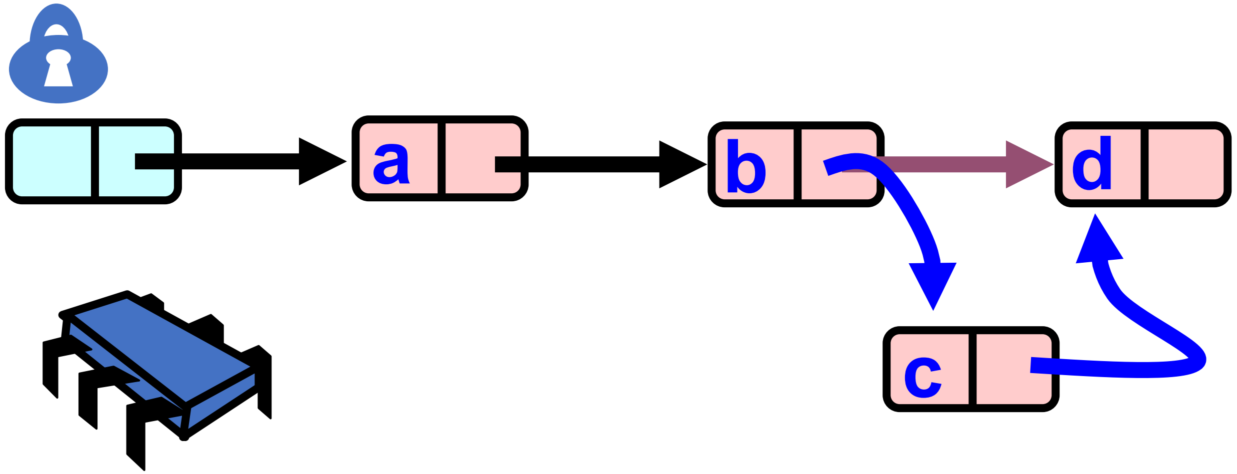
CSE113: Parallel Programming

May 4, 2021

- **Topic**: Concurrent Objects 3
  - Optimistic Linked List
  - Lazy Linked List
  - Lock Free Linked List



From The Art of Multiprocessor Textbook Slides

Announcements

- Midterm is out
    - So far no questions need answering in the discussion thread
    - We have been answering questions in Piazza and email
    - A few people have even submitted! Awesome!
    - Due on Thursday

- HW2 is out
    - Sounds like things are going okay. Visit us in office hours or ask on Piazza if you have questions
    - You can start to compare results (but not code)
    - Also Due on Thursday

Announcements

- Office hours this week:
  - Private because of midterm. We can do an open session for HW3.
  - Sign up sheet will go live at 12:30 on Wednesday. Do not sign up before hand!
  - Docker questions are best for Reese or Gan
  - Gan’s office hours will be on Thursday again this week.

- HW3 will be assigned on Thursday.
  - Due on the 20th

- HW1 grades will be released on Thursday.
  - You need to discuss any discrepancies with us within 2 weeks

Announcements

- Erica Kleinman (phd student in computational media) has a short announcement
- https://docs.google.com/forms/d/e/1FAIpQLSfU-Zf7553T_v7qNCi0mYIR_bqc_vbUDoFjqhFXOdkwjASHqw/viewform
    - I’ll post this in chat too

Quiz

Quiz

- Discuss answers

Schedule

- Review linked list set interface
- Optimistic locking implementation
- Two-step remove implementation (lazy deletion)
- Lock free implementation

Schedule

- **Review linked list set interface**
- Optimistic locking implementation
- Two-step remove implementation (lazy deletion)
- Lock free implementation

Set Interface

- Unordered collection of items
- No duplicates

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

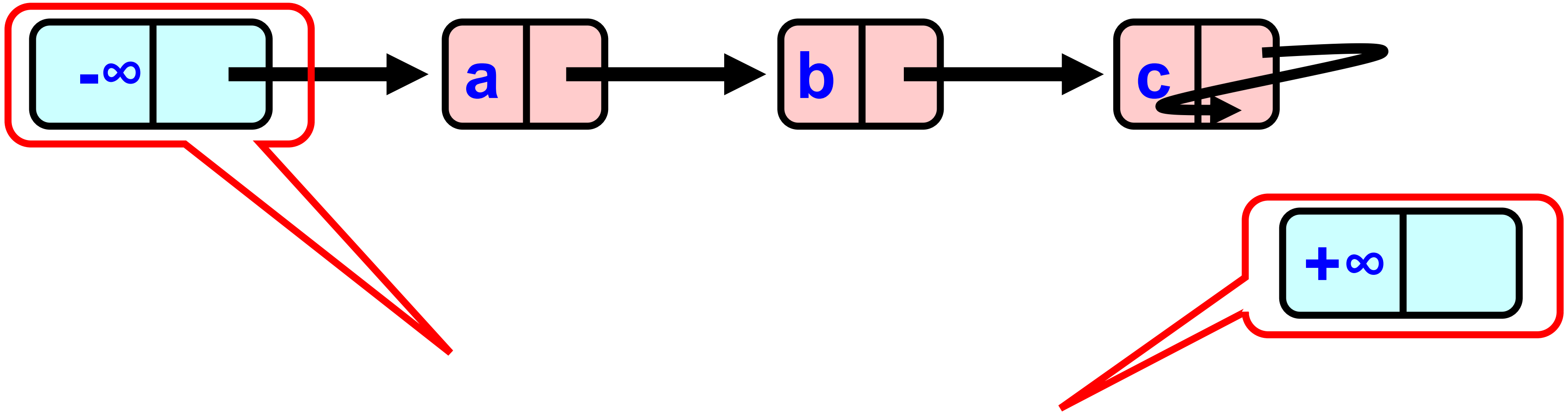
Set Interface

- 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;
}
```

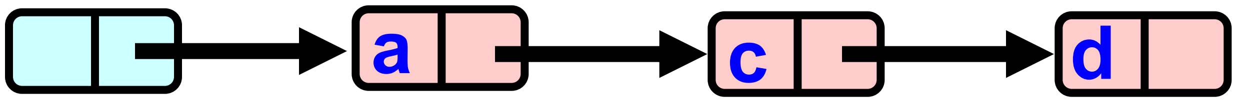
The List-Based Set



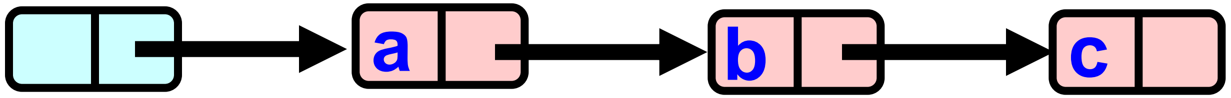
Sorted with Sentinel nodes
(min & max possible keys)

Sequential List Based Set

add(b)

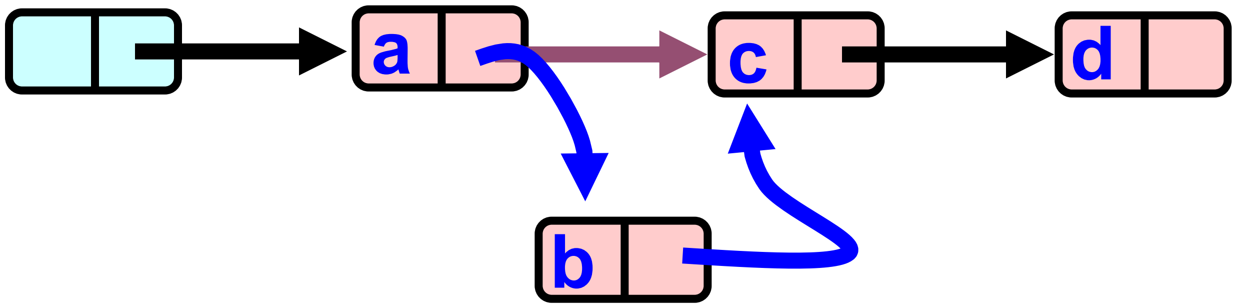


remove(b)

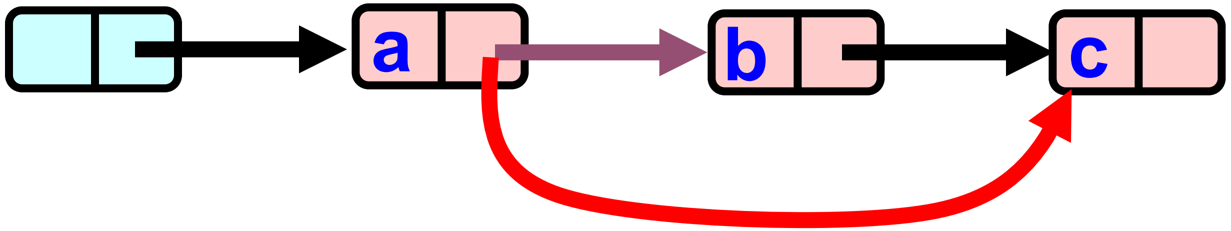


Sequential List Based Set

add(b)

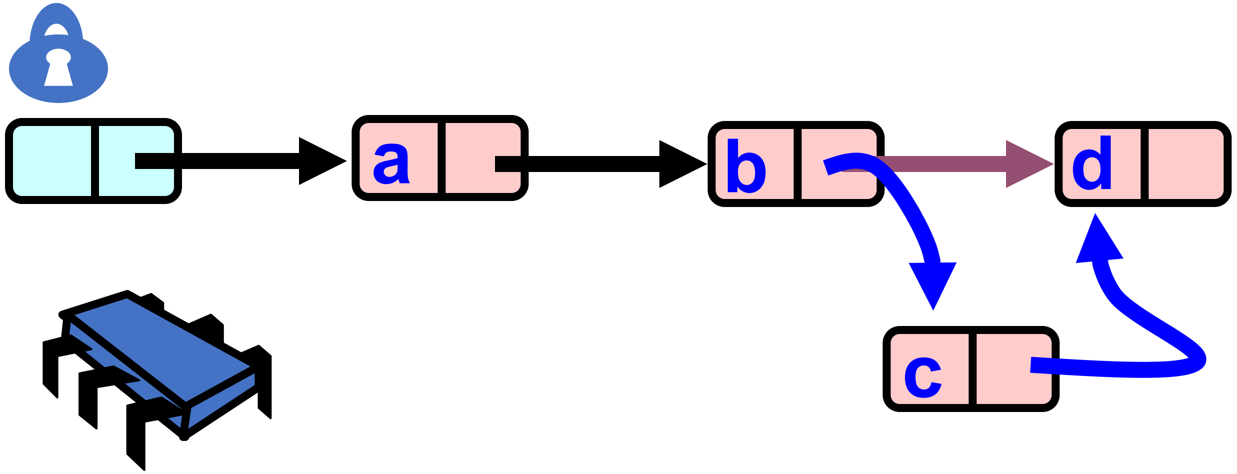


remove(b)

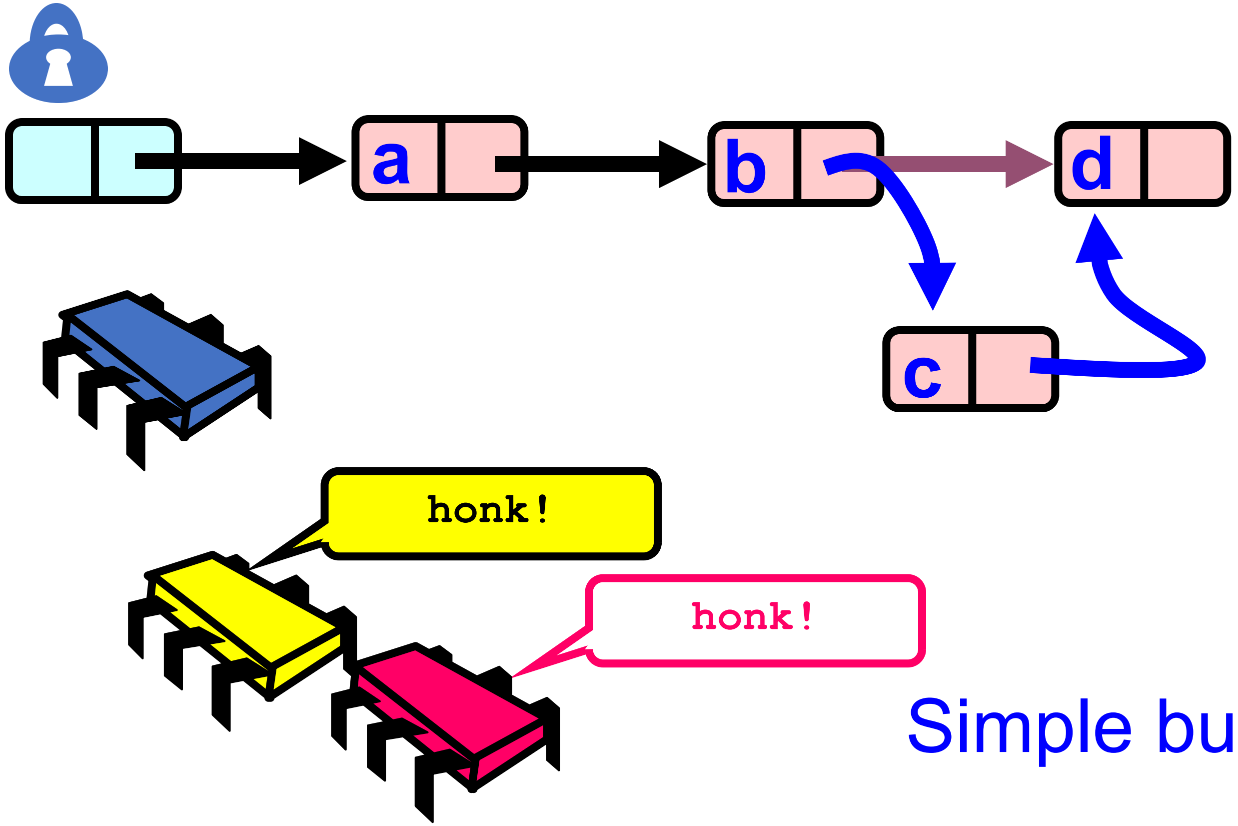


Two approaches so far:

Coarse-Grained Locking



Coarse-Grained Locking

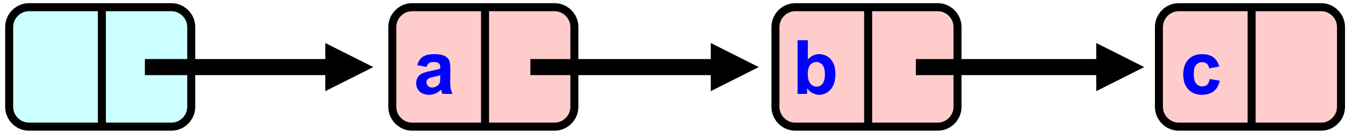


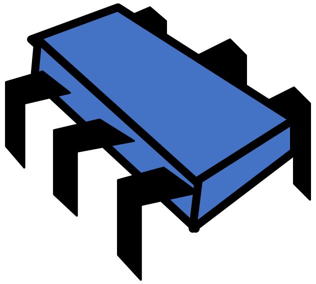
Simple but inefficient!

Second approach

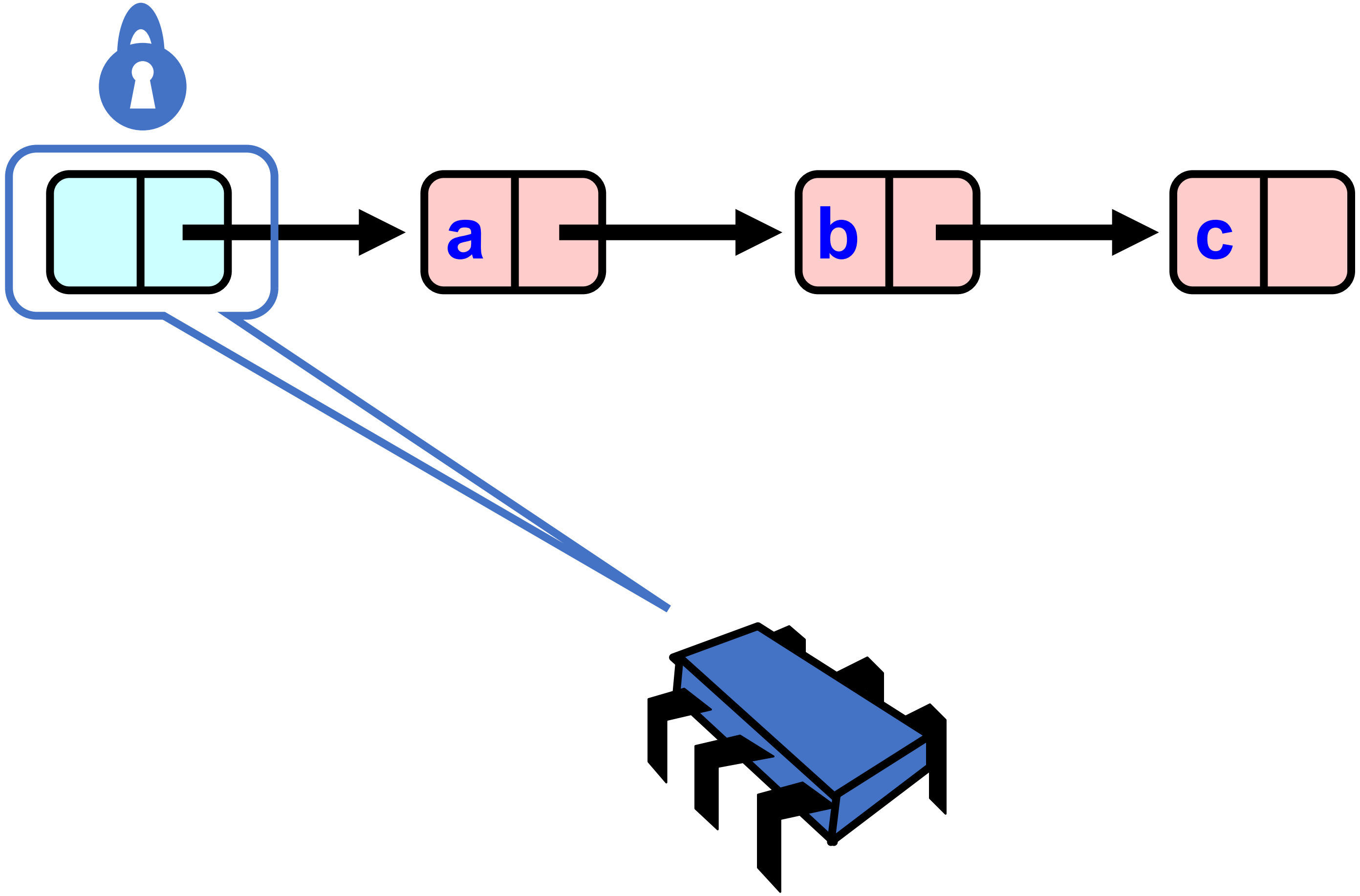
- Fine grained locking

Hand-over-Hand locking

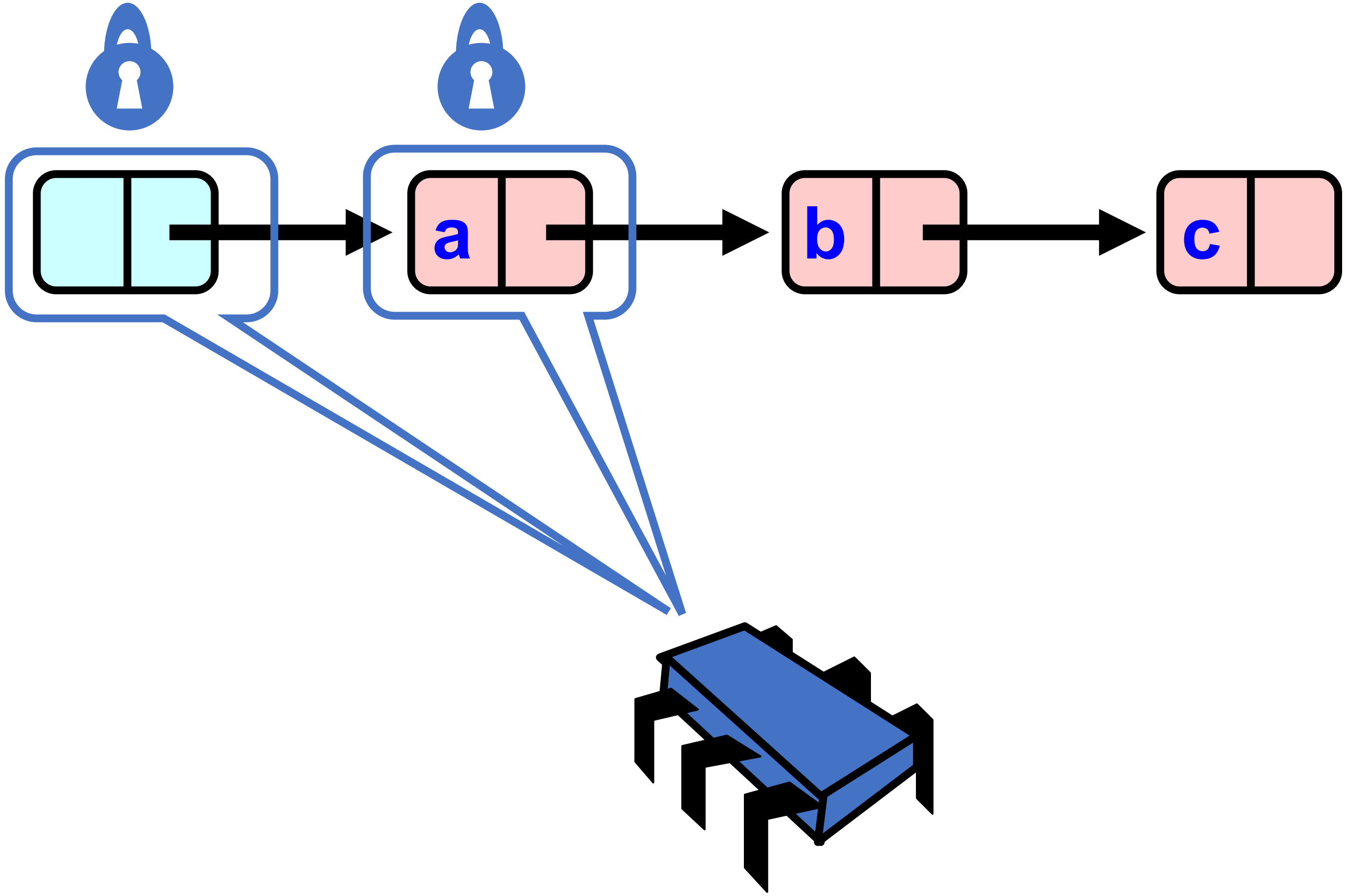




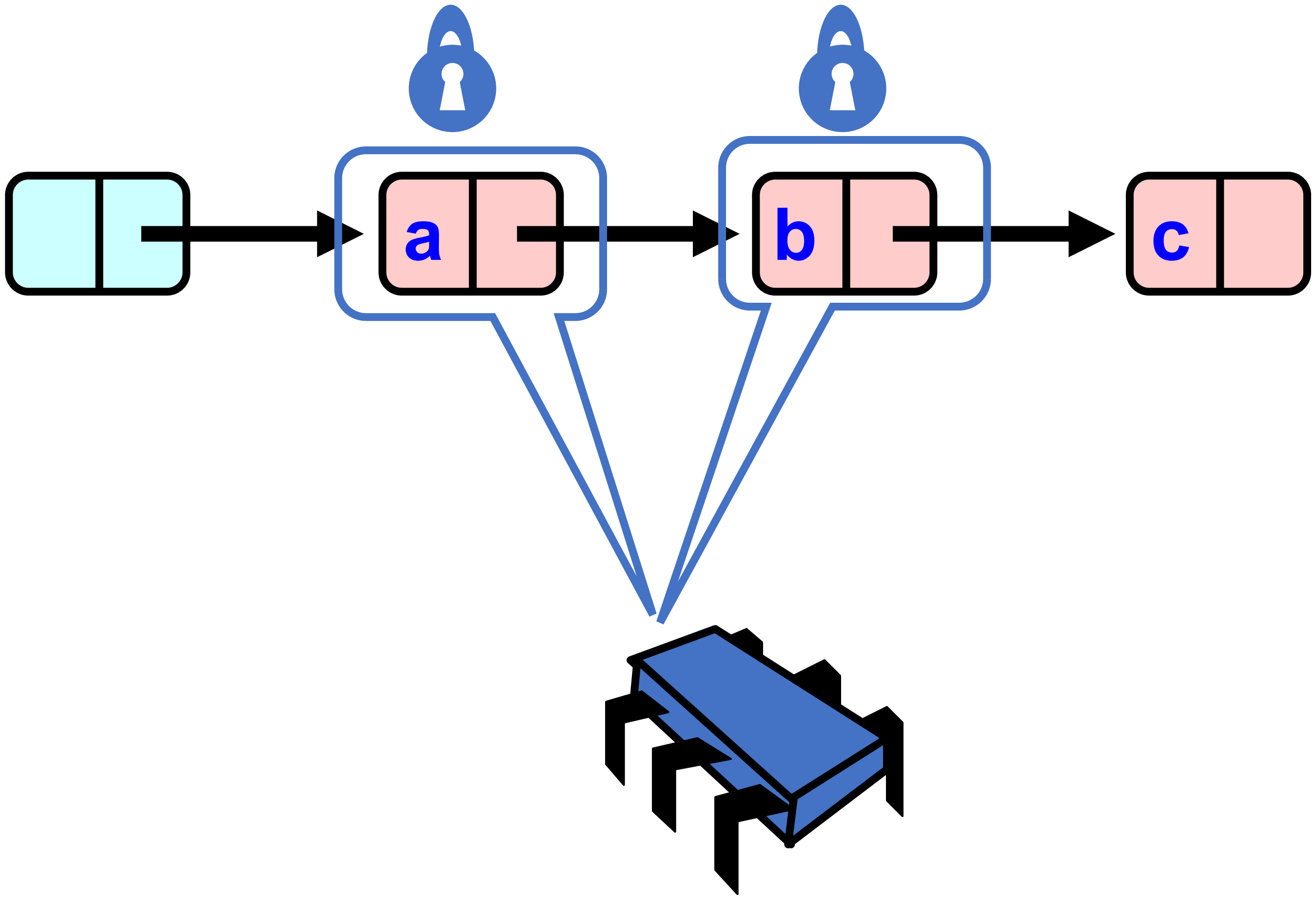
Hand-over-Hand locking



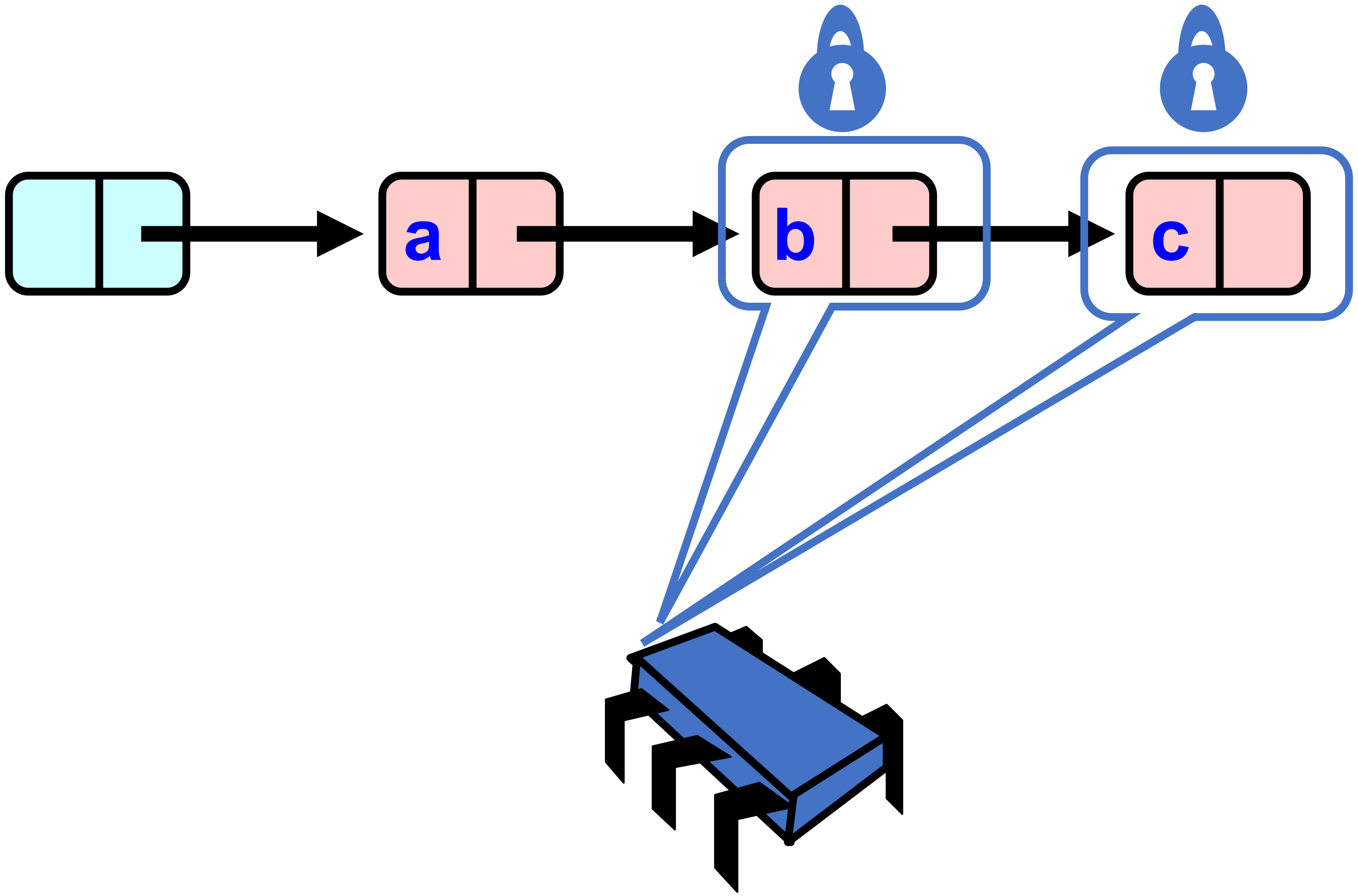
Hand-over-Hand locking



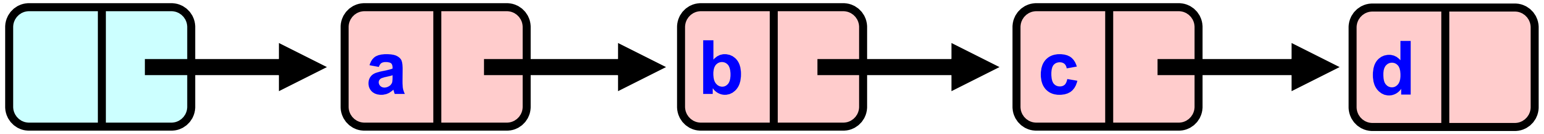
Hand-over-Hand locking

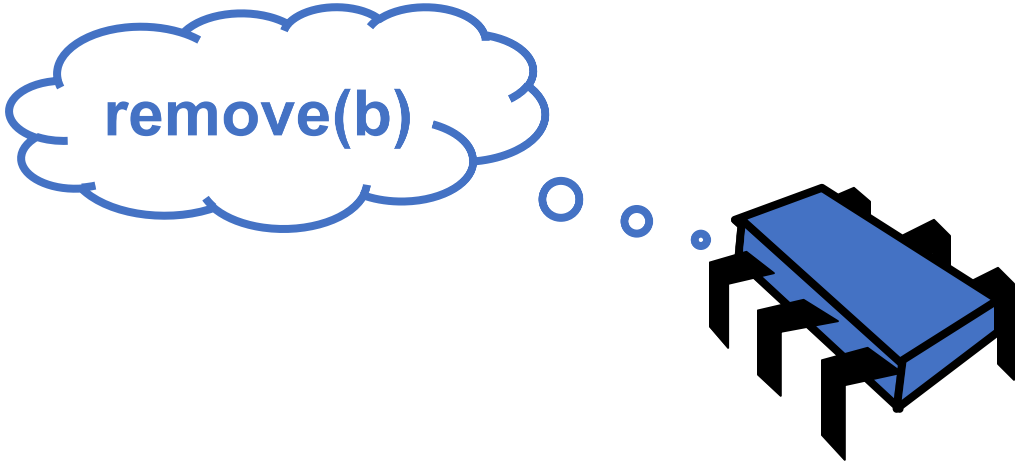


Hand-over-Hand locking

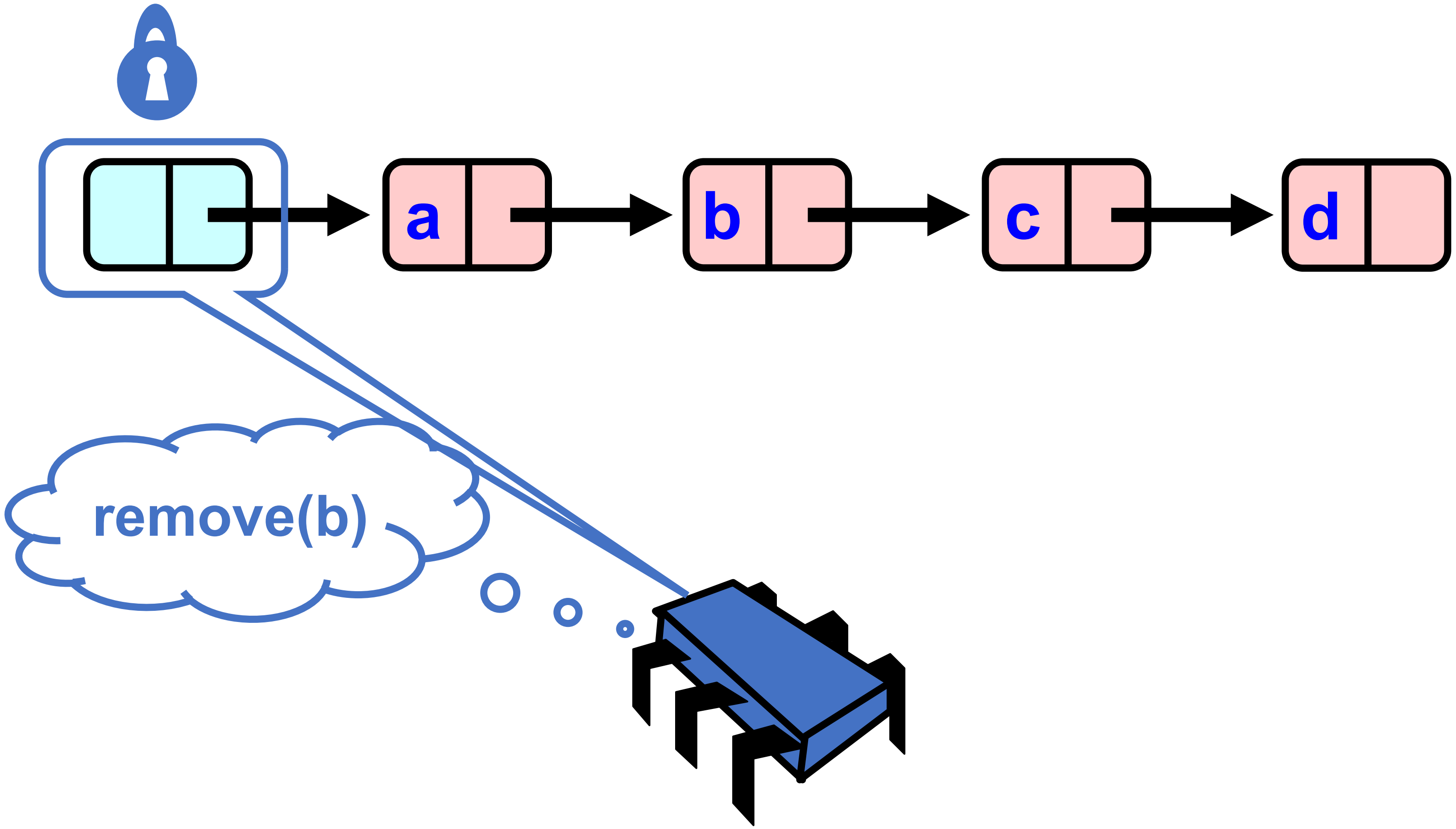


Removing a Node

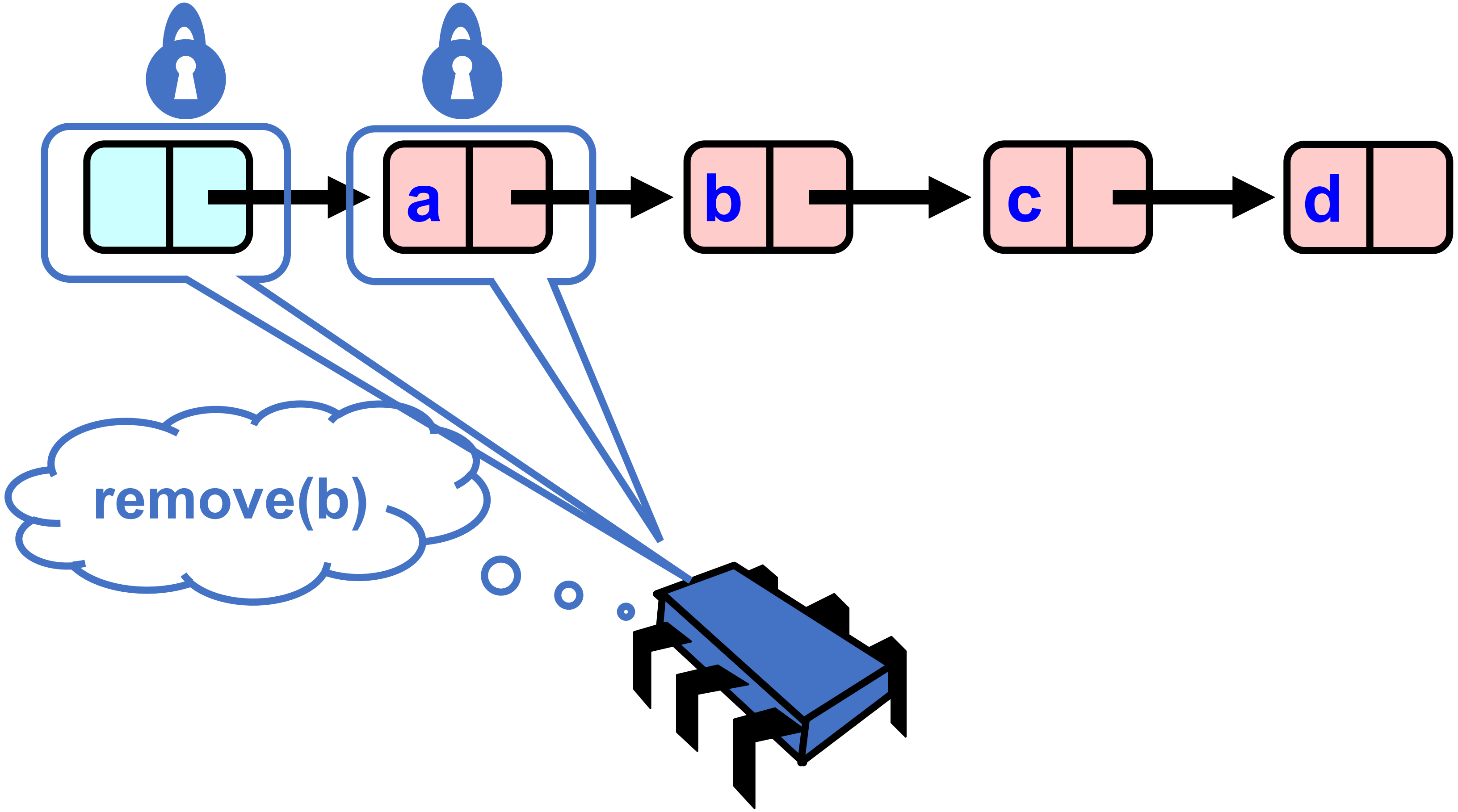




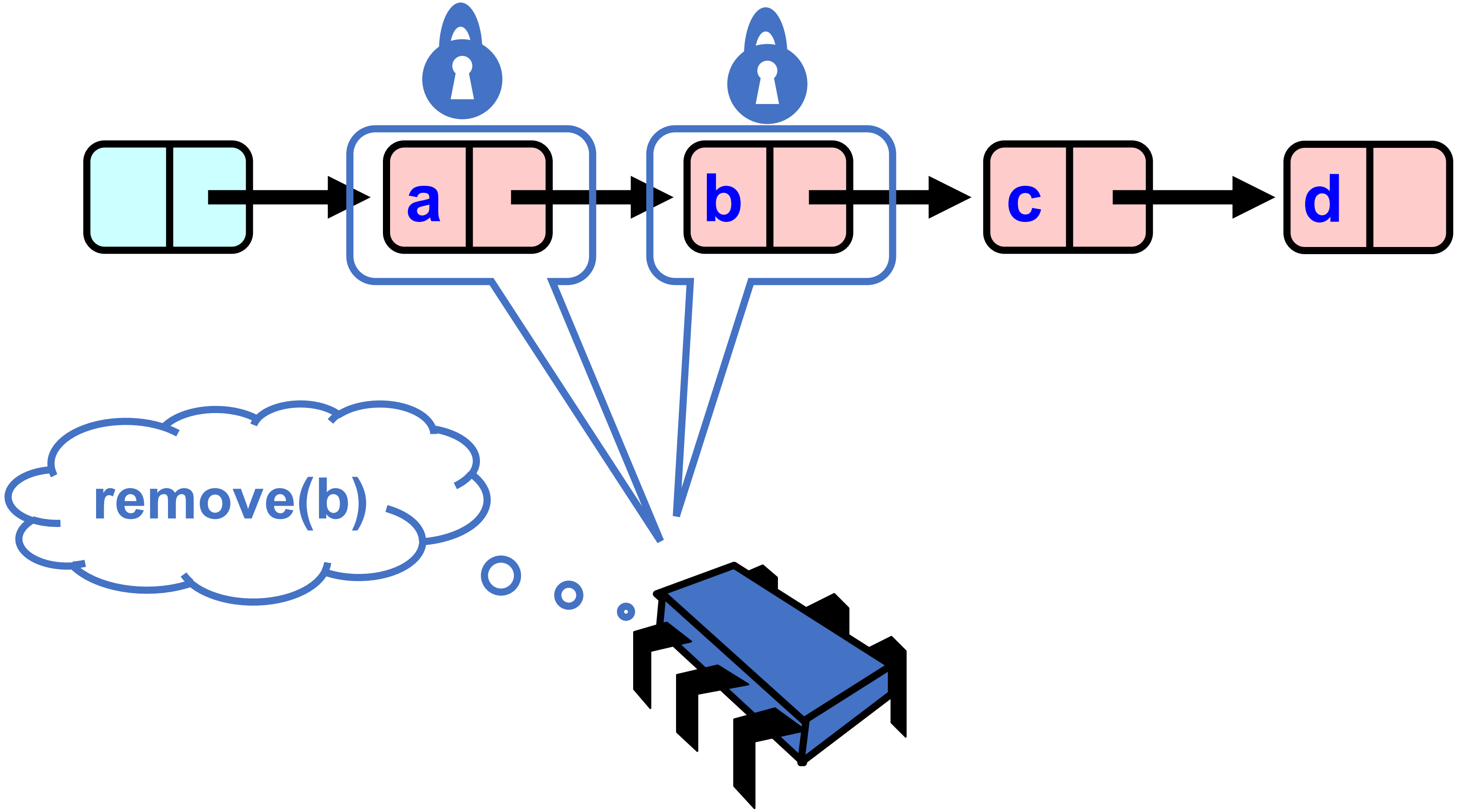
Removing a Node



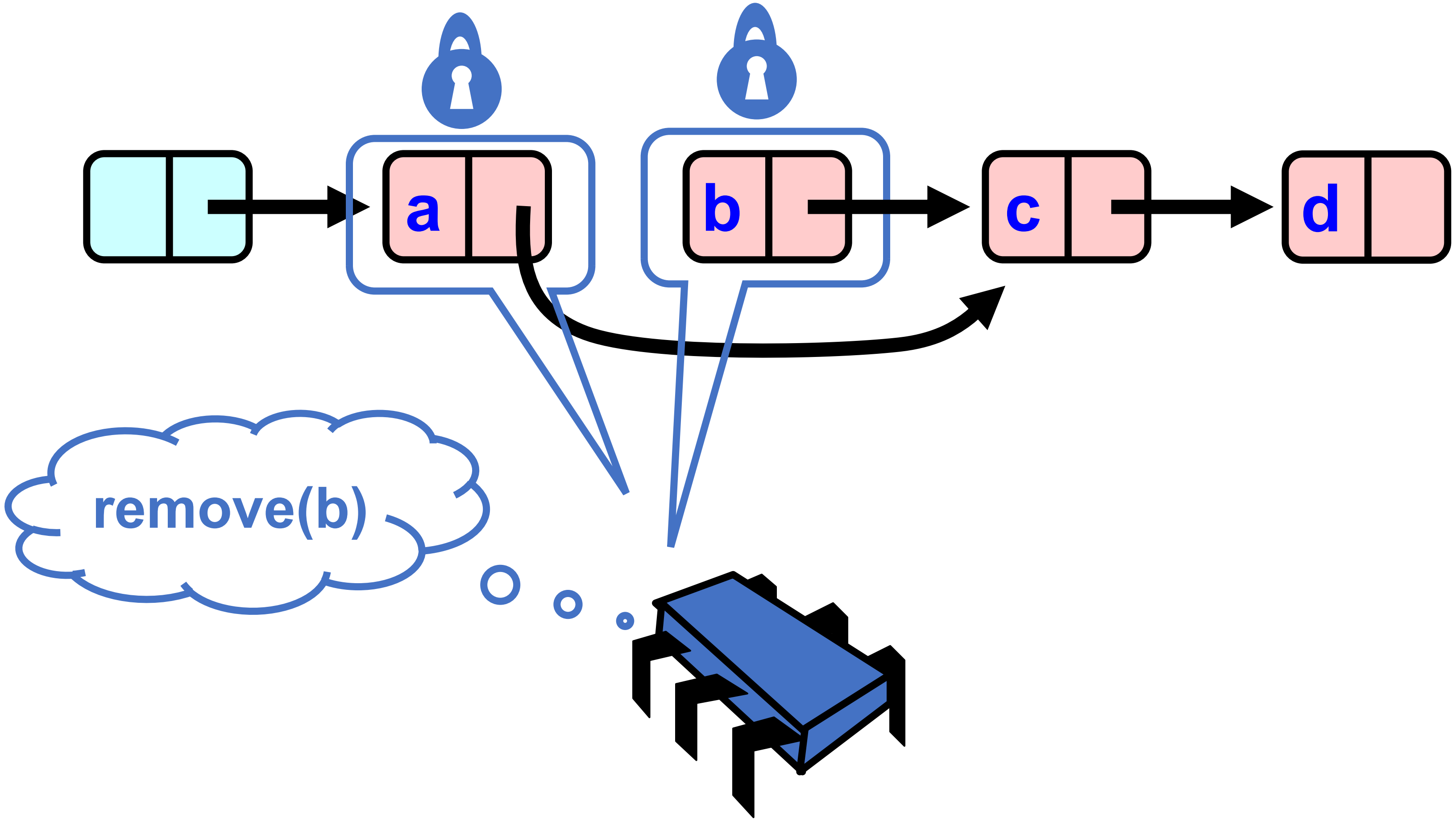
Removing a Node



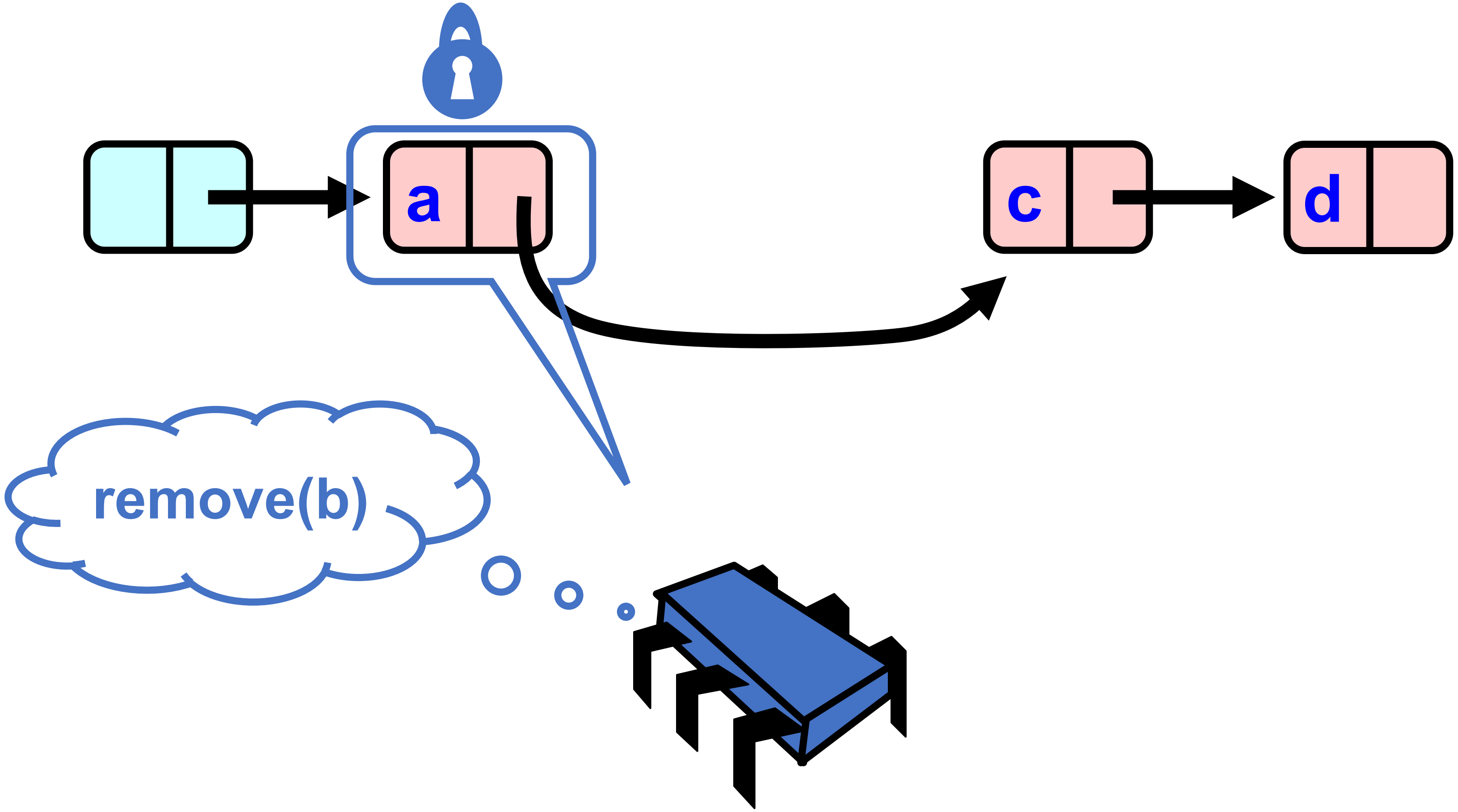
Removing a Node



Removing a Node



Removing a Node



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)

Schedule

- Review linked list set interface
- **Optimistic locking implementation**
- Two-step remove implementation (lazy deletion)
- Lock free implementation

Optimistic Synchronization

We’ve seen this term before... Where?

Optimistic Synchronization

We’ve seen this term before... Where?

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

Optimistic Synchronization

We’ve seen this term before... Where?

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

What was the alternative?

Optimistic Synchronization

- Find nodes without locking

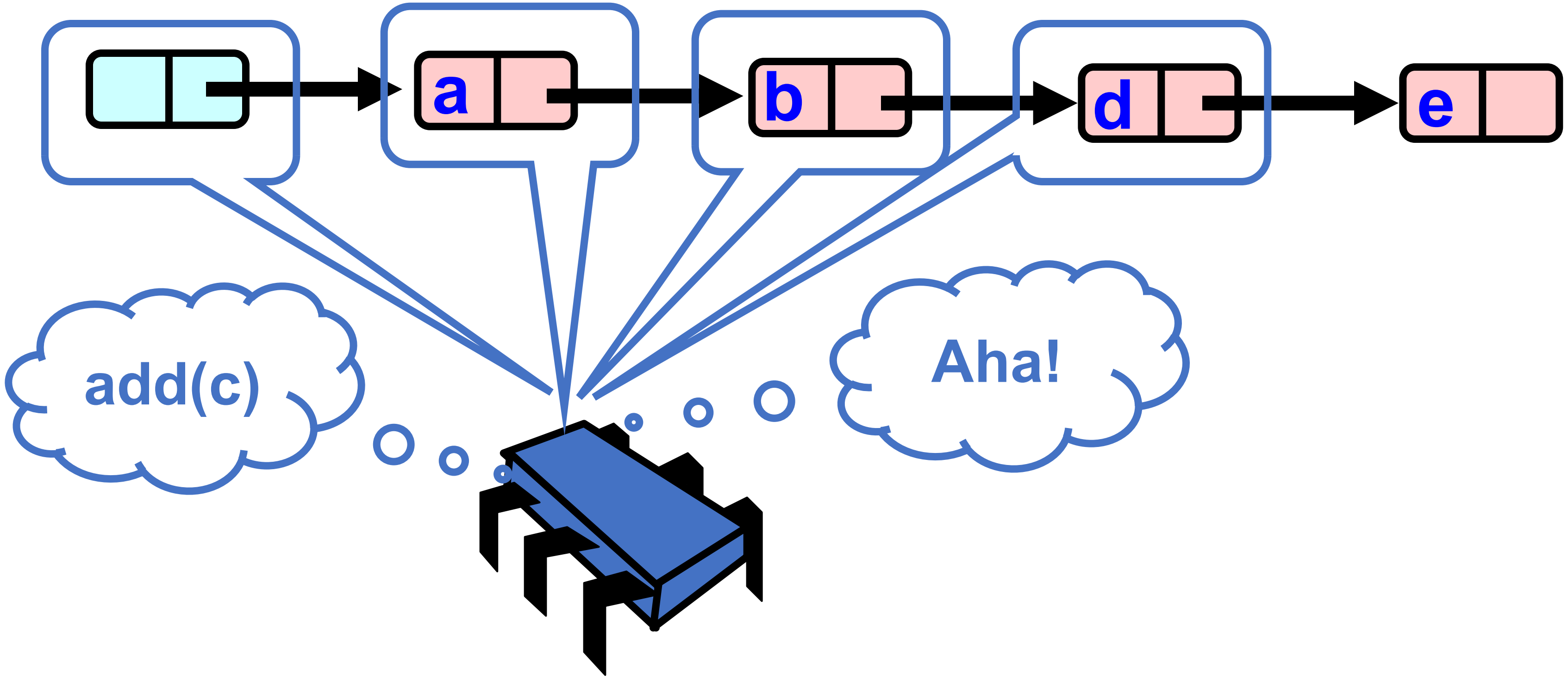
Optimistic Synchronization

- Find nodes without locking
- Lock nodes

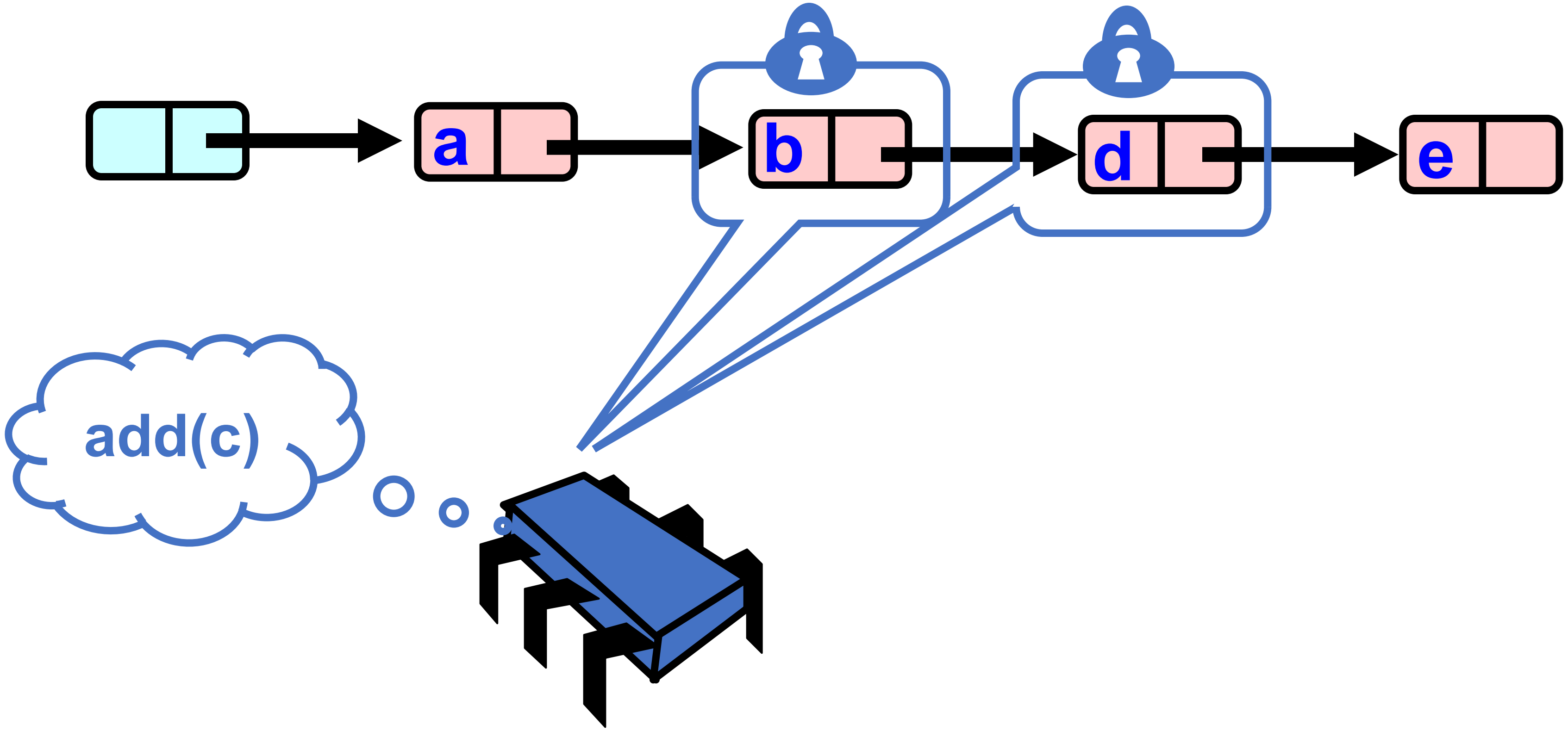
Optimistic Synchronization

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

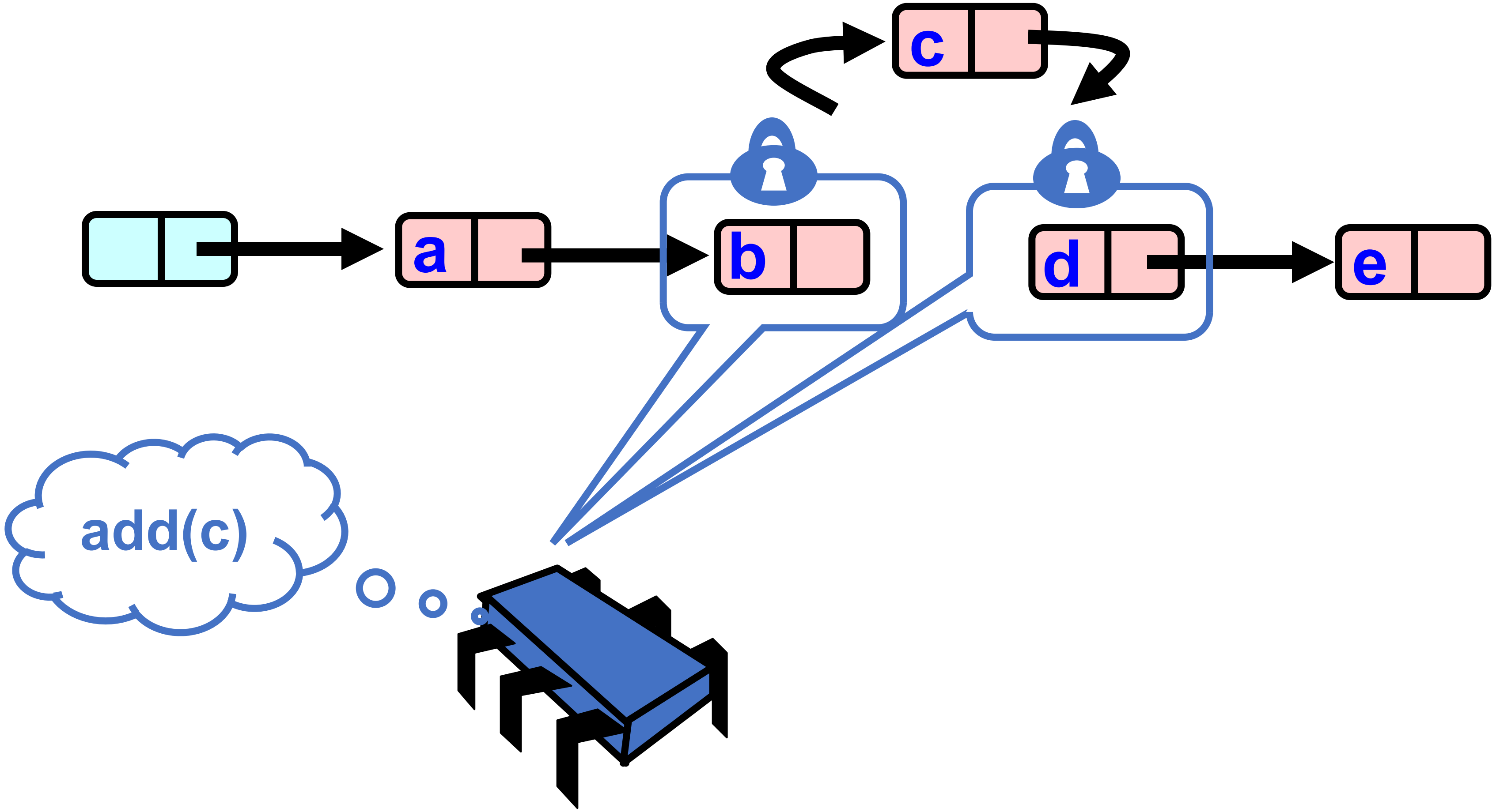
Optimistic: Traverse without Locking



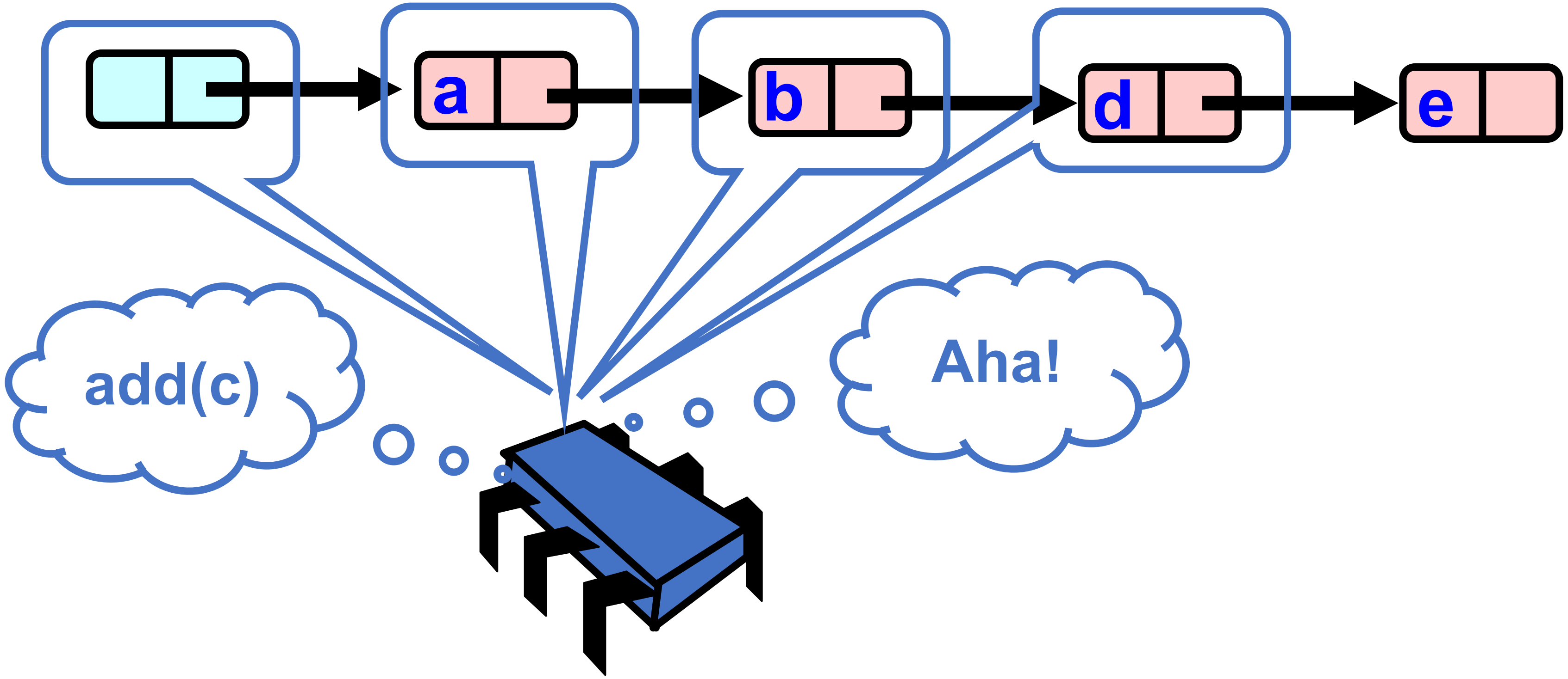
Optimistic: Lock and Load



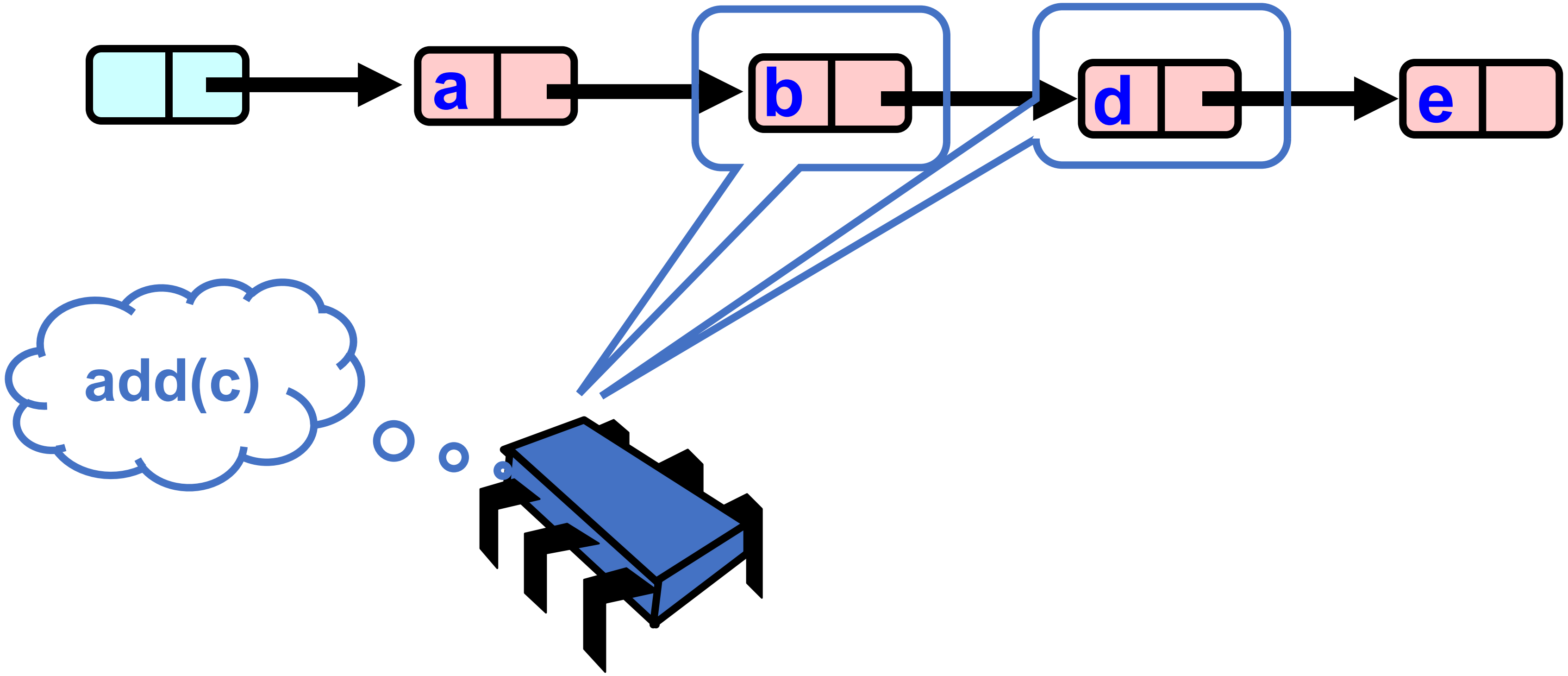
Optimistic: Lock and Load



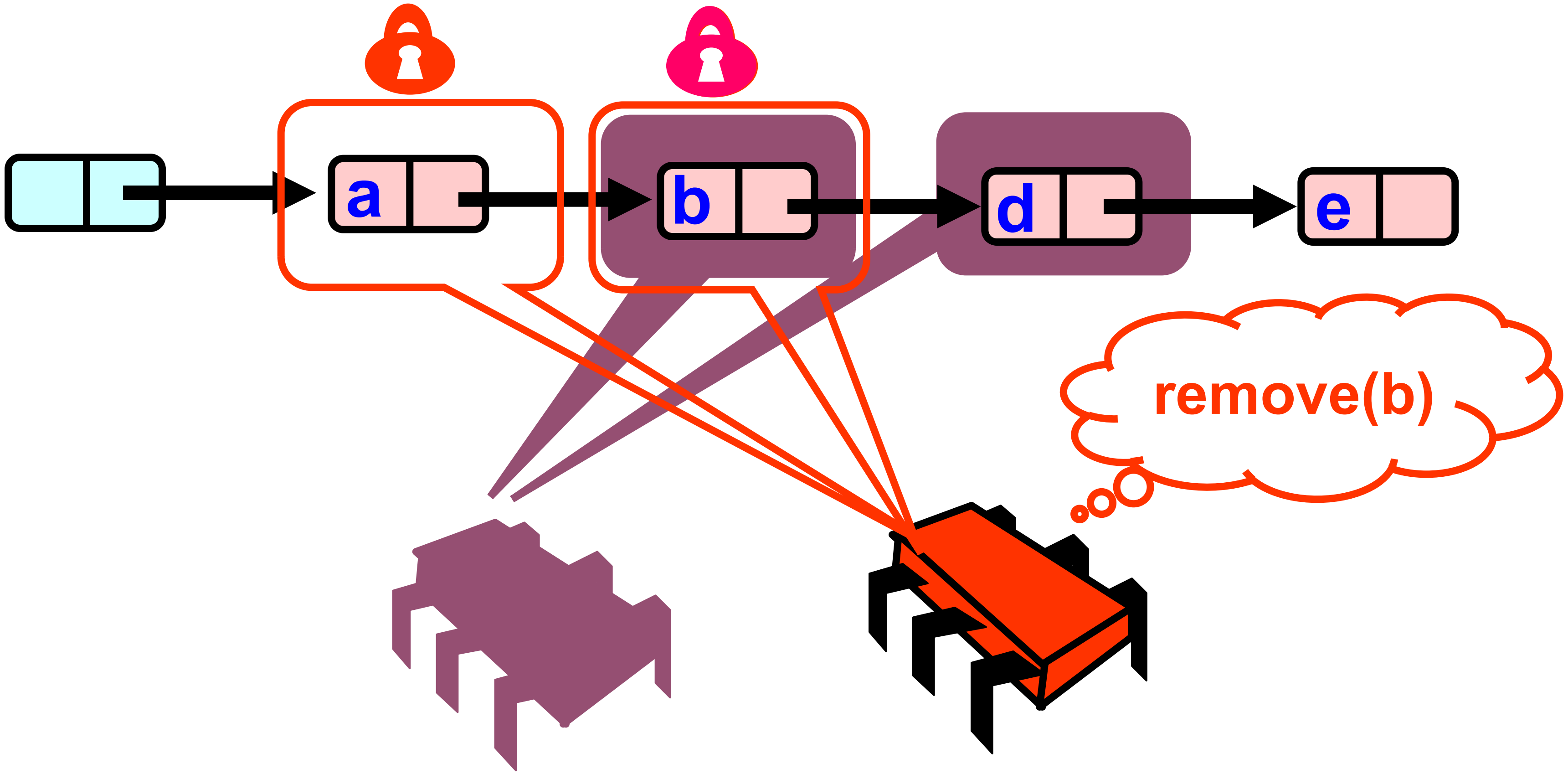
What could go wrong?



What could go wrong?



What could go wrong?



Data conflict!

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

- What do we do?

Data conflict!

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

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

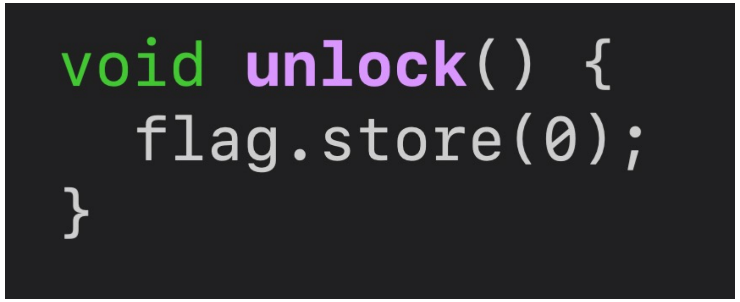
Lock-free reasoning

- We can use atomic variables

- Recall reasoning about the mutexes

Analysis

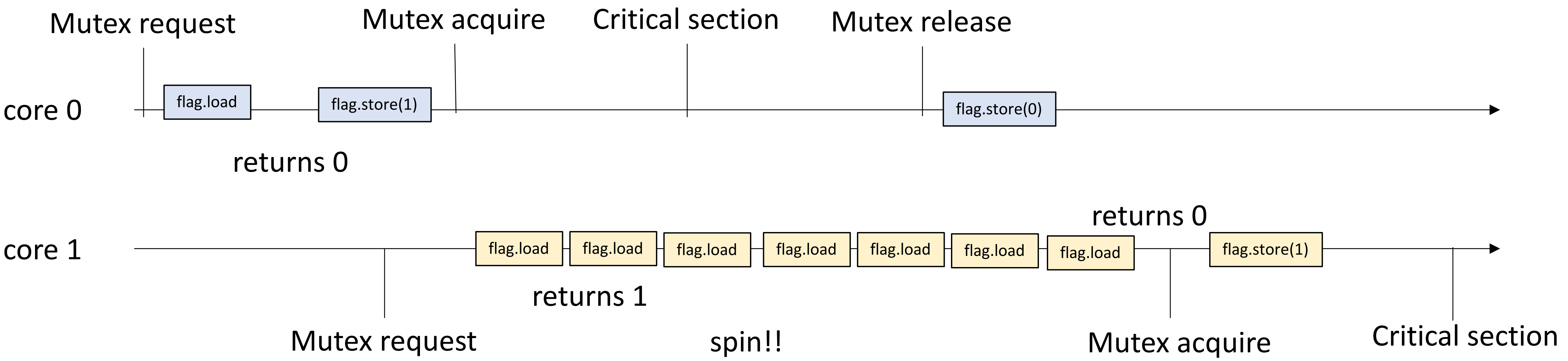
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```



```
void unlock() {
  flag.store(0);
}
```

Thread 0:
`m.lock();`
`m.unlock();`

Thread 1:
`m.lock();`
`m.unlock();`



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

Lock-free reasoning

- Default atomic accesses are documented to be sequentially consistent.

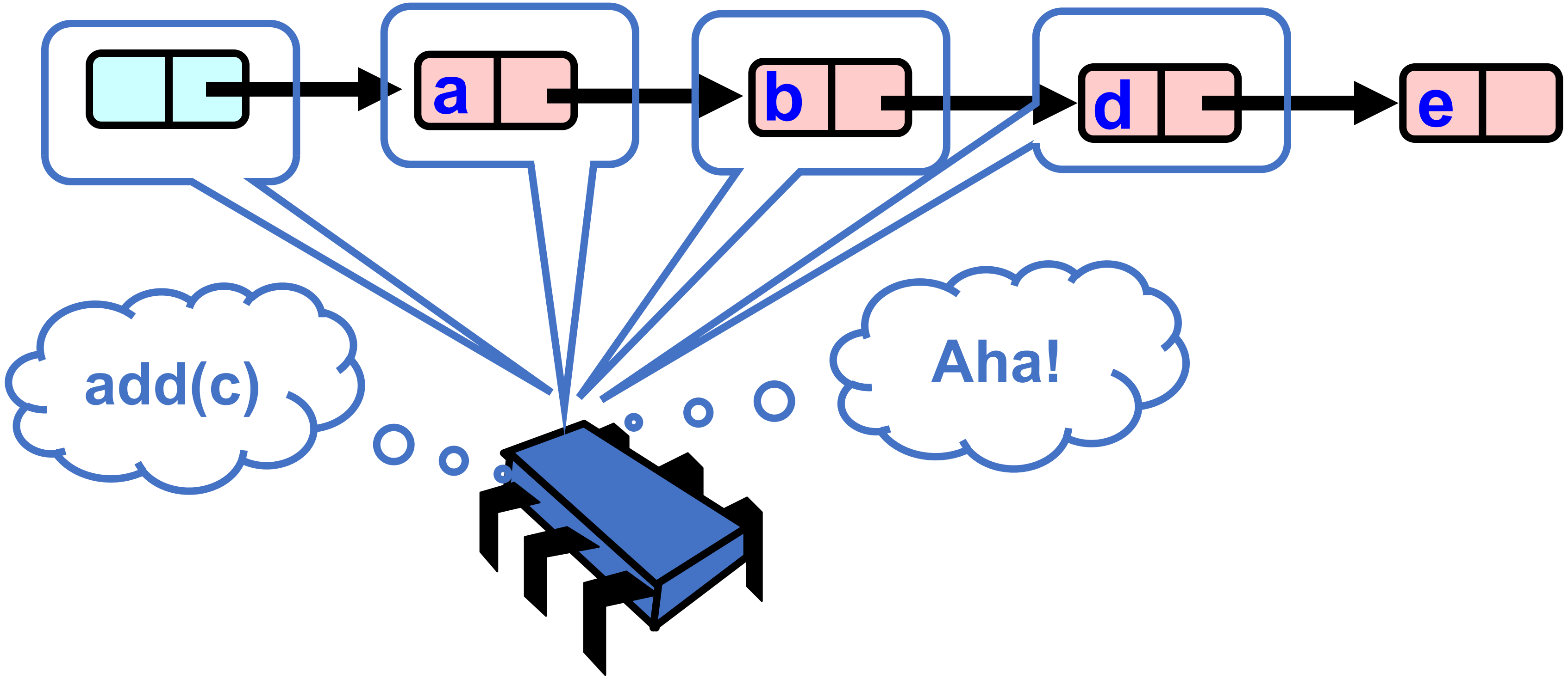
```
void traverse(node *n) {
  while (n->next != NULL) {
    n = n->next;
  }
}
```

Lock-free reasoning

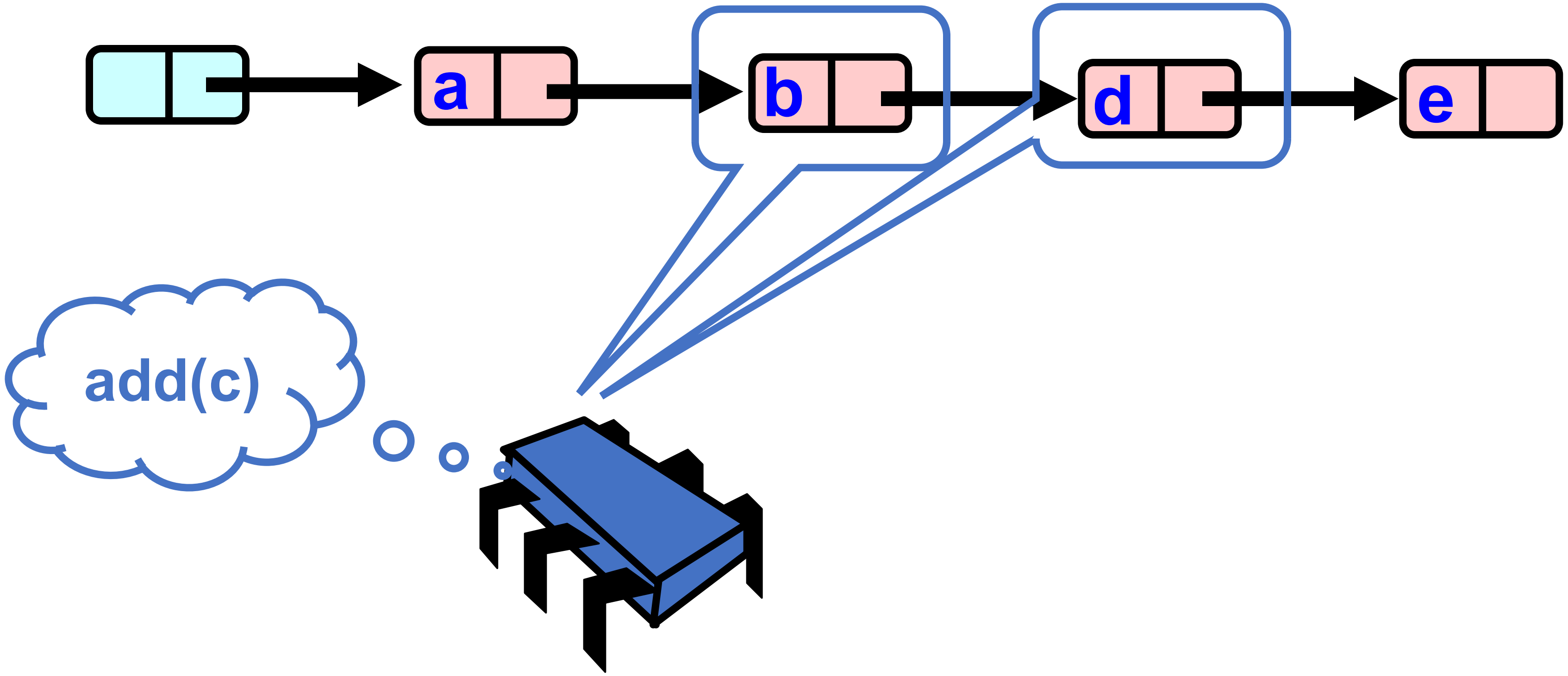
- Default atomic accesses are documented to be sequentially consistent.

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

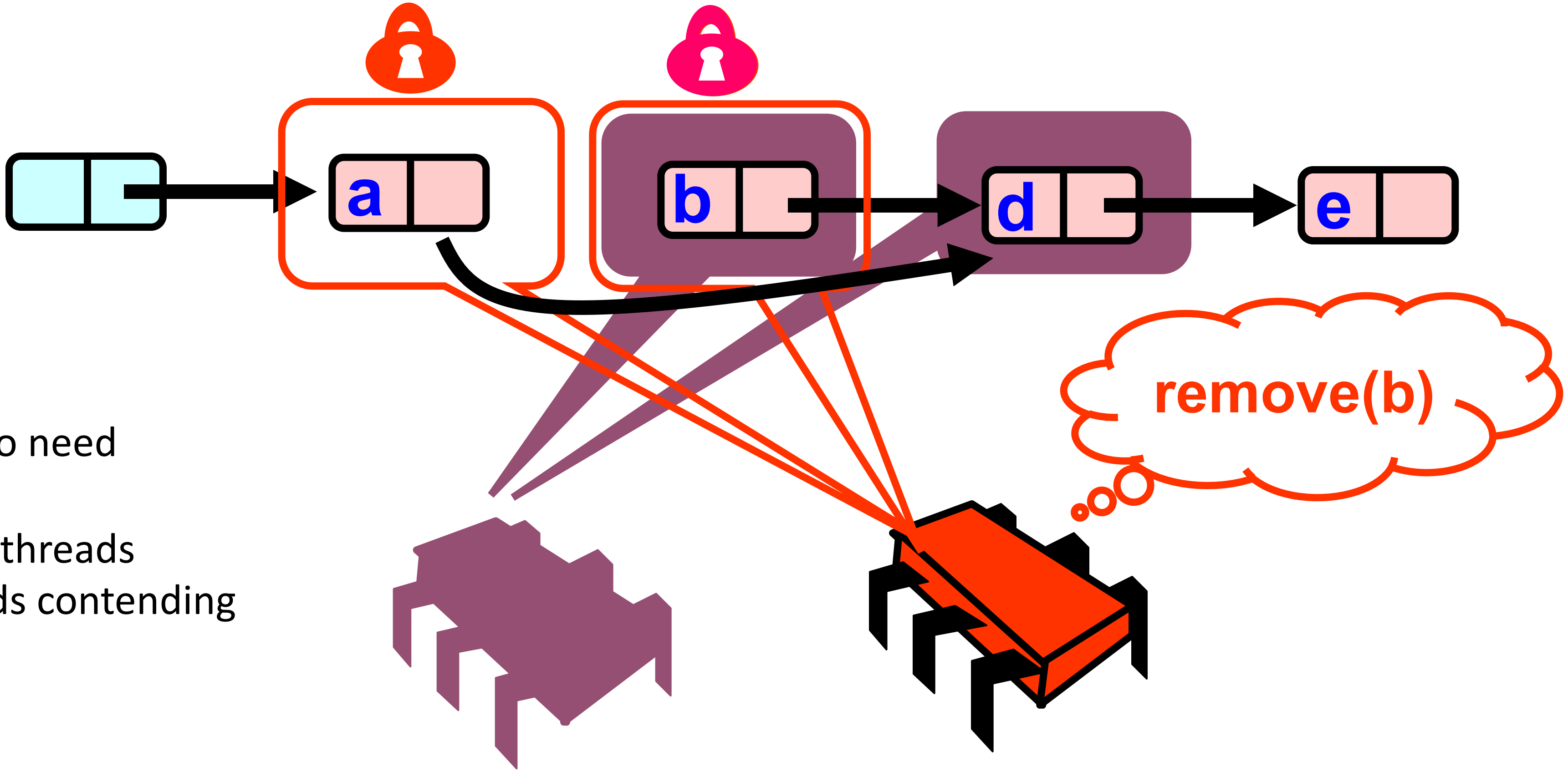
What could go wrong?



What could go wrong?

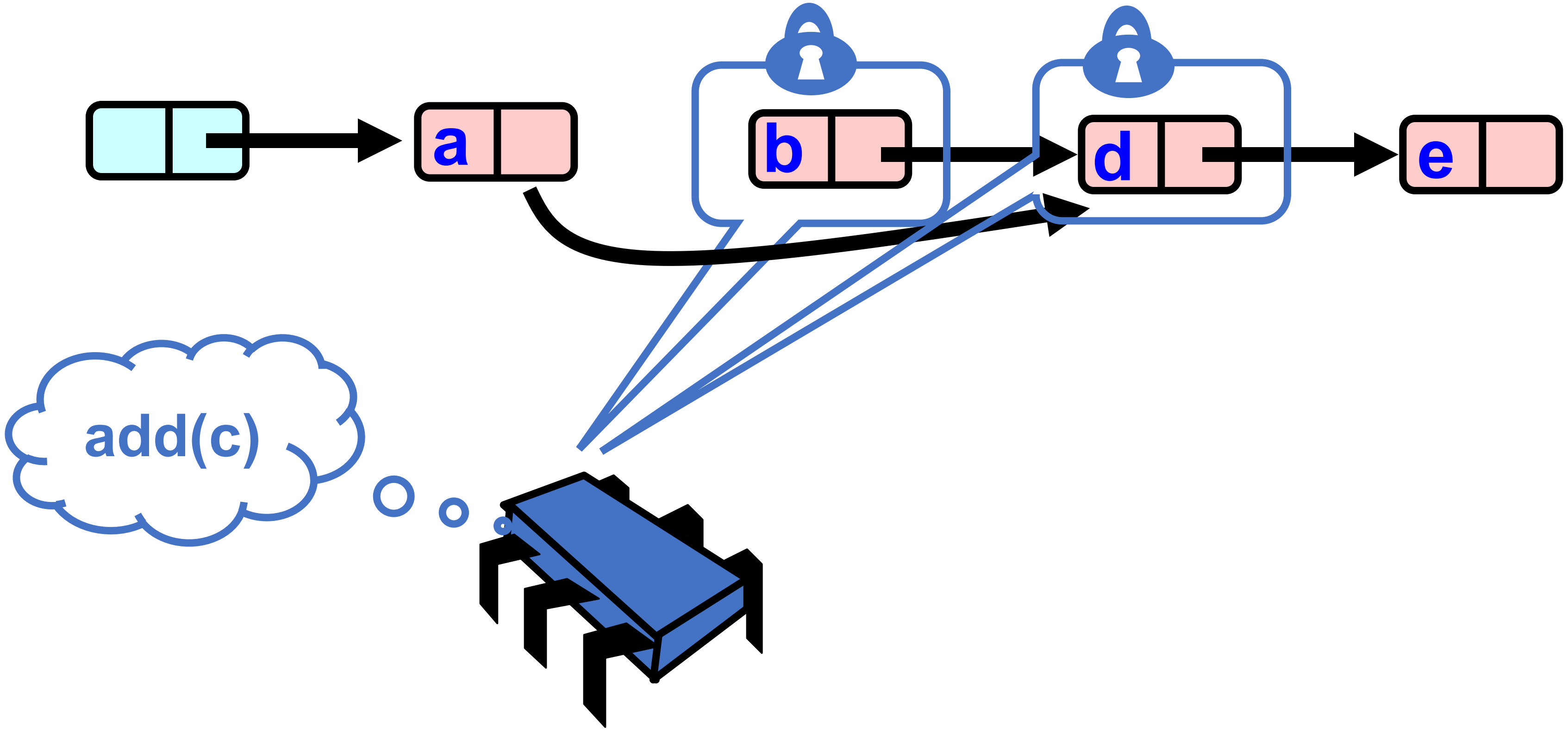


What could go wrong?

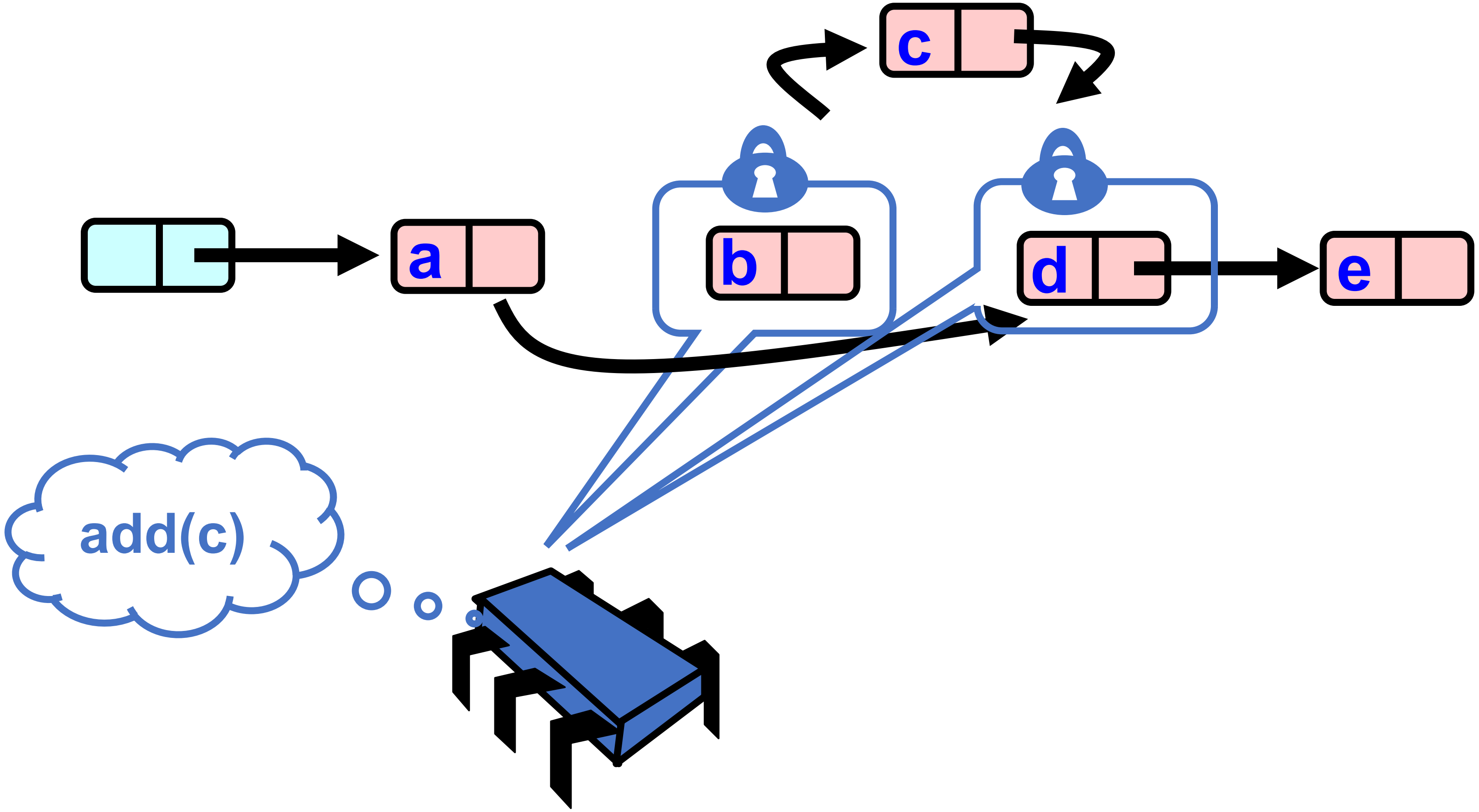


No more data
conflict, but we do need
to reason about
interleavings and threads
concurrent threads contending
for values.

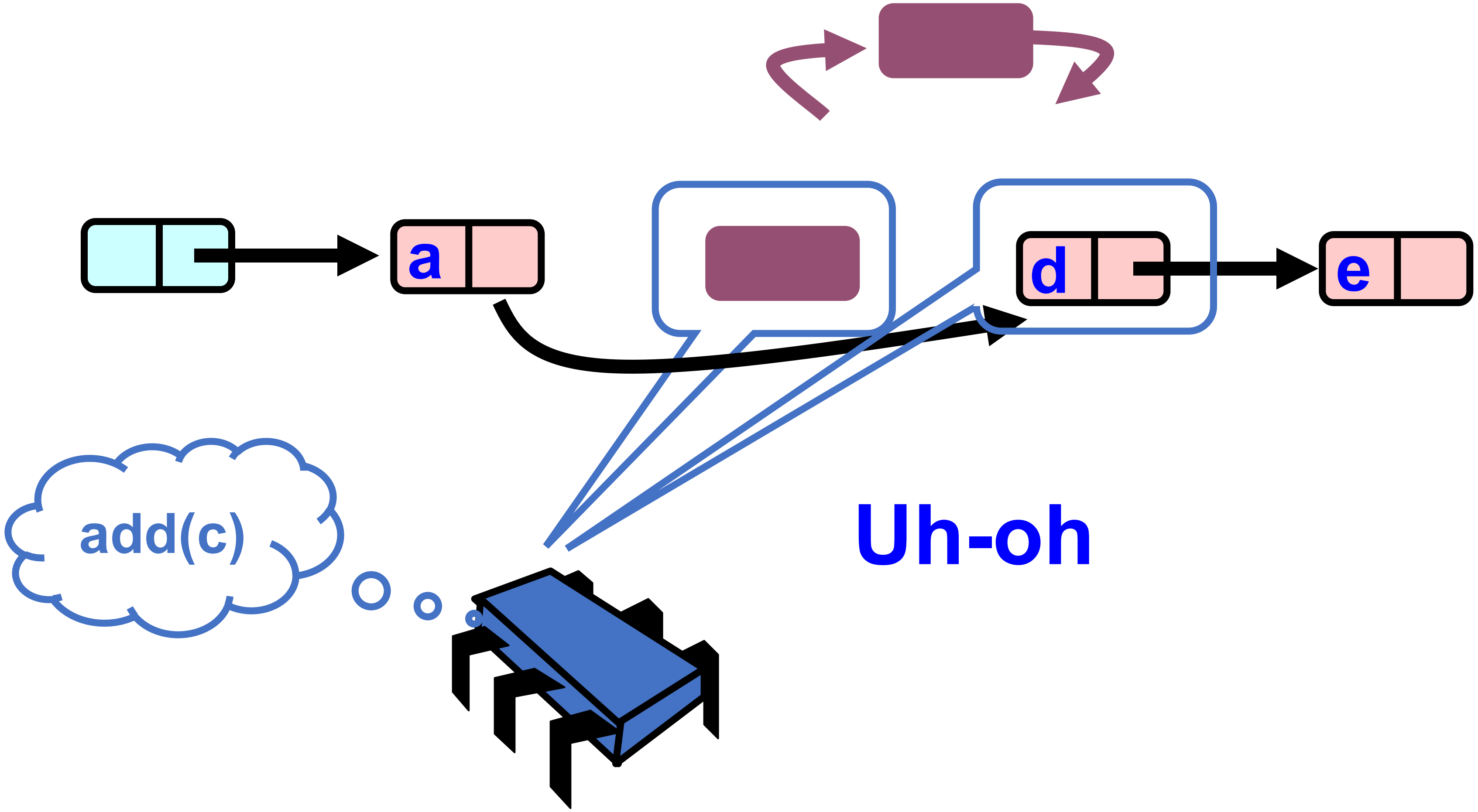
What could go wrong?



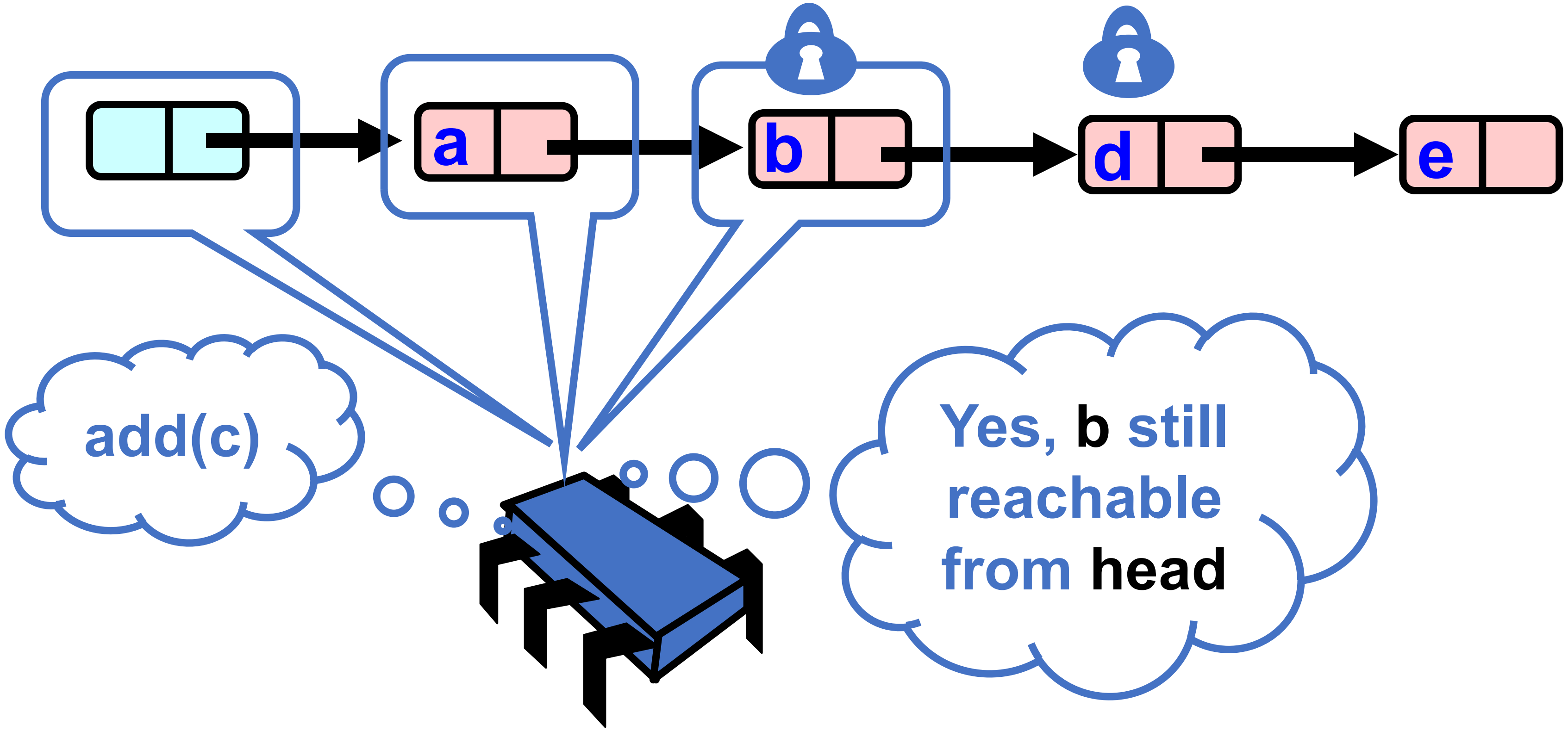
What could go wrong?



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!

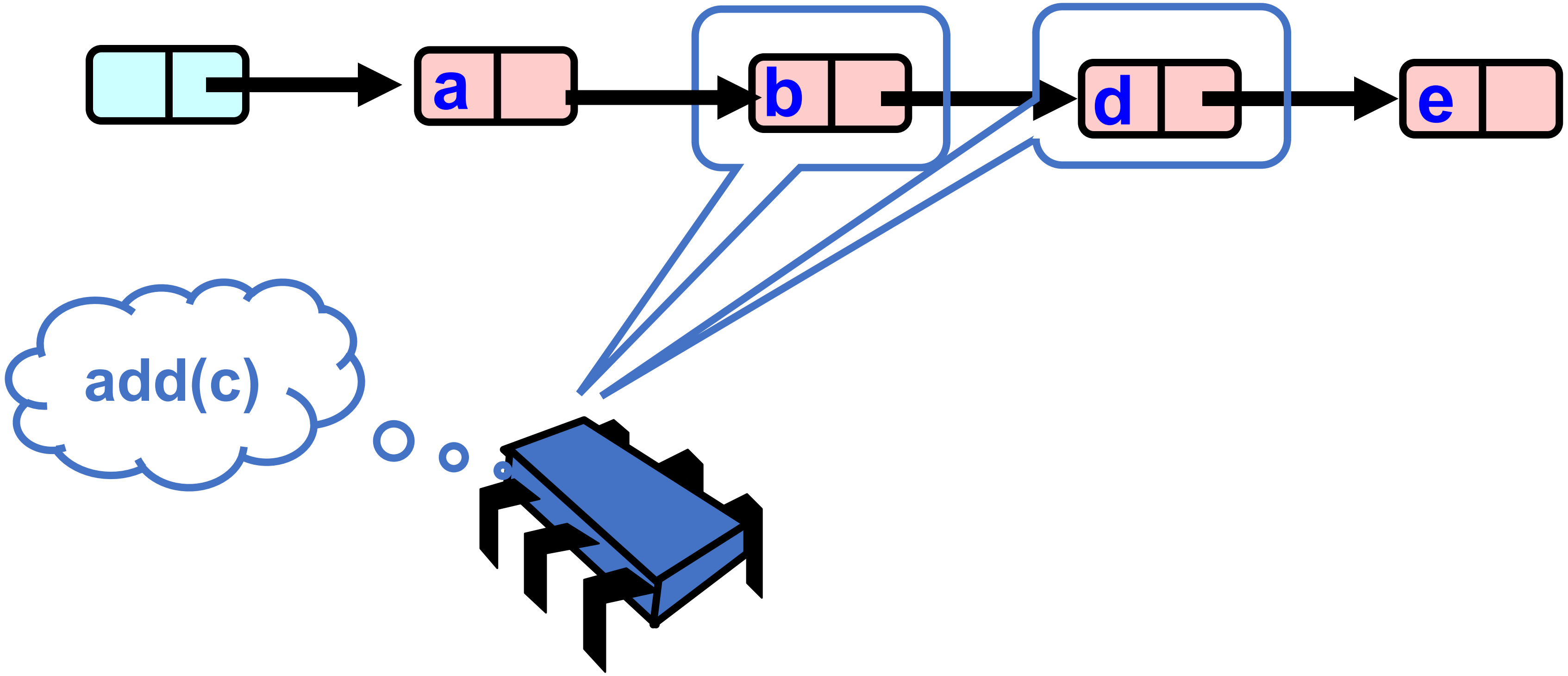
What happens if failure?

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

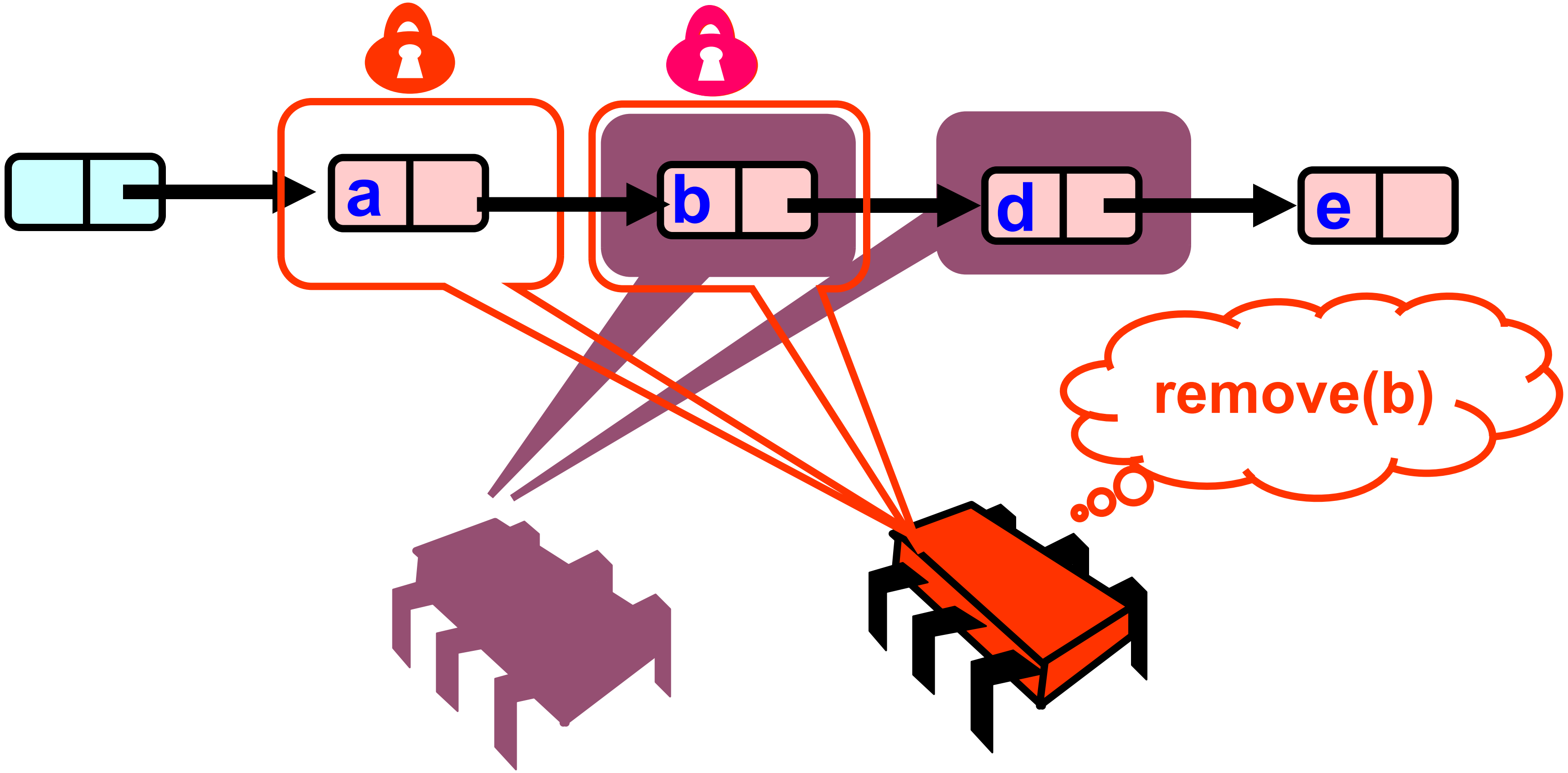
- Private method:
  - try_remove
  - remove loops on try_remove until it succeeds

What about deletion?

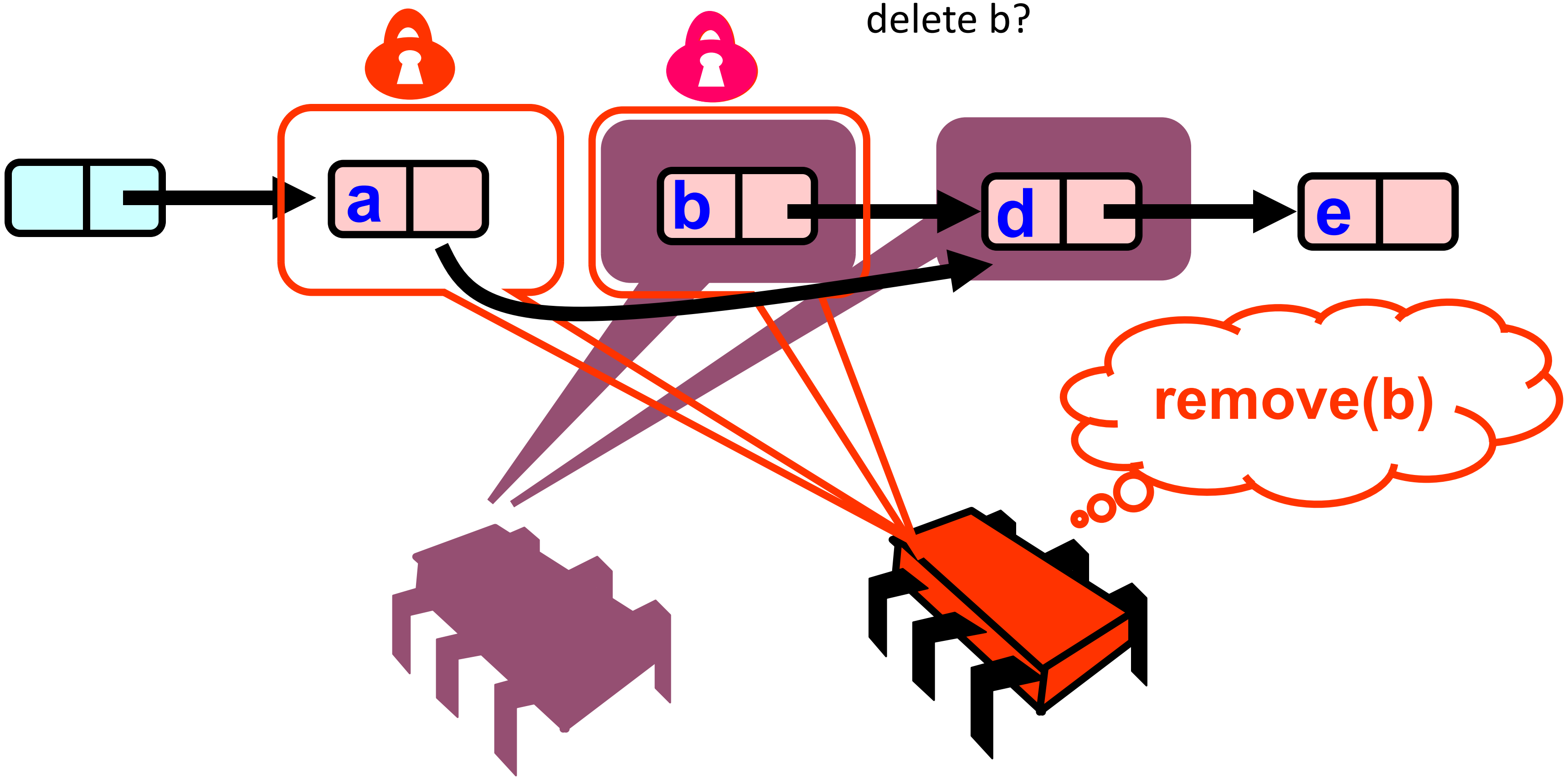
Can threads that remove a node delete it?



Can threads that remove a node delete it?

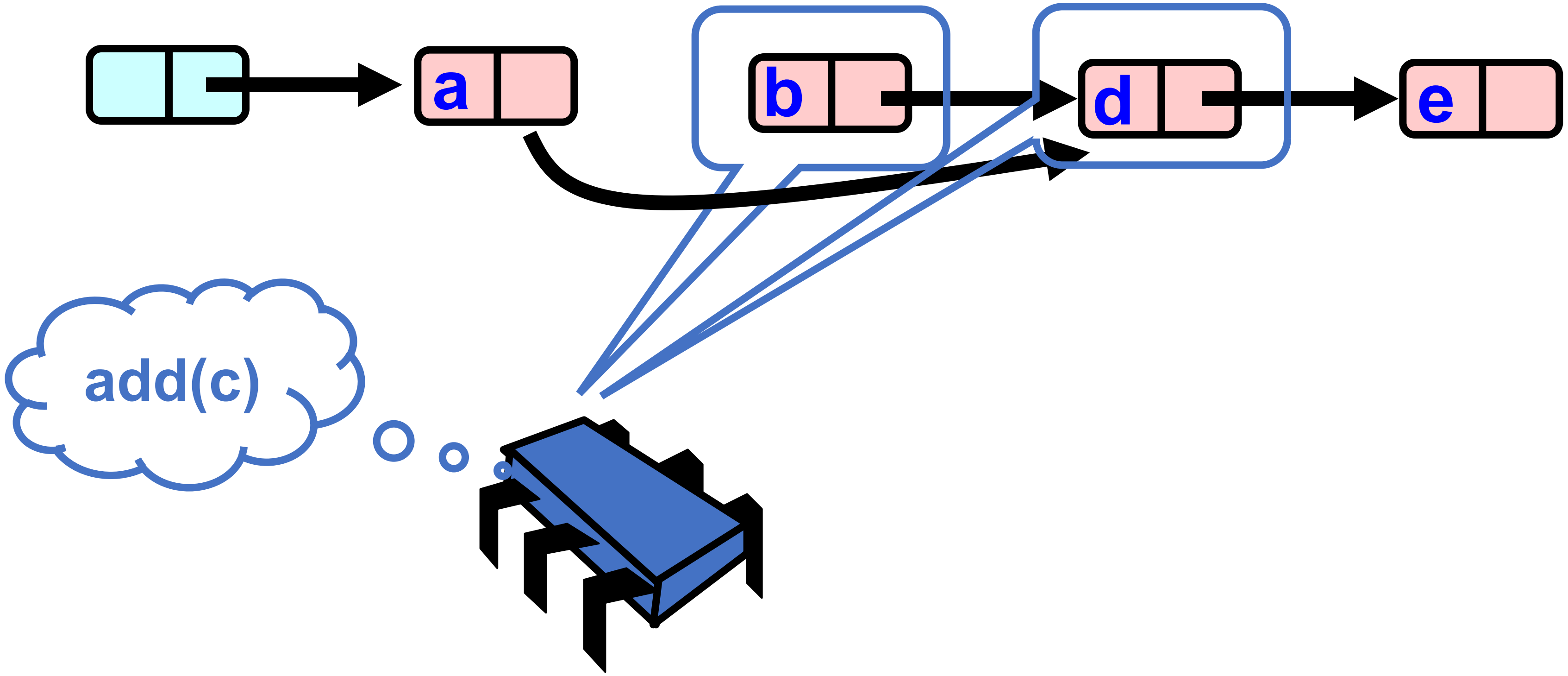


Can threads that remove a node delete it?

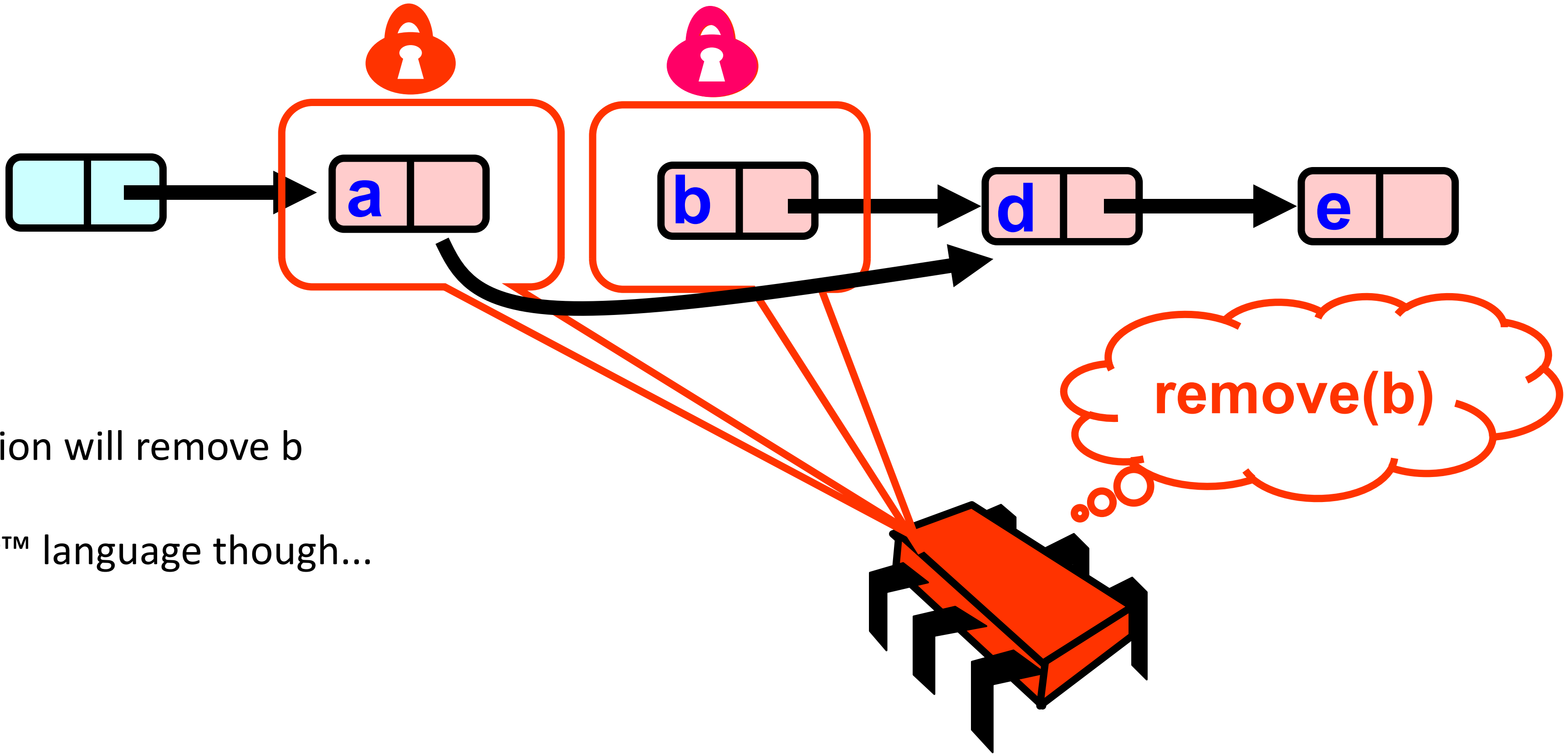

delete b?

Can threads that remove a node delete it?

still on b!



Our own garbage collector



Java’s garbage collection will remove b

We are using a better™ language though...

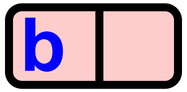
maintain a list to delete:

Our own garbage collector

Java's garbage collection will remove b

We are using a better™ language though...

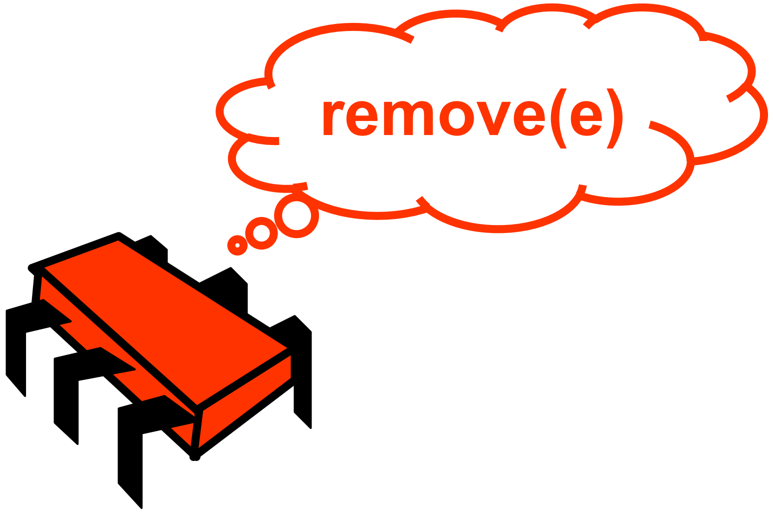
maintain a list to delete:



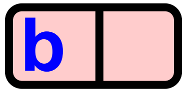
Our own garbage collector

Java's garbage collection will remove b

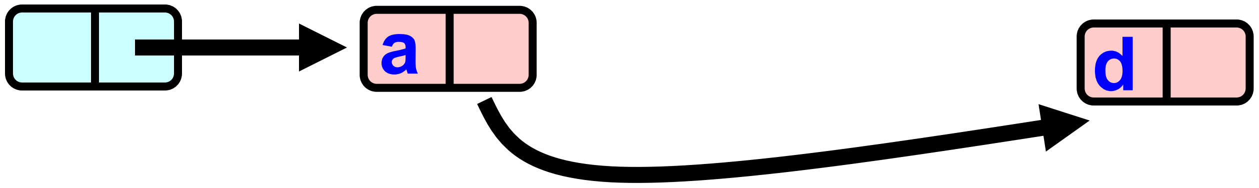
We are using a better™ language though...



maintain a list to delete:

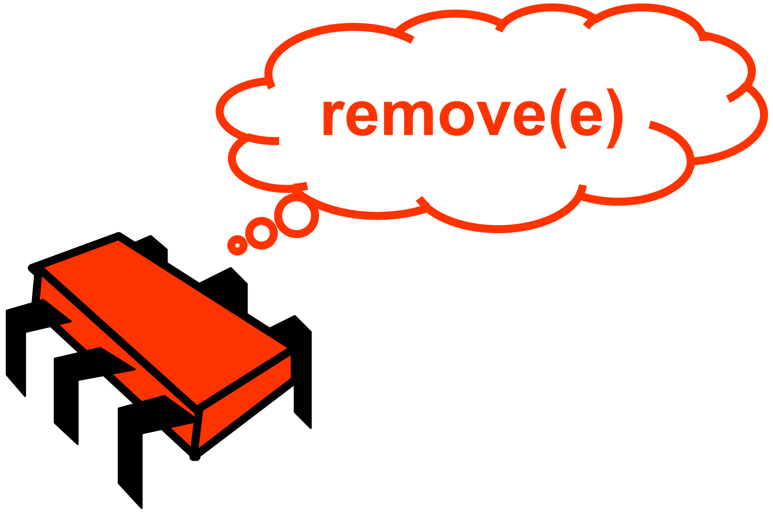


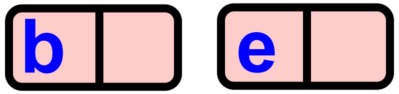
Our own garbage collector



Java's garbage collection will remove b

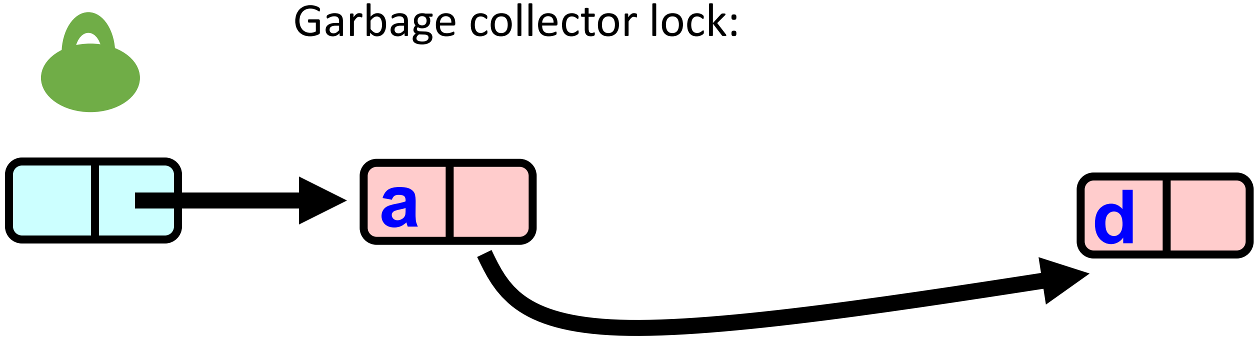
We are using a better™ language though...

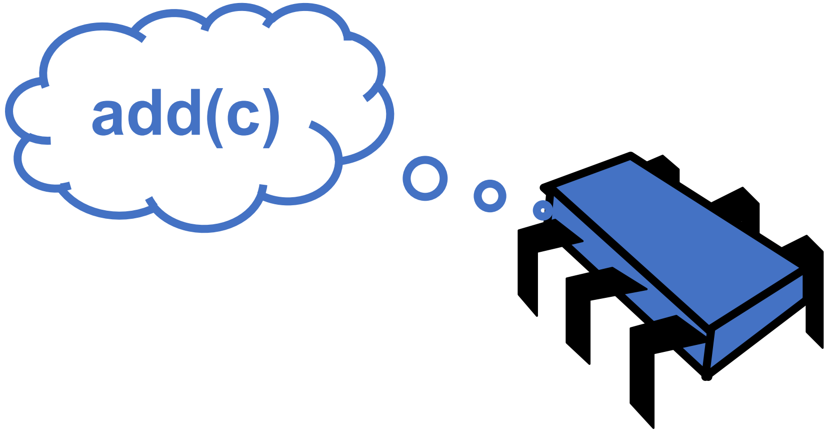


maintain a list to delete: 

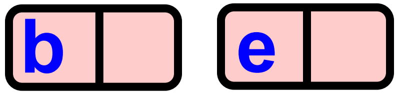
Our own garbage collector

Garbage collector lock:



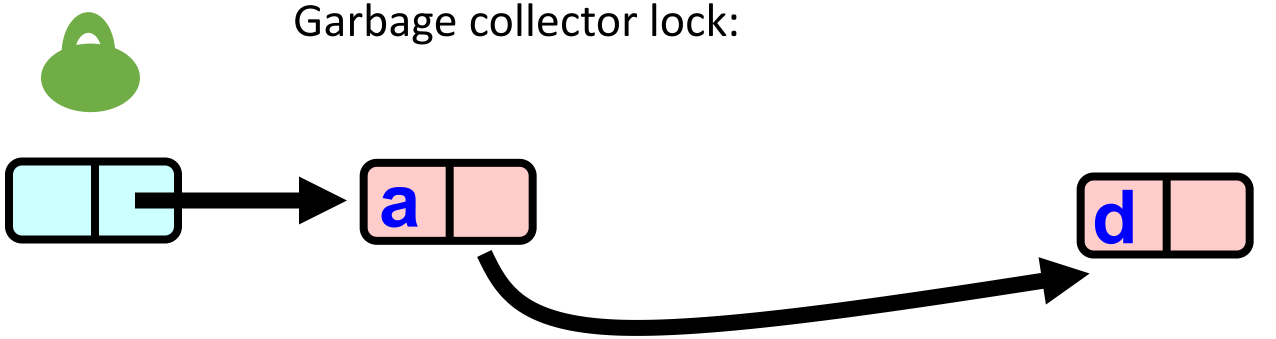


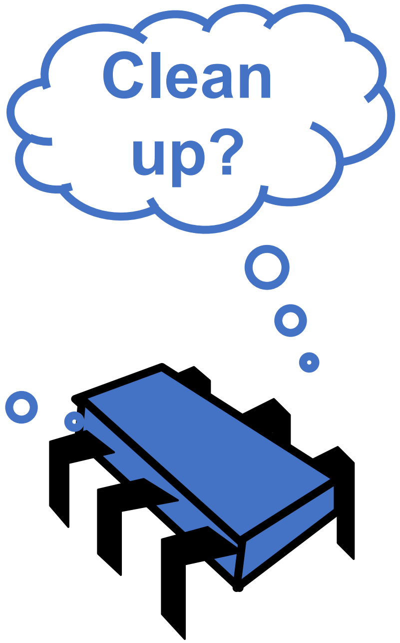
maintain a list to delete:



Our own garbage collector

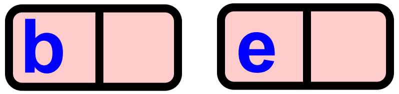
Garbage collector lock:





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

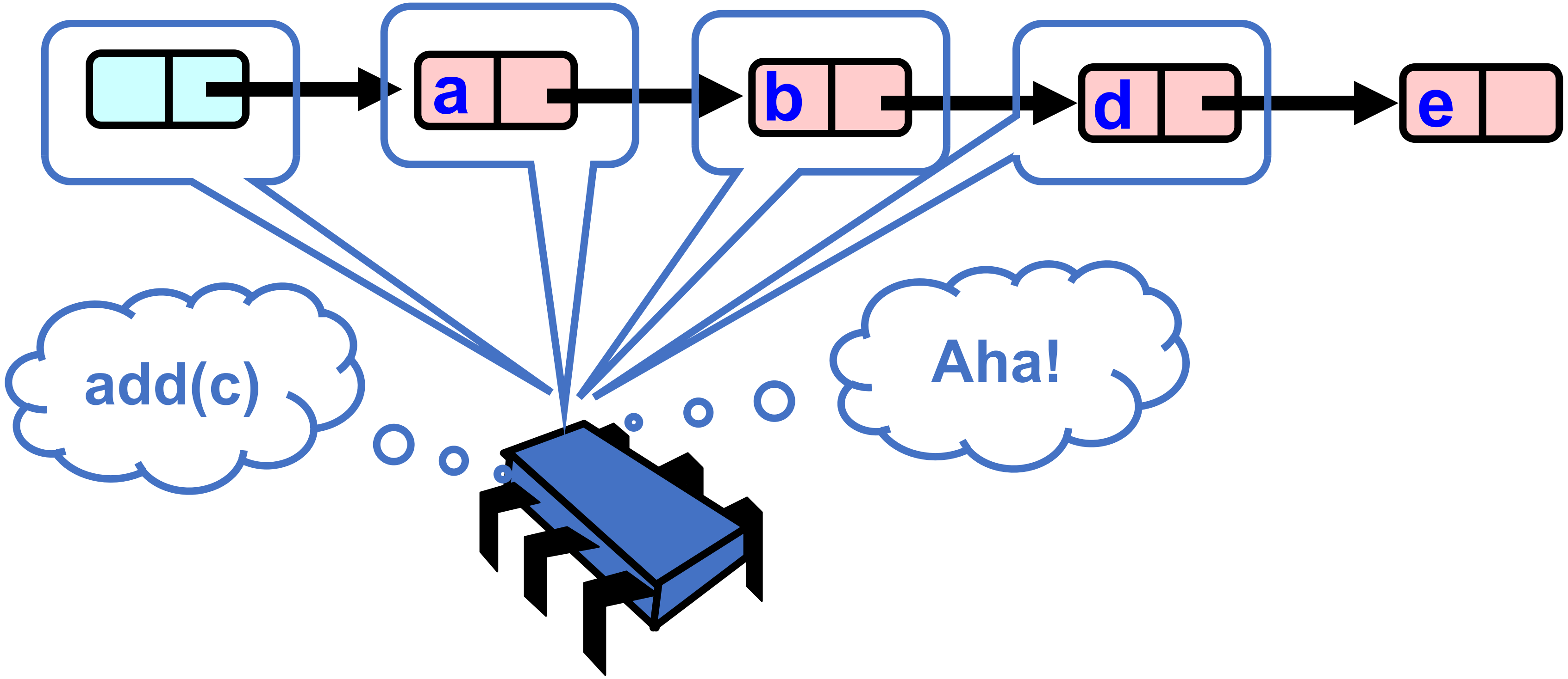
- **Strat 1:** Threads always try once to take the garbage collector lock:
  - if failed, no worries, the next operation will get a chance
  - if succeeded, then there was no contention
  - can starve garbage collection

- **Strat 2:** Wait until size grows to a threshold:
  - Wait on the lock (hope for a fair implementation!)
  - Can cause performance spikes

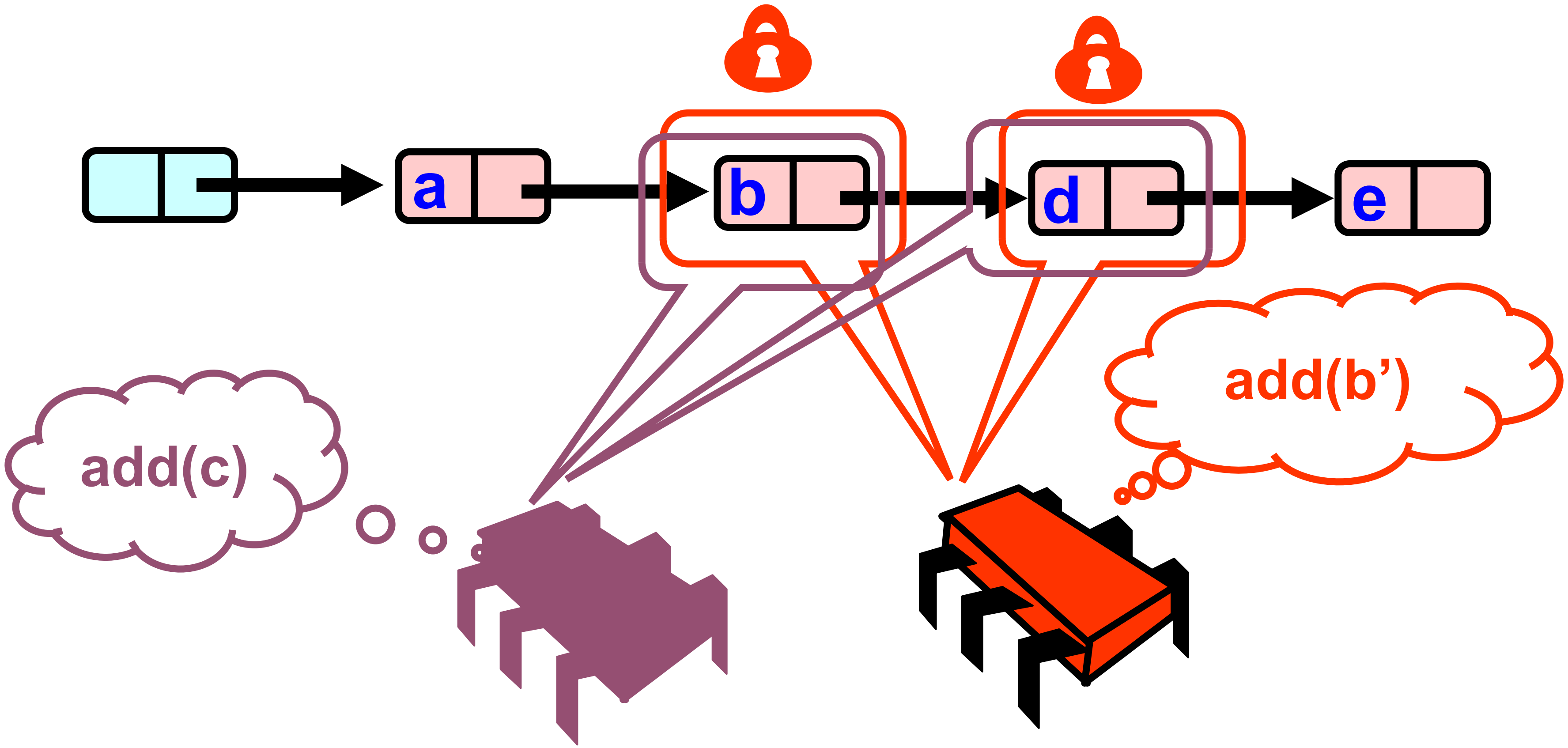
Back to the linked list

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

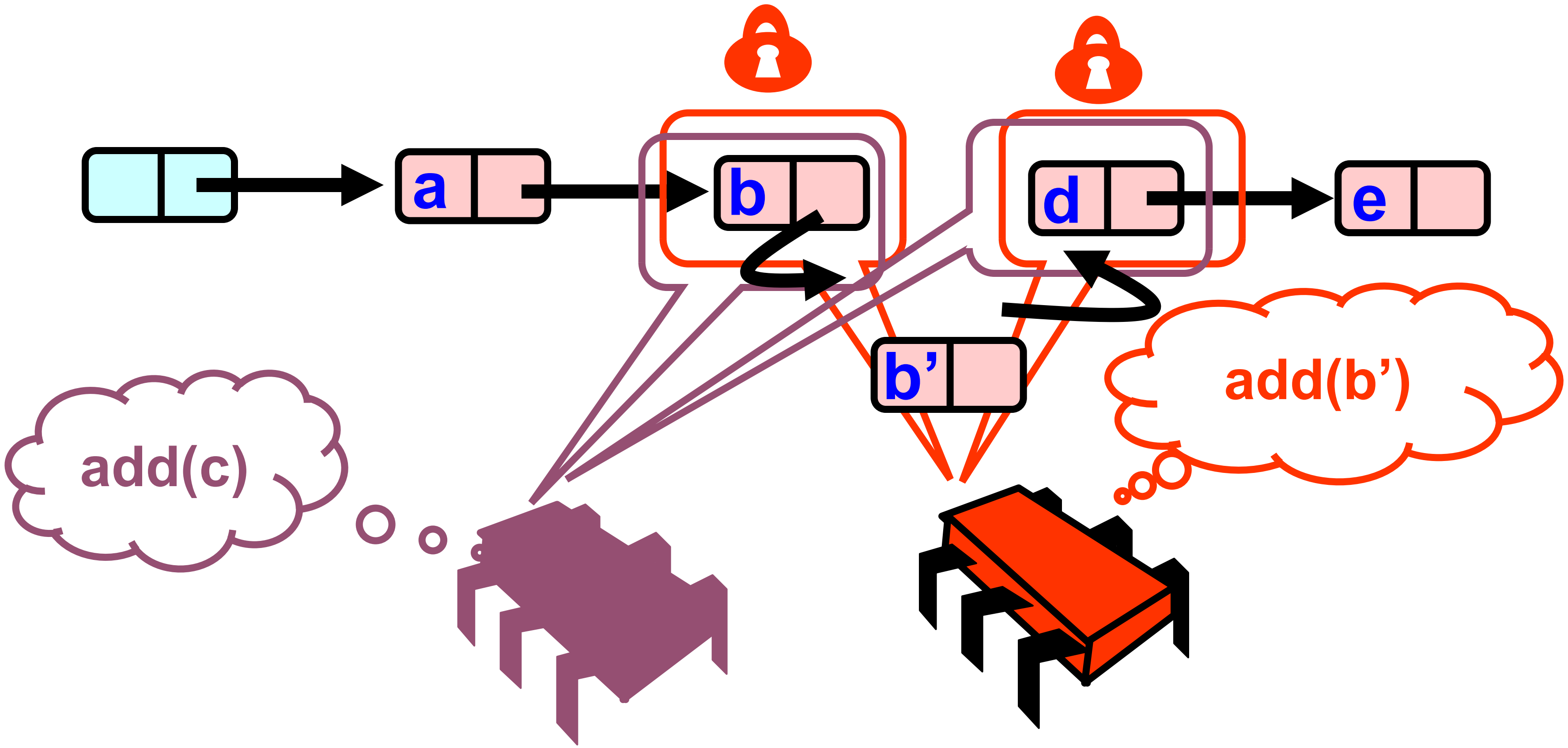
What Else Could Go Wrong?



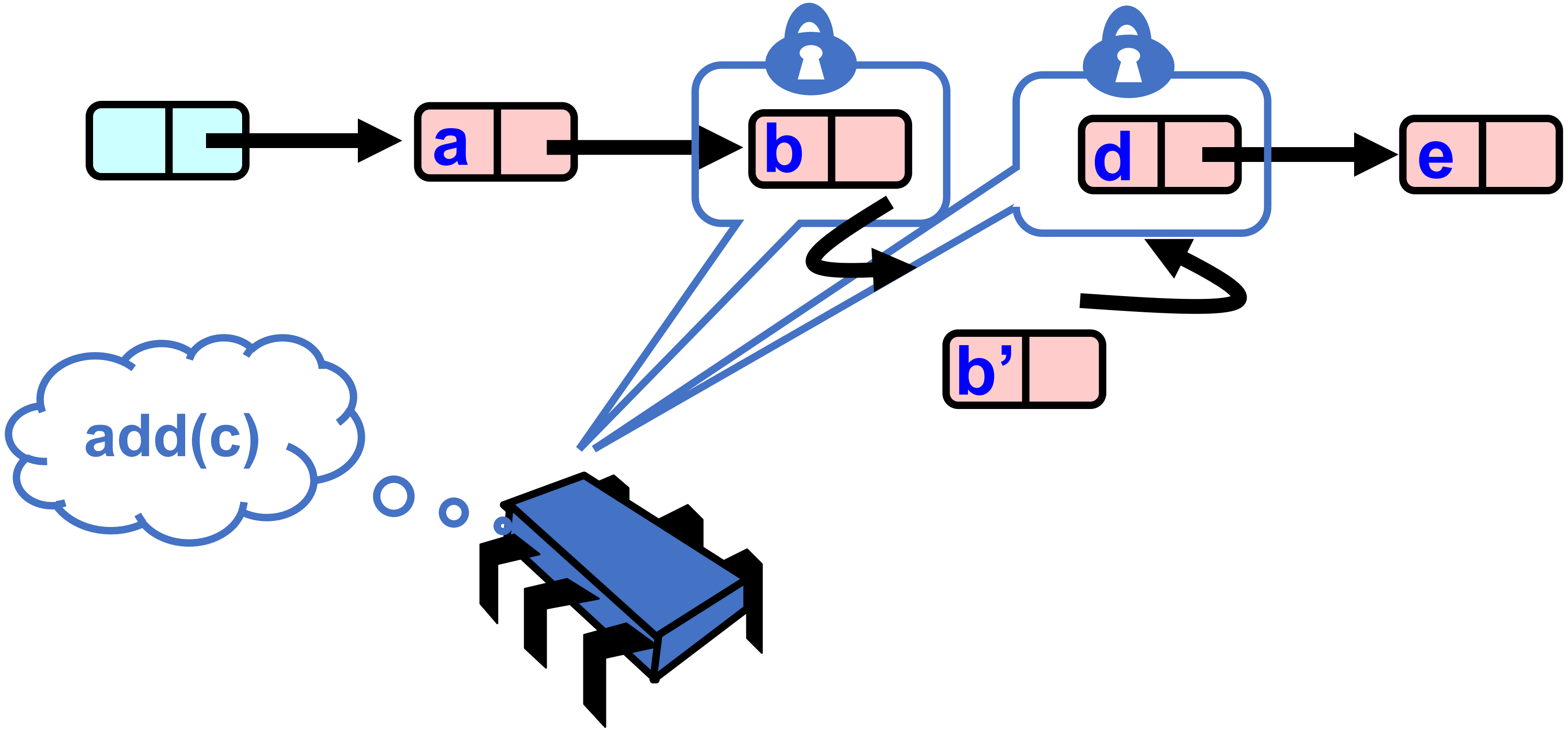
What Else Coould Go Wrong?



What Else Coould Go Wrong?

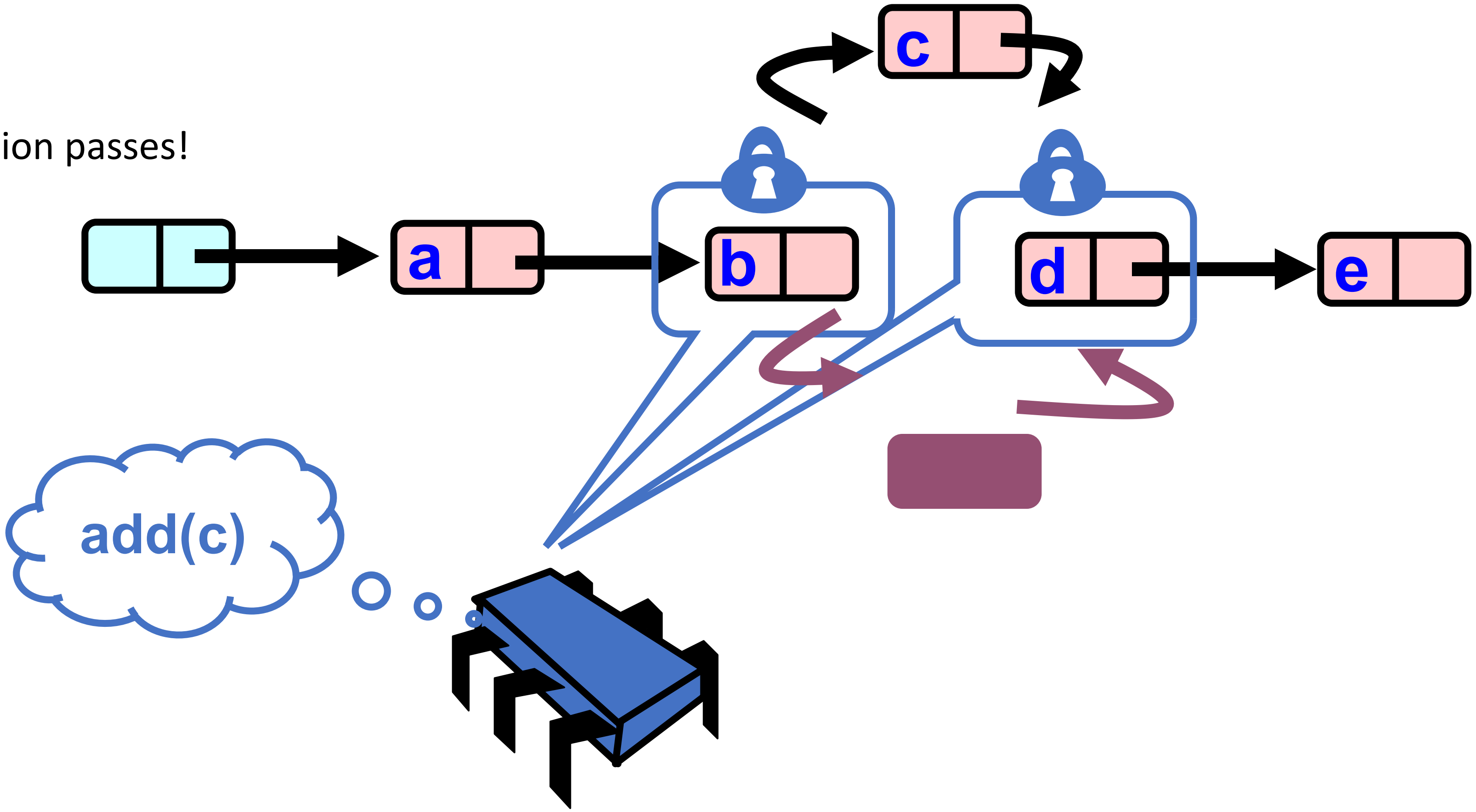


What Else Could Go Wrong?

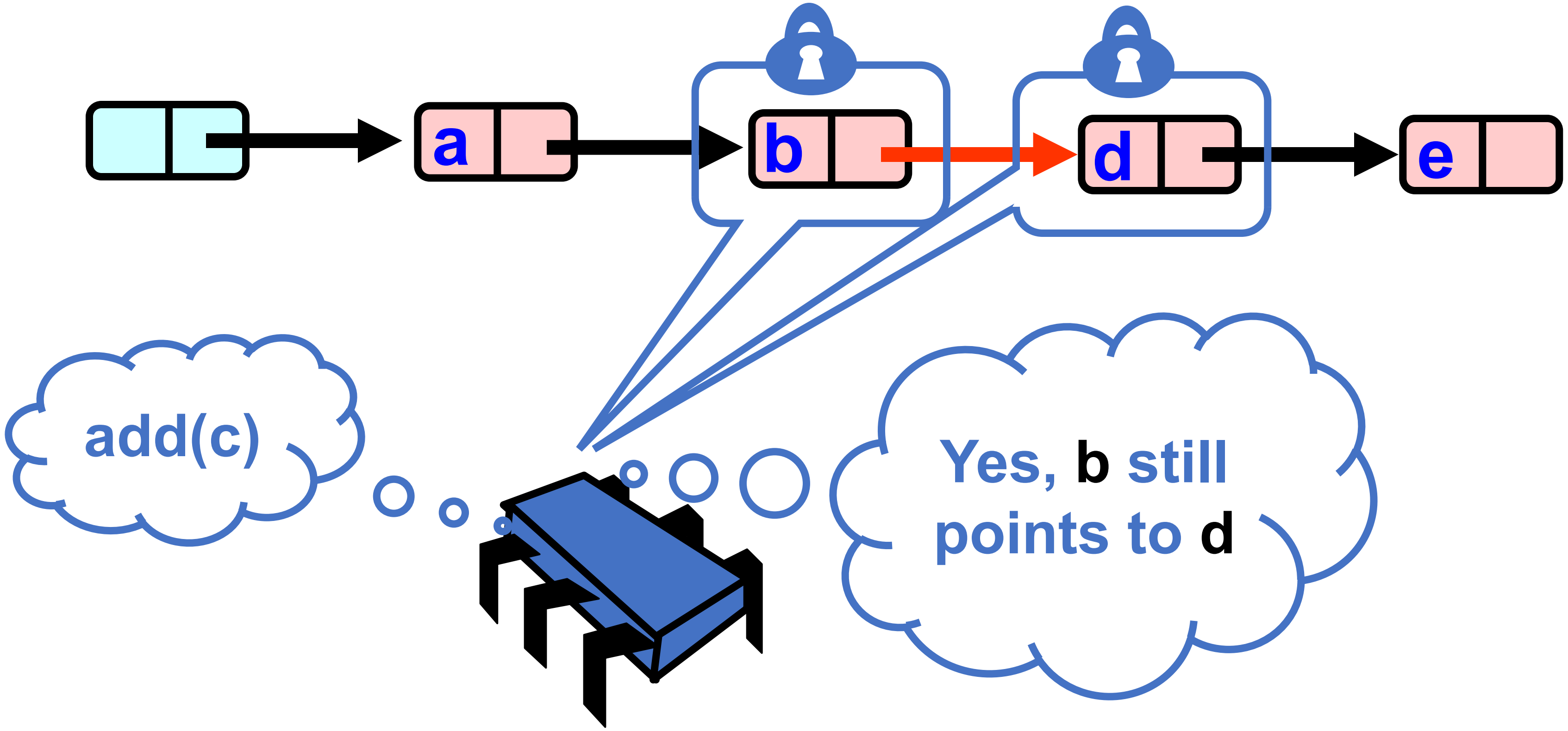


What Else Could Go Wrong?

Validation passes!



Validate Part 2
(while holding locks)



Pause for a breath of air

- 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

Further reading on optimistic list

- Implementation details in the book
- Arguments about linearizability points

How can we improve

- Most operations require two traversals:
    - One to find the interesting point
    - Take the locks
    - and Another pass to validate

Schedule

- Review linked list set interface
- Optimistic locking implementation
- **Two-step remove implementation (lazy deletion)**
- Lock free implementation

Schedule

- 5 minute break:

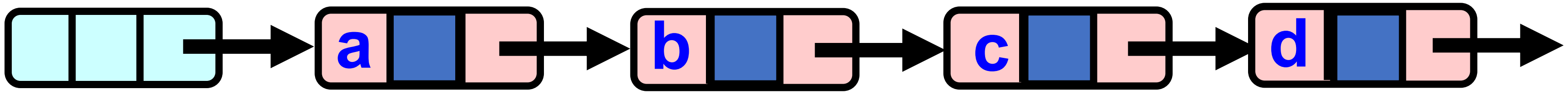
Two step removal (lazy list)

- Like optimistic, except
  - Scan once
- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”

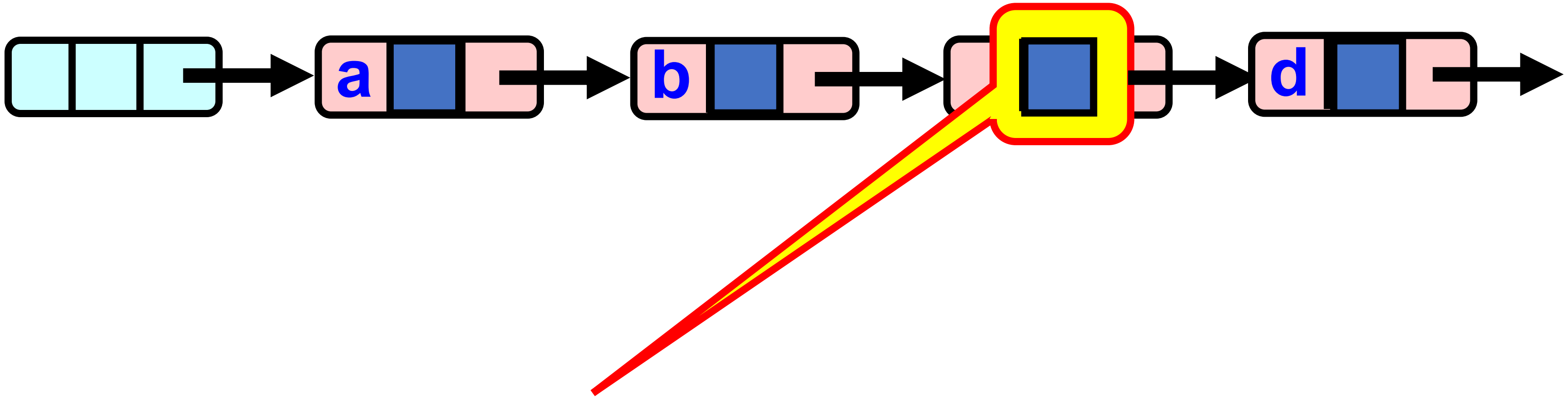
Two step removal List

- **`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

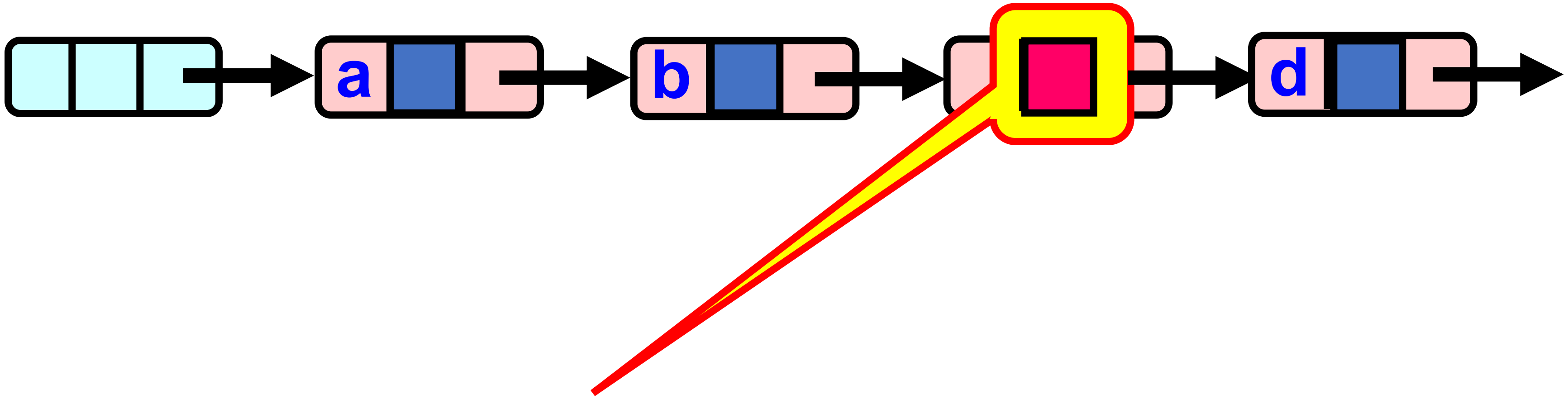


Two step removal Removal



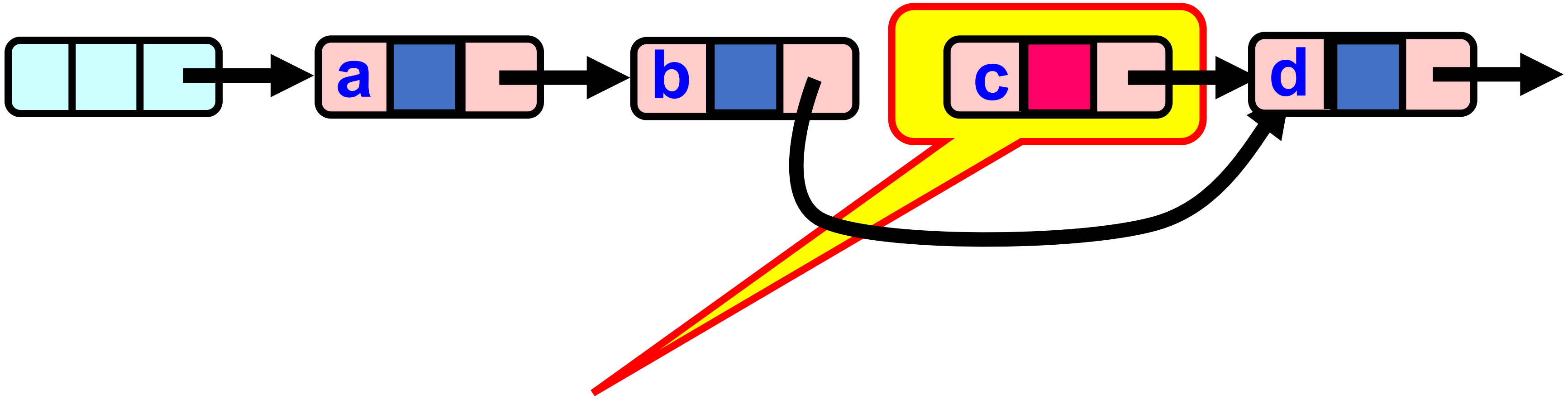
Present in list

Two step removal Removal



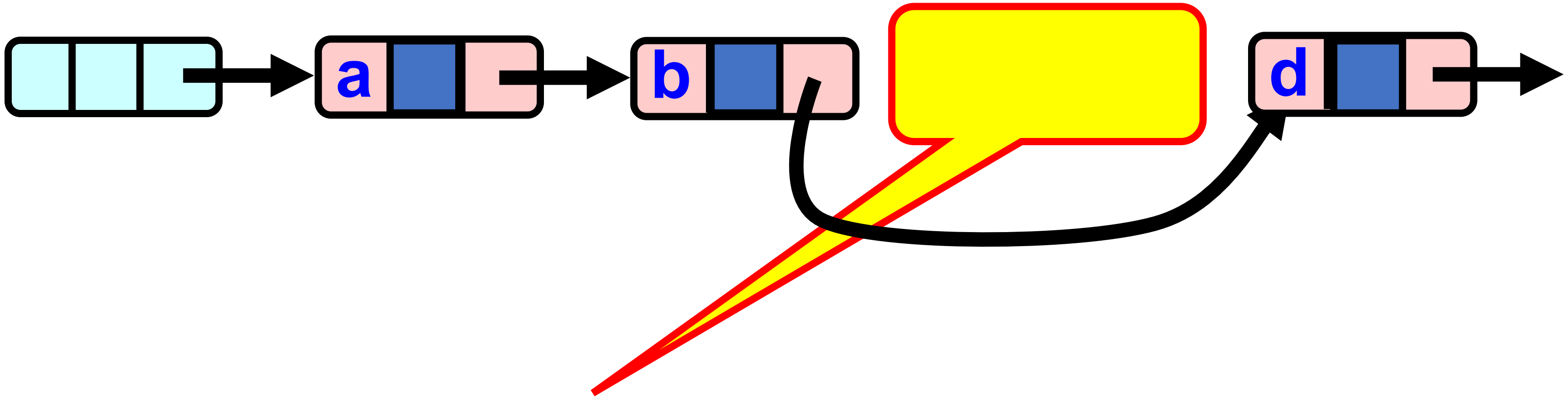
Logically deleted

Two step removal Removal



Physically deleted

Two step removal Removal



Physically deleted

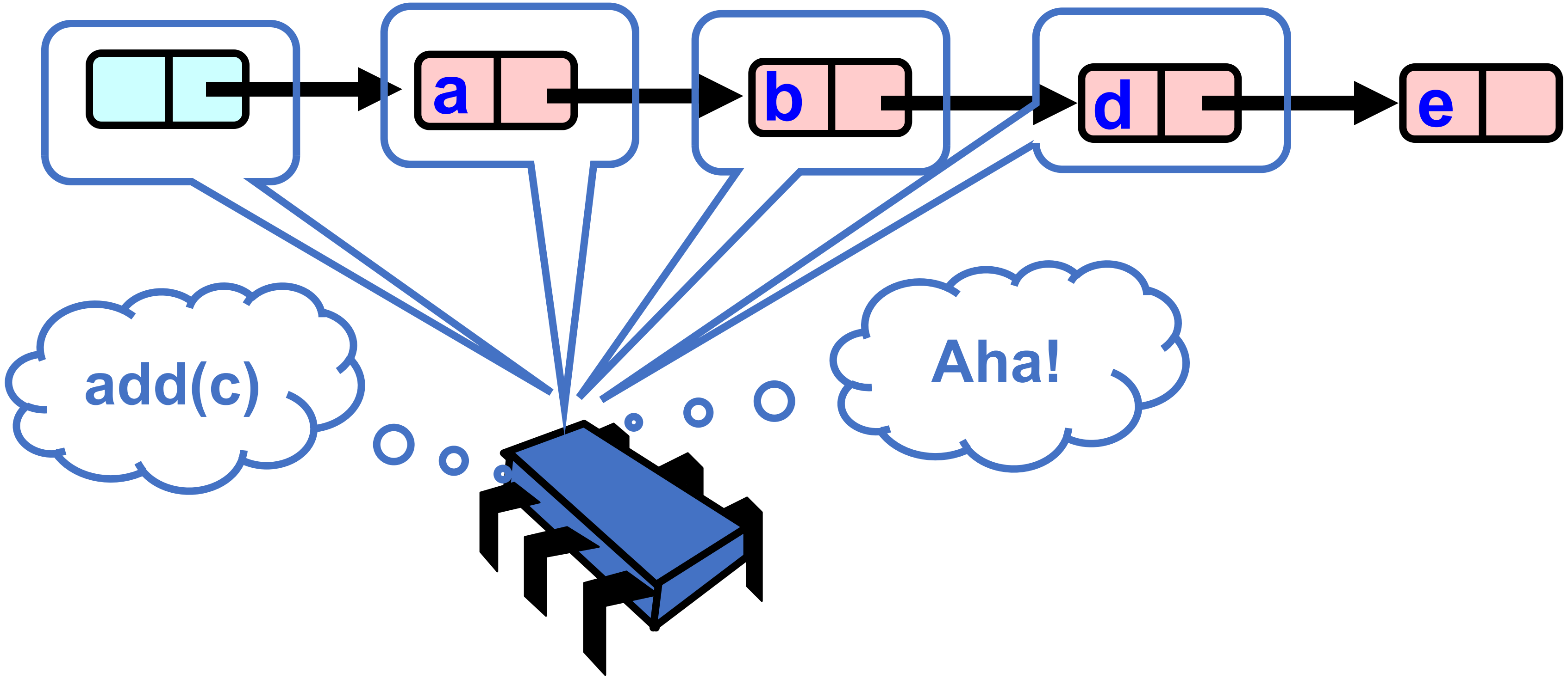
Lazy List

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

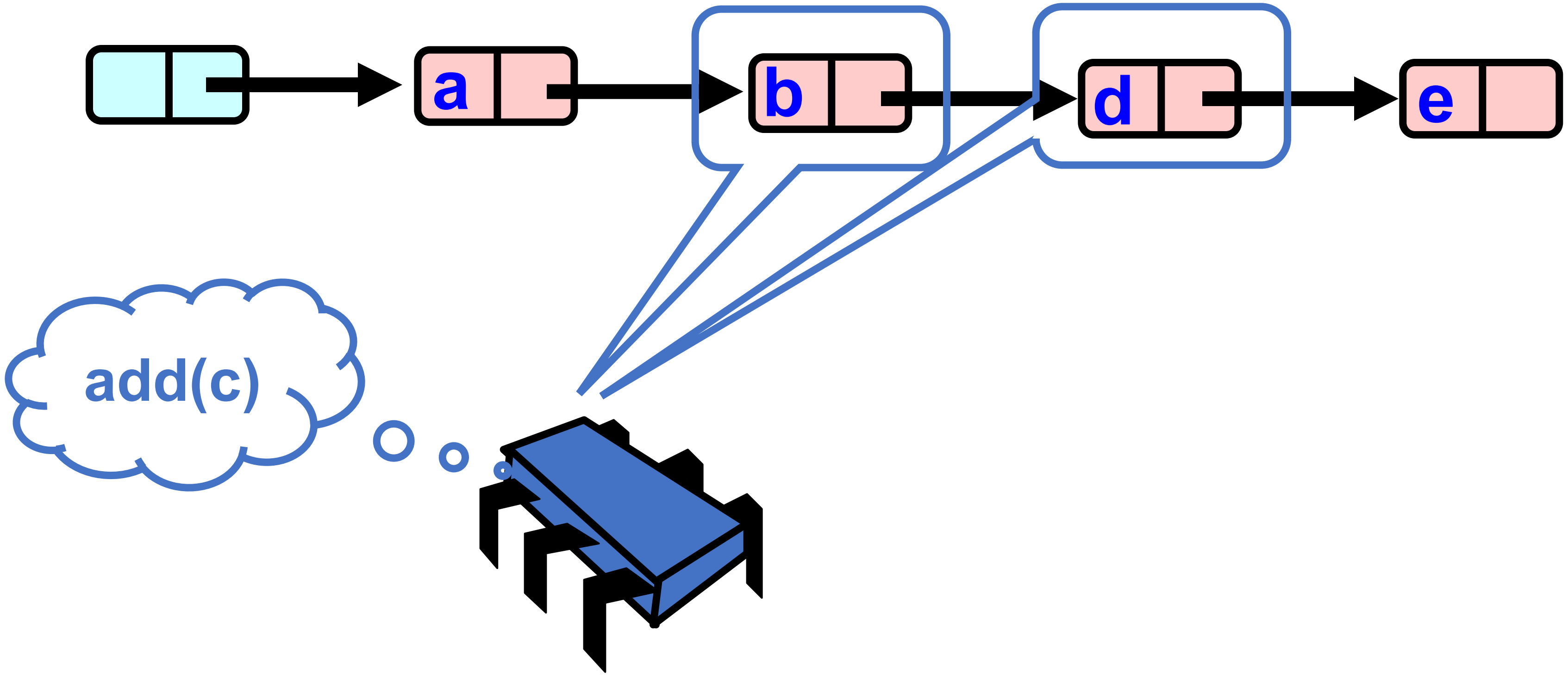
Validation

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

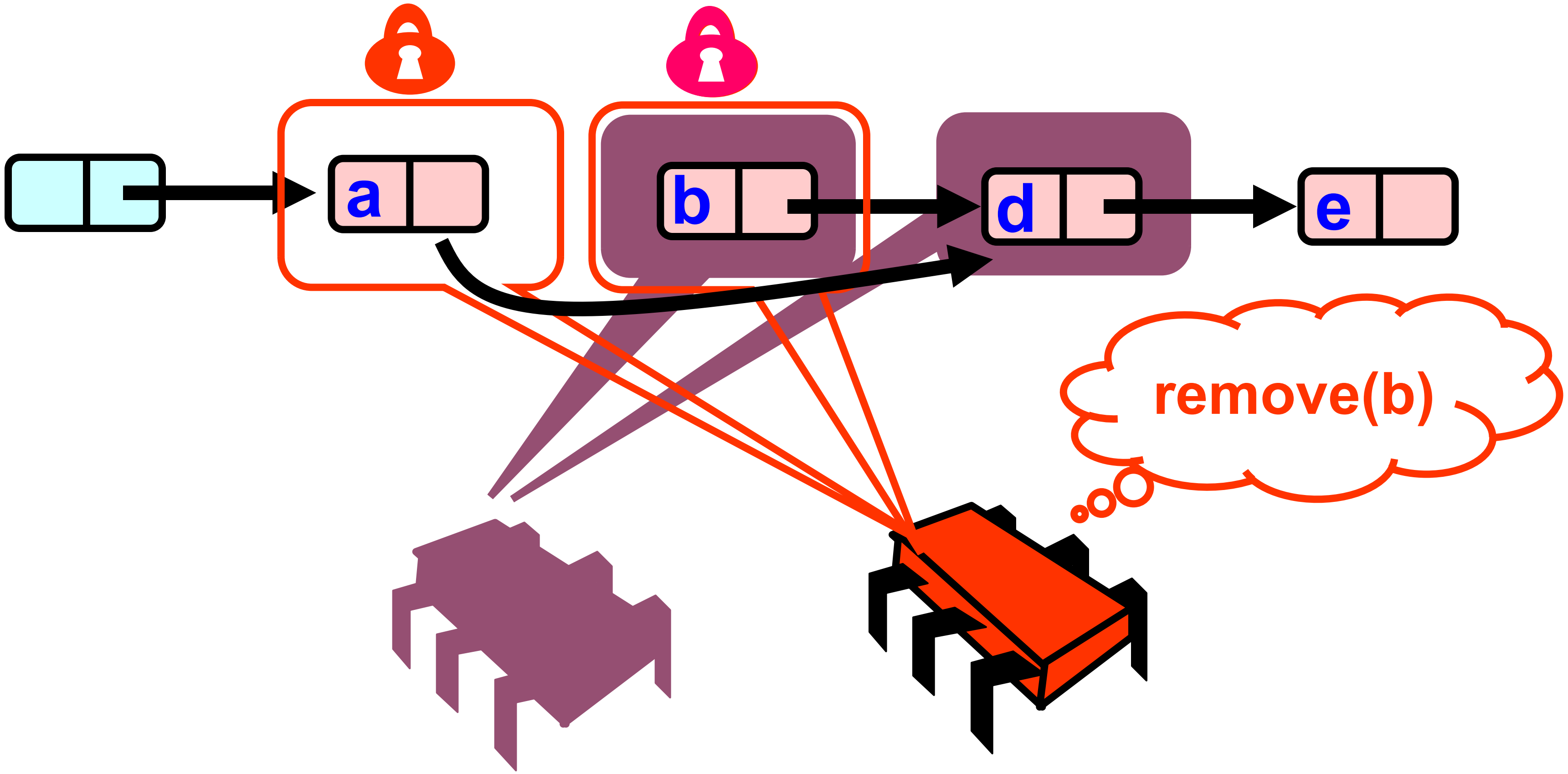
What could go wrong?



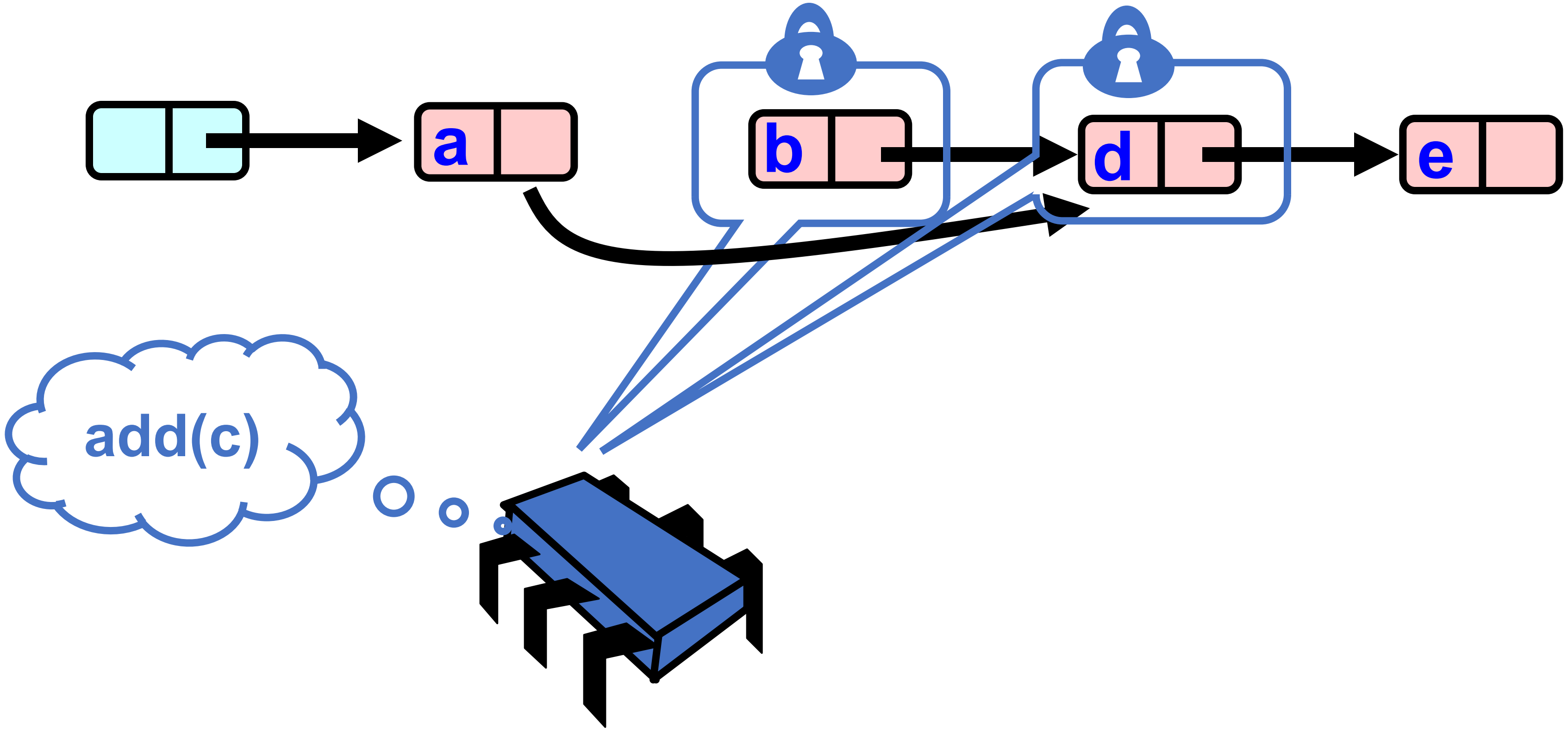
What could go wrong?



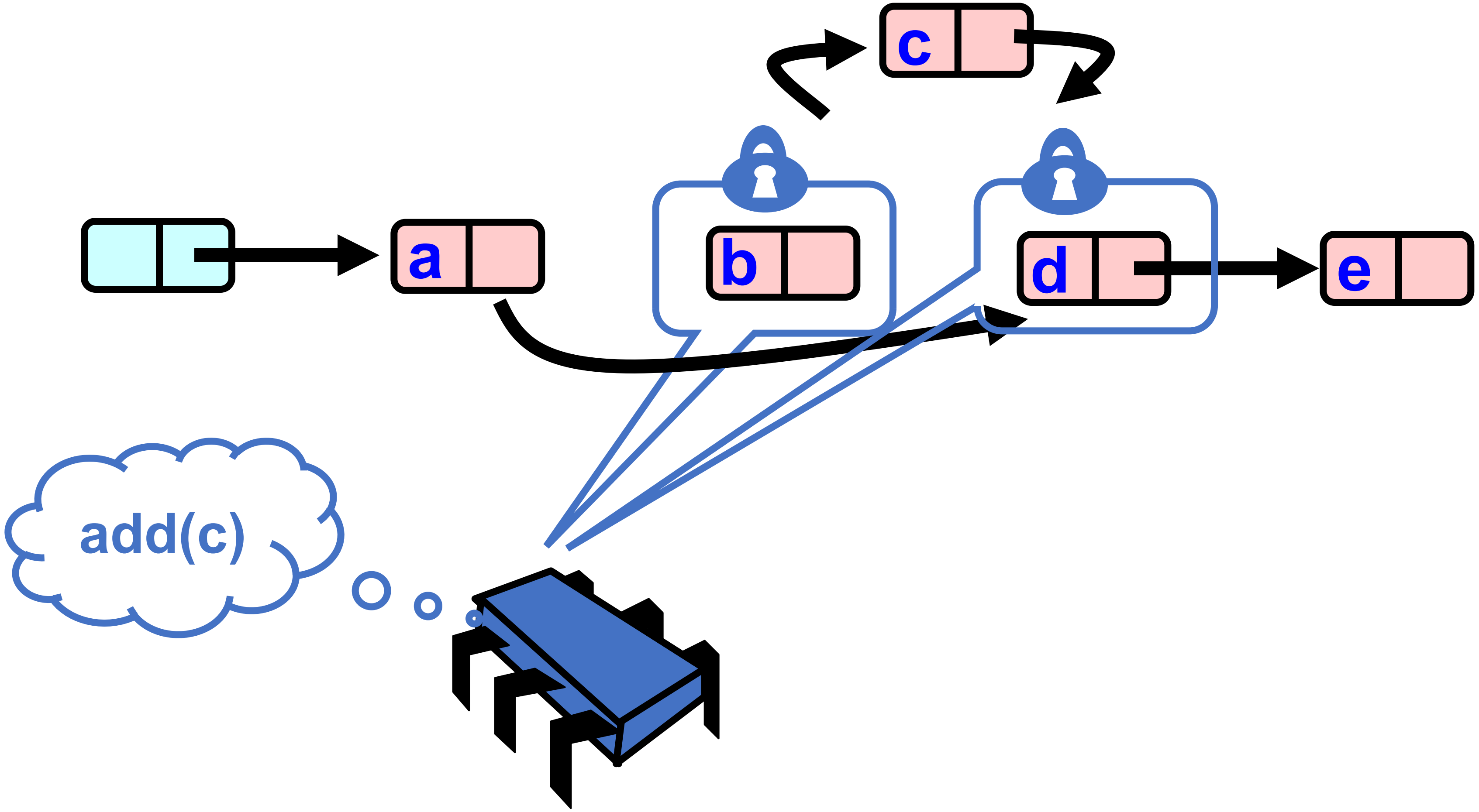
What could go wrong?



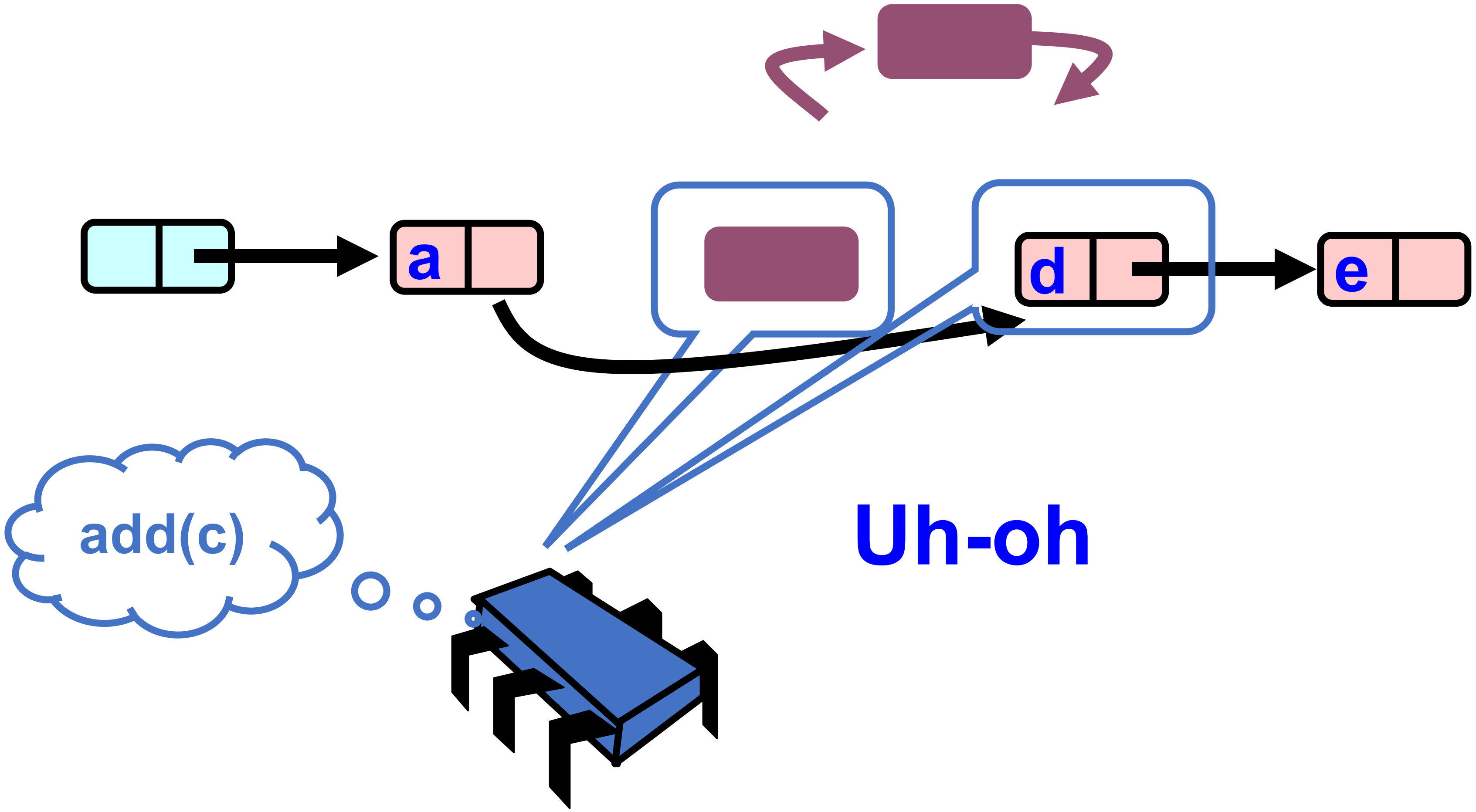
What could go wrong?



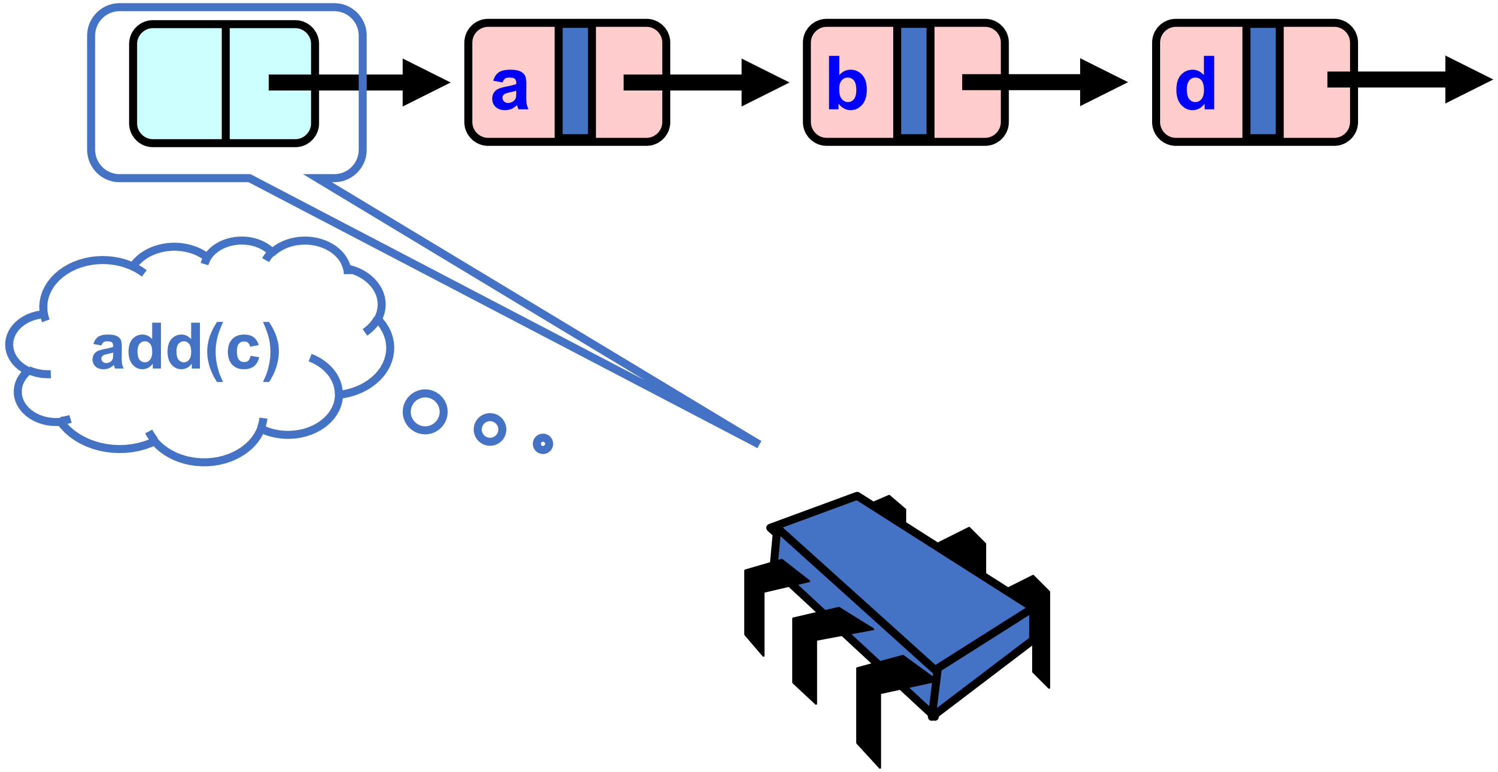
What could go wrong?



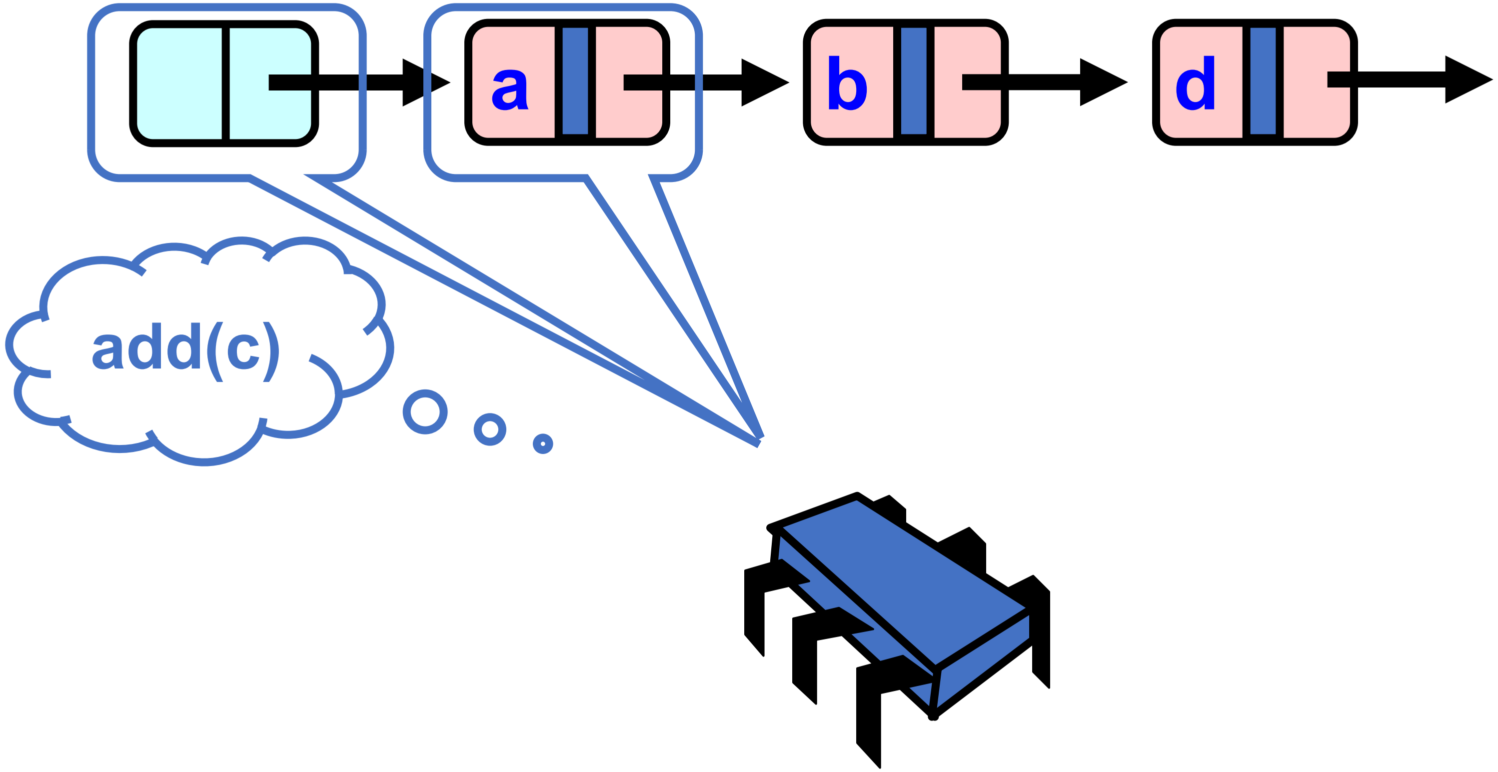
What could go wrong?



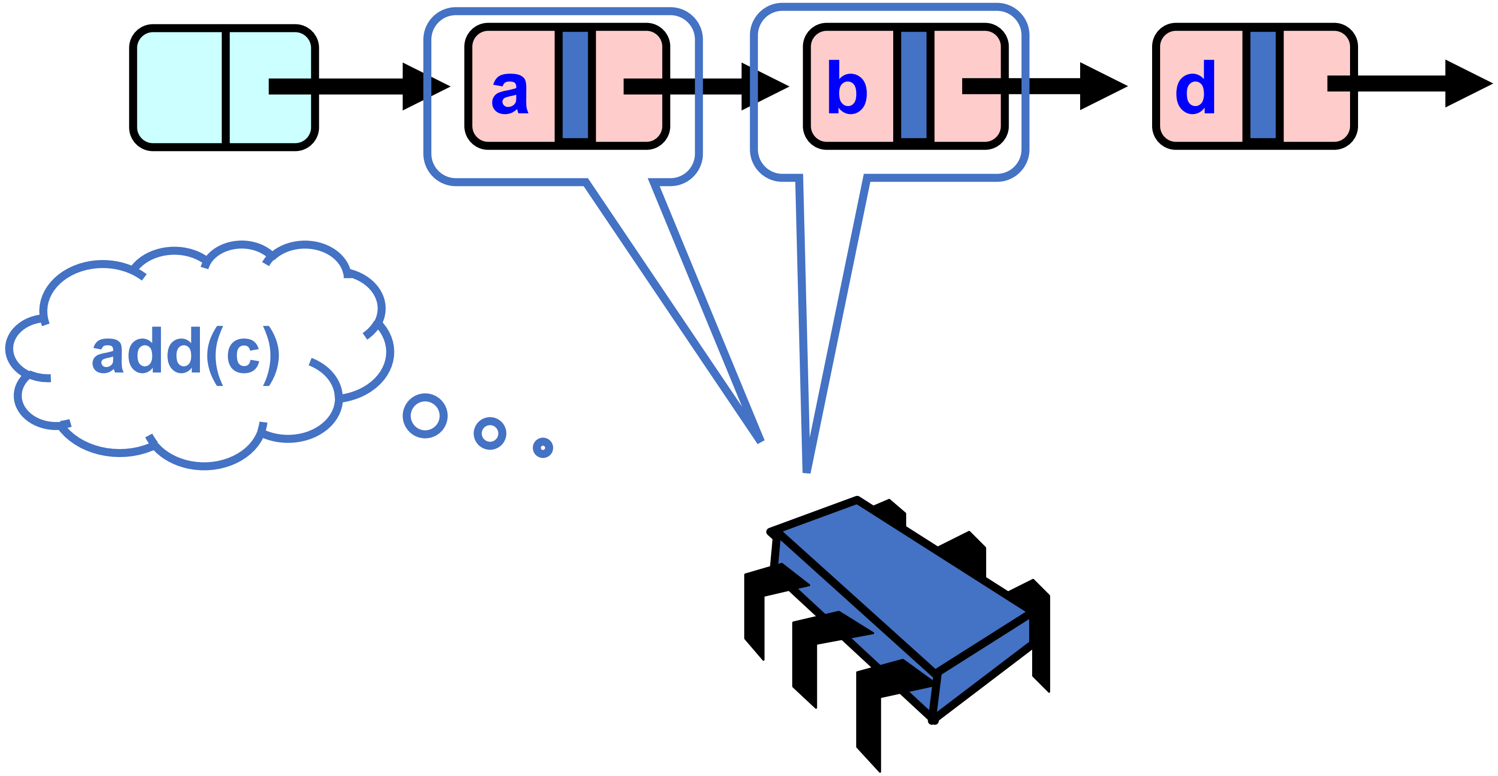
Fixed with logical flag



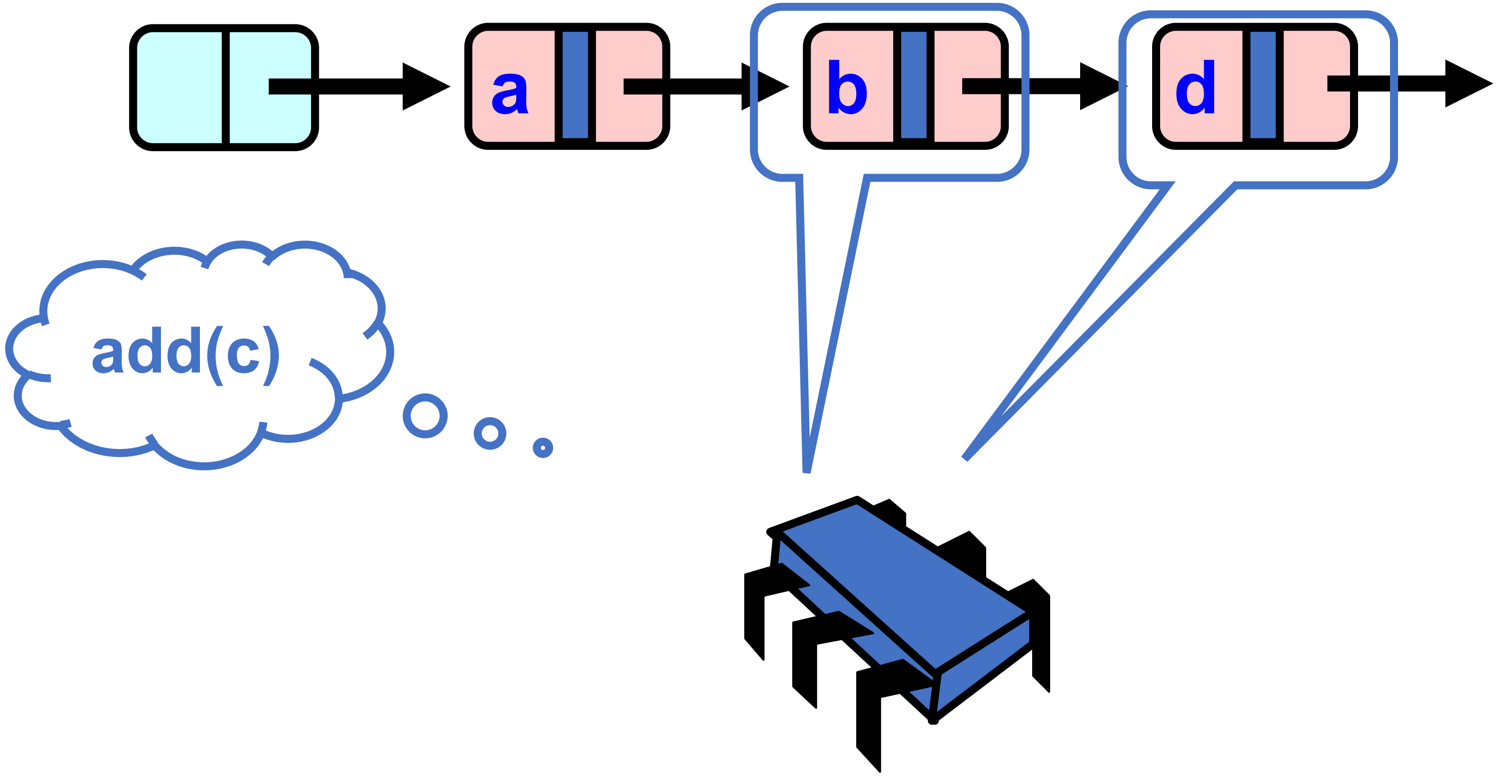
Fixed with logical flag



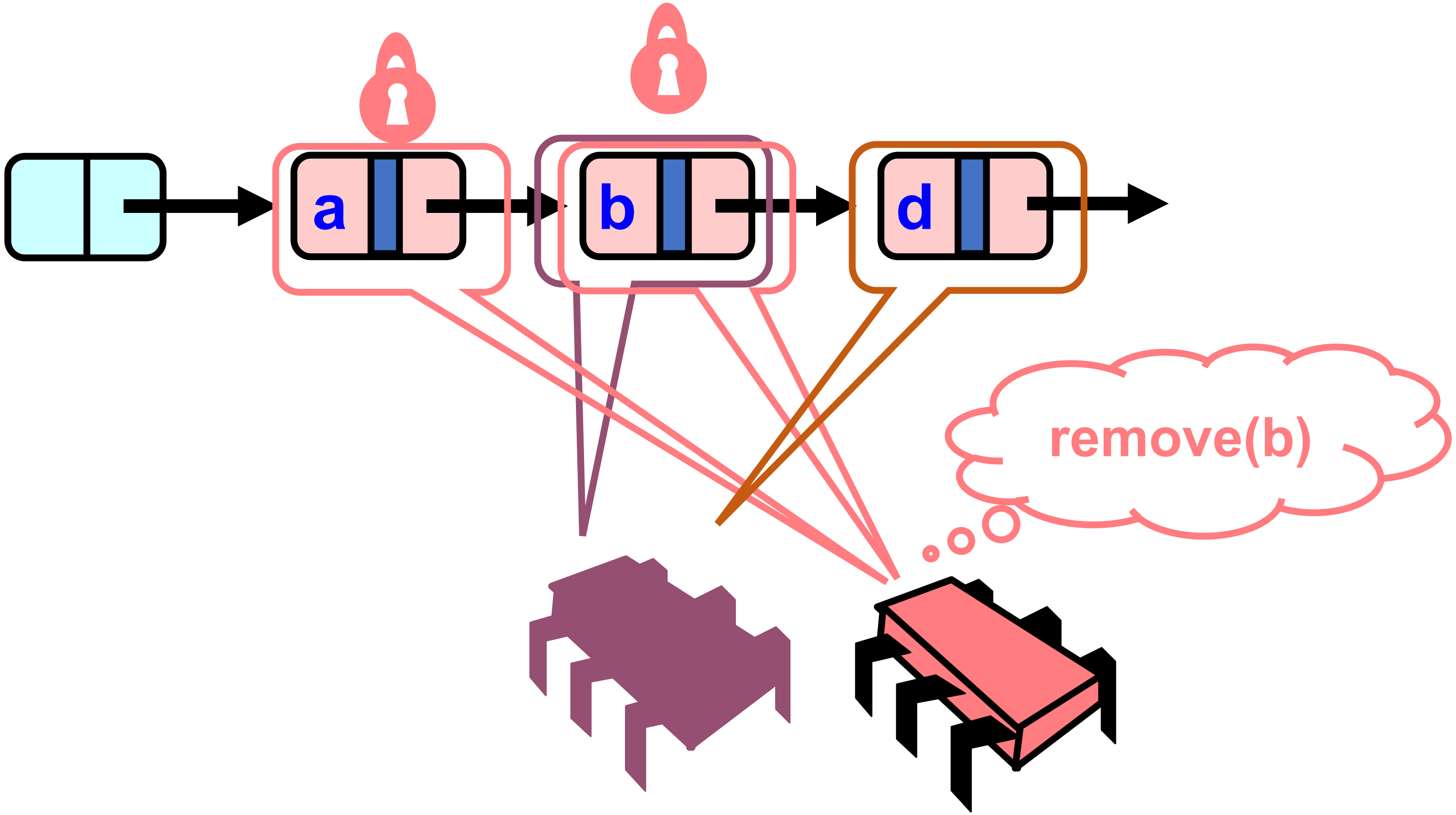
Fixed with logical flag



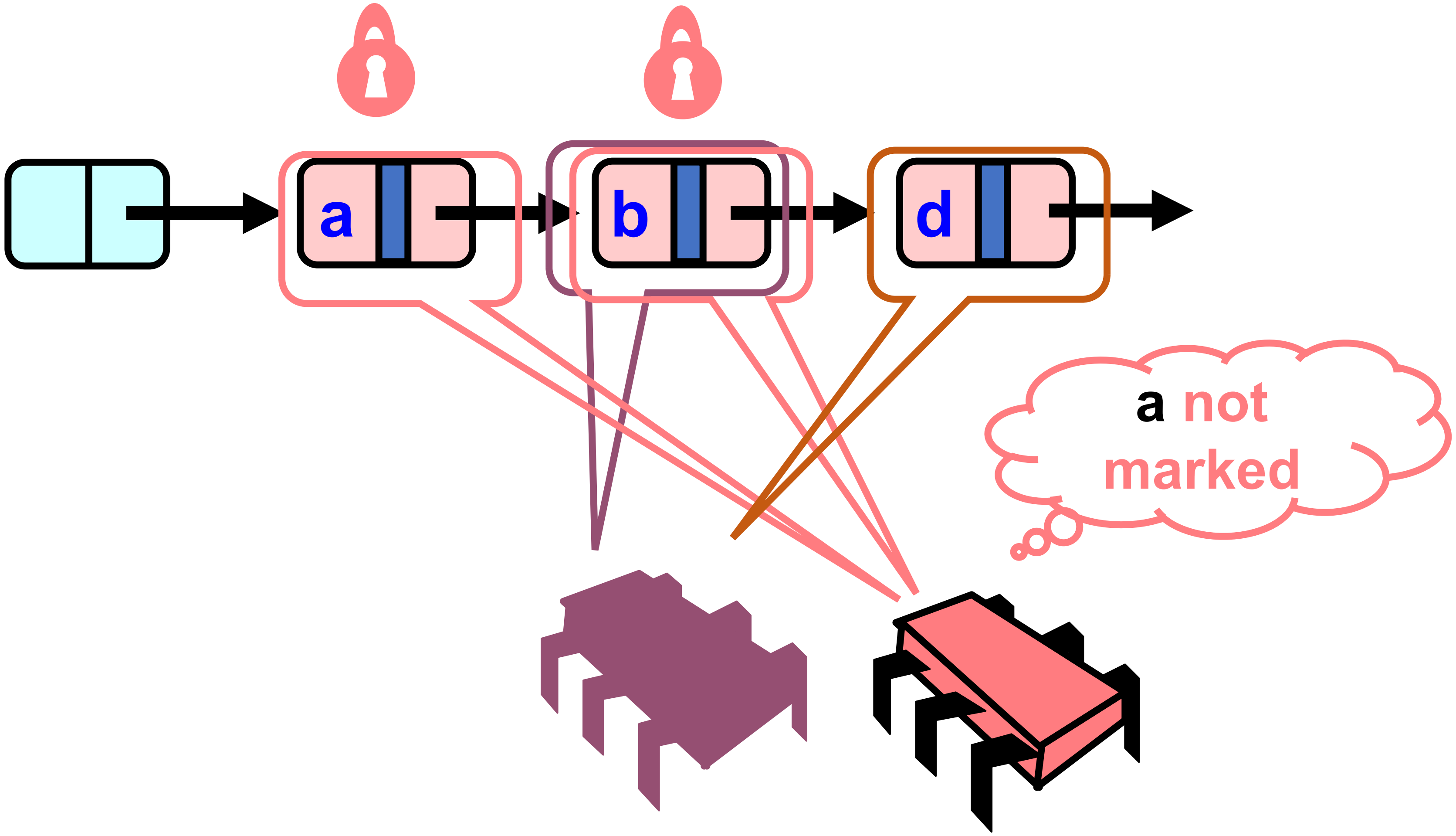
Fixed with logical flag



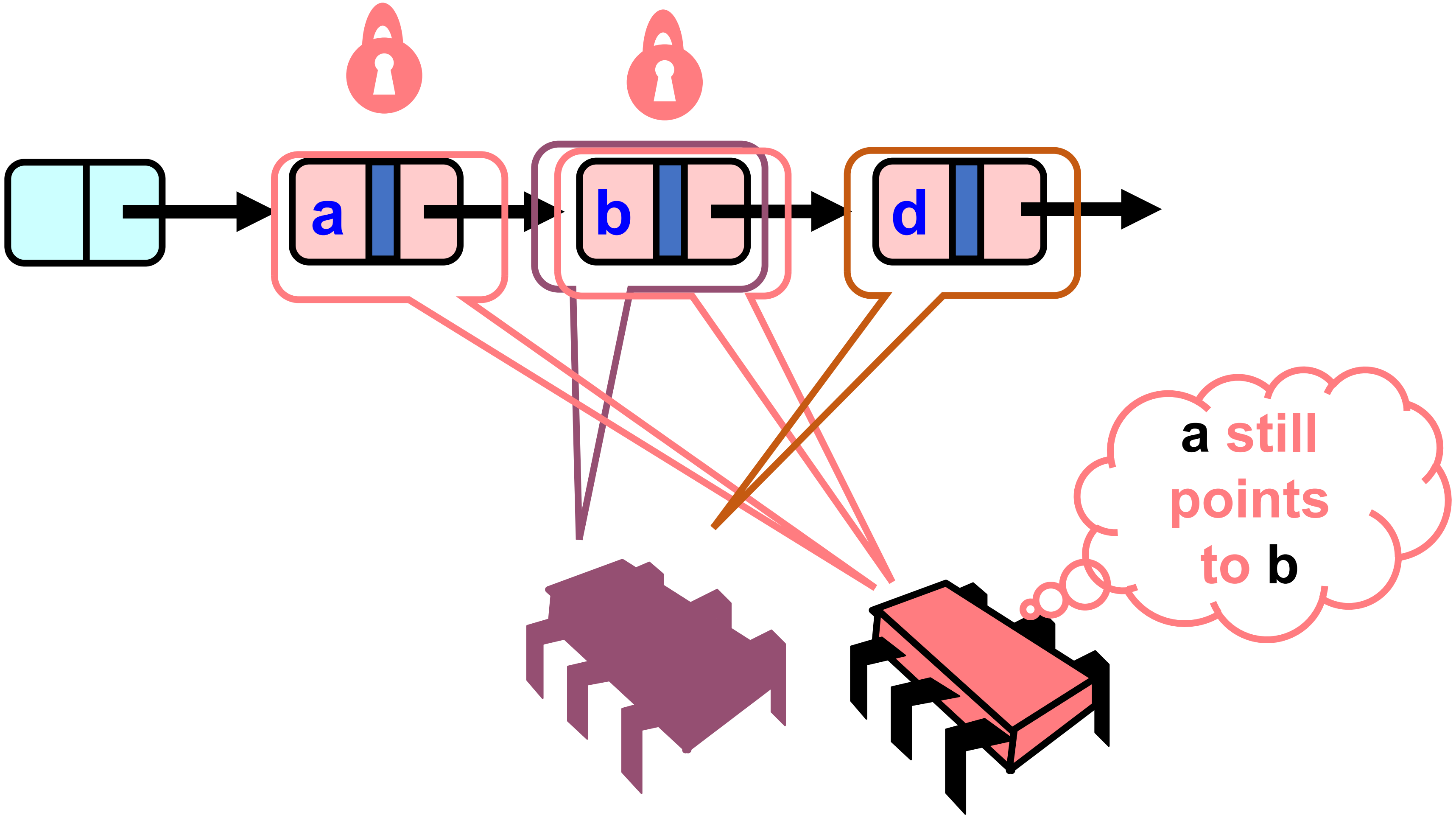
Fixed with logical flag



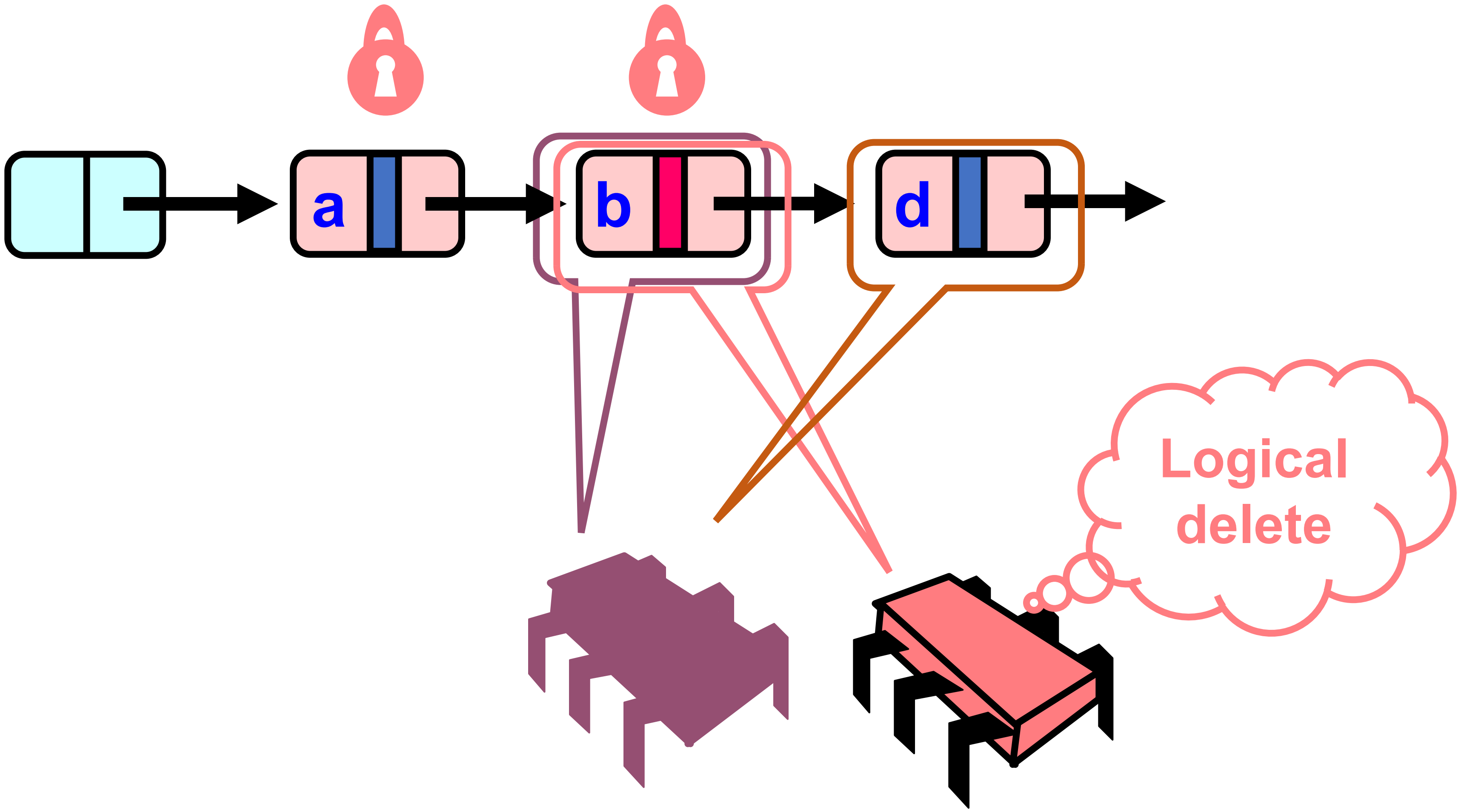
Fixed with logical flag



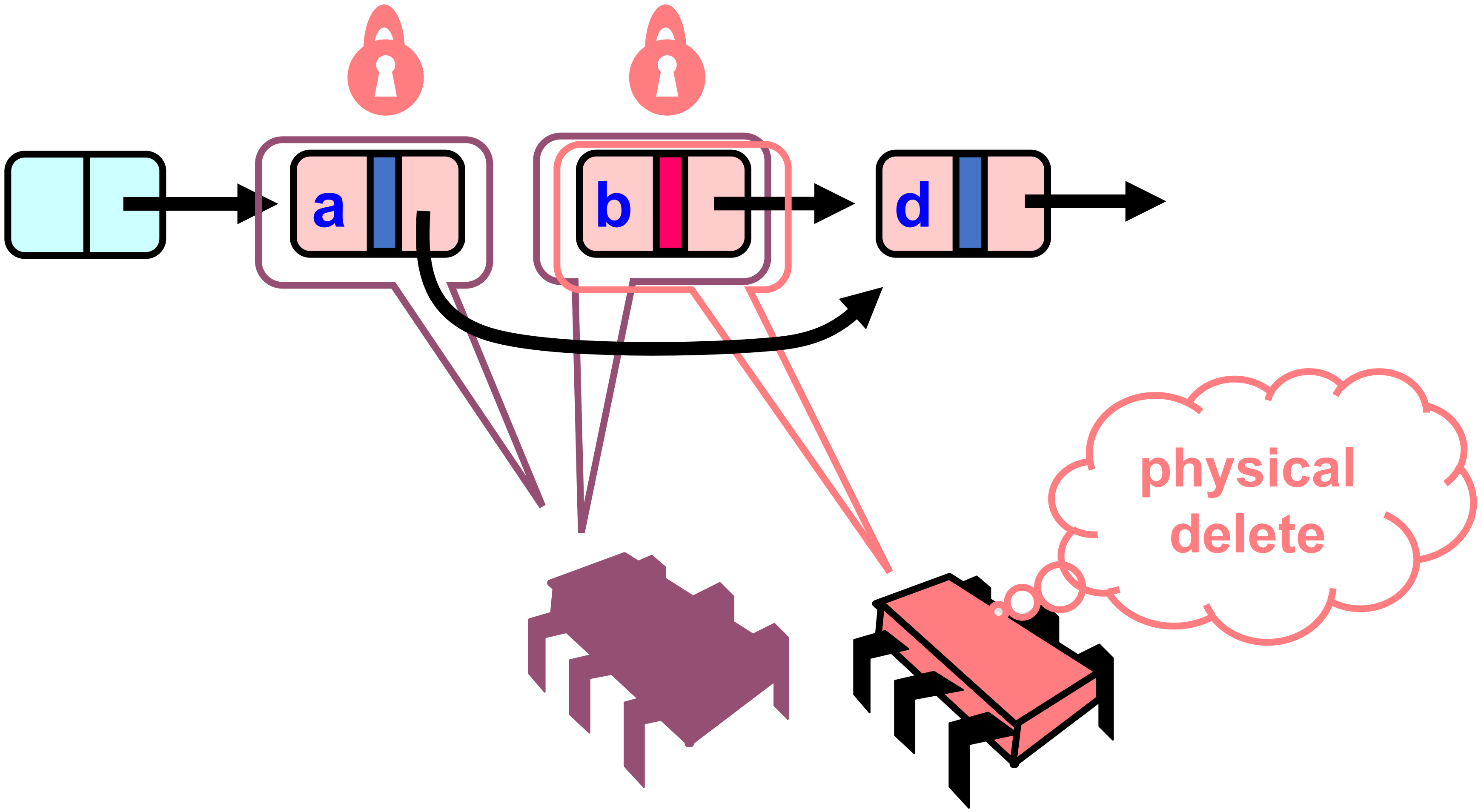
Fixed with logical flag



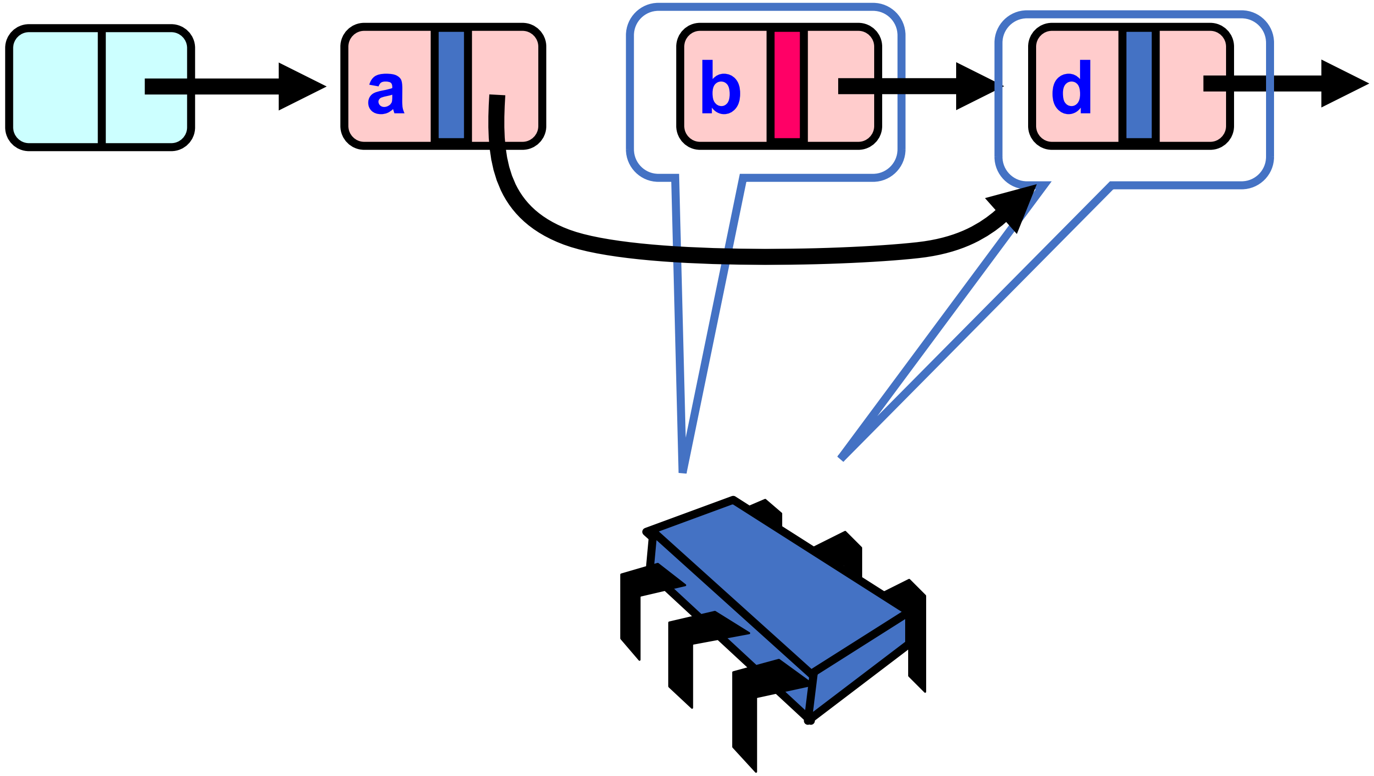
Fixed with logical flag



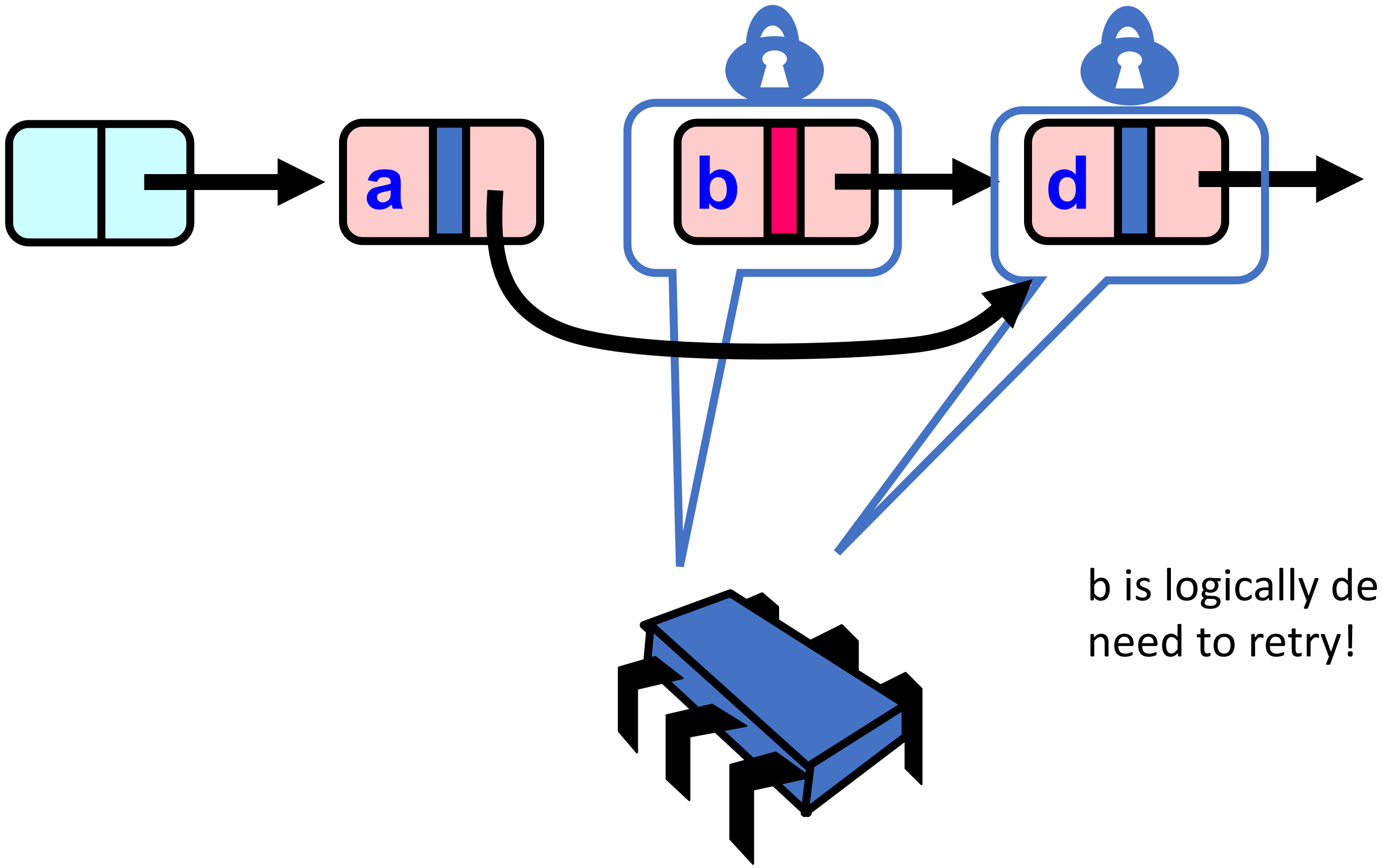
Fixed with logical flag



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

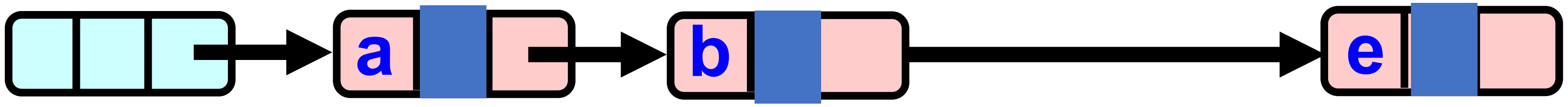
Schedule

- Review linked list set interface
- Optimistic locking implementation
- Two-step remove implementation (lazy deletion)
- **Lock free implementation**

Lock-free Lists

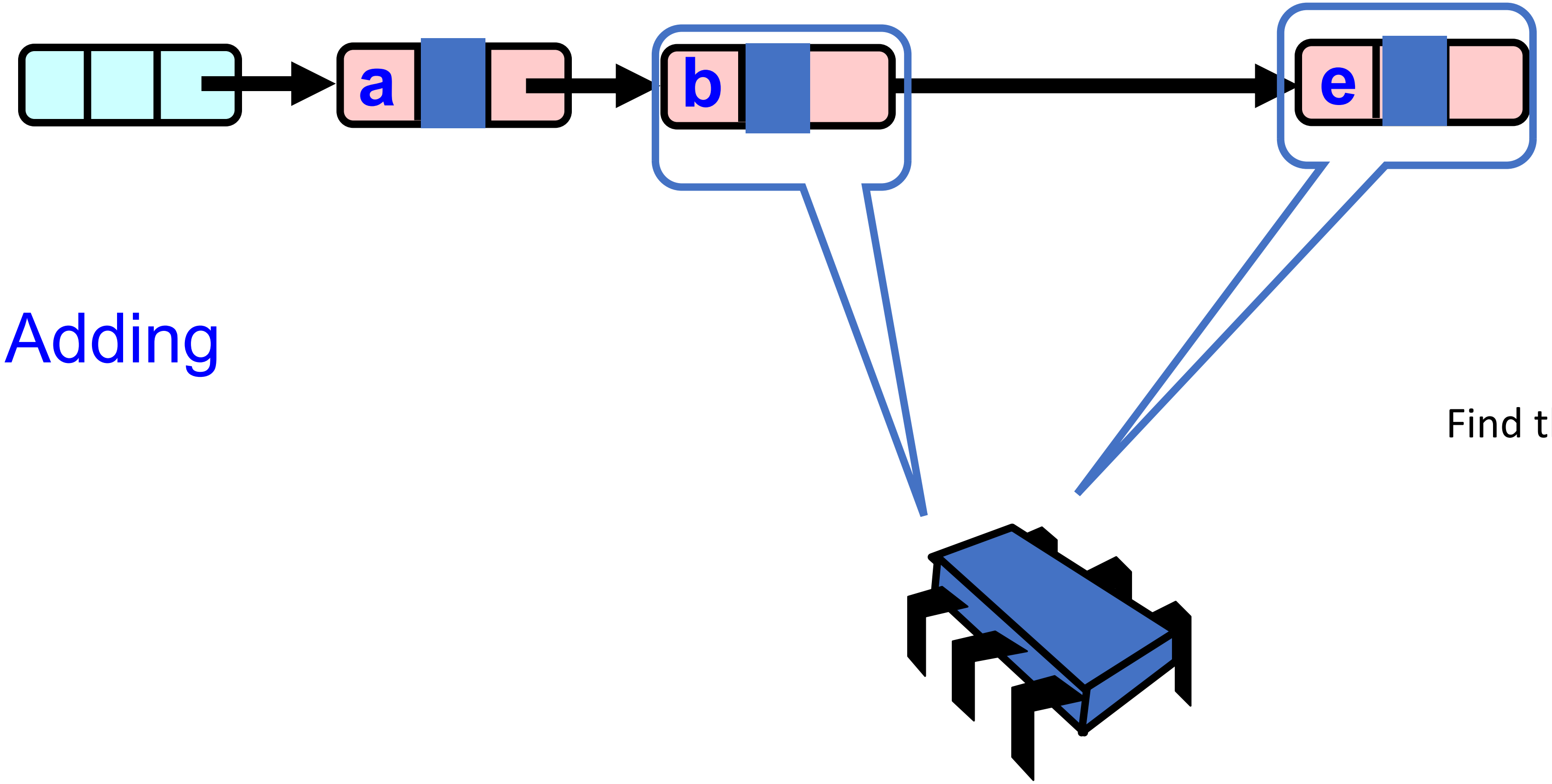
- Next logical step
  - lock-free add() and remove()
- What sort of atomics do we need?
  - Loads/stores?
  - RMWs?

Lock-free Lists



Adding

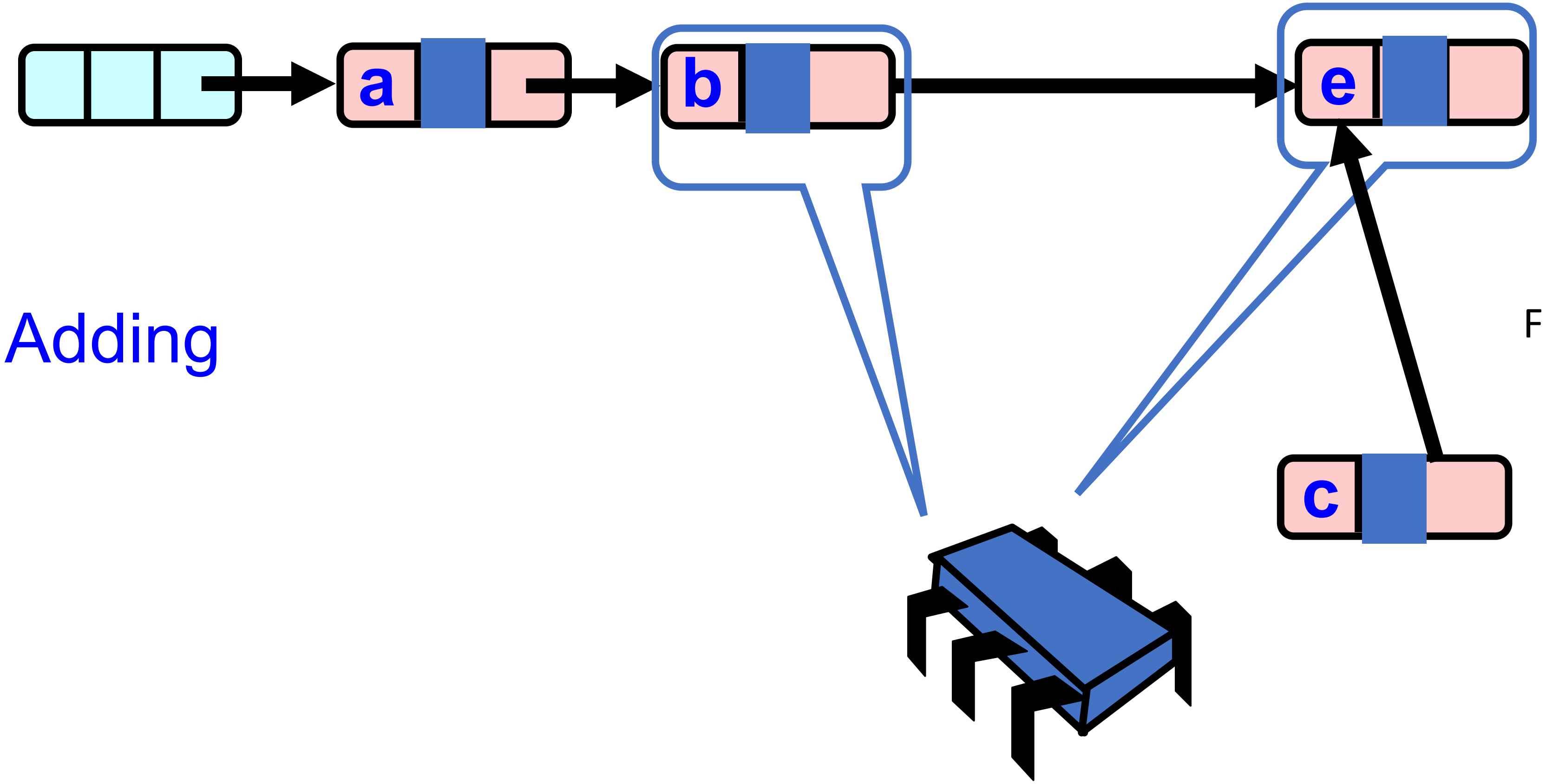
Lock-free Lists



Adding

Find the location

Lock-free Lists

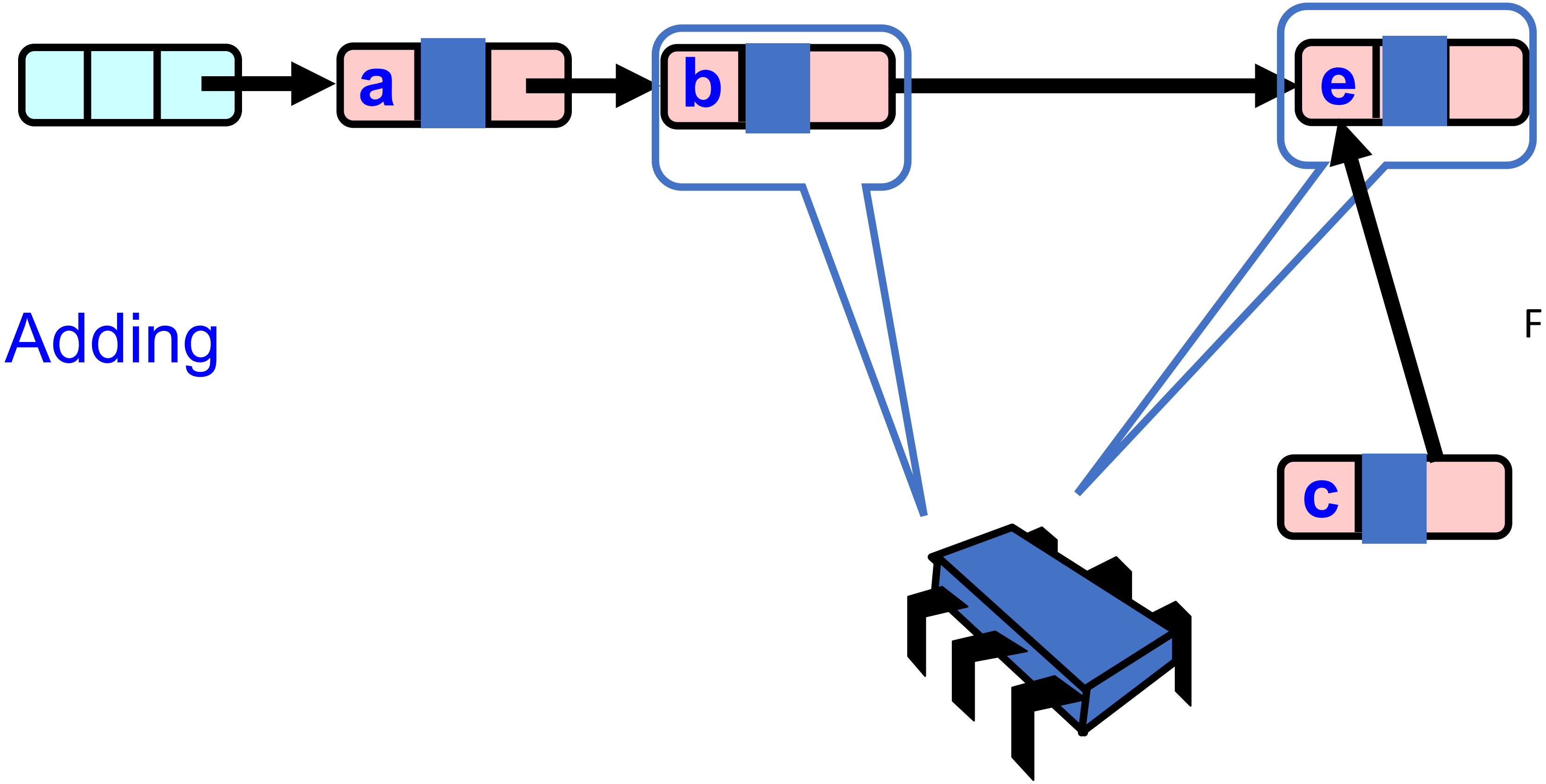


Adding

Find the location

create “c”

Lock-free Lists



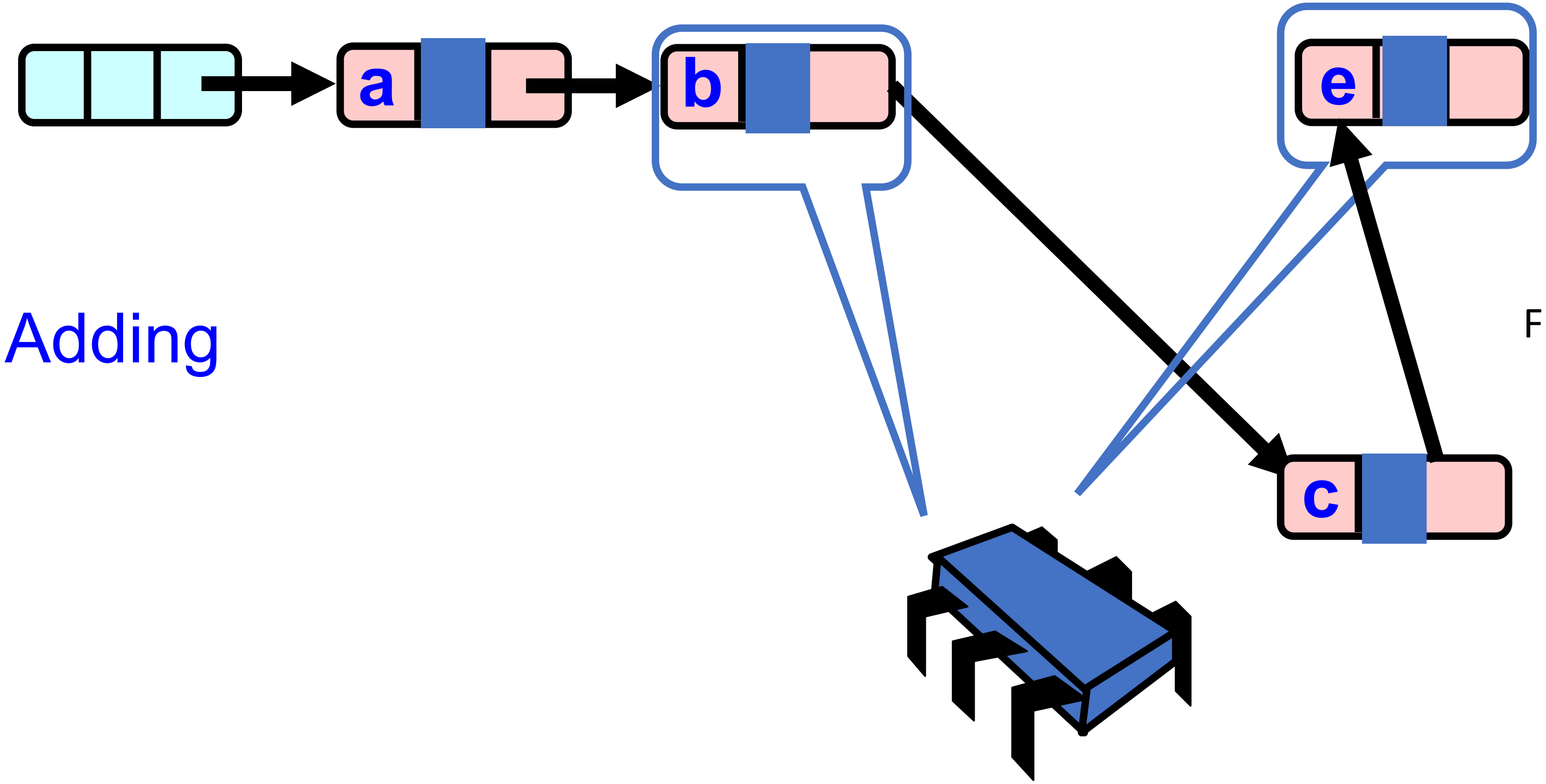
Adding

Find the location

create “c”

insert “c”

Lock-free Lists



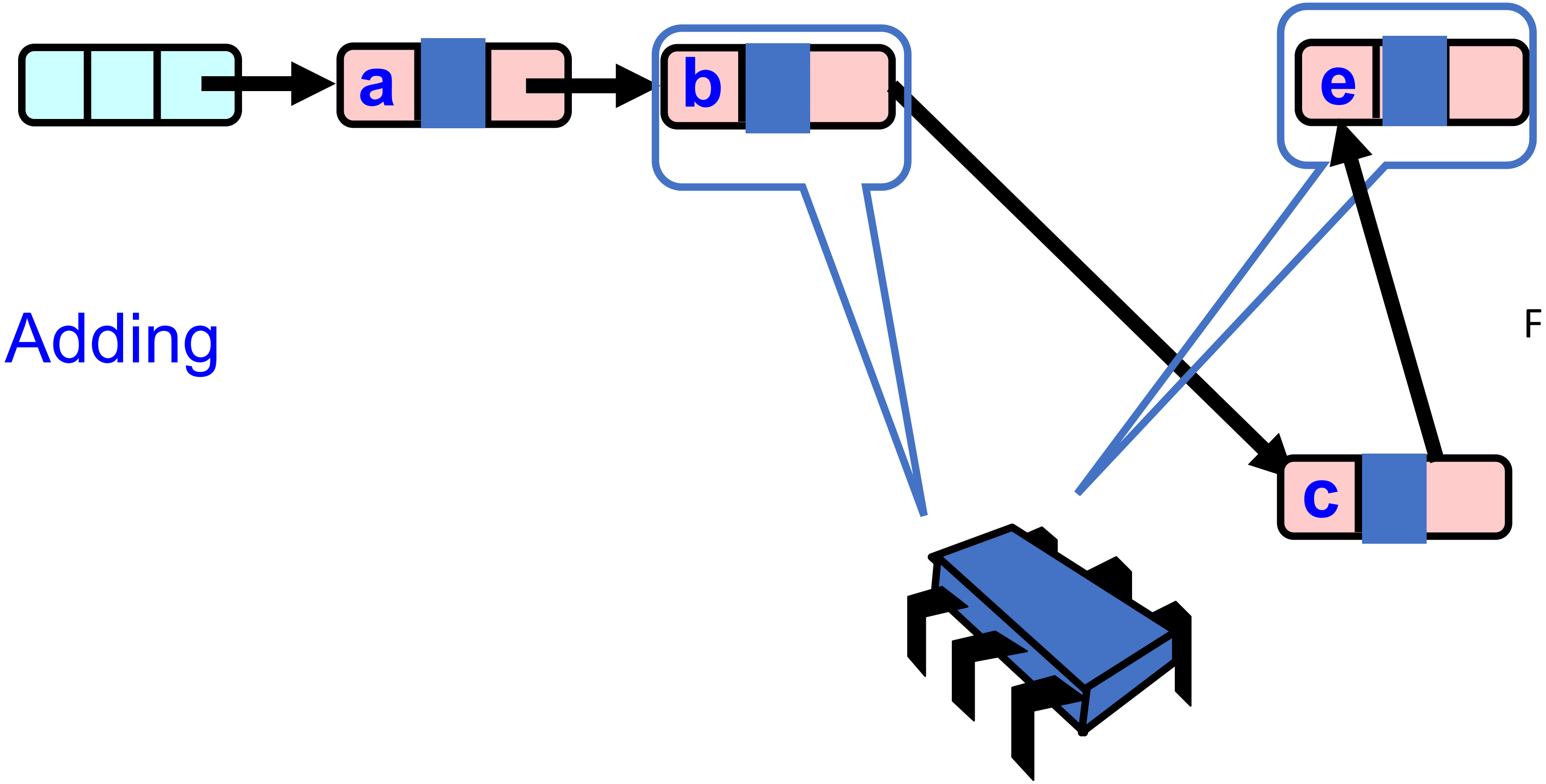
Adding

Find the location

create “c”

insert “c”

Lock-free Lists



Adding

Find the location

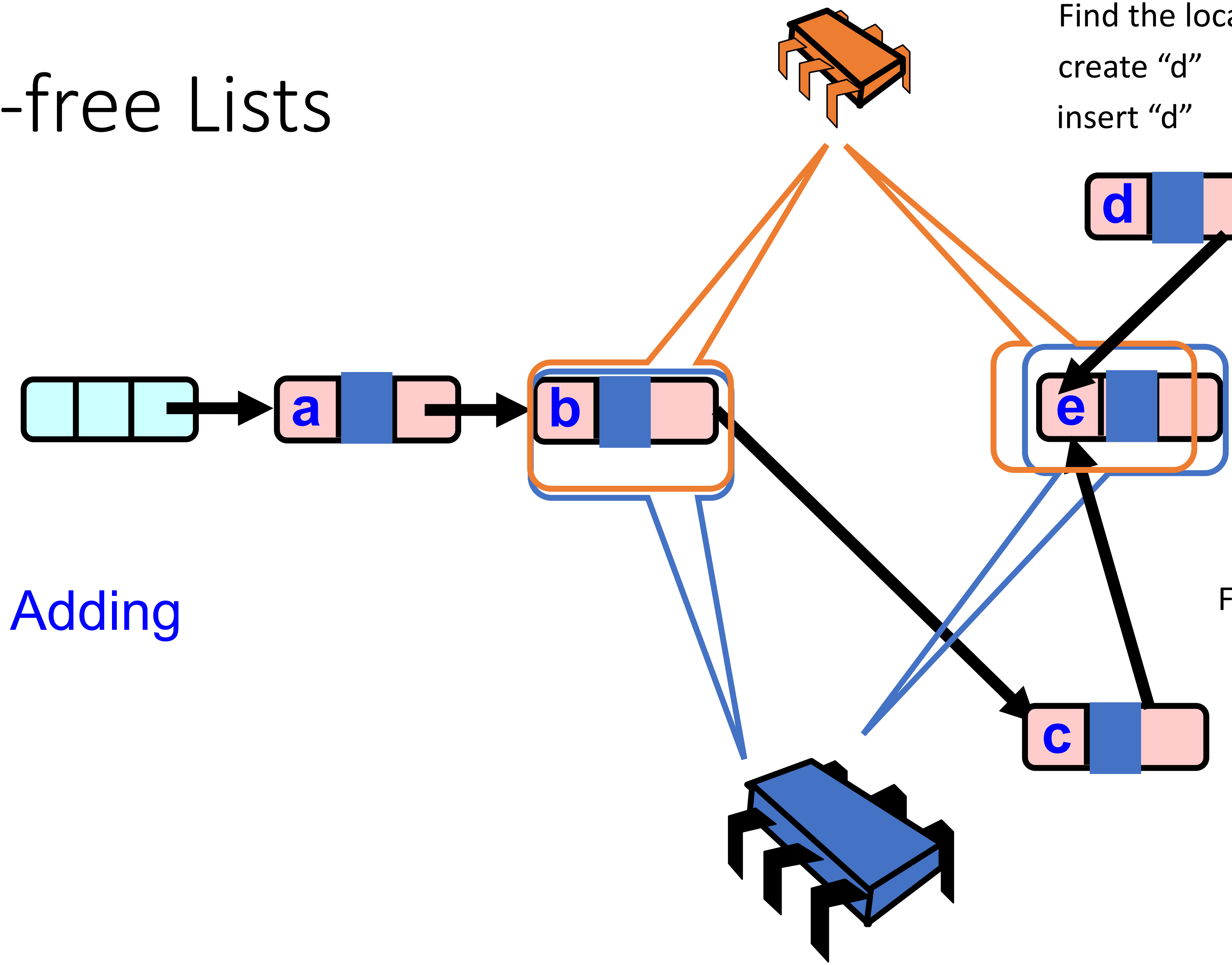
create “c”

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?

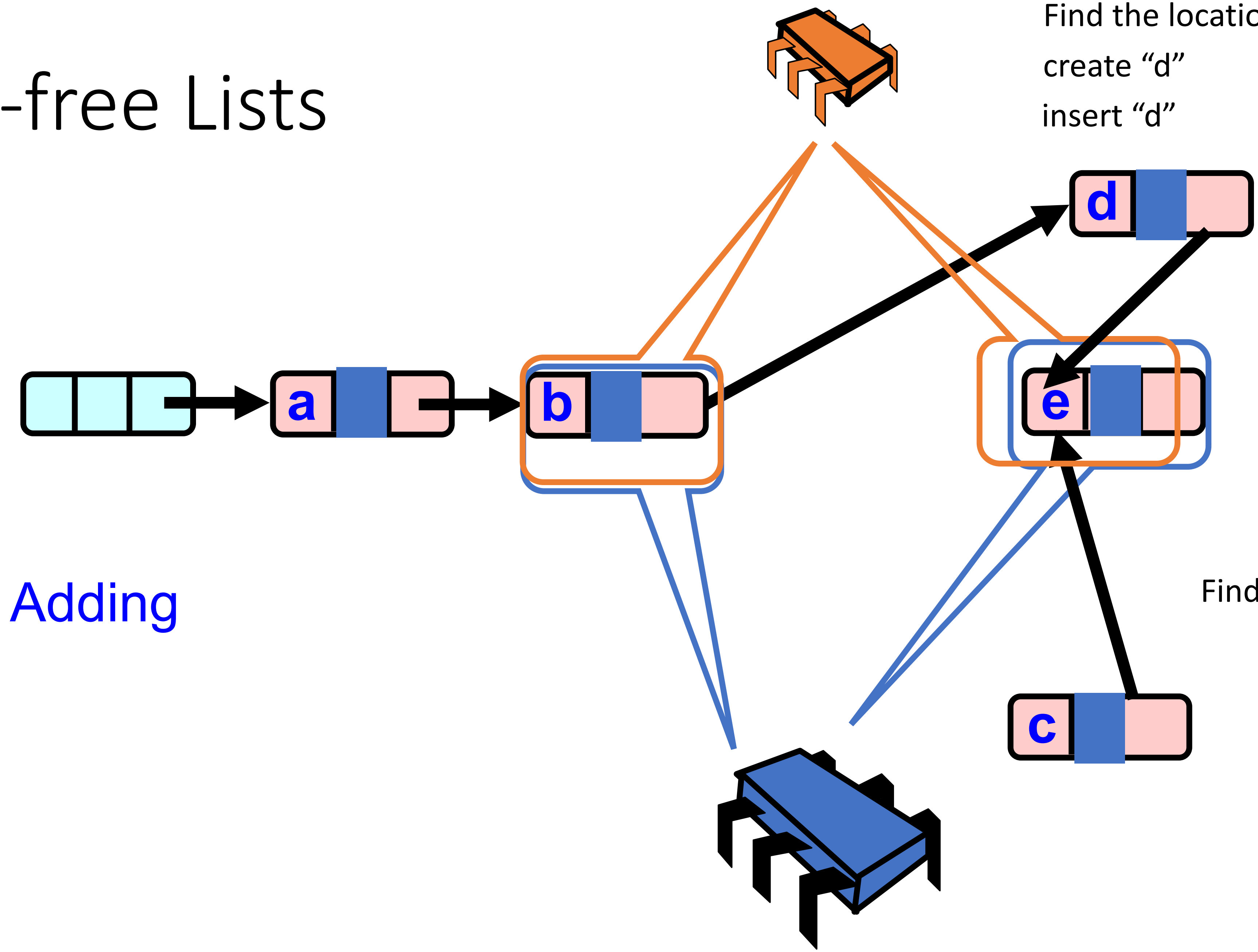
Find the location

create “c”

insert “c”

Can this just be a regular store?

Lock-free Lists



Find the location

create “d”

insert “d”

Can this just
be a regular store?

Adding

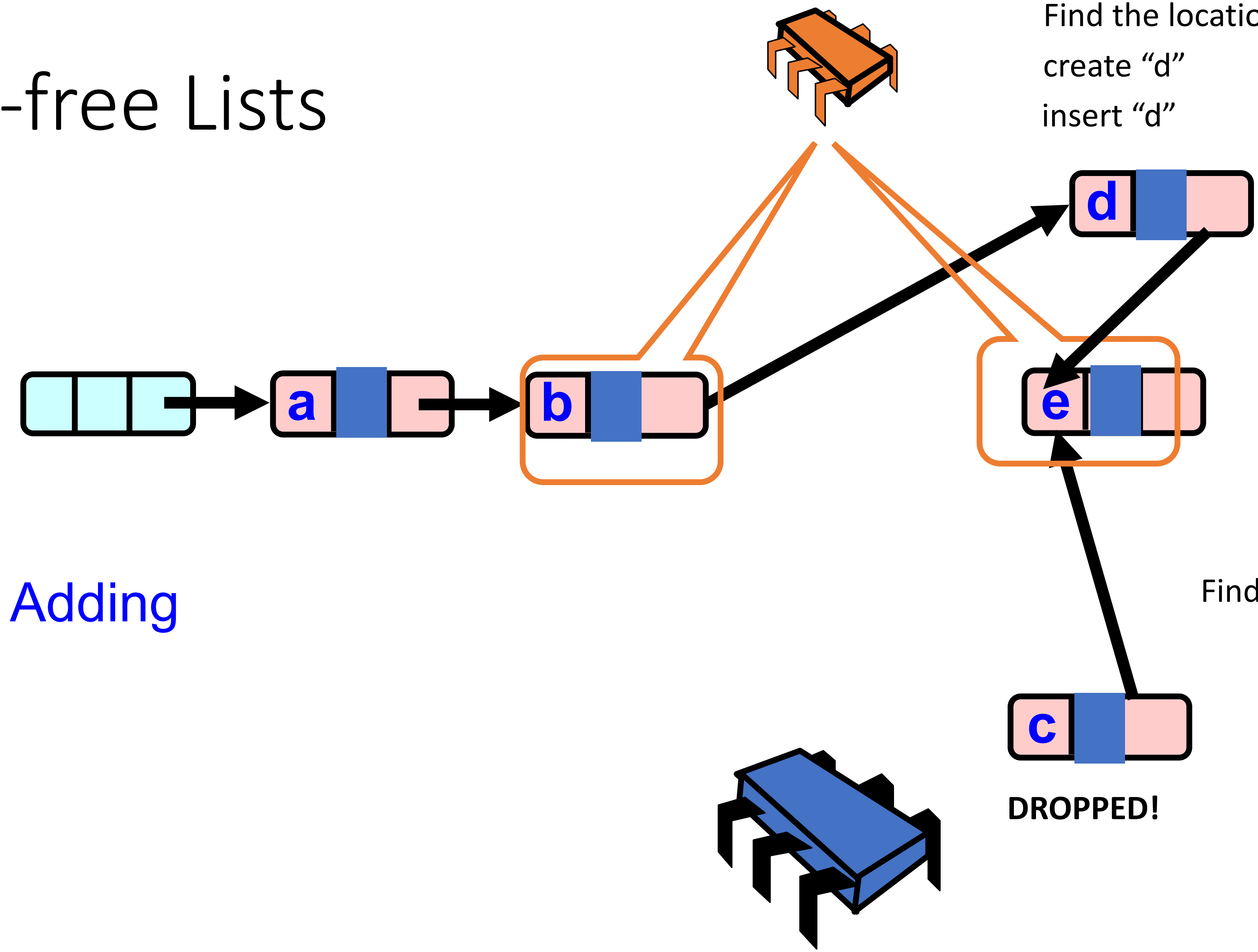
Find the location

create “c”

insert “c”

Can this just
be a regular store?

Lock-free Lists



Find the location

create “d”

insert “d”

Can this just be a regular store?

Adding

Find the location

create “c”

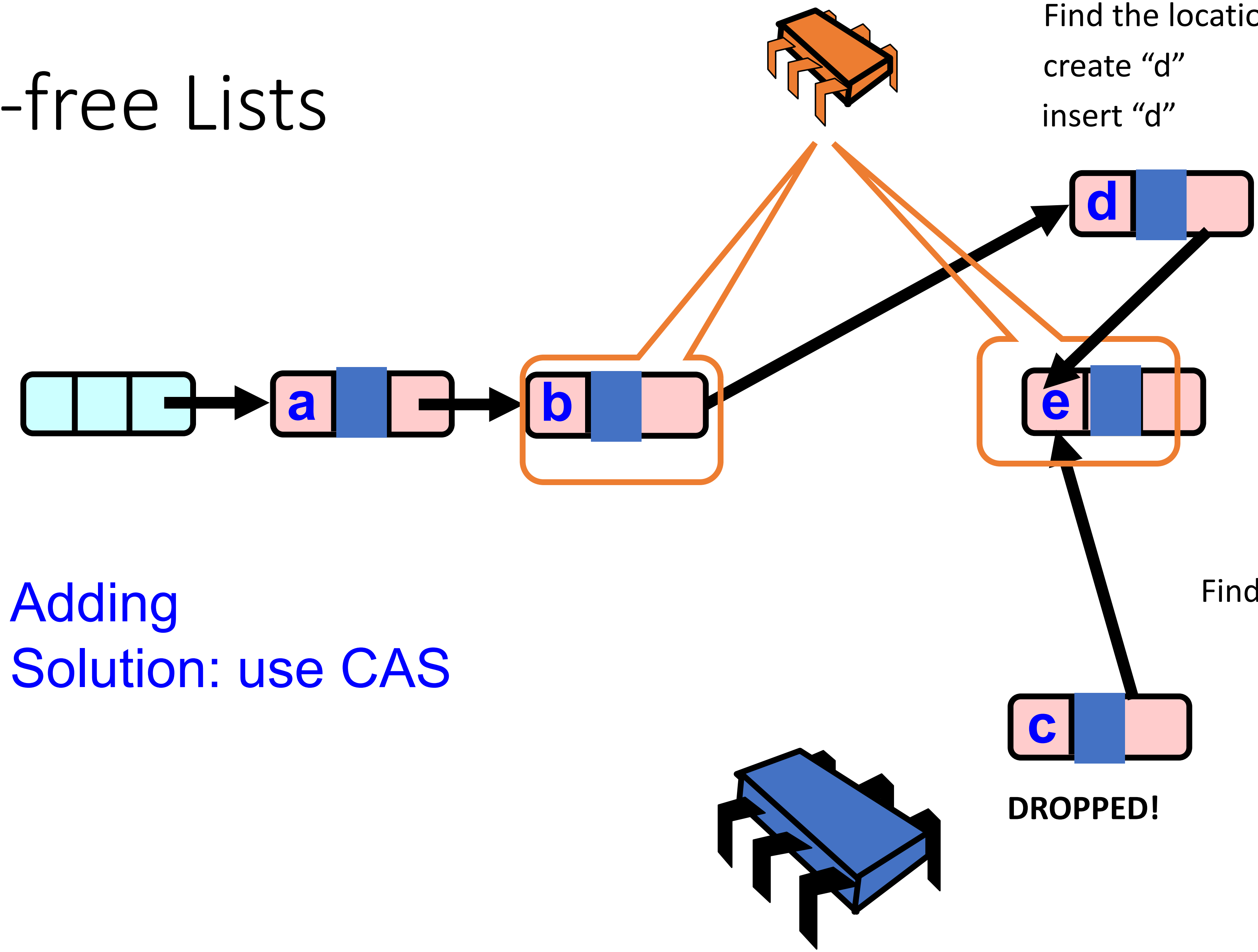
insert “c”

Can this just be a regular store?

Lock-free Lists

Adding

Solution: use CAS



Find the location

create “d”

insert “d”

Can this just be a regular store?

Find the location

create “c”

insert “c”

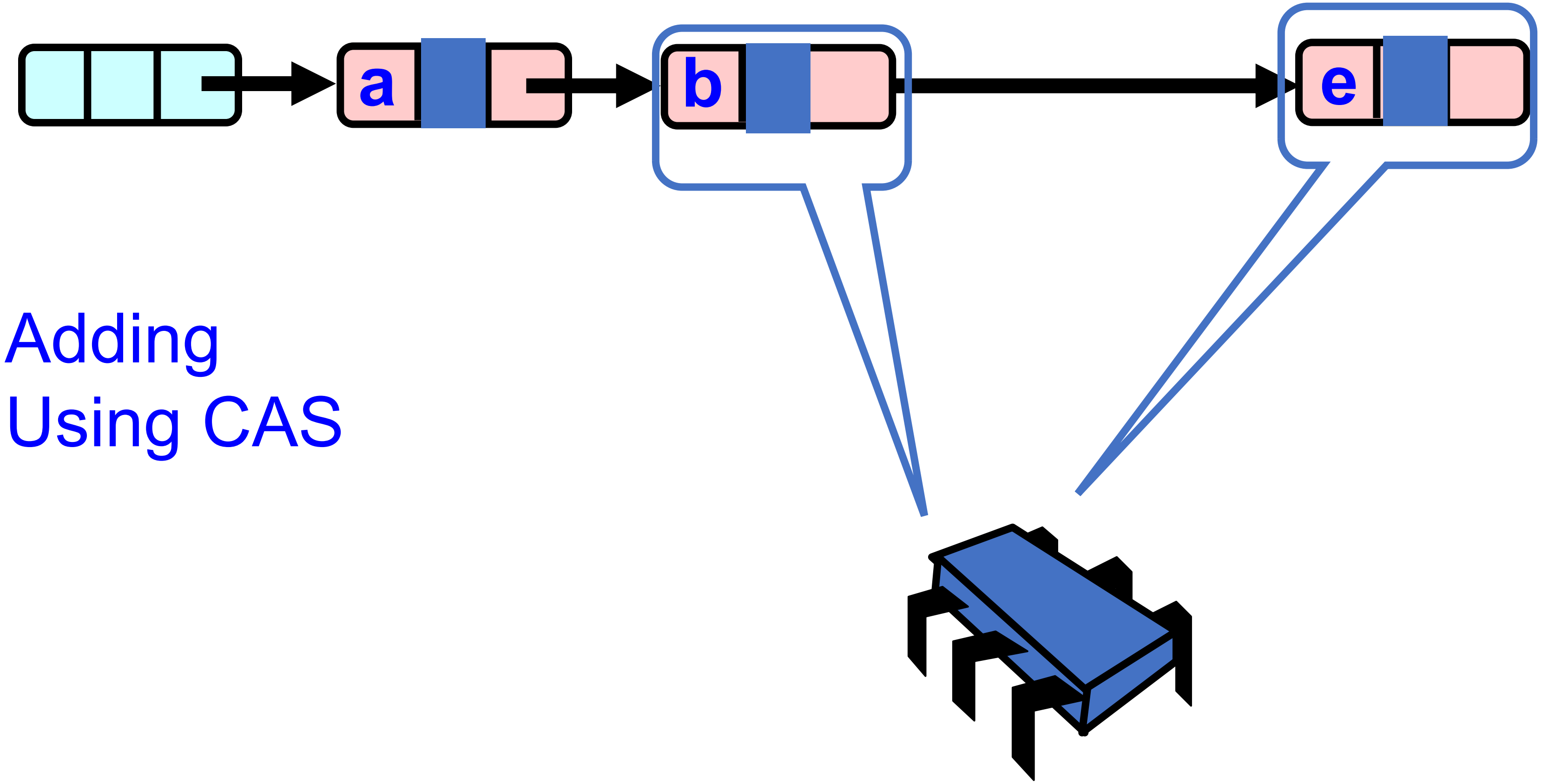
Can this just be a regular store?

Lock-free Lists

Find the location

Cache your insertion point!

b.next == e



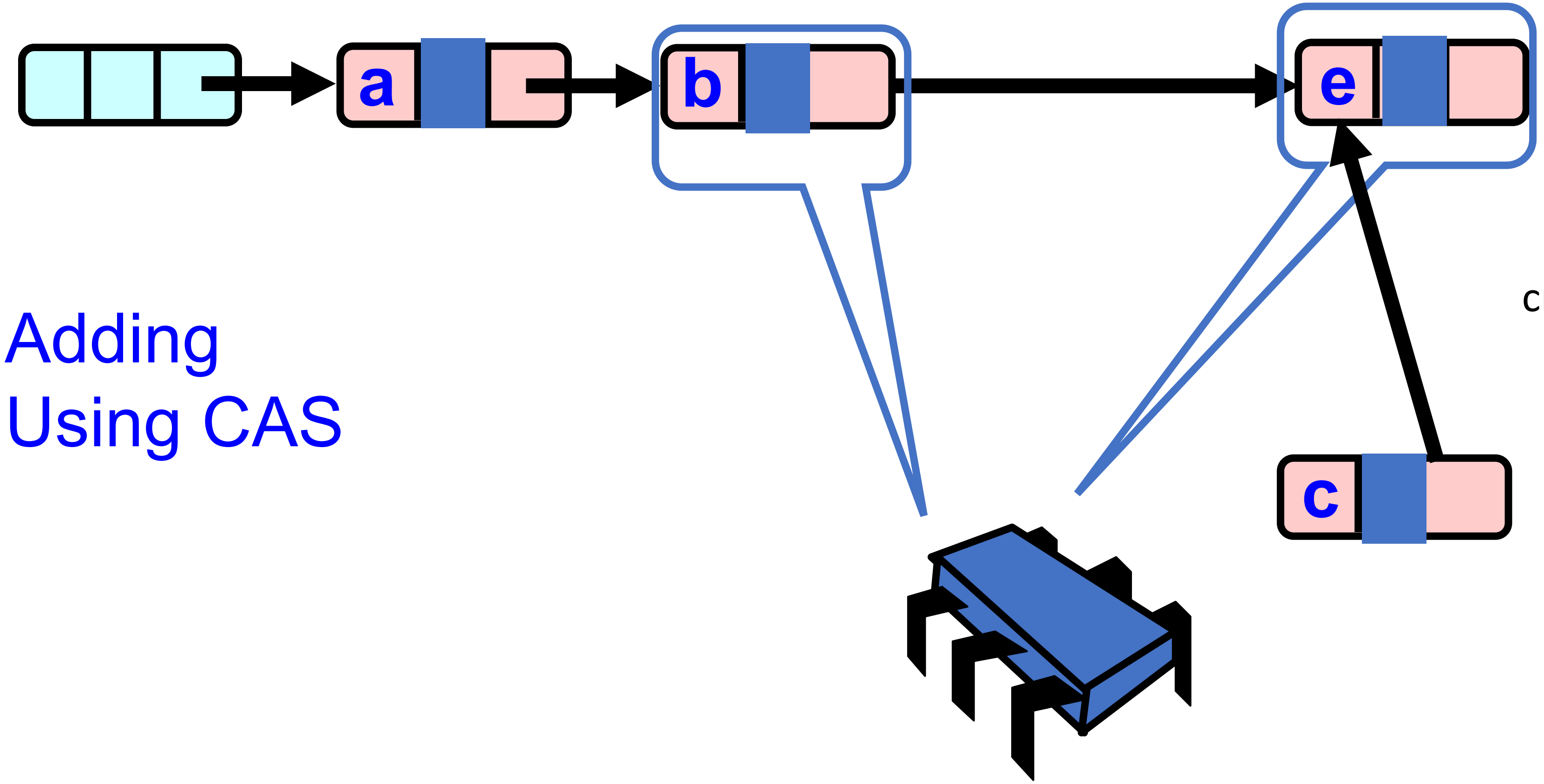
Adding
Using CAS

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!

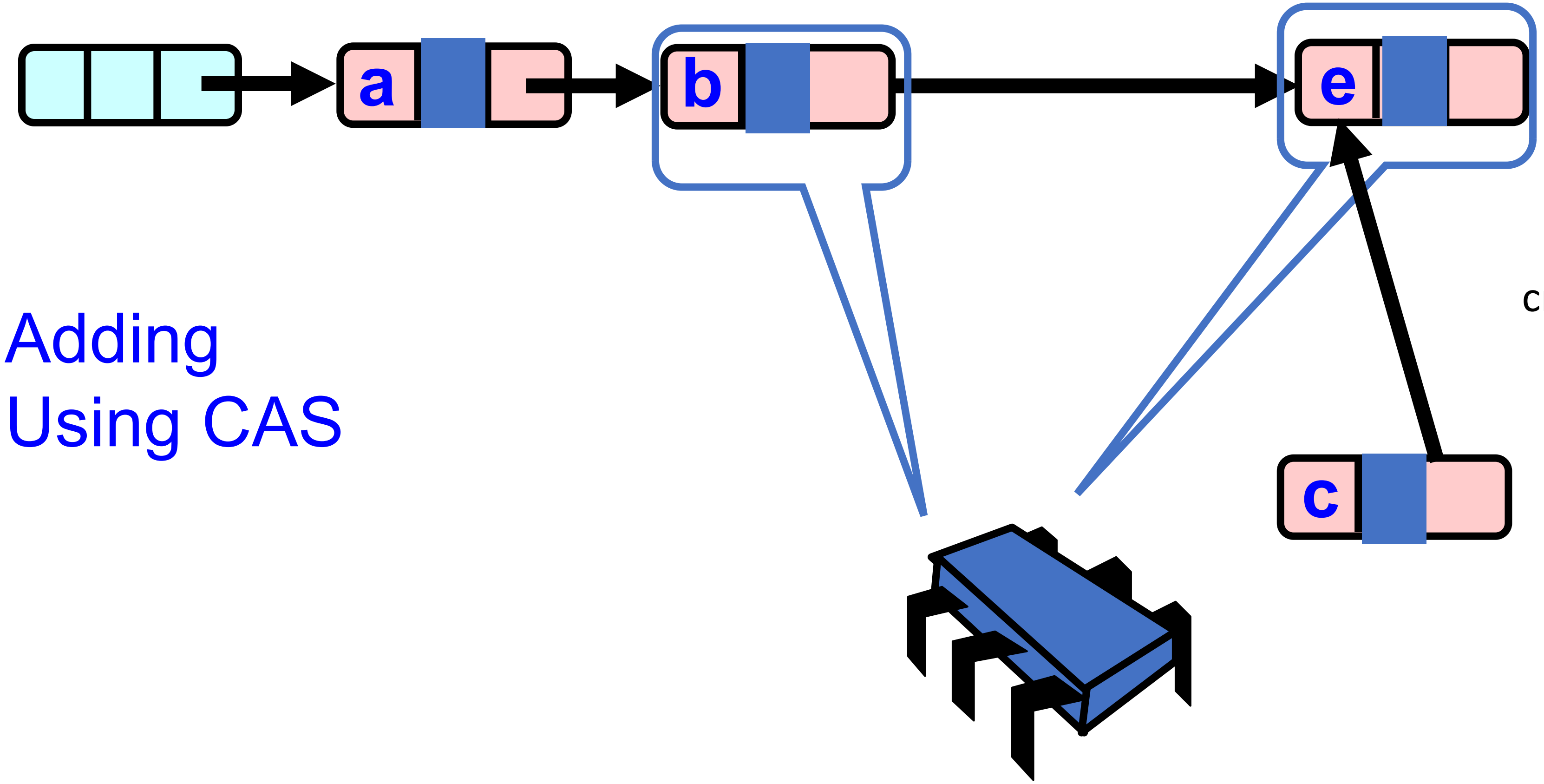
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



create “c”

Adding
Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

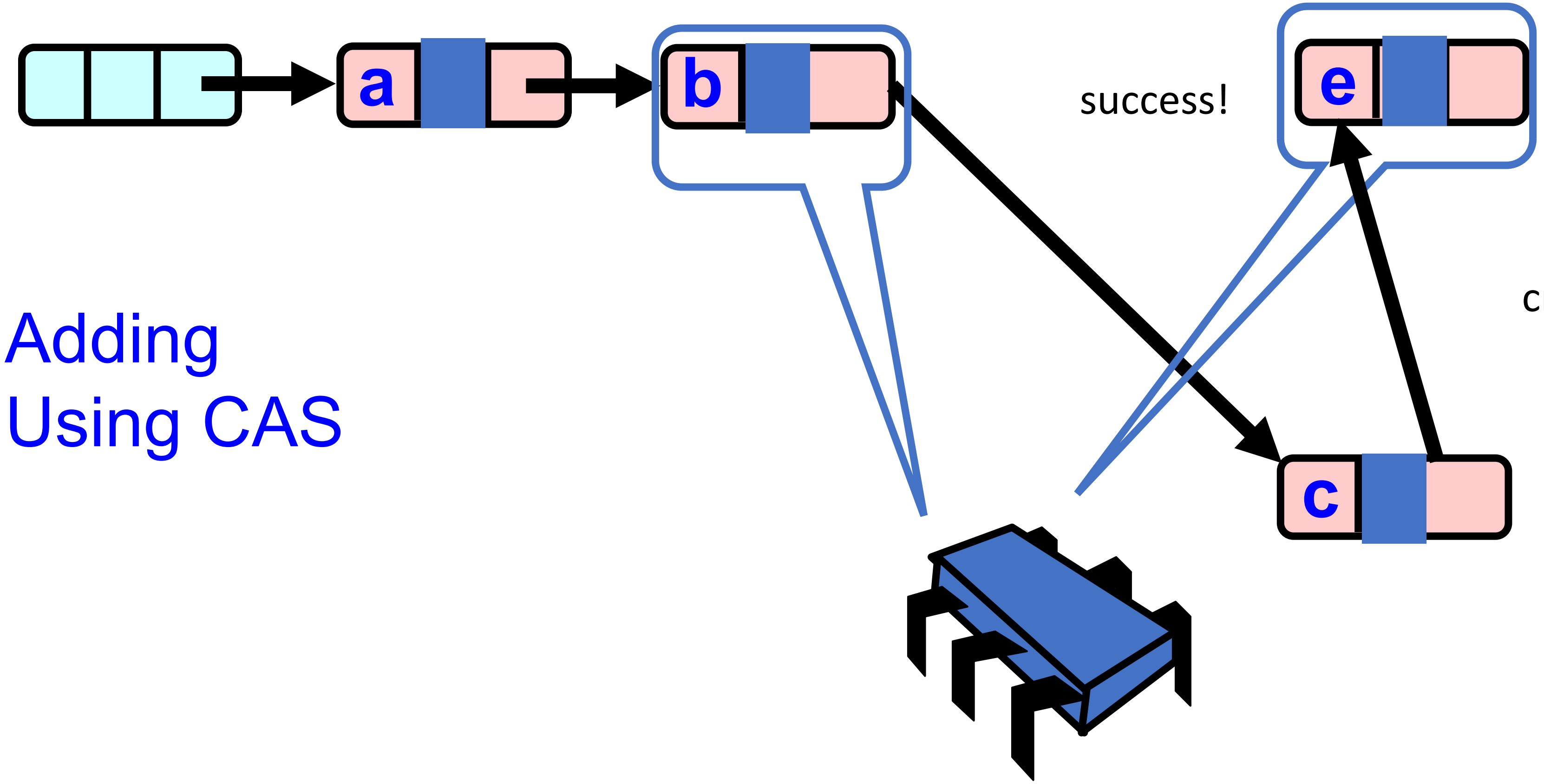
CAS(b.next, e, c);

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!

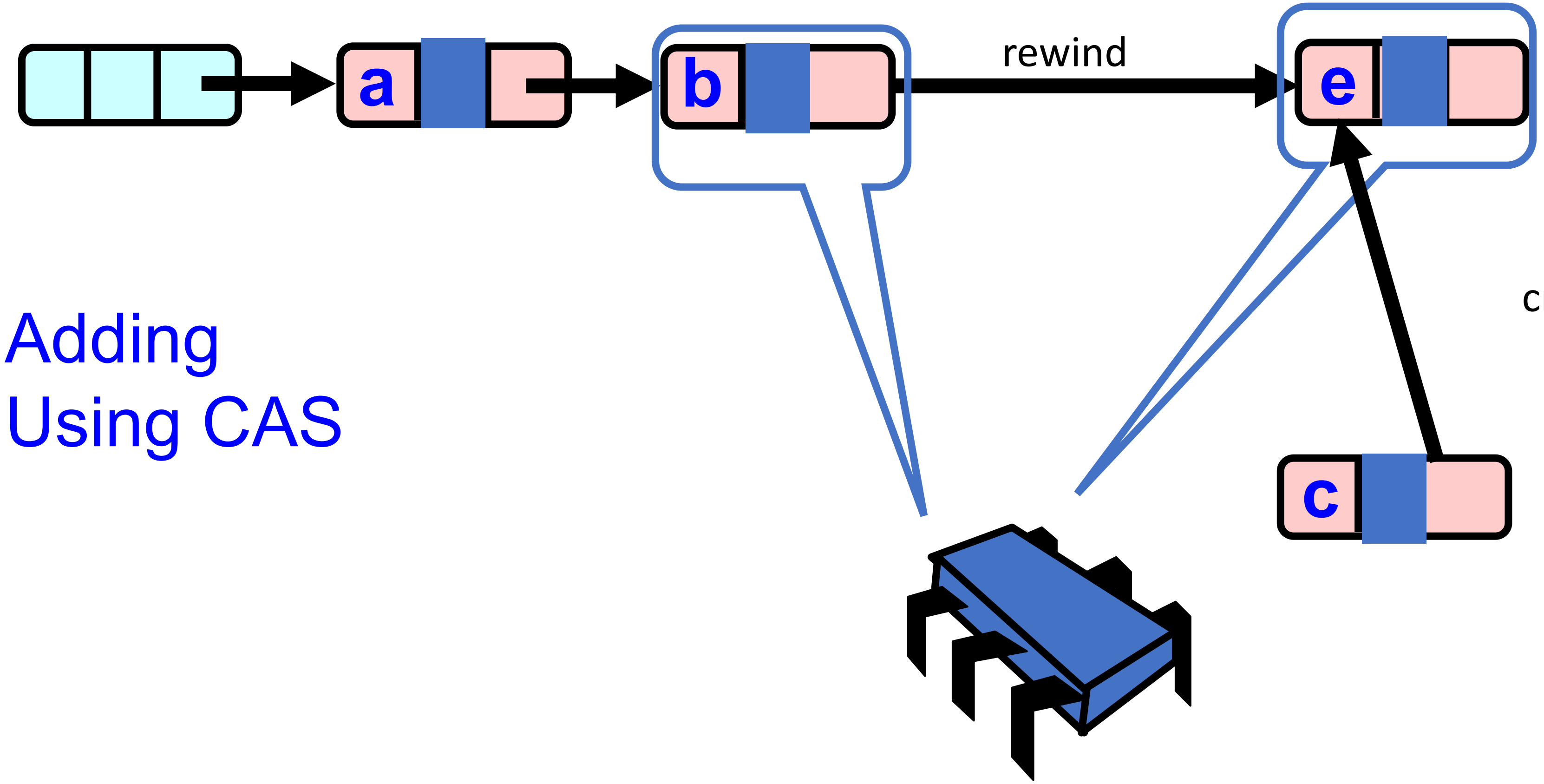
CAS(b.next, e, c);

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!

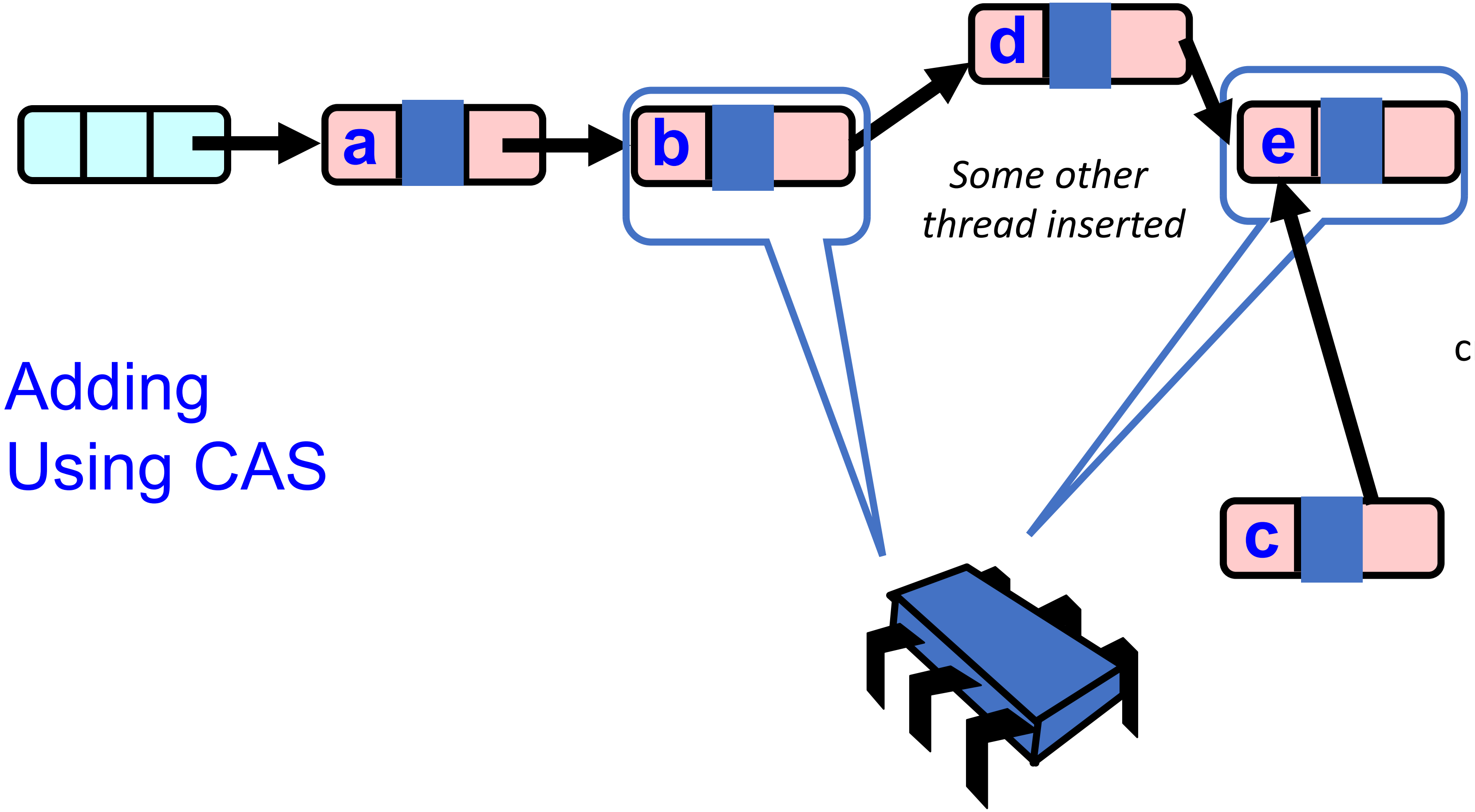
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Some other thread inserted

create “c”

Adding
Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

CAS(b.next, e, c);

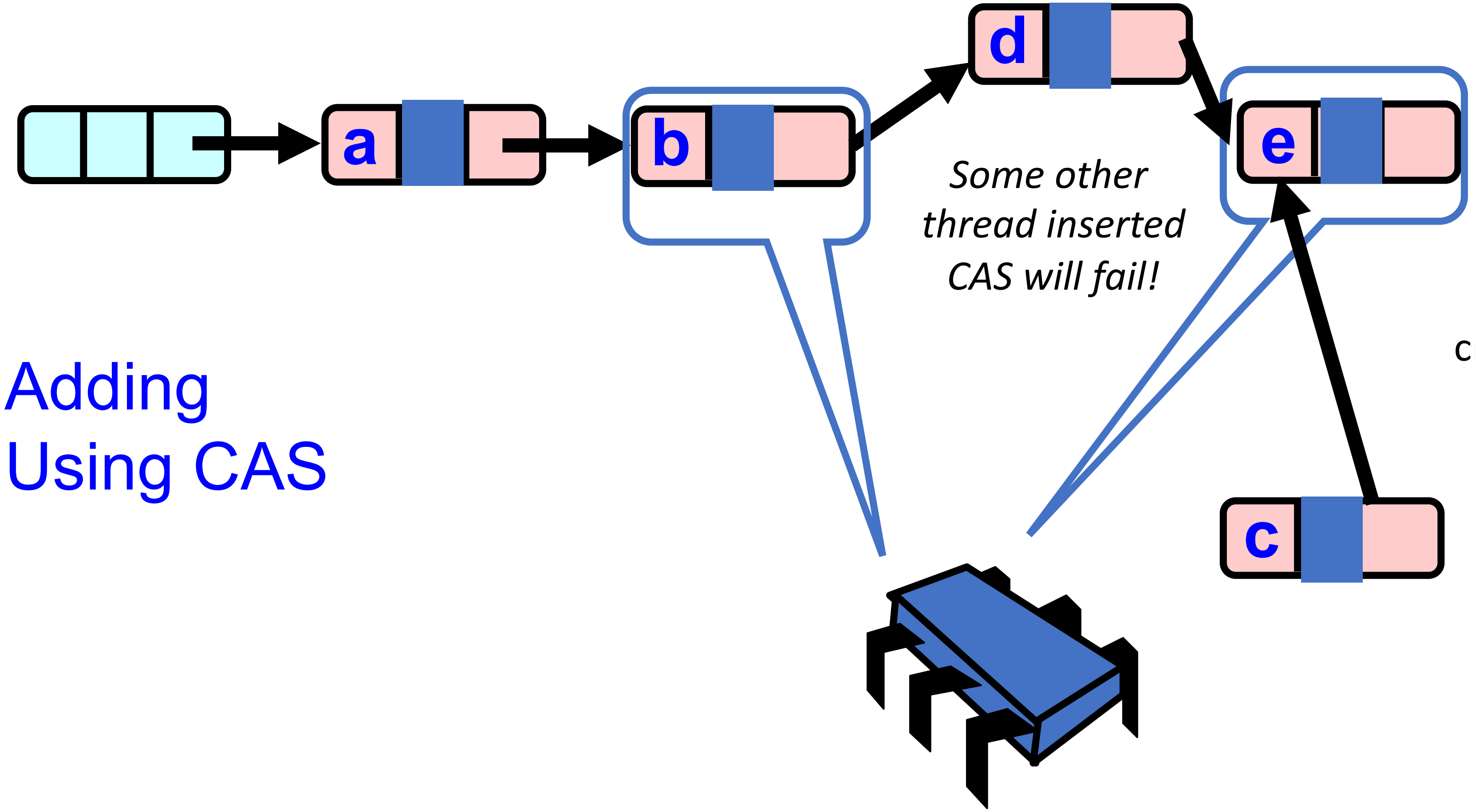
Find the location

Cache your insertion point!

b.next == e

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

Adding Using CAS



Some other thread inserted CAS will fail!

create “c”

Lock-free Lists

Only insert if your insertion point is valid!

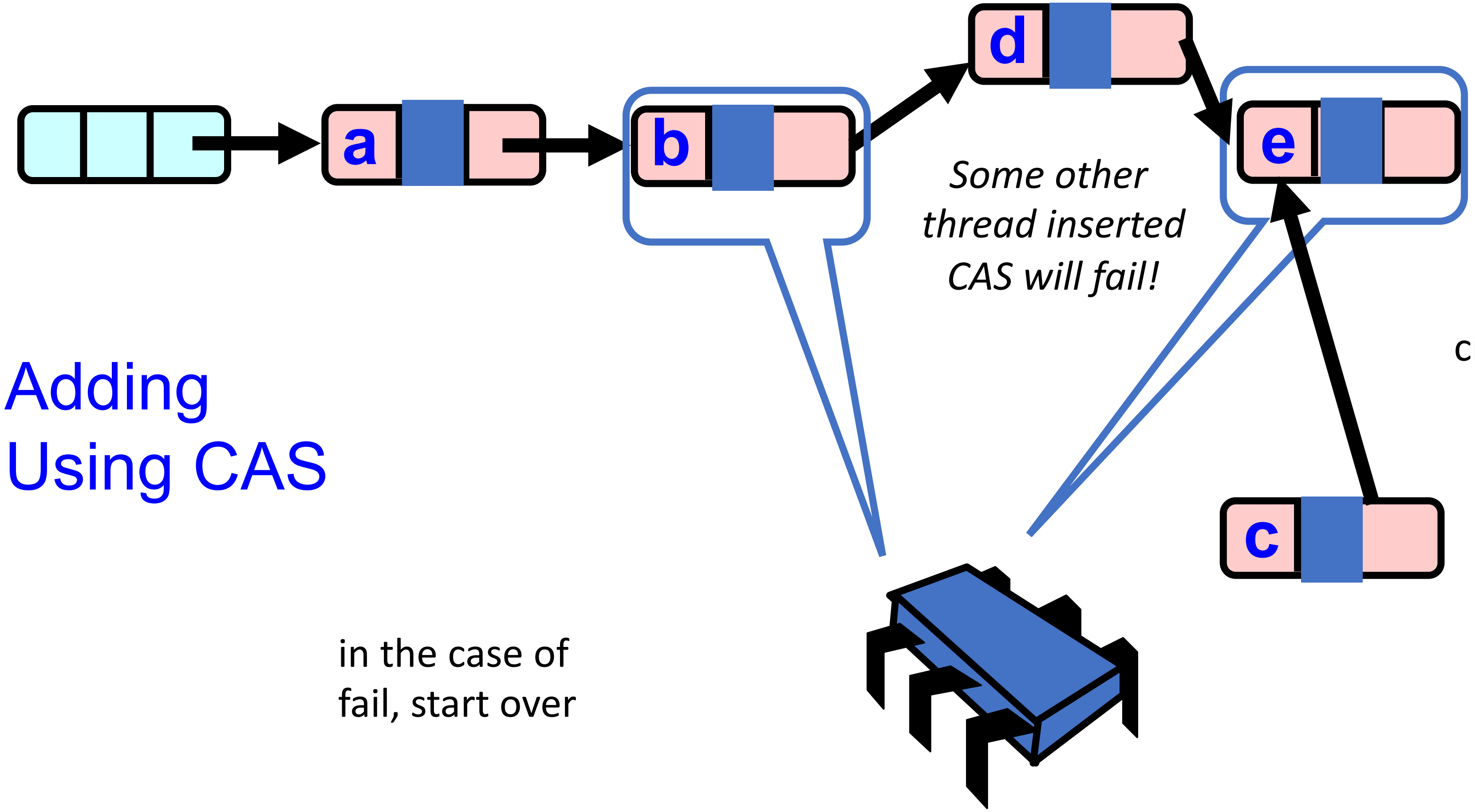
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



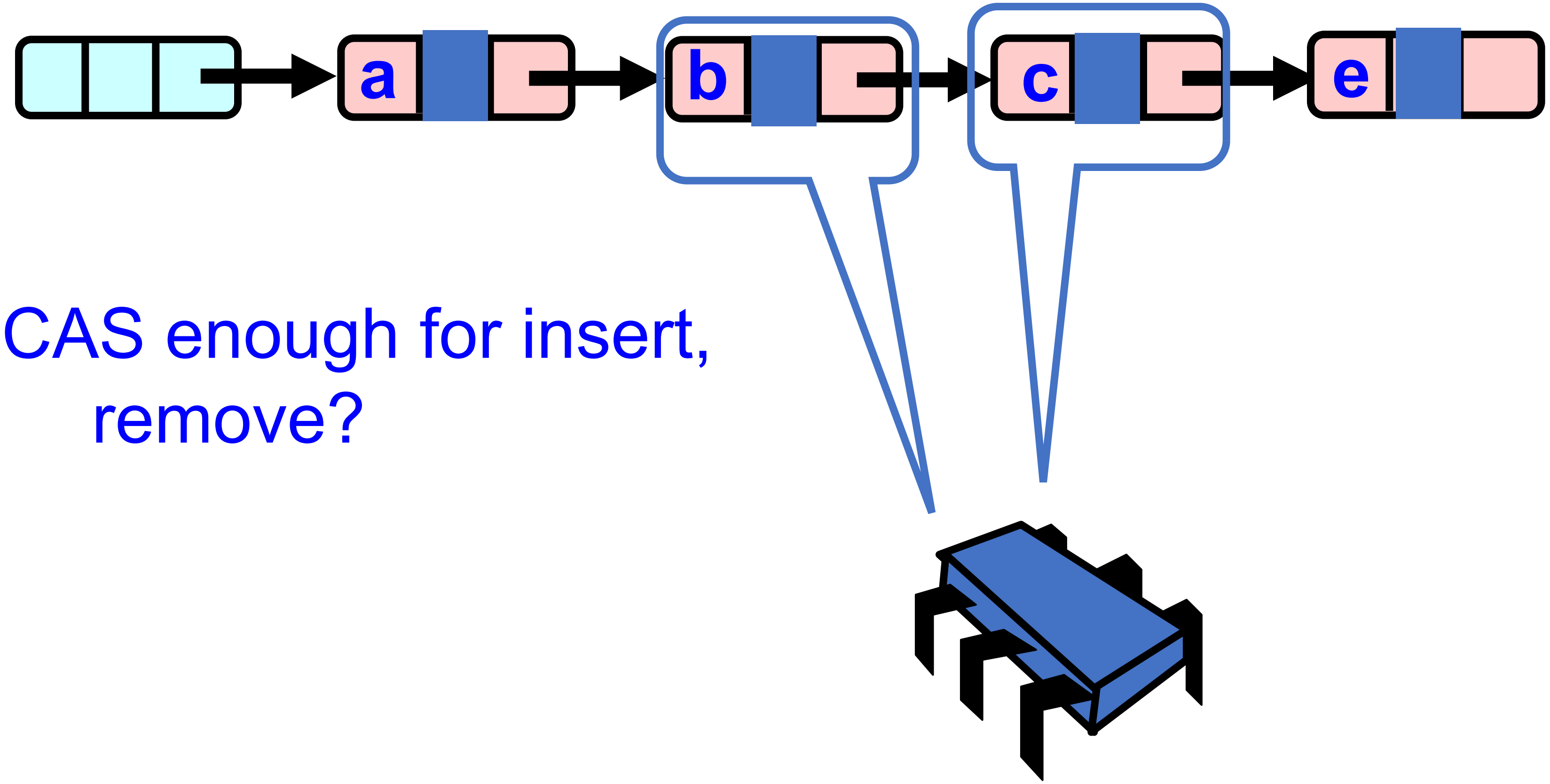
Some other thread inserted CAS will fail!

create “c”

Adding
Using CAS

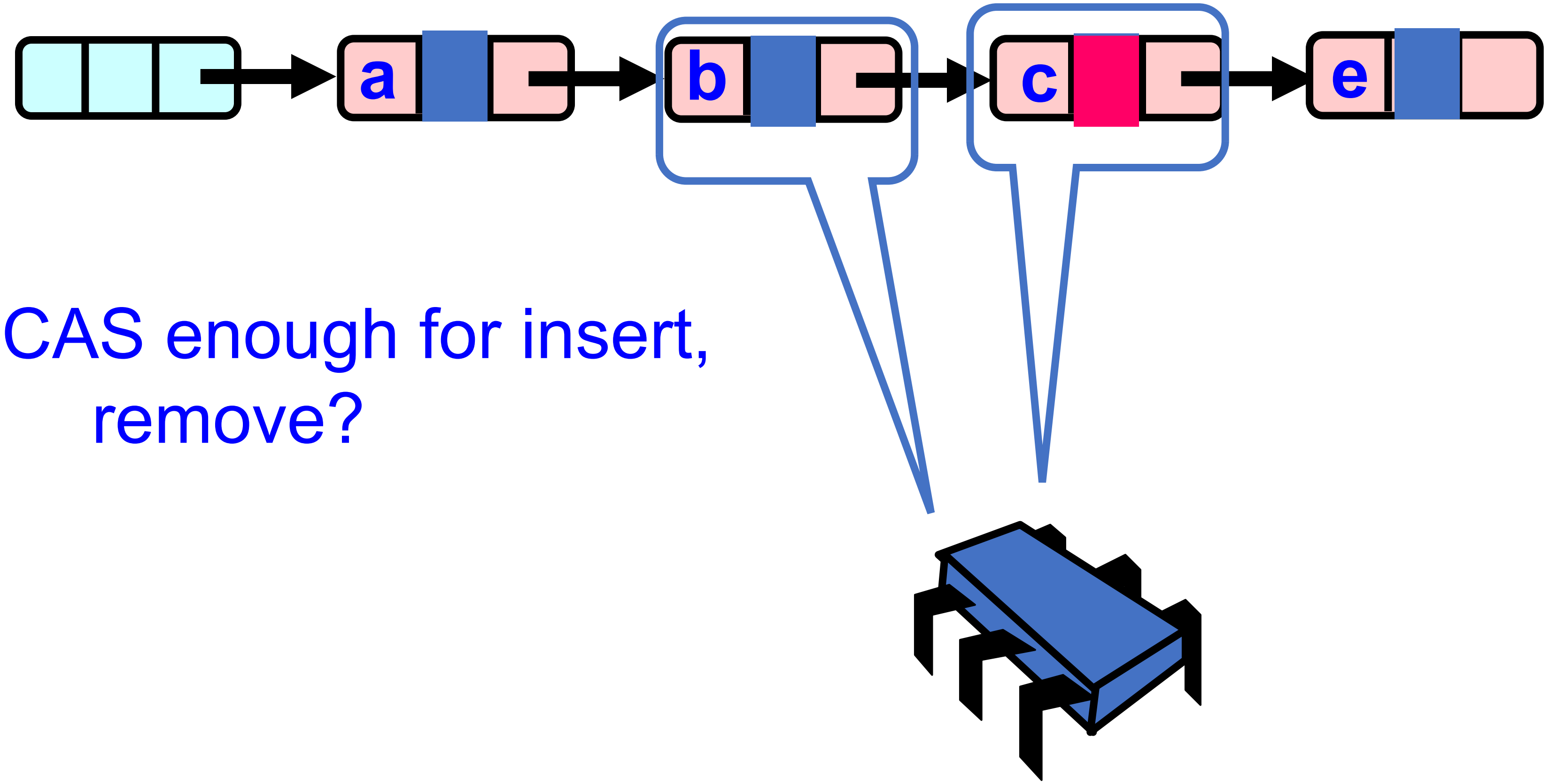
in the case of fail, start over

Lock-free Lists



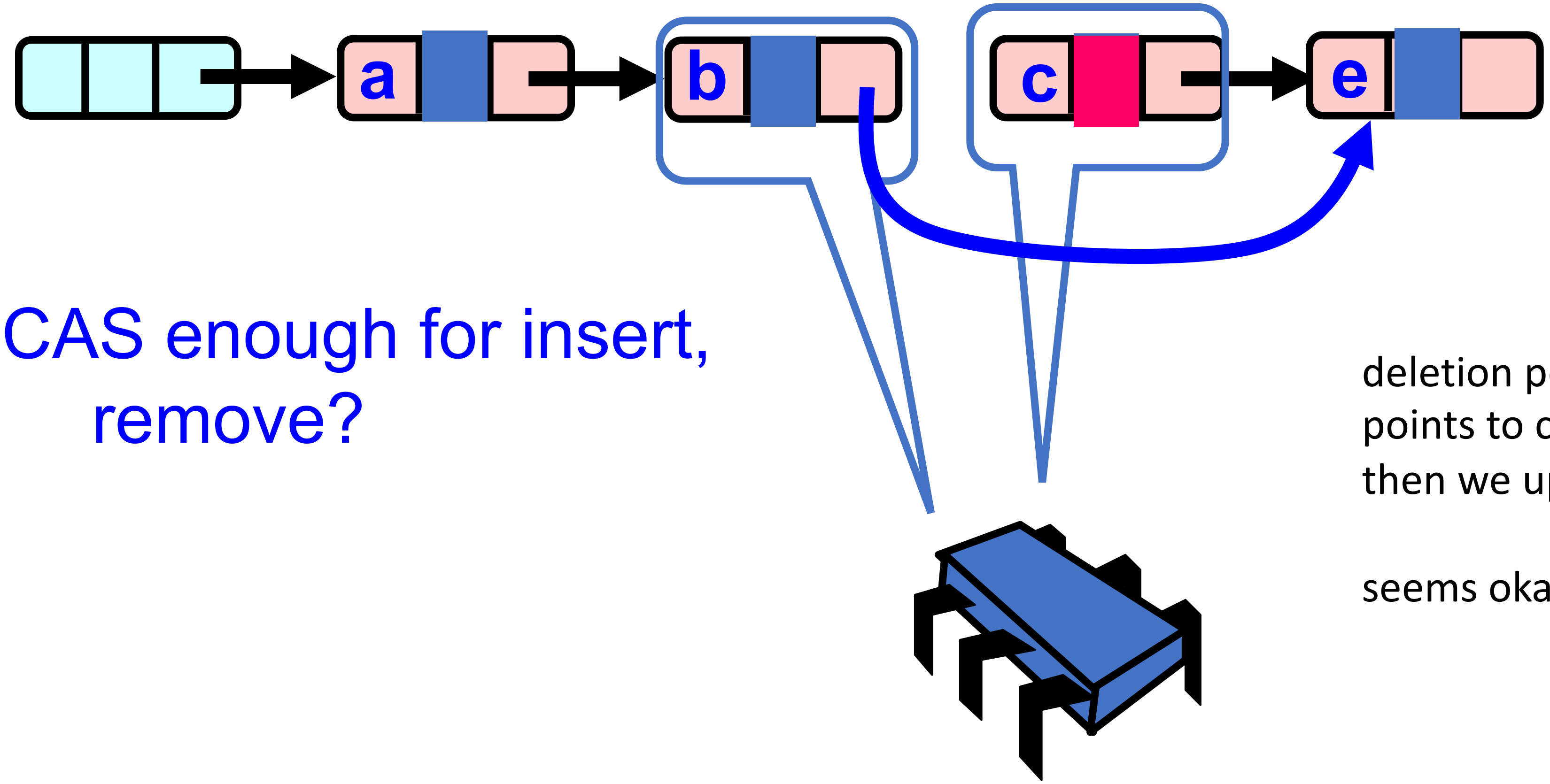
CAS enough for insert,
remove?

Lock-free Lists



CAS enough for insert, remove?

Lock-free Lists



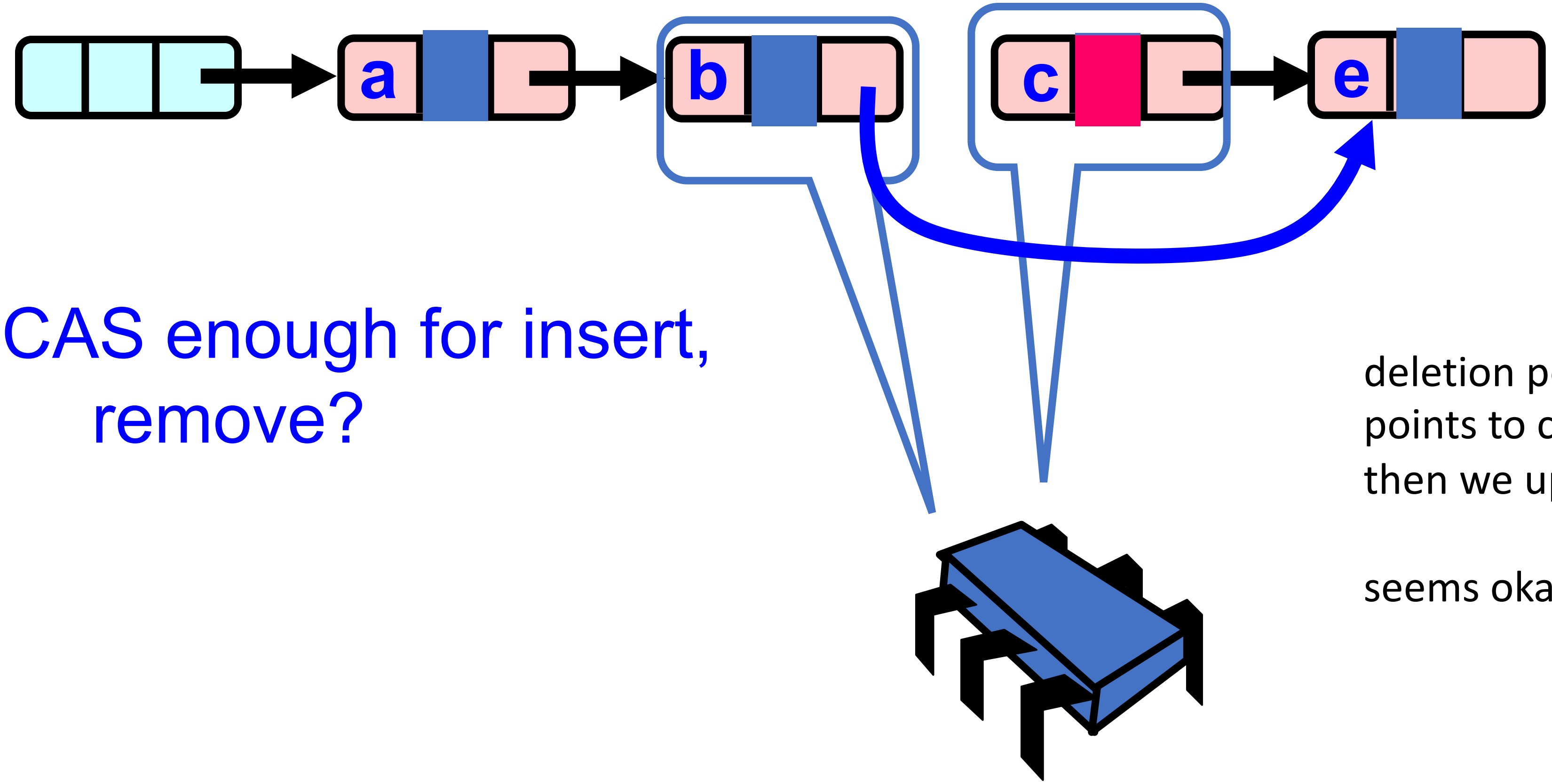
CAS enough for insert, remove?

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



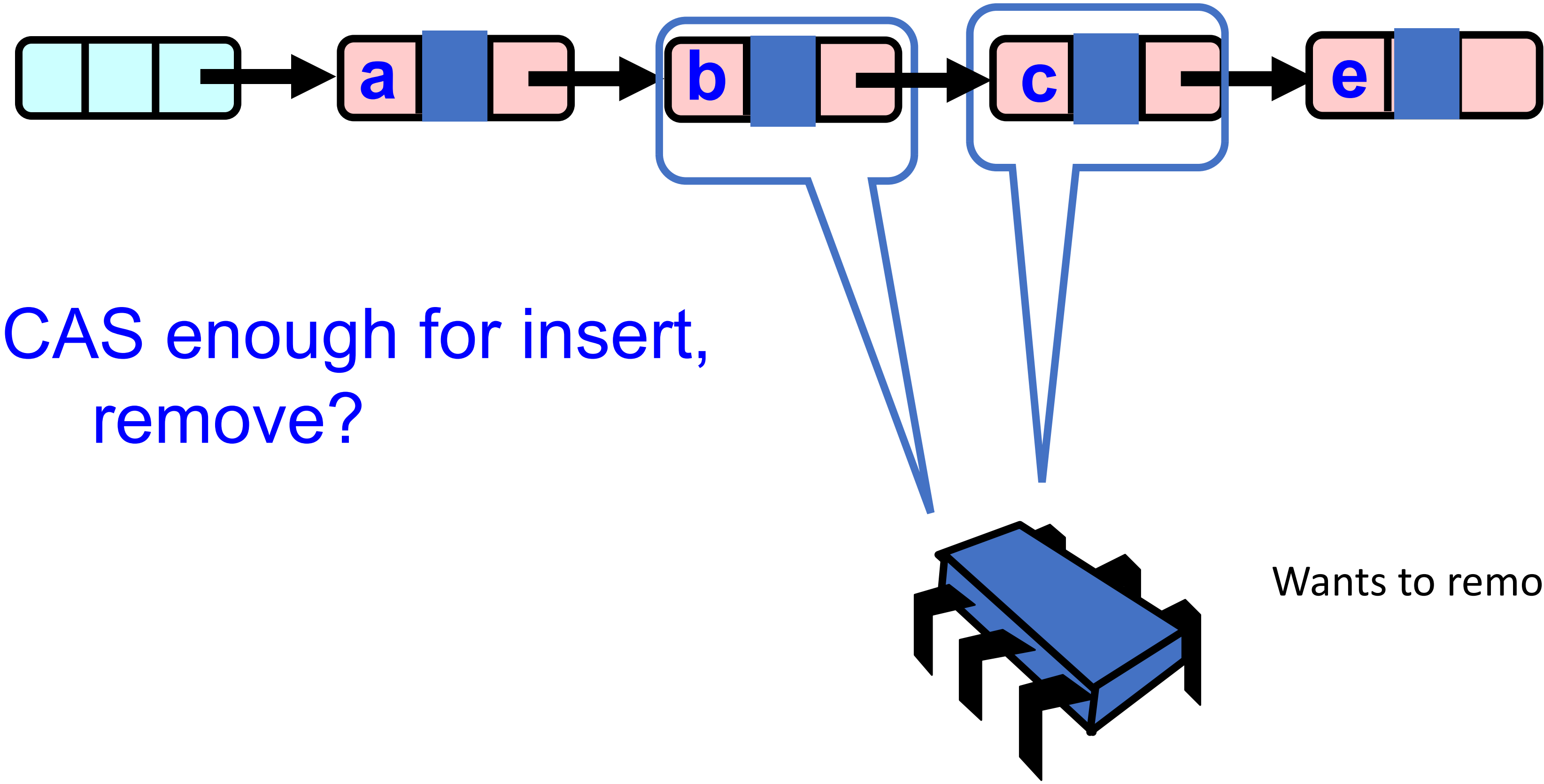
CAS enough for insert, remove?

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

seems okay...

Lock-free Lists

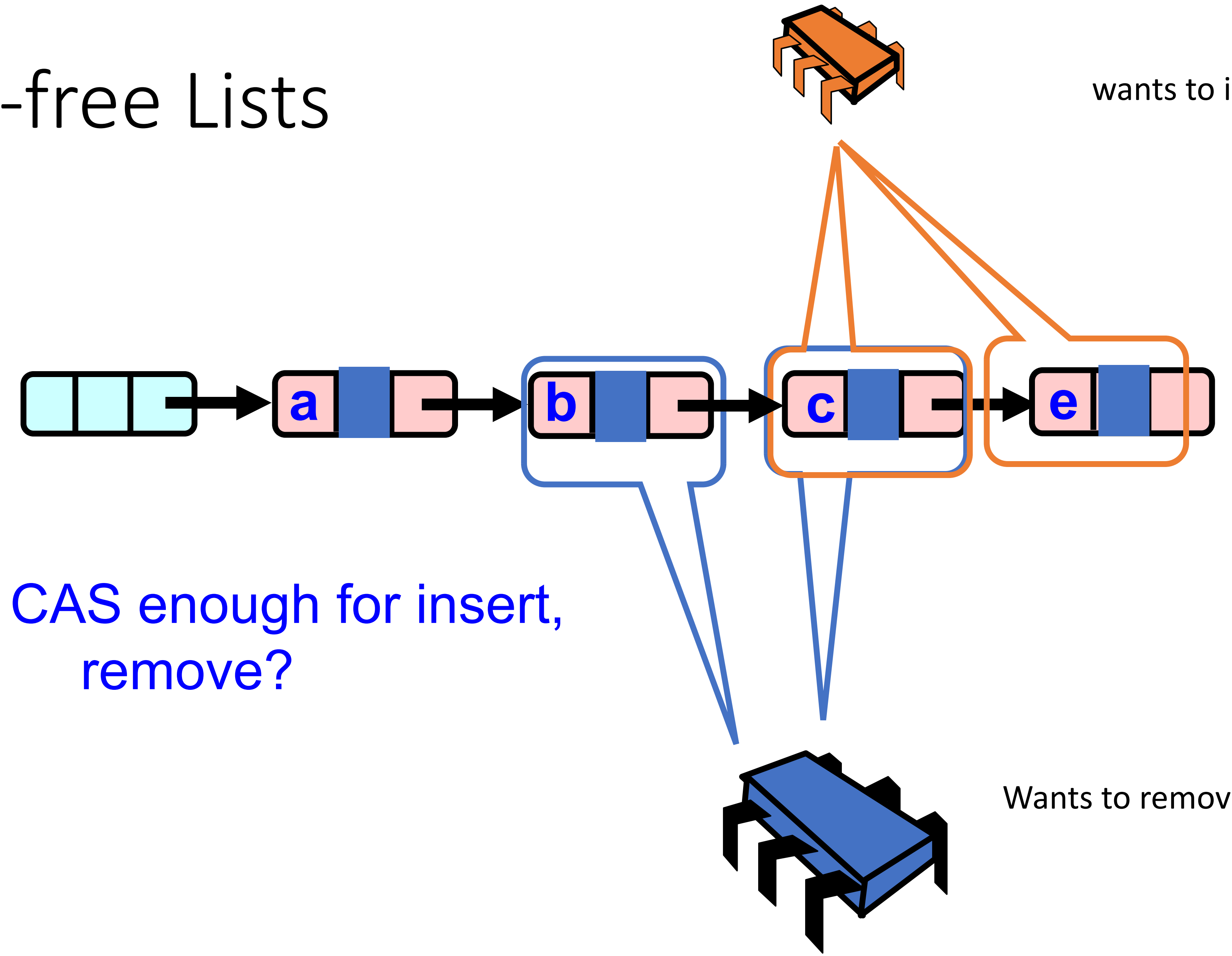
Rewind



CAS enough for insert, remove?

Wants to remove c

Lock-free Lists

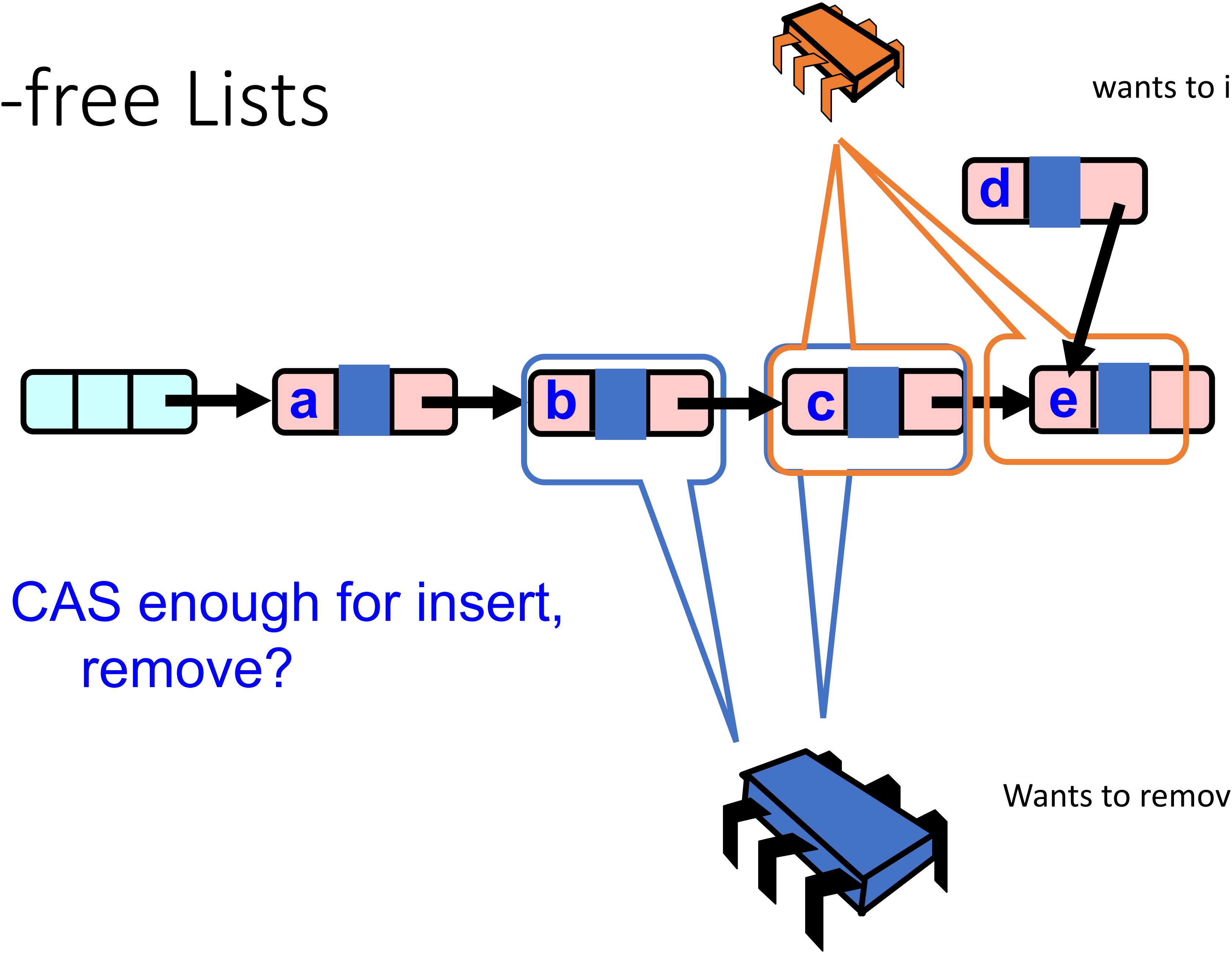


wants to insert d

Wants to remove c

CAS enough for insert, remove?

Lock-free Lists

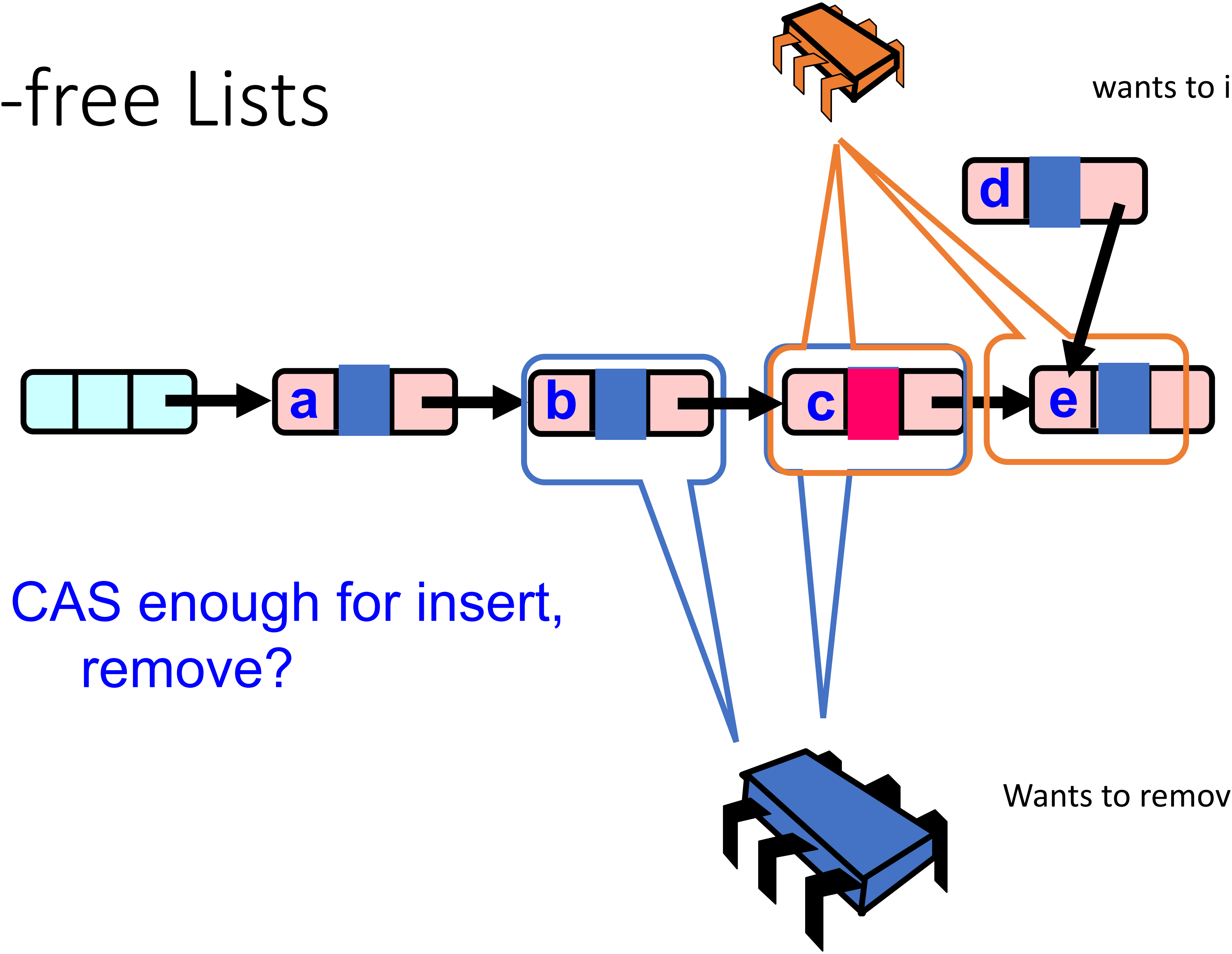


wants to insert d

Wants to remove c

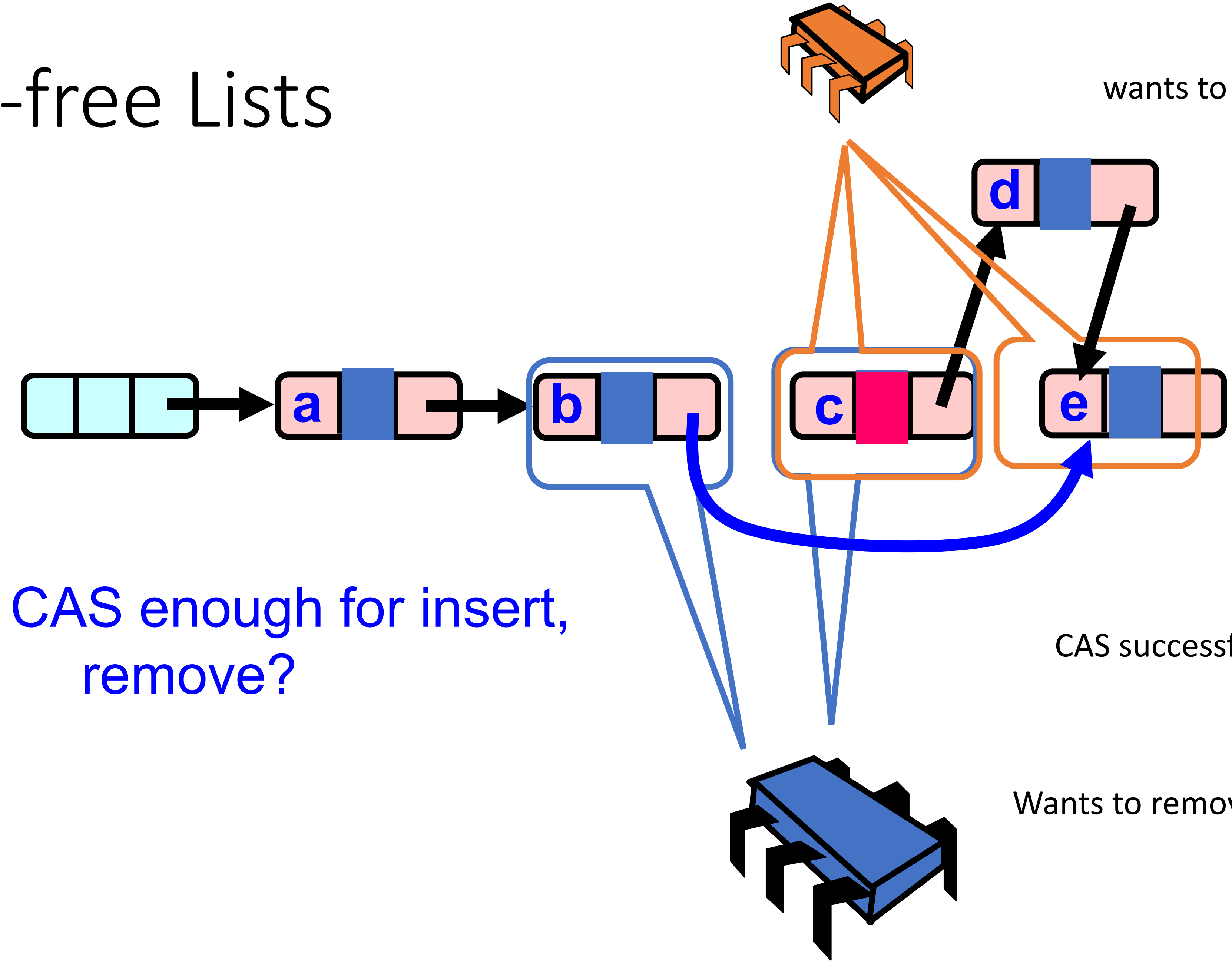
CAS enough for insert, remove?

Lock-free Lists



CAS enough for insert, remove?

Lock-free Lists



wants to insert d

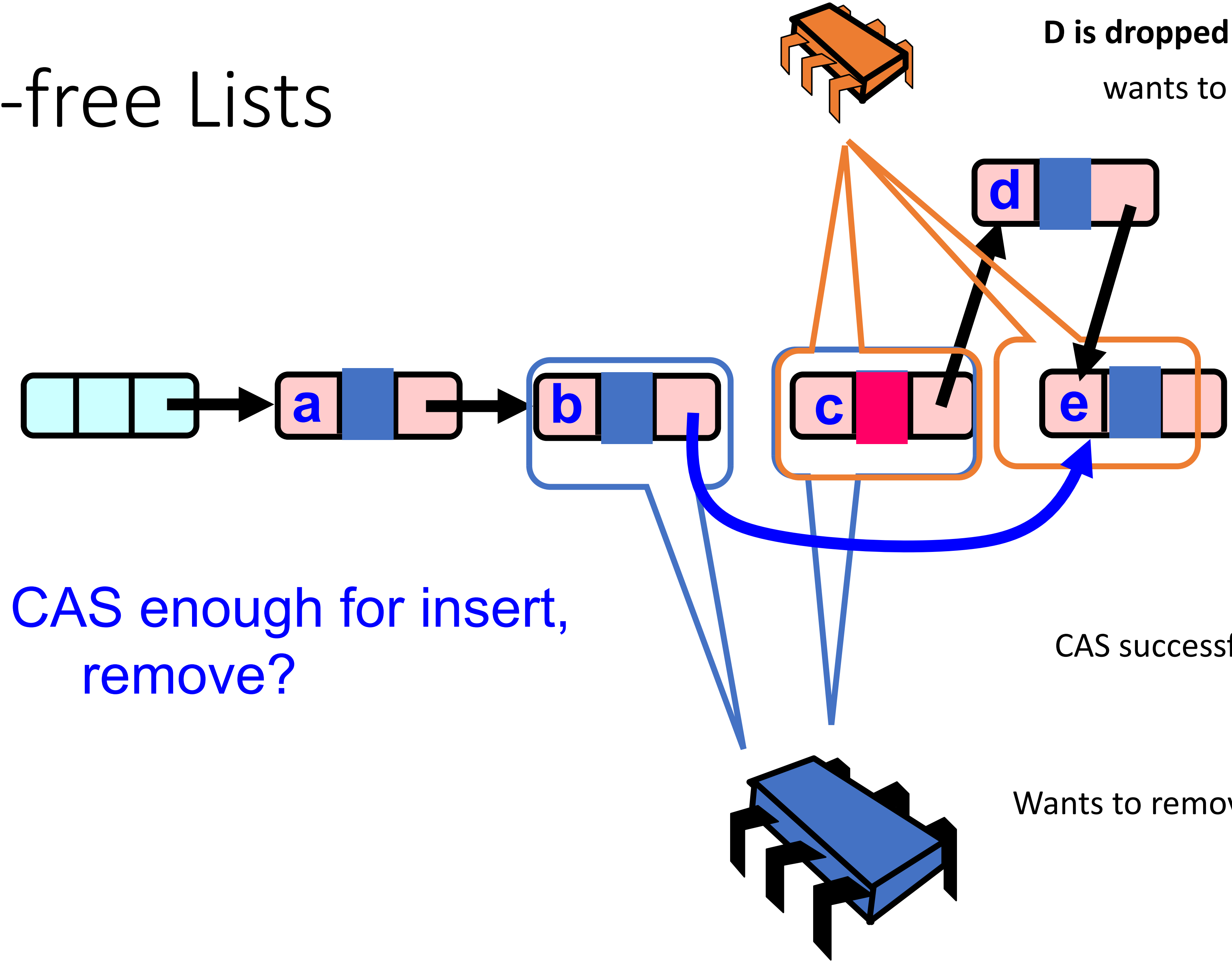
CAS successful!

CAS successful!

Wants to remove c

CAS enough for insert, remove?

Lock-free Lists



D is dropped!

wants to insert d

CAS successful!

CAS enough for insert, remove?

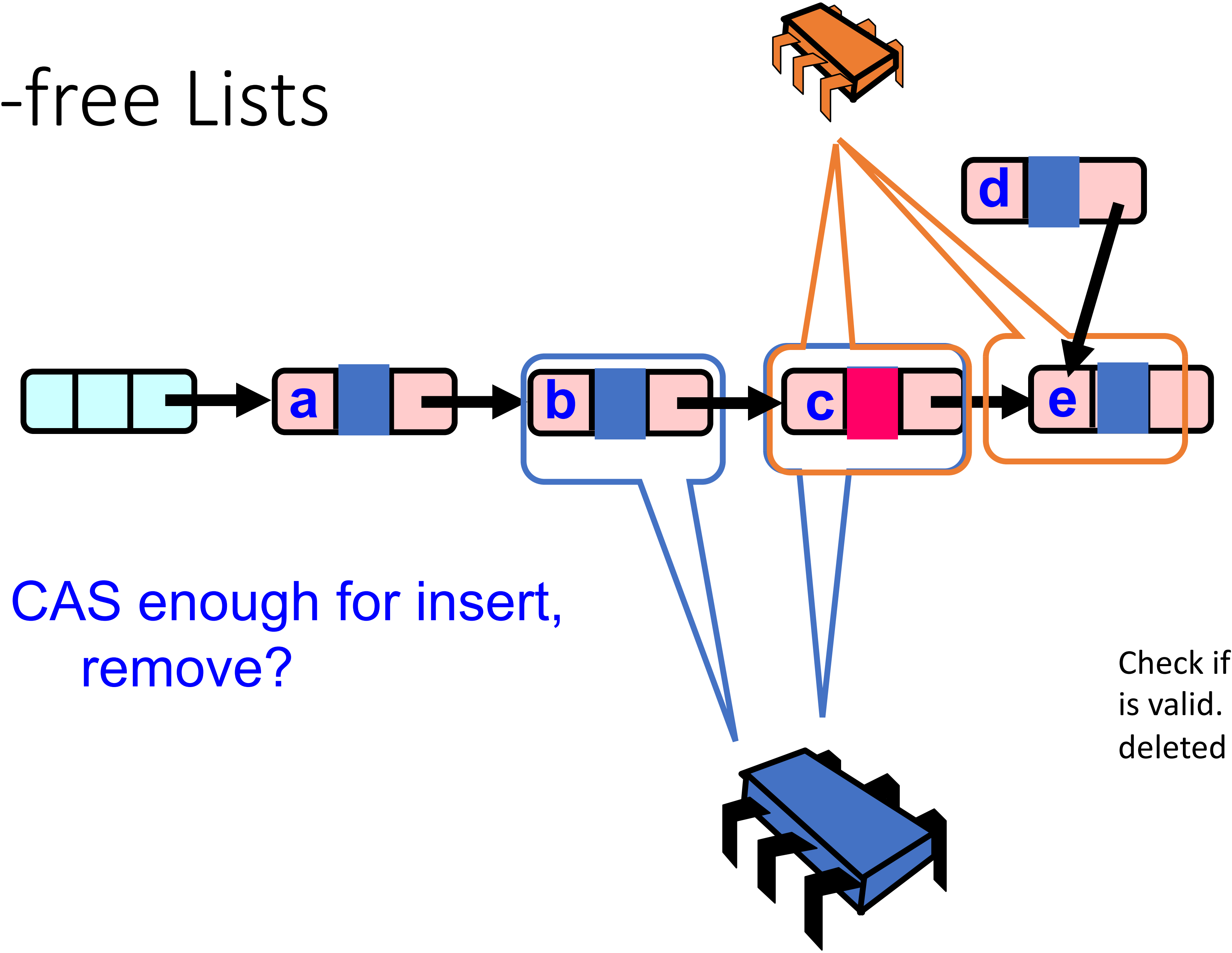
CAS successful!

Wants to remove c

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 physically 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

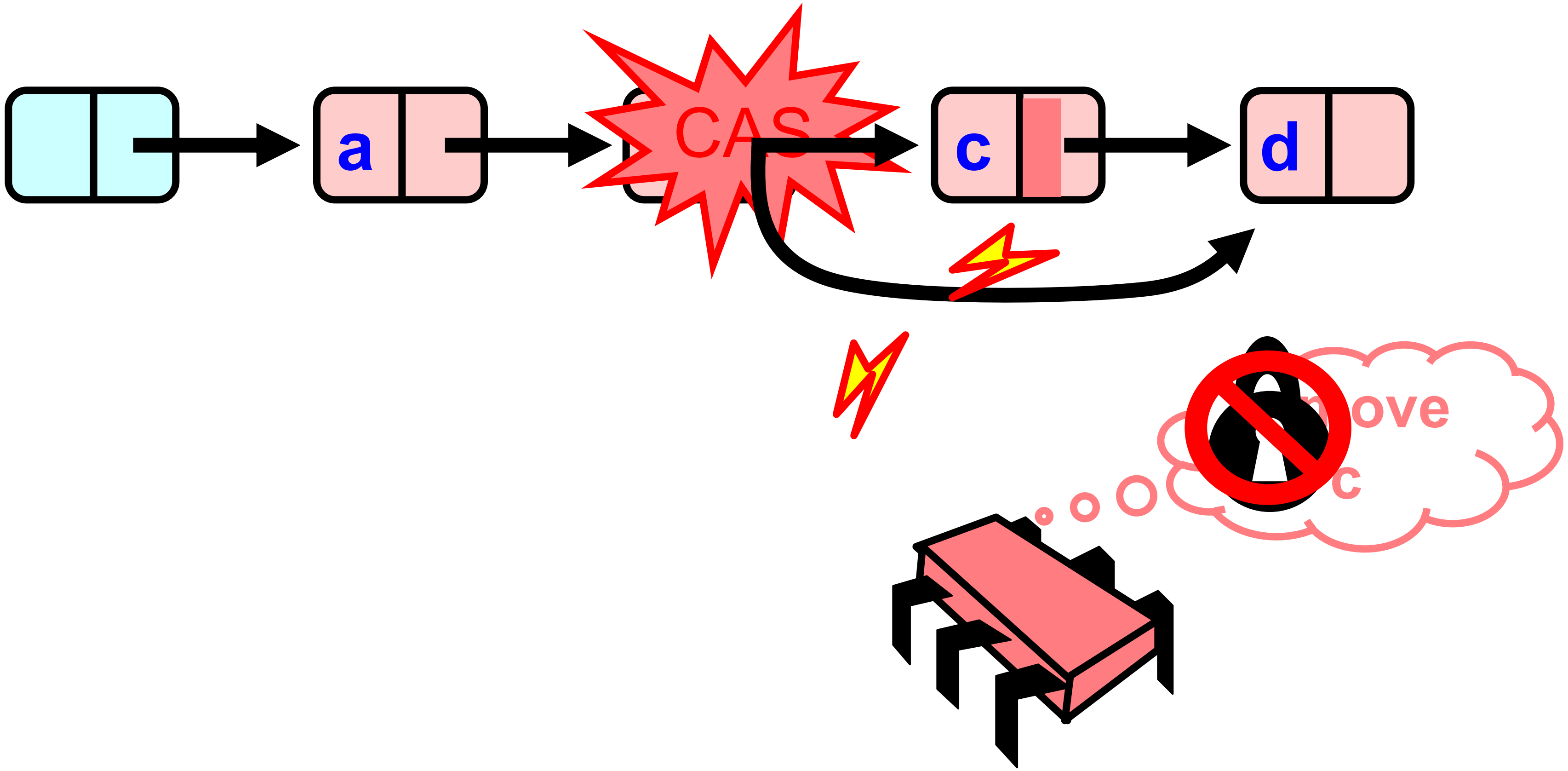
Marking a Node

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

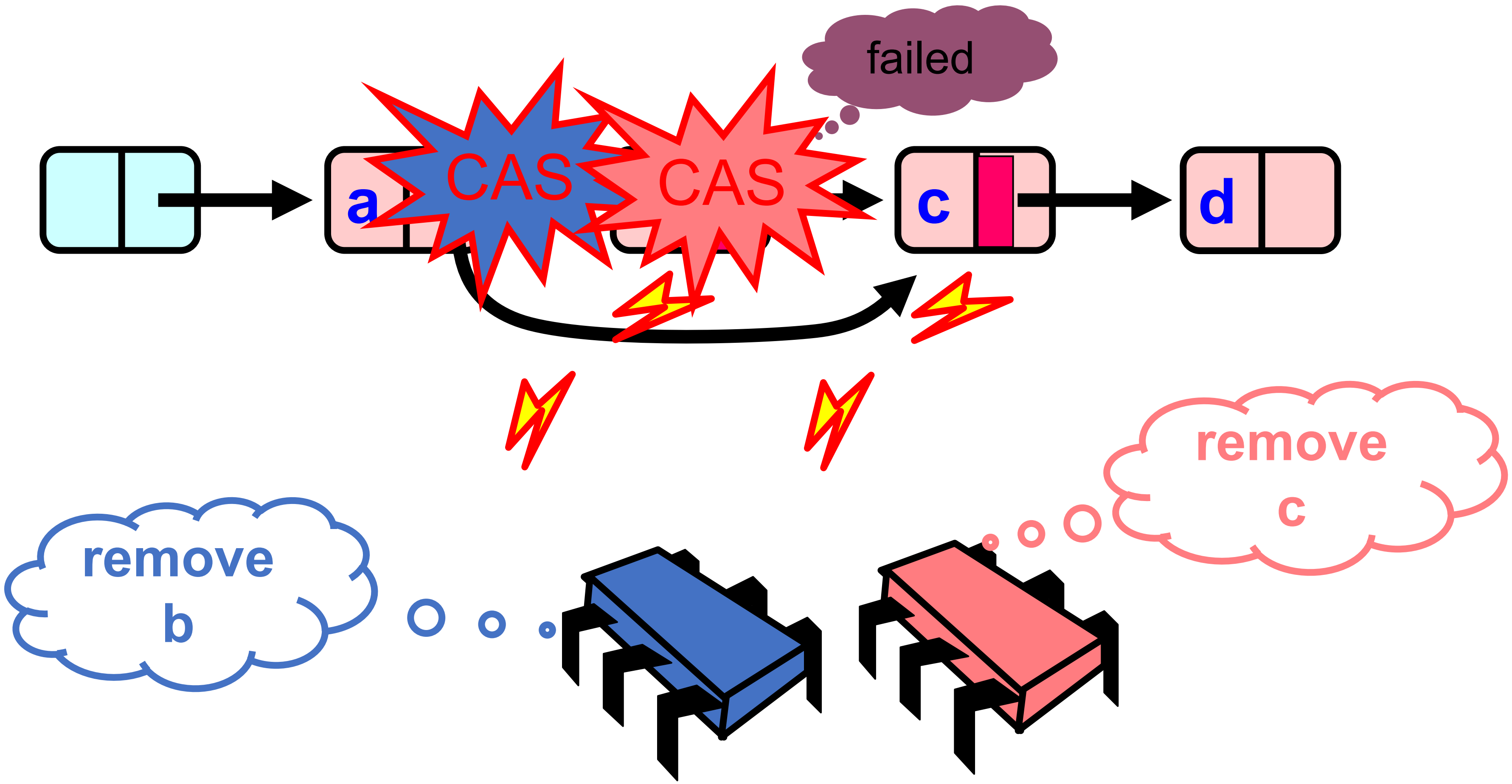
```
class AtomicMarkedNodePtr {
  private:
    atomic<node *> ptr;
  public:
     AtomicMarkedNodePtr(node *p) {
        node * marked = p | 1;
        ptr.store(marked);
     }

     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);

     }
}
```

Lazy node removal

Removing a Node

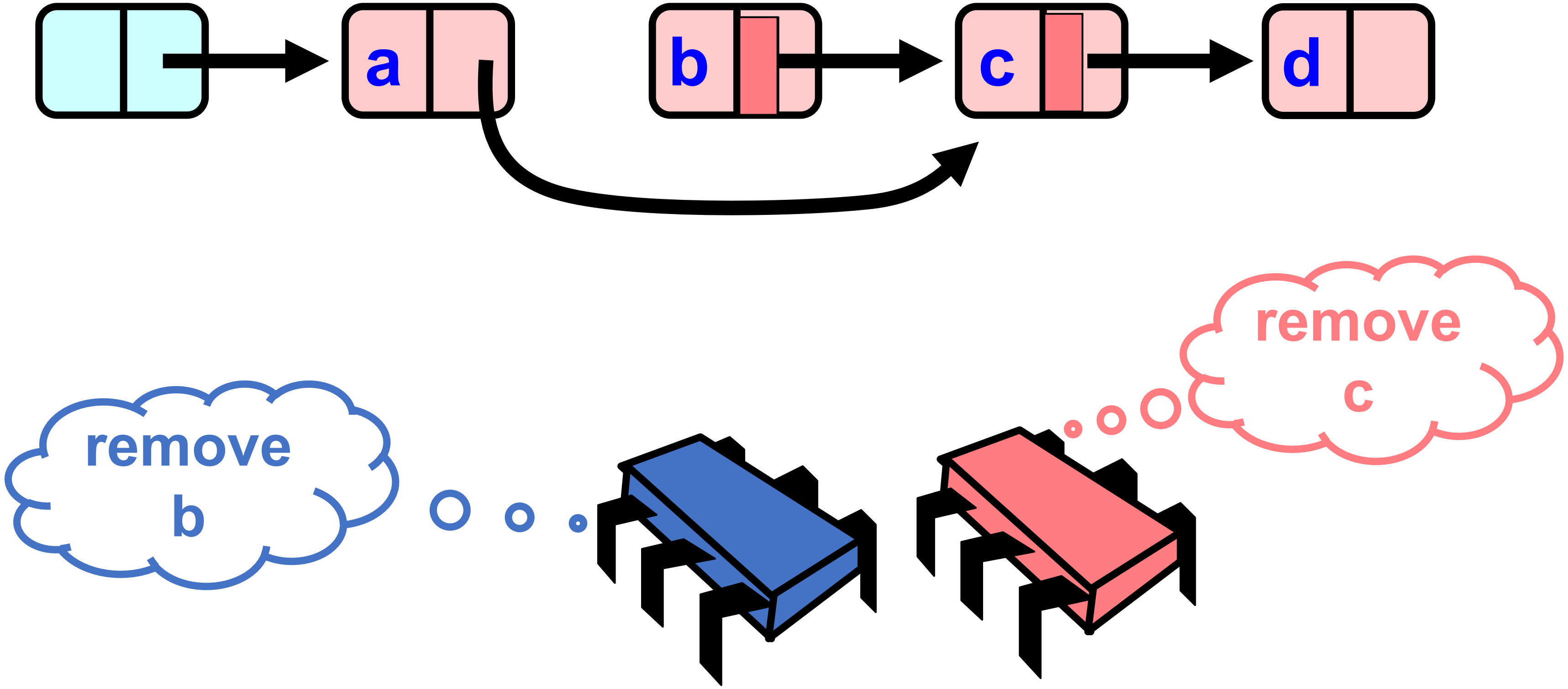


Removing a Node



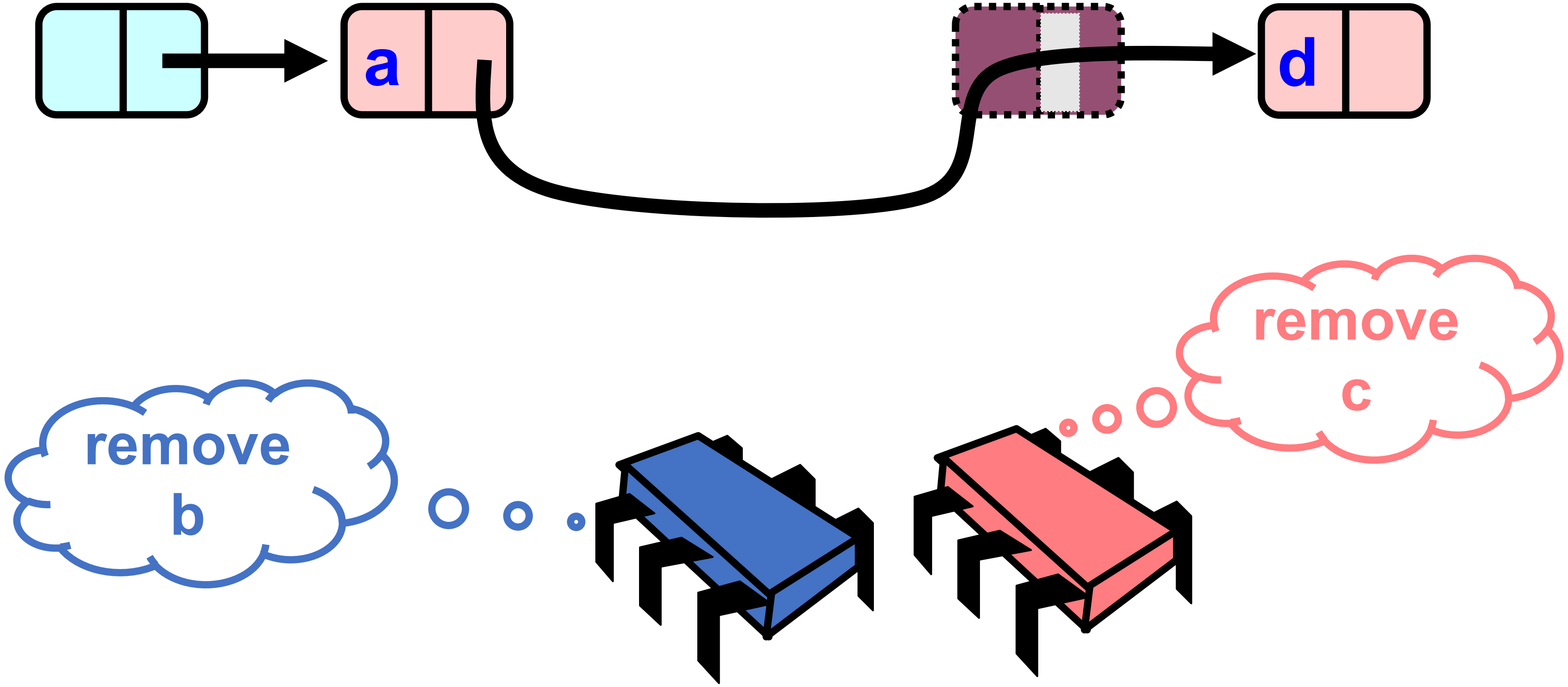
Removing a Node

Two options:
Try removing C again
or...



Removing a Node

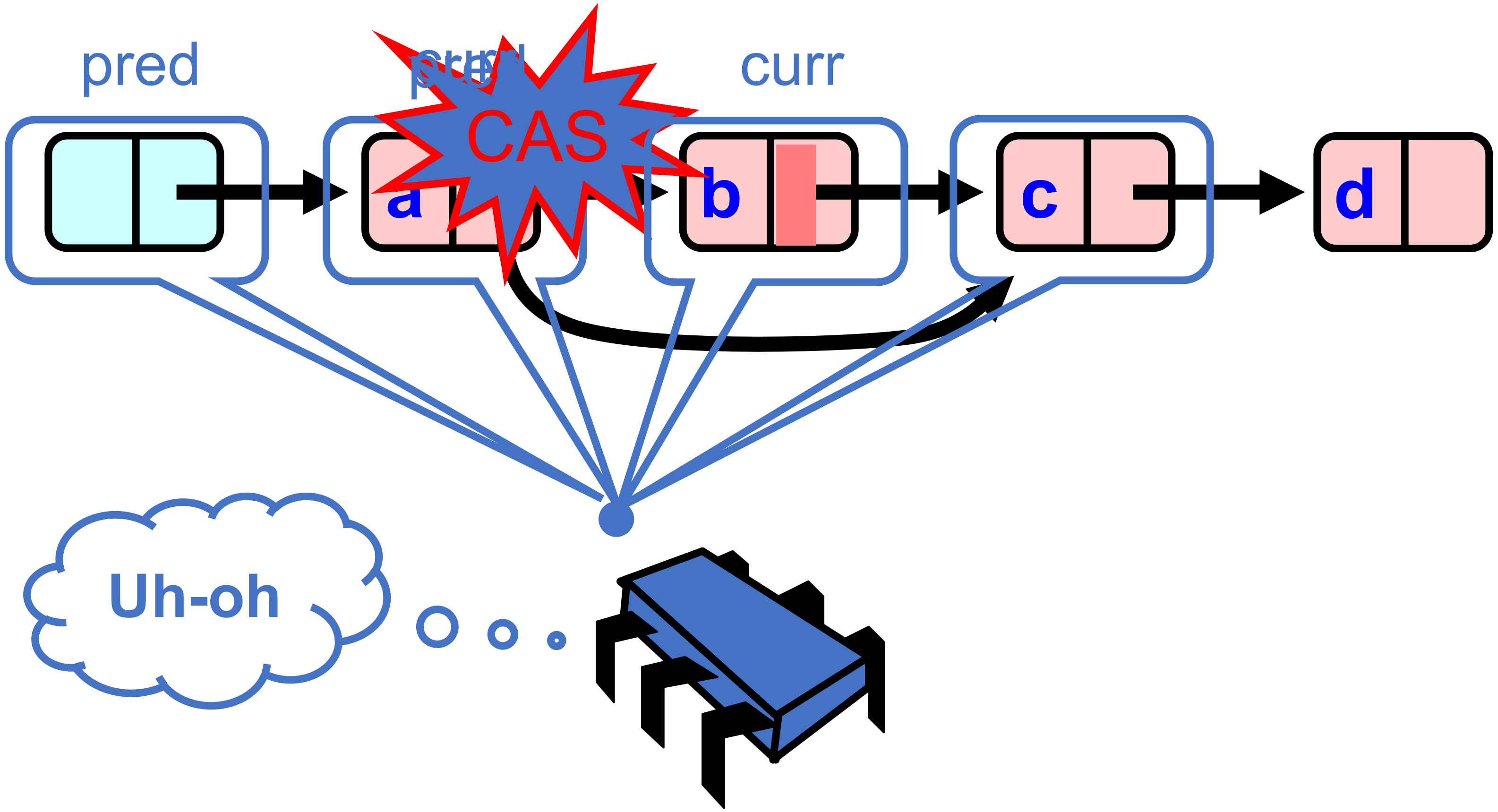
c stays in the list as logically deleted



Traversing the List

- Q: what do you do when you find a “logically” deleted node in your path?
- A: finish the job.
    - CAS the predecessor’s next field
    - Proceed (repeat as needed)

Lock-Free Traversal



Further Reading

- Chapter 9 goes over implementations in detail.
  - This is tricky stuff! Please read to get a different perspective!

- Skip Lists
  - Binary search over linked list (log(n) lookup time)
  - Chapter 14 of the book

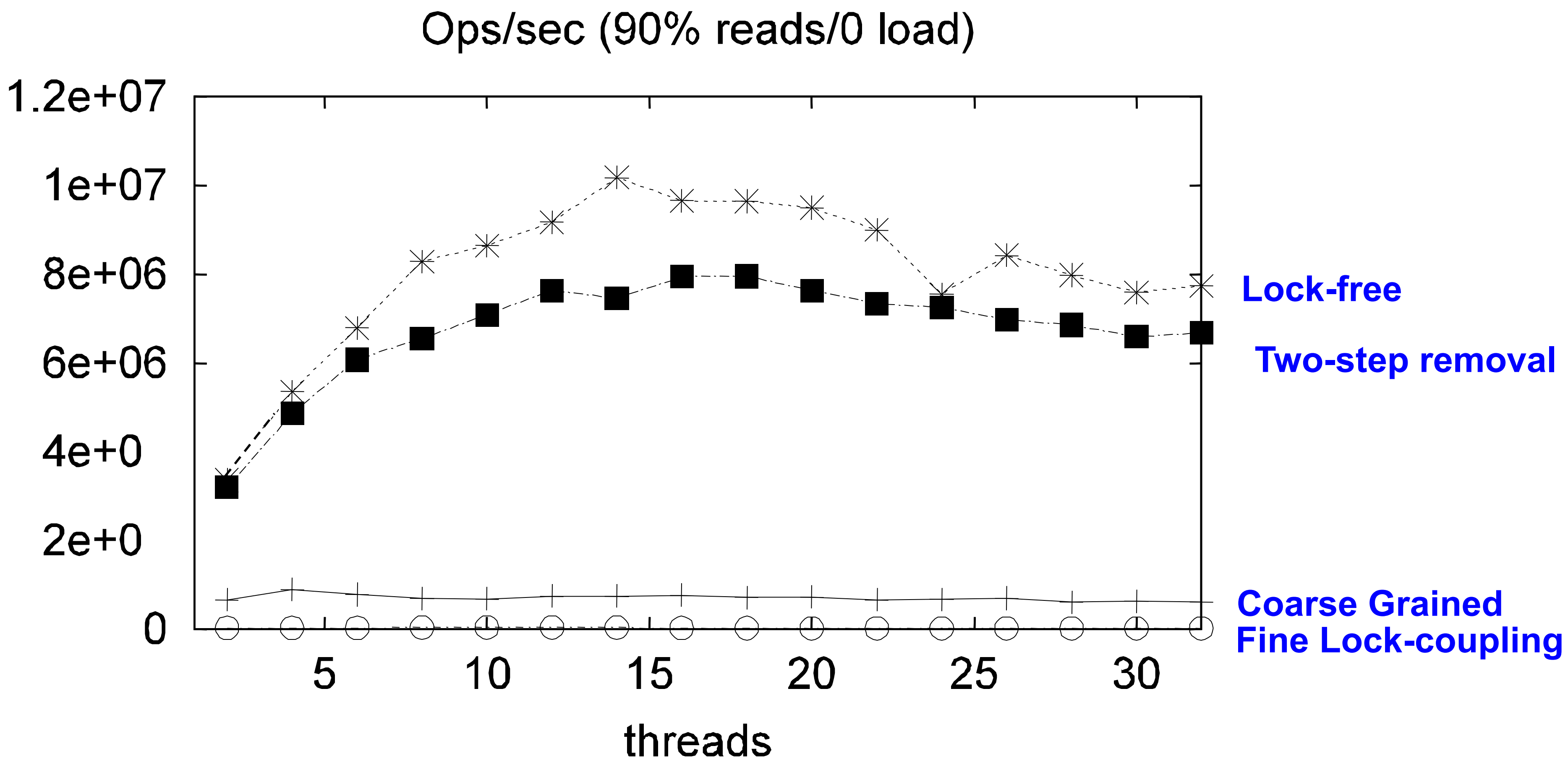
Performance

- Issues:
  - Lazy removal makes benchmarking traversals very tricky
  - Garbage collection makes benchmarking very tricky

Some performance results

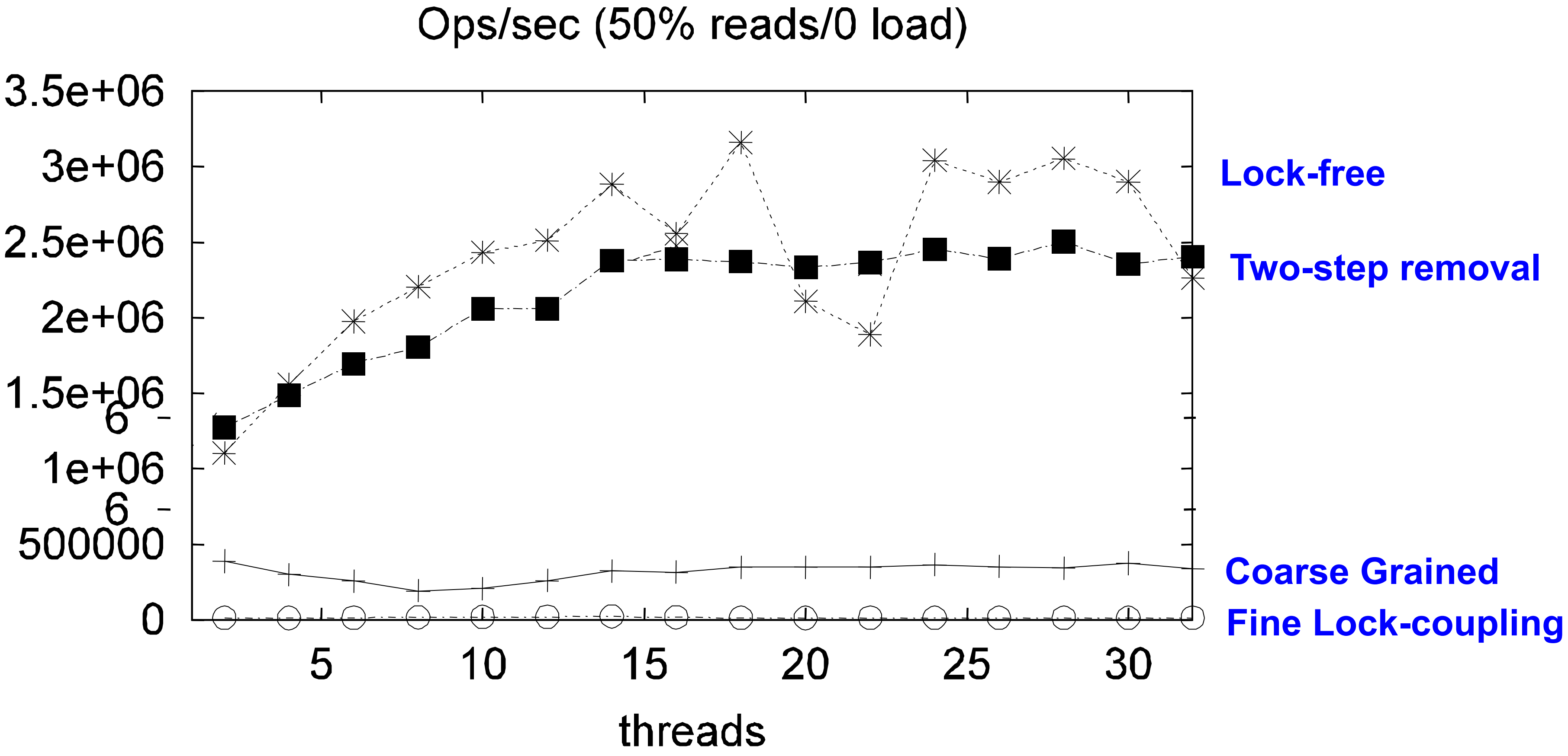
From: A Lazy Concurrent List-Based Set Algorithm: 2005 publication from the textbook authors research group

High Contains Ratio



Low Contains Ratio

noisy!



Next Class

- Concurrent Queues
- Load balancing

- Midterm due!
- HW2 due!
- HW3 assigned!

- Good luck on the exam and HW!