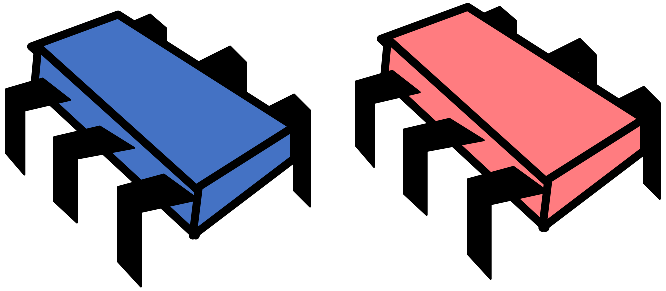
CSE113: Parallel Programming

Feb. 27, 2023

- **Topics**:
  - General concurrent sets



Announcements

- Midterm grades should be released
  - Let us know within 1 week if there are any issues
- Expect HW 2 grades by the end of the week
- HW 3 is due on Wednesday
  - Two additional late days provided because of the storm
- HW 4 is released on Wednesday
  - Should have enough material to get started

Announcements

- Last day on concurrent data structures module!

- Moving to reasoning about concurrency on Wednesday

Previous quiz

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.

Schedule

- Parallelizing DOALL loops
- How atomics are implemented in hardware
- Lock-free concurrent set

Practical Parallel DOALL Loops

- Languages have various features to enable easy and flexible parallel DOALL Loops

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

Iterateble-object

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

Higher order function for iterating over object

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                  //do stuff with item
              });
```

Execution policy types

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

more in a few slides!

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

Iterator range

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                  cout << item << endl;
              });
```

Functor or Lambda:
Execute the function
with each item in the iterated
range

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                  //do stuff with item
              });
```

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

Difference between these two?

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                  //do stuff with item
              });
```

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::float> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                    tmp += 1.0;
                    tmp += 2.0;
                    tmp += 3.0;
                    ...
              });
```

what would we like to do here?

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

C++

```
std::vector<std::int> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                    tmp += 1.0;
                    tmp += 2.0;
                    tmp += 3.0;
                    ...
              });
```

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

what would we like to do here?

```
tmp0 += 1.0;  // for item0
tmp1 += 1.0;  // for item1
tmp2 += 1.0;  // for item2
....
```

Just like in HW 1!

par_unseq requires that instructions in loops can interleaved!

C++

```
std::vector<std::int> foo;
std::for_each(std::execution::par,
              foo.begin(), foo.end(),
              [](auto& item) {
                 tyler_account += item
              });
```

global variable account, now we'd have a data race!

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

C++

```
std::vector<std::int> foo;
std::mutex m;
std::for_each(std::execution::par,
              foo.begin(), foo.end(),
              [](auto& item) {
                 m.lock();
                 tyler_account += item
                 m.unlock();
              });
```

We can fix it with mutexes

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

C++

```
std::vector<std::int> foo;
std::mutex m;
std::for_each(std::execution::par,
              foo.begin(), foo.end(),
              [](auto& item) {
                 m.lock();
                 tyler_account += item
                 m.unlock();
              });
```

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

But now we can't interleave

deadlock!

```
m.lock(); // for item 0
m.lock(); // for item 1
tyler_account += item0;
tyler_account += item1;
```

We need to use `std::execution::par` if iterations cannot be interleaved (e.g. if they use mutexes)

C++ shortcomings

- Have to modify code
- No control over the parallel schedule

OpenMP

- Pragma based extension to C/C++/Fortran

```
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
```

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
// add -fopenmp to compile line
```

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
// add -fopenmp to compile line
```

launches threads to perform loop in parallel. Joins threads afterward

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
// add -fopenmp to compile line
```

if its so easy, why don’t compilers just do this for us automatically?

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
// add -fopenmp to compile line
```

Performance considerations:
when is parallelism going to provide a speedup vs. slowdown?

Correctness considerations:
very difficult to determine if loop is safe to do in parallel

if its so easy, why don't compilers just do this for us automatically?

OpenMP

- Pragma based extension to C/C++/Fortran

What about irregular loops?

```
for (x = 0; x < SIZE; x++) {
  for (y = x; y < SIZE; y++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for schedule(dynamic)
for (x = 0; x < SIZE; x++) {
  for (y = x; y < SIZE; y++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

What about irregular loops?

Schedule keyword

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for schedule(dynamic)
for (x = 0; x < SIZE; x++) {
  for (y = x; y < SIZE; y++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

What about irregular loops?

Schedule keyword

different types of schedules

OpenMP

- Schedules:
  - From http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

`schedule(static, chunk-size)`

```
schedule(static):
****************
                ****************
                                ****************
                                                ****************
```

```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

```
schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

from: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html `schedule(dynamic, chunk-size)`

```
schedule(dynamic, 1):
    *    *     *        *   *    * *  *  *         *  * *  * *
*  *  *   * *     *  * * *    * *      *   ***  *   *         *
 *   *  *  *  *    ** *    *      *  *  * *   *  *   *   *
  *    *     *  **        *  * *    *          *  *    *  * *  *
```

```
schedule(dynamic, 4):
            ****                    ****                    ****
****            ****    ****            ****        ****
    ****            ****    ****            ****        ****
        ****                    ****            ****
```

```
schedule(dynamic, 8):
                ********                                ********
                        ********        ********
********                        ********        ********
        ********
```

from: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Schedule

- Parallelizing DOALL loops

- How atomics are implemented in hardware

- Lock-free concurrent set

How is CAS (and others) implemented?

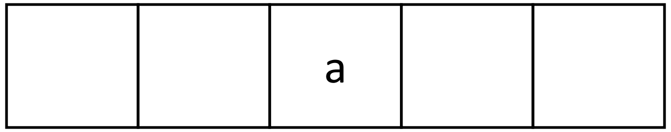
- X86 has an actual instruction
- ARM and POWER are load linked store conditional

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

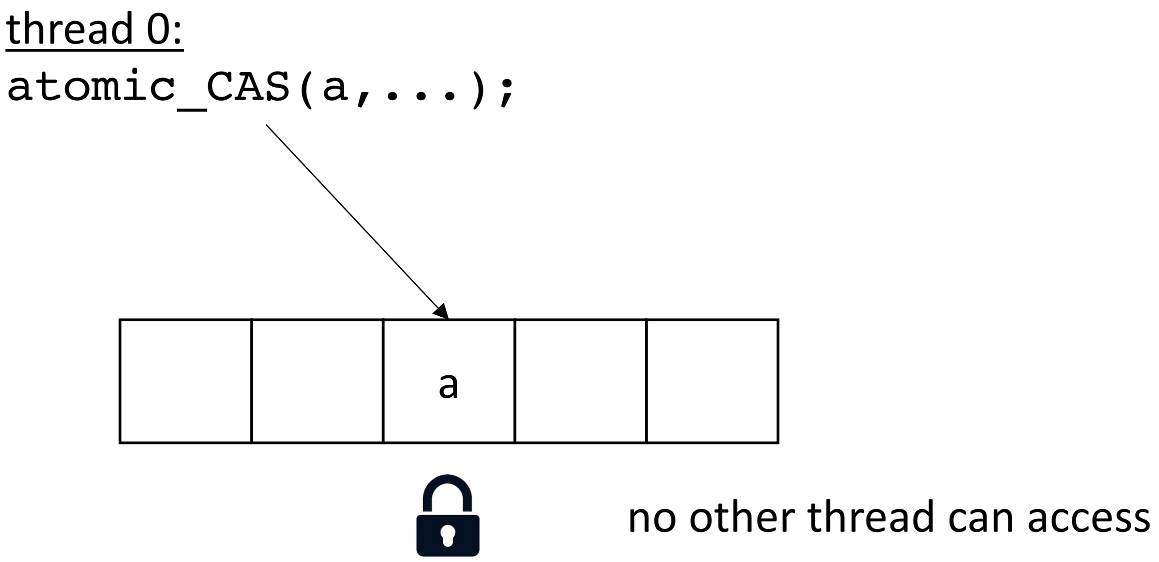
<u>thread 0:</u>

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



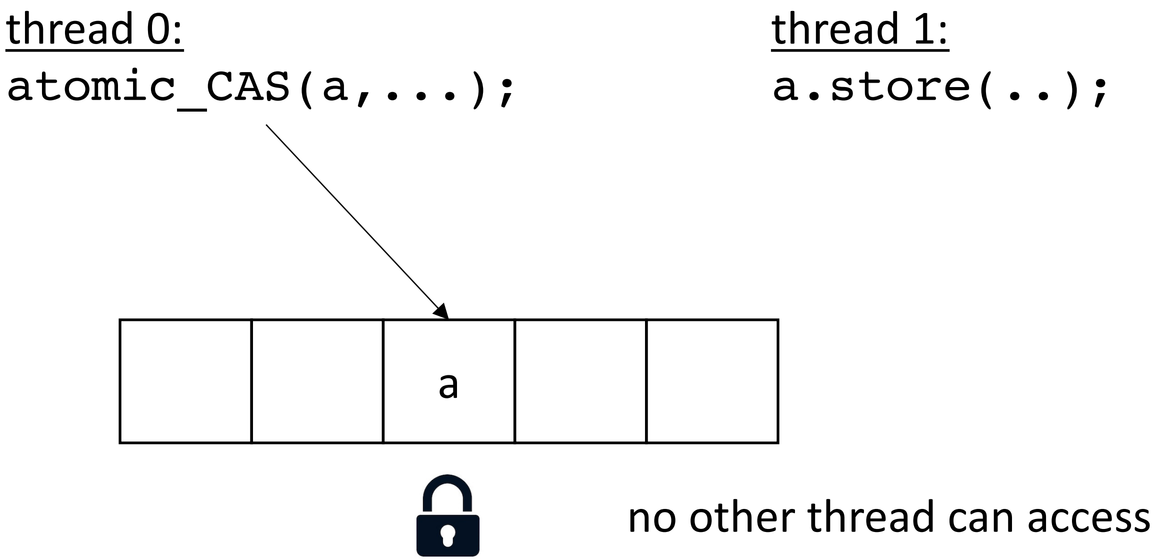
Pessimistic Concurrency

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



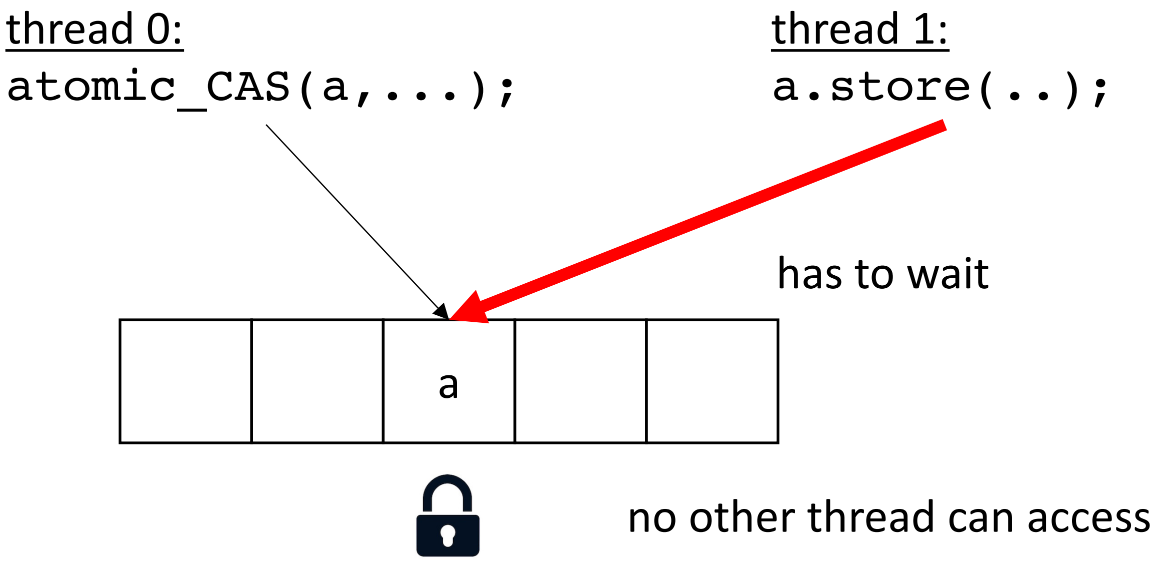
Pessimistic Concurrency

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



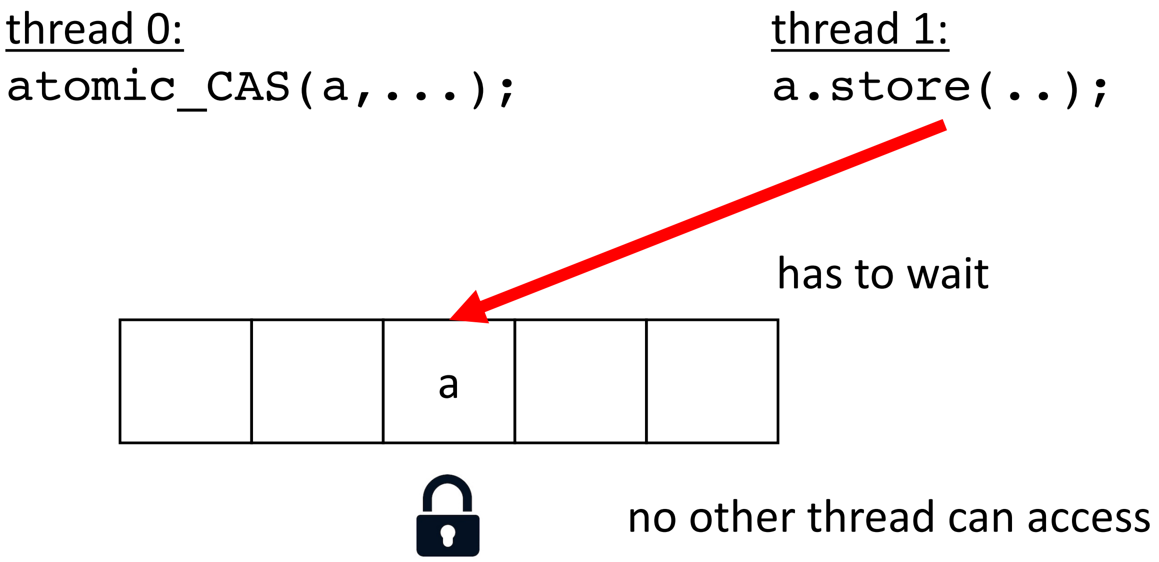
Pessimistic Concurrency

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



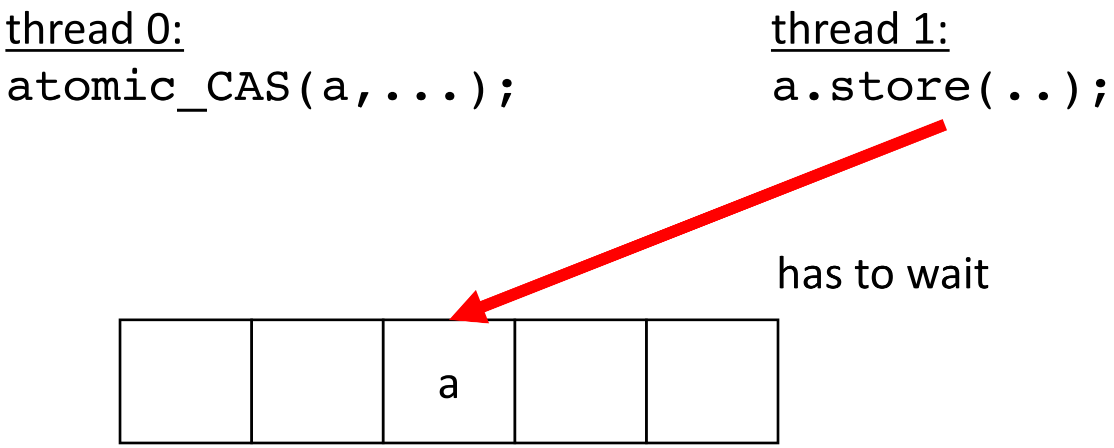
Pessimistic Concurrency

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



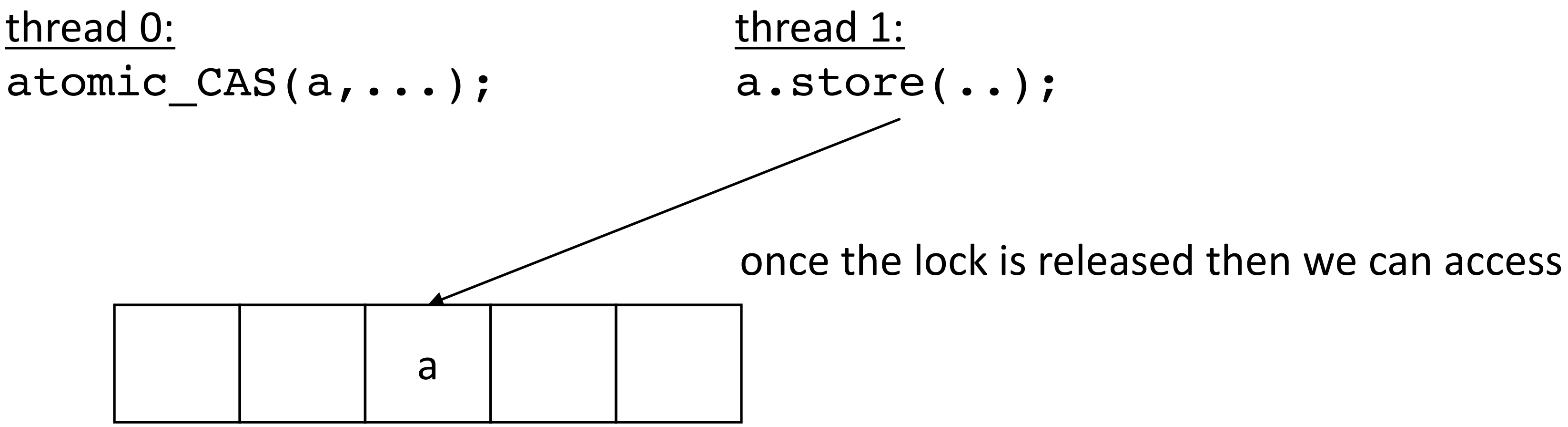
Pessimistic Concurrency

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



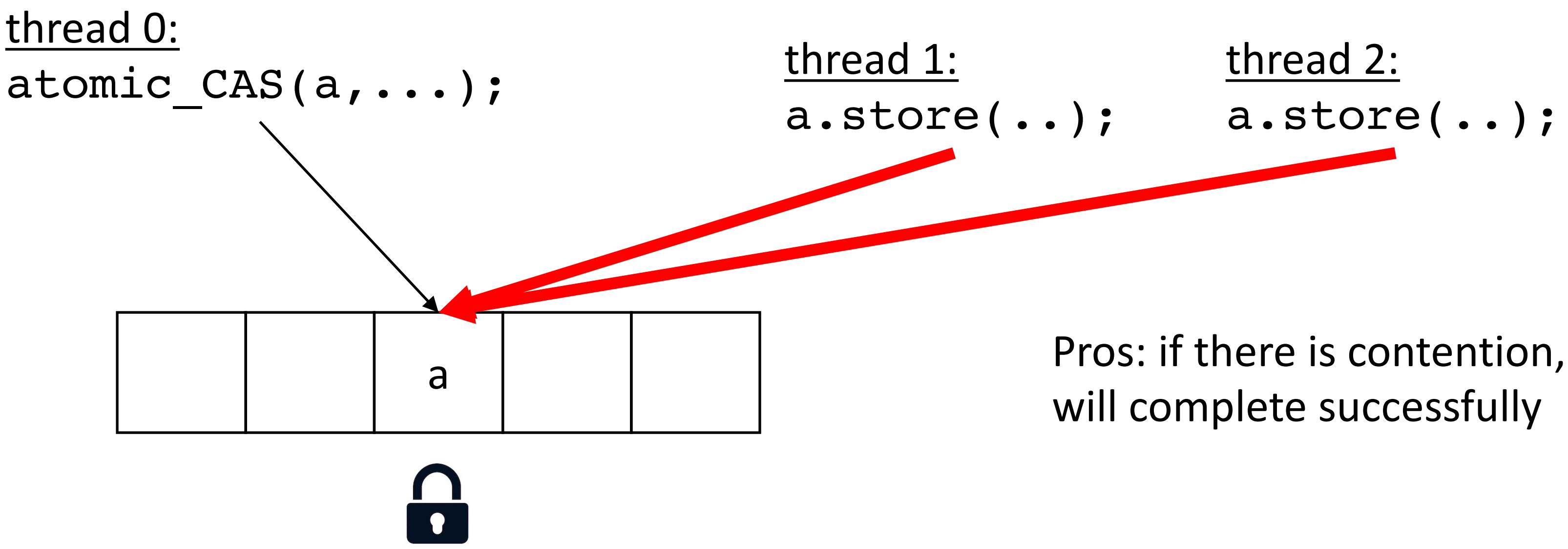
Pessimistic Concurrency

- 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

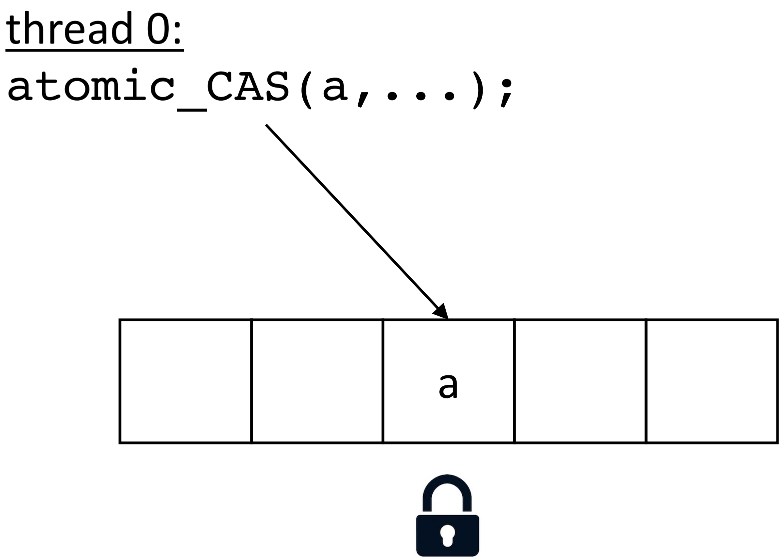
- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



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

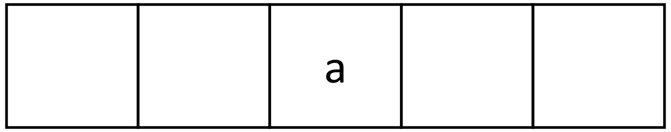
Optimistic Concurrency

- 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

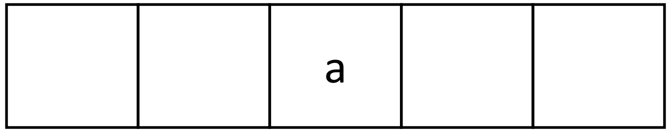


Optimistic Concurrency

- 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);
```



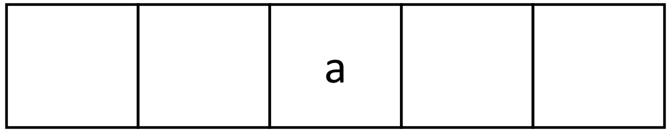
T0_exclusive = 1

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

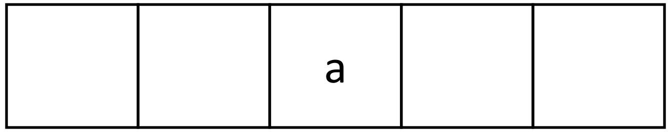
Optimistic Concurrency

- 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

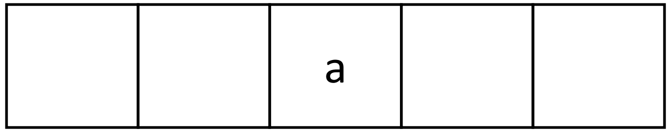
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

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

thread 1:

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



Optimistic Concurrency

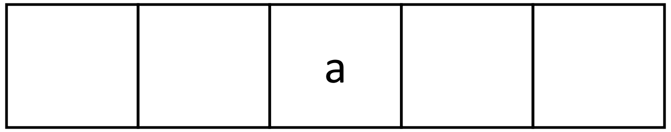
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

<u>thread 0:</u>

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

<u>thread 1:</u>

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



T0_exclusive = 1

Optimistic Concurrency

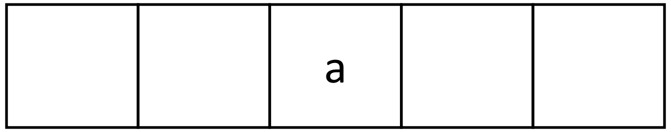
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

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

thread 1:

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



T0_exclusive = 1

Optimistic Concurrency

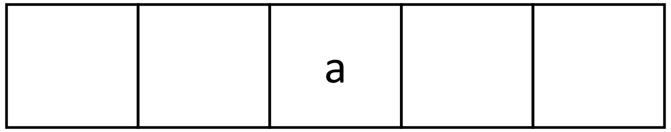
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

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

thread 1:

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



T0_exclusive = 0

Optimistic Concurrency

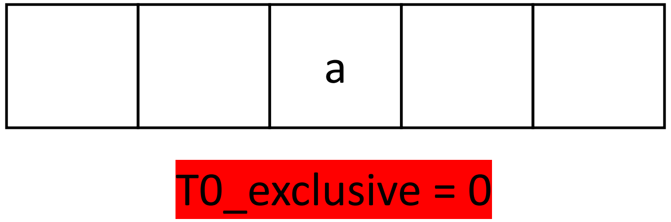
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

<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.

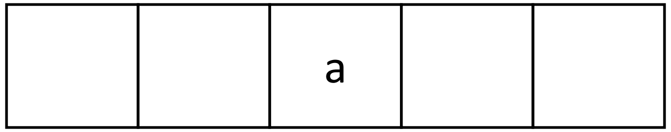
thread 0:

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

thread 1:

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

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



T0_exclusive = 0

Optimistic Concurrency

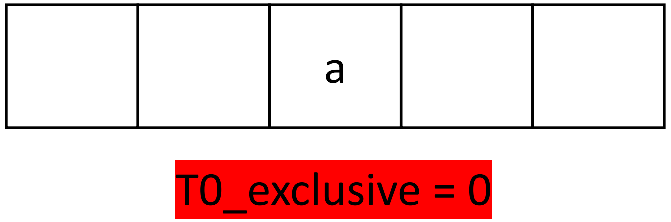
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

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!

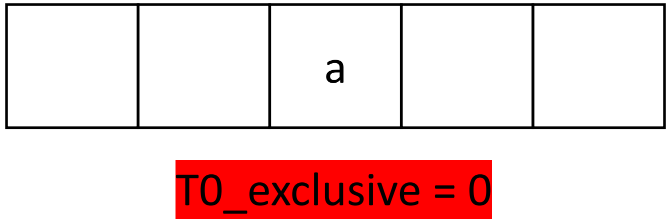
solution: loop until success:

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));
```

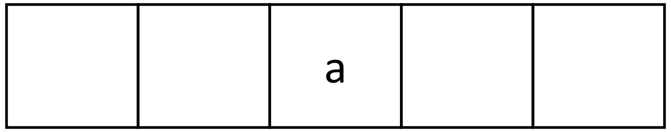


Optimistic Concurrency

- 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));
```



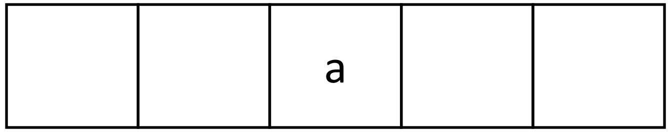
T0_exclusive = 1

Optimistic Concurrency

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

thread 0:

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



T0_exclusive = 1

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.

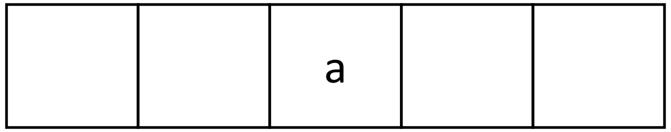
Optimistic Concurrency

- 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

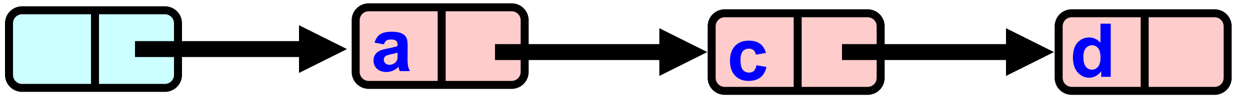
- Parallelizing DOALL loops

- How atomics are implemented in hardware

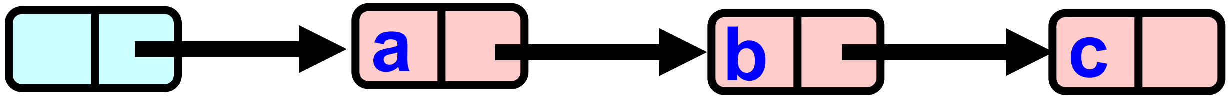
- Lock-free concurrent set

Sequential List Based Set

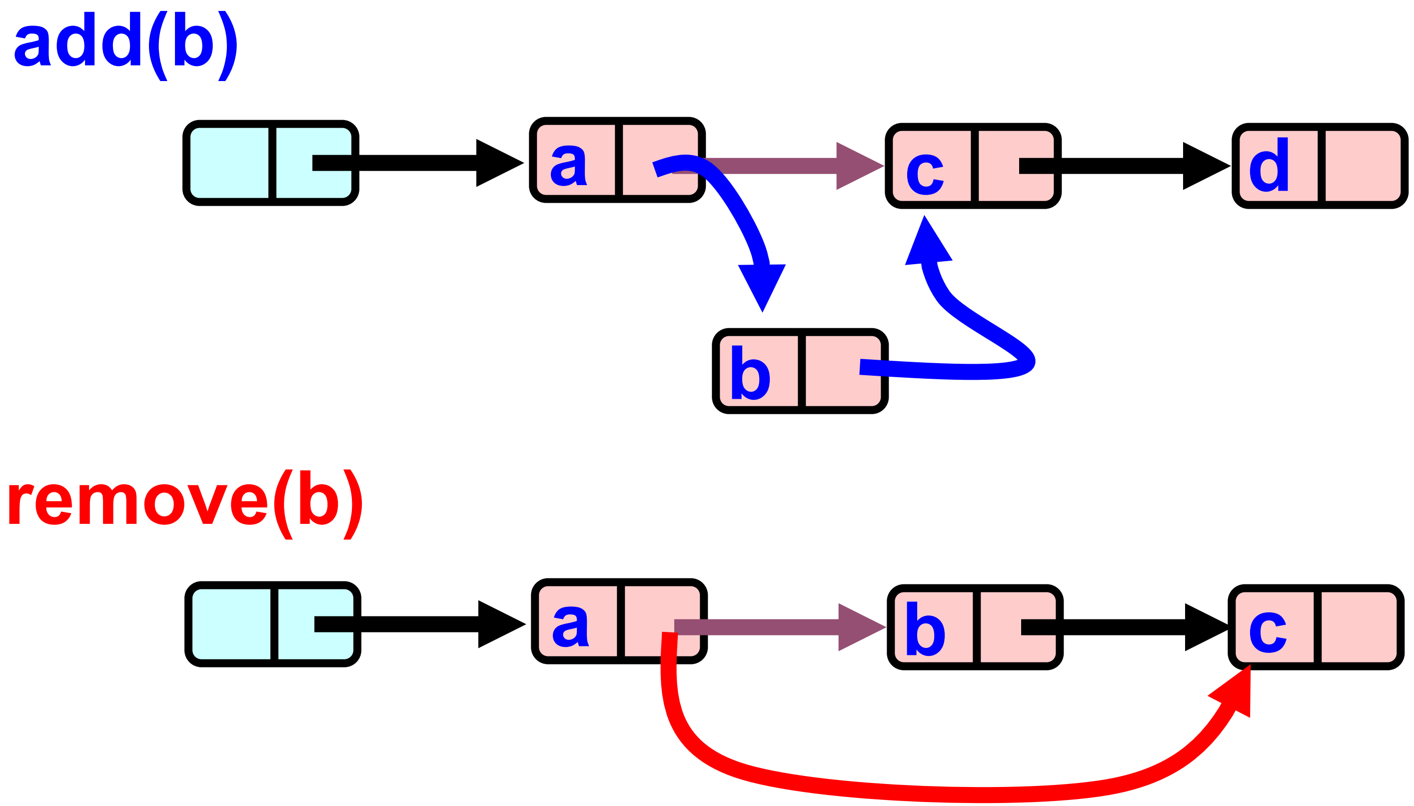
add(b)



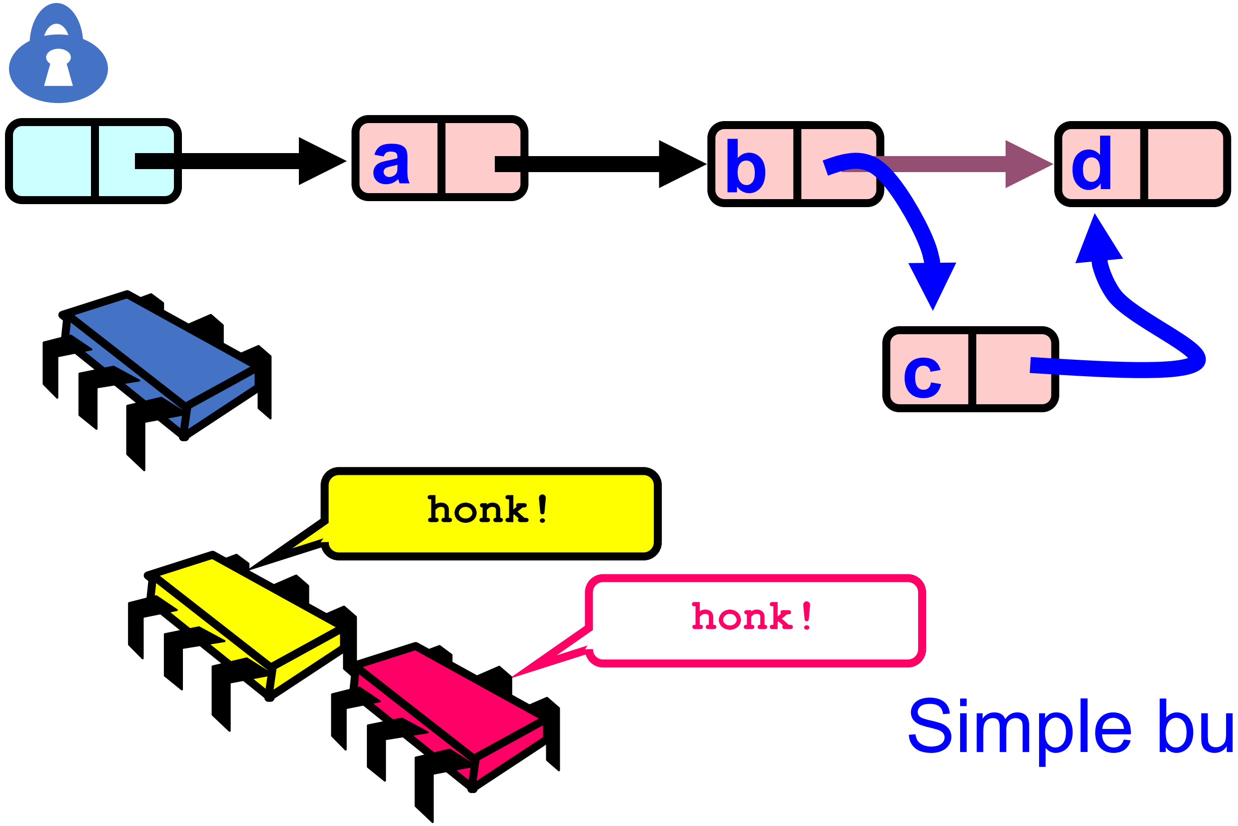
remove(b)



Sequential List Based Set

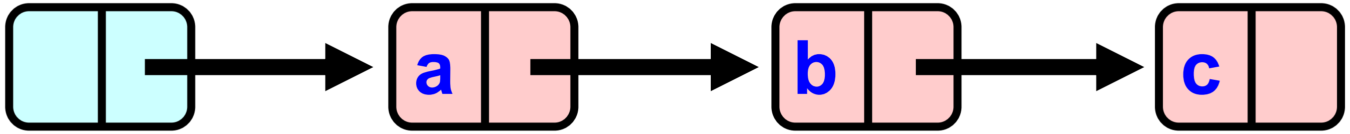


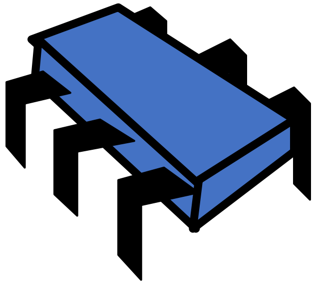
Coarse-Grained Locking



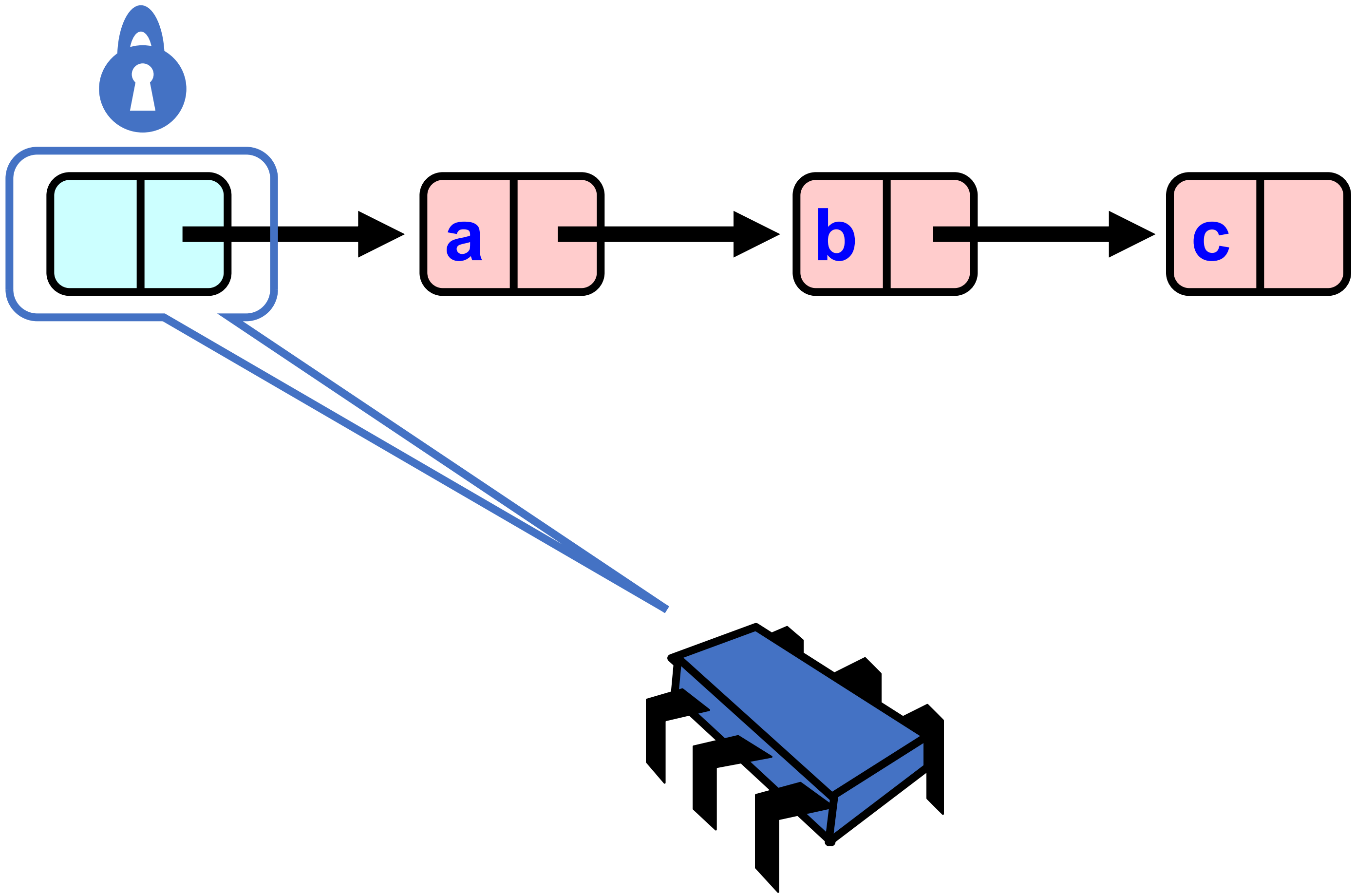
Simple but inefficient!

Hand-over-Hand locking

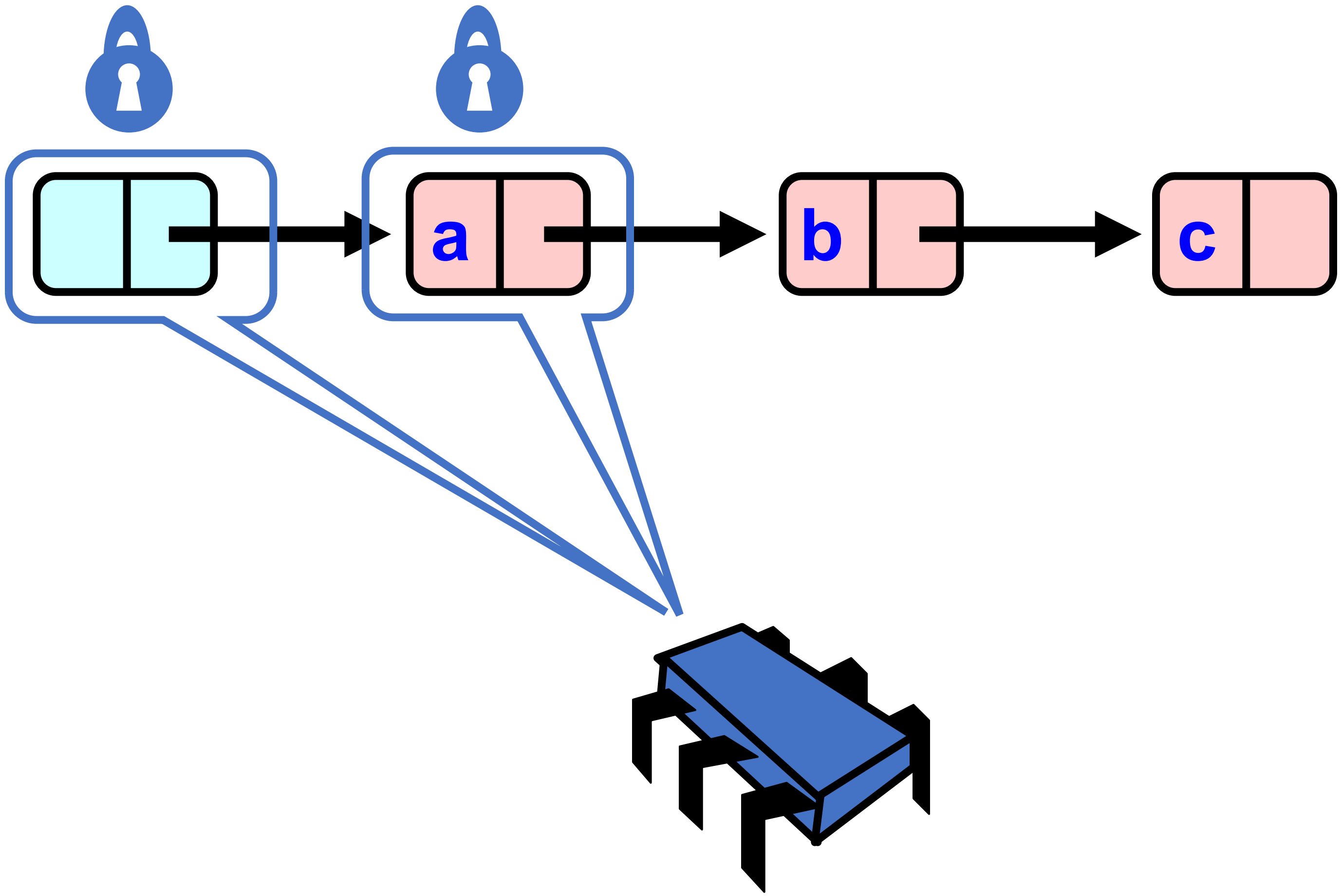




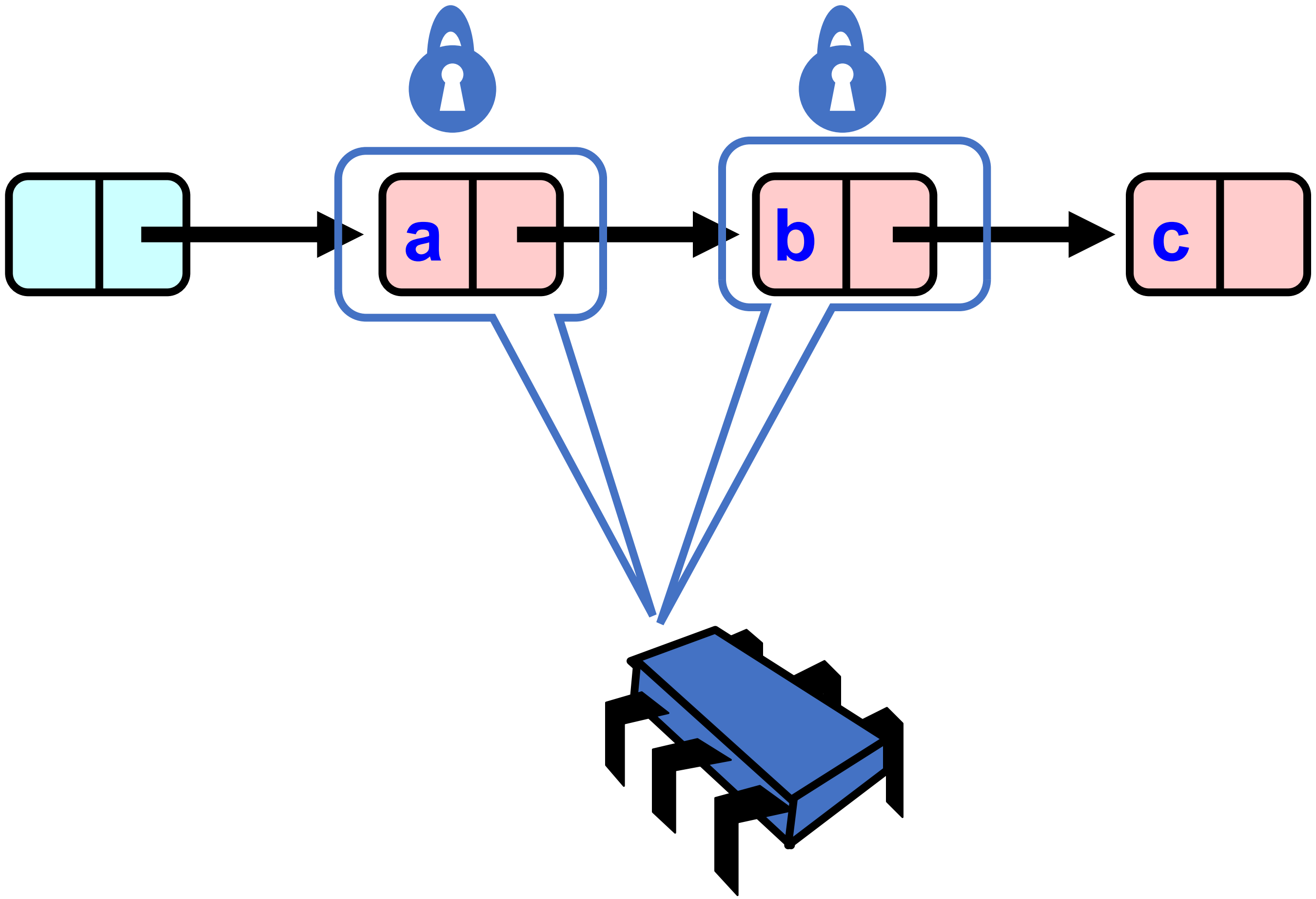
Hand-over-Hand locking



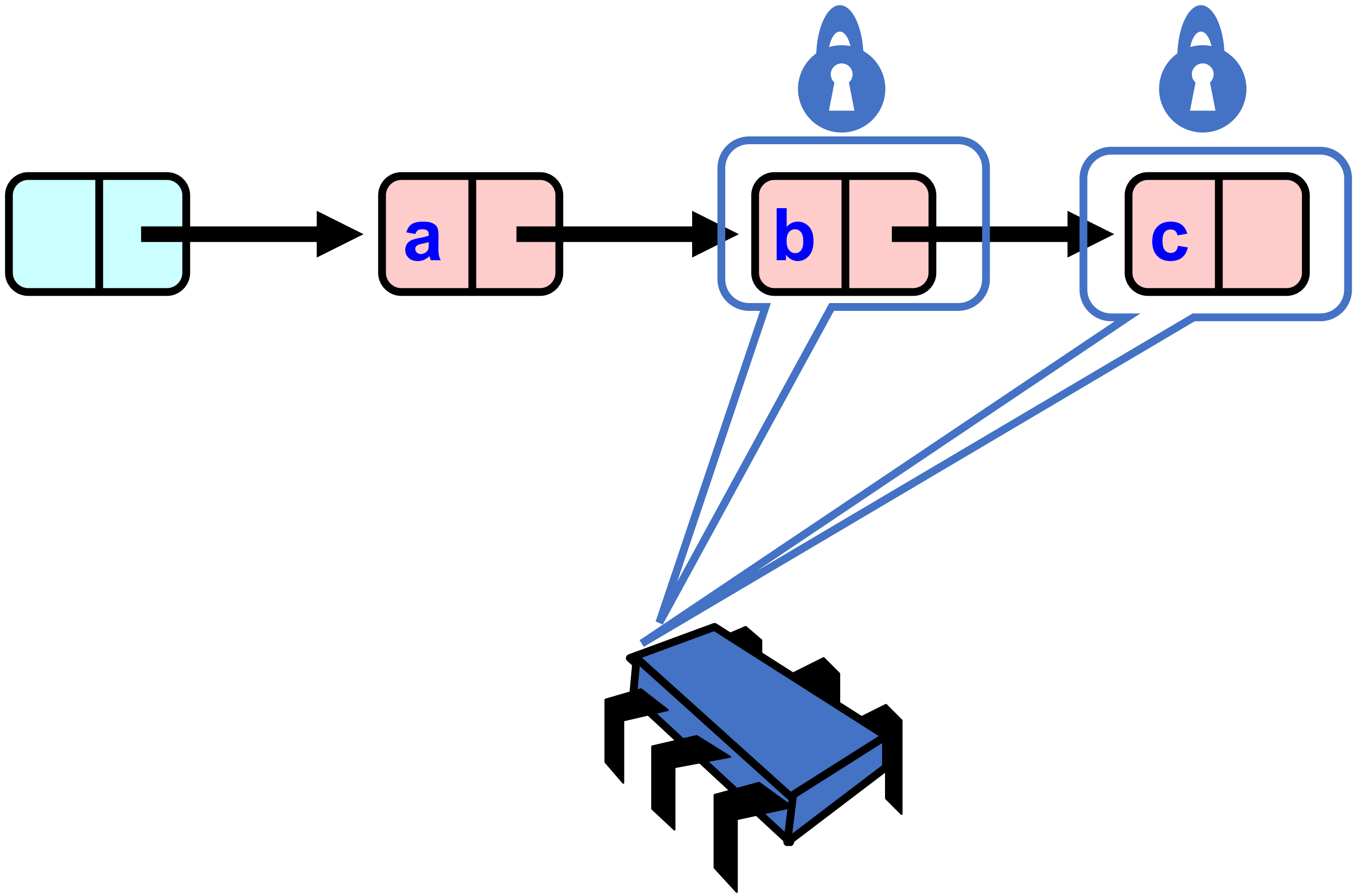
Hand-over-Hand locking



Hand-over-Hand locking

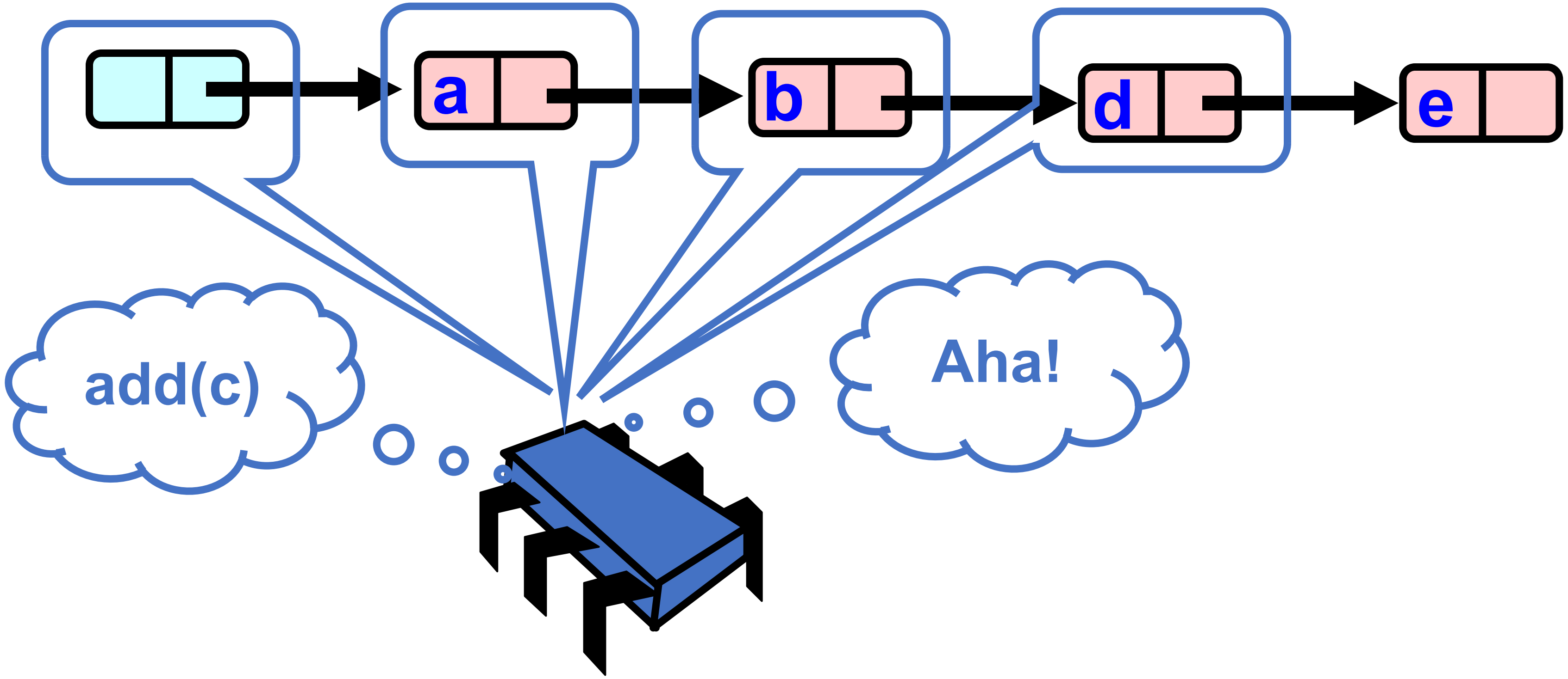


Hand-over-Hand locking

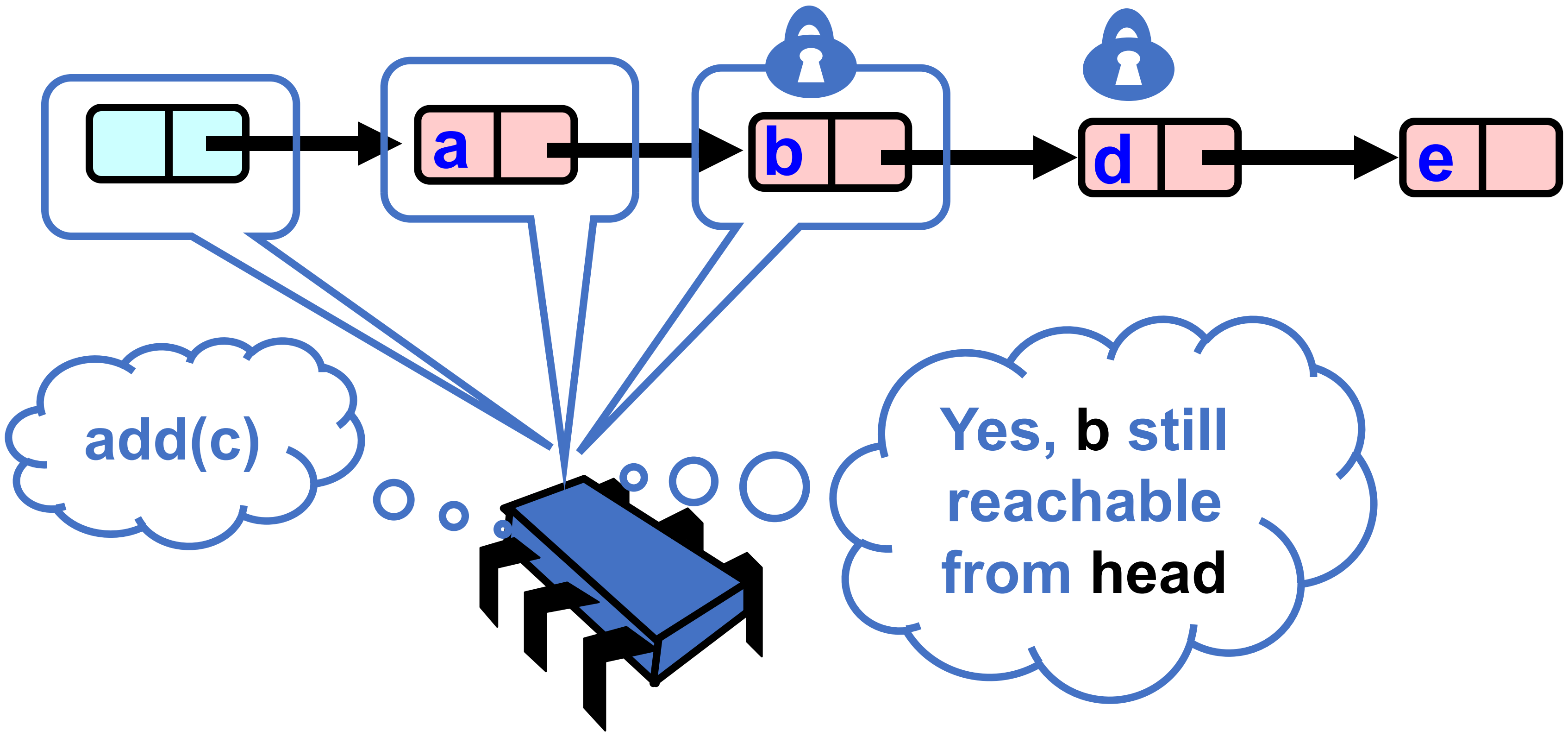


Optimistic traversals

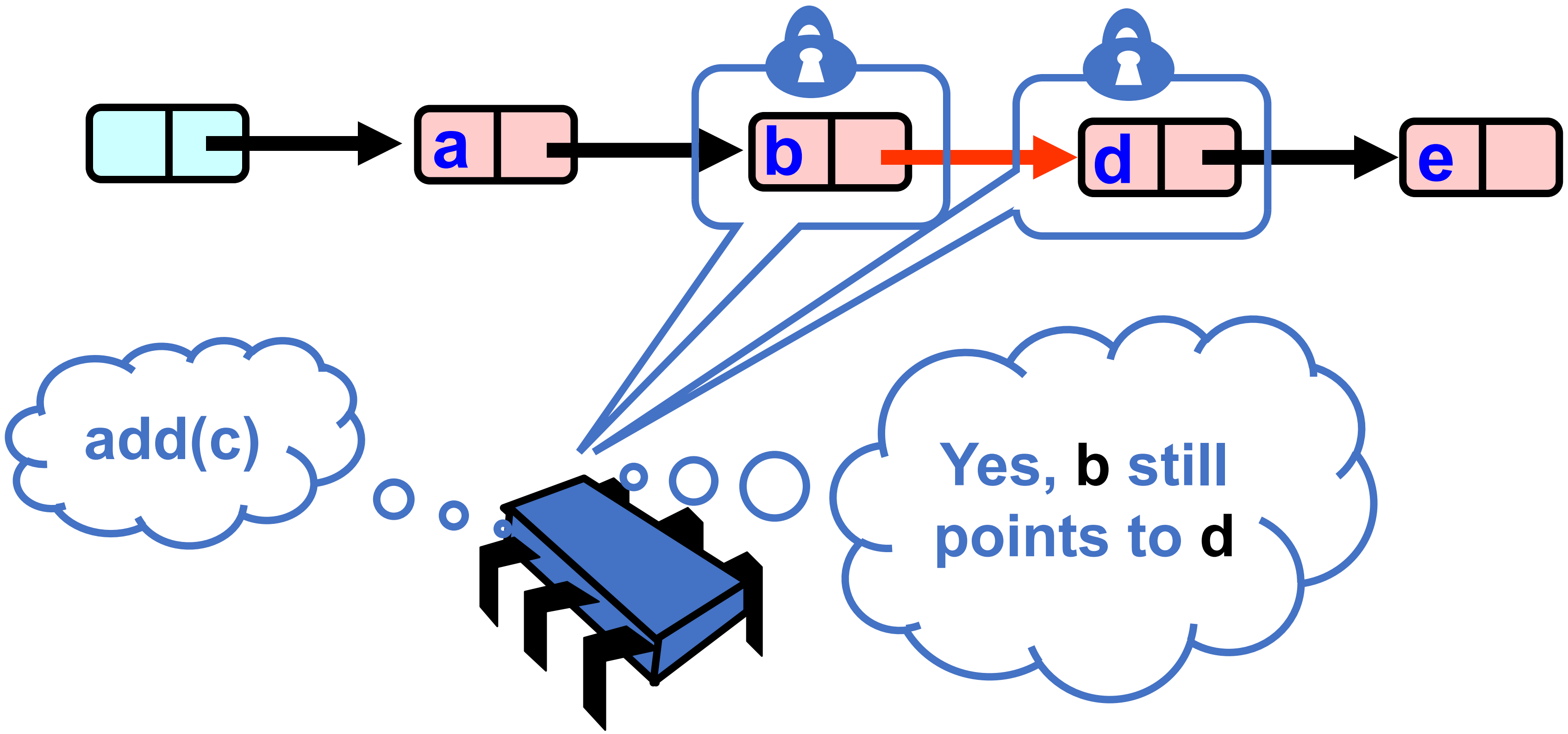
What could go wrong?



Validate – Part 1



Validate Part 2
(while holding locks)



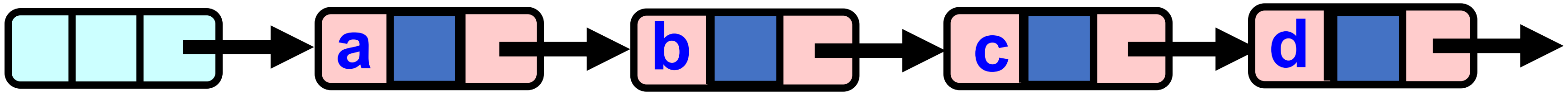
Can we optimize more?

- Scan the list once?

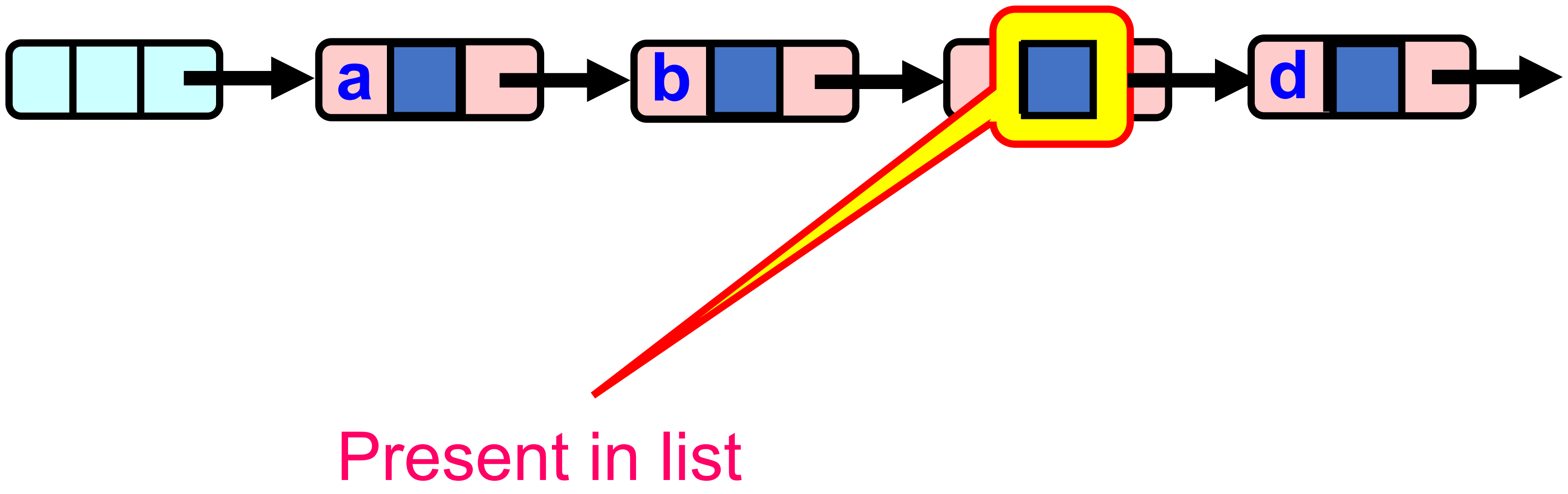
Two step removal List

- **`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)

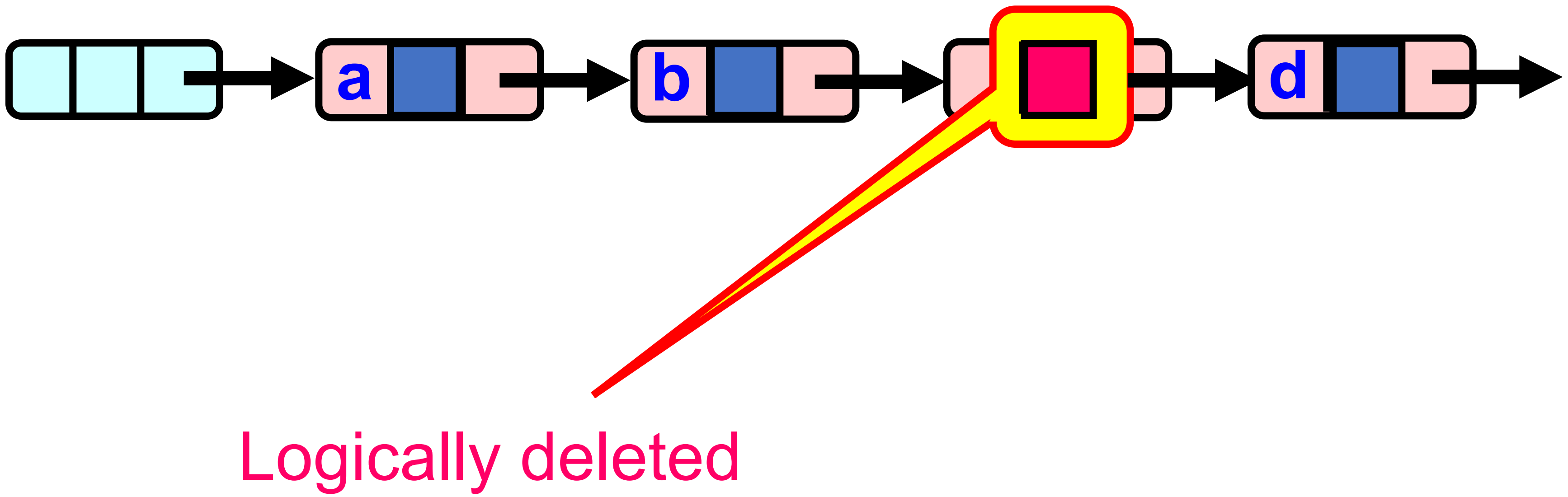
Two step removal Removal



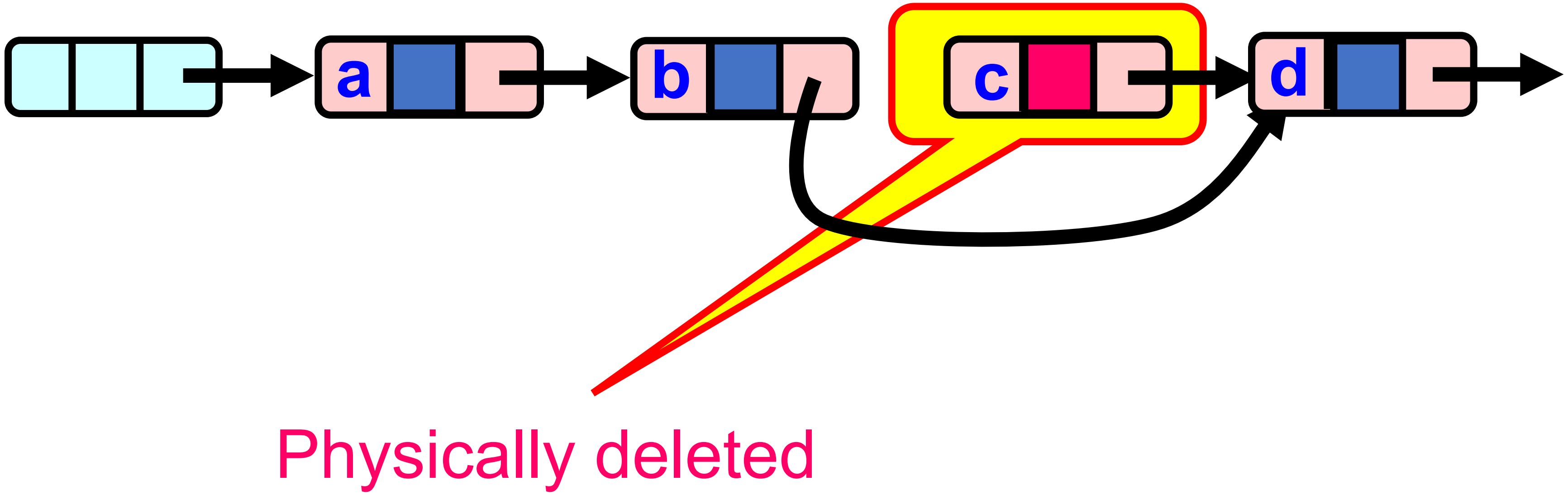
Two step removal Removal



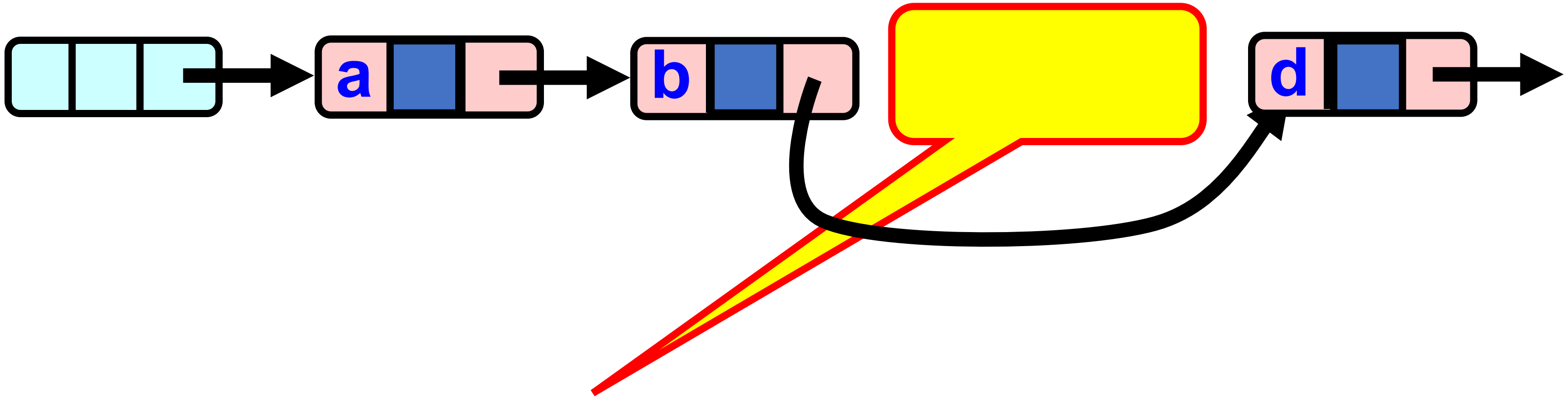
Two step removal Removal



Two step removal Removal



Two step removal Removal



Physically deleted

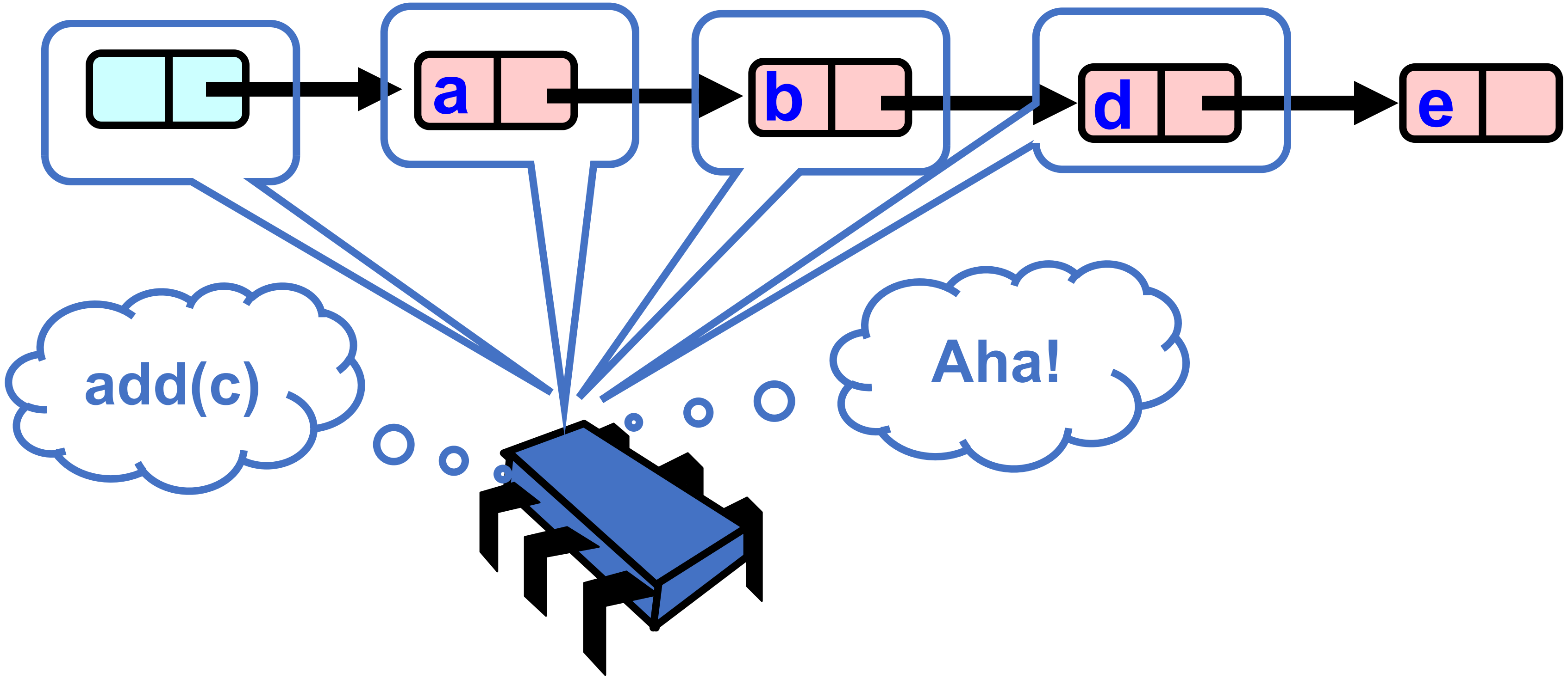
Two step remove list

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

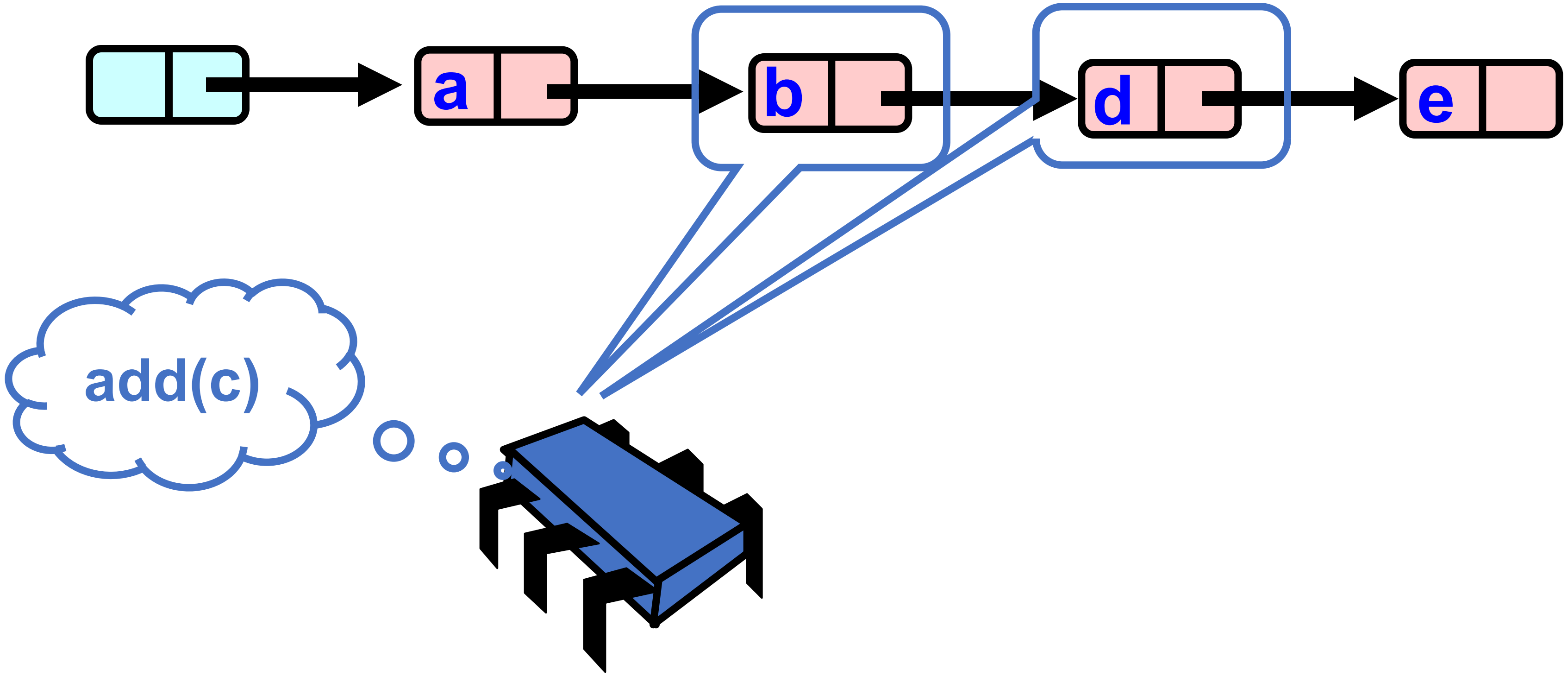
Validation

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

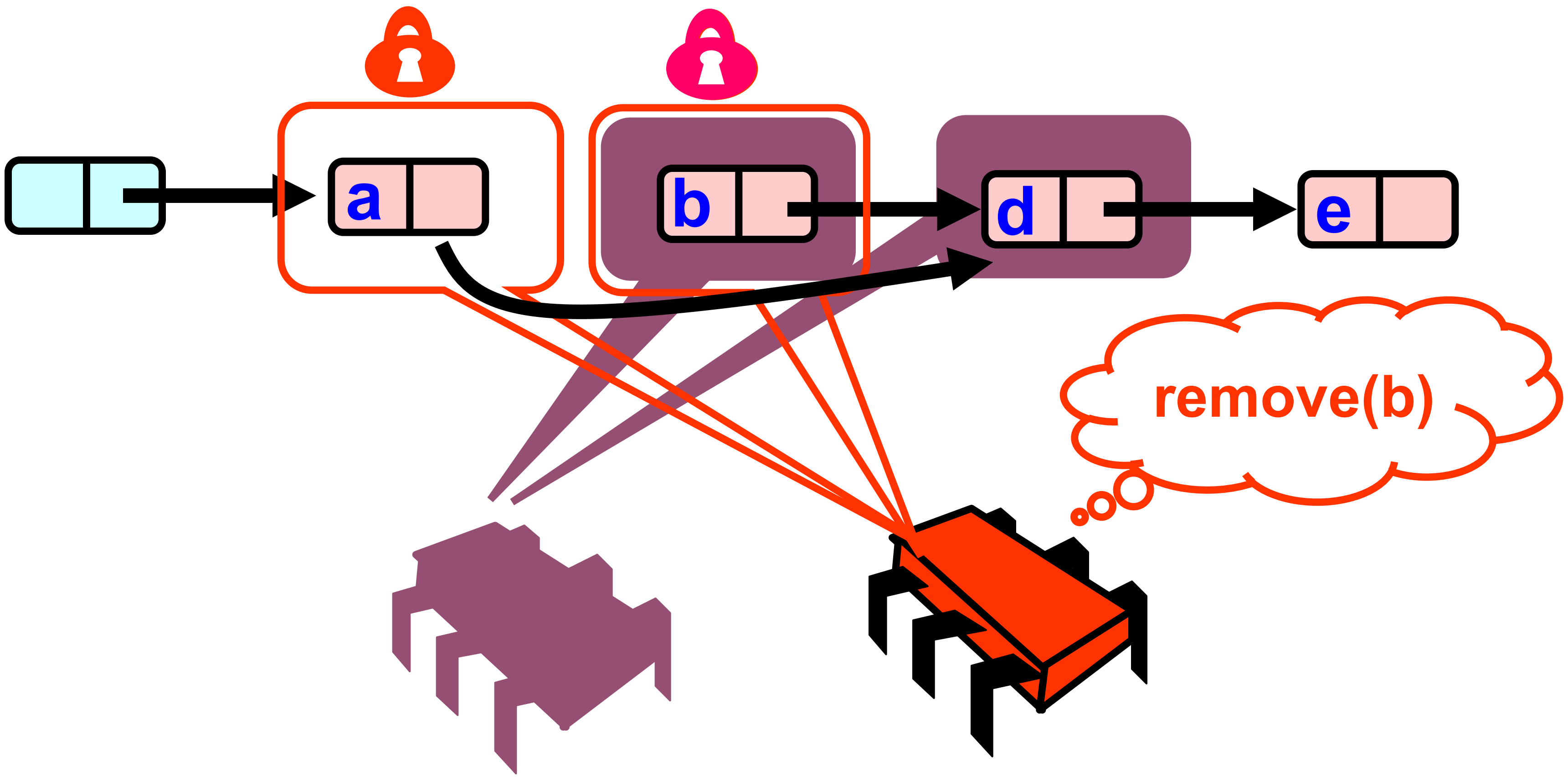
What could go wrong?



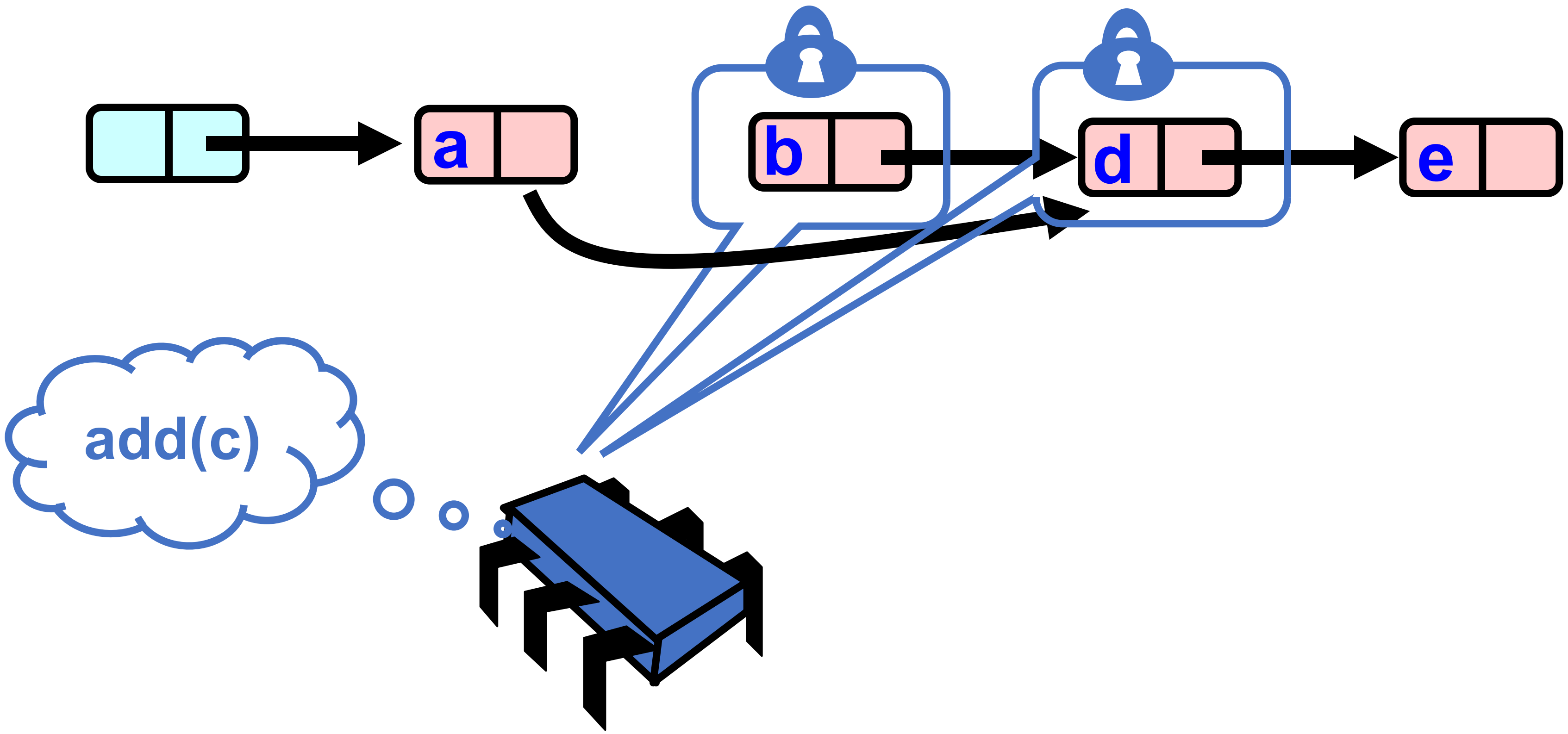
What could go wrong?



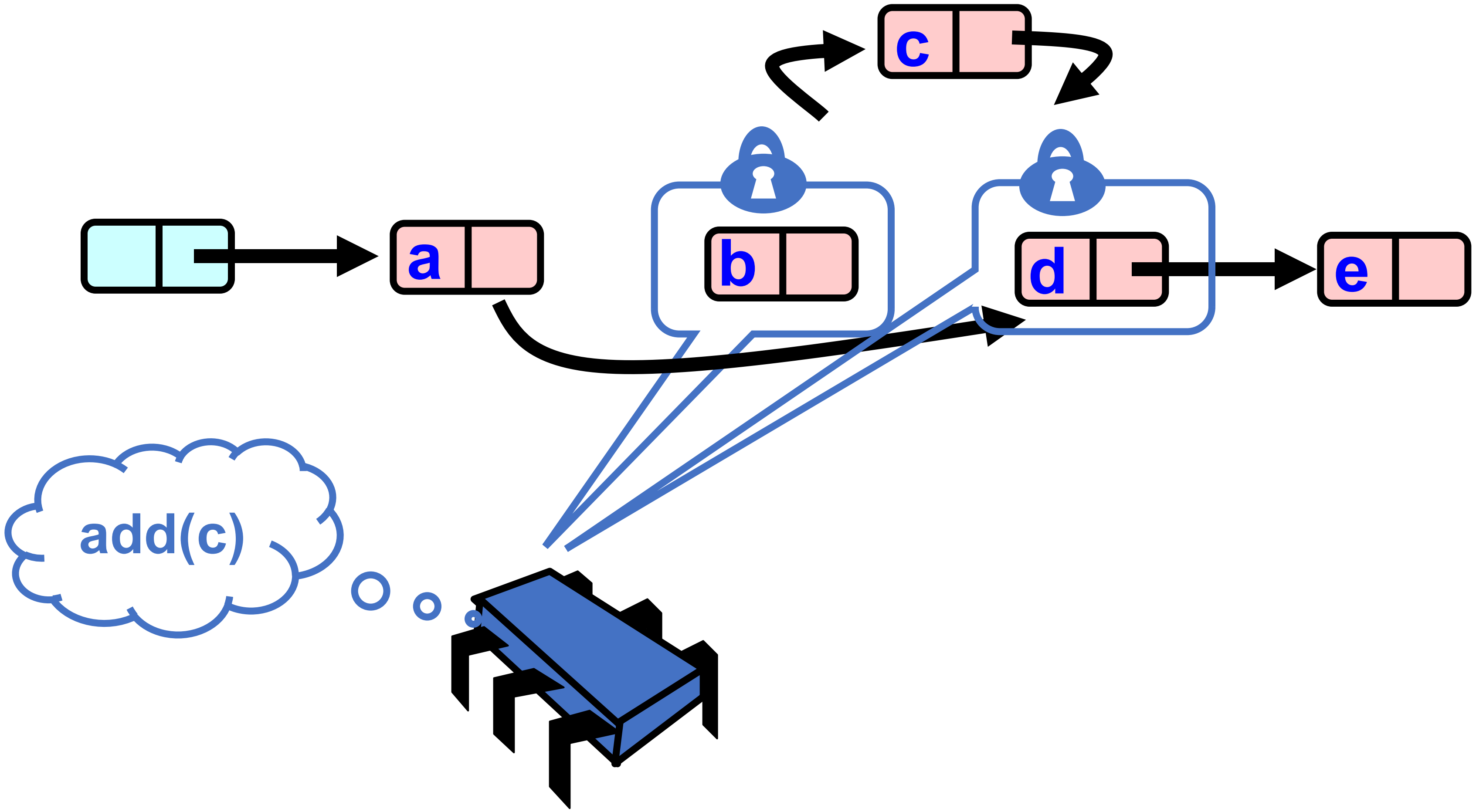
What could go wrong?



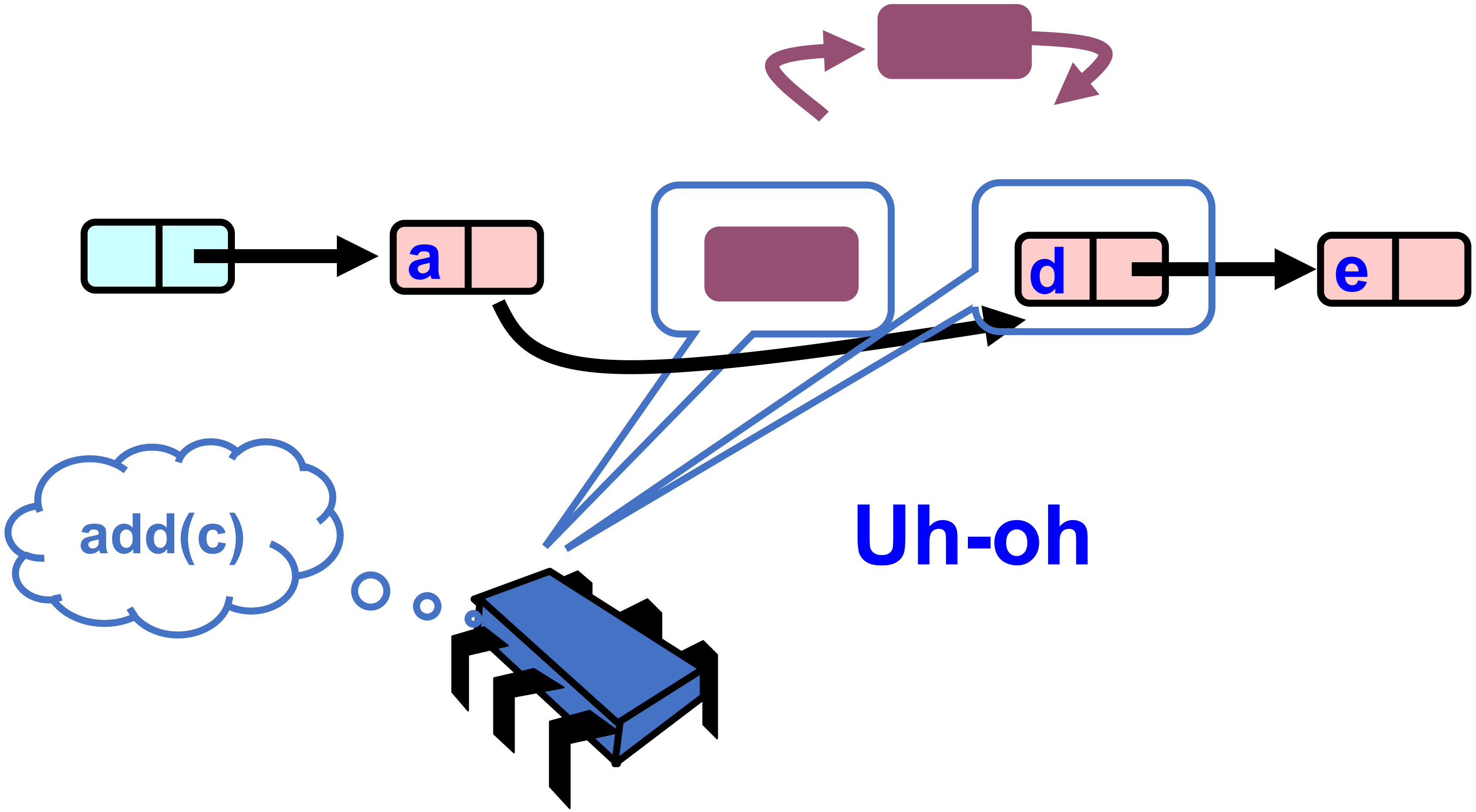
What could go wrong?



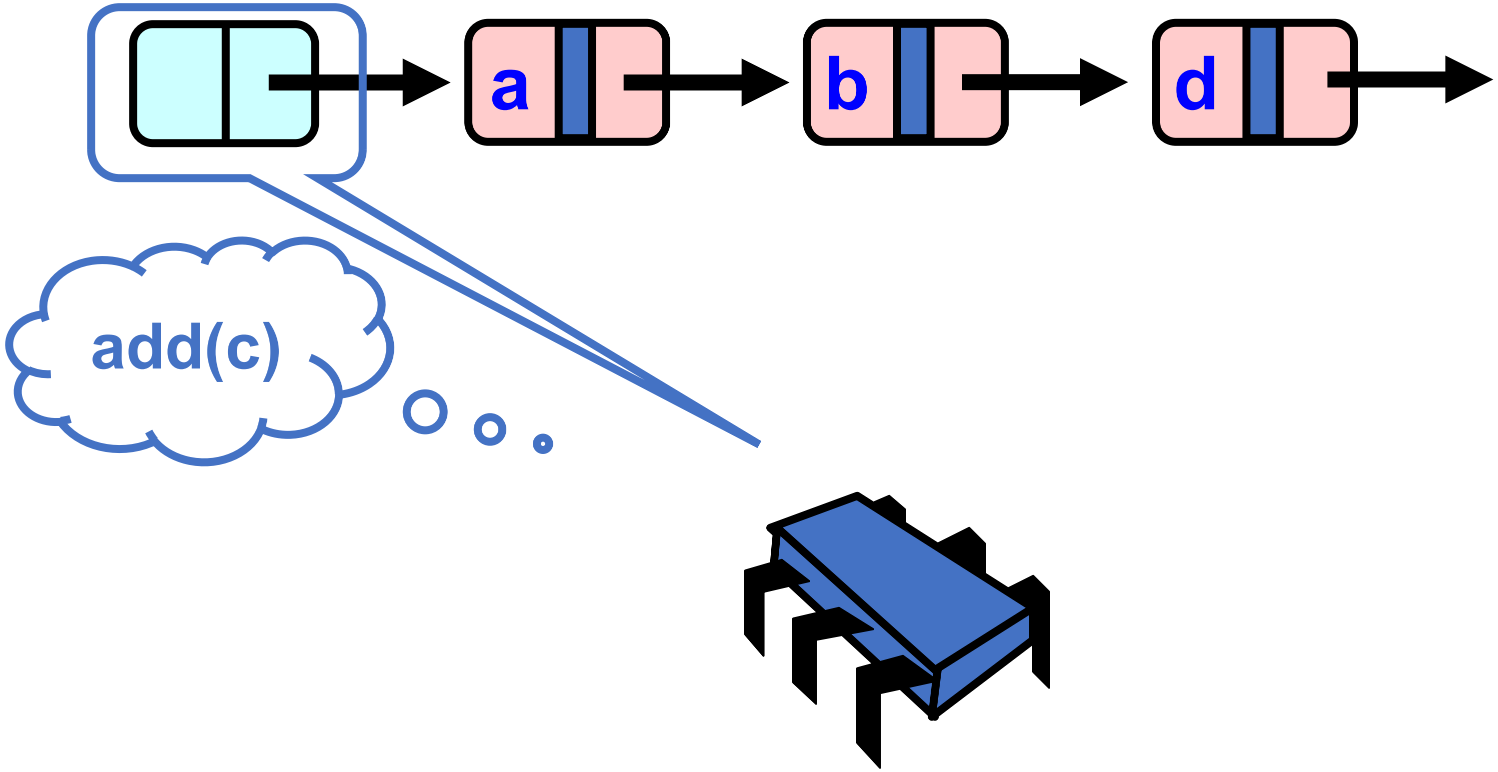
What could go wrong?



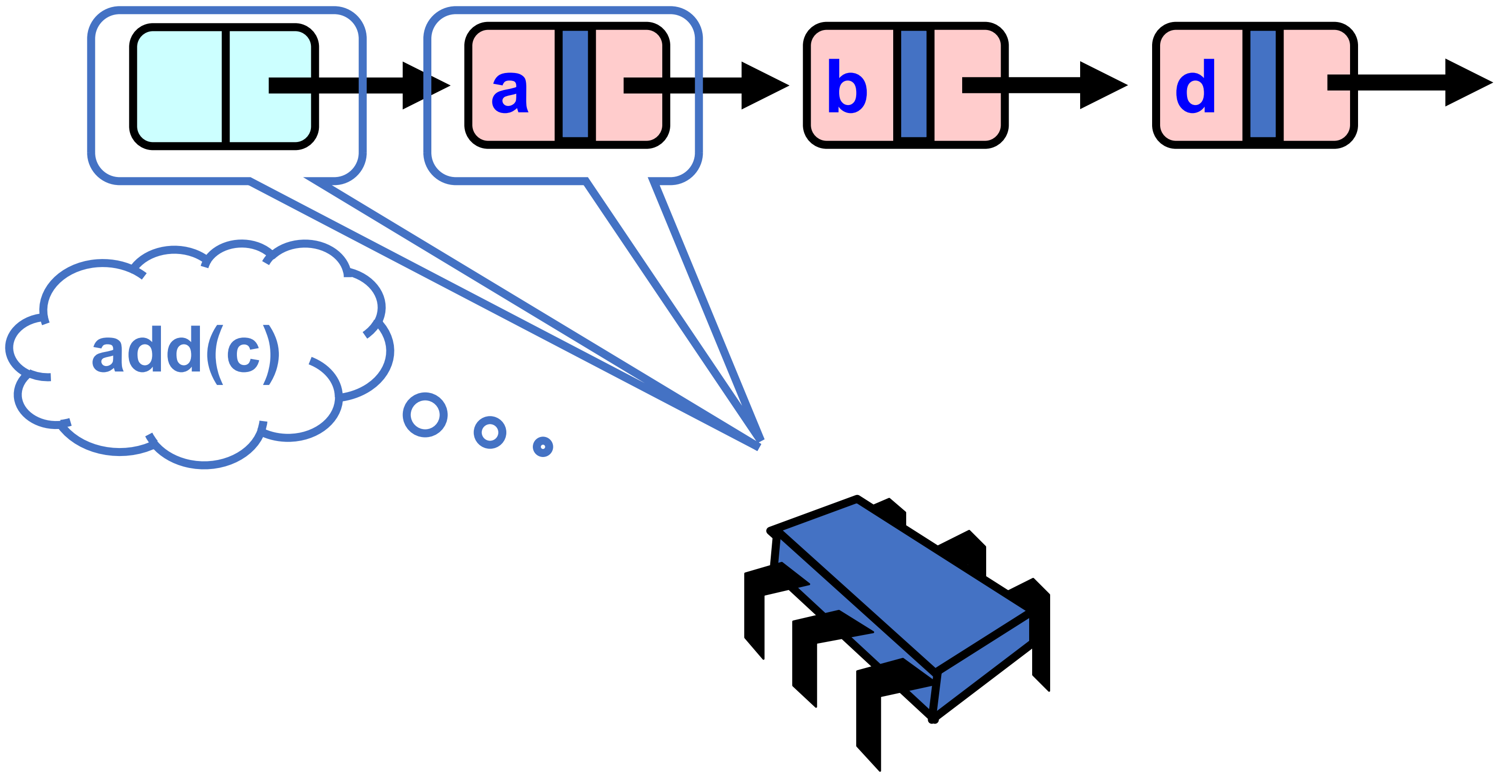
What could go wrong?



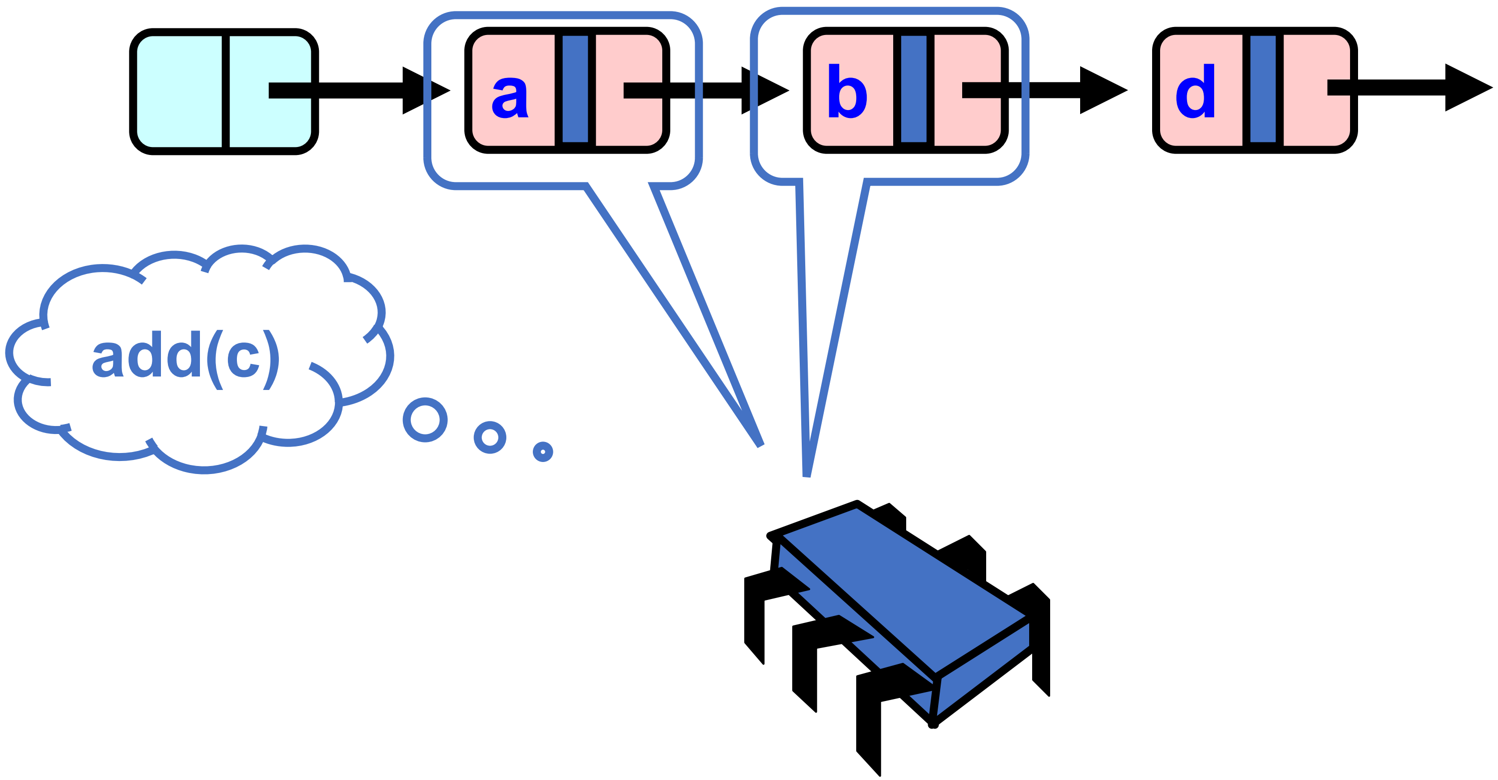
Fixed with logical flag



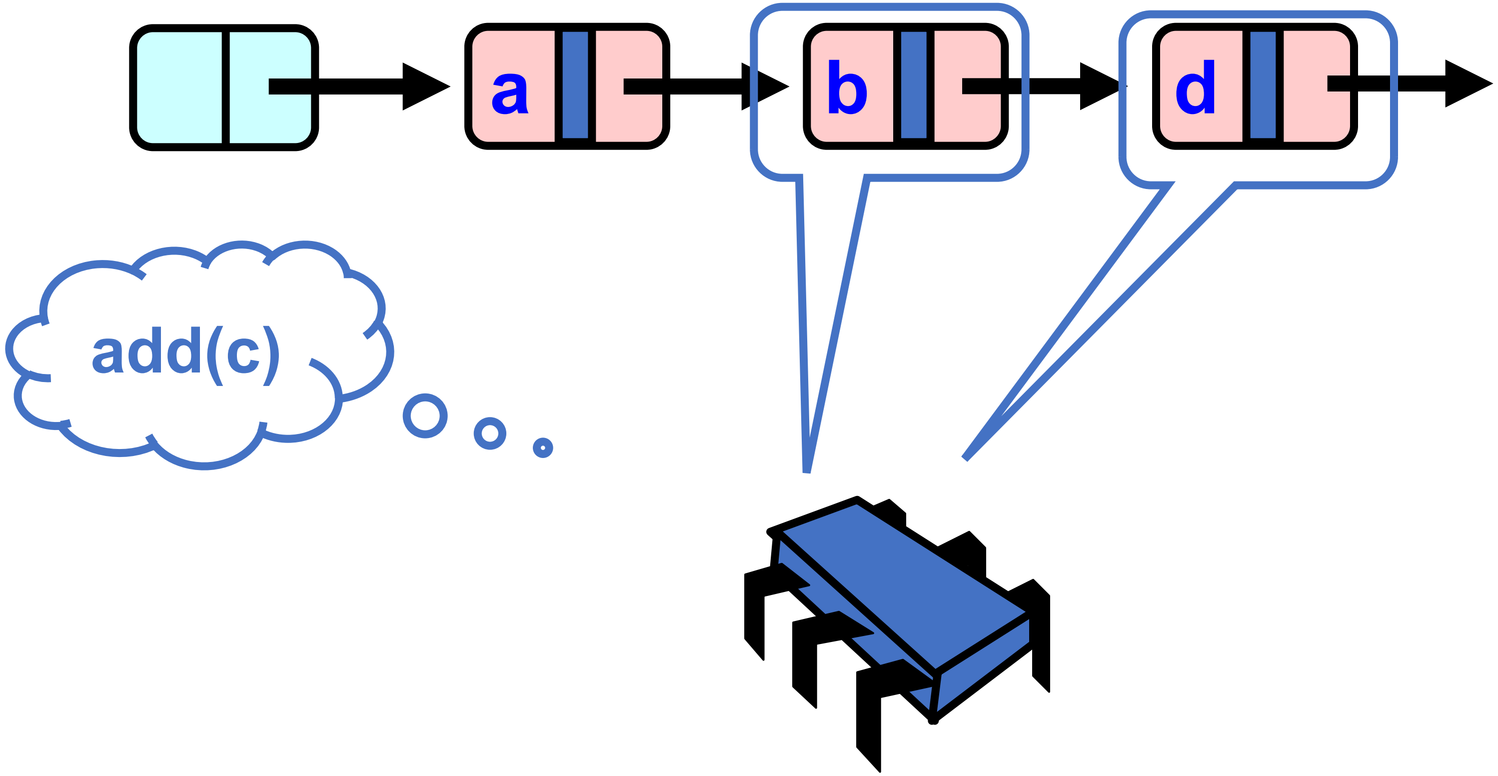
Fixed with logical flag



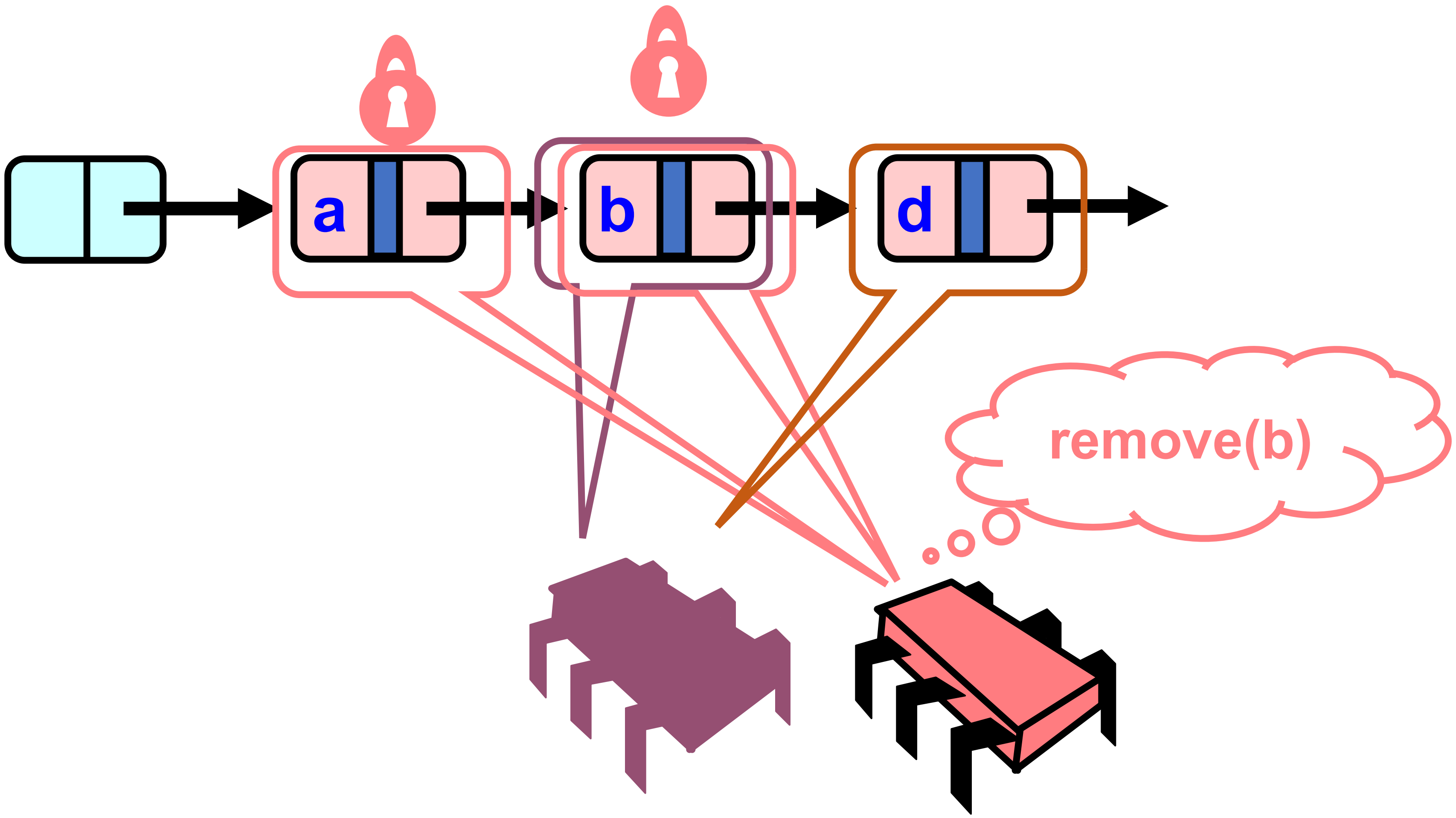
Fixed with logical flag



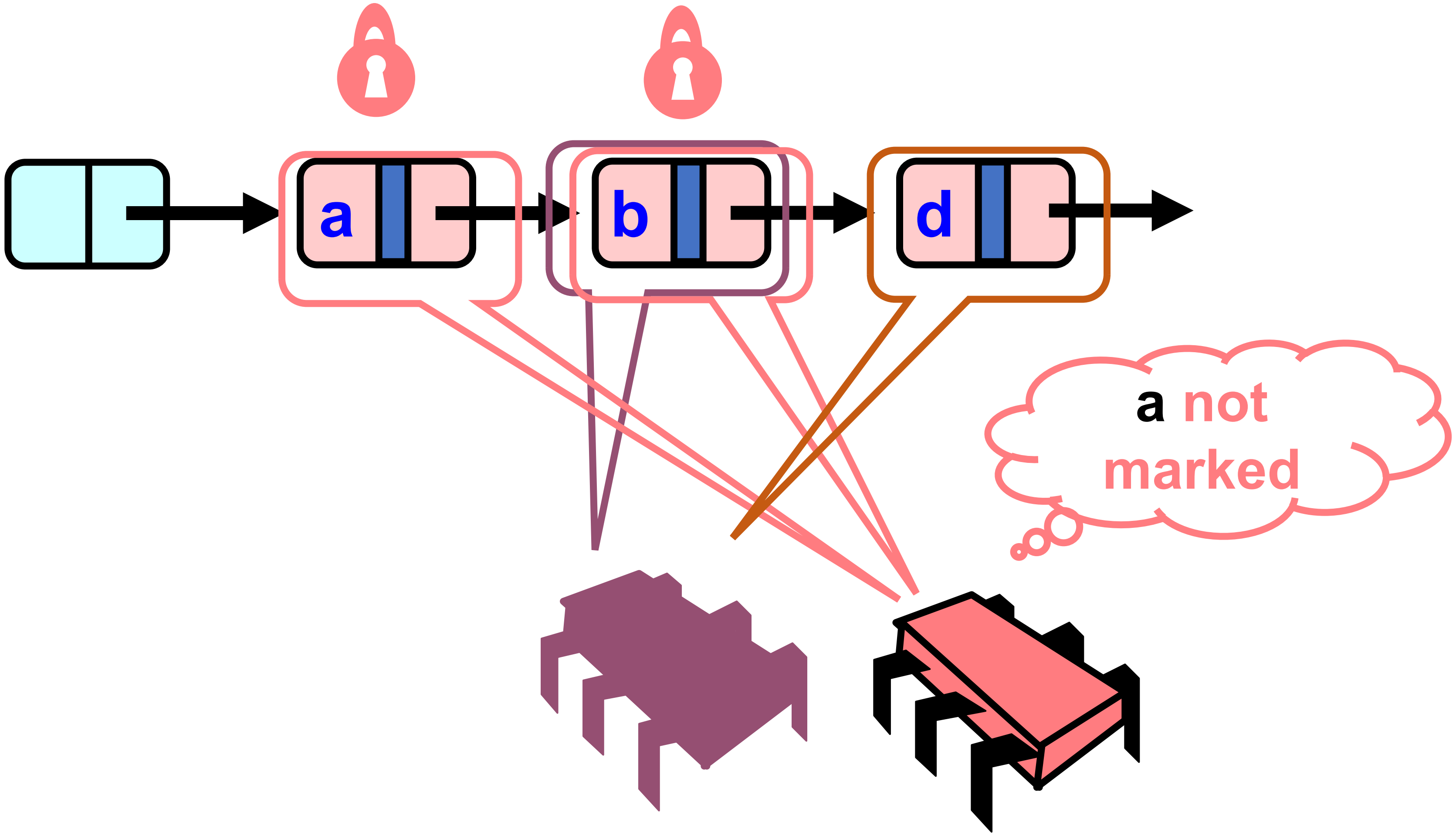
Fixed with logical flag



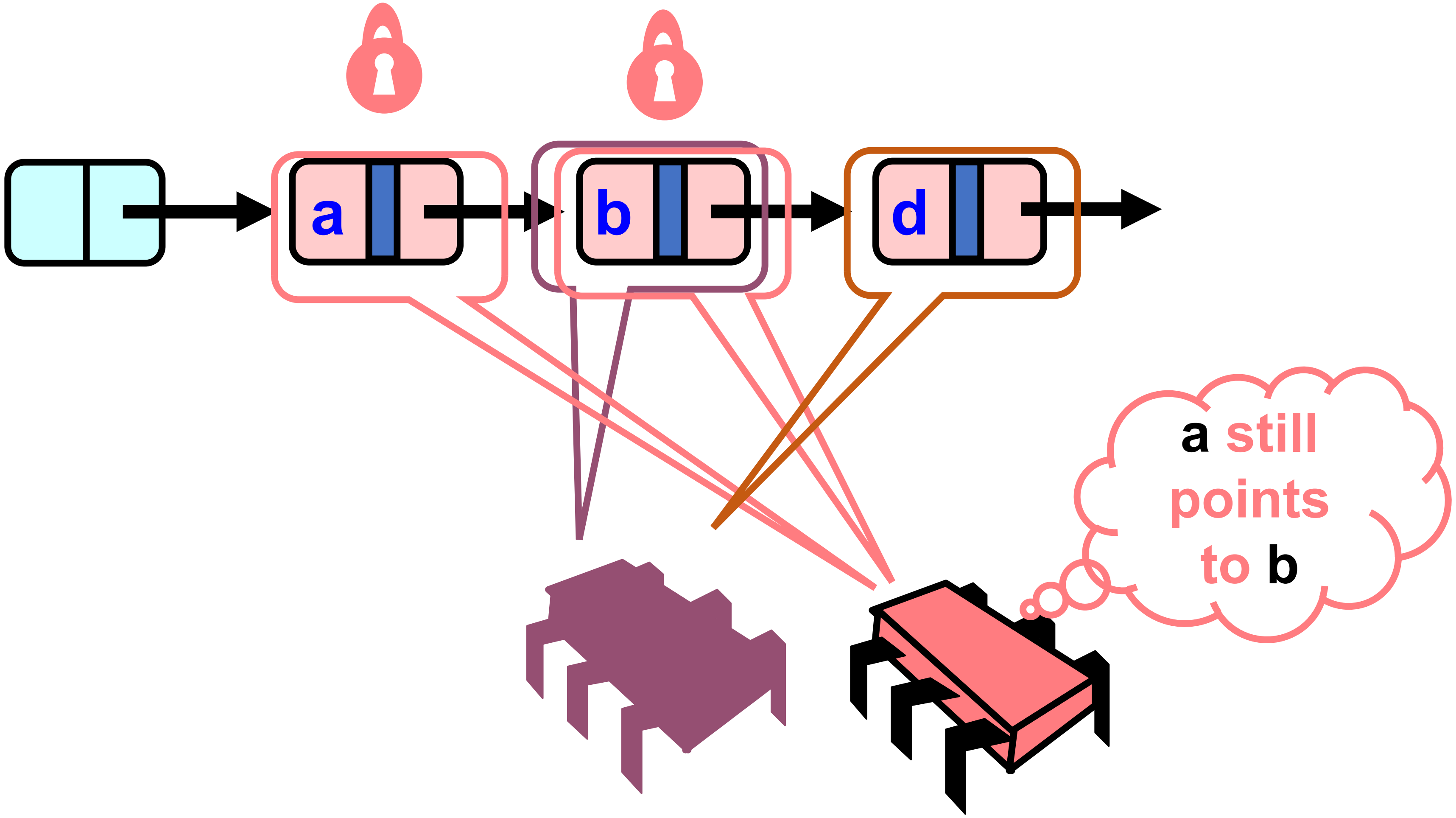
Fixed with logical flag



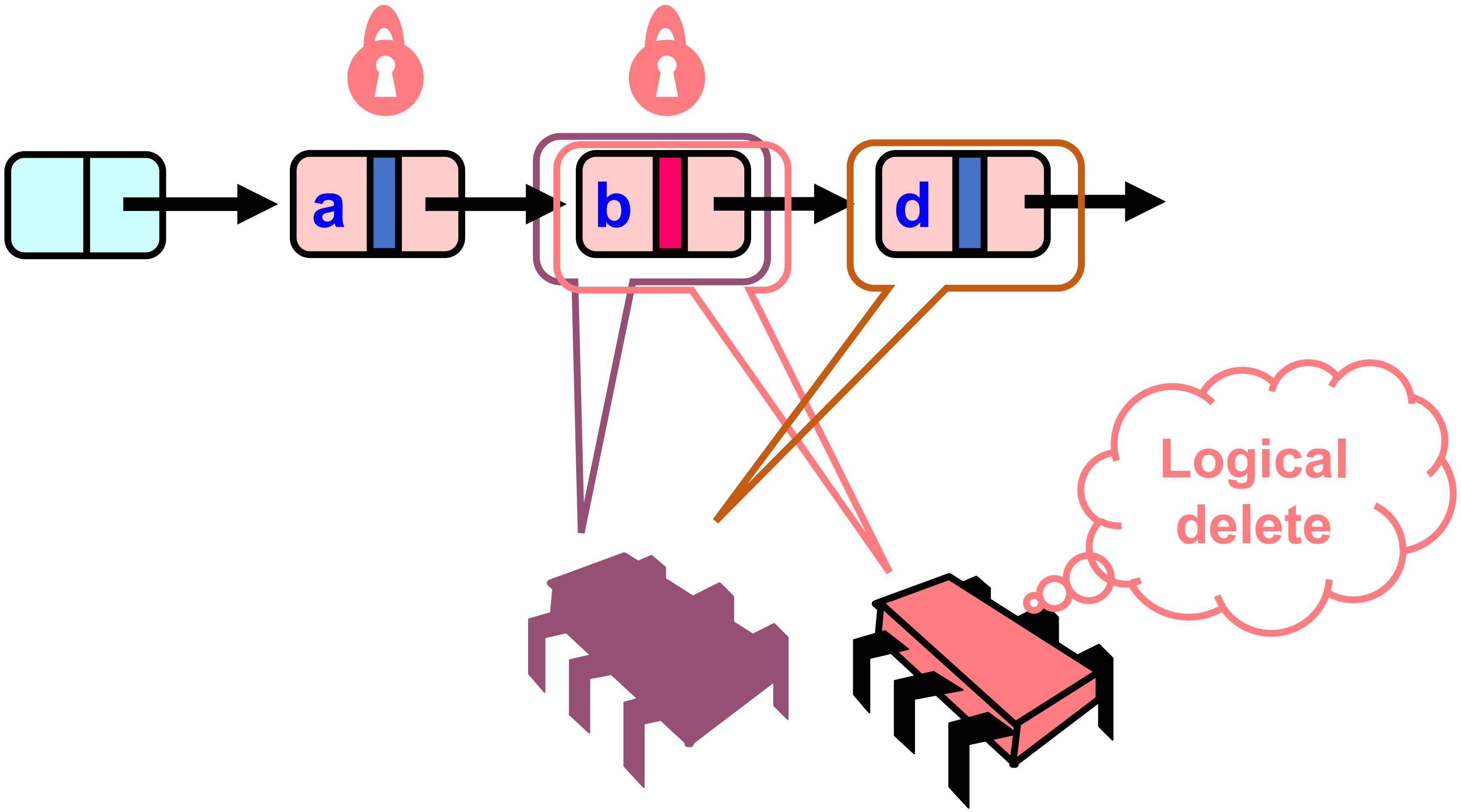
Fixed with logical flag



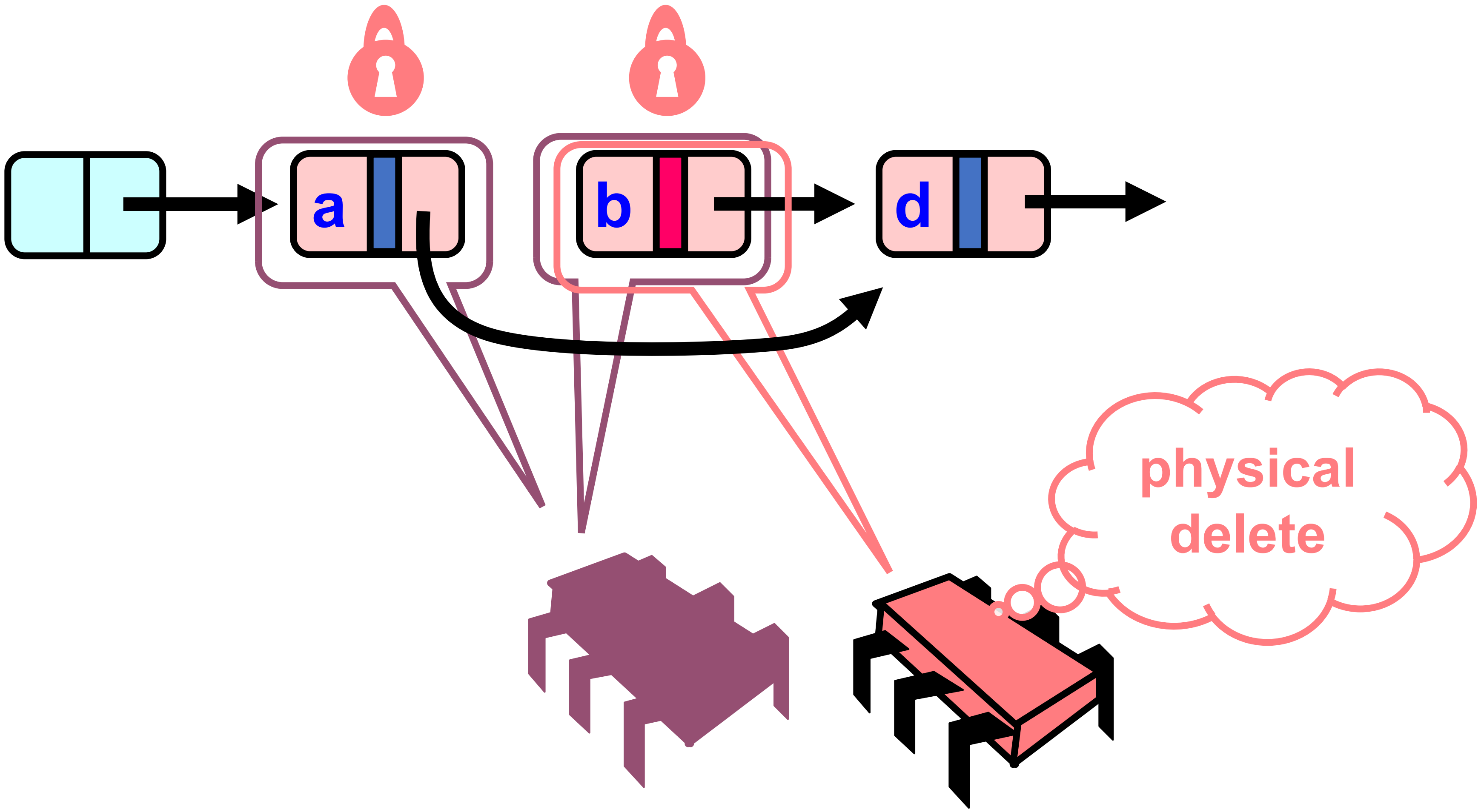
Fixed with logical flag



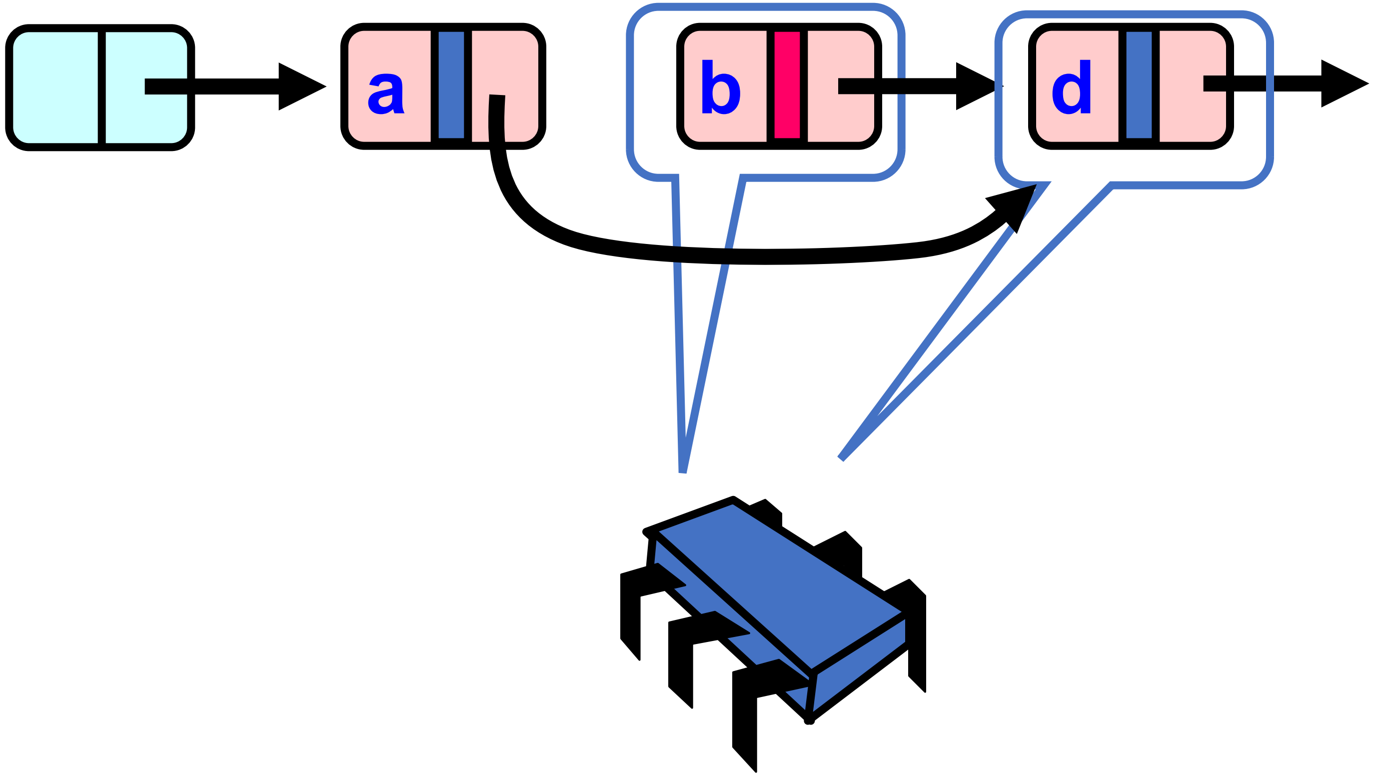
Fixed with logical flag



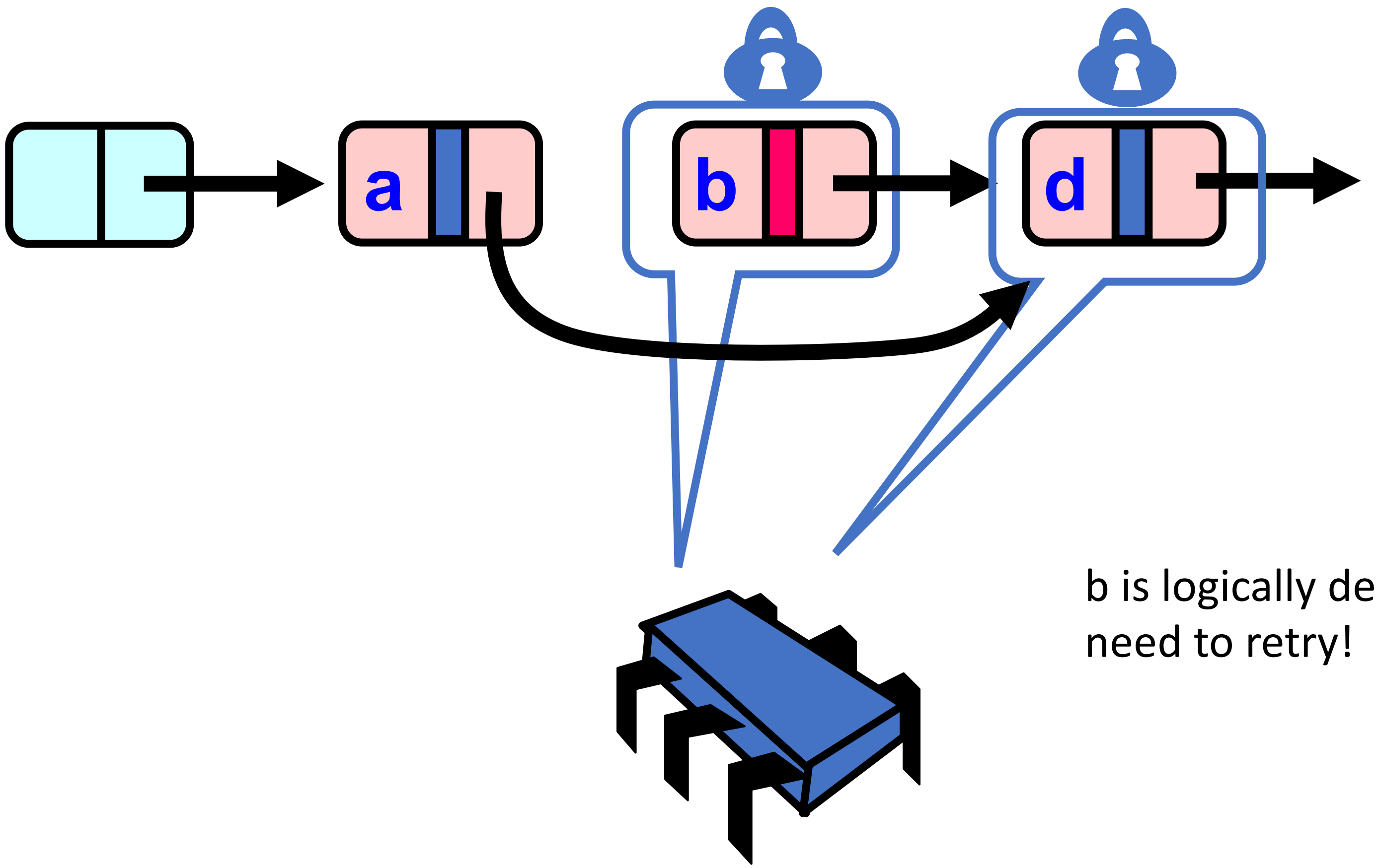
Fixed with logical flag



Fixed with logical flag



Fixed with logical flag



b is logically deleted so we need to retry!

To complete the picture

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

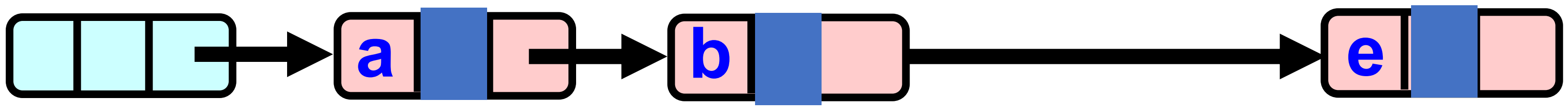
Evaluation

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

Lock-free Lists

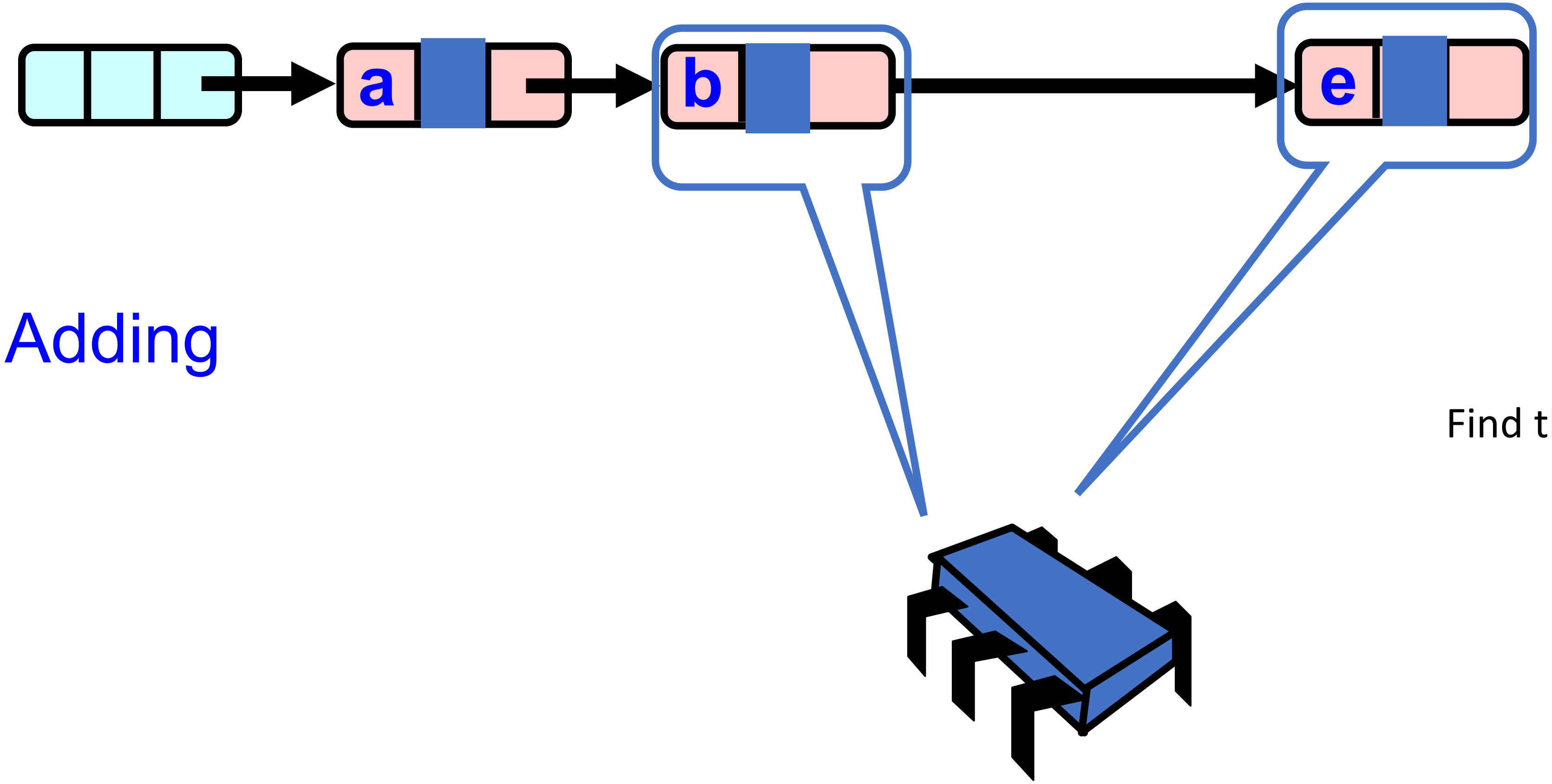
- Next logical step
  - lock-free add() and remove()
- What sort of atomics do we need?
  - Loads/stores?
  - RMWs?

Lock-free Lists



Adding

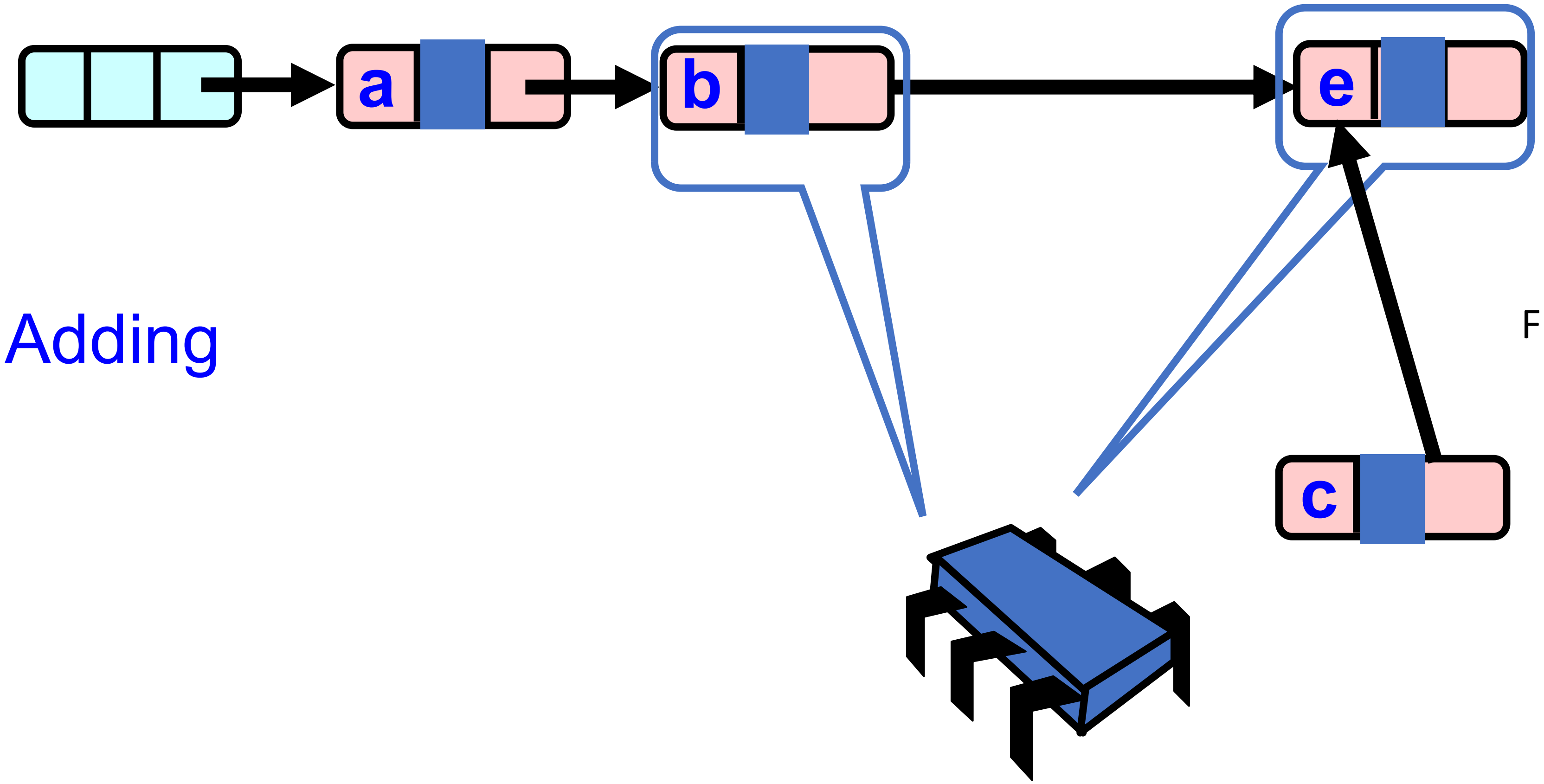
Lock-free Lists



Adding

Find the location

Lock-free Lists

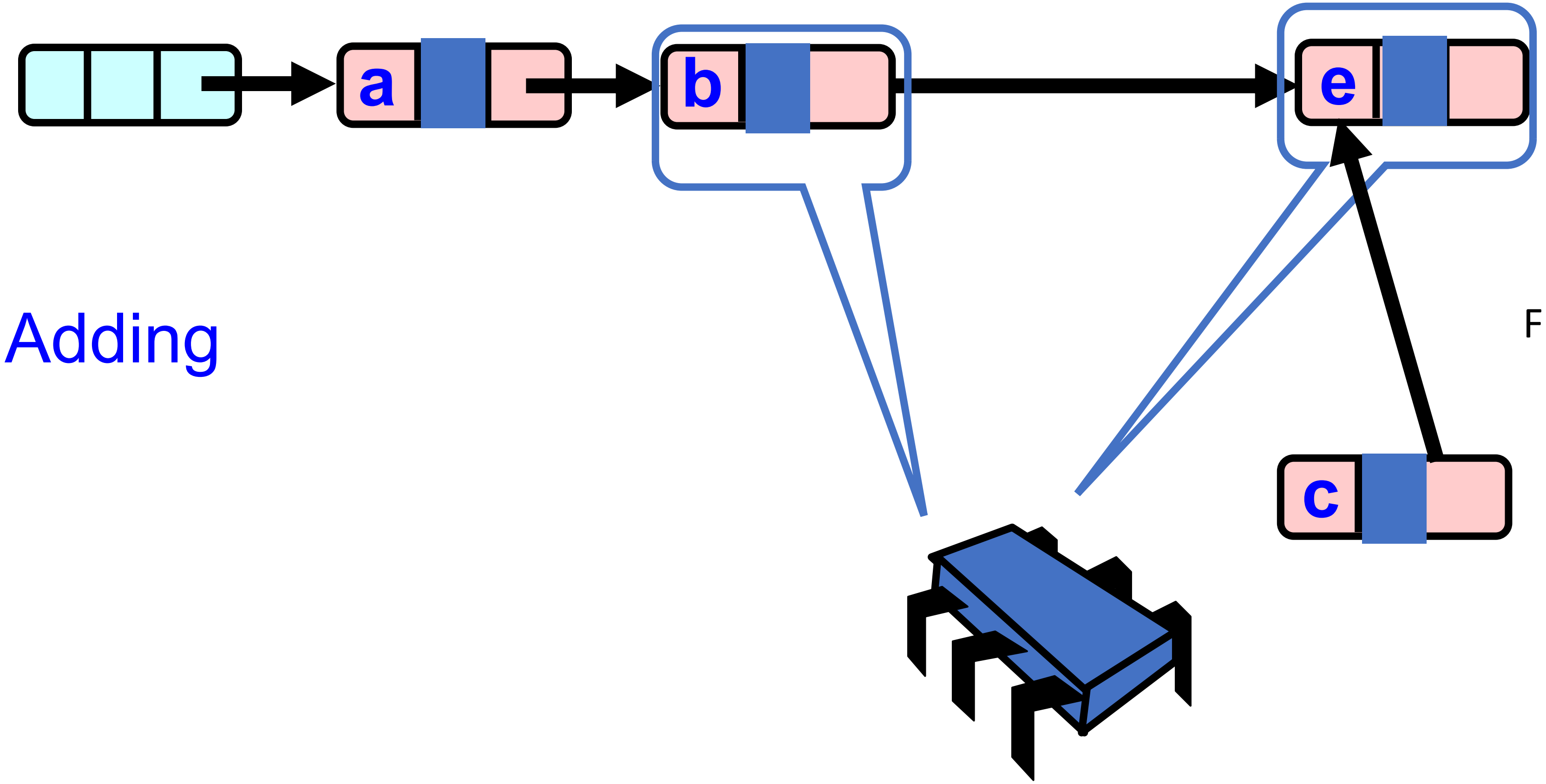


Adding

Find the location

create "c"

Lock-free Lists



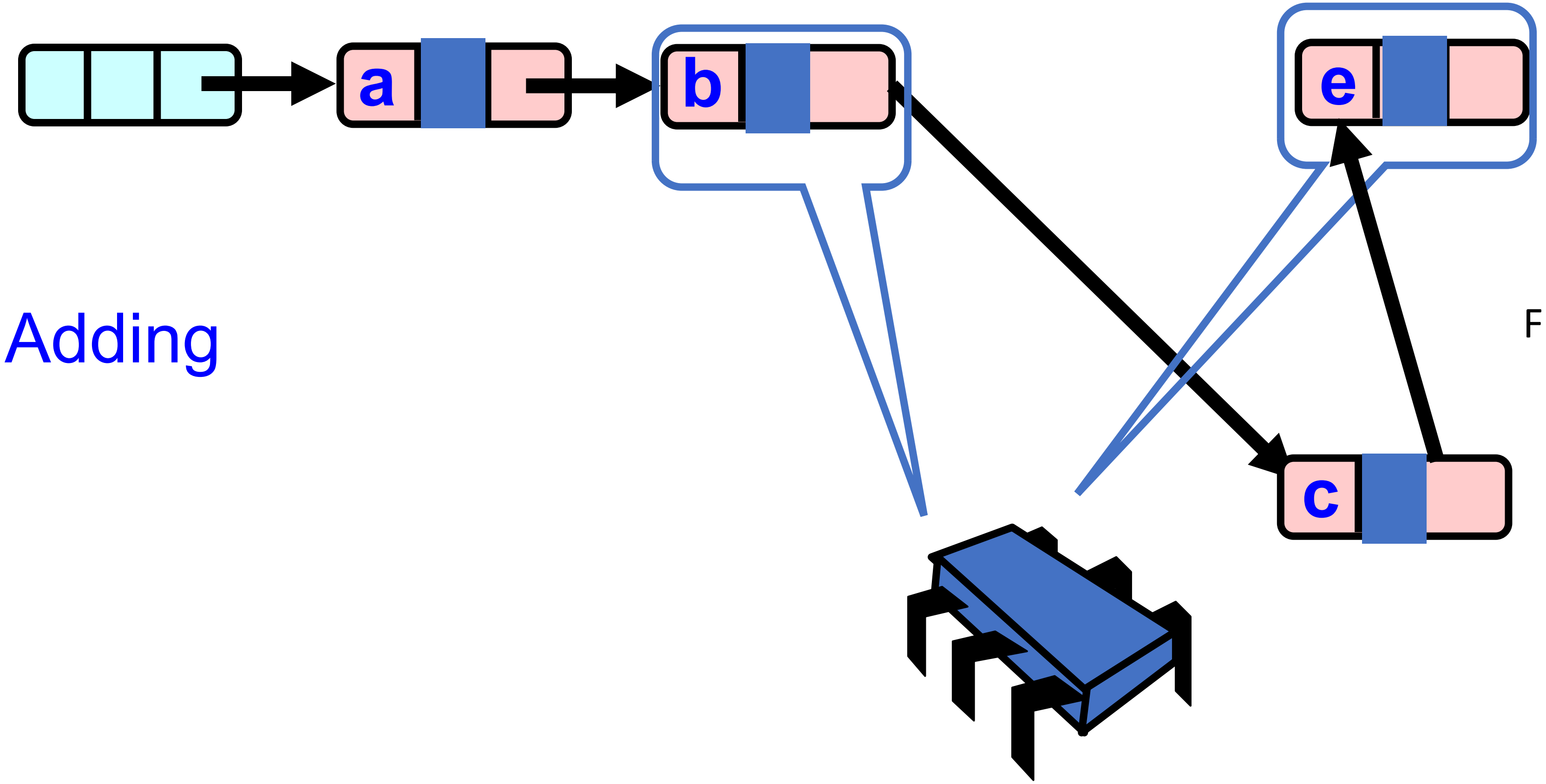
Adding

Find the location

create “c”

insert “c”

Lock-free Lists



Adding

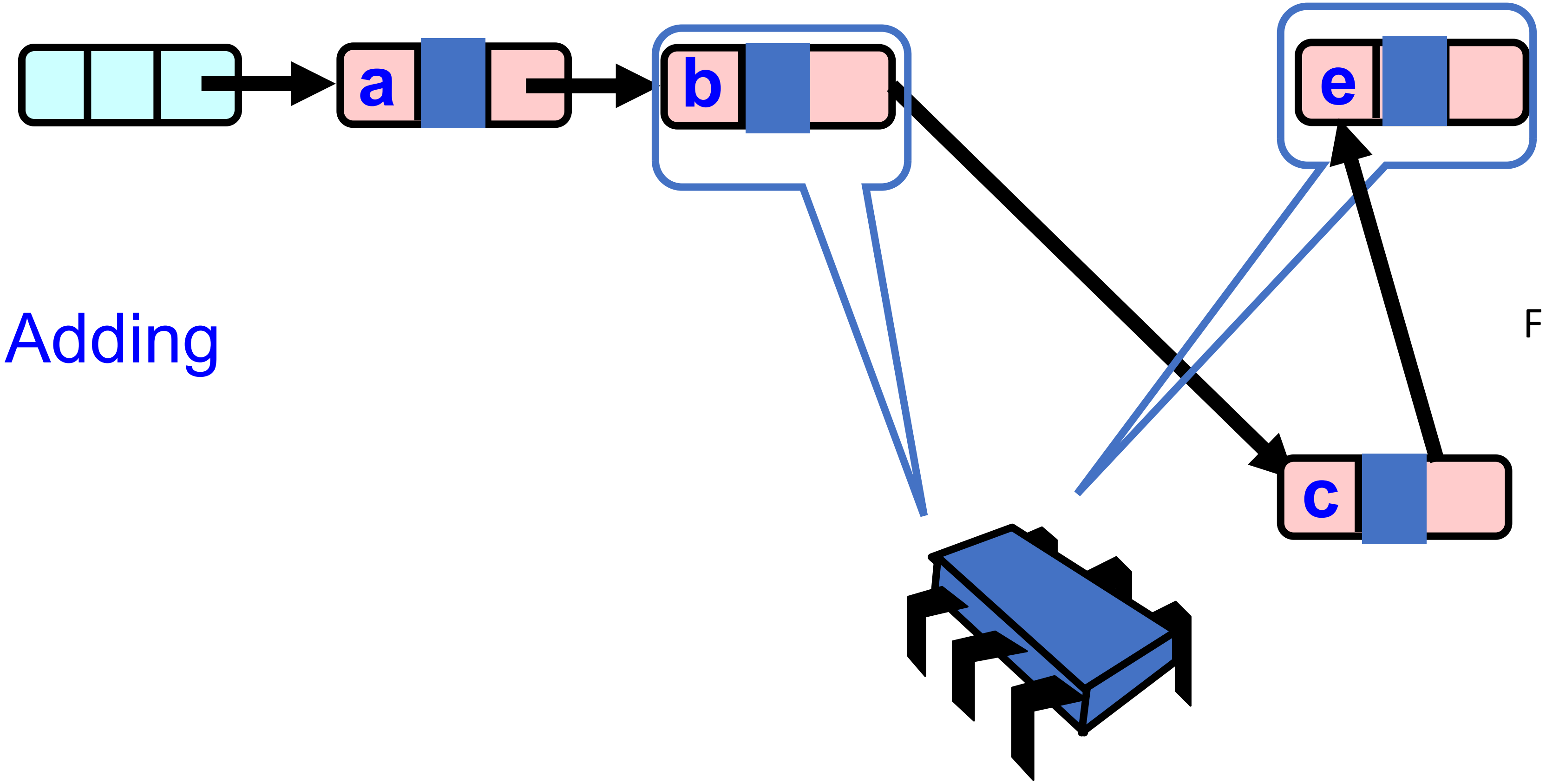
Find the location

create “c”

insert “c”

Lock-free Lists

Adding



Find the location

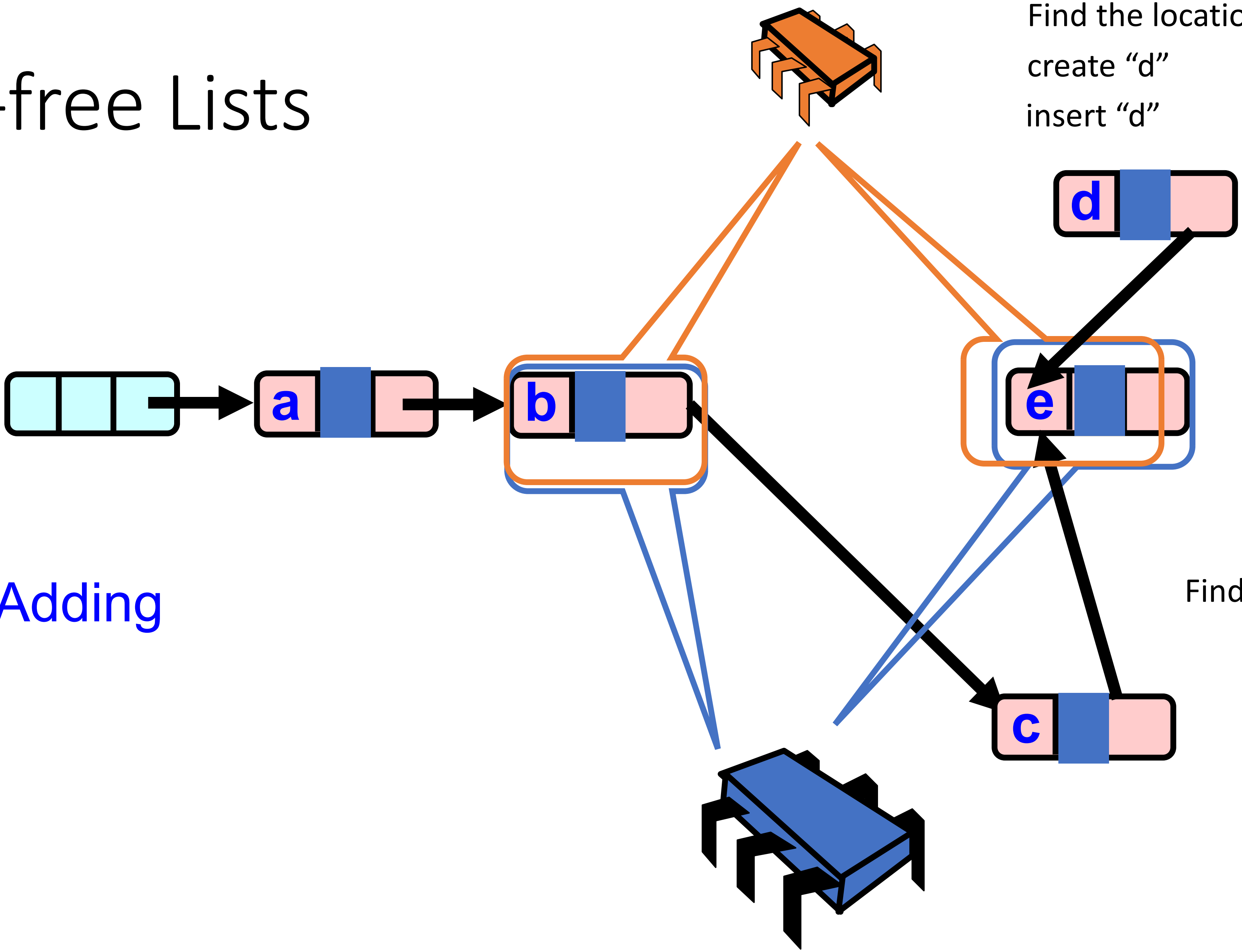
create “c”

insert “c”

Can this just be a regular store?

Lock-free Lists

Adding



Find the location

create "d"

insert "d"

Can this just be a regular store?

Find the location

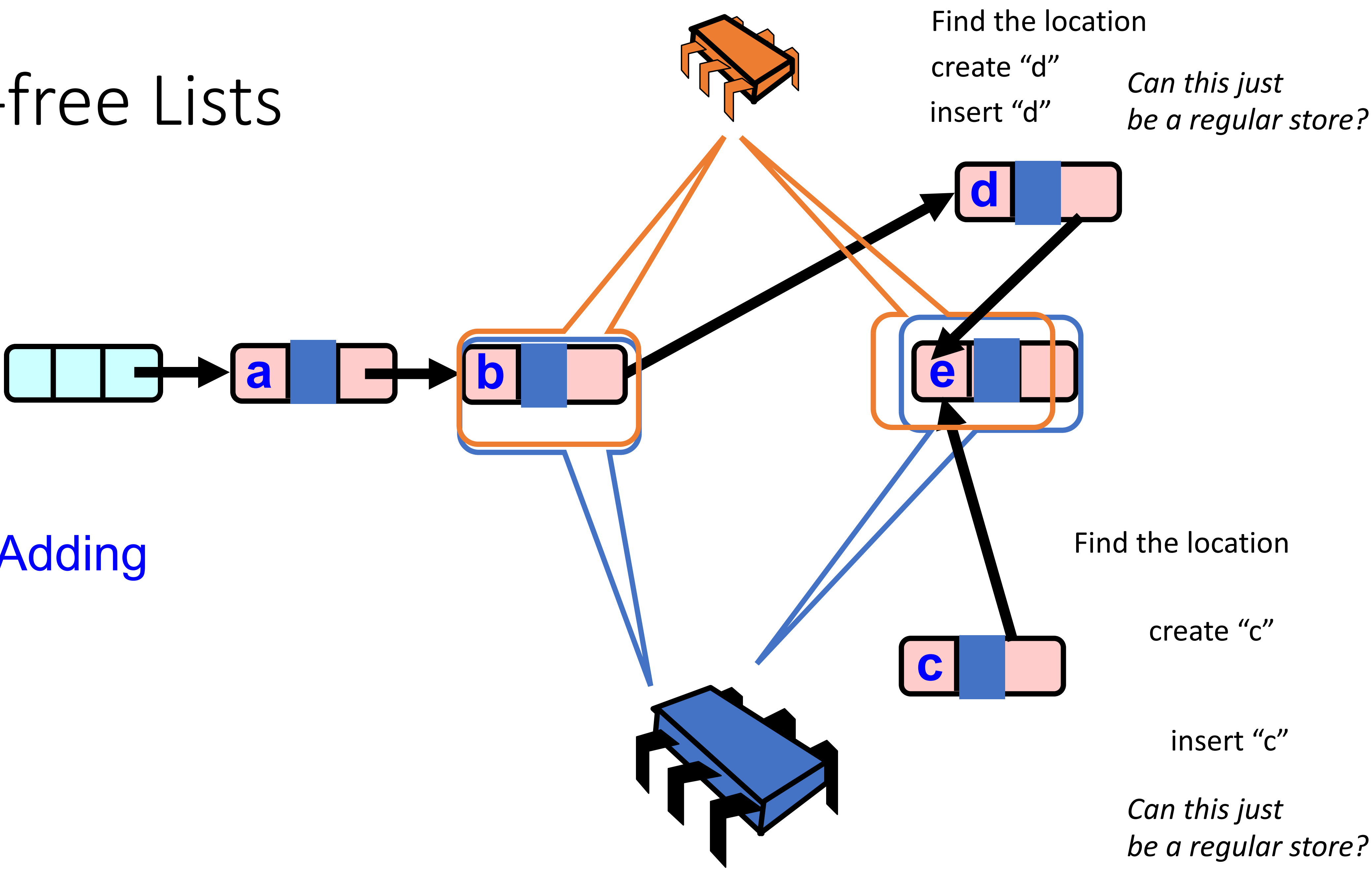
create "c"

insert "c"

Can this just be a regular store?

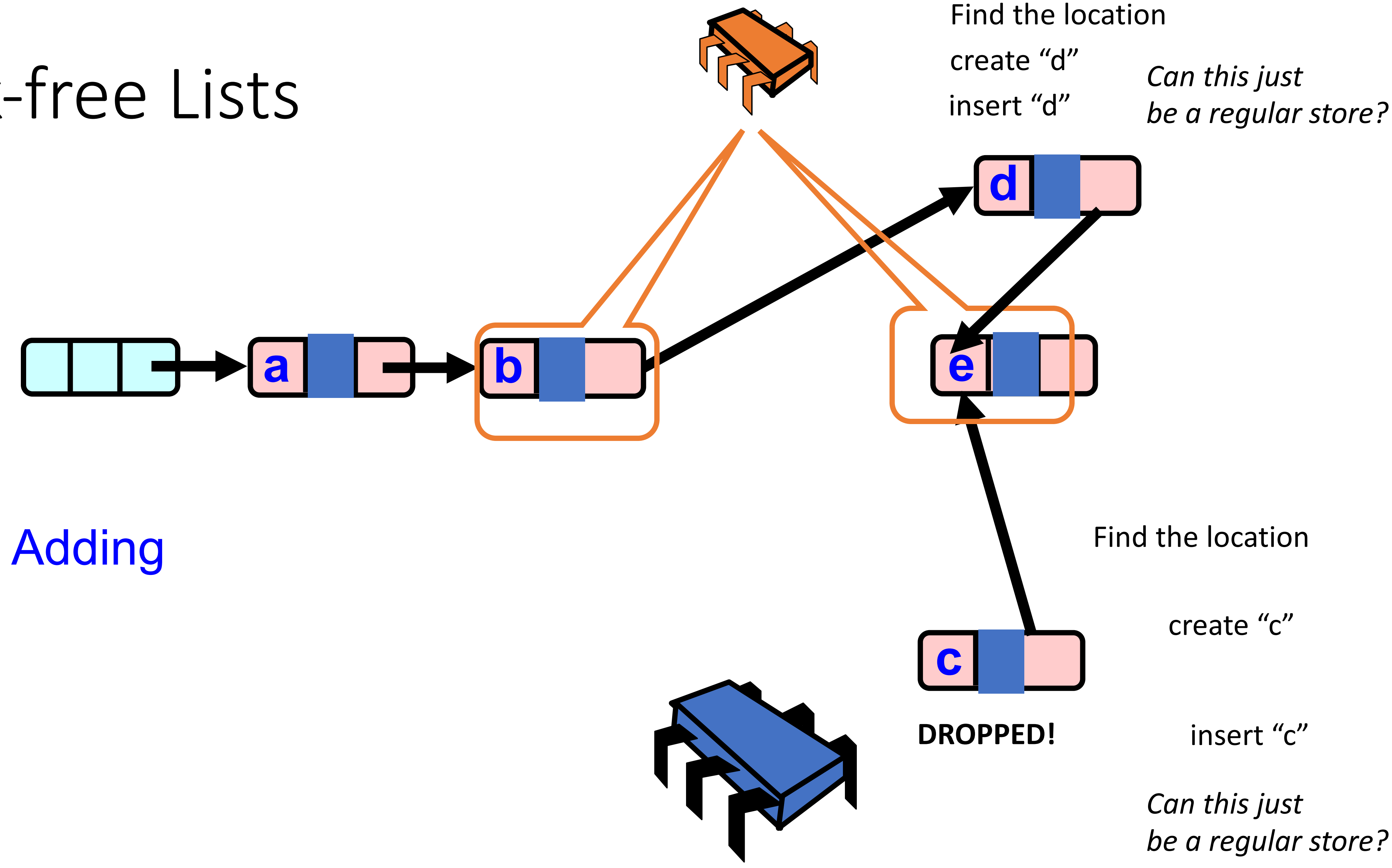
Lock-free Lists

Adding

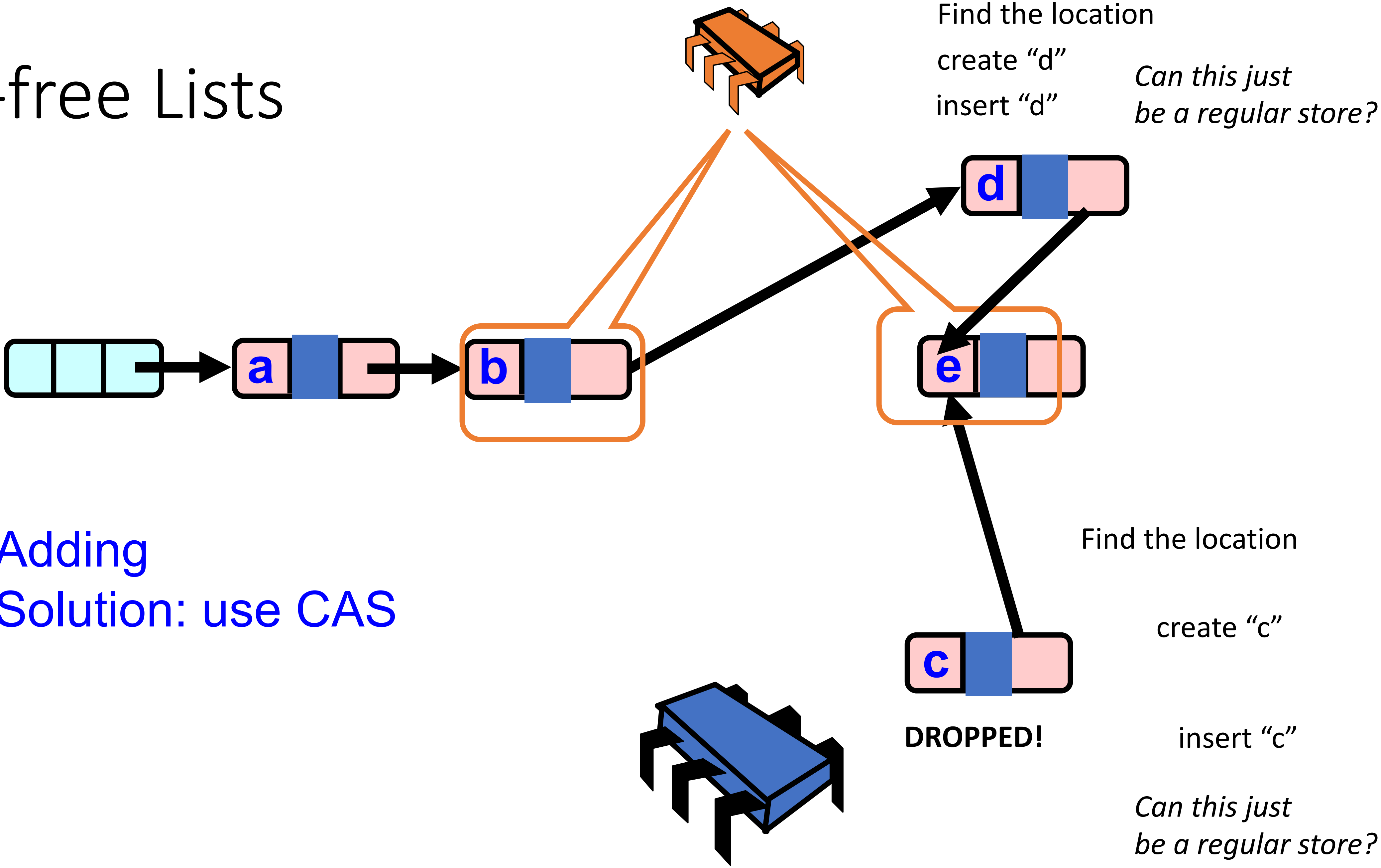


Lock-free Lists

Adding



Lock-free Lists



Adding
Solution: use CAS

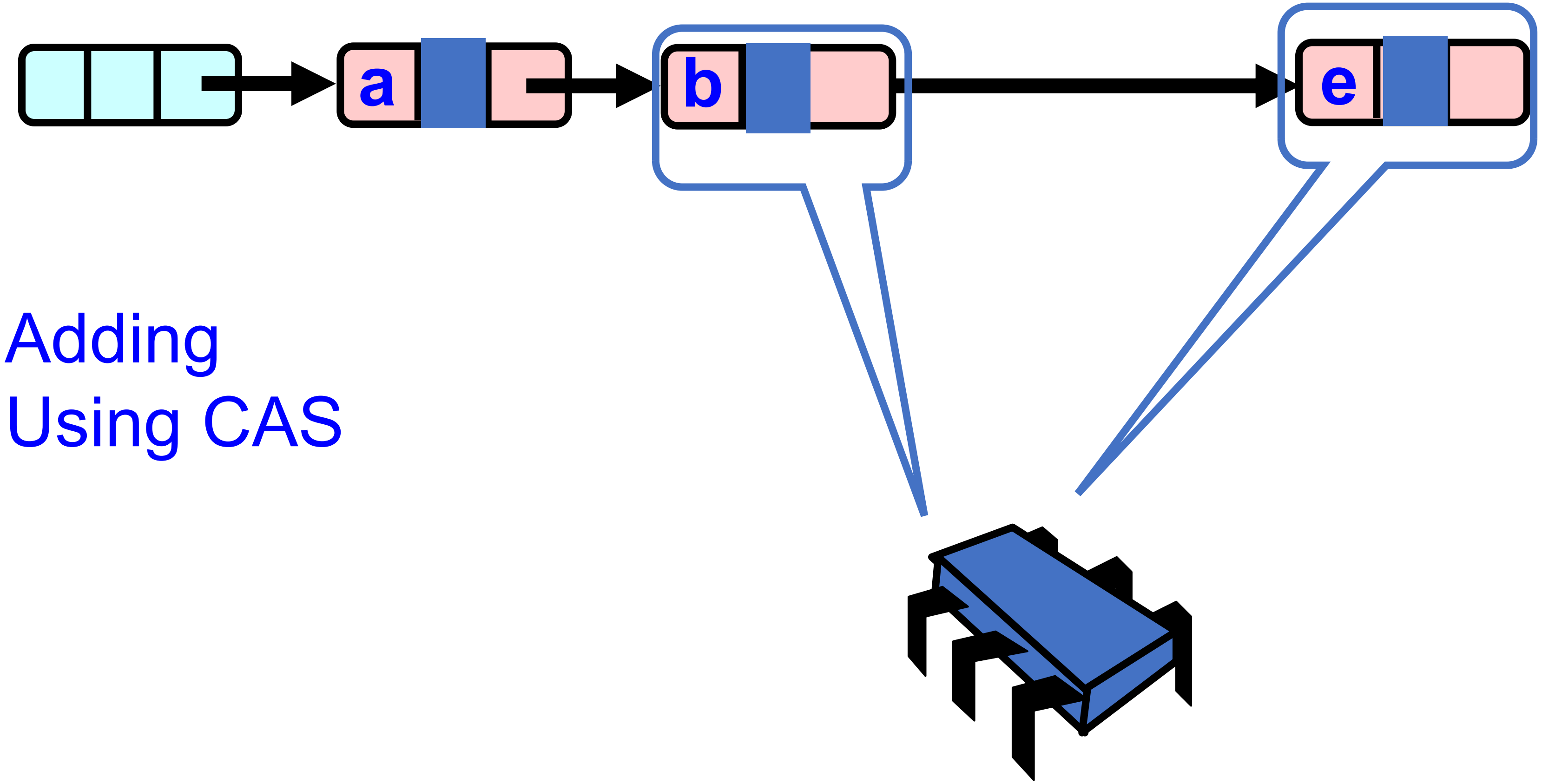
Lock-free Lists

Find the location

Cache your insertion point!

b.next == e

Adding
Using CAS

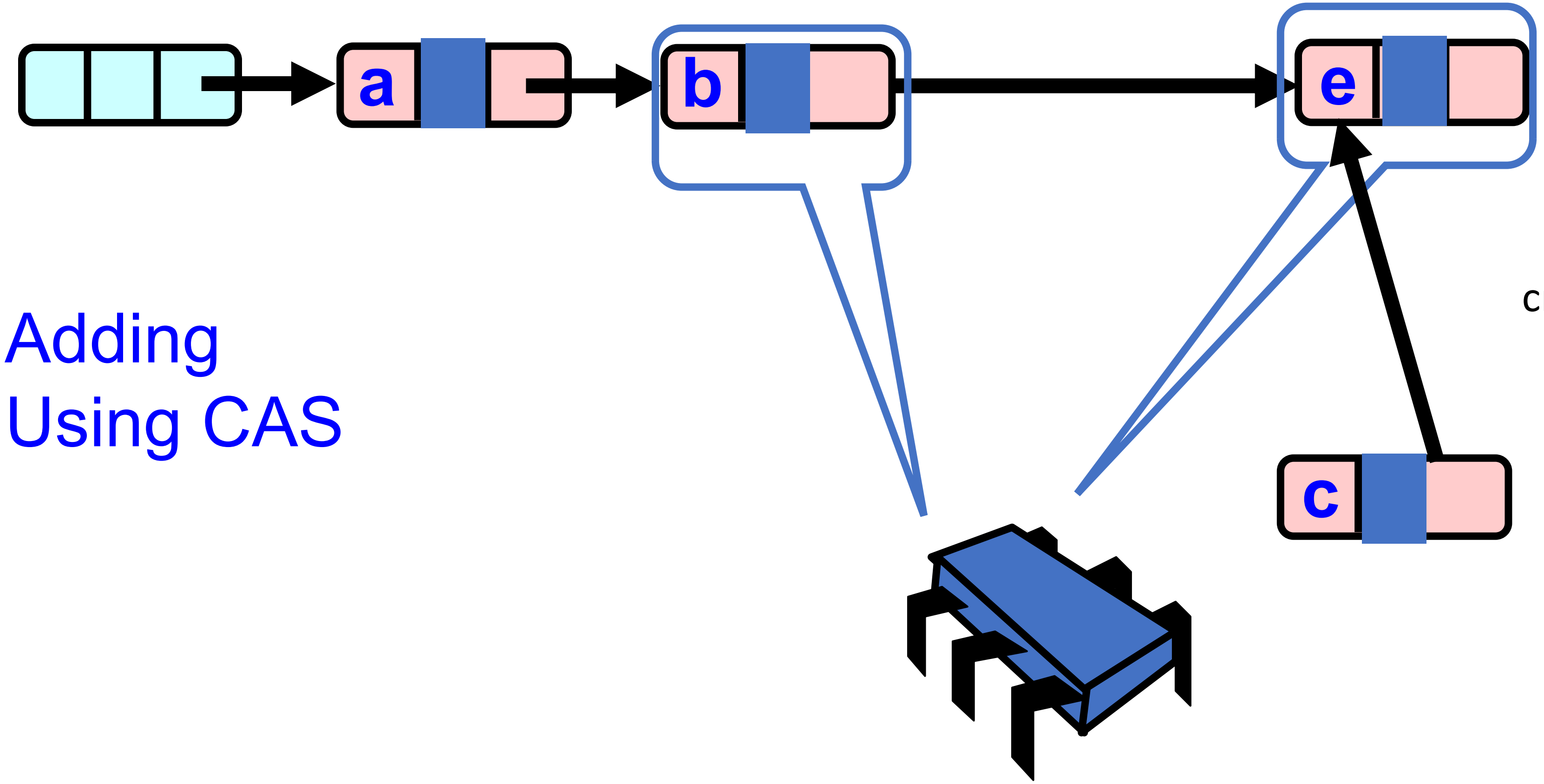


Lock-free Lists

Find the location

Cache your insertion point!

b.next == e



Adding
Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

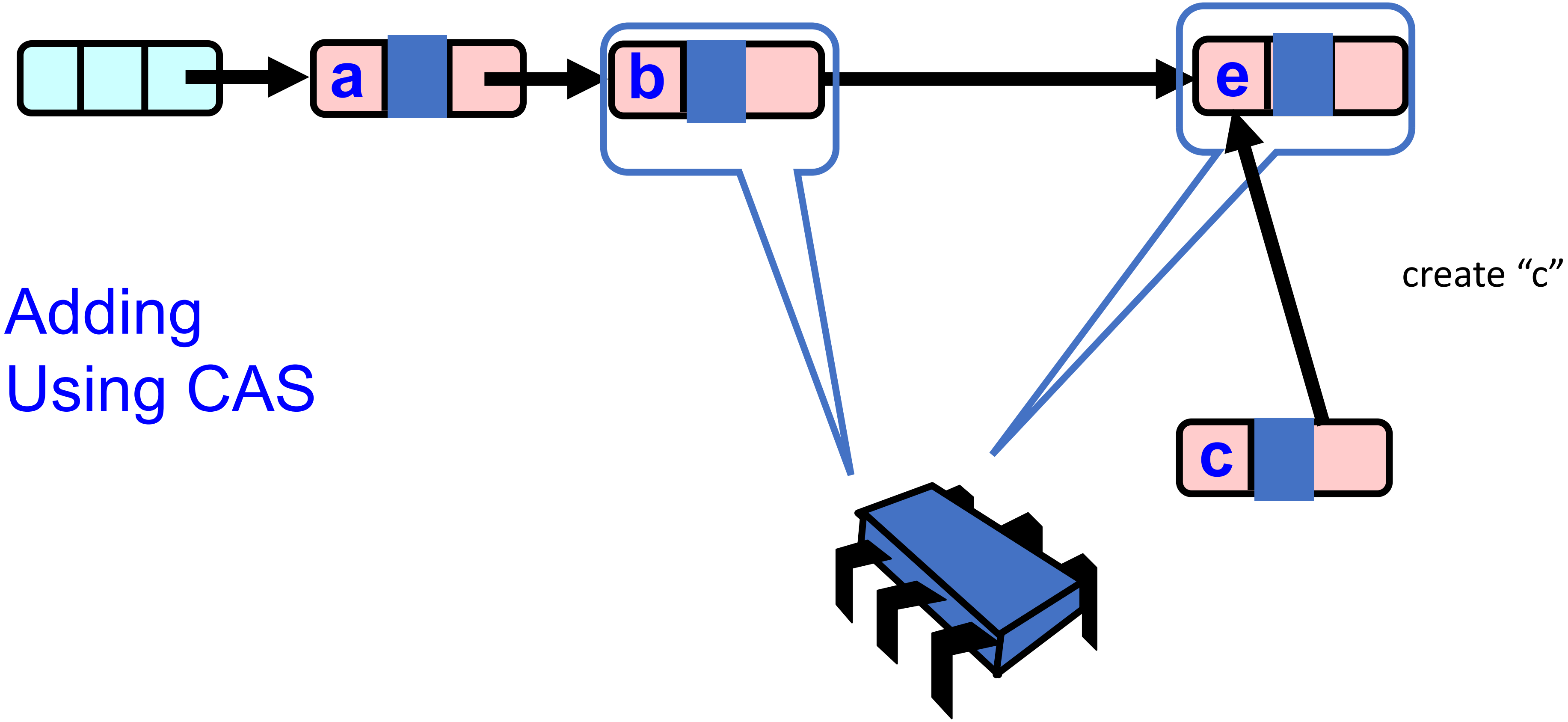
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

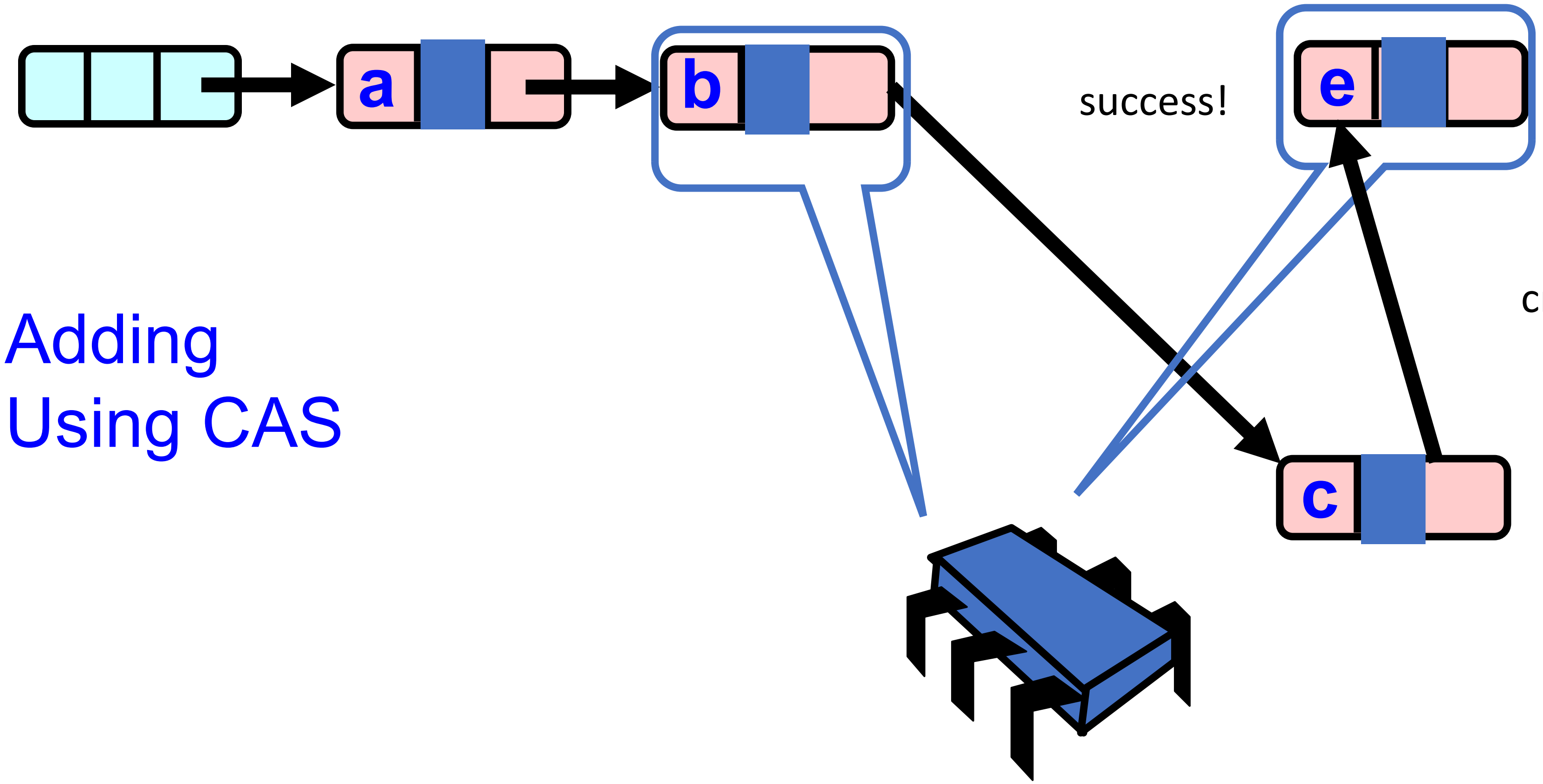
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

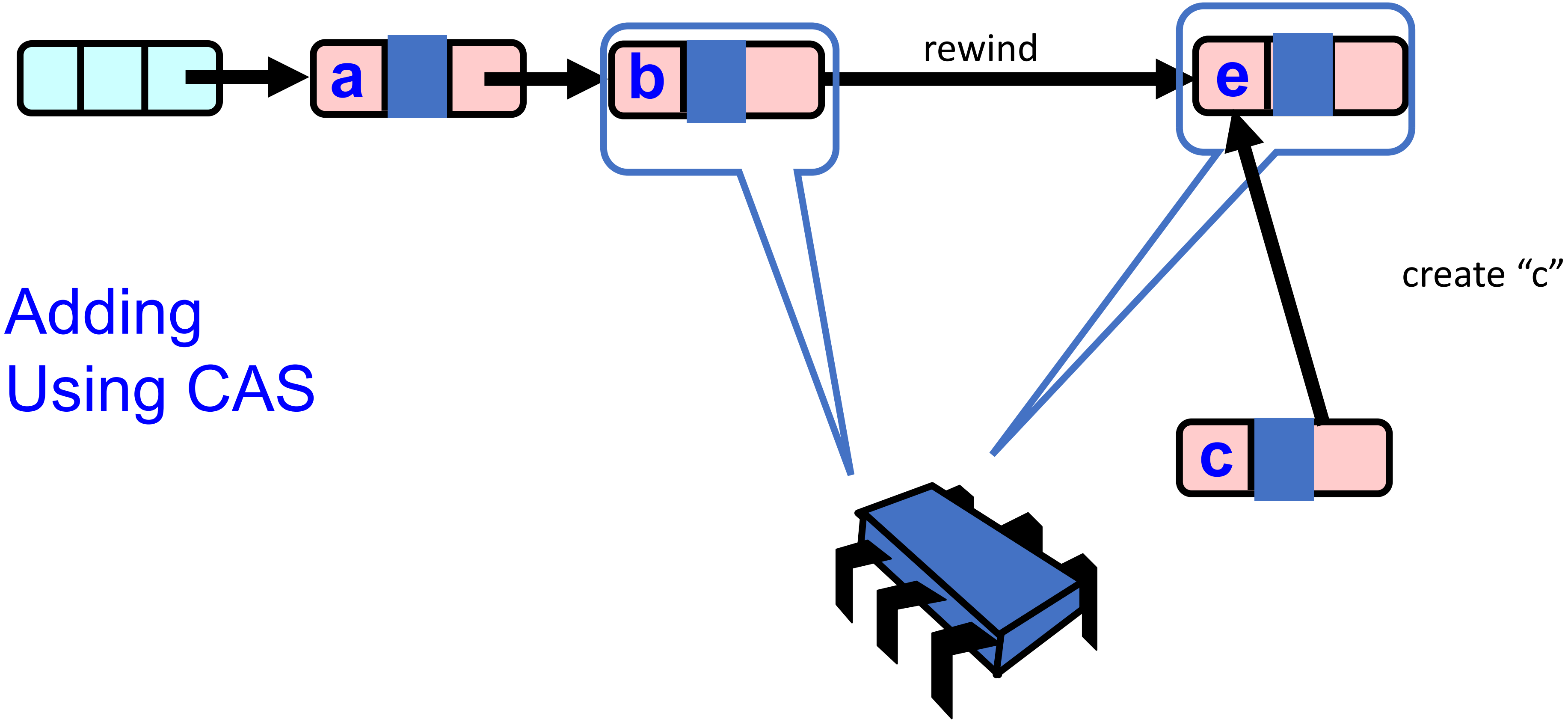
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

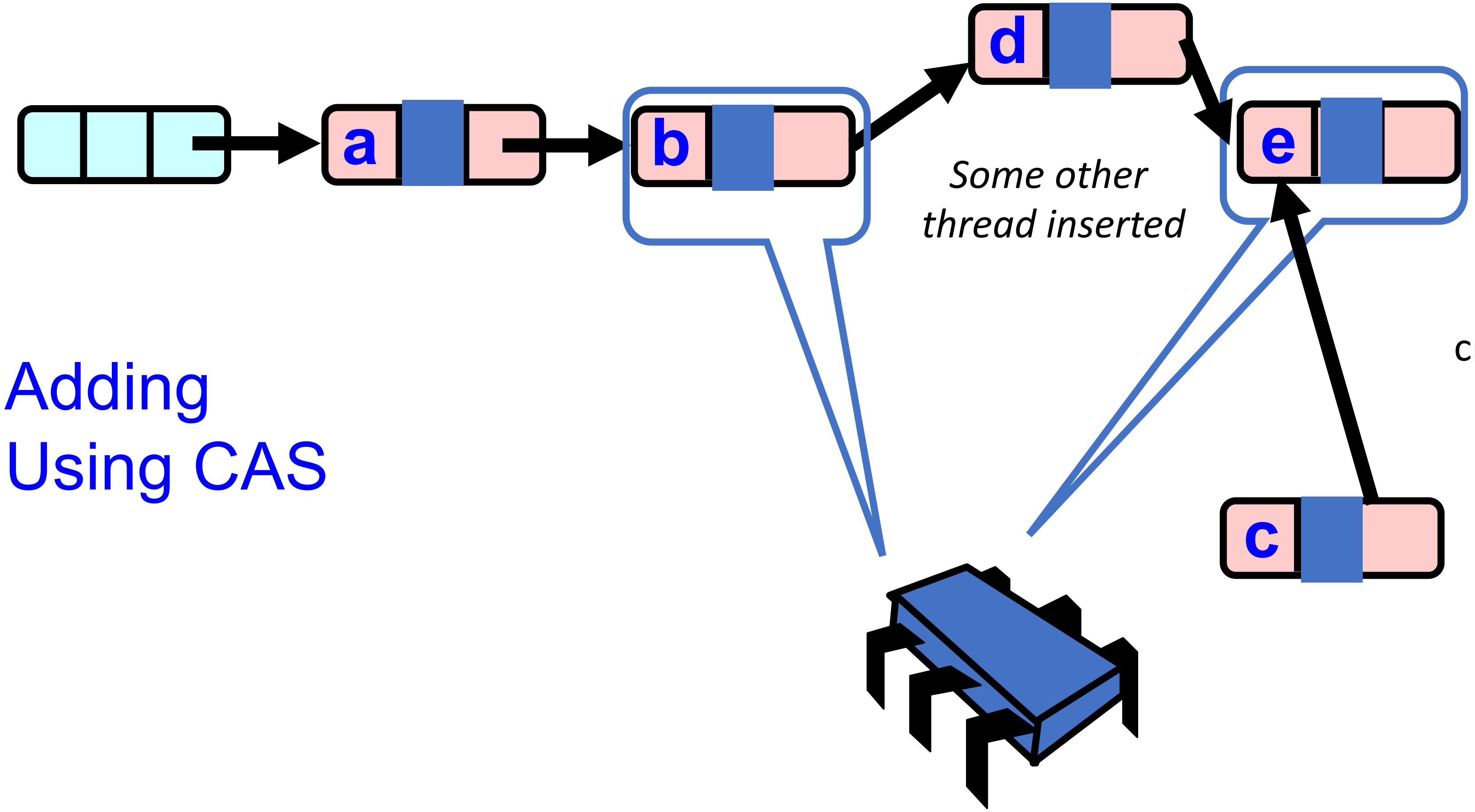
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding
Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

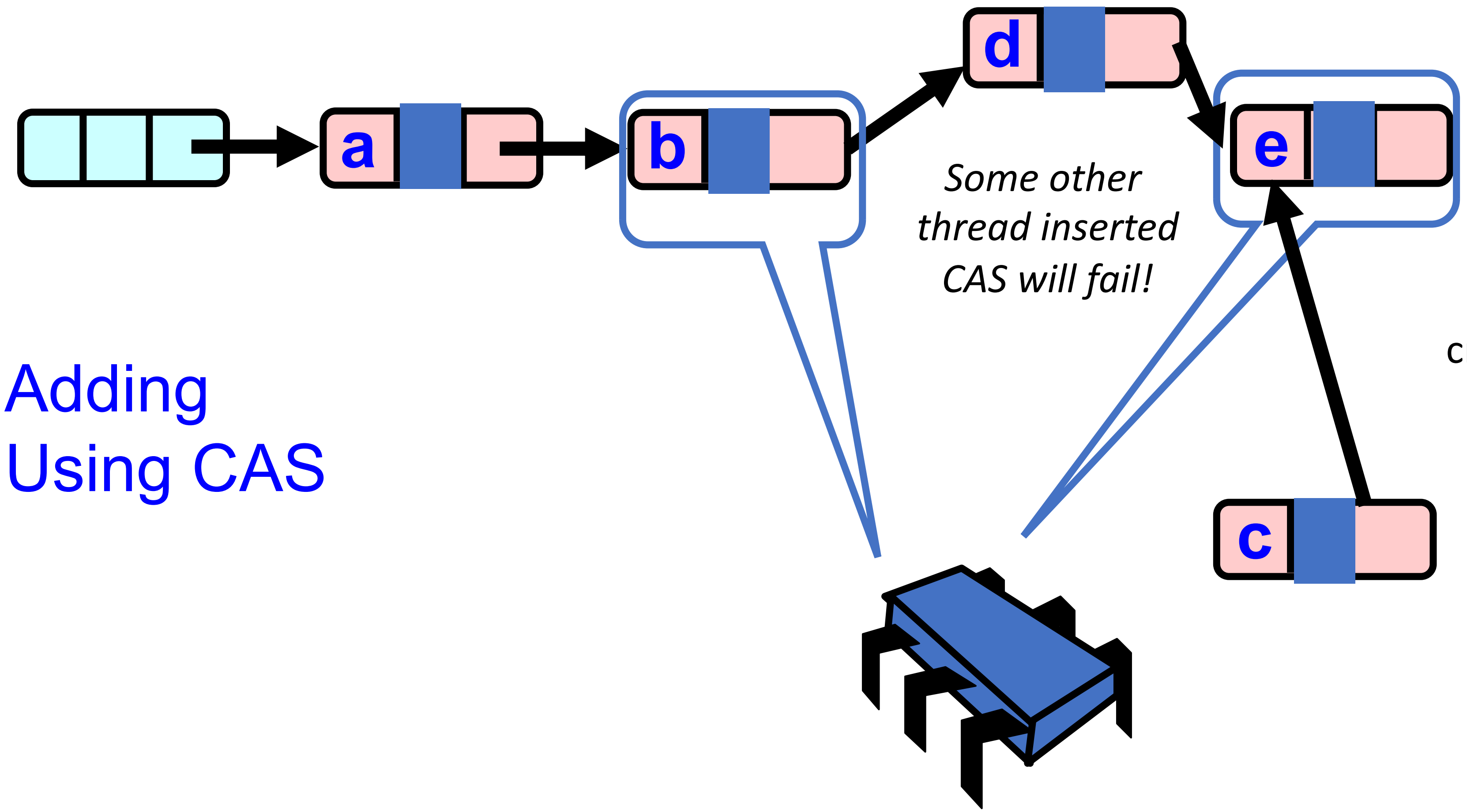
CAS(b.next, e, c);

Find the location

Cache your insertion point!

b.next == e

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



Adding Using CAS

Lock-free Lists

Only insert if your insertion point is valid!

CAS(b.next, e, c);

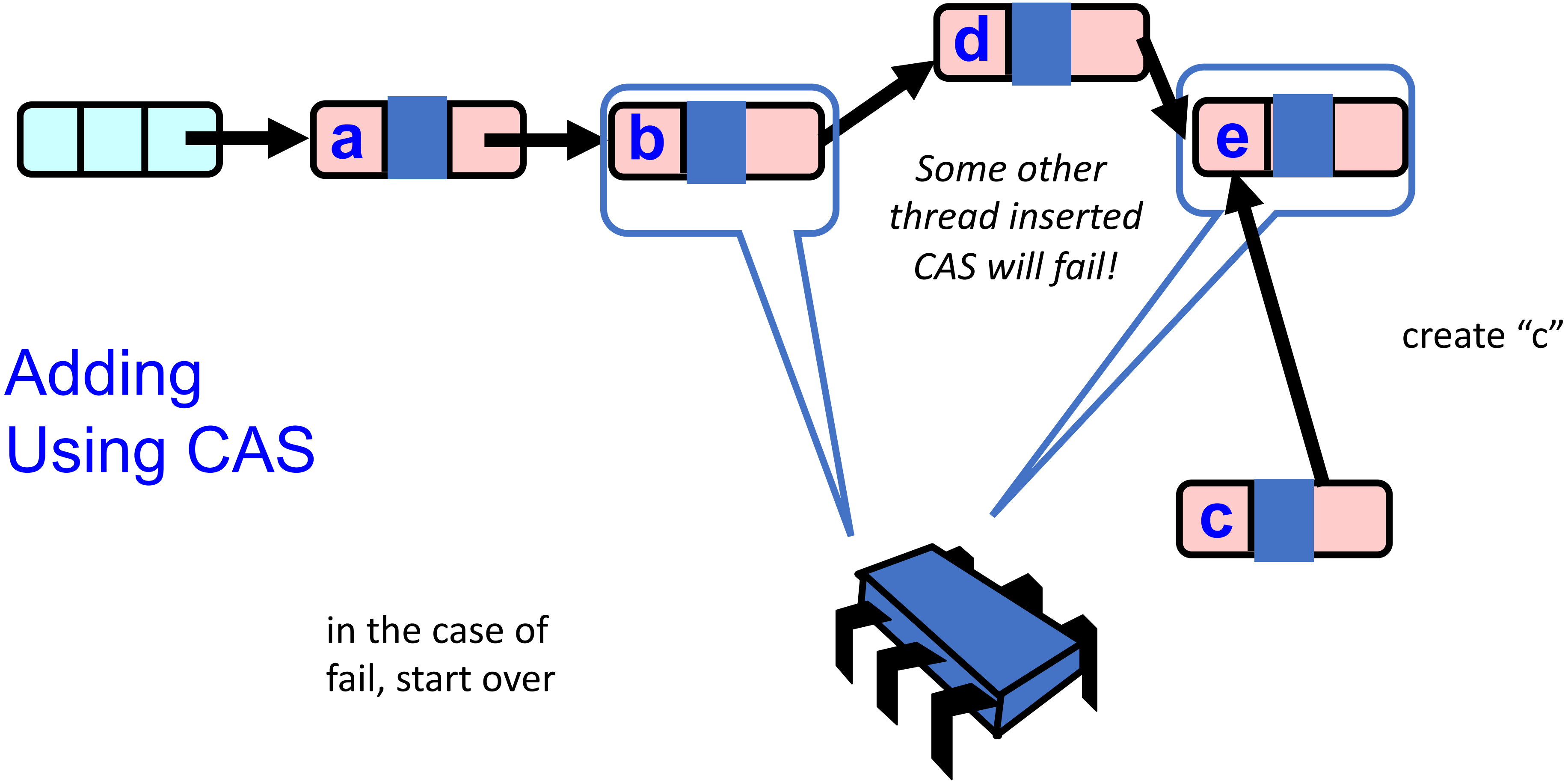
Find the location

Cache your insertion point!

b.next == e

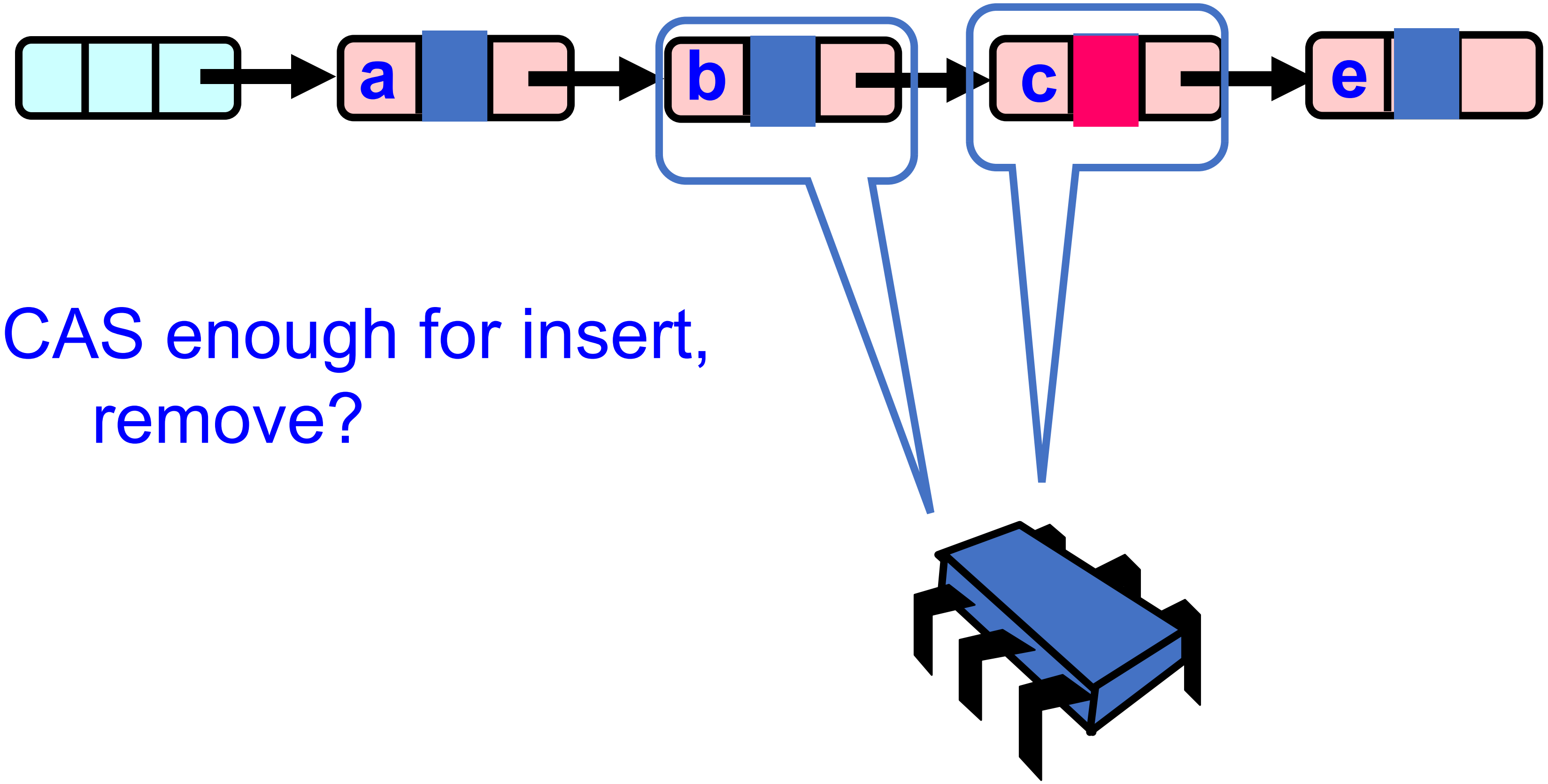
*notion is being abused here: e and c will be node **

Adding Using CAS



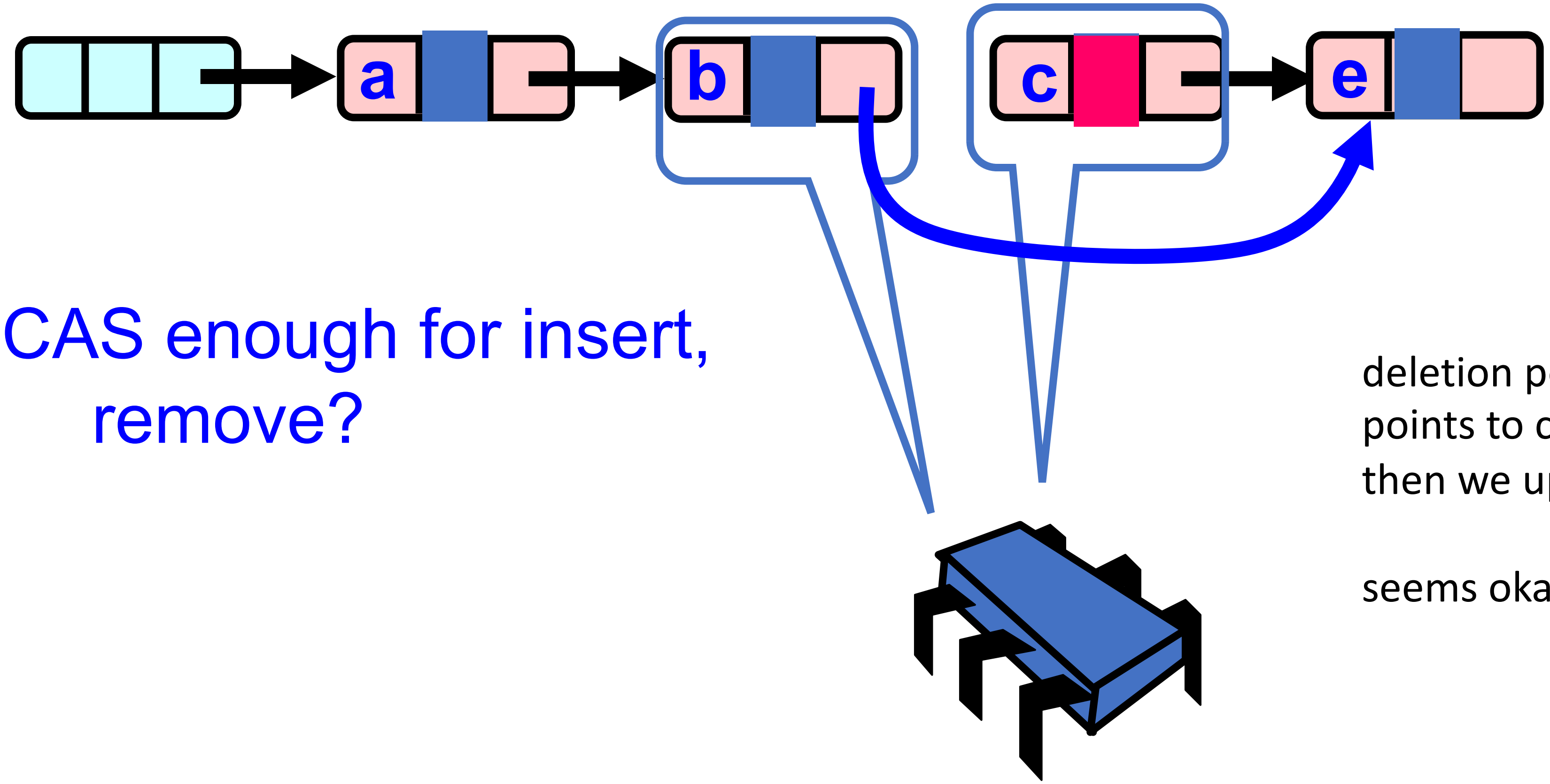
in the case of fail, start over

Lock-free Lists



CAS enough for insert,
remove?

Lock-free Lists



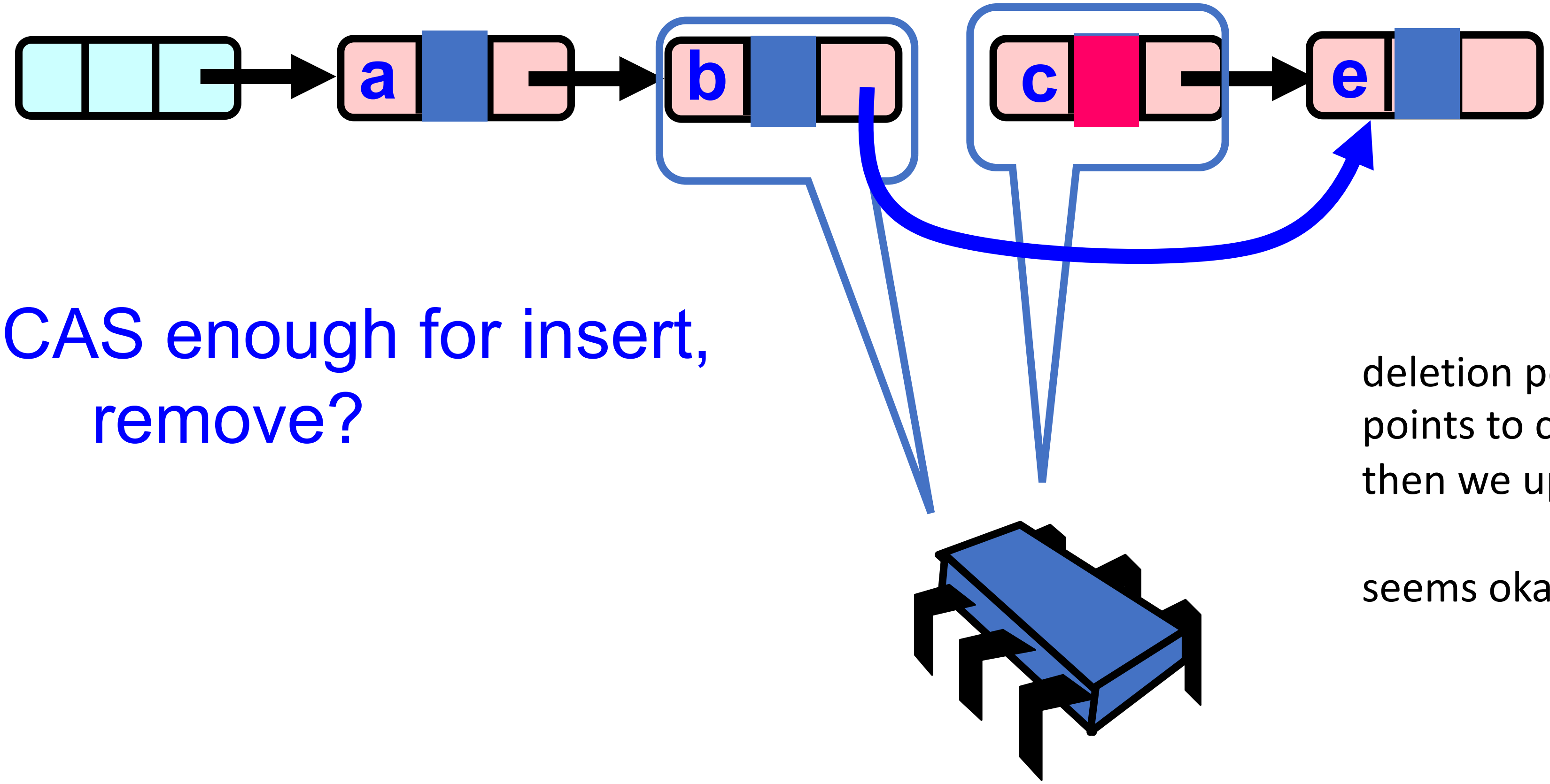
CAS enough for insert, remove?

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

seems okay...

Lock-free Lists

ensures that nobody has inserted a node between b and c



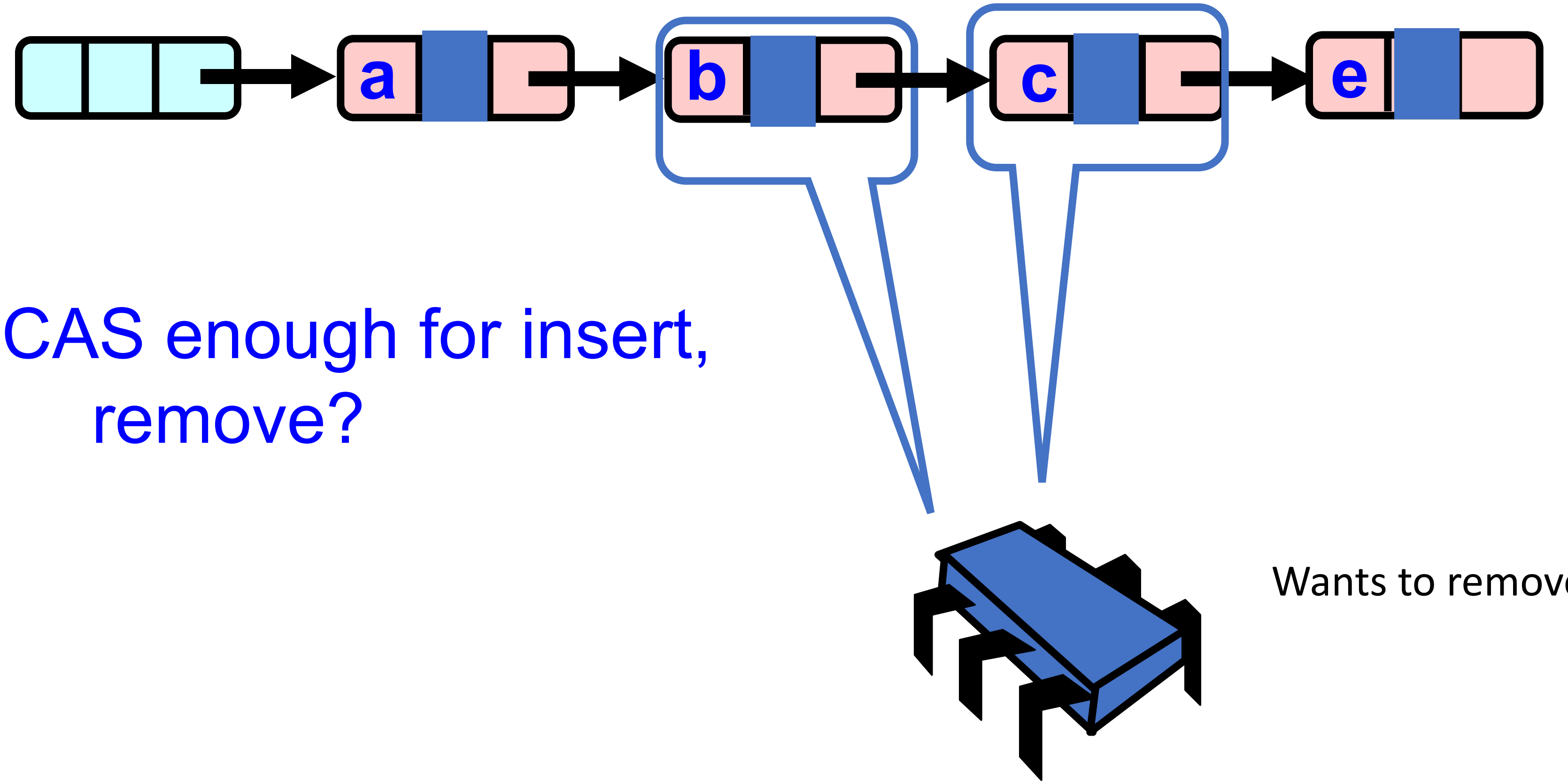
CAS enough for insert, remove?

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

seems okay...

Lock-free Lists

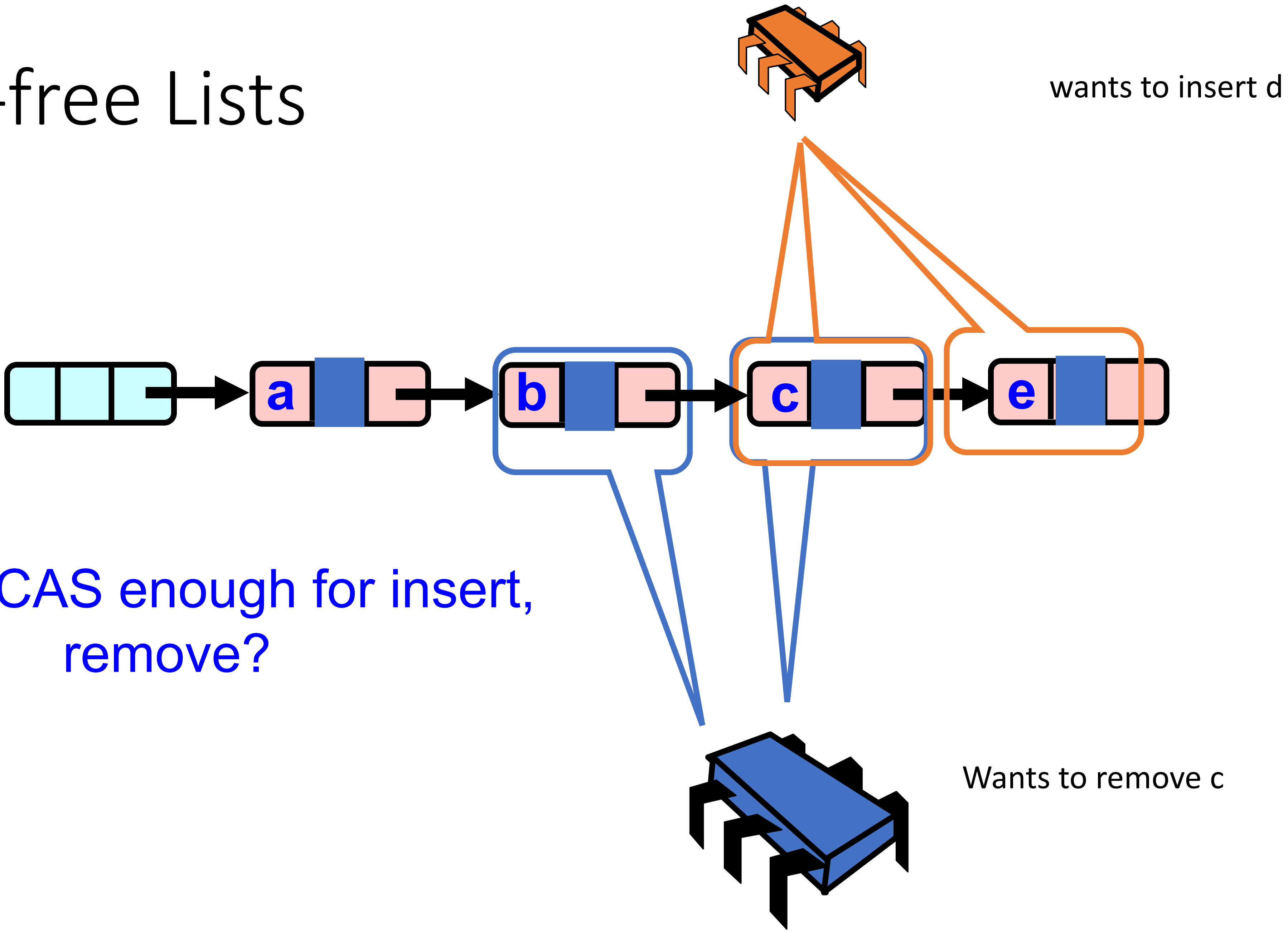
Rewind



CAS enough for insert, remove?

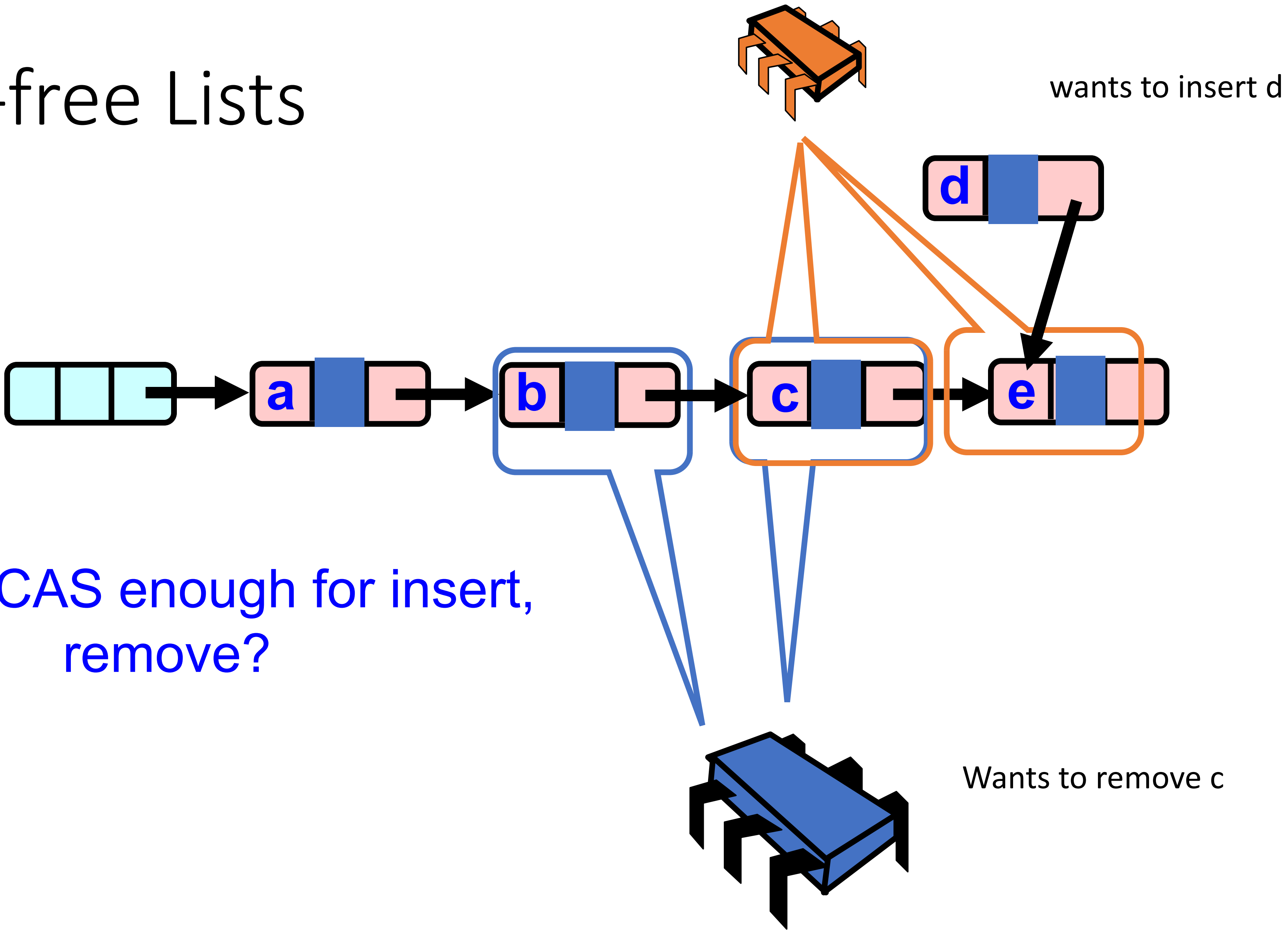
Wants to remove c

Lock-free Lists



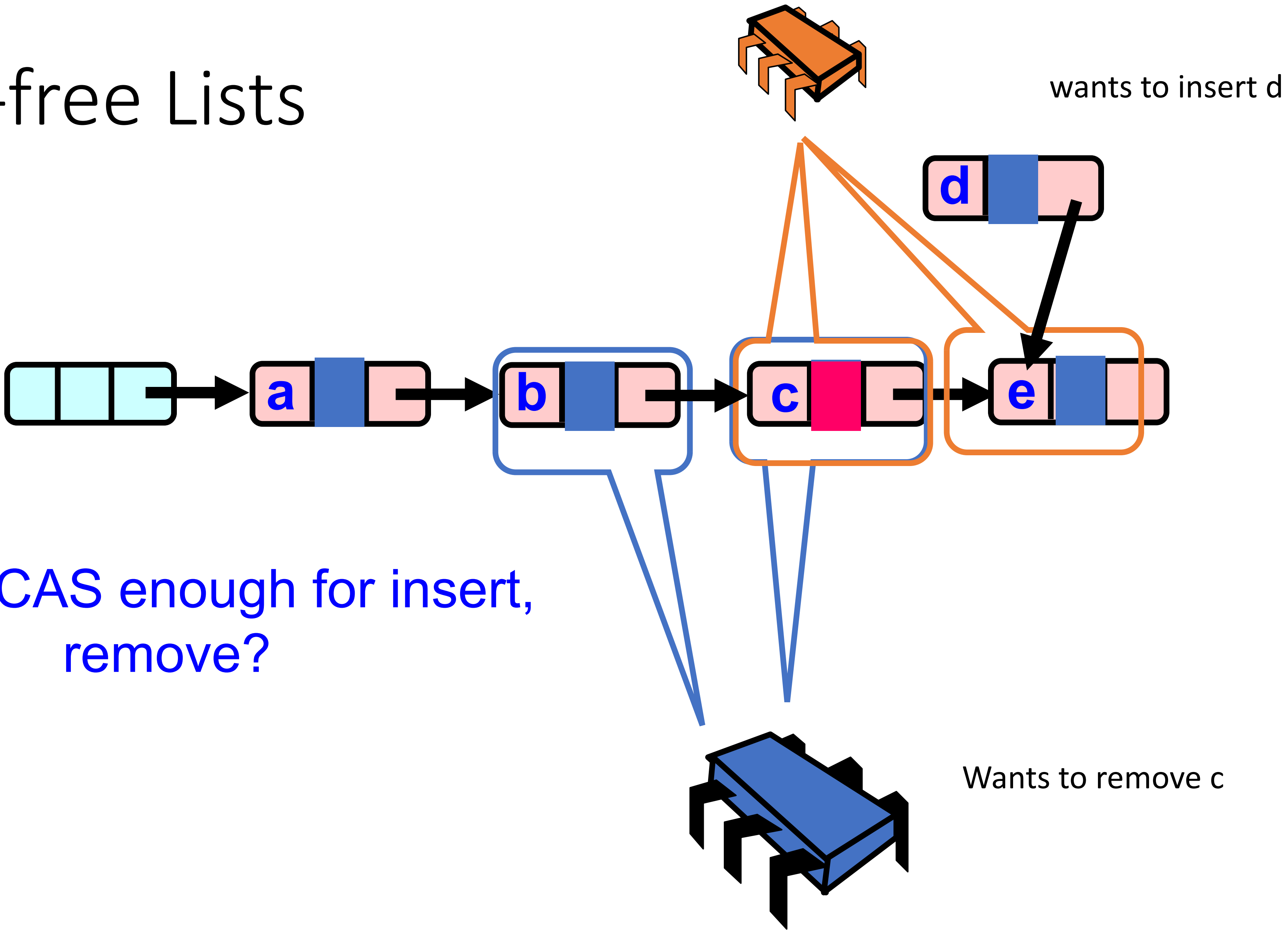
CAS enough for insert, remove?

Lock-free Lists



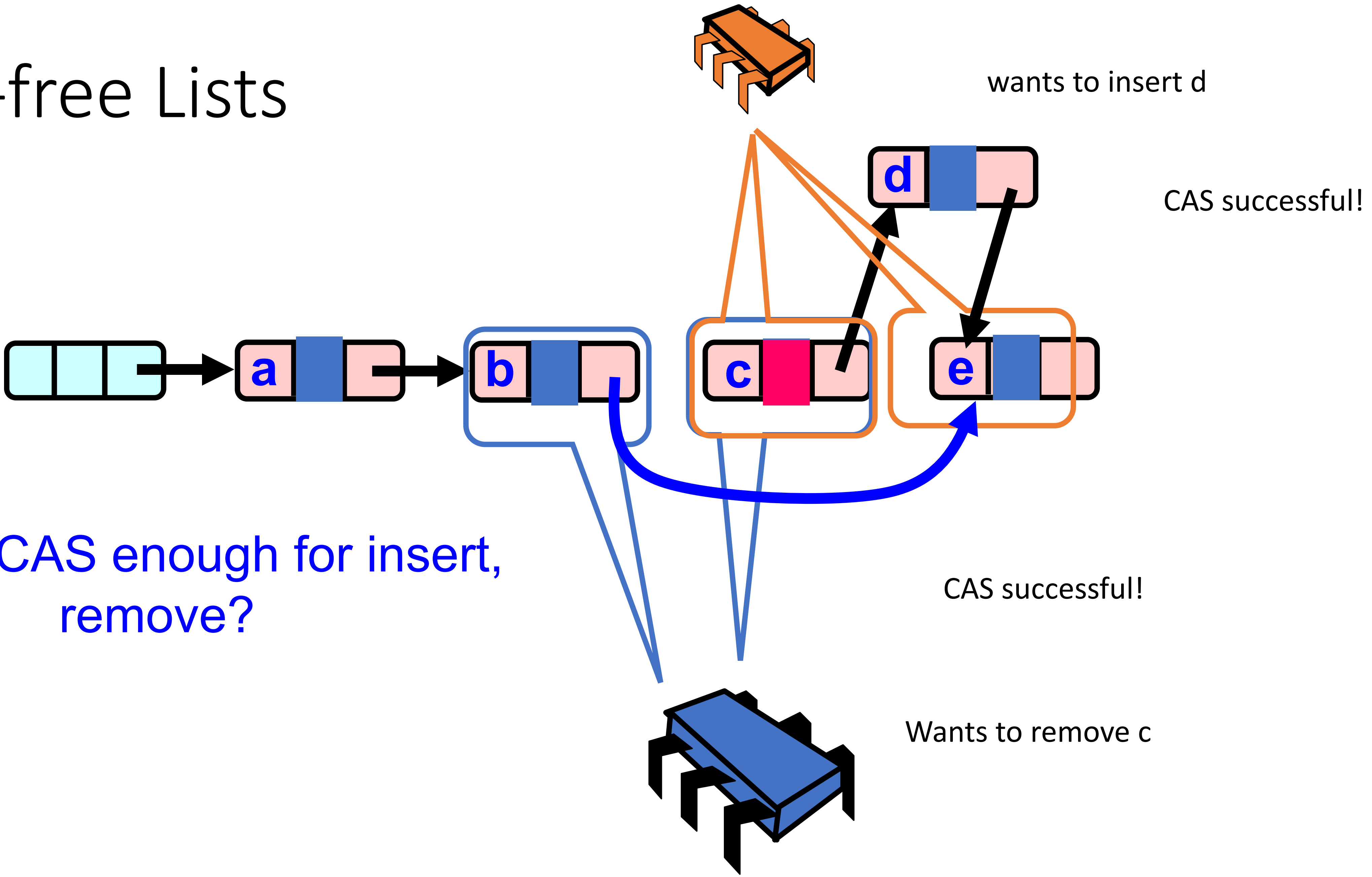
CAS enough for insert, remove?

Lock-free Lists



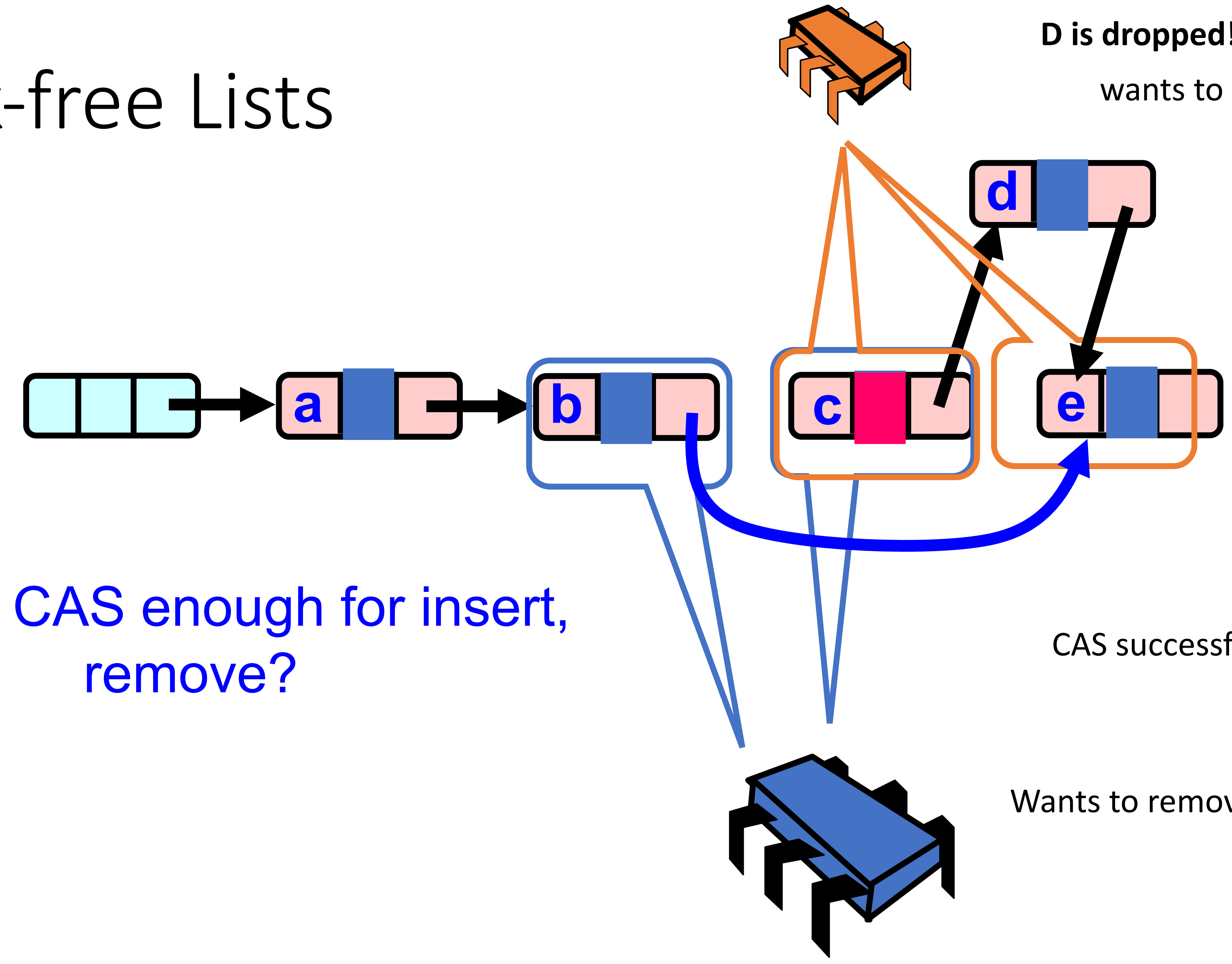
CAS enough for insert, remove?

Lock-free Lists



CAS enough for insert, remove?

Lock-free Lists



D is dropped!

wants to insert d

CAS successful!

CAS enough for insert, remove?

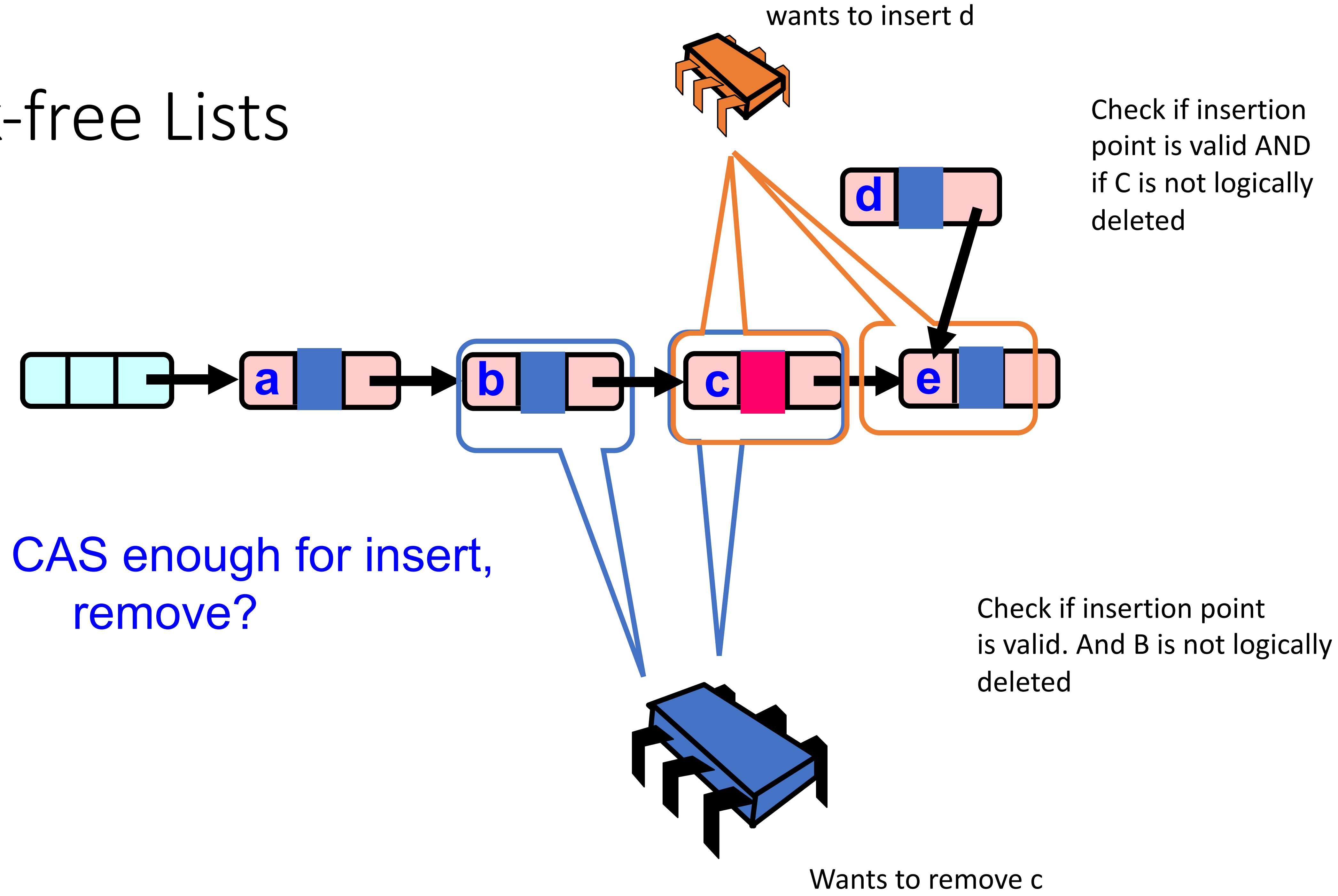
CAS successful!

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 logically removed.

Lock-free Lists



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!)

See you next time!

- Work on HW 3

- Keep an eye out for midterm grades