CSE113: Parallel Programming

Feb. 11, 2022

- **Topics**:
  - General concurrent sets

Announcements

- Midterm is out!
  - You have until next Monday at midnight to do it.
  - Do not discuss with your classmates
  - Do not google specific questions or ask on online forums
  - Ask any clarifying questions as a private post on piazza
  - Late tests will not be accepted (prioritize the midterm!)
  - You can ask me or Reese about the midterm, not Tim or Sanya

Don’t expect help on Piazza on the weekend or after 5 PM

- Homework 3 is out
  - You should have everything you need by end of today
  - Due next Friday by midnight
- Grades for HW 1 are released
  - You have until next Tuesday to discuss any issues

Announcements

- You can start sharing results for HW 3 on Monday

Today’s Quiz

- Due Monday by class time. Please do it!

Previous quiz

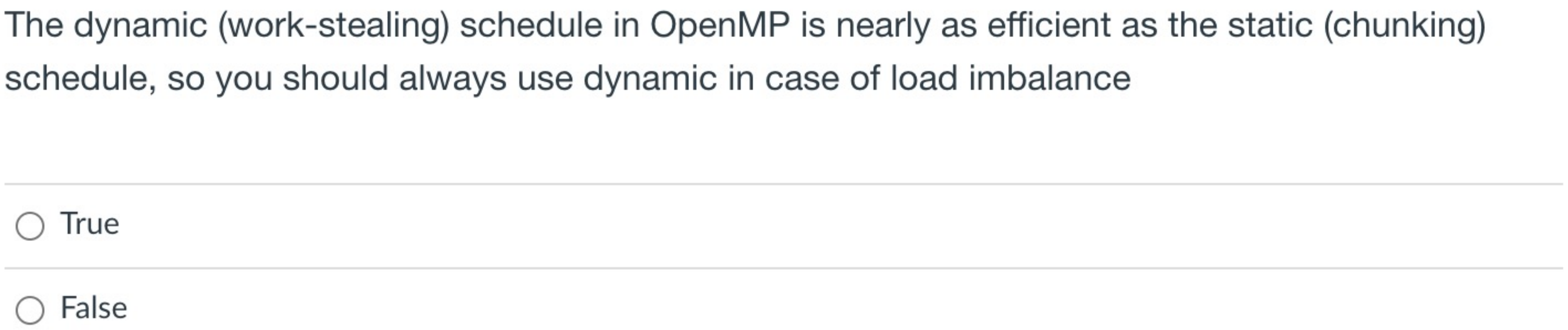
OpenMP does NOT allow you to specify the following properties when specifying that a DOALL loop should be executed in parallel

○ Number of threads

○ whether to use a fair mutex

○ what parallel schedule

Previous quiz

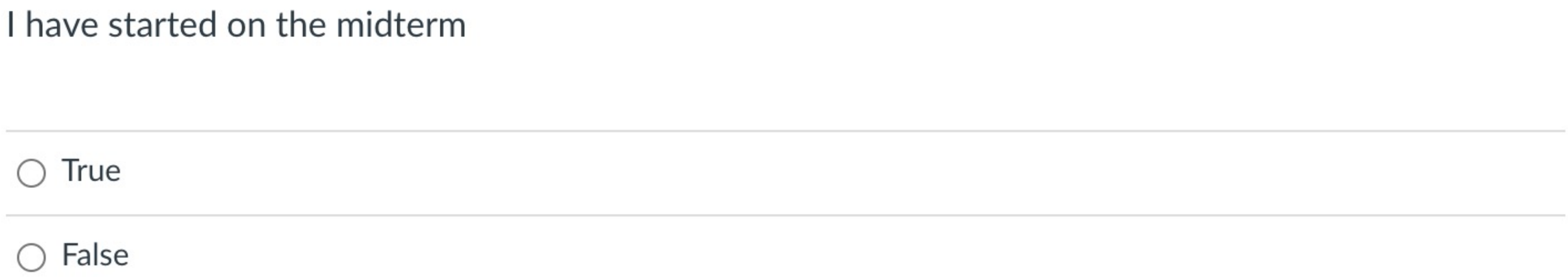


The dynamic (work-stealing) schedule in OpenMP is nearly as efficient as the static (chunking) schedule, so you should always use dynamic in case of load imbalance

○ True

○ False

Previous quiz

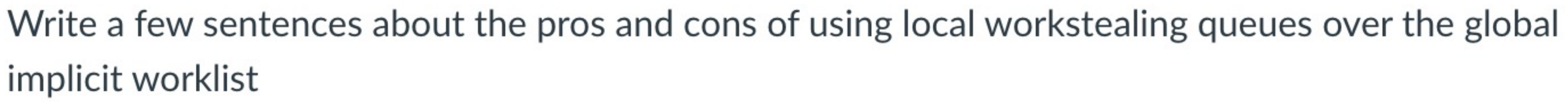


I have started on the midterm

○ True

○ False

Previous quiz



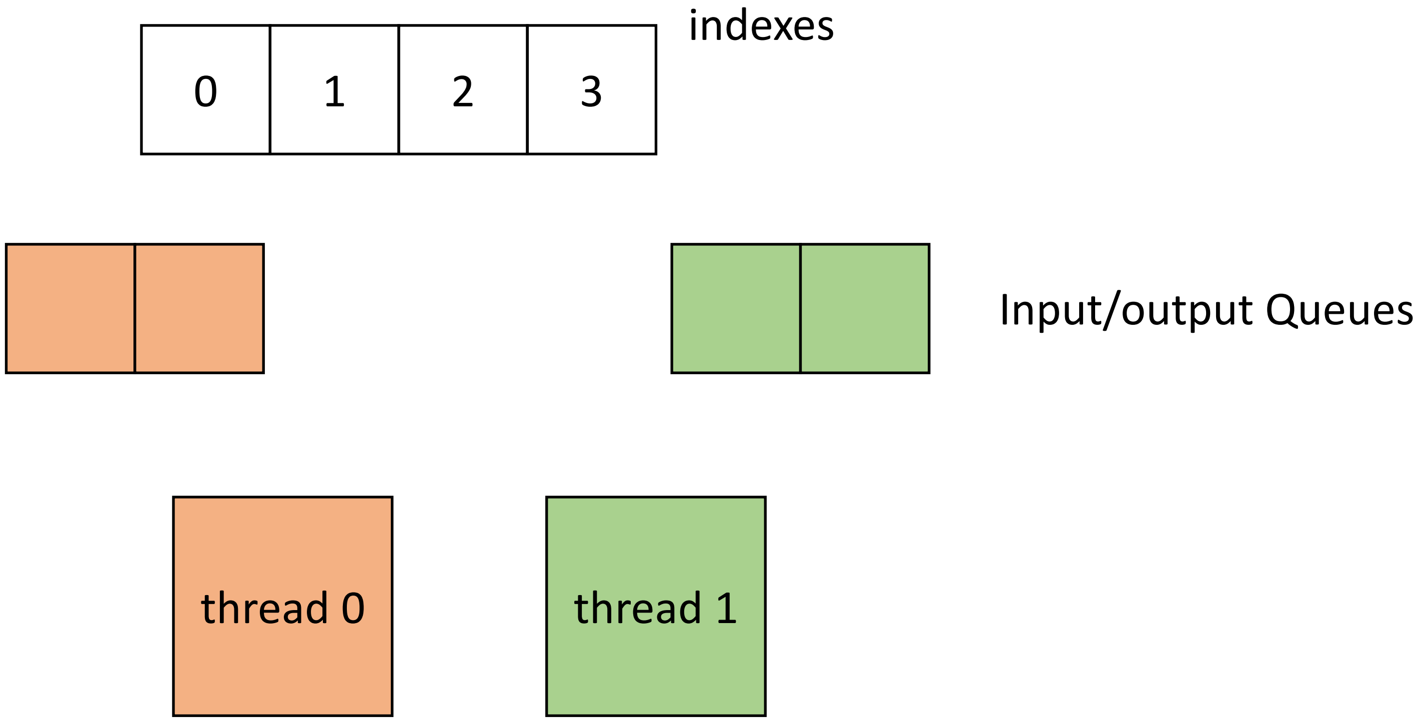
Write a few sentences about the pros and cons of using local workstealing queues over the global implicit worklist

Review

Local worklists

Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



0	1	2	3

indexes

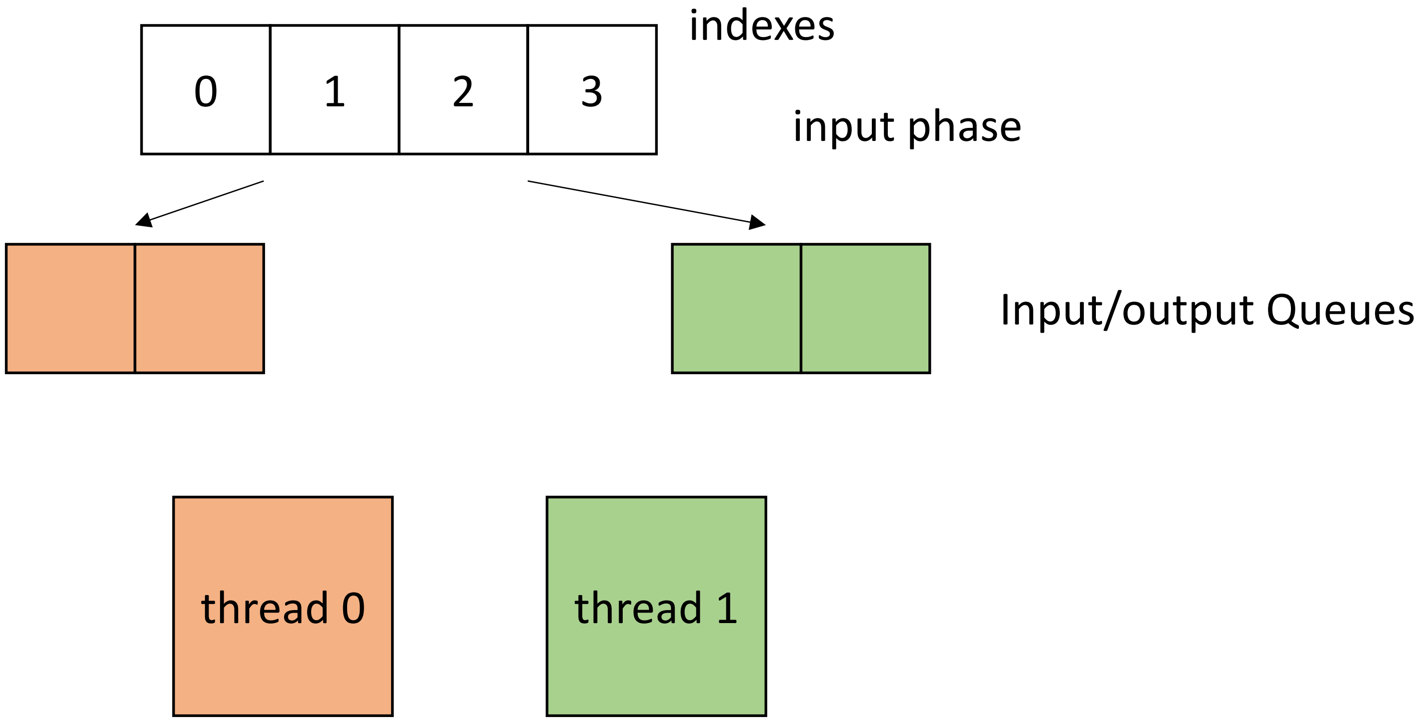
Input/output Queues

thread 0

thread 1

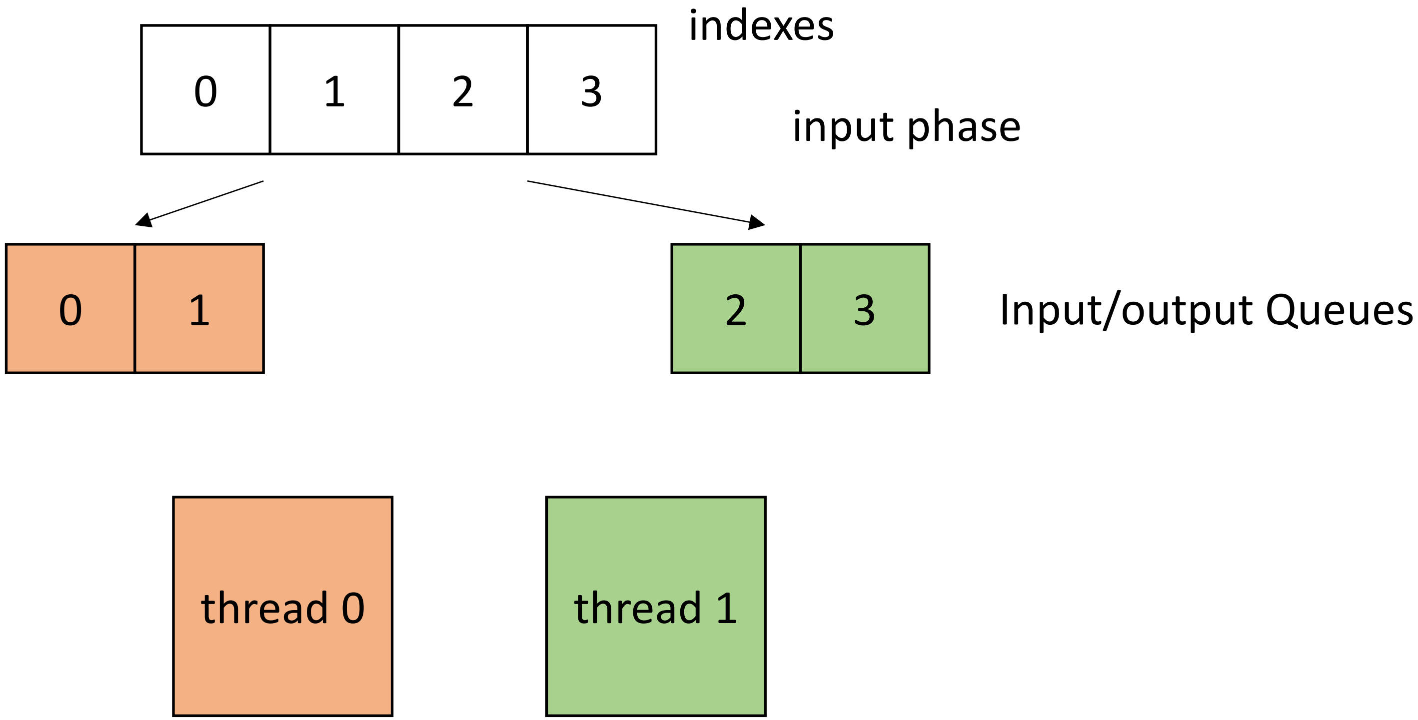
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



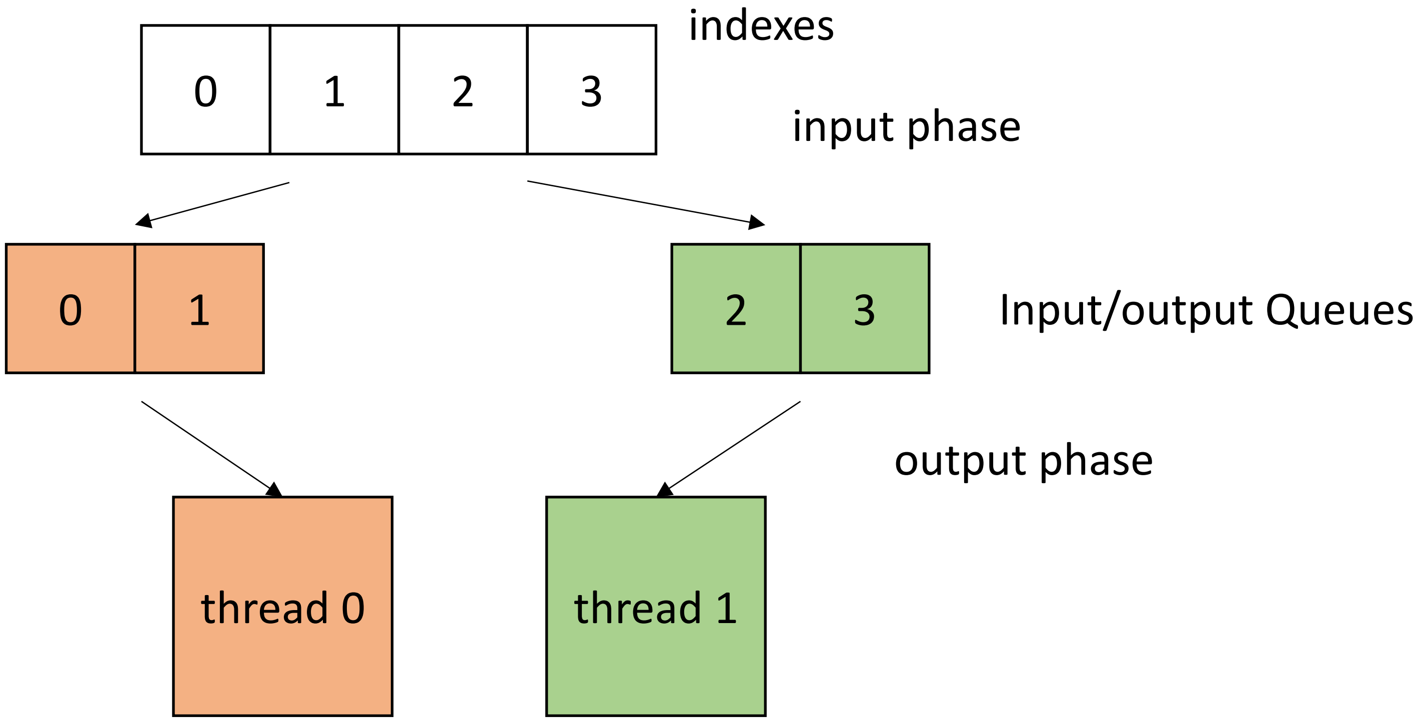
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread

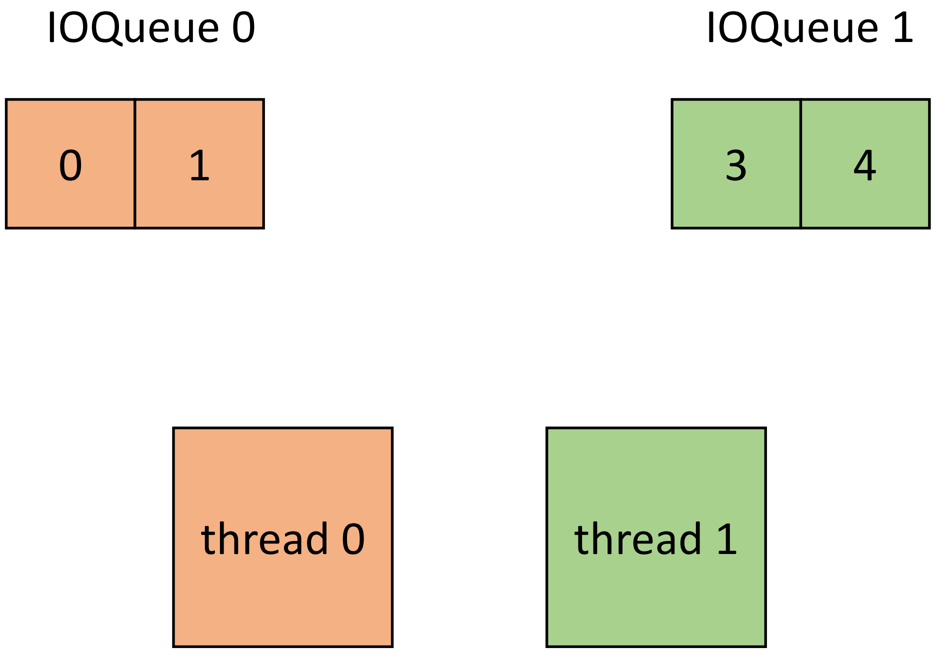


Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



Work stealing - local worklists



```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

OpenMP

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 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

New material

Schedule

- C++ Atomic Template

- Concurrent set
    - Coarse-grained lock
    - fine-grained lock
    - optimistic locking

C++ Atomic template

Schedule

- C++ Atomic Template

- **Concurrent set**
    - Coarse-grained lock
    - fine-grained lock
    - optimistic locking

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

Set Interface

- Unordered collection of items
- No duplicates

- We will implement this as a sorted linked list

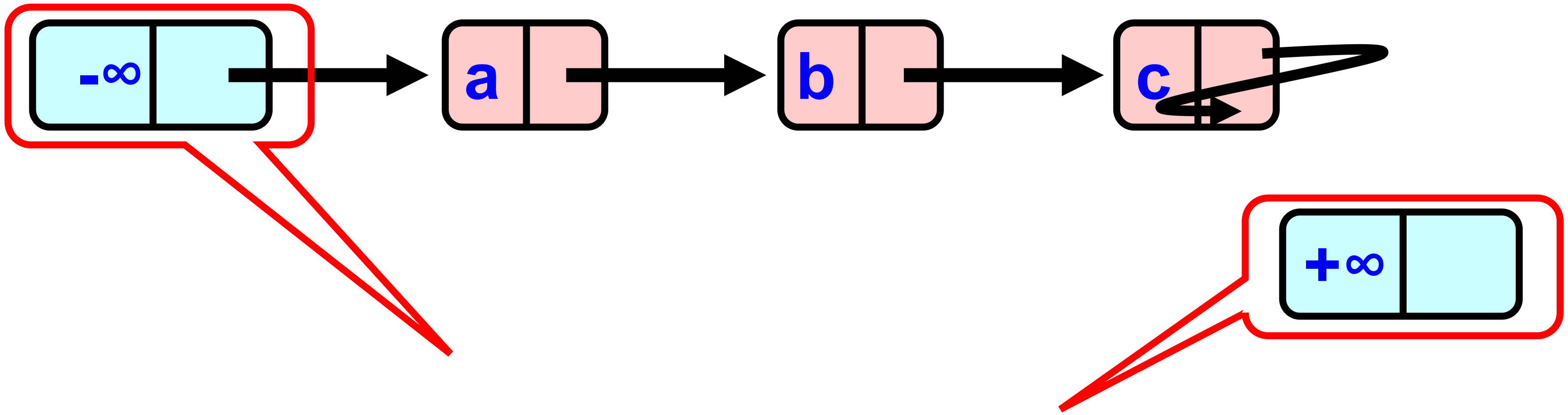
Set Interface

- Unordered collection of items
- No duplicates
- Methods
    - **`add(x)`** put **`x`** in set
    - **`remove(x)`** take **`x`** out of set
    - **`contains(x)`** tests if **`x`** in set

List Node

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

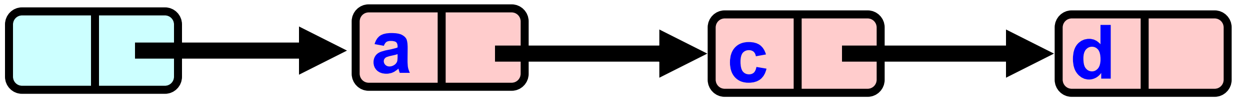
The List-Based Set



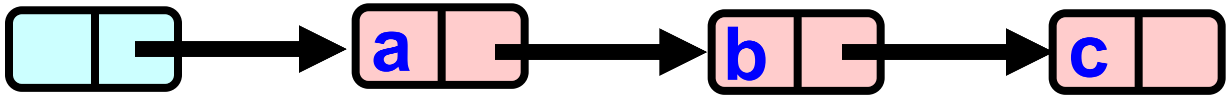
Sorted with Sentinel nodes
(min & max possible keys)

Sequential List Based Set

add(b)

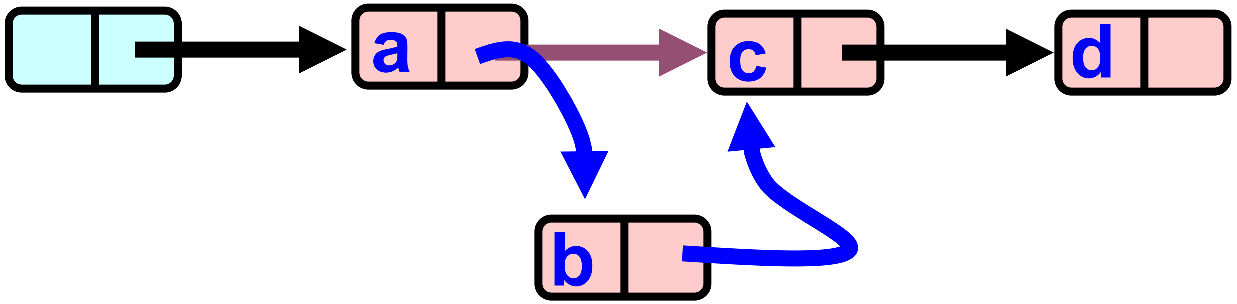


remove(b)

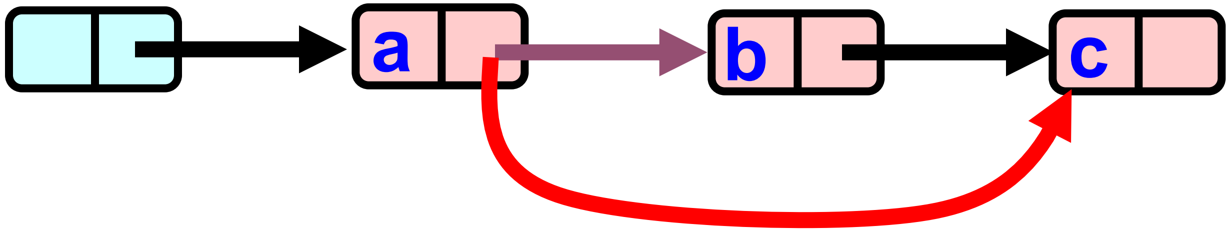


Sequential List Based Set

add(b)



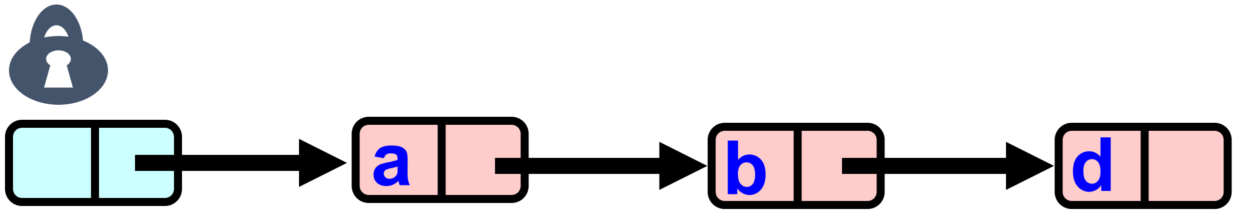
remove(b)



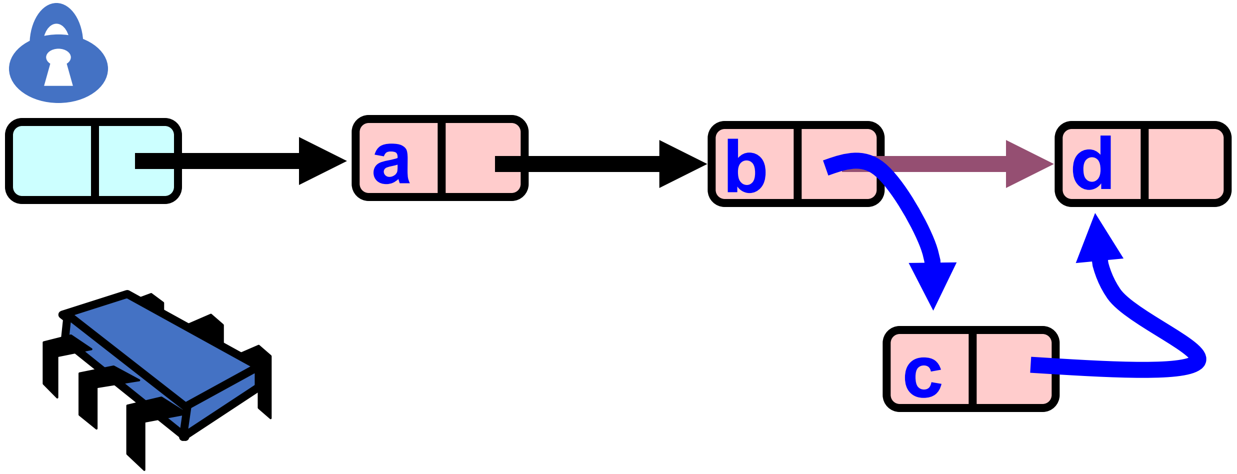
Schedule

- C++ Atomic Template
- Concurrent set
    - **Coarse-grained lock**
    - fine-grained lock
    - optimistic locking

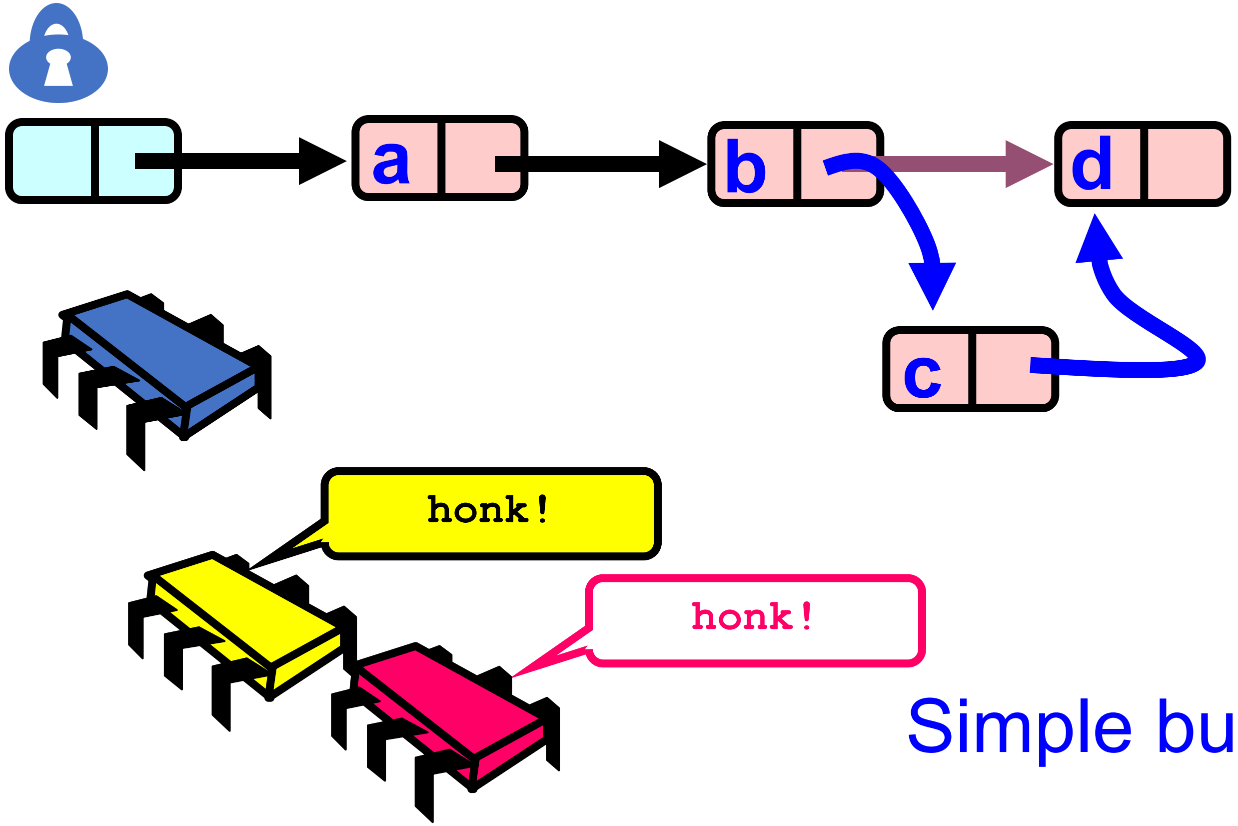
Coarse-Grained Locking



Coarse-Grained Locking



Coarse-Grained Locking



Simple but inefficient!

Schedule

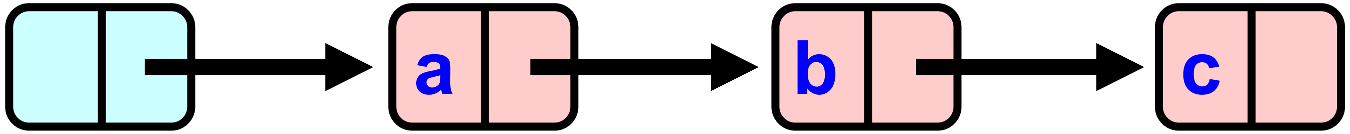
- C++ Atomic Template

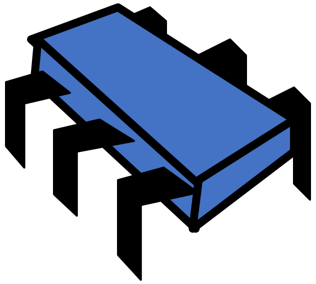
- Concurrent set
    - Coarse-grained lock
    - **fine-grained lock**
    - optimistic locking

Fine-grained Locking

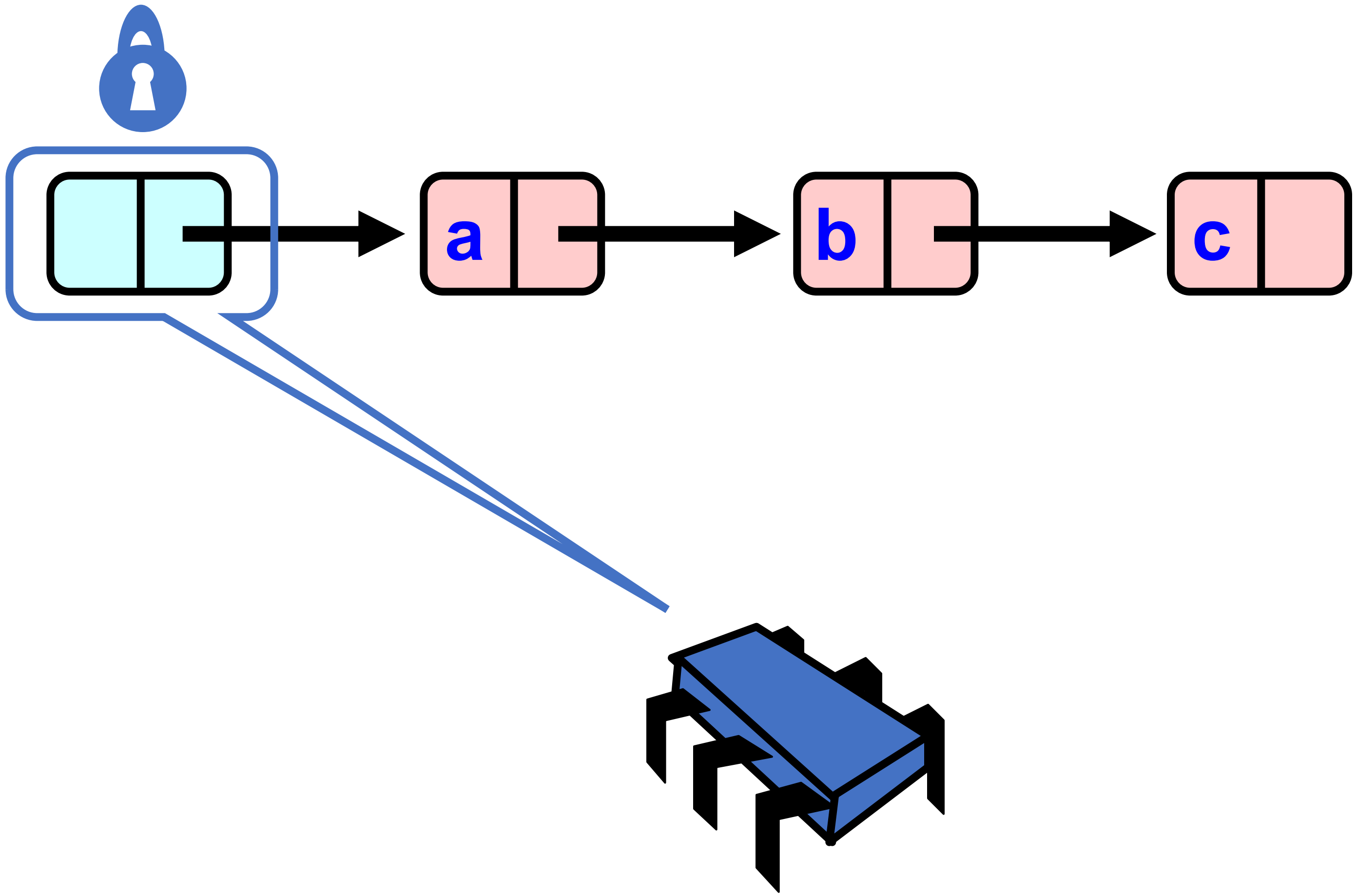
- Requires **careful** thought
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand locking

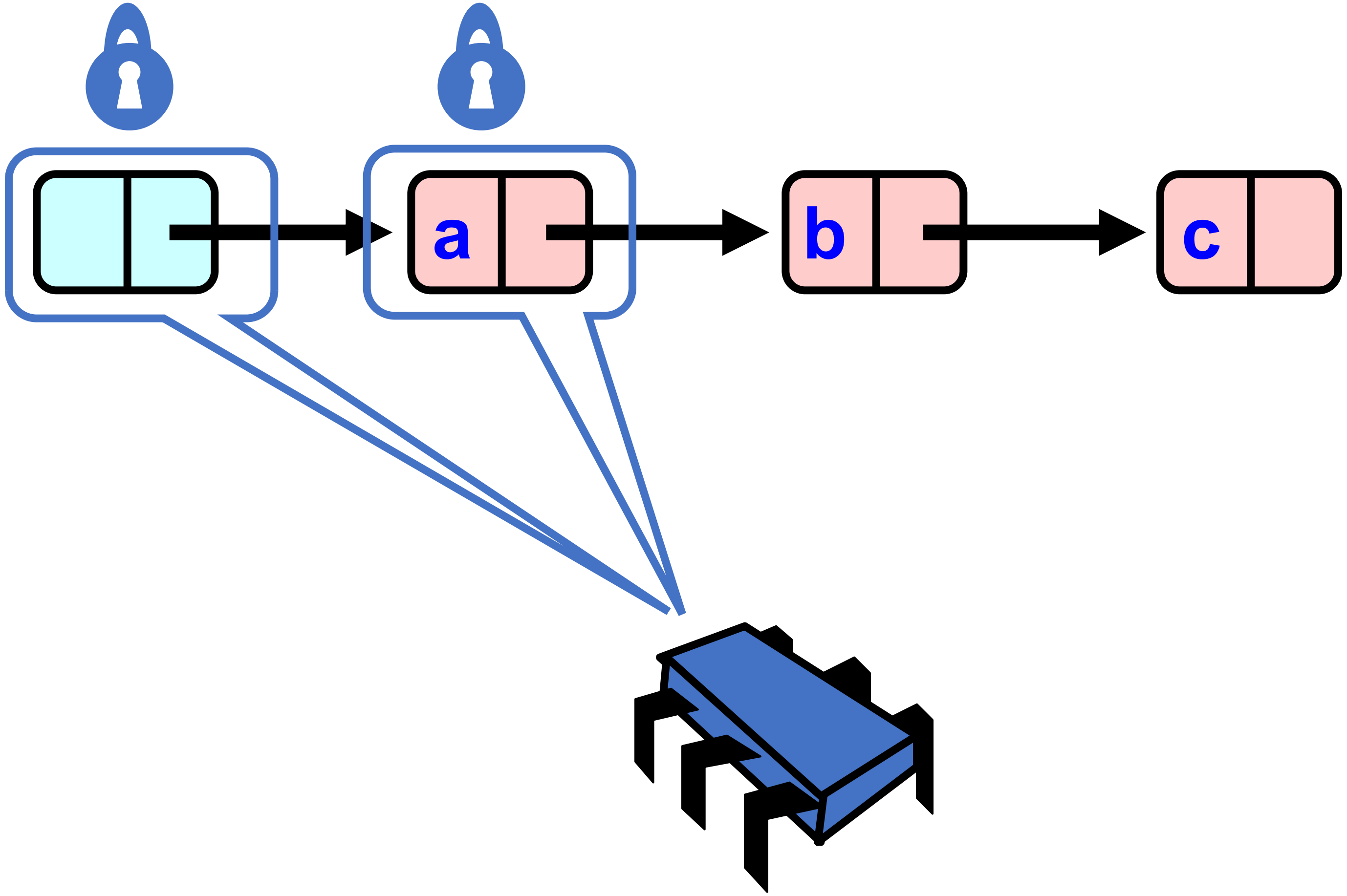




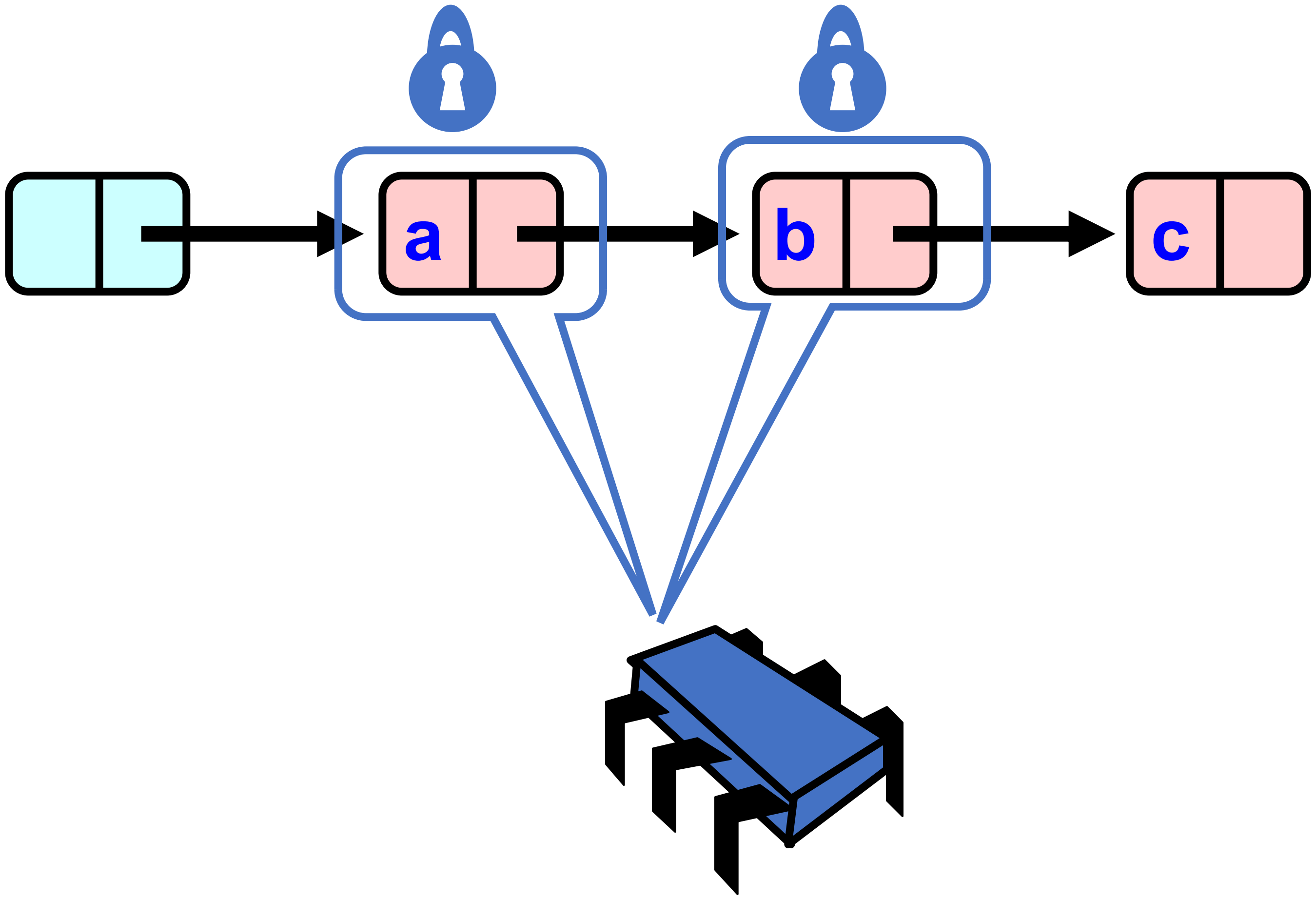
Hand-over-Hand locking



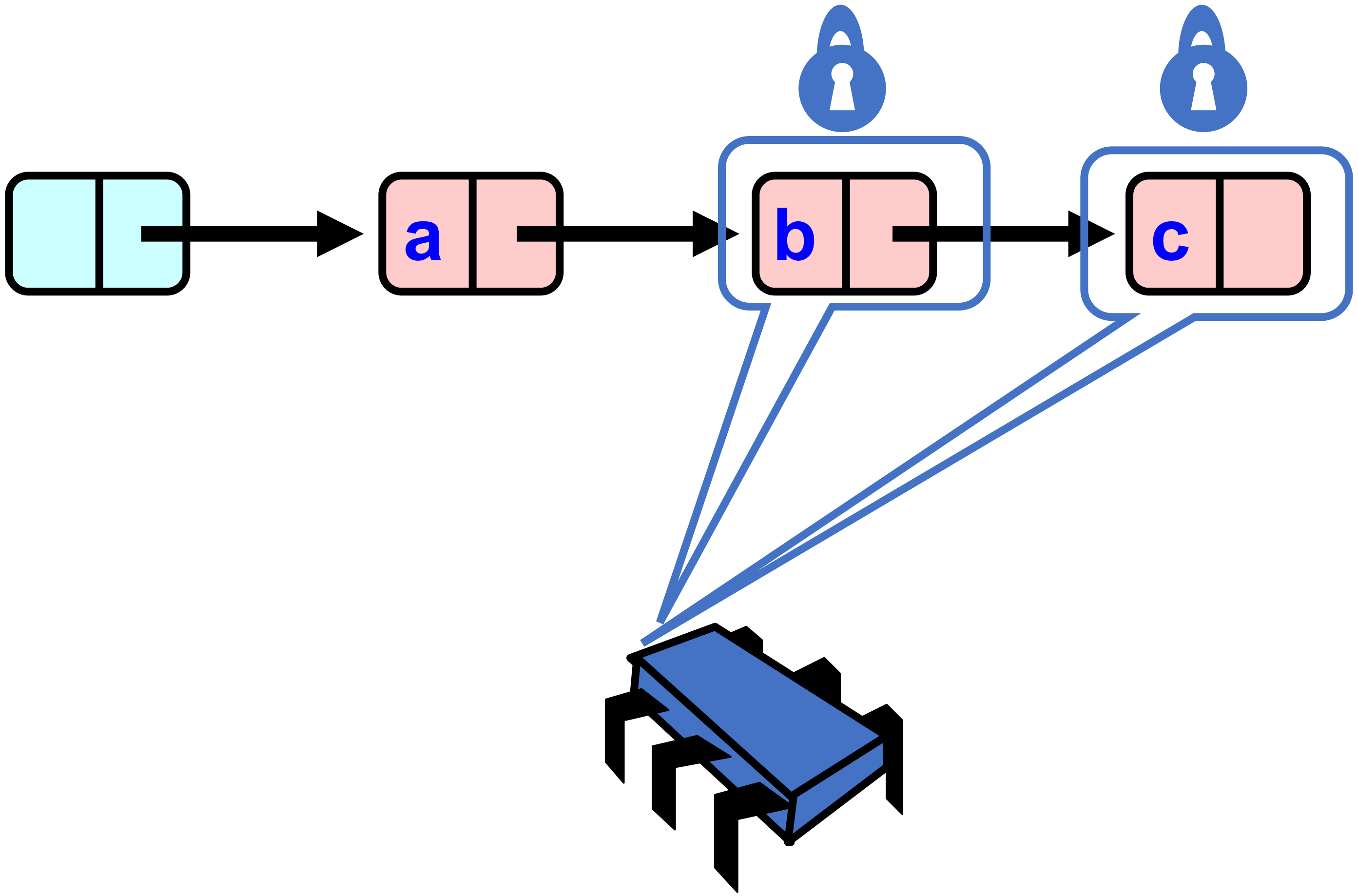
Hand-over-Hand locking



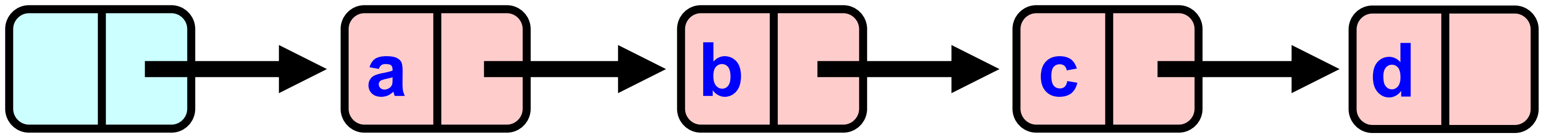
Hand-over-Hand locking

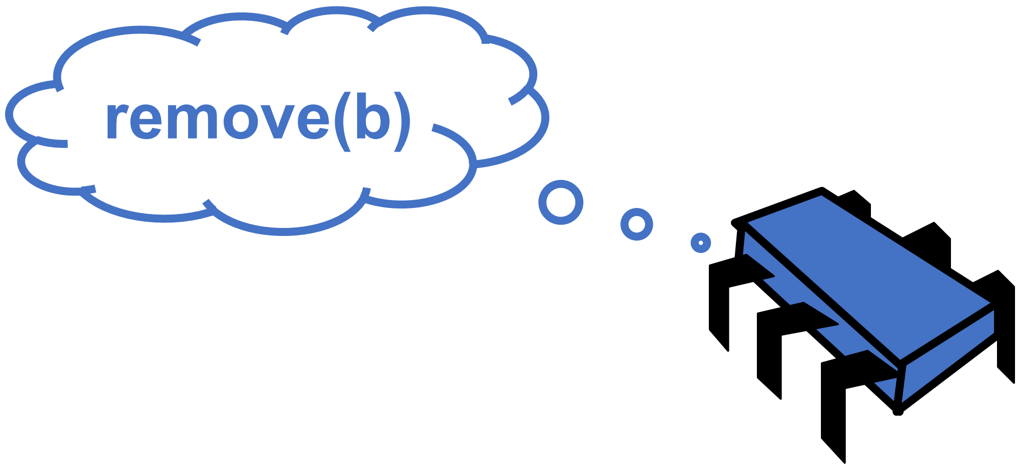


Hand-over-Hand locking

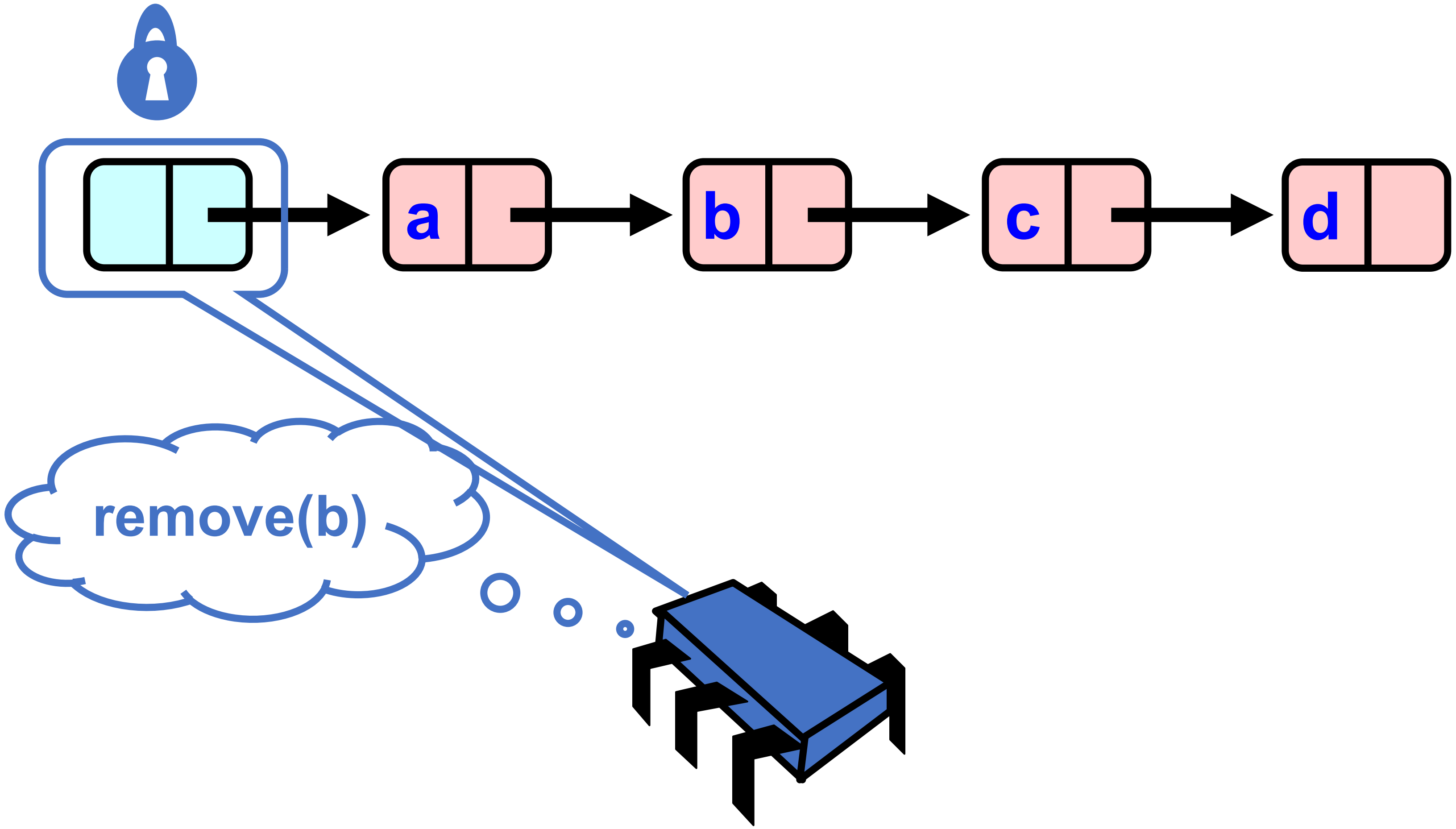


Removing a Node

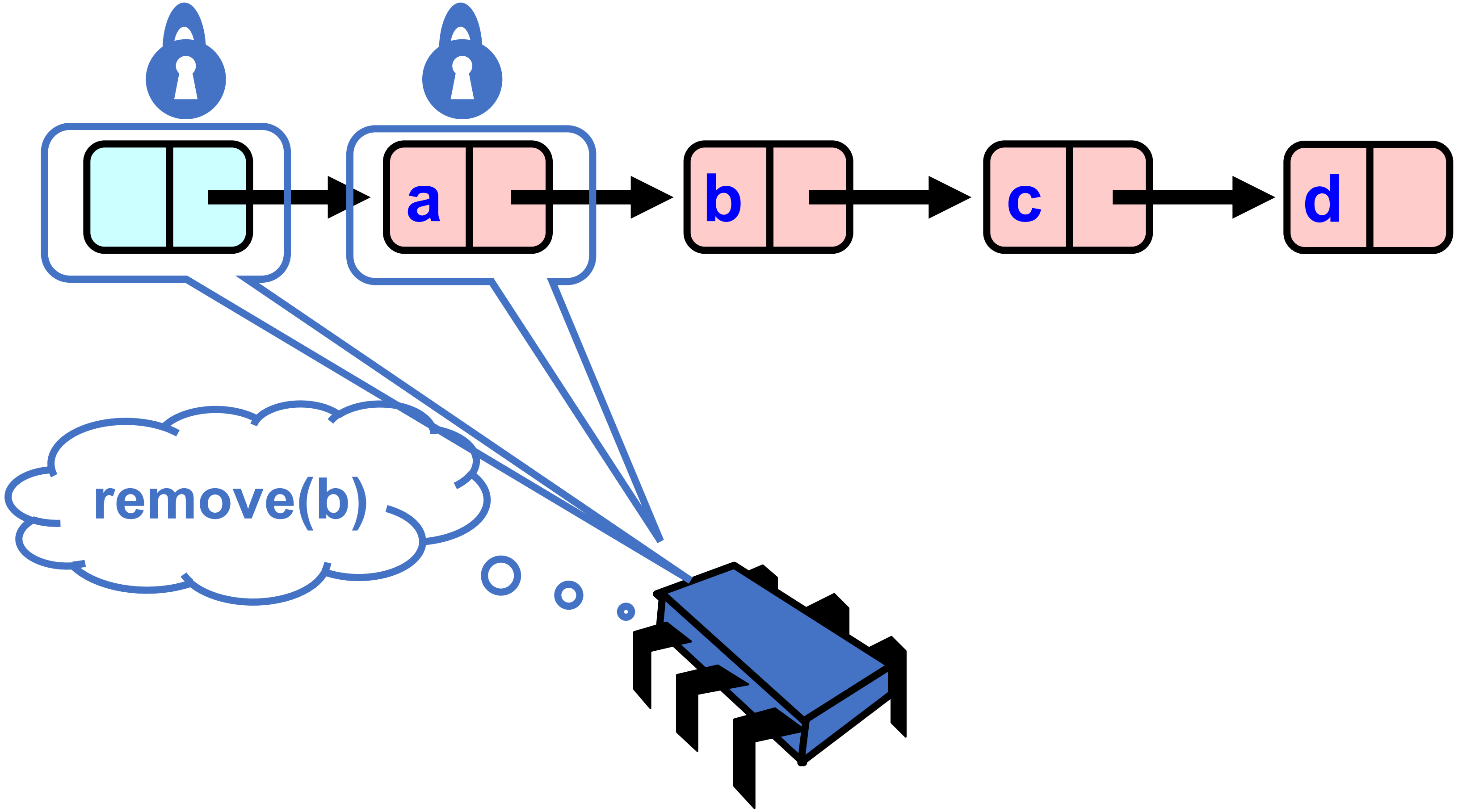




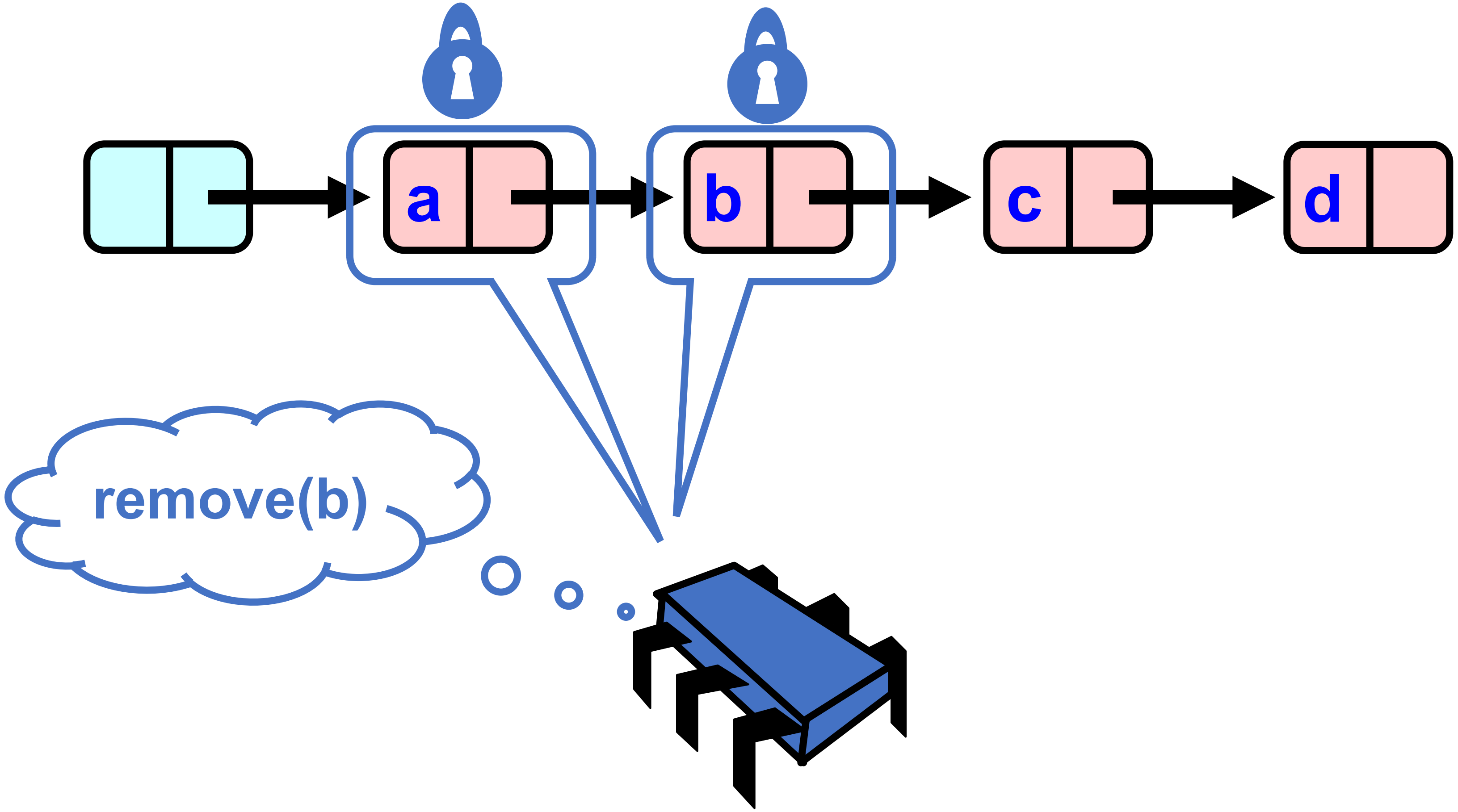
Removing a Node



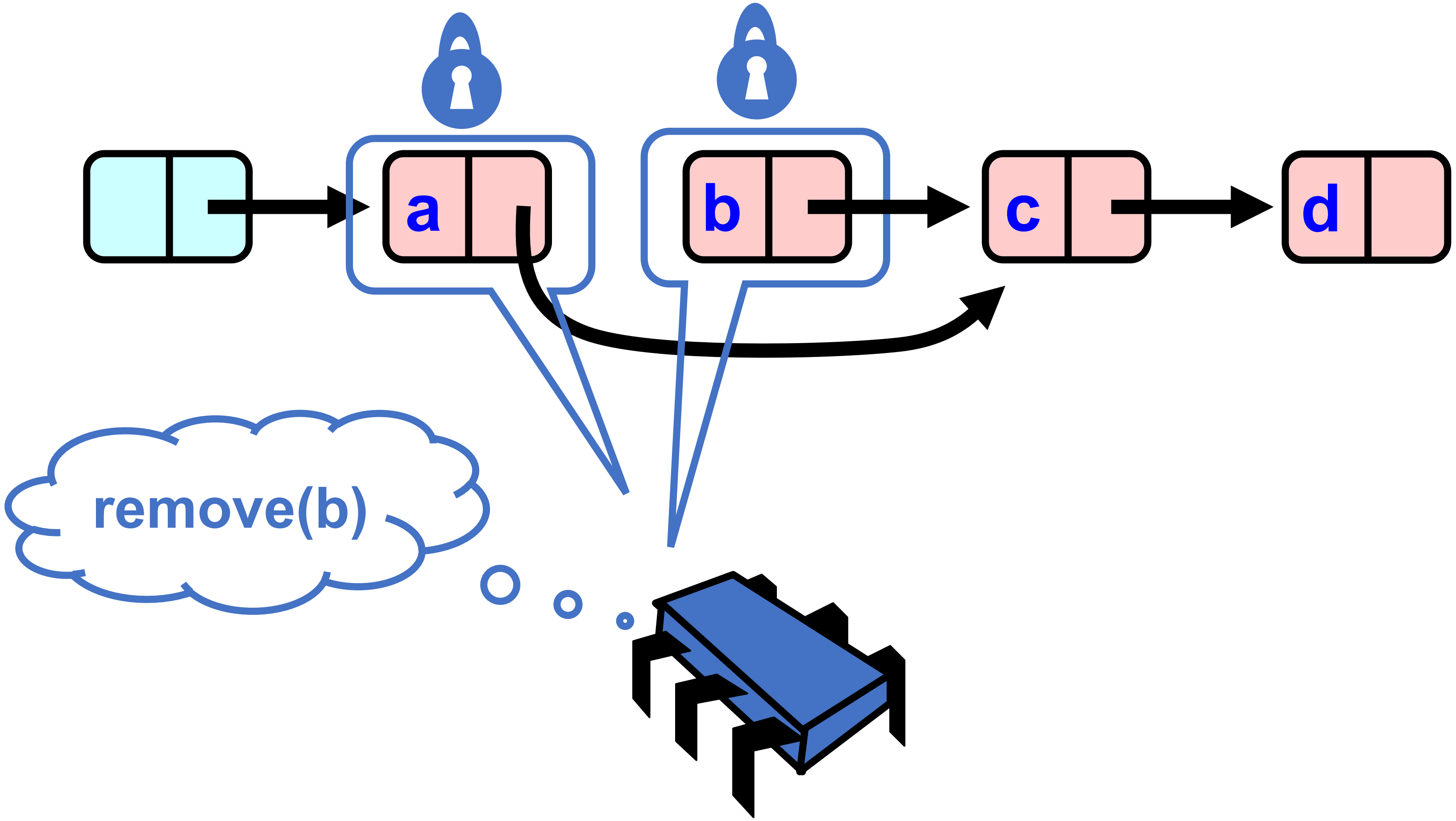
Removing a Node



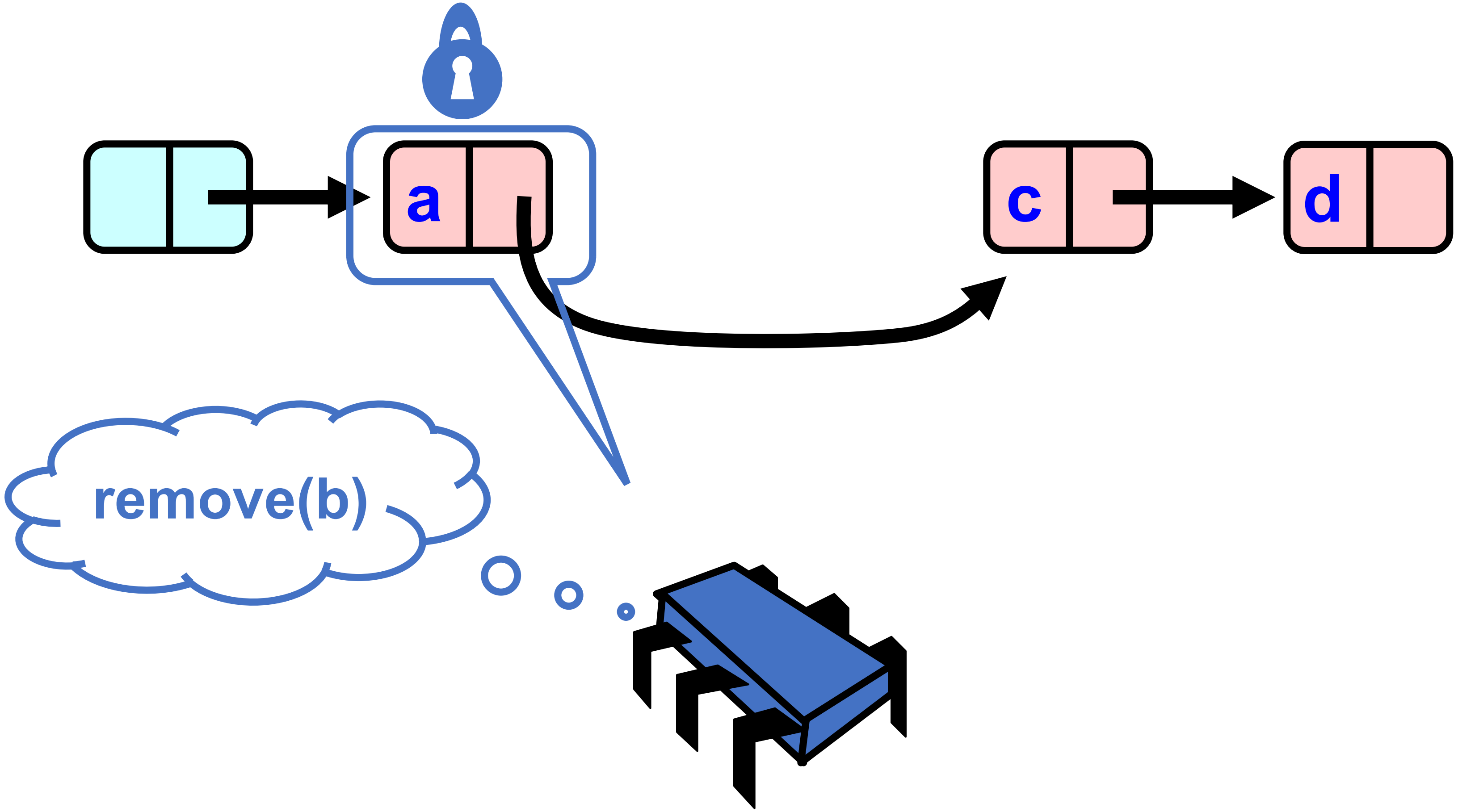
Removing a Node



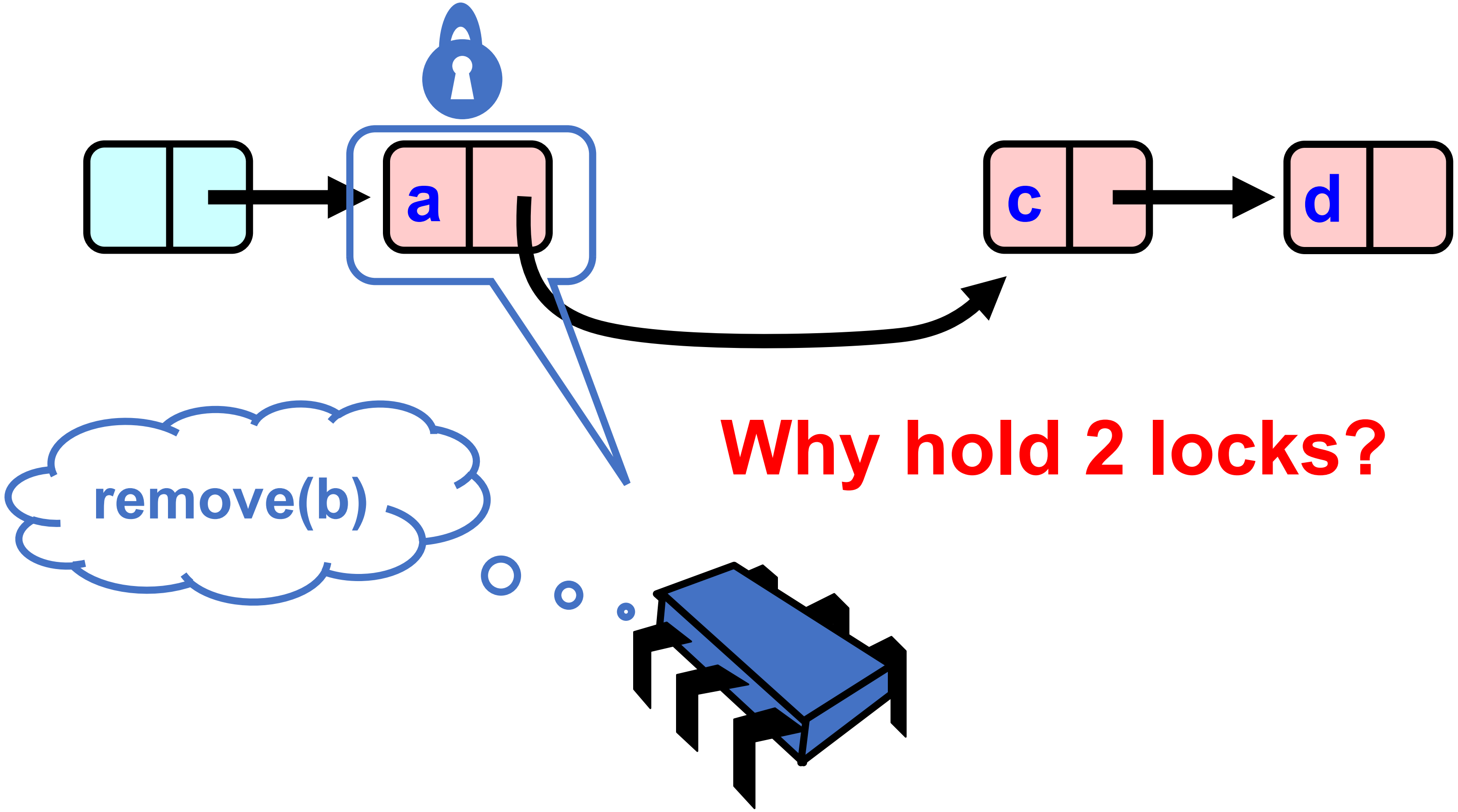
Removing a Node



Removing a Node

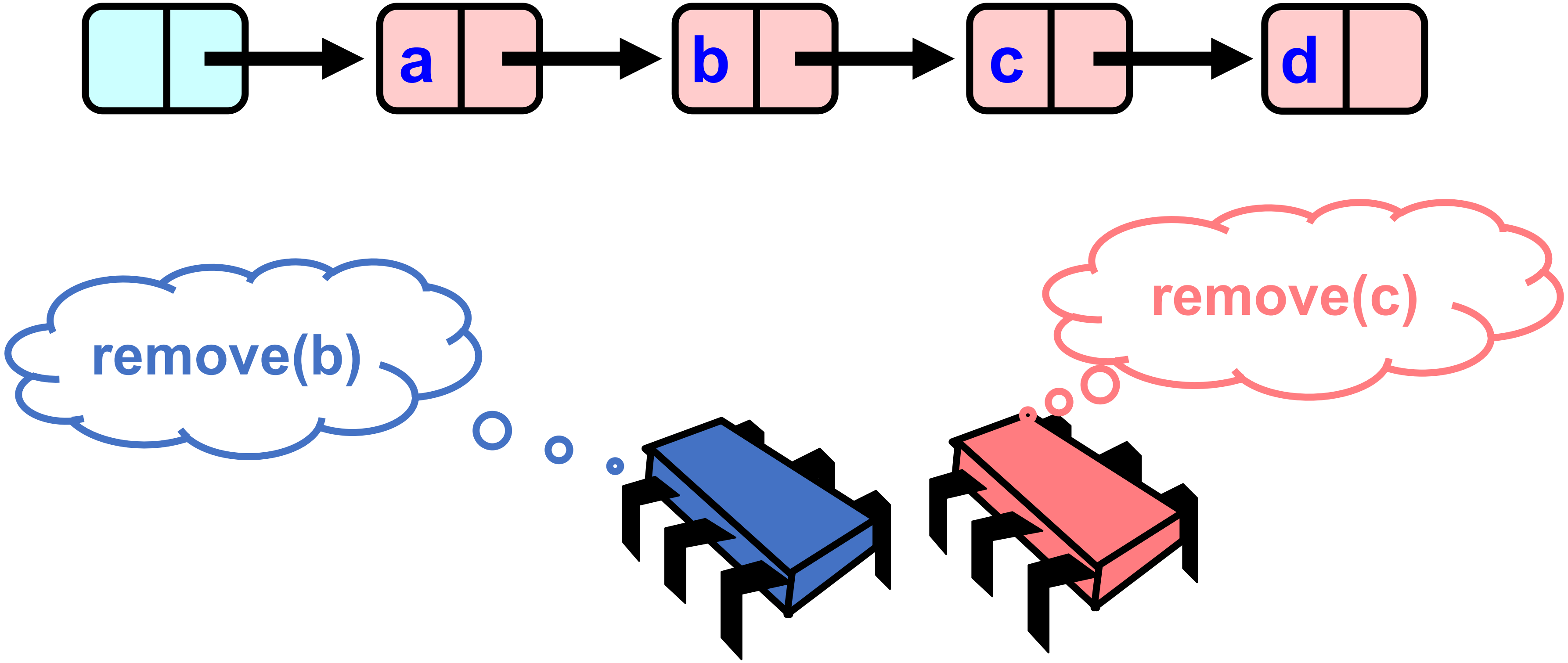


Removing a Node

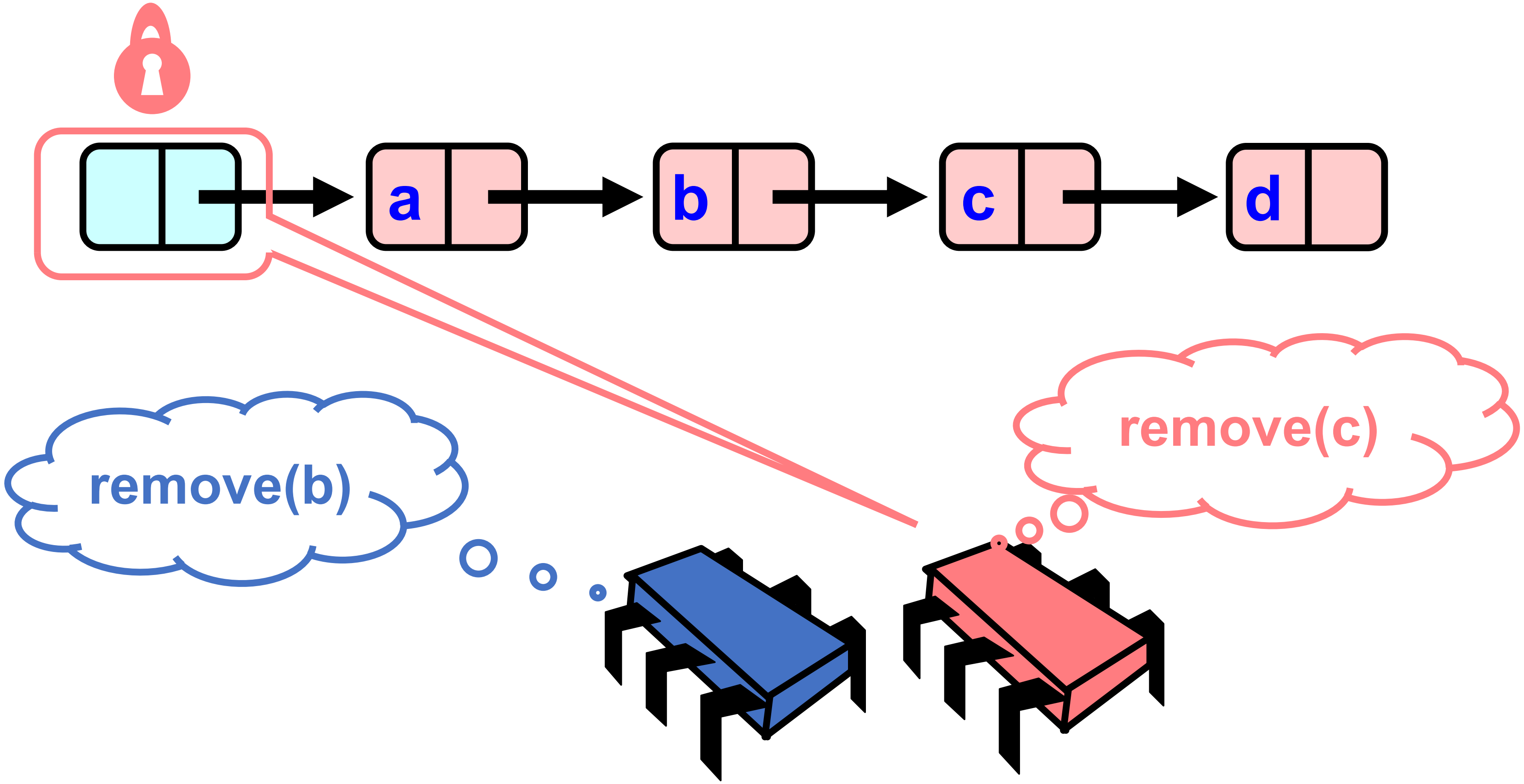


Why hold 2 locks?

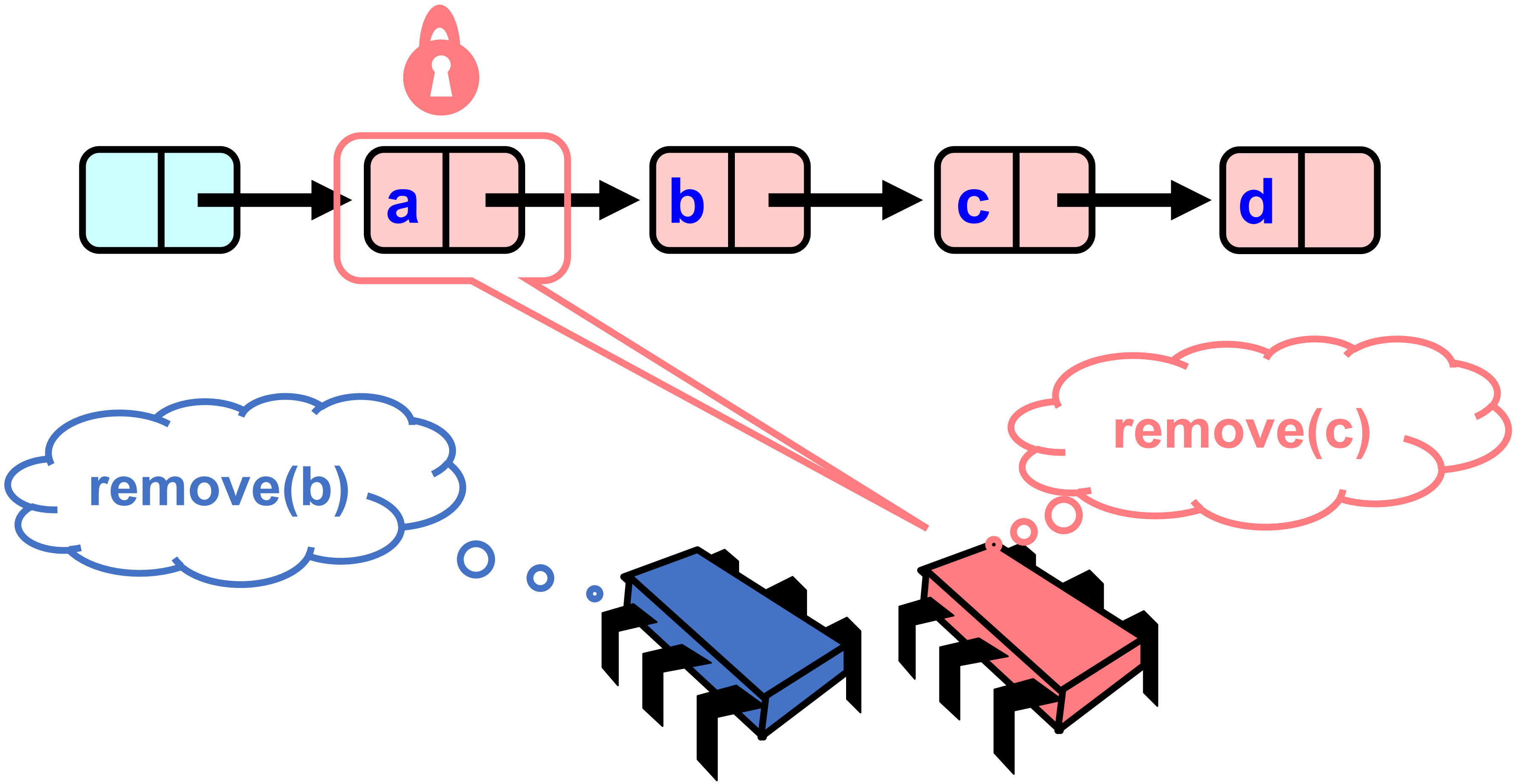
Concurrent Removes



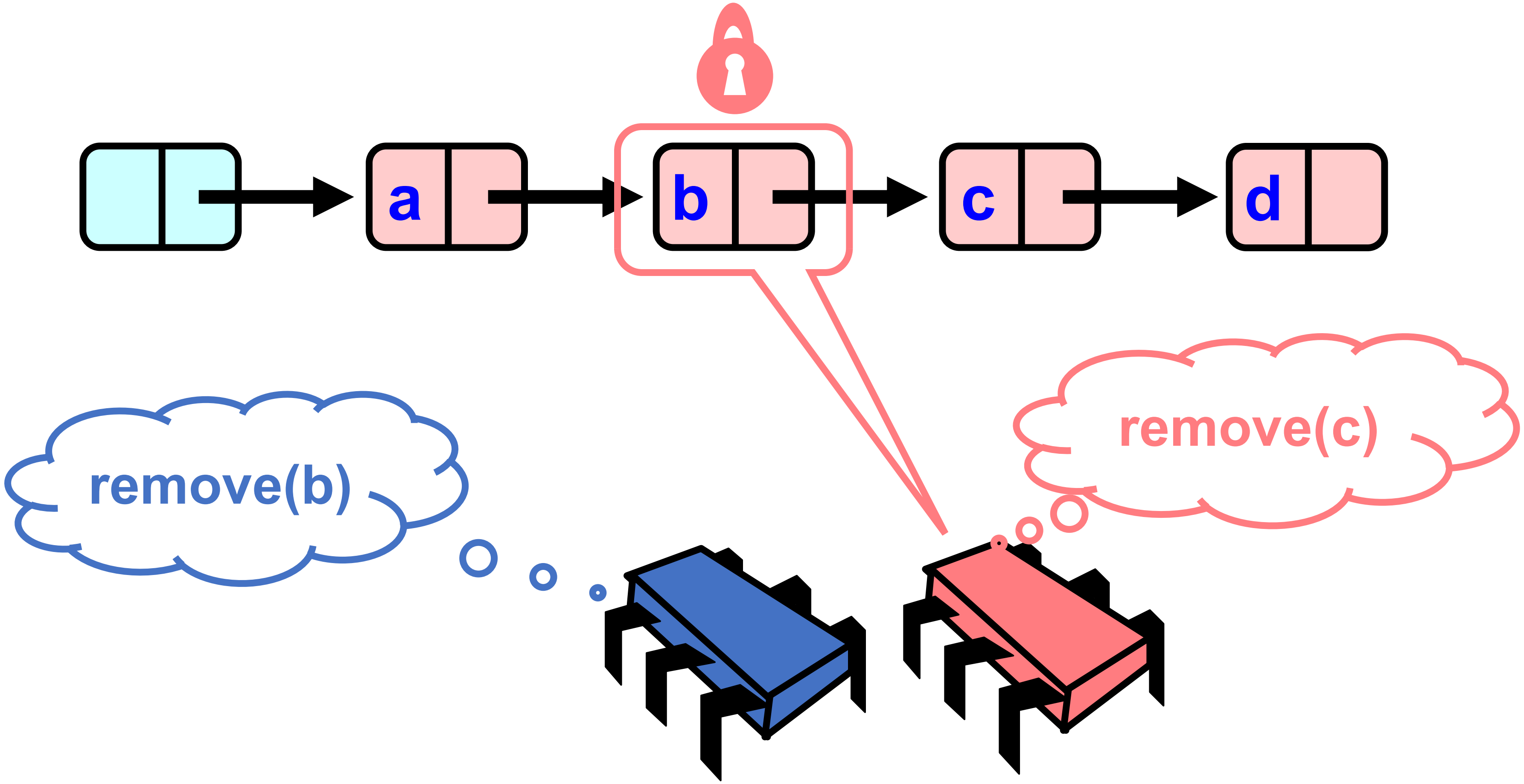
Concurrent Removes



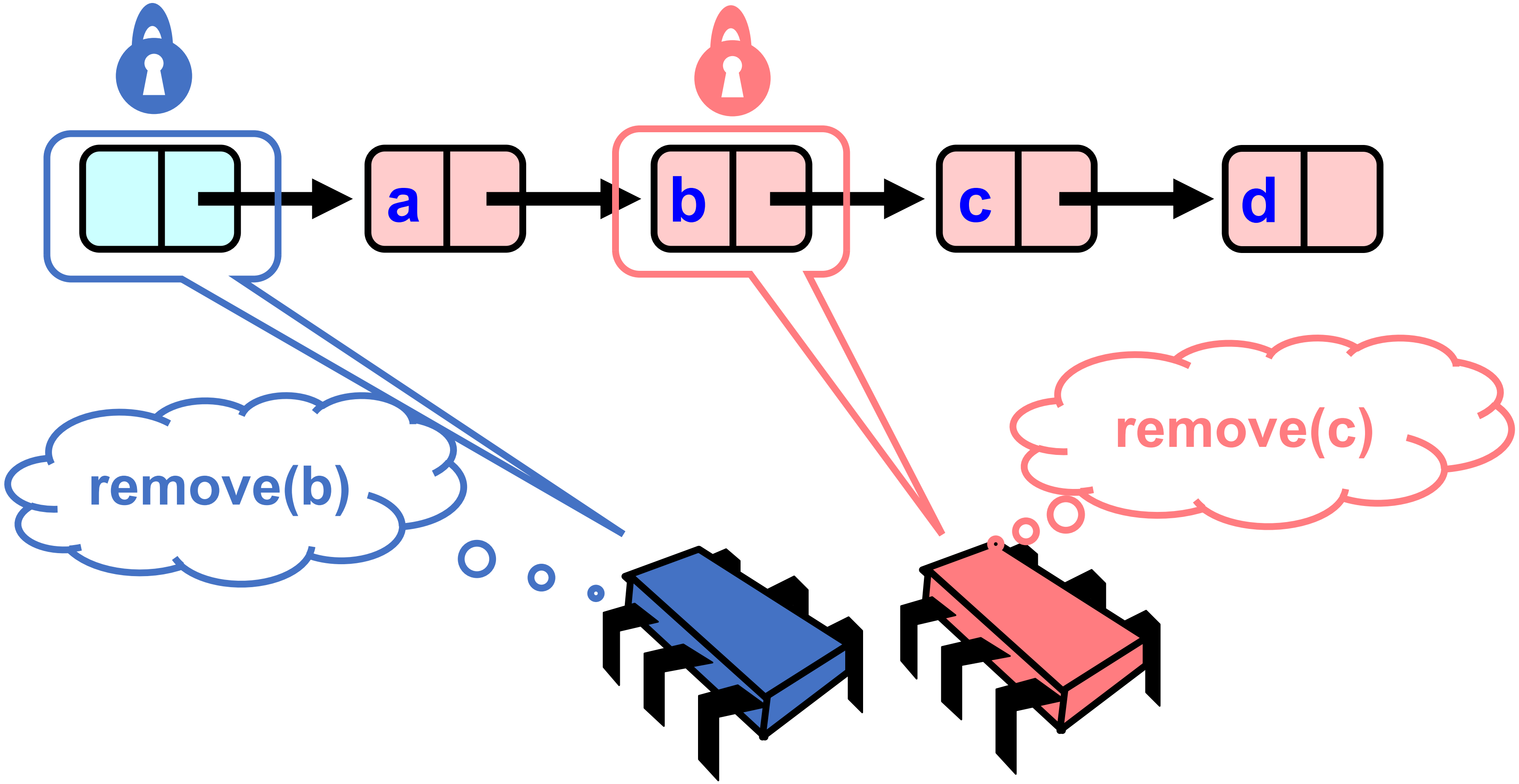
Concurrent Removes



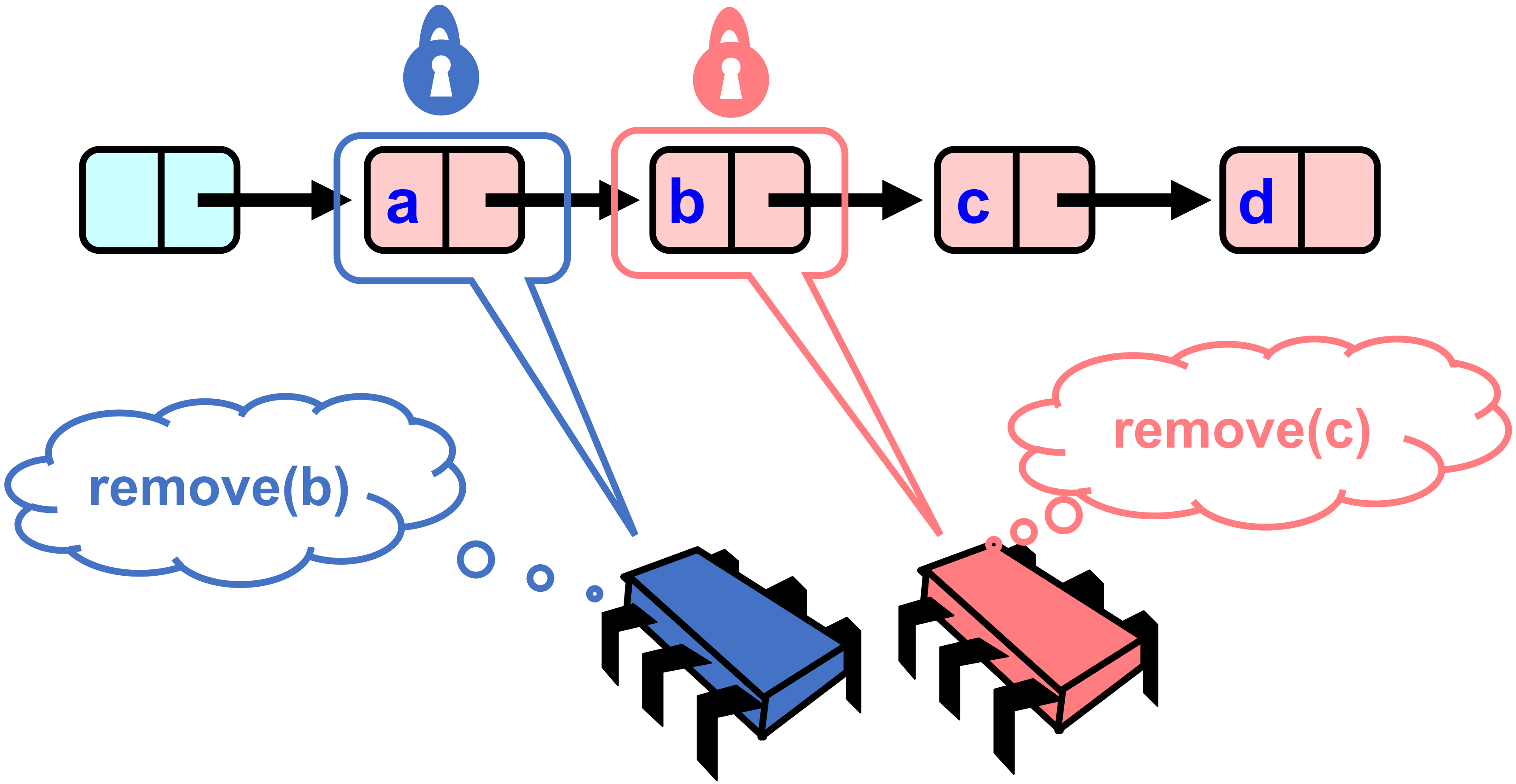
Concurrent Removes



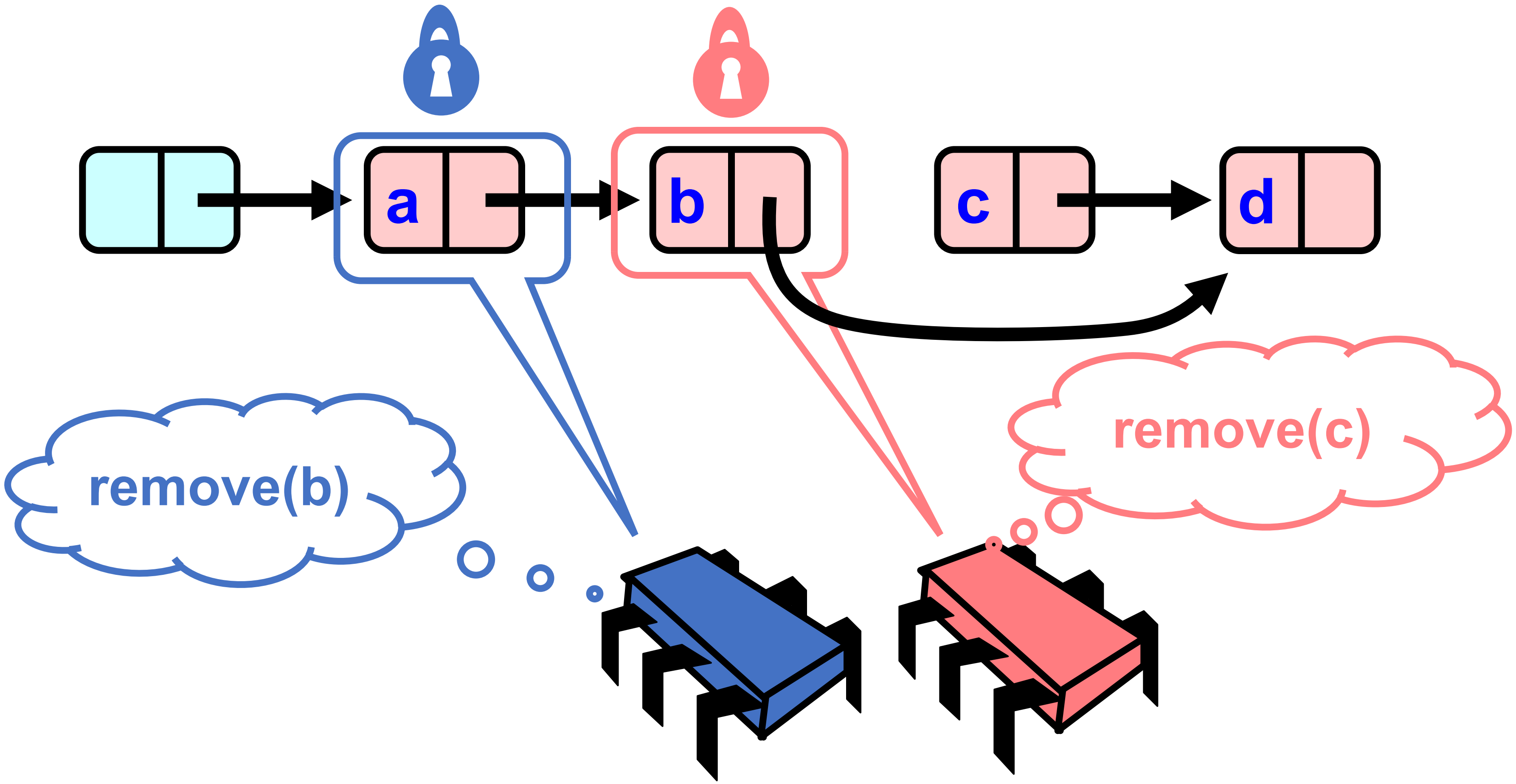
Concurrent Removes



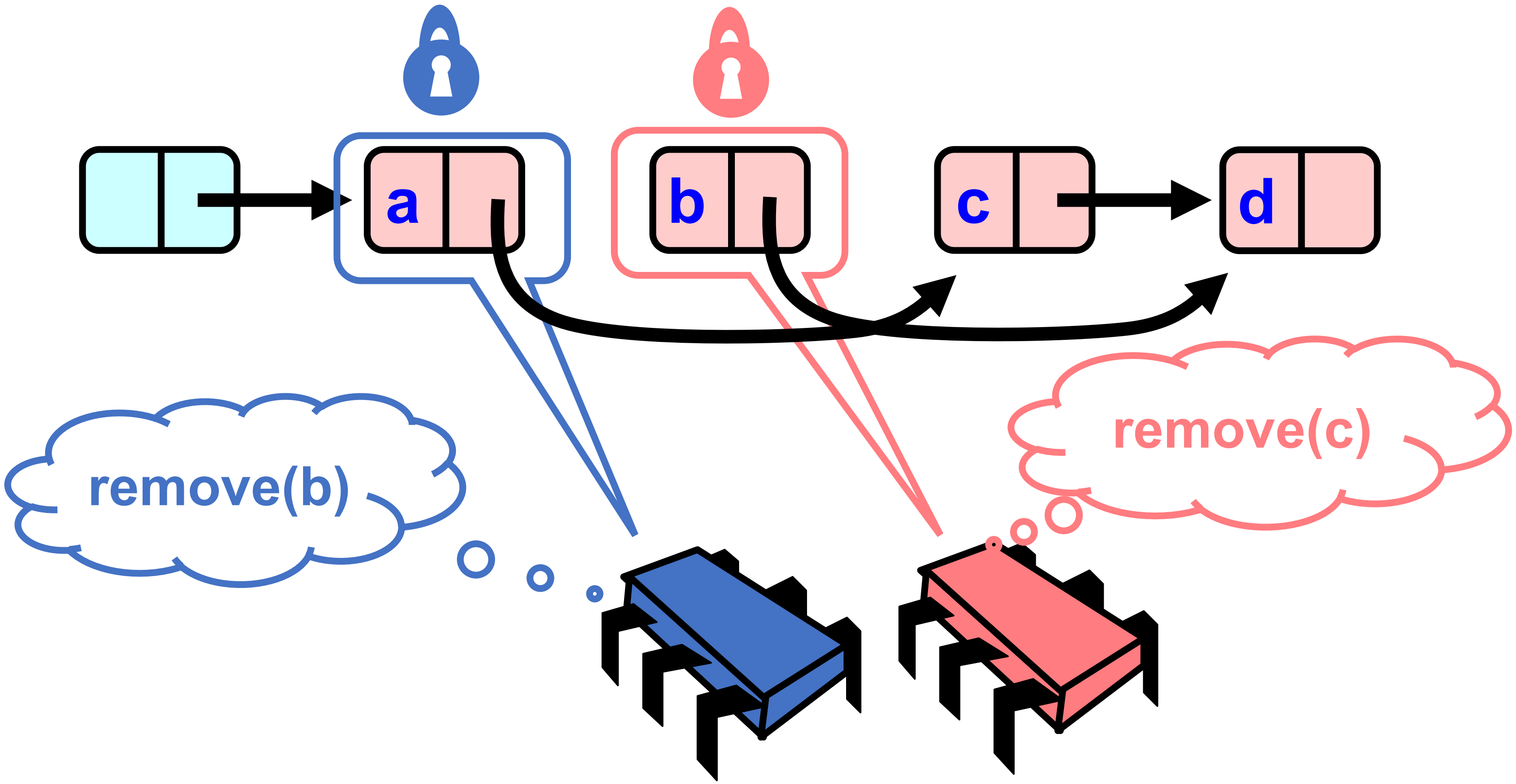
Concurrent Removes



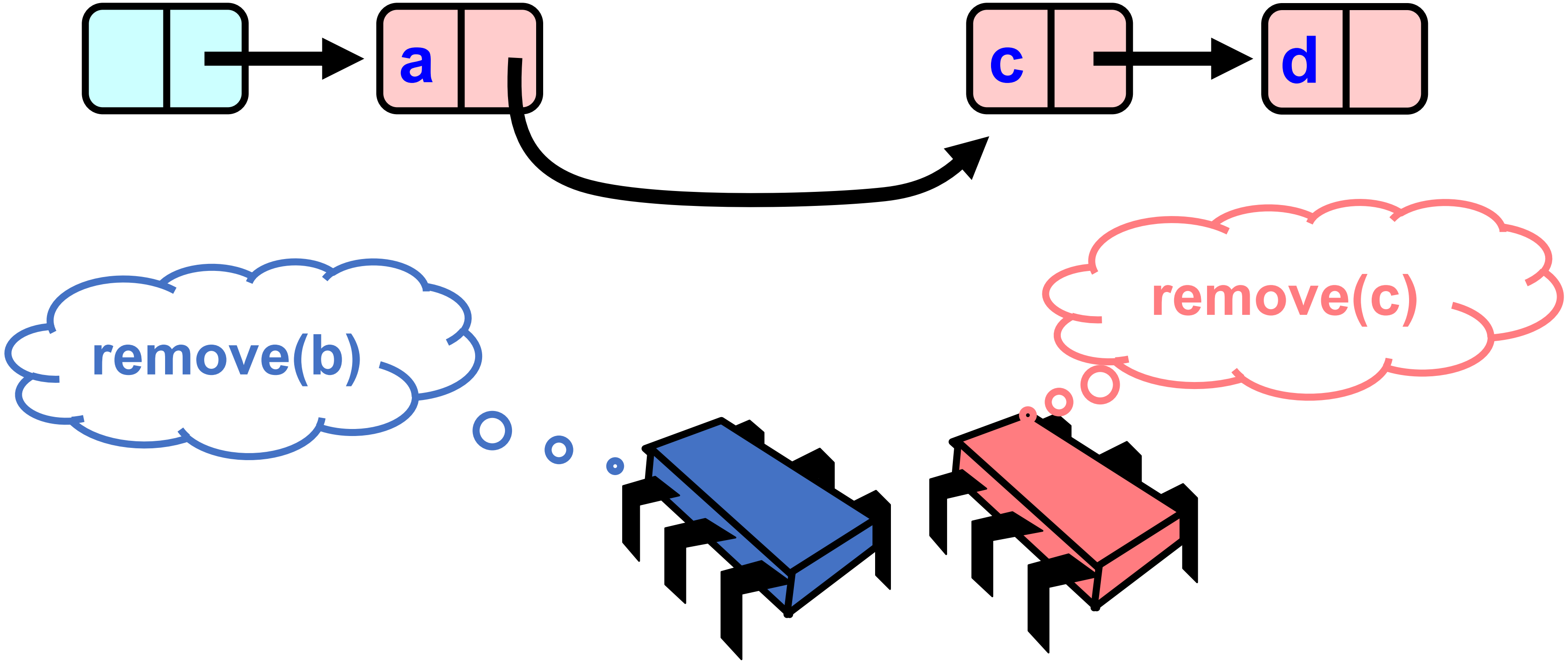
Concurrent Removes



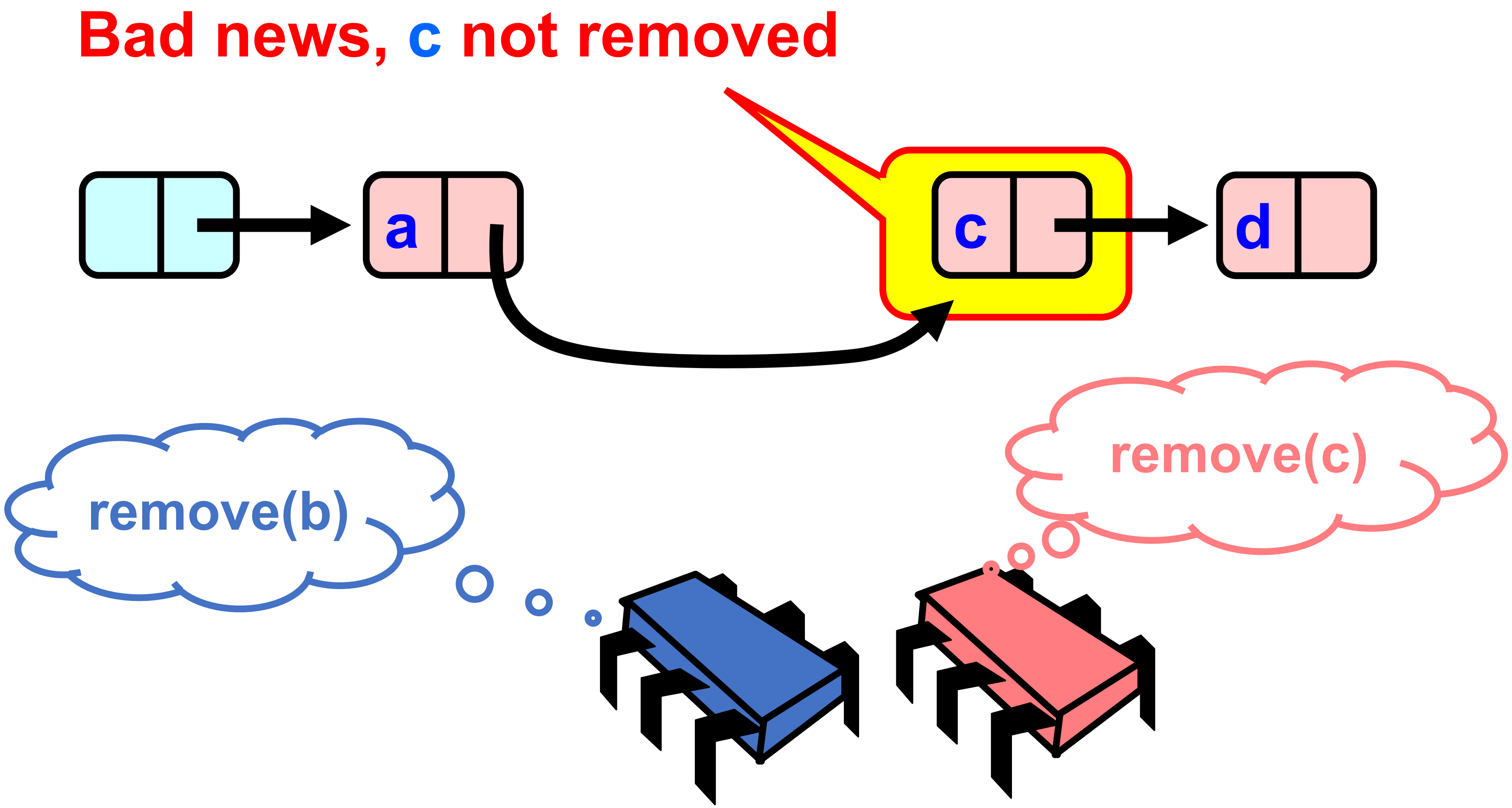
Concurrent Removes



Uh, Oh

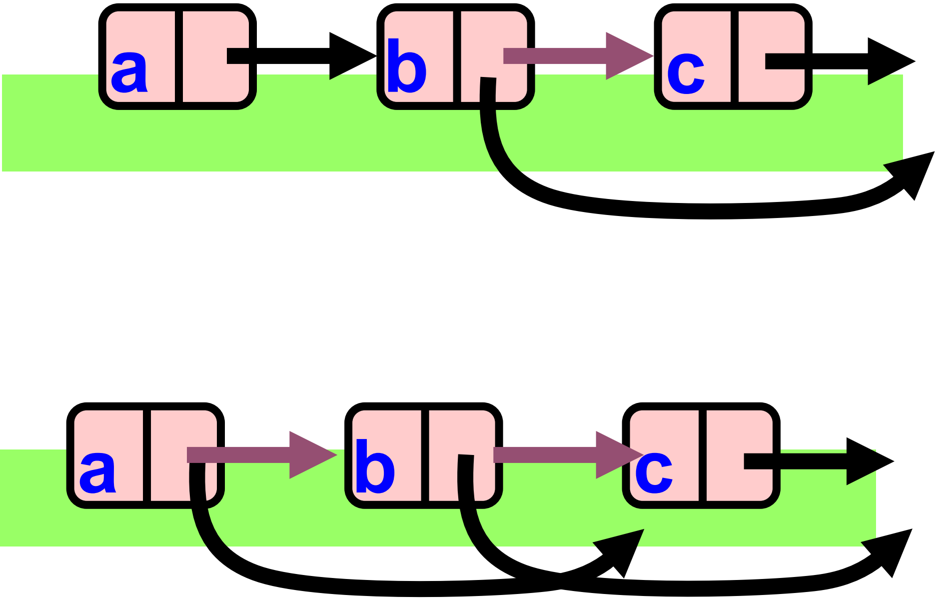


Uh, Oh



Problem

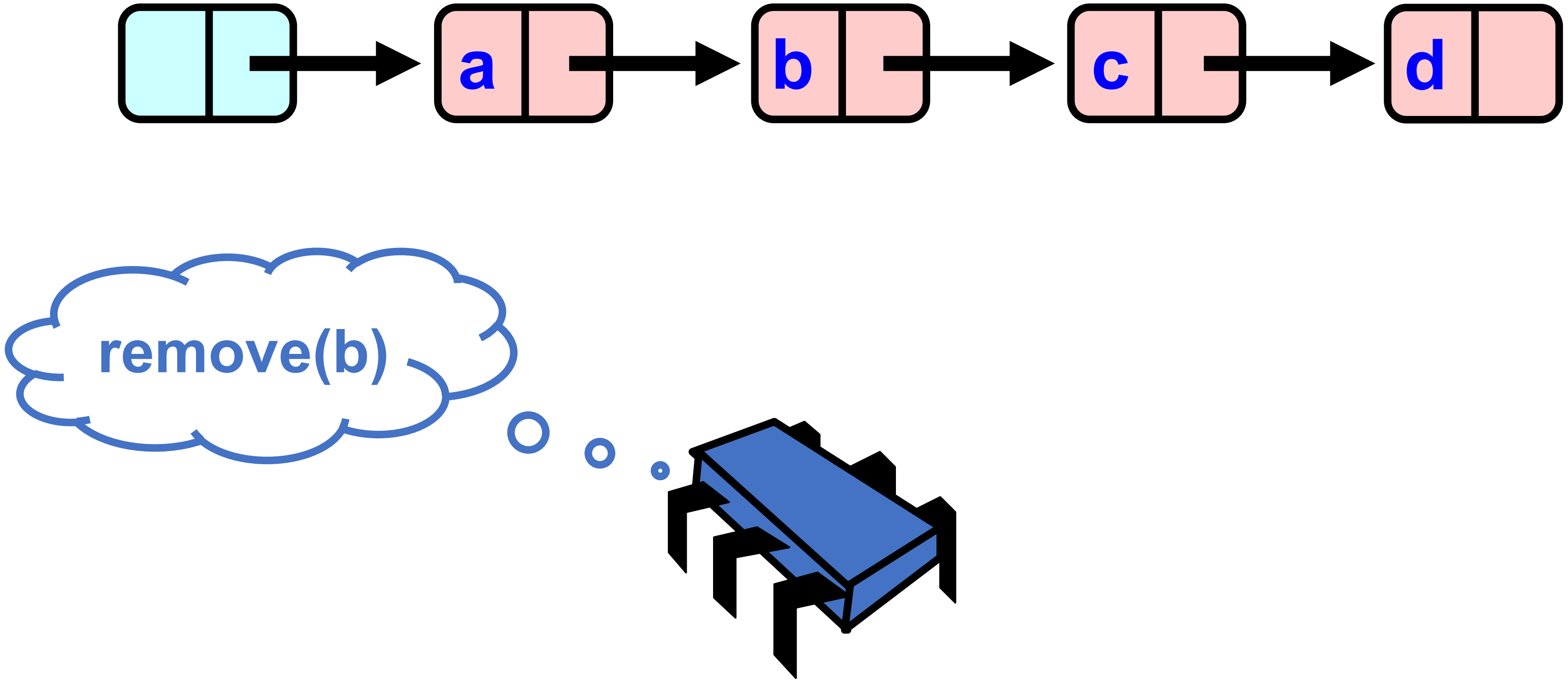
- To delete node c
  - Swing node b's next field to d
- Problem is,
  - ***Data conflict:***
  - Someone deleting b concurrently could direct a pointer to C



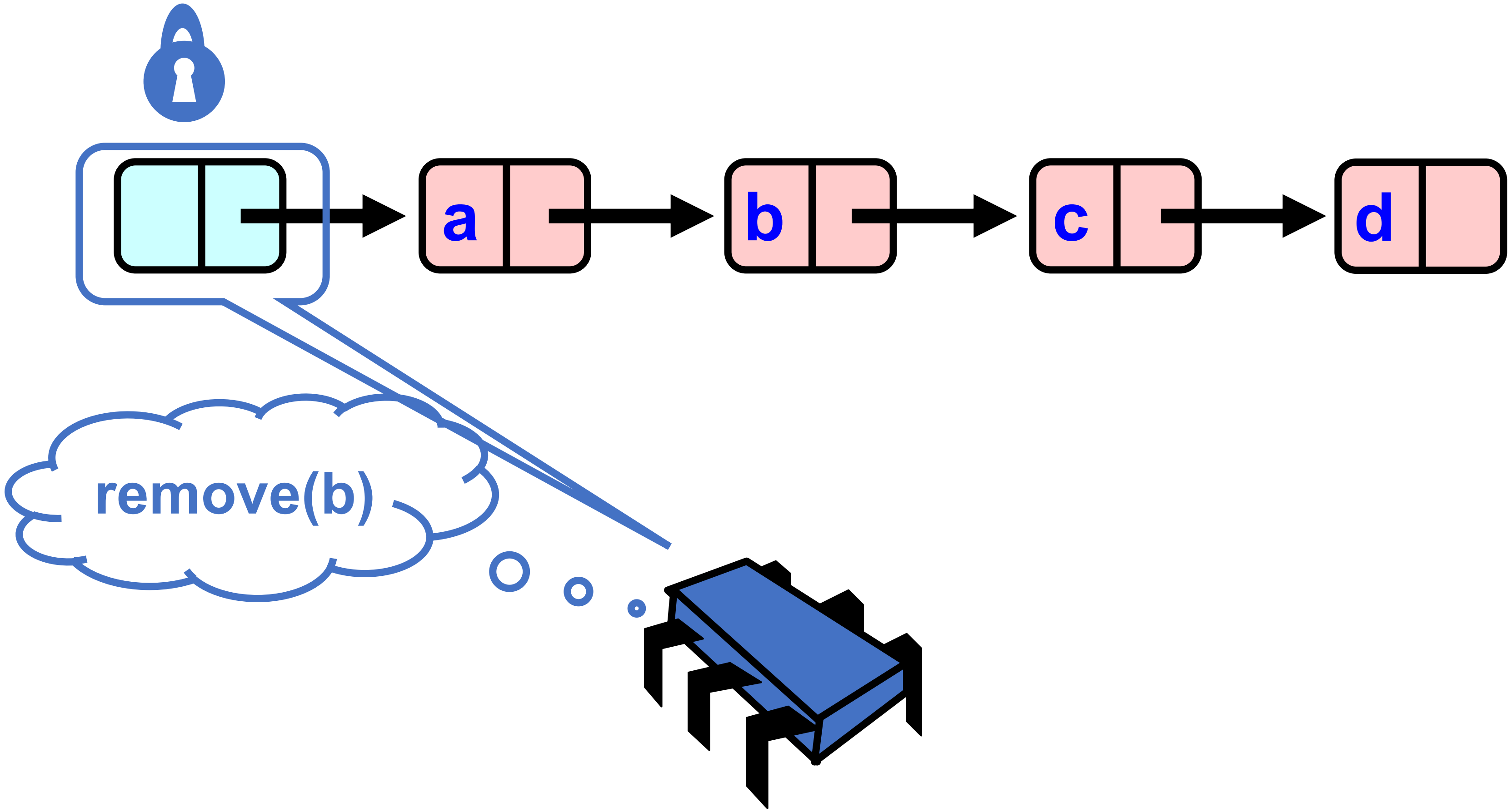
Insight

- 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

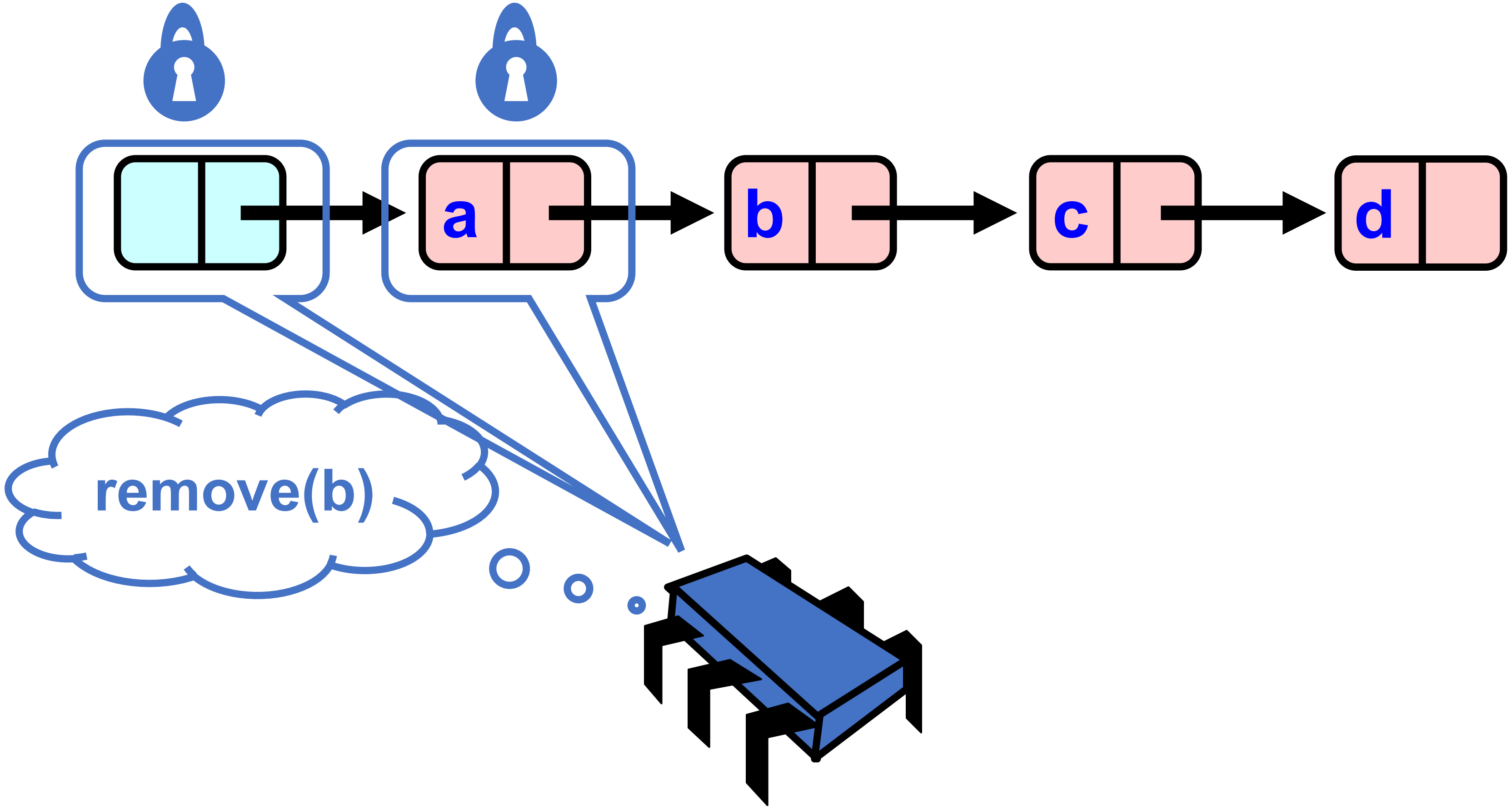
Hand-Over-Hand Again



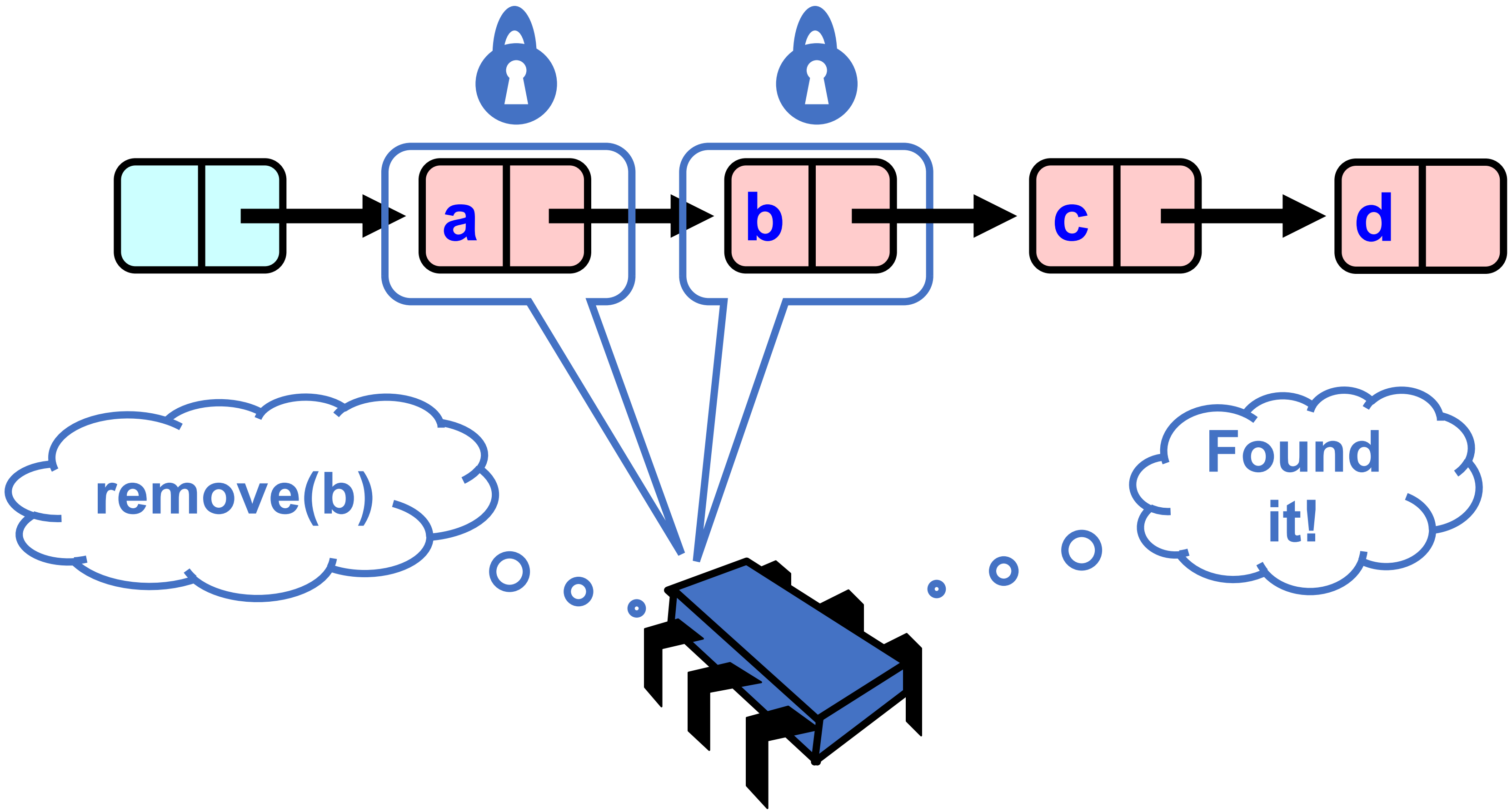
Hand-Over-Hand Again



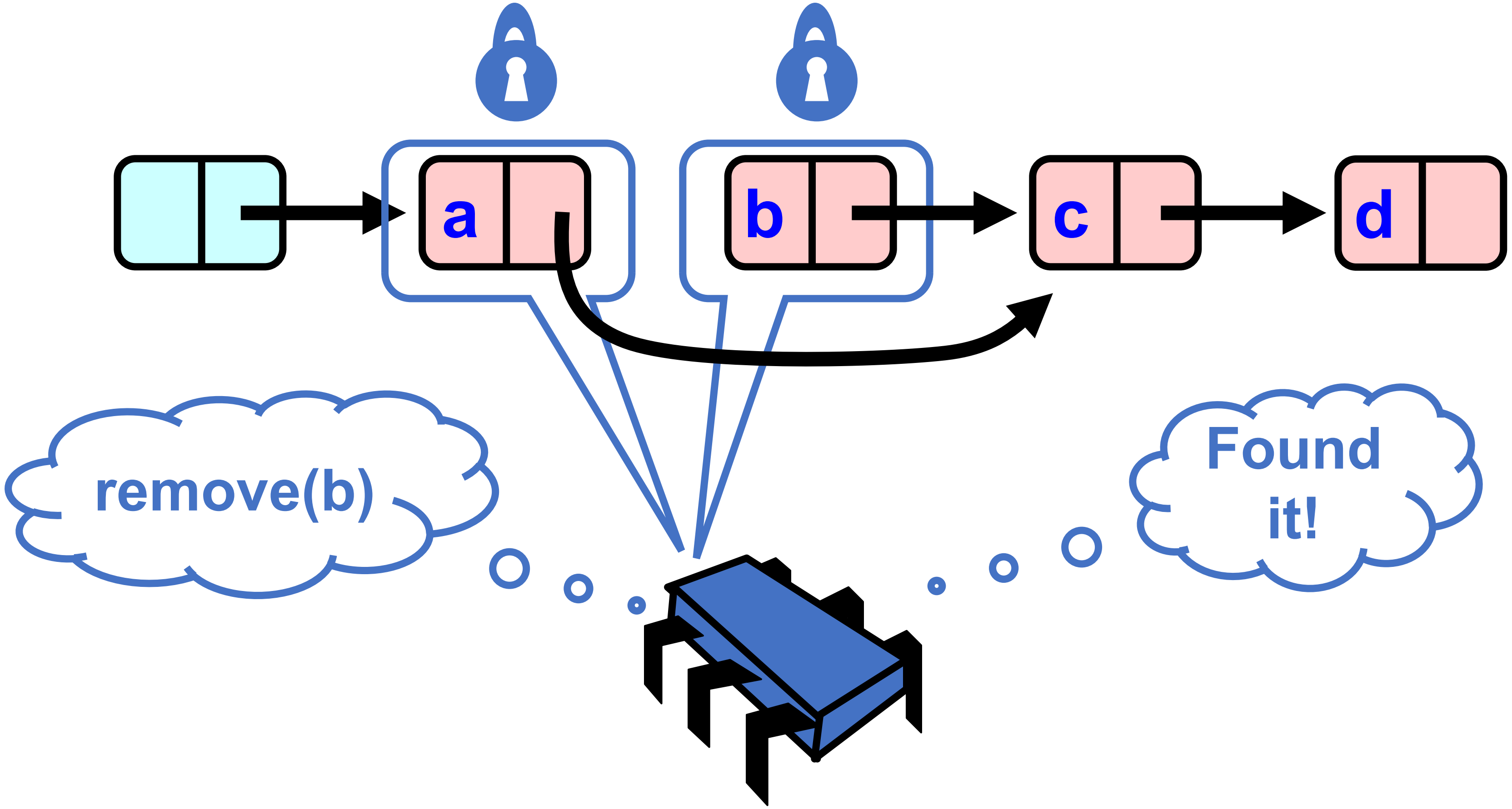
Hand-Over-Hand Again



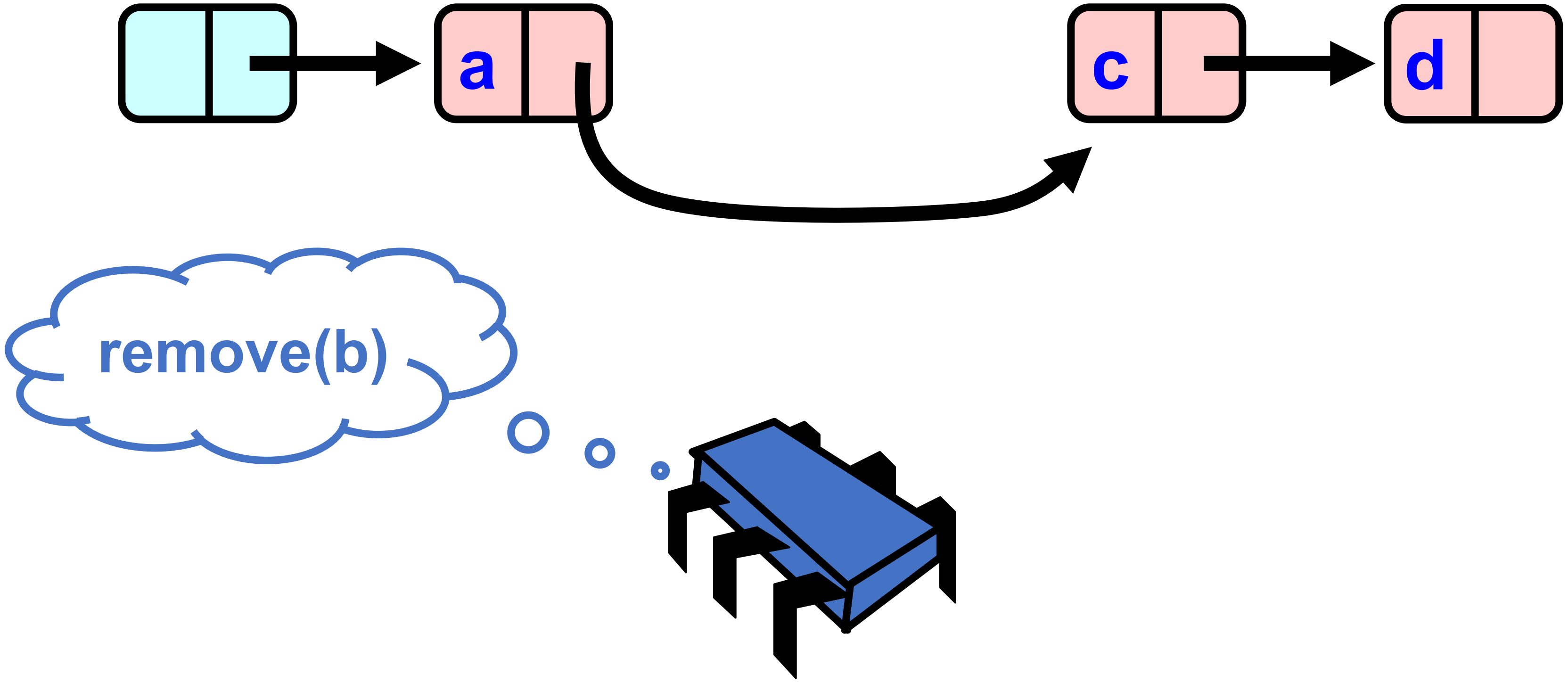
Hand-Over-Hand Again



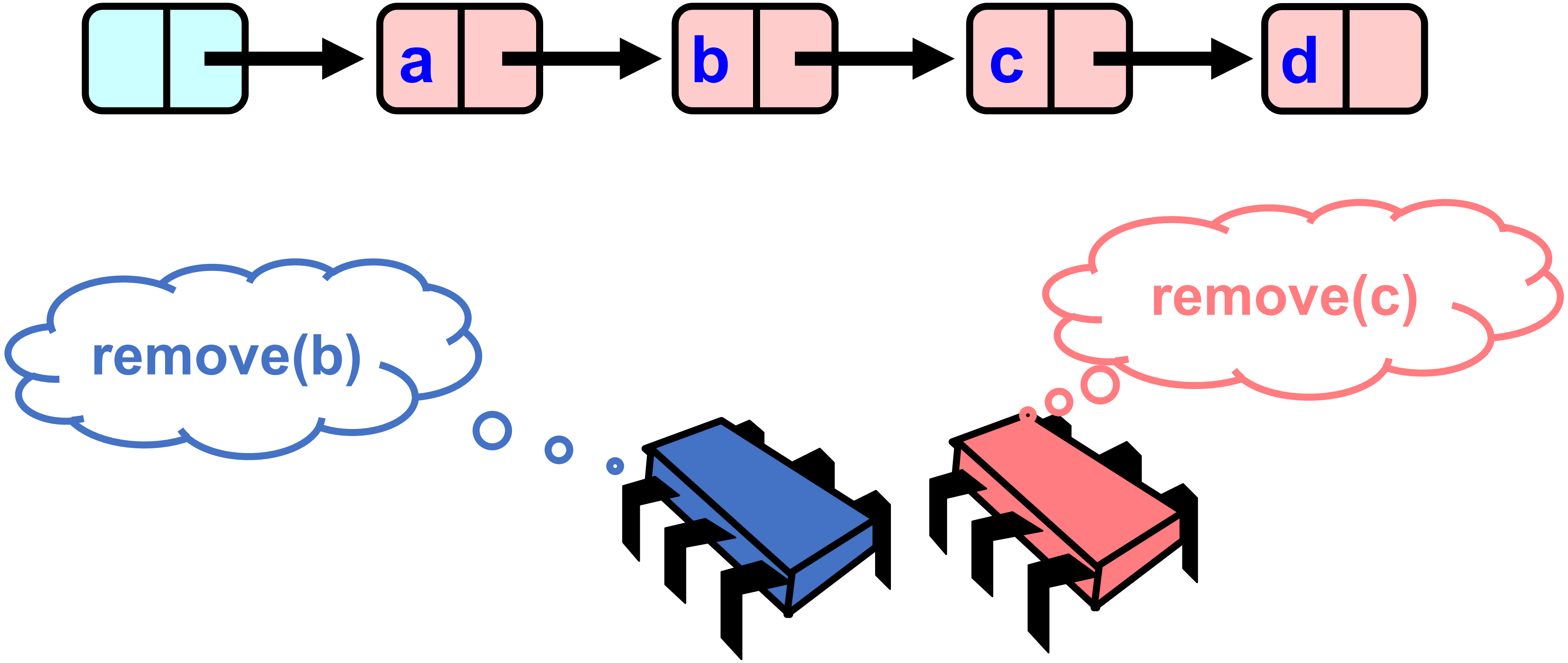
Hand-Over-Hand Again



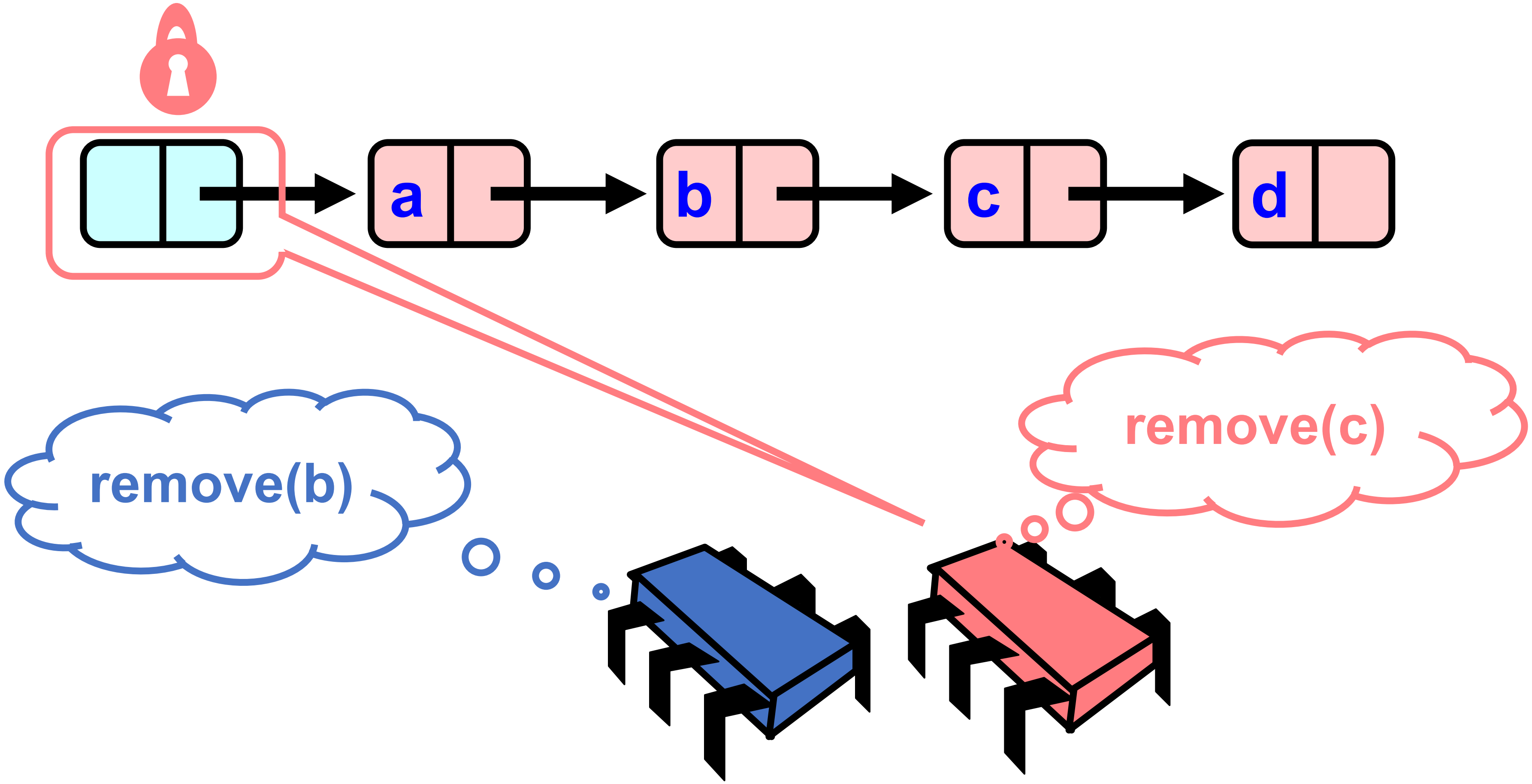
Hand-Over-Hand Again



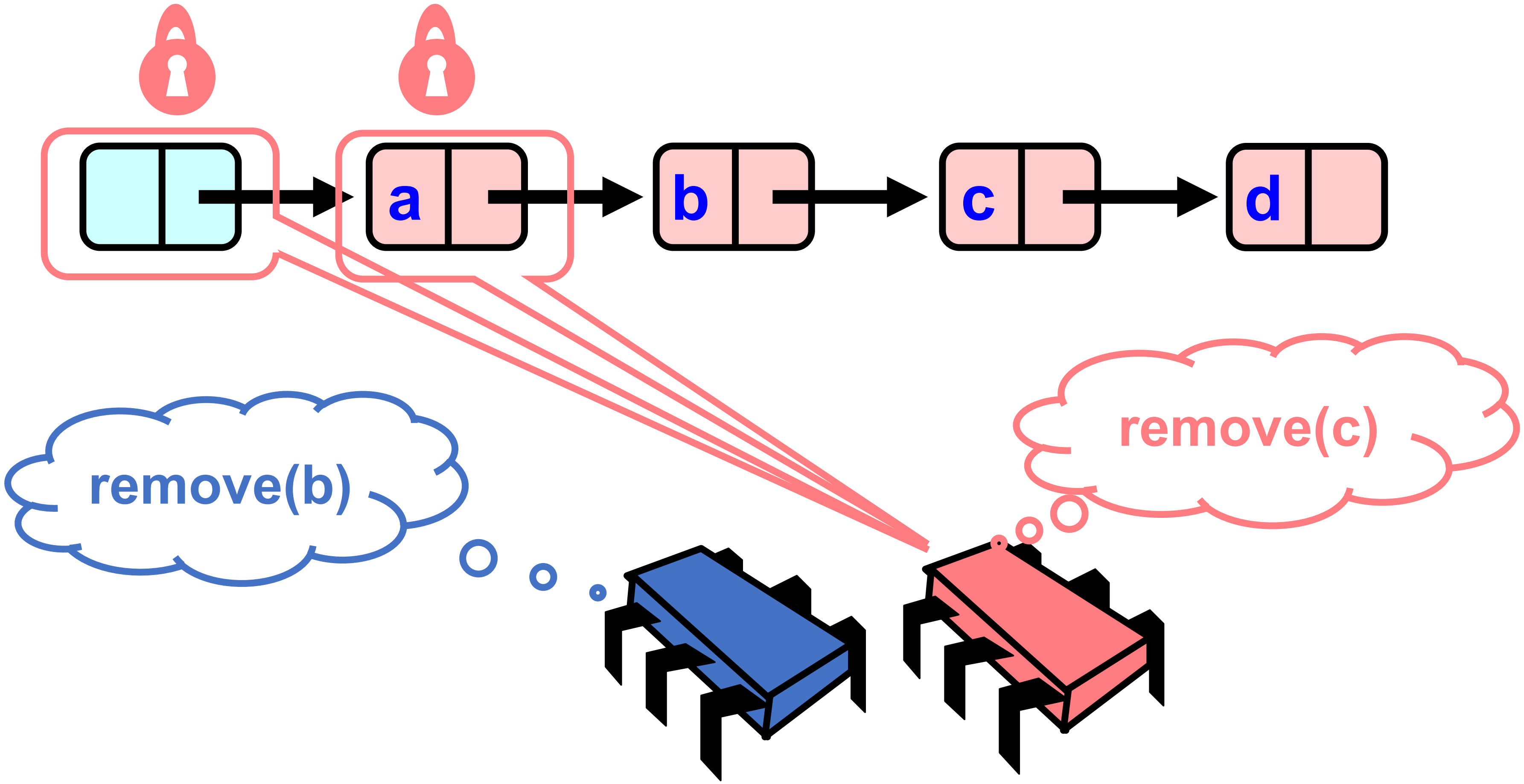
Removing a Node



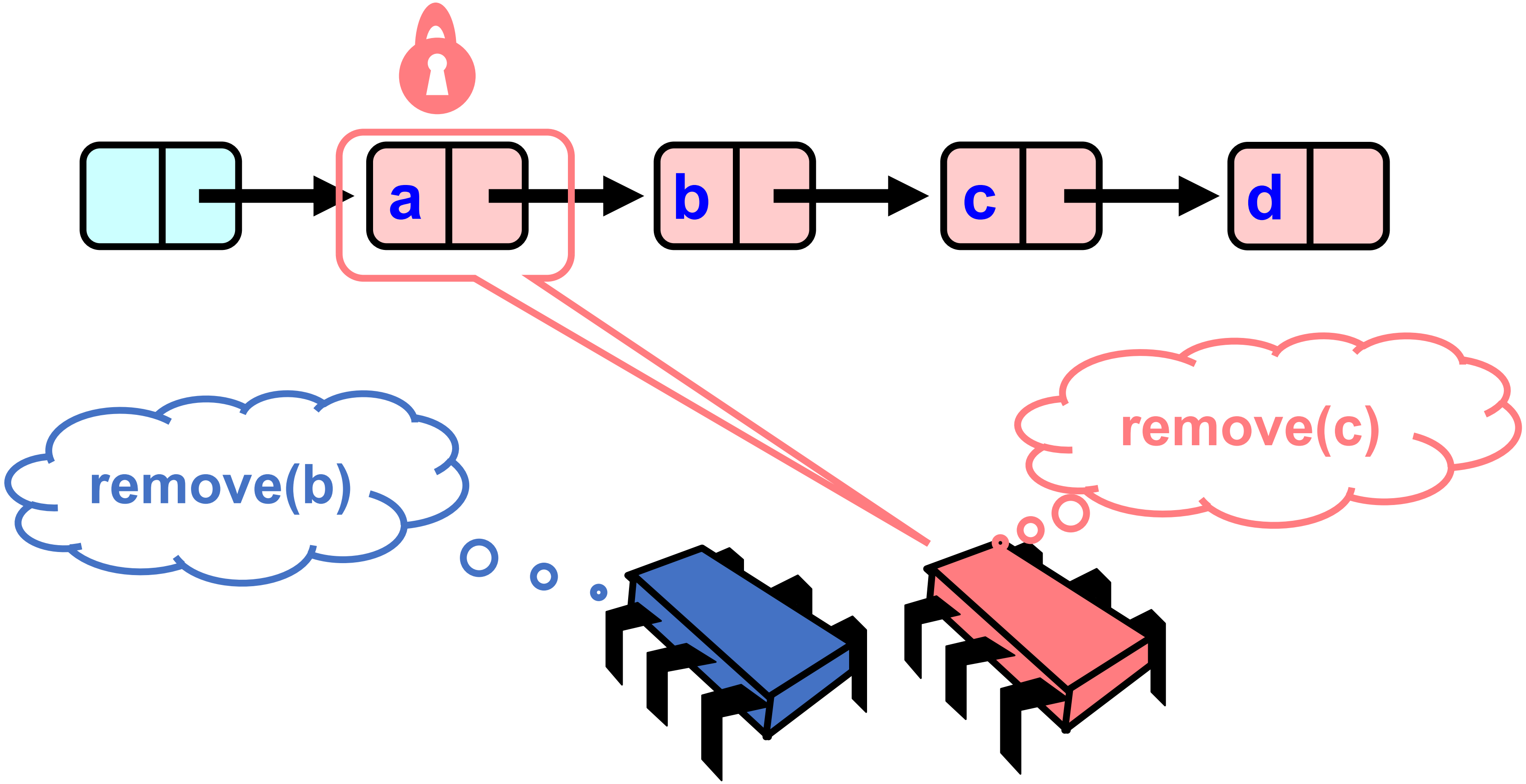
Removing a Node



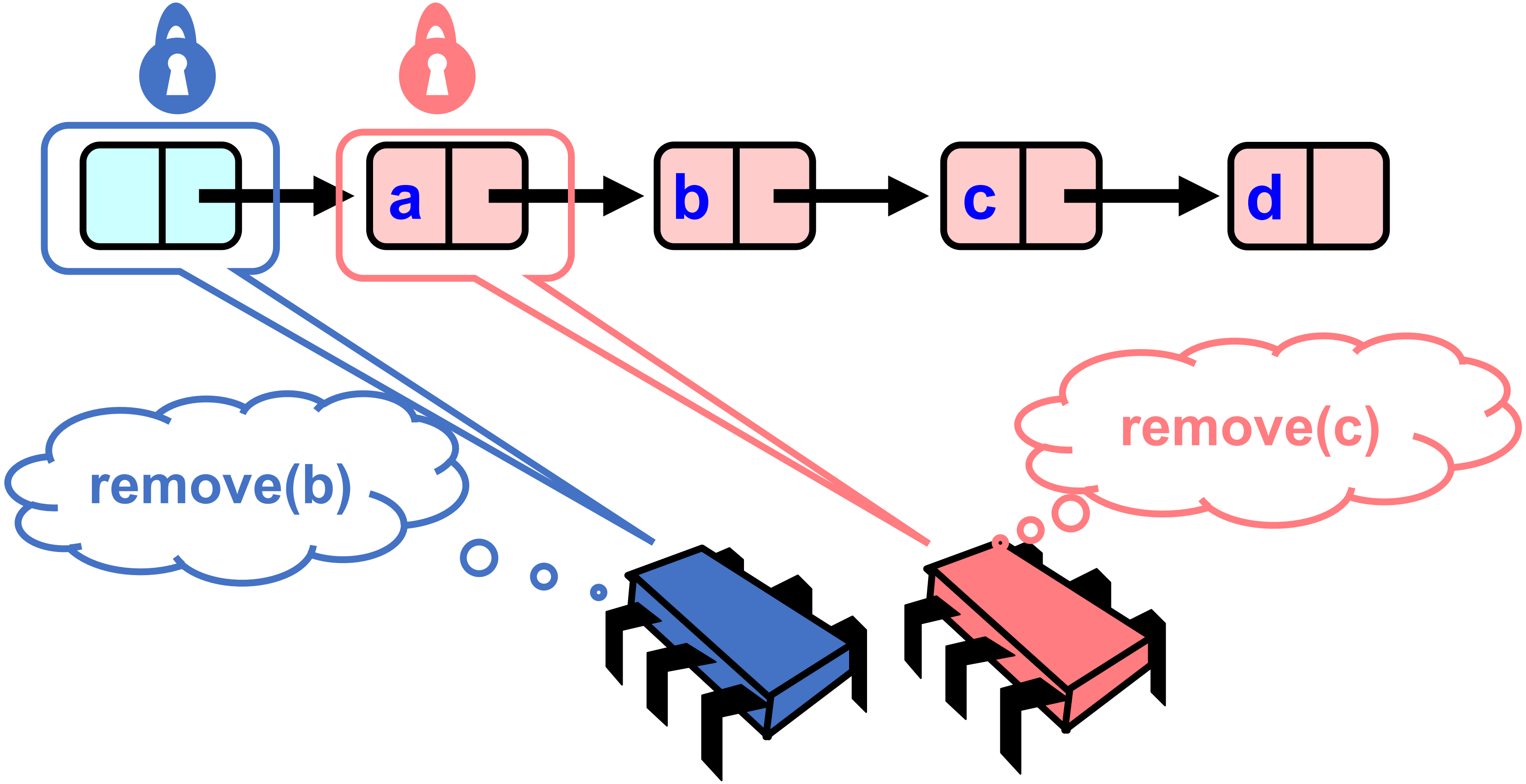
Removing a Node



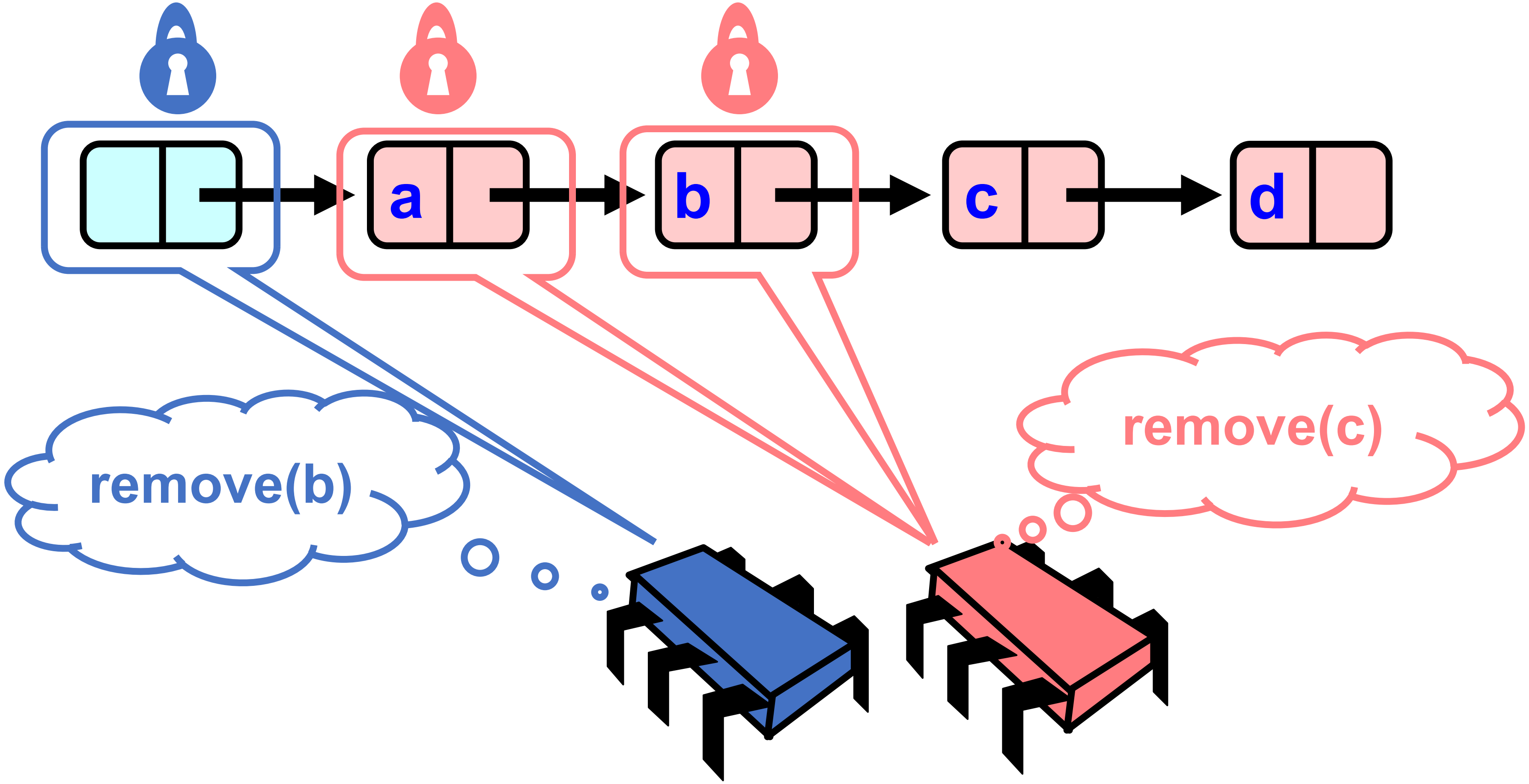
Removing a Node



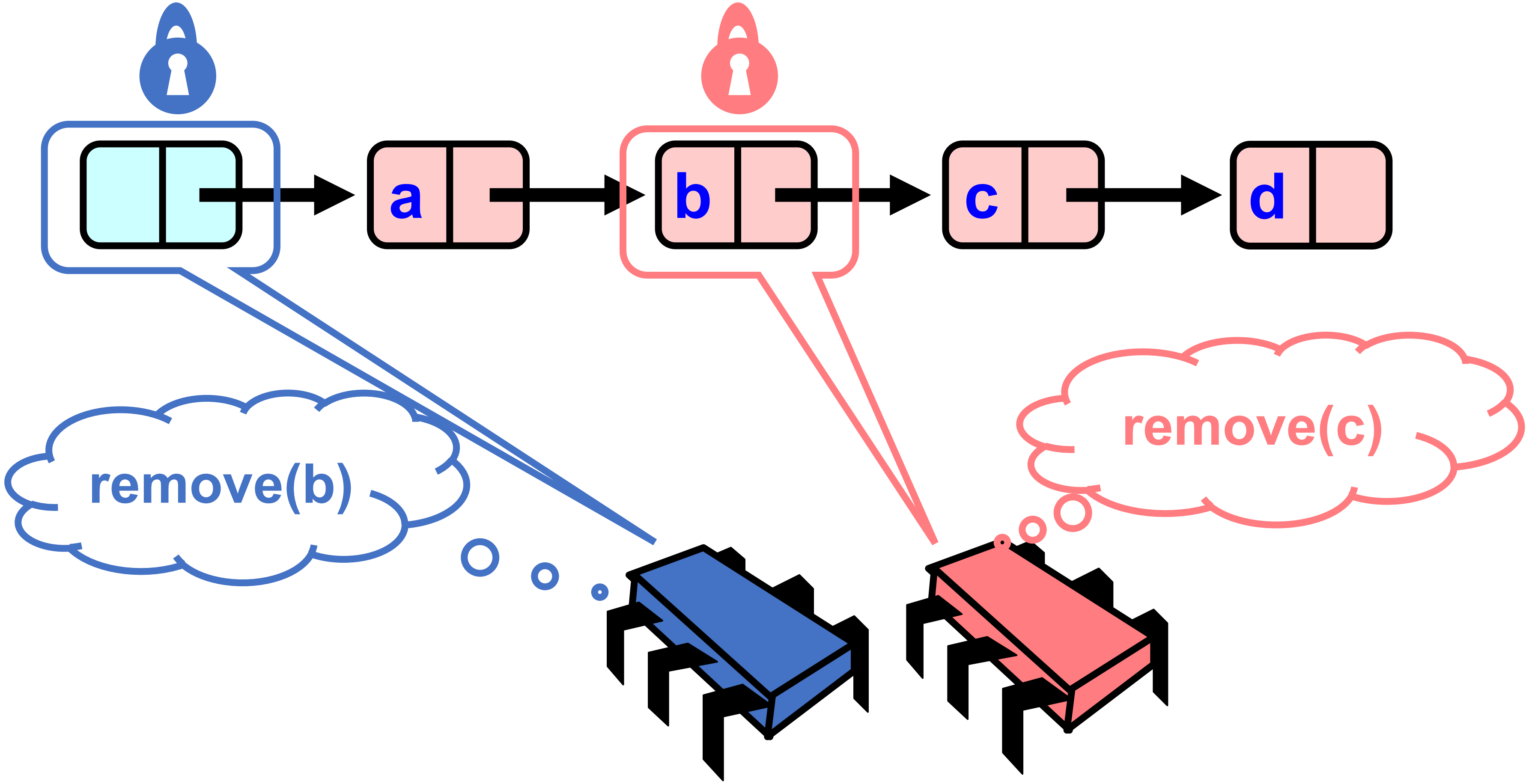
Removing a Node



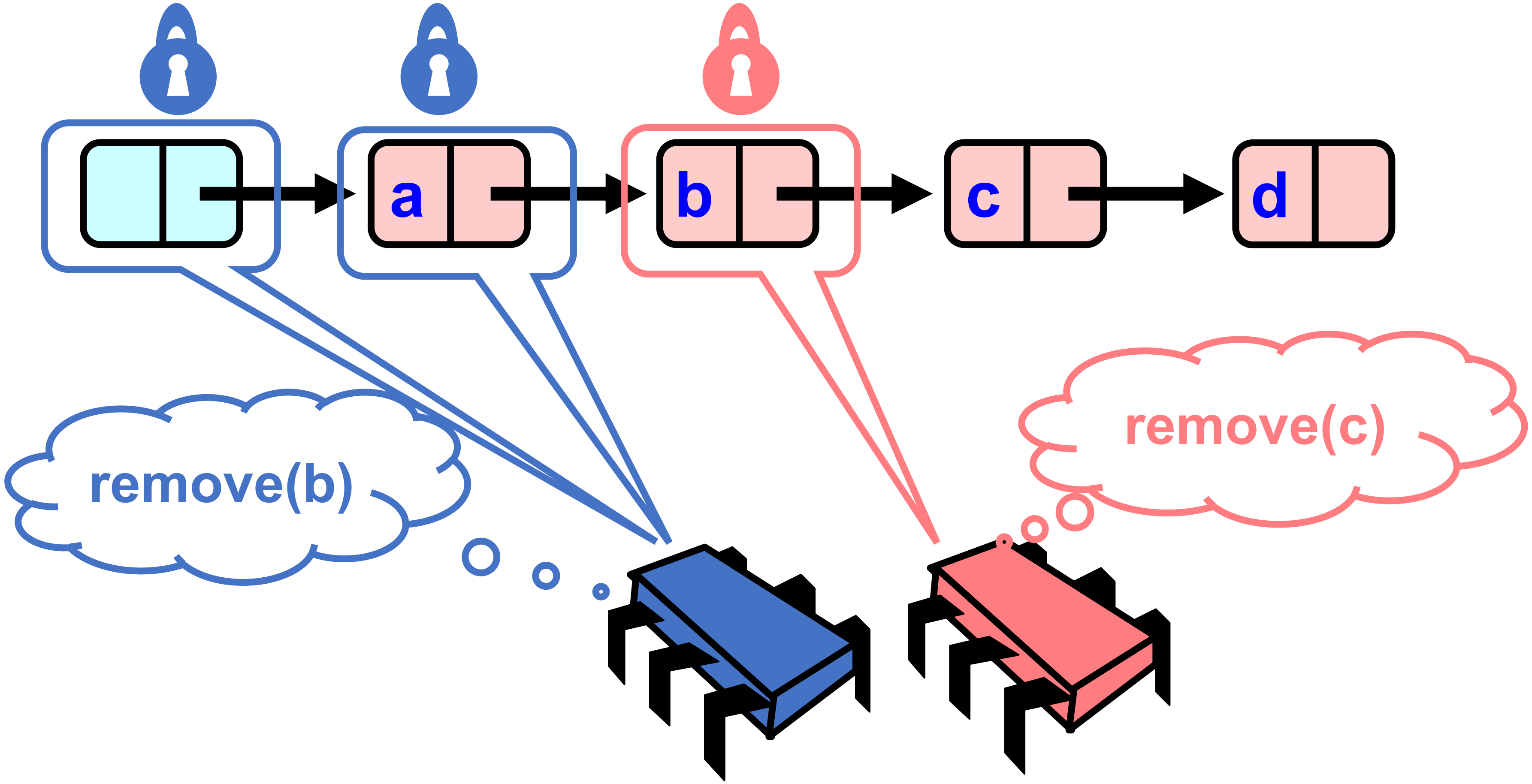
Removing a Node



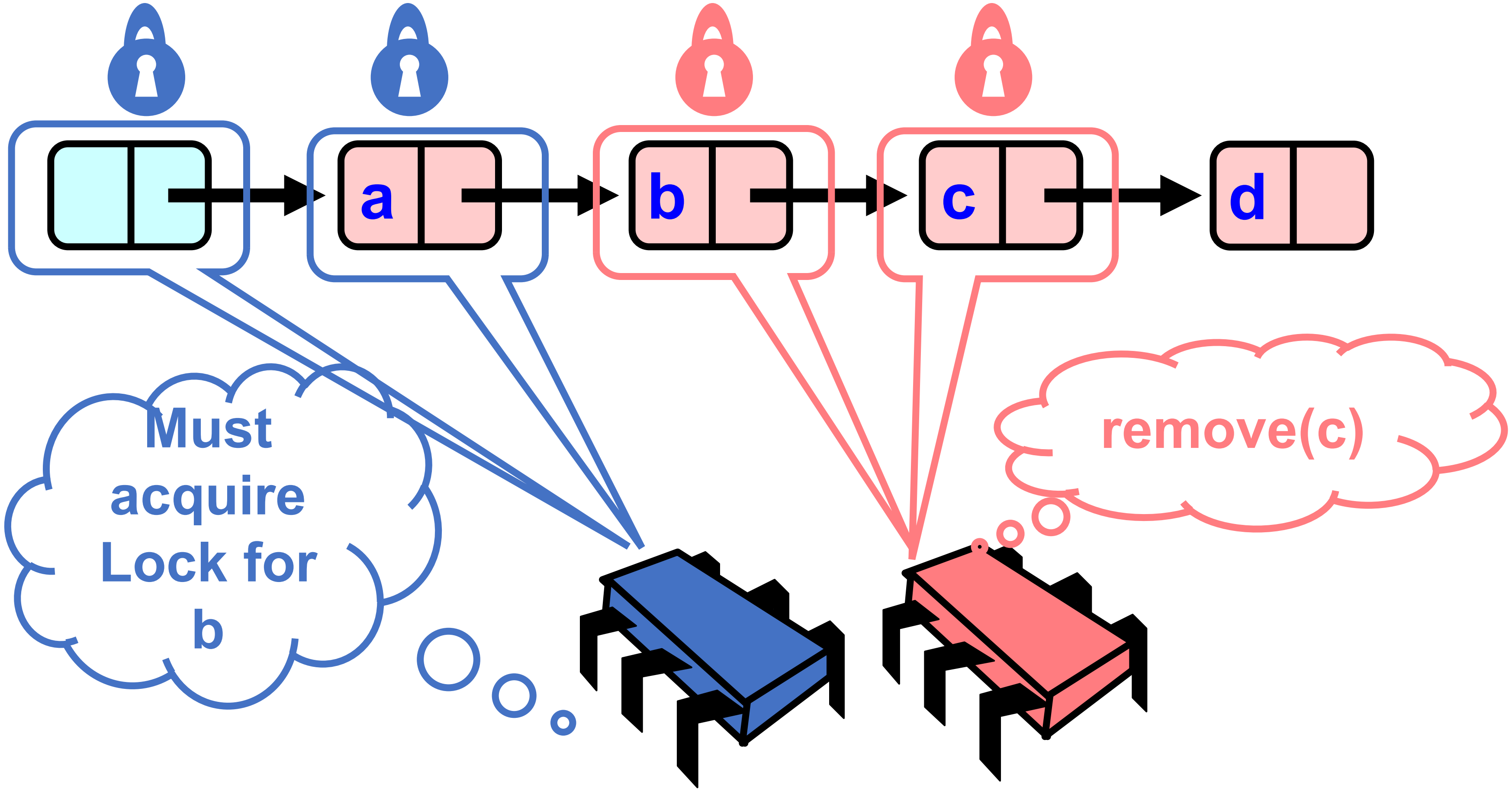
Removing a Node



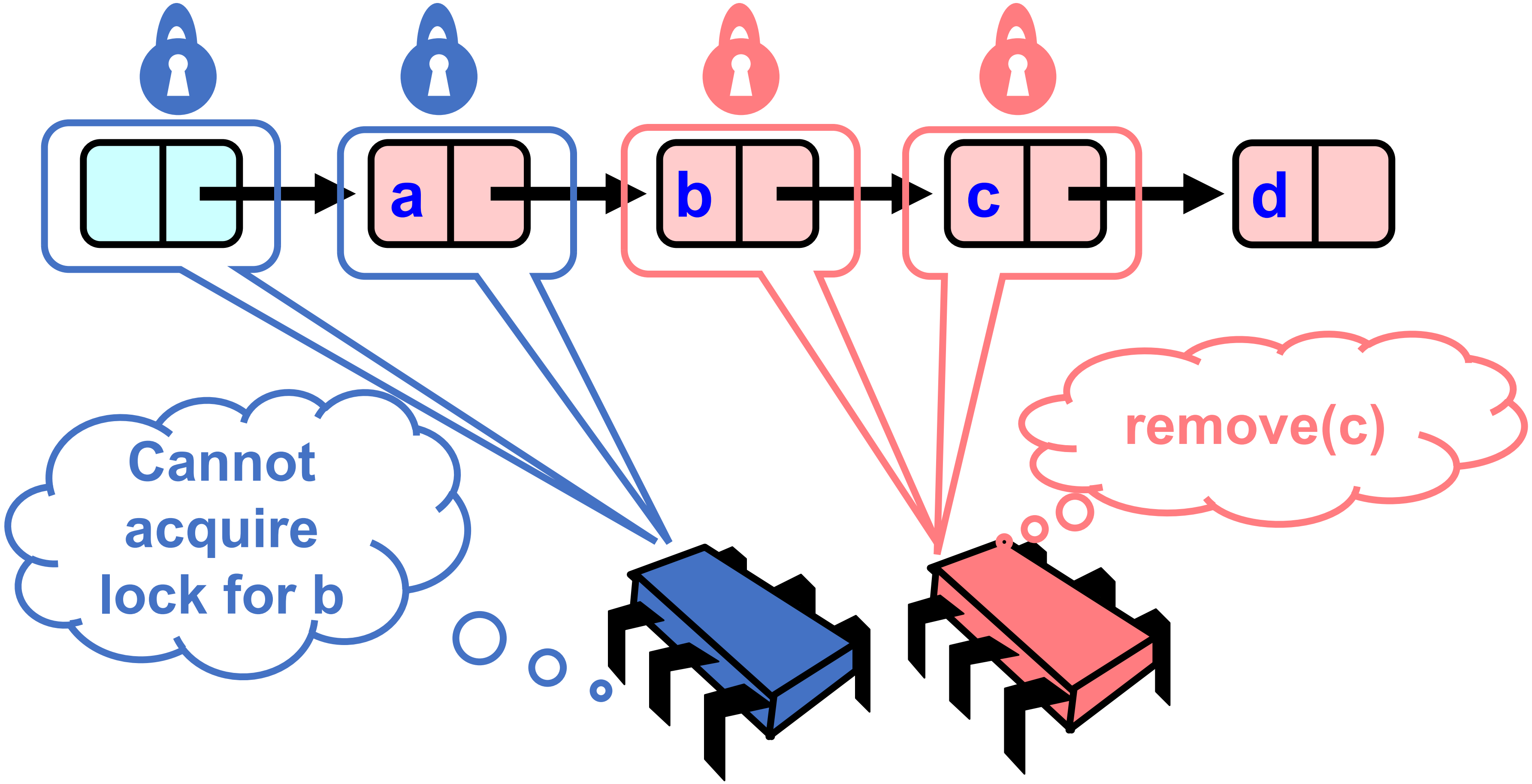
Removing a Node



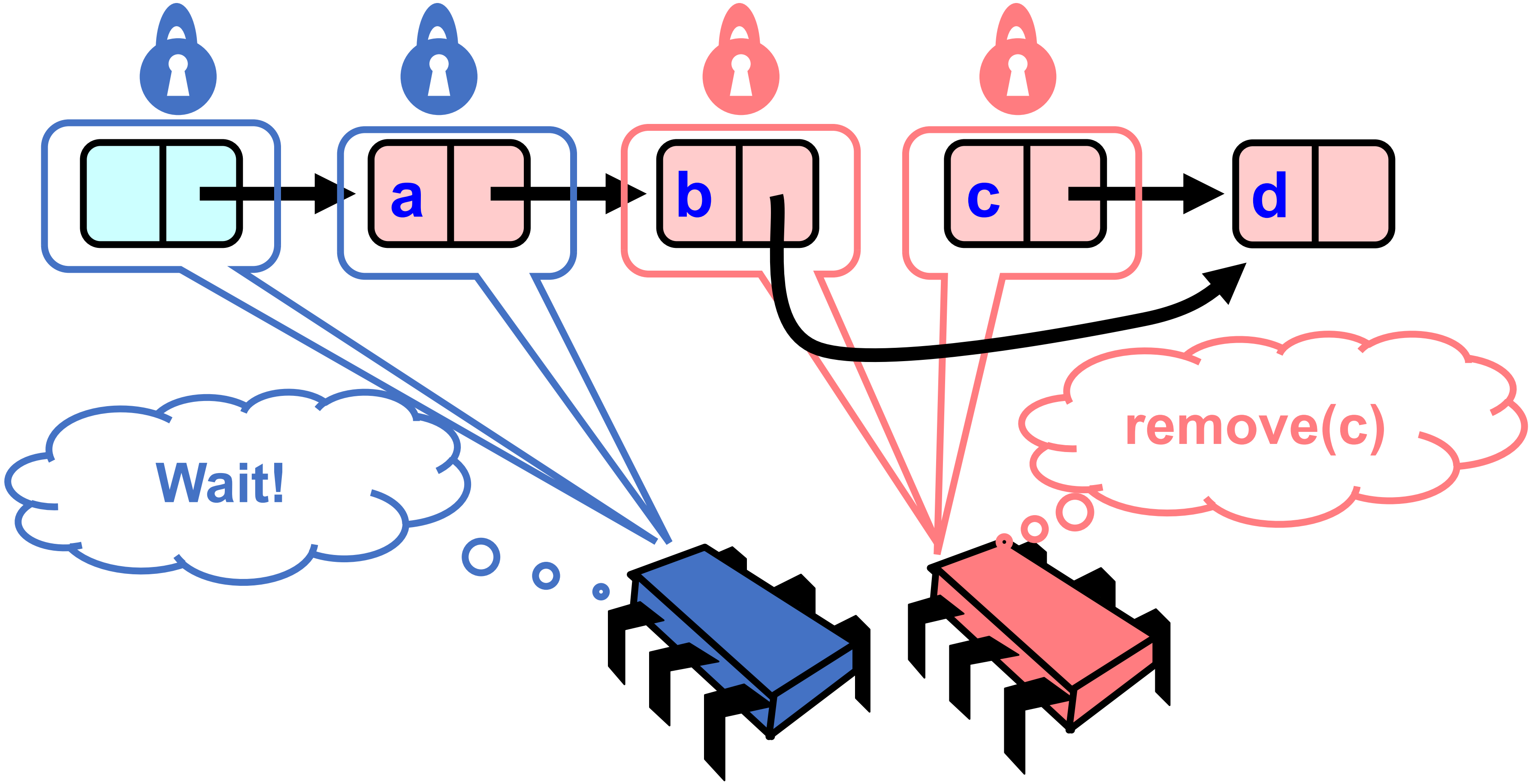
Removing a Node



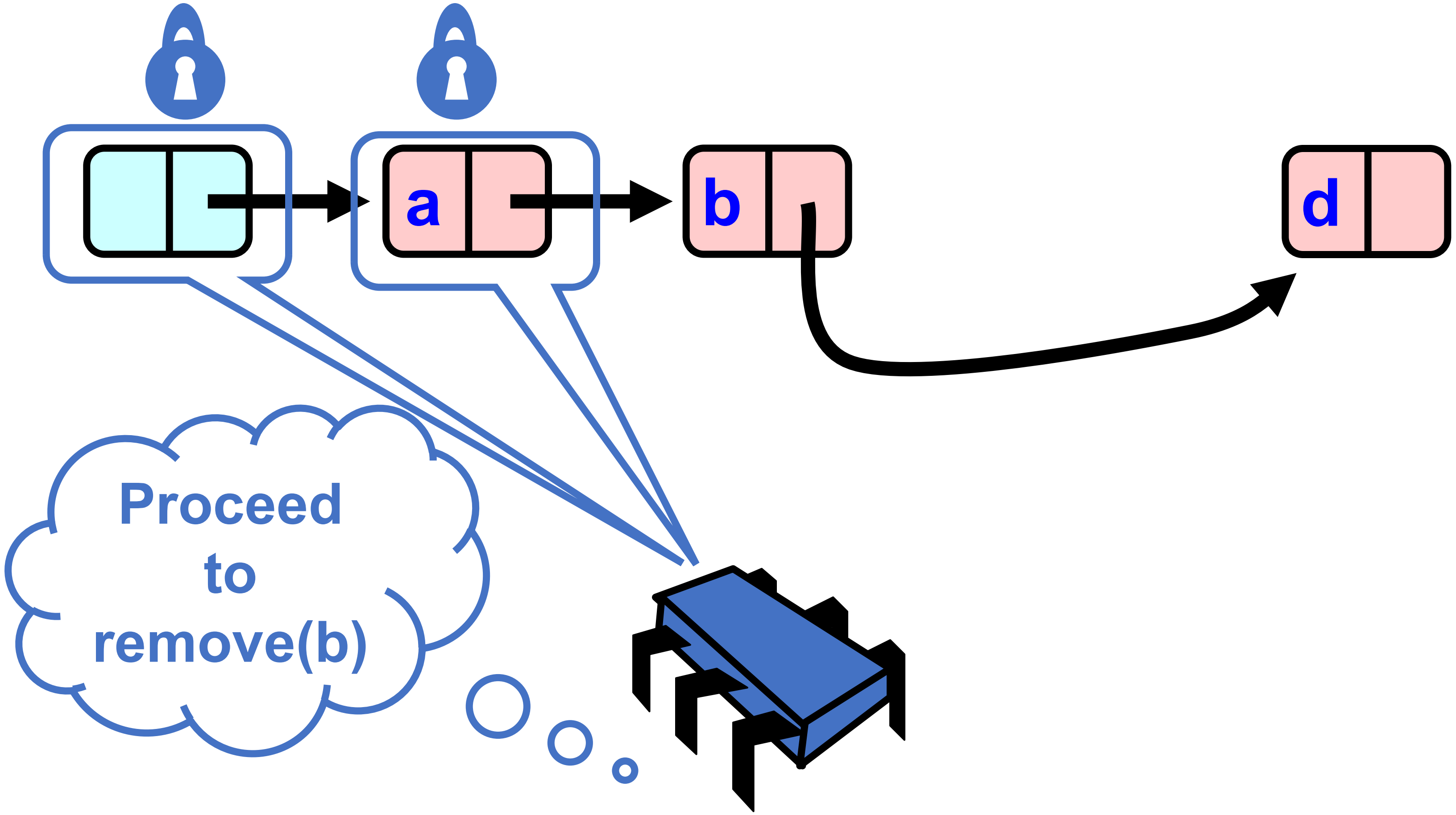
Removing a Node



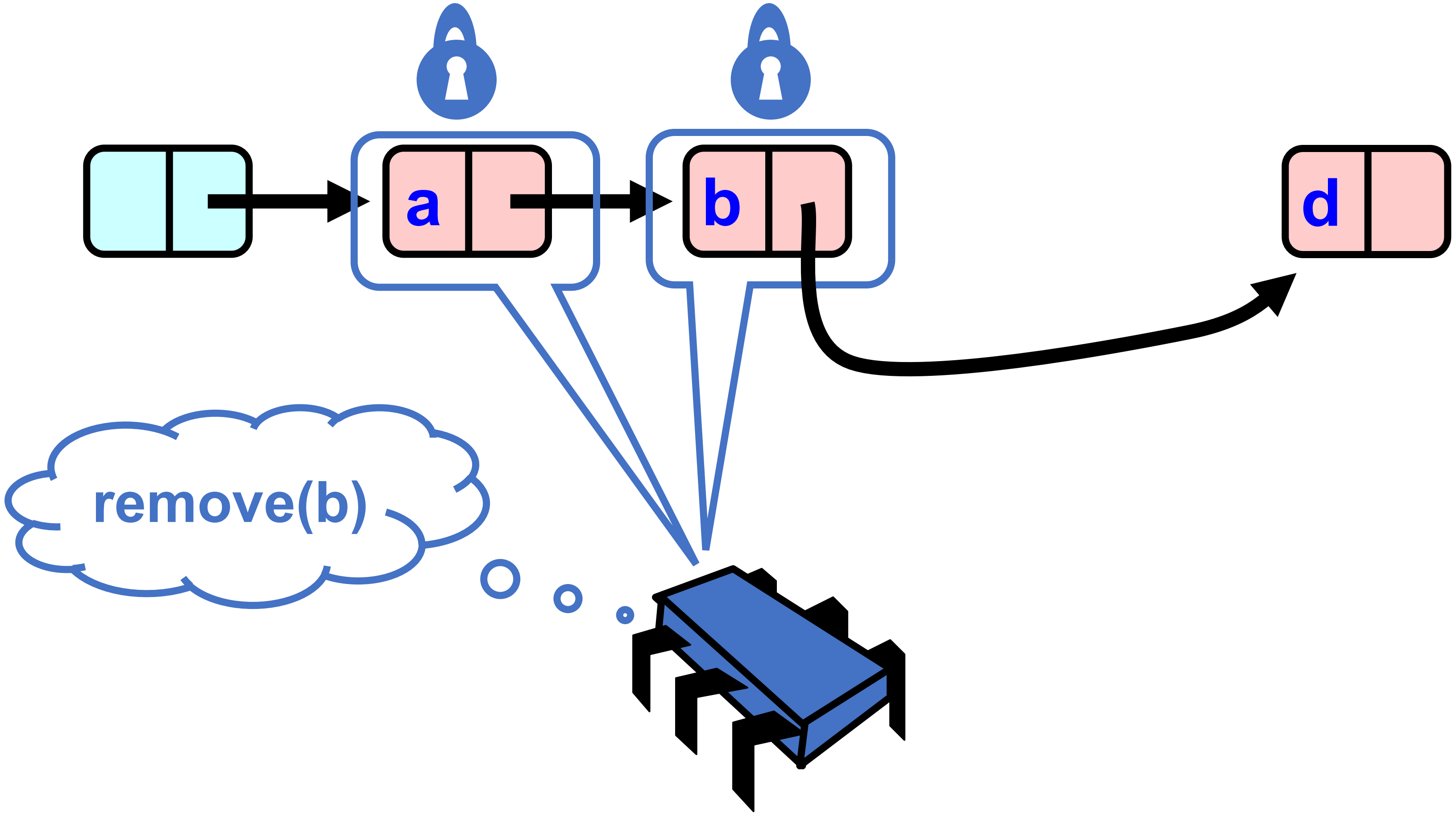
Removing a Node



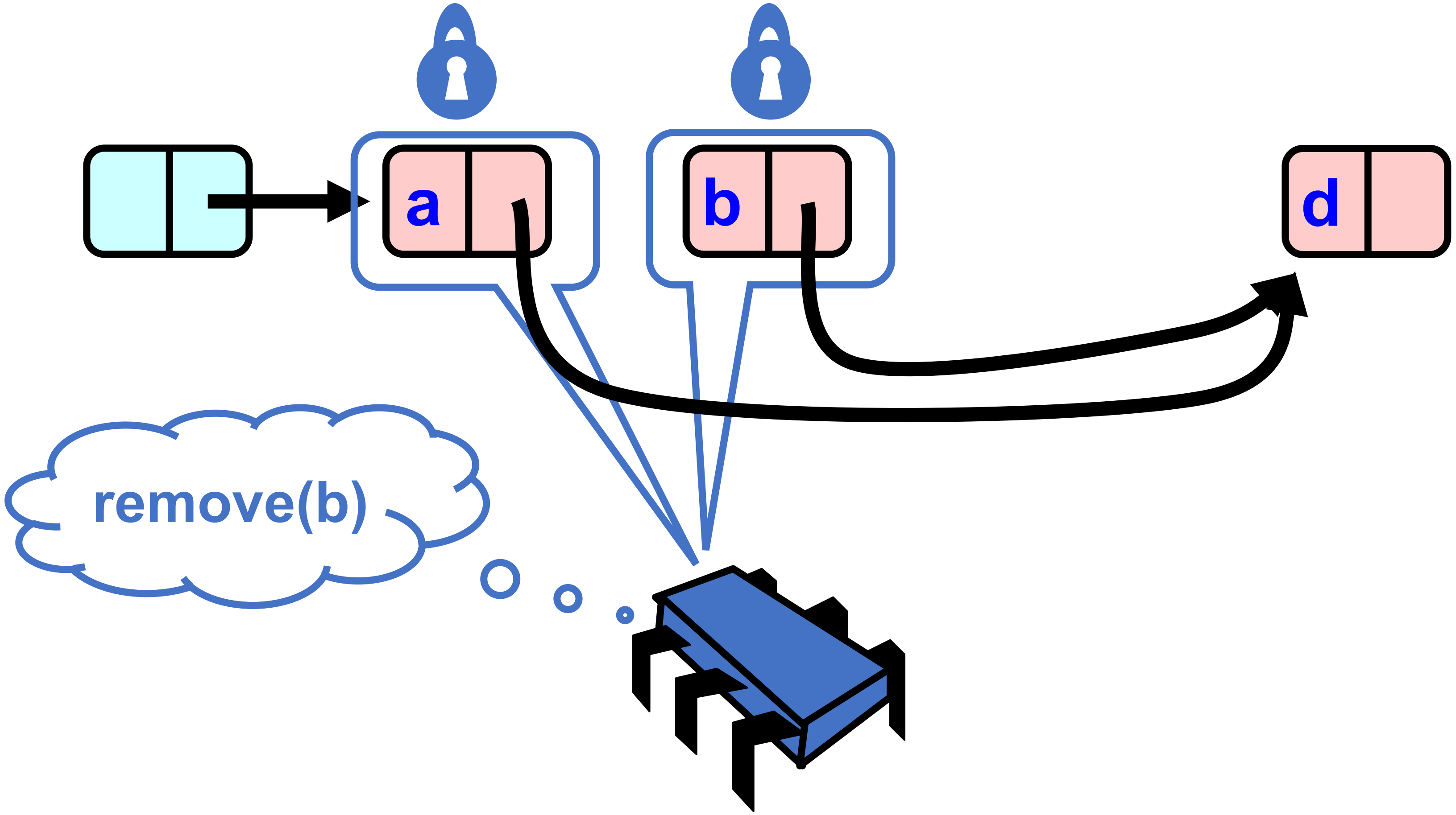
Removing a Node



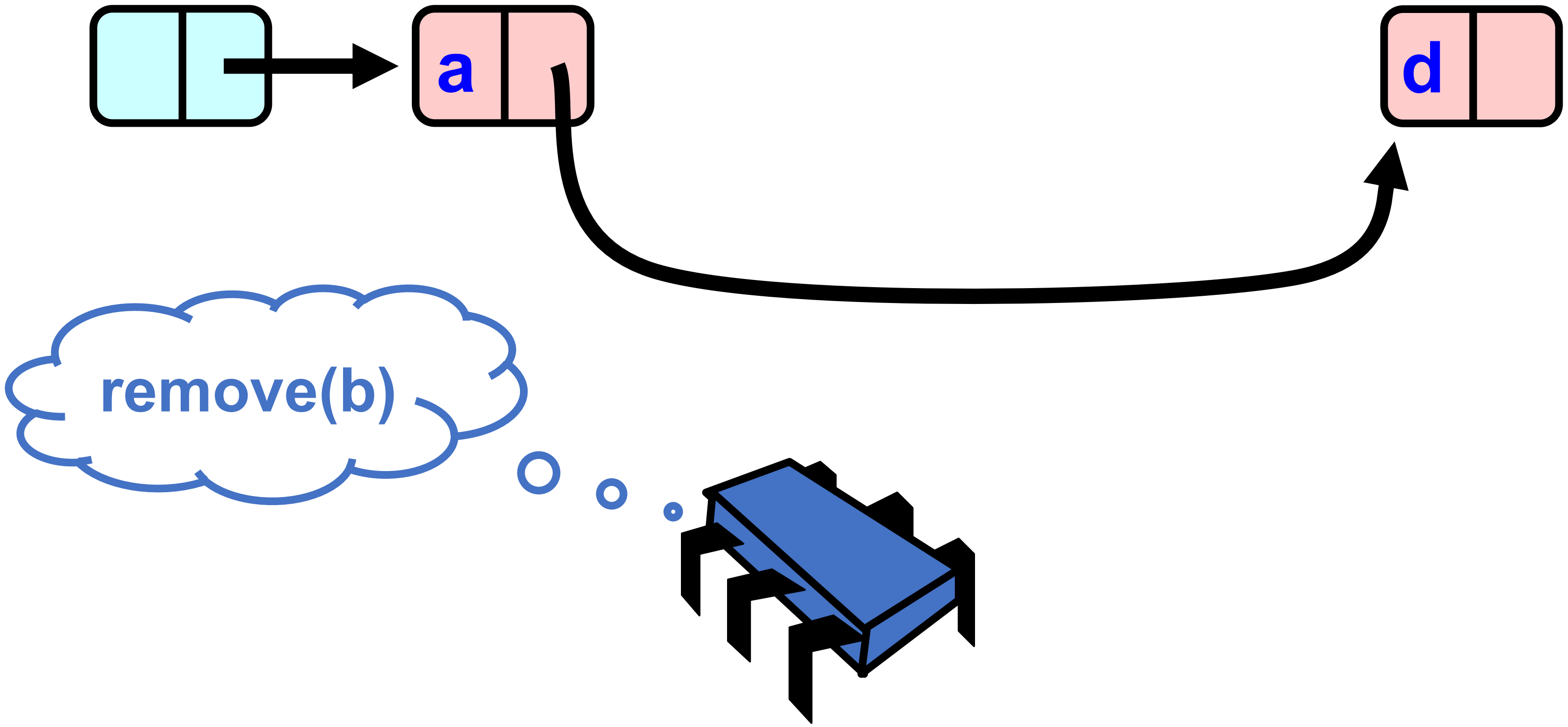
Removing a Node



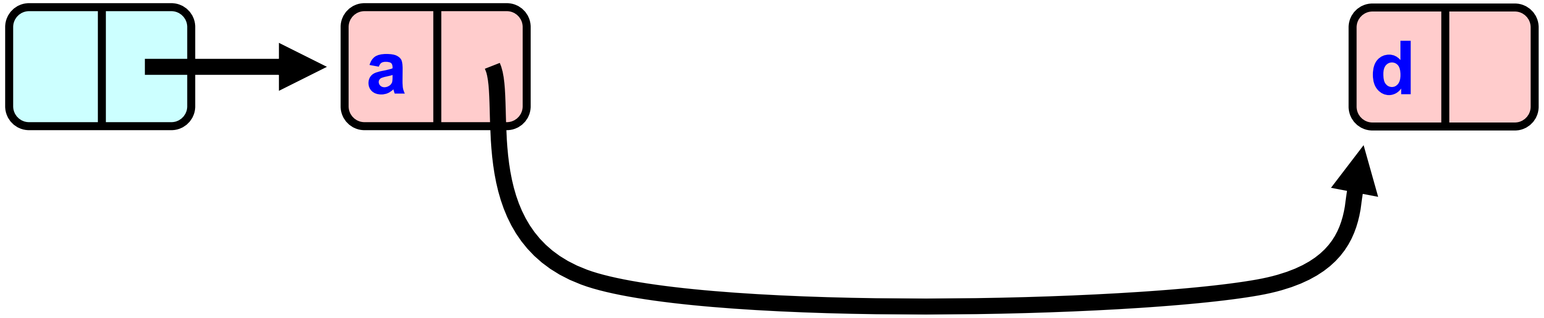
Removing a Node



Removing a Node



Removing a Node



Adding Nodes

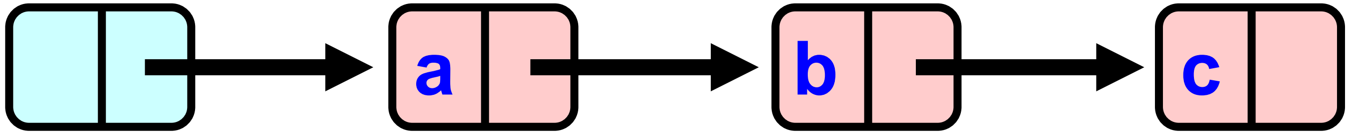
- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted
  - Is successor lock actually required?

Drawbacks

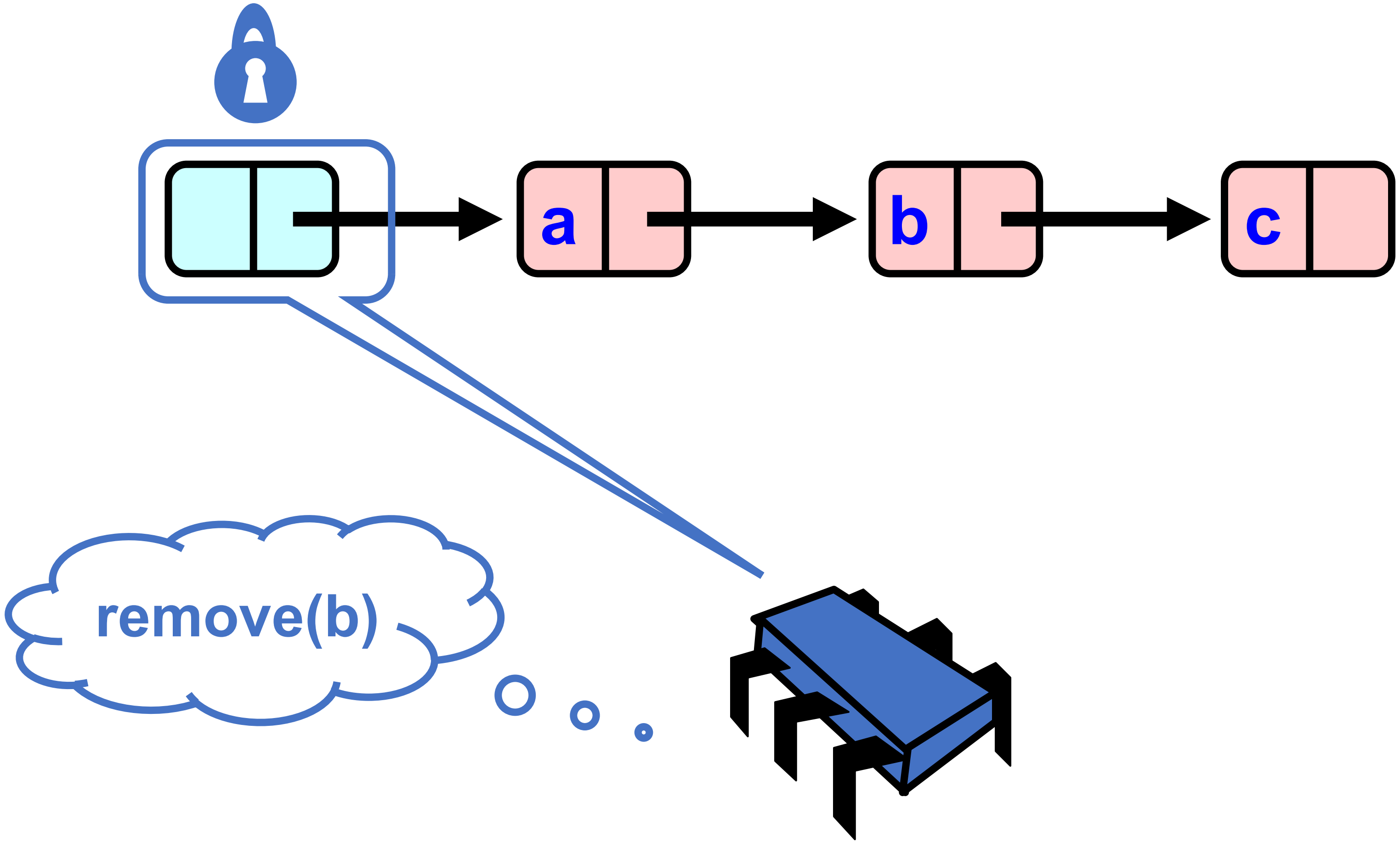
- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient

```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```

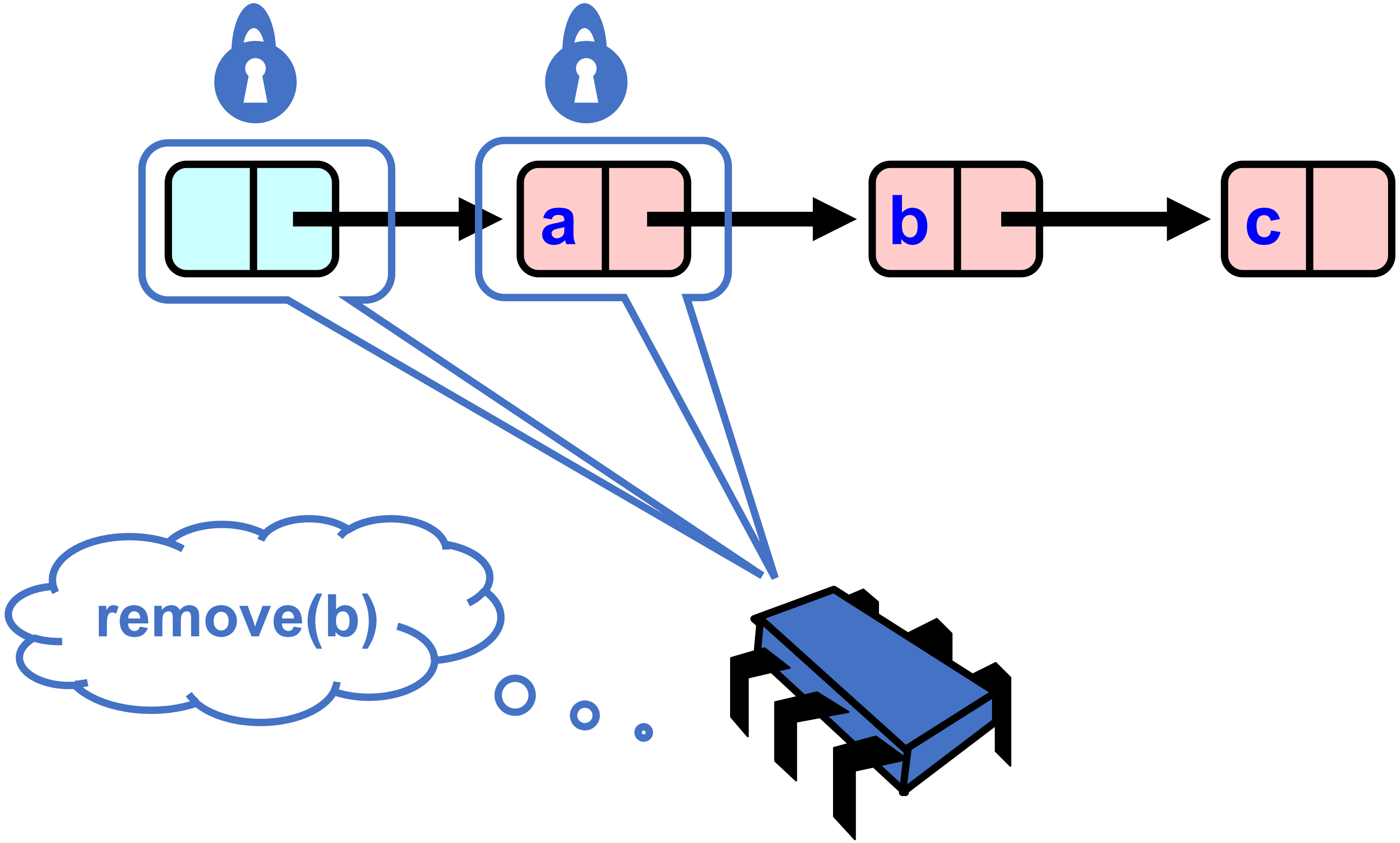
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



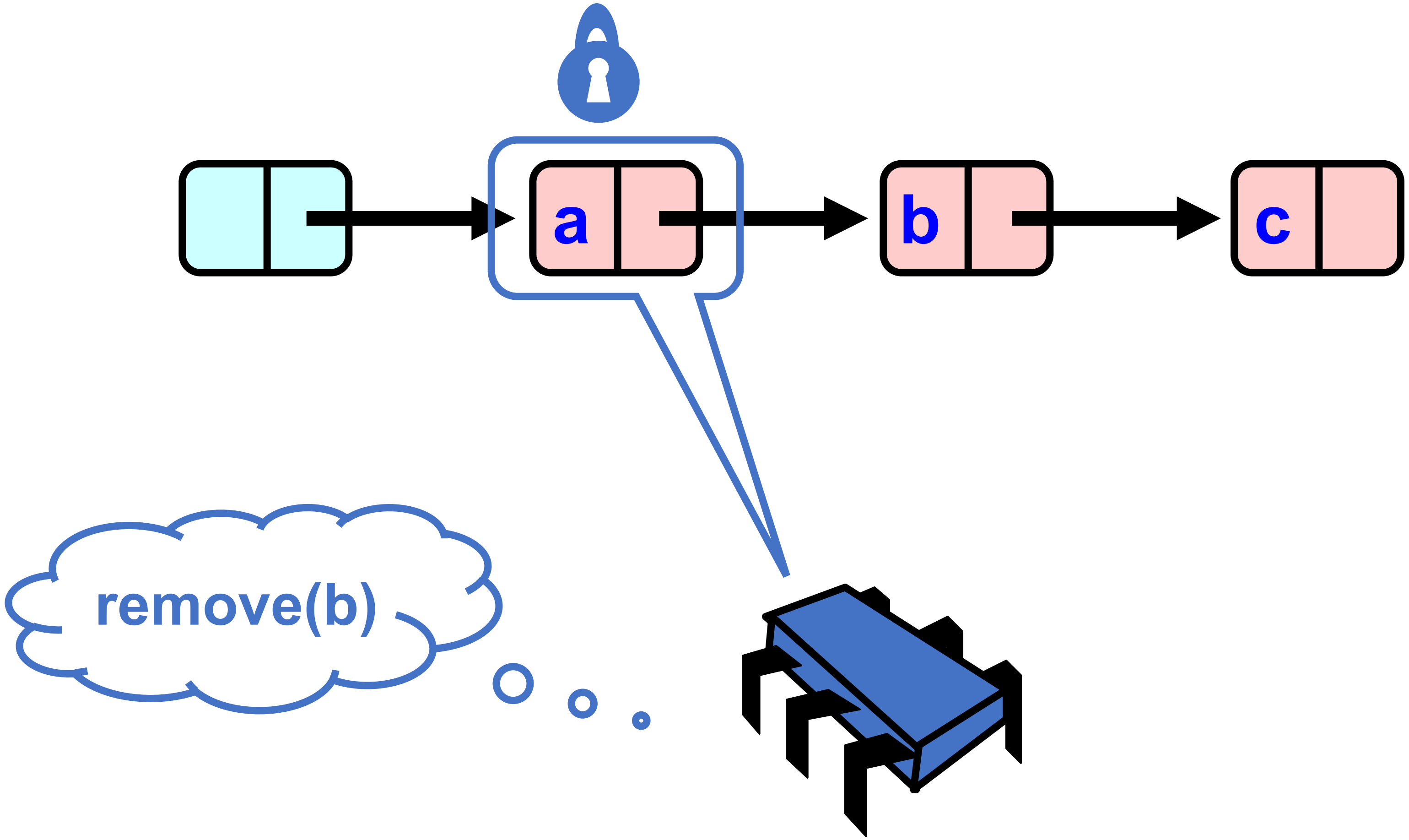
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



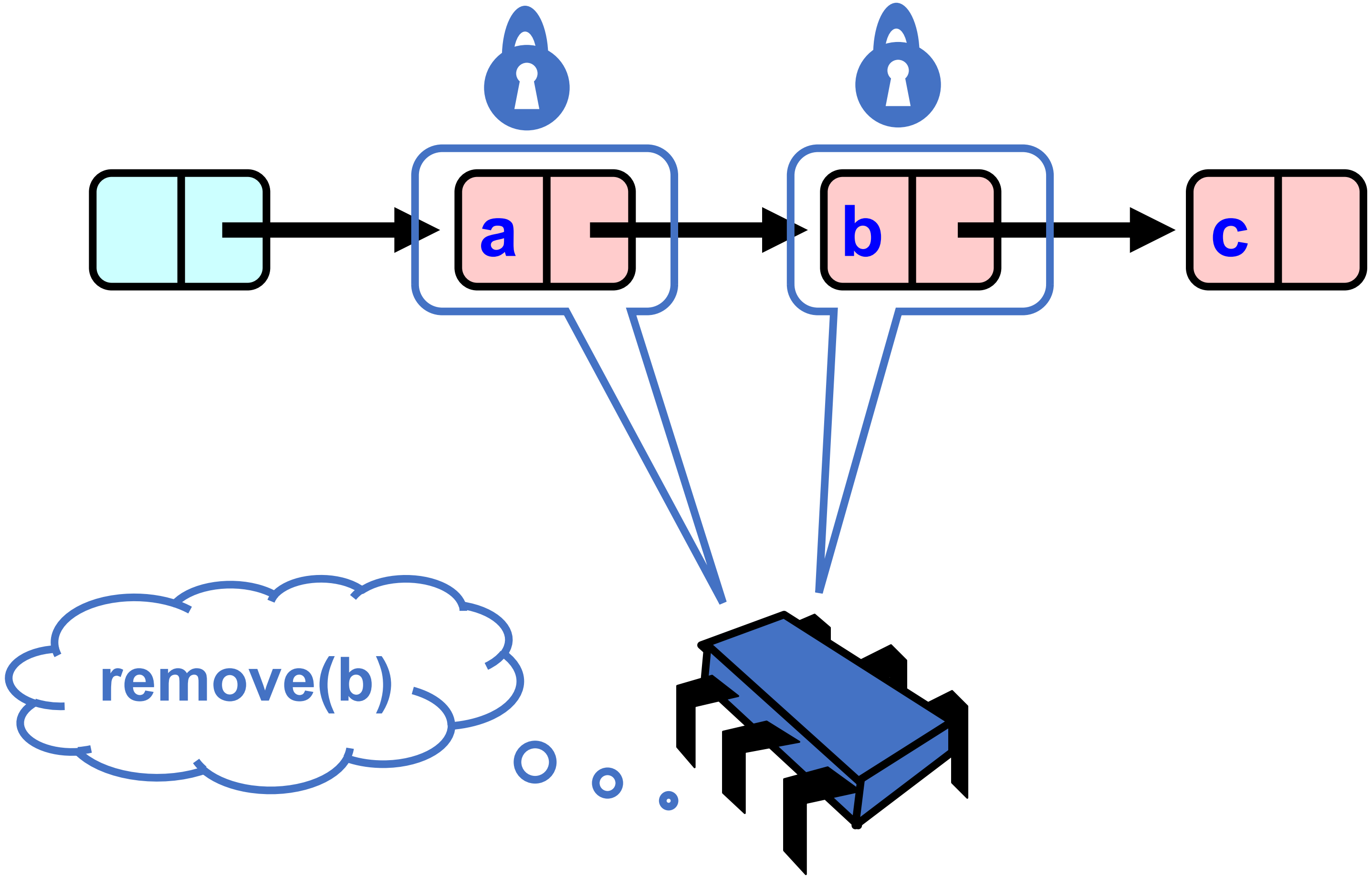
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



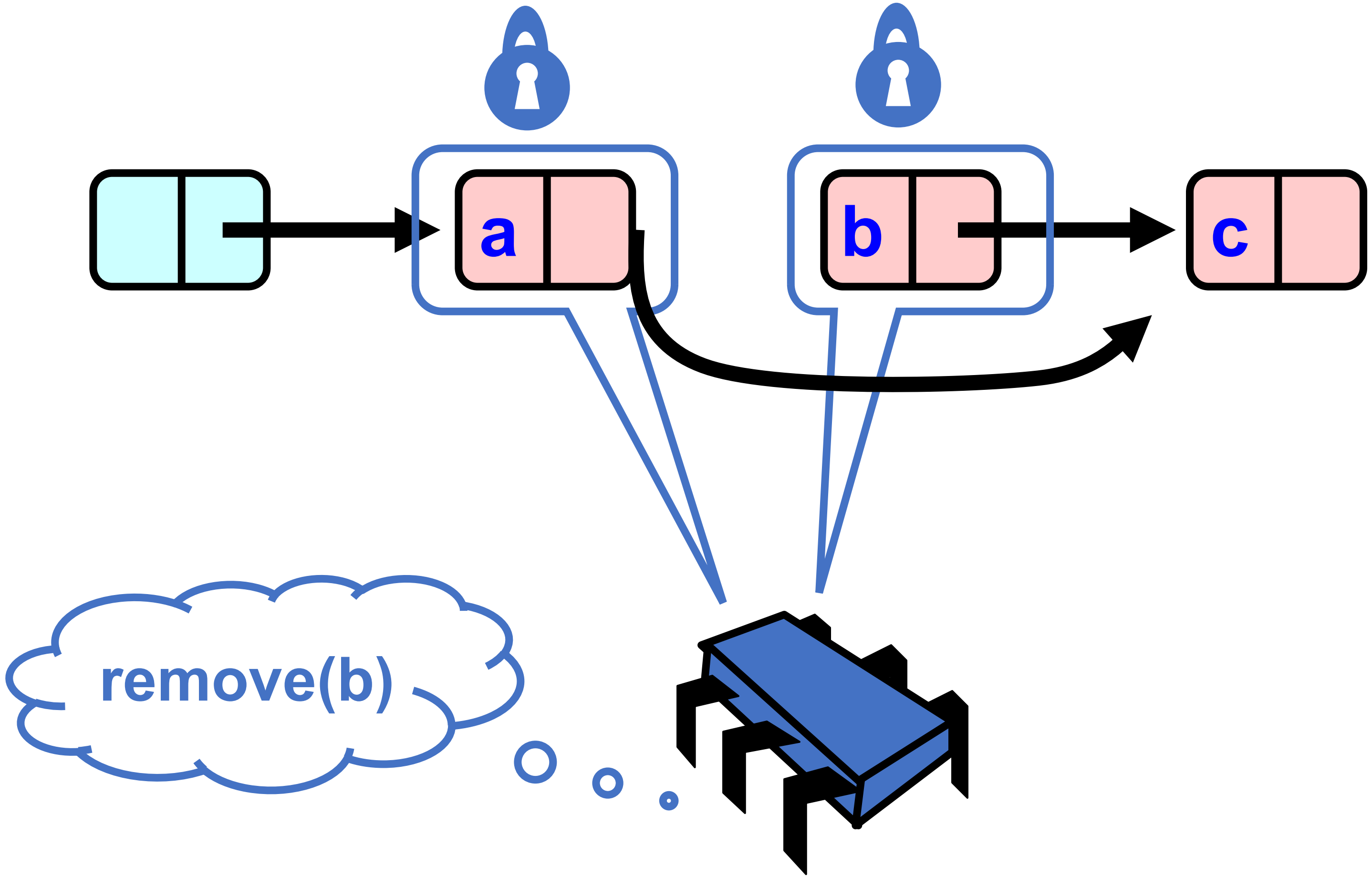
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



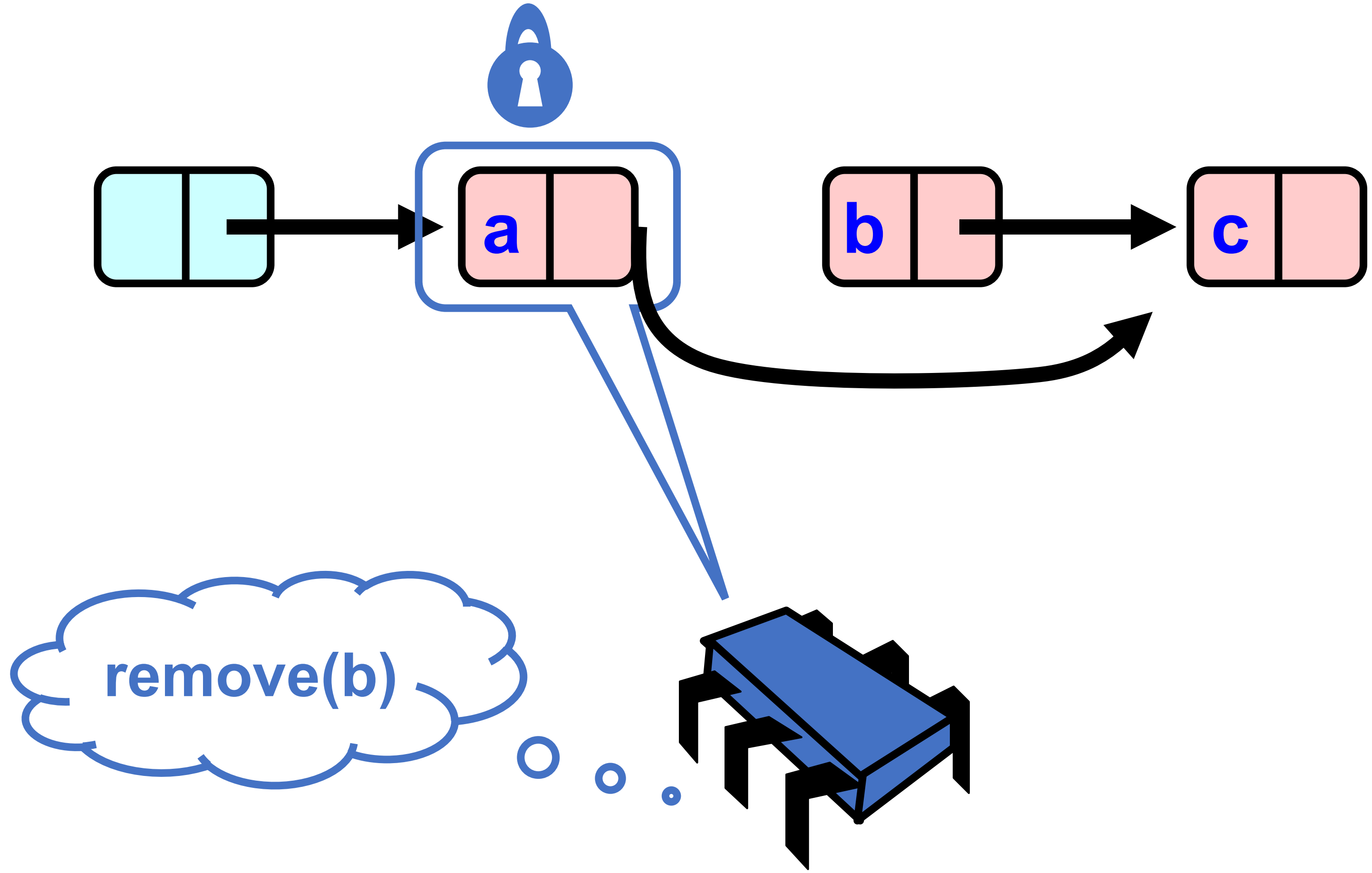
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



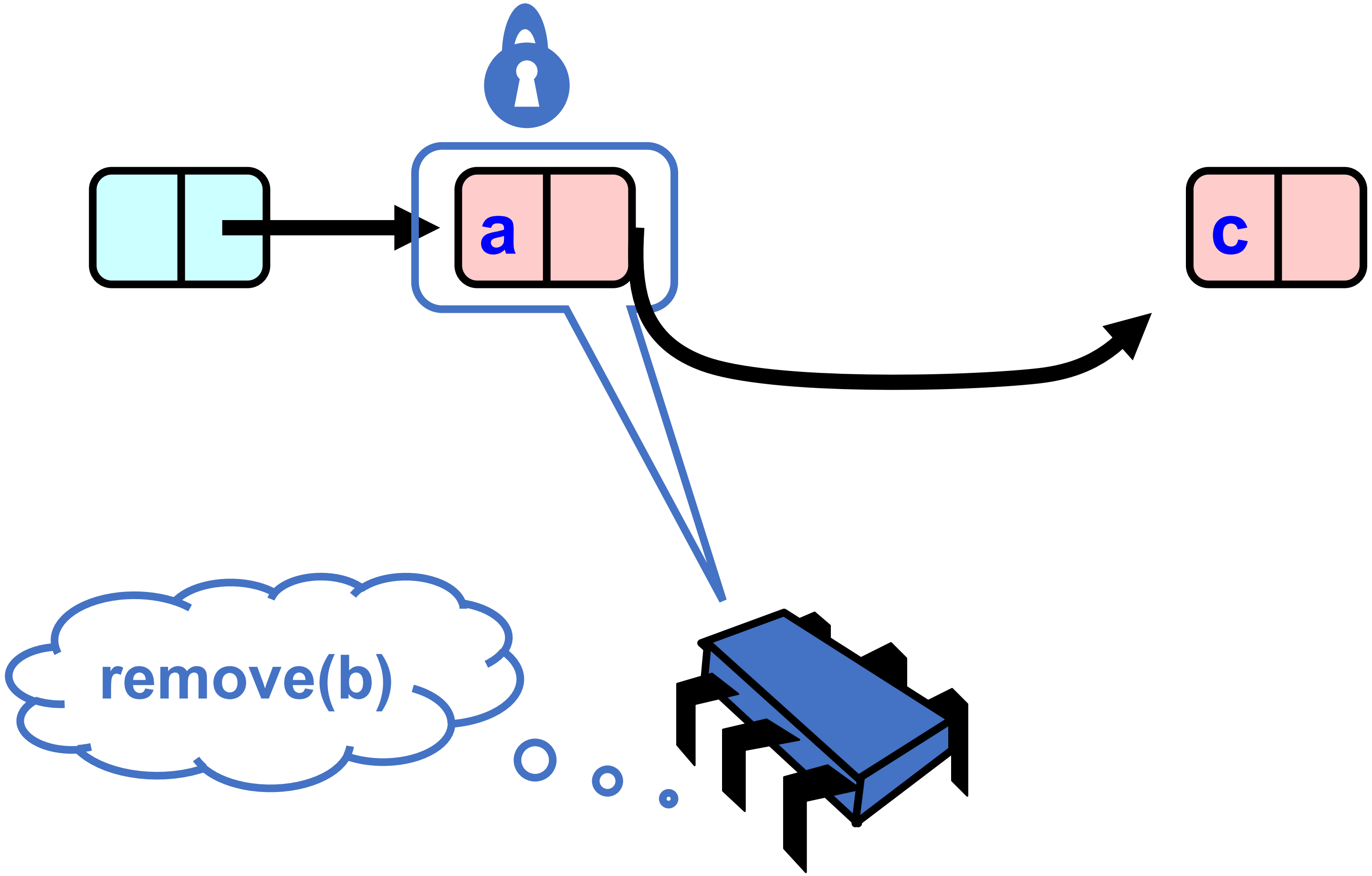
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



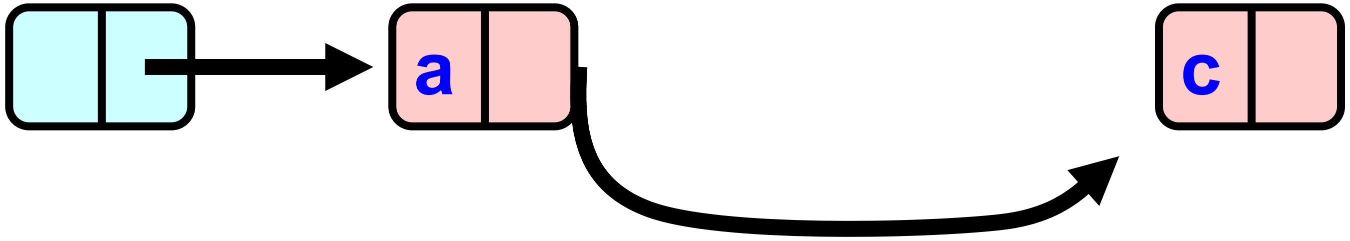
```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```

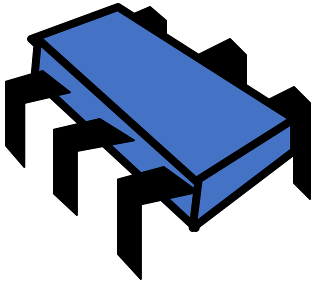


```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```



```
void remove(Value v) {
  Node* pred = NULL, *curr = NULL;
  head.lock();
  pred = head;
  curr = pred.next();
  curr.lock();
  while (curr.value != v) {
    pred.ulock();
    pred = curr;
    curr = curr.next();
    curr.lock();
  }
  pred.next = curr.next;
  curr.unlock();
  pred.unlock();
}
```





Schedule

- C++ Atomic Template

- Concurrent set
    - Coarse-grained lock
    - fine-grained lock
    - **optimistic locking**

How can we improve

- Acquires and releases lock for every node traversed
  - If we have a long list to search, it can be bad!
  - reduces concurrency (traffic jams)

Optimistic Synchronization

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

Optimistic Synchronization

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

What was the alternative?

Optimistic Synchronization

- Find nodes without locking

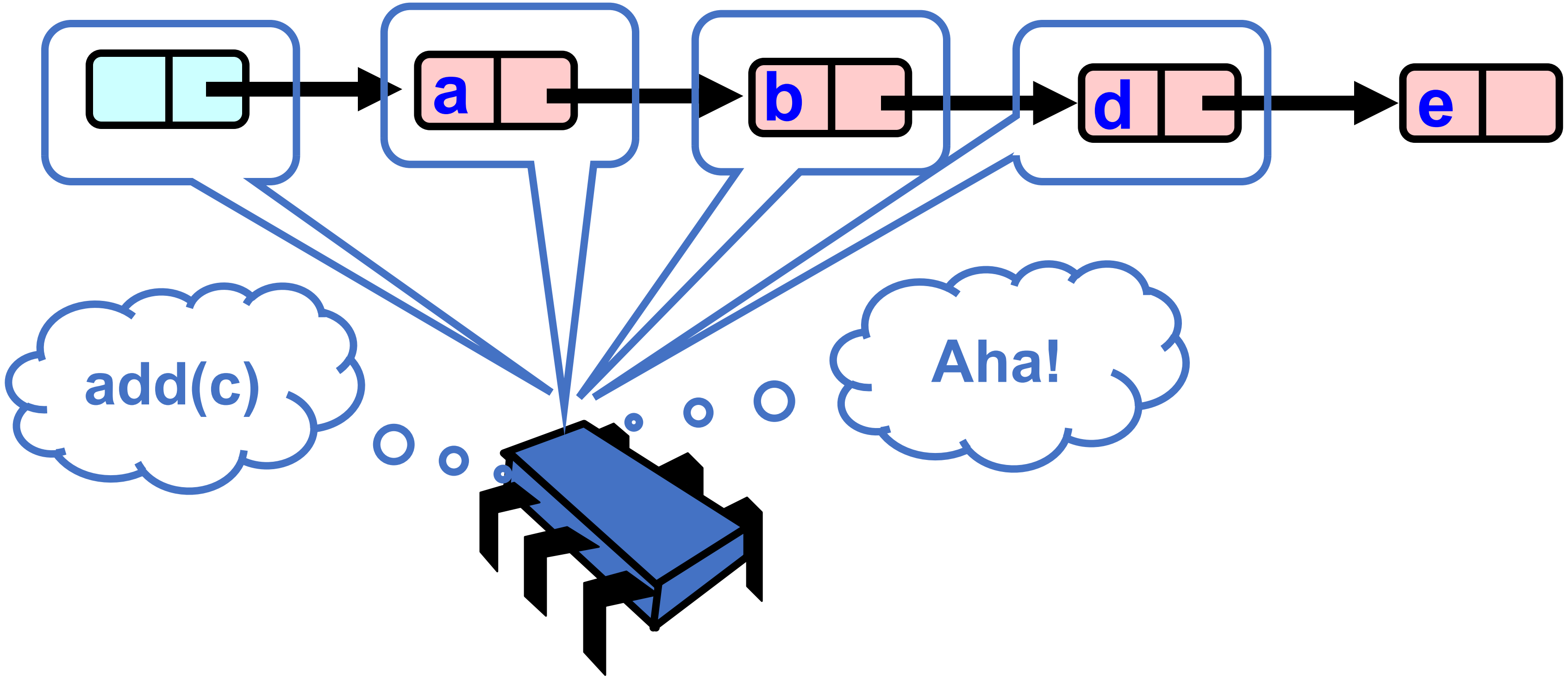
Optimistic Synchronization

- Find nodes without locking
- Lock nodes

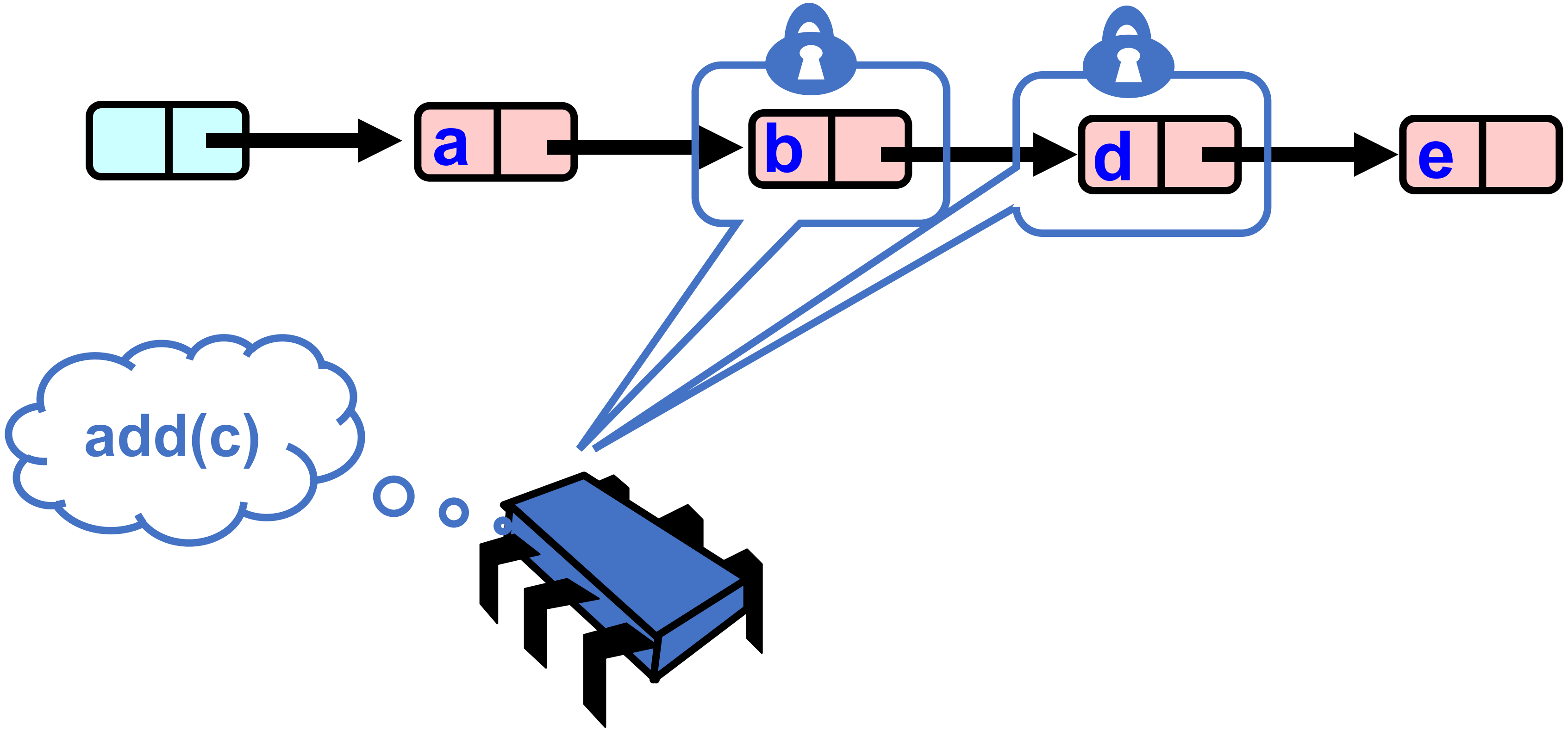
Optimistic Synchronization

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

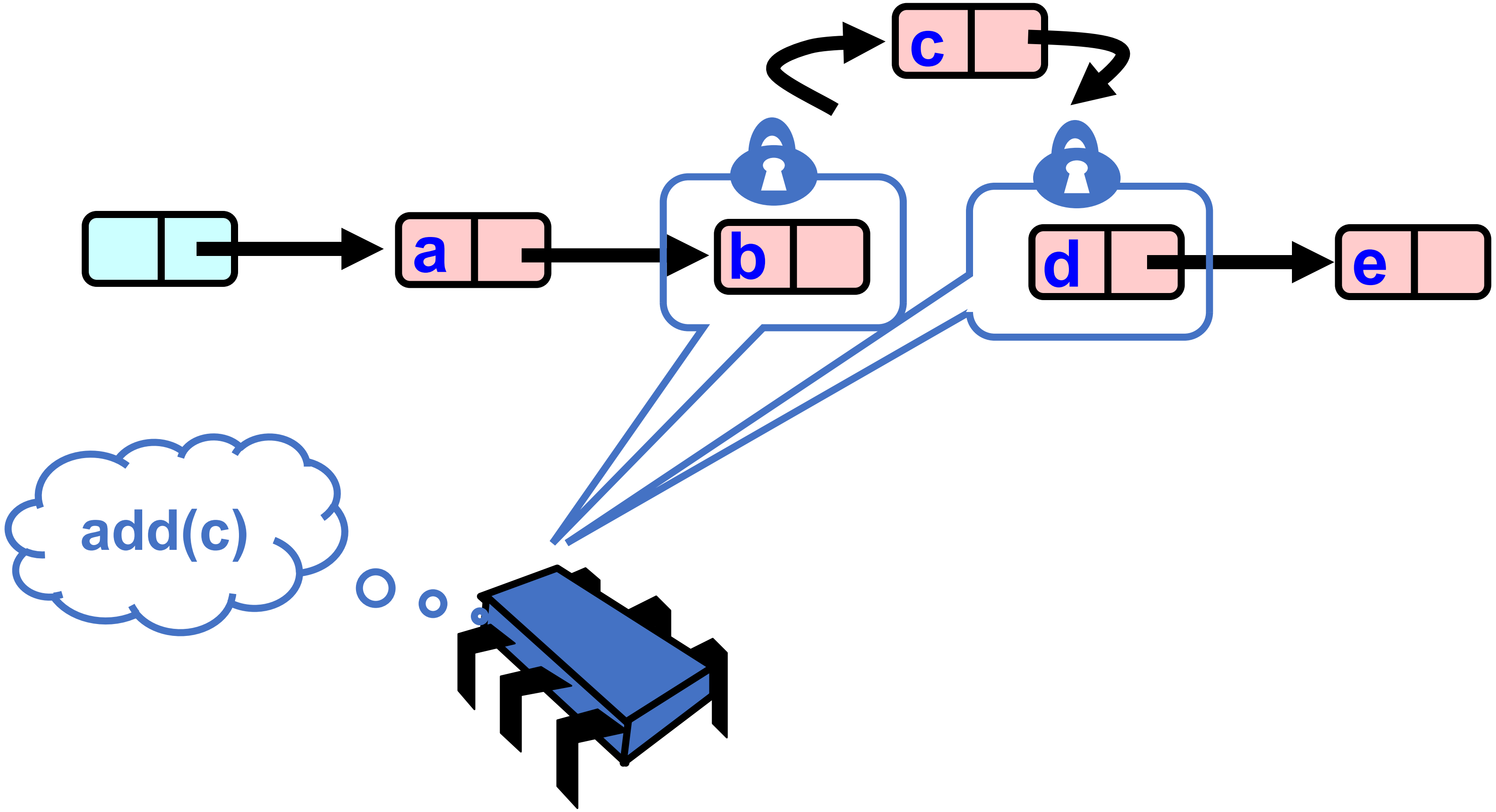
Optimistic: Traverse without Locking



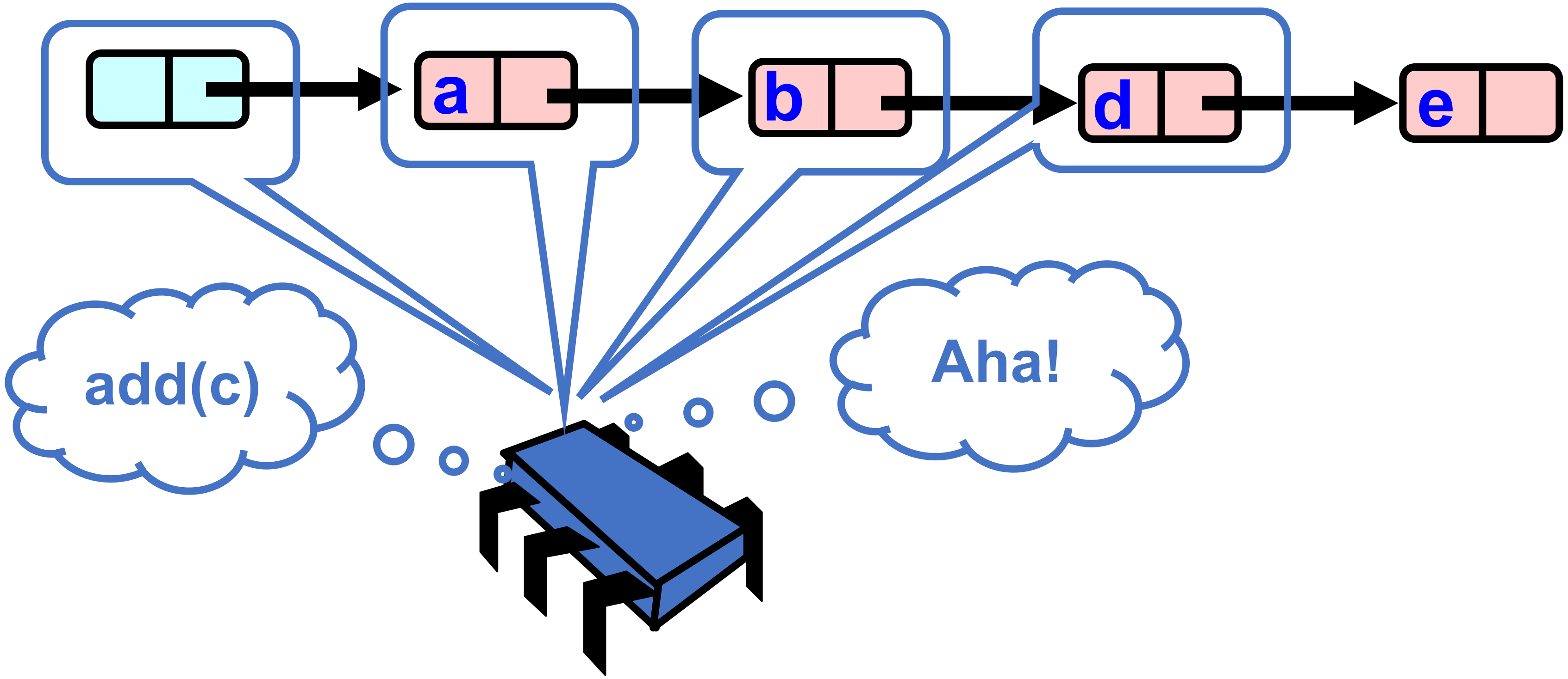
Optimistic: Lock and Load



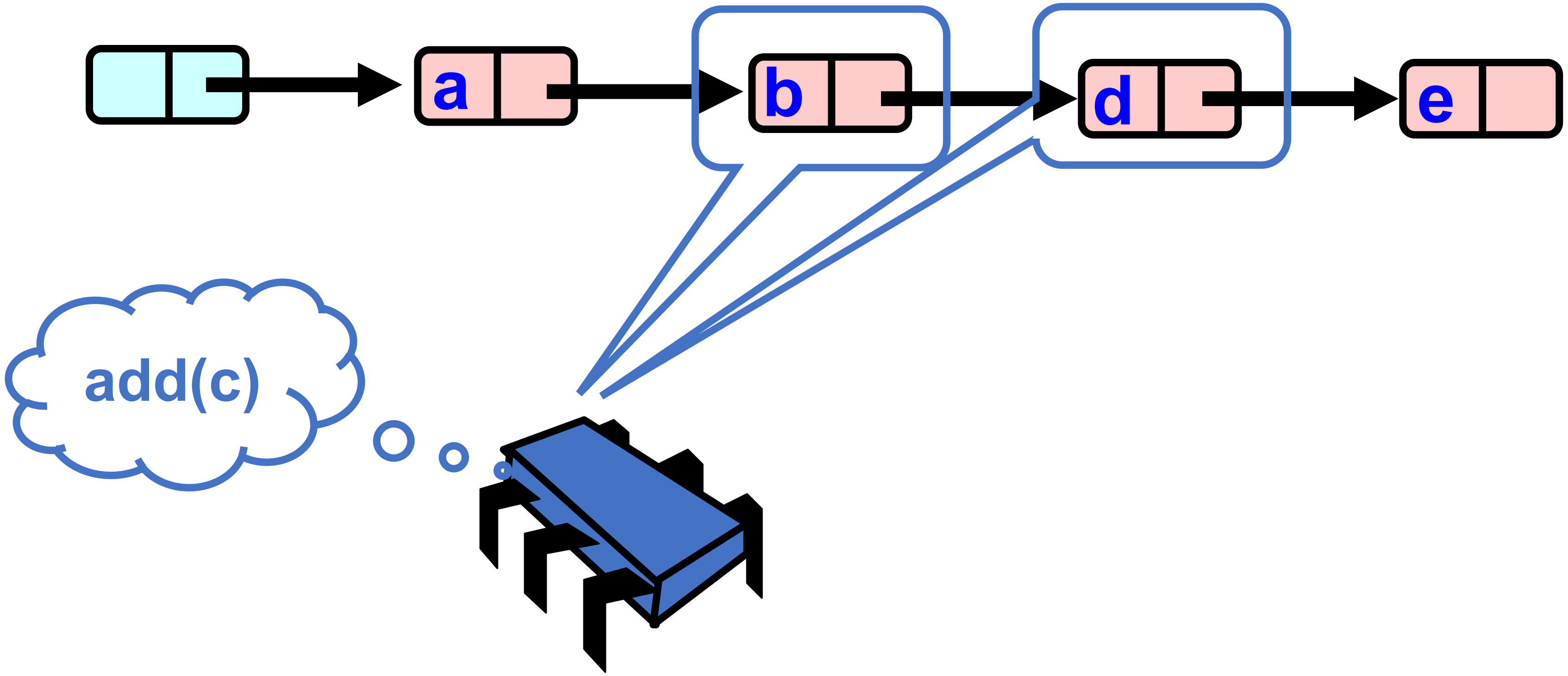
Optimistic: Lock and Load



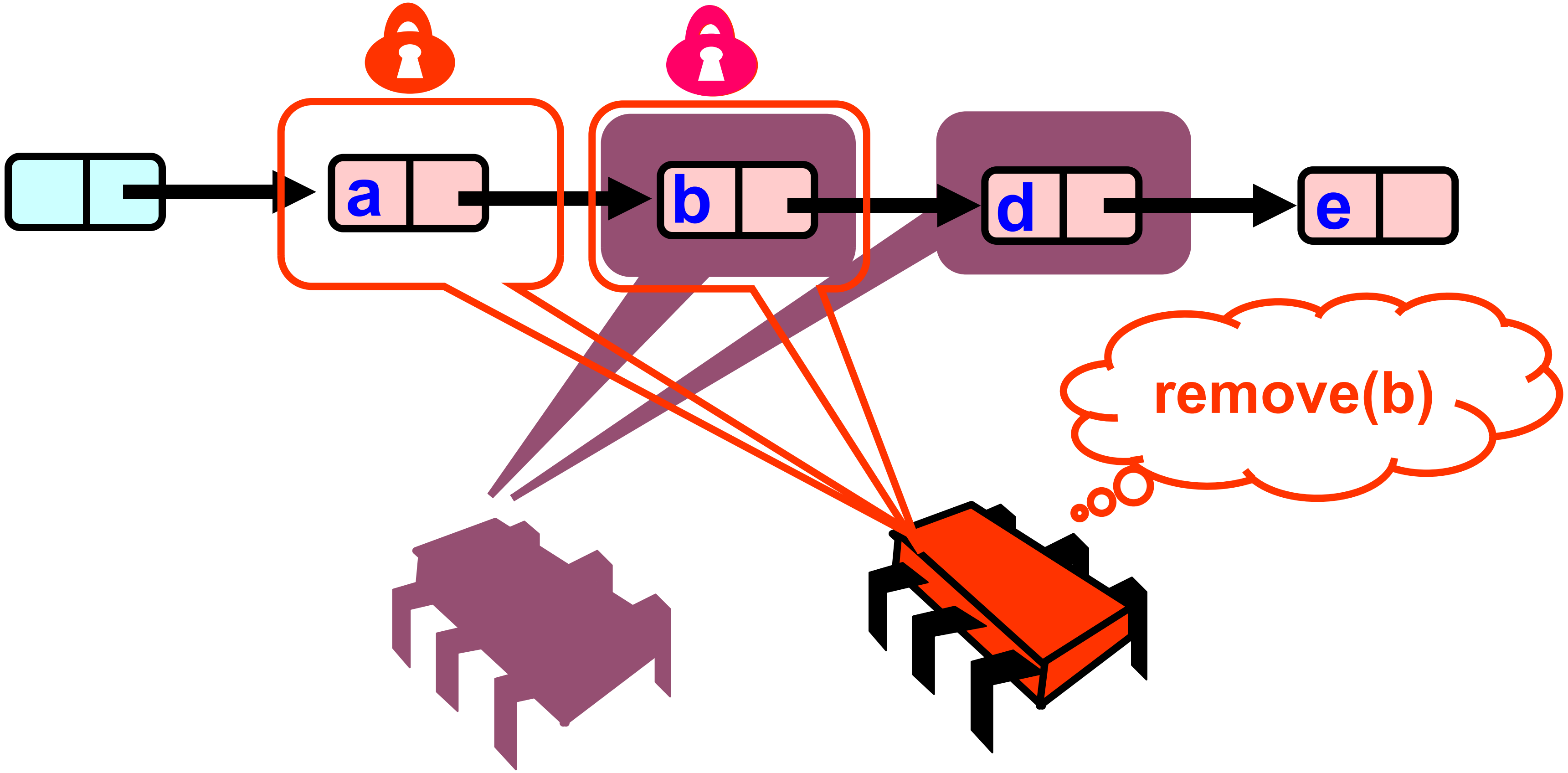
What could go wrong?



What could go wrong?



What could go wrong?



Data conflict!

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

- What do we do?

Data conflict!

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

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

Lock-free reasoning

- We can use atomic variables

Lock-free reasoning

- Default atomic accesses are documented to be sequentially consistent.

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

Lock-free reasoning

- Default atomic accesses are documented to be sequentially consistent.

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

Create an atomic pointer type using C++ templates

Lock-free reasoning

- Default atomic accesses are documented to be sequentially consistent.

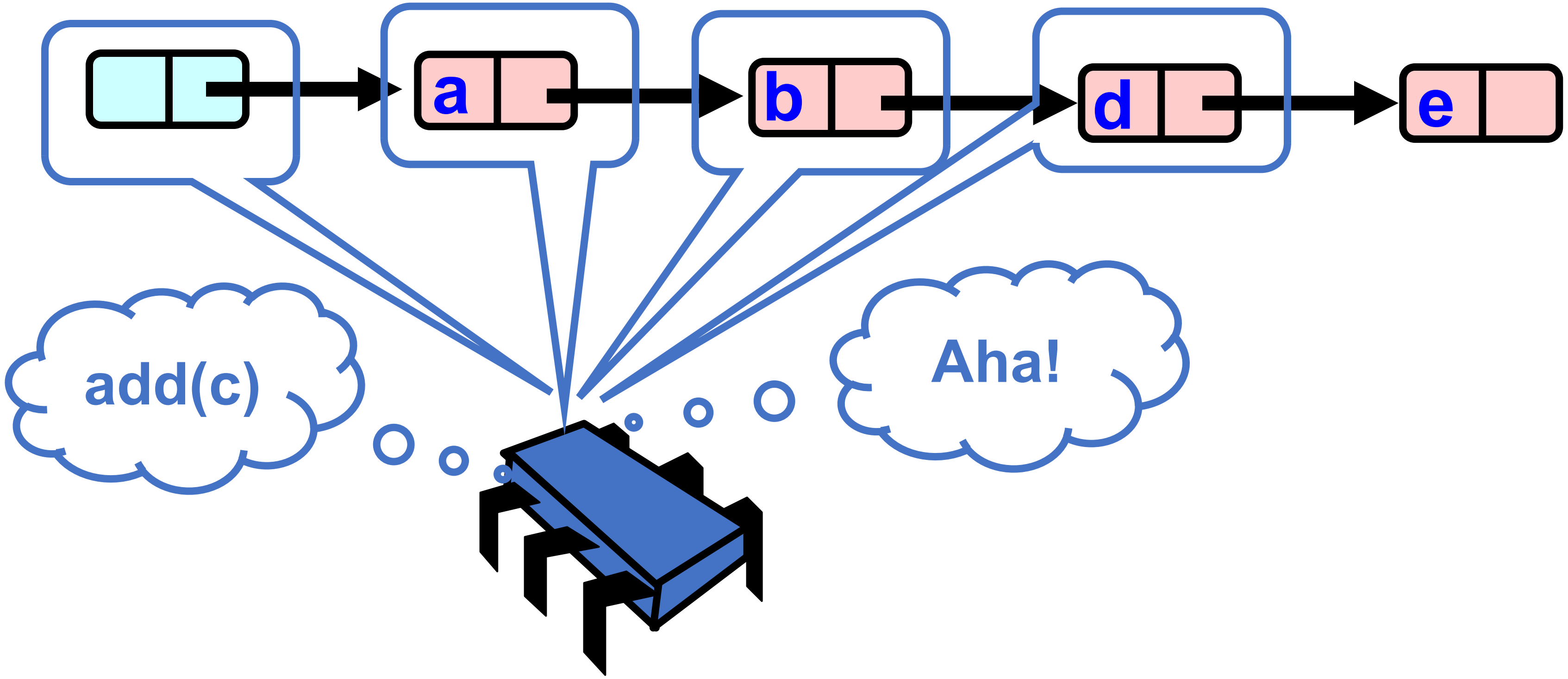
```
void traverse(node *n) {
  while (n->next != NULL) {
    n = n->next;
  }
}
```

Lock-free reasoning

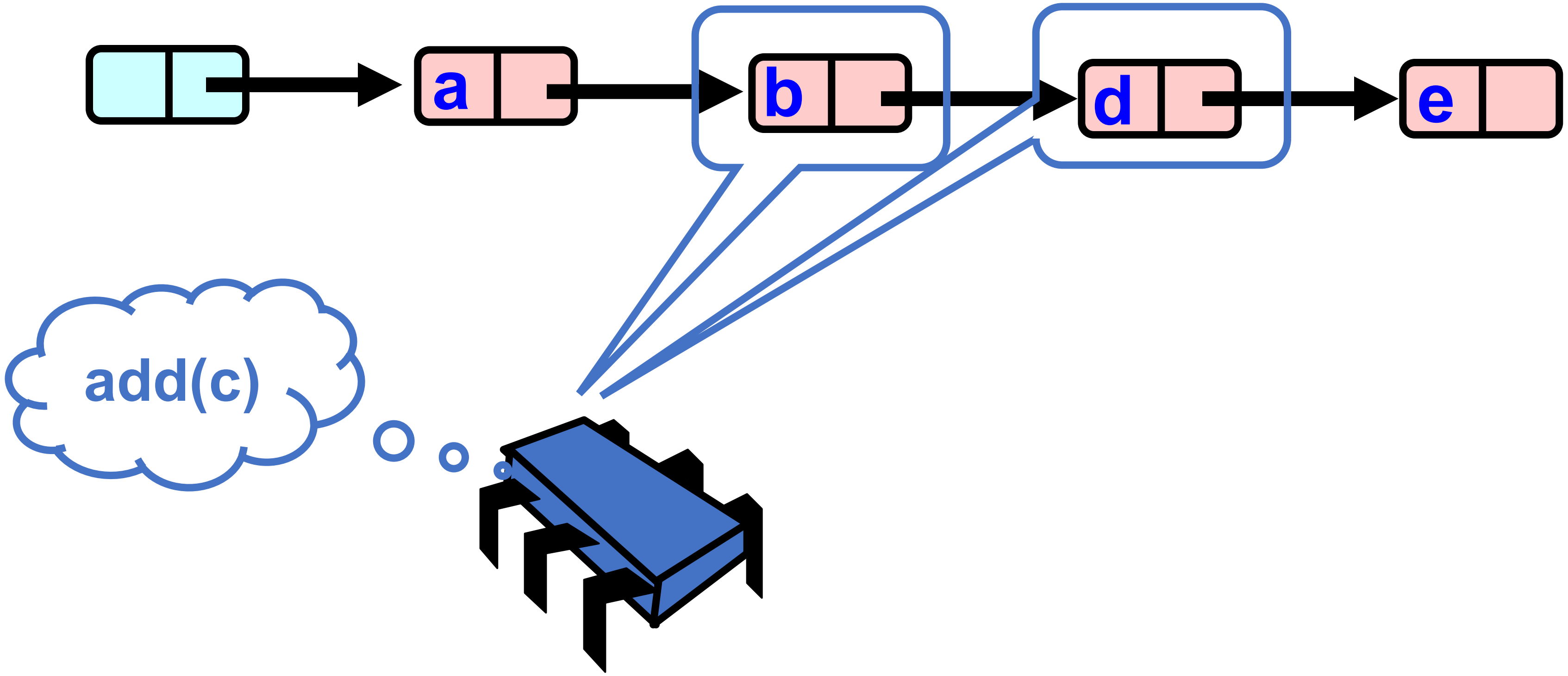
- Default atomic accesses are documented to be sequentially consistent.

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

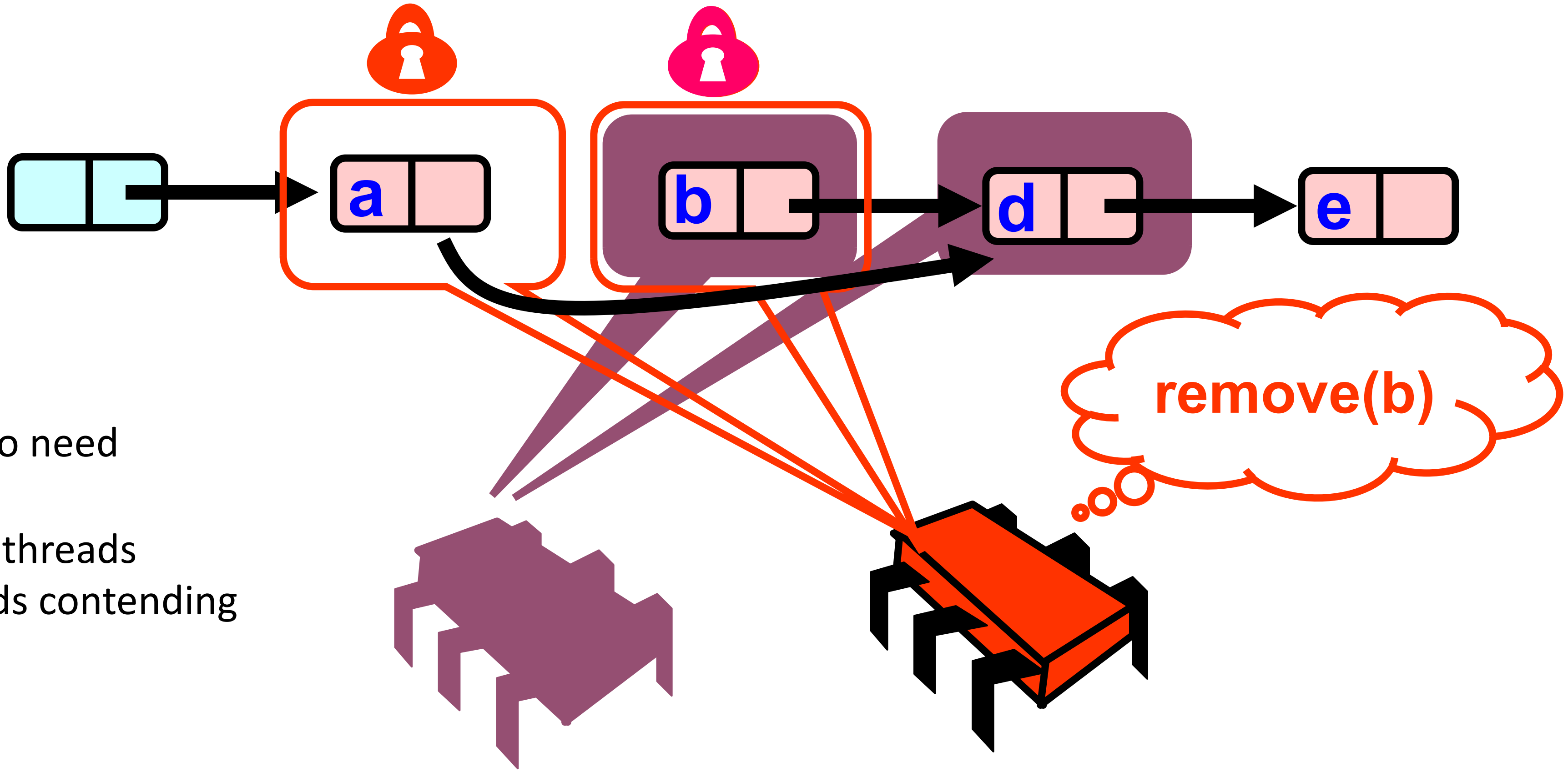
What could go wrong?



What could go wrong?

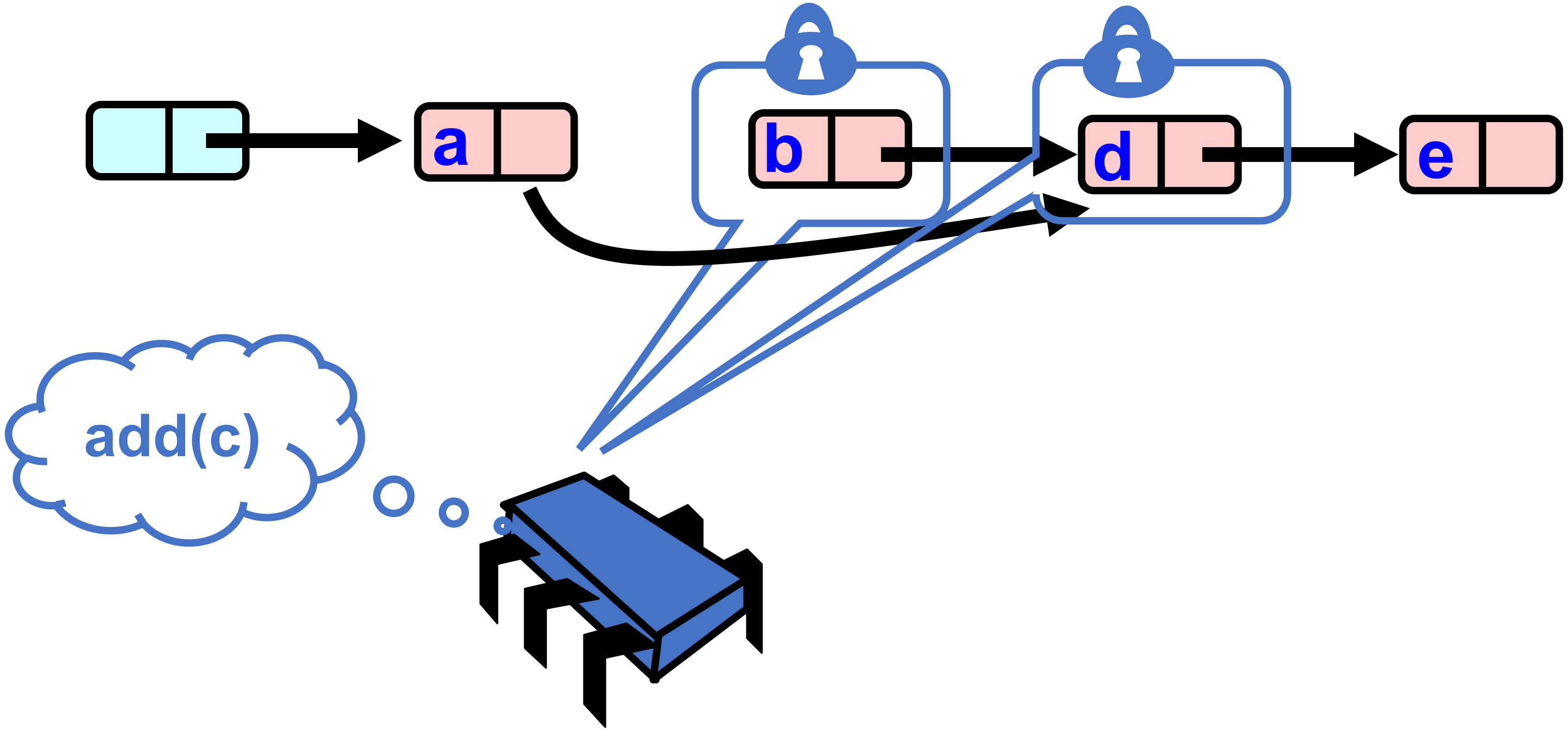


What could go wrong?

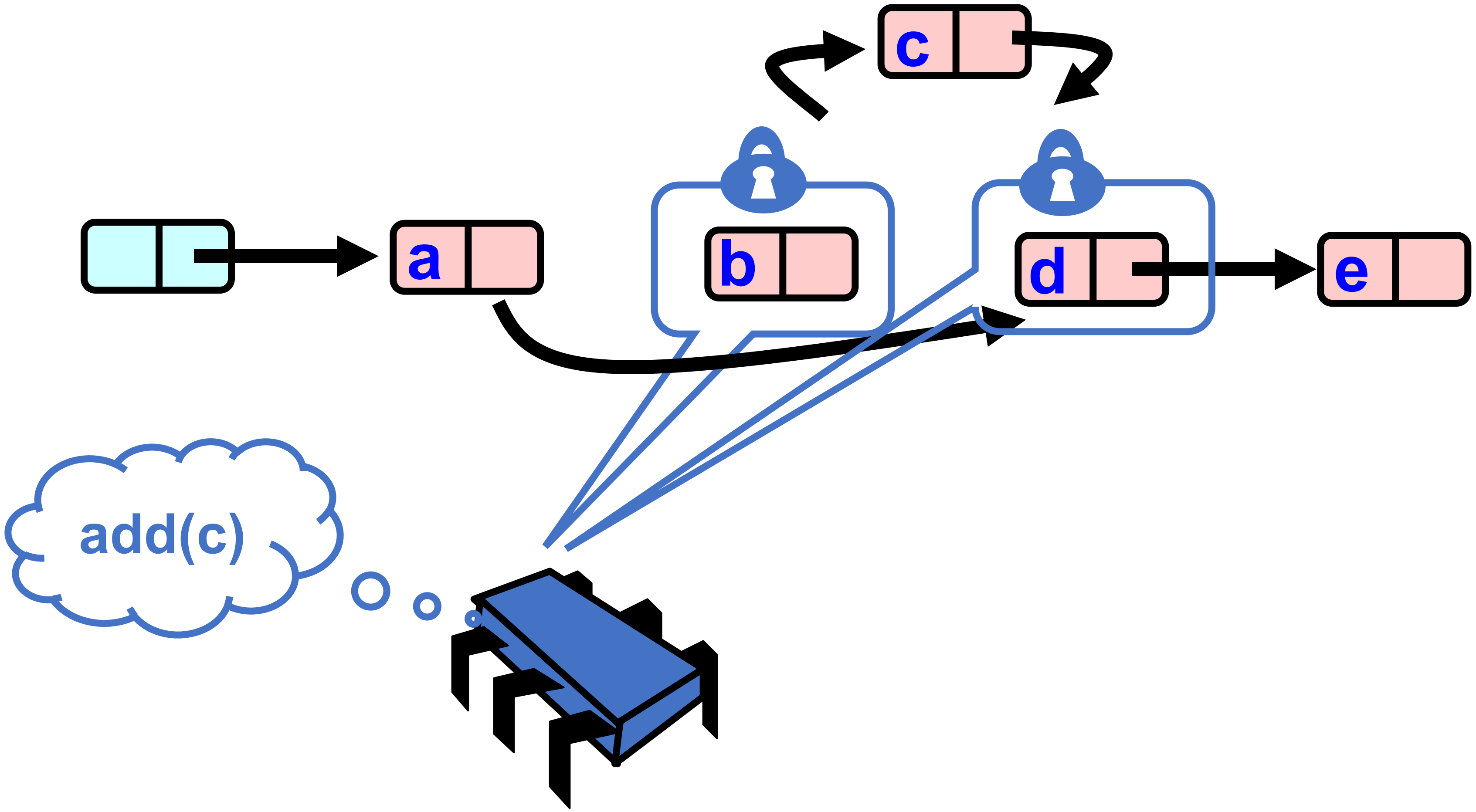


No more data conflict, but we do need to reason about interleavings and threads concurrent threads contending for values.

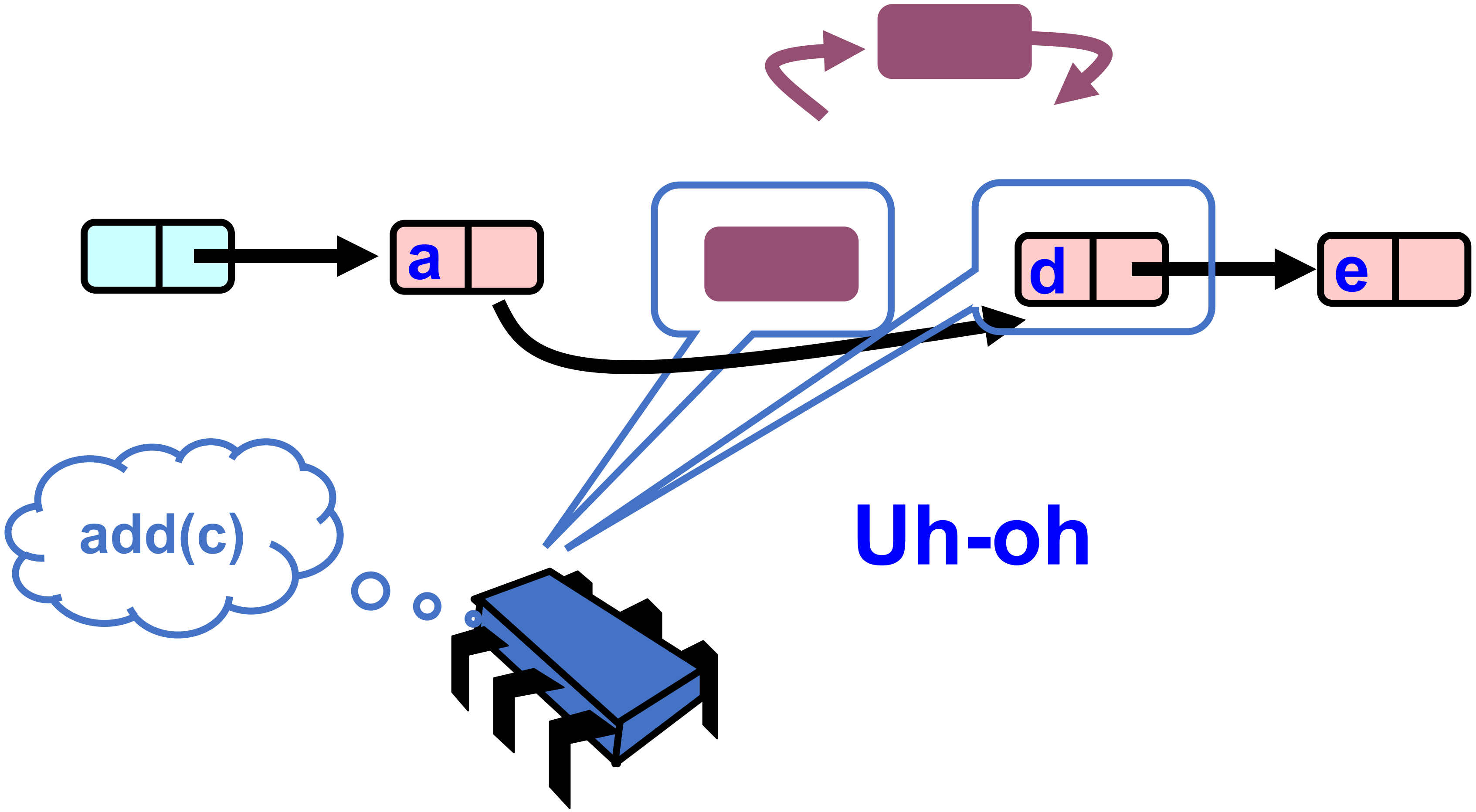
What could go wrong?



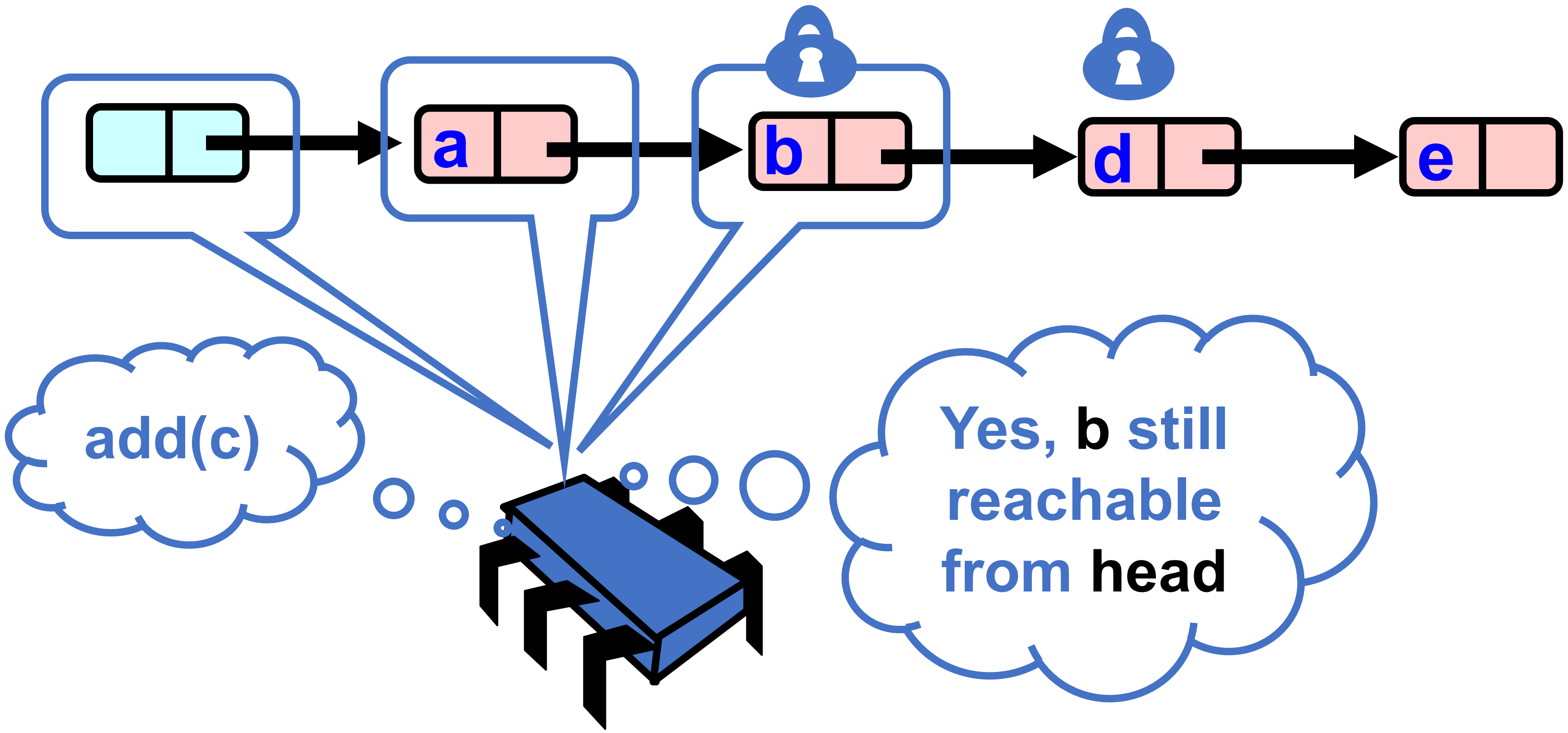
What could go wrong?



What could go wrong?



Validate – Part 1



What happens if failure?

- Ideas?

What happens if failure?

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

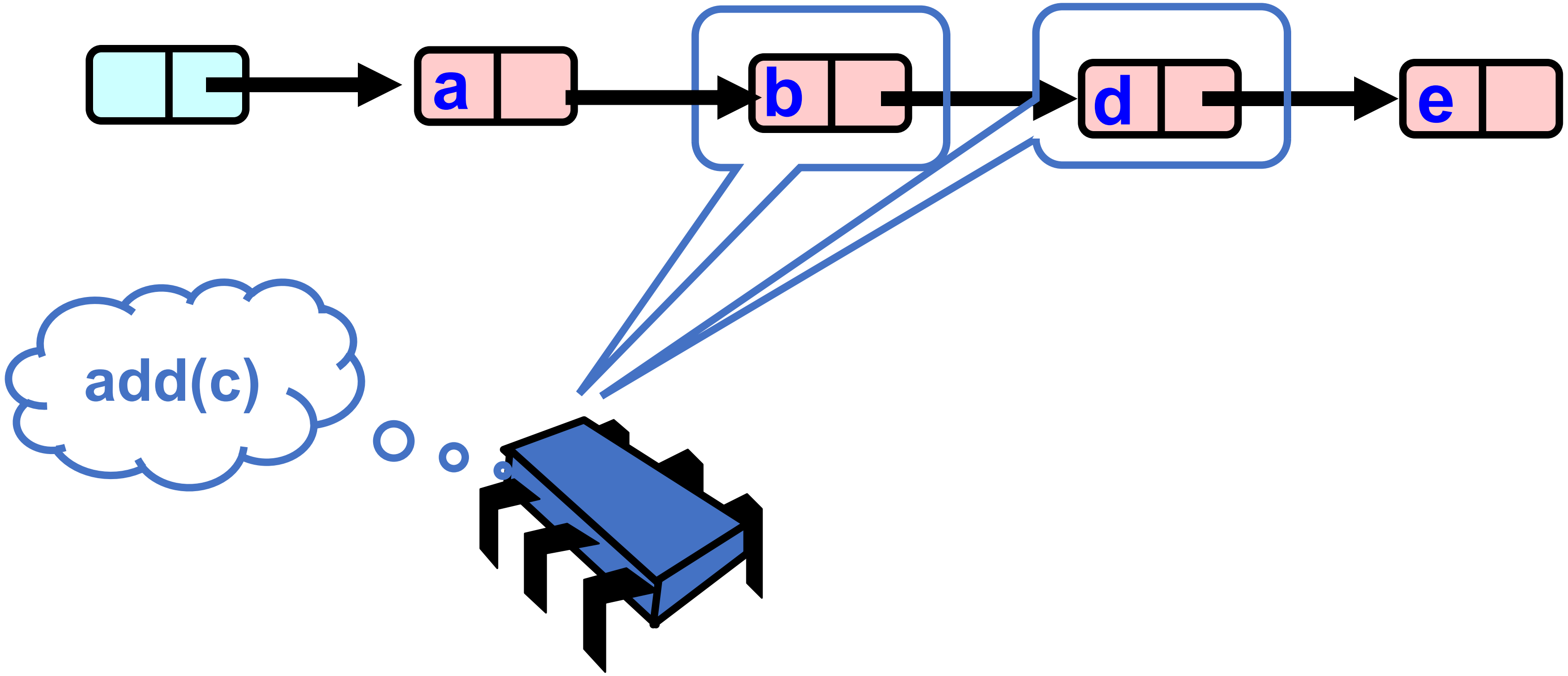
What happens if failure?

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

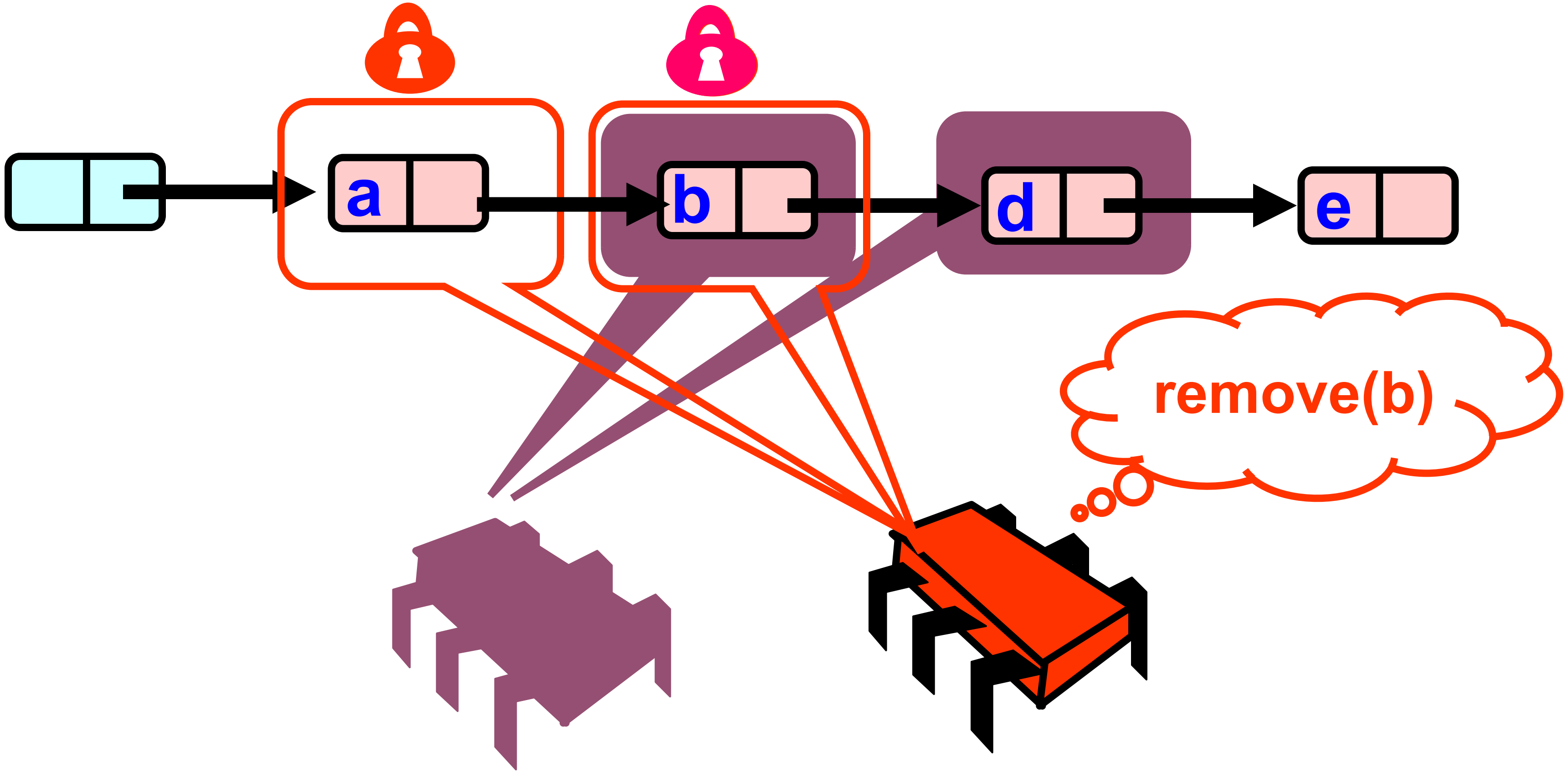
- Private method:
  - try_remove
  - remove loops on try_remove until it succeeds

What about deletion?

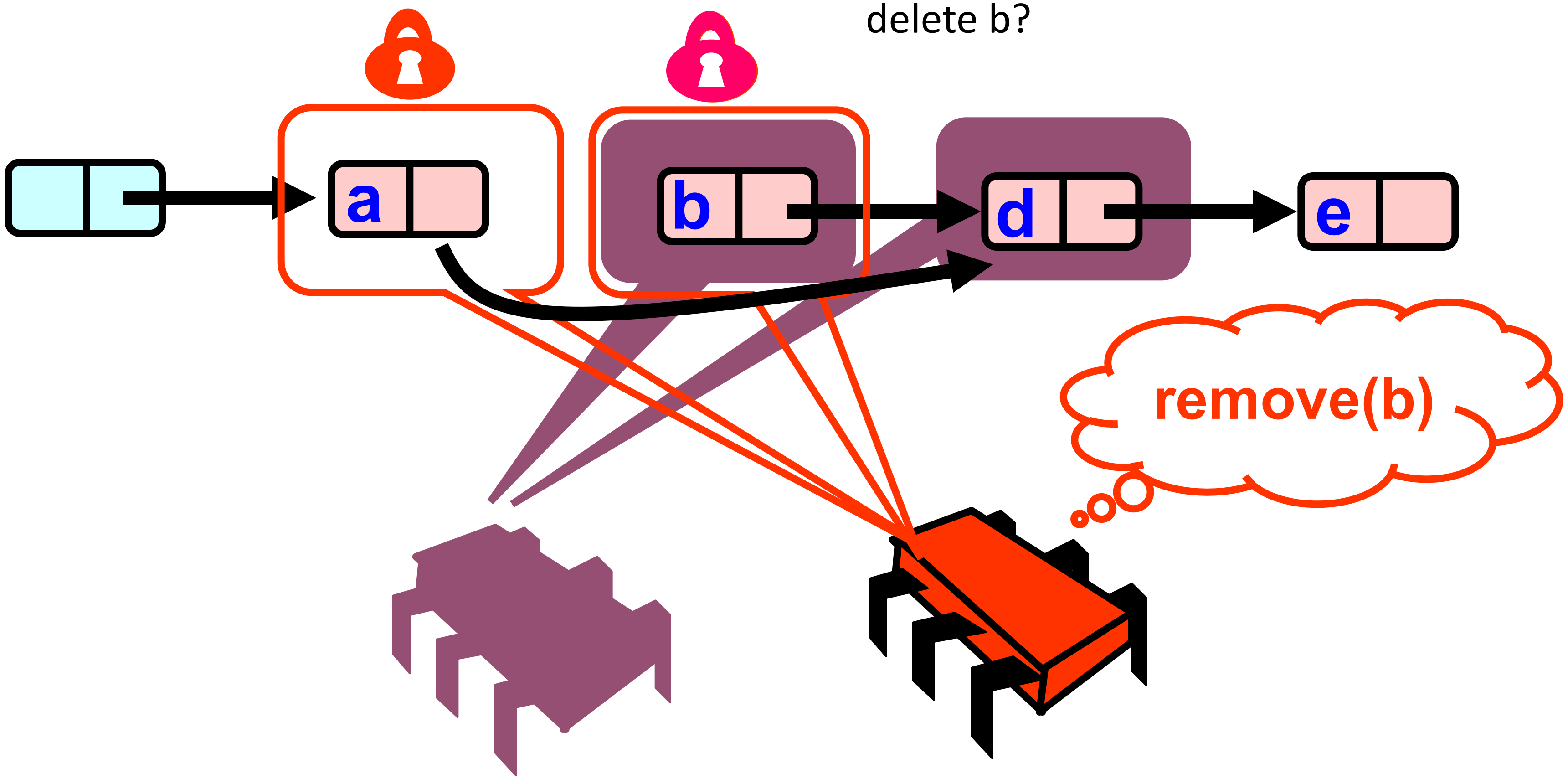
Can threads that remove a node delete it?



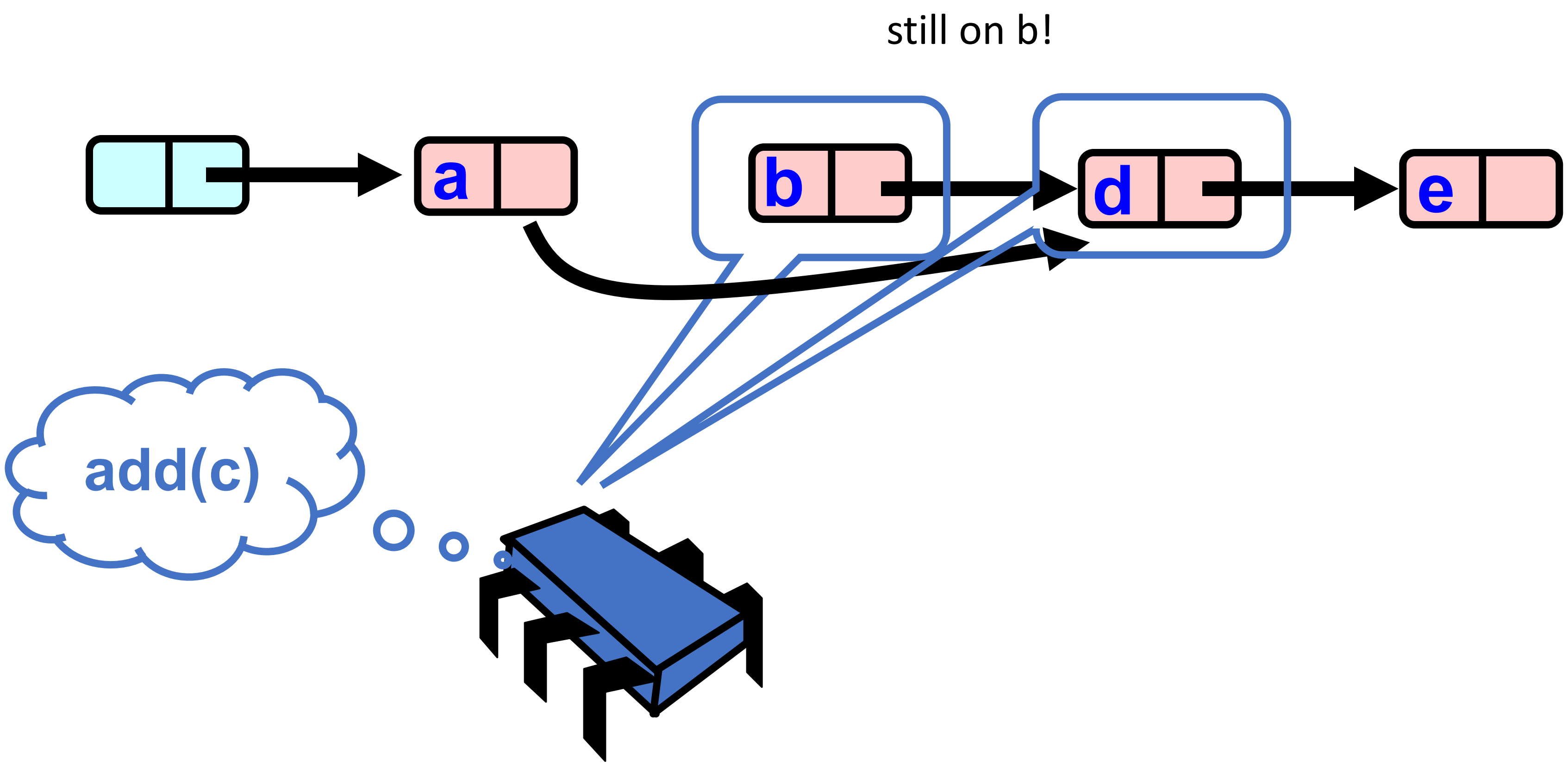
Can threads that remove a node delete it?



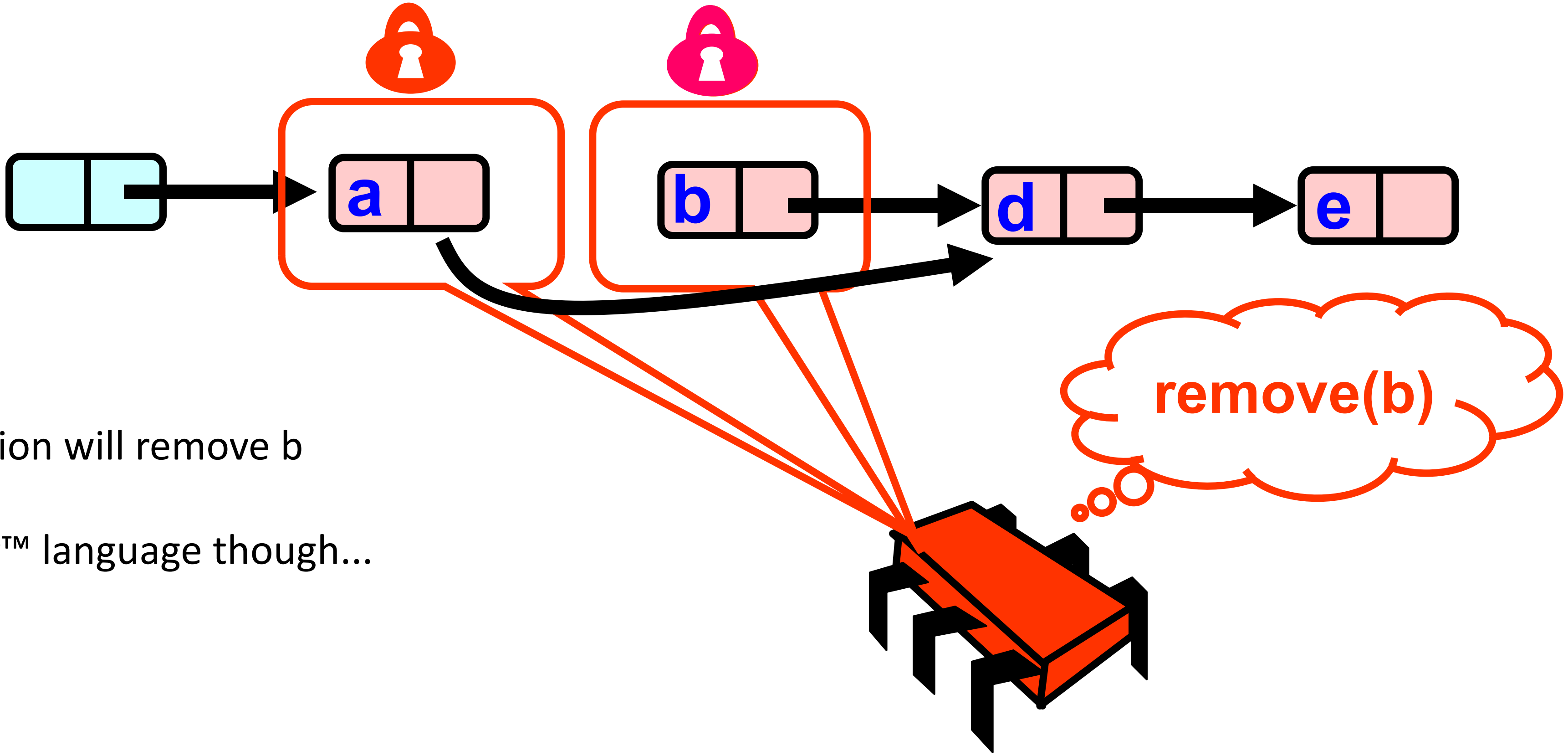
Can threads that remove a node delete it?



Can threads that remove a node delete it?



Our own garbage collector



Java's garbage collection will remove b

We are using a better™ language though...

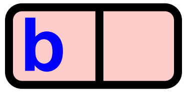
maintain a list to delete:

Our own garbage collector

Java’s garbage collection will remove b

We are using a better™ language though...

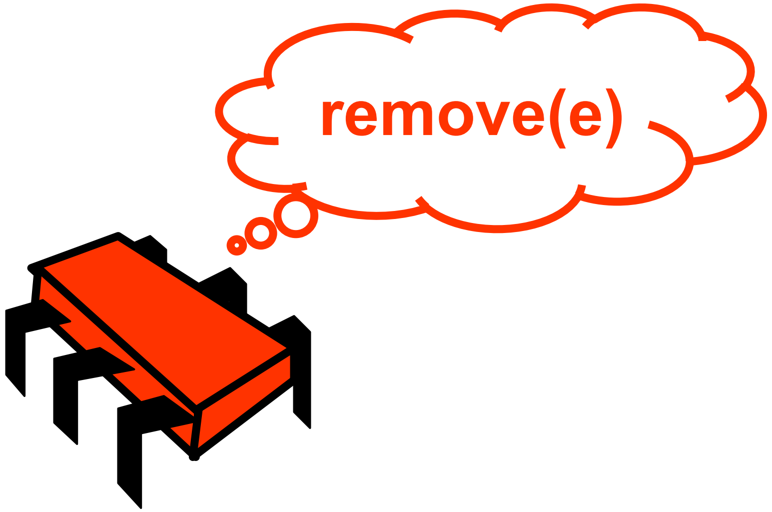
maintain a list to delete:



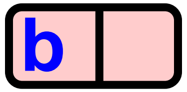
Our own garbage collector

Java's garbage collection will remove b

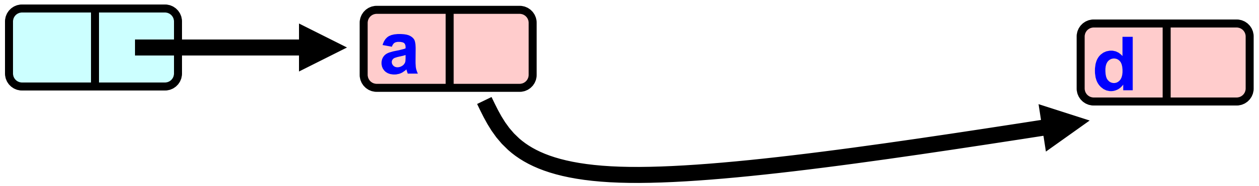
We are using a better™ language though...



maintain a list to delete:

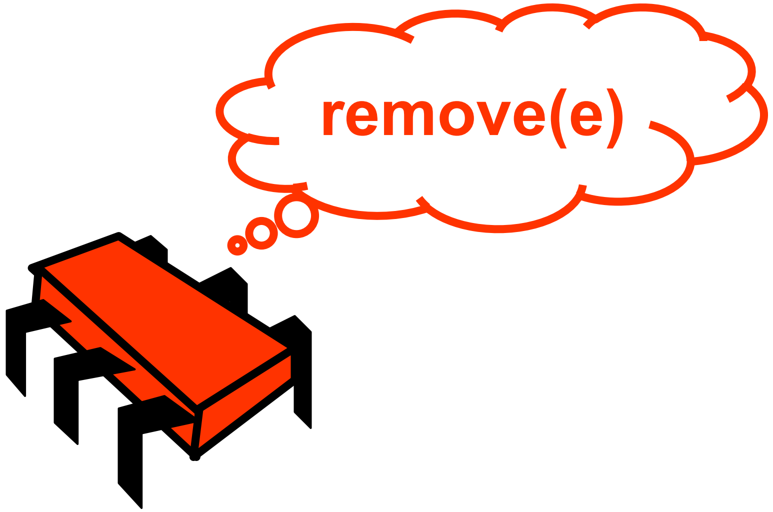


Our own garbage collector

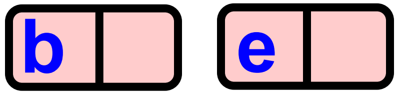


Java’s garbage collection will remove b

We are using a better™ language though...

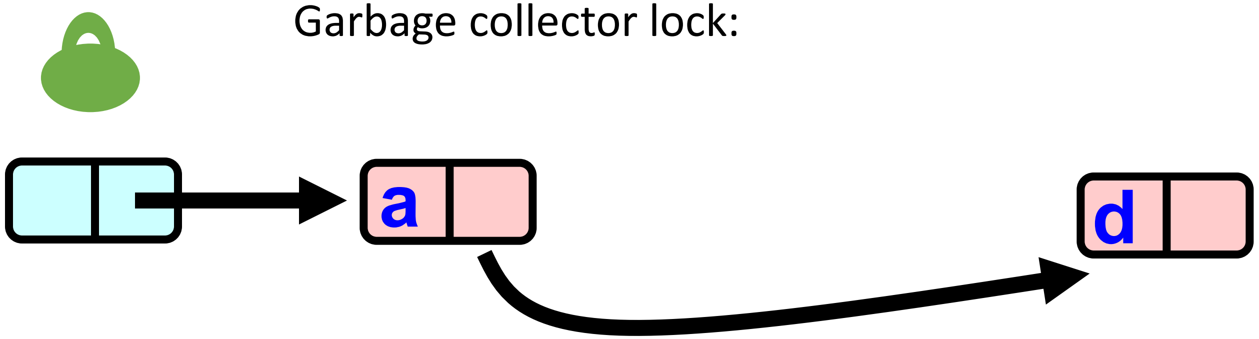


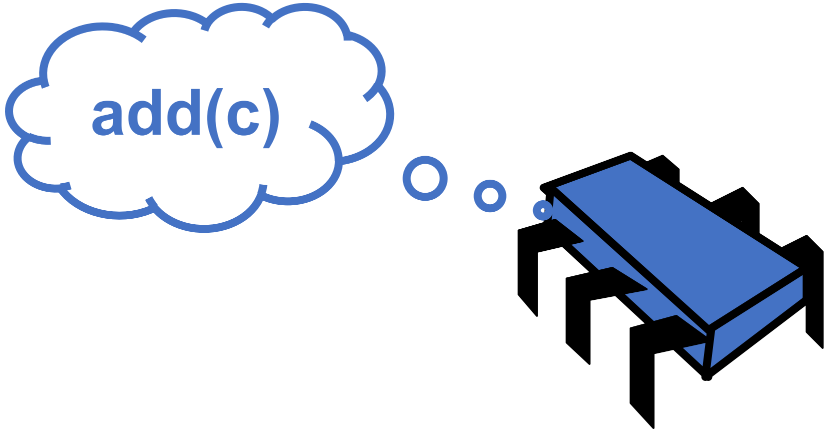
maintain a list to delete:



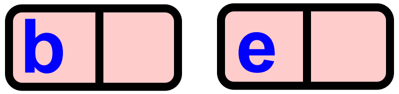
Our own garbage collector

Garbage collector lock:

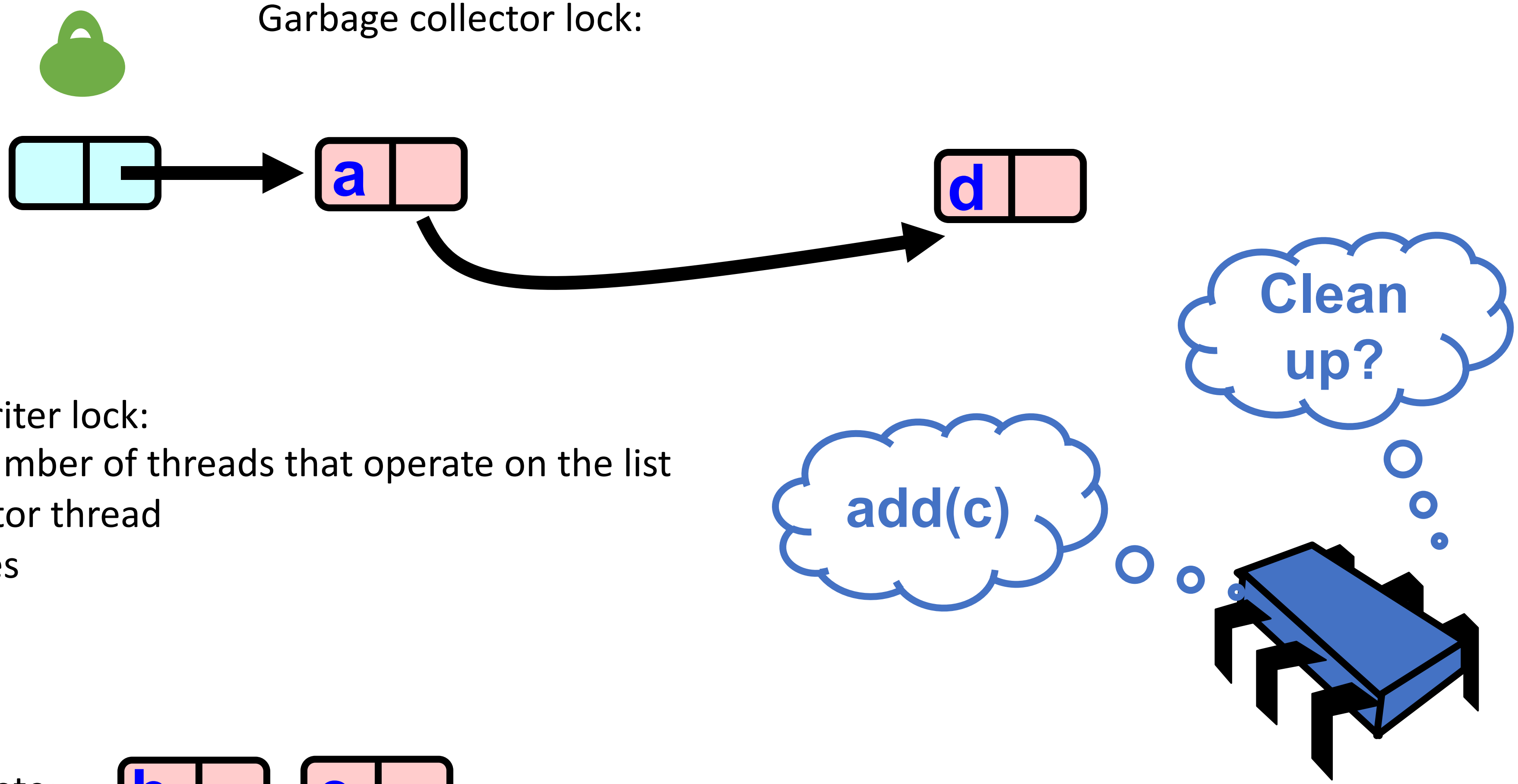




maintain a list to delete:

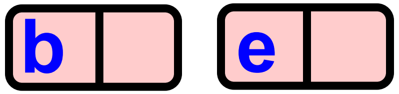


Our own garbage collector


Garbage collector lock:

Similar to a reader/writer lock:
Allows an arbitrary number of threads that operate on the list
Only 1 garbage collector thread
Erases the list of nodes

maintain a list to delete:



Garbage collector lock

- Many strategies!
  - A big research area ~10 years ago

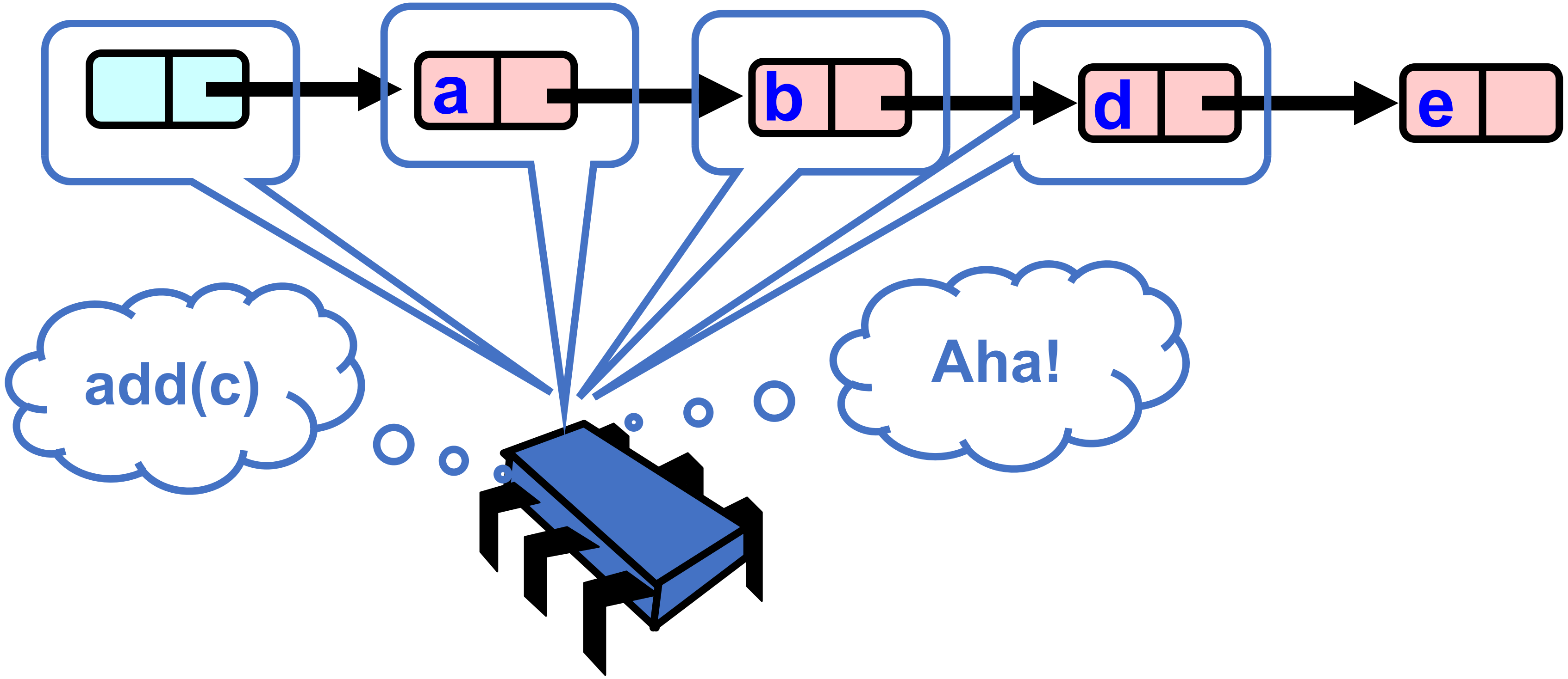
- **Strat 1:** Threads always try once to take the garbage collector lock:
  - if failed, no worries, the next operation will get a chance
  - if succeeded, then there was no contention
  - can starve garbage collection

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

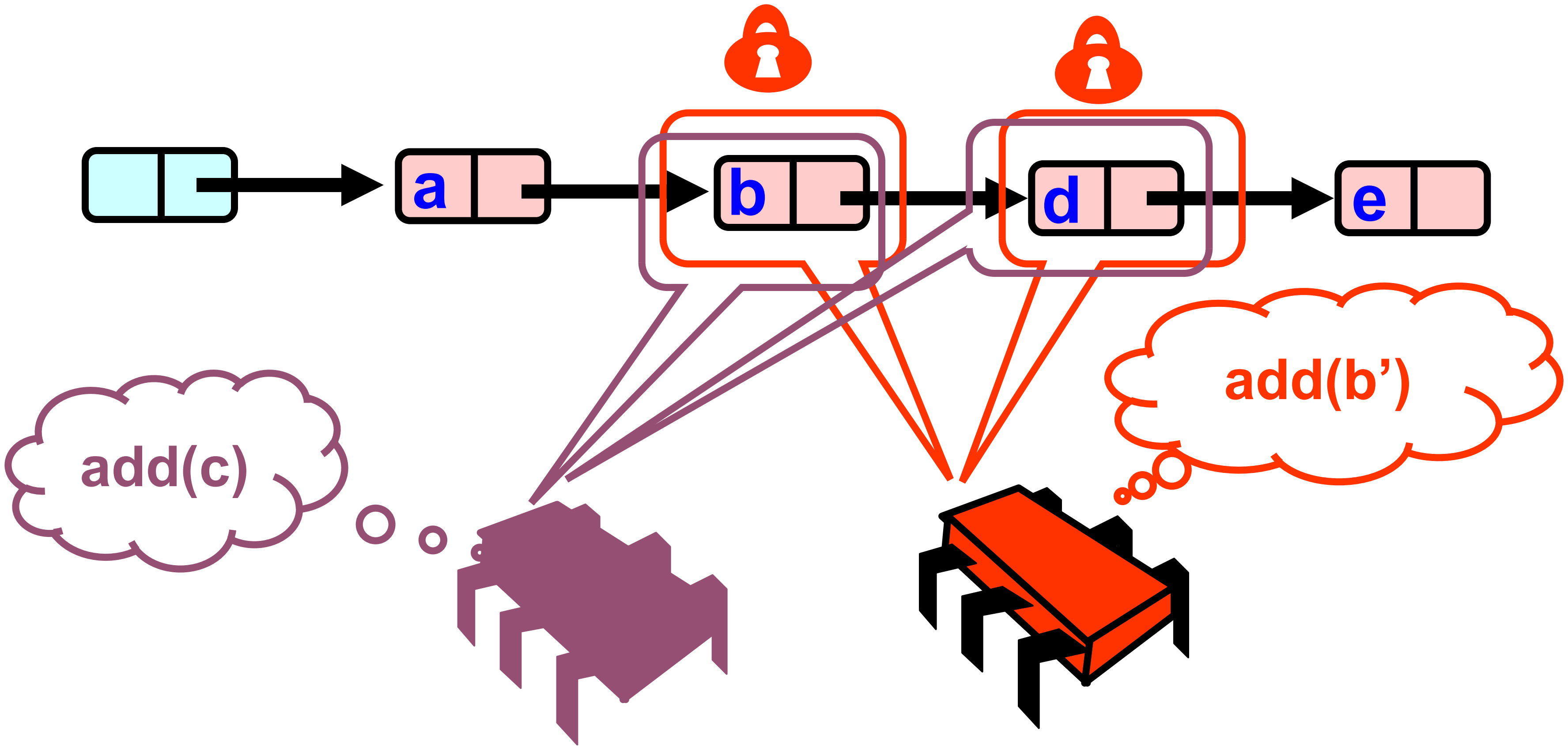
Back to the linked list

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

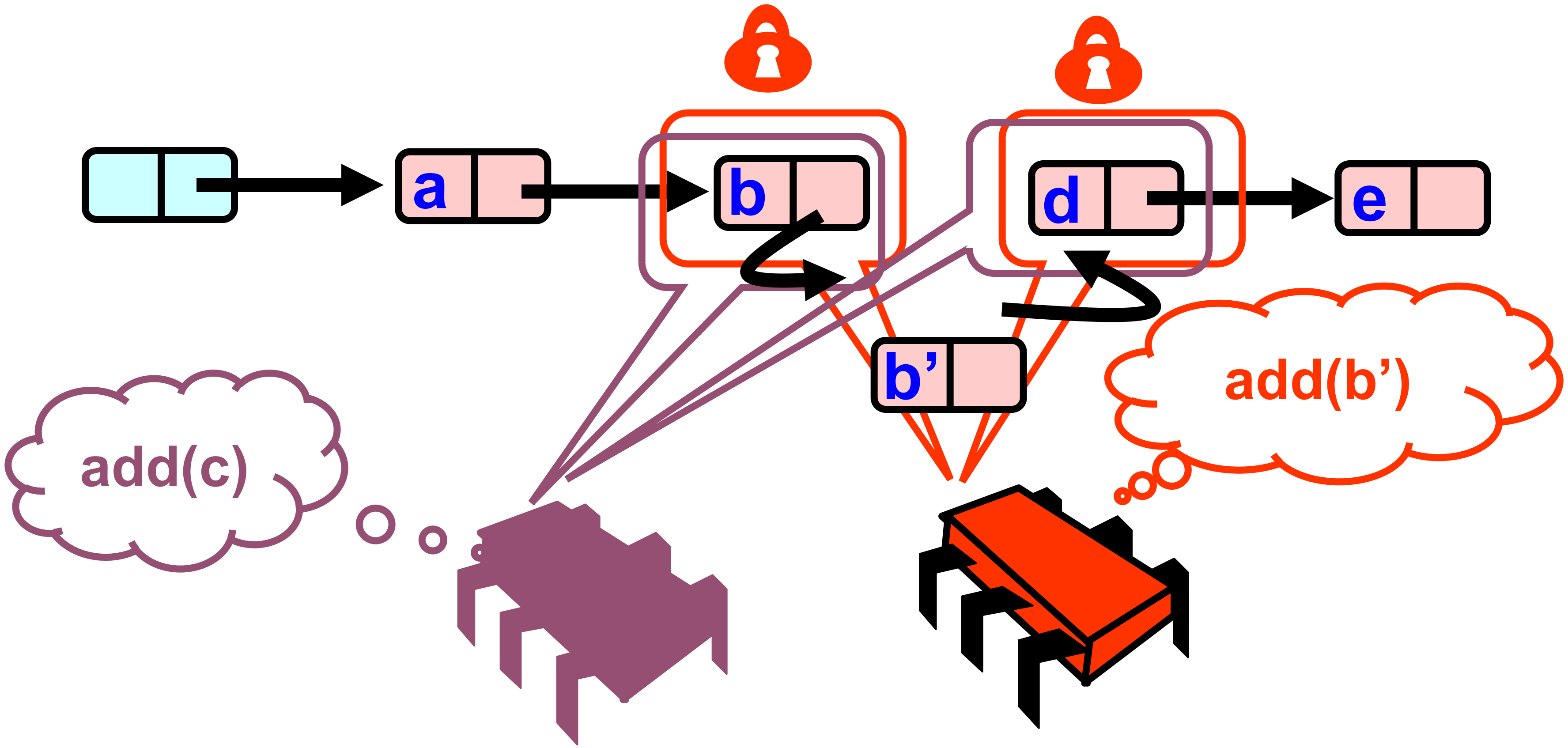
What Else Could Go Wrong?



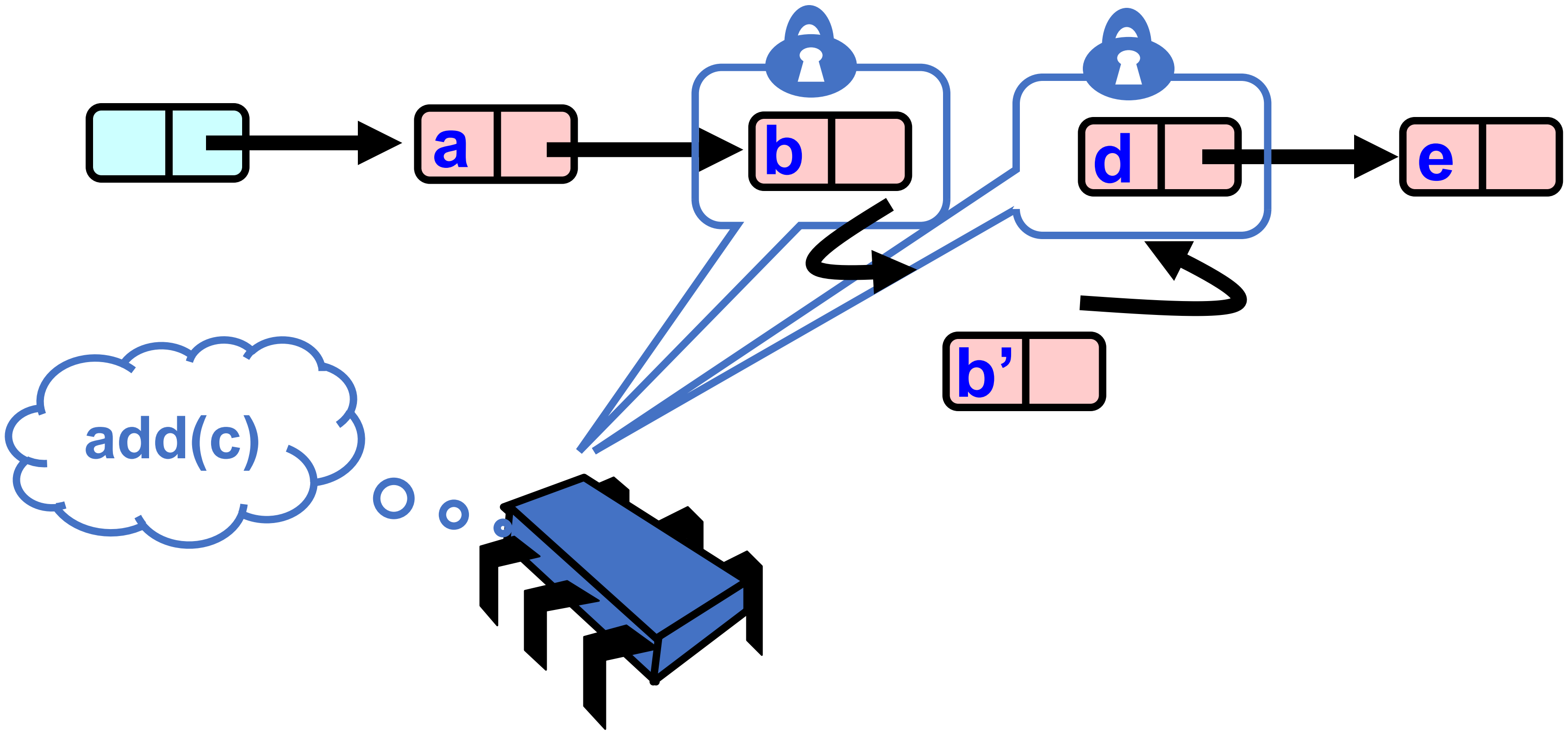
What Else Coould Go Wrong?



What Else Coould Go Wrong?

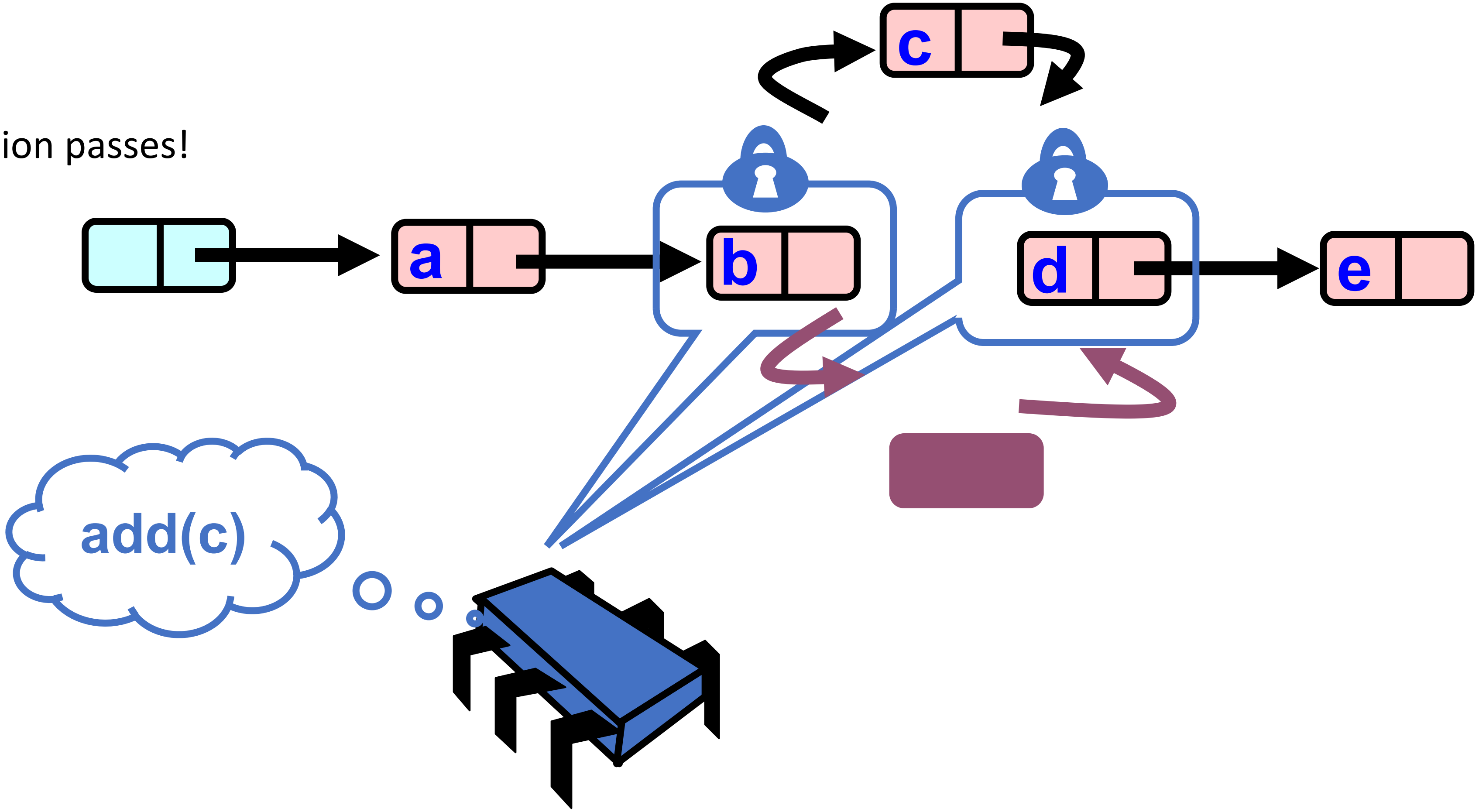


What Else Could Go Wrong?

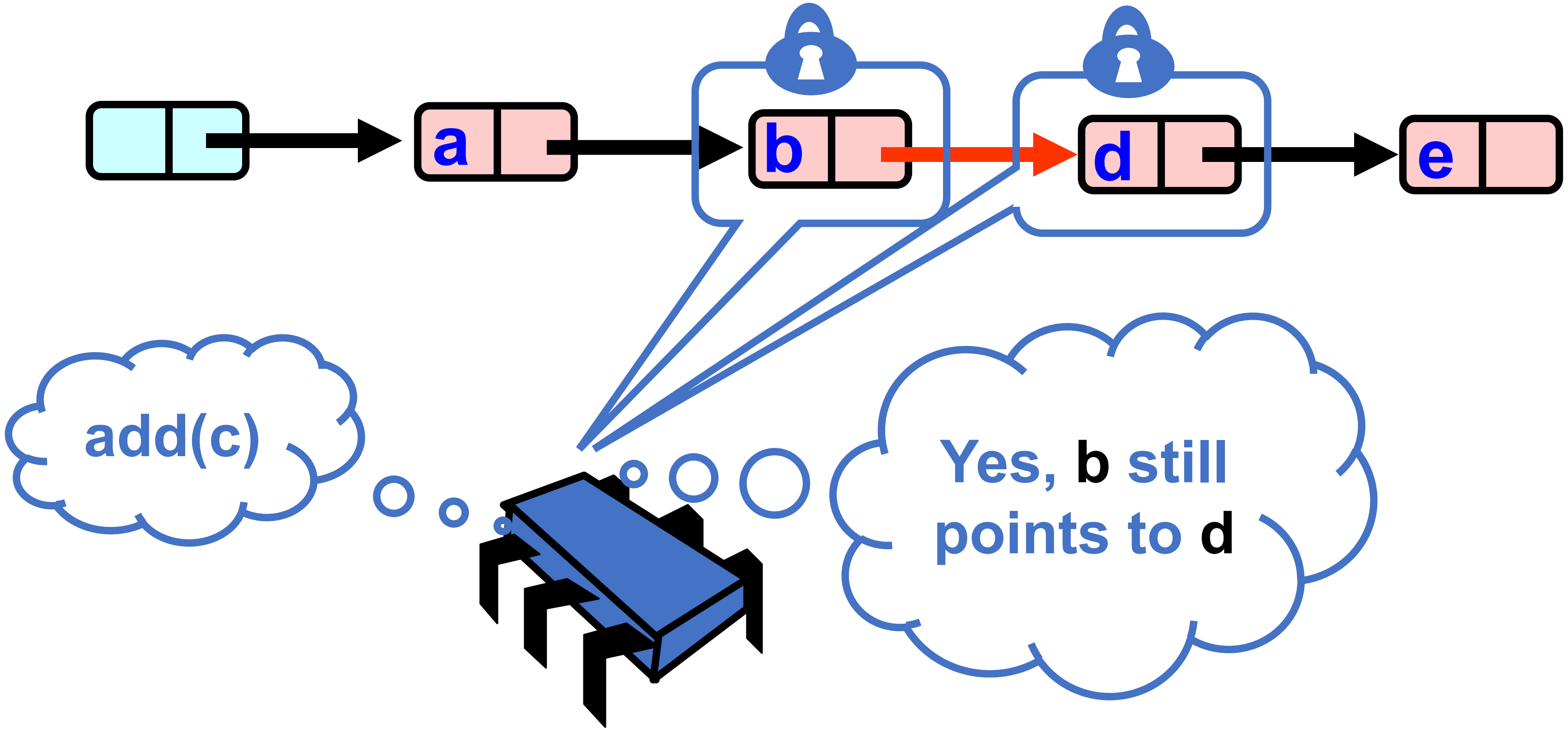


What Else Could Go Wrong?

Validation passes!



Validate Part 2
(while holding locks)



Summary

- We traverse without lock
  - Traversal may access nodes that are locked
  - Its okay because we have atomic pointers!

- We might traverse deleted nodes
  - Its okay because we validate after we obtain locks
  - Two validations:
    - our node is still reachable (it was not deleted)
    - Our insertion point is still valid (no thread has inserted in the meantime)

- We don’t actually free node memory, but we put them in a list to be freed later

Enjoy your weekend!

- On Monday: making the list lock-free!
  - One extra lecture on module 3

- I really should be feeling better by Monday

- Hopefully you have started the midterm!

- Get started on HW 2