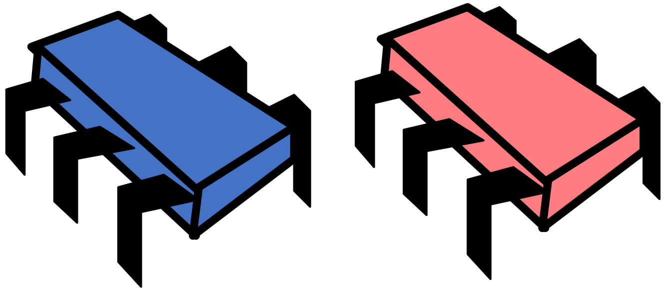
CSE113: Parallel Programming

Feb. 14, 2022

- **Topics**:
  - C++ atomic object templates
  - RMW implementations
  - Continue: general concurrent sets



Announcements

- Midterm is out!
  - Midterm is due today at midnight
  - No late midterms will be accepted
  - No guaranteed help after 5 PM
  - Don’t discuss with friends or internet
- Homework 3 is out
  - Due Friday at midnight
  - You can start discussing results with classmates
- Grades for HW 1 are released
  - You have until Tuesday

Announcements

- Last day on concurrent data structures!
- Moving onto reasoning about concurrency on Wednesday

Today’s Quiz

- Due tomorrow by midnight, please do it!

Previous quiz

Concurrent linked lists can be implemented using locks on every node if:

- ○ locks are always acquired in the same order
- ○ two locks are acquired at a time
- ○ Both of the above
- ○ Neither of the above

Previous quiz

Lock coupling provides higher performance than a single global lock because threads can traverse the list in parallel

- ○ True
- ○ False

Previous quiz

Optimistic concurrency refers to the pattern where functions optimistically assume that no other thread will interfere. In the case where another thread interferes, the program is left in an erroneous state, but since this is so rare, it does not tend to happen in practice.

- ○ True
- ○ False

Previous quiz

After this lecture, do you think you would be able to optimize your implementation of the concurrent stack in homework 2? Write a few sentences on what you might try.

On to the lecture

- Review is baked in

Schedule

- Using atomic templates for objects in C++
- How atomics are implemented in hardware
- Lock-free concurrent set

Schedule

- **Using atomic templates for objects in C++**

- How atomics are implemented in hardware

- Lock-free concurrent set

C++ Atomic template

- C++ lets you wrap custom objects/types as an atomic type
- included in `<atomic>`
- use like this:
  - `atomic<int> i;`
  - `atomic<float> f;`

C++ Atomic template

- Lets you:
  - `load`
  - `store`
  - `exchange`
  - `compare_and_swap`

- It may use a lock behind the scenes!

- *Examples*

C++ Atomic template

- If you do this to a class, you will lose access to your methods!

- Example:

C++ Atomic template

- How to get your methods back?
  - Make a local copy!

```
bank_account local = tylers_account.load();
local.buy_coffee();
tylers_account.store(local);
```

```
bank_account local = tylers_account.load();
local.work_one_hour();
tylers_account.store(local);
```

What happens when we run this?

C++ Atomic template

- How to get your methods back?
  - Make a local copy!

Consider 1 iteration
(should end with balance of 0):

thread 0 loads an account of balance 0
thread 1 loads an account of balance 0
thread 0 buys coffee (local account -1)
thread 1 works 1 hour (local account 1)
thread 0 stores local back (global balance -1)
thread 1 stores local back (global balance 1)

thread 0

```
bank_account local = tylers_account.load();
local.buy_coffee();
tylers_account.store(local);
```

thread 1

```
bank_account local = tylers_account.load();
local.work_one_hour();
tylers_account.store(local);
```

C++ Atomic template

- How to get your methods back?
  - Make a local copy!

Consider 1 iteration
(should end with balance of 0):

thread 0 loads an account of balance 0
thread 1 loads an account of balance 0
thread 0 buys coffee (local account -1)
thread 1 works 1 hour (local account 1)
thread 0 stores local back (global balance -1)
thread 1 stores local back (global balance 1)

thread 0

```
bank_account local = tylers_account.load();
local.buy_coffee();
tylers_account.store(local);
```

thread 1

```
bank_account local = tylers_account.load();
local.work_one_hour();
tylers_account.store(local);
```

Overall issue: memory accesses are atomic, actions are not!

C++ Atomic template

- CAS to the rescue!
- Optimistically load object and operate on it.
- Before we store, check to see if its changed
- If it has changed, try again.
- Need to do check and store atomically.

C++ Atomic template

```
bank_account snapshot;
bank_account update;
bool success;

do {
bank_account snapshot = tylers_account.load();
bank_account update = snapshot;
update.buy_coffee();
success = tylers_account.compare_exchange_strong(snapshot, update);
} while (success == false);
```

Recall atomic templates allow you to do compare and swap

```
bank_account snapshot;
bank_account update;
bool success;

do {
bank_account snapshot = tylers_account.load();
bank_account update = snapshot;
update.buy_coffee();
success = tylers_account.compare_exchange_strong(snapshot, update);
} while (success == false);
```

Consider 2 threads thread 0 is buying coffee and thread 1 is working

```
thread 0 loads global balance 0 (snapshots 0)

thread 1 loads global balance 0 (snapshots 0)

thread 0 buys coffee            (updated account -1)

thread 1 works 1 hour           (updated account 1)

thread 0 CAS updated back       (global balance 0, snapshot 0, updated 1) - SUCCESS

thread 1 CAS updated back       (global balance 1, snapshot 0, updated -1) - FAIL
```

thread 1 retries:

```
thread 1 loads global balance 1 (snapshots 1)
```

Question:

- Is it fair?

- What are the pros and cons of this approach?

Demo

- Performance

- How does lock freedom effect things?

Schedule

- Using atomic templates for objects in C++
- **How atomics are implemented in hardware**
- Lock-free concurrent set

How is CAS (and others) implemented?

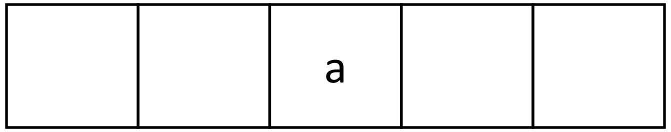
- X86 has an actual instruction
- ARM and POWER are load linked store conditional

Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start

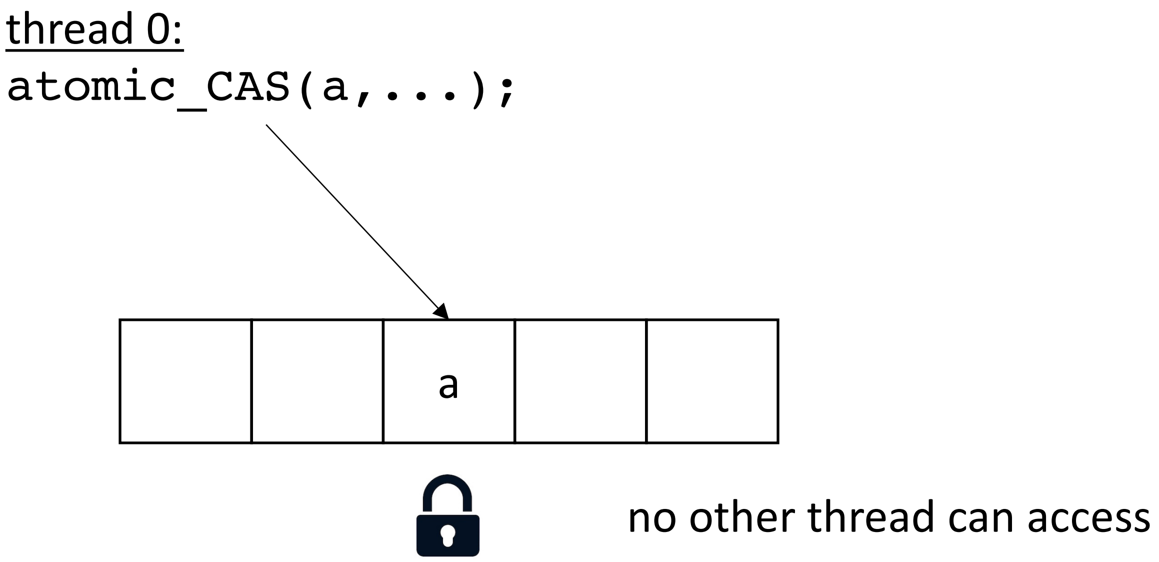
thread 0:

```
atomic_CAS(a,...);
```



Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start

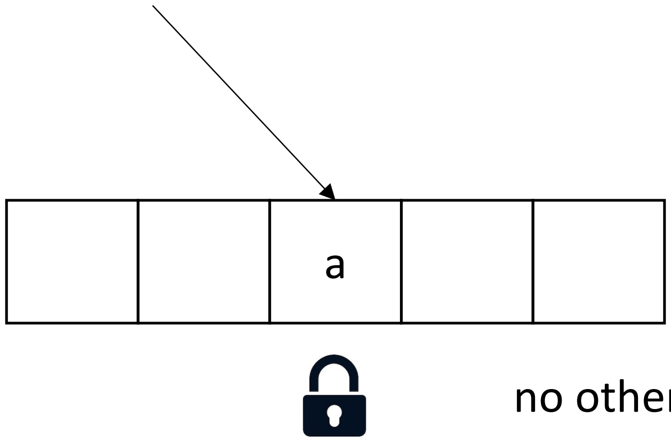


Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start

thread 0:
```
atomic_CAS(a,...);
```

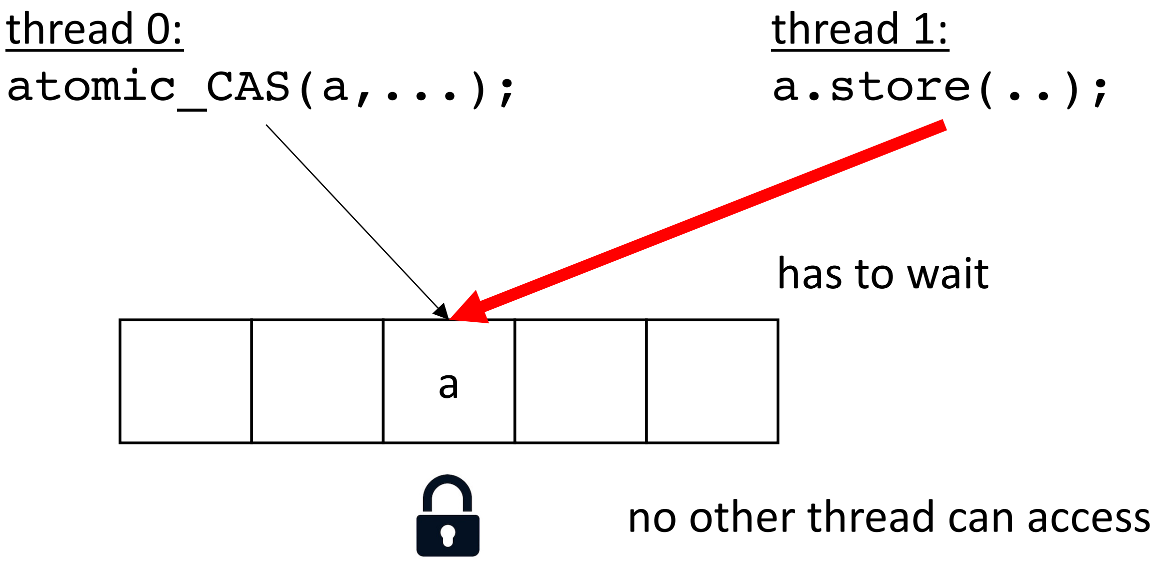
thread 1:
```
a.store(..);
```



no other thread can access

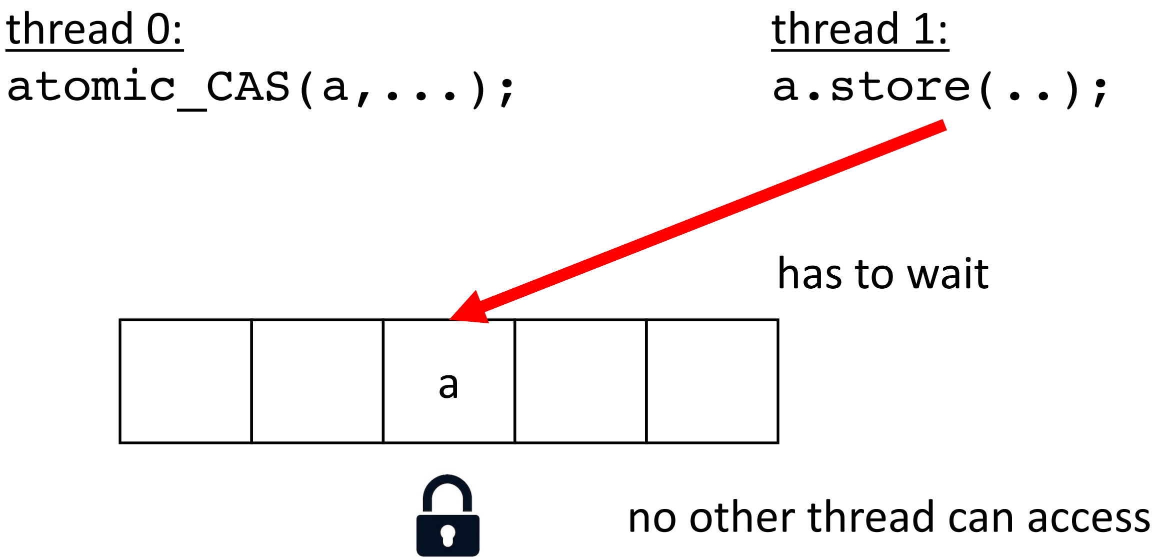
Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



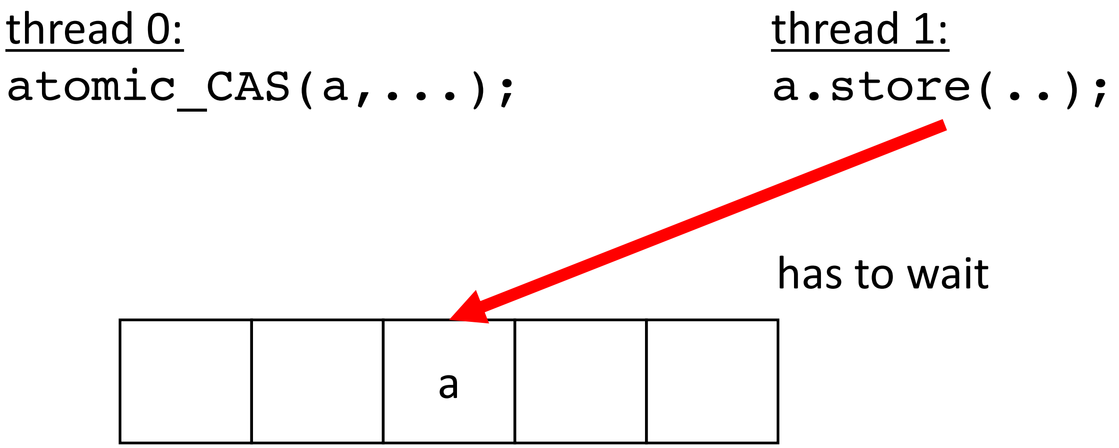
Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



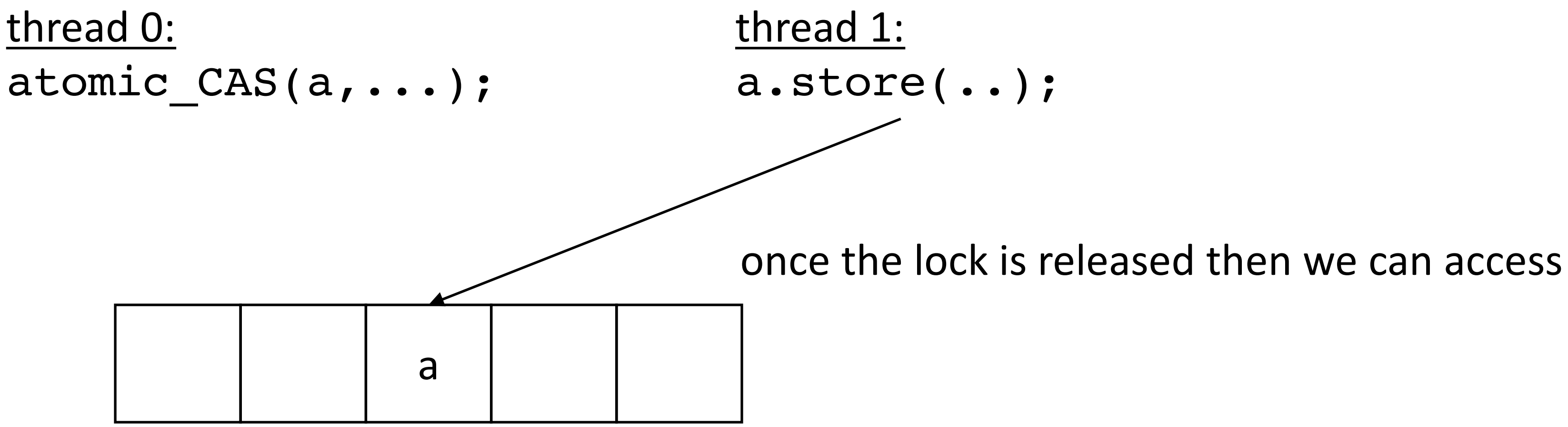
Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



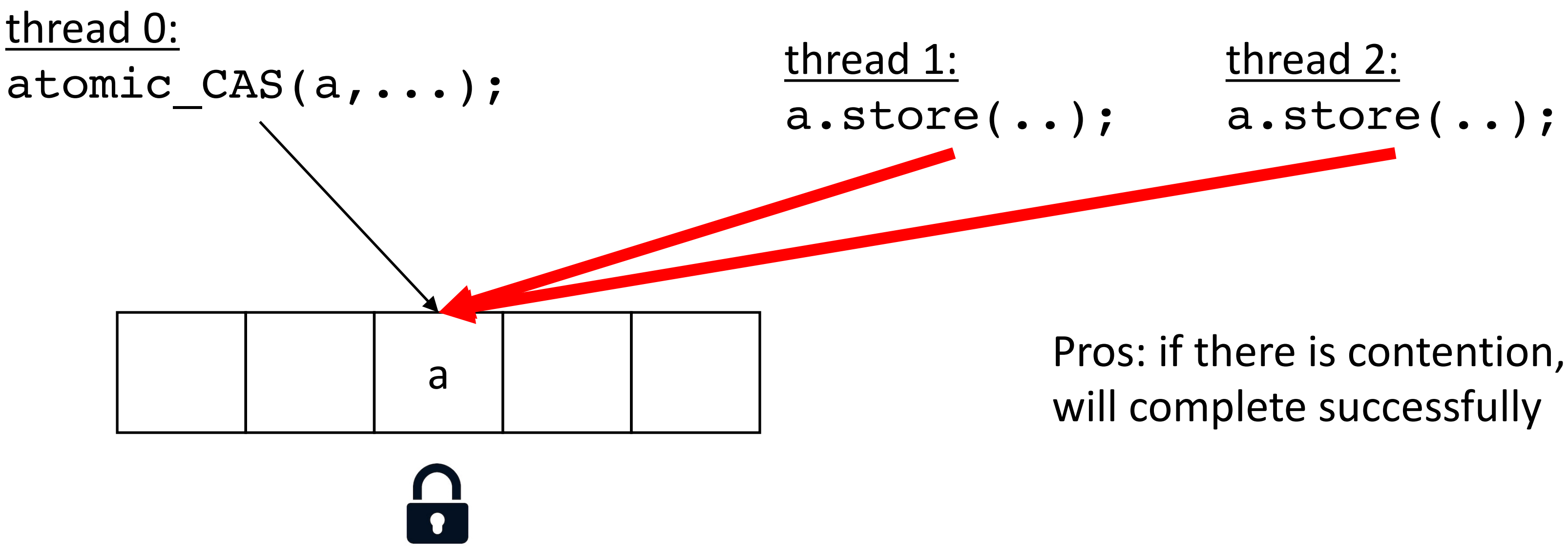
Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



Pessimistic Concurrency

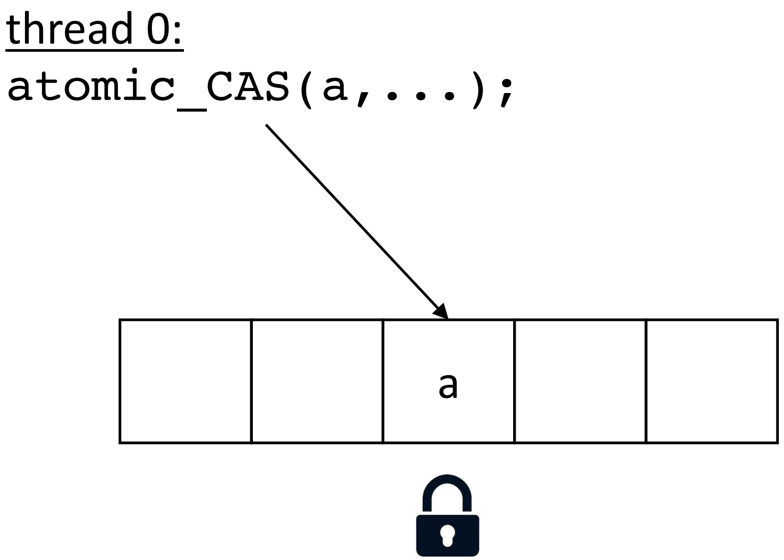
- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



Pros: if there is contention, the CAS will complete successfully

Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



Cons: if no other threads are contending, lock overhead is high

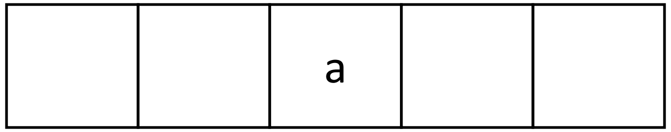
Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

For this example consider an atomic increment

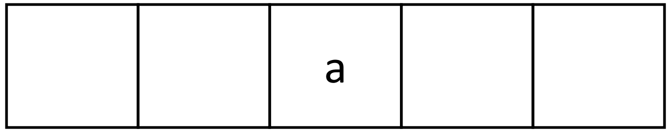


Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```



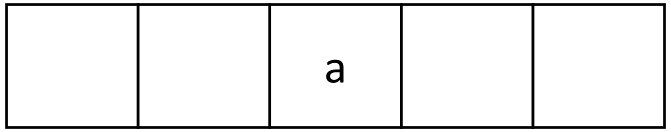
T0_exclusive = 1

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```



T0_exclusive = 1

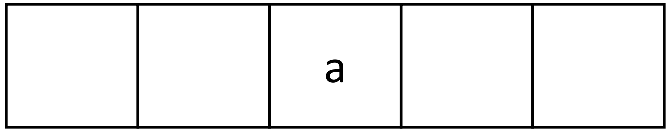
Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

before we store, we have to check if there was a conflict.



T0_exclusive = 1

Optimistic Concurrency

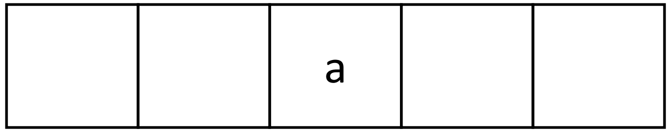
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

<u>thread 0:</u>

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

<u>thread 1:</u>

```
a.store(...)
```



Optimistic Concurrency

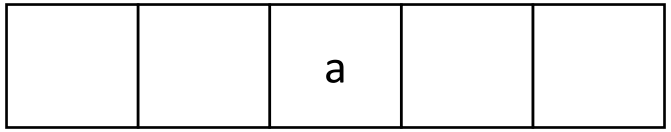
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```



T0_exclusive = 1

Optimistic Concurrency

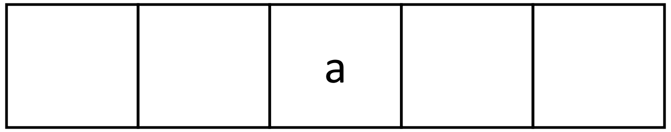
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```



T0_exclusive = 1

Optimistic Concurrency

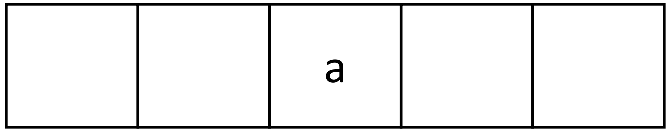
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```



T0_exclusive = 0

Optimistic Concurrency

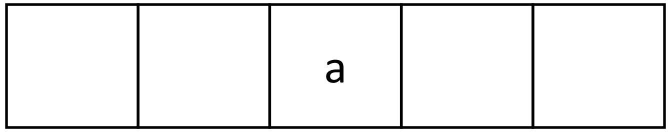
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```



T0_exclusive = 0

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

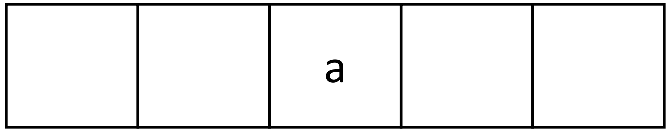
thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```

can't store because our exclusive bit was changed, i.e. there was a conflict!



T0_exclusive = 0

Optimistic Concurrency

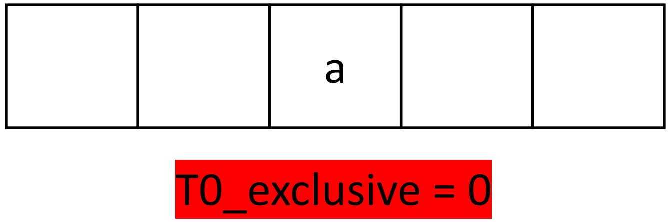
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```



can't store because our exclusive bit was changed, i.e. there was a conflict!

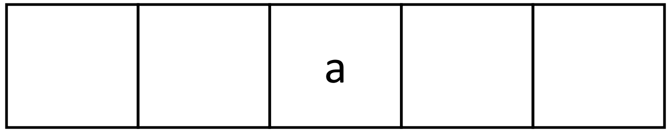
solution: loop until success:

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
do {
tmp = load_exclusive(a,...);
tmp += 1;
} while(!store_exclusive(a, tmp));
```



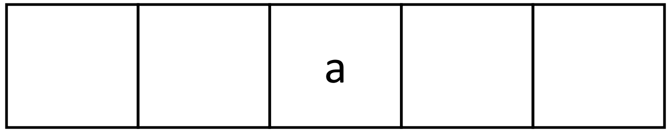
T0_exclusive = 0

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

<u>thread 0:</u>

```
do {
tmp = load_exclusive(a,...);
tmp += 1;
} while(!store_exclusive(a, tmp));
```



T0_exclusive = 1

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

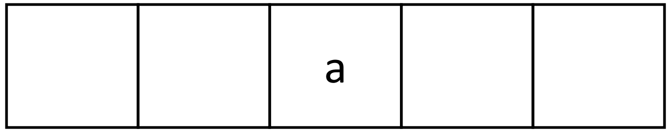
<u>thread 0:</u>

```
do {
tmp = load_exclusive(a,...);
tmp += 1;
} while(!store_exclusive(a, tmp));
```

Pros: very efficient when there is no conflicts!

Cons: conflicts are very expensive!

Spinning thread might starve (but not indefinitely) if other threads are constantly writing.



T0_exclusive = 1

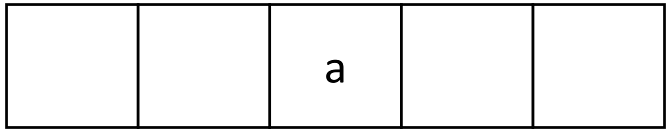
Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
do {
tmp = load_exclusive(a,...);
tmp += 1;
} while(!store_exclusive(a, tmp));
```

ARM implements all atomics this way!



T0_exclusive = 1

Godbolt example

- Show compiler examples

Schedule

- Using atomic templates for objects in C++
- How atomics are implemented in hardware
- **Lock-free concurrent set**

Thanks to Roberto Palmieri (Lehigh University) and material from the text book for some of the slide content/ideas.

Review our set

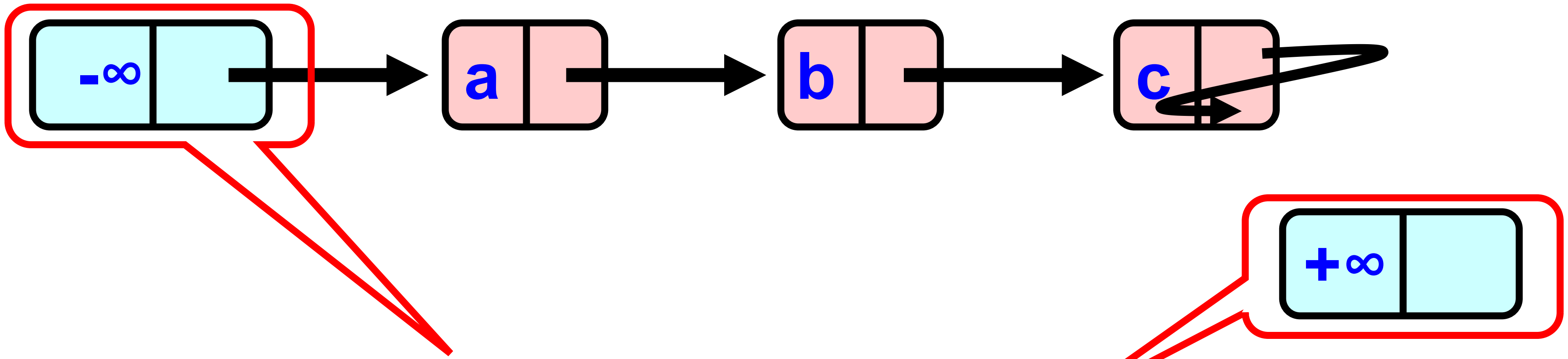
Set Interface

- 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

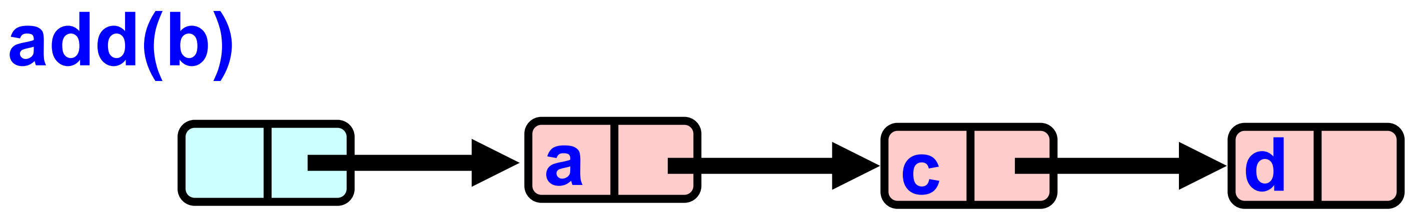
```
class Node {
 public:
   Value v;
   int key;
   Node *next;
}
```

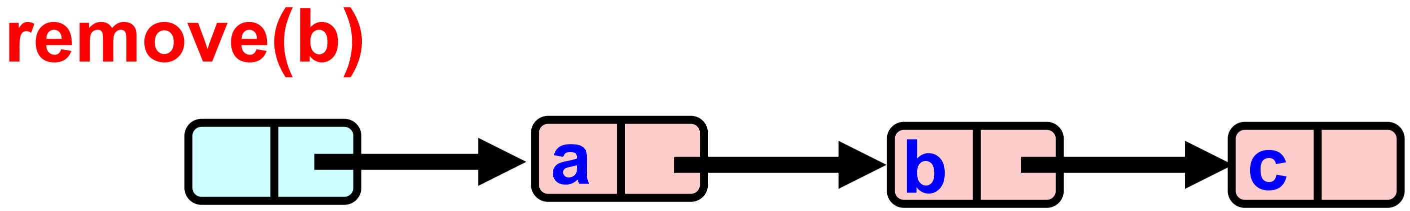
The List-Based Set



Sorted with Sentinel nodes
(min & max possible keys)

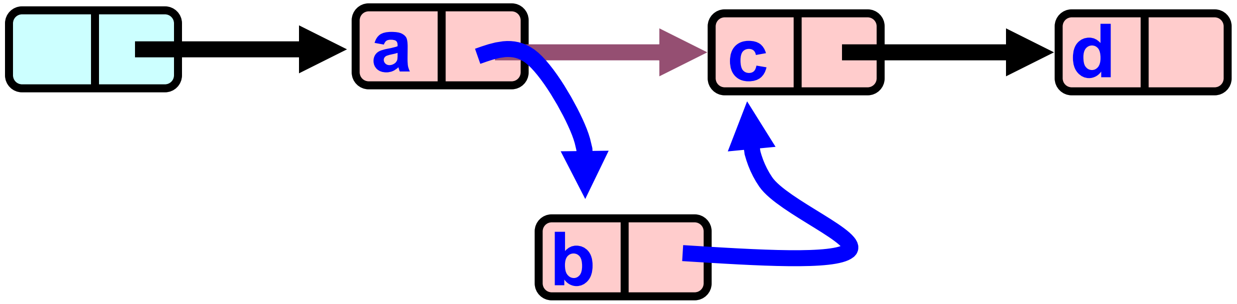
Sequential List Based Set



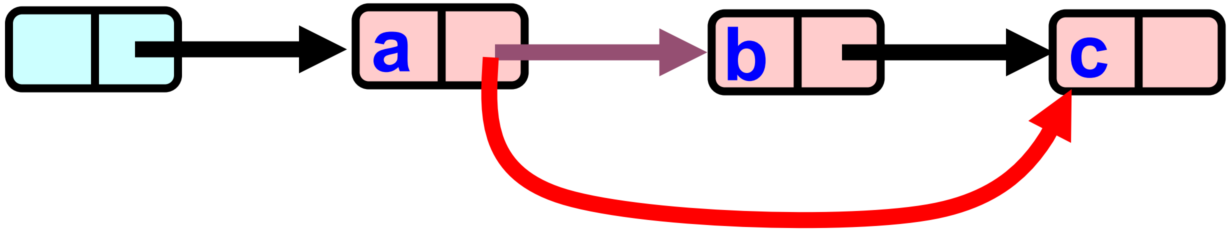


Sequential List Based Set

add(b)



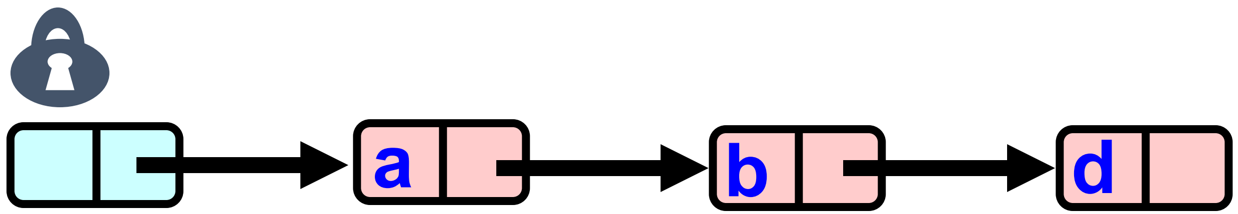
remove(b)



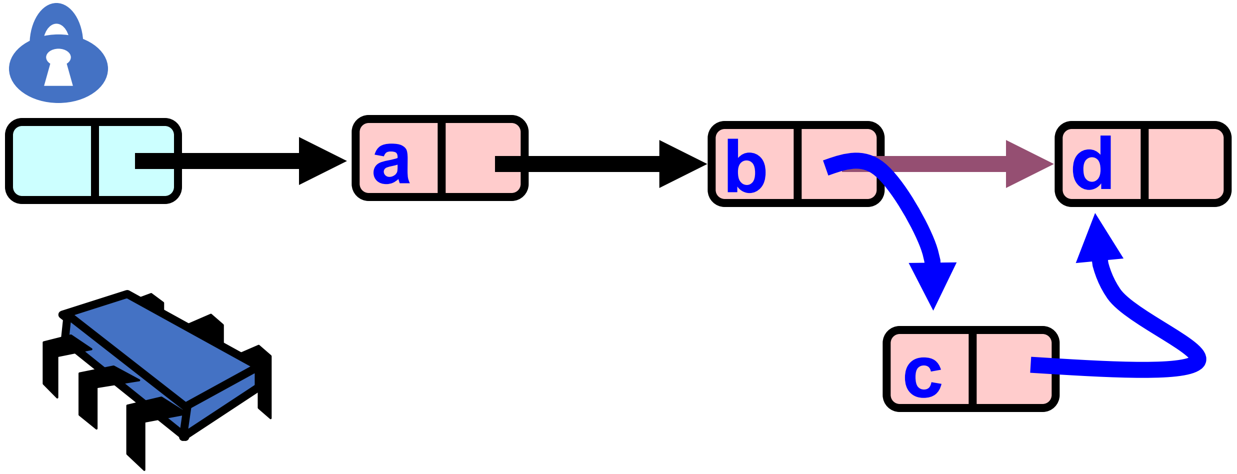
Schedule

- 3 approaches so far: each one slightly more complex

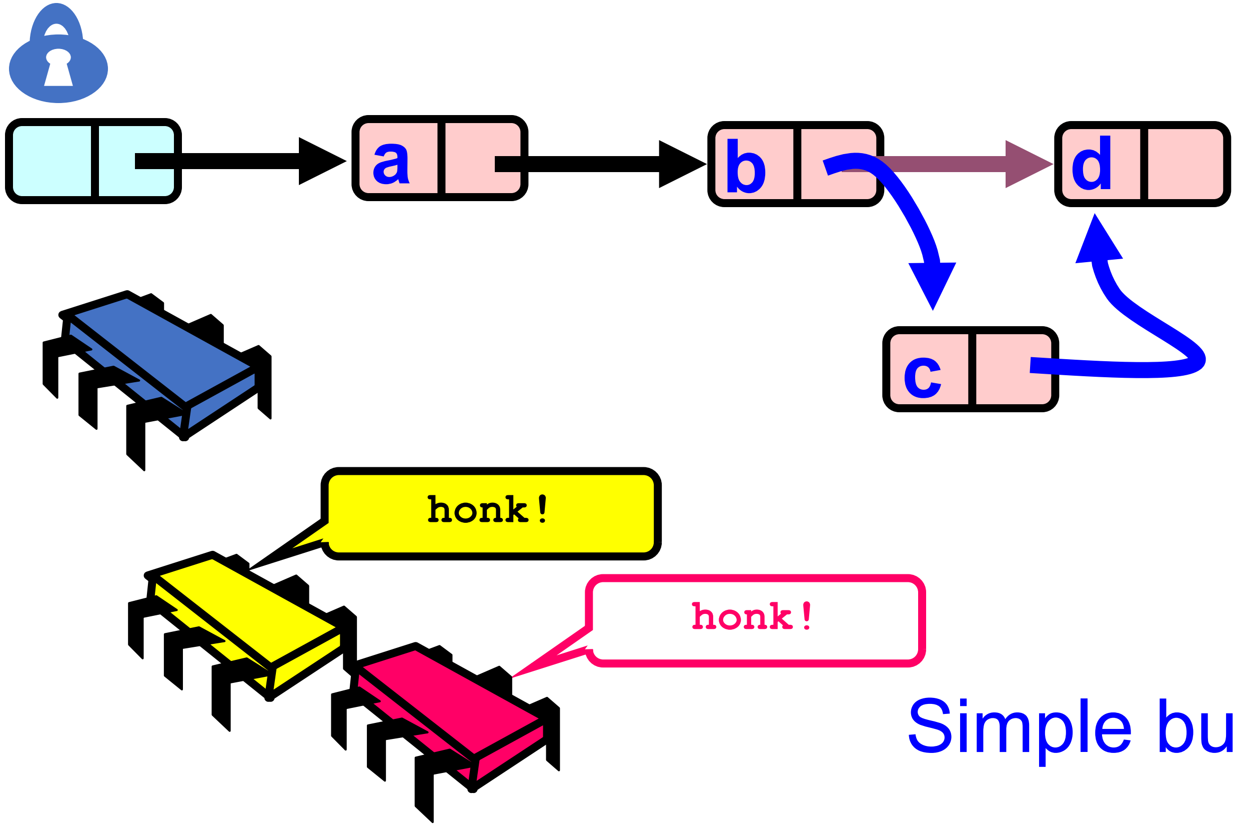
Coarse-Grained Locking



Coarse-Grained Locking



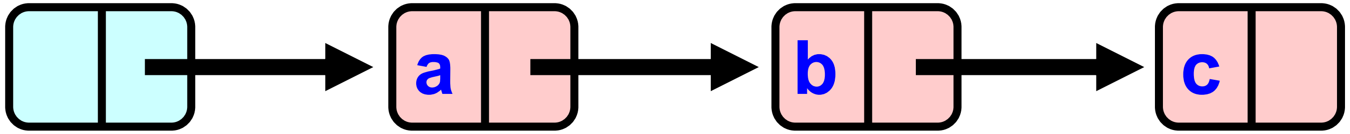
Coarse-Grained Locking

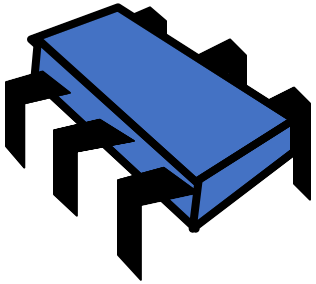


Simple but inefficient!

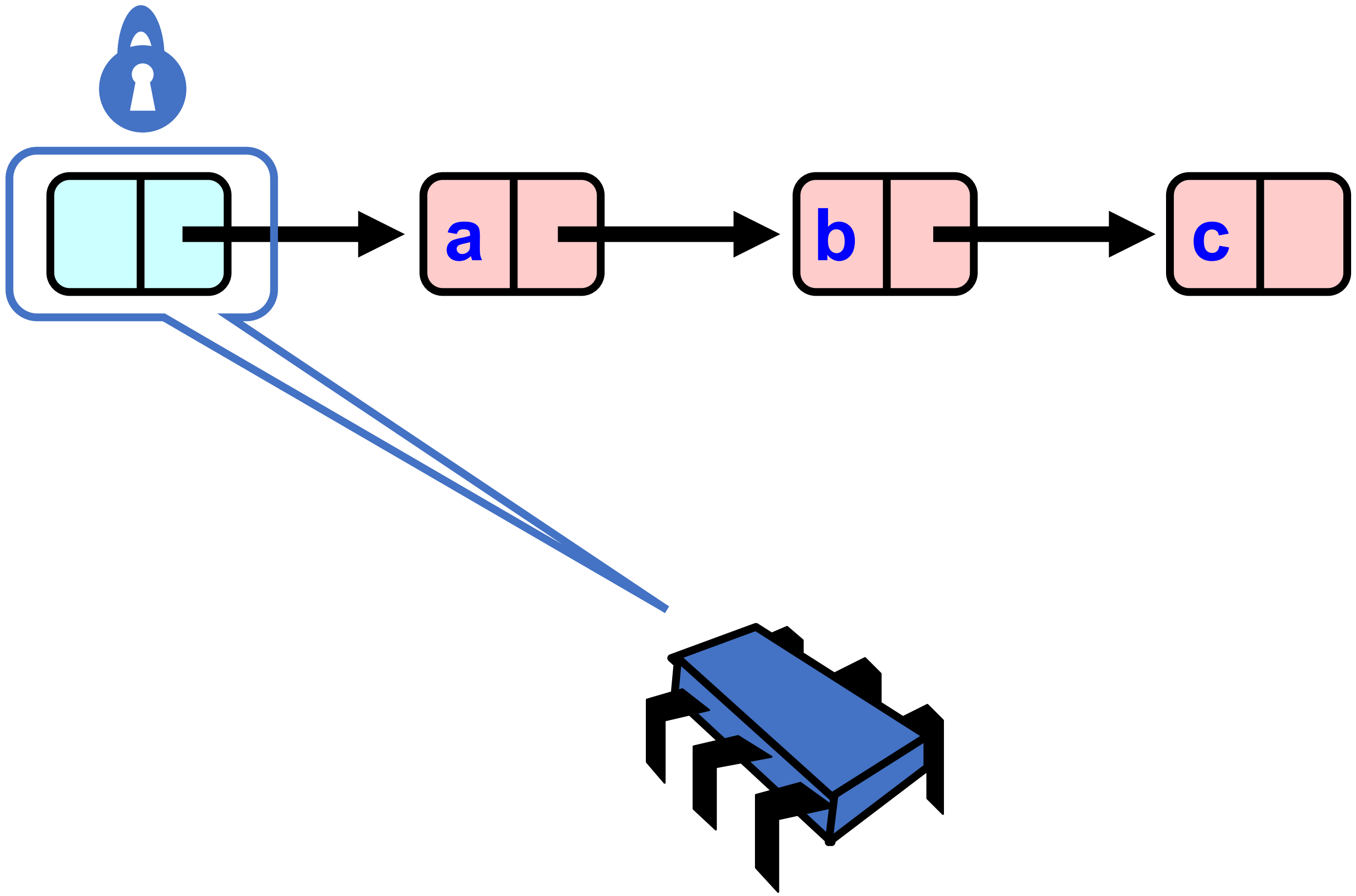
Lock coupling

Hand-over-Hand locking

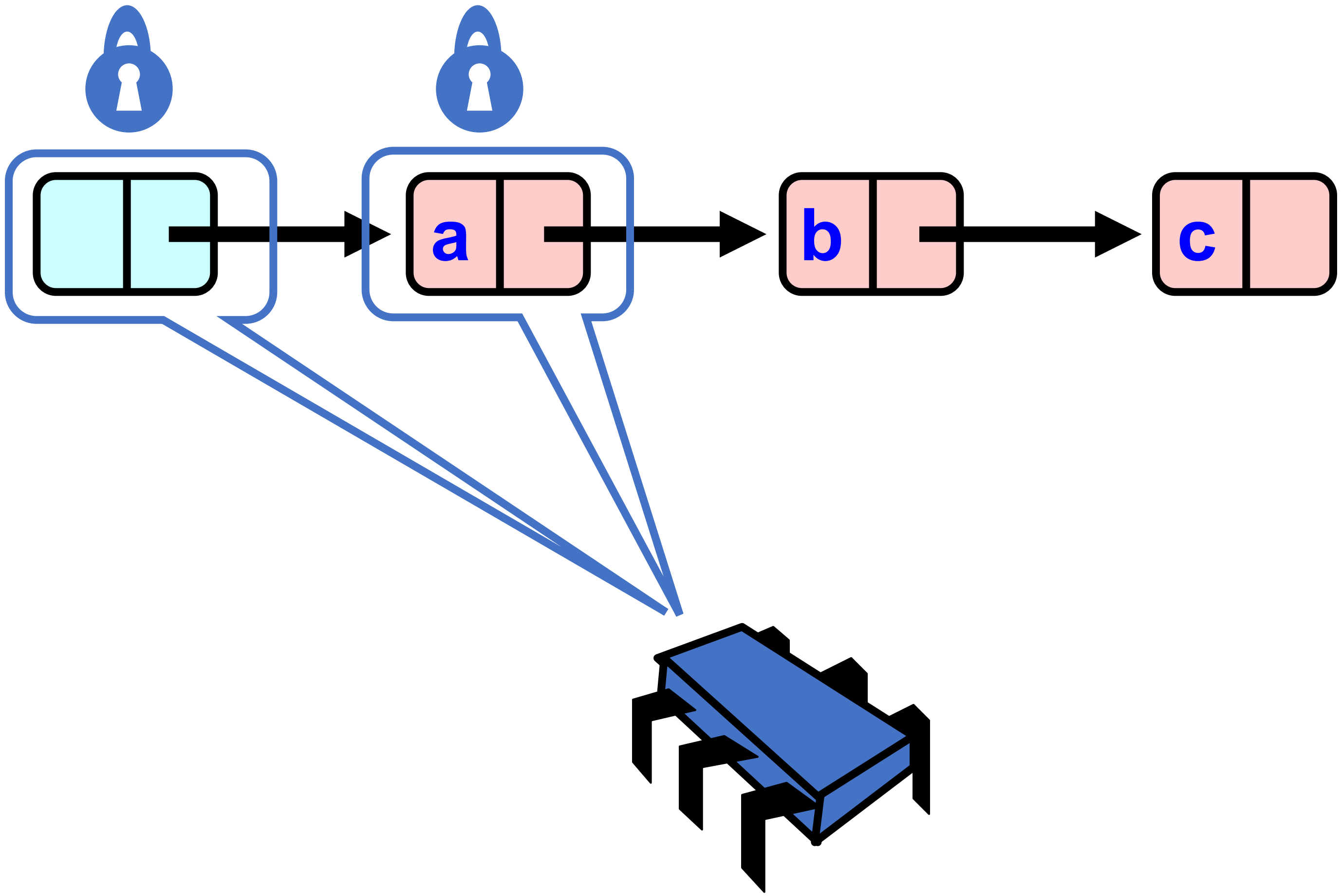




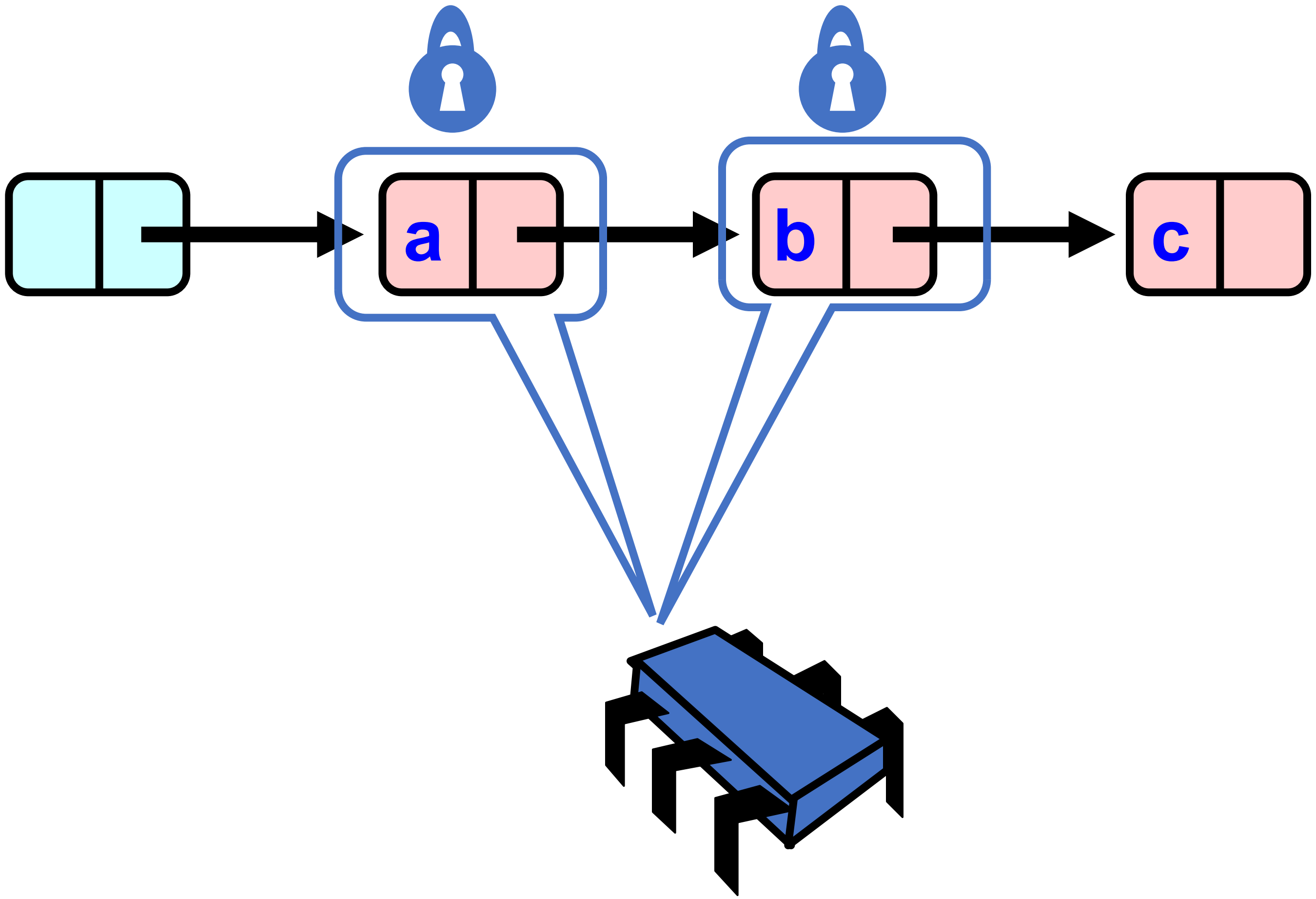
Hand-over-Hand locking



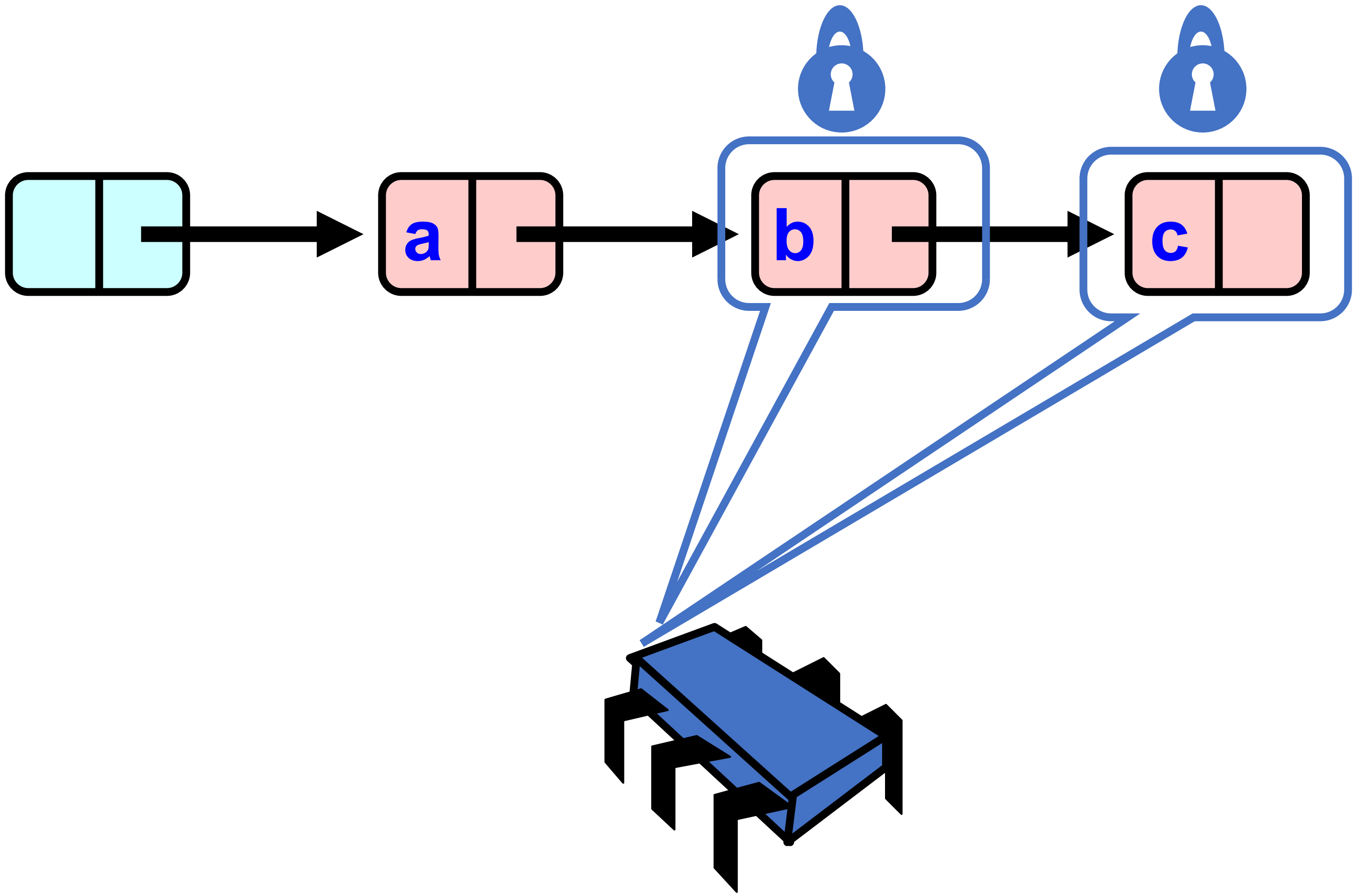
Hand-over-Hand locking



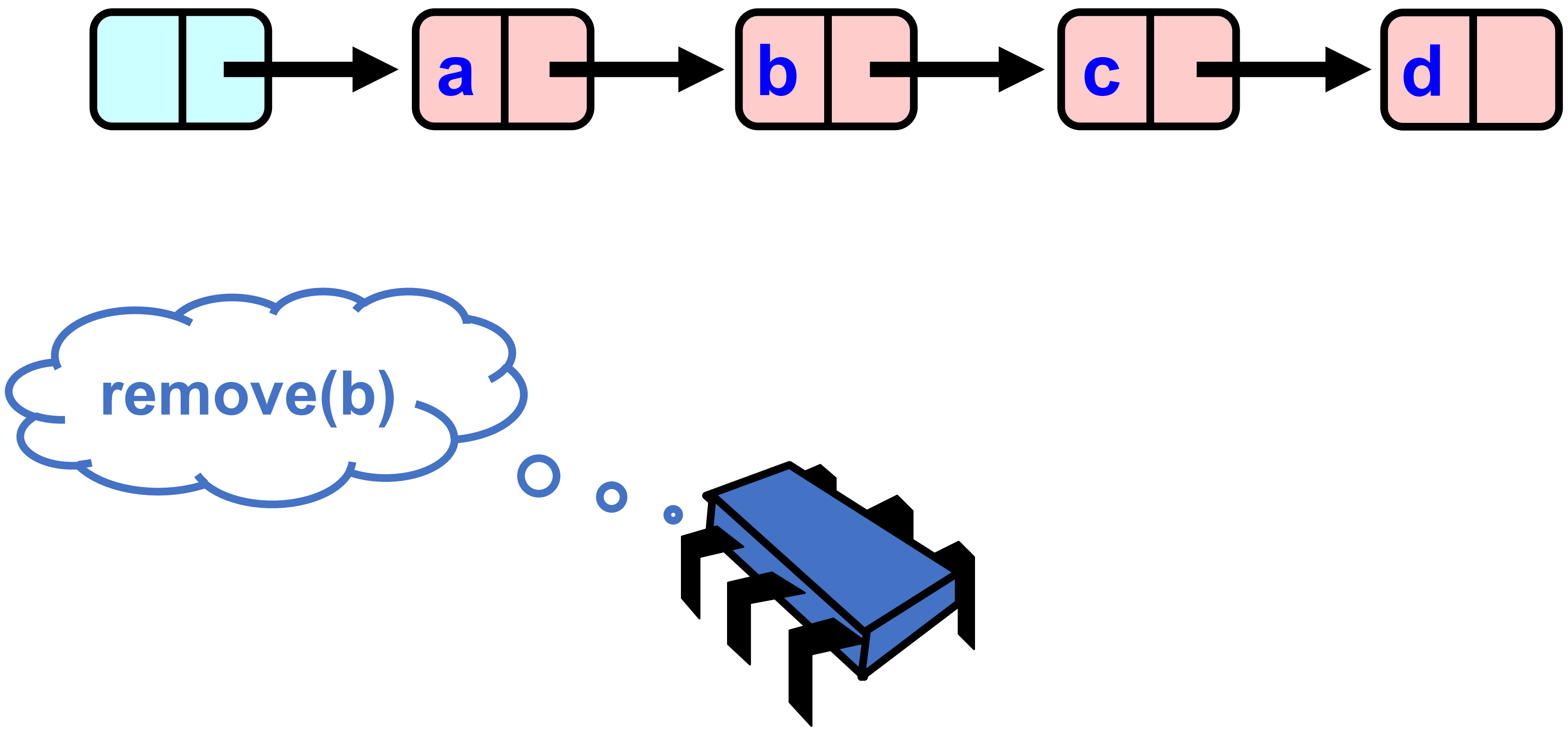
Hand-over-Hand locking



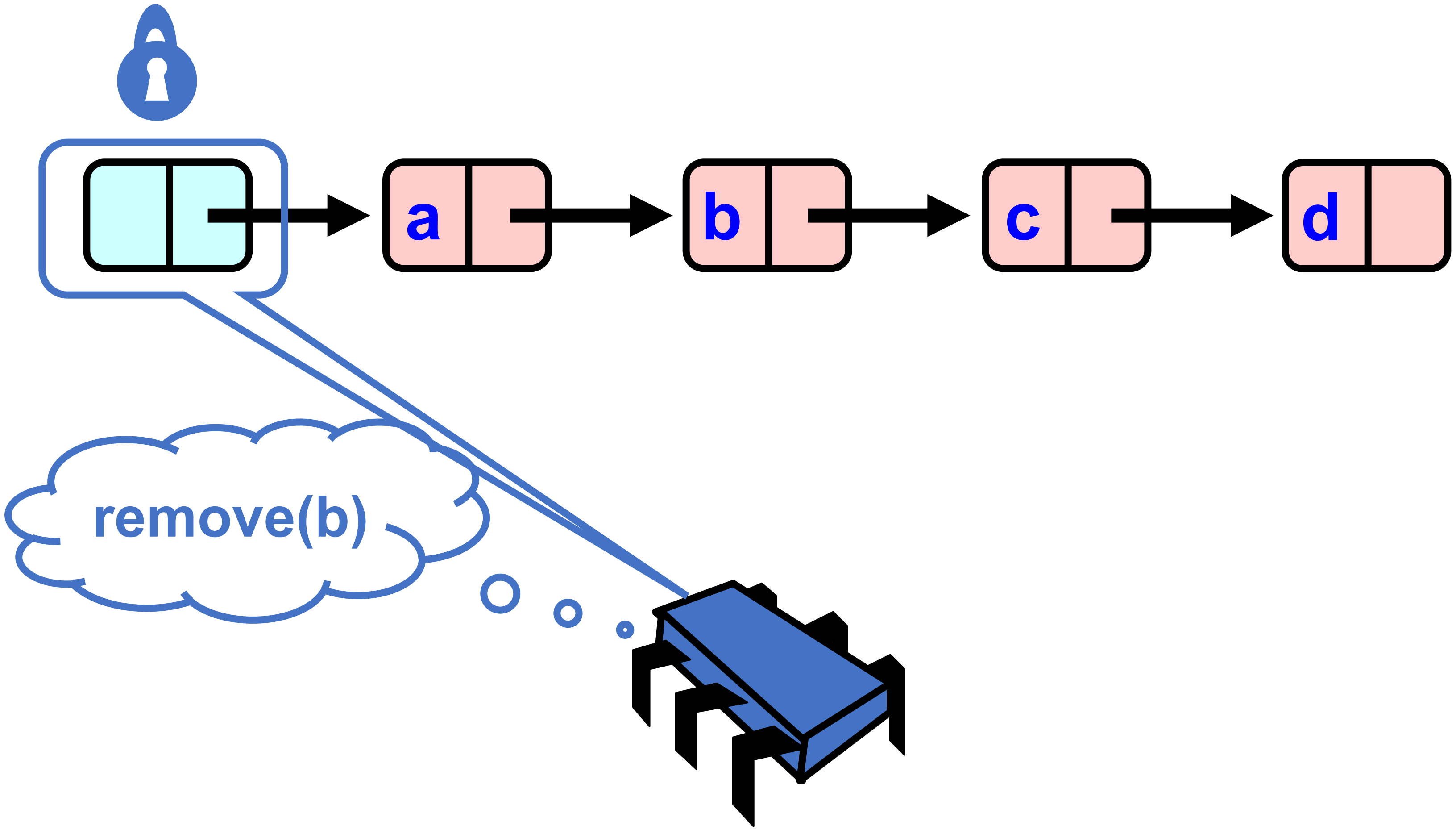
Hand-over-Hand locking



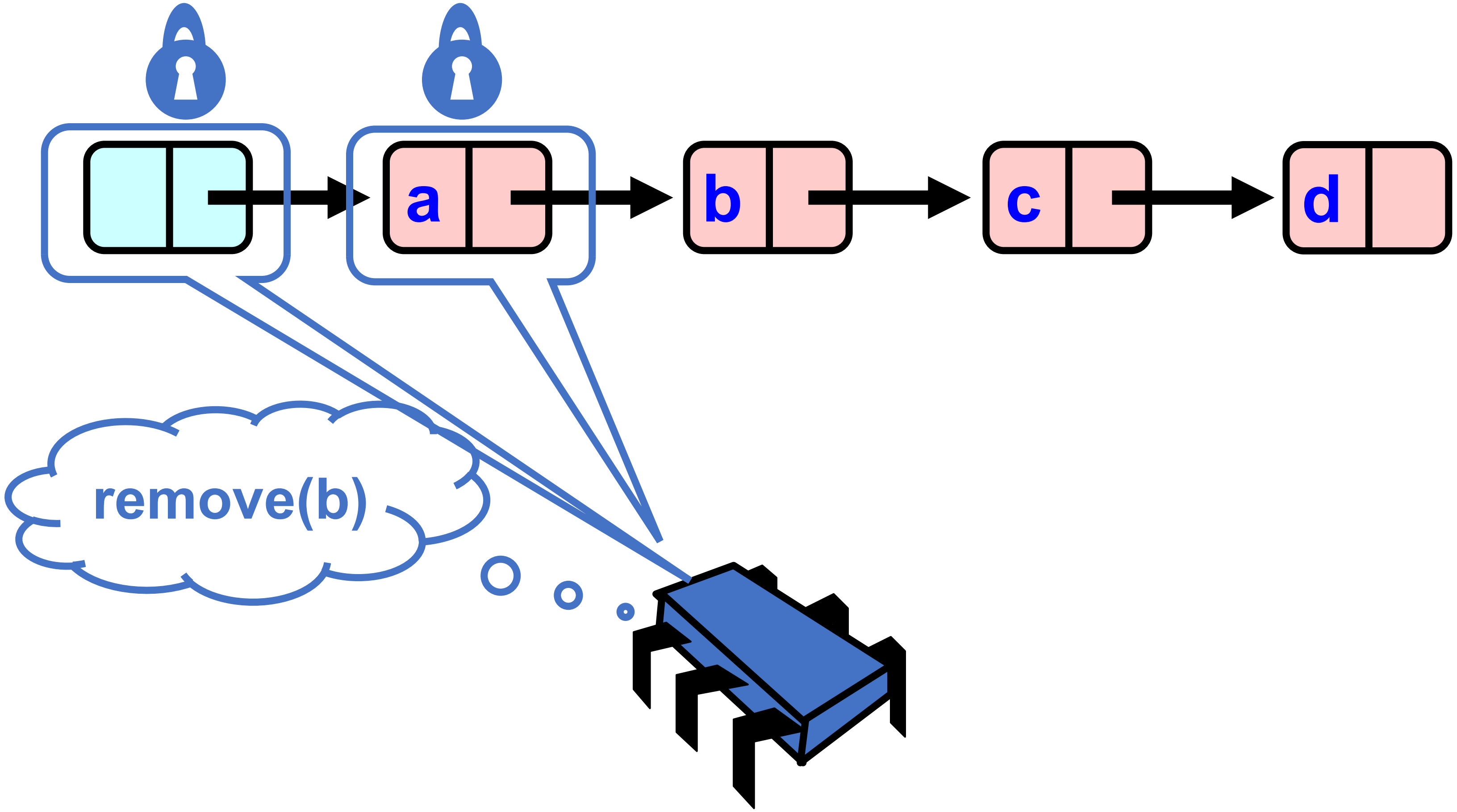
Removing a Node



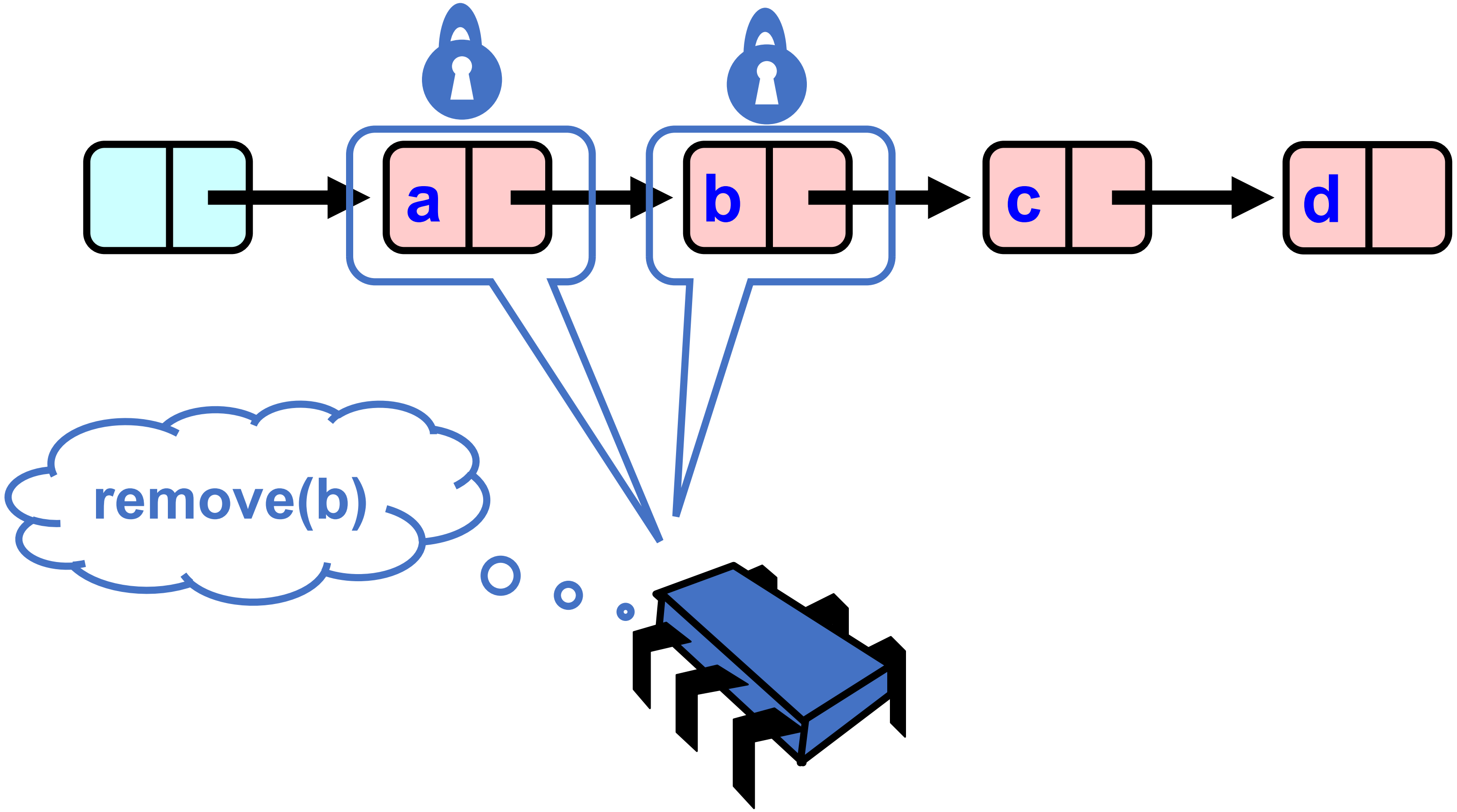
Removing a Node



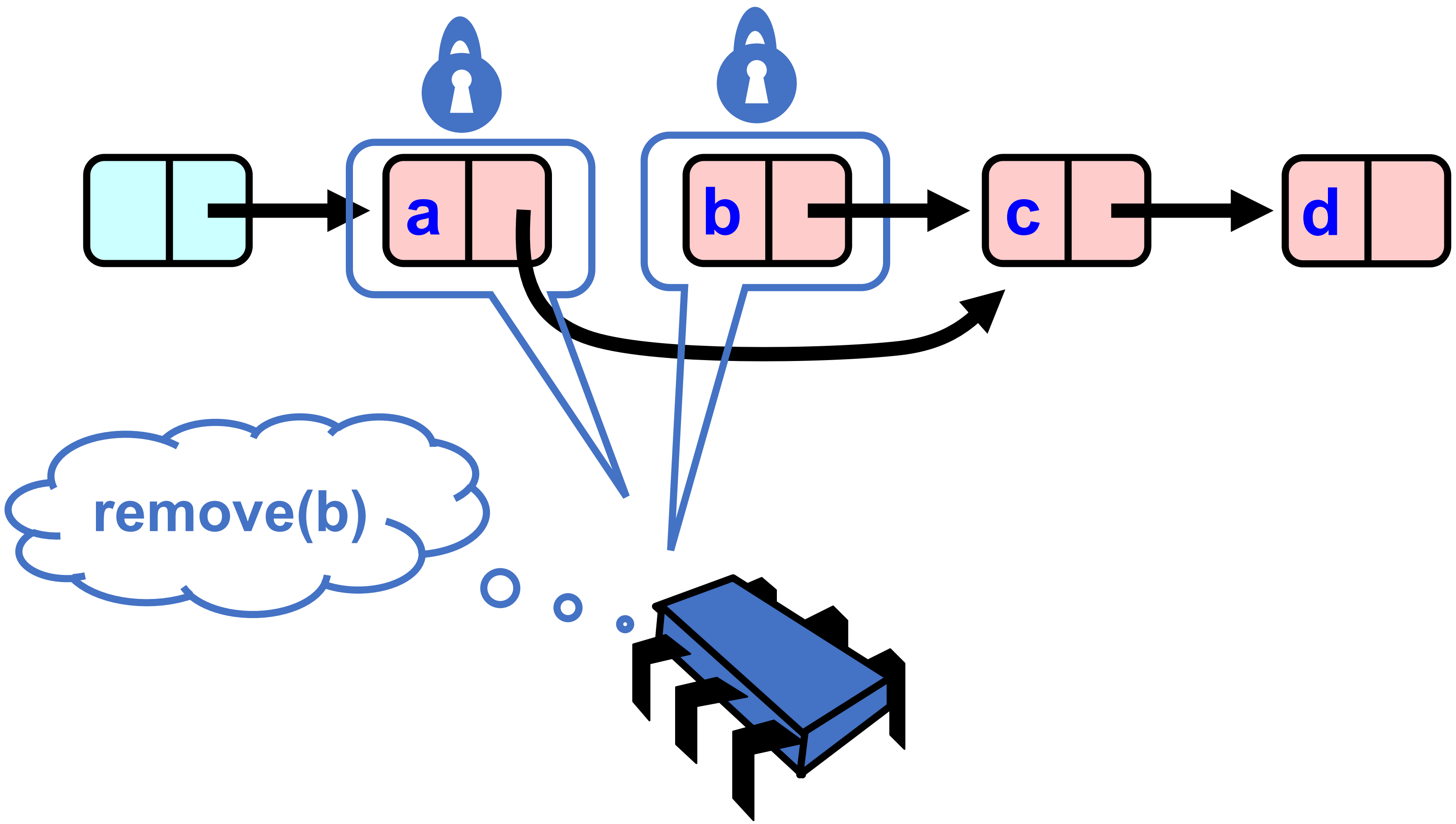
Removing a Node



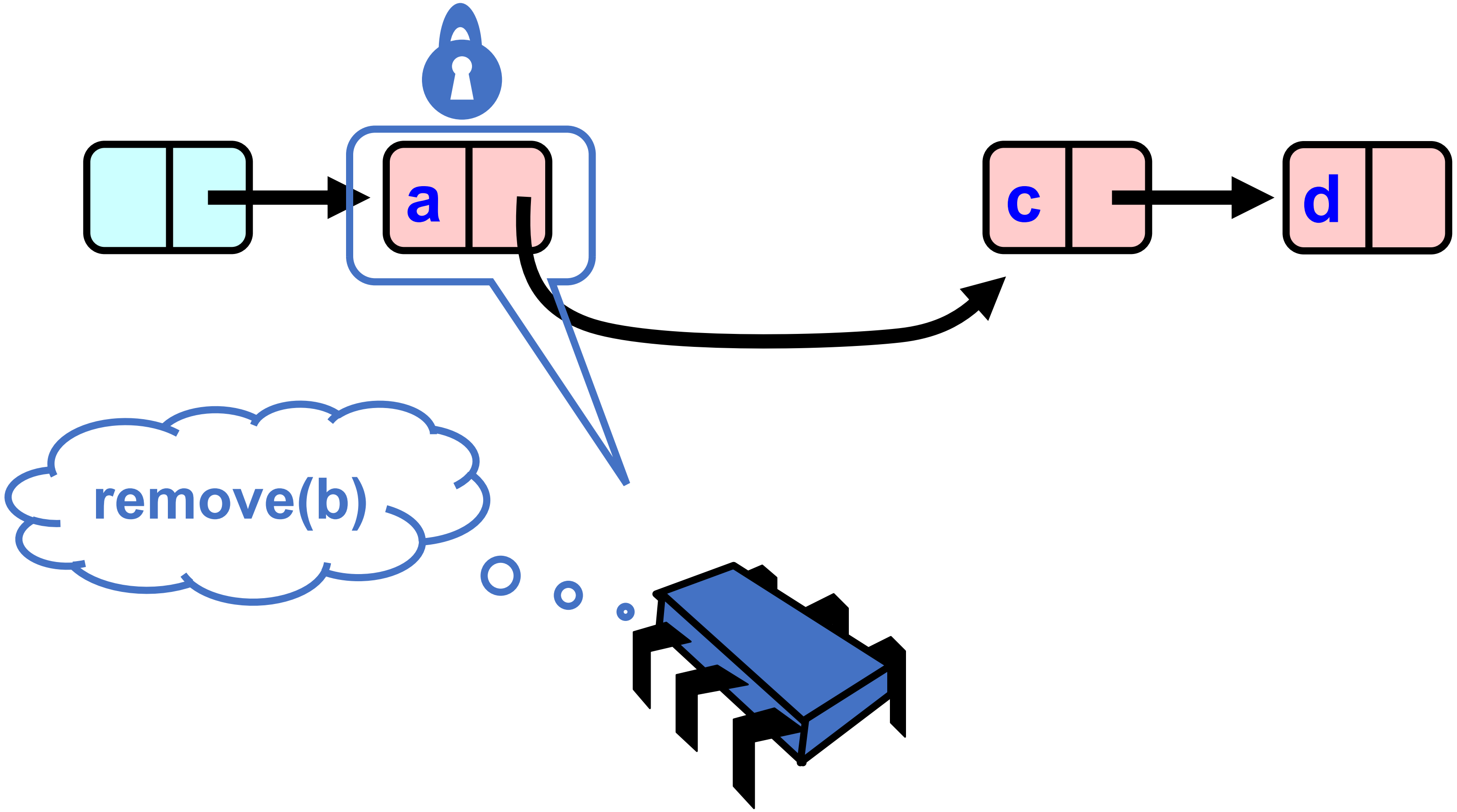
Removing a Node



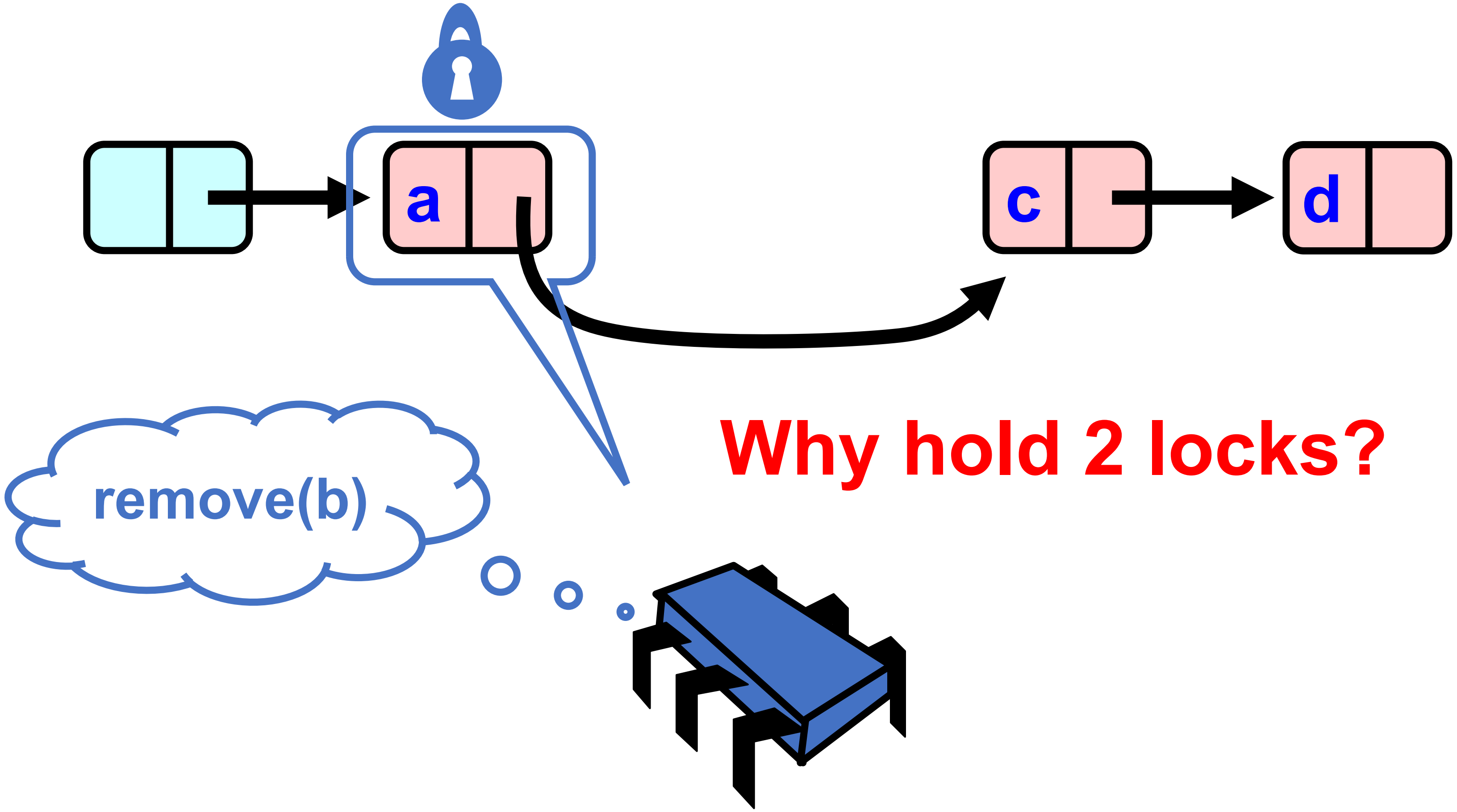
Removing a Node



Removing a Node

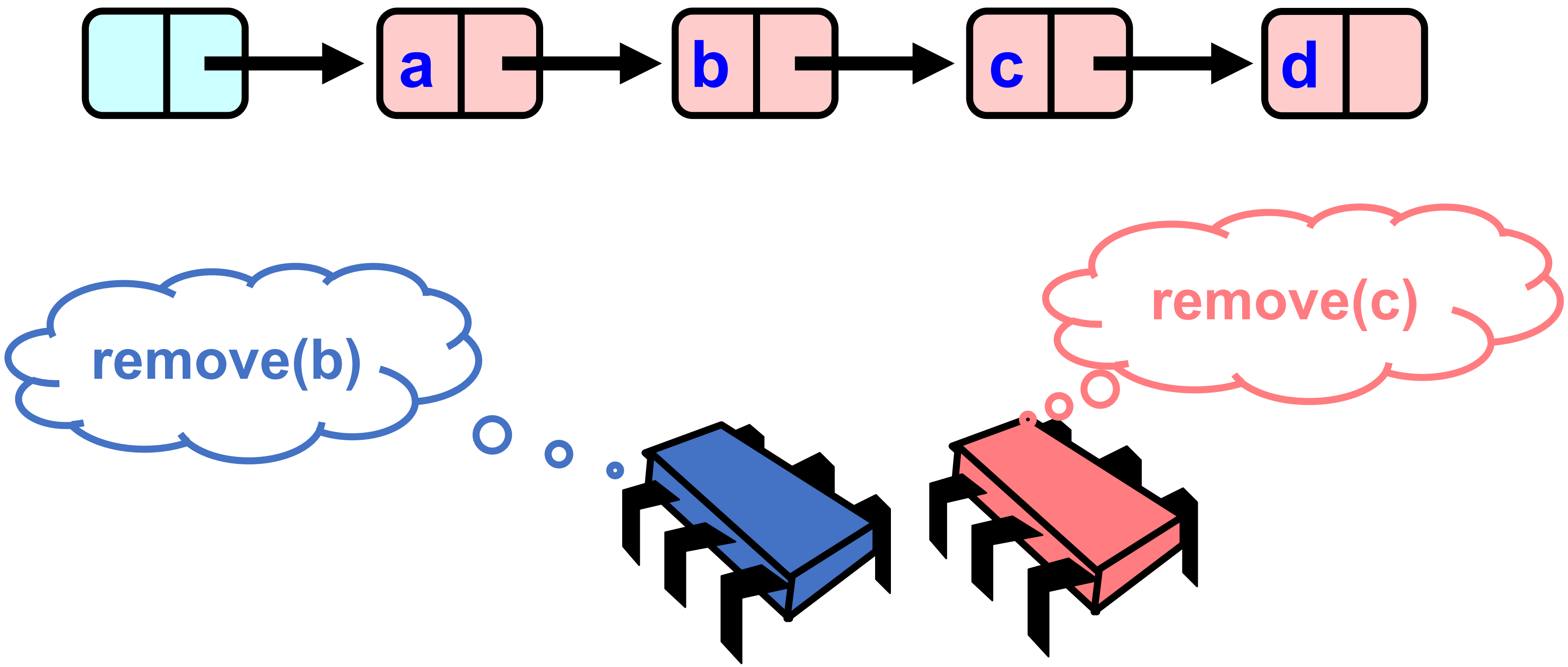


Removing a Node

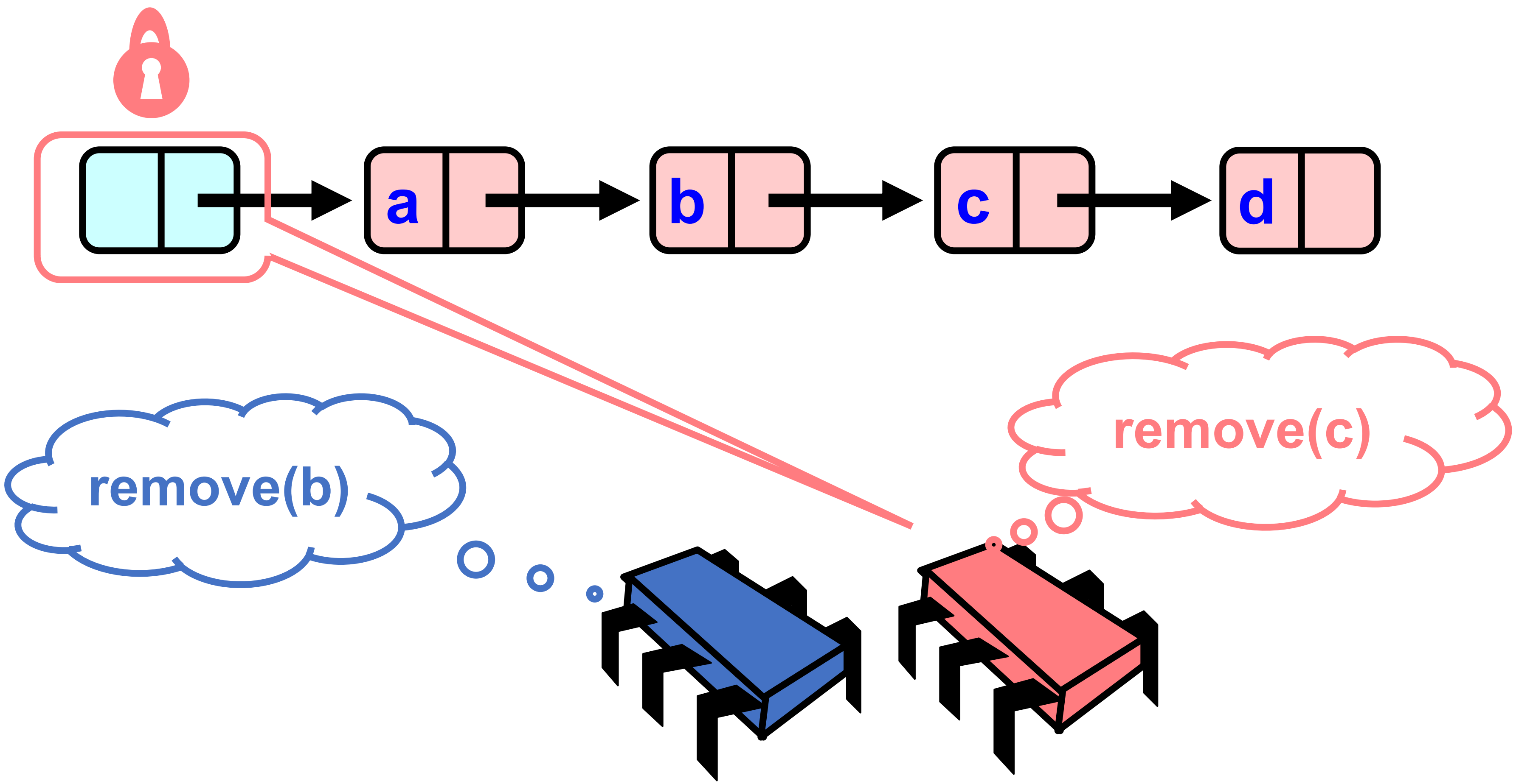


Why hold 2 locks?

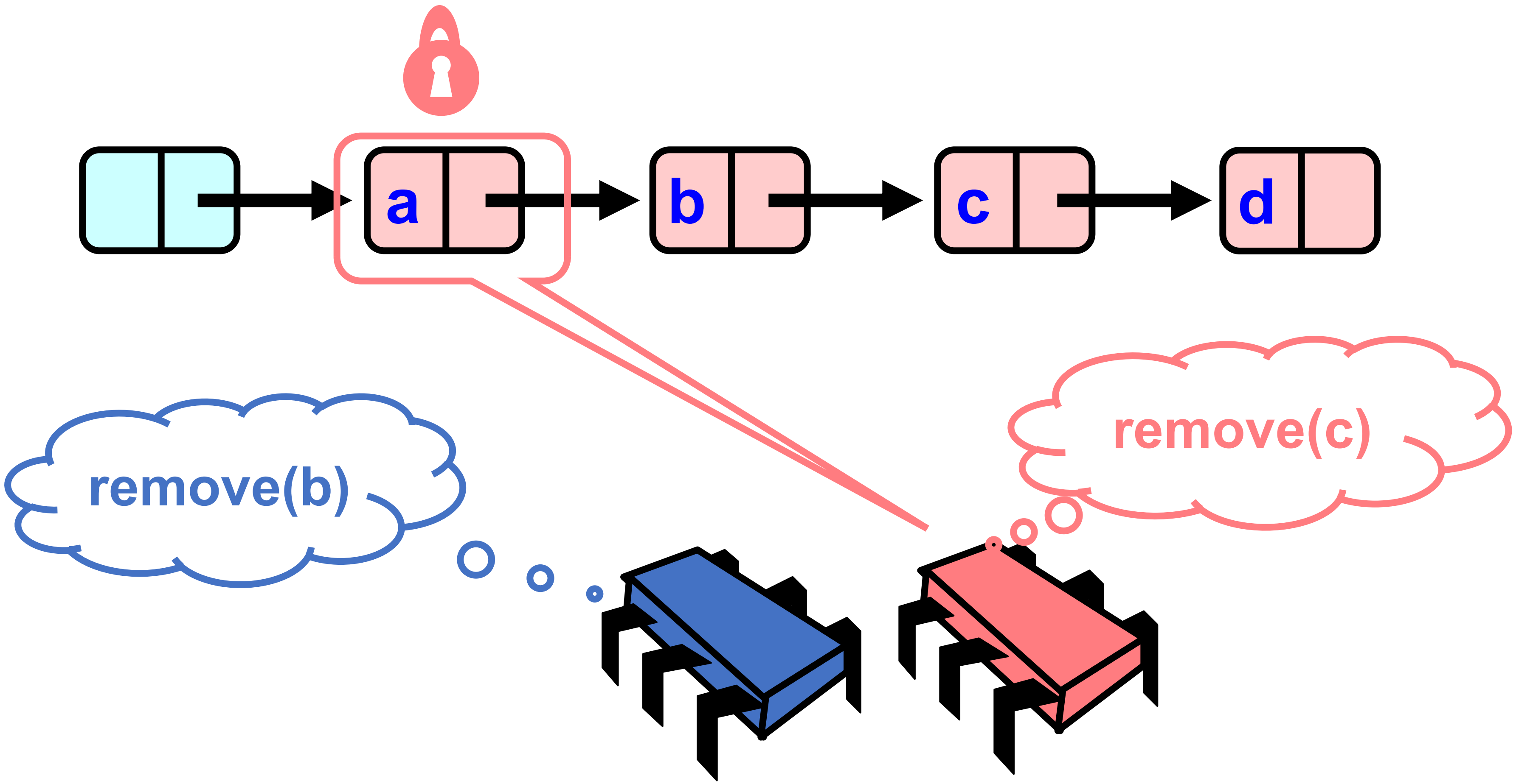
Concurrent Removes



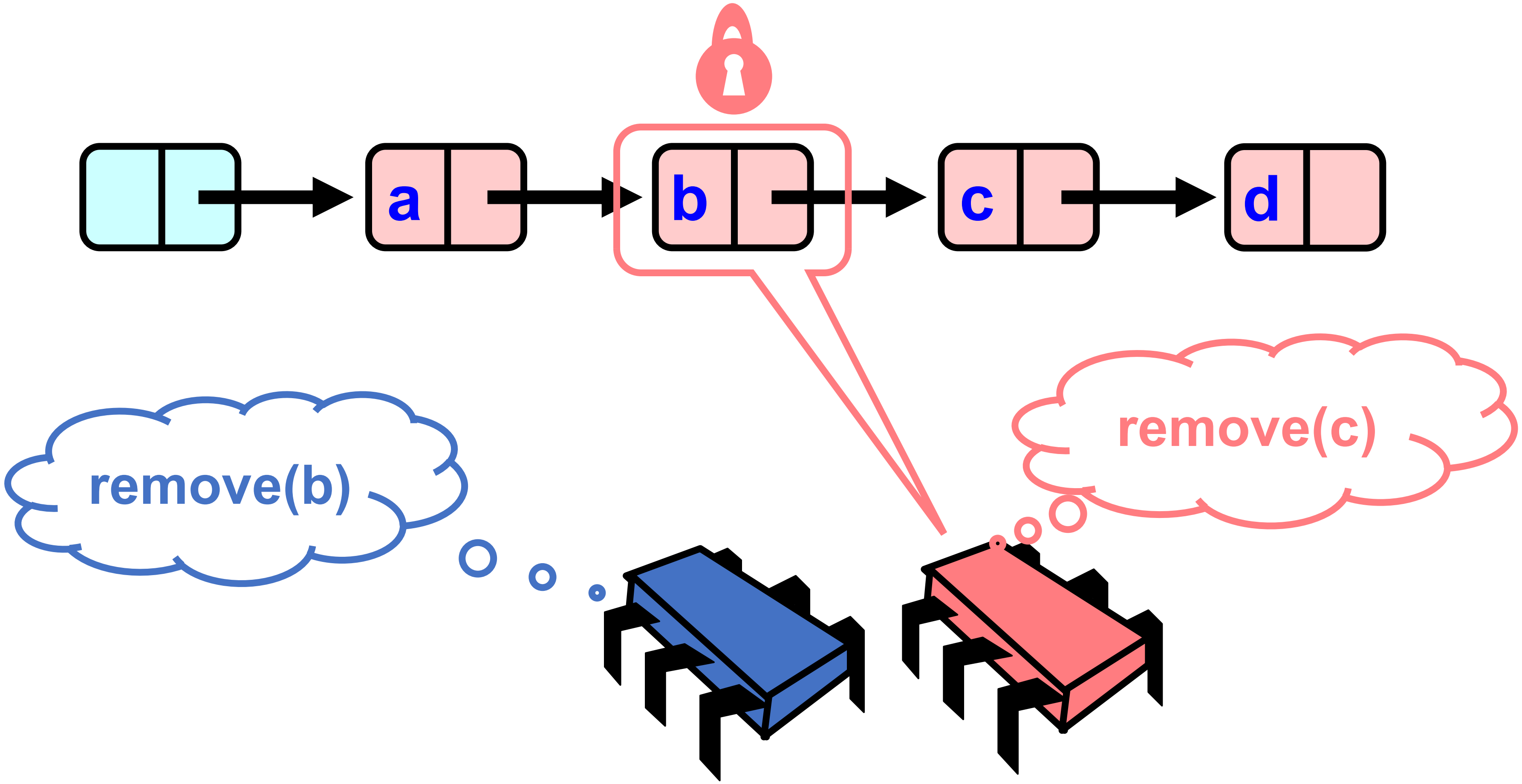
Concurrent Removes



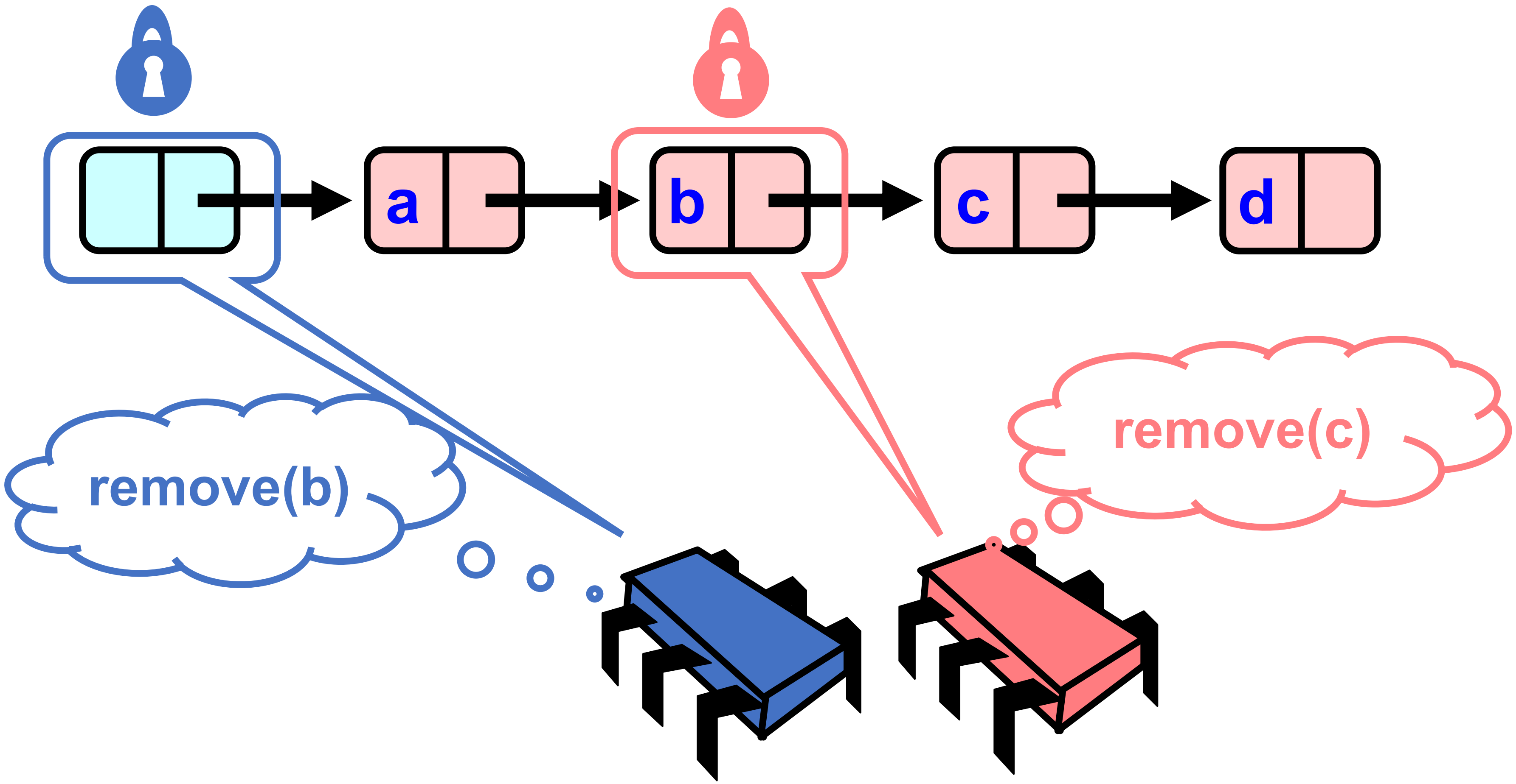
Concurrent Removes



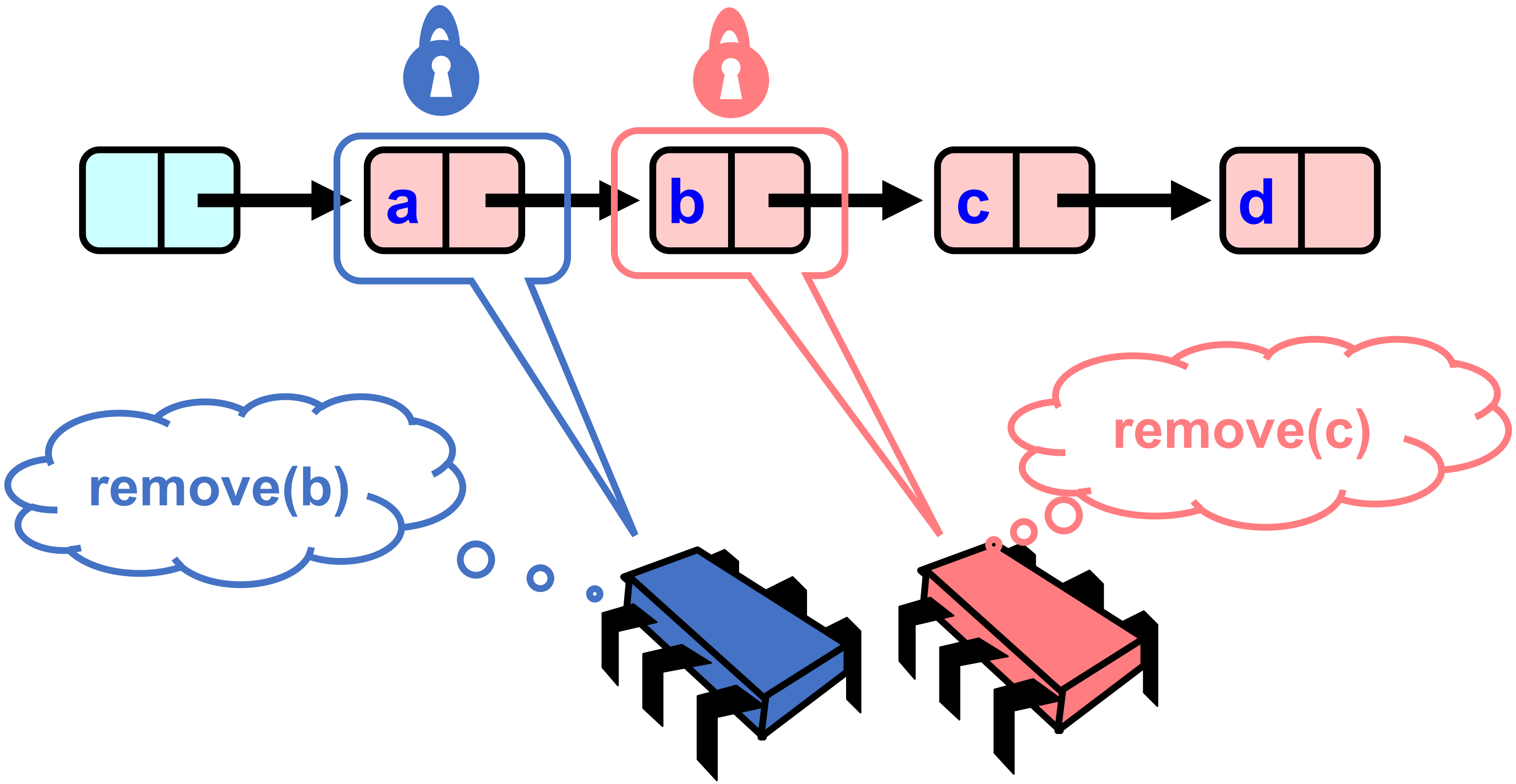
Concurrent Removes



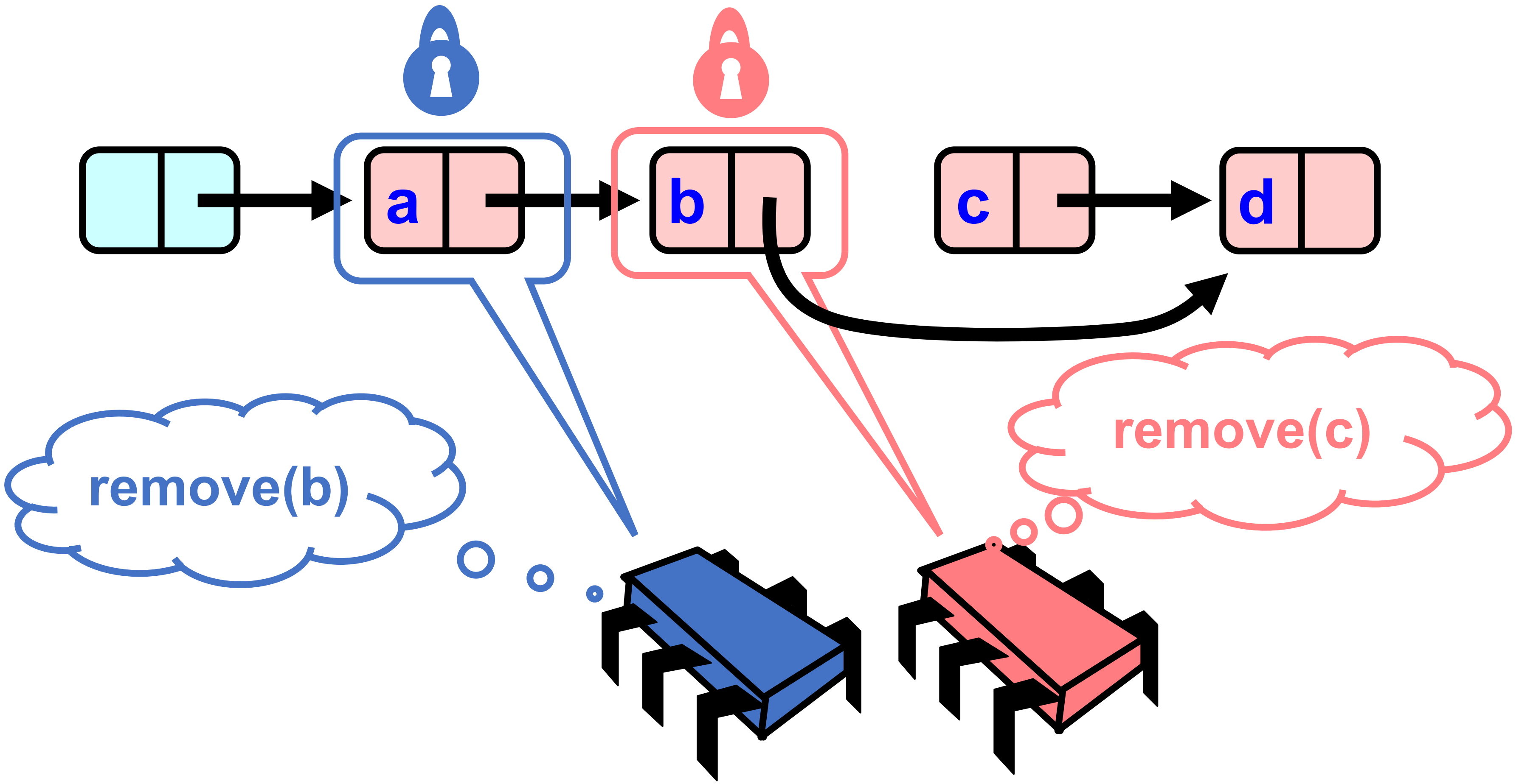
Concurrent Removes



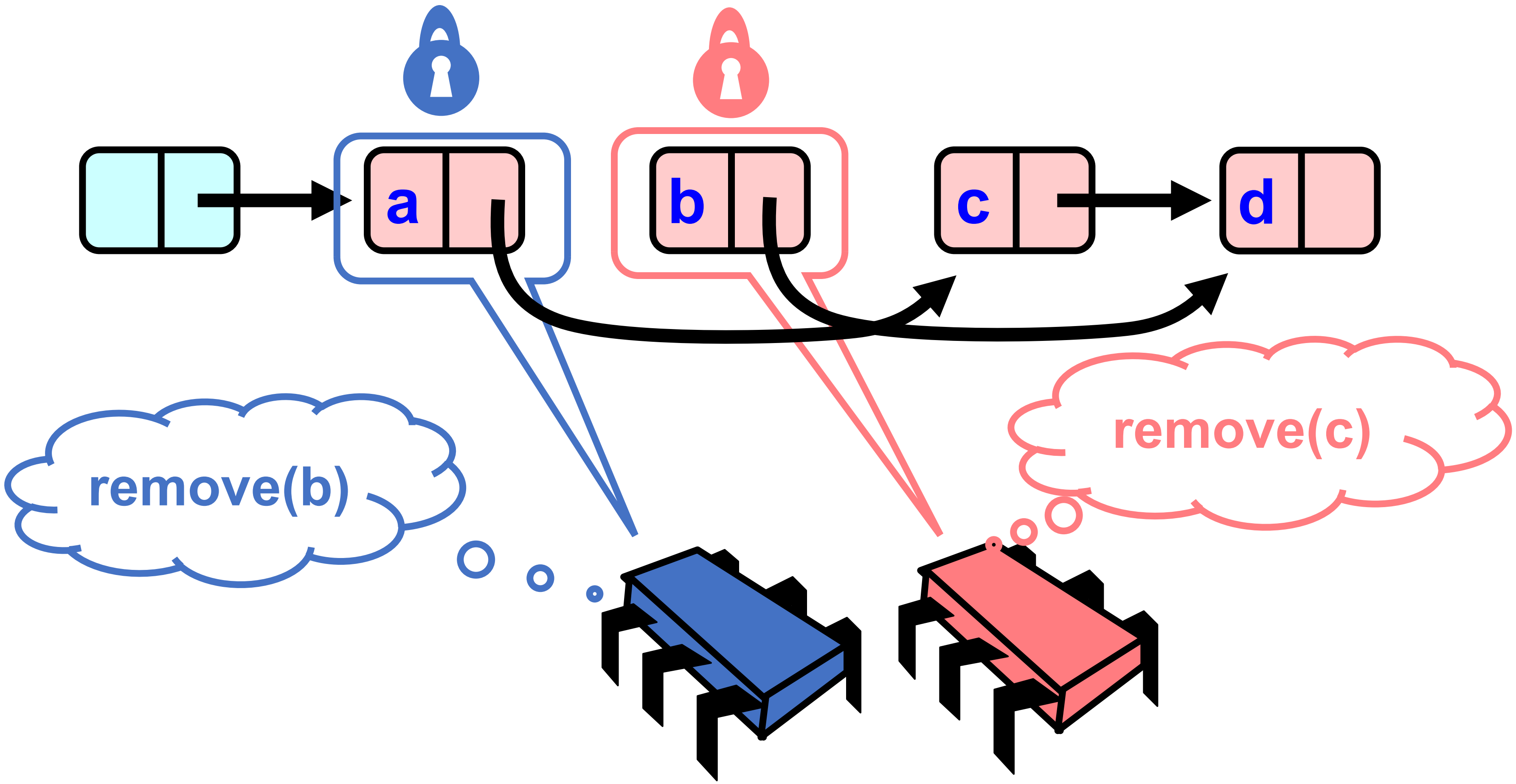
Concurrent Removes



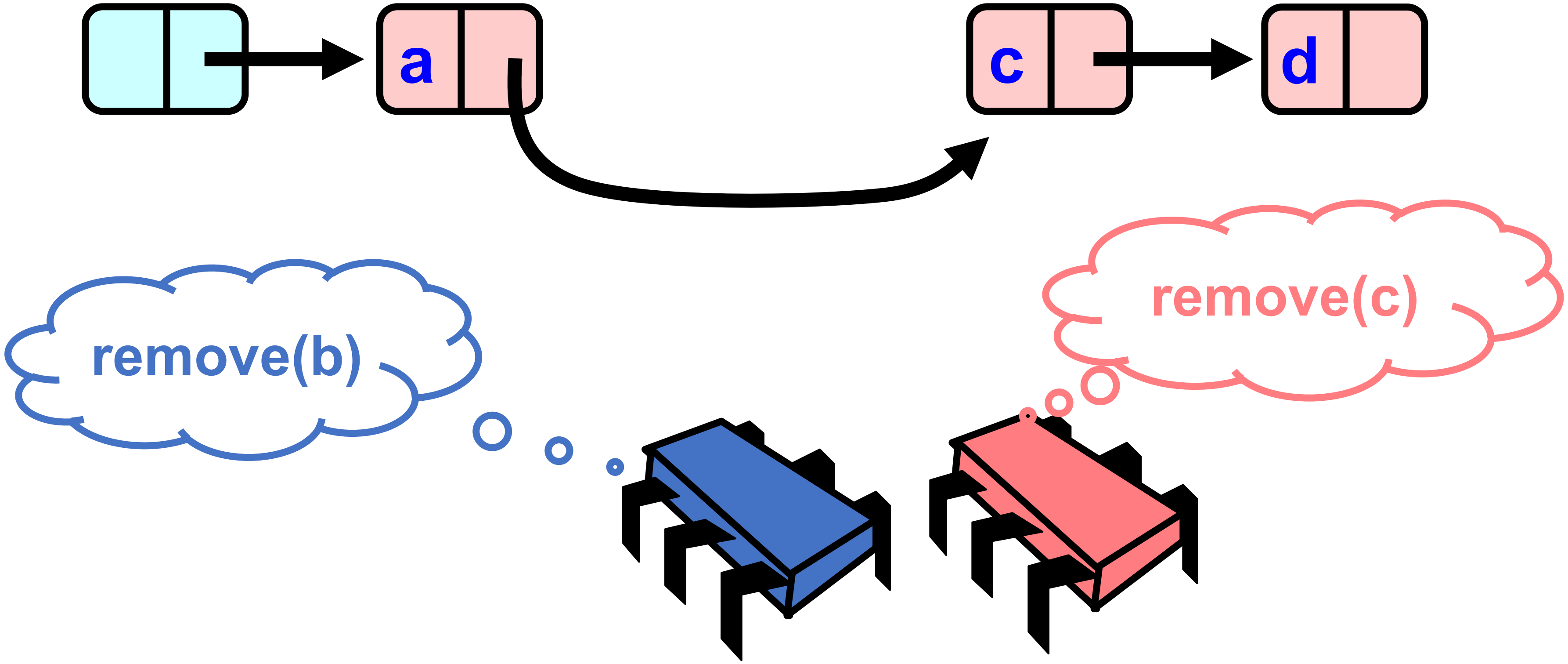
Concurrent Removes



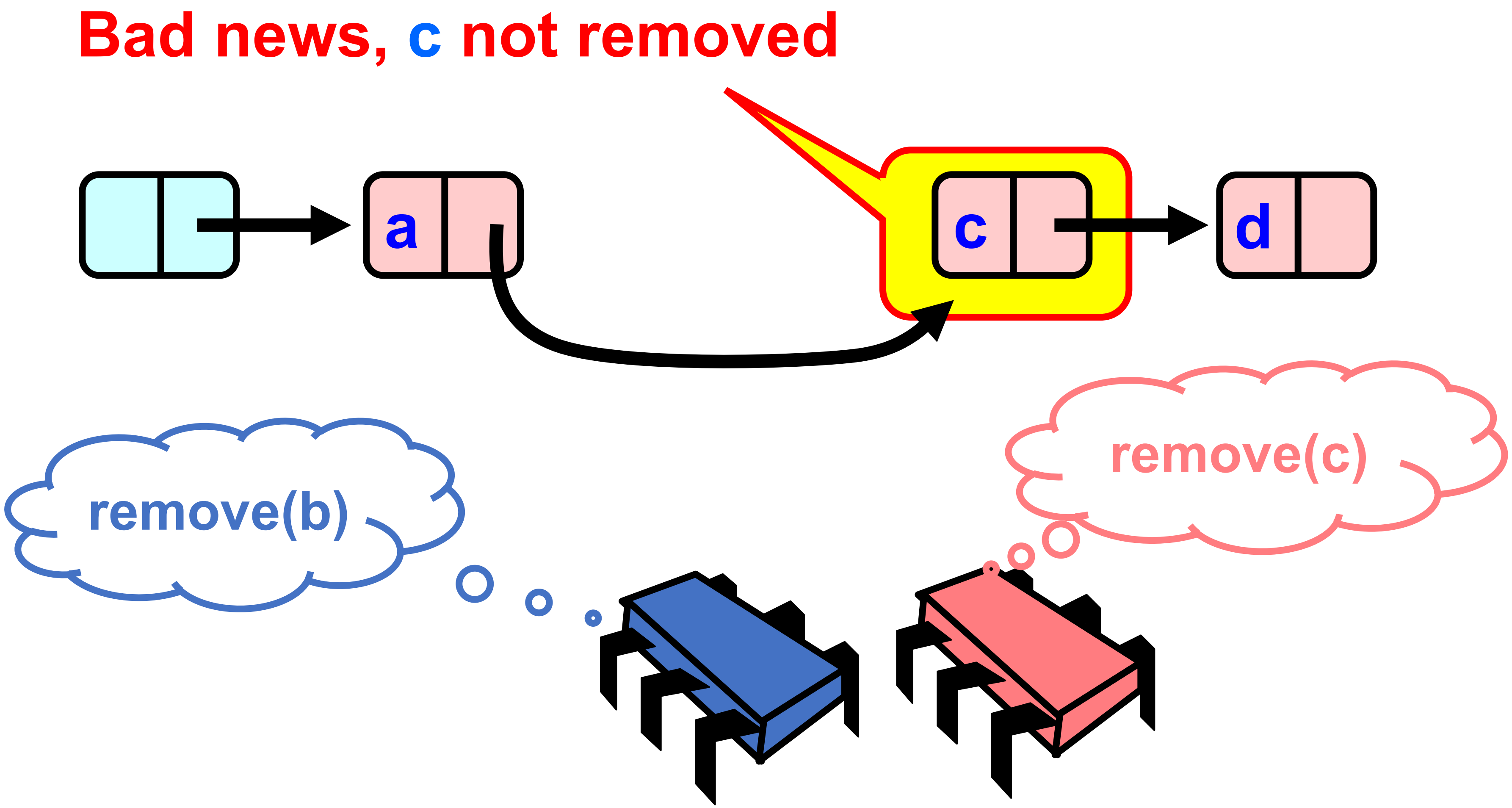
Concurrent Removes



Uh, Oh

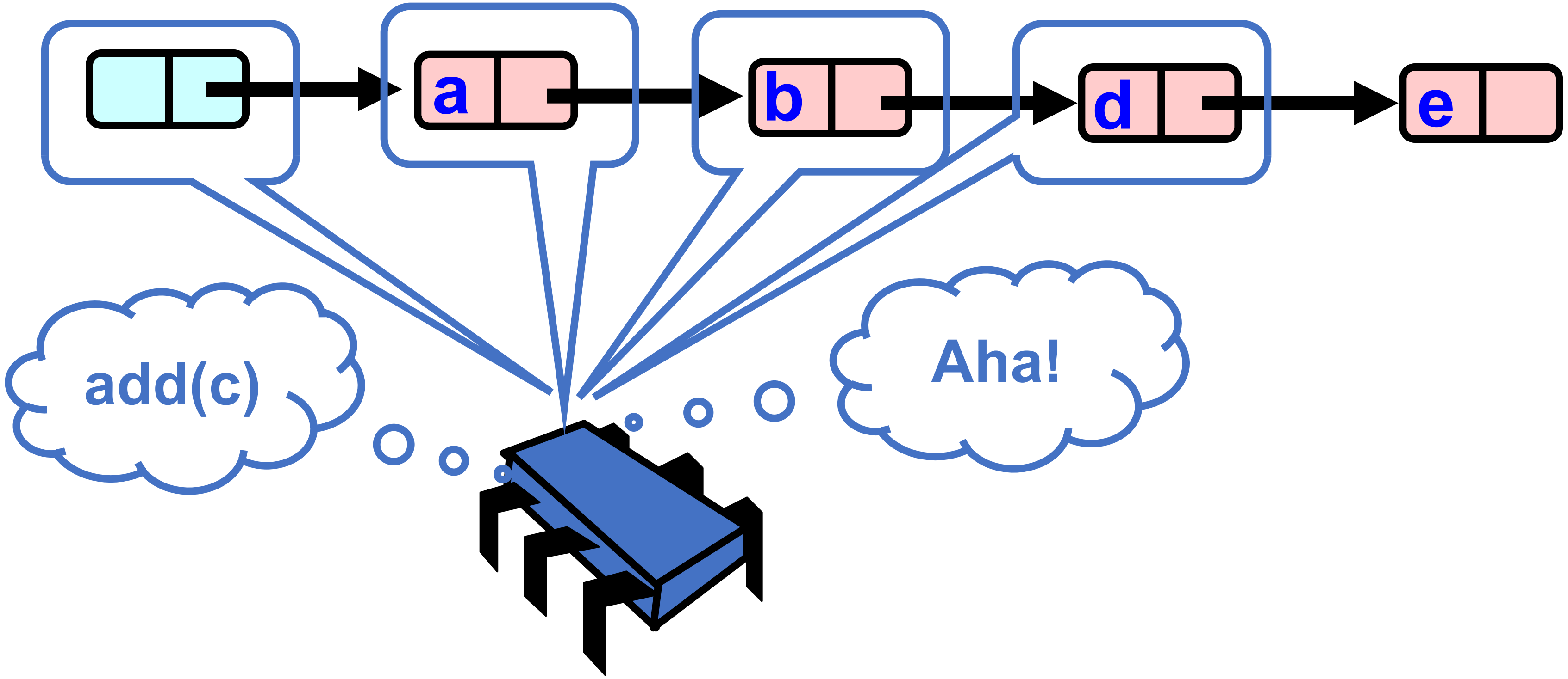


Uh, Oh

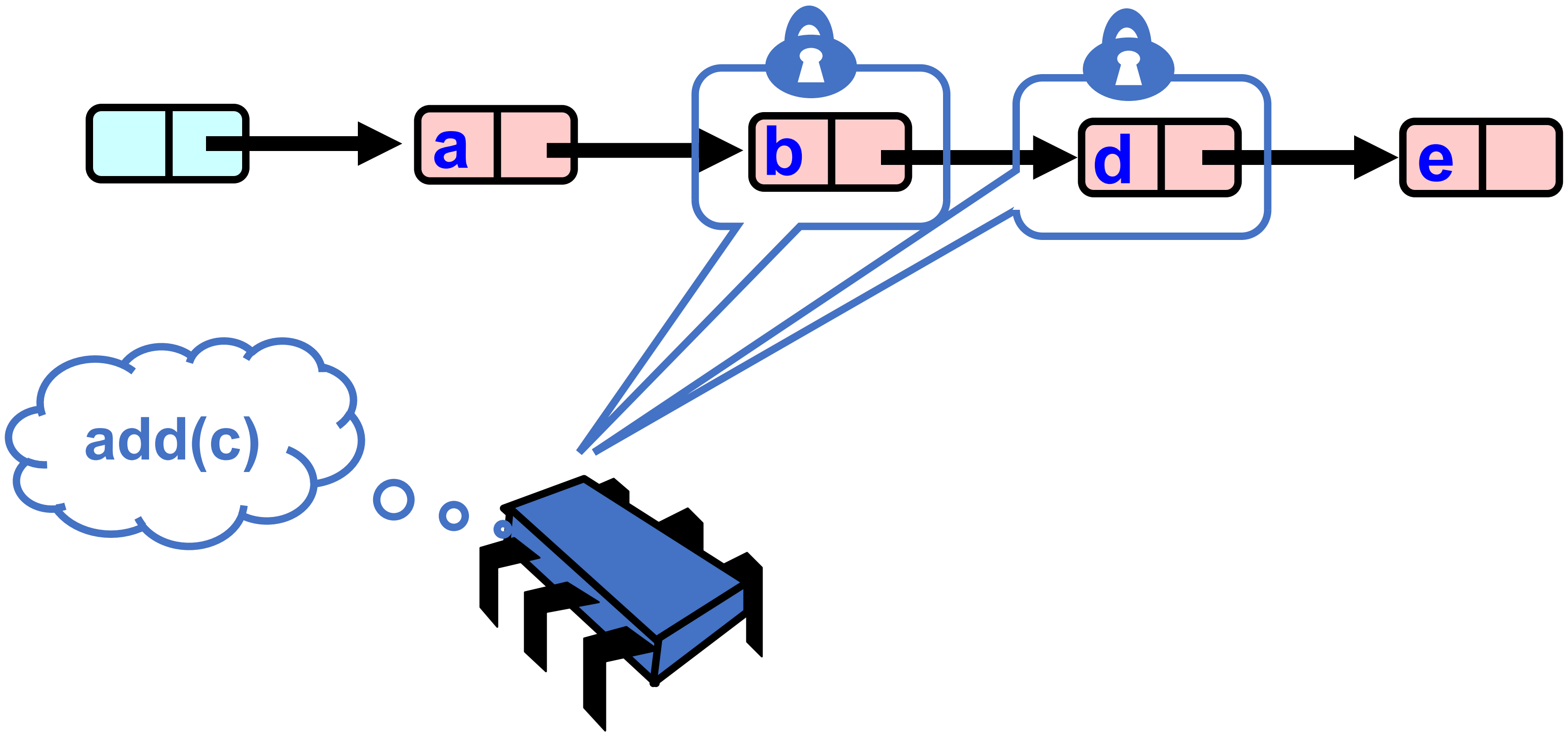


Optimistic traversing

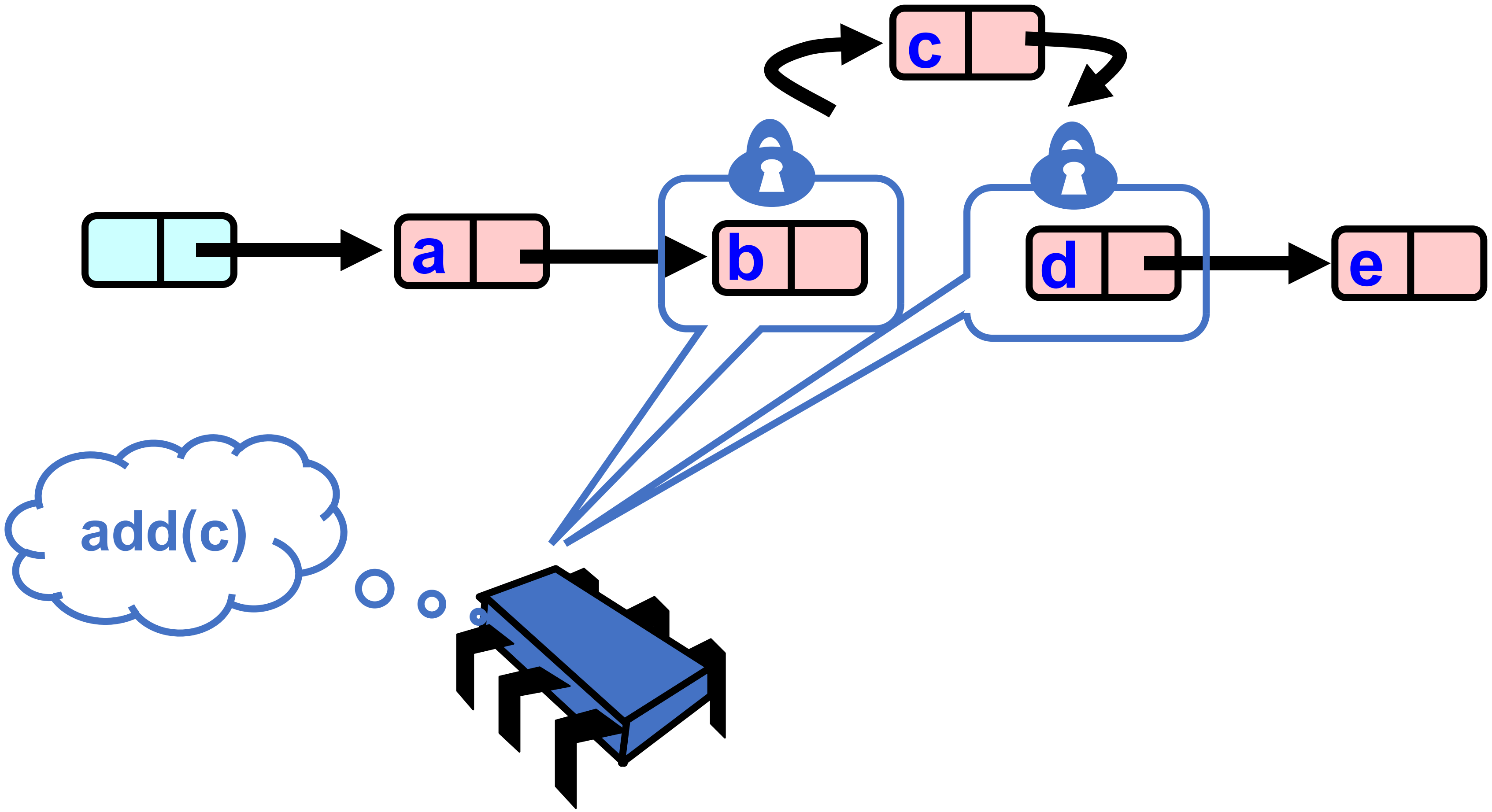
Optimistic: Traverse without Locking



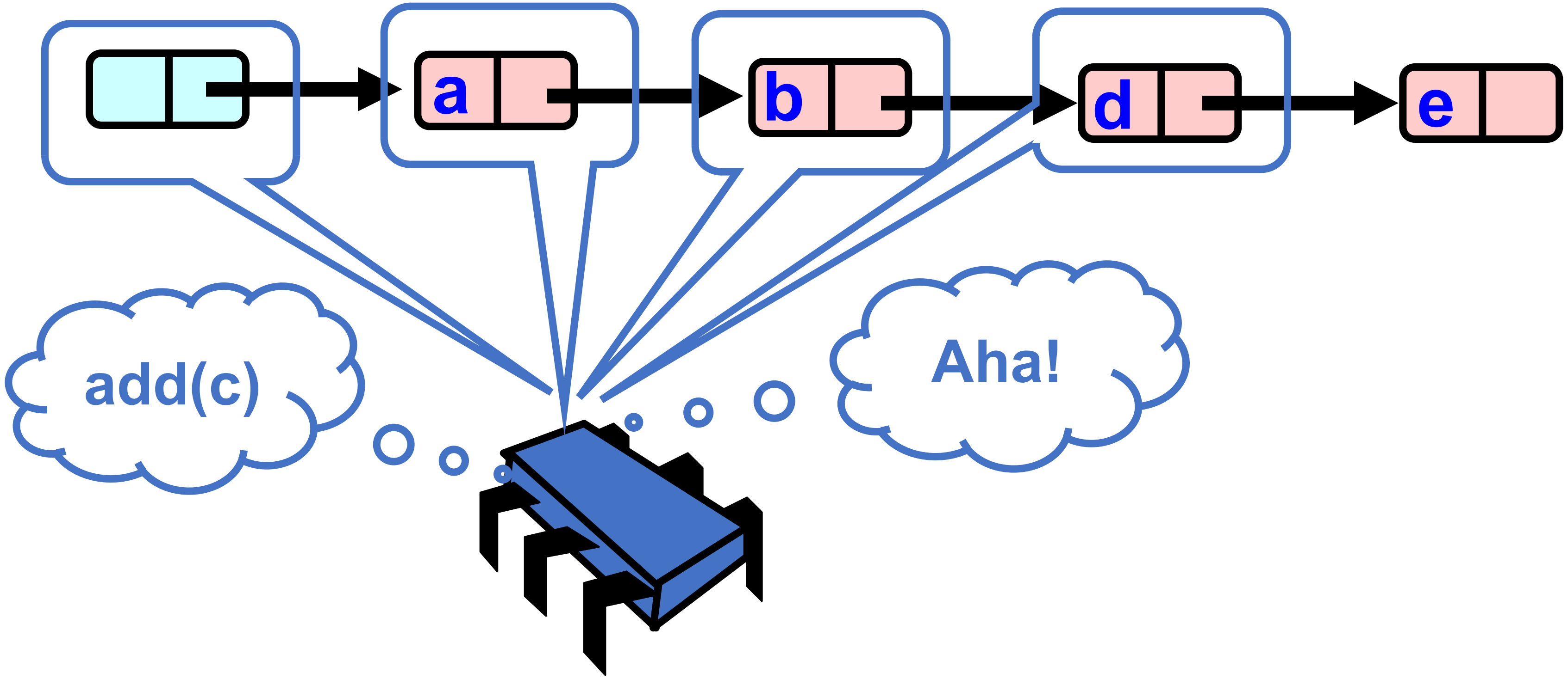
Optimistic: Lock and Load



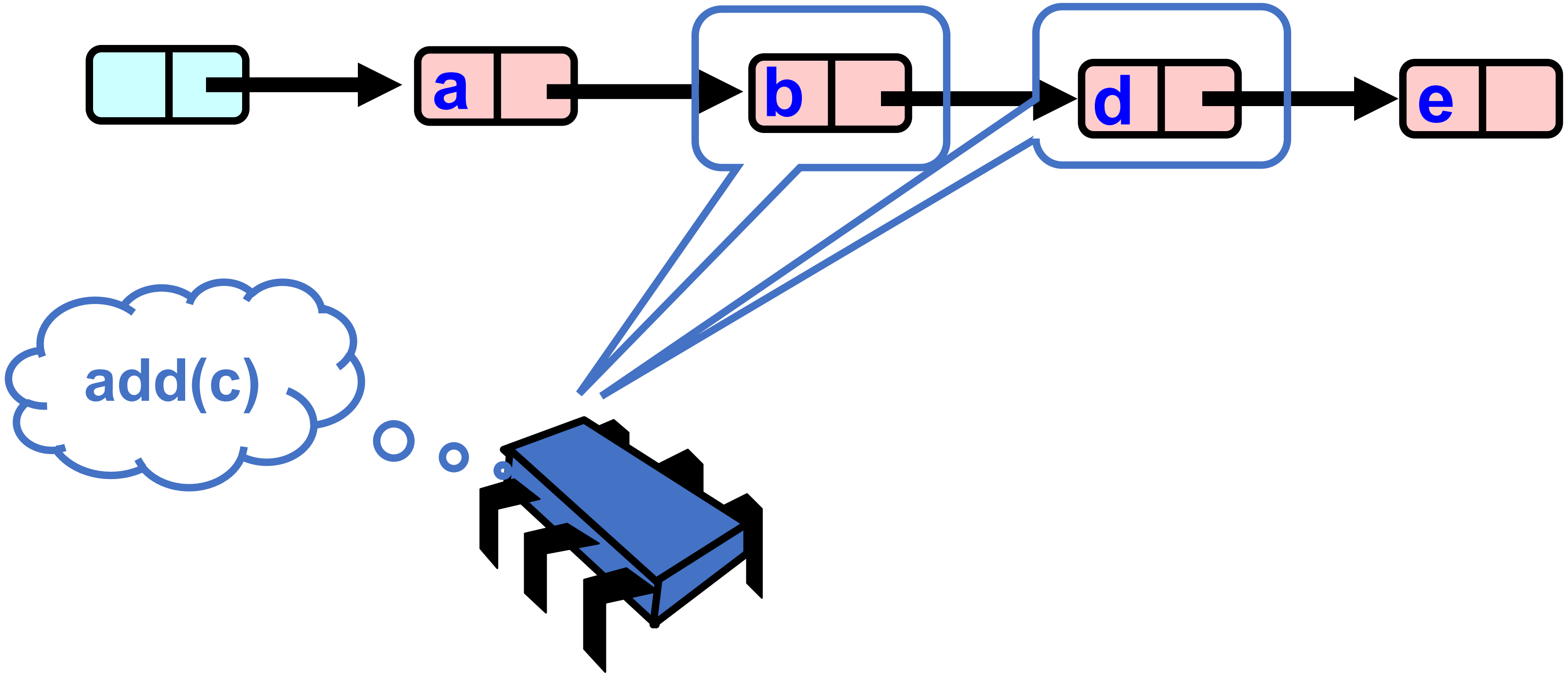
Optimistic: Lock and Load



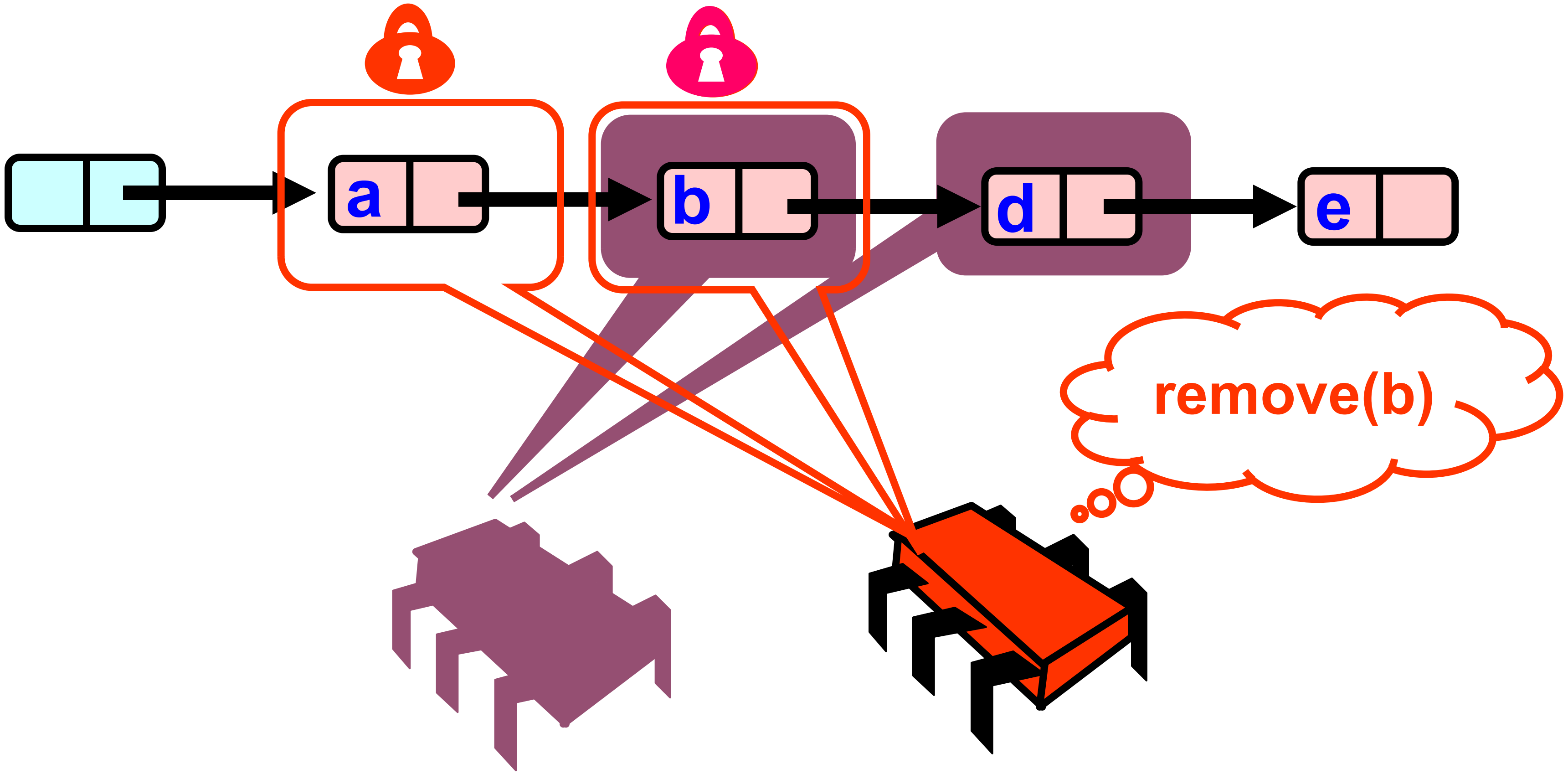
What could go wrong?



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

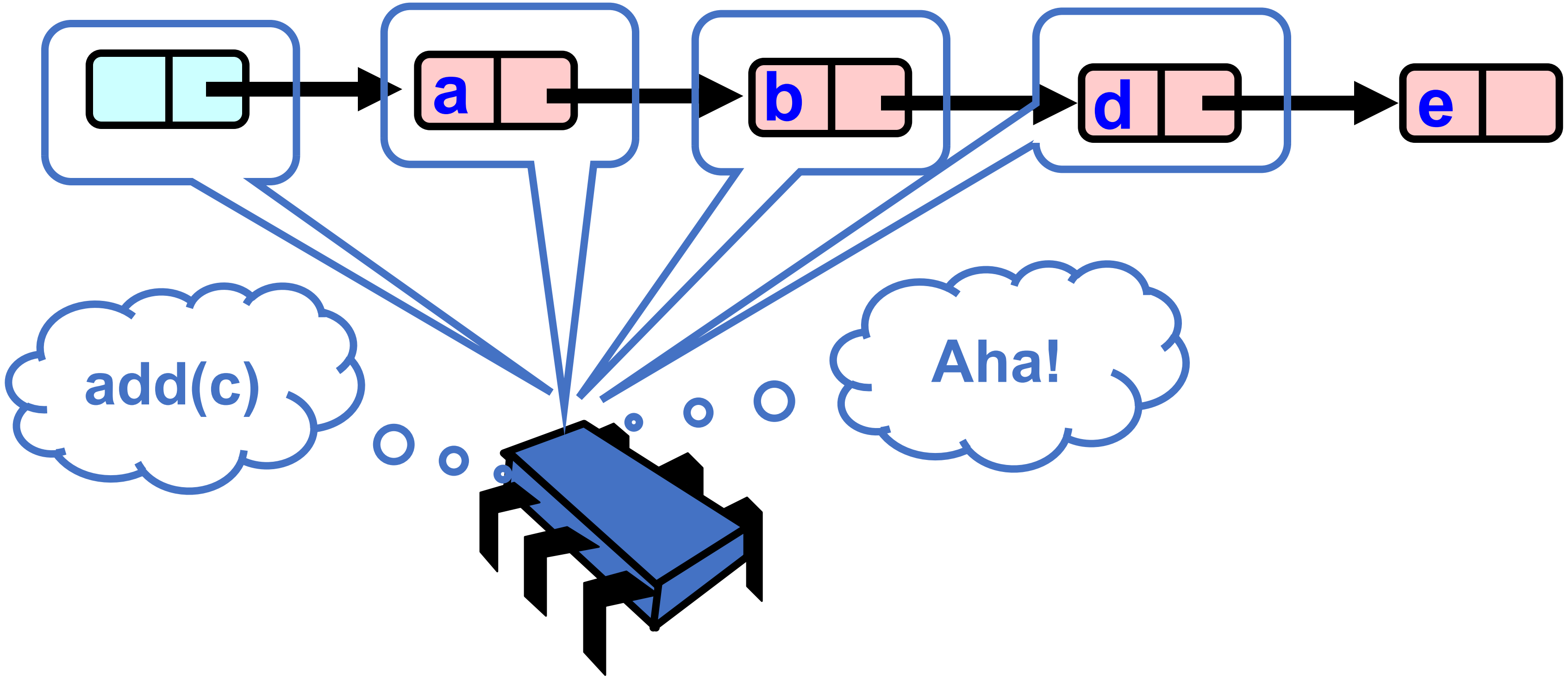
- What do we do?

Lock-free reasoning

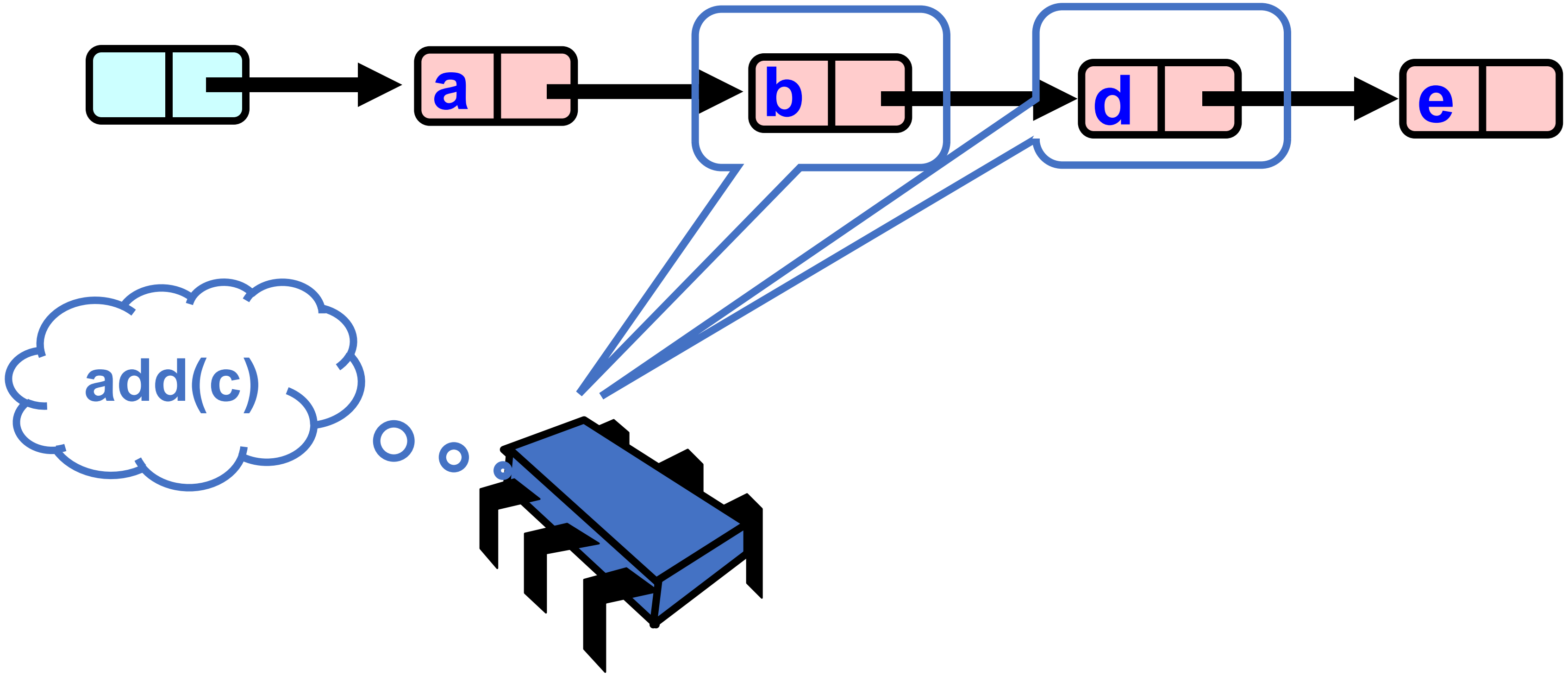
- Default atomic accesses are documented to be sequentially consistent.

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

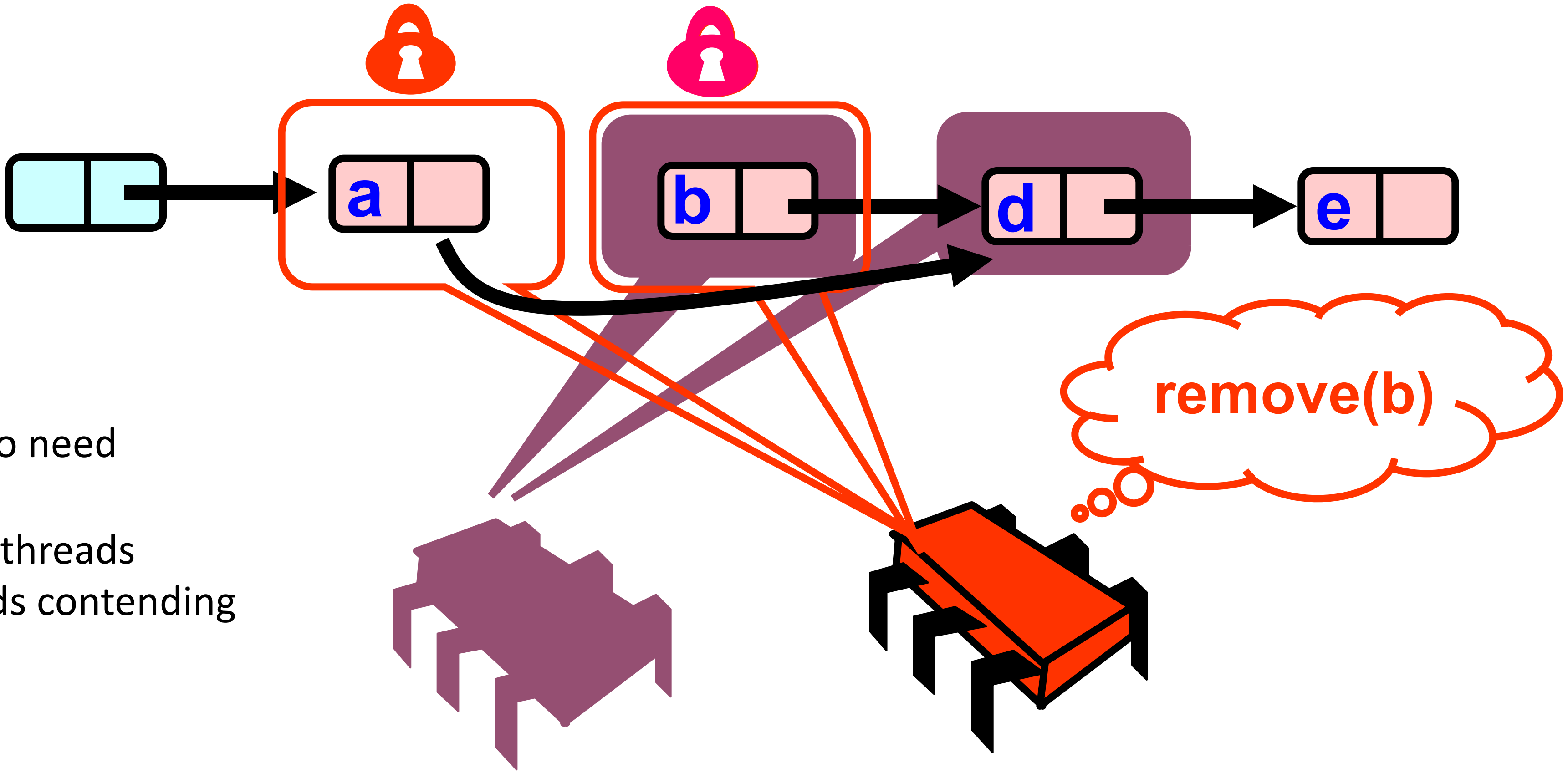
What could go wrong?



What could go wrong?

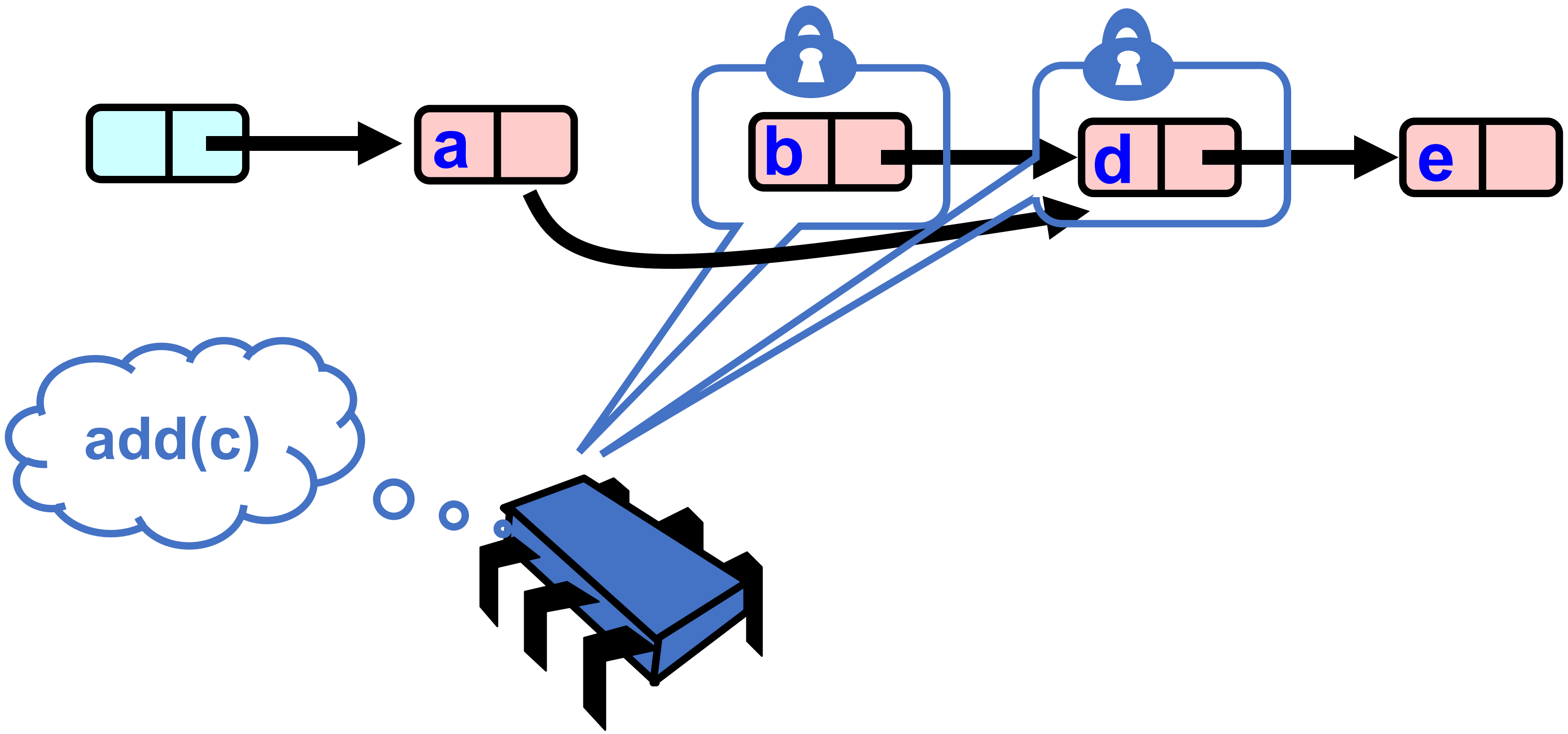


What could go wrong?

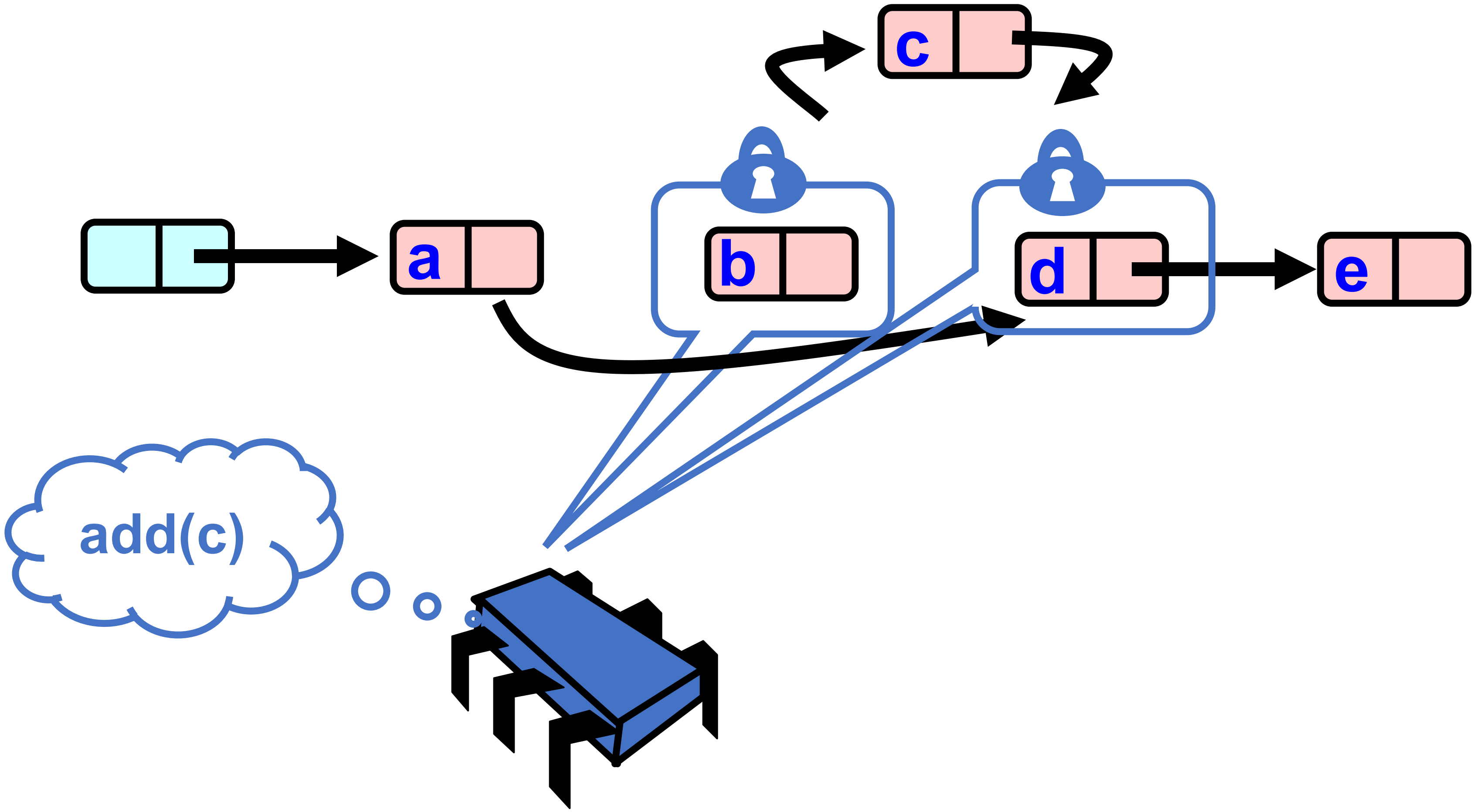


No more data conflict, but we do need to reason about interleavings and threads concurrent threads contending for values.

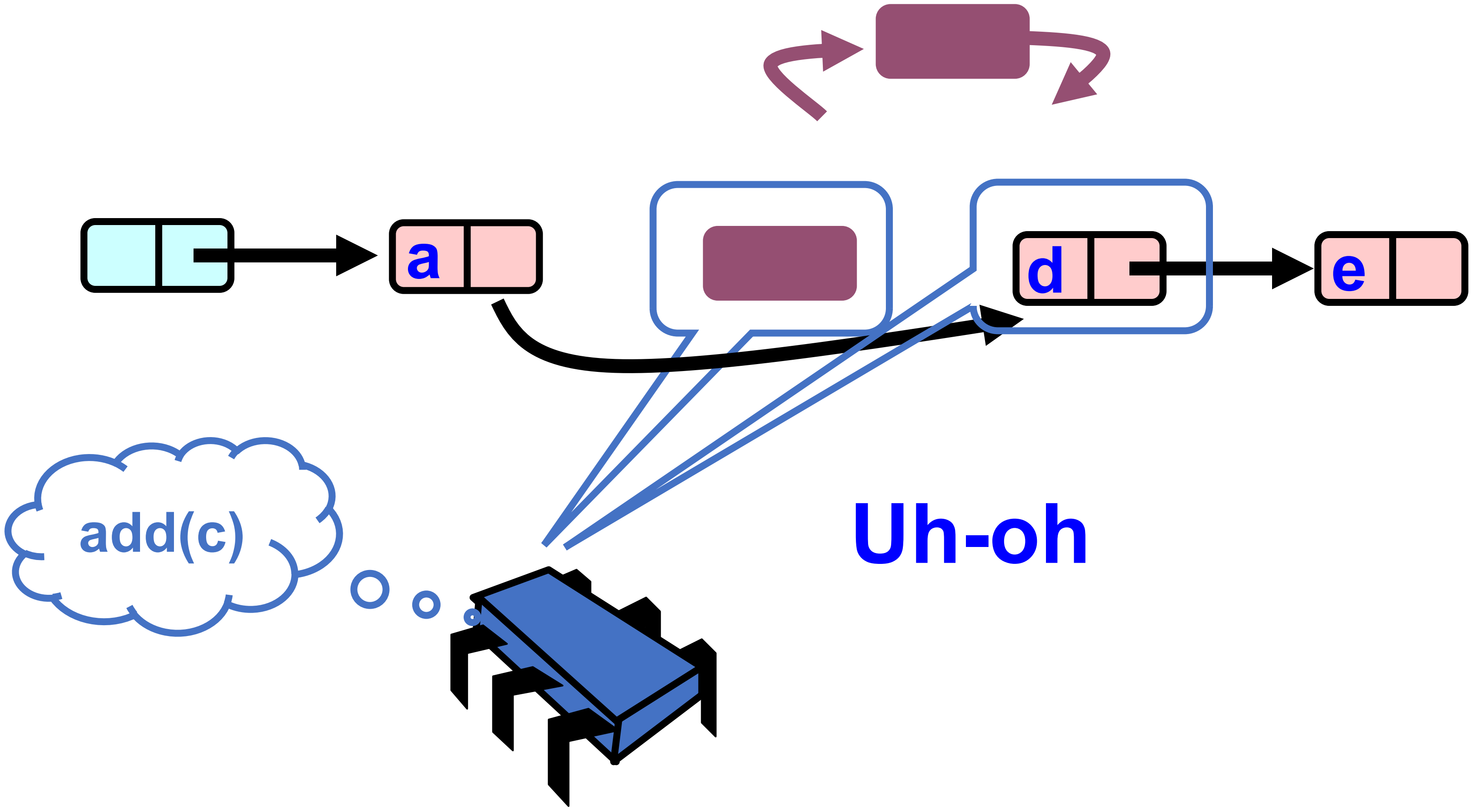
What could go wrong?



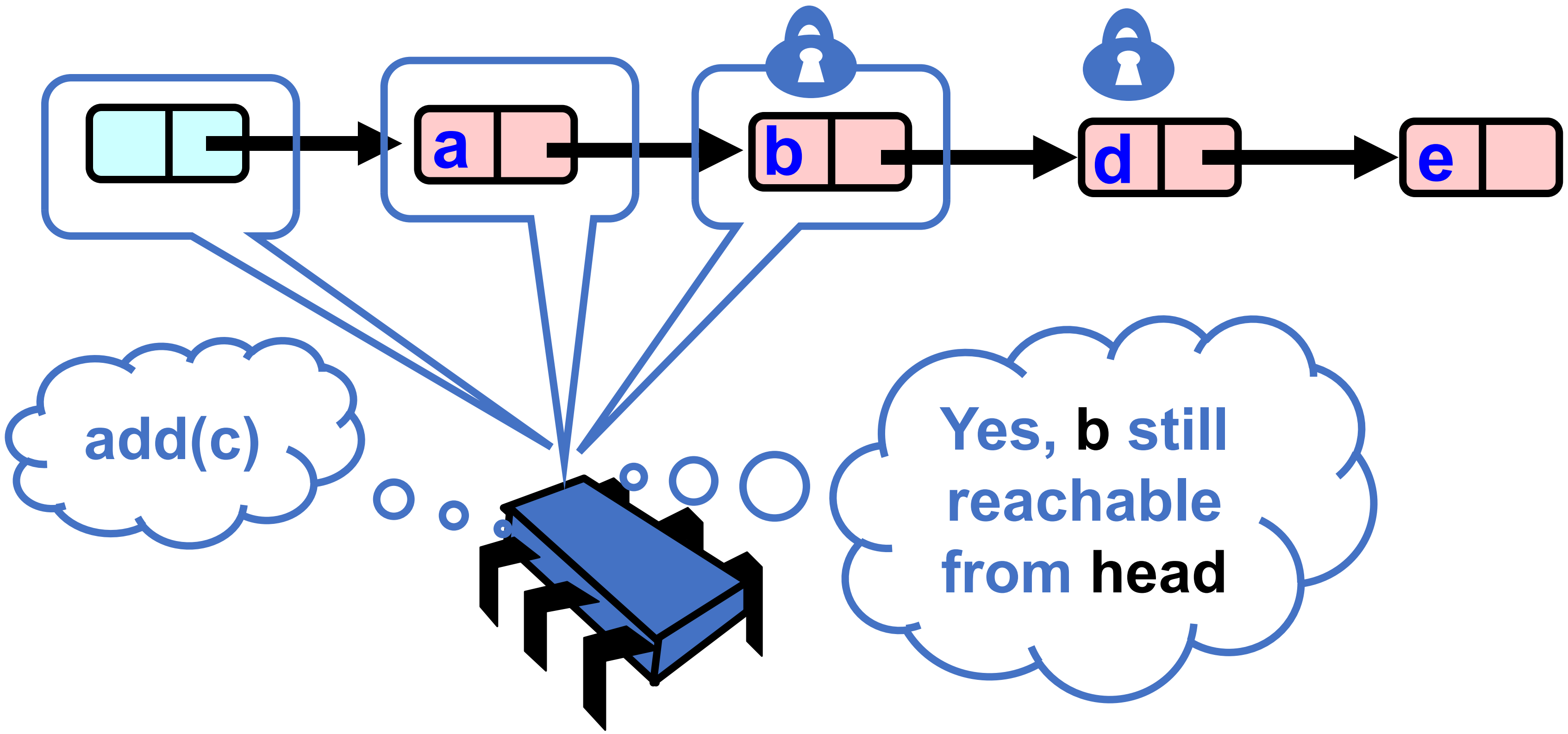
What could go wrong?



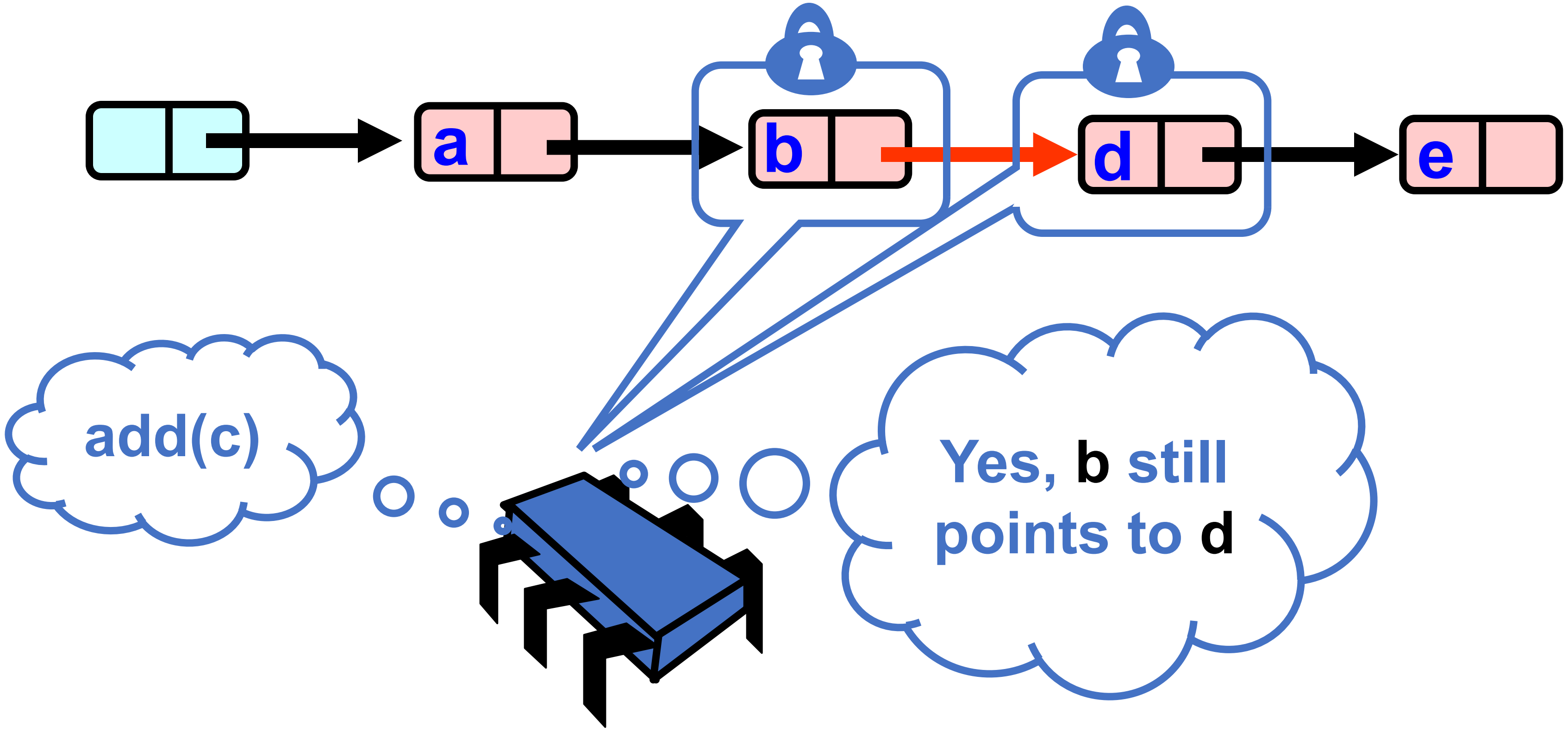
What could go wrong?



Validate – Part 1



Validate Part 2
(while holding locks)



Even more difficulties

- We had to implement our own garbage collector ☹

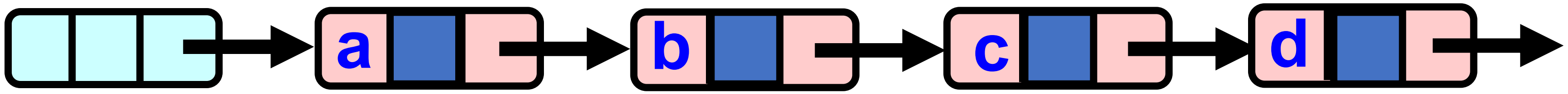
Can we optimize more?

- Scan the list once?

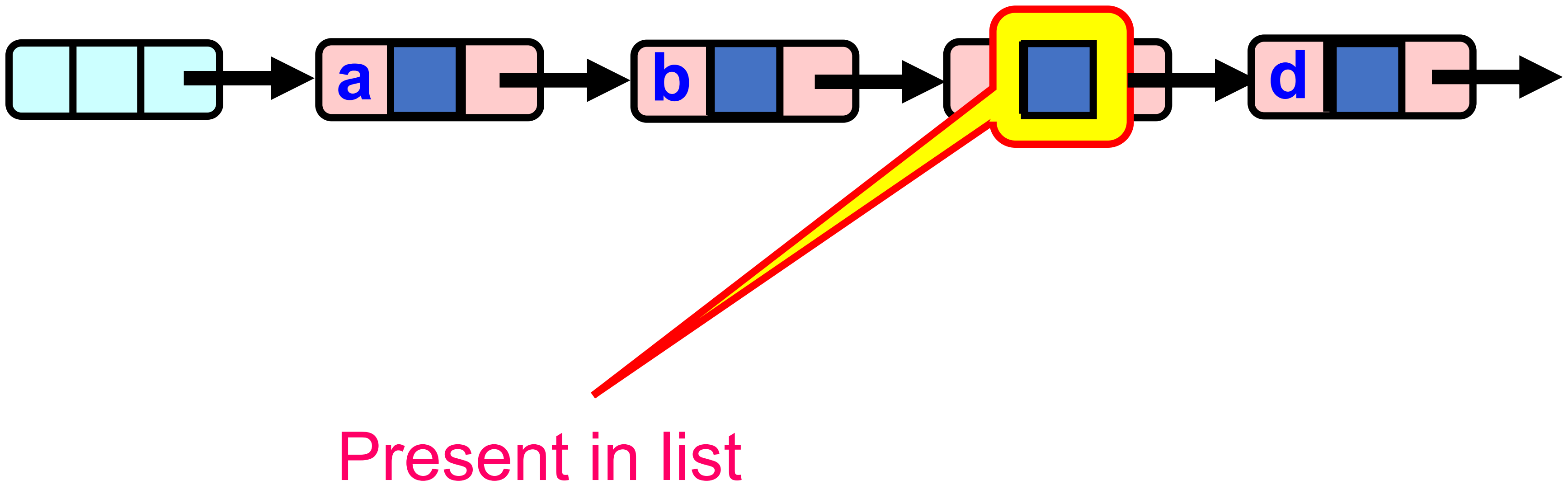
Two step removal List

- **`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)

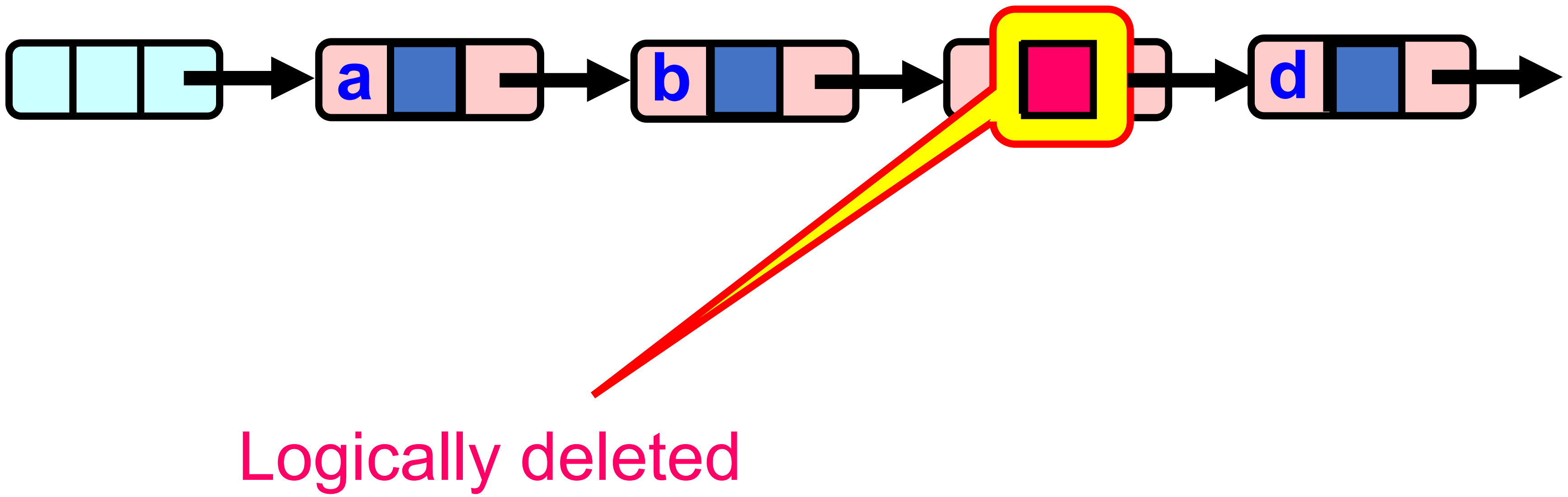
Two step removal Removal



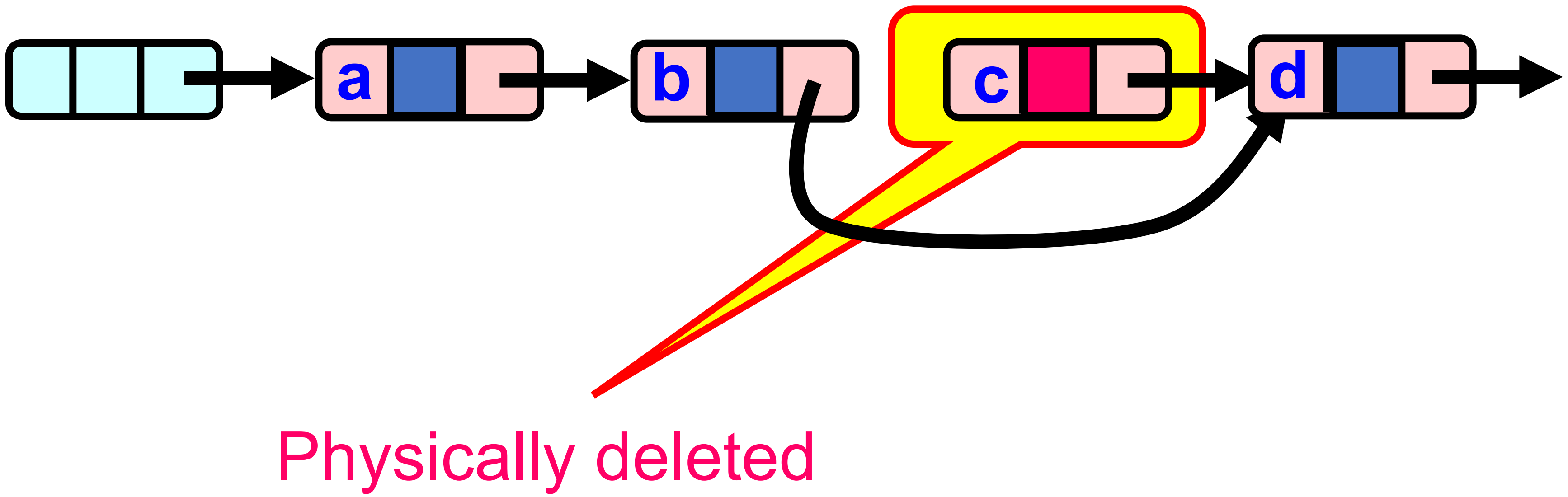
Two step removal Removal



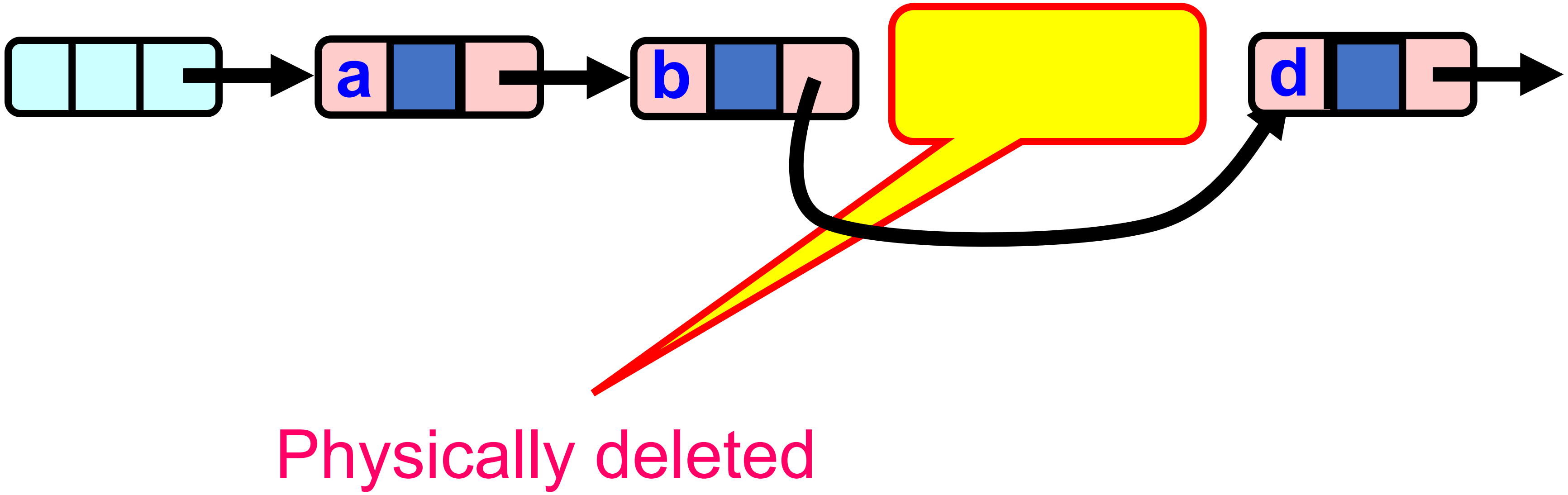
Two step removal Removal



Two step removal Removal



Two step removal Removal



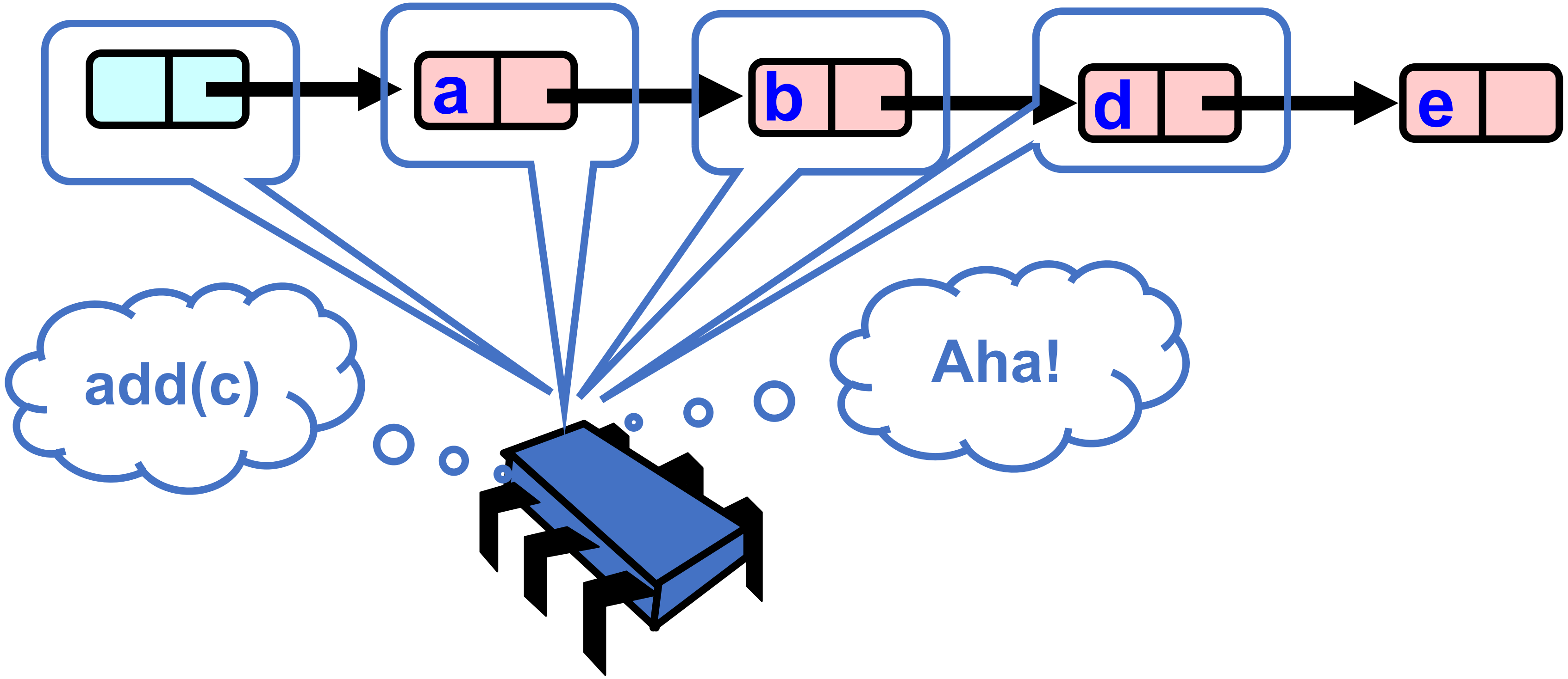
Two step remove list

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

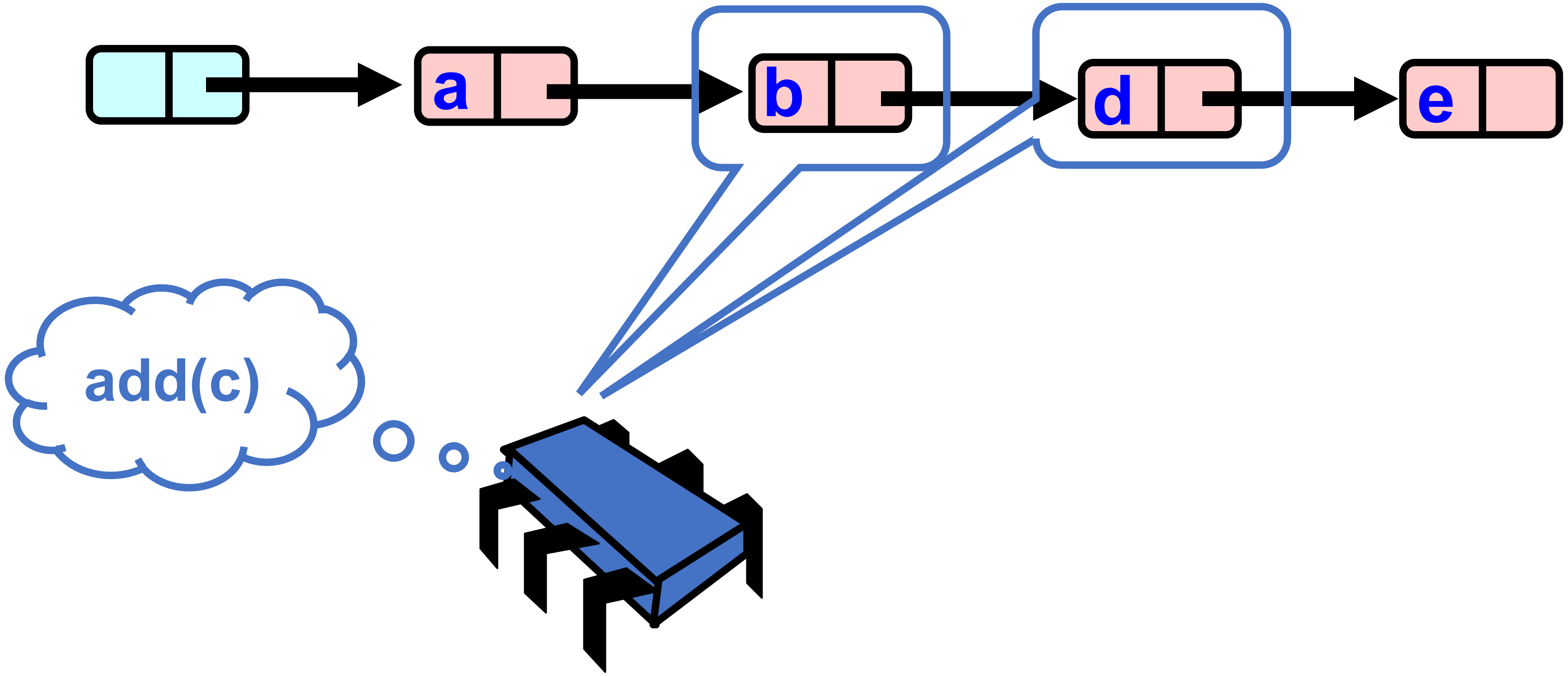
Validation

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

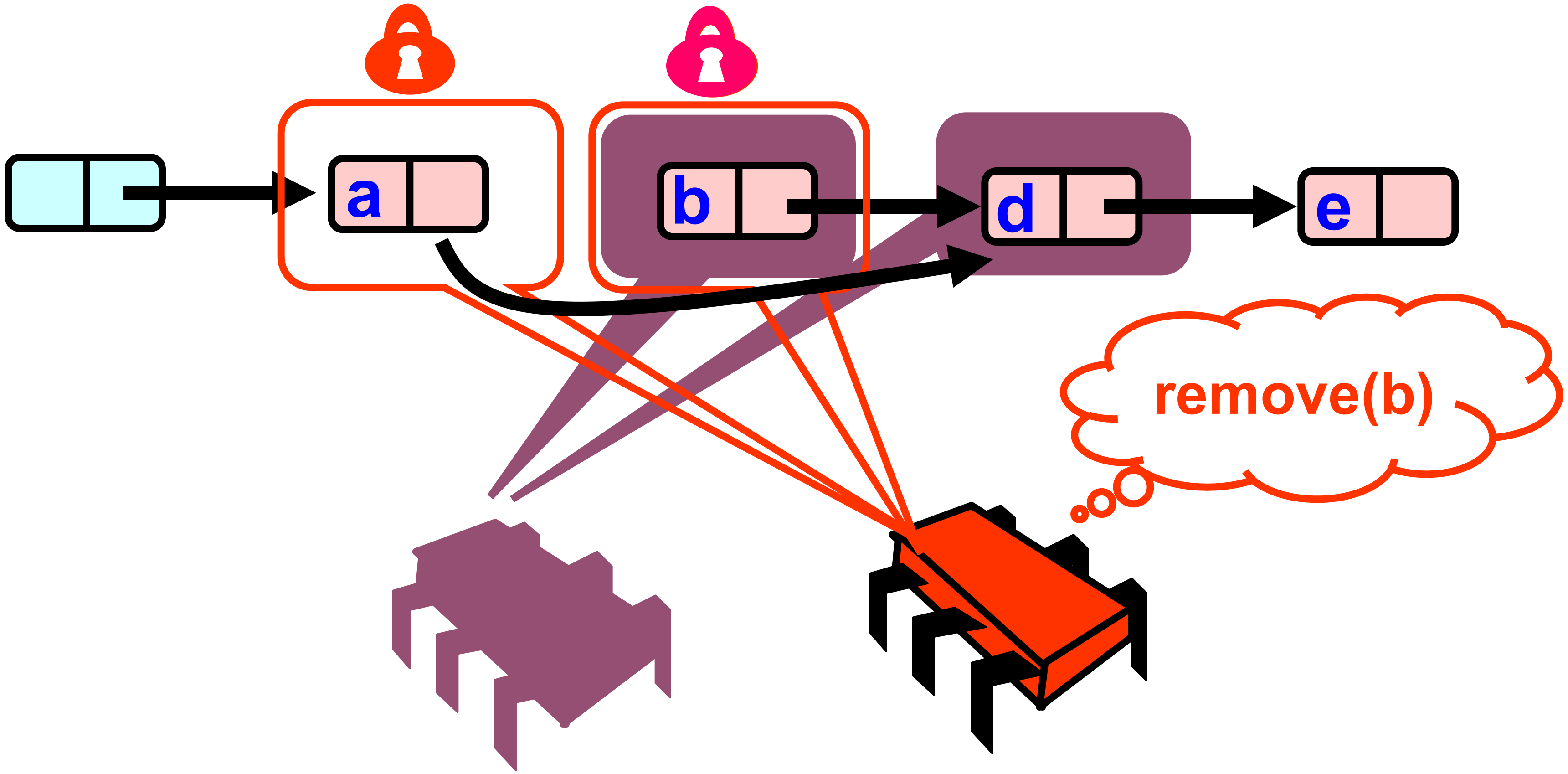
What could go wrong?



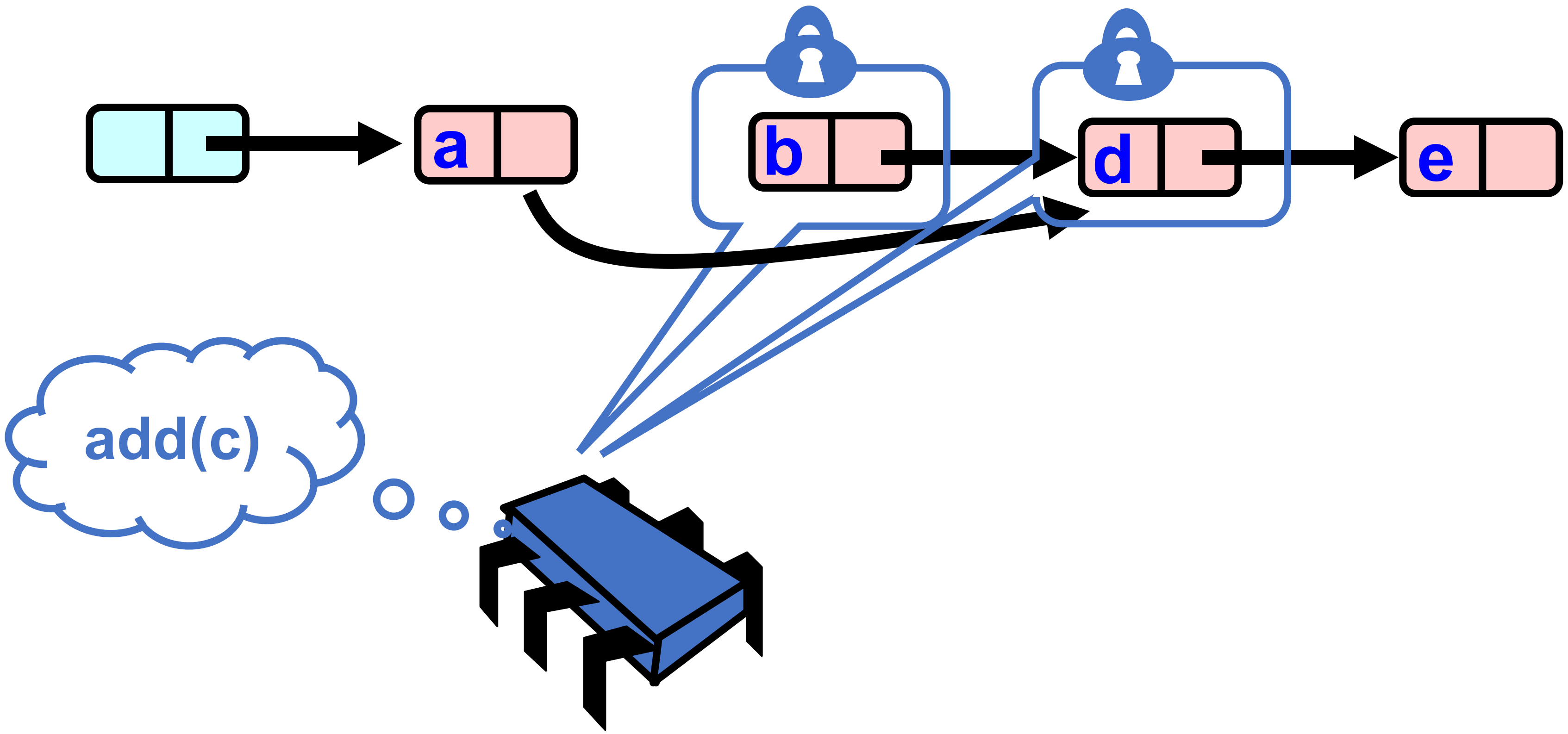
What could go wrong?



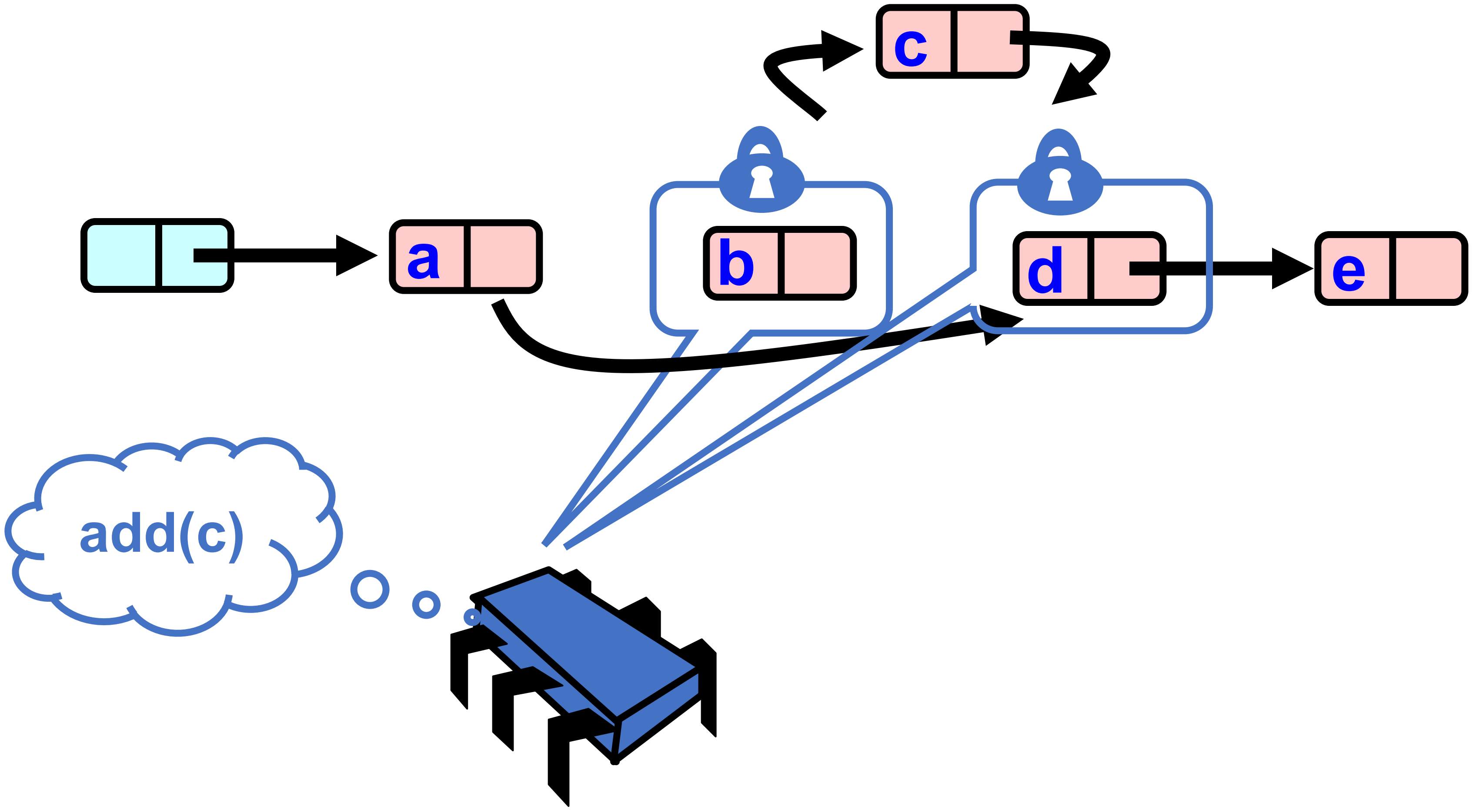
What could go wrong?



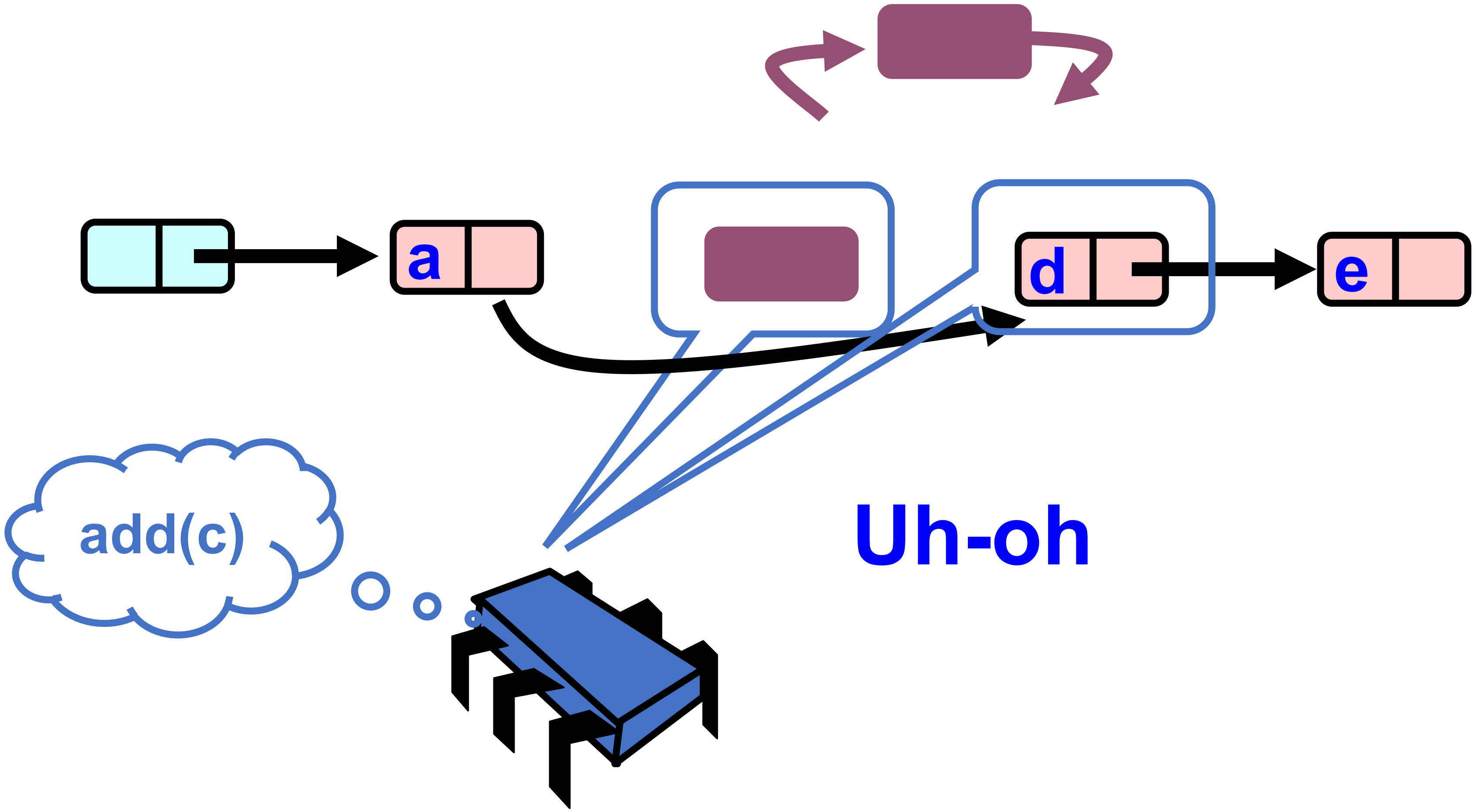
What could go wrong?



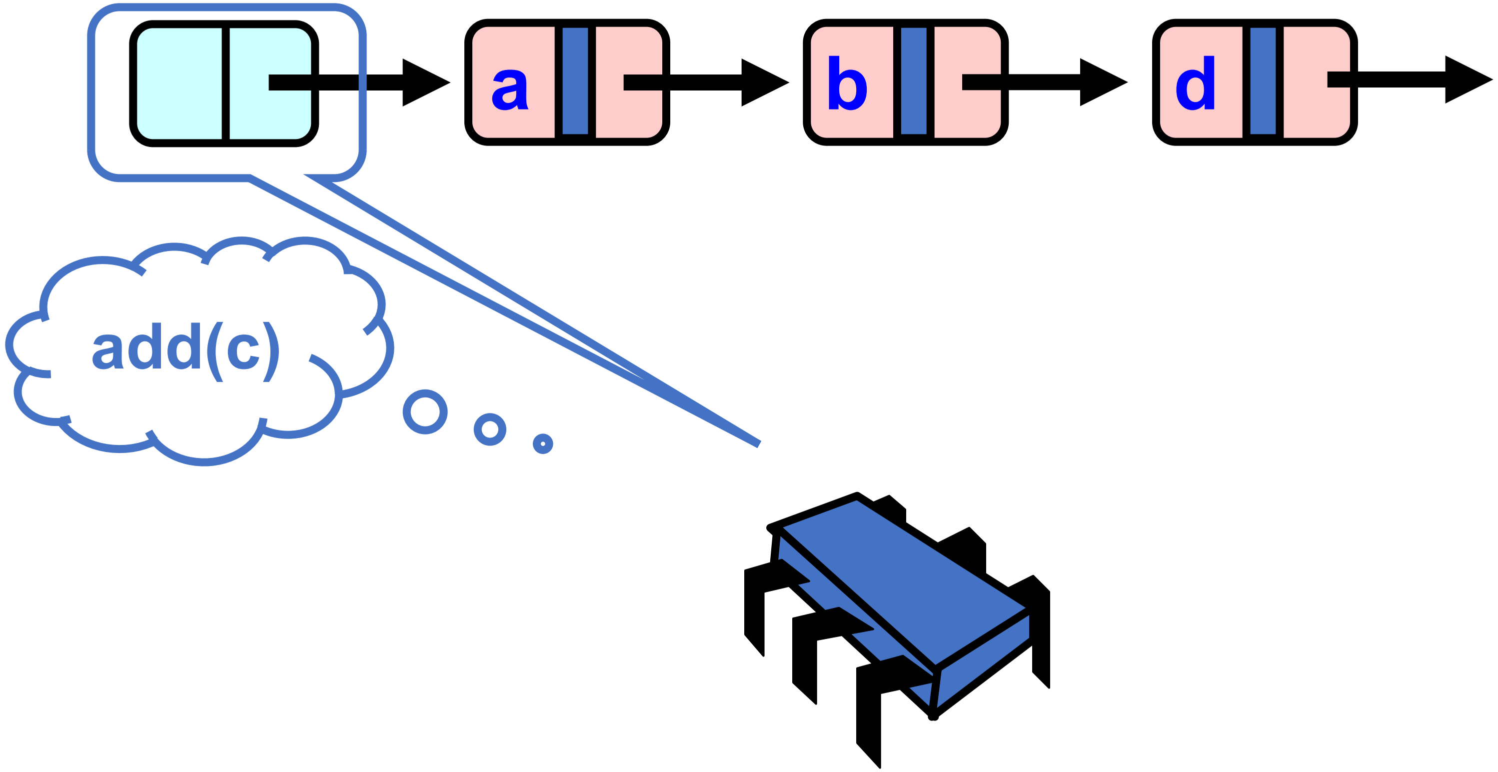
What could go wrong?



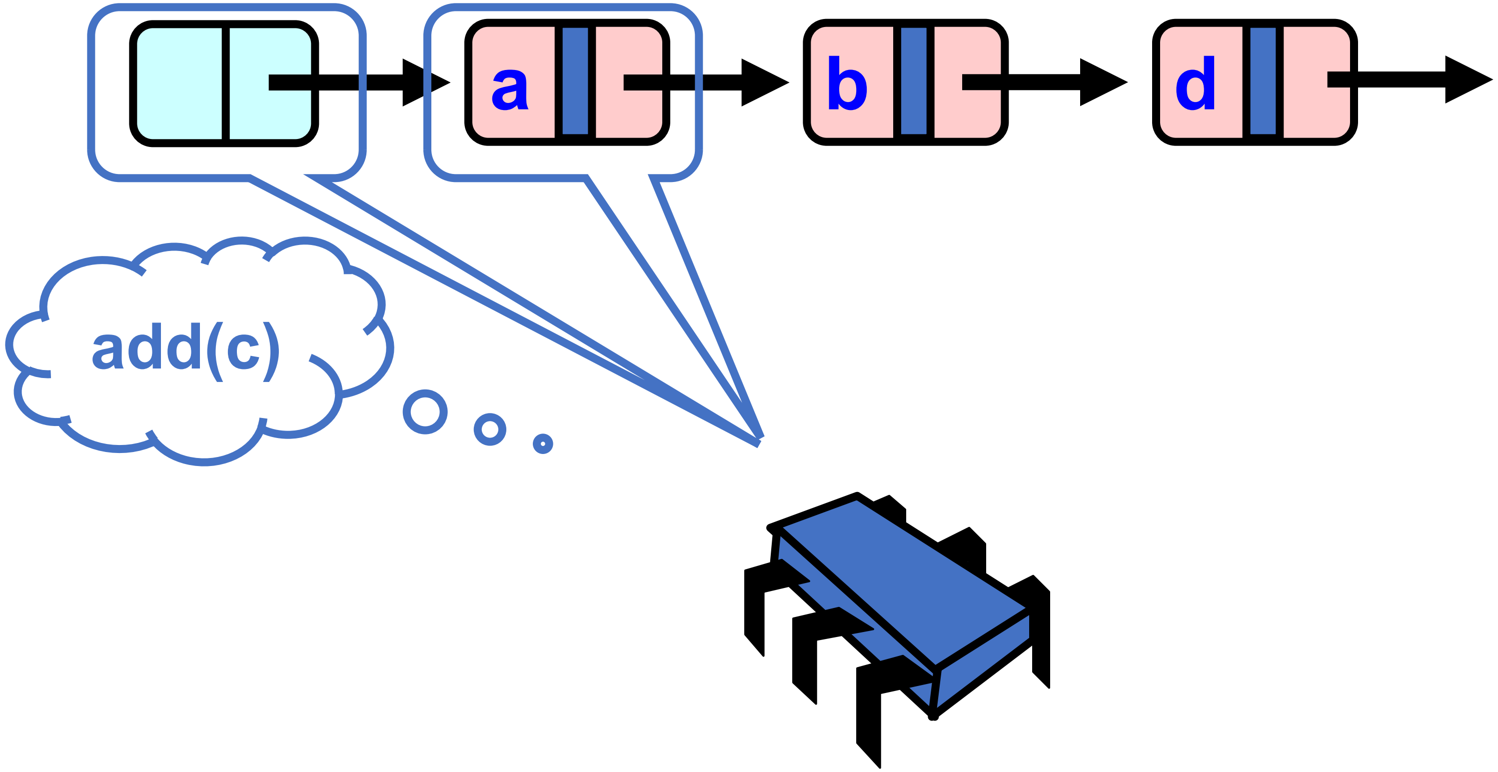
What could go wrong?



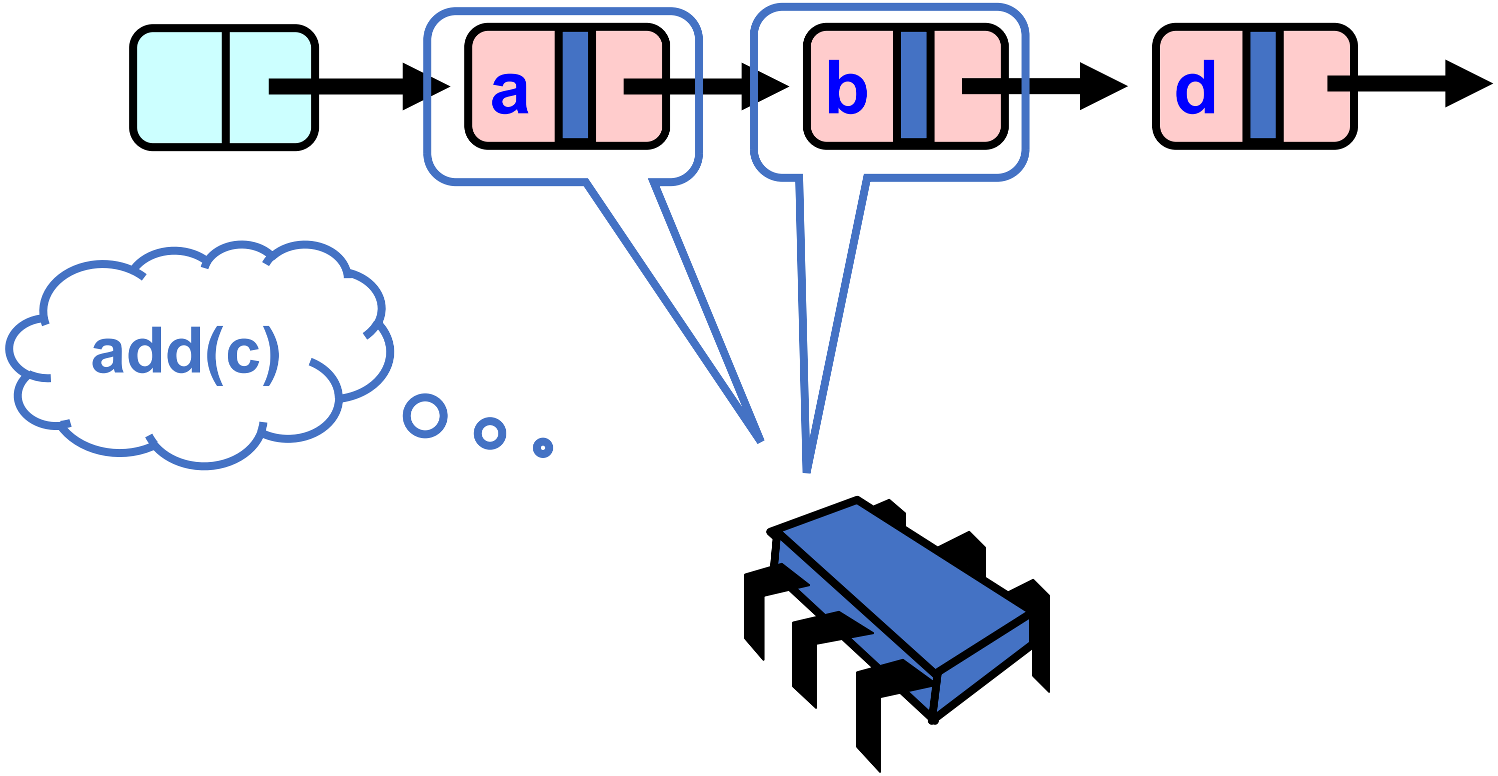
Fixed with logical flag



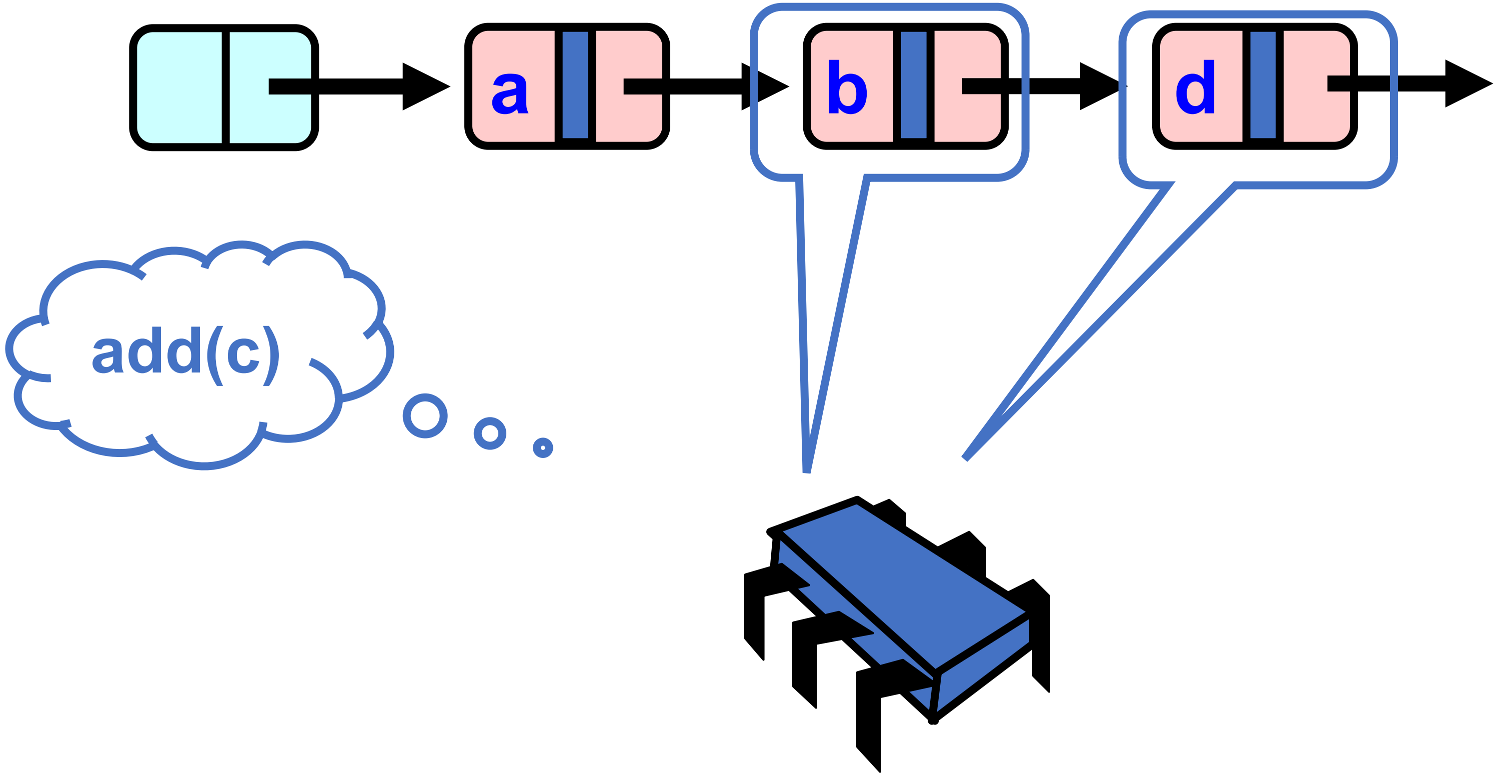
Fixed with logical flag



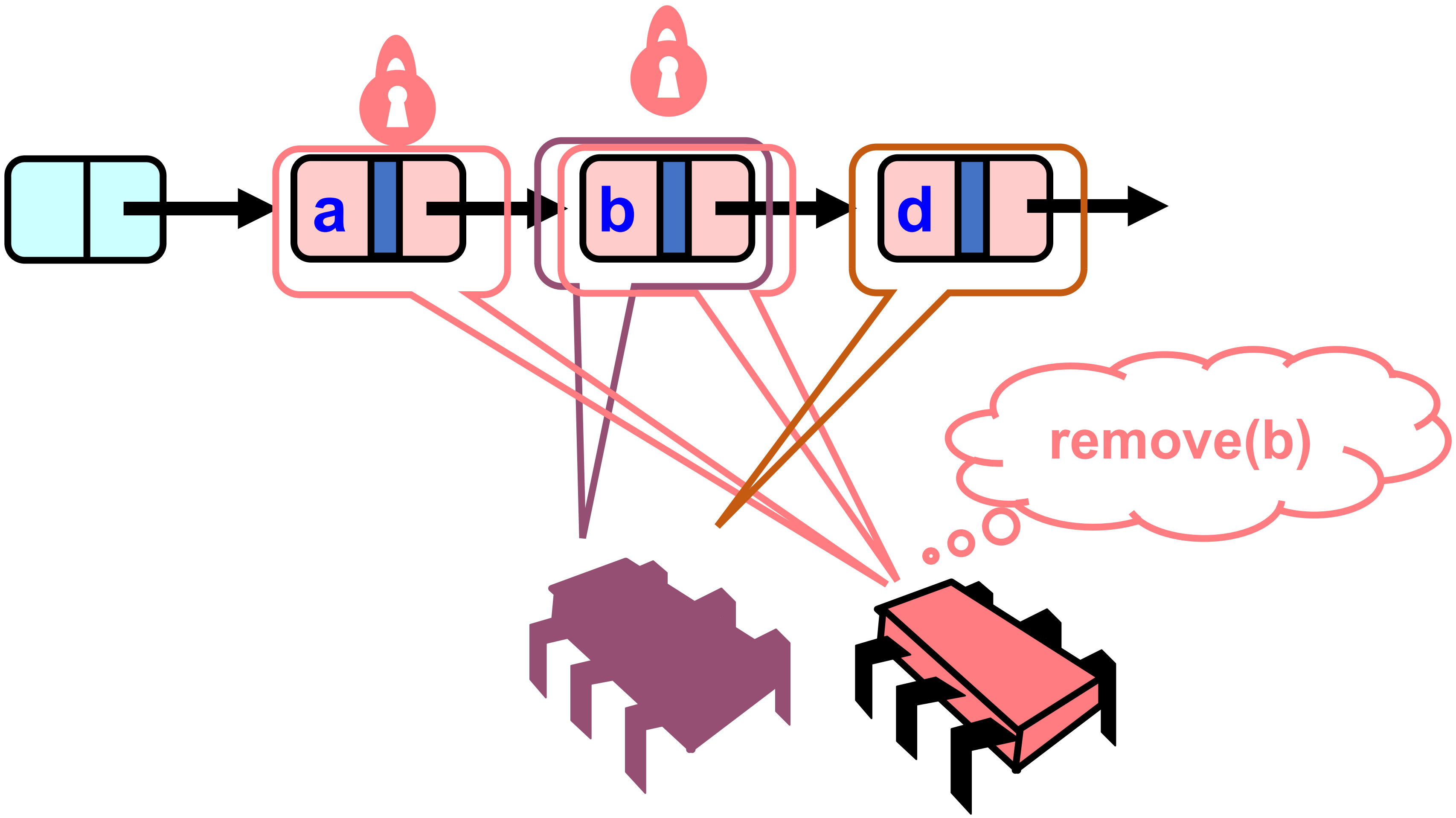
Fixed with logical flag



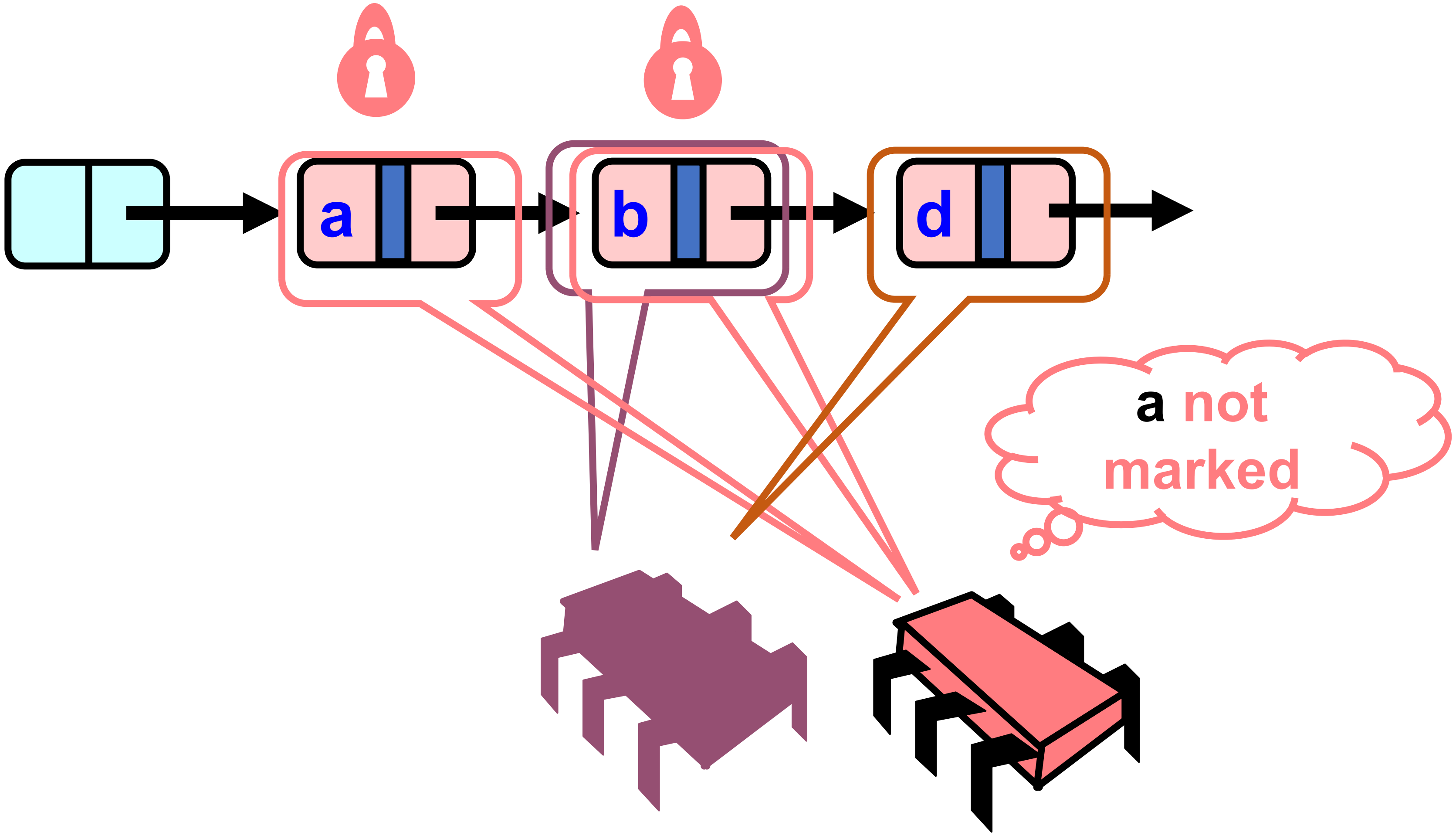
Fixed with logical flag



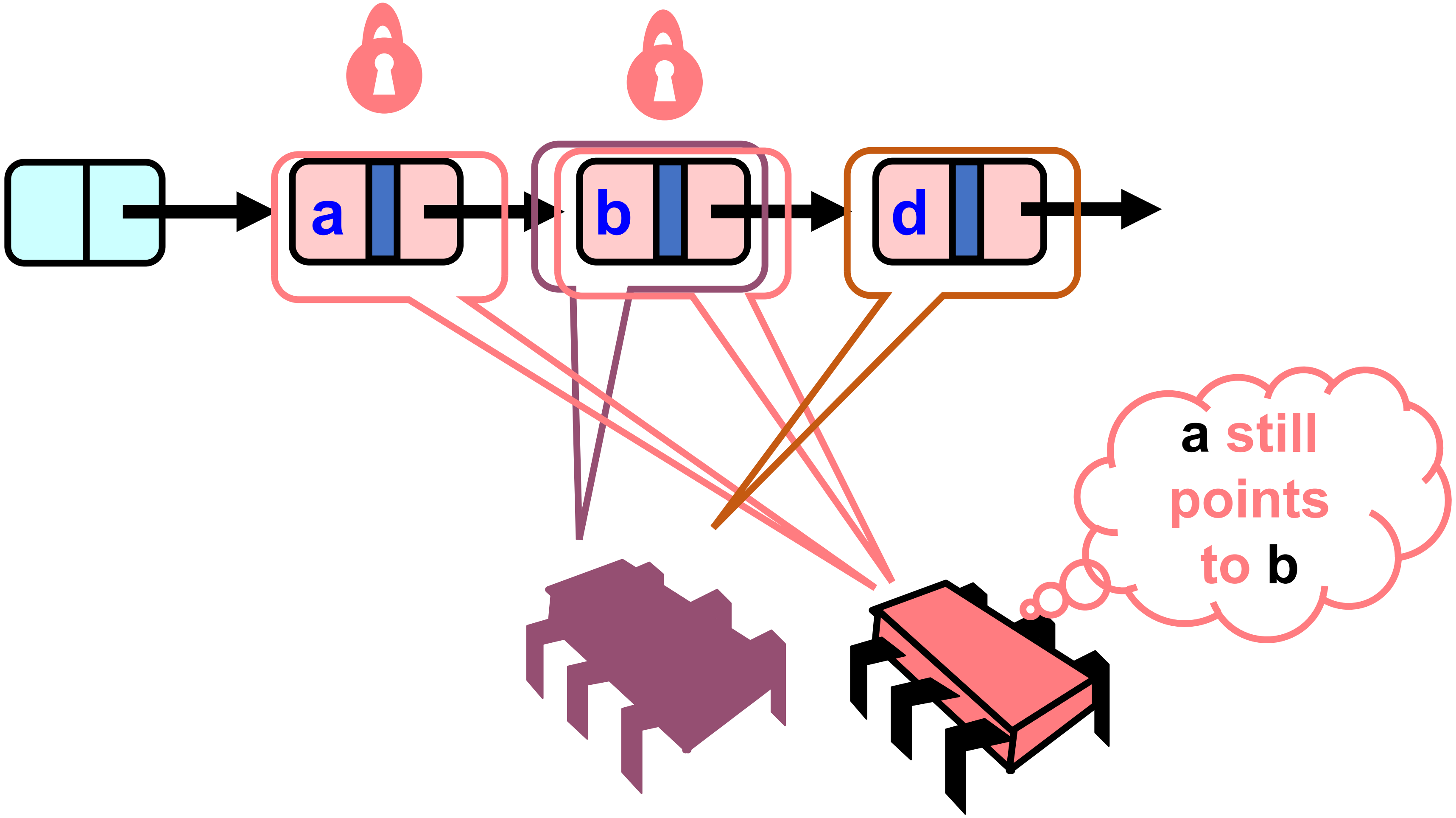
Fixed with logical flag



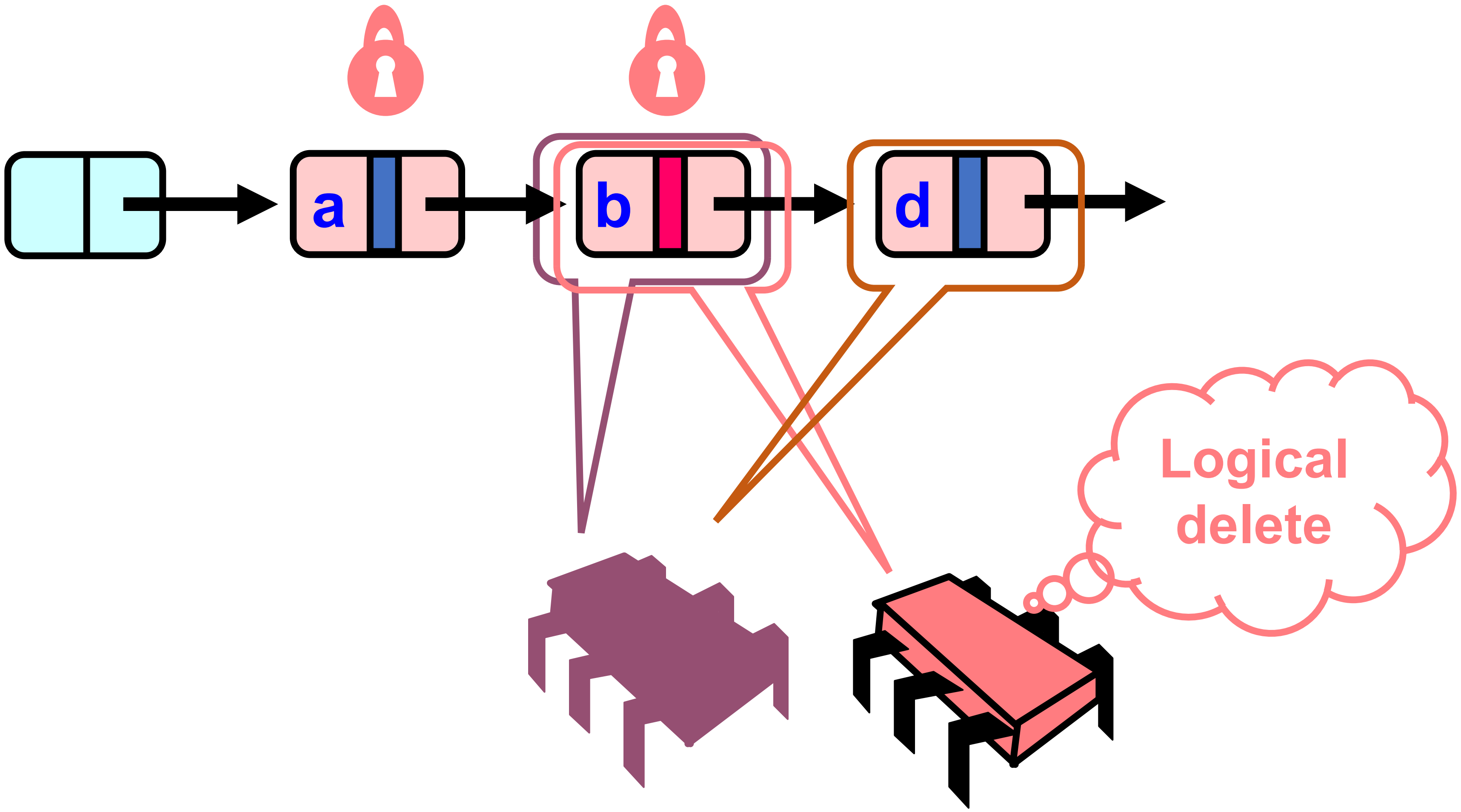
Fixed with logical flag



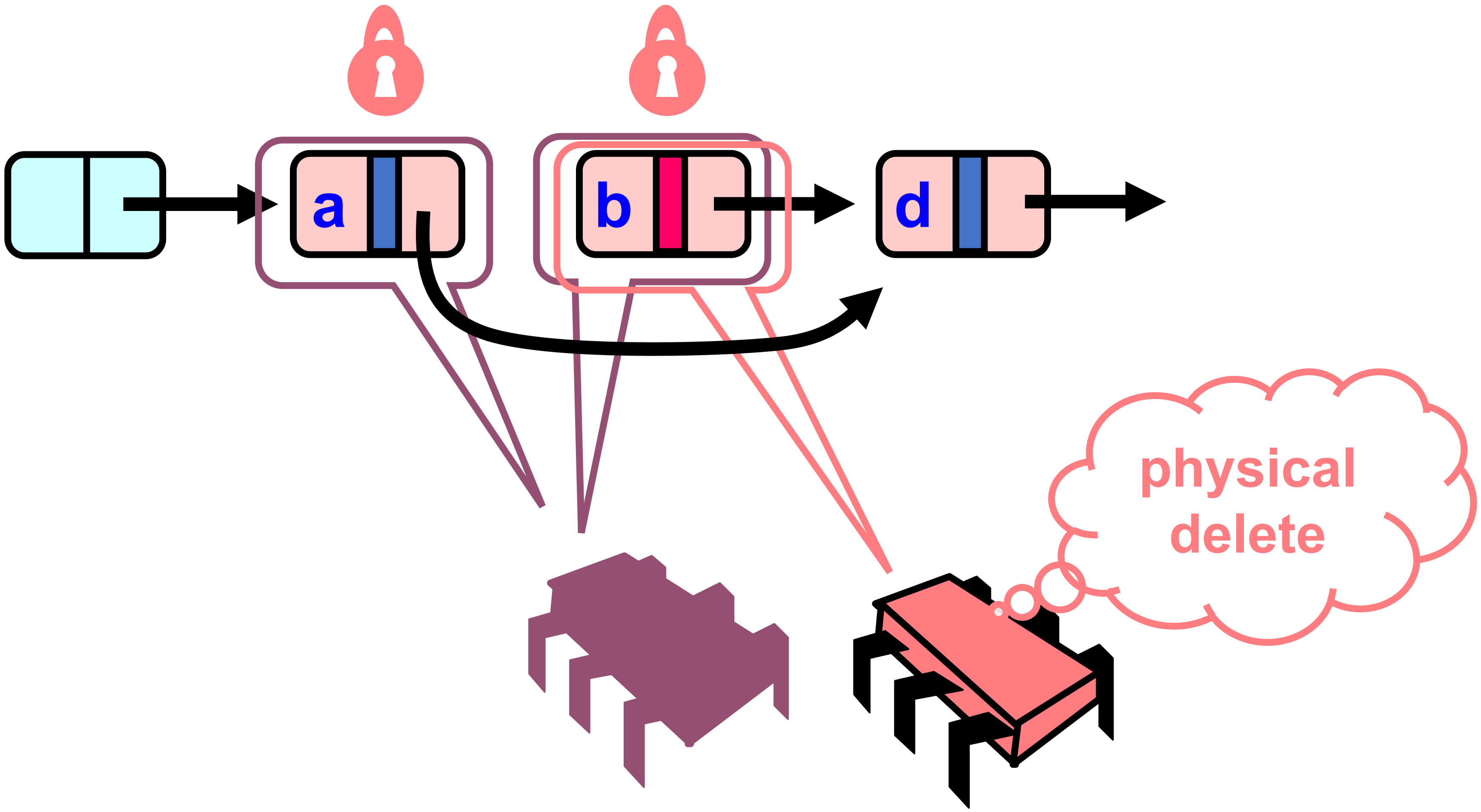
Fixed with logical flag



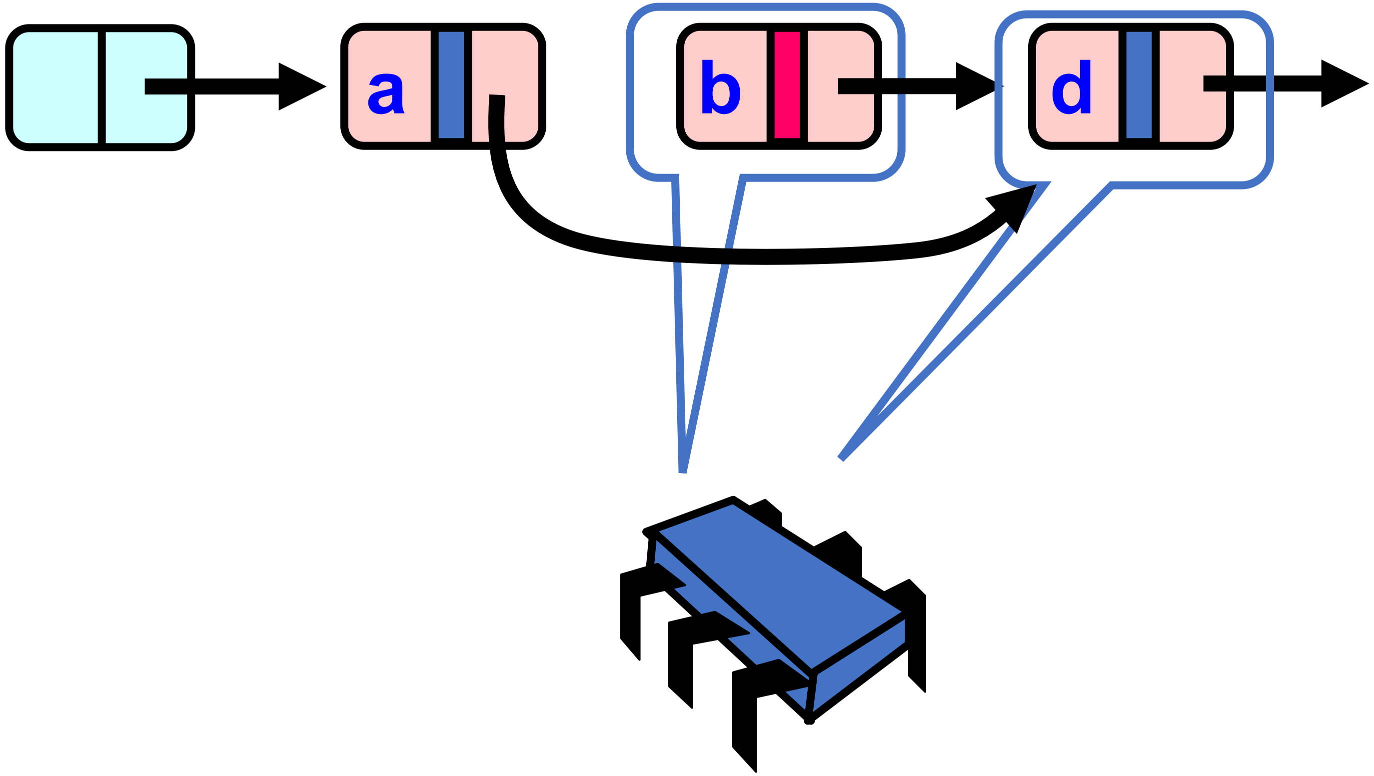
Fixed with logical flag



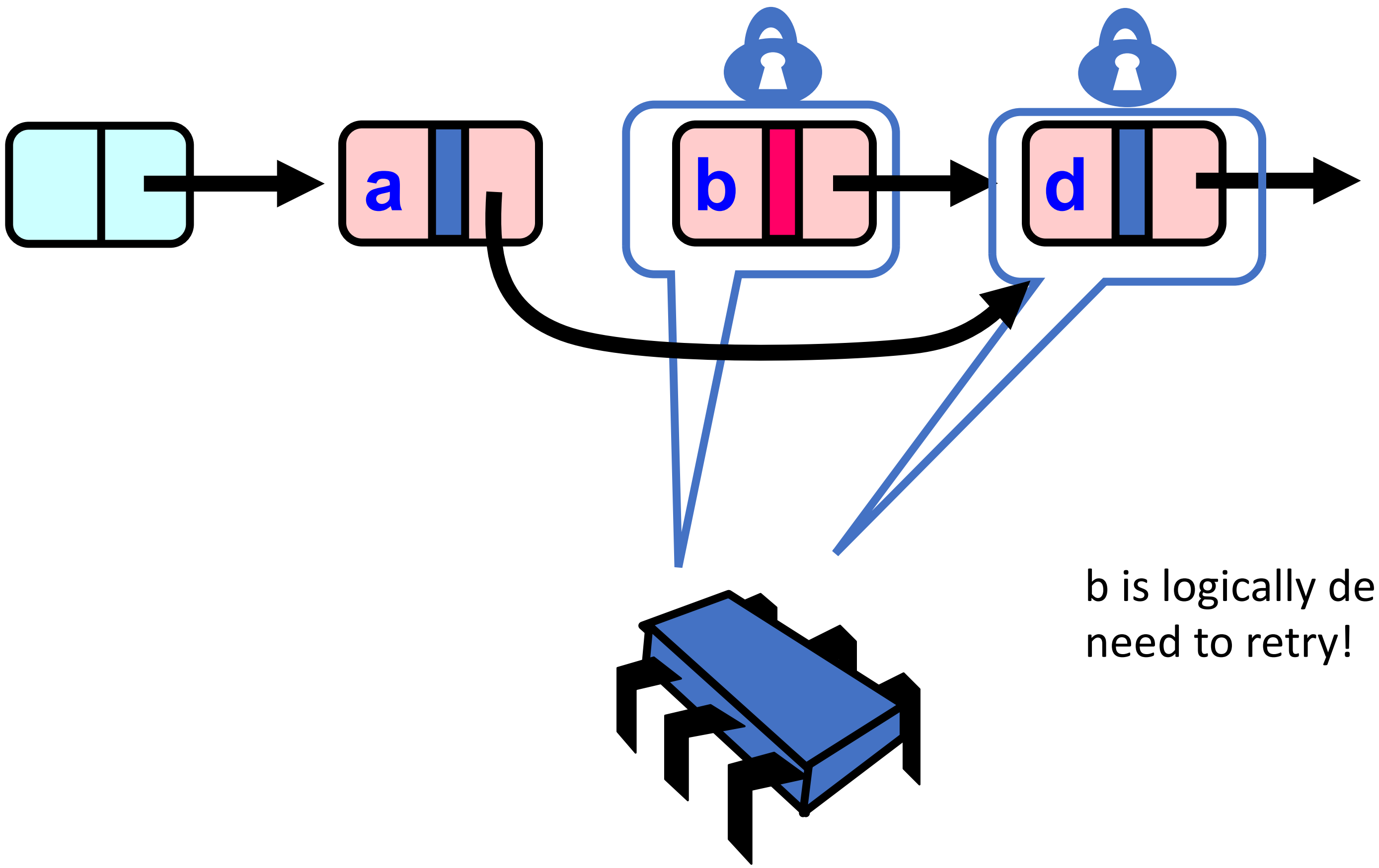
Fixed with logical flag



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!

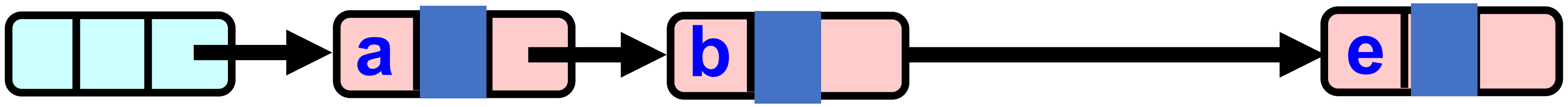
Evaluation

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

Lock-free Lists

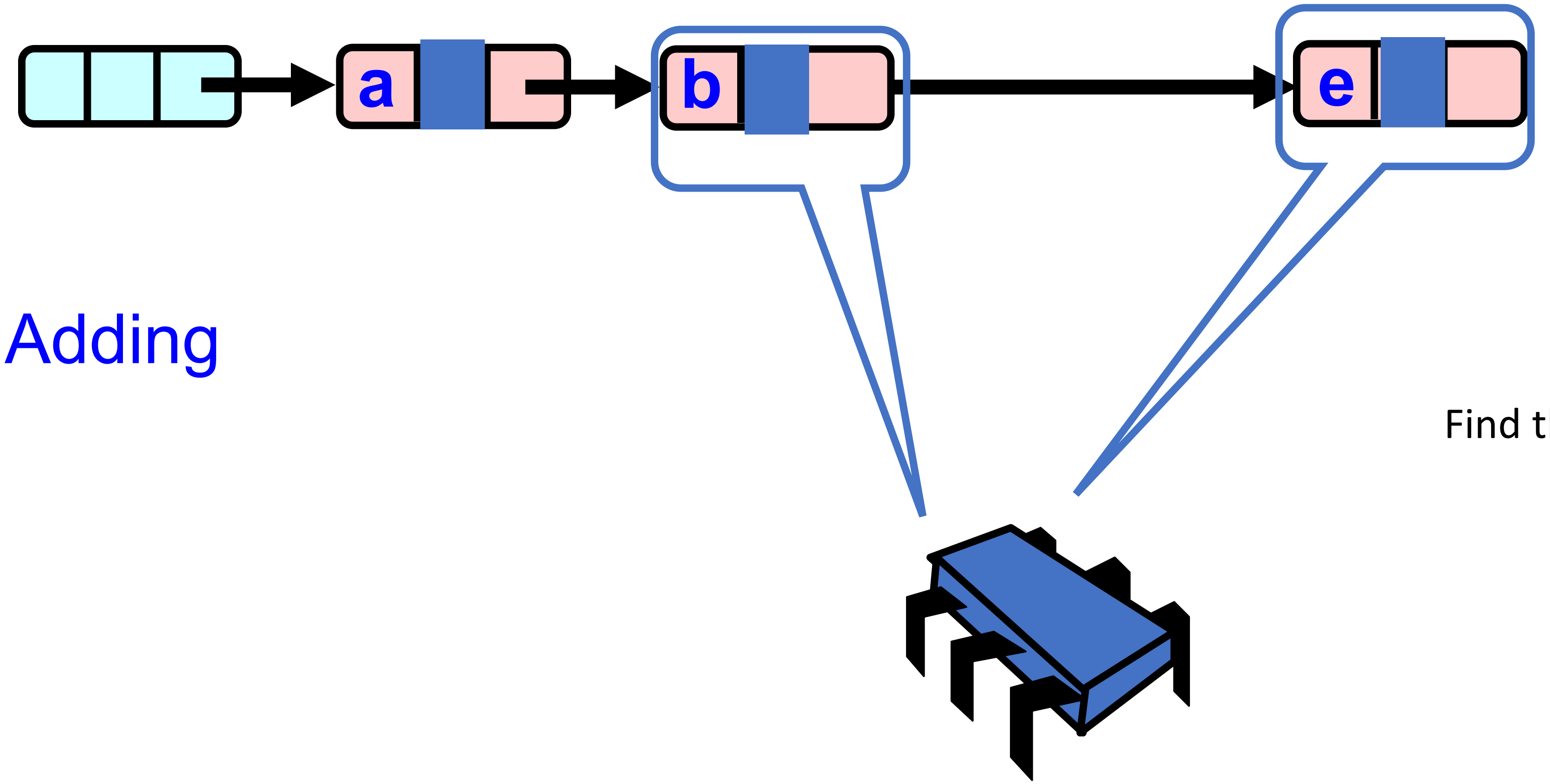
- Next logical step
  - lock-free add() and remove()
- What sort of atomics do we need?
  - Loads/stores?
  - RMWs?

Lock-free Lists



Adding

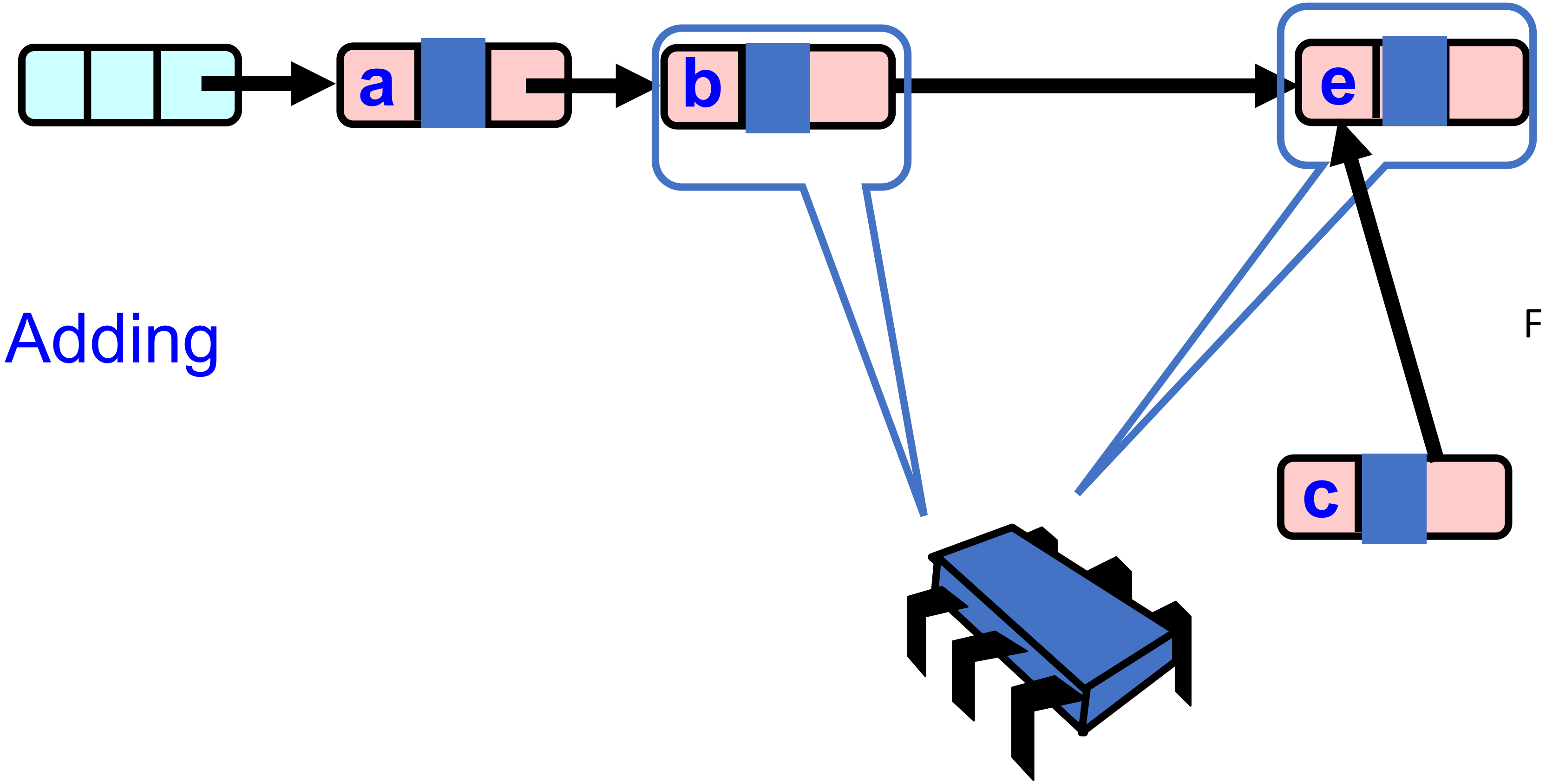
Lock-free Lists



Adding

Find the location

Lock-free Lists

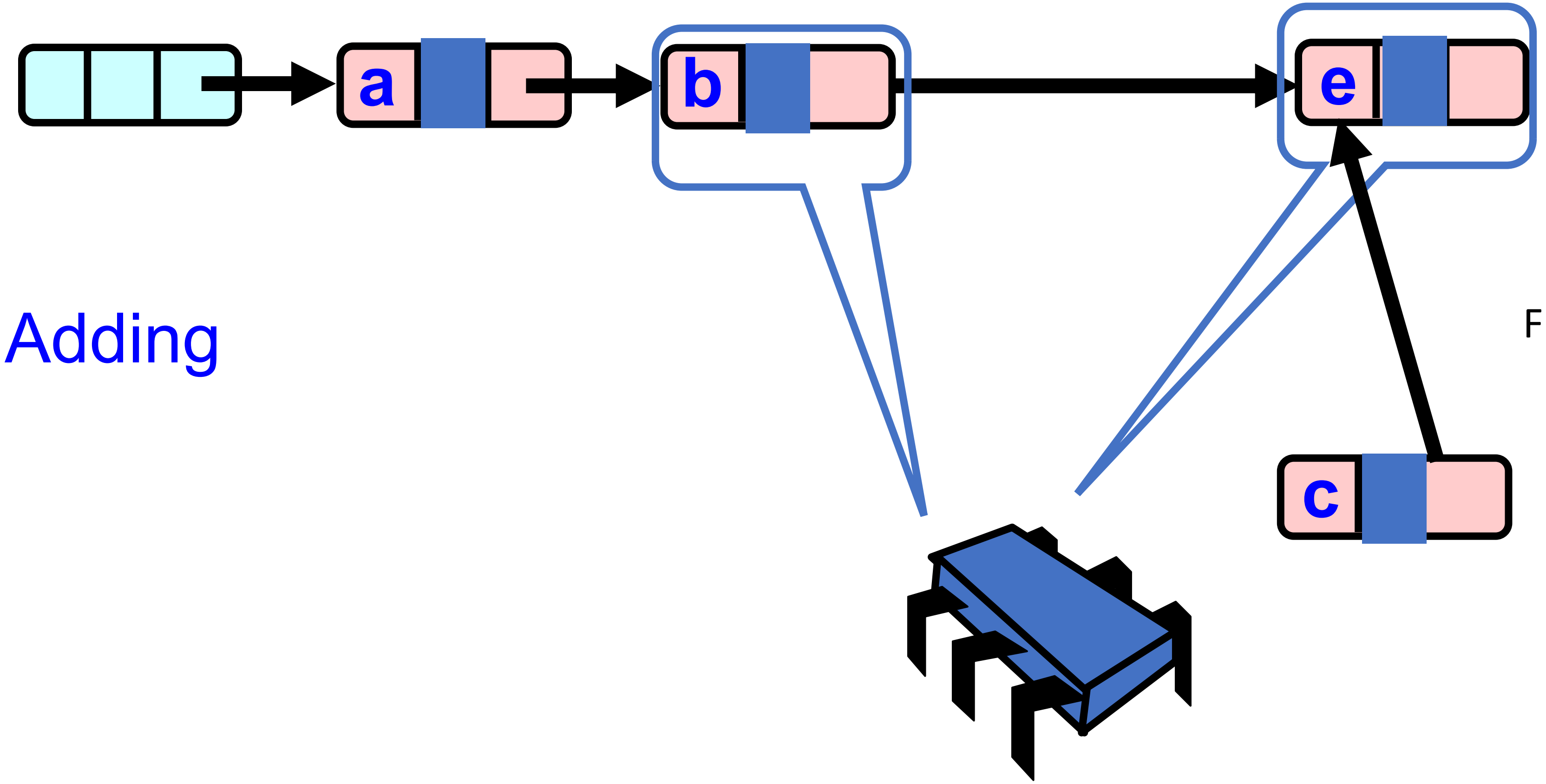


Adding

Find the location

create “c”

Lock-free Lists



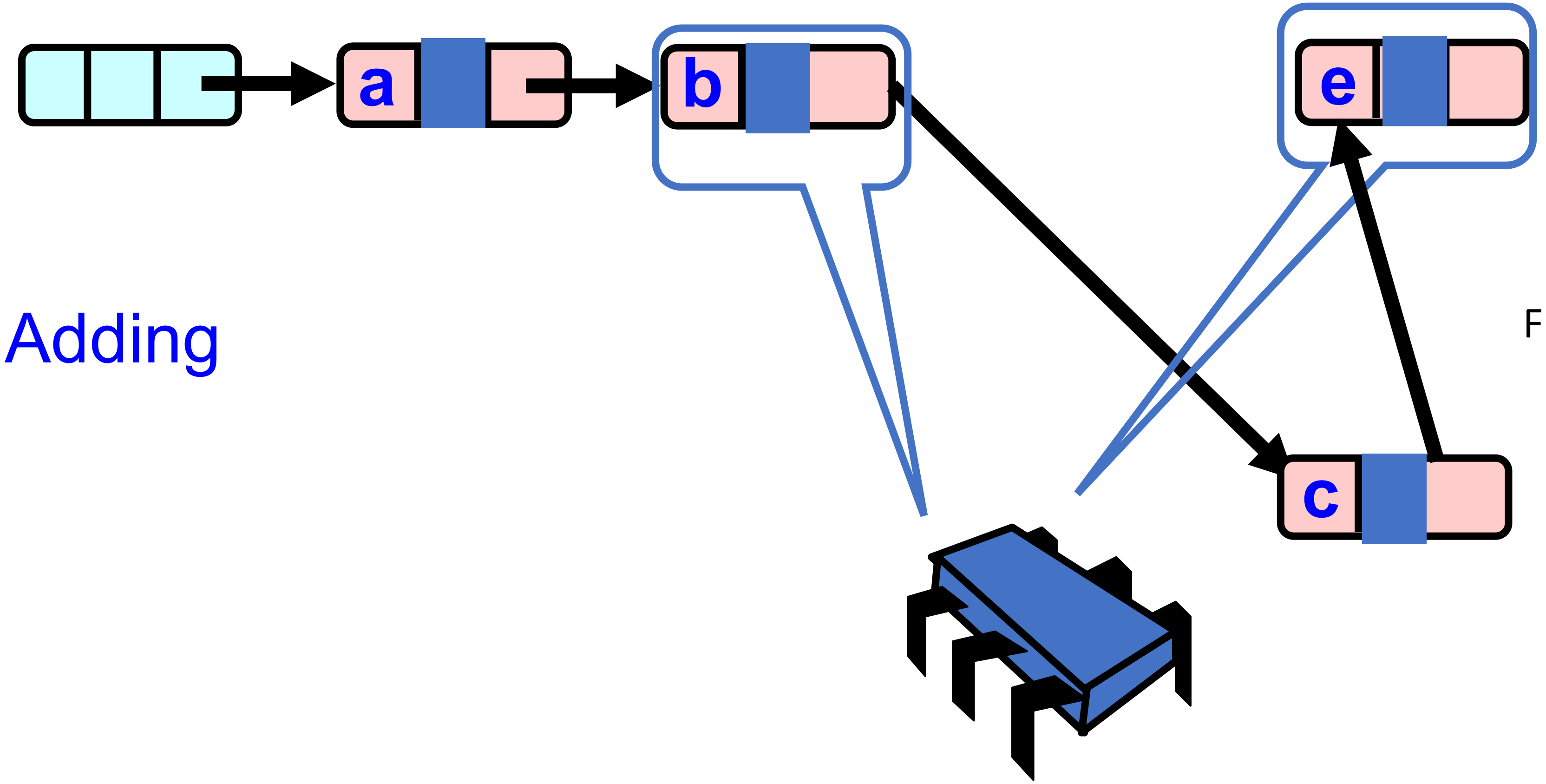
Adding

Find the location

create “c”

insert “c”

Lock-free Lists



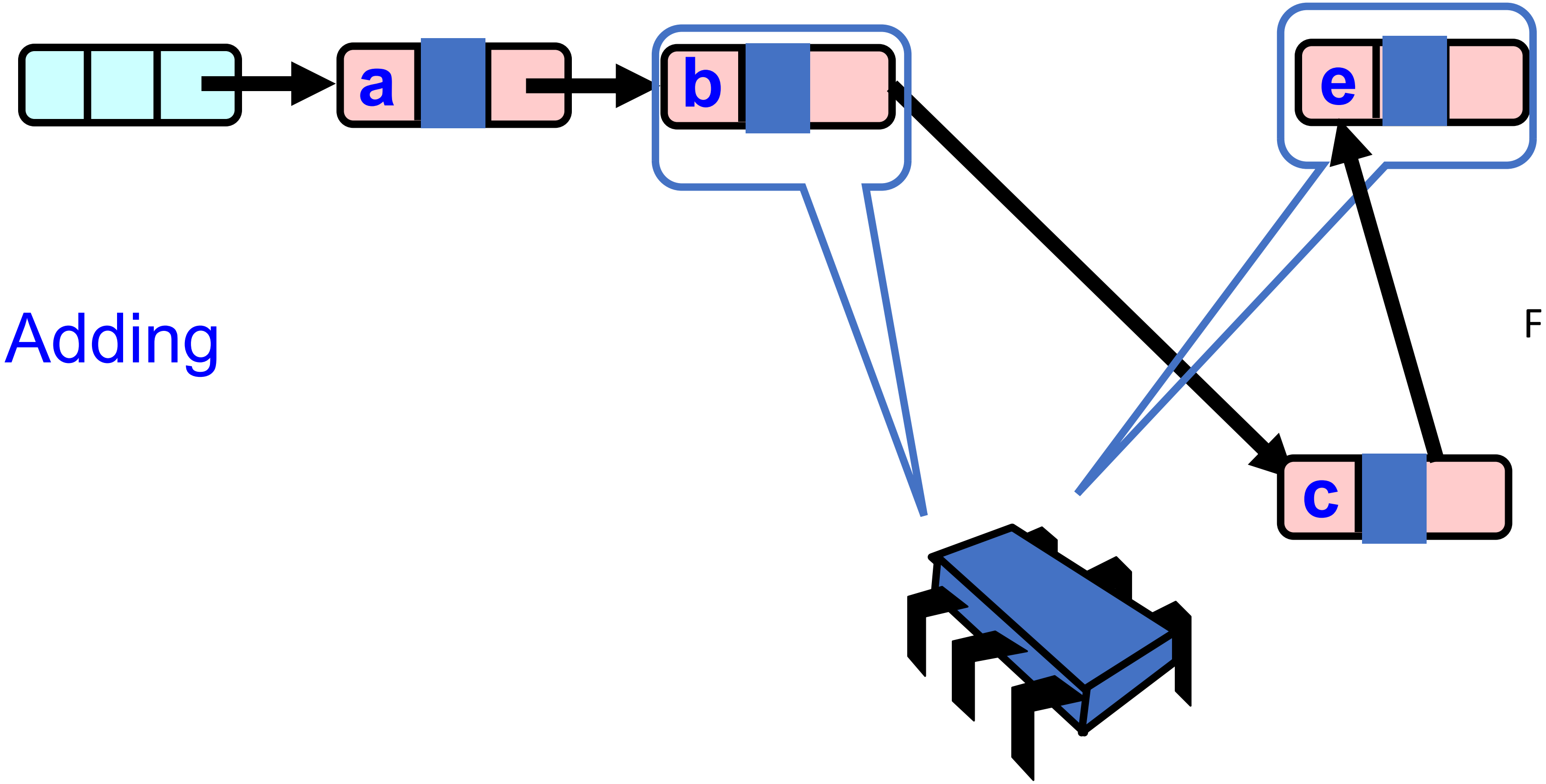
Adding

Find the location

create “c”

insert “c”

Lock-free Lists



Adding

Find the location

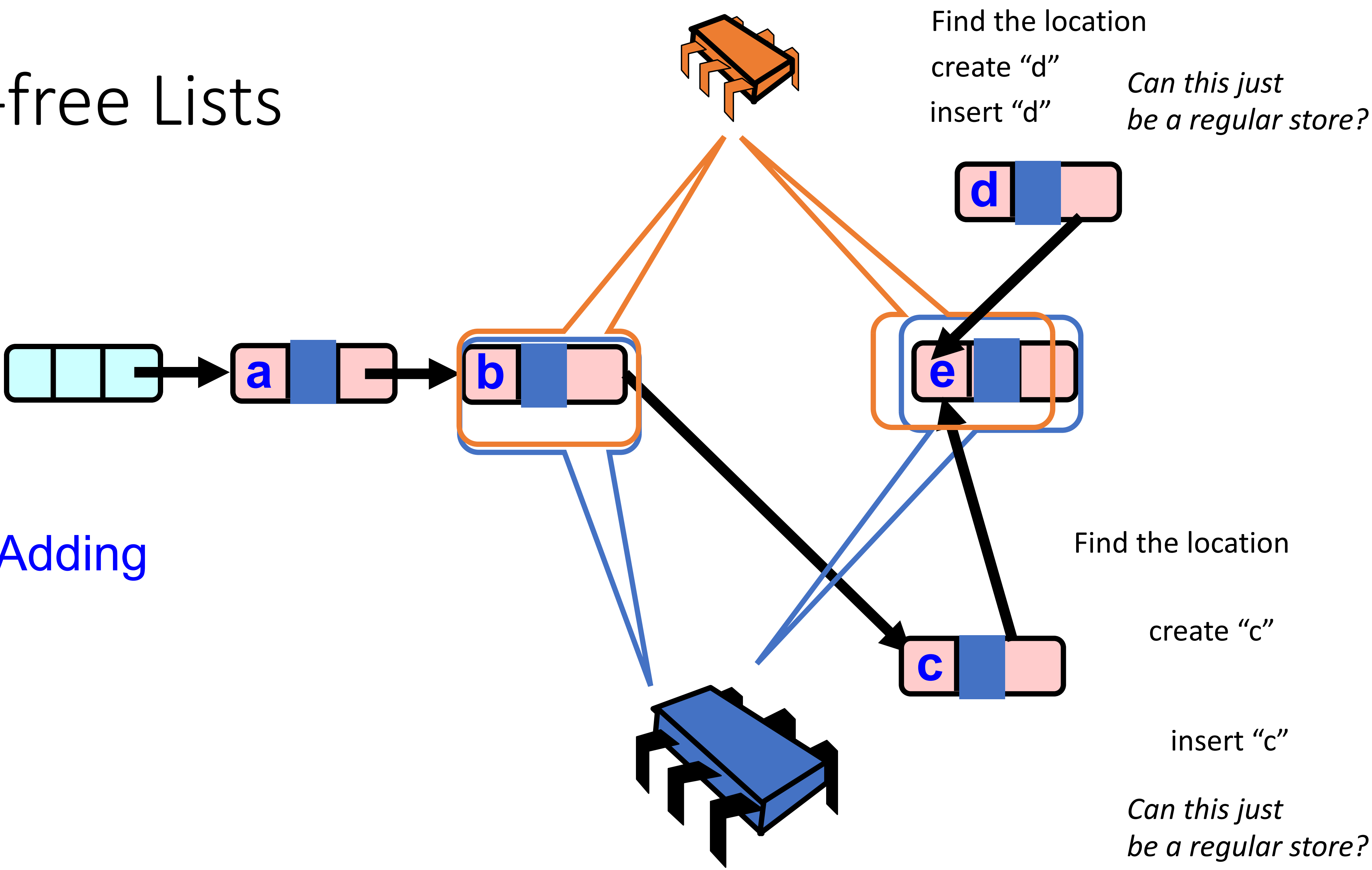
create “c”

insert “c”

Can this just be a regular store?

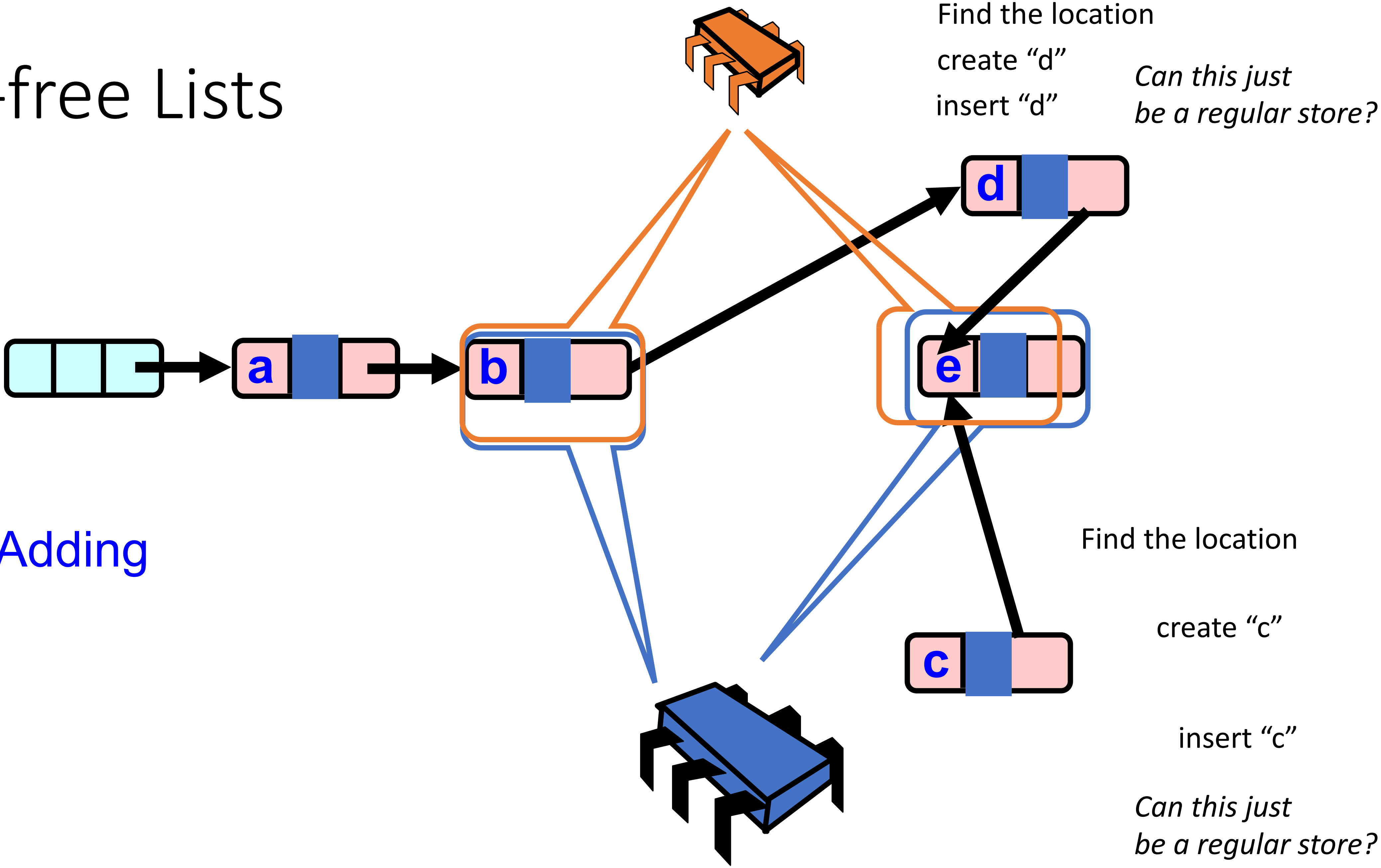
Lock-free Lists

Adding

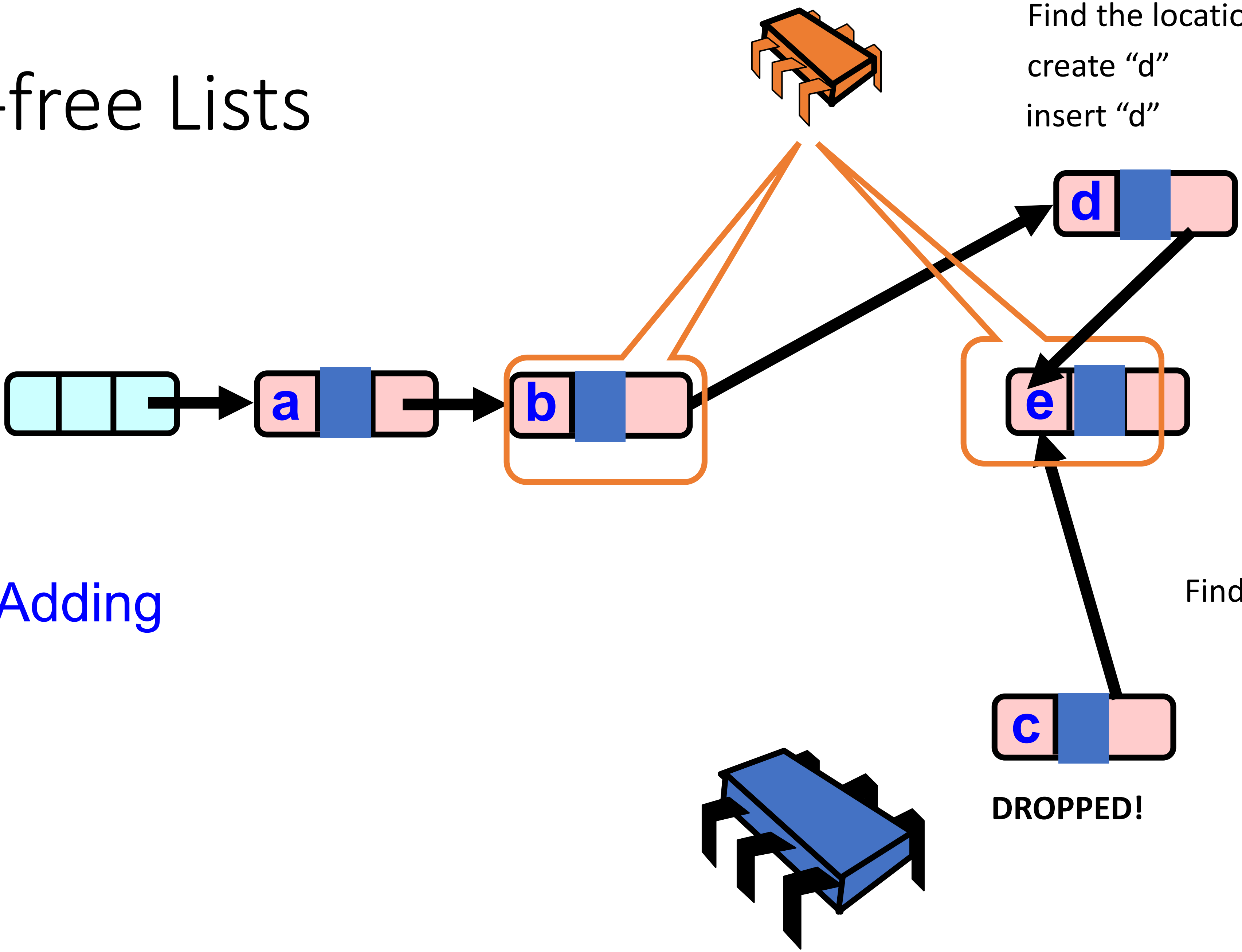


Lock-free Lists

Adding



Lock-free Lists



Find the location

create “d”

insert “d”

Can this just be a regular store?

Adding

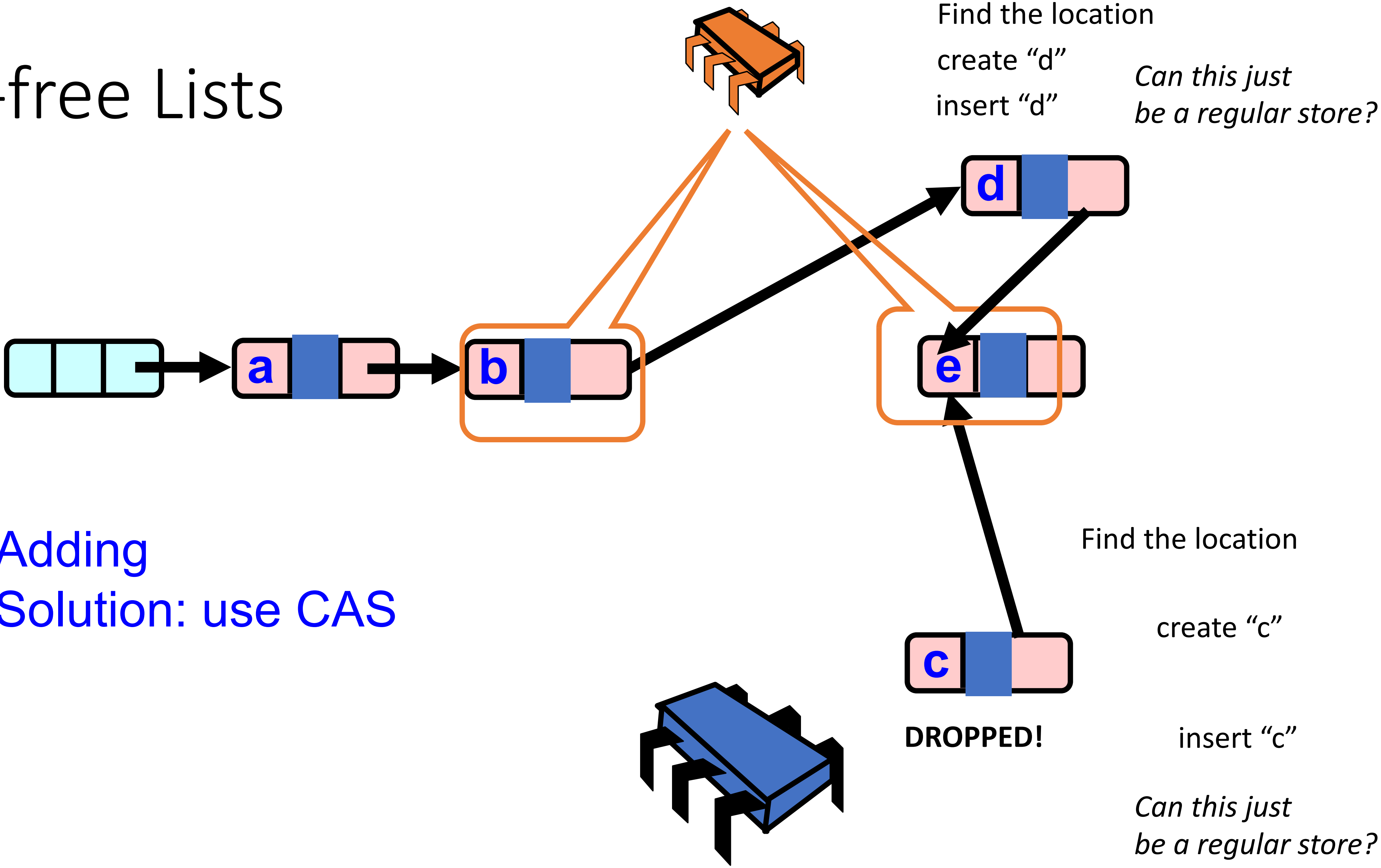
Find the location

create “c”

insert “c”

Can this just be a regular store?

Lock-free Lists



Adding

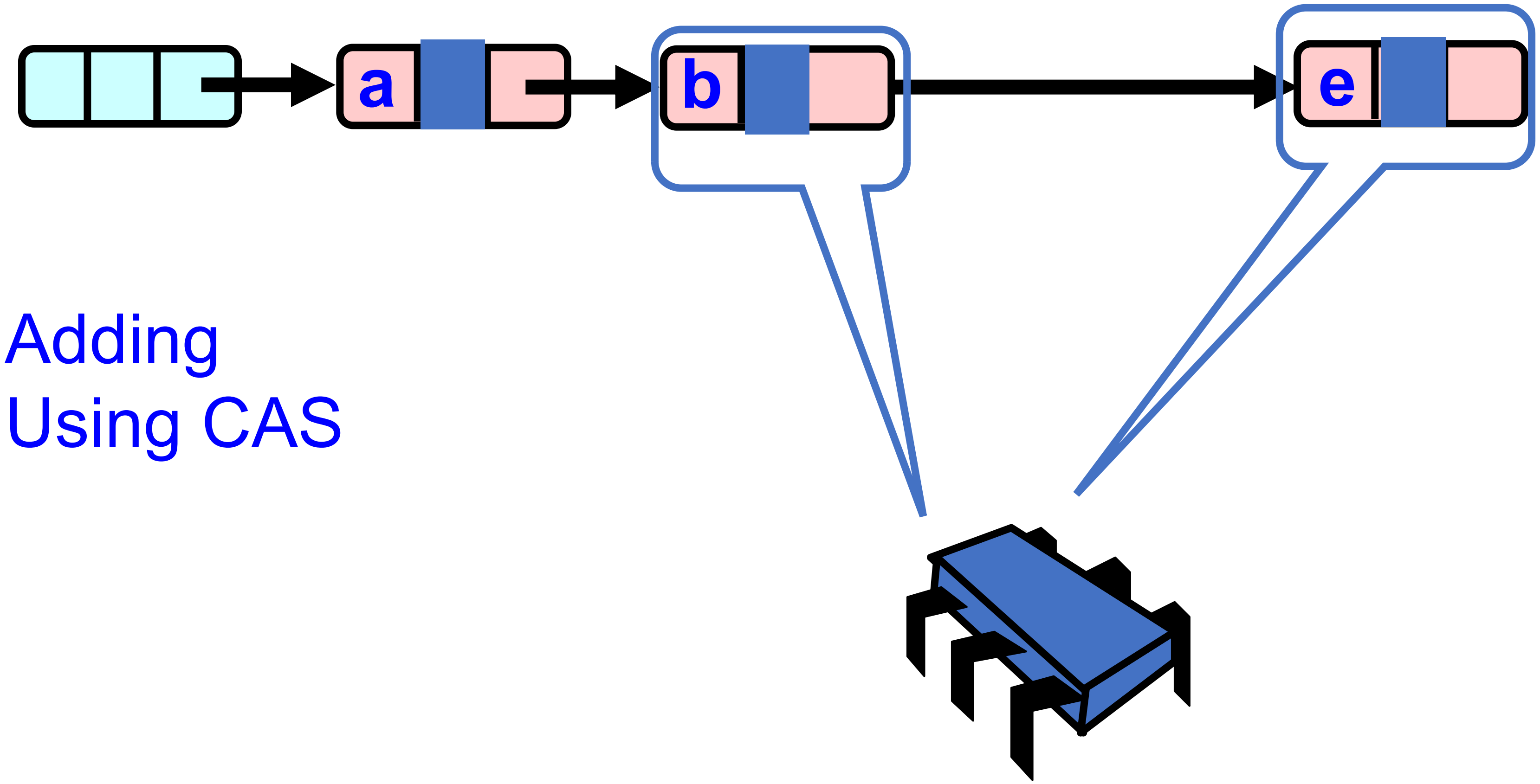
Solution: use CAS

Lock-free Lists

Find the location

Cache your insertion point!

b.next == e



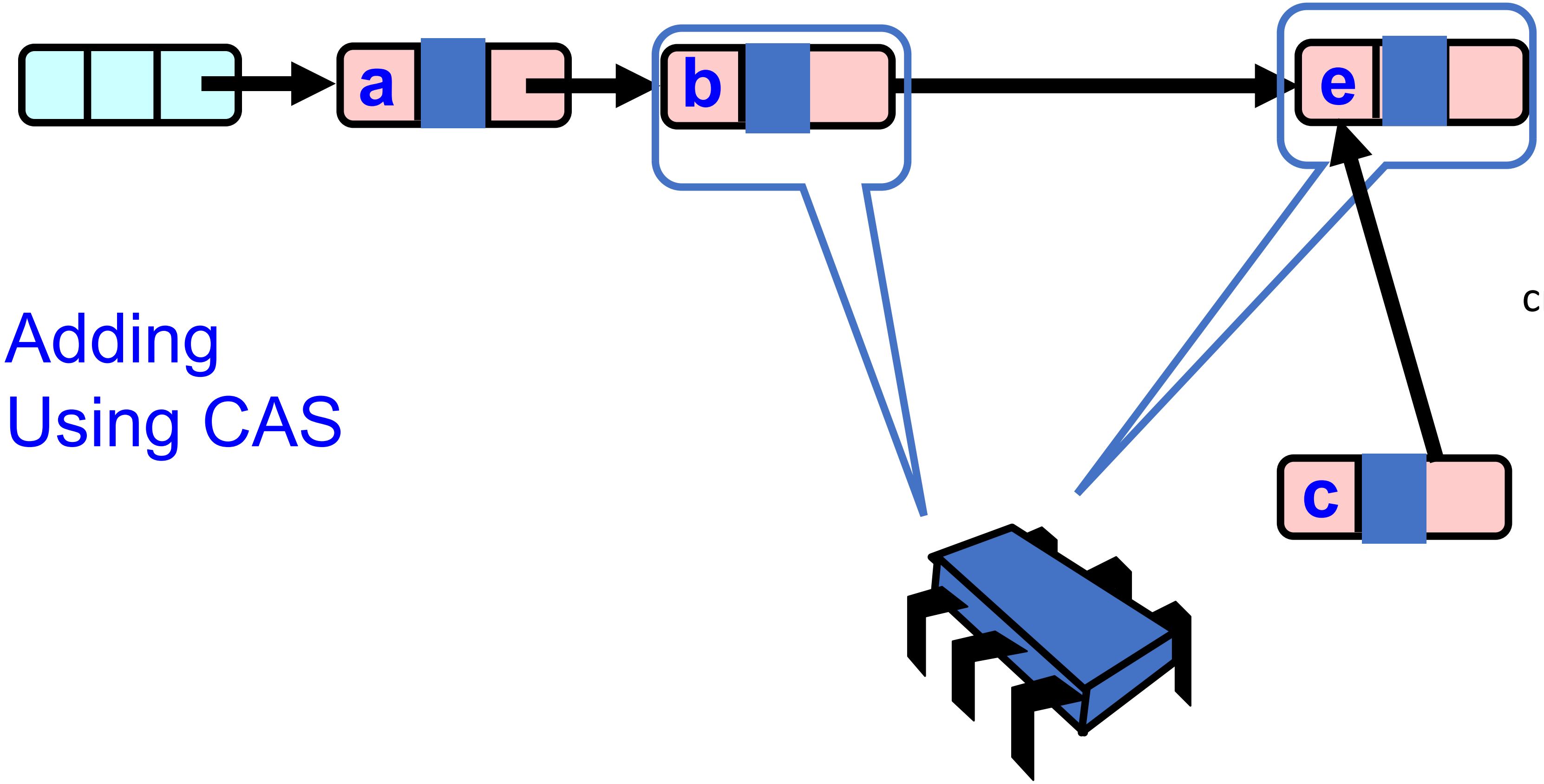
Adding
Using CAS

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!

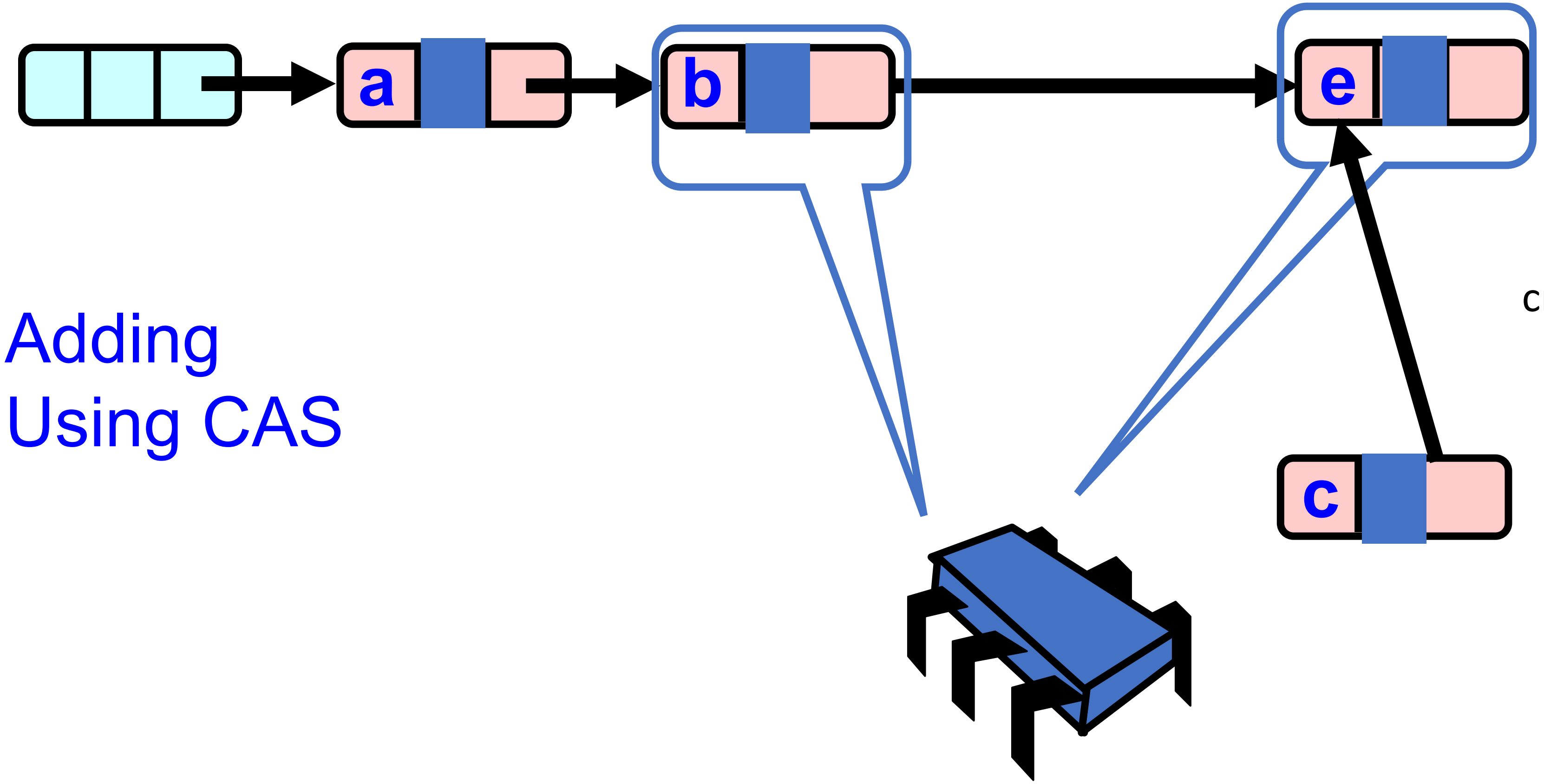
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding Using CAS

create “c”

Lock-free Lists

Only insert if your insertion point is valid!

CAS(b.next, e, c);

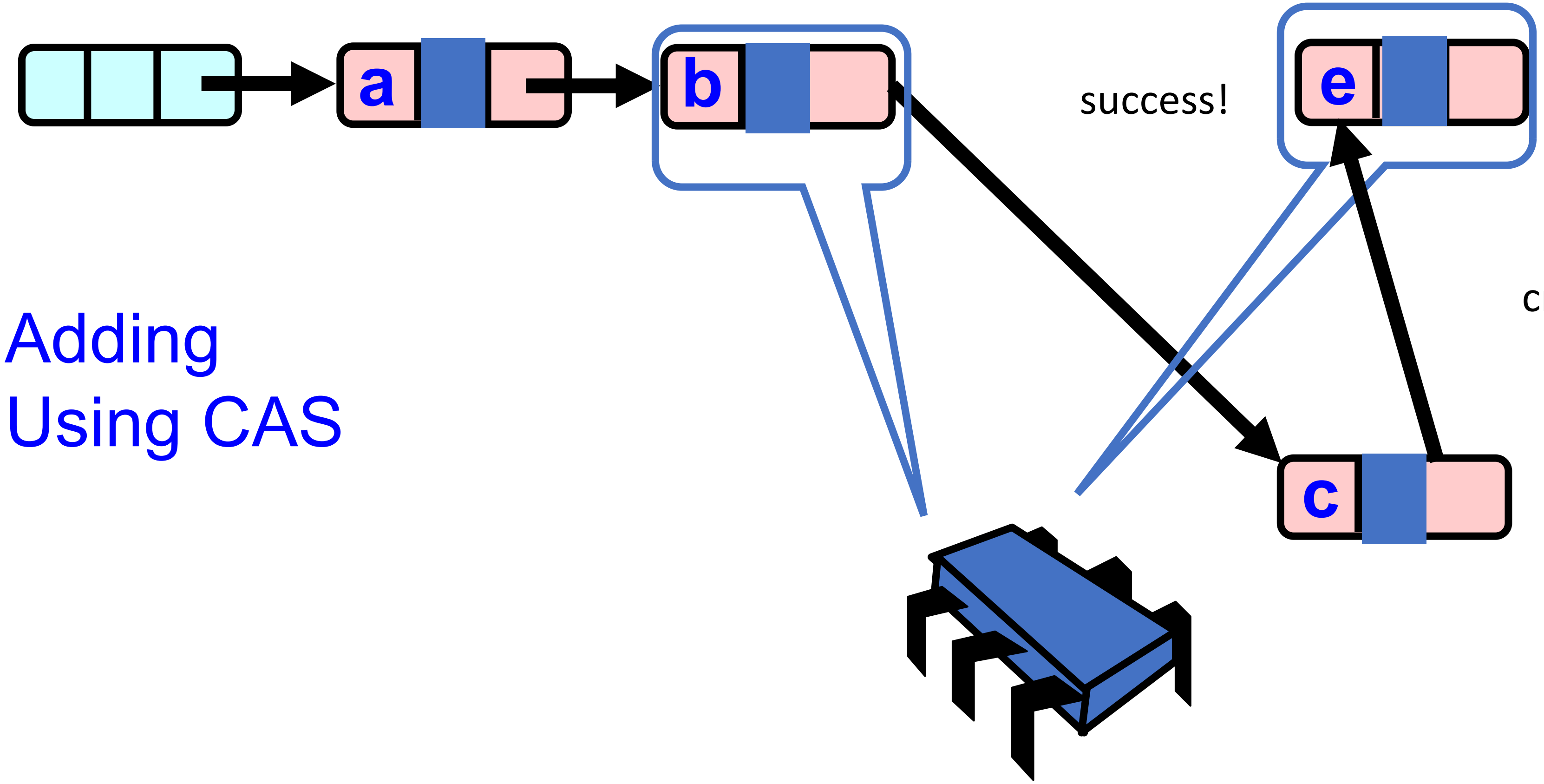
Find the location

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!

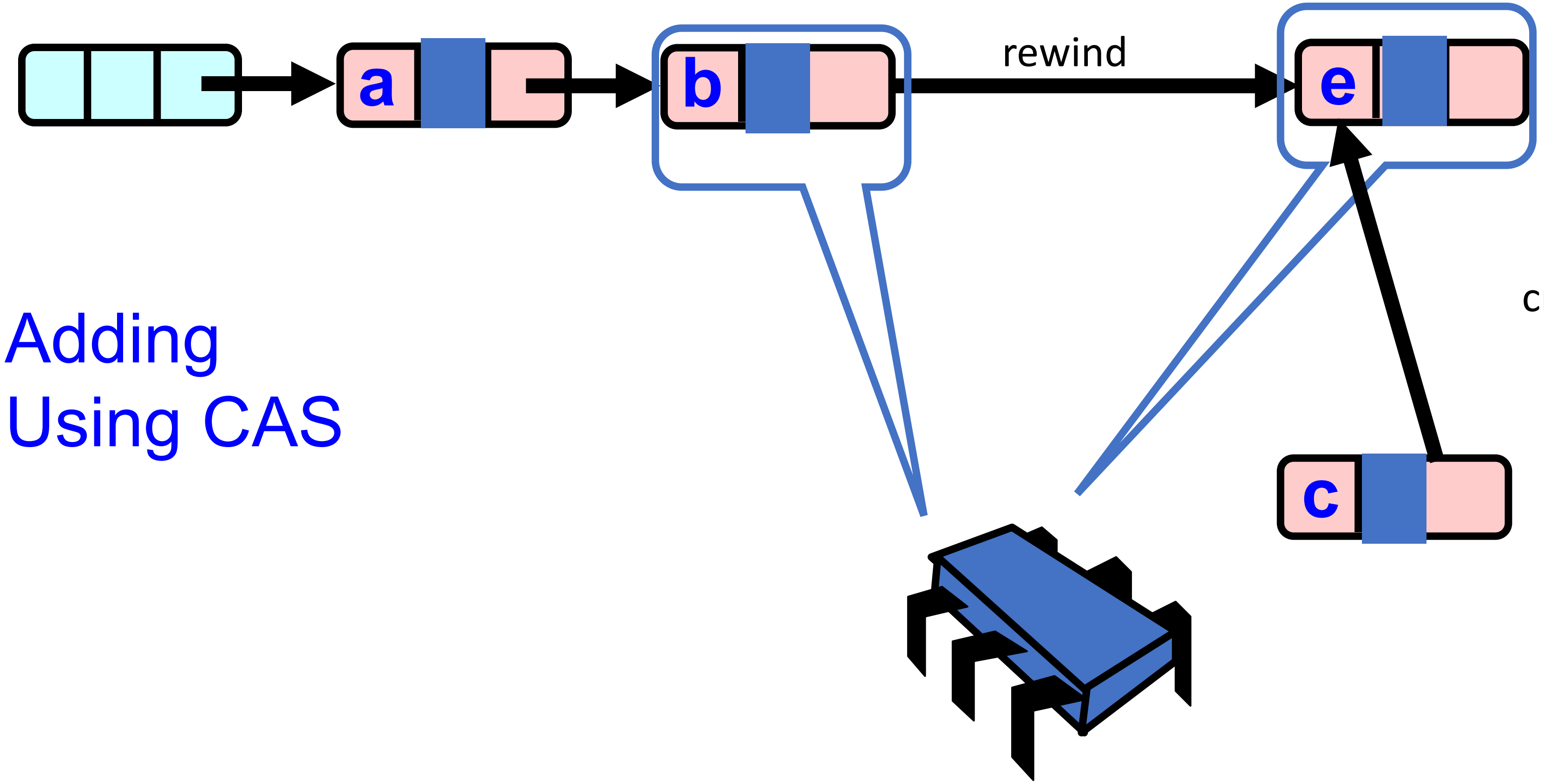
CAS(b.next, e, c);

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);

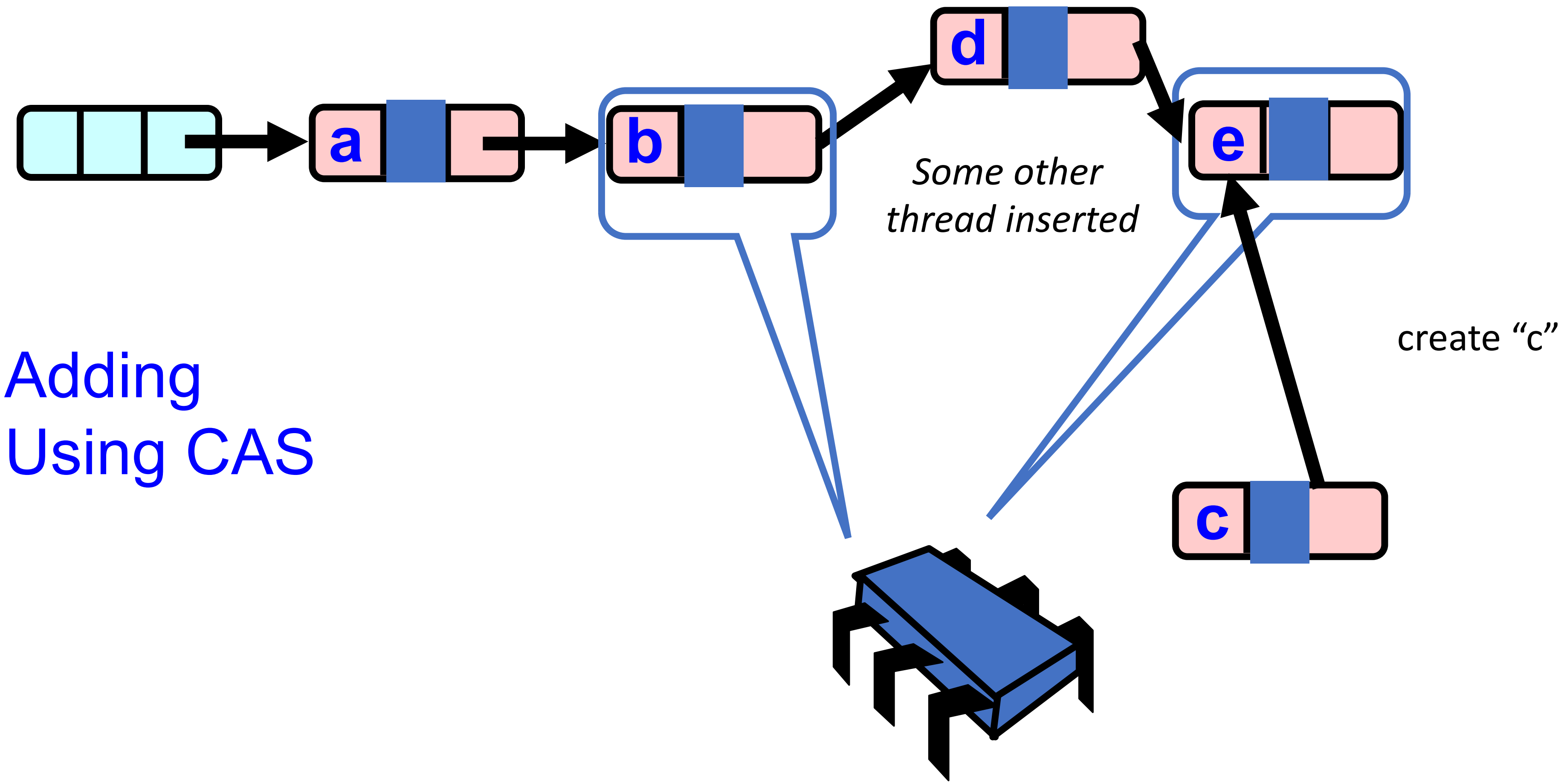
Find the location

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!

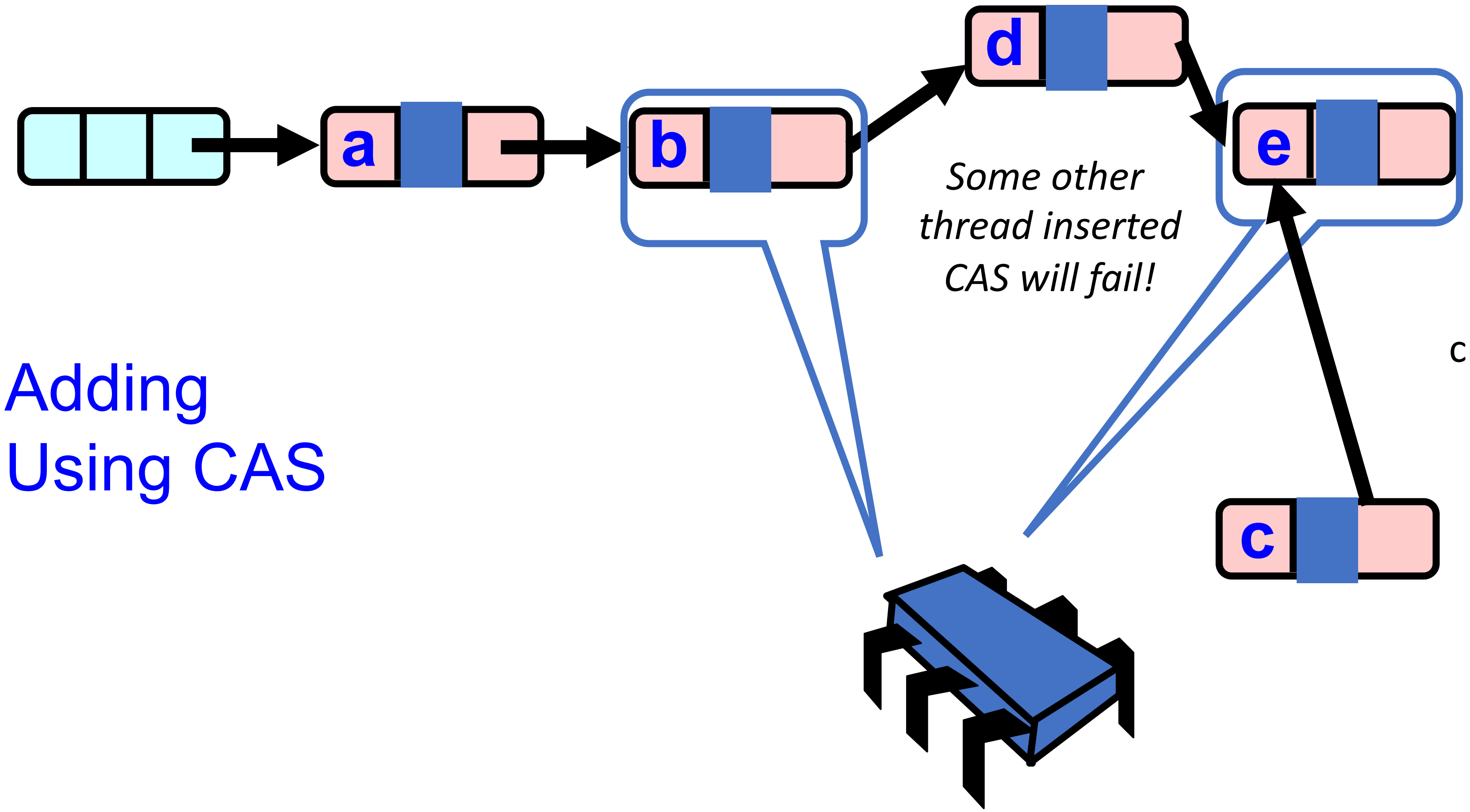
CAS(b.next, e, c);

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);

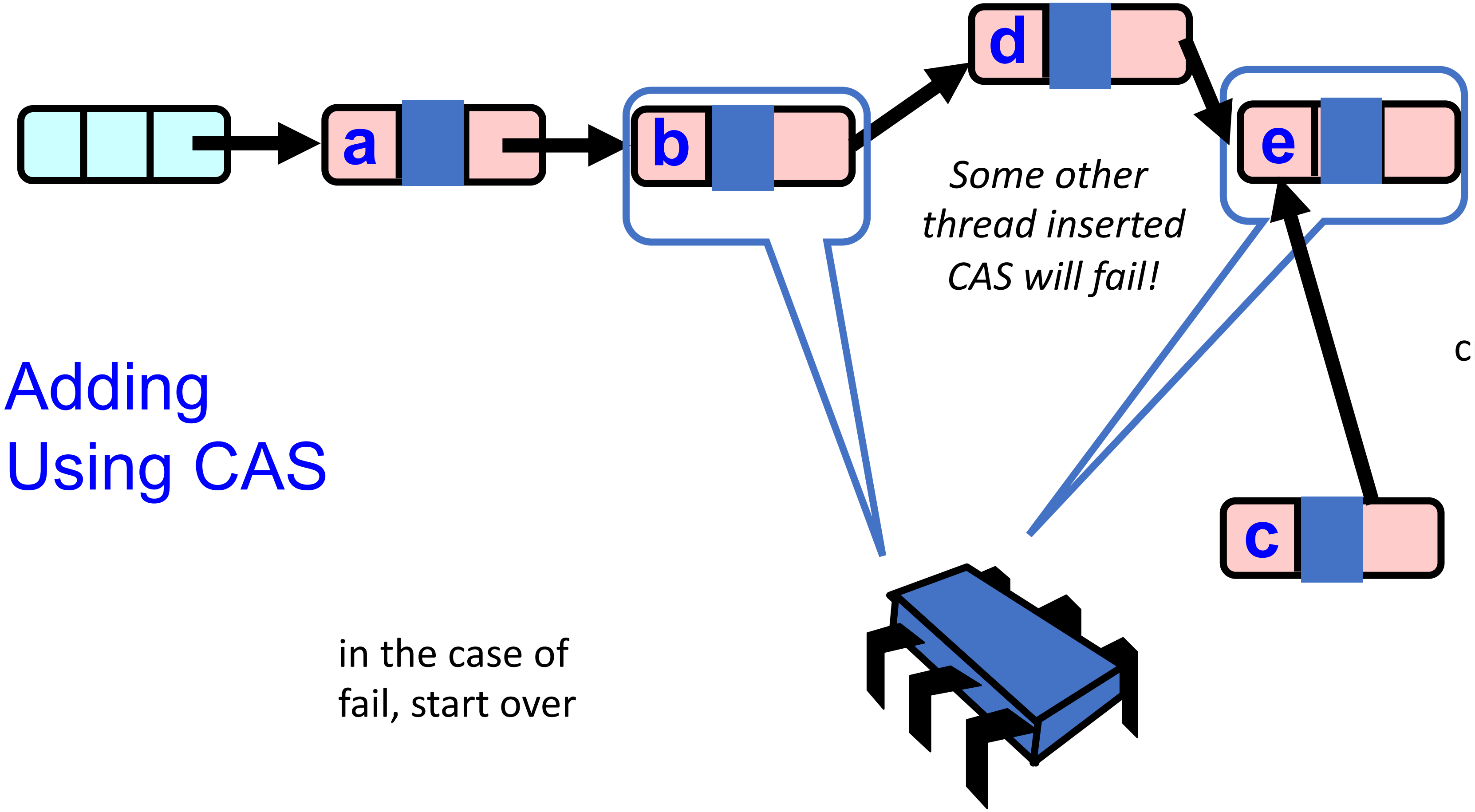
Find the location

Cache your insertion point!

b.next == e

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

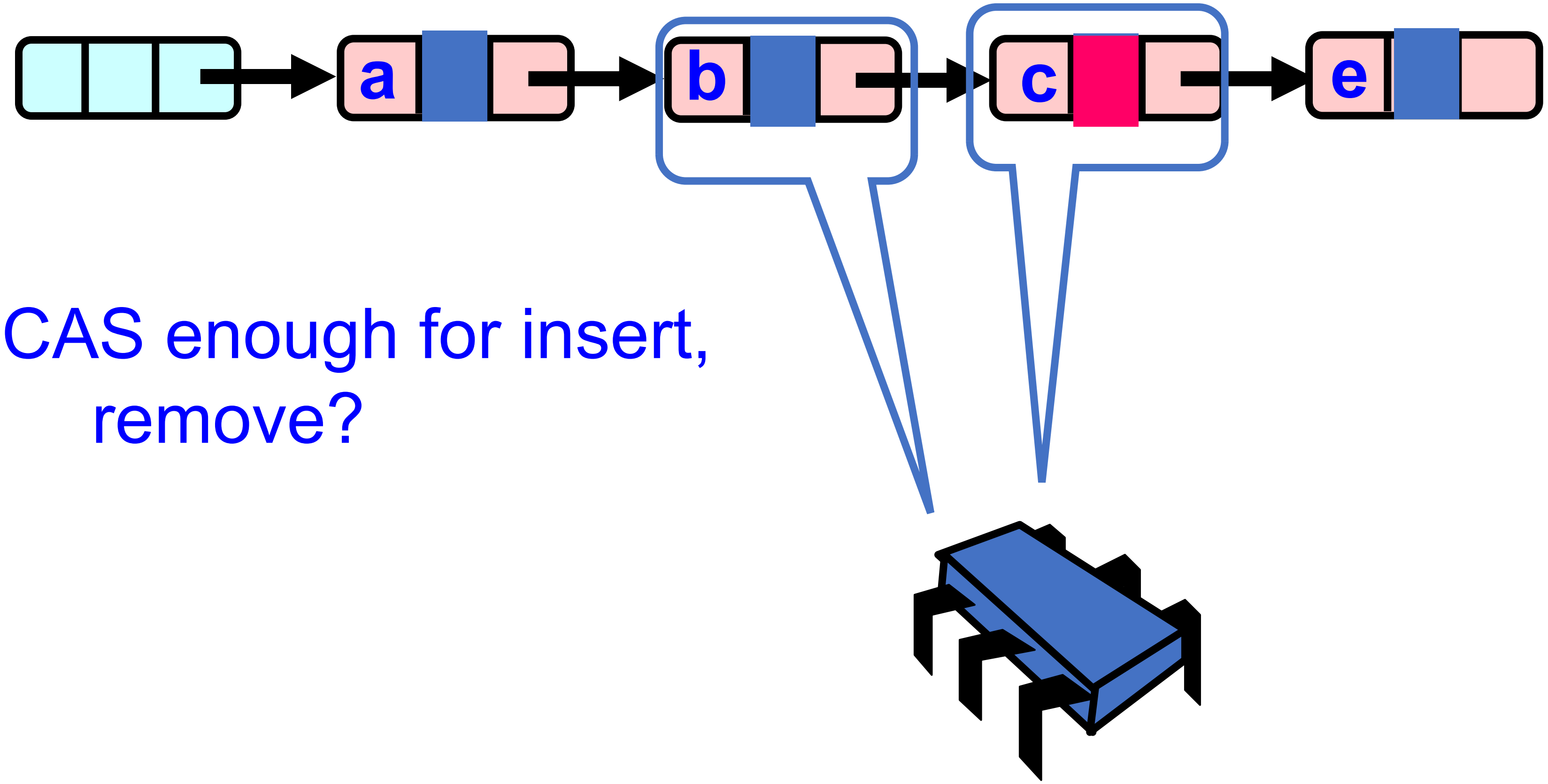
Adding Using CAS



create “c”

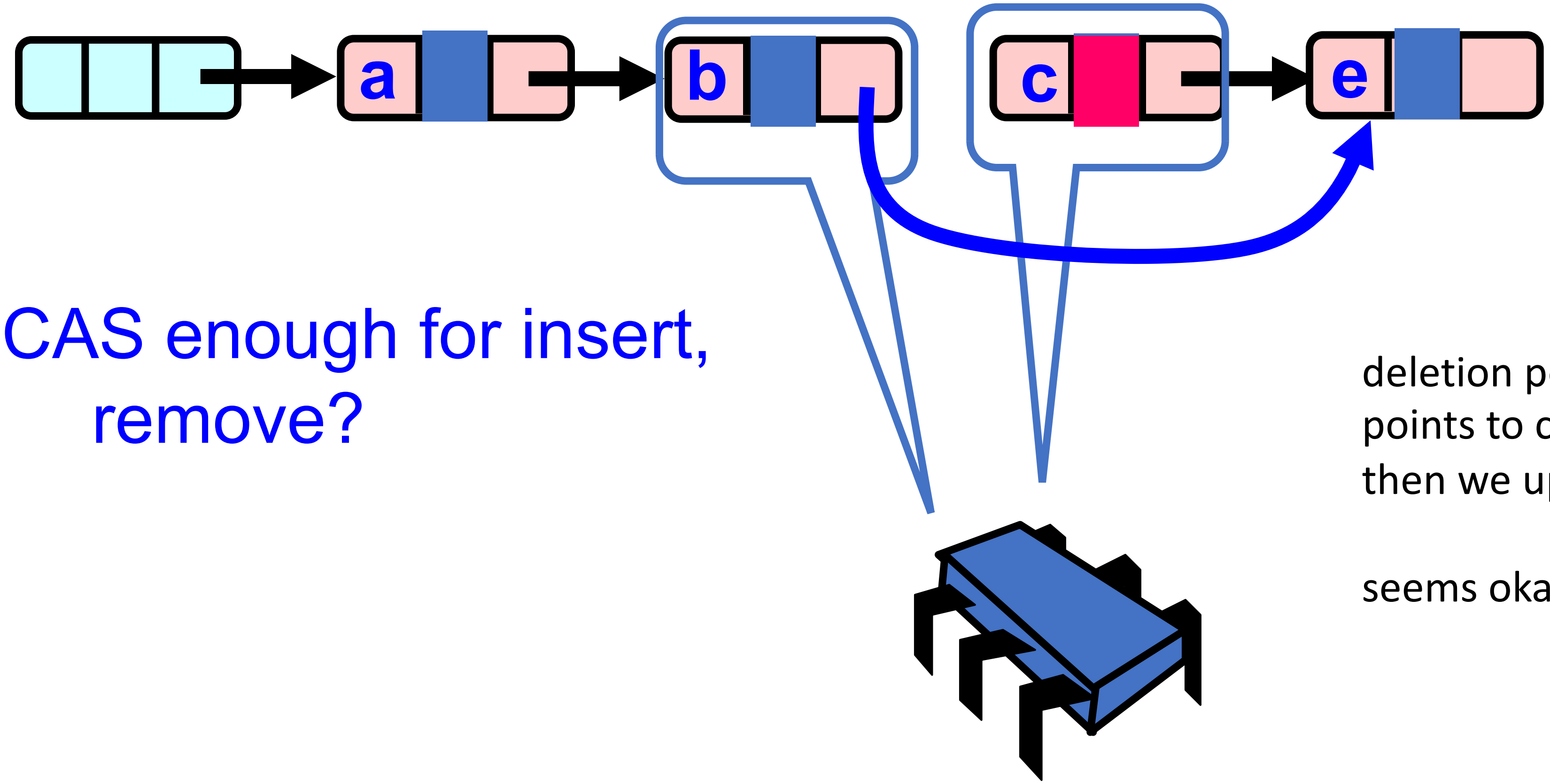
in the case of fail, start over

Lock-free Lists



CAS enough for insert, remove?

Lock-free Lists



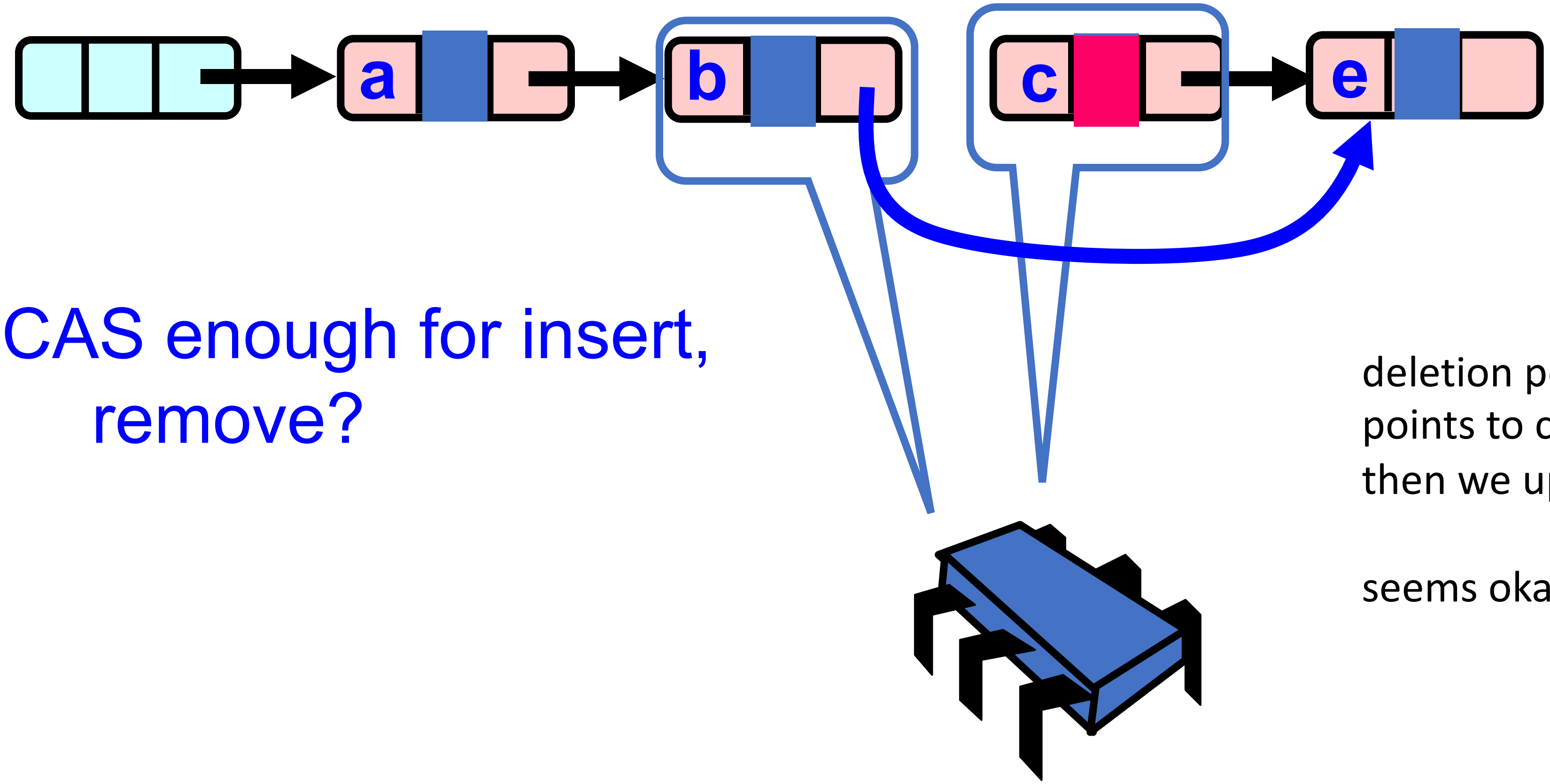
CAS enough for insert, remove?

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



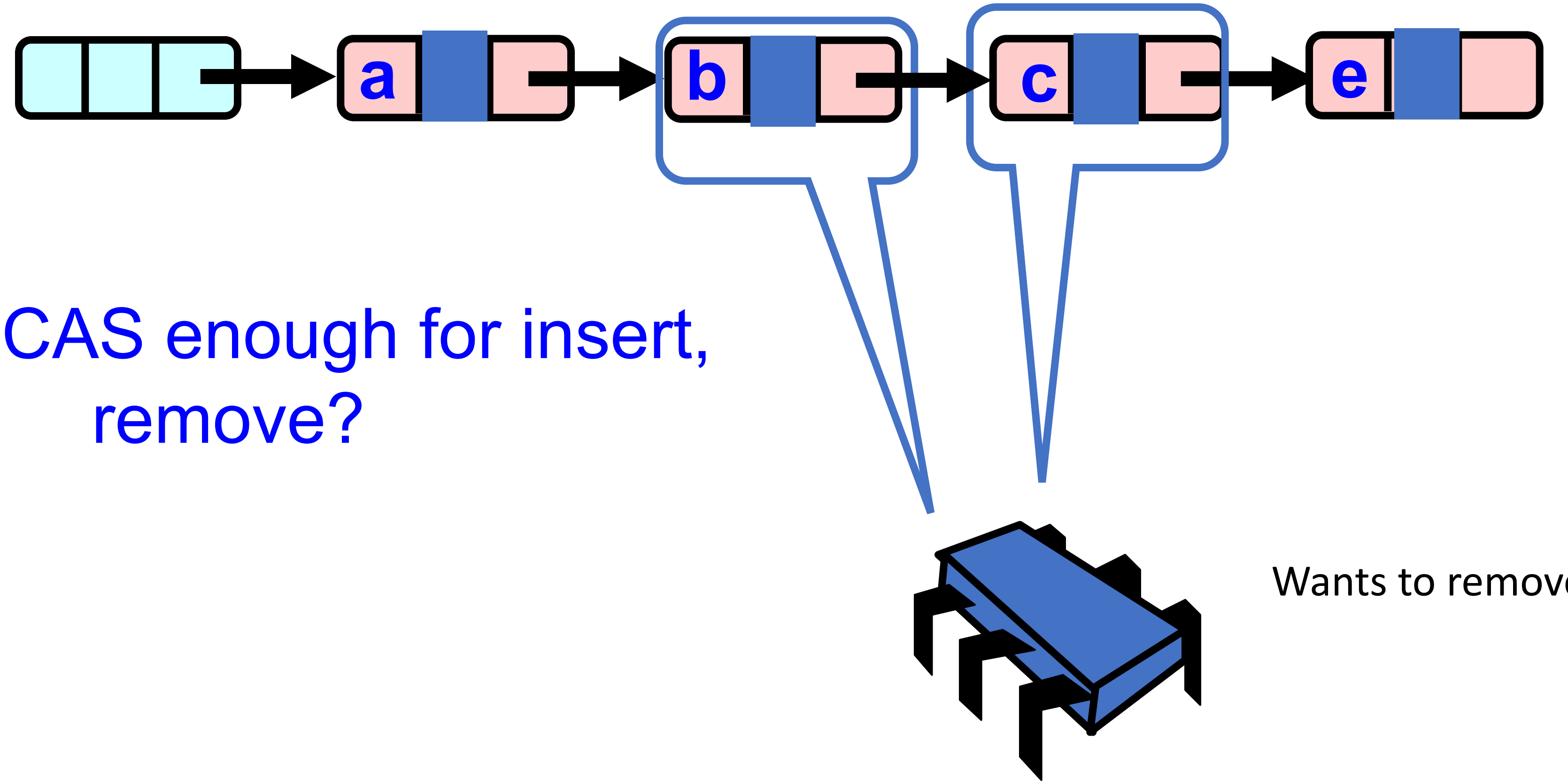
CAS enough for insert, remove?

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

seems okay...

Lock-free Lists

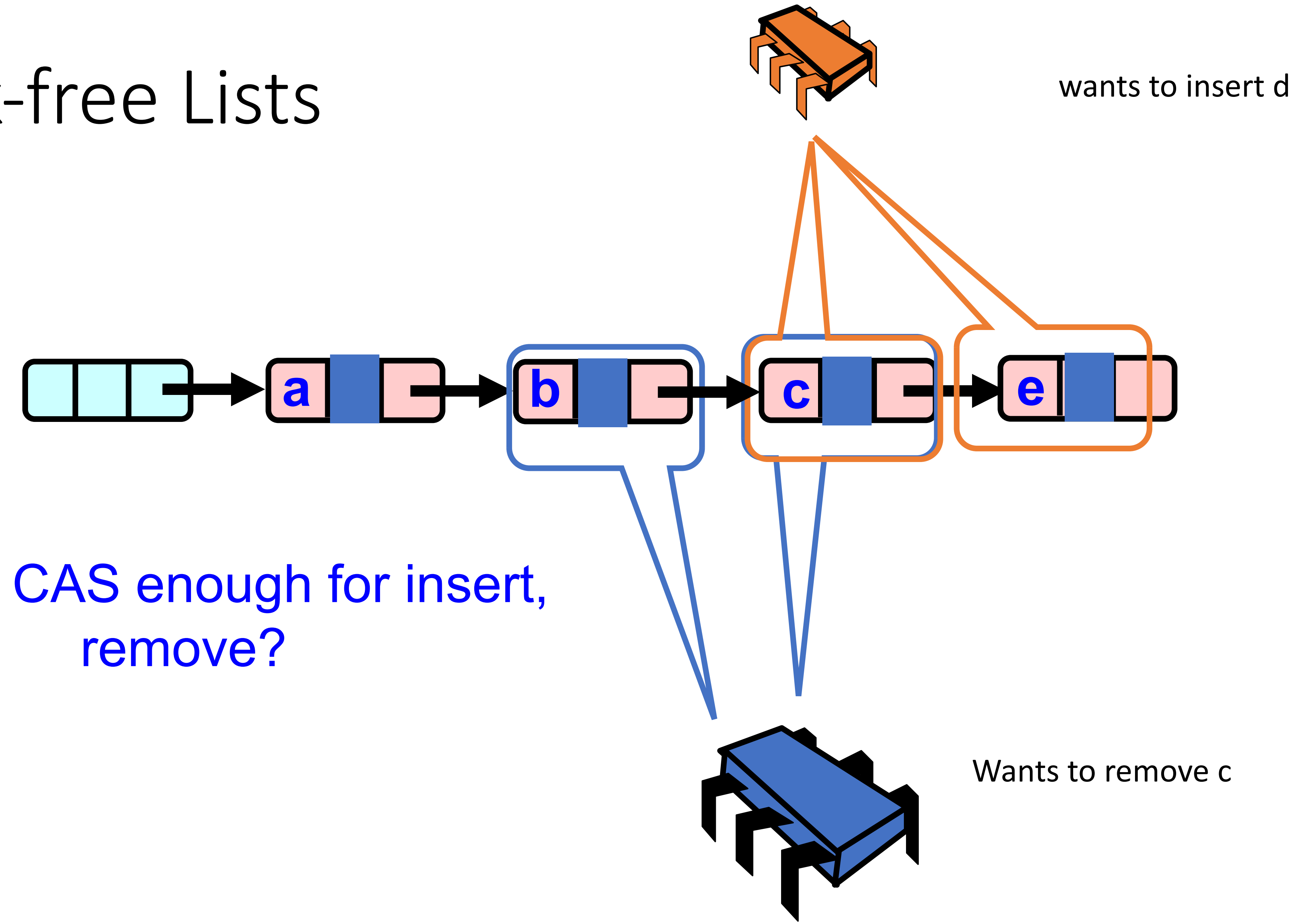
Rewind



CAS enough for insert, remove?

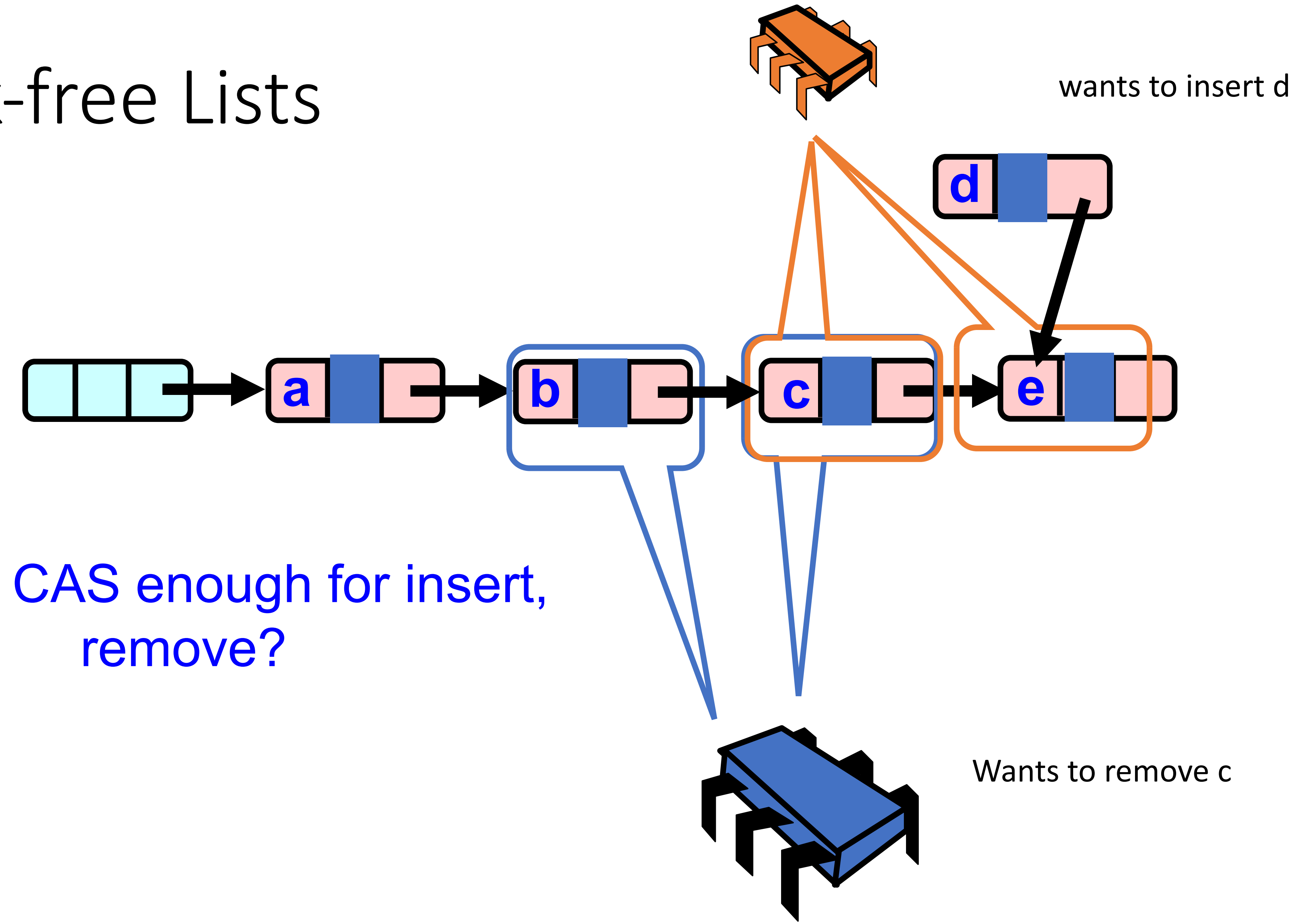
Wants to remove c

Lock-free Lists



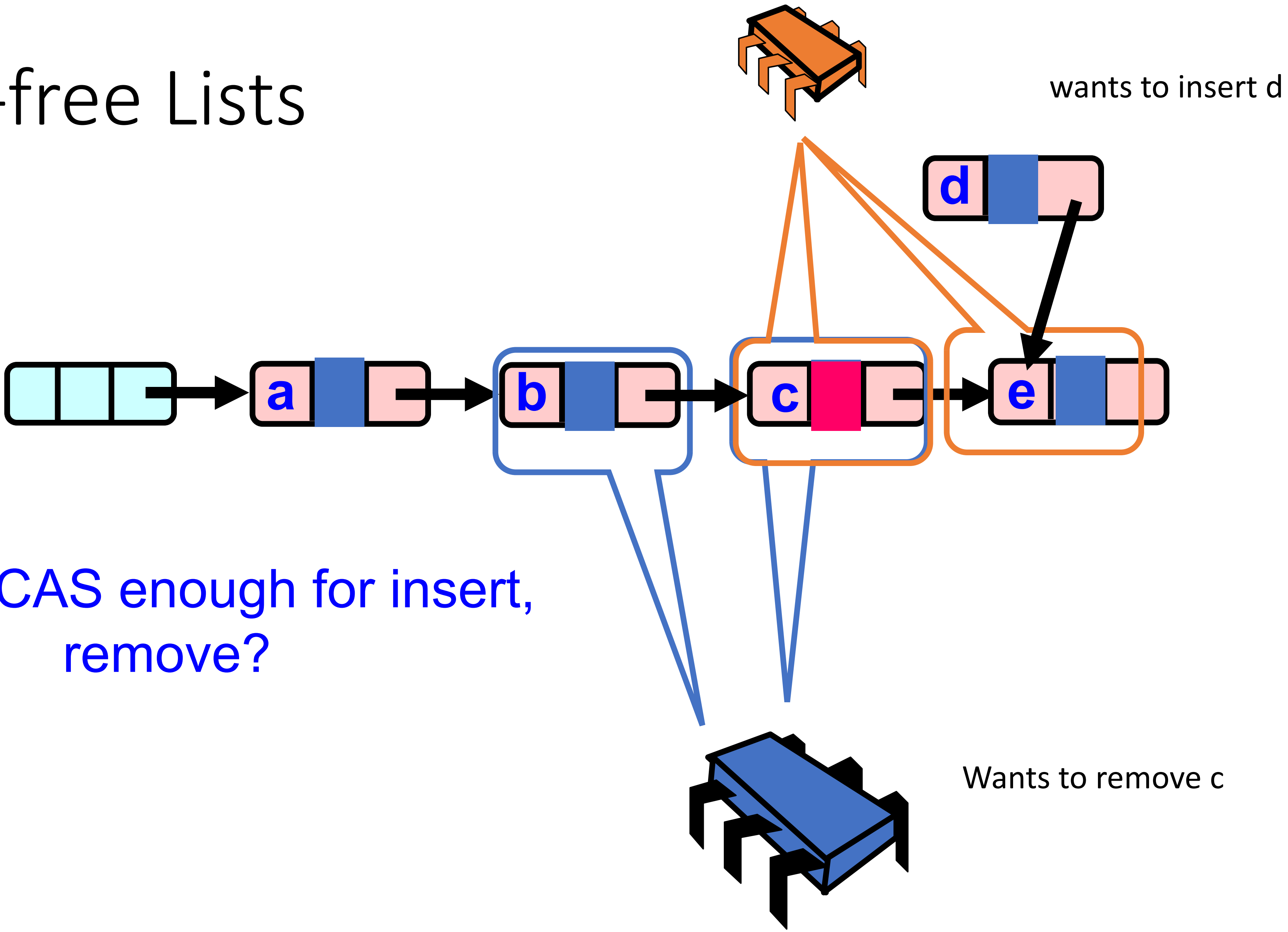
CAS enough for insert, remove?

Lock-free Lists



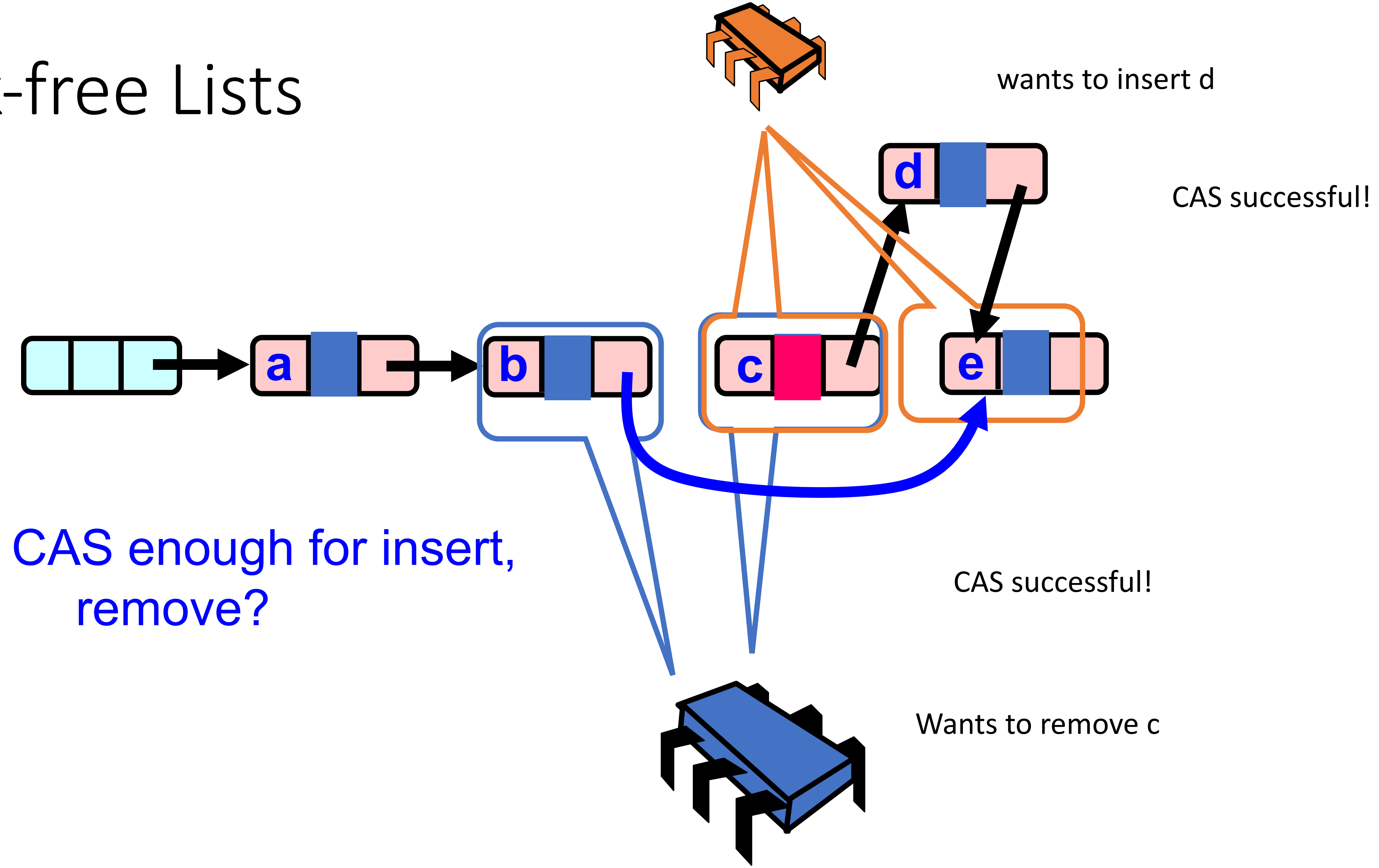
CAS enough for insert, remove?

Lock-free Lists

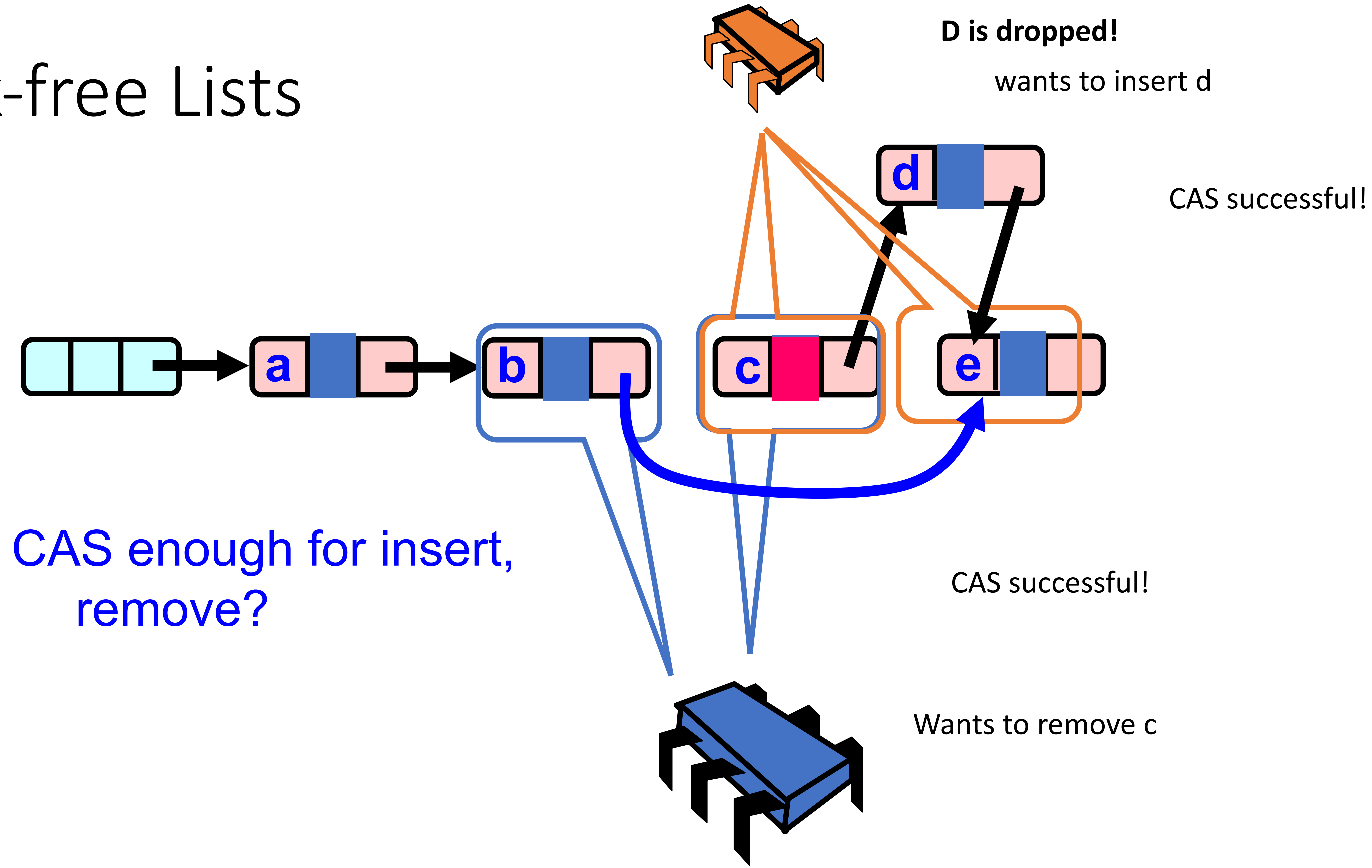


CAS enough for insert, remove?

Lock-free Lists



Lock-free Lists

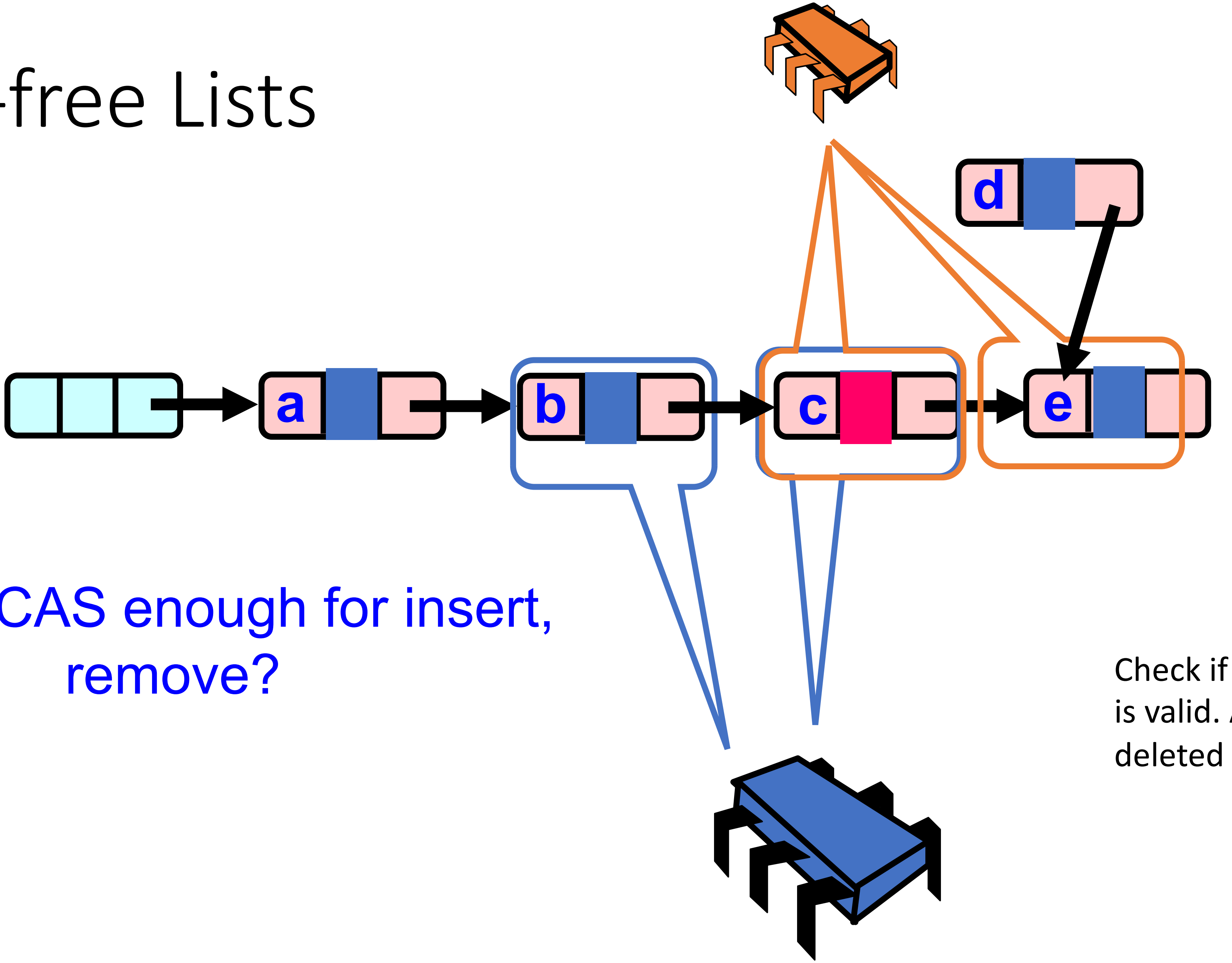


Solution

- 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 if the node has not been logically removed.

Lock-free Lists

wants to insert d



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

Wants to remove c

Marking a Node

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

```
class AtomicMarkedNodePtr {
  private:
    atomic<node *> ptr;
  public:
     AtomicMarkedNodePtr(node *p) {
        node * marked = p | 1;
        ptr.store(marked);
     }

     void logically_delete() {
        // how to store the marked bit atomically?
     }

     node * get_ptr() {
        return ptr.load() & (~1);
     }

     bool CAS (node *e, node *n) {
        node * expected = e | 1;
        node * new_node = n | 1;
        return atomic_compare_exchange(&ptr, &e, new_node);

     }
}
```

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!)

This stuff is tricky

- In this class, you won’t be asked to implement anything this tricky from scratch!

See you on Wedensday!

- Starting on module 4

- Get your midterm in!

- Work on HW 3