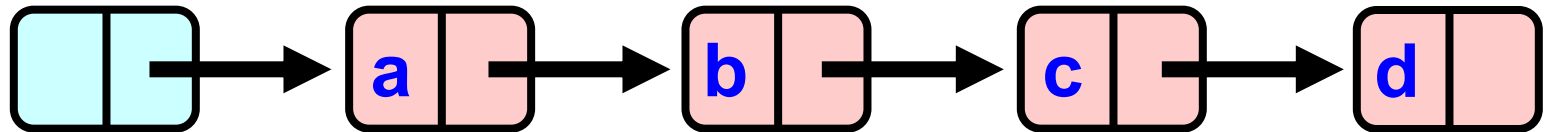
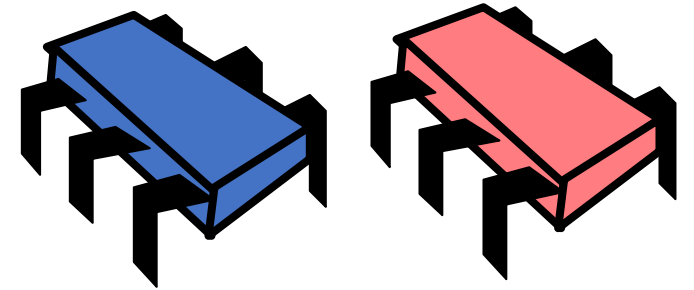


# CSE113: Parallel Programming

- **Topics:**

- Concurrent general set
- Barriers



# Announcements

- HW 4 grades will be released this week.

# Announcements

- HW 5 was released last week.

# Announcements

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

# Quiz

# 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

# 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

# Quiz

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

- ☐ True
- ☐ False



# Quiz

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

- ☒ True
- ☐ False

# 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

# 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

# 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.

# Quiz

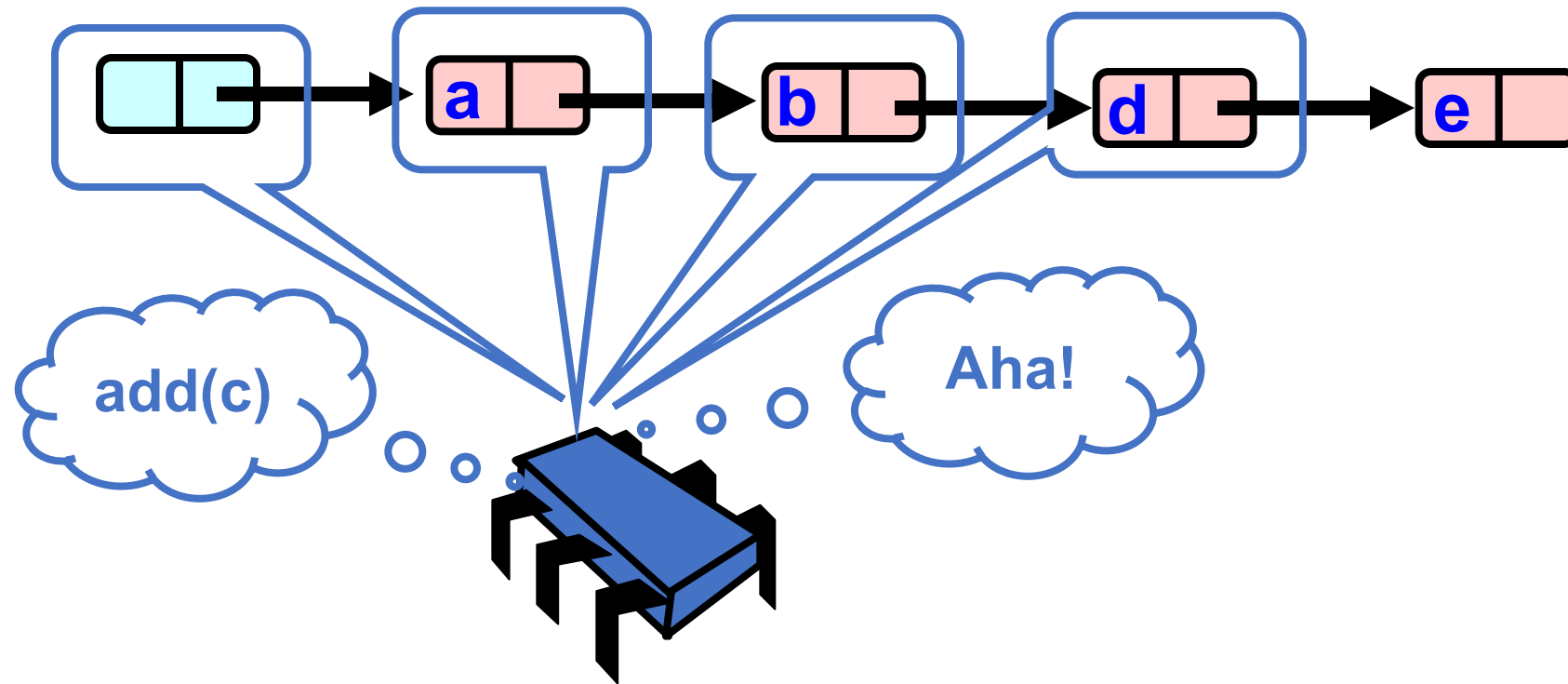
After this lecture, do you think you would be able to optimize your implementation of the concurrent stack in homework 2? Write a few sentences on what you might try.

Coarse-grained vs. Fine-grained locking?

# Review

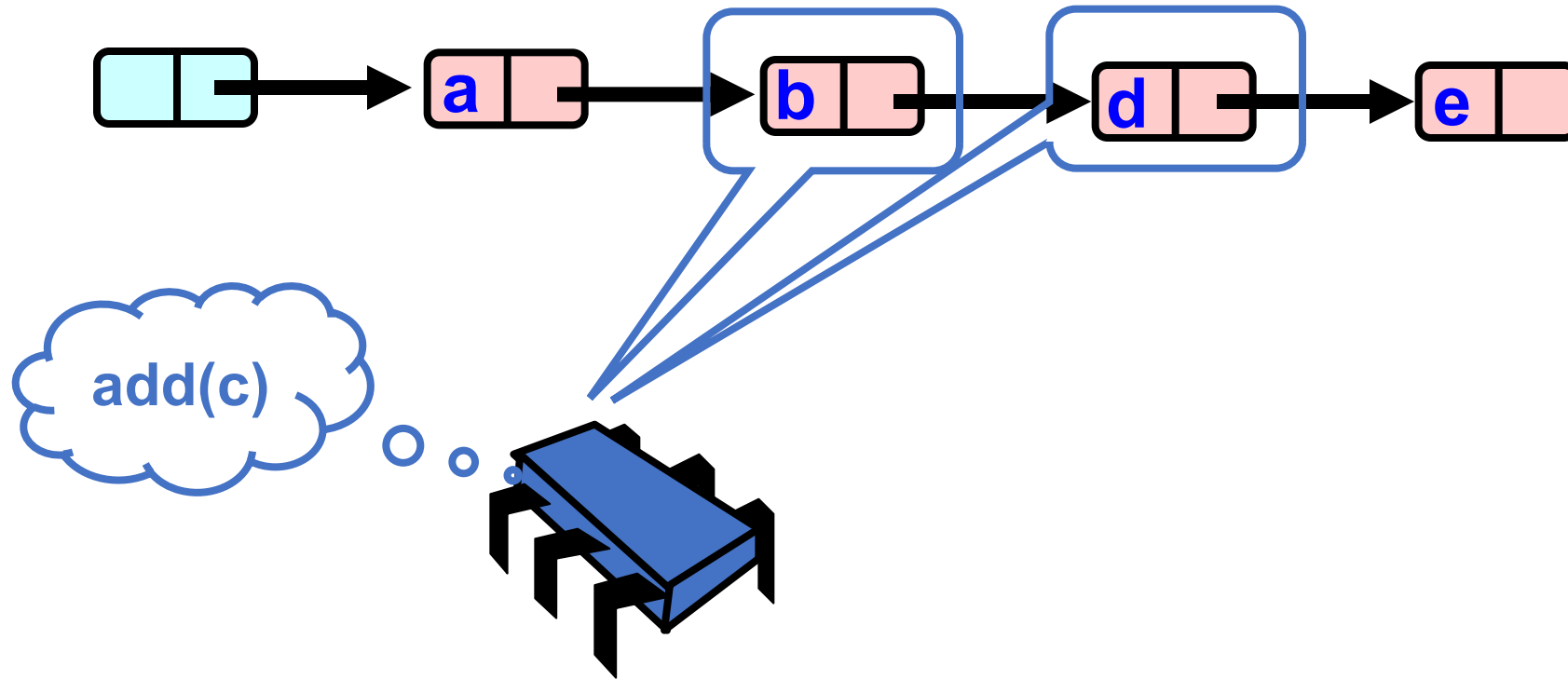
Add and remove: What could go wrong?

# Add and remove: What could go wrong?

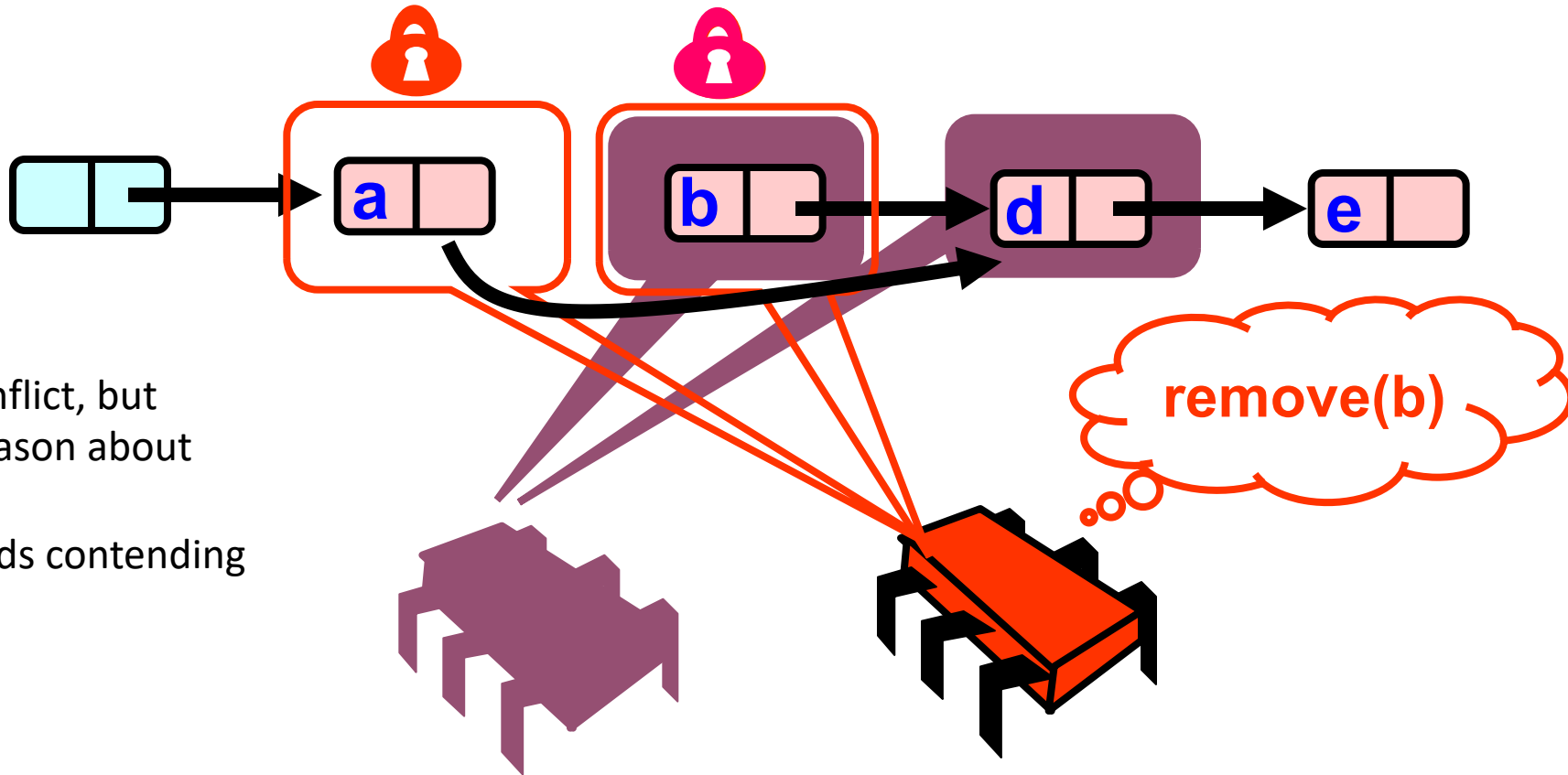




# What could go wrong?



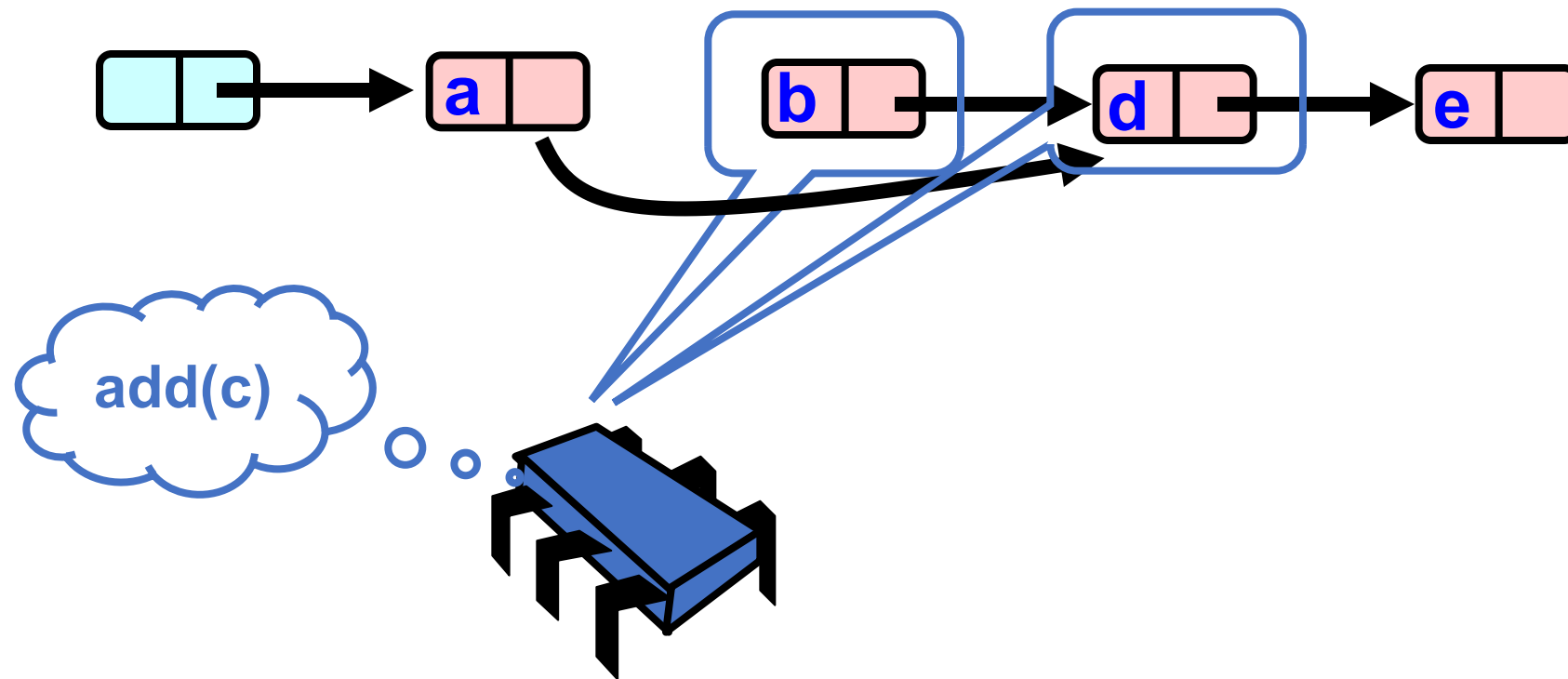
# What could go wrong?



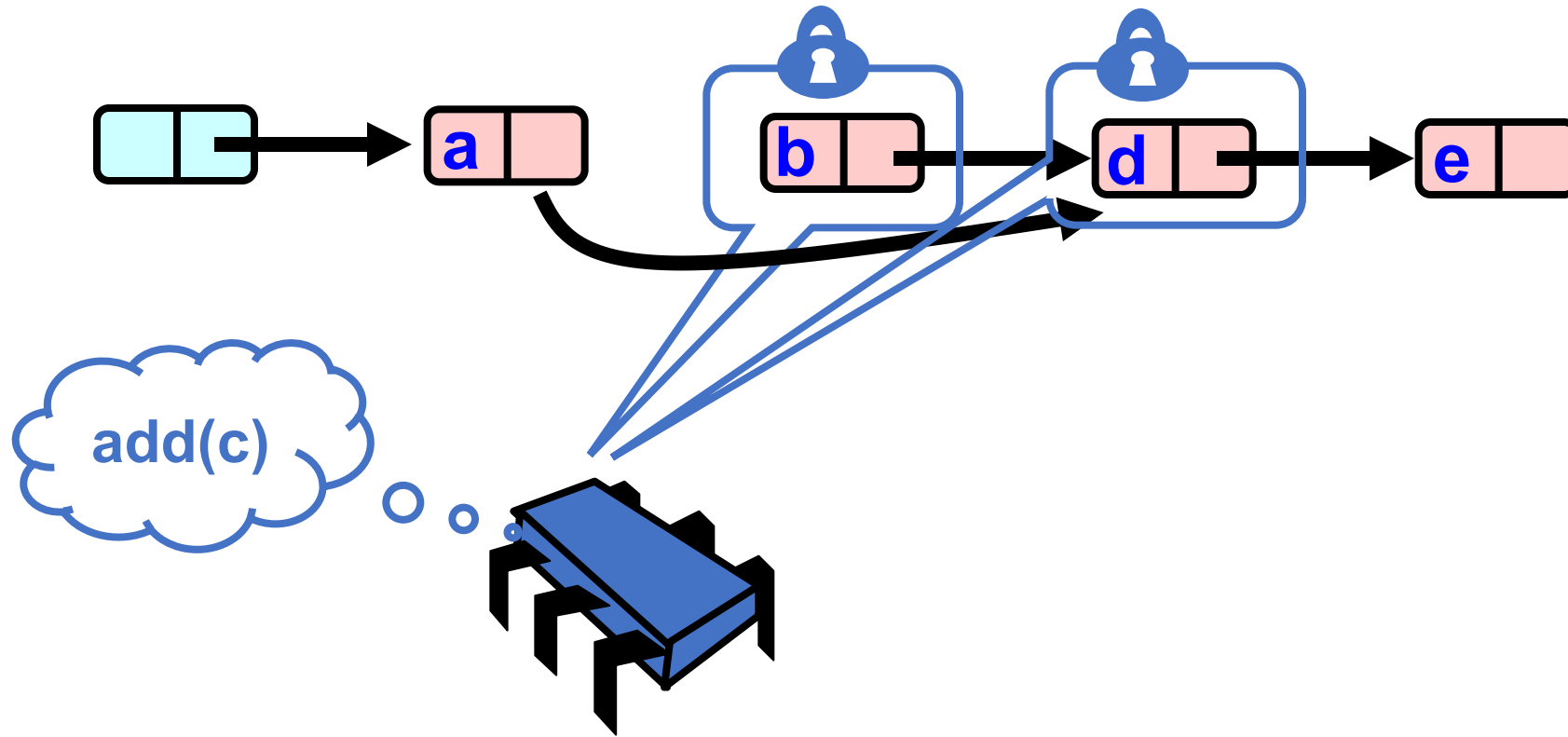
No more data conflict, but we do need to reason about interleavings.

Concurrent threads contending for values.

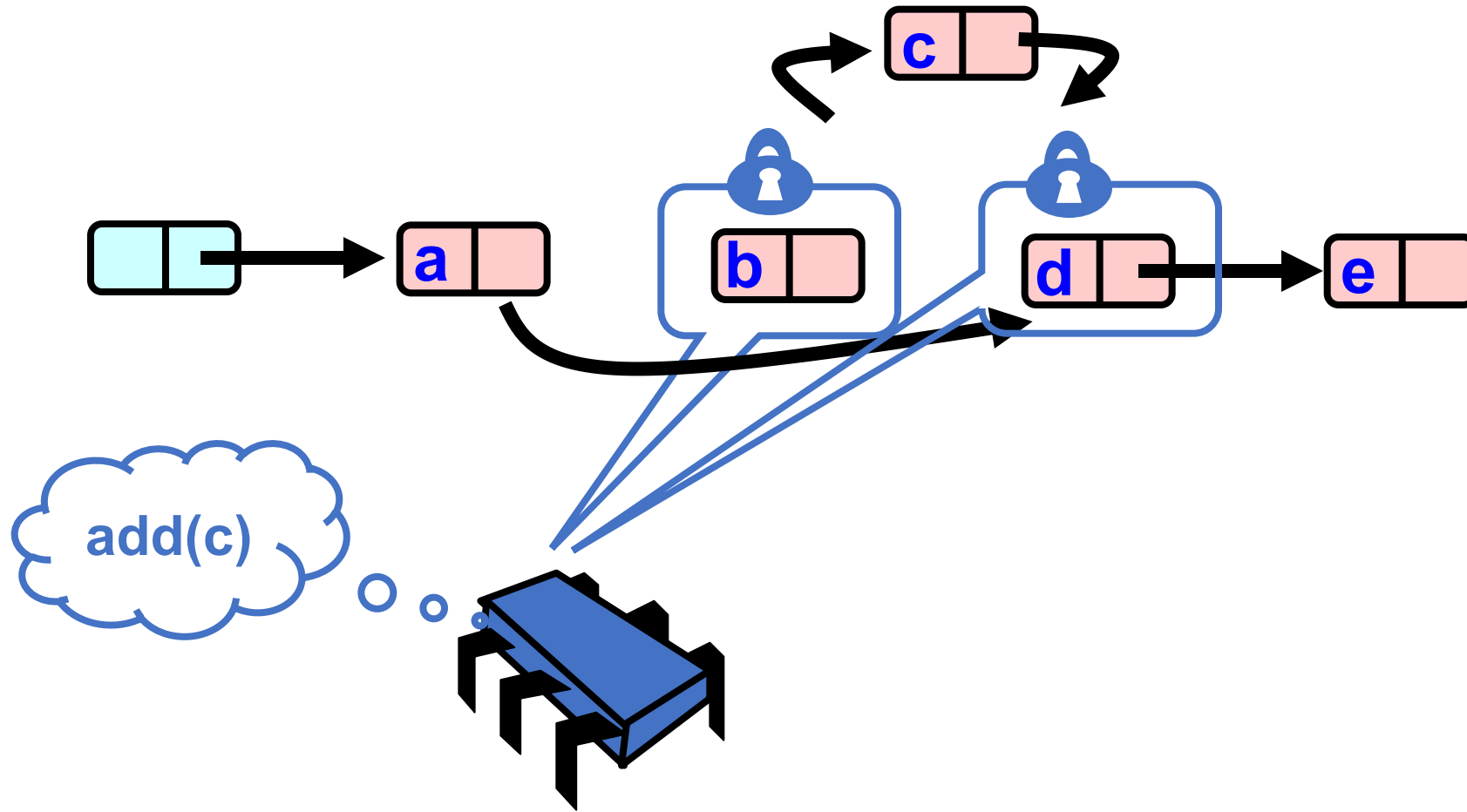
# What could go wrong?



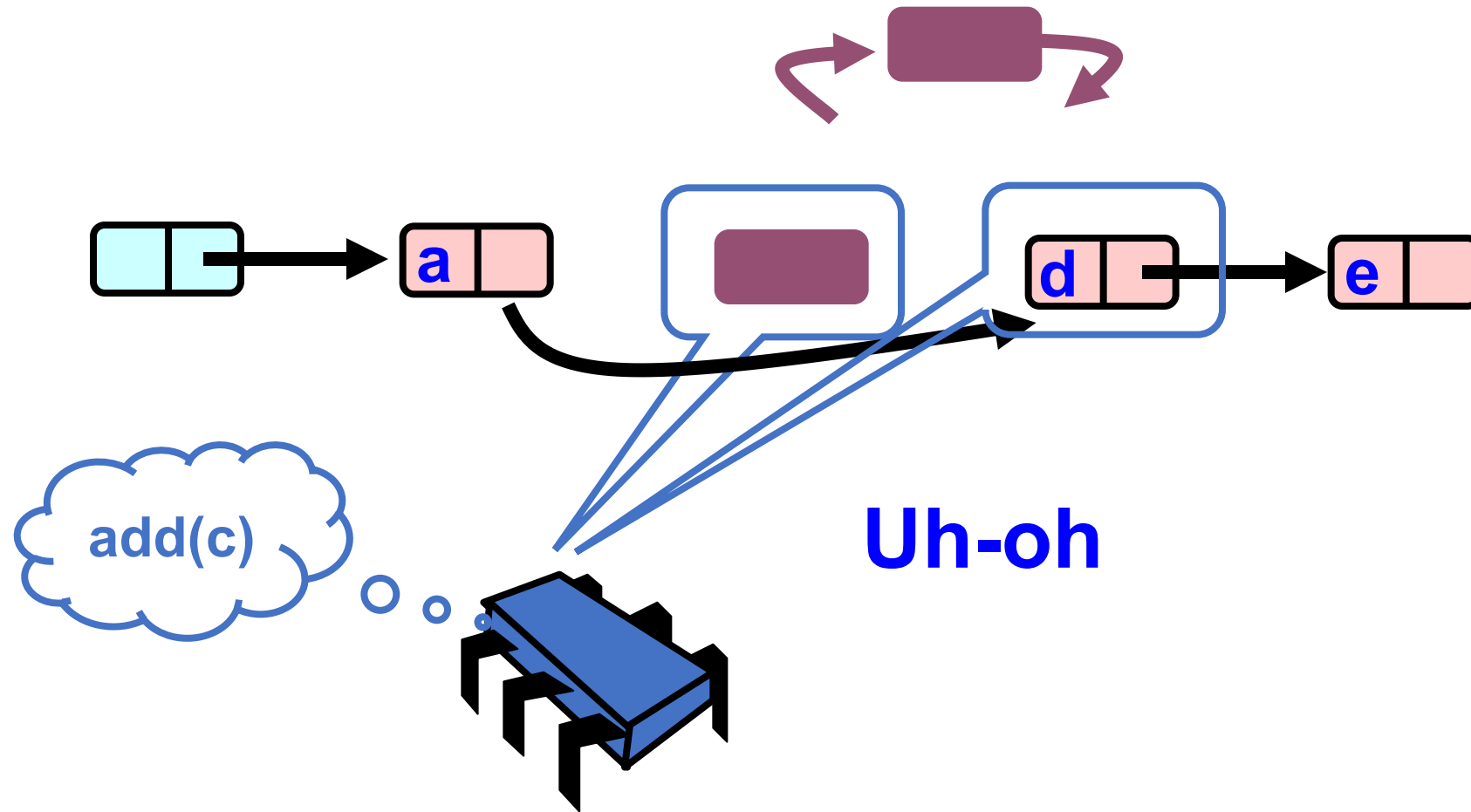
What could go wrong?



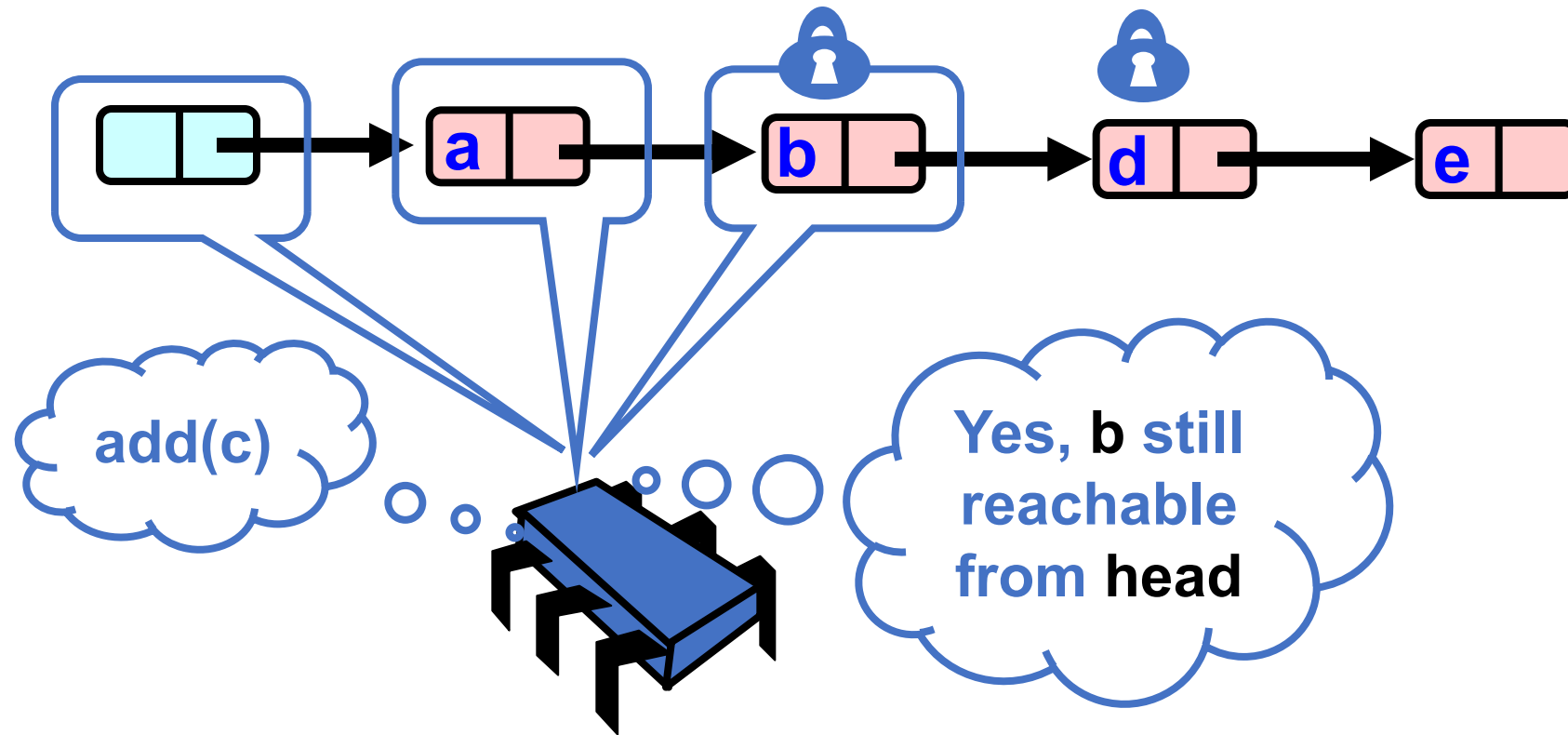
What could go wrong?



What could go wrong?



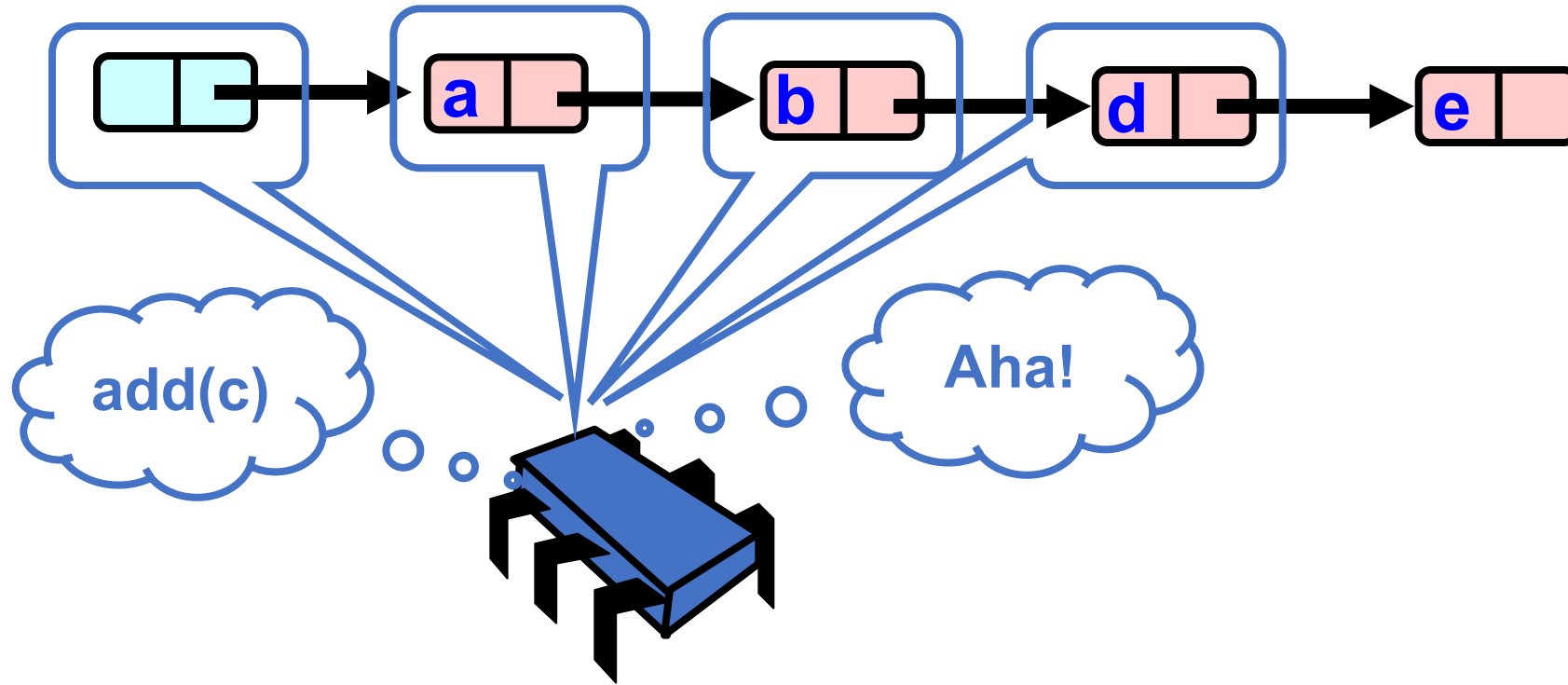
# Validate – Part 1



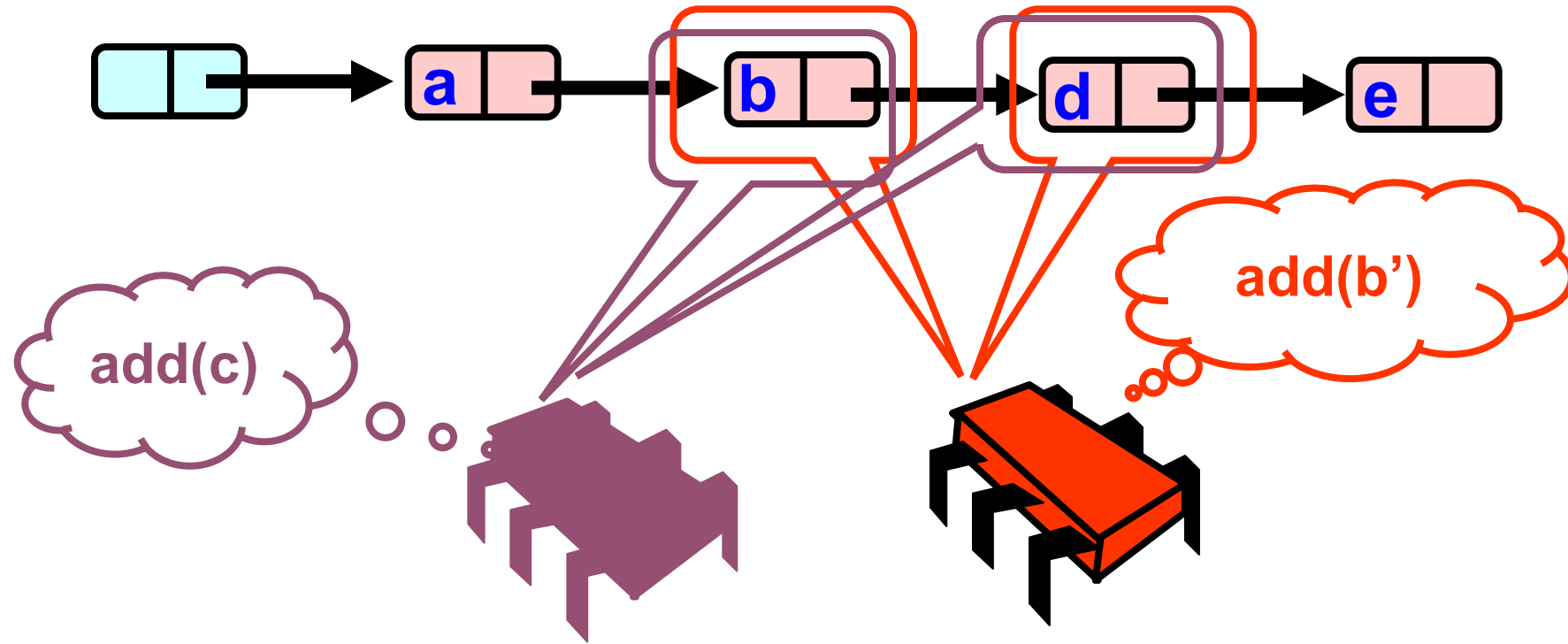
Add and Add: What could go wrong?



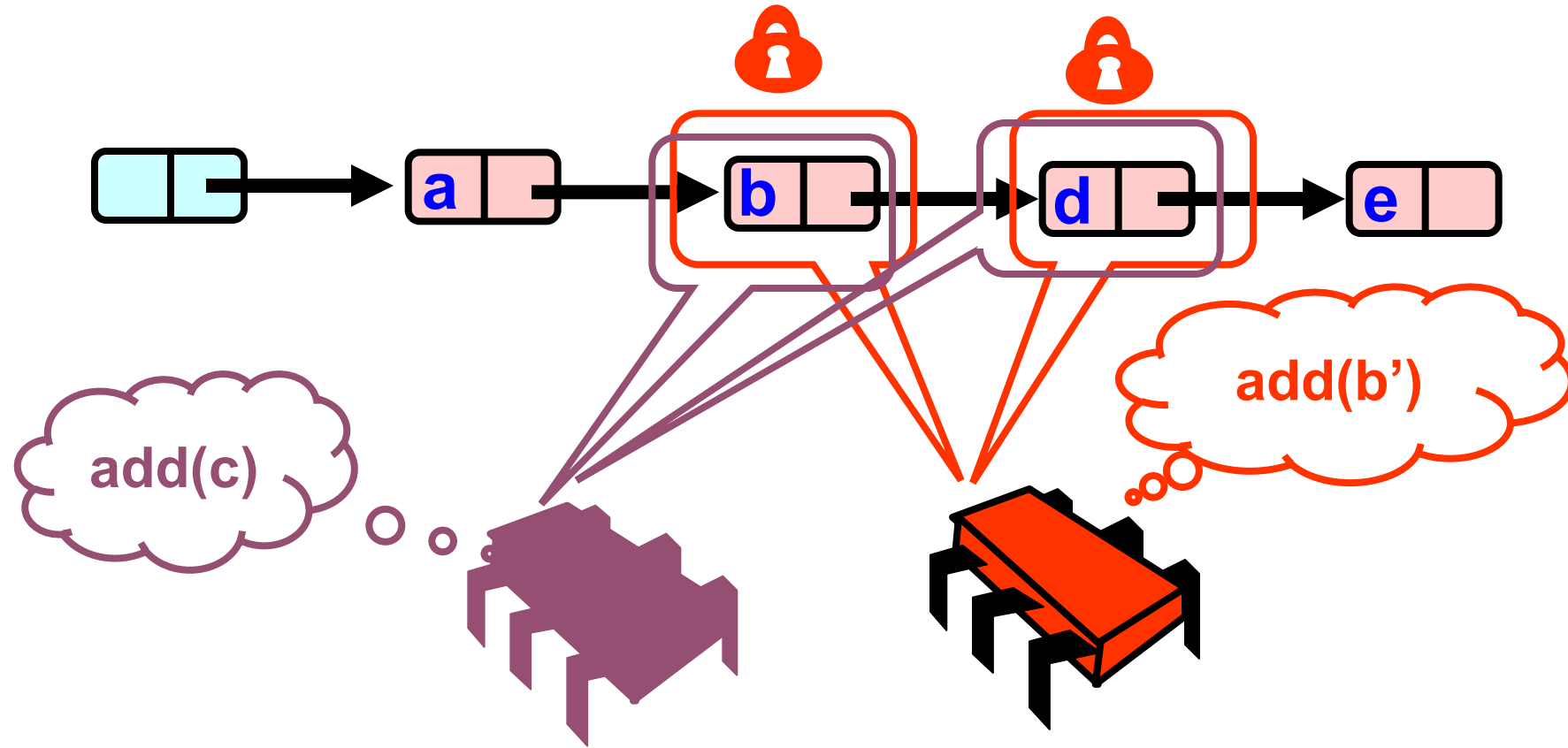
# What Else Could Go Wrong?



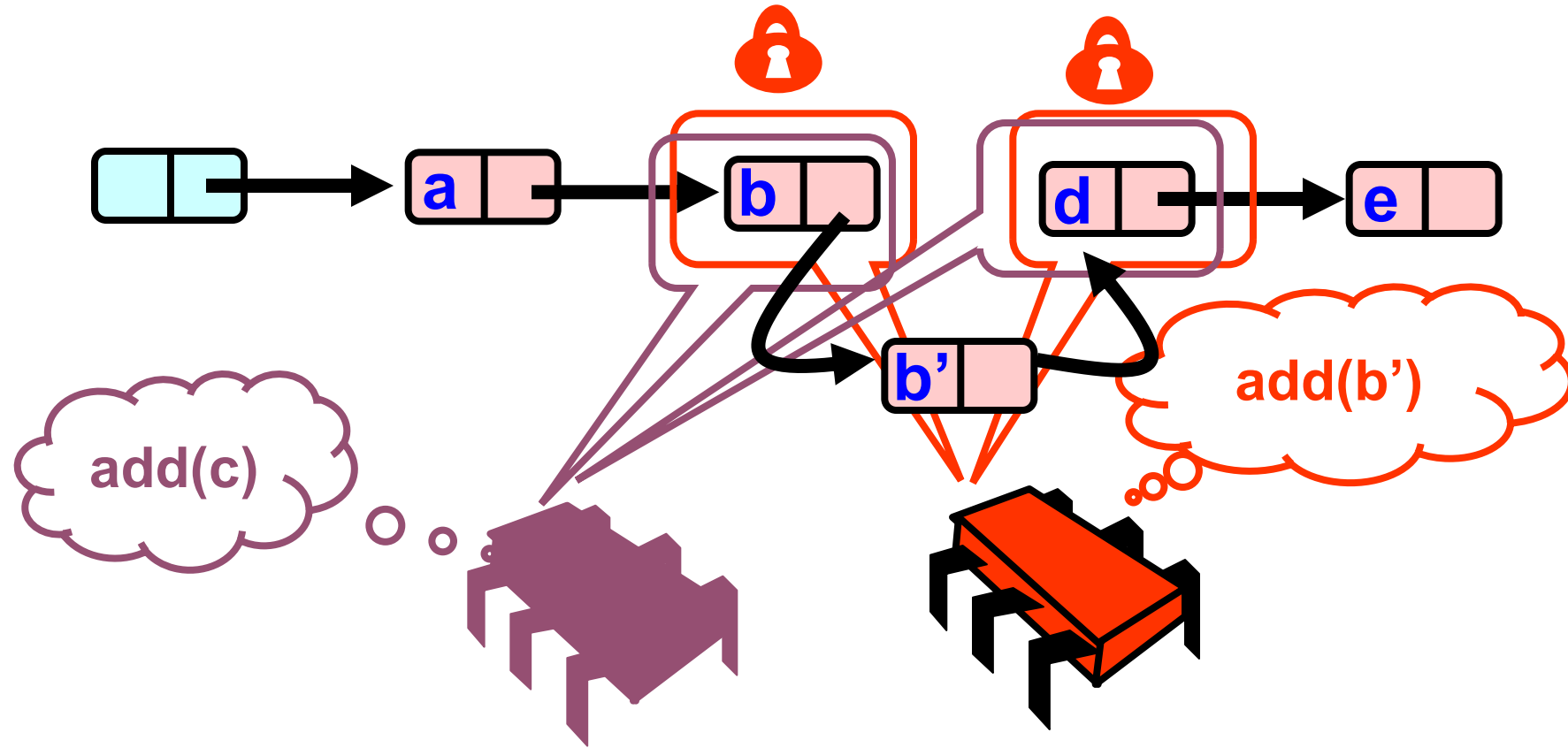
# What Else Could Go Wrong?



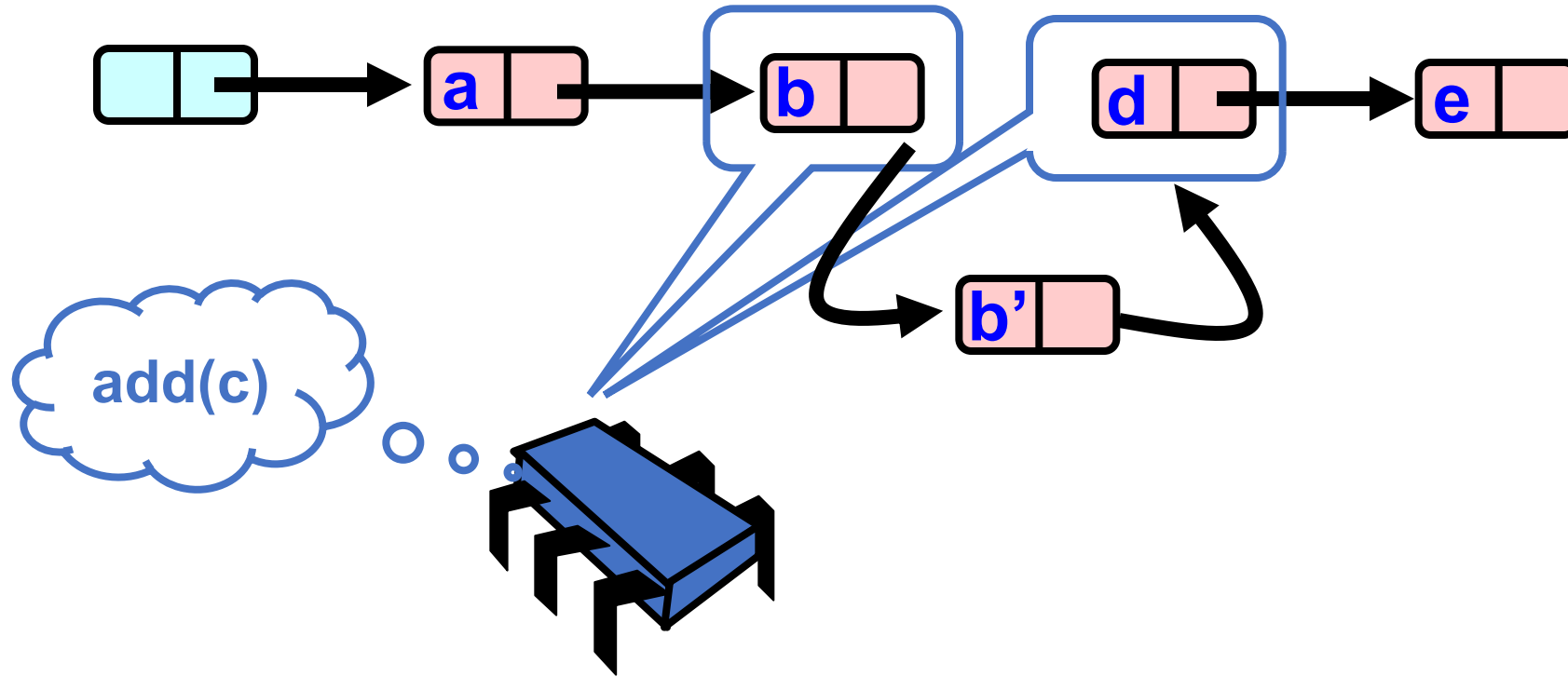
# What Else Could Go Wrong?



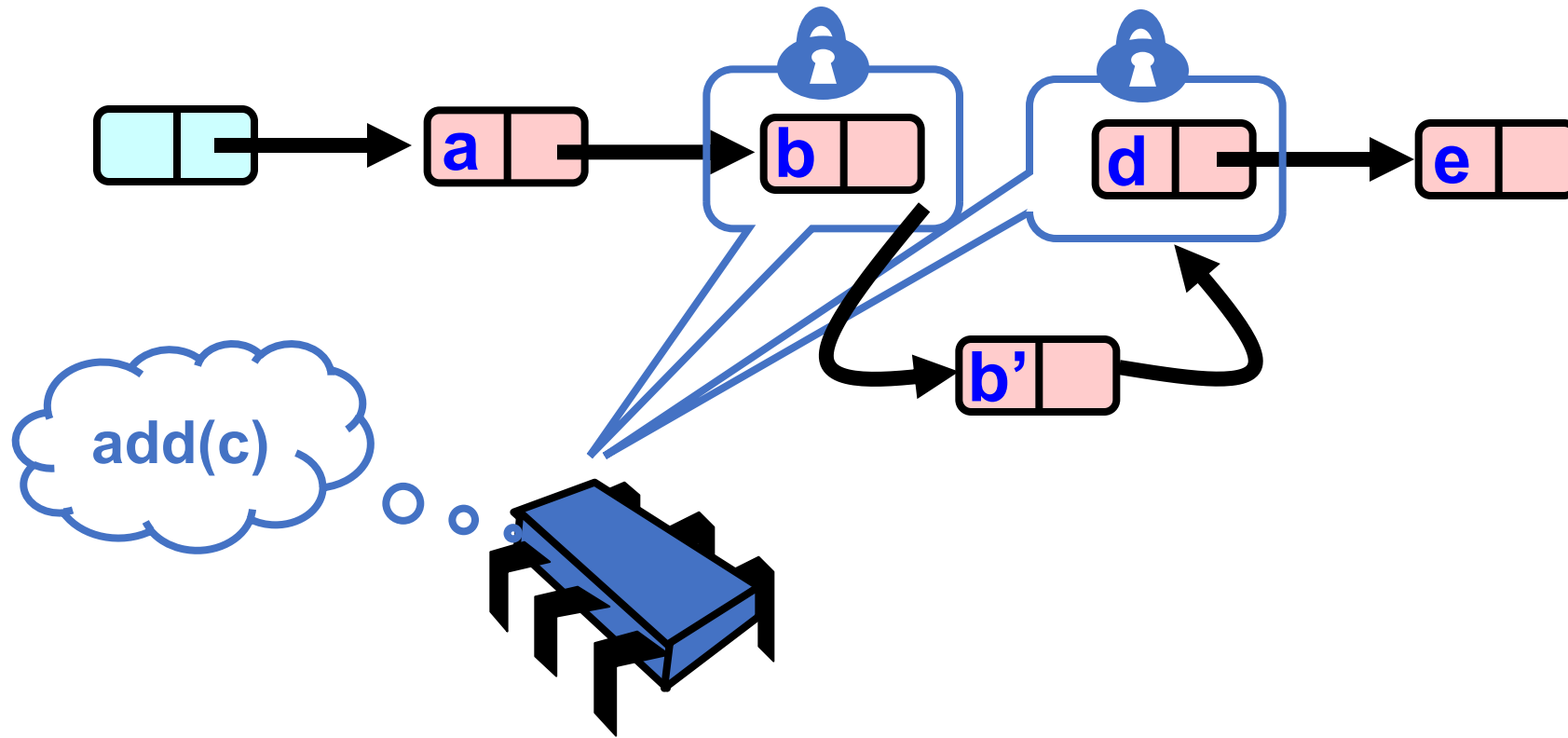
# What Else Could Go Wrong?



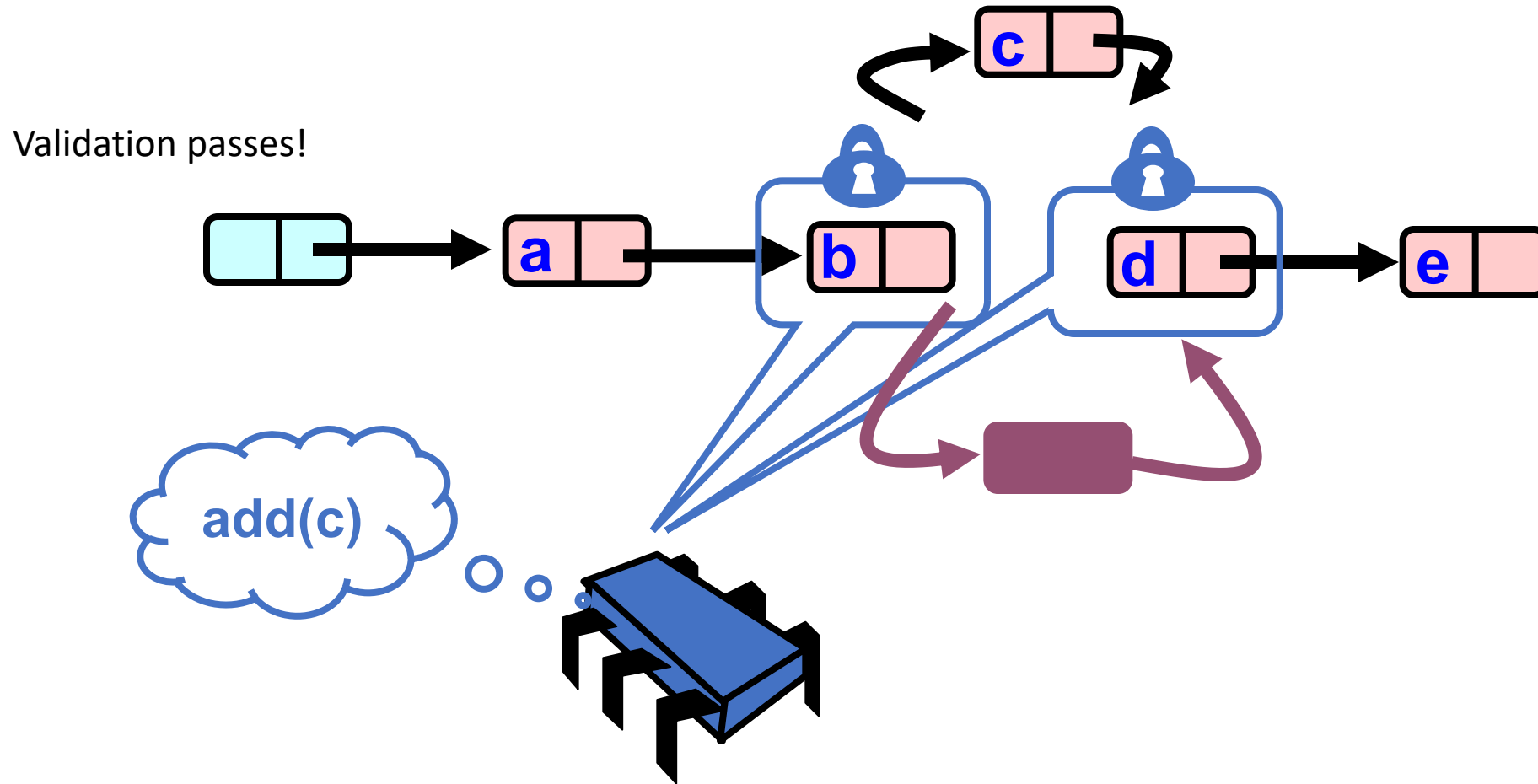
# What Else Could Go Wrong?



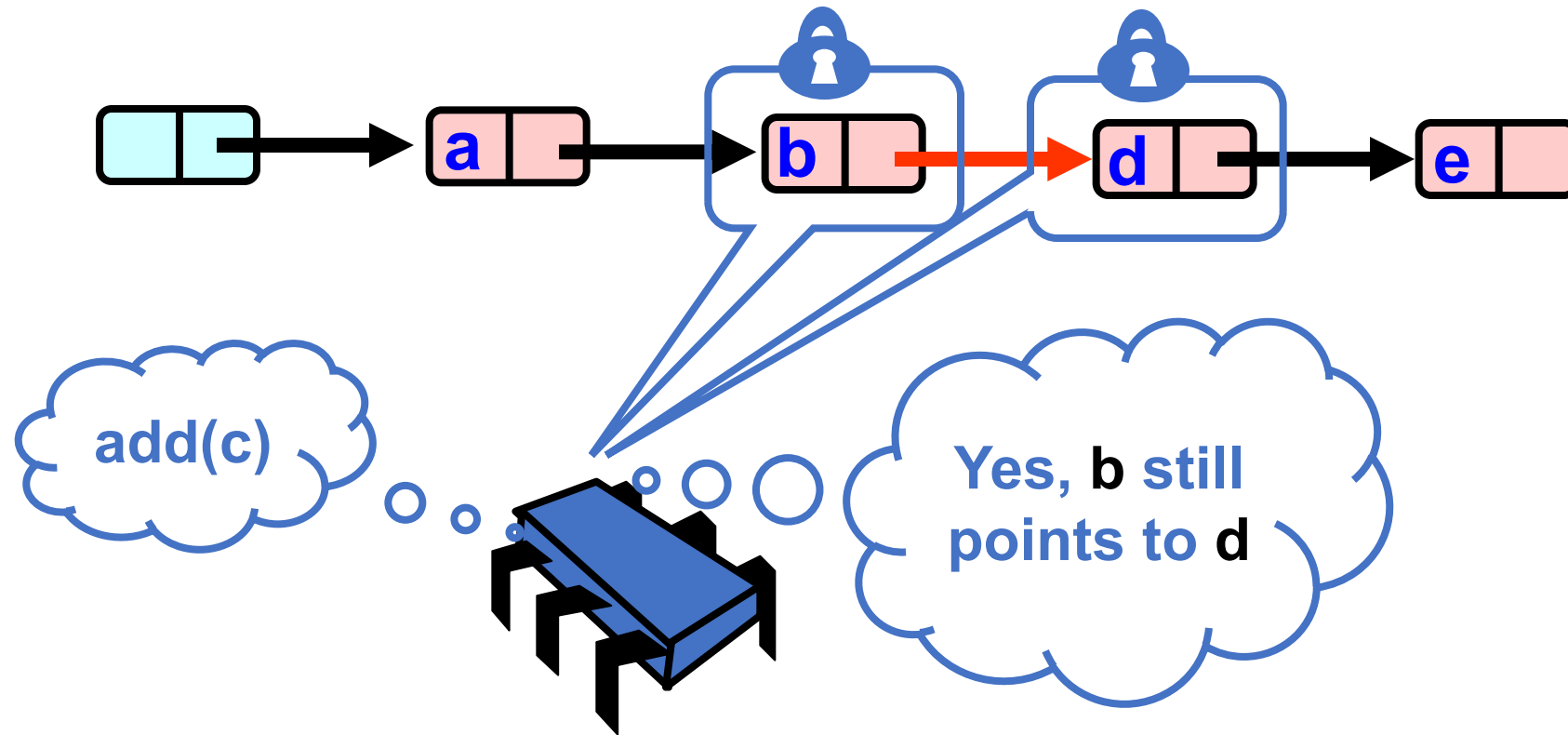
# What Else Could Go Wrong?



# What Else Could Go Wrong?

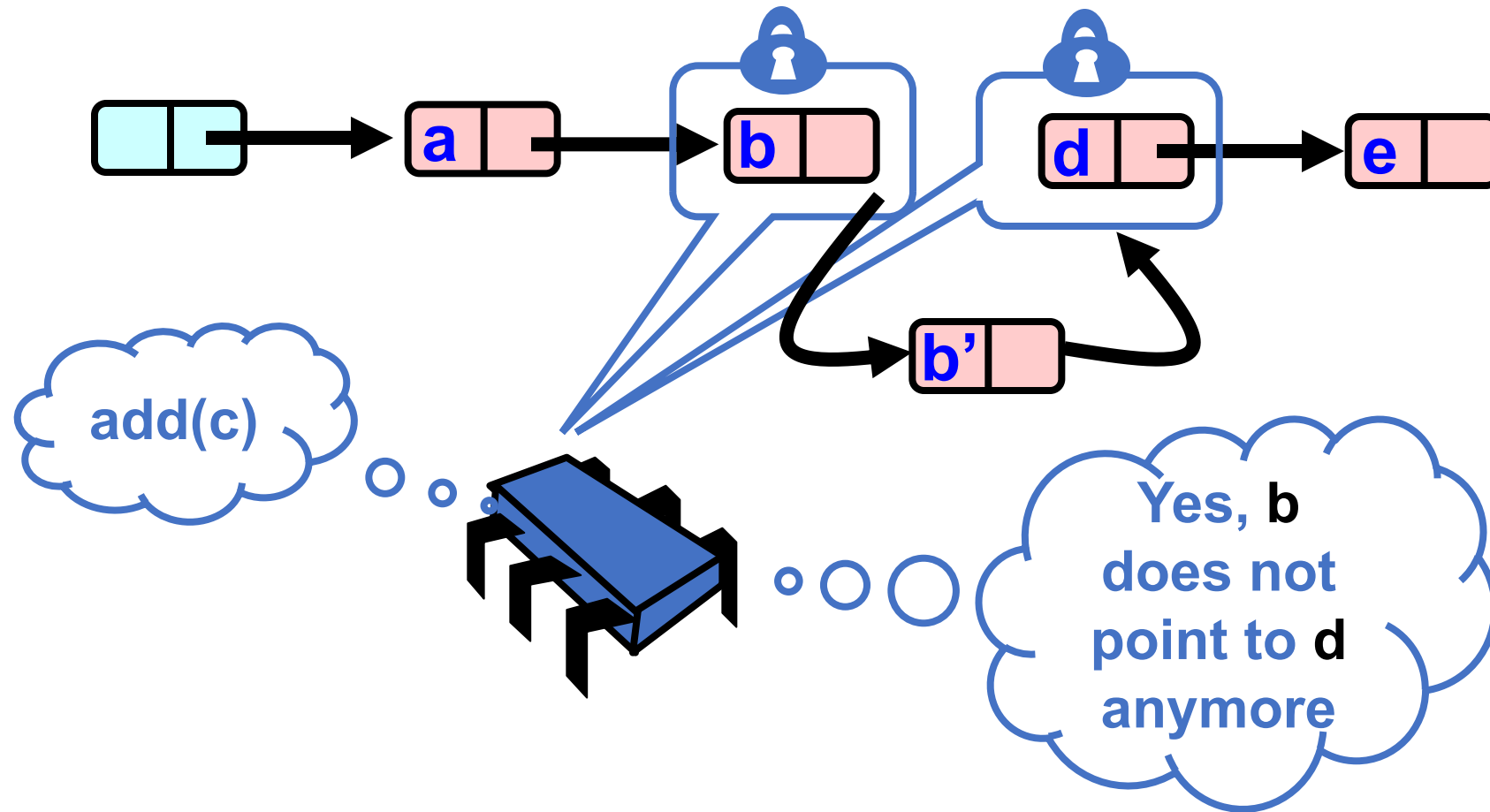


## Validate Part 2 (while holding locks)





# What Else Could Go Wrong?



New Material

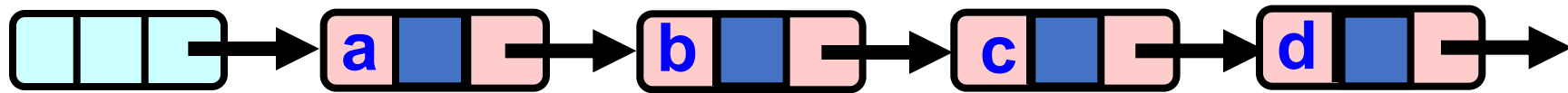
# Can we optimize more?

- Scan the list once?
- We need to make sure that the node is not removed.
- Instead of scanning to check reachability, leave a mark on removed nodes. Now the information that a node is removed is inside the node; we don't need to get it from the predecessor.

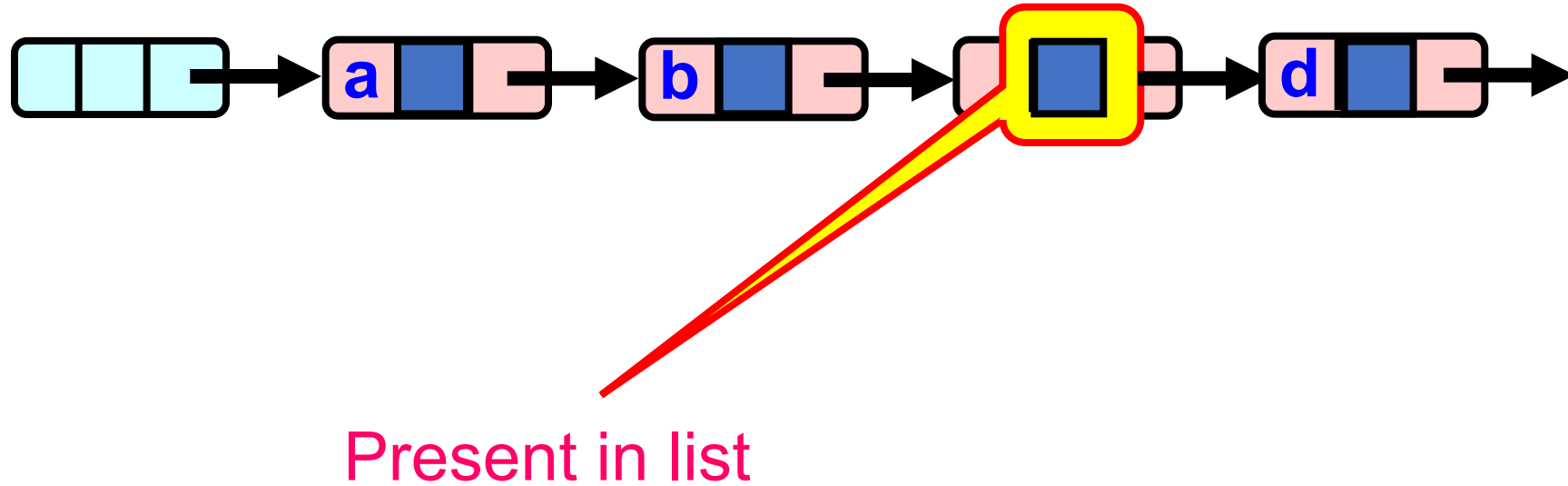
# Two step removal List

- **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)

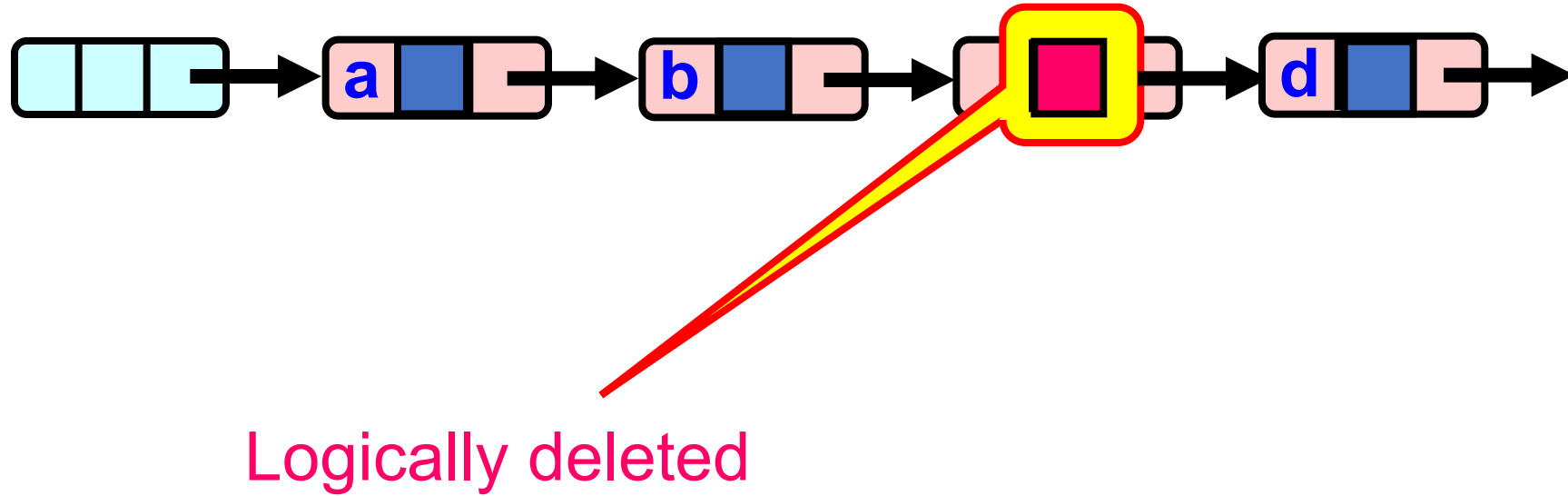
# Two step removal Removal



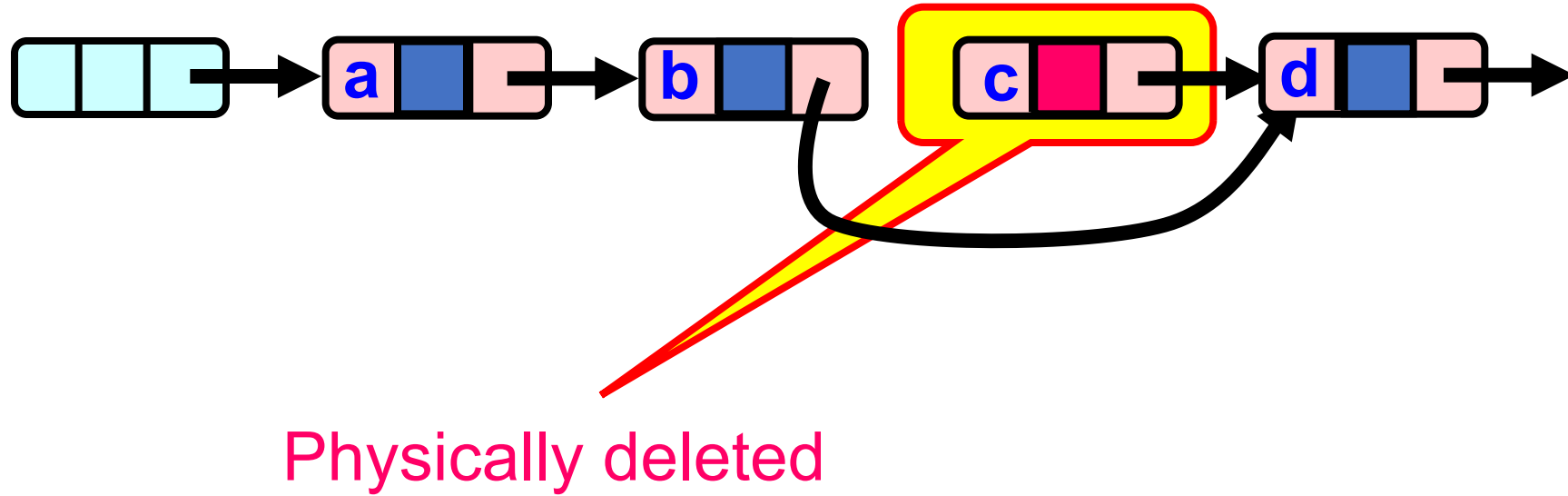
# Two step removal Removal



# Two step removal Removal

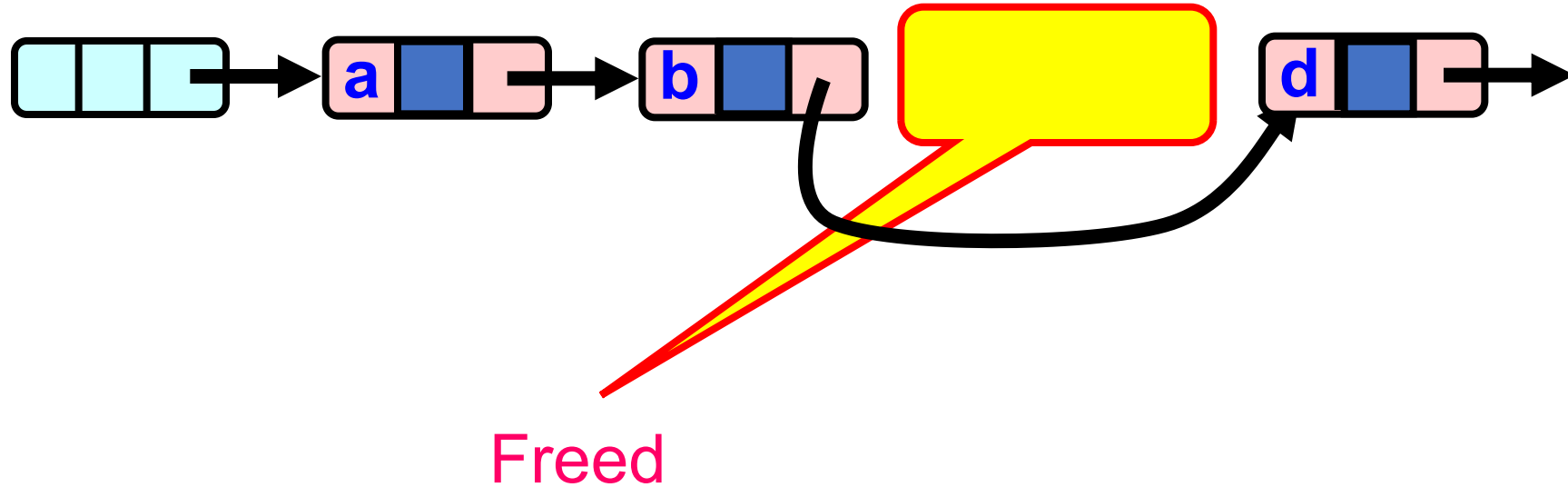


# Two step removal Removal





# Two step removal Removal



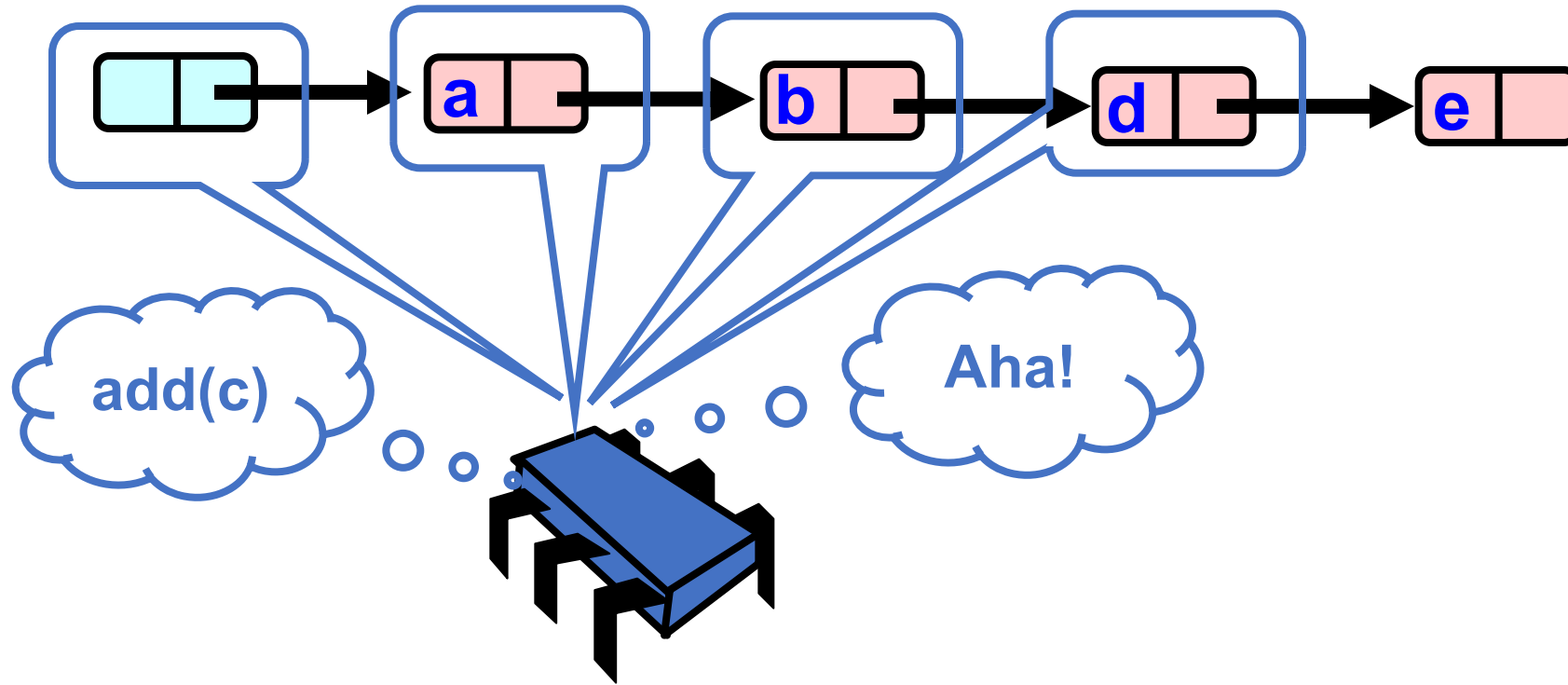
# Two step remove list

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

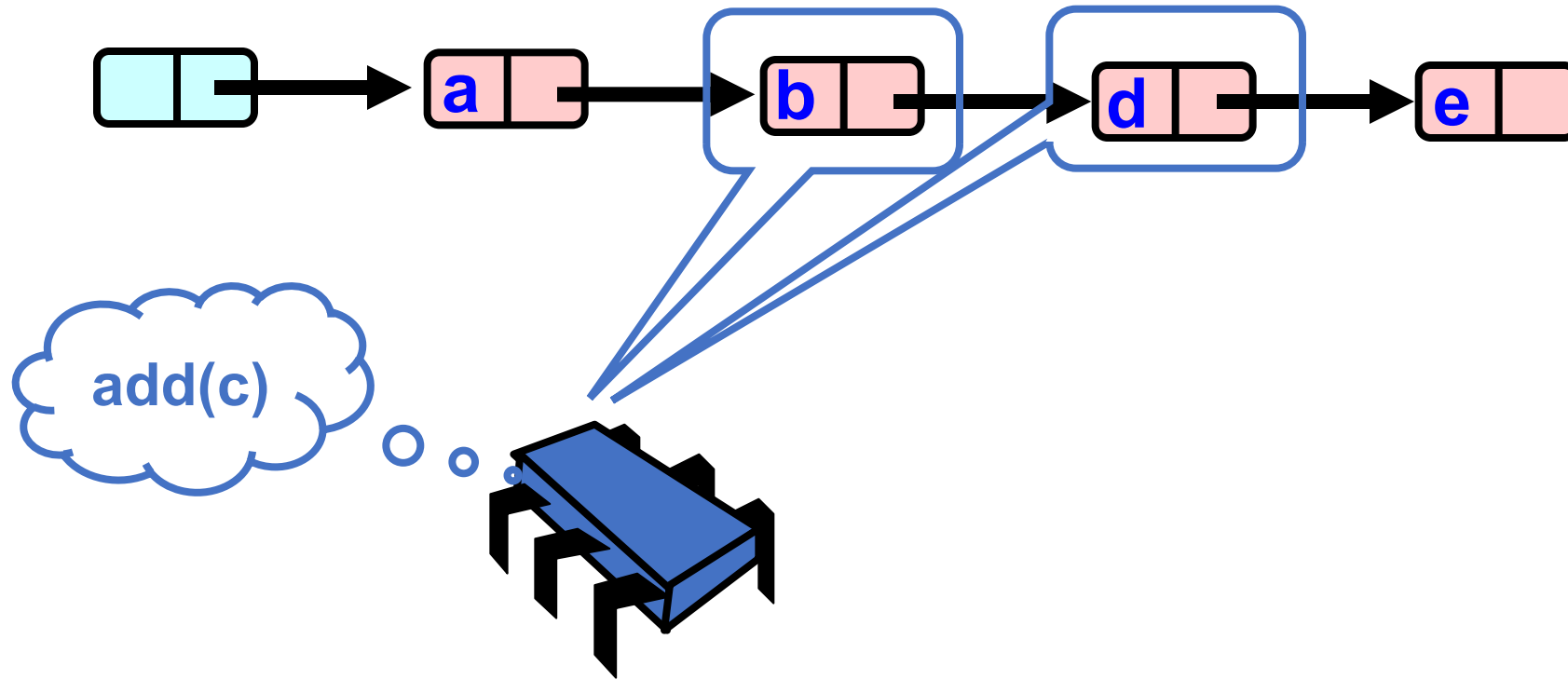
# Validation

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

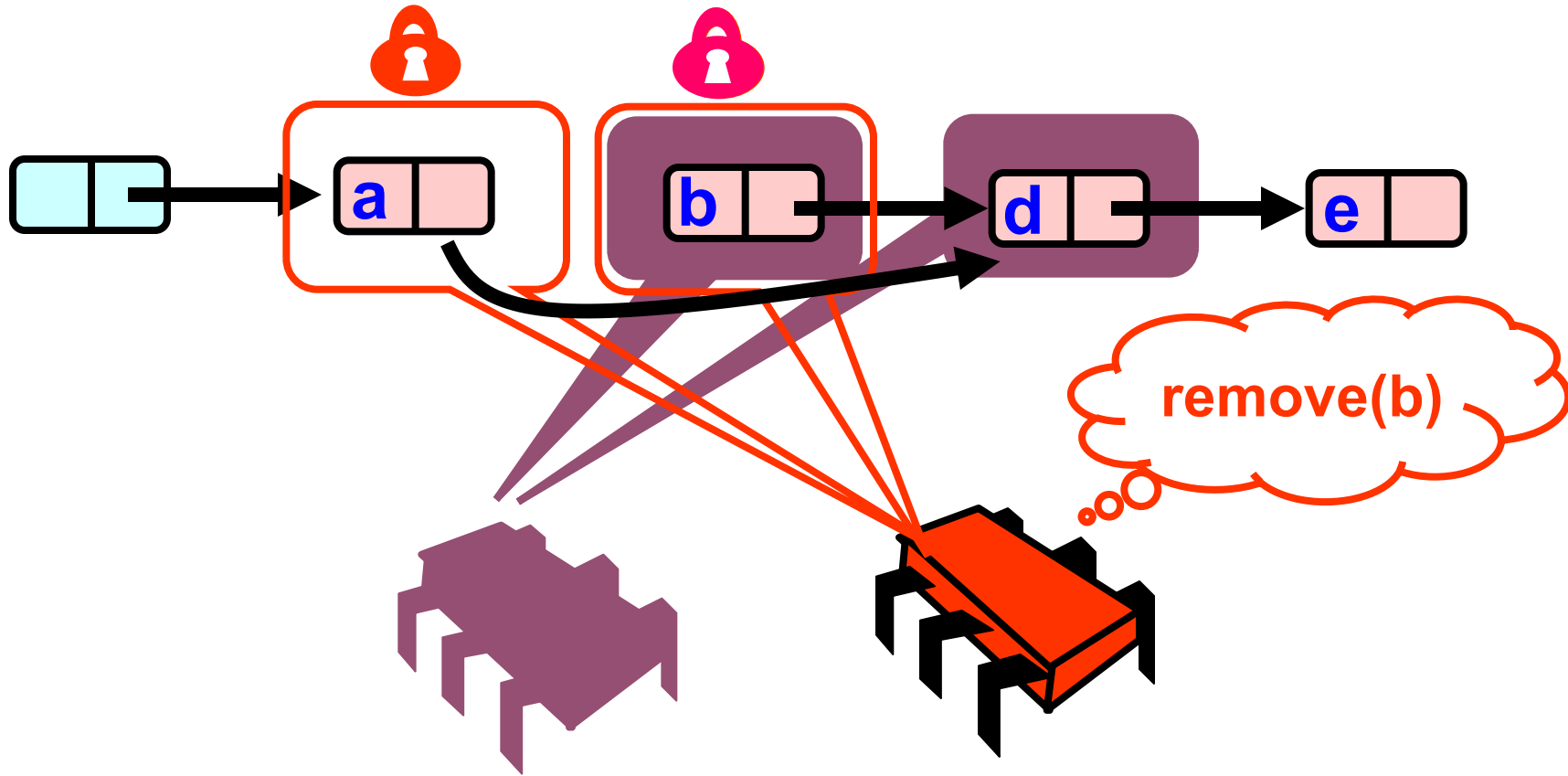
# What could go wrong?



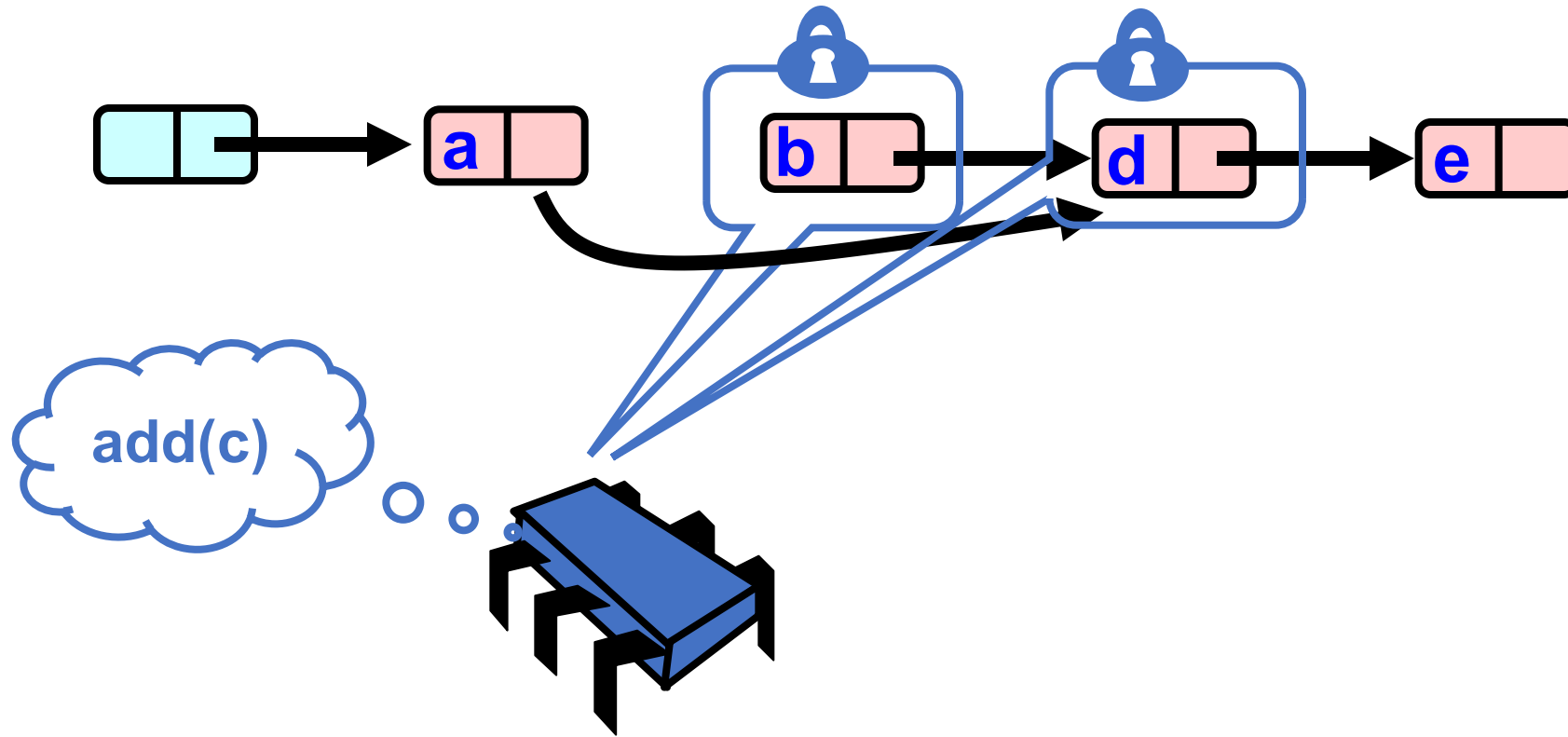
# What could go wrong?



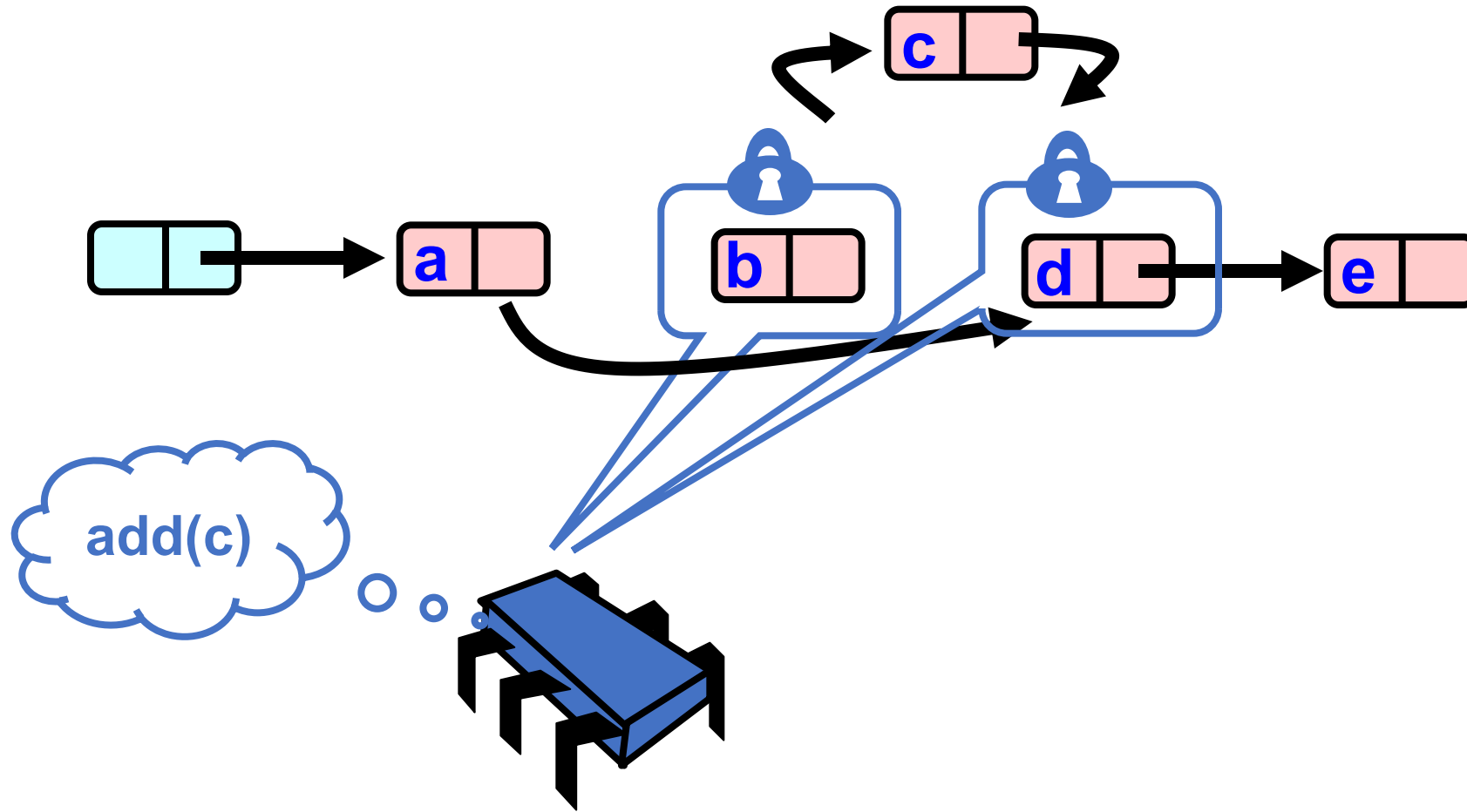
What could go wrong?



What could go wrong?

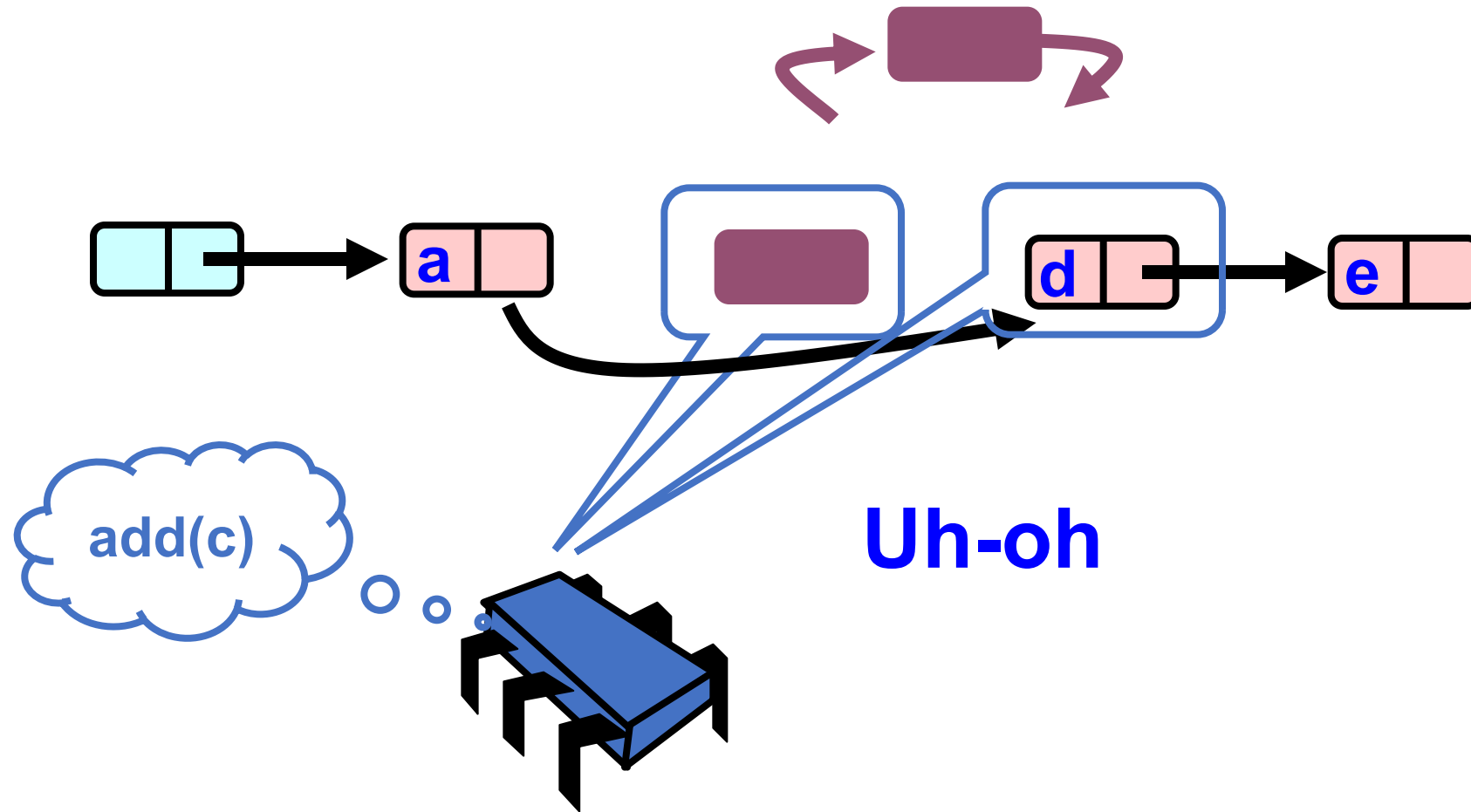


What could go wrong?

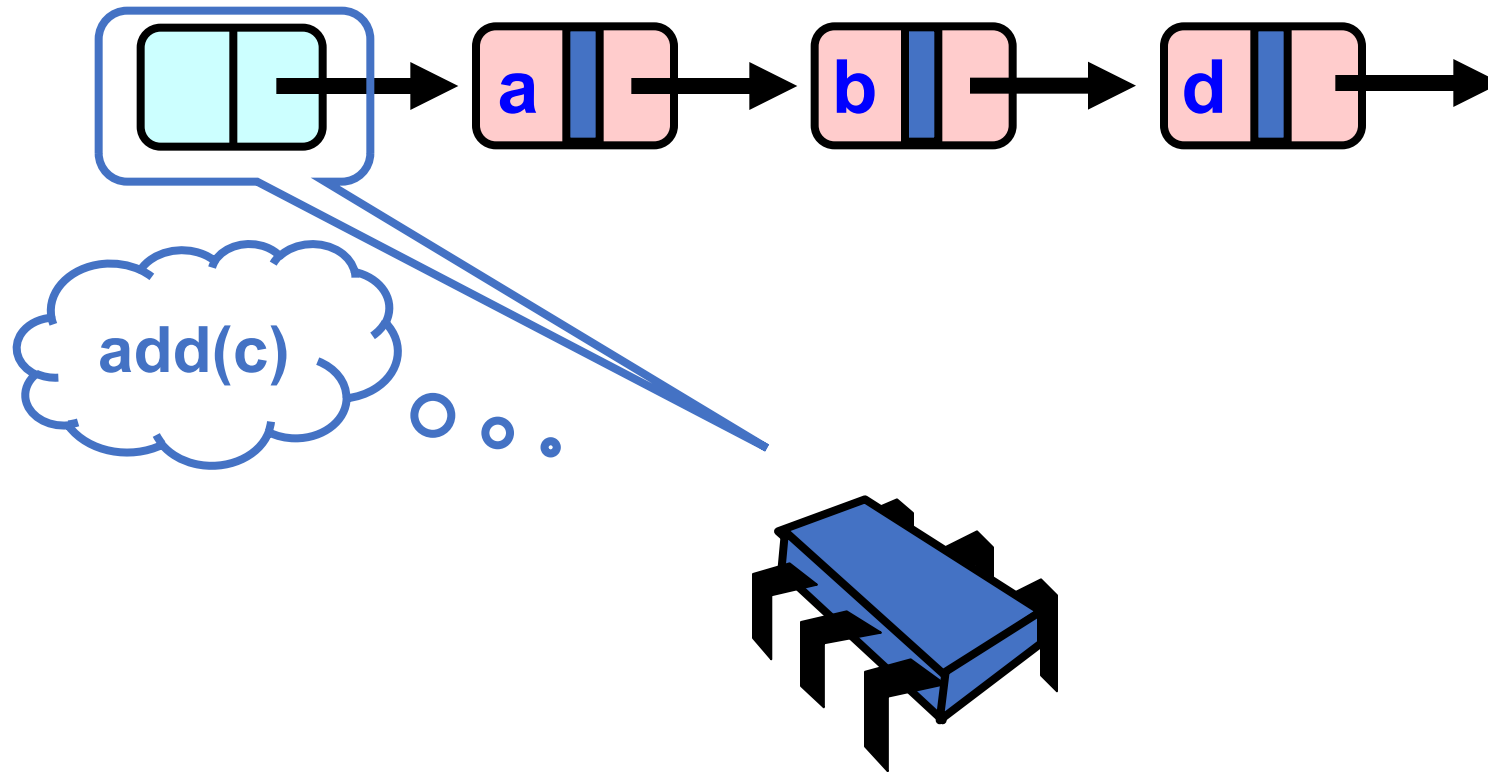




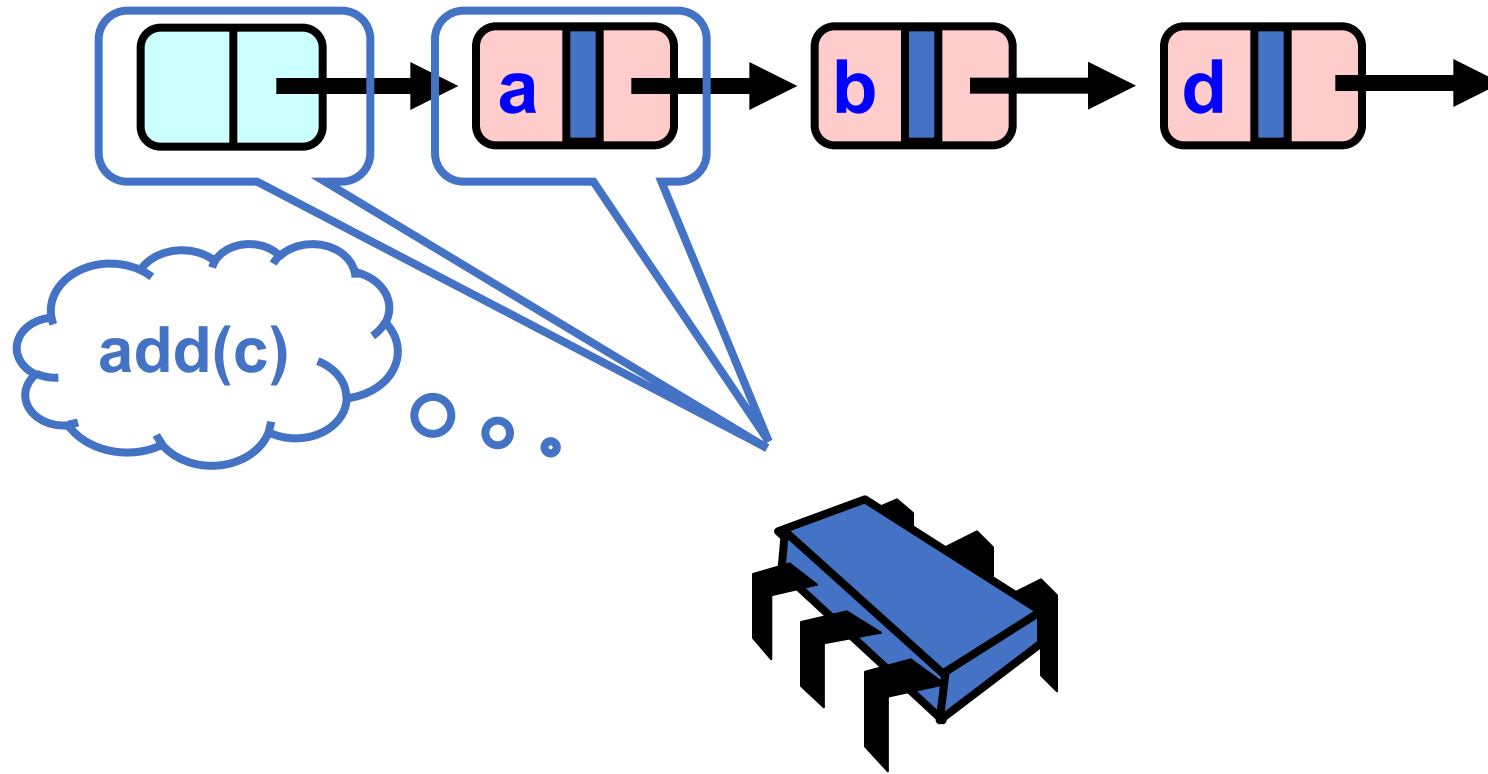
What could go wrong?



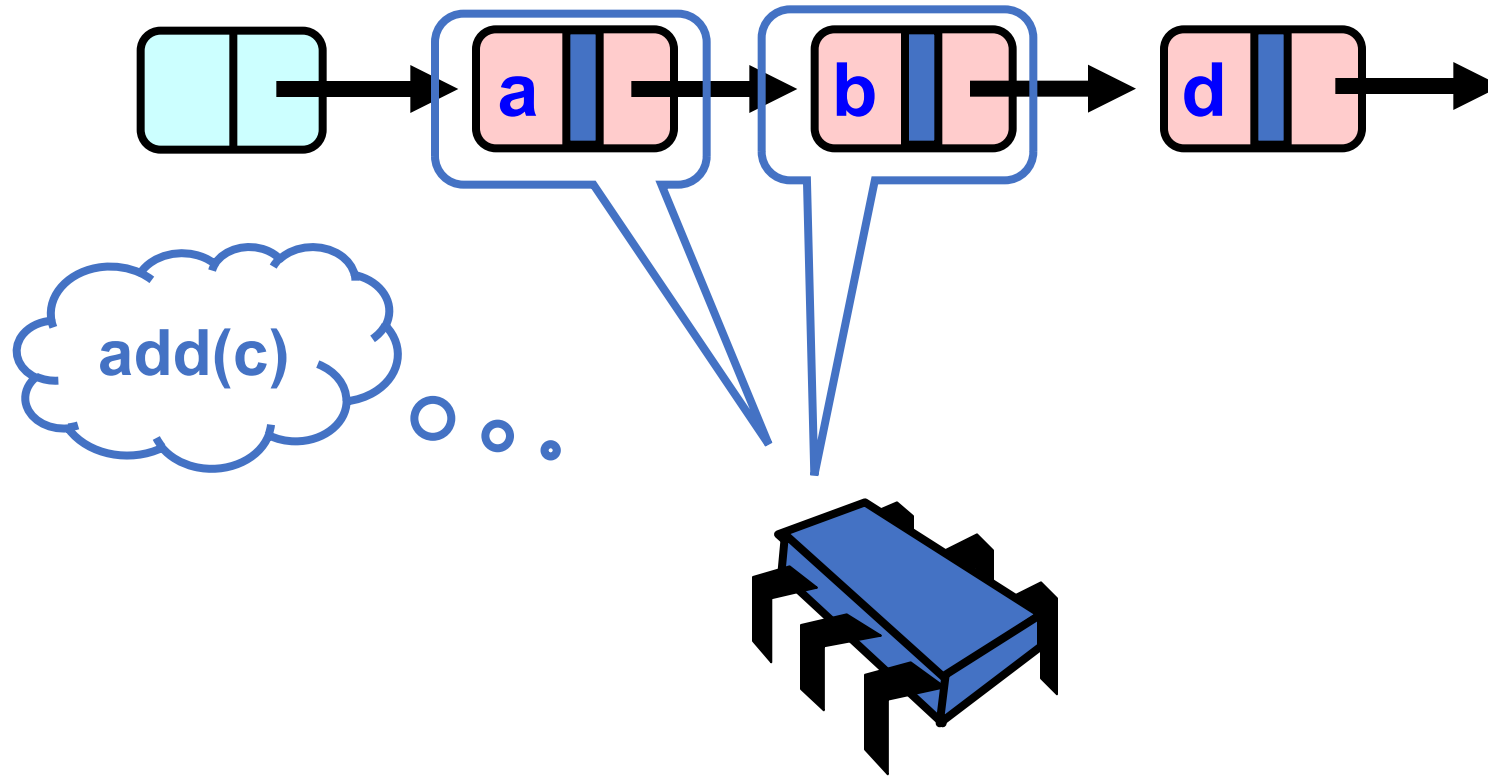
# Fixed with logical flag



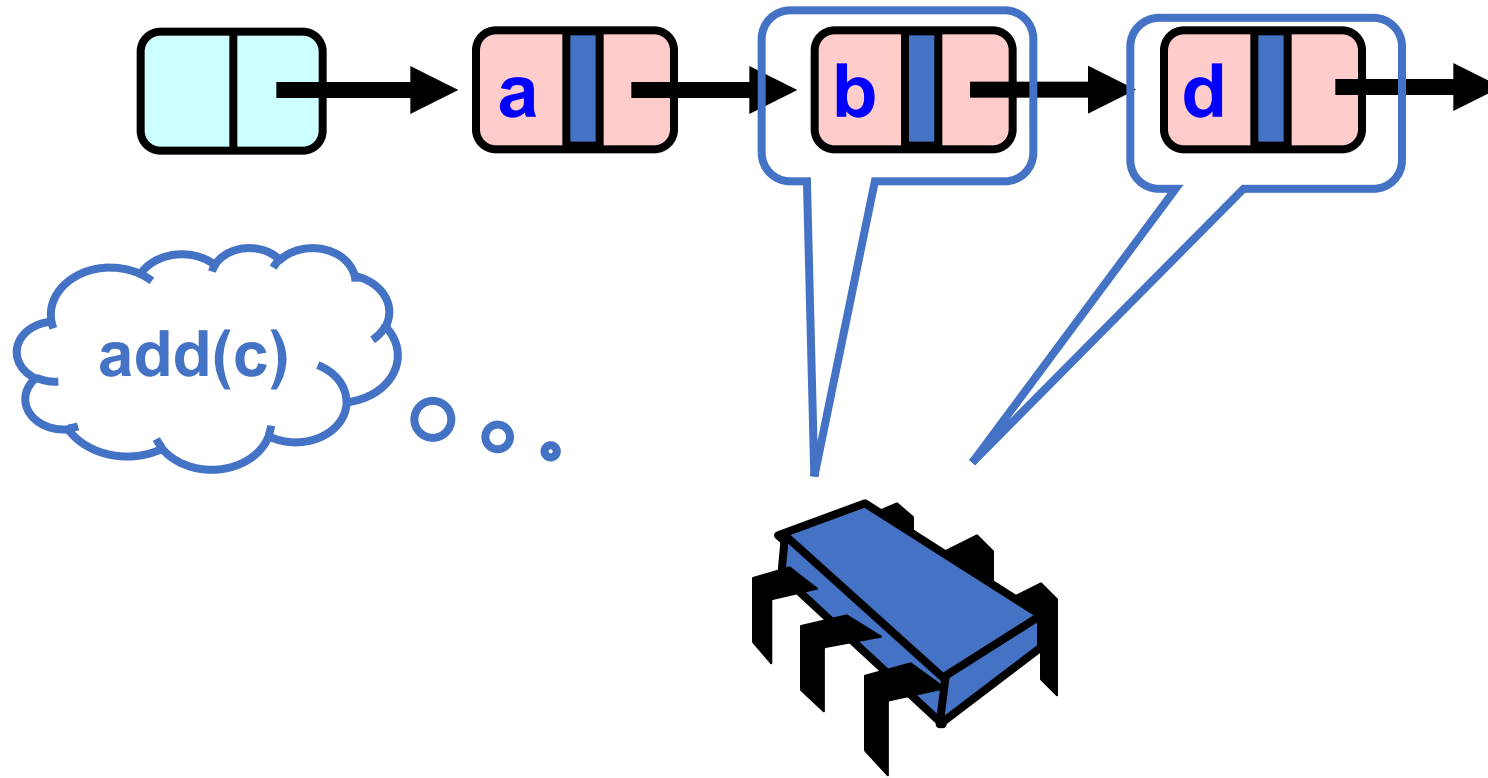
# Fixed with logical flag



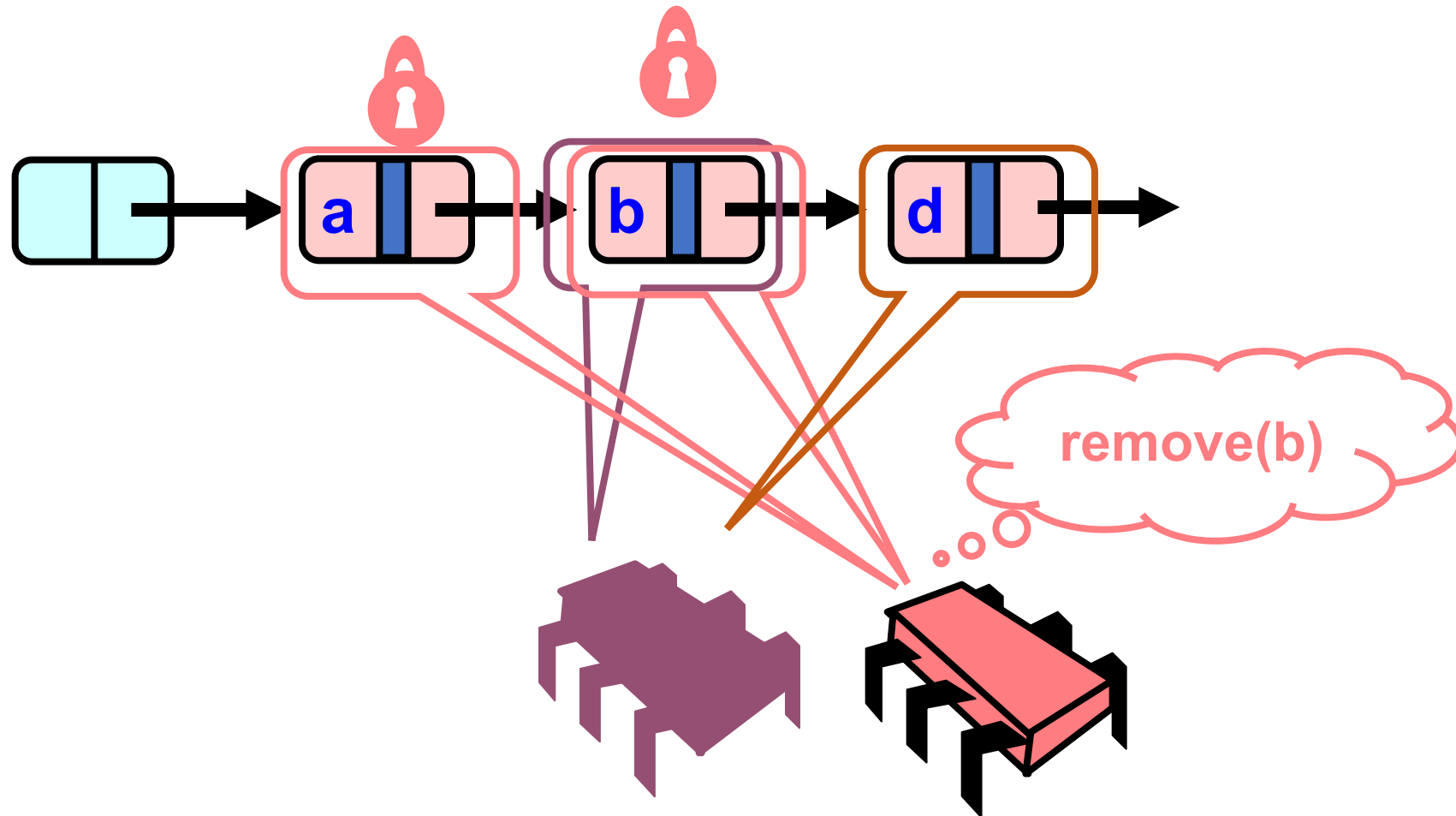
# Fixed with logical flag



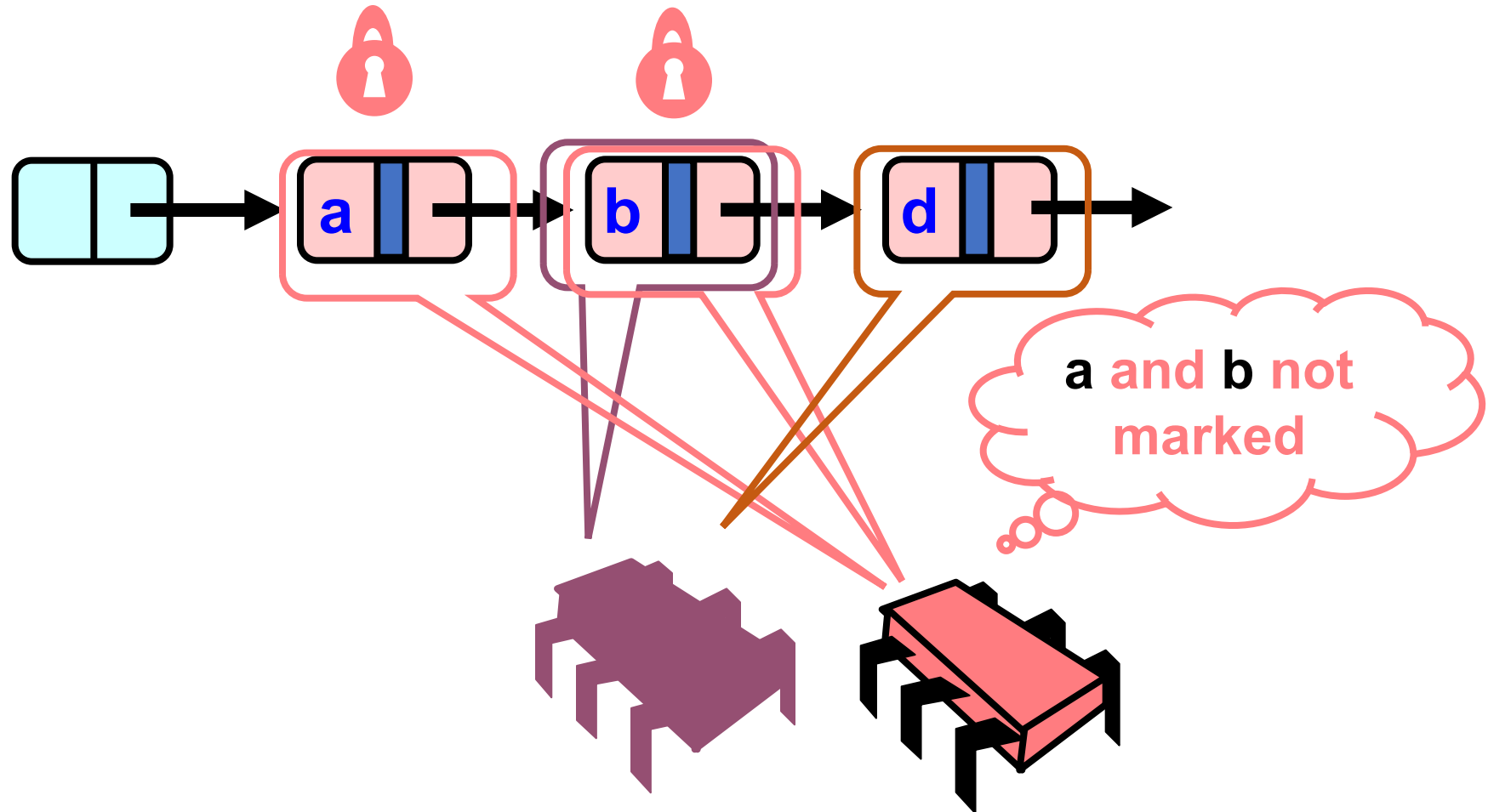
# Fixed with logical flag



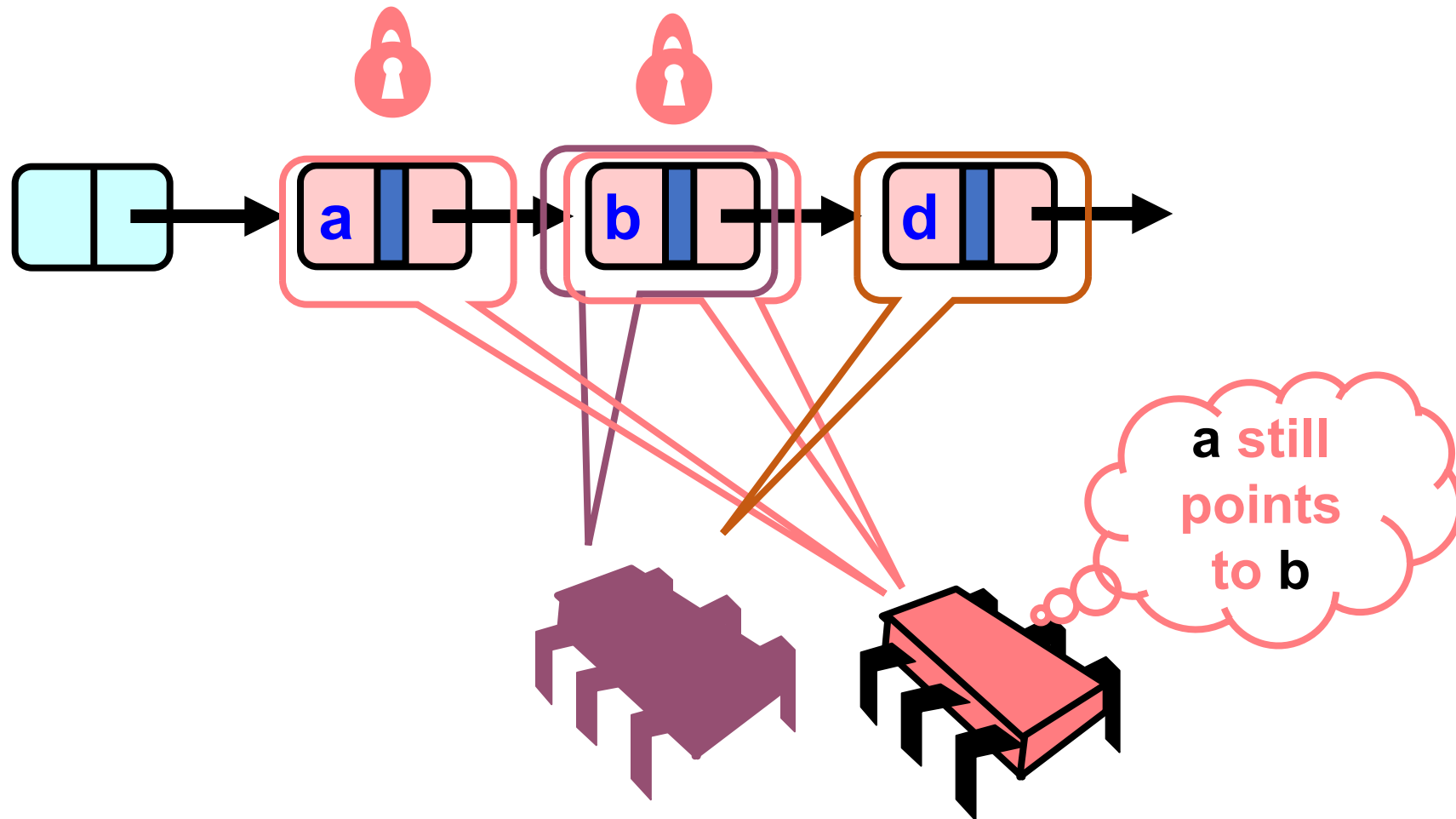
# Fixed with logical flag



# Fixed with logical flag

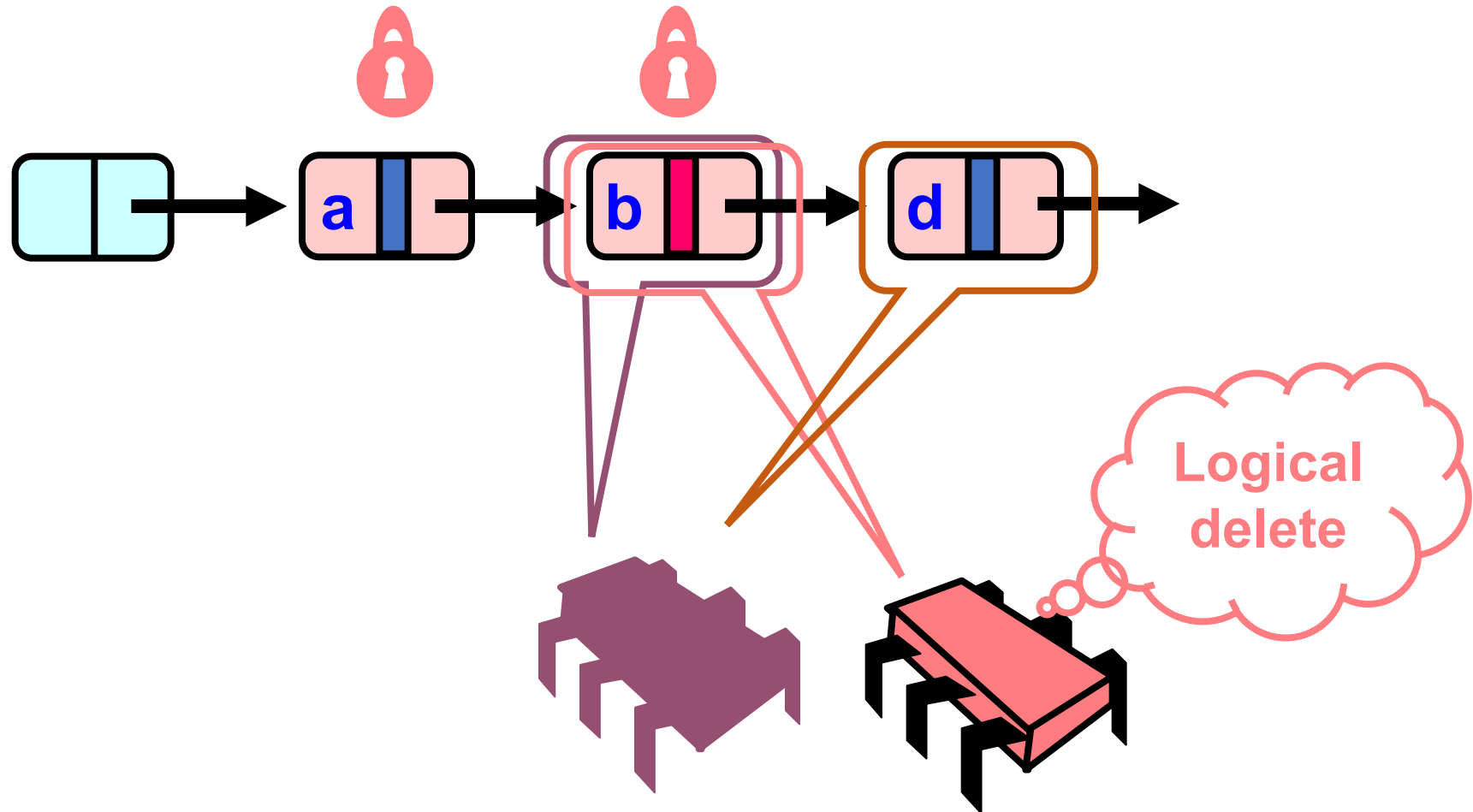


# Fixed with logical flag

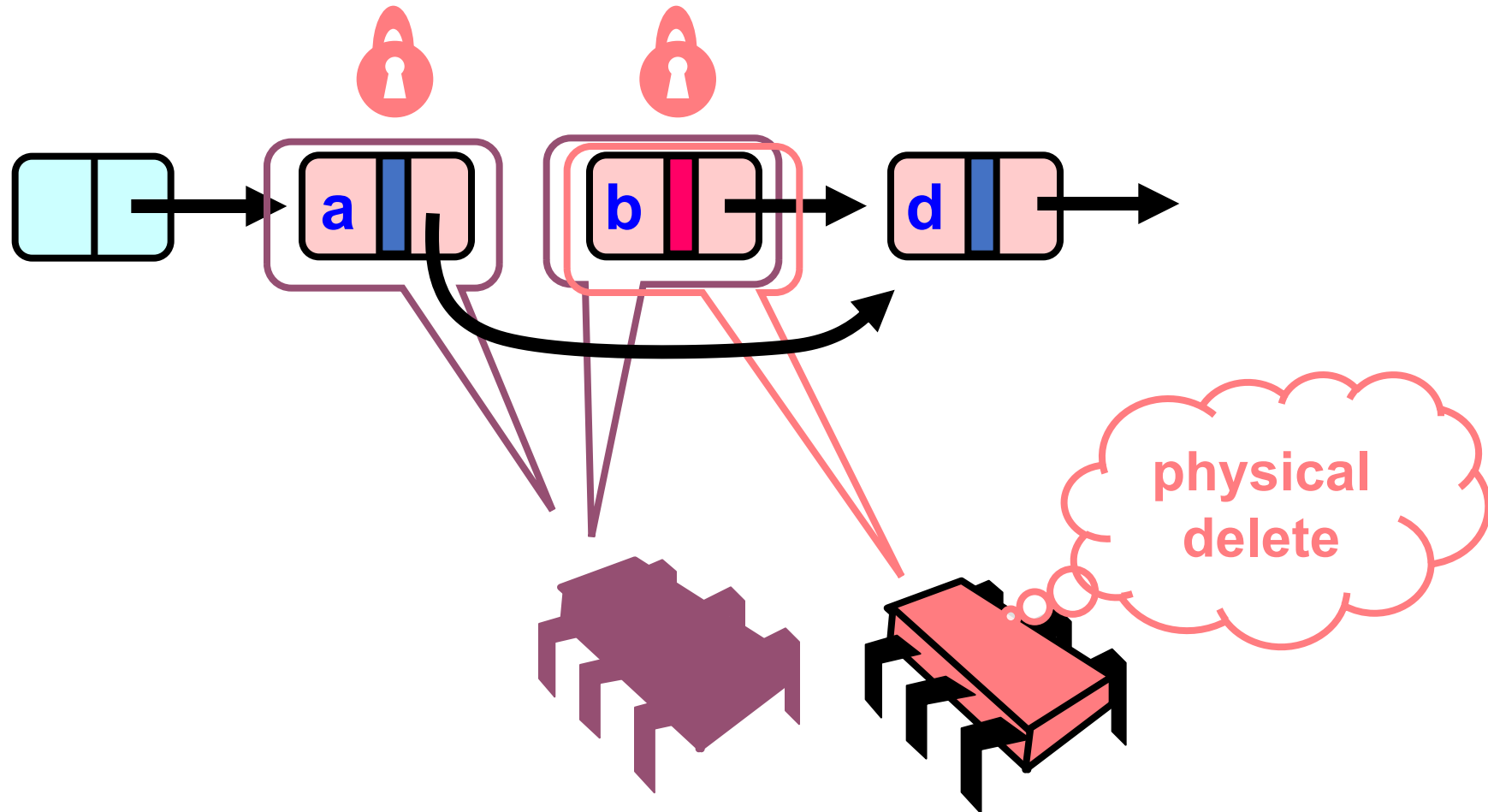




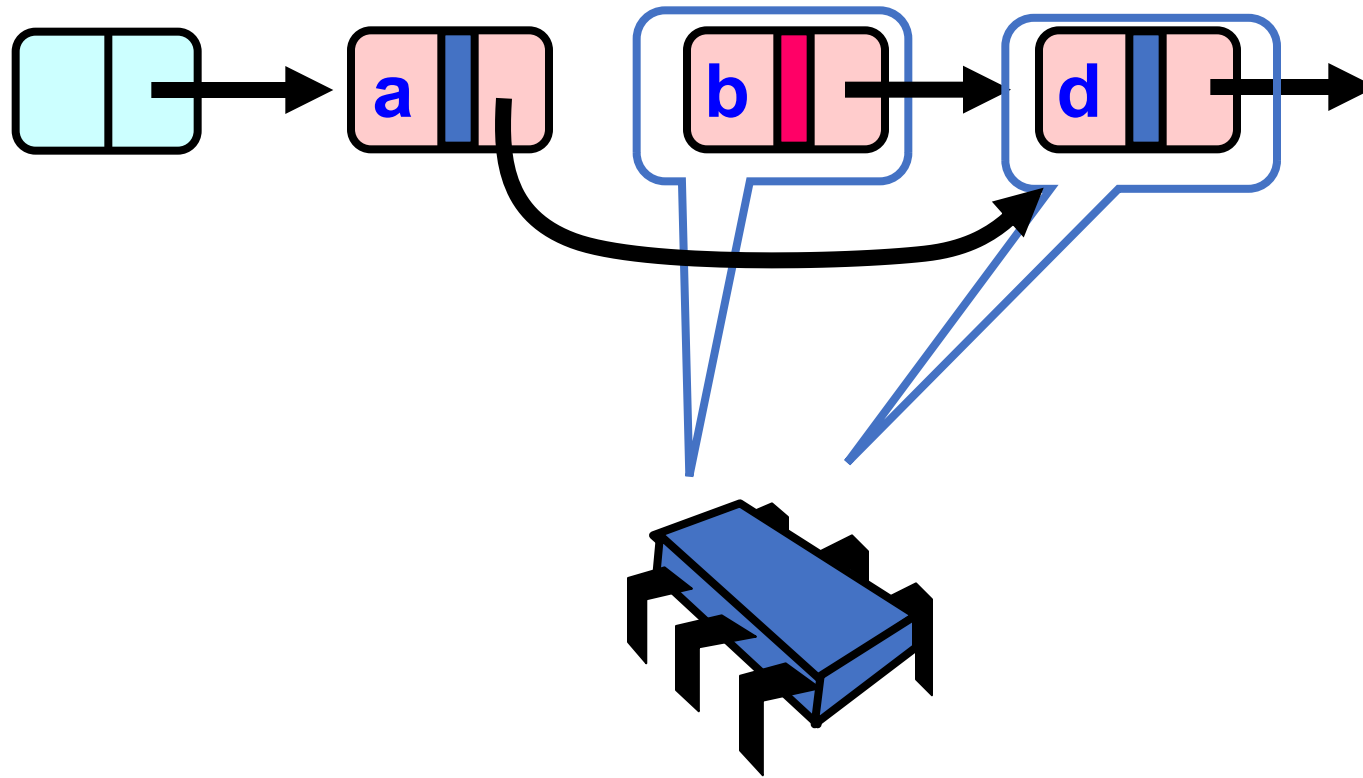
# Fixed with logical flag



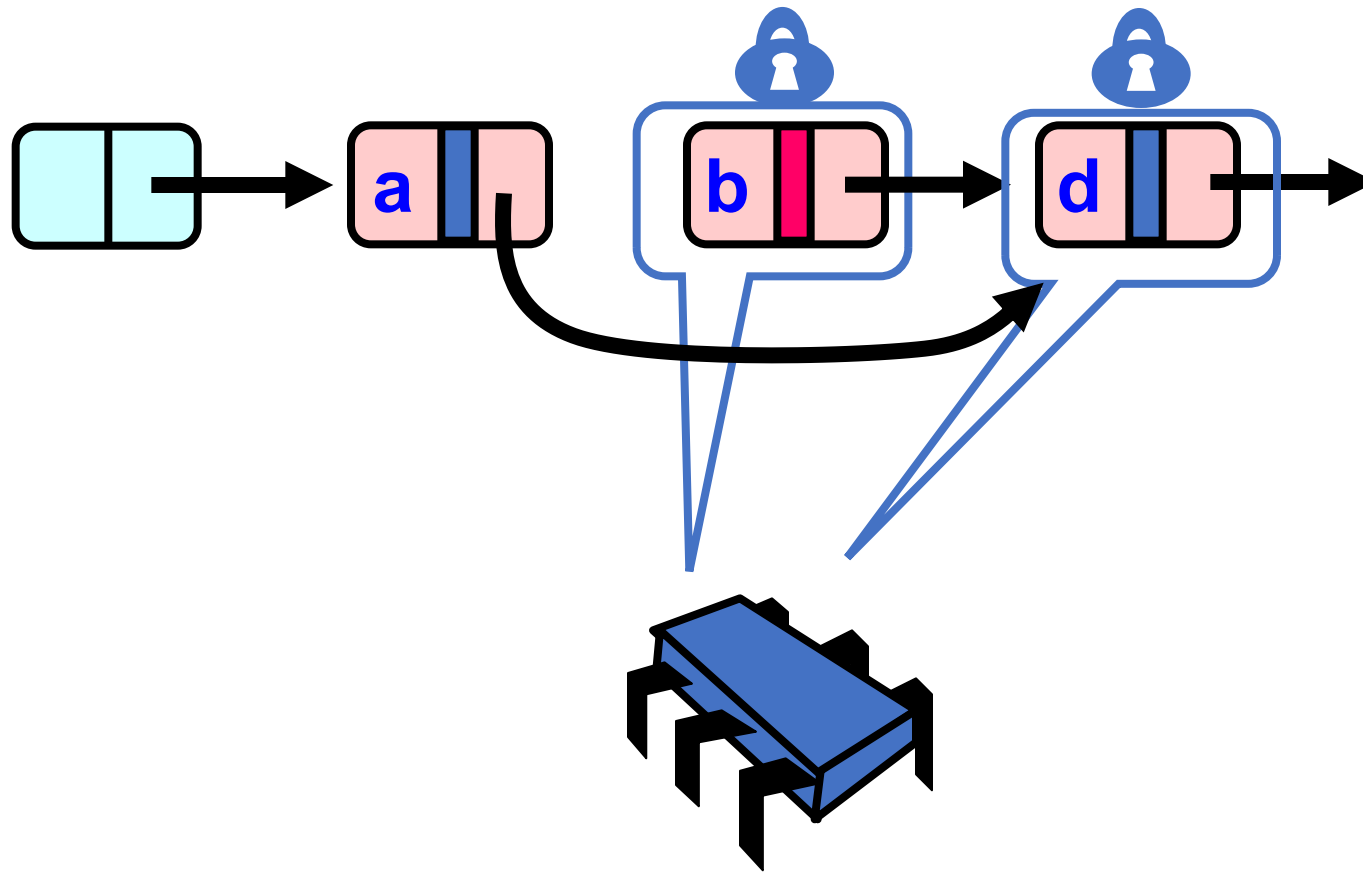
# Fixed with logical flag



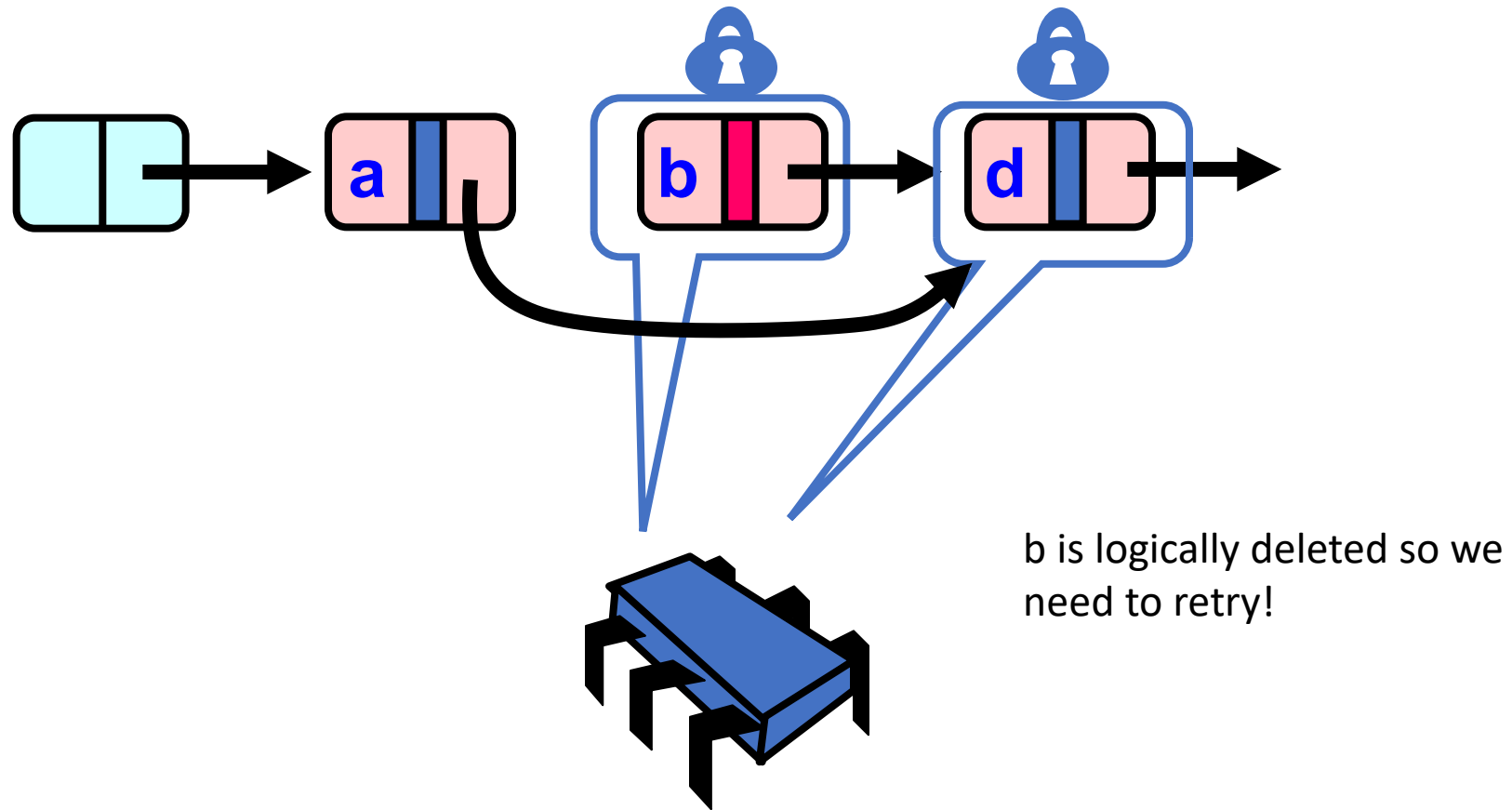
Fixed with logical flag



Fixed with logical flag



# Fixed with logical flag



# To complete the picture

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

# How good?

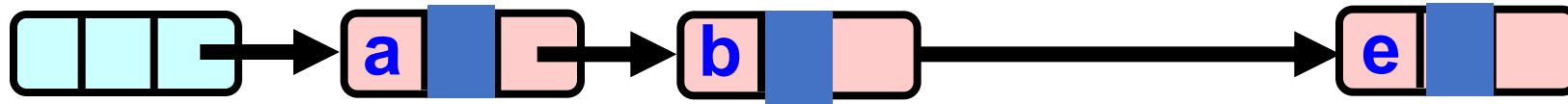
- Good:
  - Uncontended calls don't re-traverse
- Bad
  - `add()` and `remove()` use locks

# Lock-free Lists

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

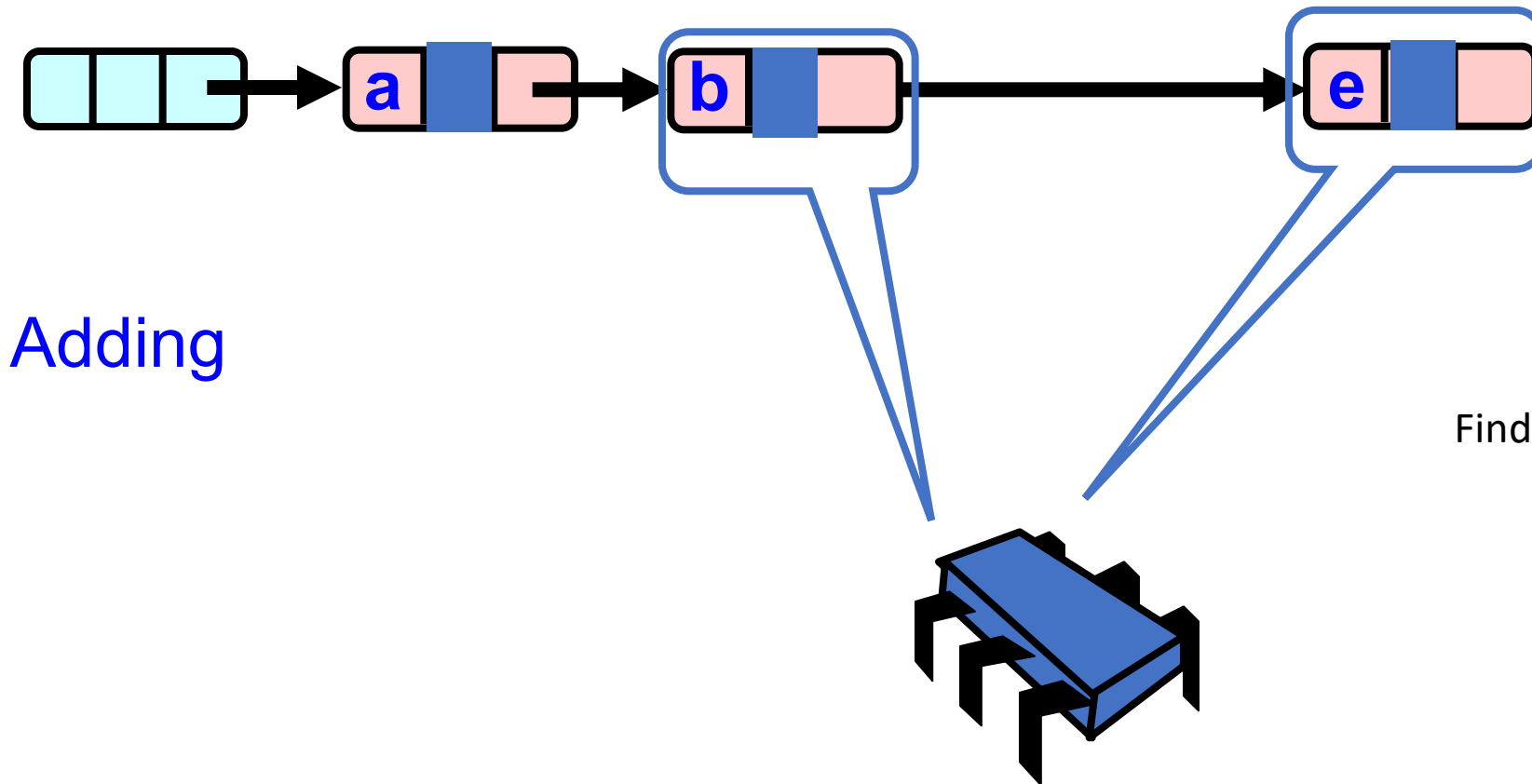


# Lock-free Lists

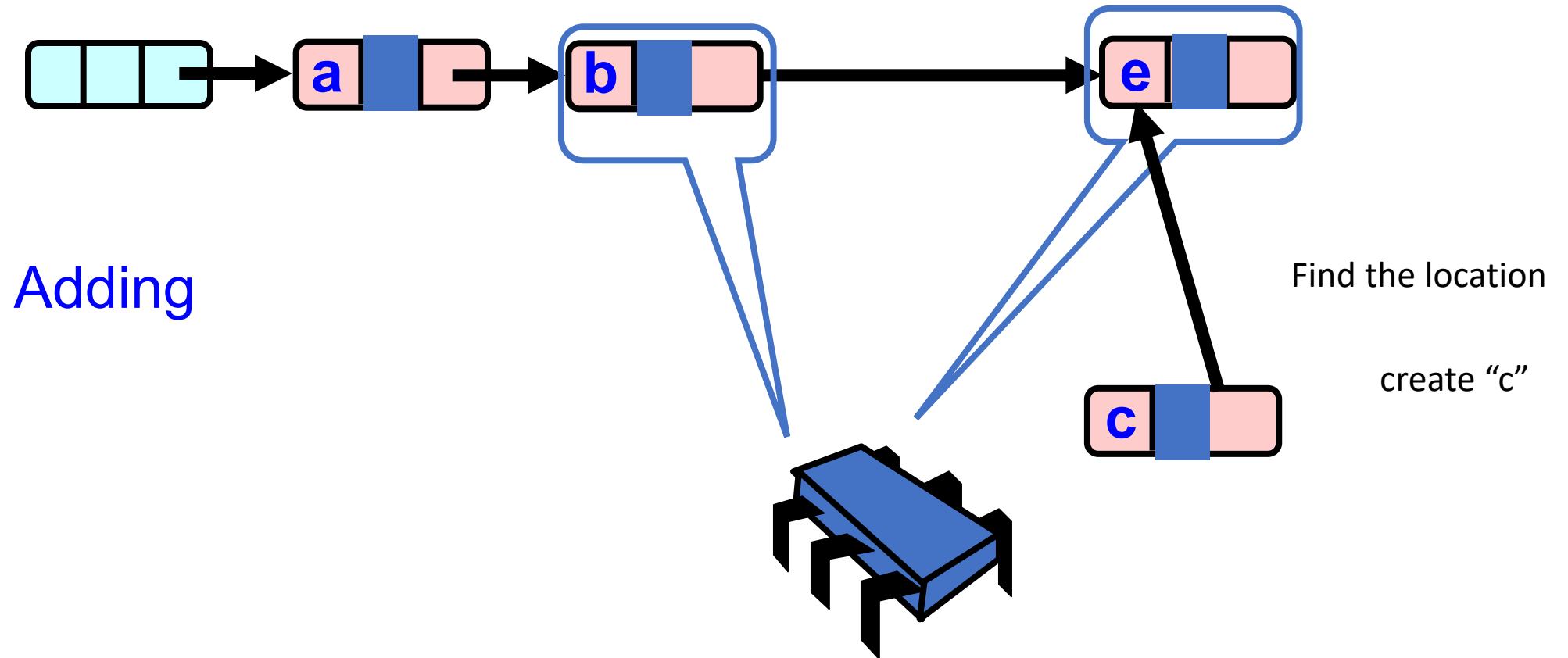


Adding

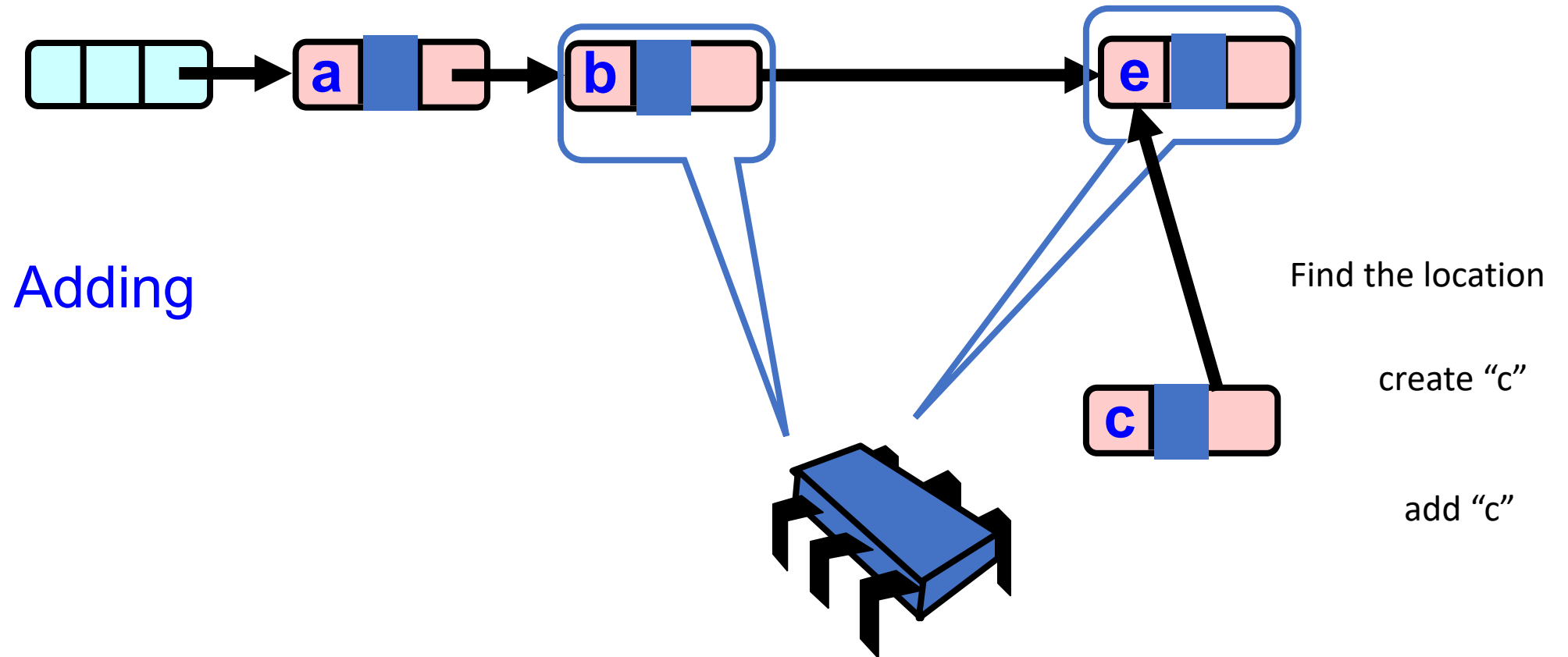
# Lock-free Lists



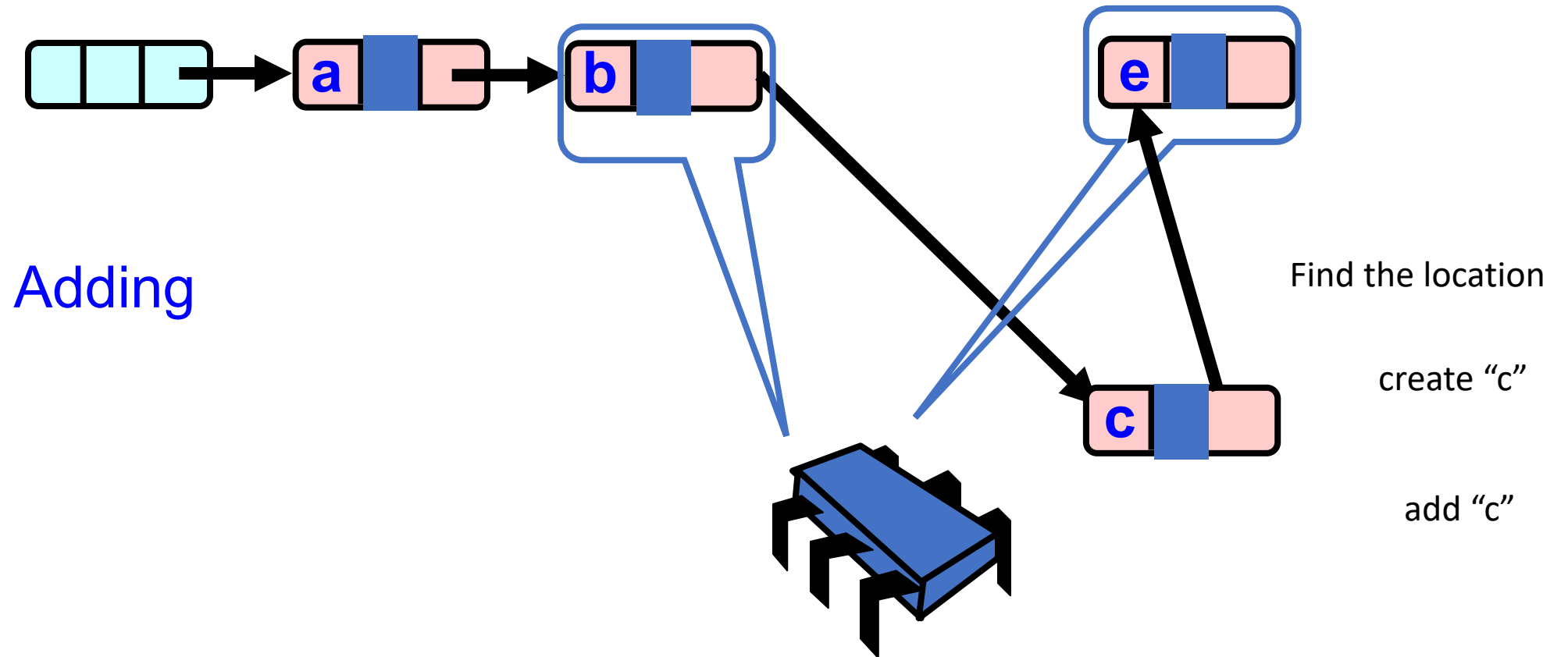
# Lock-free Lists



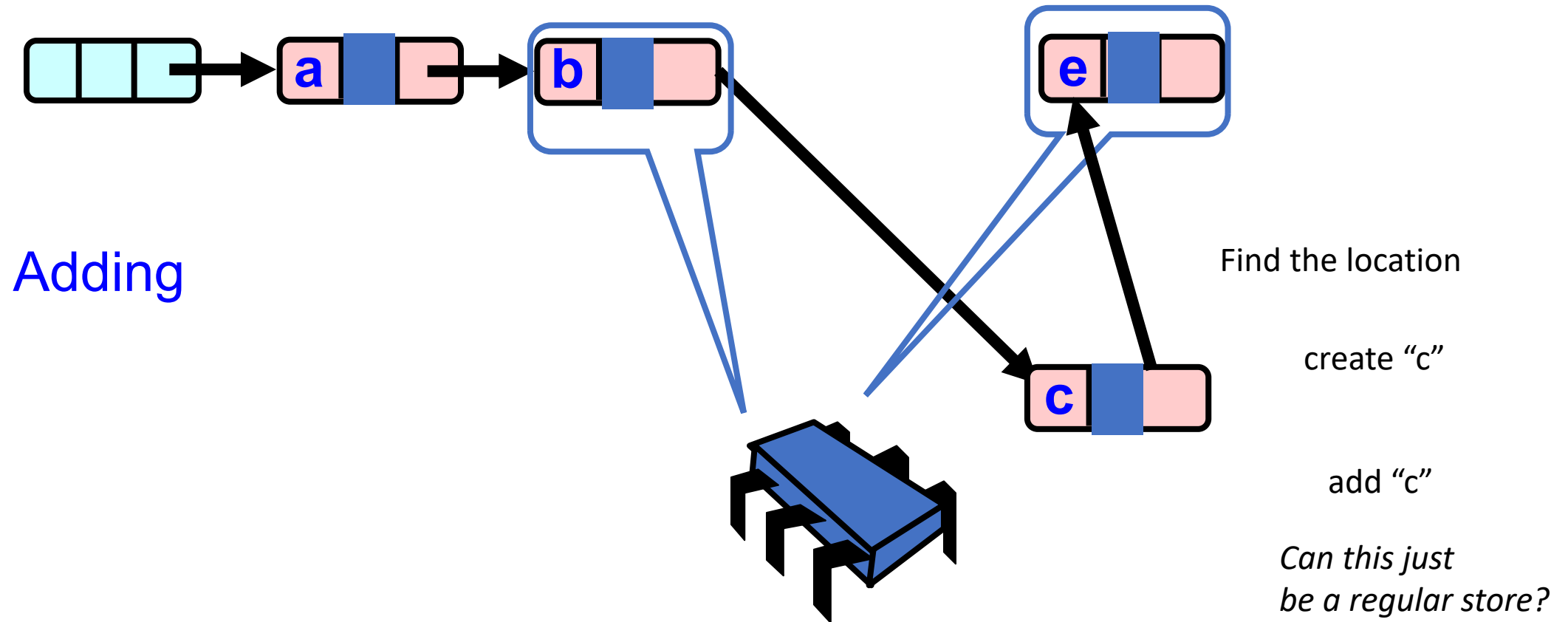
# Lock-free Lists



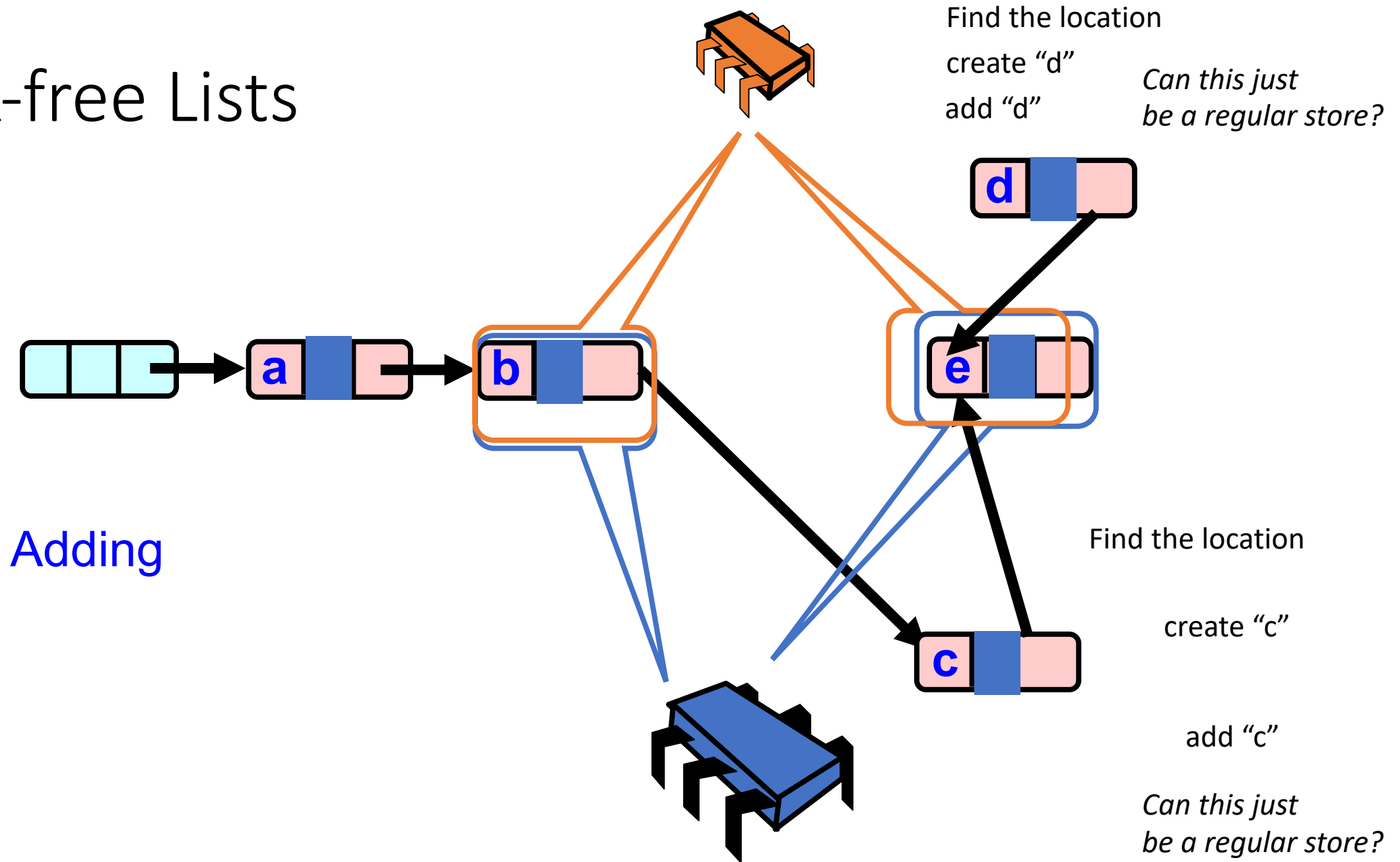
# Lock-free Lists



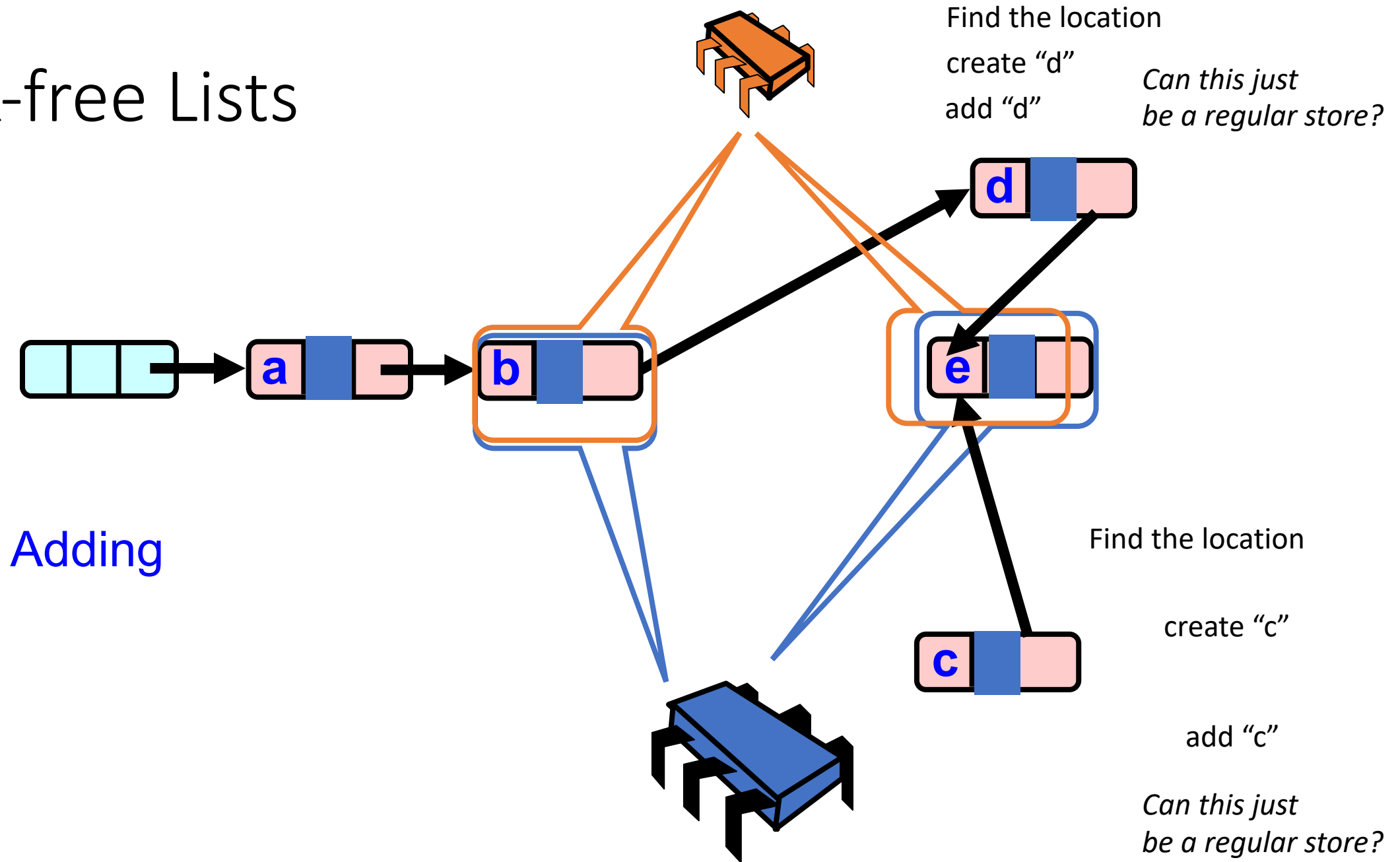
# Lock-free Lists



# Lock-free Lists

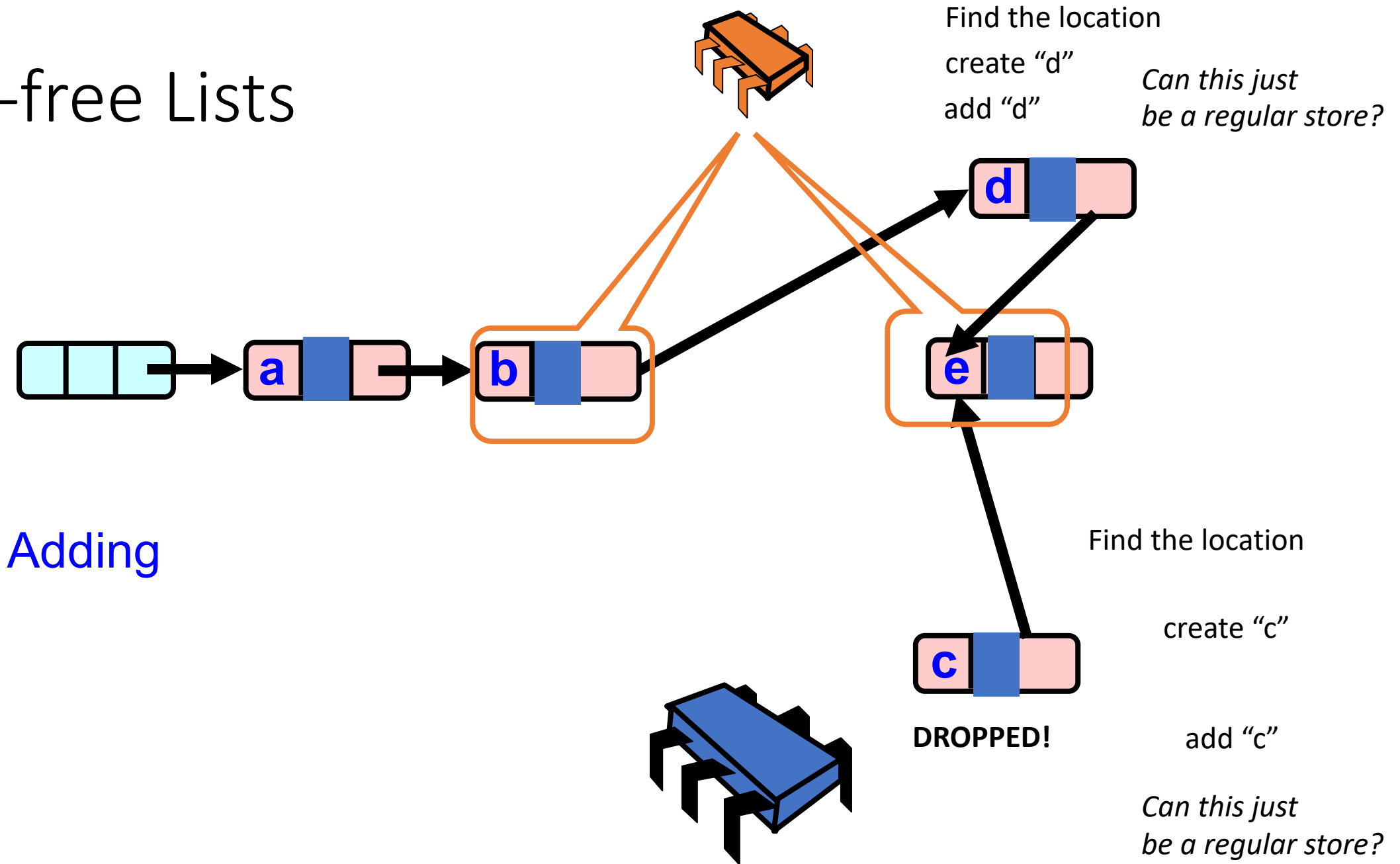


# Lock-free Lists

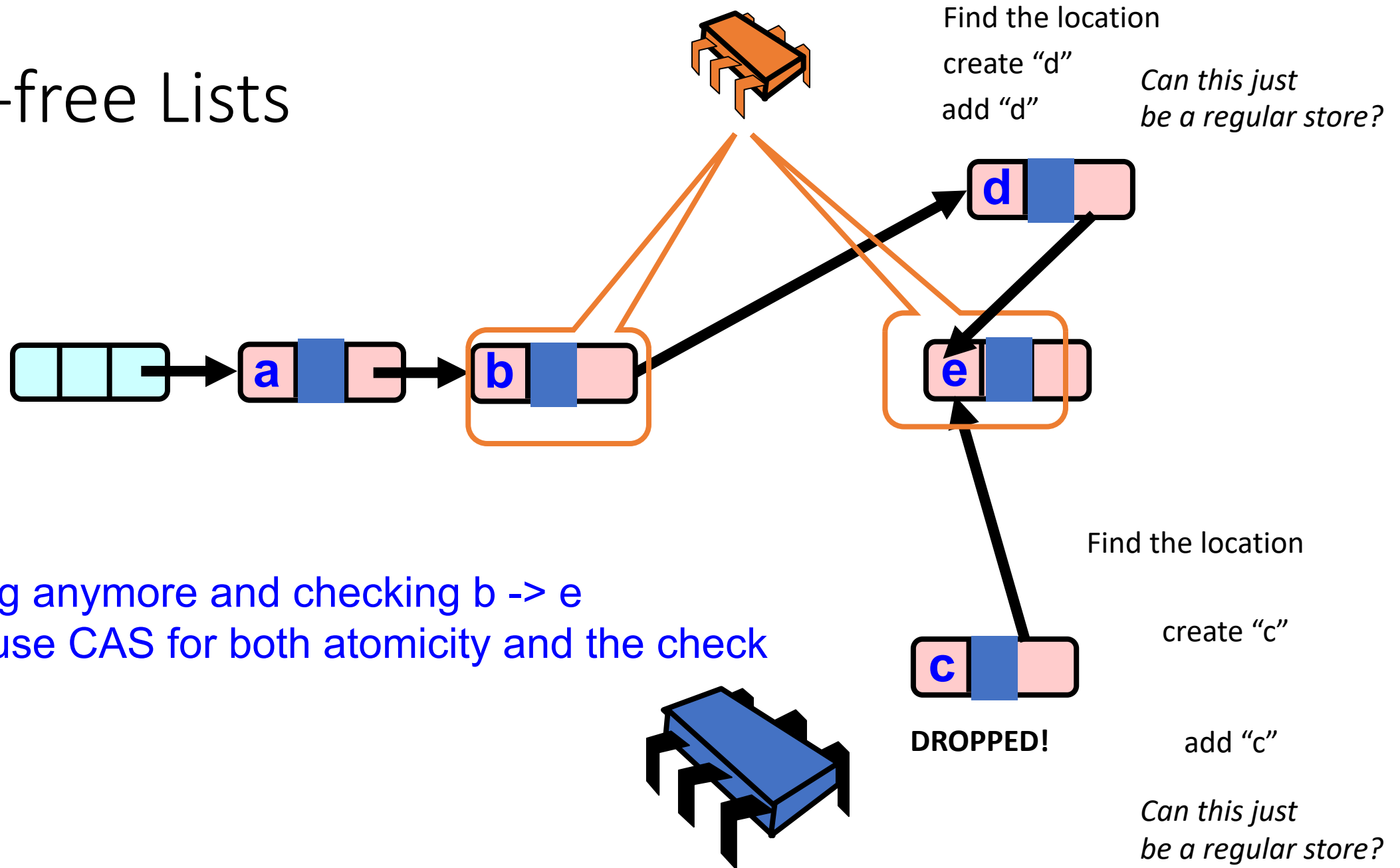




# Lock-free Lists



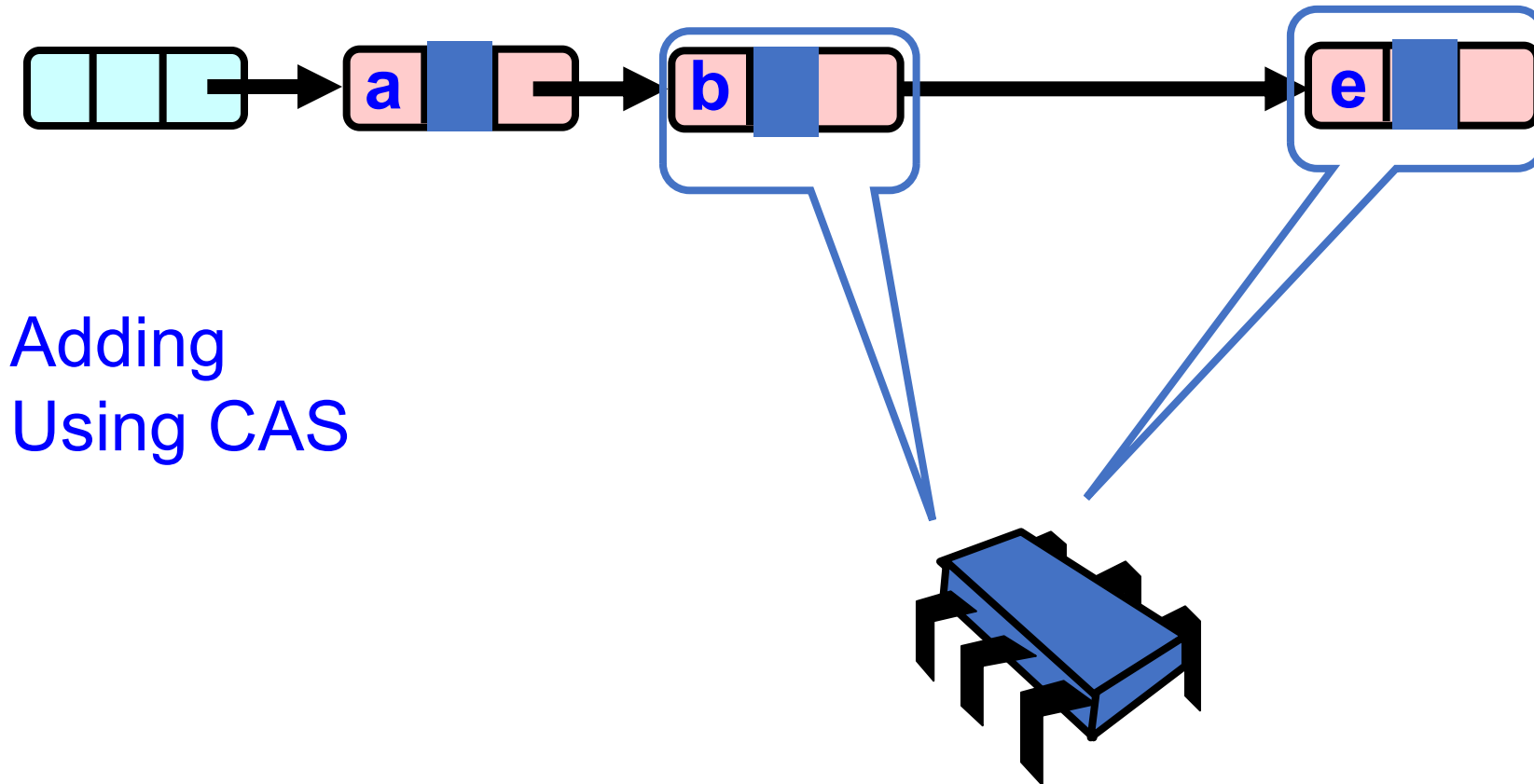
# Lock-free Lists



# Lock-free Lists

Find the location  
Cache your insertion  
point!

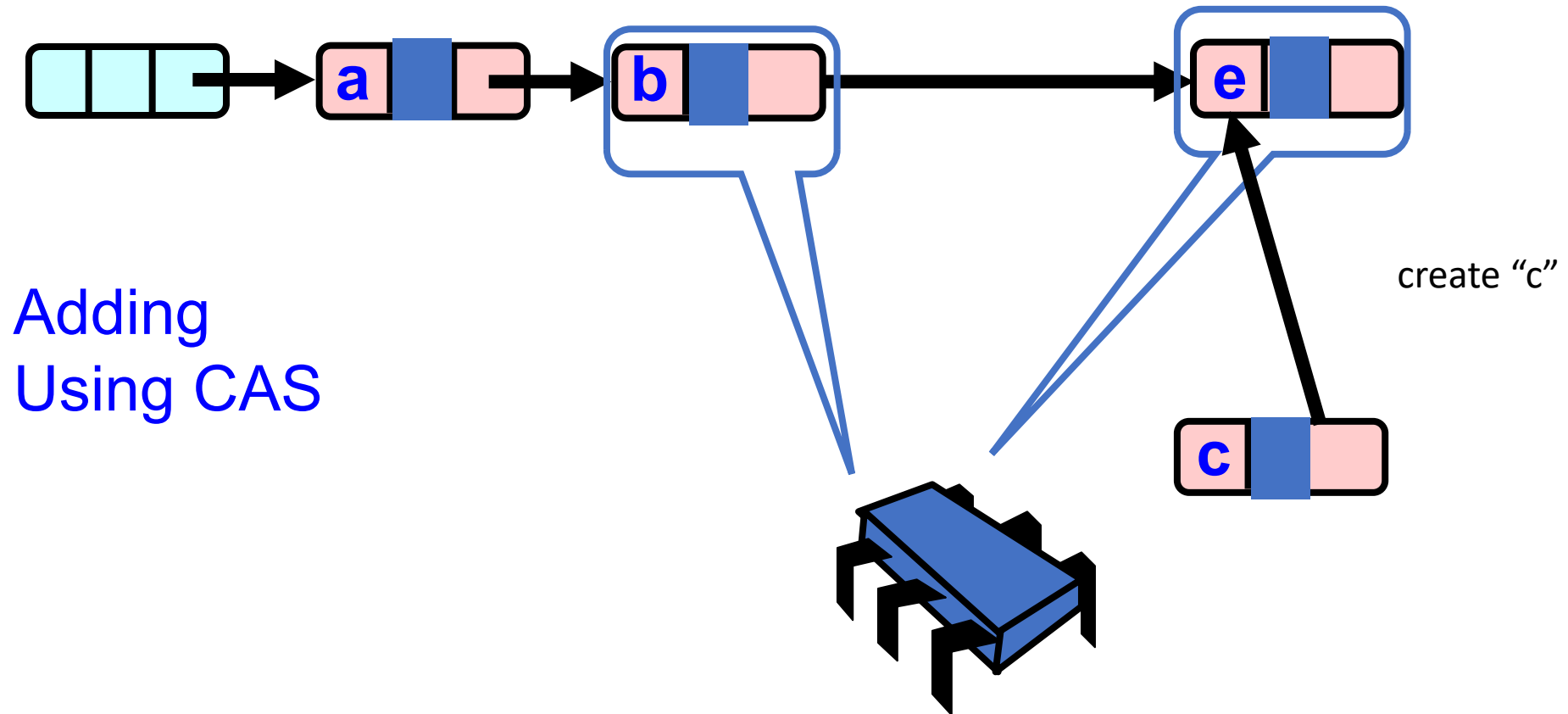
`b.next == e`



# Lock-free Lists

Find the location  
Cache your insertion  
point!

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



# Lock-free Lists

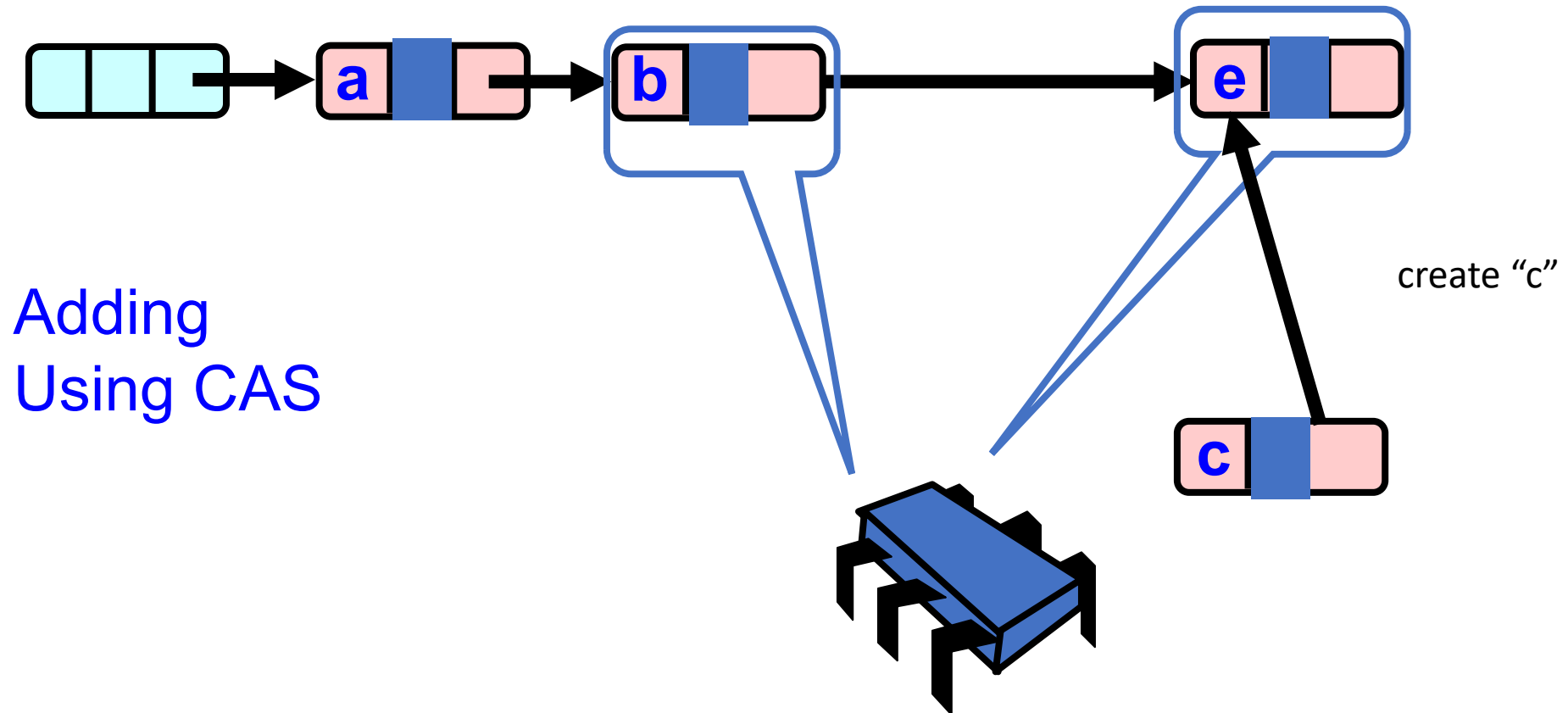
Only add if your insertion point is valid!

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

Find the location  
Cache your insertion point!

`b.next == e`

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



# Lock-free Lists

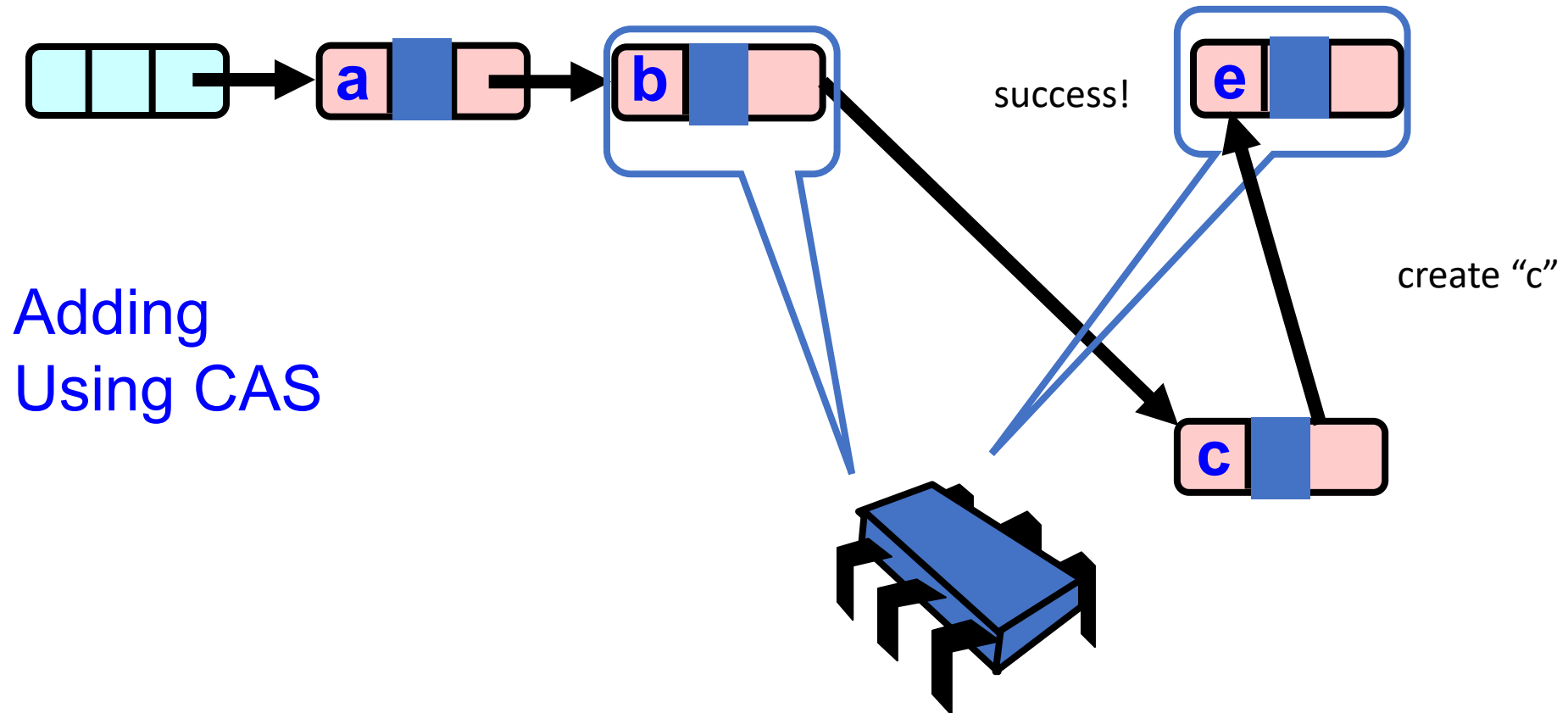
Only add if your insertion point is valid!

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

Find the location  
Cache your insertion point!

`b.next == e`

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



# Lock-free Lists

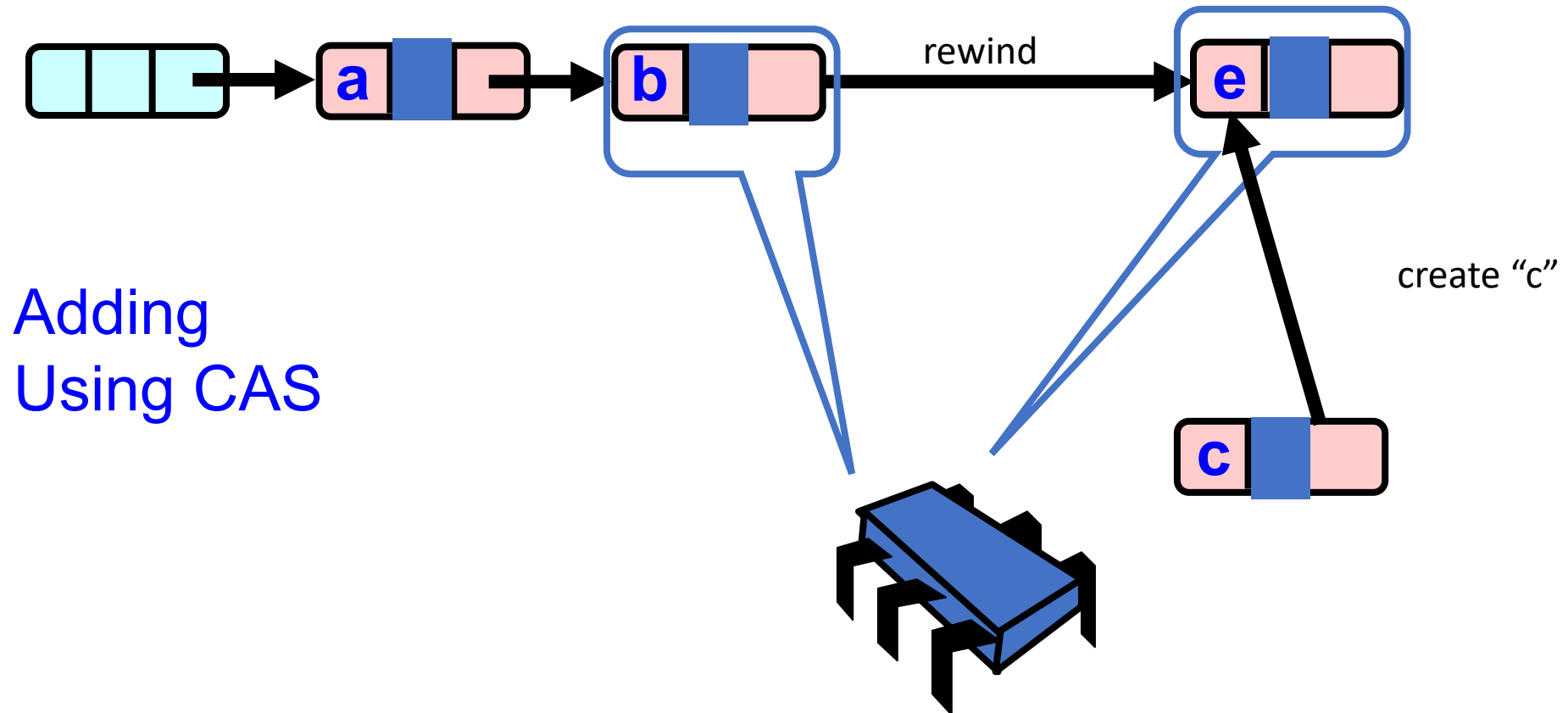
Only add if your insertion point is valid!

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

Find the location  
Cache your insertion point!

`b.next == e`

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



# Lock-free Lists

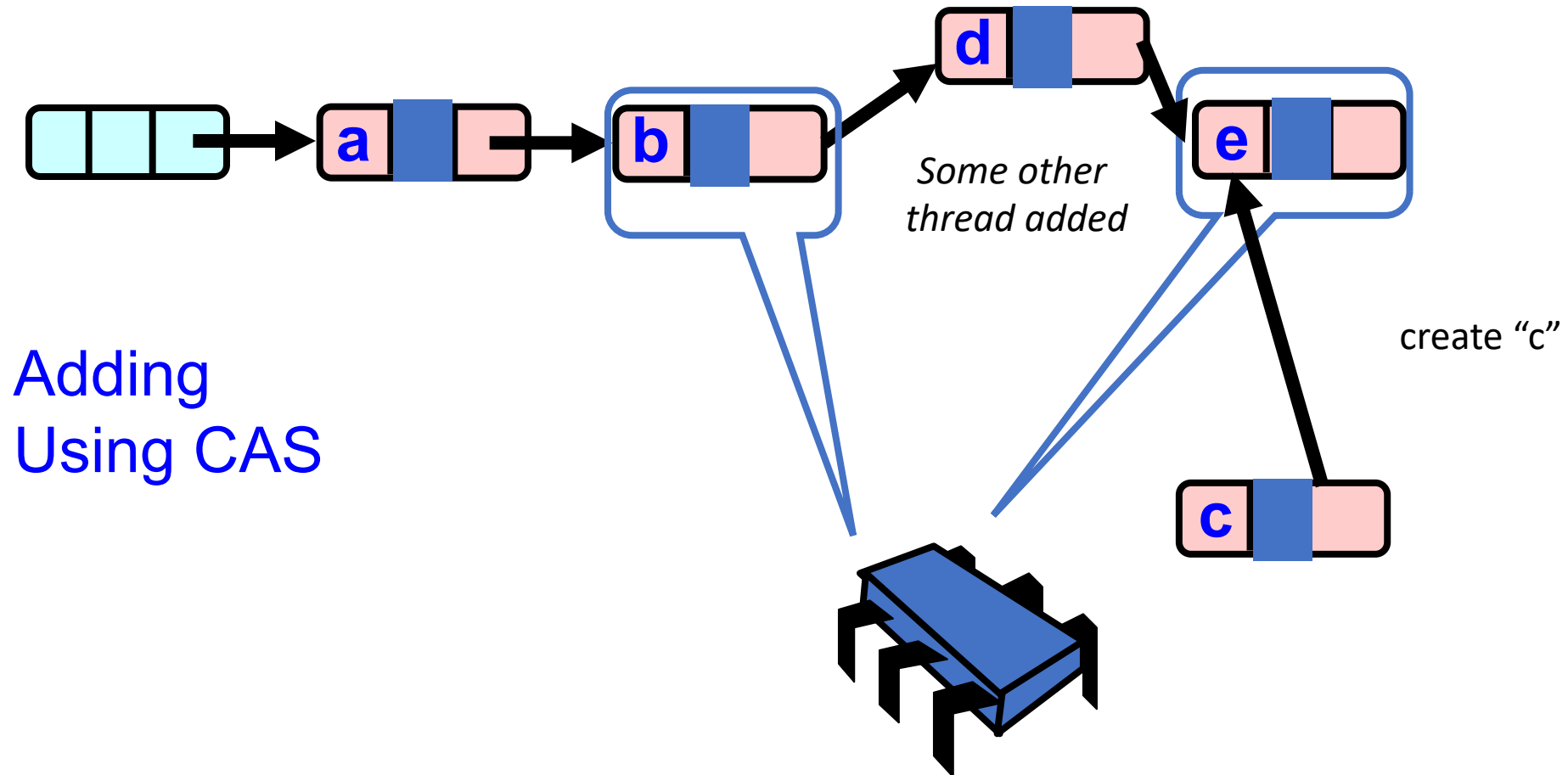
Only add if your insertion point is valid!

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

Find the location  
Cache your insertion point!

`b.next == e`

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





# Lock-free Lists

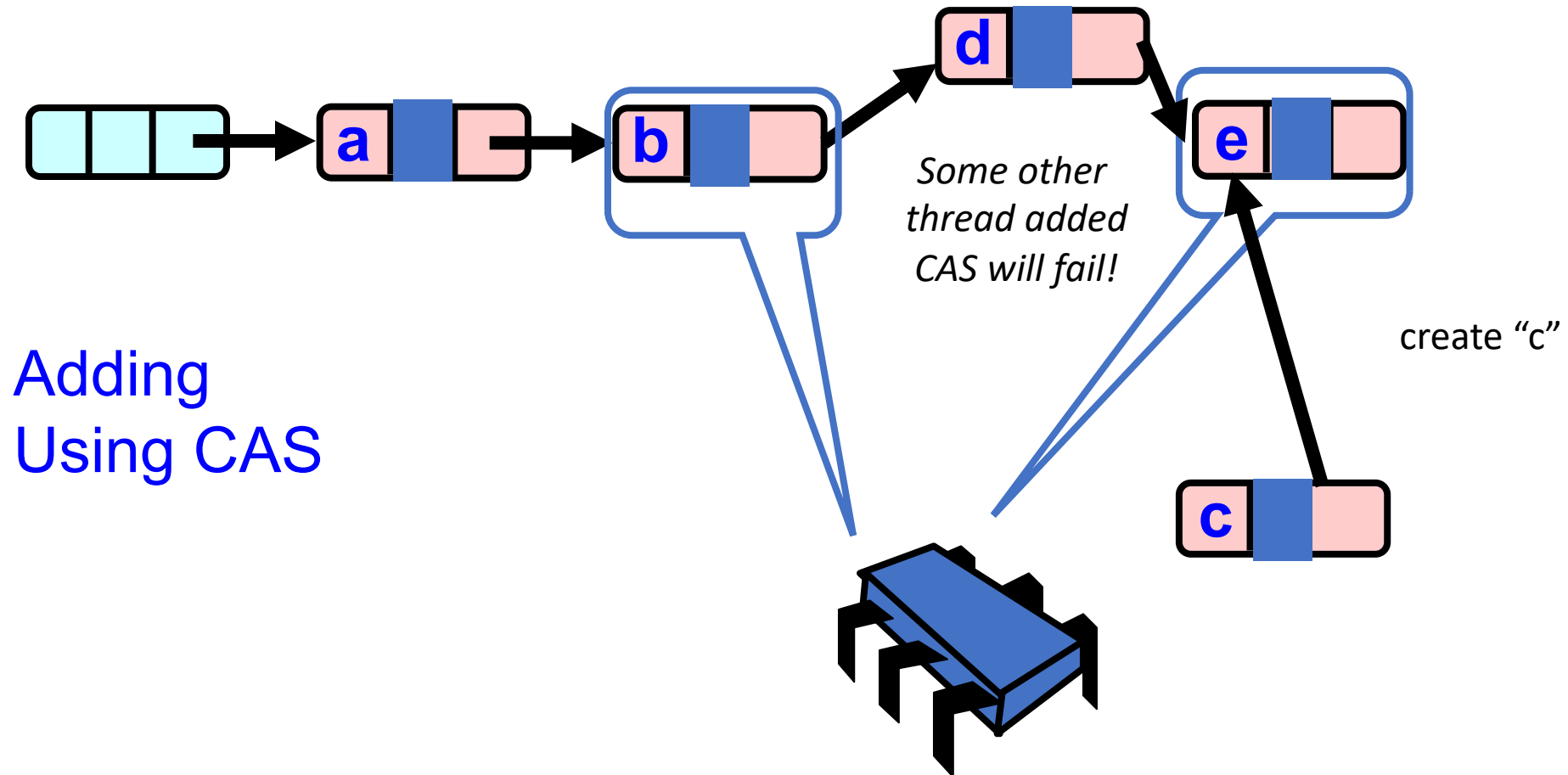
Only add if your insertion point is valid!

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

Find the location  
Cache your insertion point!

`b.next == e`

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



# Lock-free Lists

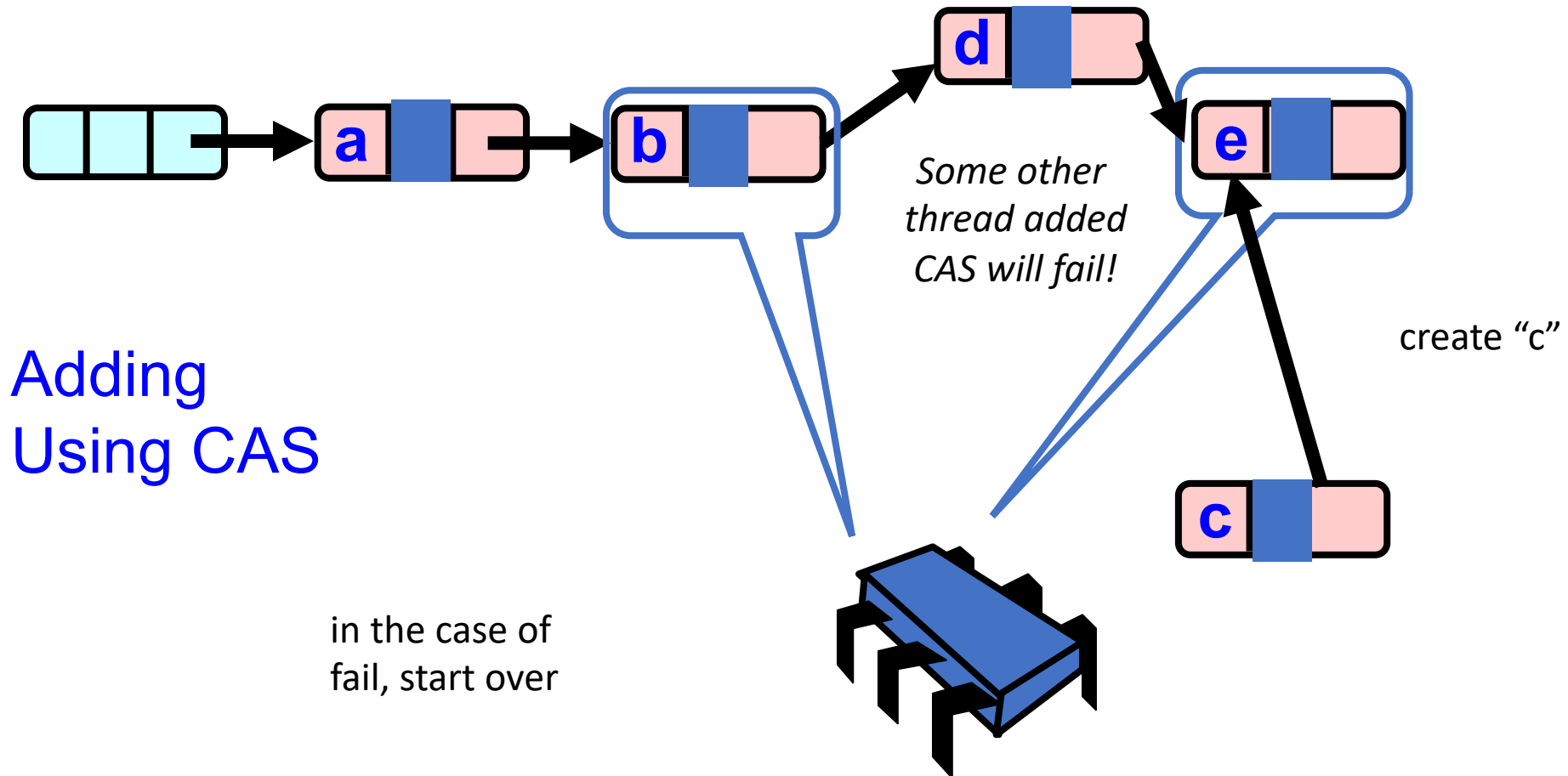
Only add if your insertion point is valid!

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

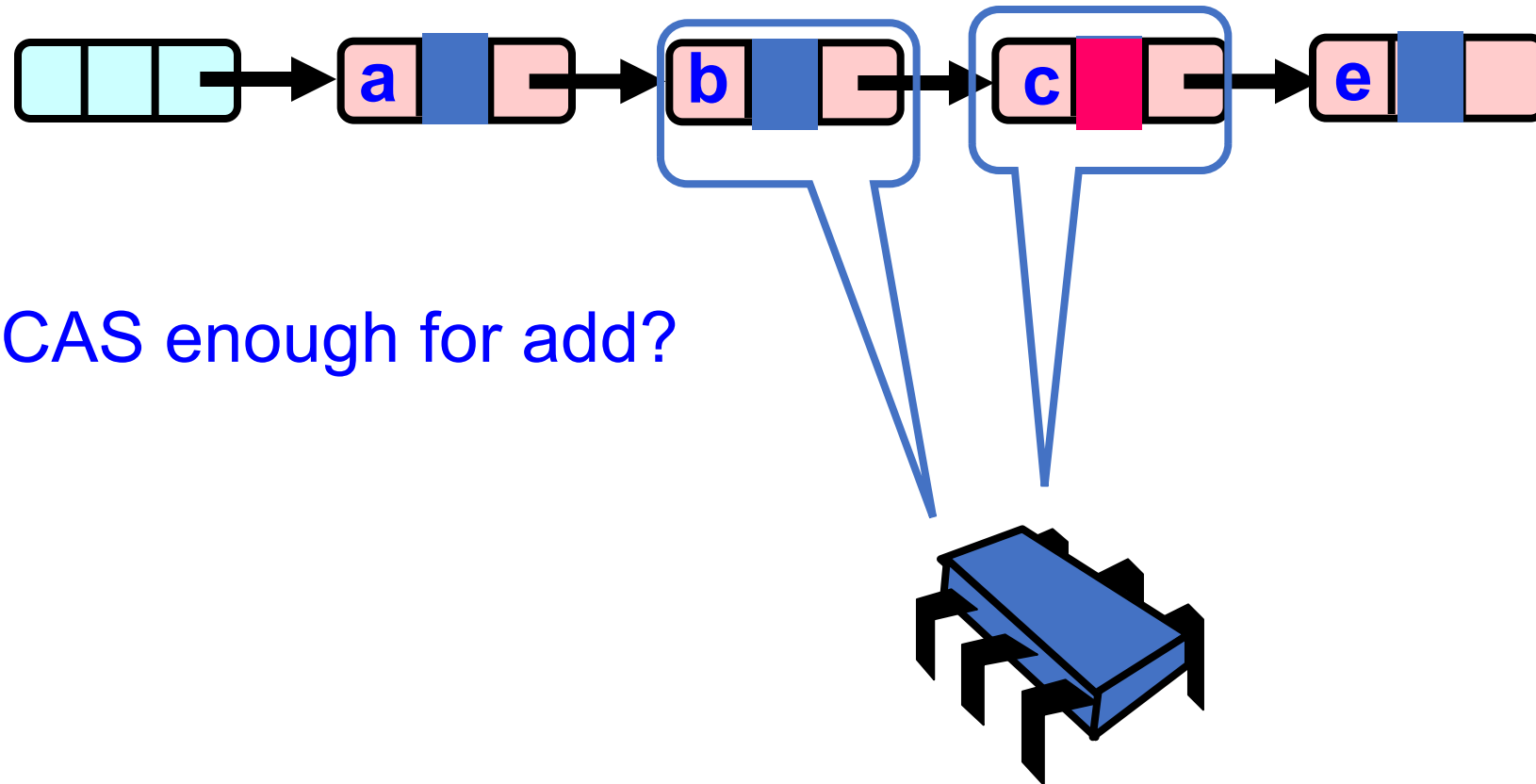
Find the location  
Cache your insertion point!

`b.next == e`

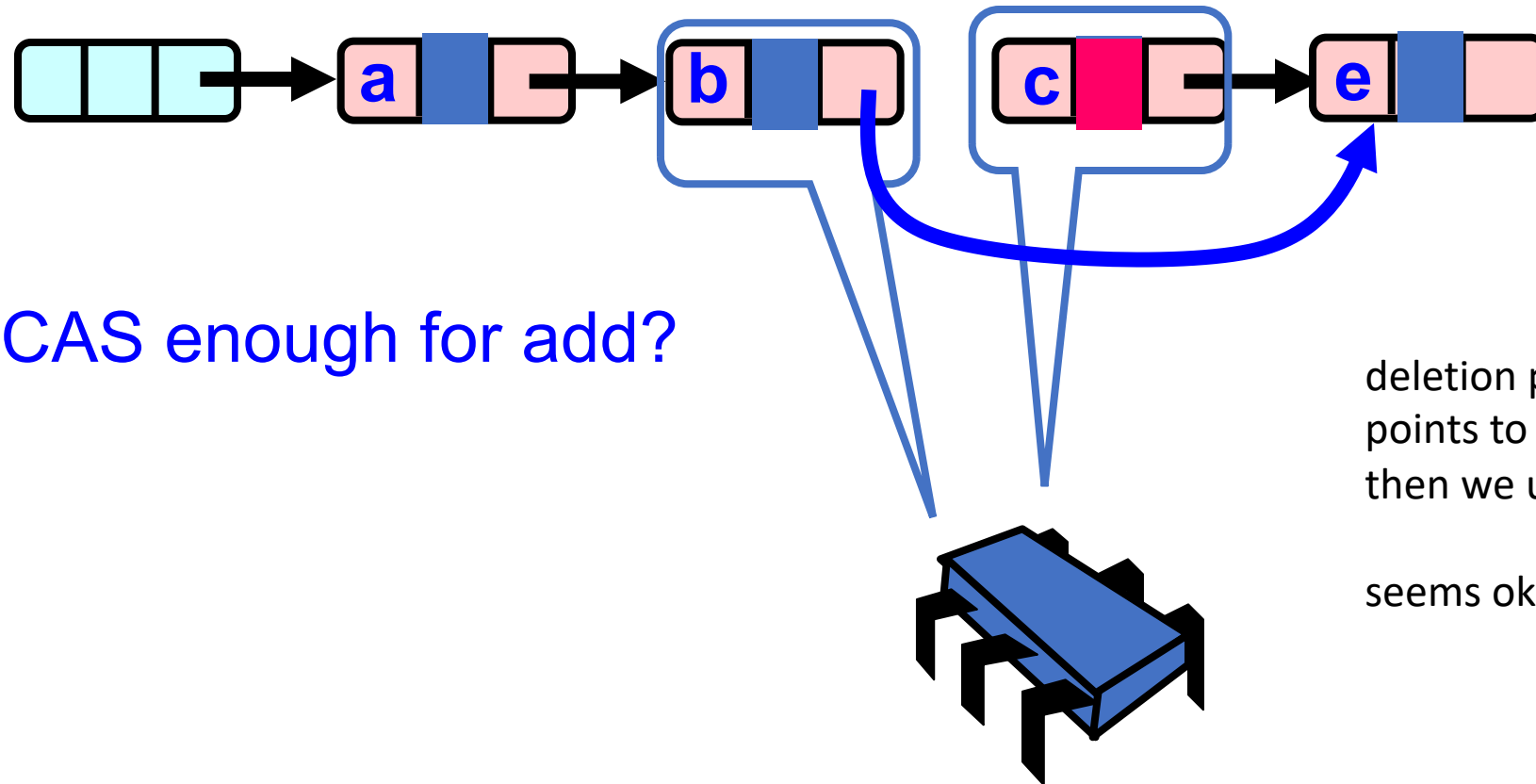
*notation is being abused here: e and c will be node \**



# Lock-free Lists



# Lock-free Lists



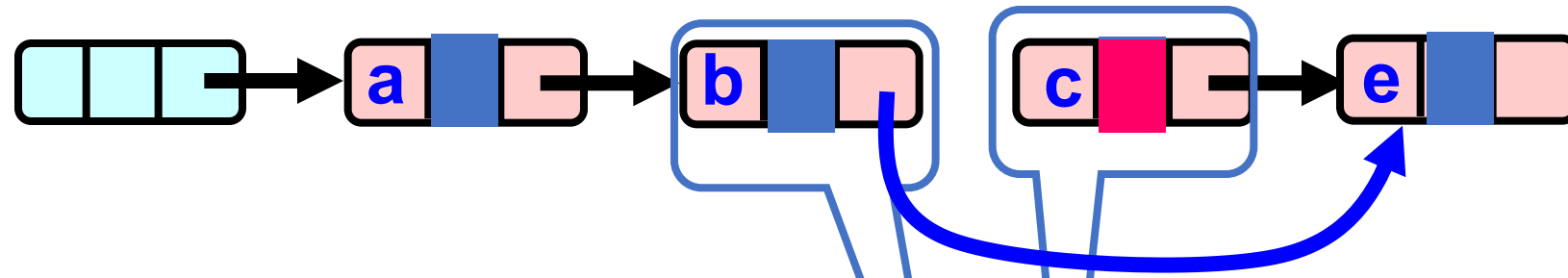
CAS enough for add?

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

seems okay...

# Lock-free Lists

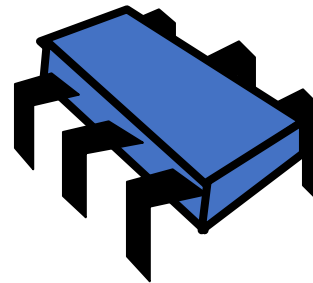
*ensures that nobody has added a node  
between b and c*



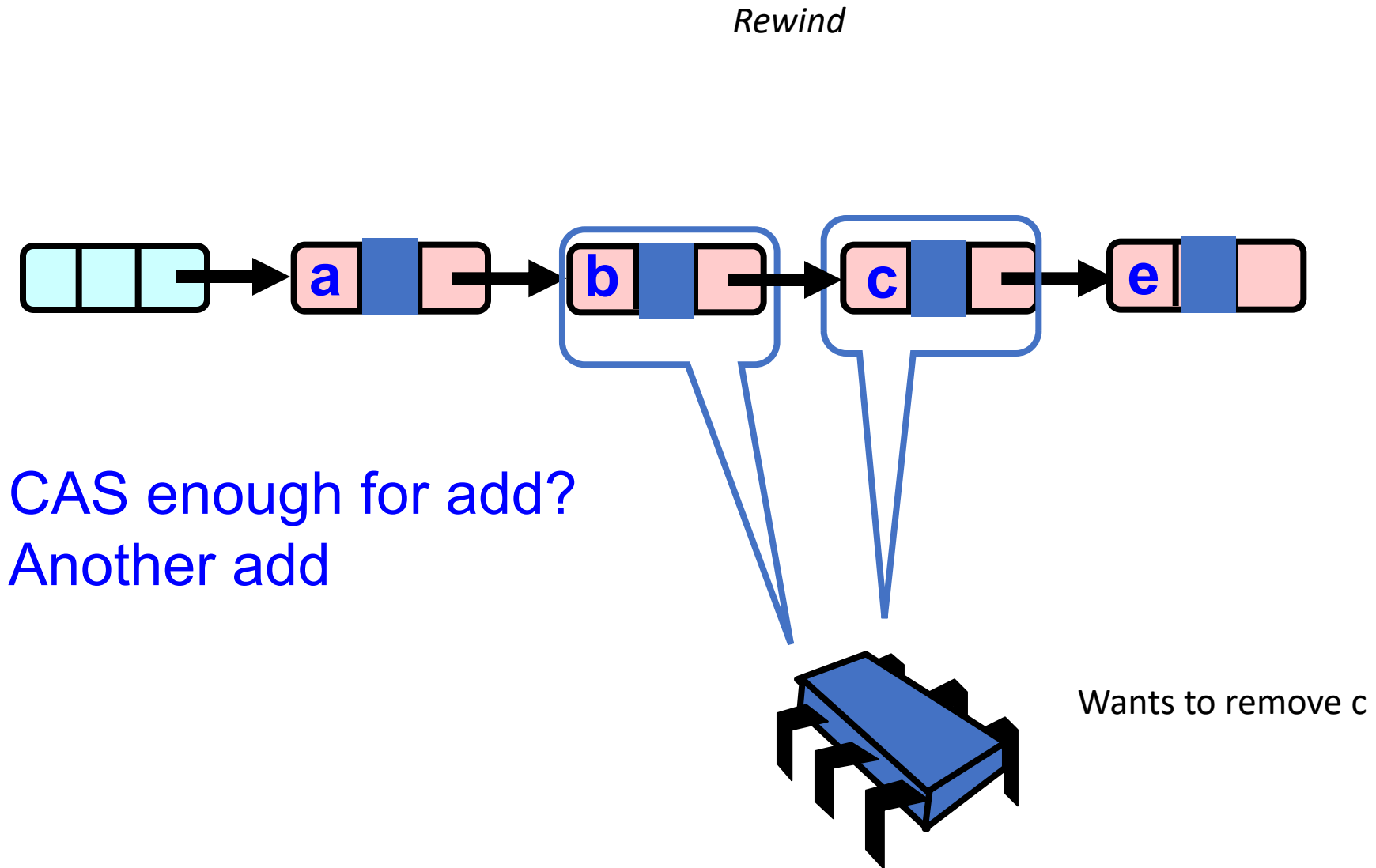
CAS enough for add?

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

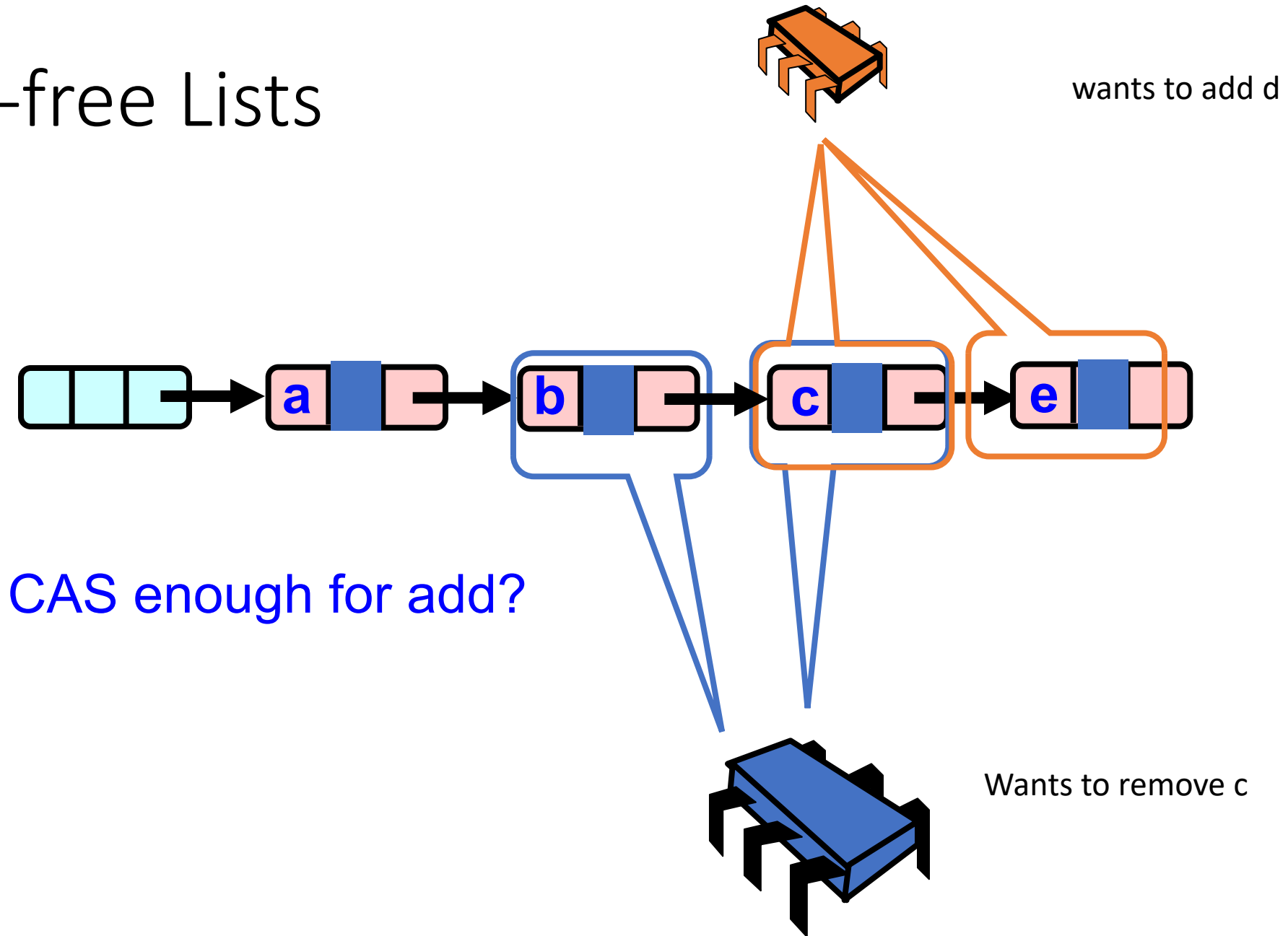
seems okay...



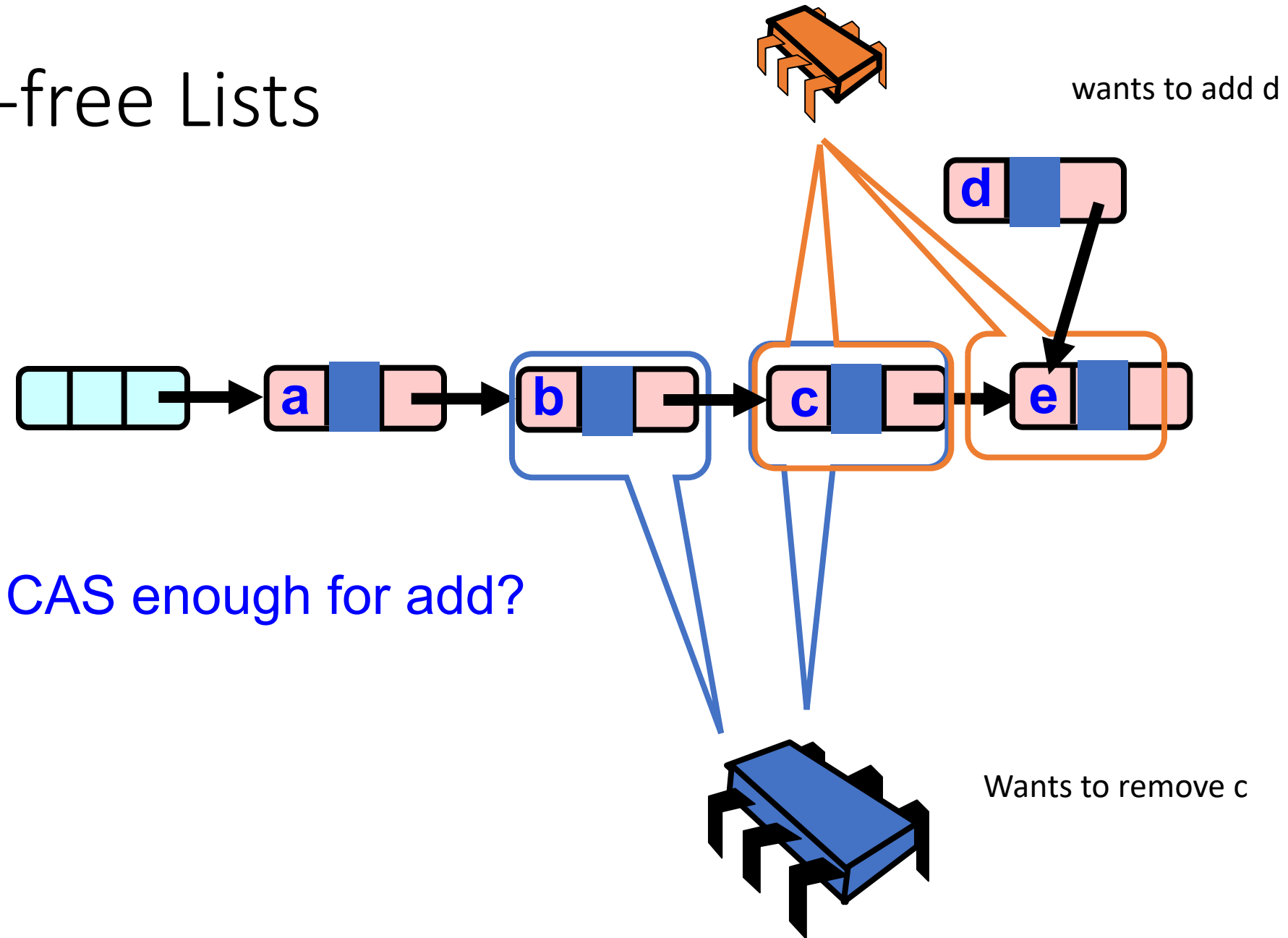
# Lock-free Lists



# Lock-free Lists

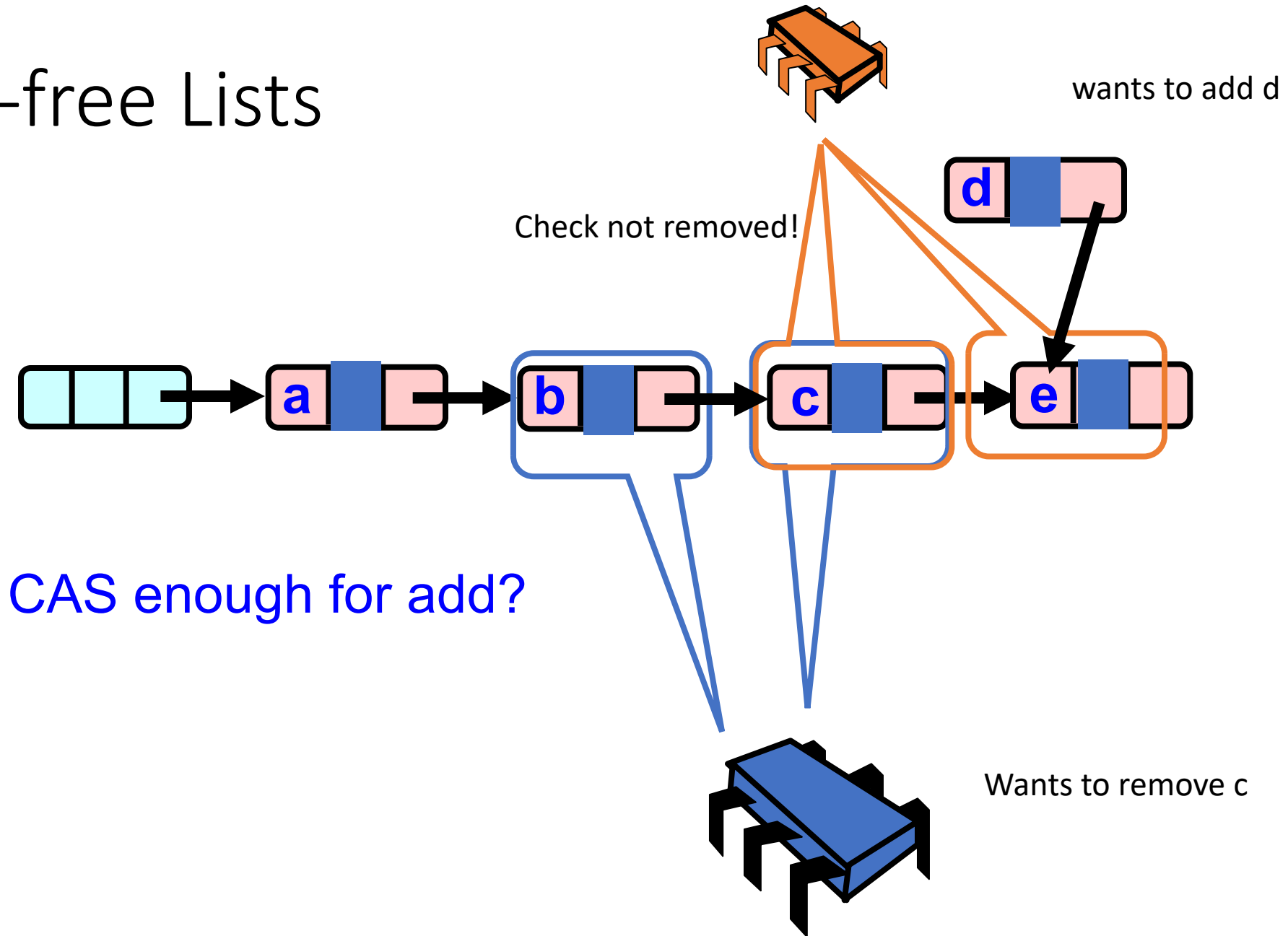


# Lock-free Lists

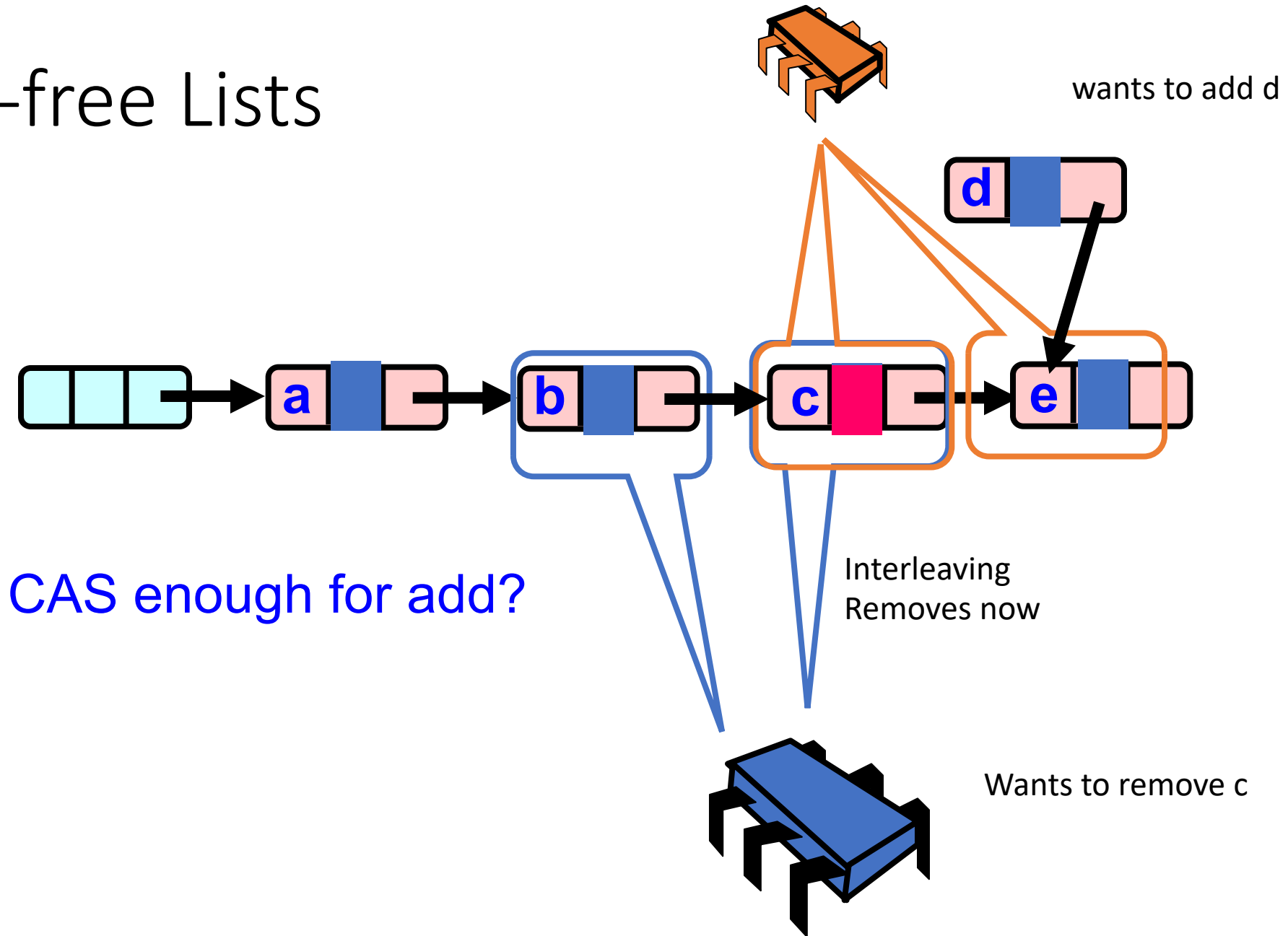




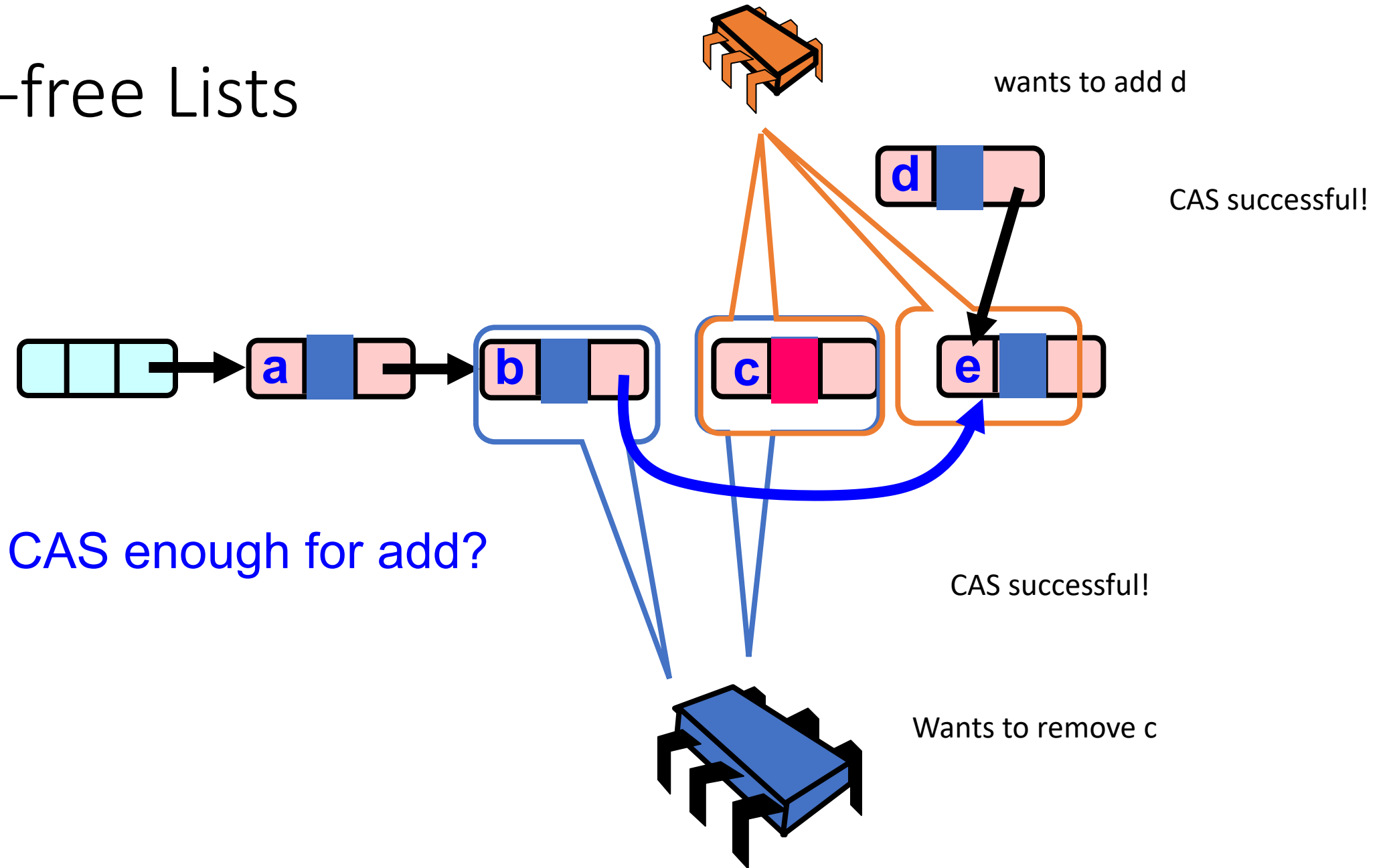
# Lock-free Lists



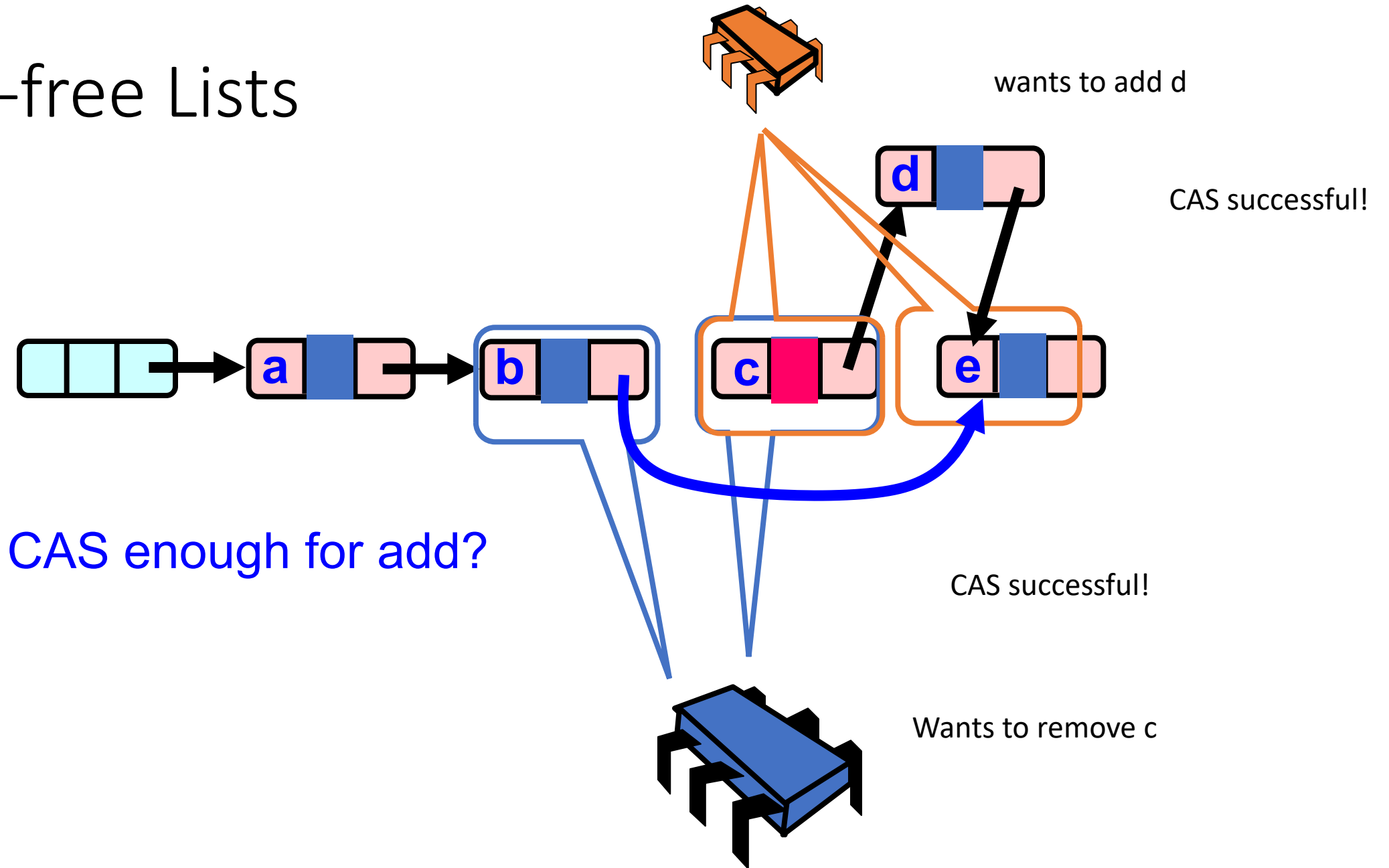
# Lock-free Lists



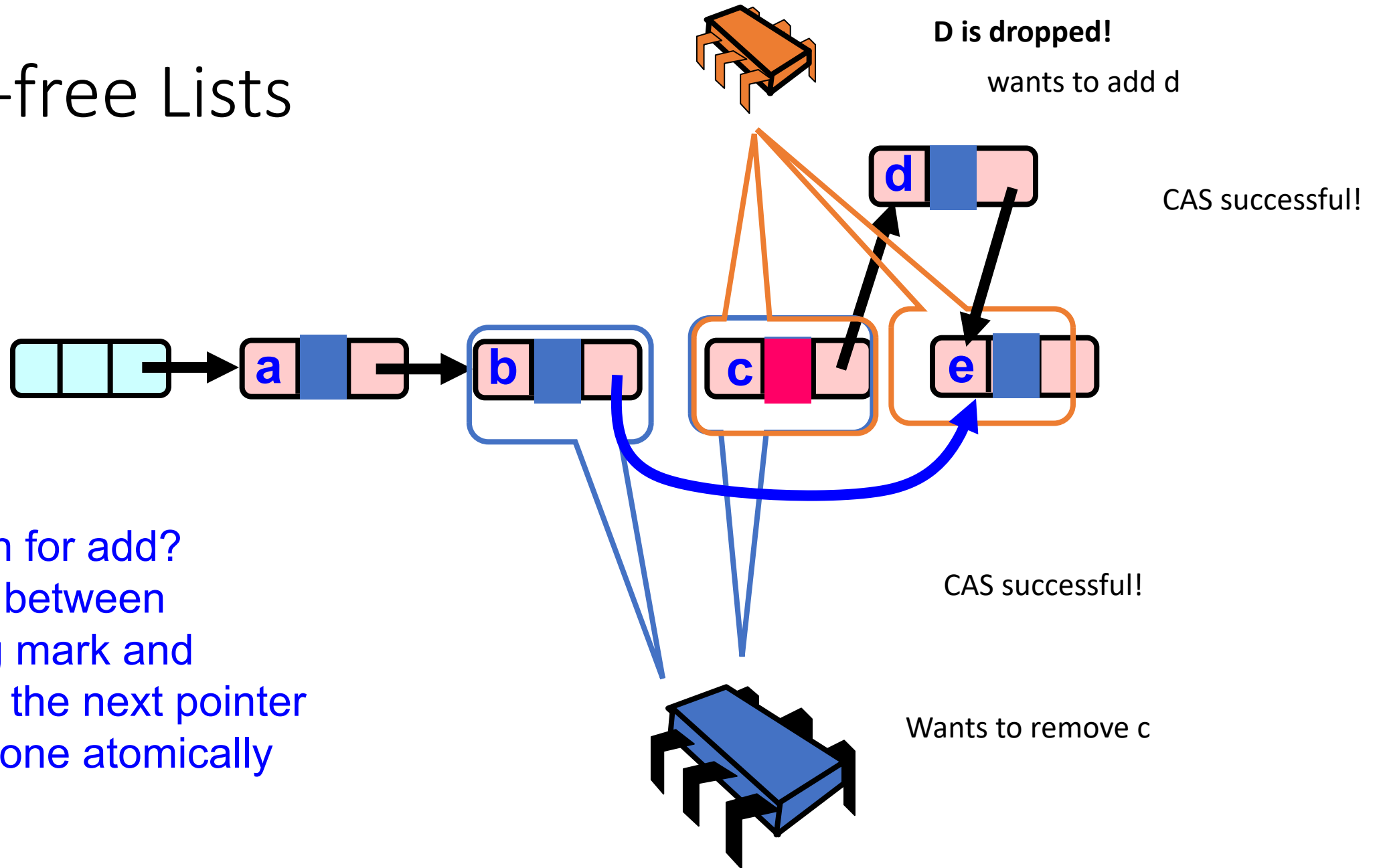
# Lock-free Lists



# Lock-free Lists



# Lock-free Lists



CAS enough for add?  
Interleaving between  
checking mark and  
updating the next pointer  
Should be done atomically

# Solution

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

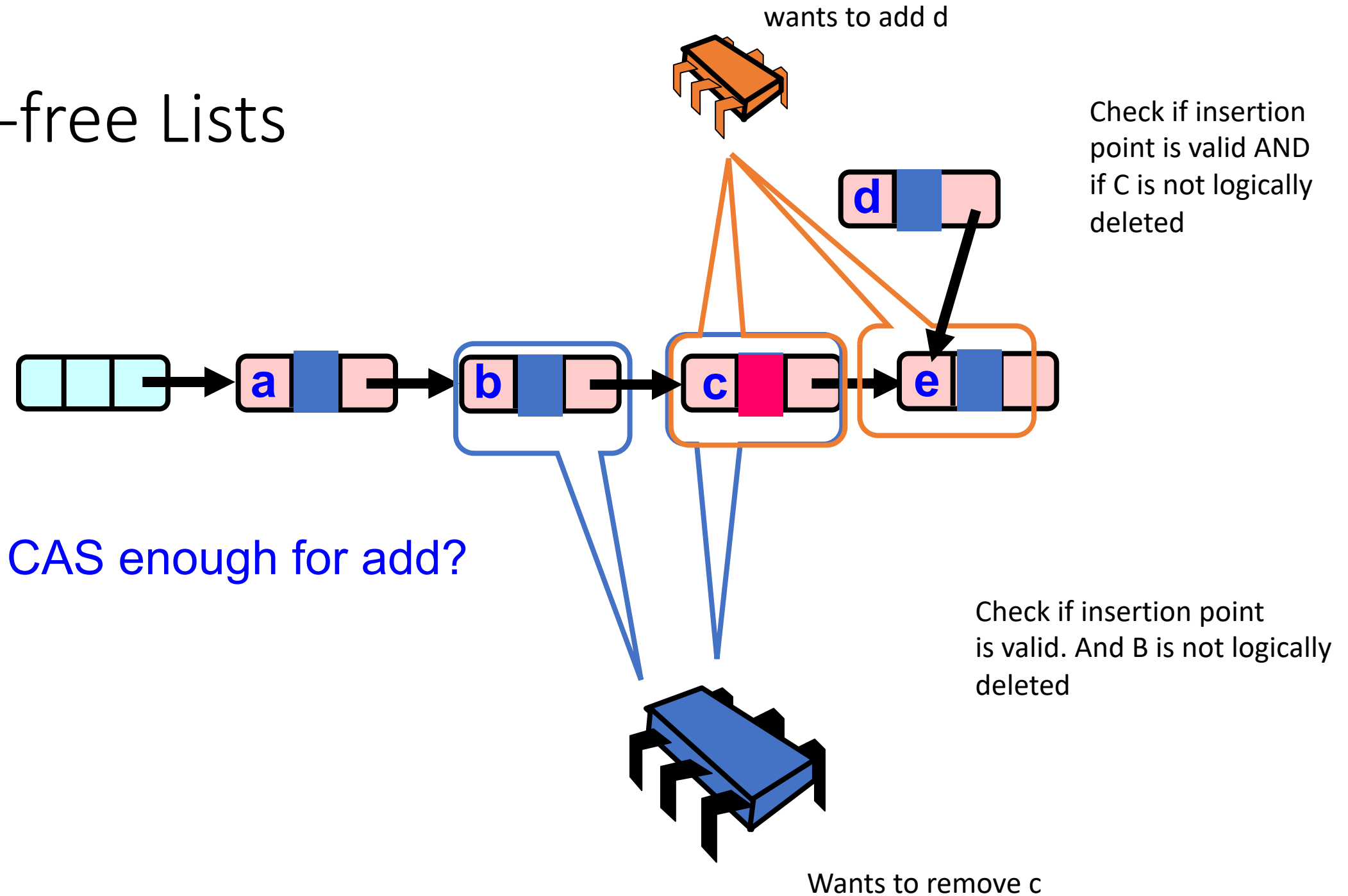
# Marking a Node

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



`CAS(c.next, <0, e>, <0, d>)`

# Lock-free Lists





# This stuff is tricky

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

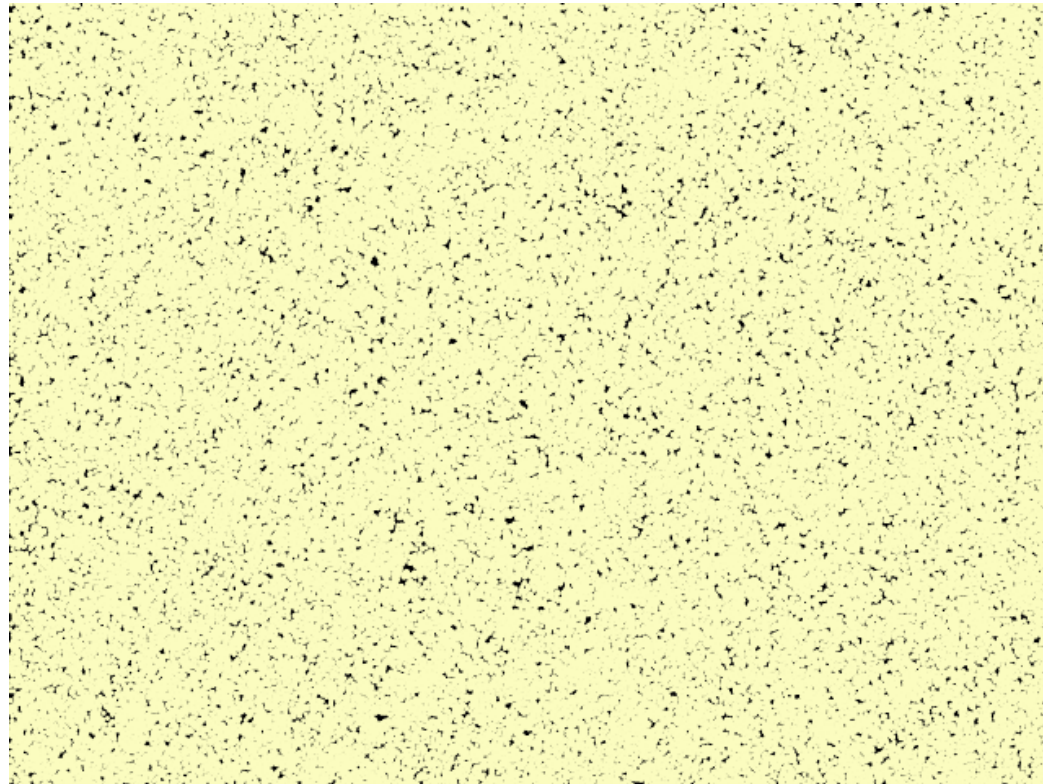
Barriers

# Barriers

- A barrier is a concurrent object (like a mutex):
  - Only one method: `barrier` (called `await` in the book)
- Separates computational phases

# Barrier Examples

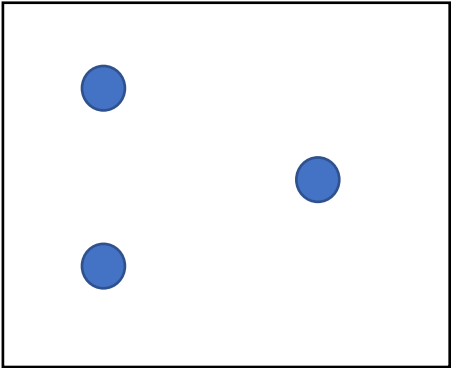
Particle simulation



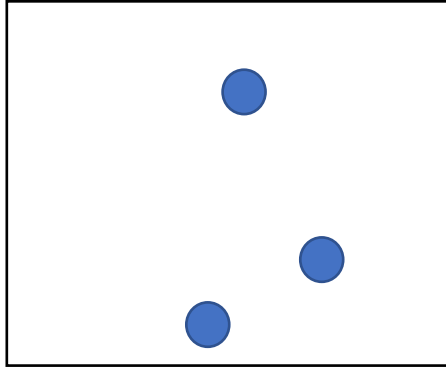
by Yanwen Xu

# Barrier Examples

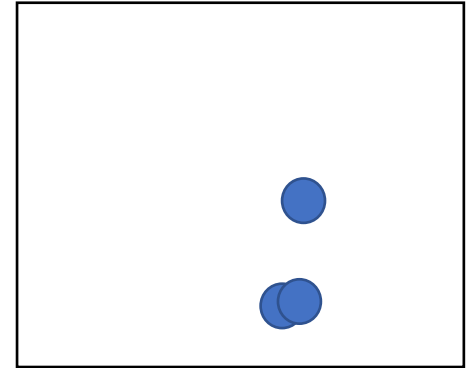
Particle simulation



time = 0



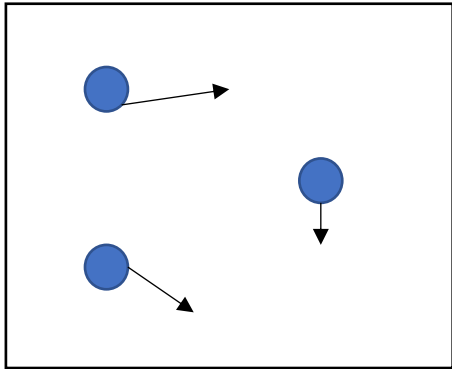
time = 1



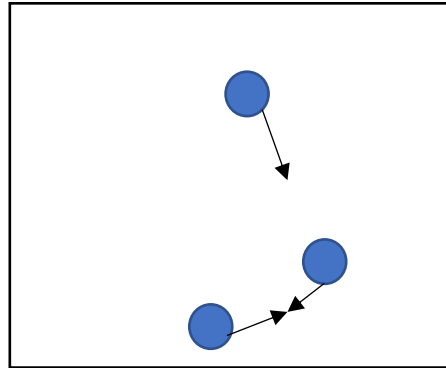
time = 2

# Barrier Examples

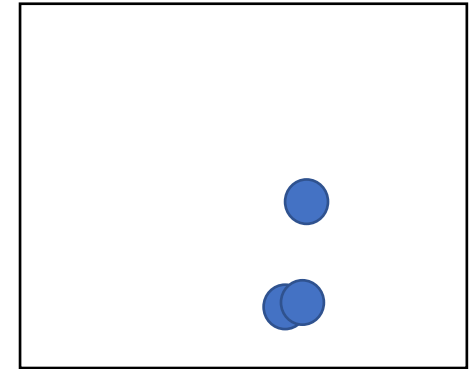
## Particle simulation



time = 0



time = 1

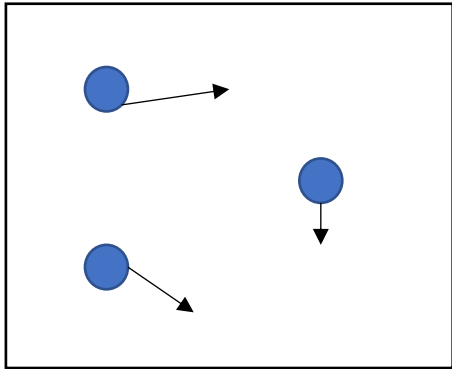


time = 2

At each time, compute  
new positions for each particle  
(in parallel)

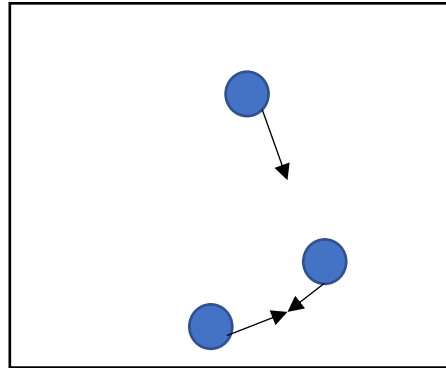
# Barrier Examples

## Particle simulation



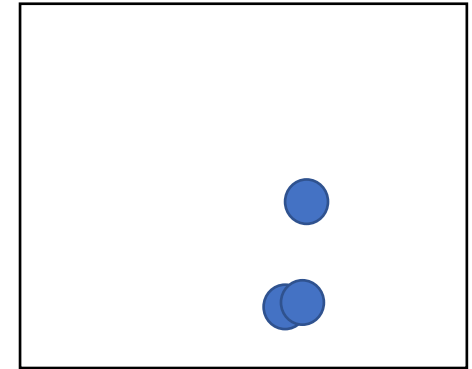
time = 0

`barrier();`



time = 1

`barrier();`



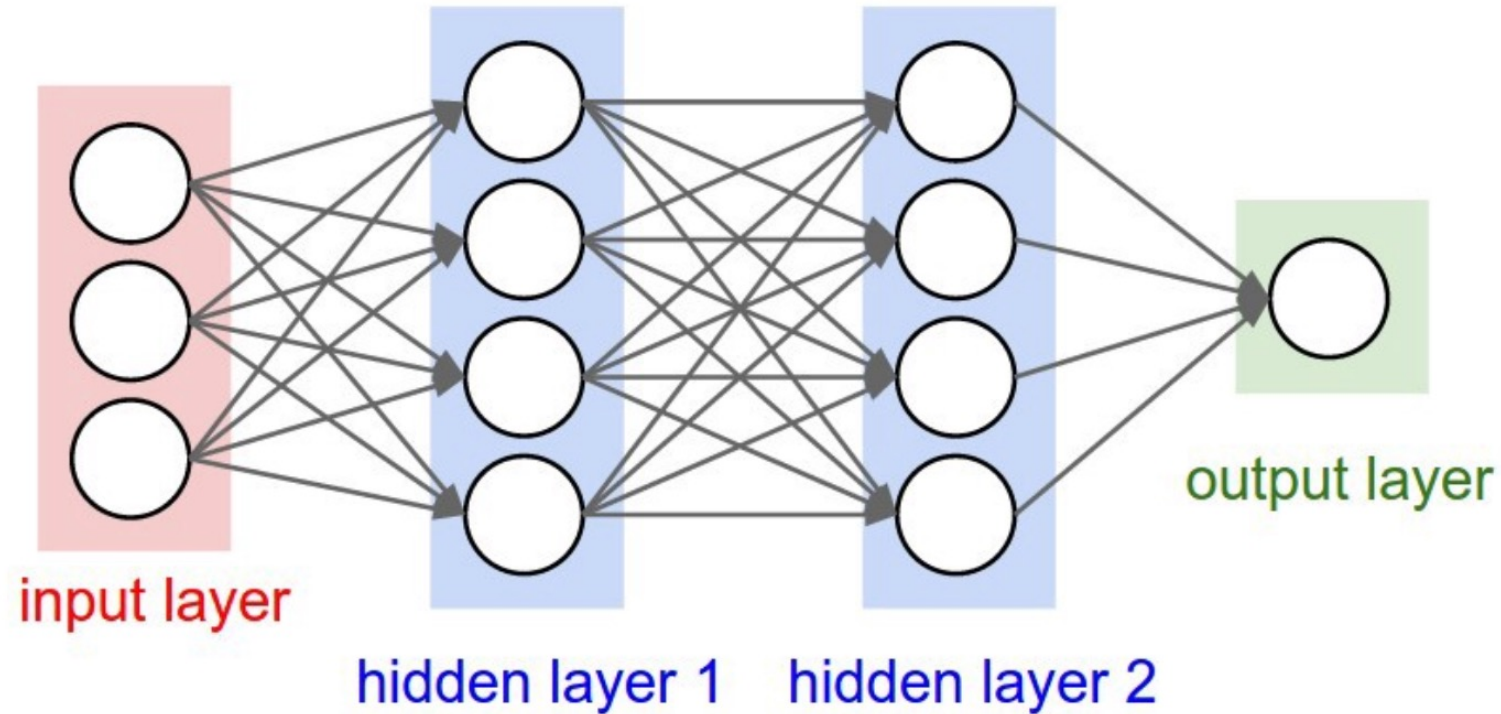
time = 2

At each time, compute  
new positions for each particle  
(in parallel)

But you need to wait for all particles to be  
computed before starting the next time step

# Barrier Examples

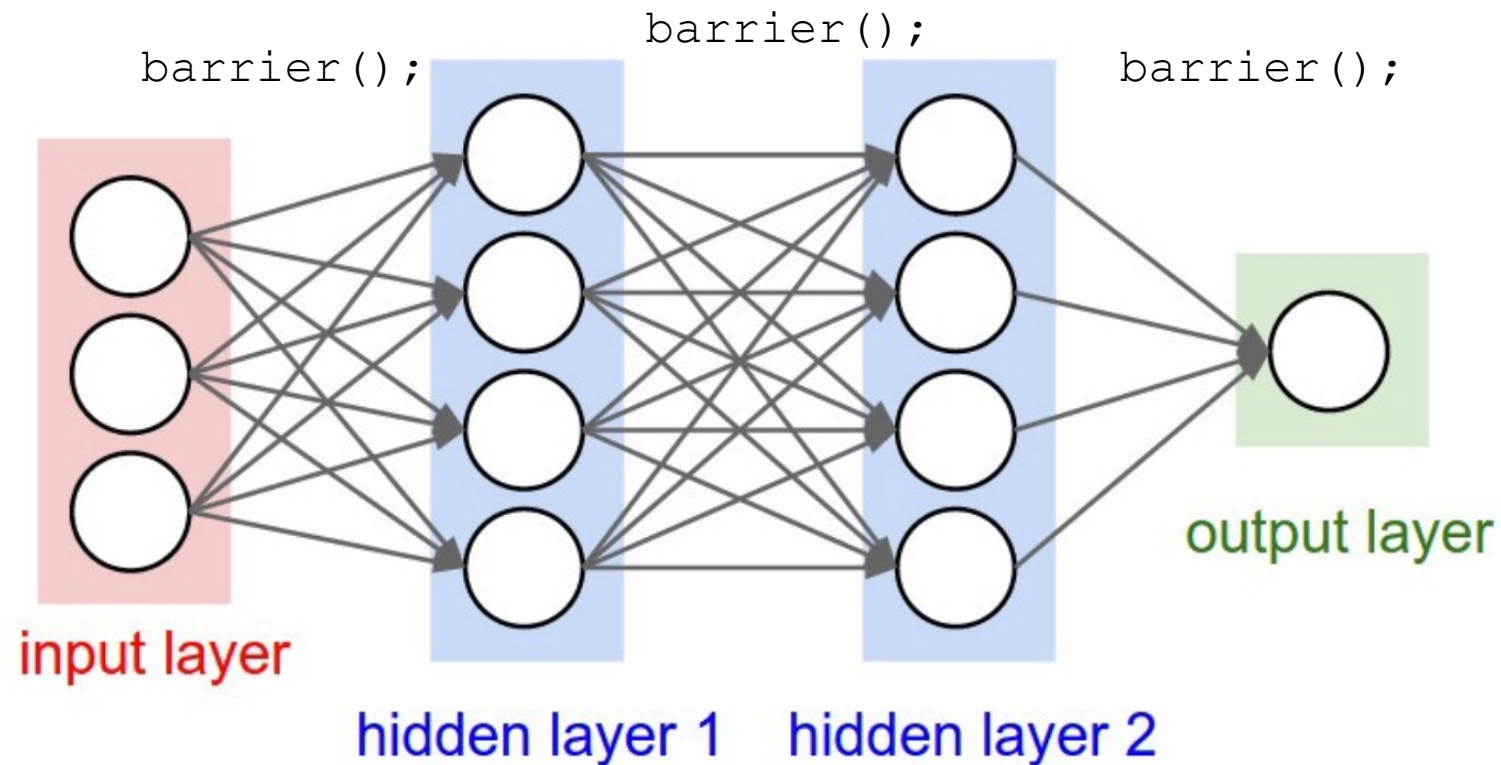
- Deep neural networks





# Barrier Examples

- Deep neural networks



# Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

# Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

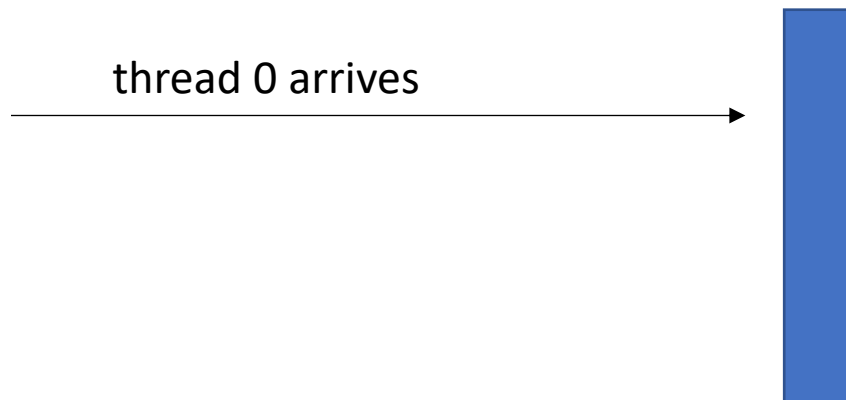
Example: say there are 4 threads: `barrier();`



# Barriers

- Intuition: threads stop and wait for each other:
  - Threads **arrive** at the barrier
  - Threads **wait** at the barrier
  - Threads **leave** the barrier once all other threads have arrived

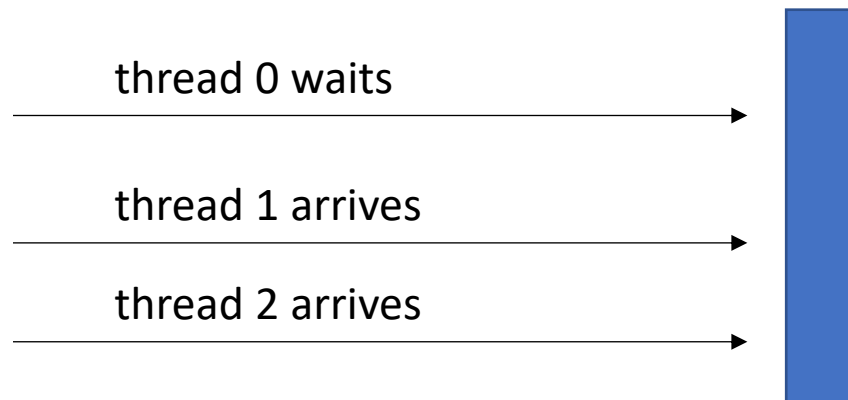
Example: say there are 4 threads: `barrier();`



# Barriers

- Intuition: threads stop and wait for each other:
  - Threads **arrive** at the barrier
  - Threads **wait** at the barrier
  - Threads **leave** the barrier once all other threads have arrived

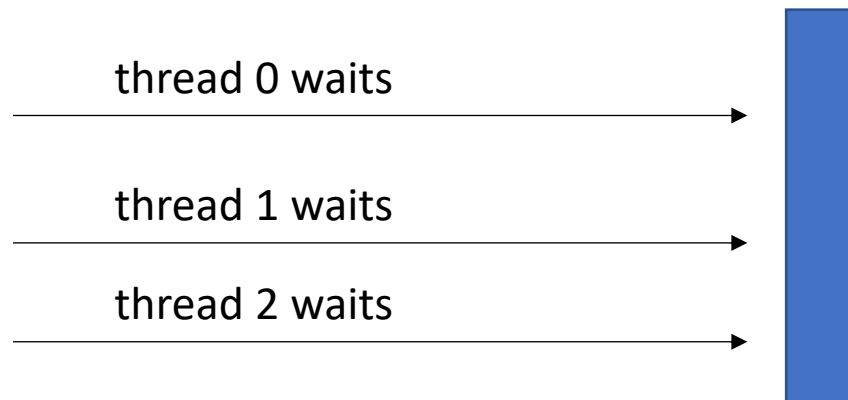
Example: say there are 4 threads: `barrier();`



# Barriers

- Intuition: threads stop and wait for each other:
  - Threads **arrive** at the barrier
  - Threads **wait** at the barrier
  - Threads **leave** the barrier once all other threads have arrived

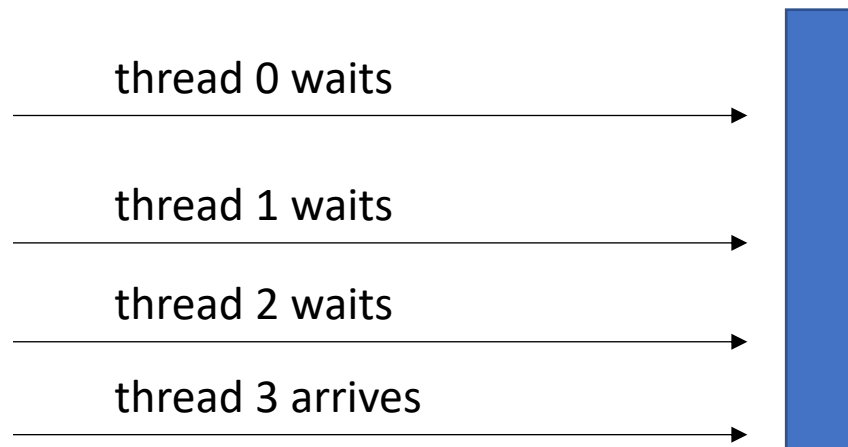
Example: say there are 4 threads: `barrier();`



# Barriers

- Intuition: threads stop and wait for each other:
  - Threads **arrive** at the barrier
  - Threads **wait** at the barrier
  - Threads **leave** the barrier once all other threads have arrived

Example: say there are 4 threads: `barrier();`

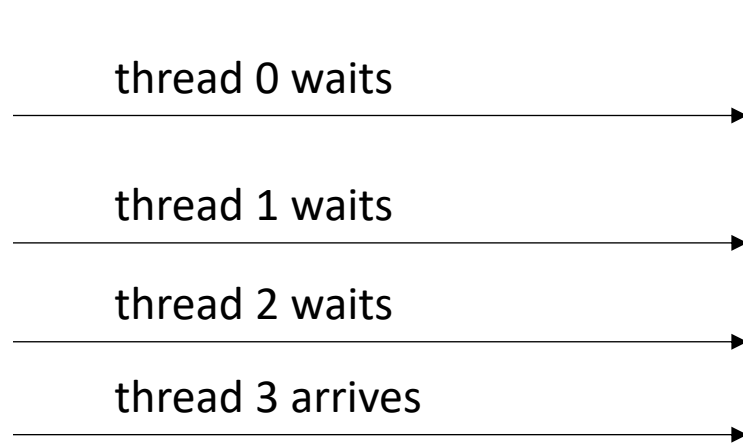


# Barriers

- Intuition: threads stop and wait for each other:
  - Threads **arrive** at the barrier
  - Threads **wait** at the barrier
  - Threads **leave** the barrier once all other threads have arrived

Example: say there are 4 threads:

`barrier();`



now that they have all arrived

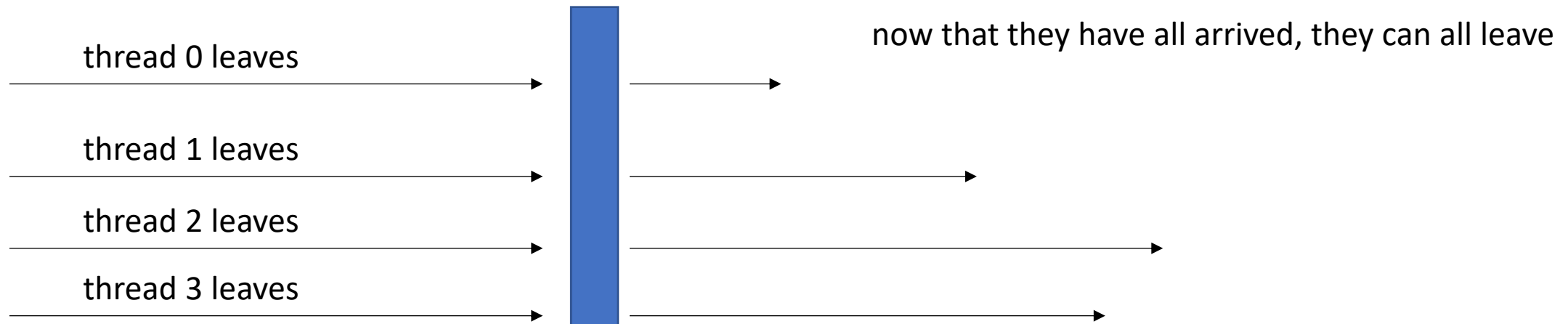


# Barriers

- Intuition: threads stop and wait for each other:
  - Threads **arrive** at the barrier
  - Threads **wait** at the barrier
  - Threads **leave** the barrier once all other threads have arrived

Example: say there are 4 threads:

`barrier();`



## A more formal specification

Given a global barrier  $B$   
and a global memory location  $x$  where  
initially  $*x = 0$ ;

First, what would we expect  
 $var$  to be after this program?

*Thread 0:*

```
*x = 1;
```

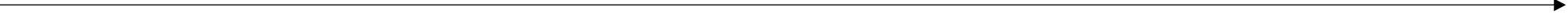
```
B.barrier();
```

*Thread 1:*

```
B.barrier();
```

```
var = *x;
```

thread 0 

thread 1 

## A more formal specification

Given a global barrier  $B$   
and a global memory location  $x$  where  
initially  $*x = 0$ ;

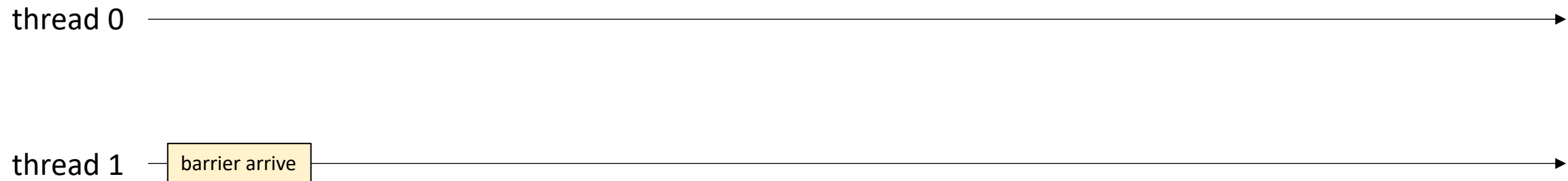
*Thread 0:*

```
*x = 1;  
B.barrier();
```

*Thread 1:*

```
B.barrier();  
var = *x;
```

gives an event:  
barrier arrive



## A more formal specification

Given a global barrier  $B$   
and a global memory location  $x$  where  
initially  $*x = 0$ ;

Thread 0:

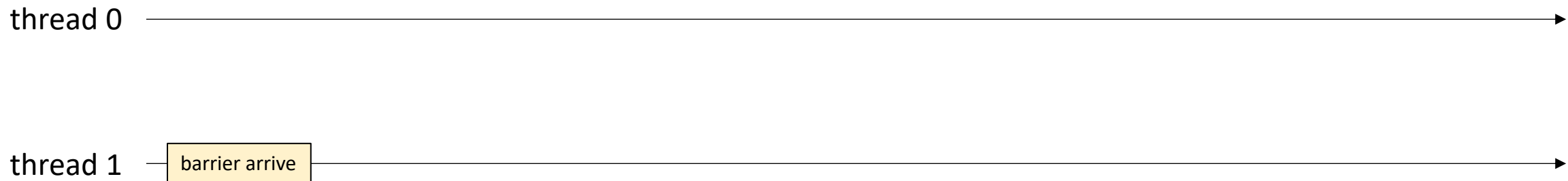
```
*x = 1;  
B.barrier();
```

Thread 1:

```
B.barrier();  
var = *x;
```

gives an event:  
barrier arrive

barrier arrive needs to wait for all threads  
to arrive (similar to how a mutex request must wait for  
another to release)



## A more formal specification

Given a global barrier  $B$   
and a global memory location  $x$  where  
initially  $*x = 0$ ;

Thread 0:

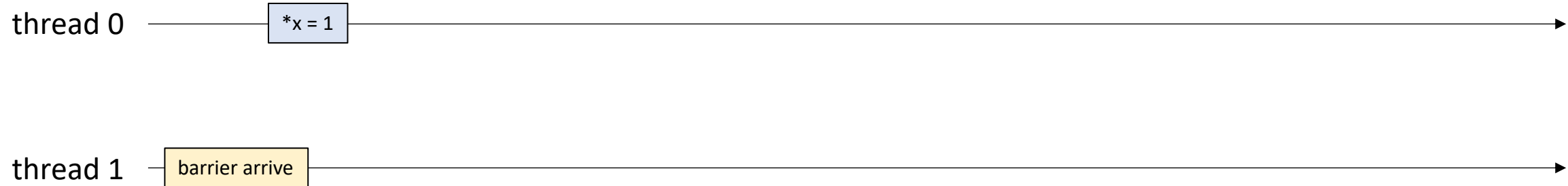
```
*x = 1;
```

```
B.barrier();
```

Thread 1:

```
B.barrier();
```

```
var = *x;
```



## A more formal specification

Given a global barrier  $B$   
and a global memory location  $x$  where  
initially  $*x = 0$ ;

Thread 0:

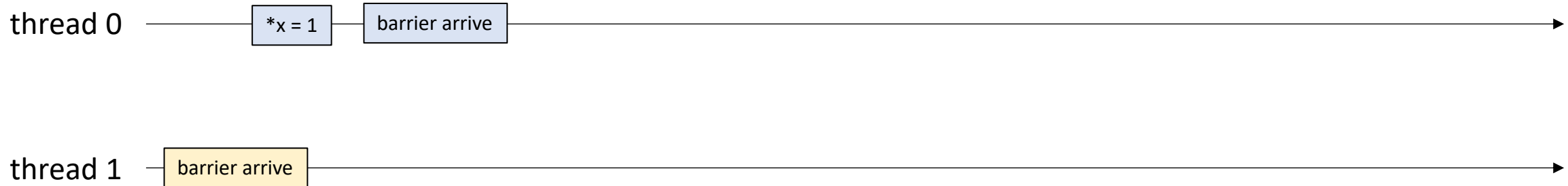
```
*x = 1;
```

```
B.barrier();
```

Thread 1:

```
B.barrier();
```

```
var = *x;
```



## A more formal specification

Given a global barrier  $B$   
and a global memory location  $x$  where  
initially  $*x = 0$ ;

Thread 0:

```
*x = 1;
```

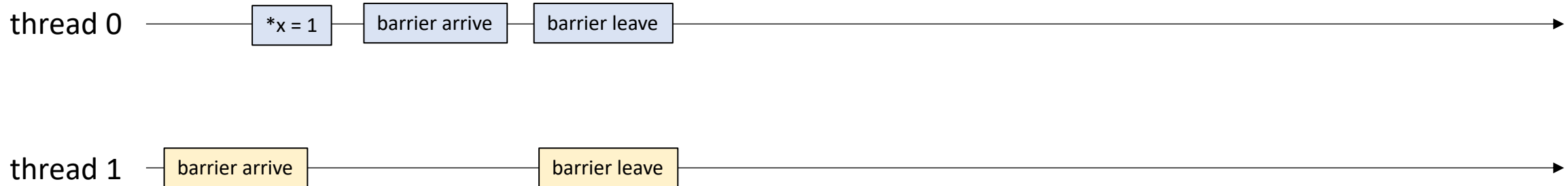
```
B.barrier();
```

Thread 1:

```
B.barrier();
```

```
var = *x;
```

now that all threads have arrived:  
They can leave (1 event at the same time)



## A more formal specification

Given a global barrier  $B$   
and a global memory location  $x$  where  
initially  $*x = 0$ ;

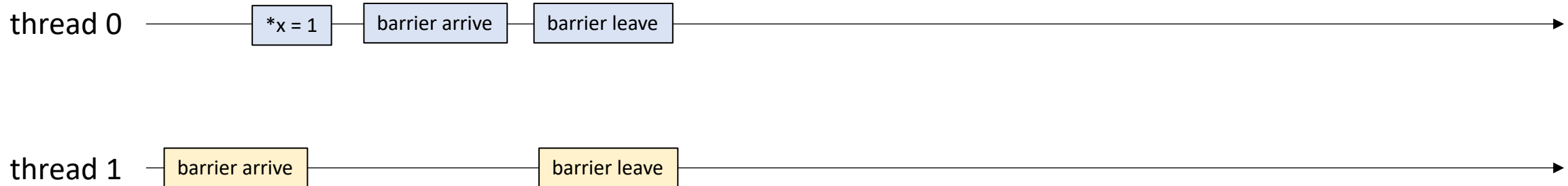
### Thread 0:

```
*x = 1;  
B.barrier();
```

### Thread 1:

```
B.barrier();  
var = *x;
```

This finishes the barrier execution





## A more formal specification

Given a global barrier B  
and a global memory location x where  
initially  $*x = 0$ ;

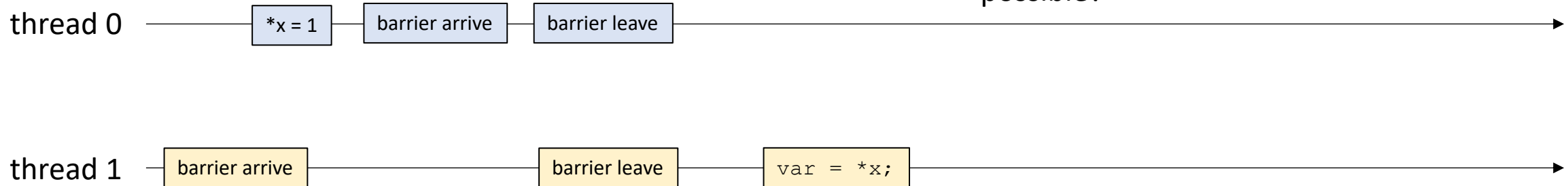
### Thread 0:

```
*x = 1;  
B.barrier();
```

### Thread 1:

```
B.barrier();  
var = *x;
```

what value must this read? Any other value possible?



One more example, assume initially  $*x = *y = 0$

*Thread 0:*

```
*x = 1;  
B.barrier();
```

*Thread 1:*

```
*y = 2;  
B.barrier();
```

*Thread 2:*

```
B.barrier();  
var = *x + *y;
```

thread 0 →

thread 1 →

thread 2 →

One more example, assume initially  $*x = *y = 0$

Thread 0:

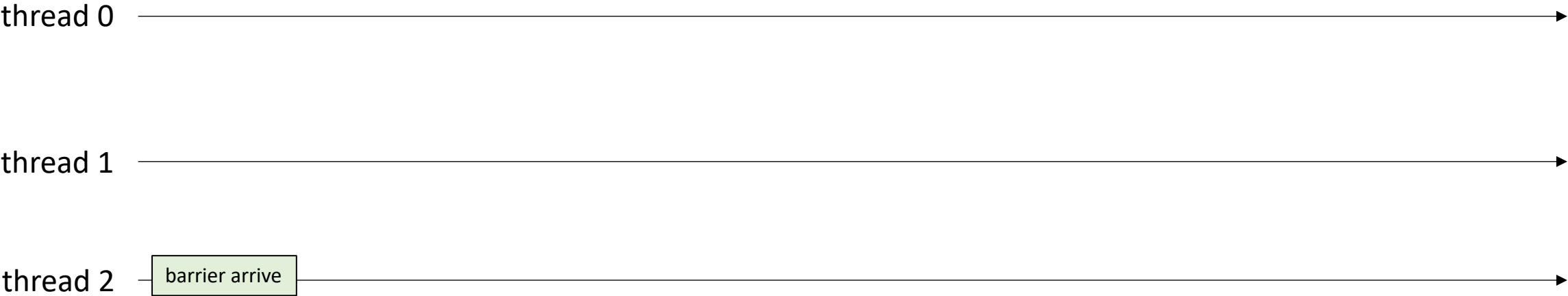
```
*x = 1;  
B.barrier();
```

Thread 1:

```
*y = 2;  
B.barrier();
```

Thread 2:

```
B.barrier();  
var = *x + *y;
```

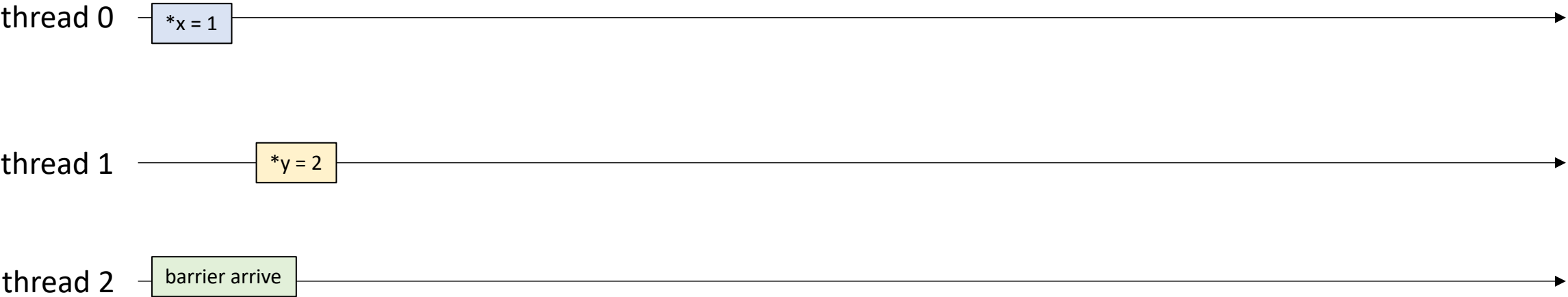


One more example, assume initially  $*x = *y = 0$

Thread 0:  
`*x = 1;`  
`B.barrier();`

Thread 1:  
`*y = 2;`  
`B.barrier();`

Thread 2:  
`B.barrier();`  
`var = *x + *y;`

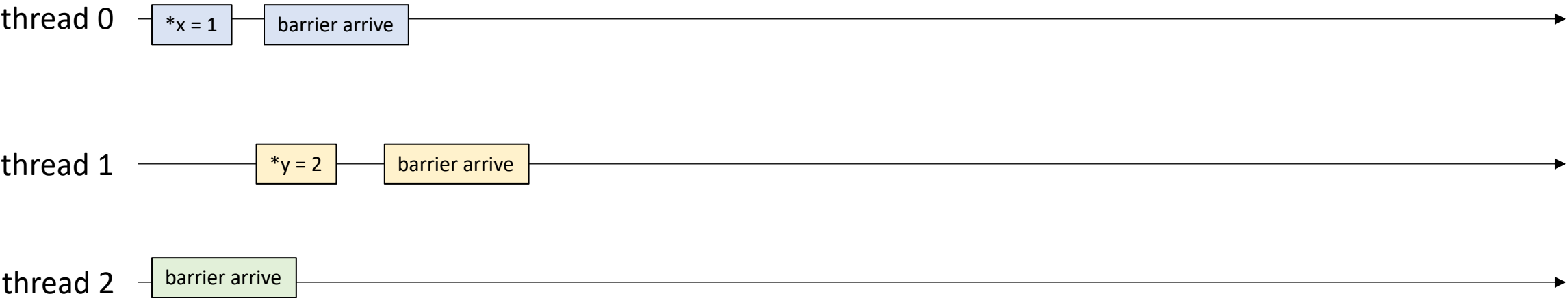


One more example, assume initially  $*x = *y = 0$

Thread 0:  
`*x = 1;`  
`B.barrier();`

Thread 1:  
`*y = 2;`  
`B.barrier();`

Thread 2:  
`B.barrier();`  
`var = *x + *y;`



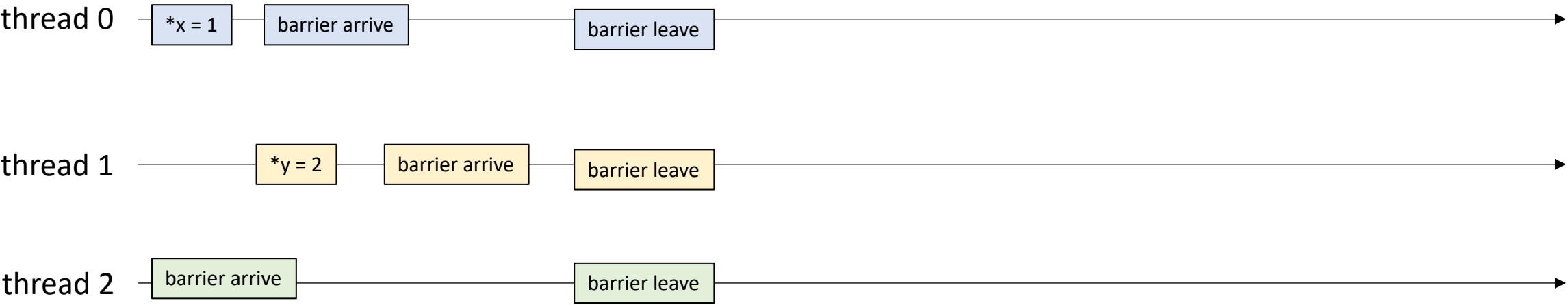
One more example, assume initially  $*x = *y = 0$

Thread 0:  
`*x = 1;`  
`B.barrier();`

Thread 1:  
`*y = 2;`  
`B.barrier();`

Thread 2:  
`B.barrier();`  
`var = *x + *y;`

They've all arrived



One more example, assume initially  $*x = *y = 0$

Thread 0:

```
*x = 1;  
B.barrier();
```

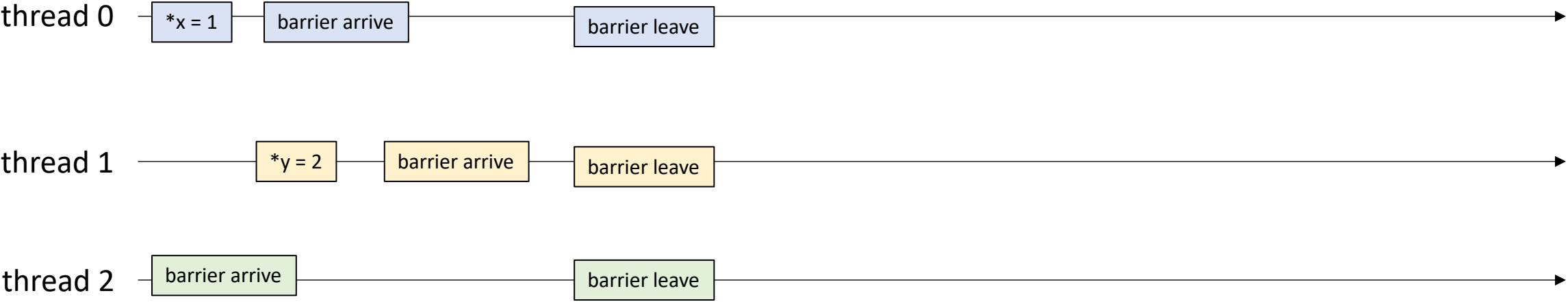
Thread 1:

```
*y = 2;  
B.barrier();
```

Thread 2:

```
B.barrier();  
var = *x + *y;
```

They've all arrived

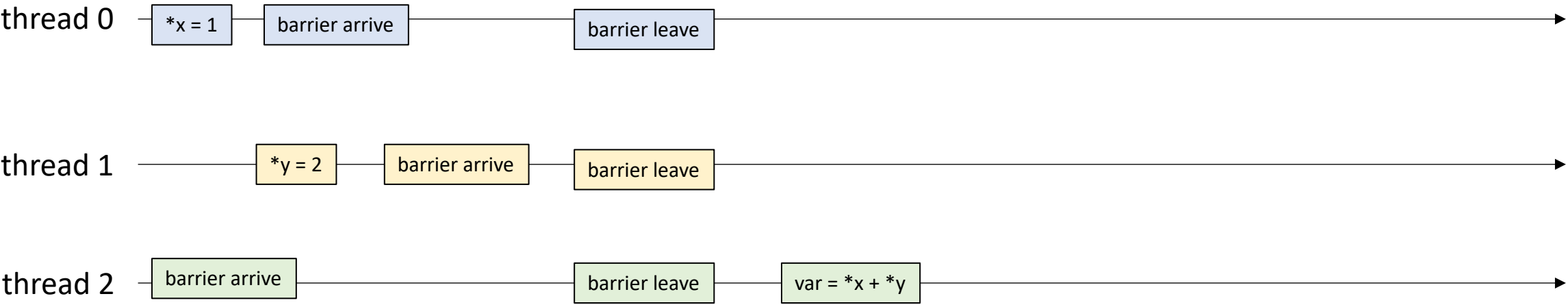


One more example, assume initially  $*x = *y = 0$

Thread 0:  
`*x = 1;`  
`B.barrier();`

Thread 1:  
`*y = 2;`  
`B.barrier();`

Thread 2:  
`B.barrier();`  
`var = *x + *y;`



What is this guaranteed to be?



One more example, assume initially  $*x = *y = 0$

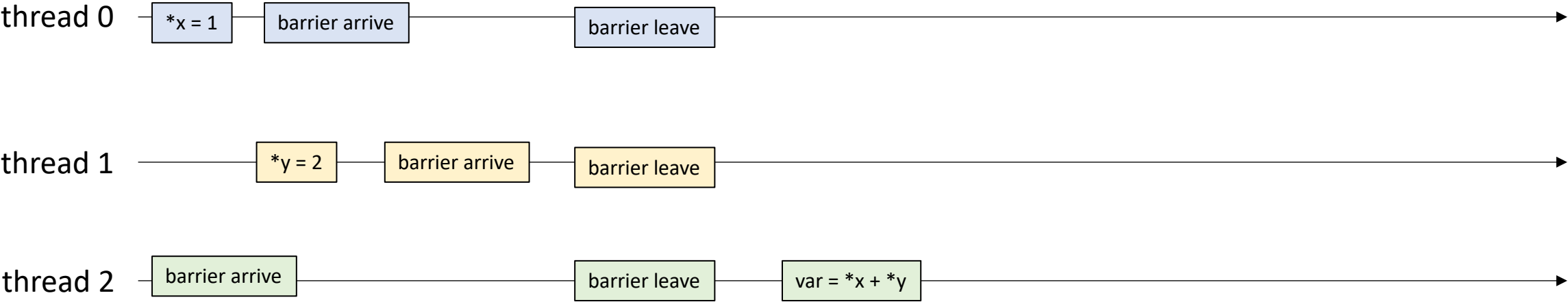
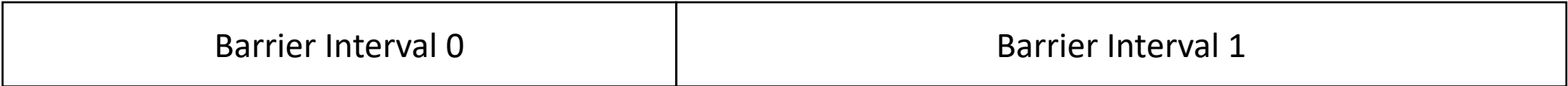
Thread 0:  
`*x = 1;`  
`B.barrier();`

Thread 1:  
`*y = 2;`  
`B.barrier();`

Thread 2:  
`B.barrier();`  
`var = *x + *y;`

sometimes called a *phase*

extending to the next *barrier leave*

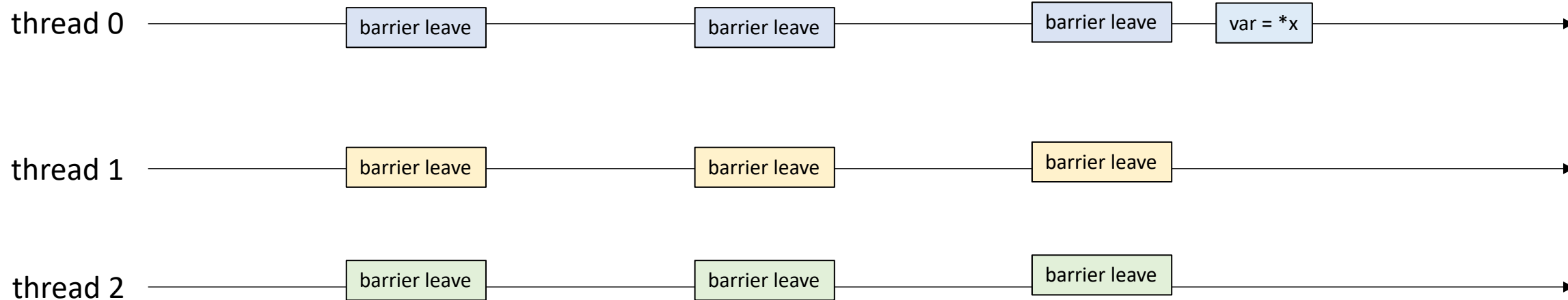
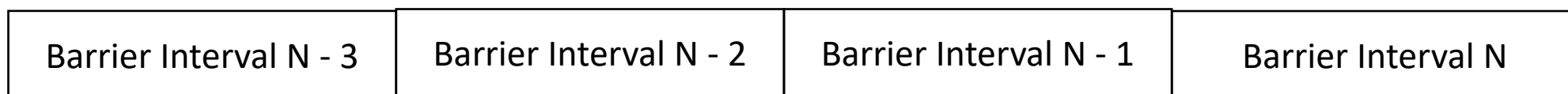


# Barriers

- Barrier Property:
  - If the only concurrent object you use in your program is a barrier (no mutexes, concurrent data-structures, atomic accesses)
  - If every barrier interval contains no data conflicts, then  
***your program will be deterministic (only 1 outcome allowed)***
  - much easier to reason about 😊

Assume we are reading  
from x

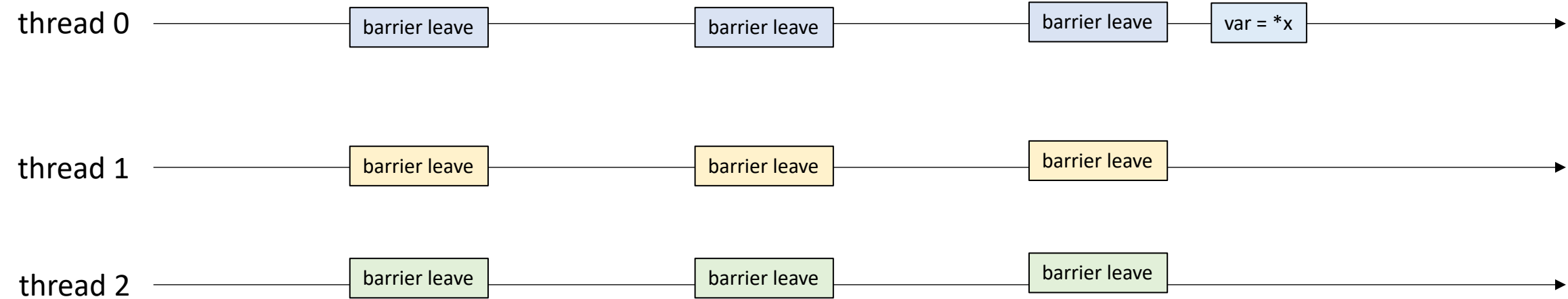
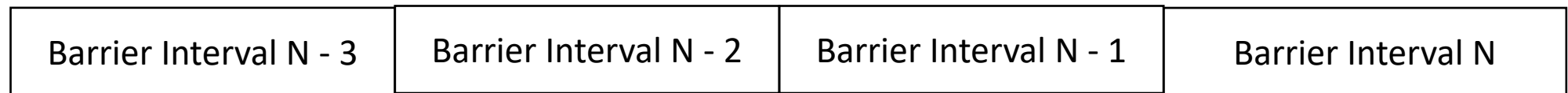
We are only allowed to  
return one possible  
value



no data conflicts means that x is written to at most once  
per barrier interval

Assume we are reading  
from x

We are only allowed to  
return one possible  
value

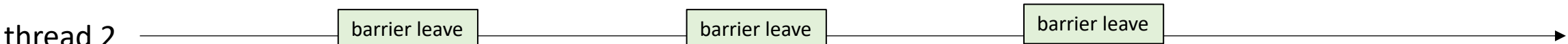
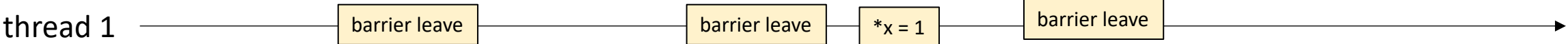
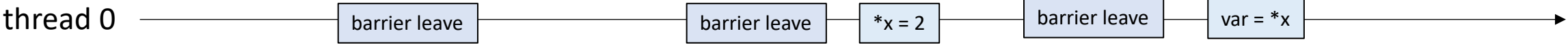
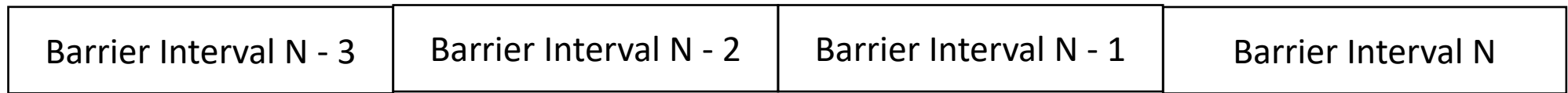


no data conflicts means that x is written to at most once  
per barrier interval

Assume we are reading  
from x

We are only allowed to  
return one possible  
value

not allowed

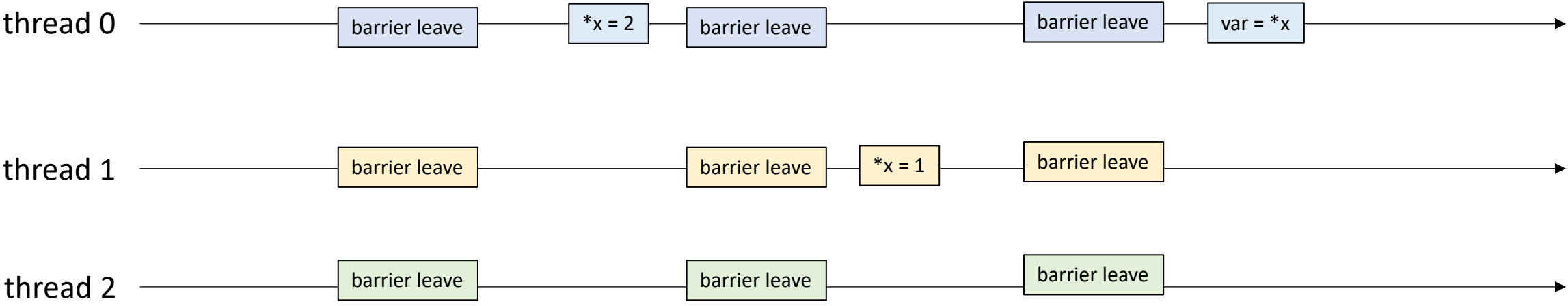
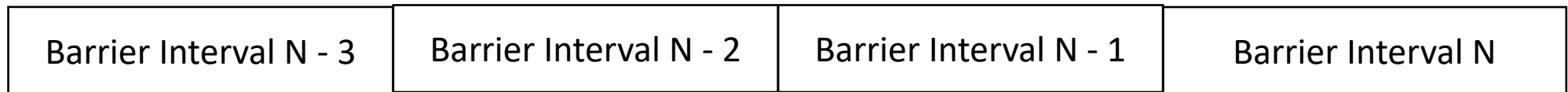


no data conflicts means that x is written to at most once  
per barrier interval

Assume we are reading  
from x

we will read from the write  
from the most recent barrier interval

We are only allowed to  
return one possible  
value



# Schedule

- **Barriers**
  - Specification
  - **Implementation**

# Barrier Implementation

- First attempt at implementation

```
class Barrier {  
    private:  
        atomic_int counter;  
        int num_threads;  
    public:  
        Barrier(int num_threads) {  
            counter = 0;  
            this->num_threads = num_threads;  
        }  
  
        void barrier() {  
            // ??  
        }  
}
```



# Barrier Implementation

```
class Barrier {  
    private:  
        atomic_int counter;  
        int num_threads;  
    public:  
        Barrier(int num_threads) {  
            counter = 0;  
            this->num_threads = num_threads;  
        }  
  
        void barrier() {  
            int arrival_num = atomic_fetch_add(&counter, 1);  
            // What next?  
        }  
}
```

# Barrier Implementation

First handle the case where the thread is the last thread to arrive

```
class Barrier {
private:
    atomic_int counter;
    int num_threads;
public:
    Barrier(int num_threads) {
        counter = 0;
        this->num_threads = num_threads;
    }

    void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
            counter.store(0);
        }
        // What next?
    }
}
```

# Barrier Implementation

Spin while there  
is a thread waiting  
at the barrier

```
class Barrier {  
    private:  
        atomic_int counter;  
        int num_threads;  
    public:  
        Barrier(int num_threads) {  
            counter = 0;  
            this->num_threads = num_threads;  
        }  
  
        void barrier() {  
            int arrival_num = atomic_fetch_add(&counter, 1);  
            if (arrival_num == num_threads - 1) {  
                counter.store(0);  
            }  
            else {  
                while (counter.load() != 0);  
            }  
        }  
}
```

# Barrier Implementation

Spin while there  
is a thread waiting  
at the barrier

Does this work?

```
class Barrier {  
    private:  
        atomic_int counter;  
        int num_threads;  
    public:  
        Barrier(int num_threads) {  
            counter = 0;  
            this->num_threads = num_threads;  
        }  
  
        void barrier() {  
            int arrival_num = atomic_fetch_add(&counter, 1);  
            if (arrival_num == num_threads - 1) {  
                counter.store(0);  
            }  
            else {  
                while (counter.load() != 0);  
            }  
        }  
}
```

**Thread 0:**

```
B.barrier();  
B.barrier();
```

```
void barrier() {  
    int arrival_num = atomic_fetch_add(&counter, 1);  
    if (arrival_num == num_threads) {  
        counter.store(0);  
    }  
    else {  
        while (counter.load() != 0);  
    }  
}
```

**Thread 1:**

```
B.barrier();  
B.barrier();
```

thread 0 →

thread 1 →

num\_threads == 2

Thread 0:

B.barrier();

B.barrier();

```
void barrier() {  
    int arrival_num = atomic_fetch_add(&counter, 1);  
    if (arrival_num == num_threads - 1) {  
        counter.store(0);  
    }  
    else {  
        while (counter.load() != 0);  
    }  
}
```

Thread 1:

B.barrier();

B.barrier();

arrival\_num = 1

arrival\_num = 0

thread 0 →

thread 1 →

```
num_threads == 2
counter == 2
```

Thread 0:

B.barrier();

B.barrier();

```
void barrier() {
    int arrival_num = atomic_fetch_add(&counter, 1);
    if (arrival_num == num_threads - 1) {
        counter.store(0);
    }
    else {
        while (counter.load() != 0);
    }
}
```

Thread 1:

B.barrier();

B.barrier();

arrival\_num = 1

arrival\_num = 0

thread 0 →

thread 1 →

```
num_threads == 2
counter == 0
```

Thread 0:

**B.barrier();**

B.barrier();

```
void barrier() {
    int arrival_num = atomic_fetch_add(&counter, 1);
    if (arrival_num == num_threads - 1) {
        counter.store(0);
    }
    else {
        while (counter.load() != 0);
    }
}
```

Thread 1:

**B.barrier();**

B.barrier();

arrival\_num = 1

arrival\_num = 0

thread 0 →

thread 1 →



```
num_threads == 2
counter == 0
```

### Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
    int arrival_num = atomic_fetch_add(&counter, 1);
    if (arrival_num == num_threads - 1) {
        counter.store(0);
    }
    else {
        while (counter.load() != 0);
    }
}
```

### Thread 1:

```
B.barrier();
B.barrier();
```

Leaves barrier

arrival\_num = 0

in a perfect world,  
thread 1 executes now and leaves the barrier

thread 0 →

thread 1 →

```
num_threads == 2  
counter == 0
```

### Thread 0:

```
B.barrier();  
B.barrier();
```

```
void barrier() {  
    int arrival_num = atomic_fetch_add(&counter, 1);  
    if (arrival_num == num_threads - 1) {  
        counter.store(0);  
    }  
    else {  
        while (counter.load() != 0);  
    }  
}
```



Leaves barrier

### Thread 1:

```
B.barrier();  
B.barrier();
```

arrival\_num = 0

in a perfect world,  
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it  
was asleep?

```
num_threads == 2  
counter == 0
```

Thread 0:

```
B.barrier();
```

```
B.barrier();
```

```
void barrier() {  
    int arrival_num = atomic_fetch_add(&counter, 1);  
    if (arrival_num == num_threads - 1) {  
        counter.store(0);  
    }  
    else {  
        while (counter.load() != 0);  
    }  
}
```



enters next barrier

Thread 1:

```
B.barrier();
```

```
B.barrier();
```

arrival\_num = 0

in a perfect world,  
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it  
was asleep?

```
num_threads == 2  
counter == 1
```

Thread 0:

```
B.barrier();  
B.barrier();
```

```
void barrier() {  
    int arrival_num = atomic_fetch_add(&counter, 1);  
    if (arrival_num == num_threads - 1) {  
        counter.store(0);  
    }  
    else {  
        while (counter.load() != 0);  
    }  
}
```



arrival\_num == 0

Thread 1:

```
B.barrier();  
B.barrier();
```

arrival\_num = 0

in a perfect world,  
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it  
was asleep?

```
num_threads == 2  
counter == 1
```

Thread 0:

```
B.barrier();  
B.barrier();
```

```
void barrier() {  
    int arrival_num = atomic_fetch_add(&counter, 1);  
    if (arrival_num == num_threads - 1) {  
        counter.store(0);  
    }  
    else {  
        while (counter.load() != 0);  
    }  
}
```

*Thread 1 wakes up! Doesn't think its missed anything*

arrival\_num == 0

Thread 1:

```
B.barrier();  
B.barrier();
```

arrival\_num = 0

in a perfect world,  
thread 1 executes now and leaves the barrier

```
num_threads == 2
counter == 1
```

Thread 0:

```
B.barrier();
```

```
B.barrier();
```

```
void barrier() {
    int arrival_num = atomic_fetch_add(&counter, 1);
    if (arrival_num == num_threads - 1) {
        counter.store(0);
    }
    else {
        while (counter.load() != 0);
    }
}
```

Thread 1:

```
B.barrier();
```

```
B.barrier();
```

*Thread 1 wakes up! Doesn't think its missed anything*

arrival\_num == 0

arrival\_num = 0

in a perfect world,  
thread 1 executes now and leaves the barrier

Both threads get stuck here!