departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Locking Strategies

lecture 16 (2020-04-22)

Master in Computer Science and Engineering

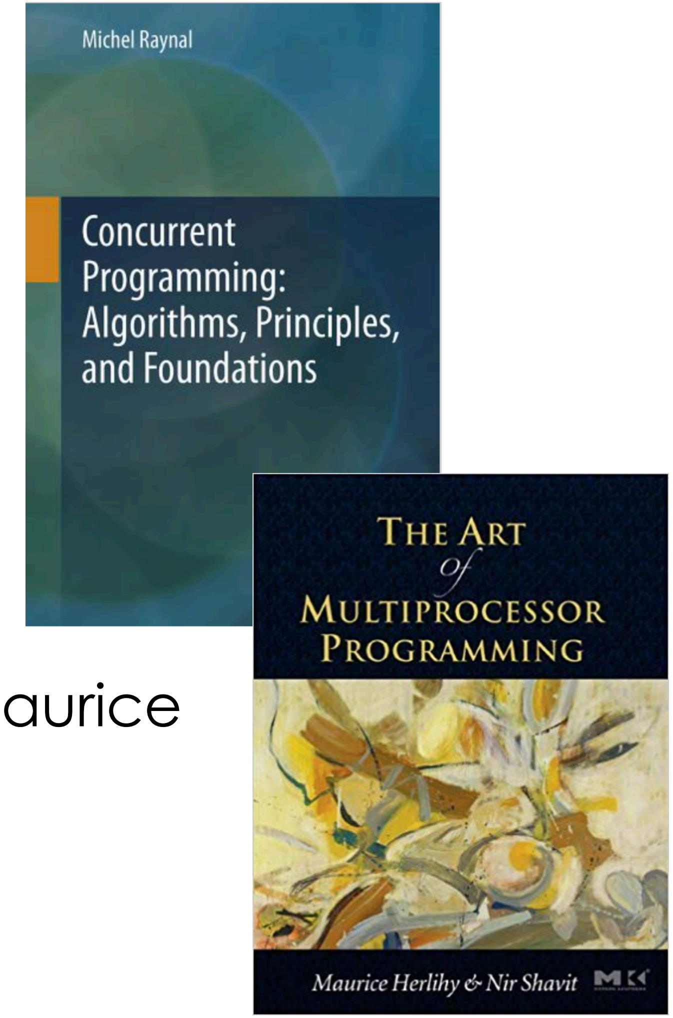
— Concurrency and Parallelism / 2019-20 —

João Lourenço <joao.lourenco@fct.unl.pt>

Locking Strategies

- **Contents:**
  - Coarse-Grained Synchronization
  - Fine-Grained Synchronization

- **Reading list:**
  - Chapter 5 of the Textbook
  - Chapter 9 (9.1-9.5) of “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit *(available at clip)*


Michel Raynal
Concurrent Programming: Algorithms, Principles, and Foundations
THE ART of MULTIPROCESSOR PROGRAMMING
Maurice Herlihy & Nir Shavit
MK

Coarse-Grained Synchronization

- Use a single lock…
- Methods are always executed in mutual exclusion
  - Methods never conflict
- Eliminates all the concurrency within the object

Fine-Grained Synchronization

- Instead of using a single lock…
- Split object into multiple independently-synchronized components
- Methods conflict when they access
  - The same component…
  - (And) at the same time!

Linked List

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

Set Interface

- Unordered collection of items
- No duplicates
- Methods

– *add(x)*	put x in set	*true if x was not in the set*
– *remove(x)*	take x out of set	*true if x was in the set*
– *contains(x)*	tests if x in set	*true if x is in the set*

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;
}
```

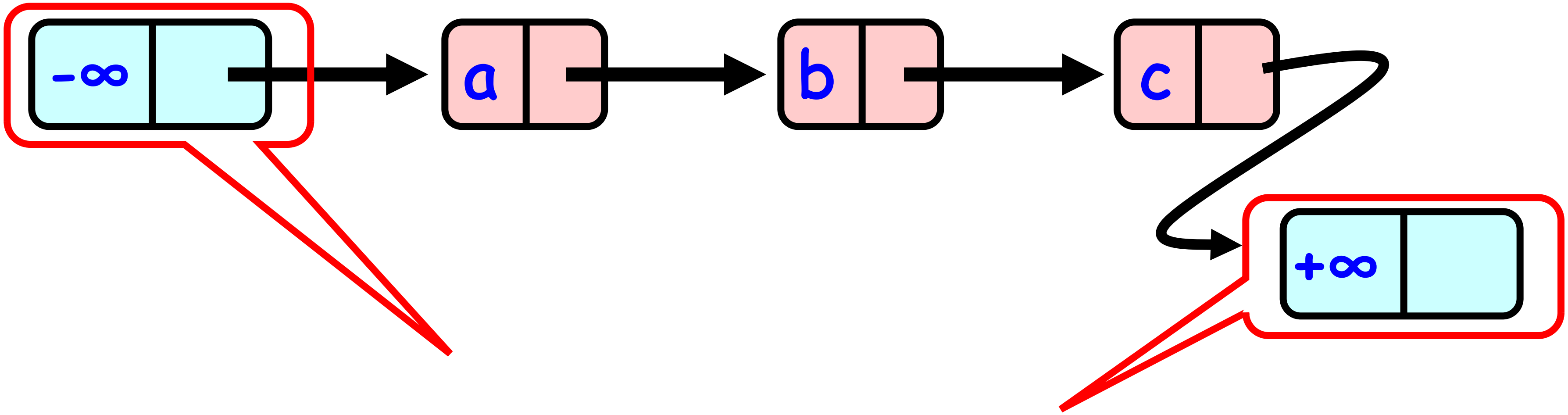
Usually hash code

List Node

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

Reference to next node

The List-Based Set



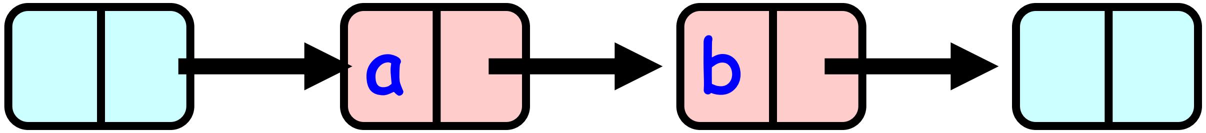
Sorted with Sentinel nodes
(min & max possible keys)

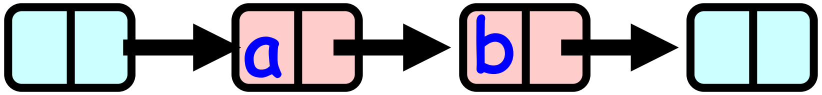
Reasoning about Concurrent Objects

- Invariant
  - Property that always holds
  - Established because
    - True when object is created
    - Truth preserved by each method
      - Each step of each method
- Assertion
  - Property valid in a specific location (code line)
  - Weaker than invariants, but much easier to define

Abstract Data Types

- Concrete representation

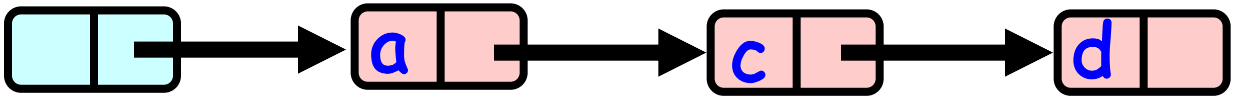


- S() = {a, b}

- Abstract Type
  - {a, b}

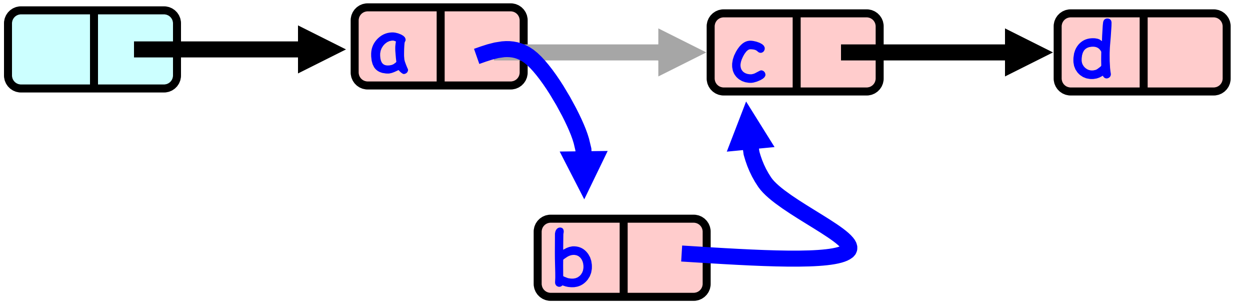
Sequential List Based Set

Add()



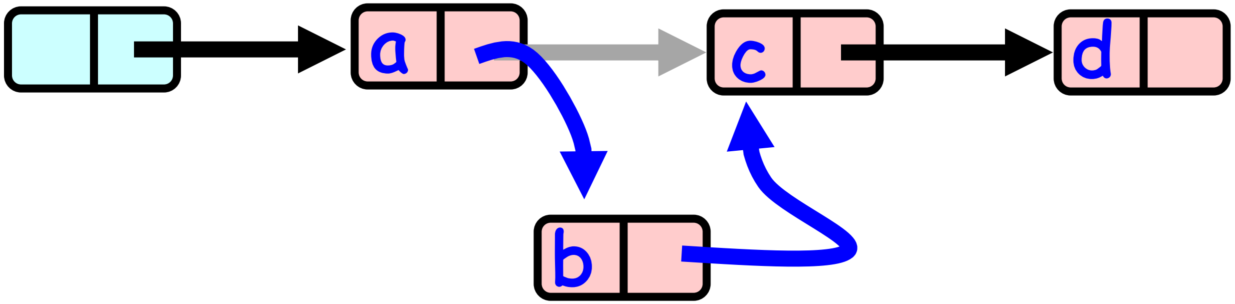
Sequential List Based Set

Add()

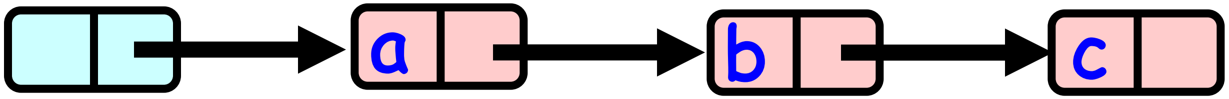


Sequential List Based Set

Add()

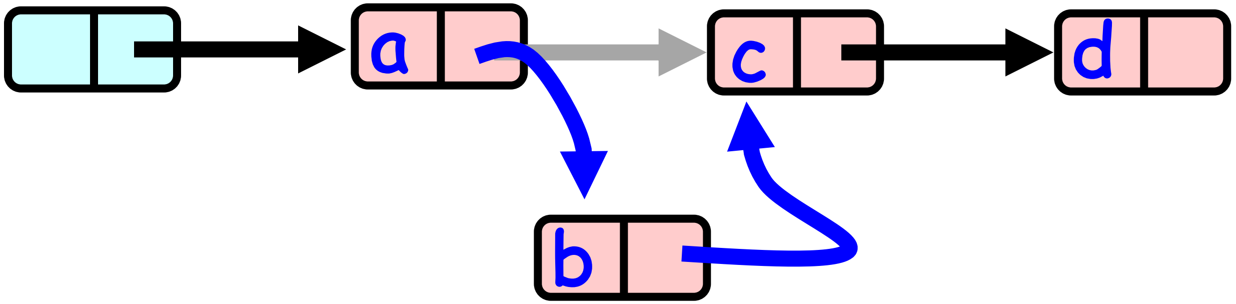


Remove()

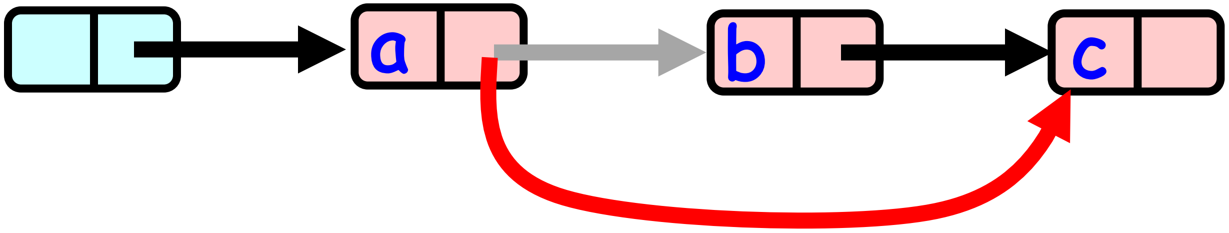


Sequential List Based Set

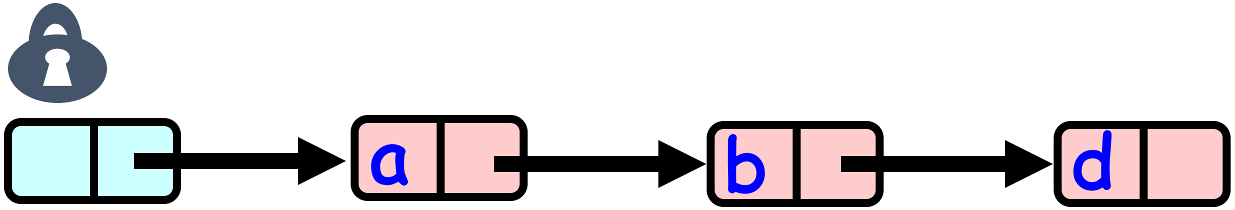
Add()



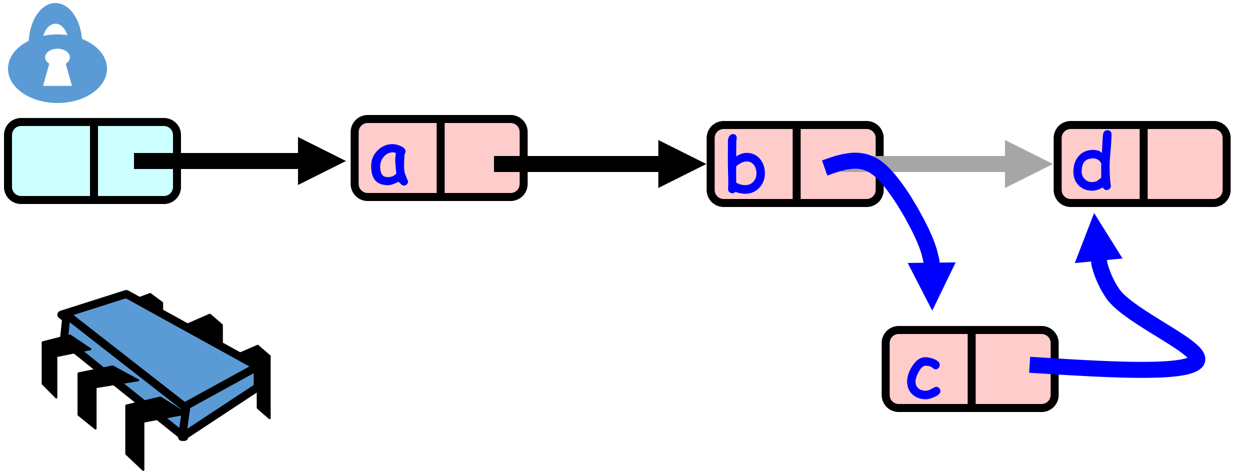
Remove()



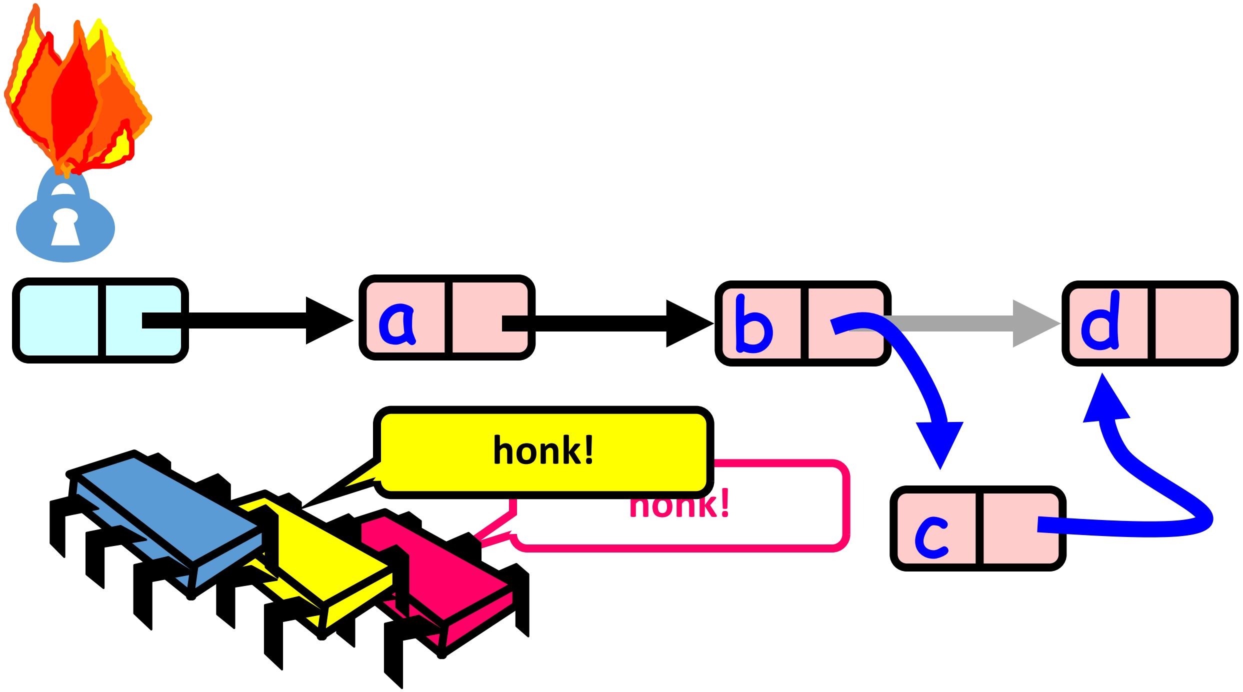
Coarse Grained Locking



Coarse Grained Locking



Coarse Grained Locking



Simple but hotspot + bottleneck

Coarse Grained Locking

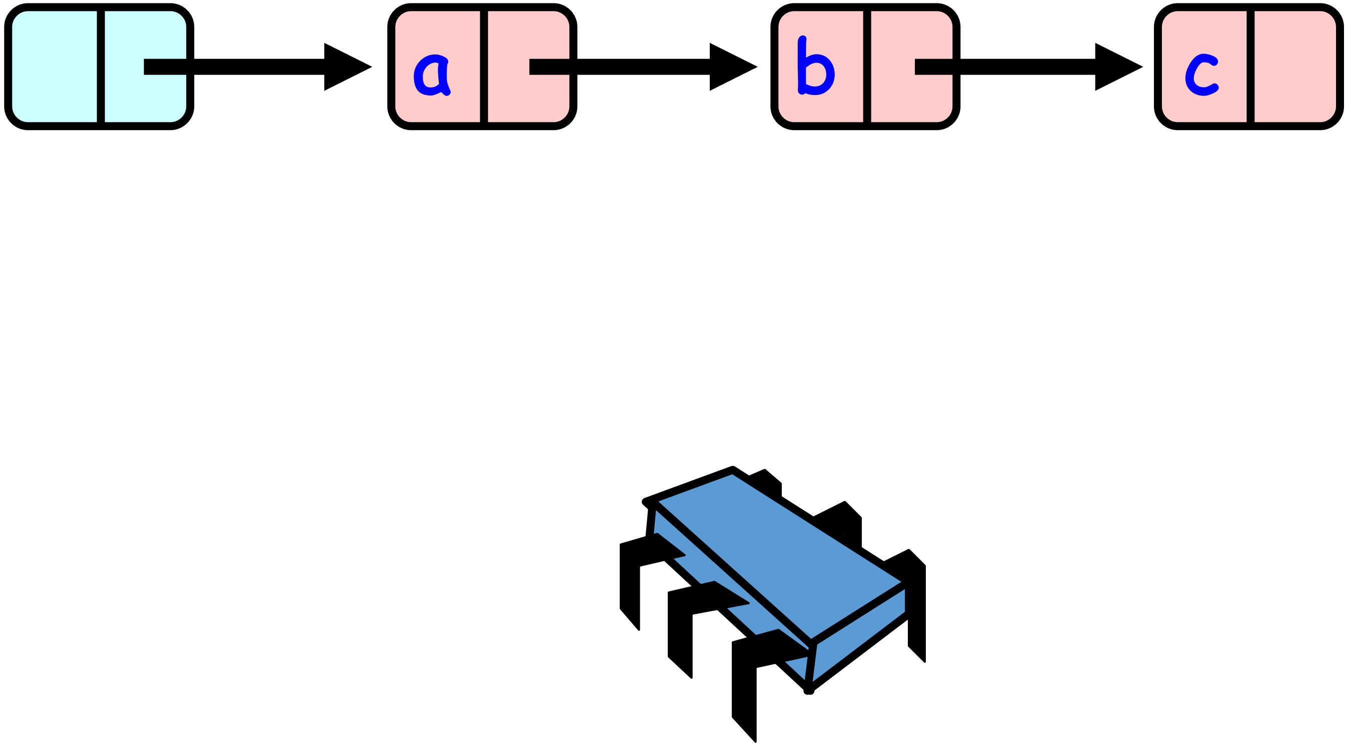
- Easy, same as synchronized methods
  - "One lock to rule them all ..."
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention

Fine-grained Locking

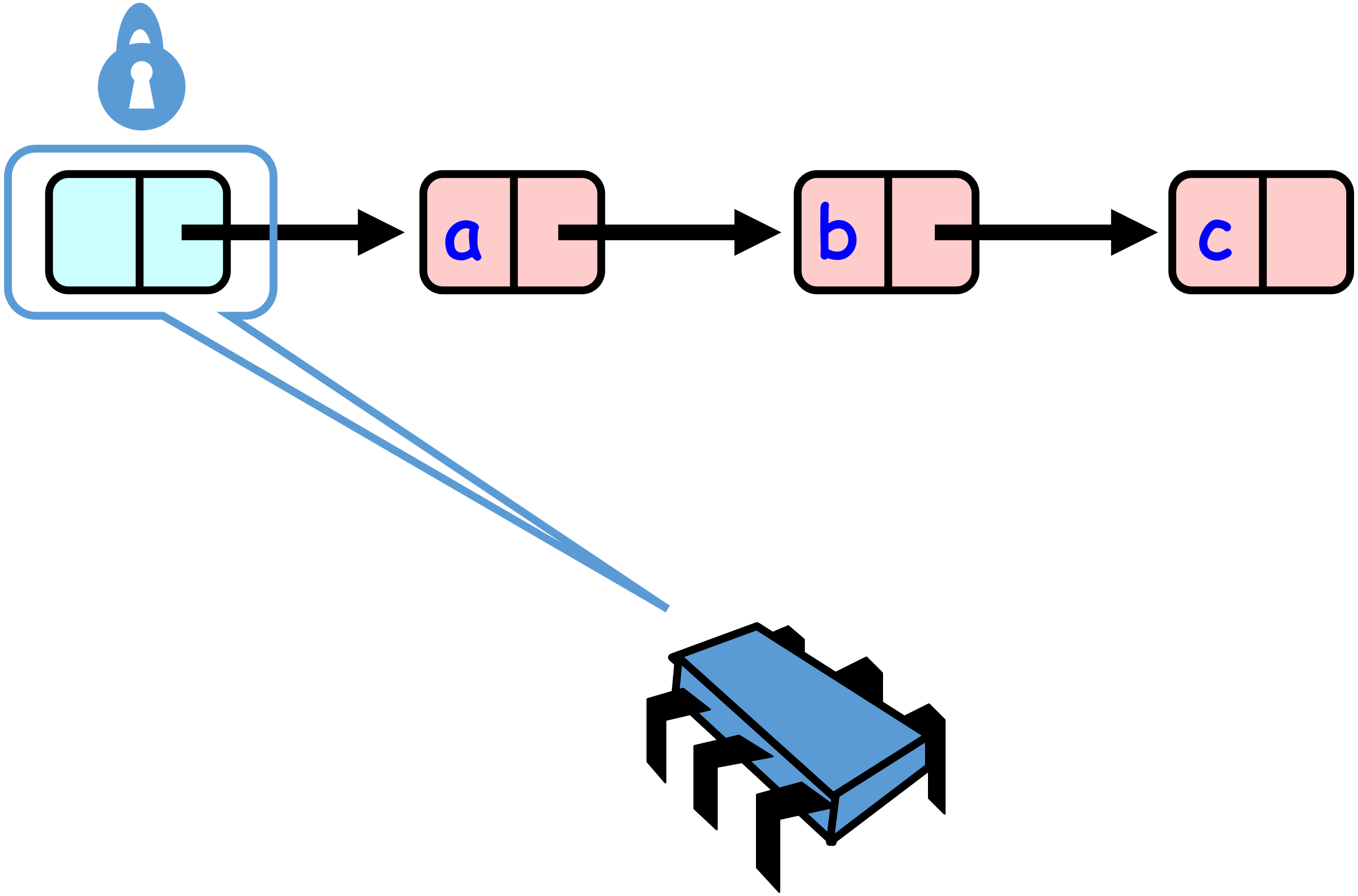
- Requires careful thought
  - “*Do not meddle in the affairs of wizards, for they are subtle and quick to anger*”

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

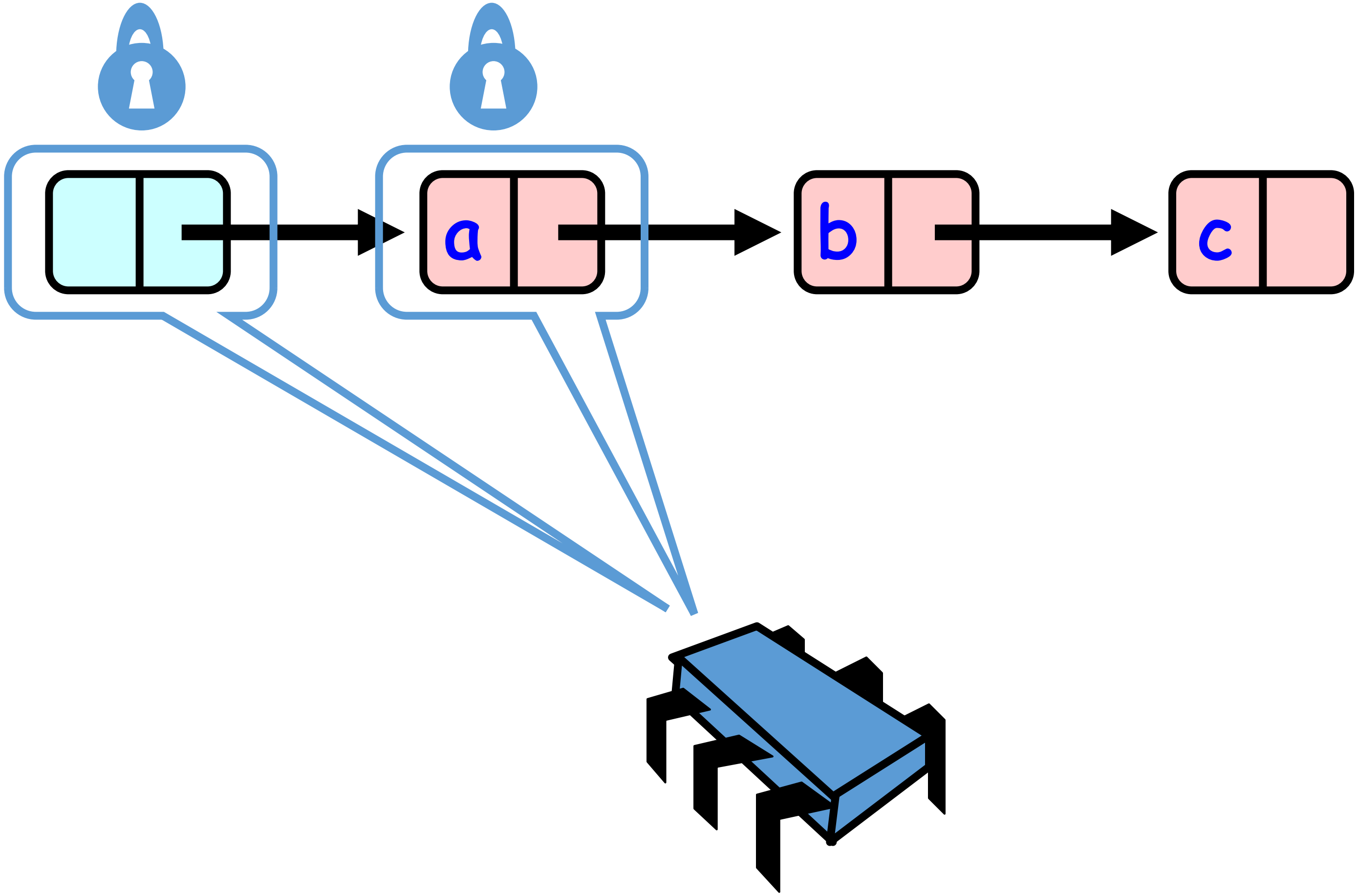
Hand-over-Hand locking



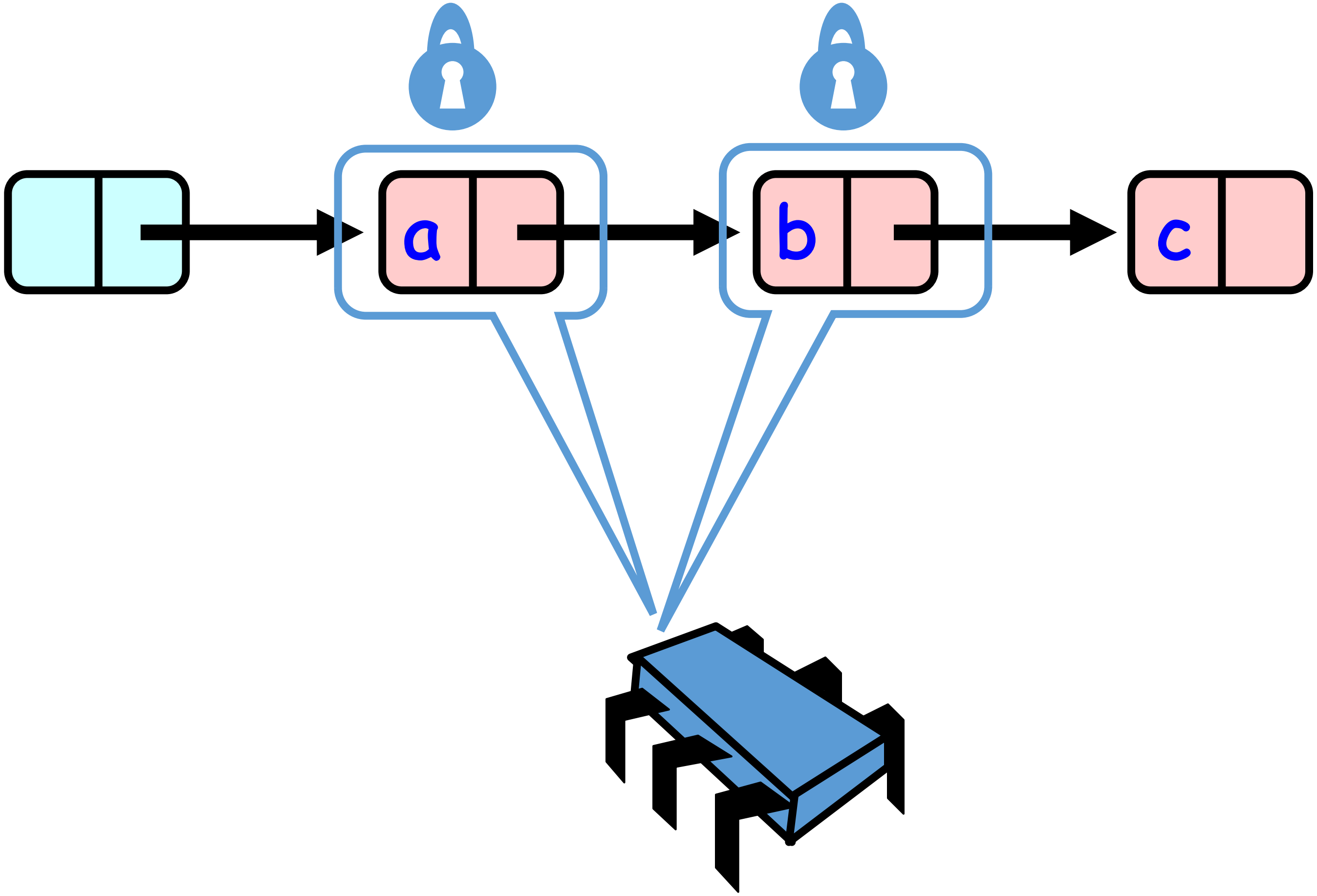
Hand-over-Hand locking



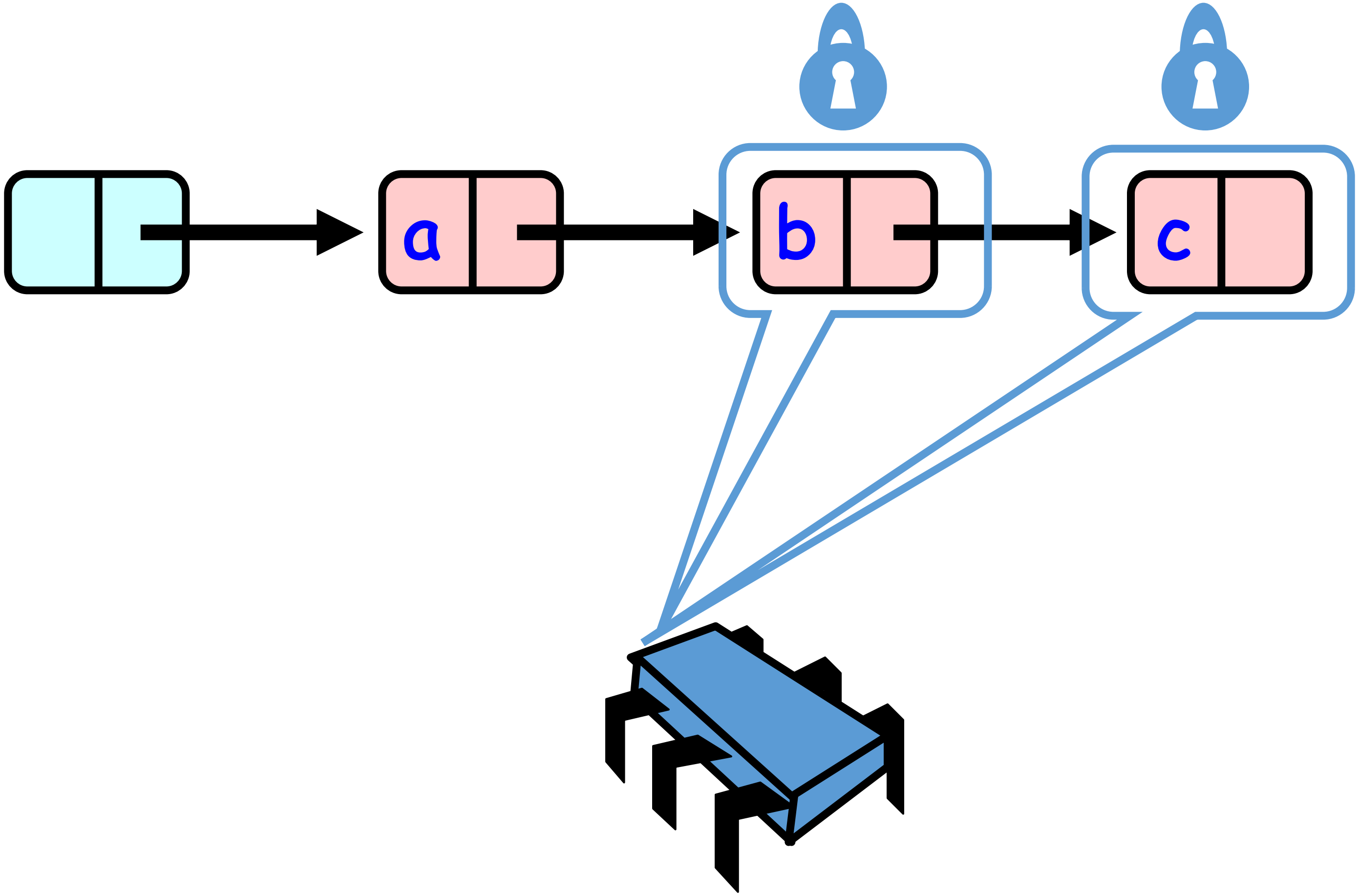
Hand-over-Hand locking



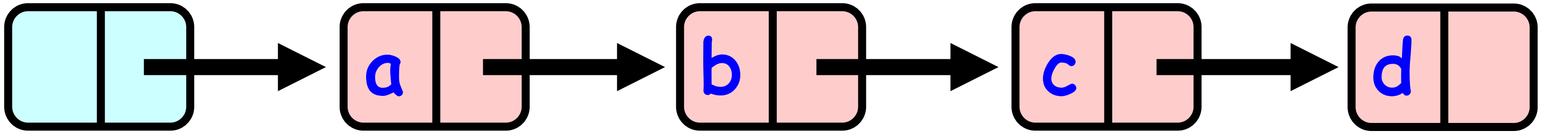
Hand-over-Hand locking

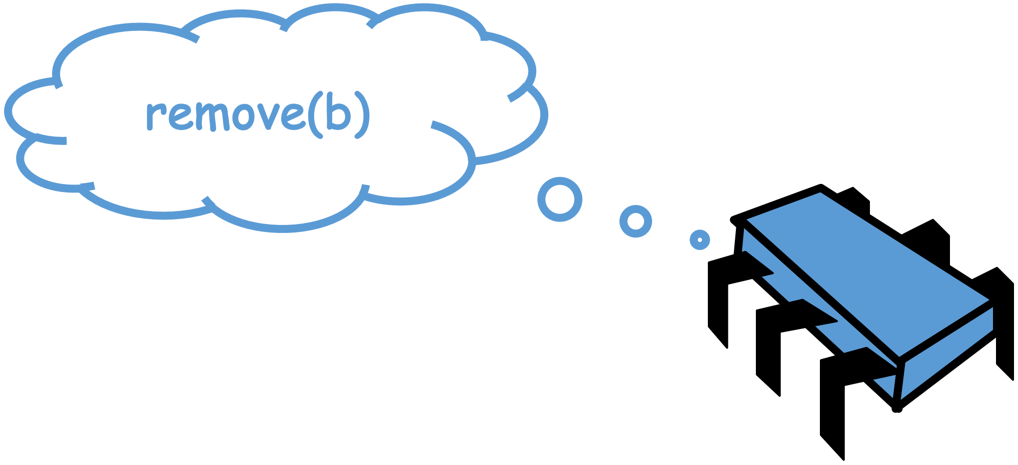


Hand-over-Hand locking

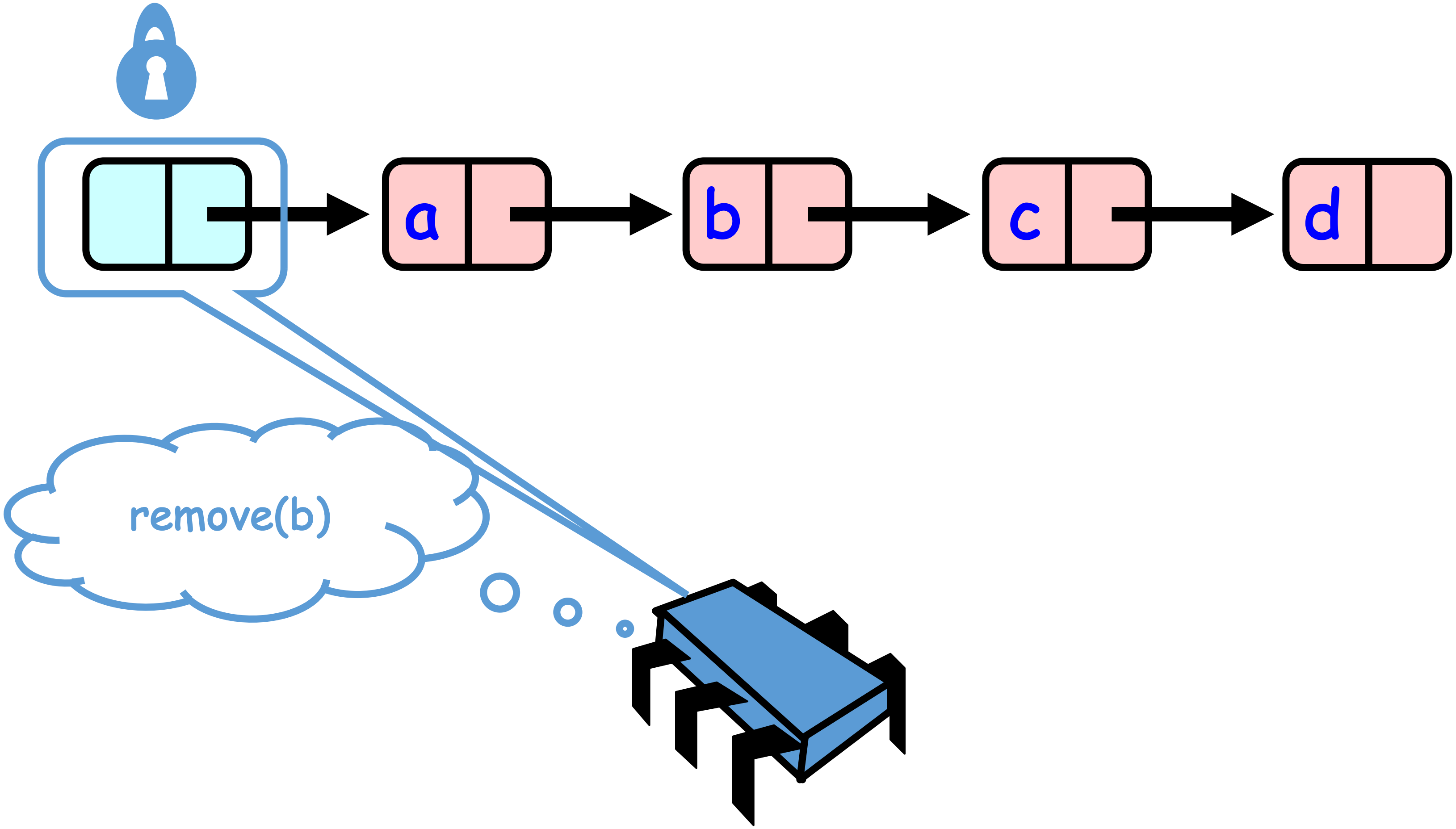


Removing a Node

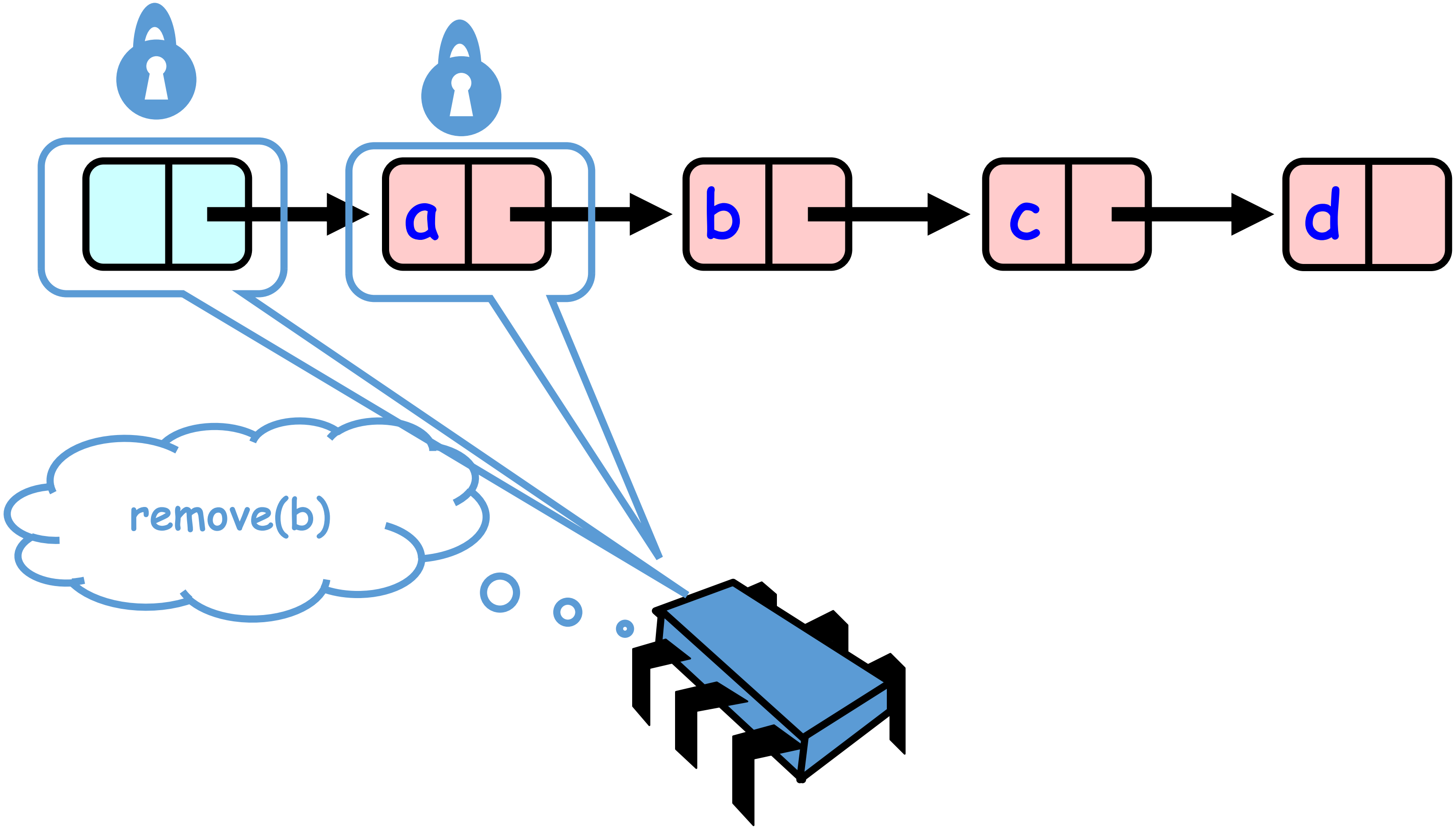




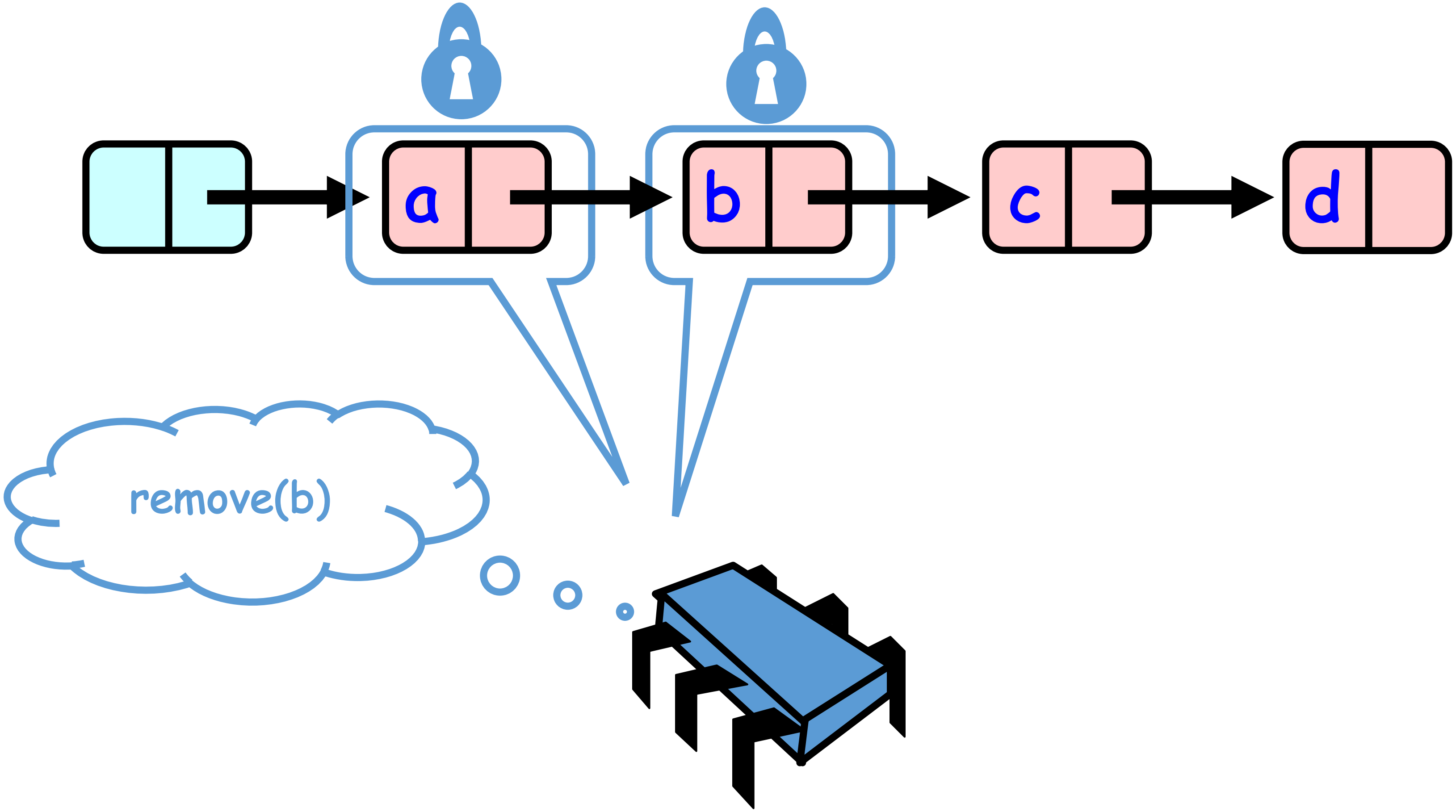
Removing a Node



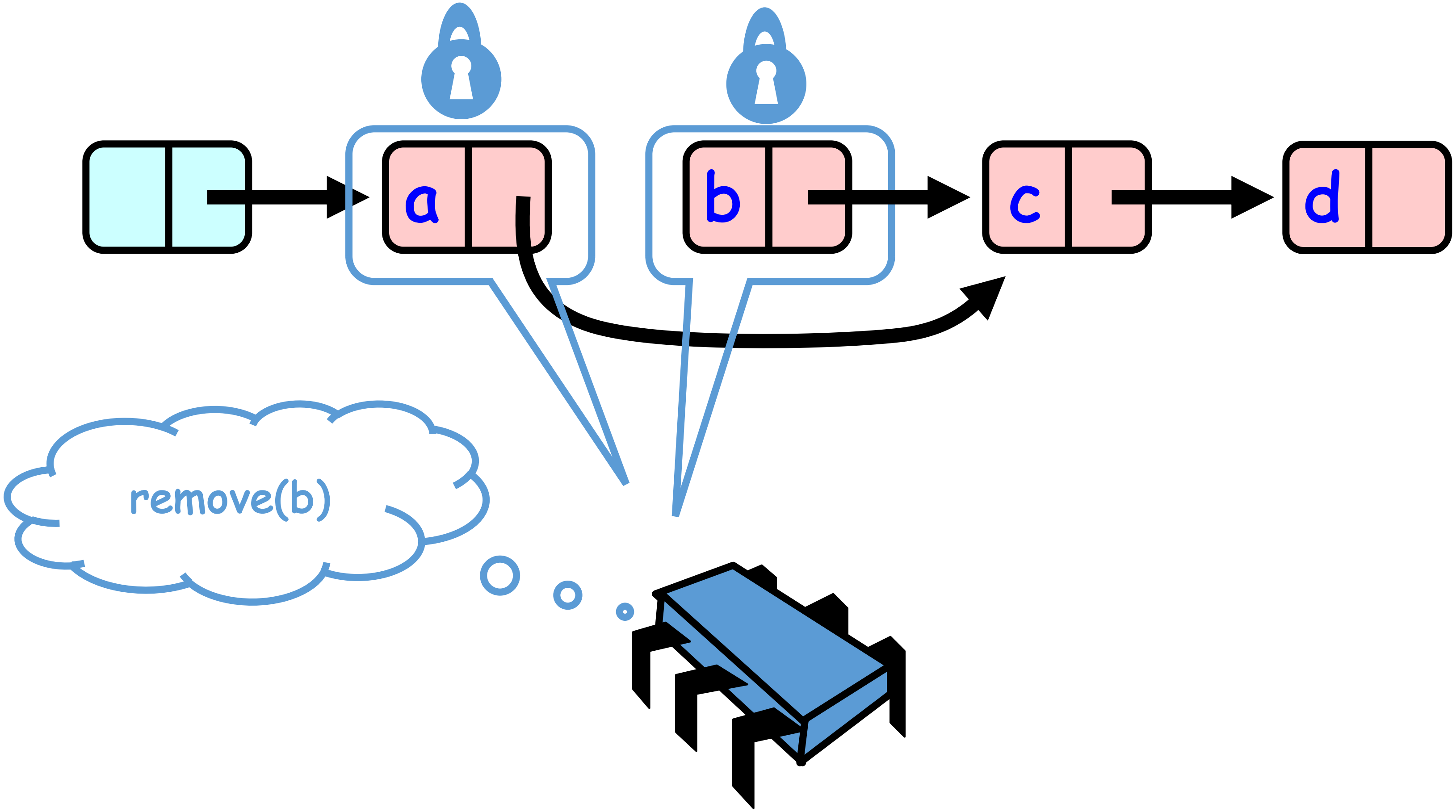
Removing a Node



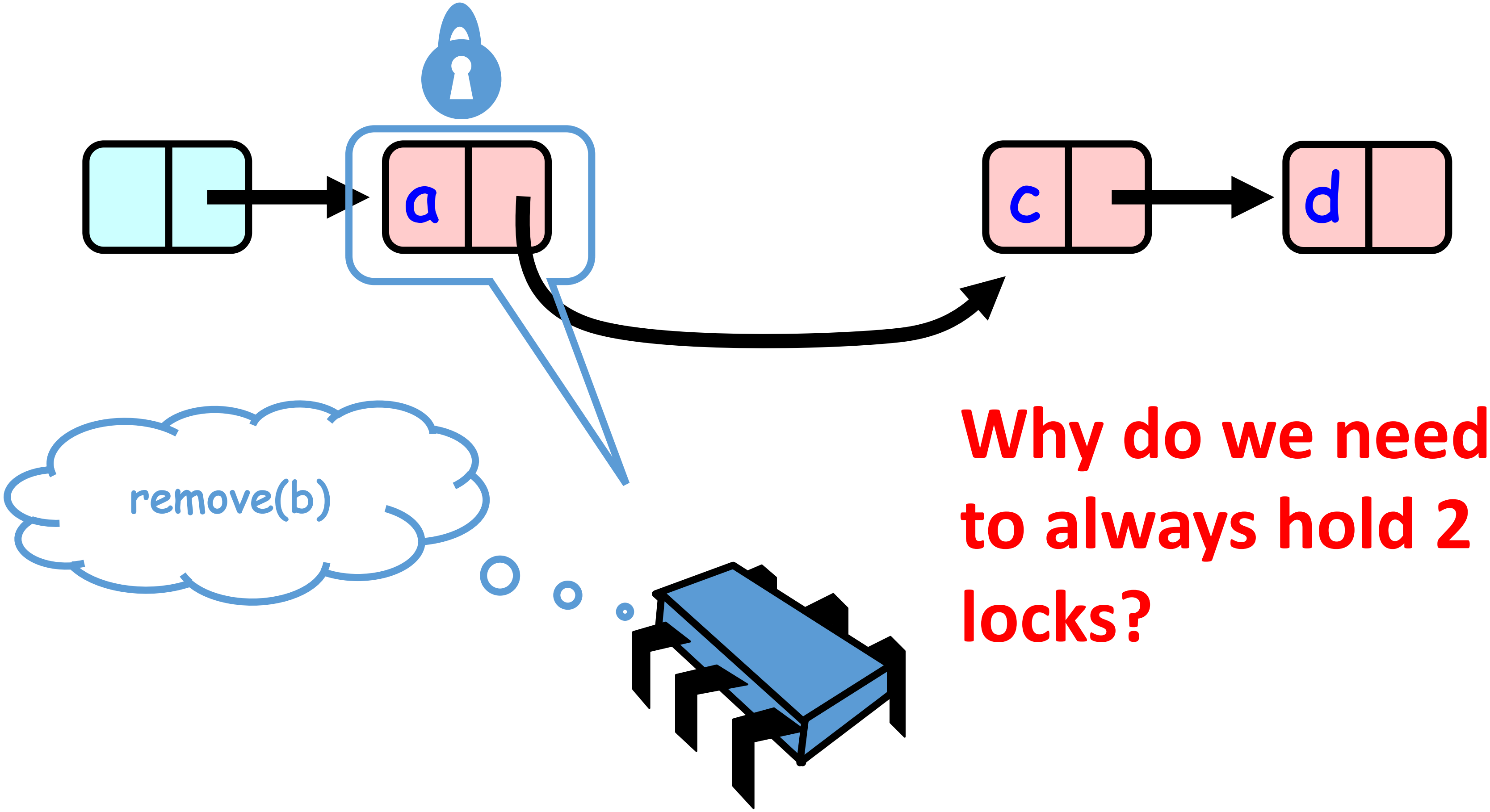
Removing a Node



Removing a Node



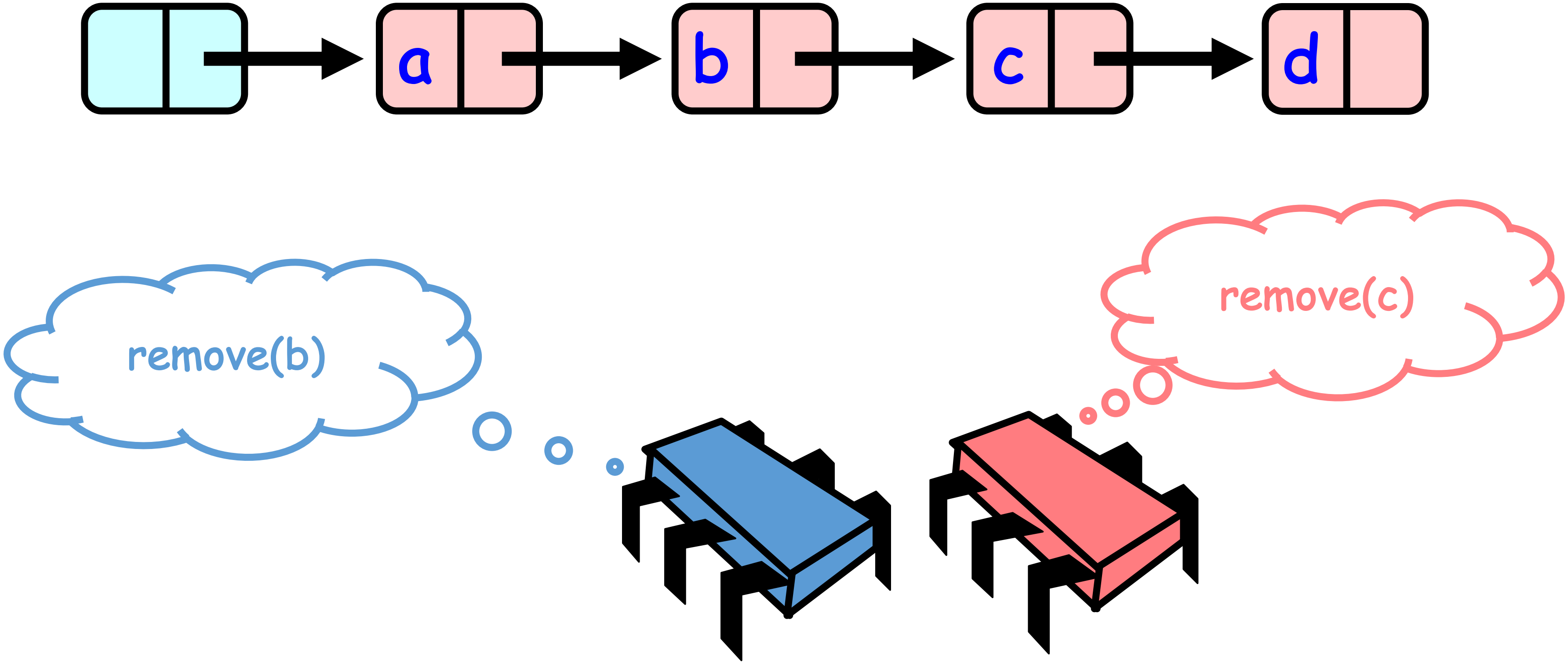
Removing a Node



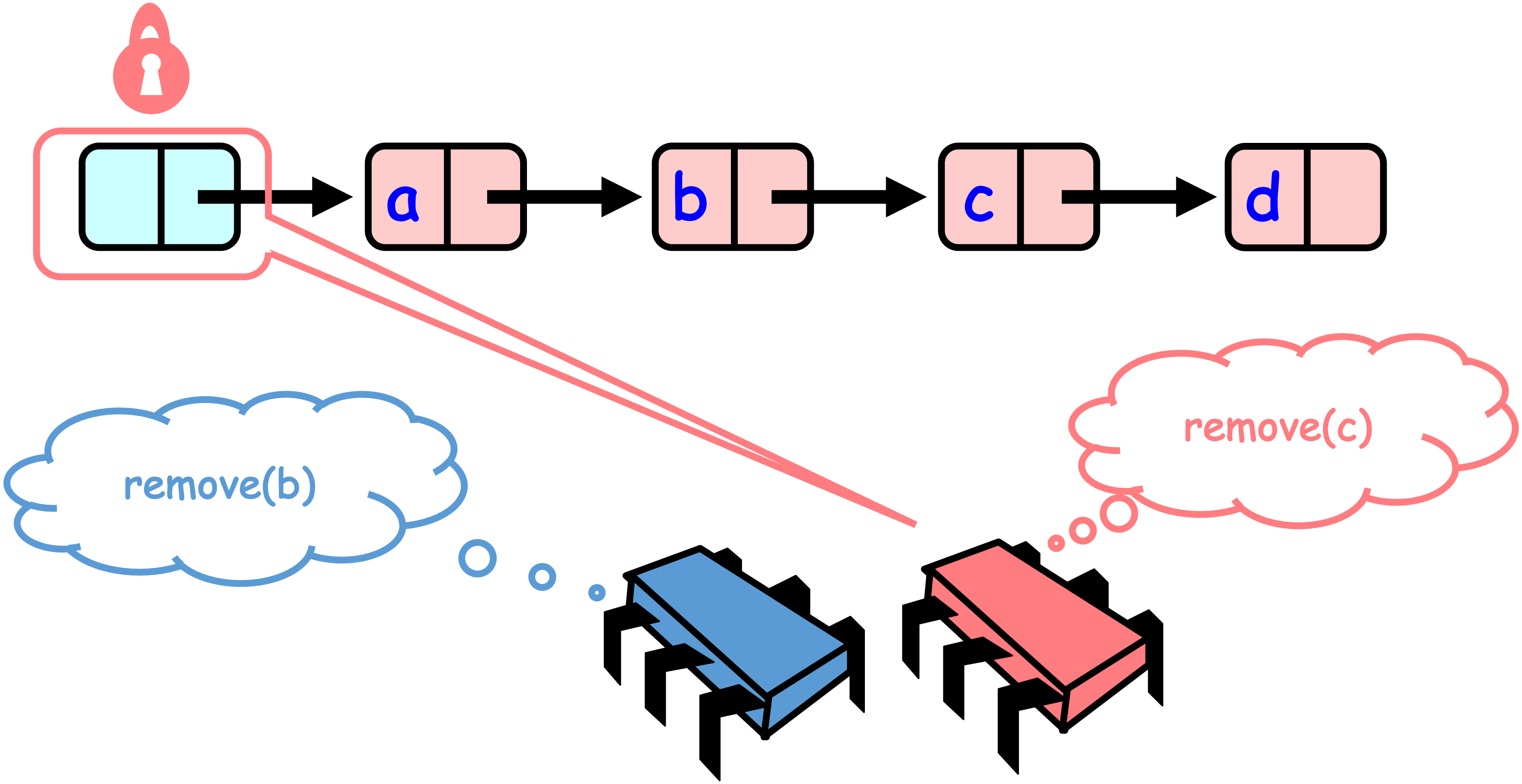
Why do we need to always hold 2 locks?

Concurrent Removes

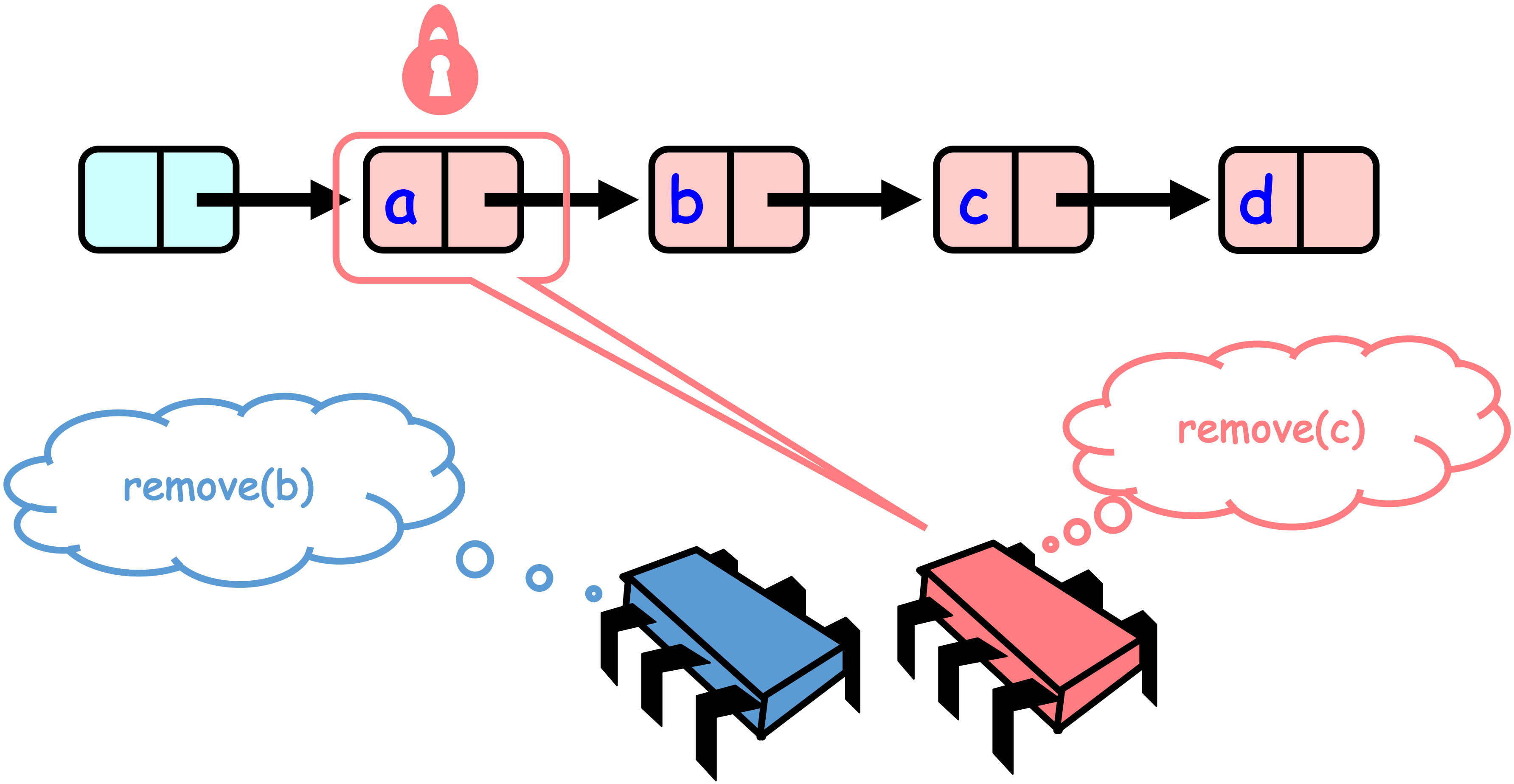
- Holding just one lock (to the node to be changed)



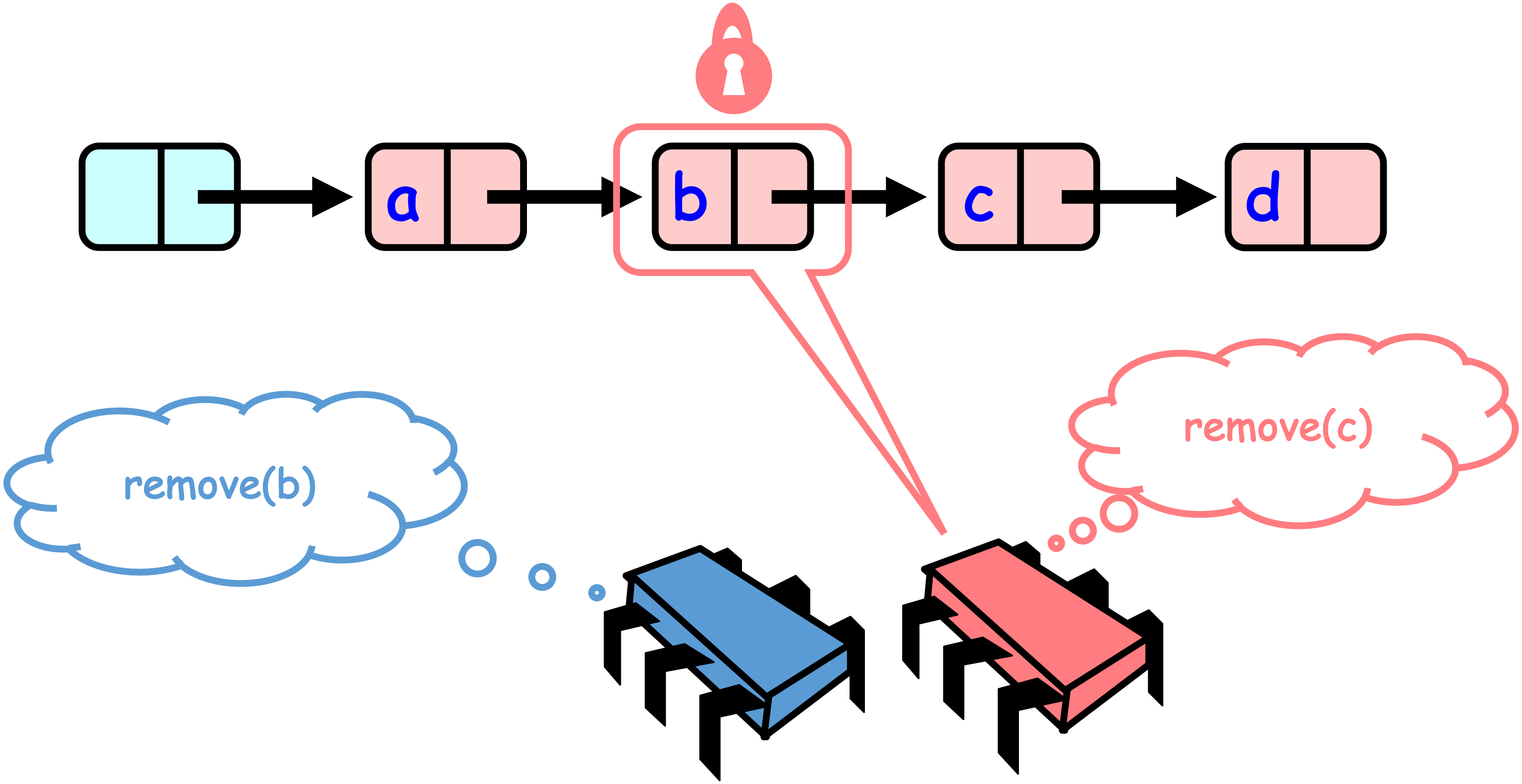
Concurrent Removes



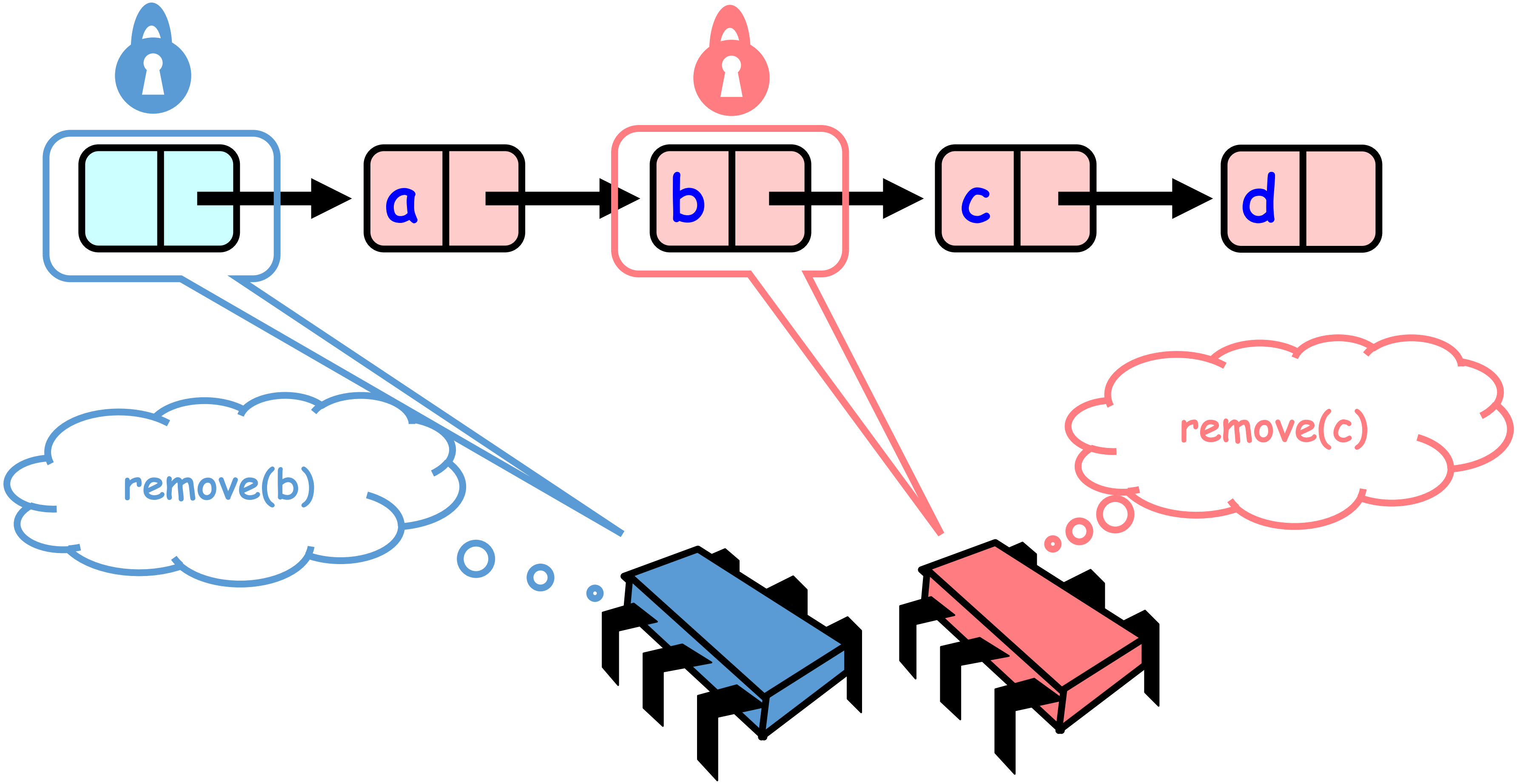
Concurrent Removes



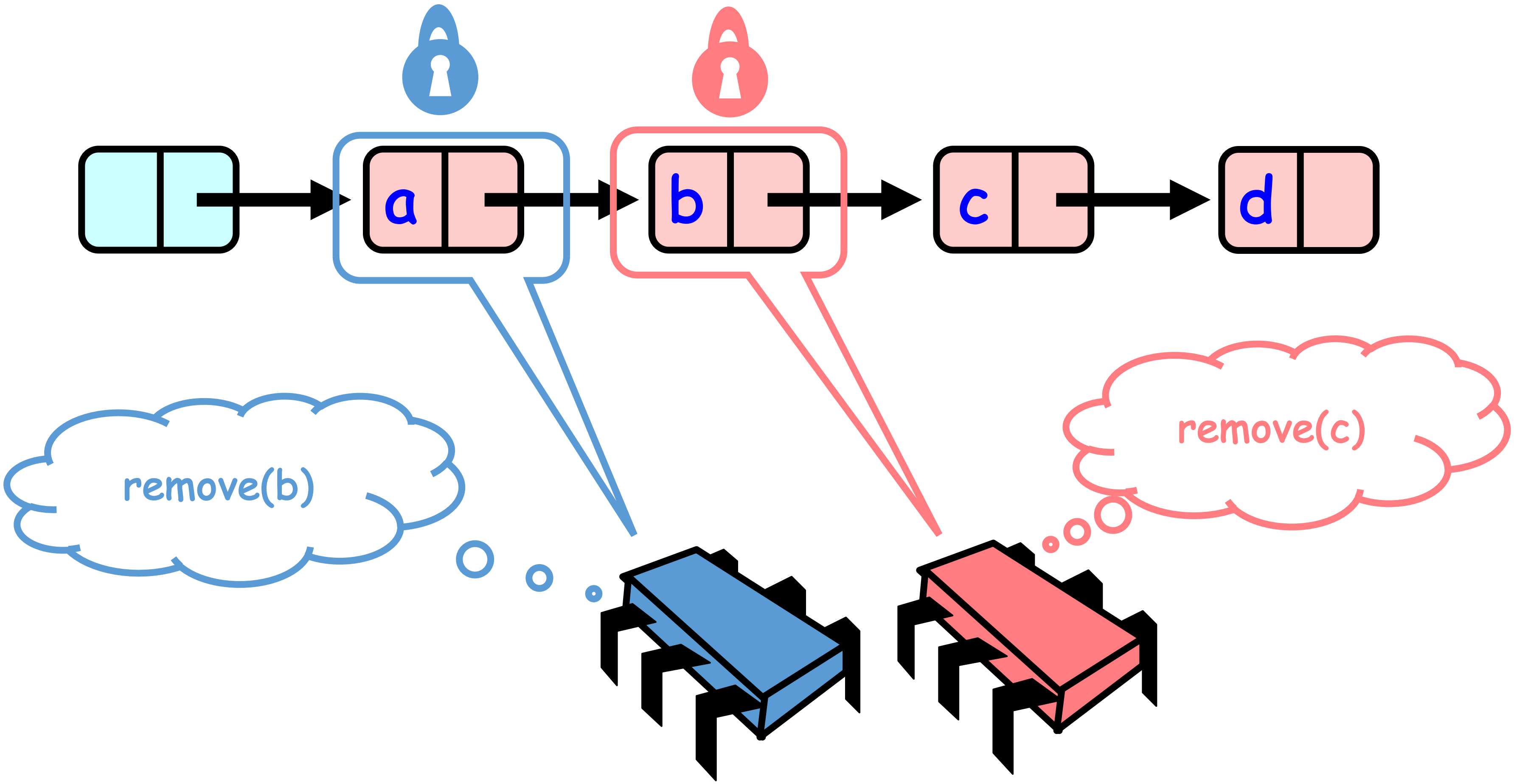
Concurrent Removes



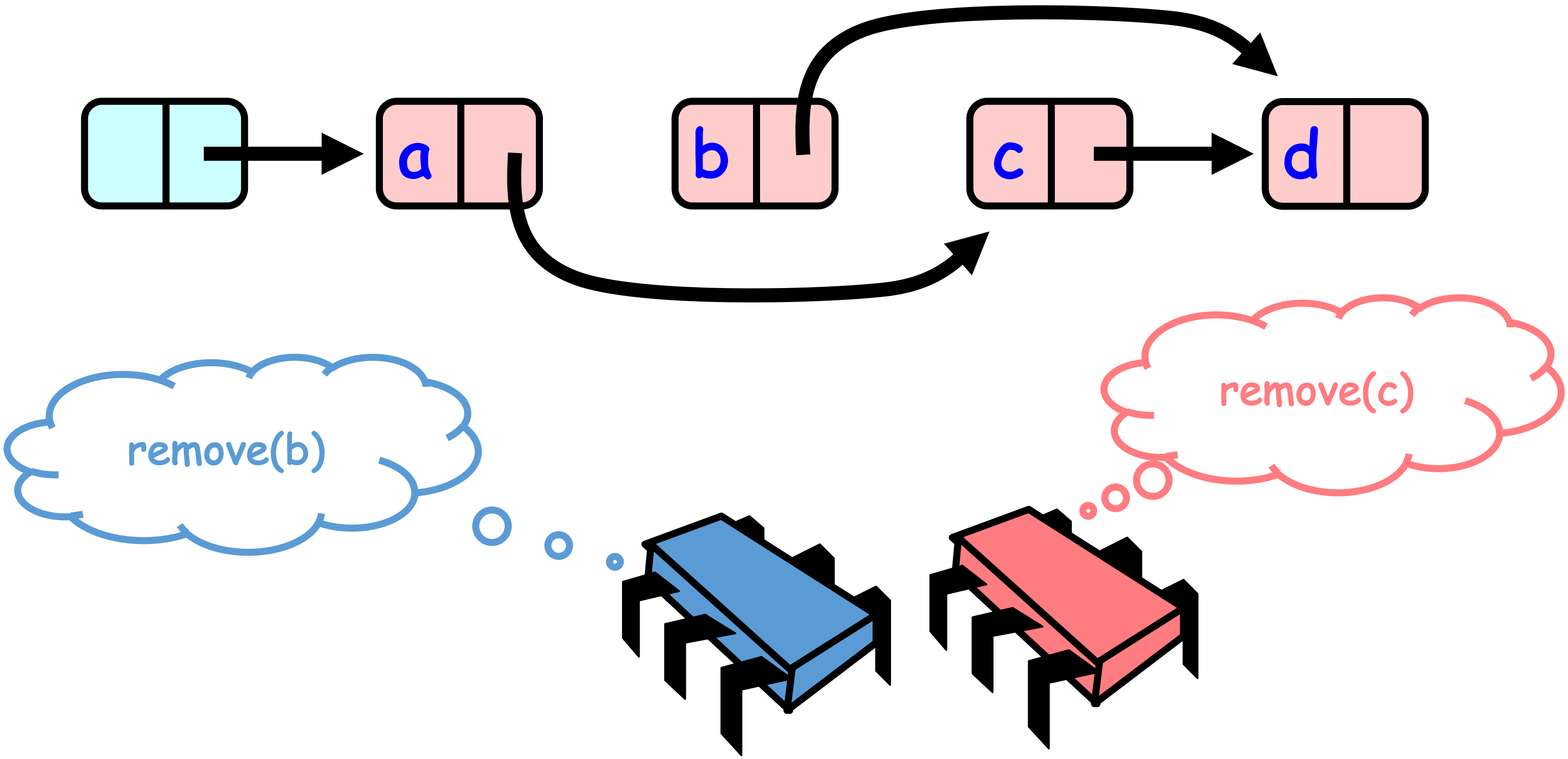
Concurrent Removes



Concurrent Removes

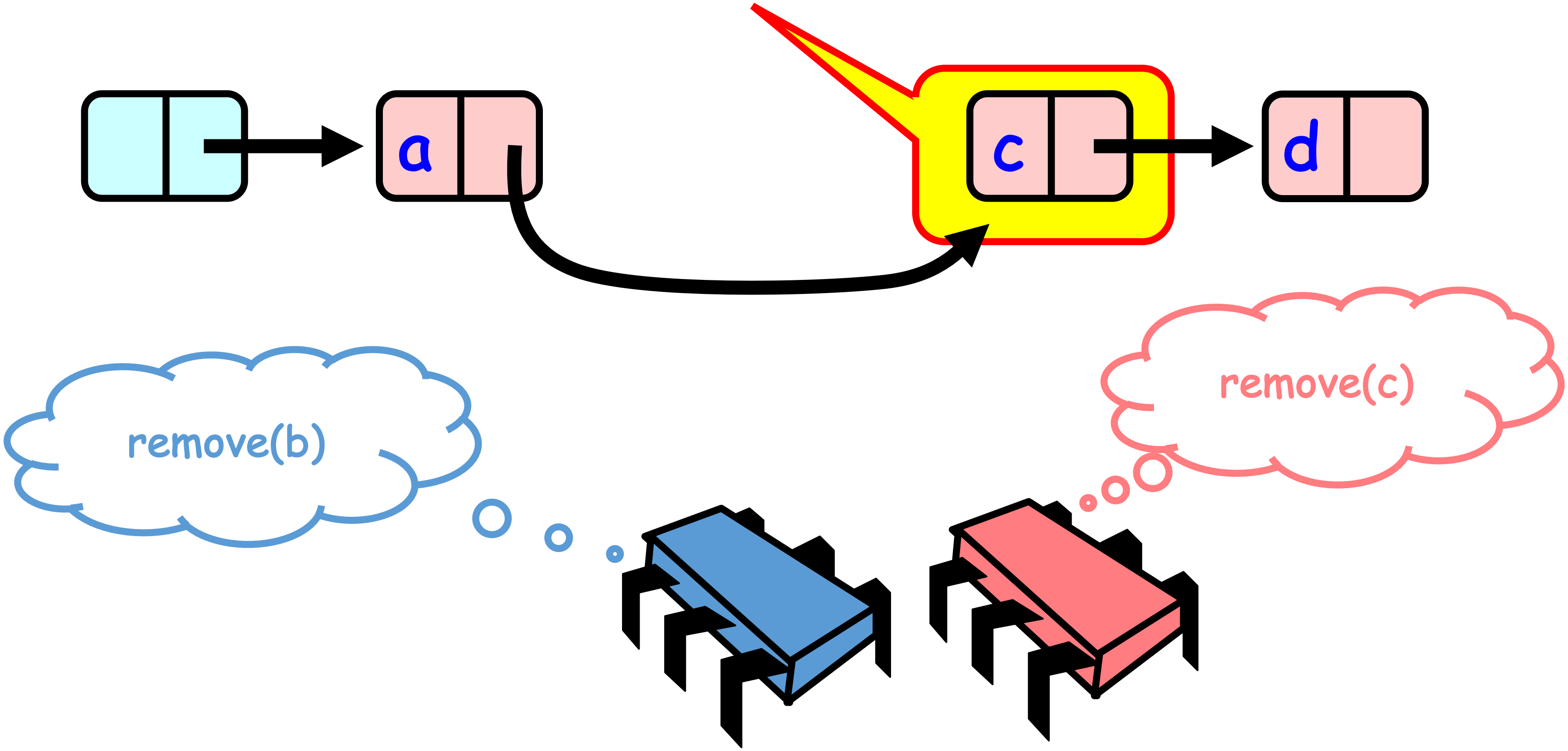


Concurrent Removes



Uh, Oh

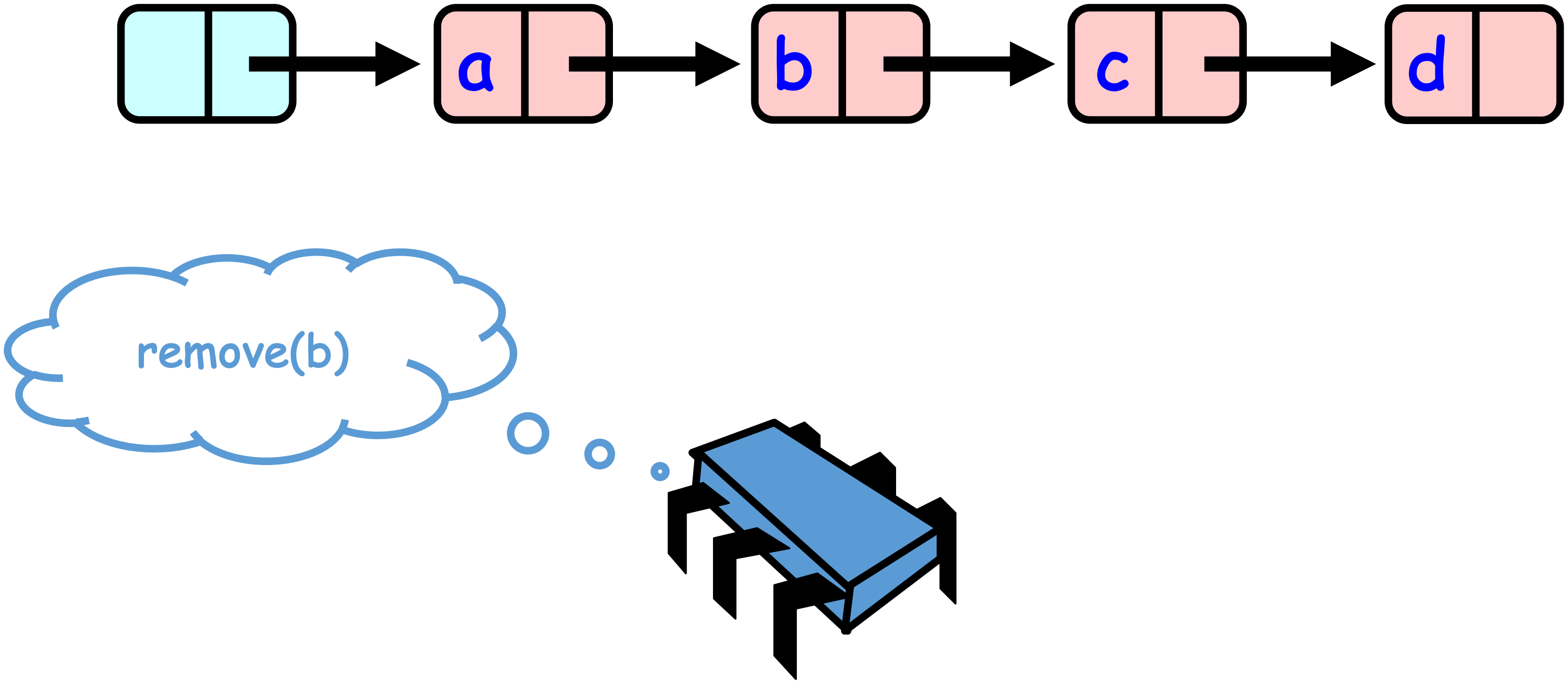
Bad news, C not removed



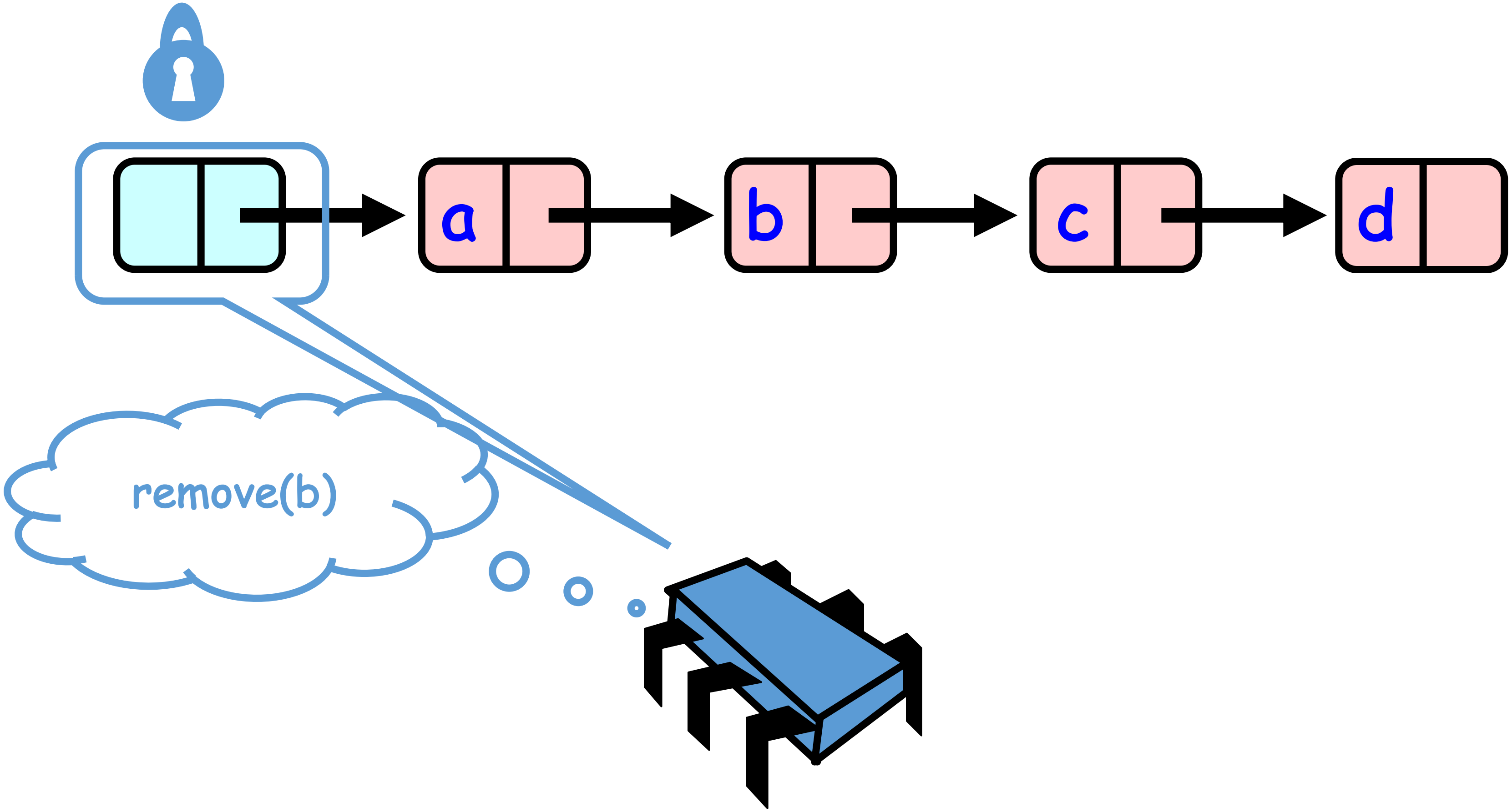
Insight

- 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

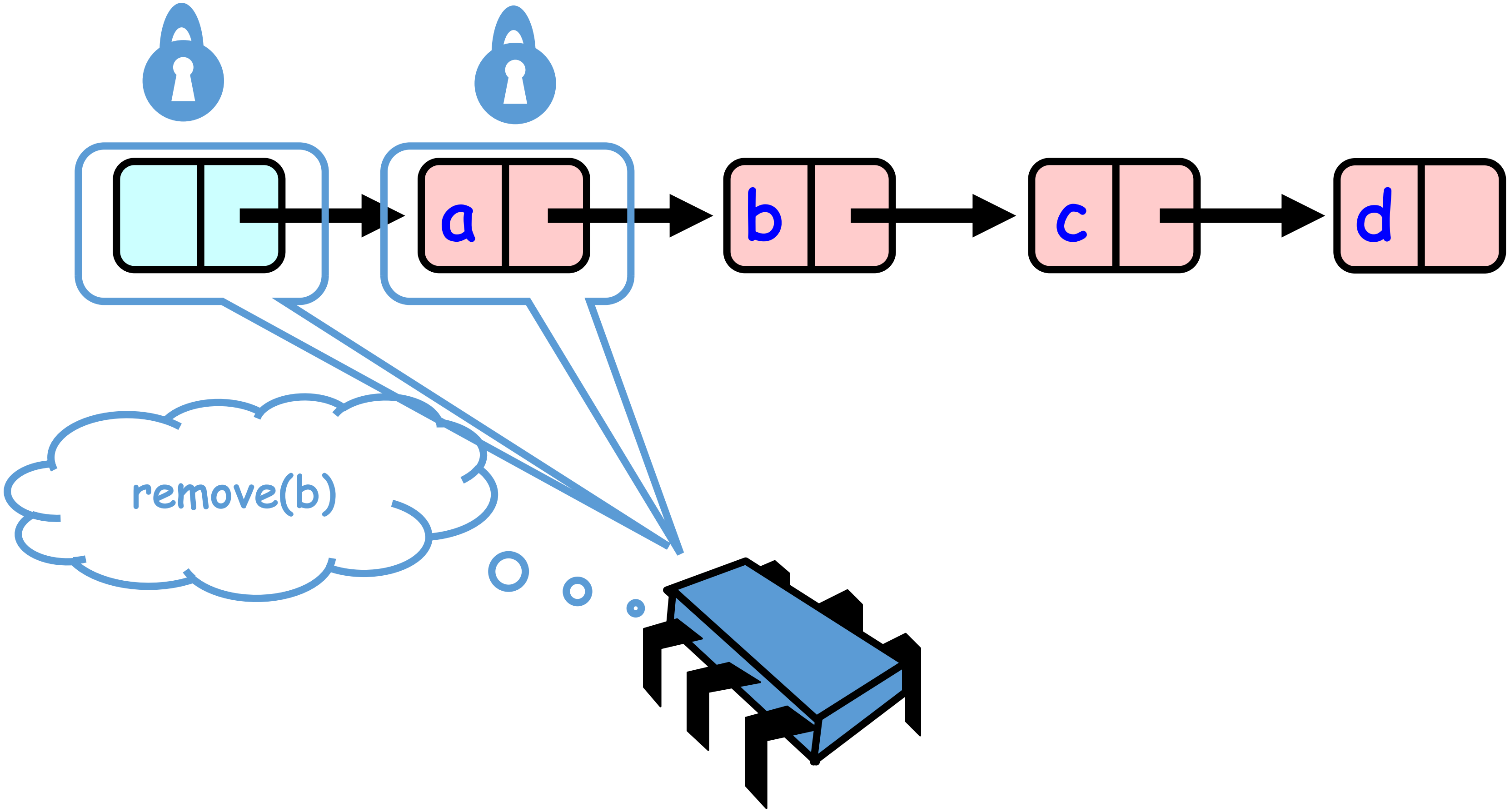
Hand-Over-Hand Again



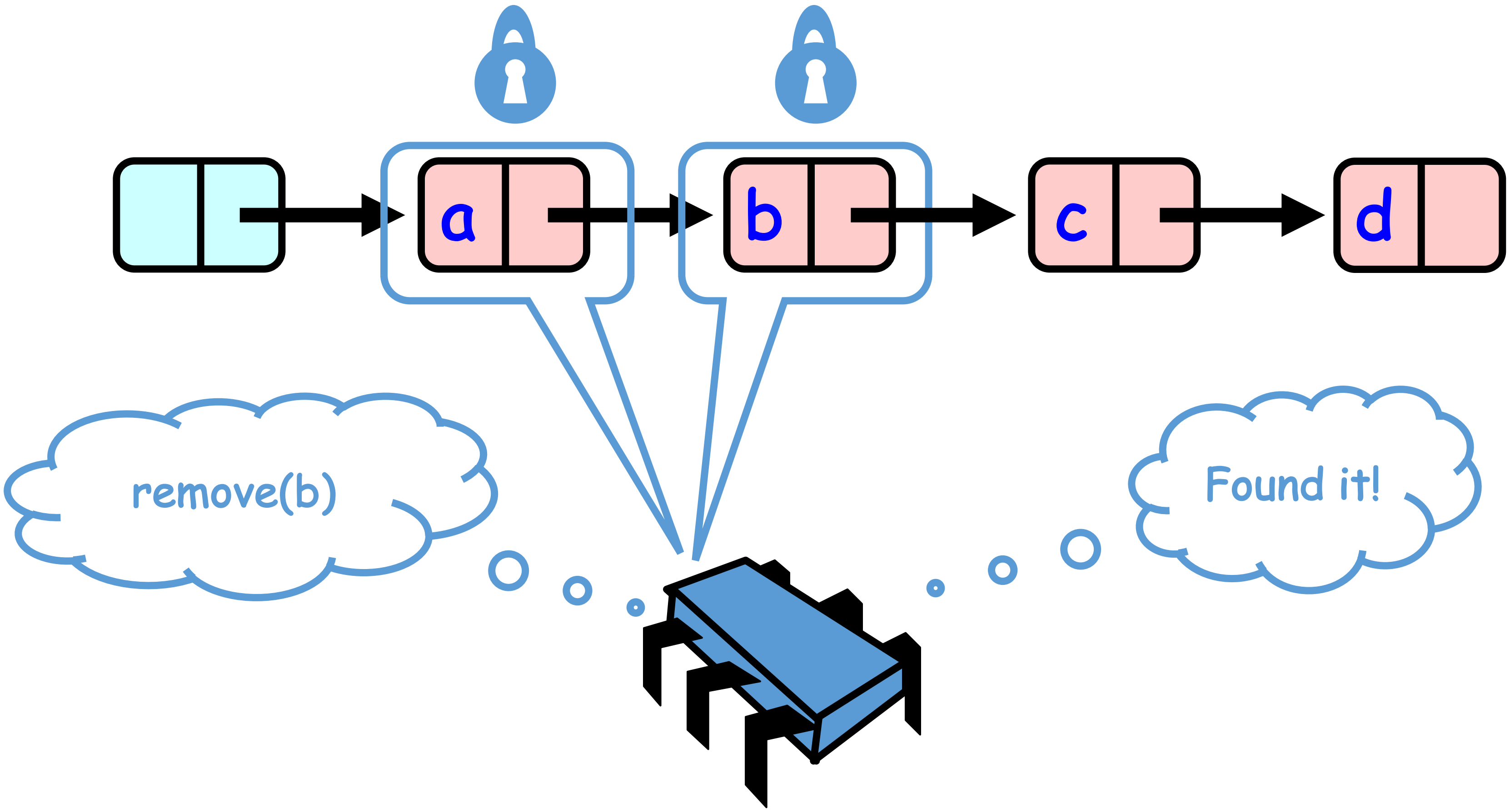
Hand-Over-Hand Again



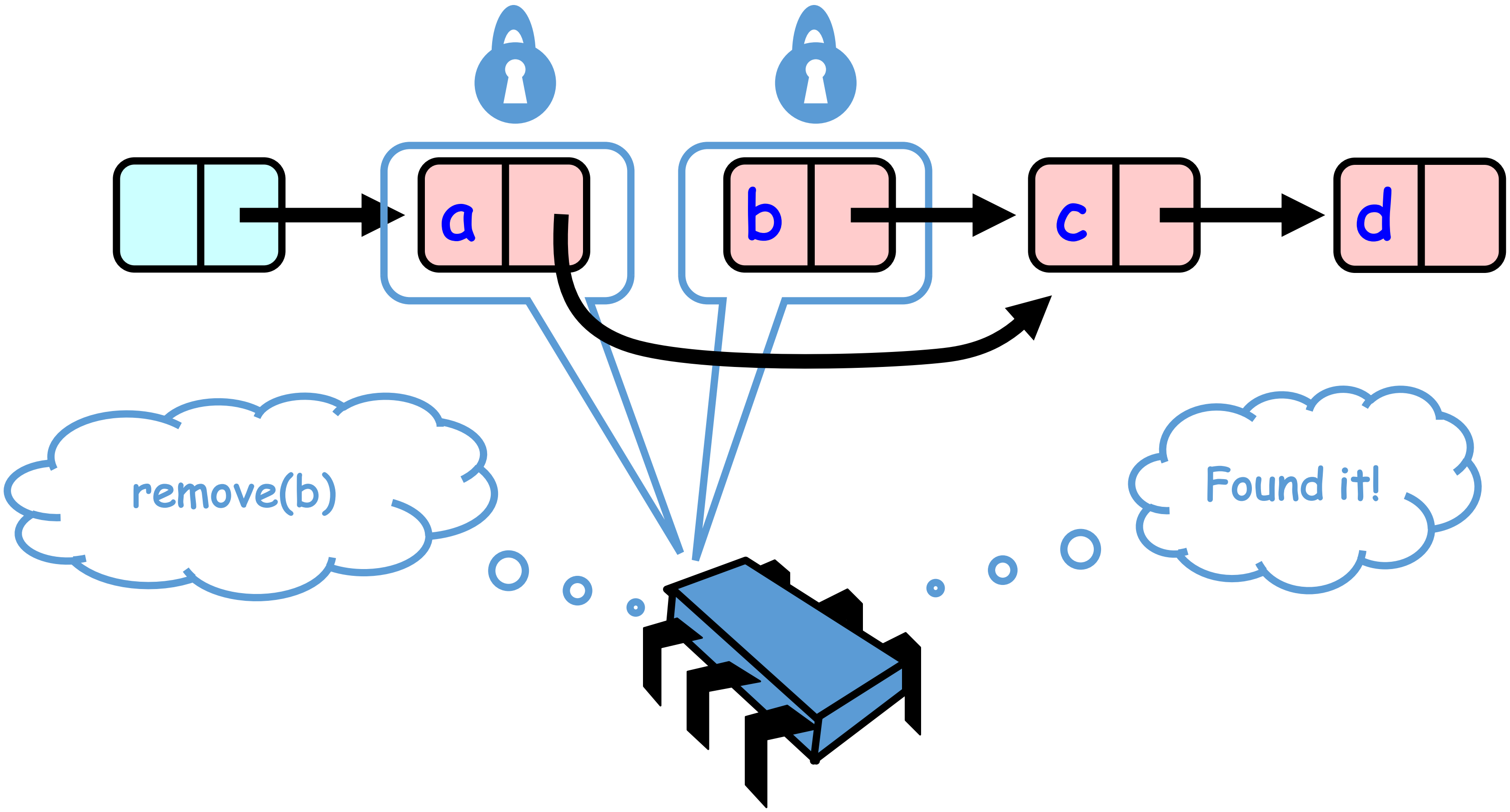
Hand-Over-Hand Again



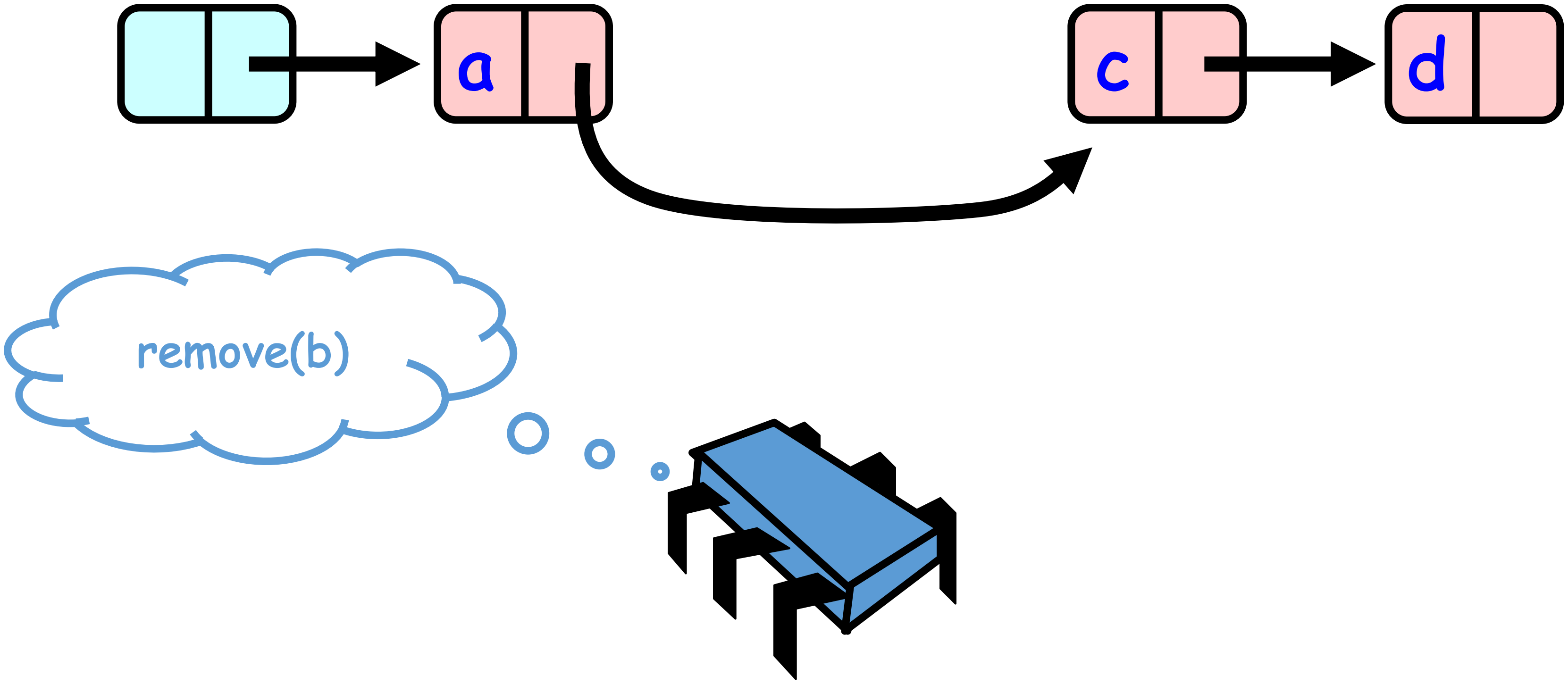
Hand-Over-Hand Again



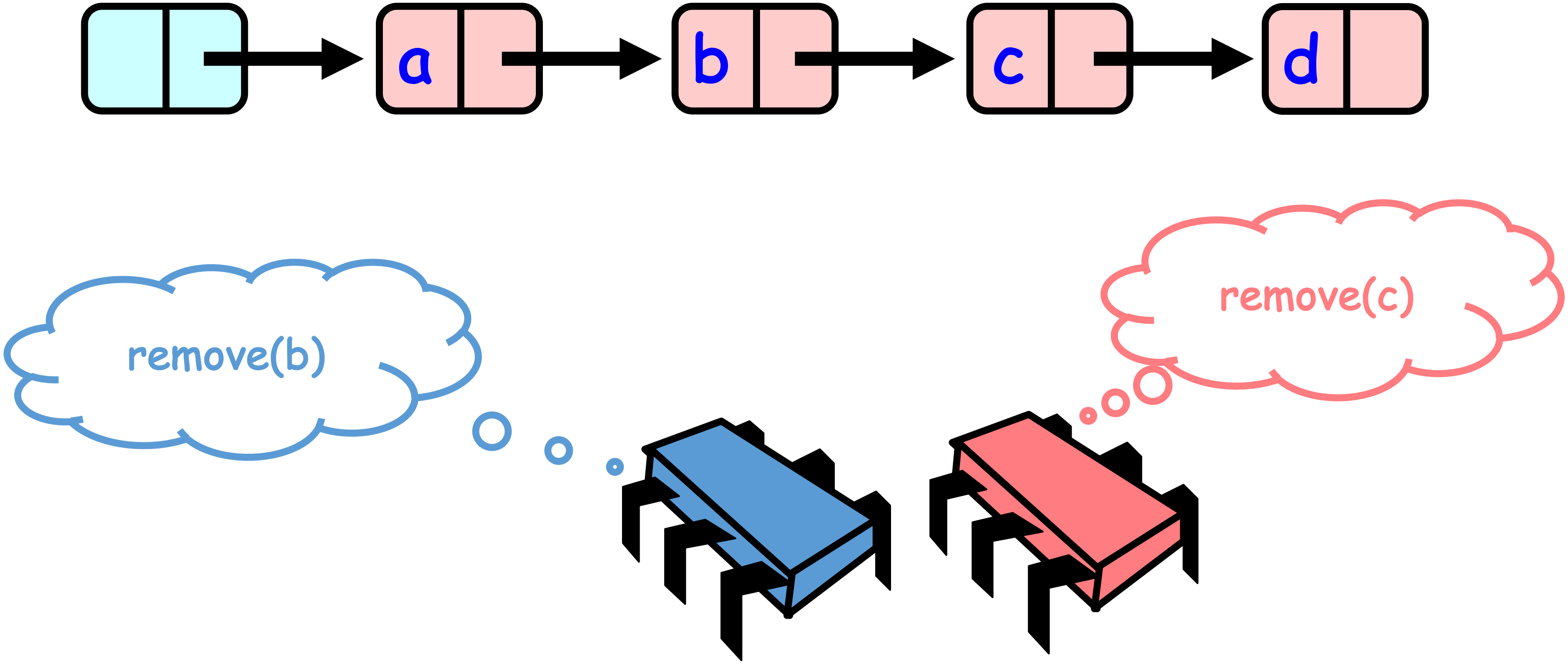
Hand-Over-Hand Again



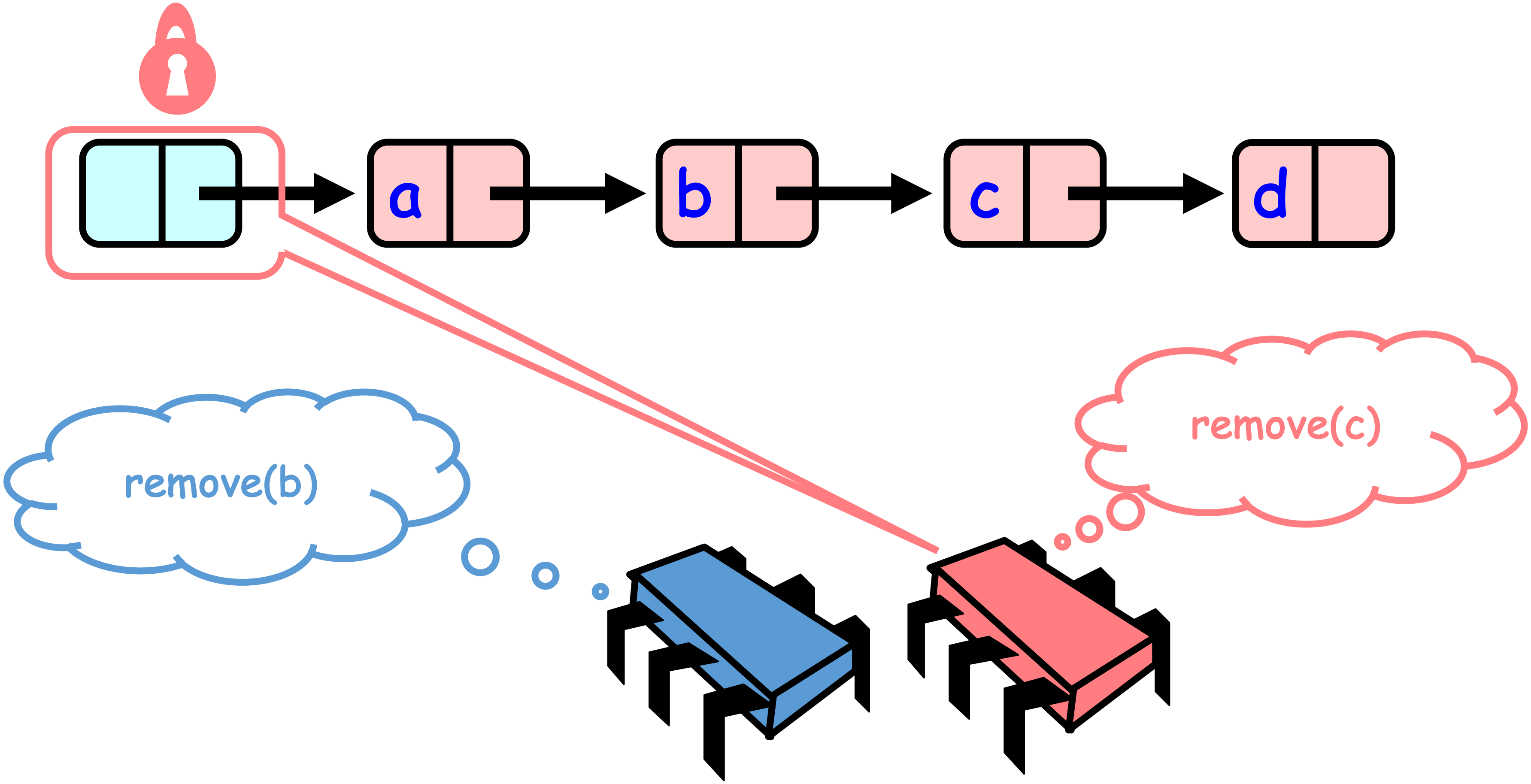
Hand-Over-Hand Again



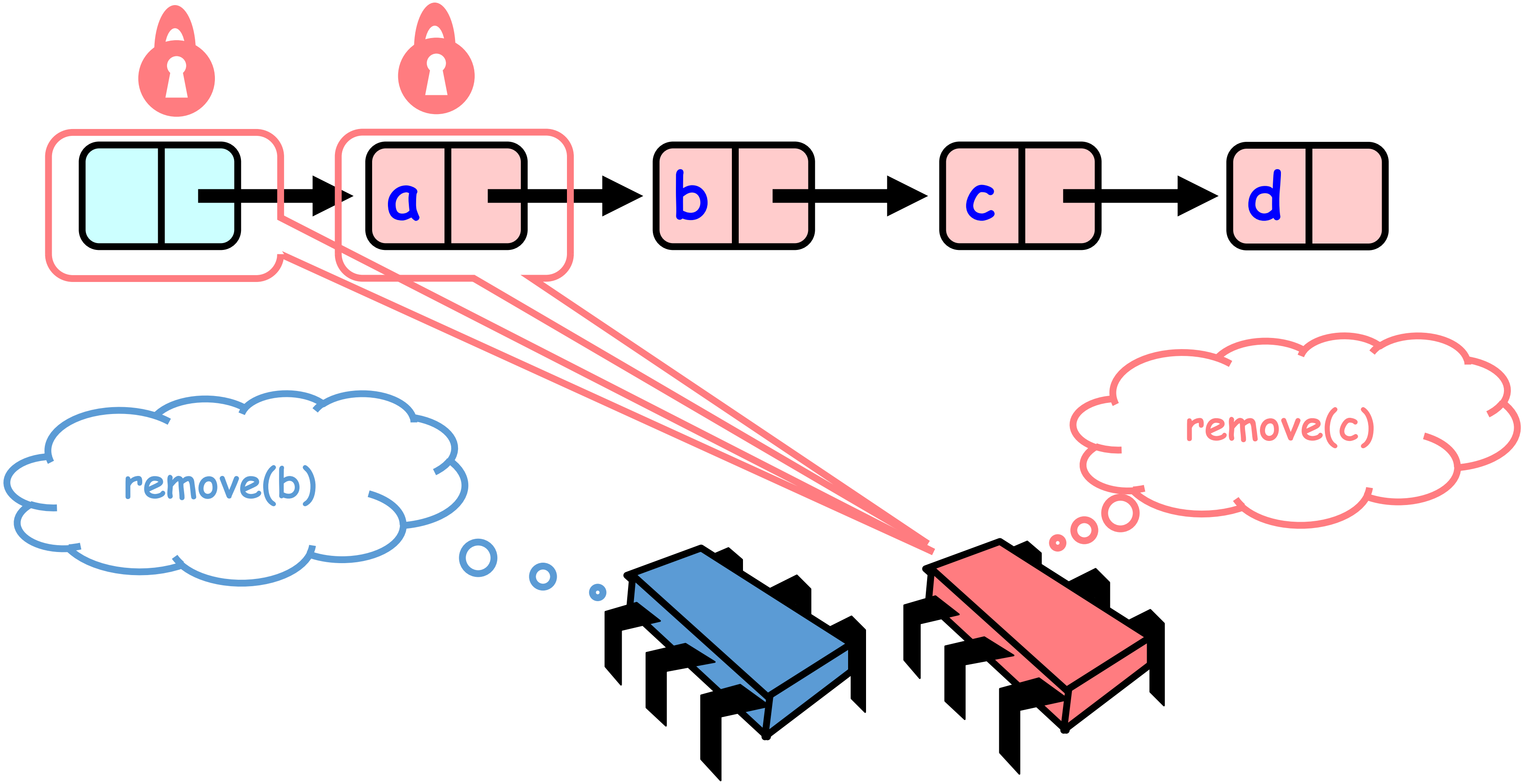
Removing a Node



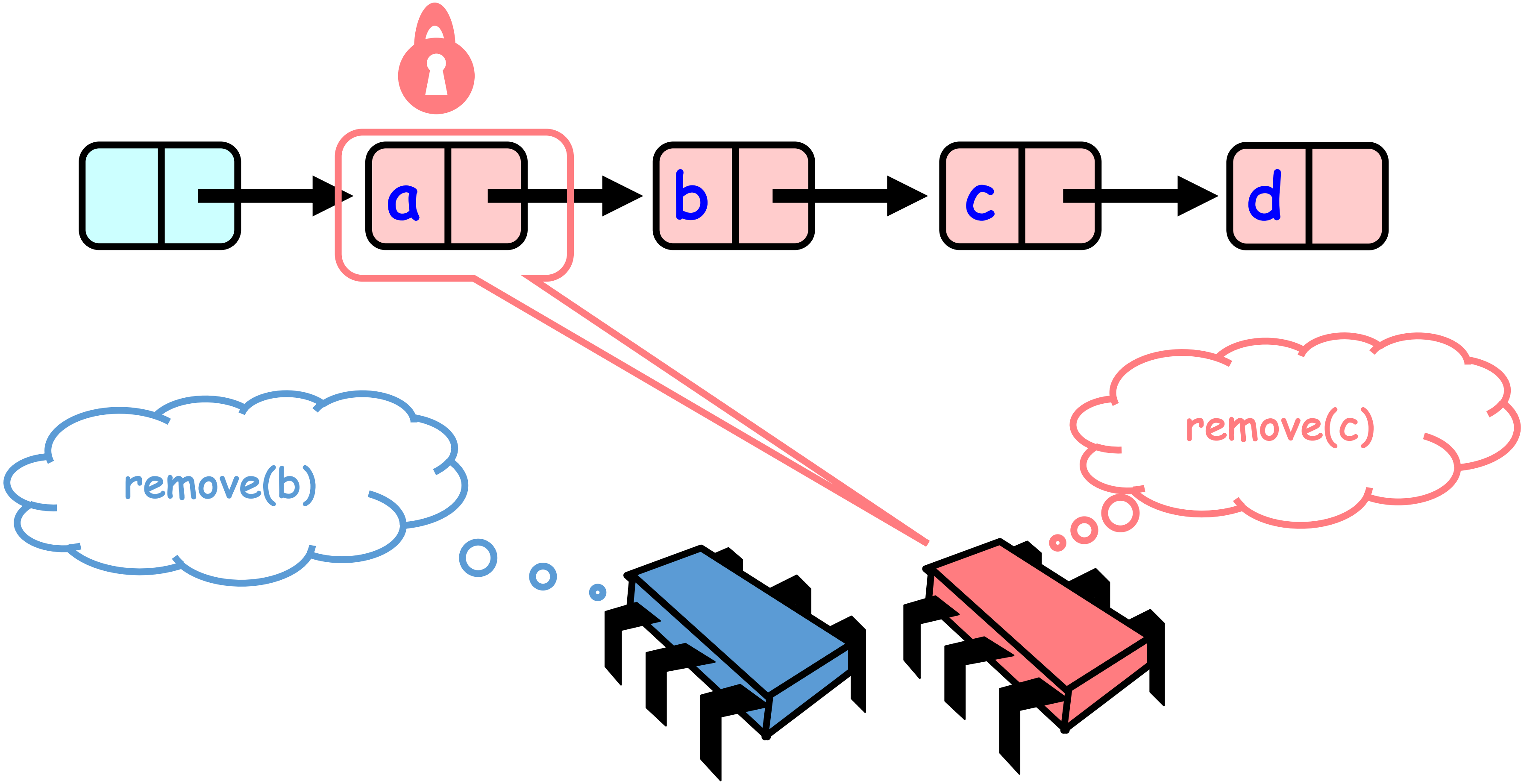
Removing a Node



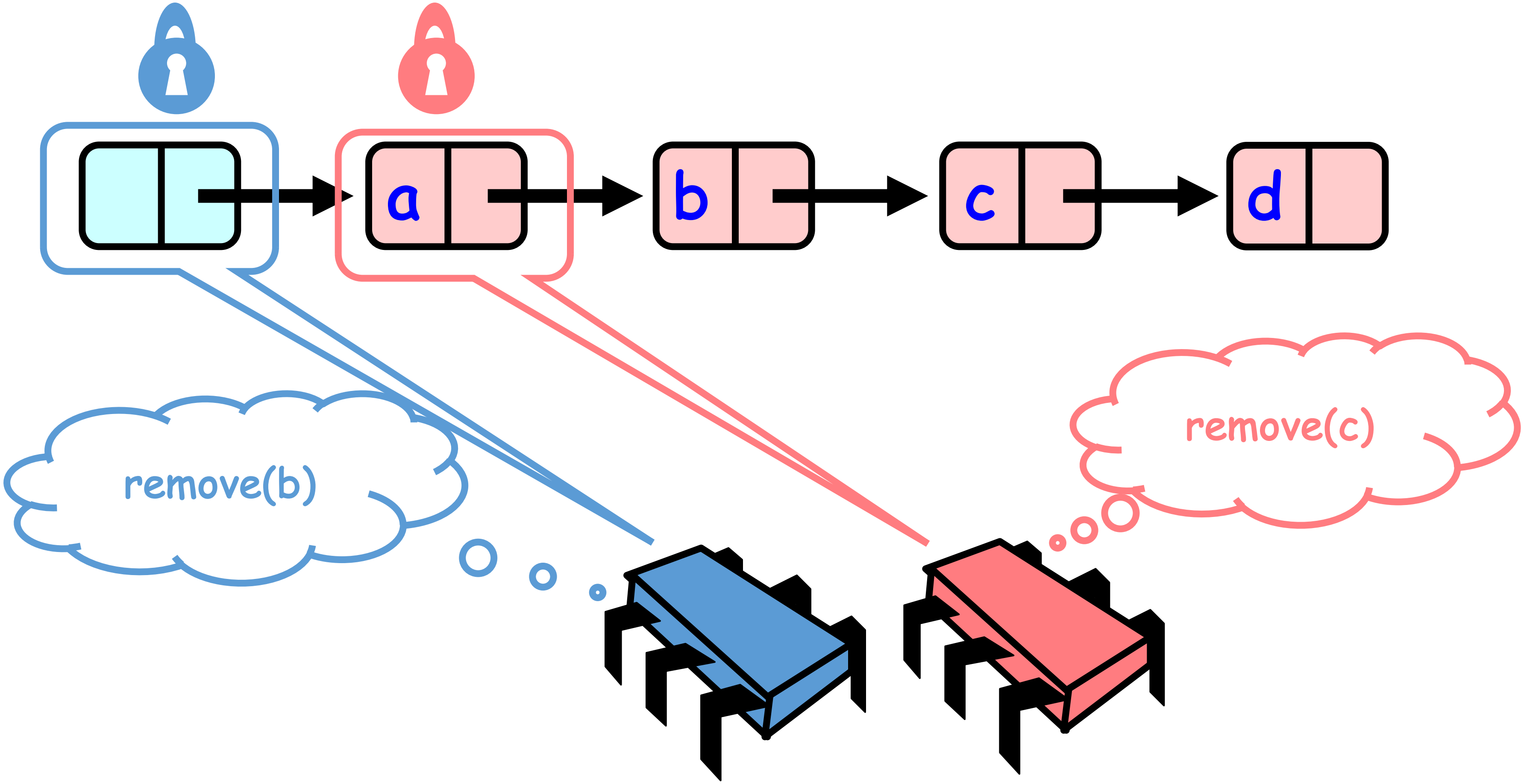
Removing a Node



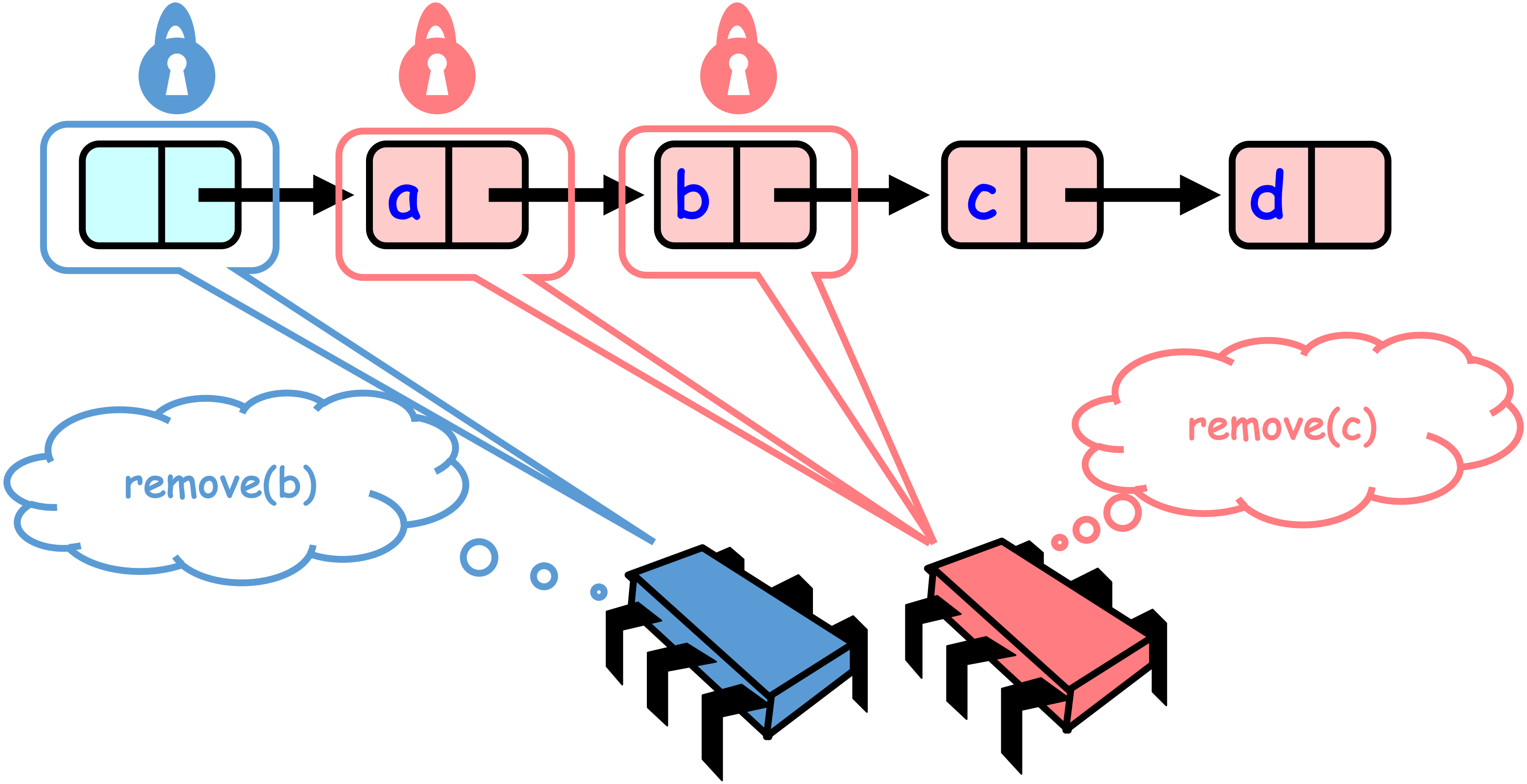
Removing a Node



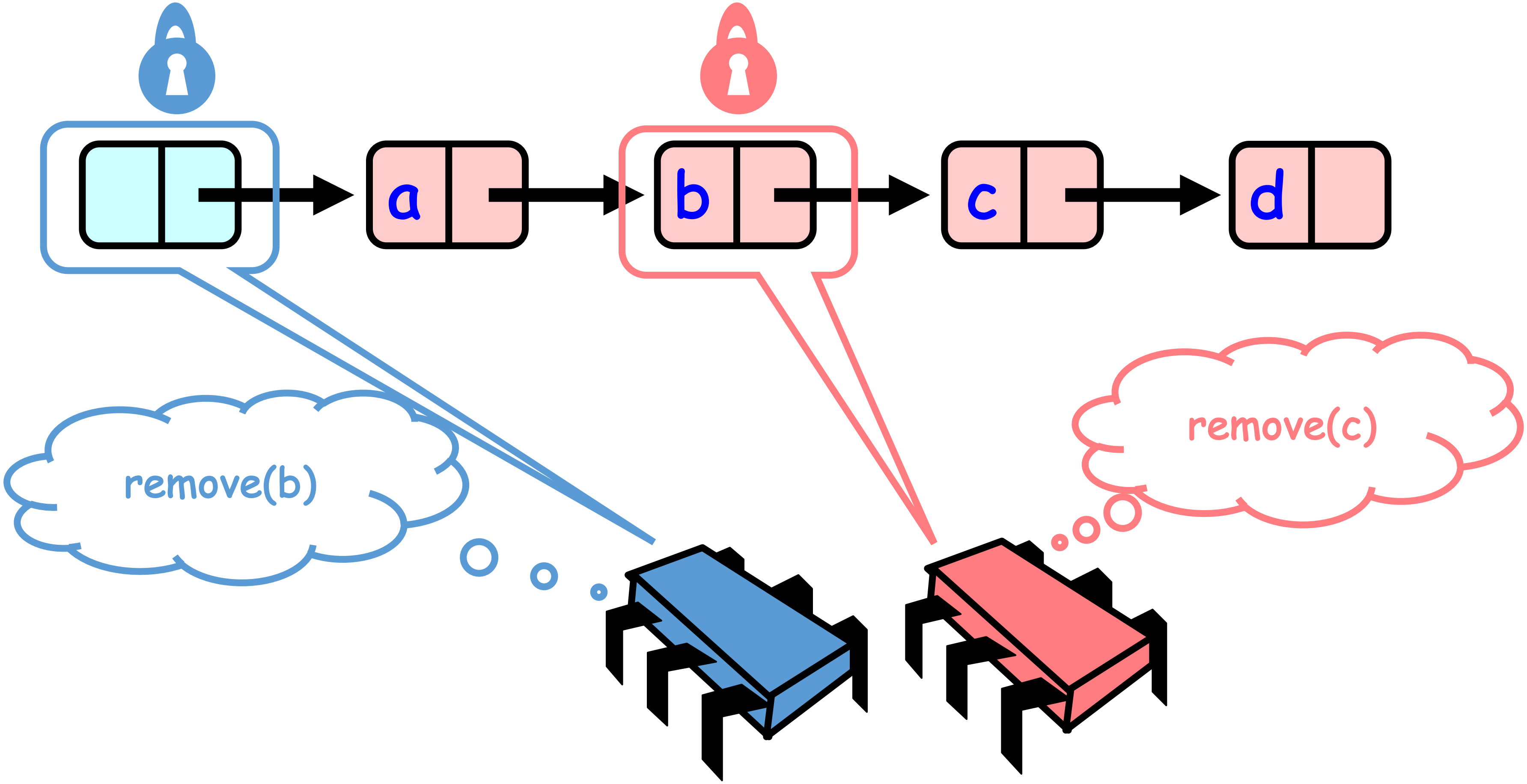
Removing a Node



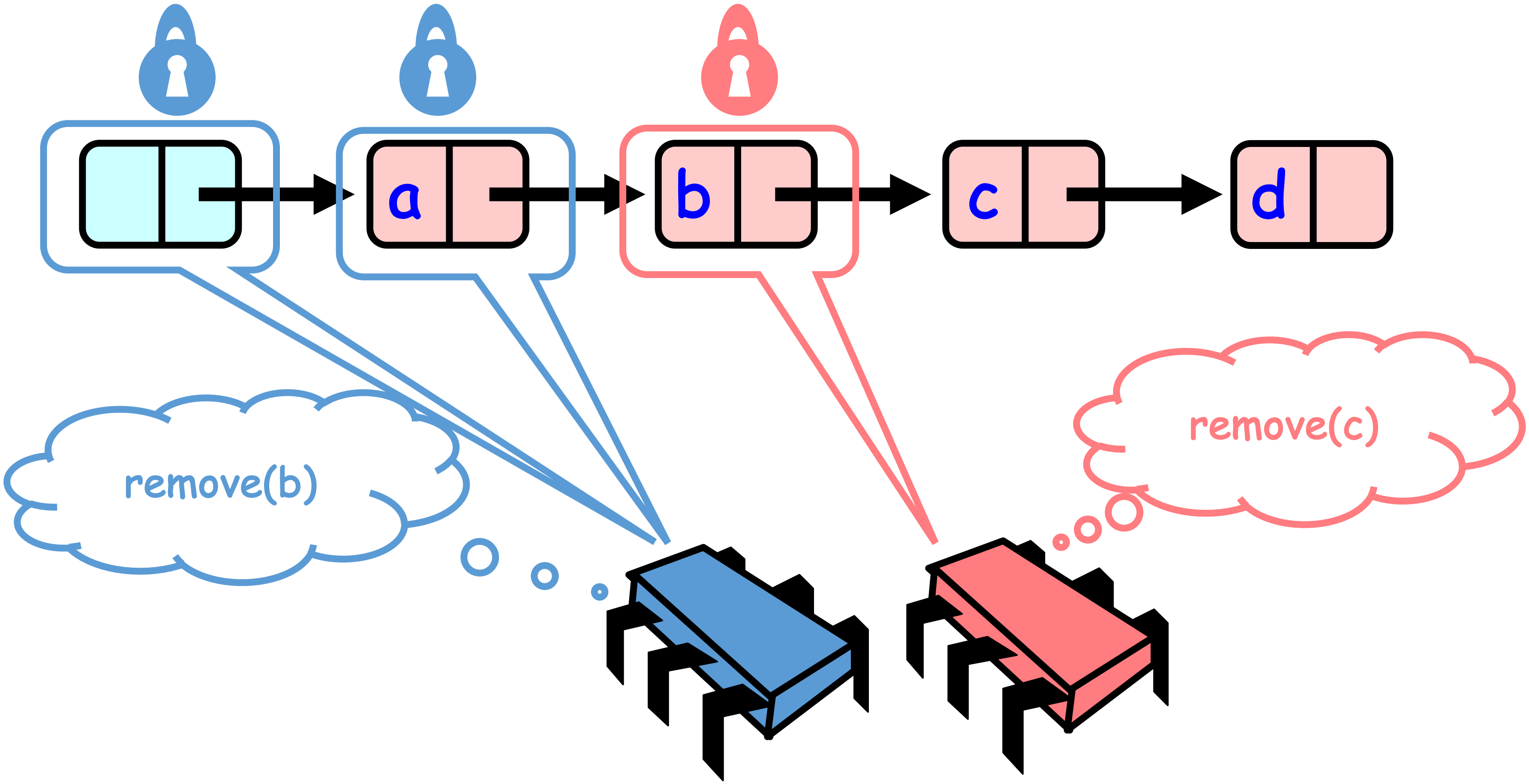
Removing a Node



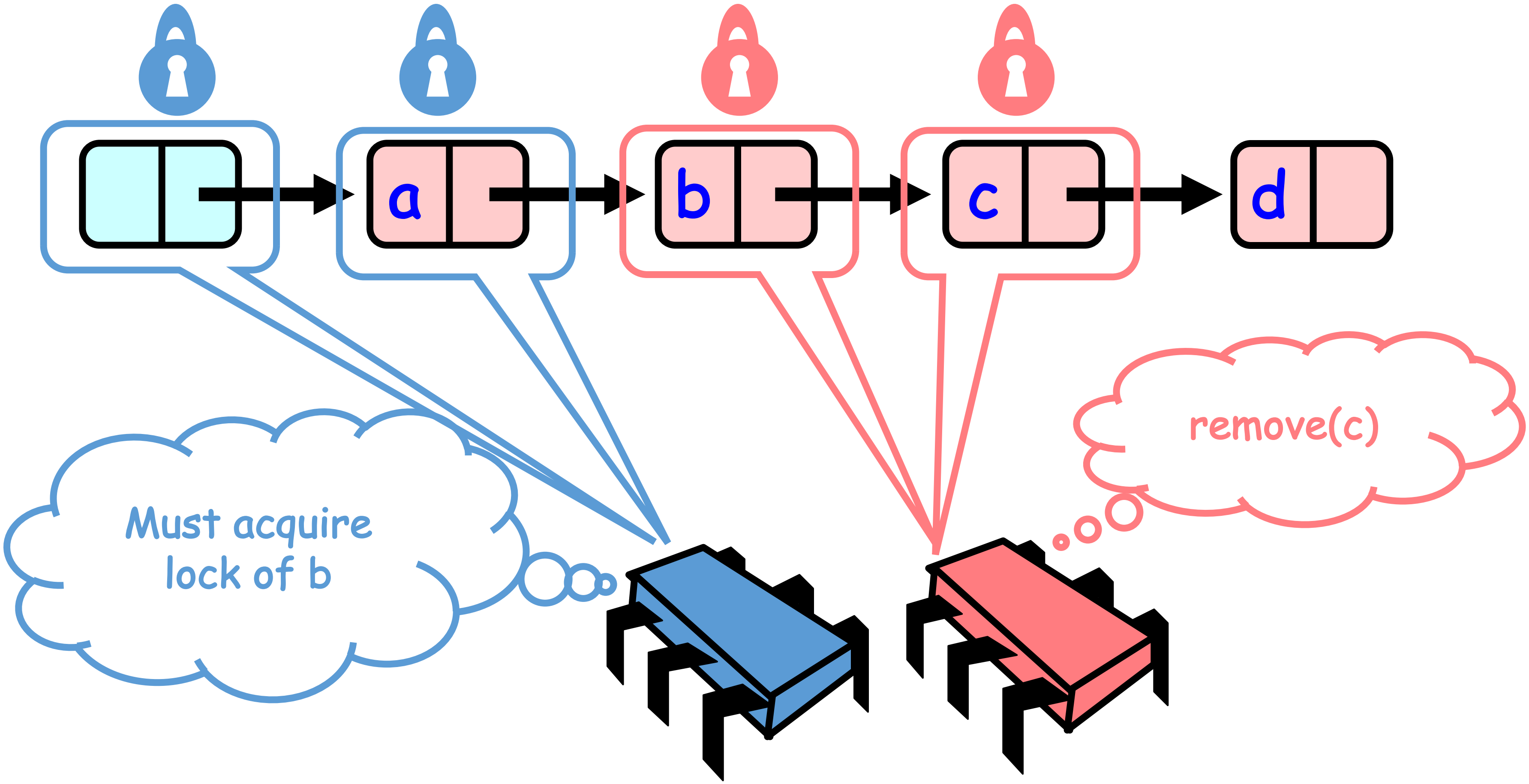
Removing a Node



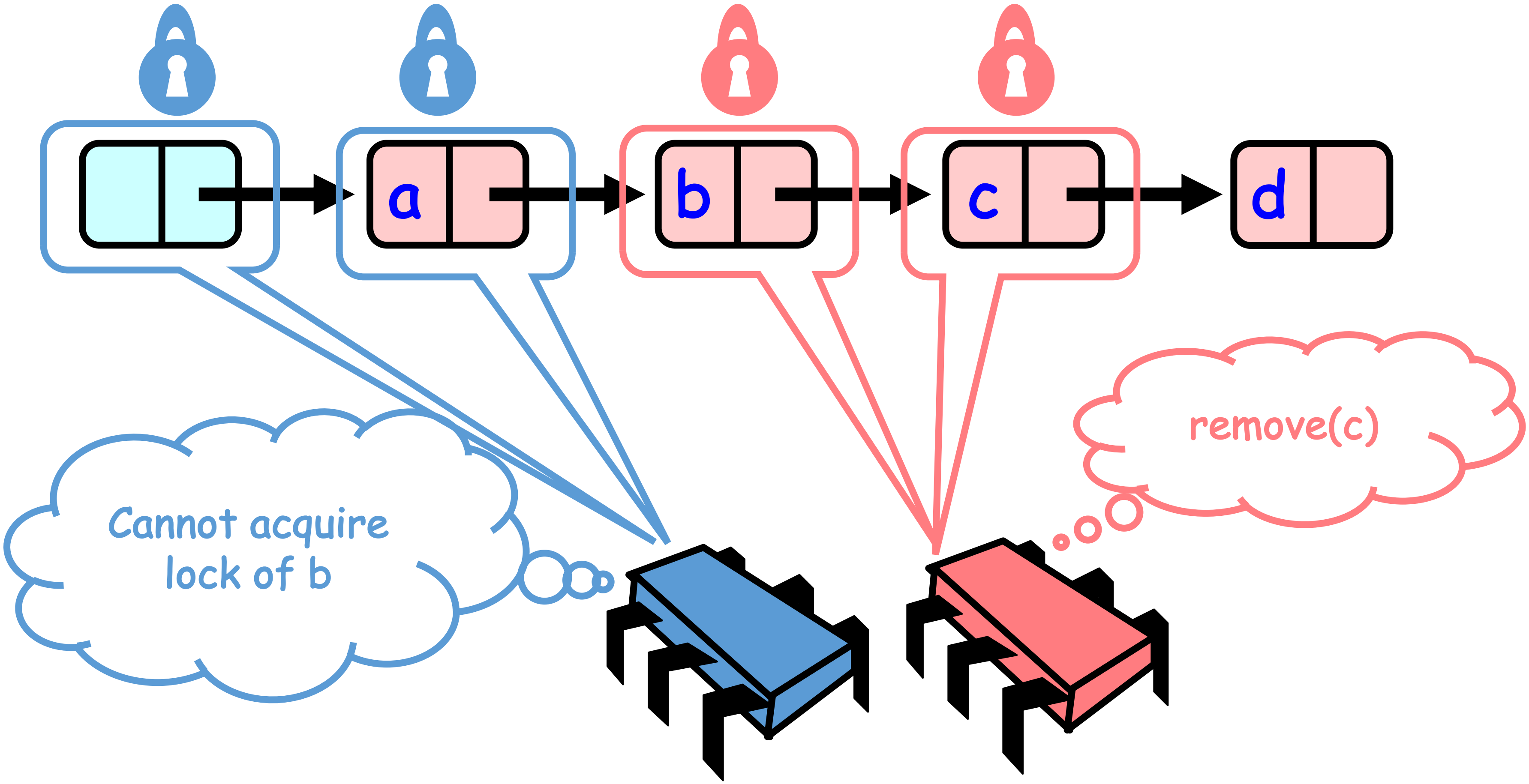
Removing a Node



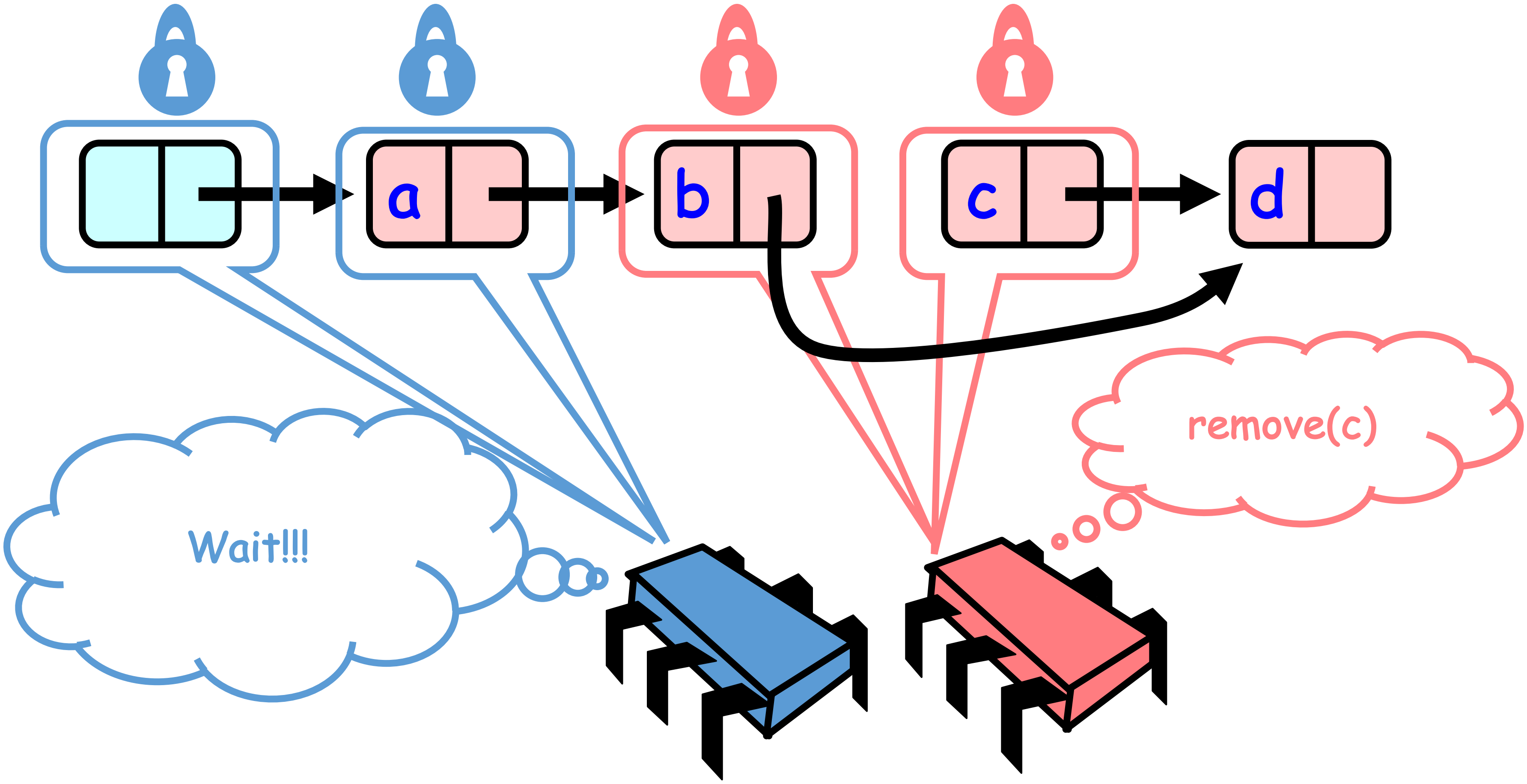
Removing a Node



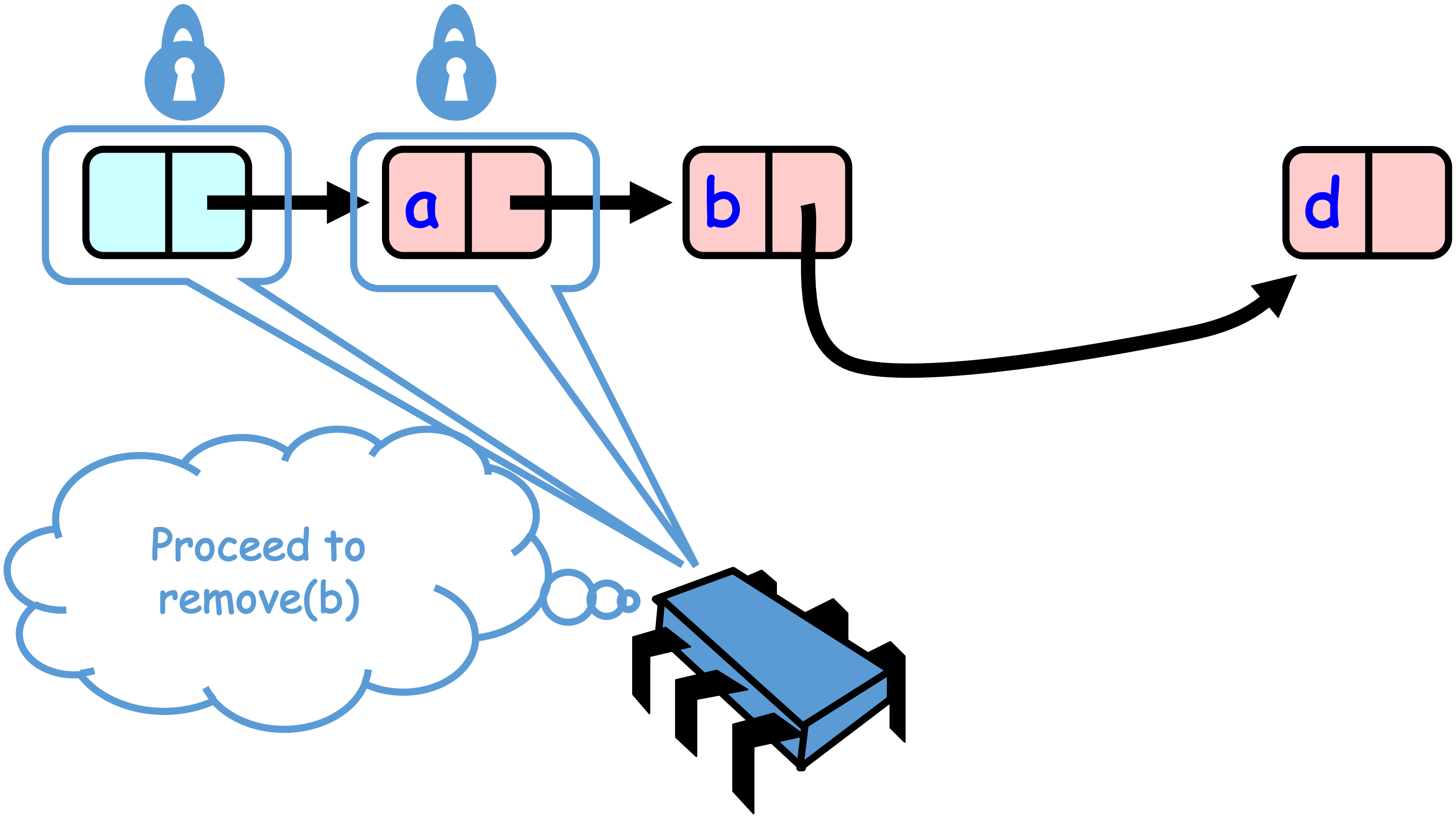
Removing a Node



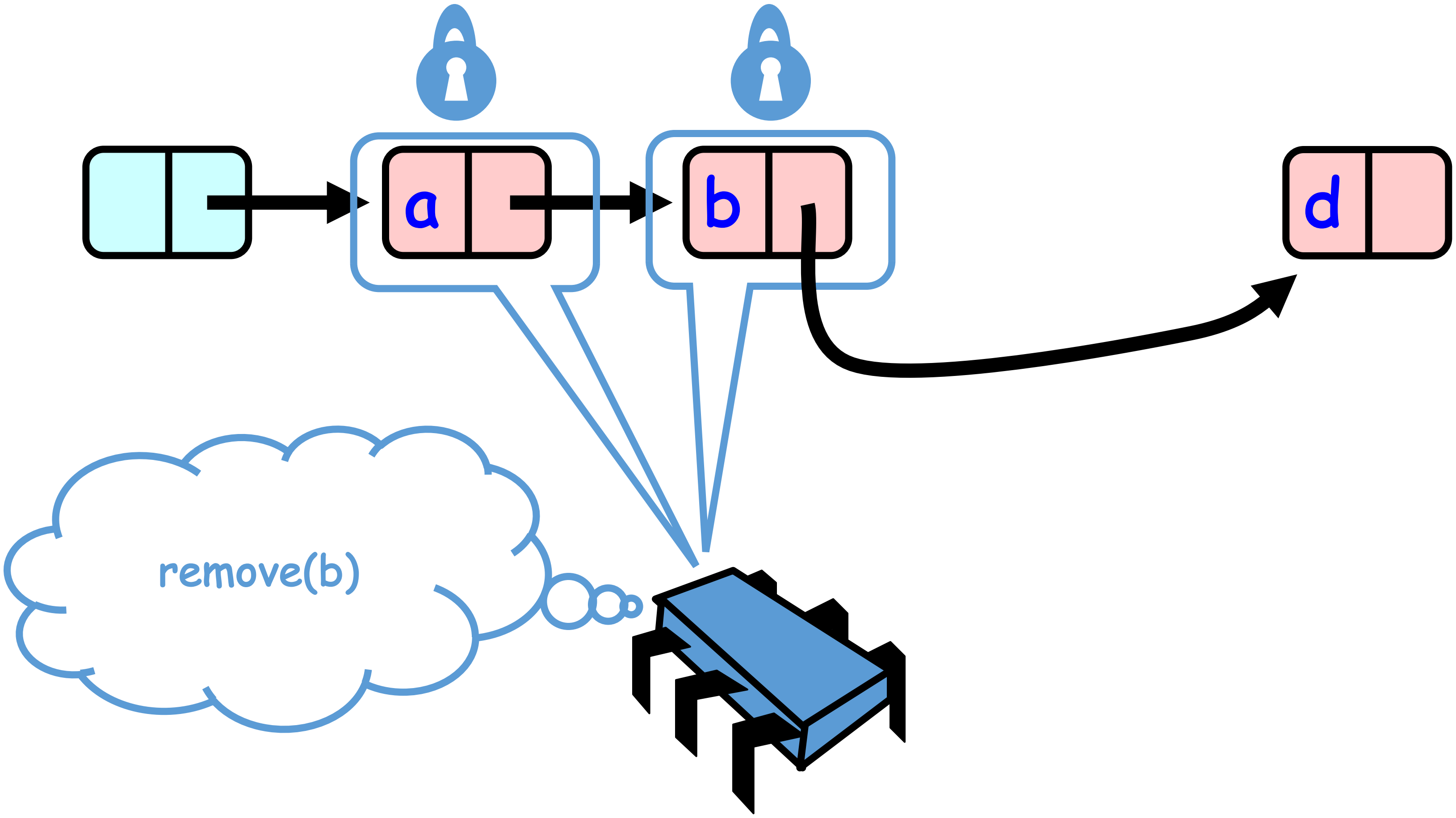
Removing a Node



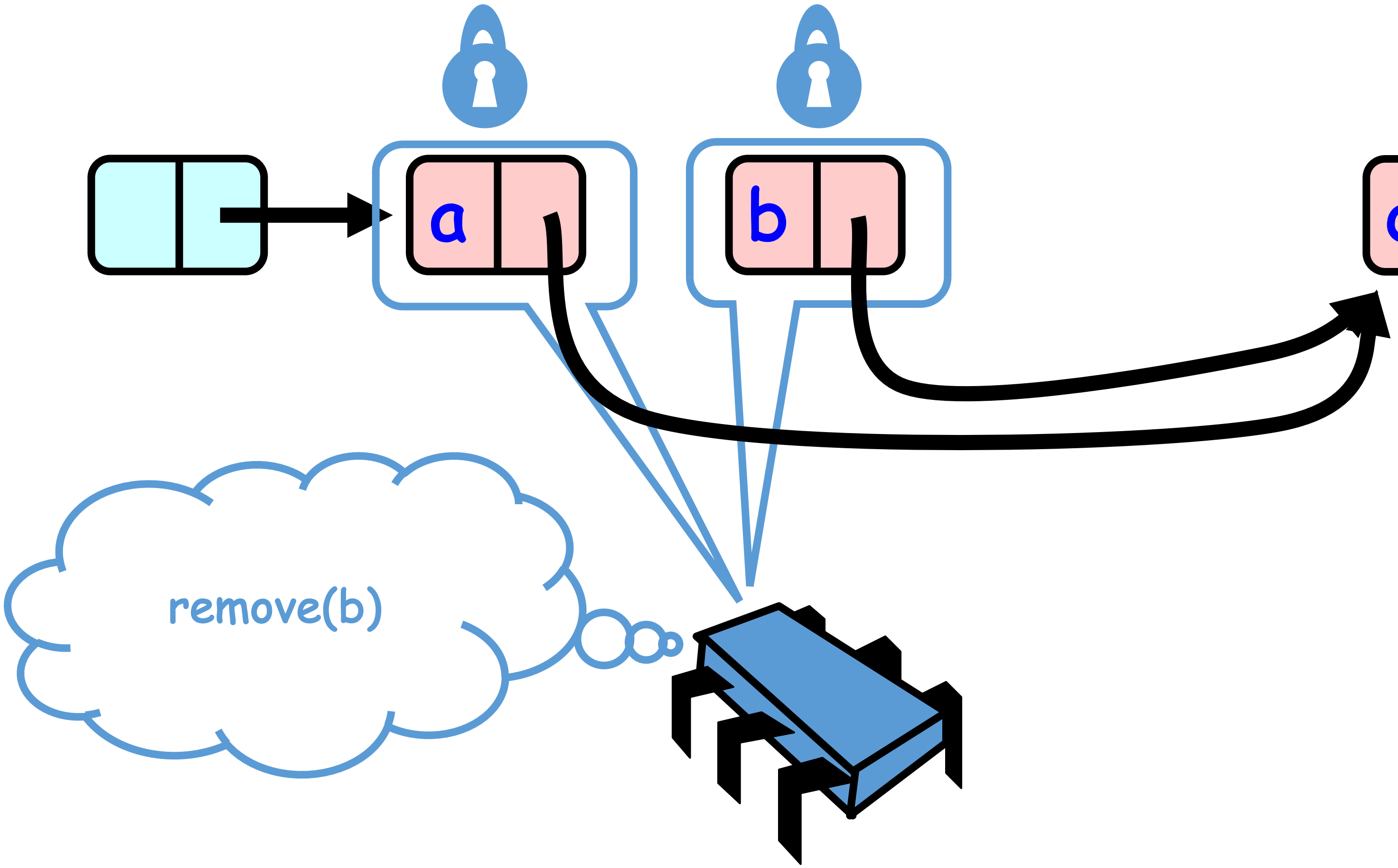
Removing a Node



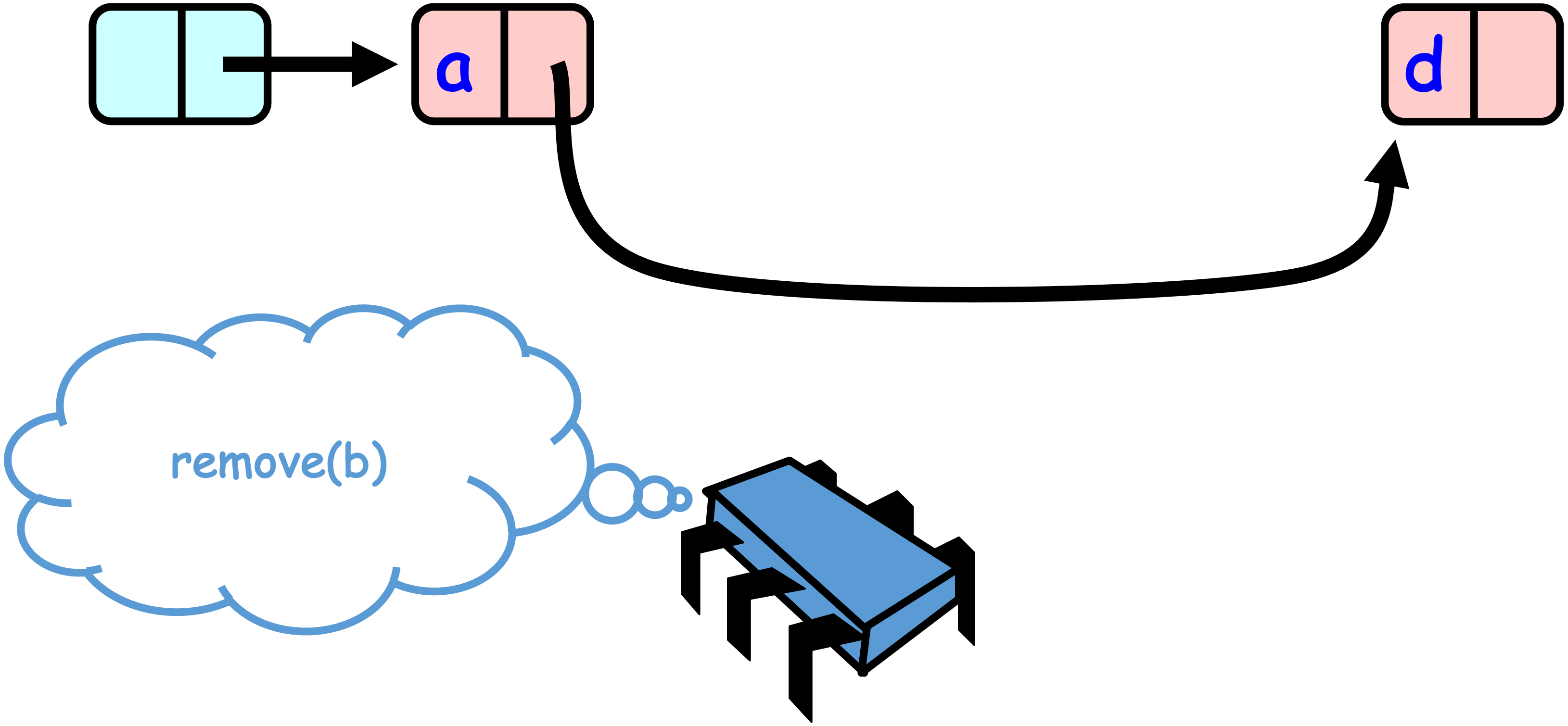
Removing a Node



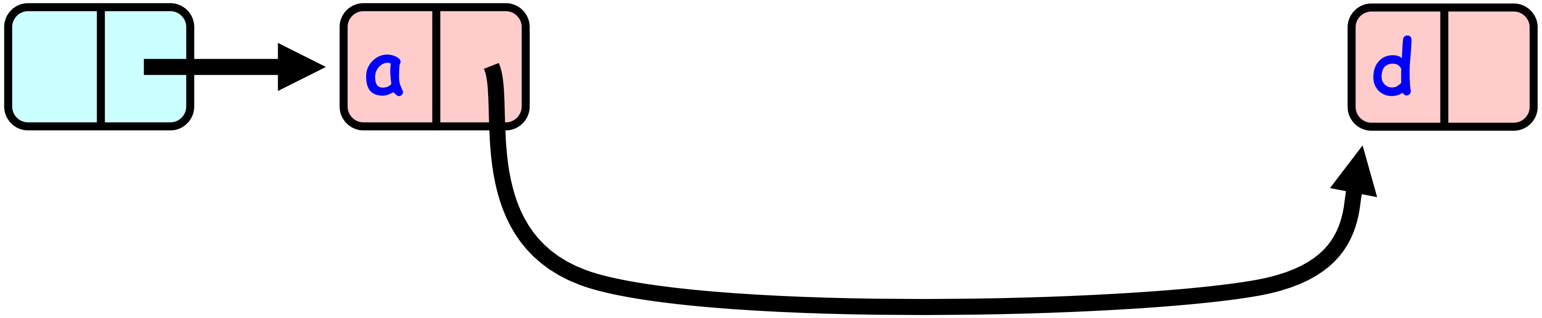
Removing a Node



Removing a Node



Removing a Node



Remove method

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

Remove method

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

Key used to order node

Remove method

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  currNode.unlock();
  predNode.unlock();
 }}
```

Predecessor and current nodes

Remove method

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

Make sure locks released

Remove method

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

Everything else

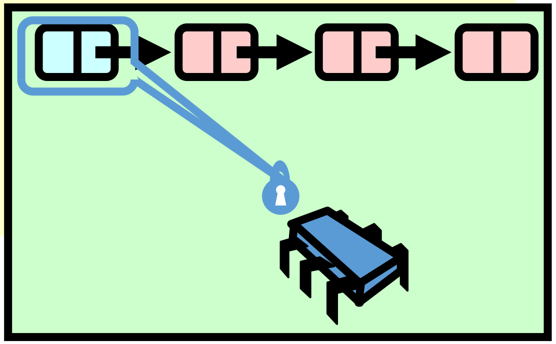
Remove method

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

Remove method

lock pred == head

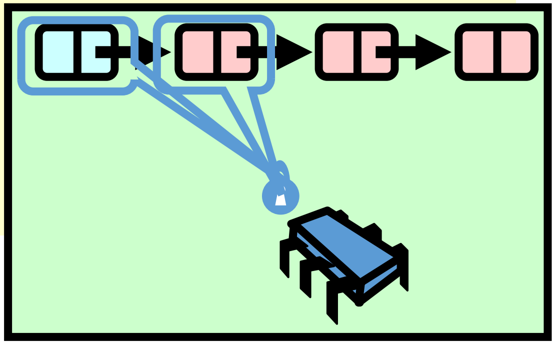
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```



Remove method

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

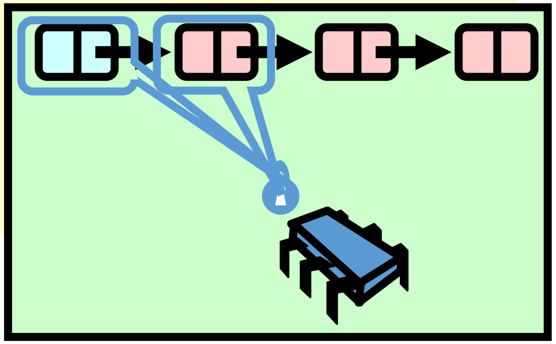
Lock current



Remove method

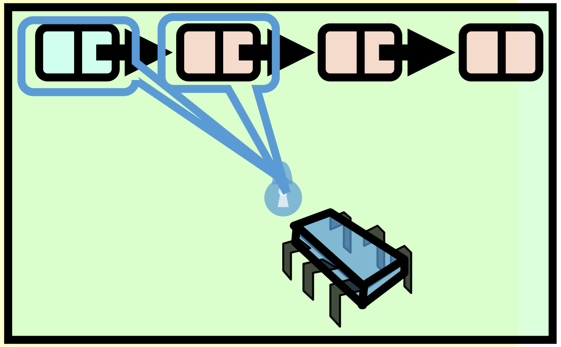
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 ...
} finally { … }
```

Traversing list



Remove: searching

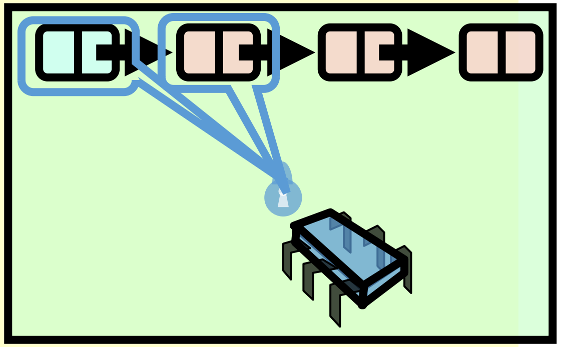
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

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

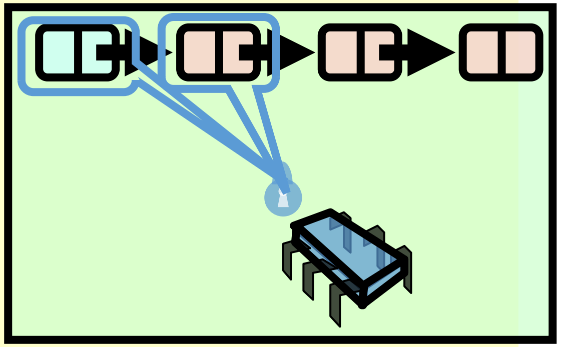
Search key range



Remove: searching

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

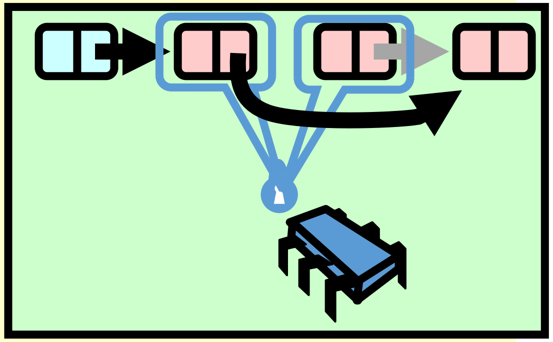
Lock invariant: At start of each loop: curr and pred locked



Remove: searching

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

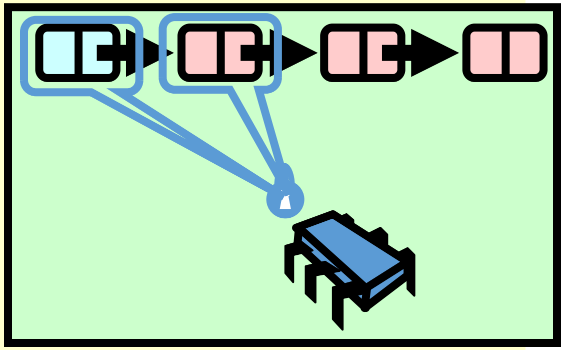
If item found, remove node



Remove: searching

Unlock predecessor

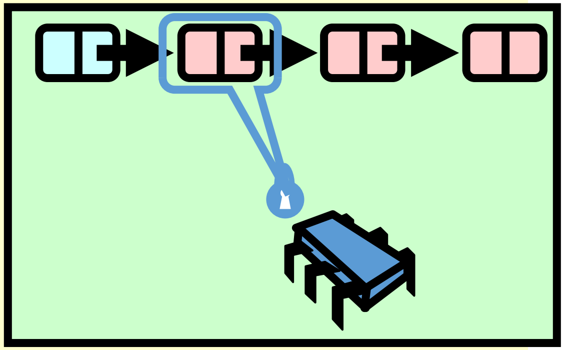
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

Only one node locked!

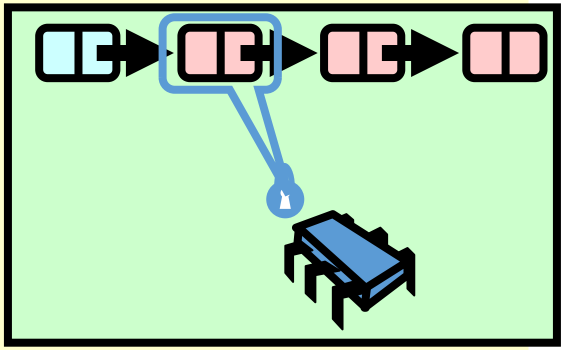
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

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

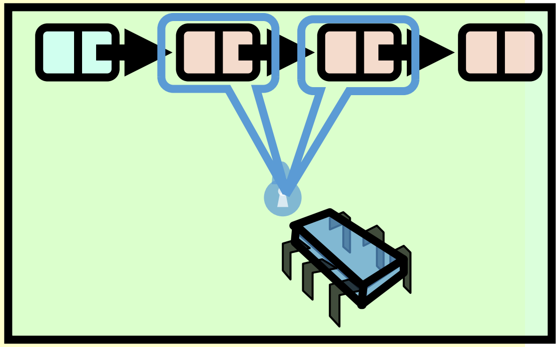
demote current



Remove: searching

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = currNode;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

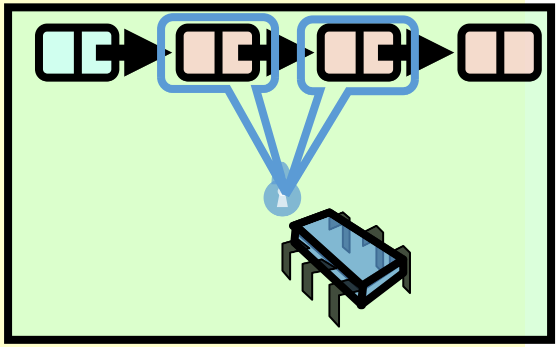
Find and lock new current



Remove: searching

Lock invariant restored

```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = currNode;
  curr = curr.next;
  curr.lock();
}
return false;
```



Remove: searching

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

Otherwise, not present

The END