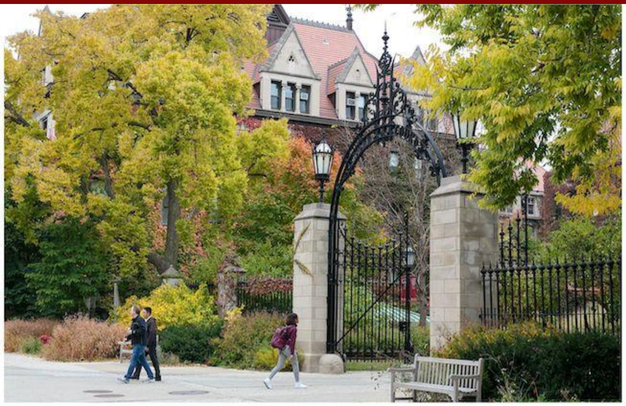
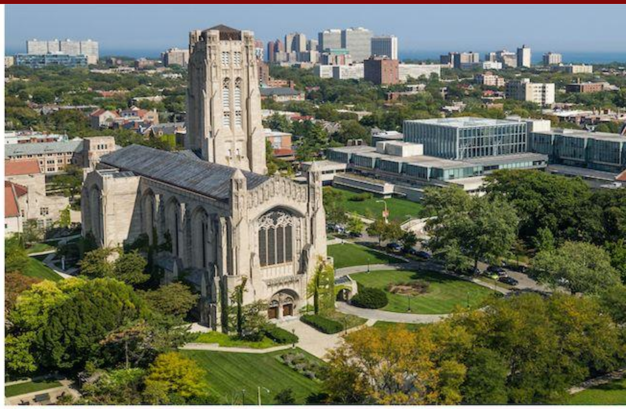
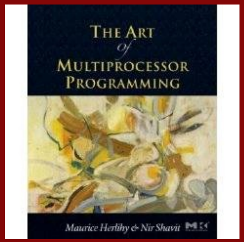
# MPCS 52060 - Parallel Programming
# M4: Concurrent Data Structures (Part 1)




Original slides from "The Art of Multiprocessor Programming" by Maurice Herlihy & Nir Shavit with modifications by  Lamont Samuels

THE UNIVERSITY OF CHICAGO | MASTERS PROGRAM IN COMPUTER SCIENCE

# More Low-Level Synchronization Primitives

# Motivation for Semaphores

- What if we wanted to control access to shared resource?
  - For example, A system can only handle a certain number of users concurrently signed on. After the maximum number of logged in users is reached, then others must wait until others logout.
- A semaphore is a synchronization primitive used to control access to a shared resource by multiple threads.
  - It has a capacity (c), which allows for having at most (c) threads in a critical section. Unlike with locks, where only one thread can be a critical section at a time.
  - You can also think of them as a way to control how many resources are available of a particular entity by allowing resources to be concurrently acquired and released in a safe way.
  - Tracks only how many resources are free; it does not keep track of which of the resources are free

# Semaphore Pseudo-Implementation

- The capacity variable of a semaphore is an integer value that cannot be directly accessed.
  - Go does not have a semaphore construct so the below examples are pseudocode similar to the implementations of semaphores in other languages:
- **Creation**: Must initialize it to some capacity integer value

```
var sema *Semaphore
// NewSemaphore allocates and
// returns a *Semaphore with its
// internal capacity initialized
sema = NewSemaphore(0)
```

- **Behaviors**: It has two main operations (methods in our case) that modify the integer capacity value

```
// Decrement semaphore
sema.Down()
// Increment semaphore
sema.Up()
```

# Semaphore Pseudo-Implementation

```
func (s *Semaphore) Down() {
  // Wait if the value of semaphore s is less than or equal to 0
  //  Decrement the value of semaphore s by one
}
func (s *Semaphore) Up() {
  //Increment the value of semaphore s by 1
  //If there are 1 or more threads waiting, wake one up
}
```

# Semaphores & Mutual Exclusion

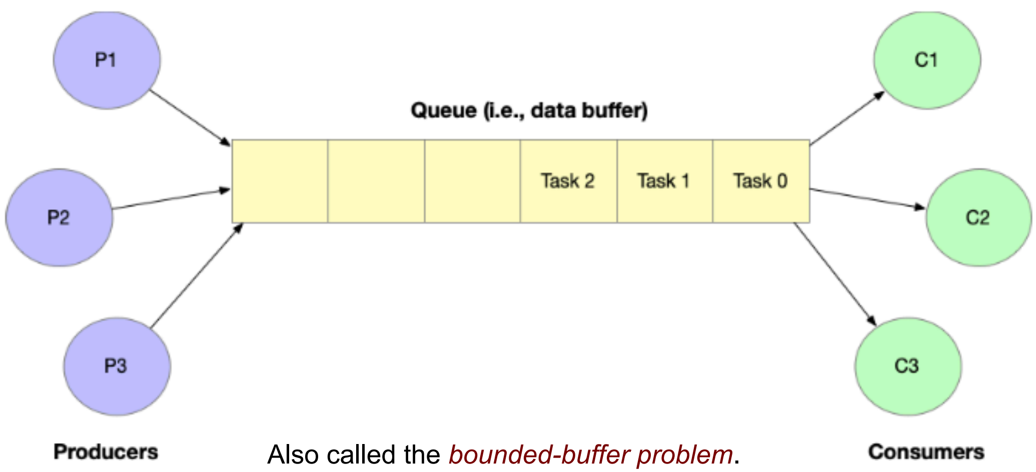
- There are two well known types of semaphores
  - Binary semaphore - acts like a mutex by setting its capacity initial value to 1.

```
var mutex_sema *Semaphore
mutex_sema = NewSemaphore(1)
mutex_sema.Down()
//critical section
mutex_sema.Up()
```

  - Counting semaphore - initialize the semaphore to be equal to the number of available resources.

# Real-World Example: Producer and Consumer Problem

- One or more threads generate tasks (producers) and one or more threads receive and process them (consumers).
- Producers and consumers communicate using a queue of maximum size N and must adhere to the following conditions
  - Consumers must wait for a producer to produce a task if the queue is empty.
  - Producer must wait for the consumer to consume a task if the queue is full.

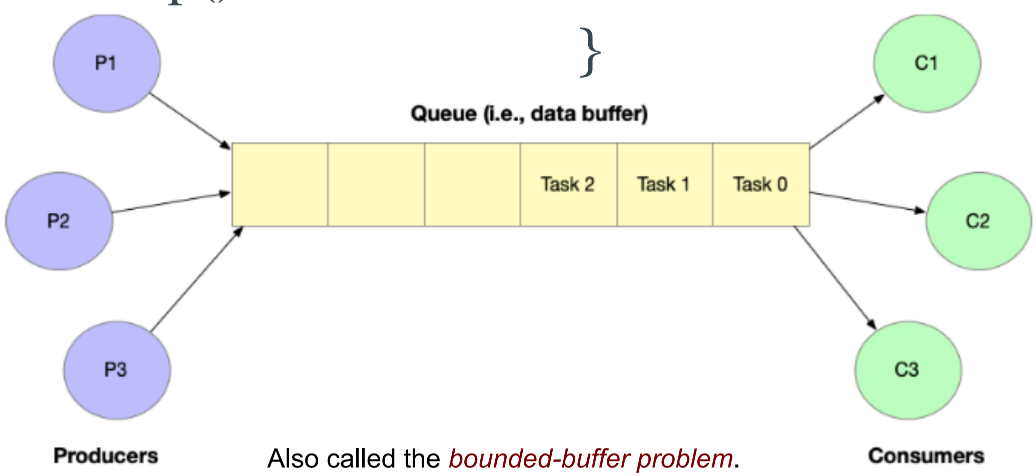



Also called the *bounded-buffer problem*.

# Real-World Example: Producer and Consumer Problem

```
//Producer
for {
   //Generate Task
   sema_emptyCount.Down()
   sema_mutex.Down()
   //Put task in Queue
   sema_mutex.Up()
   sema_fullCount.Up()
}
```

```
//Consumer
for {
   sema_fullCount.Down()
   sema_mutex.Down()
   //Remove task from Queue
   sema_mutex.Up()
   sema_emptyCount.Up()
   //Process Task
}
```




Also called the *bounded-buffer problem*.

# Condition Variables

- A data object that allows a thread to suspend execution until a certain event or condition occurs.
- When the event or condition occurs another thread can signal the thread to “wake up.”
- A condition variable is always associated with a mutex.

```
// lock mutex
if (condition has occurred) {
   // signal thread(s);
}else {
   // 1. Wait until another thread signals to wake up by unlocking
   // the mutex and block (e.g., sleep, or spin etc.);
   // 2. After the signal happens then the thread wakes, requires the lock
   // and checks to make sure the condition is still true.
}
// unlock mutex
```

# Condition Variables in Go

- sync.Cond represents conditional variables in Go.
- **Creation:** NewCond(l Locker) *Cond
- **Operations** on condition variables:
  - func (c *Cond) Wait(): suspends the calling thread and releases the monitor lock. When it resumes, reacquire the lock. Called when condition is not true.
  - func (c *Cond) Signal(): resumes one thread waiting in wait() if any. Called when condition becomes true and wants to wake up one waiting thread.
  - func (c *Cond) Broadcast(): resumes all threads waiting in wait(). Called when condition becomes true and wants to wake up all waiting threads.

# Demo: Condition Variables

# Concurrent Data Structures

# Concurrent Data Structures

- We assume
  - shared-memory multiprocessors environment
  - concurrently execute multiple threads which communicate and synchronize through data structures in shared memory

# Concurrent Data Structures

- Far more difficult to design than sequential ones
  - Correctness
    - Primary source of difficulty is concurrency
    - The steps of different threads can be interleaved arbitrarily
  - Scalability (performance)
- We will look at
  - Concurrent Linked List/Queue/Stack

# Main performance issue of lock based system

- Sequential bottleneck
  - At any point in time, at most one lock-protected operation is doing useful work.
- Memory contention
  - Overhead in traffic as a result of multiple threads concurrently attempting to access the same memory location.
- Blocking
  - If thread that currently holds the lock is delayed, then all other threads attempting to access are also delayed.
  - Implementation of locks is known as a blocking algorithm
  - Consider non-blocking (lock-free) algorithm

# Nonblocking algorithms

- implemented by a hardware operation
  - atomically combines a load and a store
  - Ex) compare-and-swap(CAS)
- lock-free
  - if there is guaranteed system-wide progress;
  - while a given thread might be blocked by other threads, all CPUs can continue doing other useful work without stalls.
- wait-free
  - if there is also guaranteed per-thread progress.
  - in addition to all CPUs continuing to do useful work, no computation can ever be blocked by another computation.

# 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
  - remove(x) take x out of set
  - contains(x) tests if x in set

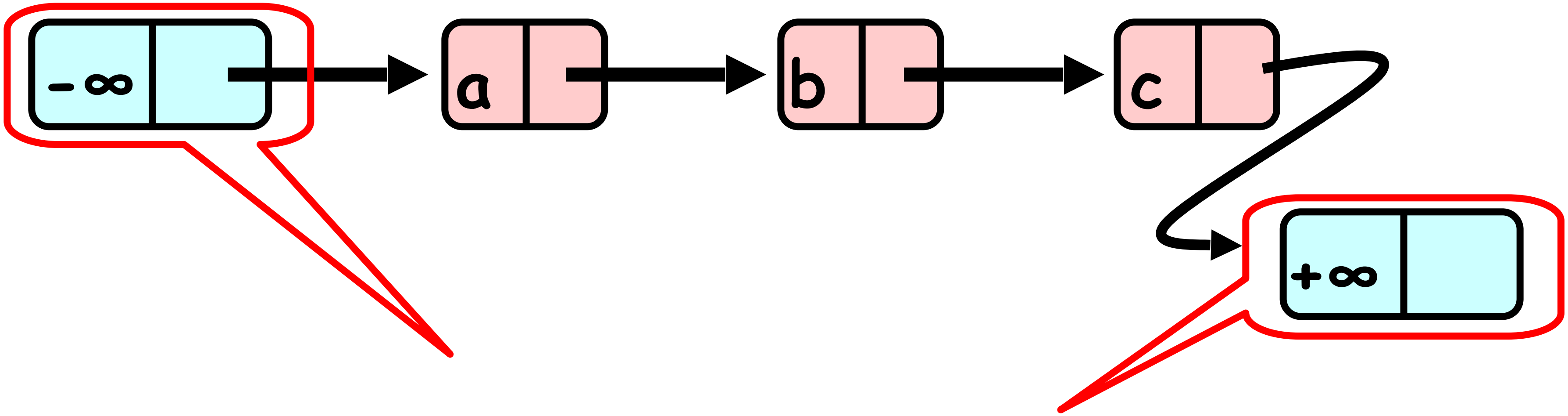
# List-Based Sets

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

# List Node

```
public class Node {
 public T item;      // item of interest
 public int key;     // usually hash code
 public Node next;   // reference to next node
}
```

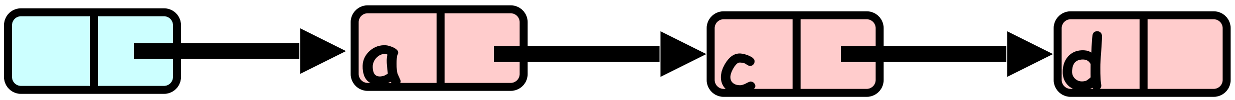
# The List-Based Set



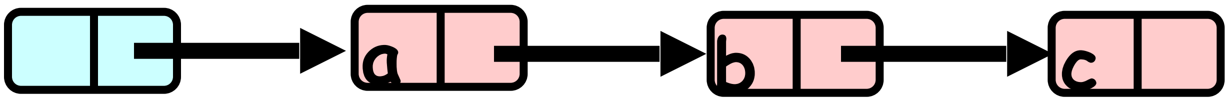

Sorted with Sentinel nodes
(min & max possible keys)

# Sequential List Based Set

Add()

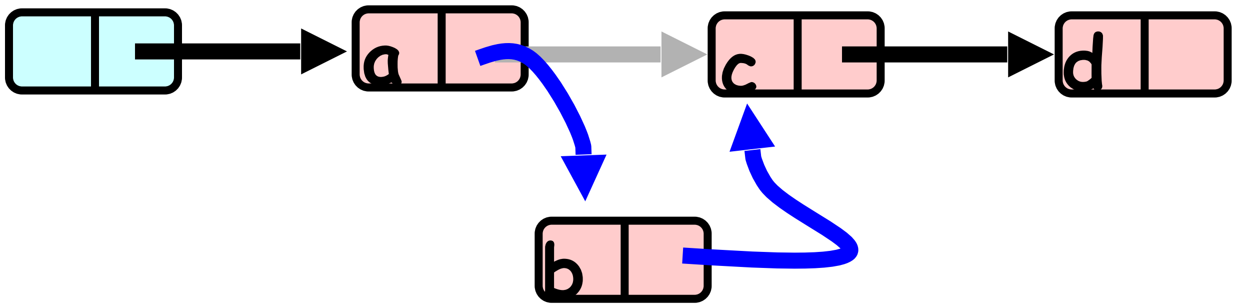



Remove()

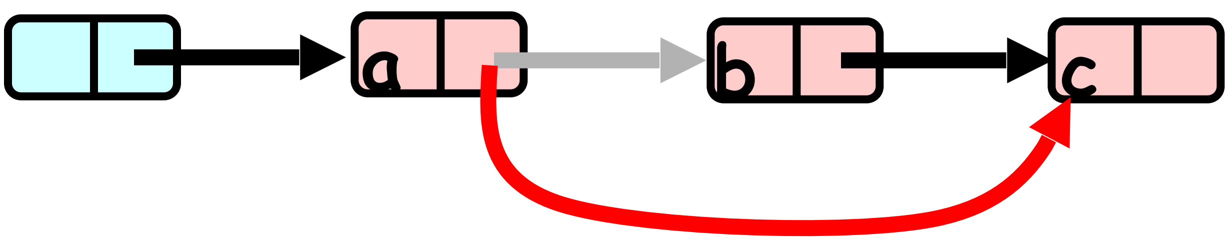


# Sequential List Based Set

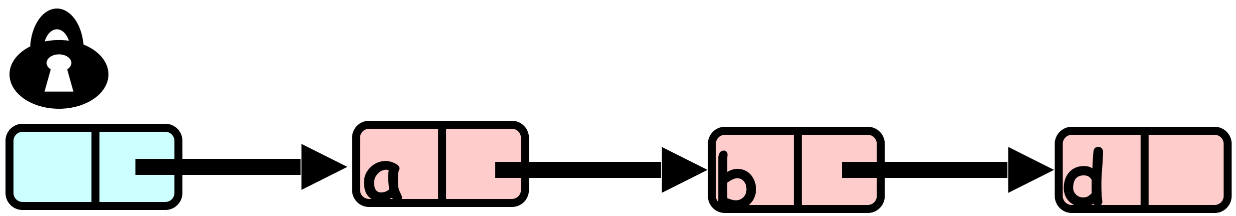
Add()



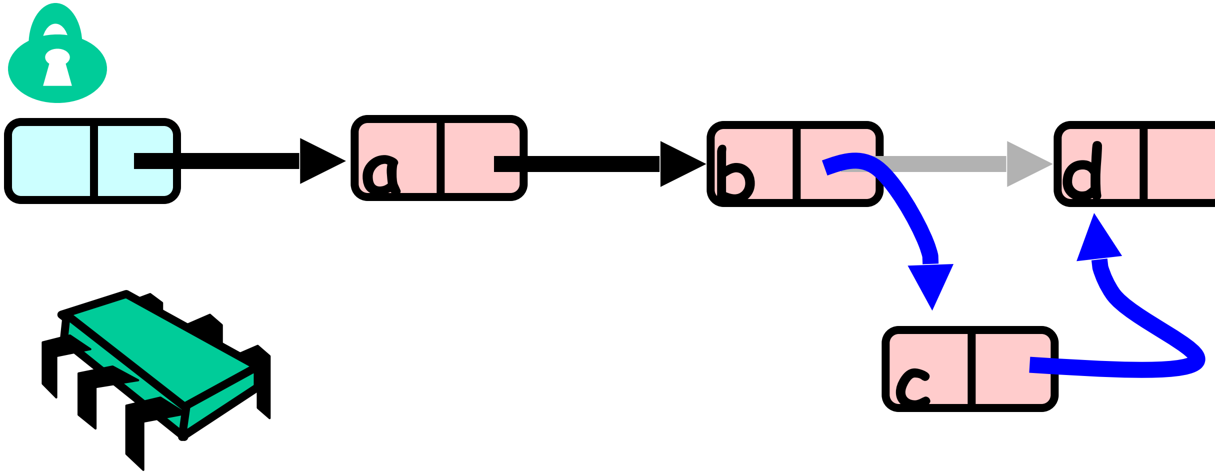

Remove()



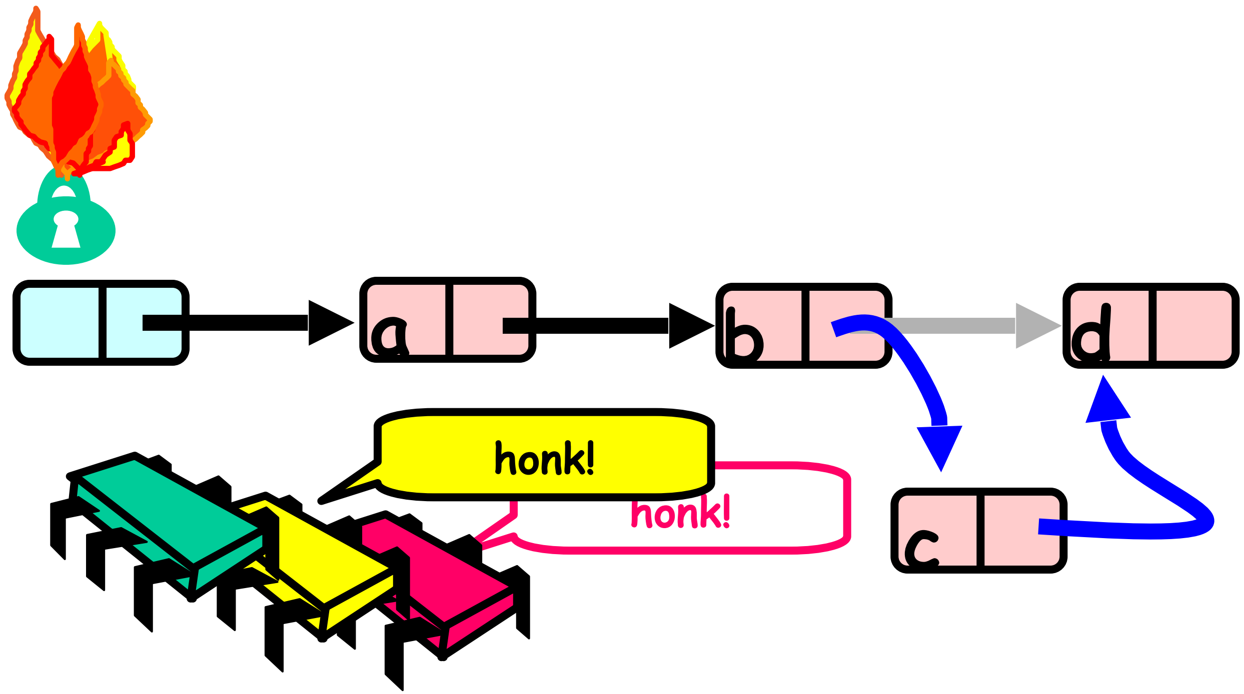
# Course Grained Locking



# Course Grained Locking



# Course Grained Locking




Simple but hotspot + bottleneck

# 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 inherently parallel …

# Coarse-Grained Synchronization (Linked List)

```
public class CoarseList<T> {
private Node head;
private Node tail;
private Lock lock = new ReentrantLock();

public CoarseList() {
    // Add sentinels to start and end
    head  = new Node(Integer.MIN_VALUE);
    tail  = new Node(Integer.MAX_VALUE);
    head.next = this.tail;
  }
```

```
public boolean add(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();
    try {
      pred = head;
      curr = pred.next;
      while (curr.key < key) {
        pred = curr;
        curr = curr.next;
      }
      if (key == curr.key) {
        return false;
      } else {
        Node node = new Node(item);
        node.next = curr;
        pred.next = node;
        return true;
      }
    } finally {
      lock.unlock();
    }
  }
```

```java
public boolean remove(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();
    try {
      pred = this.head;
      curr = pred.next;
      while (curr.key < key) {
        pred = curr;
        curr = curr.next;
      }
      if (key == curr.key)
        pred.next = curr.next;
        return true;
      } else {
        return false;
      }
    } finally {
      lock.unlock();
    }
  }
```

```java
public boolean contains(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();
    try {
      pred = head;
      curr = pred.next;
      while (curr.key < key) {
        pred = curr;
        curr = curr.next;
      }
      return (key == curr.key);
    } finally {
      lock.unlock();
    }
  }
```

# Coarse-Grained Locking

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

# Performance Improvement

- For highly-concurrent objects
- Goal:
  - Concurrent access
  - More threads, more throughput

# First: 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

# Second: Optimistic Synchronization

- Search without locking …
- If you find it, lock and check …
  - OK: we are done
  - Oops: start over
- Evaluation
  - Usually cheaper than locking
  - Mistakes are expensive

# Third: Lazy Synchronization

- Postpone hard work
- Removing components is tricky
  - Logical removal
    - Mark component to be deleted
  - Physical removal
    - Do what needs to be done

# Fourth:
# Lock-Free Synchronization

- Don't use locks at all
  - Use compareAndSet() & relatives …
- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead

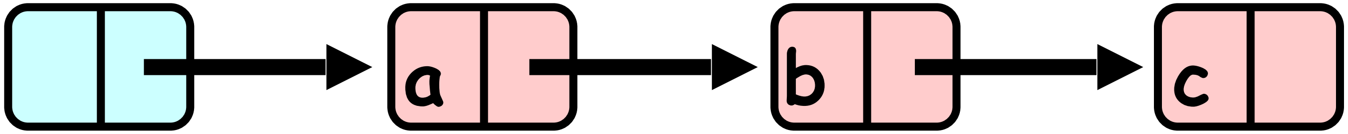
# Fine-grained Locking

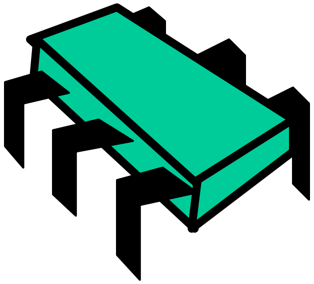
- Requires **careful** thought
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

# Fine-grained Locking

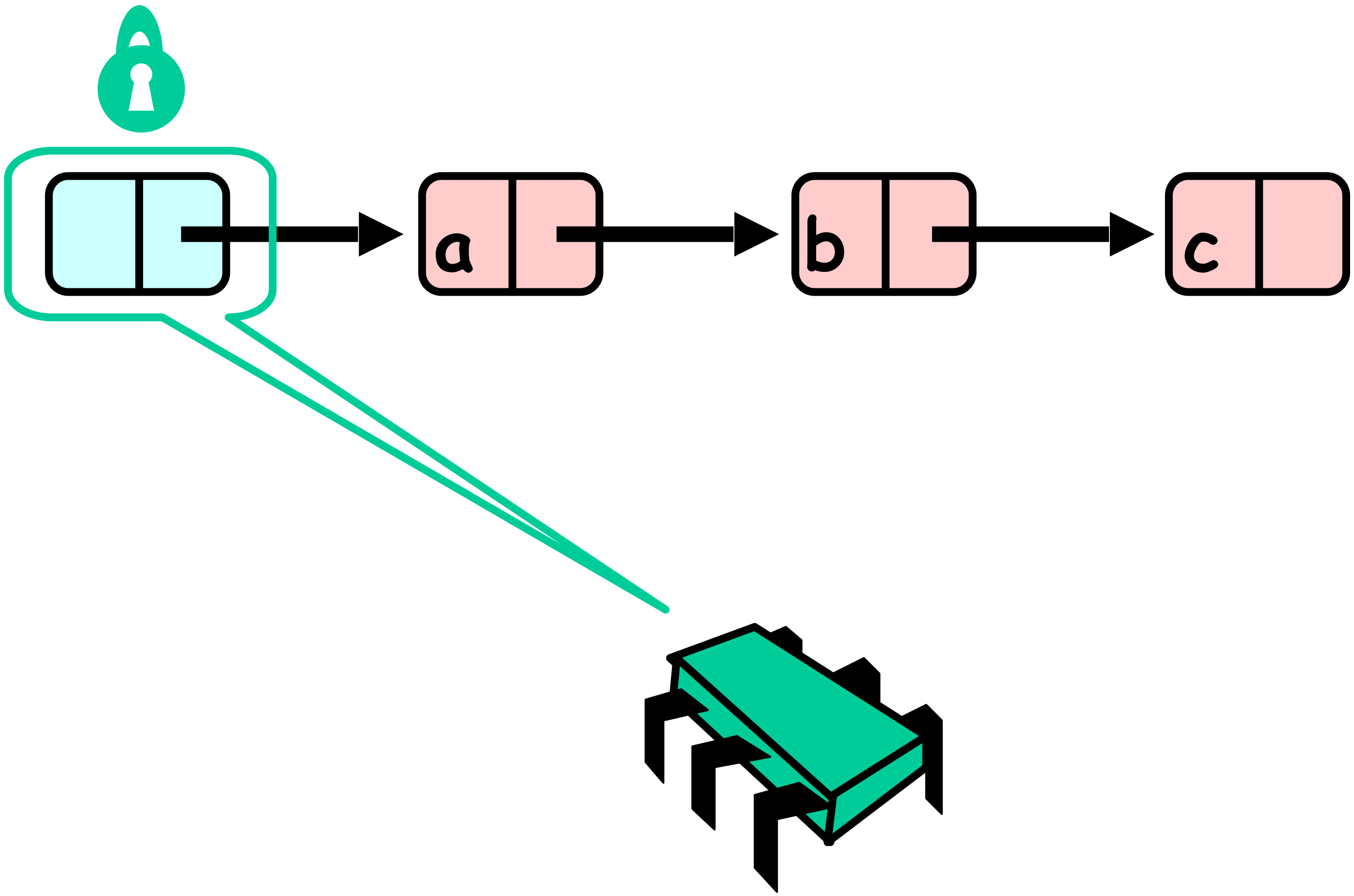
- Use multiple locks of small granularity to protect different parts of the data structure
- Goal
  - To allow concurrent operations to proceed in parallel when they do not access the same parts of the data structure

# Hand-over-Hand locking

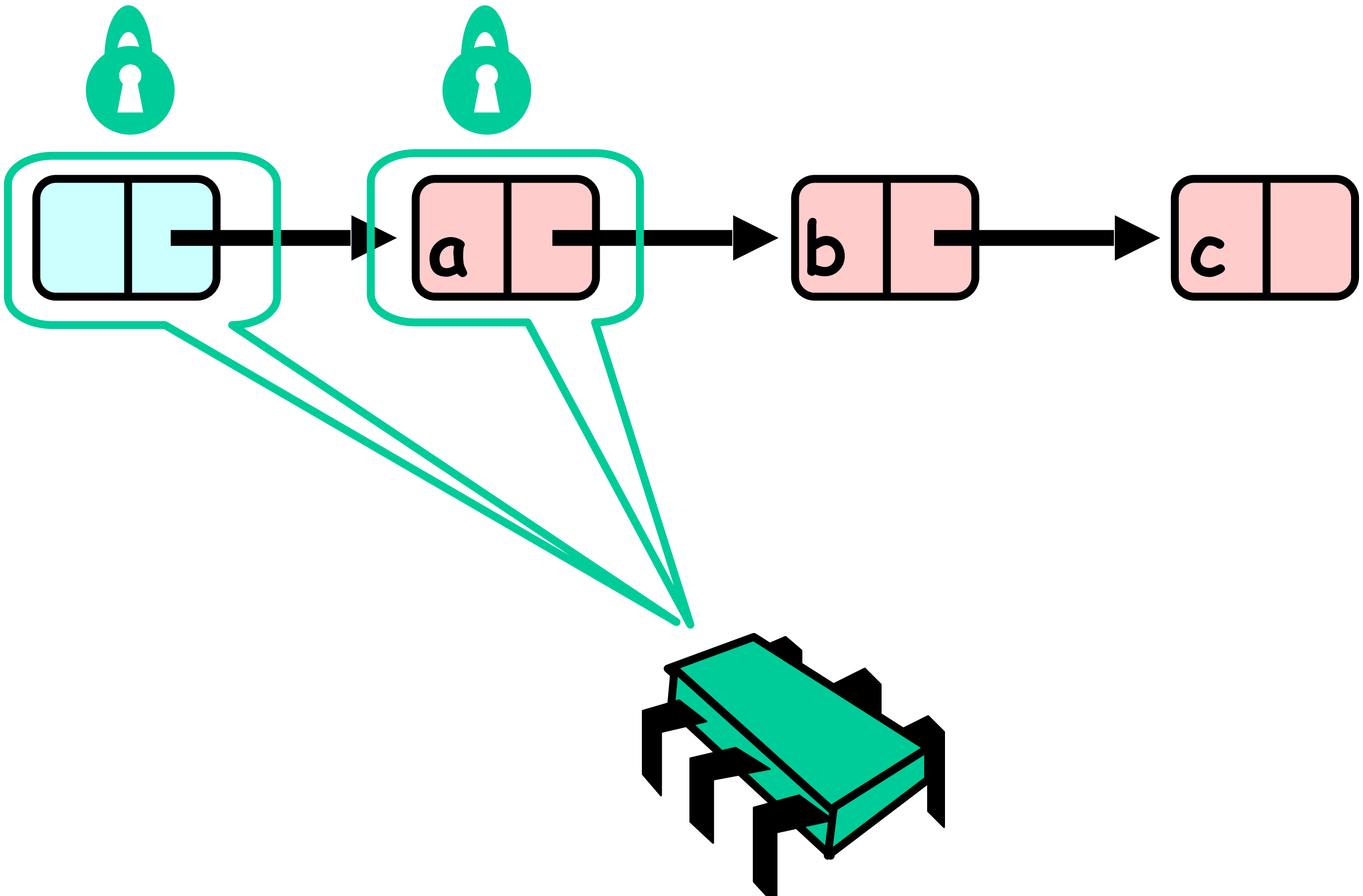


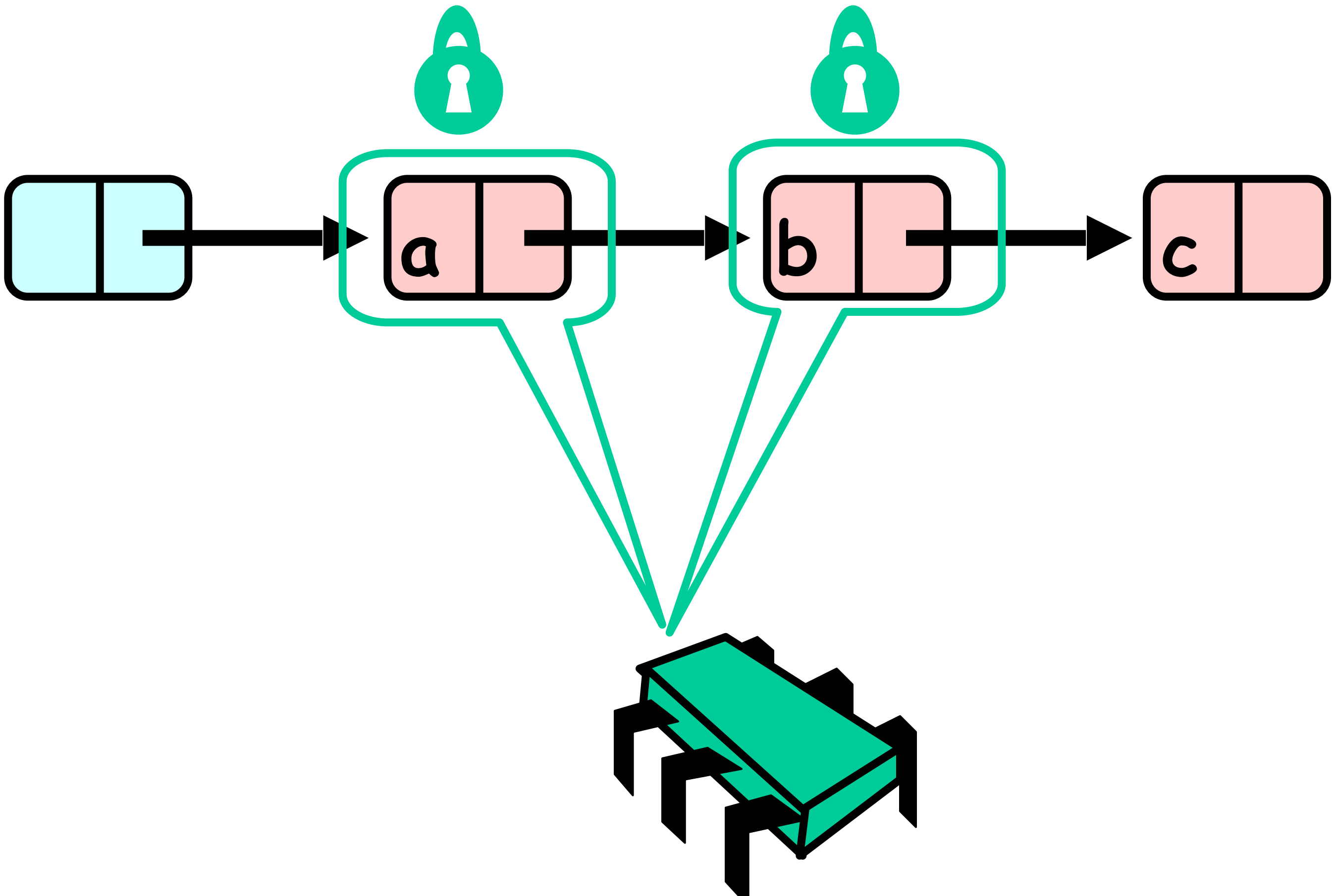
# Hand-over-Hand locking



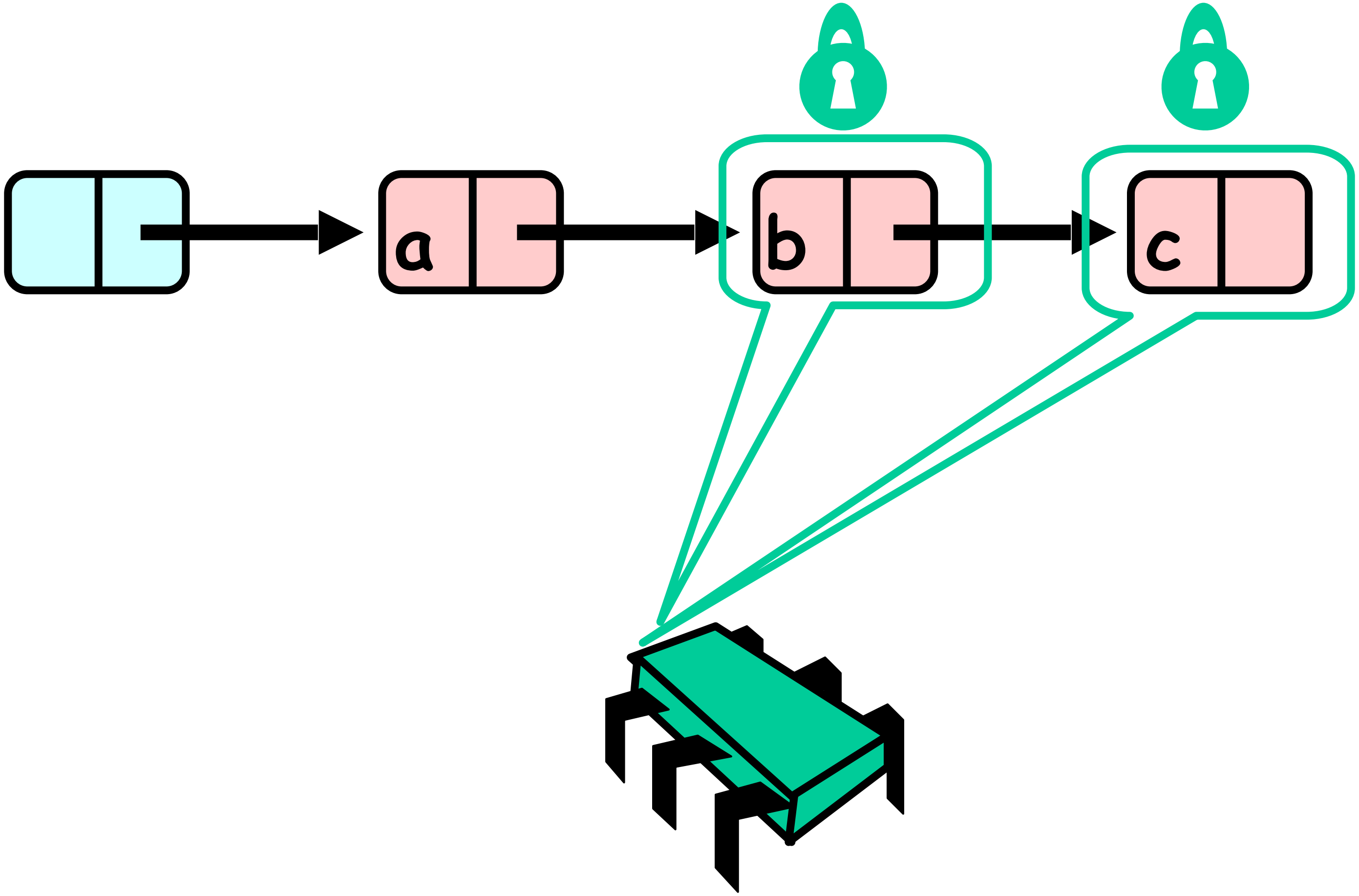
# Hand-over-Hand locking



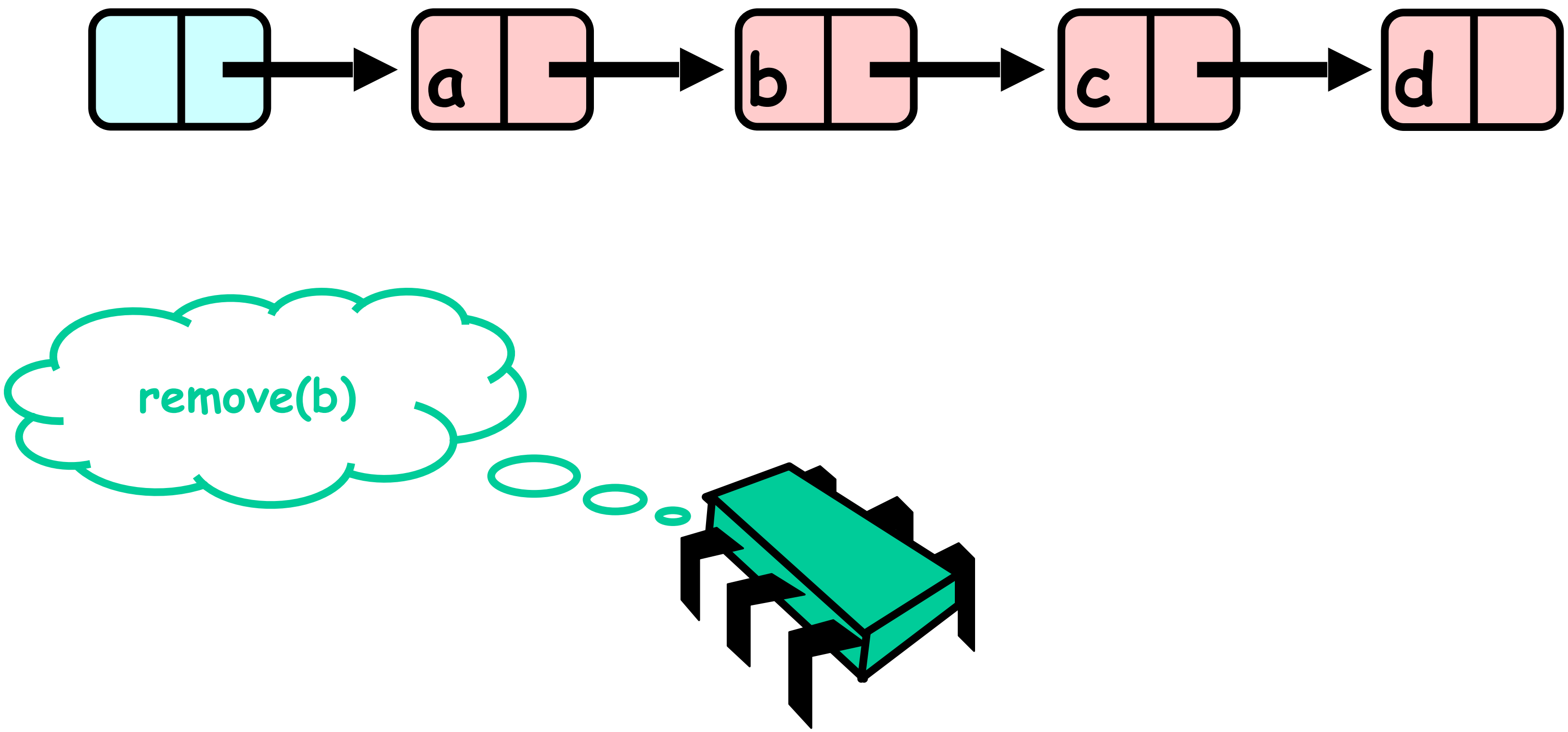
# Hand-over-Hand locking



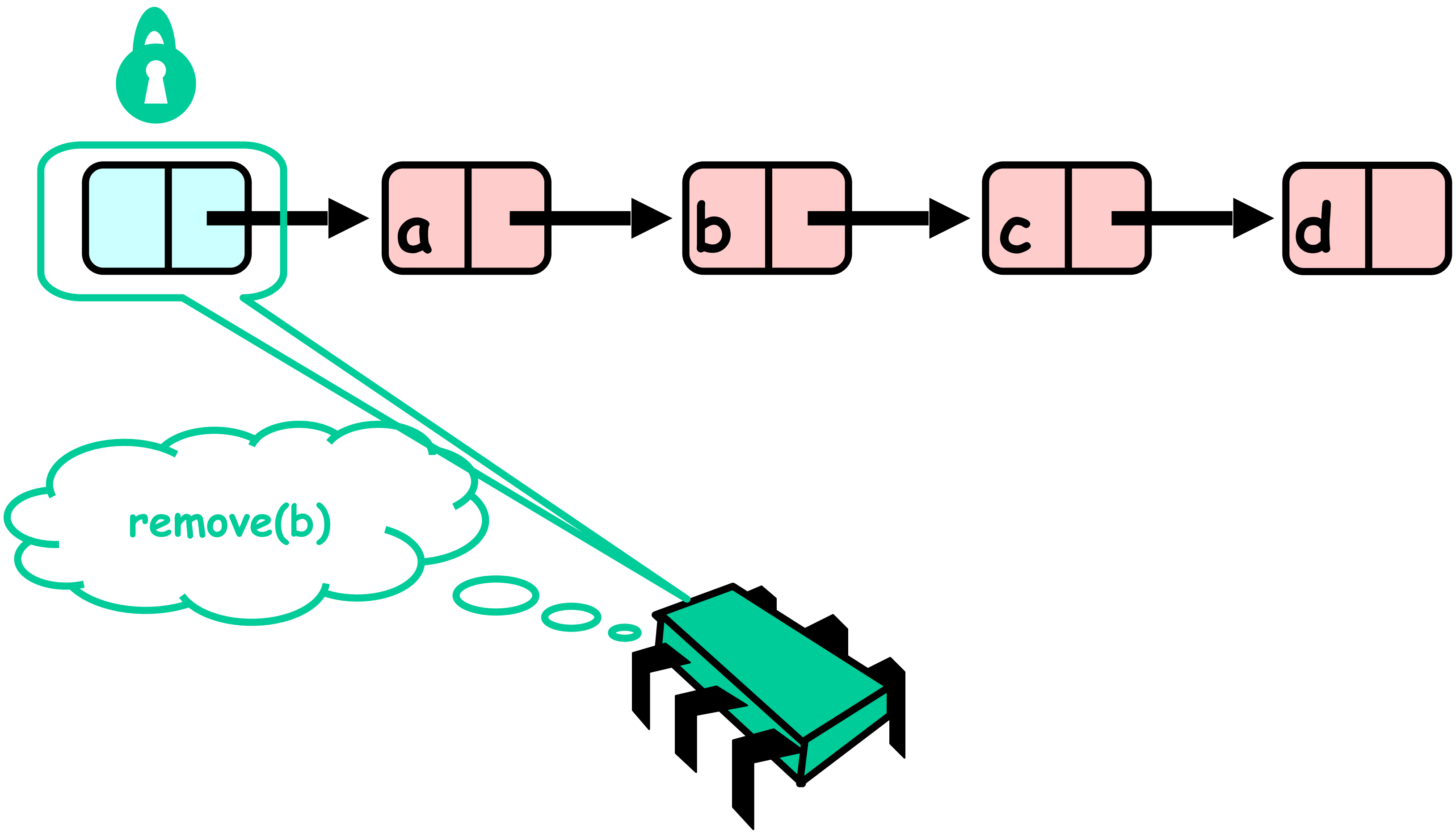
# Hand-over-Hand locking



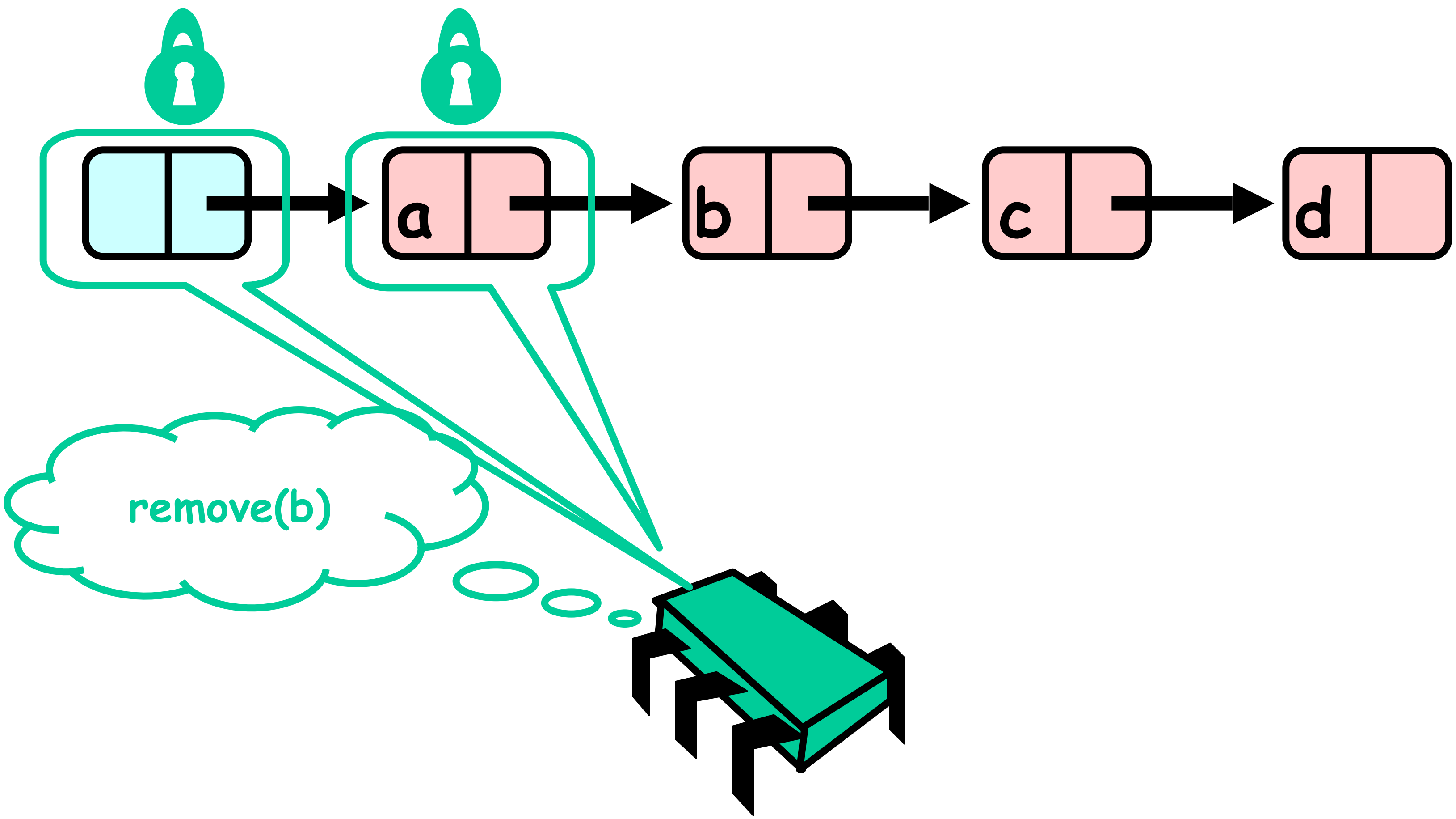
# Removing a Node



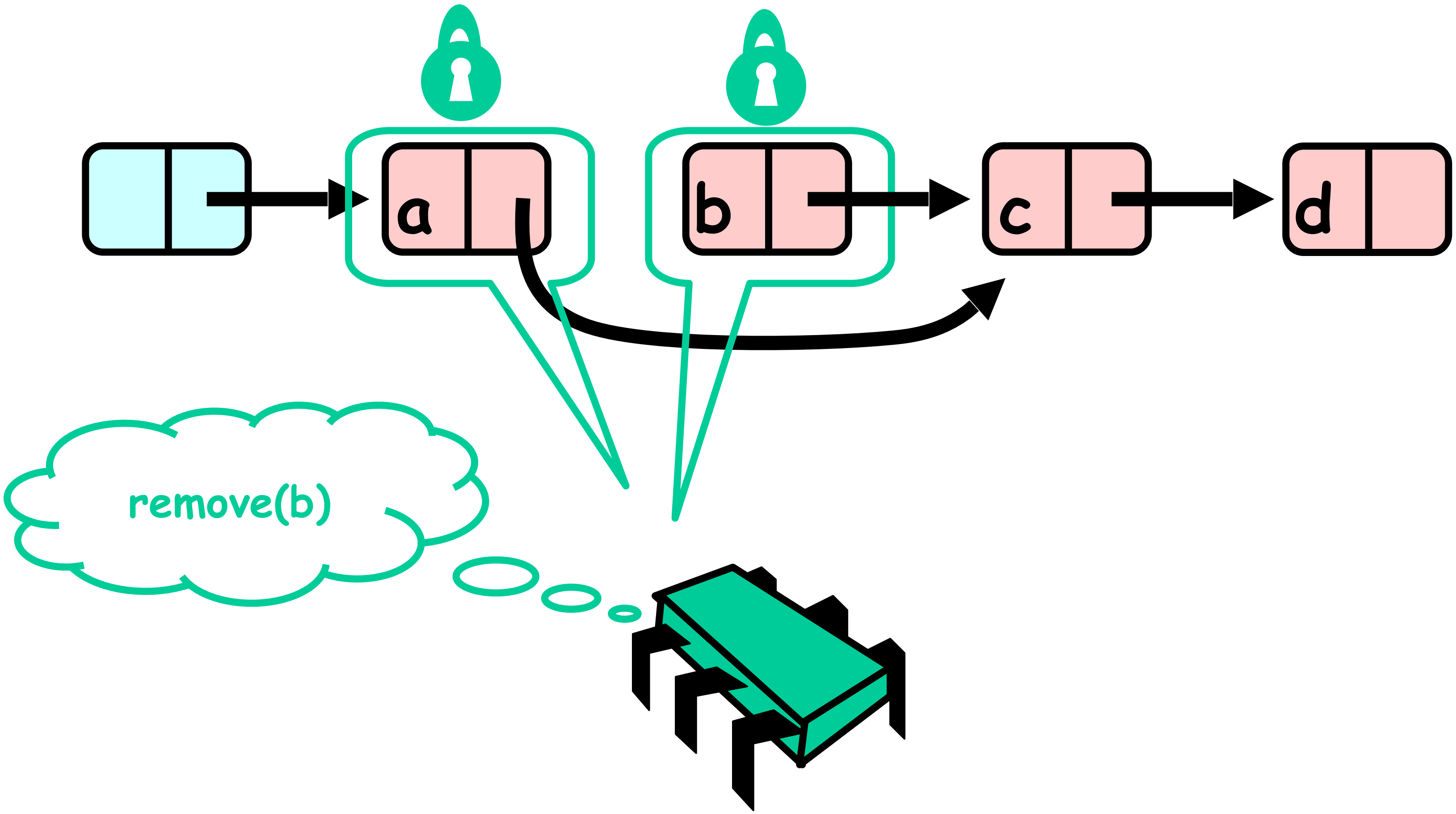
# Removing a Node



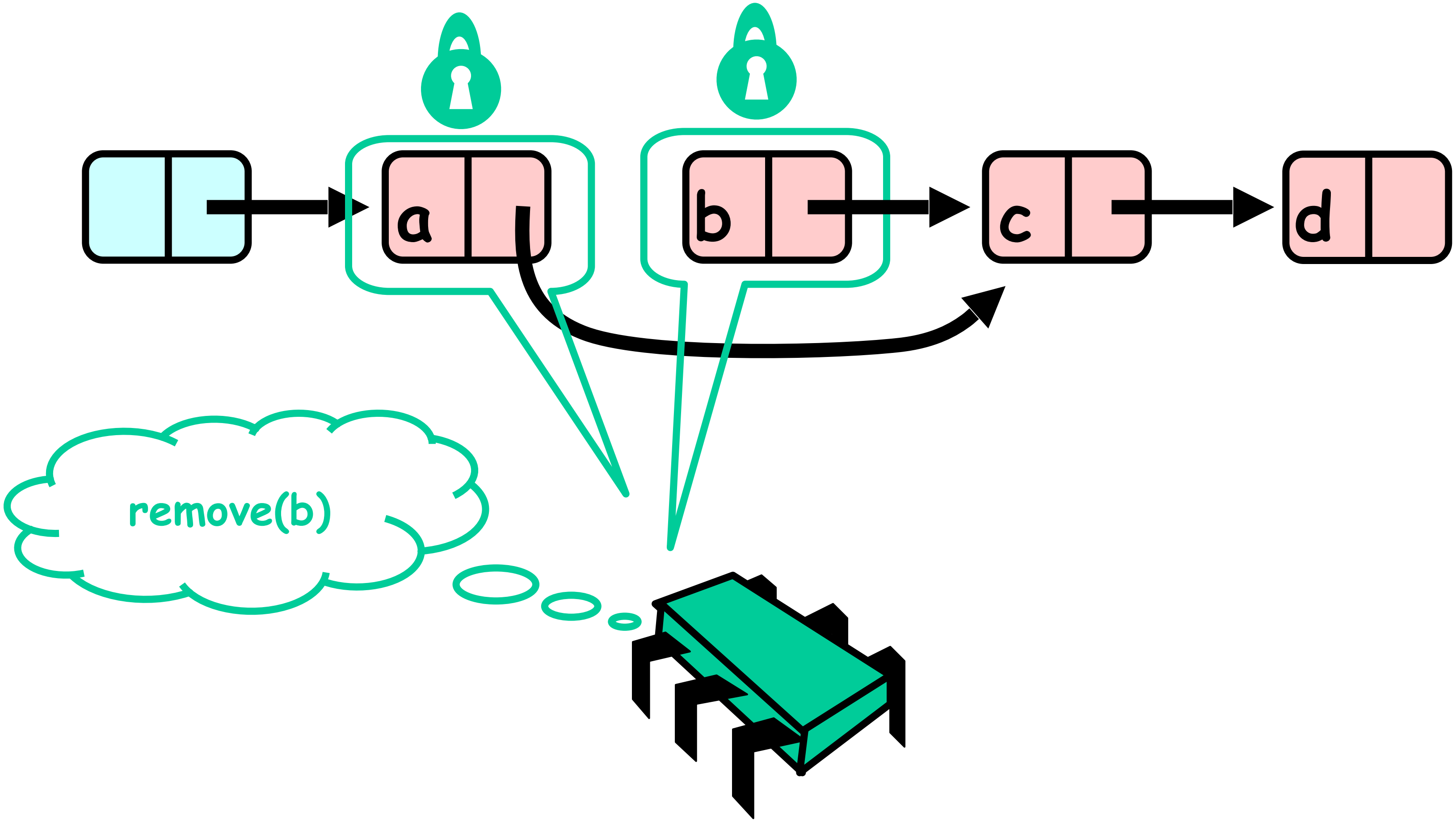
# Removing a Node



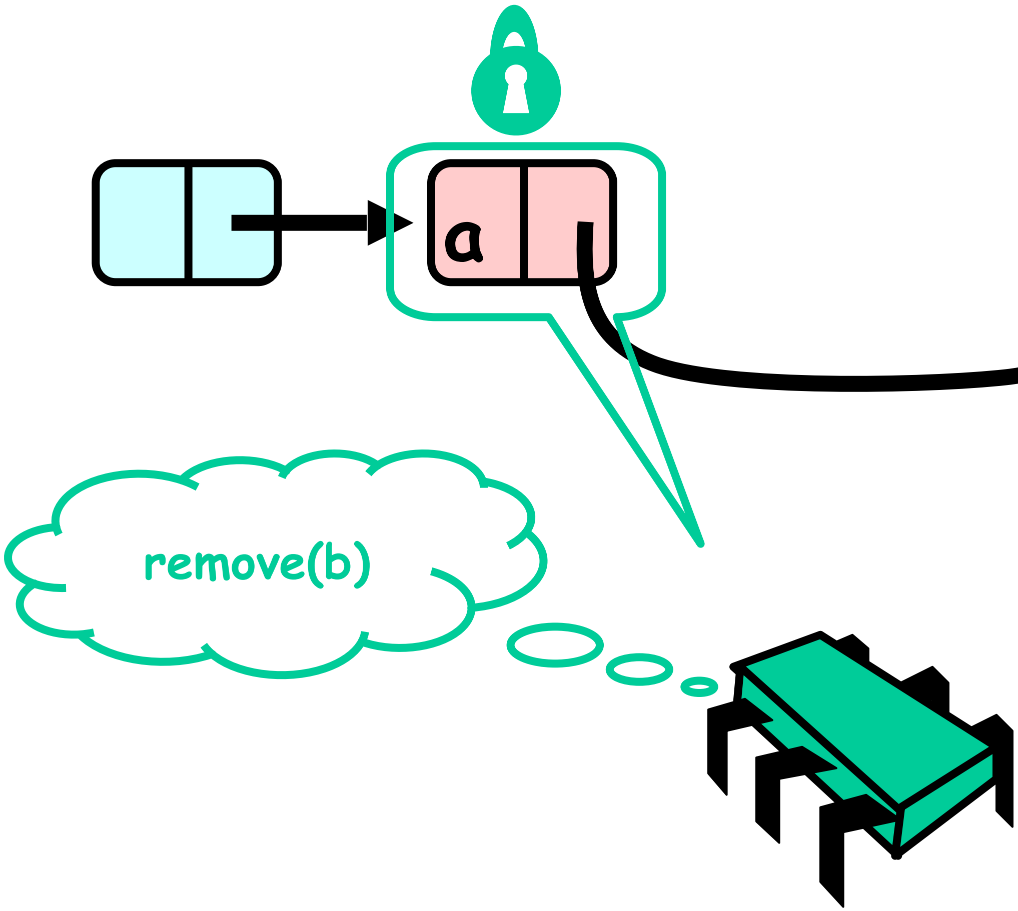
# Removing a Node



# Removing a Node

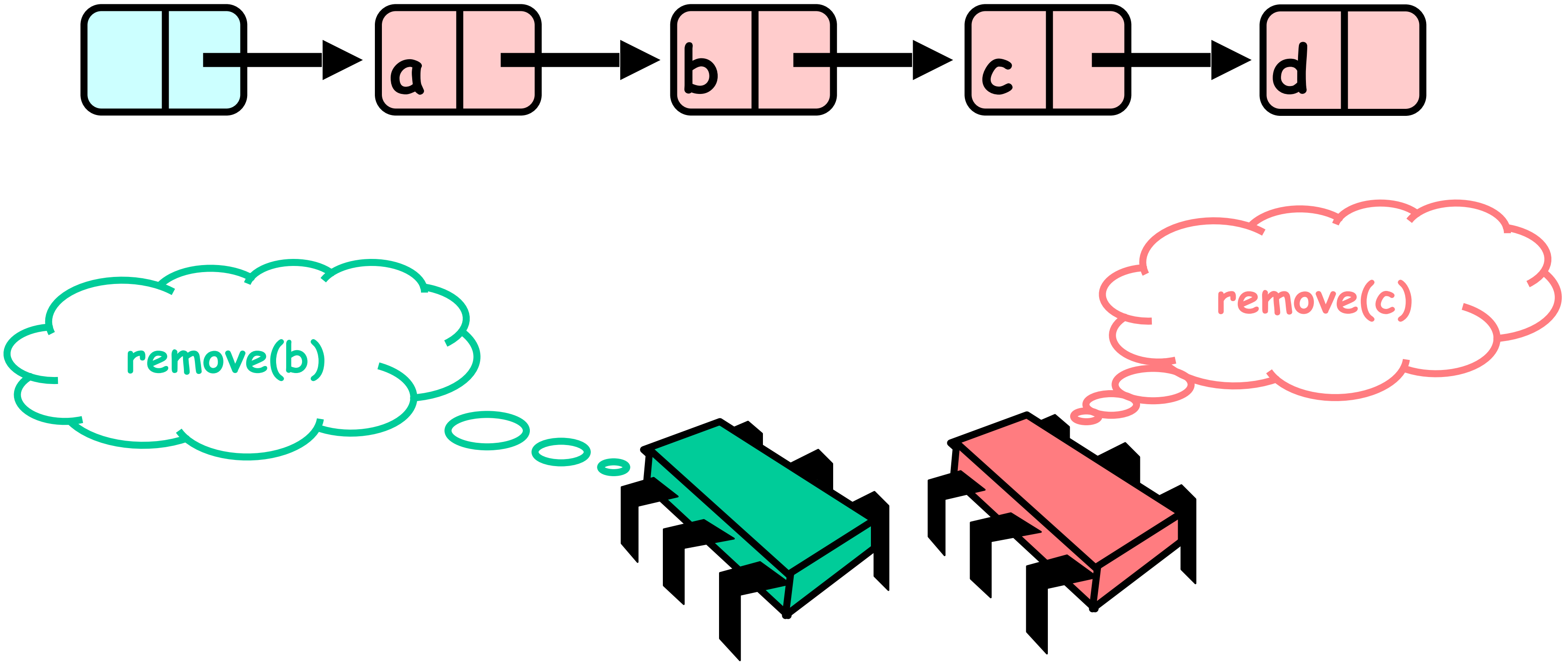


# Removing a Node

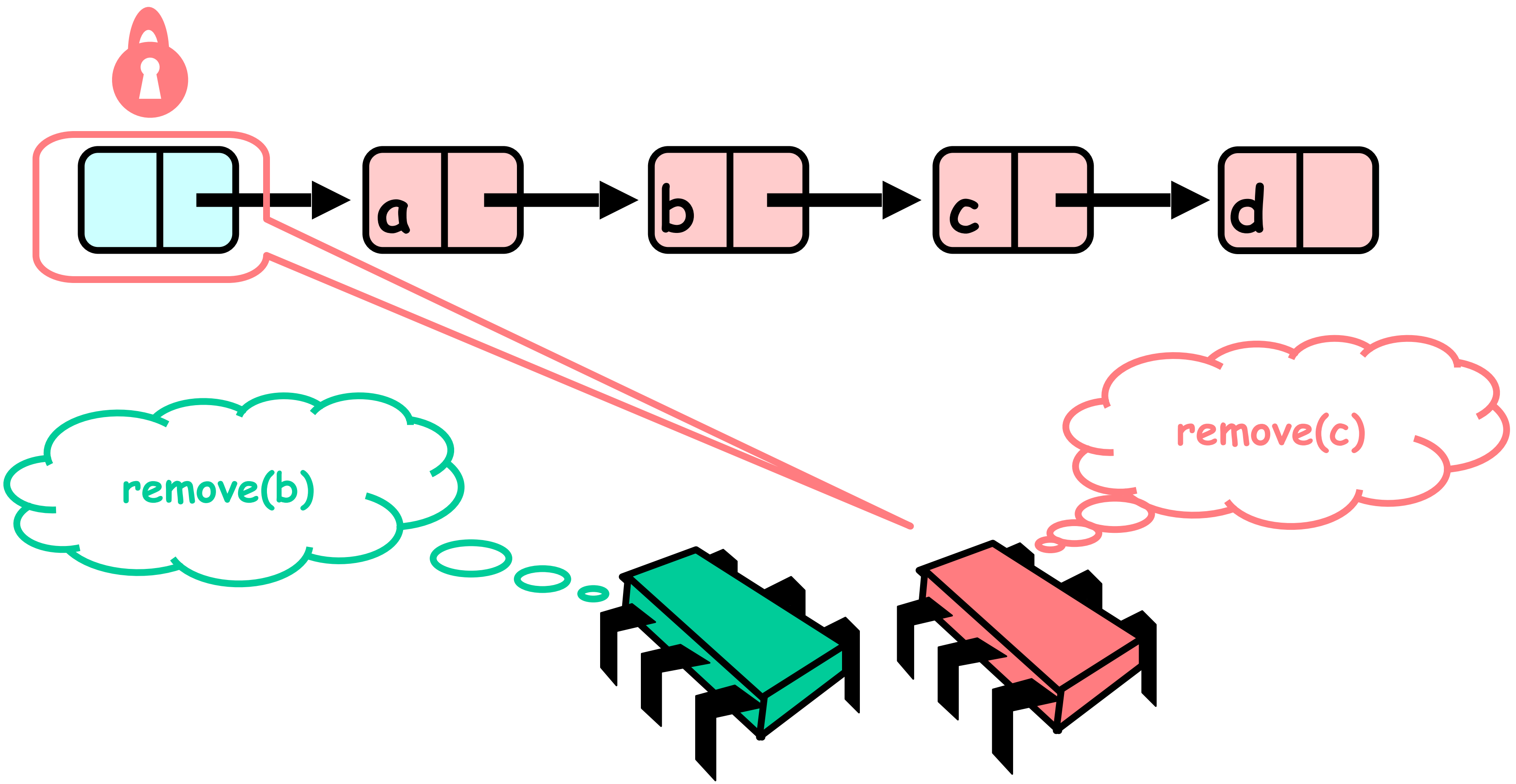



**Why do we need to always hold 2 locks?**

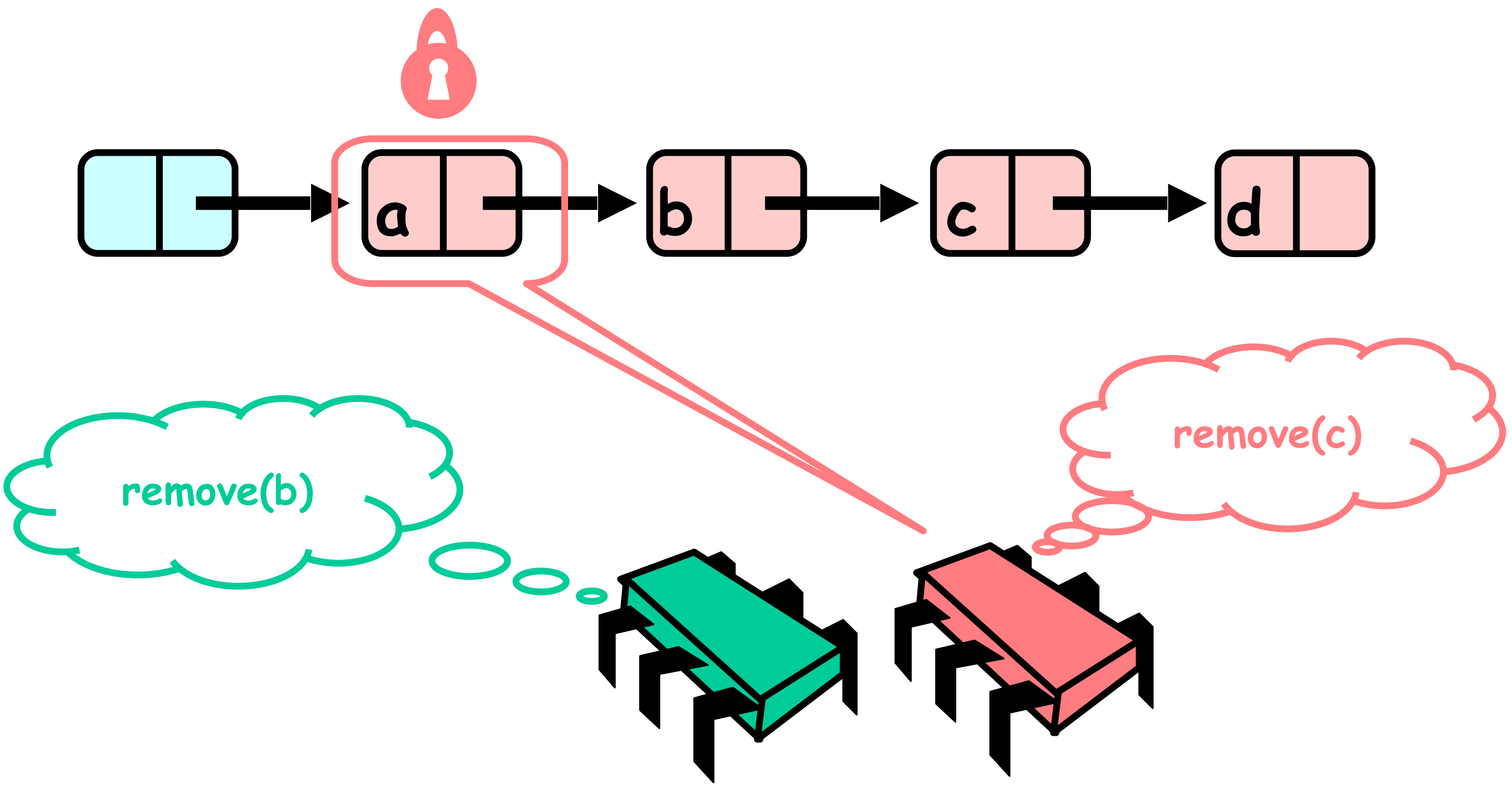
# Concurrent Removes



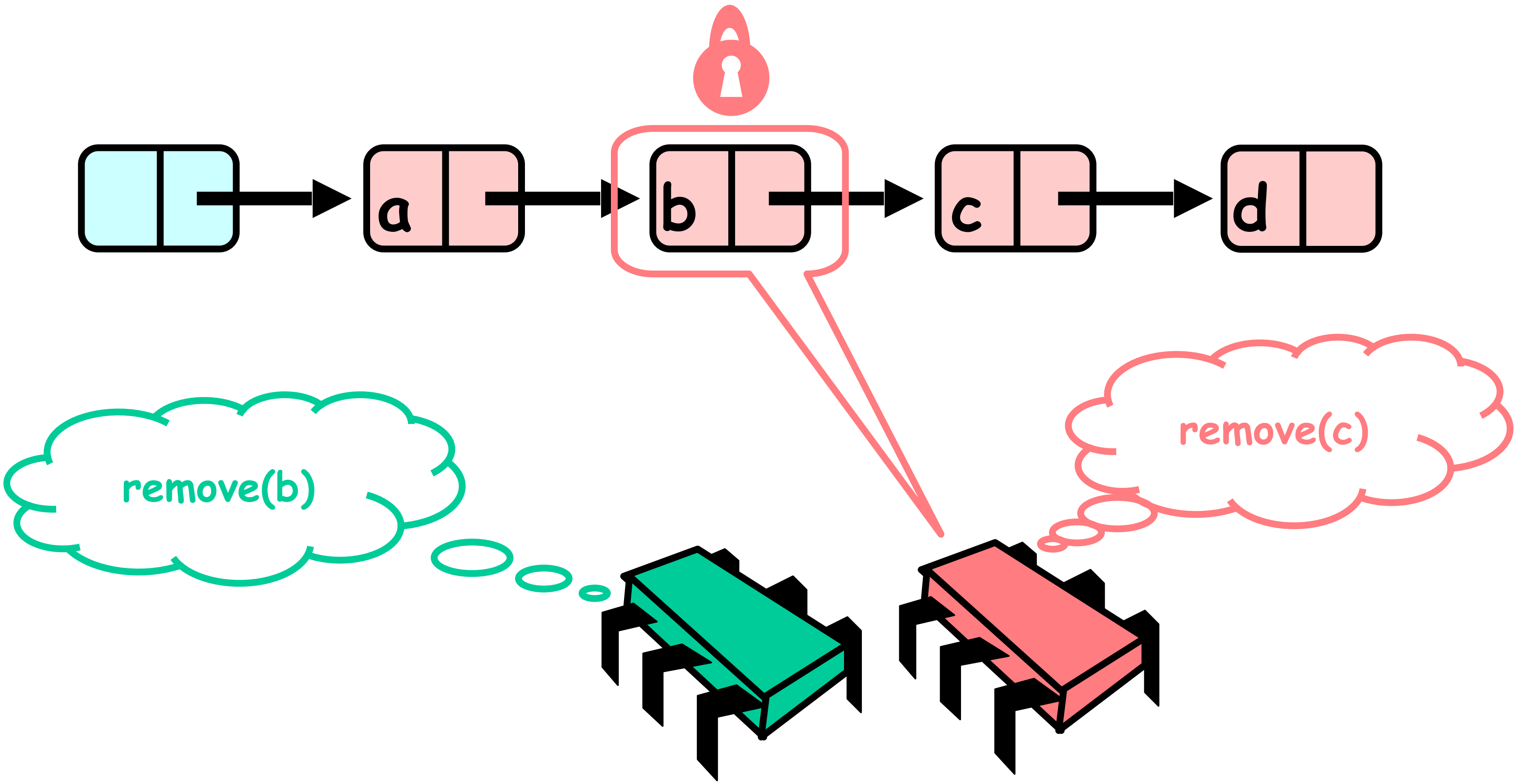
# Concurrent Removes



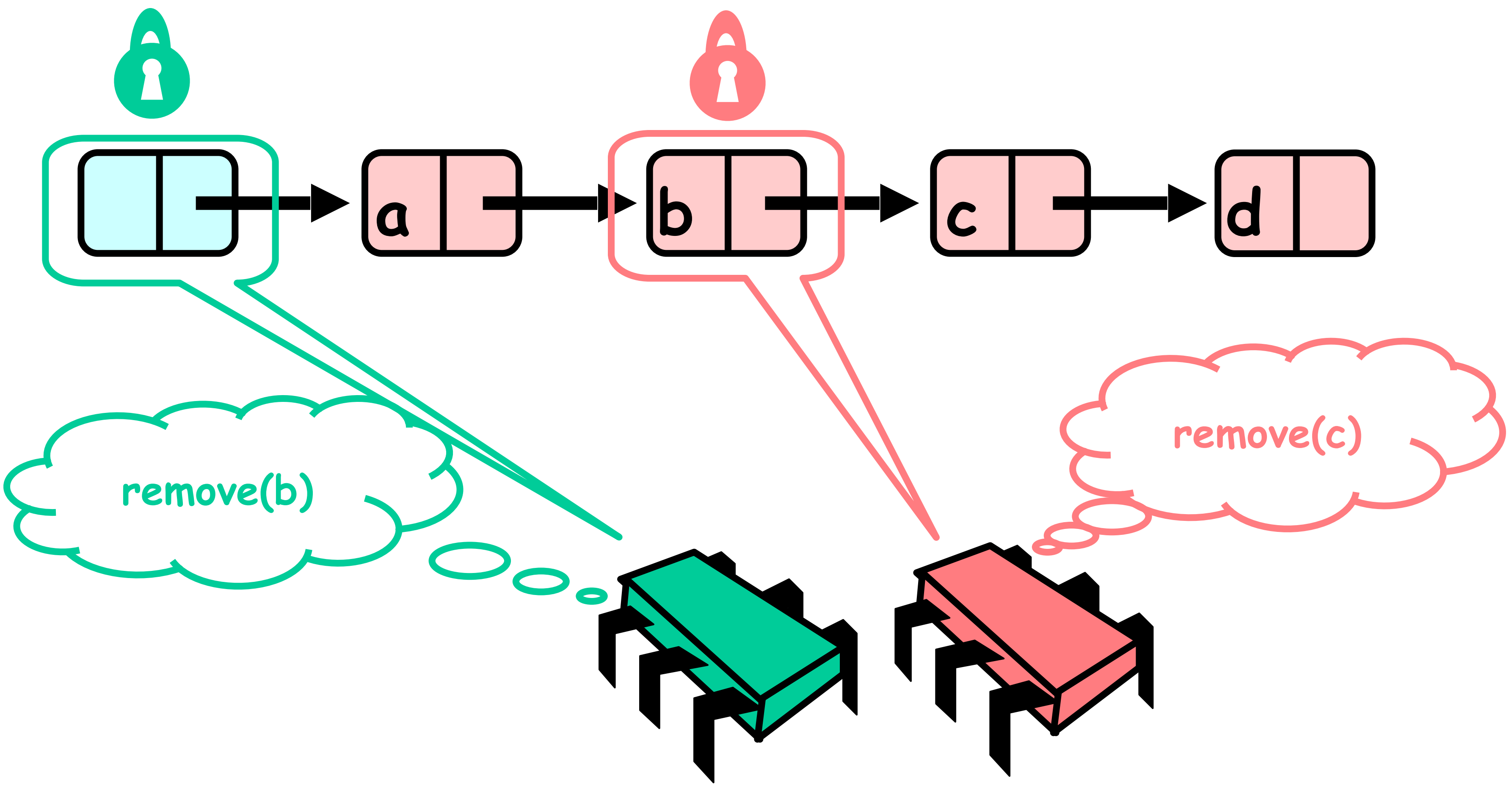
# Concurrent Removes



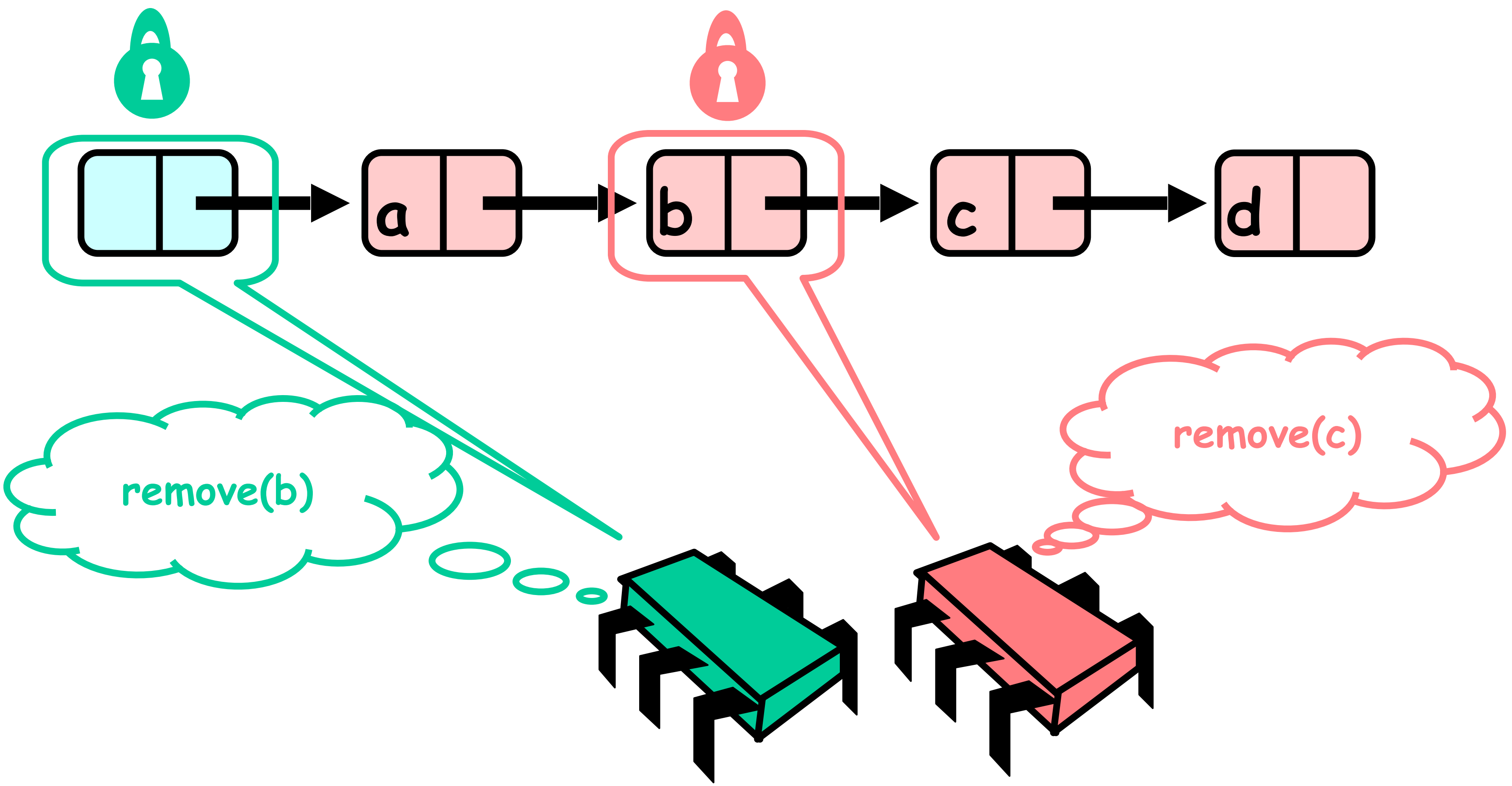
# Concurrent Removes



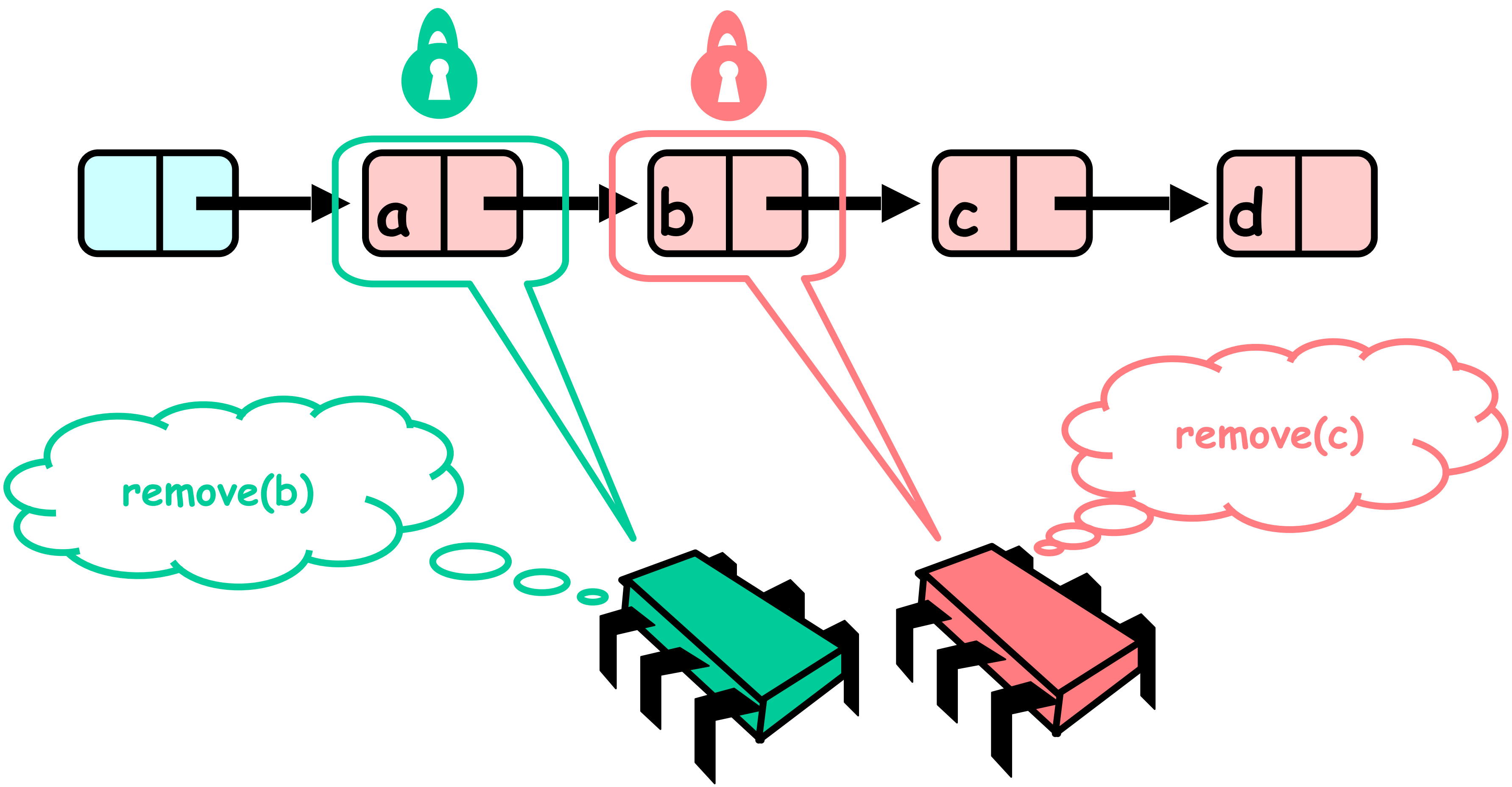
# Concurrent Removes



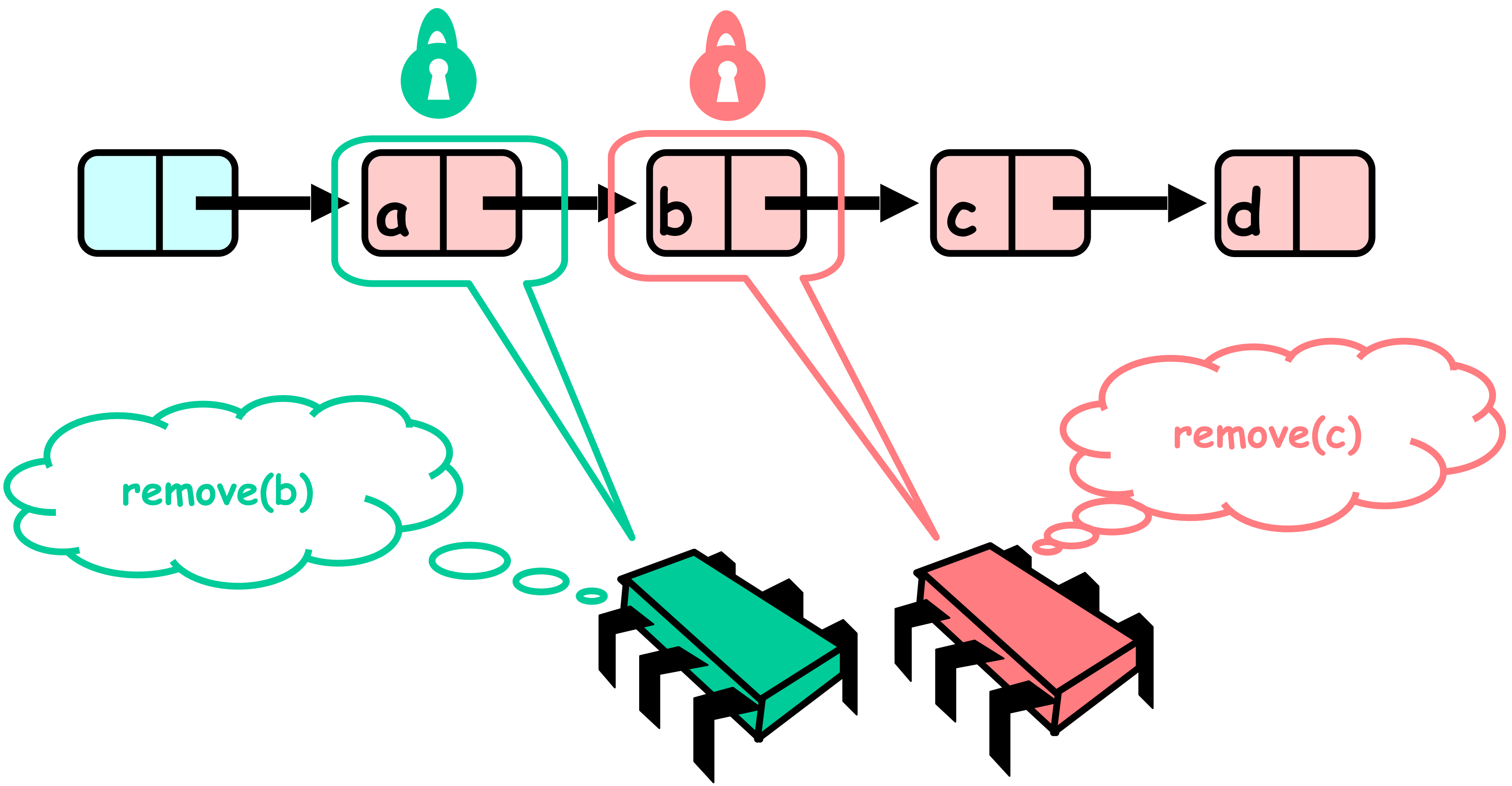
# Concurrent Removes



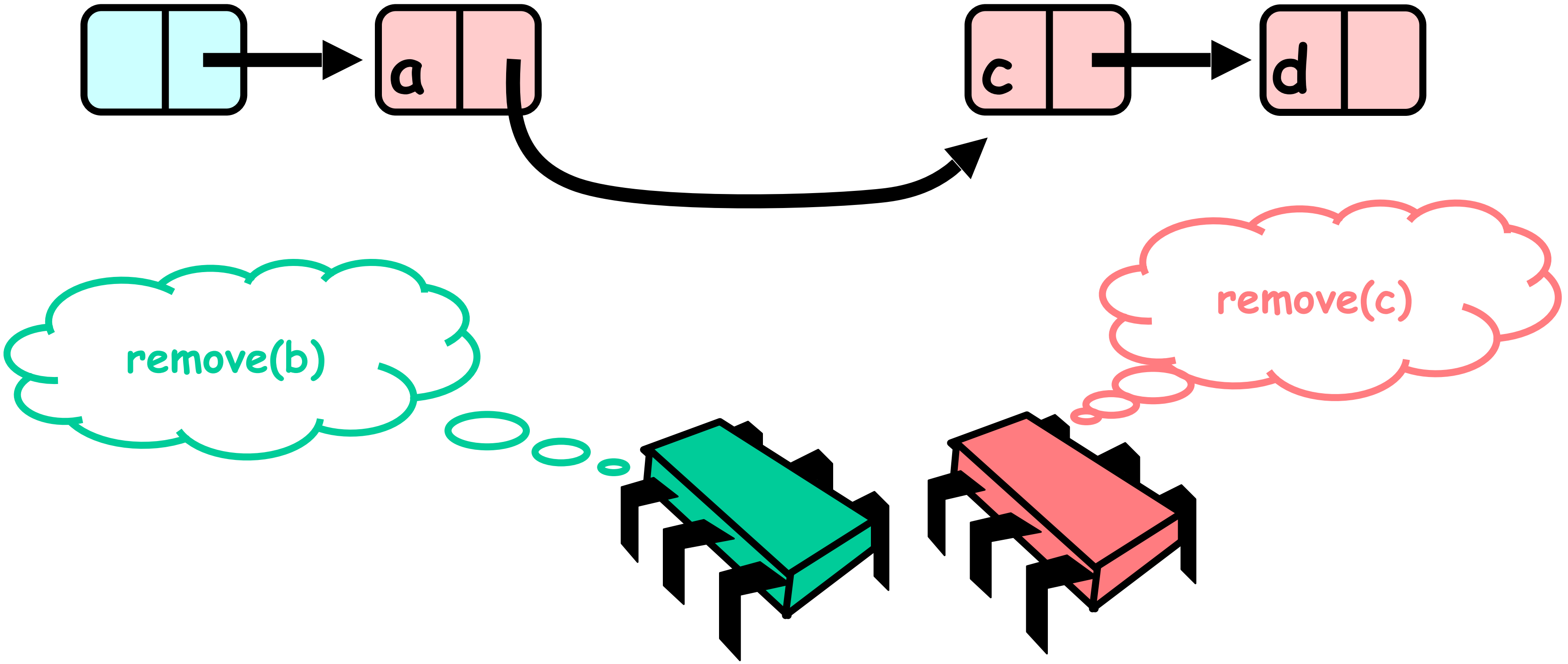
# Concurrent Removes



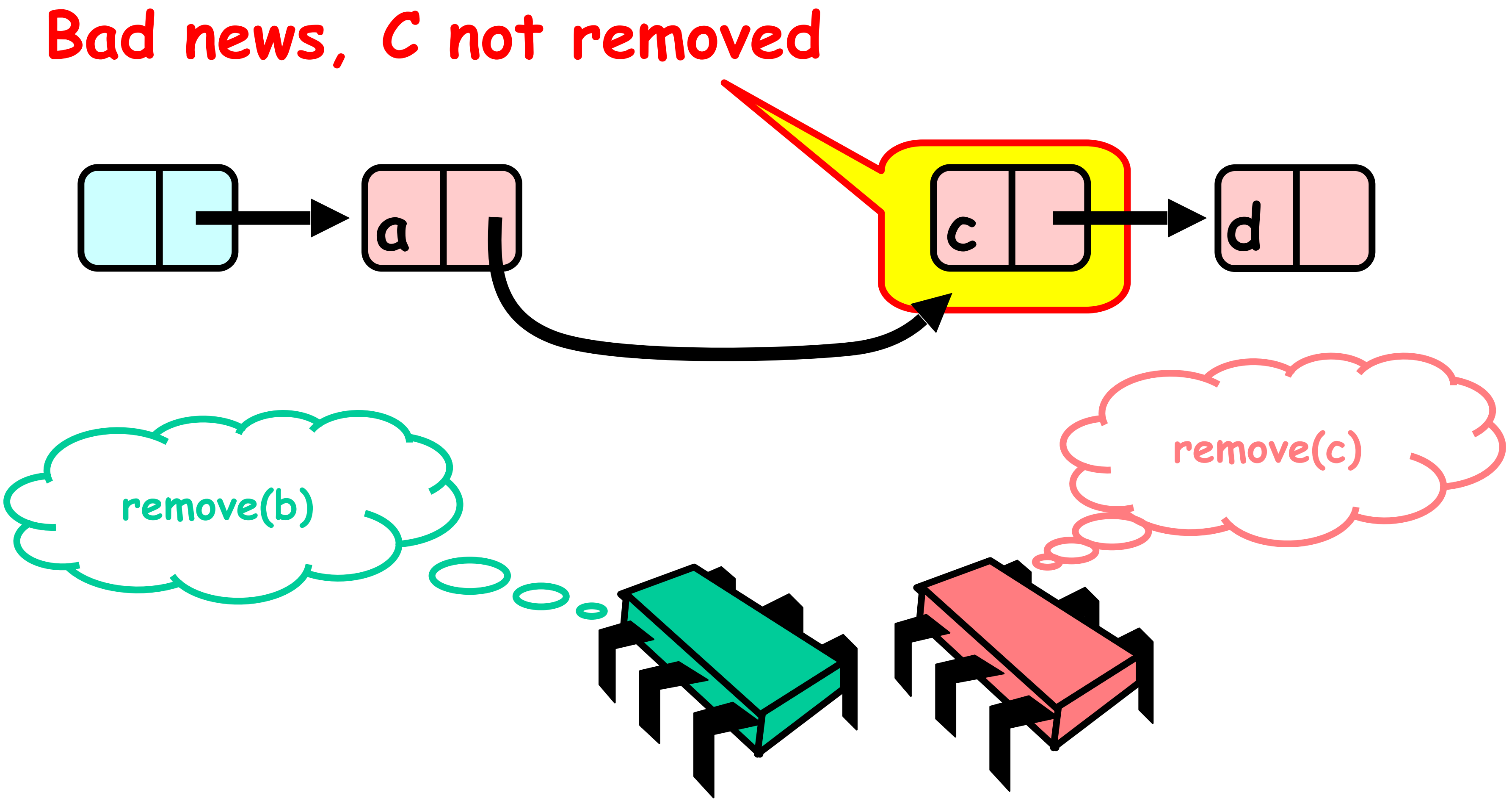
# Concurrent Removes



# Uh, Oh

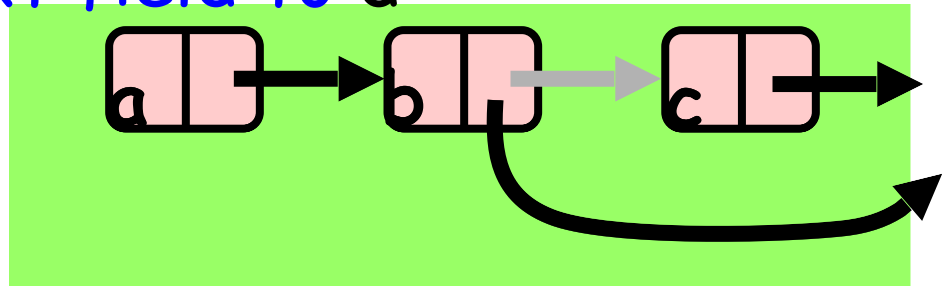


# Uh, Oh

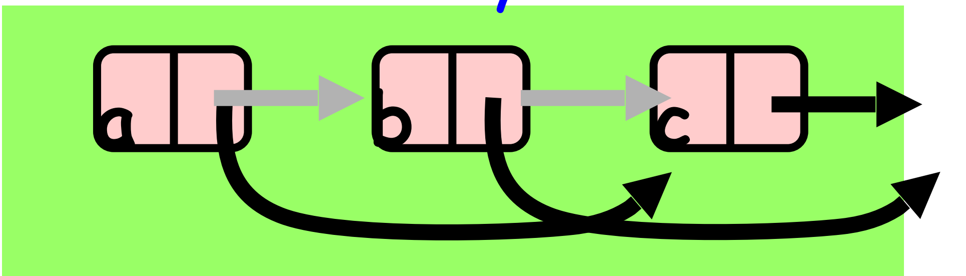


# Problem

- To delete node c
  - Swing node b's next field to d



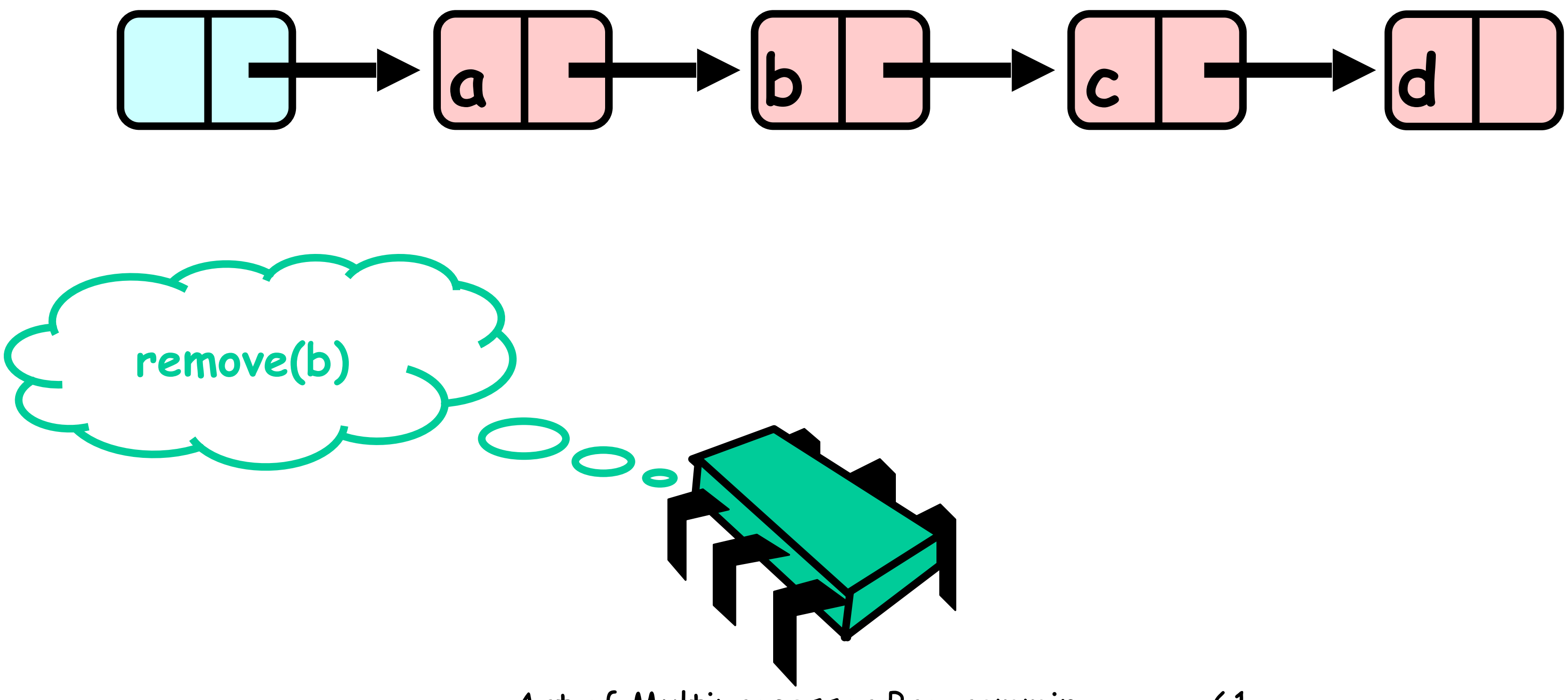

- Problem is,
  - Someone deleting b concurrently could direct a pointer to c



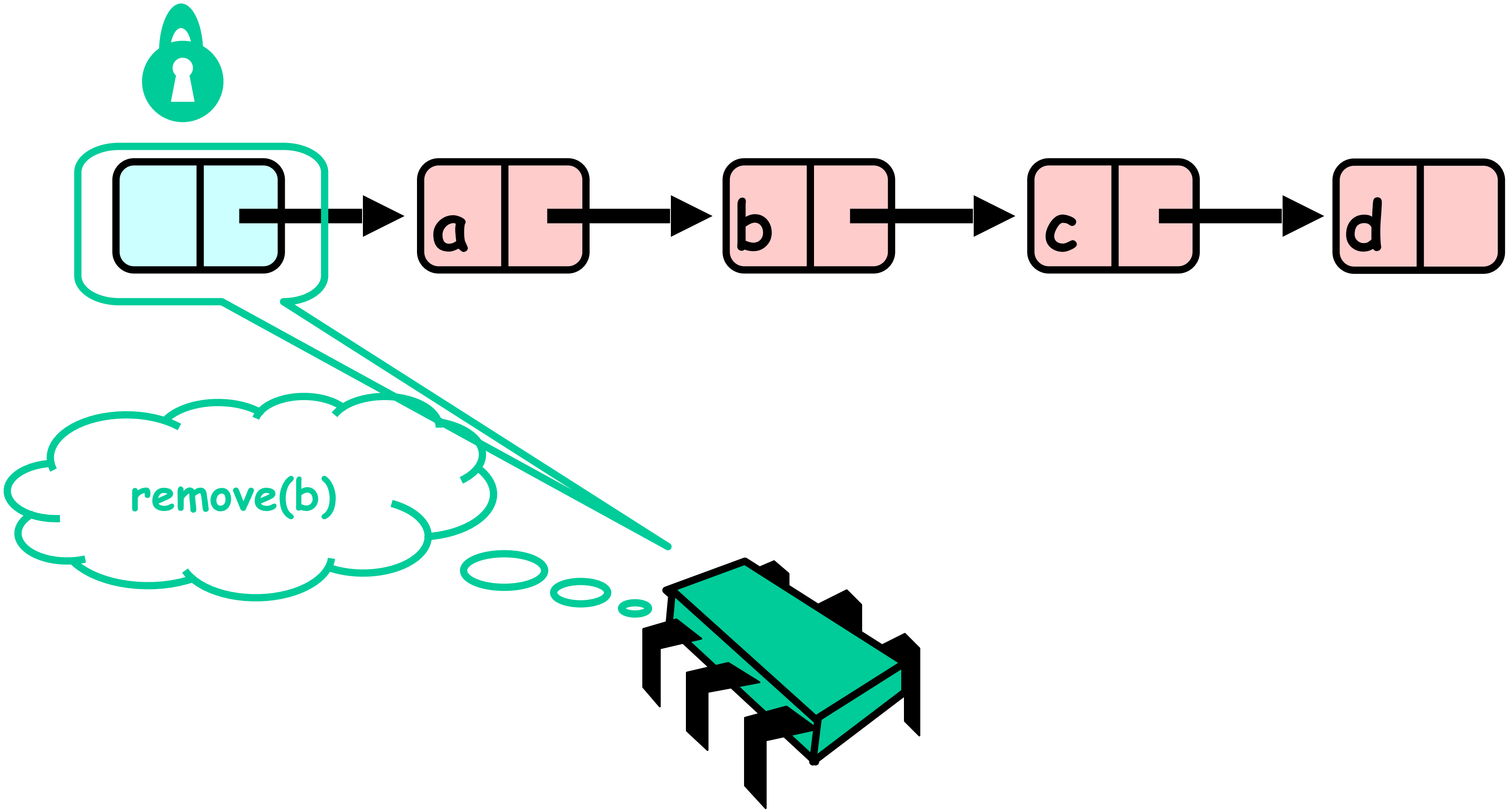
# Insight

- 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

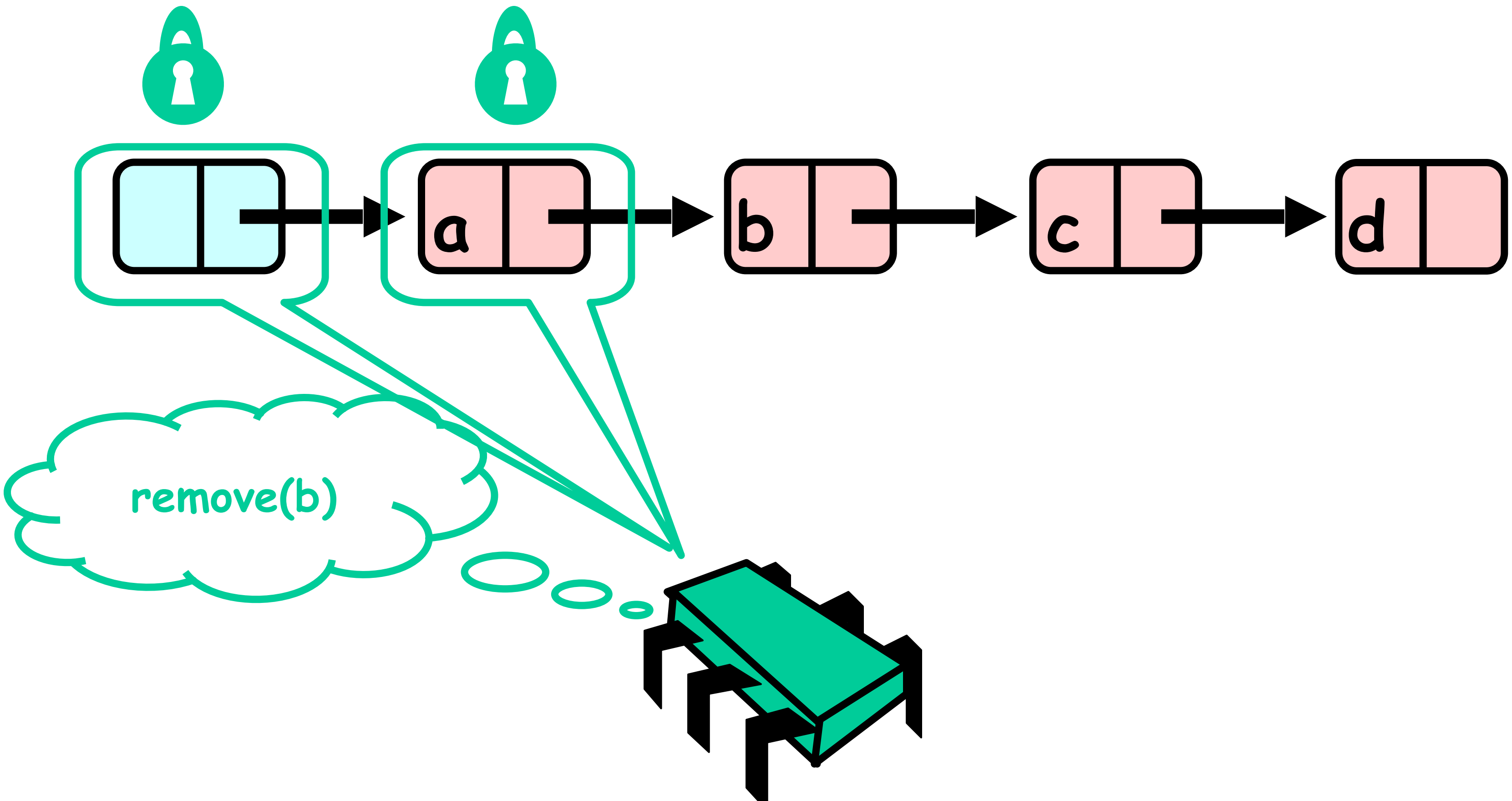
# Hand-Over-Hand Again



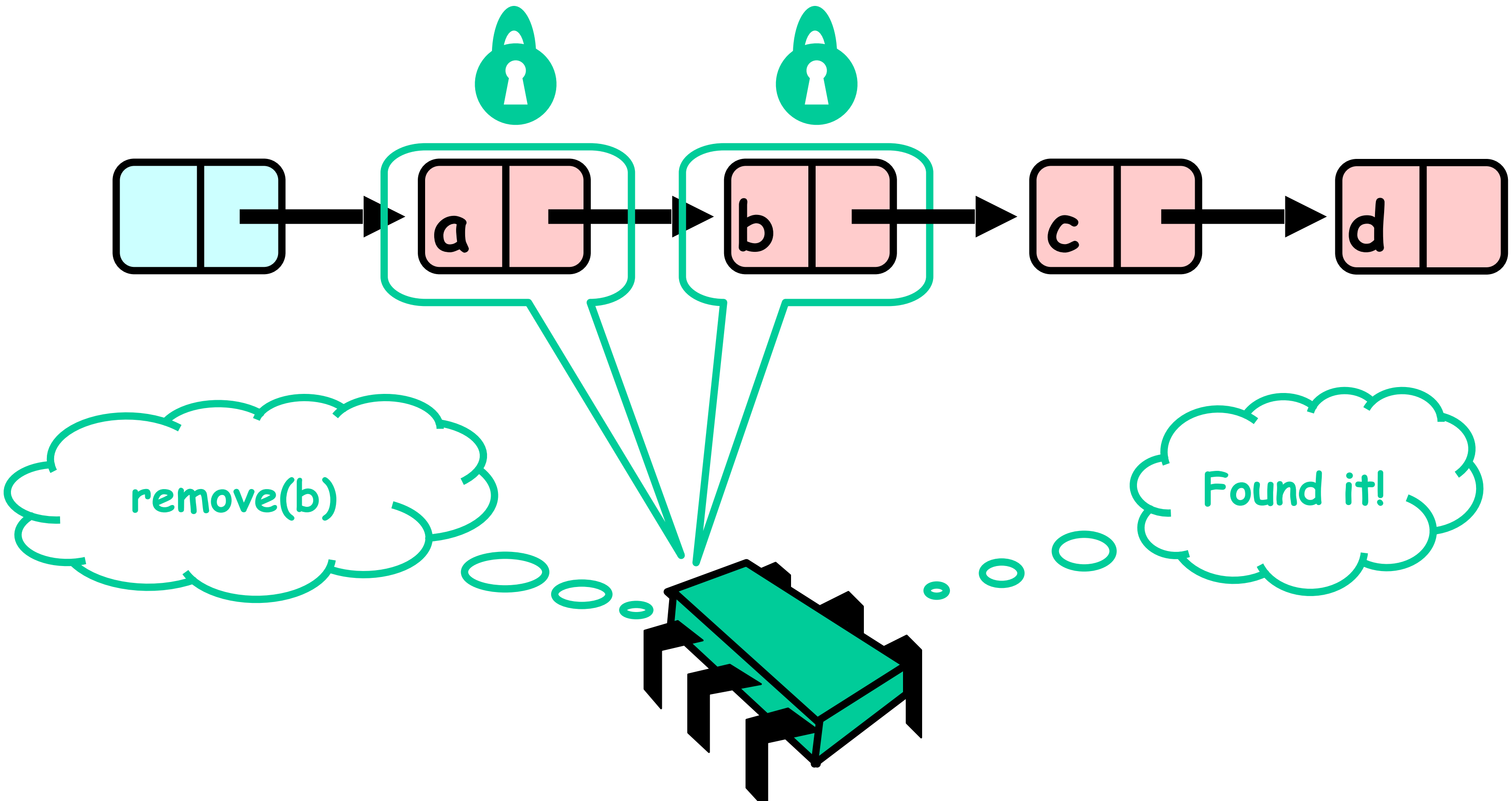
# Hand-Over-Hand Again



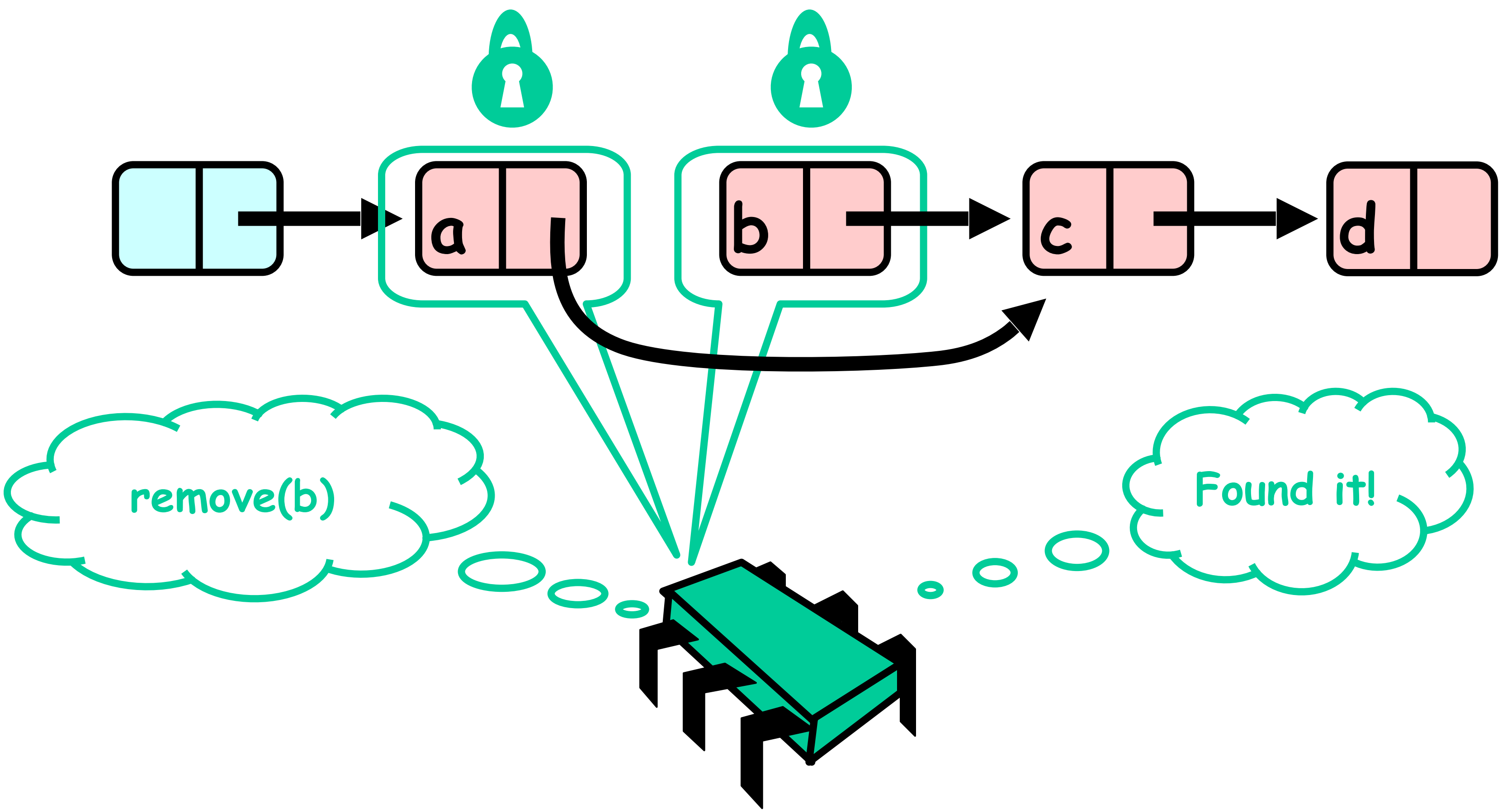
# Hand-Over-Hand Again



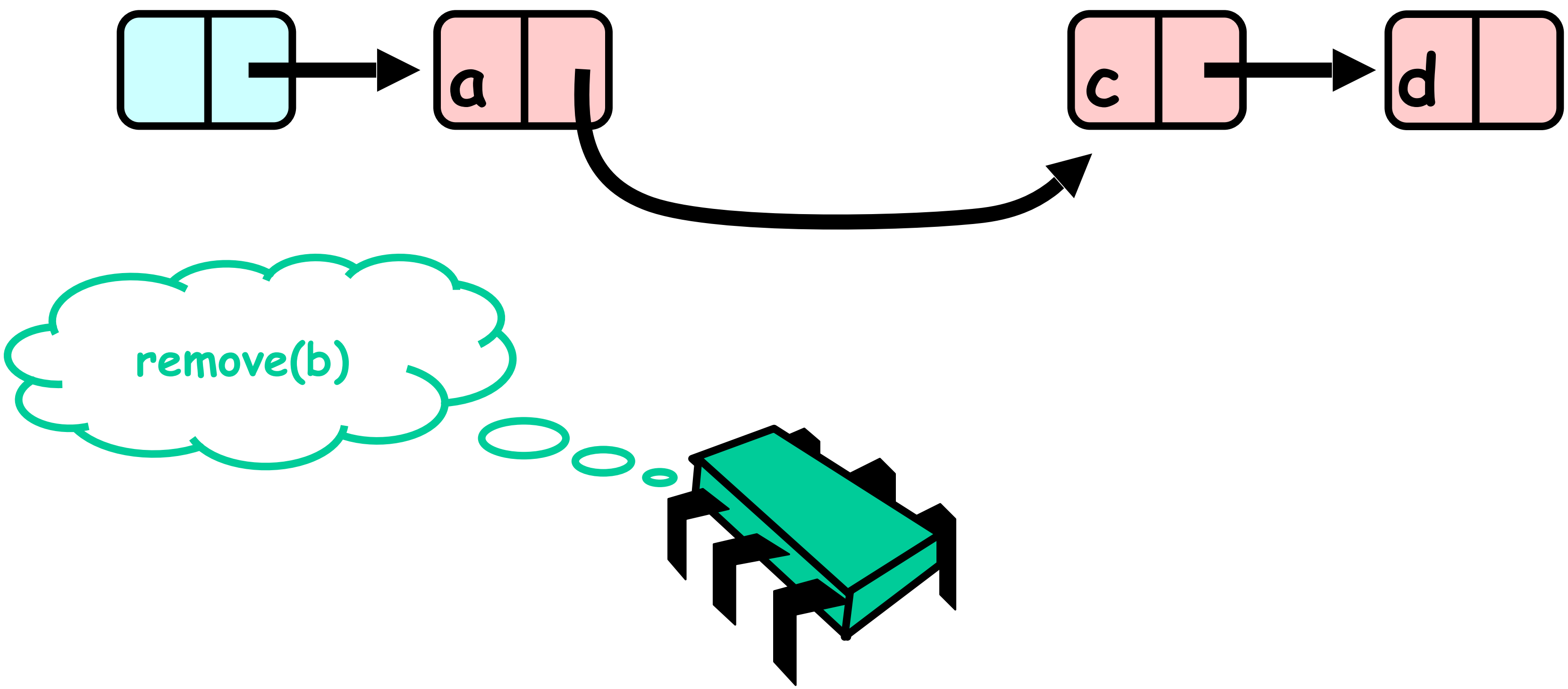
# Hand-Over-Hand Again



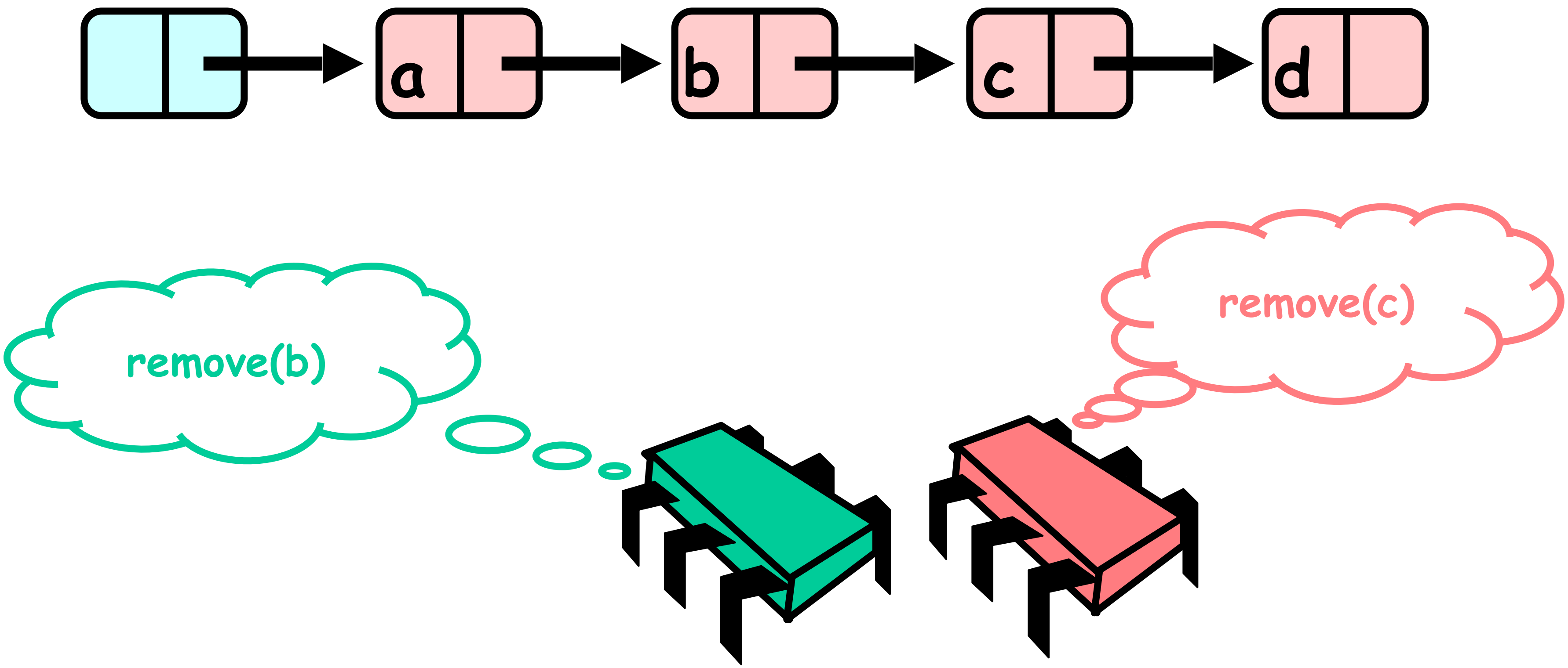
# Hand-Over-Hand Again



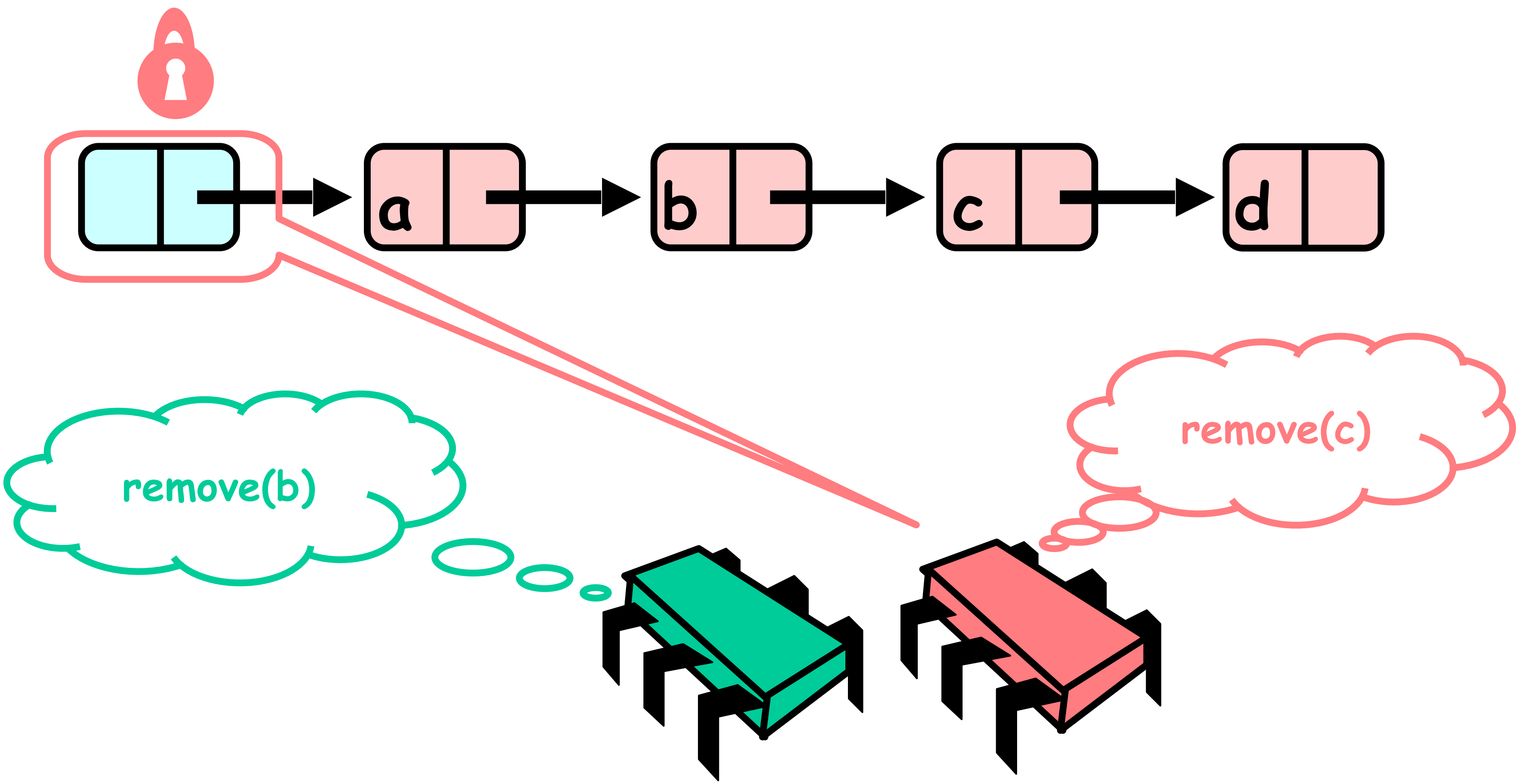
# Hand-Over-Hand Again



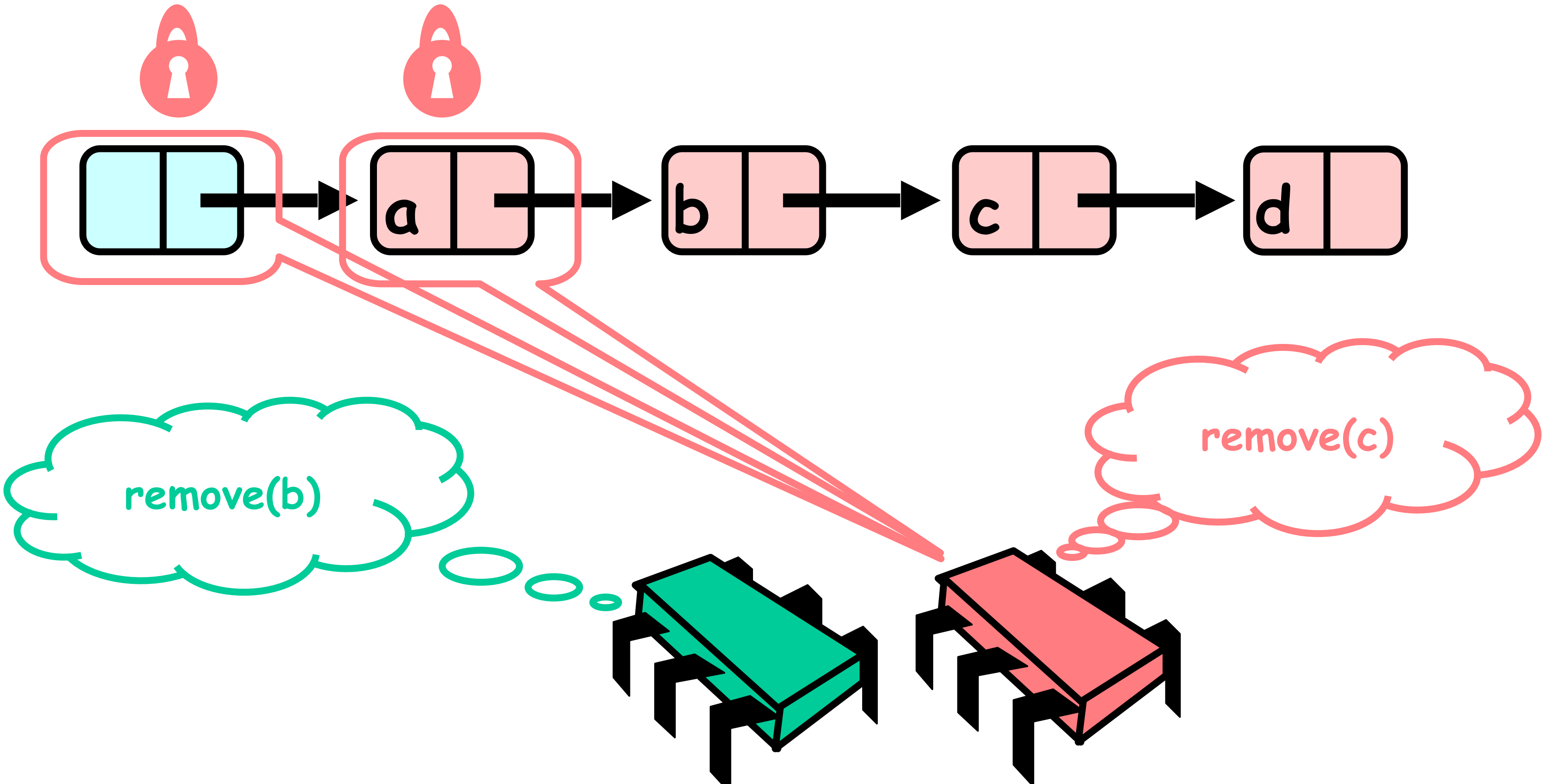
# Removing a Node



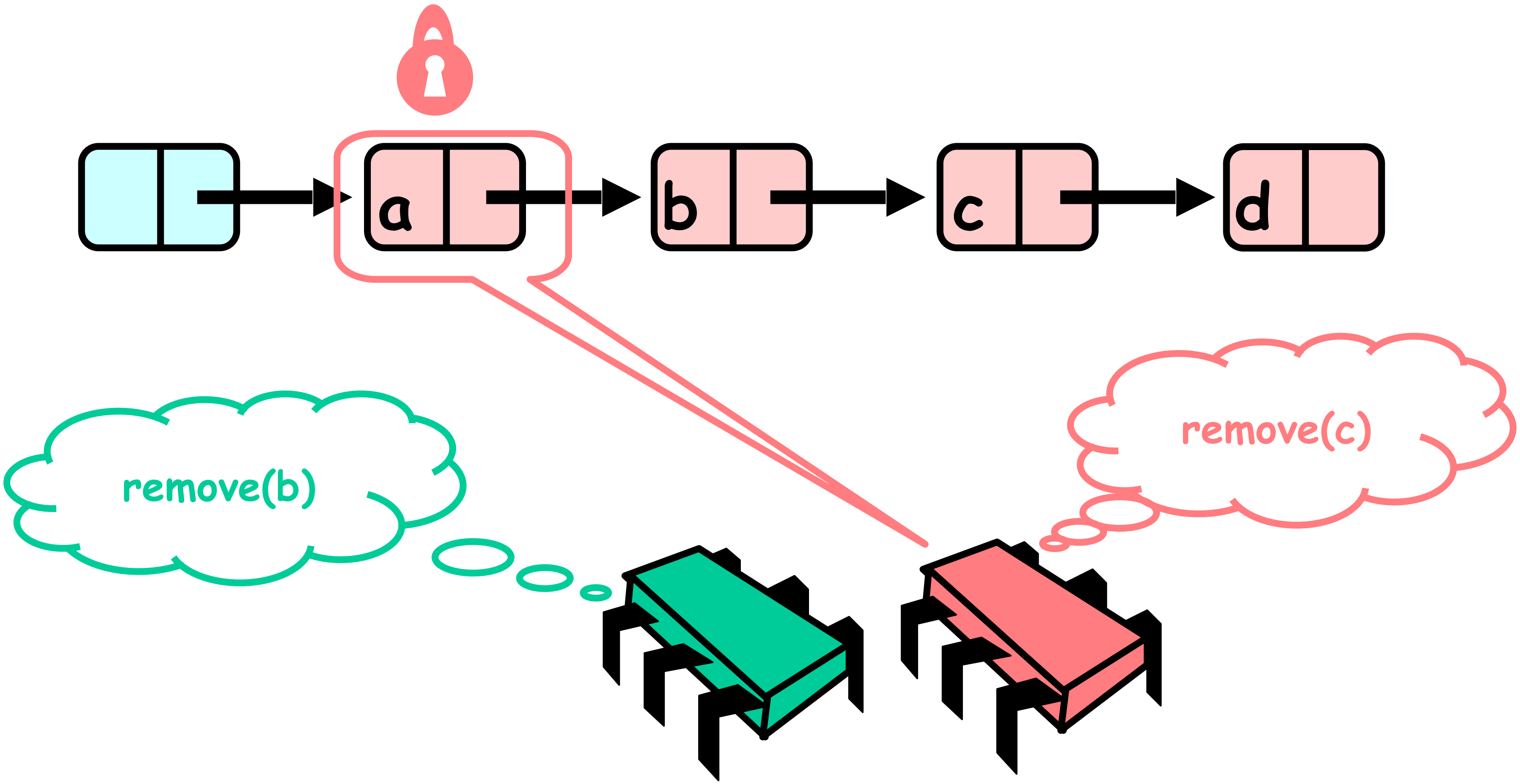
# Removing a Node



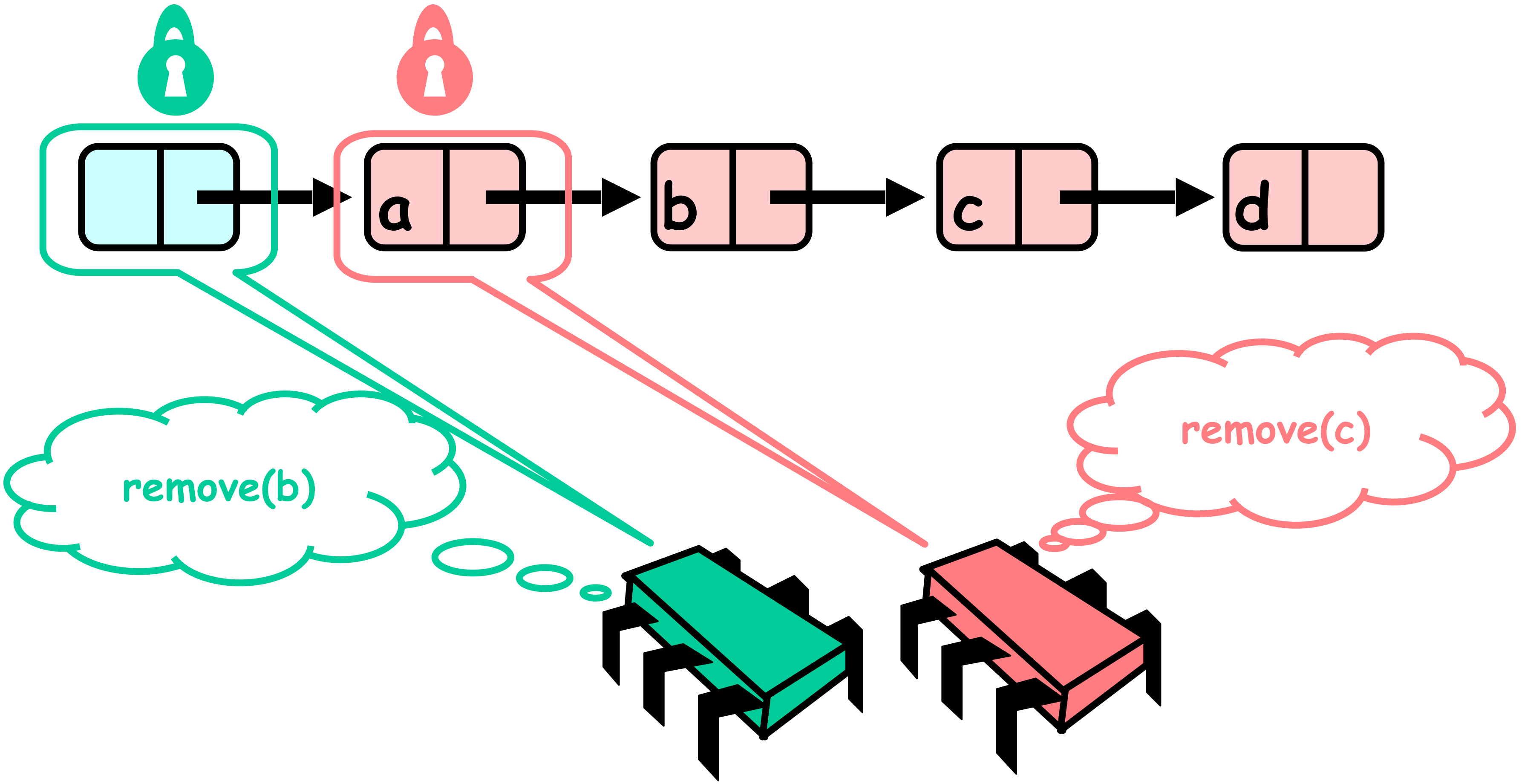
# Removing a Node



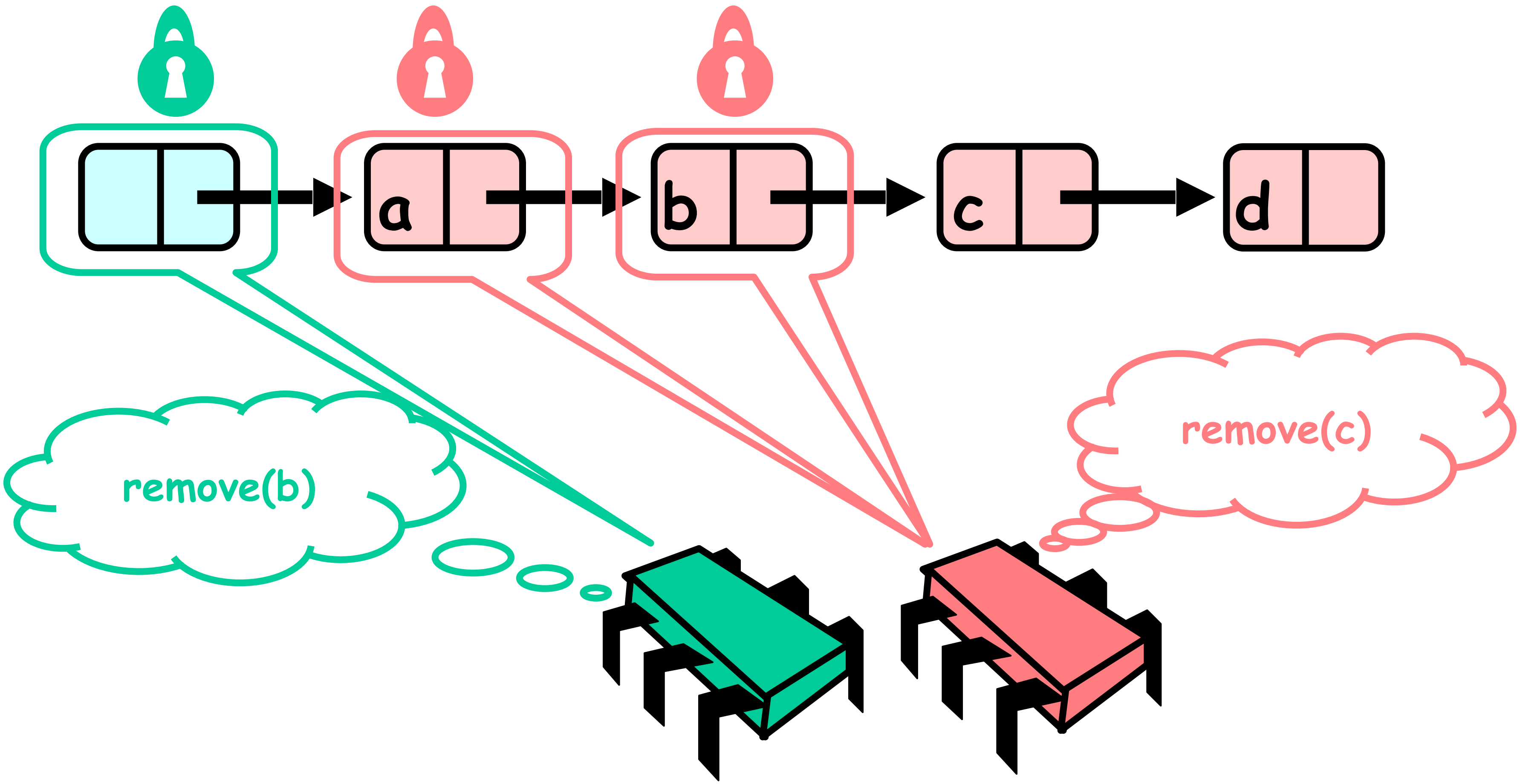
# Removing a Node



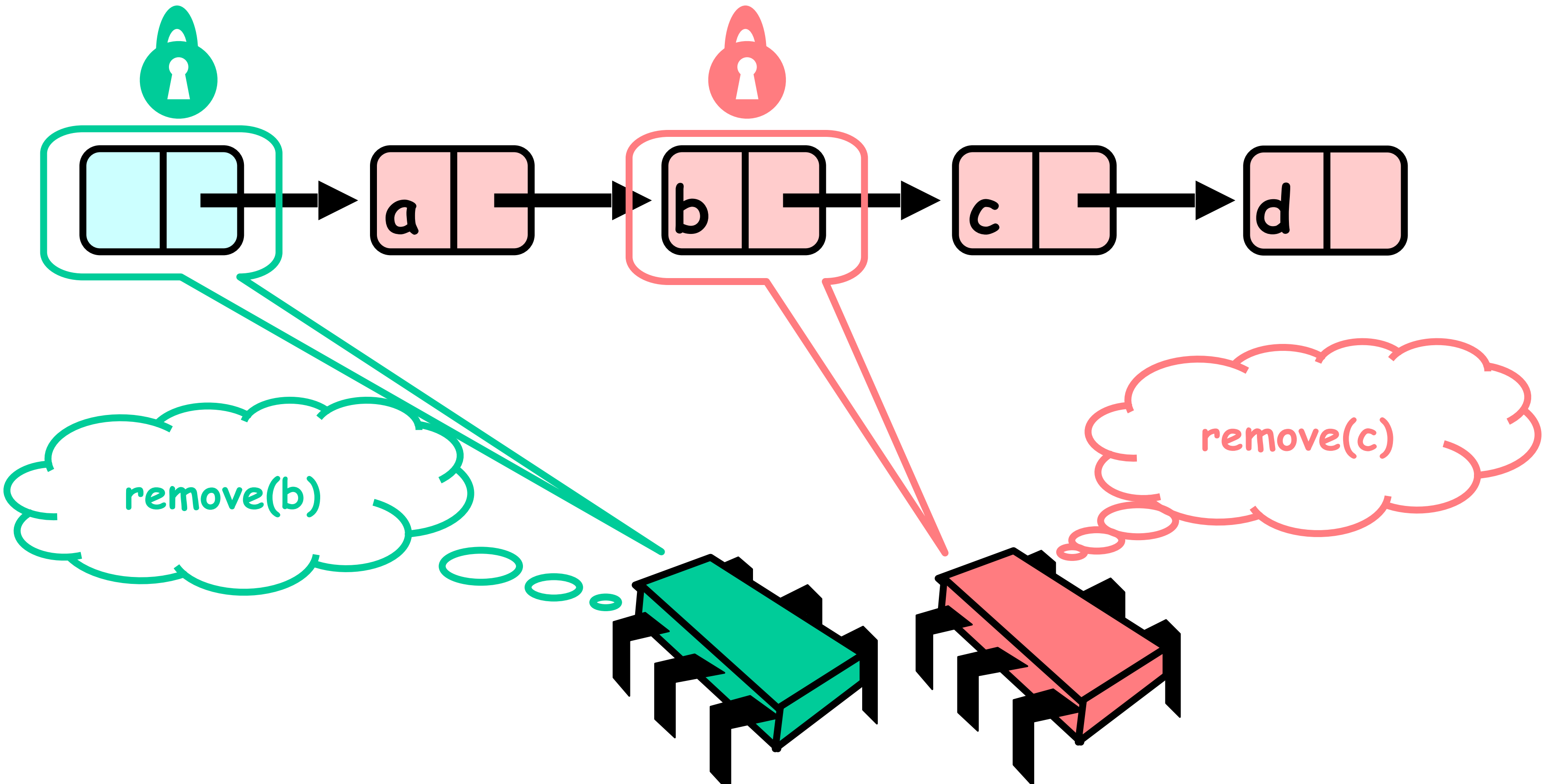
# Removing a Node



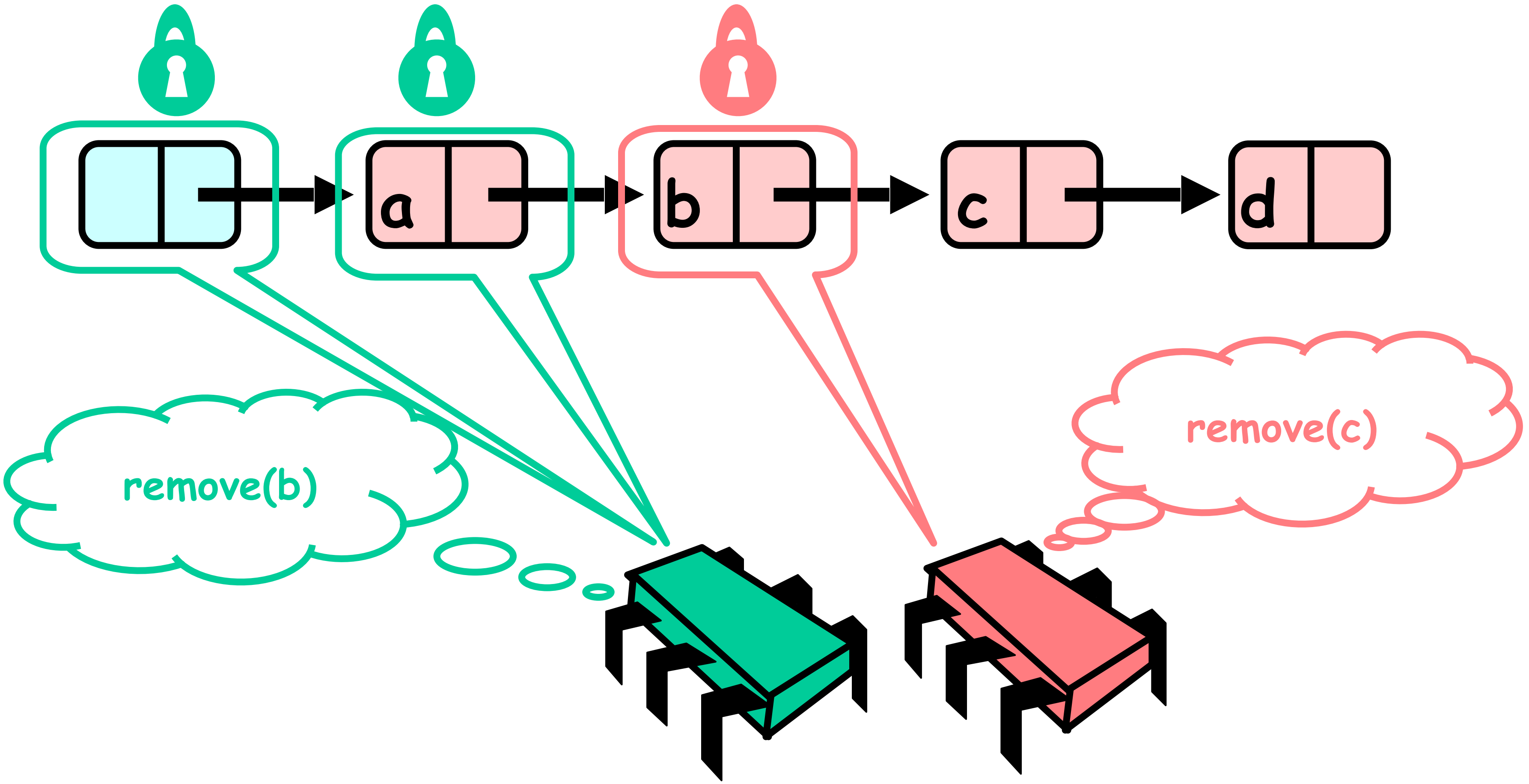
# Removing a Node



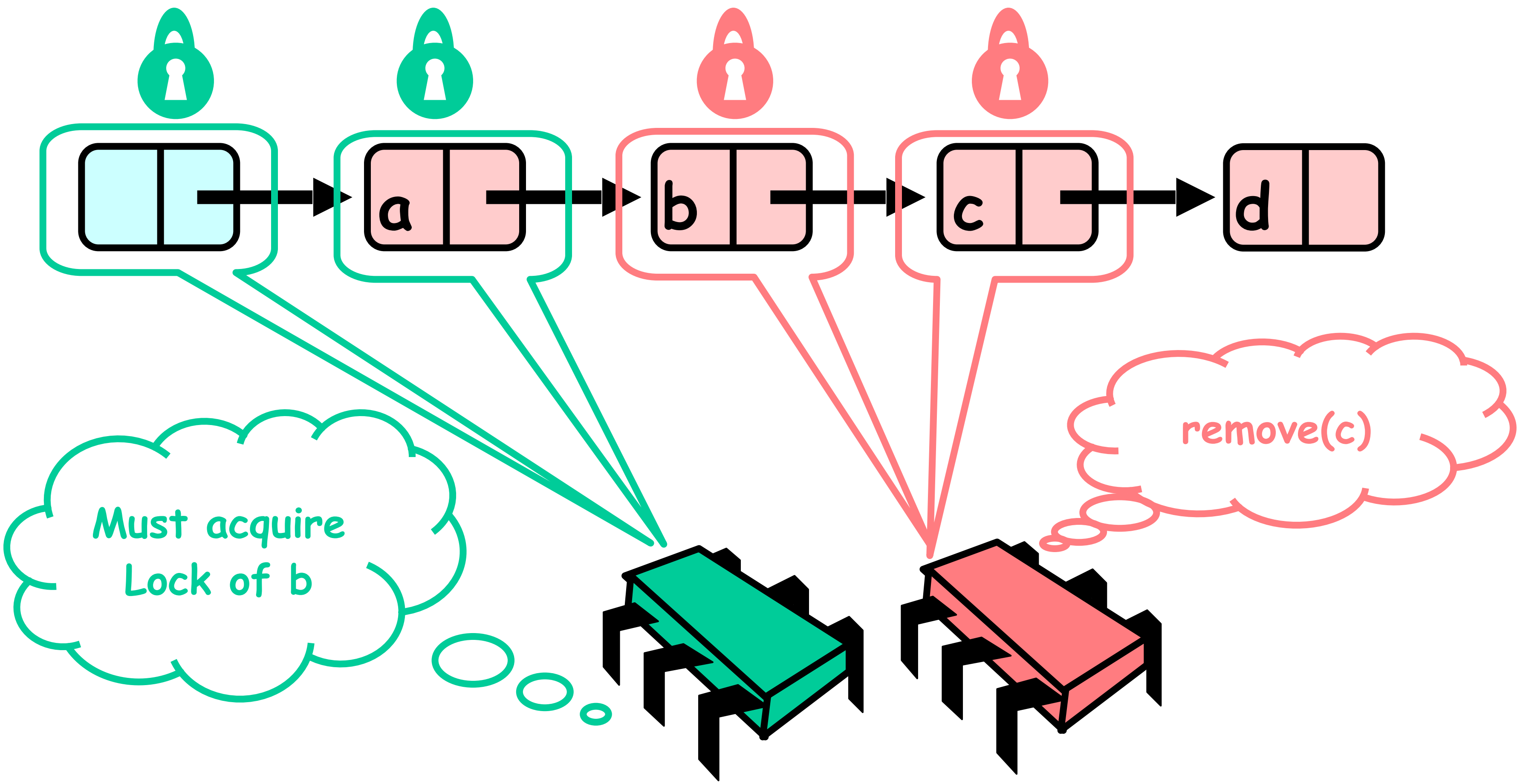
# Removing a Node



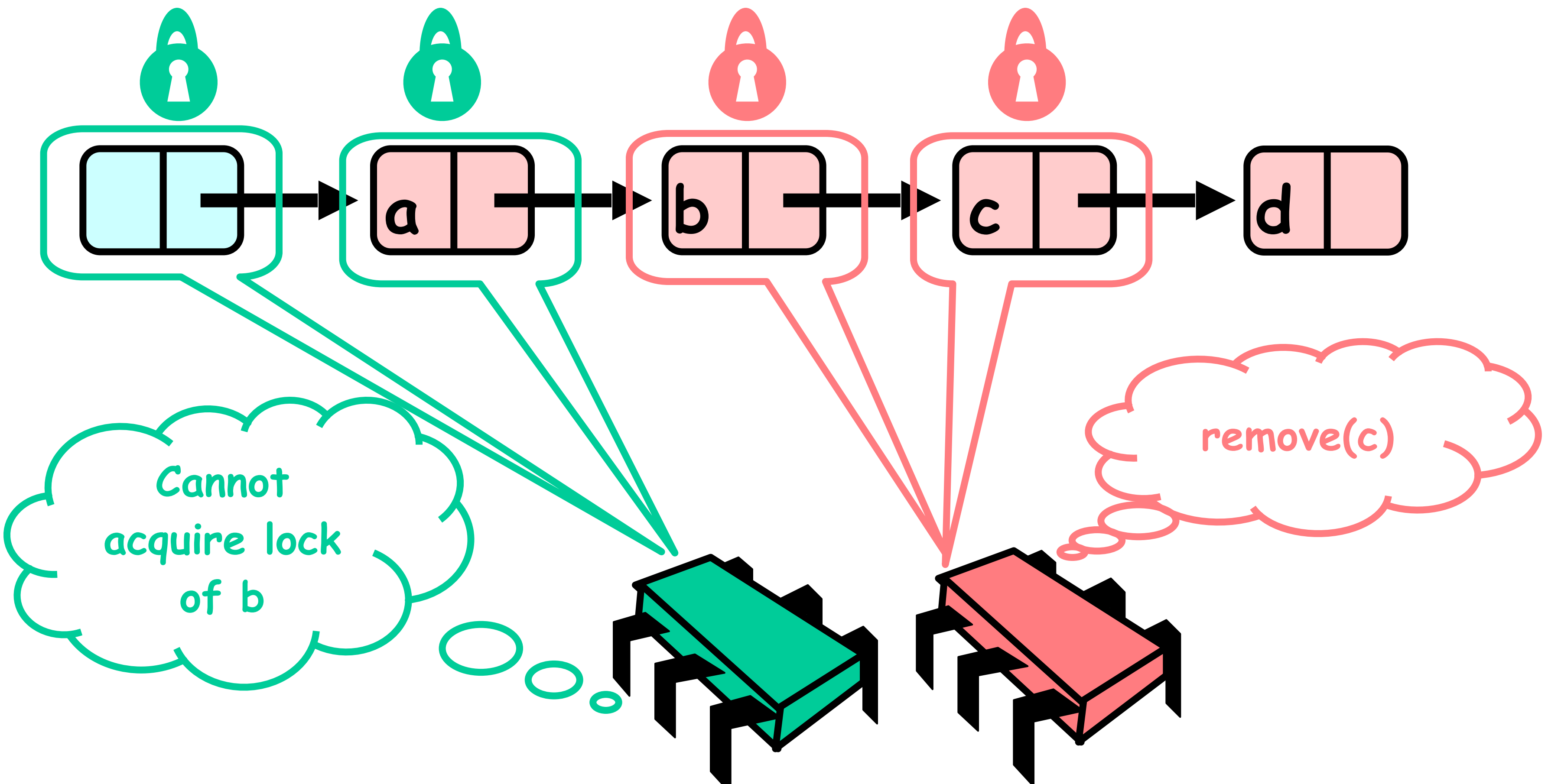
# Removing a Node



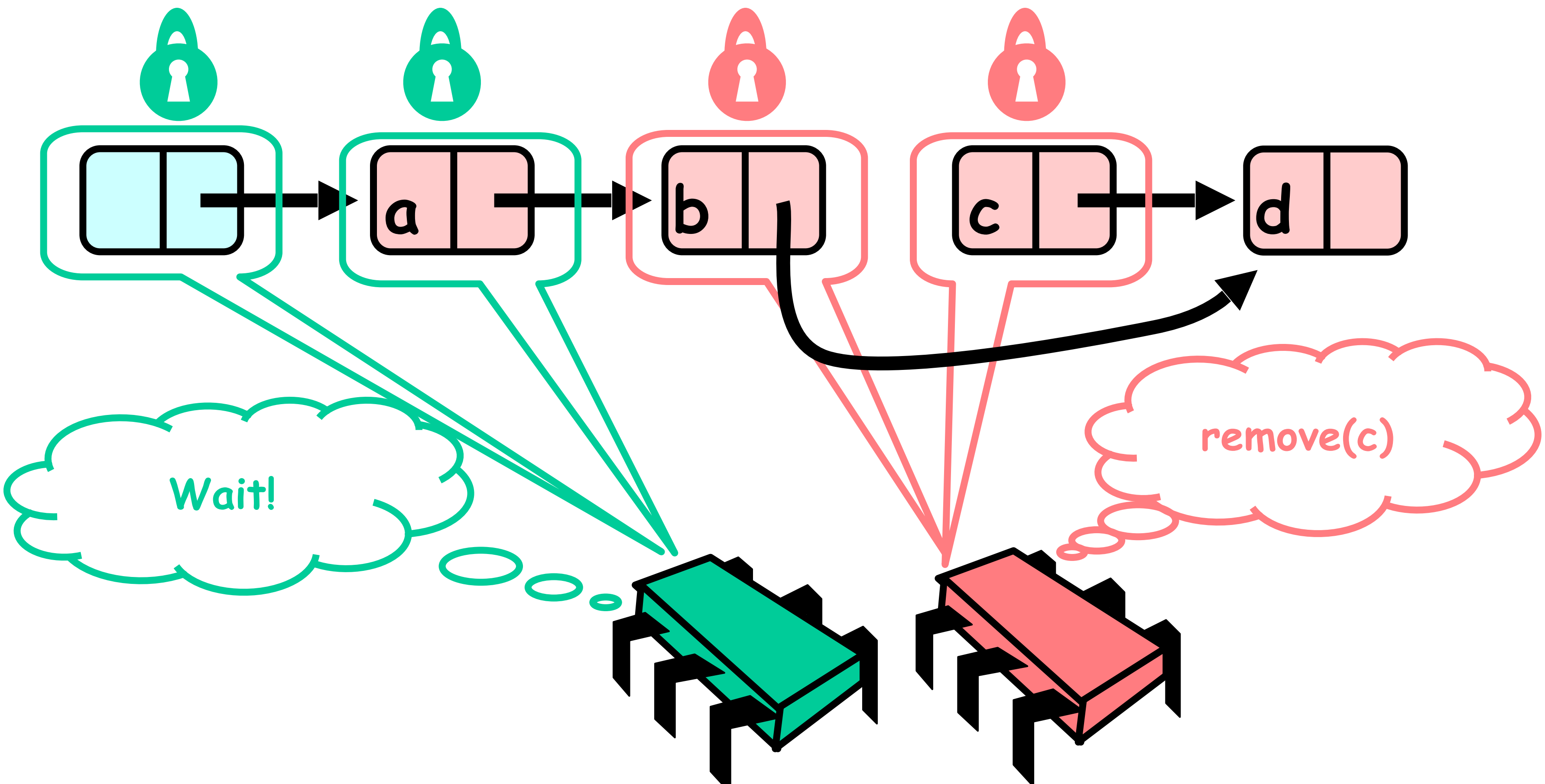
# Removing a Node



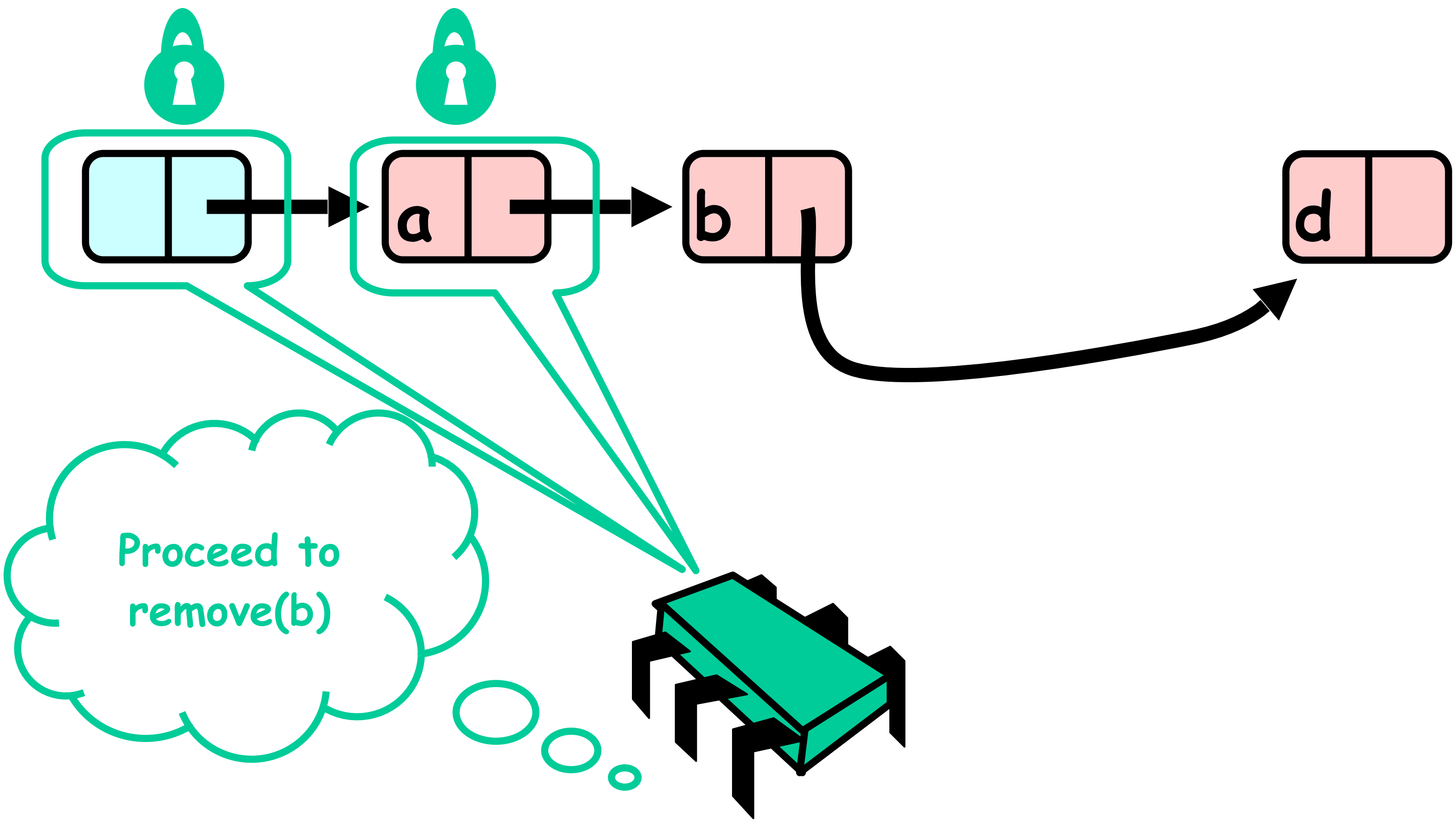
# Removing a Node



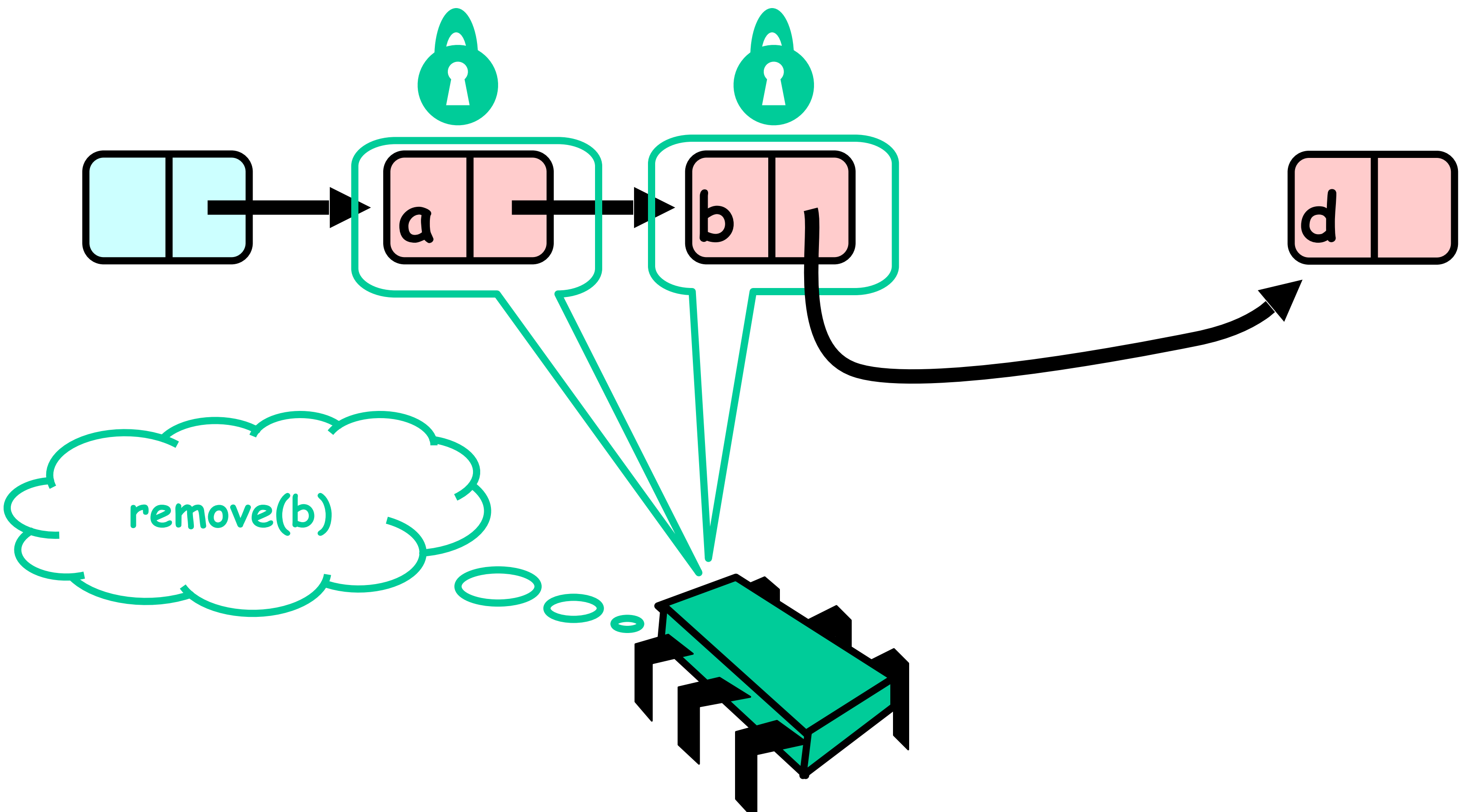
# Removing a Node



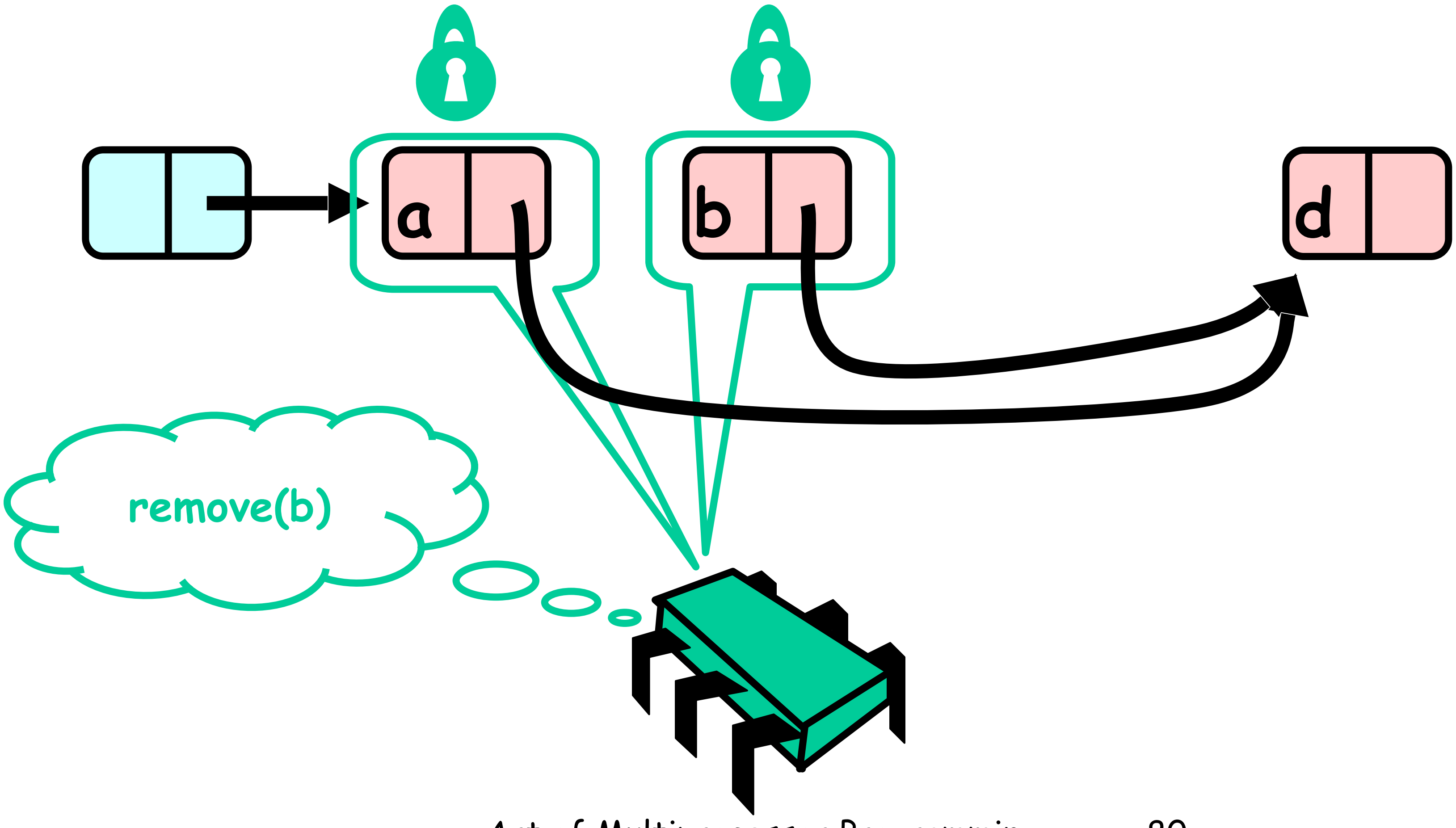
# Removing a Node



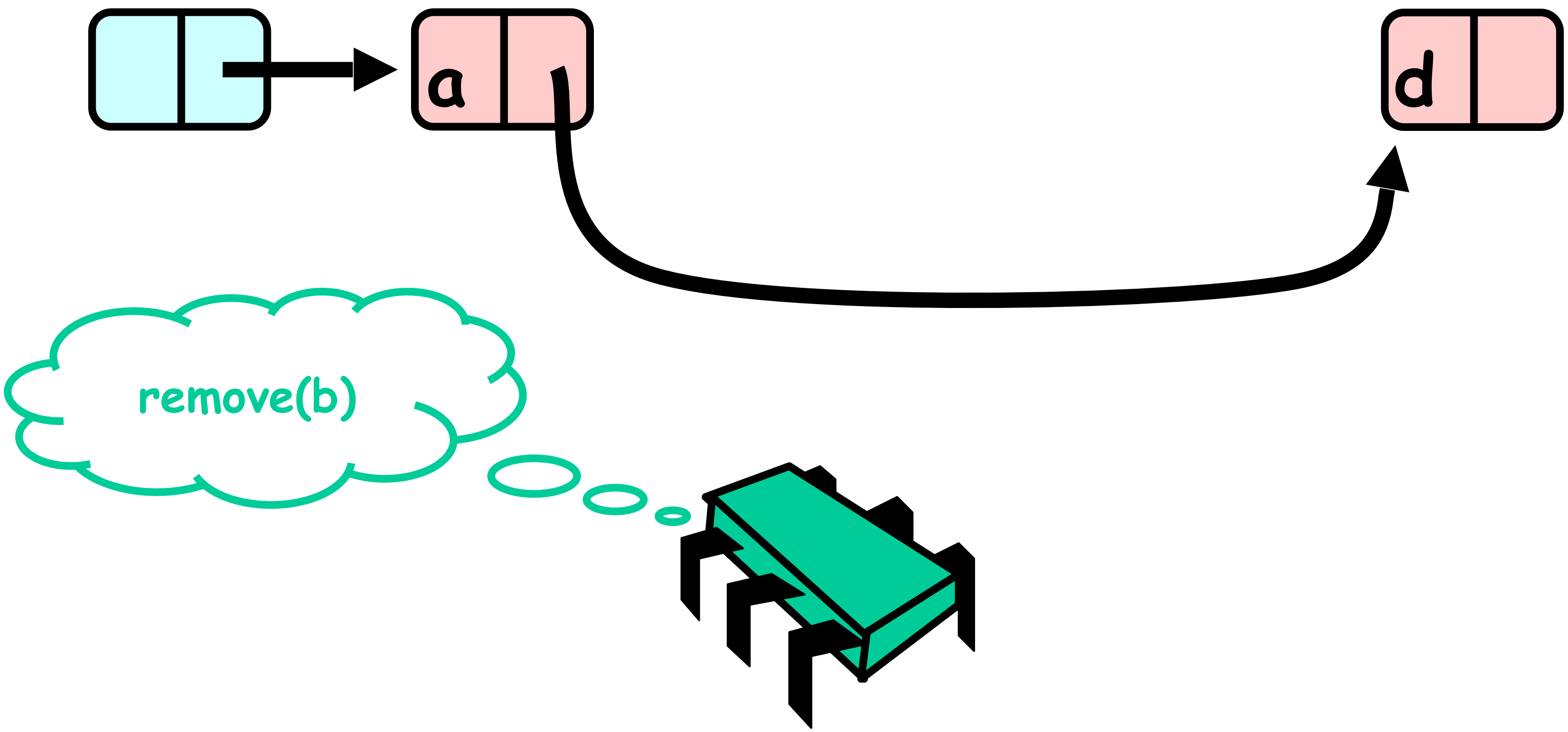
# Removing a Node



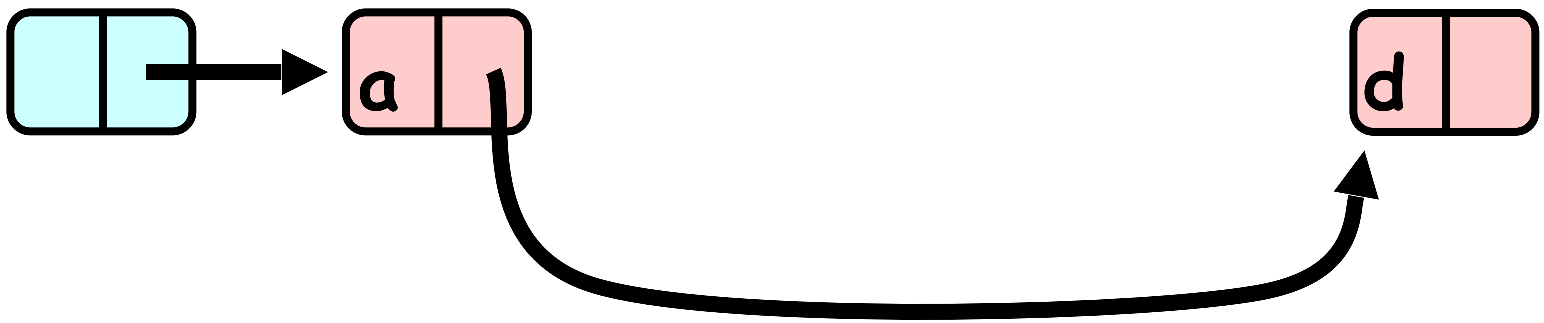
# Removing a Node



# Removing a Node



# Removing a Node



## Fine-Grained Synchronization: hand-over-hand locking Linked List

```
public boolean add(T item) {
    int key = item.hashCode();
    head.lock();
    Node pred = head;
    try {
      Node curr = pred.next;
      curr.lock();
      try {
        while (curr.key < key) {
          pred.unlock();
          pred = curr;
          curr = curr.next;
          curr.lock();
        }
        if (curr.key == key) return false;
        Node newNode = new Node(item);
        newNode.next = curr;
        pred.next = newNode;
        return true;
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
```

```java
public boolean remove(T item) {
   Node pred = null, curr = null;
   int key = item.hashCode();
   head.lock();
   try {
     pred = head;
     curr = pred.next;
     curr.lock();
     try {
       while (curr.key < key) {
         pred.unlock();
         pred = curr;
         curr = curr.next;
         curr.lock();
       }
       if (curr.key == key) {
         pred.next = curr.next;
         return true;
       }
       return false;
     } finally {
       curr.unlock();
     }
   } finally {
     pred.unlock();
   }
 }
```

```java
 public boolean contains(T item) {
    Node last = null, pred = null, curr
   = null;
    int key = item.hashCode();
    head.lock();
    try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
        while (curr.key < key) {
          pred.unlock();
          pred = curr;
          curr = curr.next;
          curr.lock();
        }
        return (curr.key == key);
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
}
```

# Adding Nodes

- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted

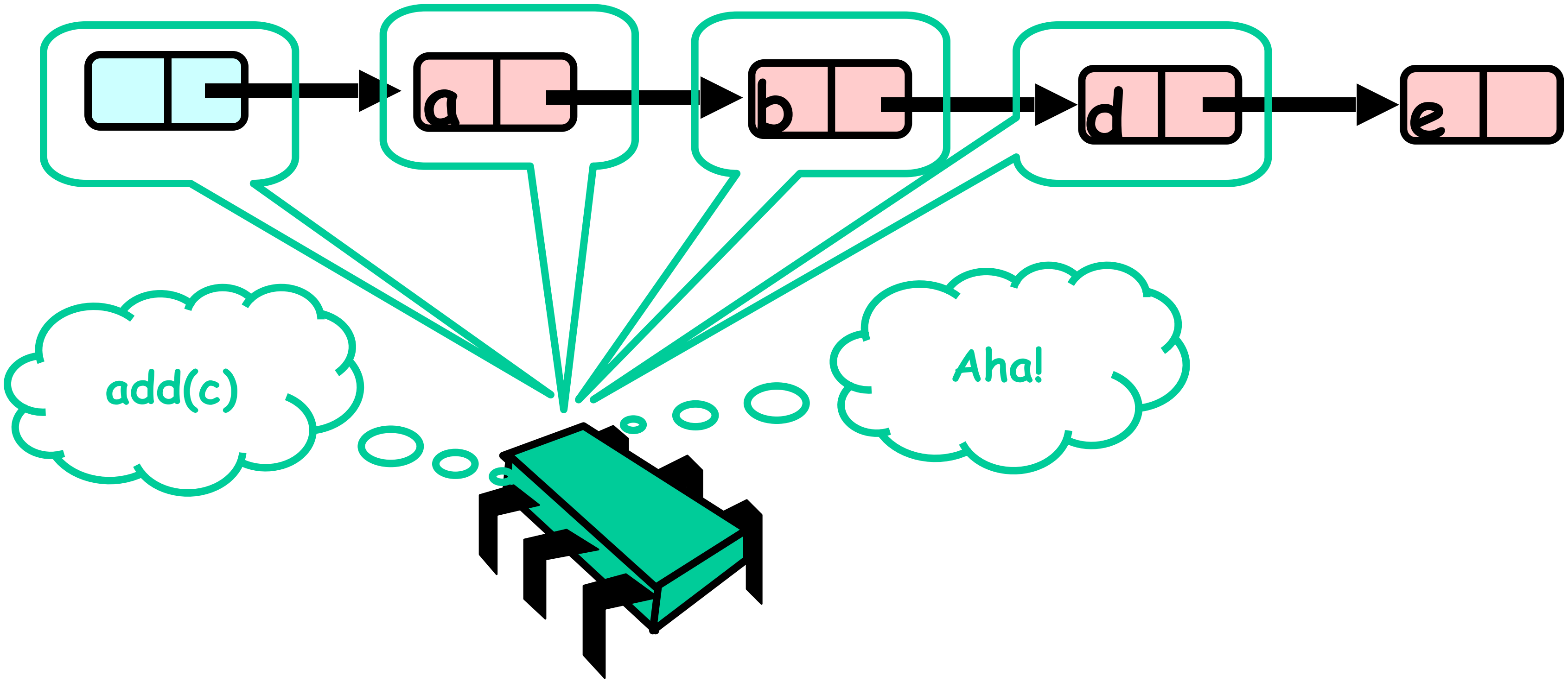
# Drawbacks

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

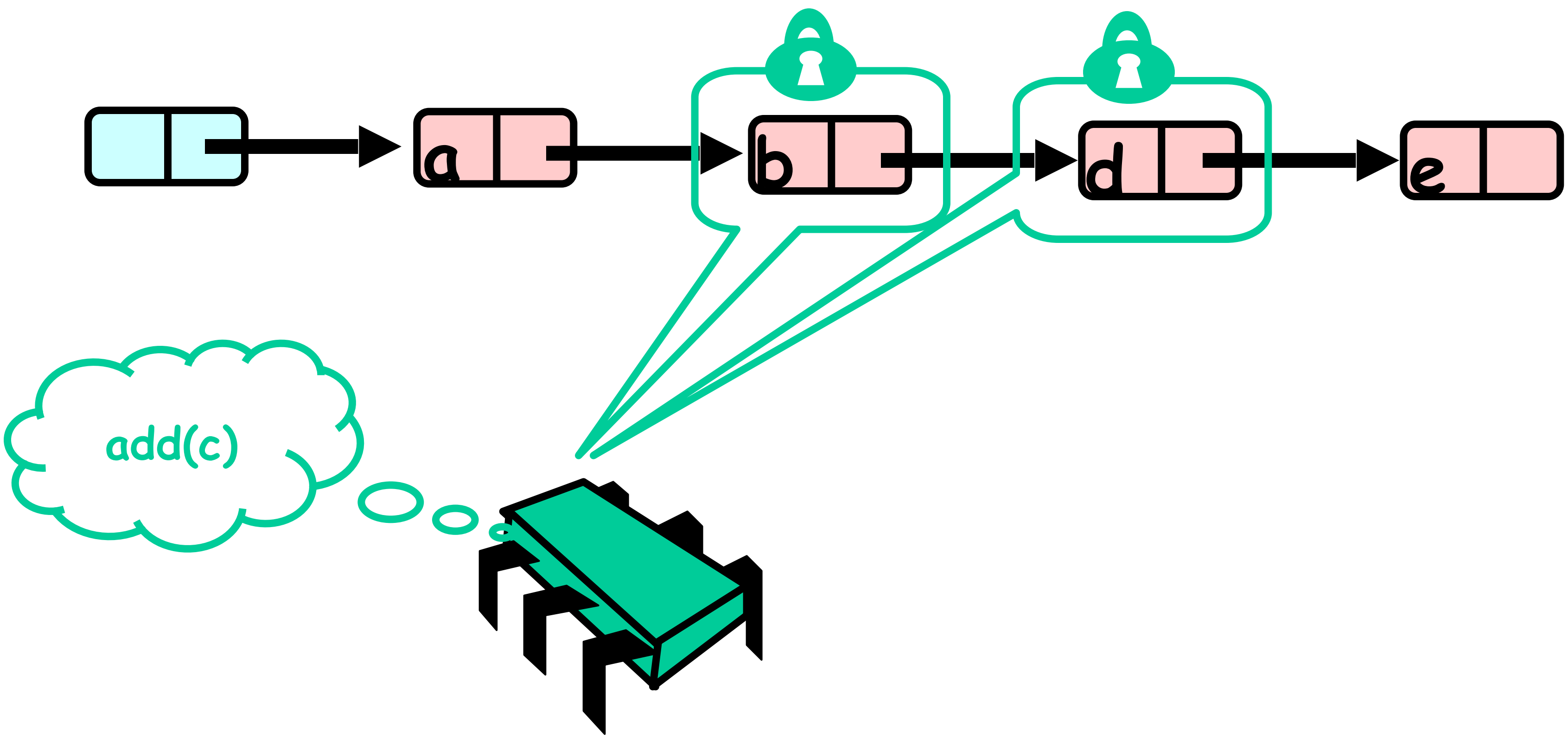
# Optimistic Synchronization

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

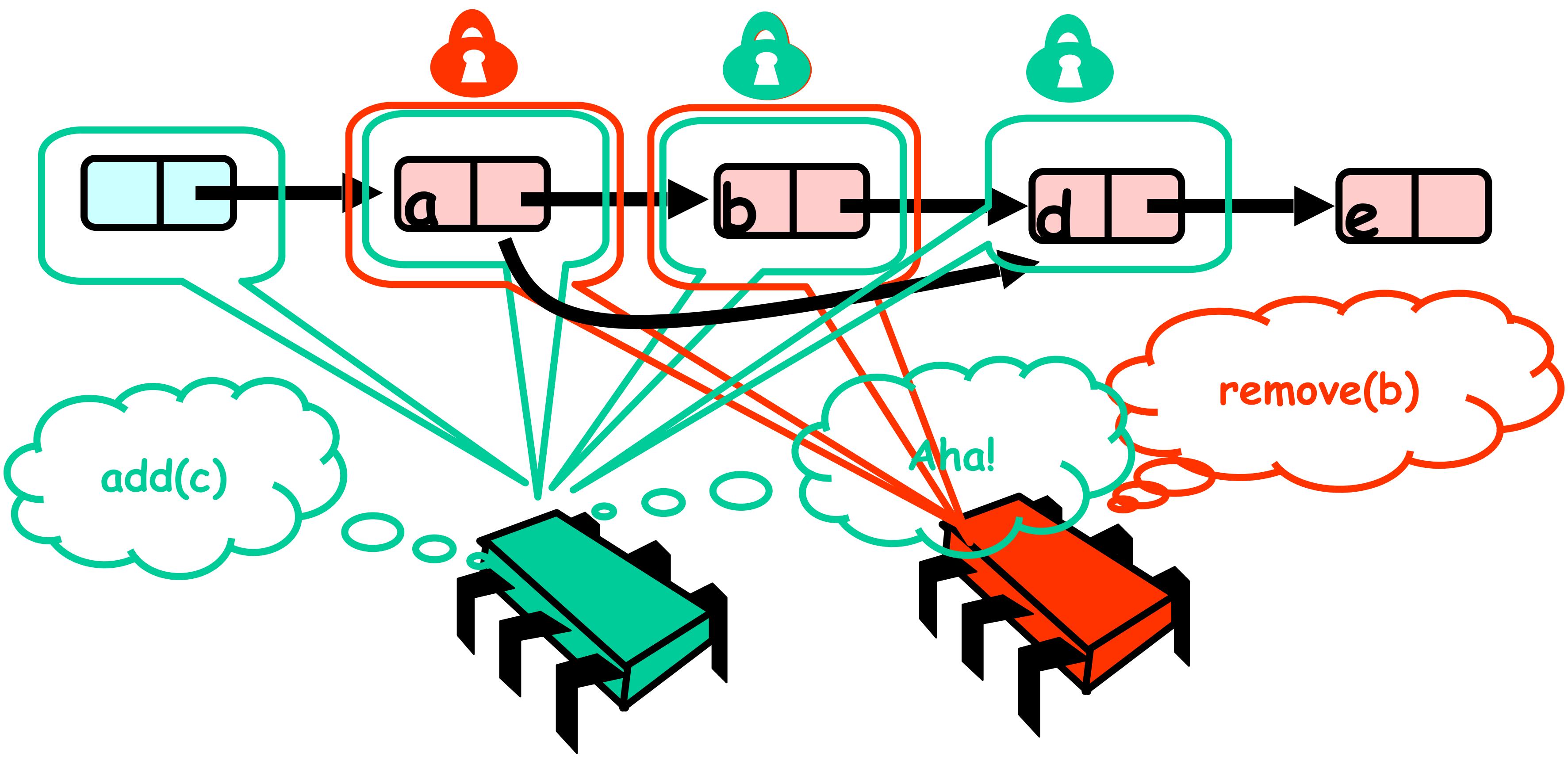
# Optimistic: Traverse without Locking



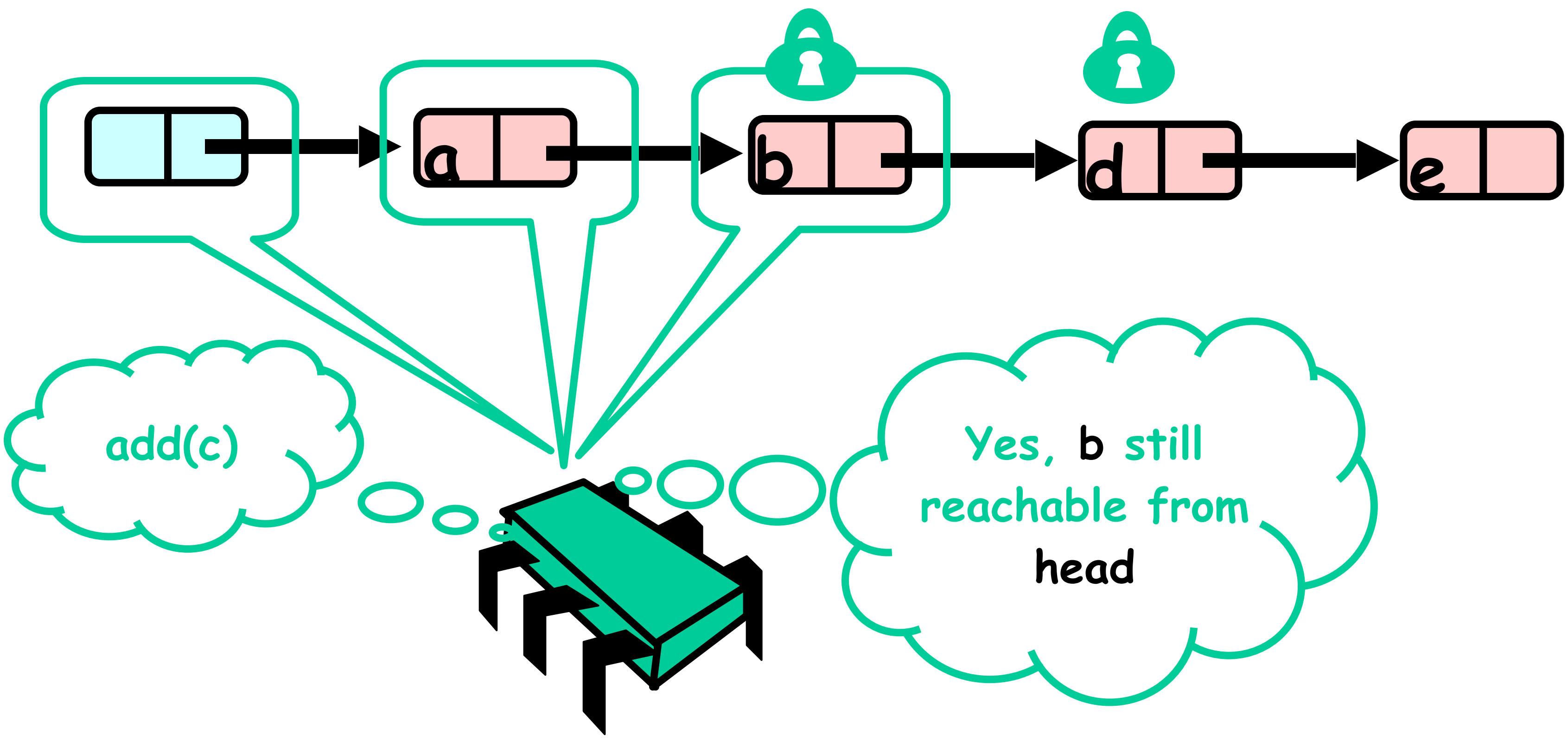
# Optimistic: Lock and Load



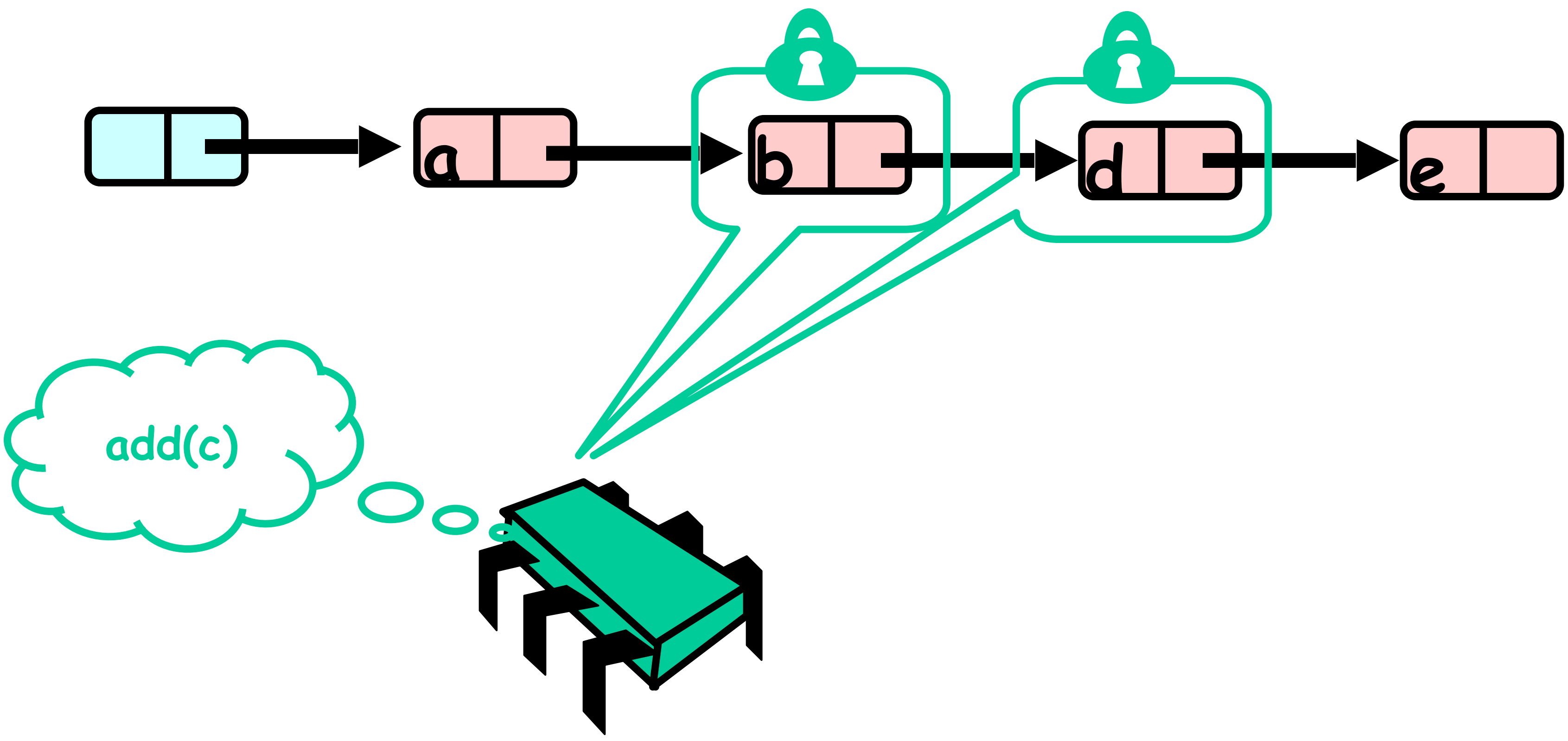
# What could go wrong?



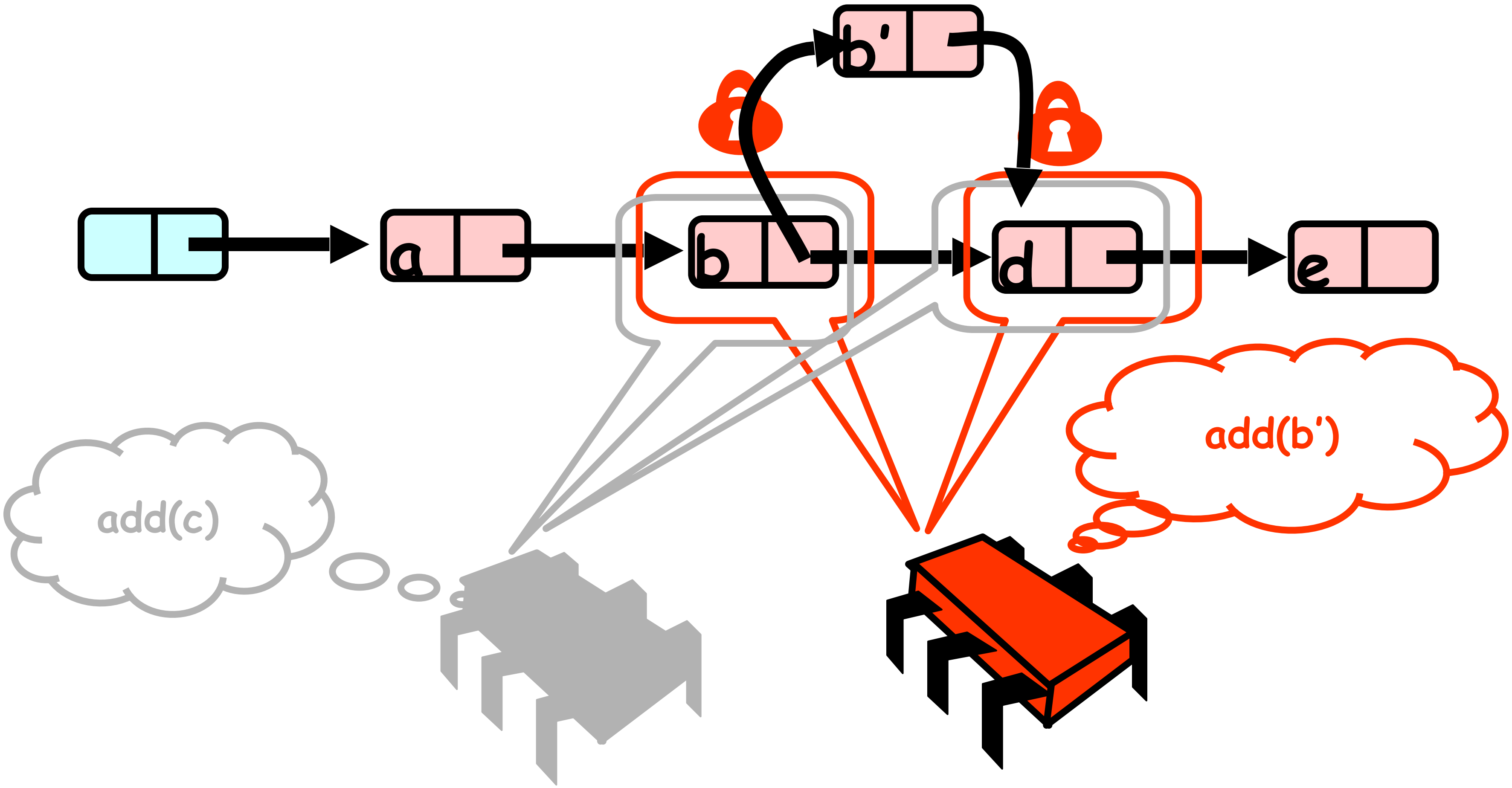
# Validate – Part 1
# (while holding locks)



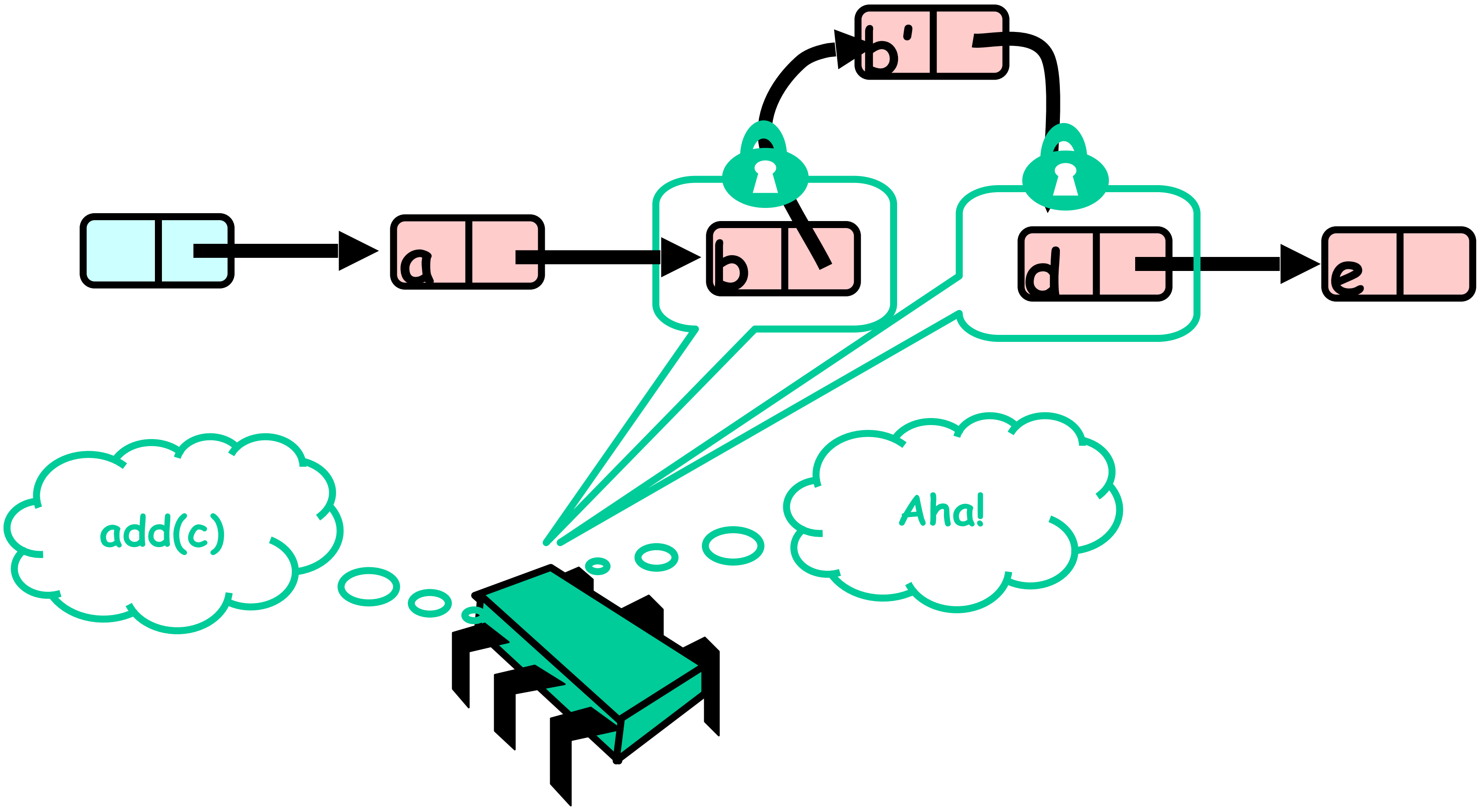
# What Else Can Go Wrong?



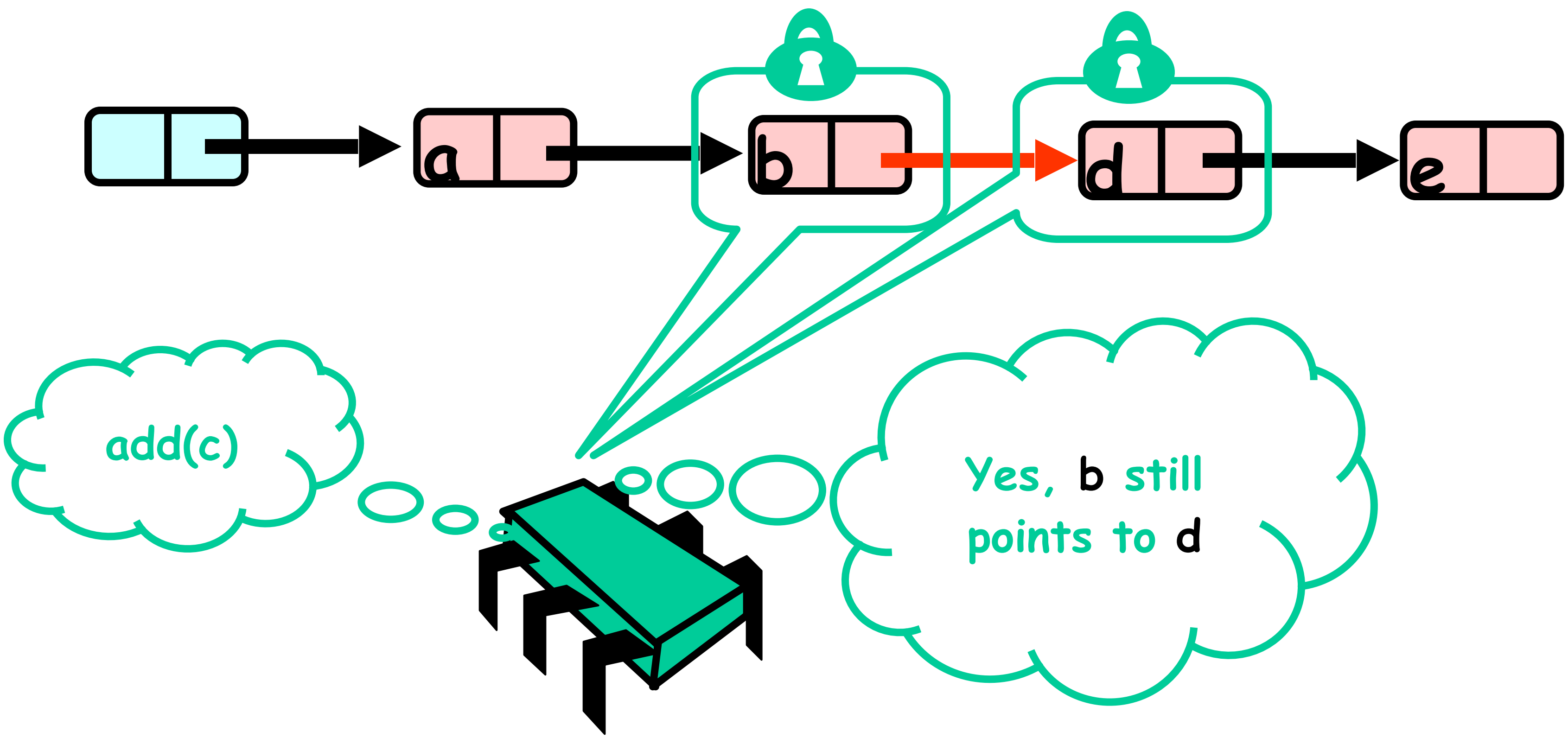
# What Else Can Go Wrong?



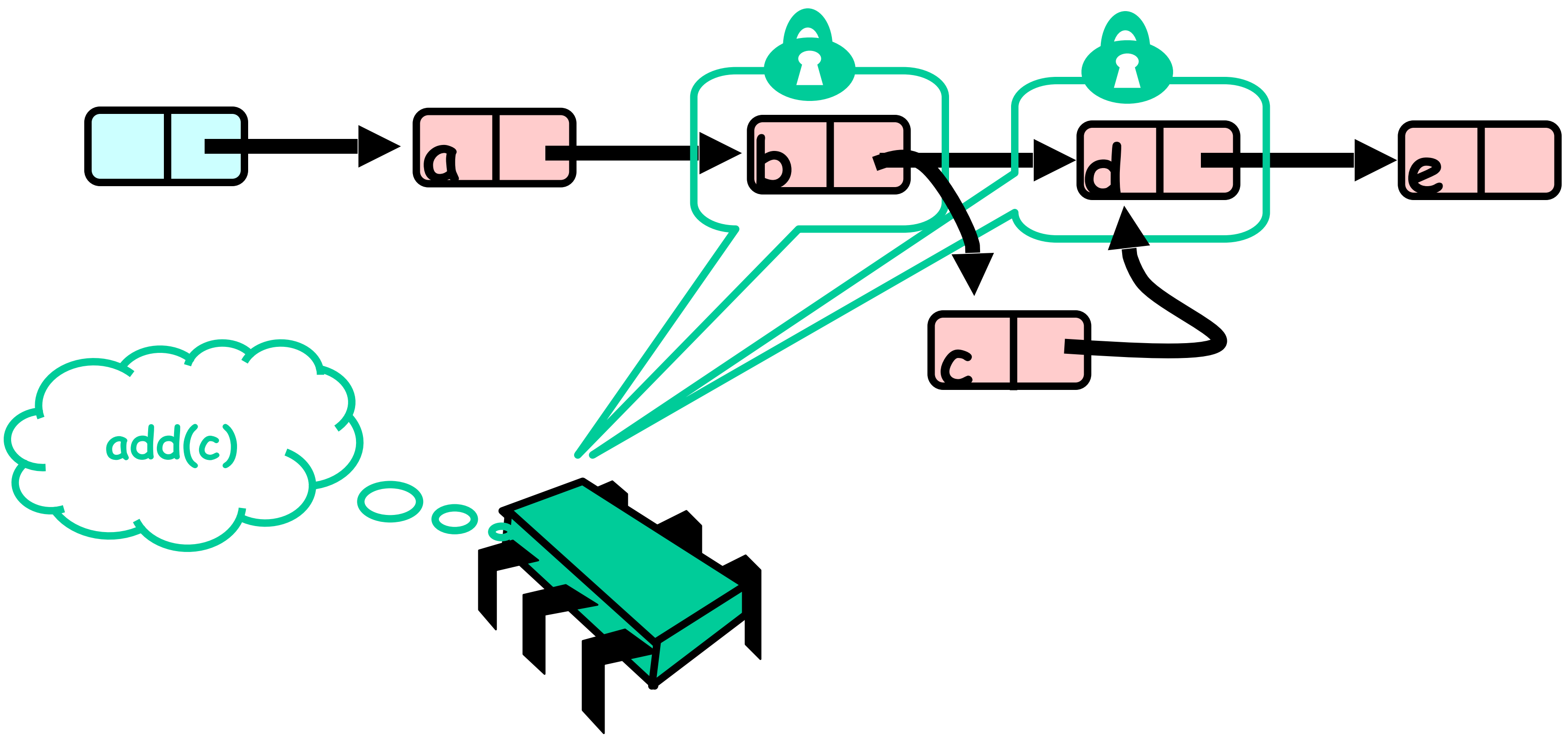
# What Else Can Go Wrong?



# Validate Part 2
# (while holding locks)



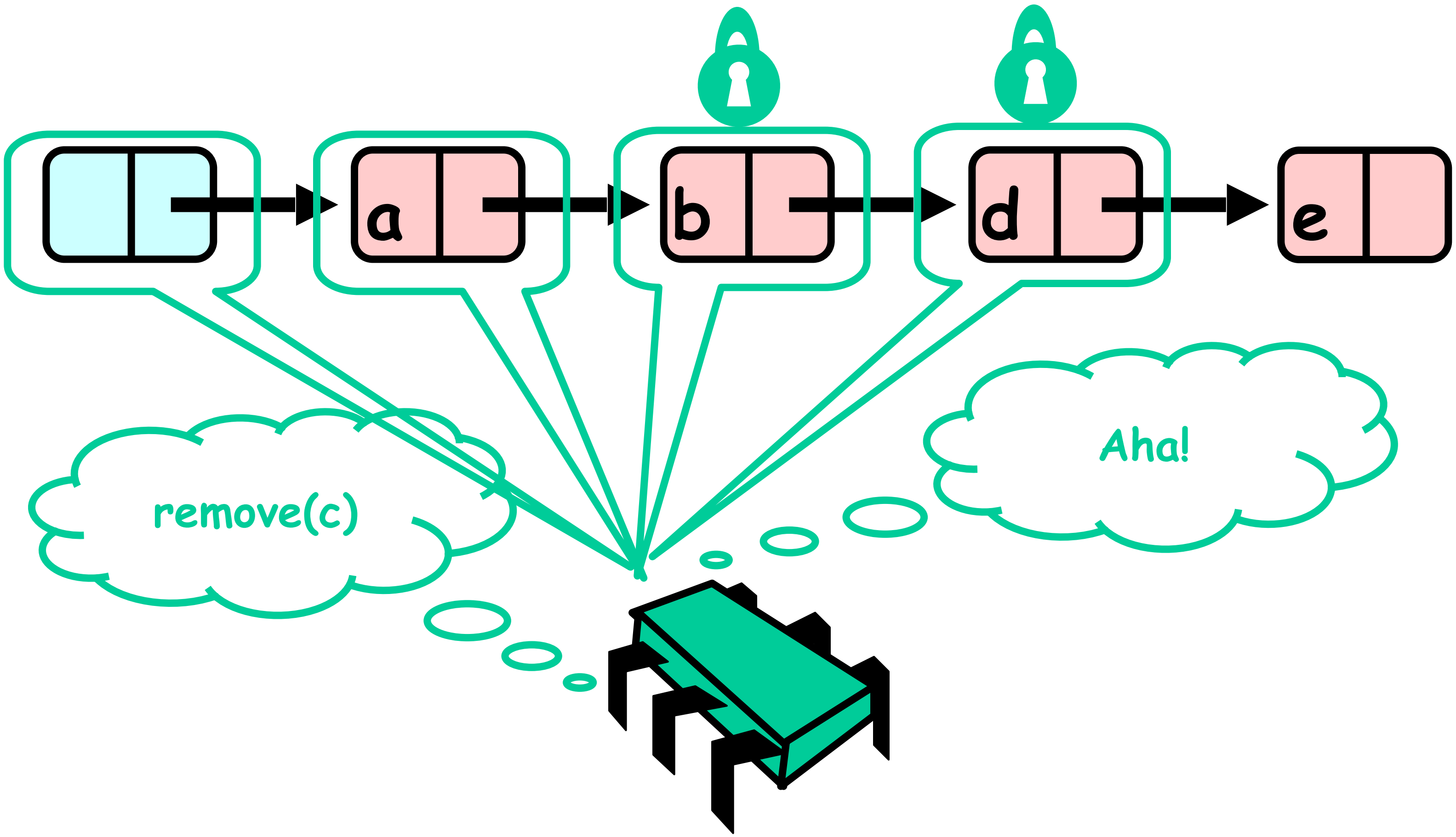
# Optimistic: Critical Point



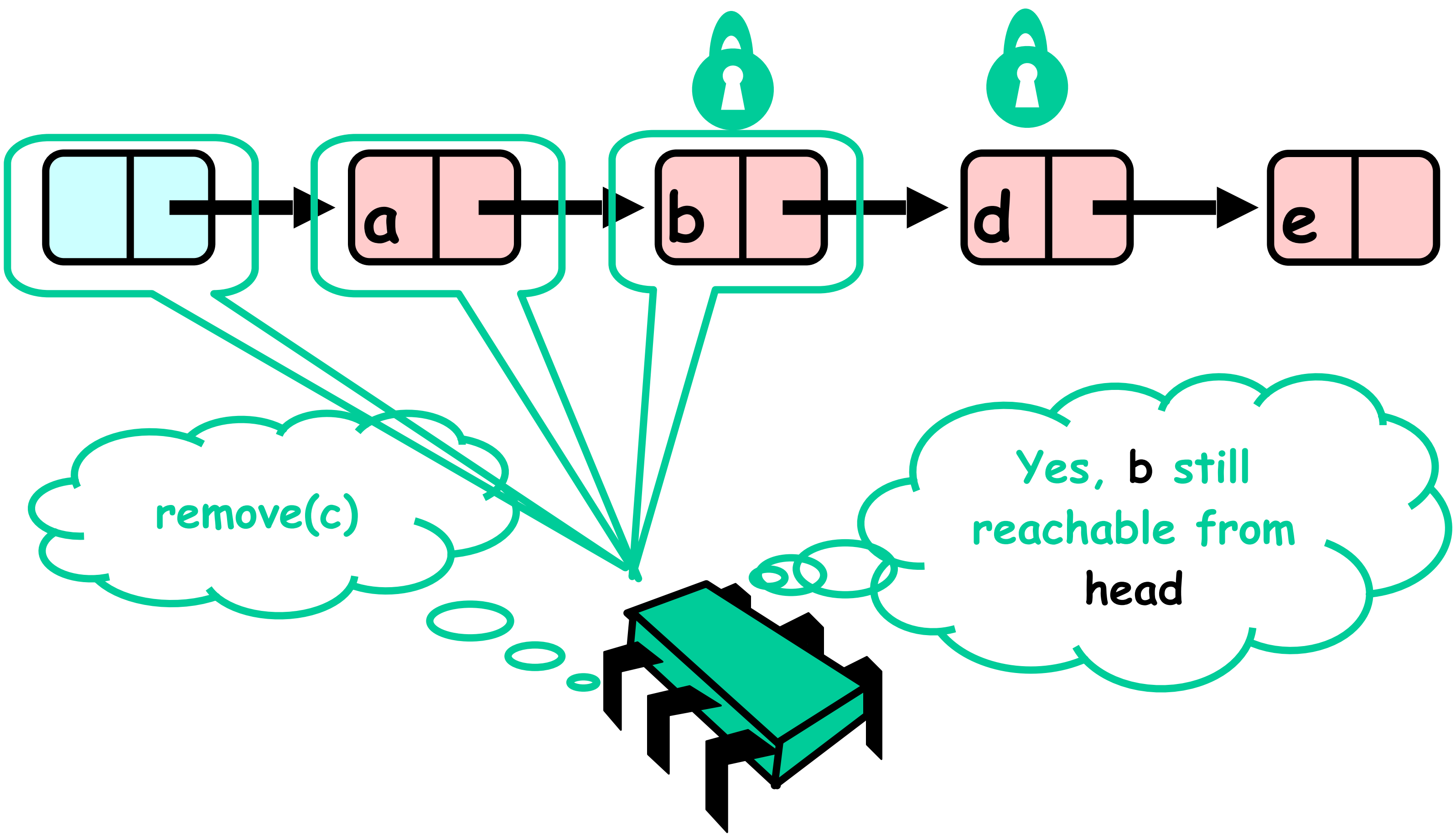
# Correctness

- If
  - Nodes b and c both locked
  - Node b still accessible
  - Node c still successor to b
- Then
  - Neither will be deleted
  - OK to delete and return true

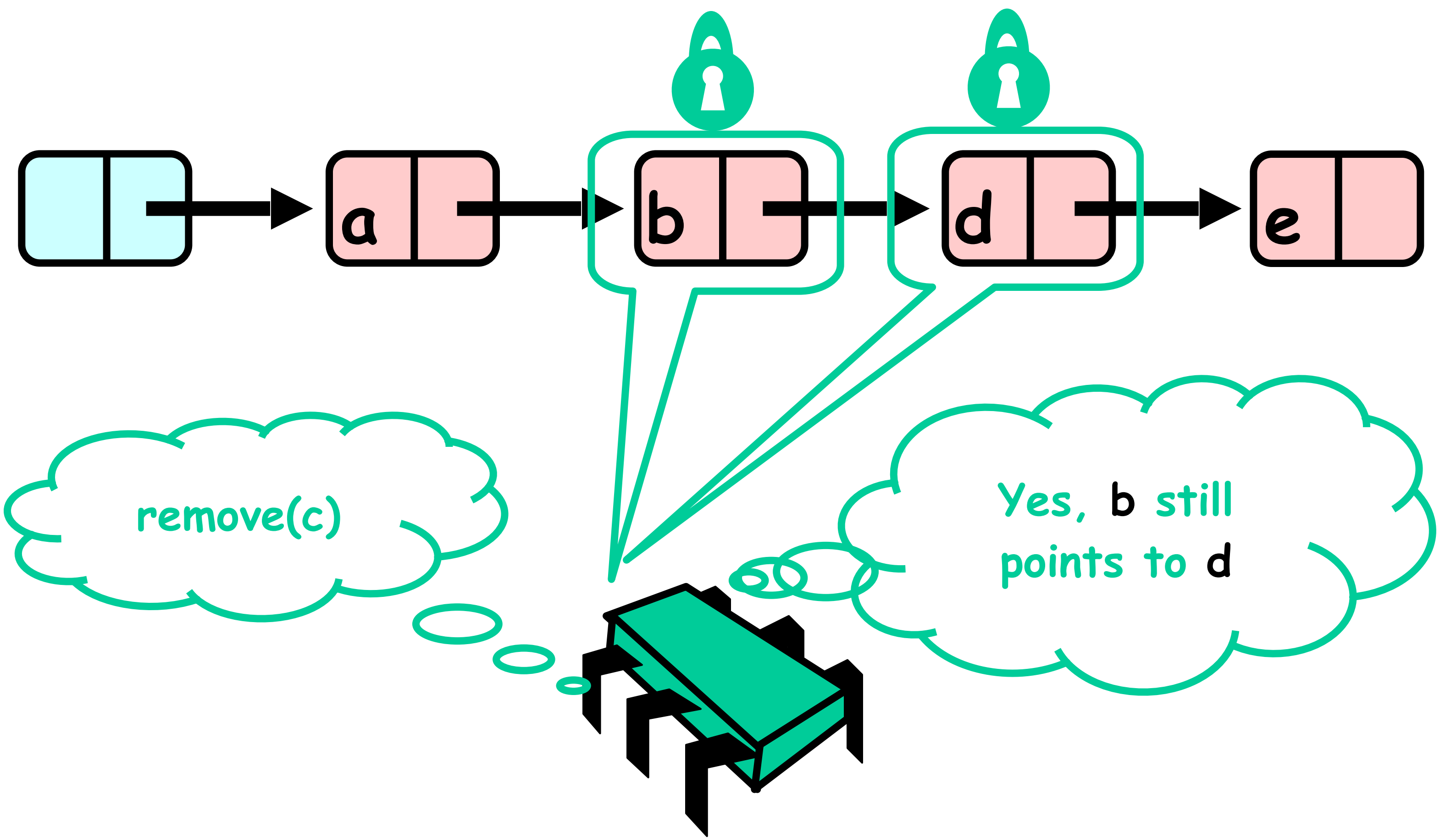
# Unsuccessful Remove



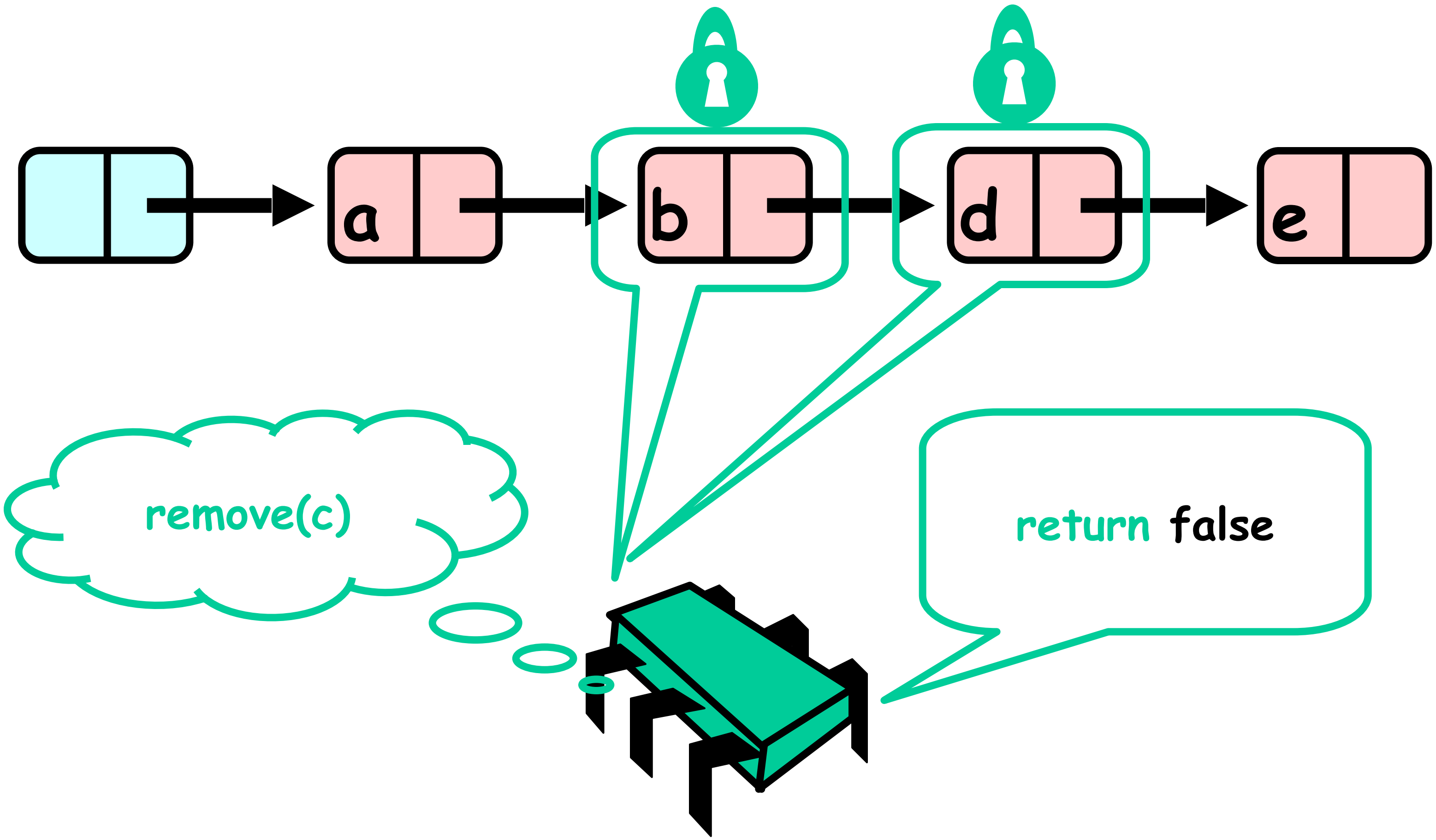
# Validate (1)



# Validate (2)



# OK Computer



# Correctness

- If
  - Nodes b and d both locked
  - Node b still accessible
  - Node d still successor to b
- Then
  - Neither will be deleted
  - No thread can add c after b
  - OK to return false

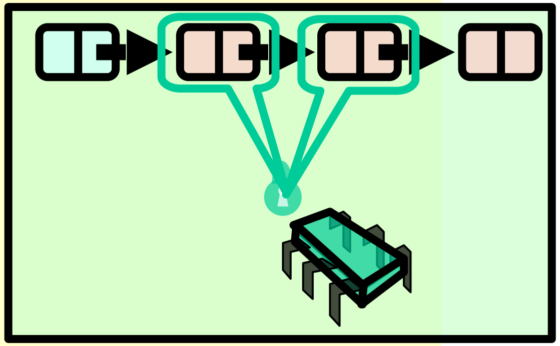
# Validation

```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

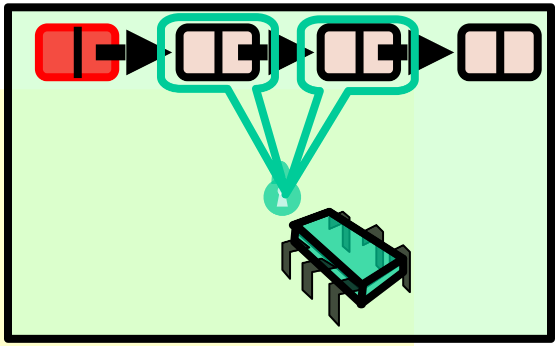
# Validation

```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Predecessor & current nodes



# Validation

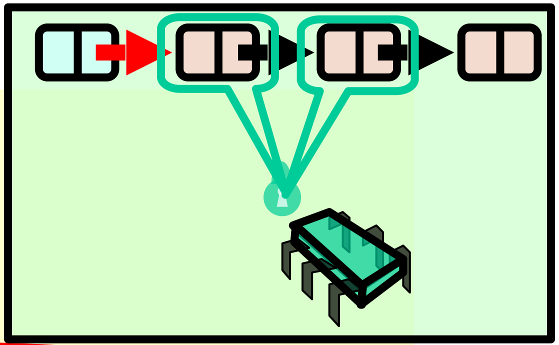


```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Begin at the beginning

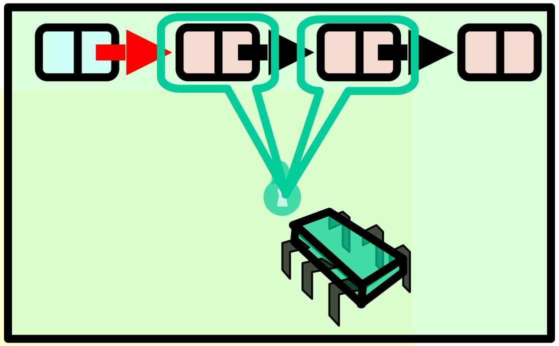
# Validation

```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```



**Search range of keys**

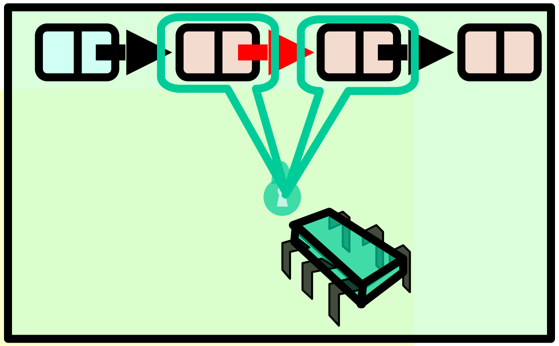
# Validation



```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

**Predecessor reachable**

# Validation



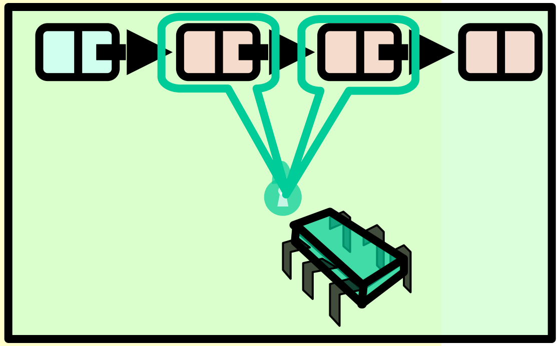
```
private boolean
 validate(Node pred,
          Node curry) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

**Is current node next?**

# Validation

```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

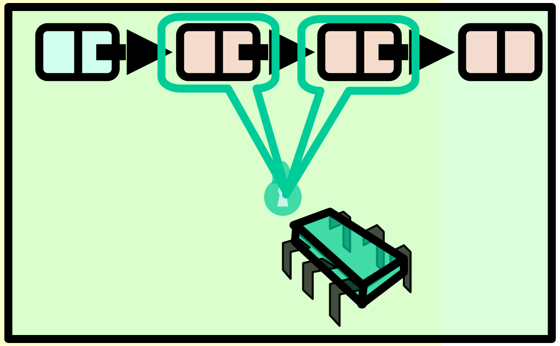
**Otherwise move on**



# Validation

```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Predecessor not reachable



## Optimistic Synchronization

```
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = this.head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) {
          return false;
        } else {
          Node node = new Entry(item);
          entry.next = curr;
          pred.next = node;
          return true;
        }
      }
    } finally {
      pred.unlock(); curr.unlock();
    }
  }
}
```

```java
public boolean remove(T item) {
   int key = item.hashCode();
   while (true) {
     Node pred = this.head;
     Node curr = pred.next;
     while (curr.key < key) {
       pred = curr; curr = curr.next;
     }
     pred.lock(); curr.lock();
     try {
       if (validate(pred, curr)) {
         if (curr.key == key) {
           pred.next = curr.next;
           return true;
         } else {
           return false;
         }
       }
     } finally {
       pred.unlock(); curr.unlock();
     }
   }
 }
```

```java
public boolean contains(T item) {
   int key = item.hashCode();
   while (true) {
     Node pred = this.head;
     Node curr = pred.next;
     while (curr.key < key) {
       pred = curr; curr = curr.next;
     }
     try {
       pred.lock(); curr.lock();
       if (validate(pred, curr)) {
         return (curr.key == key);
       }
     } finally {
       pred.unlock(); curr.unlock();
     }
   }
 }
```

```java
private boolean validate(Node pred, Node
  curr) {
  Node node = head;
  while (node.key <= pred.key) {
    if (node == pred)
      return pred.next == curr;
    Node = node.next;
  }
  return false;
}
```

# Optimistic List

- Limited hot-spots
  - Targets of add(), remove(), contains()
  - No contention on traversals
- Moreover
  - Traversals are wait-free
  - Food for thought …

# So Far, So Good

- Much less lock acquisition/release
  - Performance
  - Concurrency
- Problems
  - Need to traverse list twice
  - contains() method acquires locks

# Evaluation

- Optimistic is effective if
  - cost of scanning twice without locks is less than
  - cost of scanning once with locks
- Drawback
  - contains() acquires locks
  - 90% of calls in many apps

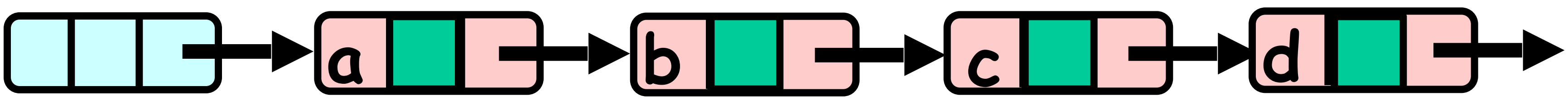
# Lazy List

- Like optimistic, except
  - Scan once
  - contains(x) never locks …
- Key insight
  - Removing nodes causes trouble
  - Do it "lazily"

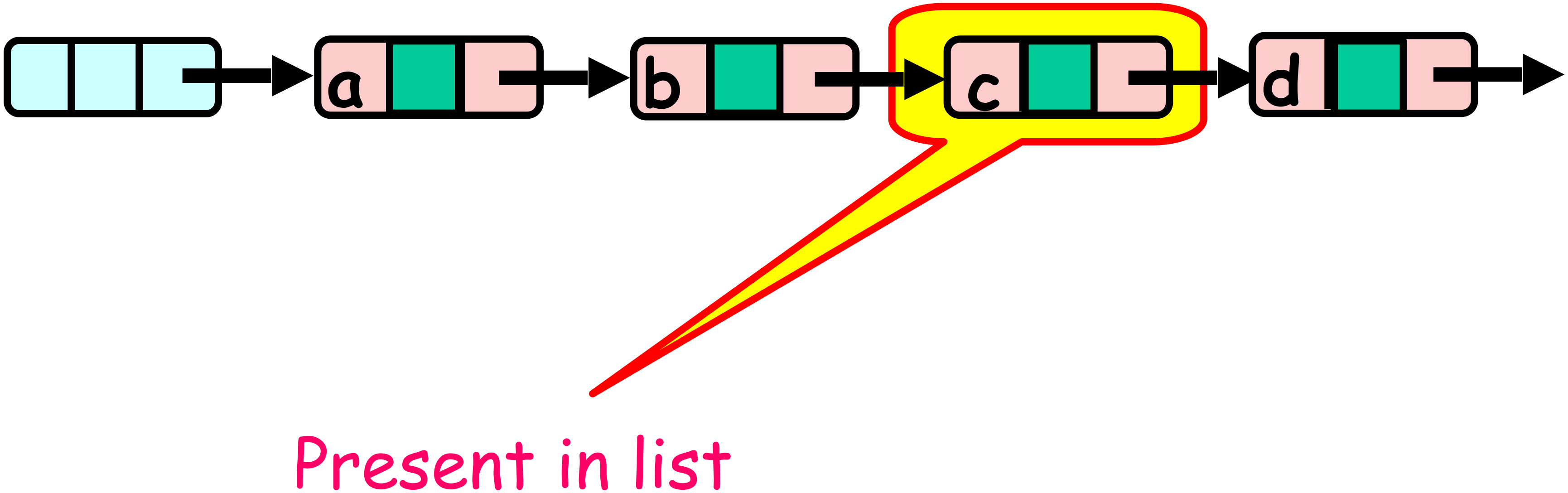
# Lazy List

- remove()
  - Scans list (as before)
  - Locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

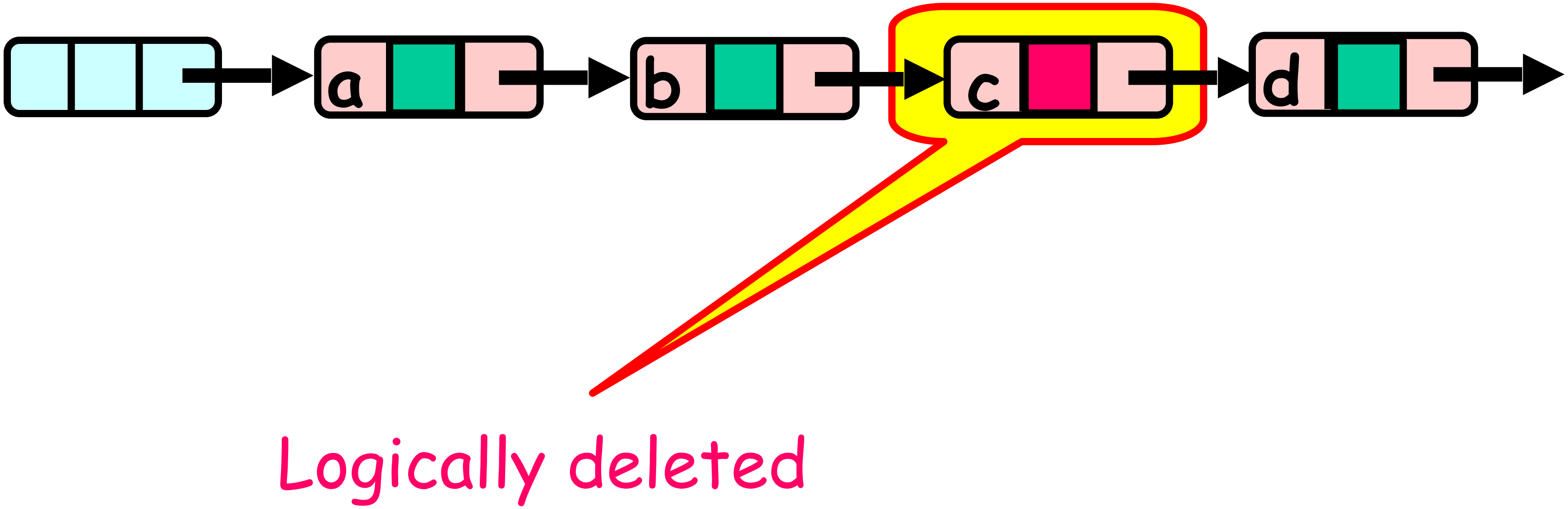
# Lazy Removal



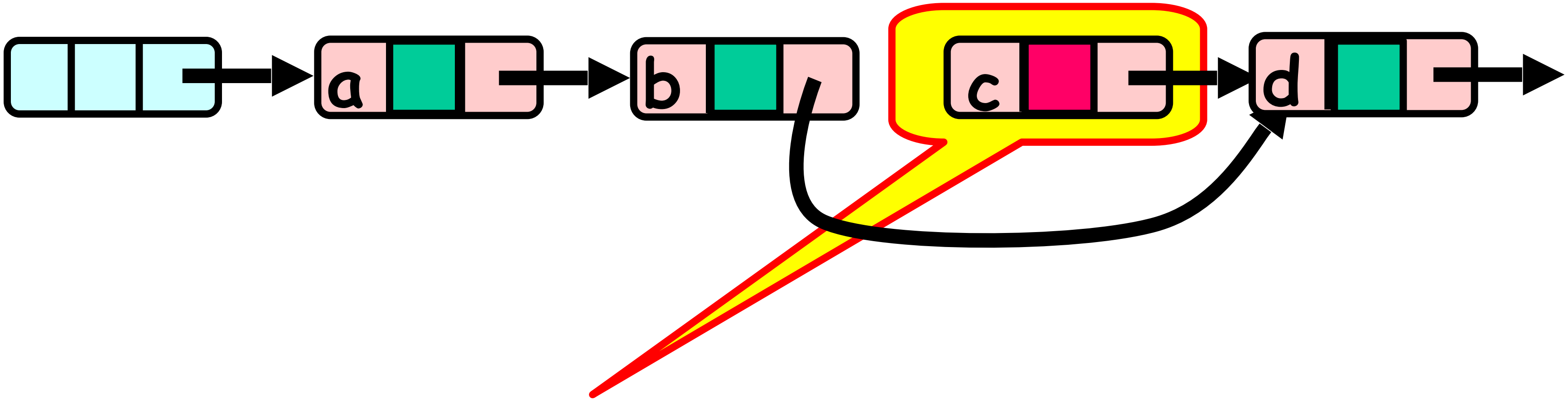
# Lazy Removal



# Lazy Removal

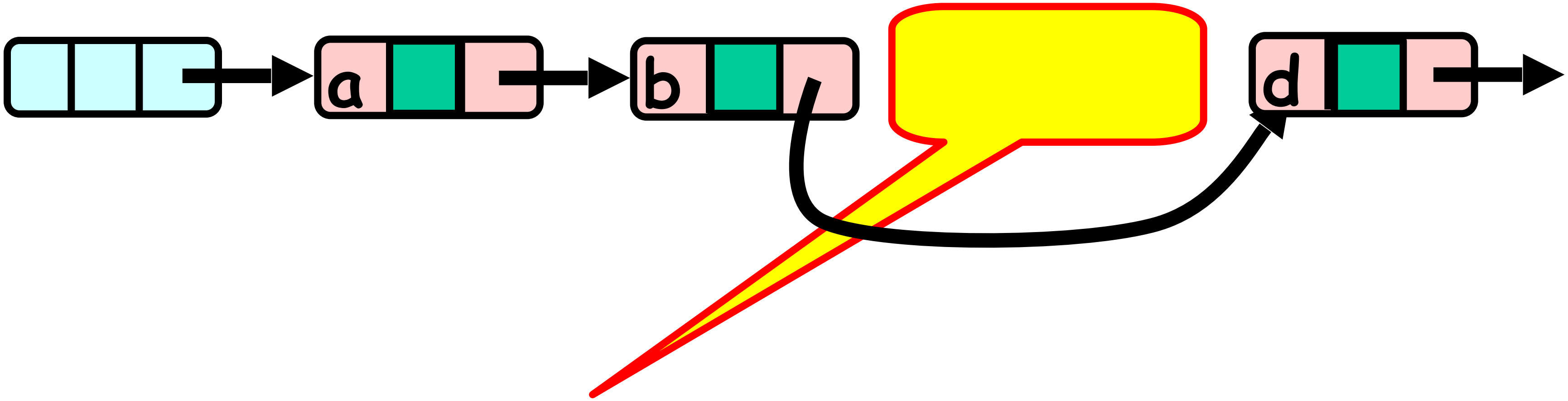


# Lazy Removal




Physically deleted

# Lazy Removal




Physically deleted

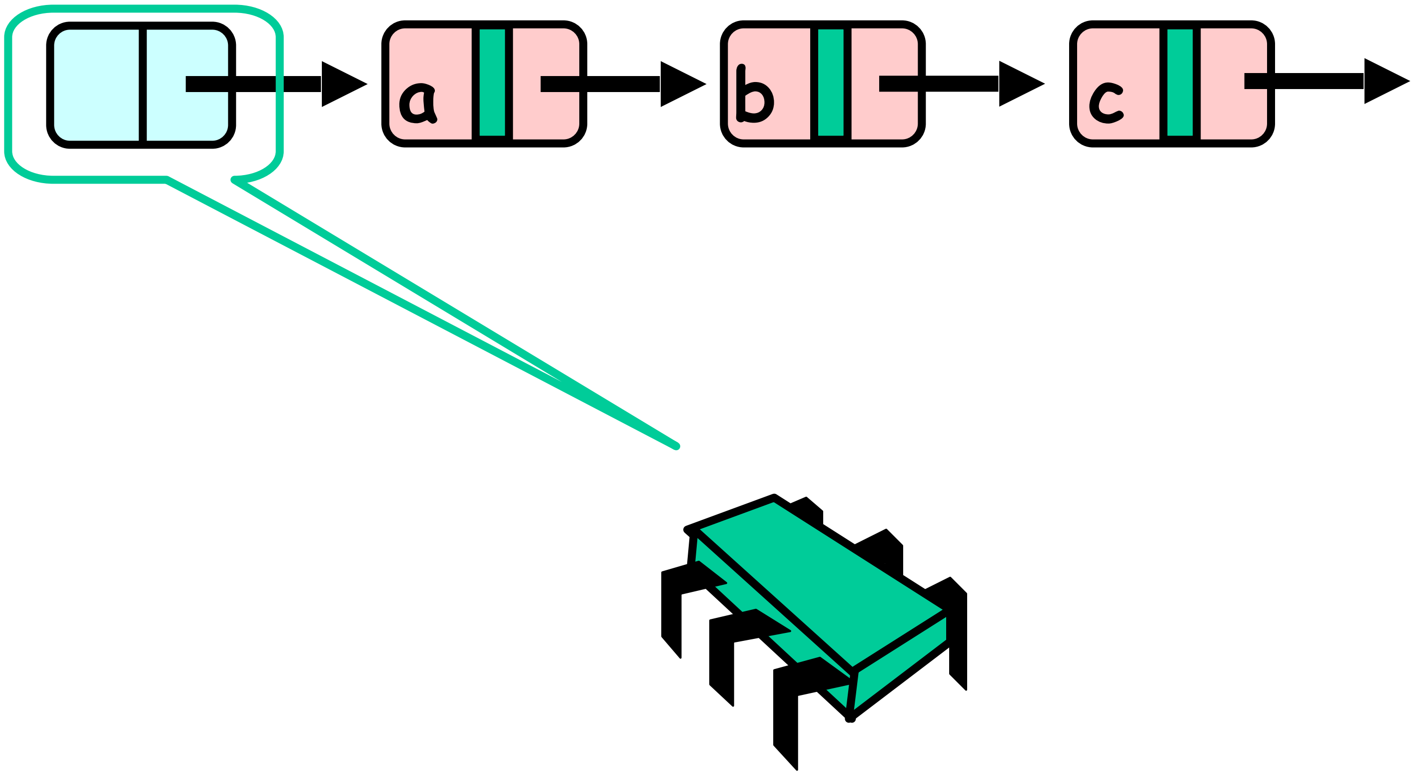
# Lazy List

- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn't slow down other method calls …
- Must still lock pred and curr nodes.

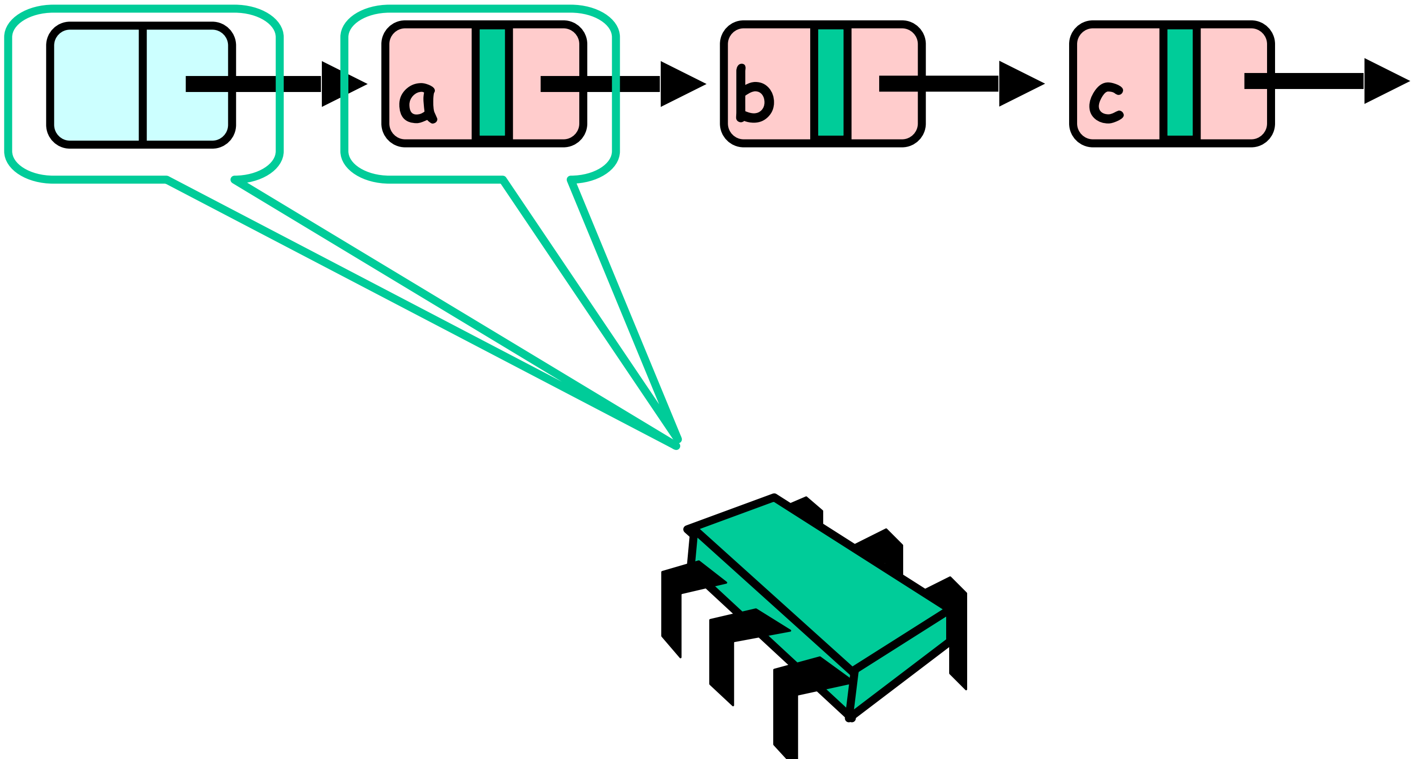
# Validation

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

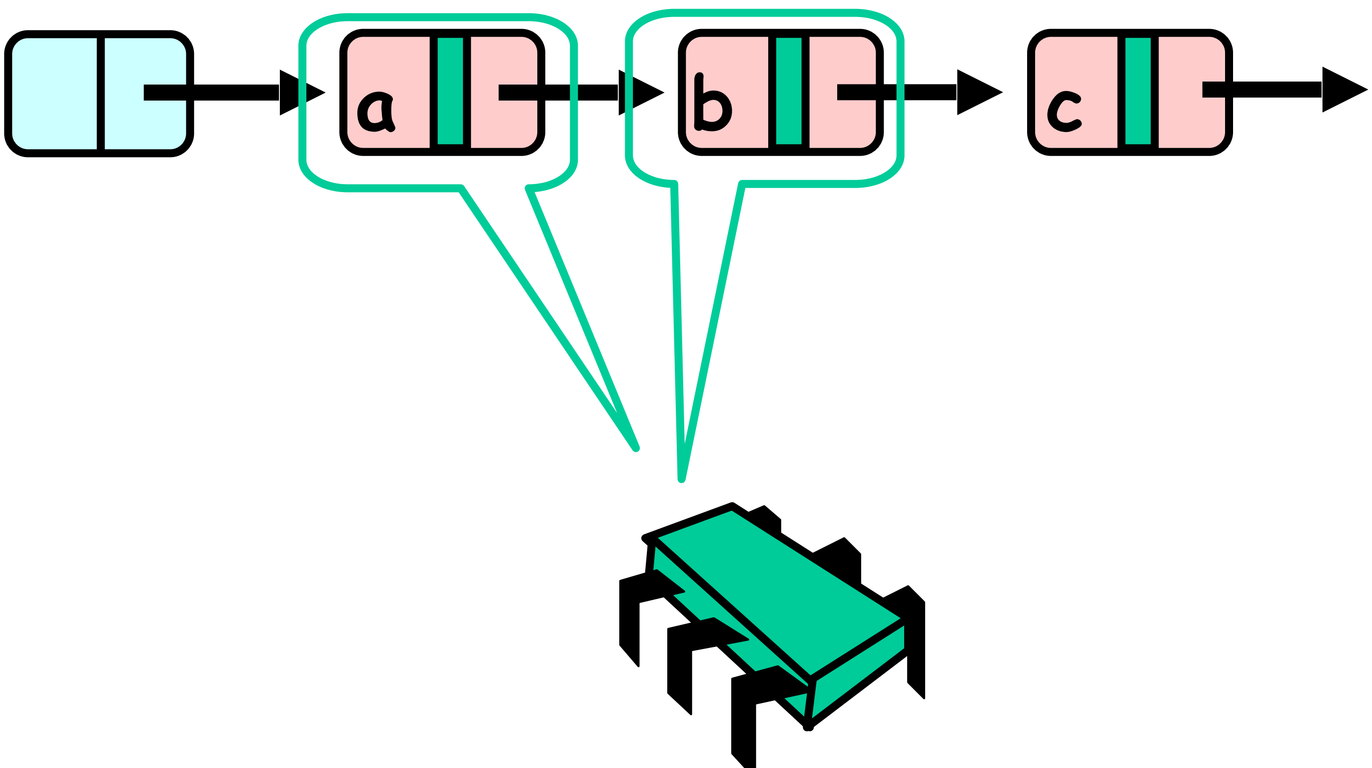
# Business as Usual



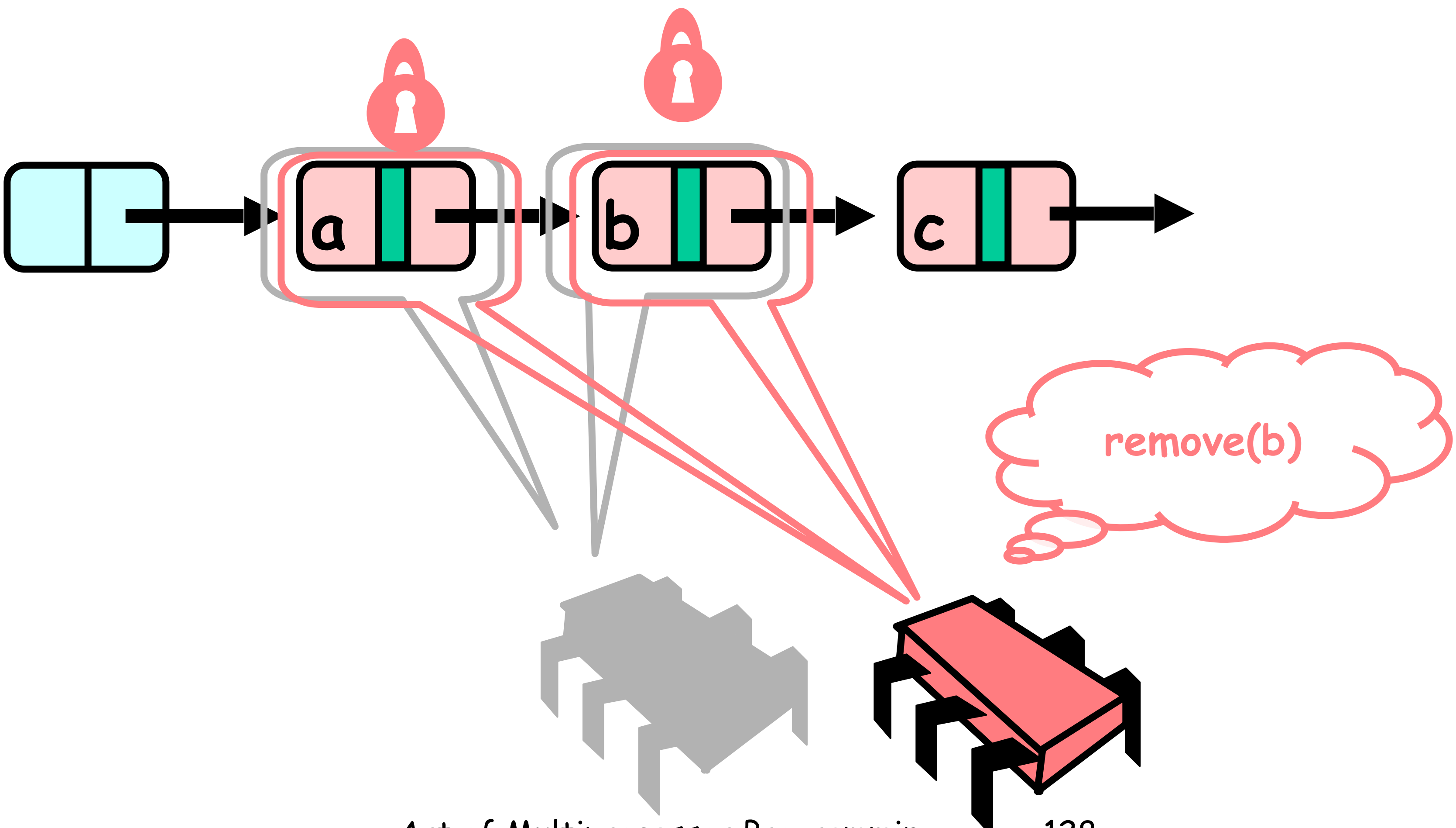
# Business as Usual



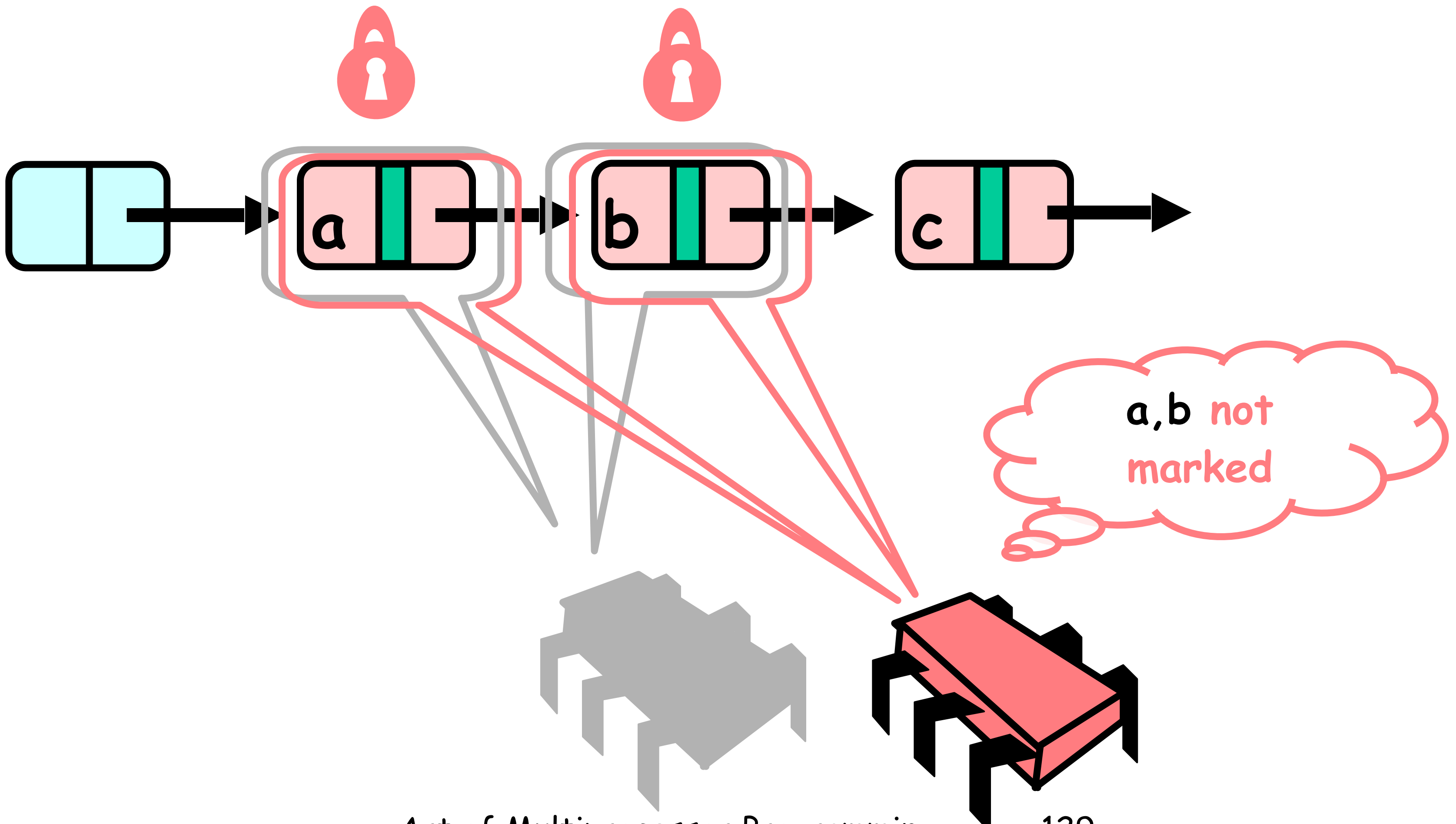
# Business as Usual



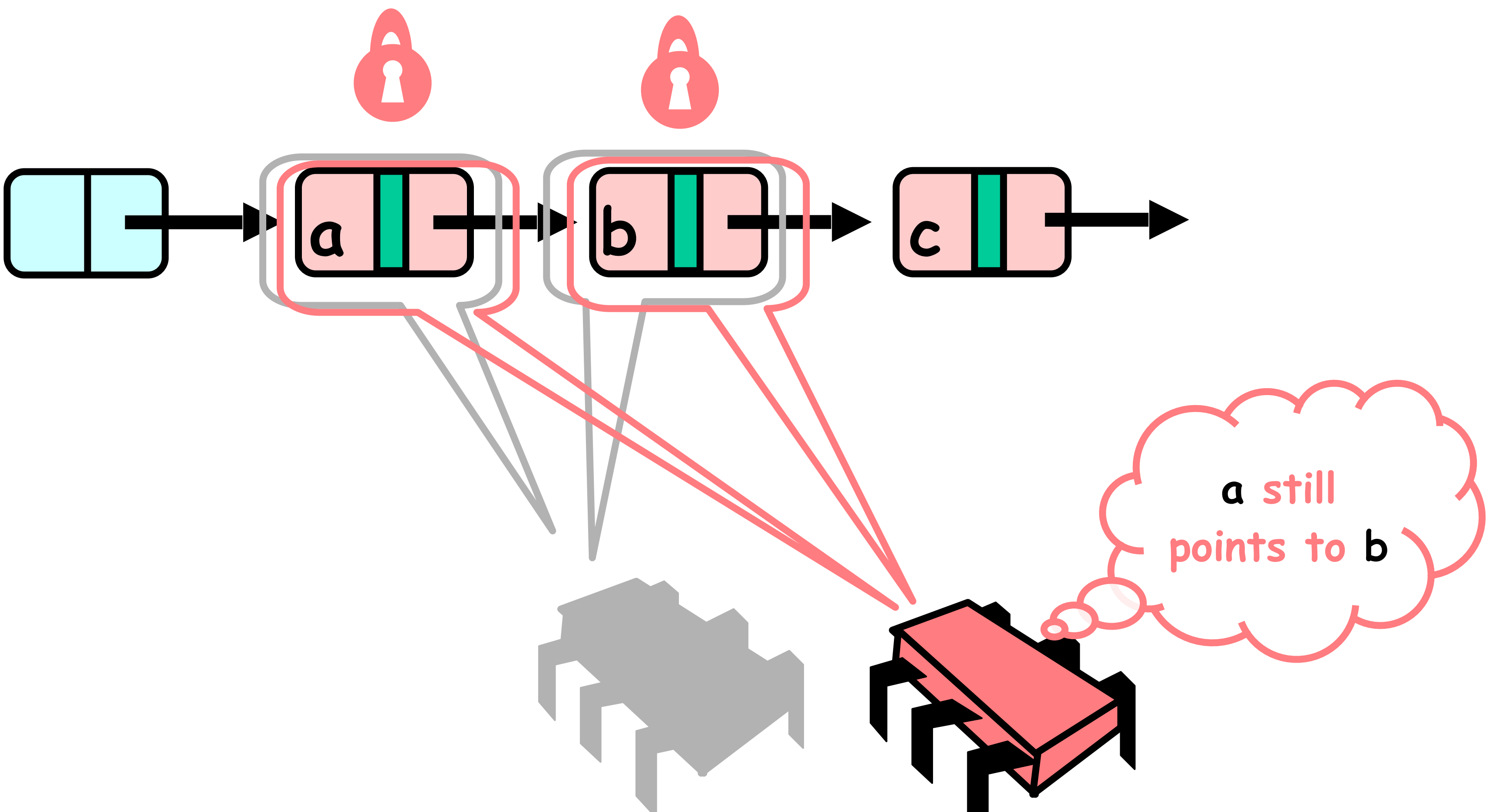
# Business as Usual



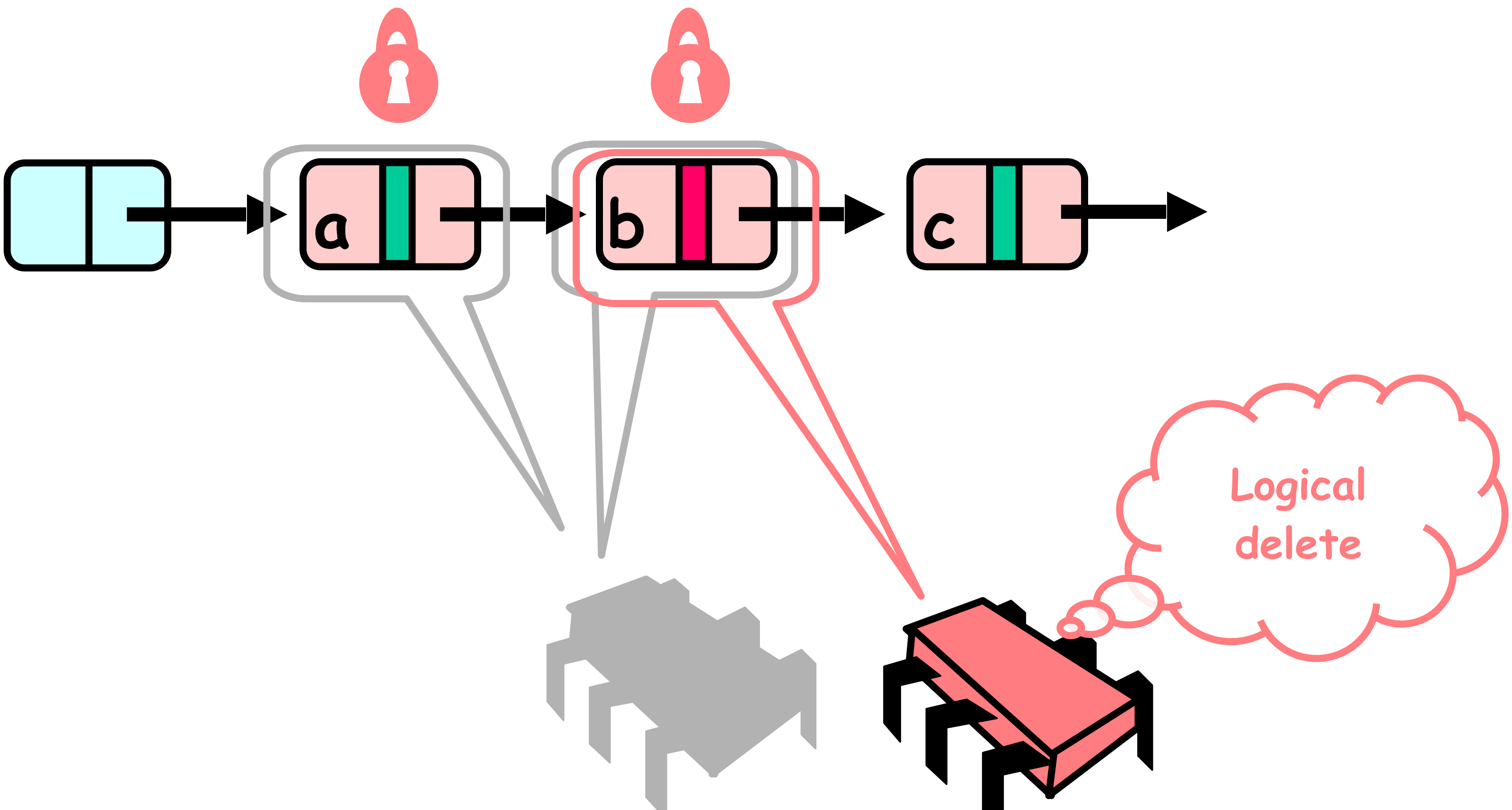
# Business as Usual



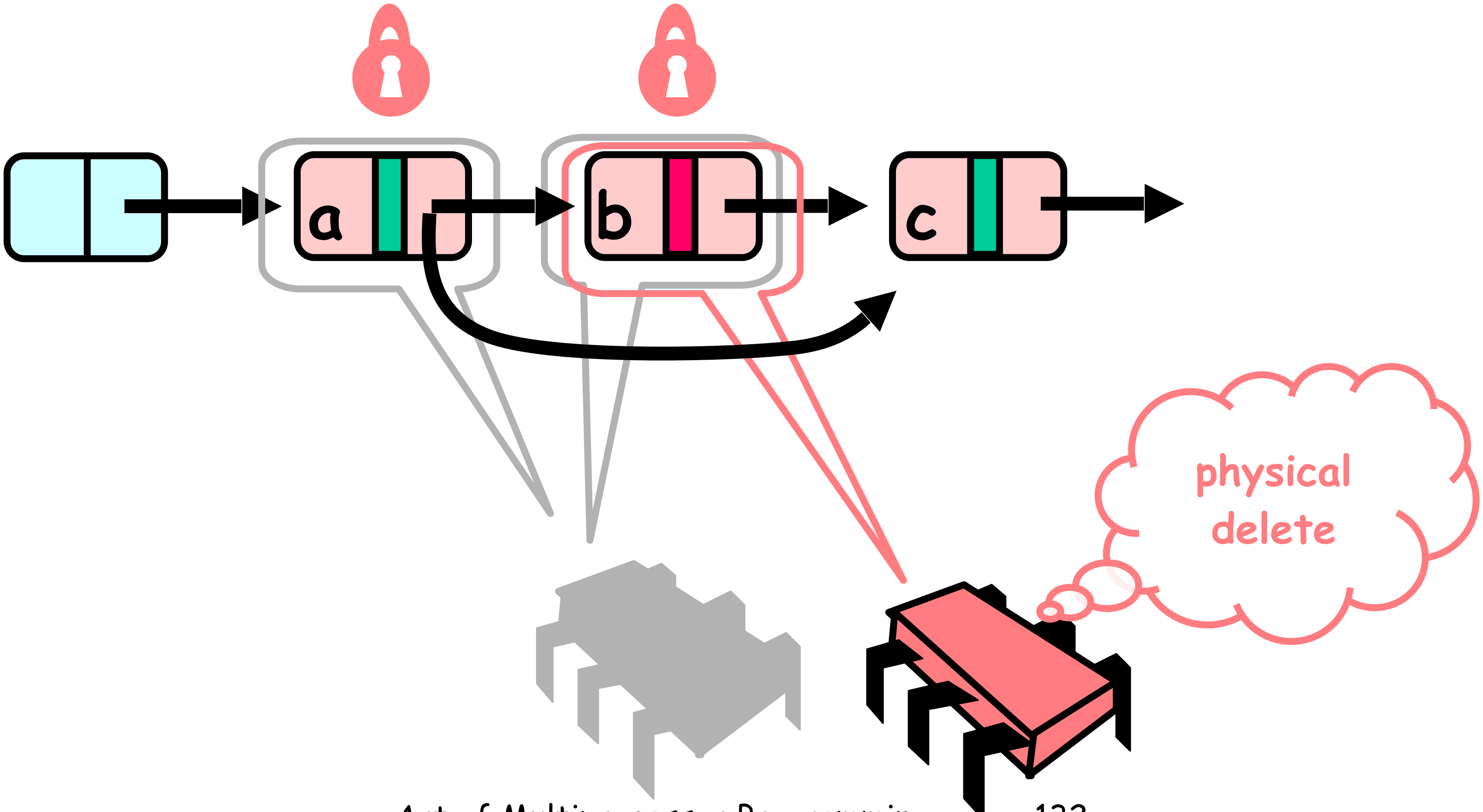
# Business as Usual



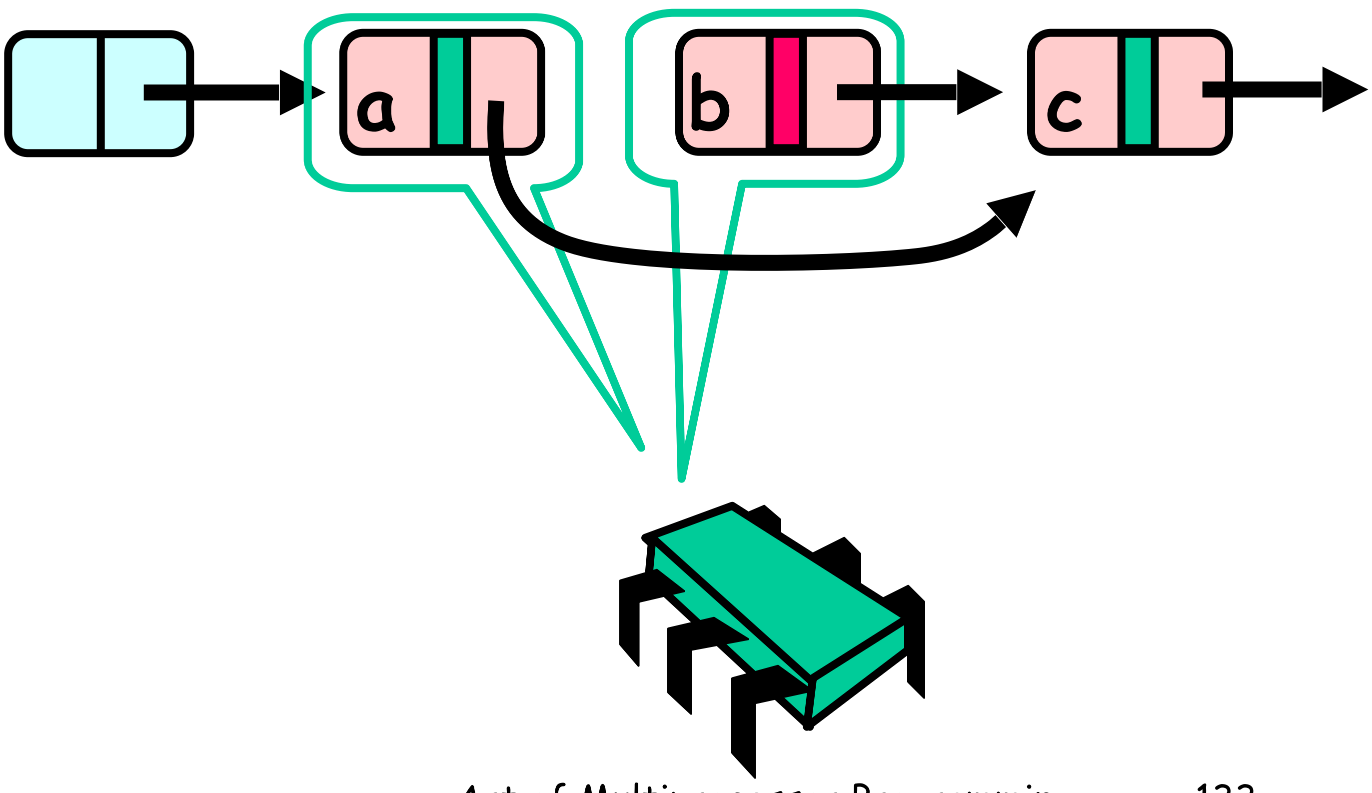
# Business as Usual



# Business as Usual



# Business as Usual



# Invariant

- If not marked then item in the set
- and reachable from head
- and if not yet traversed it is reachable from pred

# Validation

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```

# List Validate Method

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```

**Predecessor not Logically removed**

# List Validate Method

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```

Current not
Logically removed

# List Validate Method

```
private boolean
  validate(Node pred, Node curr) {
 return
  !pred.marked &&
  !curr.marked &&
  pred.next == curr);
  }
```

**Predecessor still Points to current**

# Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

# Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

**Start at the head**

# Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

**Search key range**

# Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

**Traverse without locking (nodes may have been removed)**

# Contains

```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

**Present and undeleted?**

# Lazy Synchronization

```
public boolean add(T item) {
   int key = item.hashCode();
   while (true) {
     Node pred = this.head;
     Node curr = head.next;
     while (curr.key < key) {
       pred = curr; curr = curr.next;
     }
     pred.lock();
     try {
       curr.lock();
       try {
         if (validate(pred, curr)) {
           if (curr.key == key) {
             return false;
           } else {
             Node Node = new Node(item);
             Node.next = curr;
             pred.next = Node;
             return true;
           }
         }
       } finally { // always unlock
         curr.unlock();
       }
     } finally { // always unlock
       pred.unlock();
     }
   }
 }
```

```java
public boolean remove(T item) {
   int key = item.hashCode();
   while (true) {
     Node pred = this.head;
     Node curr = head.next;
     while (curr.key < key) {
       pred = curr; curr = curr.next;
     }
     pred.lock();
     try {
       curr.lock();
       try {
         if (validate(pred, curr)) {
           if (curr.key != key) {
             return false;
           } else {
             curr.marked = true;
             pred.next = curr.next;
             return true;
           }
         }
       } finally {
         curr.unlock();
       }
     } finally {
       pred.unlock();
     }
   }
 }
```

```java
public boolean contains(T item) {
   int key = item.hashCode();
   Node curr = this.head;
   while (curr.key < key)
     curr = curr.next;
   return curr.key == key && !curr.marked;
 }
```

```java
 private boolean validate(Node pred, Node
   curr) {
   return  !pred.marked && !curr.marked &&
  pred.next == curr;
 }
```

# Evaluation

- Good:
  - contains() doesn't lock
  - Good because typically high % contains()
  - Uncontended calls don't re-traverse
- Bad
  - Contended add() and remove() calls do re-traverse
  - Traffic jam if one thread delays

# 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….

# 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