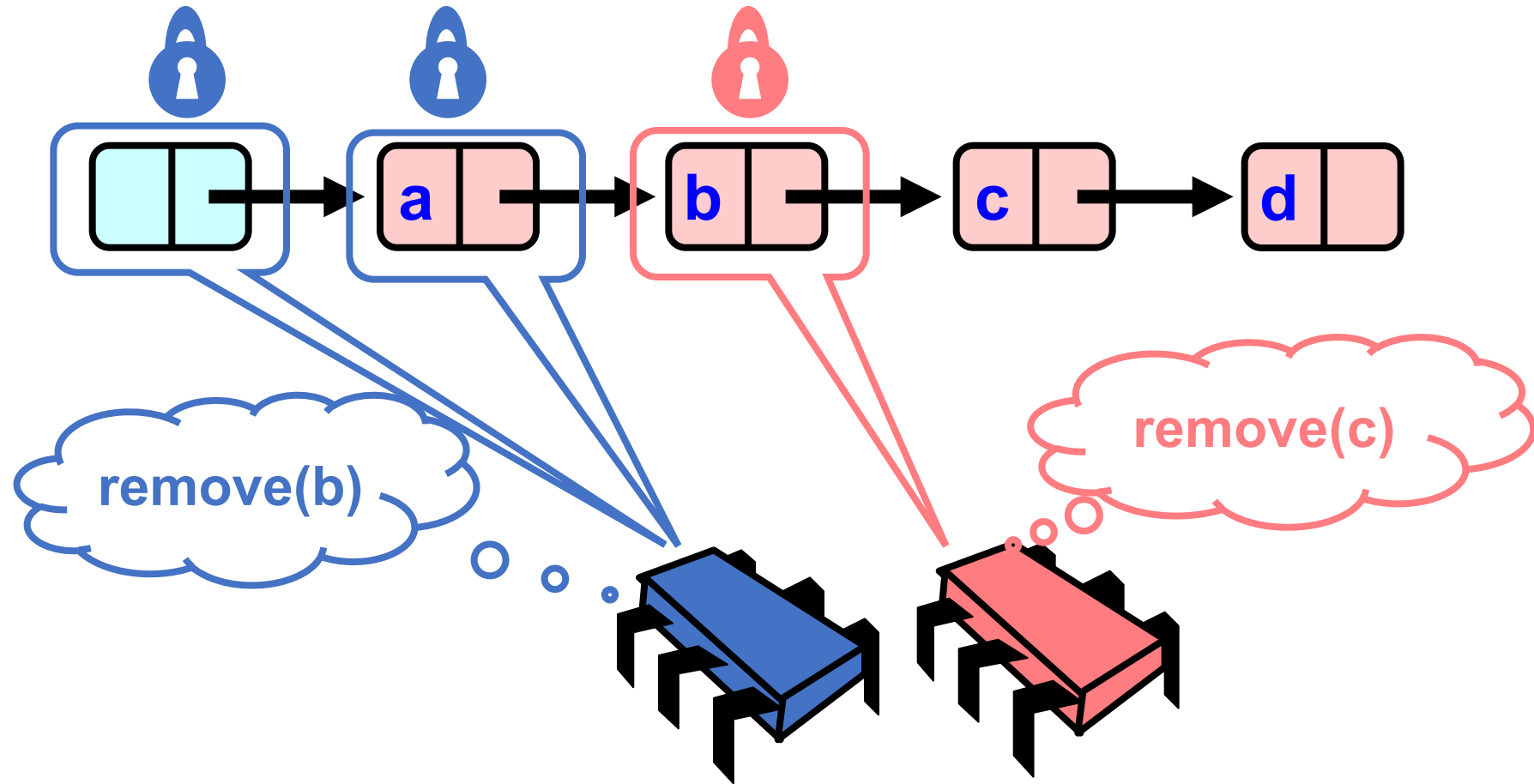
# Removing a Node



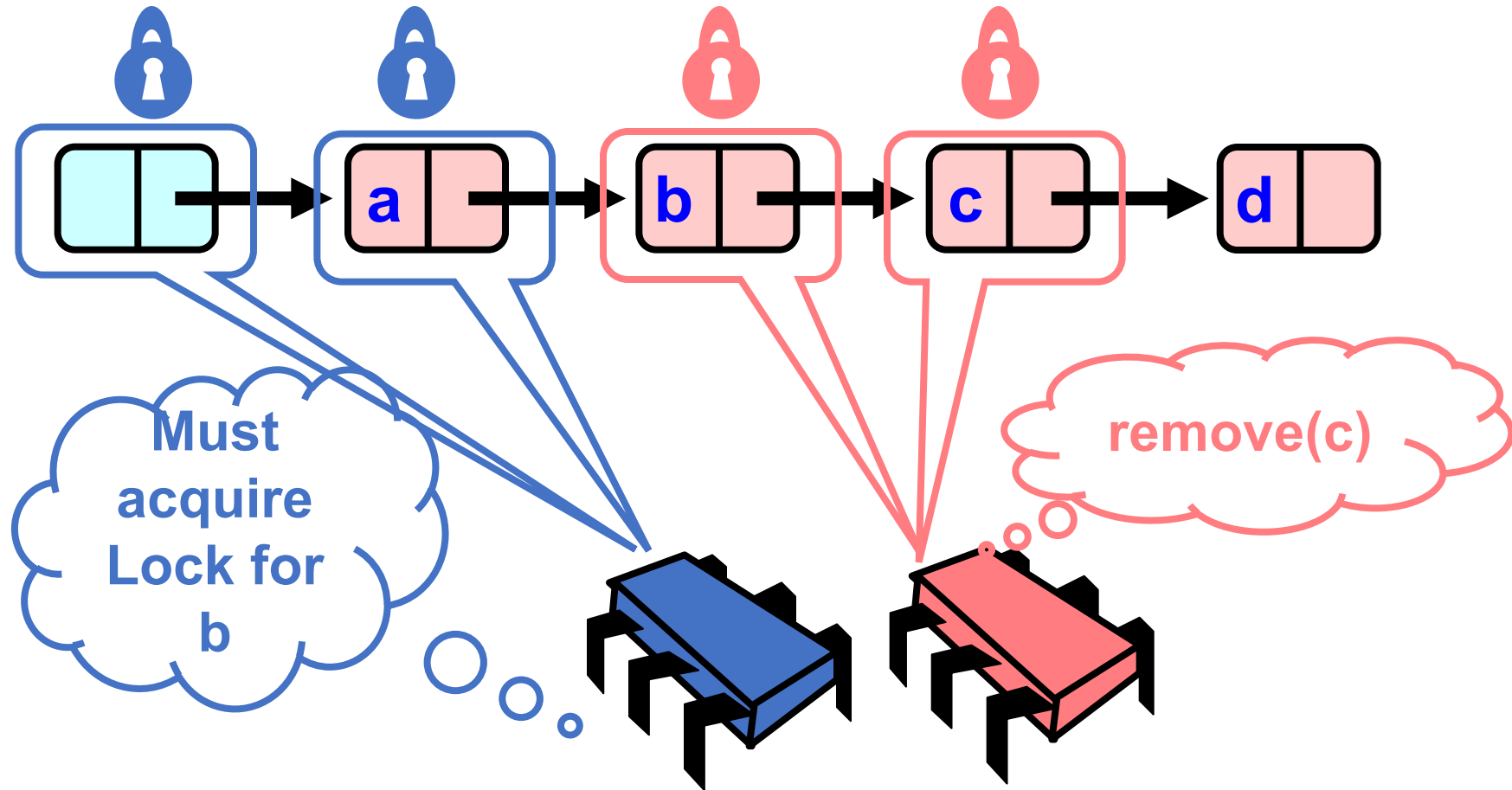
# Removing a Node



# Removing a Node

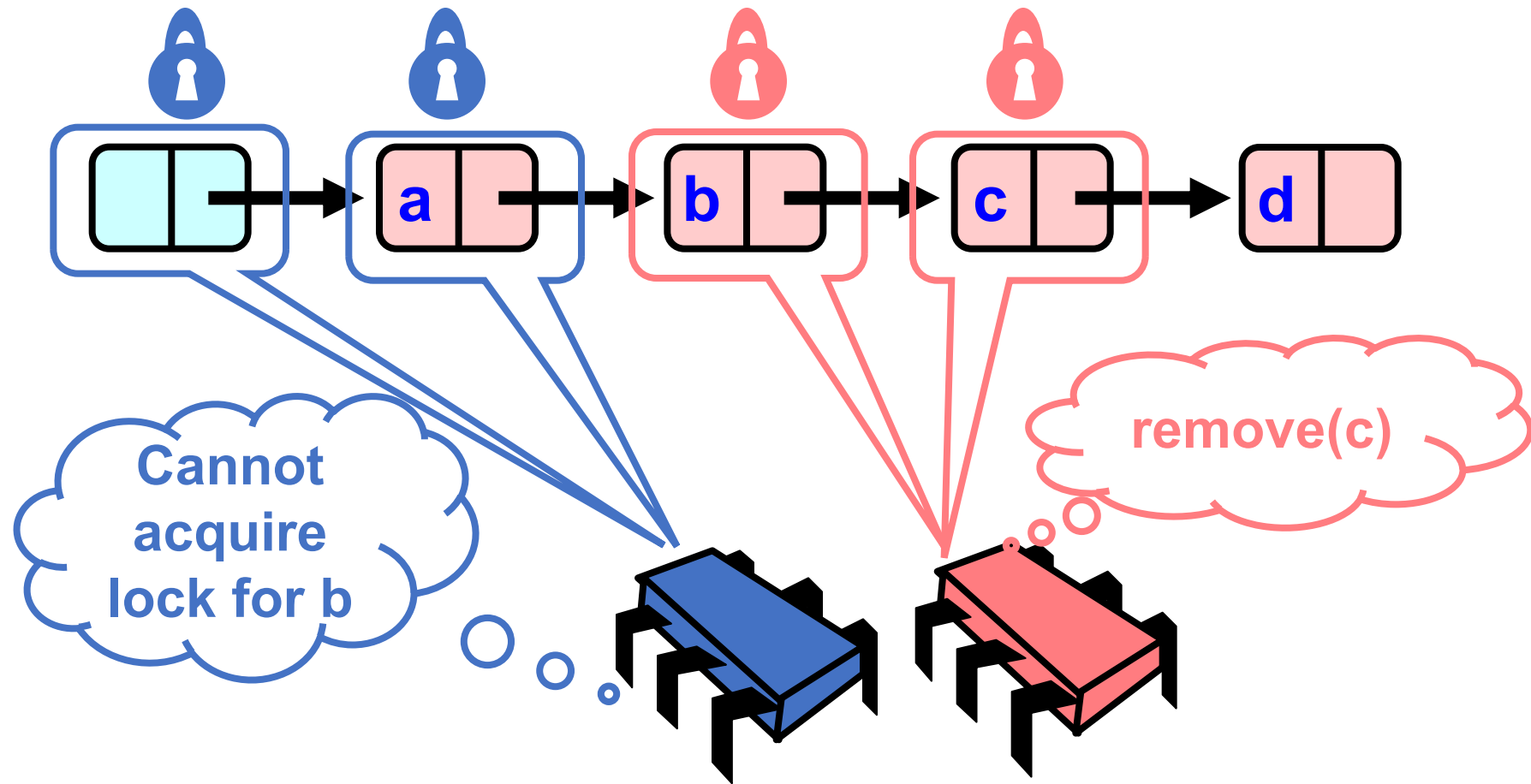


# Removing a Node

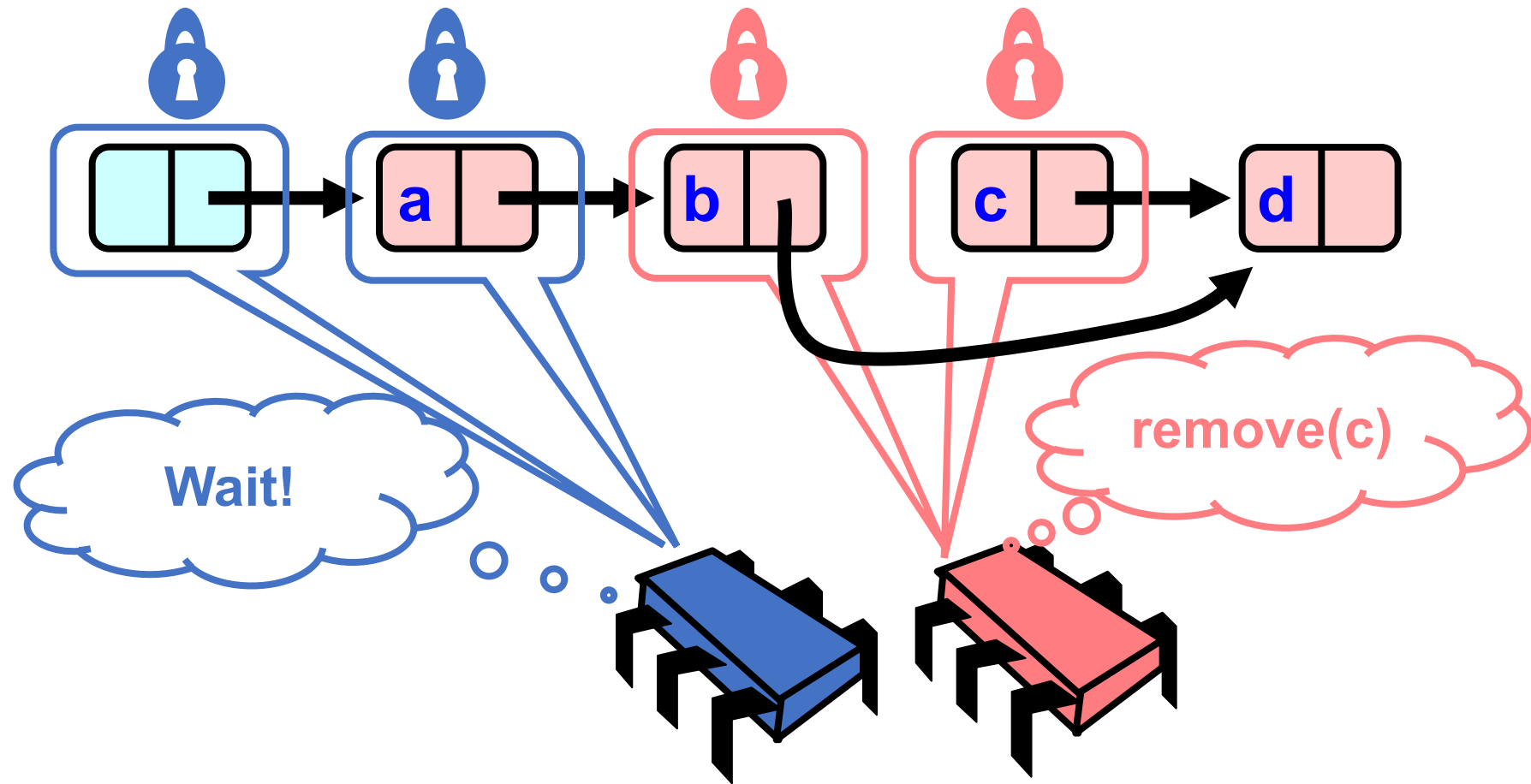




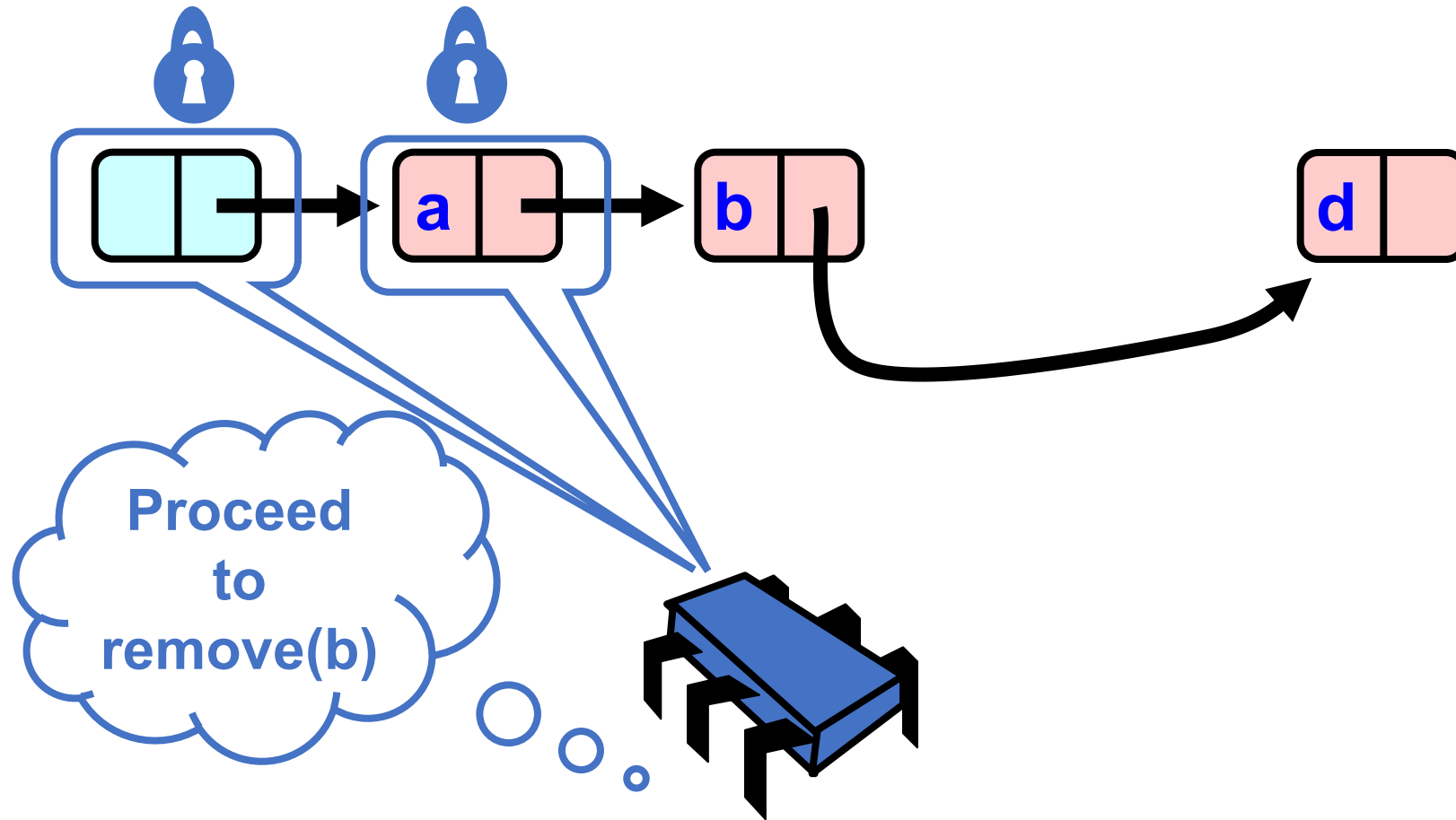
# Removing a Node



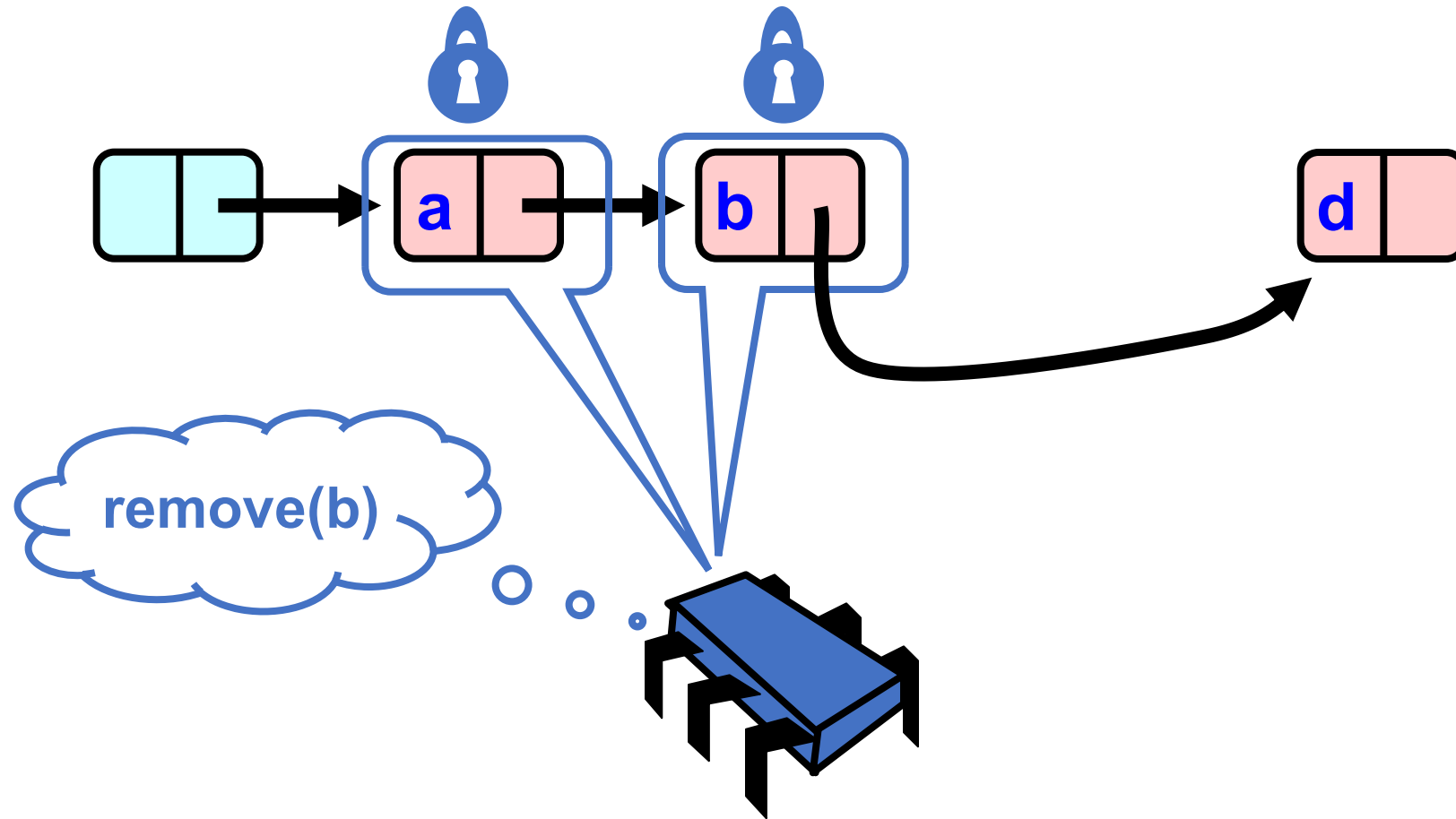
# Removing a Node



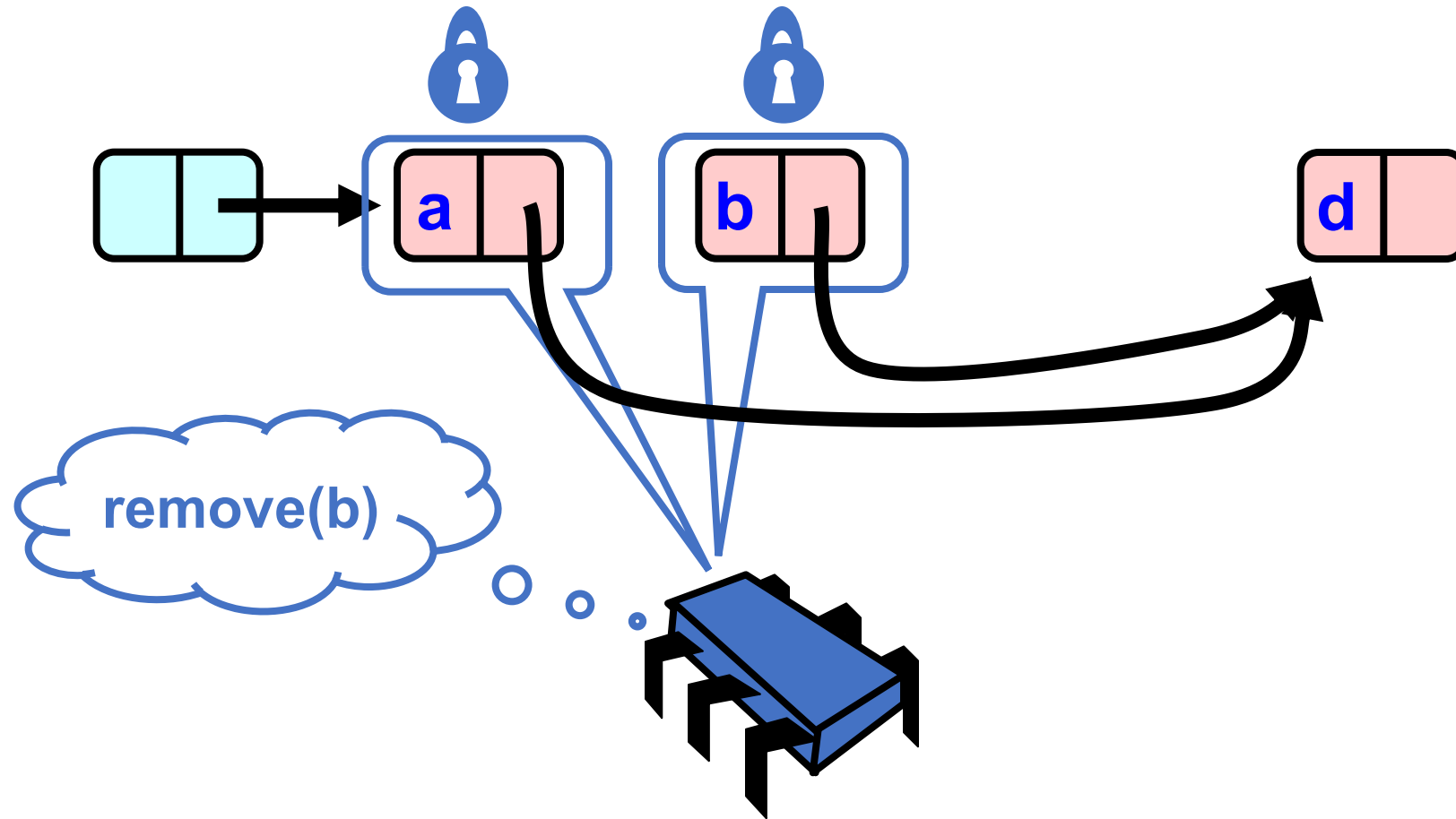
# Removing a Node



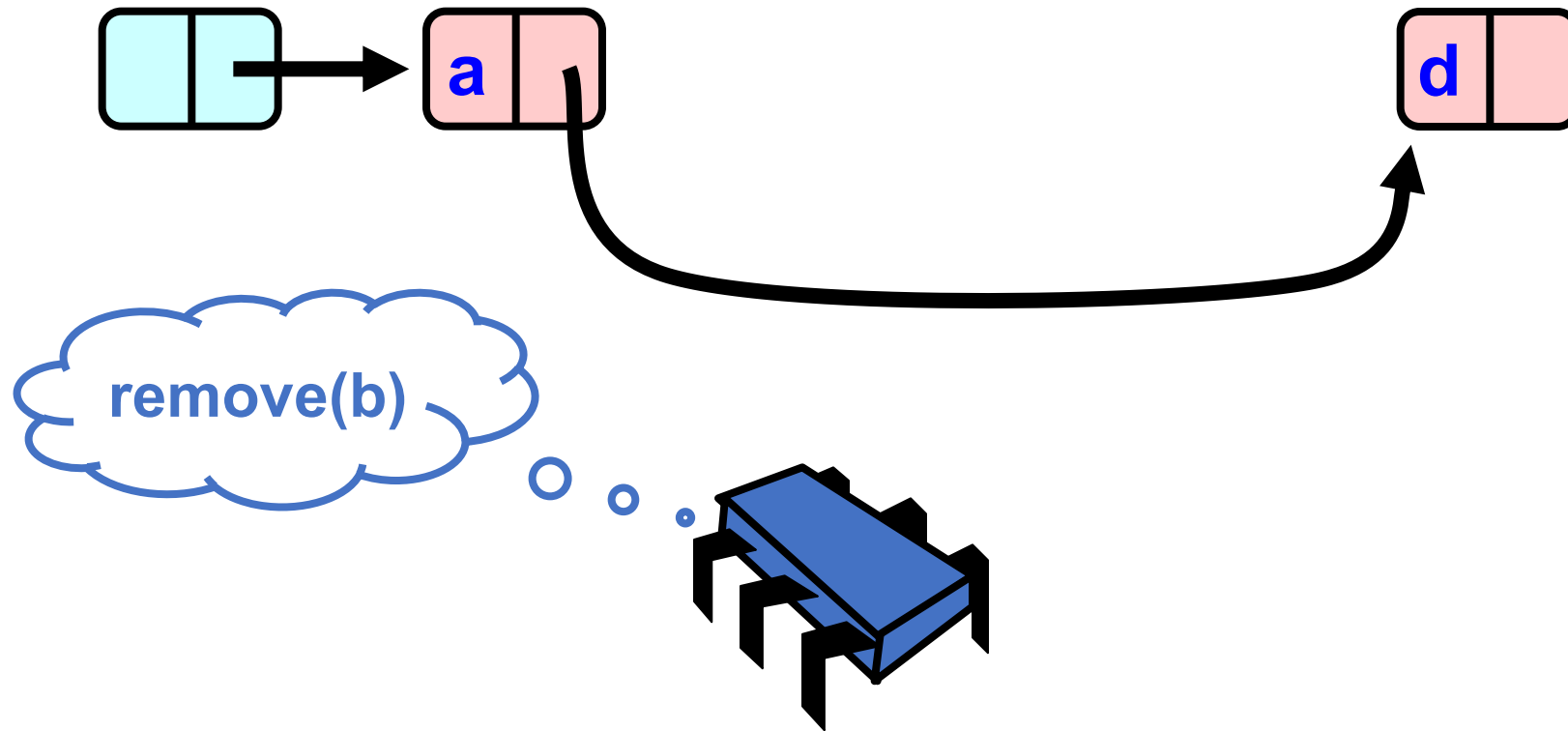
# Removing a Node



# Removing a Node



# Removing a Node



# Removing a Node



# Adding Nodes

- Similar hand-over-hand locking
- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted



# Drawbacks of fine-grained locking

- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Acquires and releases lock for every node traversed. Long chain of acquire/release.
  - Reduces concurrency (traffic jams). Inefficient.

# Optimistic Synchronization

- Assume there will be no conflicts.
- Check before committing.
- If there was a conflict, try again.

# Optimistic Synchronization

- Find nodes without locking. Then, lock the two nodes.

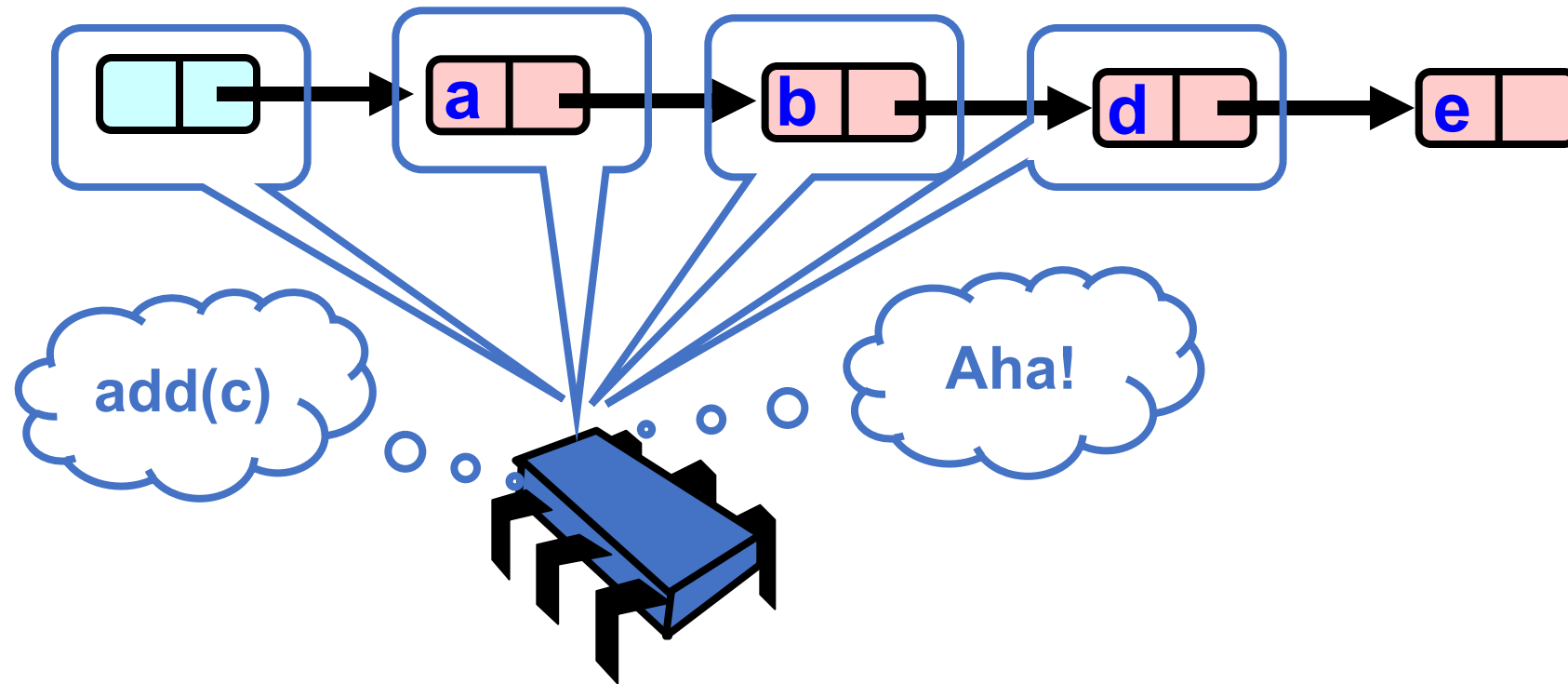
# Optimistic Synchronization

- Find nodes without locking. Then, lock the two nodes.
- Check that the two nodes are still reachable.

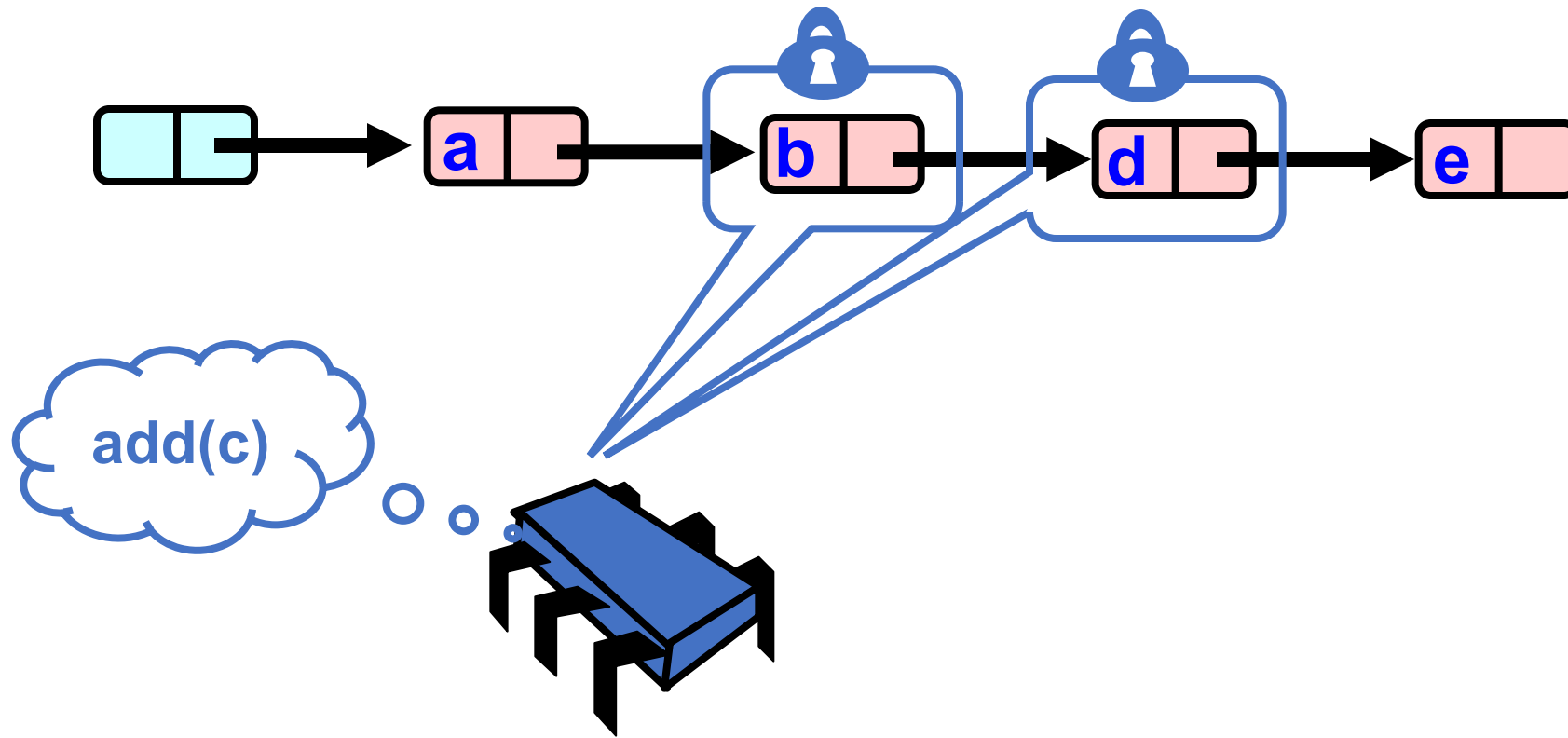
# Optimistic Synchronization

- Find nodes without locking. Then, lock the two nodes.
- Check that the two nodes are still reachable.
- If they are not, search again from the beginning of the list.

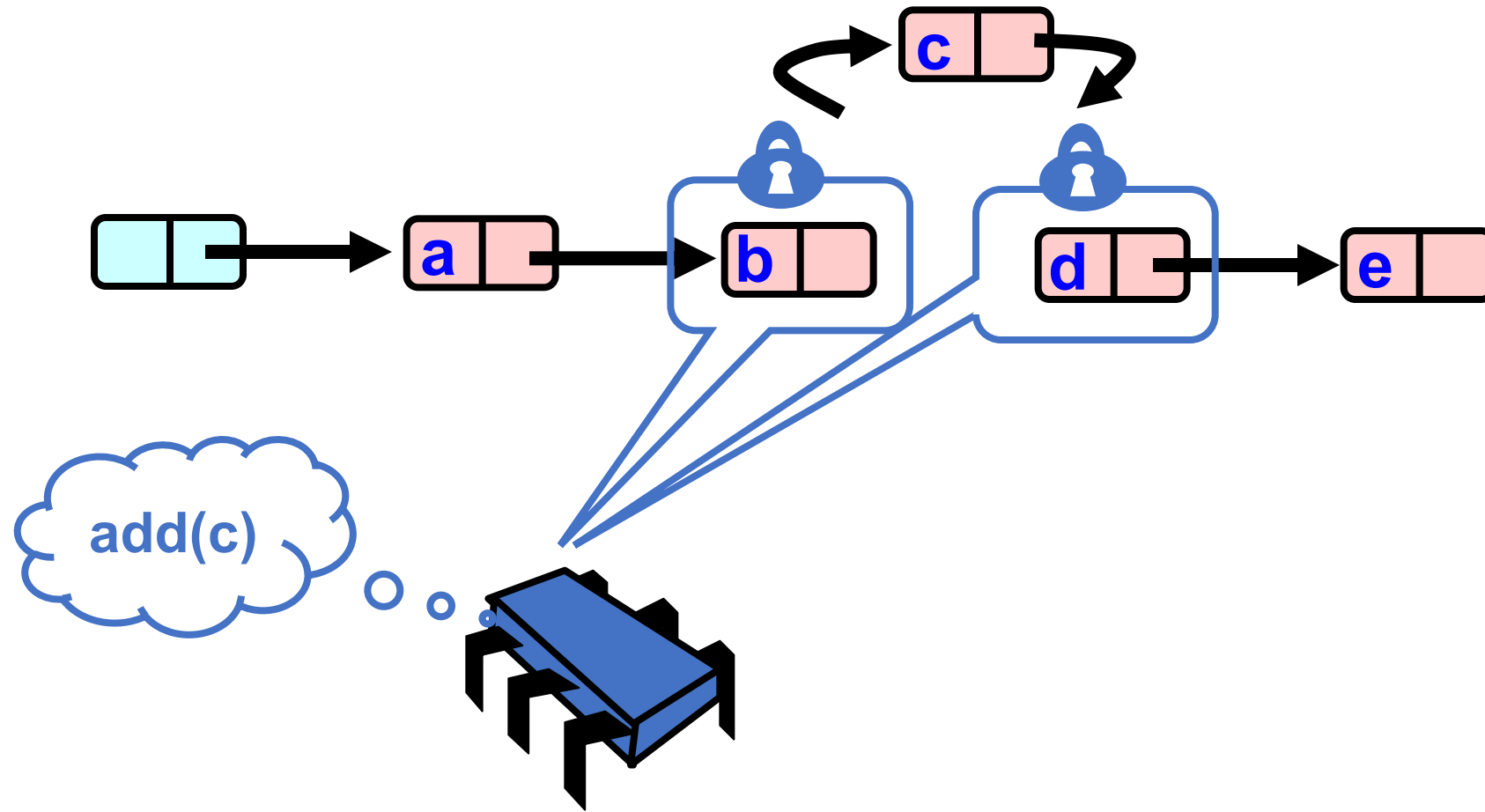
# Optimistic: Traverse without Locking



# Optimistic: Lock

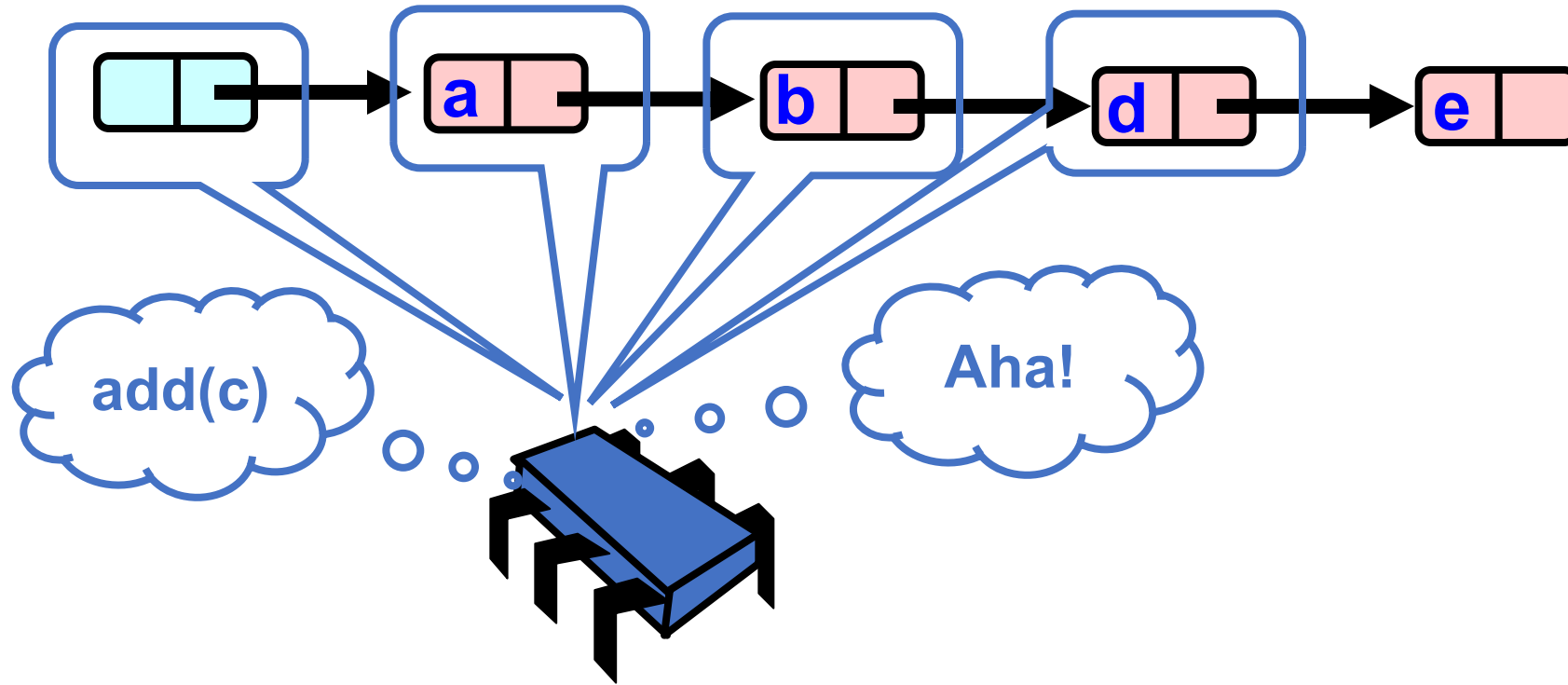


# Optimistic: Insert without checking

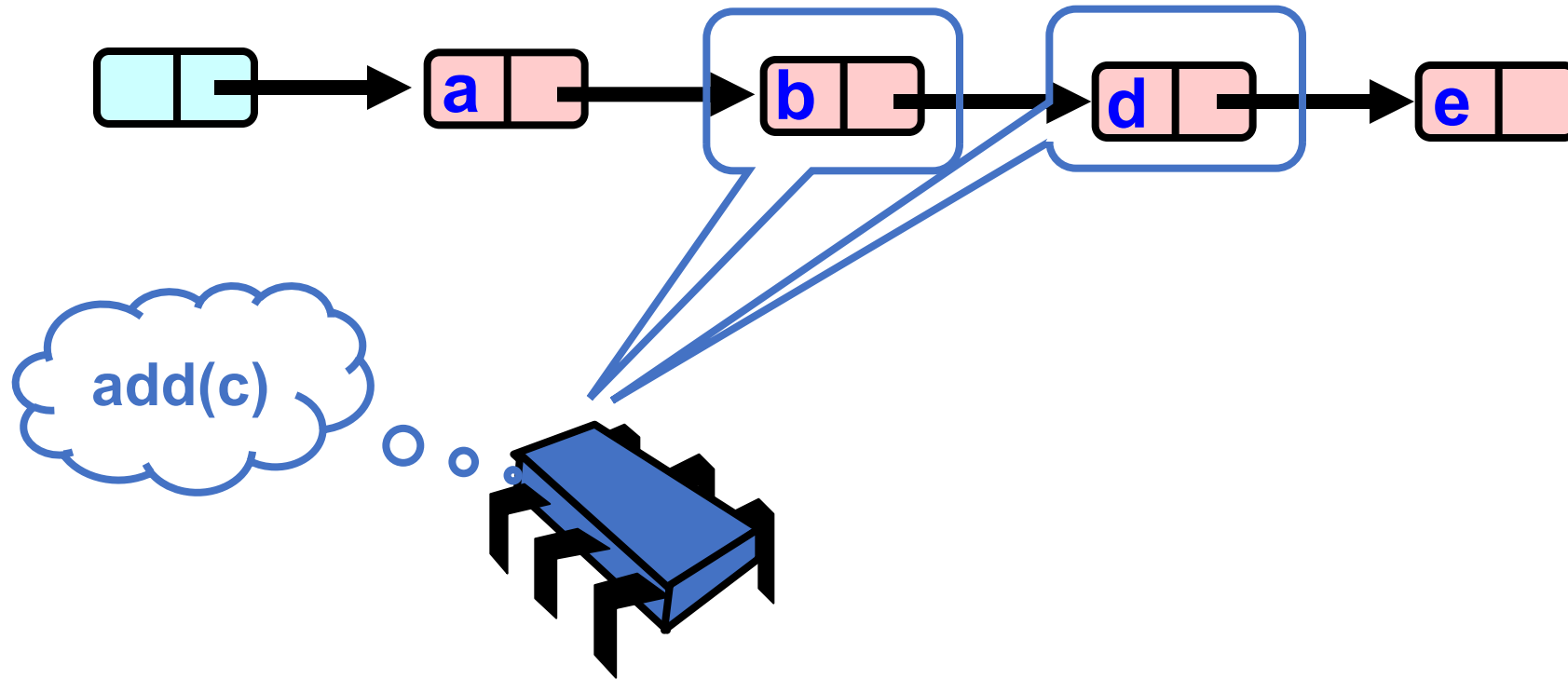




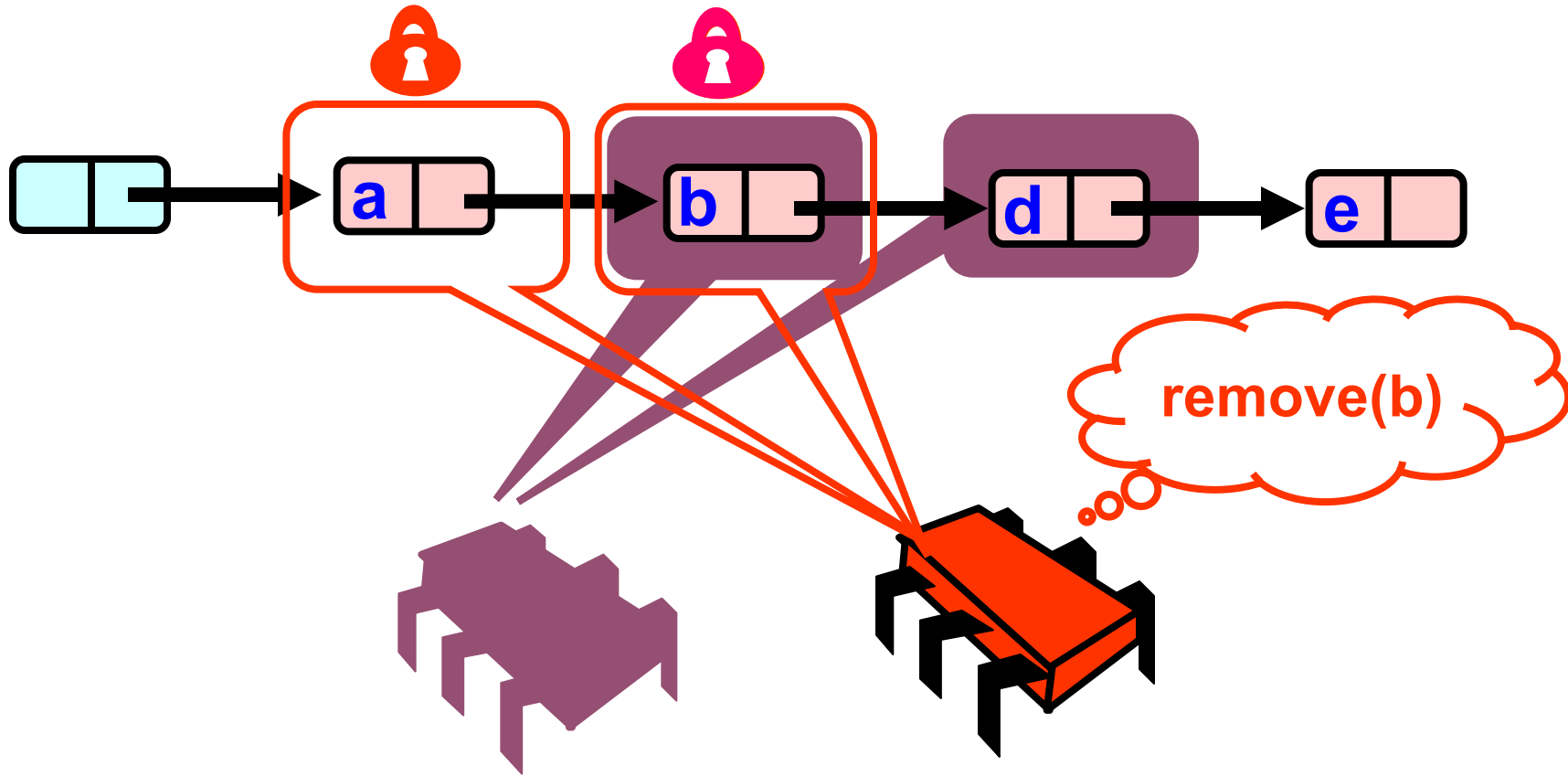
# What could go wrong?



# What could go wrong?



What could go wrong?



# Data conflict (Race)!

- Red thread has the lock on a node (so it can modify the node)
- Blue thread is traversing without locks
- What do we do?

# Race-freedom

- We can use atomic variables

# Race-freedom

```
class Node {  
  public:  
    Value v;  
    int key;  
    Node *next;  
}
```

# Race-freedom

```
class Node {  
    public:  
        Value v;  
        int key;  
        atomic<Node*> next;  
}
```

Create an atomic pointer type using C++ templates

# Race-freedom

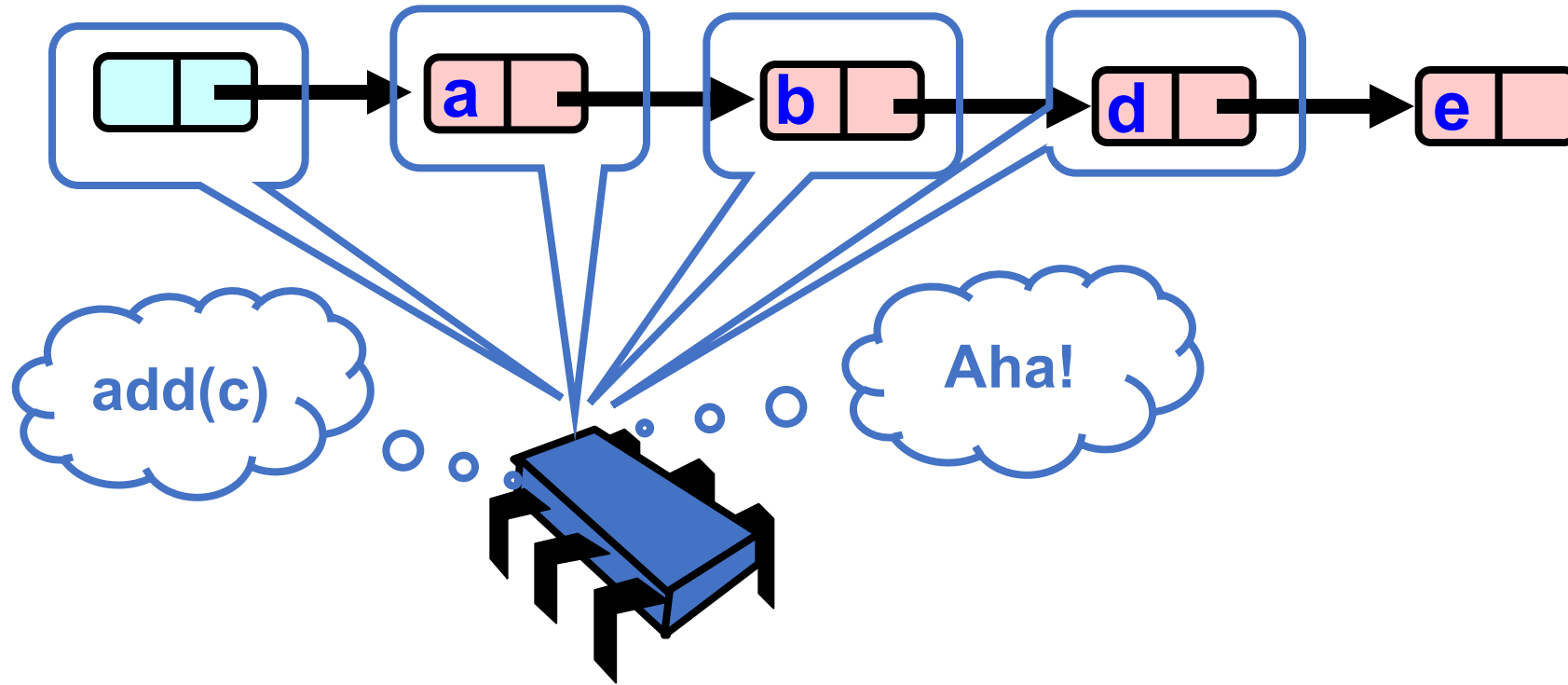
```
void traverse(node *n) {  
    while (n->next != NULL) {  
        n = n->next;  
    }  
}
```



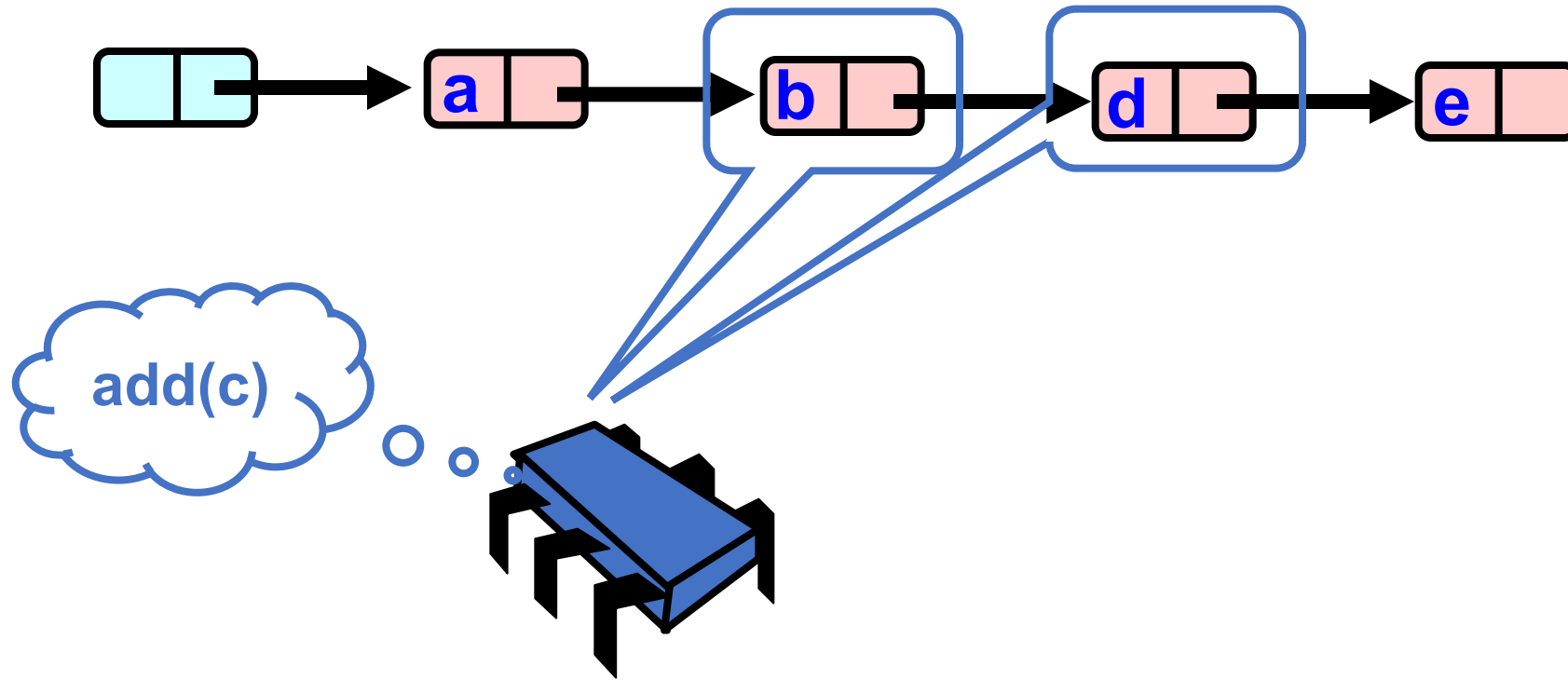
# Race-freedom

```
void traverse(node *n) {  
    while (n->next.load() != NULL) {  
        n = n->next.load();  
    }  
}
```

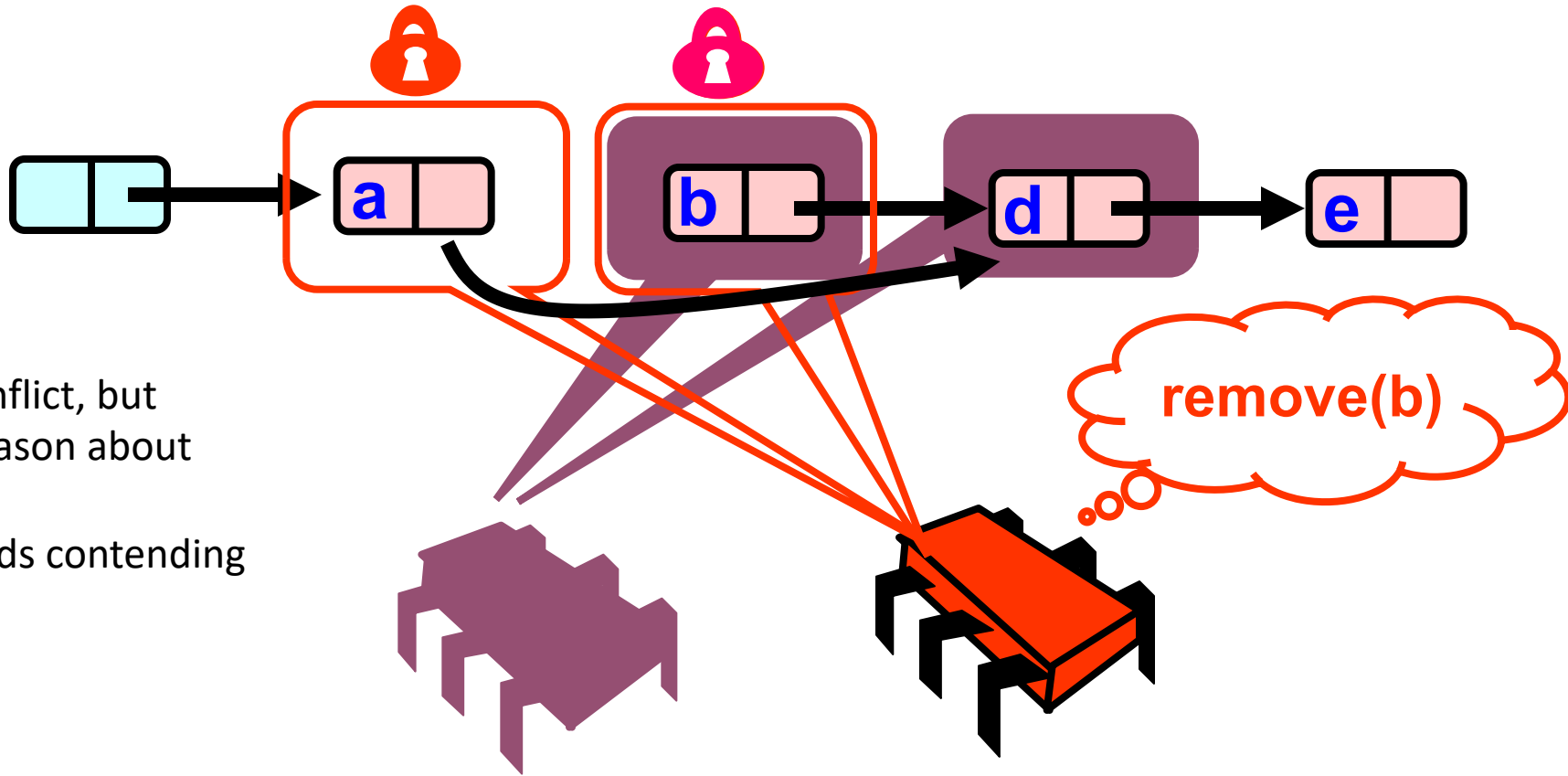
# What could go wrong?



# What could go wrong?

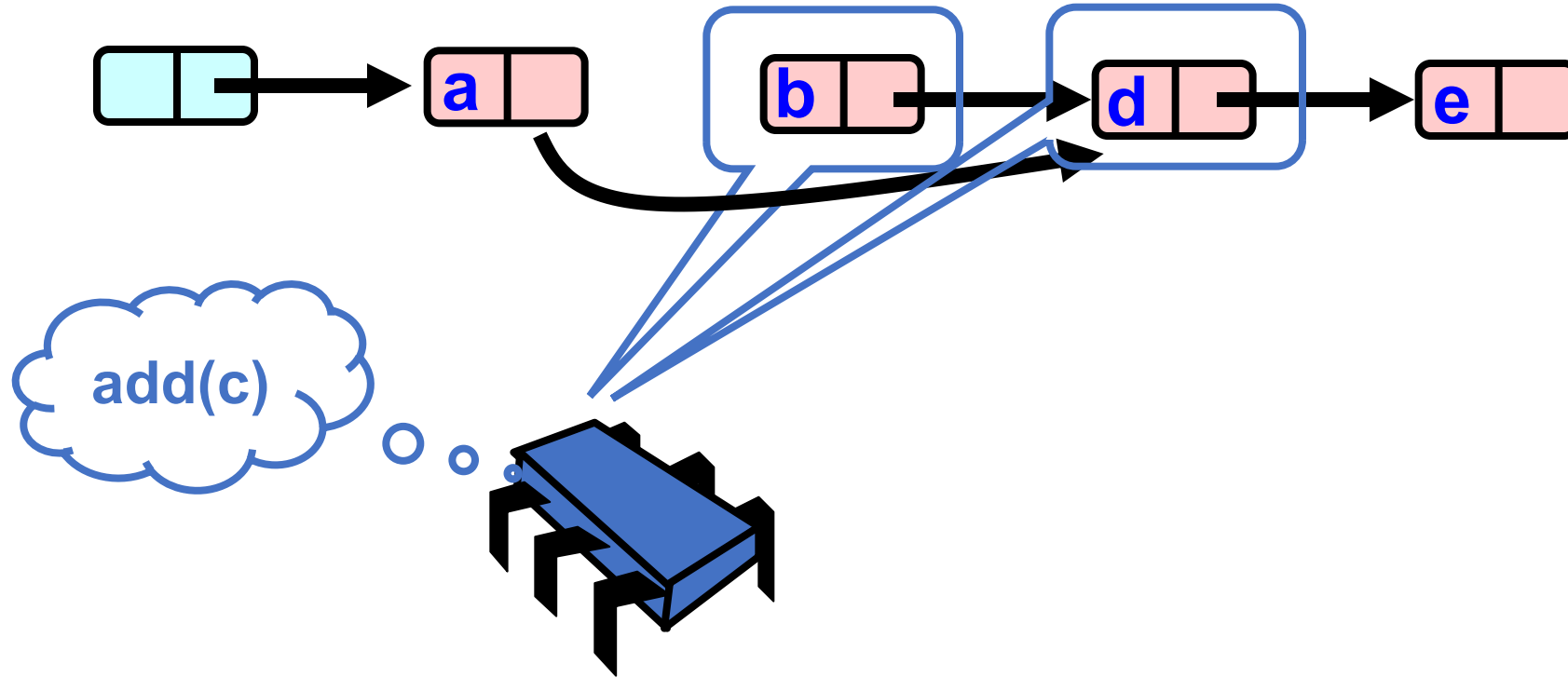


# What could go wrong?

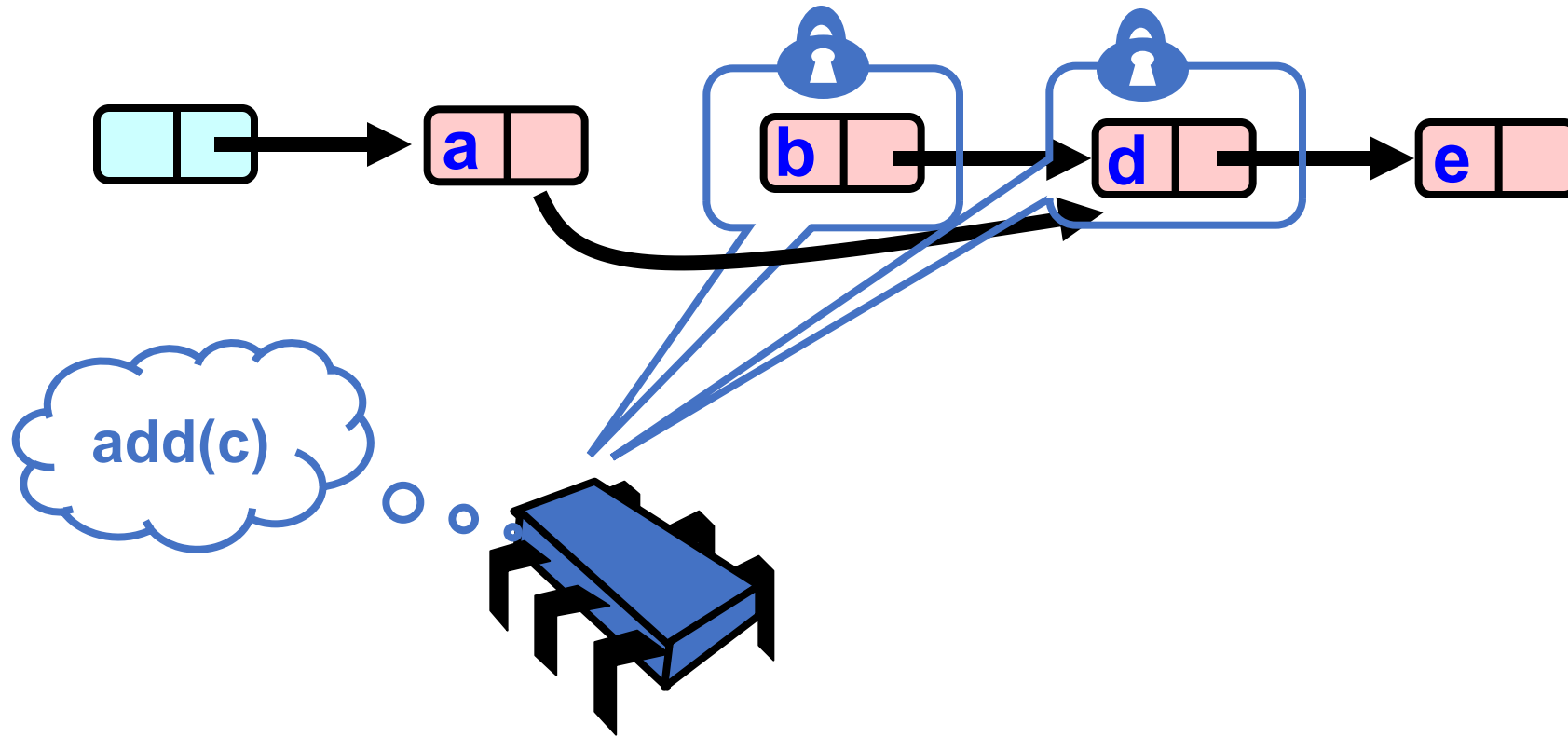


No more data conflict, but  
we do need to reason about  
interleavings.  
Concurrent threads contending  
for values.

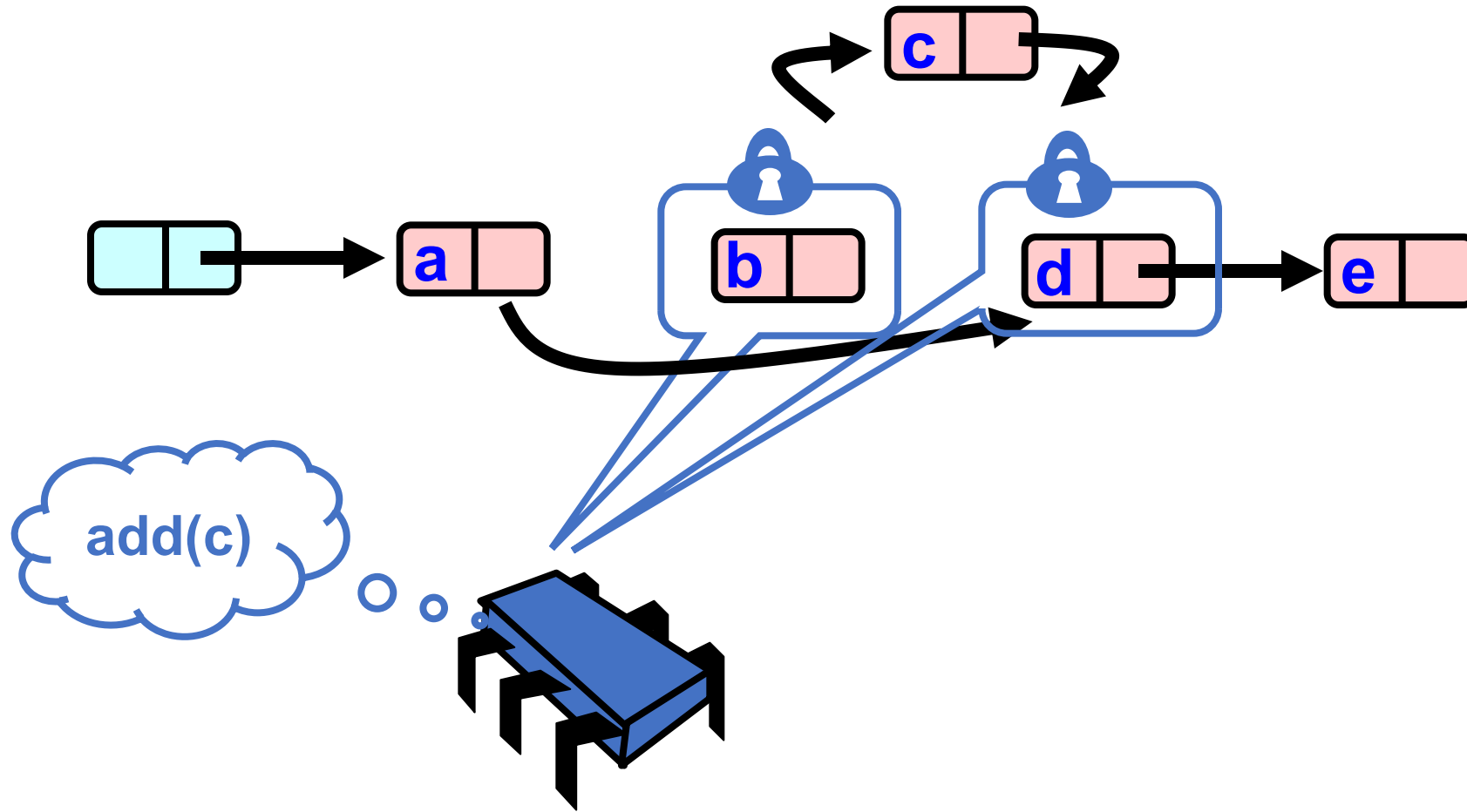
# What could go wrong?



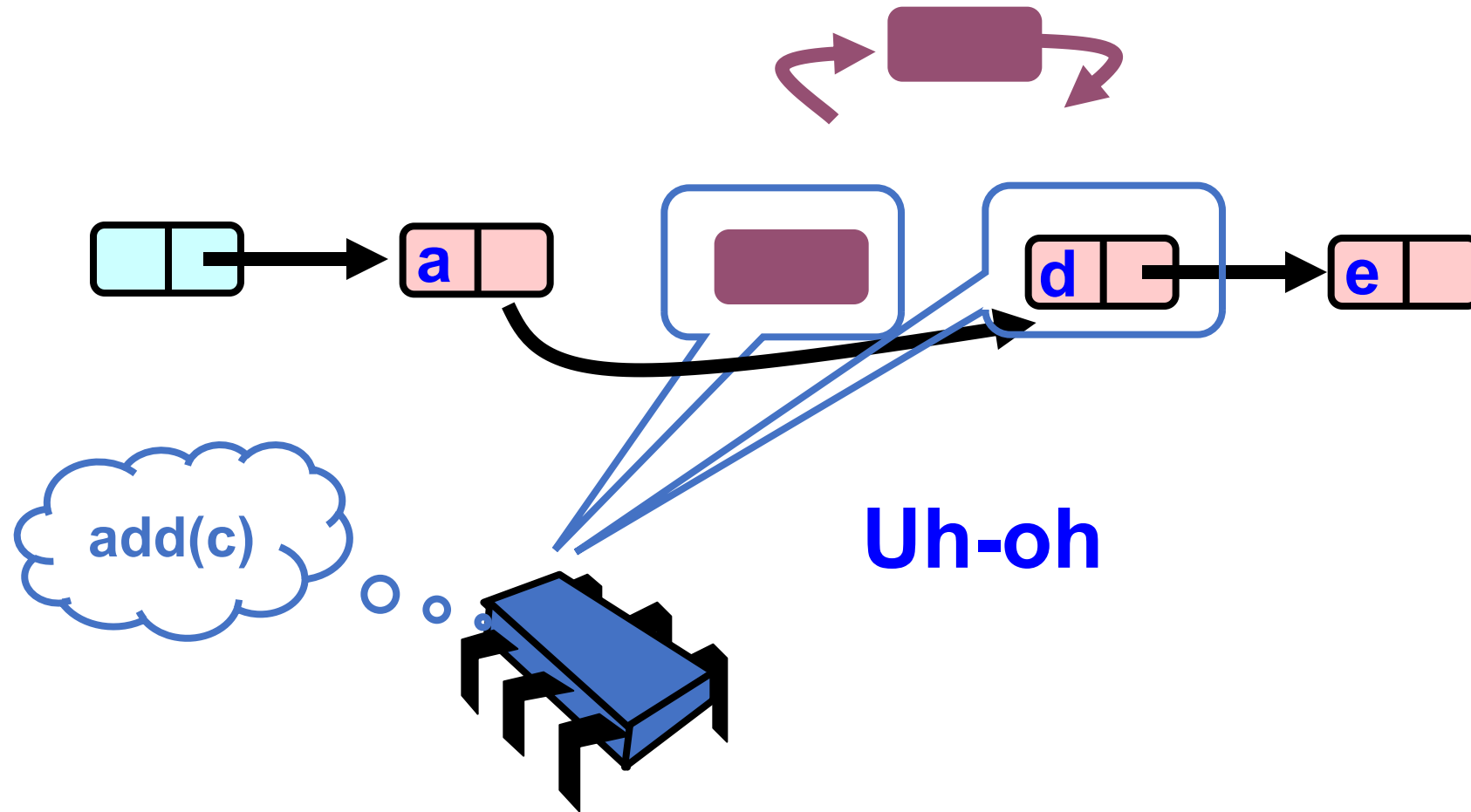
# What could go wrong?



What could go wrong?

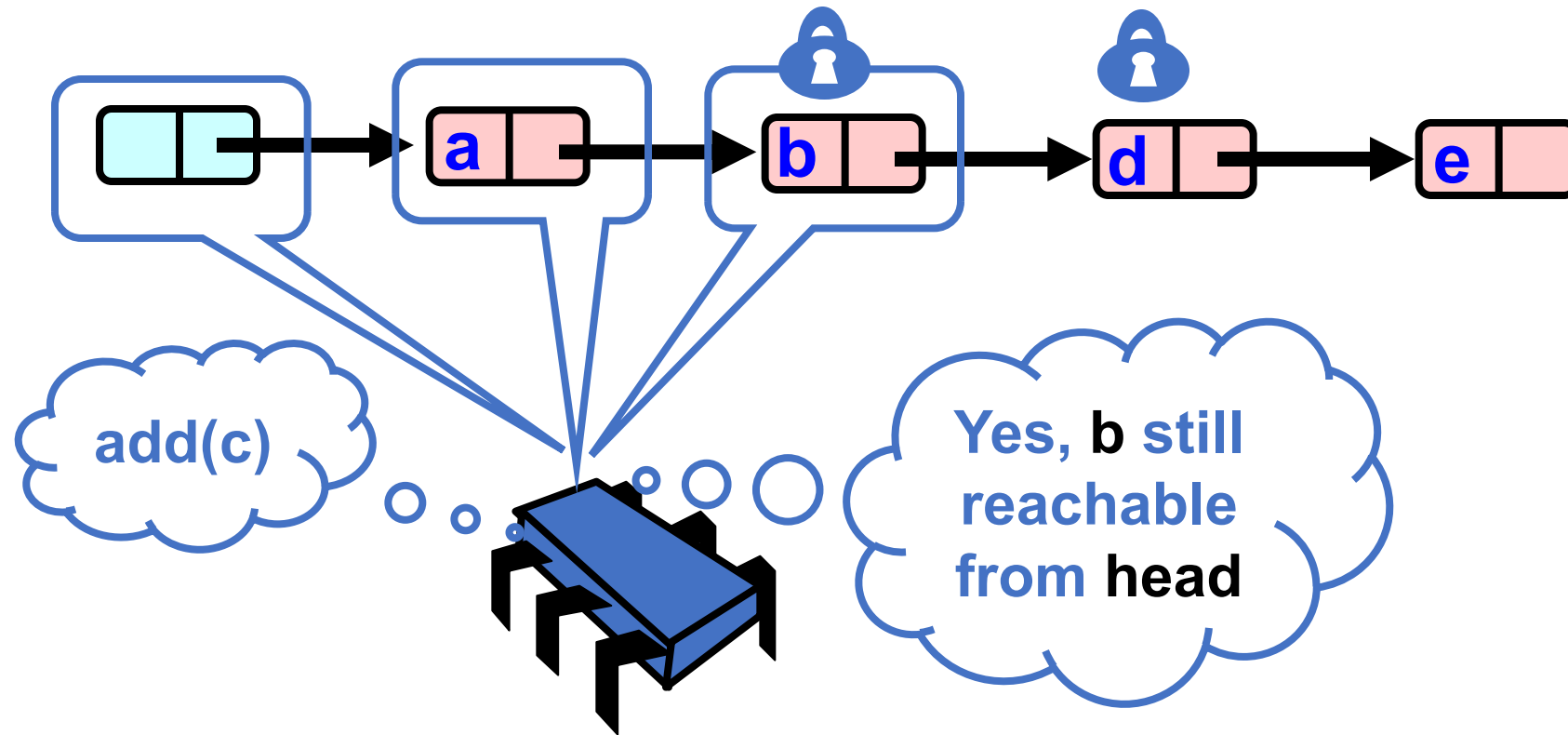


# What could go wrong?





# Validate – Part 1



# What happens if failure?

- Ideas?

# What happens if failure?

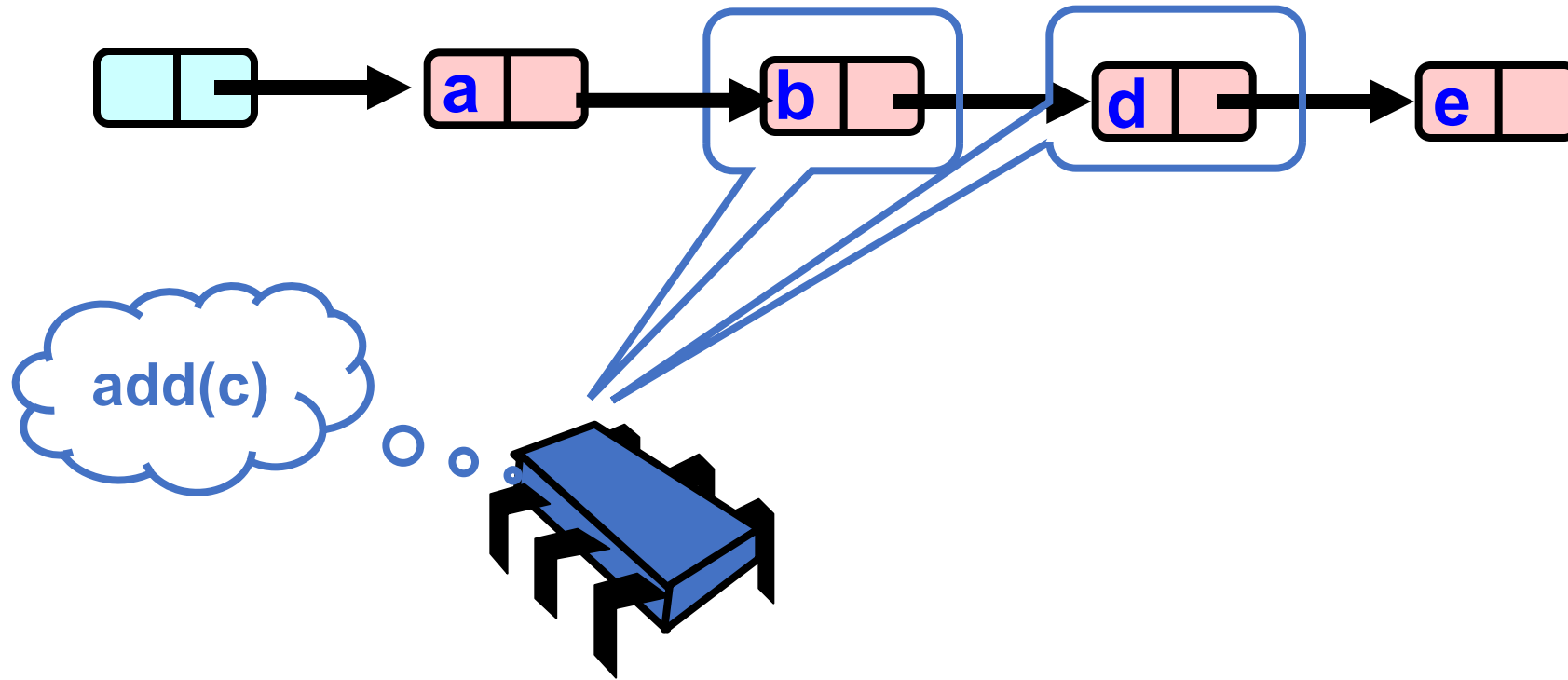
- Could try to recover? Back up a node?
  - Very tricky!
  - Just start over!

# What happens if failure?

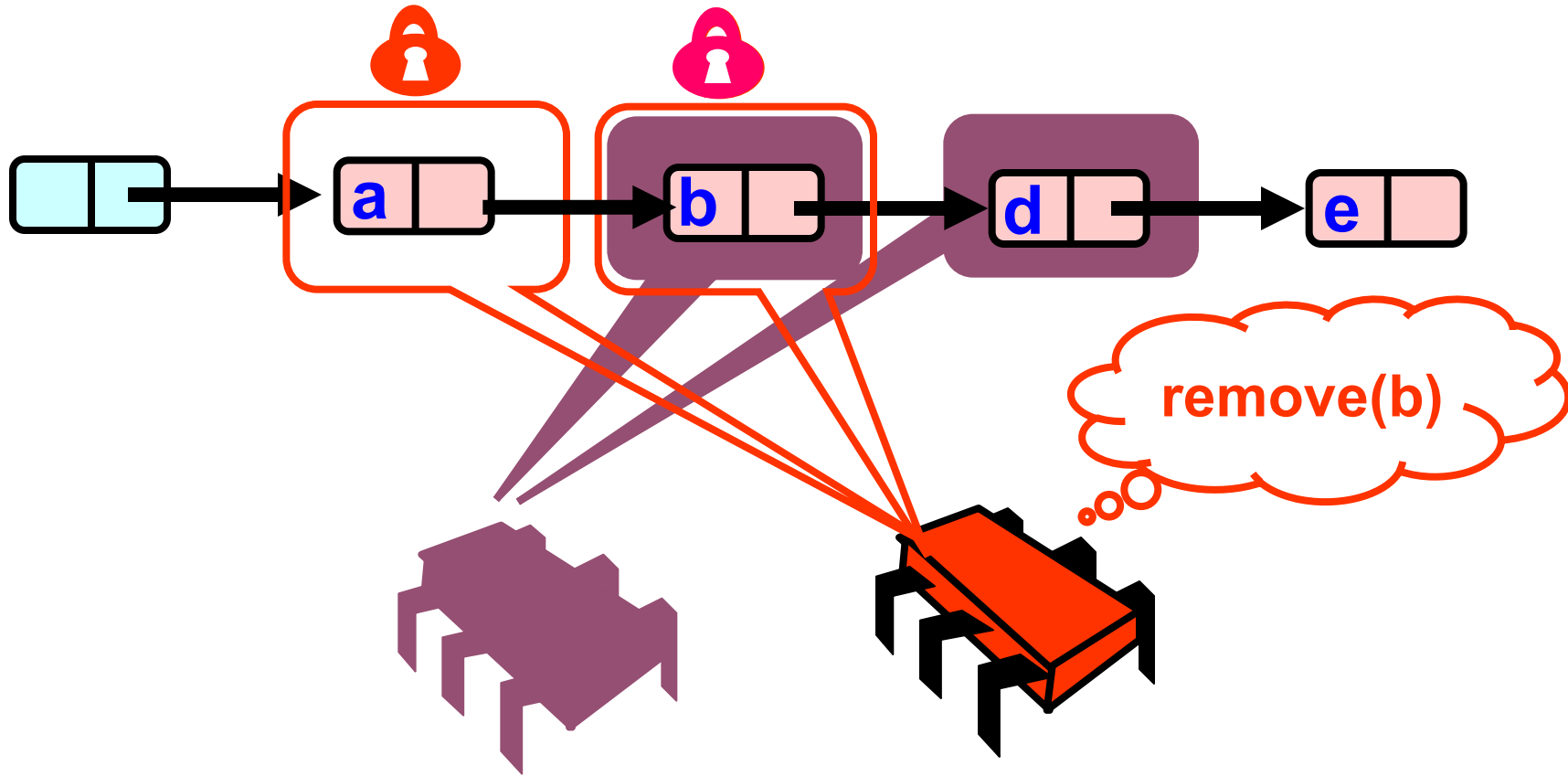
- Could try to recover? Back up a node?
  - Very tricky!
  - Just start over!
- Private method:
  - `try_remove`
  - remove loops on `try_remove` until it succeeds

What about deletion?

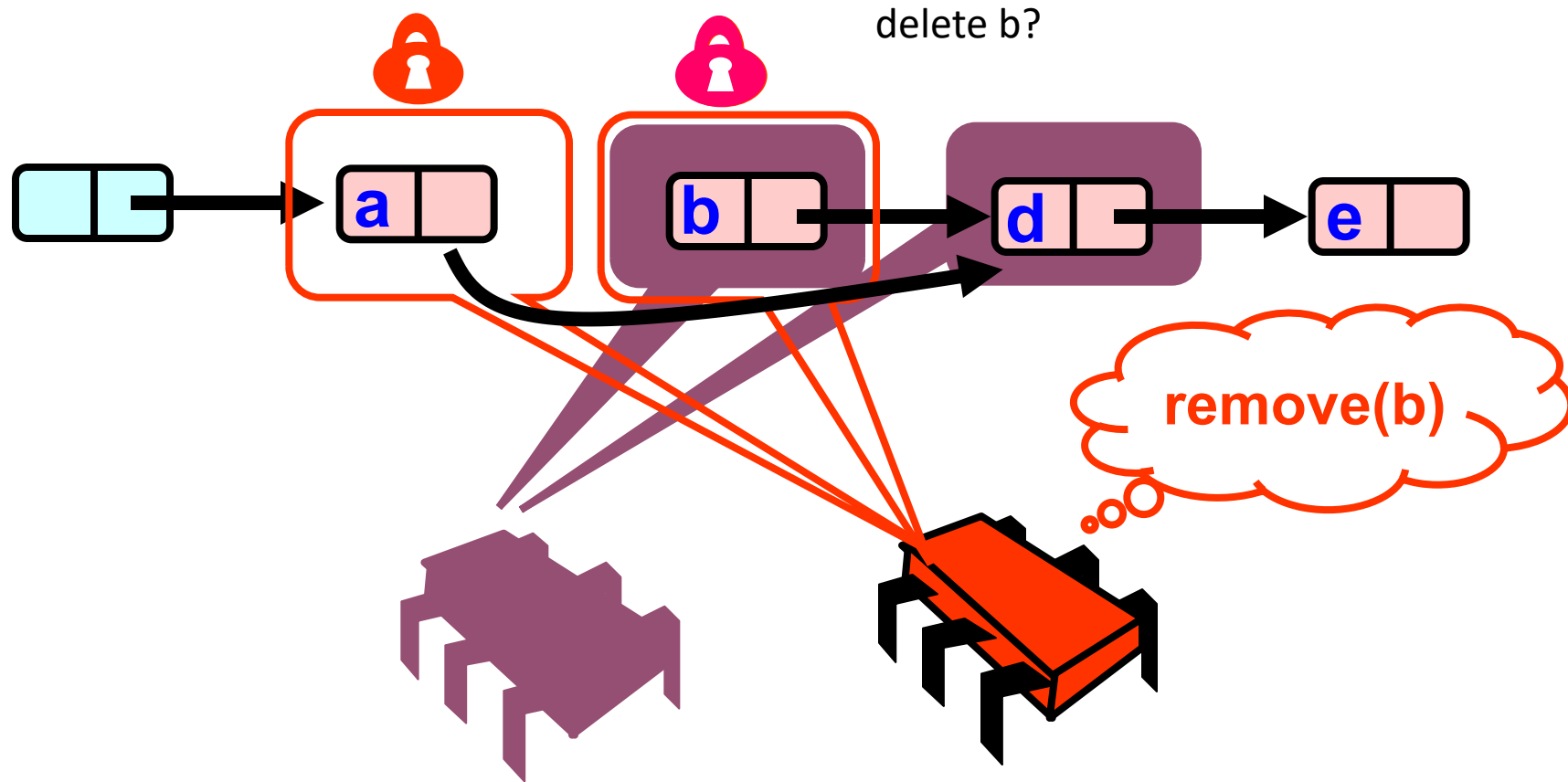
Can threads that remove a node delete it?



Can threads that remove a node delete it?

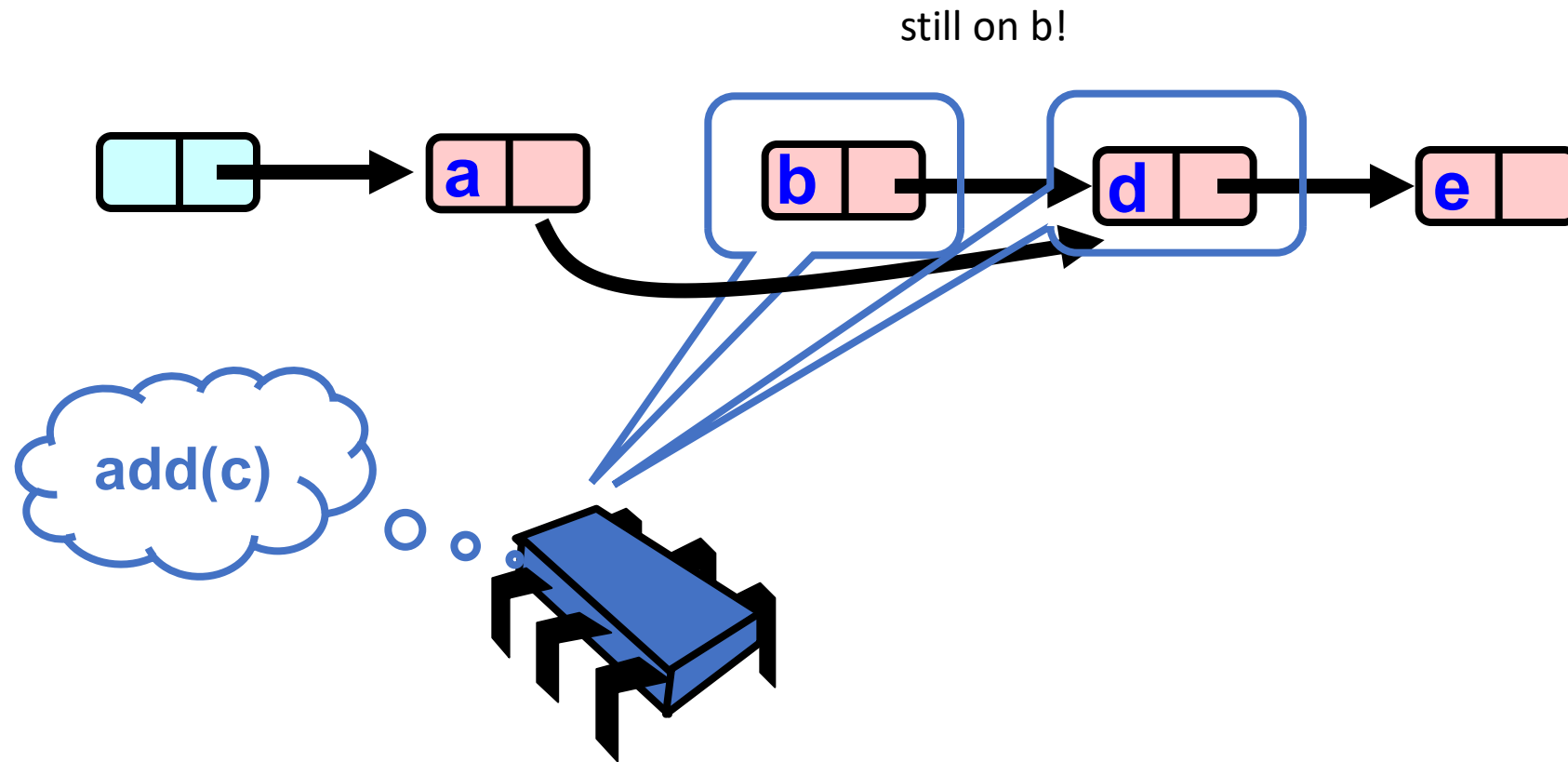


Can threads that remove a node delete it?

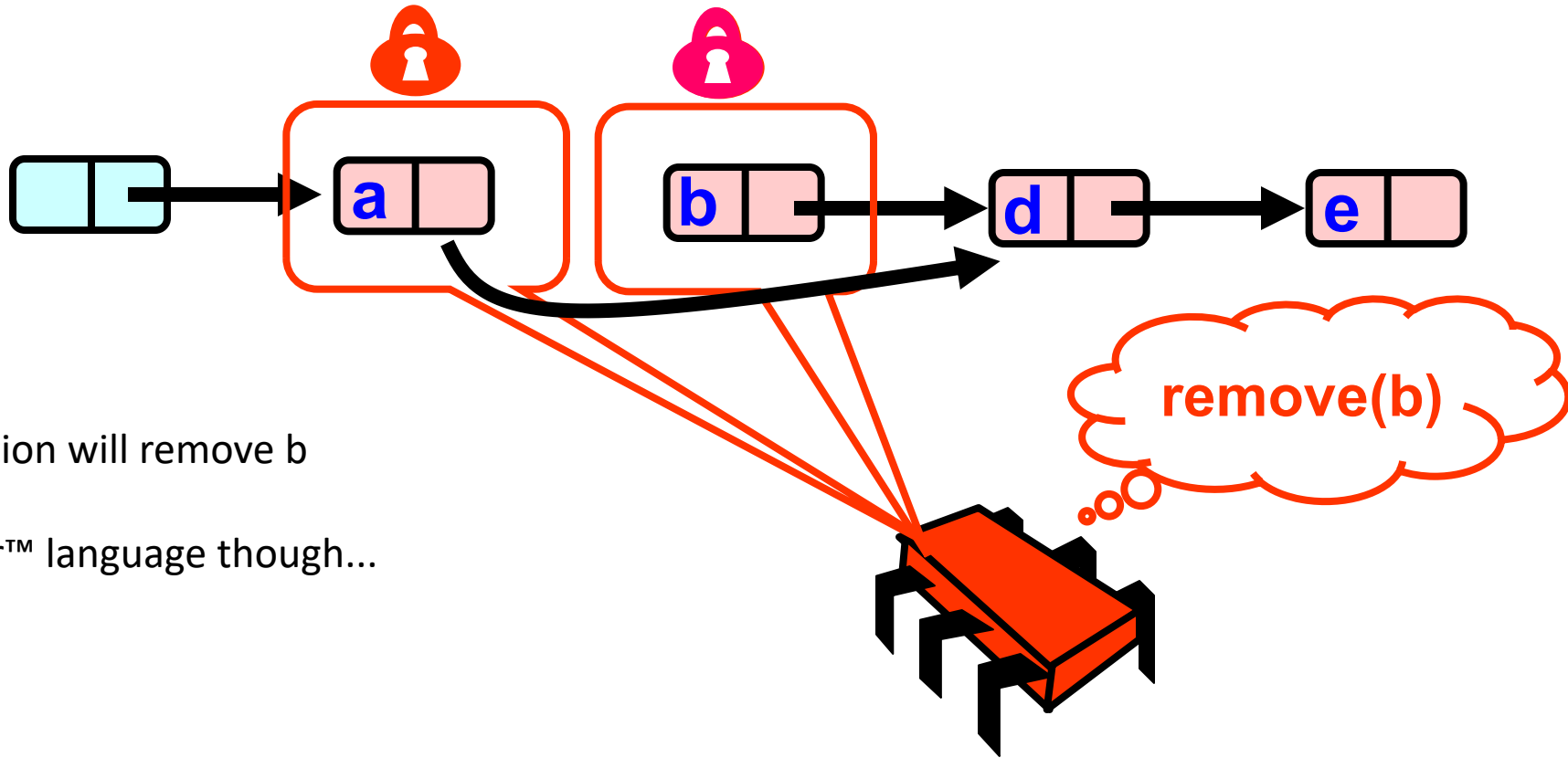




Can threads that remove a node delete it?



# Our own garbage collector



Java's garbage collection will remove b

We are using a better™ language though...

maintain a list to delete:

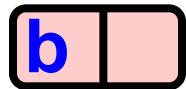
# Our own garbage collector



Java's garbage collection will remove b

We are using a better™ language though...

maintain a list to delete:



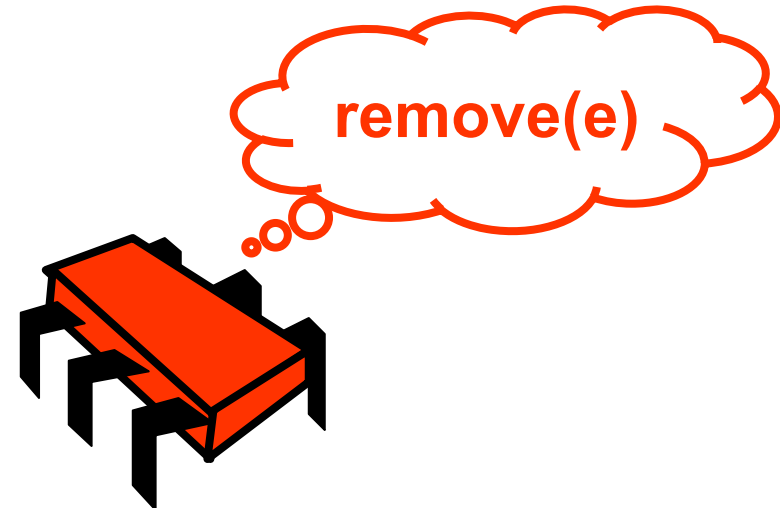
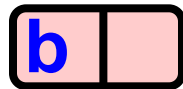
# Our own garbage collector



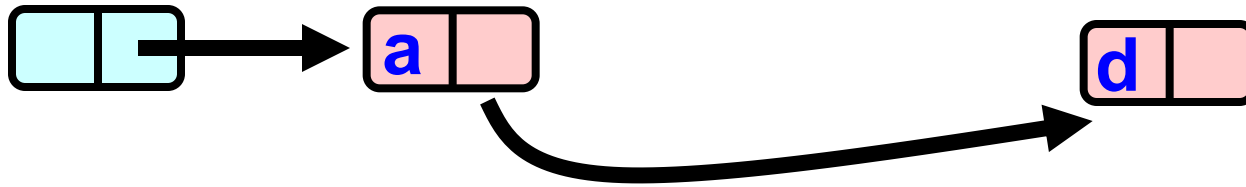
Java's garbage collection will remove b

We are using a better™ language though...

maintain a list to delete:  
can be thread-local



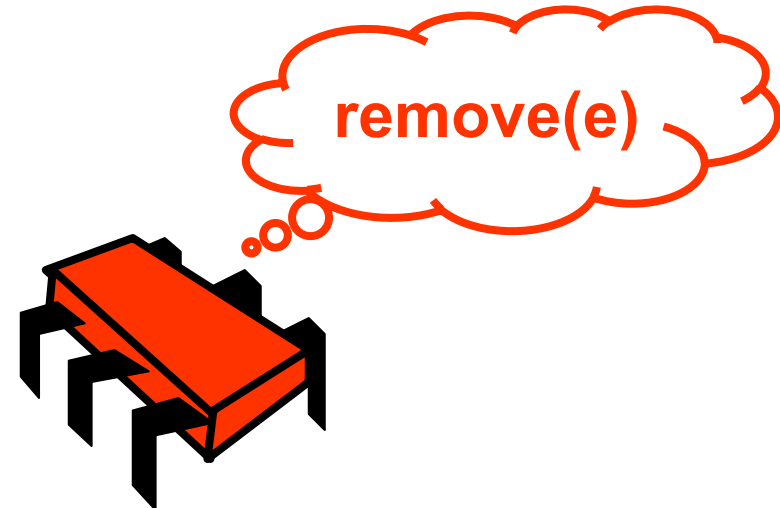
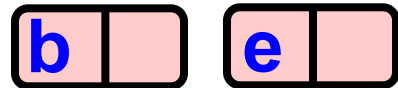
# Our own garbage collector



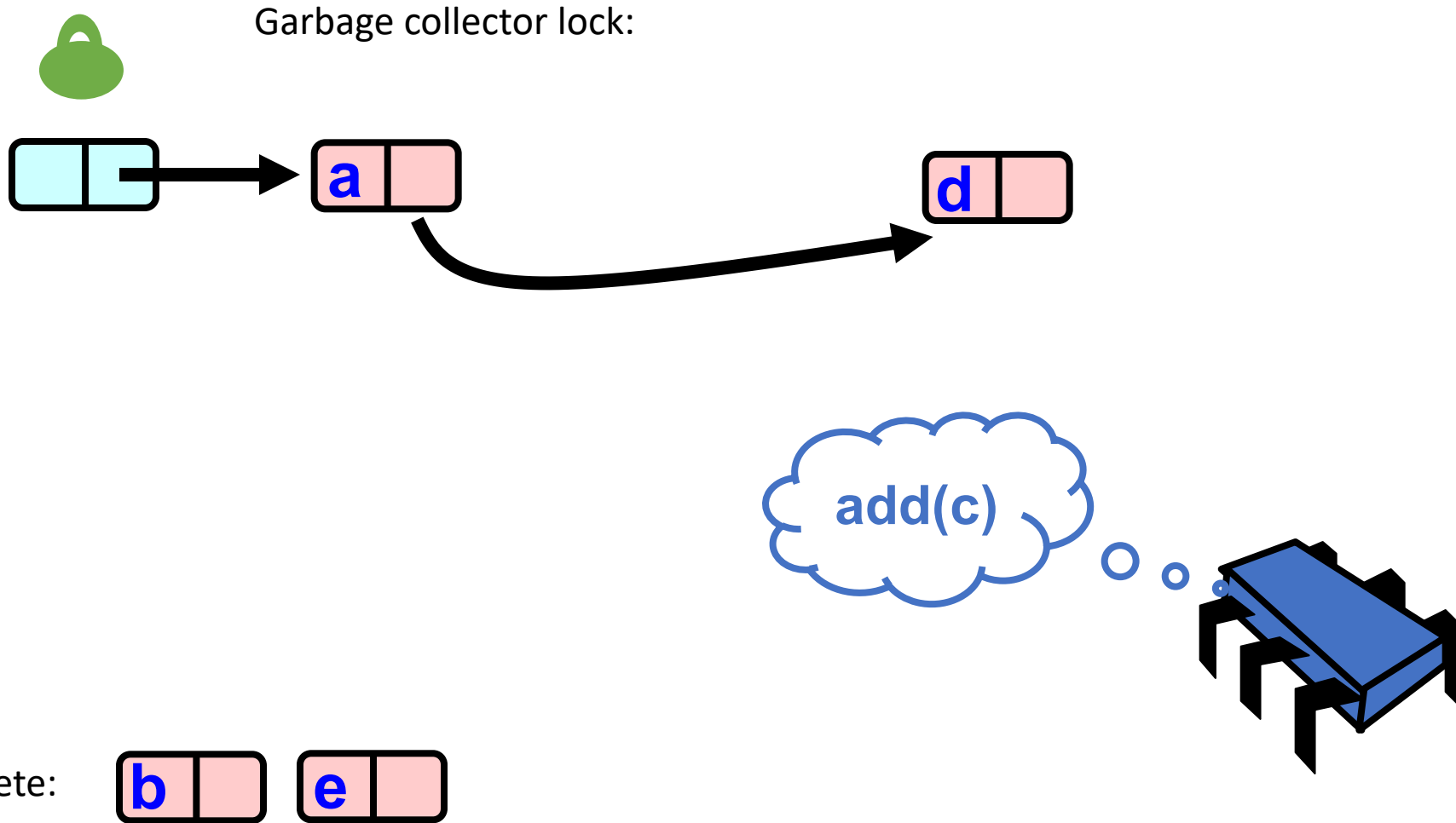
Java's garbage collection will remove b

We are using a better™ language though...

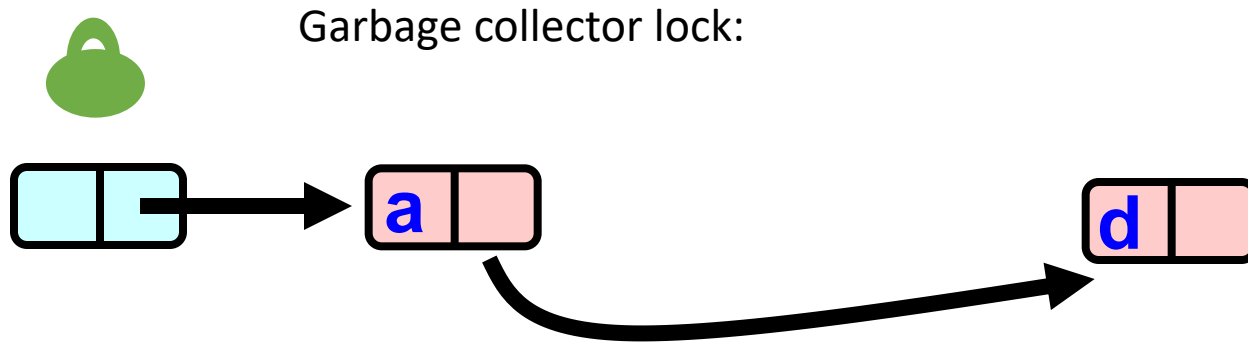
maintain a list to delete:



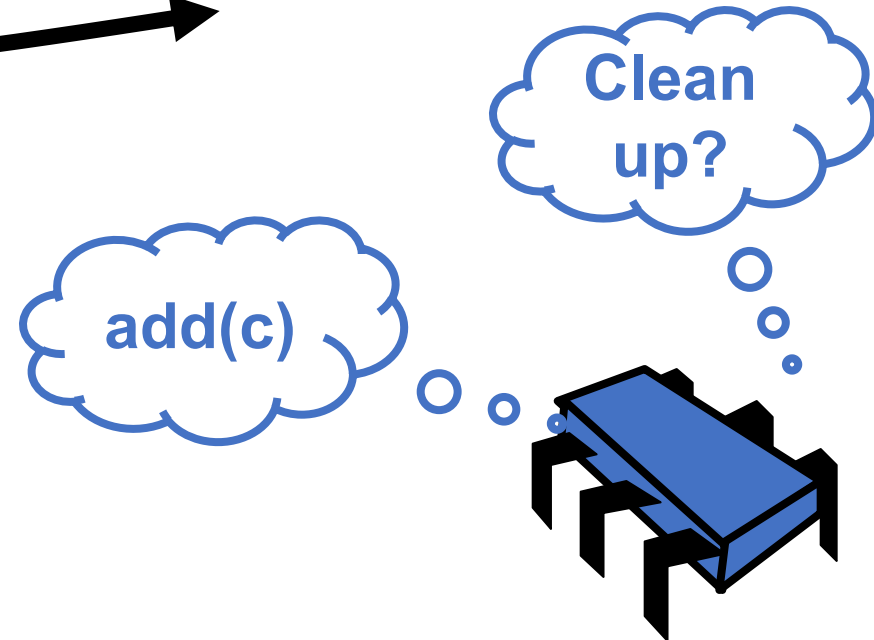
# Our own garbage collector



# Our own garbage collector



Similar to a reader/writer lock:  
Allows an arbitrary number of threads that operate on the list  
Only 1 garbage collector thread erases the list of nodes



maintain a list to delete:

The diagram shows a list of nodes to be deleted. It consists of two nodes, each represented as a rectangle divided into two sections. The first node has the letter 'b' in blue in the left section and a pink section on the right. The second node has the letter 'e' in blue in the left section and a pink section on the right.

# Garbage collector lock

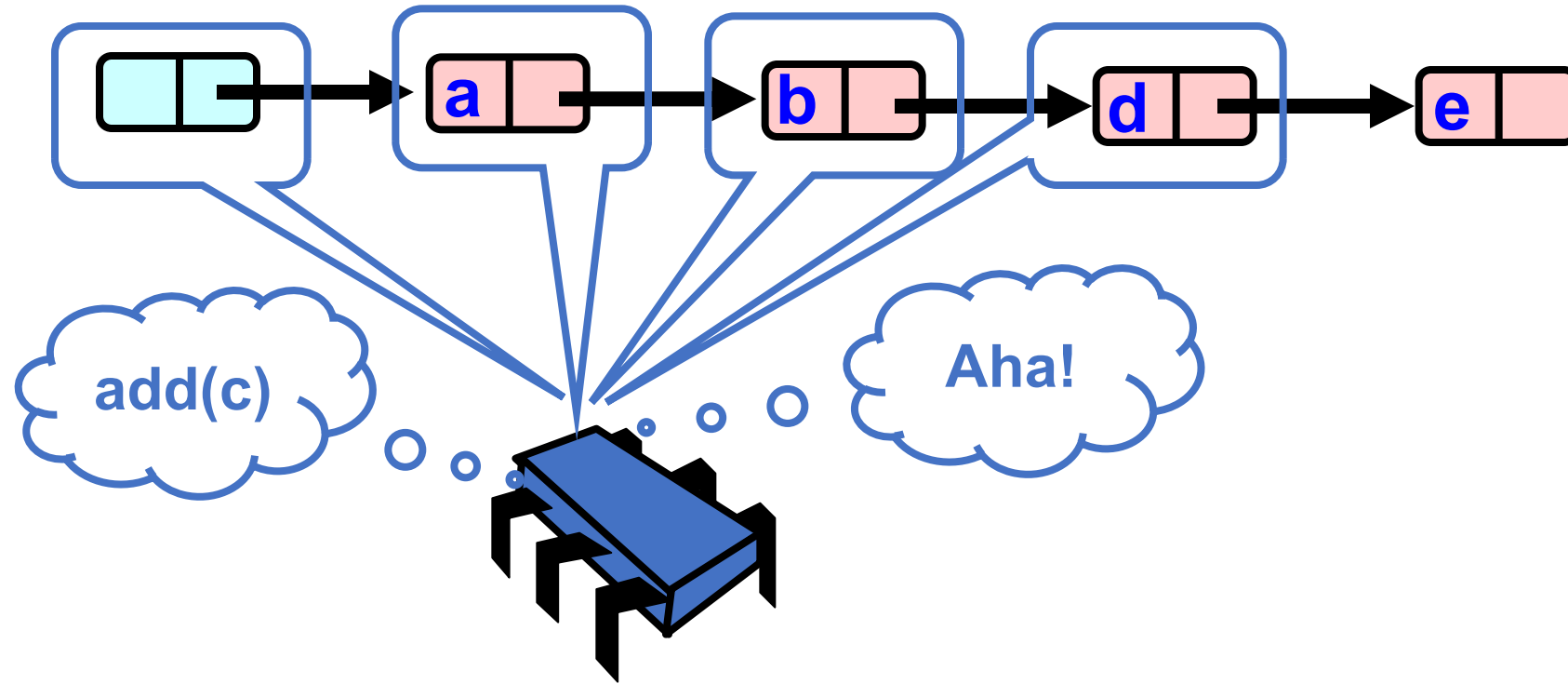
- Many strategies!
  - A big research area ~10 years ago
- **Strat 1:** Threads always try once to take the garbage collector lock:
  - if failed, no worries, the next operation will get a chance
  - can starve garbage collection
- **Strat 2:** Wait until size grows to a threshold:
  - Wait on the lock
  - Can cause performance spikes



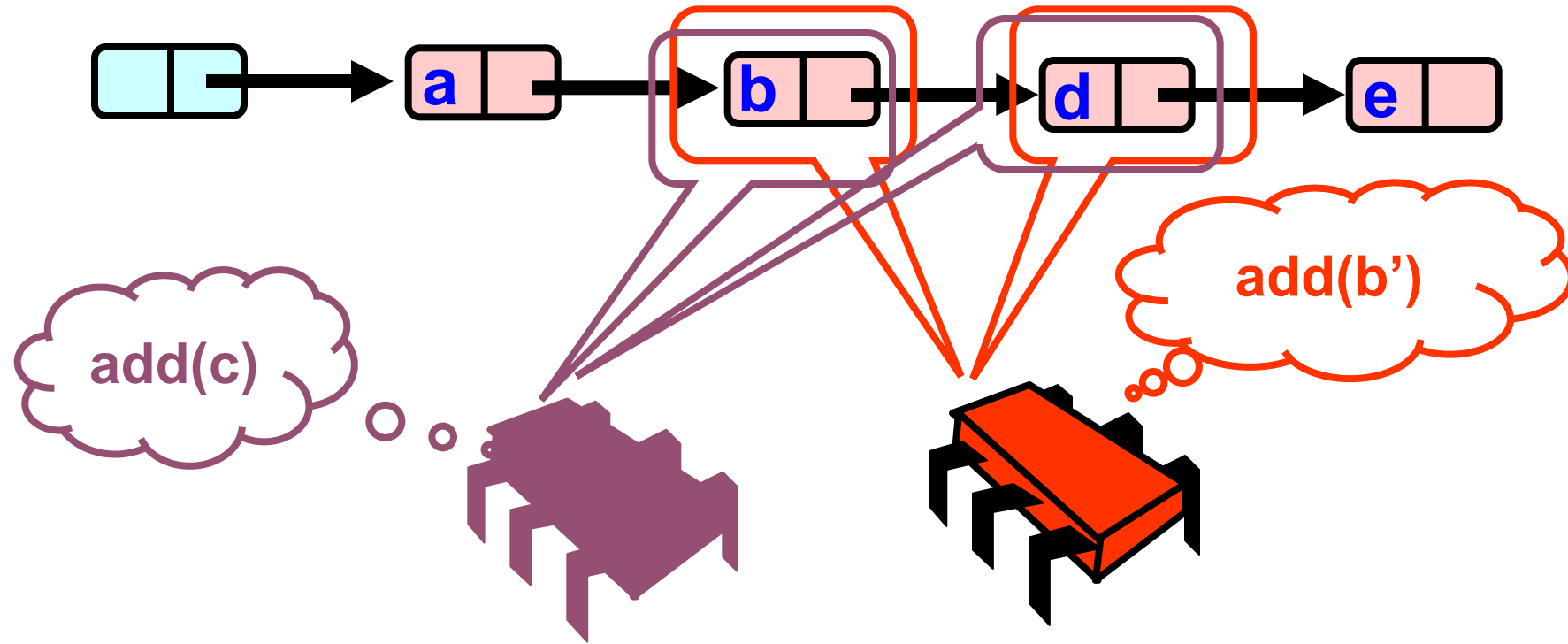
# Back to the linked list

What if 2 threads try to add a node in the same position?

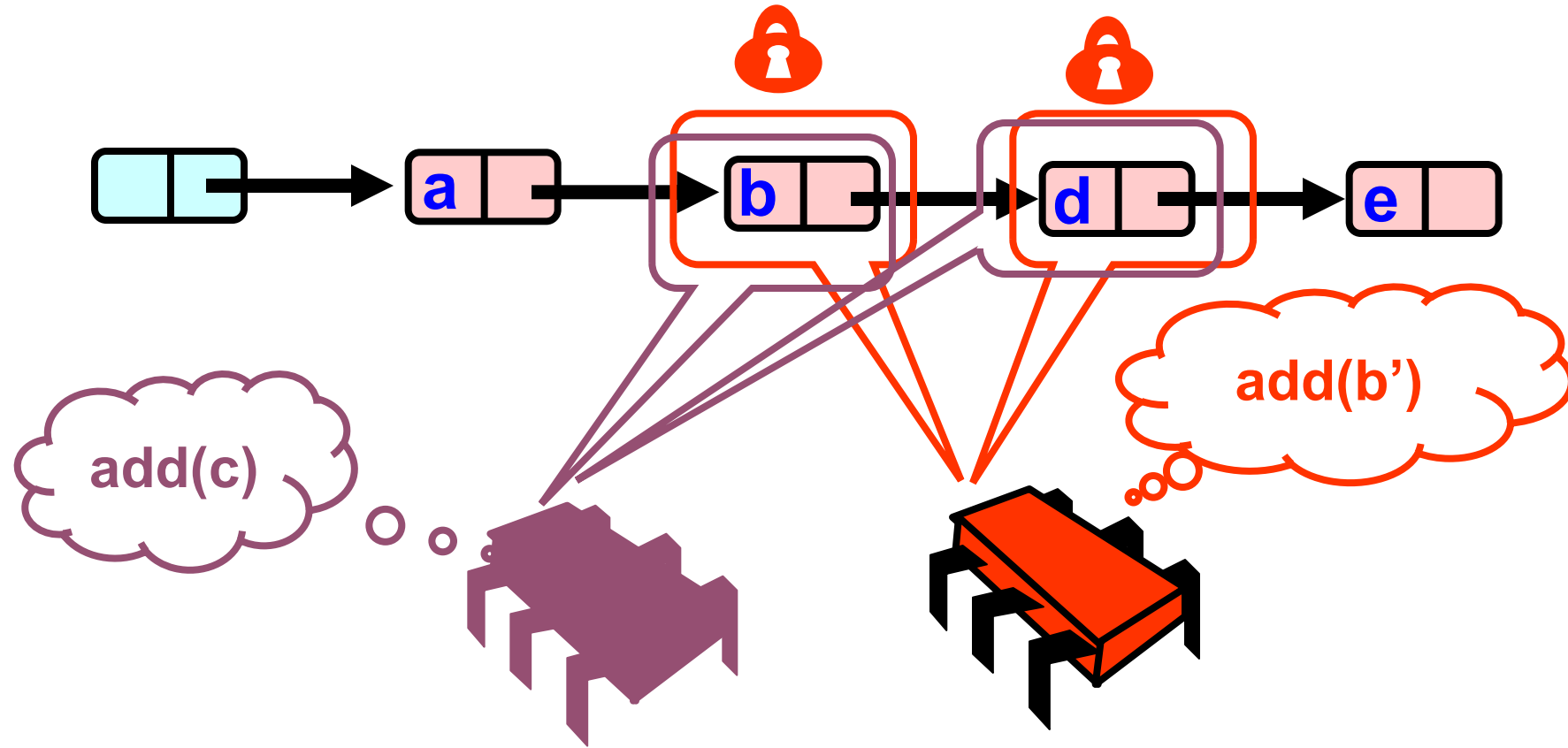
# What Else Could Go Wrong?



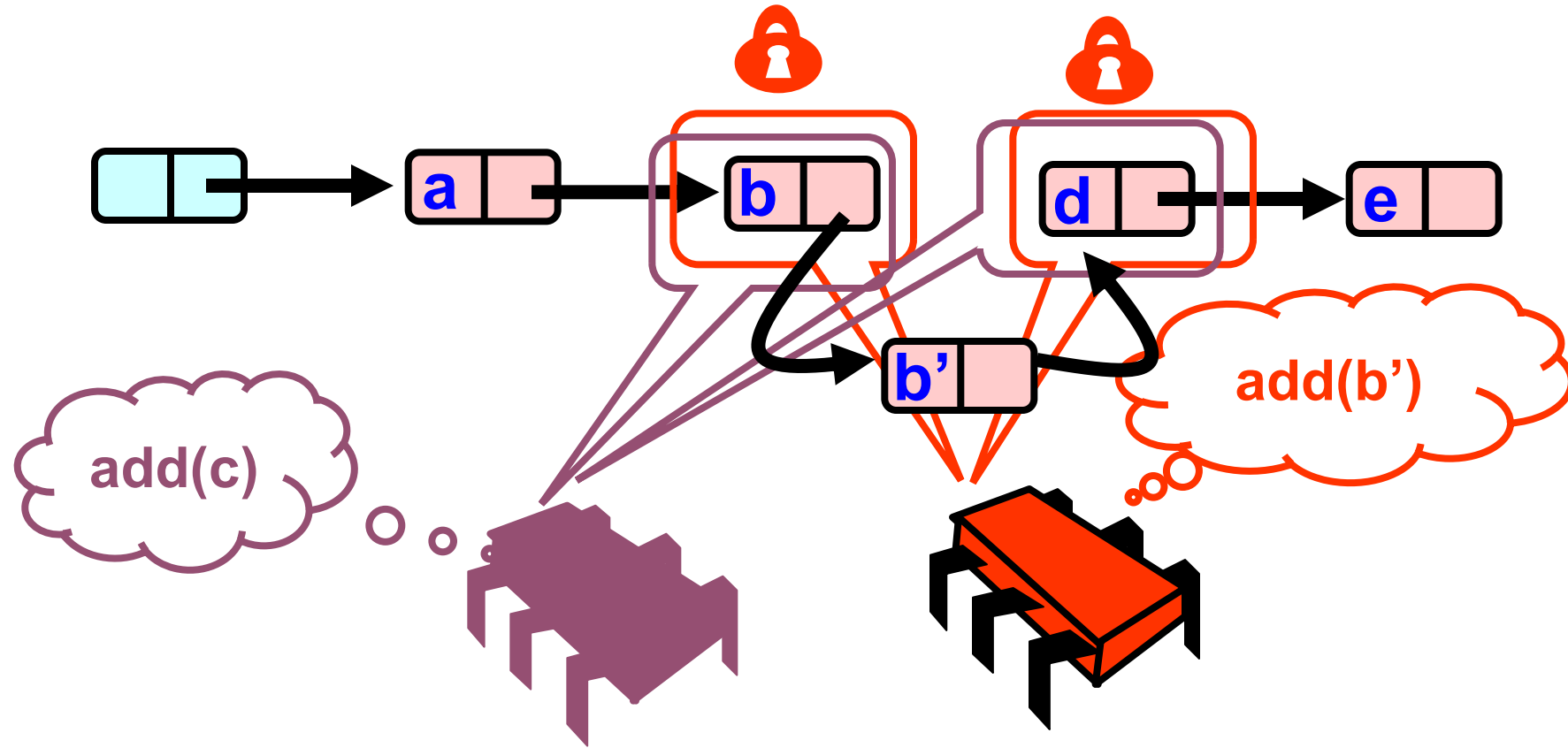
# What Else Could Go Wrong?



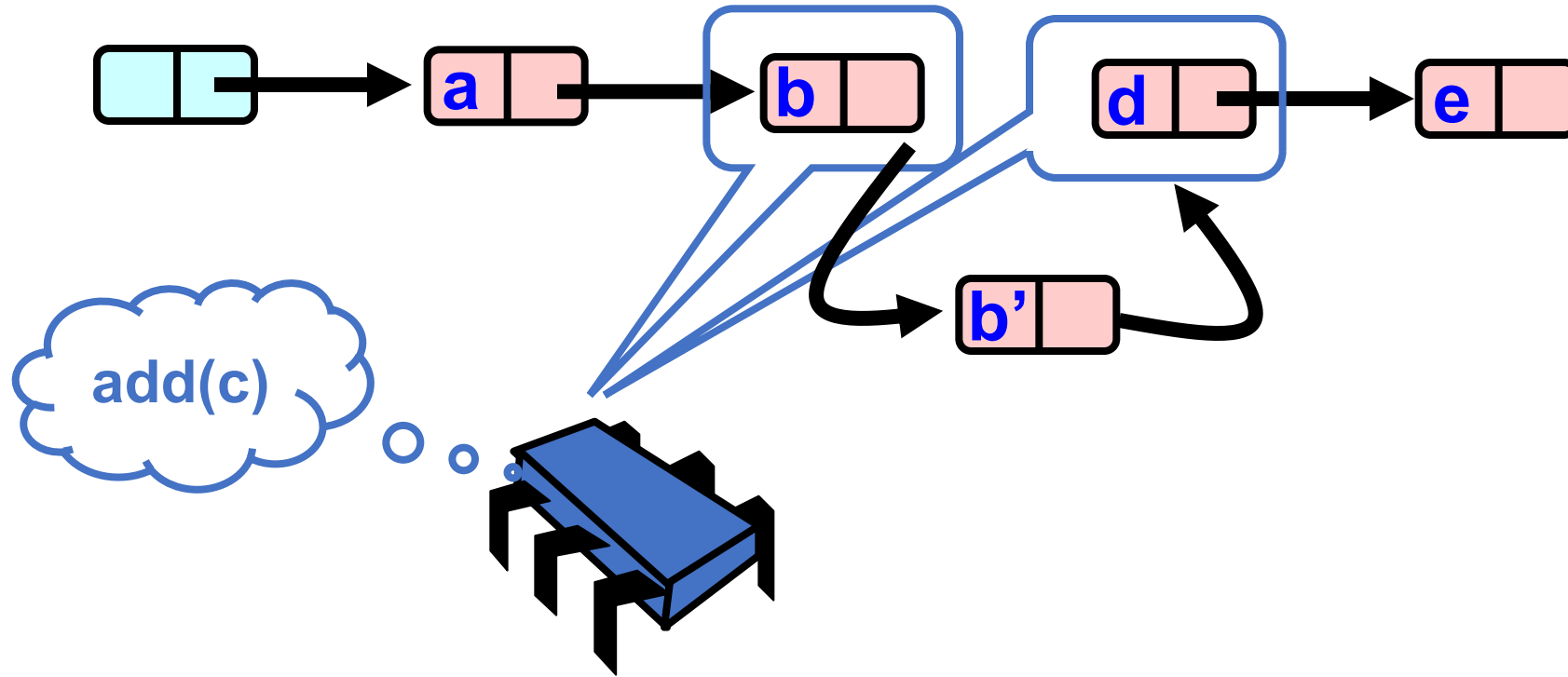
# What Else Could Go Wrong?



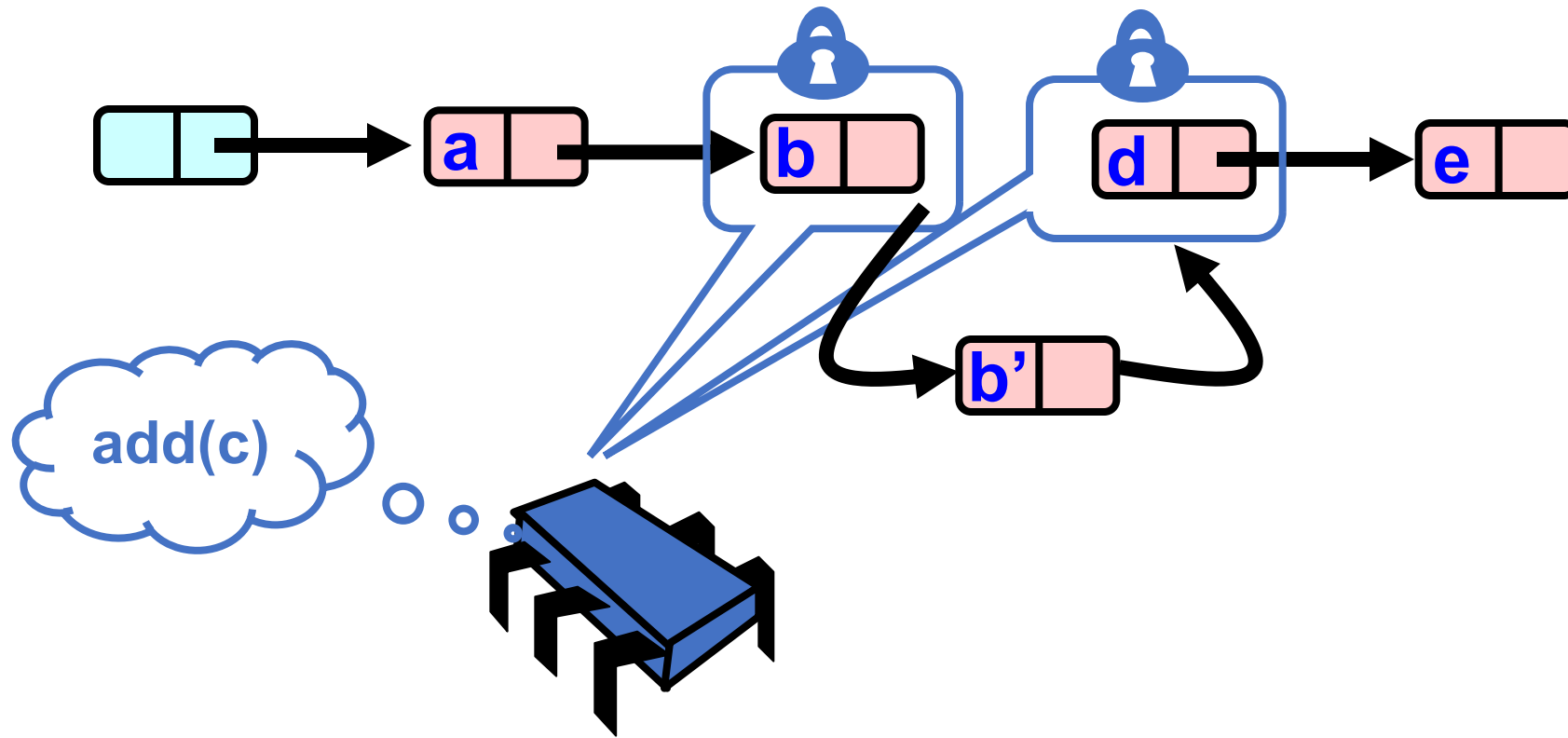
# What Else Could Go Wrong?



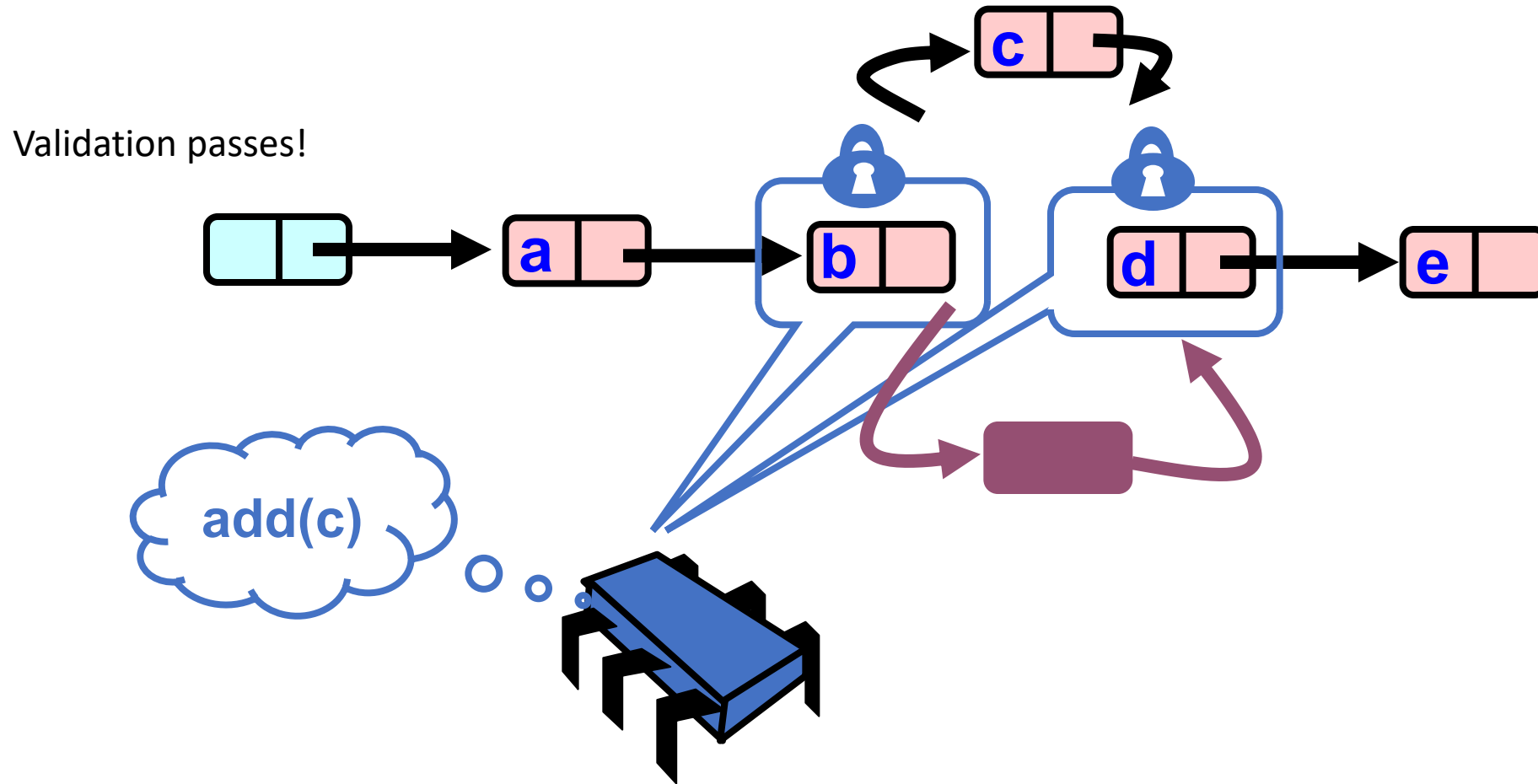
# What Else Could Go Wrong?



# What Else Could Go Wrong?

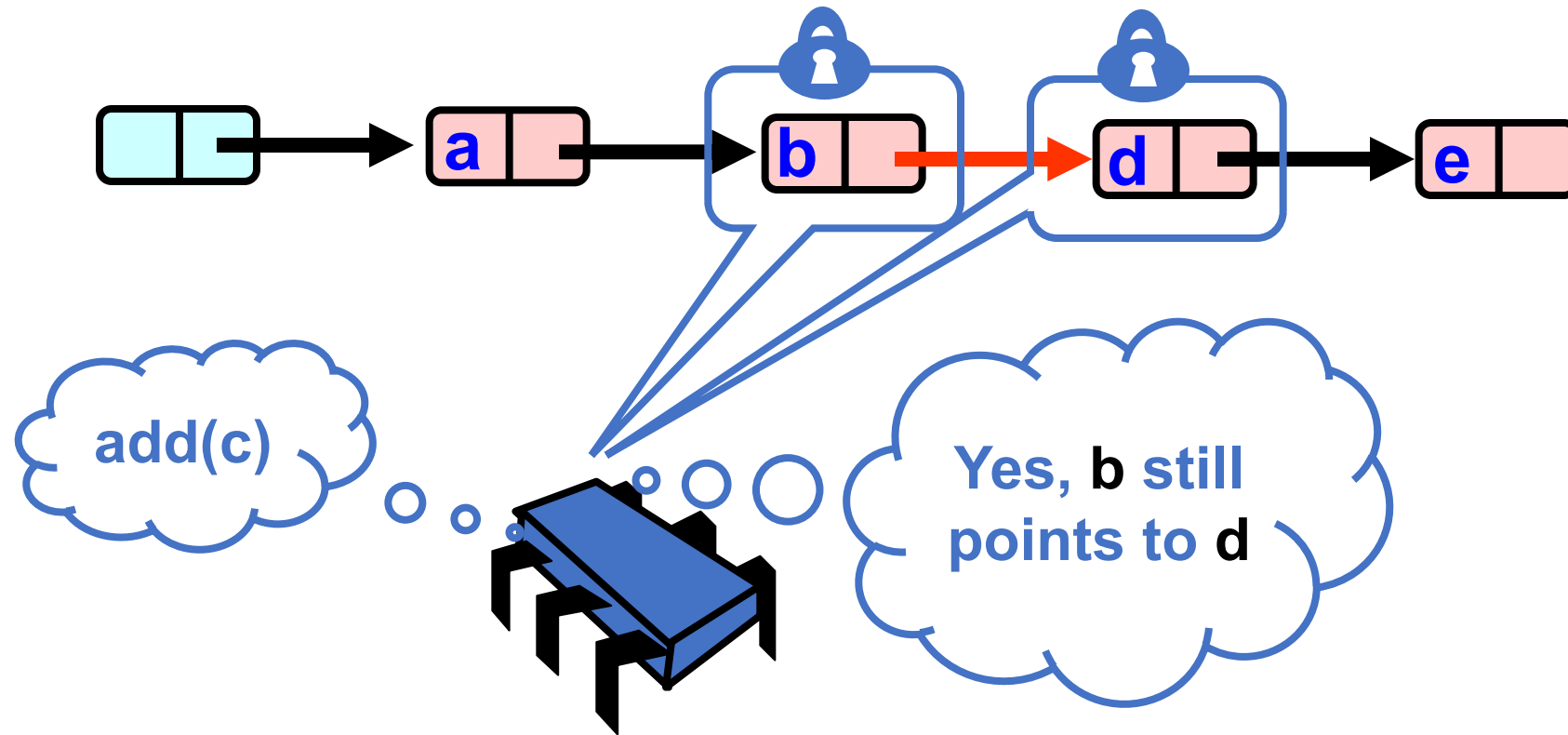


# What Else Could Go Wrong?

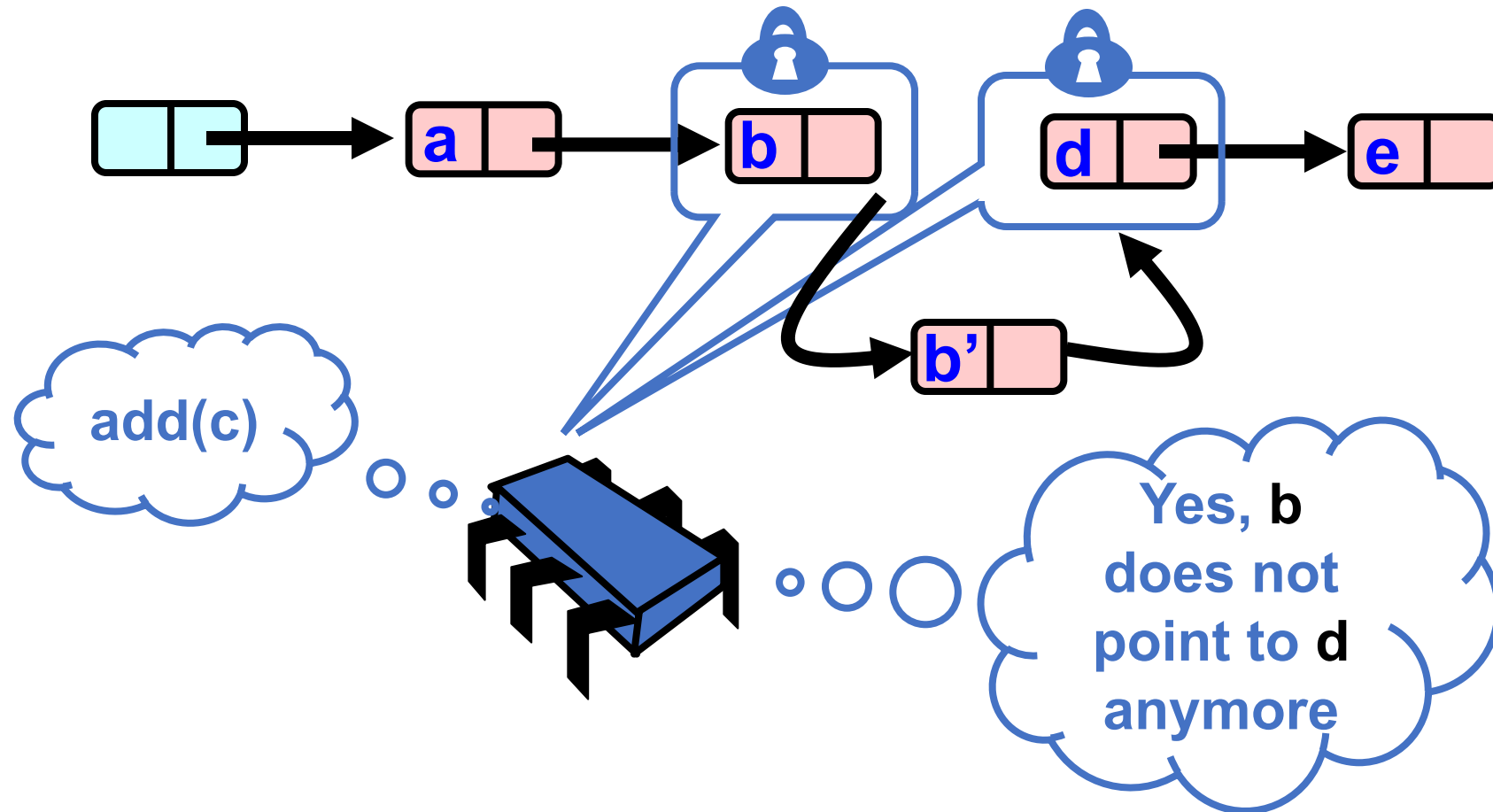




## Validate Part 2 (while holding locks)



# What Else Could Go Wrong?



# Summary

- We traverse without lock
  - Traversal may access nodes that are locked
  - Its okay because we have atomic pointers!
- We avoid traversing the old structure
  - We validate after we obtain locks.
    - Our node is still reachable (it was not deleted)
    - Our insertion point is still valid (no thread has inserted in the meantime)
- To free nodes
  - We put them in a list to be freed later.

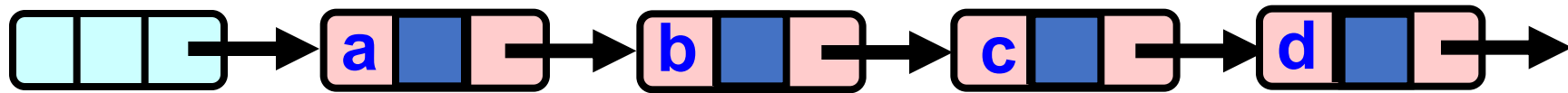
# Can we optimize more?

- Scan the list once?

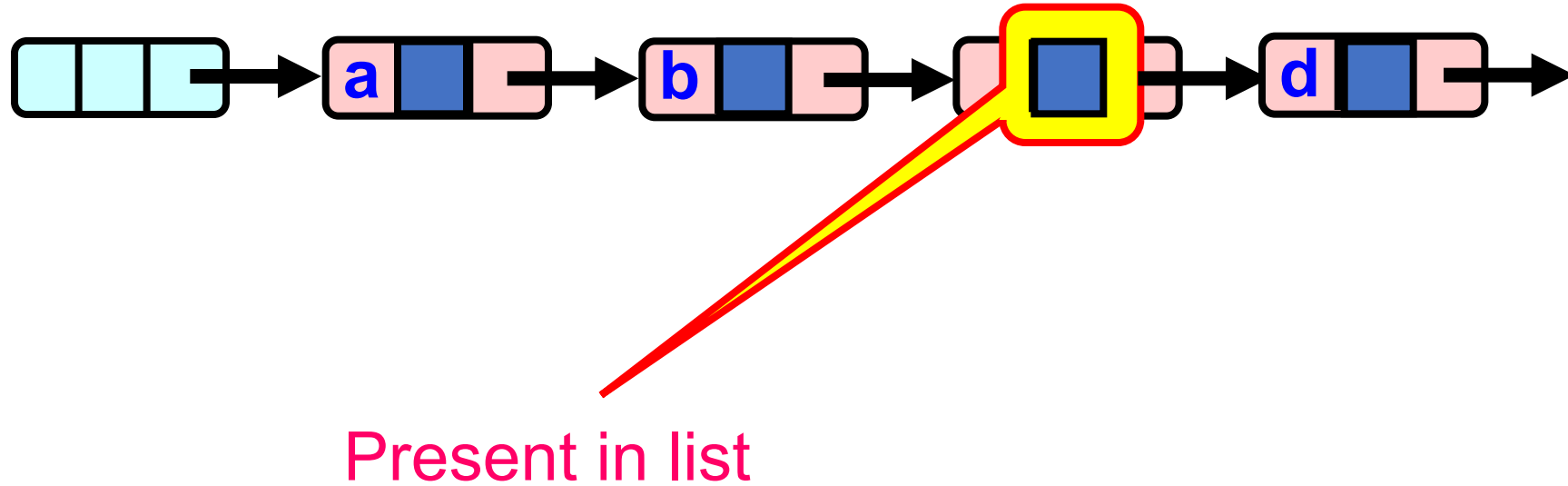
# Two step removal List

- **remove ( )**
  - Scans list (as before)
  - Locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

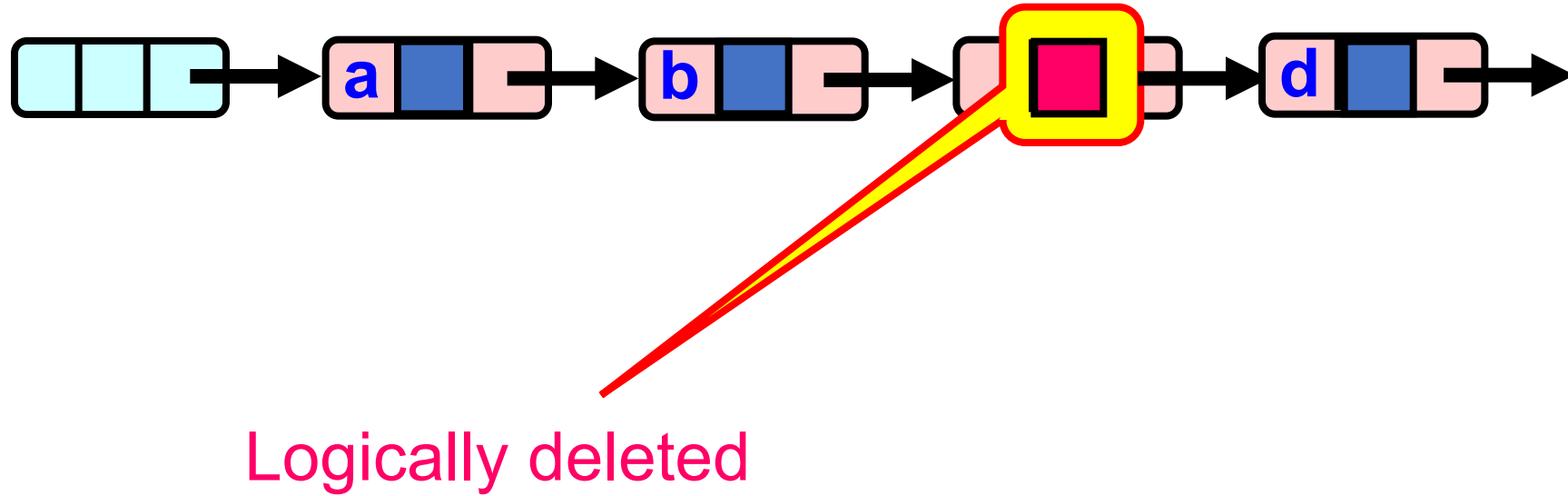
# Two step removal Removal



# Two step removal Removal

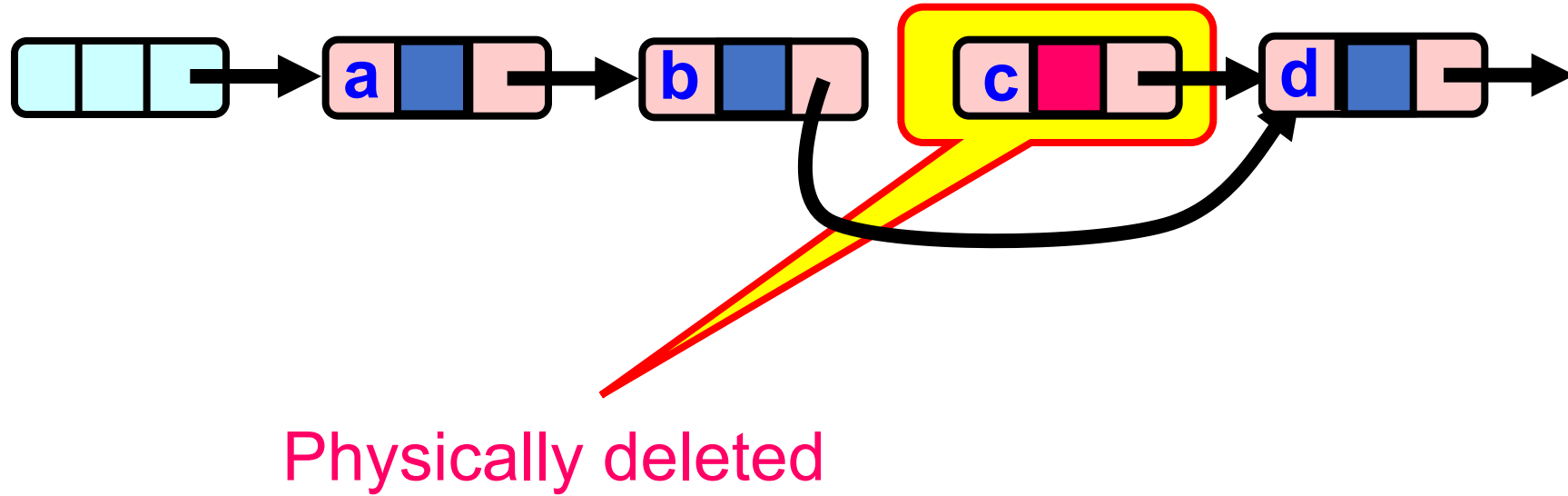


# Two step removal Removal

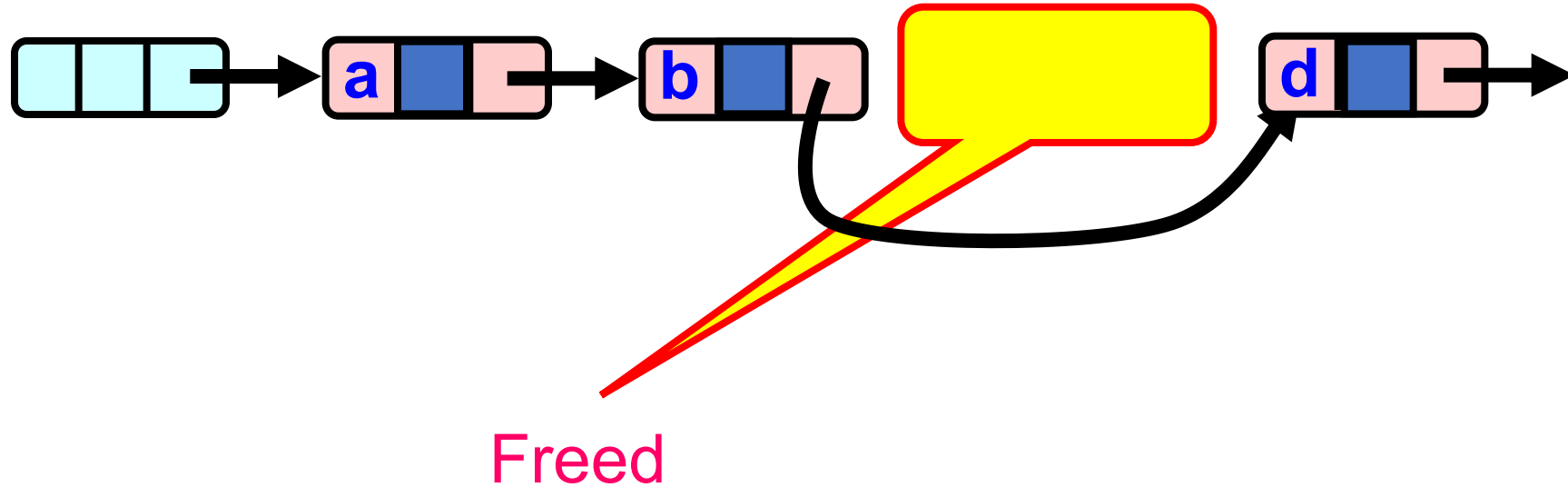




# Two step removal Removal



# Two step removal Removal



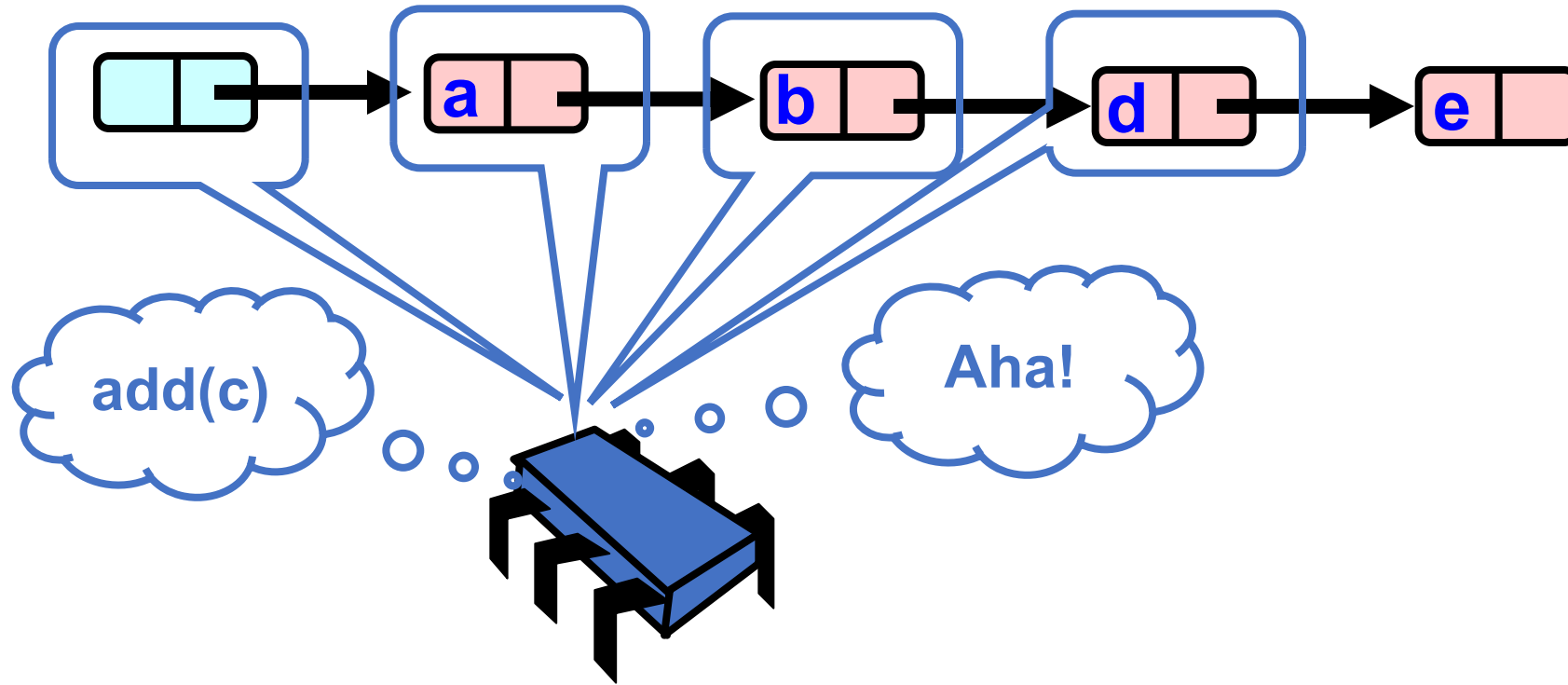
# Two step remove list

- All Methods
  - Scan through locked and marked nodes
- Must still lock pred and curr nodes.

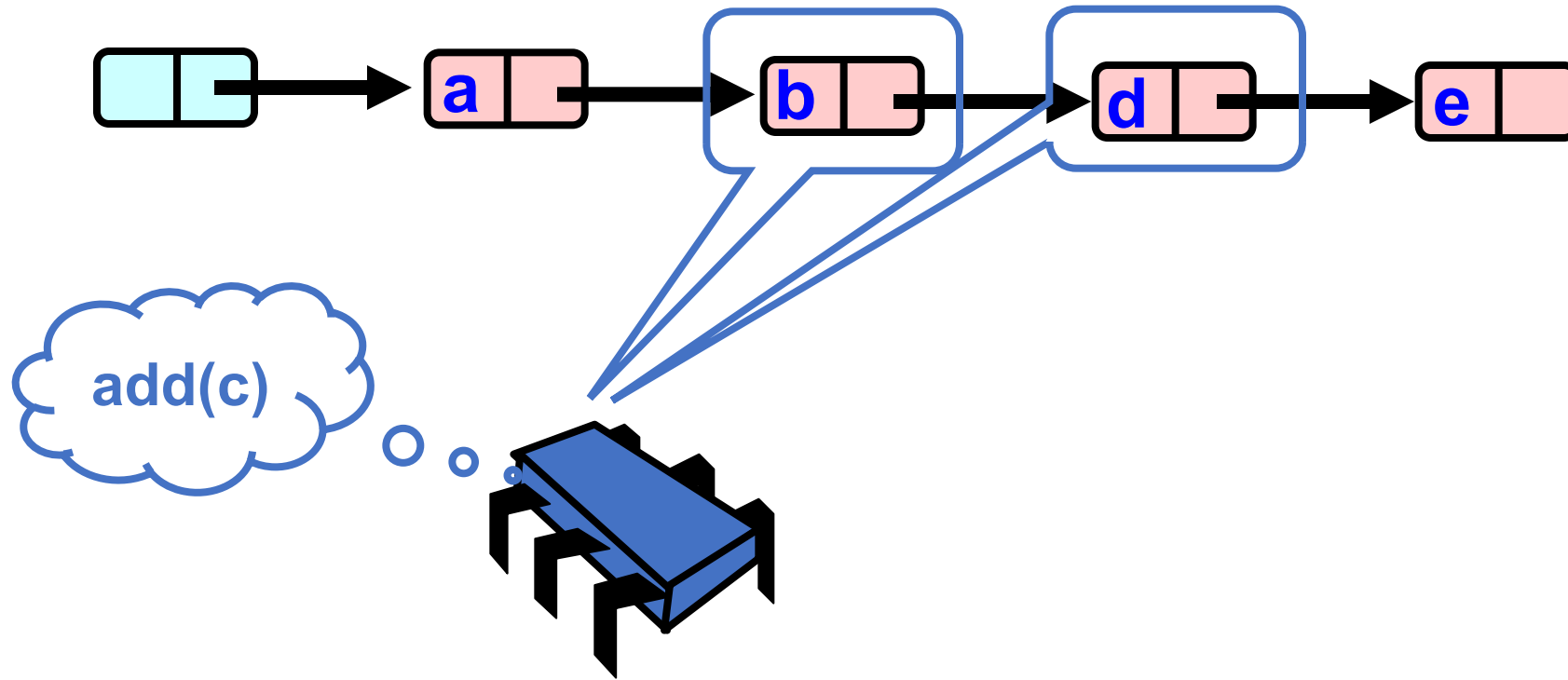
# Validation

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

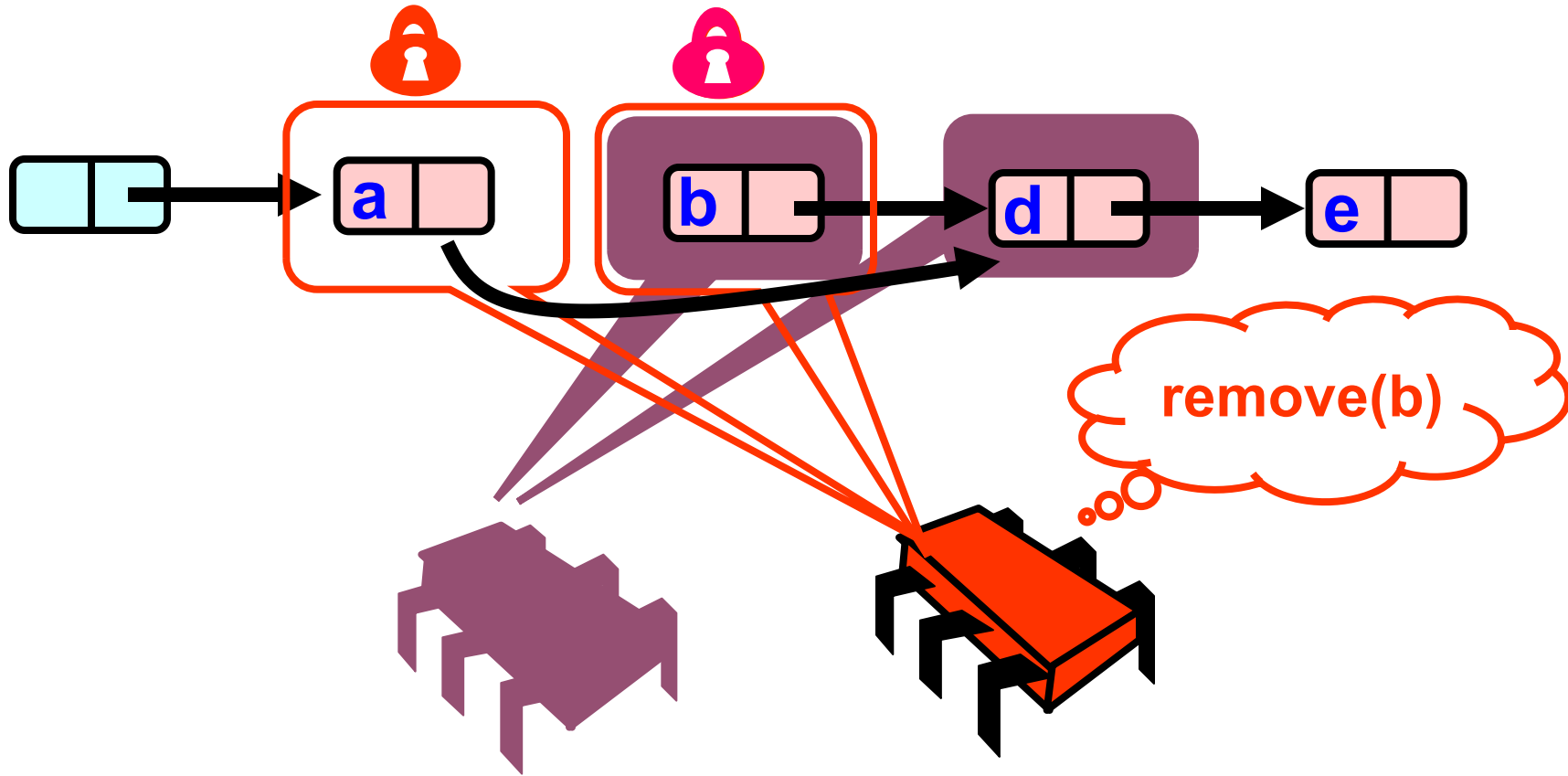
# What could go wrong?



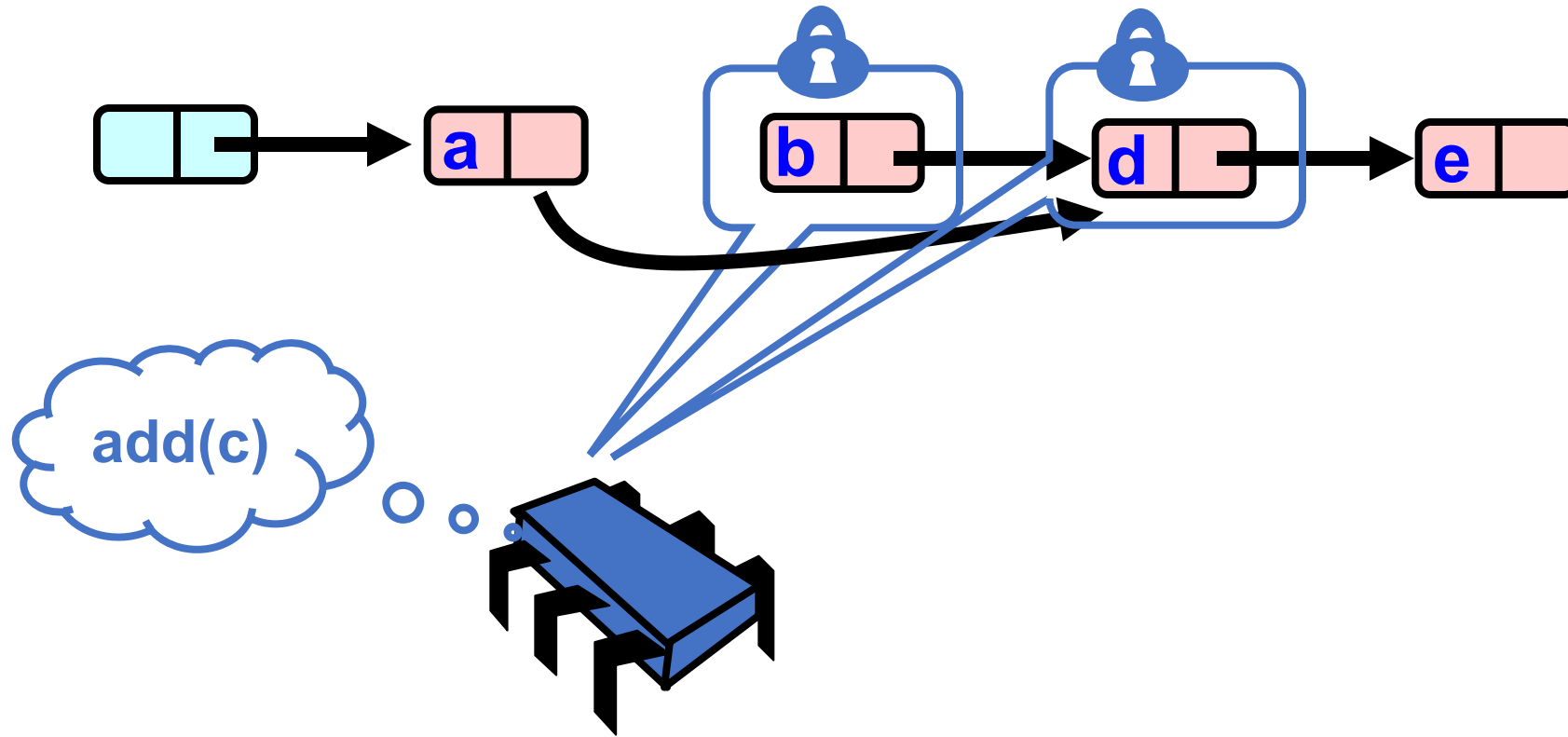
# What could go wrong?



What could go wrong?

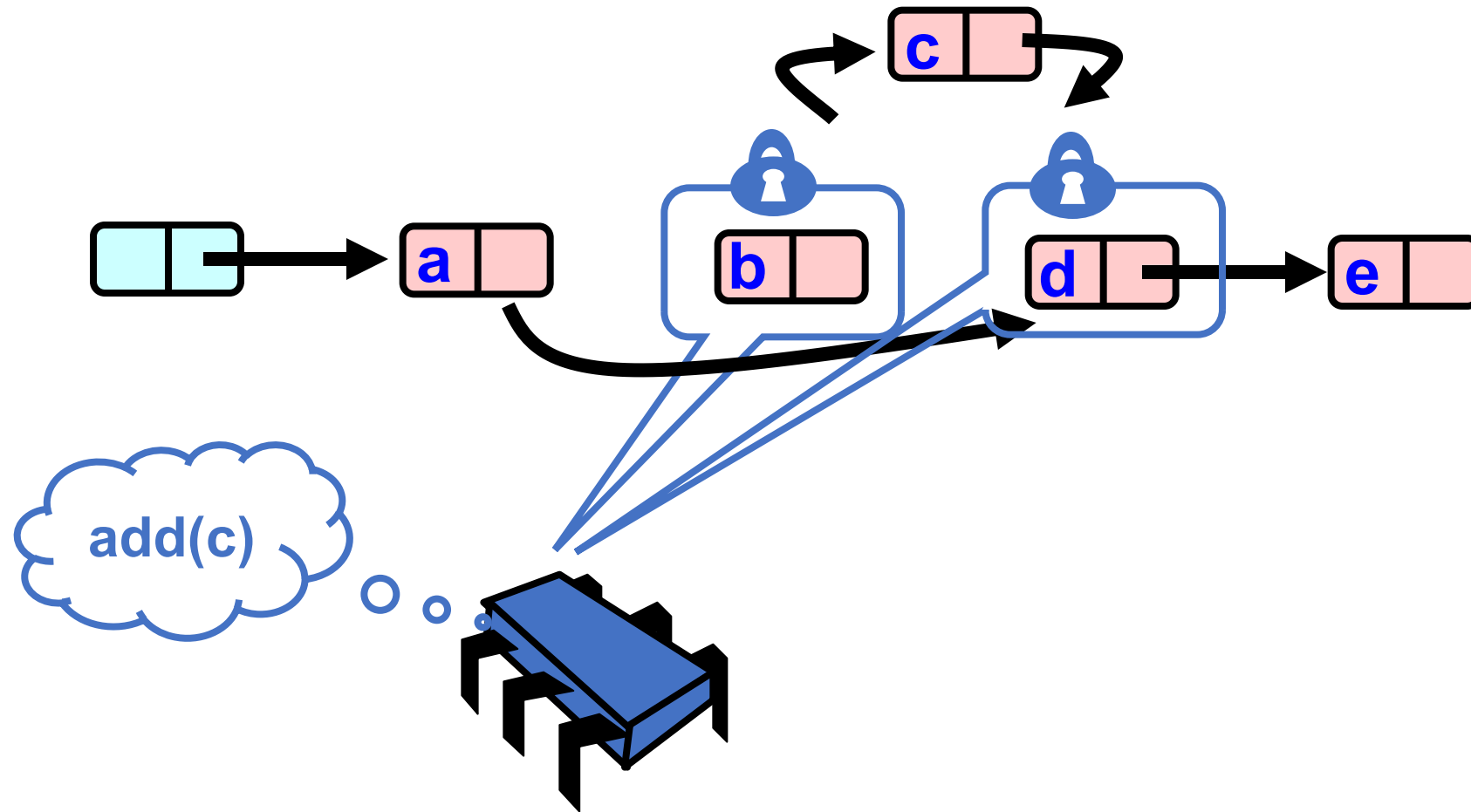


# What could go wrong?

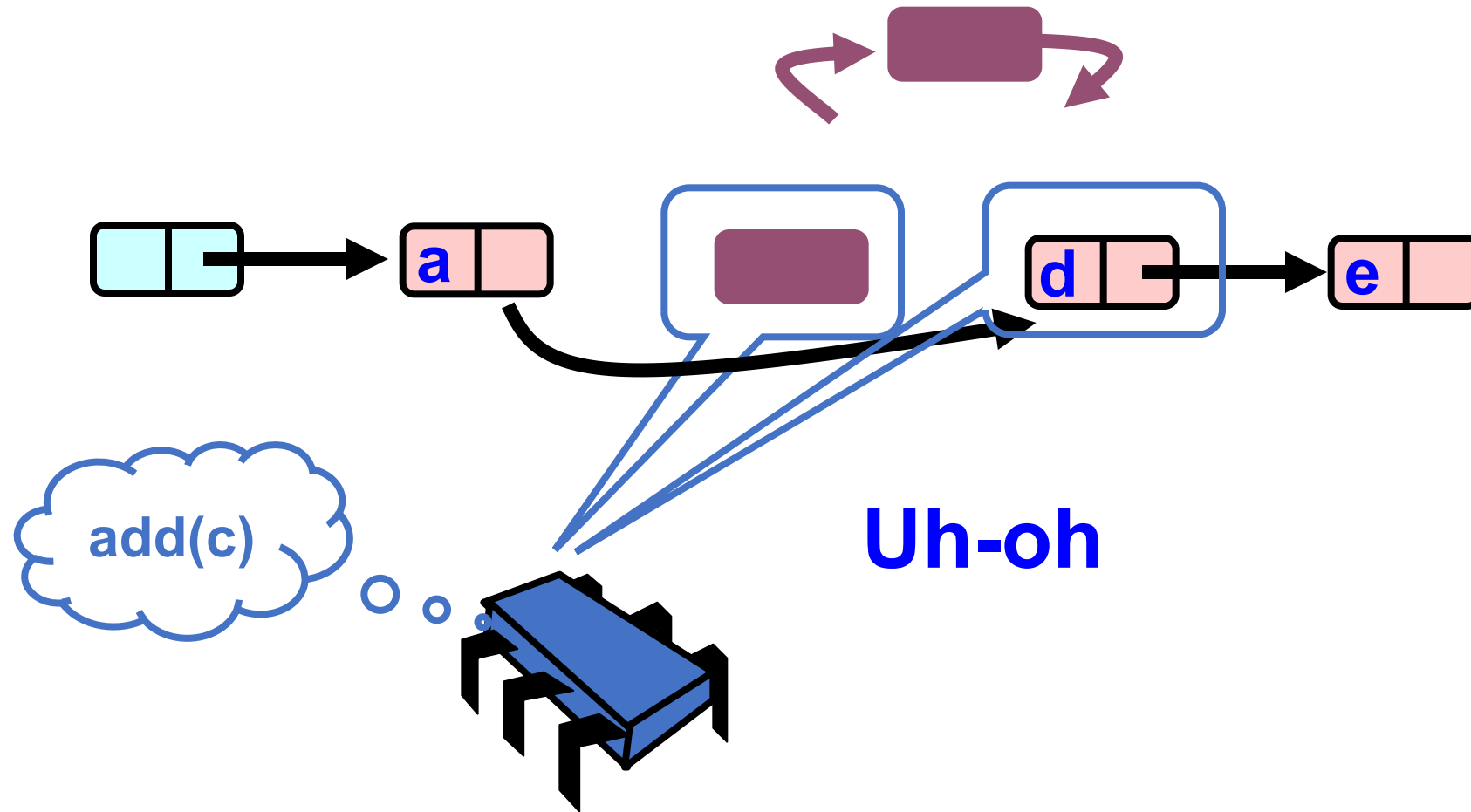




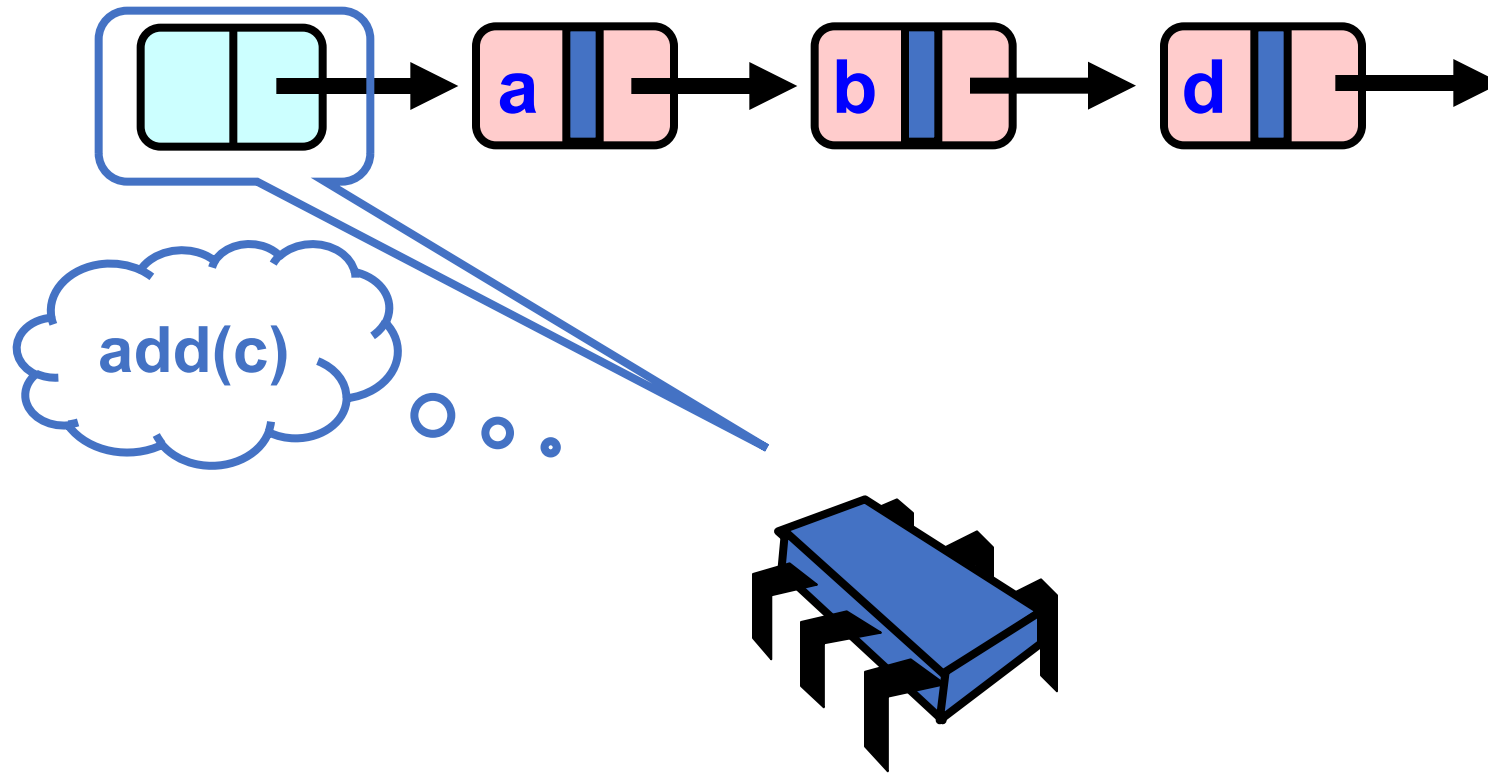
What could go wrong?



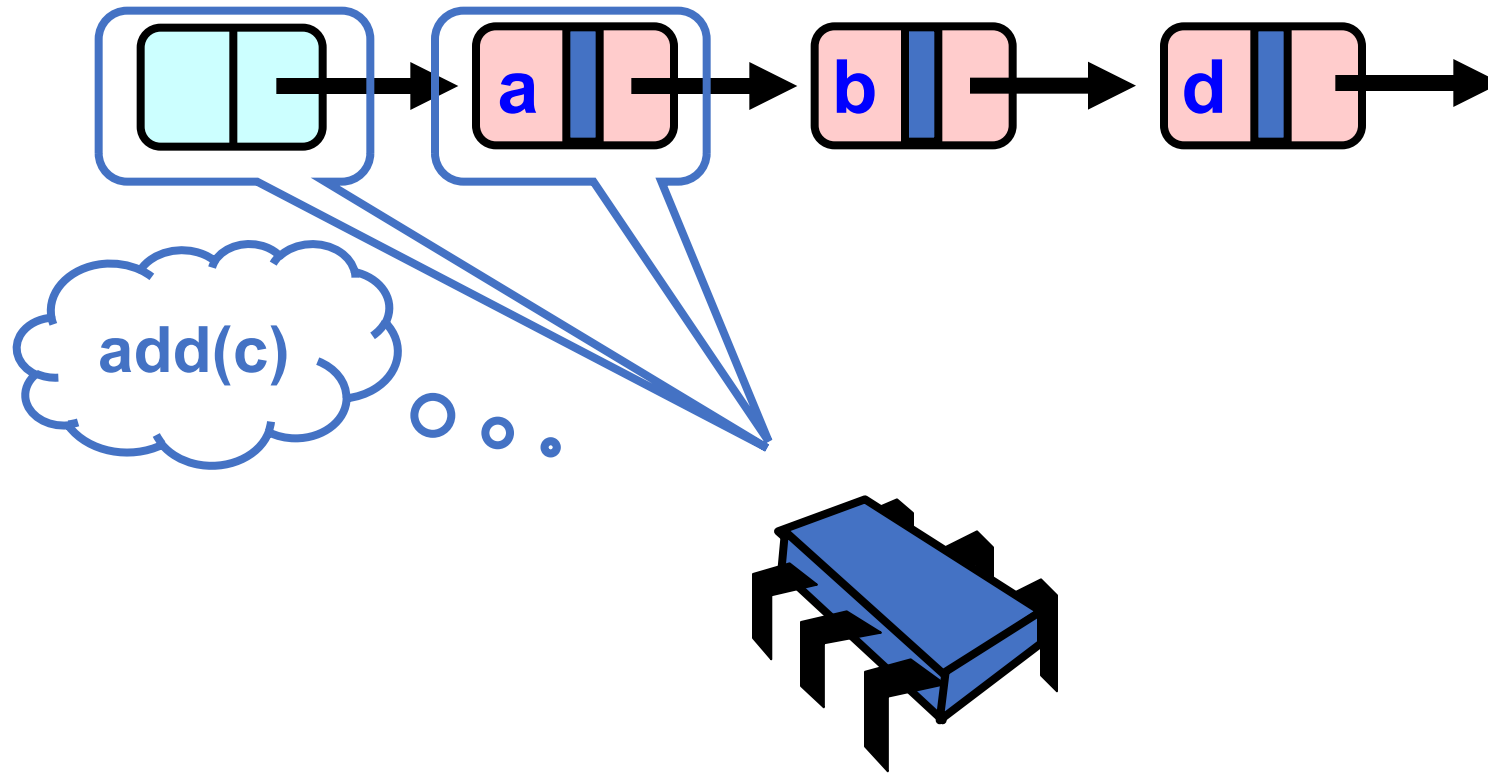
What could go wrong?



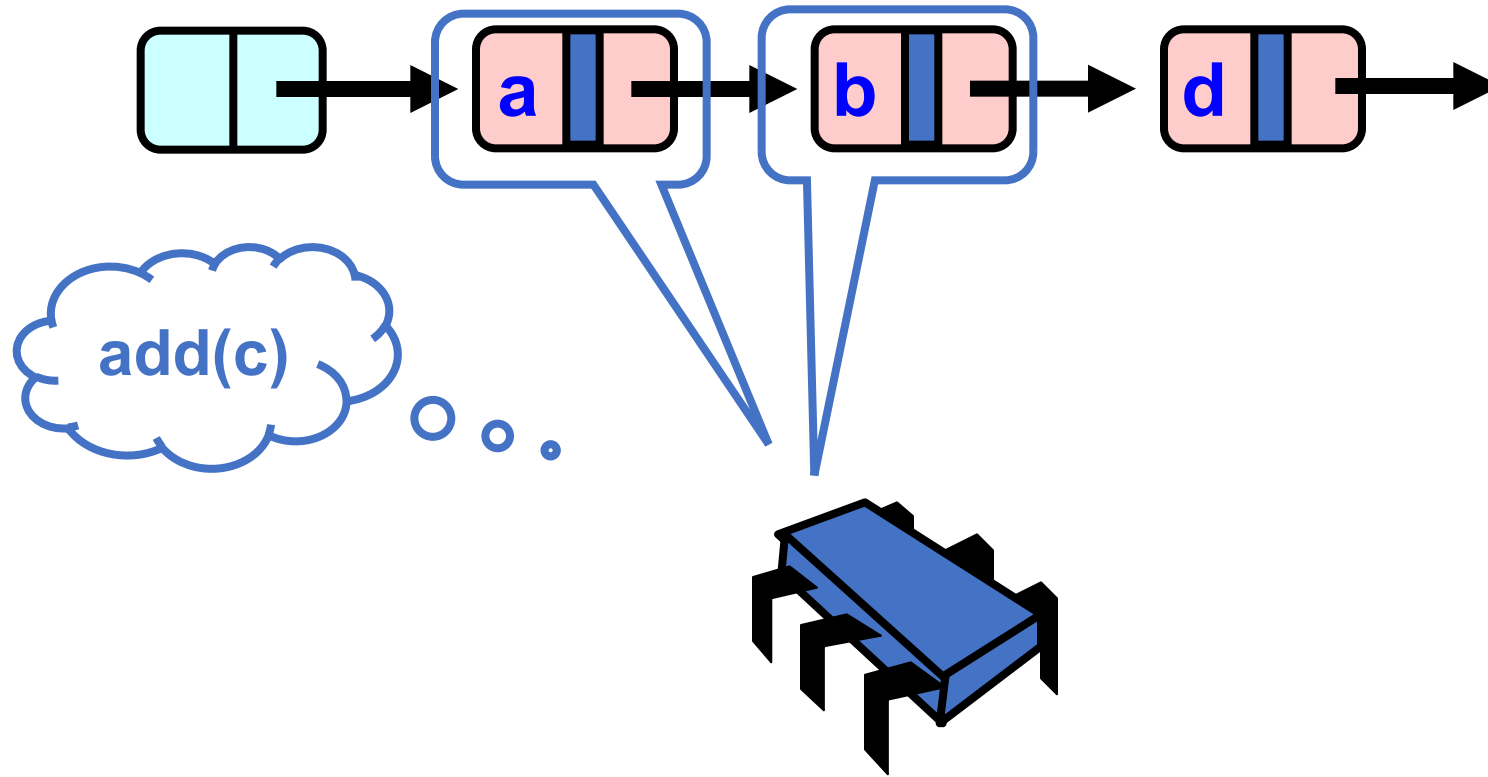
# Fixed with logical flag



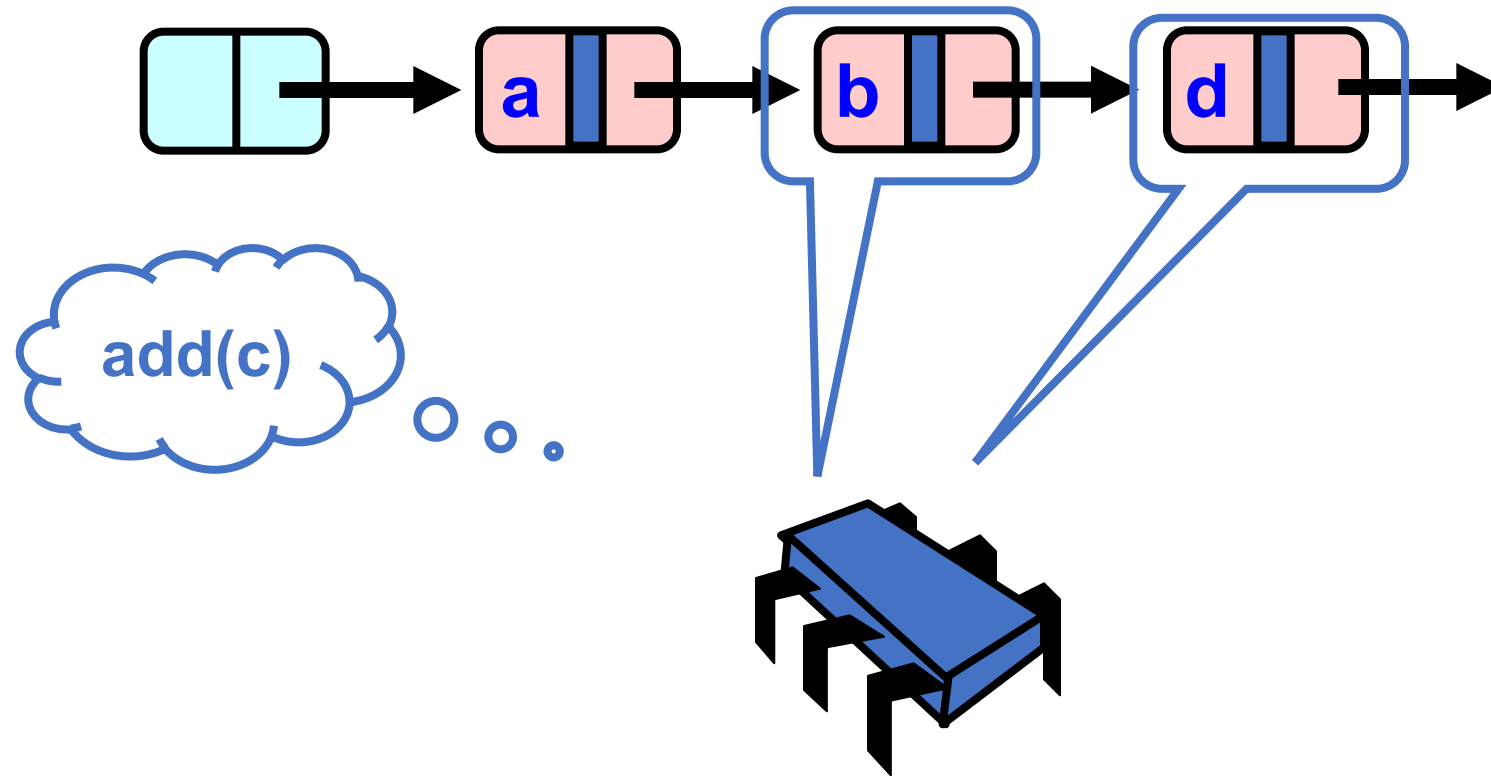
# Fixed with logical flag



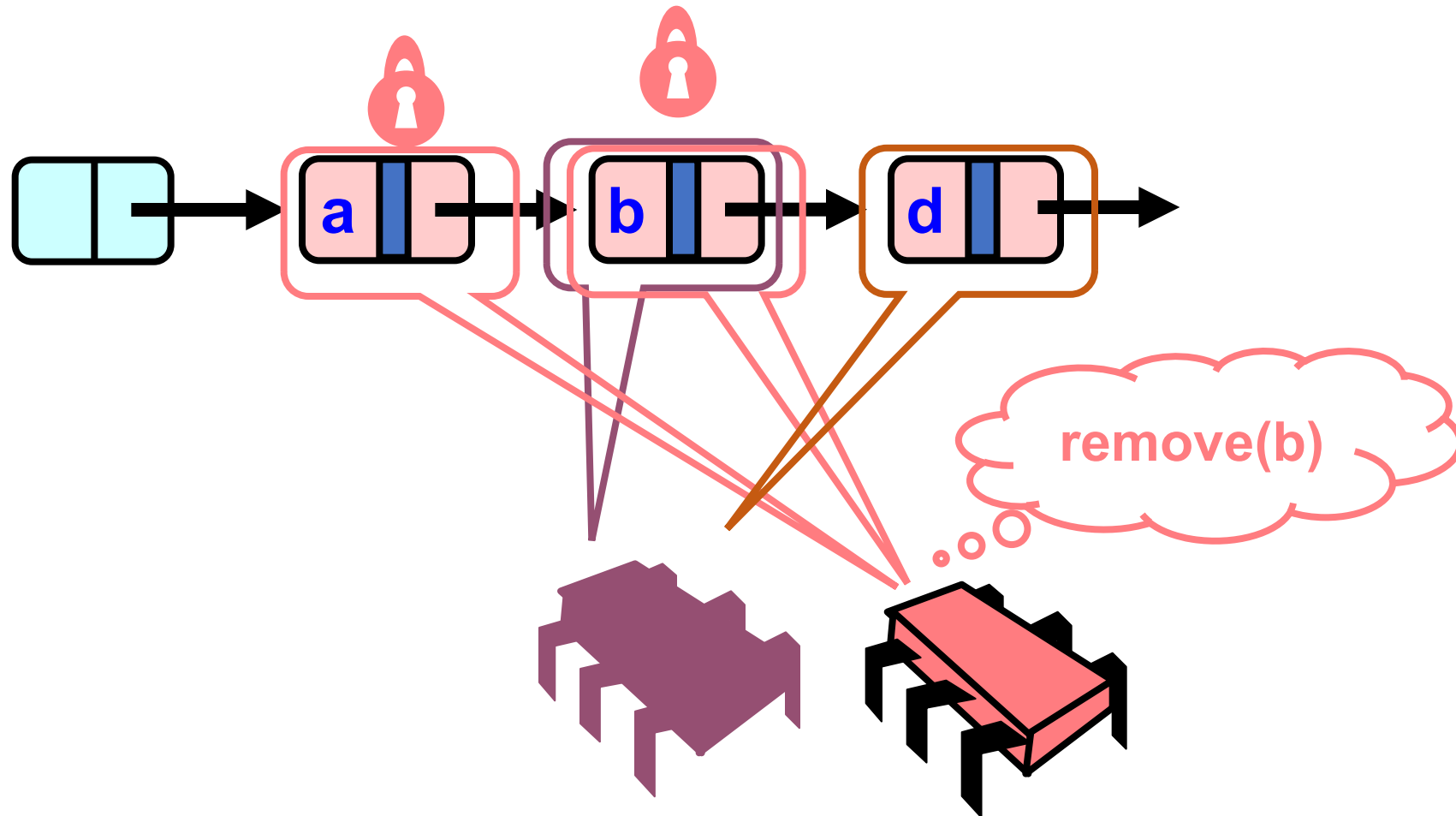
# Fixed with logical flag



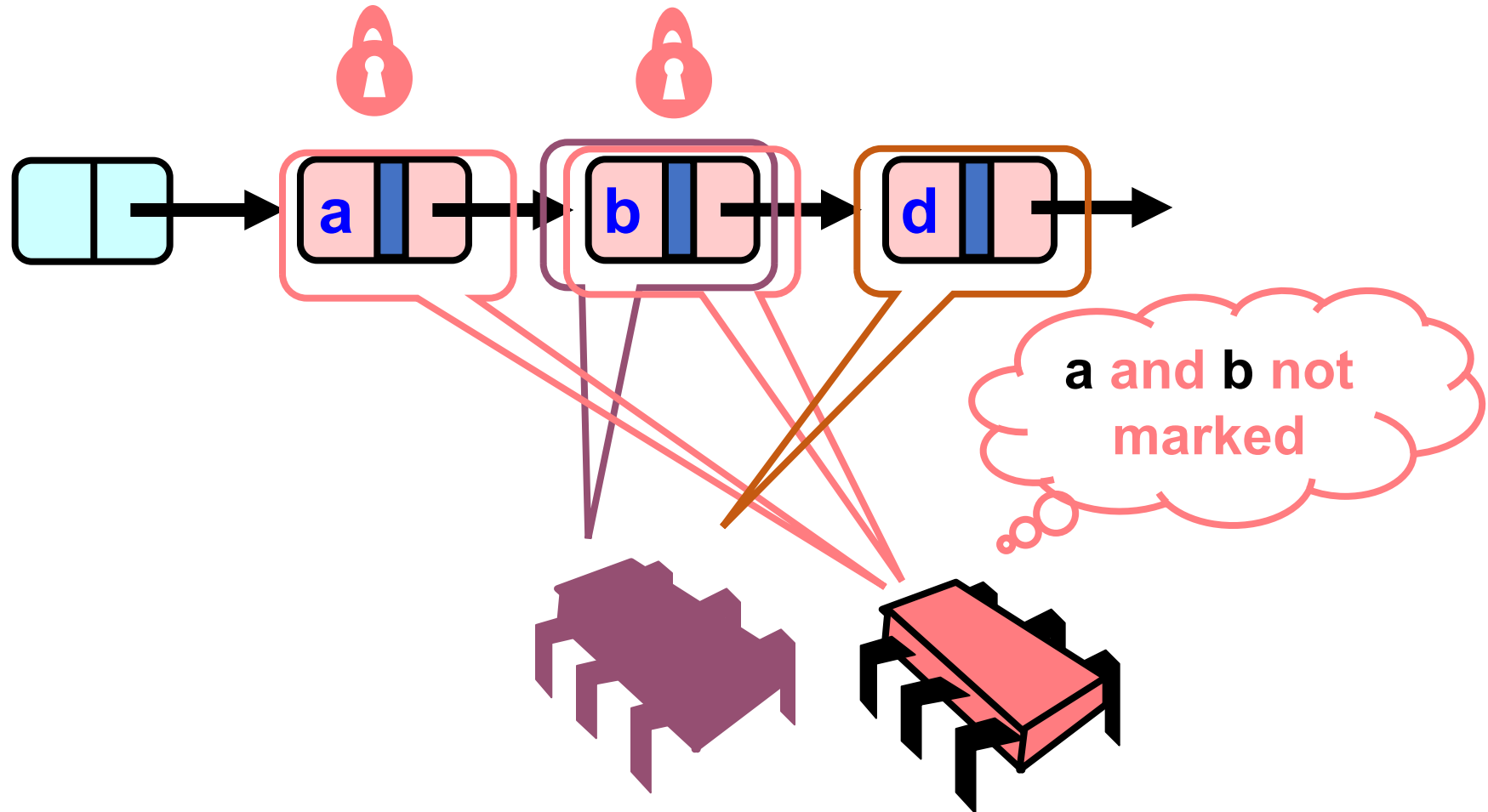
# Fixed with logical flag



# Fixed with logical flag

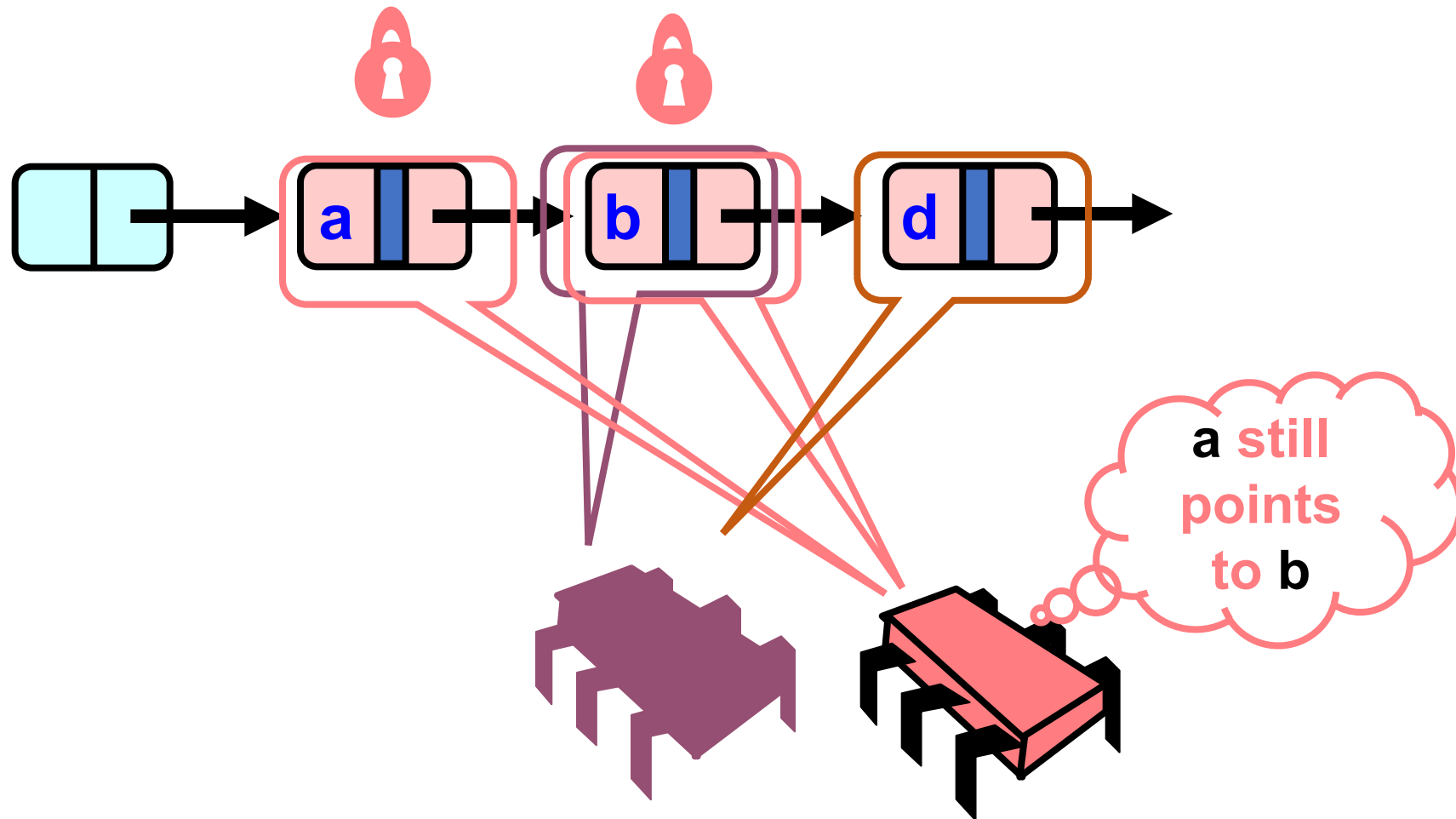


# Fixed with logical flag

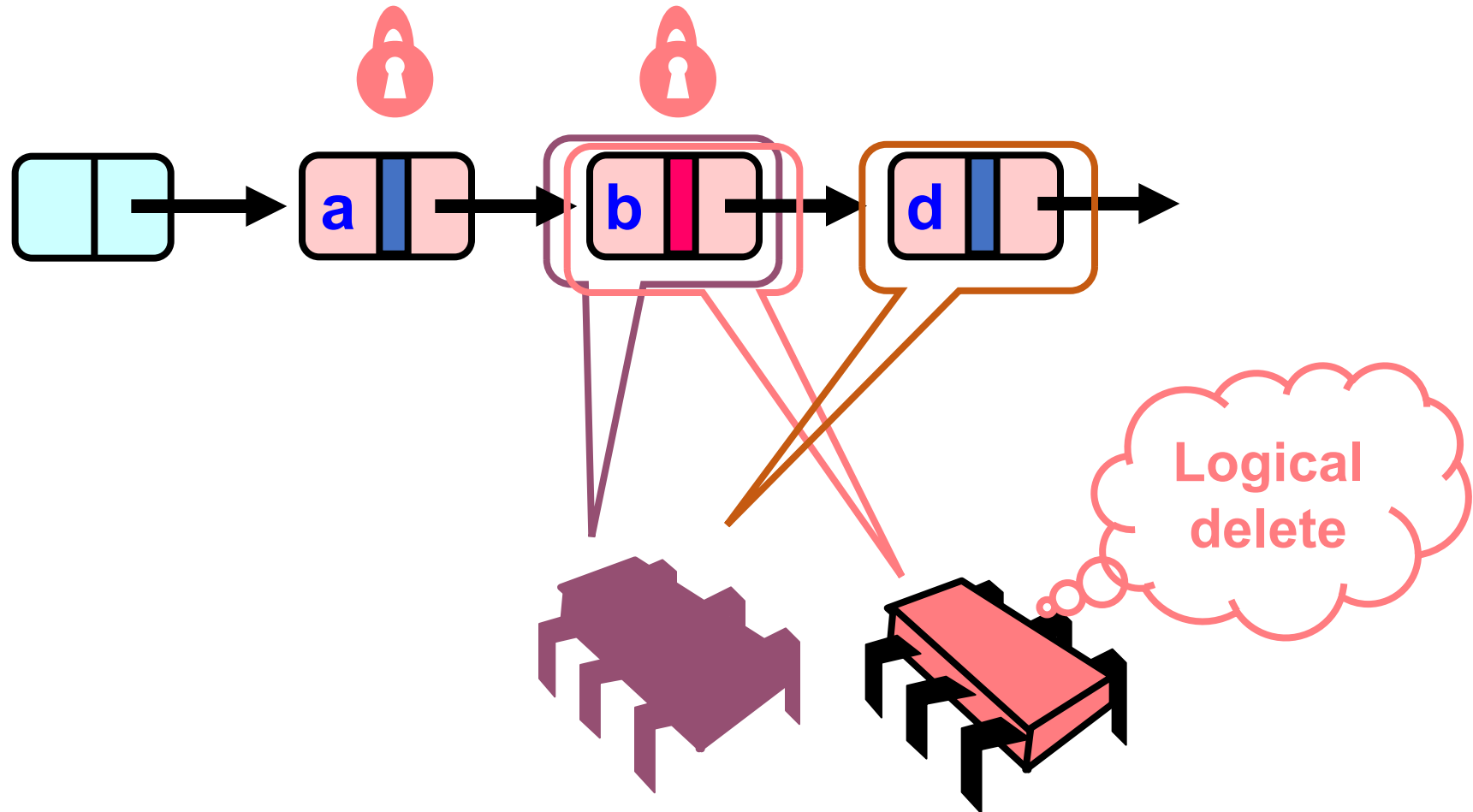




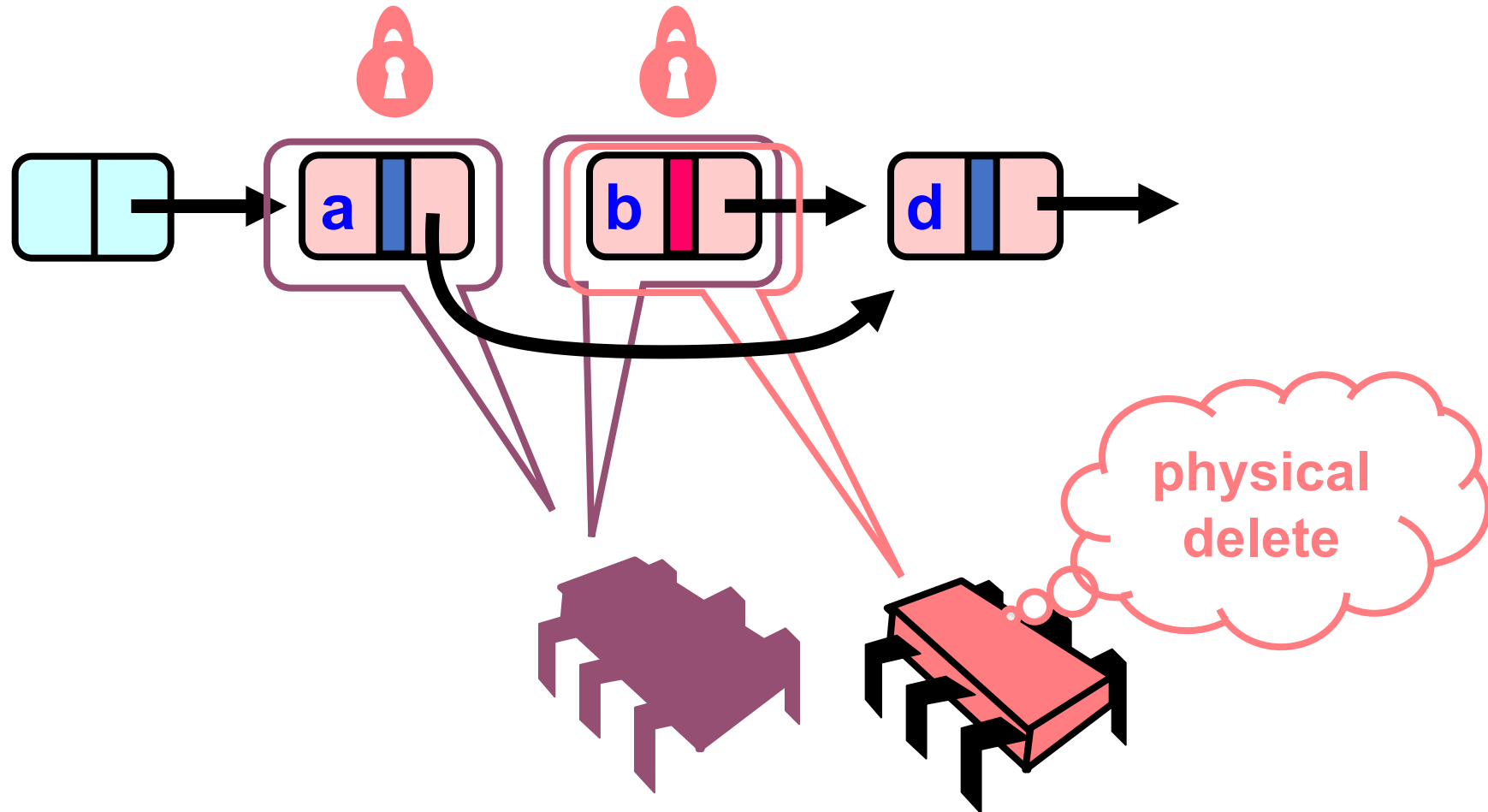
# Fixed with logical flag



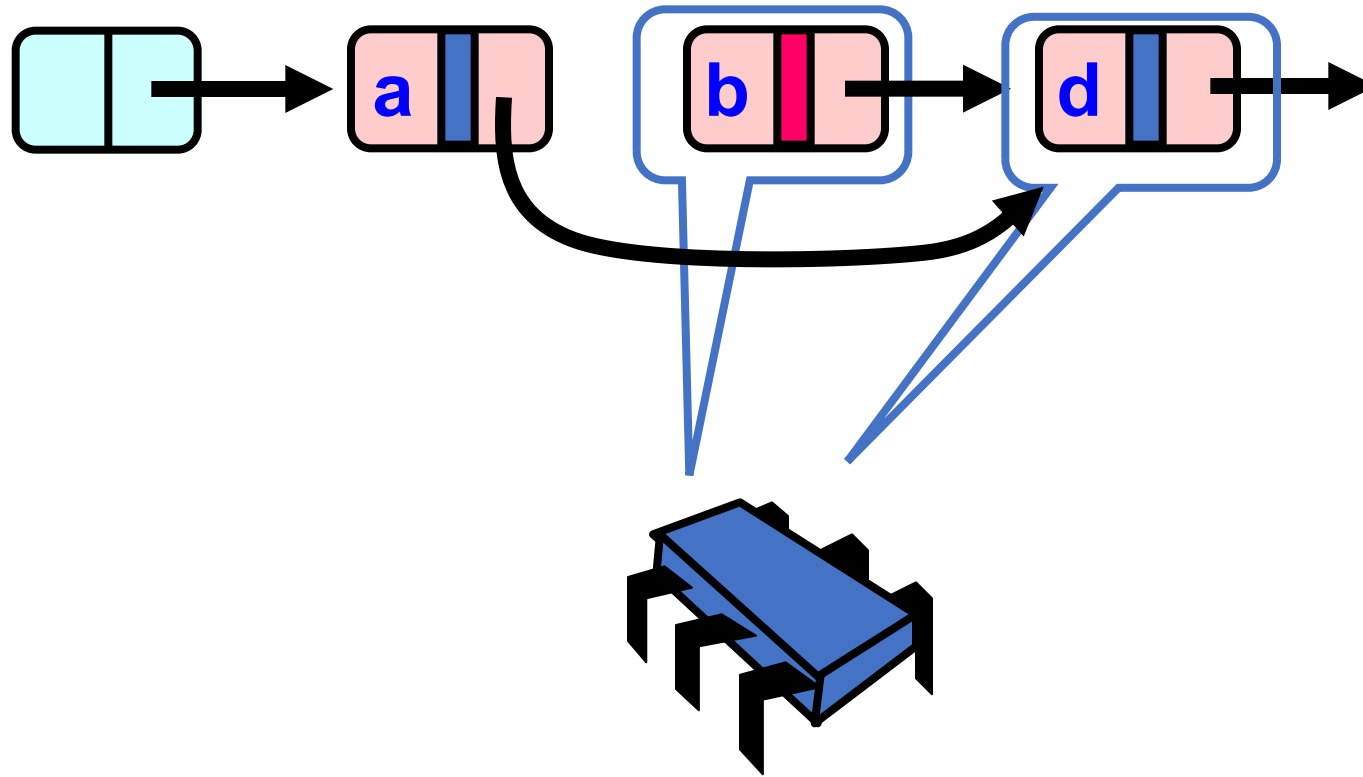
# Fixed with logical flag



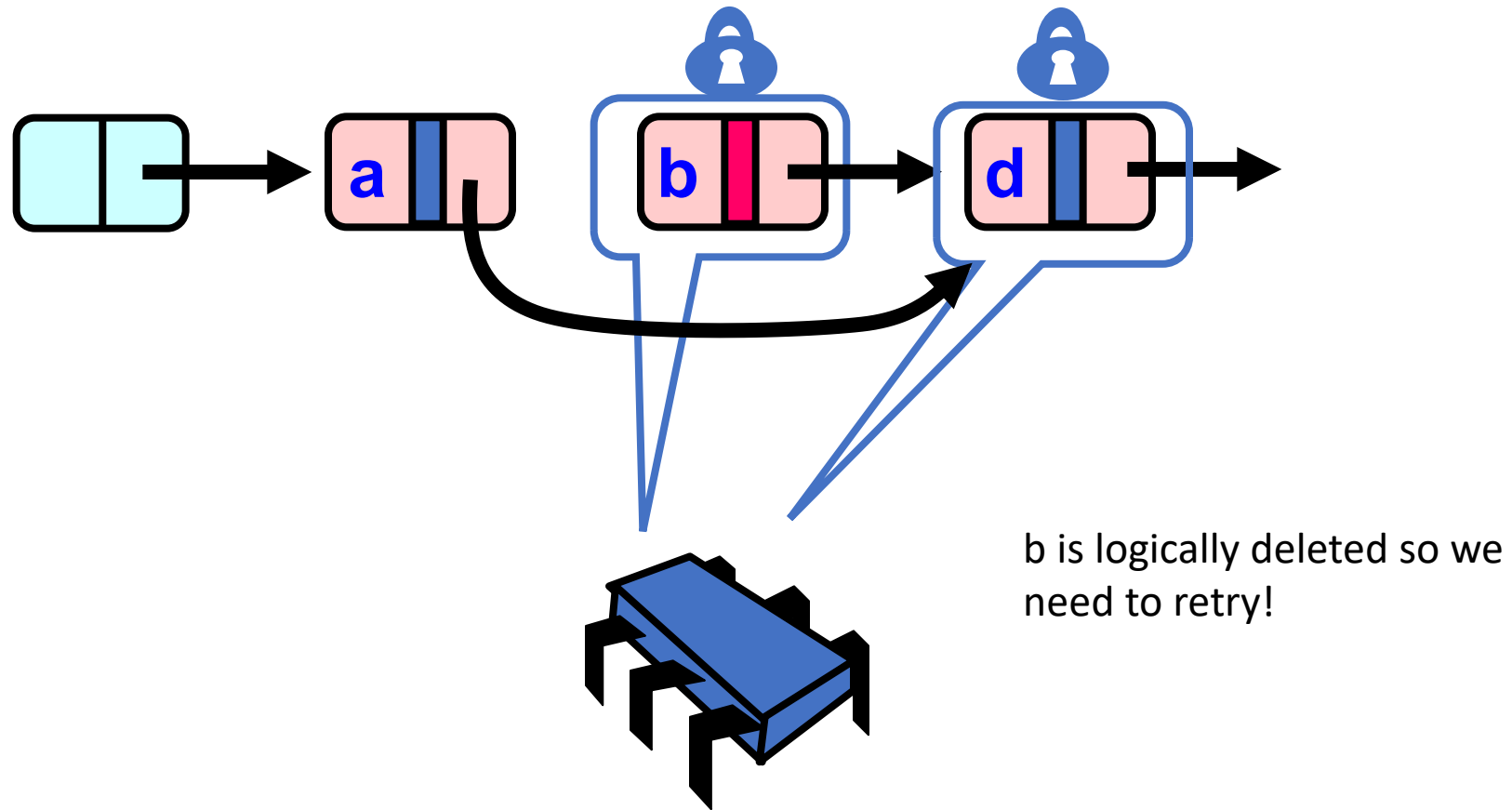
# Fixed with logical flag



Fixed with logical flag



# Fixed with logical flag



# To complete the picture

- Need to do similar reasoning with all combination of object methods.
- More information in the book!

# How good?

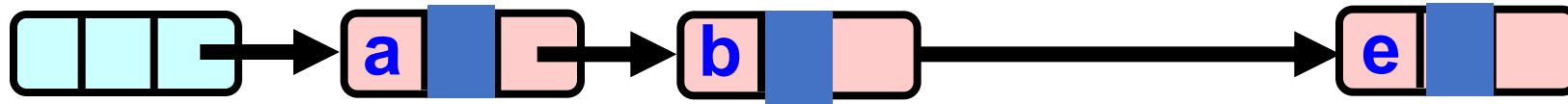
- Good:
  - Uncontended calls don't re-traverse
- Bad
  - add() and remove() use locks

# Lock-free Lists

- Next logical step
  - lock-free add() and remove()
- What sort of atomics do we need?
  - Loads/stores -> RMWs?

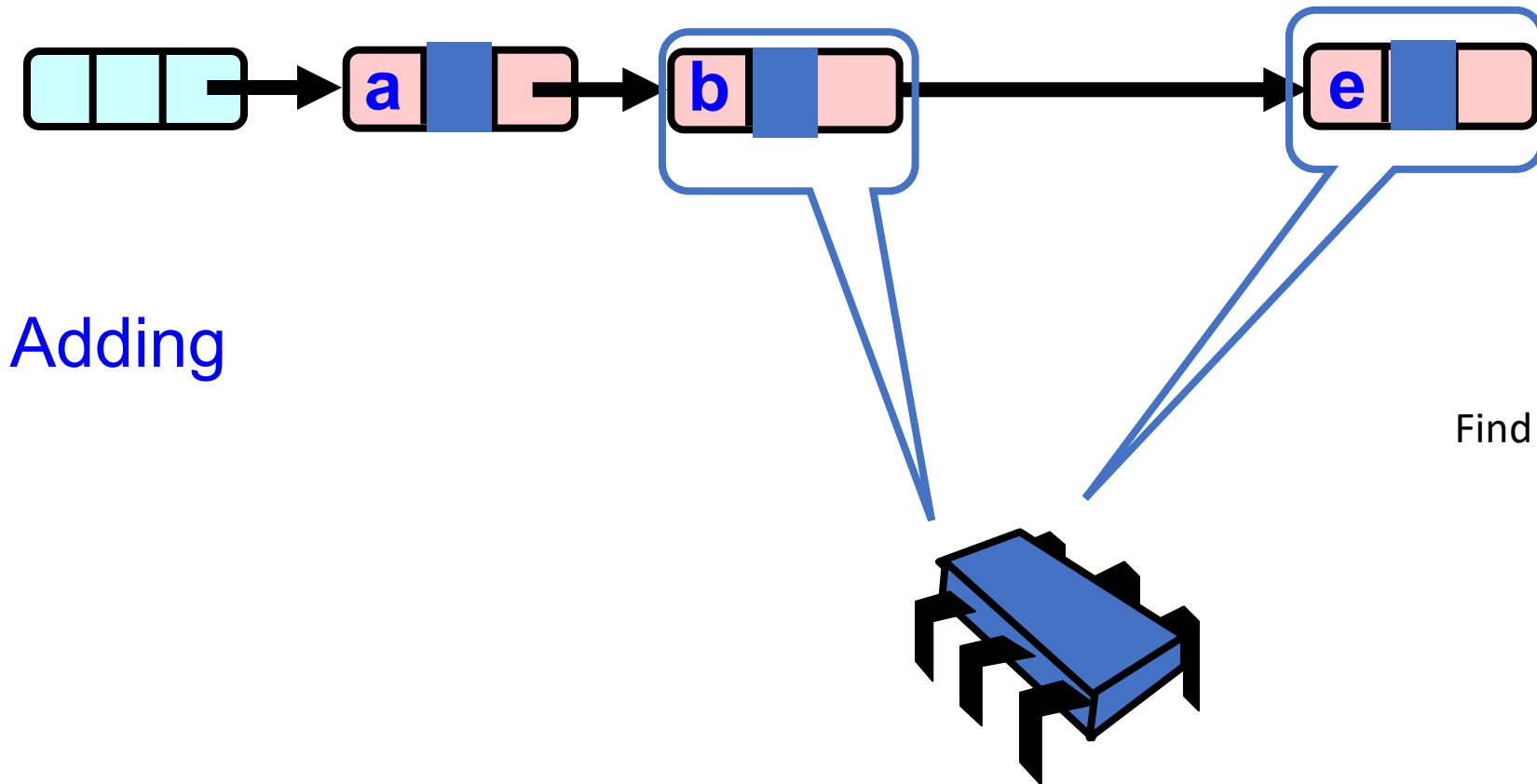


# Lock-free Lists

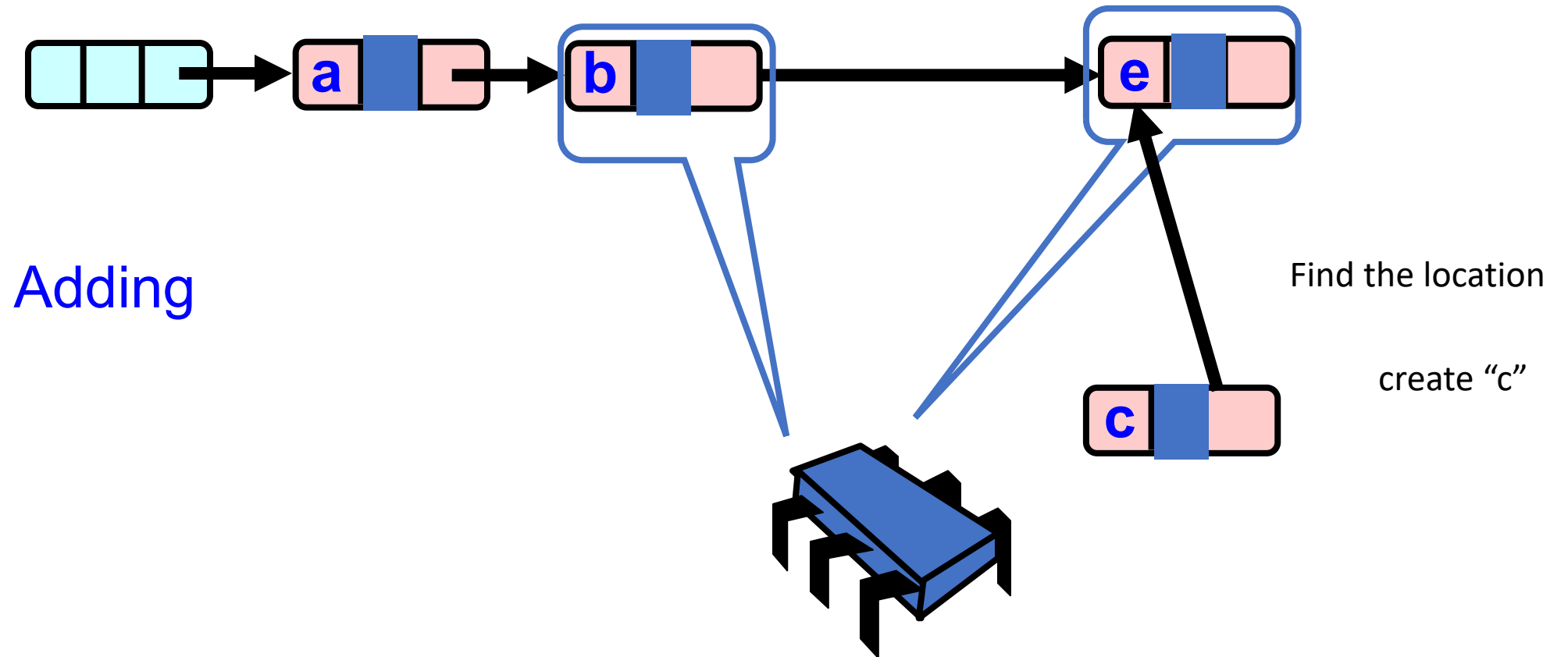


Adding

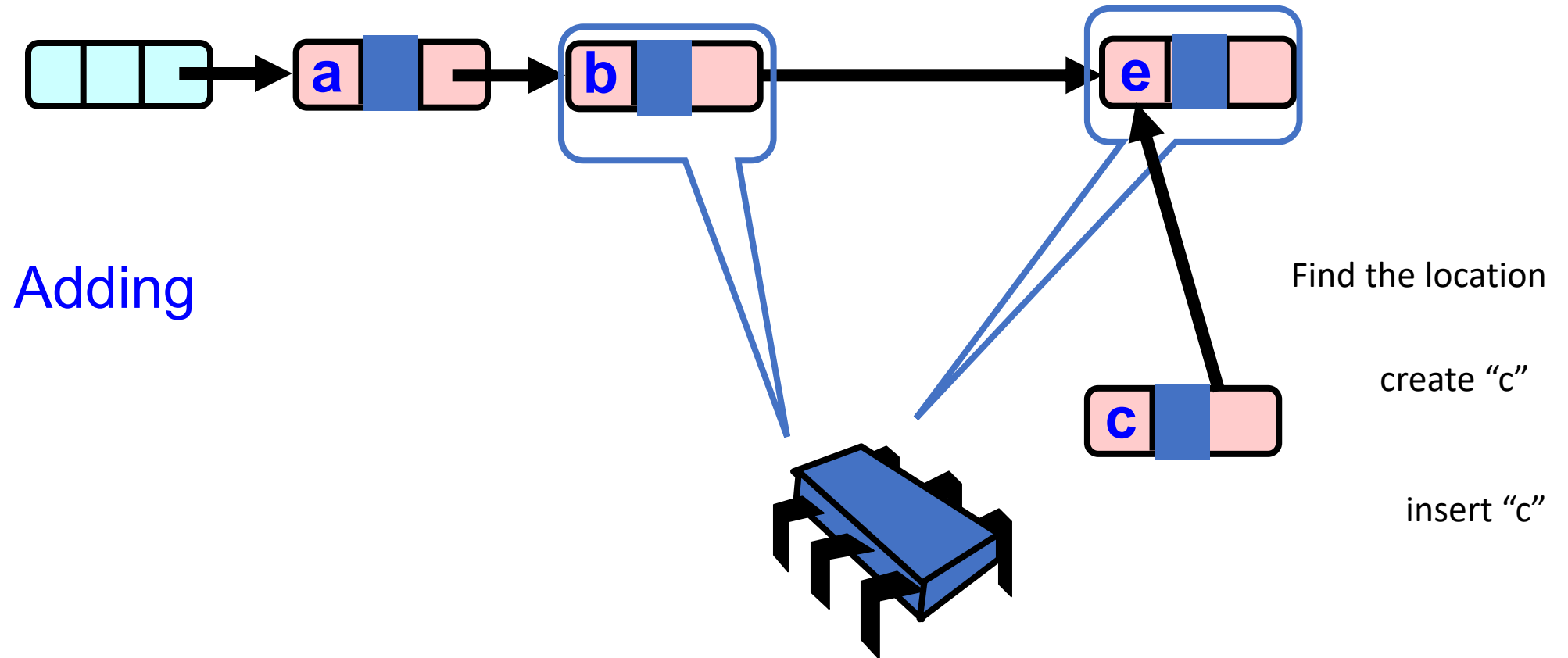
# Lock-free Lists



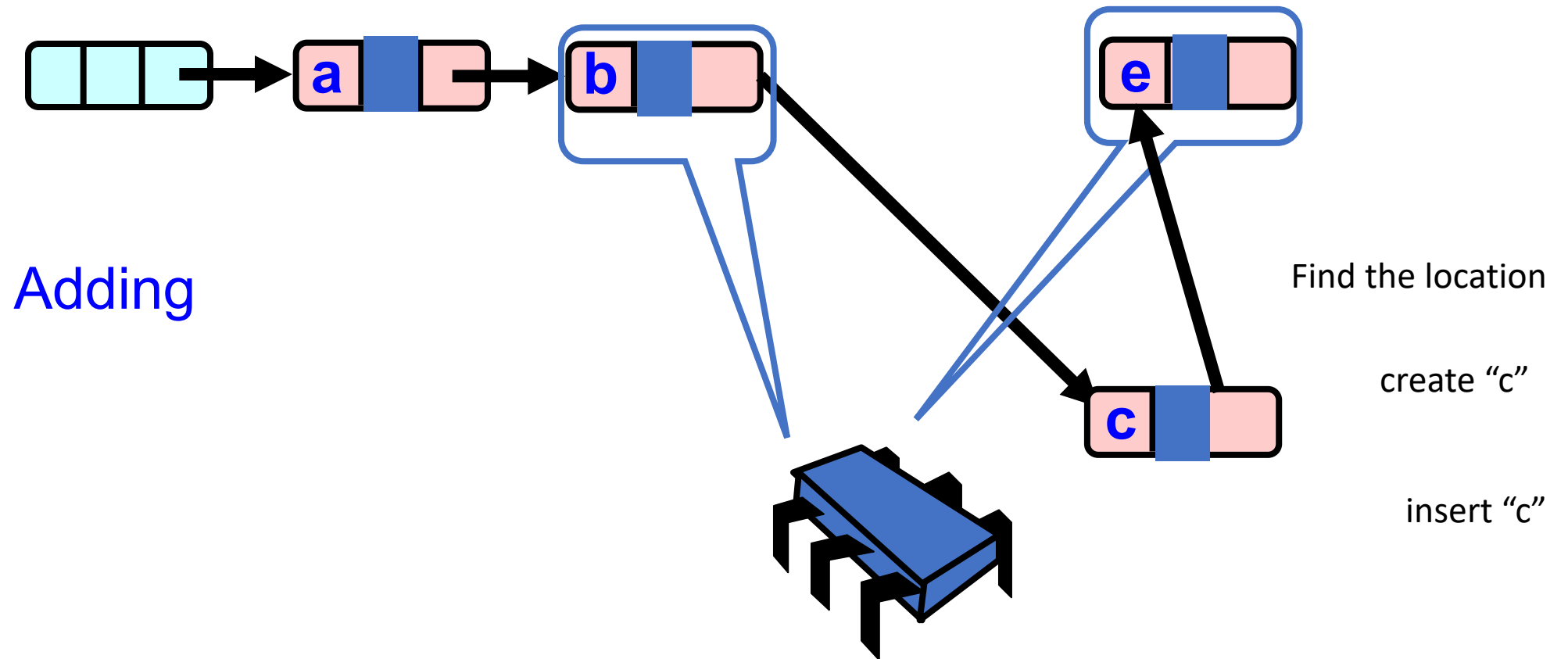
# Lock-free Lists



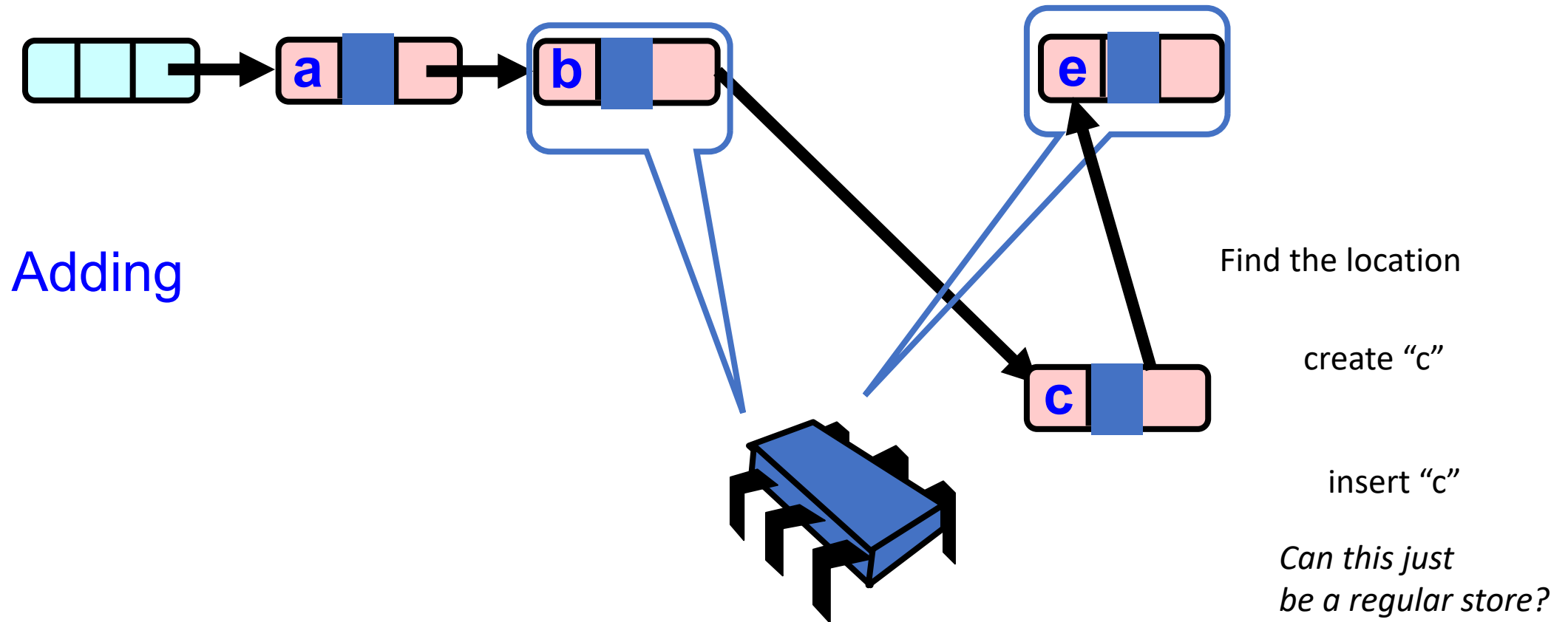
# Lock-free Lists



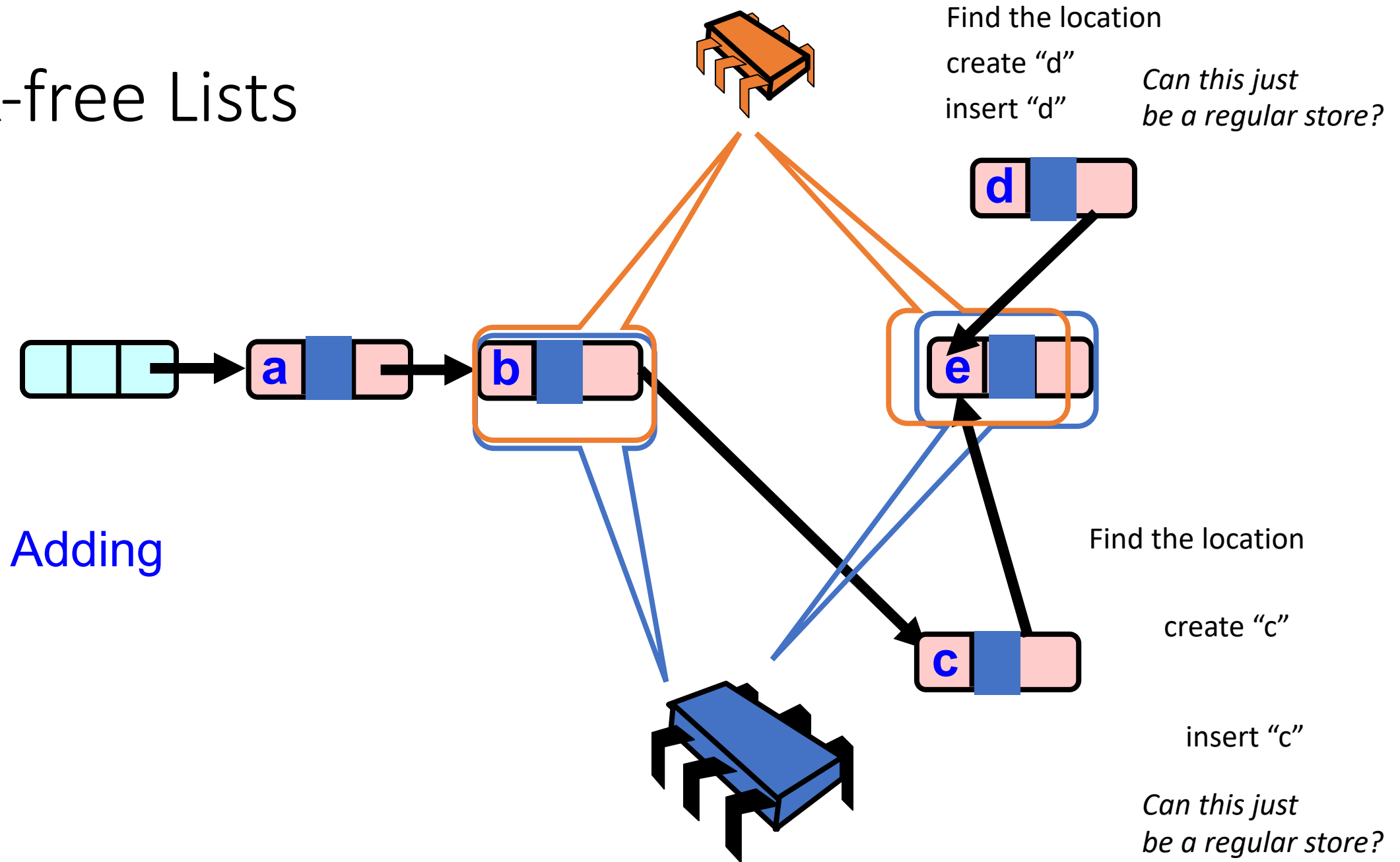
# Lock-free Lists



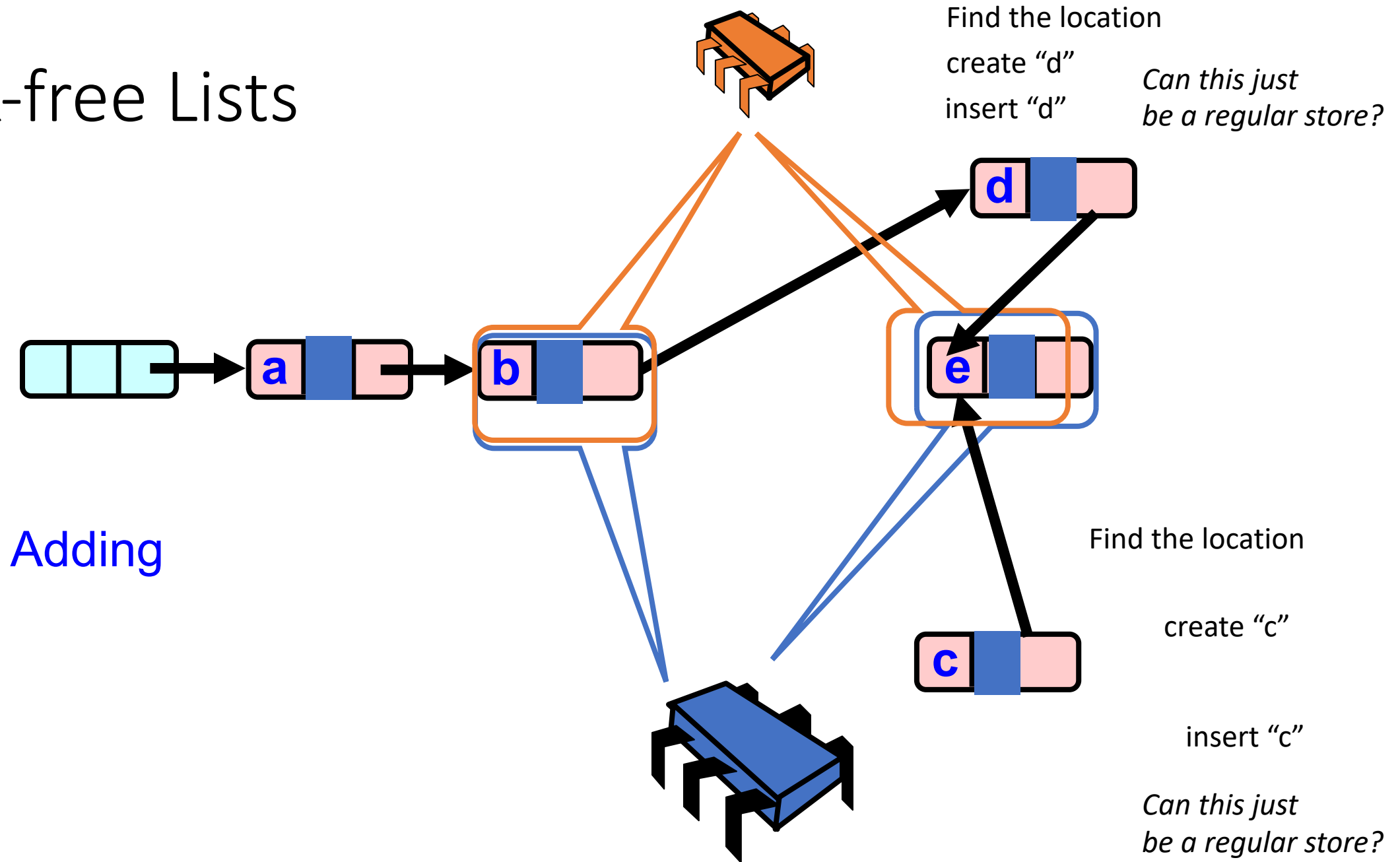
# Lock-free Lists



# Lock-free Lists

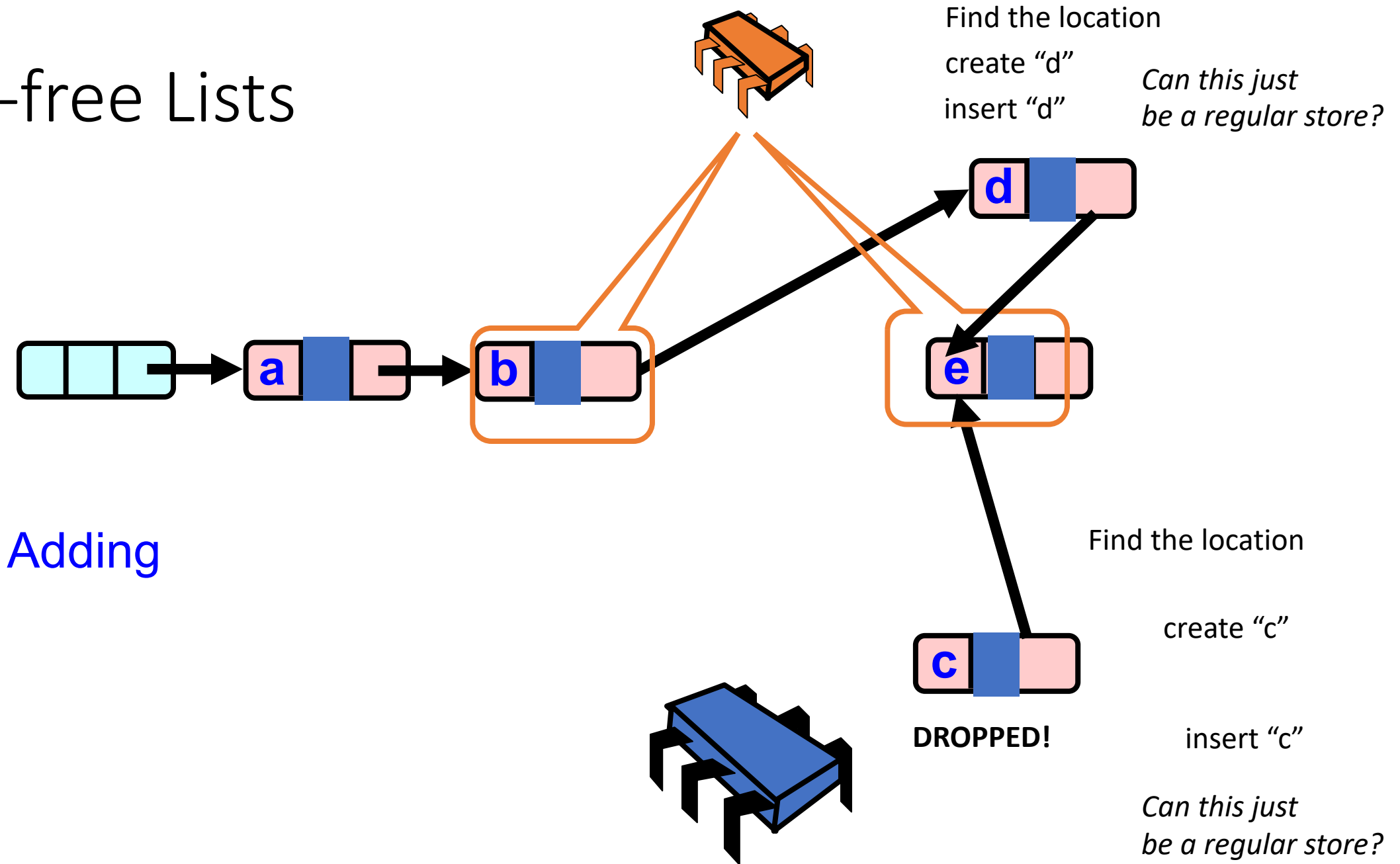


# Lock-free Lists

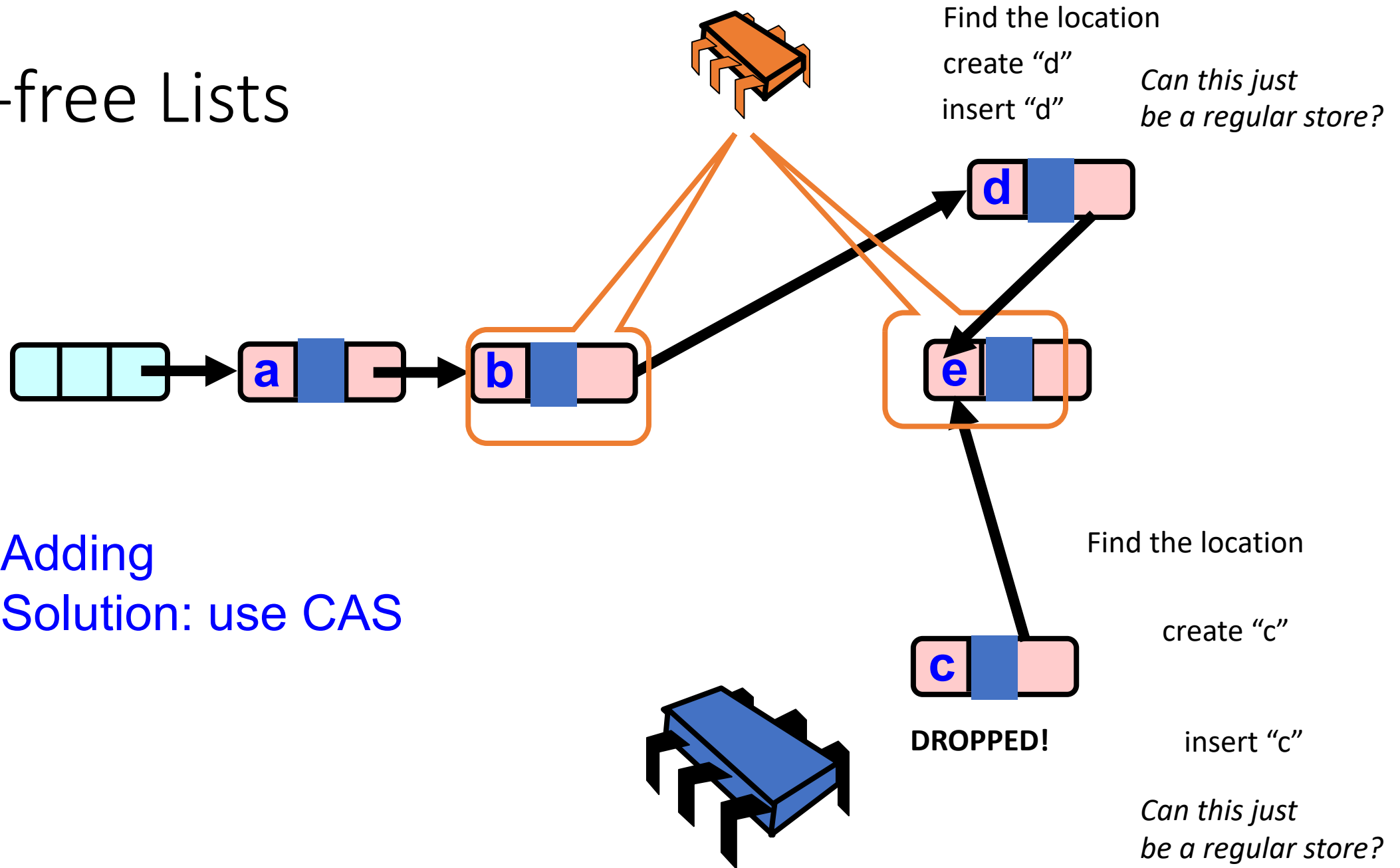




# Lock-free Lists



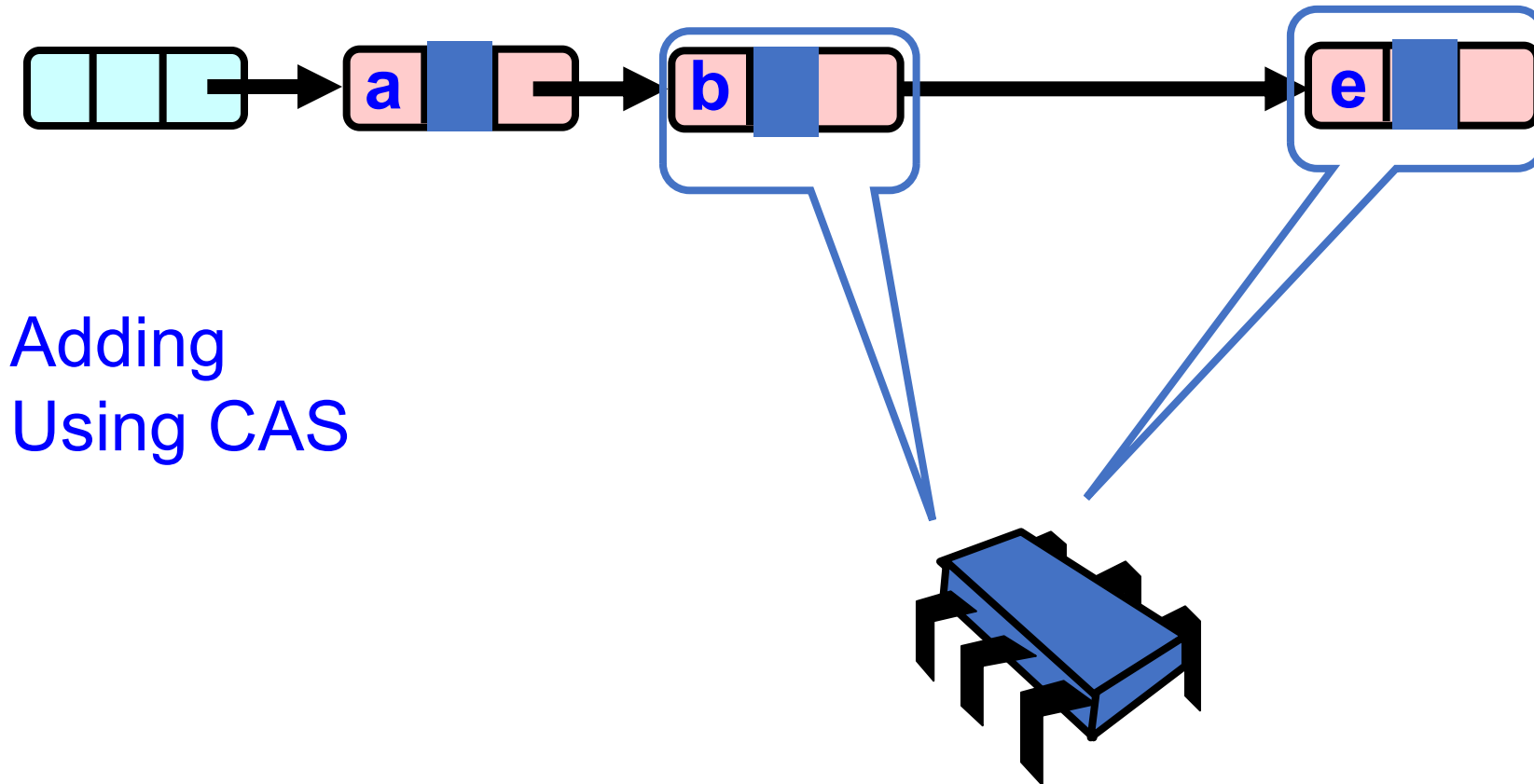
# Lock-free Lists



# Lock-free Lists

Find the location  
Cache your insertion  
point!

`b.next == e`



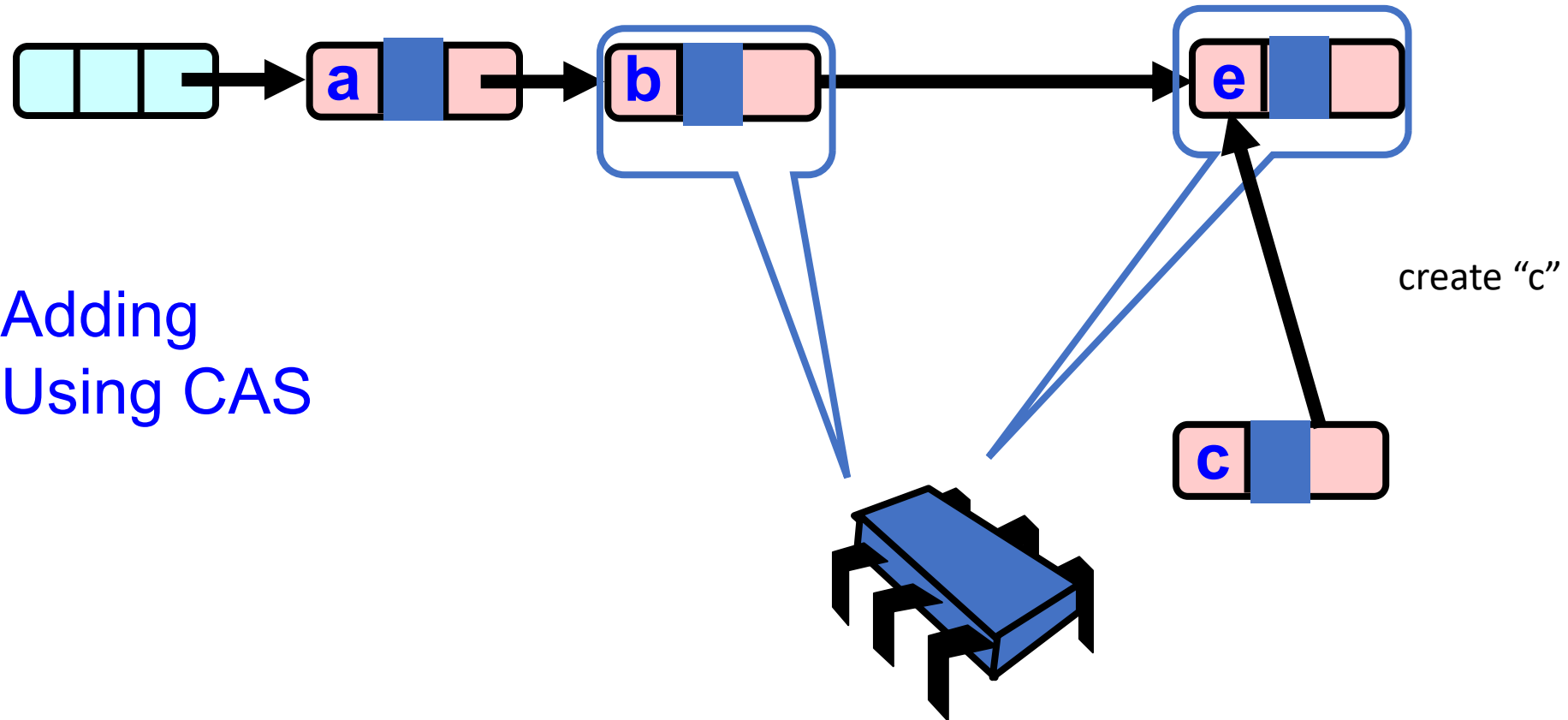
Adding  
Using CAS

# Lock-free Lists

Find the location  
Cache your insertion  
point!

```
b.next == e
```

## Adding Using CAS



# Lock-free Lists

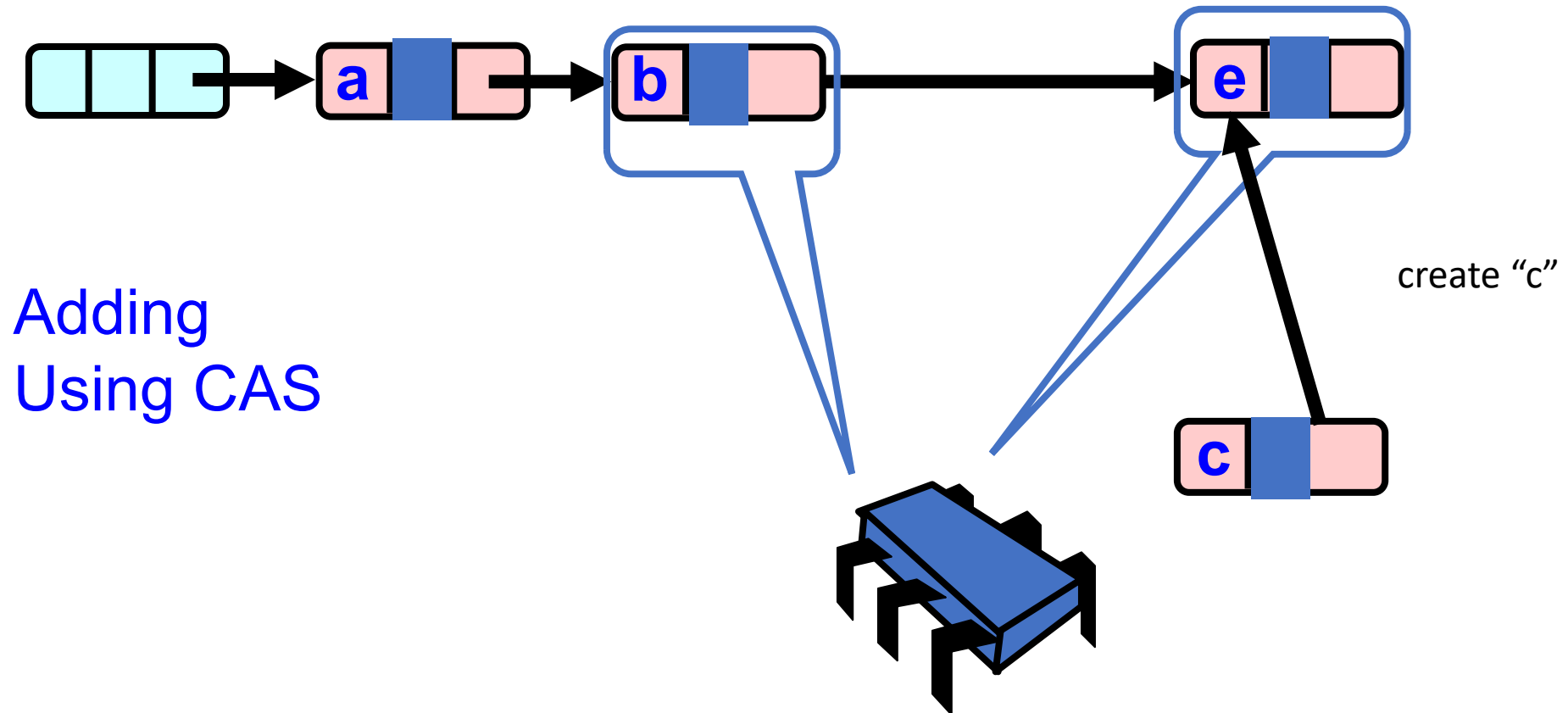
Only insert if your insertion point is valid!

`CAS(b.next, e, c);`

Find the location  
Cache your insertion point!

`b.next == e`

*notion is being abused here: e and c will be node \**



# Lock-free Lists

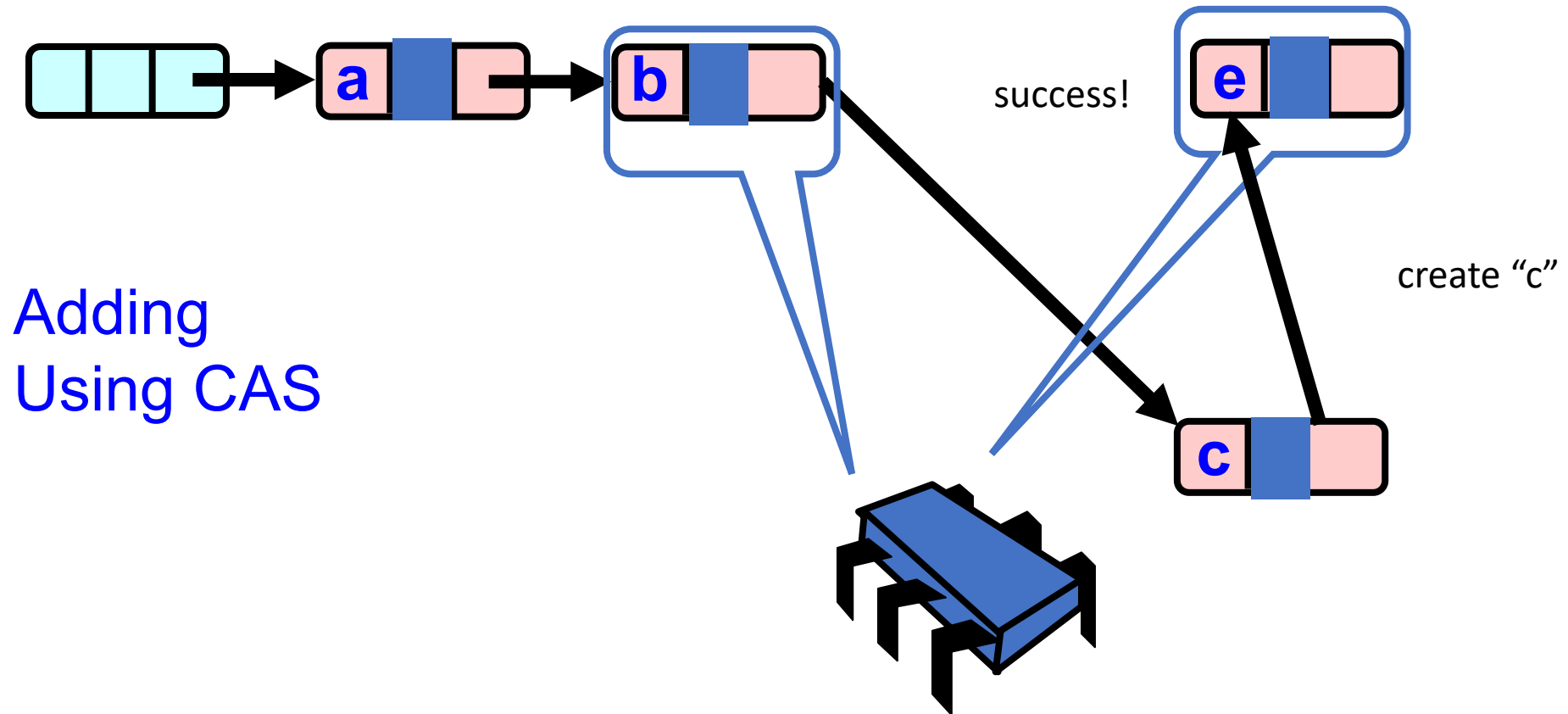
Only insert if your insertion point is valid!

`CAS(b.next, e, c);`

Find the location  
Cache your insertion point!

`b.next == e`

*notion is being abused here: e and c will be node \**



# Lock-free Lists

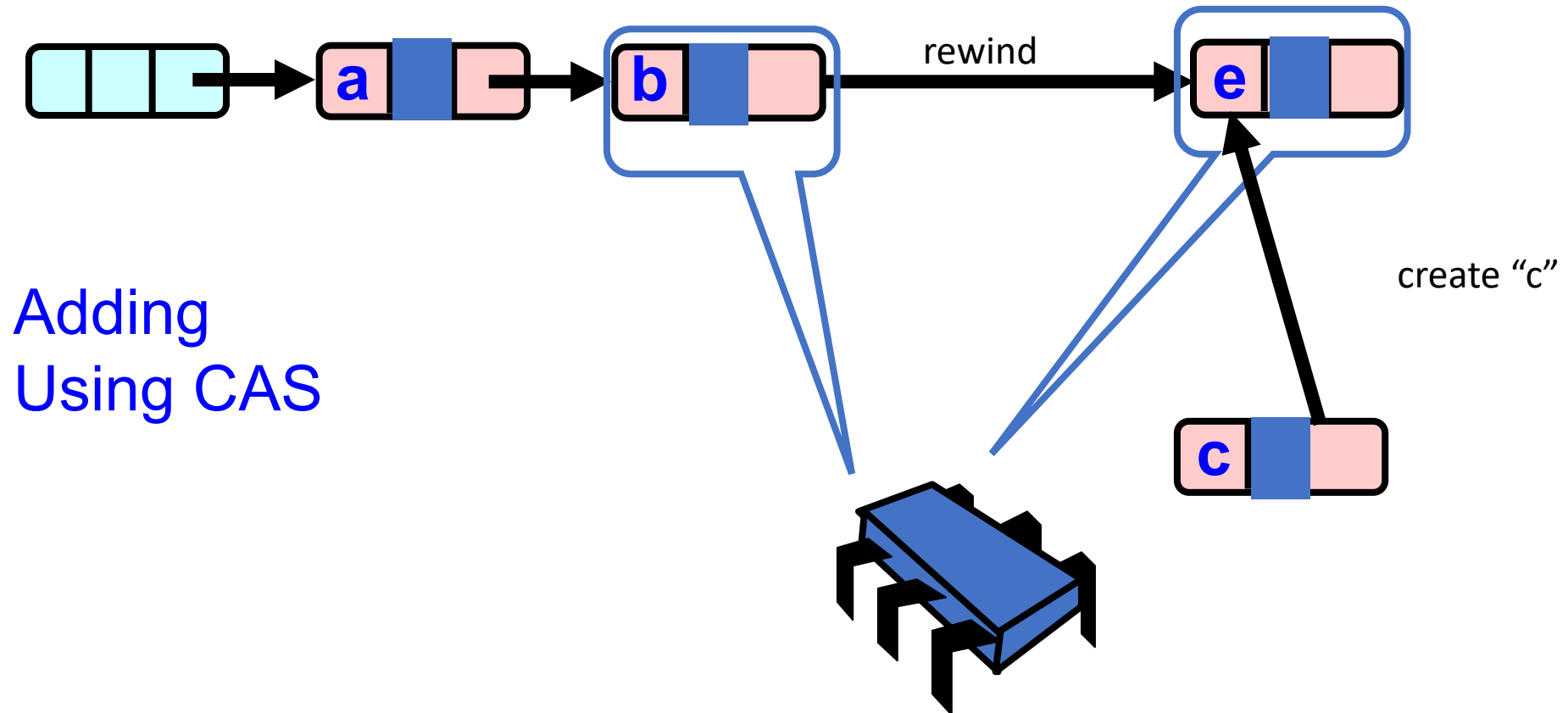
Only insert if your insertion point is valid!

`CAS(b.next, e, c);`

Find the location  
Cache your insertion point!

`b.next == e`

*notion is being abused here: e and c will be node \**



# Lock-free Lists

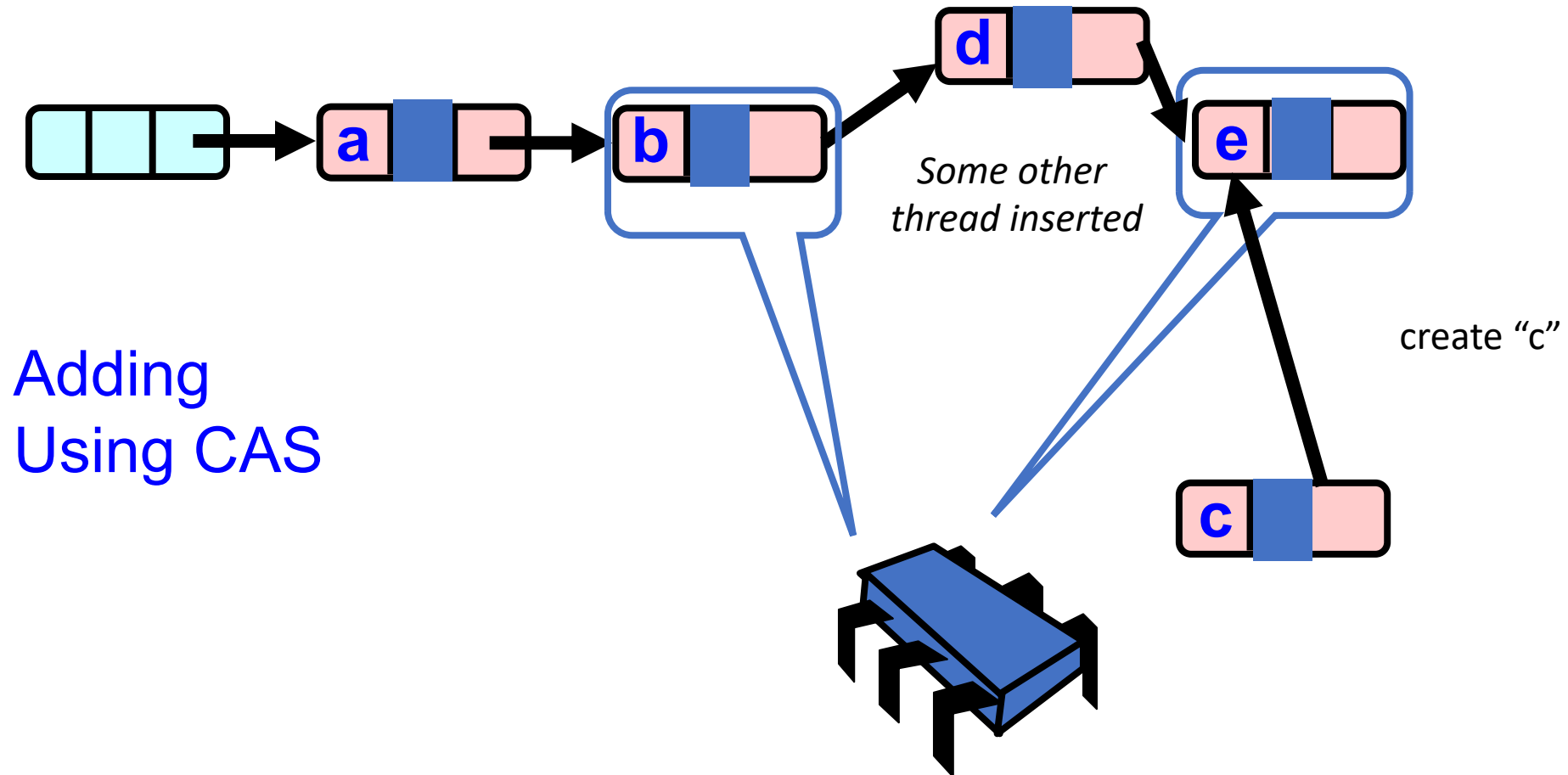
Only insert if your insertion point is valid!

`CAS(b.next, e, c);`

Find the location  
Cache your insertion point!

`b.next == e`

*notion is being abused here: e and c will be node \**





# Lock-free Lists

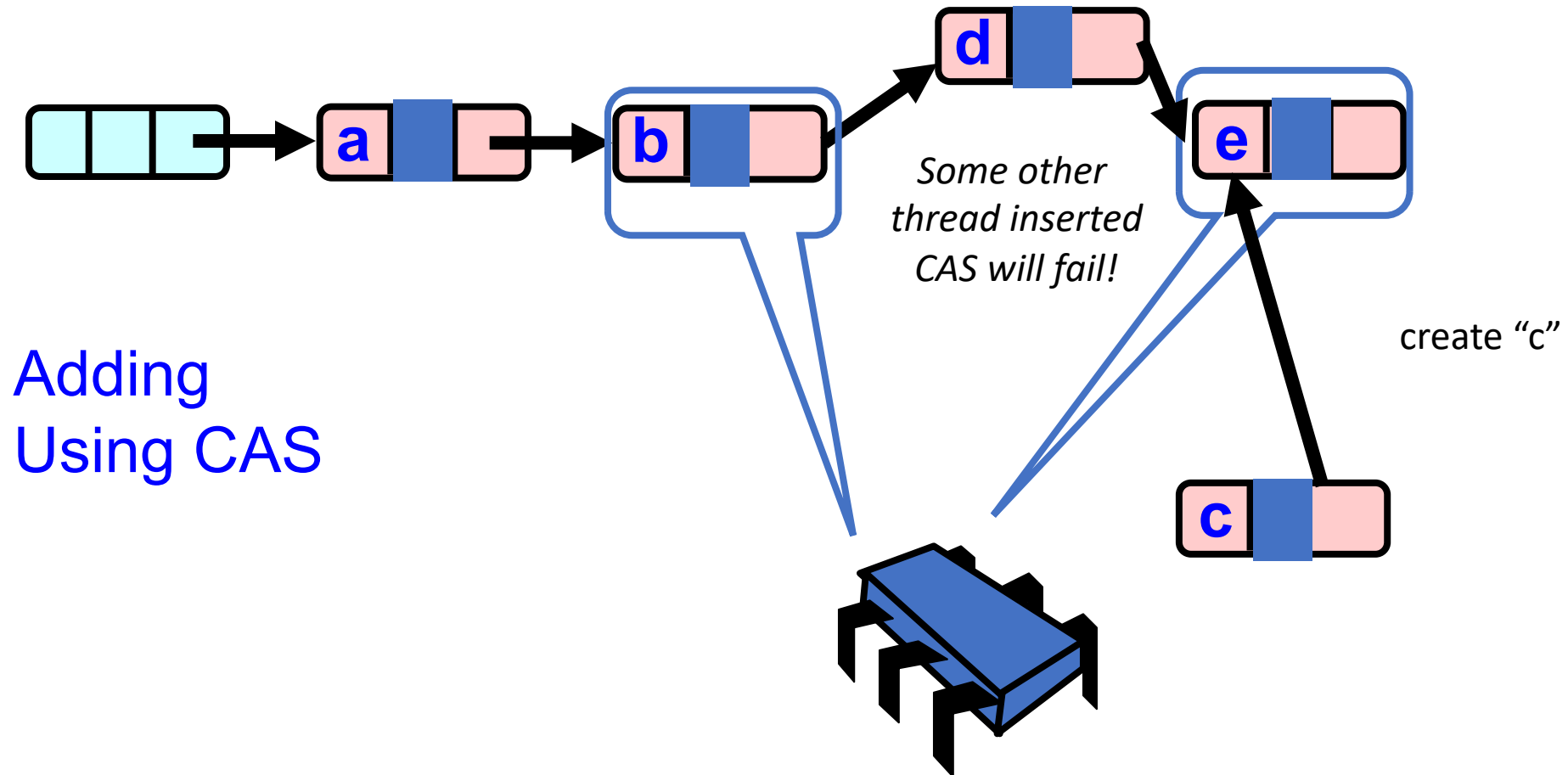
Only insert if your insertion point is valid!

`CAS(b.next, e, c);`

Find the location  
Cache your insertion point!

`b.next == e`

*notion is being abused here: e and c will be node \**



# Lock-free Lists

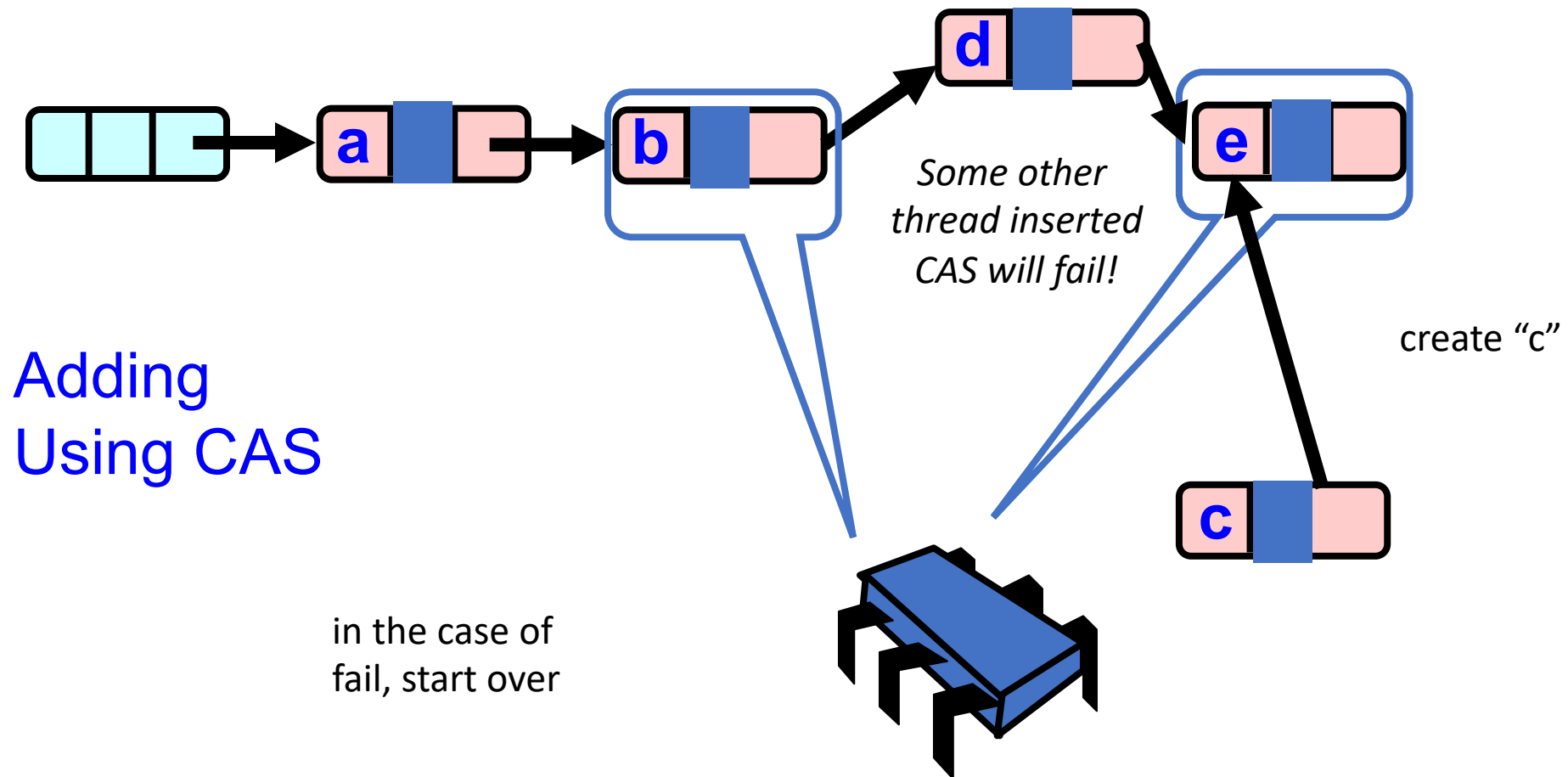
Only insert if your insertion point is valid!

`CAS(b.next, e, c);`

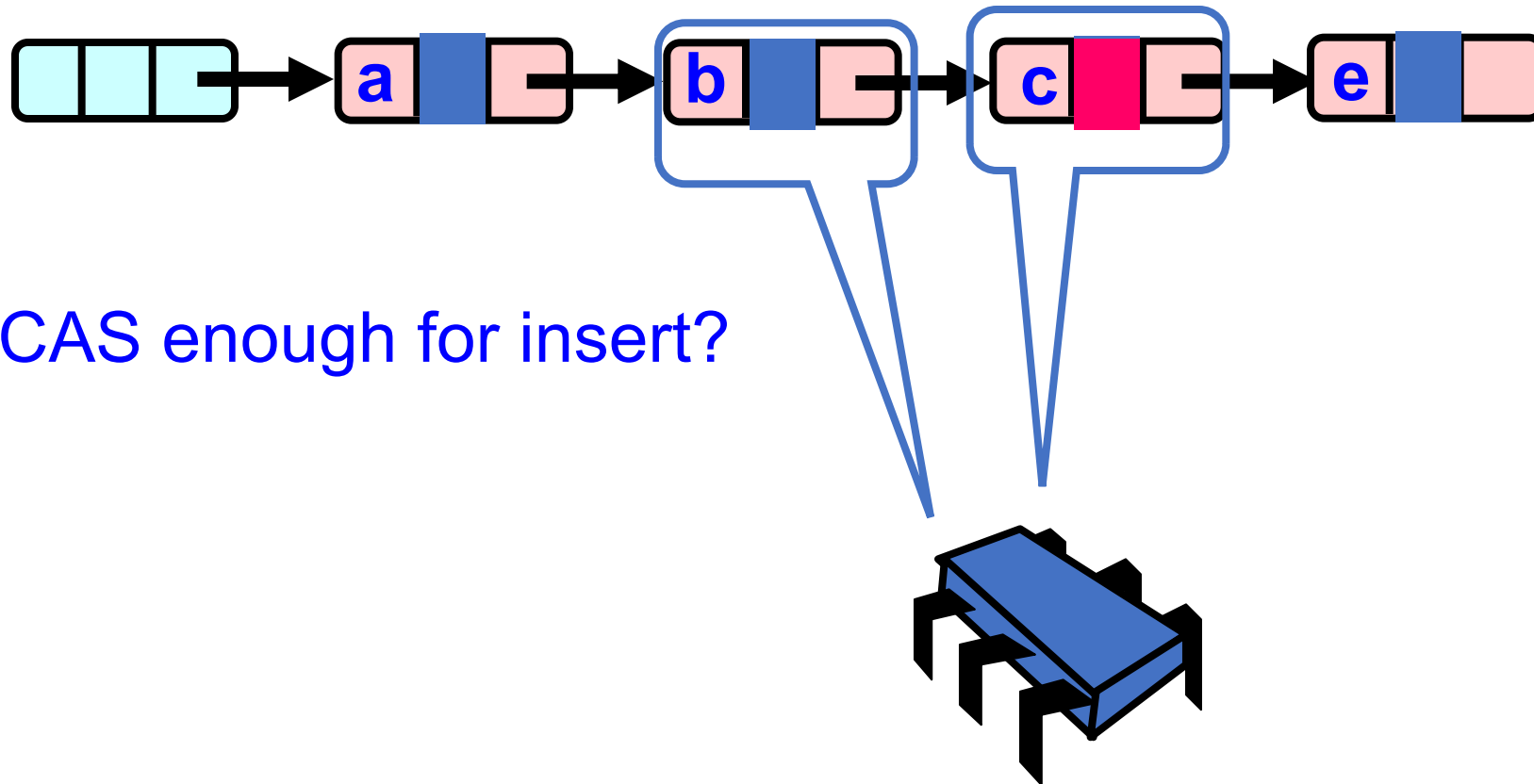
Find the location  
Cache your insertion point!

`b.next == e`

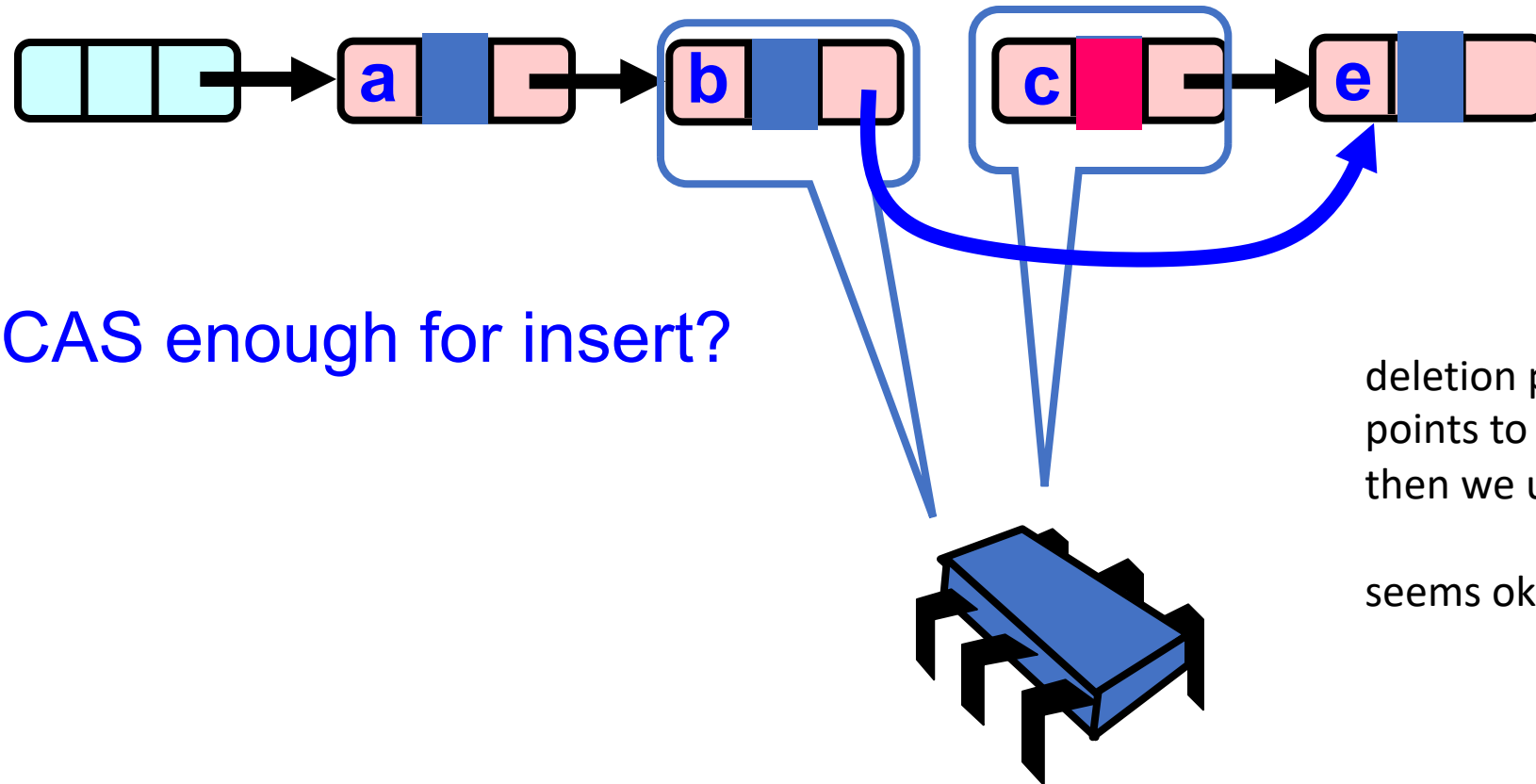
*notion is being abused here: e and c will be node \**



# Lock-free Lists



# Lock-free Lists



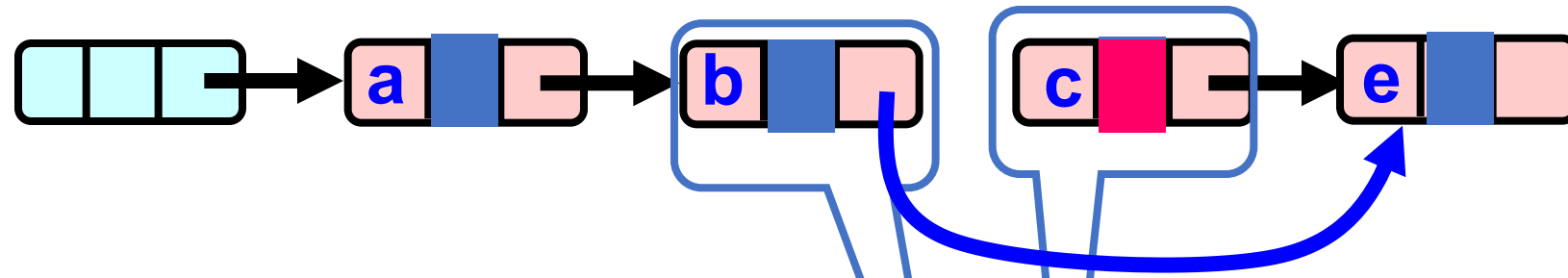
CAS enough for insert?

deletion point requires b  
points to c. If that is valid  
then we update to e.

seems okay...

# Lock-free Lists

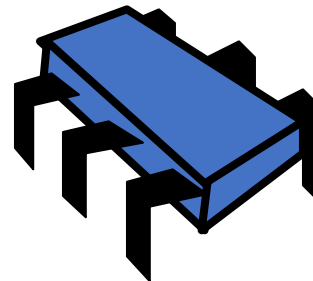
*ensures that nobody has inserted a node between b and c*



CAS enough for insert?

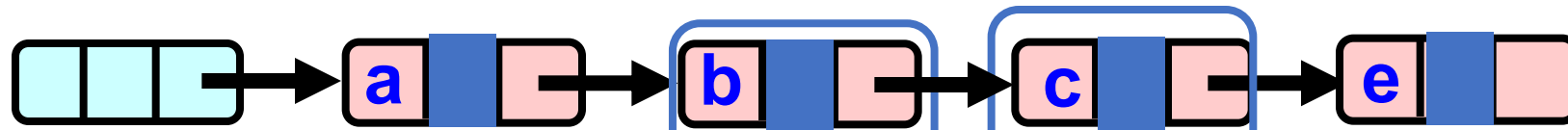
deletion point requires b points to c. If that is valid then we update to e.

seems okay...

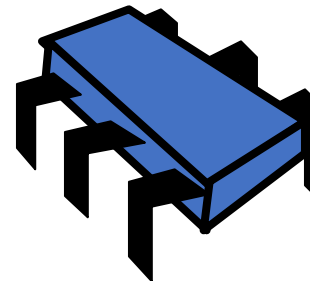


# Lock-free Lists

*Rewind*

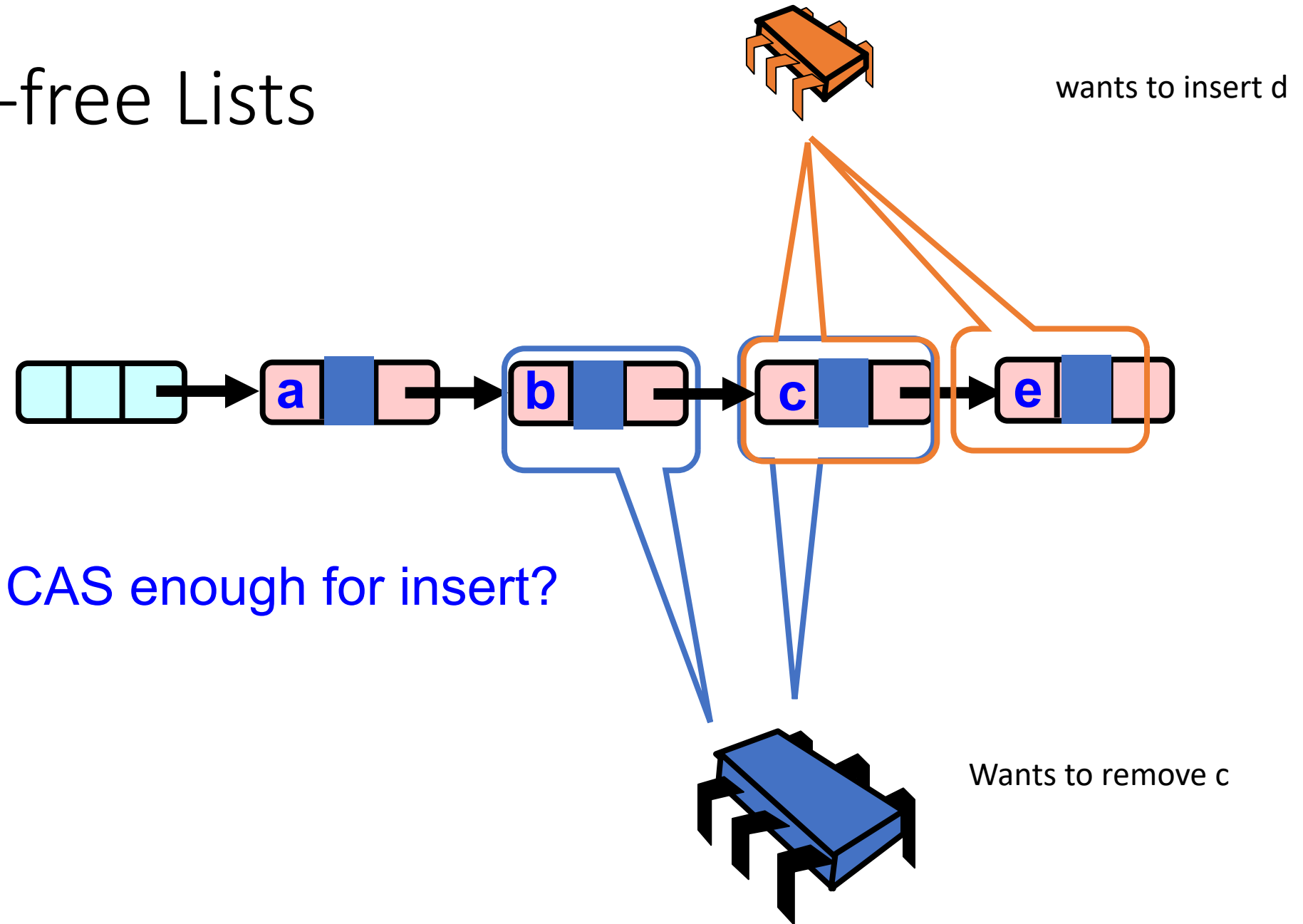


CAS enough for insert?

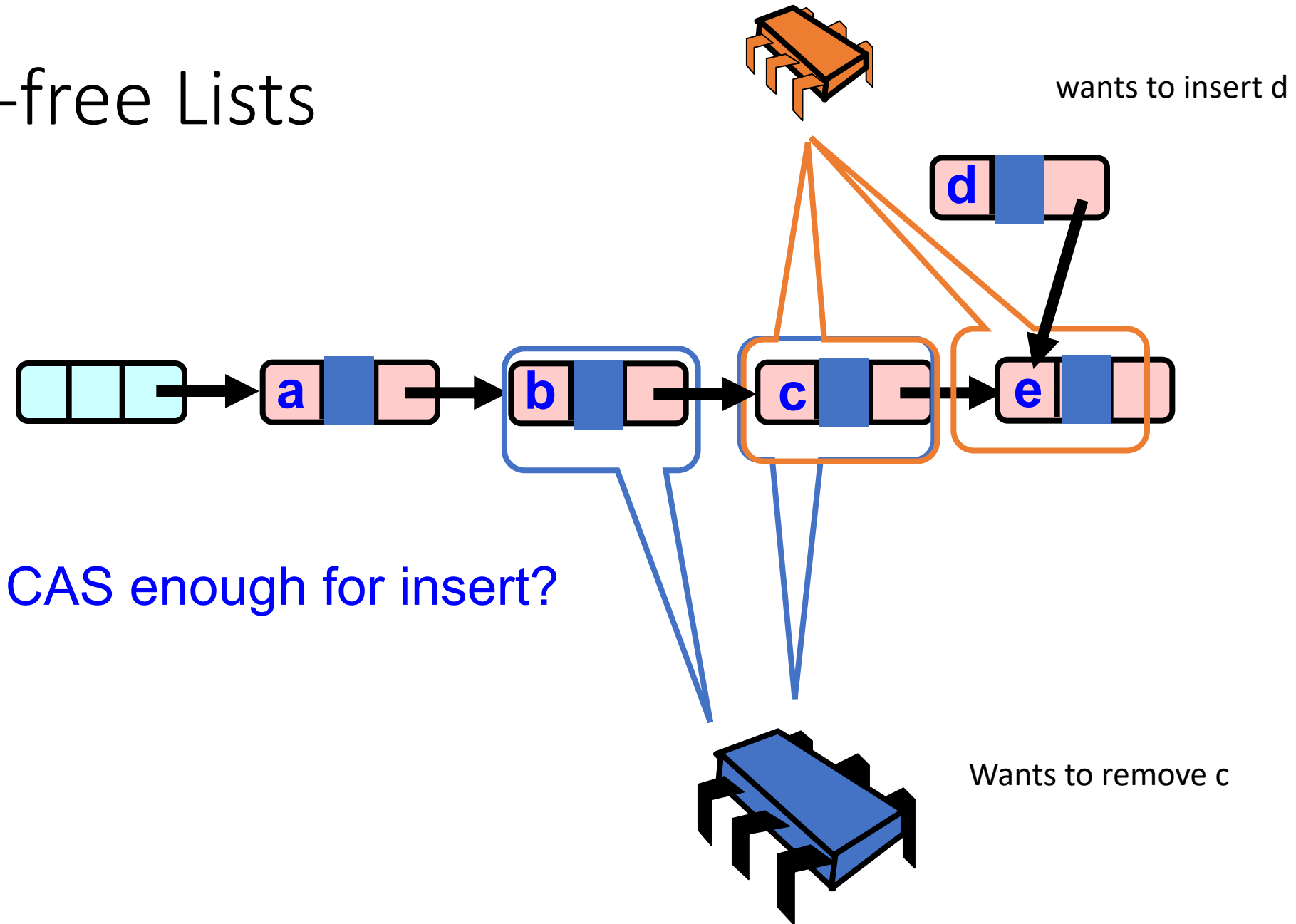


Wants to remove c

# Lock-free Lists

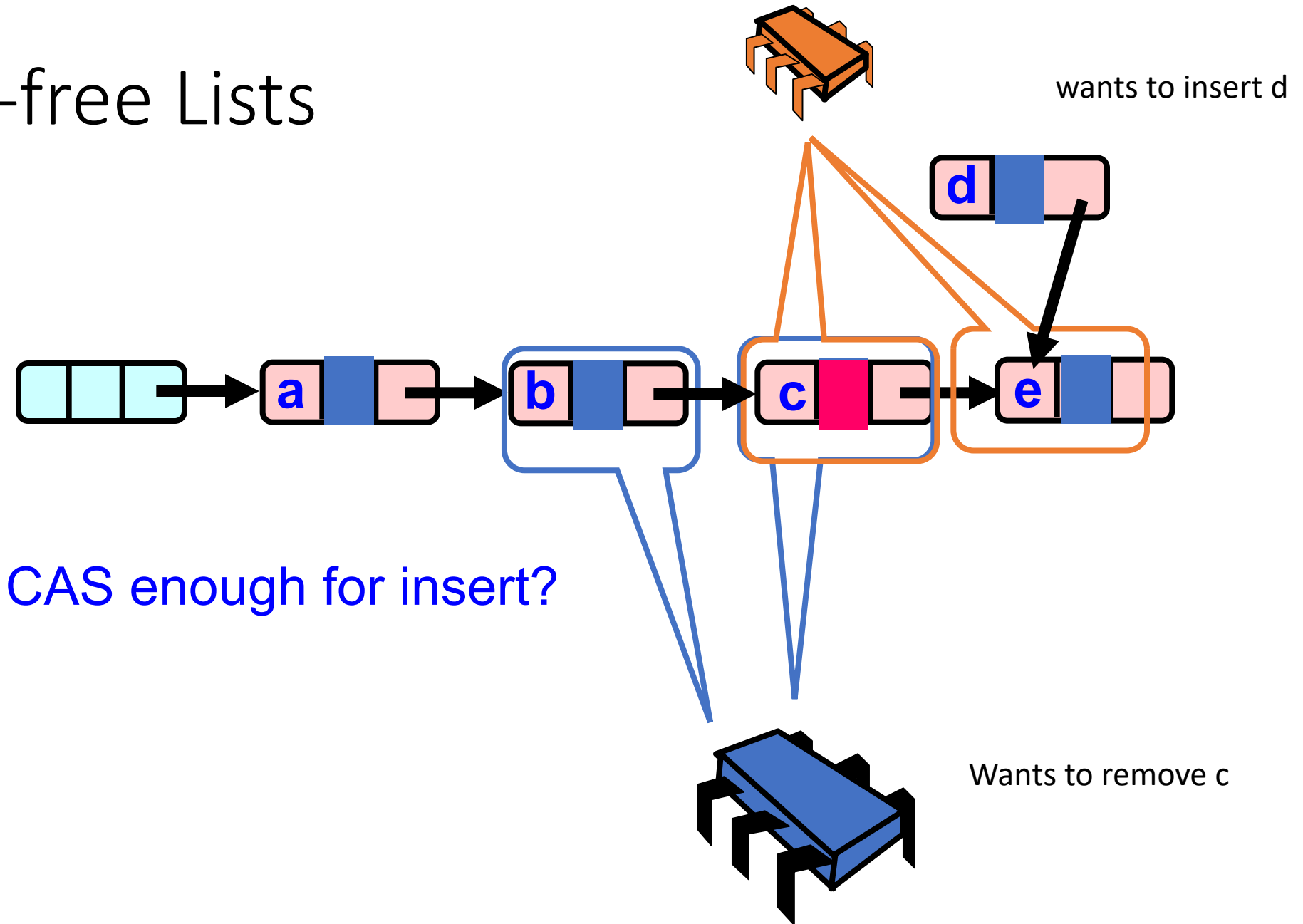


# Lock-free Lists

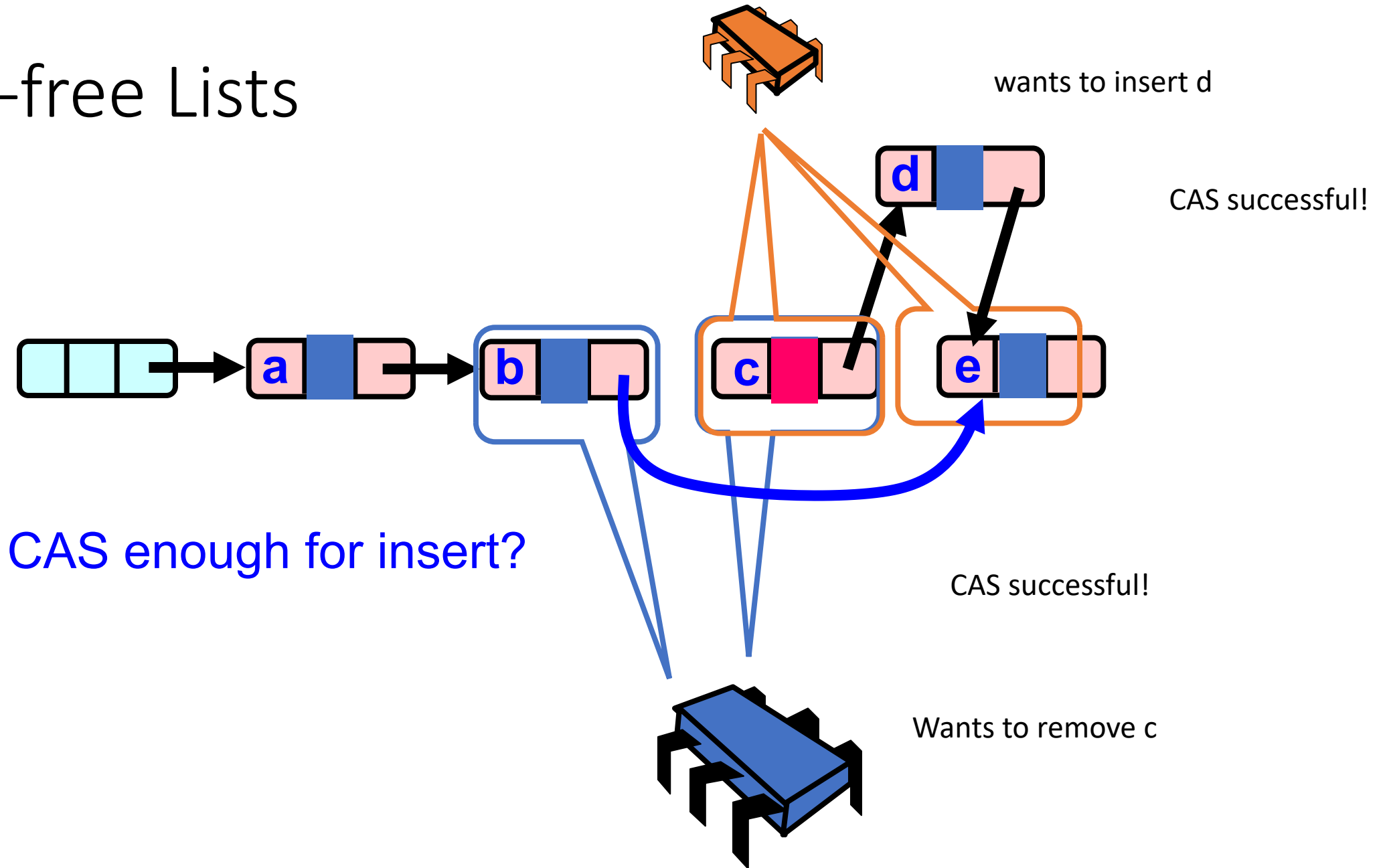




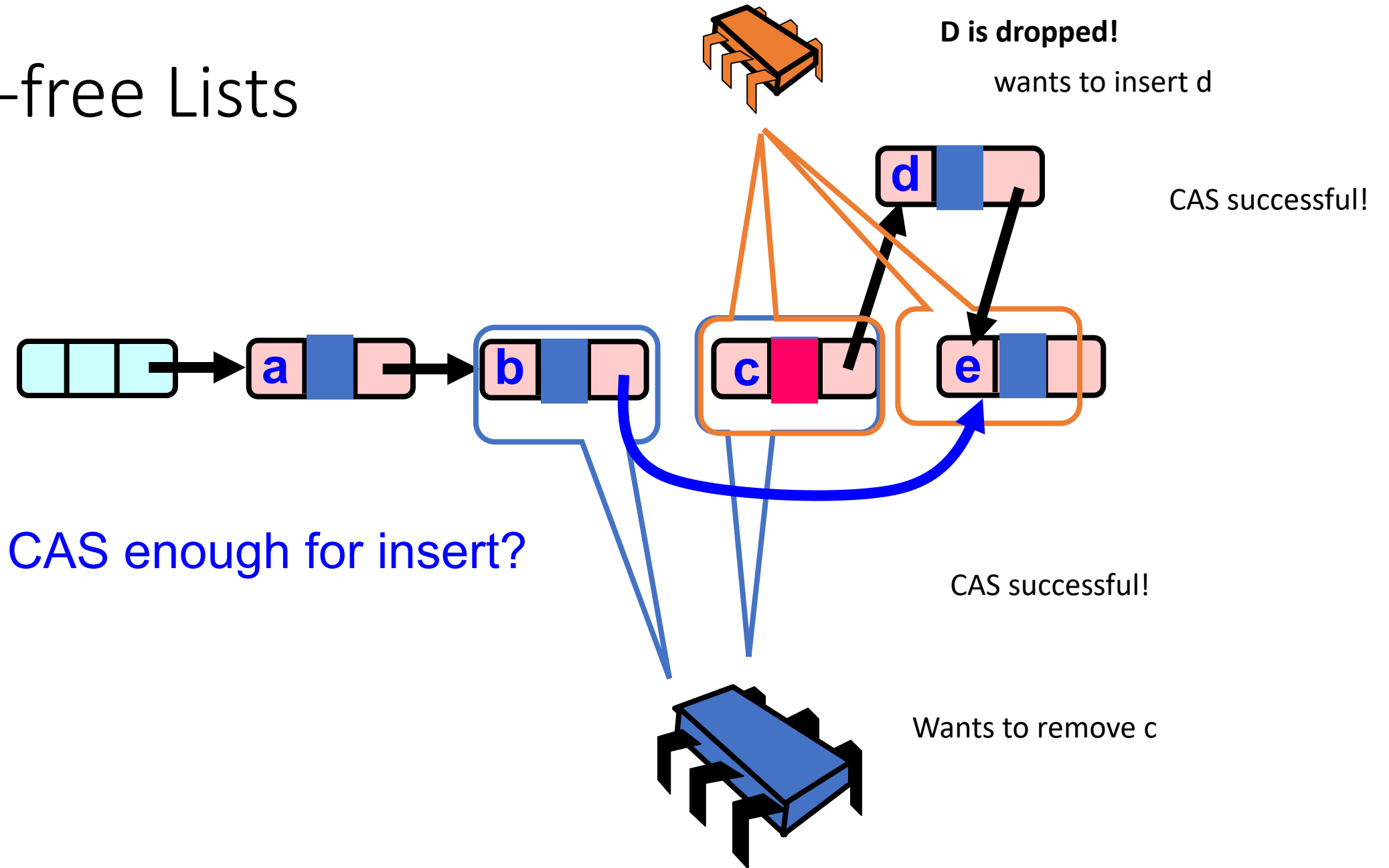
# Lock-free Lists



# Lock-free Lists



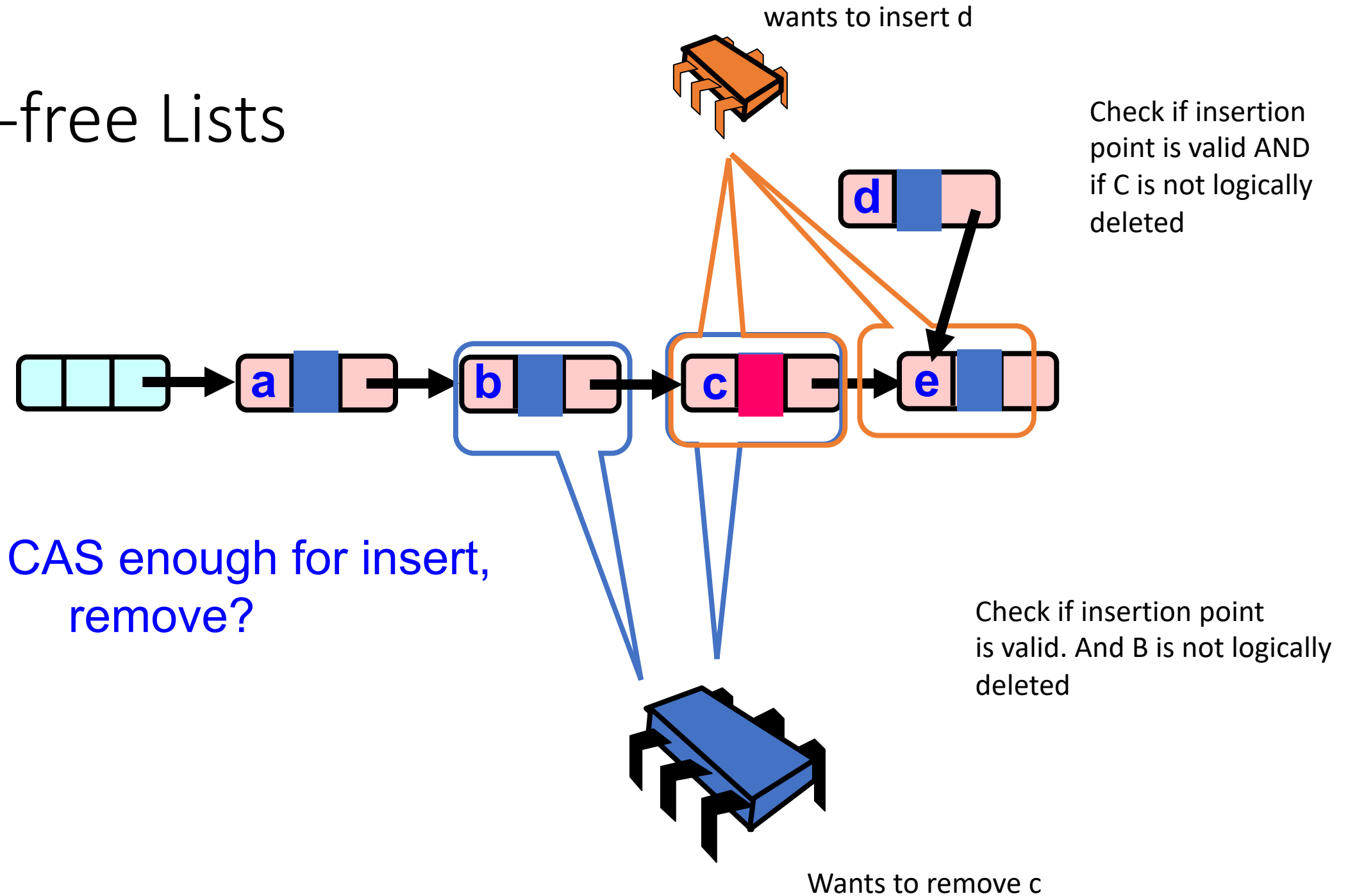
# Lock-free Lists



# Solution

- Use AtomicMarkableReference
- Atomic CAS that checks not only the address, but also a bit.
- We can say: update pointer if  
the insertion point is valid AND  
the node has not been logically removed.

# Lock-free Lists



# Marking a Node

- **AtomicMarkableReference** class
  - Java.util.concurrent.atomic package
  - But we're using a better™ language (C++)



# This stuff is tricky

- Focus on understanding the concepts:
  - Locks are easiest, but can impede performance
  - Fine-grained locks are better, but more difficult
  - Optimistic concurrency can take you far. CAS is your friend
- When reasoning about correctness:
  - You have to consider all combination of adds/removes.
  - Thread sanitizer will help, but not as much as in mutexes. Other research tools can help.