

Locking Strategies: Optimistic, Lazy

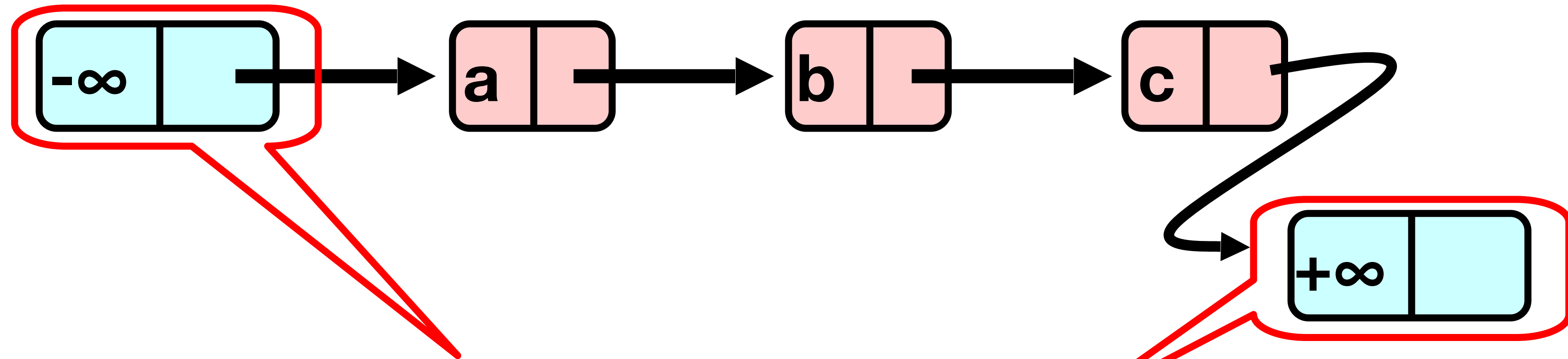
CS 475, Fall 2019

Concurrent & Distributed Systems

Sequential Consistency vs Linearizability

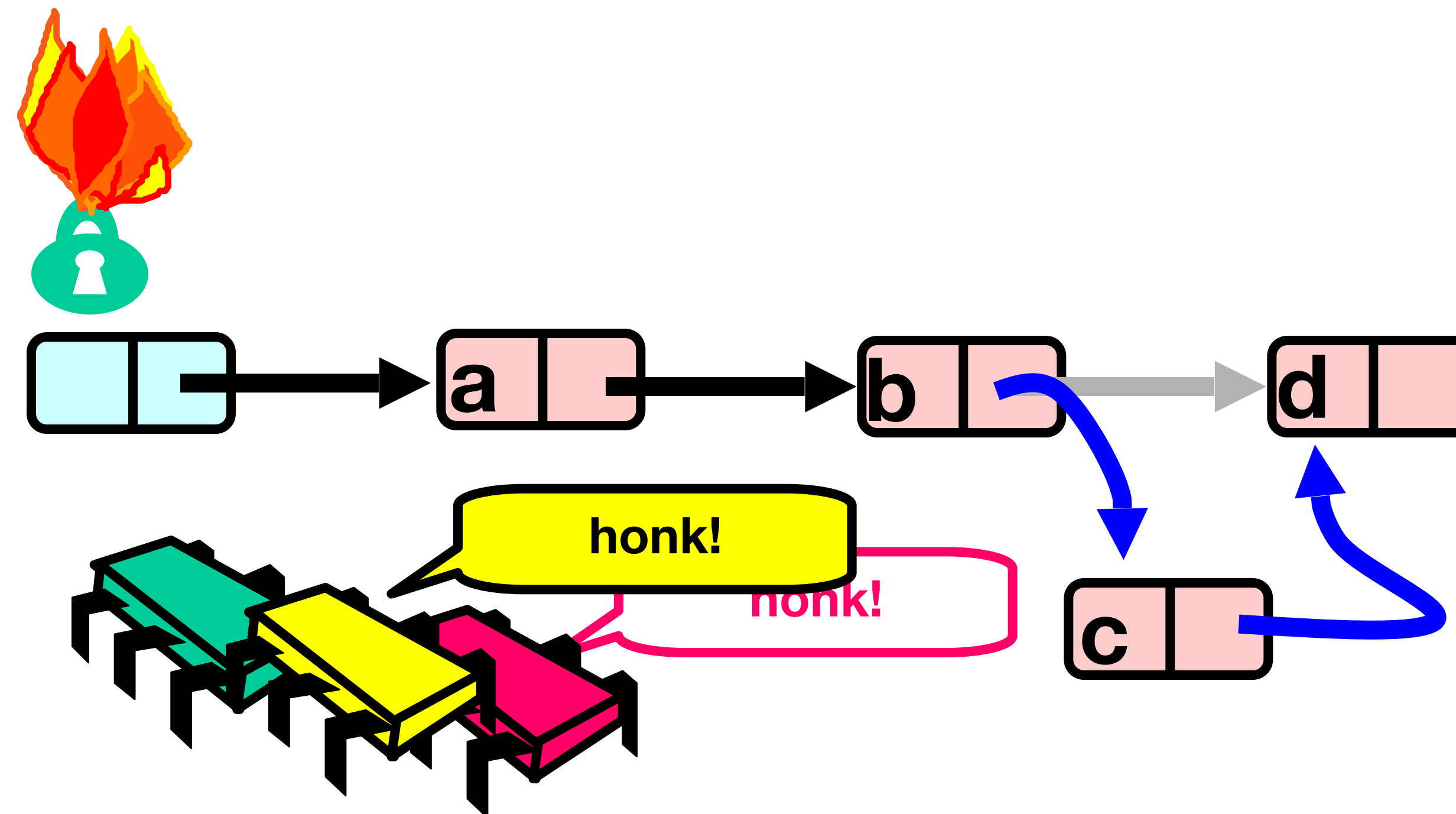
- Linearizability can be composed:
 - If p's execution and q's execution are both linearizable, then the combination must also be linearizable
- Sequential consistency can not be composed:
 - If p's execution and q's execution are both sequential, then the combination MAY also be sequential (but not guaranteed!)
- Why use sequential consistency?
 - Does not require global clock

The List-Based Set



Sorted with Sentinel nodes
(min & max possible keys)

Course Grained Locking



Simple but **hotspot + bottleneck**

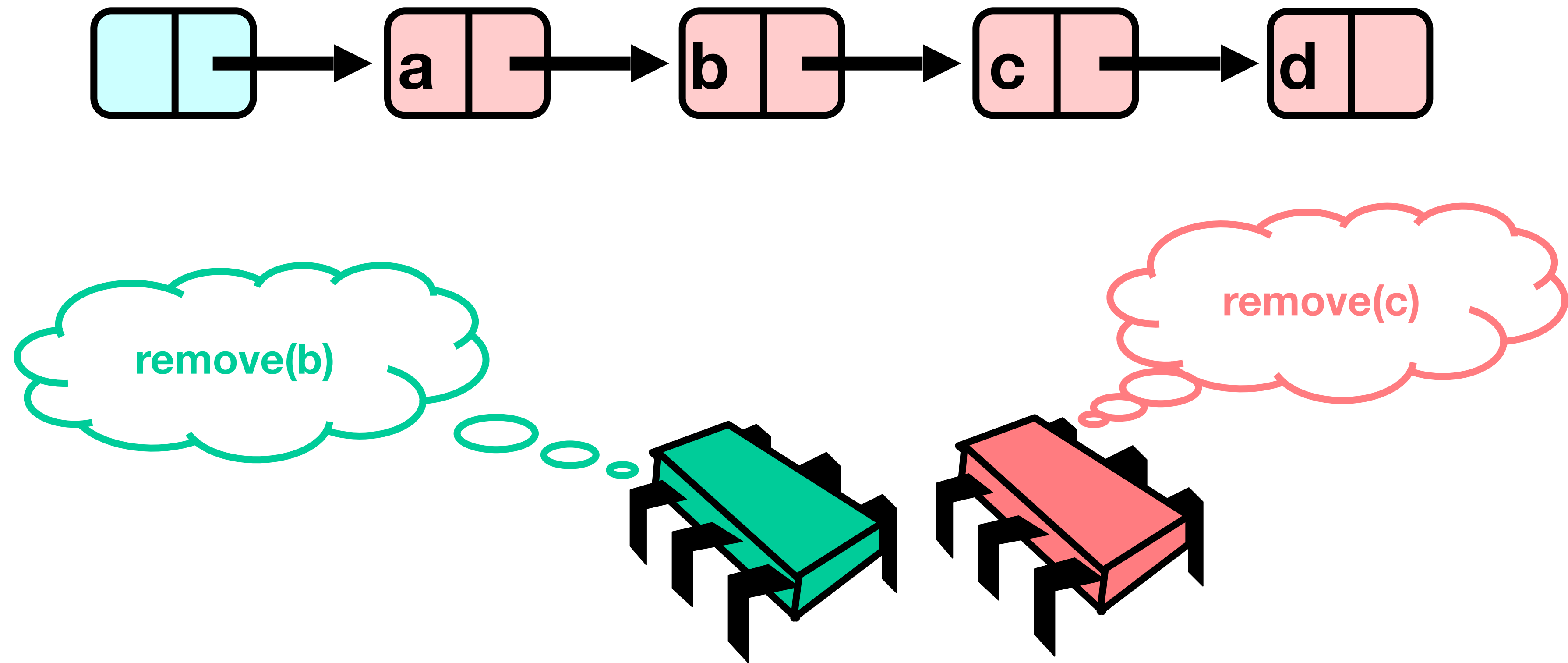
Coarse-Grained Locking

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

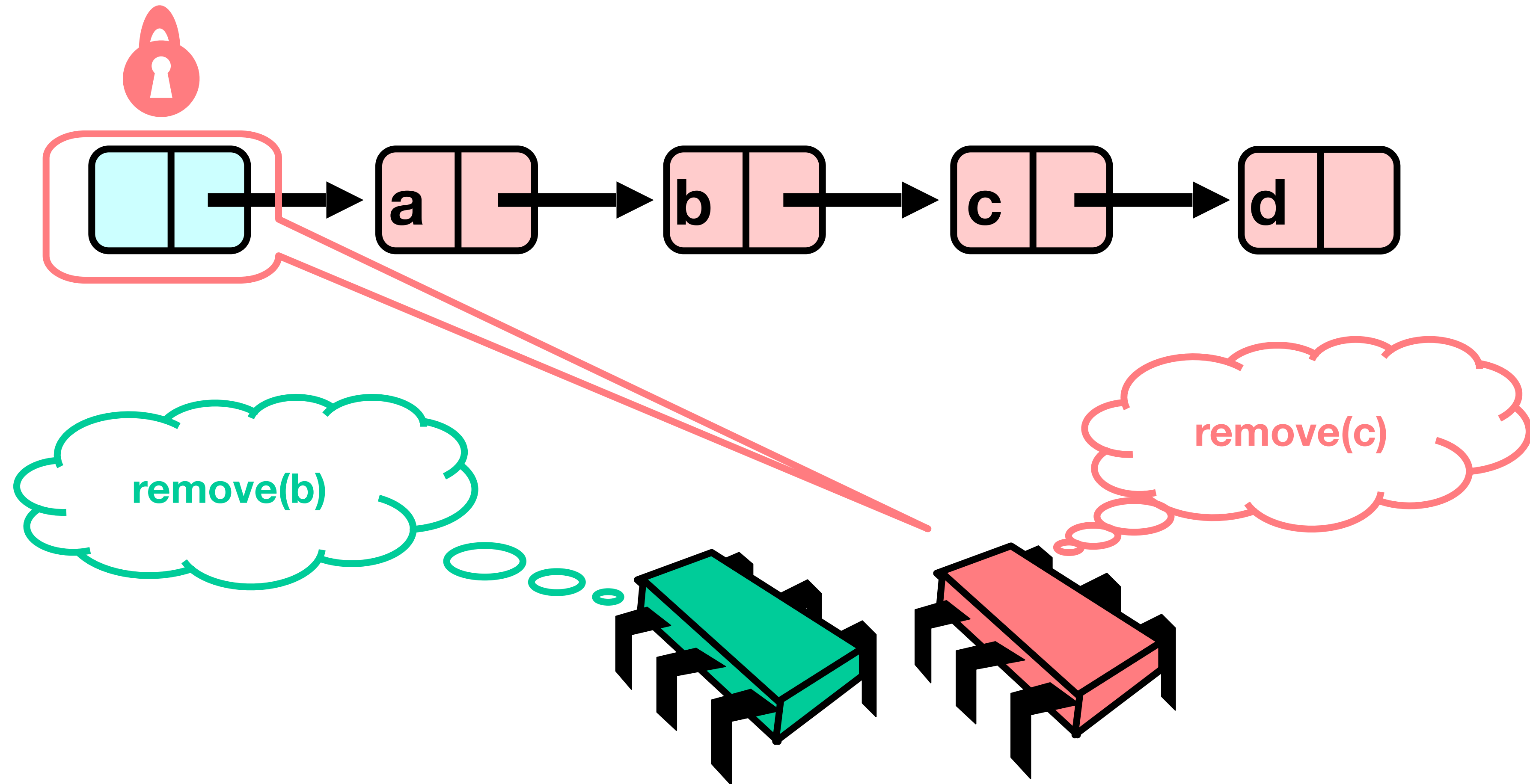
Fine-grained Locking

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

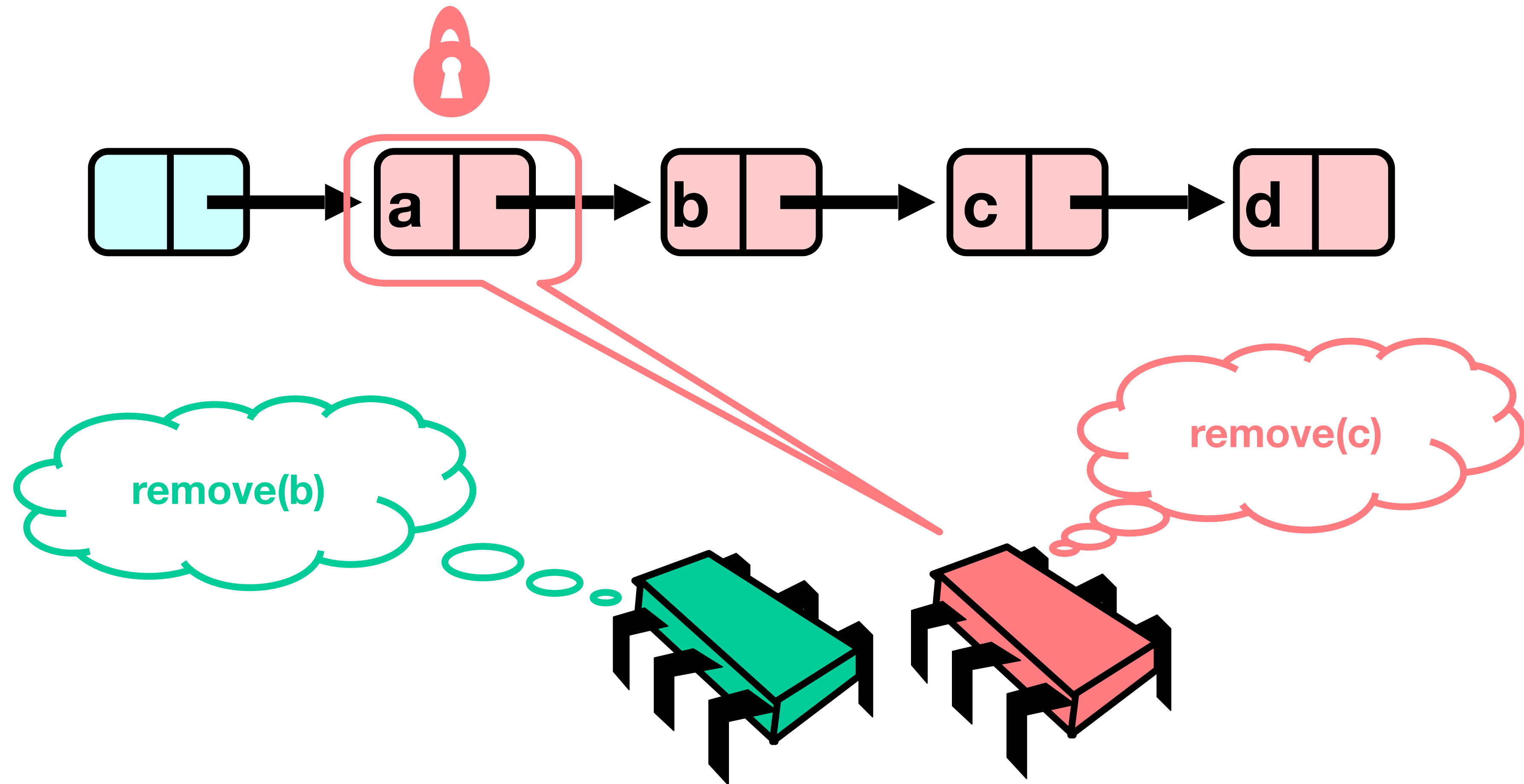
Simple Fine-Grained Locking: Remove



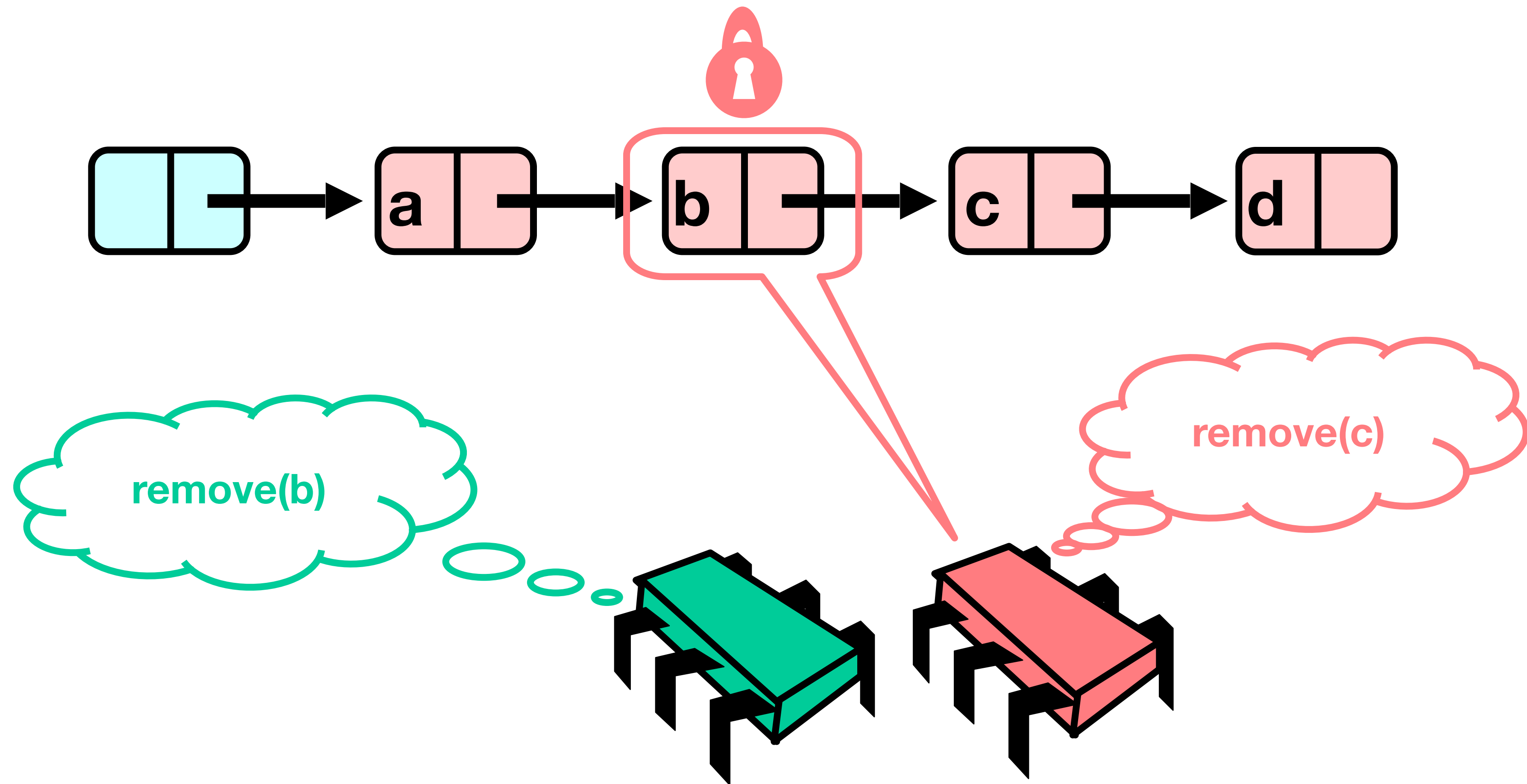
Simple Fine-Grained Locking: Remove



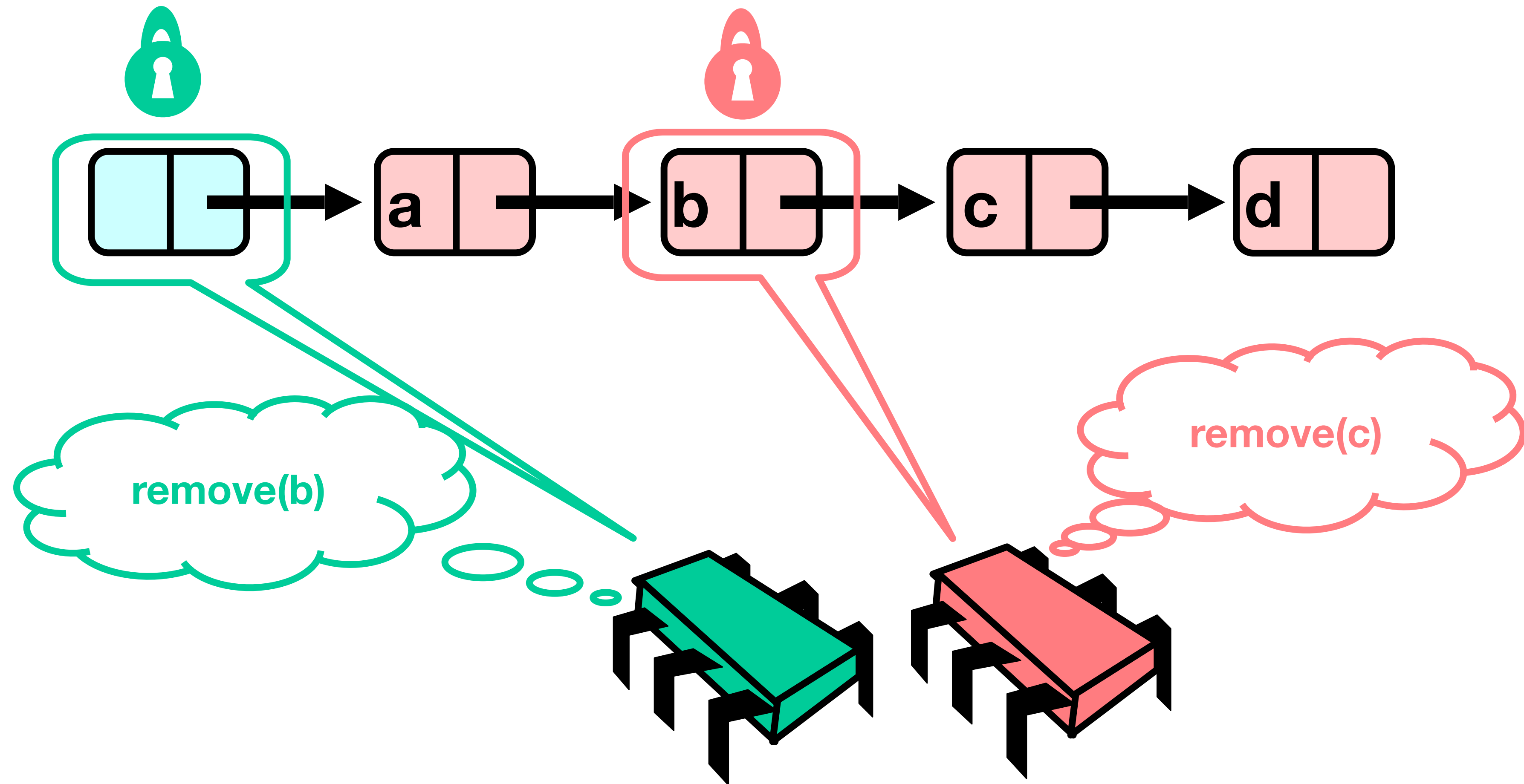
Simple Fine-Grained Locking: Remove



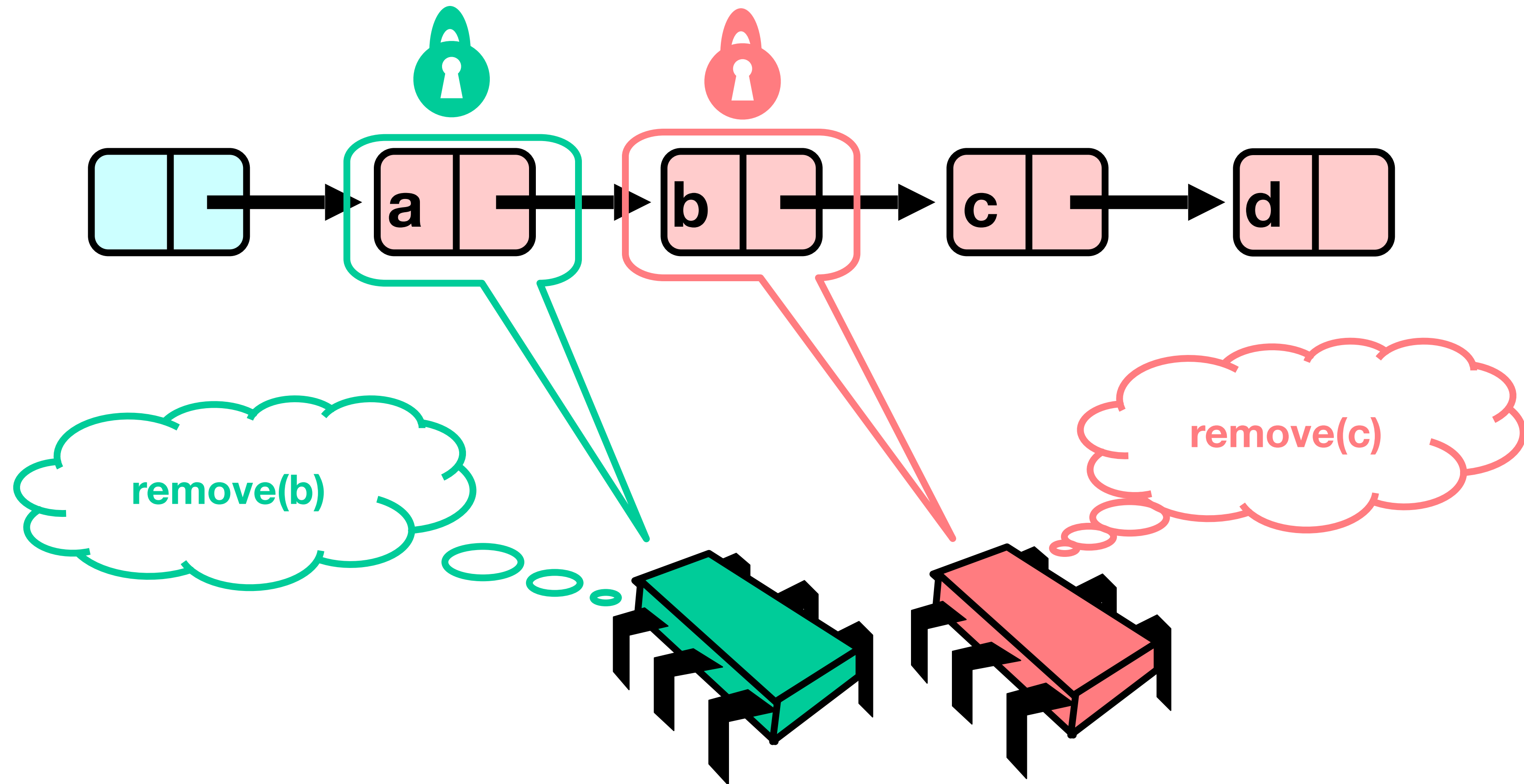
Simple Fine-Grained Locking: Remove



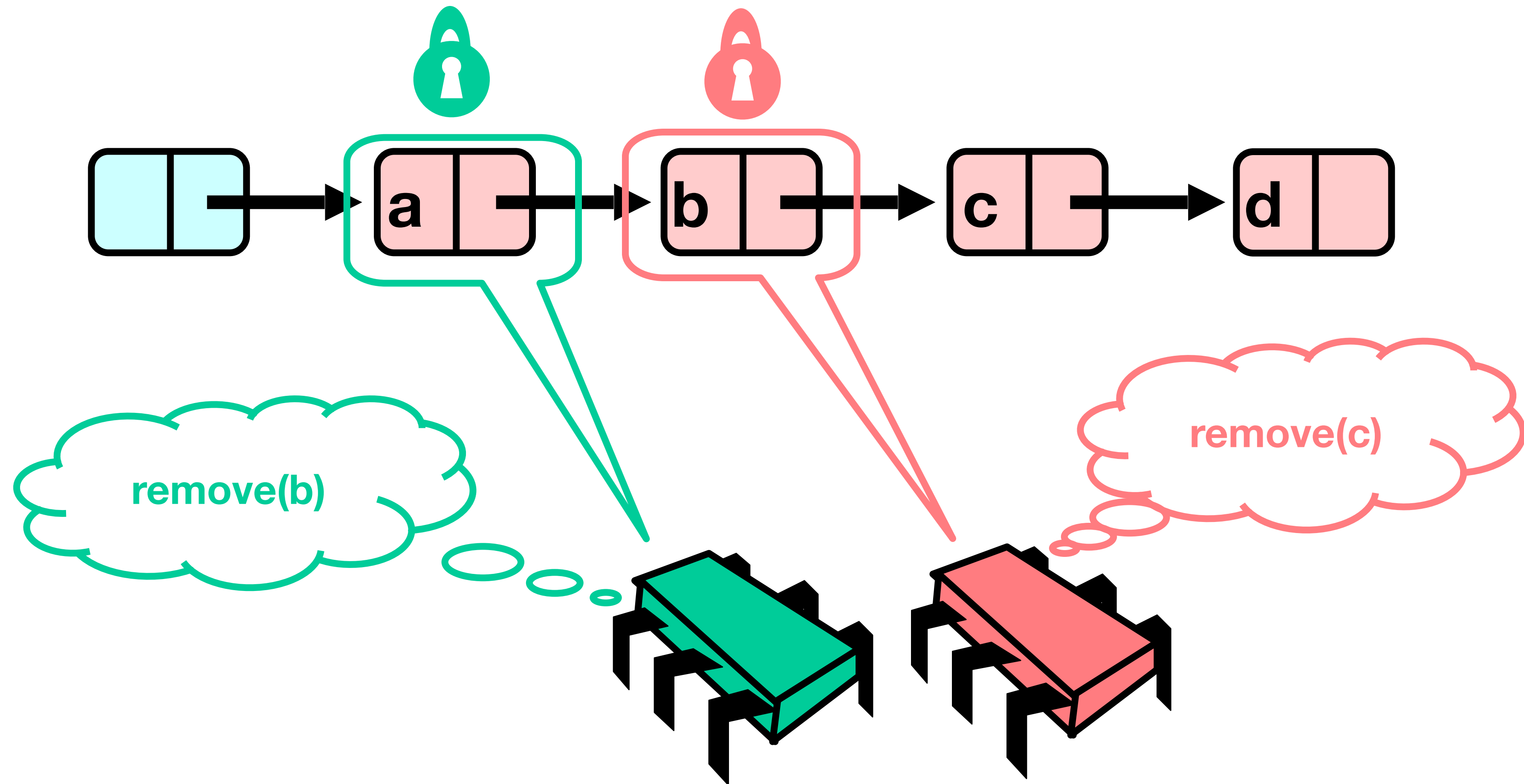
Simple Fine-Grained Locking: Remove



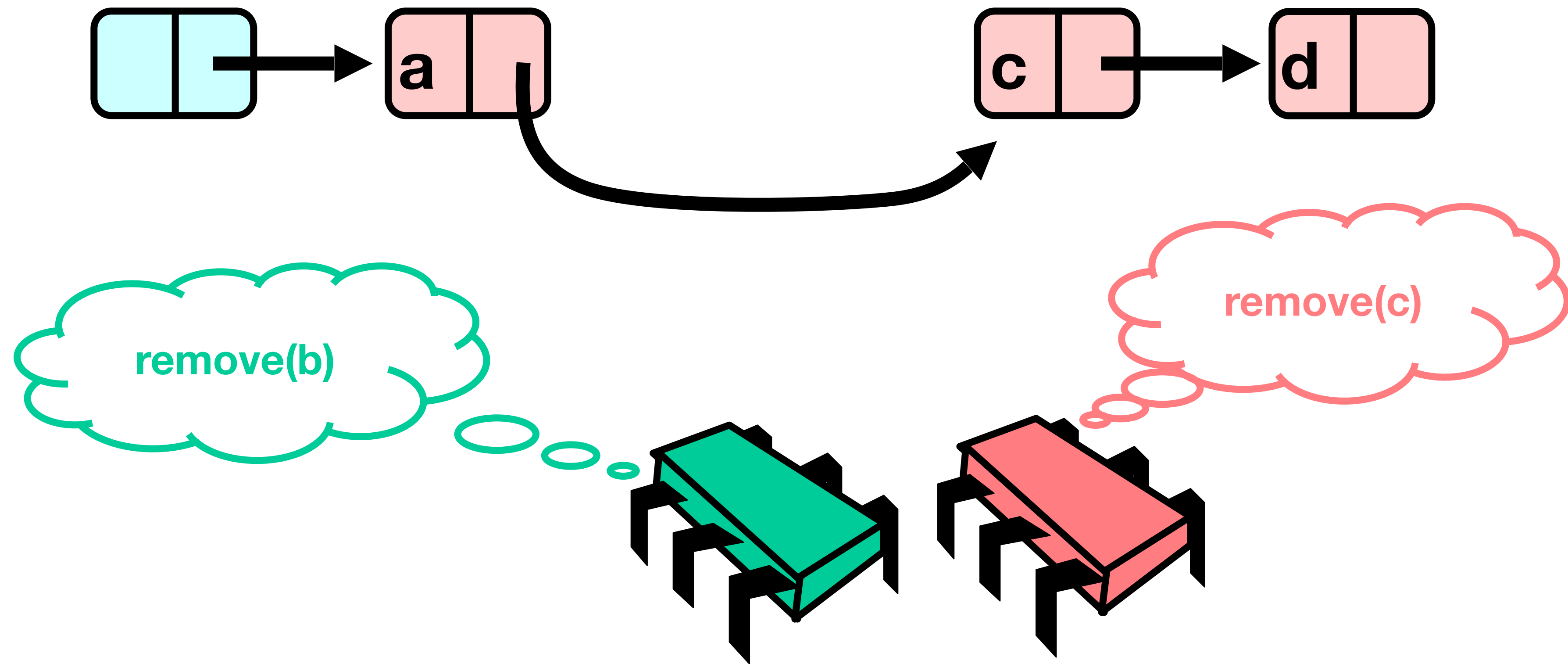
Simple Fine-Grained Locking: Remove



Simple Fine-Grained Locking: Remove

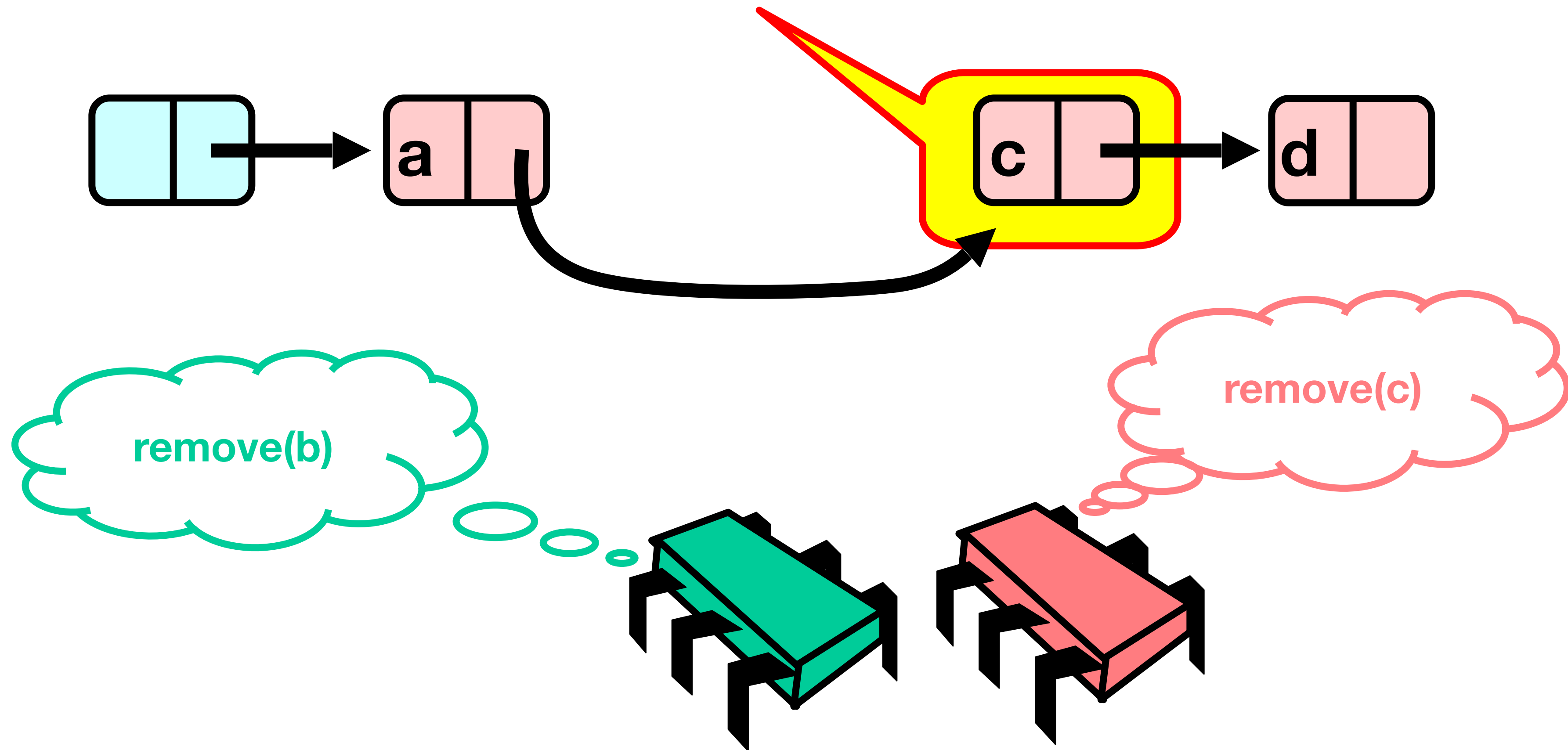


Uh, Oh

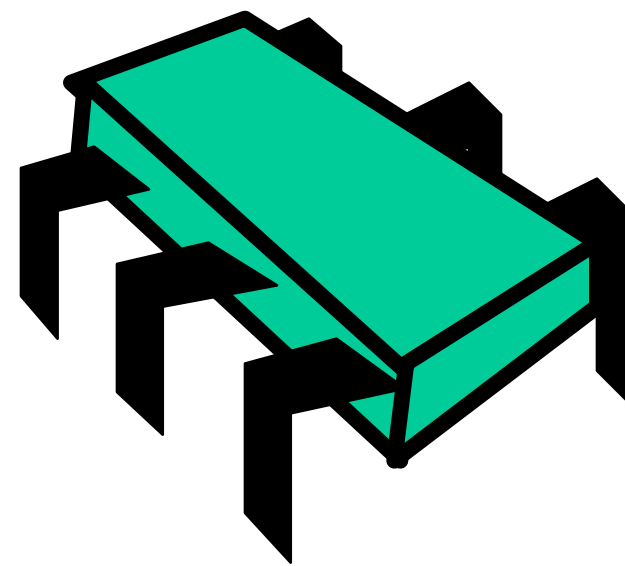
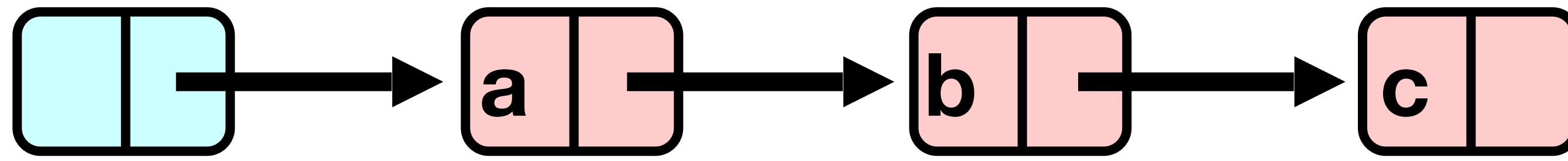


Uh, Oh

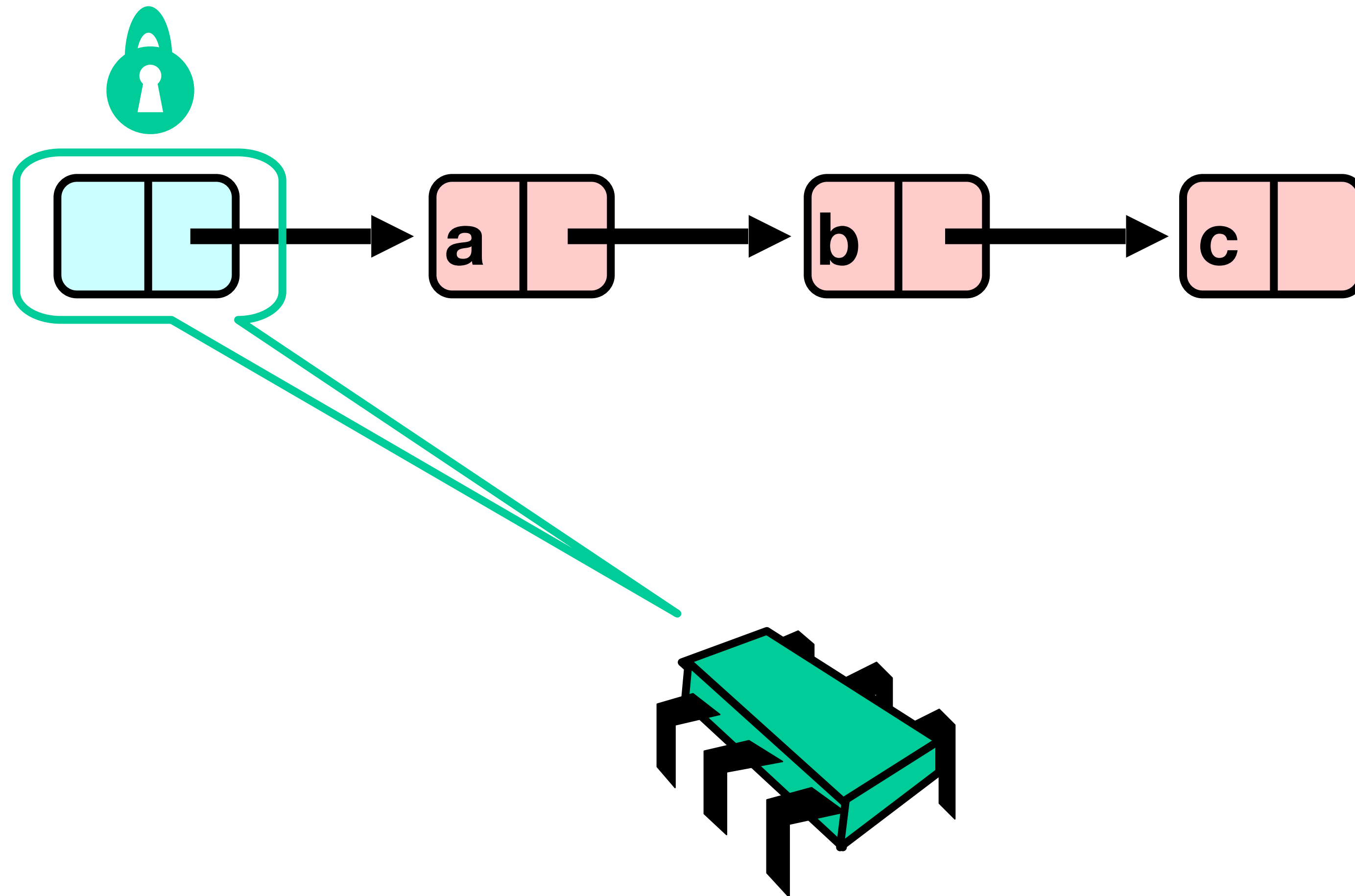
Bad news, C not removed



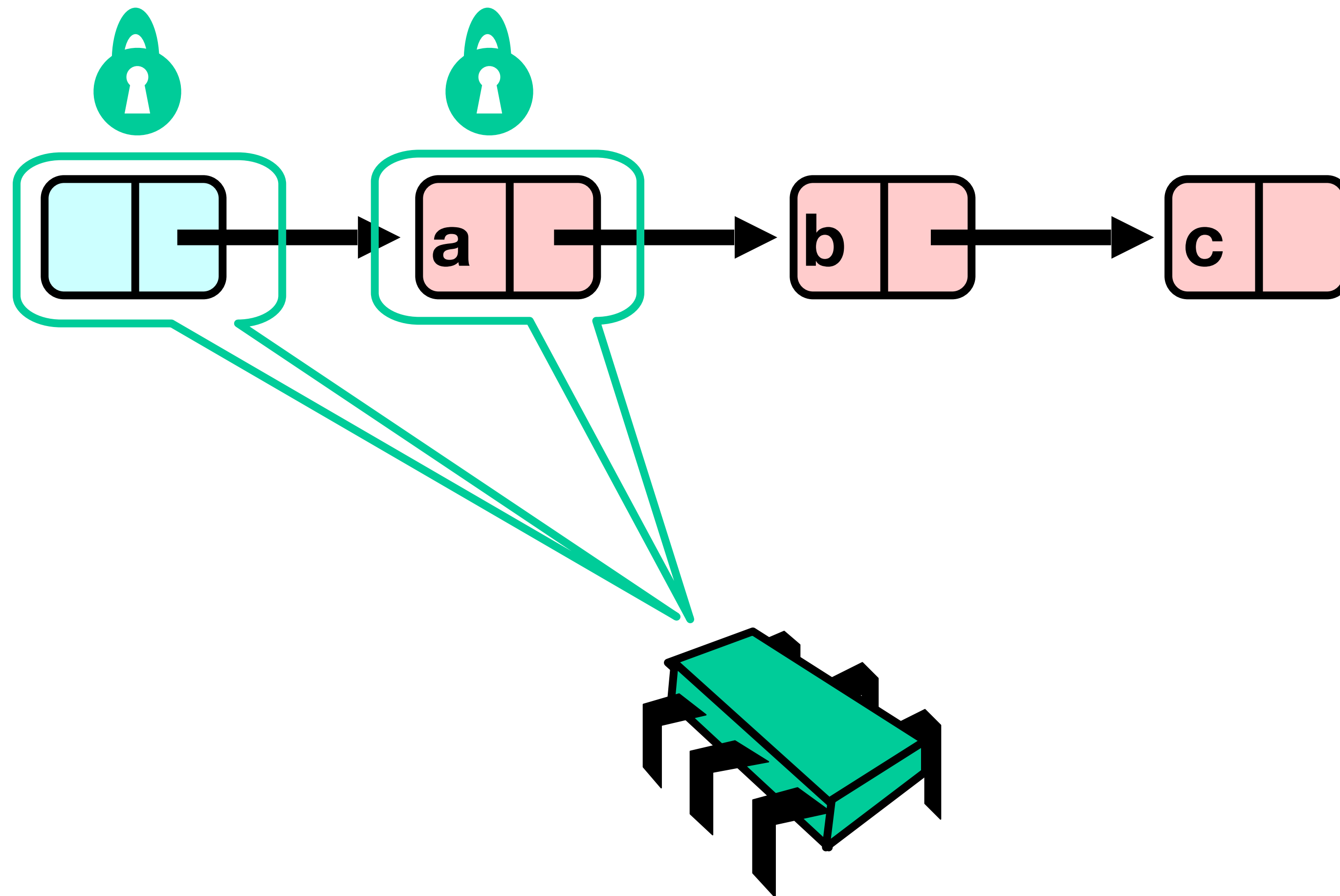
Hand-over-Hand locking



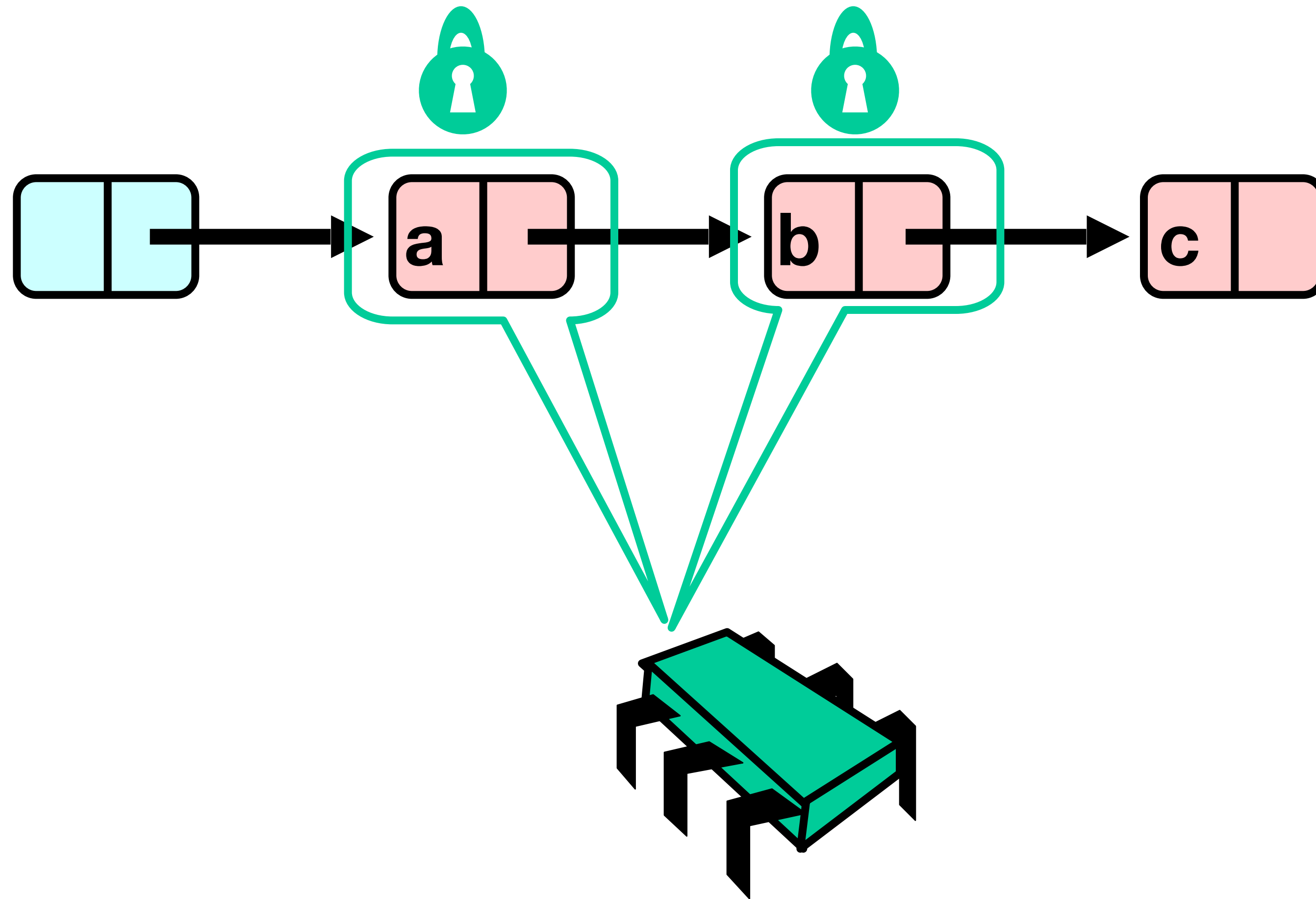
Hand-over-Hand locking



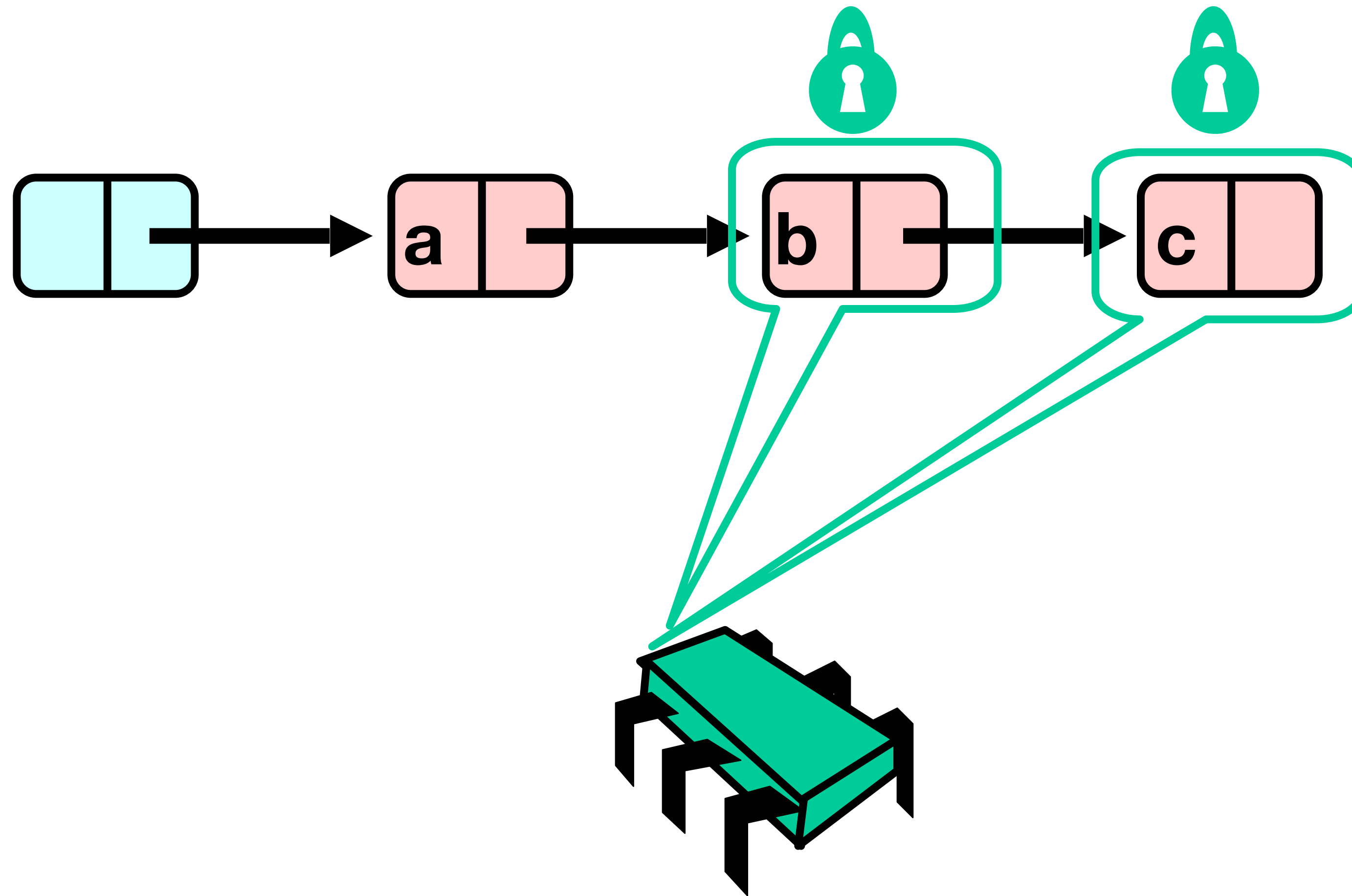
Hand-over-Hand locking



Hand-over-Hand locking

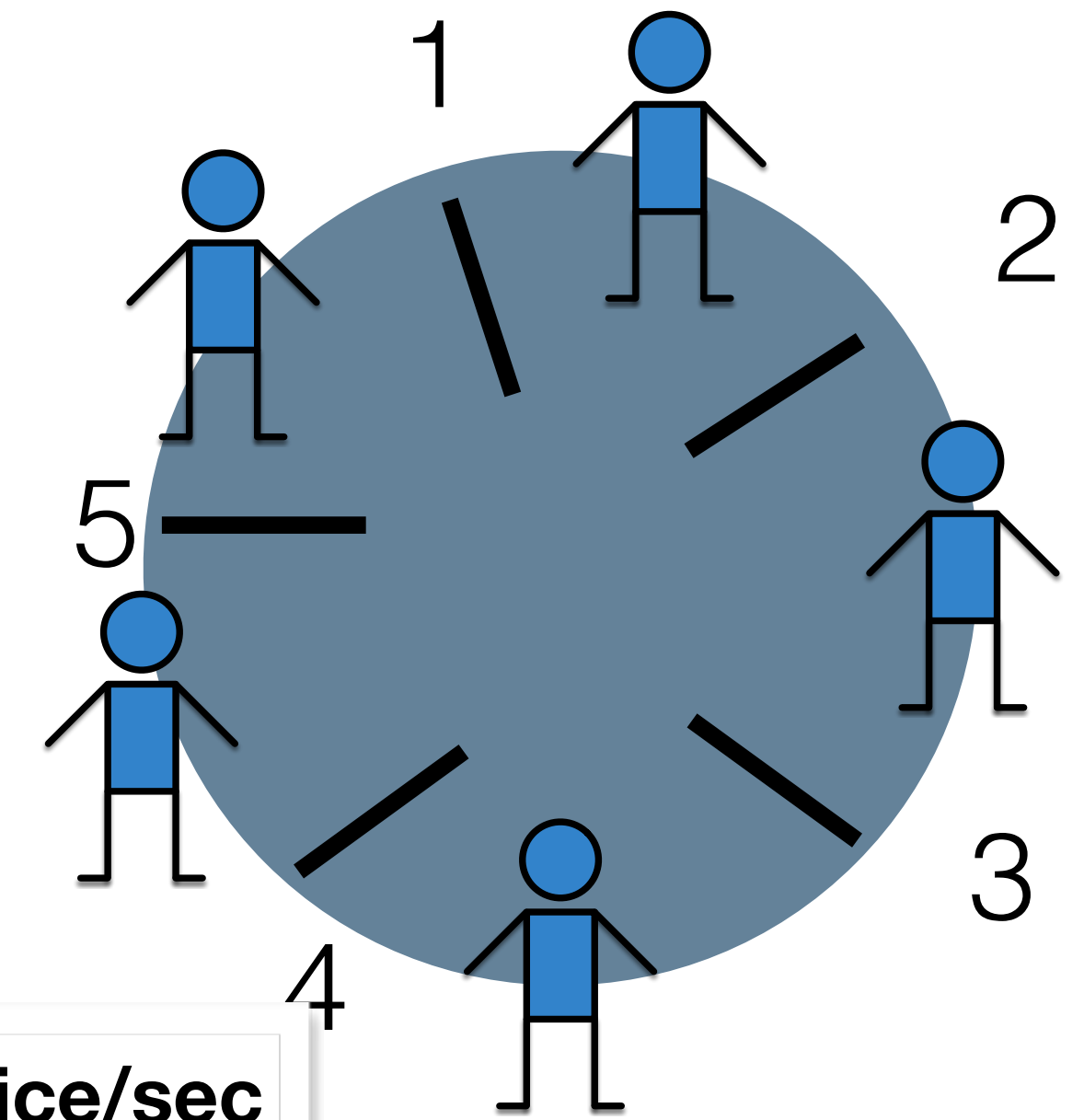


Hand-over-Hand locking



Dining Philosophers

- Give each chopstick a lock
- Is this enough?
- Could deadlock!
- Actual solutions:
 - Pick up one chopstick, wait for the other for N msec, otherwise put down what you have, wait, and try again
 - Only allow 4 philosophers to pick up chopsticks at once
 - Even # seats pick up right chopstick, odd # seats pick up left



1,450 grains of rice/sec

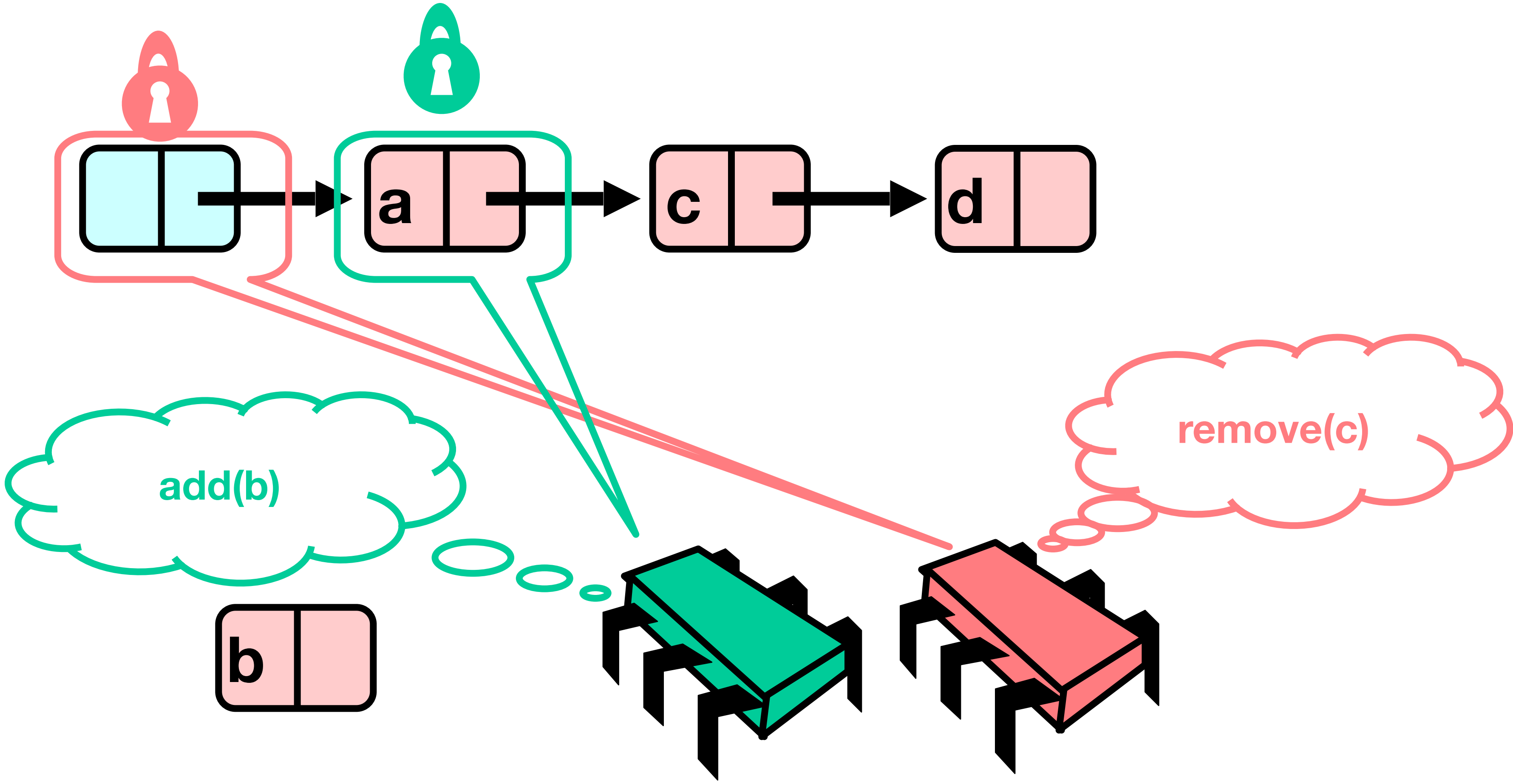
5,431,616 grains of rice/sec

12,450,856 grains of rice/sec

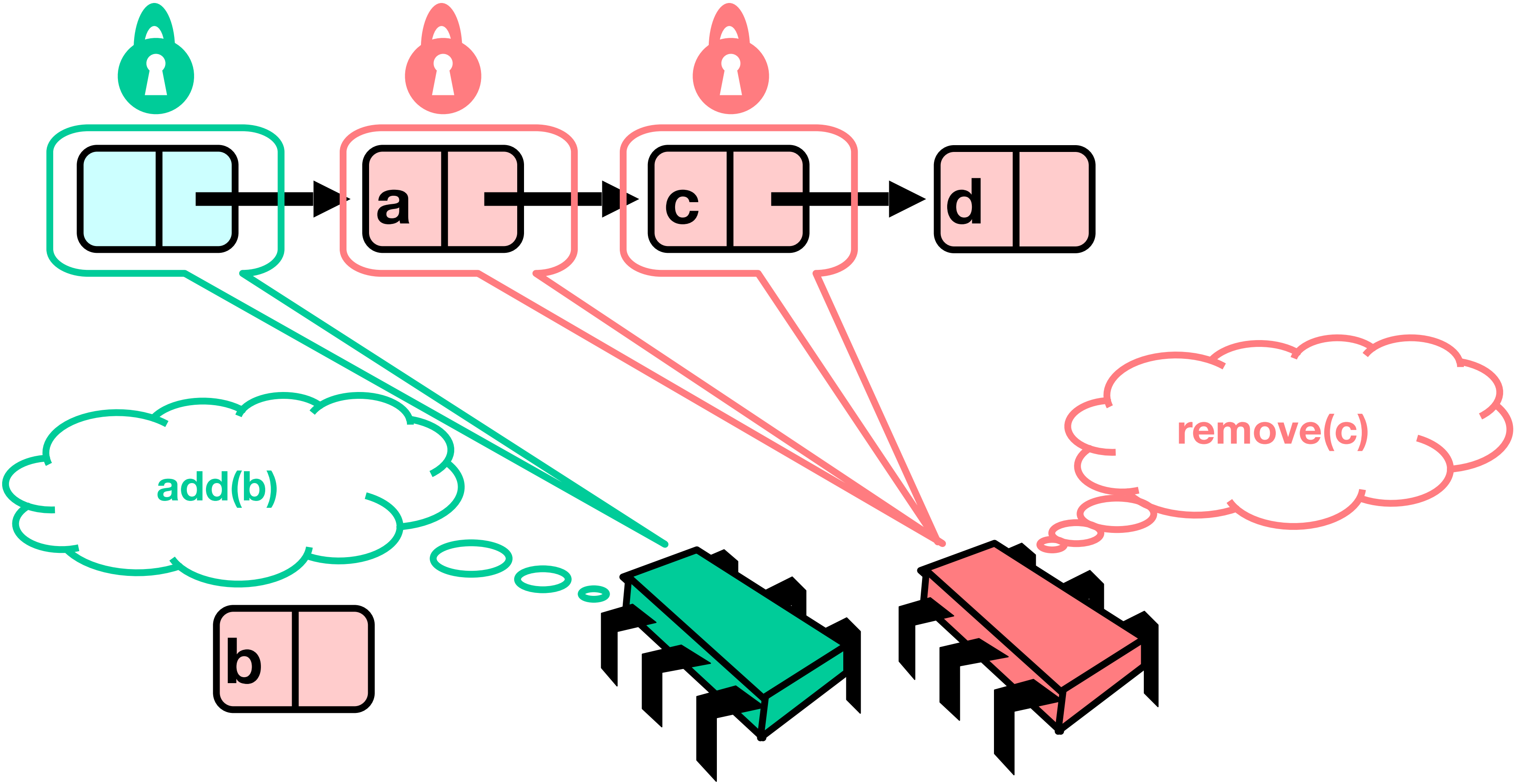
Adding Nodes

- To add node e
 - Neither successor nor predecessor can be deleted while add is running
 - Simply locking predecessor will guarantee that

Simple Fine-Grained Locking: Add (Execution Order 1)



Simple Fine-Grained Locking: Add (Execution Order 2)



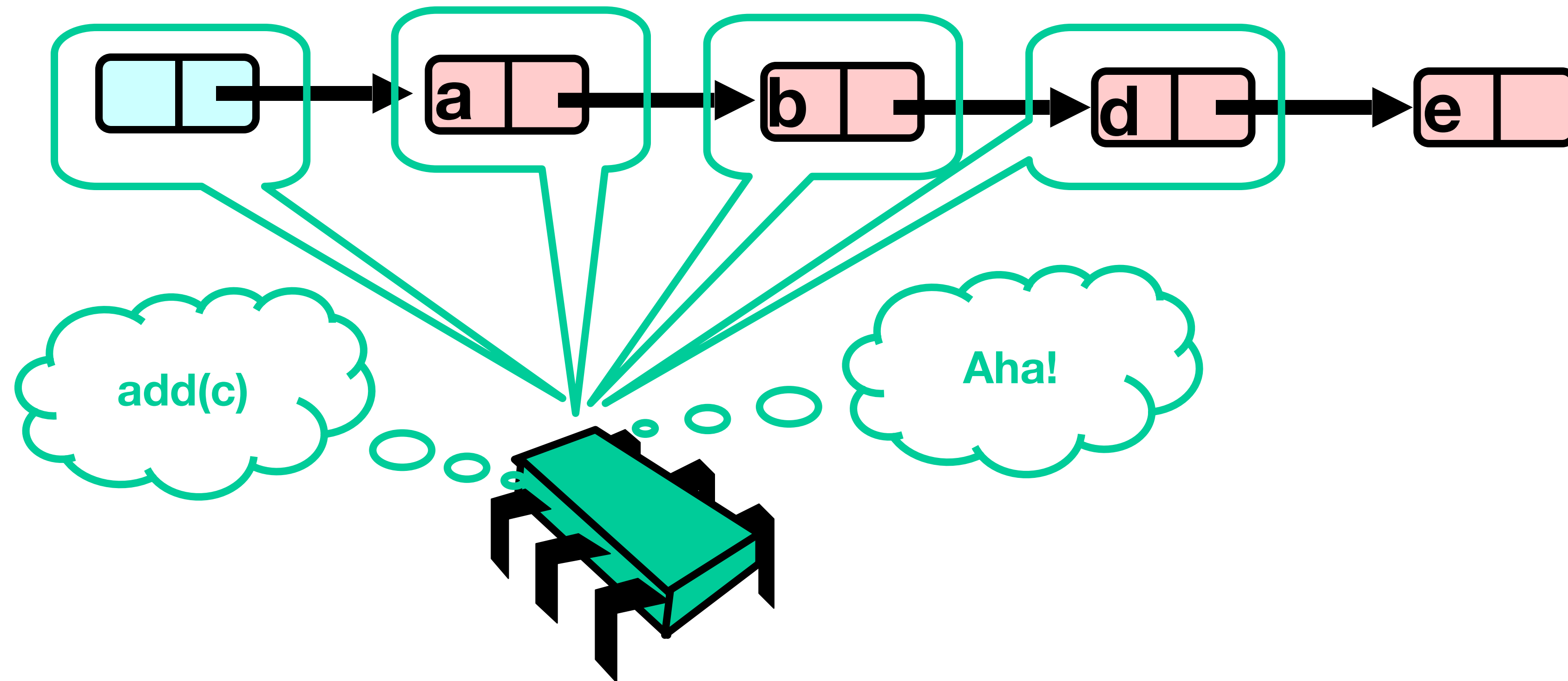
Today

- Adding threads should not lower throughput - should increase throughput
 - Not possible if inherently sequential
 - How do we structure locks for faster performance?
 - Case study on a data structure
- Reading: H&S 9.6-9.9
- Note: HW2 posted: <https://www.jonbell.net/gmu-cs-475-fall-2019/cs475-f19-homework-2/>

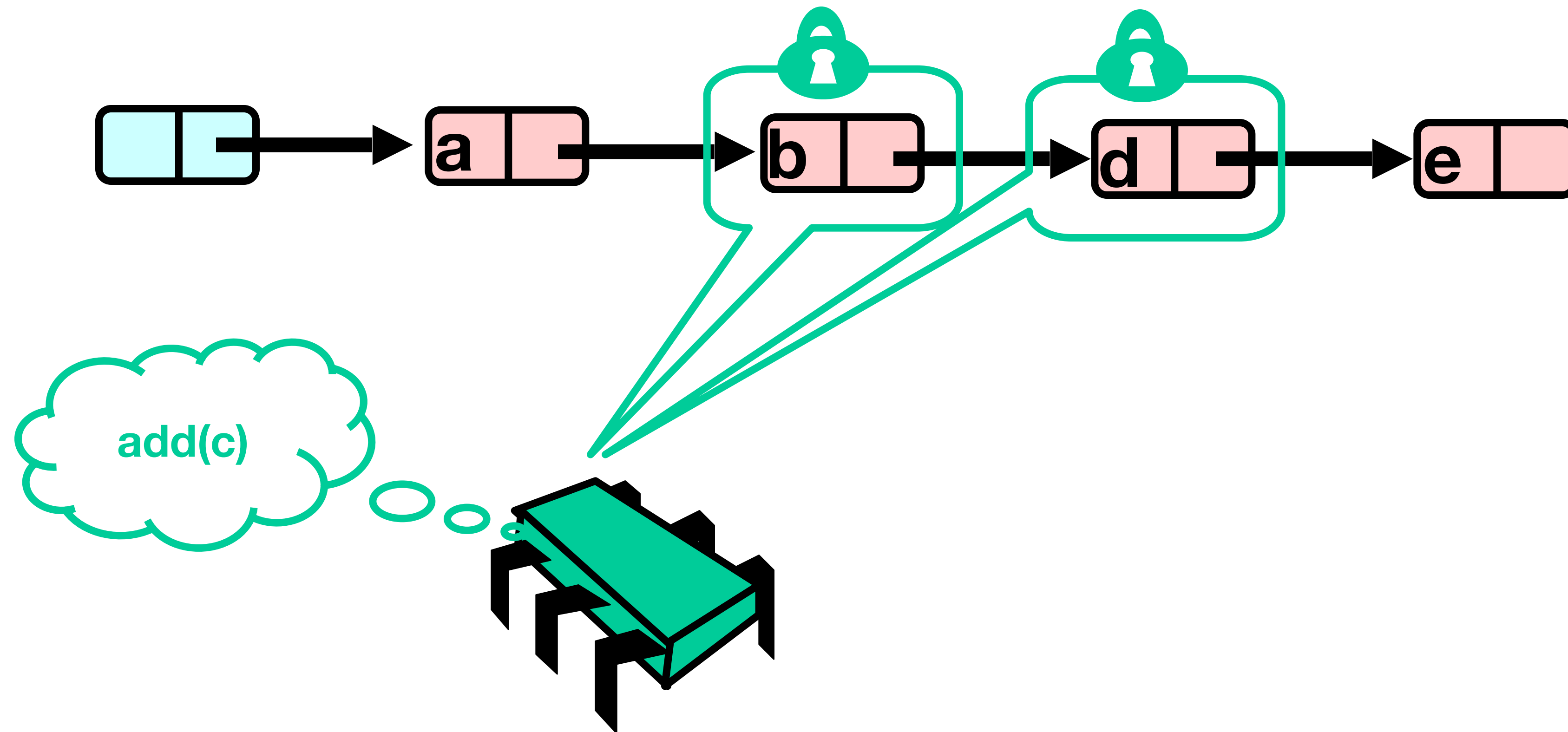
Optimistic Synchronization

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

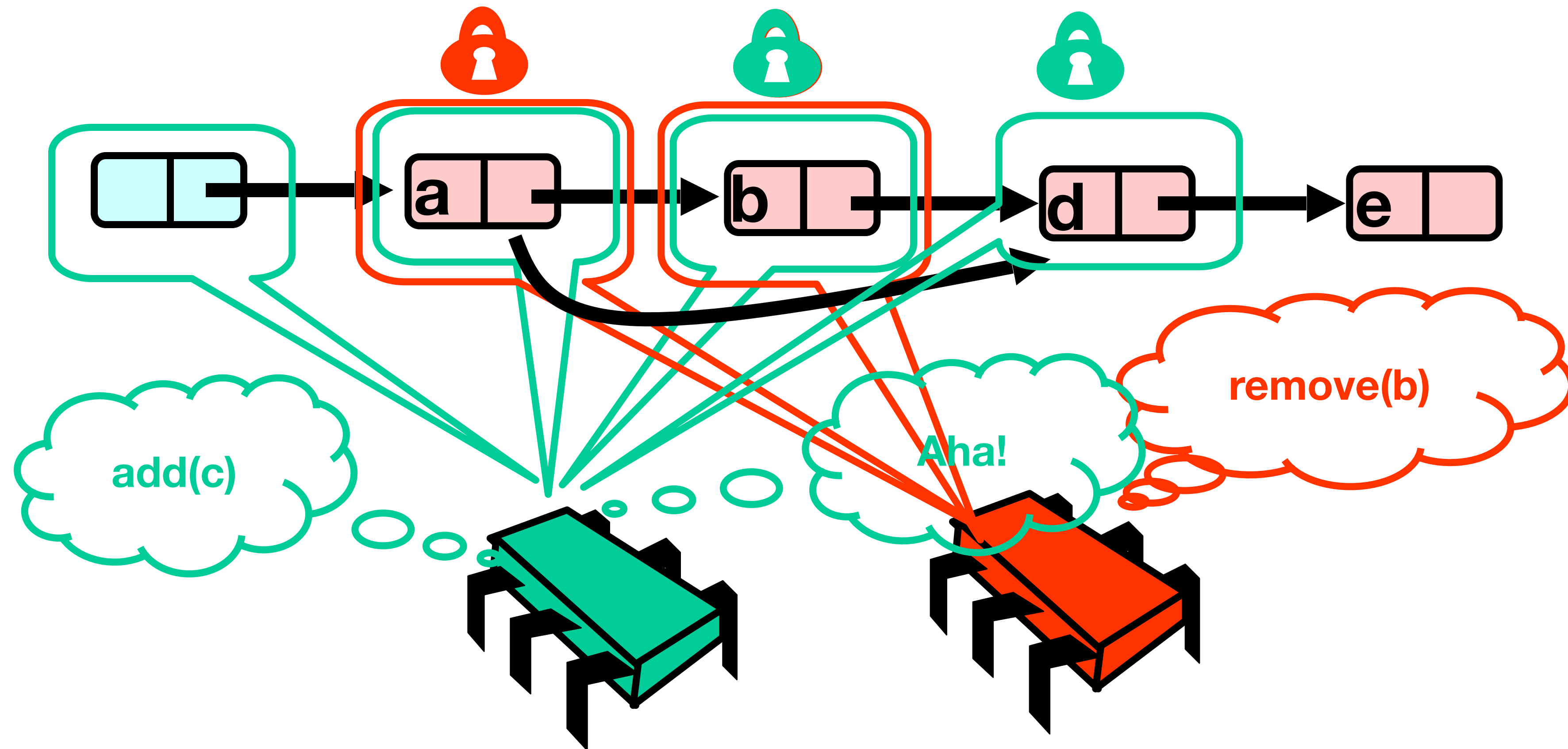
Optimistic: Traverse without Locking



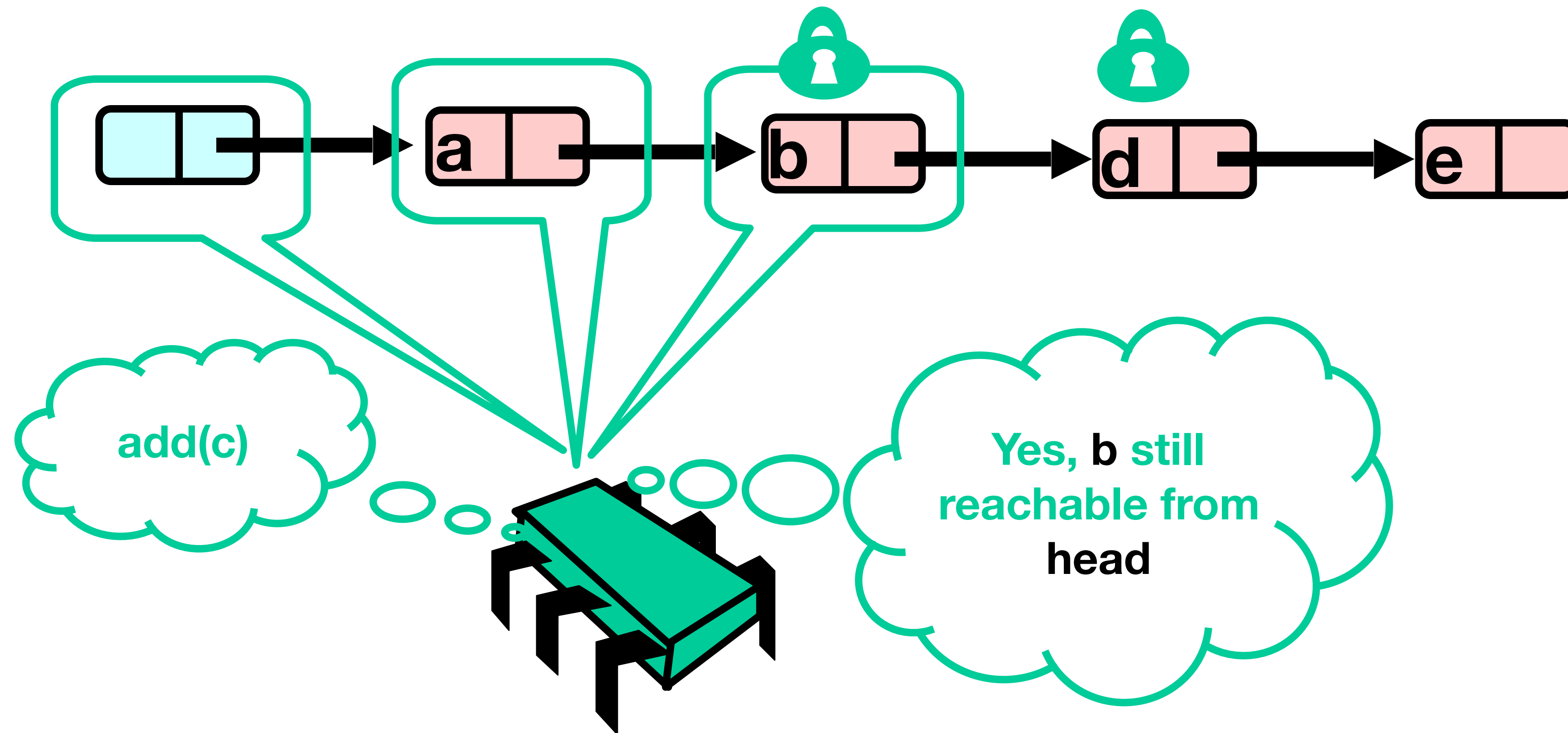
Optimistic: Lock and Load



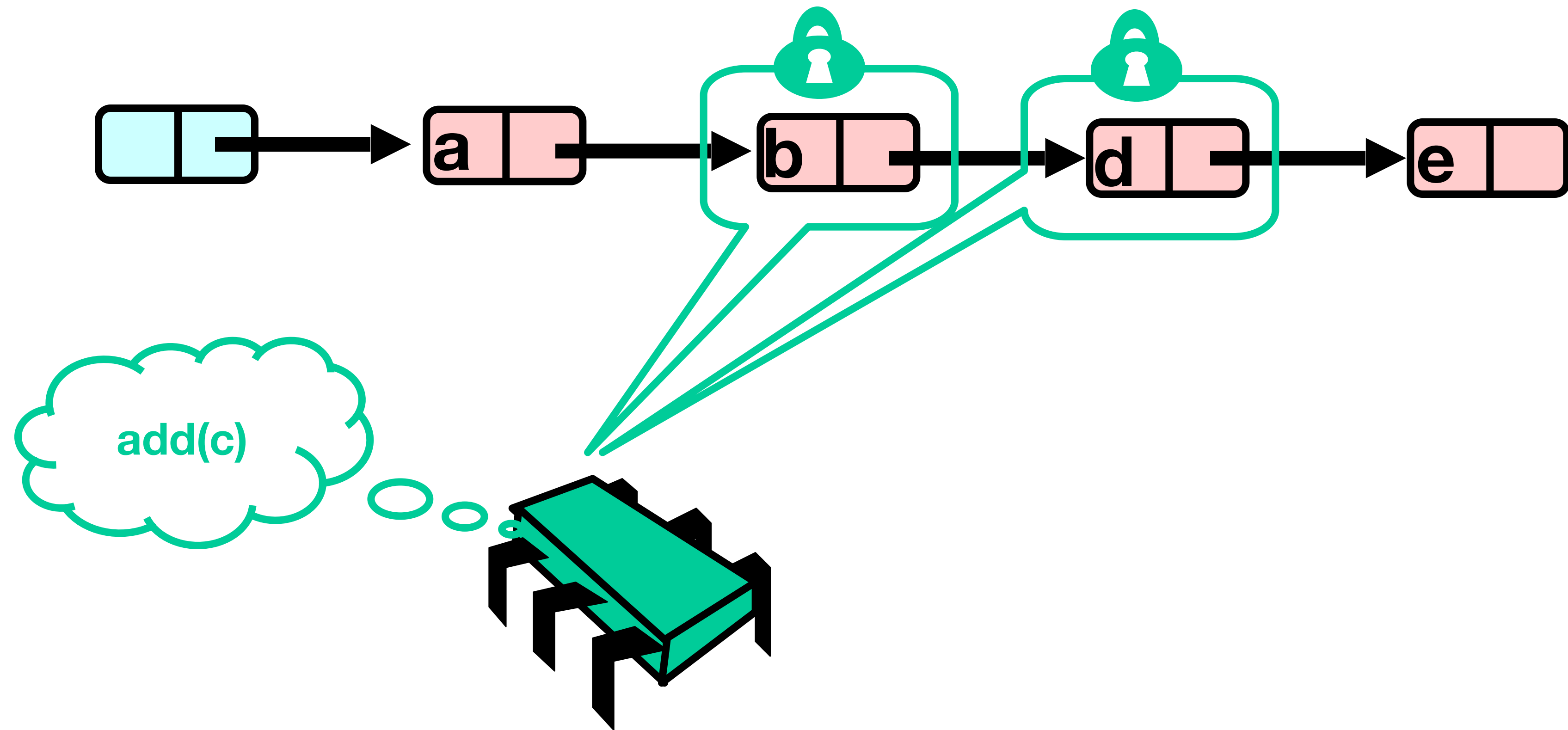
What could go wrong?



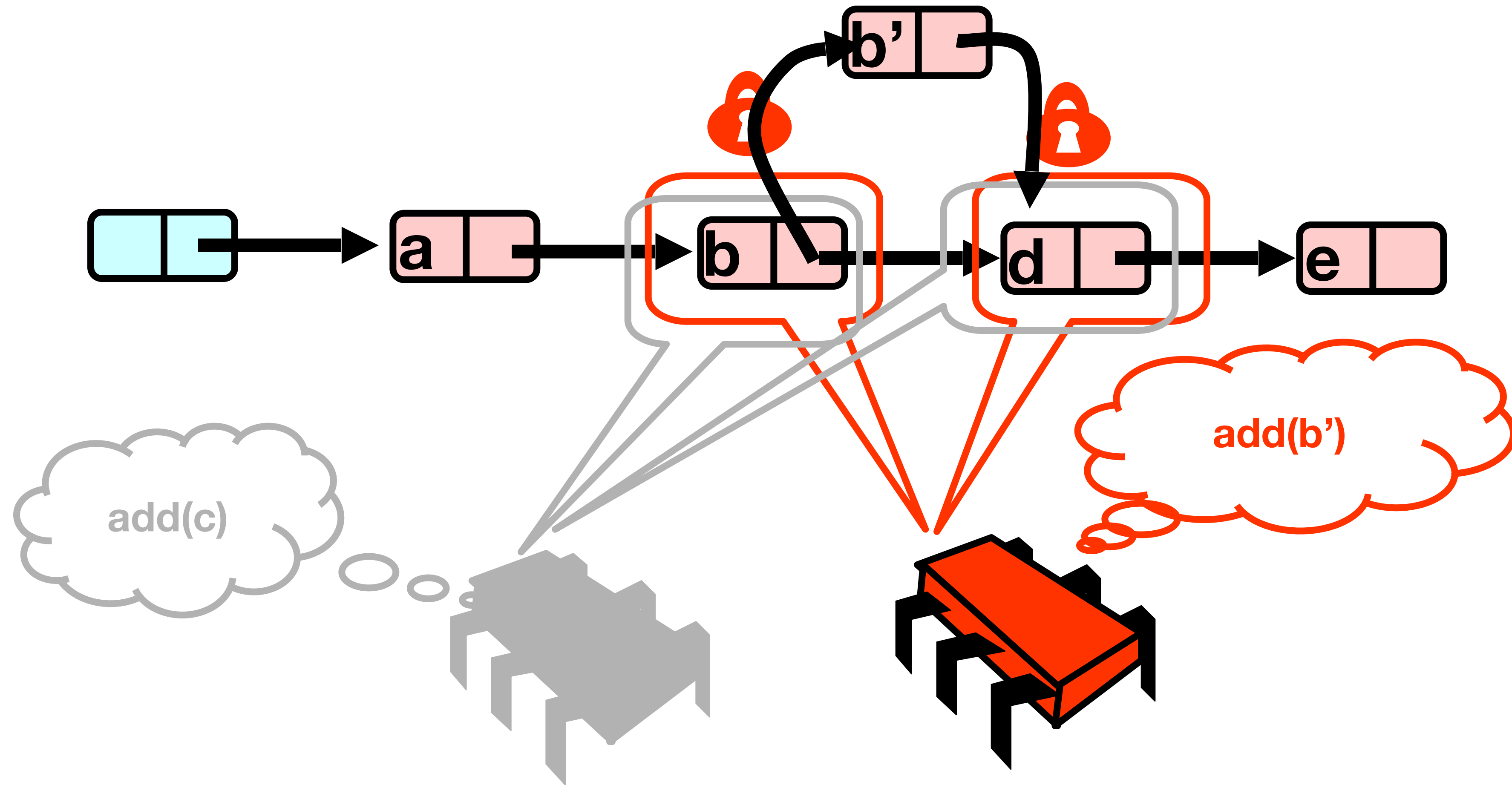
Validate – Part 1 (while holding locks)



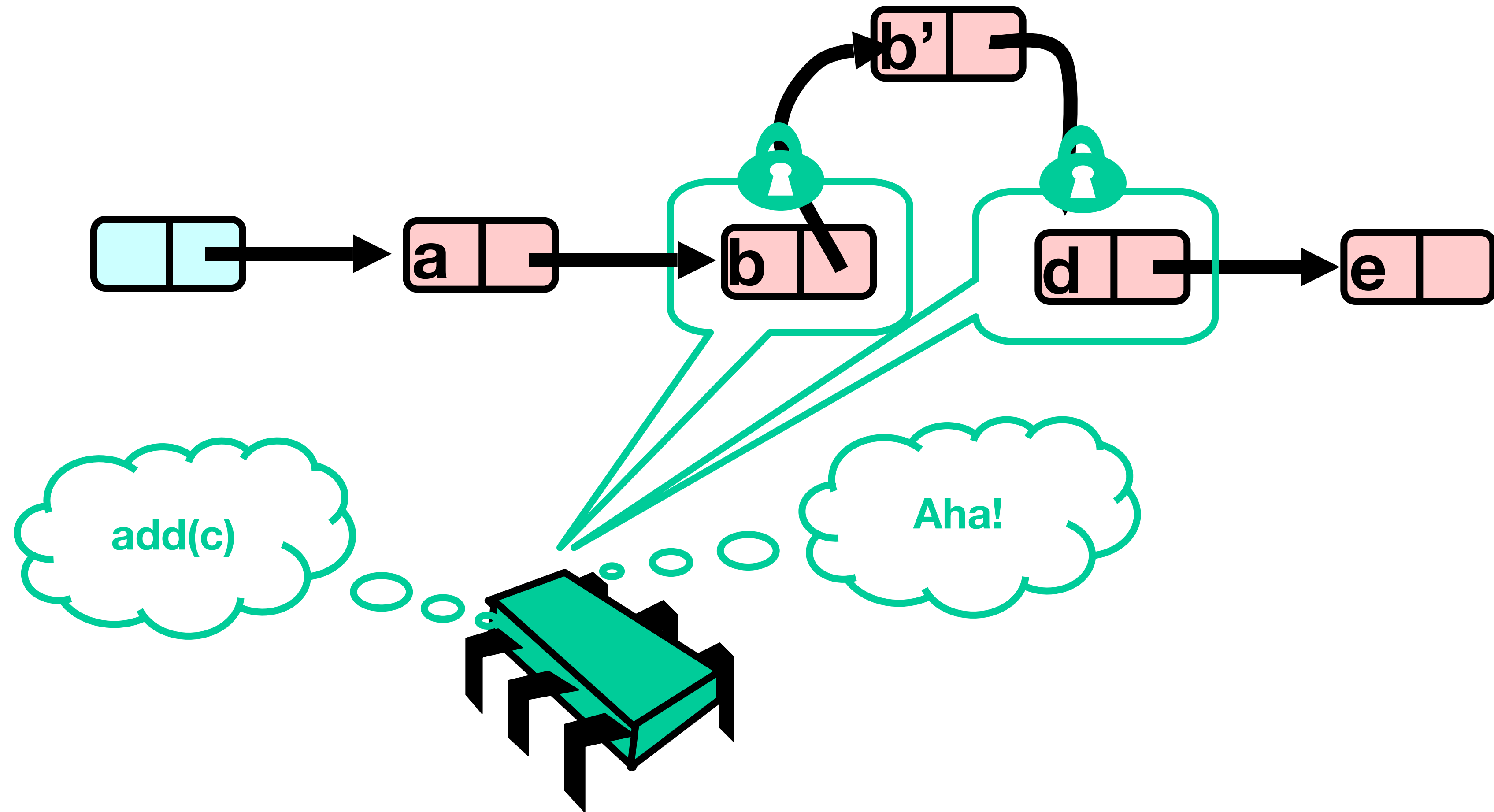
What Else Can Go Wrong?



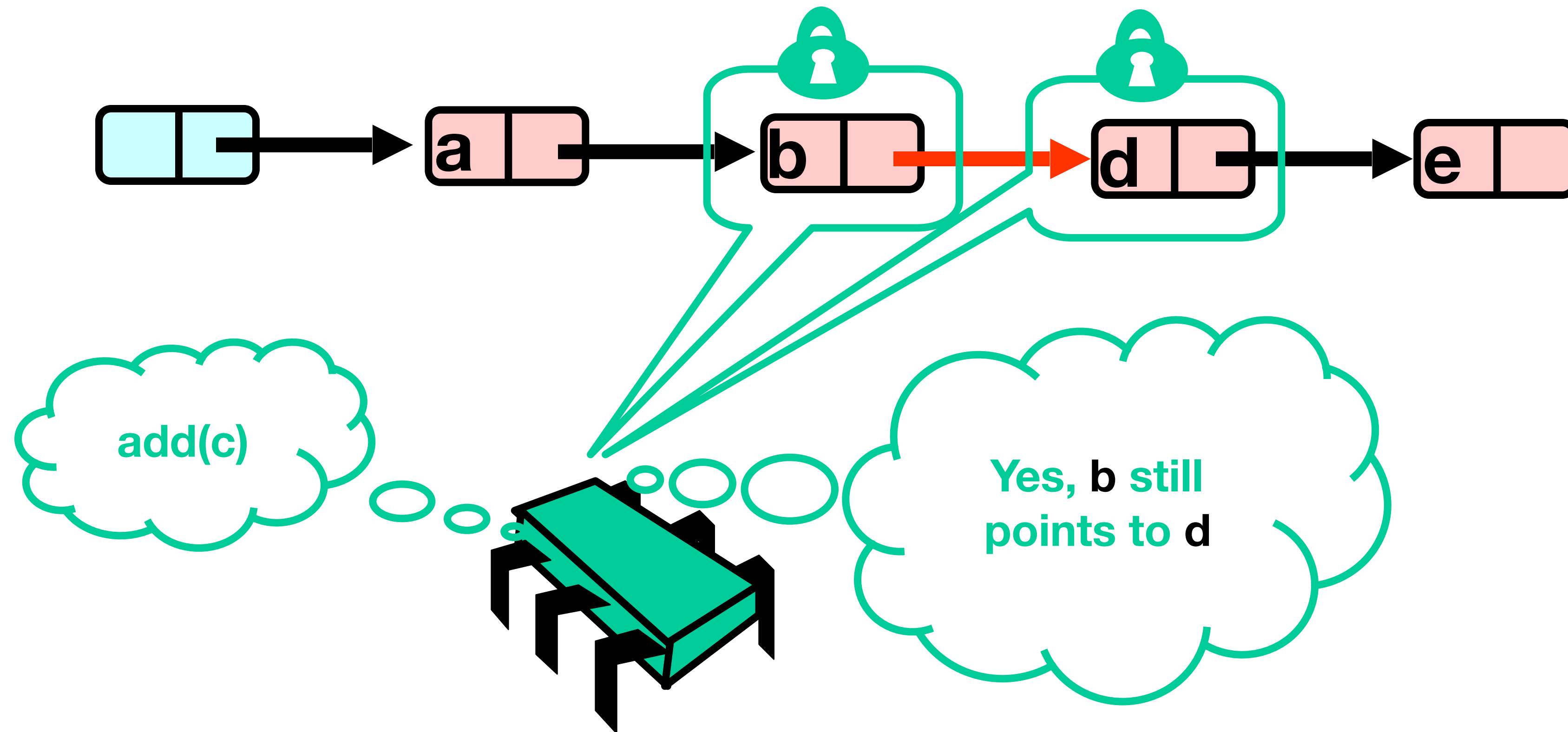
What Else Can Go Wrong?



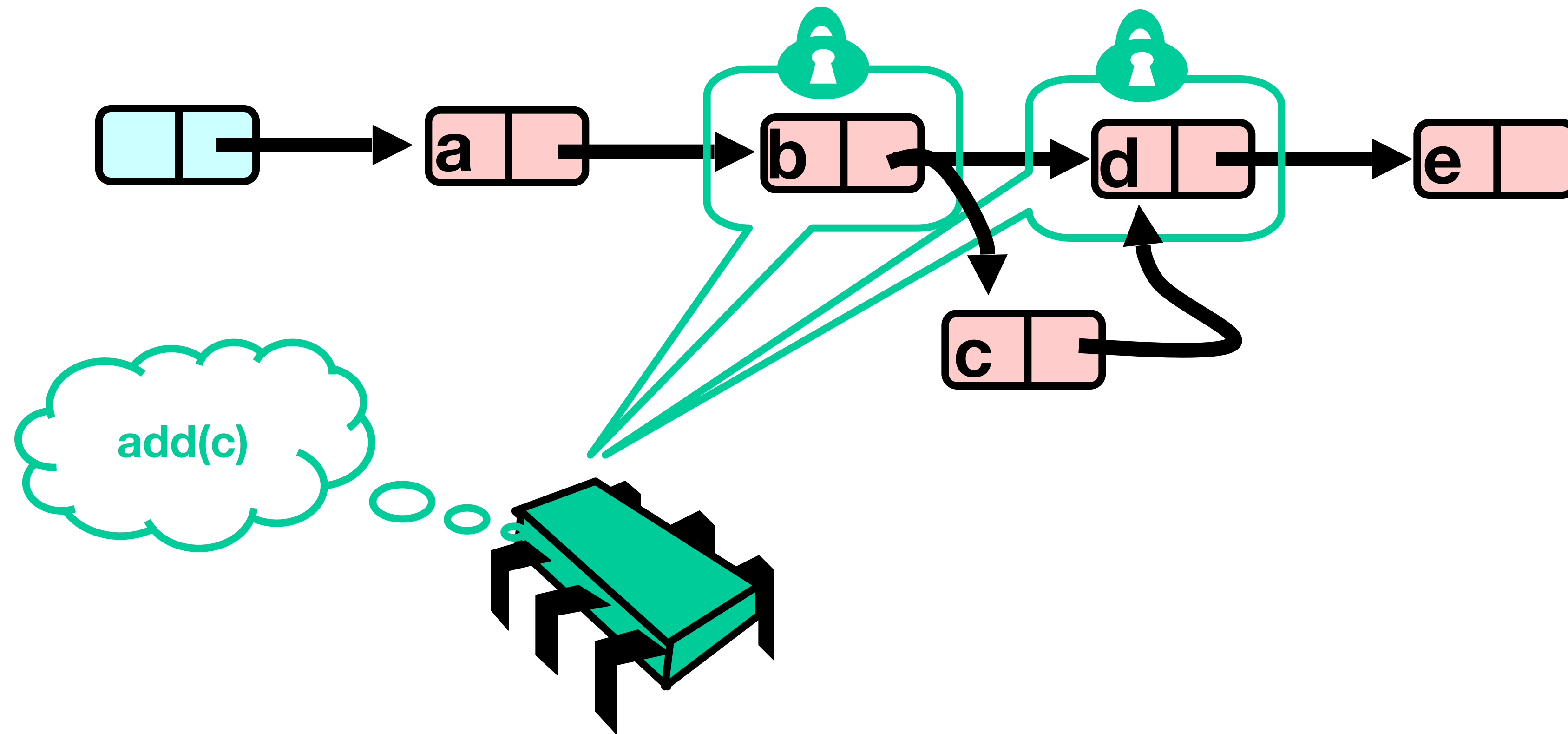
What Else Can Go Wrong?



Validate Part 2 (while holding locks)



Optimistic: Linearization Point



Same Abstraction Map

- $S(\text{head}) =$
 - $\{ x \mid \text{there exists } a \text{ such that}$
 - a **reachable from head** **and**
 - $a.\text{item} = x$
 - $\}$

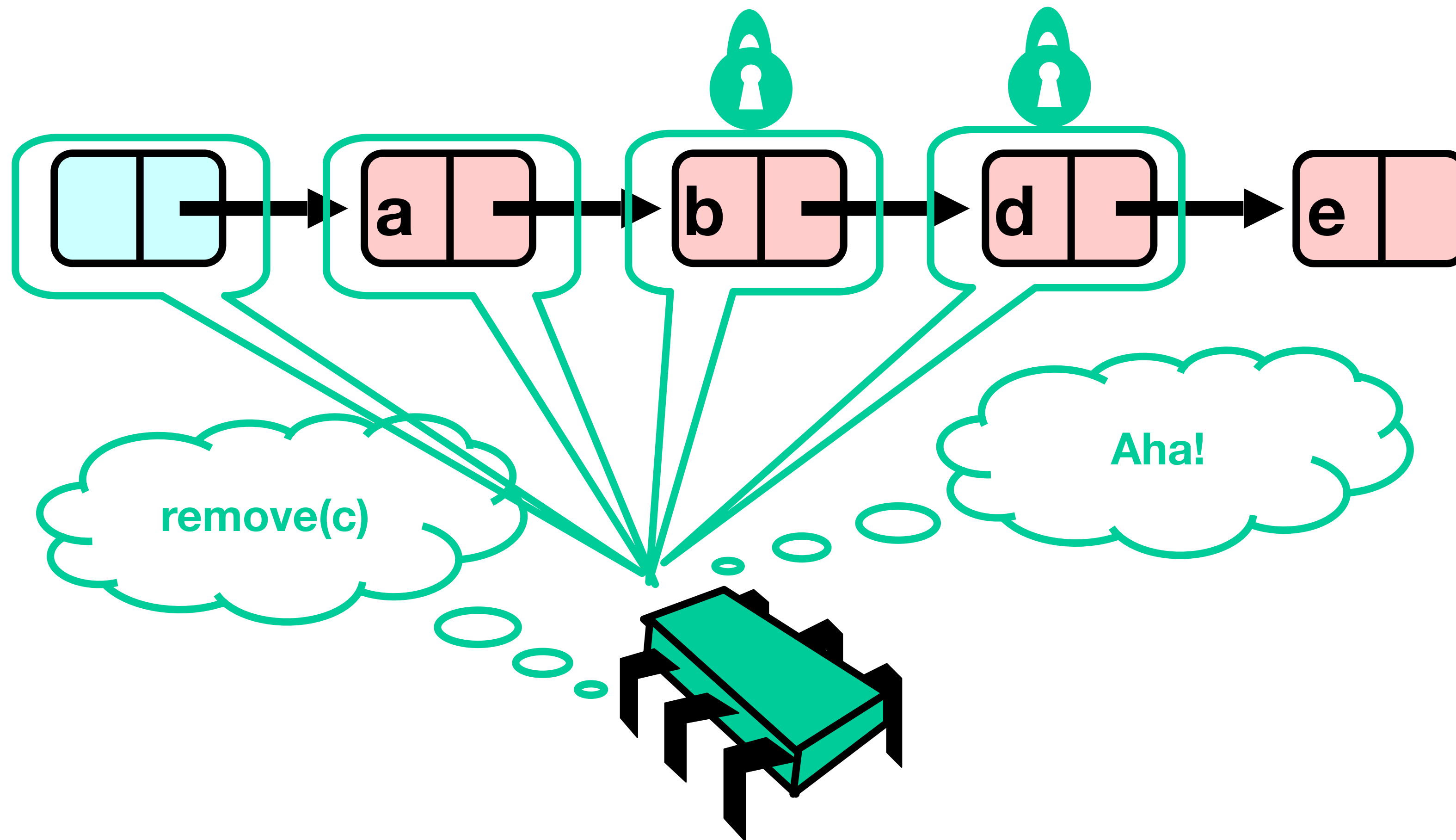
Invariants

- Careful: we may traverse deleted nodes
- But we establish properties by
 - Validation
 - After we lock target nodes

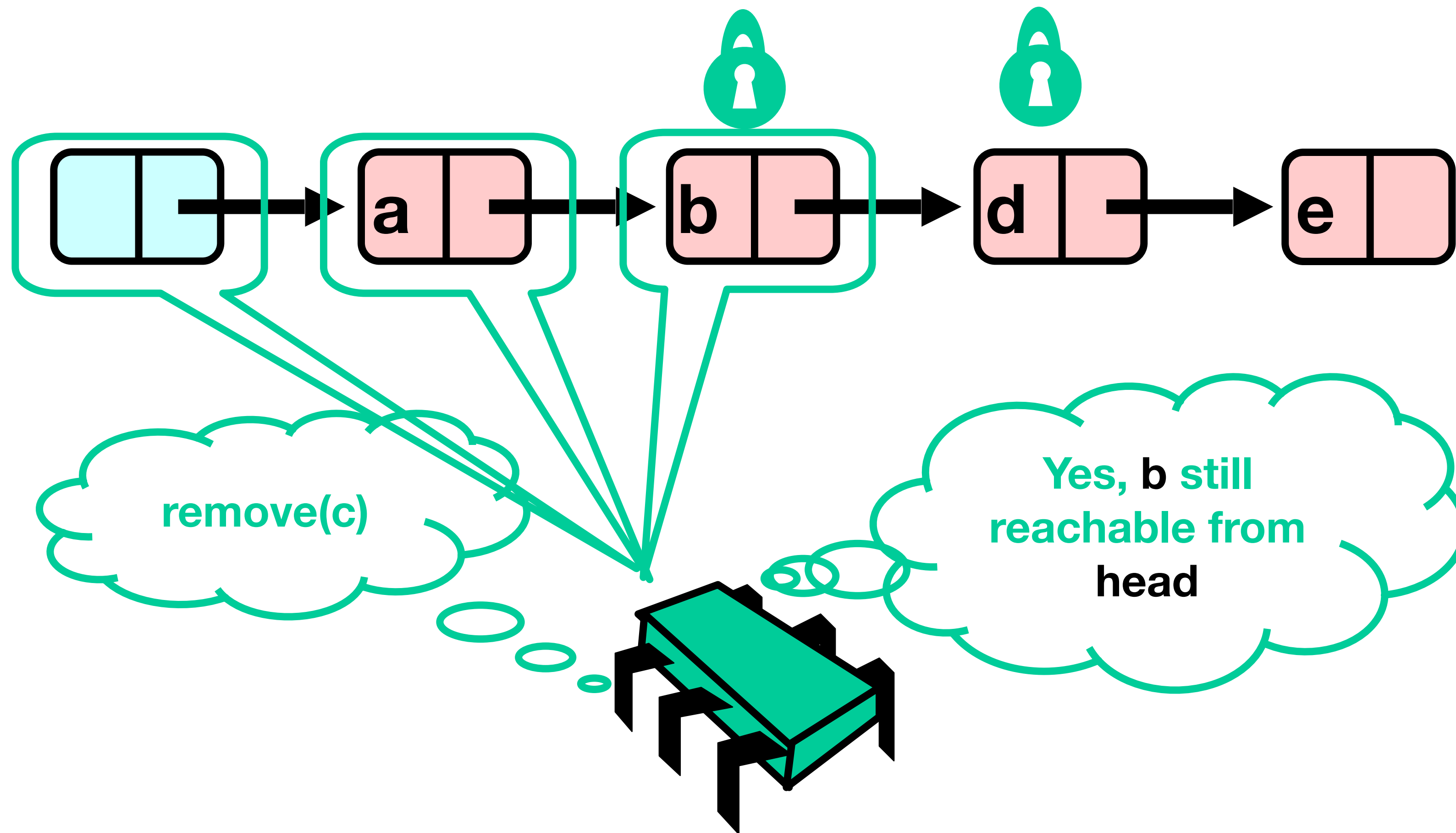
Correctness

- 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

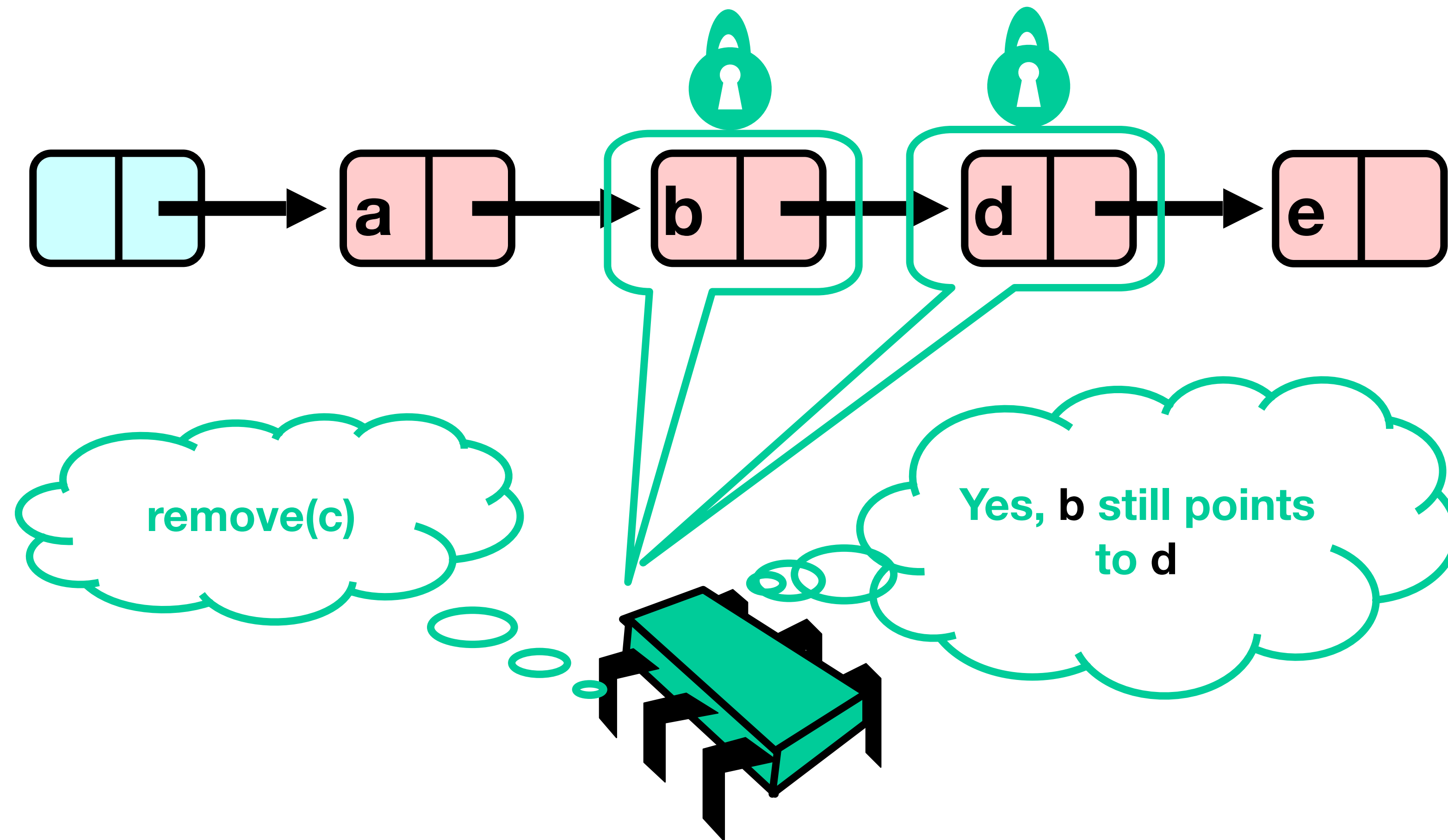
Unsuccessful Remove



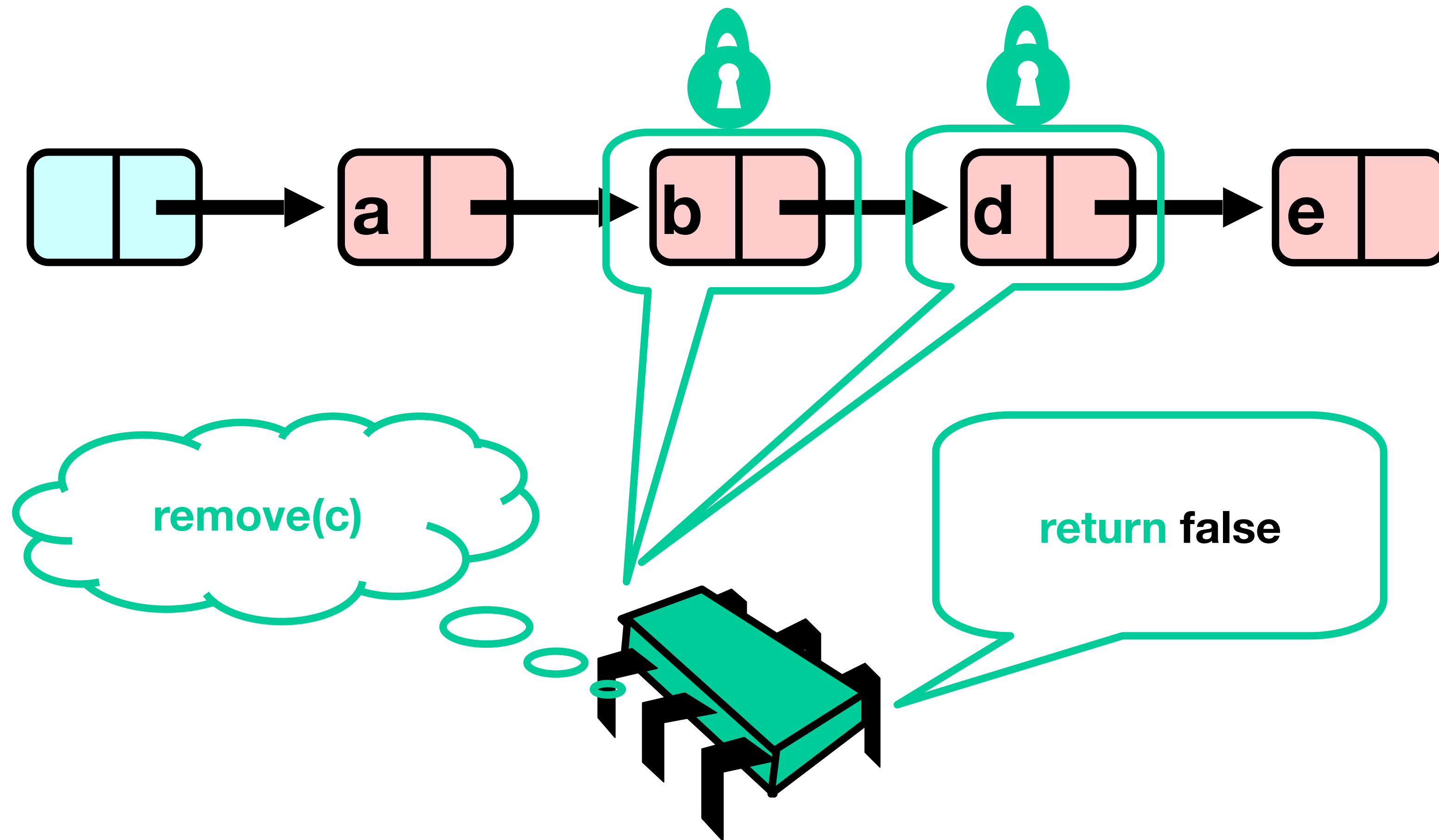
Validate (1)



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

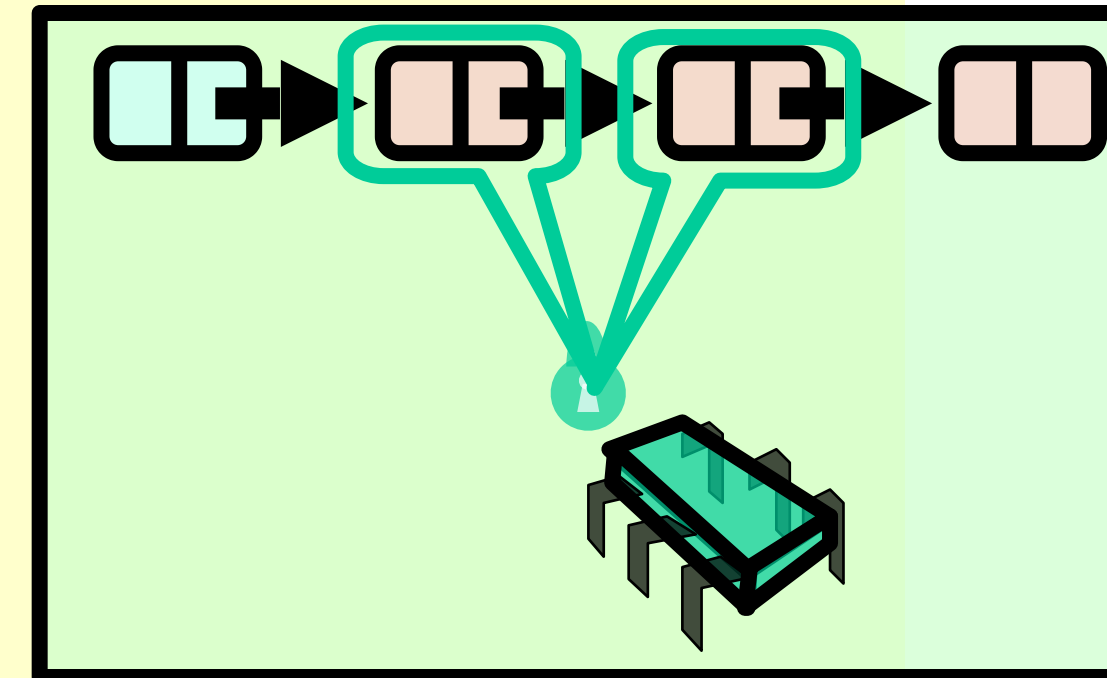
Validation

```
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

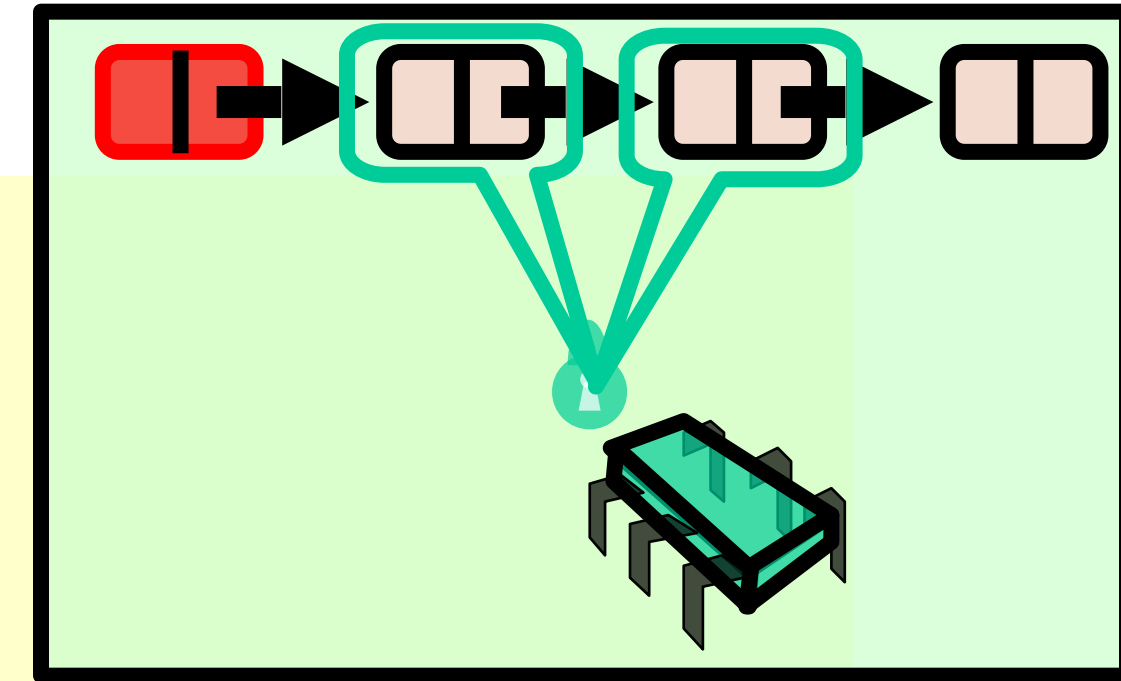
```
private boolean  
validate(Node pred,  
Node curr) {  
Node node = head;  
while (node.key <= pred.key) {  
if (node == pred)  
return pred.next == curr;  
node = node.next;  
}  
return false;  
}
```

**Predecessor &
current nodes**



Validation

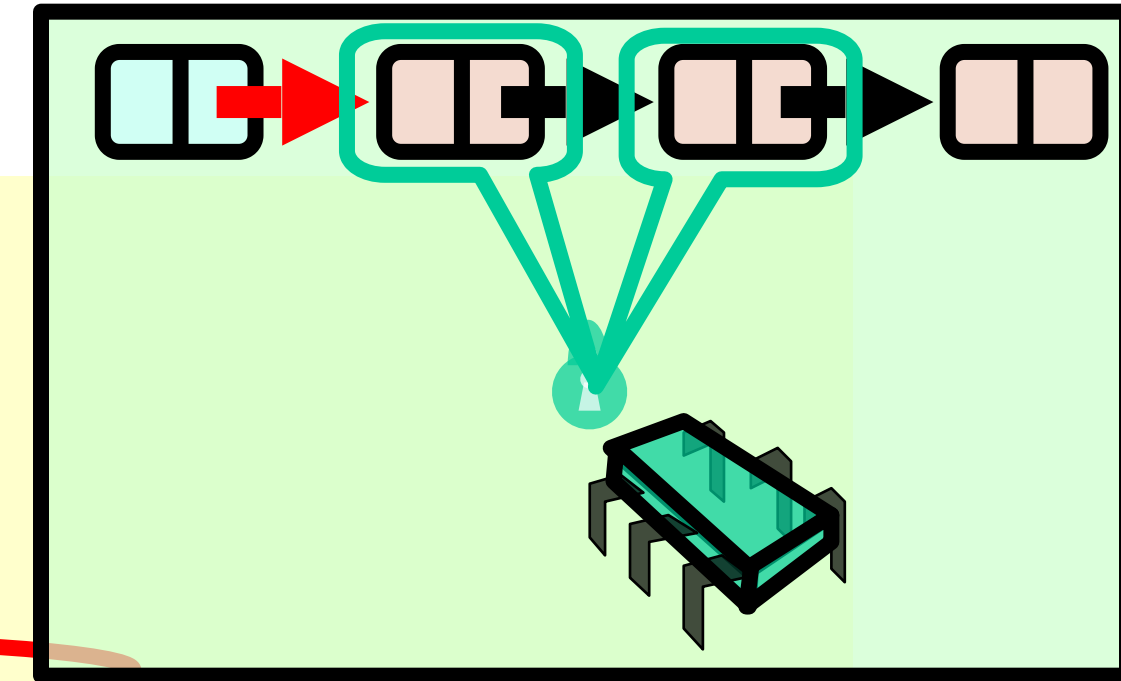
```
private boolean  
validate(Node pred,  
         Node curr) {  
Node node = head;  
while (node.key <= pred.key) {  
    if (node == pred)  
        return pred.next == curr;  
    node = node.next;  
}  
return false;  
}
```



Begin at the beginning

Validation

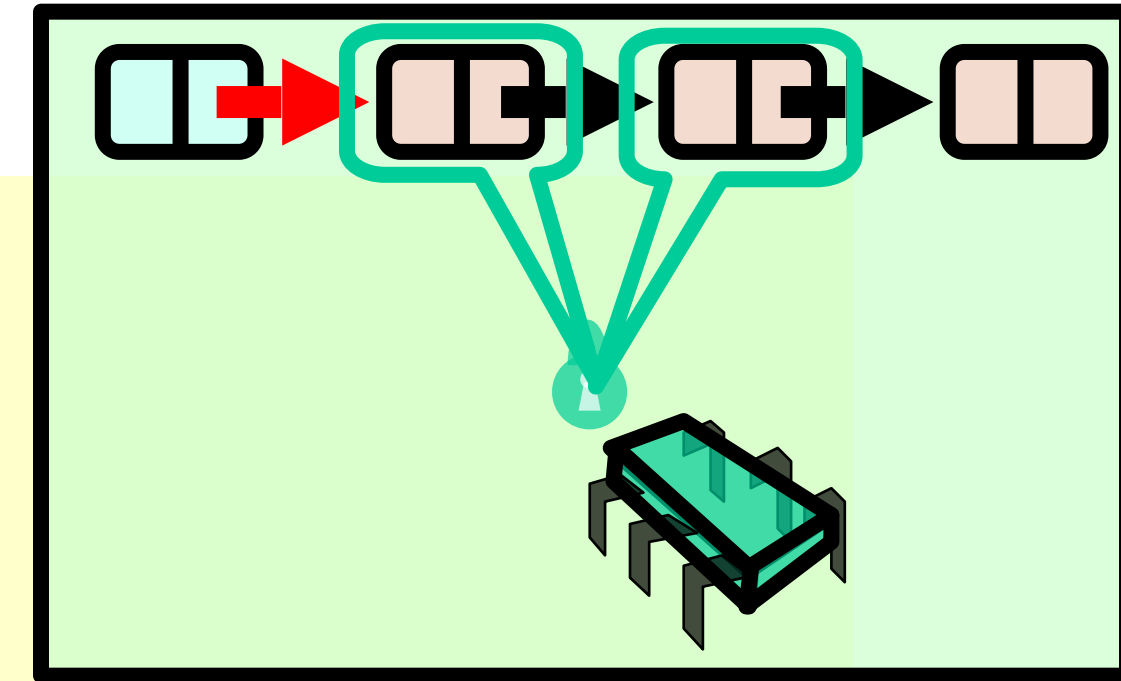
```
private boolean
validate(Node pred,
        Node curr) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}
```



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;
}
```

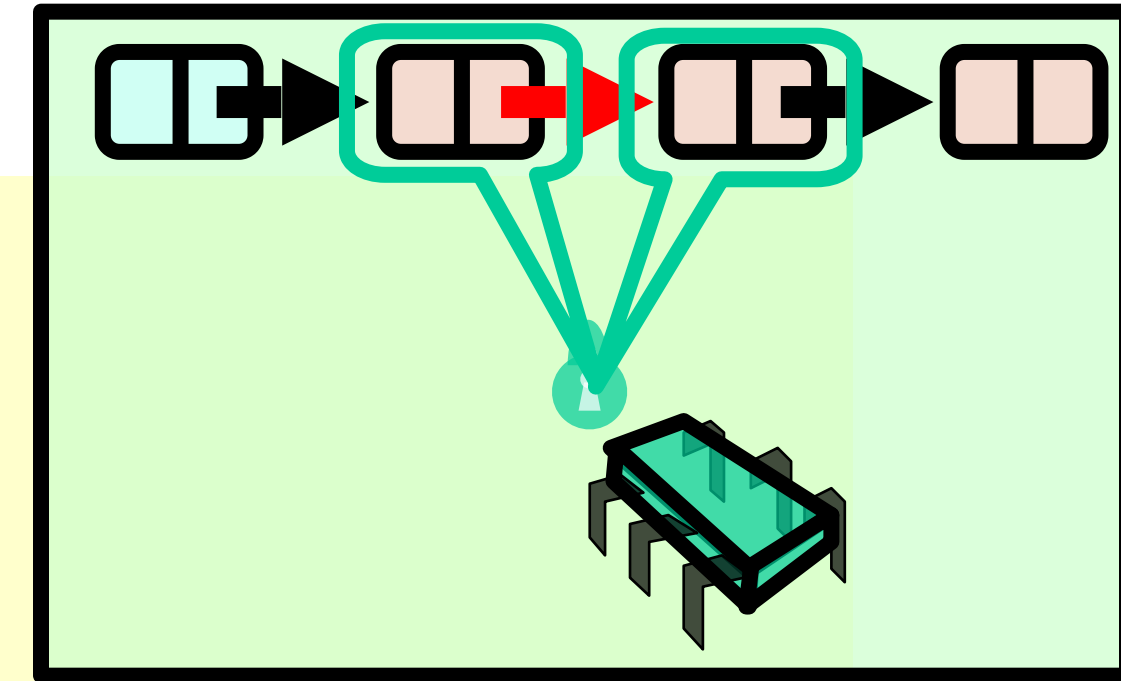


Predecessor reachable

Validation

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

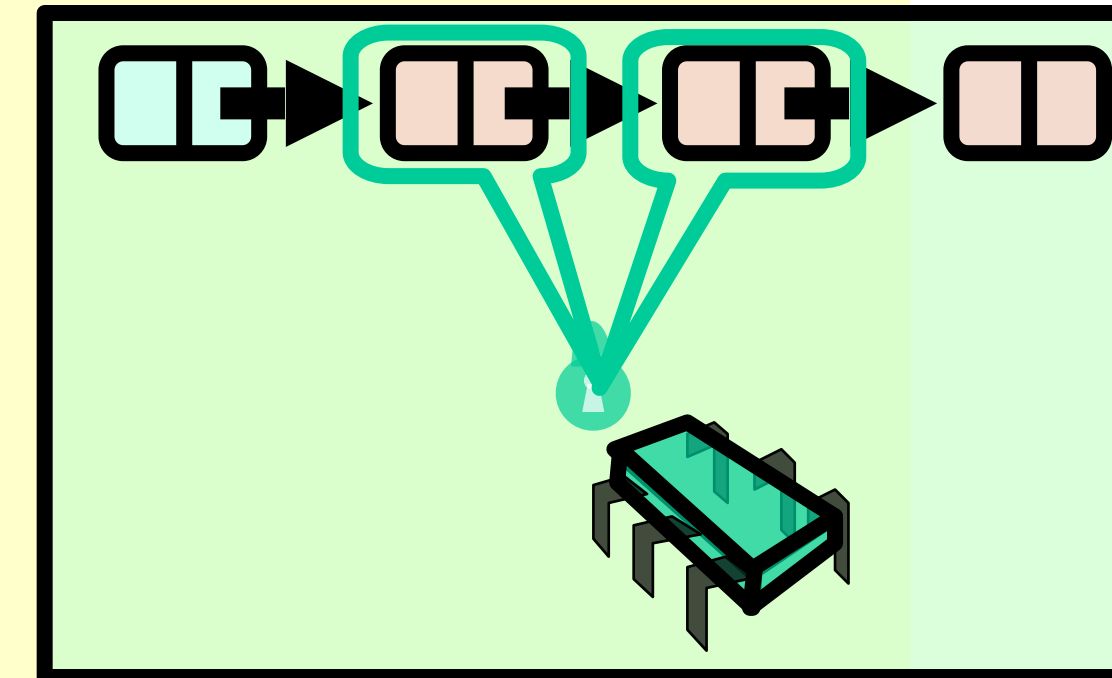
Is current node next?



Validation

Otherwise move on

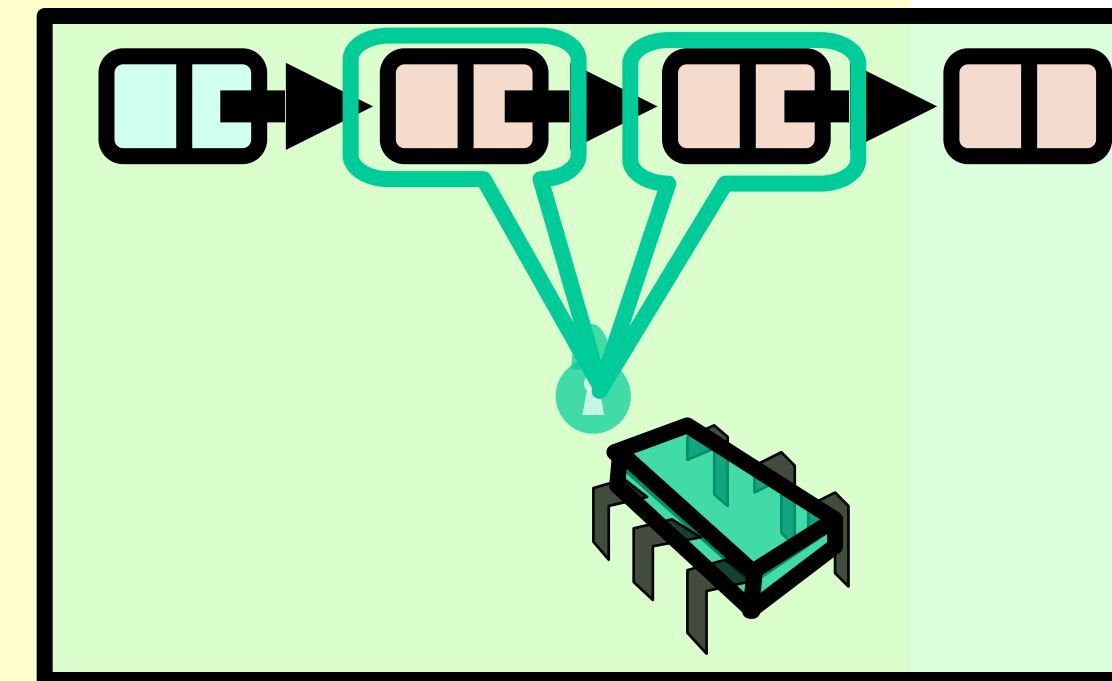
```
private boolean
validate(Node pred,
        Node curr) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}
```



Validation

Predecessor not reachable

```
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;
}
```

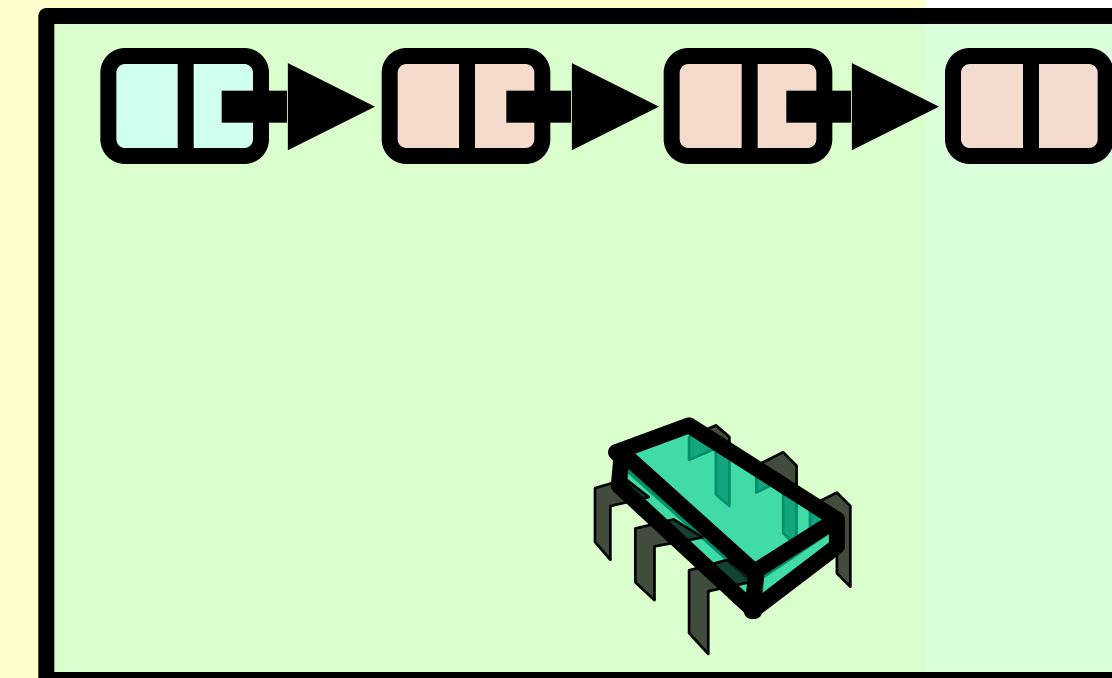


Remove: searching

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    retry: while (true) {  
        Node pred = this.head;  
        Node curr = pred.next;  
        while (curr.key <= key) {  
            if (item == curr.item)  
                break;  
            pred = curr;  
            curr = curr.next;  
        } ...  
    }  
}
```

Remove: searching

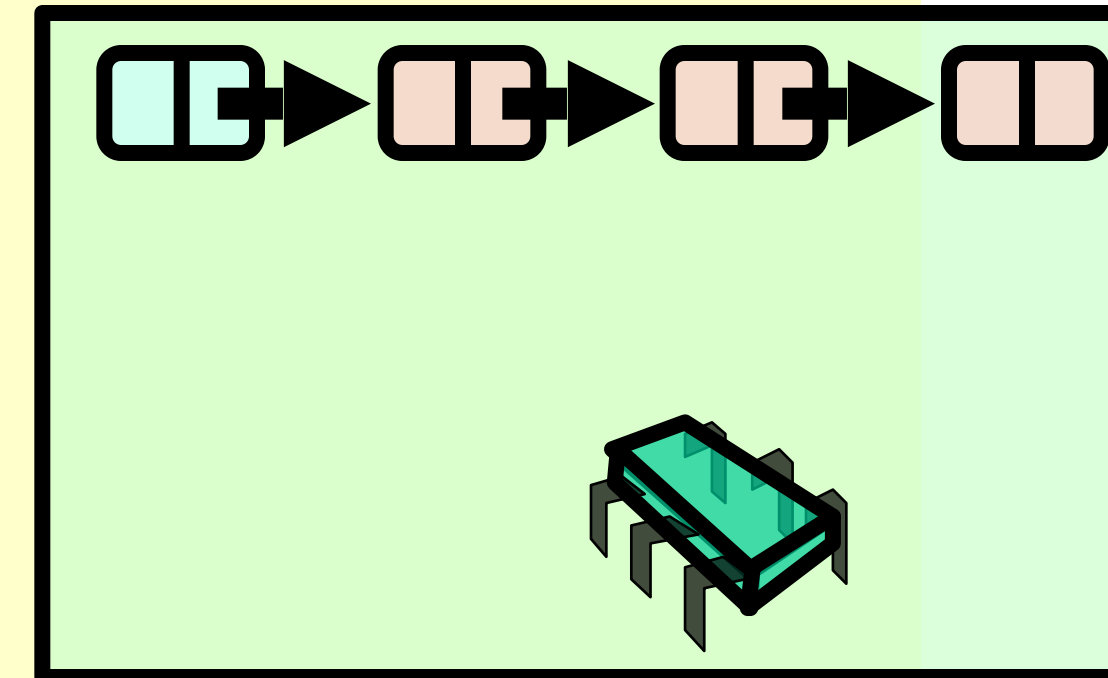
```
public boolean remove(Item item) {  
int key = item.hashCode();  
  retry: while (true) {  
    Node pred = this.head;  
    Node curr = pred.next;  
    while (curr.key <= key) {  
      if (item == curr.item)  
        break;  
      pred = curr;  
      curr = curr.next;  
    } ...  
  }  
}
```



Search key

Remove: searching

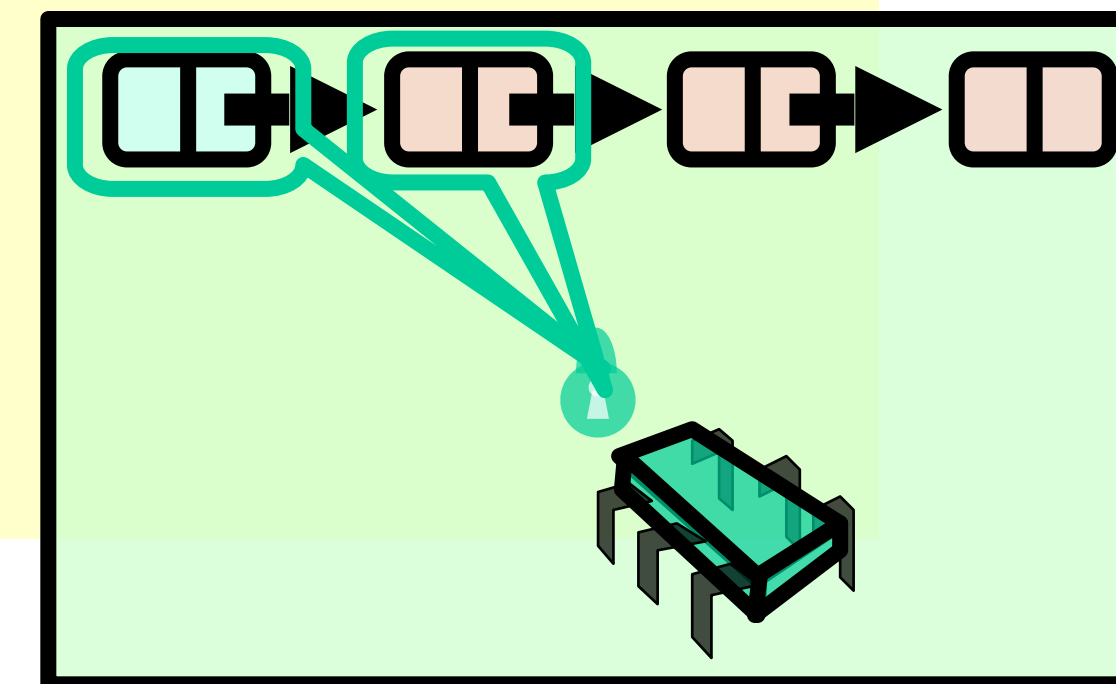
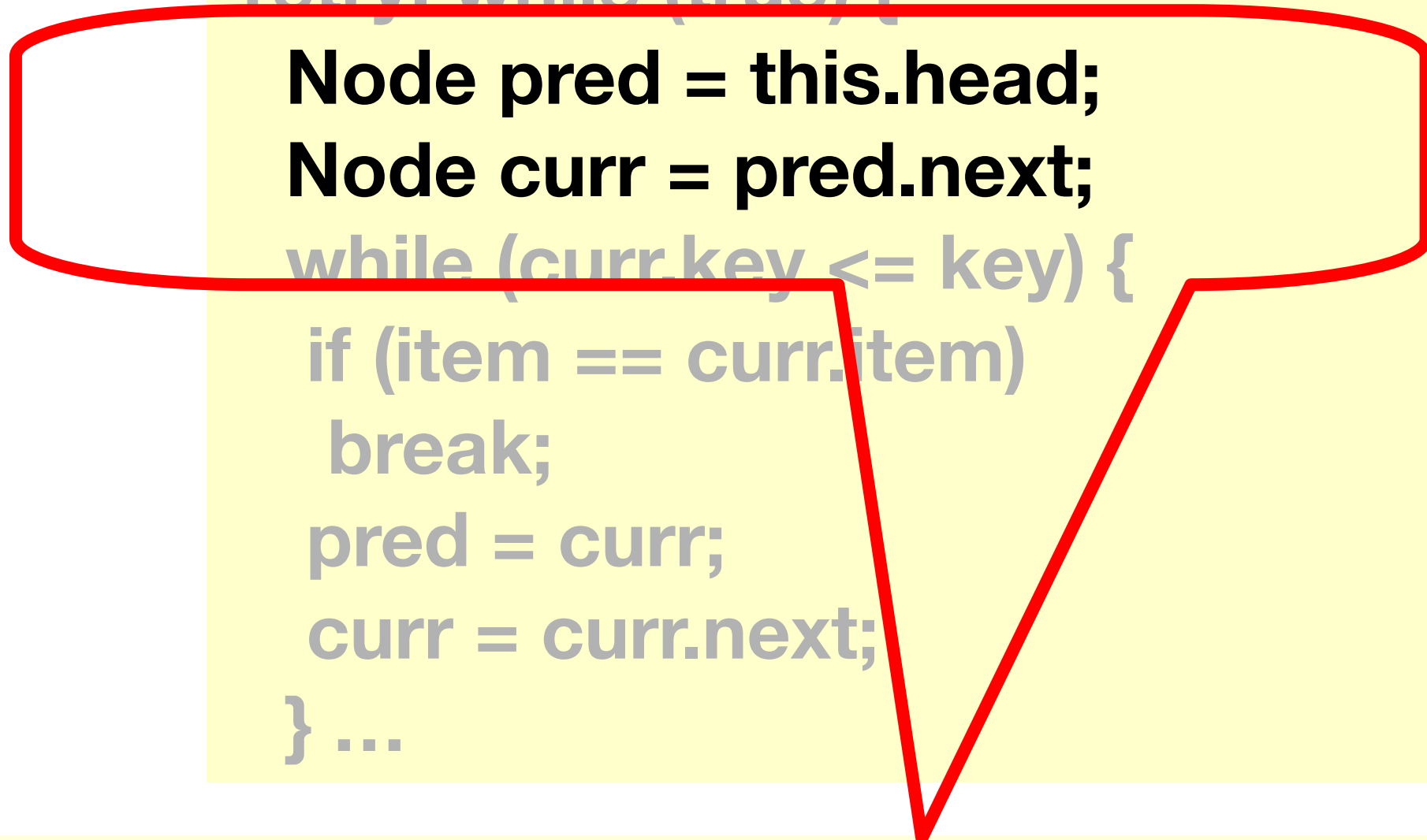
```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    retry: while (true) {  
        Node pred = this.head;  
        Node curr = pred.next;  
        while (curr.key <= key) {  
            if (item == curr.item)  
                break;  
            pred = curr;  
            curr = curr.next;  
        } ...  
    } ...  
}
```



Retry on synchronization conflict

Remove: searching

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    retry: while (true) {  
        Node pred = this.head;  
        Node curr = pred.next;  
        while (curr.key <= key) {  
            if (item == curr.item)  
                break;  
            pred = curr;  
            curr = curr.next;  
        } ...  
    }  
}
```

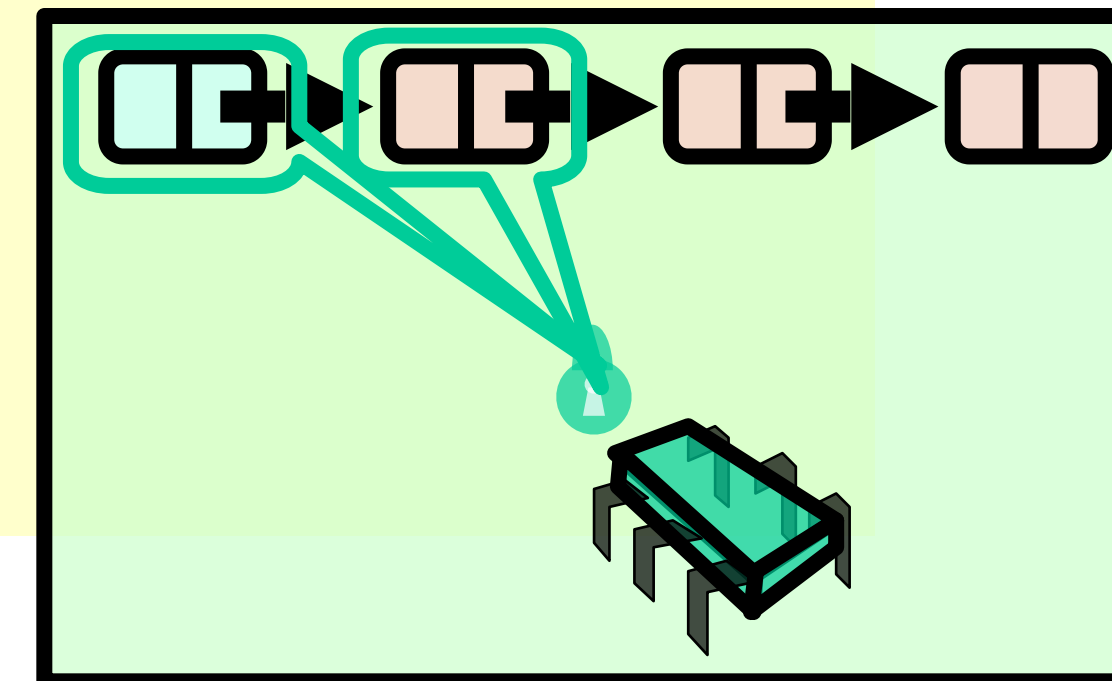


Examine predecessor and current nodes

Remove: searching

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    retry: while (true) {  
        Node pred = this.head;  
        Node curr = pred.next;  
        while (curr.key <= key) {  
            if (item == curr.item)  
                break;  
            pred = curr;  
            curr = curr.next;  
        }  
        ...  
    }  
}
```

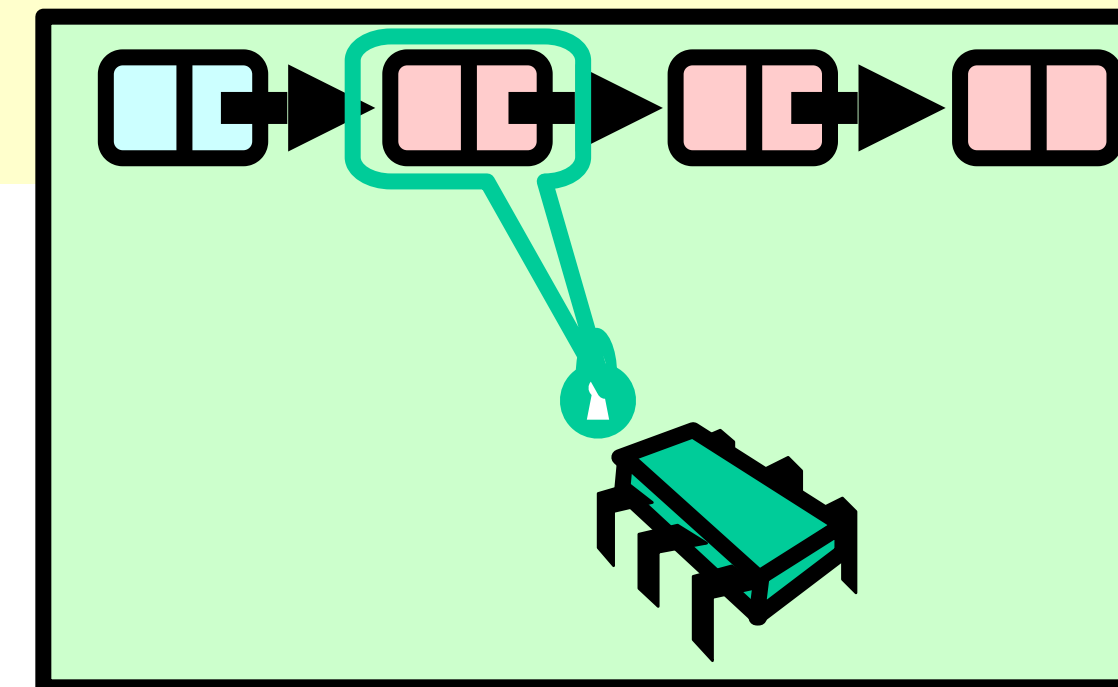
Search by key



Remove: searching

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    retry: while (true) {  
        Node pred = this.head;  
        Node curr = pred.next;  
        while (curr.key <= key) {  
            if (item == curr.item)  
                break;  
            pred = curr;  
            curr = curr.next;  
        } ...  
    }  
}
```

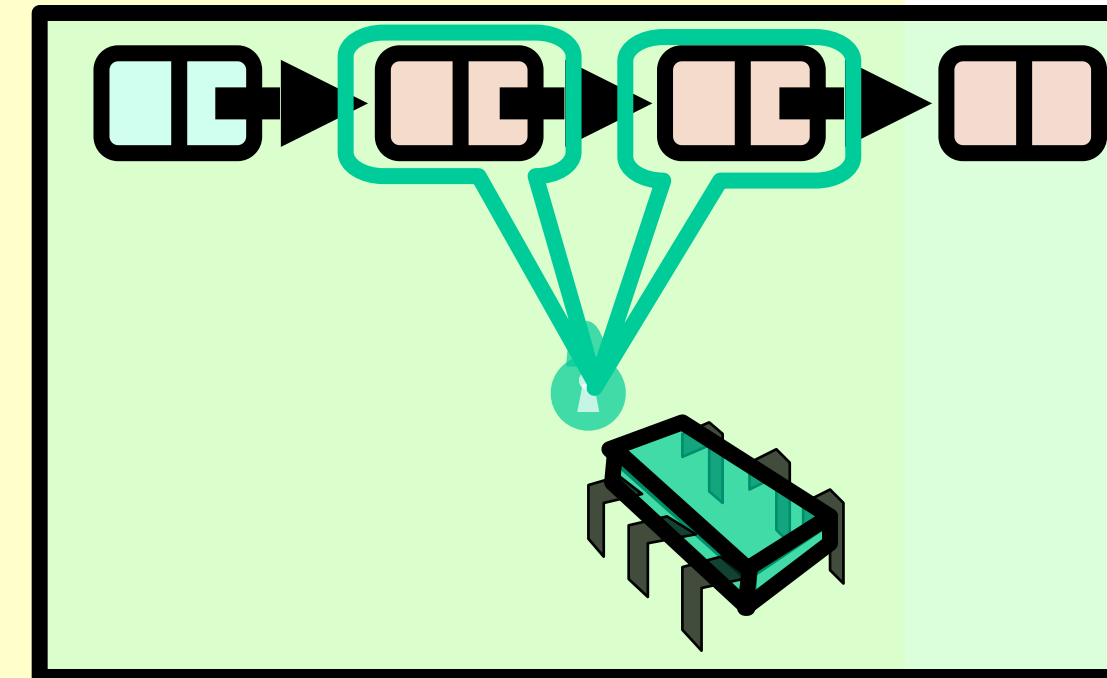
Stop if we find item



Remove: searching

Move along

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    retry: while (true) {  
        Node pred = this.head;  
        Node curr = pred.next;  
        while (curr.key <= key) {  
            if (item == curr.item)  
                break;  
            pred = curr;  
            curr = curr.next;  
        }  
    }  
}
```



On Exit from Loop

- If item is present
 - curr holds item
 - pred just before curr
- If item is absent
 - curr has first higher key
 - pred just before curr
- Assuming no synchronization problems

Remove Method

```
try {  
  pred.lock(); curr.lock();  
  if (validate(pred,curr) {  
    if (curr.item == item) {  
      pred.next = curr.next;  
      return true;  
    } else {  
      return false;  
    }  
  }  
} finally {  
  pred.unlock();  
  curr.unlock();  
}
```

Remove Method

```
try {  
    pred.lock(); curr.lock();  
    if (validate(pred,curr) {  
        if (curr.item == item) {  
            pred.next = curr.next;  
            return true;  
        } else {  
            return false;  
        }  
    }  
}} finally {  
    pred.unlock();  
    curr.unlock();  
}}
```

Always unlock

Remove Method

```
try {  
  pred.lock(); curr.lock();  
  if (validate(pred,curr) {  
    if (curr.item == item) {  
      pred.next = curr.next;  
      return true;  
    } else {  
      return false;  
    }  
  }  
} finally {  
  pred.unlock();  
  curr.unlock();  
}
```

Lock both nodes

Remove Method

```
try {  
    pred.lock(); curr.lock();  
    if (validate(pred,curr) {  
        if (curr.item == item) {  
            pred.next = curr.next;  
            return true;  
        } else {  
            return false;  
        }  
    }  
} finally {  
    pred.unlock();  
    curr.unlock();  
}
```

Check for synchronization conflicts

Remove Method

```
try {  
  pred.lock(); curr.lock();  
  if (validate(pred,curr) {  
    if (curr.item == item) {  
      pred.next = curr.next;  
      return true;  
    } else {  
      return false;  
    }  
  } finally {  
    pred.unlock();  
    curr.unlock();  
  }  
}
```

**target found, remove
node**

Remove Method

```
try {  
    pred.lock(); curr.lock();  
    if (validate(pred,curr) {  
        if (curr.item == item) {  
            pred.next = curr.next;  
            return true;  
        } else {  
            return false;  
        }  
    } finally {  
        pred.unlock();  
        curr.unlock();  
    }  
}
```

target not found

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 (it's a set!)

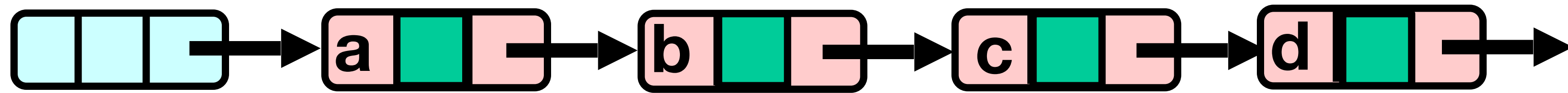
Lazy List

- Like optimistic, except
 - Scan once
 - `contains(x)` **never locks ...**
- Key insight
 - Removing nodes causes trouble
 - Do it “lazily”

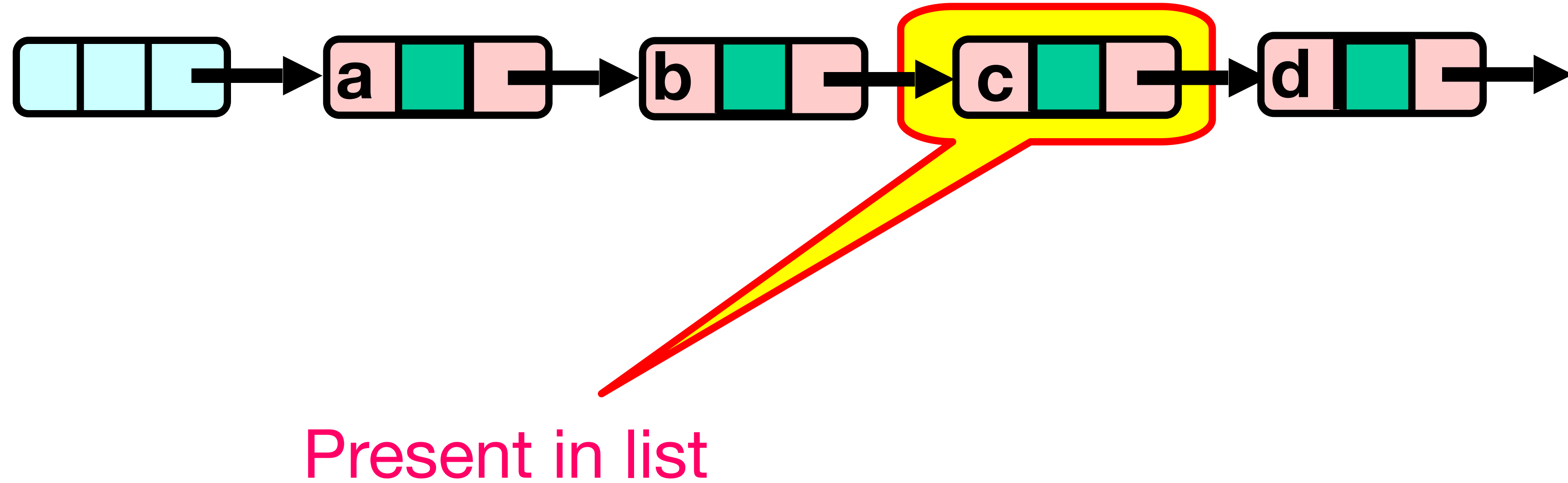
Lazy List

- `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)

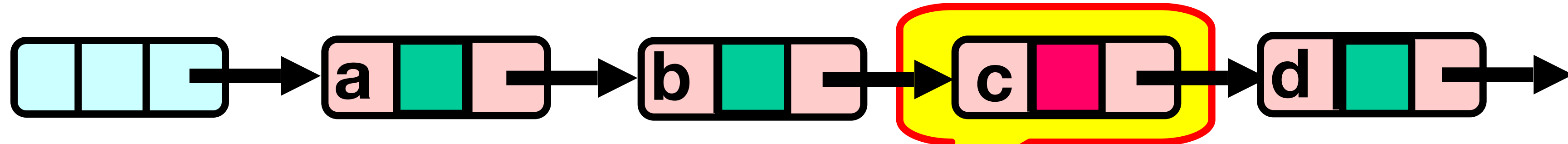
Lazy Removal



Lazy Removal

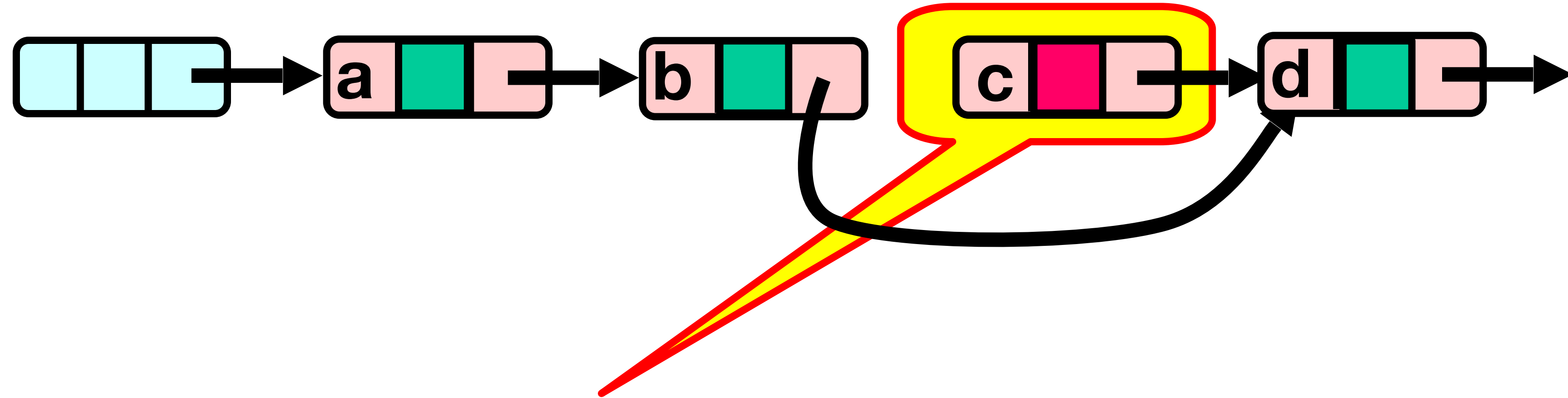


Lazy Removal



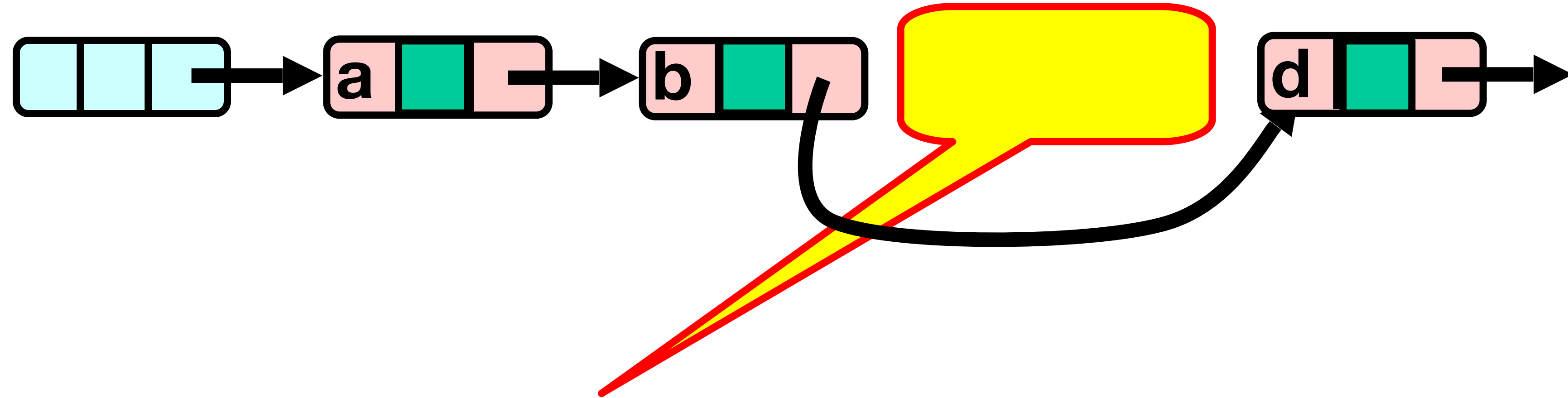
Logically deleted

Lazy Removal



Physically deleted

Lazy Removal



Physically deleted

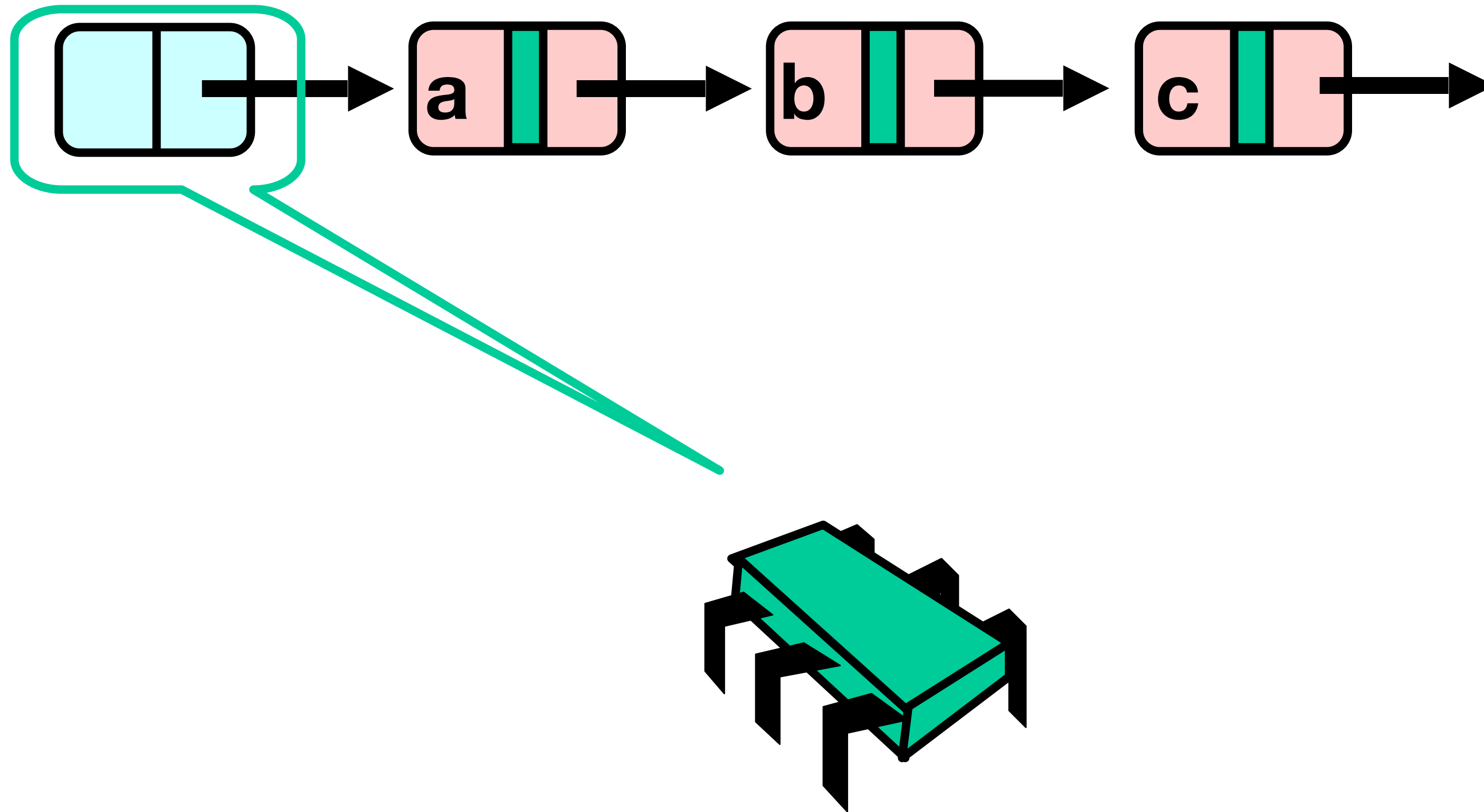
Lazy List

- 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.

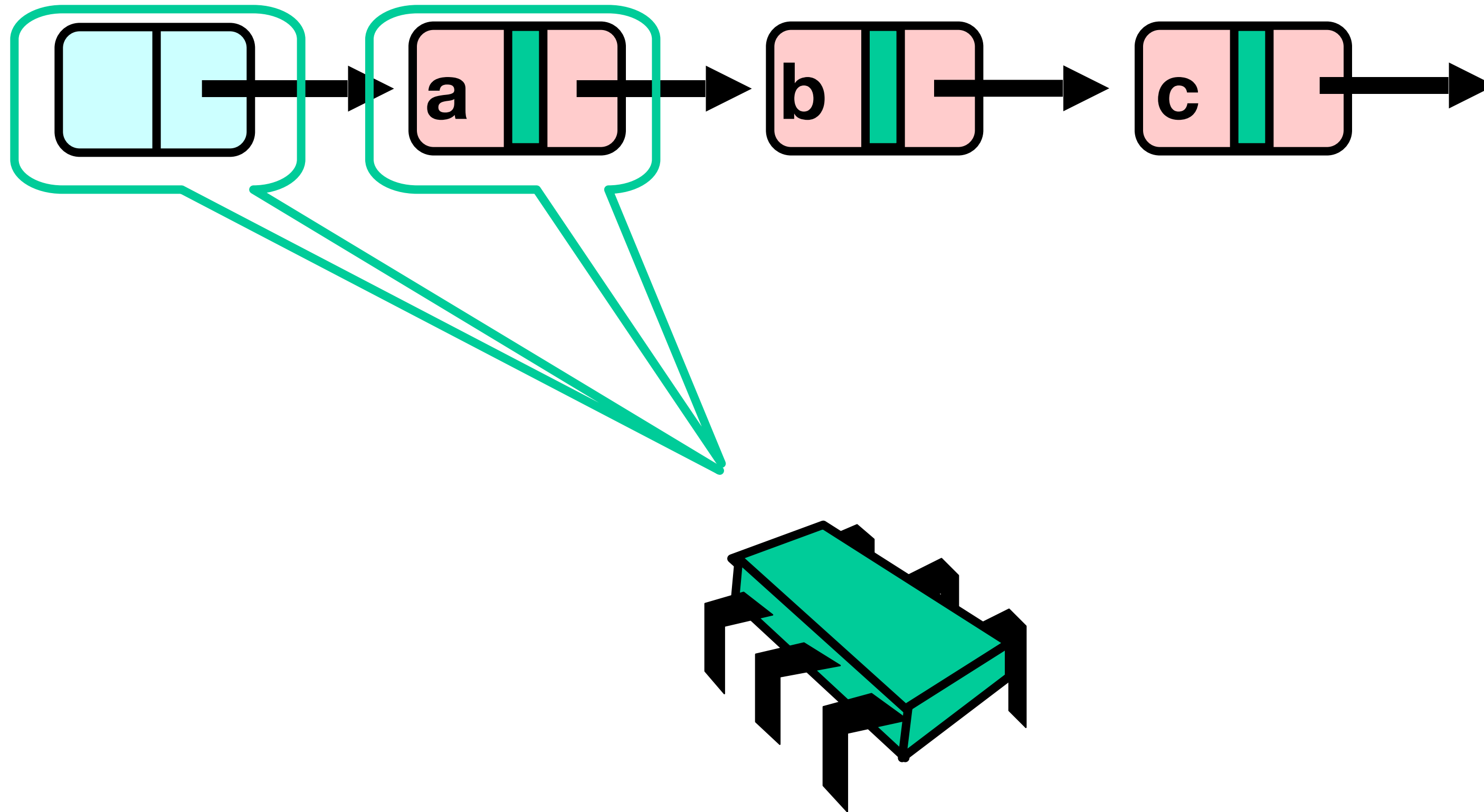
Validation

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

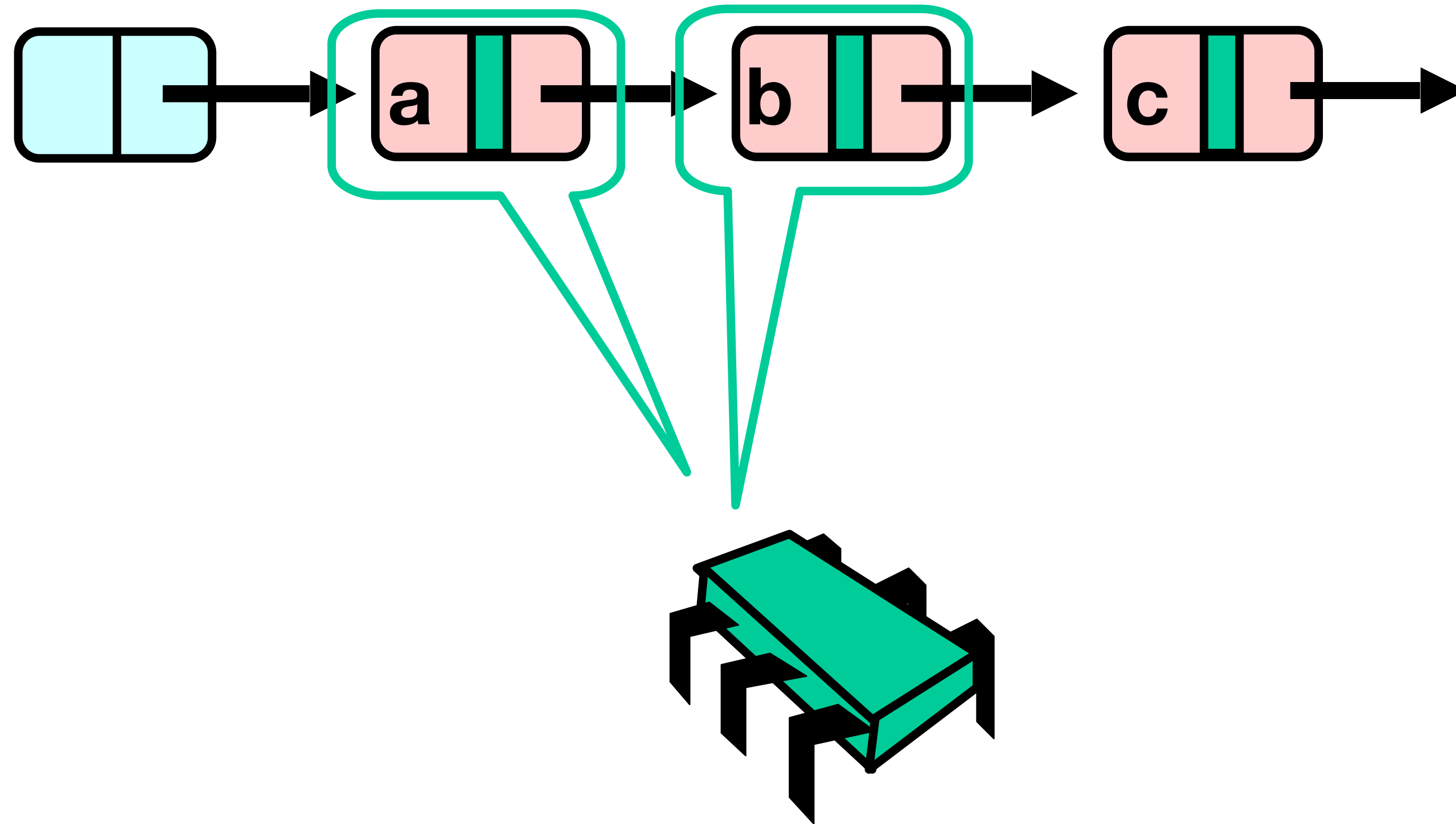
Business as Usual



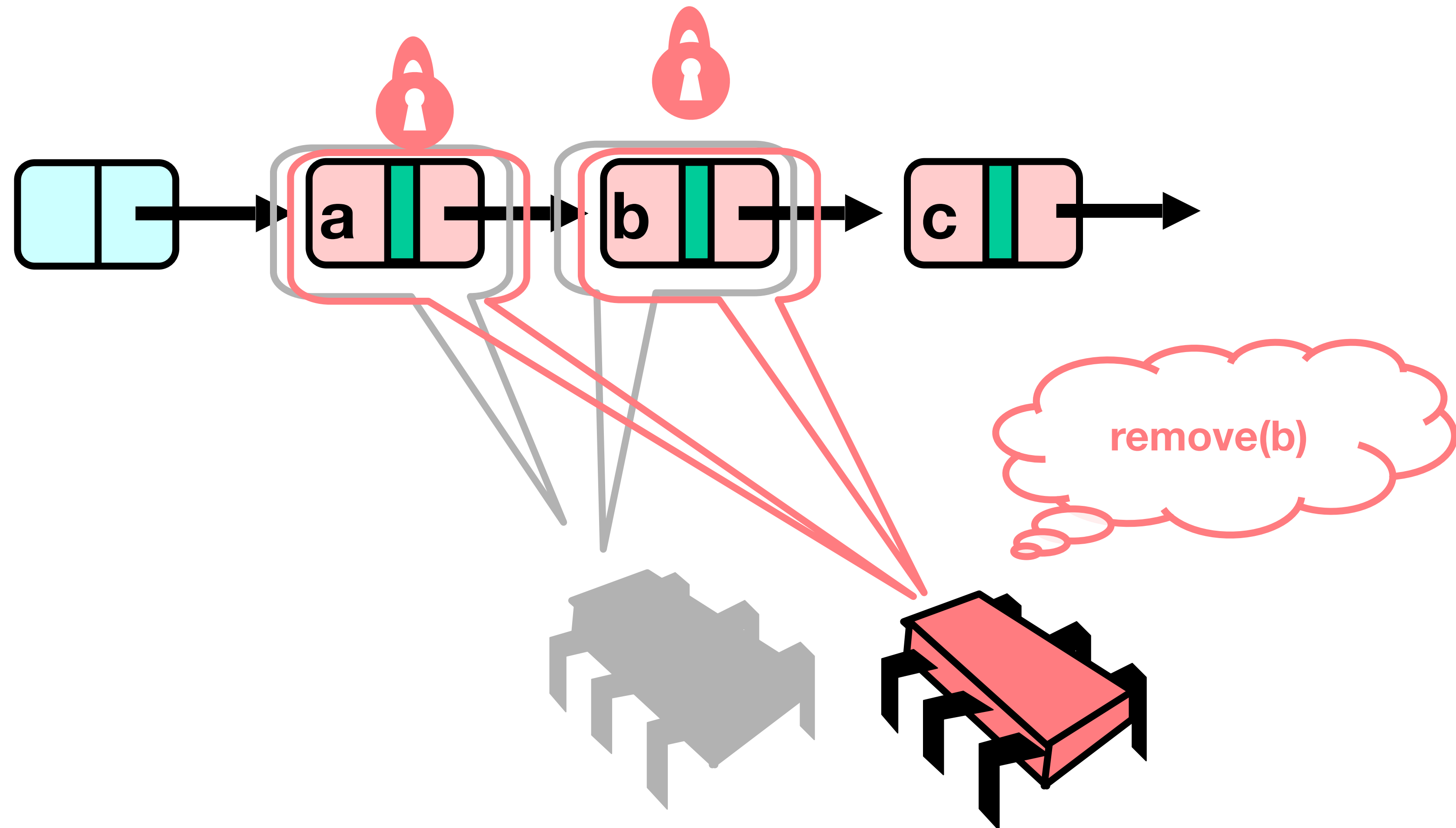
Business as Usual



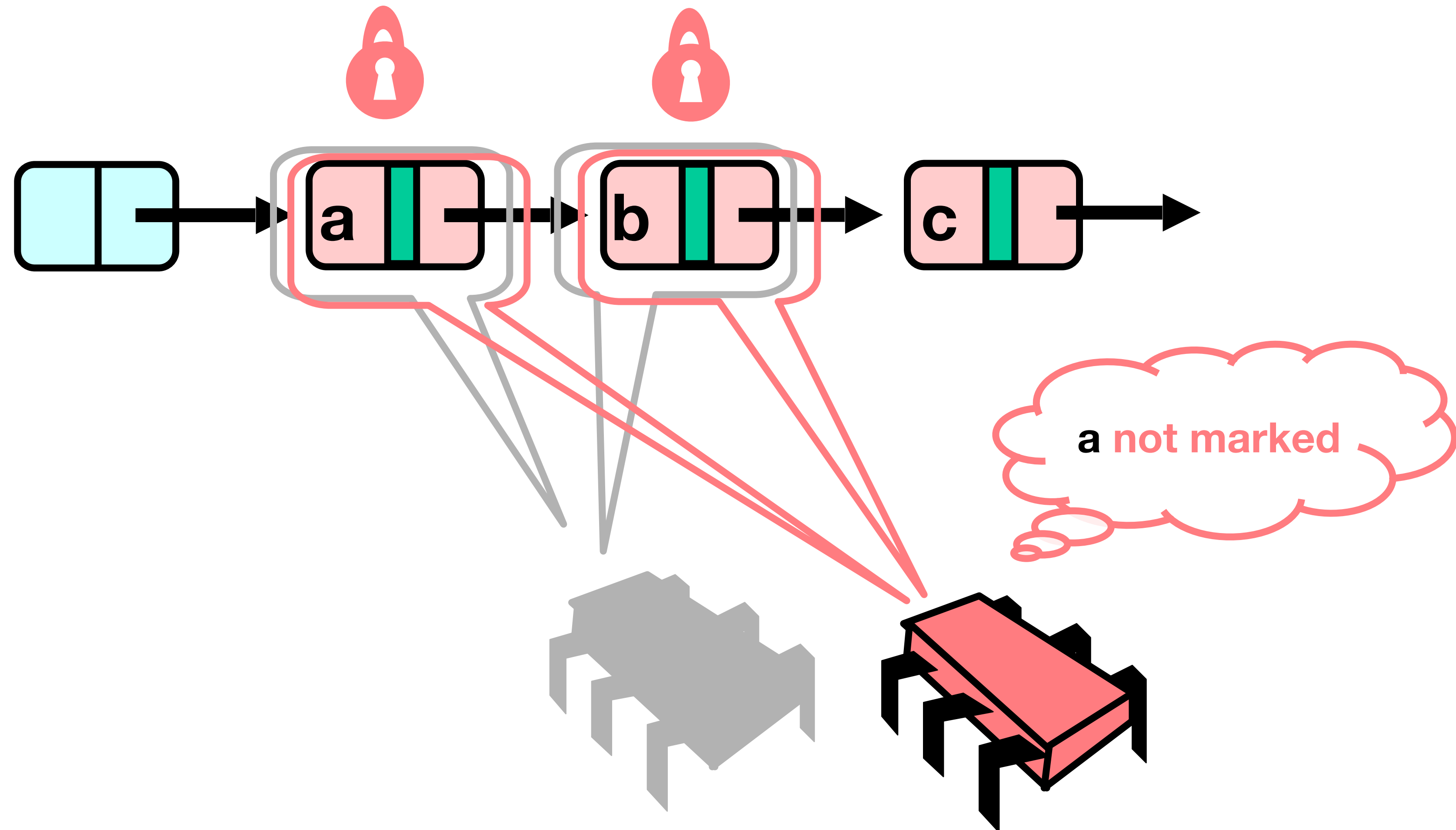
Business as Usual



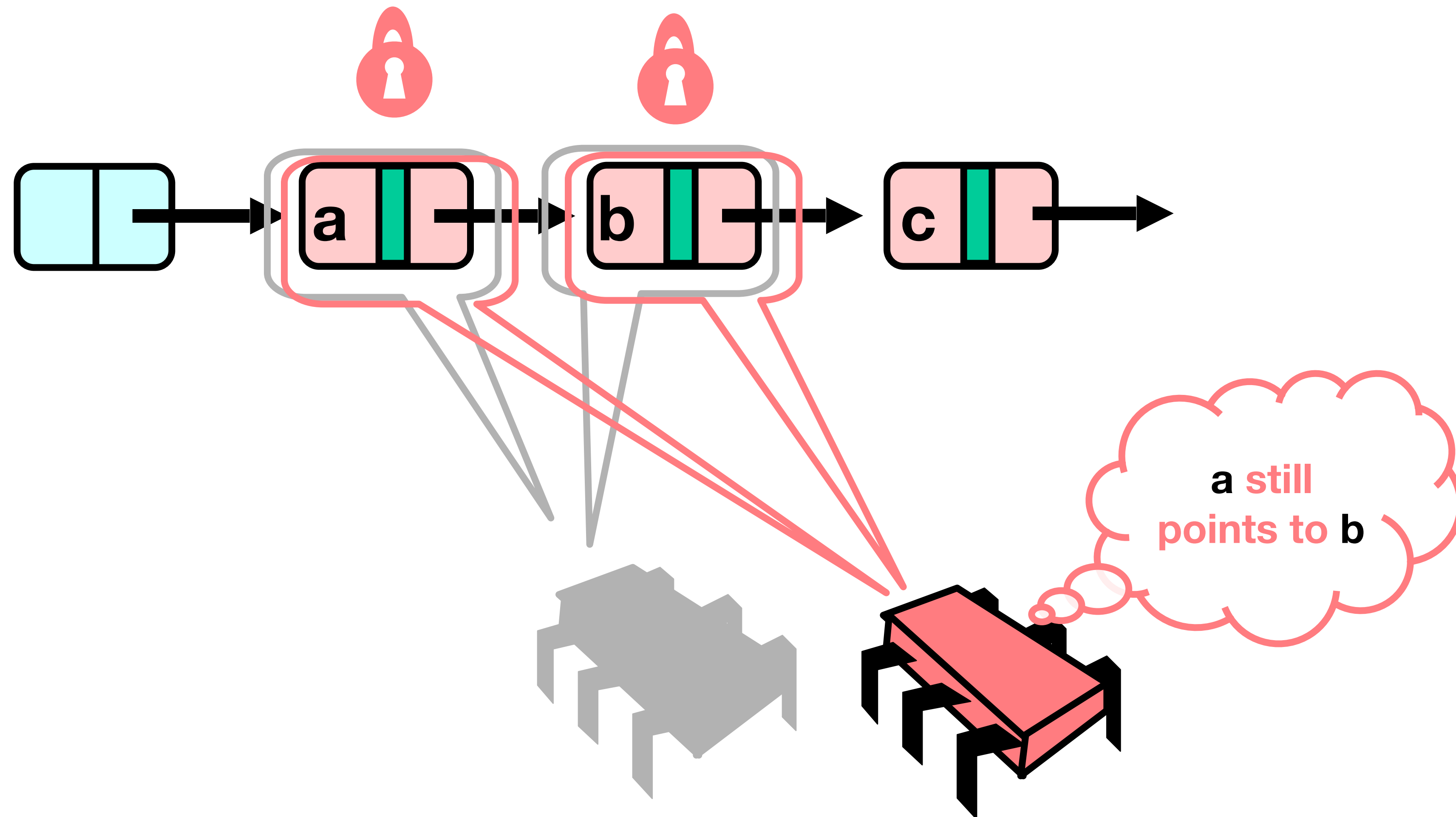
Business as Usual



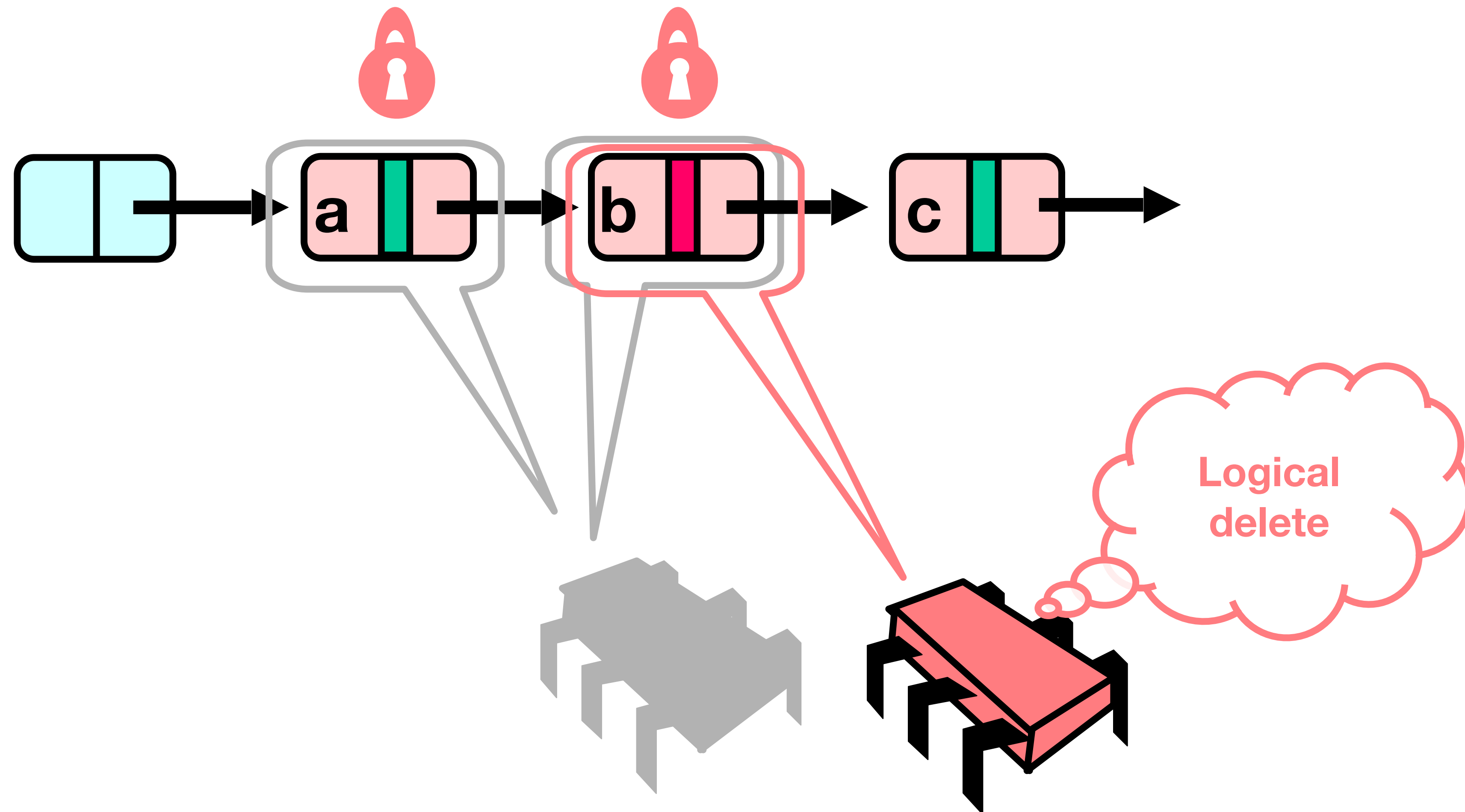
Business as Usual



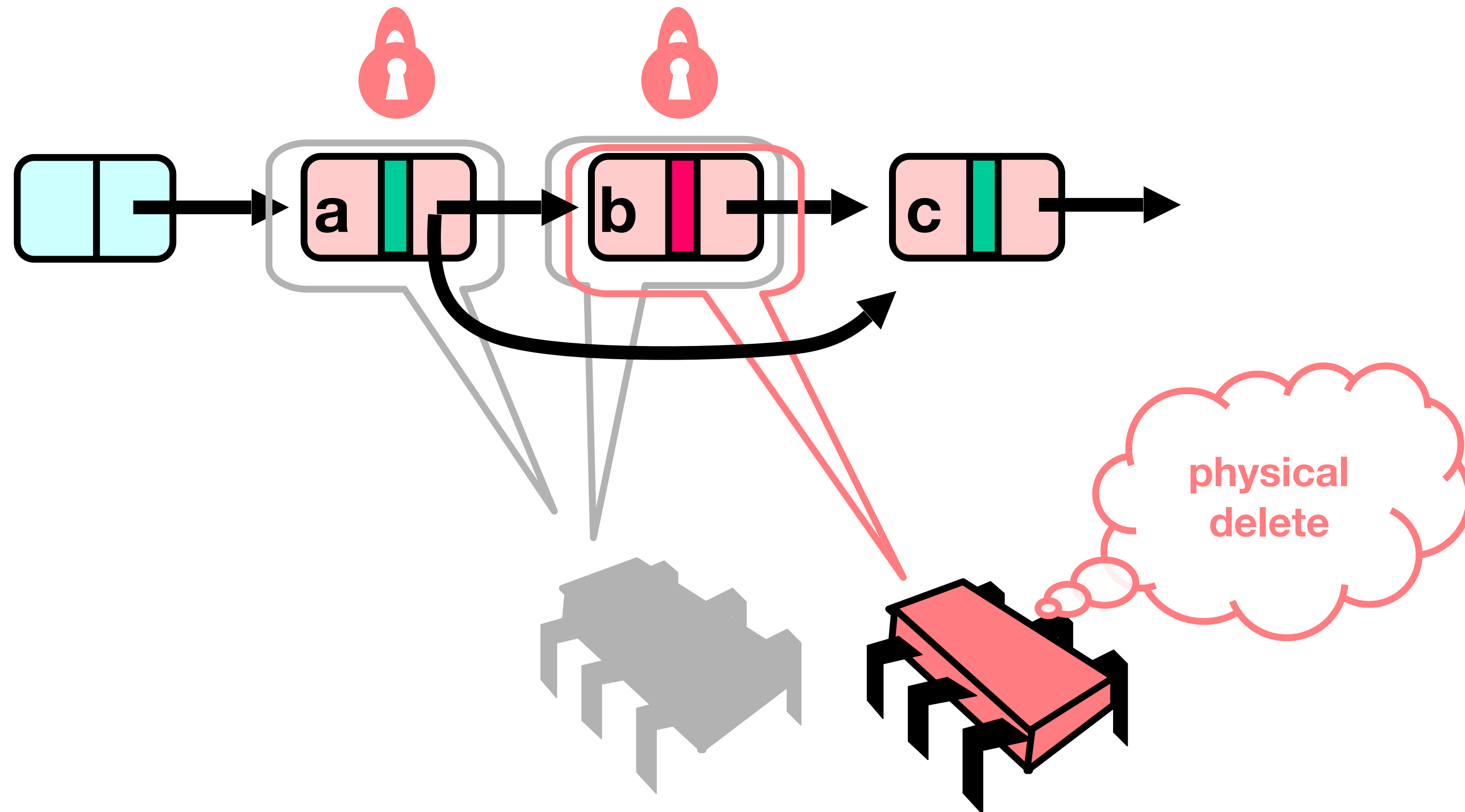
Business as Usual



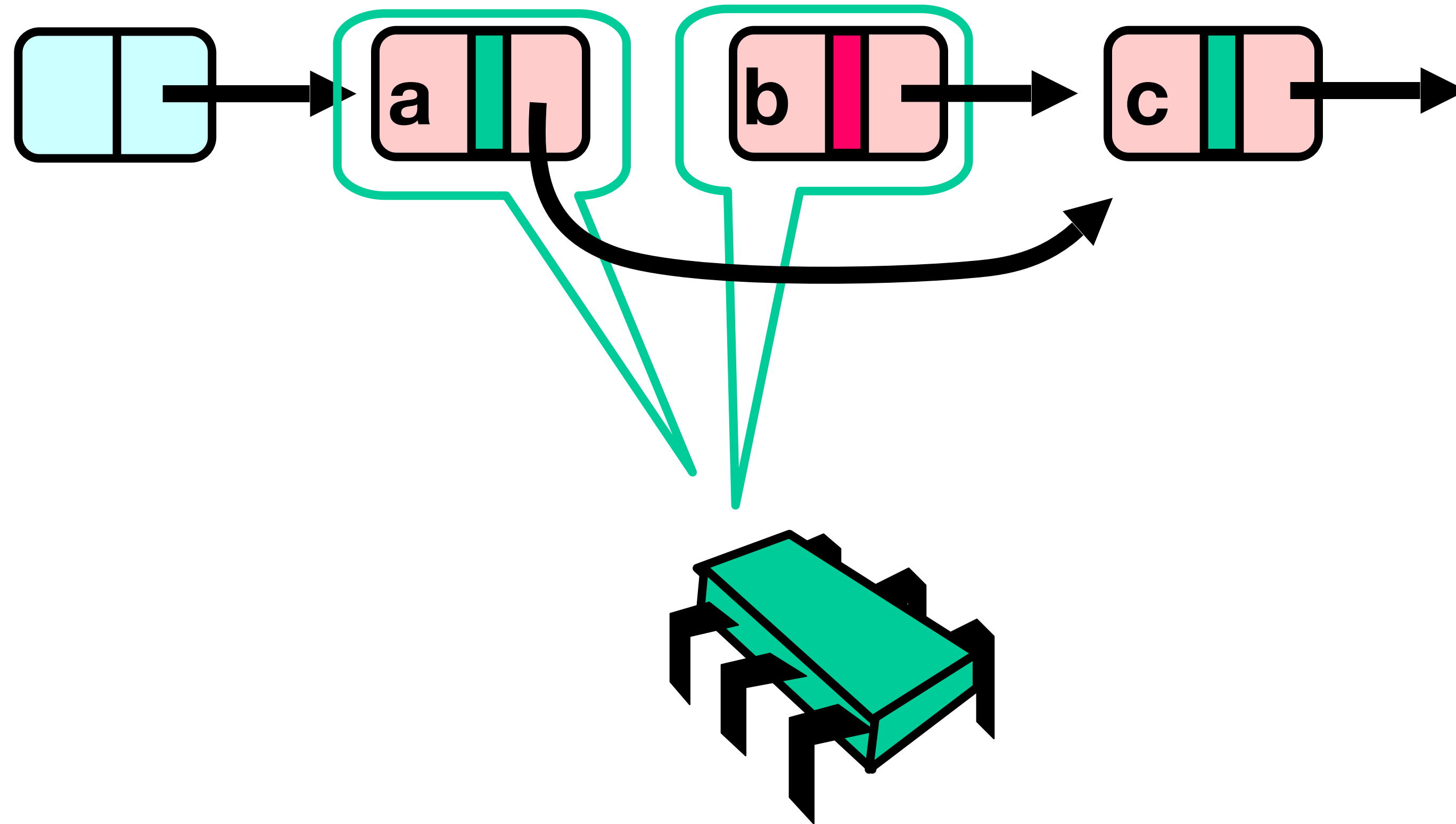
Business as Usual



Business as Usual



Business as Usual



New Abstraction Map

- $S(\text{head}) =$
 - $\{ x \mid \text{there exists node } a \text{ such that}$
 - a **reachable from head** **and**
 - $a.\text{item} = x$ **and**
 - a **is unmarked**
 - $\}$

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**

Remove

```
try {  
  pred.lock(); curr.lock();  
  if (validate(pred,curr) {  
    if (curr.key == key) {  
      curr.marked = true;  
      pred.next = curr.next;  
      return true;  
    } else {  
      return false;  
    }  
  }  
} finally {  
  pred.unlock();  
  curr.unlock();  
}
```

Remove

```
try {  
    pred.lock(); curr.lock();  
    if (validate(pred,curr) {  
        if (curr.key == key) {  
            curr.marked = true;  
            pred.next = curr.next;  
            return true;  
        } else {  
            return false;  
        }  
    }  
} finally {  
    pred.unlock();  
    curr.unlock();  
}
```

Validate as before

Remove

```
try {  
    pred.lock(); curr.lock();  
    if (validate(pred,curr) {  
        if (curr.key == key) {  
            curr.marked = true;  
            pred.next = curr.next;  
            return true;  
        } else {  
            return false;  
        }  
    }  
} finally {  
    pred.unlock();  
    curr.unlock();  
}
```

Key found

Remove

```
try {  
  pred.lock(); curr.lock();  
  if (validate(pred,curr) {  
    if (curr.key == key) {  
      curr.marked = true;  
      pred.next = curr.next;  
      return true;  
    } else {  
      return false;  
    }  
  } finally {  
    pred.unlock();  
    curr.unlock();  
  }  
}
```

Logical remove

Remove

```
try {  
  pred.lock(); curr.lock();  
  if (validate(pred,curr) {  
    if (curr.key == key) {  
      curr.marked = true;  
      pred.next = curr.next;  
      return true;  
    } else {  
      return false;  
    }  
  } finally {  
    pred.unlock();  
    curr.unlock();  
  }  
}
```

physical remove

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;  
}
```

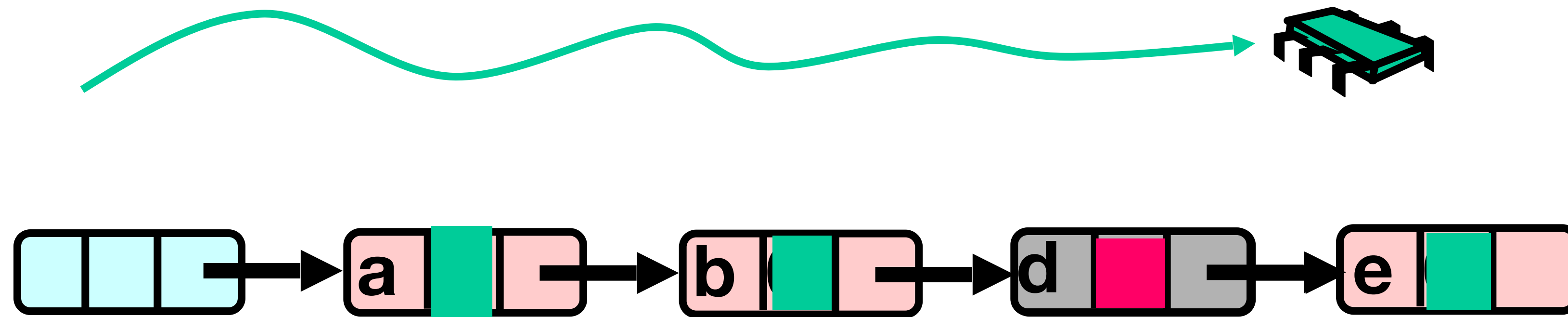
**Traverse without locking
(nodes may have been removed)**

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?

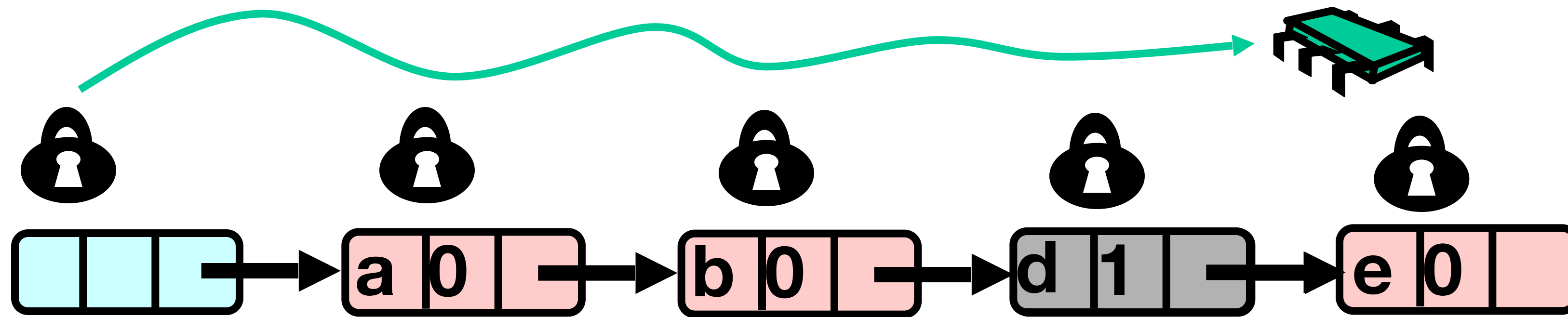
Summary: Wait-free Contains



Use Mark bit + Fact that List is ordered

1. Not marked \rightarrow in the set
2. Marked or missing \rightarrow not in the set

Lazy List



Lazy add() and remove() + Wait-free contains()

Evaluation

- Good:
 - contains() **doesn't lock**
 - In fact, its wait-free!
 - 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 OS scheduler.....
- Next week: wait-free algorithms and data structures!

HW1 Discussion

Go to socrative.com and select “Student Login” Room: CS475; ID is your G-Number

1. How fair do you think this assignment was?
2. How difficult did you think this assignment was?
3. How long did you spend on this assignment?

Reminder: If you are not in class, you may not complete the activity. If you do anyway, this will constitute a violation of the honor code.

HW 2 Discussion

<http://www.jonbell.net/gmu-cs-475-fall-2019/cs475-f19-homework-2/>

This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/>
- You are free to:
 - Share — copy and redistribute the material in any medium or format
 - Adapt — remix, transform, and build upon the material
 - for any purpose, even commercially.
- Under the following terms:
 - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
 - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
 - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.