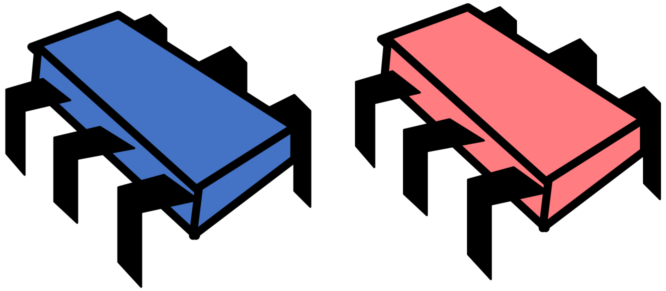
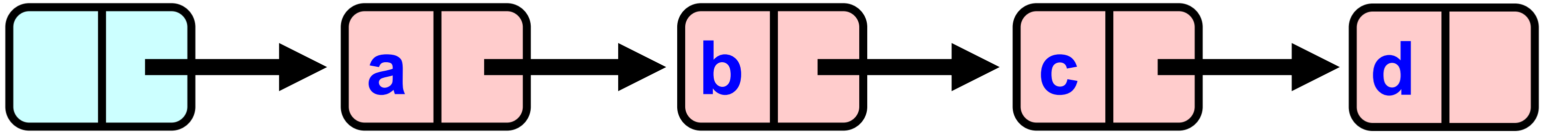
CSE113: Parallel Programming

March 13, 2024

- **Topics**:
  - Concurrent general set





Announcements

Last day of class!

Announcements

Grading

- HW 2 is graded. Please let us know ASAP if there are any issues.
  - Let us know by Thursday (tomorrow) if there are issues
  - Auto grading is difficult here, so don’t hesitate to reach out! Make a private post on piazza or see a TA
  - Trying to have HW 3 graded by the end of the week

Announcements

- HW 5 was released last Friday
  - Due on the day of the final
  - Last day to turn it in is the 21
  - It could be useful to work on for the final
  - ***You should have everything you need to complete the homework***

Announcements

Final

- Allowed 3 pages of notes, front and back
- Similar to the midterm (50% longer)
- Time: Monday March 18: 7:30 – 10:30 PM

Announcements

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

Both bad things and good things!

Announcements

Video games for parallel programming education

- Joint project with the CM department
- Performing a user-study
- Max 160 minutes, $30 cash
- Play a puzzle video game with parallel programming concepts
- Located in the UC Santa Cruz Silicon Valley Campus
- Includes a tour of the campus, meet & greet with the CM researchers there
- I’ll post more information in canvas

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

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

language
C++11 (sequential consistency)

	L	S
L	NO	NO
S	NO	NO

C++		ISA
`x.store(1);`	→	`store(x,1);` `fence;`
or		
`z = x.load()`	→	`fence;` `%z = load(x);`

target machine
TSO (x86)

	L	S
L	NO	different address
S	NO	No

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

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

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.

General concurrent set

Set Interface

- Unordered collection of items
- No duplicates

- We will implement this as a sorted linked list

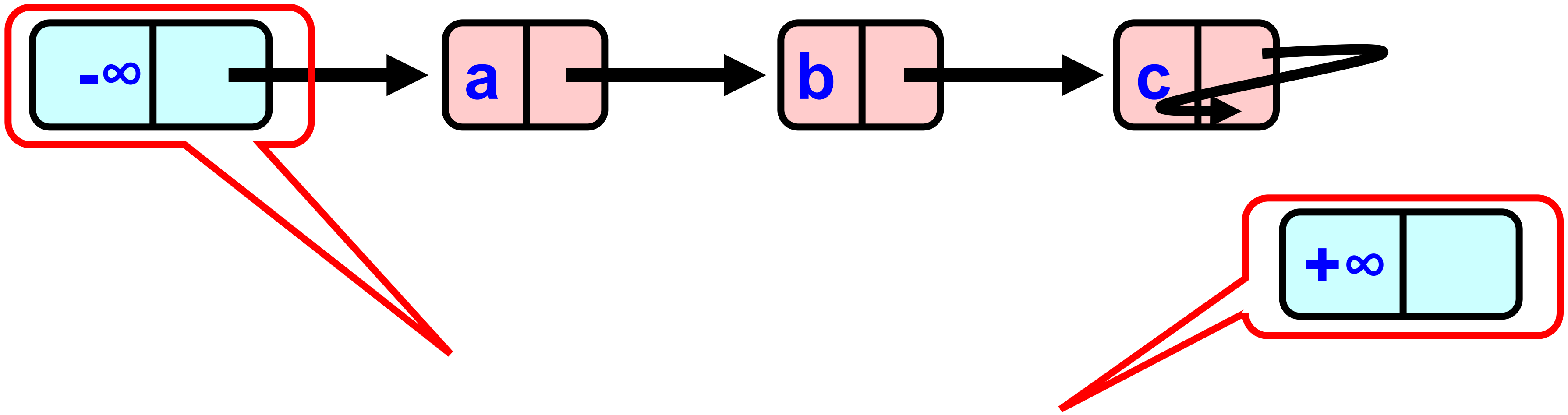
Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - **`add(x)`** put **`x`** in set
  - **`remove(x)`** take **`x`** out of set
  - **`contains(x)`** tests if **`x`** in set

List Node

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

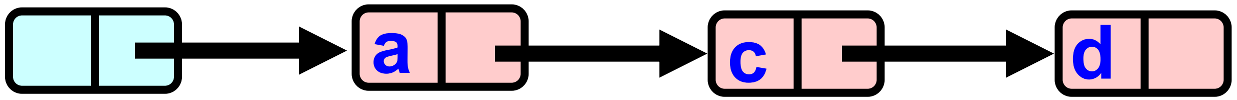
The List-Based Set



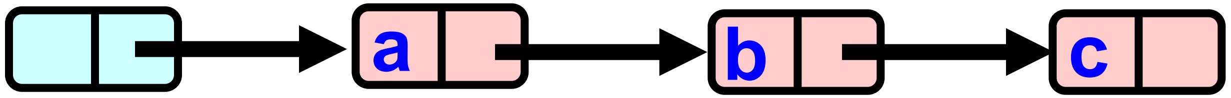
Sorted with Sentinel nodes
(min & max possible keys)

Sequential List Based Set

add(b)

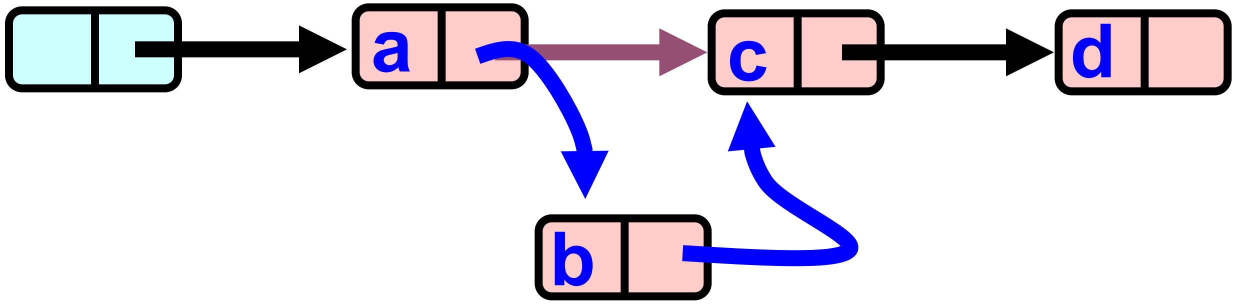


remove(b)

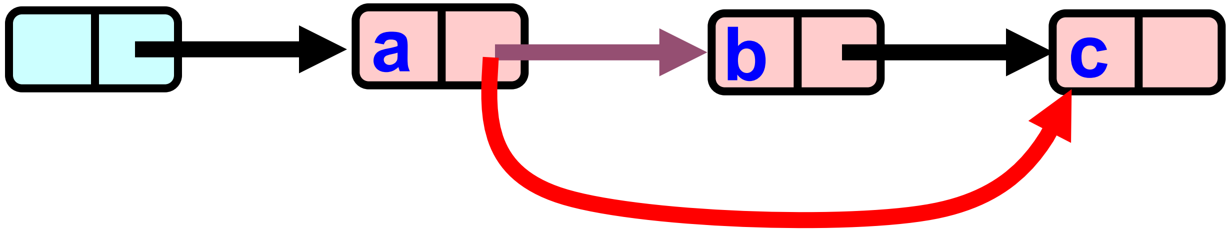


Sequential List Based Set

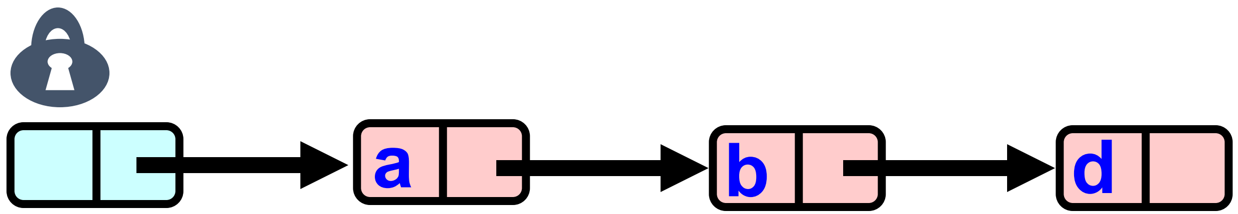
add(b)



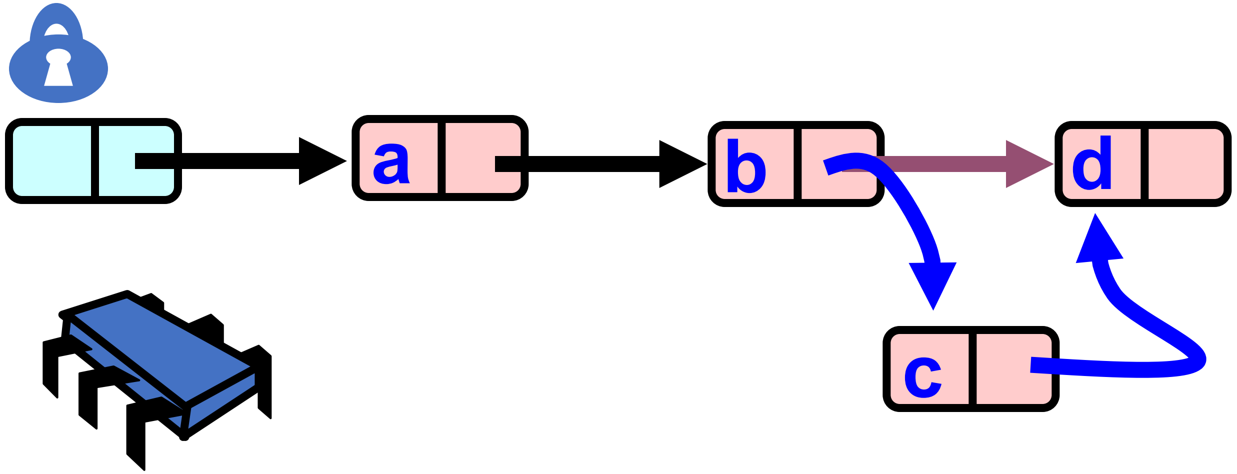
remove(b)



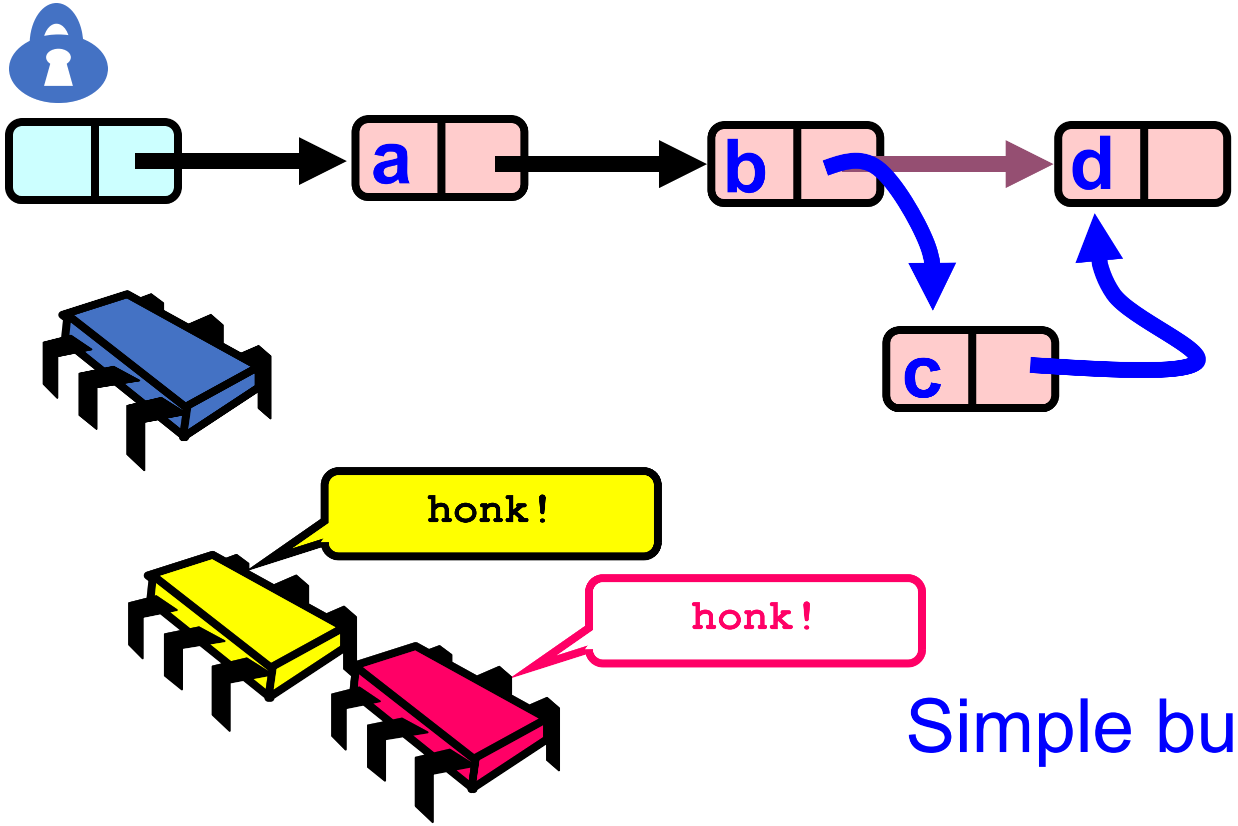
Coarse-Grained Locking



Coarse-Grained Locking



Coarse-Grained Locking

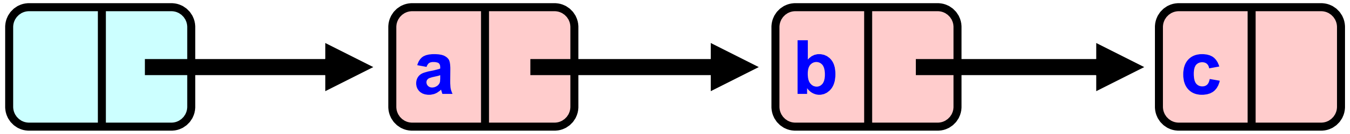


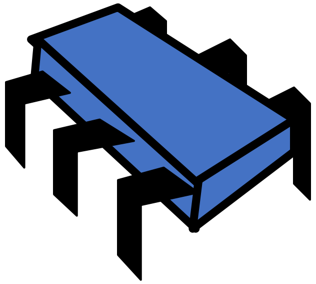
Simple but inefficient!

Fine-grained Locking

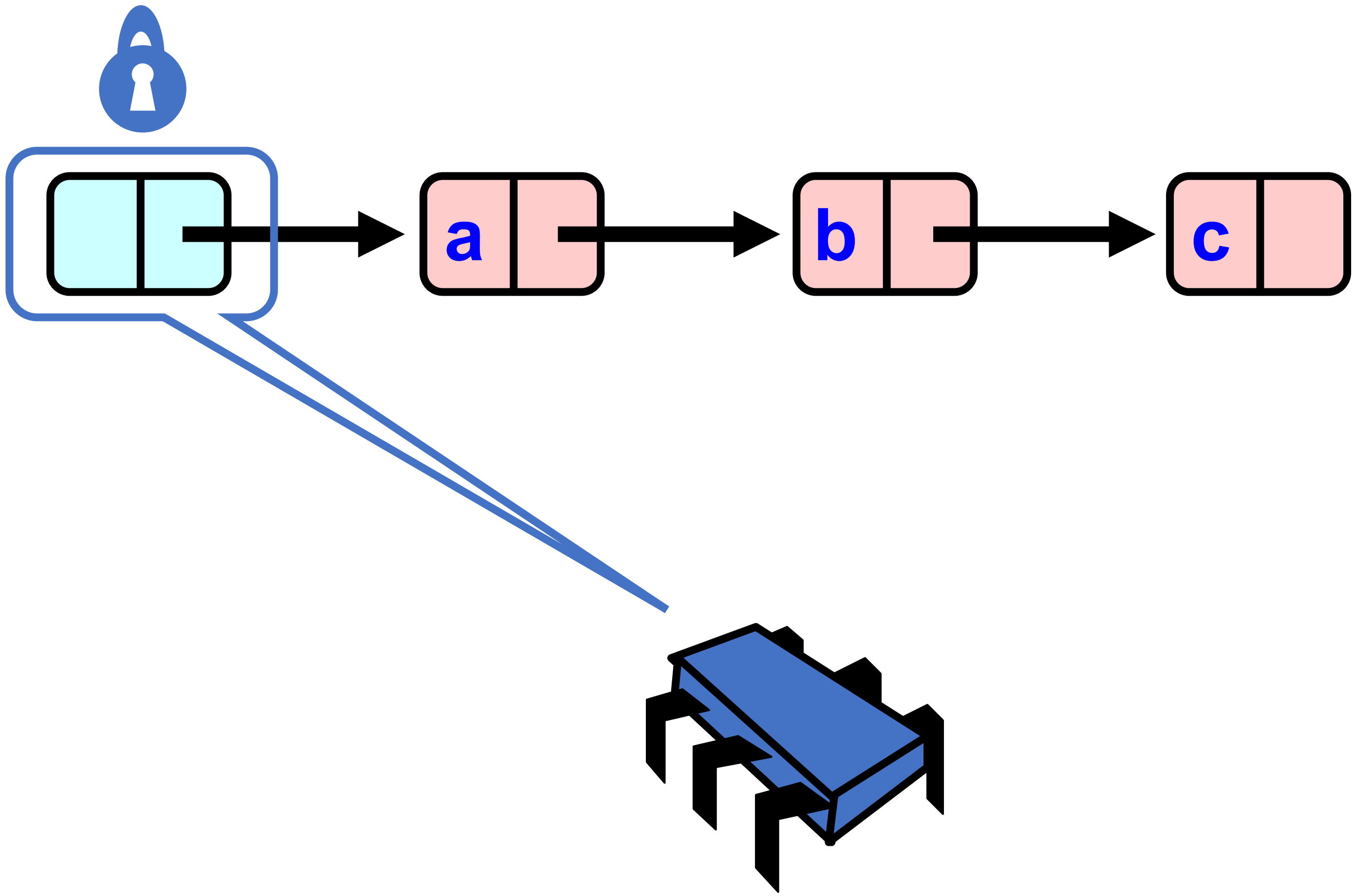
- Requires **careful** thought
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand locking

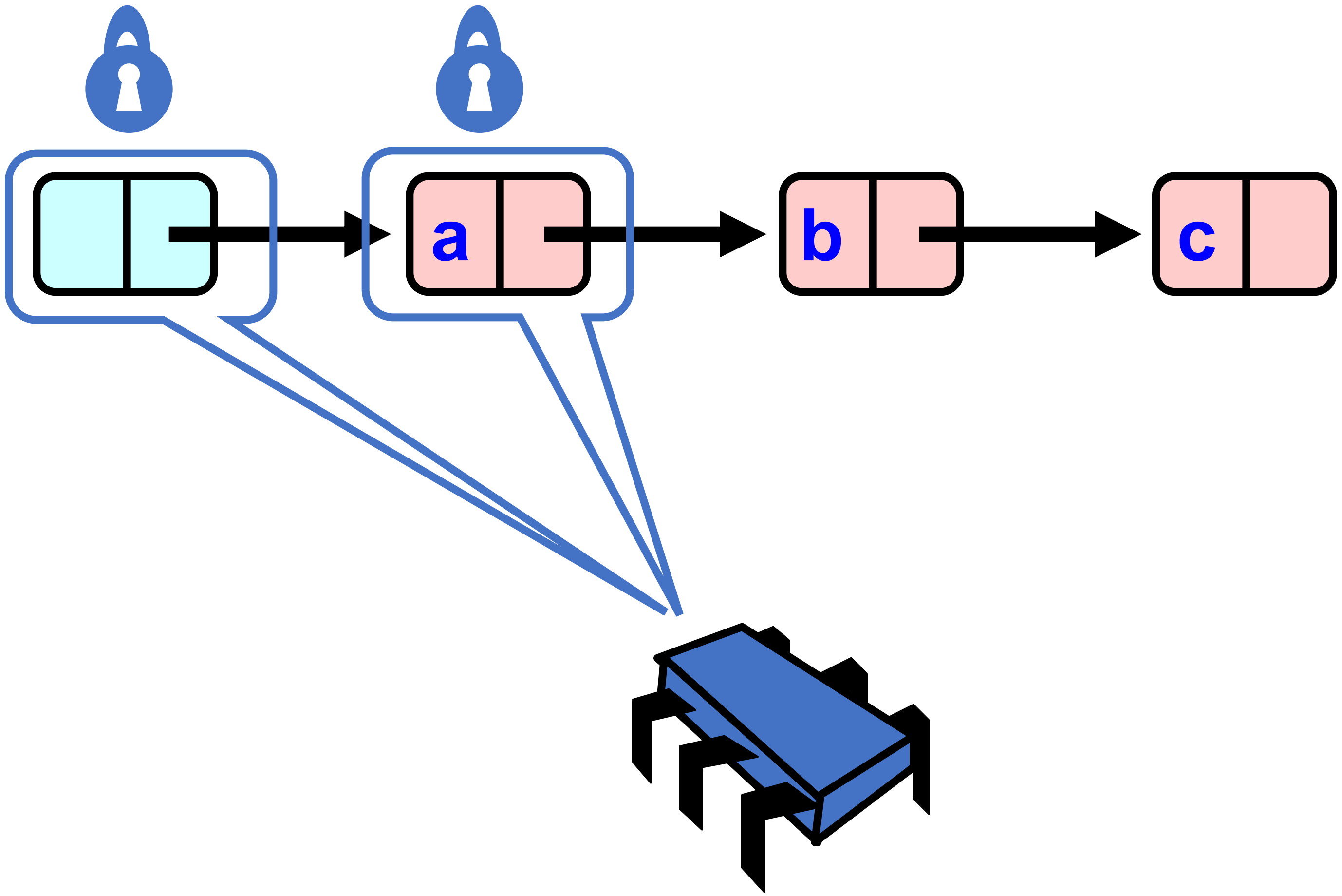




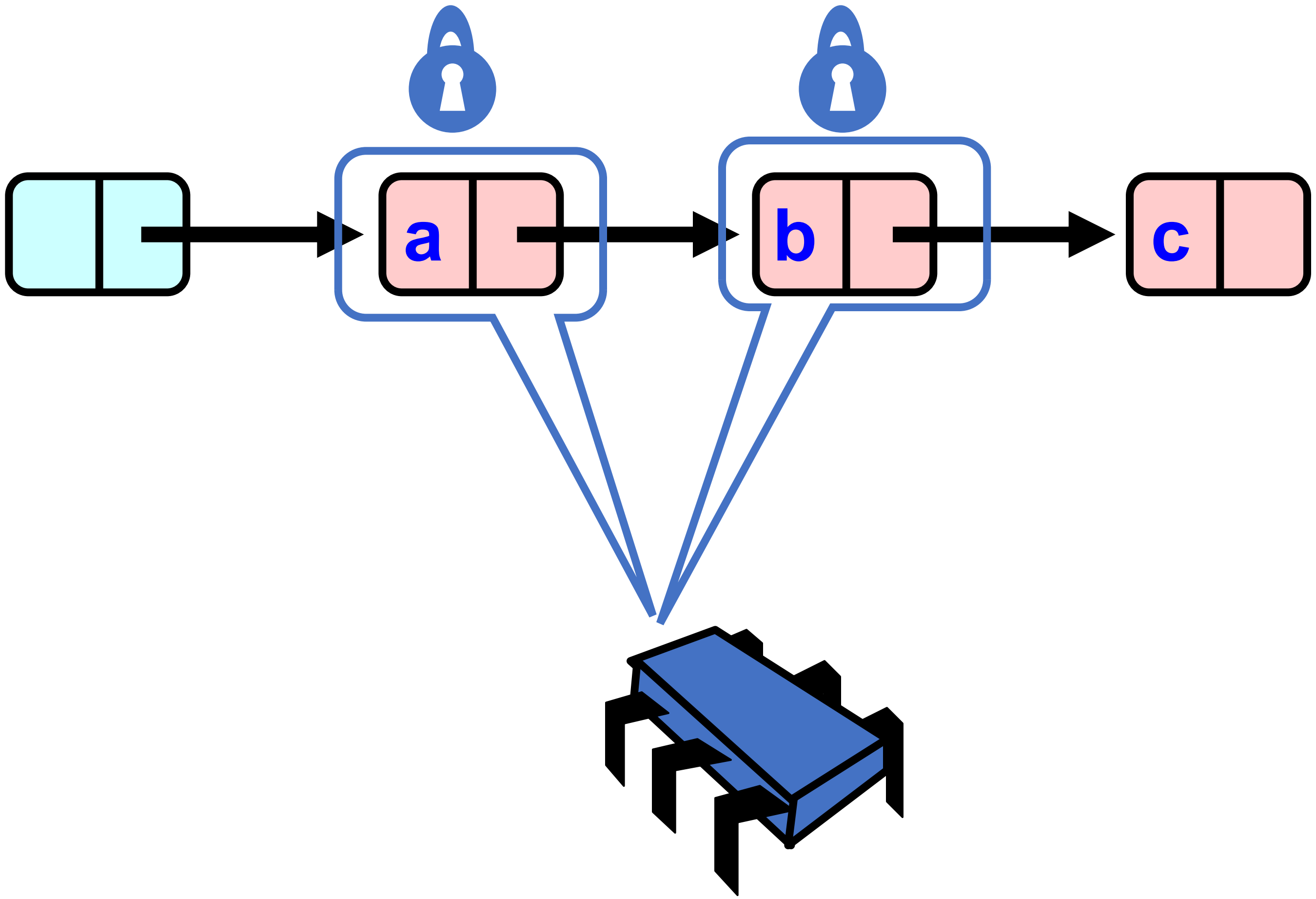
Hand-over-Hand locking



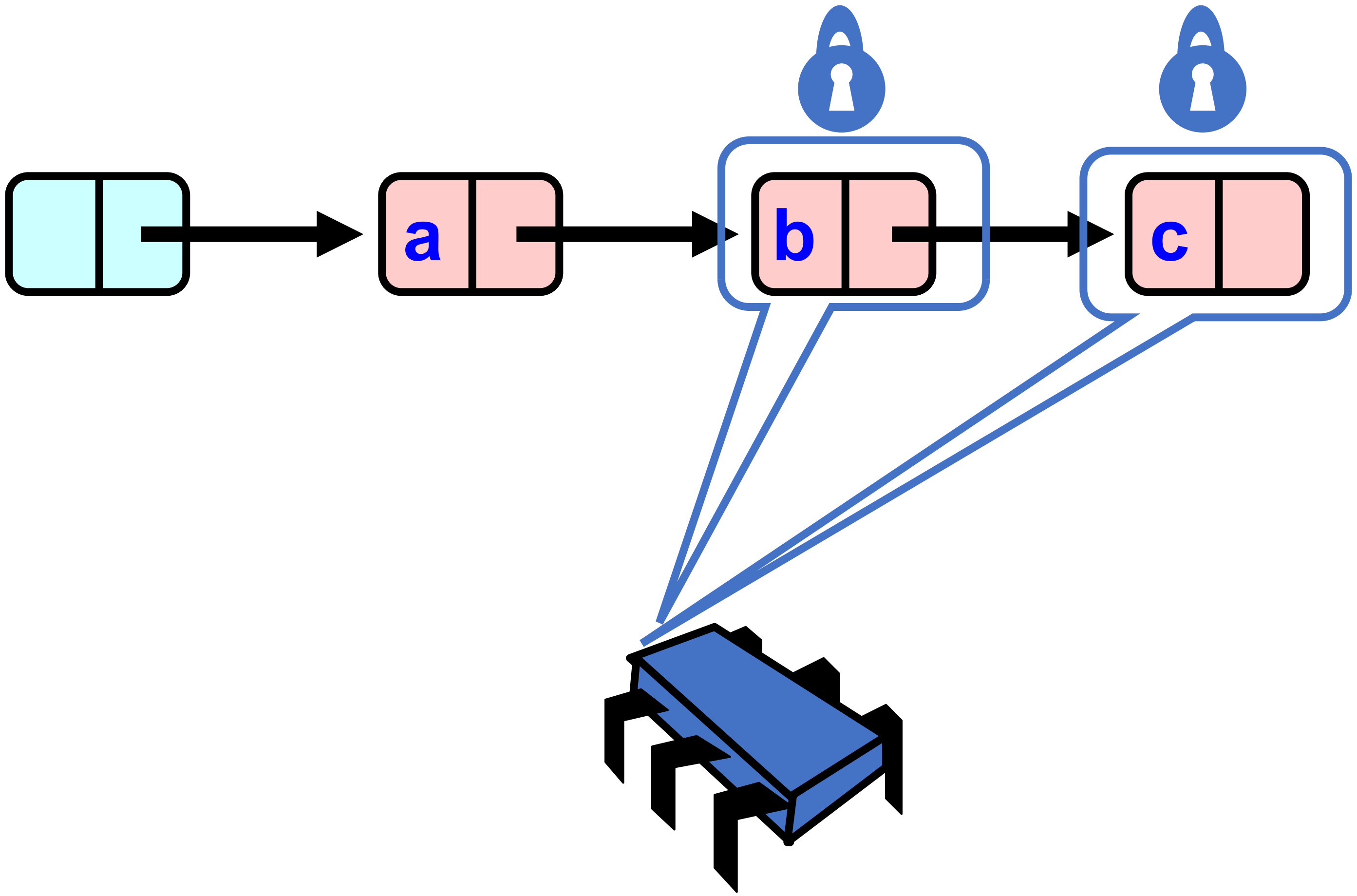
Hand-over-Hand locking



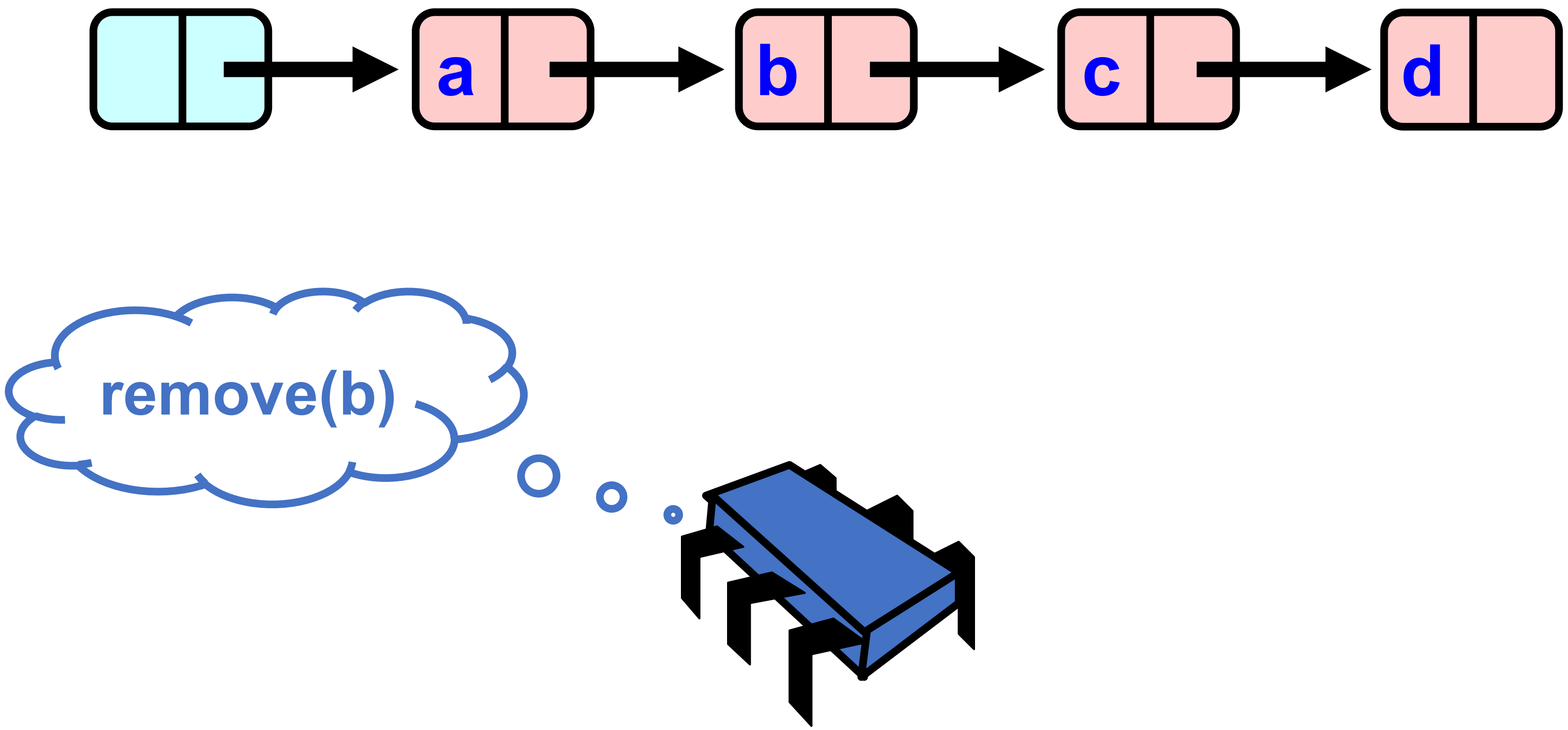
Hand-over-Hand locking



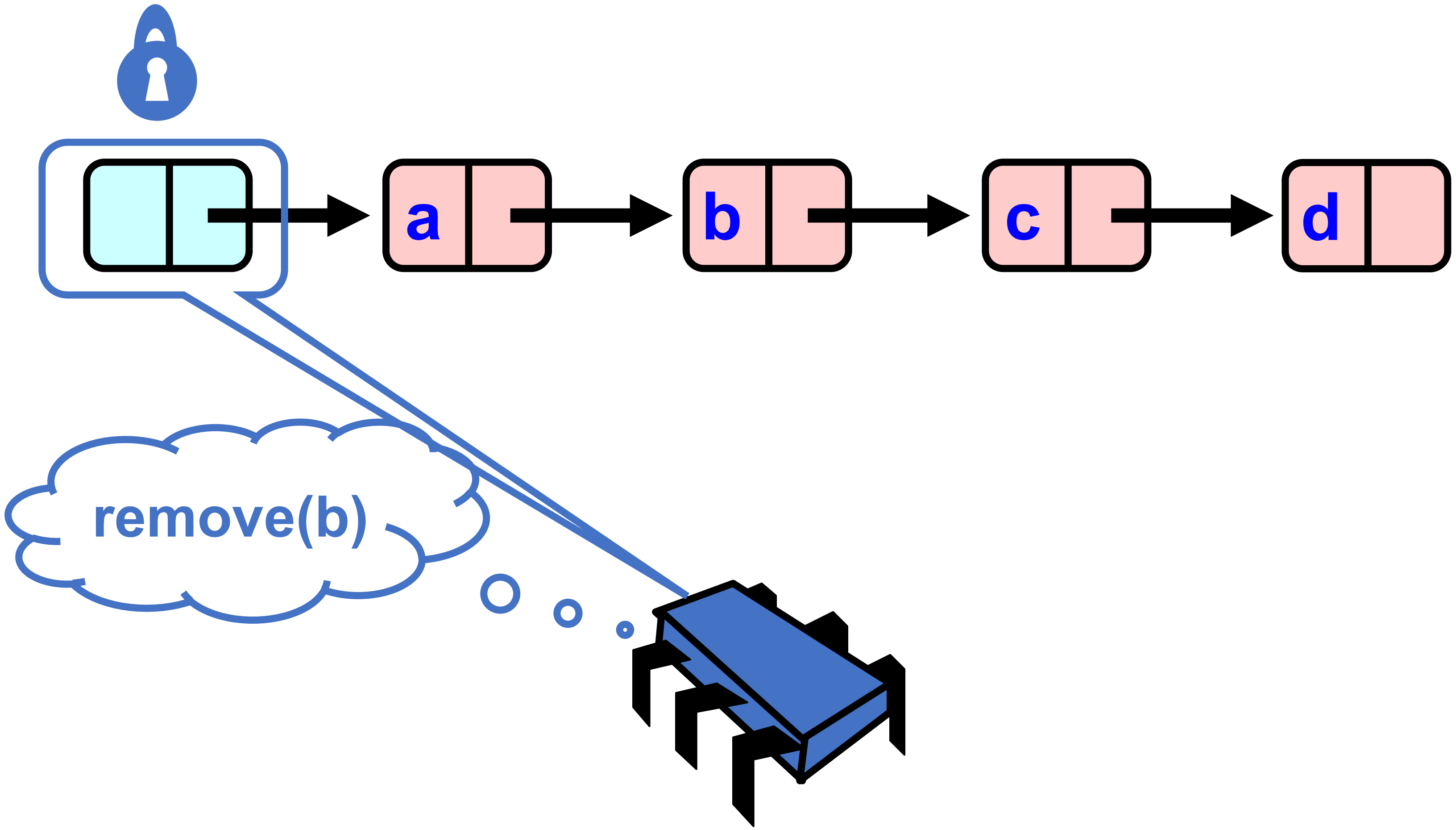
Hand-over-Hand locking



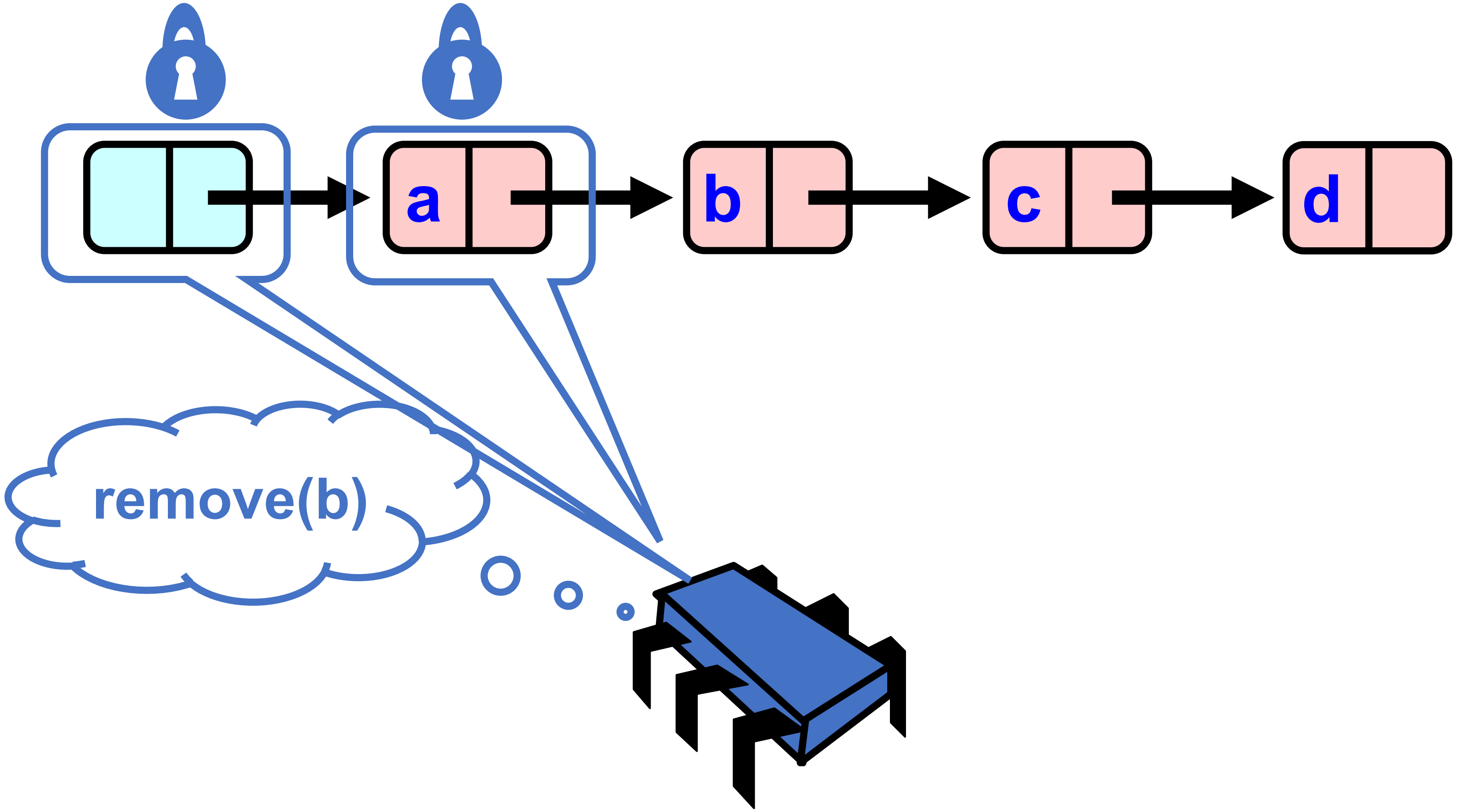
Removing a Node



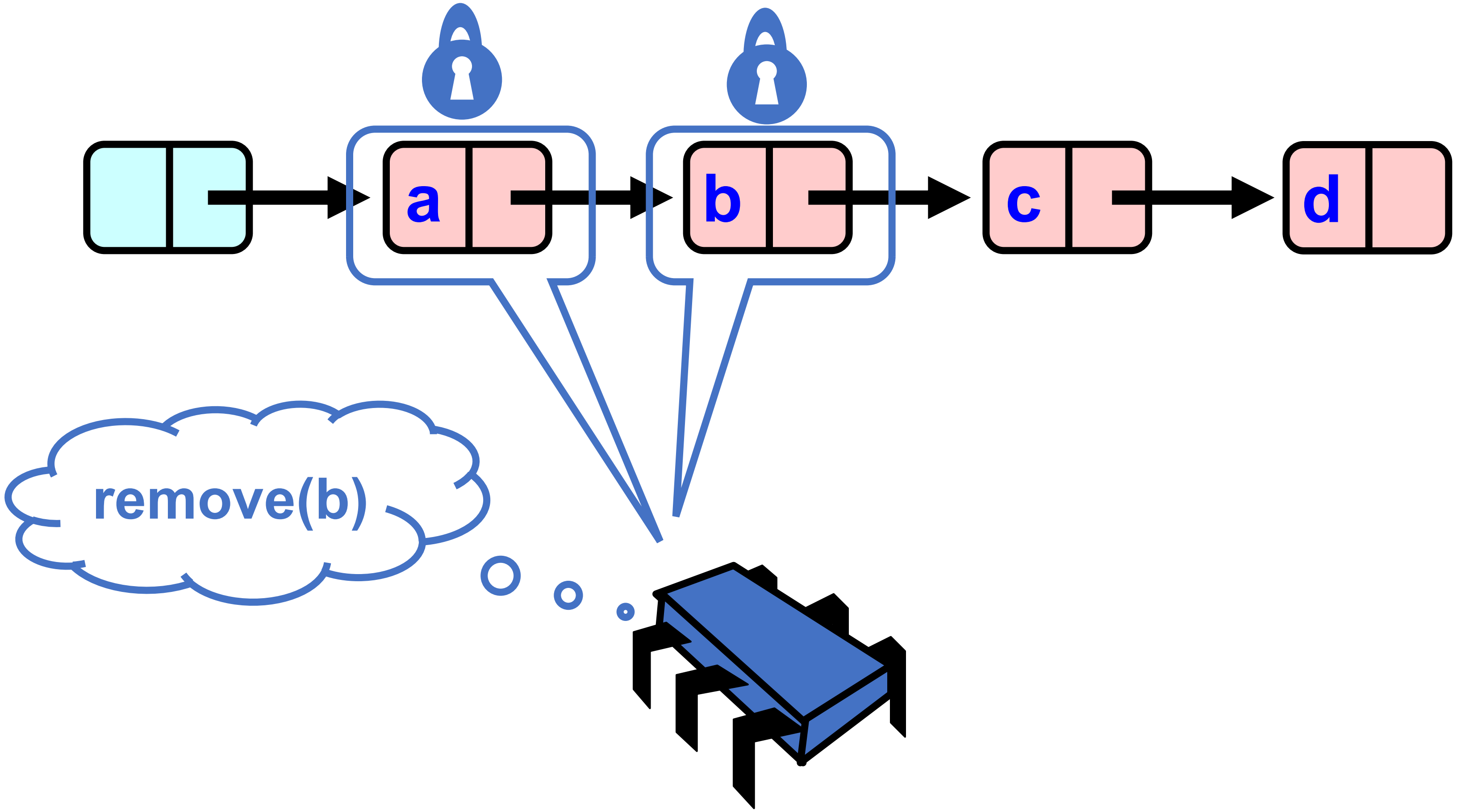
Removing a Node



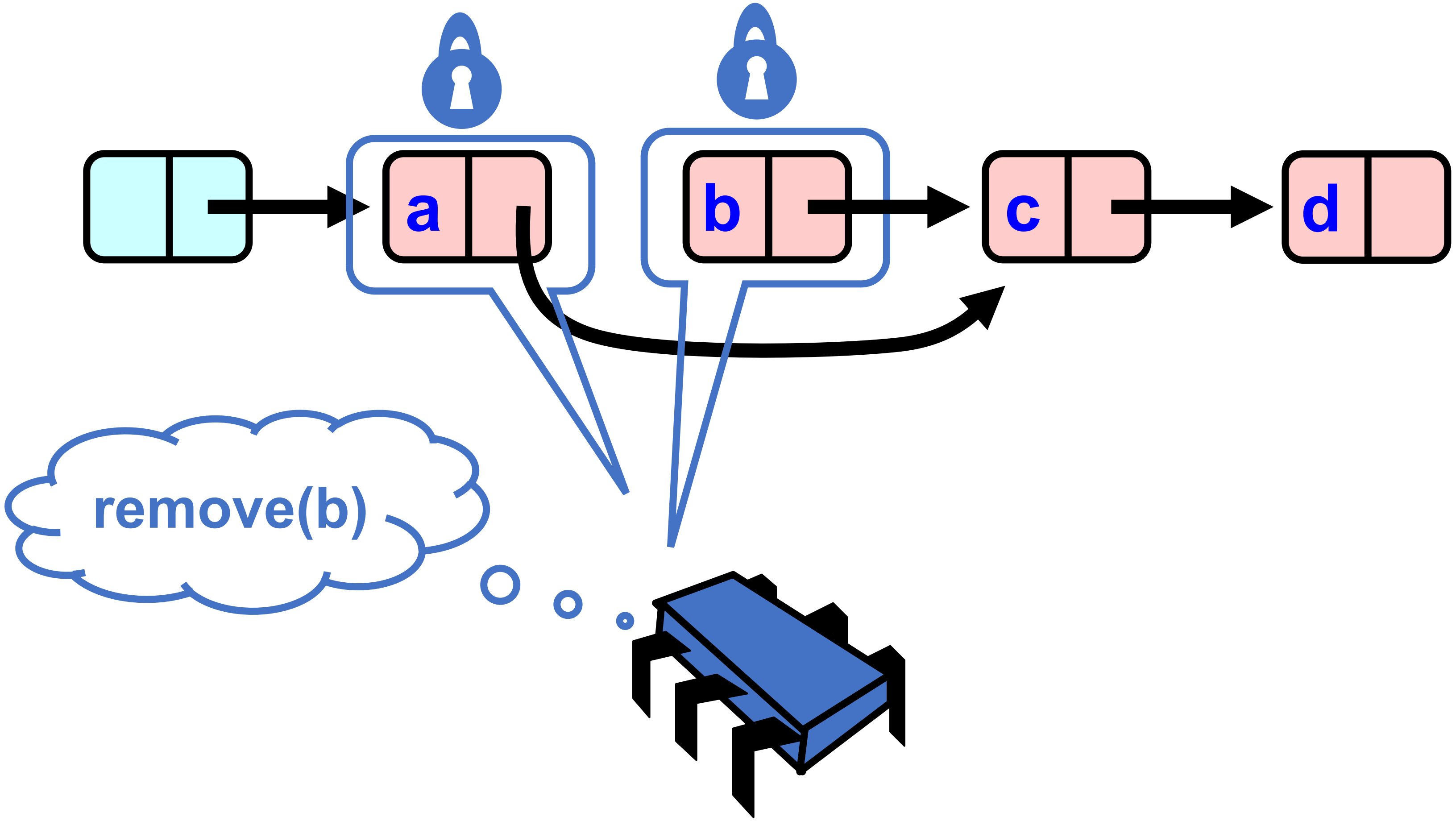
Removing a Node



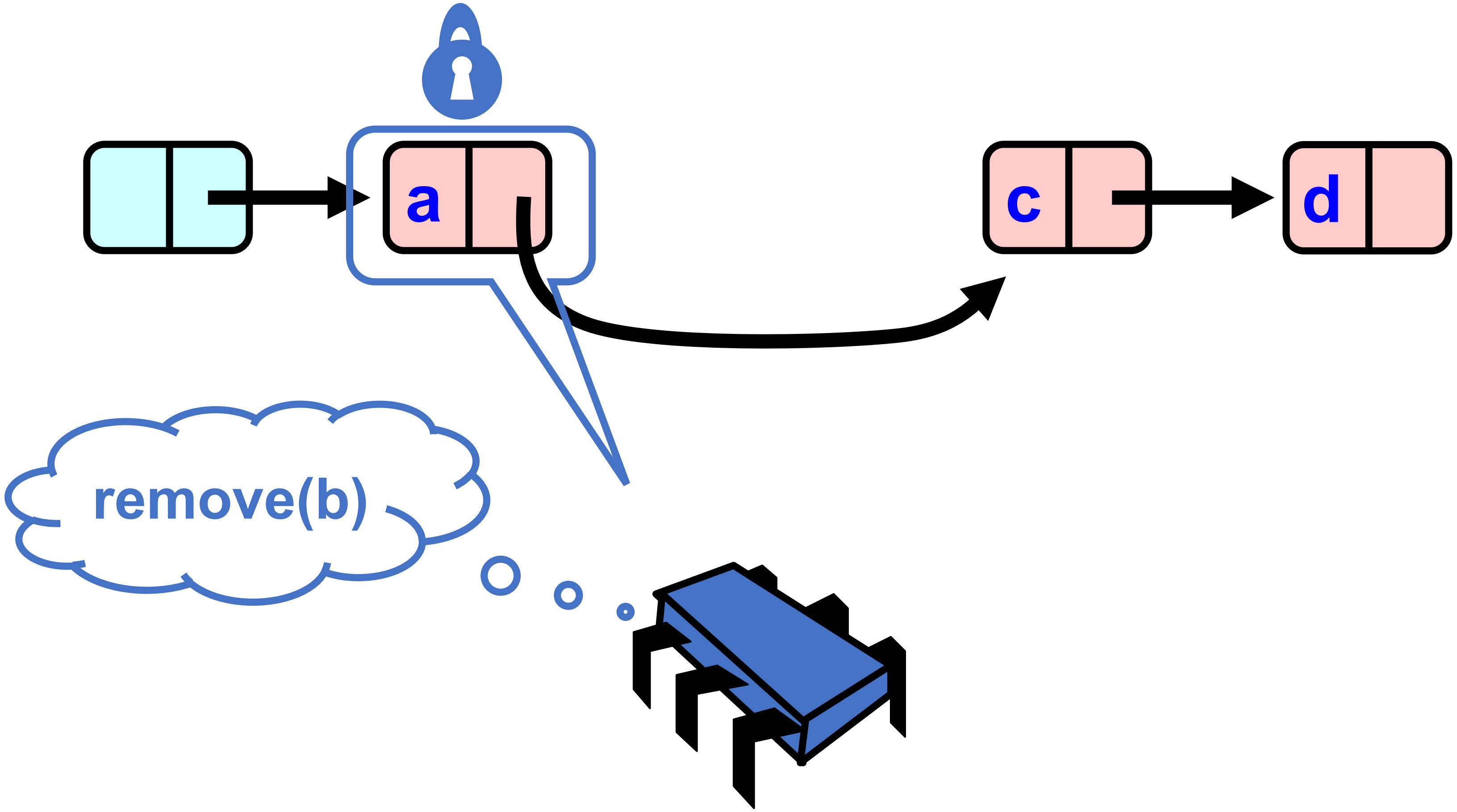
Removing a Node



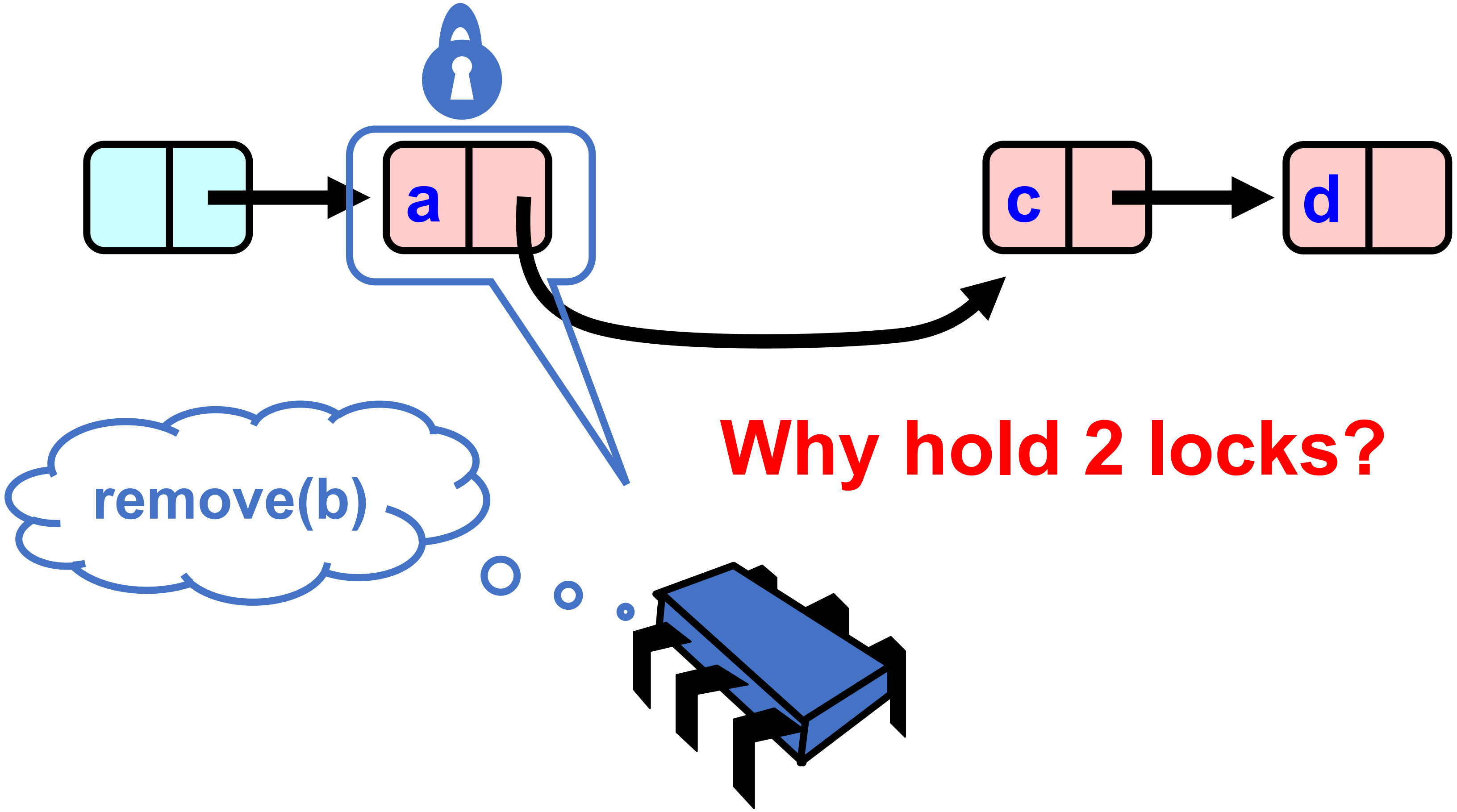
Removing a Node



Removing a Node

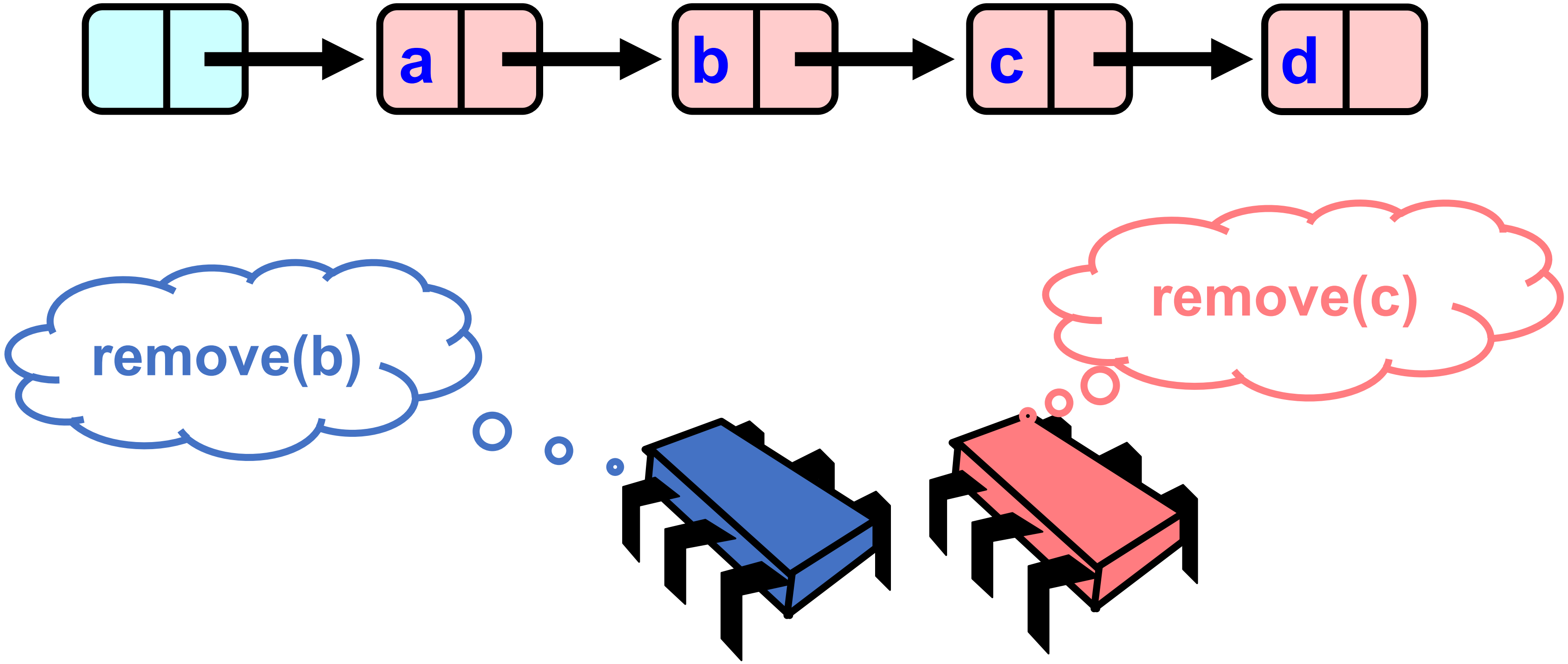


Removing a Node

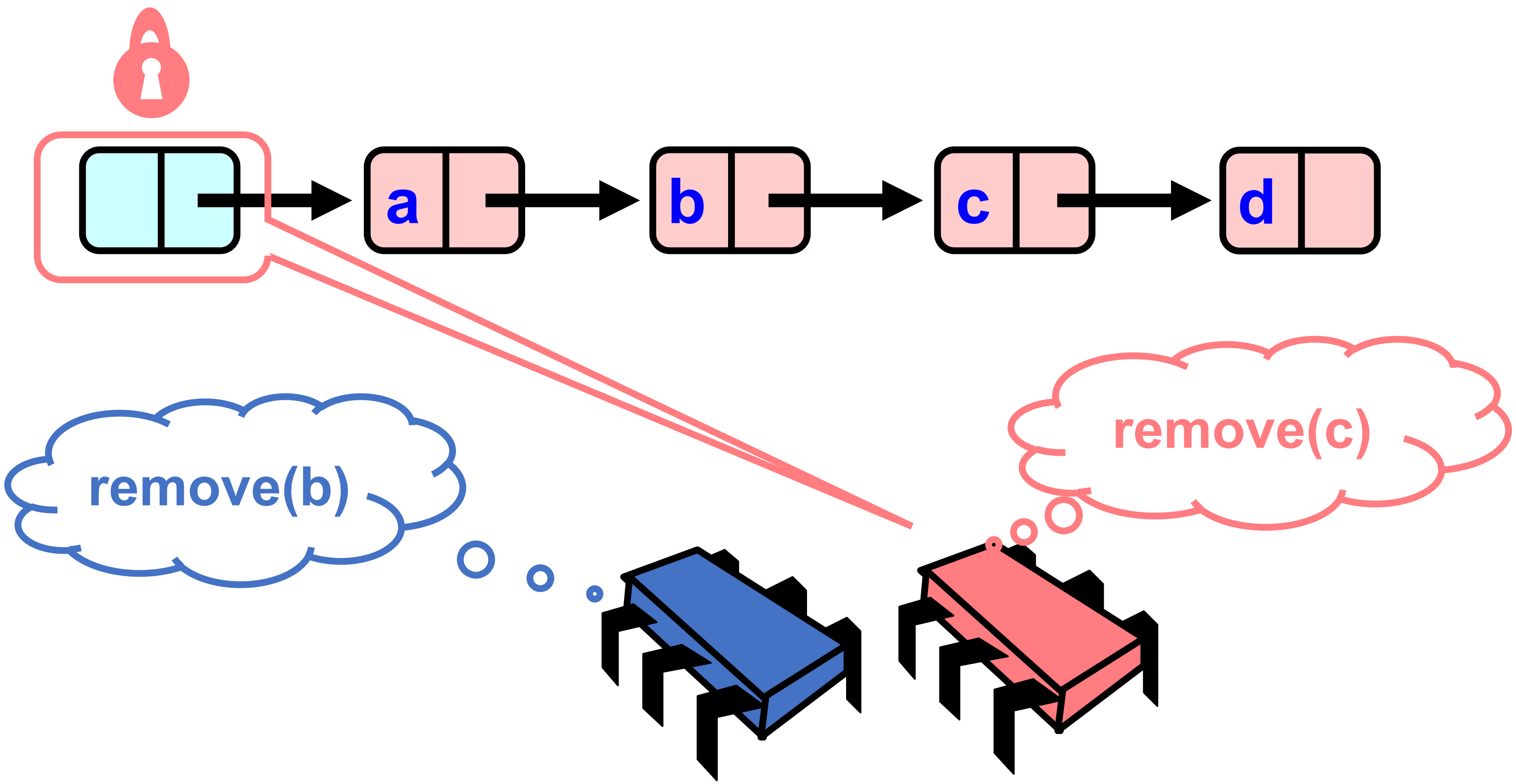


Why hold 2 locks?

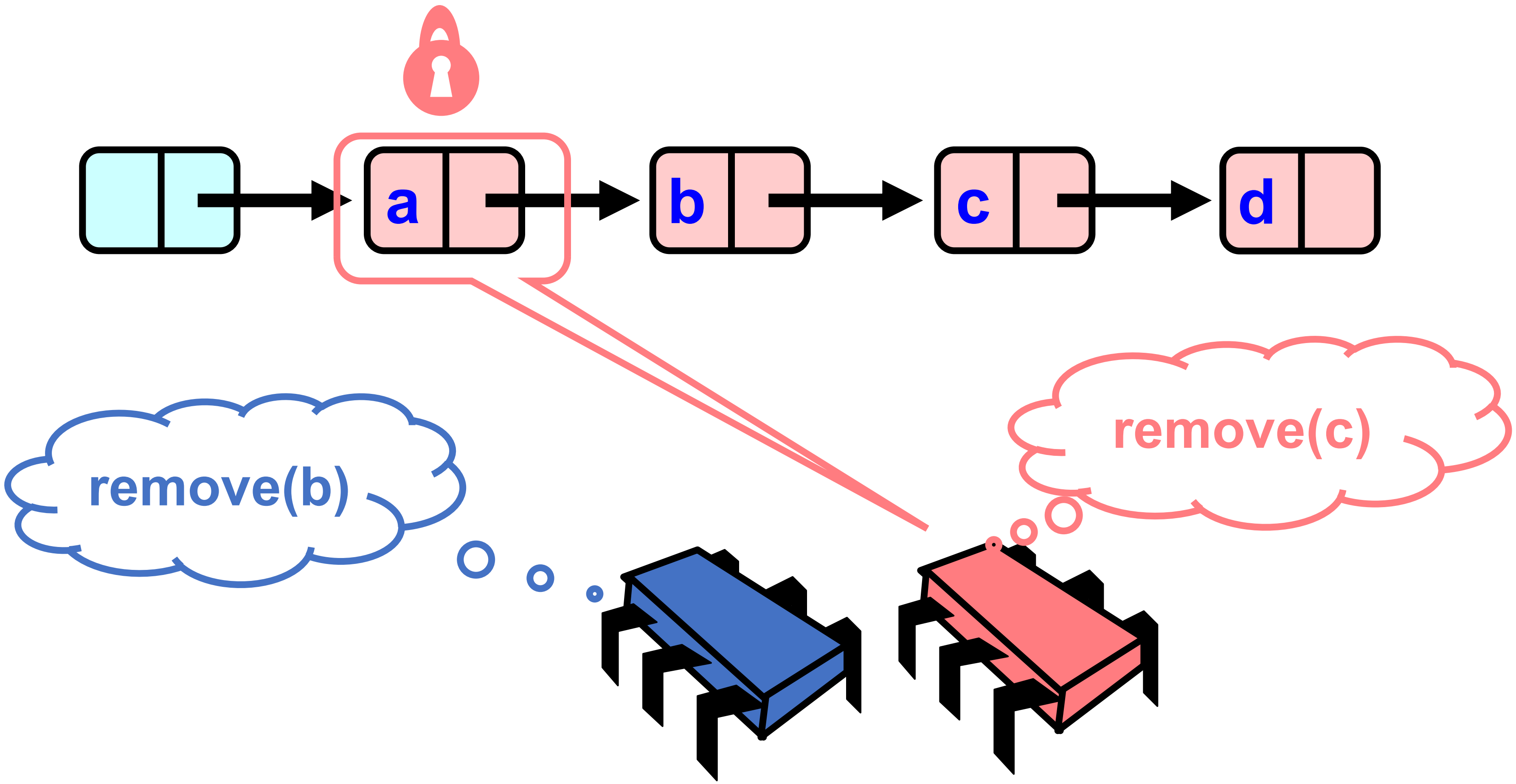
Concurrent Removes



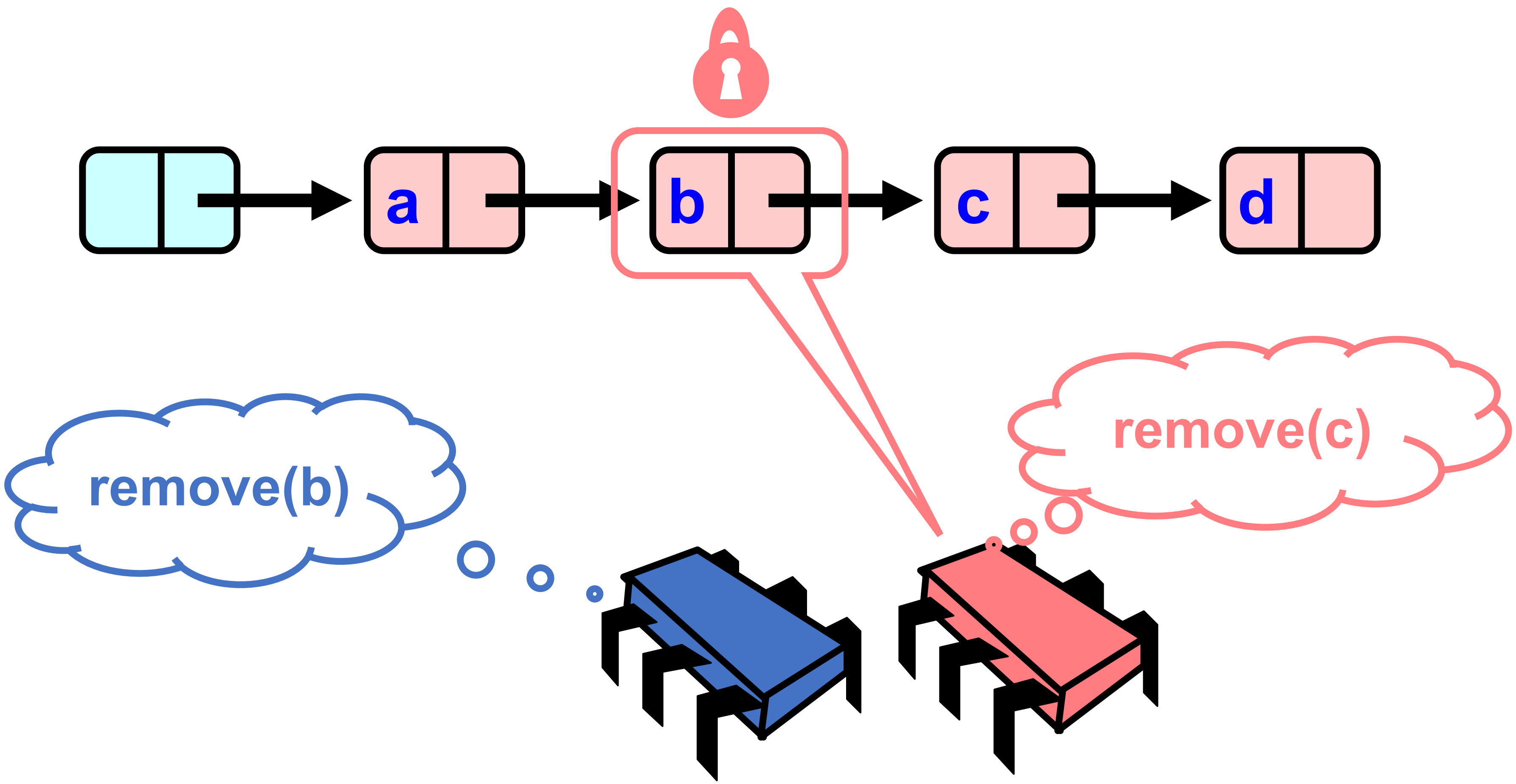
Concurrent Removes



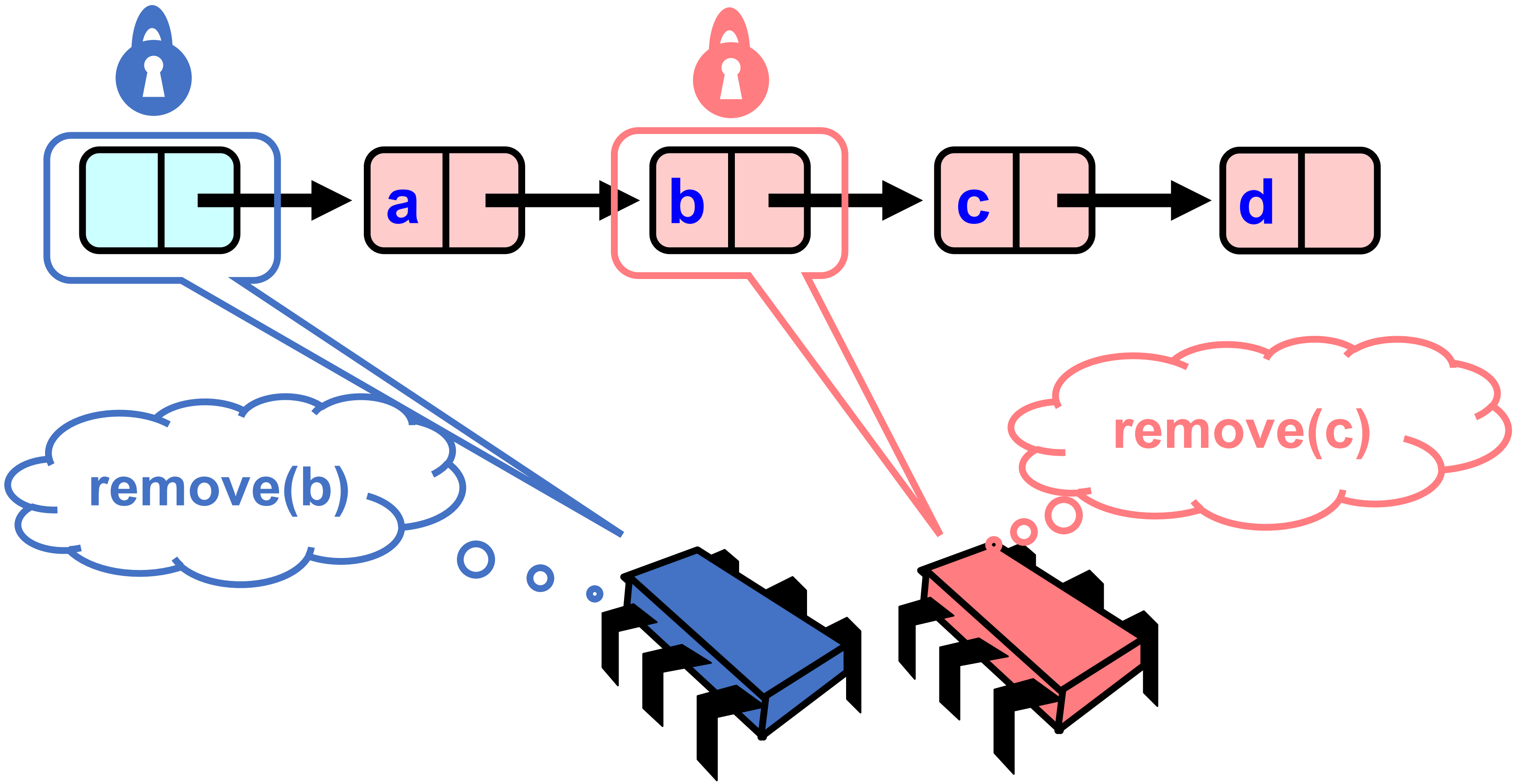
Concurrent Removes



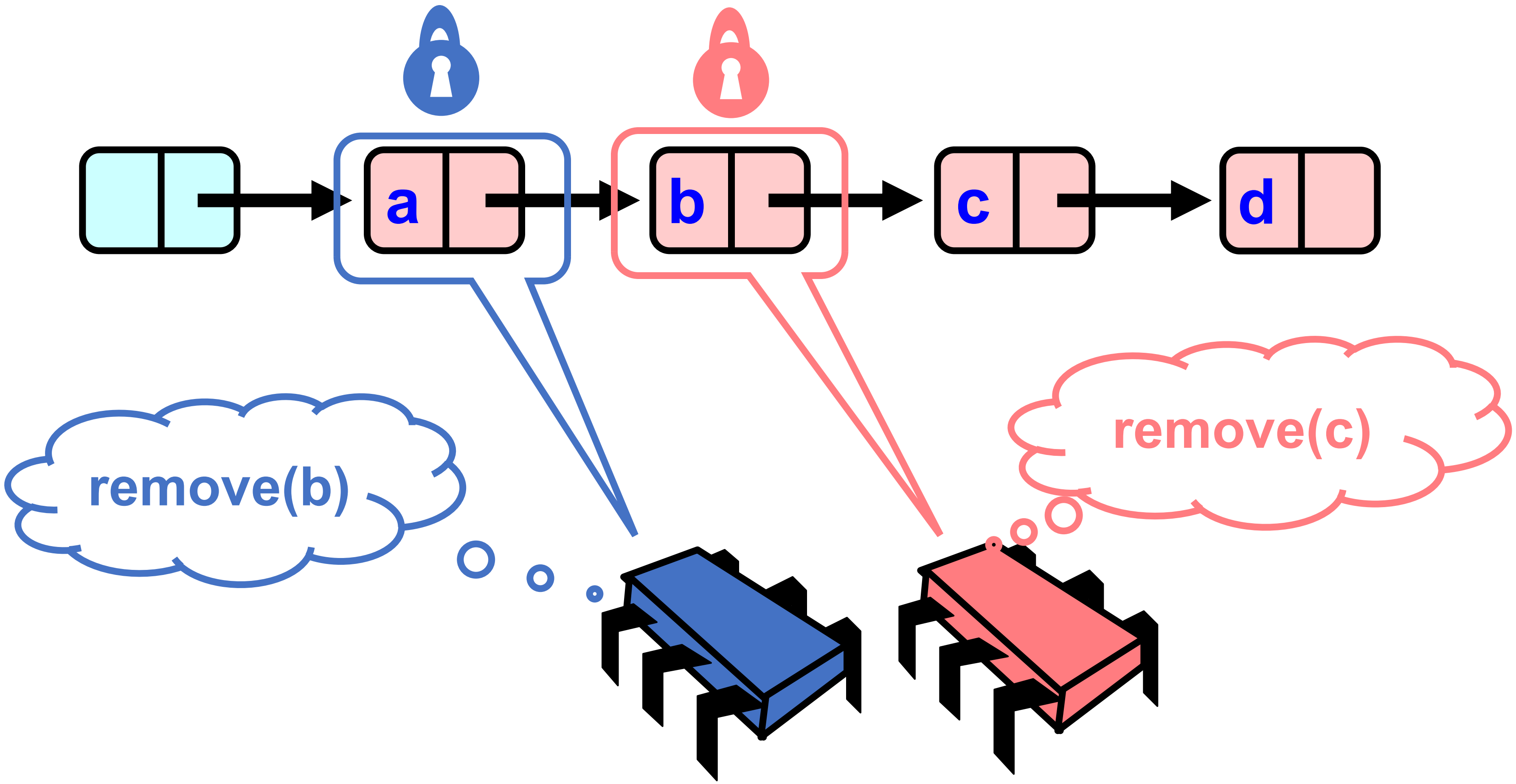
Concurrent Removes



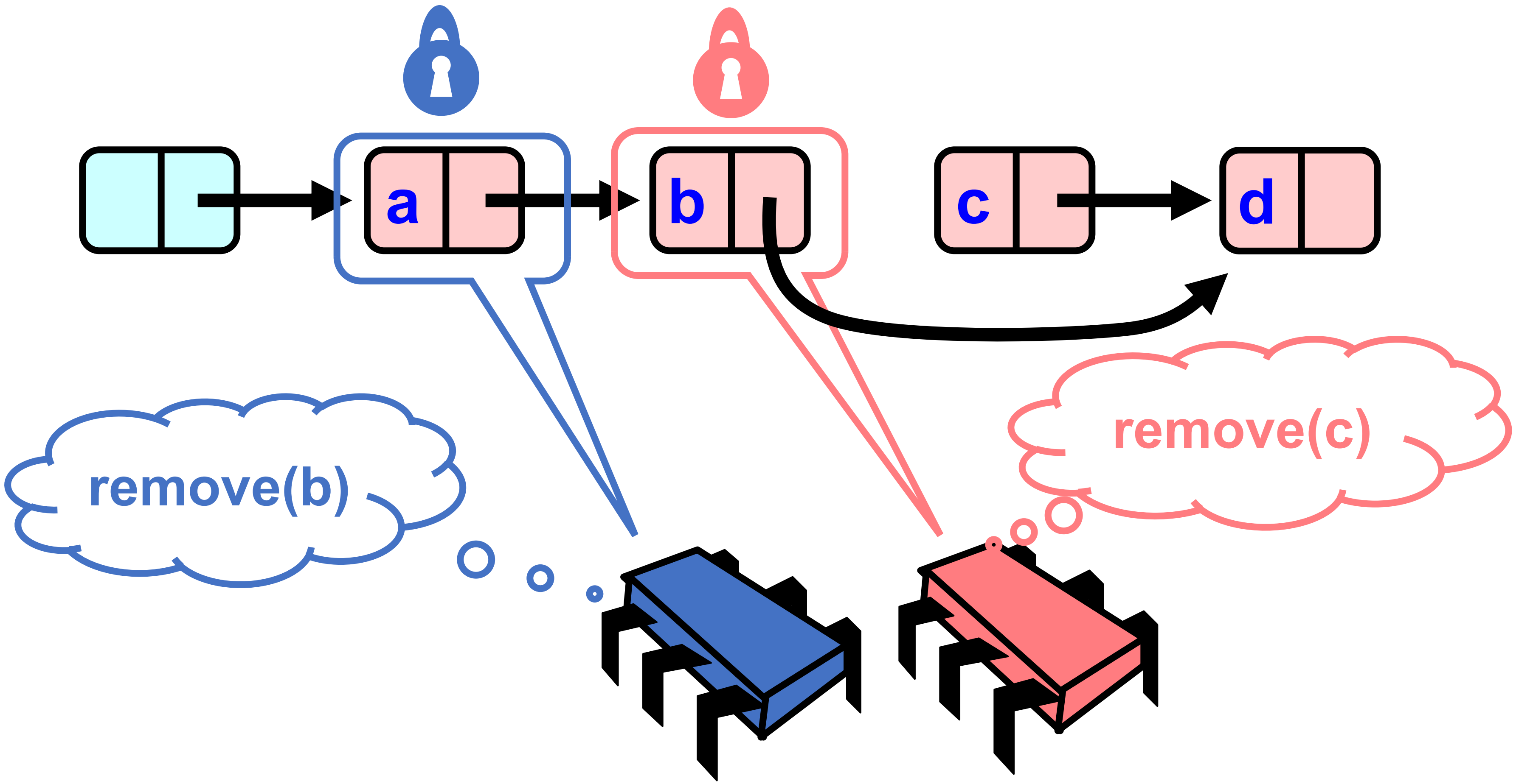
Concurrent Removes



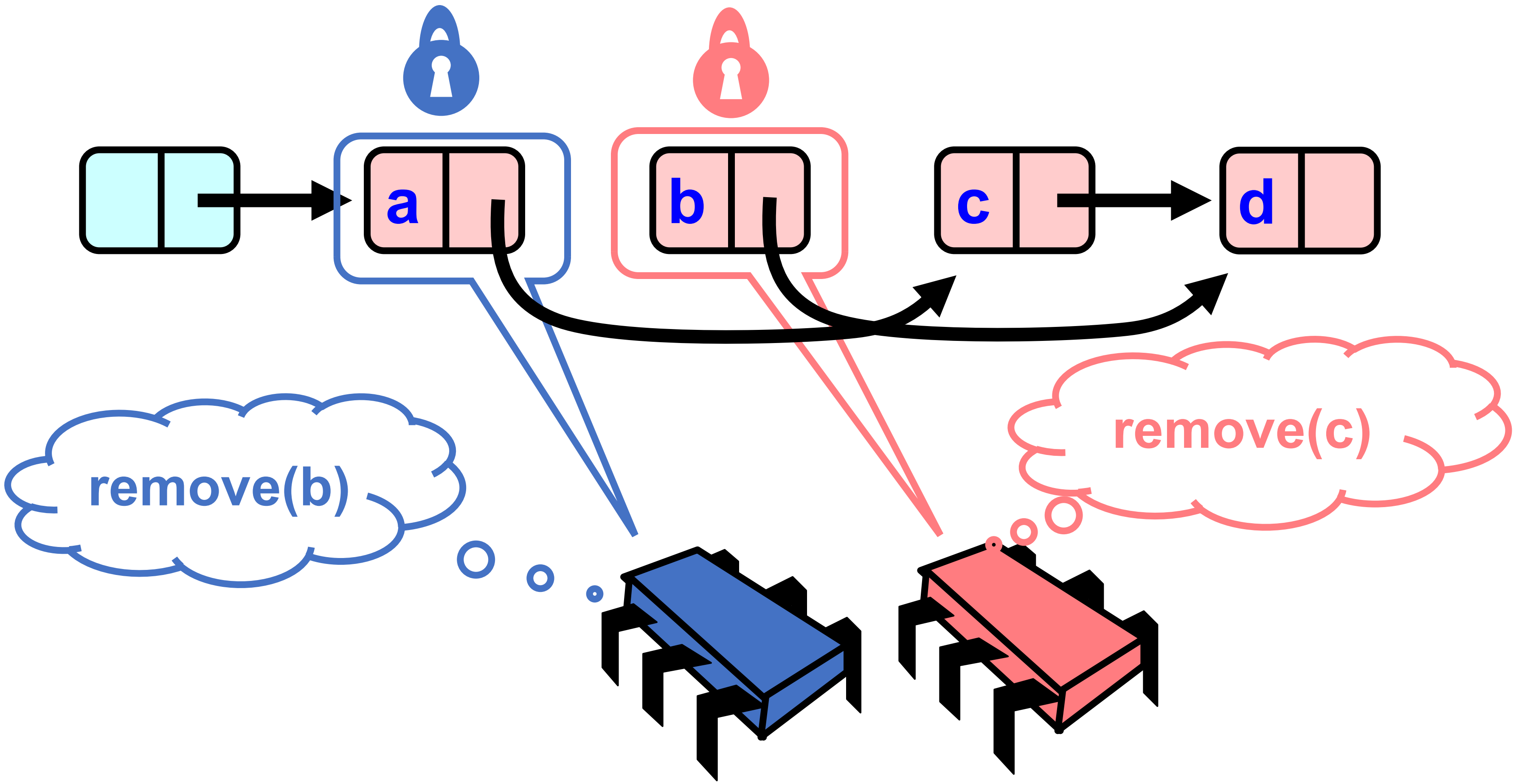
Concurrent Removes



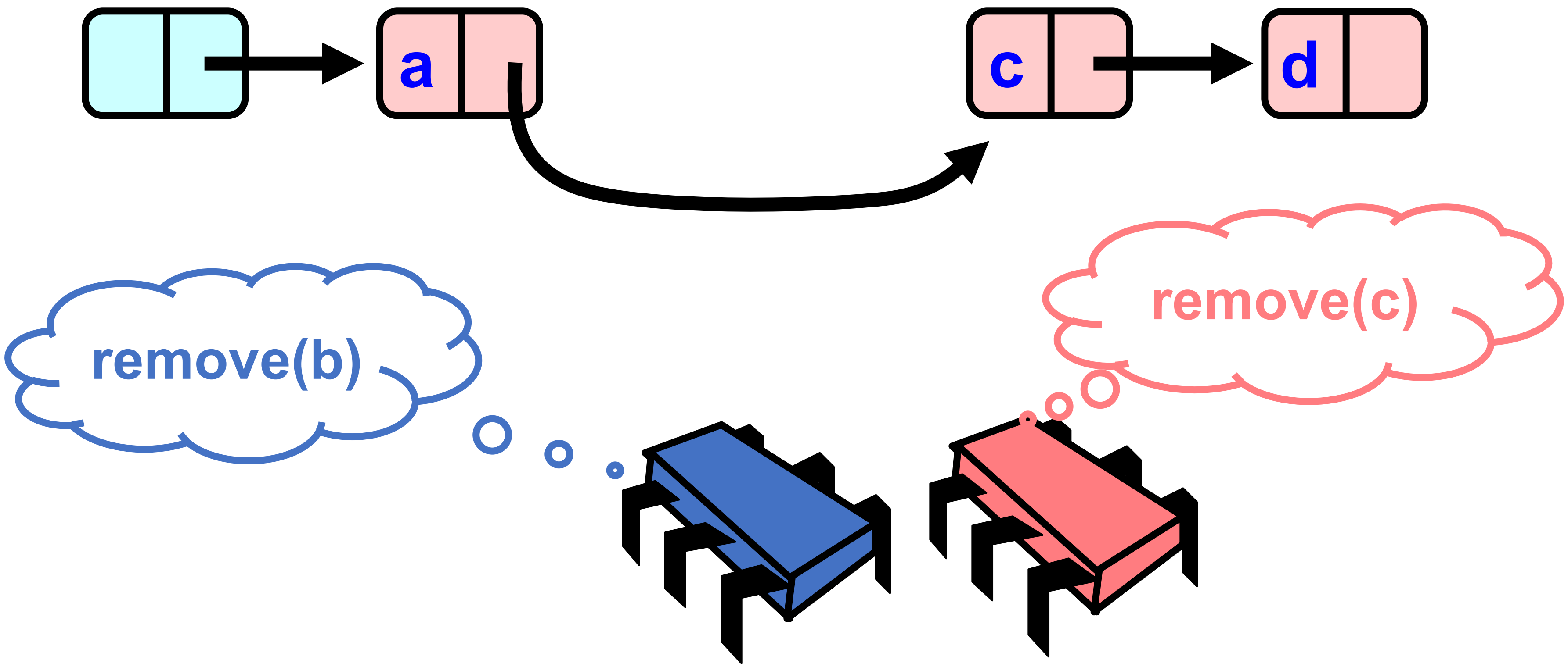
Concurrent Removes



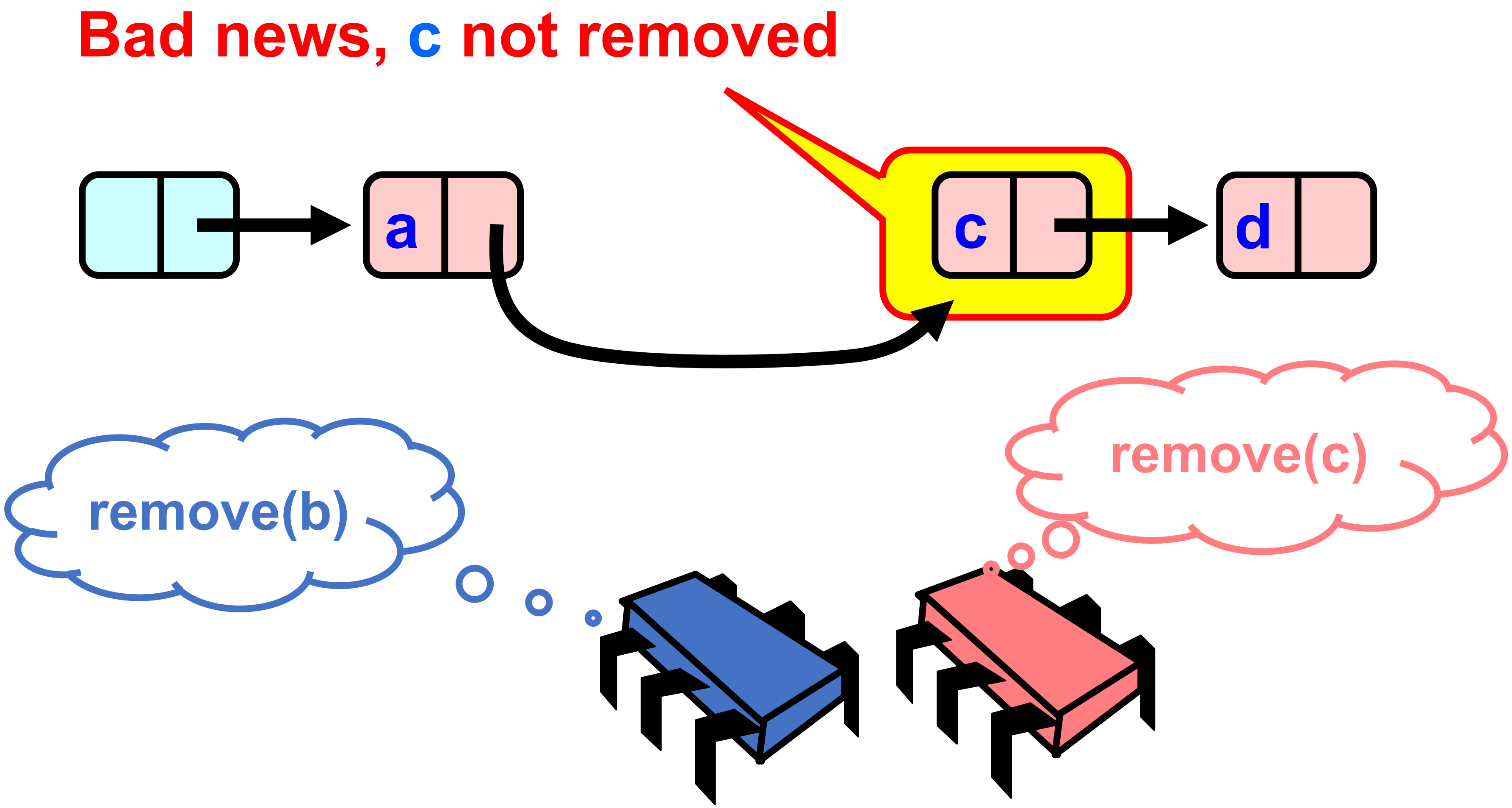
Concurrent Removes



Uh, Oh

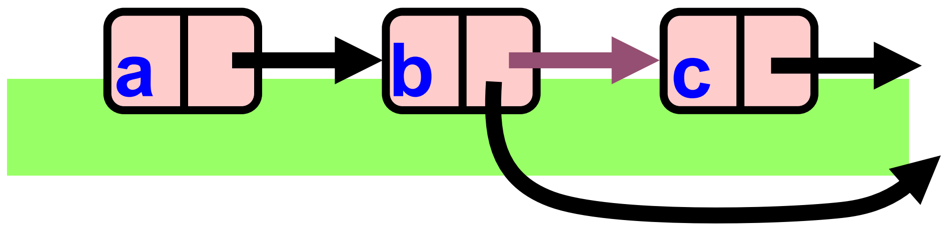


Uh, Oh

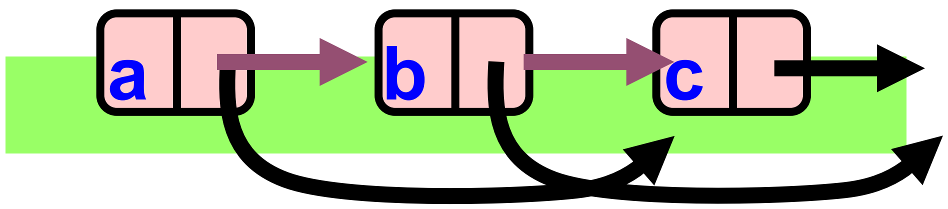


Problem

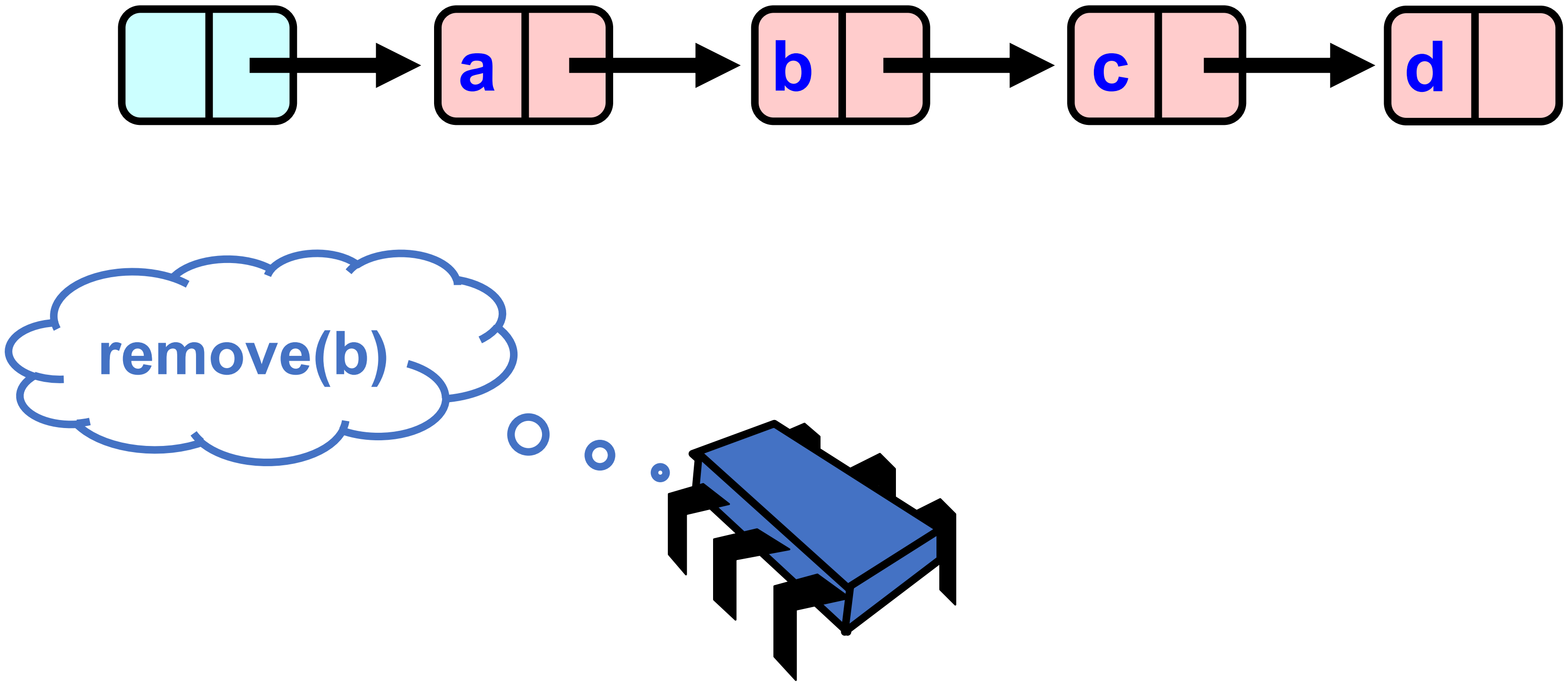
- To delete node c
  - Swing node b's next field to d



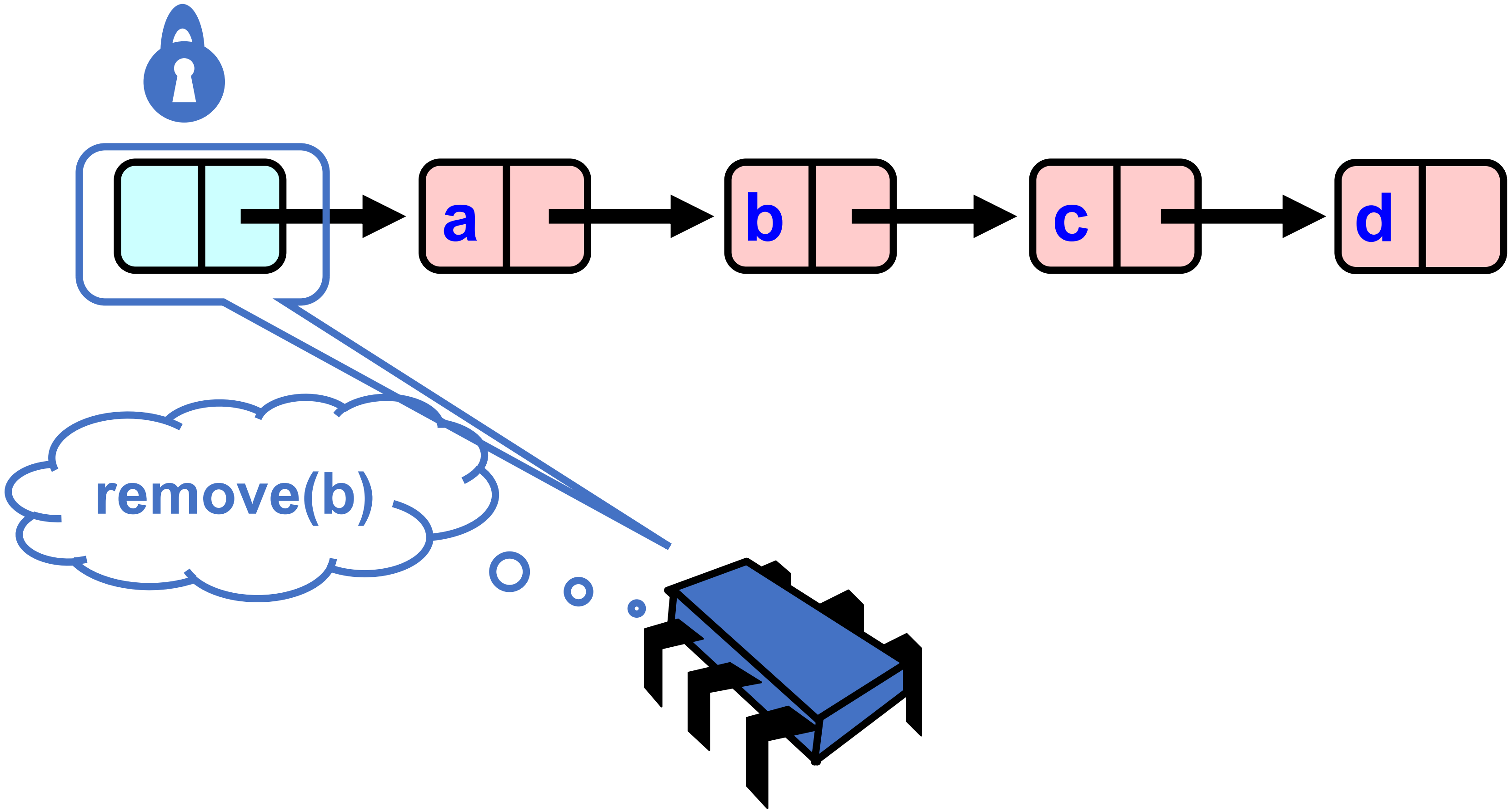
- Problem is,
  - ***Data conflict:***
  - Someone deleting b concurrently could direct a pointer to C



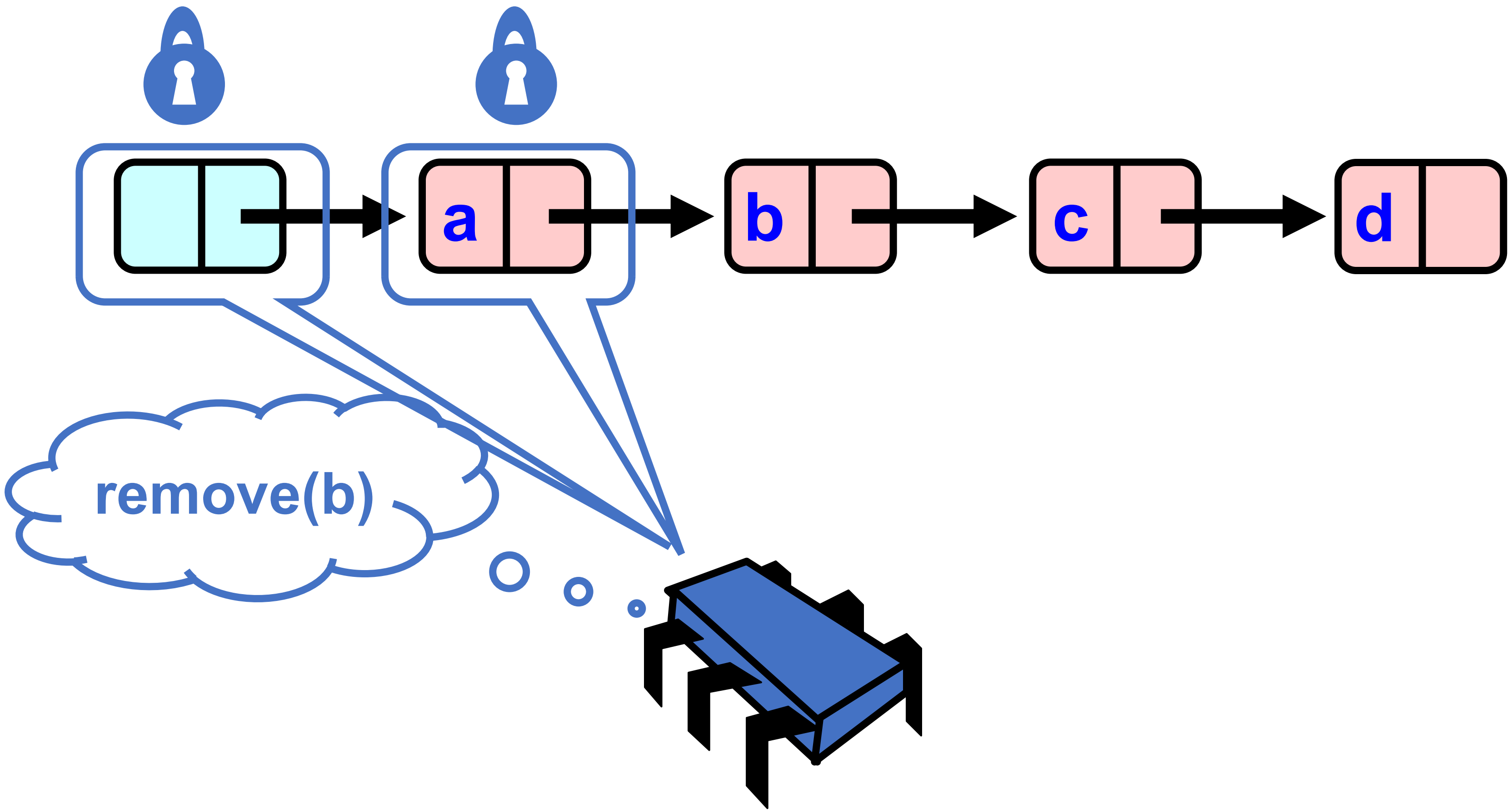
Hand-Over-Hand Again



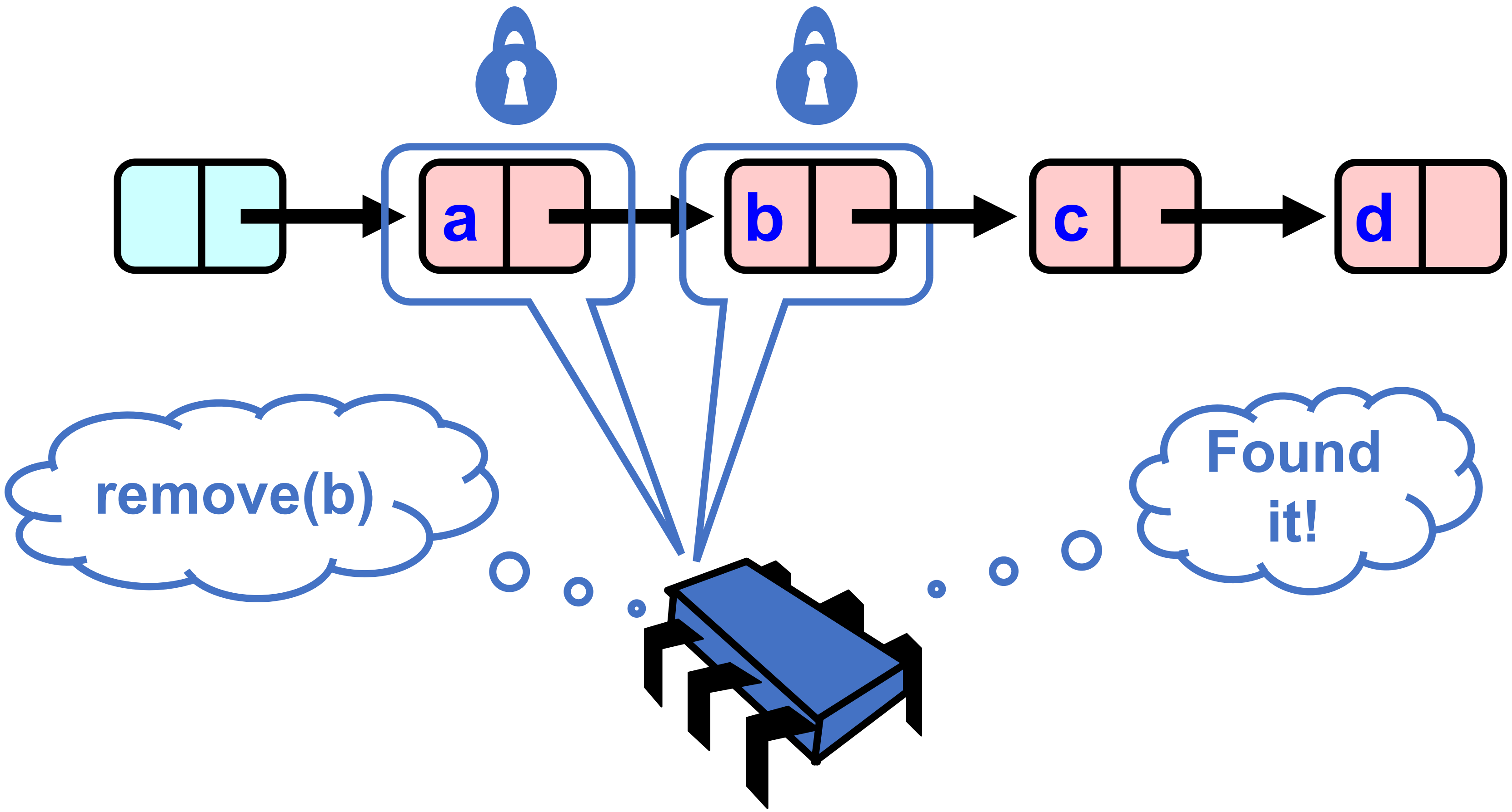
Hand-Over-Hand Again



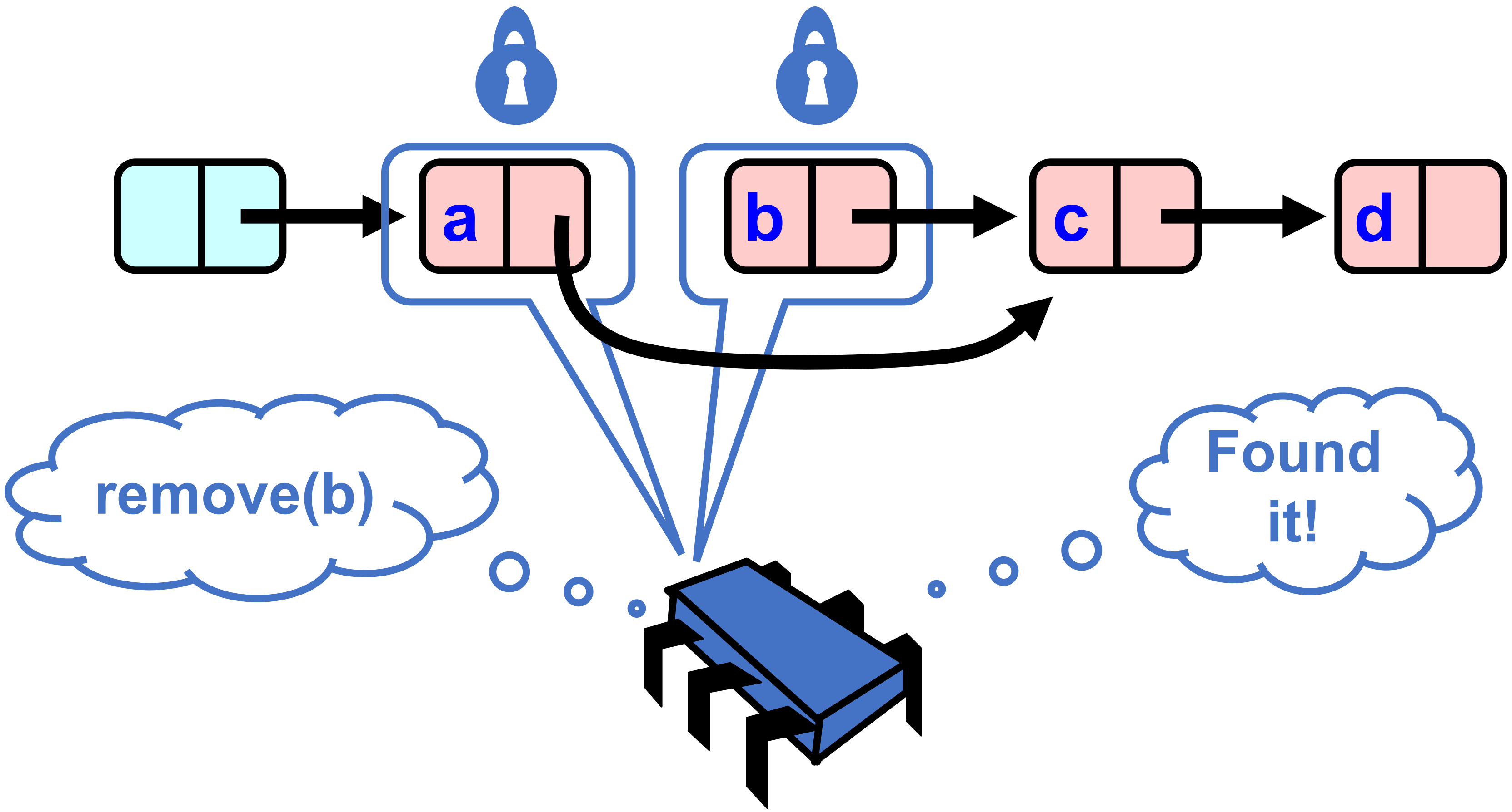
Hand-Over-Hand Again



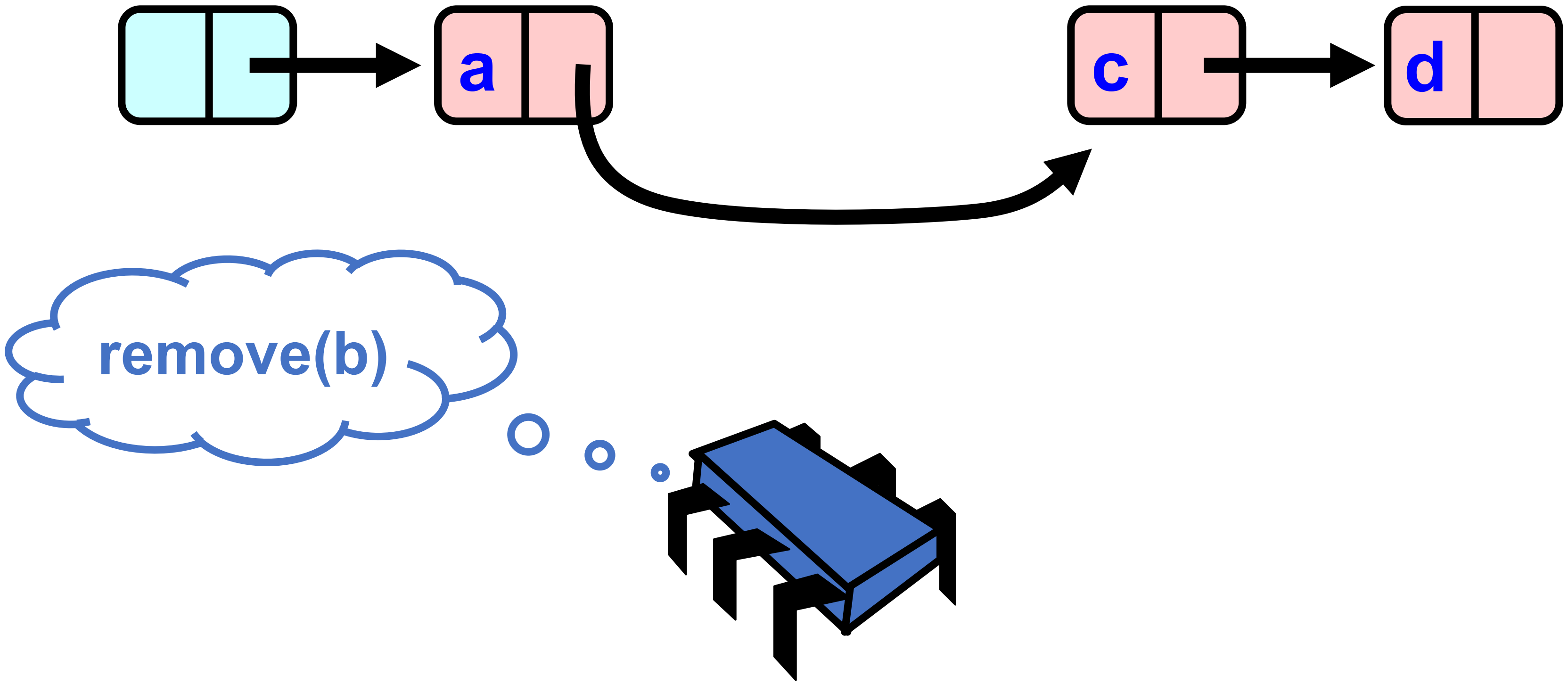
Hand-Over-Hand Again



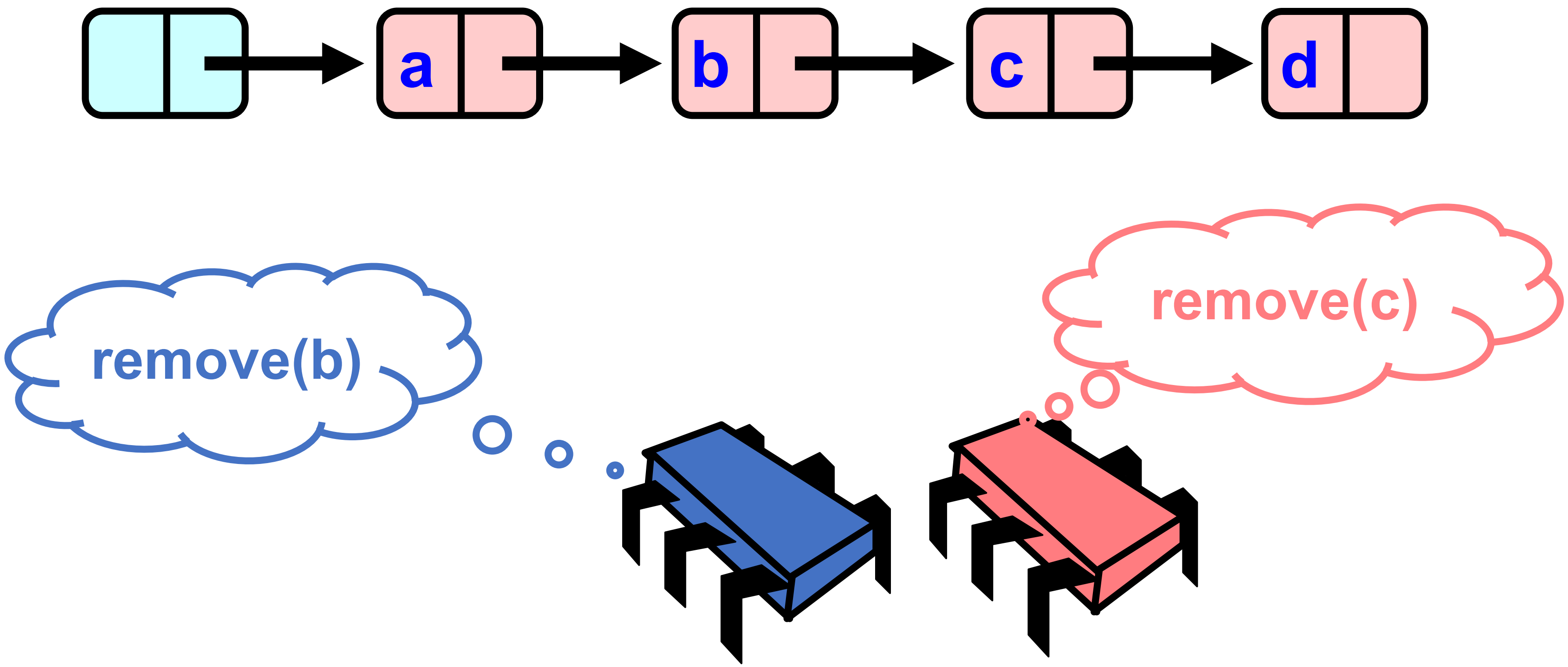
Hand-Over-Hand Again



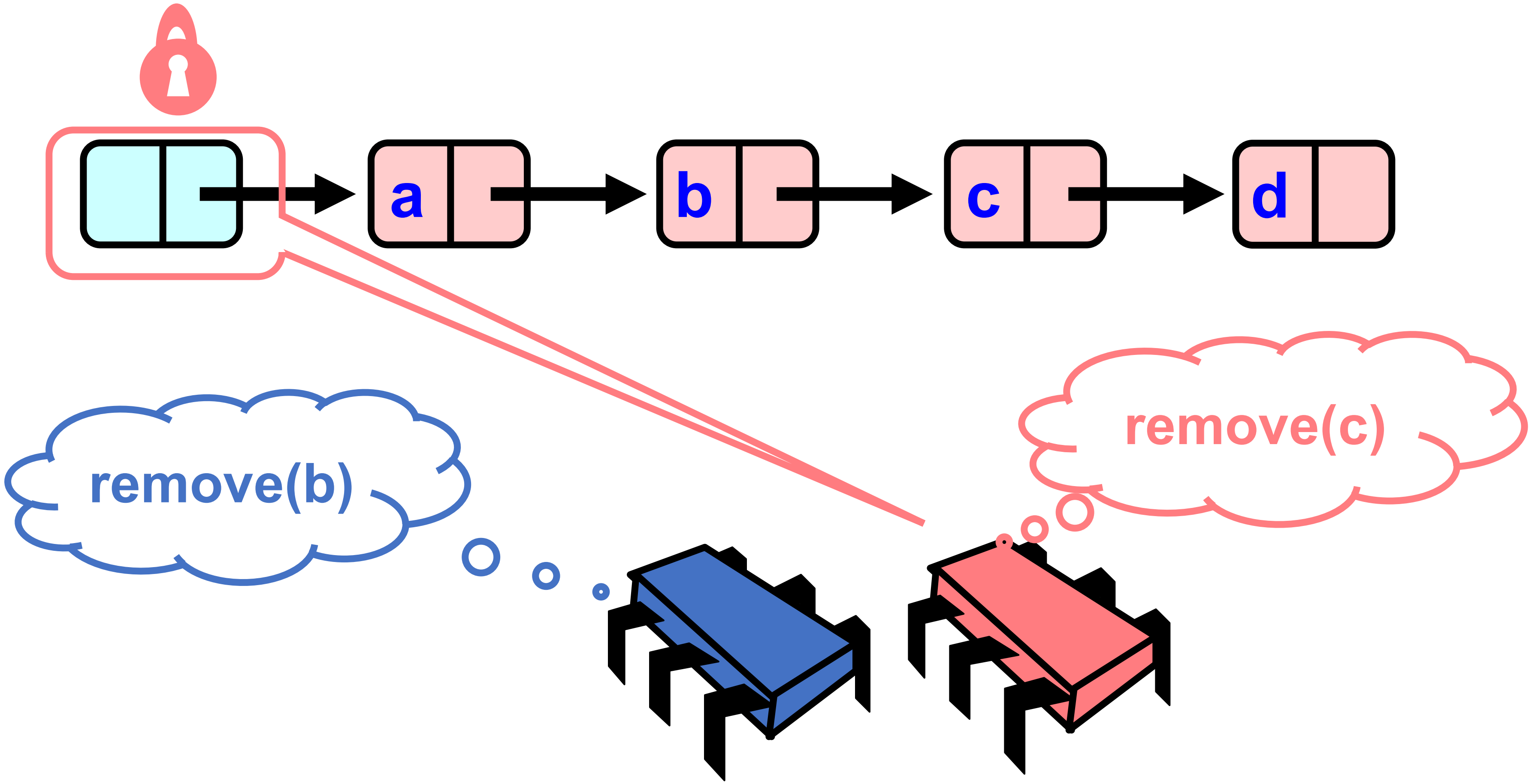
Hand-Over-Hand Again



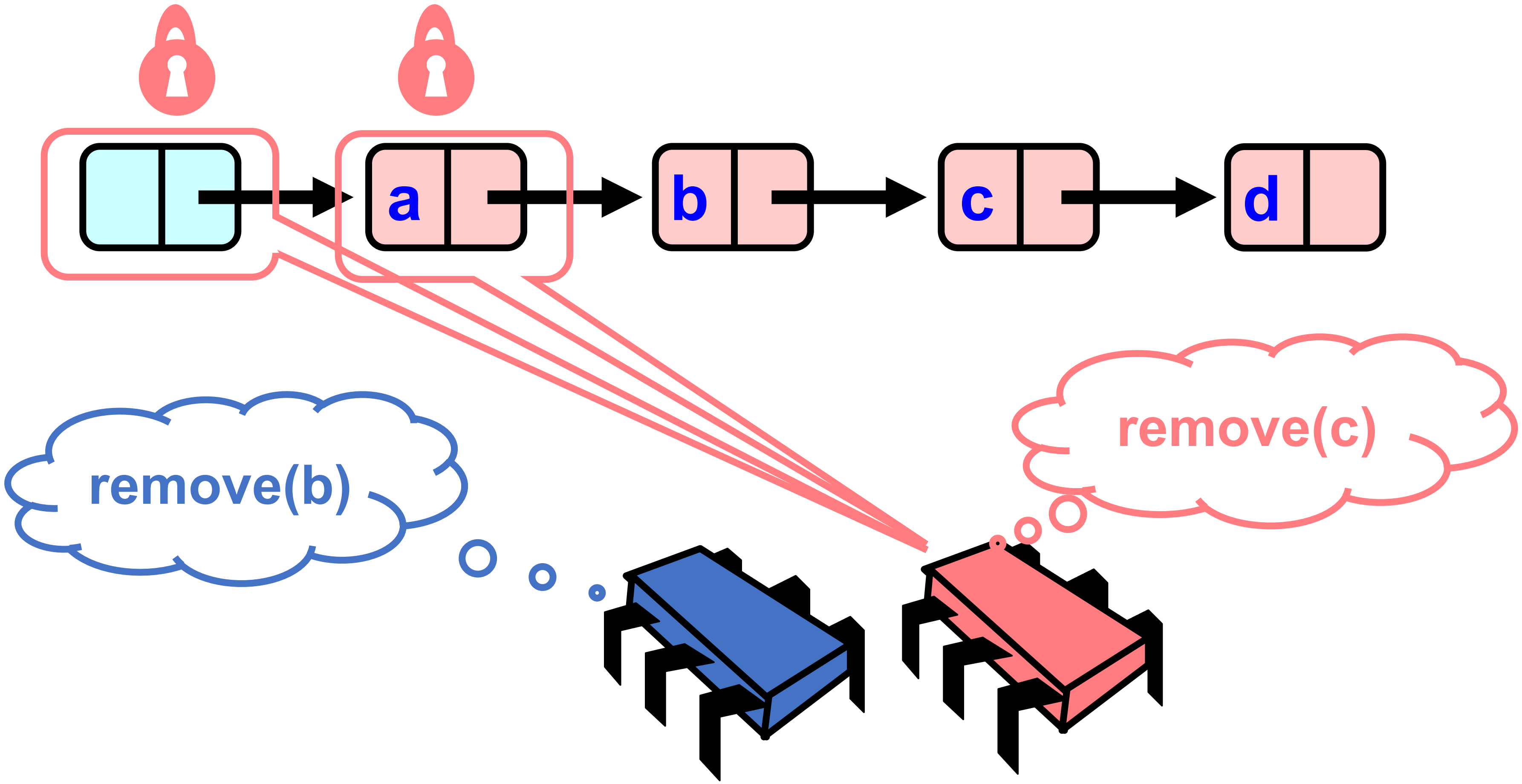
Removing a Node



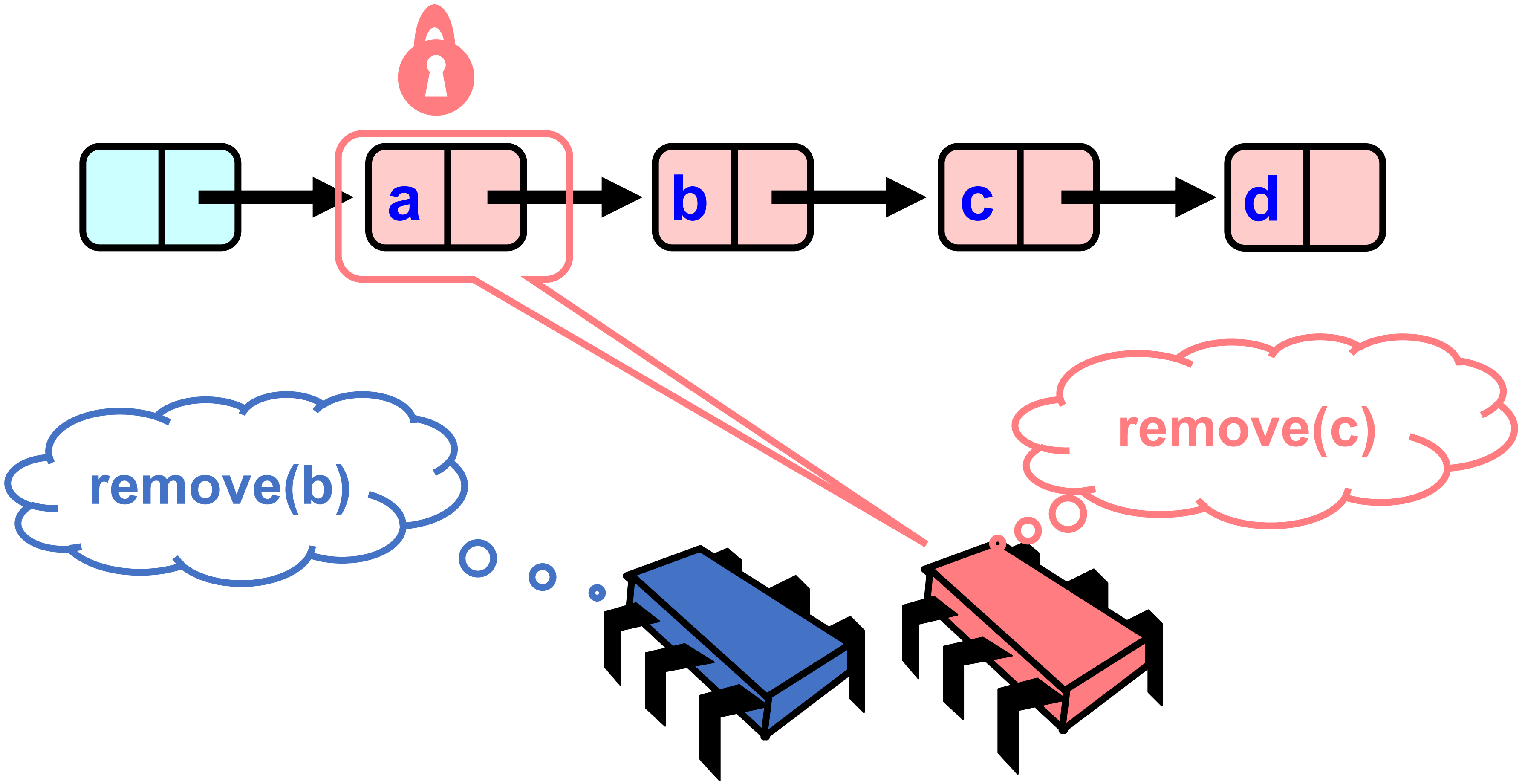
Removing a Node



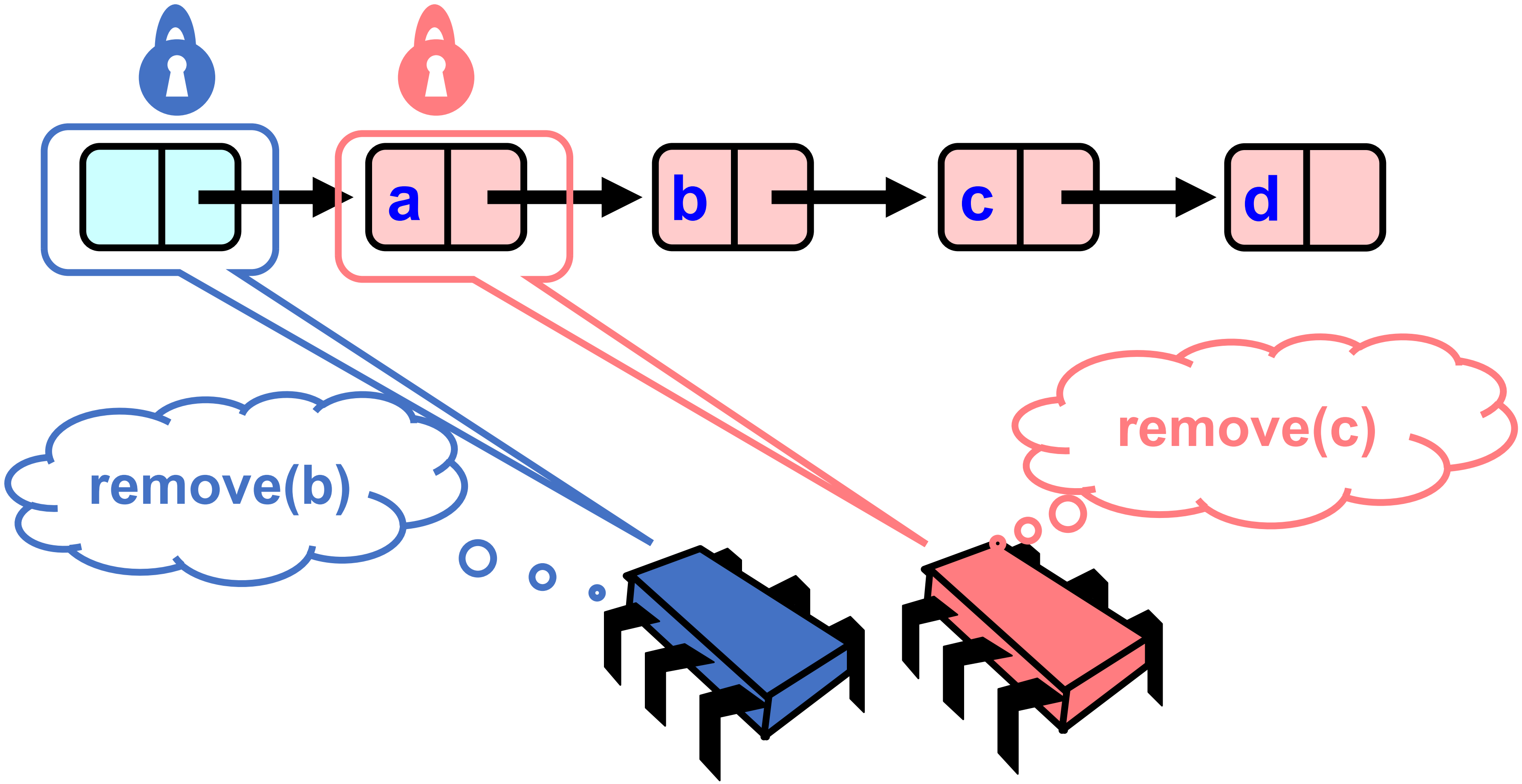
Removing a Node



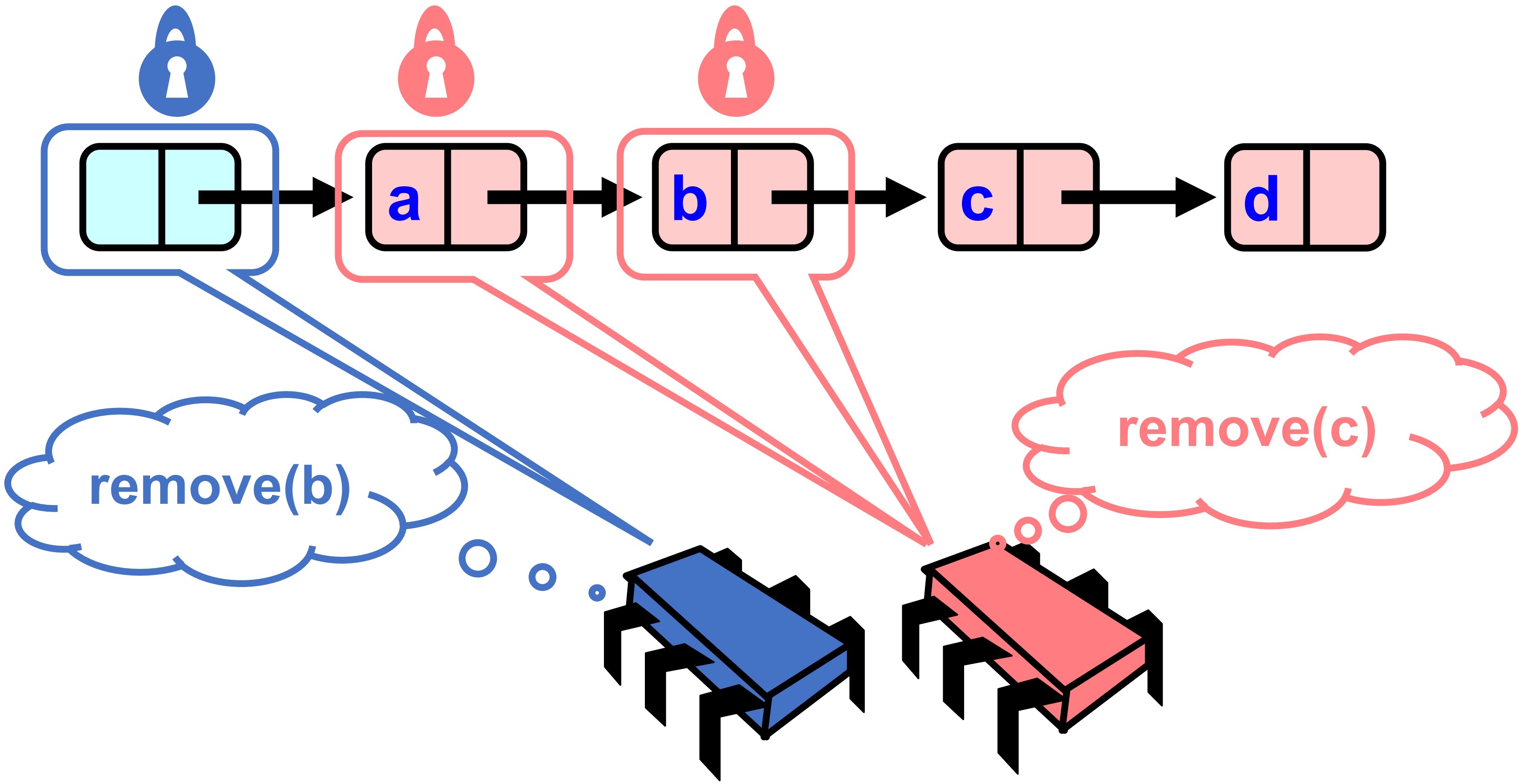
Removing a Node



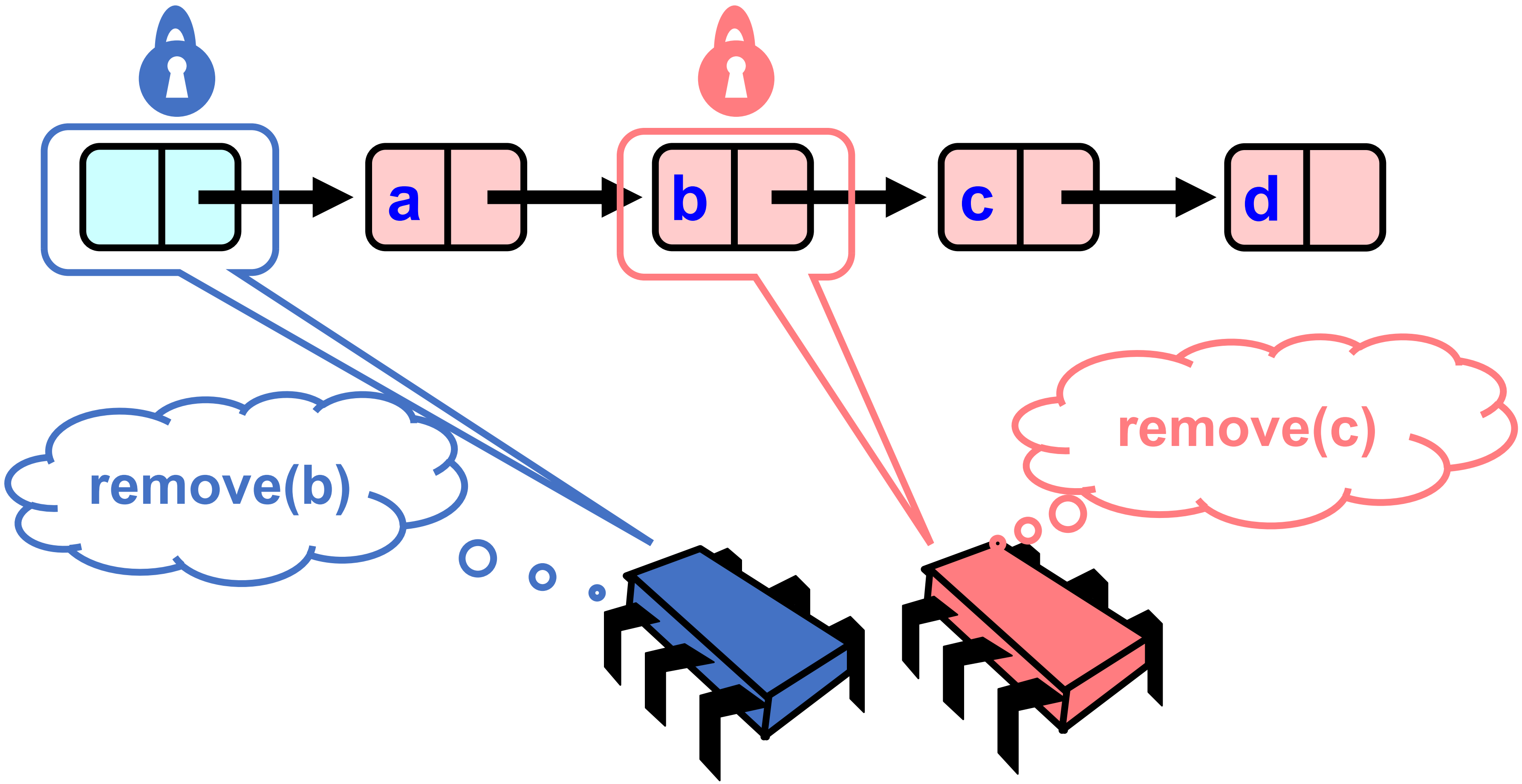
Removing a Node



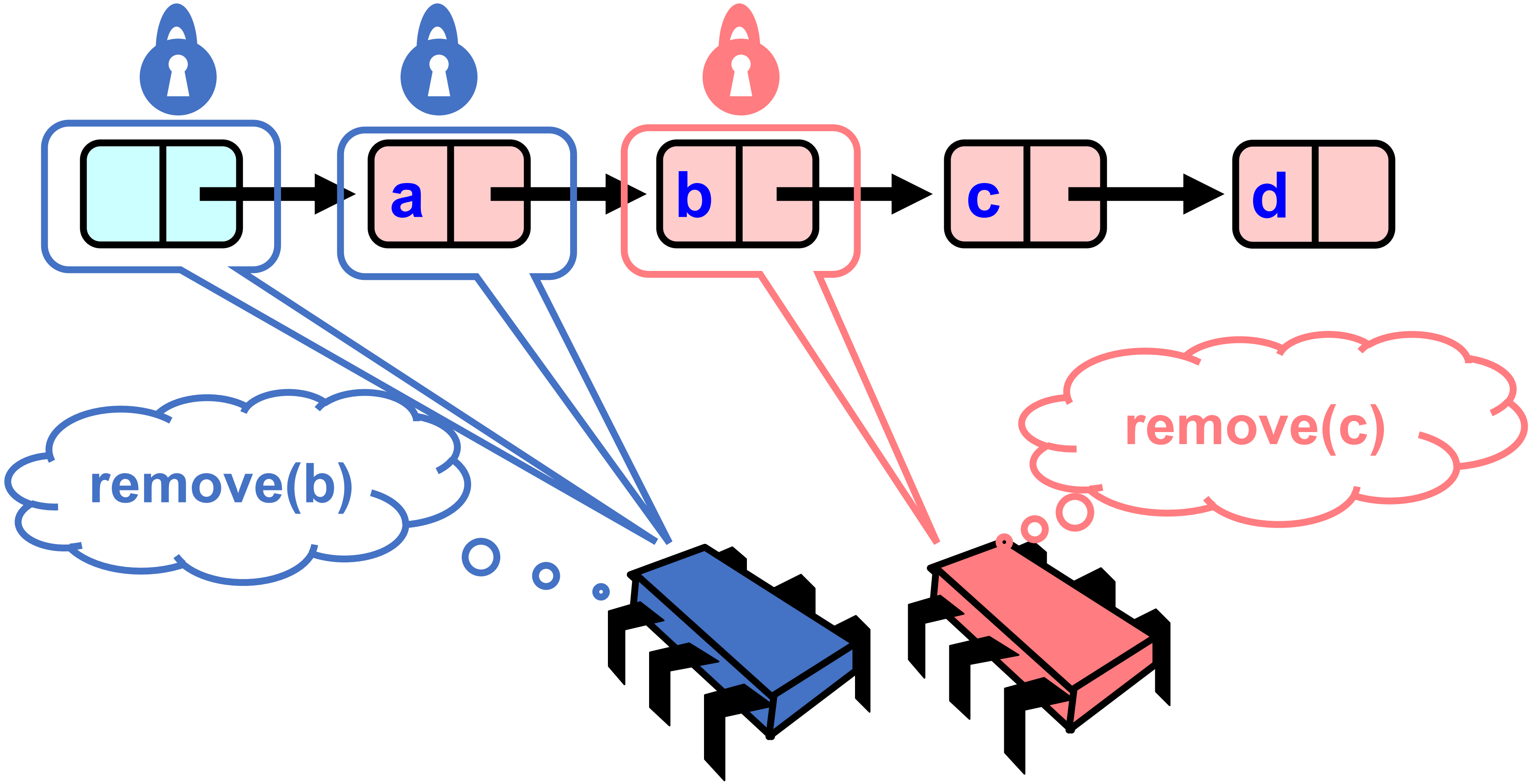
Removing a Node



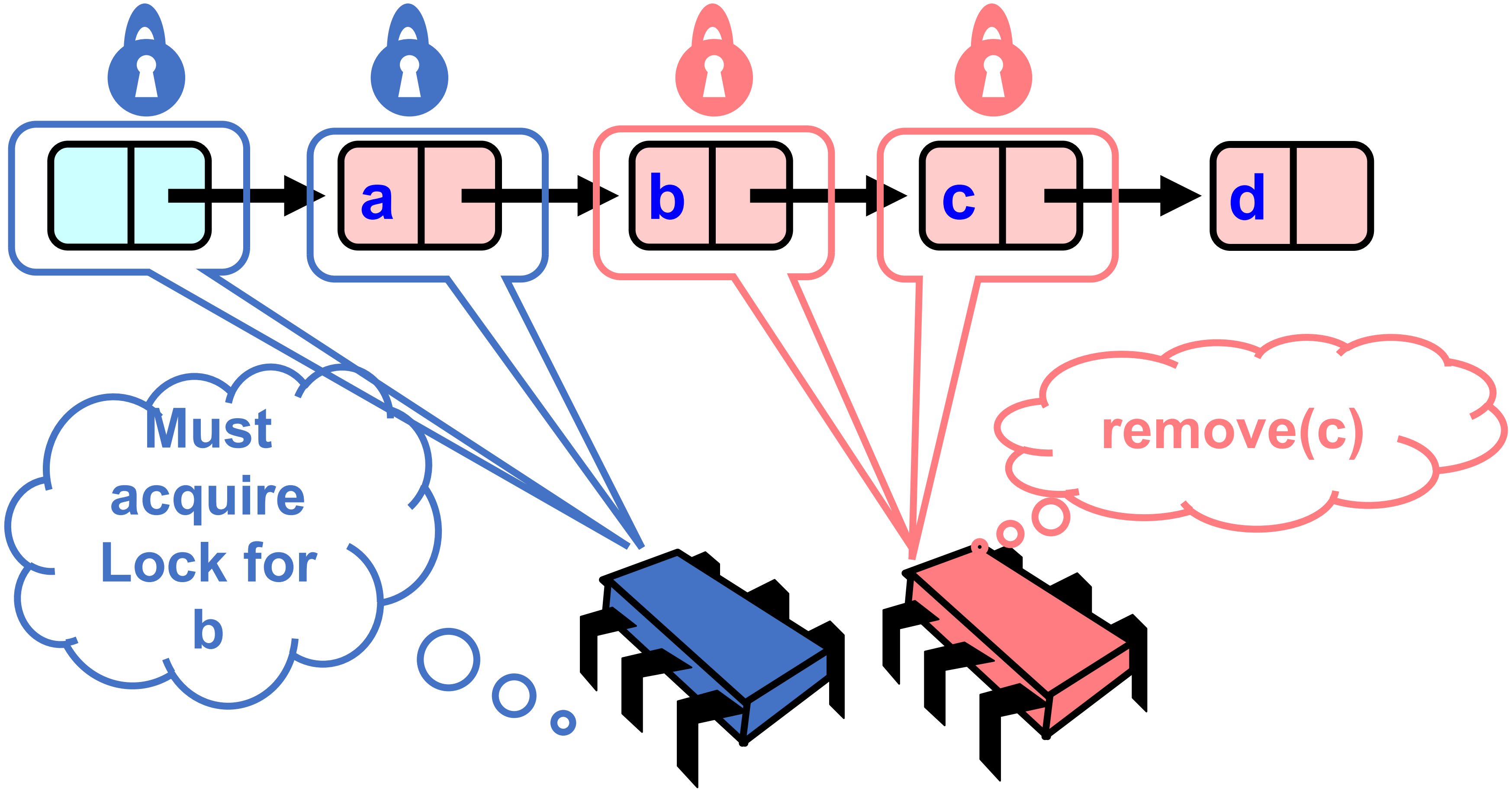
Removing a Node



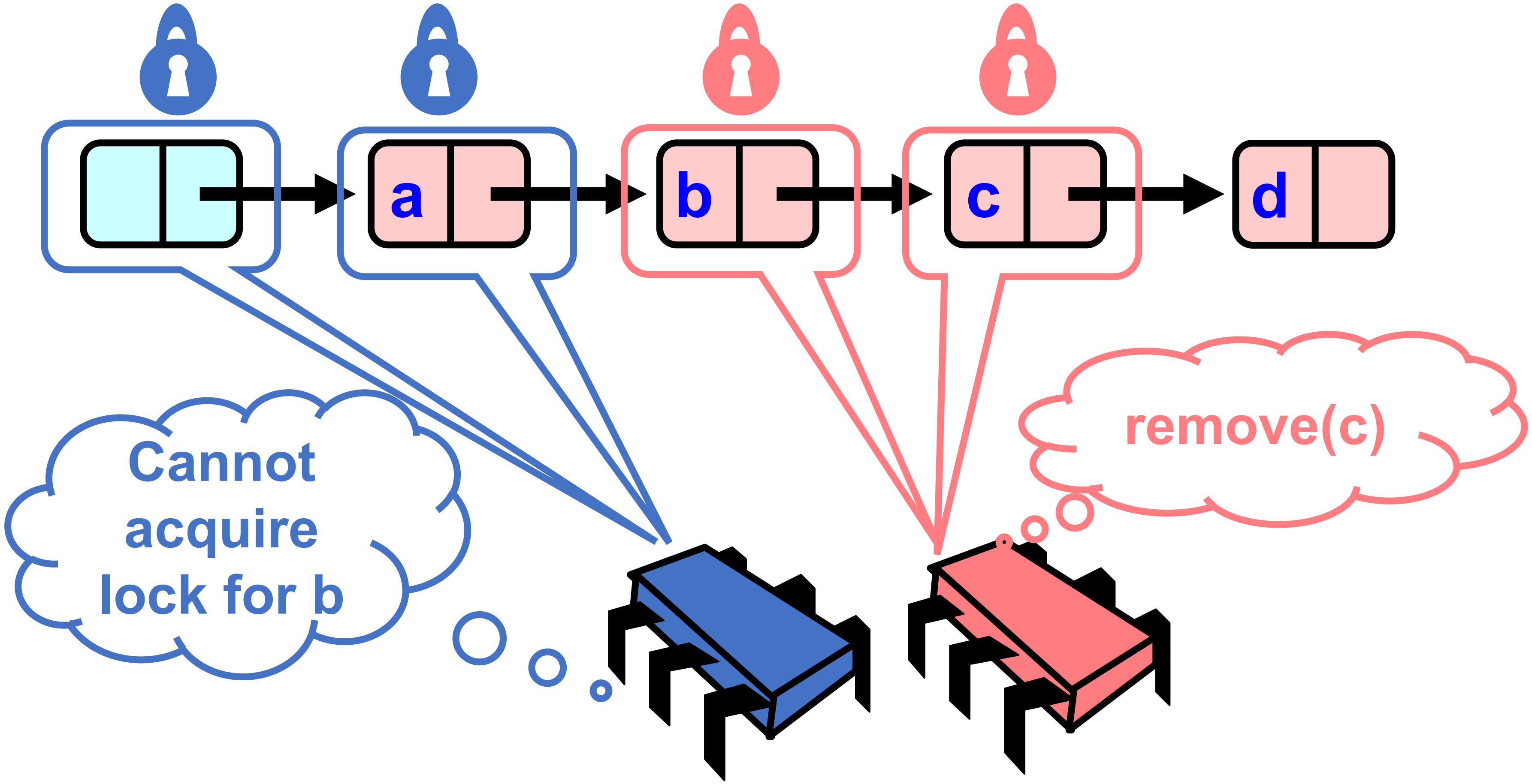
Removing a Node



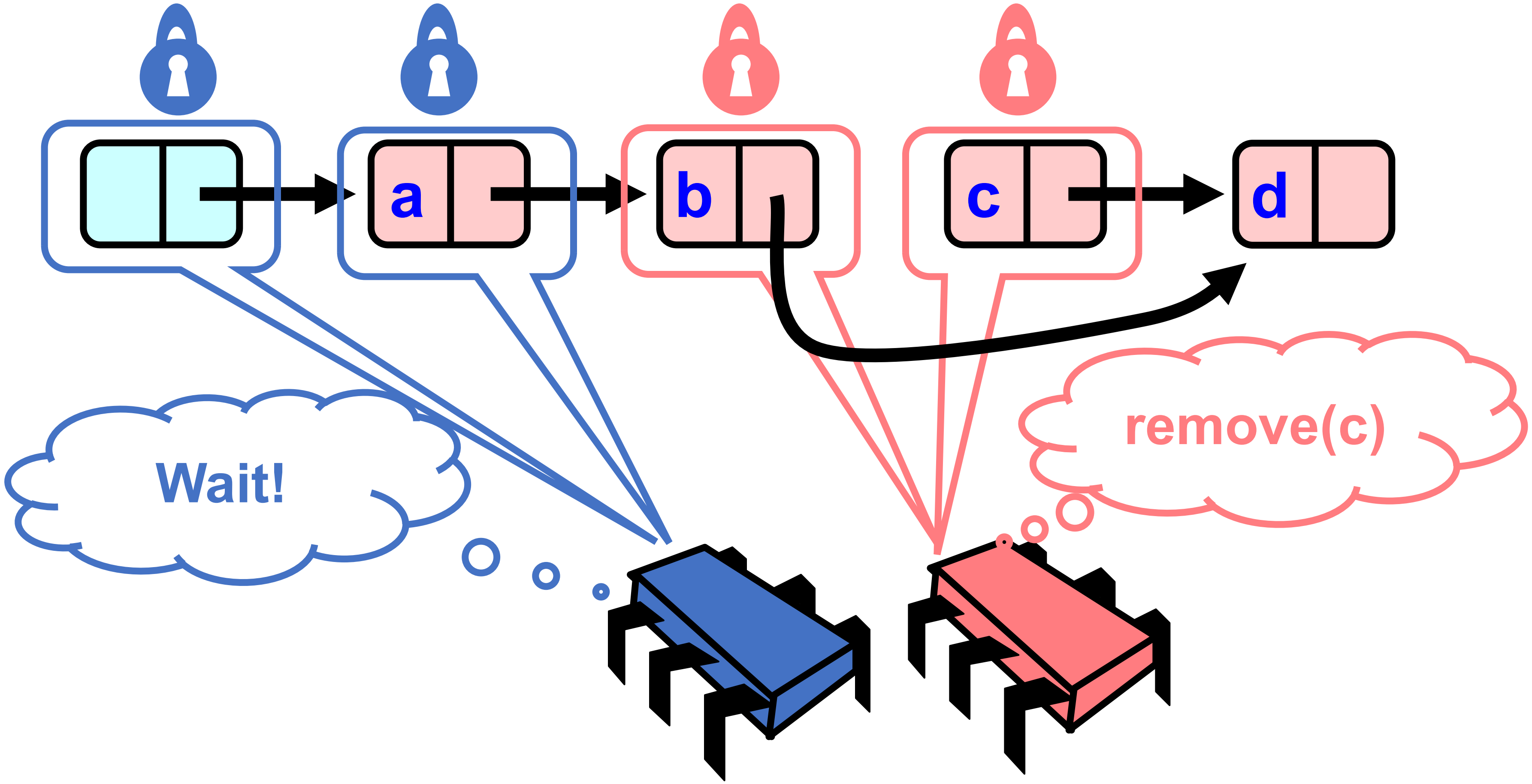
Removing a Node



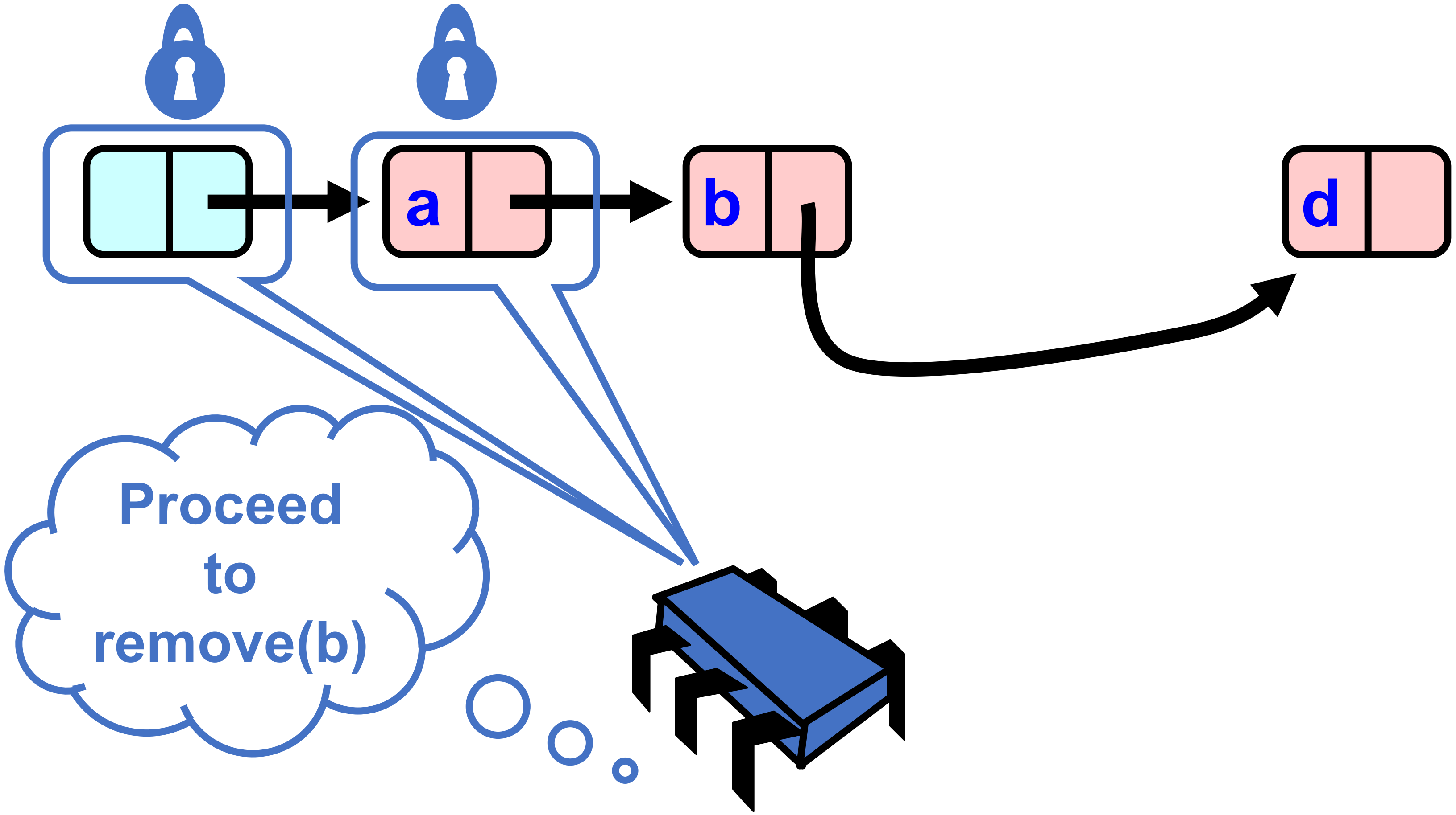
Removing a Node



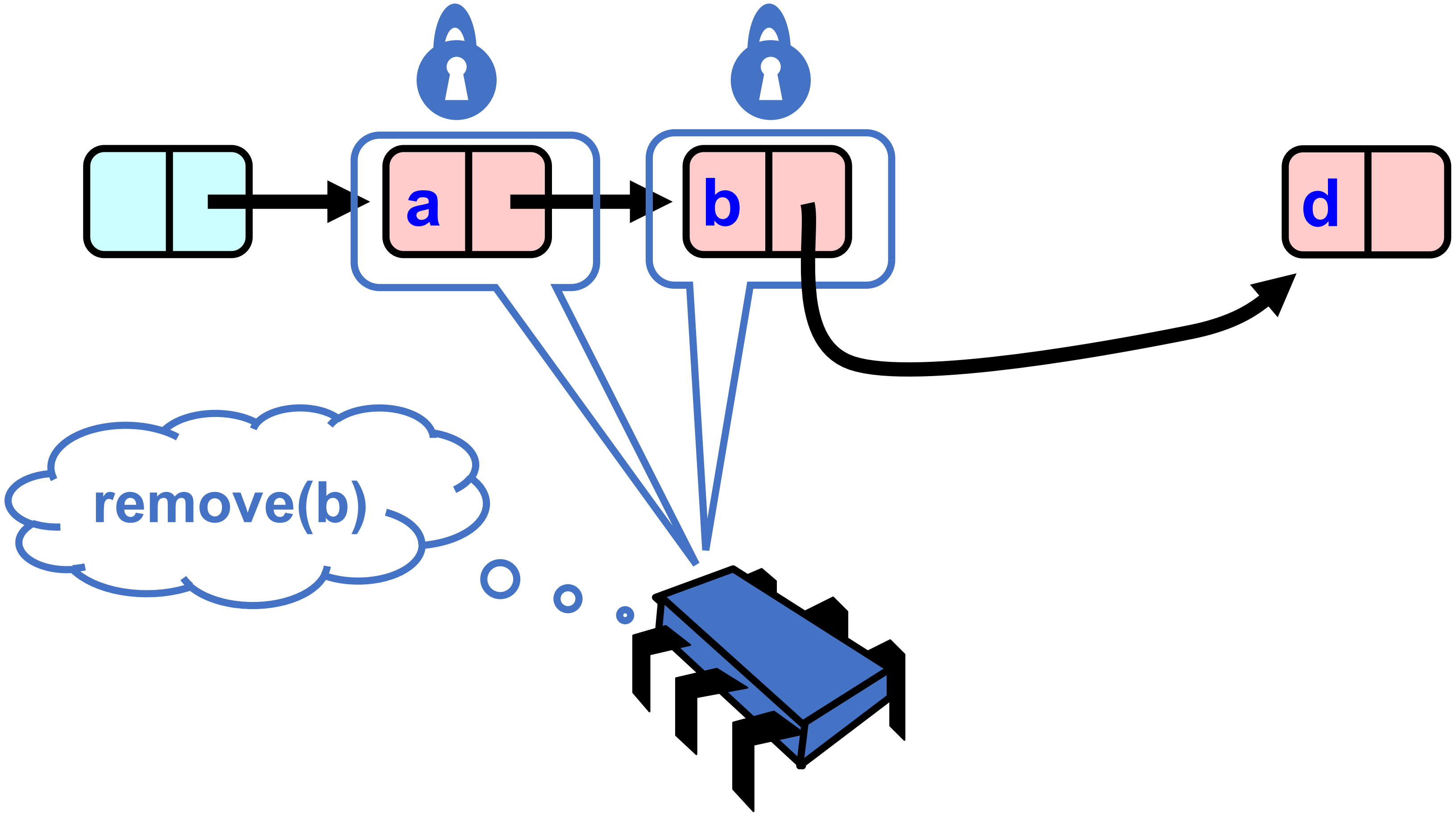
Removing a Node



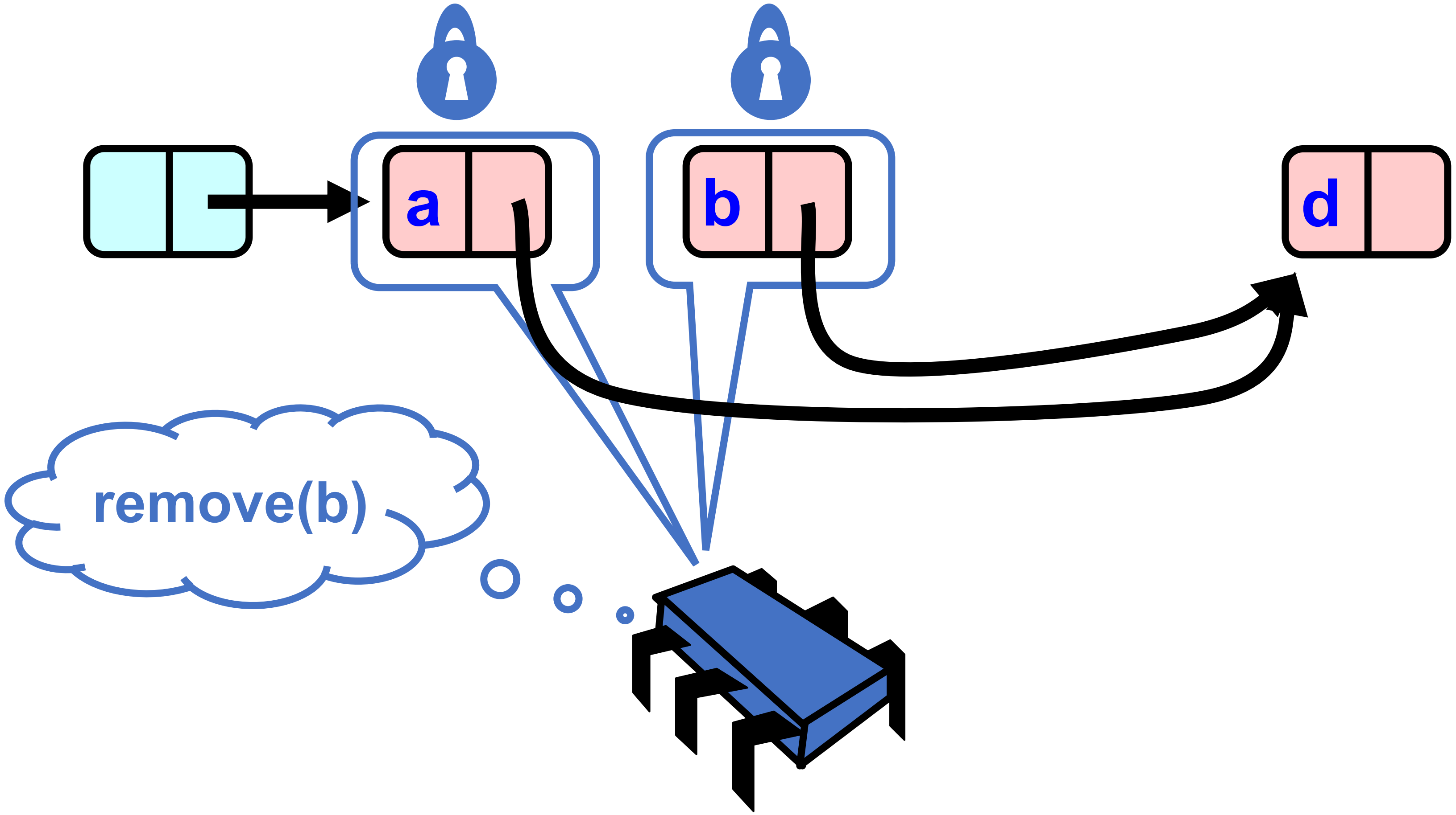
Removing a Node



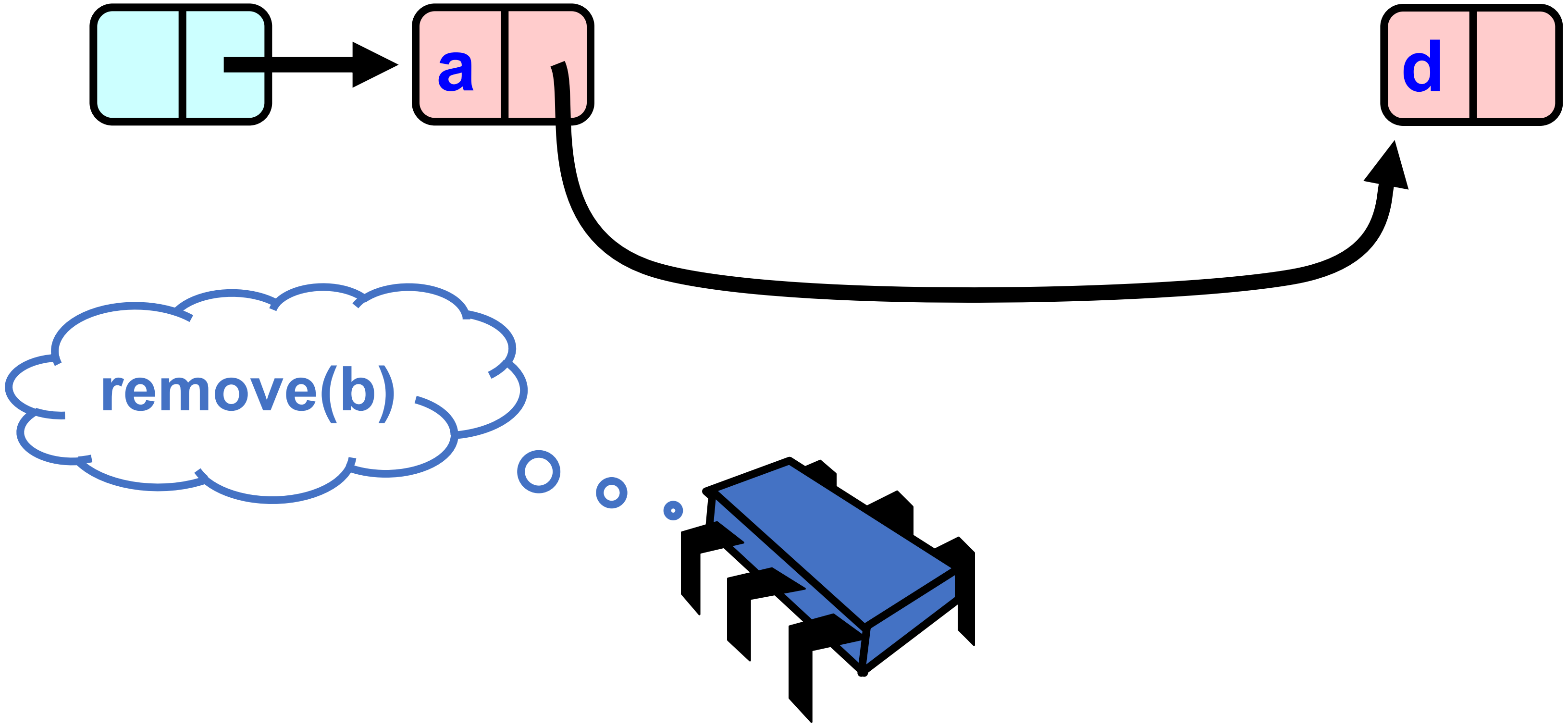
Removing a Node



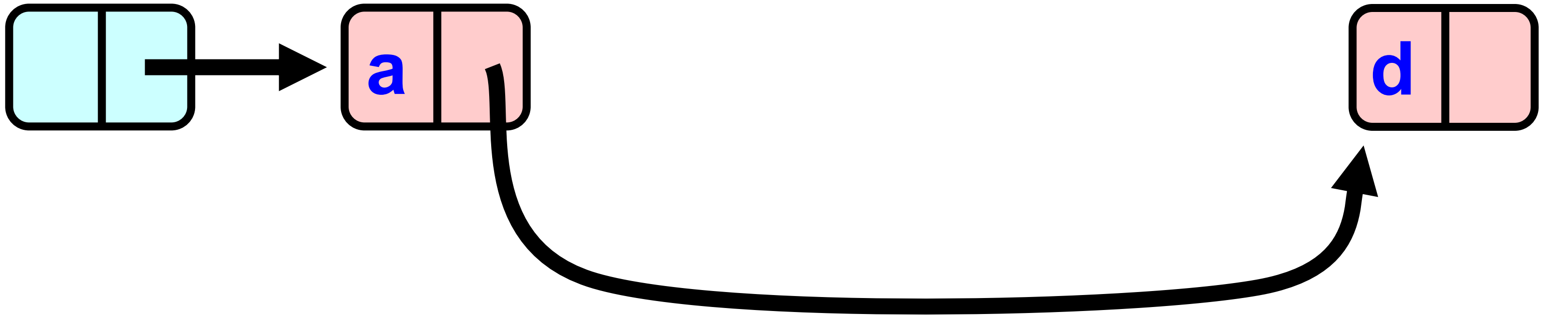
Removing a Node



Removing a Node



Removing a Node



Adding Nodes

- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted

Drawbacks

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

How can we improve

- Acquires and releases lock for every node traversed
  - If we have a long list to search, it can be bad!
  - reduces concurrency (traffic jams)

Optimistic Synchronization

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

Optimistic Synchronization

- Find nodes without locking

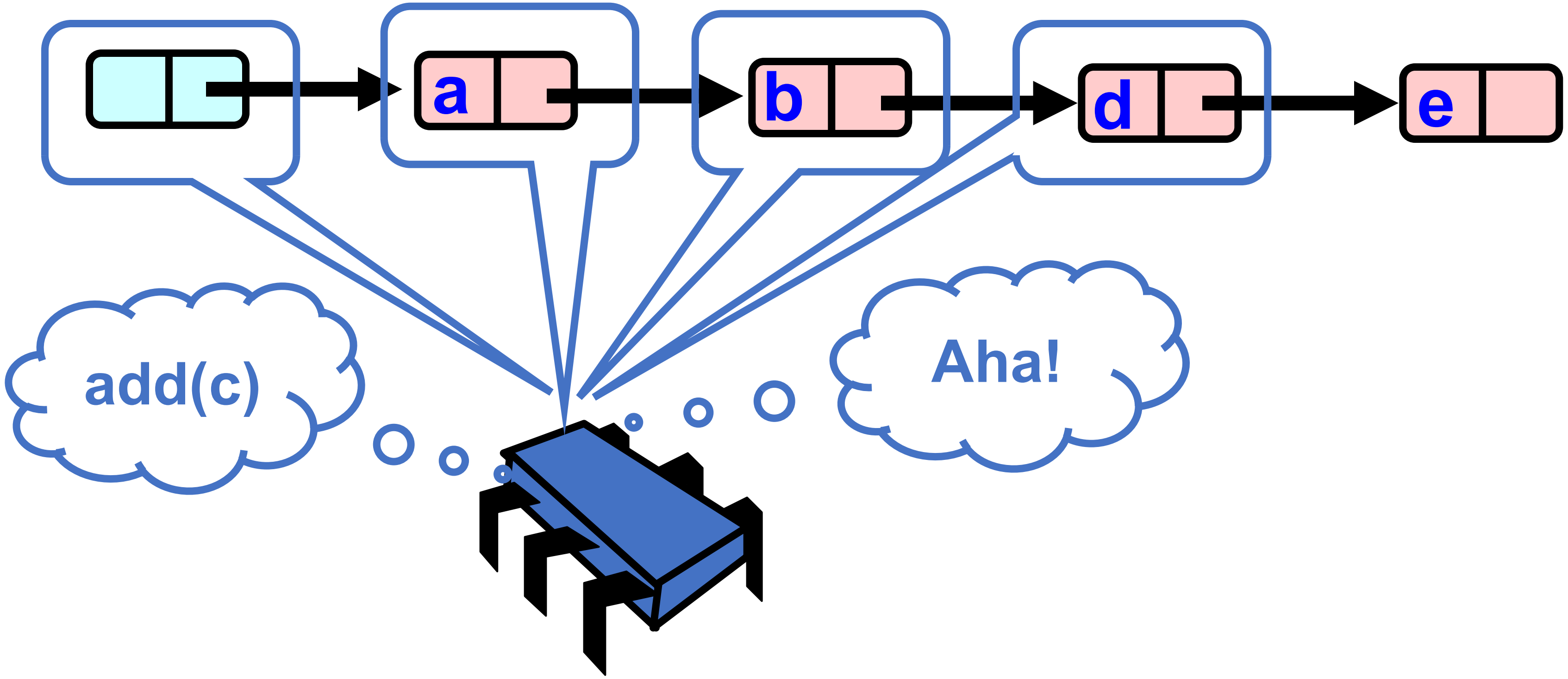
Optimistic Synchronization

- Find nodes without locking
- Lock nodes

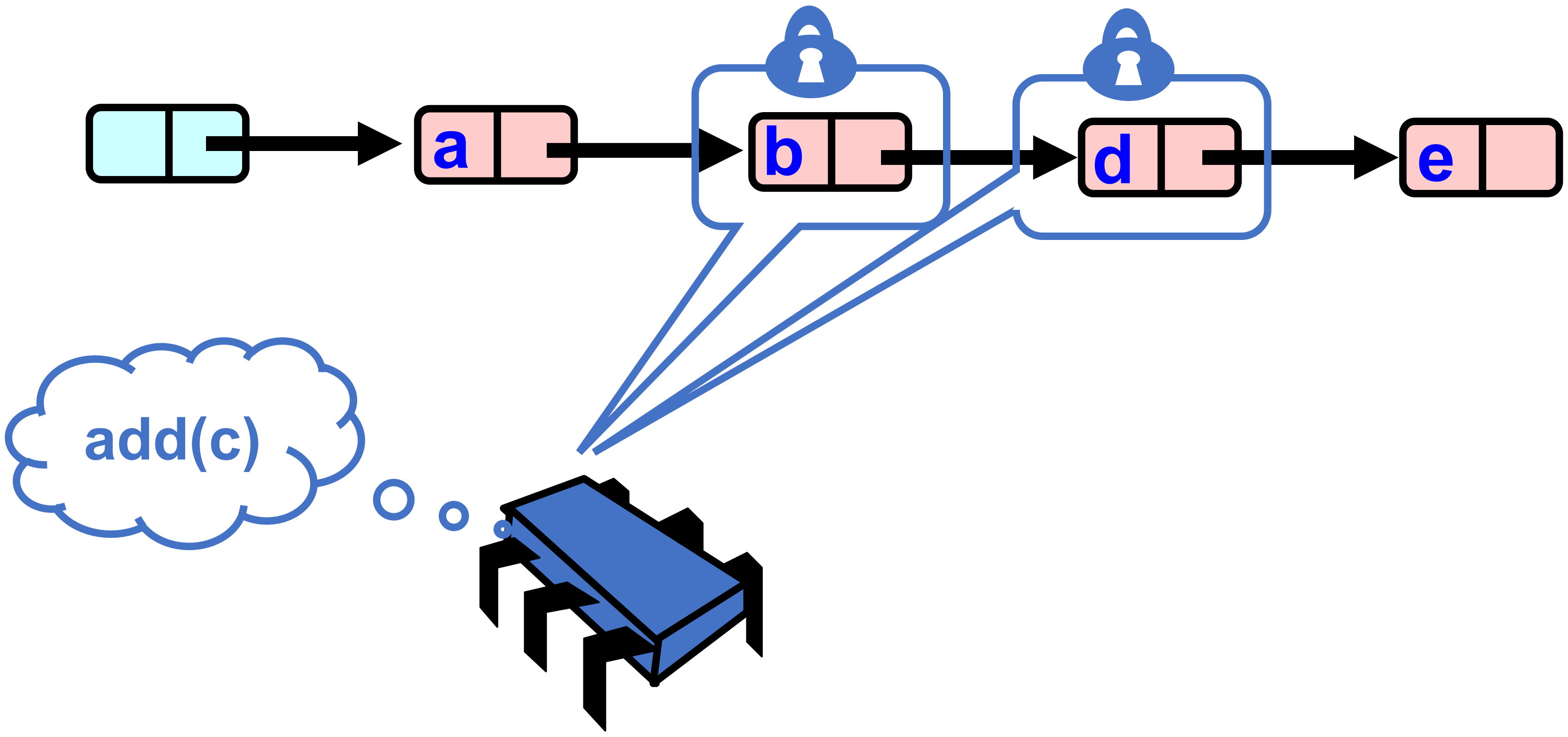
Optimistic Synchronization

- Find nodes without locking
- Lock nodes
- Check that everything is OK

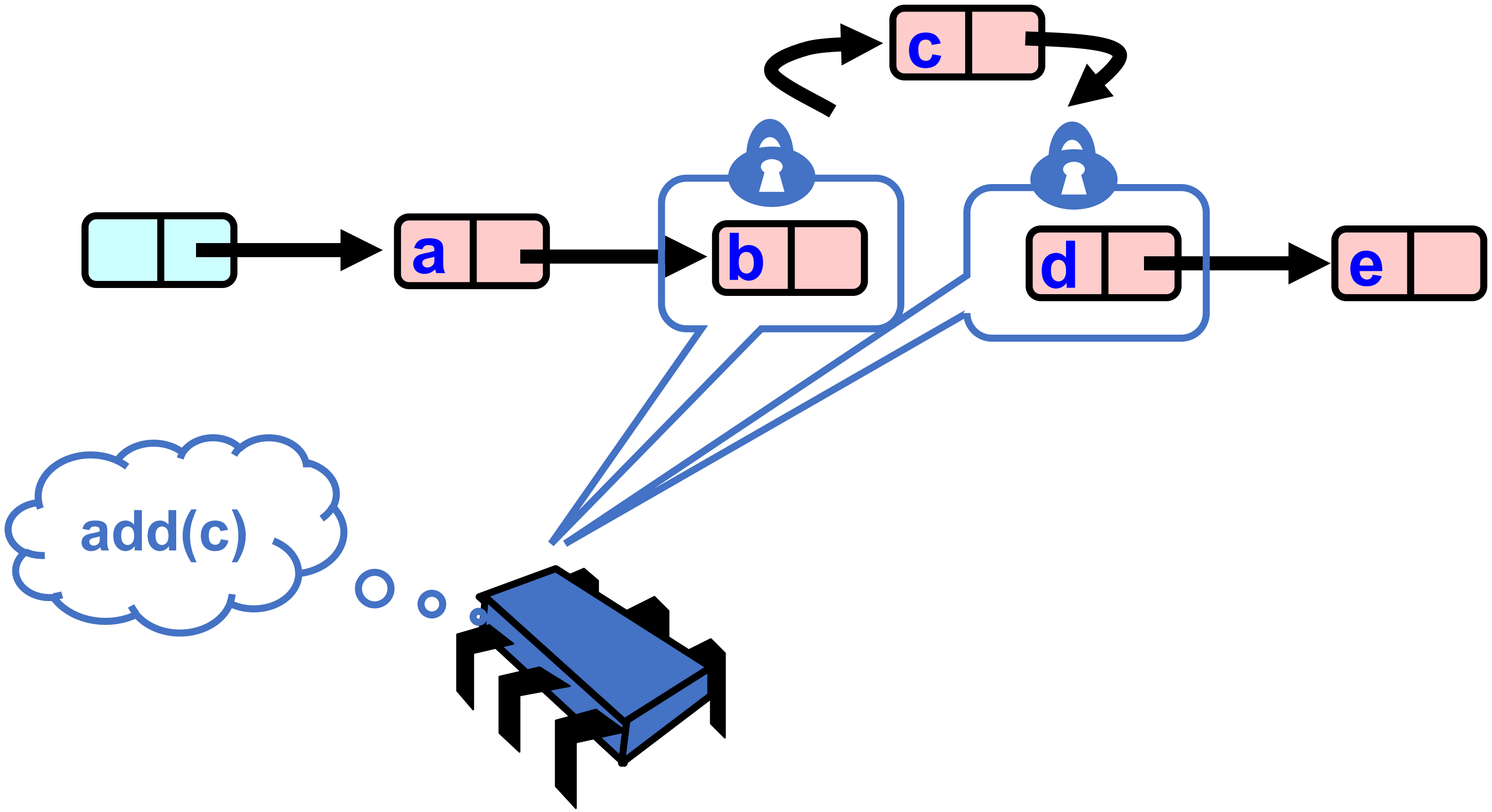
Optimistic: Traverse without Locking



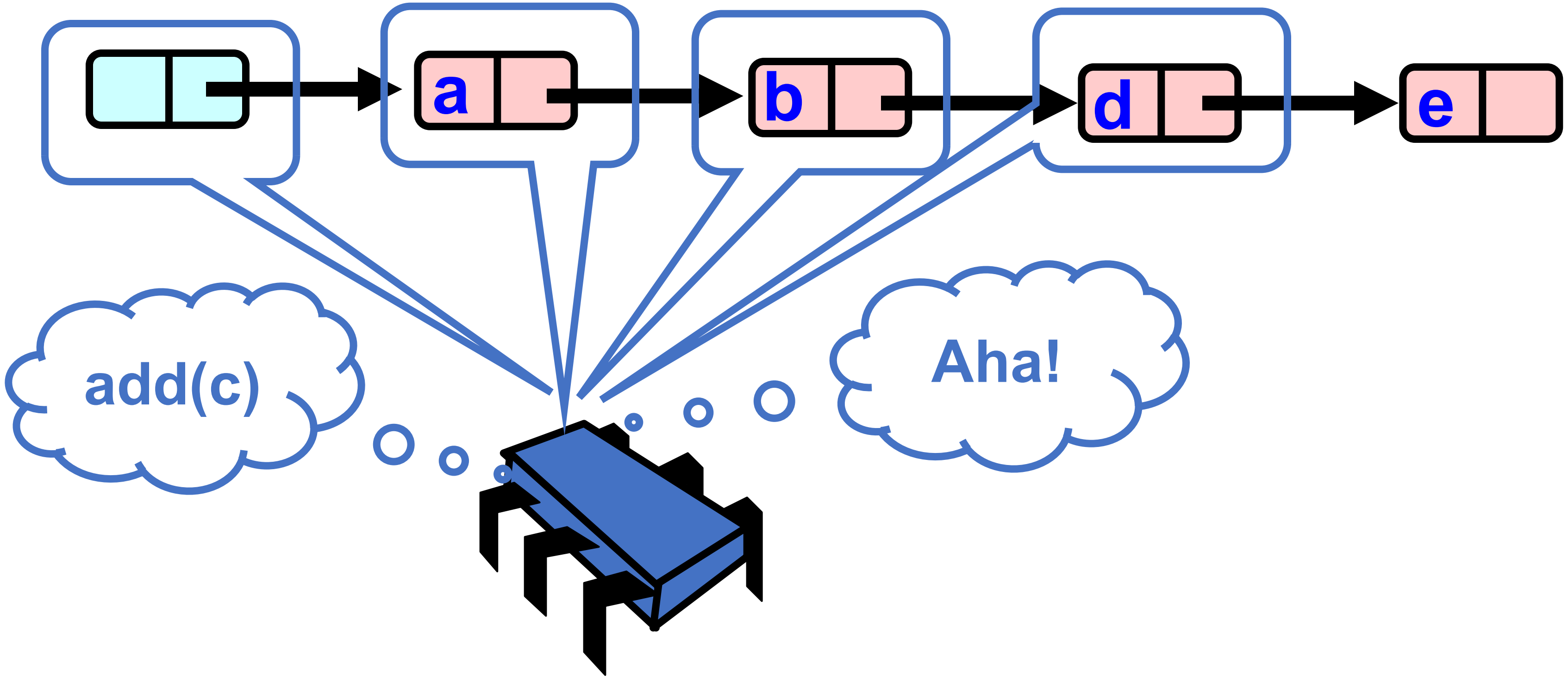
Optimistic: Lock and Load



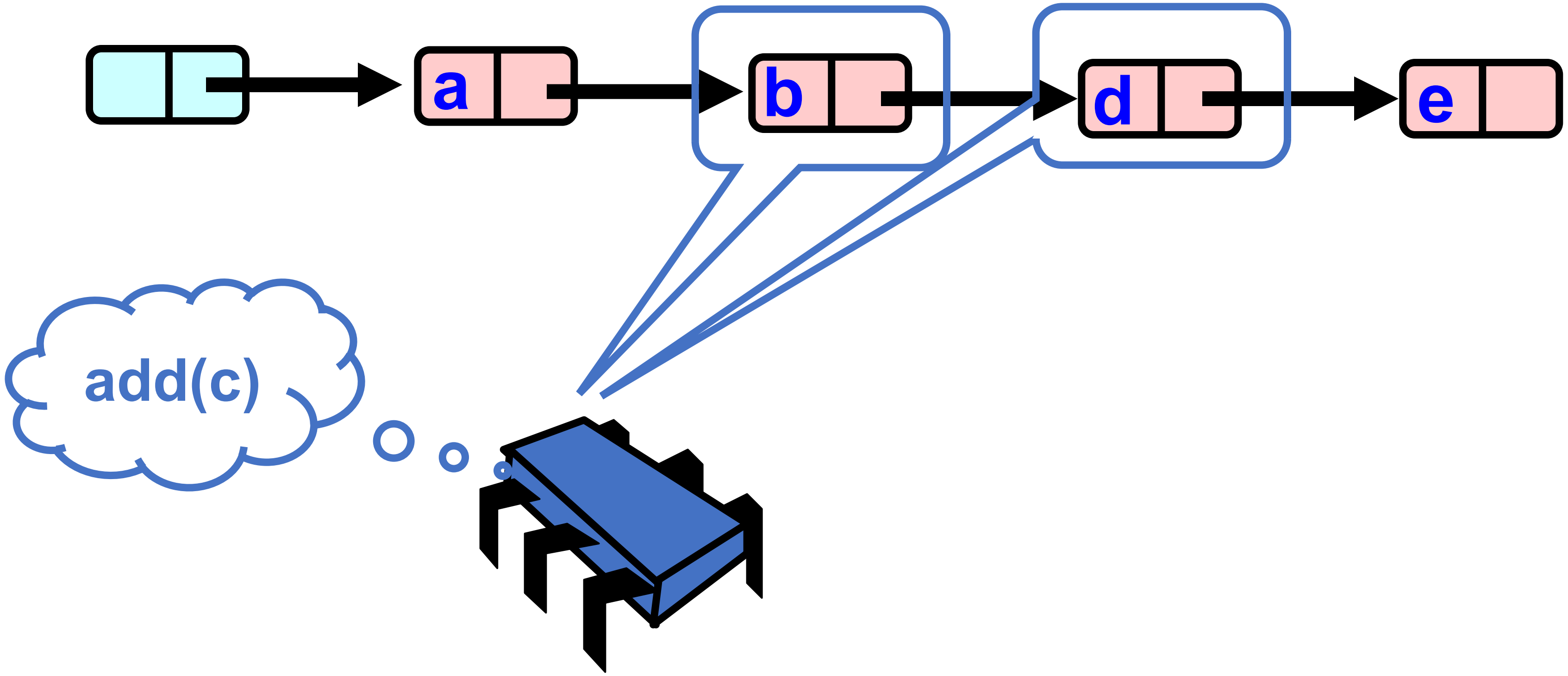
Optimistic: Lock and Load



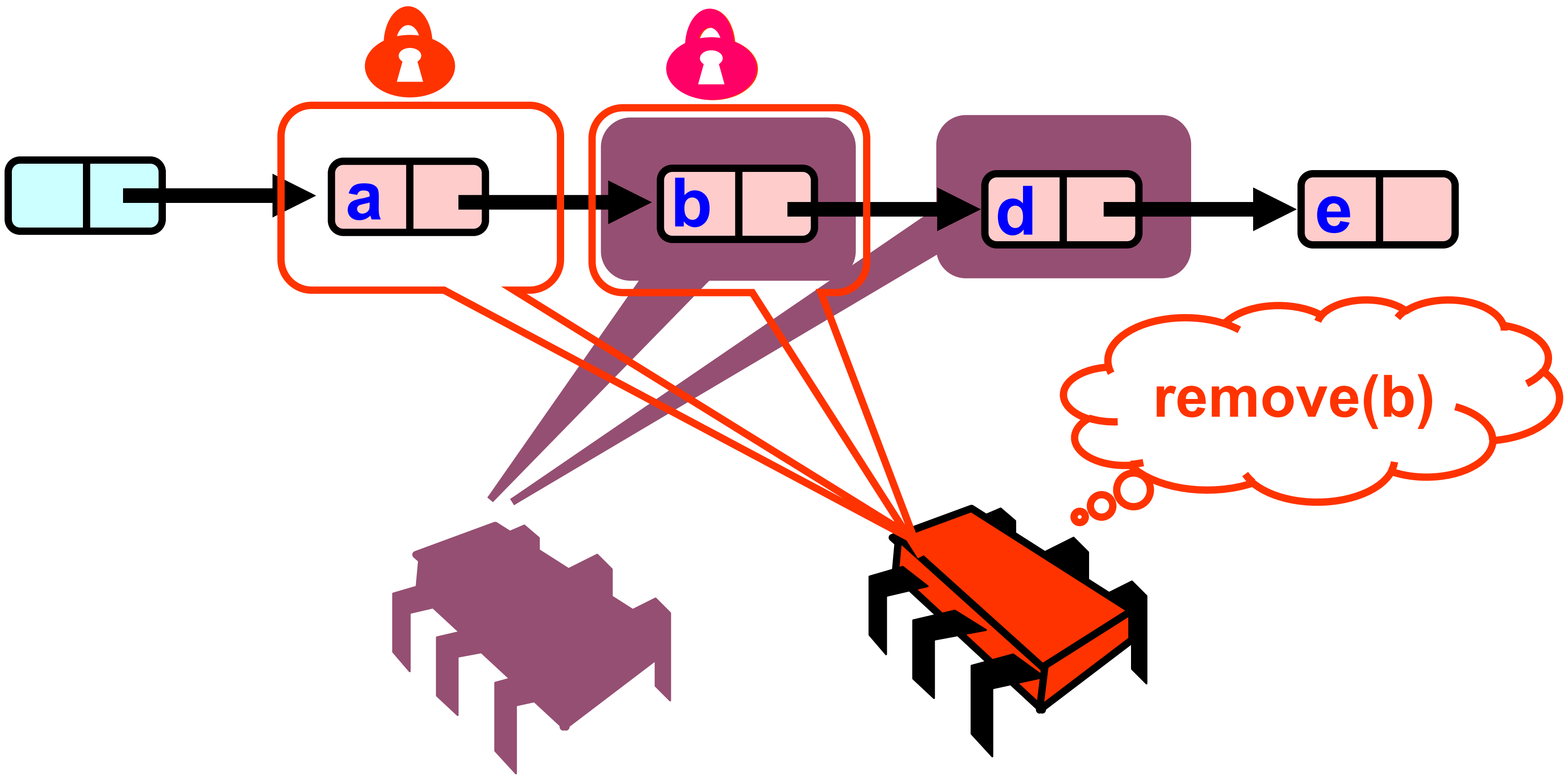
What could go wrong?



What could go wrong?



What could go wrong?



Data conflict!

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

- What do we do?

Data conflict!

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

- What do we do? We decided that locking when traversing is too expensive.

Lock-free reasoning

- We can use atomic variables

Lock-free reasoning

- Default atomic accesses are documented to be sequentially consistent.

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

Lock-free reasoning

- Default atomic accesses are documented to be sequentially consistent.

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

Create an atomic pointer type using C++ templates

Atomic template in C++

- demo

Lock-free reasoning

- Default atomic accesses are documented to be sequentially consistent.

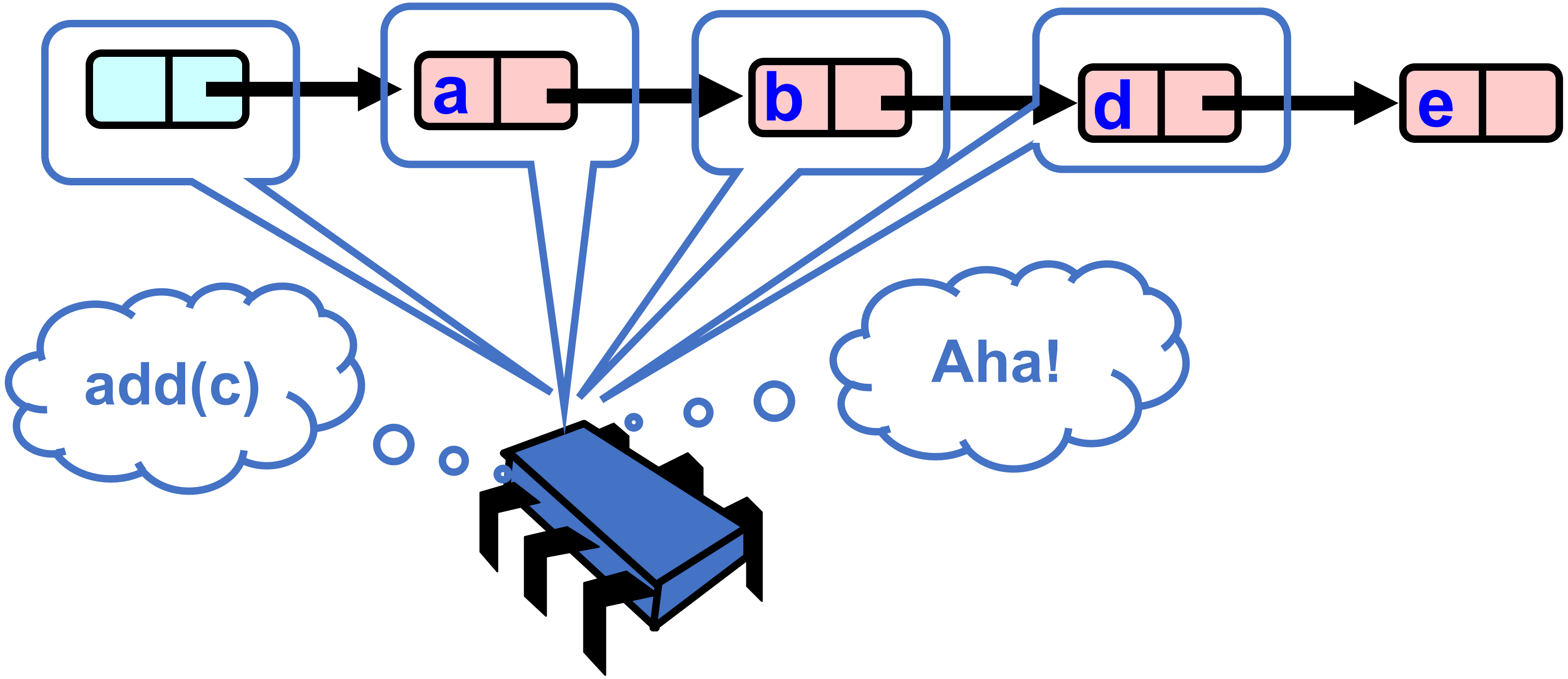
```
void traverse(node *n) {
  while (n->next != NULL) {
    n = n->next;
  }
}
```

Lock-free reasoning

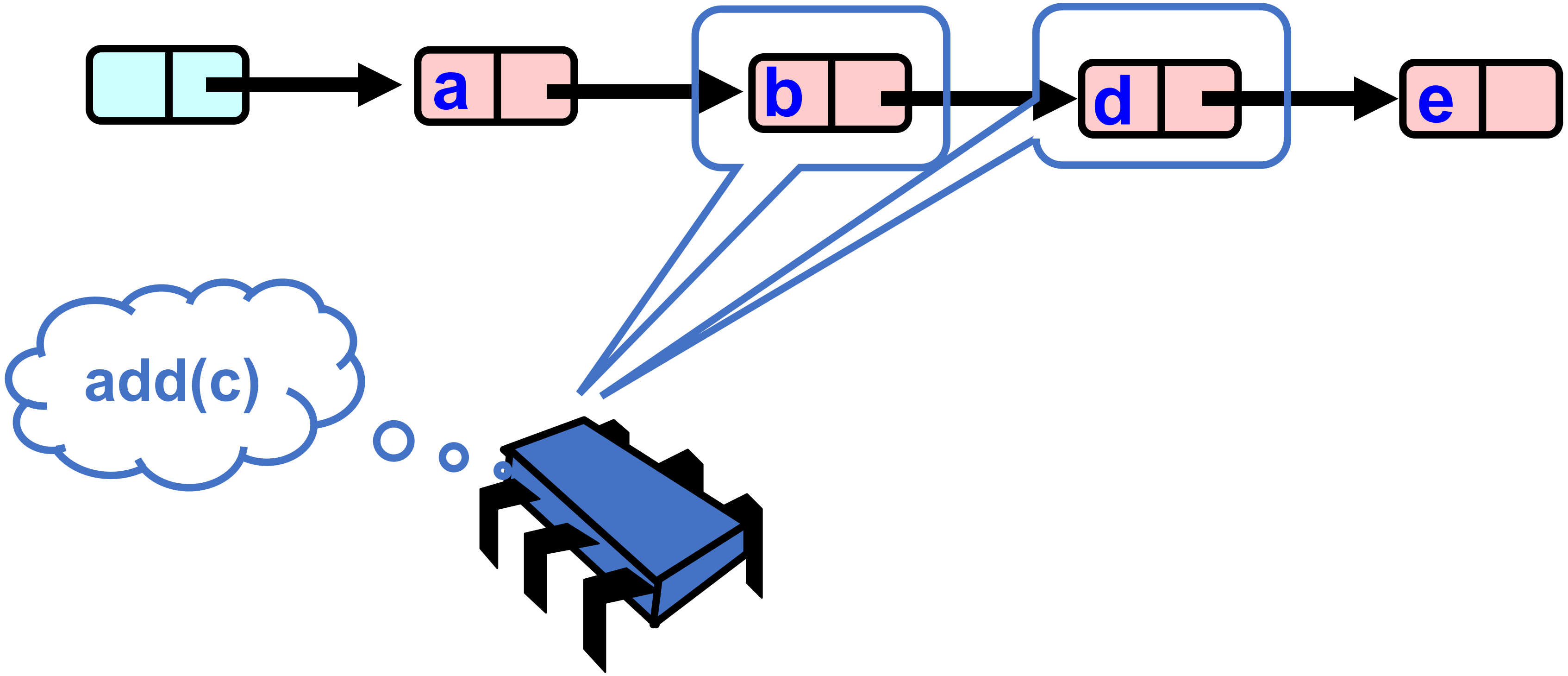
- Default atomic accesses are documented to be sequentially consistent.

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

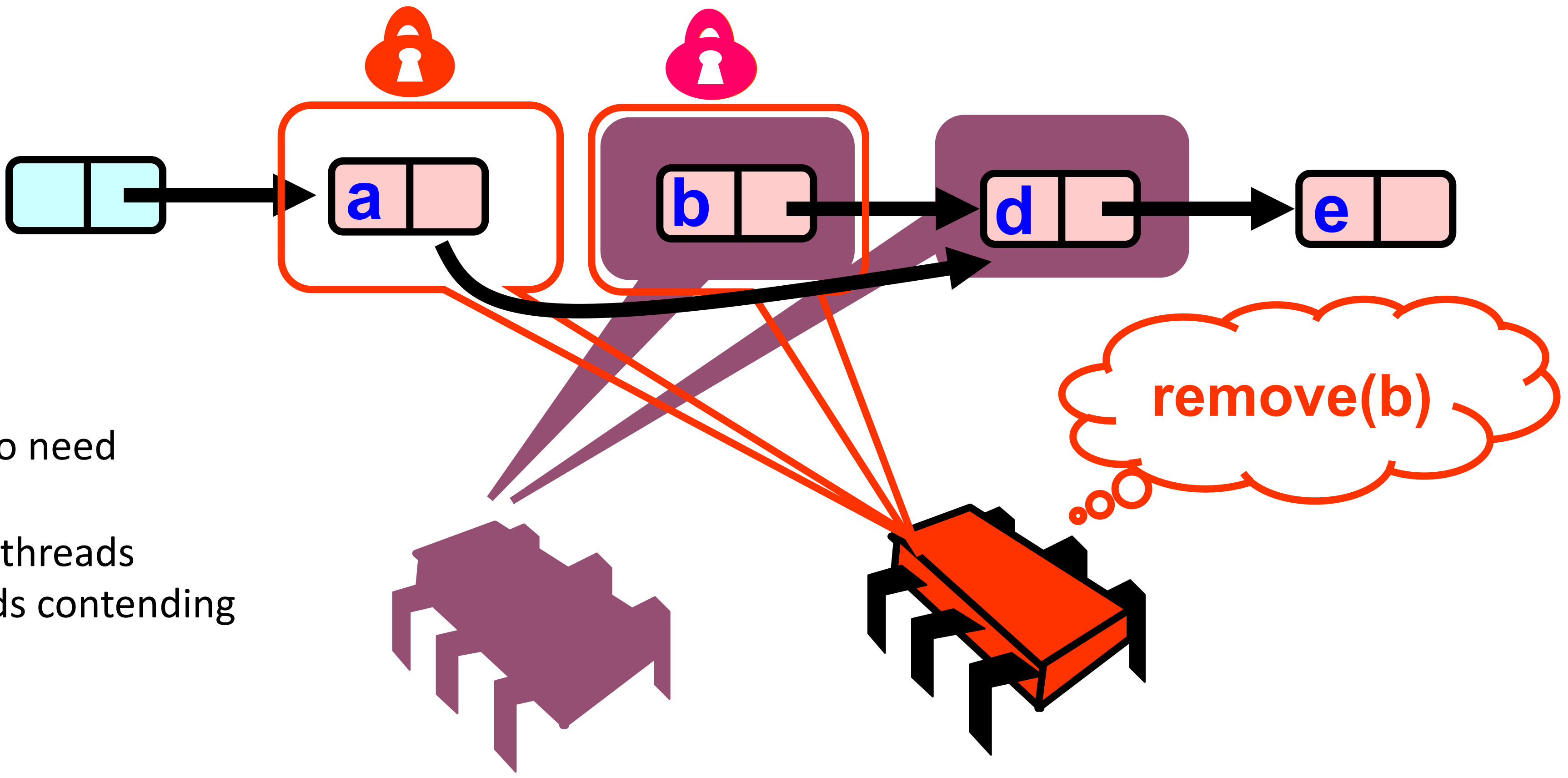
What could go wrong?



What could go wrong?

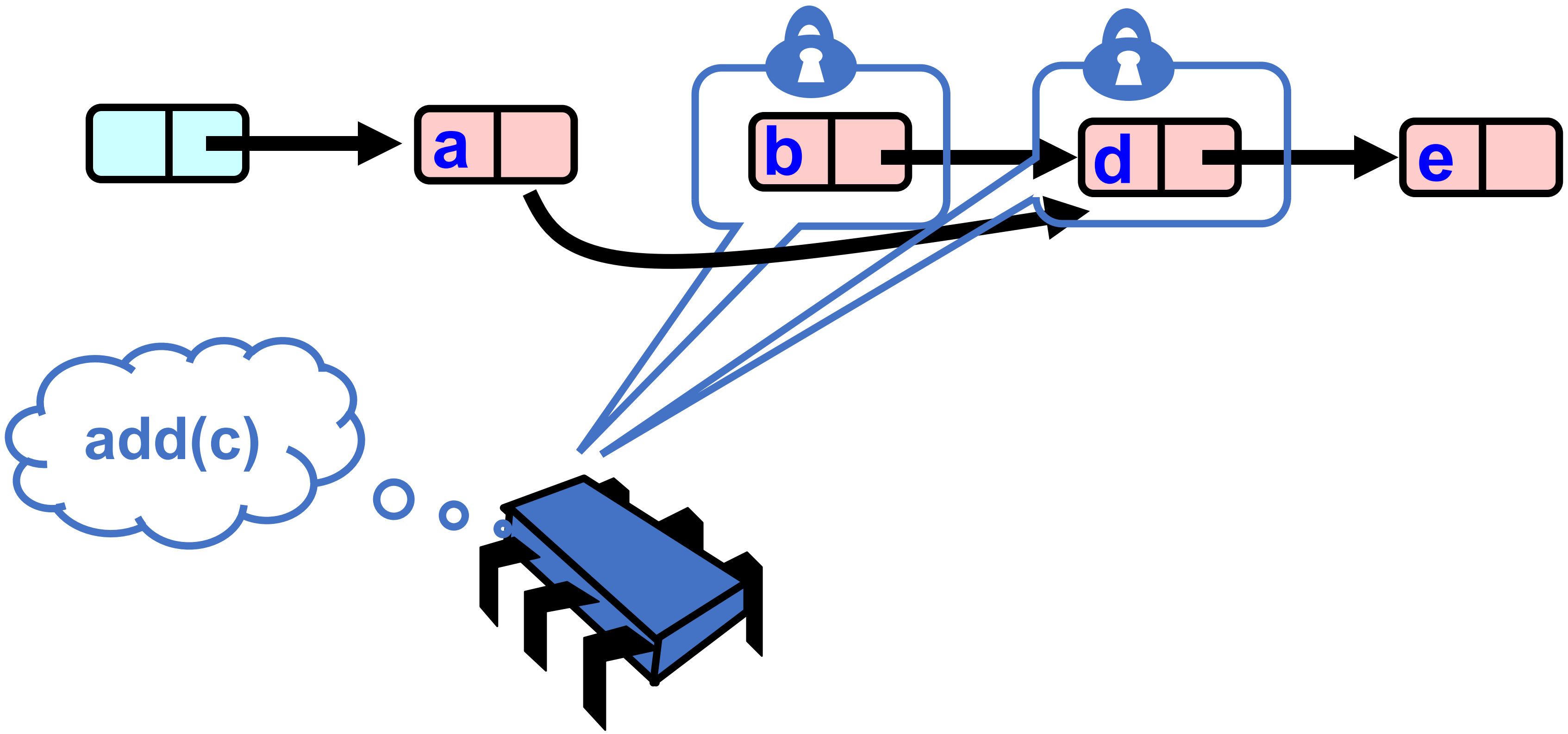


What could go wrong?

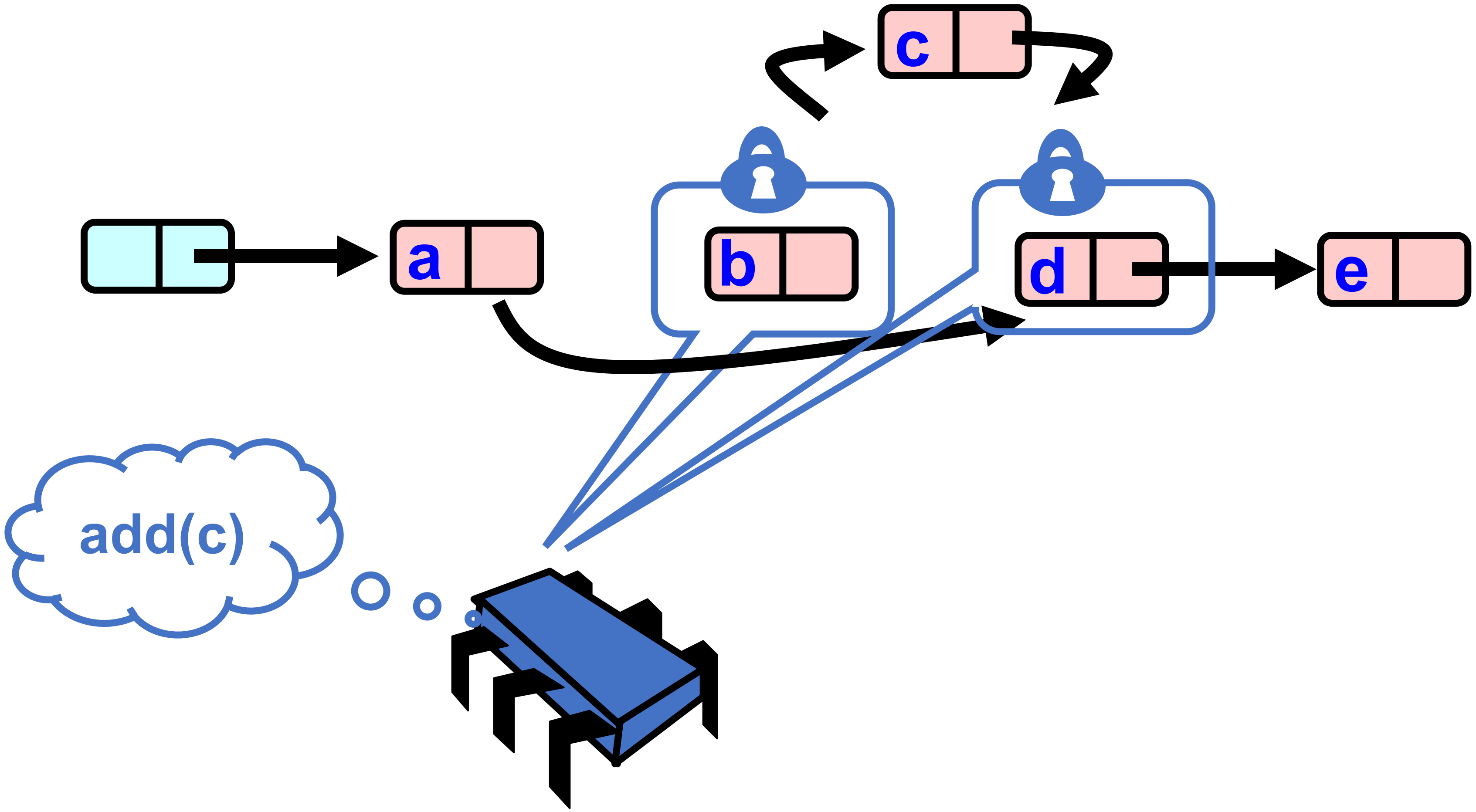


No more data
conflict, but we do need
to reason about
interleavings and threads
concurrent threads contending
for values.

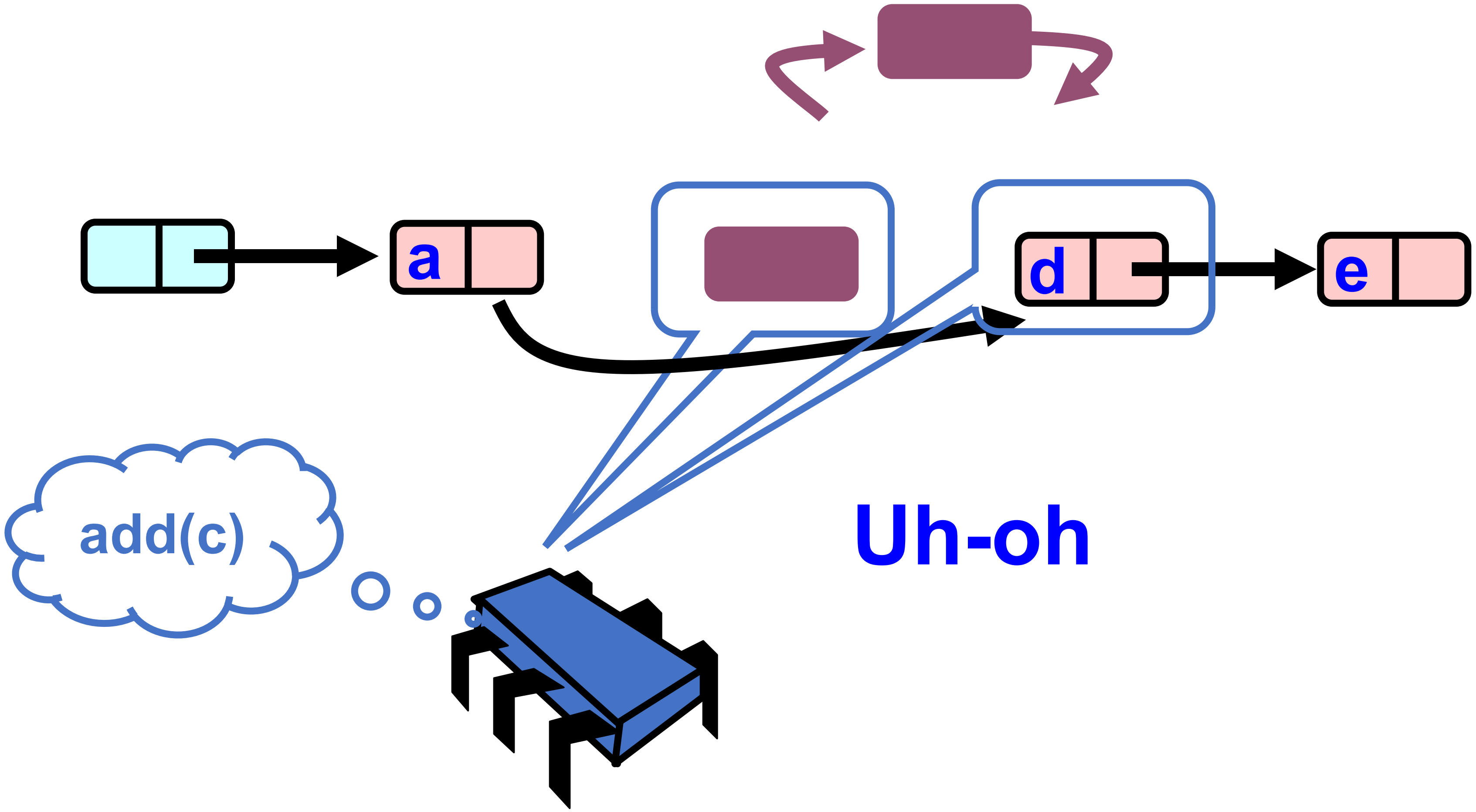
What could go wrong?



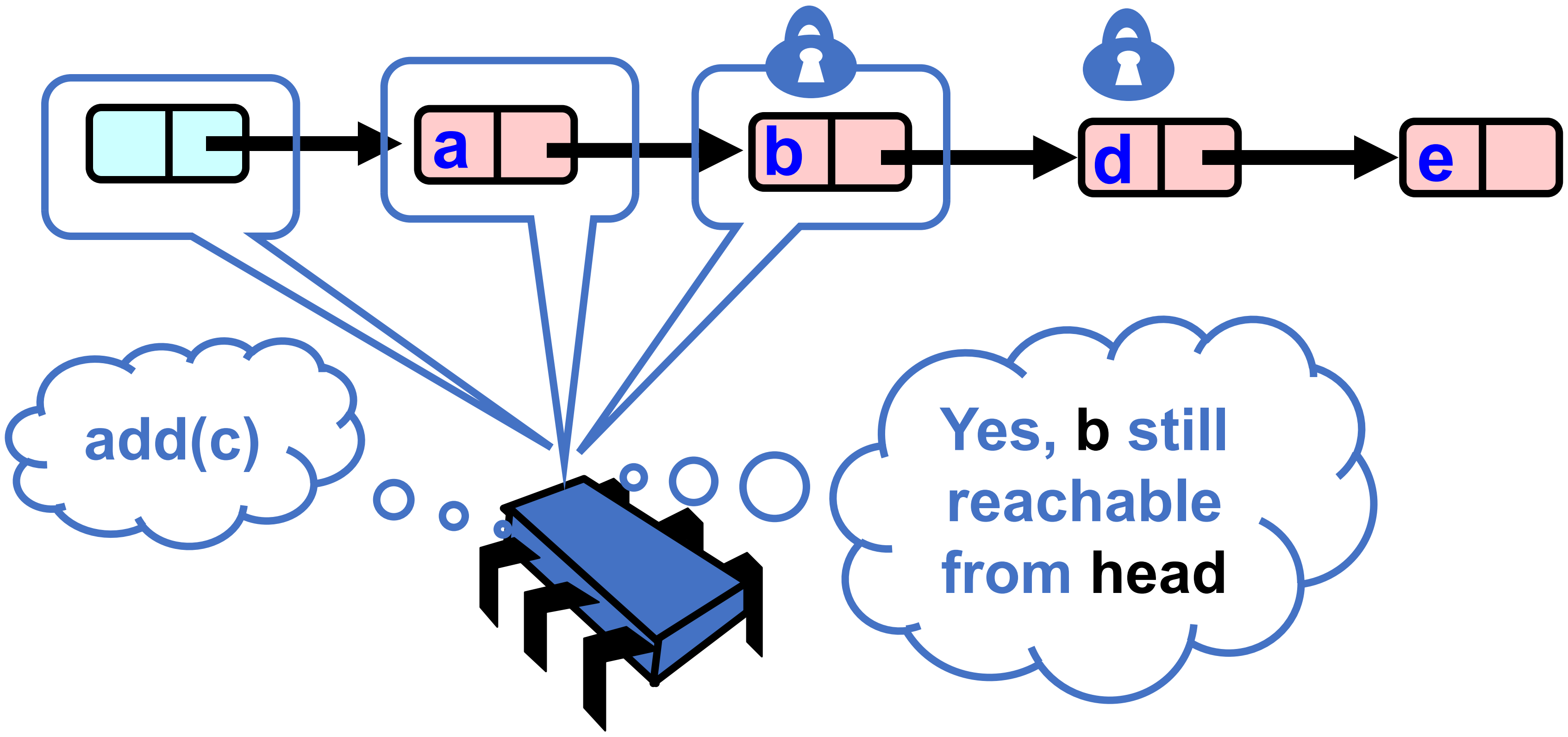
What could go wrong?



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!

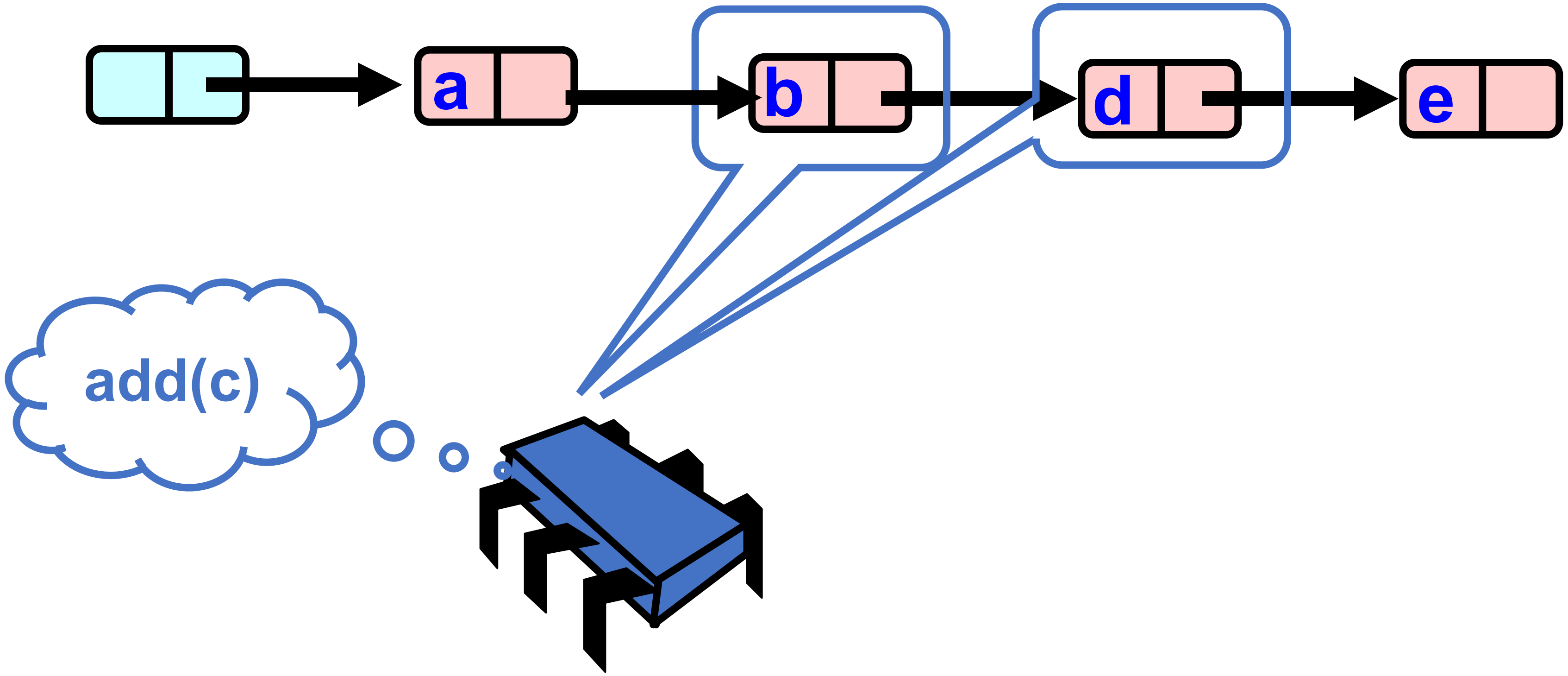
What happens if failure?

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

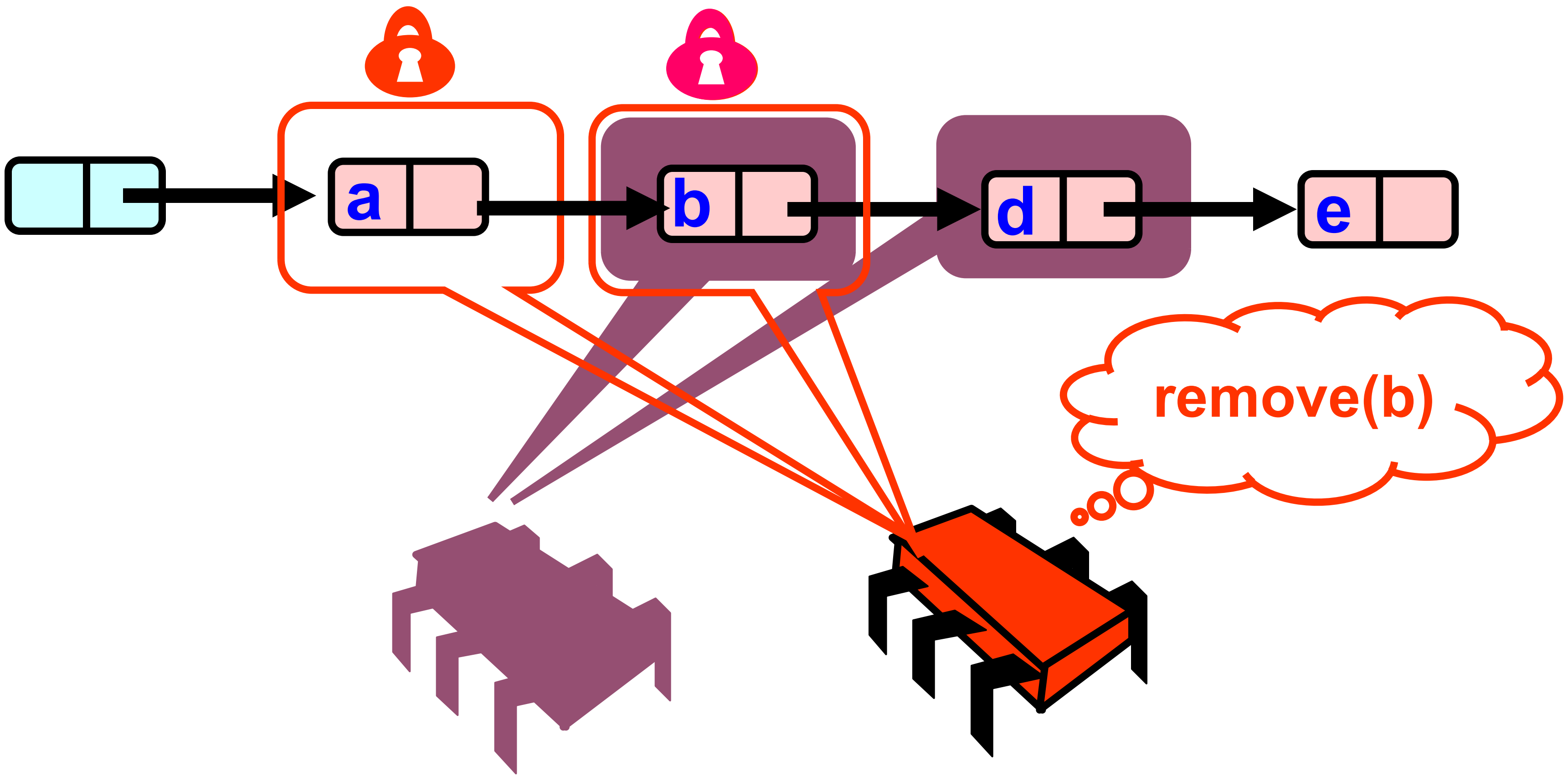
- Private method:
  - try_remove
  - remove loops on try_remove until it succeeds

What about deletion?

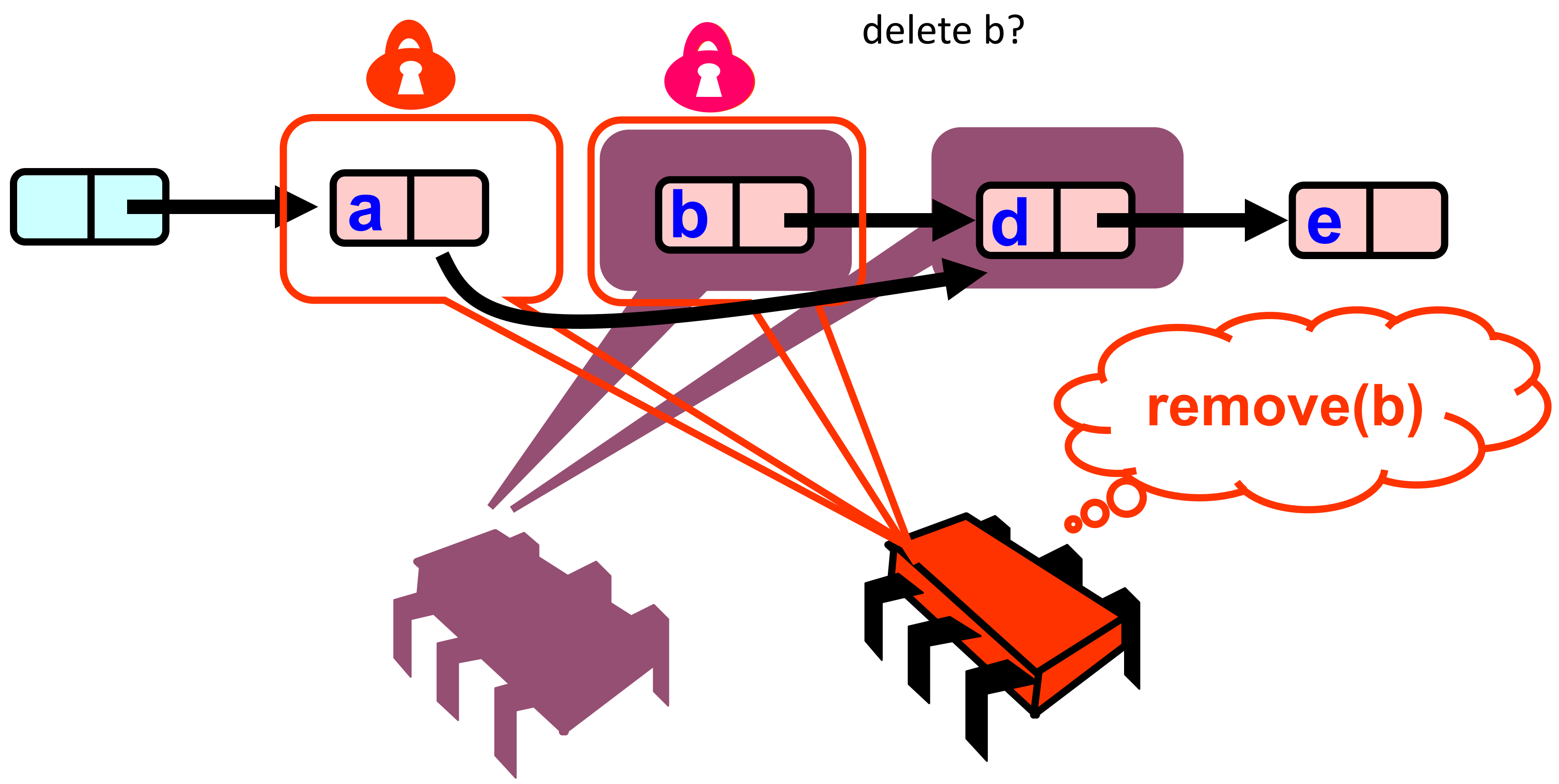
Can threads that remove a node delete it?



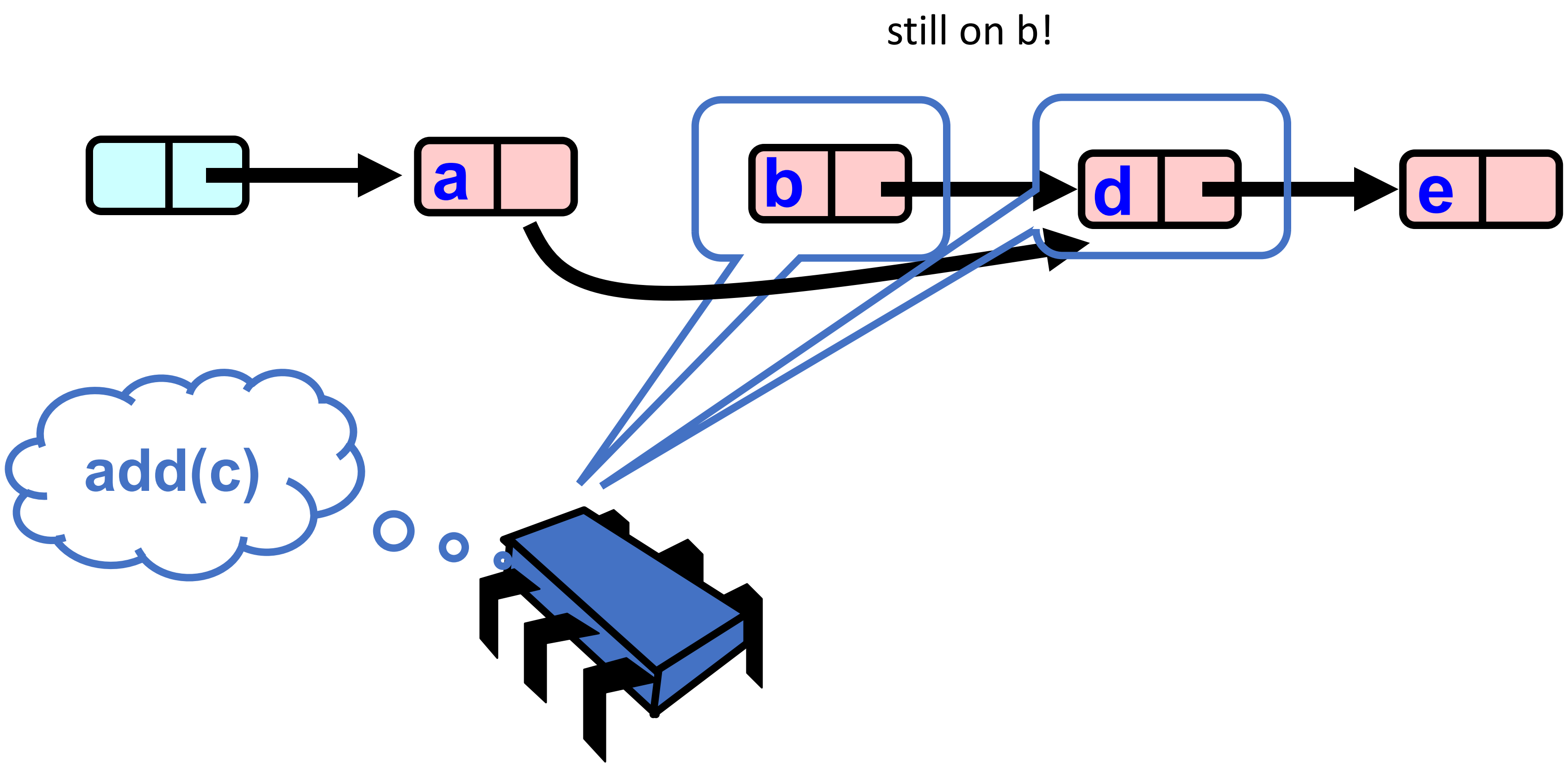
Can threads that remove a node delete it?



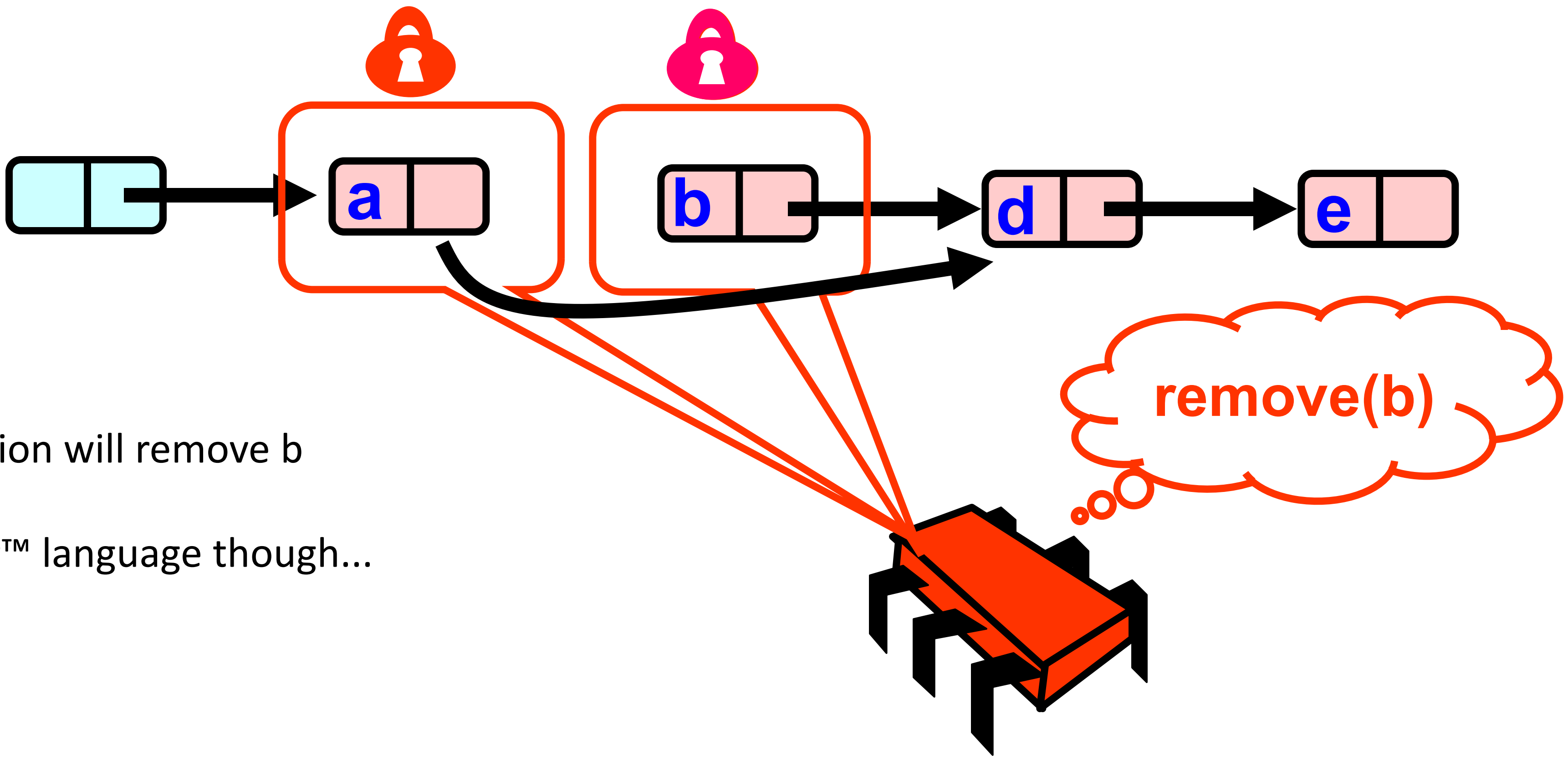
Can threads that remove a node delete it?



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...

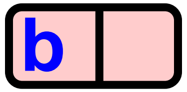
maintain a list to delete:

Our own garbage collector

Java's garbage collection will remove b

We are using a better™ language though...

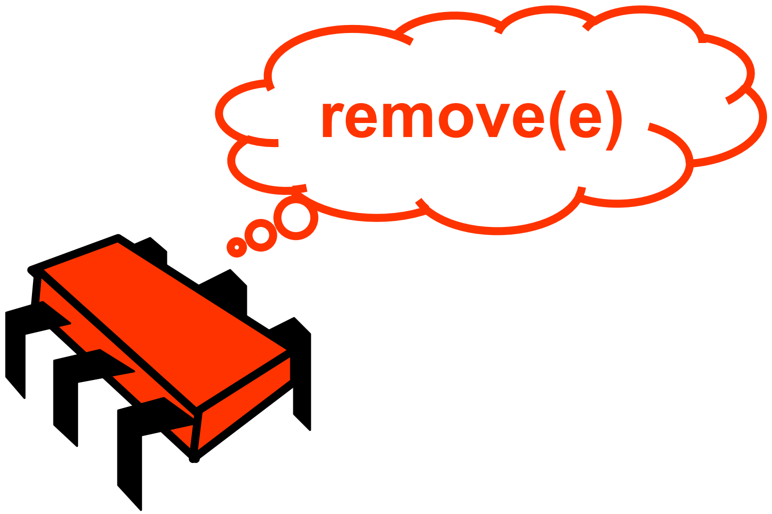
maintain a list to delete:

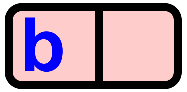


Our own garbage collector

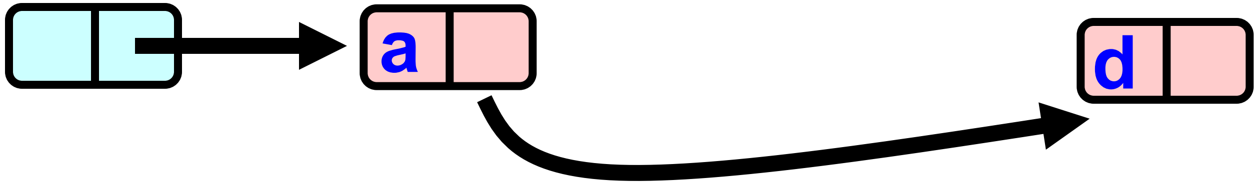
Java's garbage collection will remove b

We are using a better™ language though...



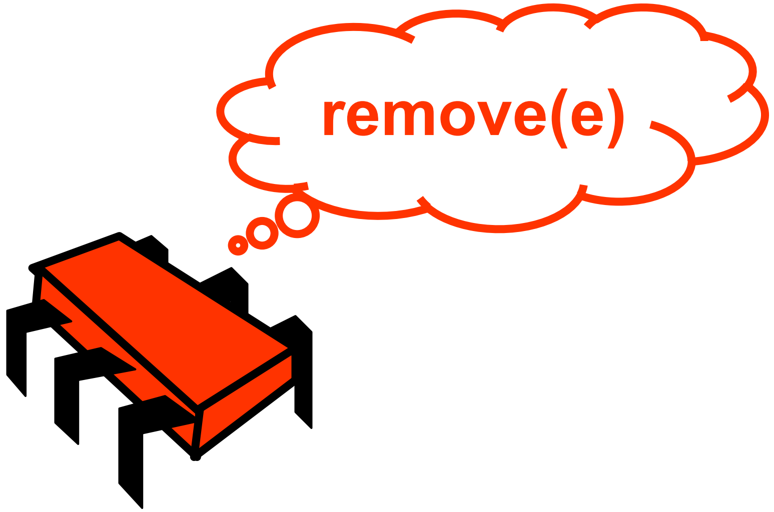
maintain a list to delete: 

Our own garbage collector

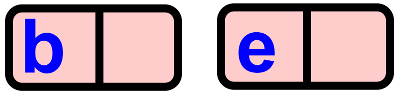


Java’s garbage collection will remove b

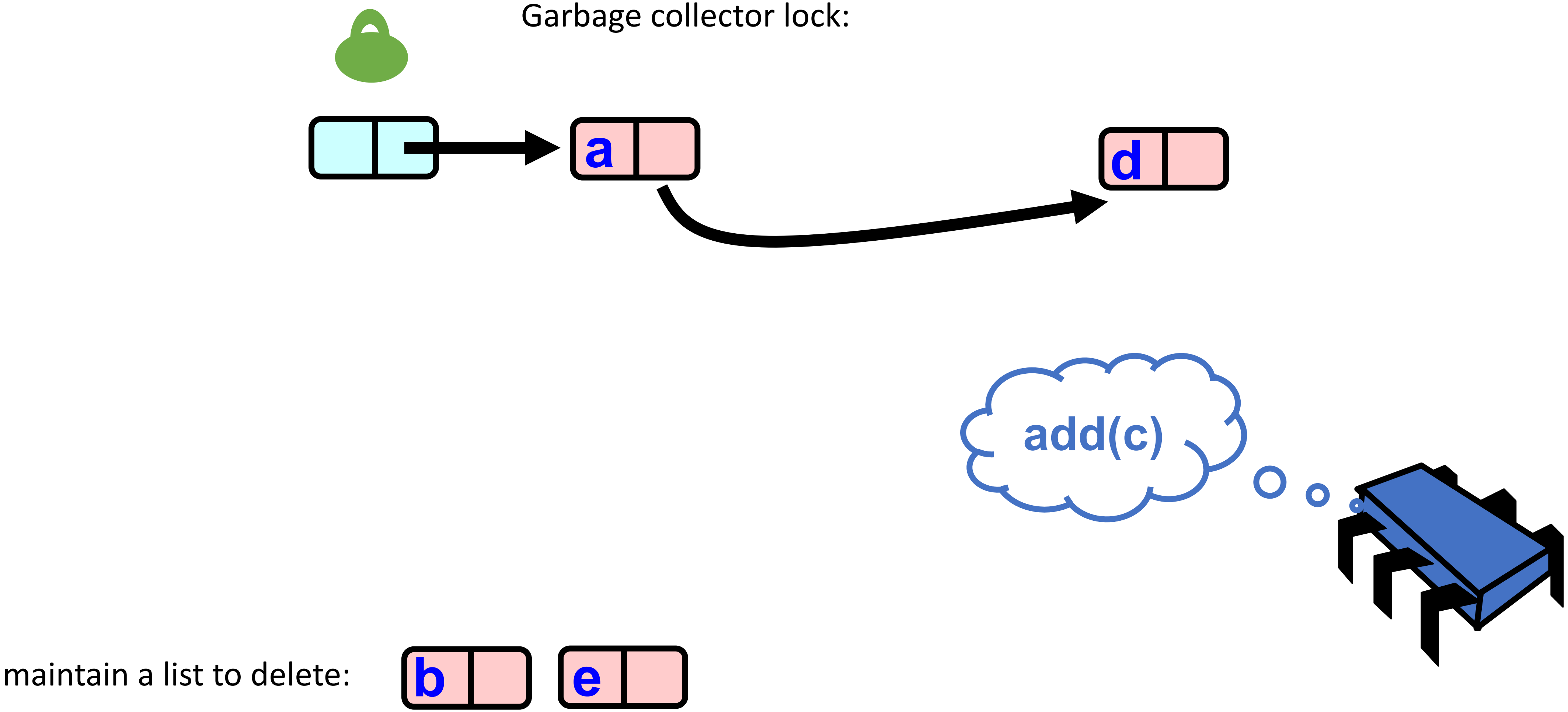
We are using a better™ language though...



maintain a list to delete:

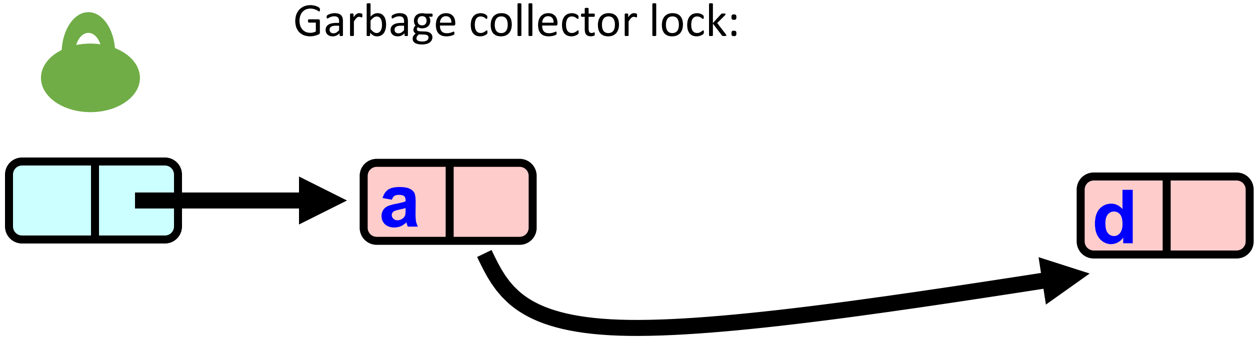


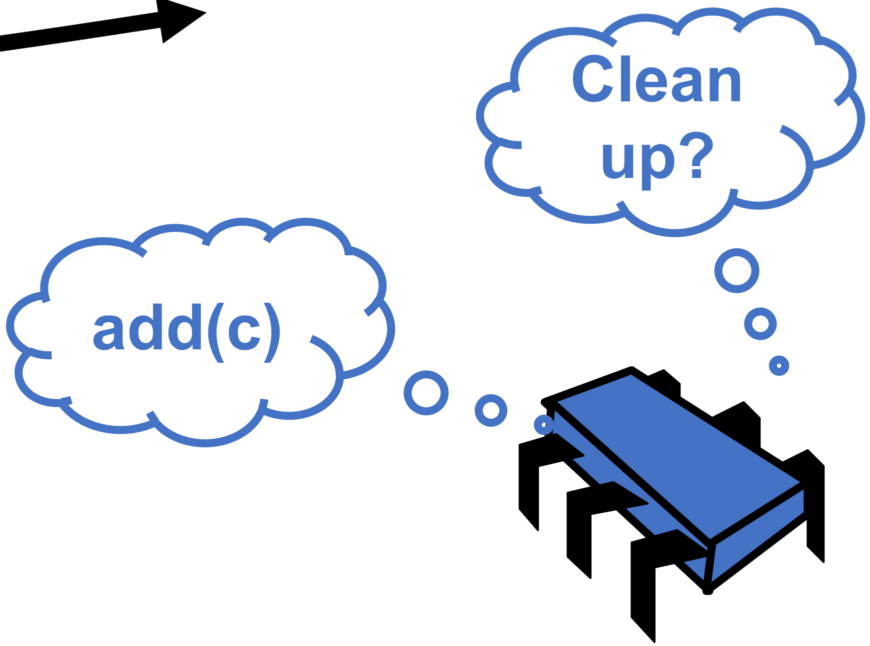
Our own garbage collector



Our own garbage collector

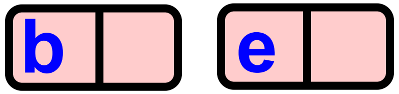
Garbage collector lock:





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:



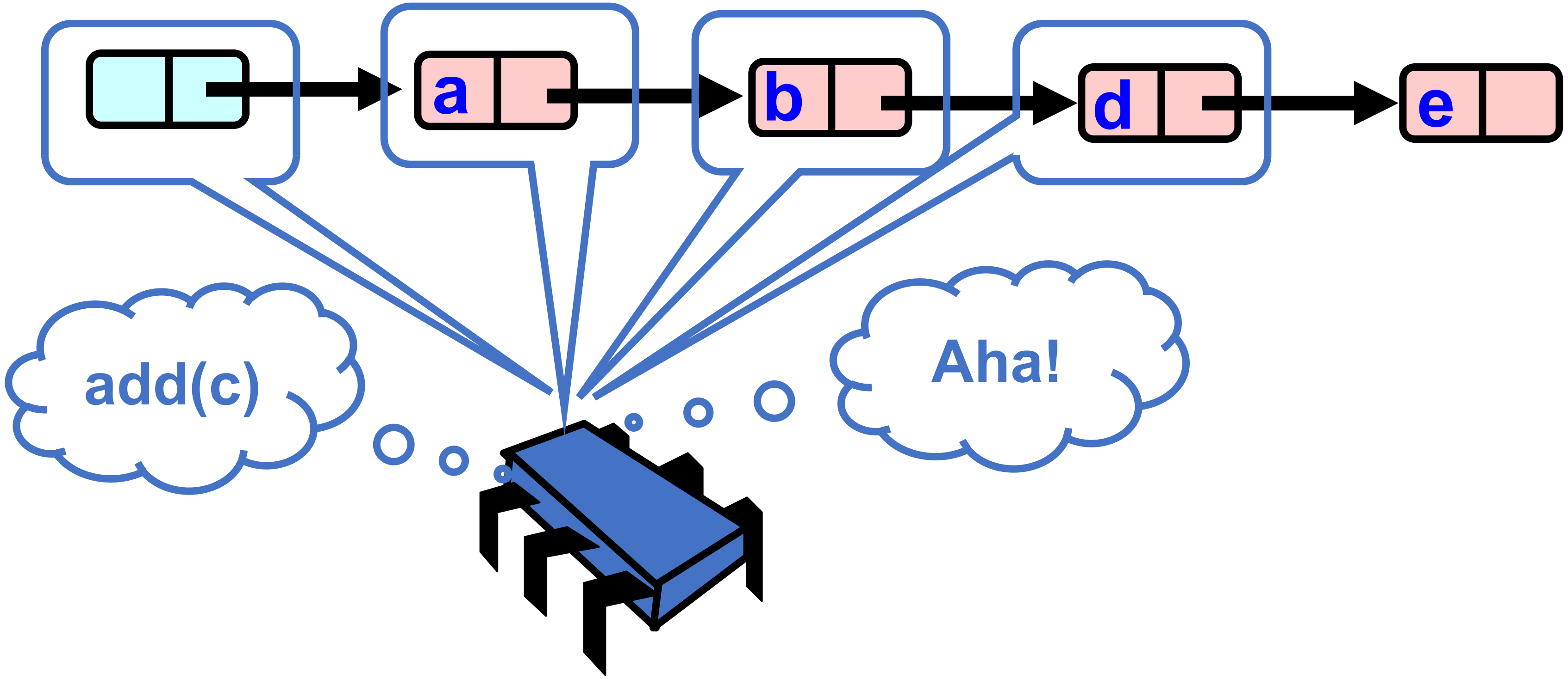
Garbage collector lock

- 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
  - if succeeded, then there was no contention
  - can starve garbage collection
- **Strat 2:** Wait until size grows to a threshold:
  - Wait on the lock (hope for a fair implementation!)
  - Can cause performance spikes

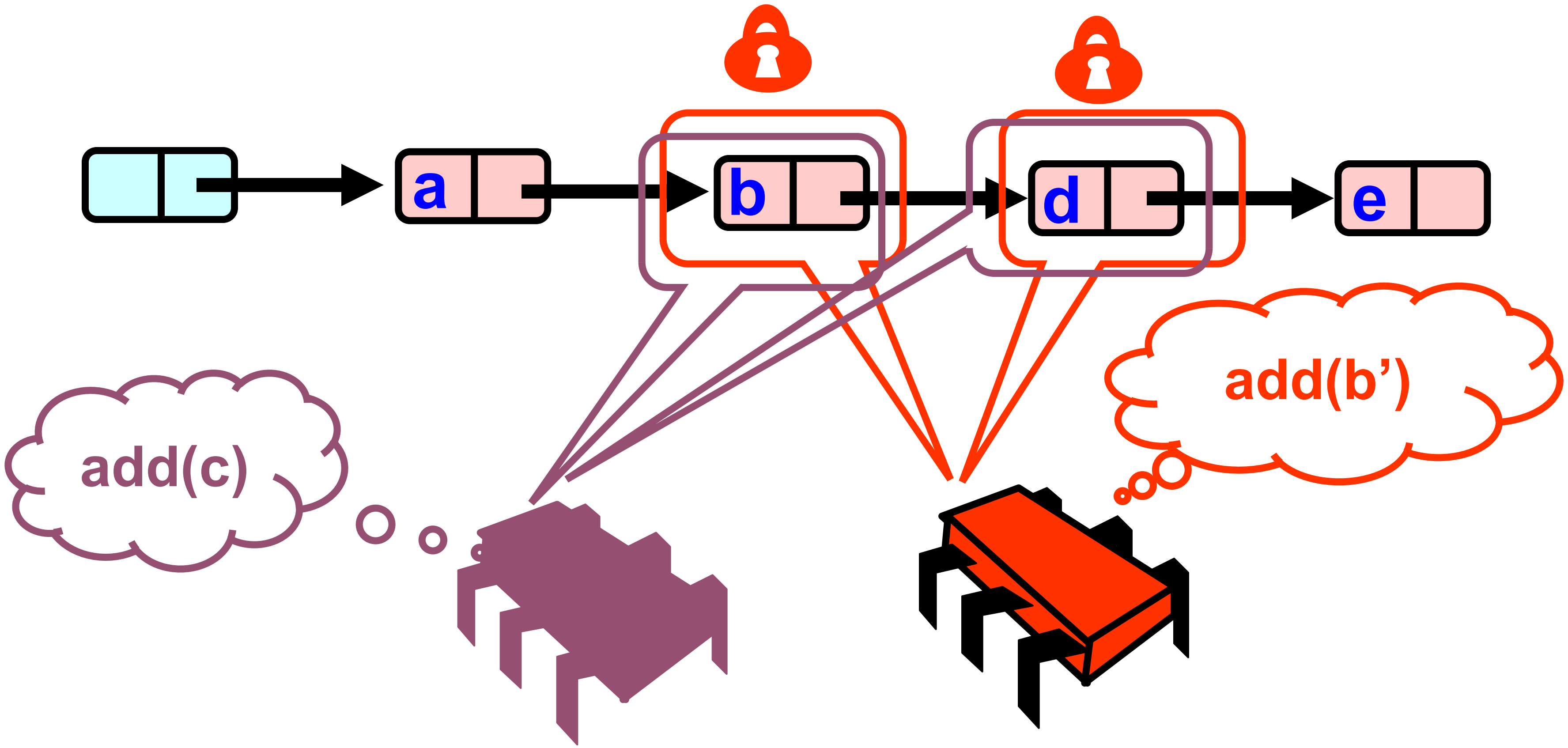
Back to the linked list

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

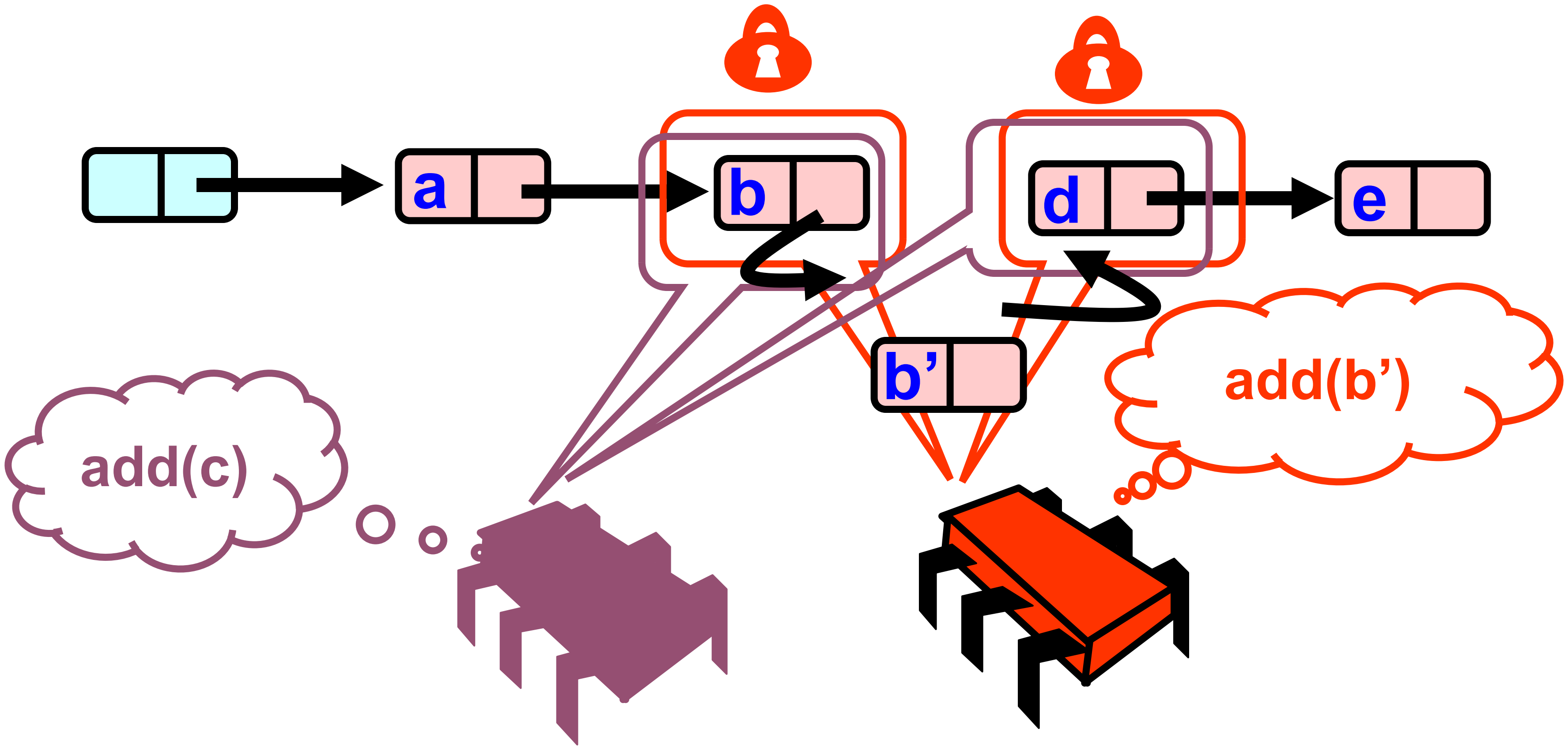
What Else Could Go Wrong?



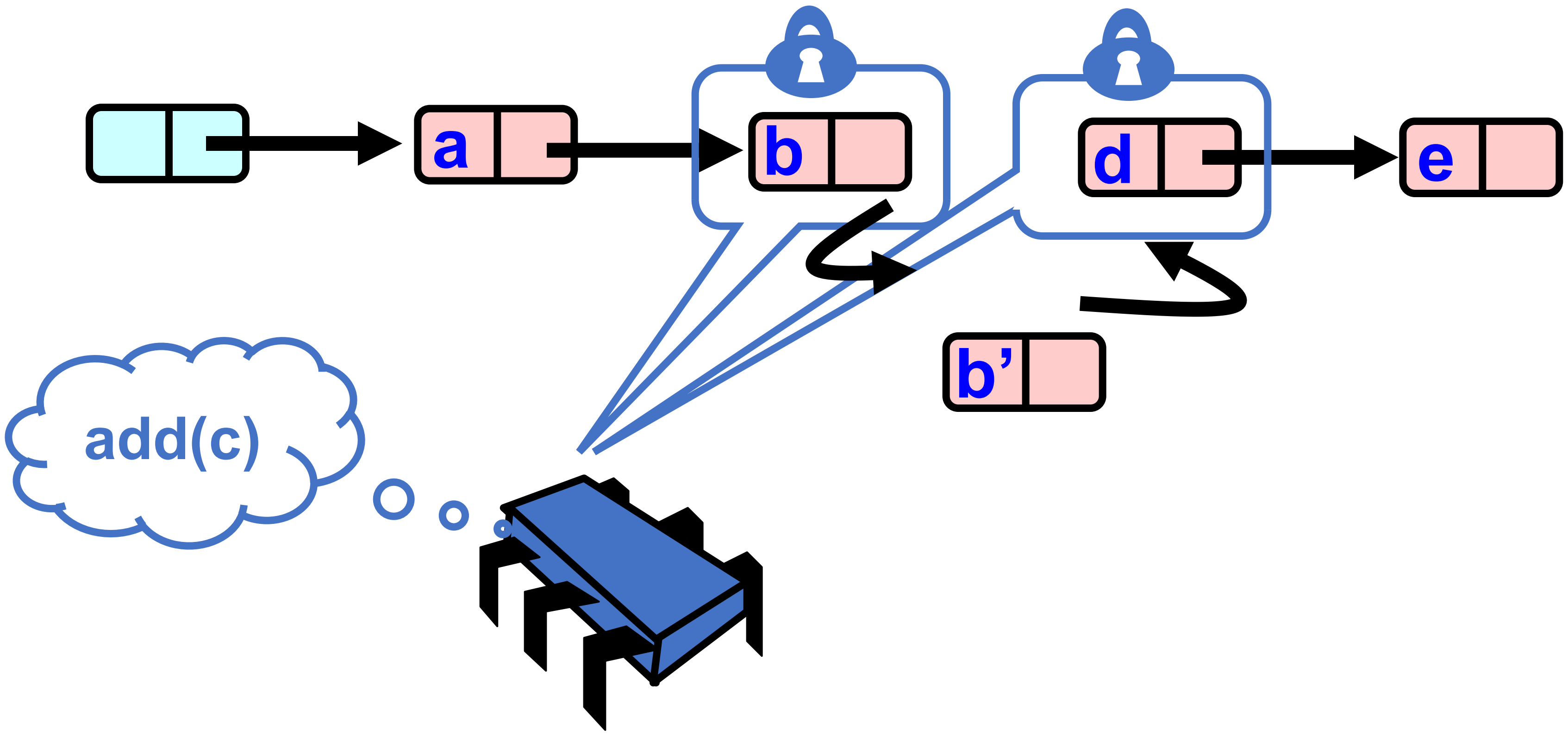
What Else Coould Go Wrong?



What Else Could Go Wrong?

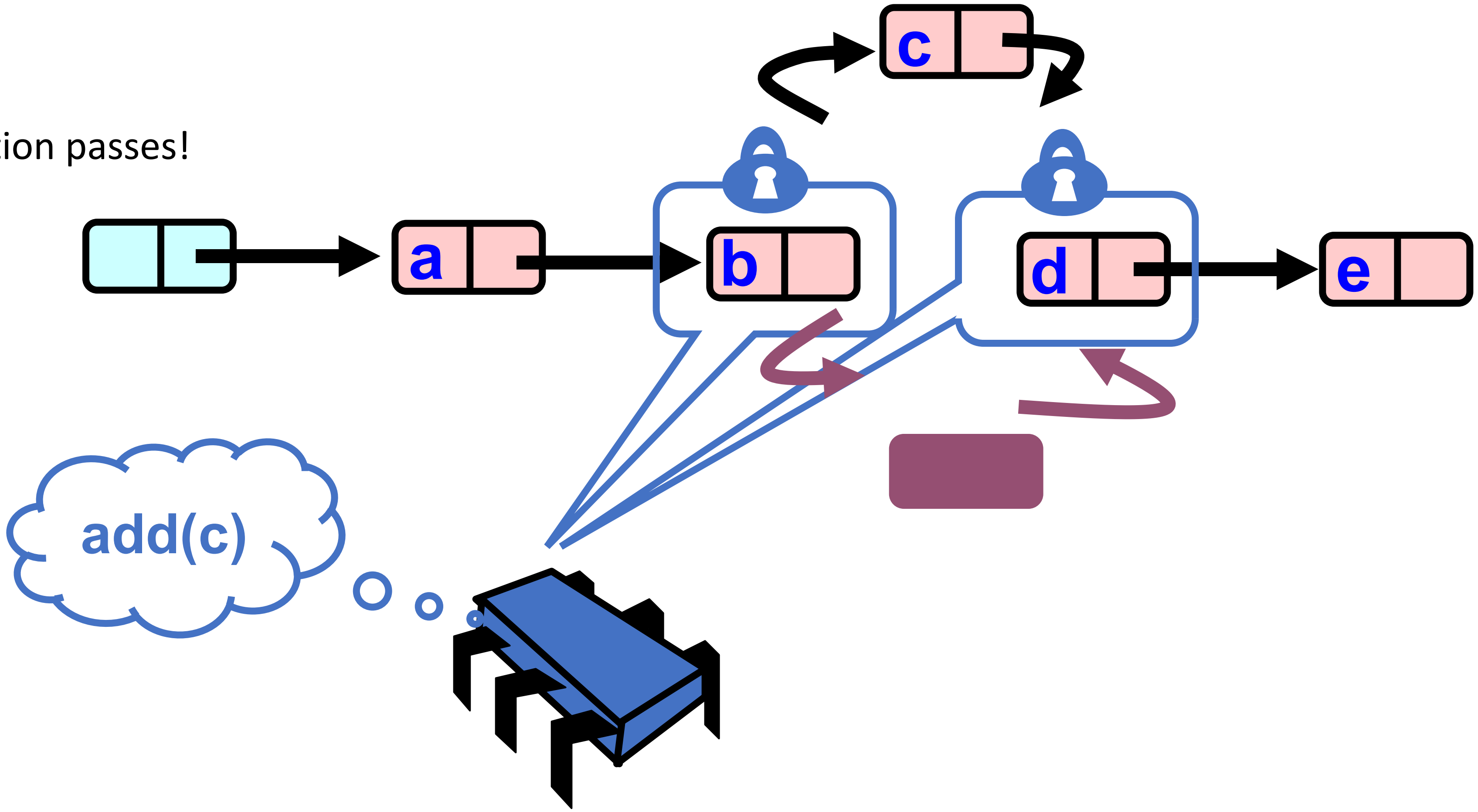


What Else Could Go Wrong?

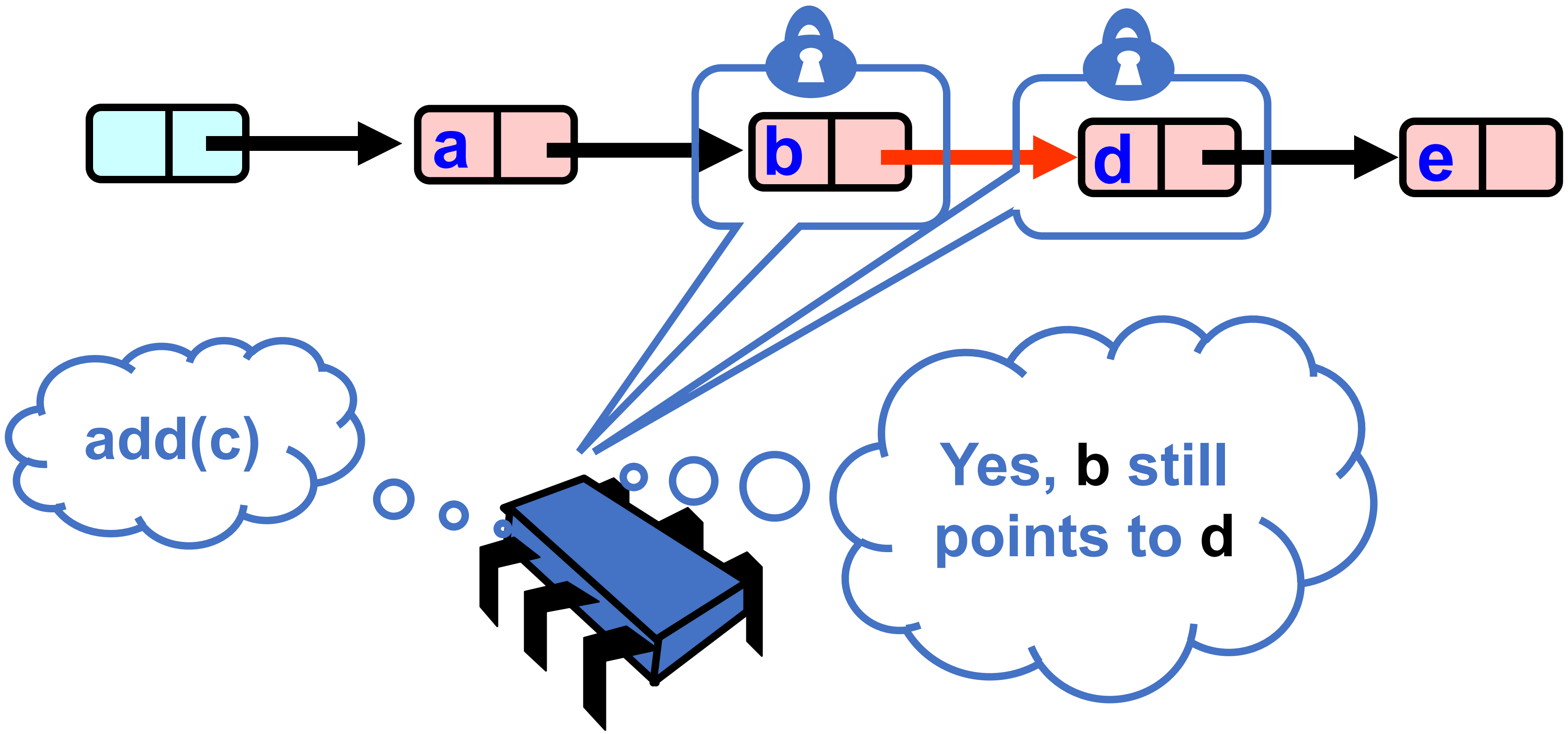


What Else Could Go Wrong?

Validation passes!



Validate Part 2
(while holding locks)



Summary

- We traverse without lock
  - Traversal may access nodes that are locked
  - Its okay because we have atomic pointers!
- We might traverse deleted nodes
  - Its okay because we validate after we obtain locks
  - Two validations:
    - our node is still reachable (it was not deleted)
    - Our insertion point is still valid (no thread has inserted in the meantime)
- We don’t actually free node memory, but we put them in a list to be freed later

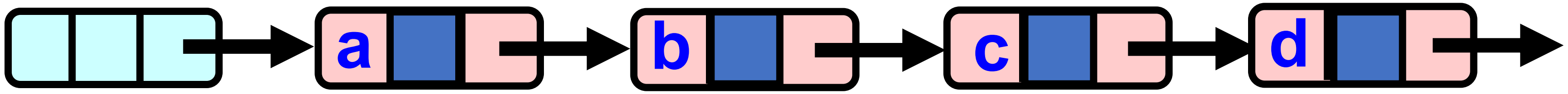
Can we optimize more?

- Scan the list once?

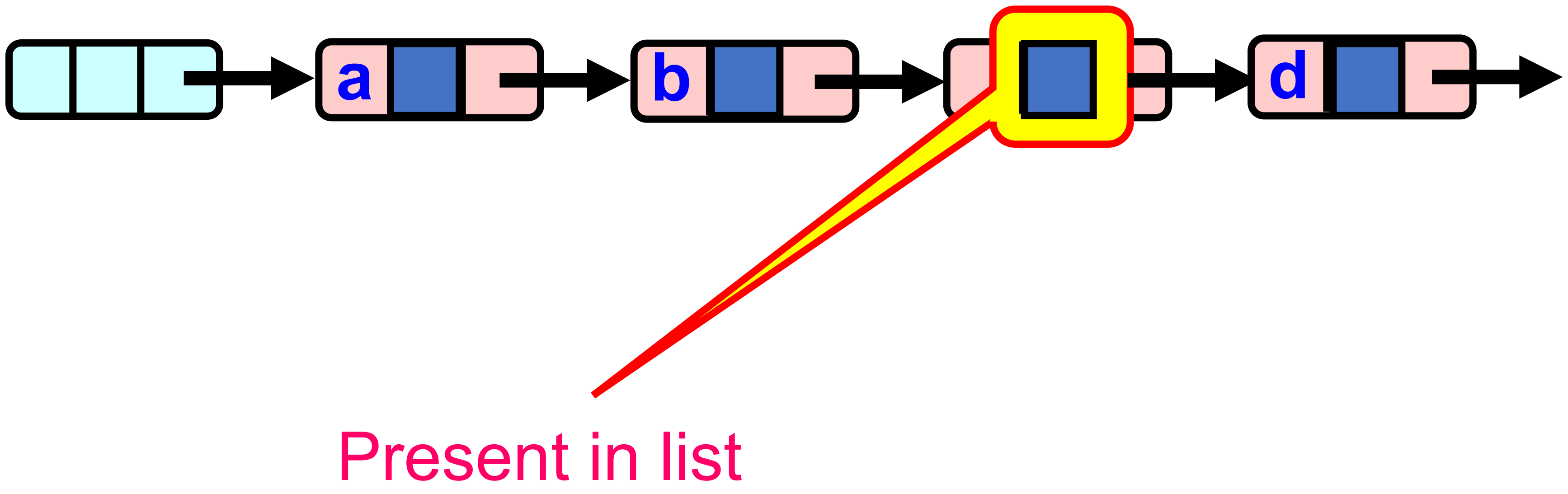
Two step removal List

- **`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)

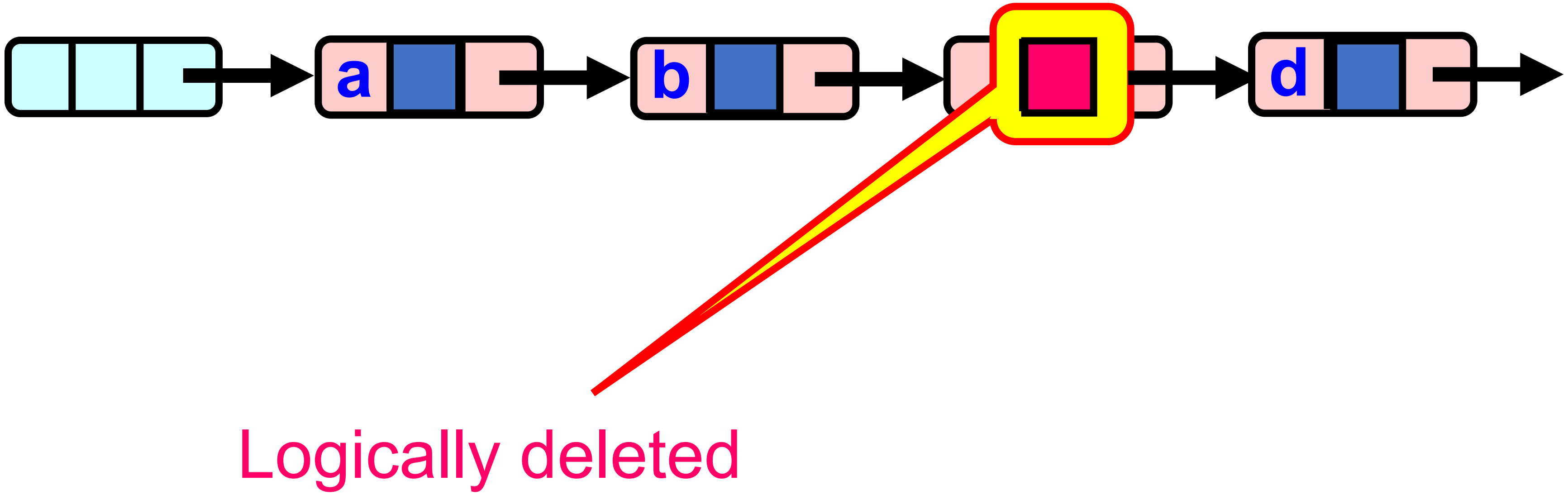
Two step removal Removal



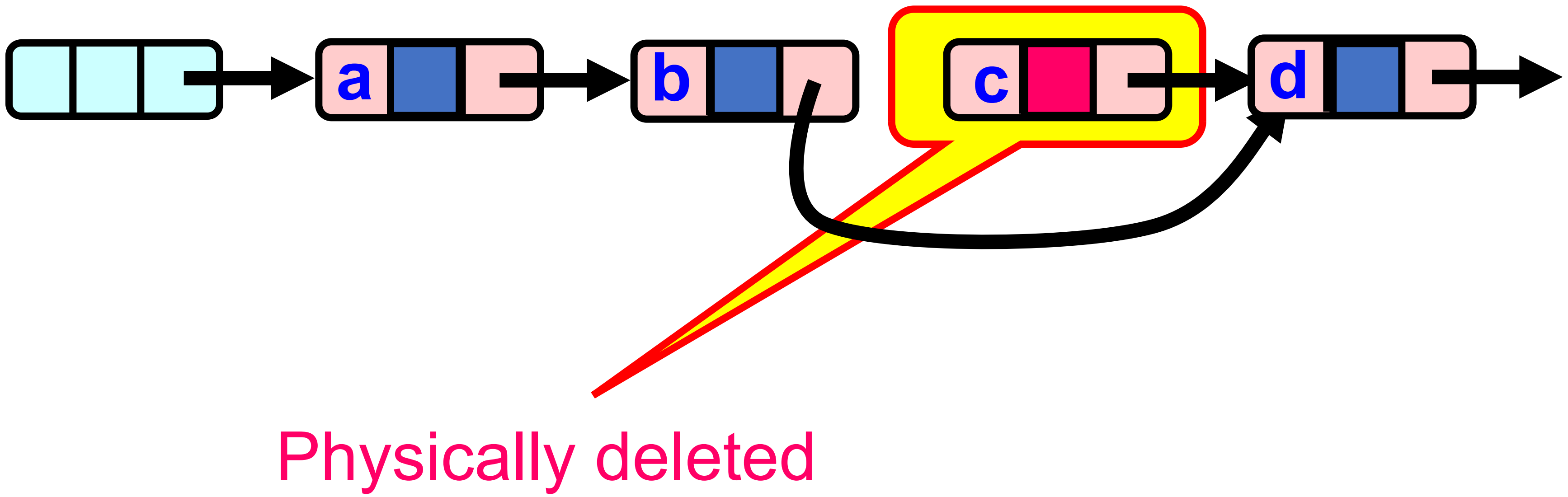
Two step removal Removal



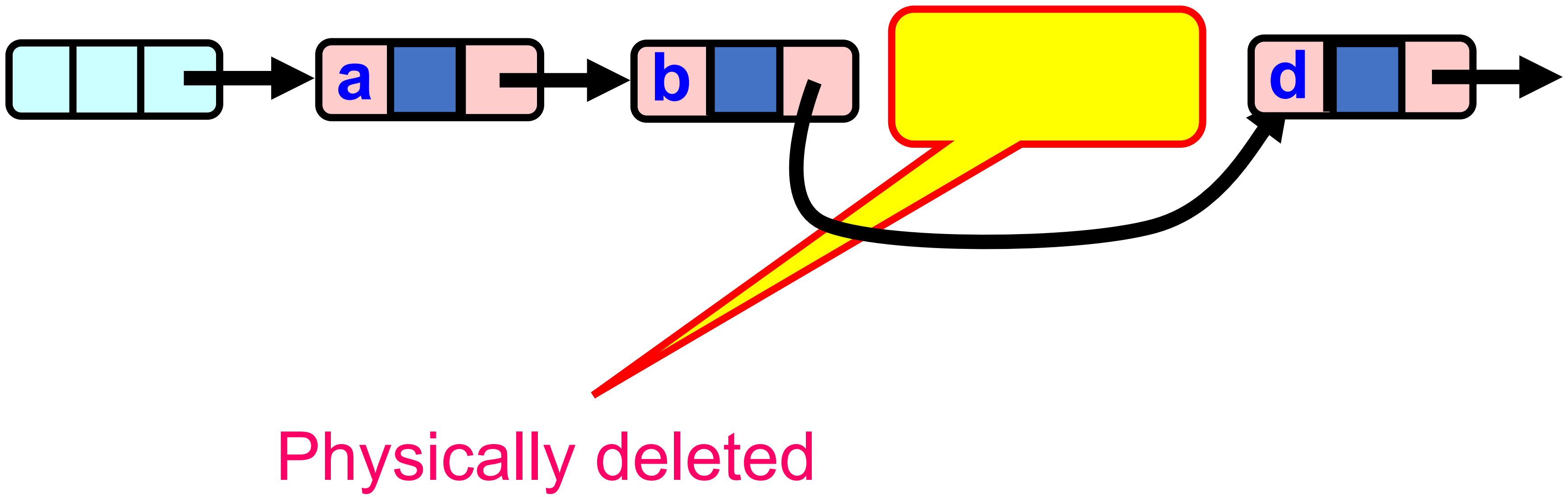
Two step removal Removal



Two step removal Removal



Two step removal Removal



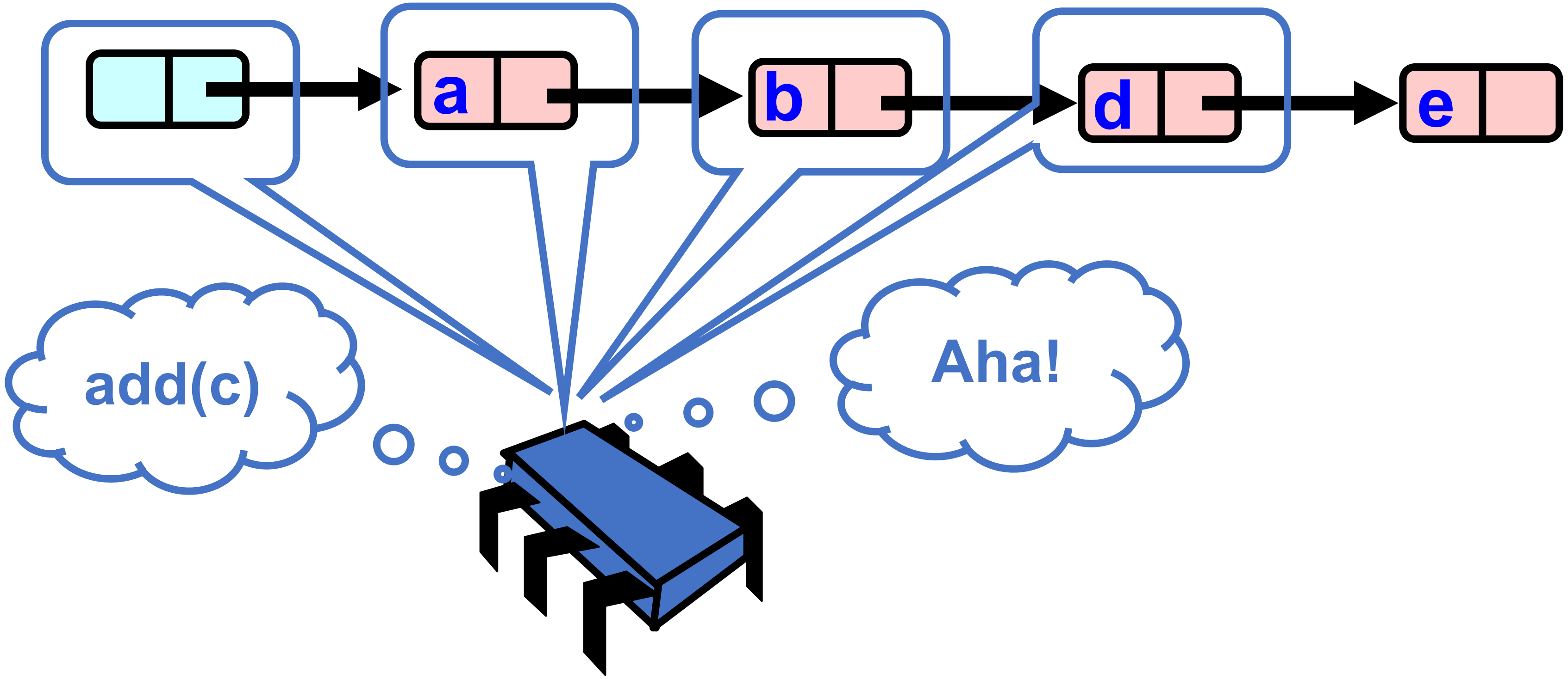
Two step remove list

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

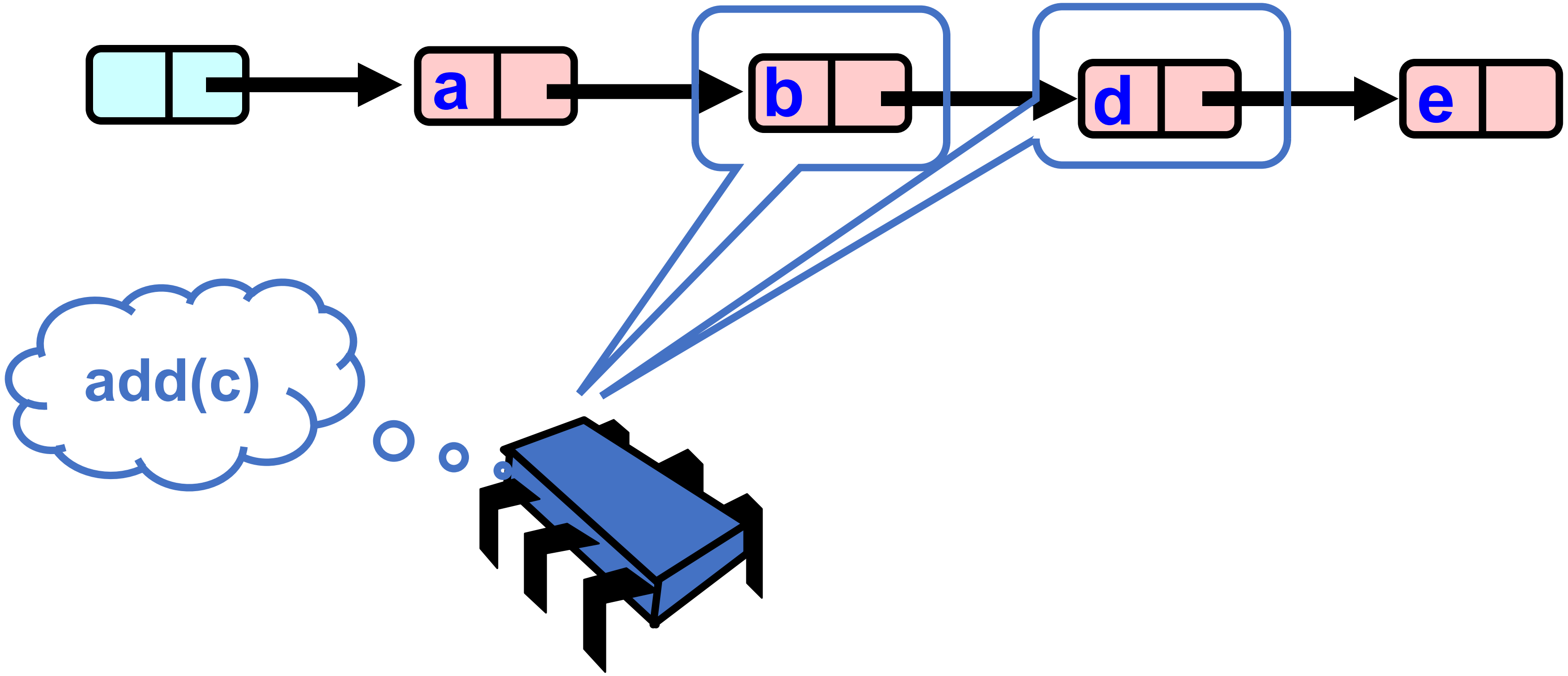
Validation

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

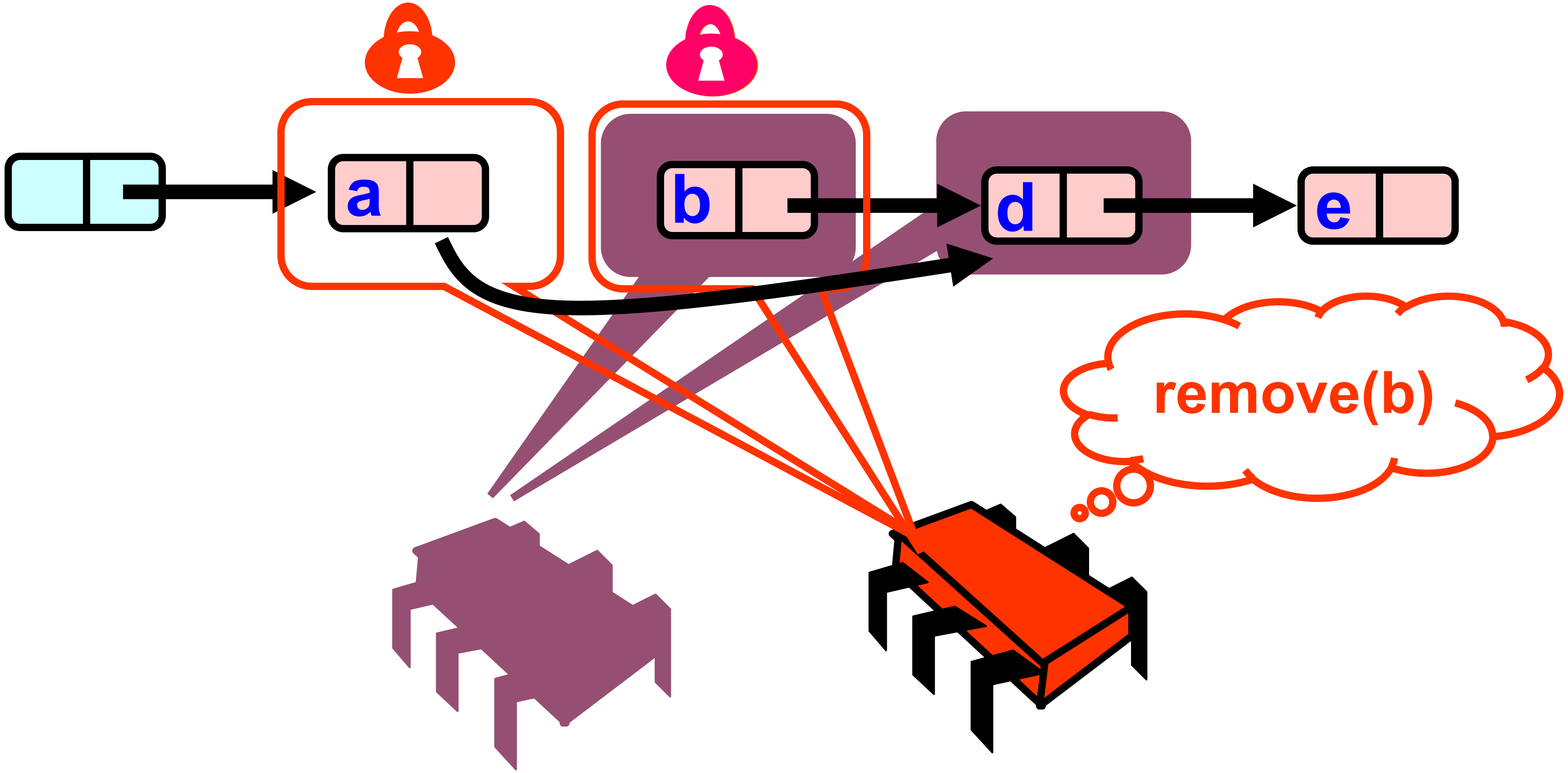
What could go wrong?



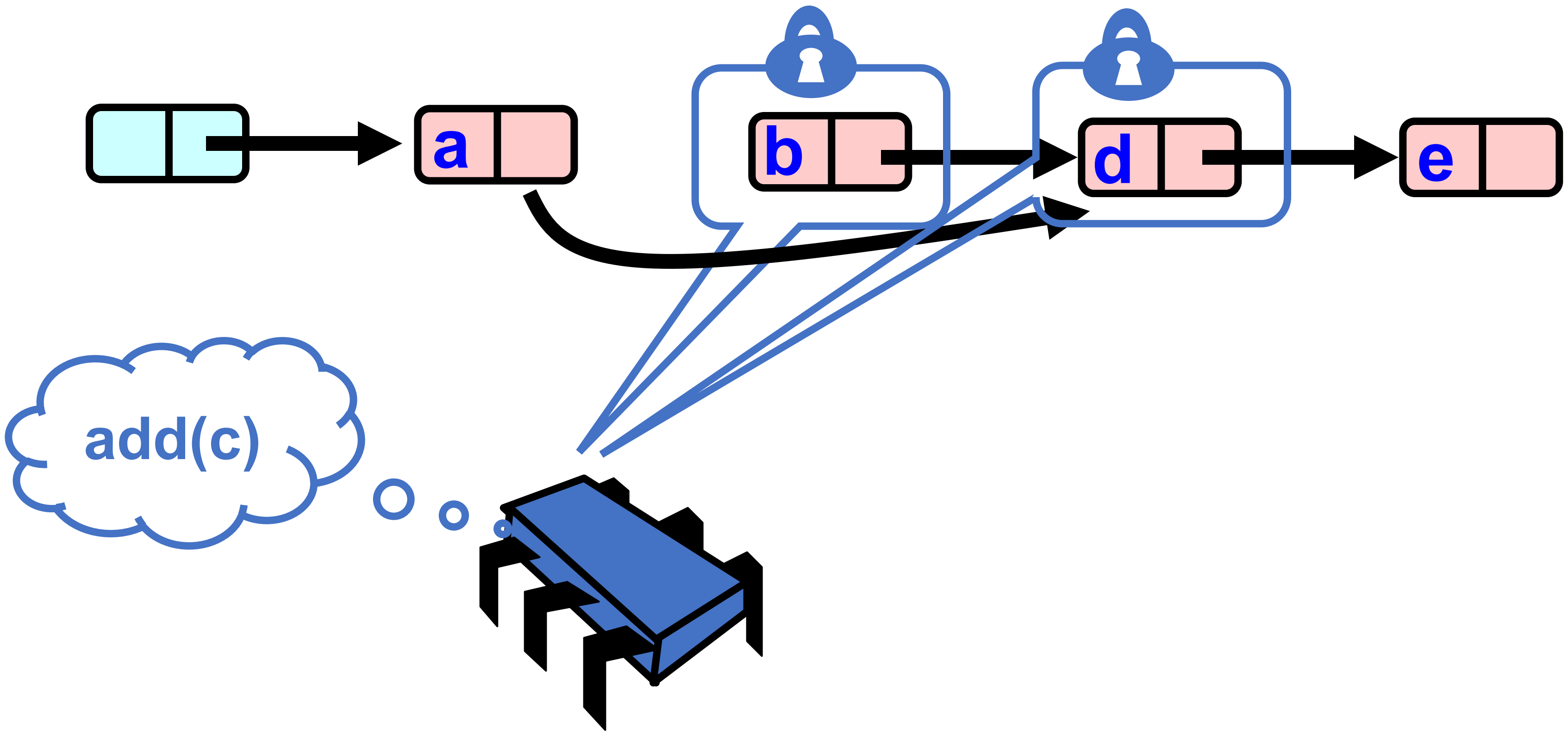
What could go wrong?



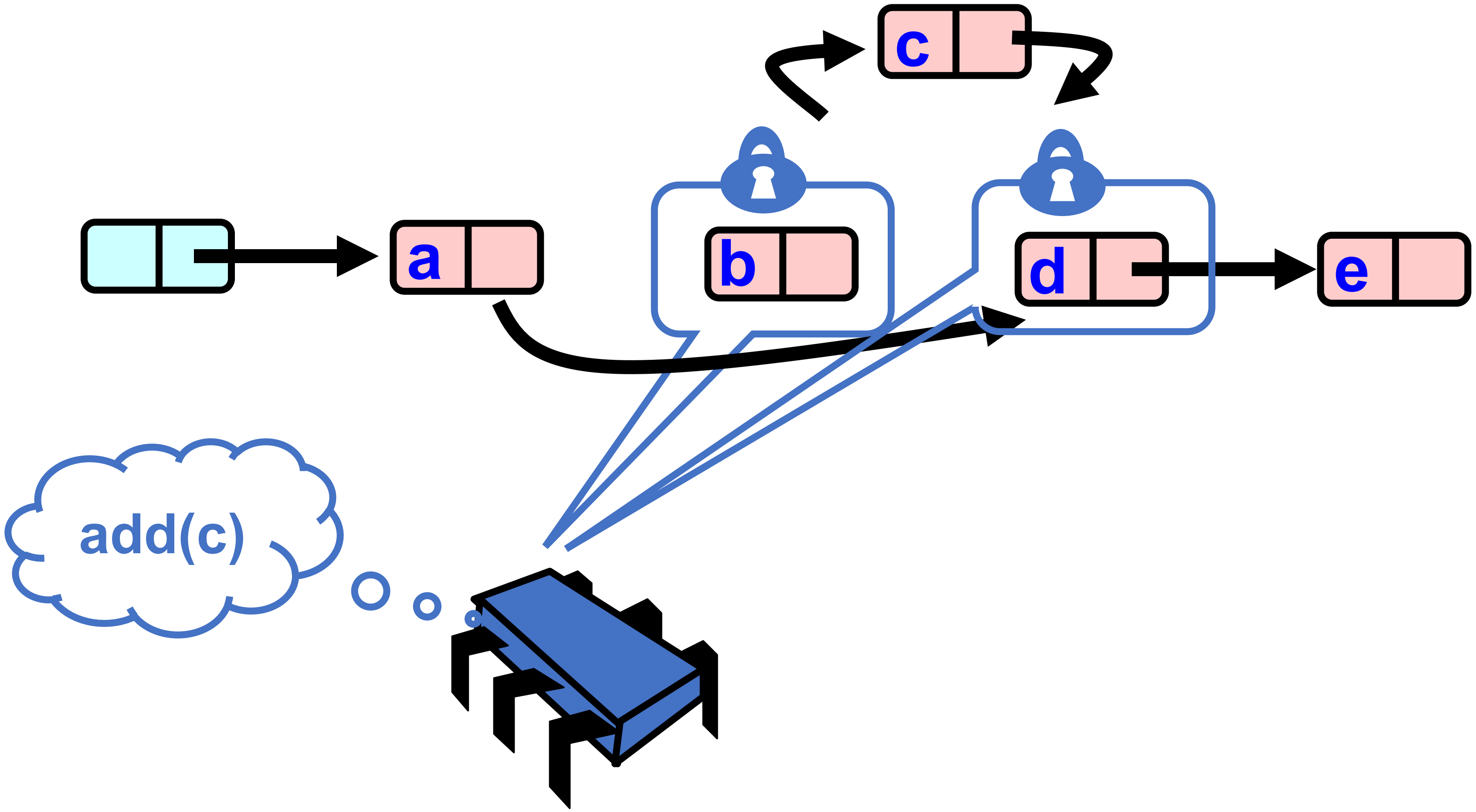
What could go wrong?



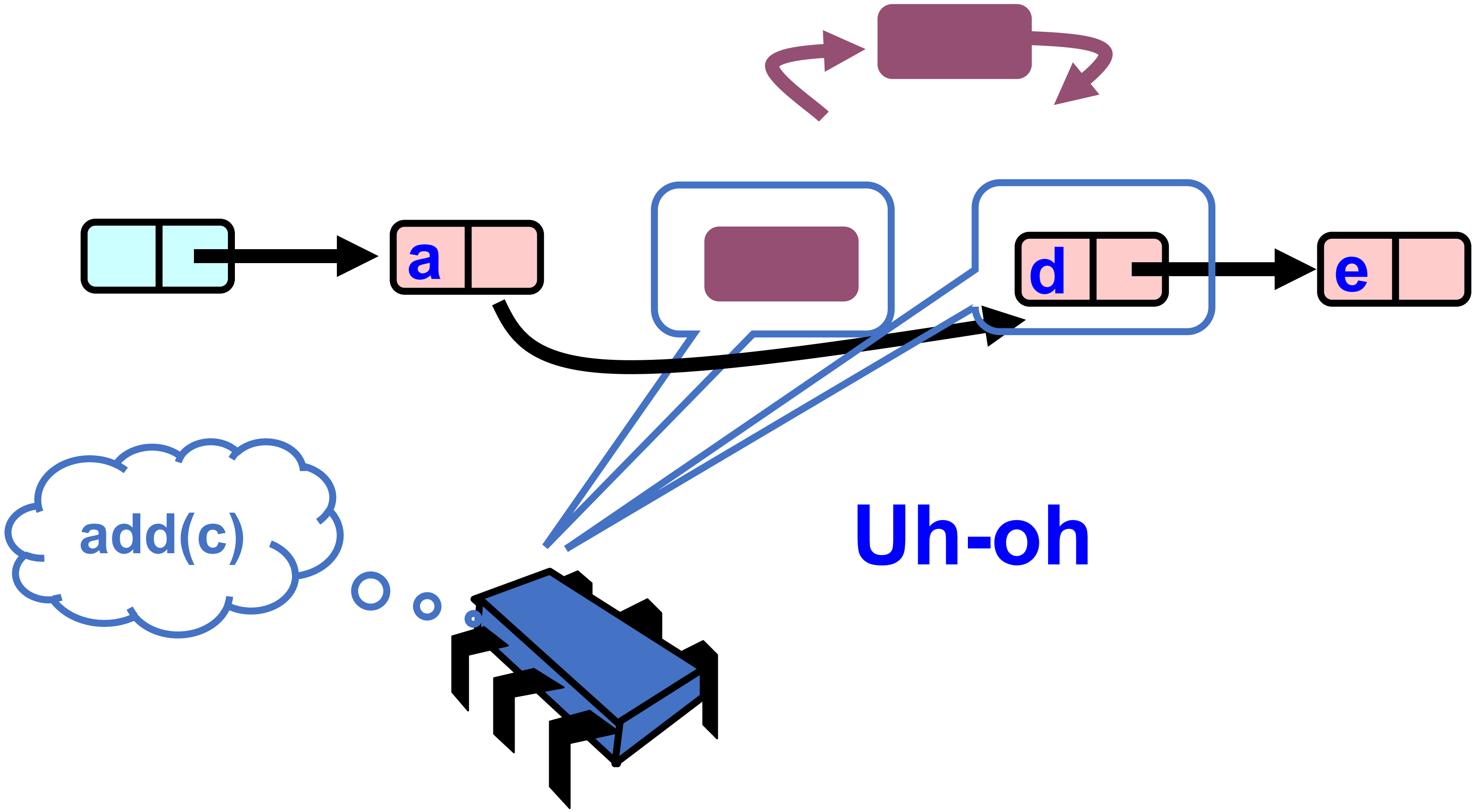
What could go wrong?



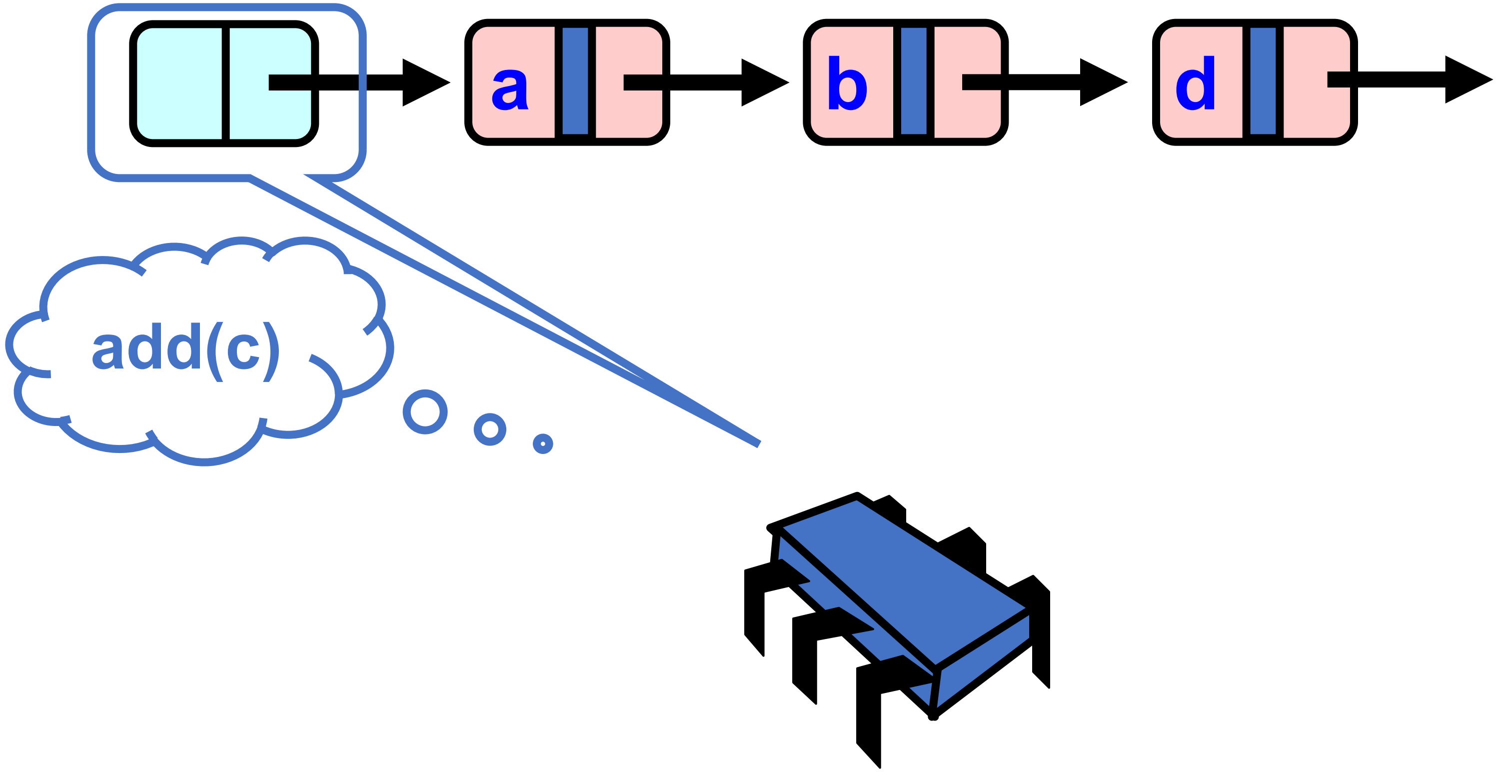
What could go wrong?



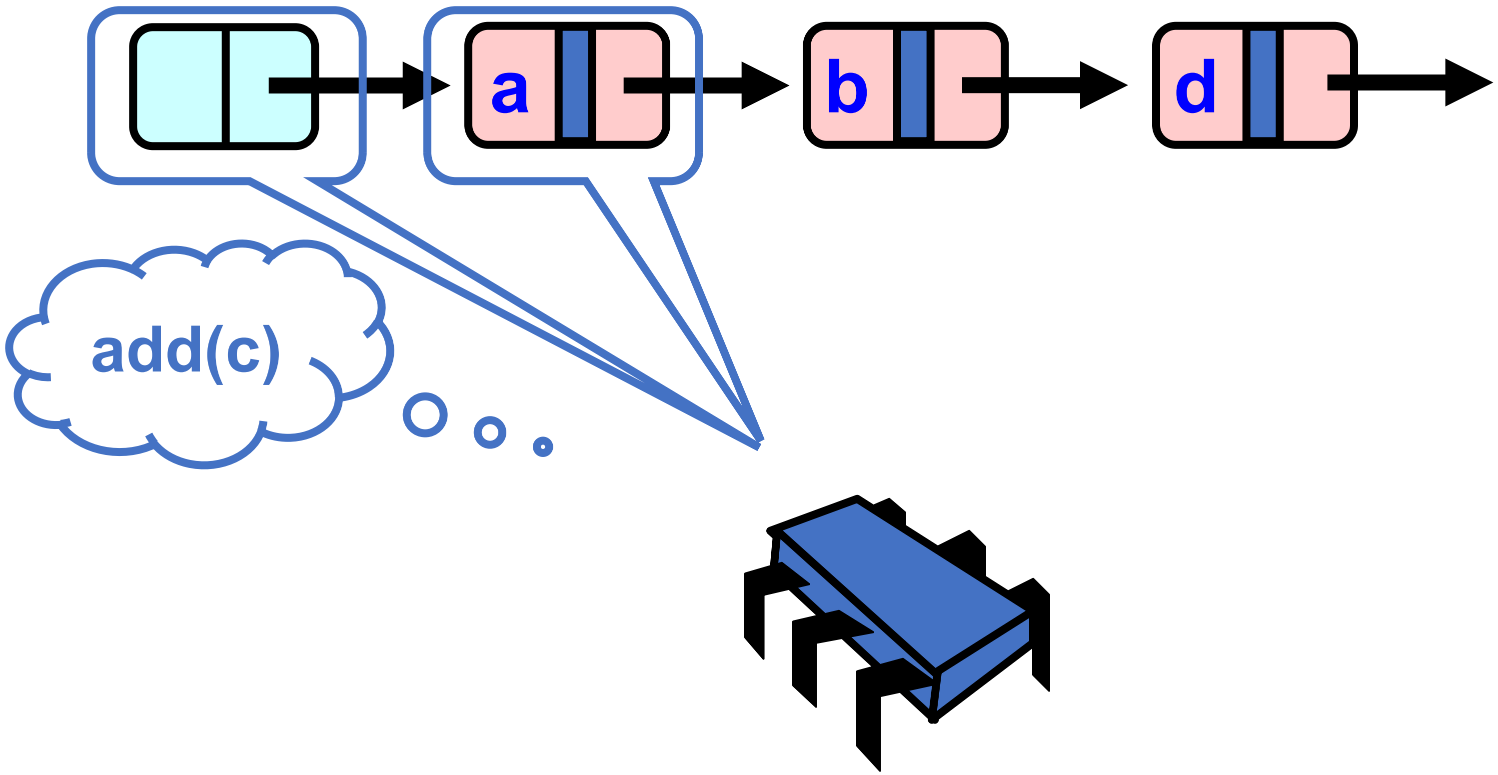
What could go wrong?



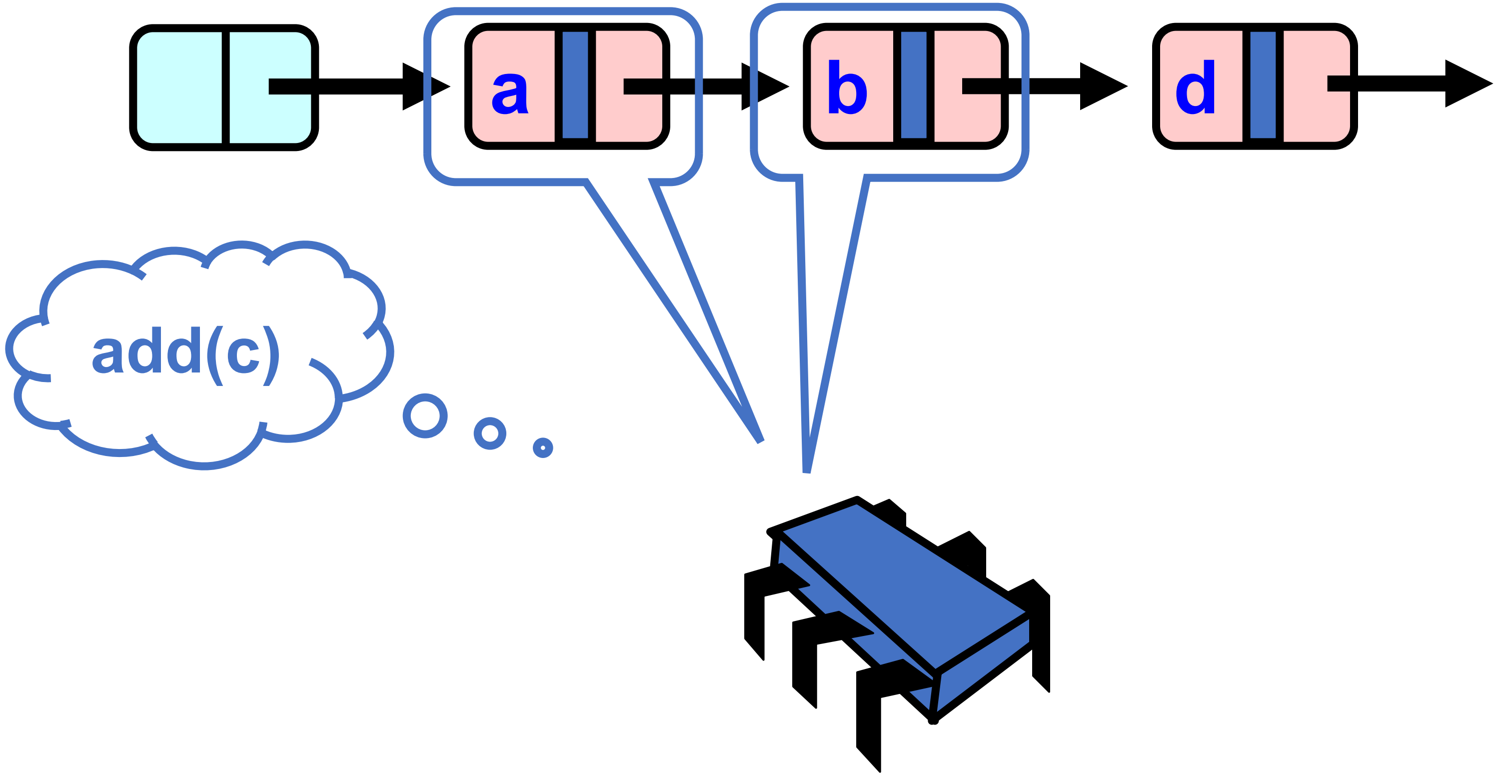
Fixed with logical flag



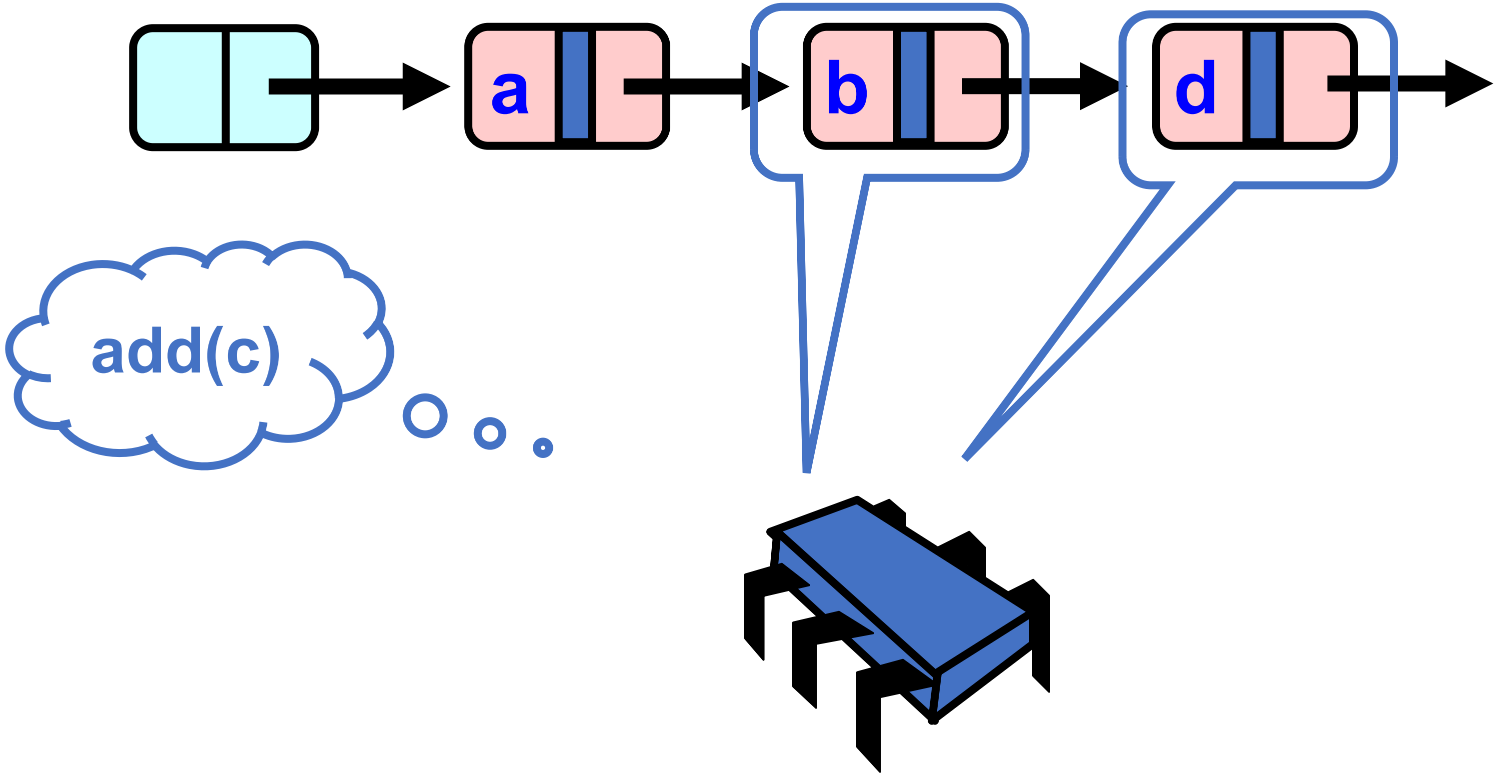
Fixed with logical flag



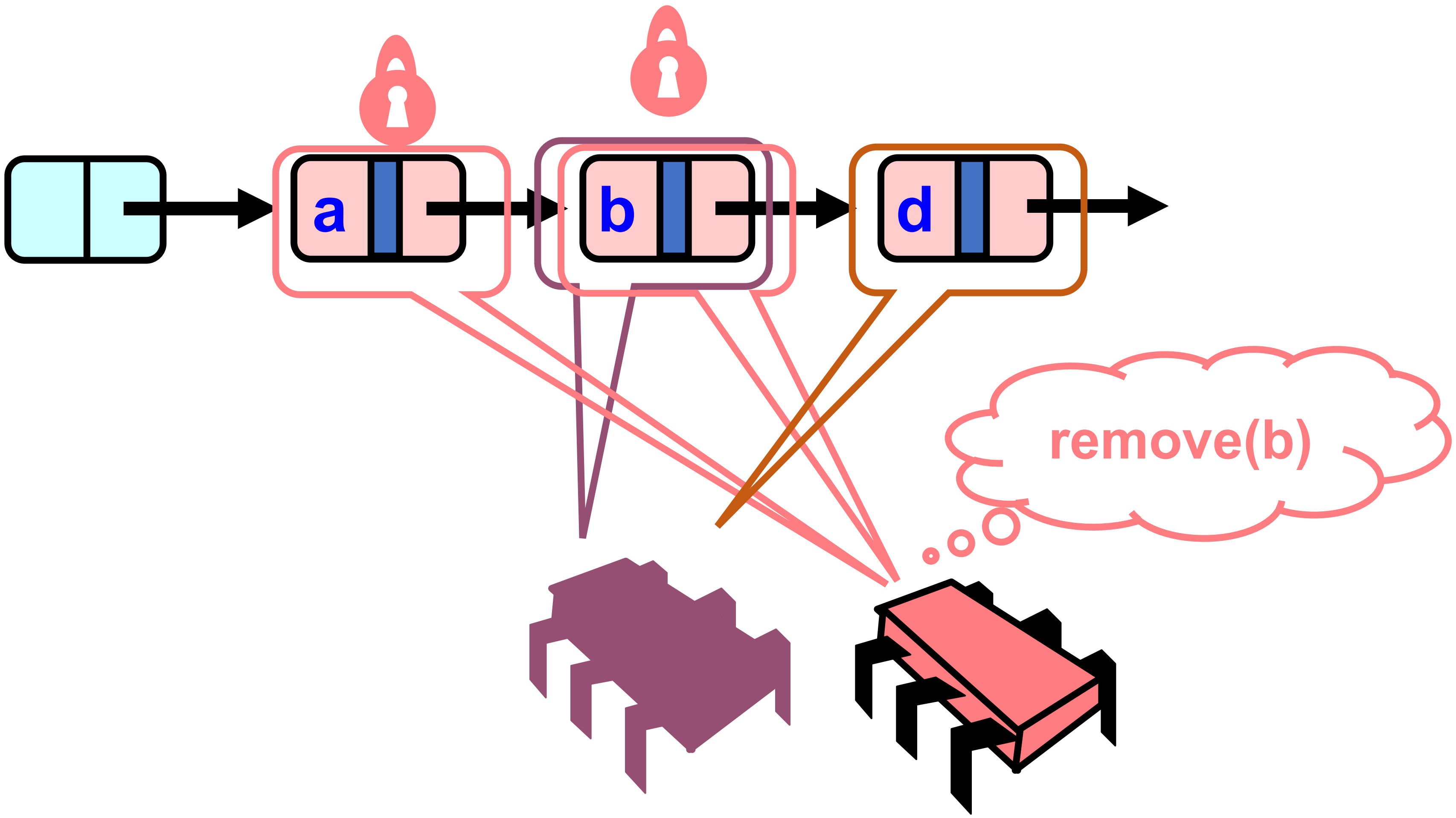
Fixed with logical flag



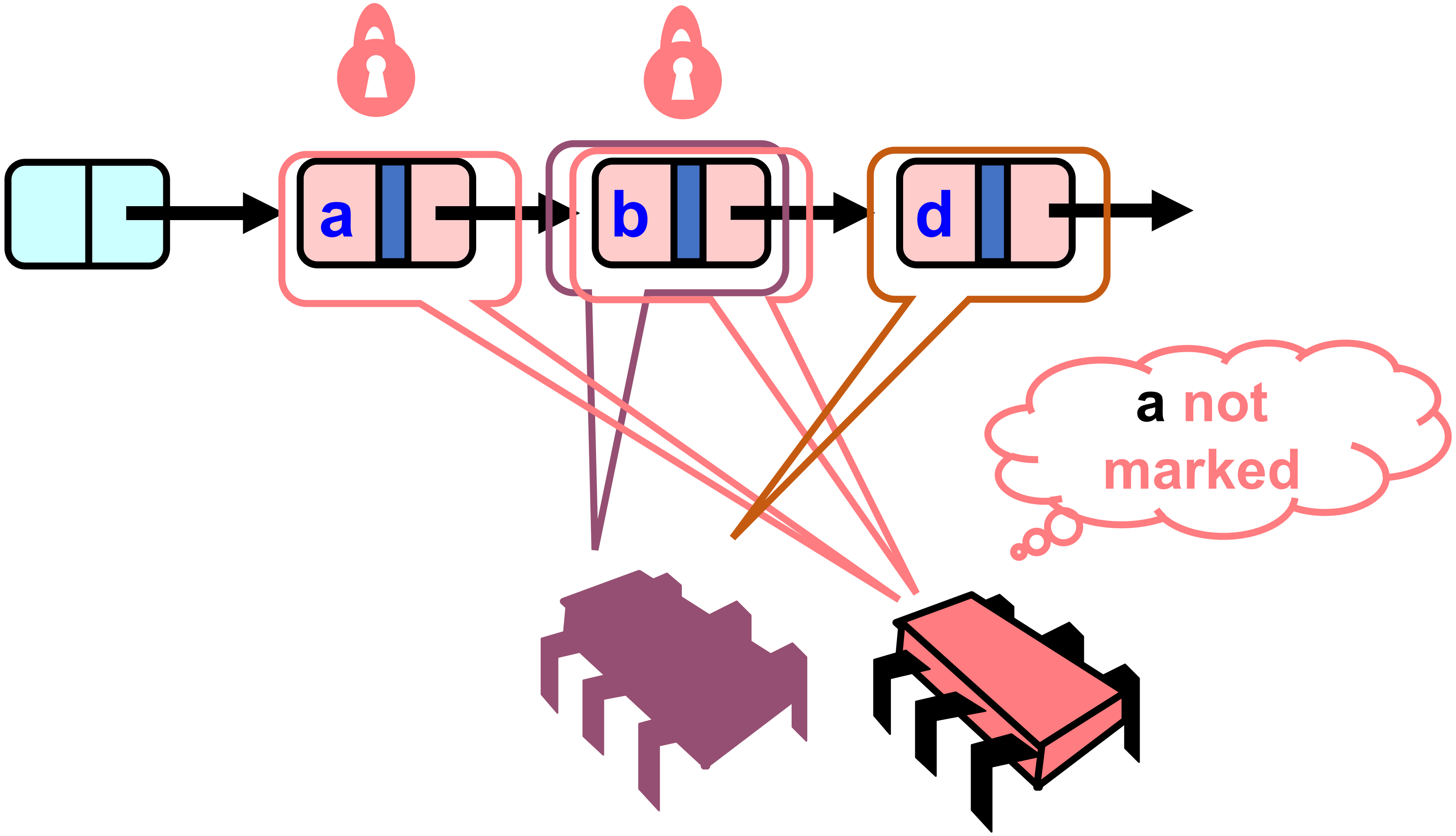
Fixed with logical flag



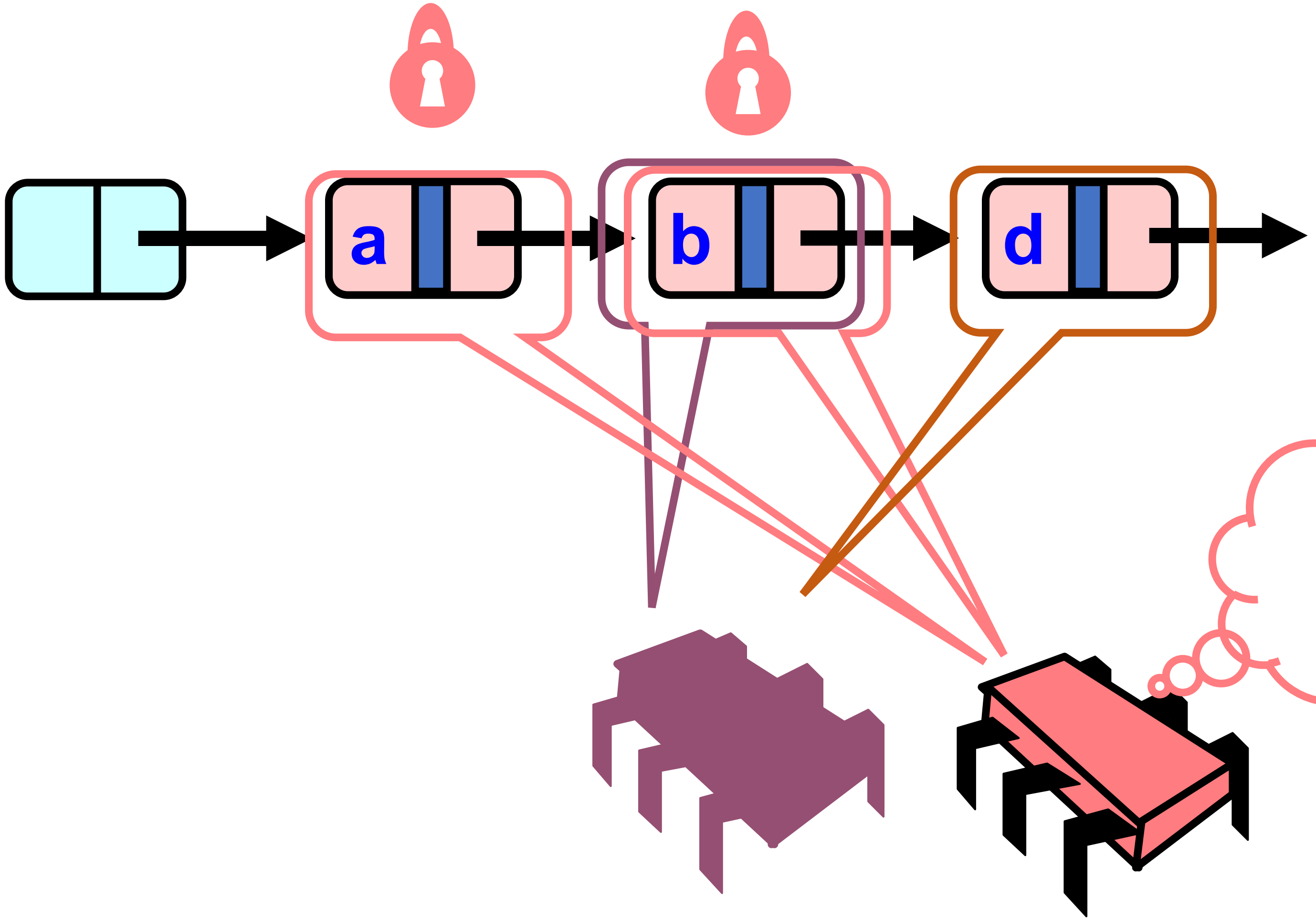
Fixed with logical flag



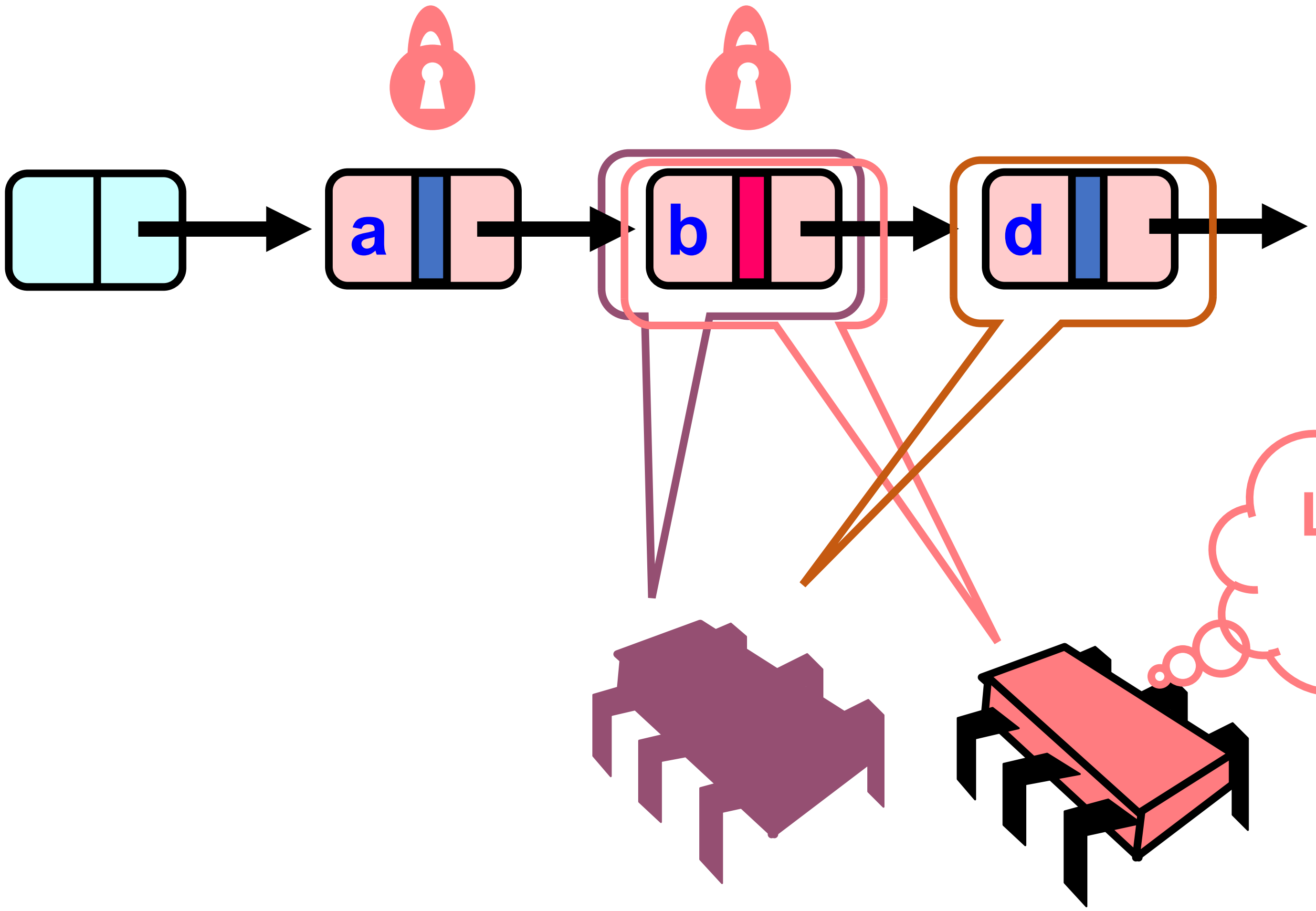
Fixed with logical flag



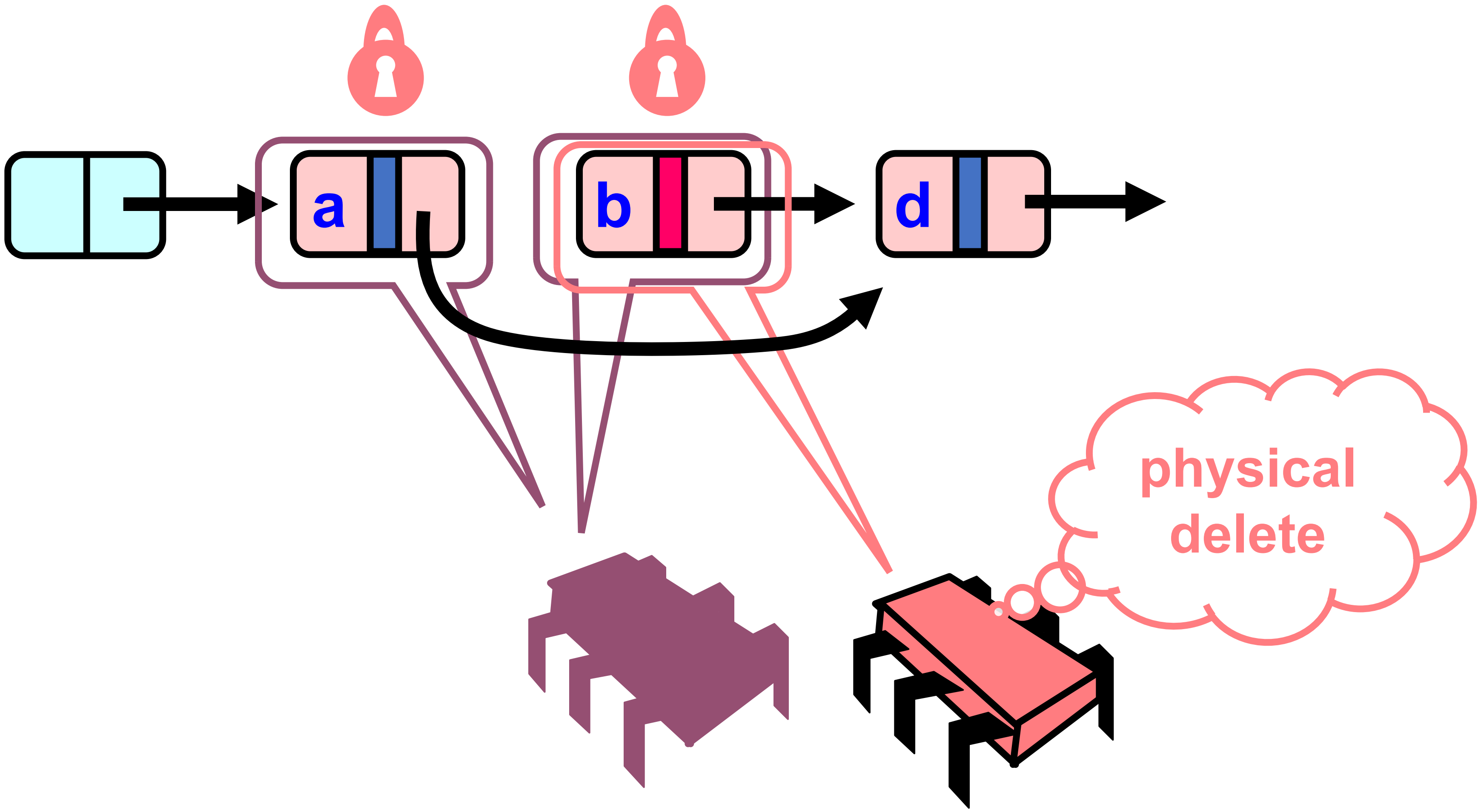
Fixed with logical flag



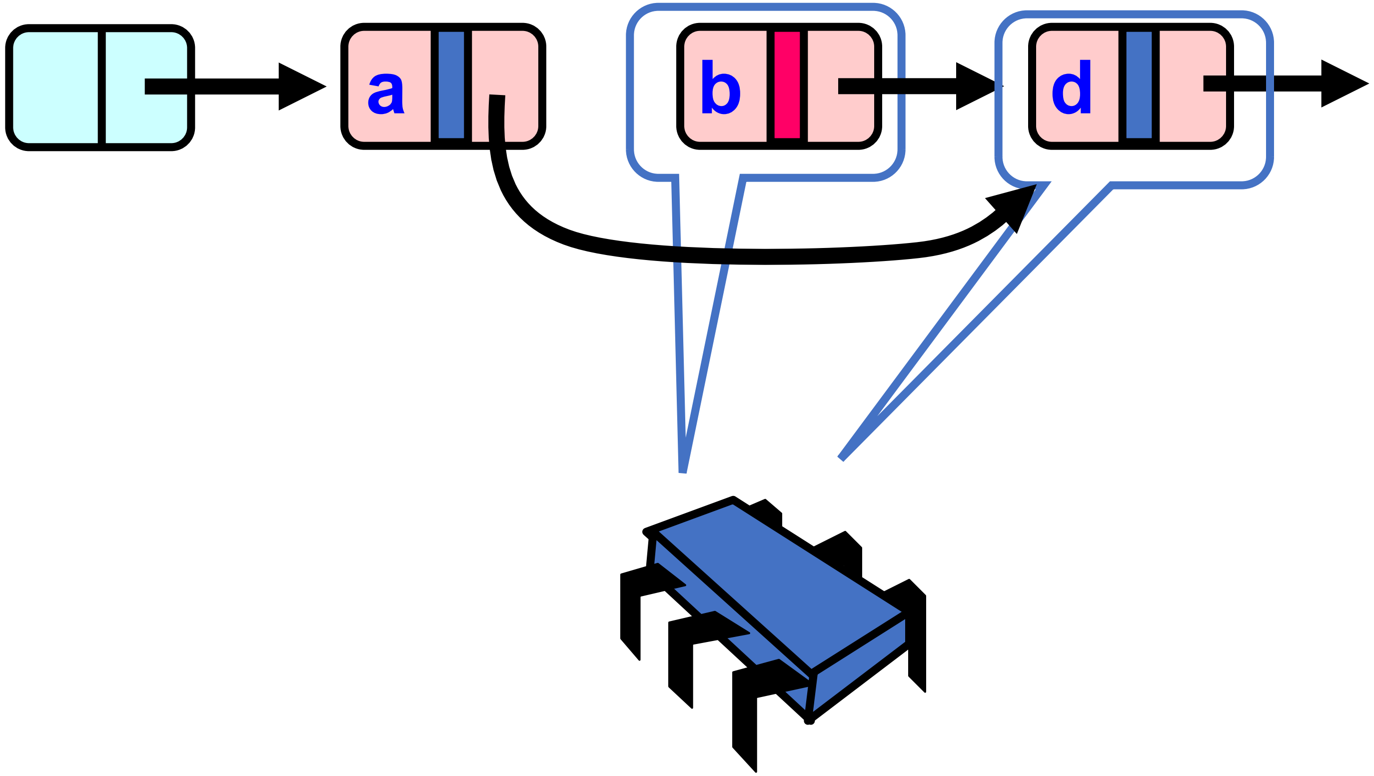
Fixed with logical flag



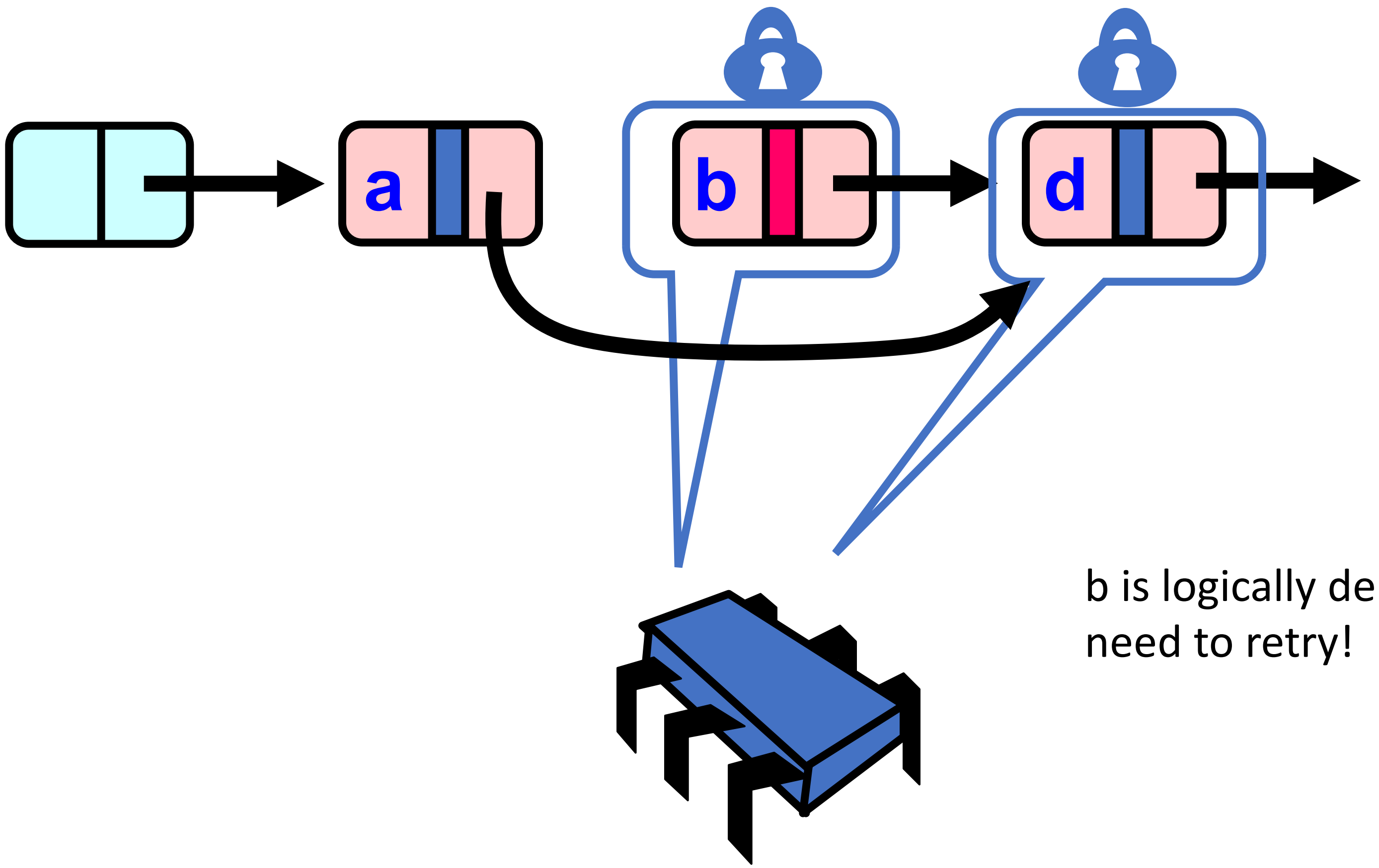
Fixed with logical flag



Fixed with logical flag



Fixed with logical flag



b is logically deleted so we need to retry!

To complete the picture

- Need to do similar reasoning with all combination of object methods.

- More information in the book!

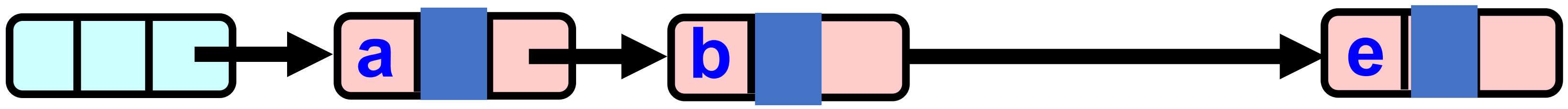
Evaluation

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

Lock-free Lists

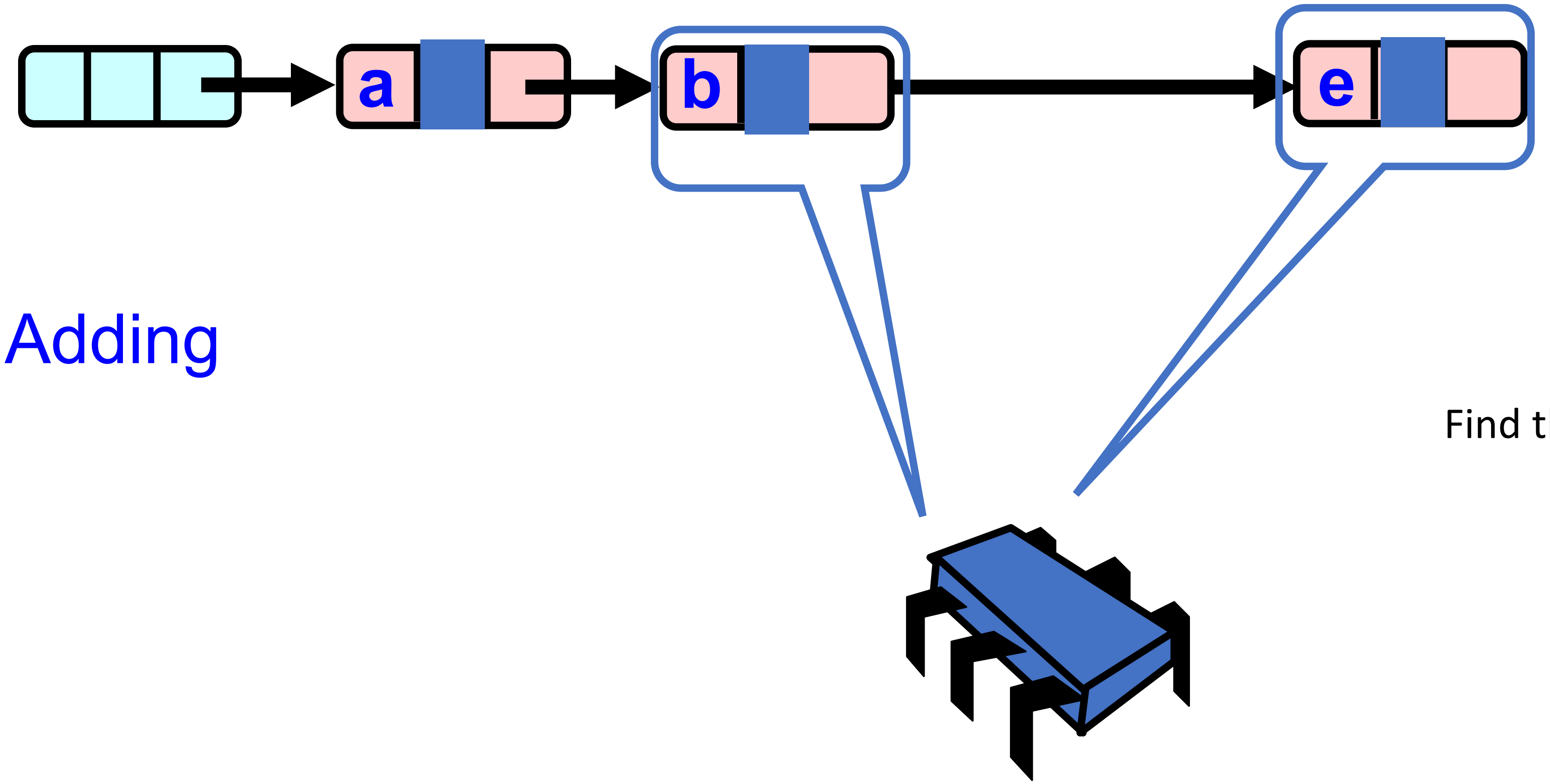
- Next logical step
  - lock-free add() and remove()
- What sort of atomics do we need?
  - Loads/stores?
  - RMWs?

Lock-free Lists



Adding

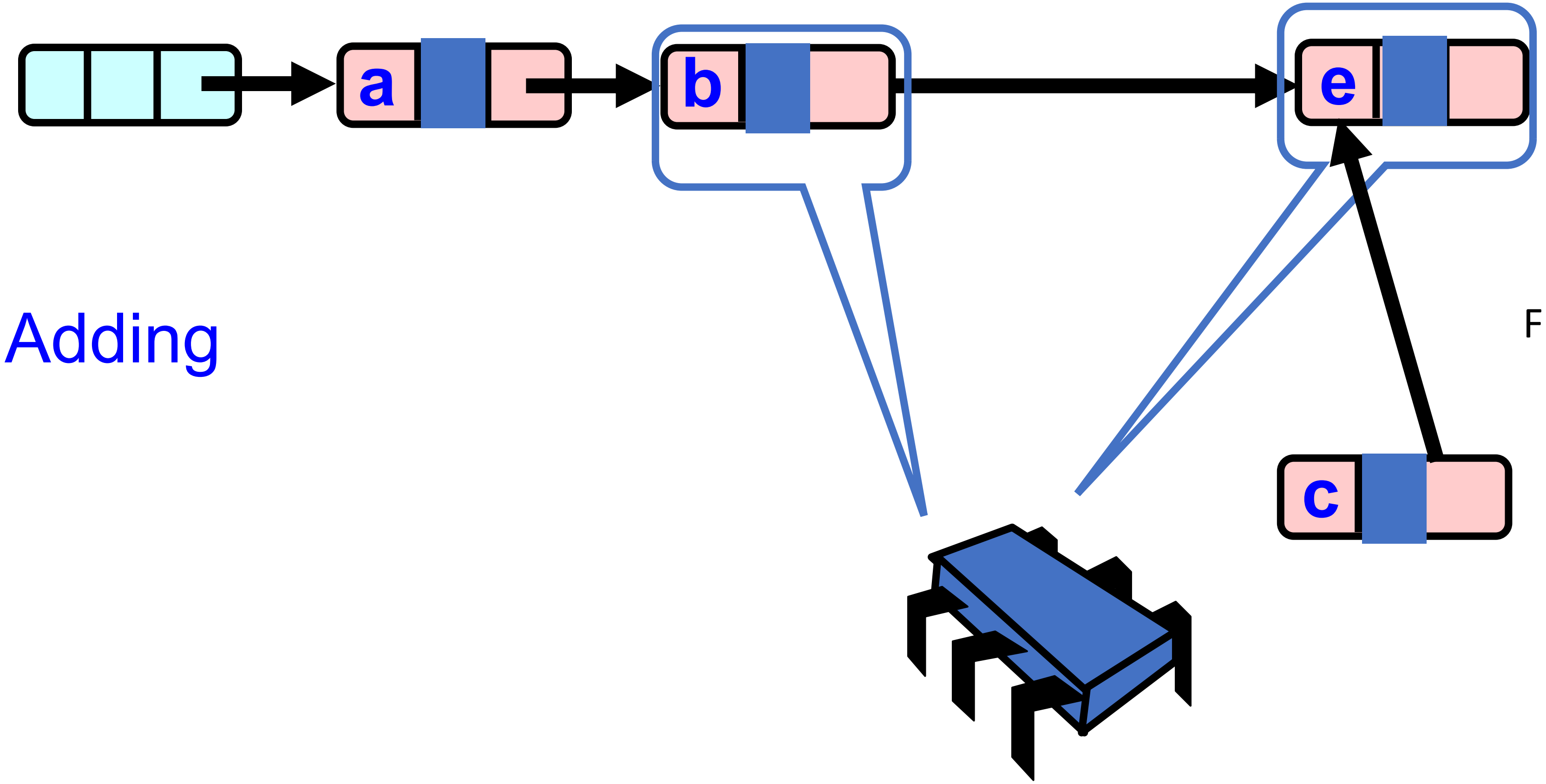
Lock-free Lists



Adding

Find the location

Lock-free Lists

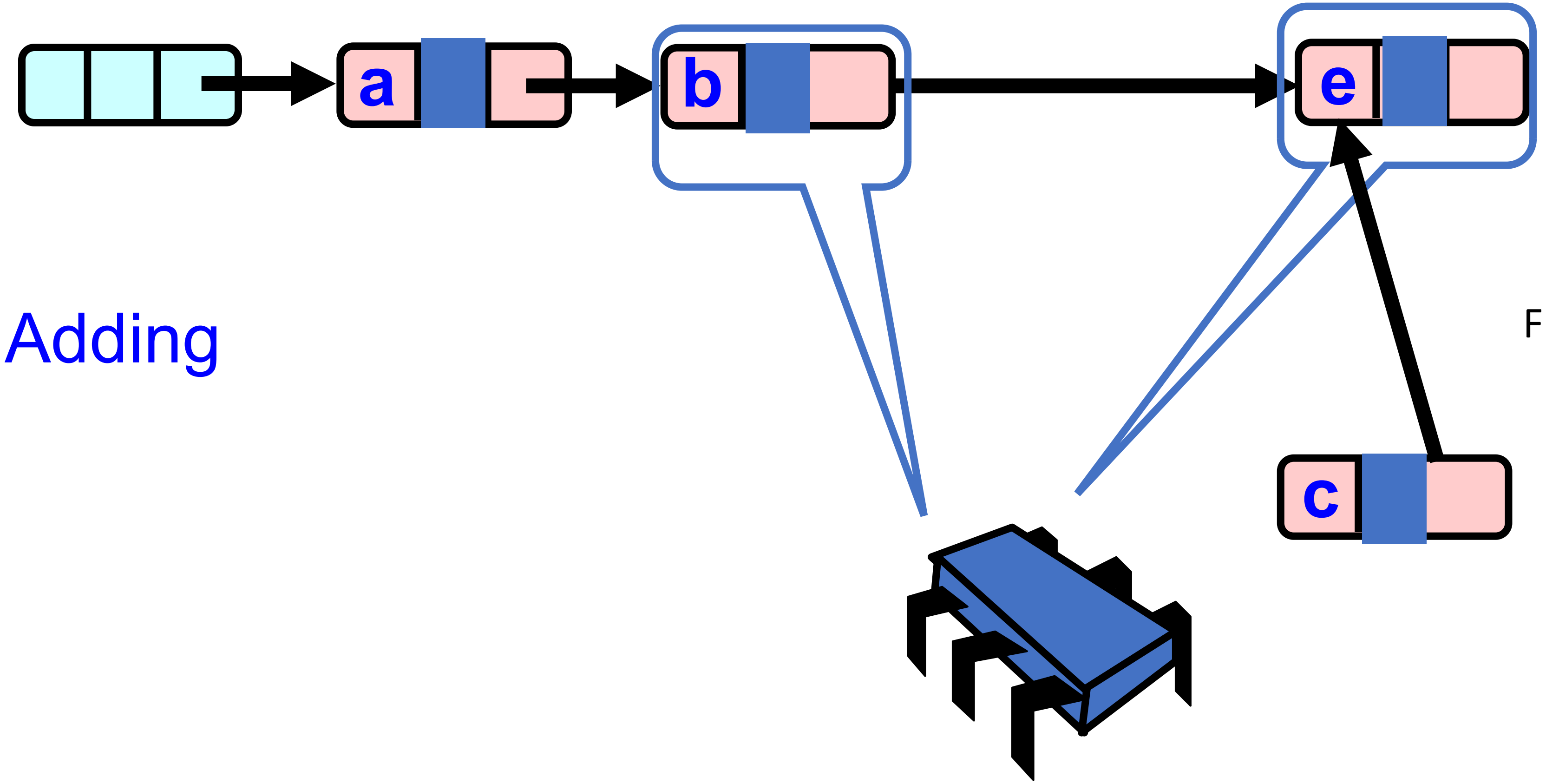


Adding

Find the location

create “c”

Lock-free Lists



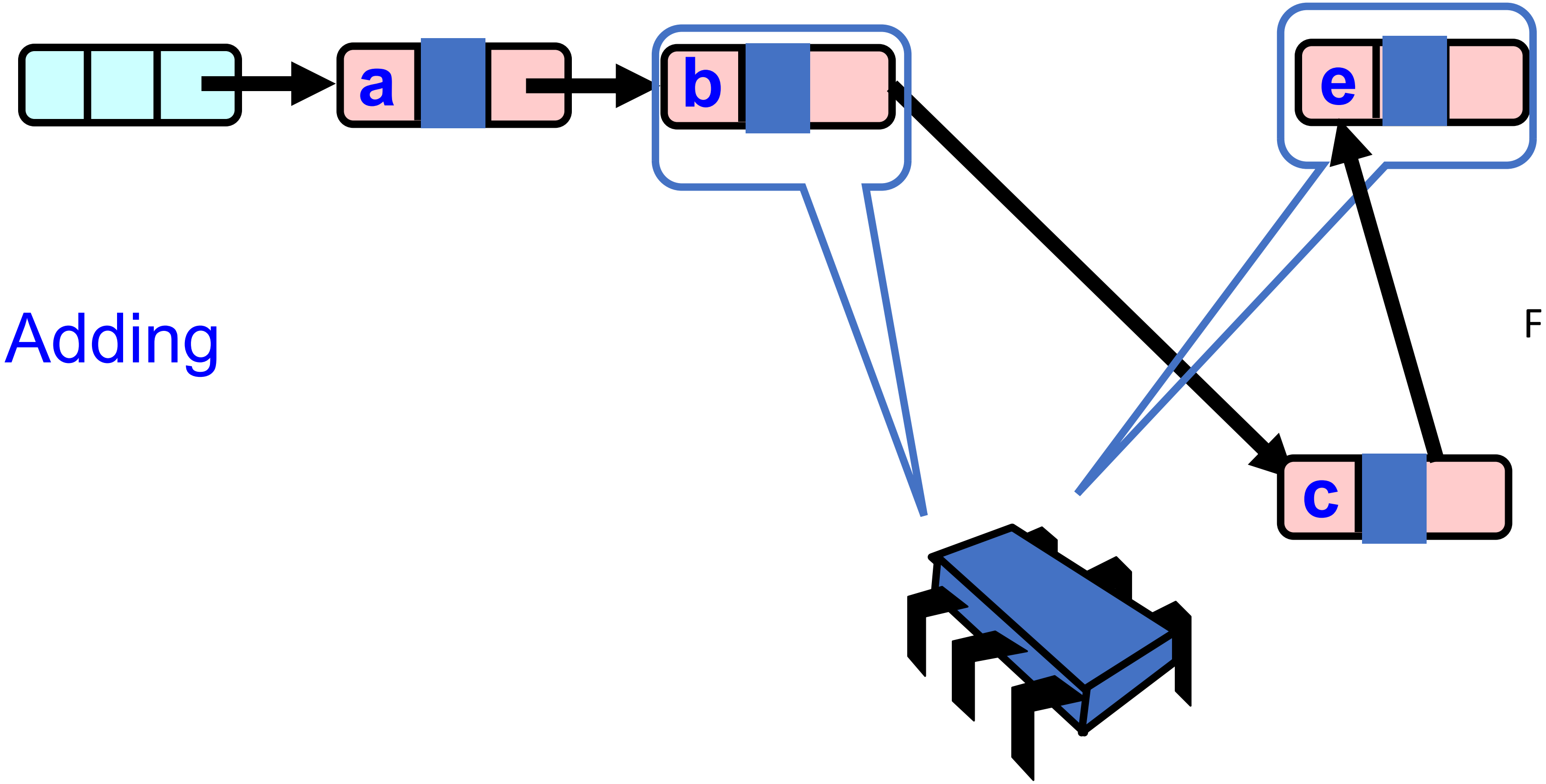
Adding

Find the location

create “c”

insert “c”

Lock-free Lists



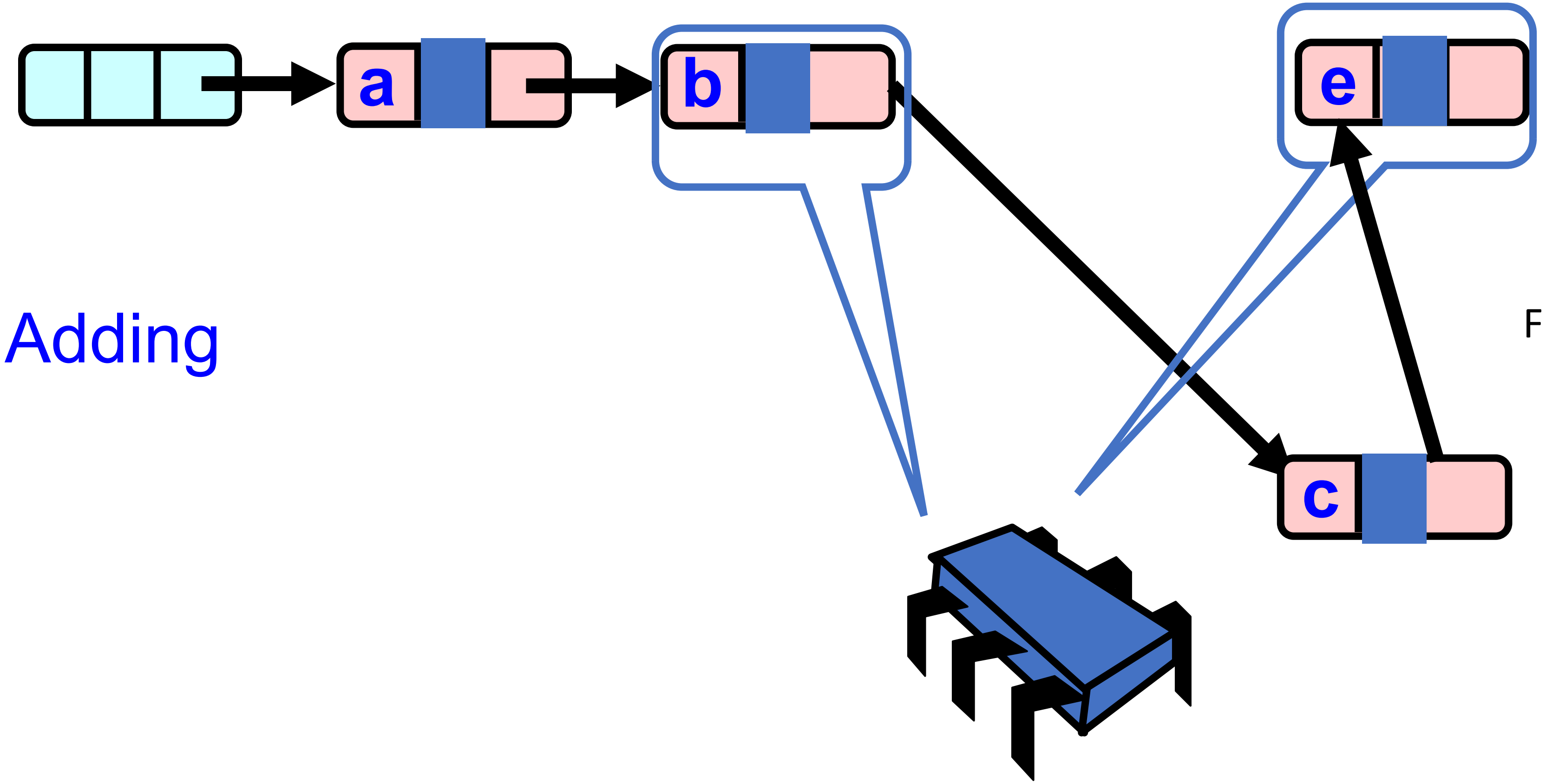
Adding

Find the location

create “c”

insert “c”

Lock-free Lists



Adding

Find the location

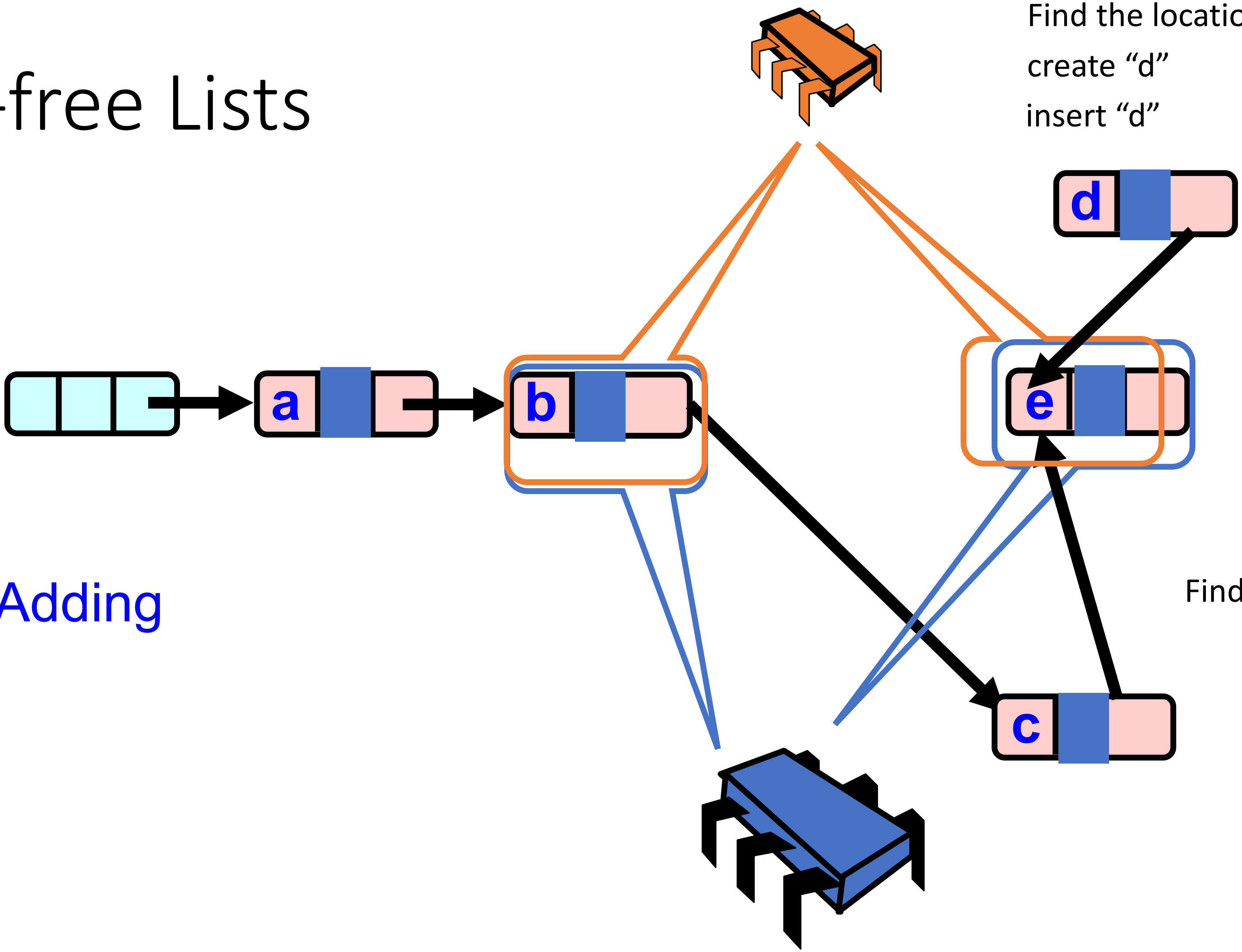
create “c”

insert “c”

Can this just be a regular store?

Lock-free Lists

Adding



Find the location

create “d”

insert “d”

Can this just be a regular store?

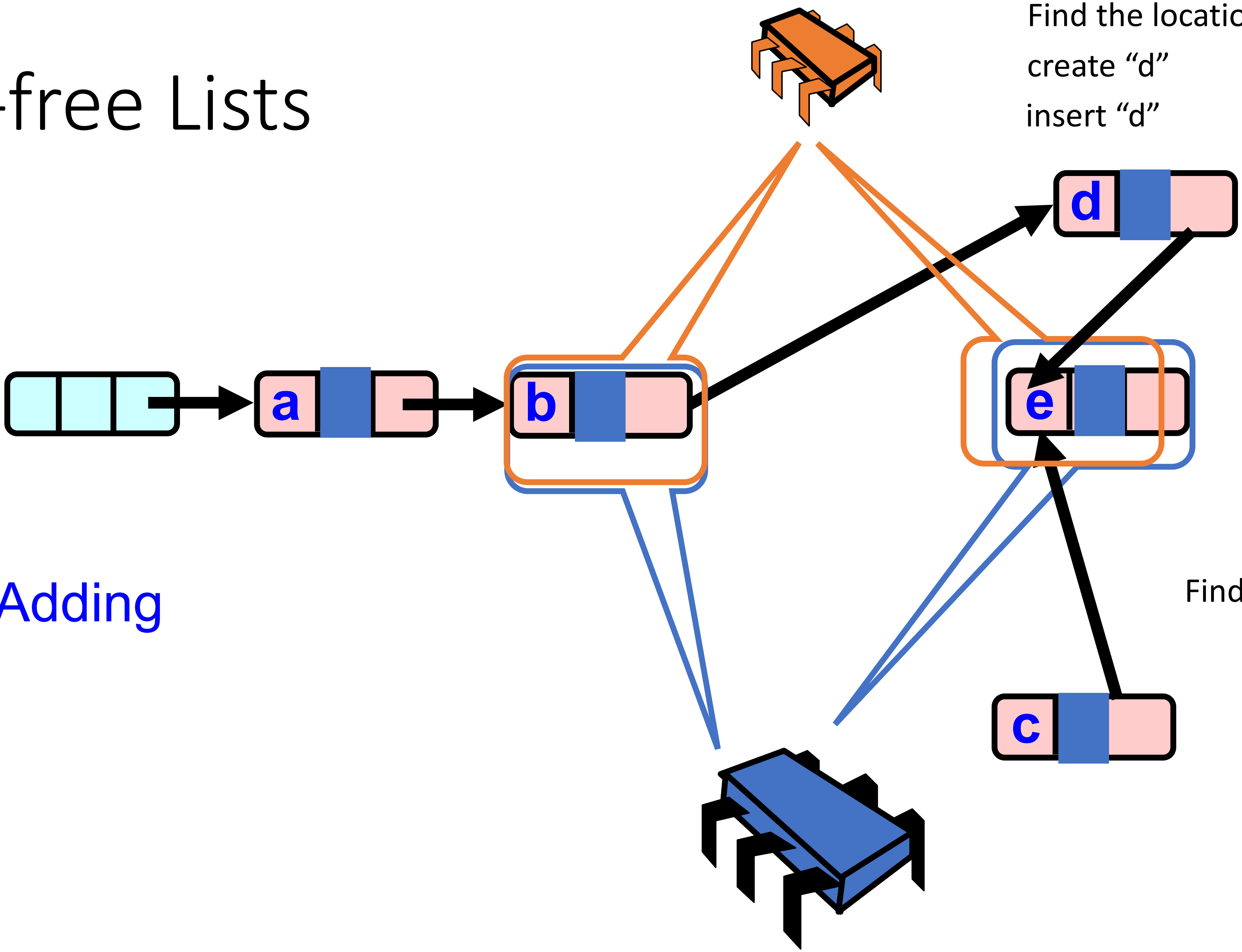
Find the location

create “c”

insert “c”

Can this just be a regular store?

Lock-free Lists



Adding

Find the location

create “d”

insert “d”

Can this just be a regular store?

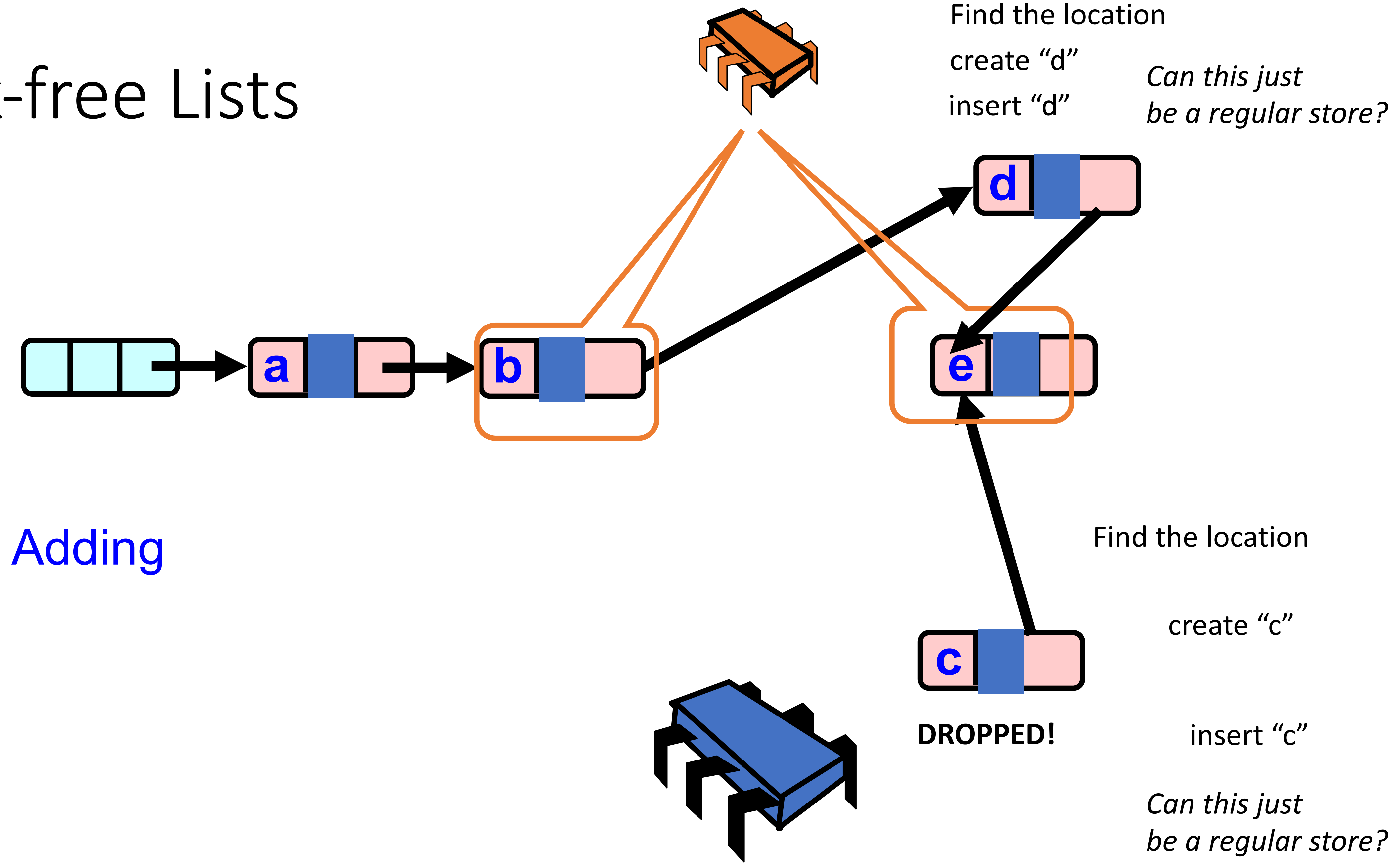
Find the location

create “c”

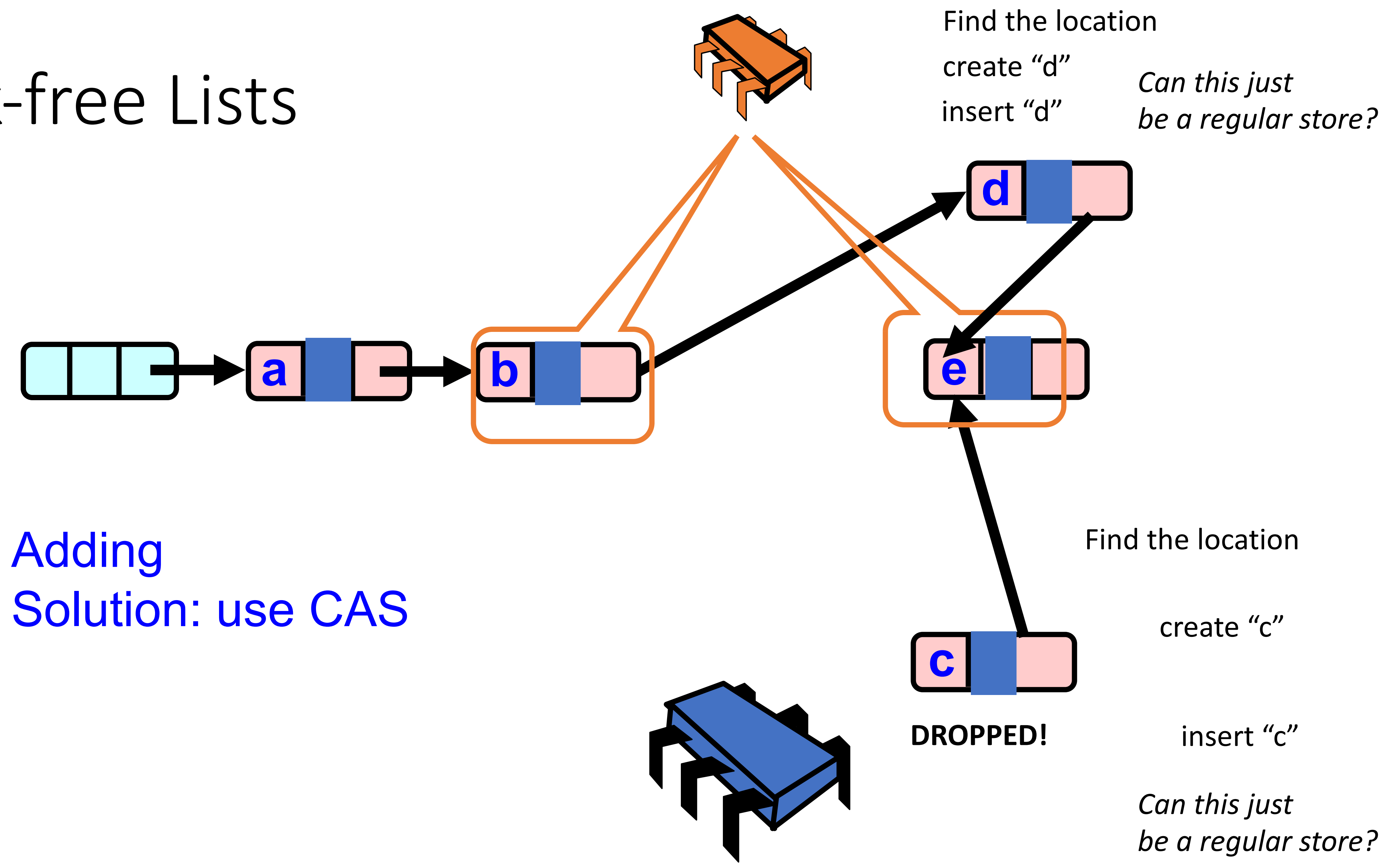
insert “c”

Can this just be a regular store?

Lock-free Lists



Lock-free Lists



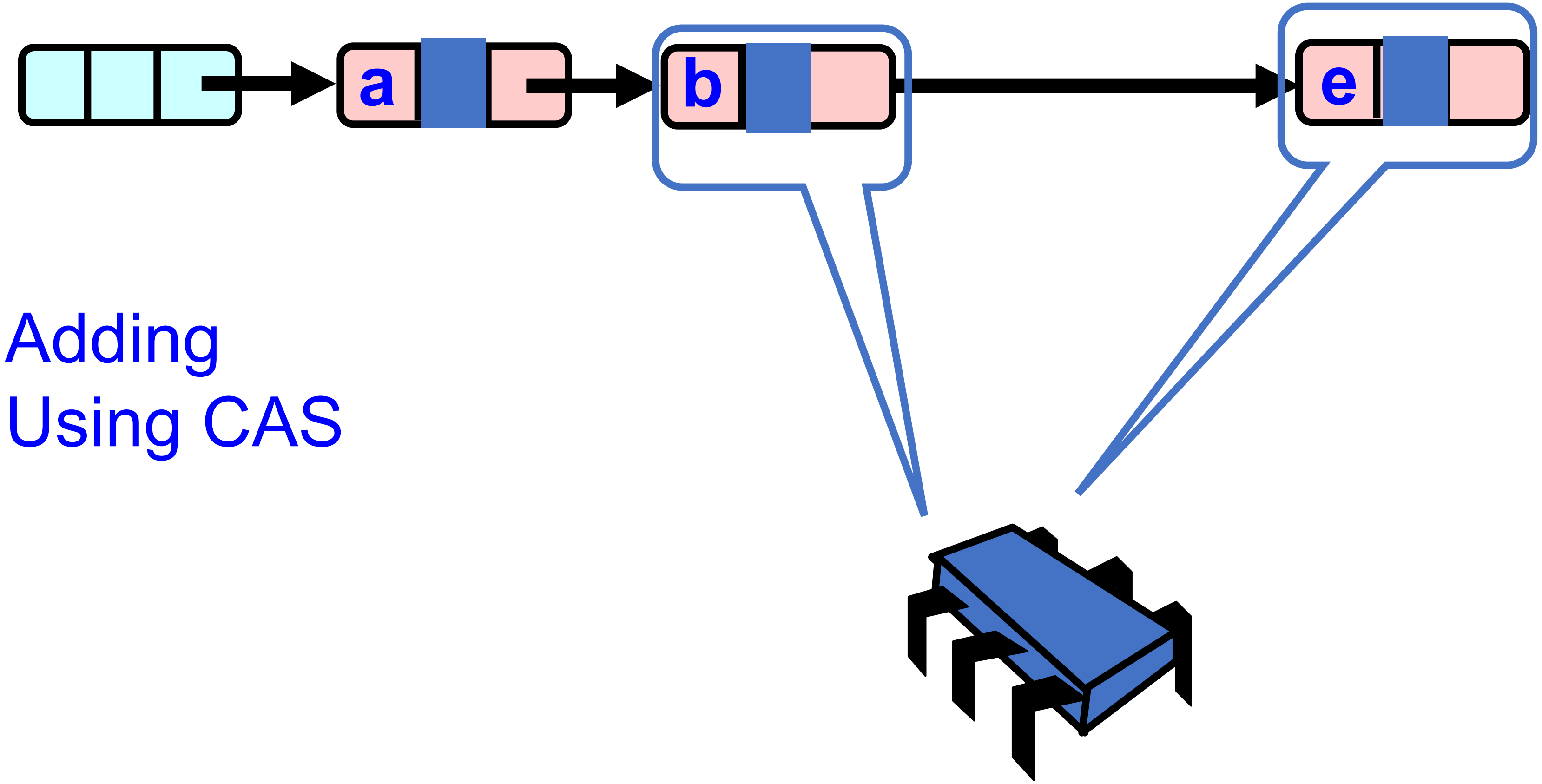
Adding
Solution: use CAS

Lock-free Lists

Find the location

Cache your insertion point!

b.next == e



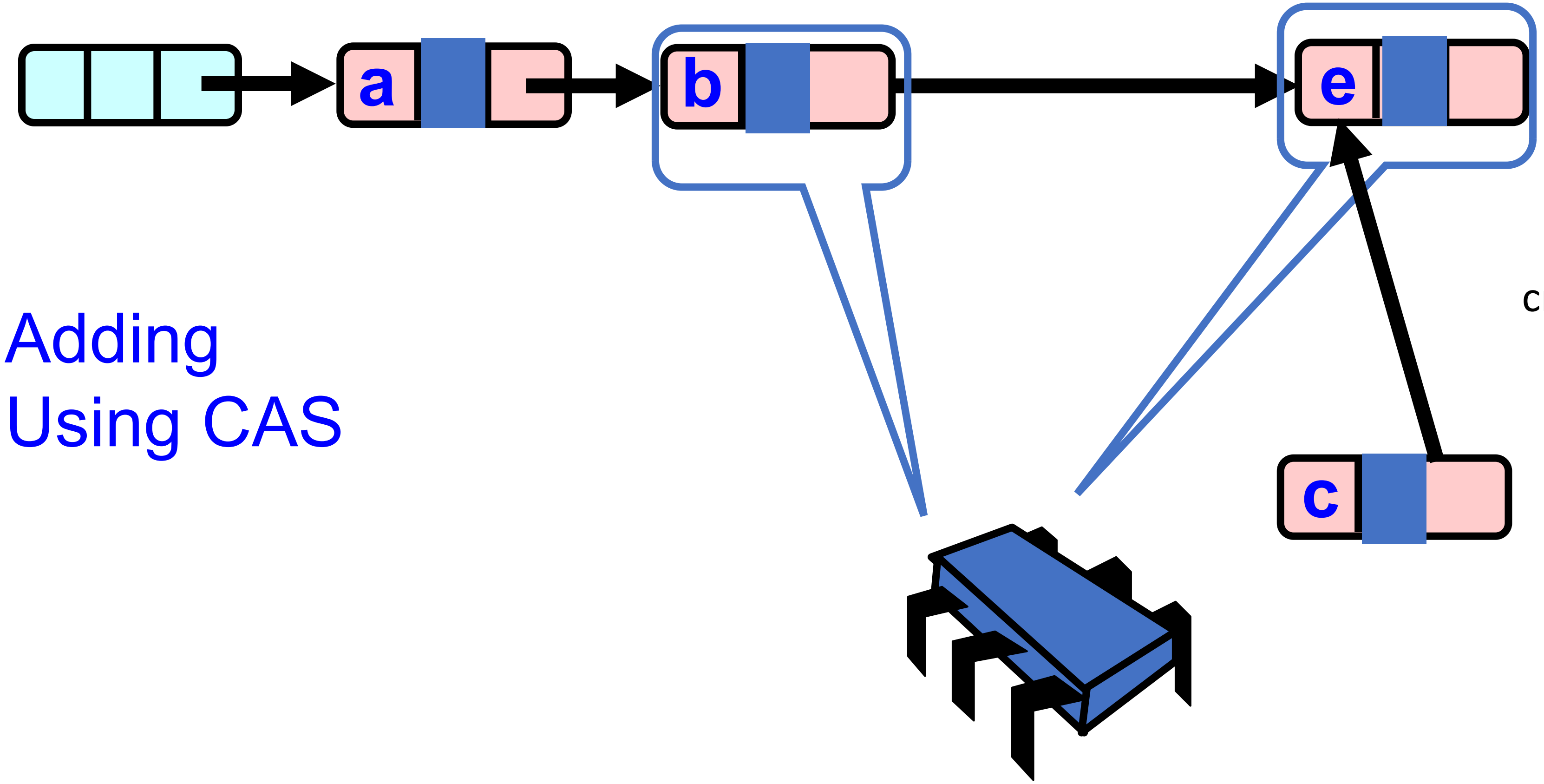
Adding
Using CAS

Lock-free Lists

Find the location

Cache your insertion point!

b.next == e



create “c”

Adding
Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

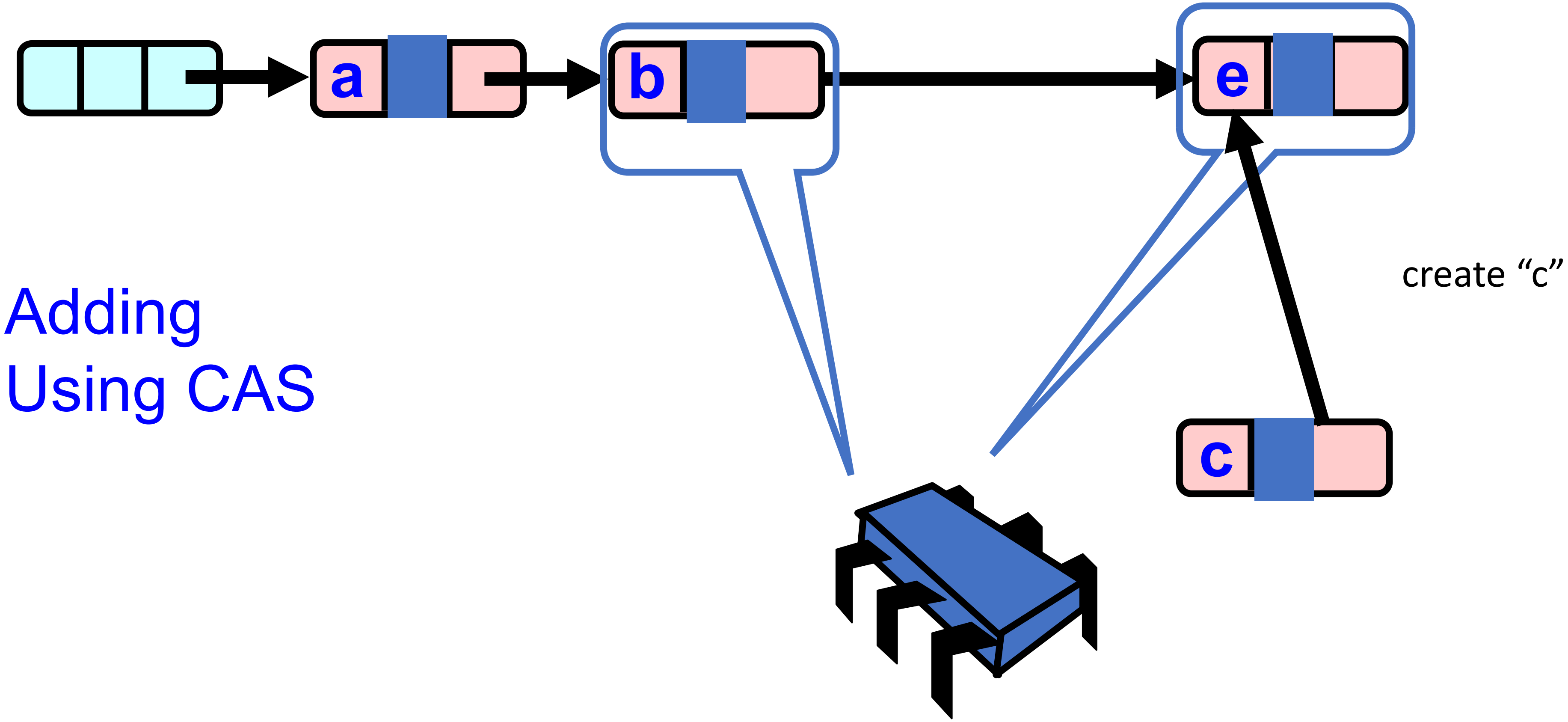
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding
Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

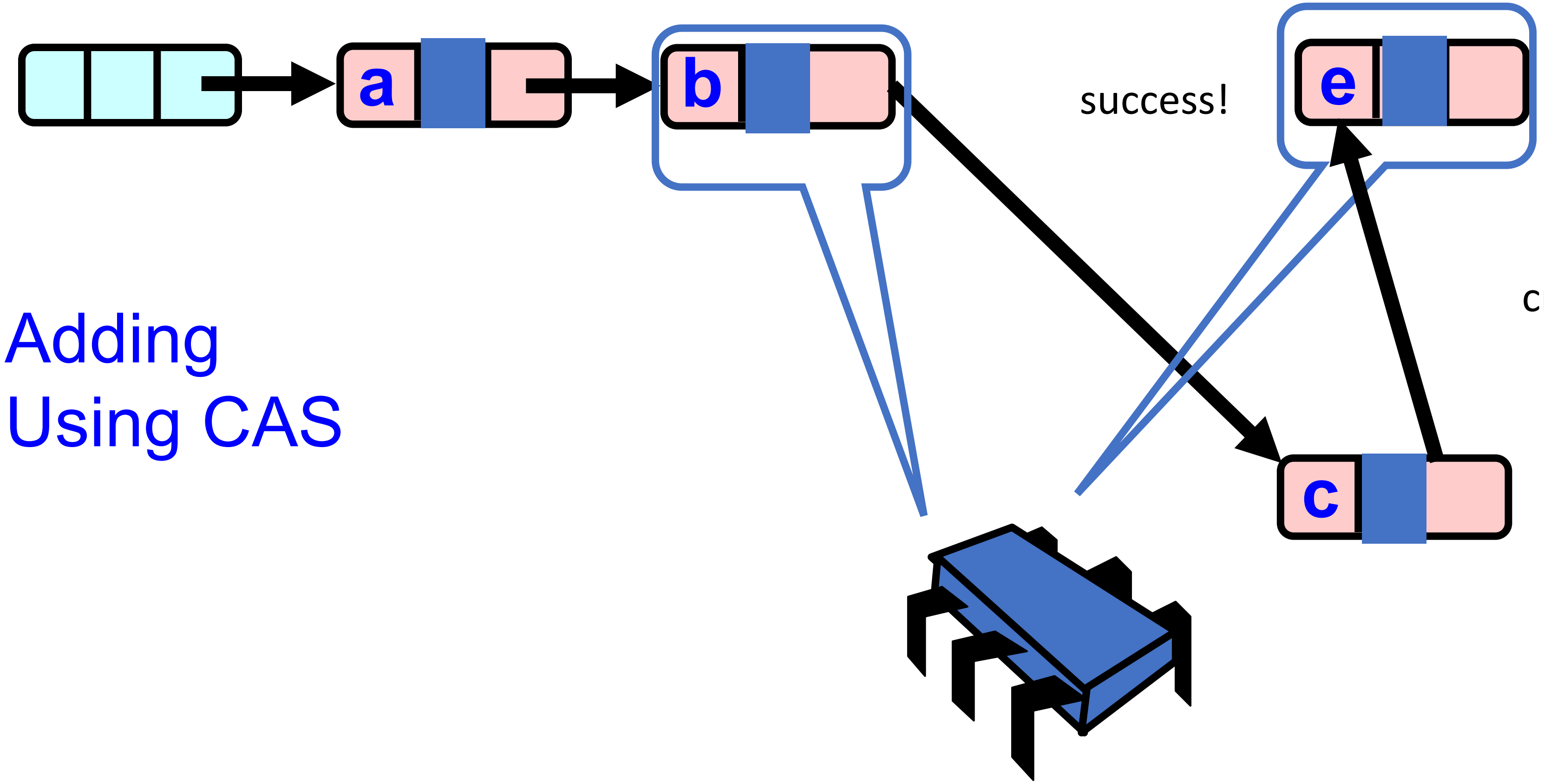
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding
Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

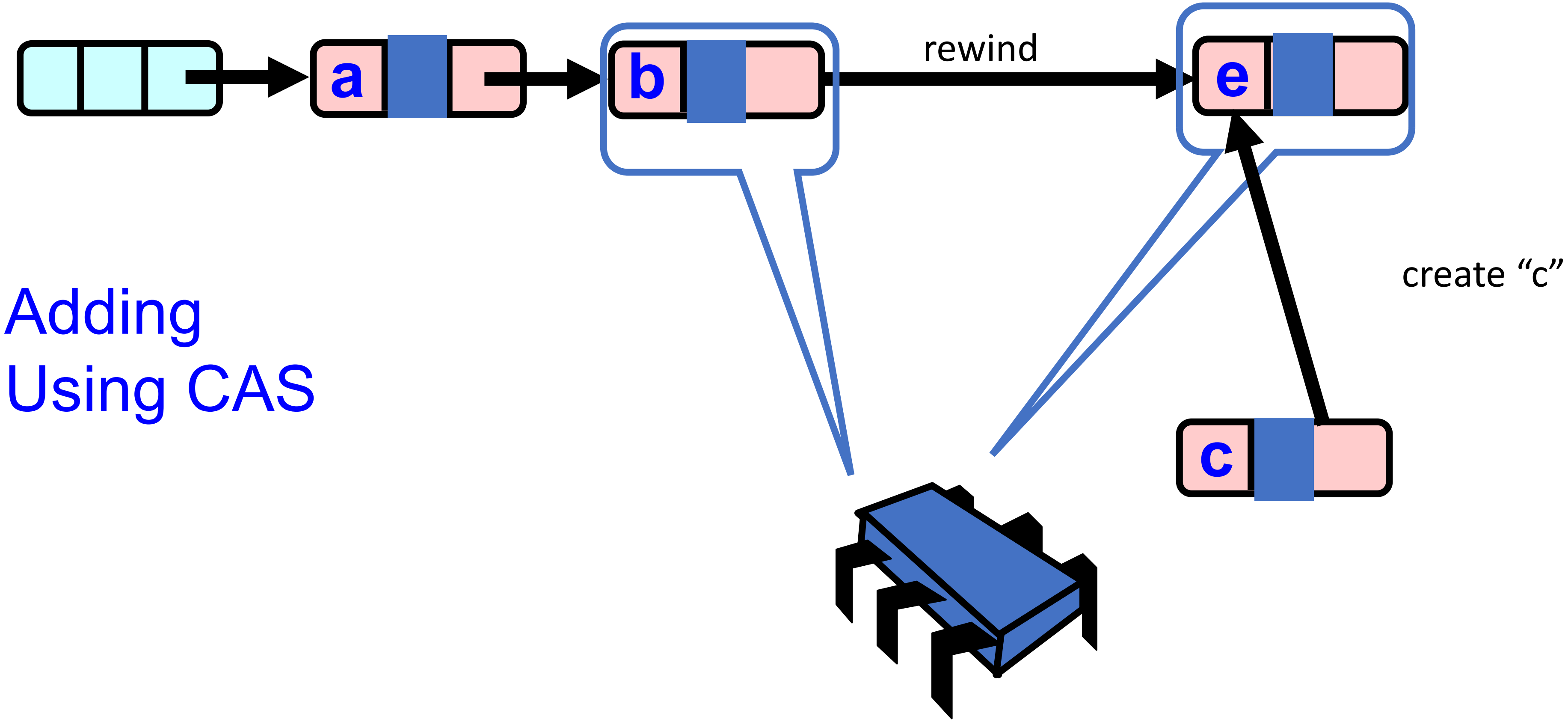
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

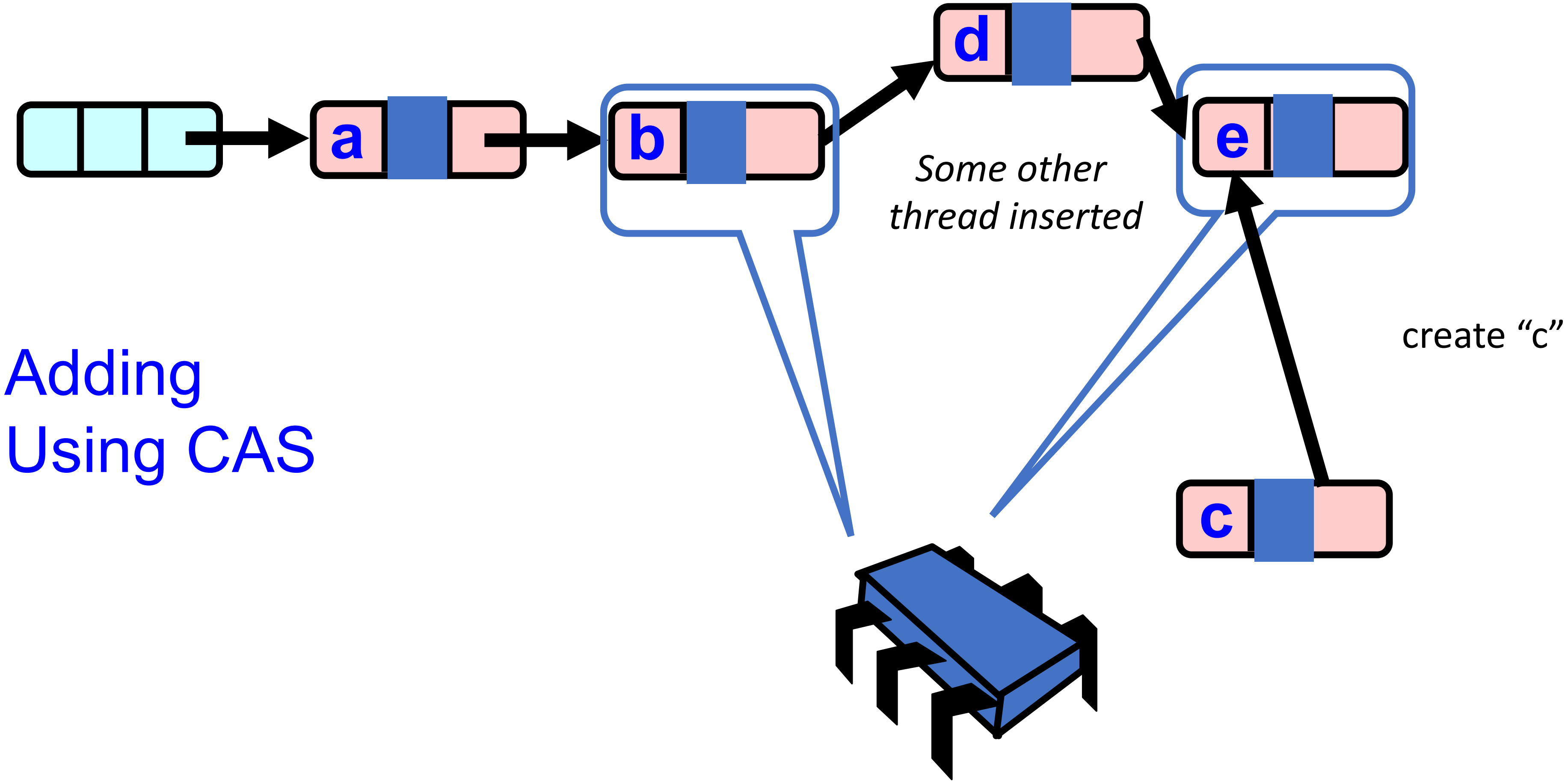
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

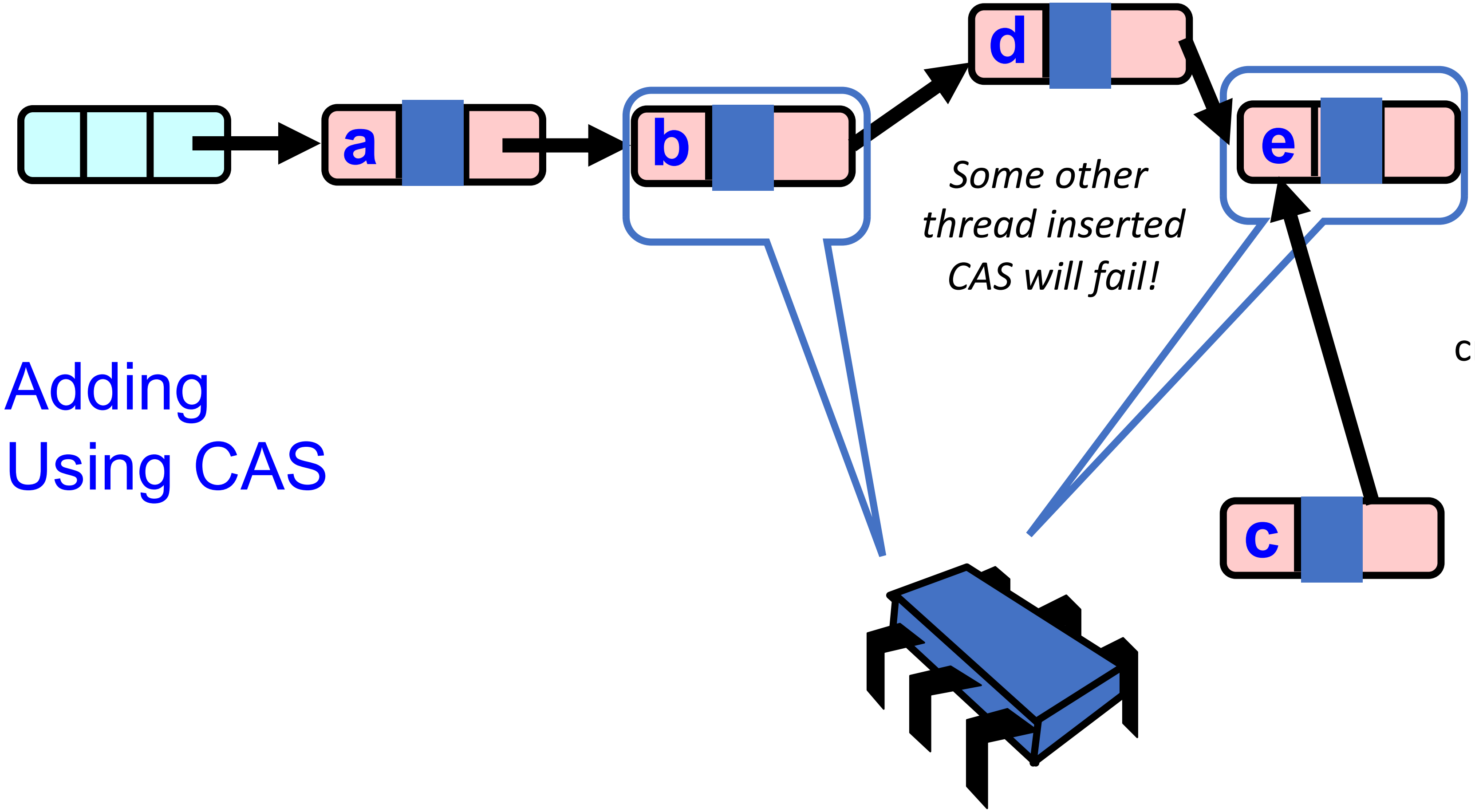
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding
Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

CAS(b.next, e, c);

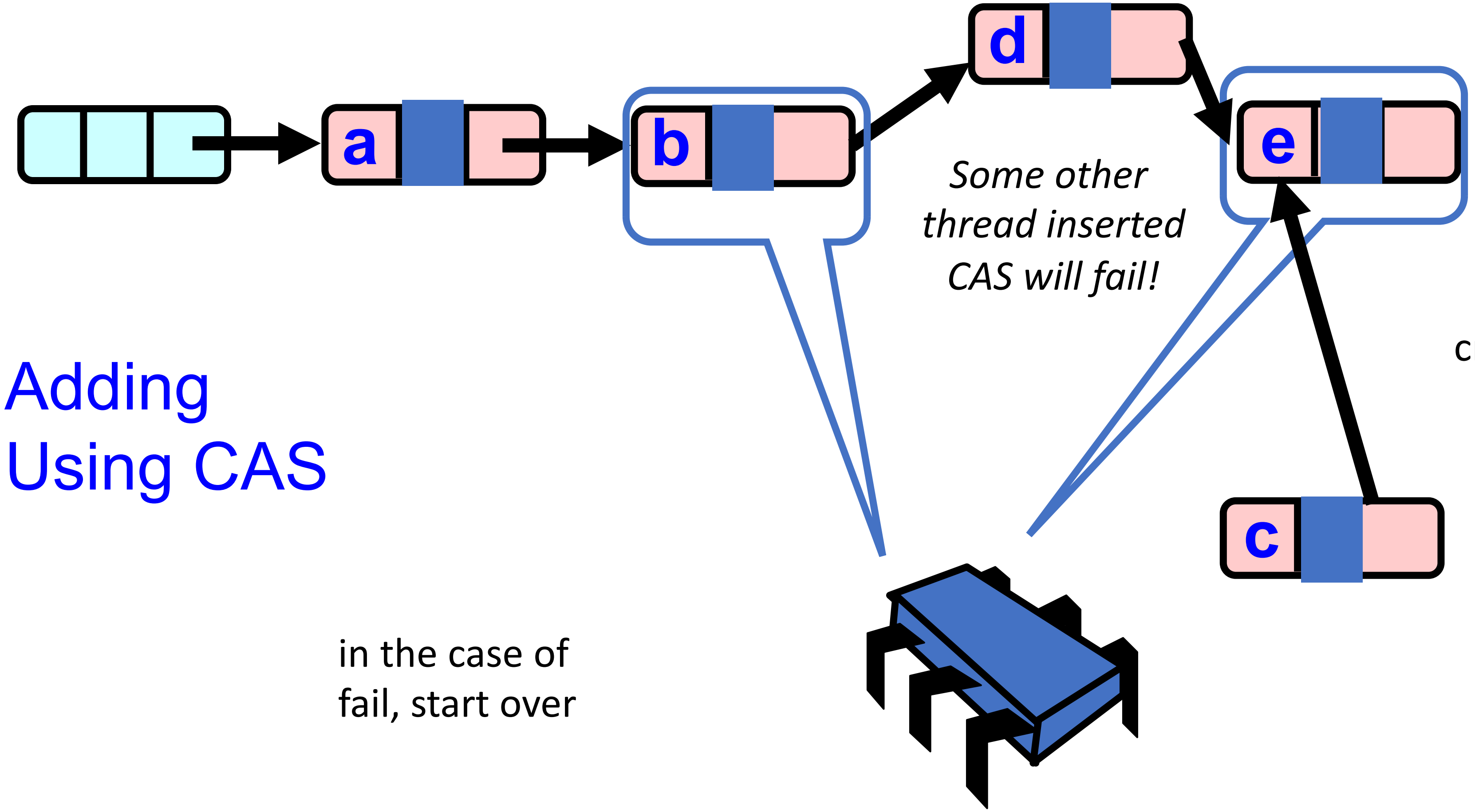
Find the location

Cache your insertion point!

b.next == e

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

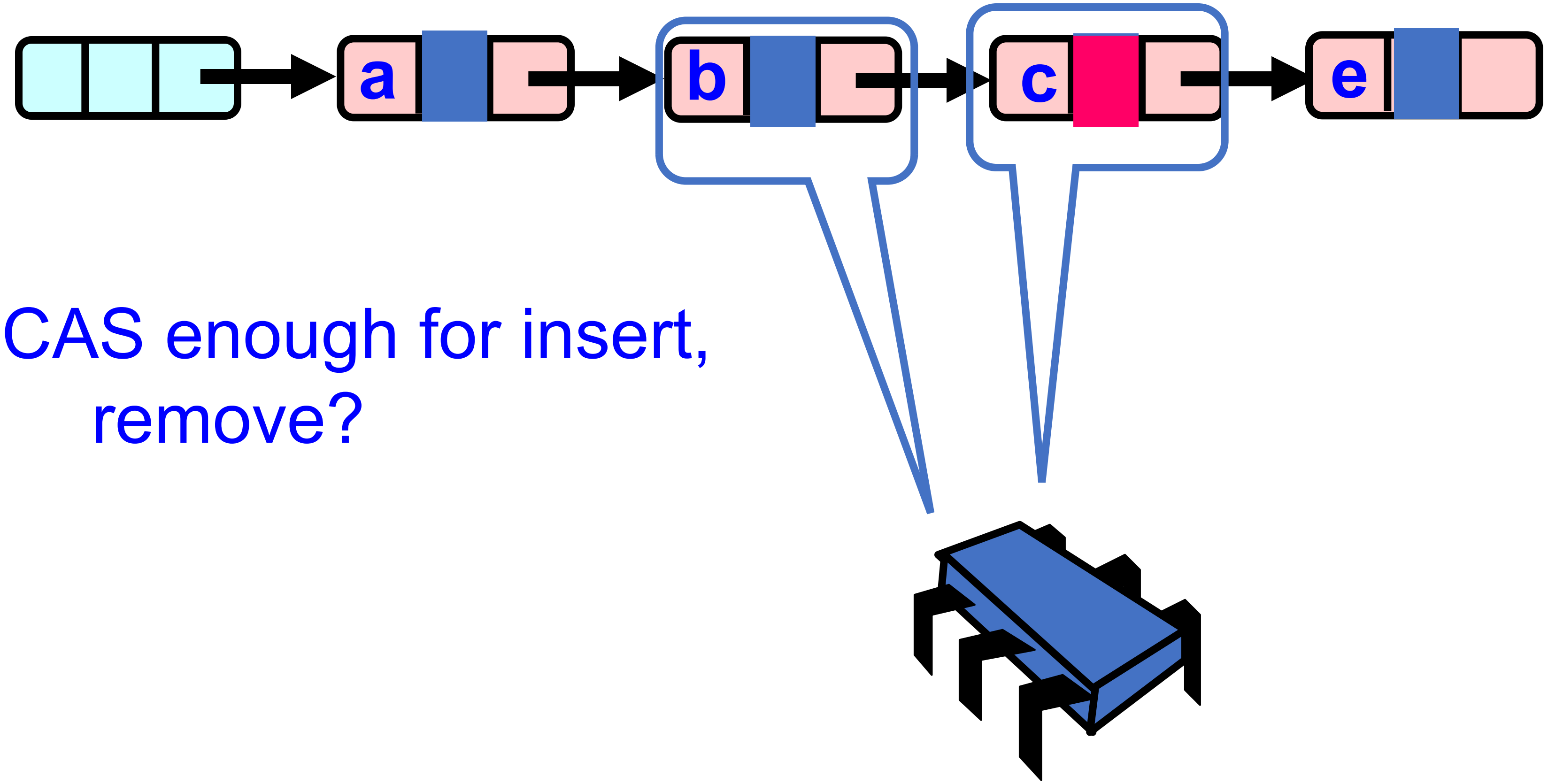
Adding Using CAS



create “c”

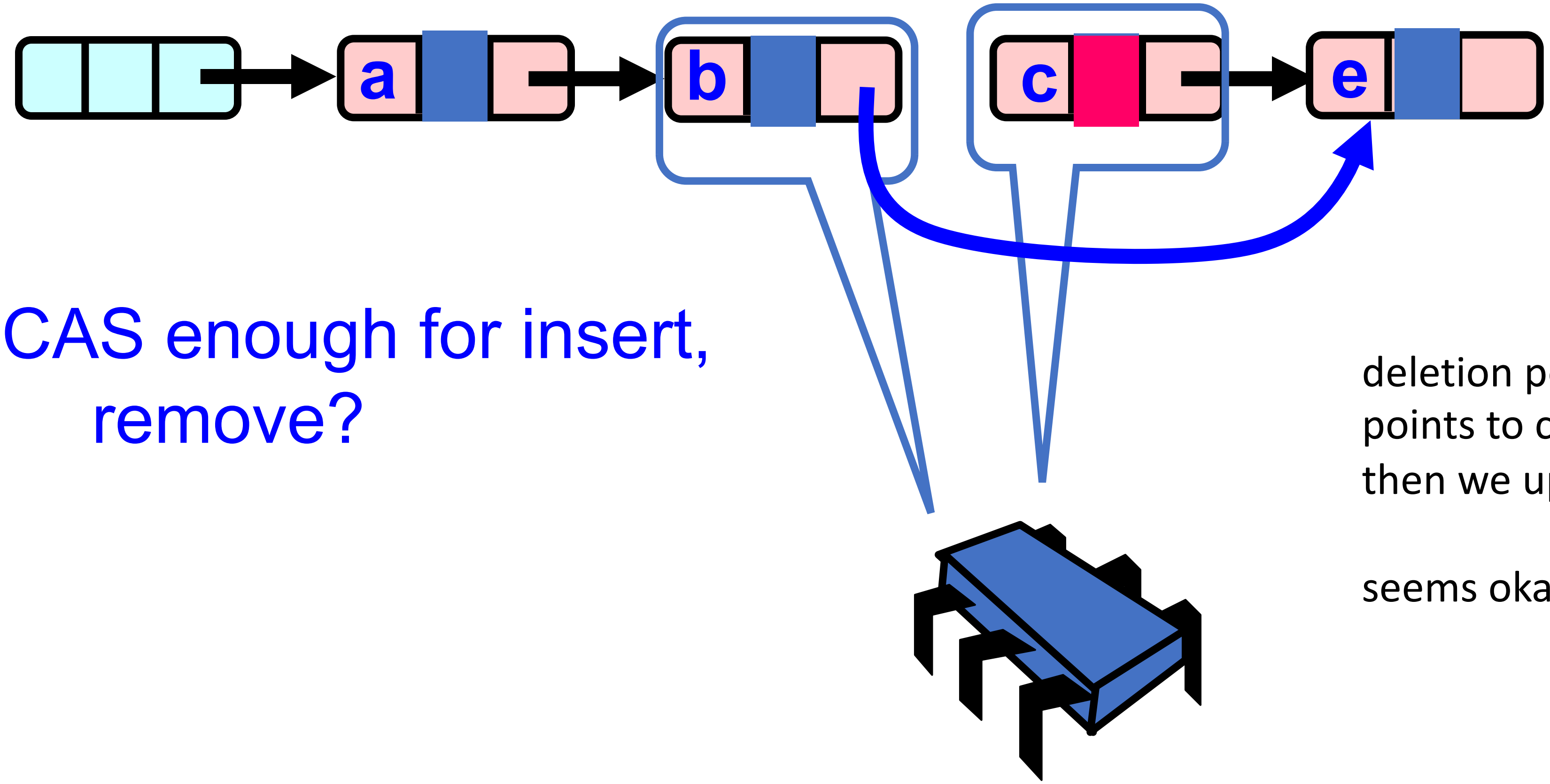
in the case of fail, start over

Lock-free Lists



CAS enough for insert,
remove?

Lock-free Lists



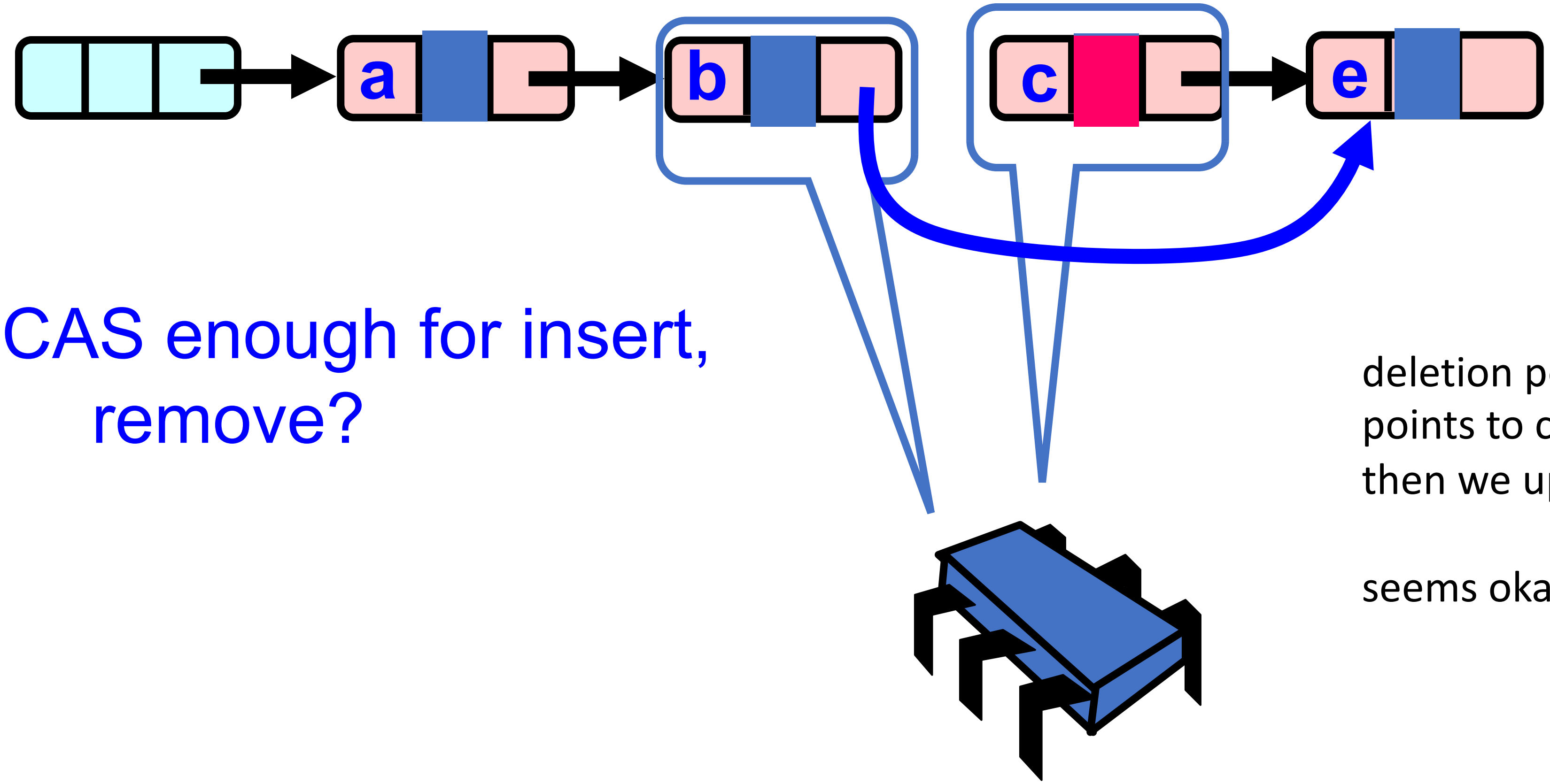
CAS enough for insert, remove?

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



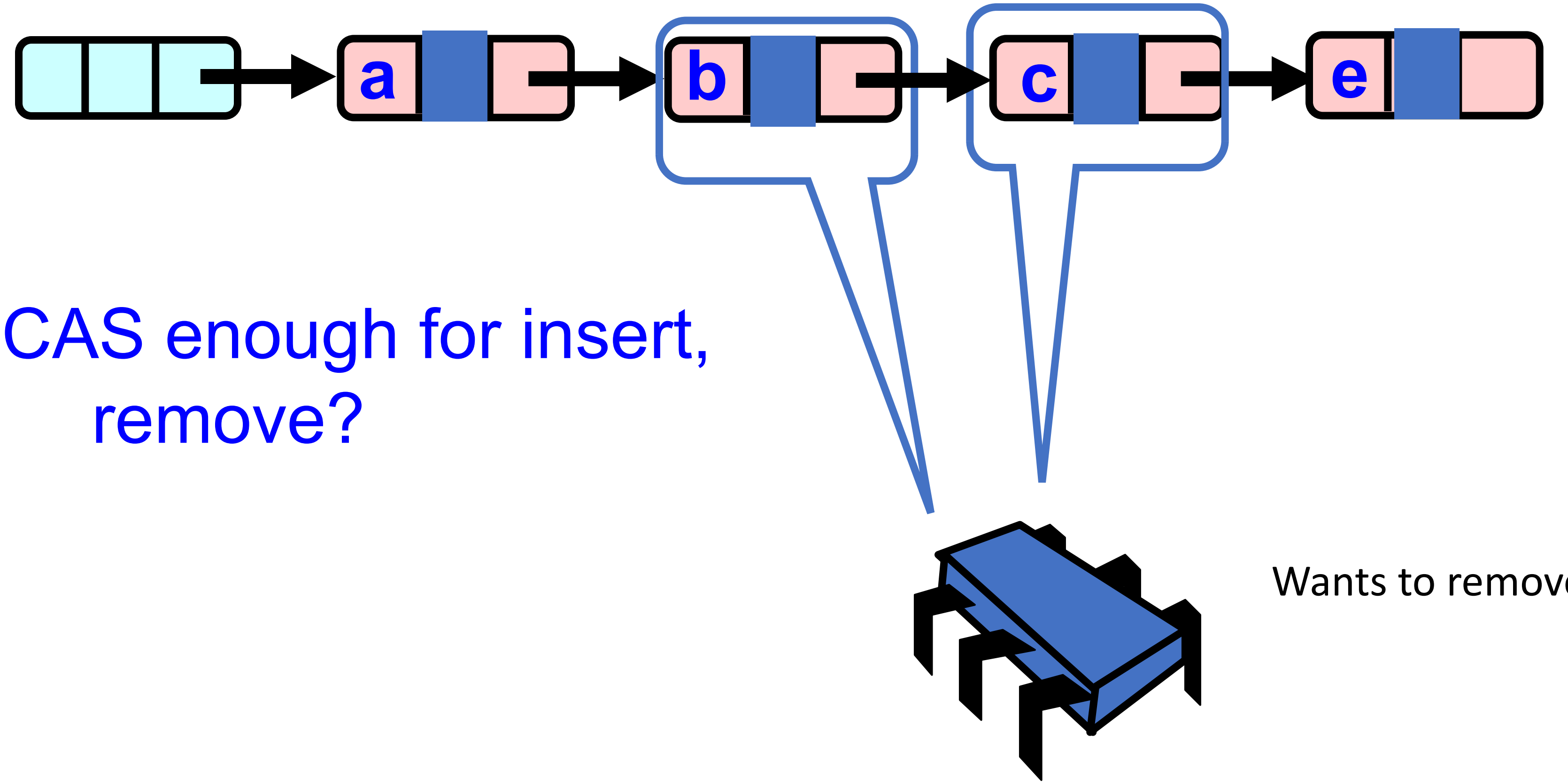
CAS enough for insert, remove?

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

seems okay...

Lock-free Lists

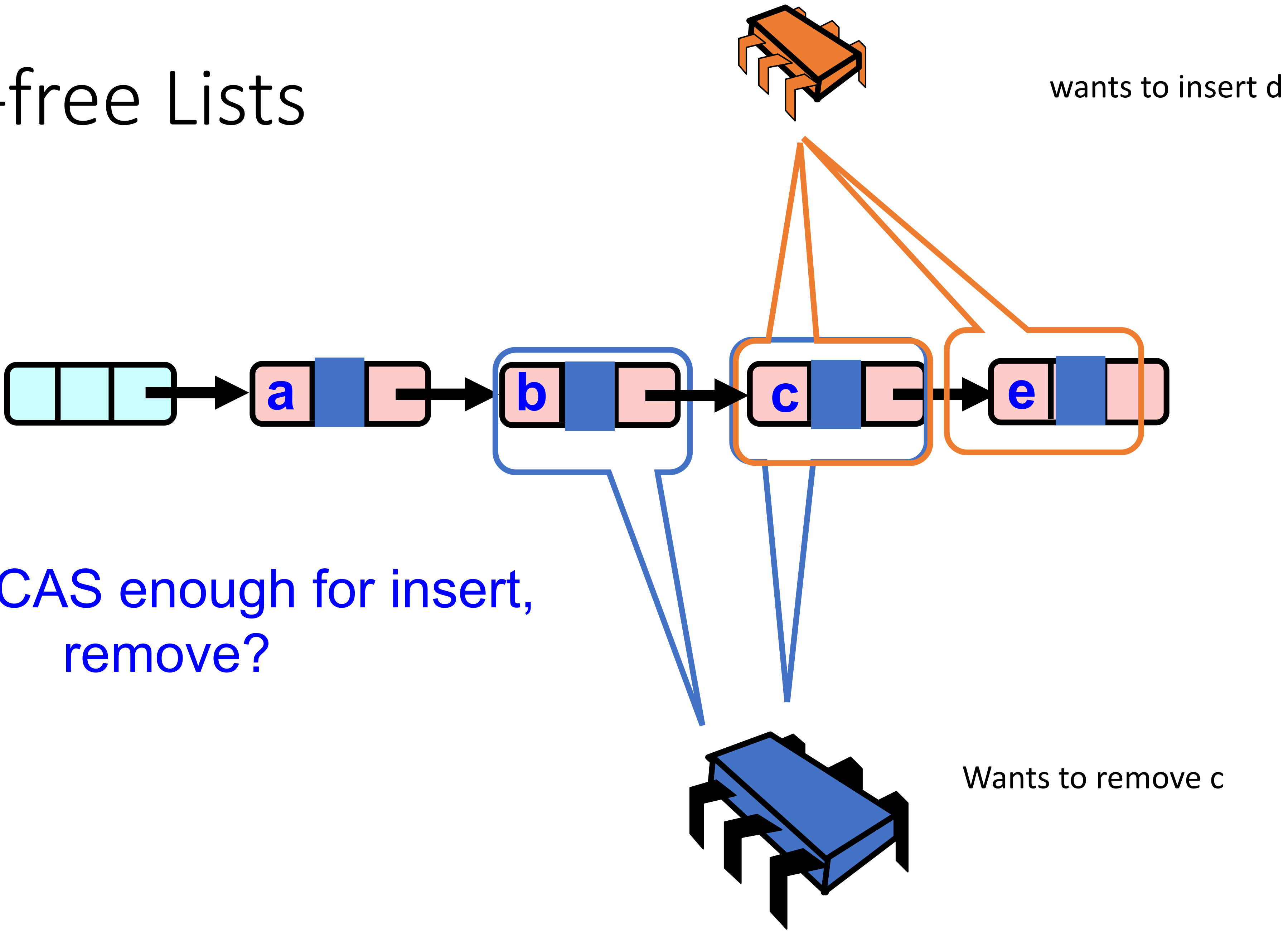
Rewind



CAS enough for insert, remove?

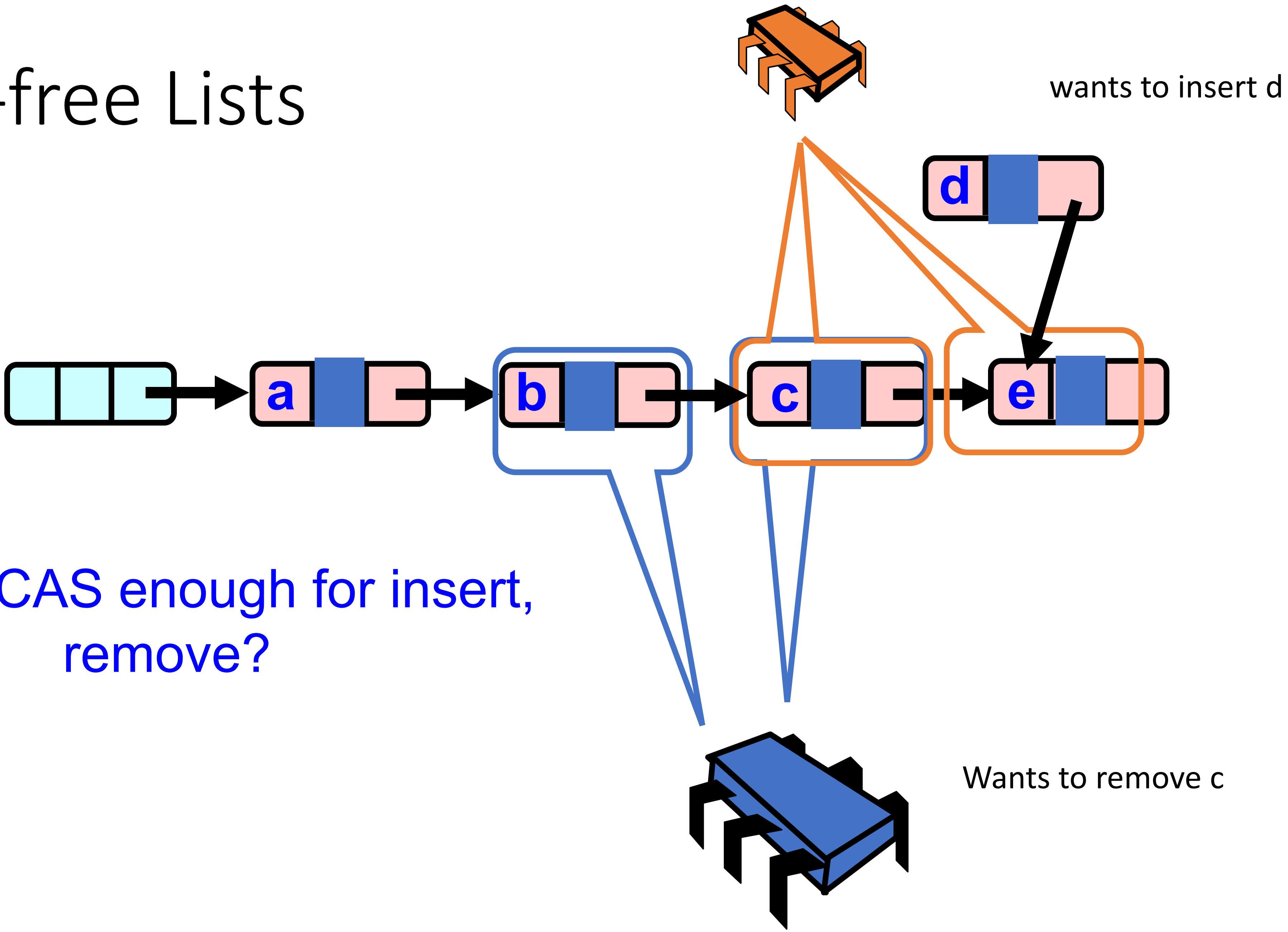
Wants to remove c

Lock-free Lists



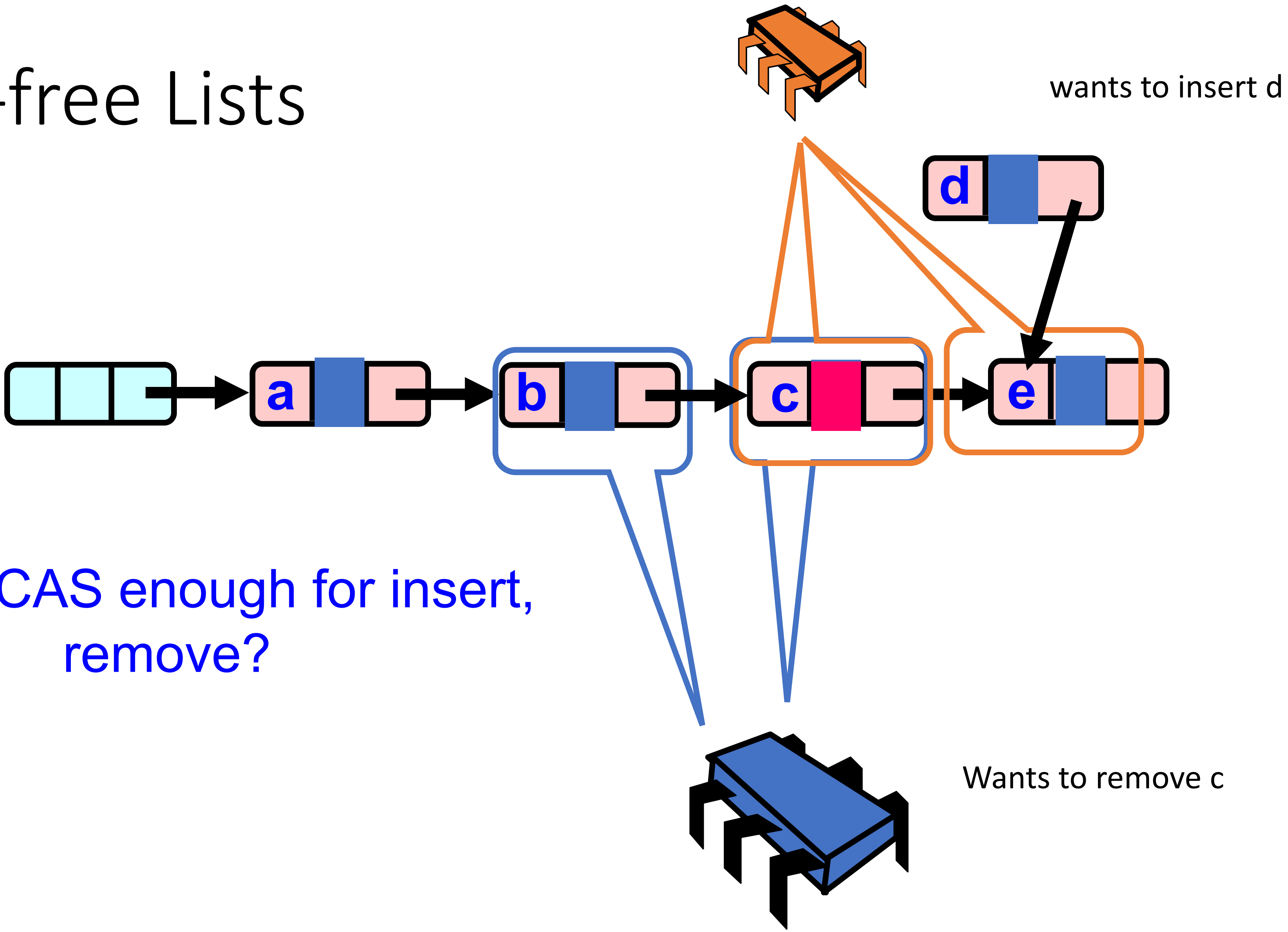
CAS enough for insert, remove?

Lock-free Lists



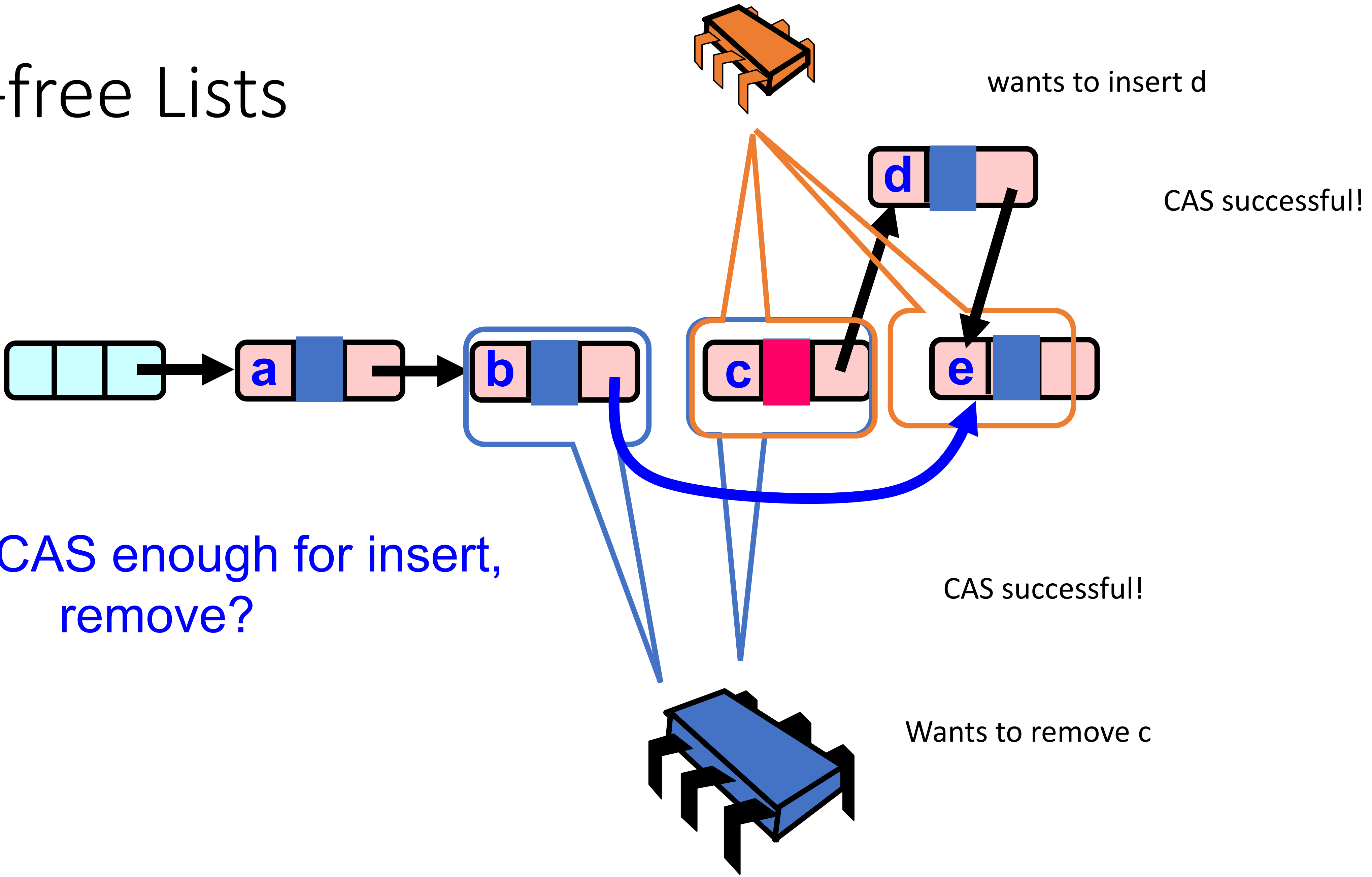
CAS enough for insert, remove?

Lock-free Lists



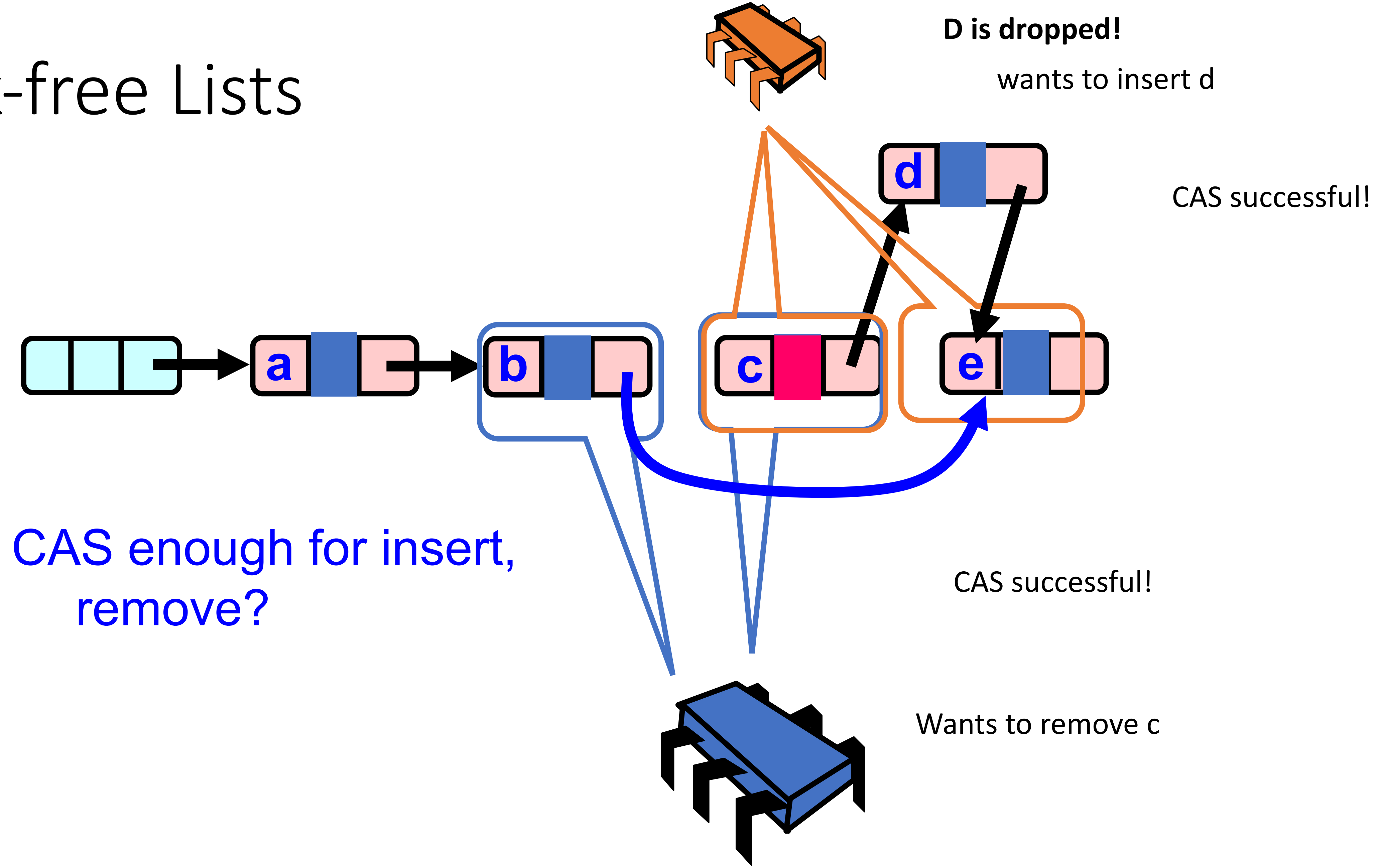
CAS enough for insert,
remove?

Lock-free Lists



CAS enough for insert,
remove?

Lock-free Lists

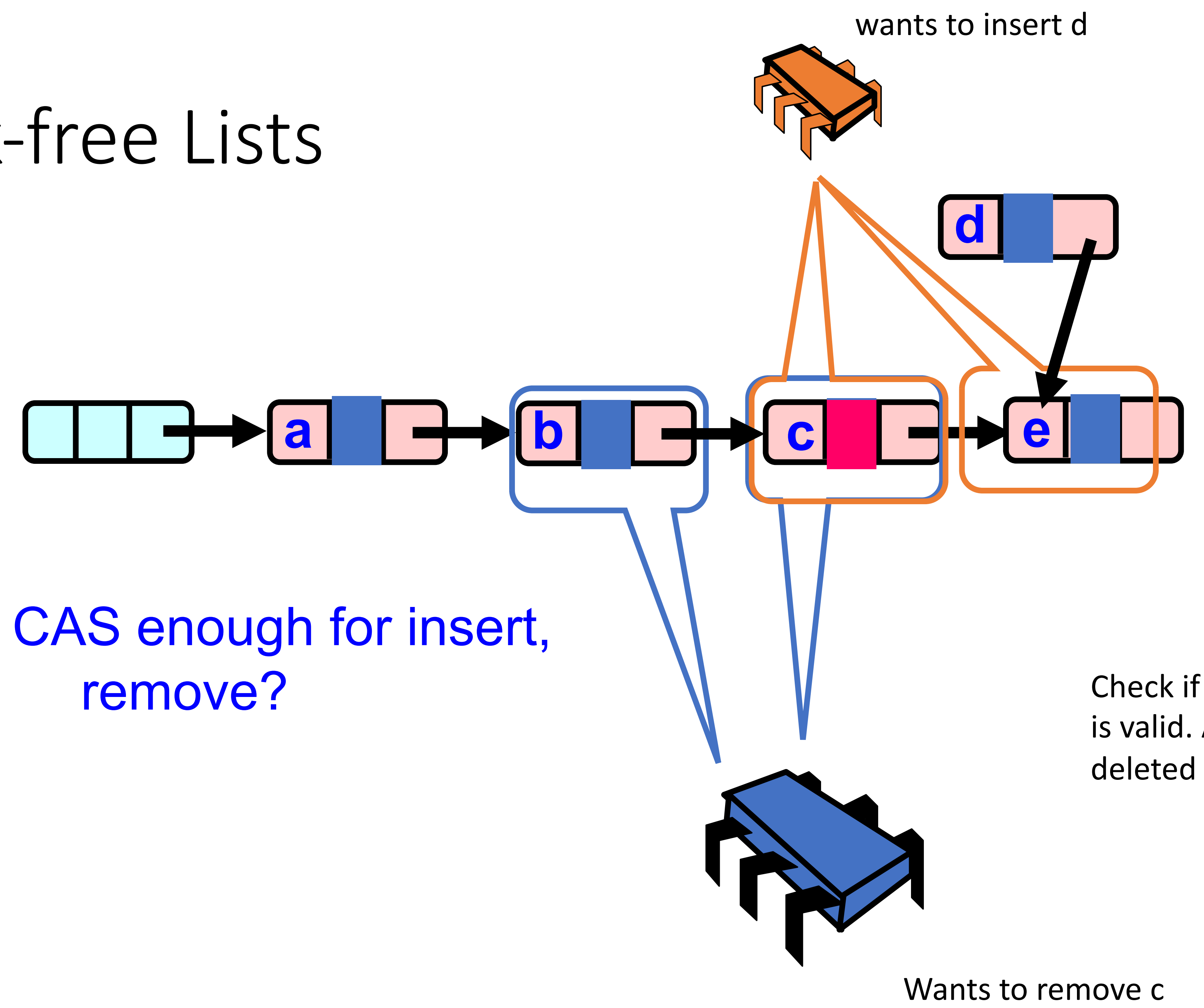


CAS enough for insert, remove?

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 if the node has not been logically removed.

Lock-free Lists



Check if insertion point is valid AND if C is not logically deleted

CAS enough for insert, remove?

Check if insertion point is valid. And B is not logically deleted

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 tools can help (Professor Flanagan is famous for this!)