

CS-206 Concurrency

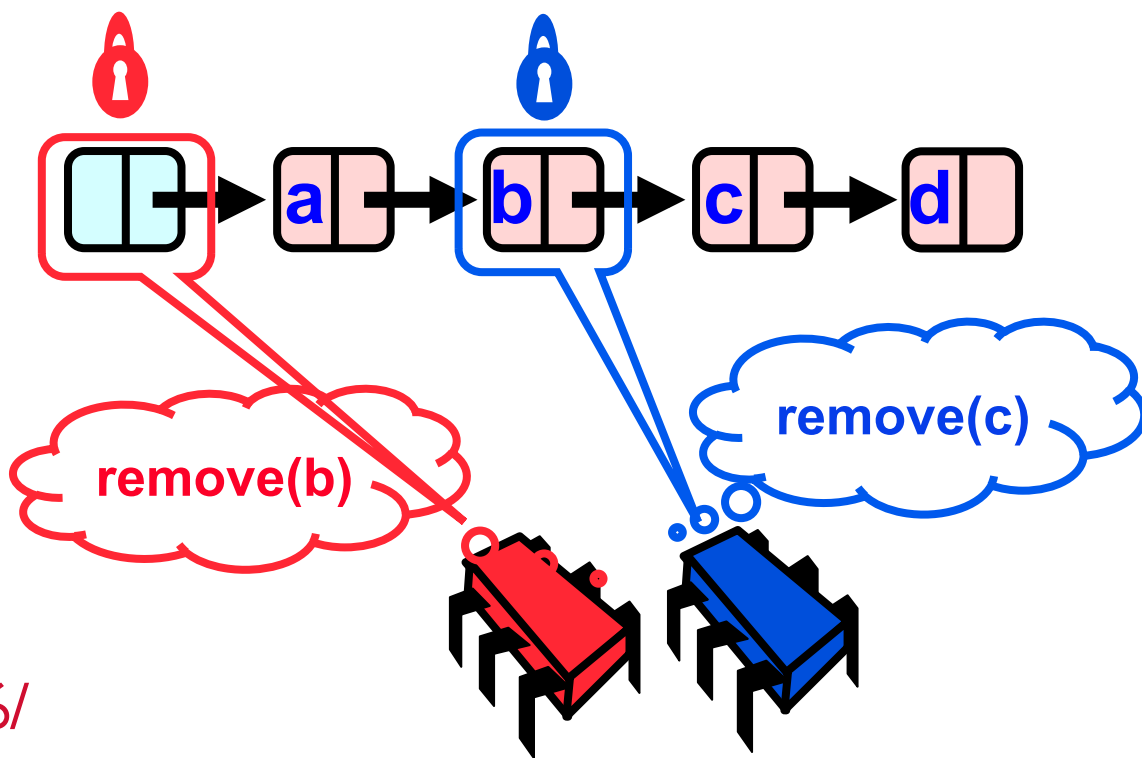
Lecture 8

Concurrent Data structures

Spring 2015

Prof. Babak Falsafi

parsa.epfl.ch/courses/cs206/



Adapted from slides originally developed by Maurice Herlihy and Nir Shavit from the Art of Multiprocessor Programming, and Babak Falsafi
EPFL Copyright 2015

Where are We?

Lecture & Lab				
M	T	W	T	F
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb
23-Feb	24-Feb	25-Feb	26-Feb	27-Feb
2-Mar	3-Mar	4-Mar	5-Mar	6-Mar
9-Mar	10-Mar	11-Mar	12-Mar	13-Mar
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar
23-Mar	24-Mar	25-Mar	26-Mar	27-Mar
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr
27-Apr	28-Apr	29-Apr	30-Apr	1-May
4-May	5-May	6-May	7-May	8-May
11-May	12-May	13-May	14-May	15-May
18-May	19-May	20-May	21-May	22-May
25-May	26-May	27-May	28-May	29-May

- ▶ Concurrent data structures
 - ▷ Coarse-grained locking
 - ▷ Fine-grained locking
 - ▷ Lock-free data structures
- ▶ Examples
 - ▷ Linked lists
- ▶ Next week
 - ▷ Higher-level data structures

Contention

- ▶ When many threads compete for a lock
- ▶ Prevents efficient multithreaded execution
 - ▷ Threads spend more time waiting for lock than doing work
- ▶ Real problem in multiprocessor programming

Today: Concurrent Objects

- ▶ Adding threads should not lower throughput
 - ▷ Contention effects

- ▶ Should increase throughput
 - ▷ Not possible if inherently sequential
 - ▷ Surprising things are parallelizable

Coarse-Grained Synchronization

- ▶ Each method locks the object
 - ▷ Avoid contention using queue locks
 - ▷ Easy to reason about
 - ▷ In simple cases

- ▶ Example
 - ▷ Solaris (Oracle's OS) first version had a single lock
 - ▷ Every time there was an OS access, one thread could get in
 - ▷ What is wrong with this picture?

Coarse-Grained Synchronization

- ▶ Sequential bottleneck
 - ▷ Threads “stand in line”

- ▶ Adding more threads
 - ▷ Does not improve throughput
 - ▷ Struggle to keep it from getting worse

- ▶ So why even use a multiprocessor?
 - ▷ Well, some apps are inherently parallel ...

Fine-Grained Synchronization

- ▶ Instead of using a single lock ...
- ▶ Split object into
 - ▷ Independently-synchronized components
- ▶ Methods conflict when they access
 - ▷ The same component ...
 - ▷ At the same time

Example with Linked List

- ▶ Illustrate these patterns ...
- ▶ Using a list-based Set
 - ▷ Common application
 - ▷ Building block for other apps

List-Based Sets

```
public interface Set<T> {  
    public boolean add(T x);  
    public boolean remove(T x);  
    public boolean contains(T x);  
}
```

List-Based Sets

```
public interface Set<T> {  
    public boolean add(T x);  
    public boolean remove(T x);  
    public boolean contains(T x);  
}
```

Add item to set

List-Based Sets

```
public interface Set<T> {  
    public boolean add(T x);  
public boolean remove(T x);  
    public boolean contains(T x);  
}
```

Remove item from set

List-Based Sets

```
public interface Set<T> {  
    public boolean add(T x);  
    public boolean remove(T x);  
    public boolean contains(T x);  
}
```

Is item in set?

List Node

```
public class Node {  
    public T item;  
    public int key;  
    public Node next;  
}
```

List Node

```
public class Node {  
    public T item;  
    public int key;  
    public Node next;  
}
```



item of interest

List Node

```
public class Node {  
    public T item;  
    public int key;  
    public Node next;  
}
```

Usually hash code

List Node

```
public class Node {  
    public T item;  
    public int key;  
    public Node next;  
}
```

Reference to next node

Reasoning about Concurrent Objects

- ▶ Invariant

- ▷ Property that always holds

- ▶ Why do we care about invariants?

- ▷ Invariant is true when object is **created**

- ▷ Invariant truth is **preserved** by each method

- ▷ Each **step** of each method

Specifically ...

- ▶ Invariants preserved by
 - ▷ **add ()**
 - ▷ **remove ()**
 - ▷ **contains ()**

- ▶ Example invariants to preserve for linked lists
 - ▷ Tail reachable from `head`
 - ▷ Sorted
 - ▷ No duplicates

Sequential List Based Set

add()

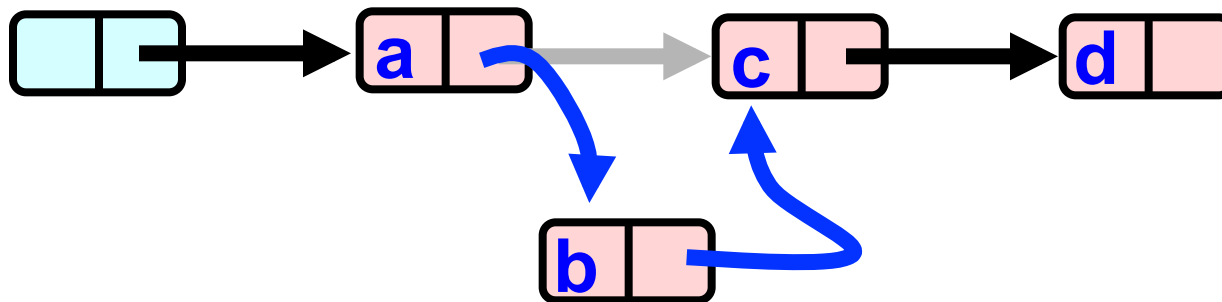


remove()

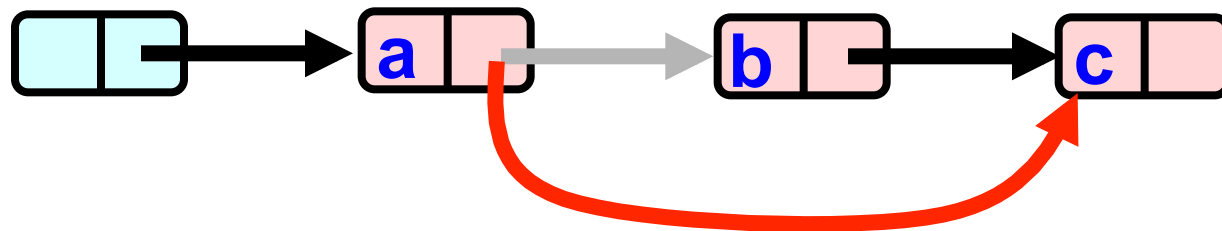


Sequential List Based Set

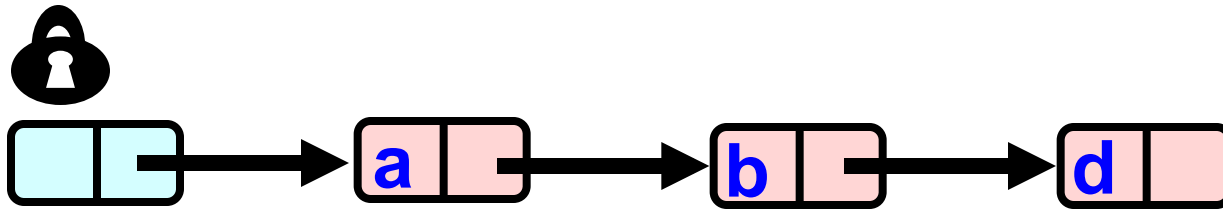
add()



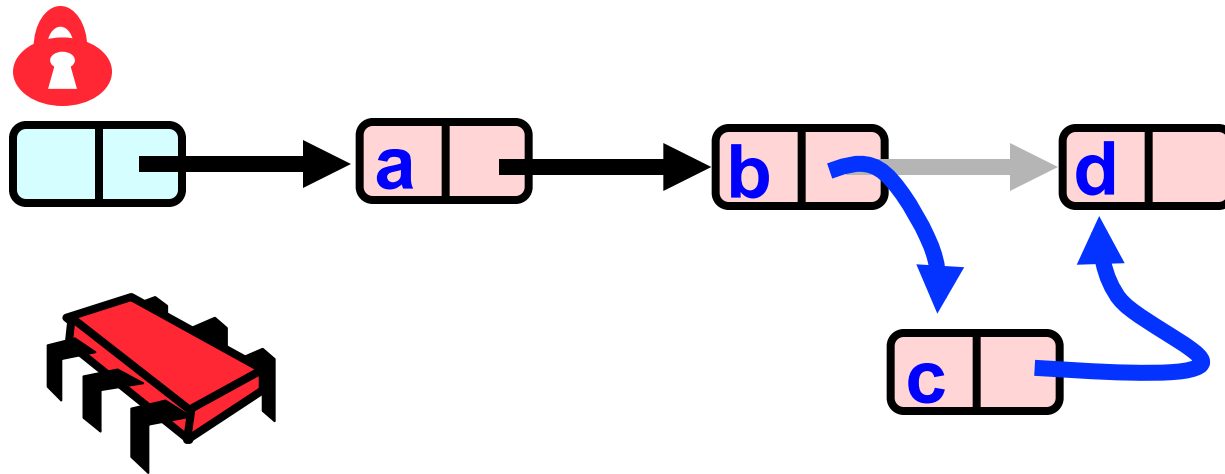
remove()



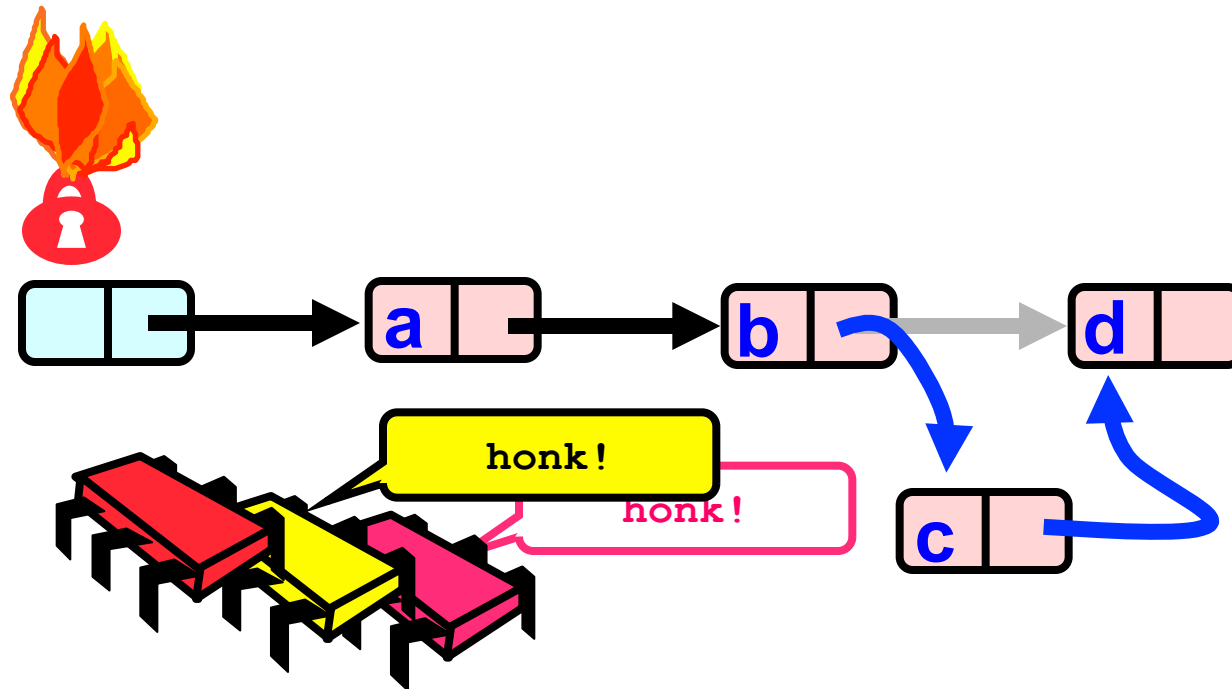
Coarse-Grained Locking



Coarse-Grained Locking



Coarse-Grained Locking



Simple but hotspot + bottleneck

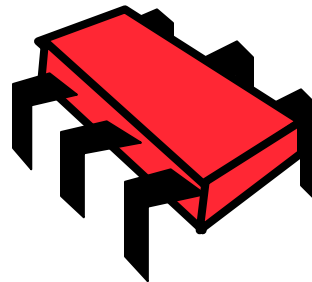
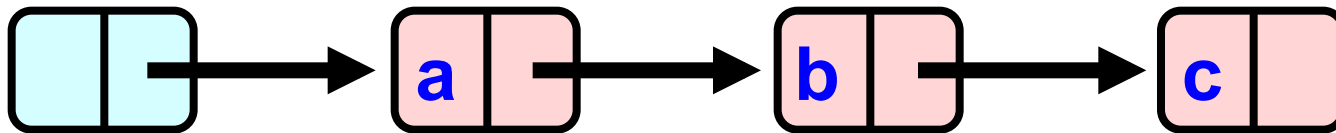
Coarse-Grained Locking

- ▶ Easy, same as synchronized methods
 - ▷ “One lock to rule them all ...”
- ▶ Simple, clearly correct
 - ▷ Deserves respect!
- ▶ Works poorly with contention
 - ▷ Queue locks help
 - ▷ But bottleneck still an issue

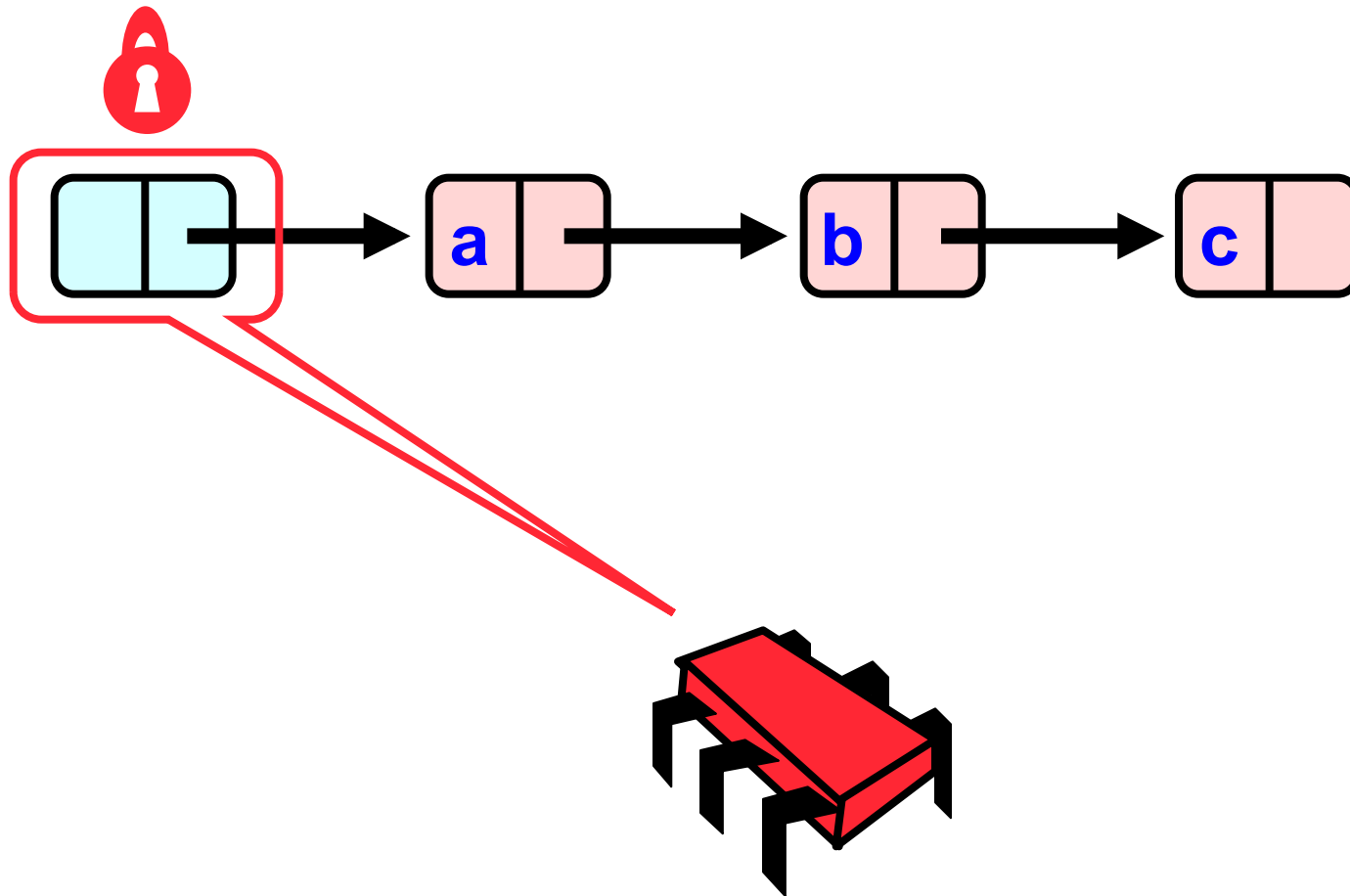
Fine-grained Locking

- ▶ Requires careful thought
 - ▷ “Do not meddle in the affairs of wizards, for they are subtle and quick to anger”
- ▶ Split object into pieces
 - ▷ Each piece has own lock
 - ▷ Methods that work on disjoint pieces need not exclude each other

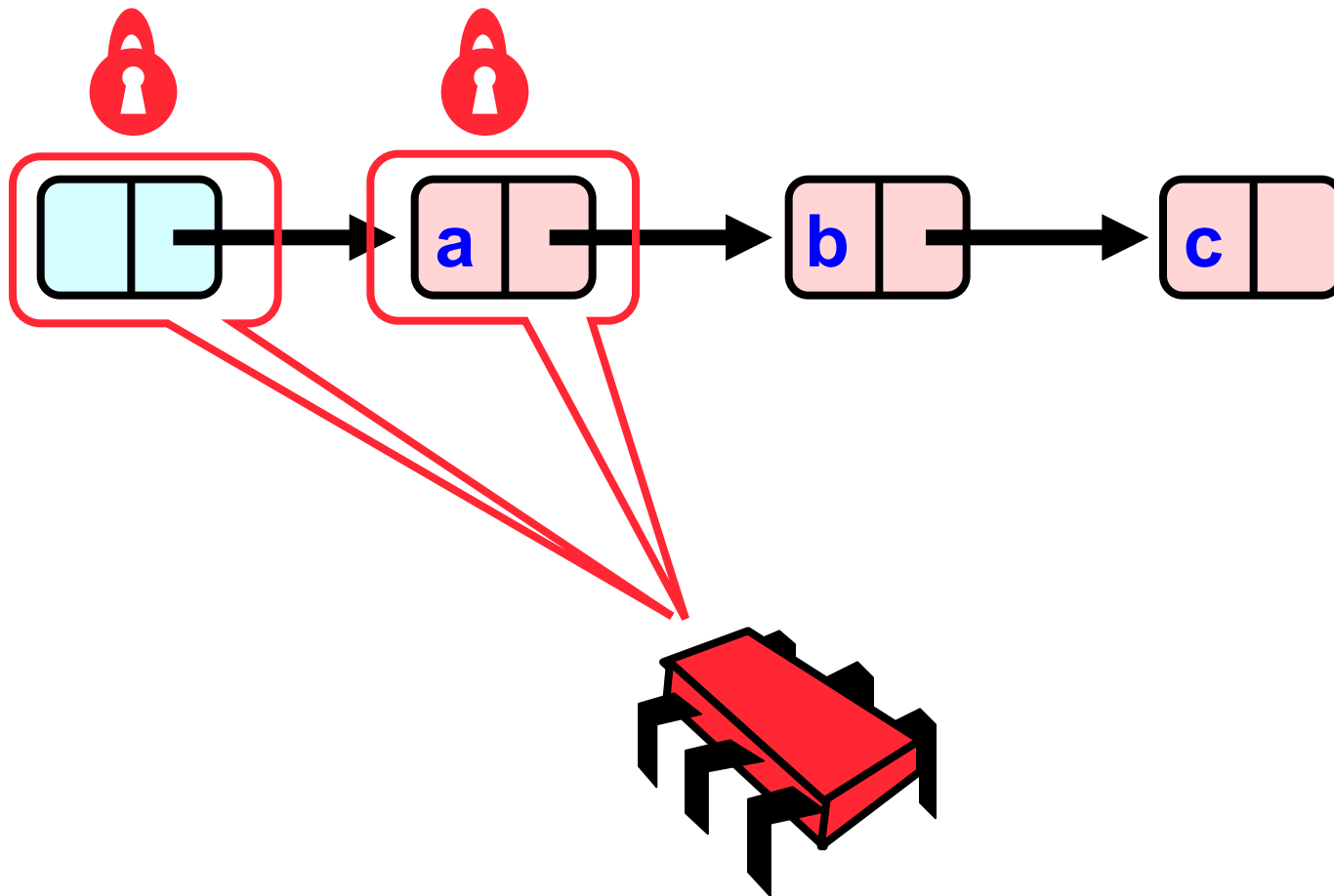
Hand-over-Hand locking



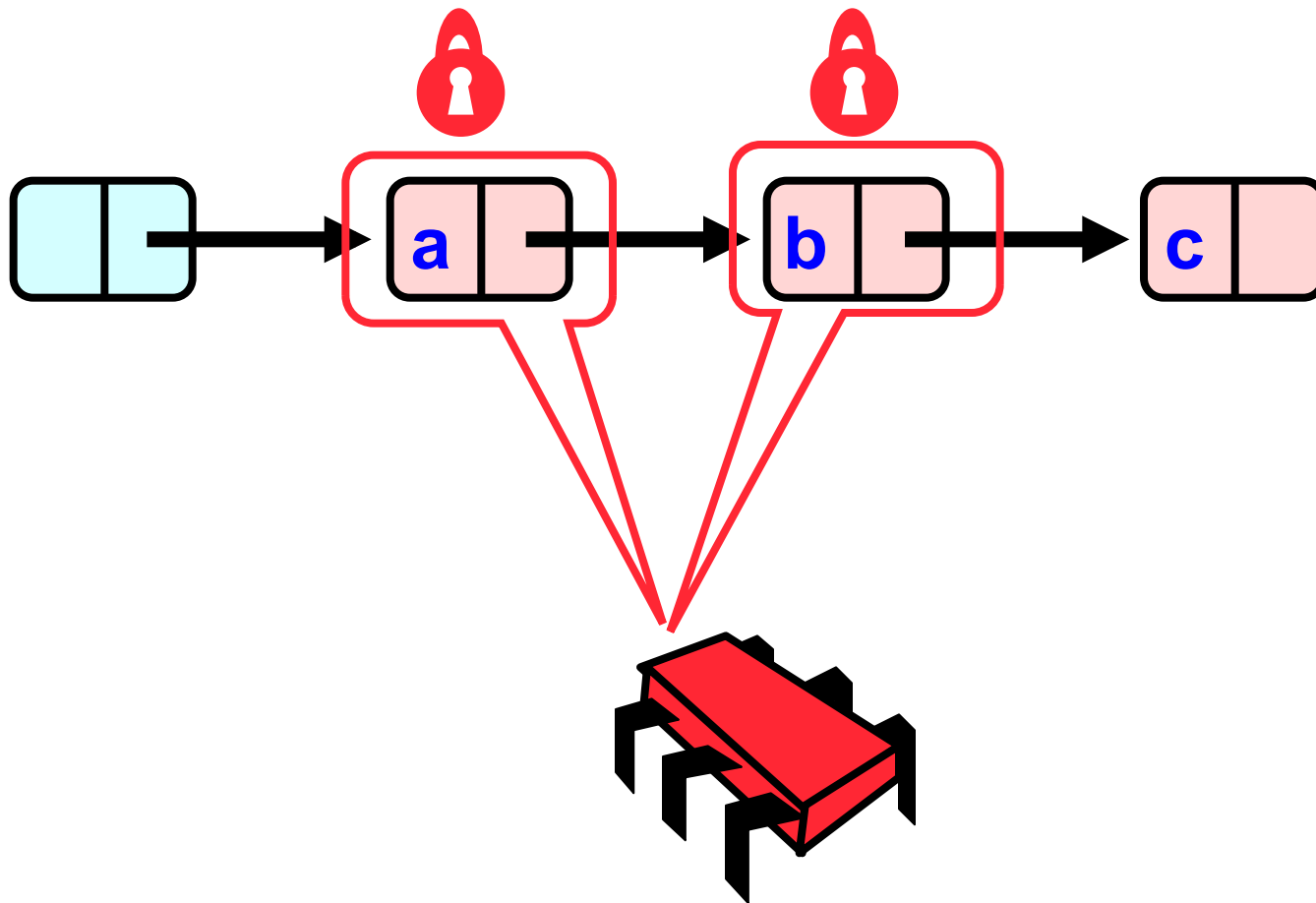
Hand-over-Hand locking



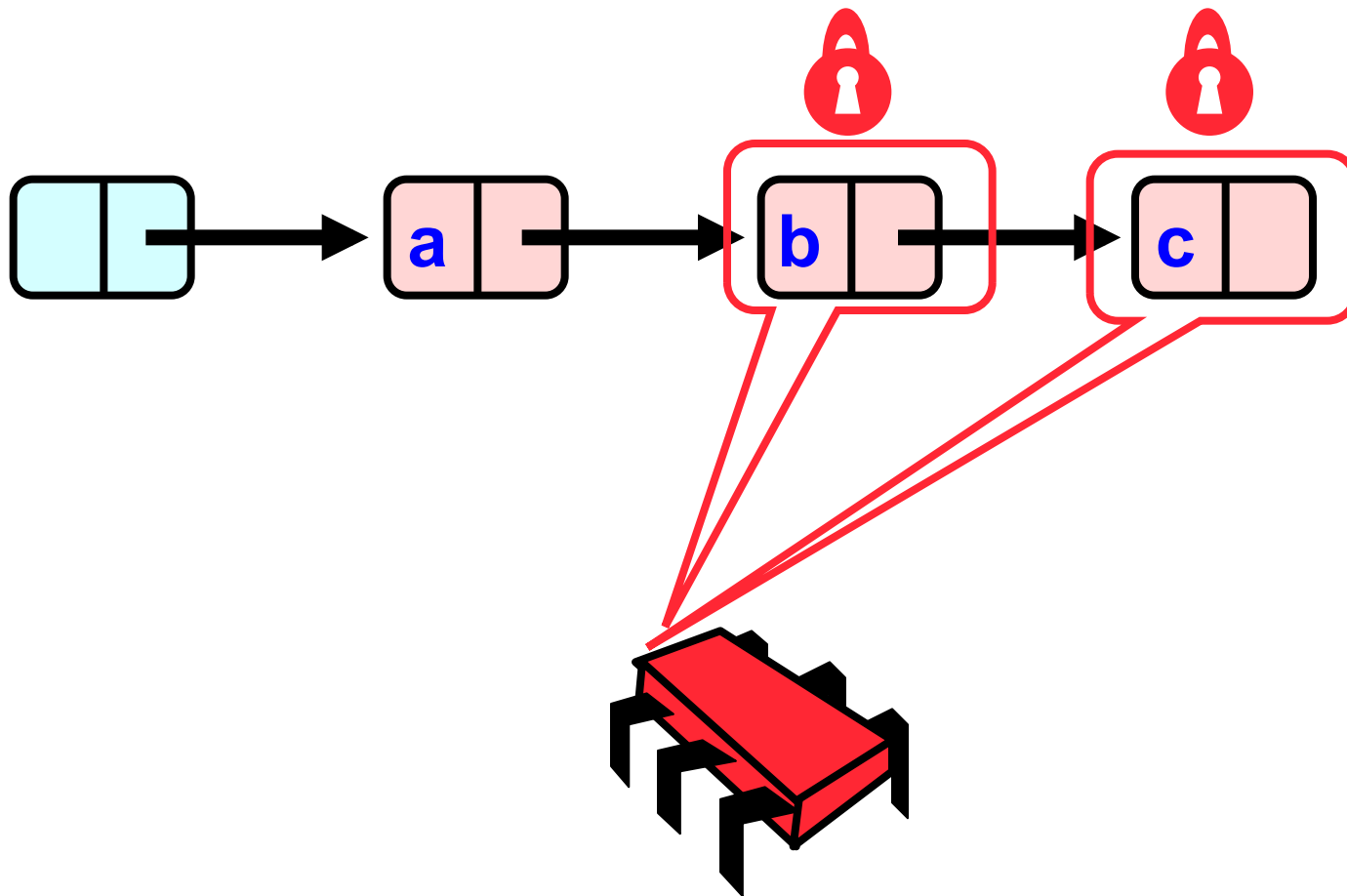
Hand-over-Hand locking



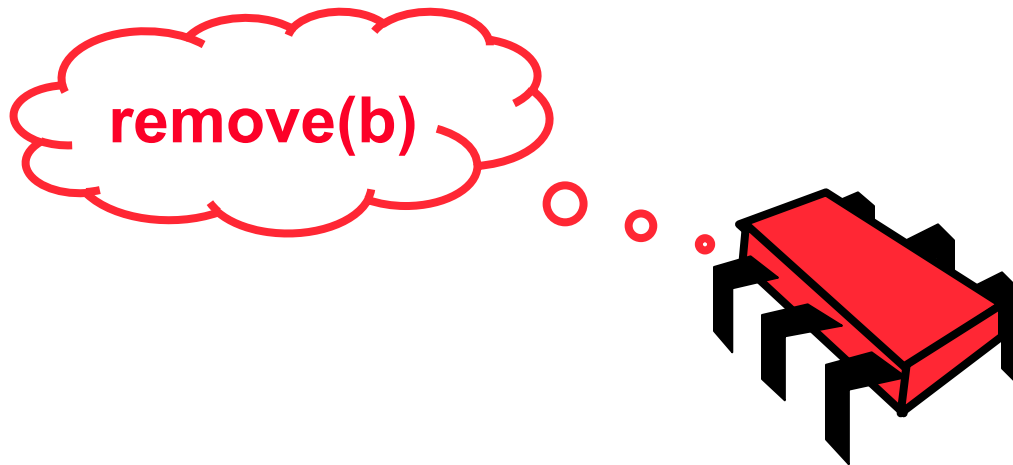
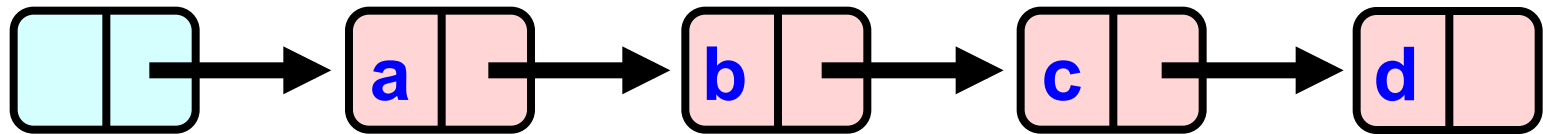
Hand-over-Hand locking



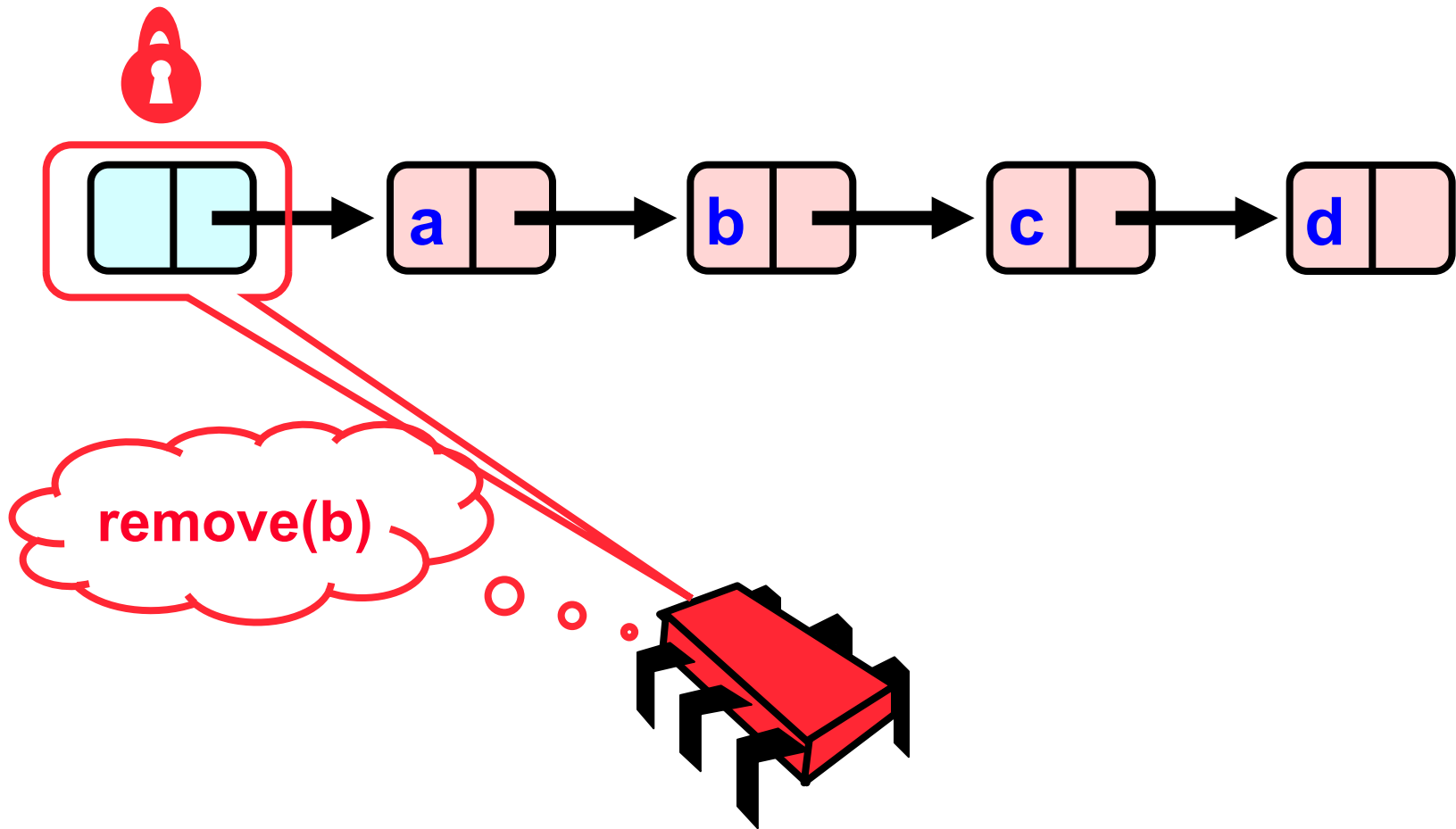
Hand-over-Hand locking



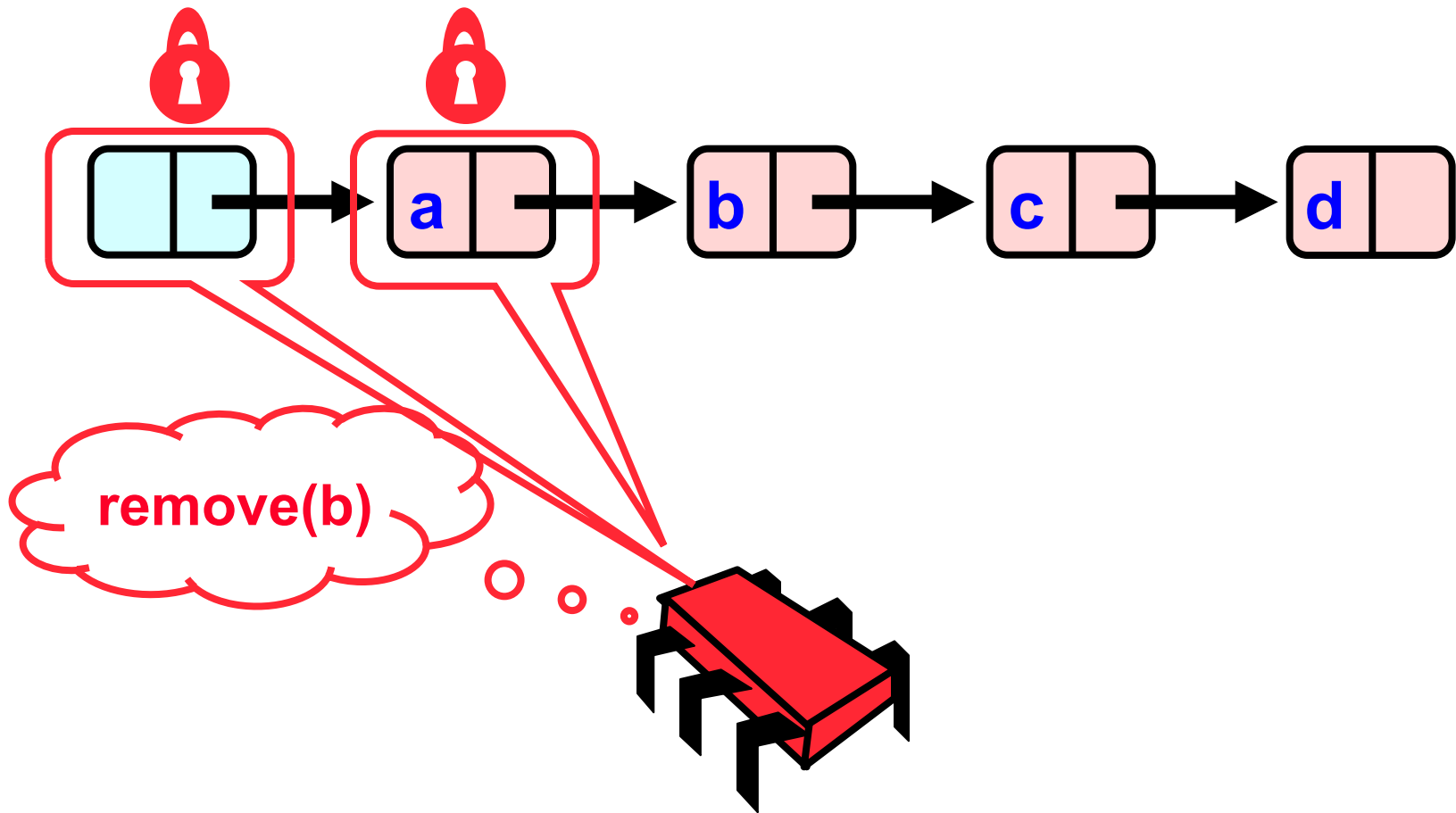
Removing a Node



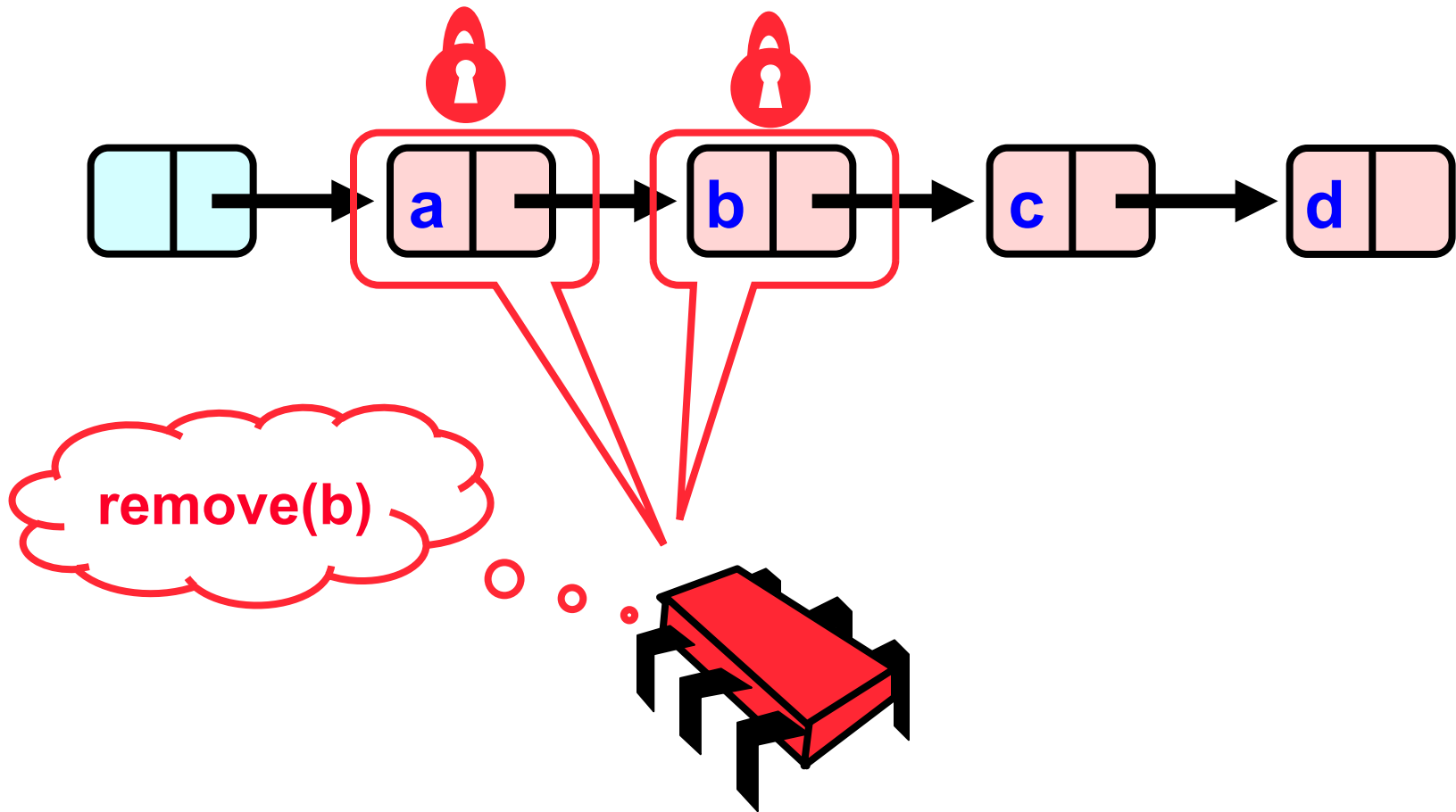
Removing a Node



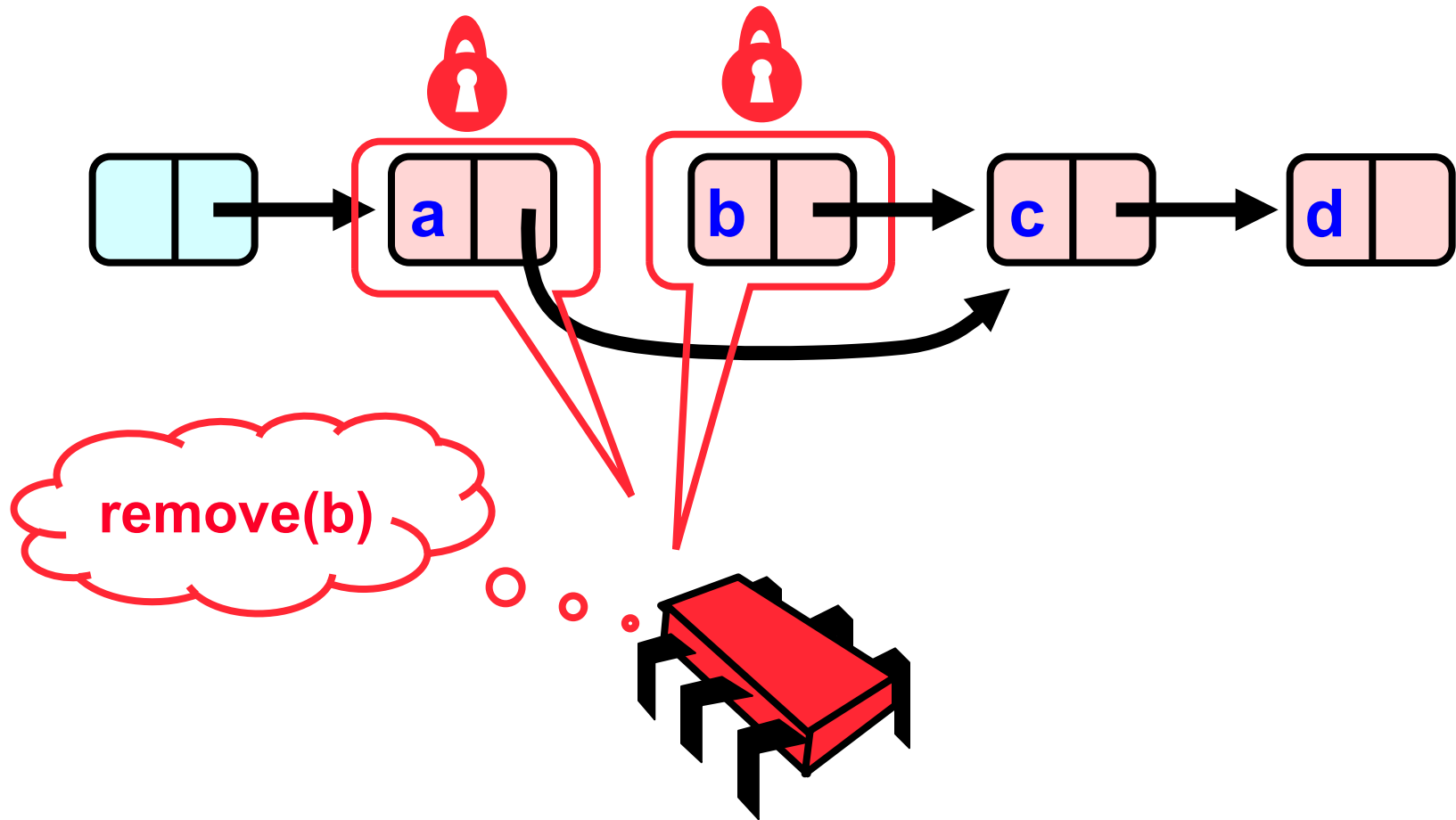
Removing a Node



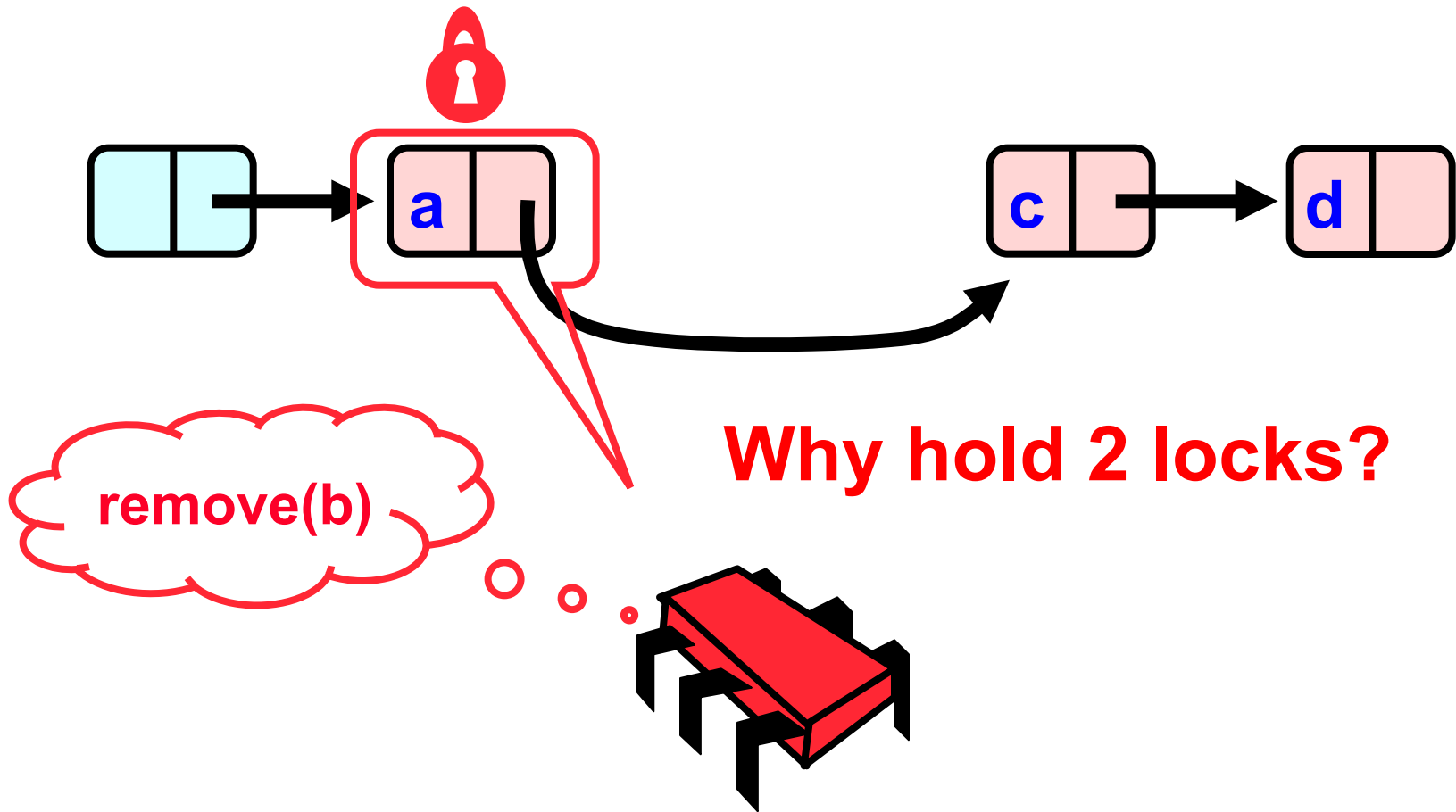
Removing a Node



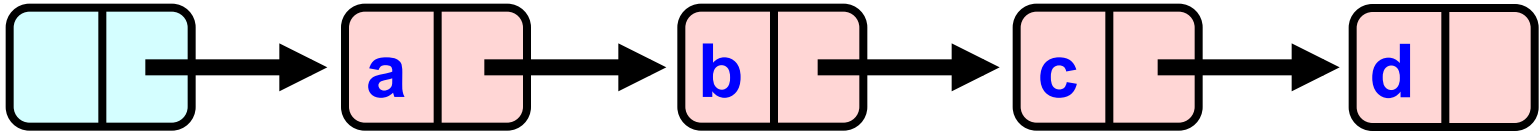
Removing a Node



Removing a Node

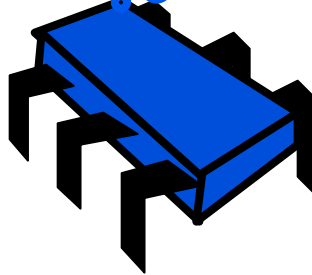
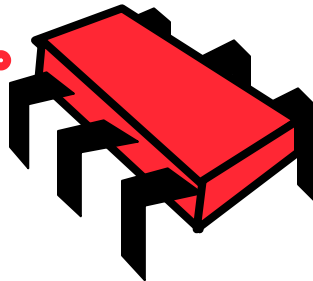


Concurrent Removes

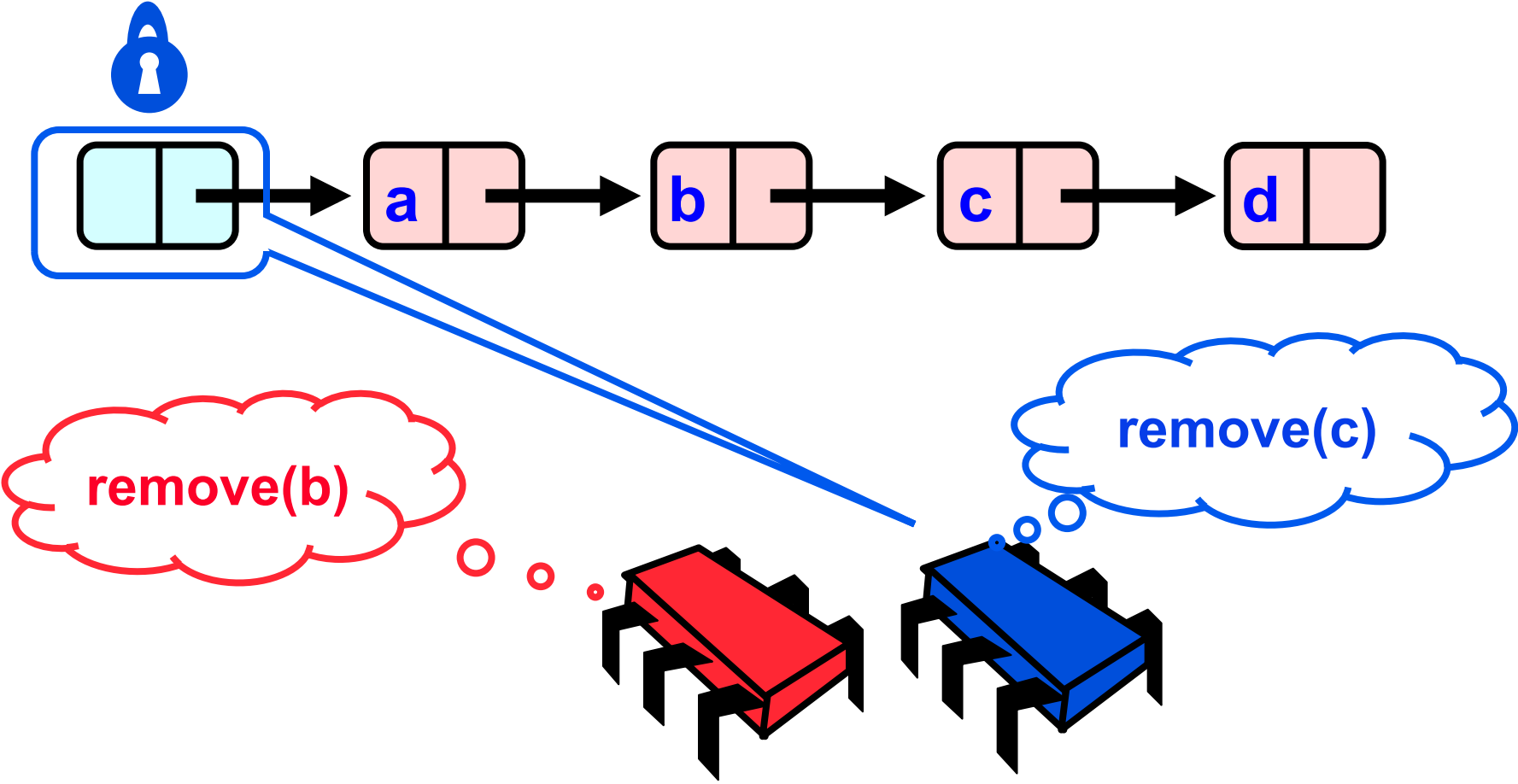


remove(b)

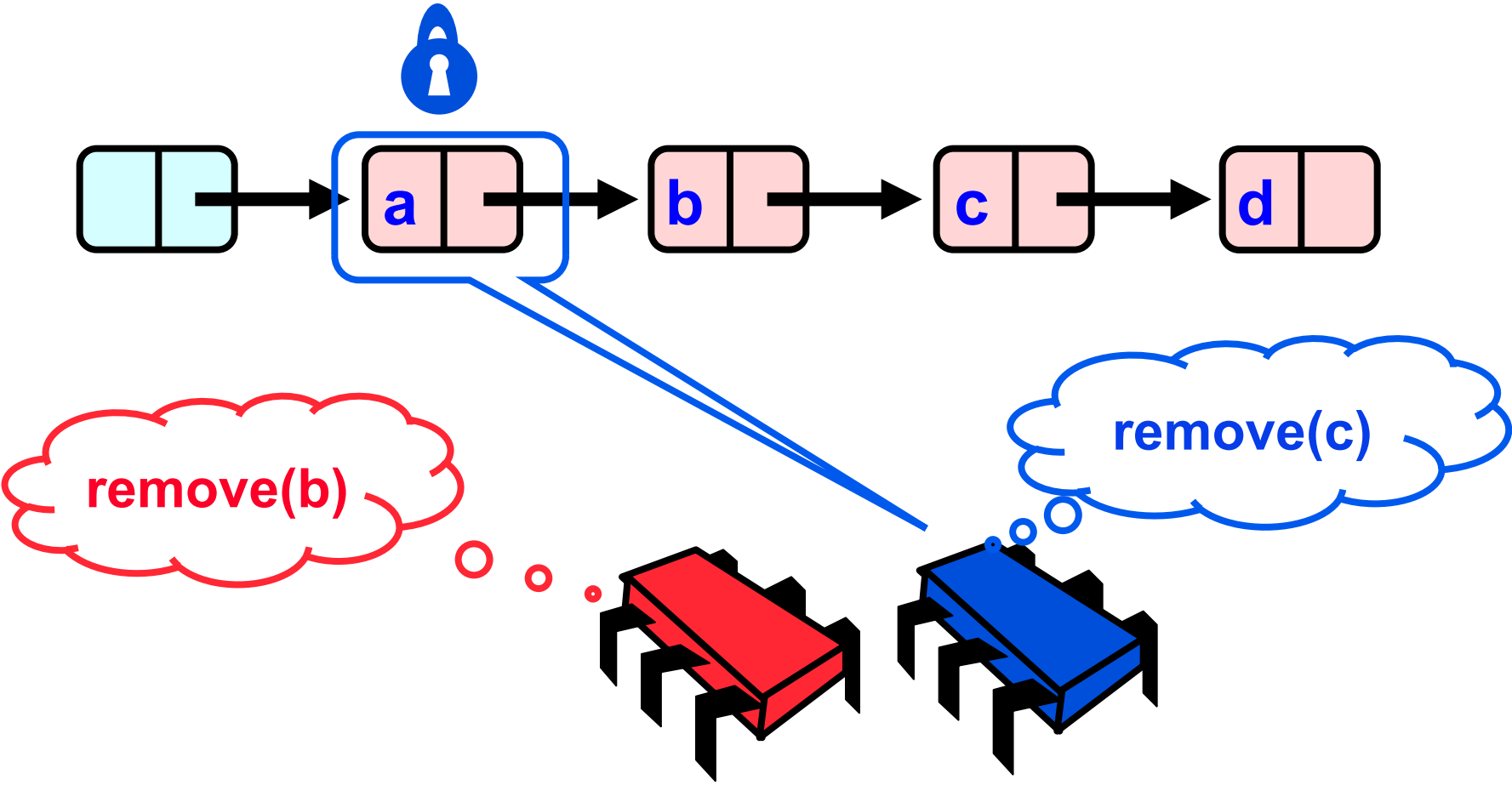
remove(c)



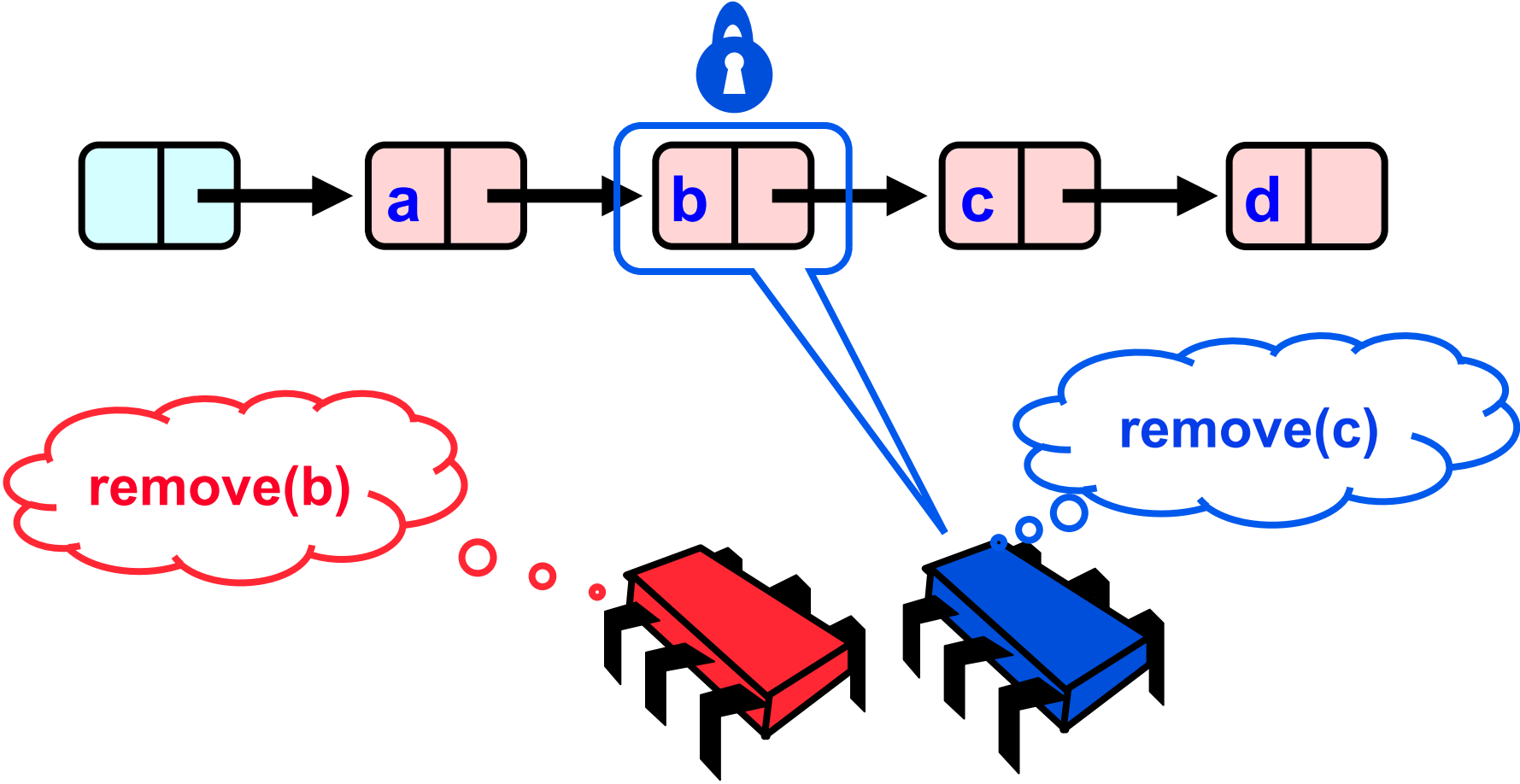
Concurrent Removes



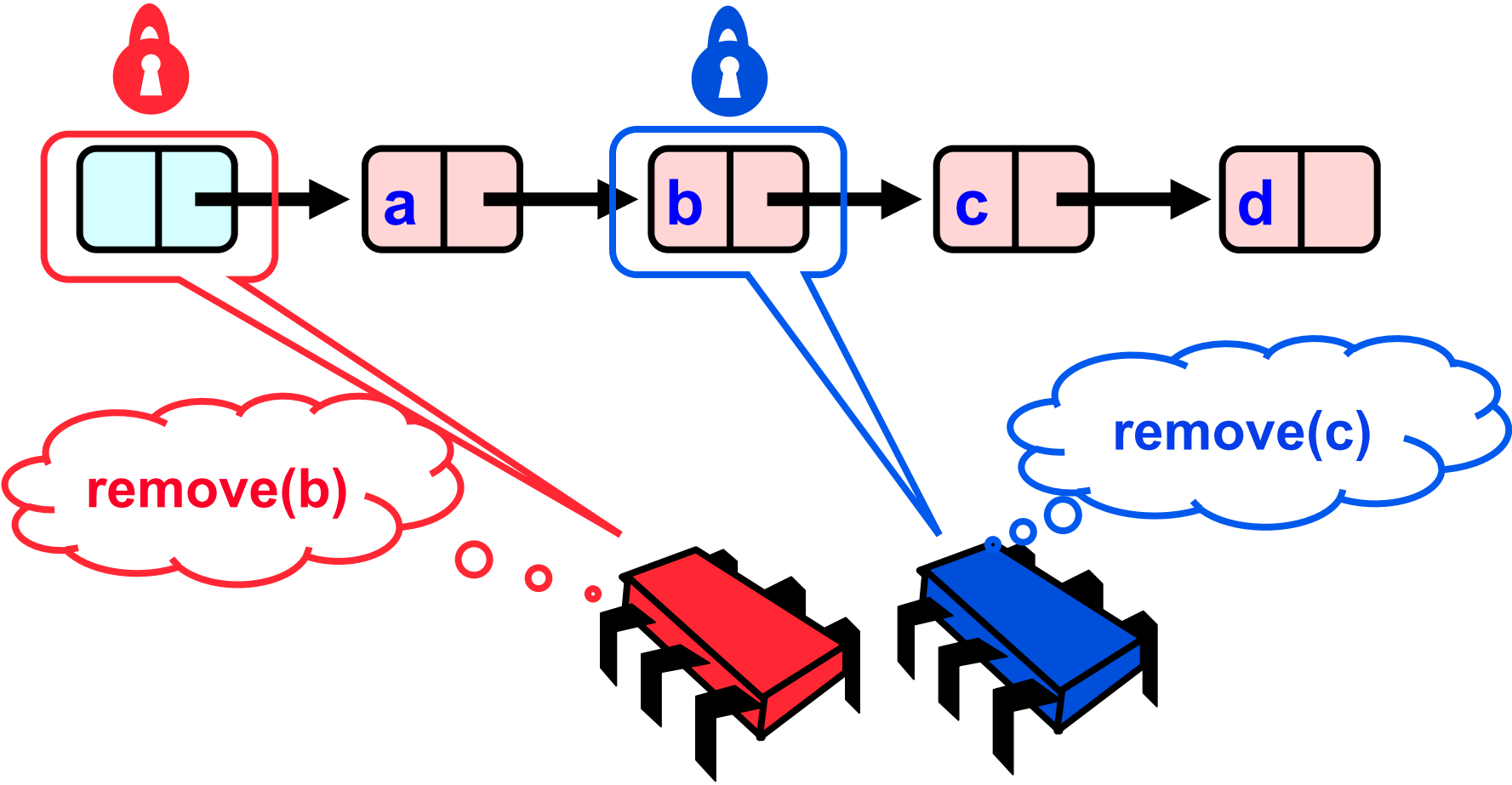
Concurrent Removes



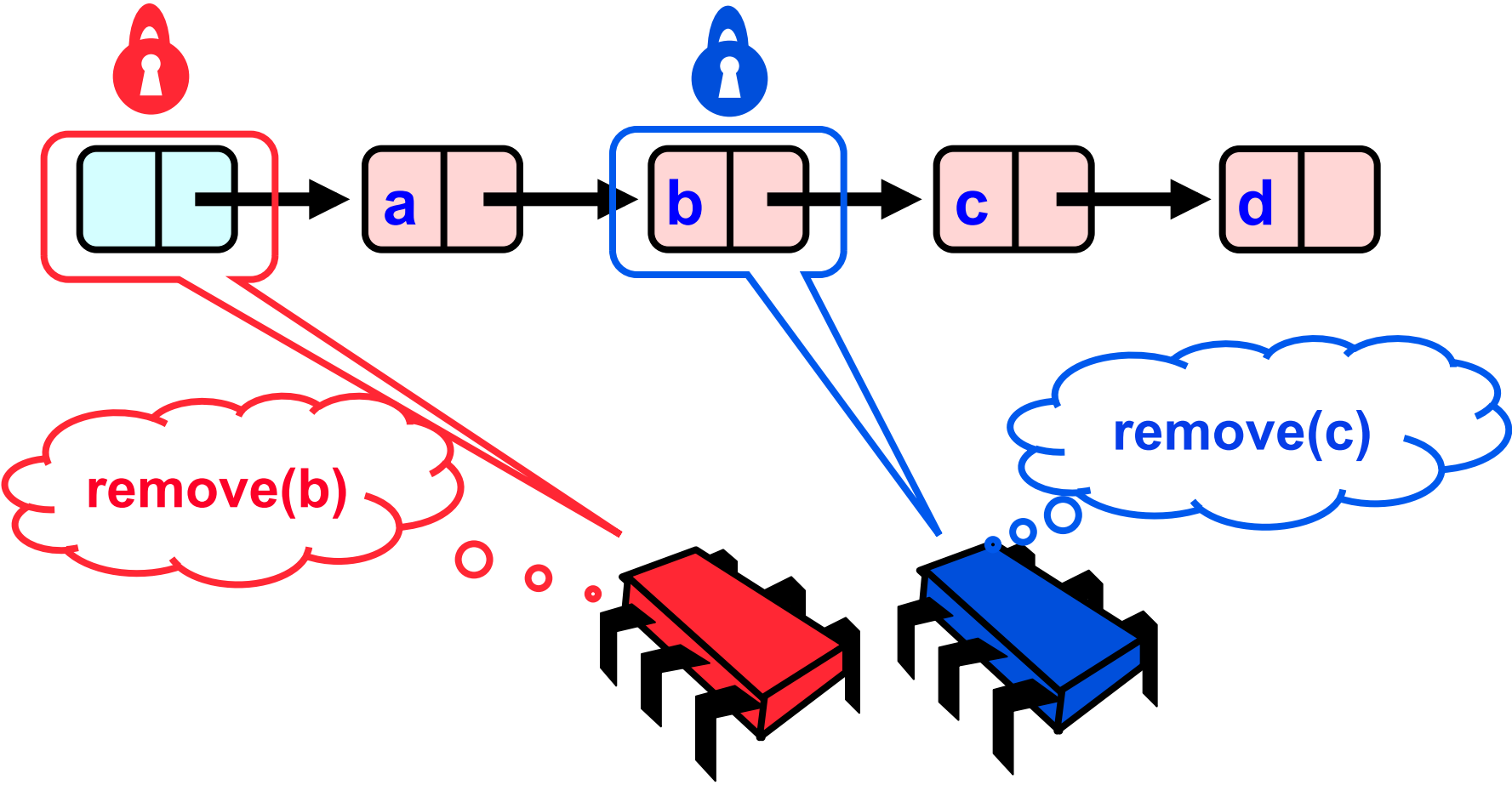
Concurrent Removes



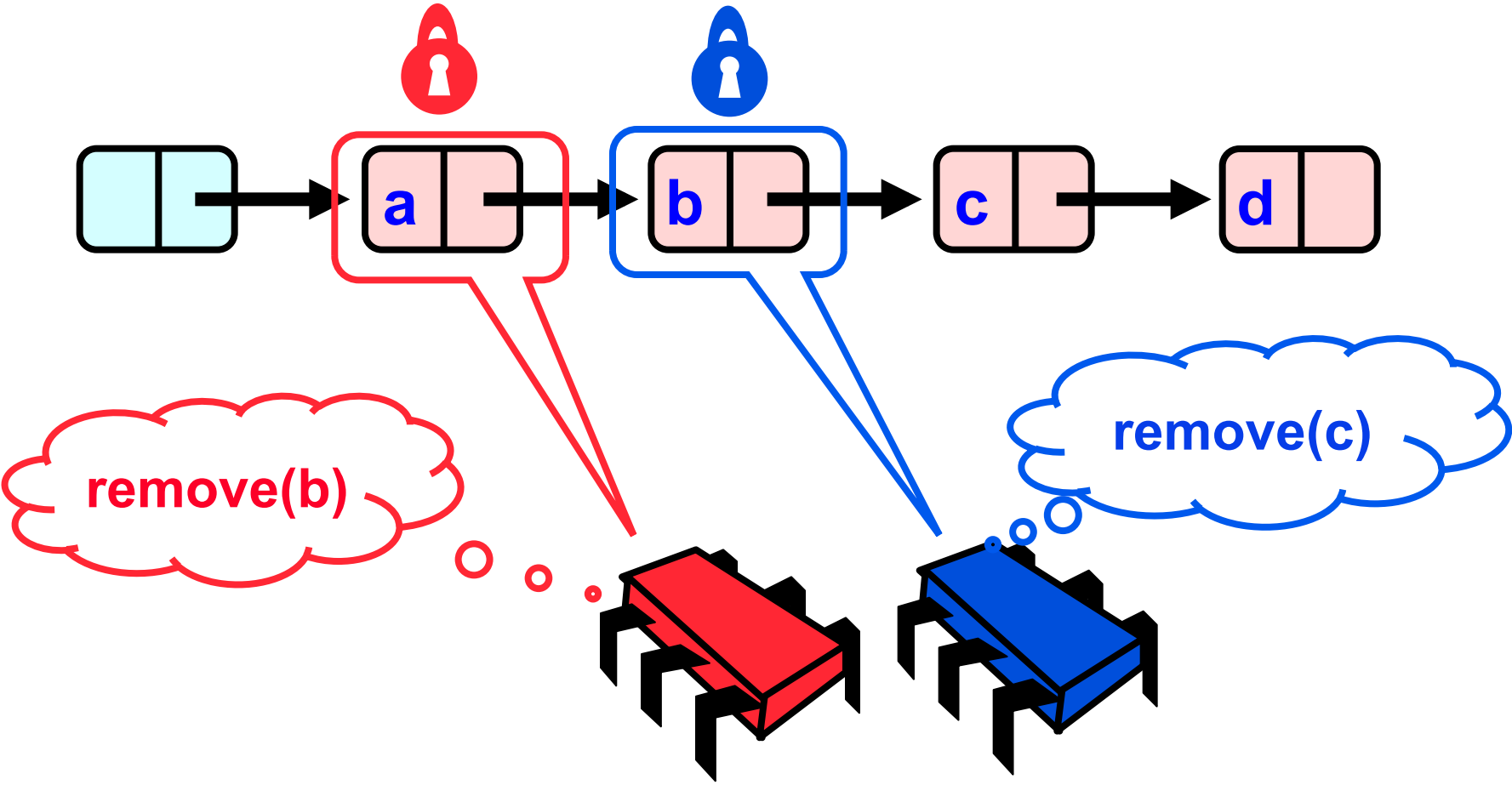
Concurrent Removes



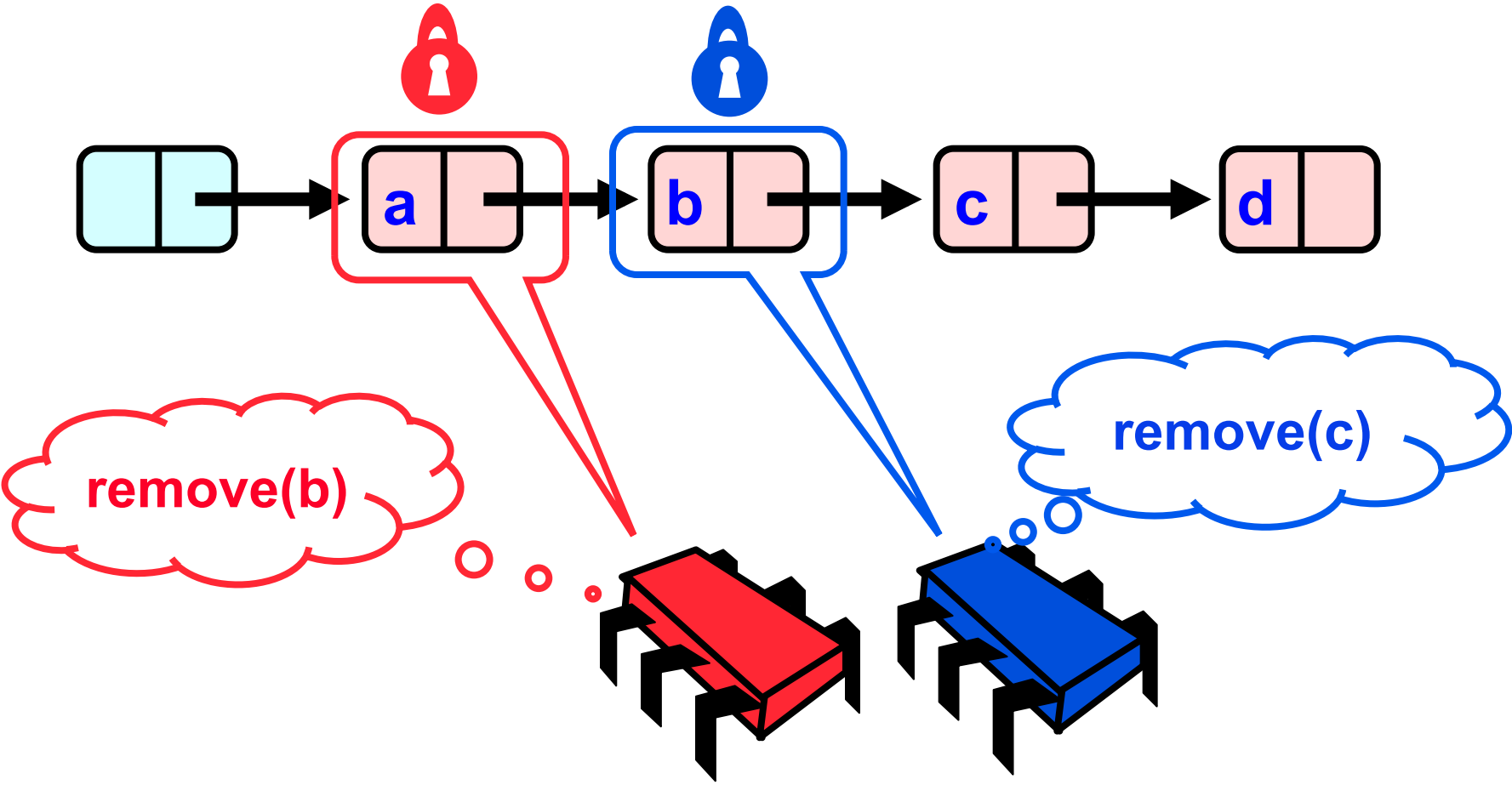
Concurrent Removes



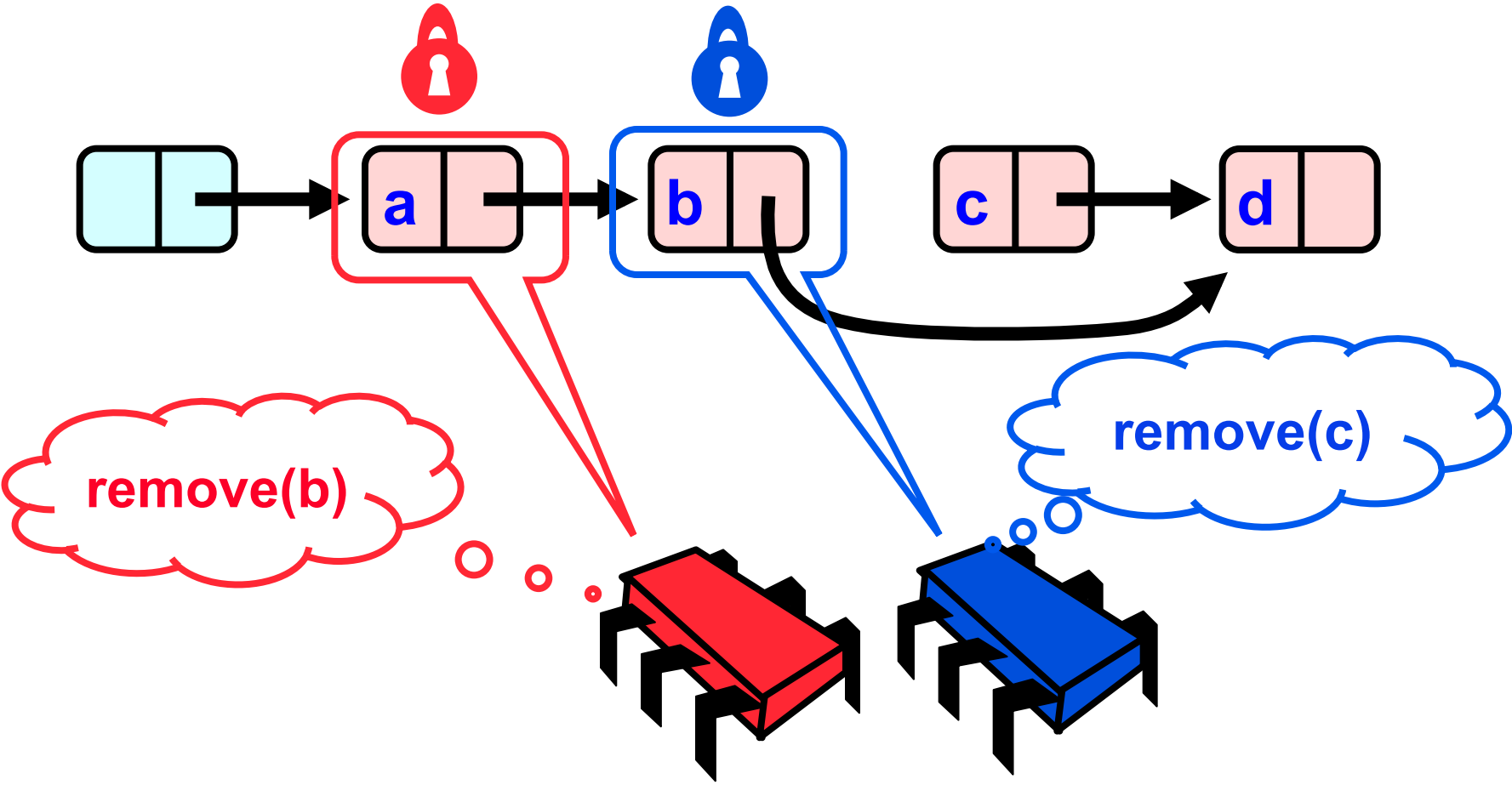
Concurrent Removes



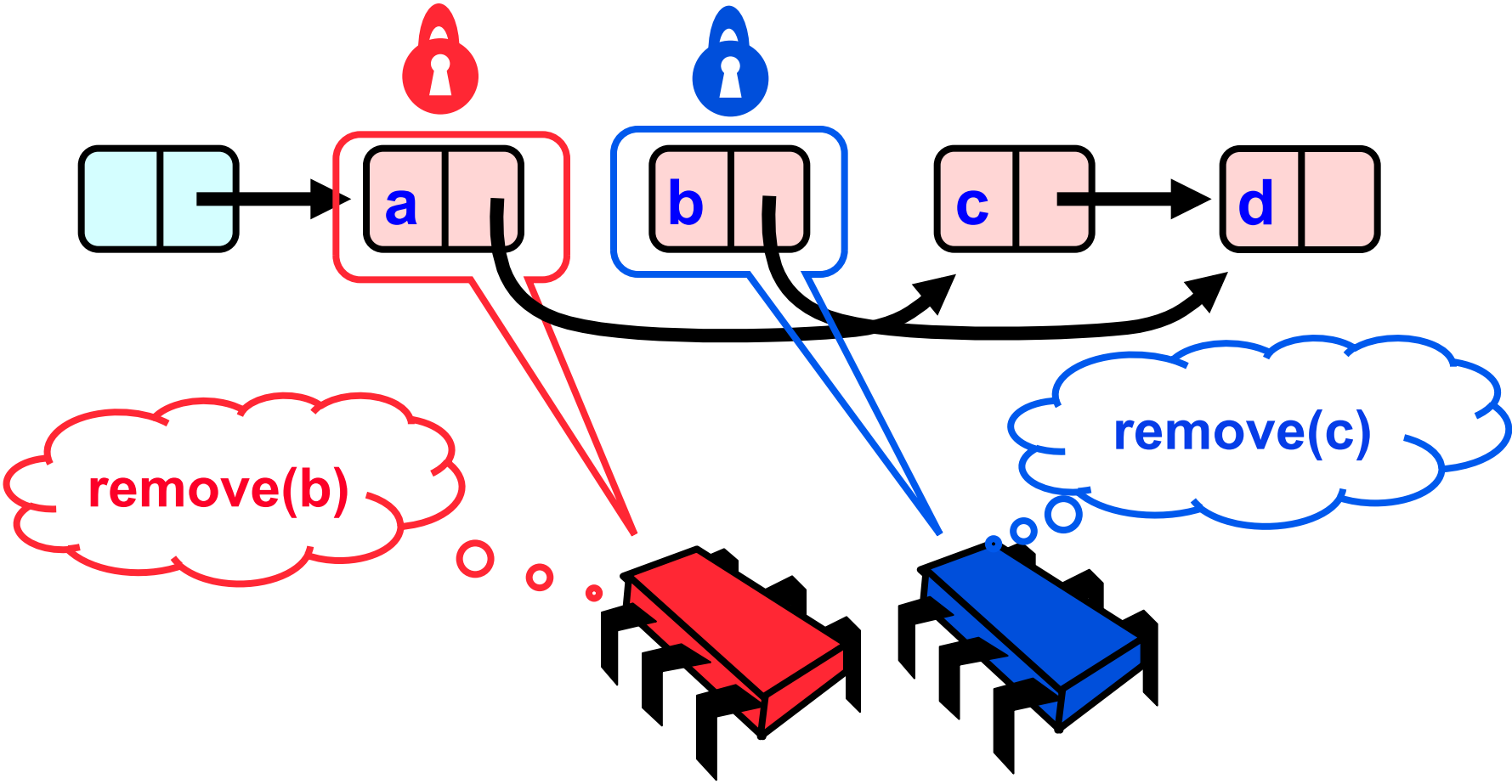
Concurrent Removes



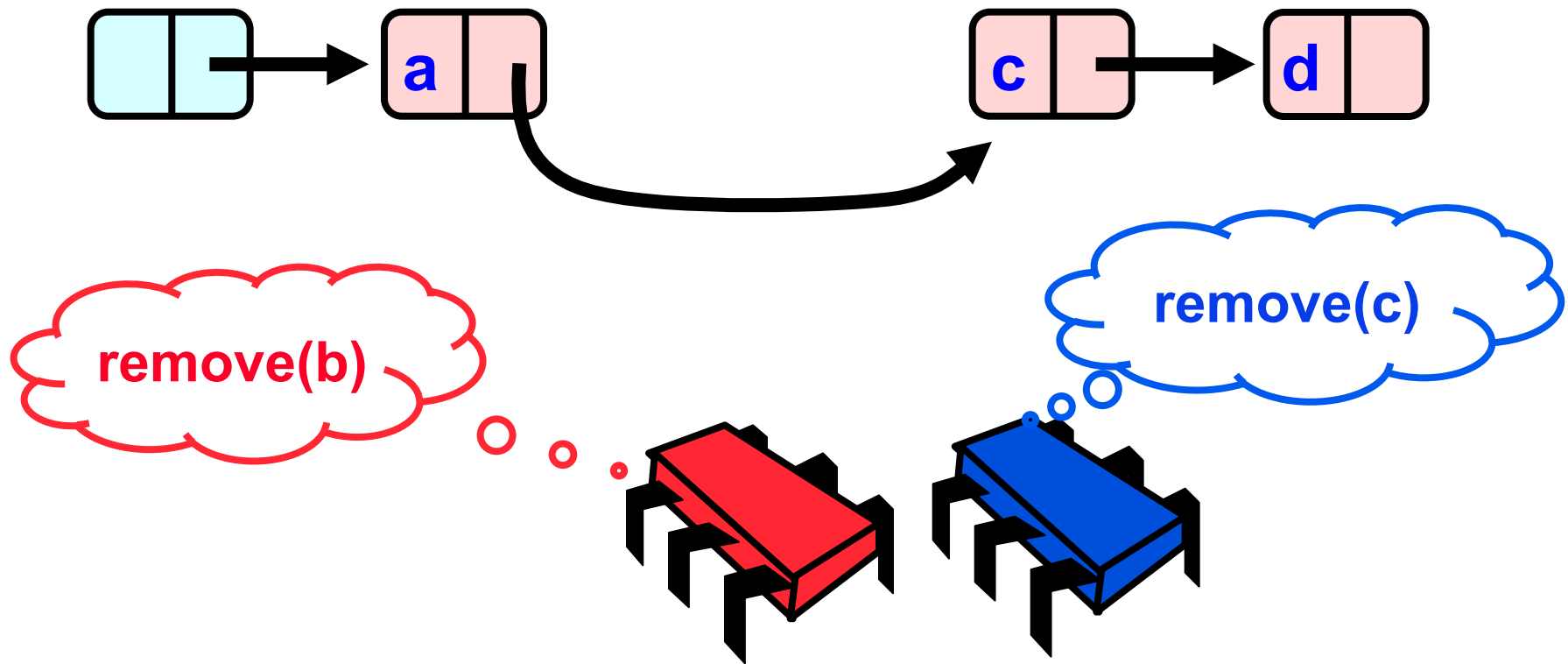
Concurrent Removes



Concurrent Removes

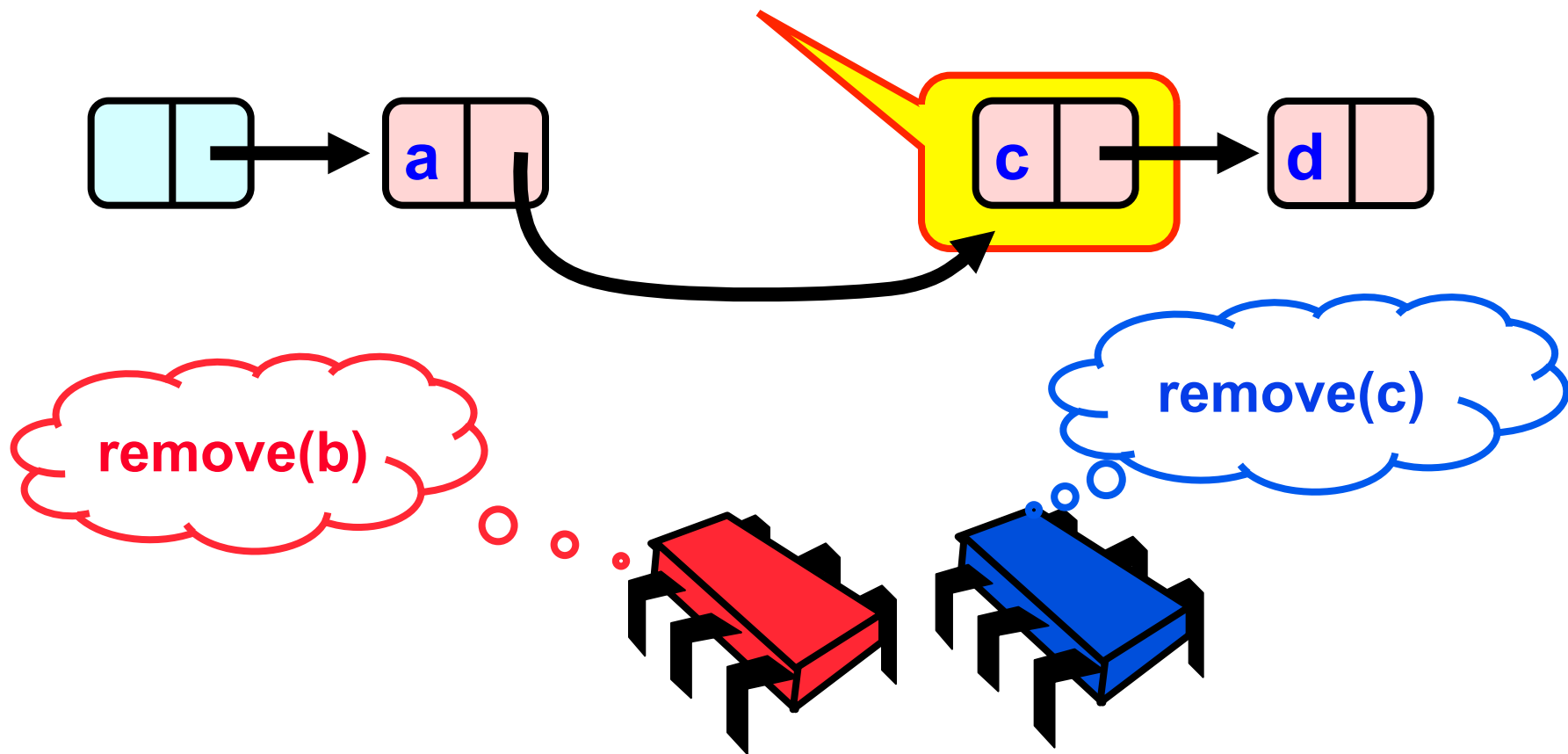


Uh, Oh



Uh, Oh

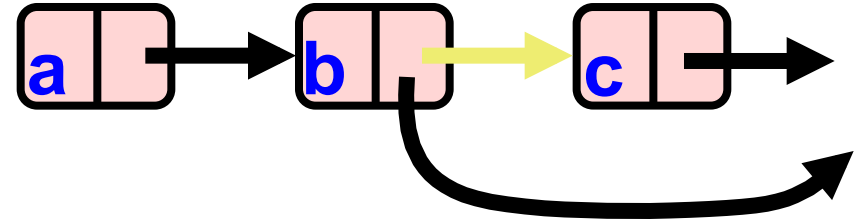
Bad news, c not removed



Problem

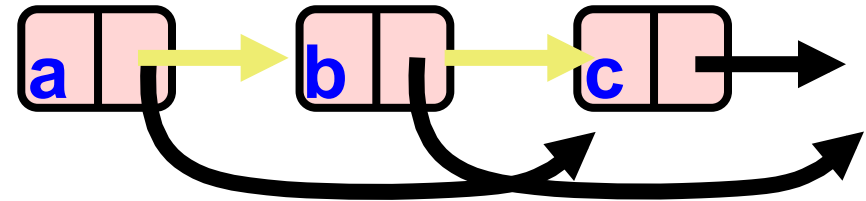
- ▶ To delete node c

- ▷ Swing node b's next field to d



- ▶ Problem is,

- ▷ Someone deleting b concurrently could direct a pointer to C

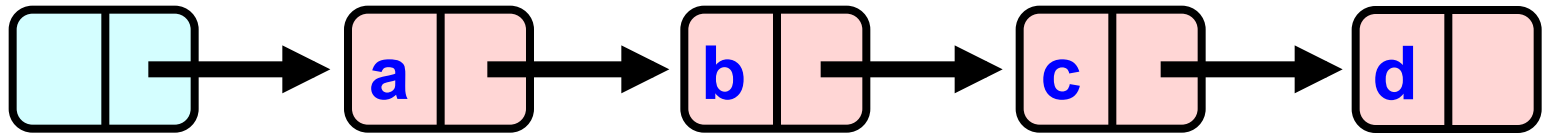


Insight

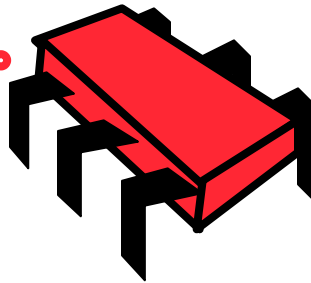
- ▶ If a node is locked
 - ▷ No one can delete node's *successor*

- ▶ If a thread locks
 - ▷ Node to be deleted
 - ▷ And its predecessor
 - ▷ Then it works

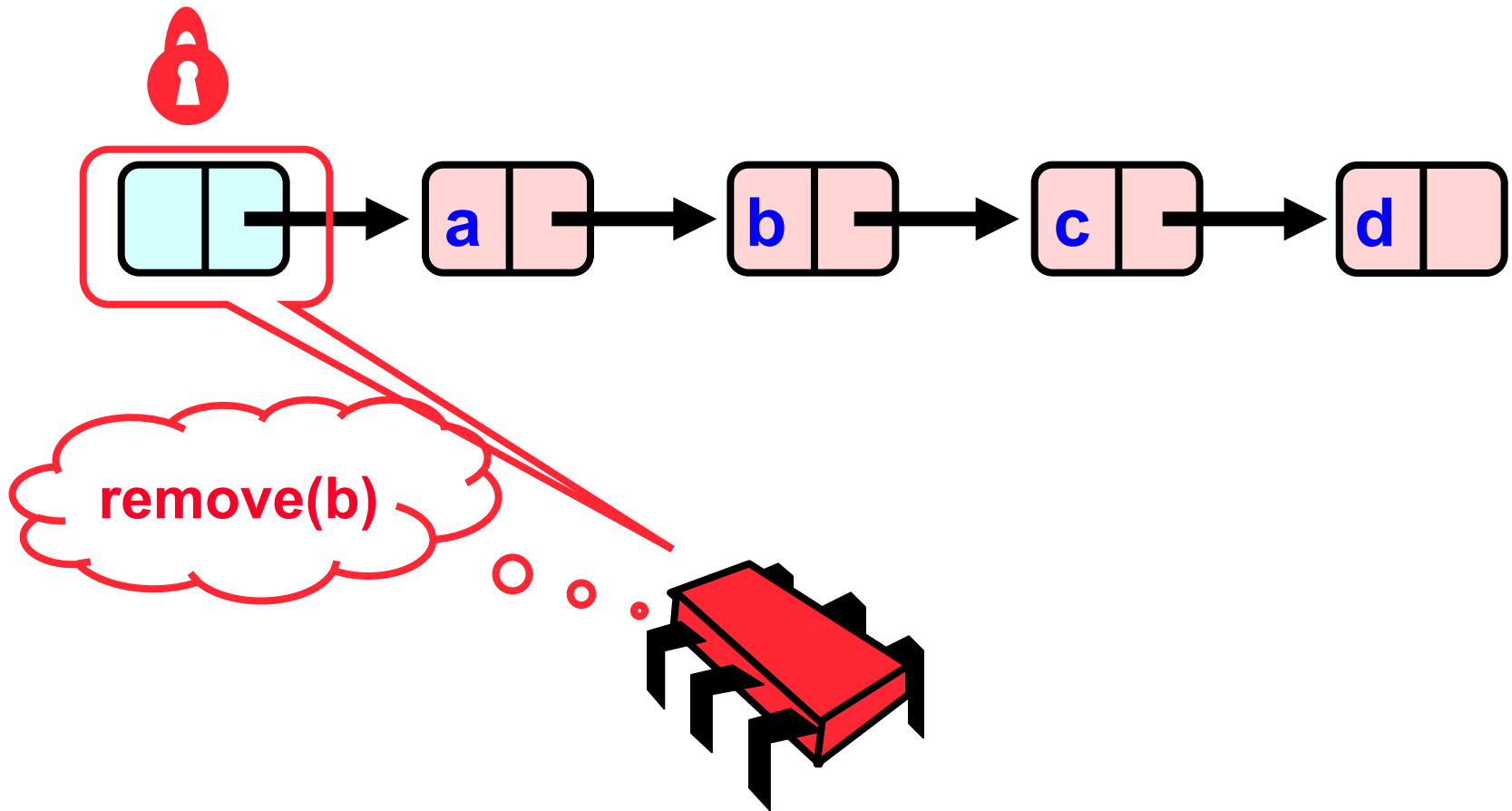
Hand-Over-Hand Again



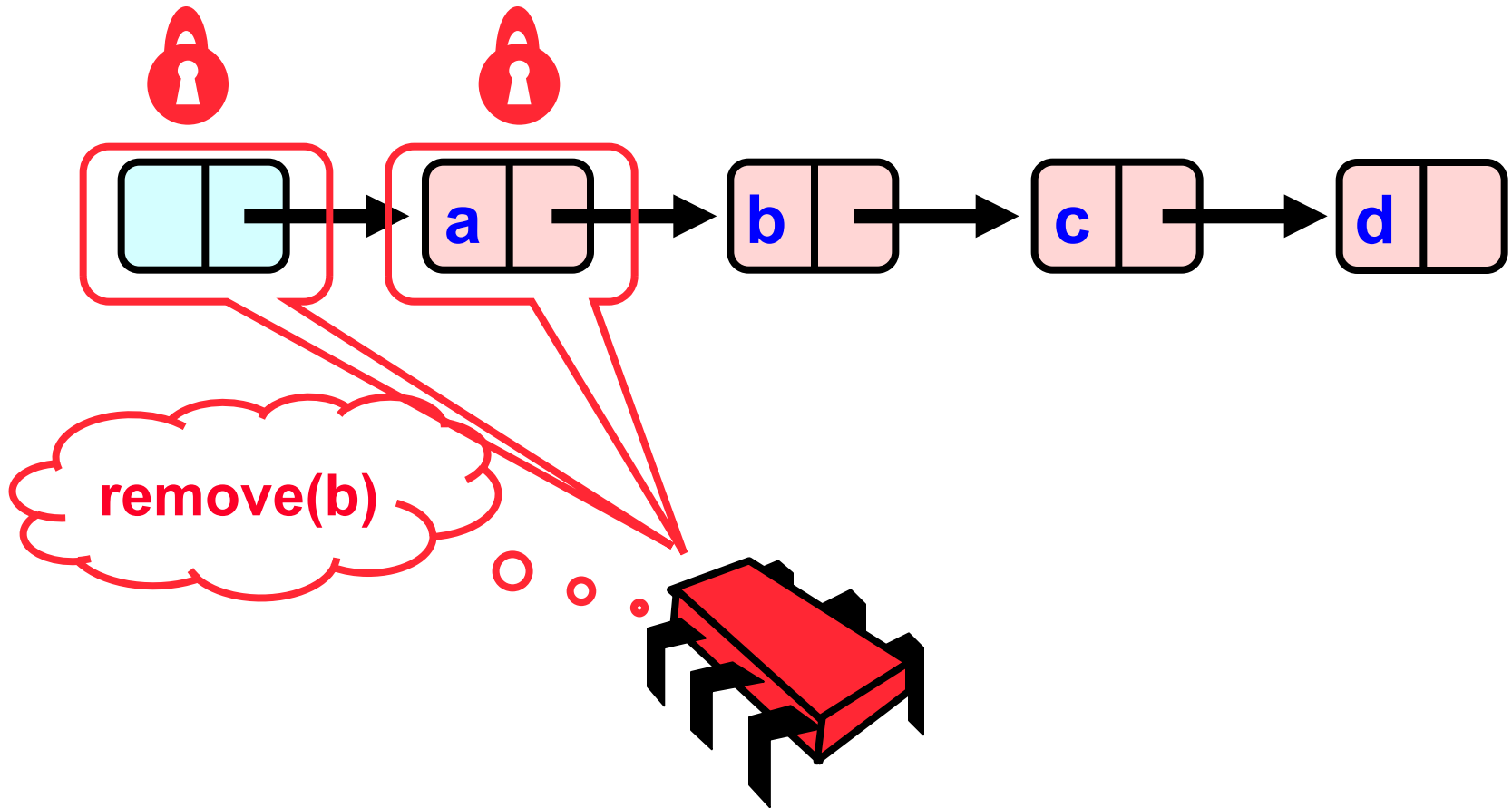
remove(b)



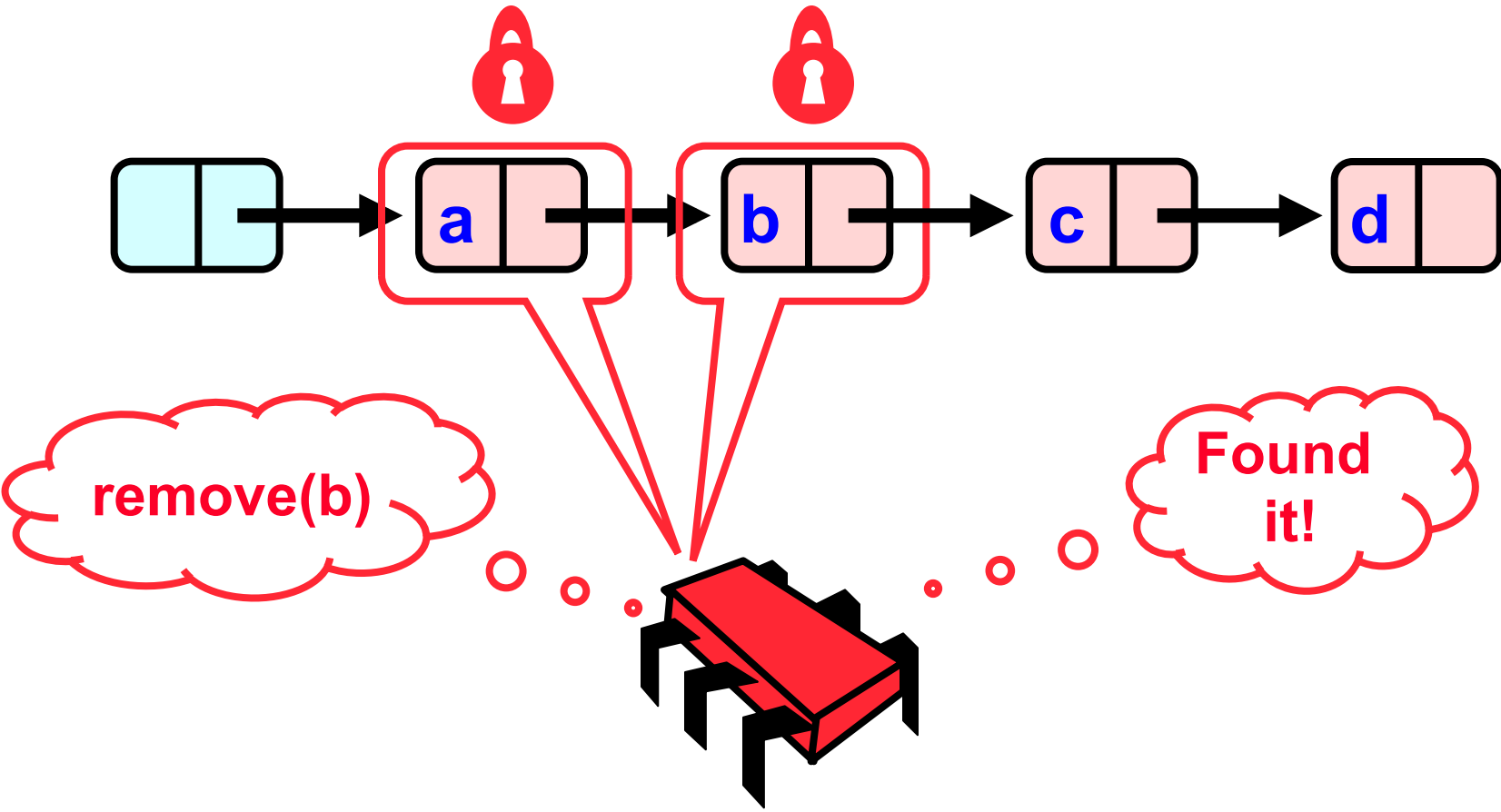
Hand-Over-Hand Again



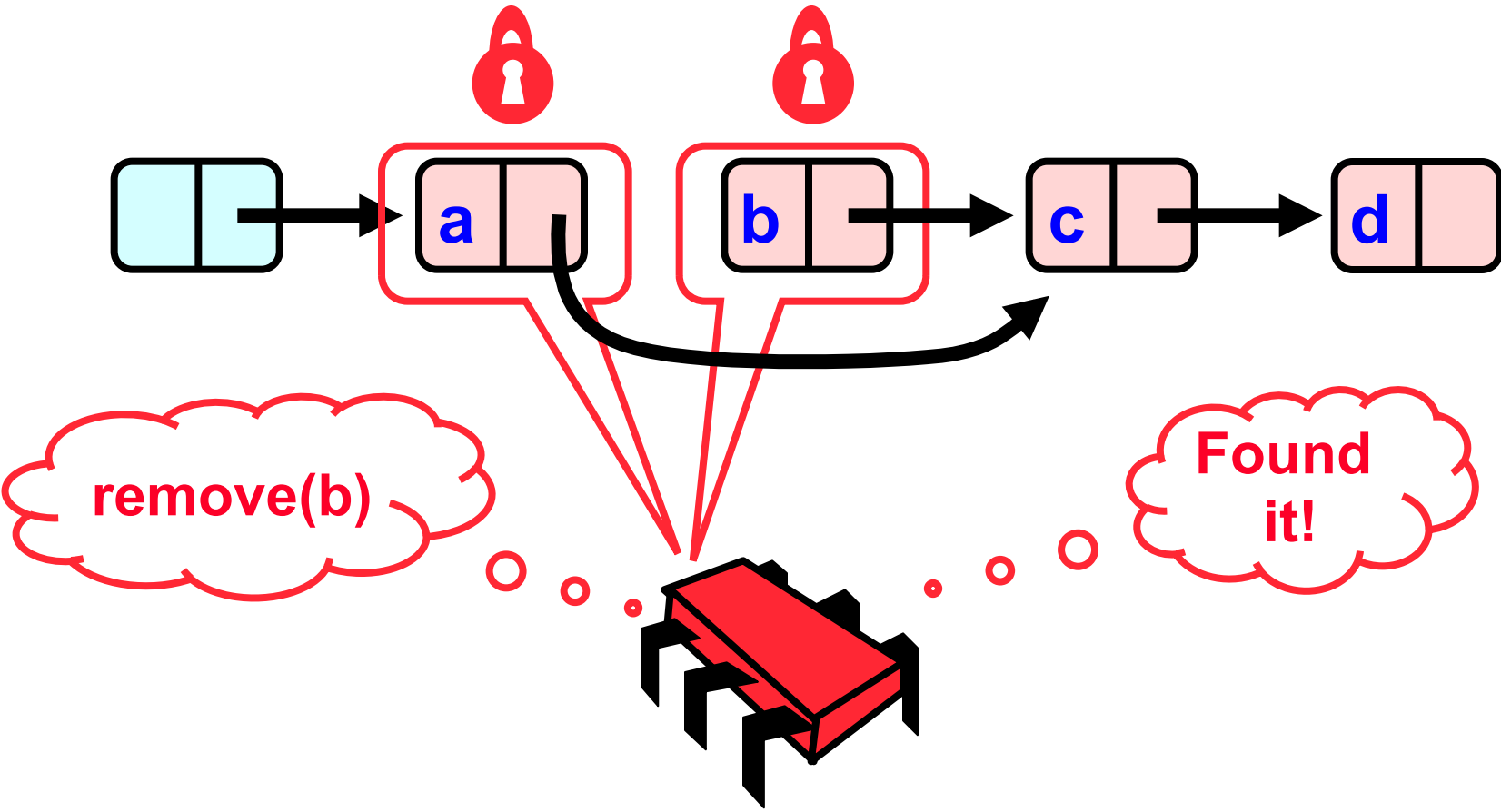
Hand-Over-Hand Again



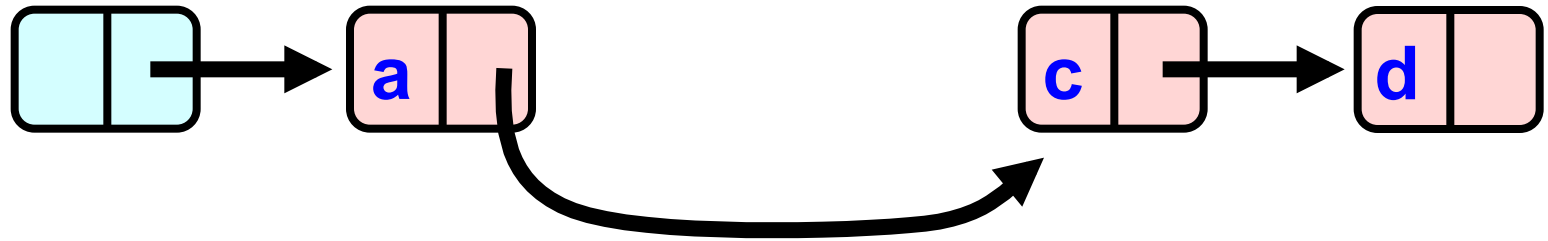
Hand-Over-Hand Again



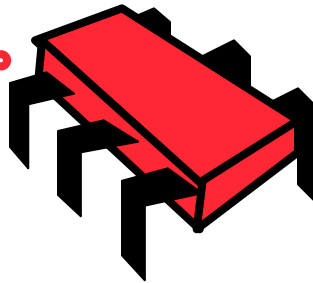
Hand-Over-Hand Again



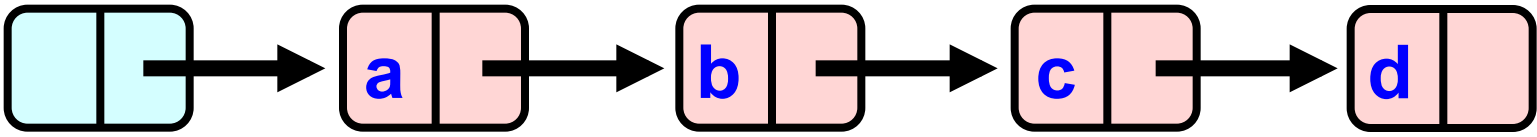
Hand-Over-Hand Again



remove(b)

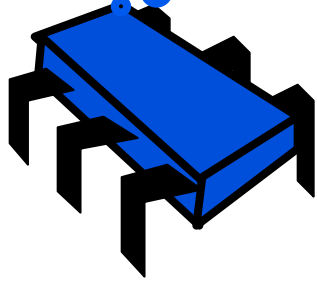
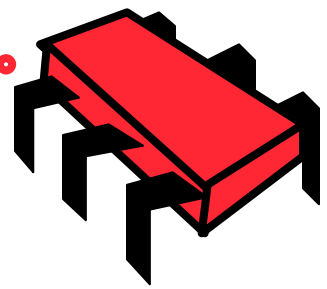


Removing a Node

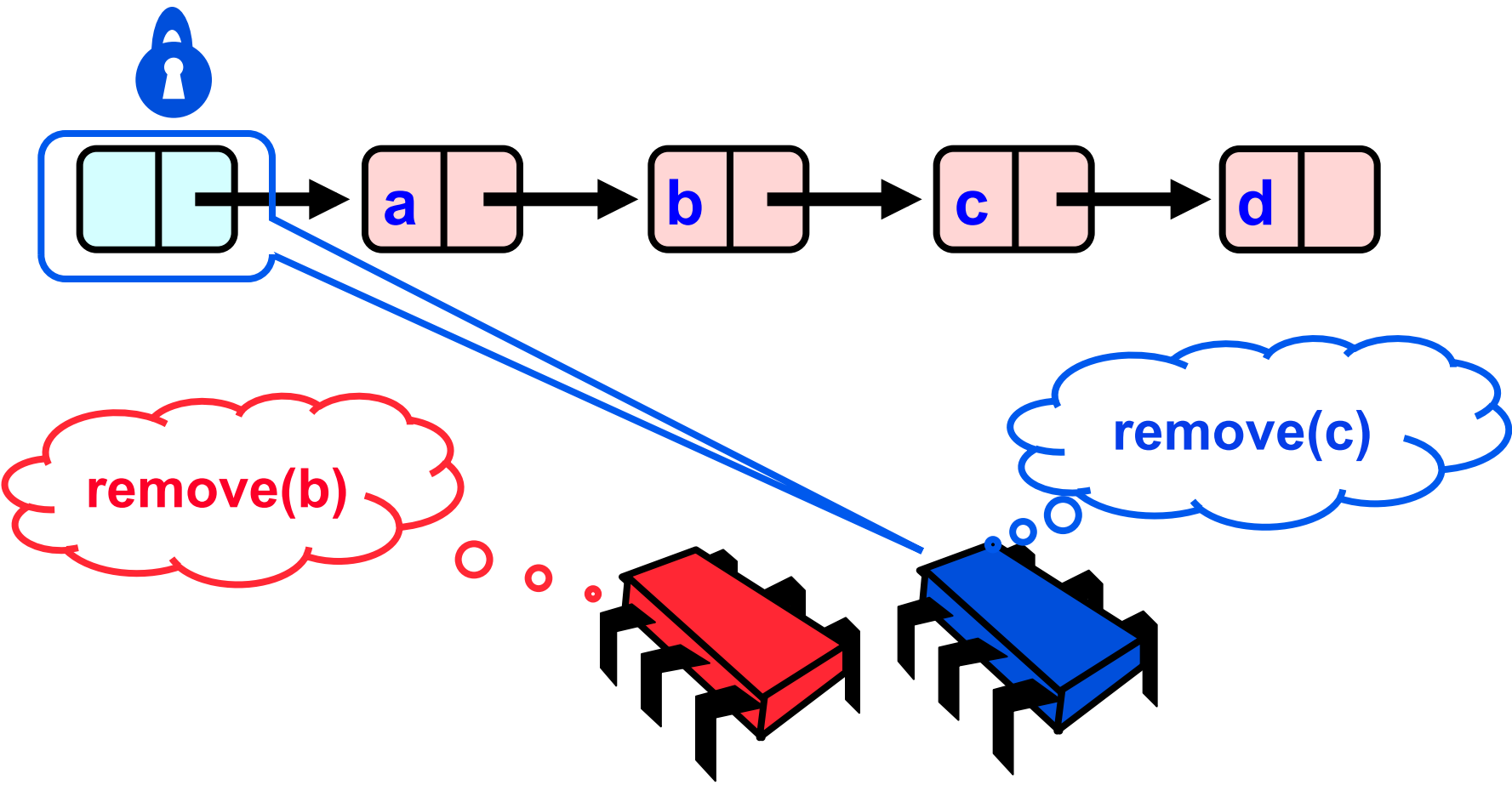


remove(b)

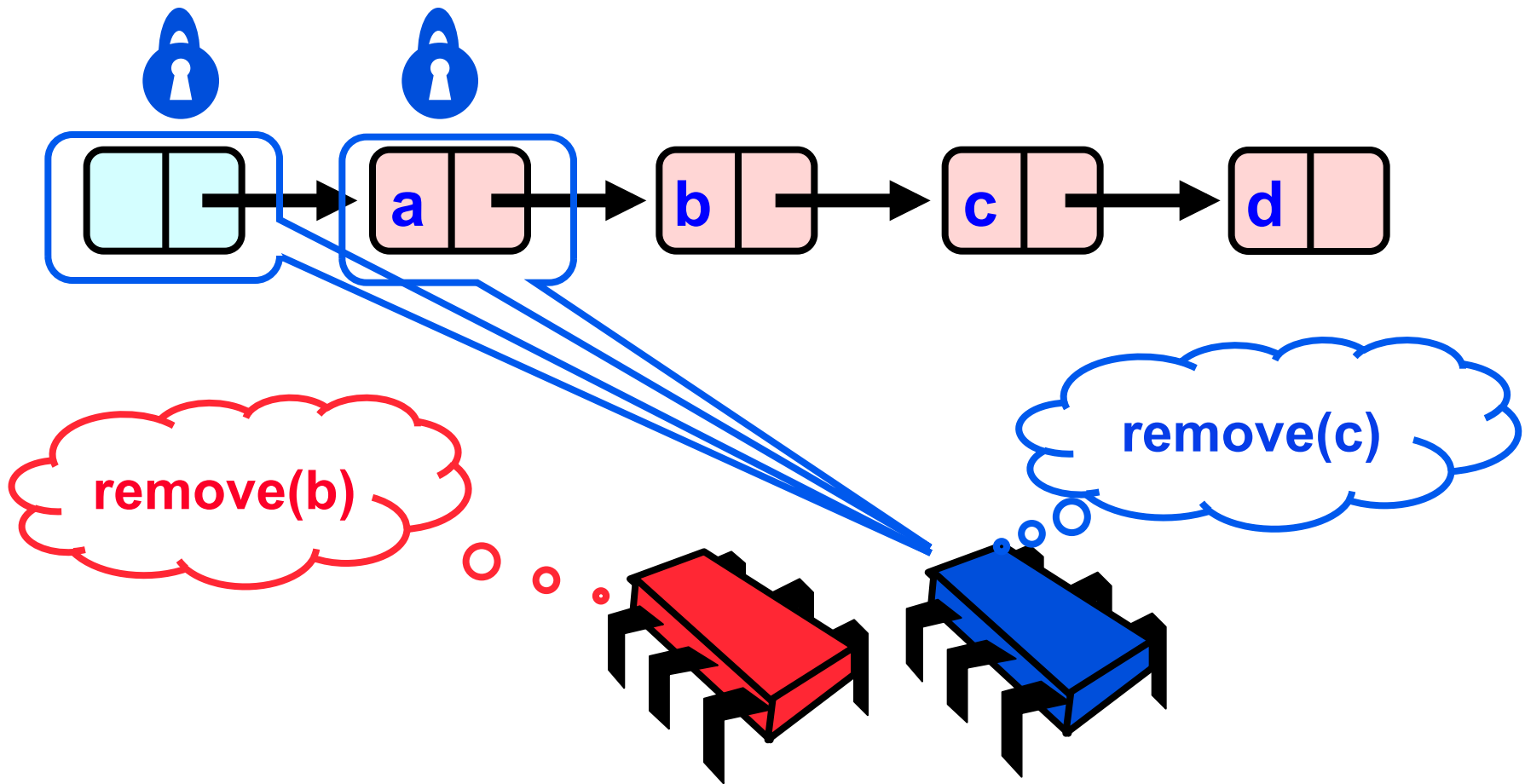
remove(c)



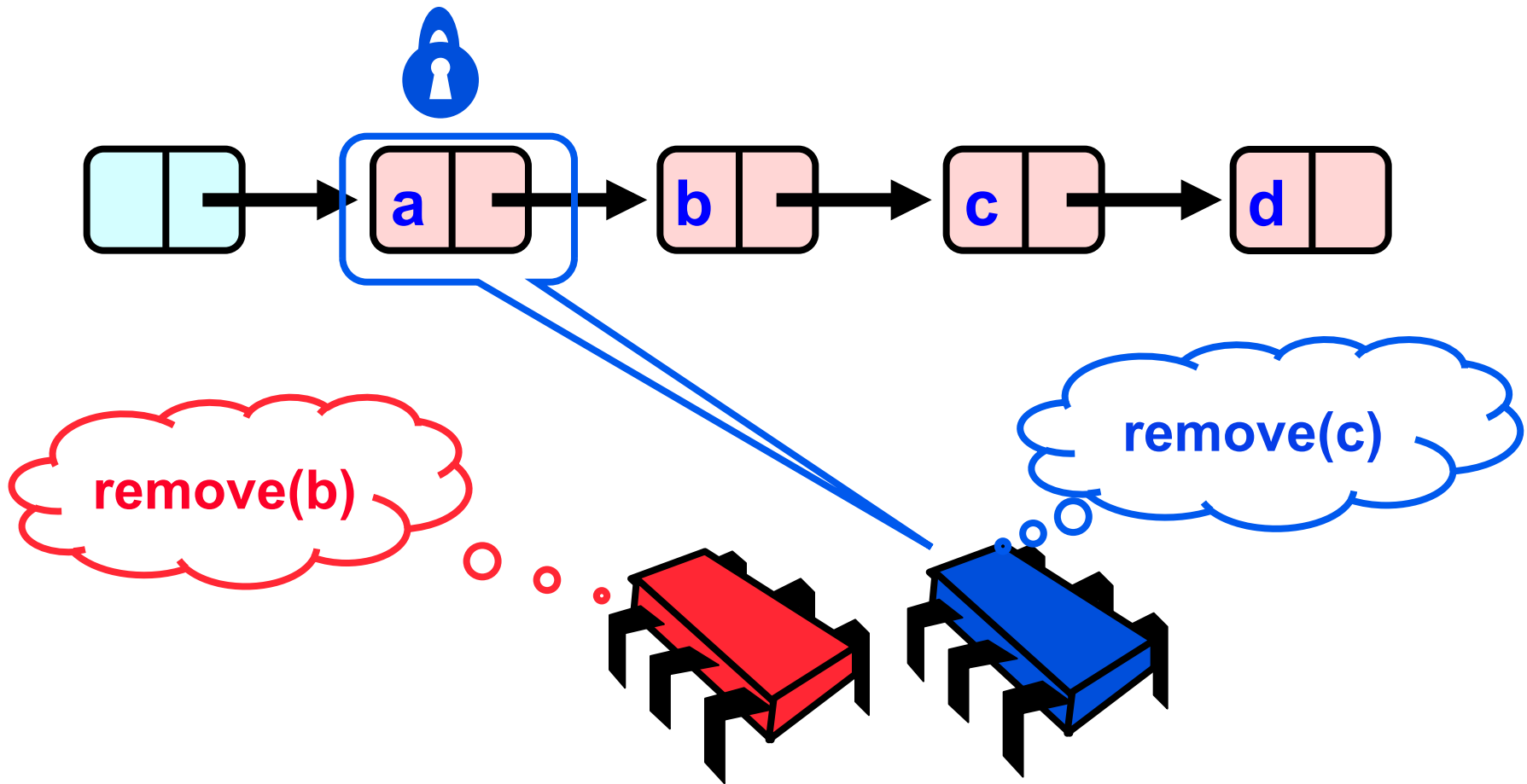
Removing a Node



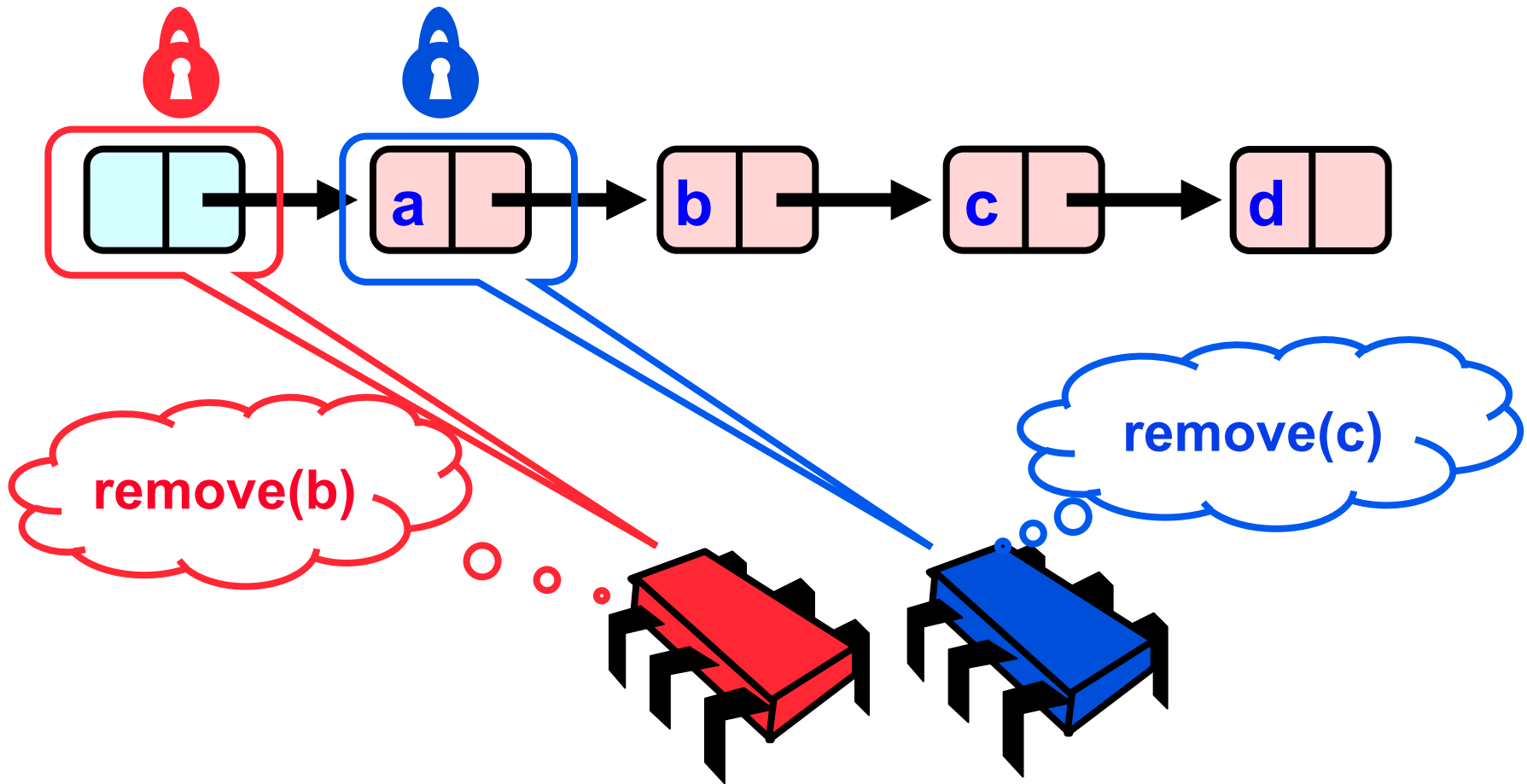
Removing a Node



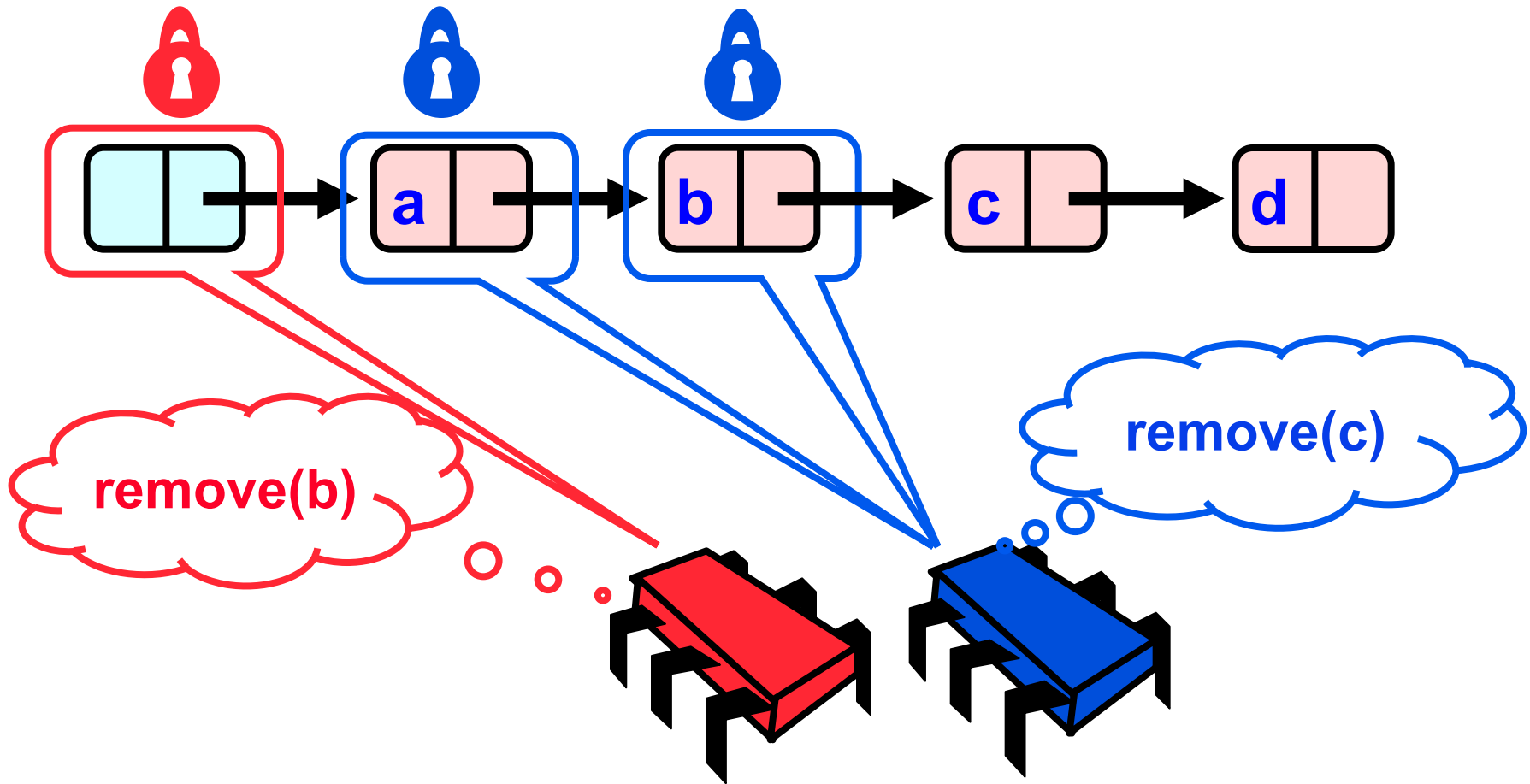
Removing a Node



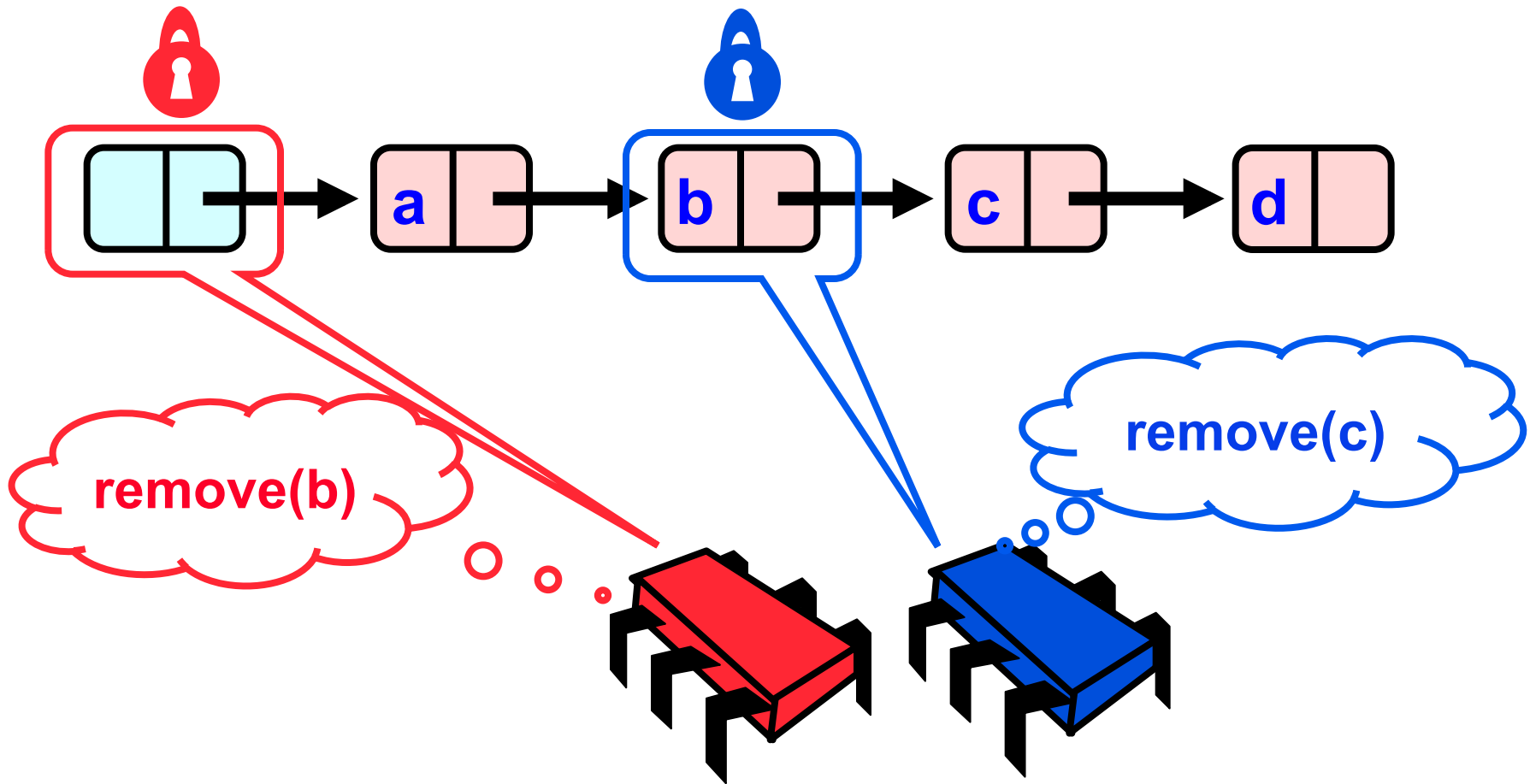
Removing a Node



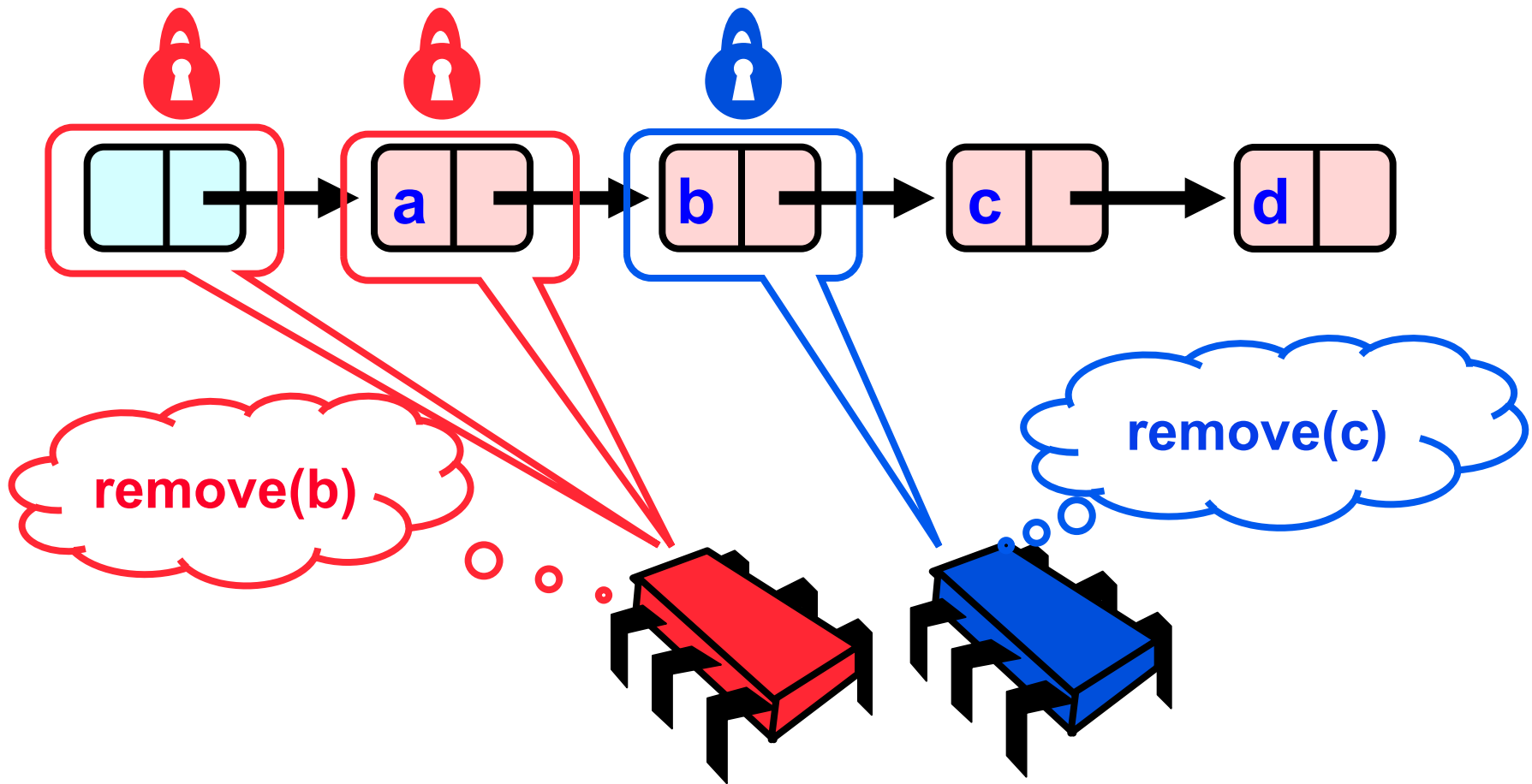
Removing a Node



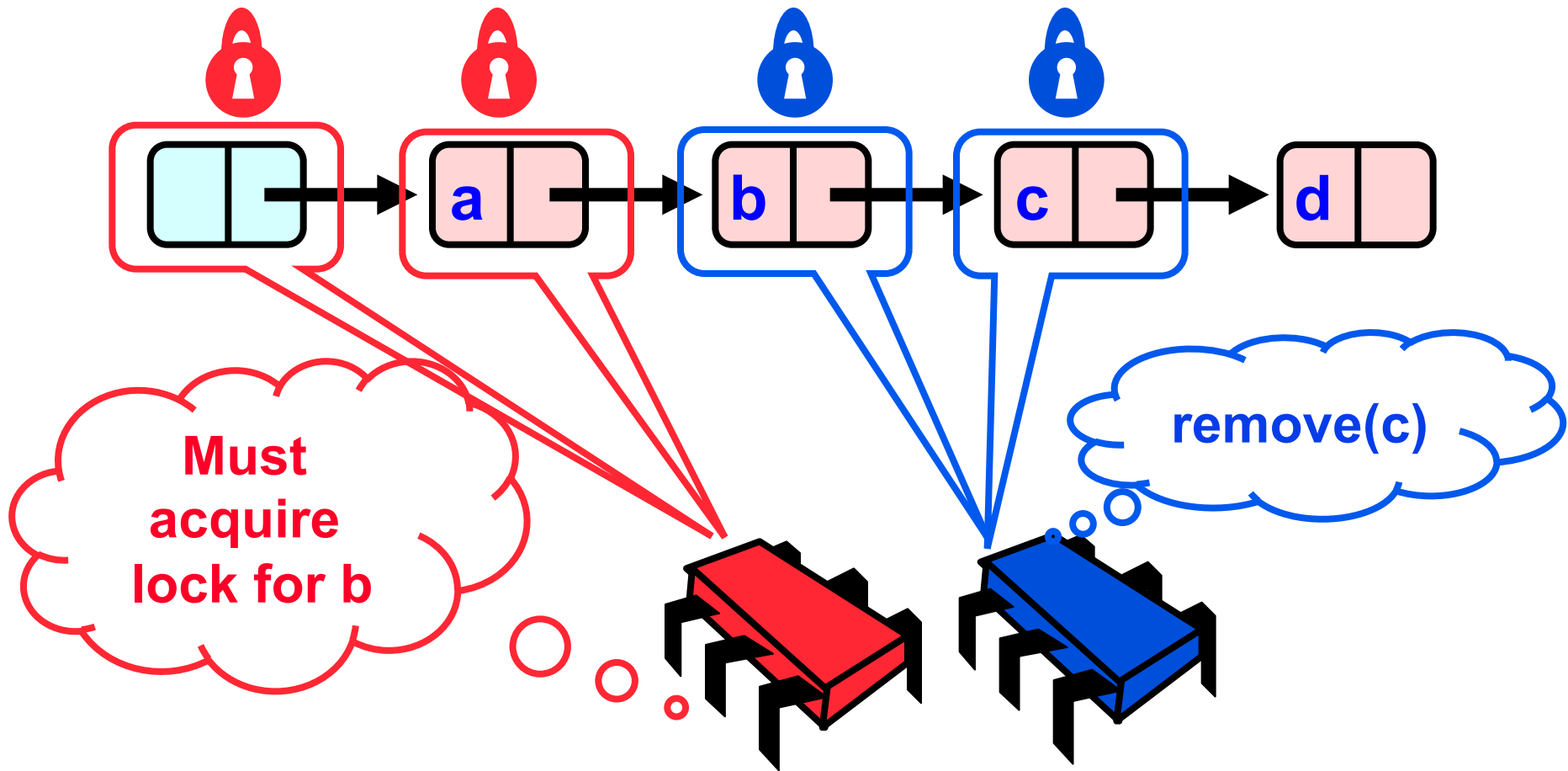
Removing a Node



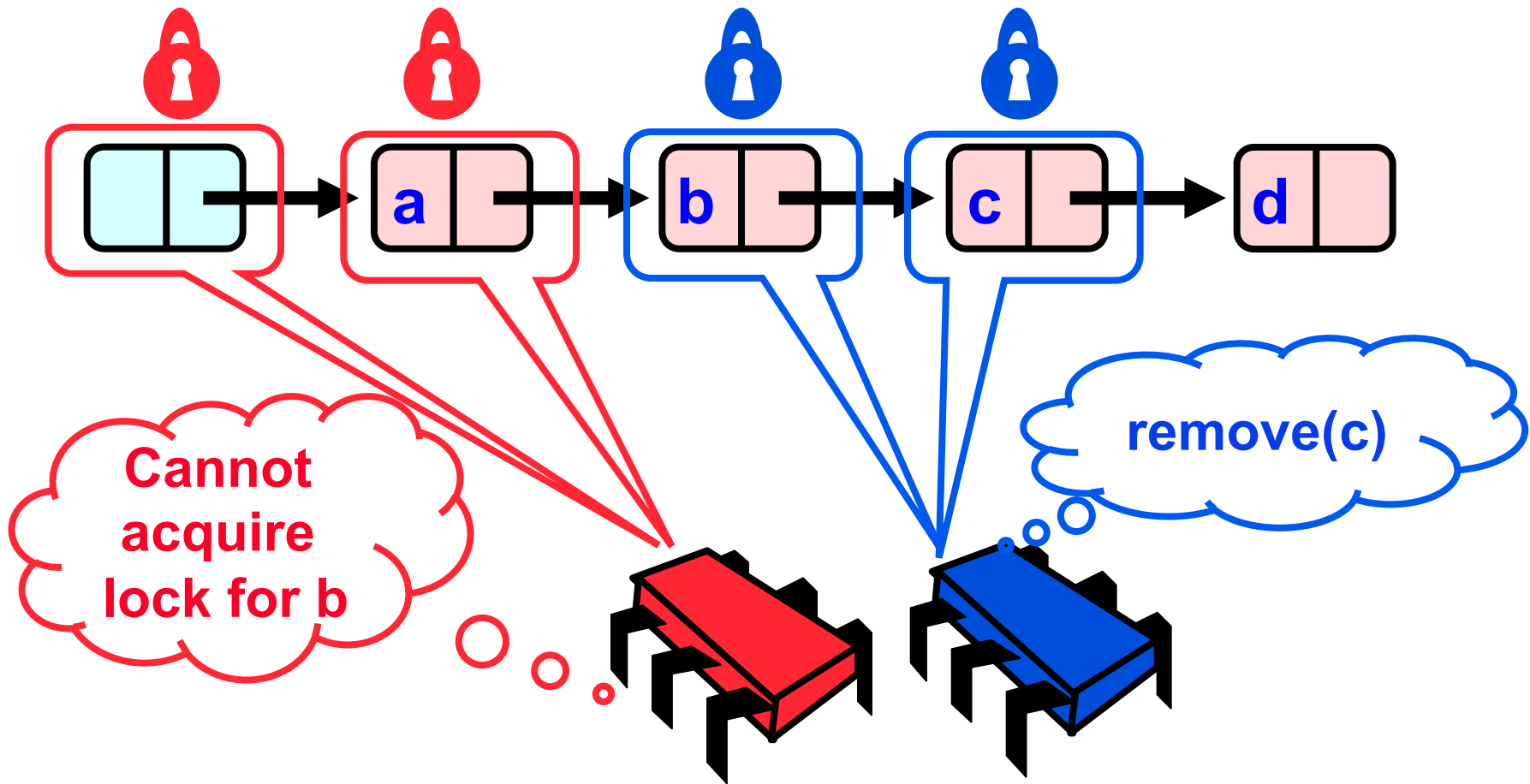
Removing a Node



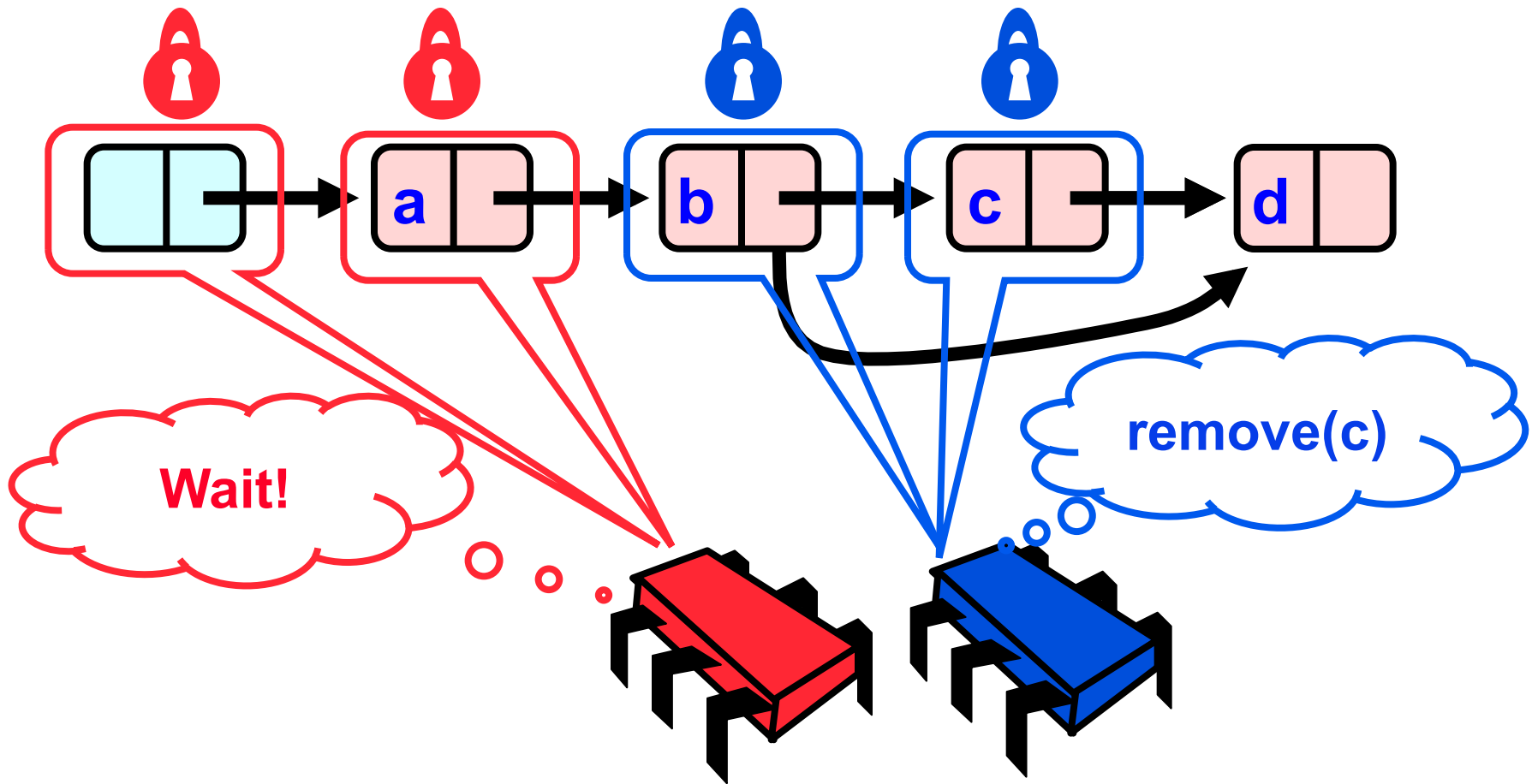
Removing a Node



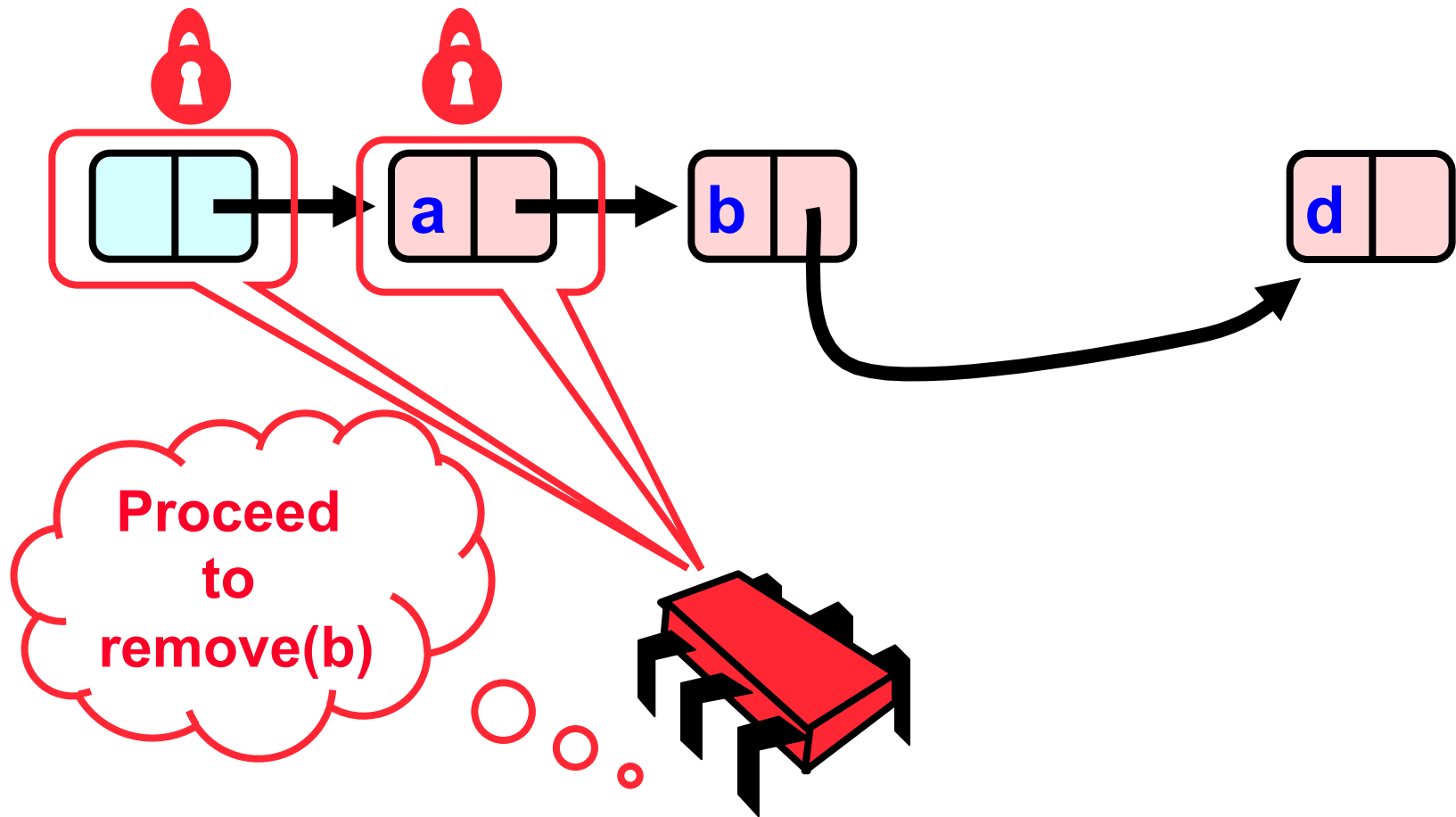
Removing a Node



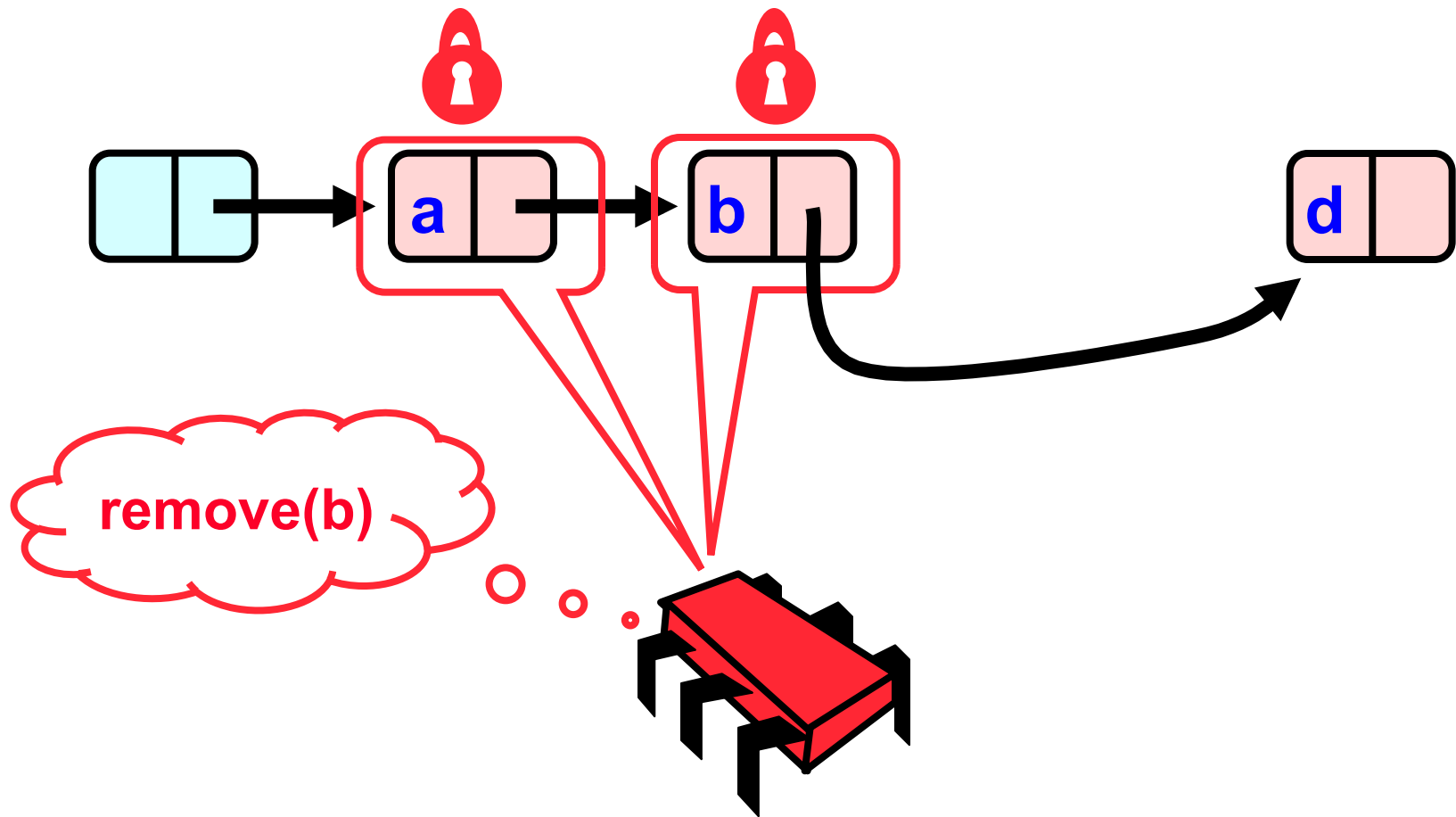
Removing a Node



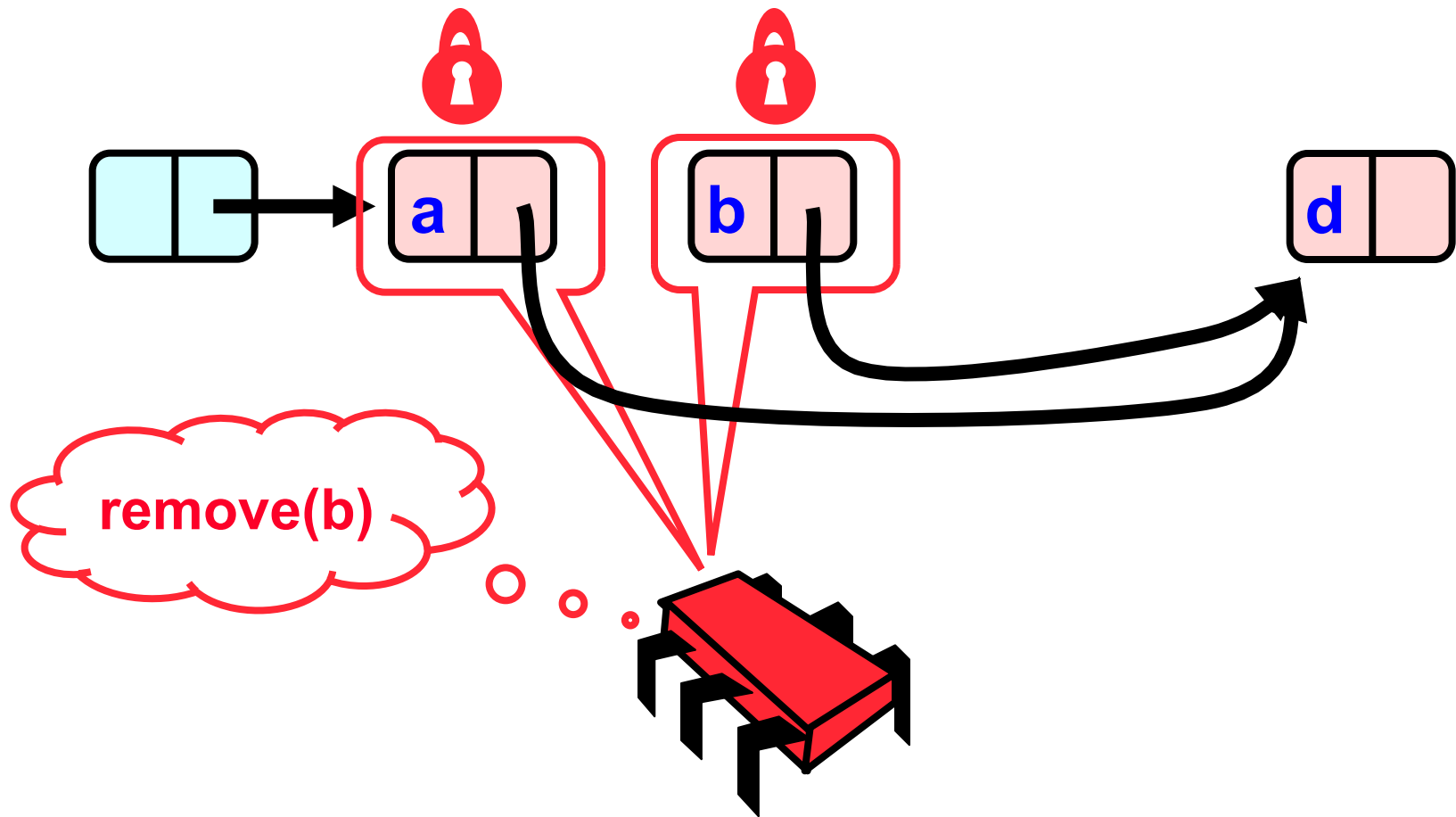
Removing a Node



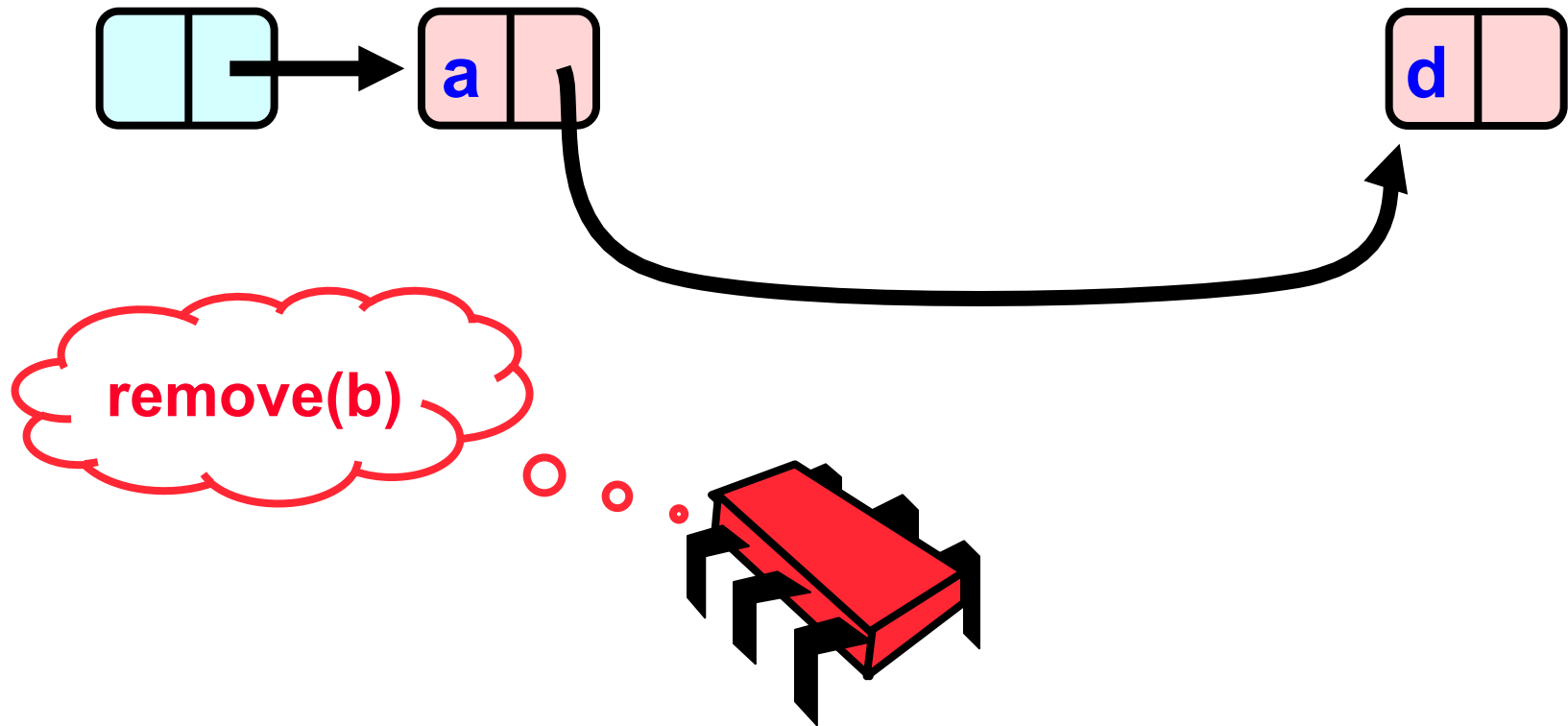
Removing a Node



Removing a Node



Removing a Node



Removing a Node



Remove method

```
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    boolean foundNode = false;
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
    return foundNode;
}
```

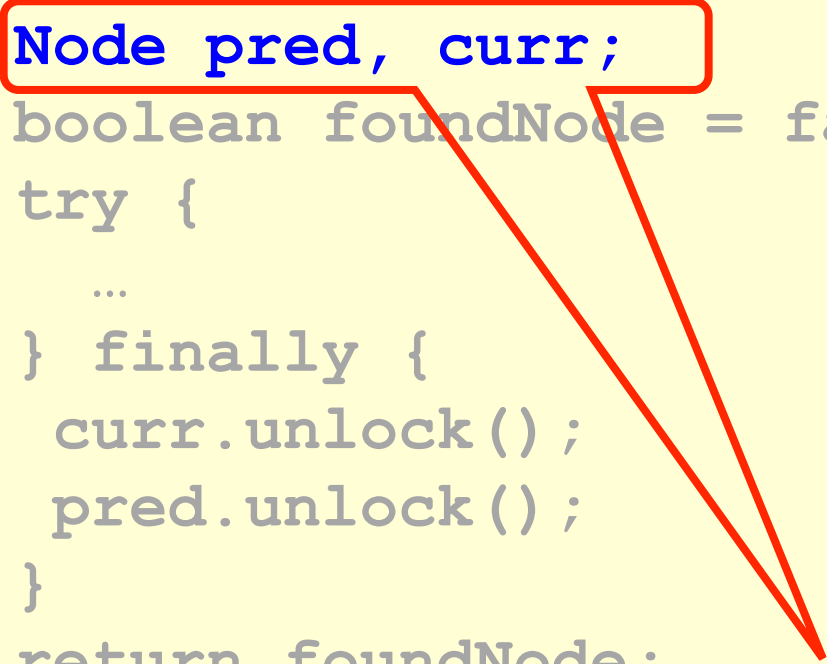
Remove method

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred, curr;  
    boolean foundNode = false;  
    try {  
        ...  
    } finally {  
        curr.unlock();  
        pred.unlock();  
    }  
    return foundNode;  
}
```

Key used to order node

Remove method

```
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    boolean foundNode = false;
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
    return foundNode;
}
```



Predecessor and current nodes

Remove method

```
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    boolean foundNode = false;
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
    return foundNode;
}
```

Node search



Remove method

```
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    boolean foundNode = false;
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
    return foundNode;
}
```

**Make sure
locks released**

Remove method

```
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    boolean foundNode = false;
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
    return foundNode;
}
```

Everything else

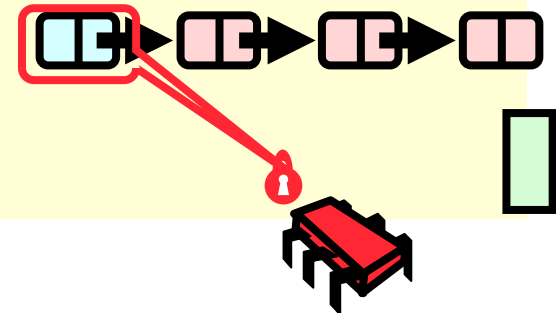
Remove method

```
try {  
    pred = this.head;  
    pred.lock();  
    curr = pred.next;  
    curr.lock();  
  
    ...  
} finally { ... }
```

Remove method

```
try {  
    pred = this.head;  
    pred.lock();  
    curr = pred.next;  
    curr.lock();  
    ...  
} finally { ... }
```

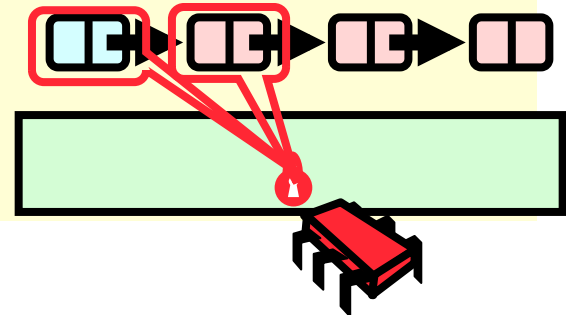
lock pred == head



Remove method

```
try {  
  pred = this.head;  
  pred.lock();  
  curr = pred.next;  
  curr.lock();  
  ...  
} finally { ... }
```

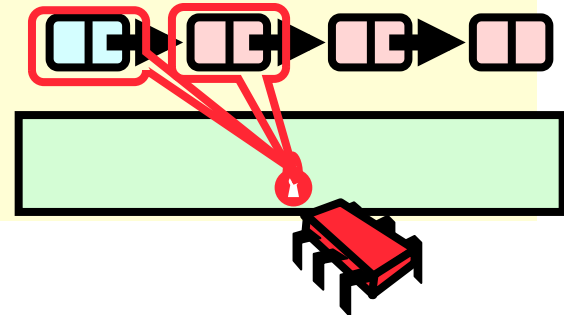
Lock current



Remove method

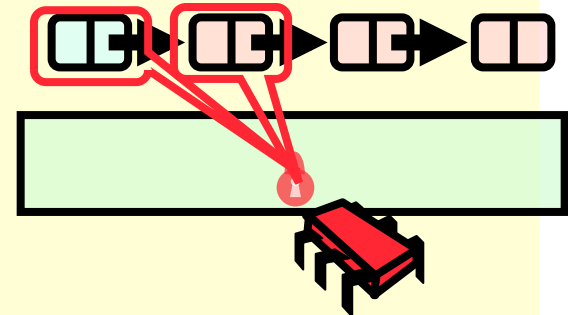
```
try {  
    pred = this.head;  
    pred.lock();  
    curr = pred.next;  
    curr.lock();  
    ...  
} finally { ... }
```

Traversing list



Remove: searching

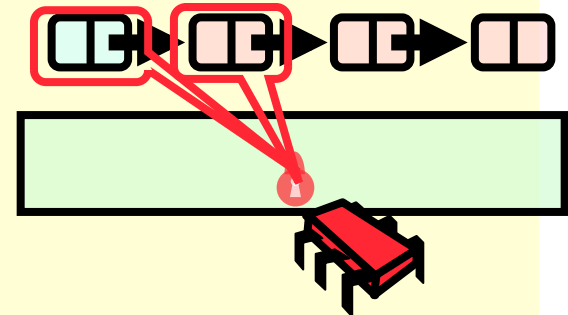
```
while (curr.key <= key) {  
  if (item == curr.item) {  
    pred.next = curr.next;  
    foundNode = true;  
    break;  
  }  
  pred.unlock();  
  pred = curr;  
  curr = curr.next;  
  curr.lock();  
}
```



Remove: searching

```
while (curr.key <= key) {  
  if (item == curr.item) {  
    pred.next = curr.next;  
    foundNode = true;  
    break;  
  }  
  pred.unlock();  
  pred = curr;  
  curr = curr.next;  
  curr.lock();  
}
```

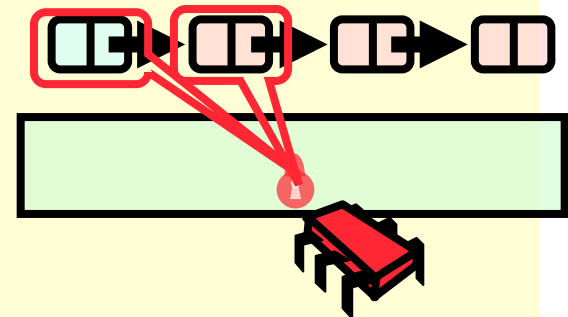
Search key range



Remove: searching

```
while (curr.key <= key) {  
  if (item == curr.item) {  
    pred.next = curr.next;  
    foundNode = true;  
    break;  
  }  
  pred.unlock();  
  pred = curr;  
  curr = curr.next;  
  curr.lock();  
}
```

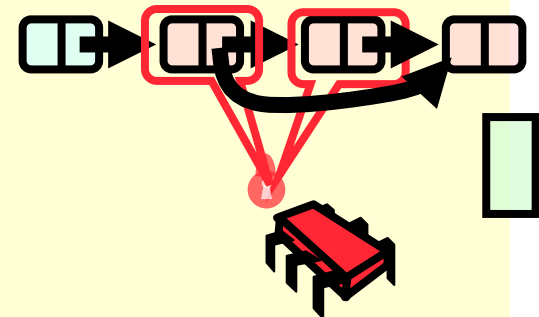
**At start of each loop:
curr and pred locked**



Remove: searching

```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        foundNode = true;  
        break;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}
```

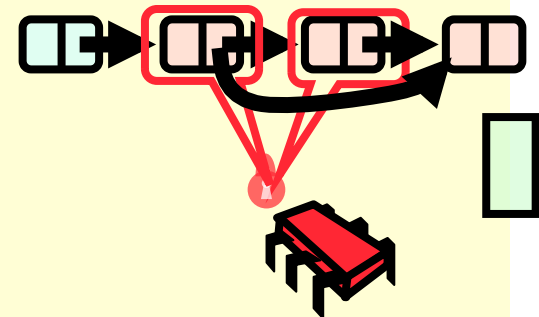
If item found, remove node



Remove: searching

```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        foundNode = true;  
        break;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}
```

If node found, remove it

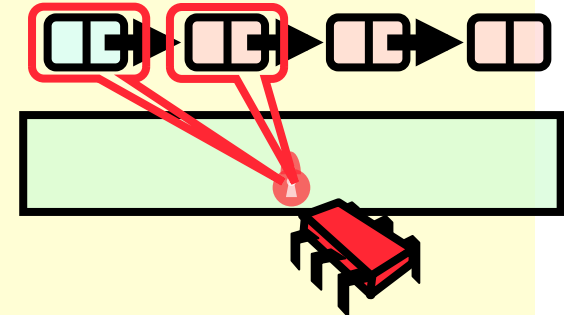


Remove: searching

```
while (curr.key <= item) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        foundNode = true;  
        break;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}
```

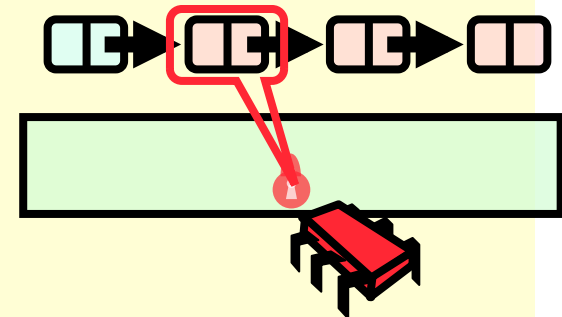
Unlock predecessor

pred.unlock();



Remove: searching

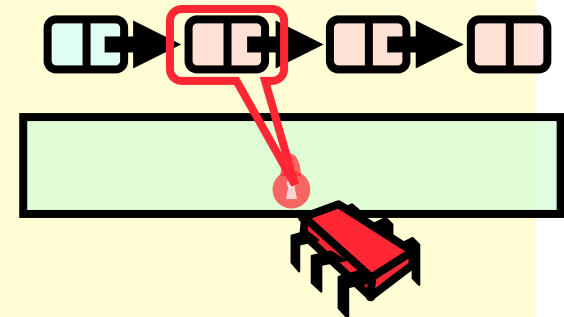
```
while (curr != null) {  
    Only one node locked!  
    if (item == curr.item) {  
        pred.next = curr.next;  
        foundNode = true;  
        break;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}
```



Remove: searching

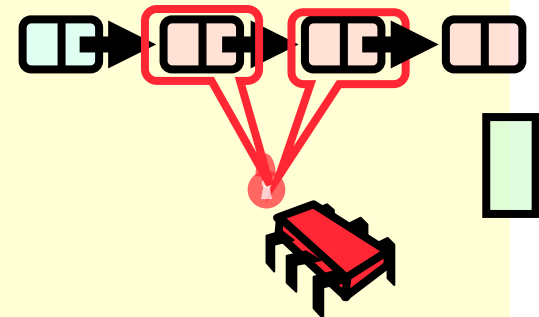
```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        foundNode = true;  
        break;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}
```

demote current



Remove: searching

```
while (curr.key <= key) {  
  if (item == curr.item) {  
    Find and lock new current  
    pred.next = curr.next;  
    foundNode = true;  
    break;  
  }  
  pred.unlock();  
  pred = curr;  
  curr = curr.next;  
  curr.lock();  
}
```

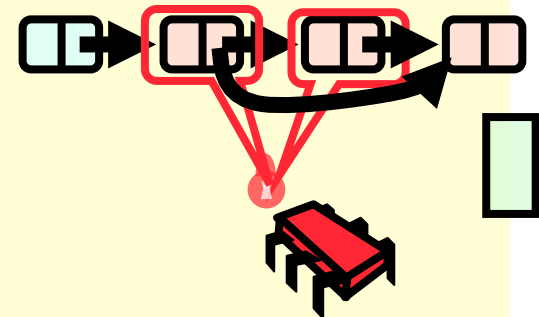


Remove: searching

```
while (curr.key <= key) {  
    if (curr.item == key) {  
        pred.next = curr.next;  
        foundNode = true;  
        break;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}
```

Lock invariant restored

`curr = curr.next;`
`curr.lock();`



Why does this work?

- ▶ To remove node e
 - ▷ Must lock e
 - ▷ Must lock e 's predecessor

- ▶ Therefore, if you lock a node
 - ▷ It can't be removed
 - ▷ And neither can its successor

Adding Nodes

- ▶ To add node e
 - ▷ Must lock predecessor
 - ▷ Must lock successor

- ▶ Add/remove must acquire locks in the same order
 - ▷ What happens in the order is compromised?
 - ▷ E.g., remove code lock predecessor first, add successor first

Properties to prove for add/remove

- ▶ Does safety hold?
- ▶ Does the liveness property hold?
- ▶ Are the invariants maintained?

Drawbacks

- ▶ Better than coarse-grained lock
 - ▷ Threads can traverse in parallel

- ▶ Still not ideal
 - ▷ Long chain of acquire/release
 - ▷ Inefficient

Traffic Jam

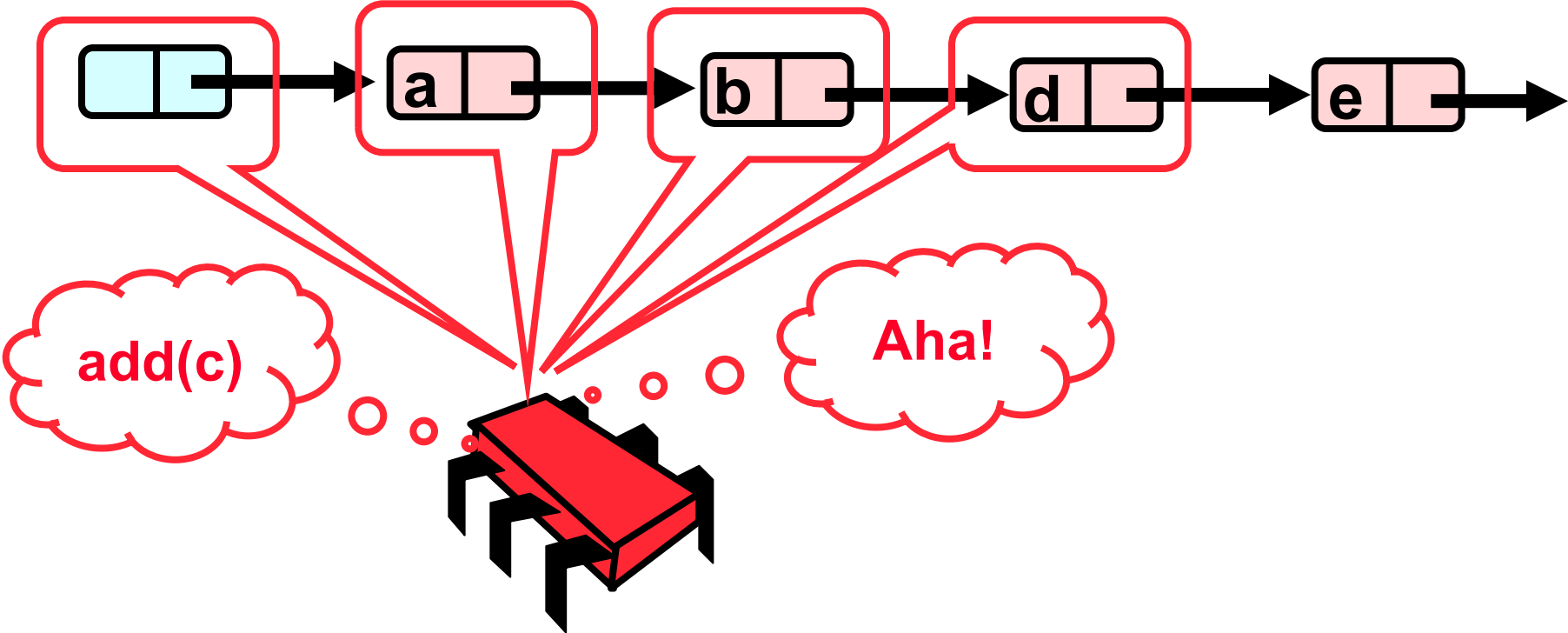
- ▶ Any concurrent data structure based on mutual exclusion has a weakness

- ▶ If one thread
 - ▷ Enters critical section
 - ▷ And “eats the big muffin”
 - ▷ Cache miss, page fault, descheduled ...
 - ▷ Everyone else using that lock is stuck!
 - ▷ Need to trust the scheduler....

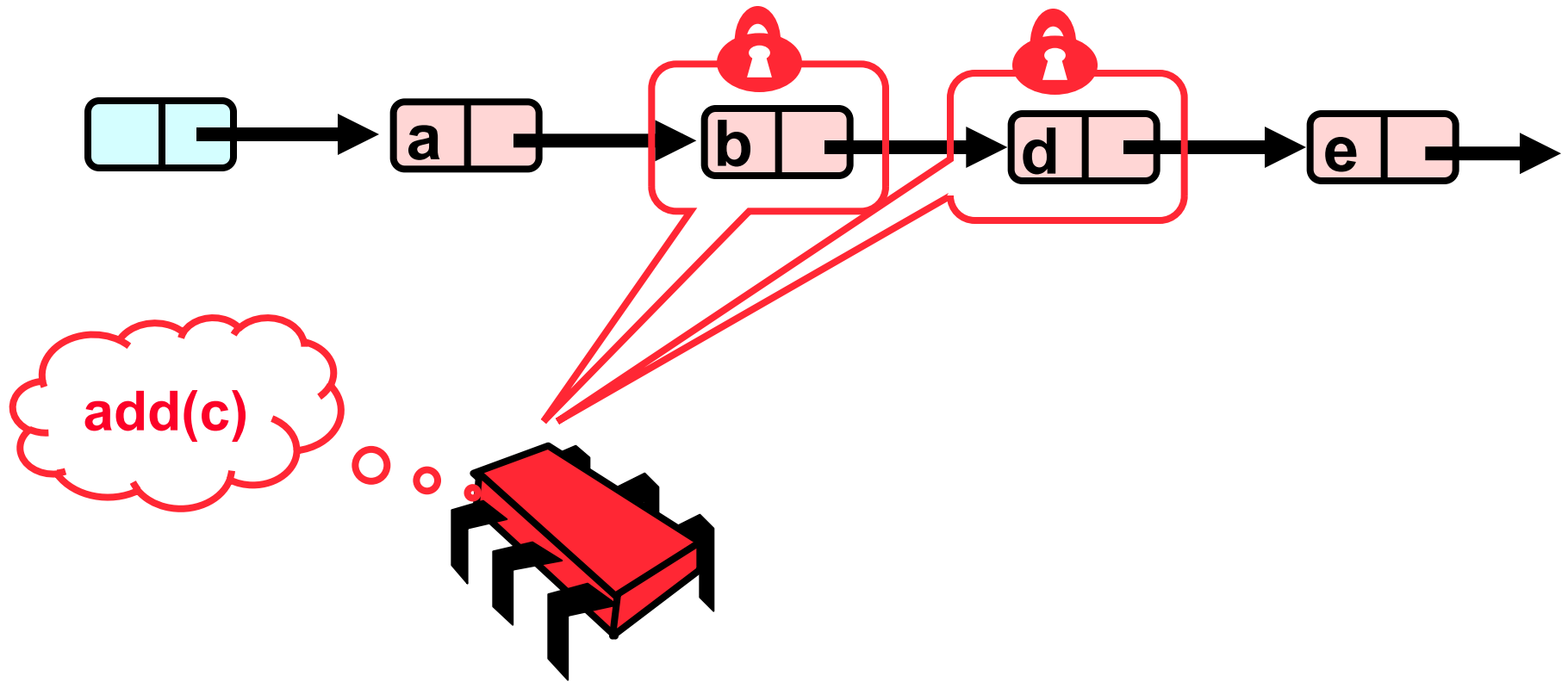
Other patterns: Optimistic Synchronization

- ▶ Search without locking...
- ▶ If you find it, lock and check...
 - ▷ Ok → we are done
 - ▷ Opps → start over
- ▶ Evaluation
 - ▷ Usually cheaper than locking
 - ▷ Mistakes are expensive

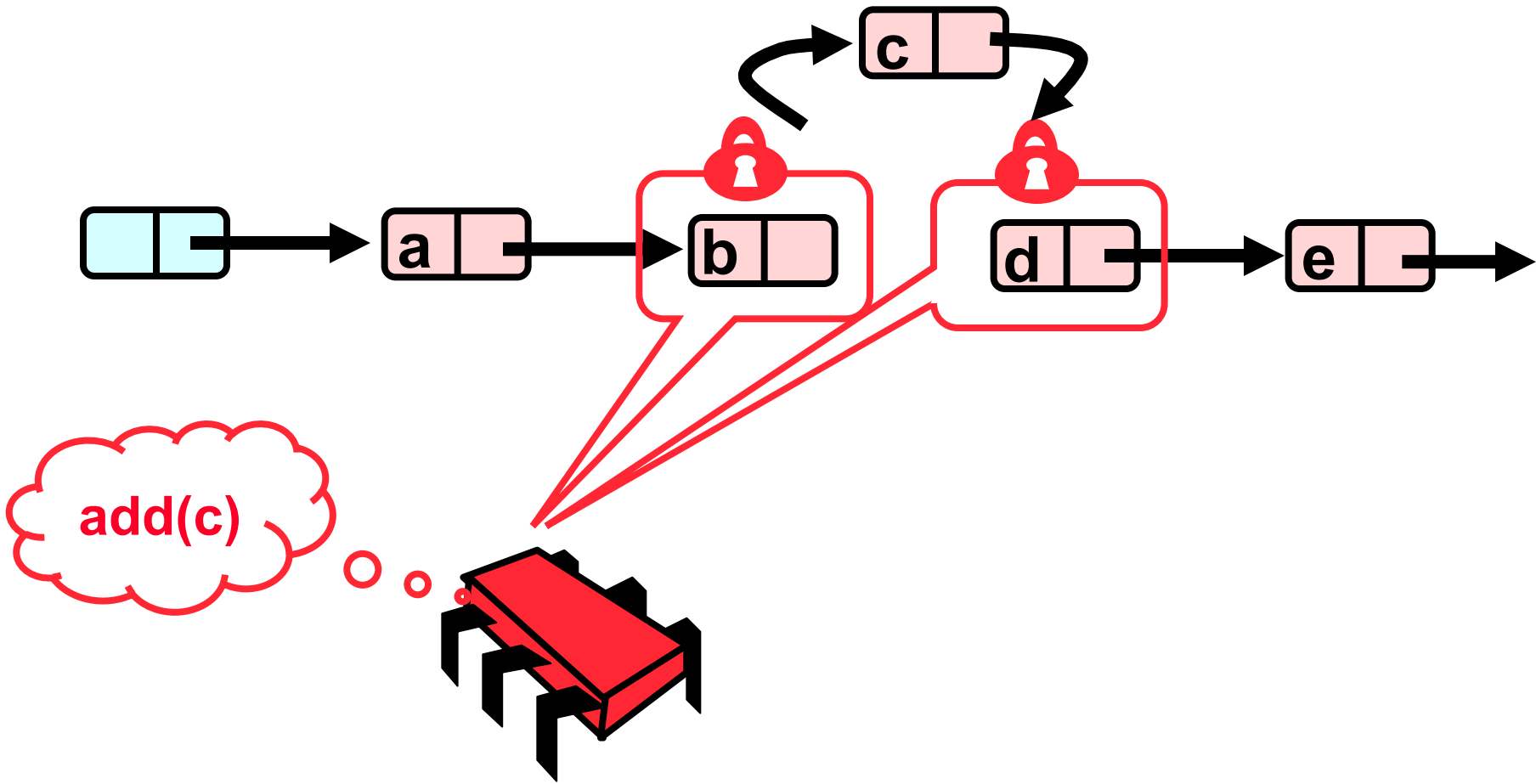
Optimistic: Traverse without Locking



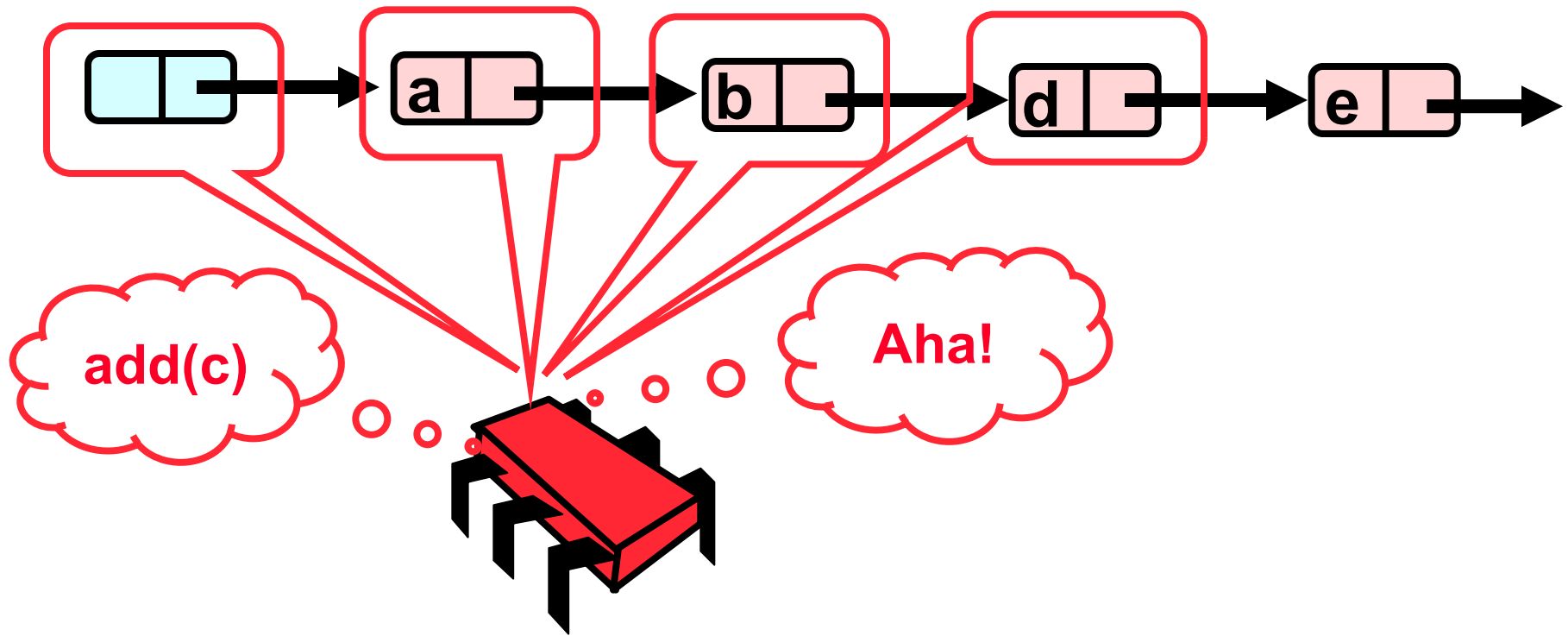
Optimistic: Lock and Load



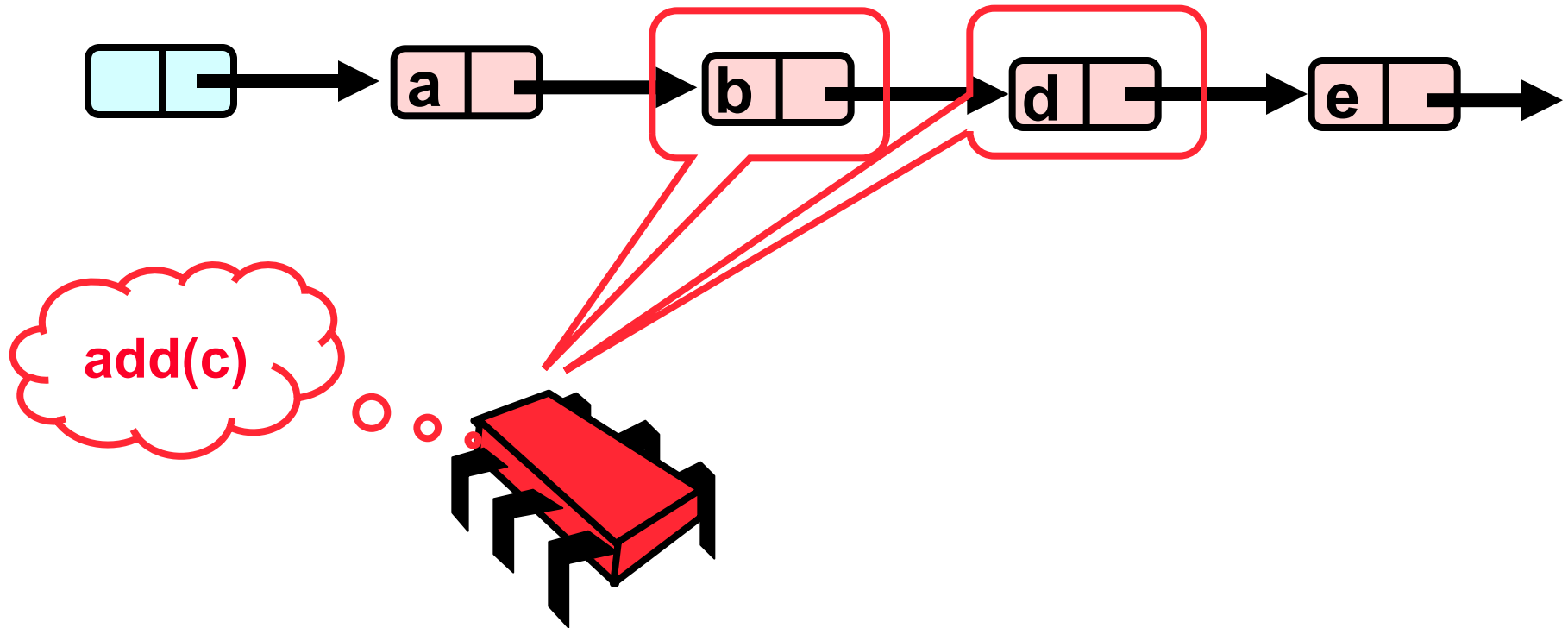
Optimistic: Lock and Load



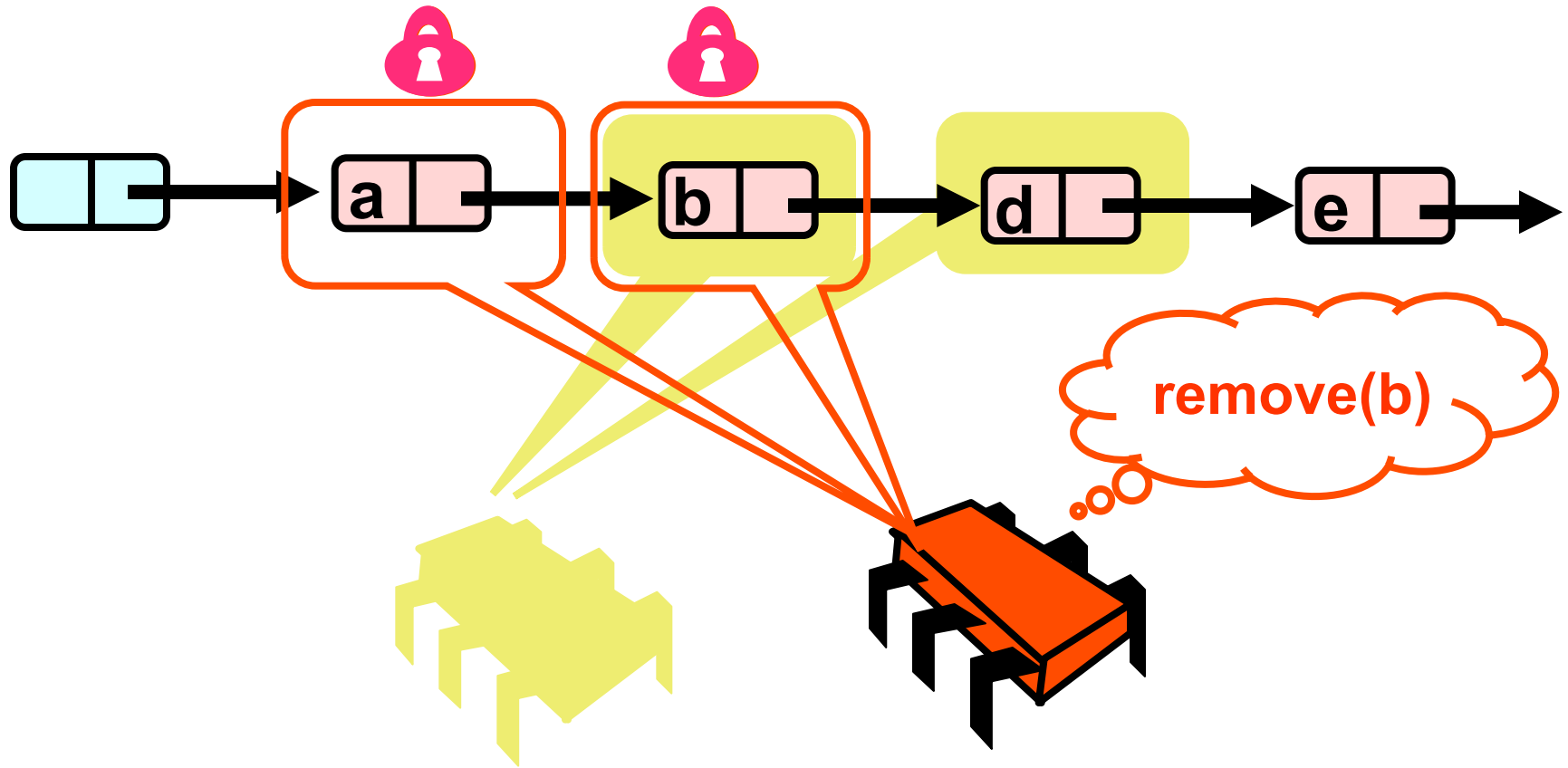
What could go wrong?



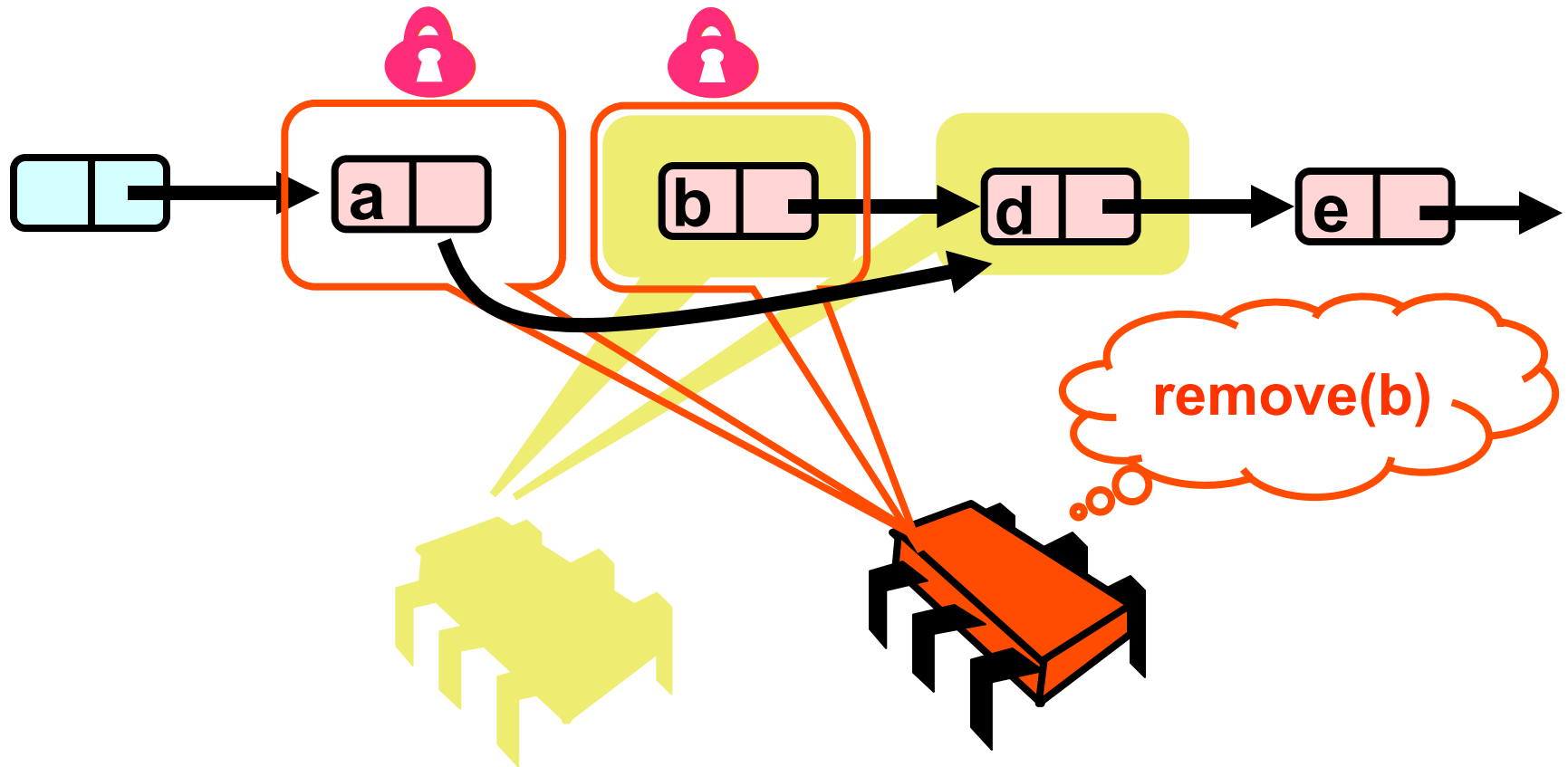
What could go wrong?



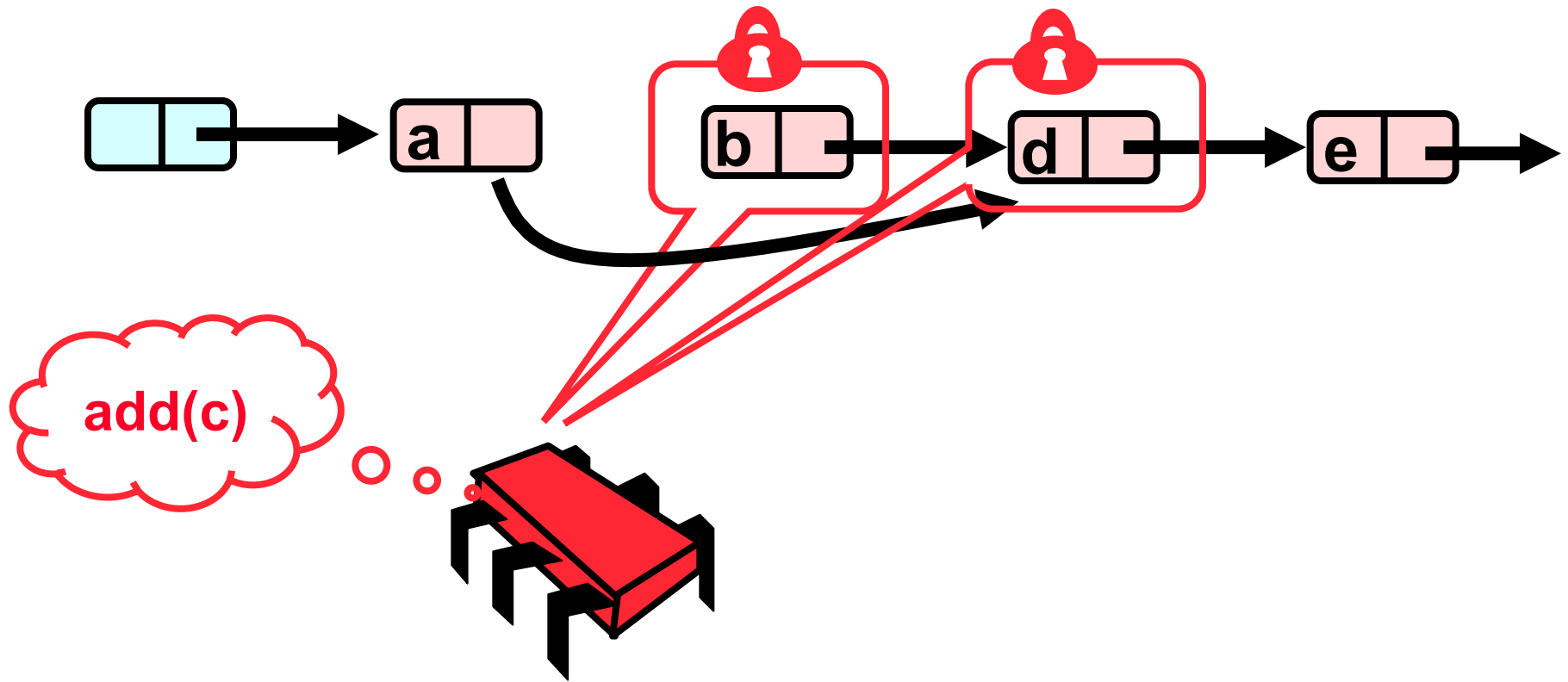
What could go wrong?



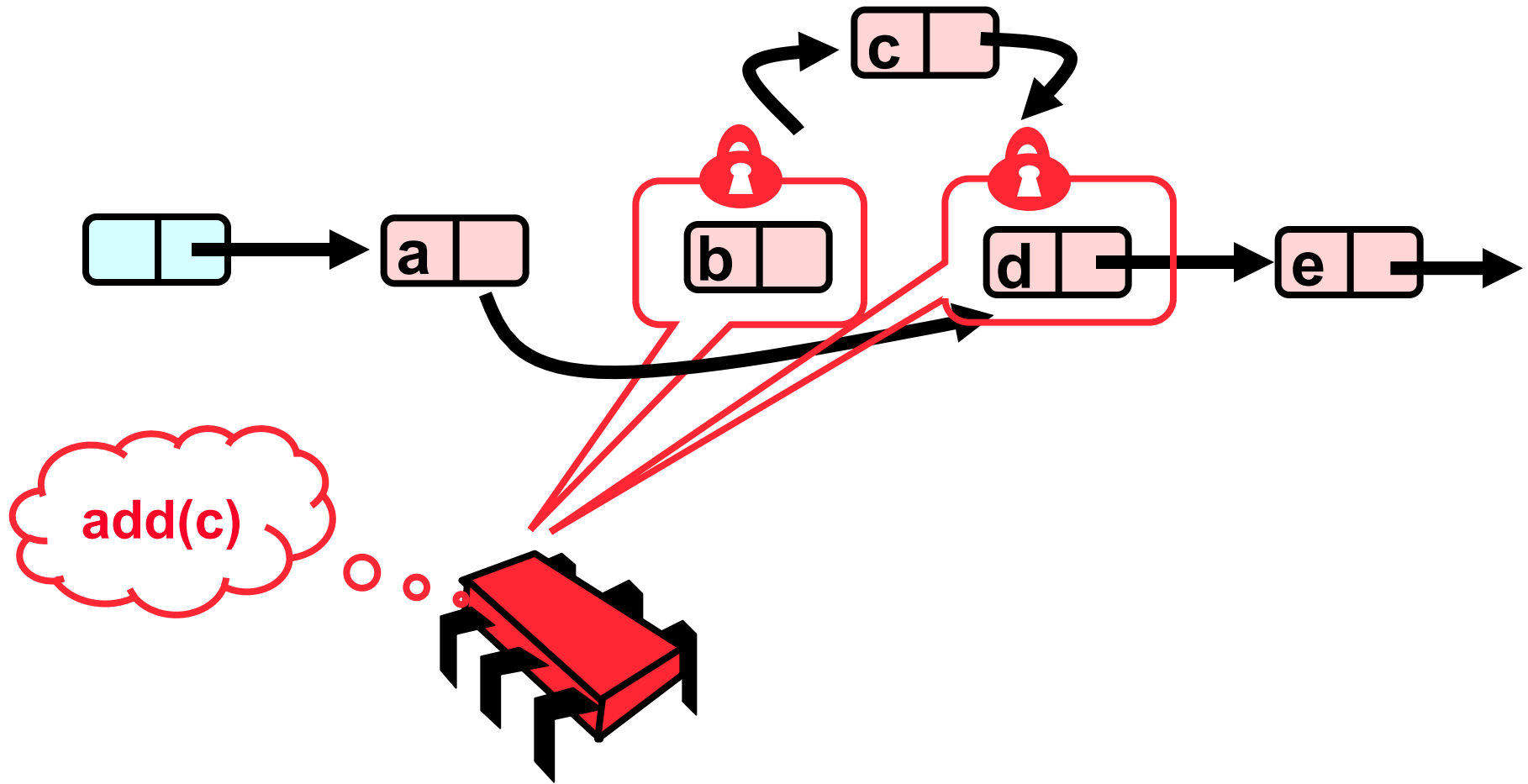
What could go wrong?



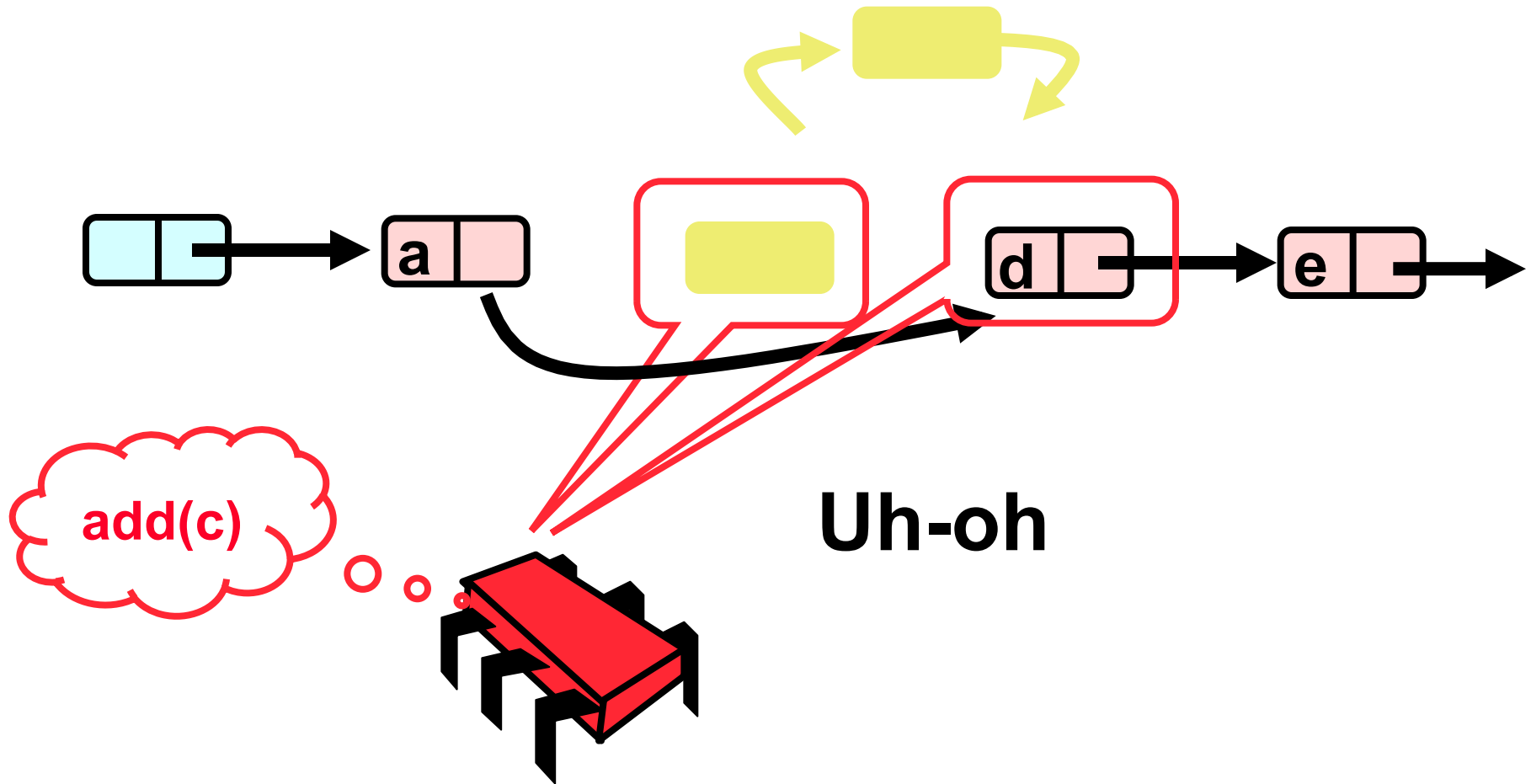
What could go wrong?



What could go wrong?



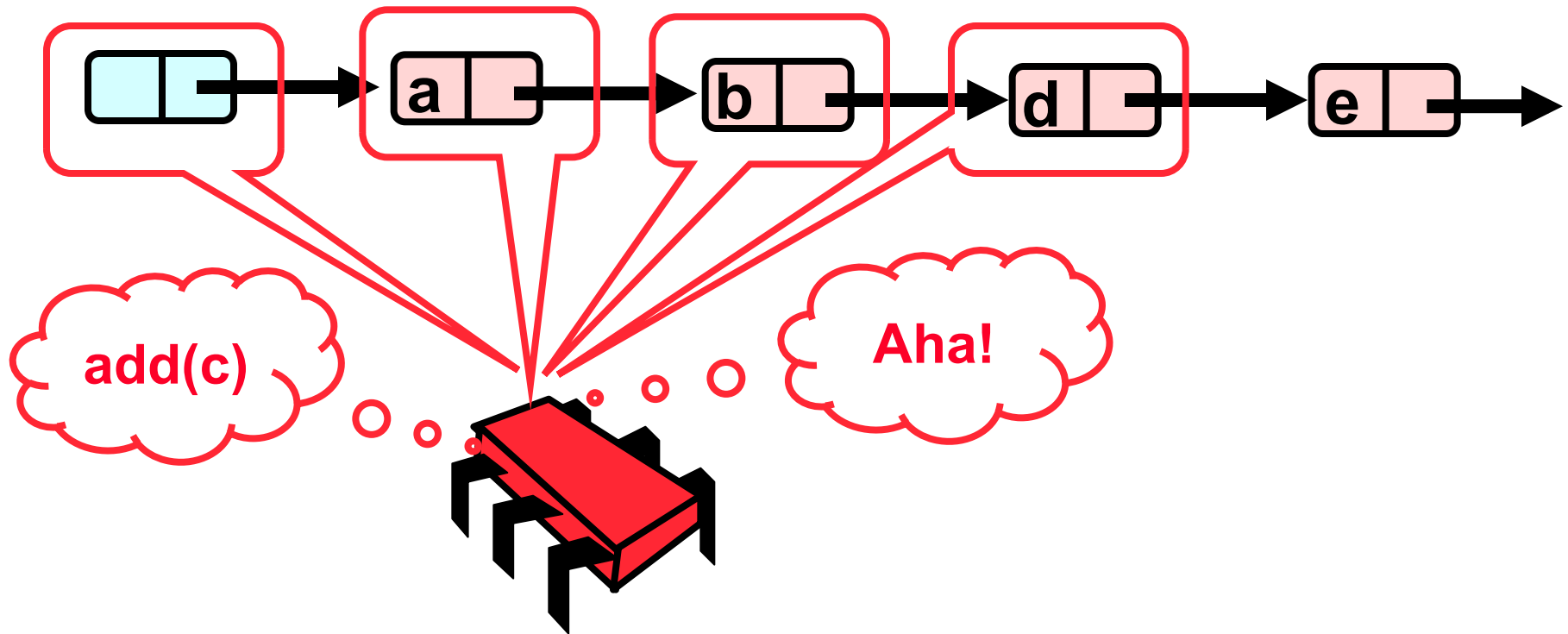
What could go wrong?



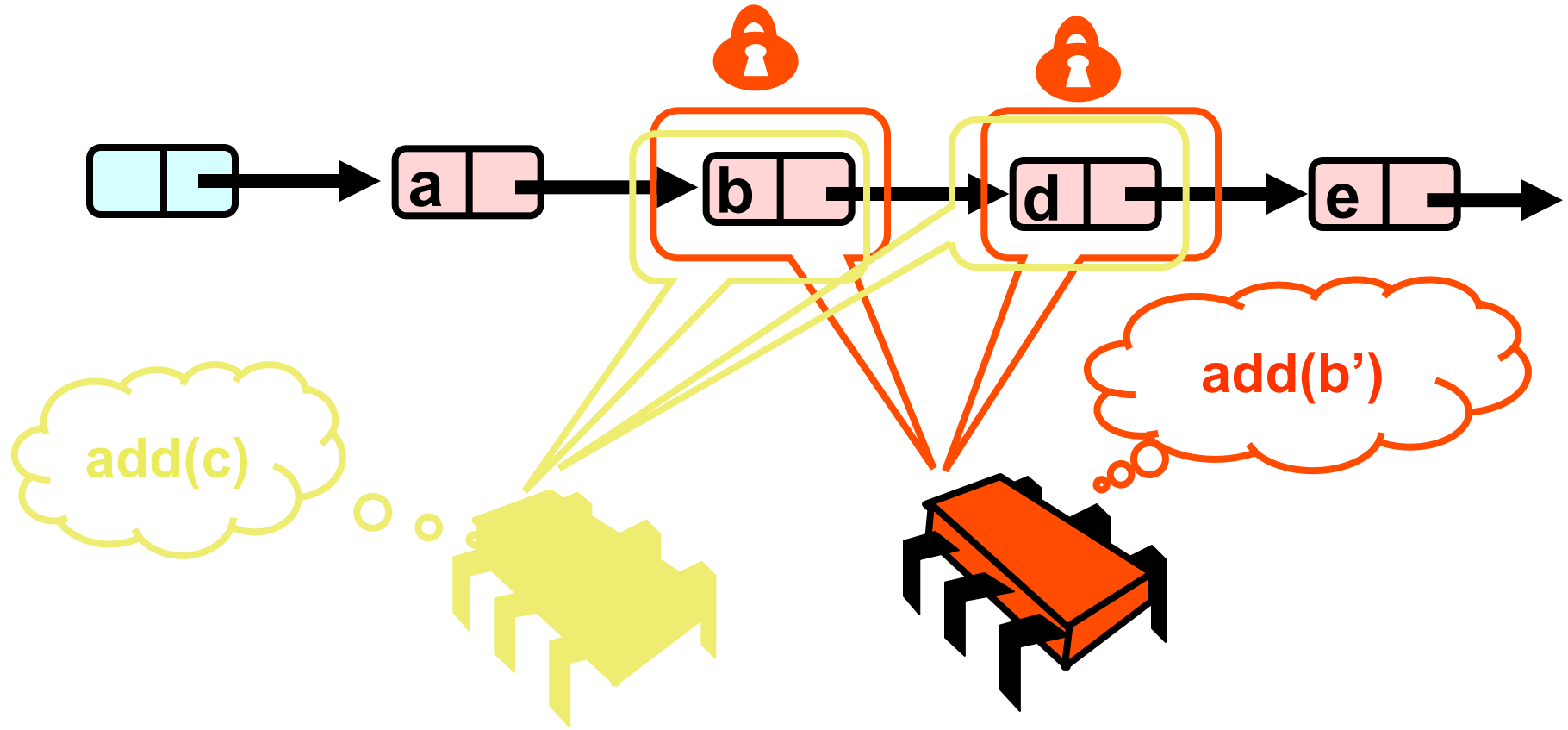
Check after lock

- ▶ By traversing optimistically, we give up any guarantees we had about the list during traversal
- ▶ The node we locked might have just been removed
- ▶ Need to check the pointers to it

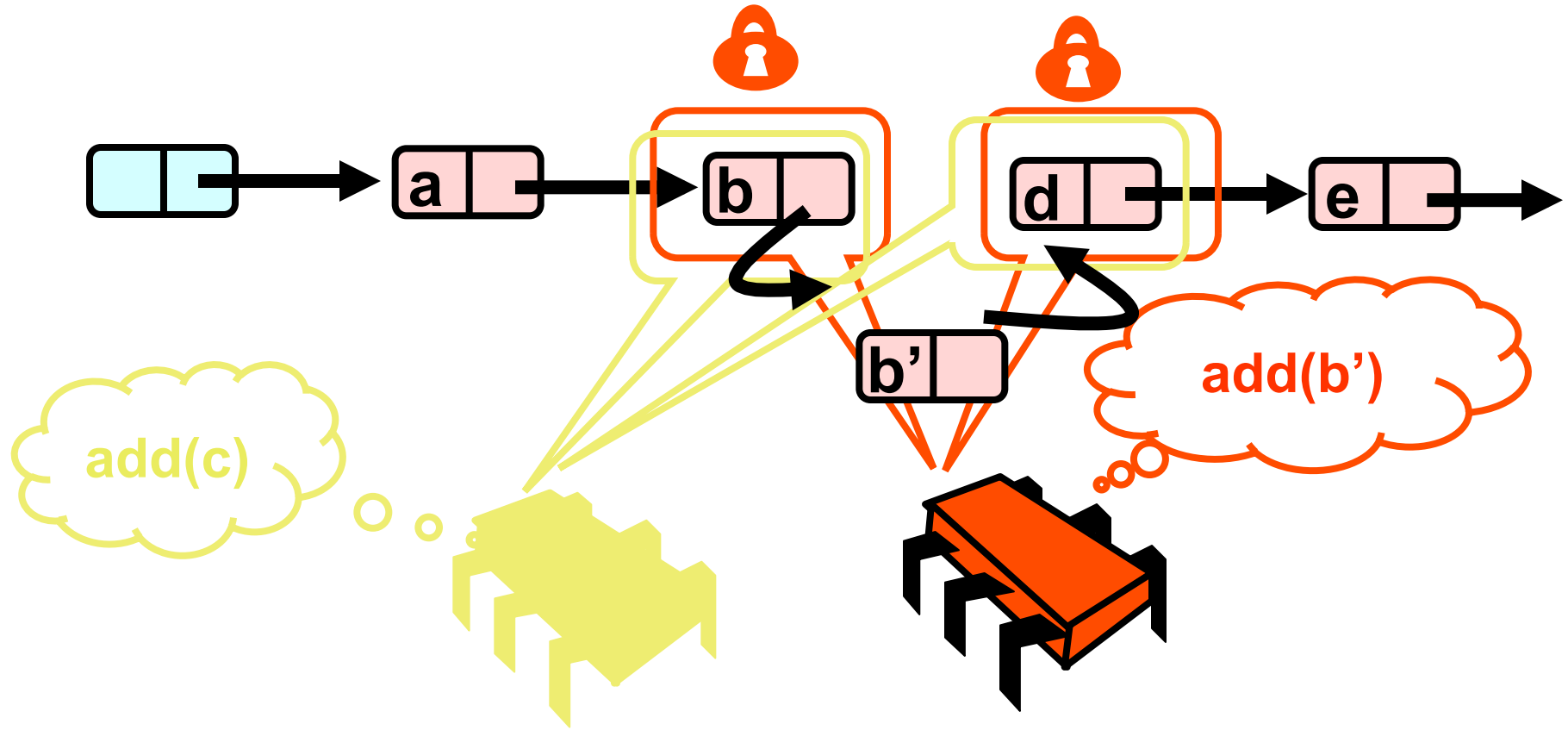
What Else Could Go Wrong?



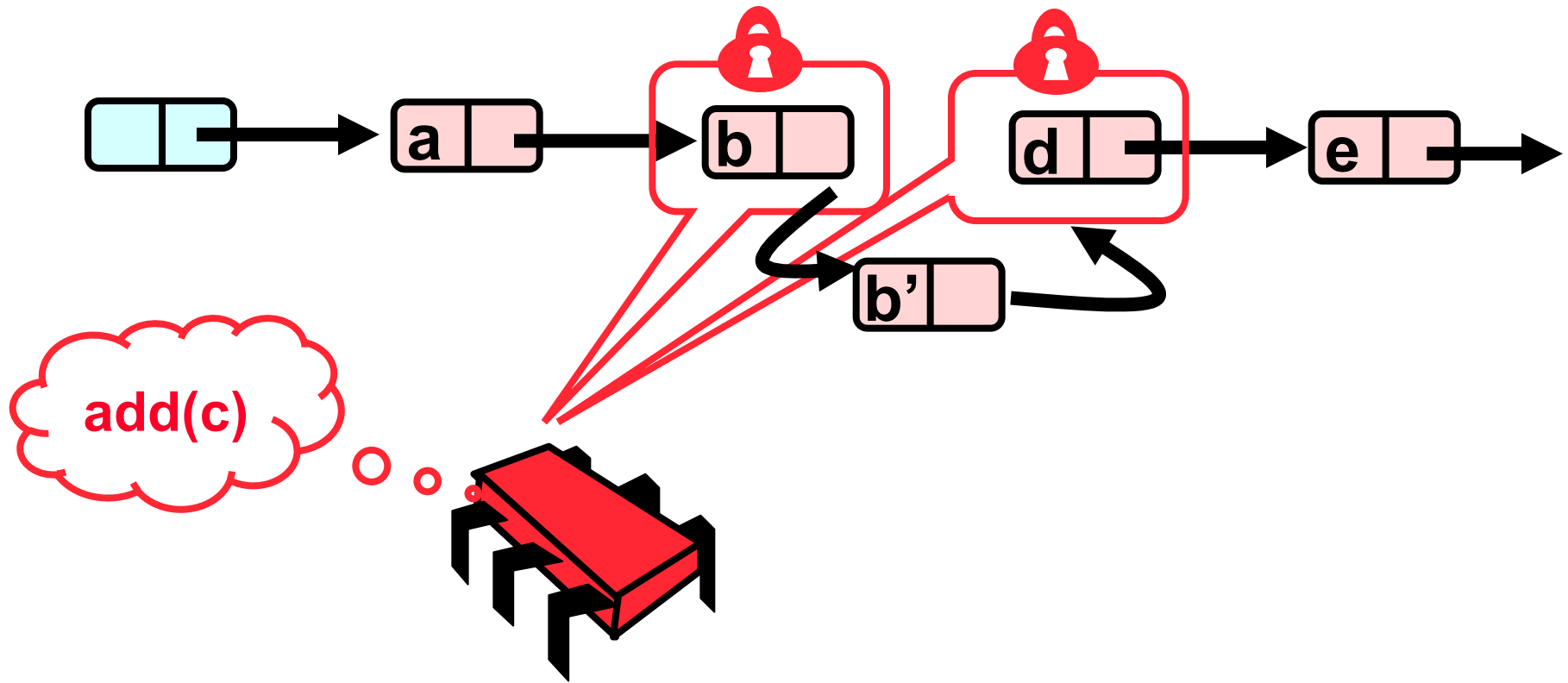
What Else Could Go Wrong?



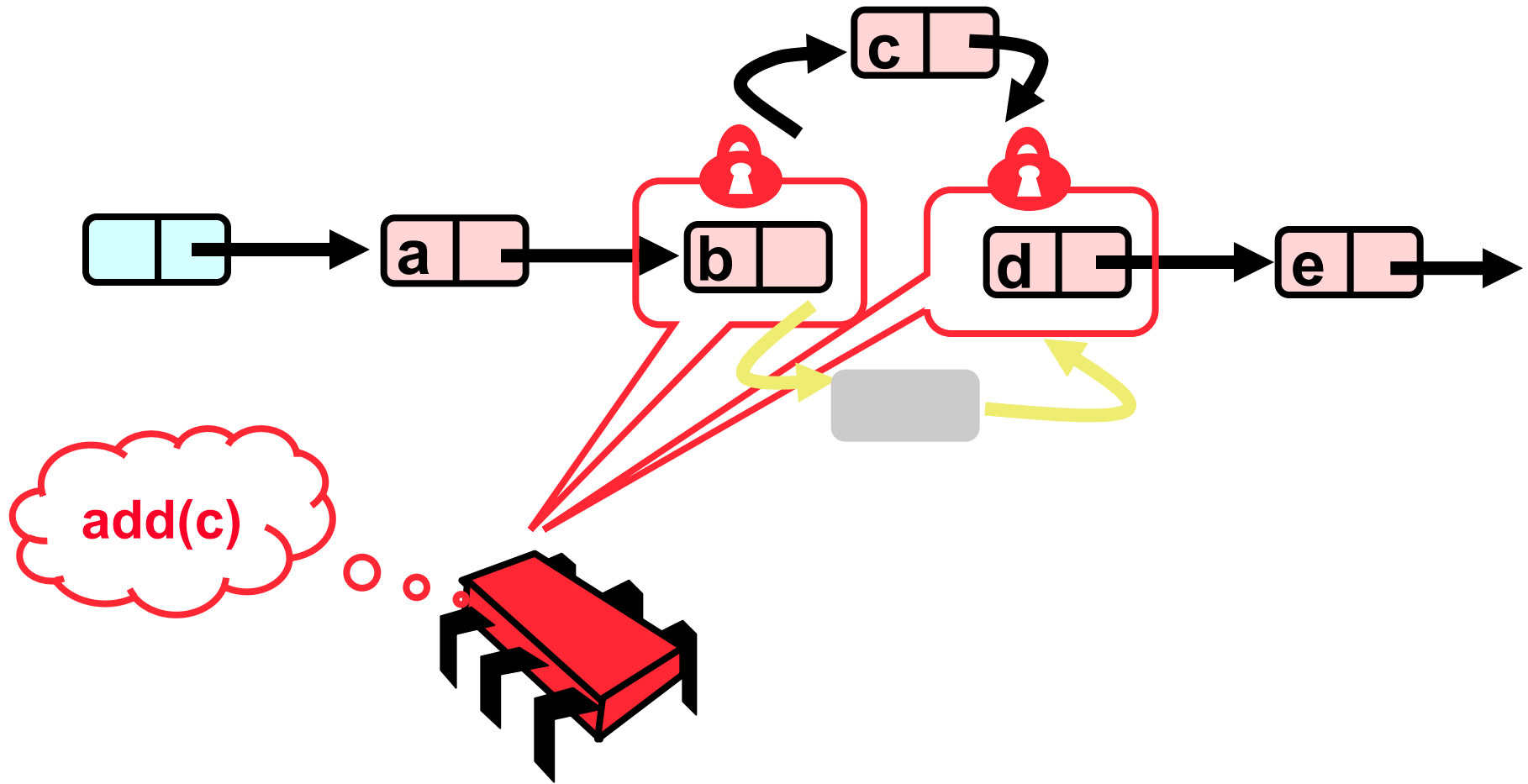
What Else Could Go Wrong?



What Else Could Go Wrong?



What Else Could Go Wrong?



Check after lock

- ▶ By traversing optimistically, we give up any guarantees we had about the list during traversal
- ▶ New nodes might just have been added
- ▶ Need to check the pointers between the locked nodes

Optimistic Traversal

- ▶ Traverse optimistically, get the lock and then check if we can move on
- ▶ Need to check after we get the lock
 - ▷ We know nothing about the items locked
 - ▷ Need to check if they are in the same situation as they were before we locked
 - ▷ After checking, we know that we hold the lock, and thus they cannot suffer further changes

Other patterns: Lazy Synchronization

- ▶ Postpone hard work
- ▶ E.g., break remove into two parts
 - ▷ Logical removal → marks component to be deleted
 - ▷ Physical removal → do what needs to be done

Reminder: Lock-Free Data Structures



- ▶ No matter what ...
 - ▷ Guarantees minimal progress in any execution
 - ▷ i.e. Some thread will always complete a method call
 - ▷ Even if others halt at malicious times
 - ▷ Implies that implementation can't use locks

Recall: Using atomics

- ▶ `compareAndSet(expectedValue,newValue)` method
 - ▷ Compares a variable with an “expectedValue” given as input
 - ▷ Sets it to “newValue” if comparison is successful
 - ▷ Does everything atomically

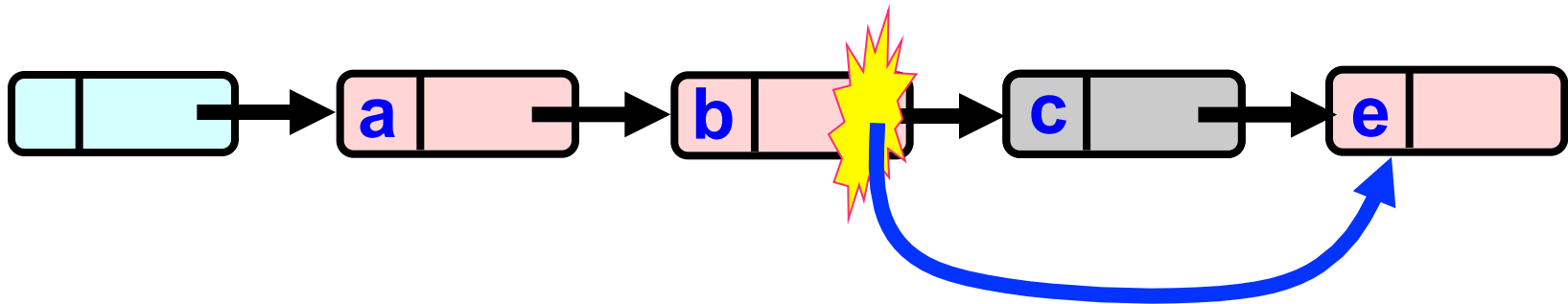
Lock-free Lists

- ▶ Next logical step
 - ▷ Wait-free contains()
 - ▷ lock-free add() and remove()

- ▶ How about turning adds/removes into atomics?
 - ▷ Use compareAndSet() or CAS

What could go wrong with only CAS?

Lock-free Lists

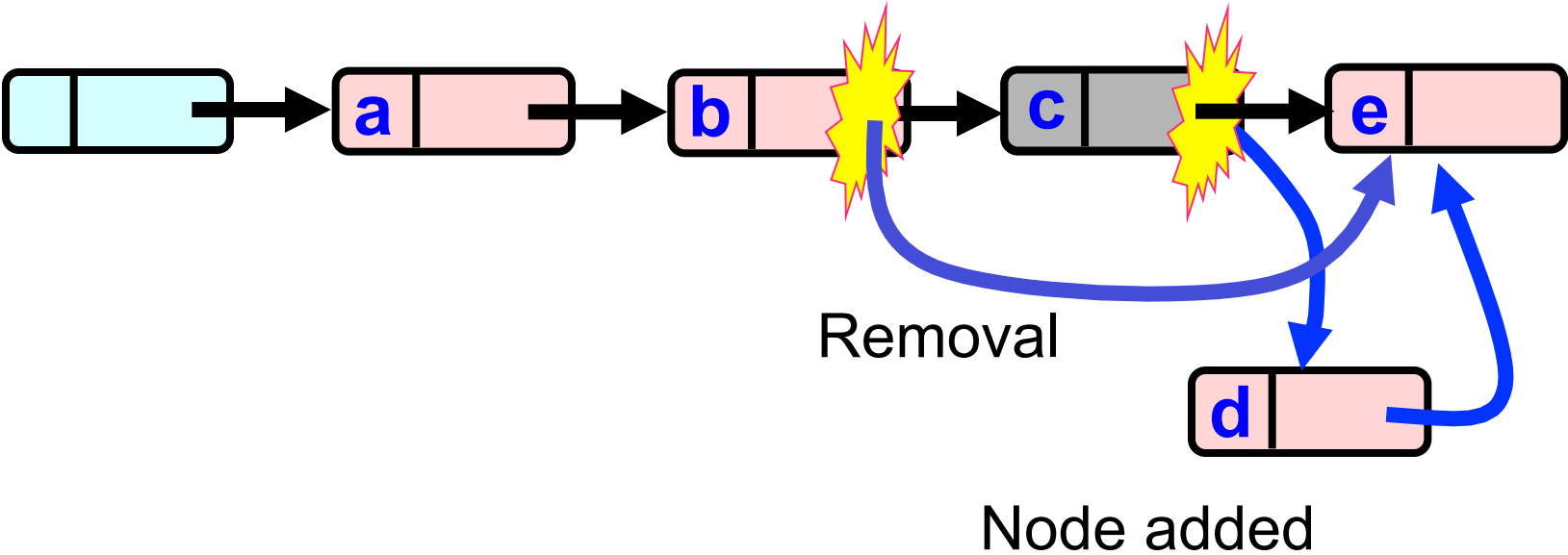


Use CAS to verify pointer
is correct

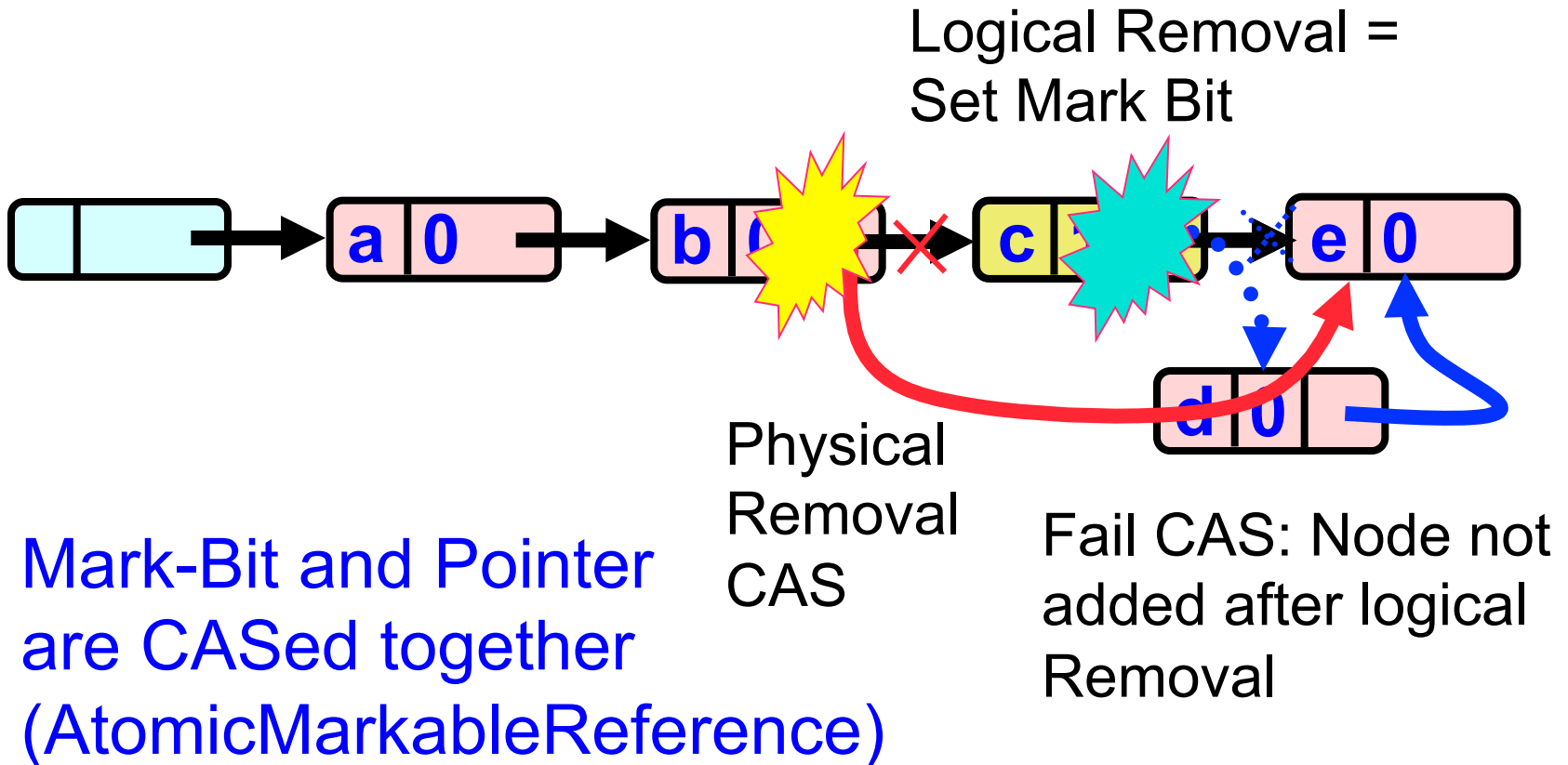
Removal

Not enough!

Problem...



The Solution: Combine Bit and Pointer

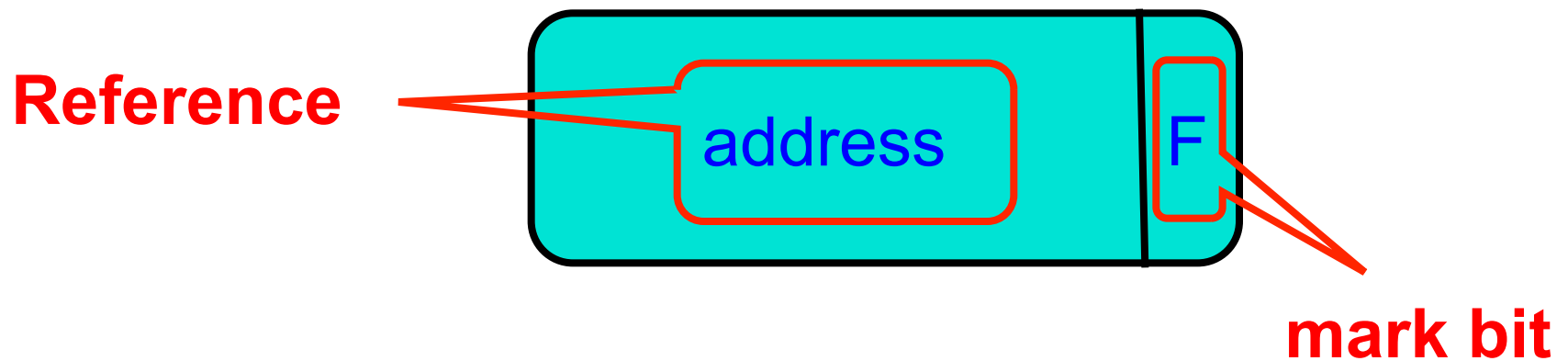


Solution

- ▶ Use AtomicMarkableReference
- ▶ Atomically
 - ▷ Swing reference and
 - ▷ Update flag
- ▶ Remove in two steps
 - ▷ Set mark bit in next field
 - ▷ Redirect predecessor's pointer

Marking a Node

- ▶ **AtomicMarkableReference** class
 - ▷ `Java.util.concurrent.atomic` package



Extracting Reference & Mark

```
Public Object get(boolean[] marked) ;
```

Extracting Reference & Mark

```
Public Object get (boolean [] marked) ;
```

**Returns
reference**

**Returns mark at array
index 0!**
**(funny use of Java arrays
to pass a pointer)**

Extracting Mark Only

```
public boolean isMarked();
```

**Value of
mark**

Changing State

```
Public boolean compareAndSet(  
    Object expectedRef,  
    Object updateRef,  
    boolean expectedMark,  
    boolean updateMark);
```

Changing State

If this is the current
reference ...

```
Public boolean compareAndSet(  
Object expectedRef,  
Object updateRef,  
boolean expectedMark,  
boolean updateMark);
```

And this is the
current mark ...

Changing State

...then change to this
new reference ...

```
Public boolean compareAndSet(  
    Object expectedRef,  
    Object updateRef,  
    boolean expectedMark,  
    boolean updateMark) ;
```

... and this new
mark

Changing State

```
public boolean attemptMark(  
    Object expectedRef,  
    boolean updateMark);
```

Changing State

```
public boolean attemptMark(  
    Object expectedRef,  
    boolean updateMark);
```

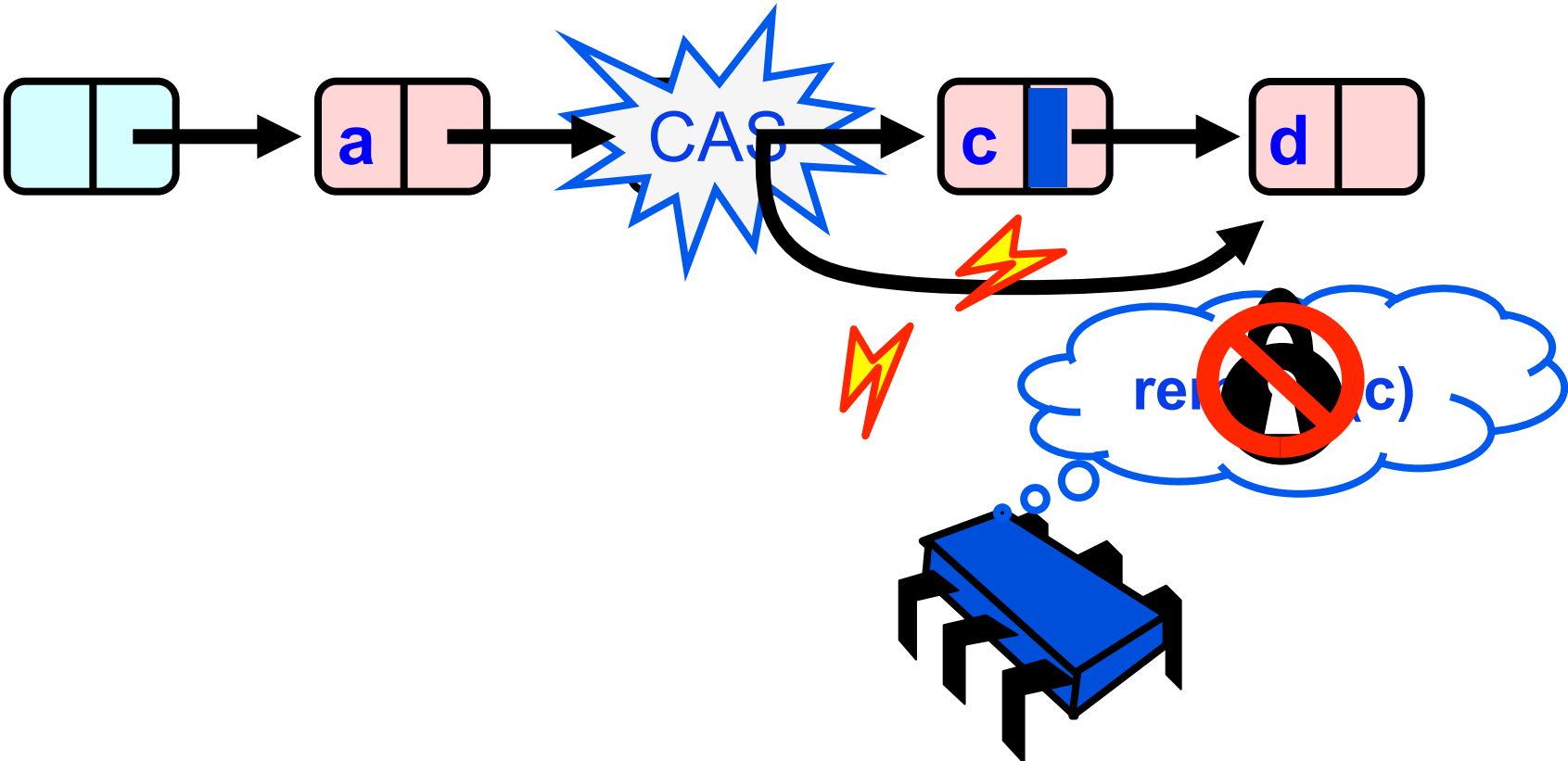
**If this is the current
reference ...**

Changing State

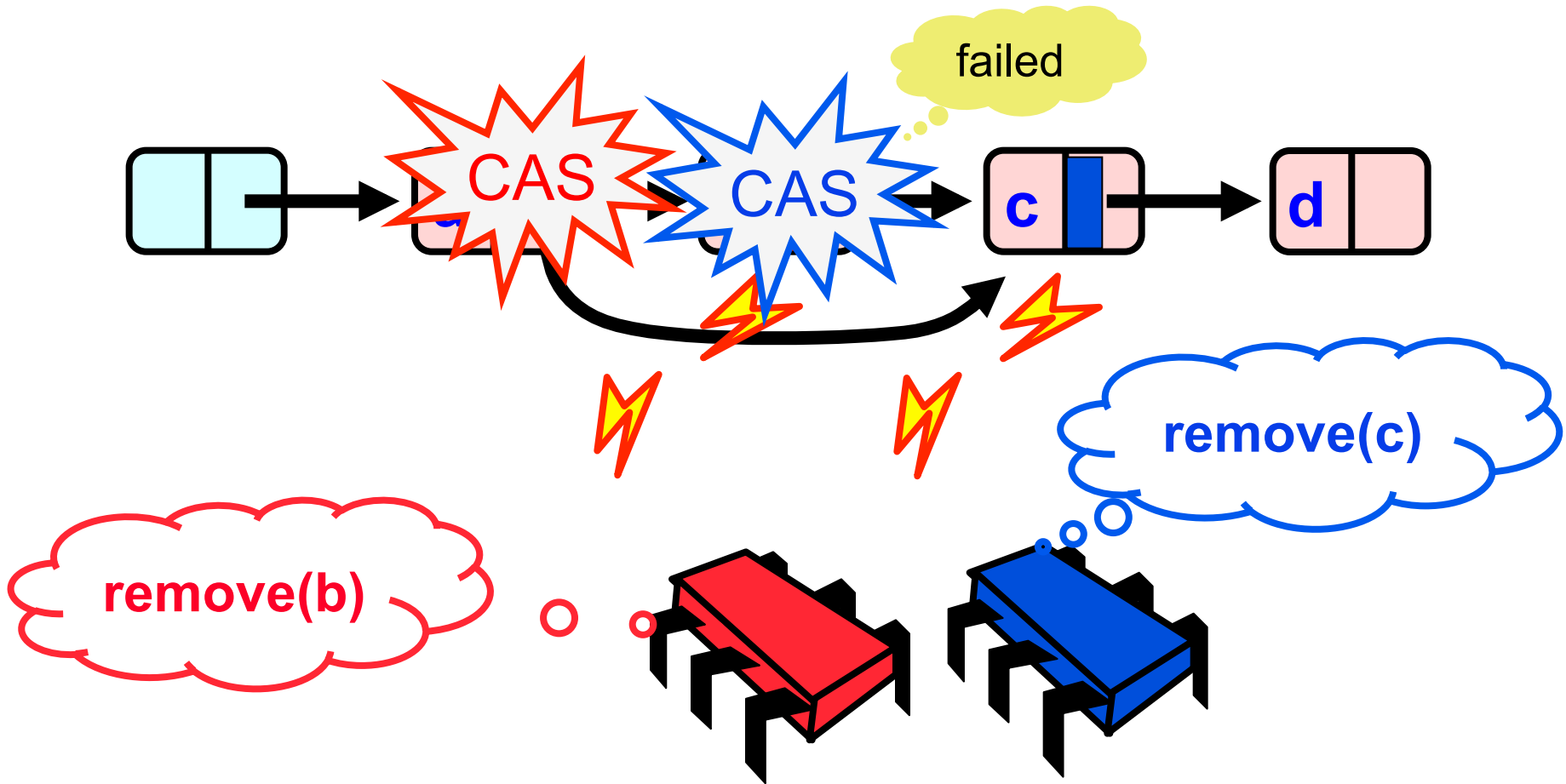
```
public boolean attemptMark(  
    Object expectedRef,  
    boolean updateMark);
```

**.. then change to
this new mark.**

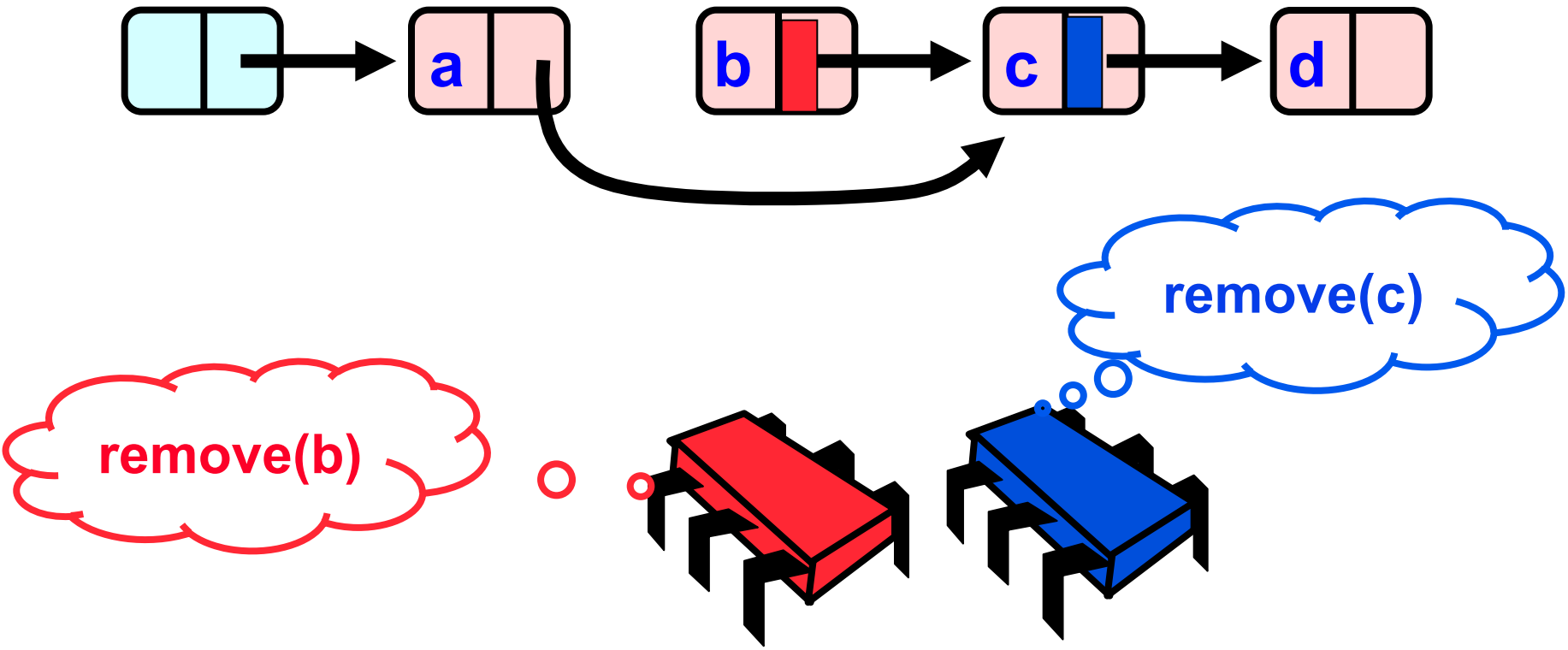
Removing a Node



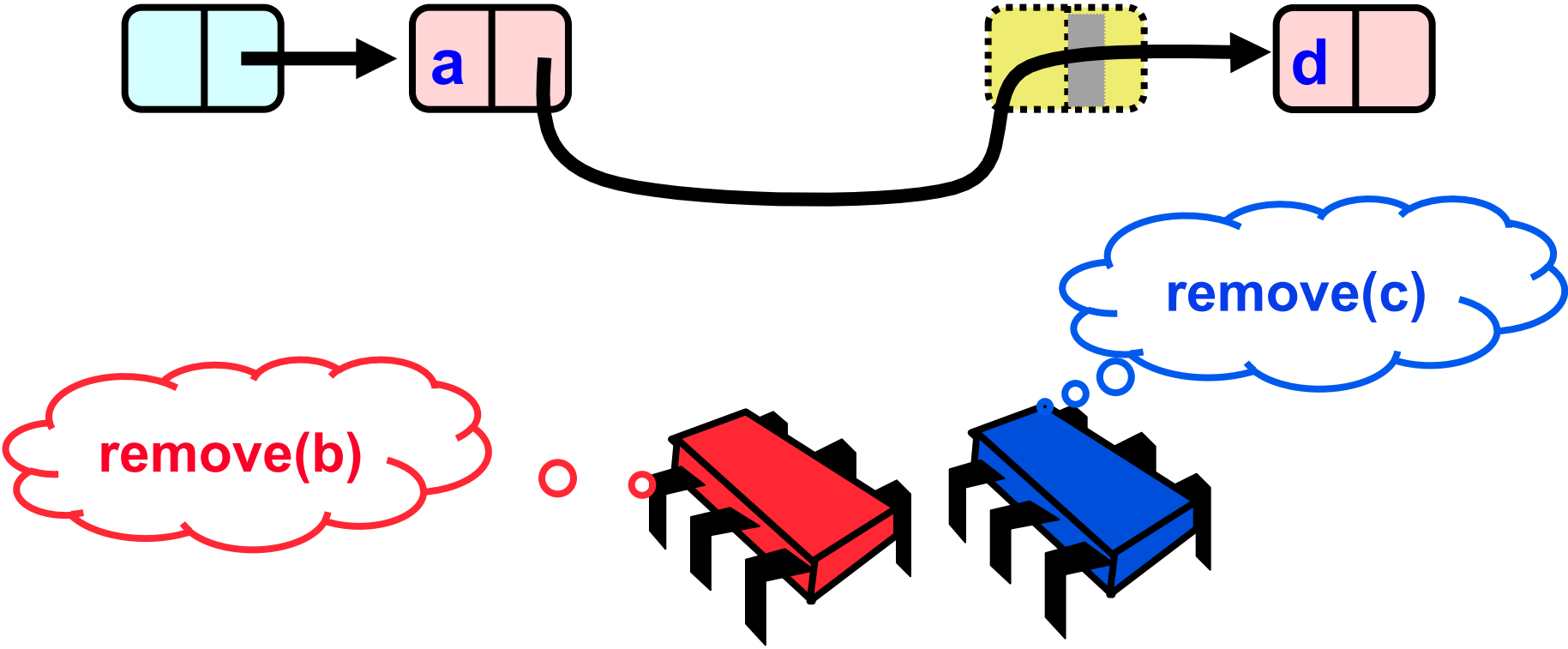
Removing a Node



Removing a Node



Removing a Node



Summary

- ▶ Coarse-grained locking
- ▶ Fine-grained locking
 - ▷ Basic synchronization
 - ▷ Optimistic synchronization
 - ▷ Lazy synchronization
- ▶ Lock-free synchronization