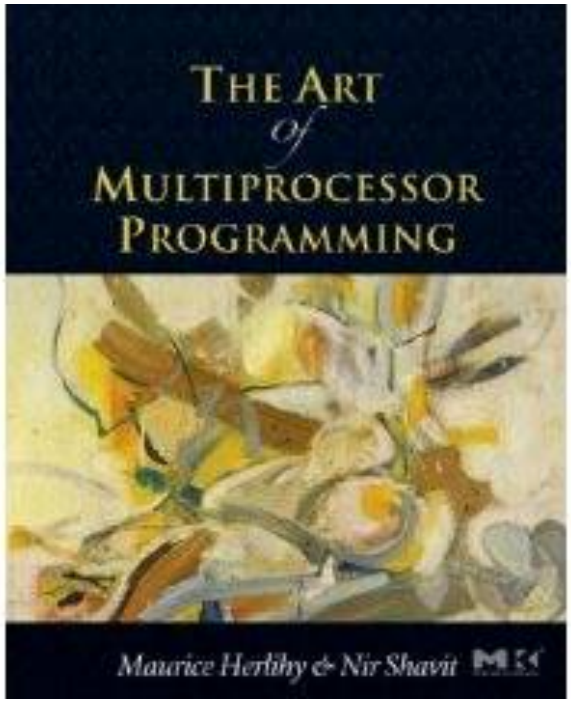
Concurrent Linked Lists

Acknowledgement:
Slides adopted from the companion slides for the book
"The Art of Multiprocessor Programming"
by Maurice Herlihy and Nir Shavit

What We'll Cover Today

Chapter 9 of:



Digital copy can be obtained via WUSTL library:

http://catalog.wustl.edu/search/

Today: Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by Queue locks
- Should increase throughput
  - Not possible if inherently sequential
  - Surprising things are parallelizable

Coarse-Grained Synchronization: the Good

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

Coarse-Grained Synchronization: the Bad

- Sequential bottleneck
  - Threads “stand in line”
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse

This Lecture

- Introduce four "patterns"
  - Bag of tricks …
  - Methods that work more than once …
- For highly-concurrent objects
  - Concurrent access
  - More threads, more throughput

This Lecture

- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization
- Lazy synchronization
- Lock-free synchronization

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, but
  - 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

Linked List

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

Set Interface

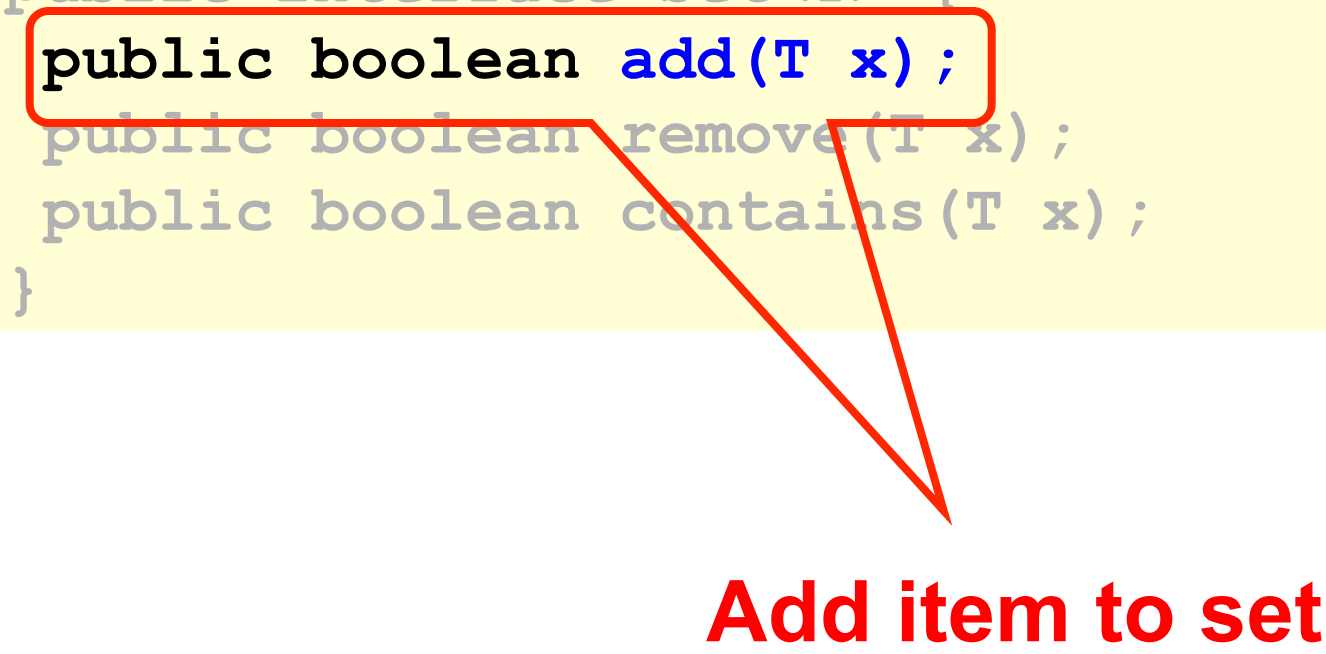
- Unordered collection of items
- No duplicates
- Methods
  - **`add(x)`** put **`x`** in set
  - **`remove(x)`** take **`x`** out of set
  - **`contains(x)`** tests if **`x`** in set

List-Based Sets

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

List-Based Sets

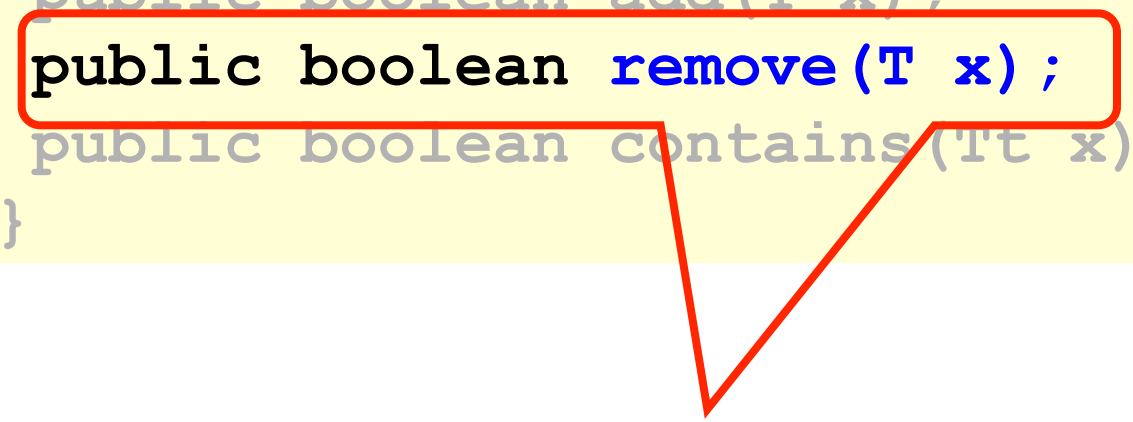
```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```



Add item to set

List-Based Sets

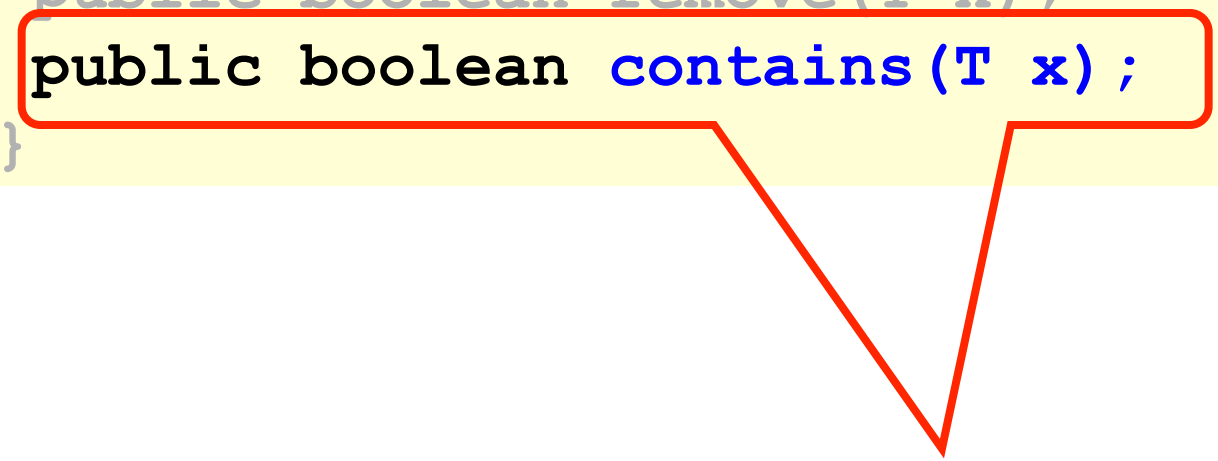
```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```



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);
}
```



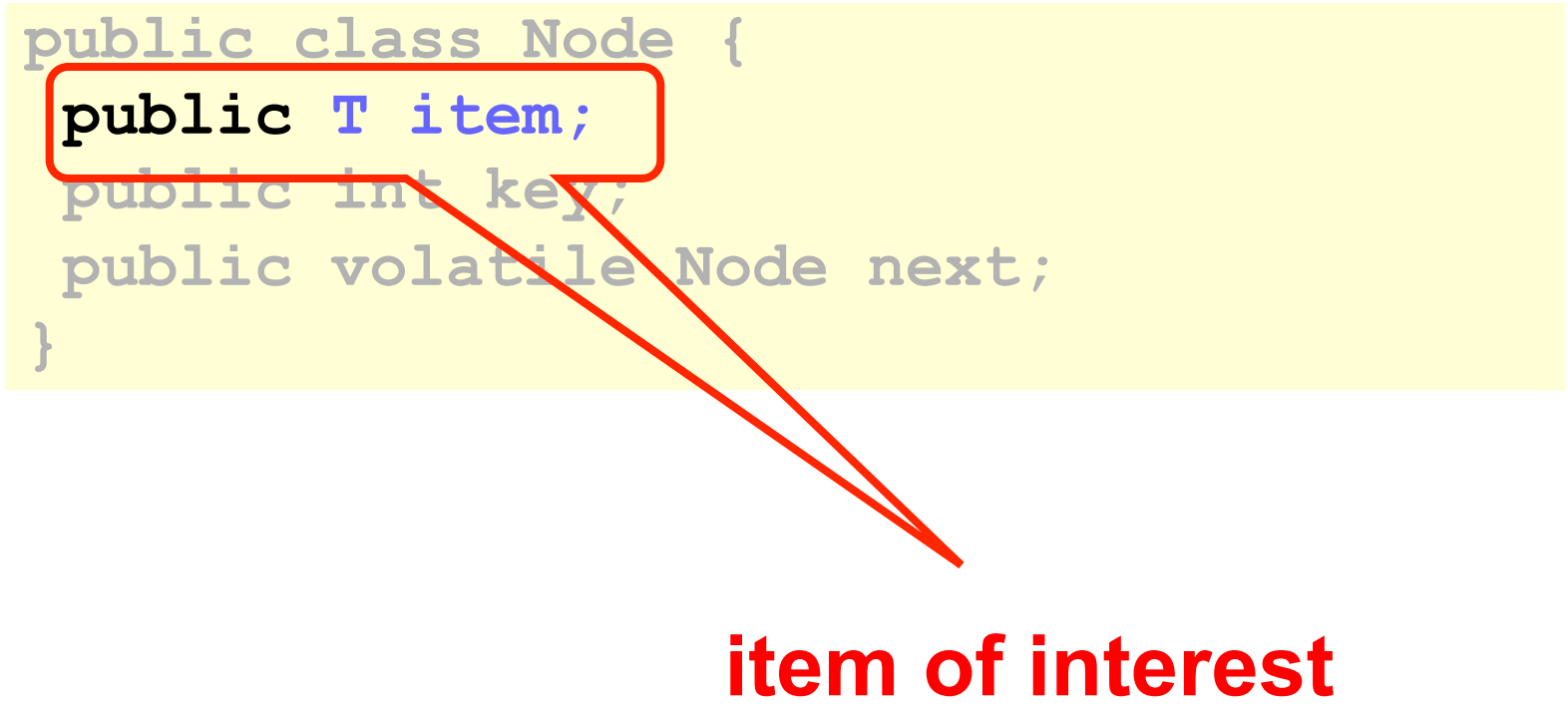
Is item in set?

List Node

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

List Node

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



List Node

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

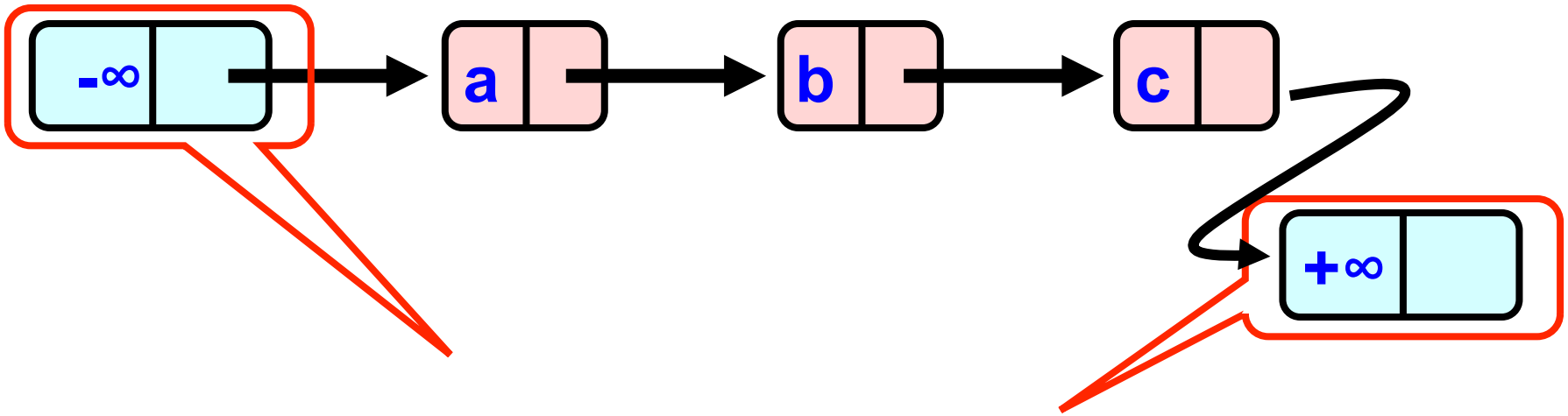
Usually hash code

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)

Once you find a key larger than the key you are searching for, you are done.

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

Specifically …

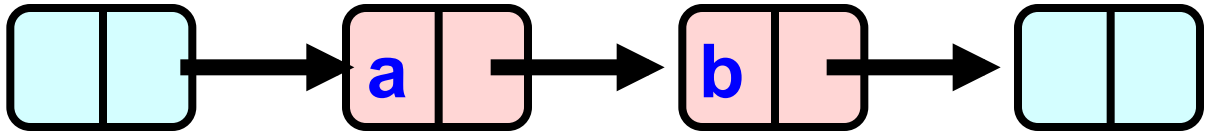
- Invariants preserved by
  - **`add()`**
  - **`remove()`**
  - **`contains()`**
- Most steps are trivial
  - Usually one step tricky
  - Often linearization point

Interference

- Invariants make sense only if we assume **freedom from interference:** methods considered are the only modifiers to the data structure.
- Language encapsulation helps
  - List nodes not visible outside class
- Freedom from interference needed even for removed nodes
  - Some algorithms traverse removed nodes
  - Careful with **`malloc()`** & **`free()`!**
- We rely on garbage collection

Abstract Data Types

- Concrete representation:

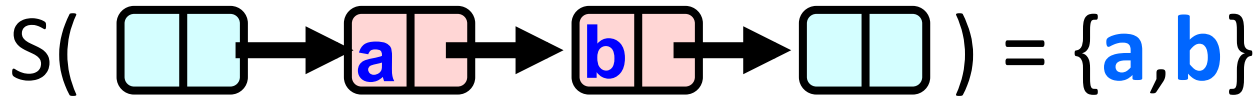


- Abstract Type:

  {a, b}

Abstract Data Types

- Meaning of representation given by ***abstraction map***, carrying lists that satisfy representation invariant to set.



Representation Invariant

- Which concrete values meaningful?
  - Sorted?
  - Duplicates?
- Rep invariant
  - Characterizes legal concrete reps
  - Preserved by methods
  - Relied on by methods

Representation Invariant

- Sentinel nodes
  - tail reachable from head
- Sorted
- No duplicates

Abstraction Map

- S(head) =

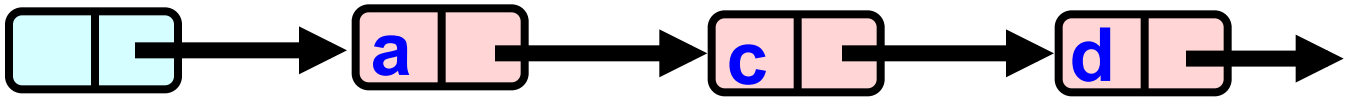
  { x | there exists a such that

    - a reachable from head and
    - a.item = x

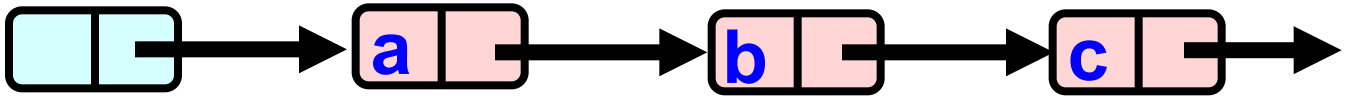
  }

Sequential List Based Set

`add()`

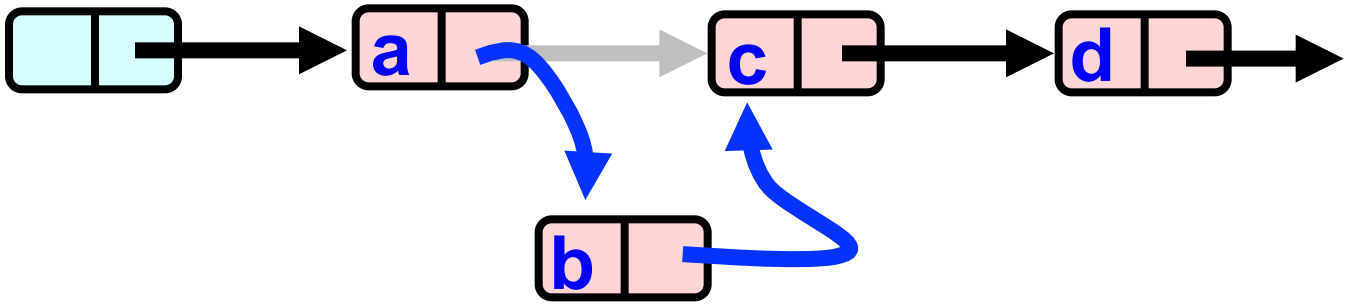


`remove()`

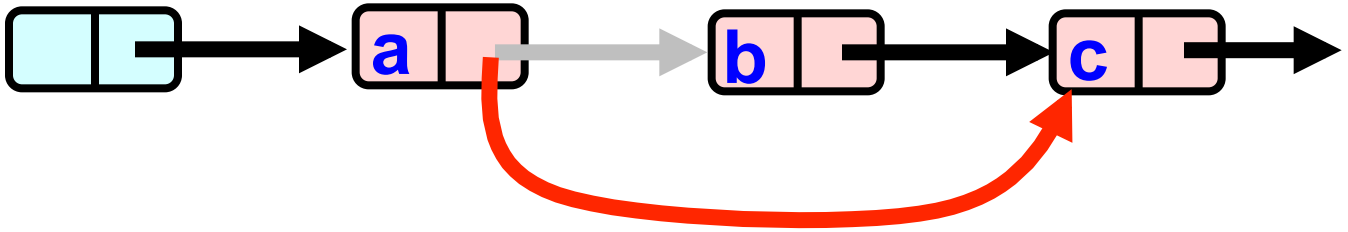


Sequential List Based Set

add()



remove()



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

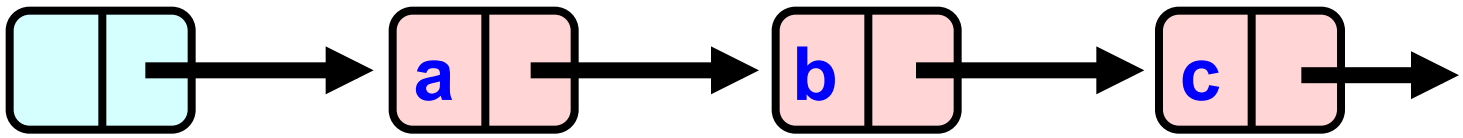
Fine-grained Locking

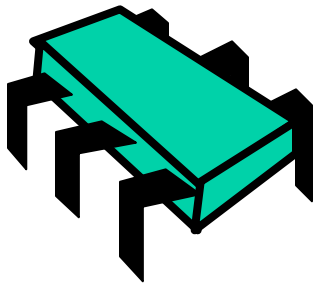
- Requires **careful** thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"

Fine-grained Locking

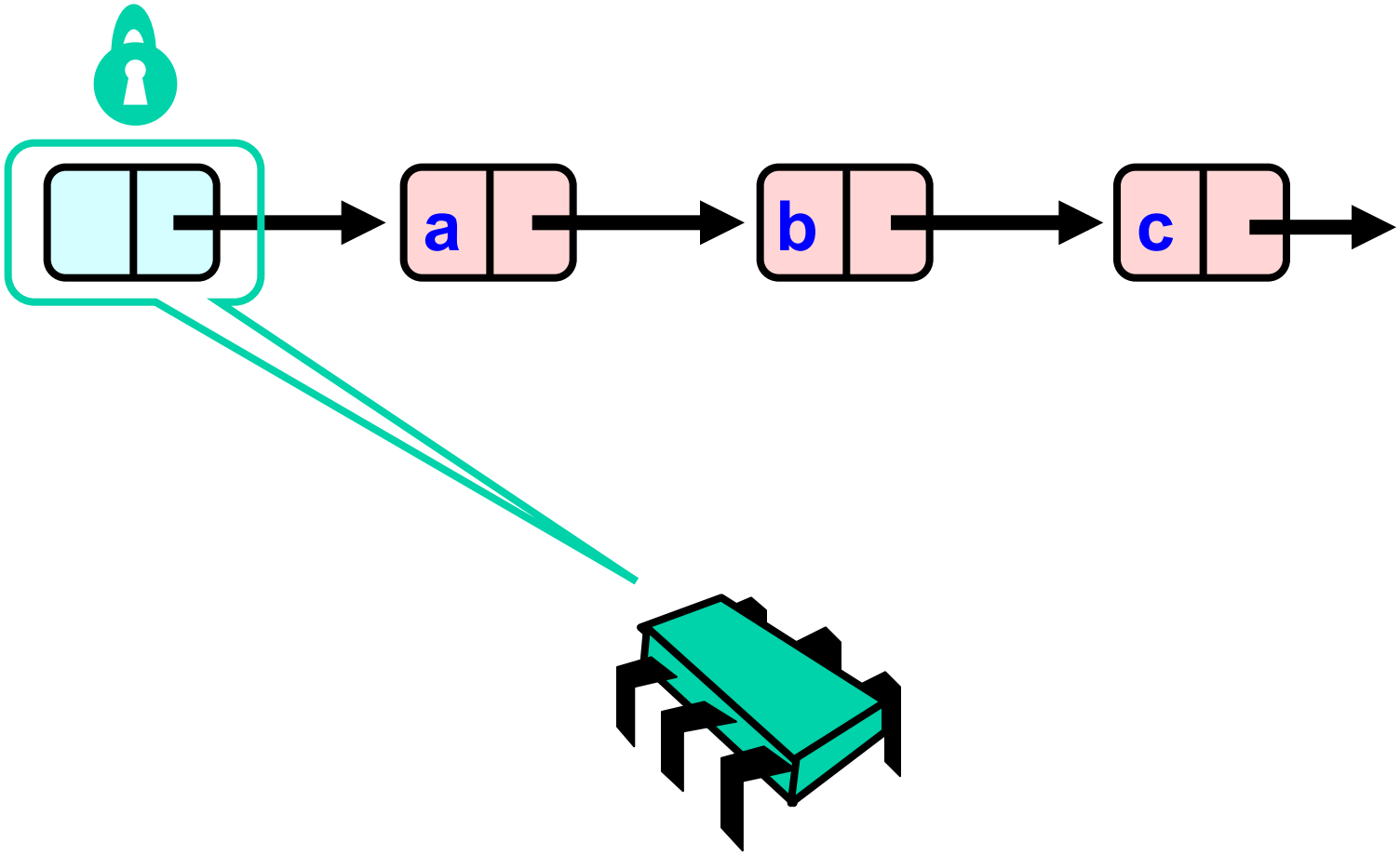
- Requires **careful** thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand locking

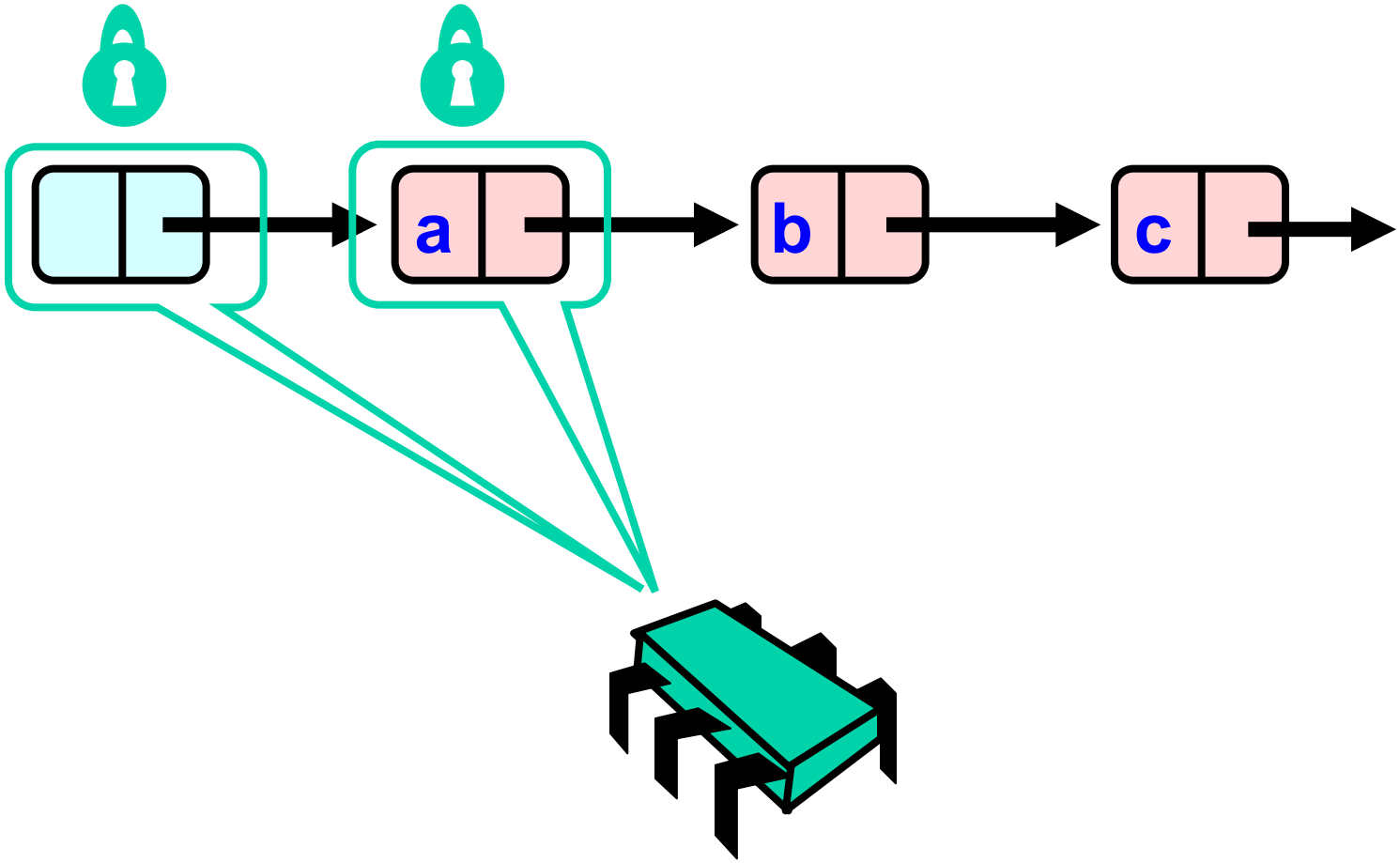




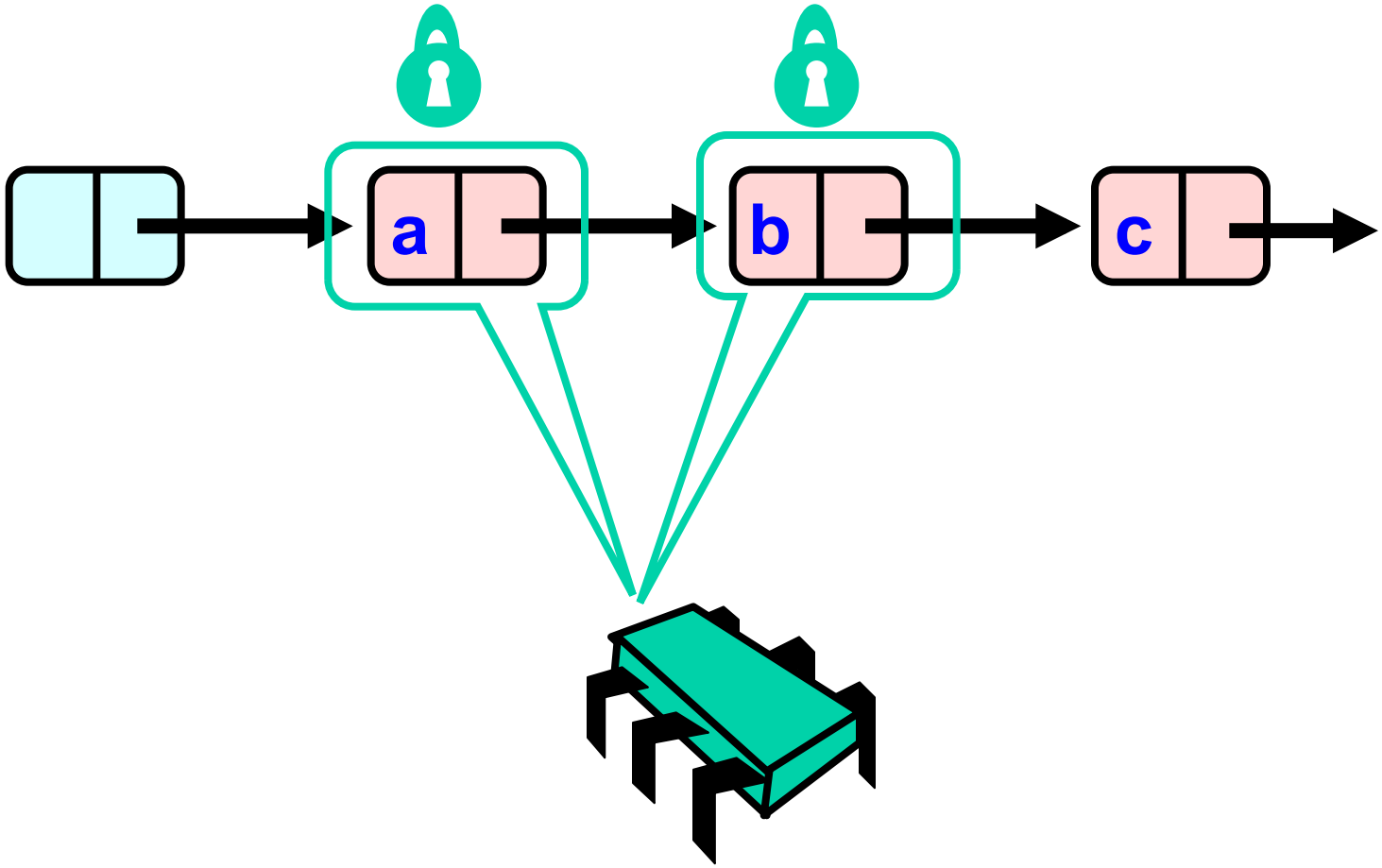
Hand-over-Hand locking



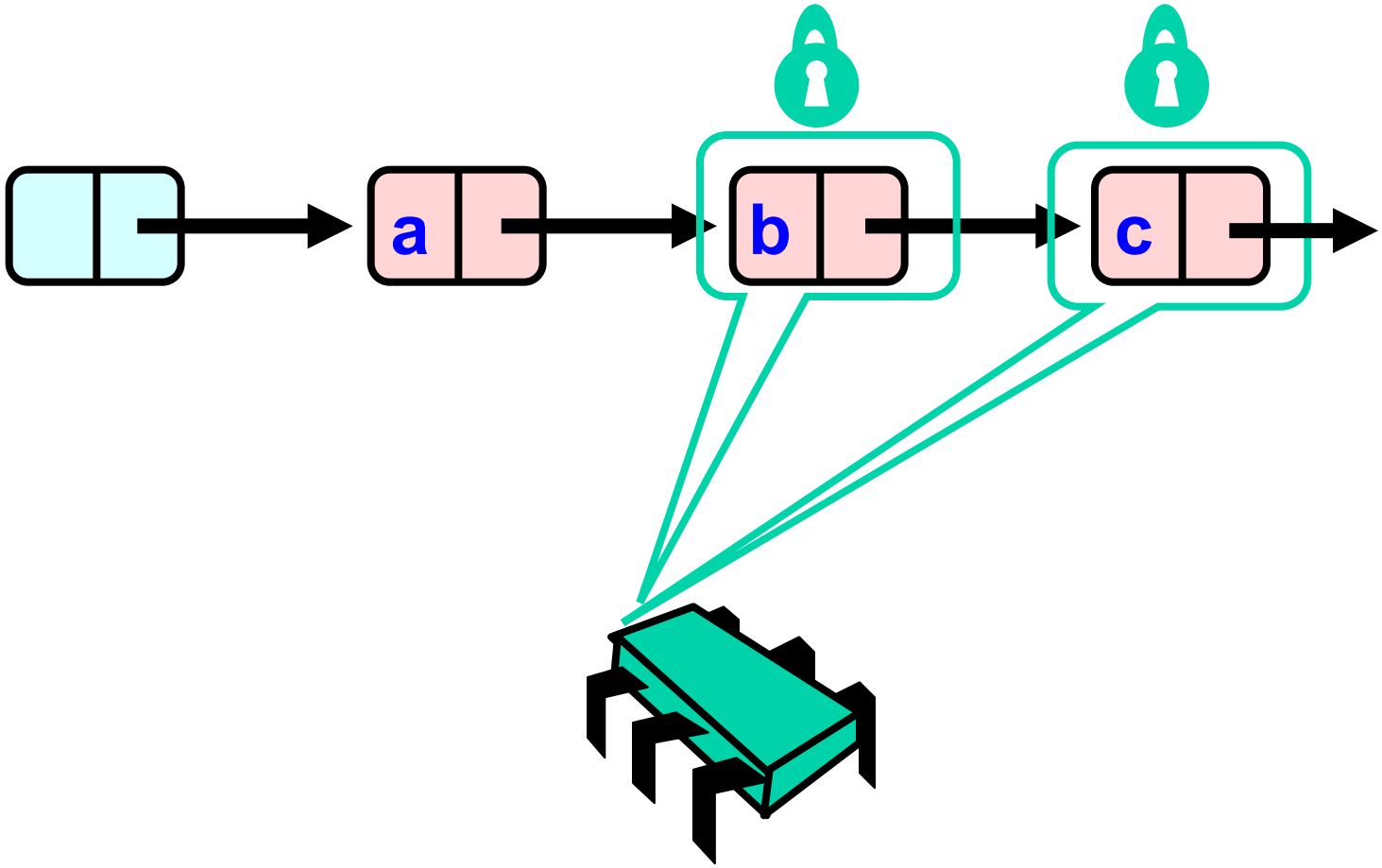
Hand-over-Hand locking



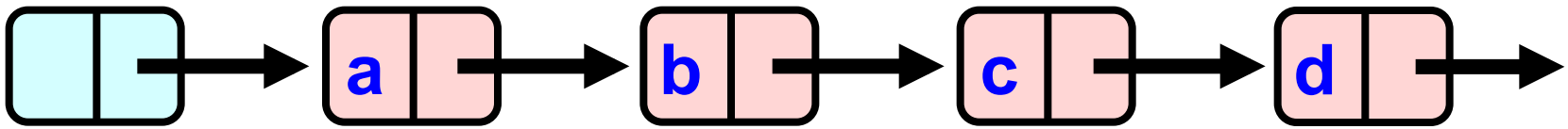
Hand-over-Hand locking

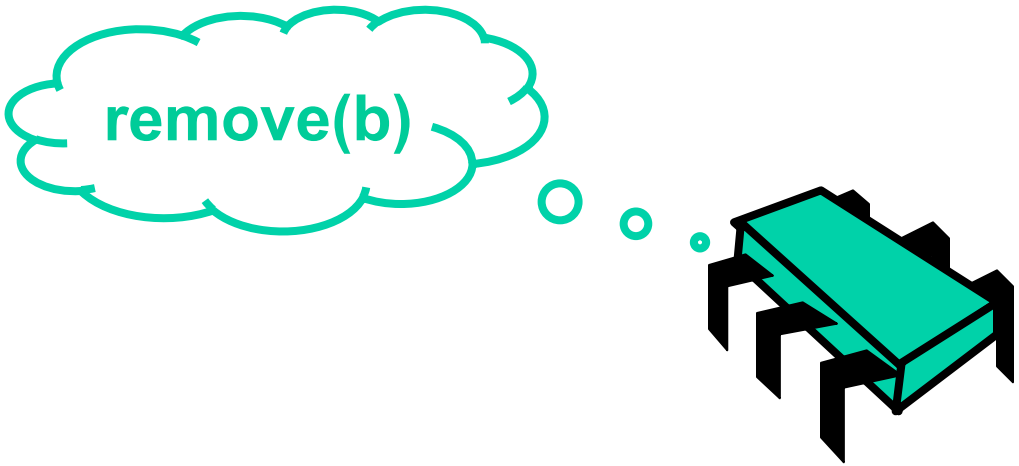


Hand-over-Hand locking

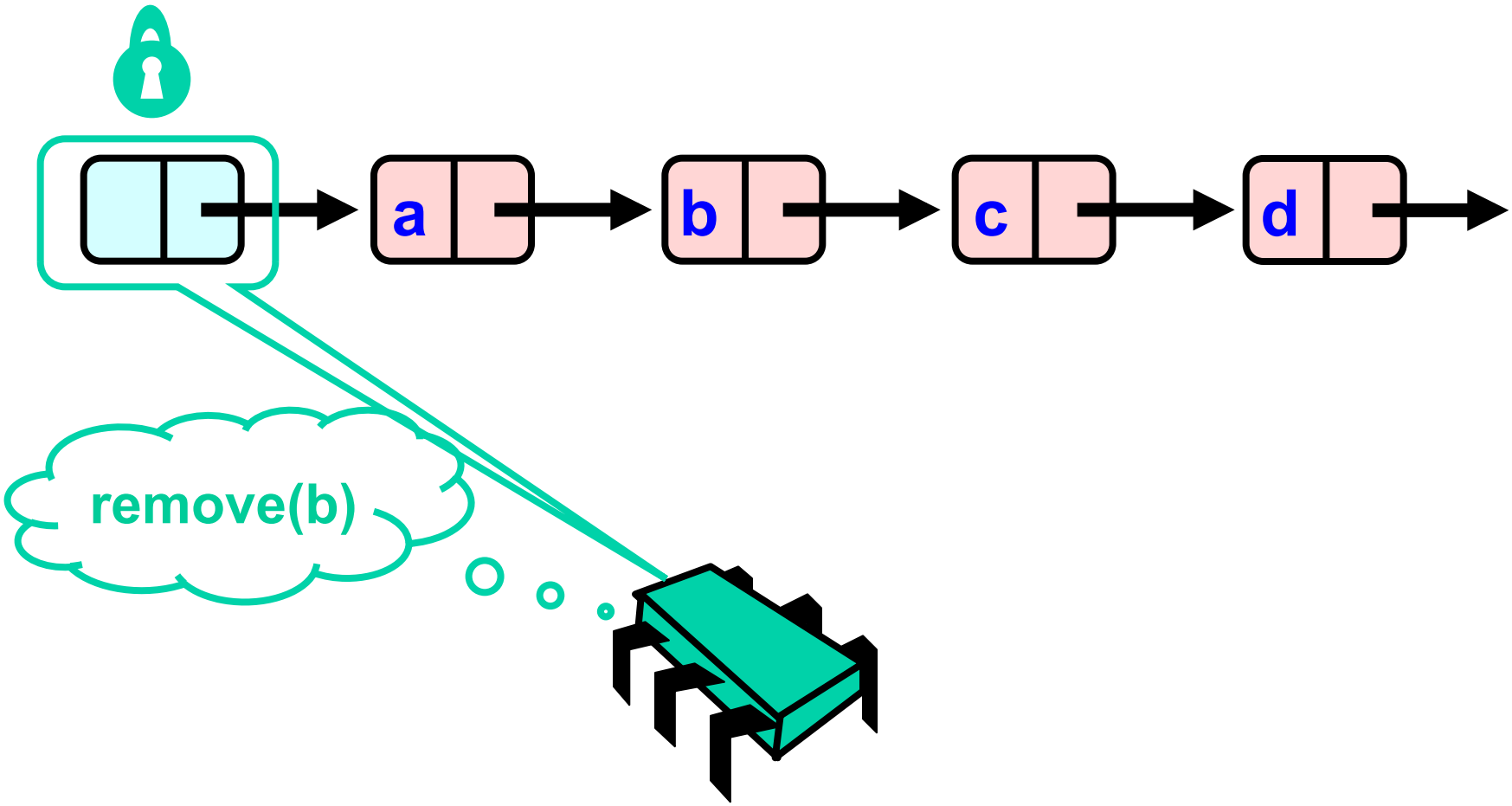


Removing a Node

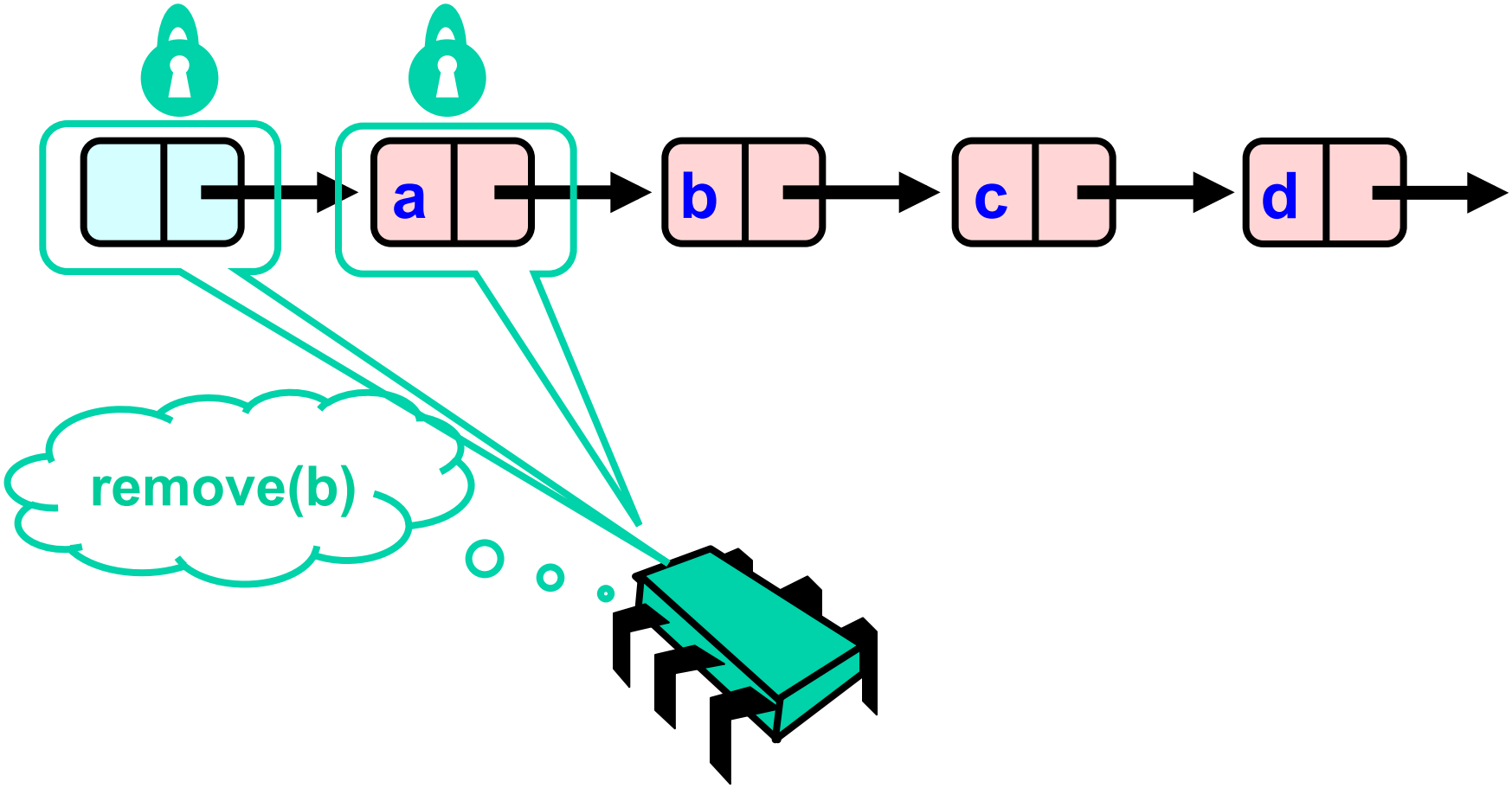




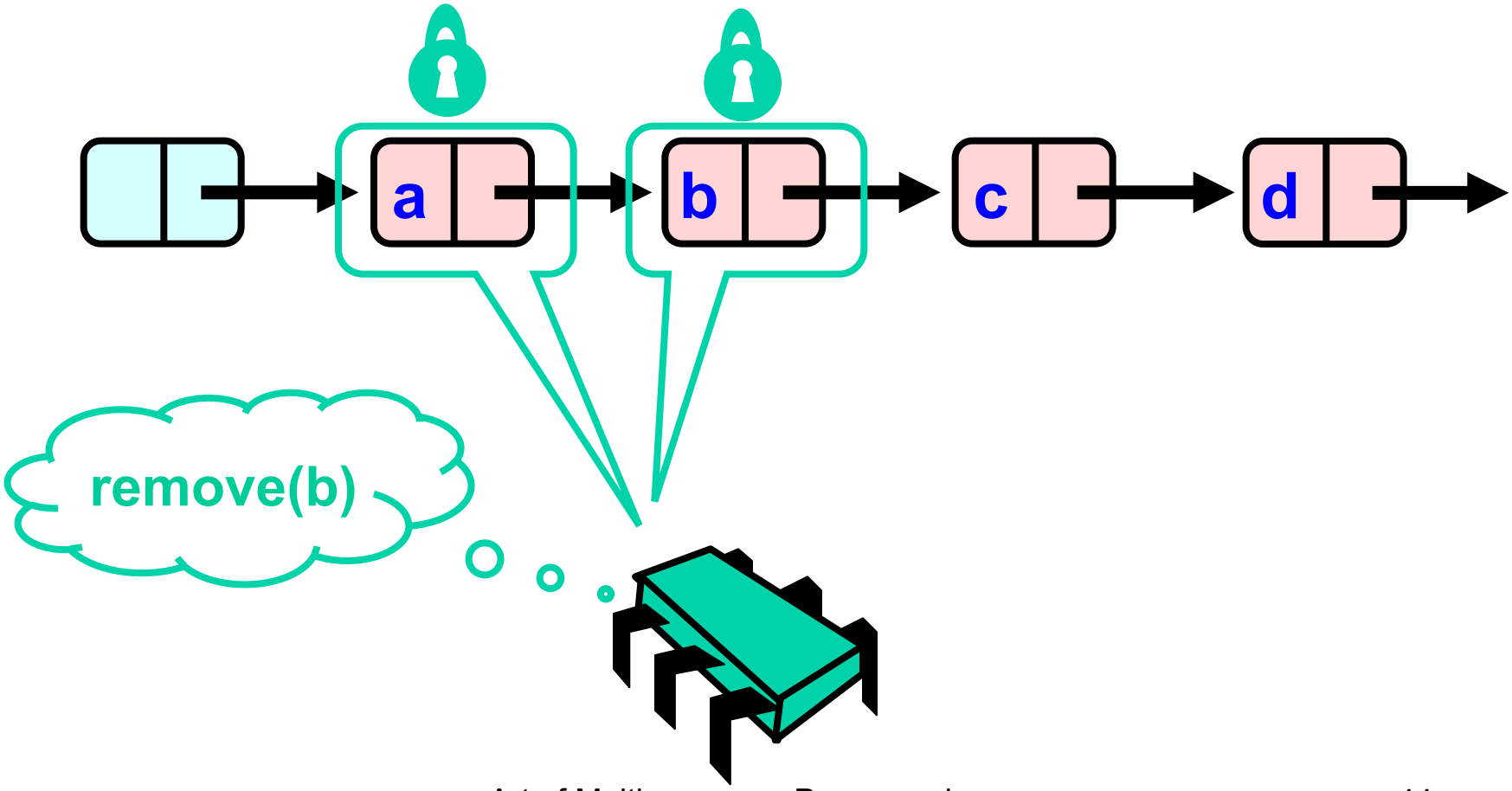
Removing a Node



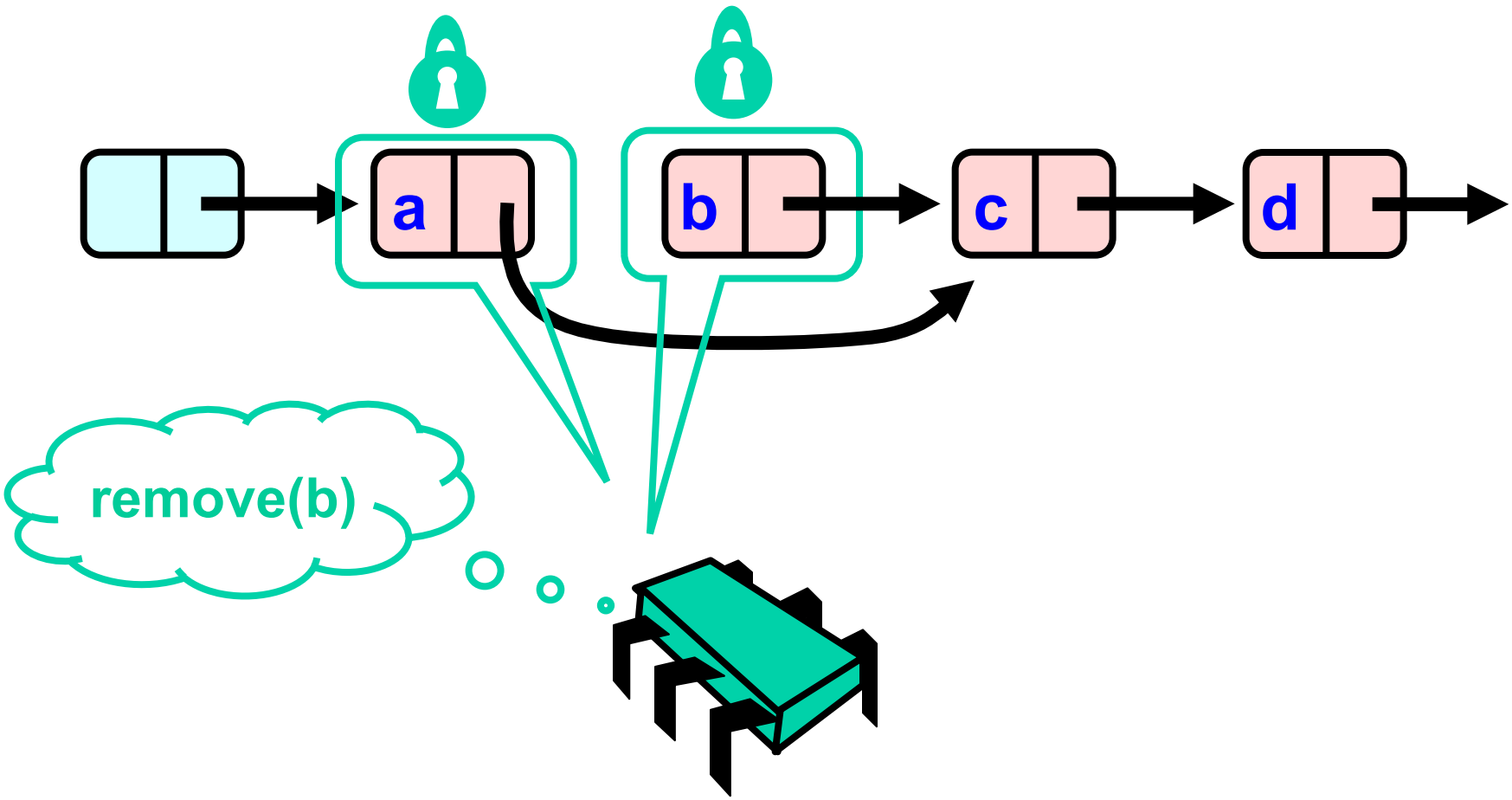
Removing a Node



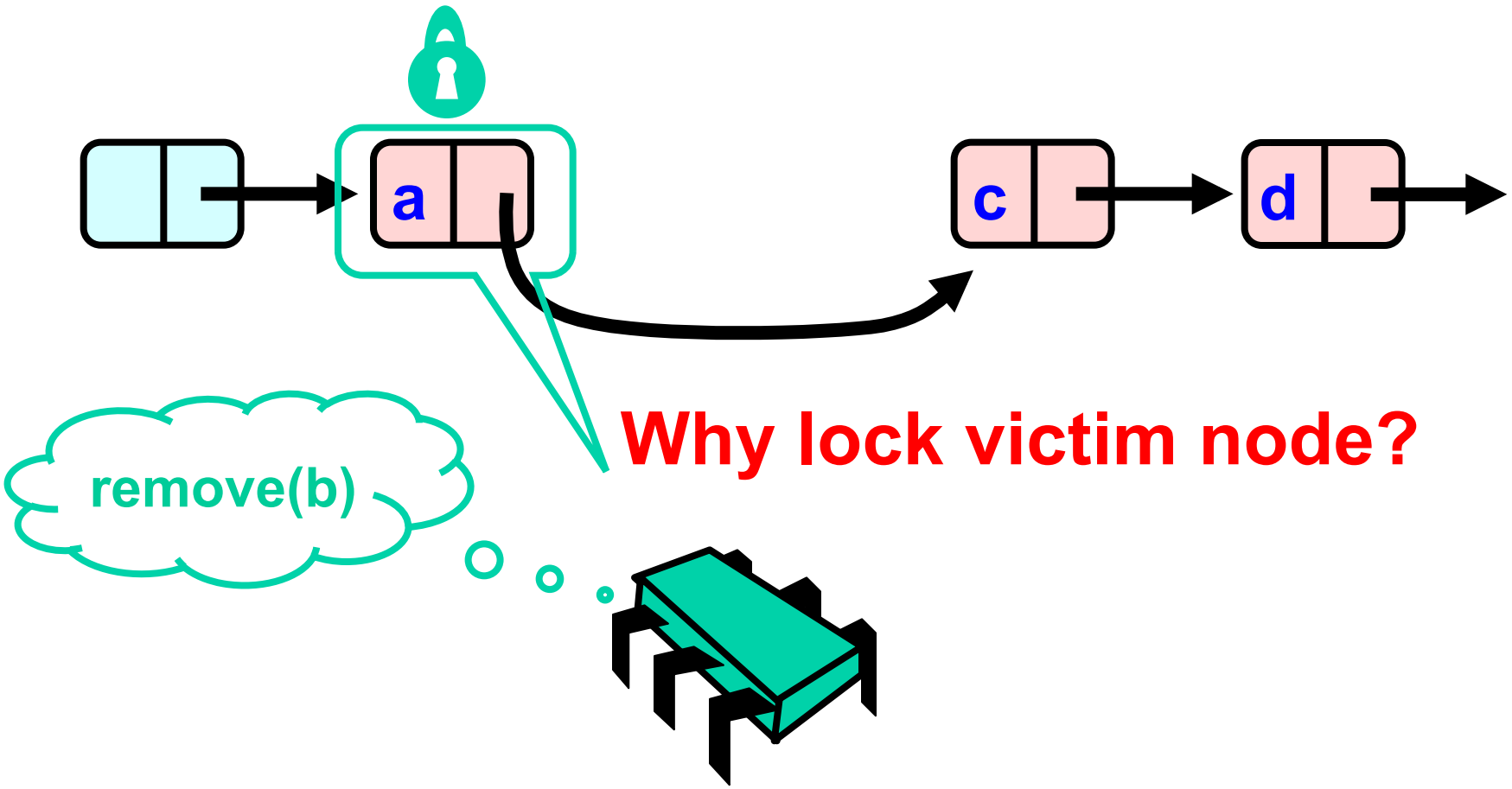
Removing a Node



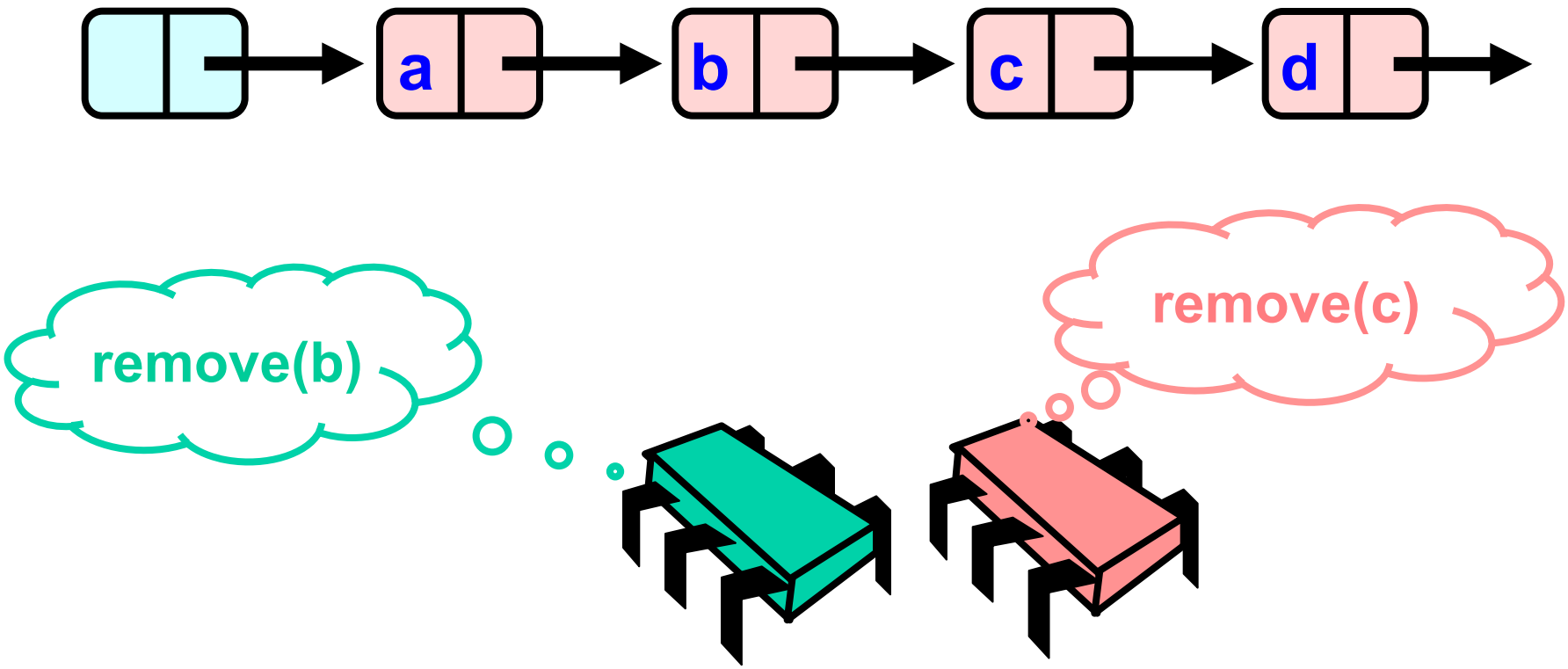
Removing a Node



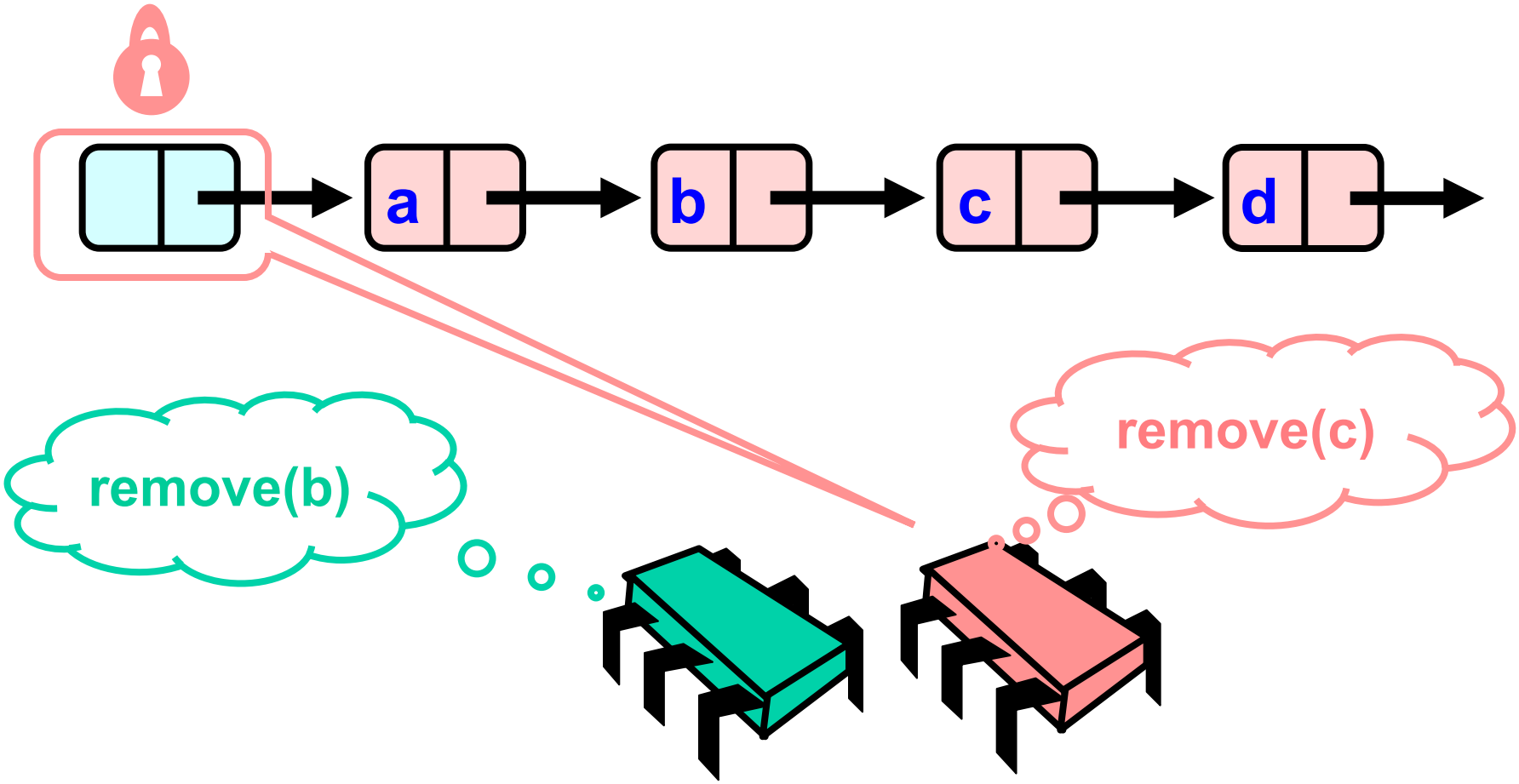
Removing a Node



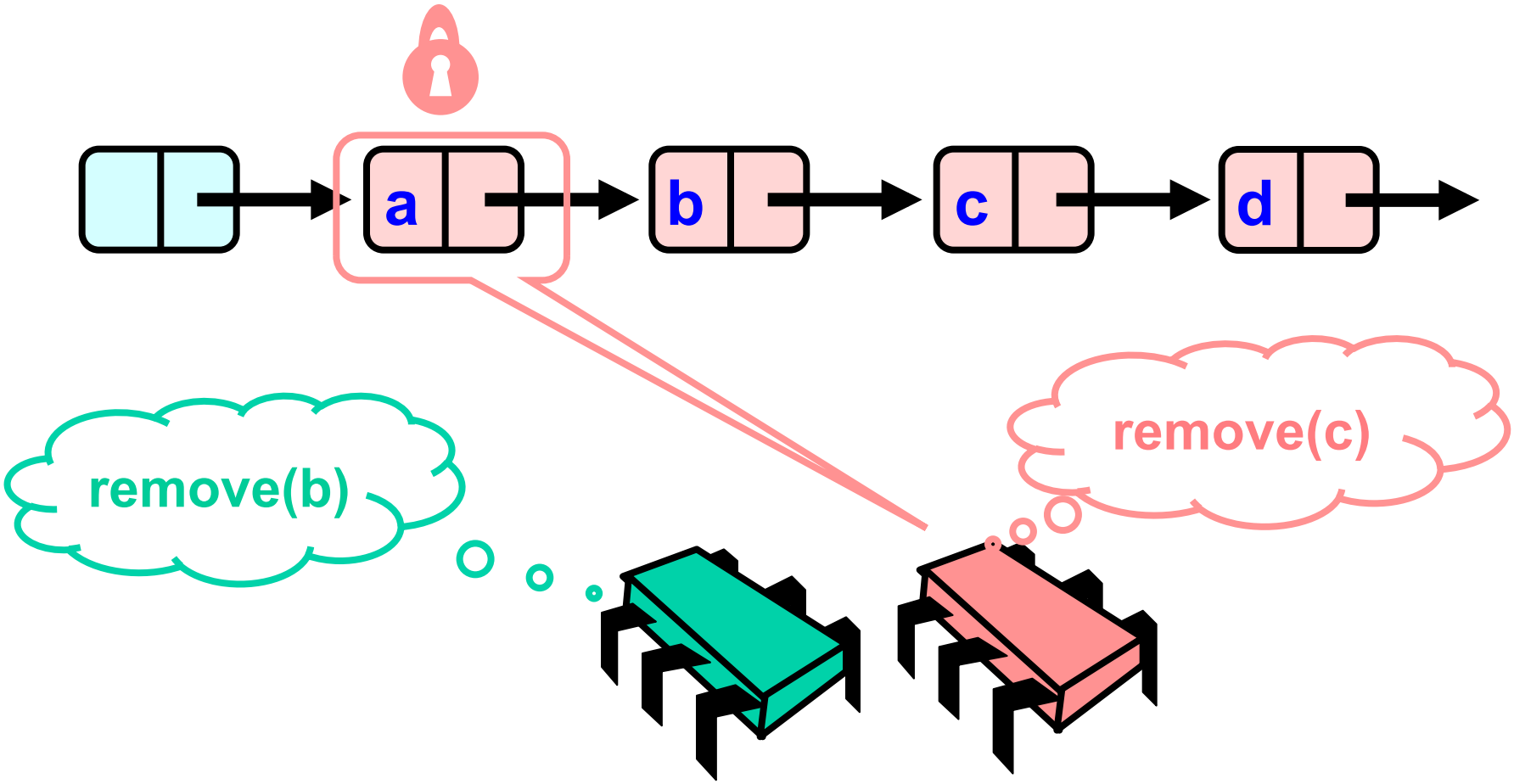
Concurrent Removes



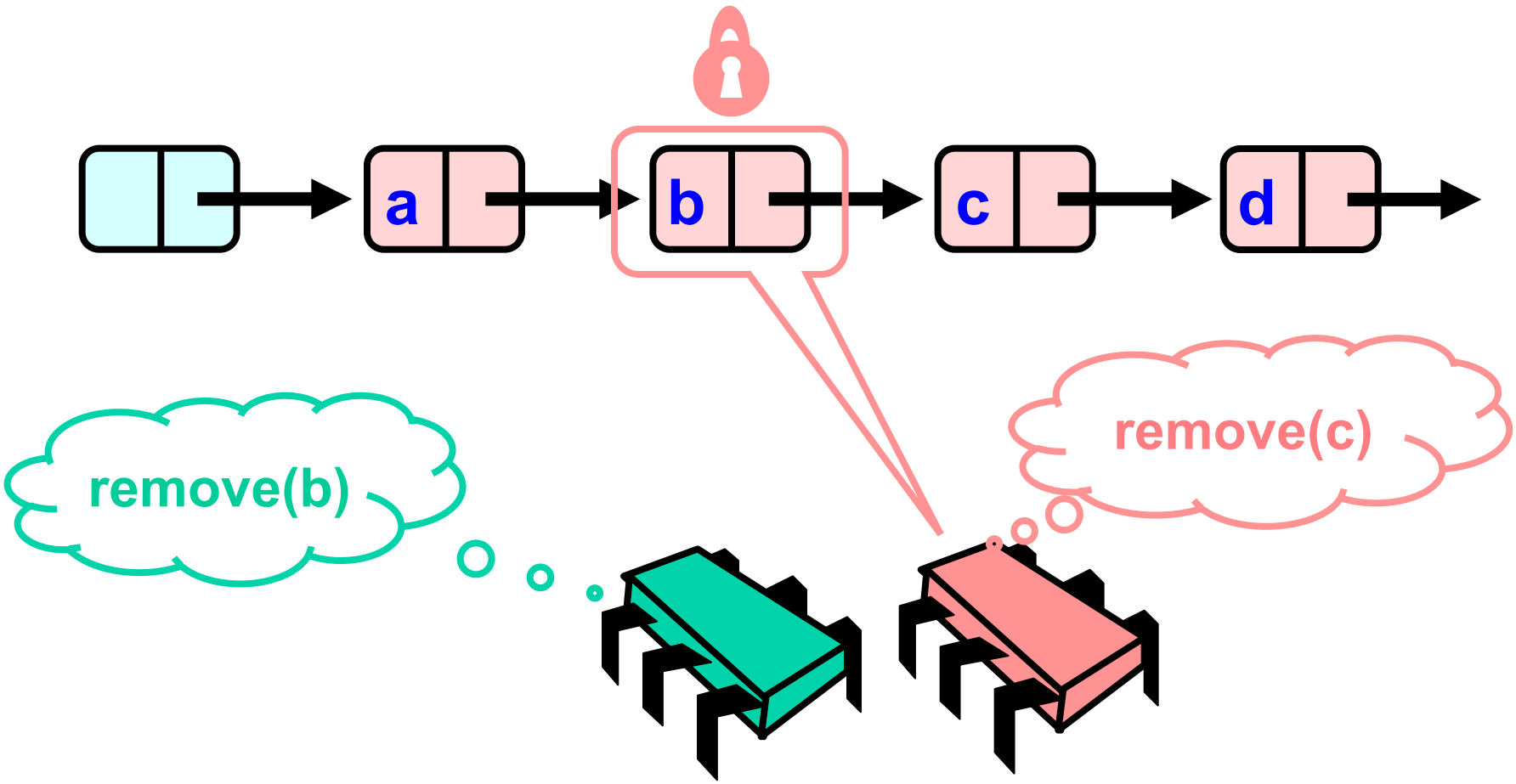
Concurrent Removes



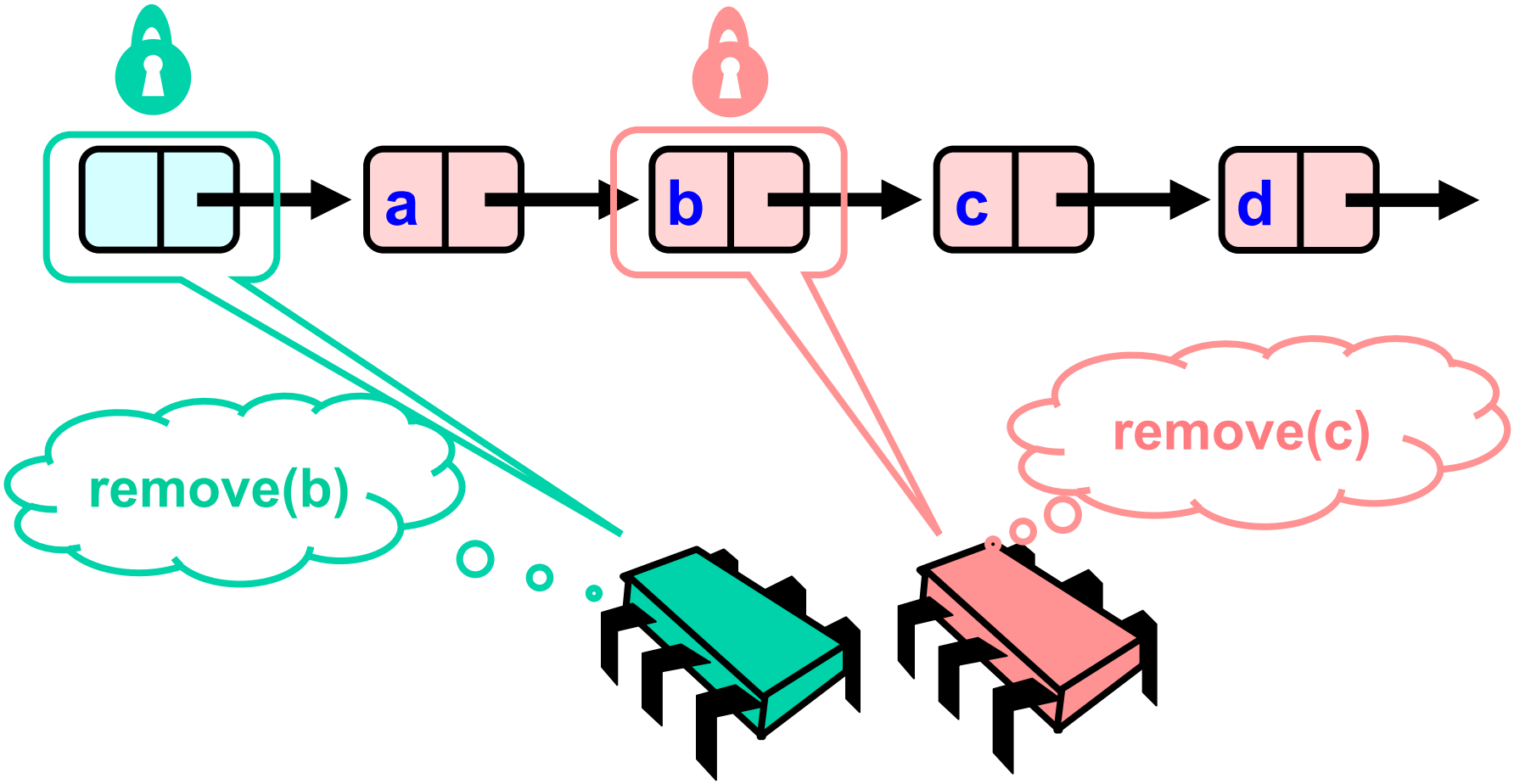
Concurrent Removes



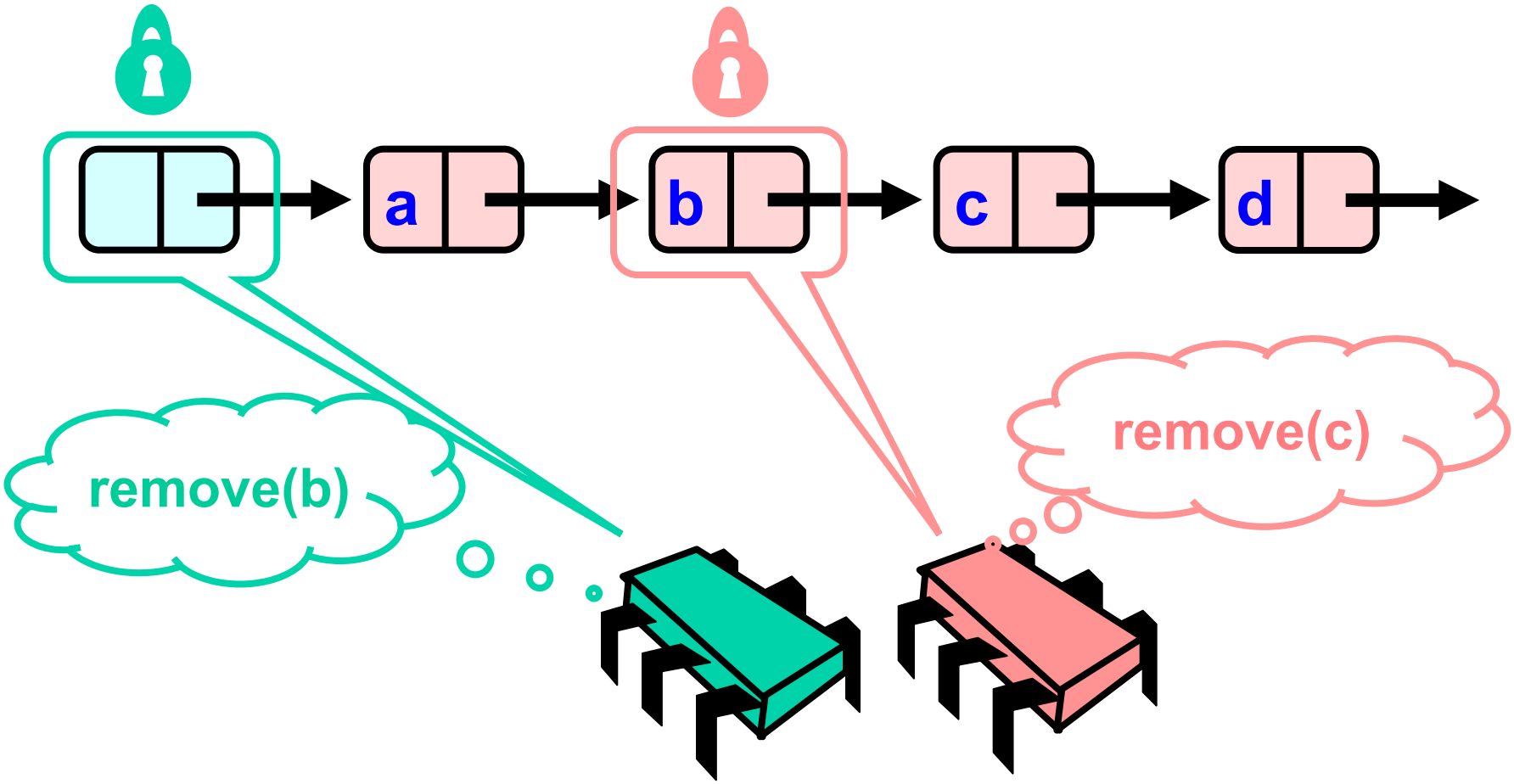
Concurrent Removes



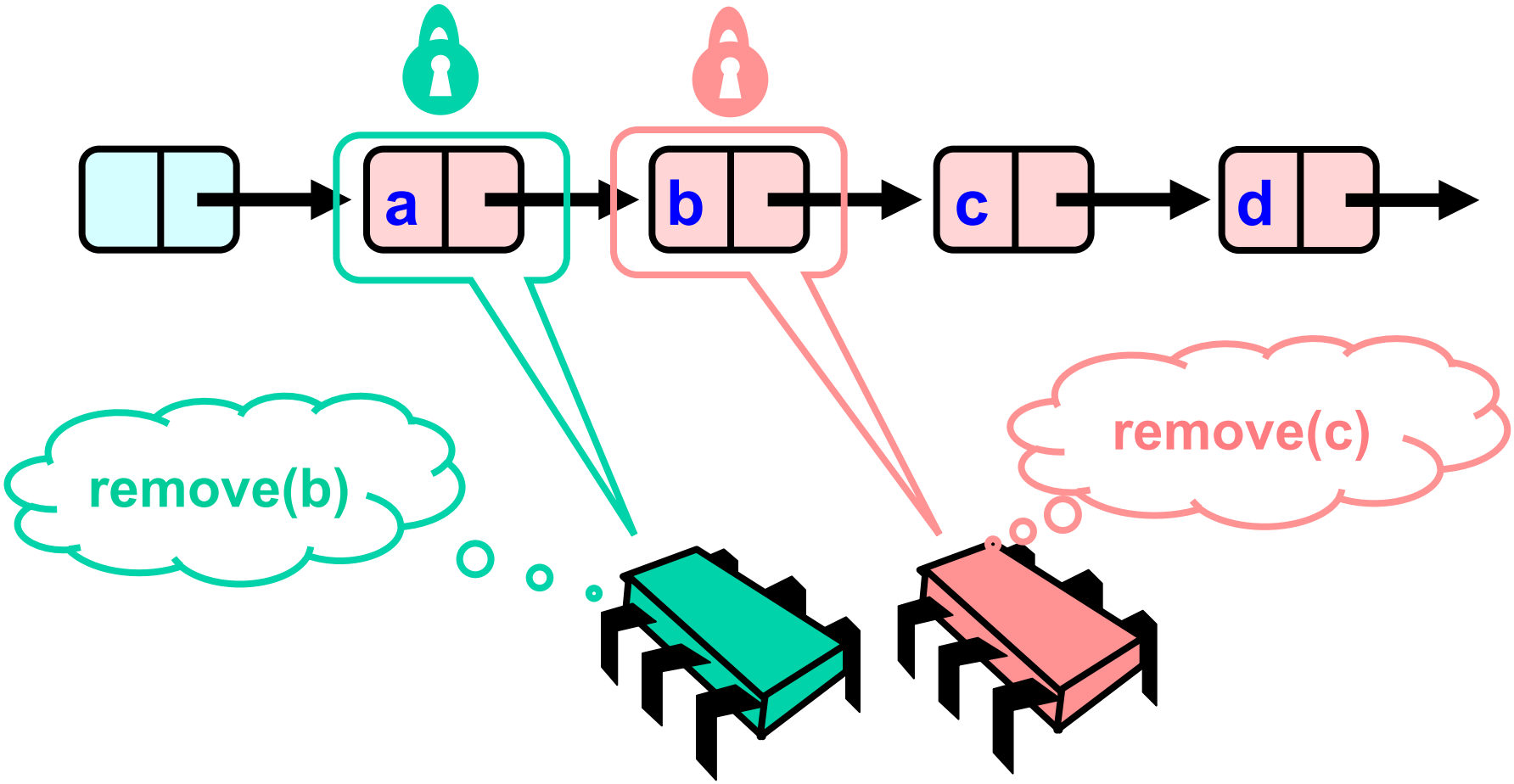
Concurrent Removes



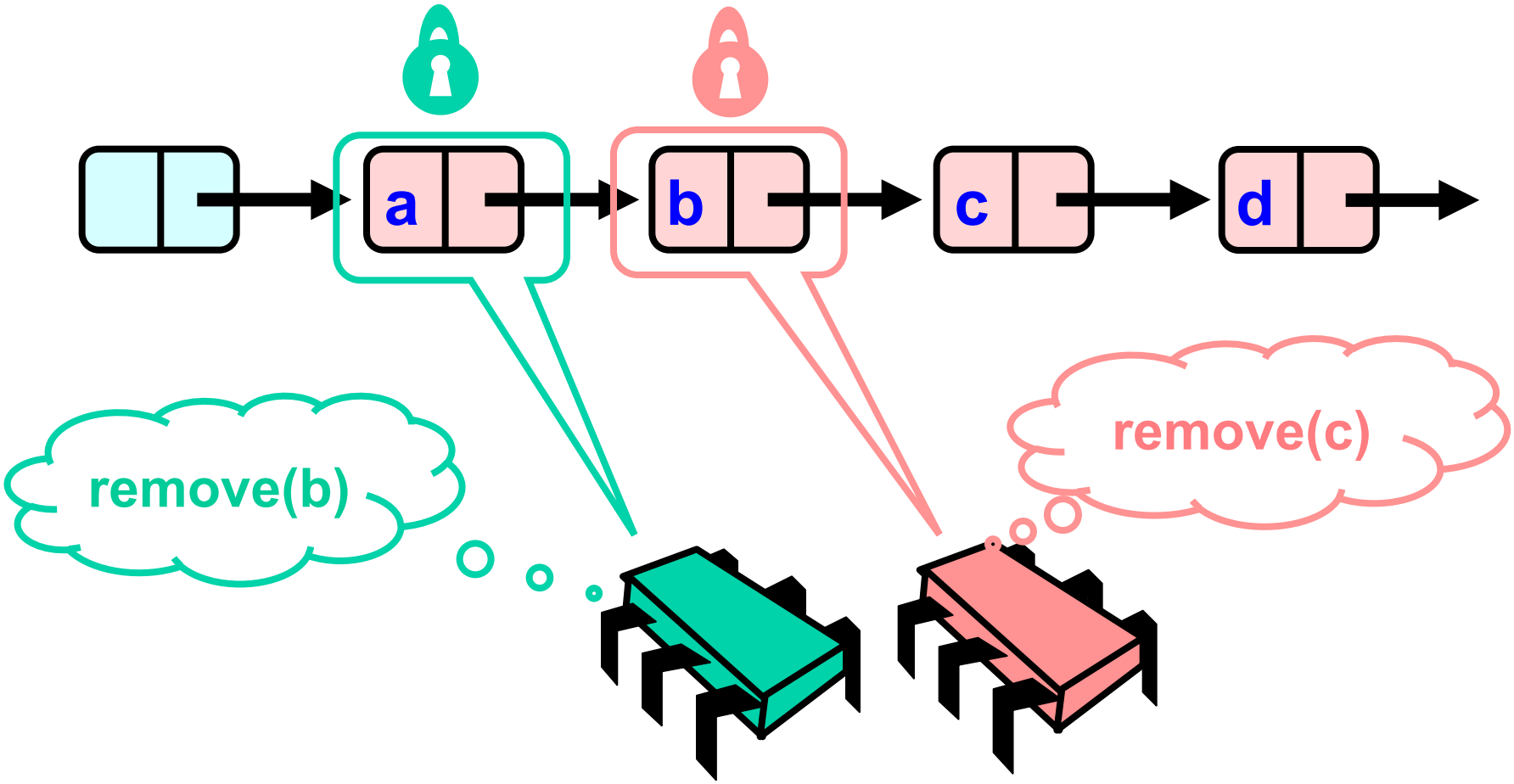
Concurrent Removes



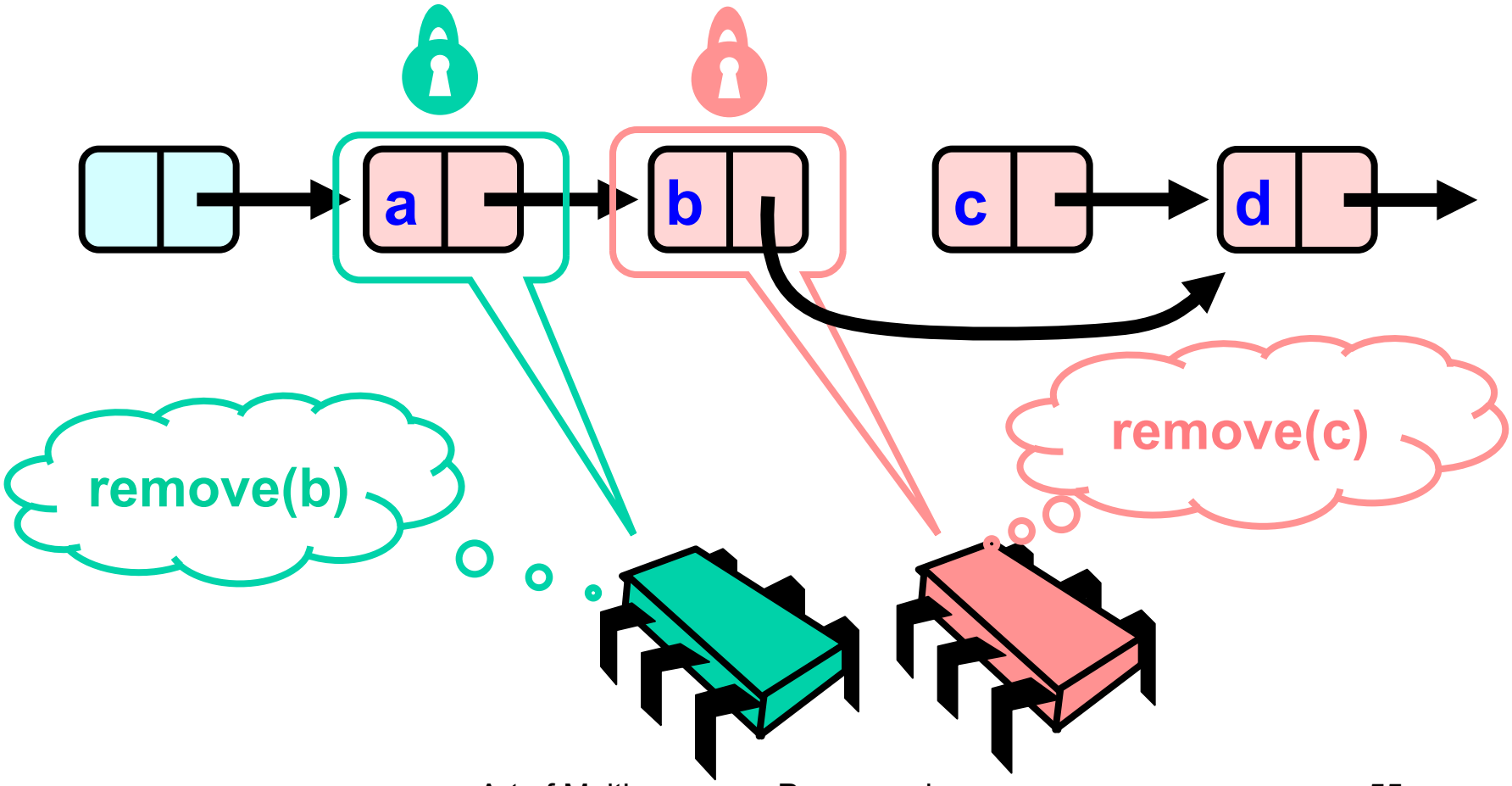
Concurrent Removes



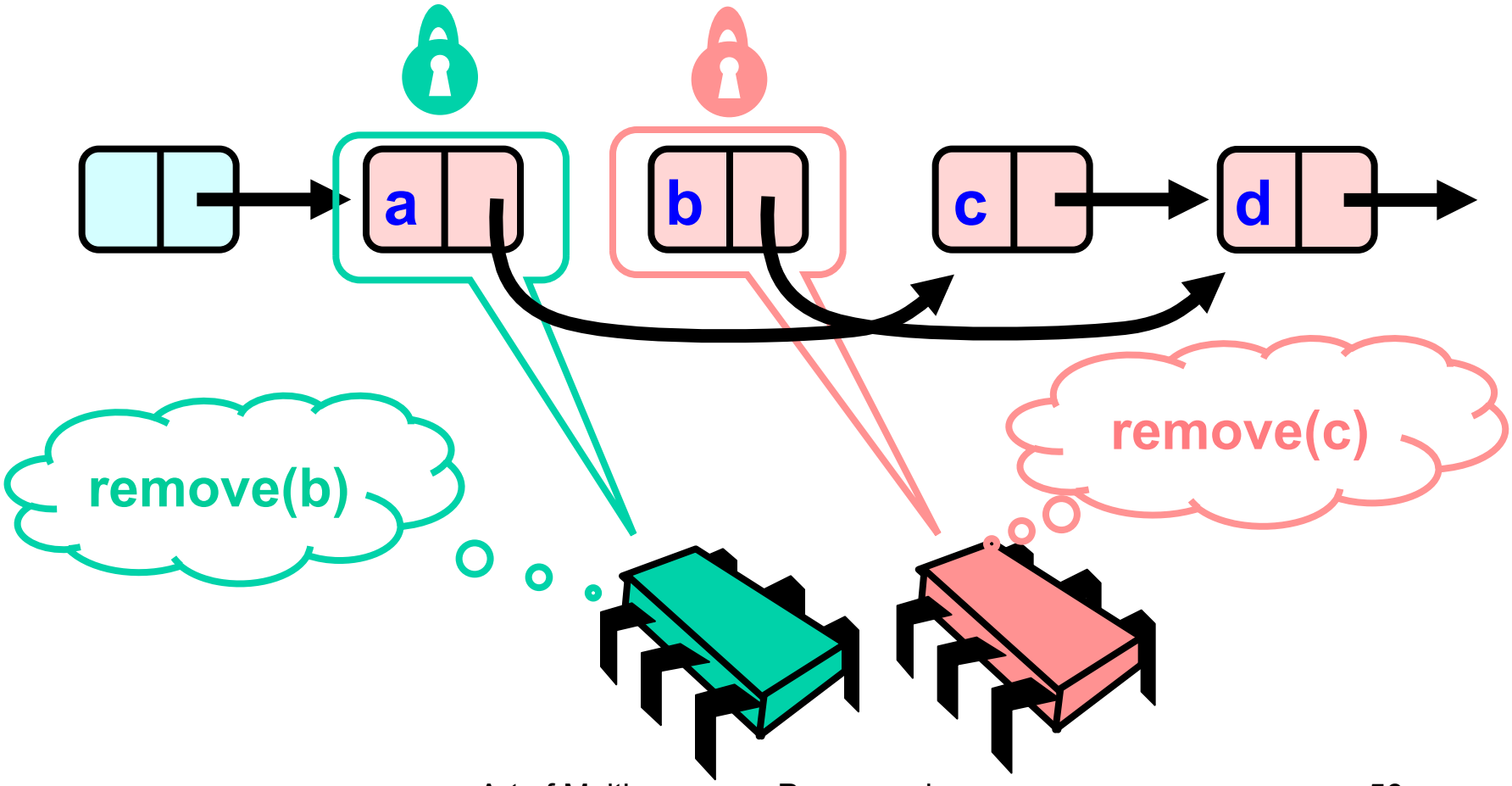
Concurrent Removes



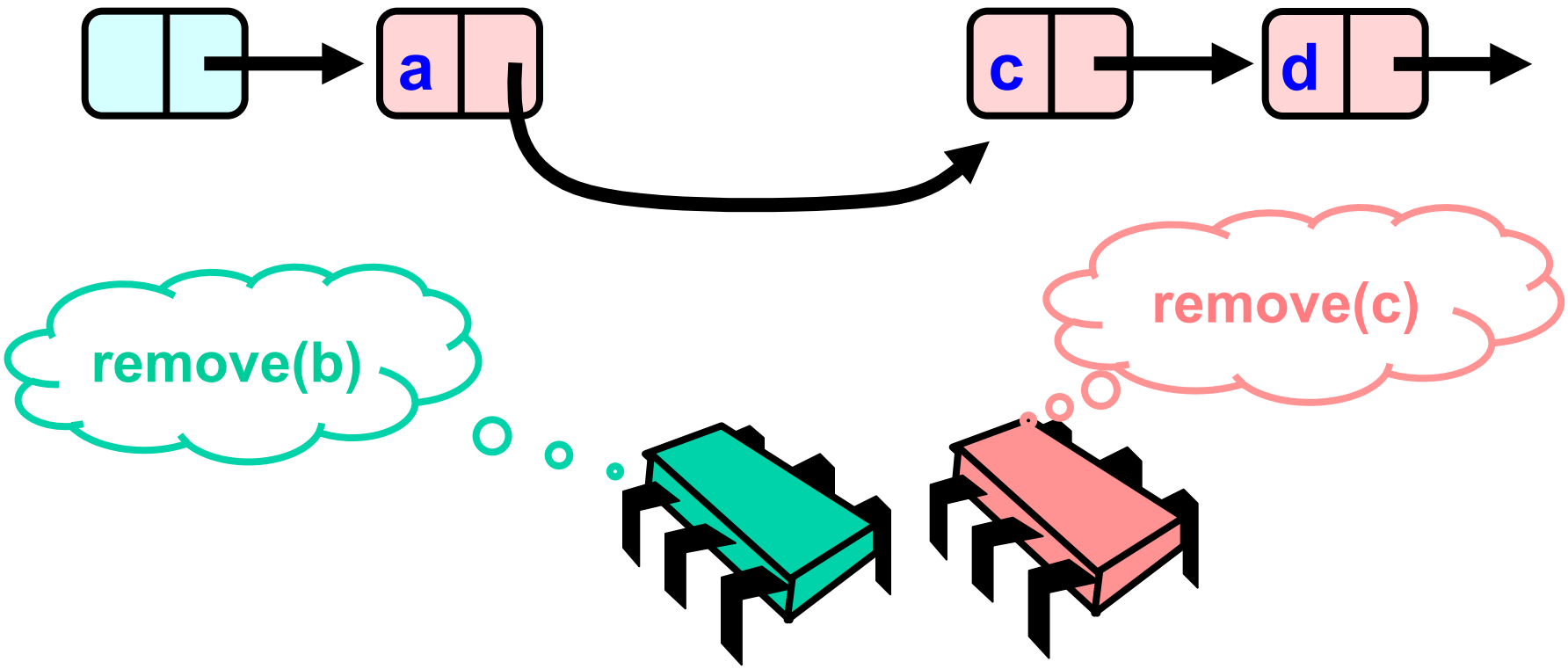
Concurrent Removes



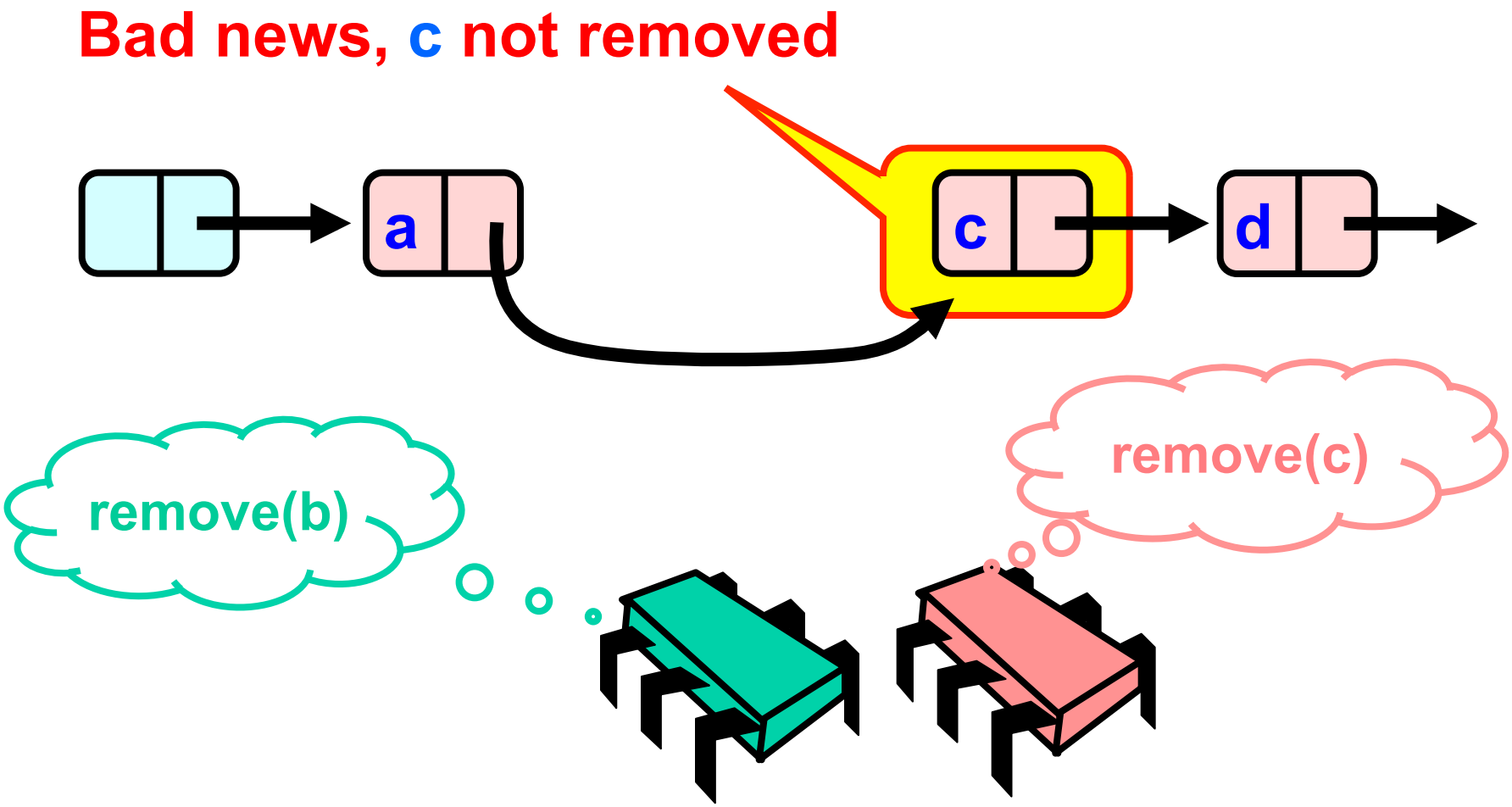
Concurrent Removes



Uh, Oh

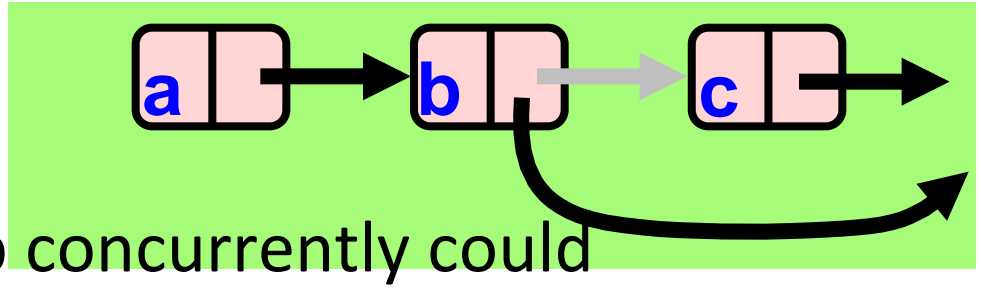


Uh, Oh

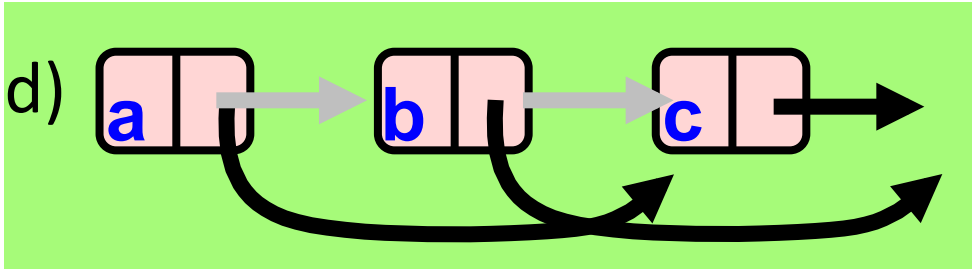


Problem

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



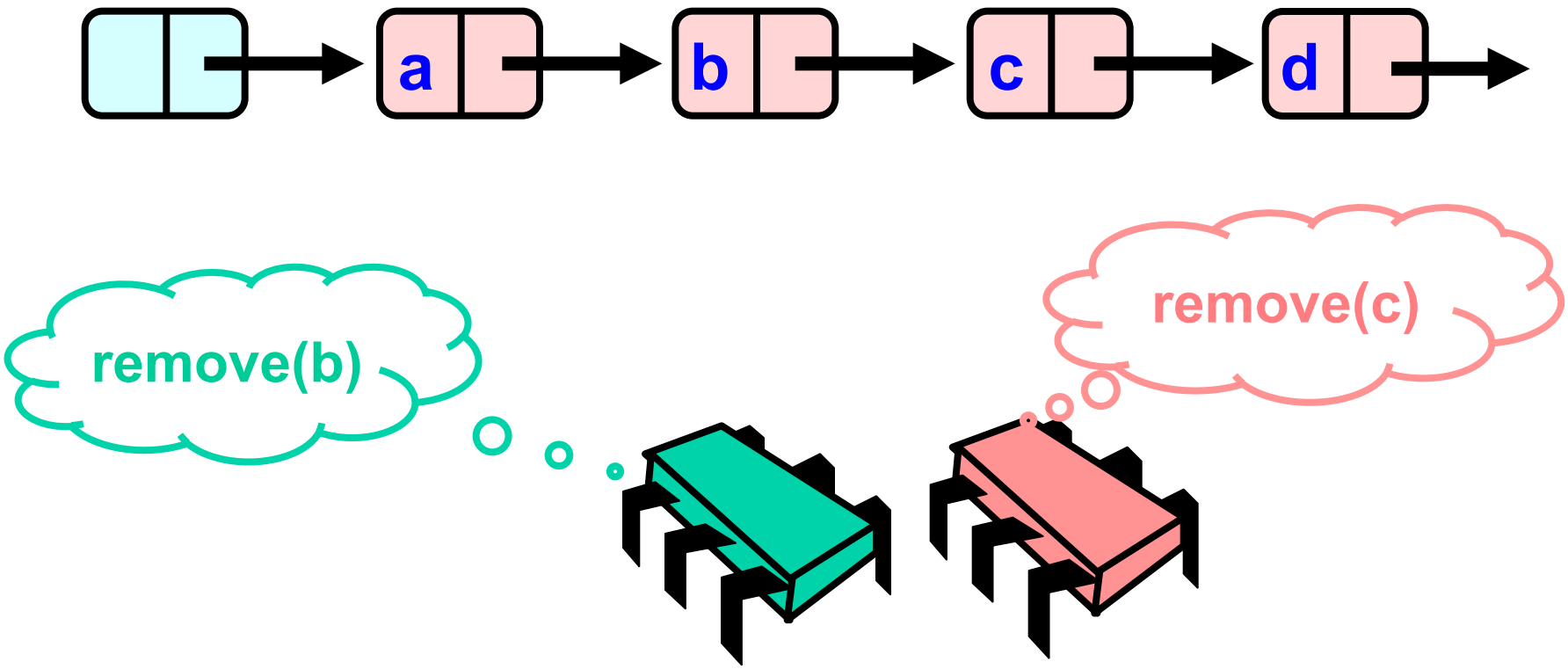
- Problem is,
  - Someone deleting b concurrently could direct a pointer to c (reading b's next field)



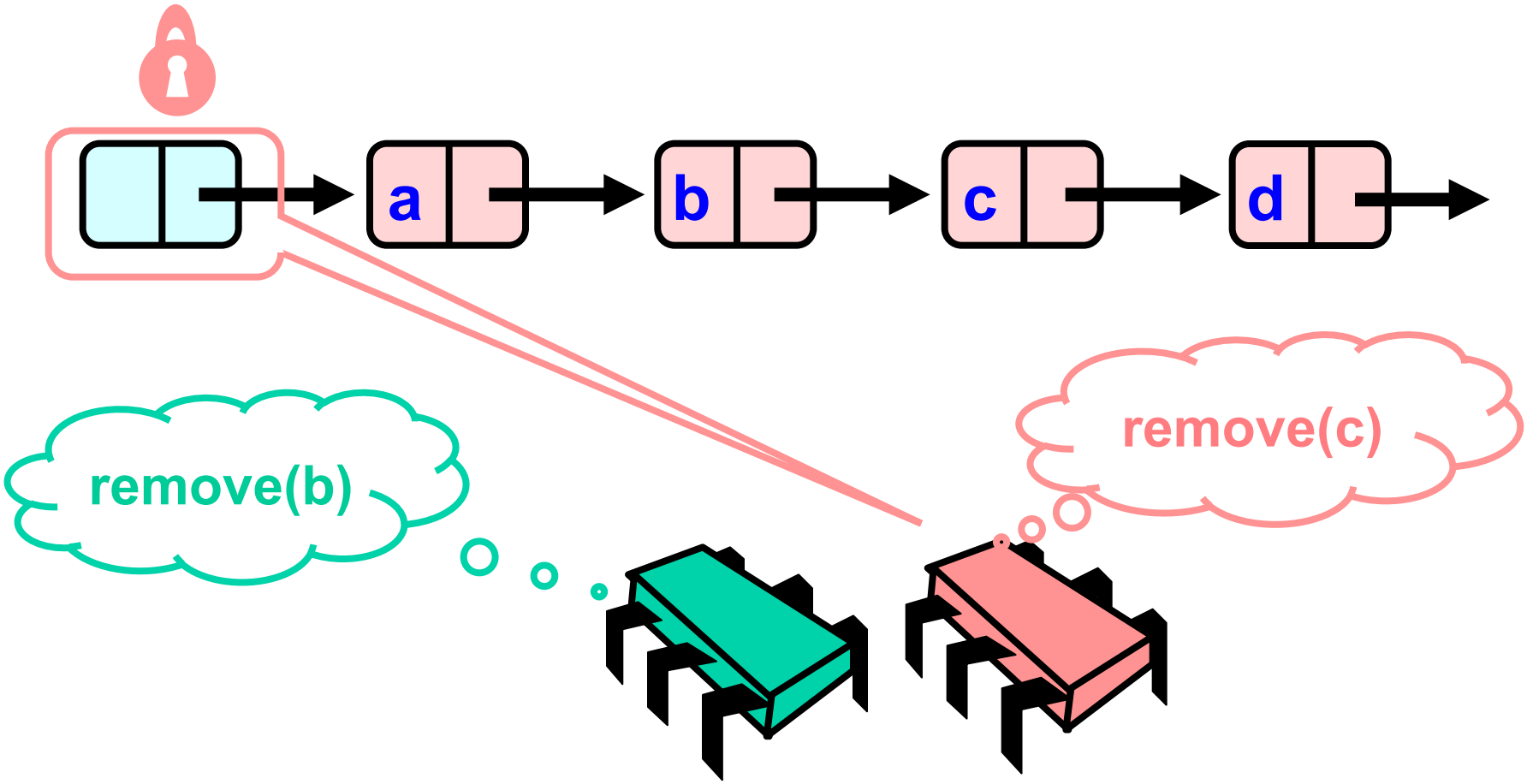
Insight

- If a node is locked
  - No one can **change** node's *successor*
- If a thread locks
  - Node to be deleted (so its successor don't change)
  - And its predecessor (so you are the only one changing its successor)
  - Then it works

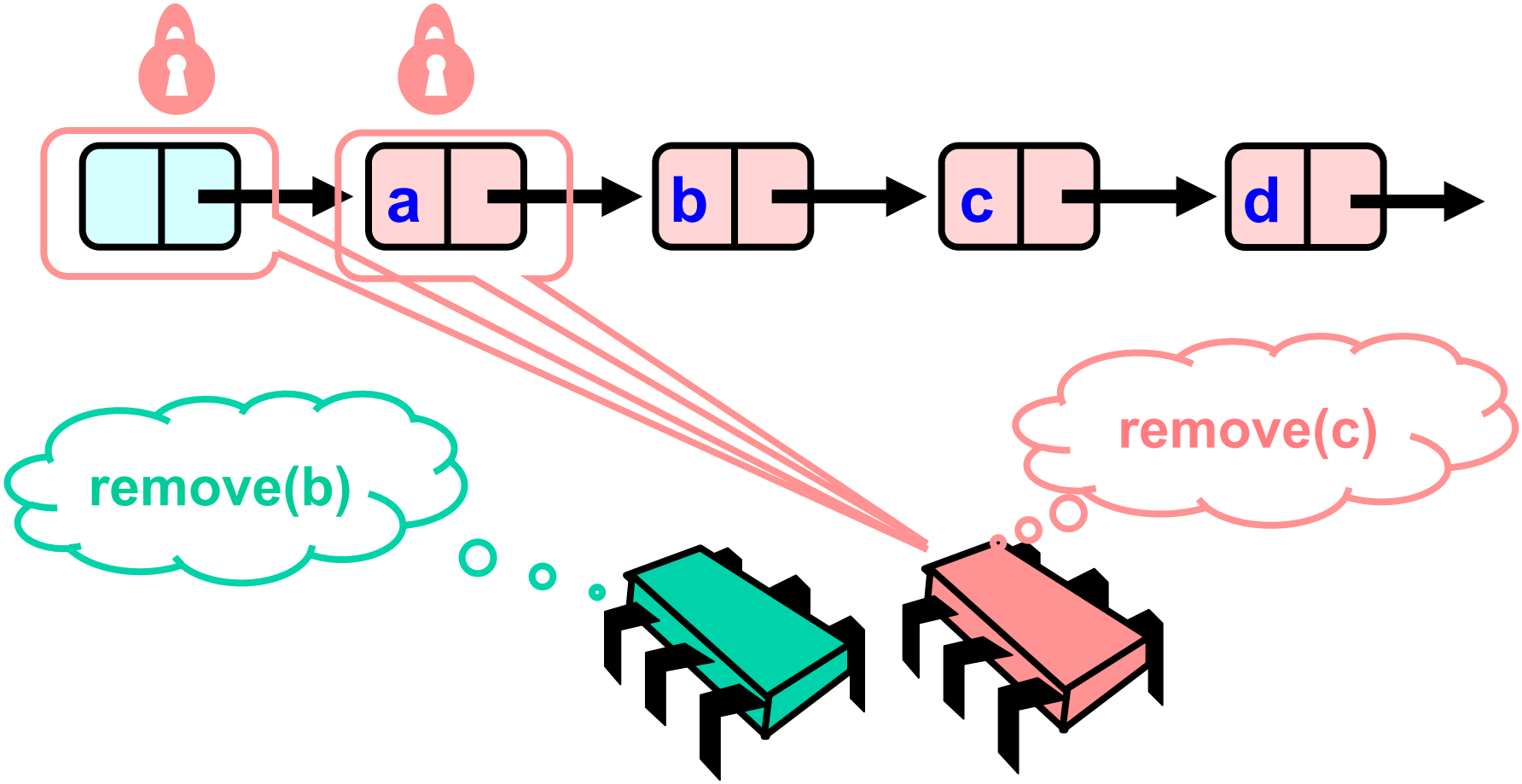
Removing a Node



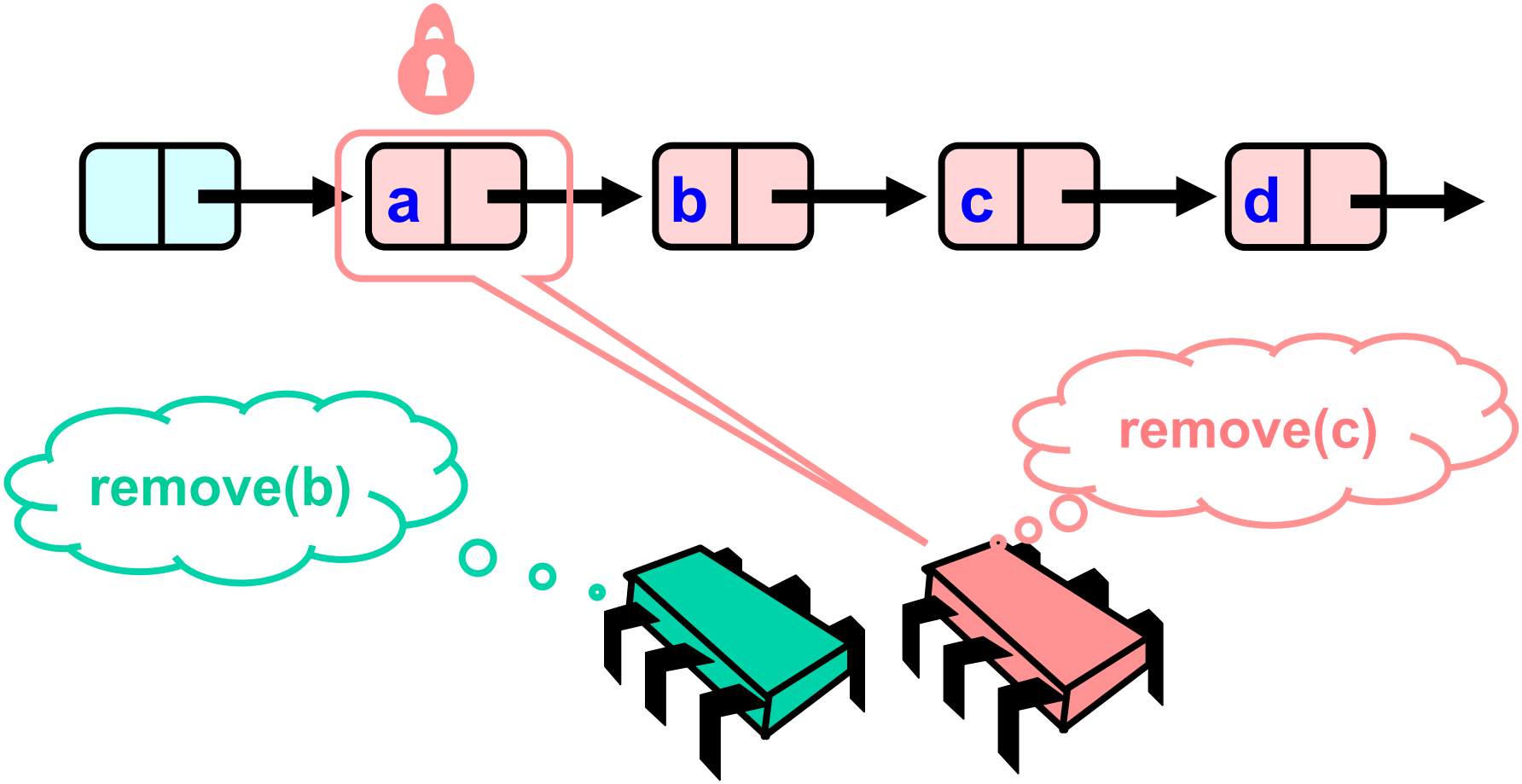
Removing a Node



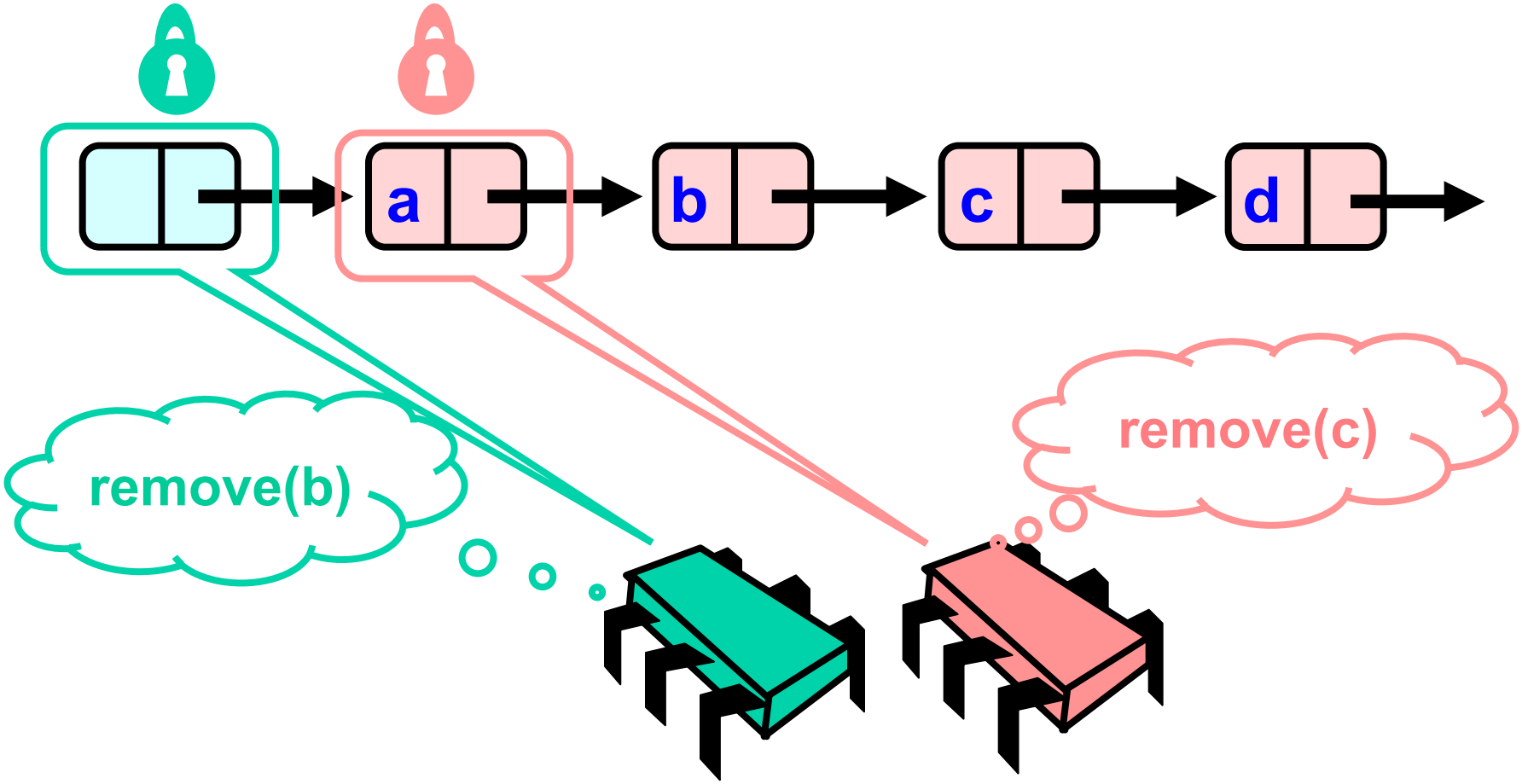
Removing a Node



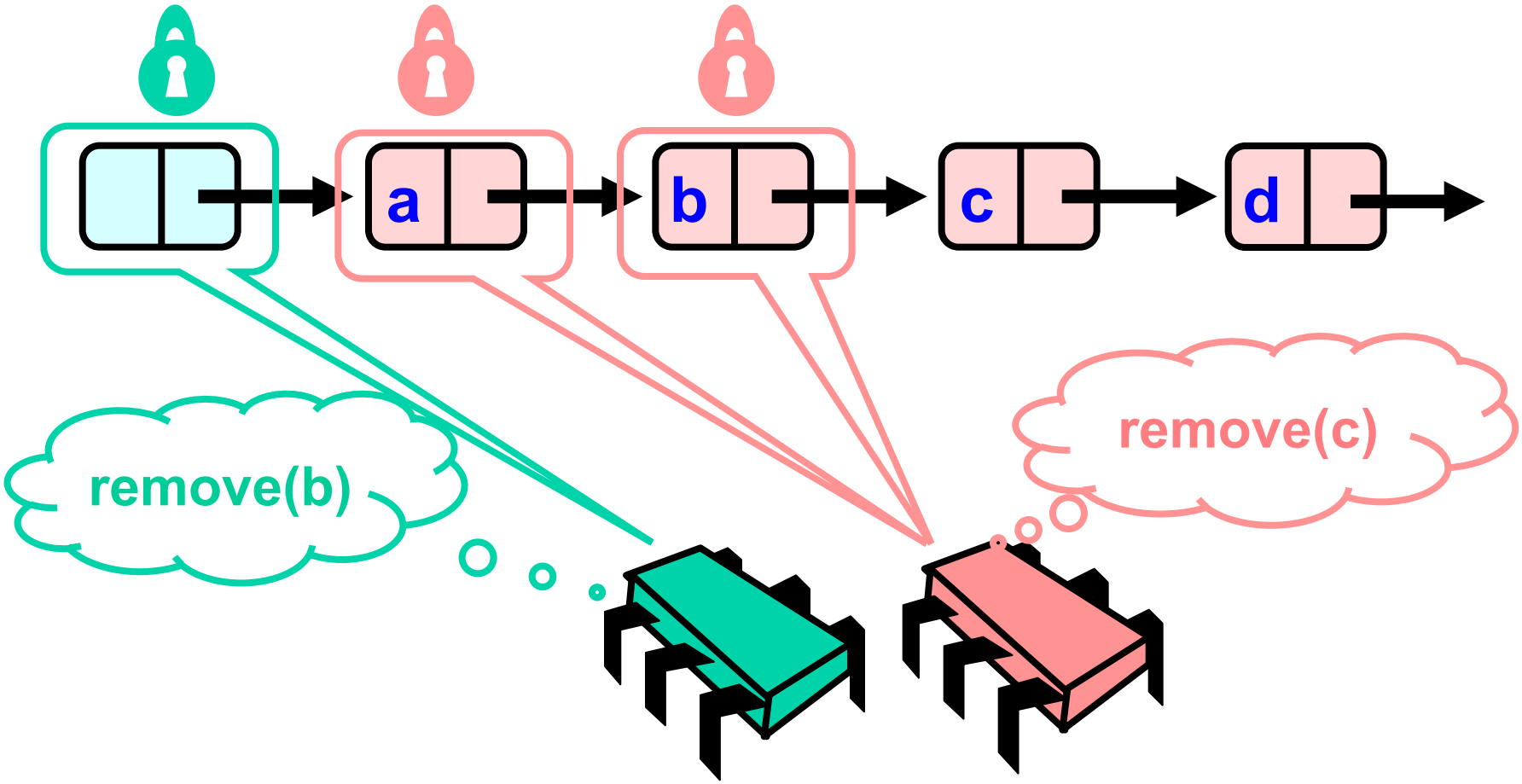
Removing a Node



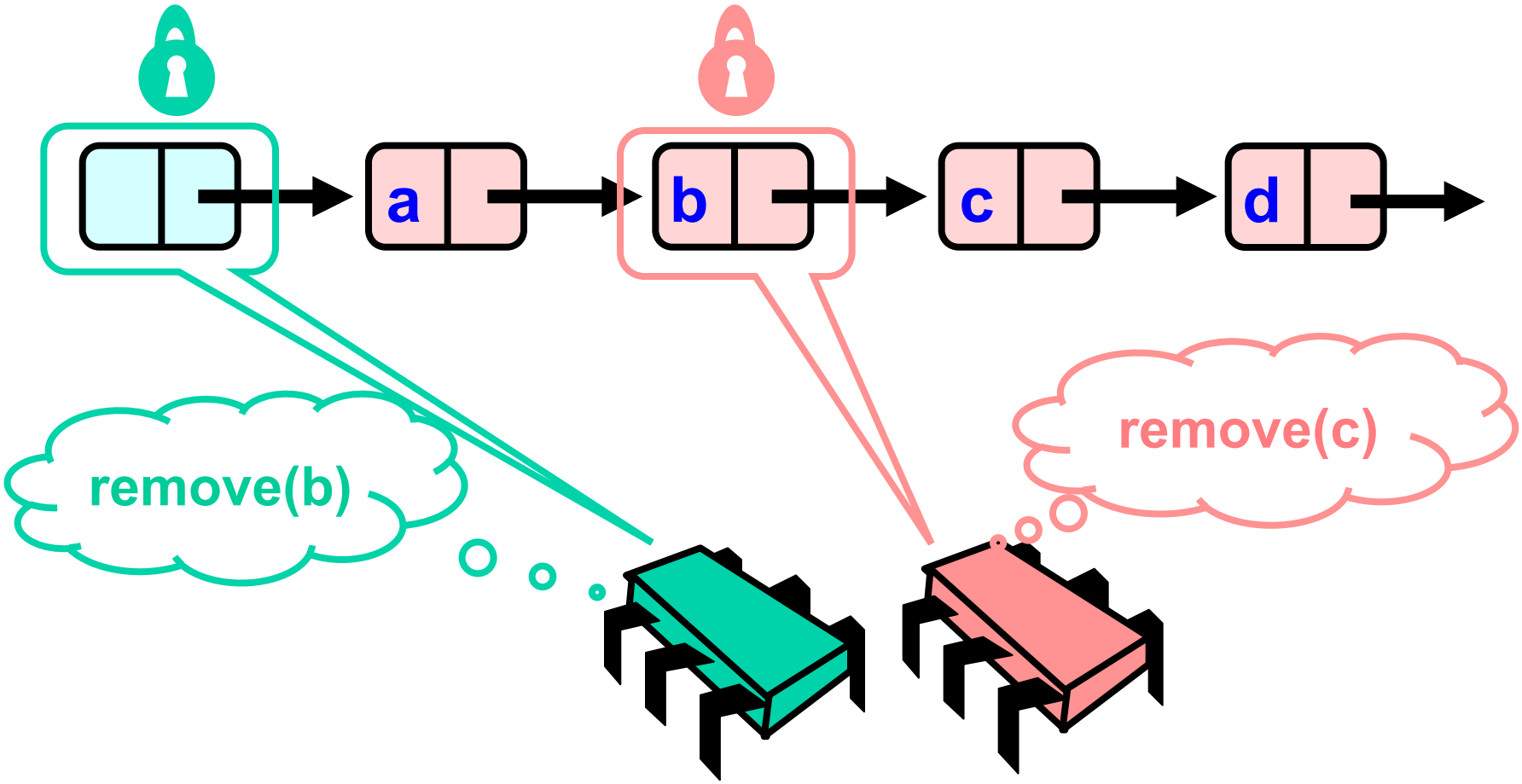
Removing a Node



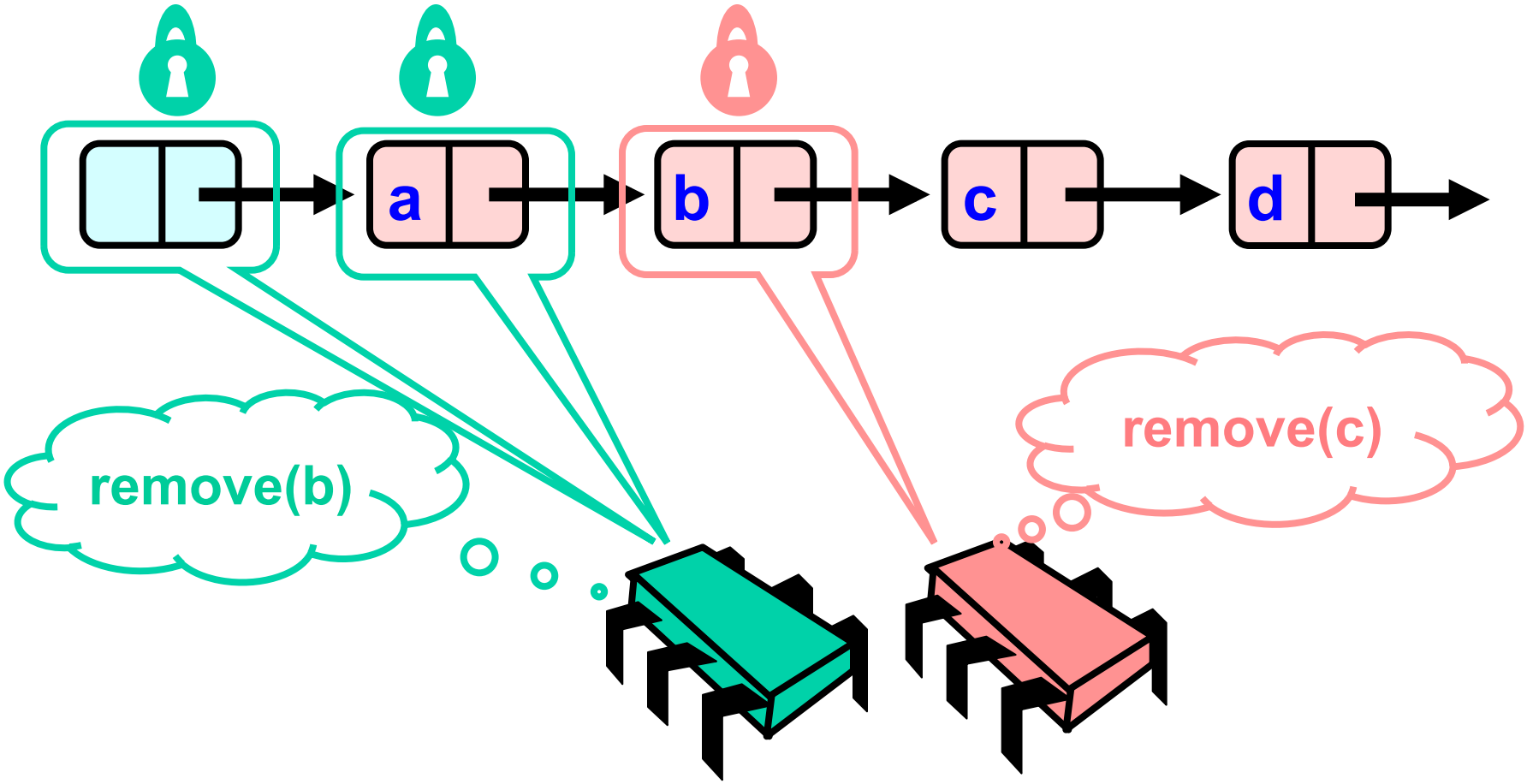
Removing a Node



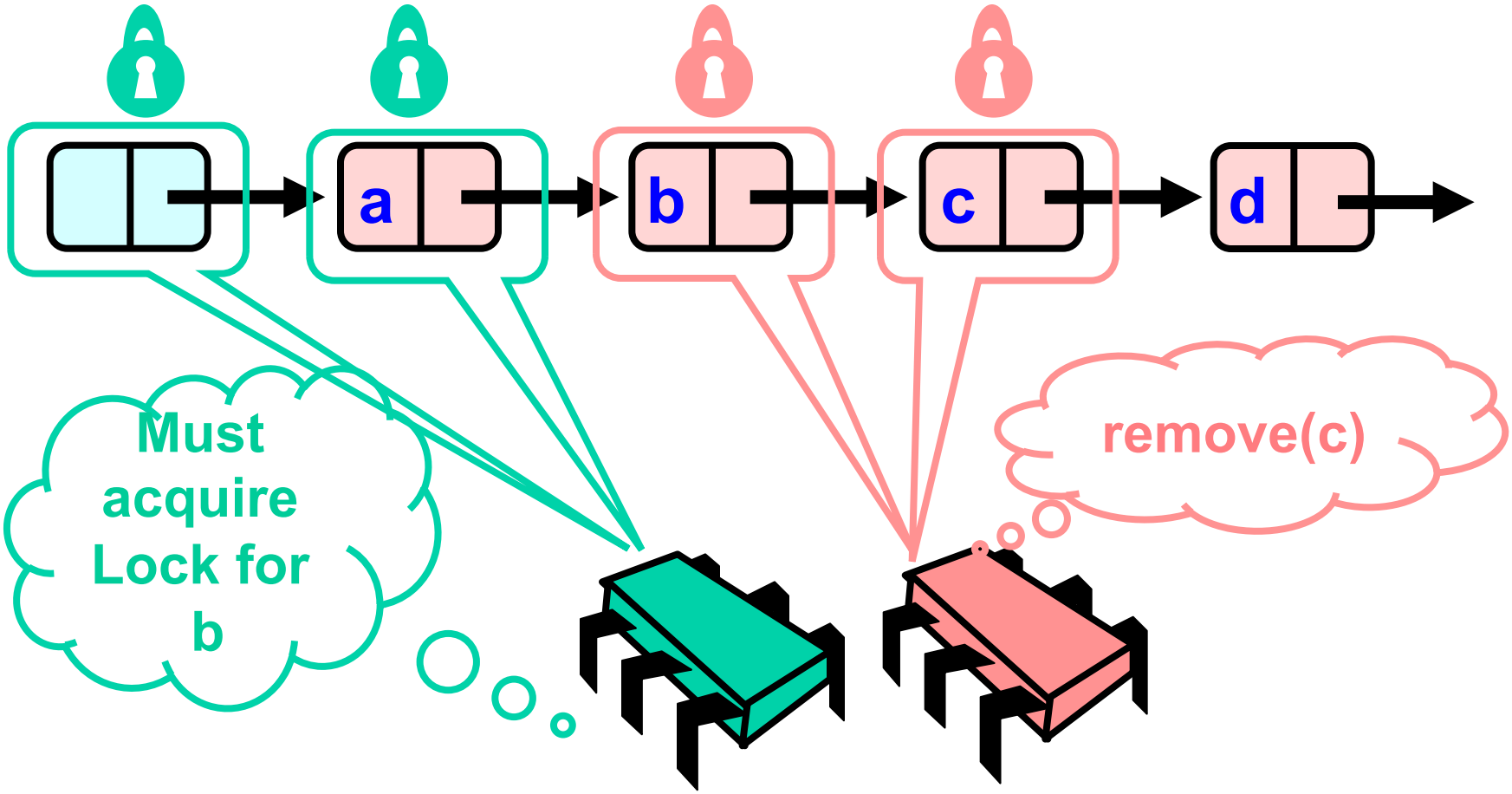
Removing a Node



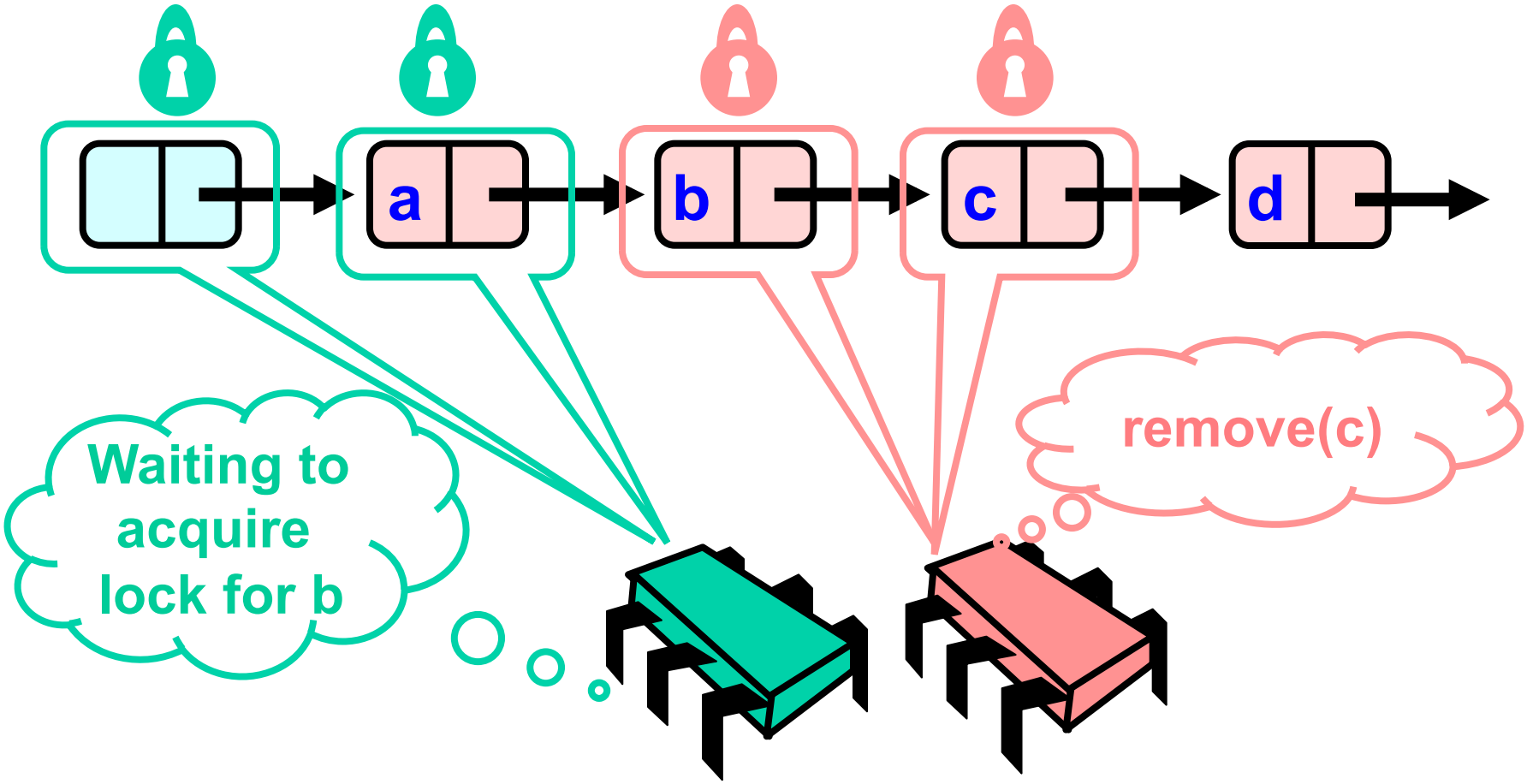
Removing a Node



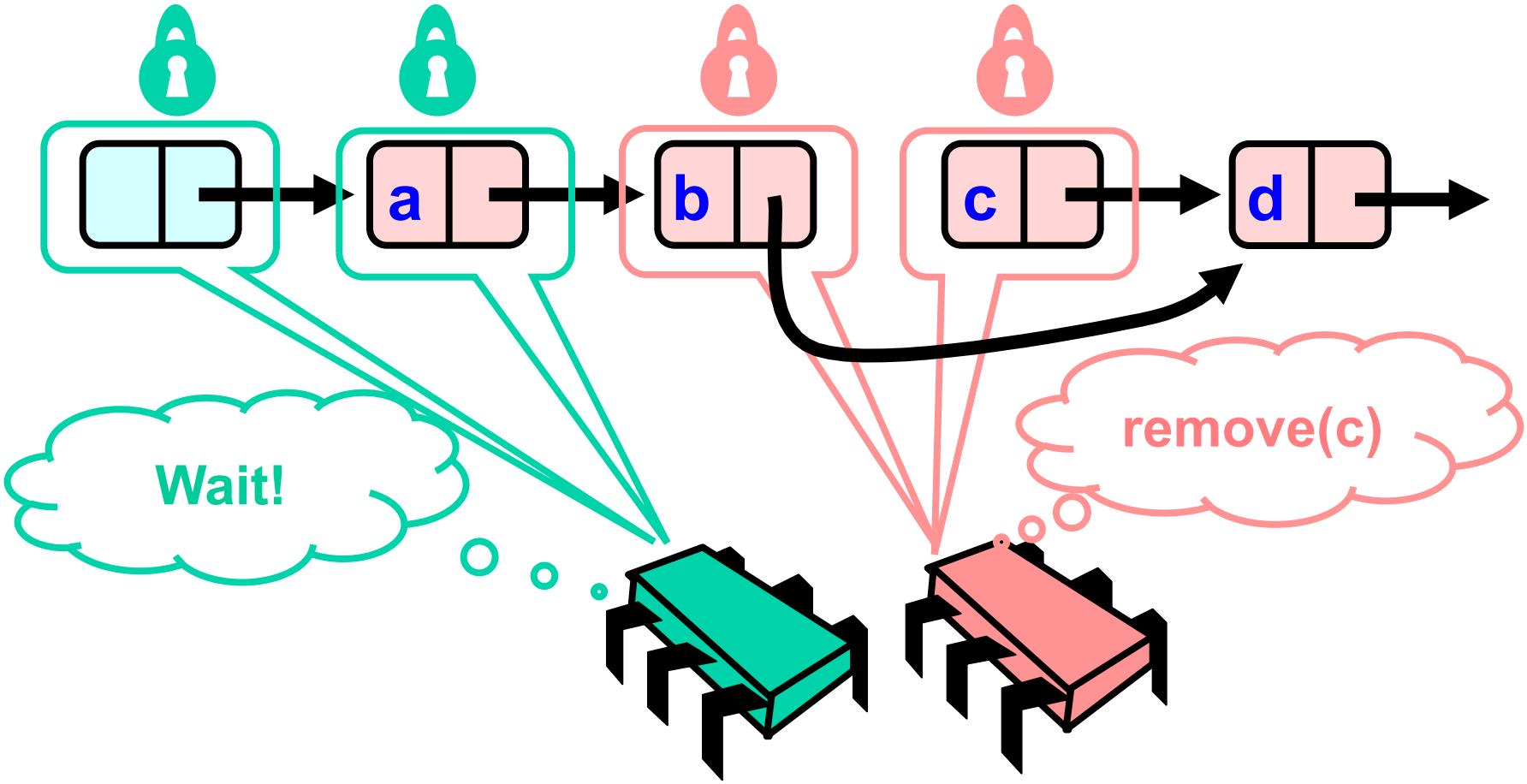
Removing a Node



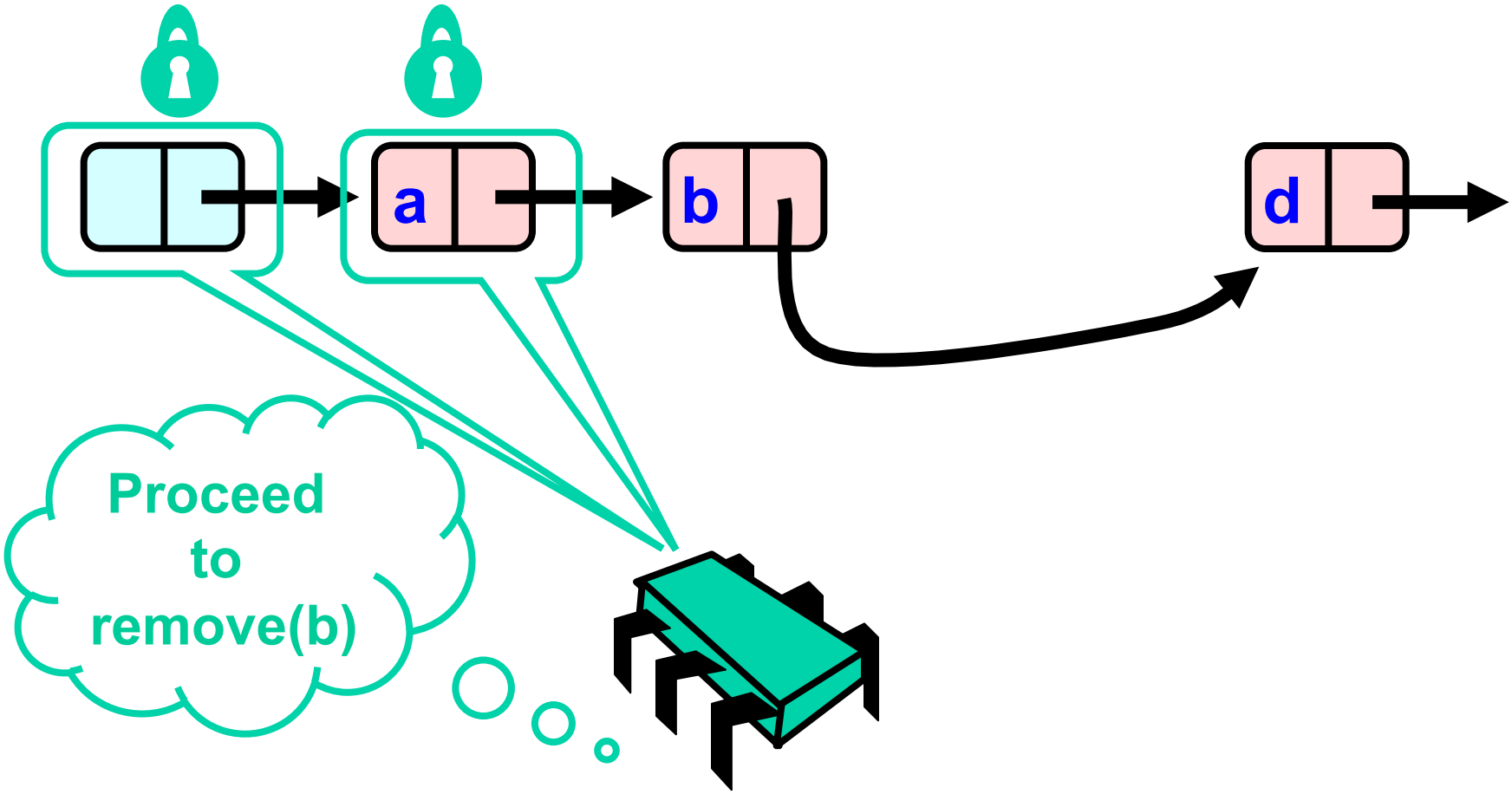
Removing a Node



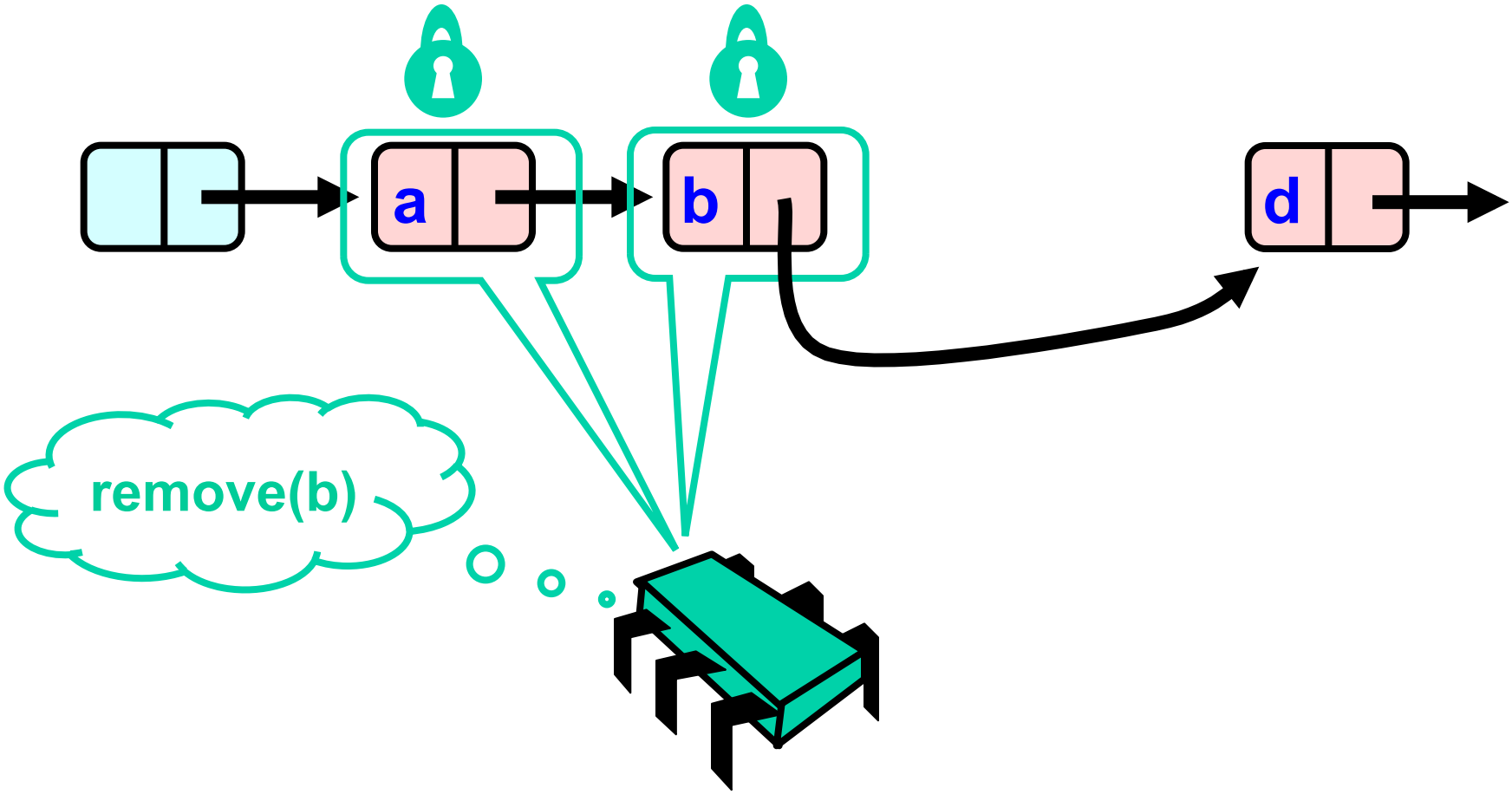
Removing a Node



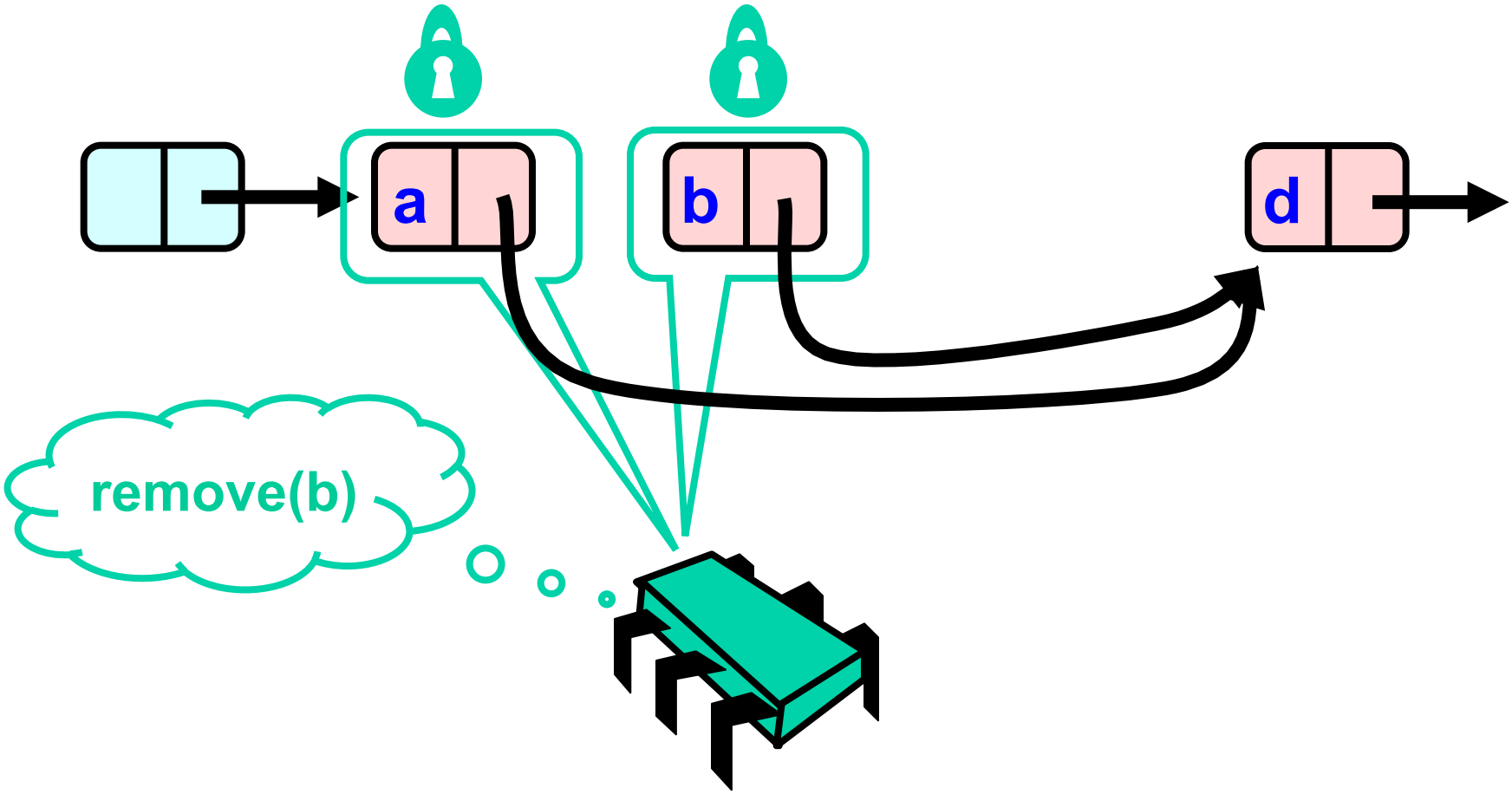
Removing a Node



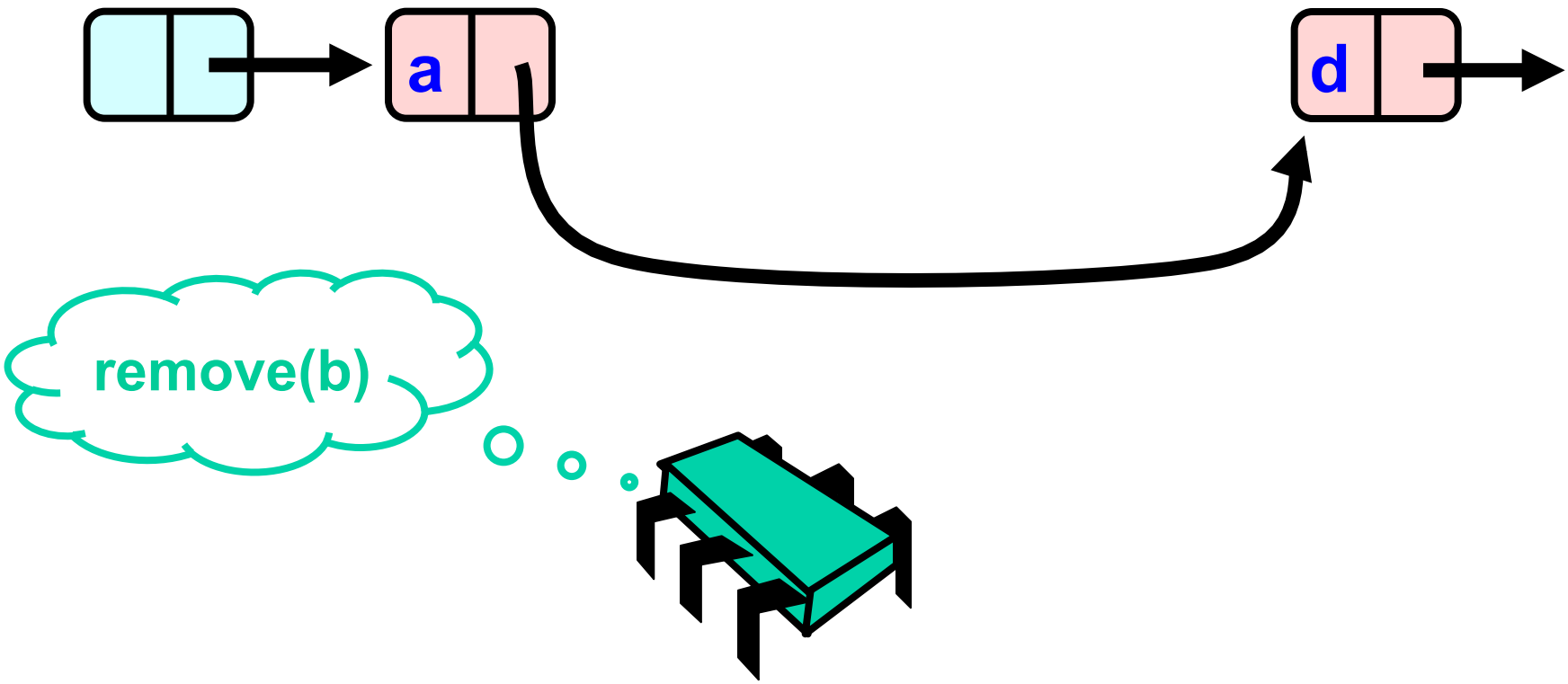
Removing a Node



Removing a Node



Removing a Node



Remove method

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

Remove method

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

Key used to order node

Remove method

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

Predecessor and current nodes

Remove method

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

lock pred == head before accessing its next field

Remove method

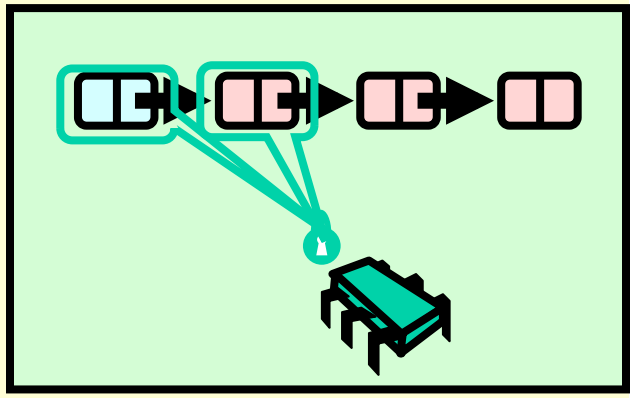
```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 pred = head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 try {
    …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

lock the node after head

Remove method

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

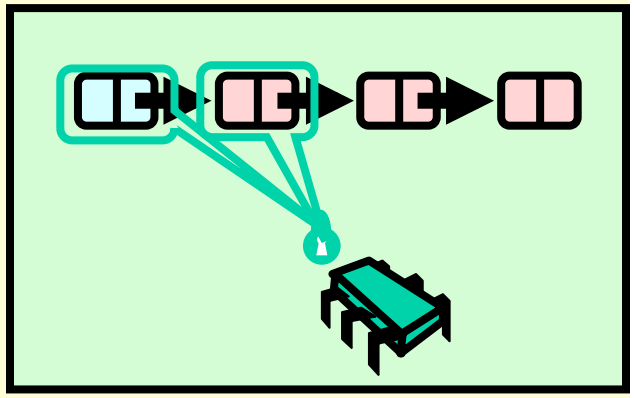
When enter try, we hold locks on pred and curr



Remove method

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

Traverse the rest of the list



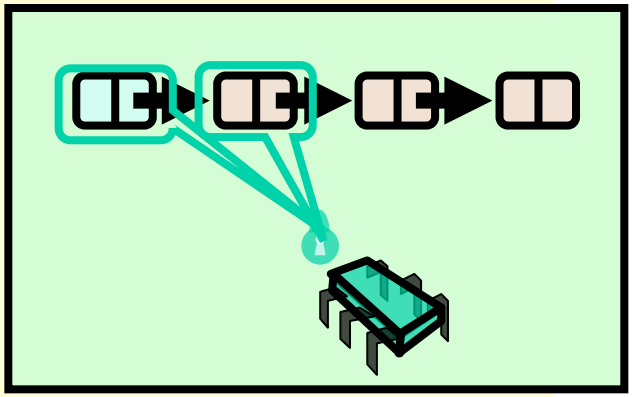
Remove method

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

Make sure locks released

Remove: searching (Inside the Try Block)

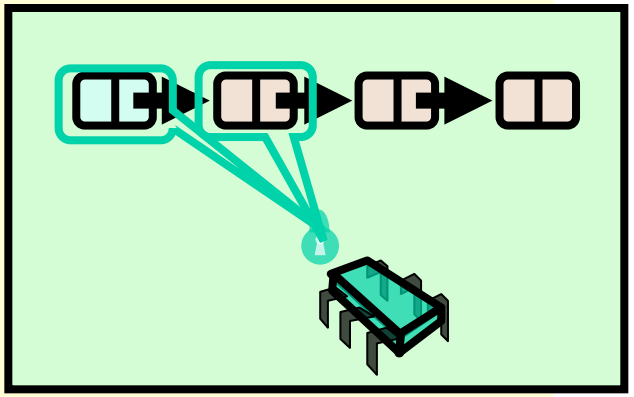
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching (Inside the Try Block)

```
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;
```

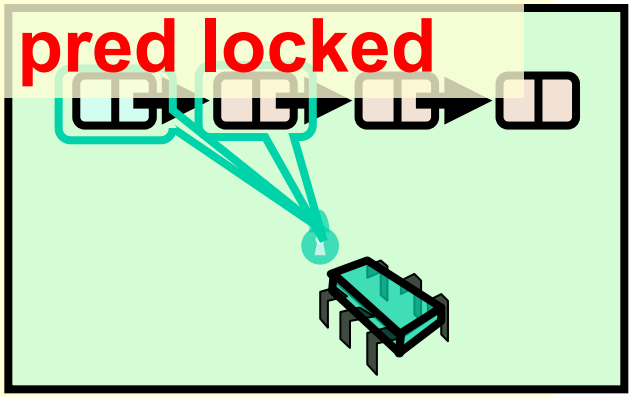
Search key range



Remove: searching (Inside the Try Block)

```
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;
```

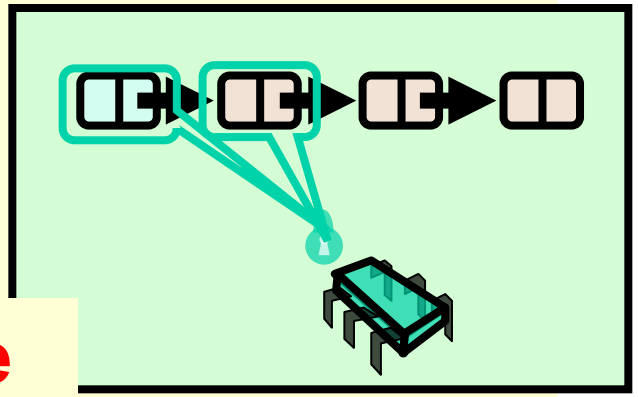
At start of each loop: curr and pred locked



Remove: searching (Inside the Try Block)

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

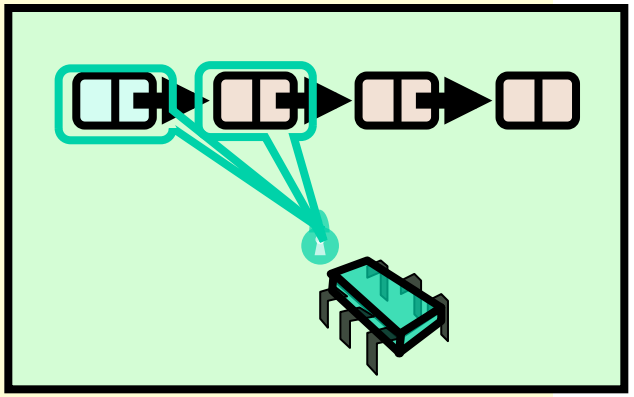
If item found, remove node



Remove: searching (Inside the Try Block)

```
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;
```

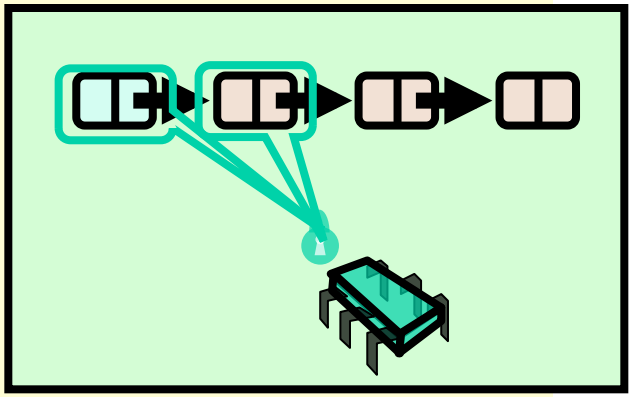
Hand-over-hand locking again otherwise



Remove: searching (Inside the Try Block)

```
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;
```

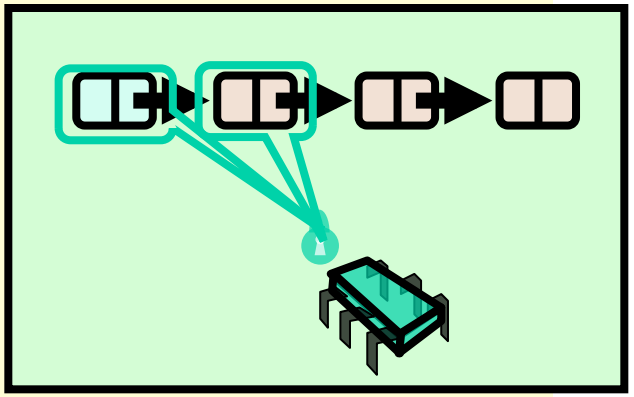
Lock invariant restored



Remove: searching (Inside the Try Block)

```
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



Aside: Next Field Must be Volatile!

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

Since we are no longer holding a lock when we read the "next" field, it needs to be volatile to avoid race conditions (more on that in future lecture).

Why remove() is linearizable

```
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;
```

- **pred** reachable from **head**
- **curr** is **pred.next**
- So **curr.item** is in the set

Why remove() is linearizable

```
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;
```

Linearization point if item is present

Why remove() is linearizable

```
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;
```

Node locked, so no other thread can remove it ….

Why remove() is linearizable

```
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;
```

Item not present

Why remove() is linearizable

```
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;
```

- **pred** reachable from **head**
- **curr** is **pred.next**
- **pred.key <** `key`
- key < **curr.key**

Why remove() is linearizable

```
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;
```

Linearization point: the most recent read before return

Adding Nodes

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

Abstraction Map

- S(head) =

  { x | there exists a such that

    - a reachable from head and
    - a.item = x

  }

Representation Invariant

- Easy to check that
  - tail always reachable from head
  - Nodes sorted, no duplicates

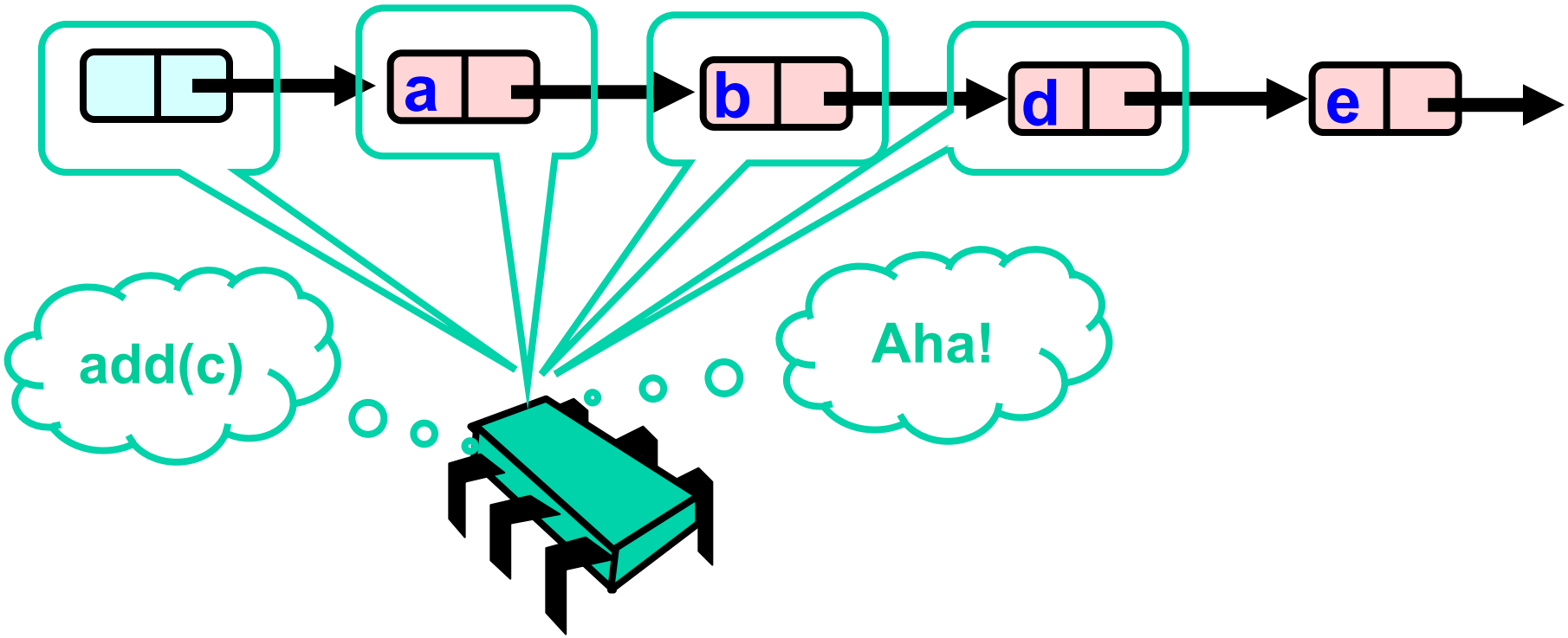
Drawbacks

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

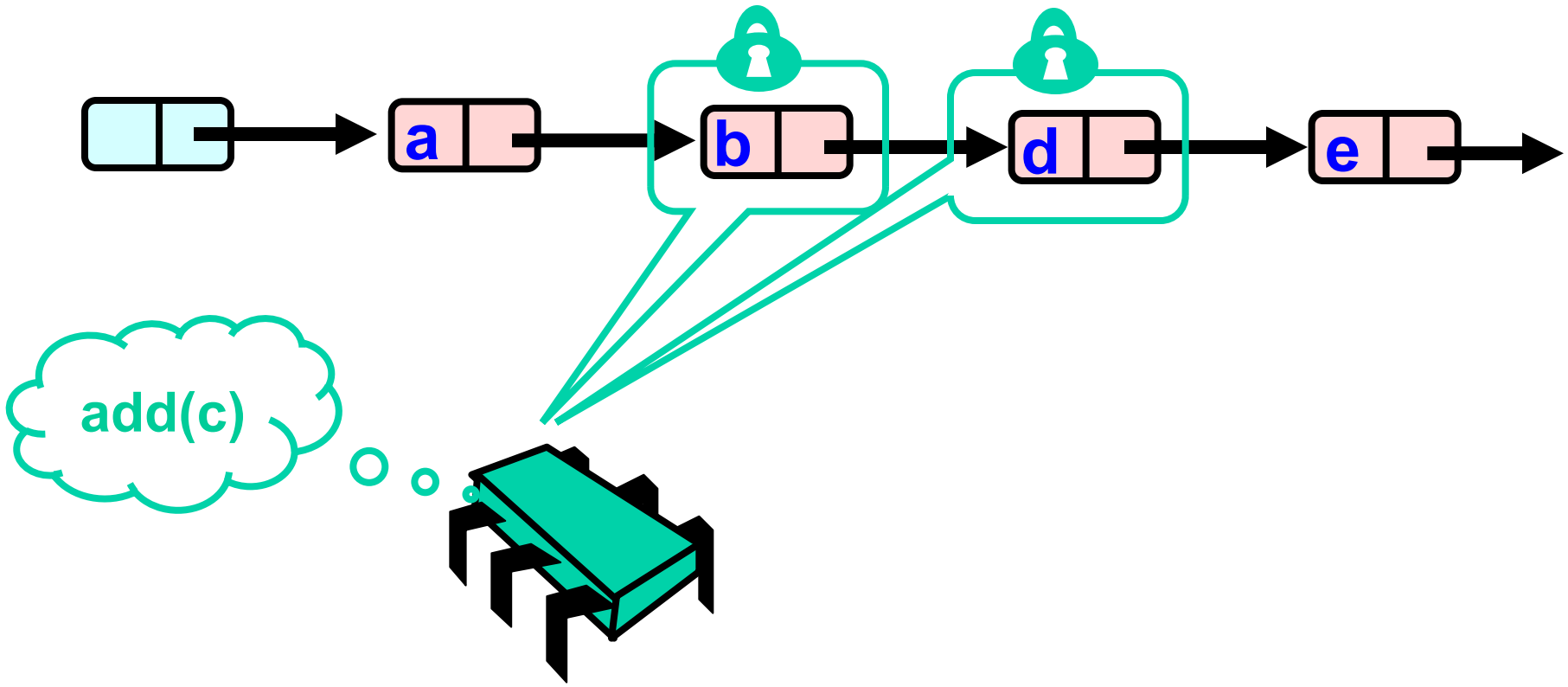
Optimistic Synchronization

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

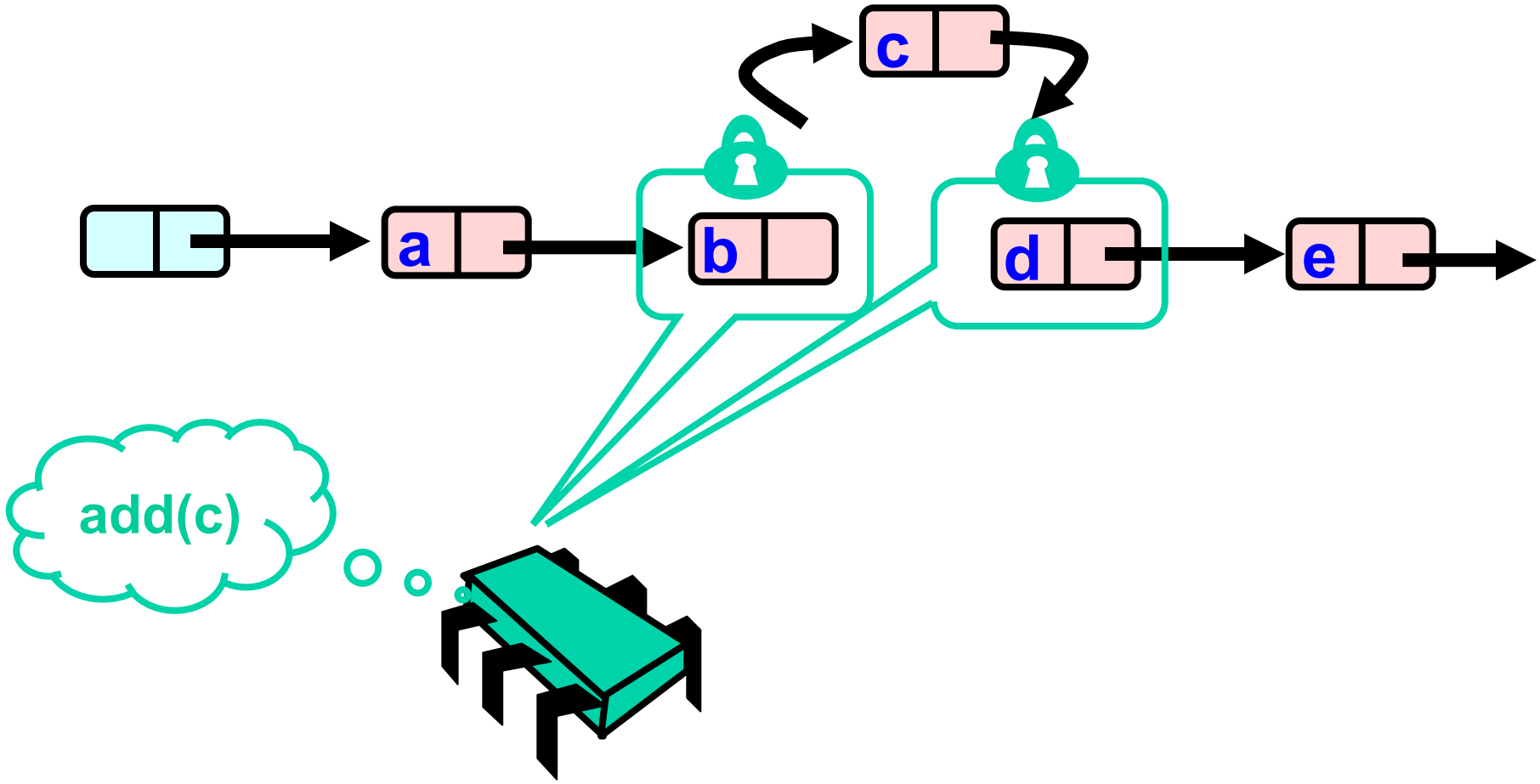
Optimistic: Traverse without Locking



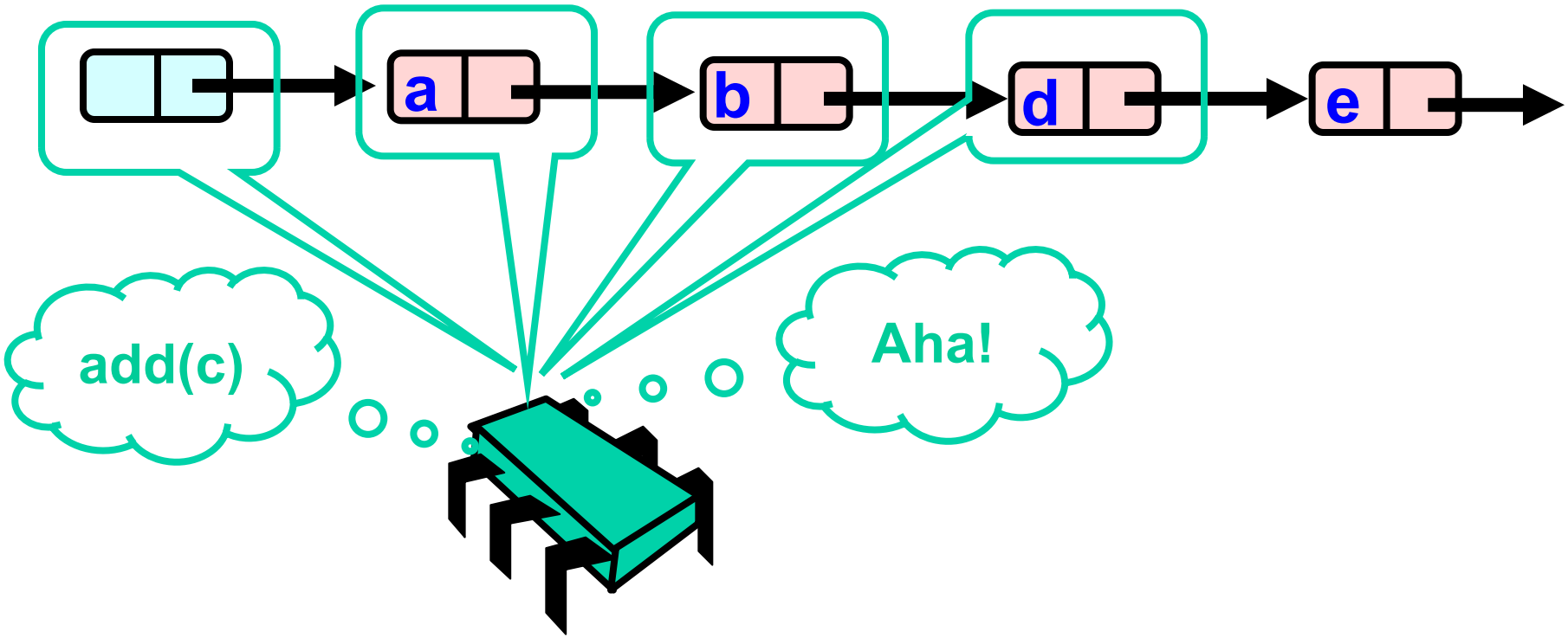
Optimistic: Lock and Load



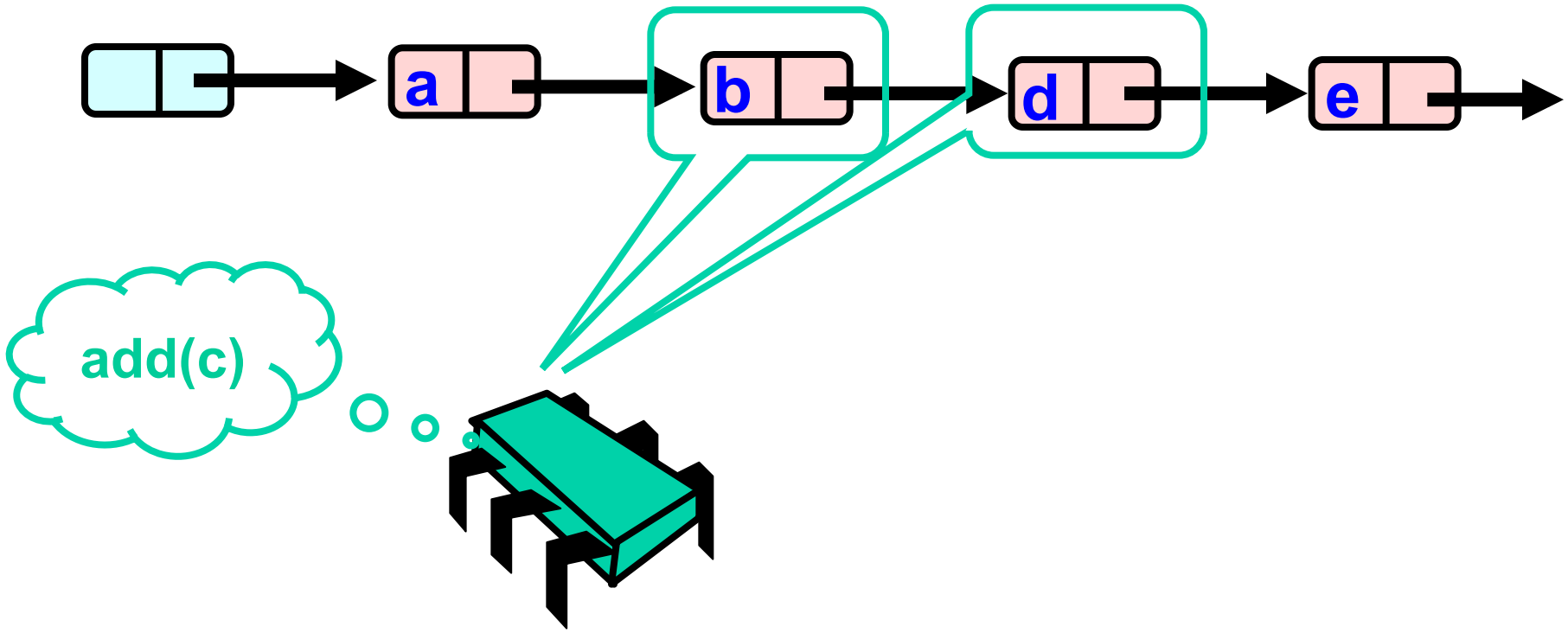
Optimistic: Lock and Load



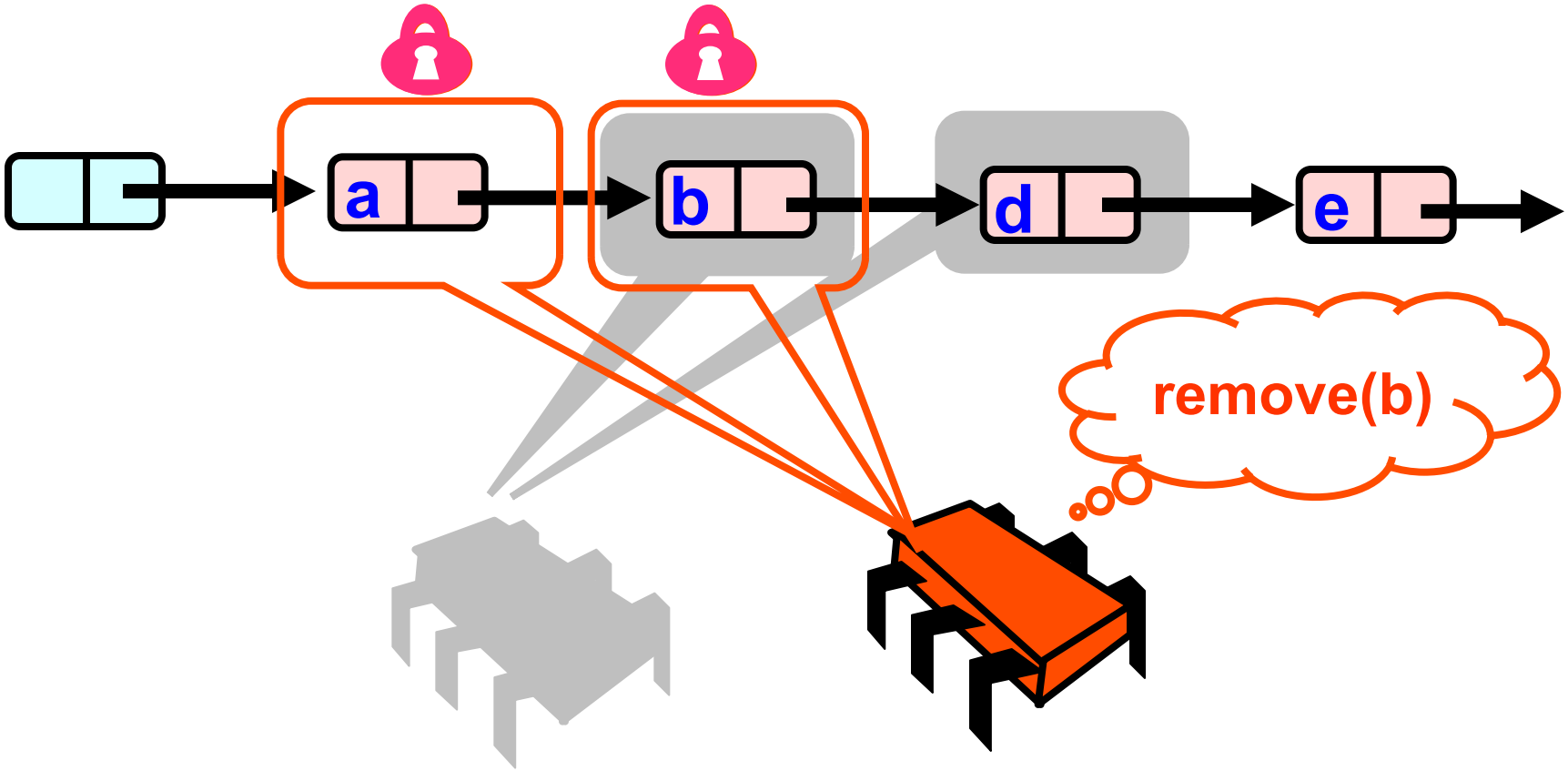
What could go wrong?



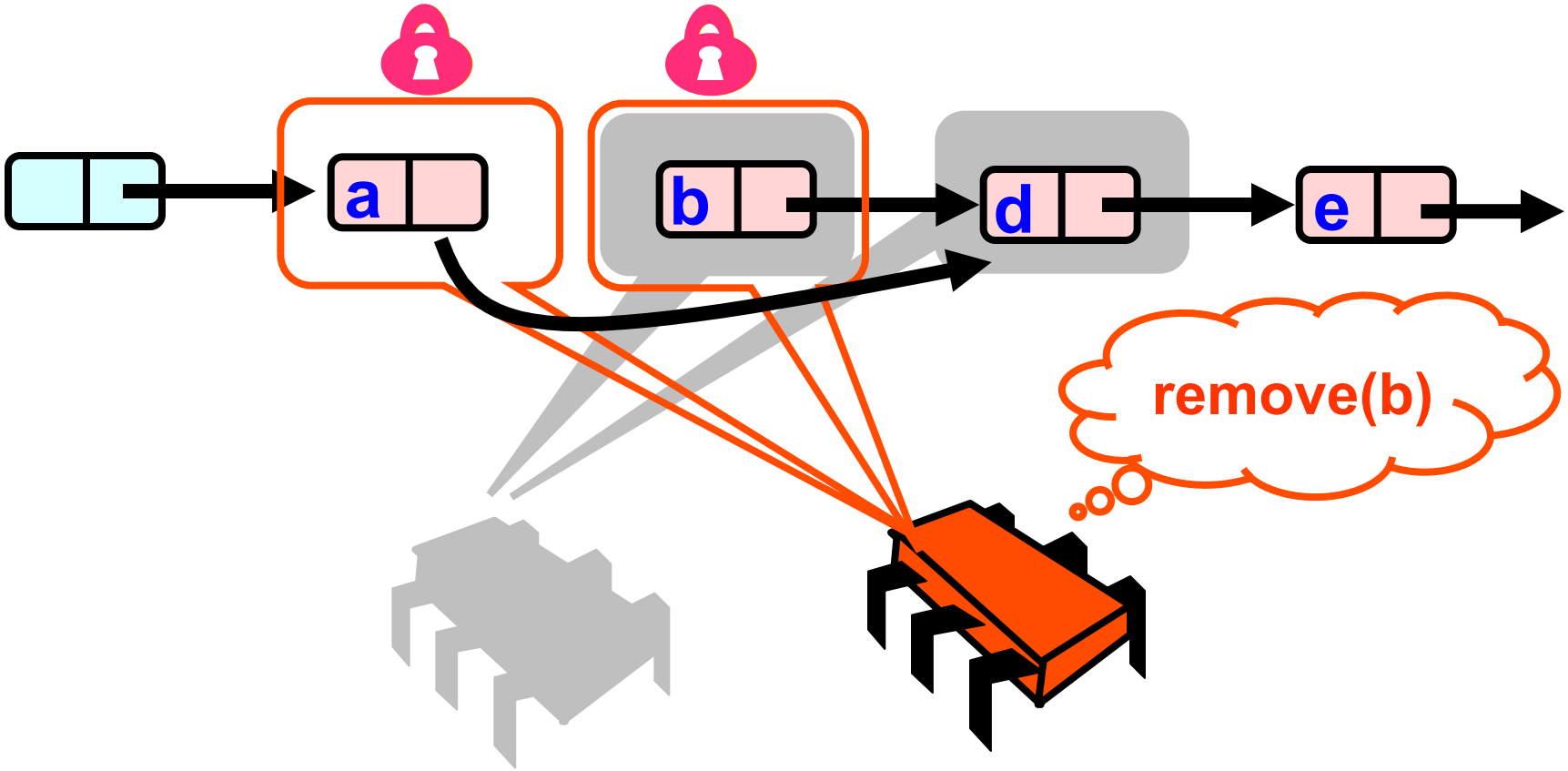
What could go wrong?



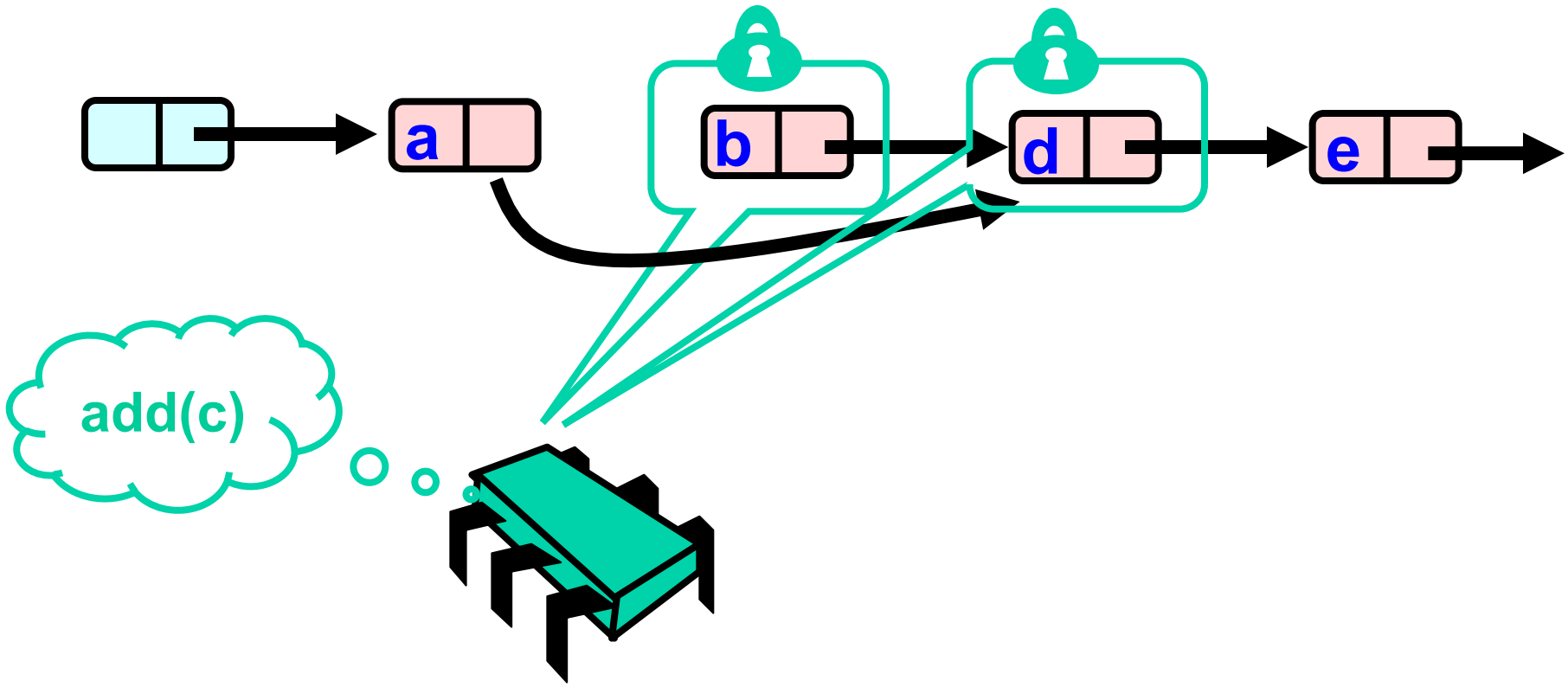
What could go wrong?



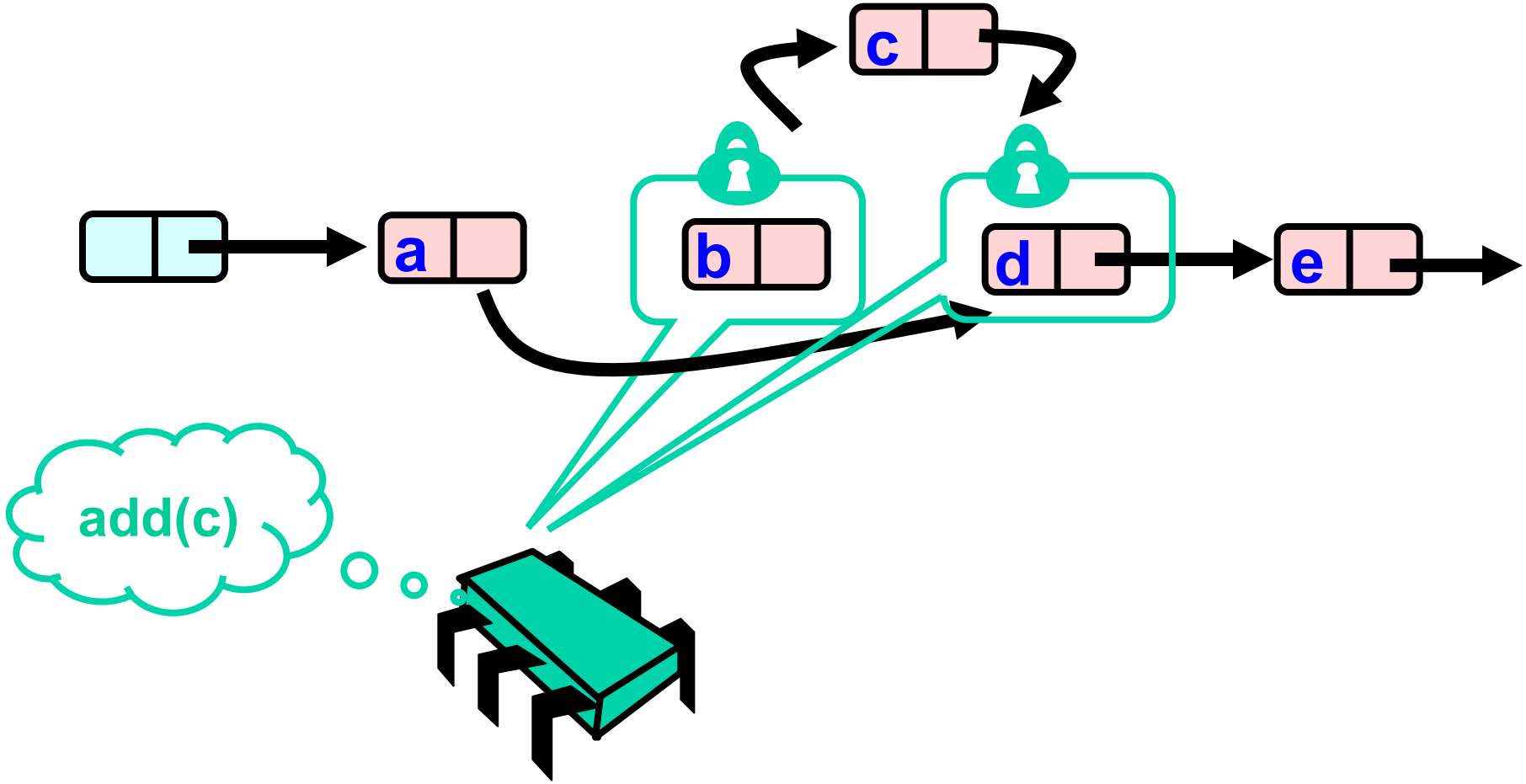
What could go wrong?



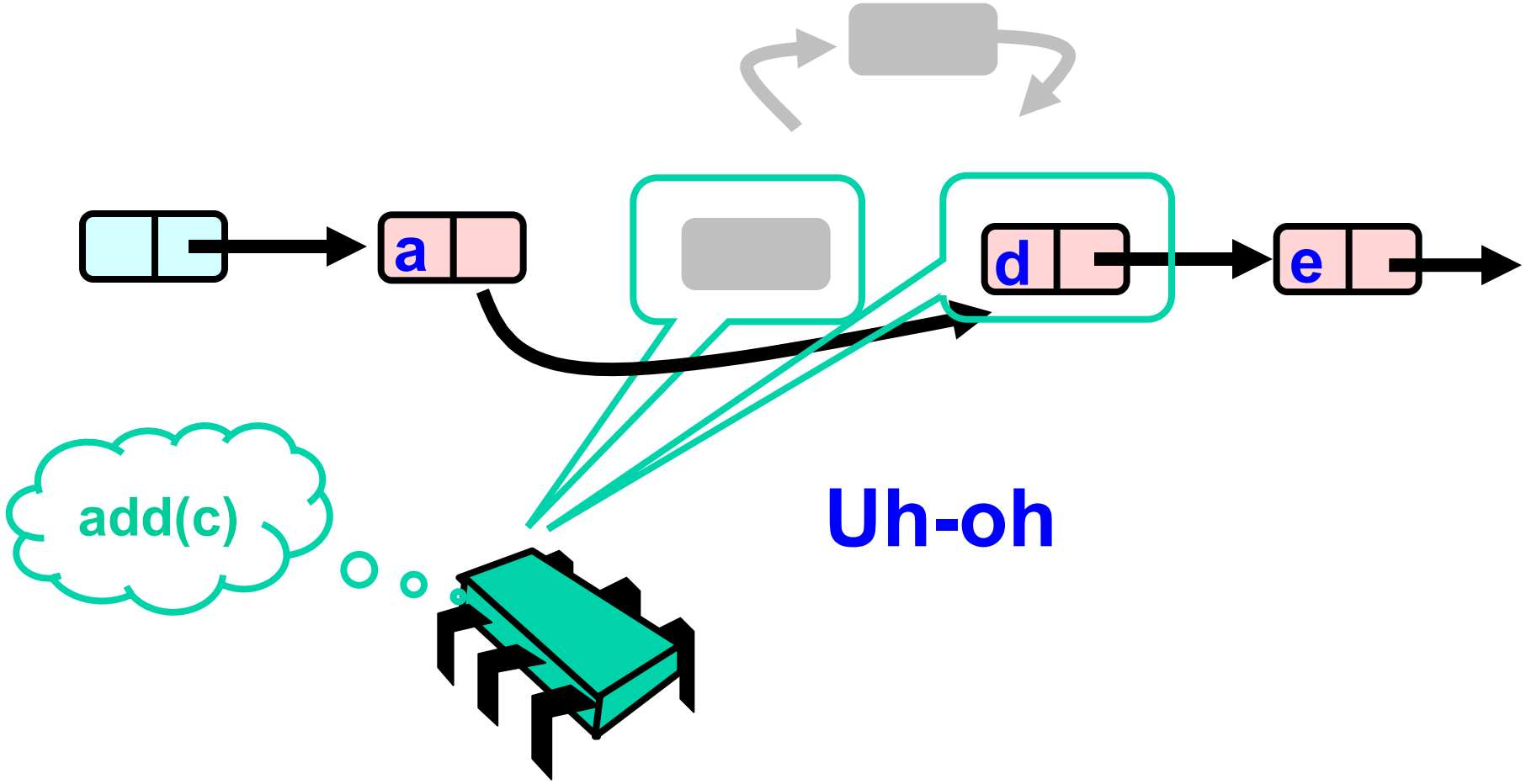
What could go wrong?



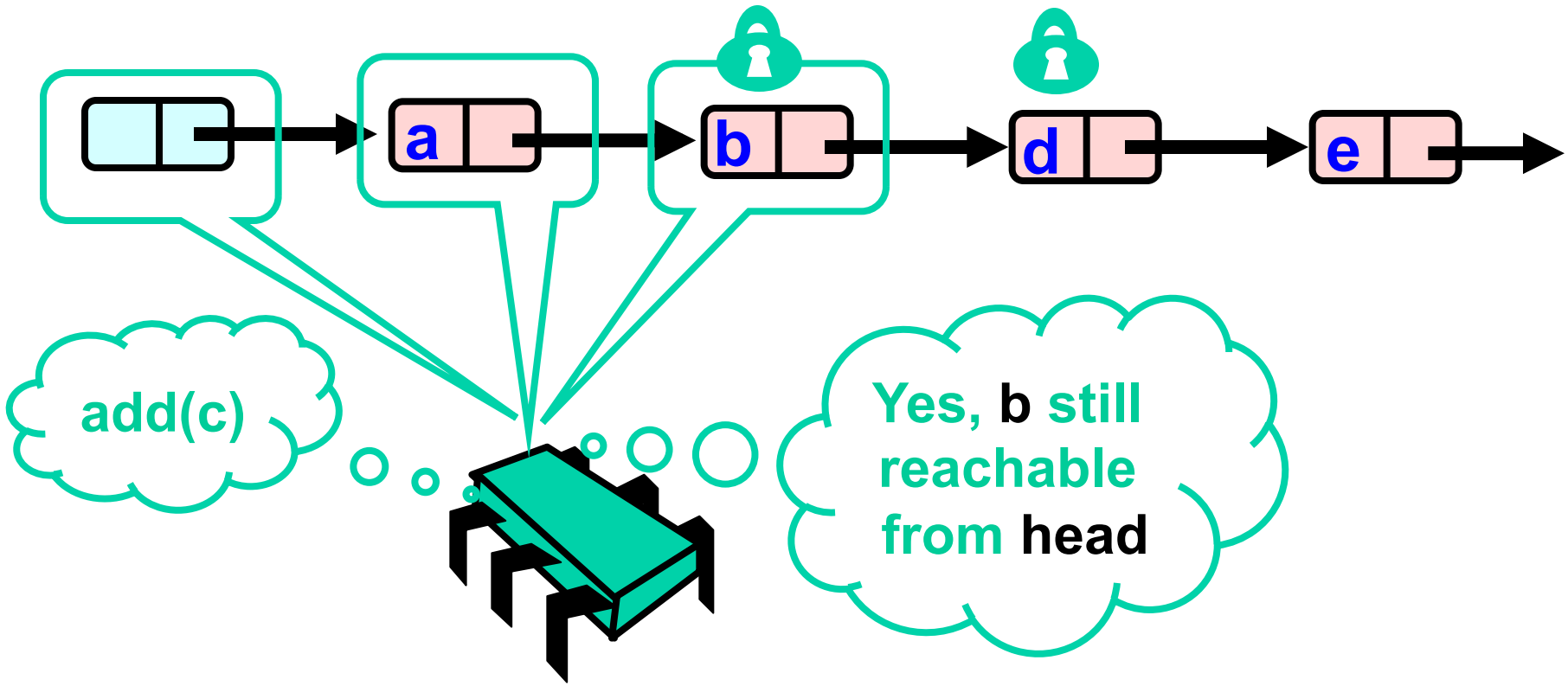
What could go wrong?



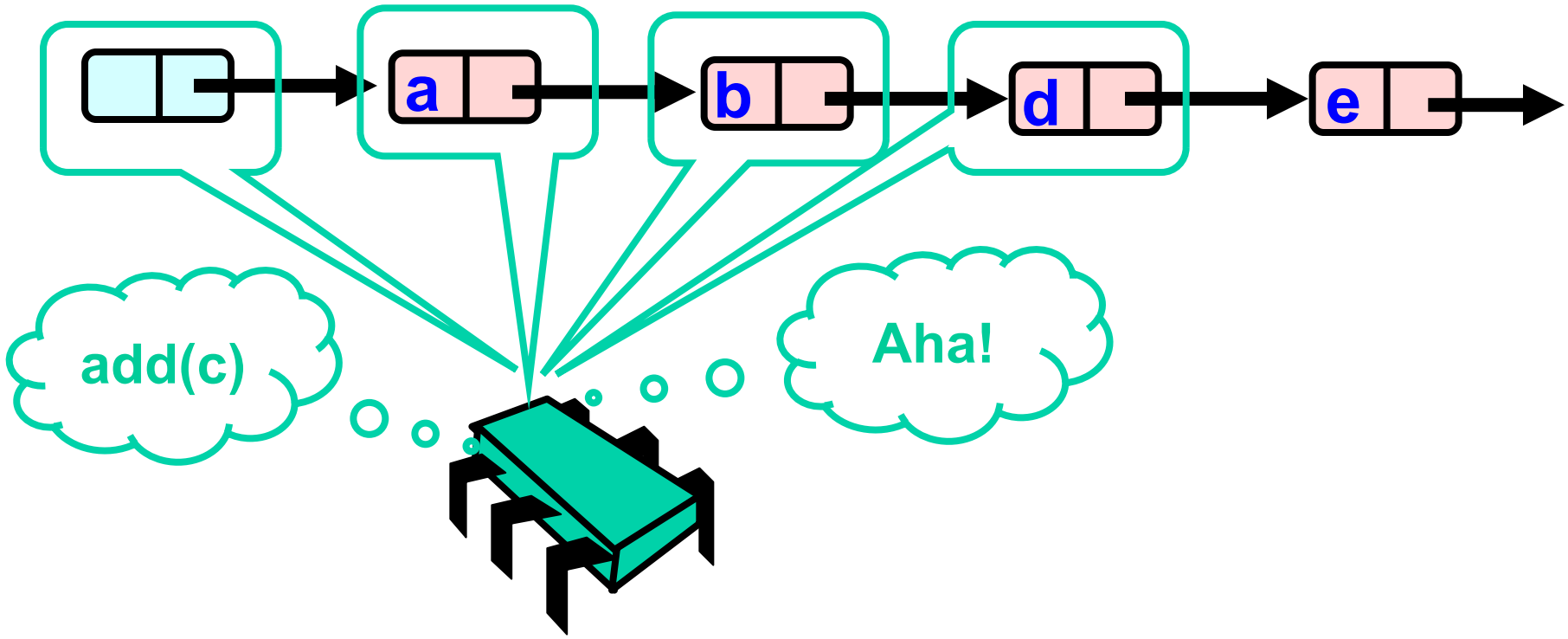
What could go wrong?



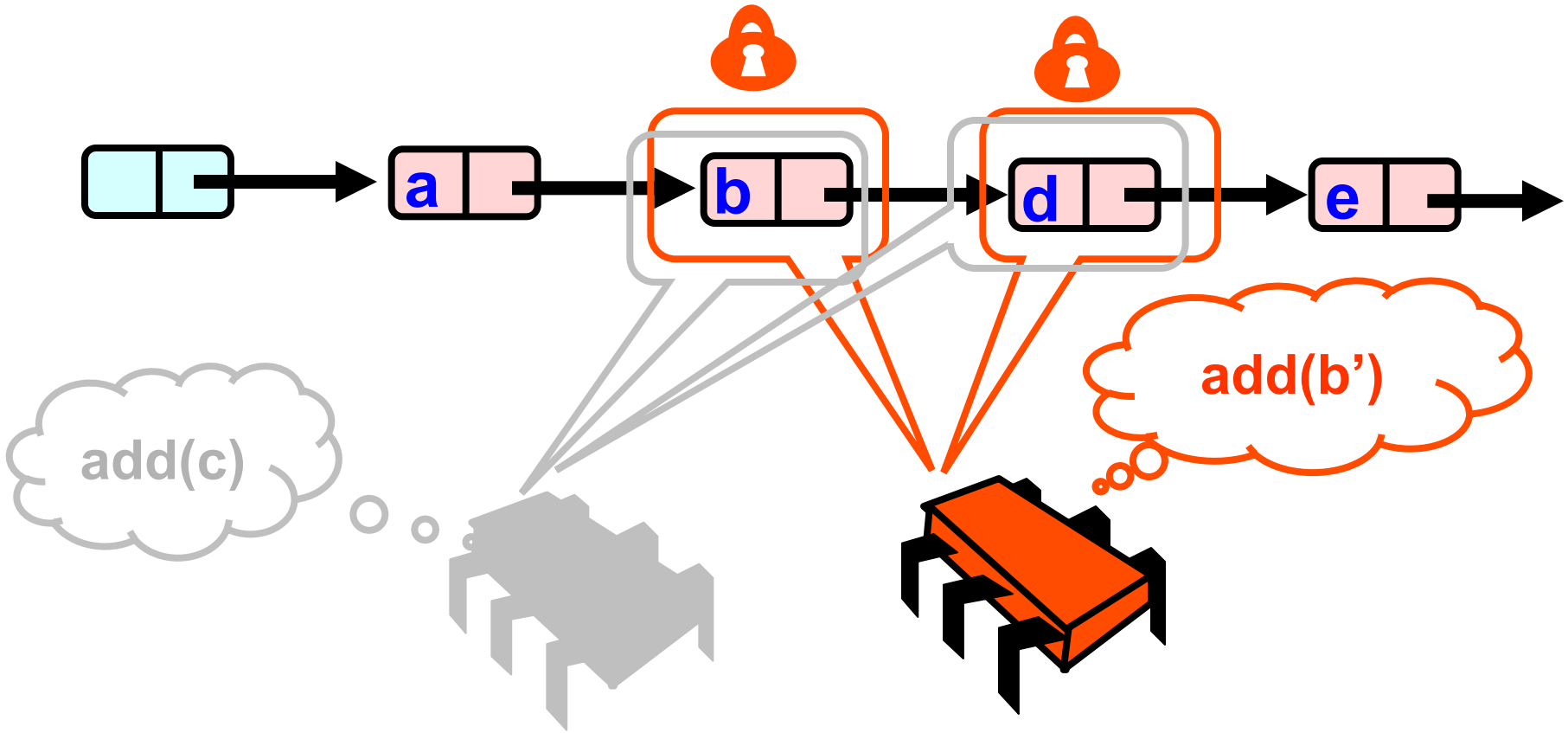
Validate – Part 1



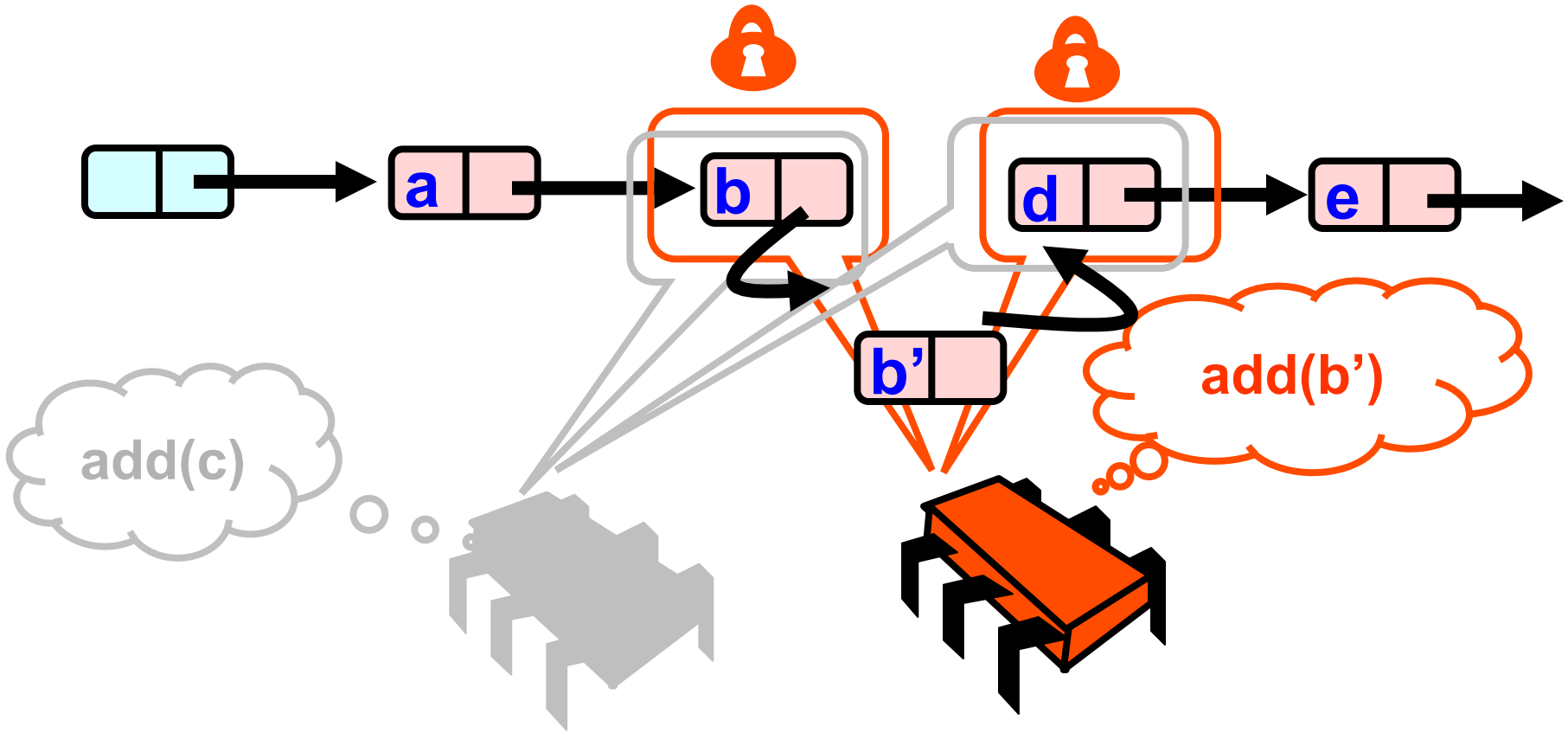
What Else Could Go Wrong?



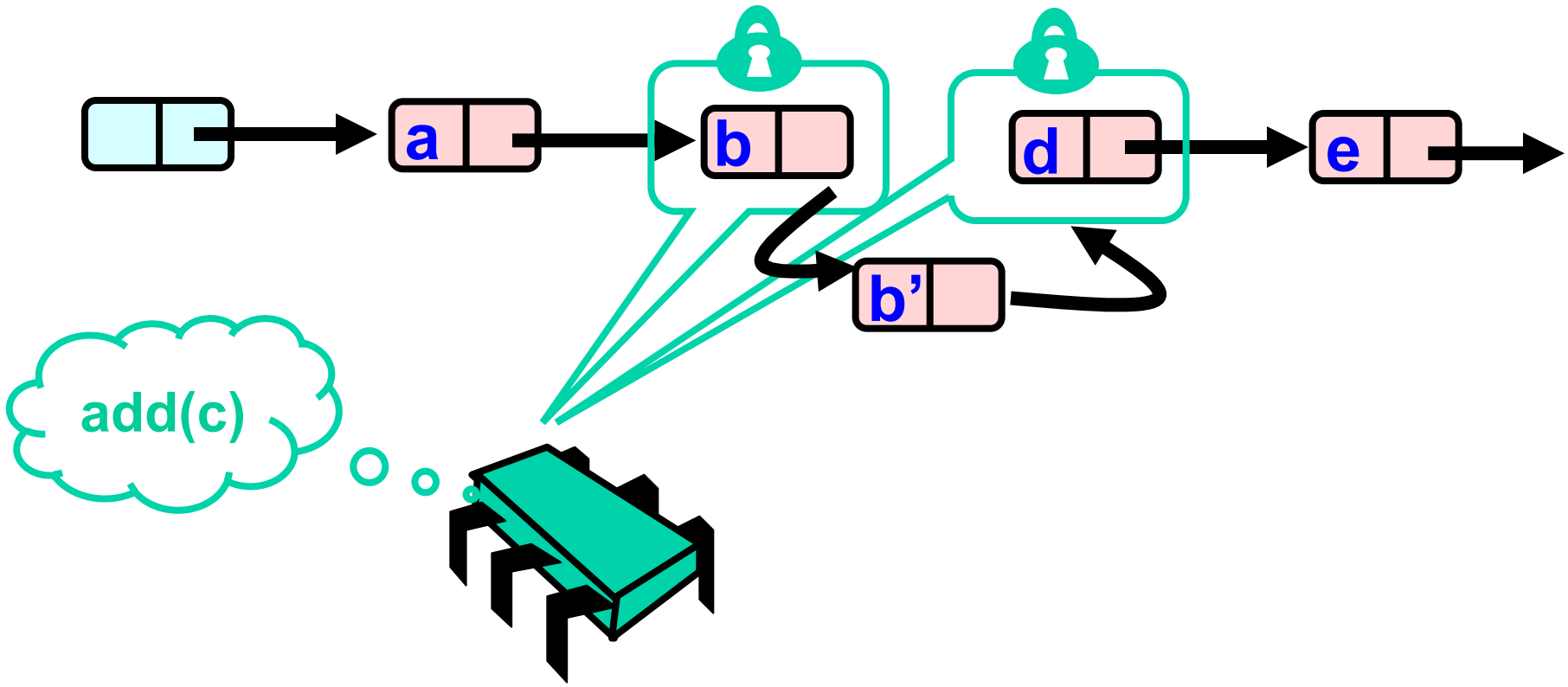
What Else Coould Go Wrong?



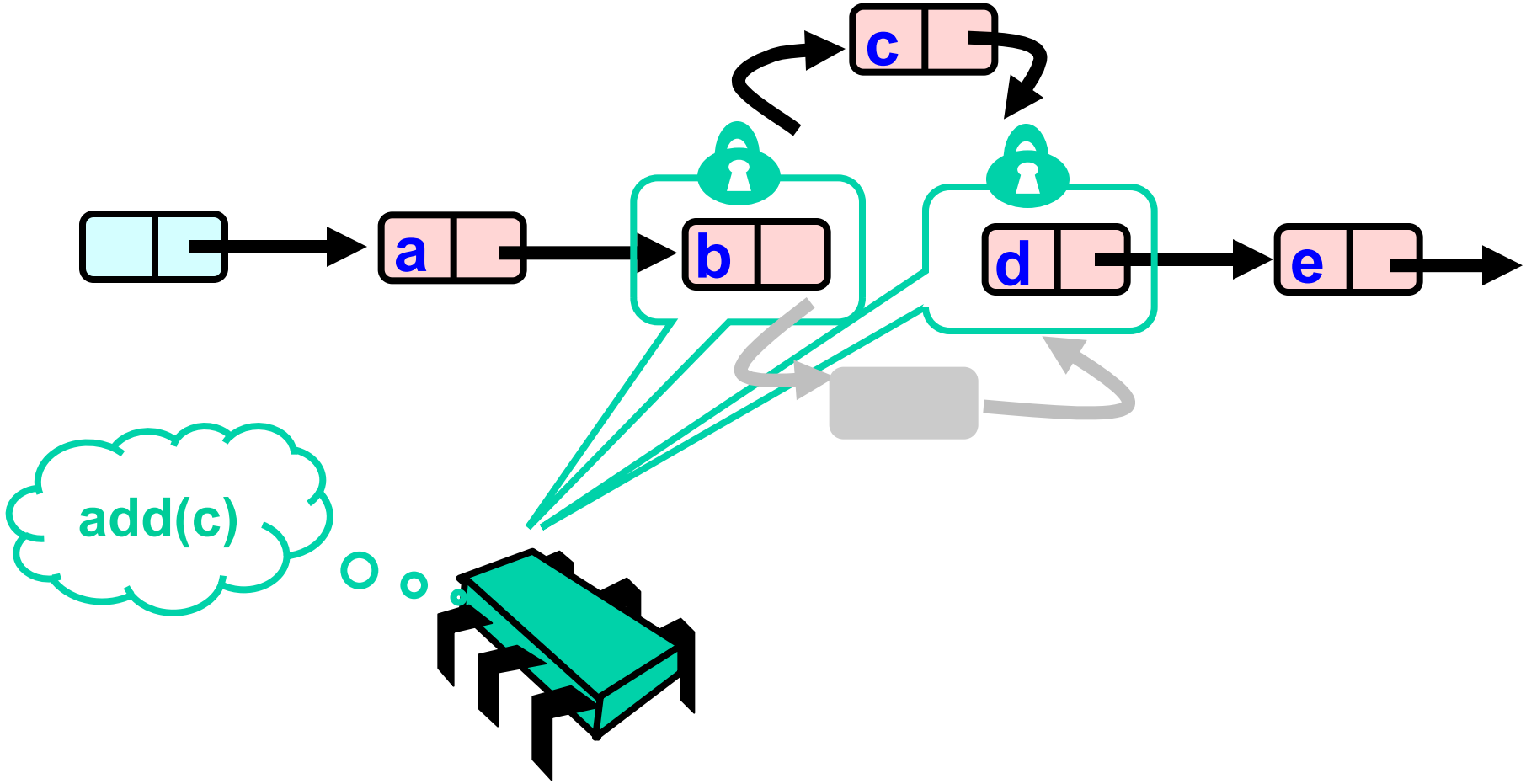
What Else Coould Go Wrong?



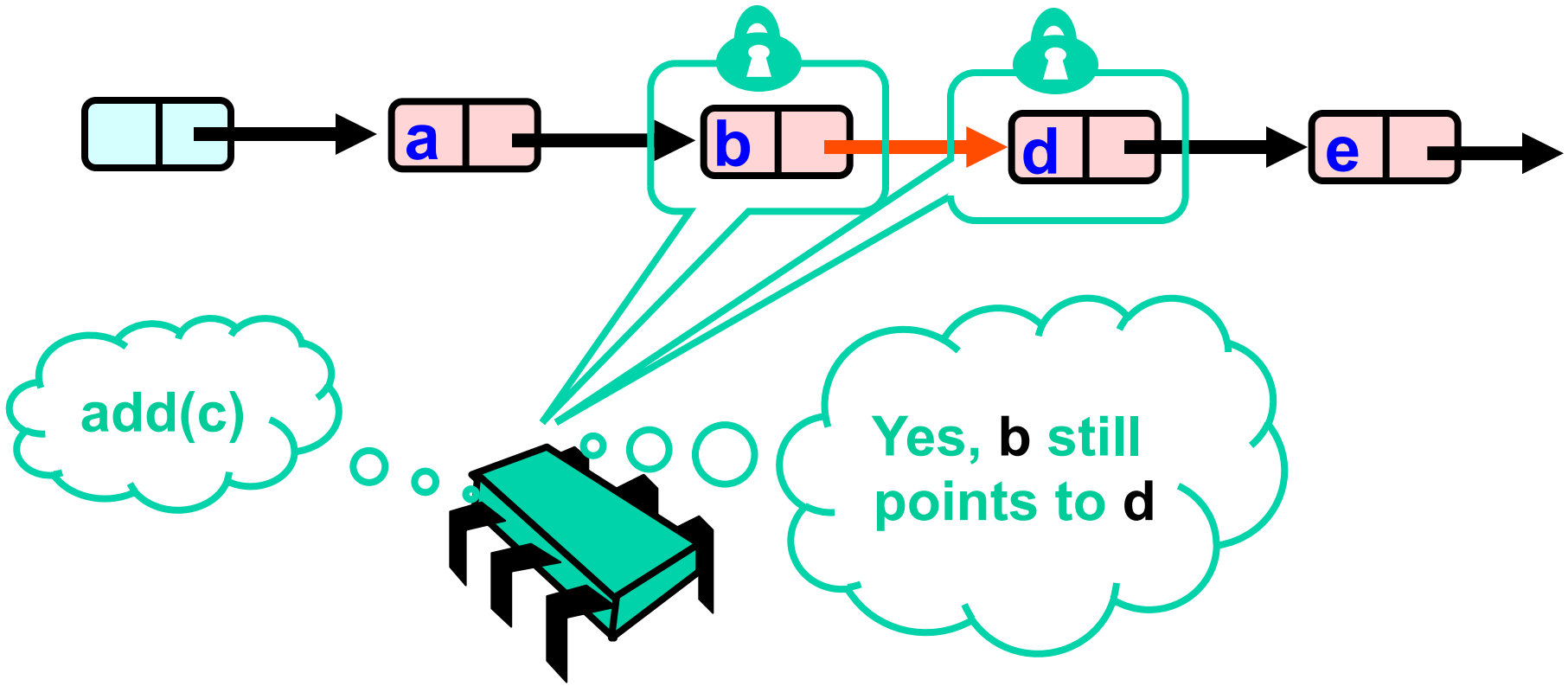
What Else Could Go Wrong?



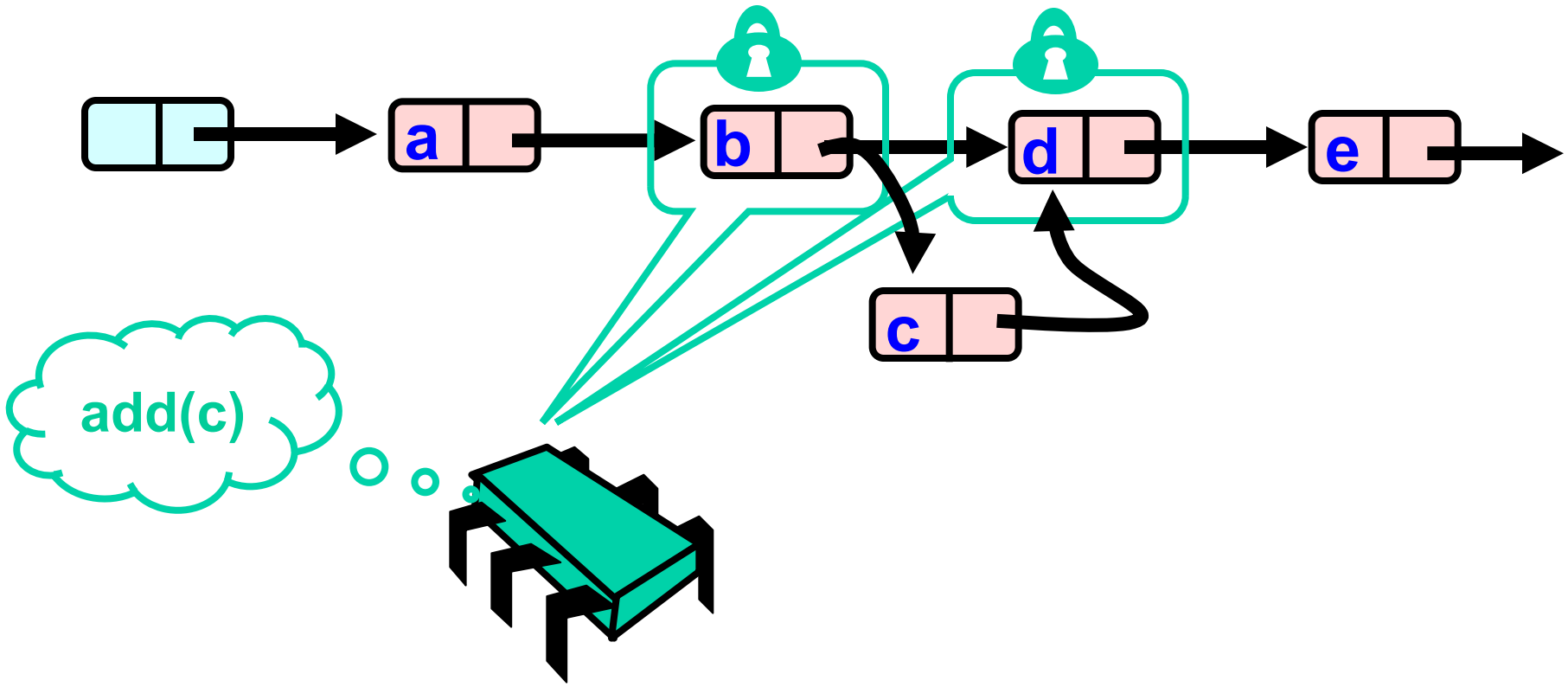
What Else Could Go Wrong?



Validate Part 2
(while holding locks)



Optimistic: Linearization Point



Same Abstraction Map

- S(head) =

  { x | there exists a such that

  - a reachable from head and
  - a.item = x

  }

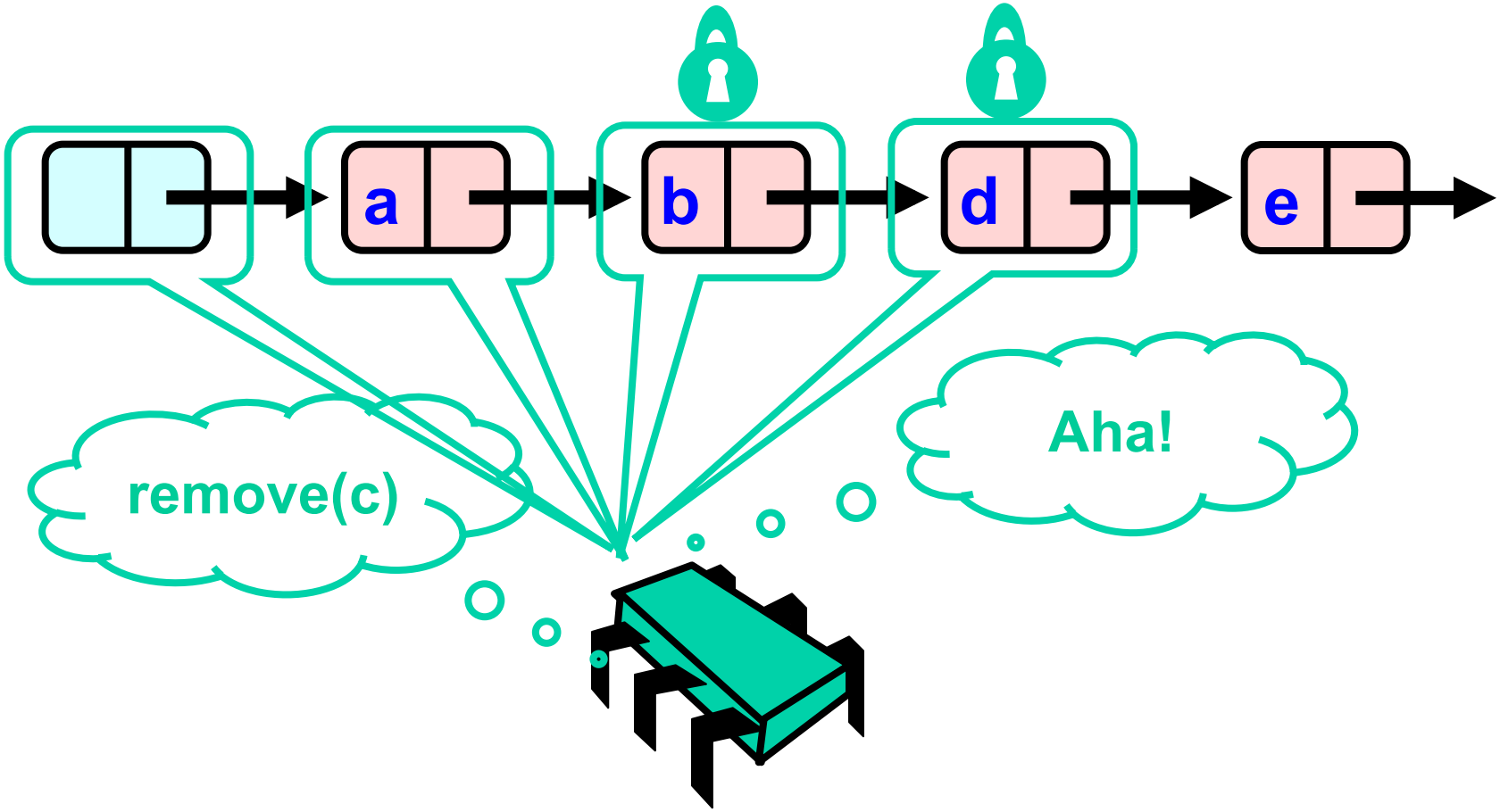
Invariants

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

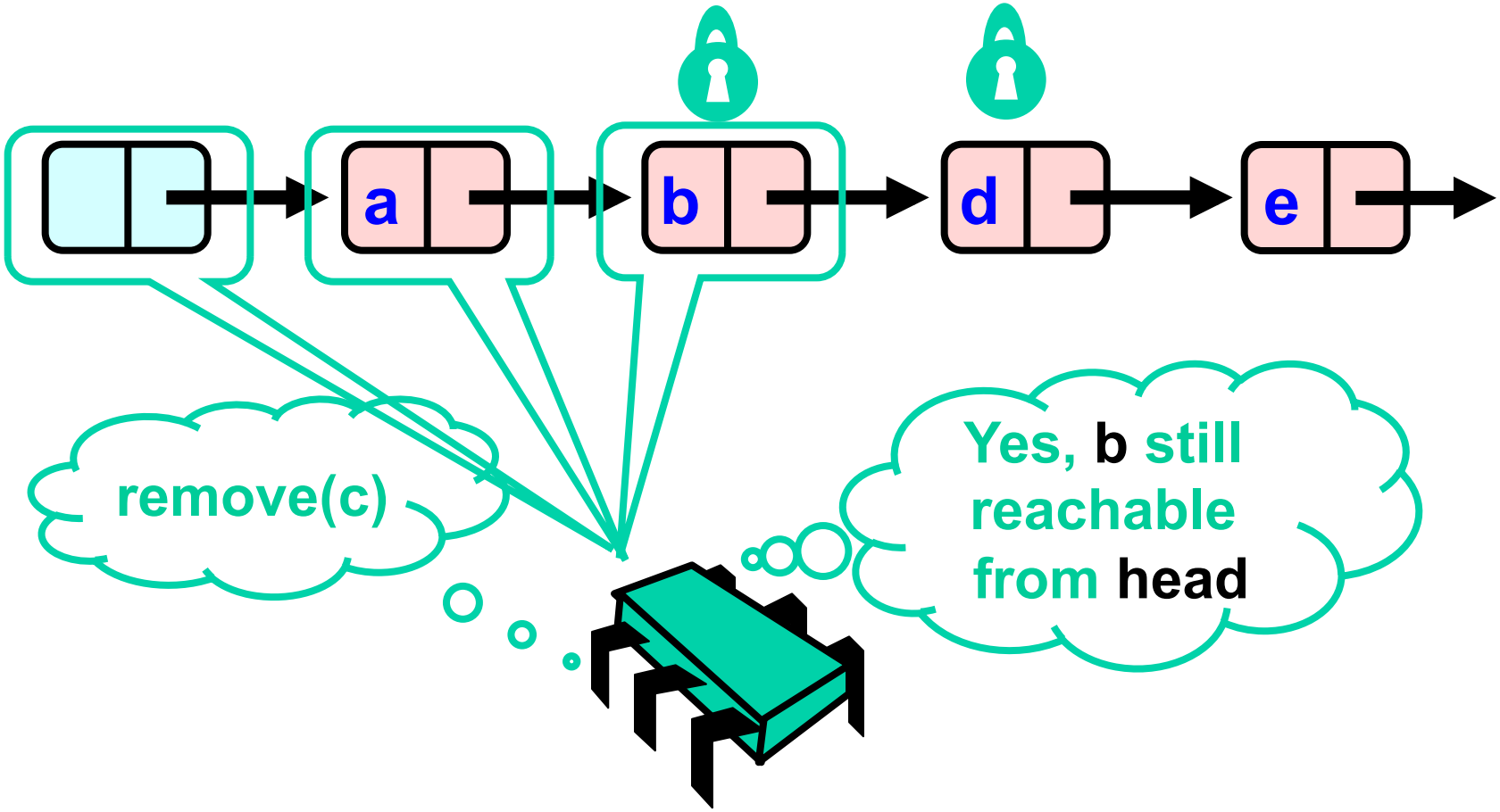
Correctness

- 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

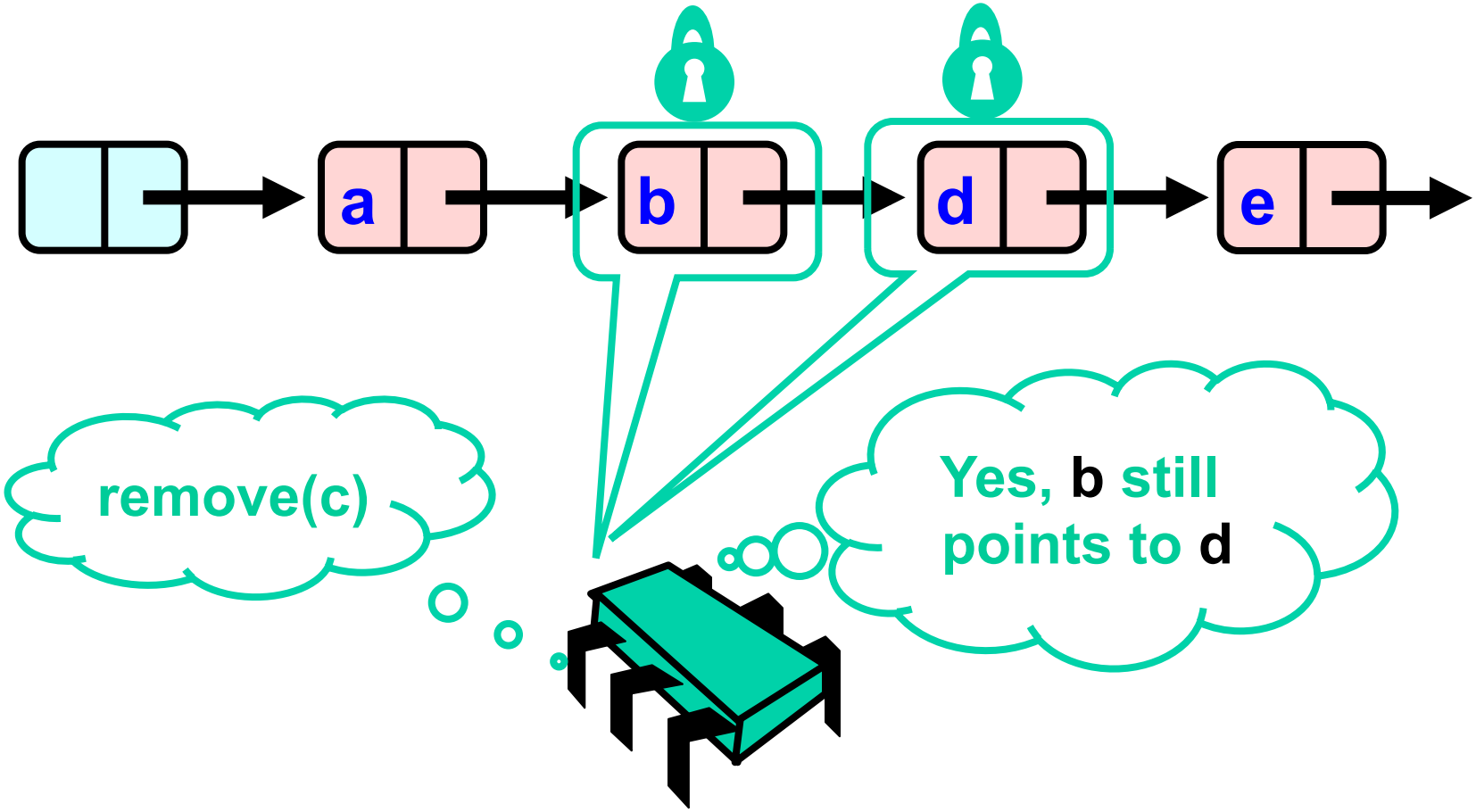
Unsuccessful Remove



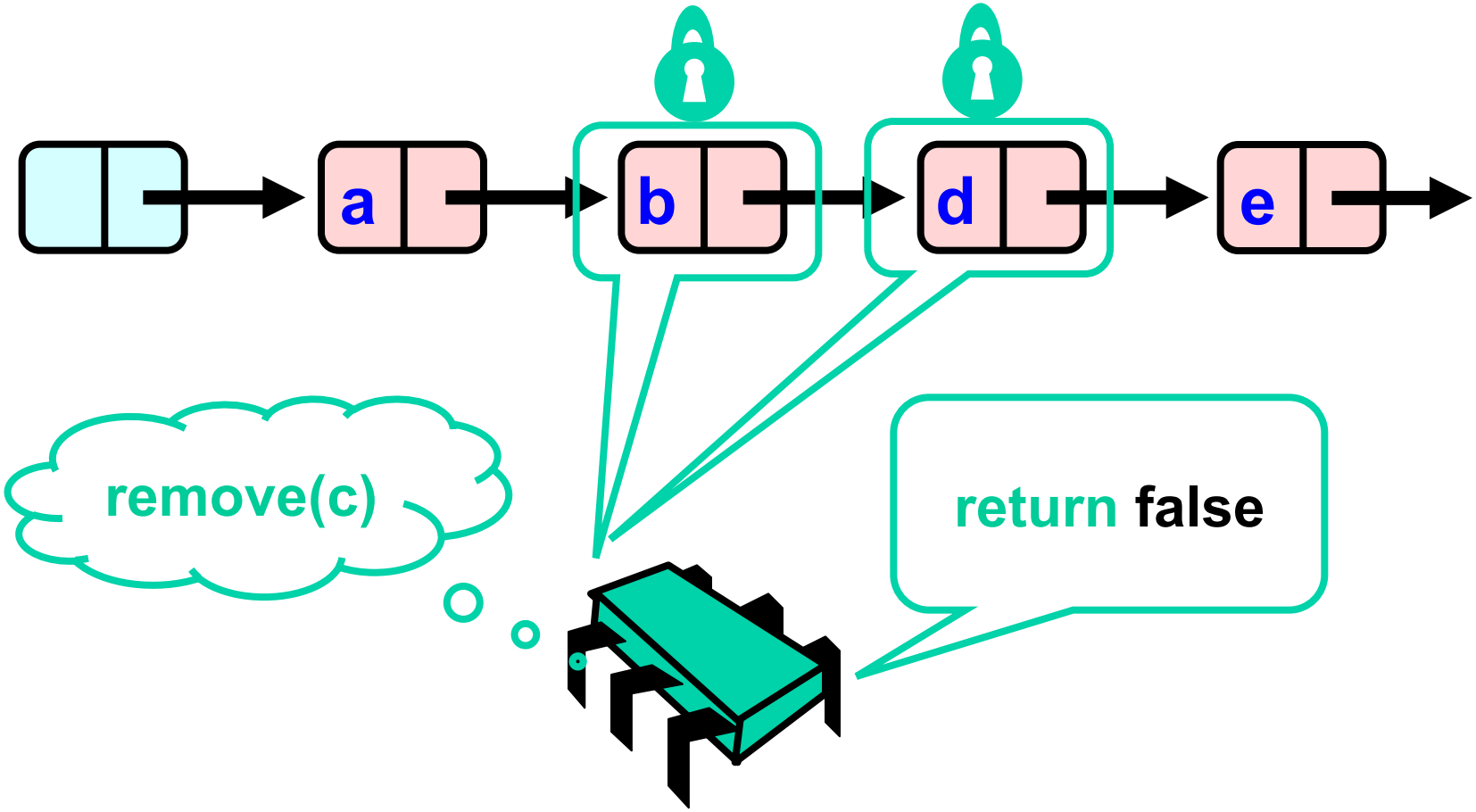
Validate (1)



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

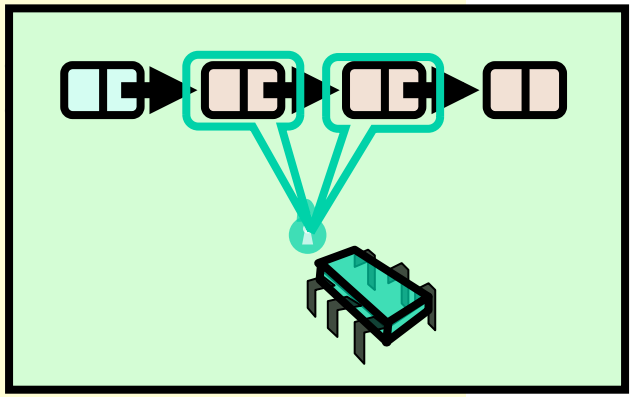
Validation

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

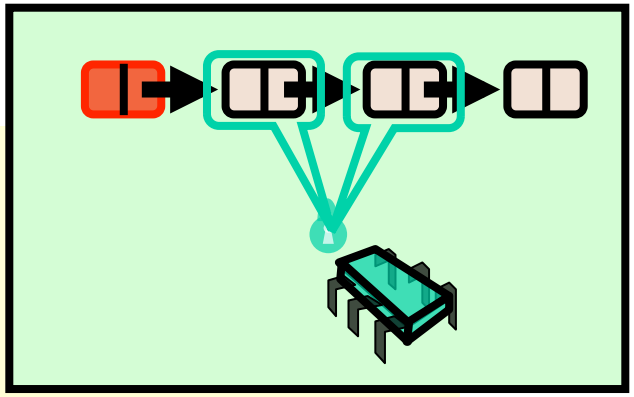
Validation

```
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



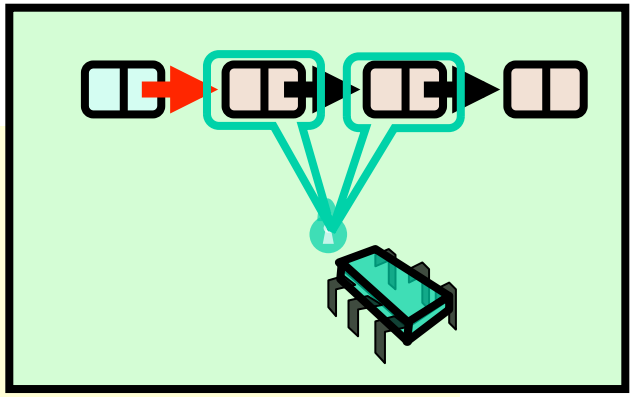
Validation



```
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

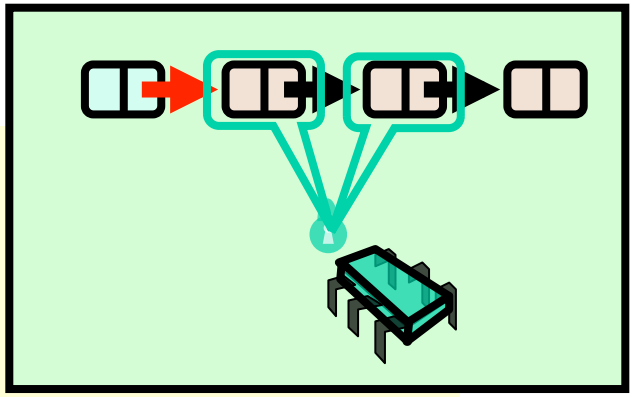
Validation



```
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

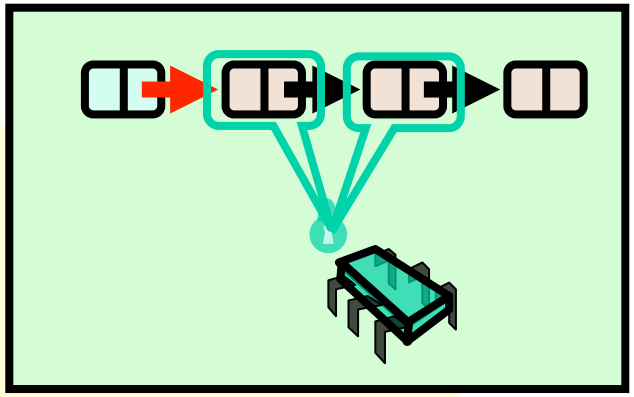
Validation



```
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



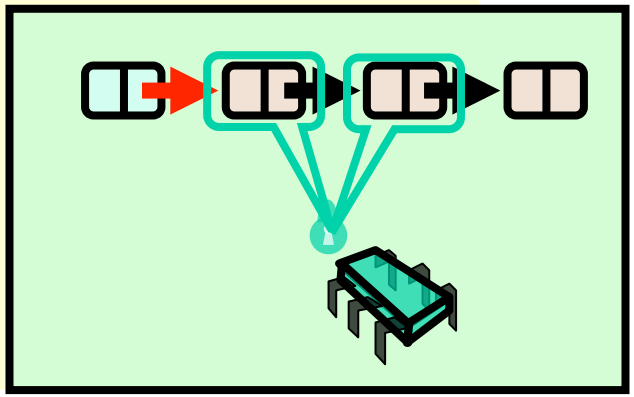
```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Is current node next?

Validation

```
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;
}
```

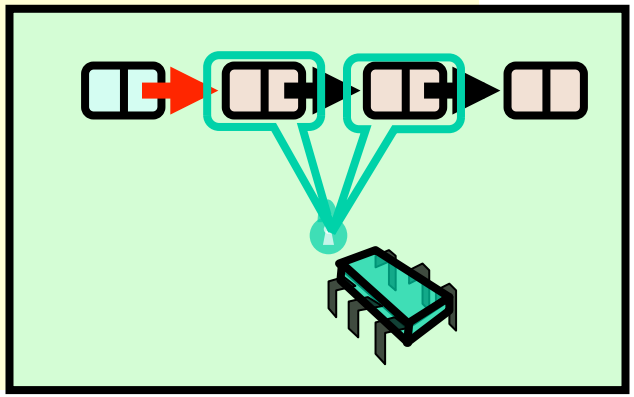
Otherwise move on



Validation

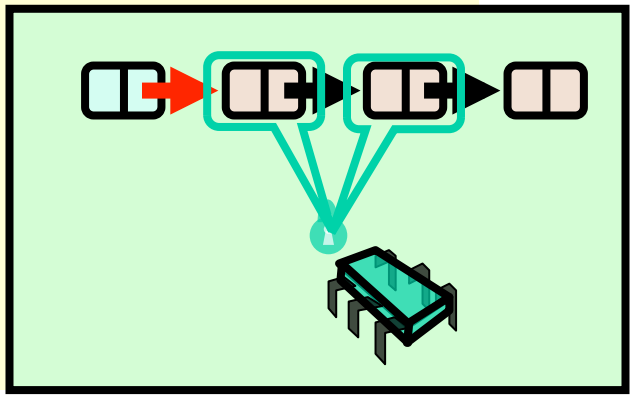
```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Predecessor not reachable



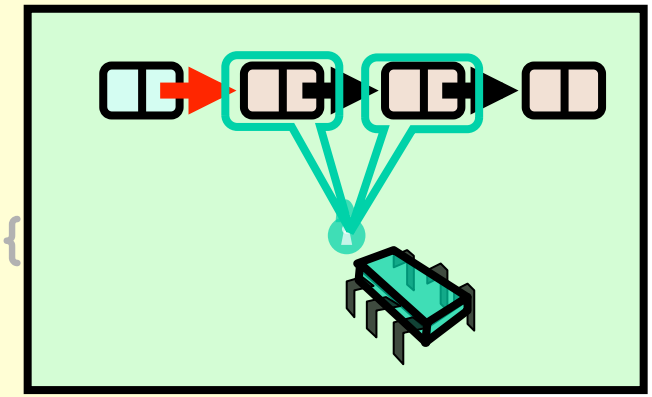
Remove: the Traversal

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



Remove: the Traversal

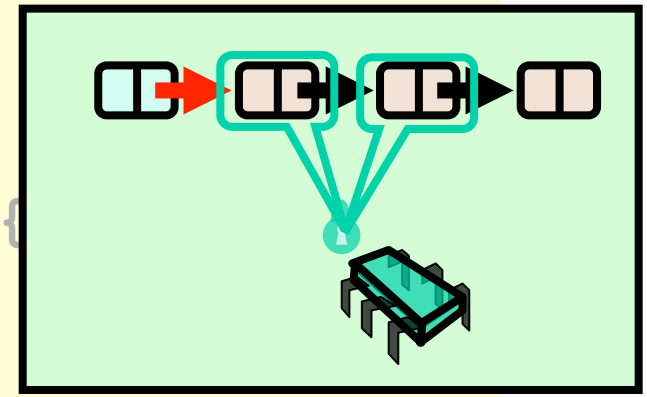
```
public boolean remove(T item) {
 int key = item.hashCode();
 retry: while (true) {
   Node pred = head;
   Node curr = pred.next;
   while (curr.key <= key) {
    if (item == curr.item)
     break;
    pred = curr;
    curr = curr.next;
   } …
```



Search key

Remove: the Traversal

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

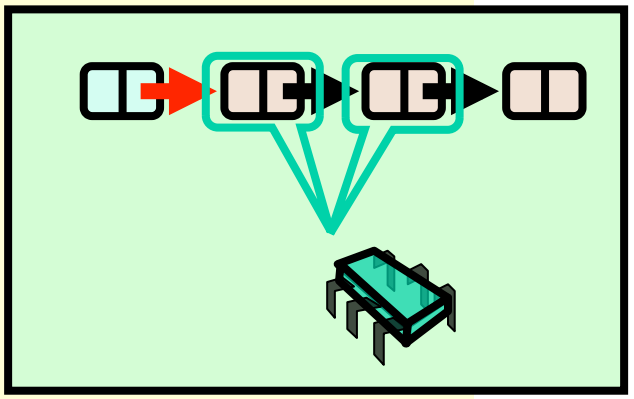


Retry on synchronization conflict

(If validation fails, we come back here.)

Remove: the Traversal

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

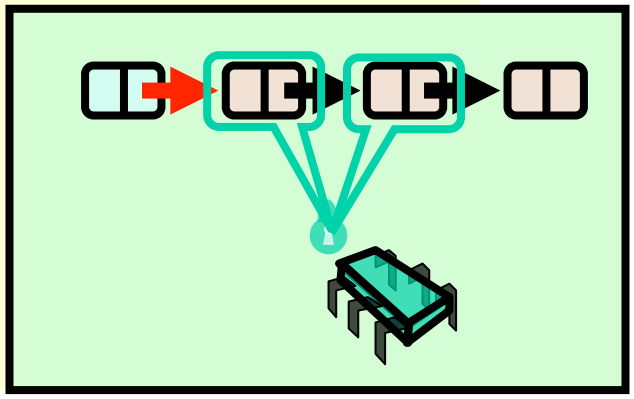


Examine predecessor and current nodes

Remove: the Traversal

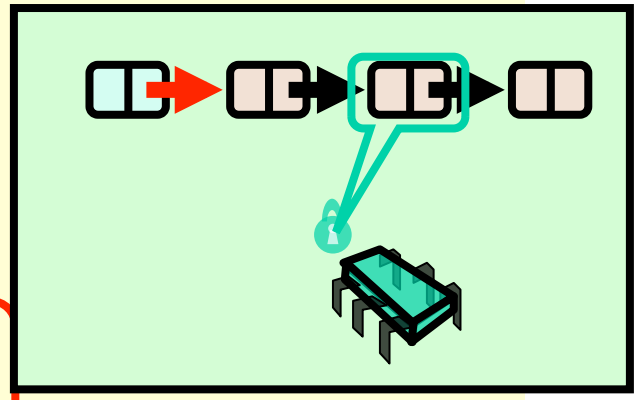
```
public boolean remove(T item) {
 int key = item.hashCode();
 retry: while (true) {
   Node pred = head;
   Node curr = pred.next;
   while (curr.key <= key) {
     if (item == curr.item)
      break;
     pred = curr;
     curr = curr.next;
   } …
```

Search by key



Remove: the Traversal

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

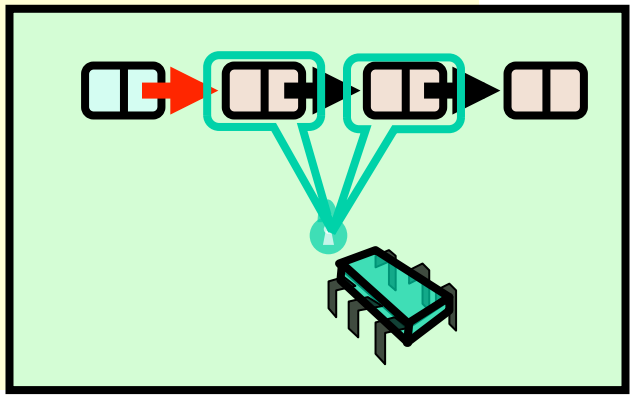


Stop if we find item

Remove: the Traversal

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

Move along



On Exit from Inner 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 Continue: the Deletion (After Existing Inner Loop)

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

Remove Continue: the Deletion (After Existing Inner Loop)

```
pred.lock(); curr.lock();
try {
   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 Continue: the Deletion (After Existing Inner Loop)

```
pred.lock(); curr.lock();
try {
  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 Continue: the Deletion (After Existing Inner Loop)

```
pred.lock(); curr.lock();
try {
  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 Continue: the Deletion (After Existing Inner Loop)

```
pred.lock(); curr.lock();
try {
  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
  - Holding locks only on the targets of add(), remove(), contains()
  - No contention on traversals
  - Traversals are "wait-free" (What's wait free?)

Progress Conditions

- ***Deadlock-free:*** some thread trying to acquire the lock eventually succeeds.
- ***Starvation-free:*** every thread trying to acquire the lock eventually succeeds.
- ***Lock-free:*** some thread calling a method eventually returns.
- ***Wait-free:*** every thread calling a method eventually returns.

Progress Conditions

	Non-Blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free

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

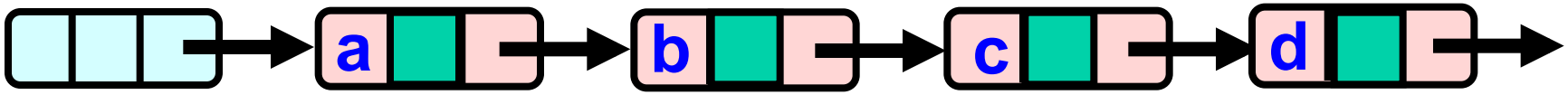
Lazy List

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

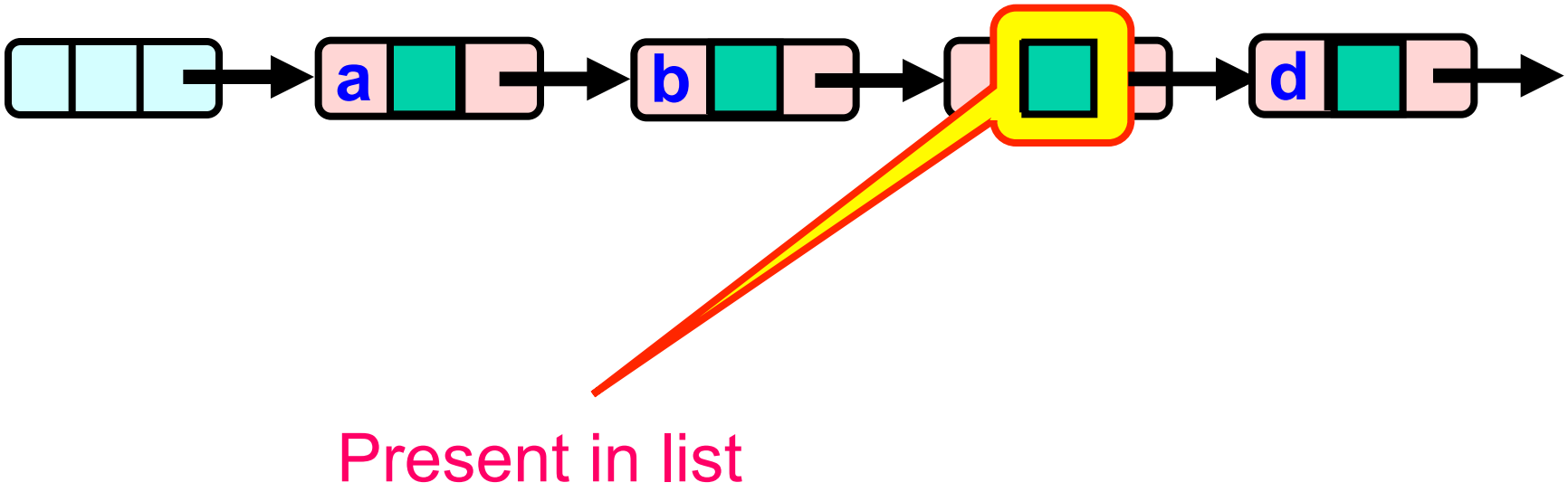
Lazy List

- **`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)

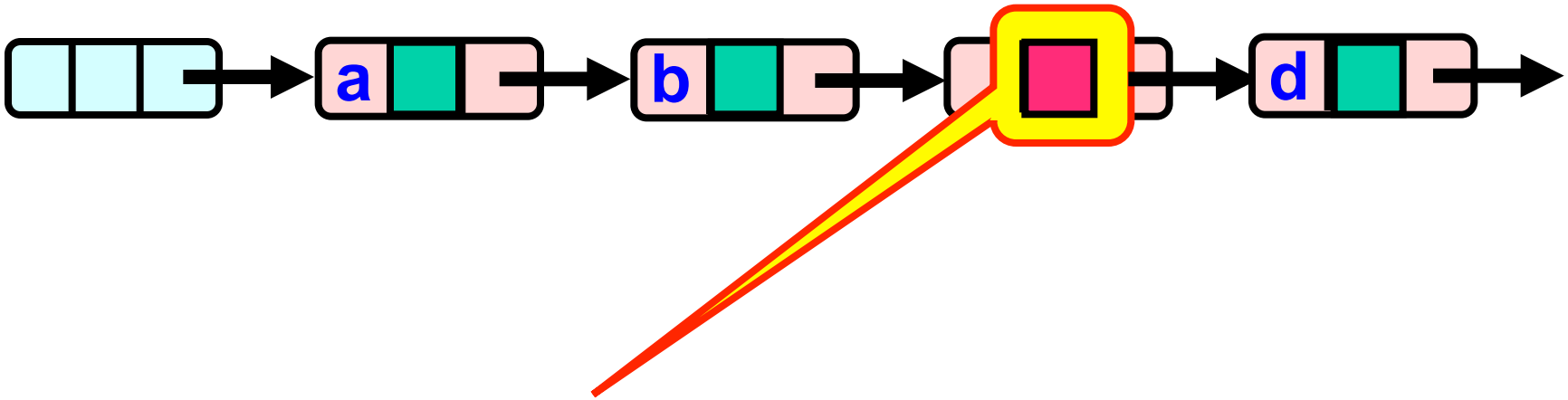
Lazy Removal



Lazy Removal

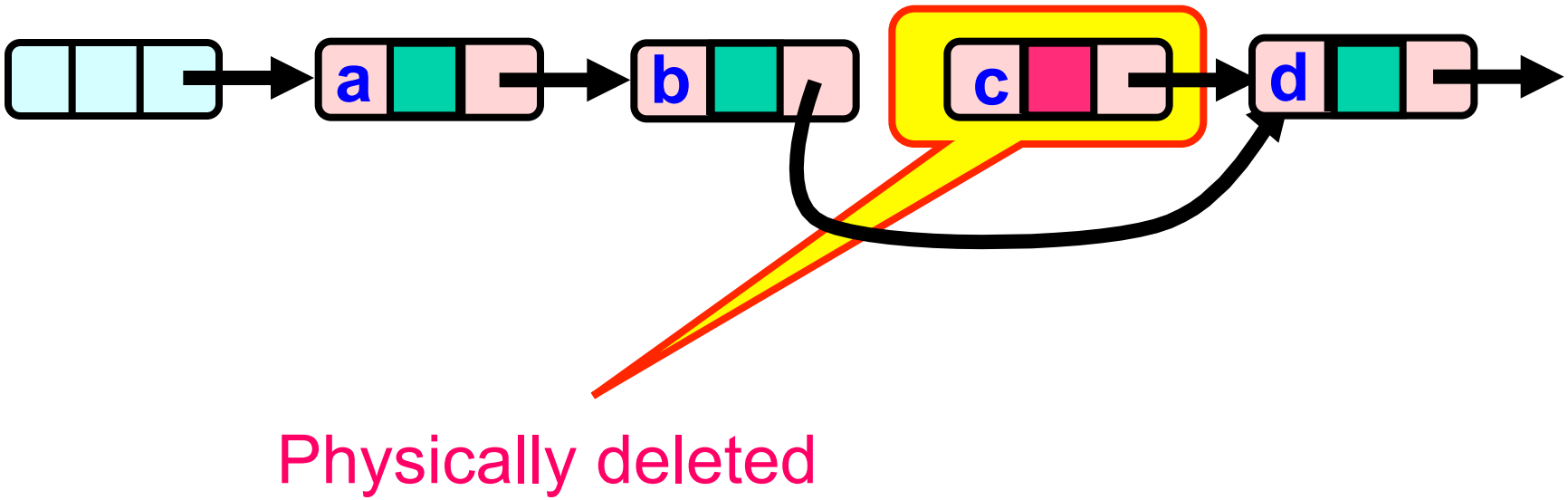


Lazy Removal

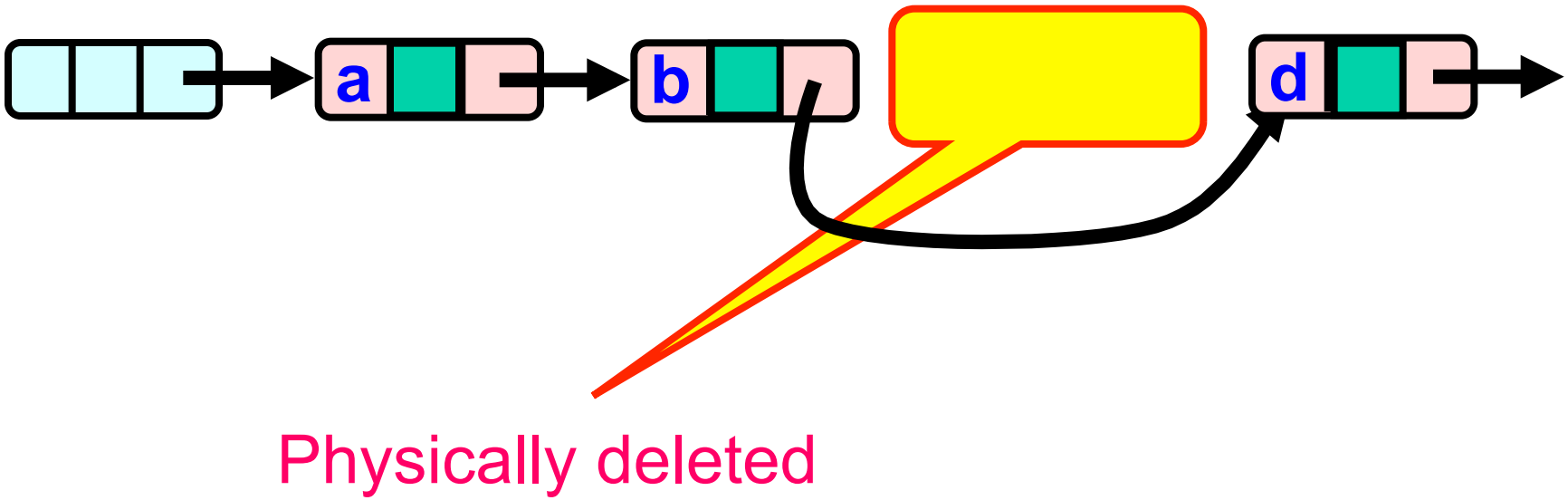


Logically deleted

Lazy Removal



Lazy Removal



Lazy List

- All Methods
  - Scan through locked and marked nodes
  - add and remove still locks pred and curr, but not contain
  - Adding / removing a node doesn't slow down contain() …

Lazy List Validation

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

New Abstraction Map

- S(head) =

  { x | there exists node a such that

  - a reachable from head and
  - a.item = x and
  - a is unmarked

  }

Invariant

- If an item is not marked, it is reachable from head and still in the set.
- Any unmarked reachable node remains reachable even if its predecessor is logically or physically removed

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

Validation

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

Both the next and marked fields need to be volatile!

Remove: the Deletion

```
... // the traversal
pred.lock(); curr.lock();
try {
  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: the Deletion

```
... // the traversal
pred.lock(); curr.lock();
try {
  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: the Deletion

```
... // the traversal
pred.lock(); curr.lock();
try {
  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

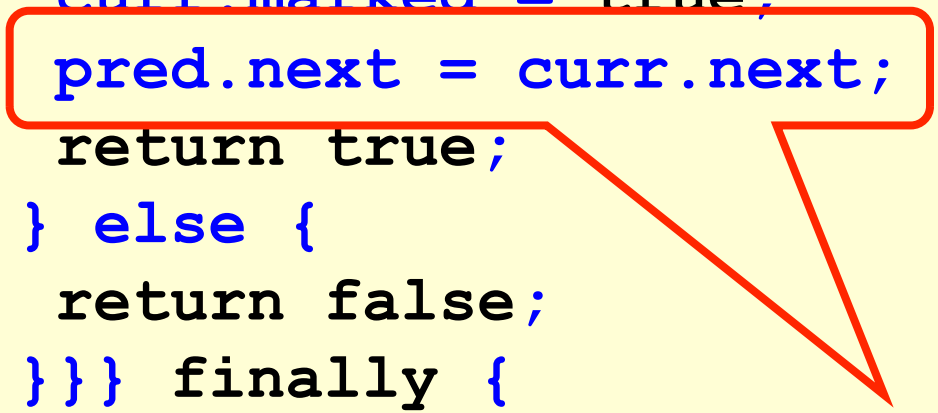
Remove: the Deletion

```
... // the traversal
pred.lock(); curr.lock();
try {
  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: the Deletion

```
... // the traversal
pred.lock(); curr.lock();
try {
  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();
   }}}
```

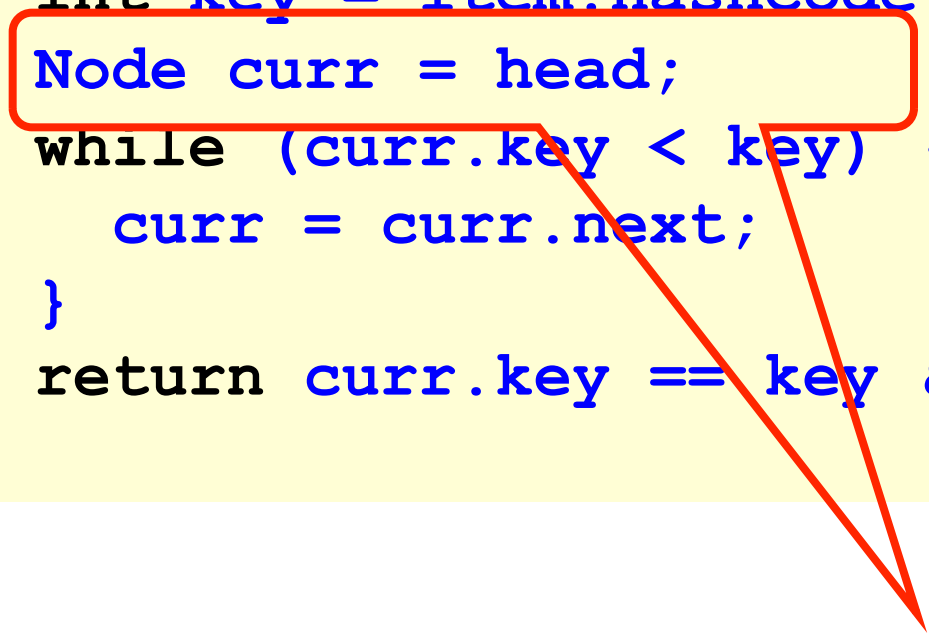


Contains

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

Contains

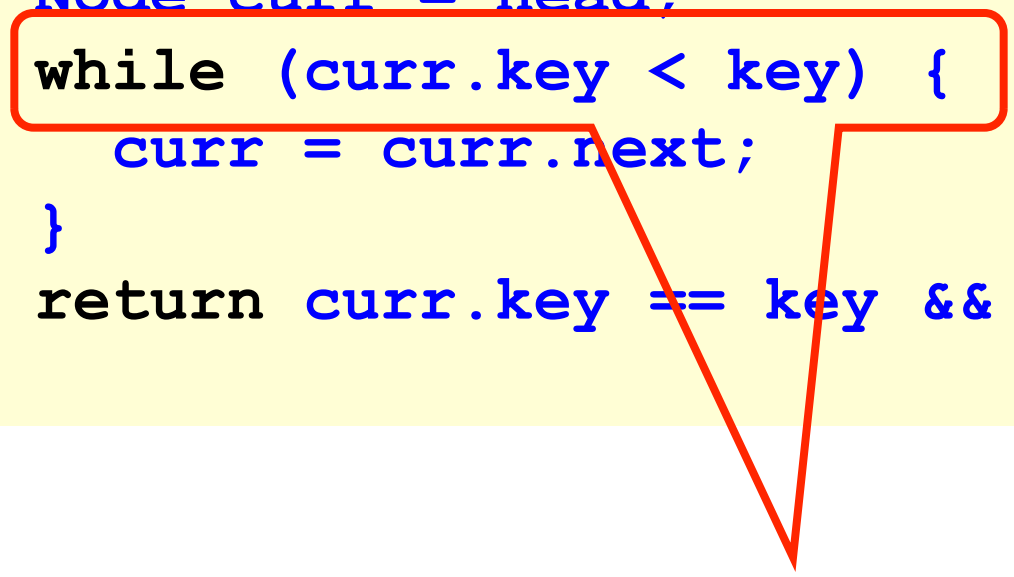
```
public boolean contains(T item) {
  int key = item.hashCode();
  Node curr = head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```



Start at the head

Contains

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



Search key range

Contains

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

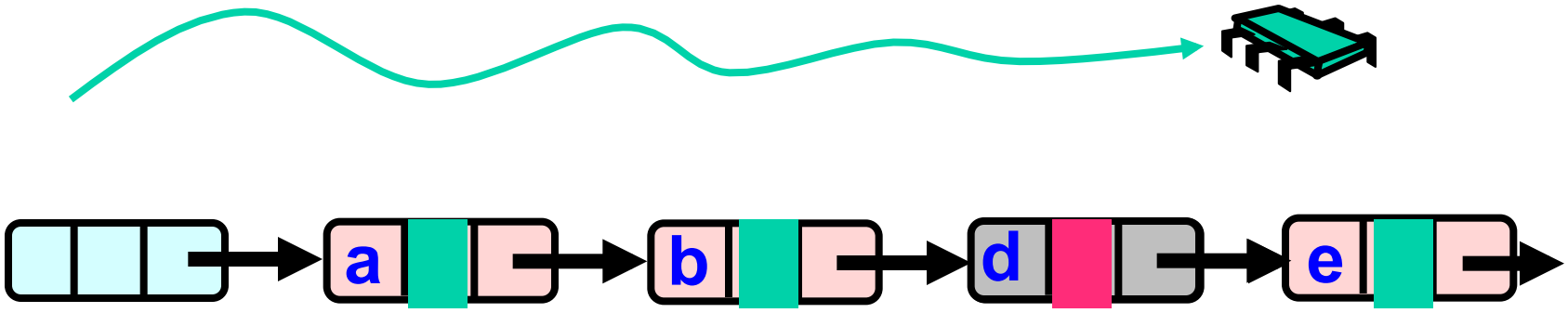
Traverse *without locking*
(nodes may have been removed)

Contains

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

Present and undeleted?

Summary: Wait-free Contains



Use Mark bit + list ordering

1. Not marked → in the set
2. Marked or missing → not in the set
3. Traverse the list only once!

Evaluation

- Good:
  - **`contains()`** doesn't lock
  - In fact, its wait-free!
  - Good because typically high % contains()
  - Uncontended calls to add and remove don't re-traverse
- Bad
  - Contended **`add()`** and **`remove()`** calls must 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….

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

Lock-free Lists

- Next logical step
  - Wait-free **`contains()`**
  - lock-free **`add()`** and **`remove()`**
- Use only **`compareAndSet()`**
  - What could go wrong?

compareAndSet

```
public abstract class CASObject {
 private int value;
 public boolean synchronized
   compareAndSet(int expected,
                 int update) {
  if (value==expected) {
   value = update; return true;
  }
  return false;
  } … }
```

compareAndSet

```
public abstract class CASObject {
 private int value;
 public boolean synchronized
   compareAndSet(int expected,
                 int update) {
  if (value==expected) {
  value = update; return true;
  }
 return false;
 } … }
```

If value is as expected, …

compareAndSet

```
public abstract class CASOBJECT{
 private int value;
 public boolean synchronized
   compareAndSet(int expected,
                 int update) {
 if (value==expected) {
  value = update; return true;
  }
 return false;
 } … }
```

… replace it

compareAndSet

```
public abstract class RMWRegister {
 private int value;
 public boolean synchronized
   compareAndSet(int expected,
                 int update) {
 if (value==expected) {
  value = update; return true;
  }
 return false;
 } … }
```

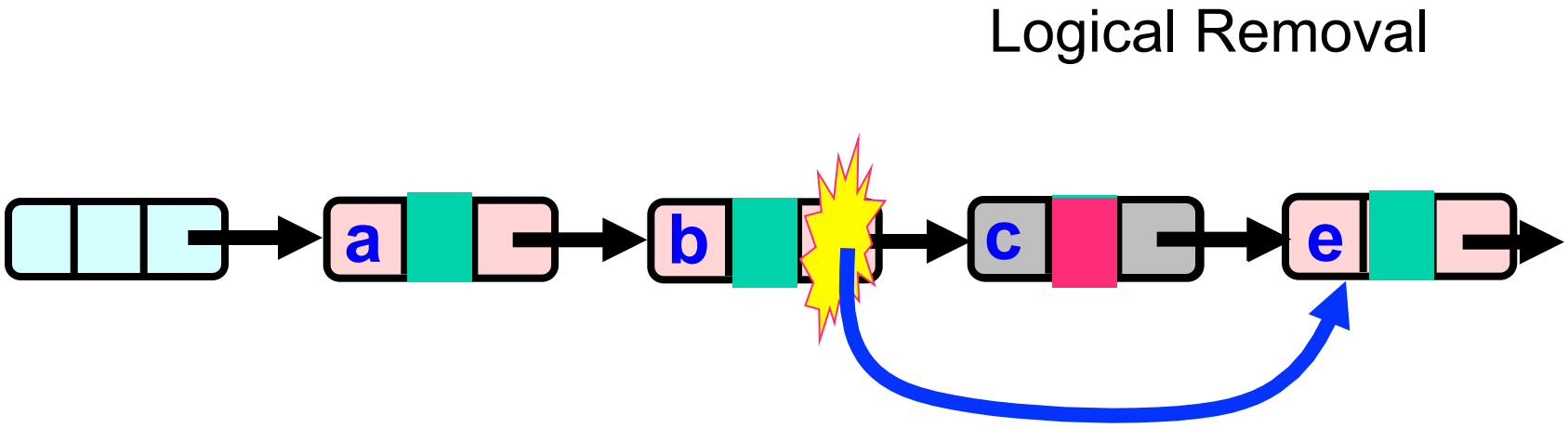
Report success

compareAndSet

```
public abstract class RMWRegister {
 private int value;
 public boolean synchronized
   compareAndSet(int expected,
                 int update) {
 if (value==expected) {
  value = update; return true;
  }
 return false;
 } … }
```

Otherwise report failure

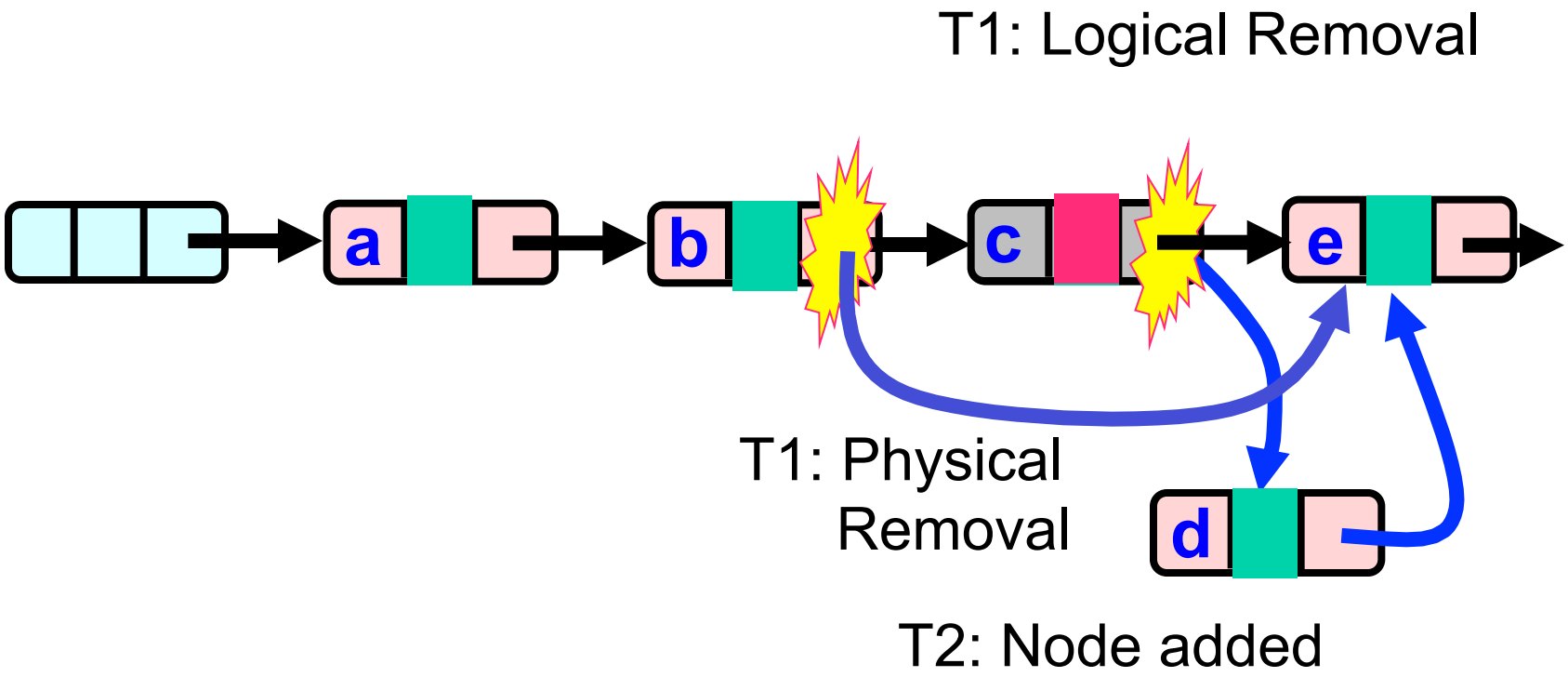
Lock-free Lists



Use CAS to verify pointer is correct

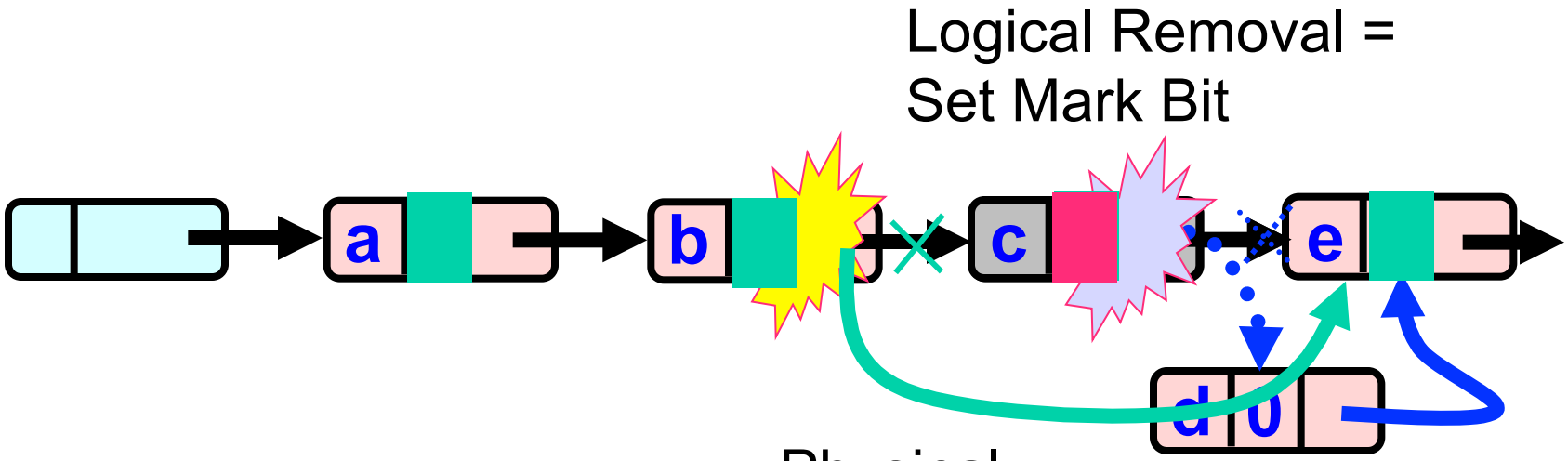
Not enough!

Problem…



Lost Update!

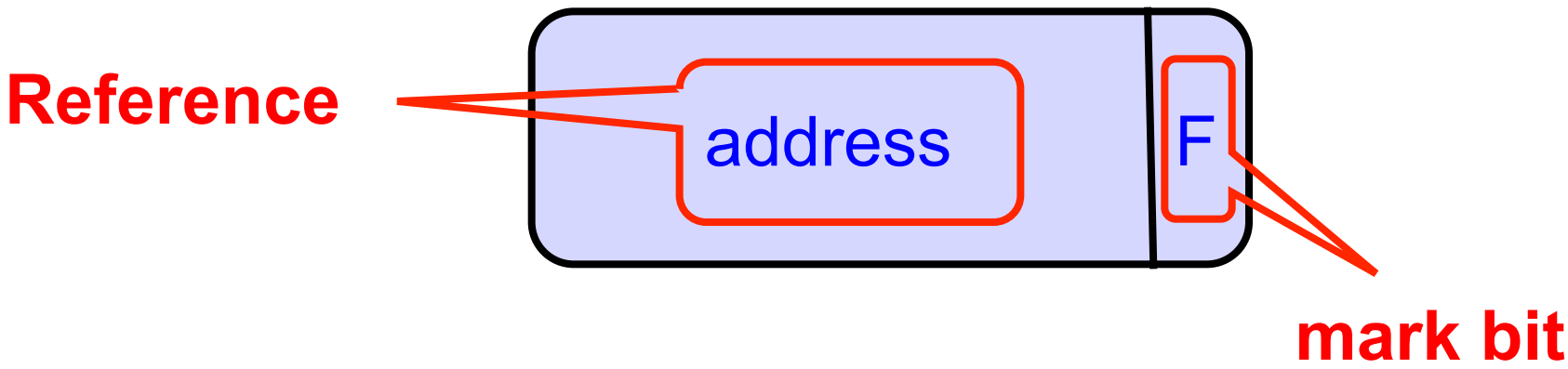
The Solution: Combine Bit and Pointer



Mark-Bit and Pointer are CASed together (AtomicMarkableReference)

Marking a Node

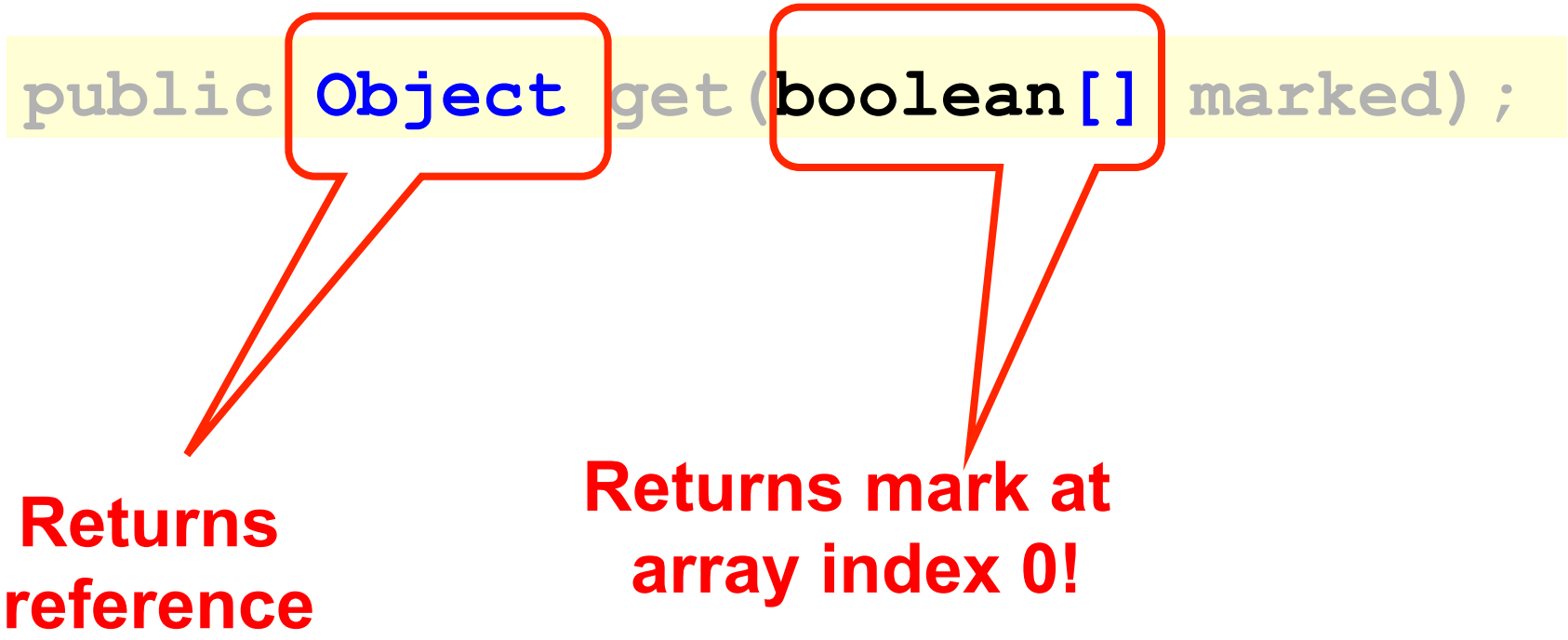
- **AtomicMarkableReference** class
  - Java.util.concurrent.atomic package



Extracting Reference & Mark

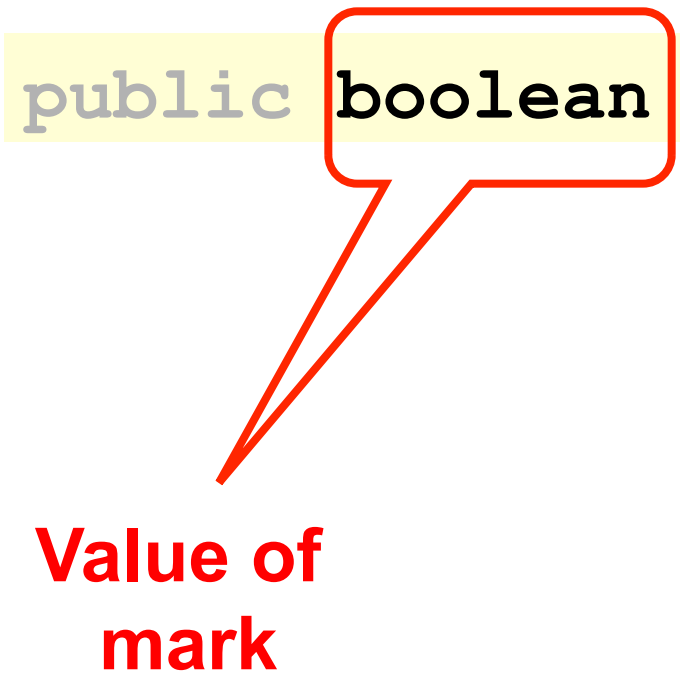
```
public Object get(boolean[] marked);
```

Extracting Reference & Mark



Extracting Mark Only

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



Changing State

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

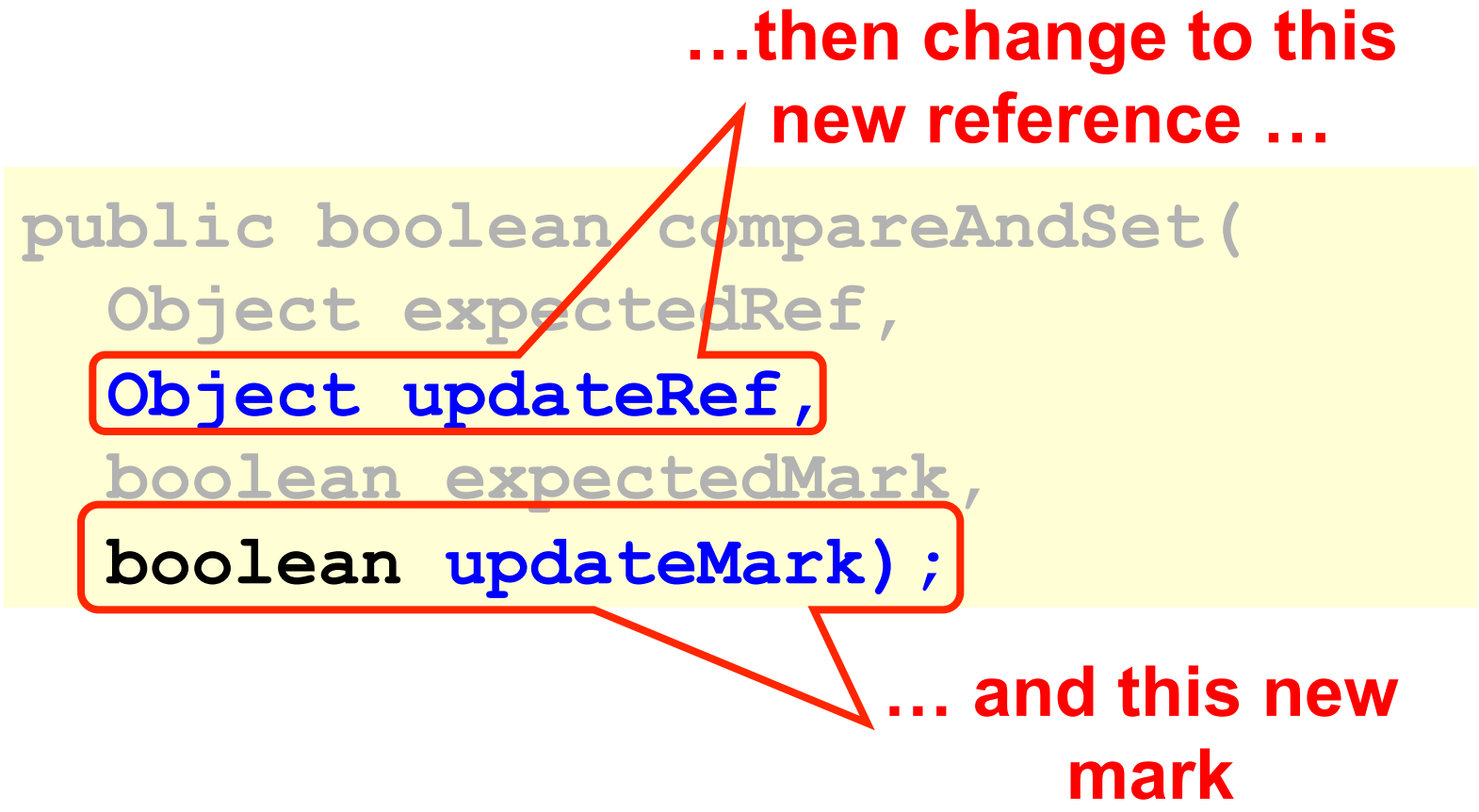
Changing State

If this is the current reference …

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

And this is the current mark …

Changing State



Changing State

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

Changing State

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

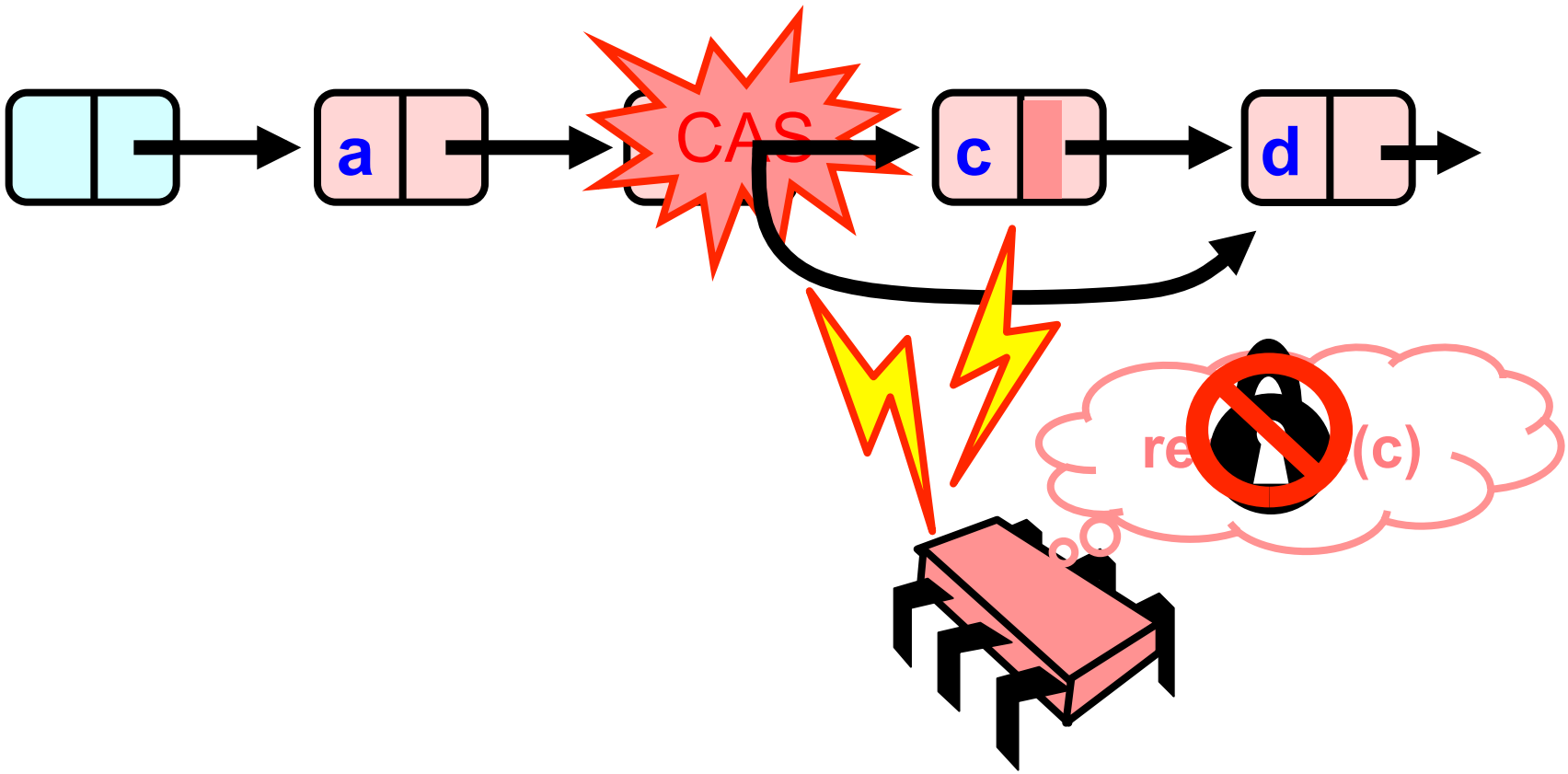
If this is the current reference …

Changing State

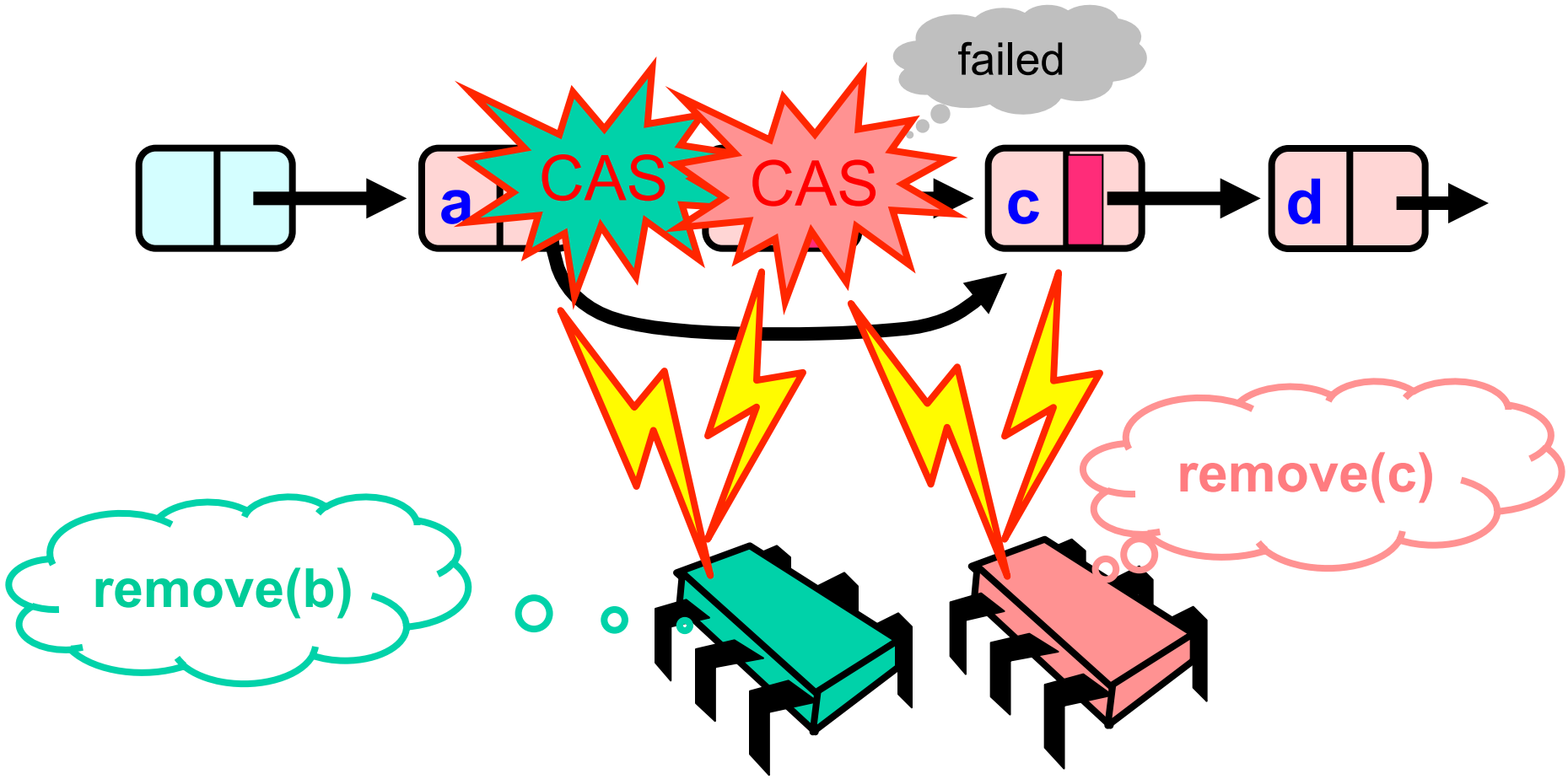
```
public boolean attemptMark(
   Object expectedRef,
   boolean updateMark);
```

.. then change to this new mark.

Removing a Node



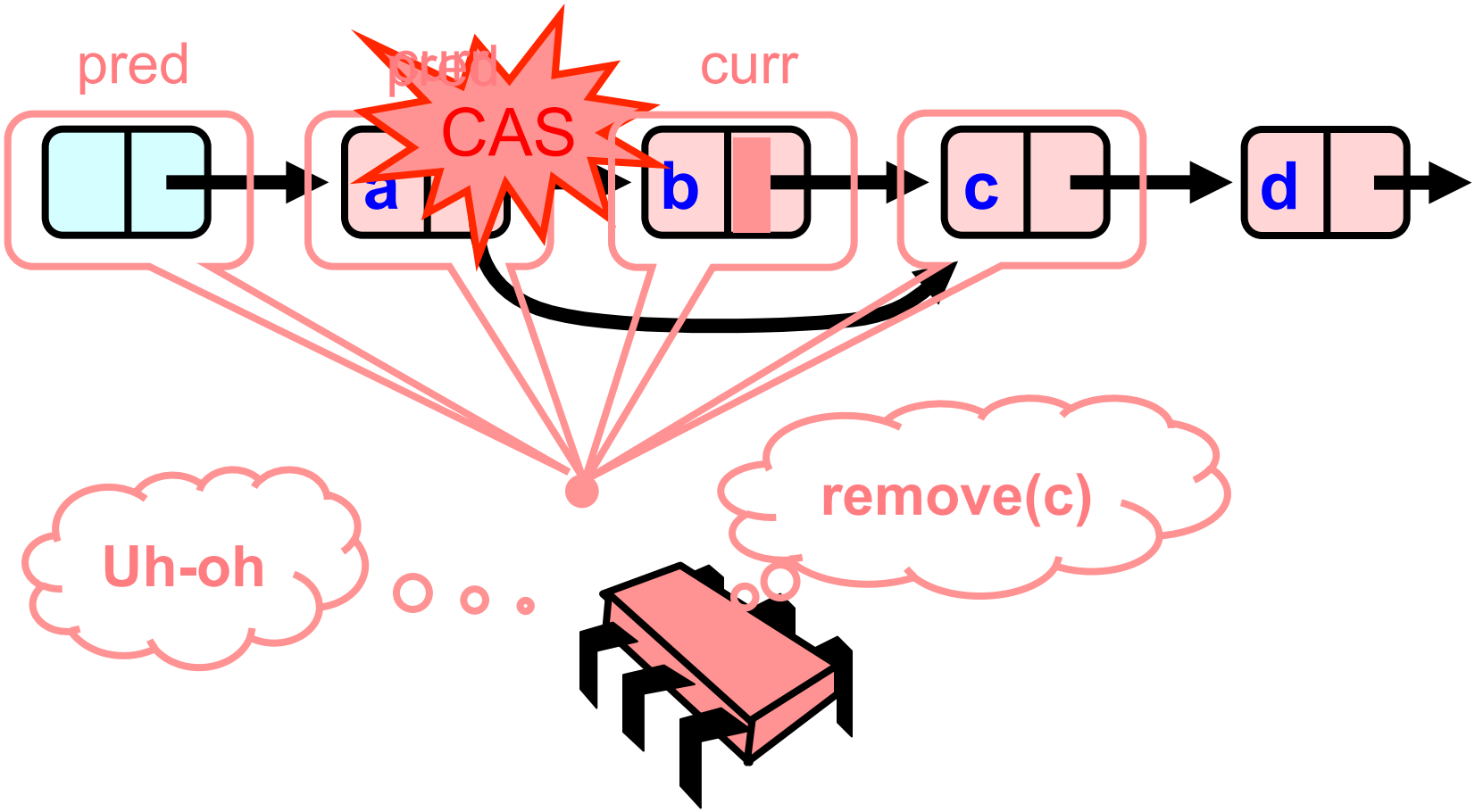
Removing a Node



Traversing the List

- Q: what do you do when you find a "logically" deleted node in your path?
- A: finish the job.
  - CAS the predecessor's next field
  - Proceed (repeat as needed)

Lock-Free Traversal
(only Add and Remove)



The Window Class

```
class Window {
 public Node pred;
 public Node curr;
 Window(Node pred, Node curr) {
   pred = pred; curr = curr;
 }
}
```

The Window Class

```
class Window {
 public Node pred;
 public Node curr;
 Window(Node pred, Node curr) {
   pred = pred; curr = curr;
 }
}
```

A container for pred and current values

Using the Find Method

```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

Using the Find Method

```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

Find returns window

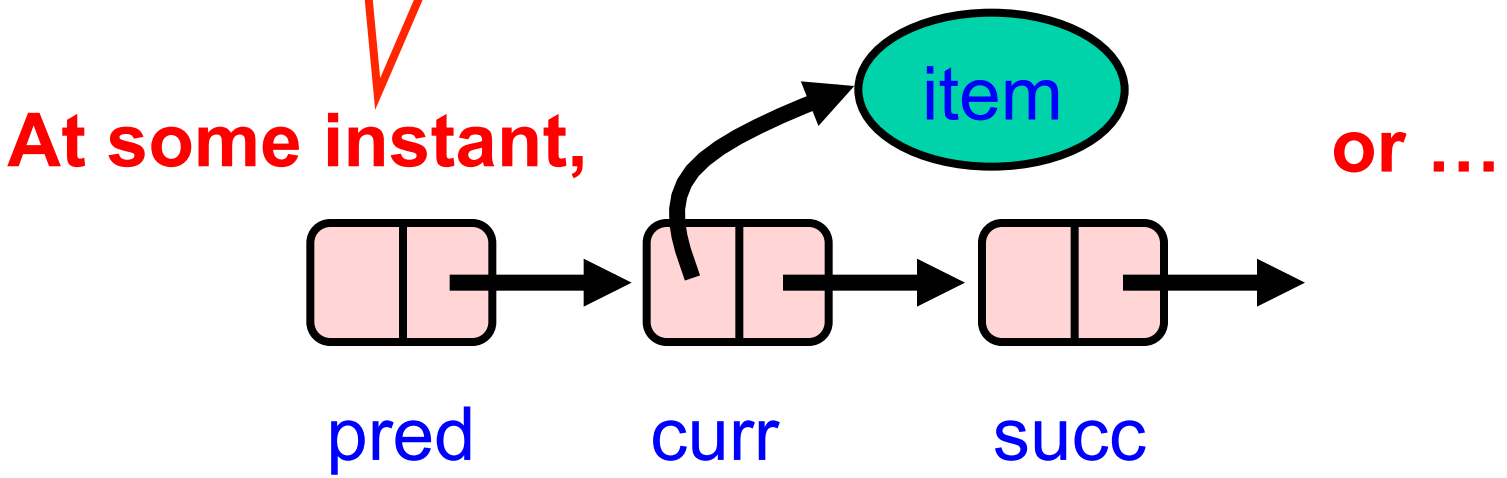
Using the Find Method

```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

Extract pred and curr

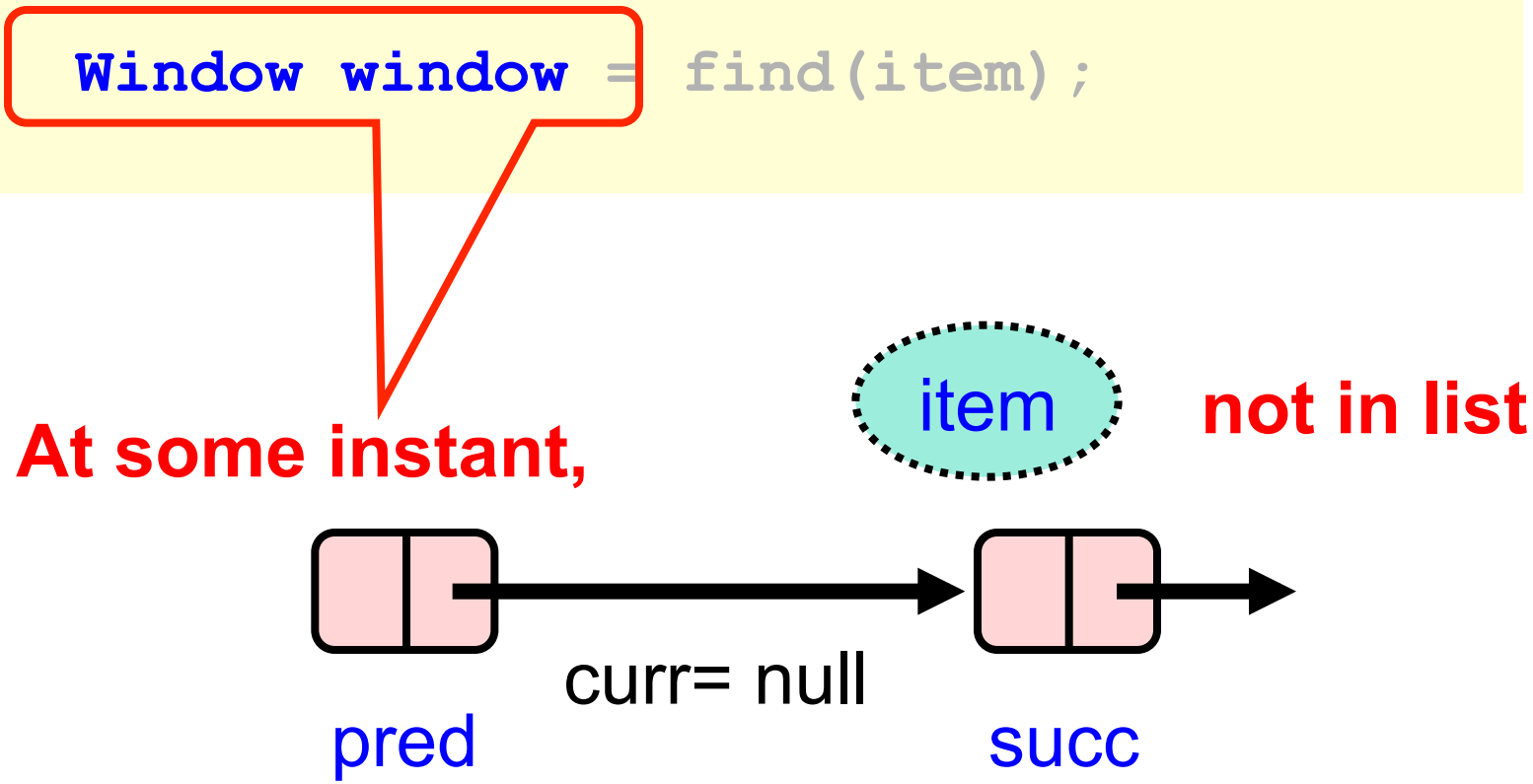
The Find Method

```
Window window = find(item);
```



The Find Method

```
Window window = find(item);
```



Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if (!snip) continue;
  pred.next.compareAndSet(curr, succ, false, false);
  return true;
}}}
```

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet (succ, succ, false,
true);
  if (!snip) continue;
  pred.next.compareAndSet(curr, succ, false, false);
  return true;
}}}
```

Keep trying

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
Window window = find(head, key);
Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet (succ, succ, false,
true);
  if (!snip) continue;
  pred.next.compareAndSet(curr, succ, false, false);
  return true;
}}}
```

Find neighbors

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
   Node succ = curr.next.getReference();
   snip = curr.next.compareAndSet(succ, succ, false,
true);
   if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false, false);
   return true;
}}}
```

Not there …

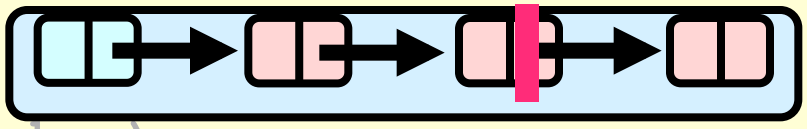
Remove

Try to mark node as deleted

```
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
  pred.next.compareAndSet(curr, succ, false, false);
  return true;
}}}
```

Remove

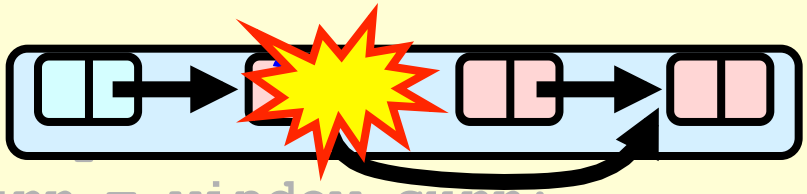
If it doesn't work, just retry, if it does, job essentially done



```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key == key) {
      return false;
    } else {
      Node succ = curr.next.getReference();
      snip = curr.next.compareAndSet(succ, succ, false,
true);
      if (!snip) continue;
      pred.next.compareAndSet(curr, succ, false, false);
      return true;
}}}
```

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head,
 Node pred = window.pred, curr = window.curr;
 if (curr.key != key) {
  return false;
  }
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false,
true);
  if (!snip) continue;
  pred.next.compareAndSet(curr, succ, false, false);
  return true;
}}}
```



Try to advance reference (if we don't succeed, someone else did or will).

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr =
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if (!snip) continue;
  pred.next.compareAndSet(curr, succ, false, false);
  return true;
}}}
```

Linearization point if removal is successful

Remove

```
public boolean remove(T item) {
Boolean snip;
while (true) {
 Window window = find(head, key);
 Node pred = window.pred, curr = window.curr;
  if (curr.key != key) {
     return false;
  } else {
  Node succ = curr.next.getReference();
  snip = curr.next.compareAndSet(succ, succ, false
true);
  if (!snip) continue;
  pred.next.compareAndSet(curr, succ, false, false);
  return true;
}}}
```

Linearization point is when we found this node (in Find()) if removal returns false.

Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

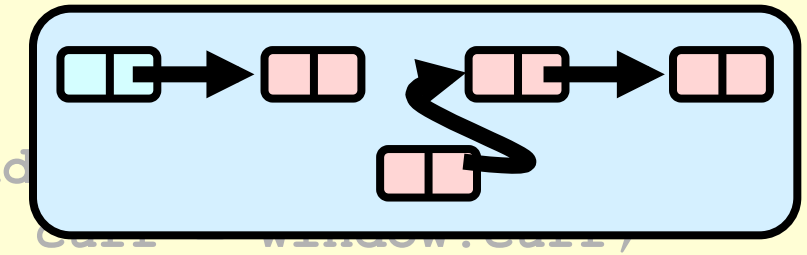
Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

Item already there

Add

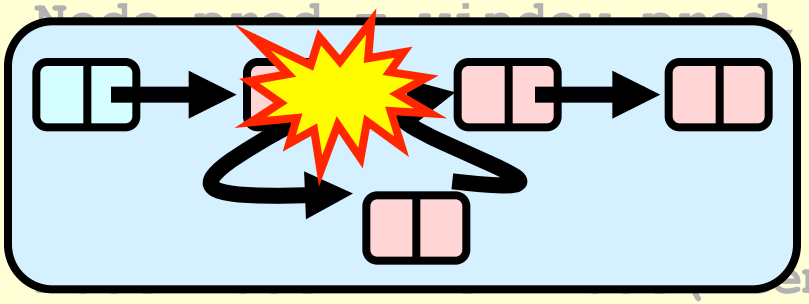
```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head
   Node pred = window.pred, curr
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```



create new node

Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
                                    em);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```



Install new node, else retry loop

Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

Linearization point if add is successful

Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

Linearization point is when we found this node (in Find()) if removal returns false.

Wait-free Contains

```
public boolean contains(T item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = head;
    while (curr.key < key)
      curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
  }
```

Wait-free Contains

```
public boolean contains(T item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = head;
    while (curr.key < key)
      curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
  }
```

Only difference is that we get and check marked

Wait-free Contains

```
public boolean contains(T item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = head;
    while (curr.key < key)
      curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
  }
```

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
       return new Window(pred, curr);
    pred = curr;
    curr = succ;
  }
}}
```

Lock-free Find

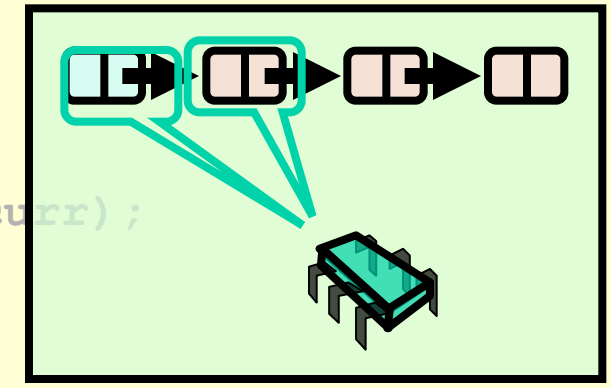
```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
    pred = head;
    curr = pred.next.getReference();
    while (true) {
     succ = curr.next.get(marked);
     while (marked[0]) {
     …
     }
     if (curr.key >= key)
       return new Window(pred, curr);
     pred = curr;
     curr = succ;
   }
}}
```

If list changes while traversed, start over

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr, succ;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
    while (true) {
     succ = curr.next.get(marked);
     while (marked[0]) {
     …
     }
    if (curr.key >= key)
      return new Window(pred, curr);
    pred = curr;
    curr = succ;
  }
}}
```

Start looking from head



Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
      return new Window(pred, curr);
    pred = curr;
    curr = succ;
  }
}}
```

Move down the list

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
      return new Window(pred, curr);
    pred = curr;
    curr = succ;
  }
}}
```

Get ref to successor and current deleted bit

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if (curr.key >= key)
      return new Window(pred, curr);
    pred = curr;
```

Try to remove deleted nodes in path…code details soon

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (marked[0]) {
    …
    }
   if (curr.key >= key)
     return new Window(pred, curr);
   pred = curr;
   curr = succ;
  }
}}
```

If curr key that is greater or equal, return pred and curr

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
     ...
    }
    if (curr.key >= key)
          return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
}}
```

Otherwise advance window and loop again

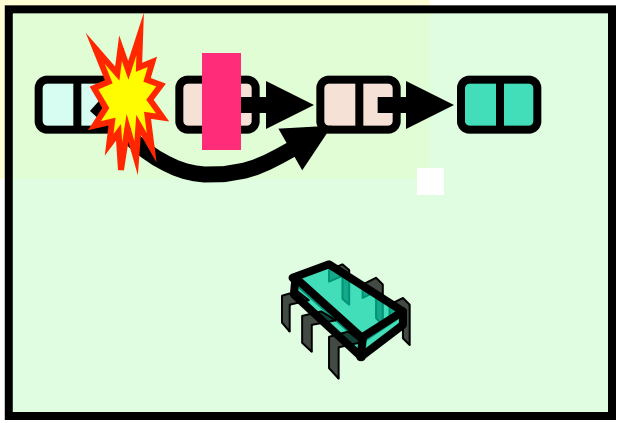
Lock-free Find

```
retry: while (true) {
    …
    while (marked[0]) {
      snip = pred.next.compareAndSet(curr,
                                succ, false, false);
      if (!snip) continue retry;
      curr = succ;
      succ = curr.next.get(marked);
    }
…
```

Lock-free Find

Try to snip out node

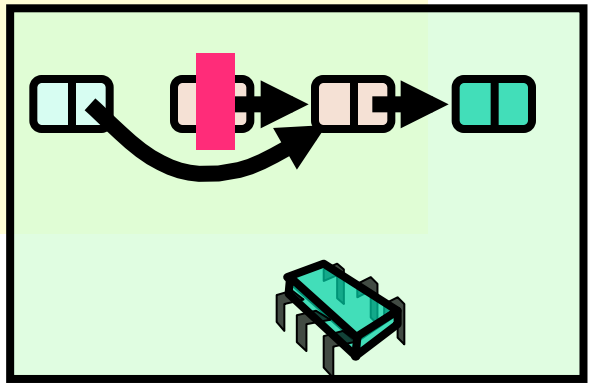
```
retry: while (true) {
    …
    while (marked[0]) {
      snip = pred.next.compareAndSet(curr,
                                     succ, false, false);
      if (!snip) continue retry;
      curr = succ;
      succ = curr.next.get(marked);
    }
…
```



Lock-free Find

if predecessor's next field changed, retry whole traversal

```
retry: while (true) {
    …
    while (marked[0]) {
      snip = pred.next.compareAndSet(curr,
                                     succ, false, false);
      if (!snip) continue retry;
      curr = succ;
      succ = curr.next.get(marked);
    }
…
```



Lock-free Find

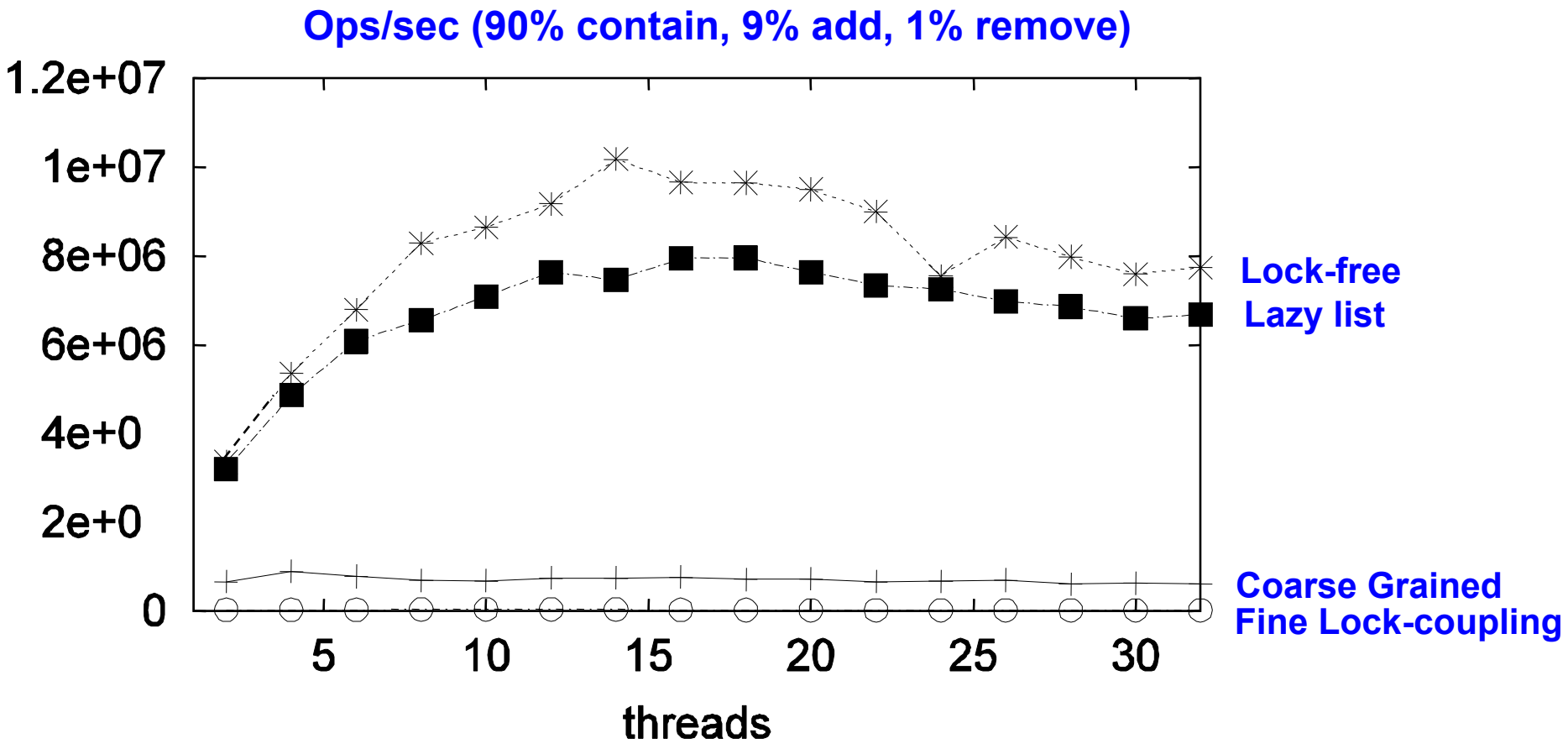
Otherwise move on to check if next node deleted

```
retry: while (true) {
    …
    while (marked[0]) {
      snip = pred.next.compareAndSet(curr,
                                     succ, false, false);
      if (!snip) continue retry;
      curr = succ;
      succ = curr.next.get(marked);
    }
…
```

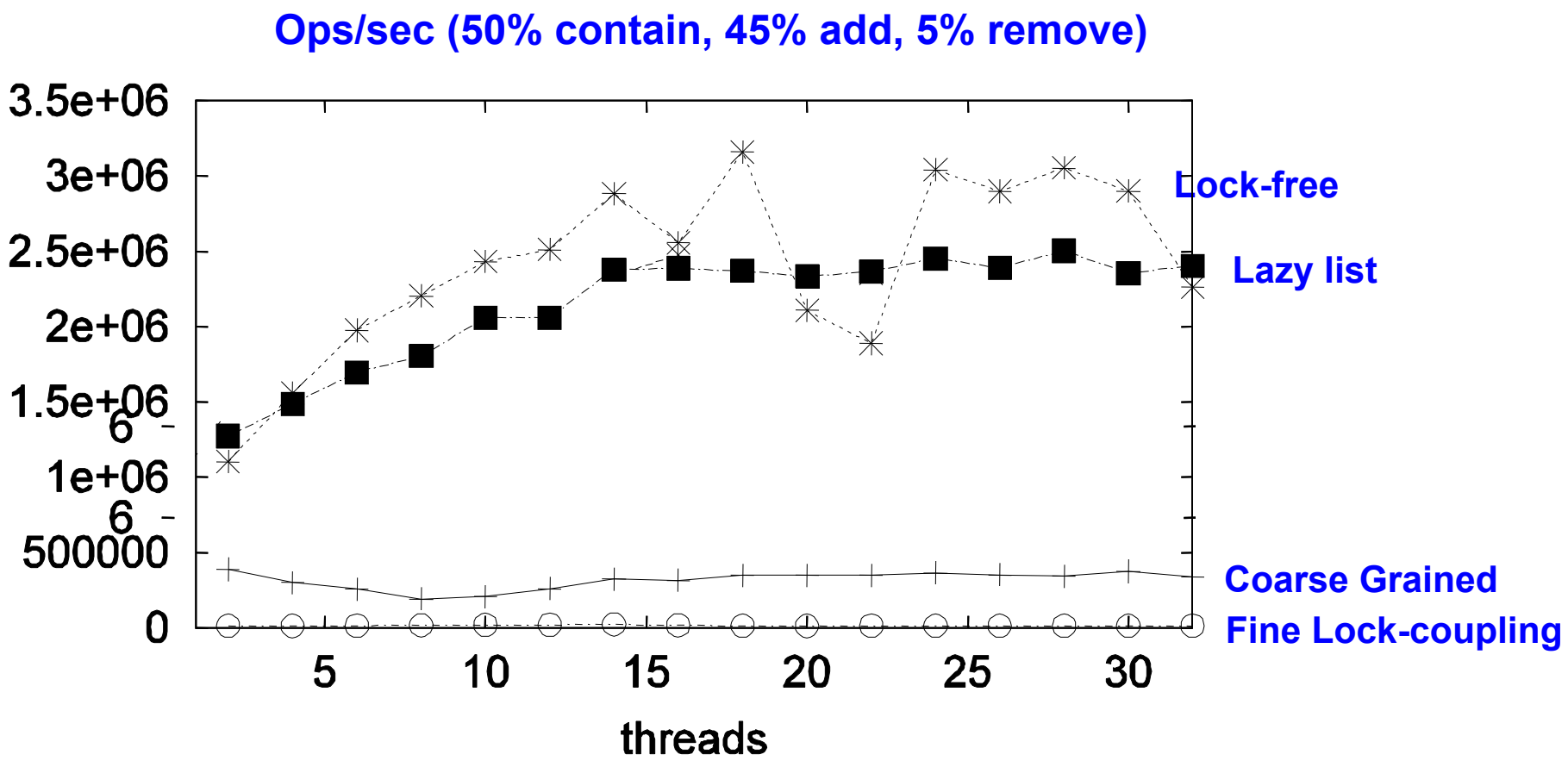
Performance

- Different list-based set implementaions
- SunFire 6800 (bus based cache coherence)
- 16-node machine, each 1.2 GHz
- Vary percentage of **`contains()`** calls

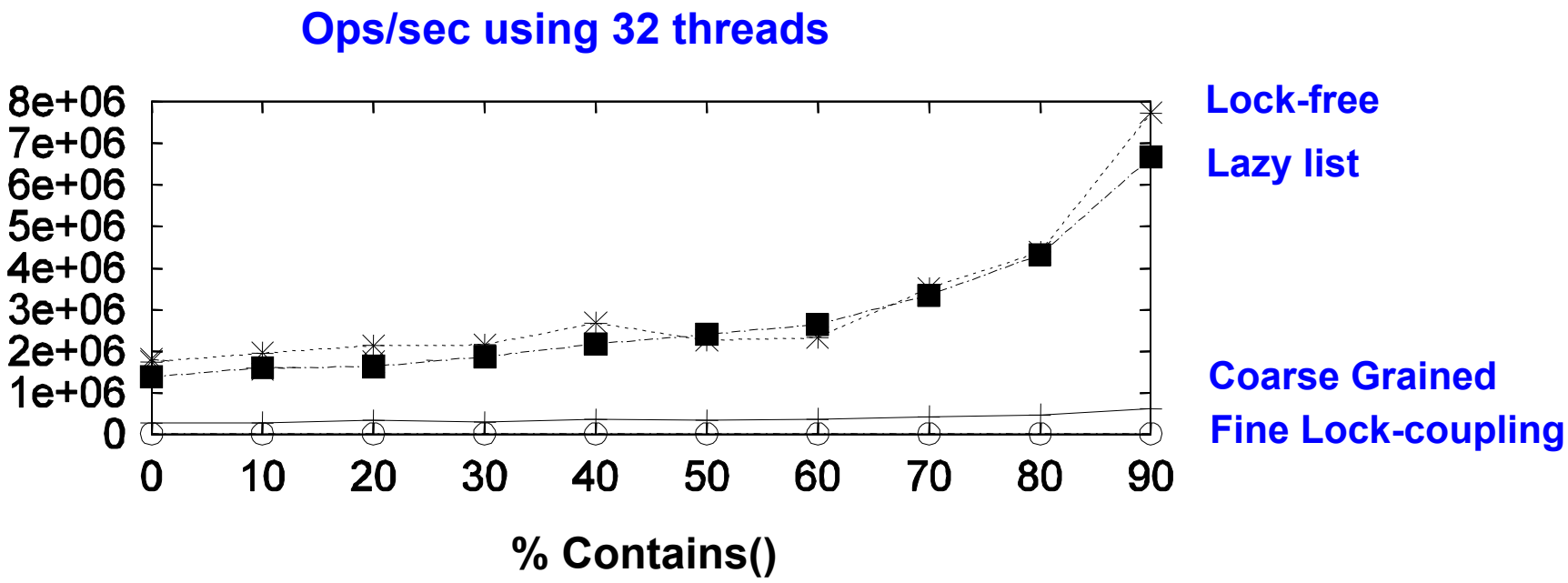
High Contains Ratio



Low Contains Ratio



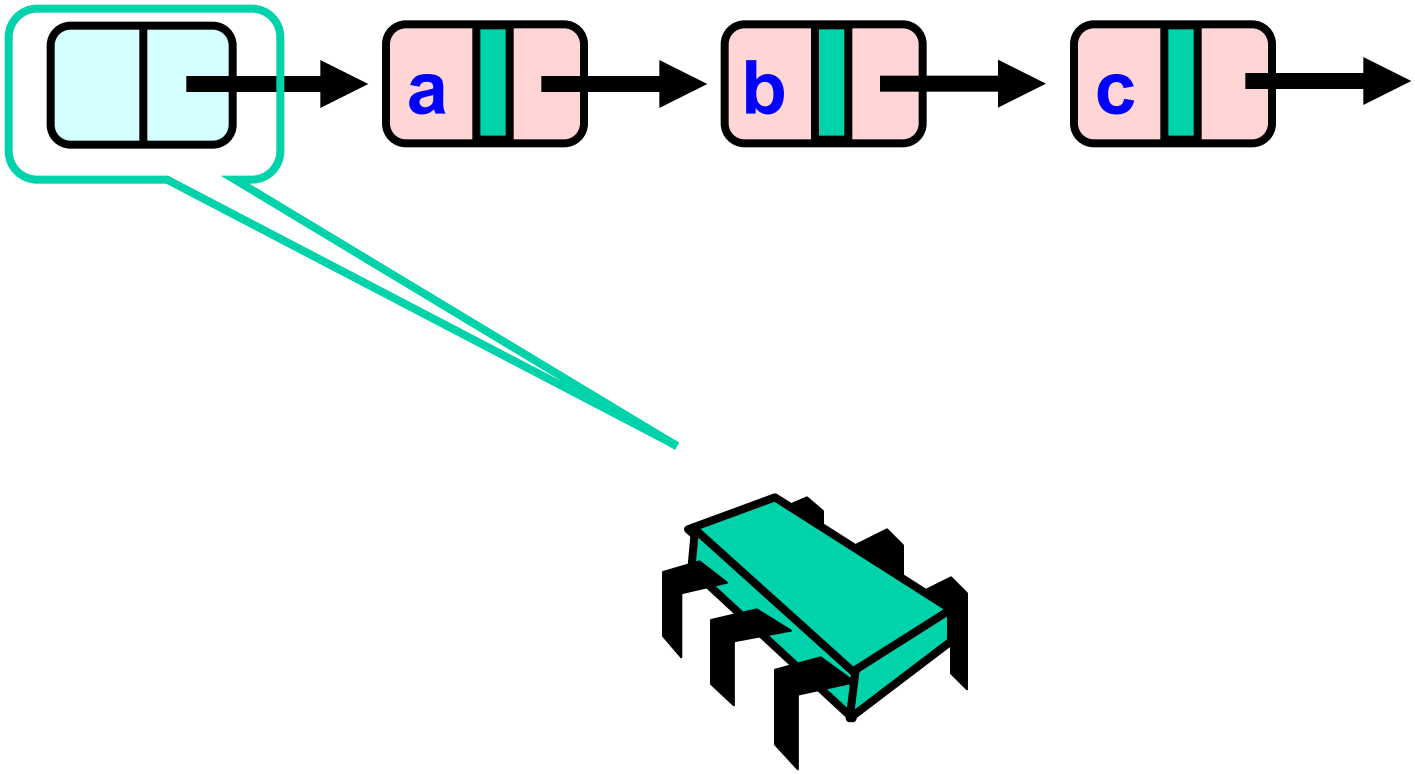
As Contains Ratio Increases



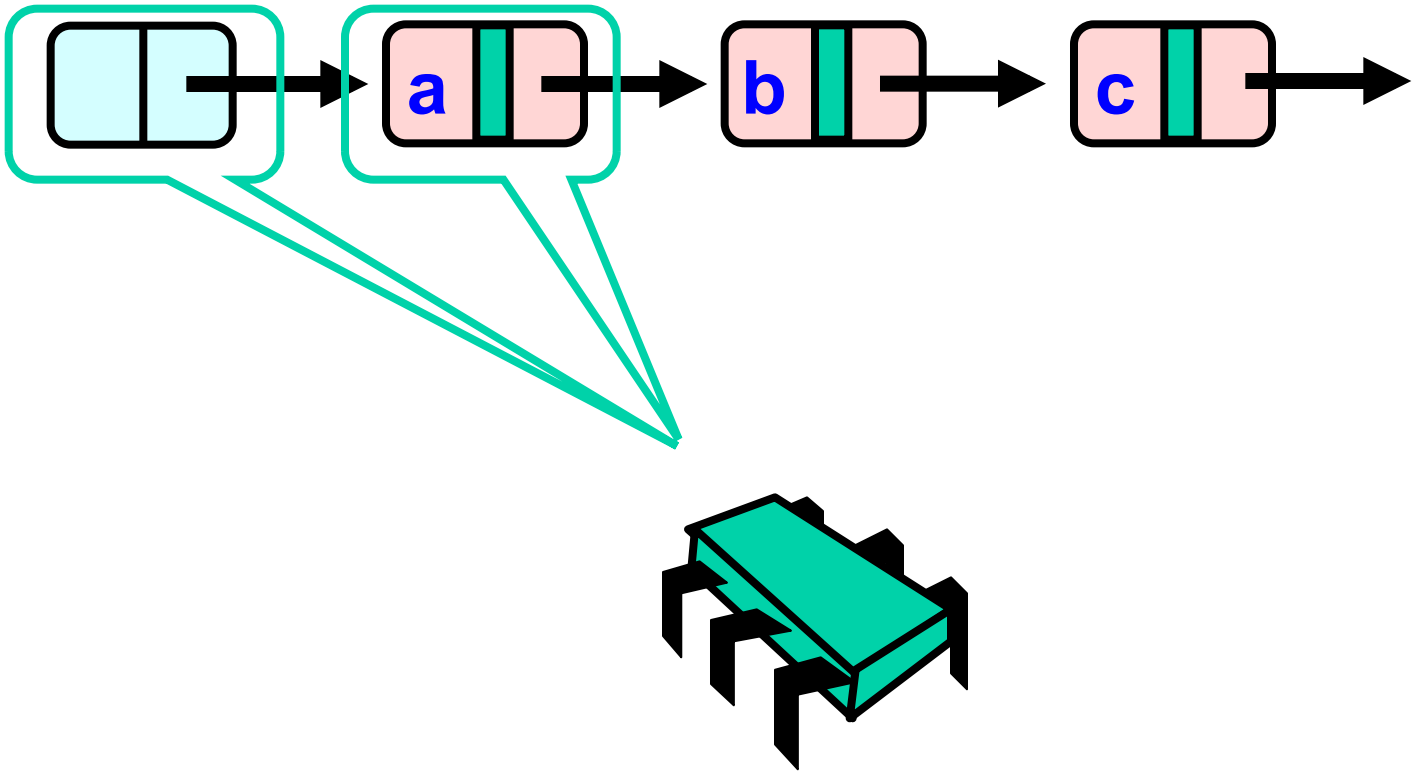
“To Lock or Not to Lock”

- Locking vs. Non-blocking:
  - Depending on the application usage
- The answer: nobler to compromise
  - Example: Lazy list combines blocking **`add()`** and **`remove()`** and a wait-free **`contains()`**
  - Remember: Blocking/non-blocking is a property of a method

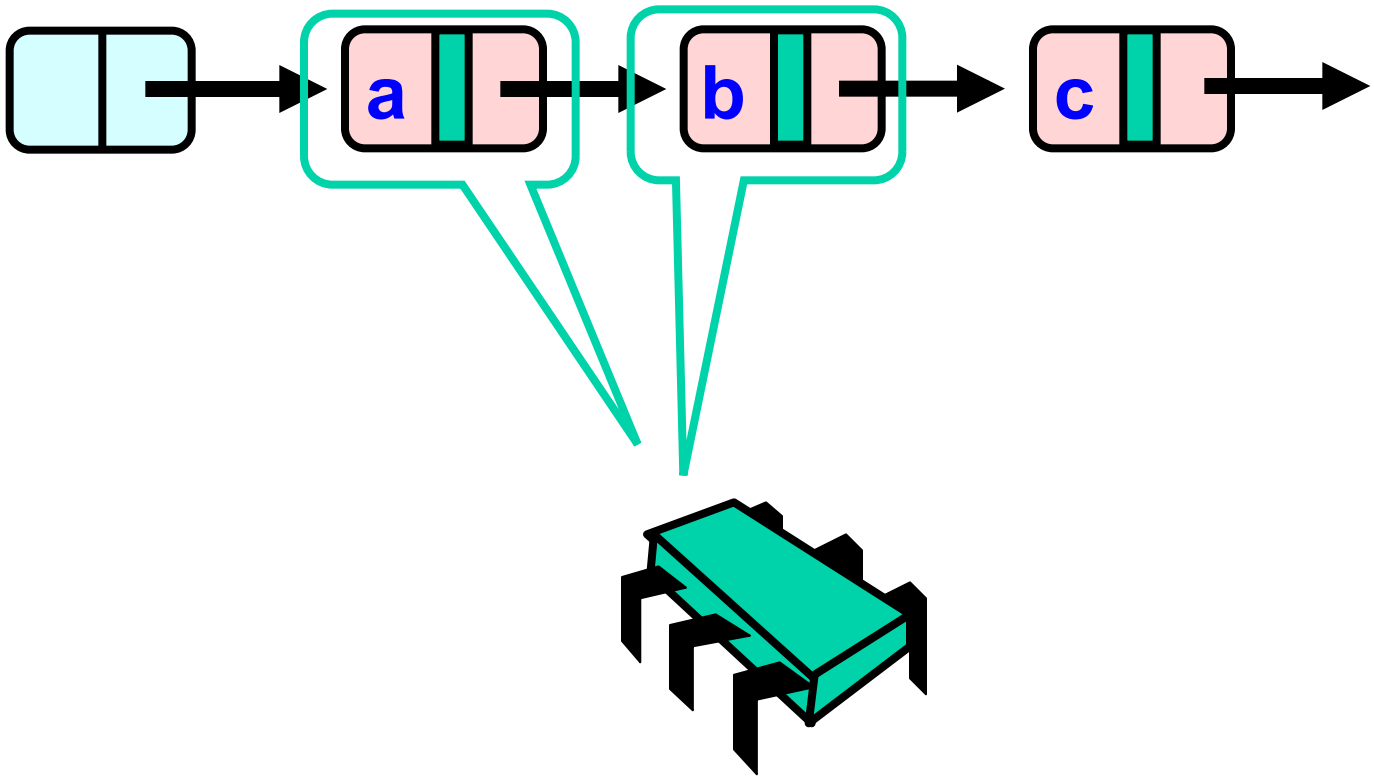
Business as Usual



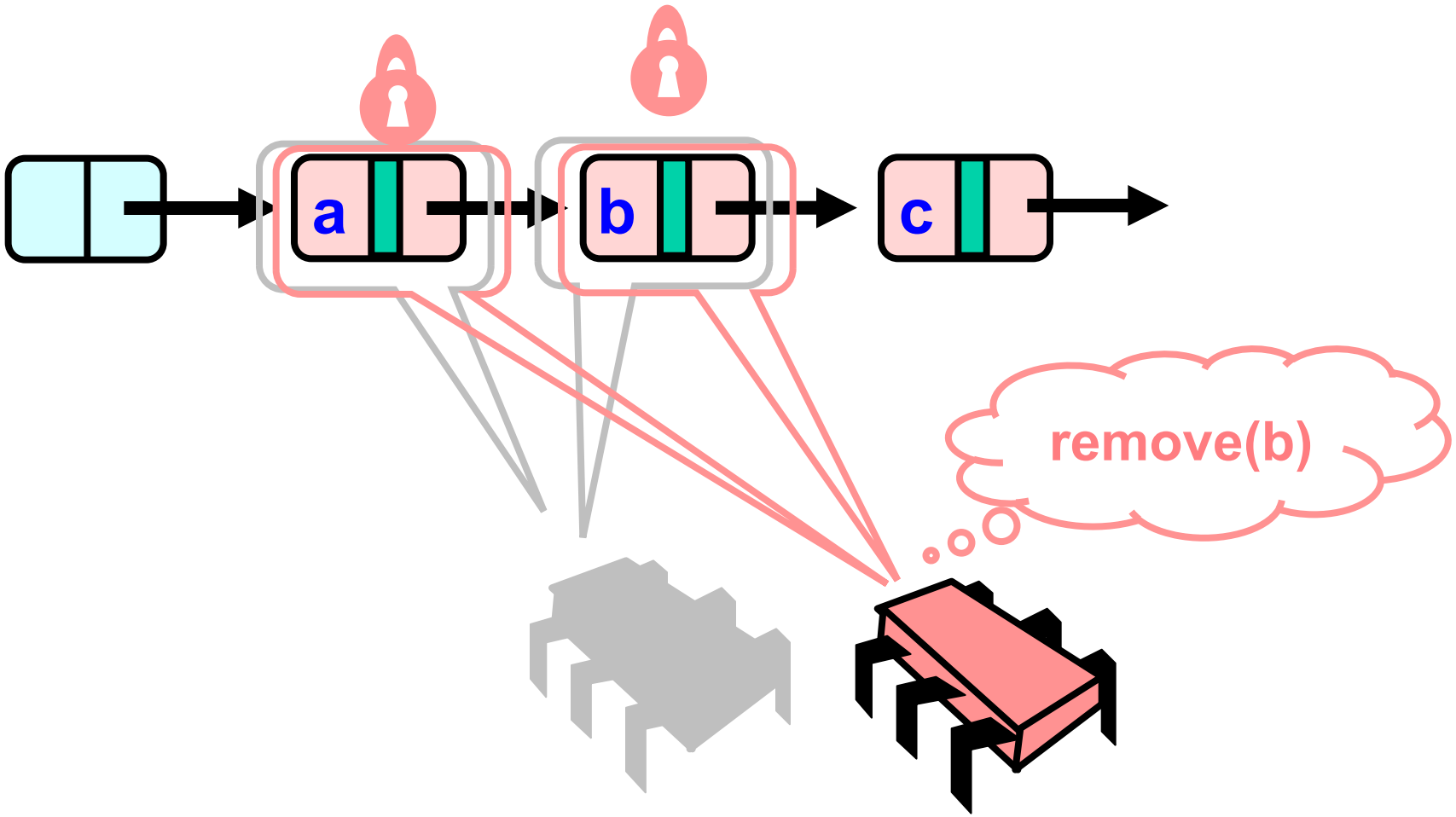
Business as Usual



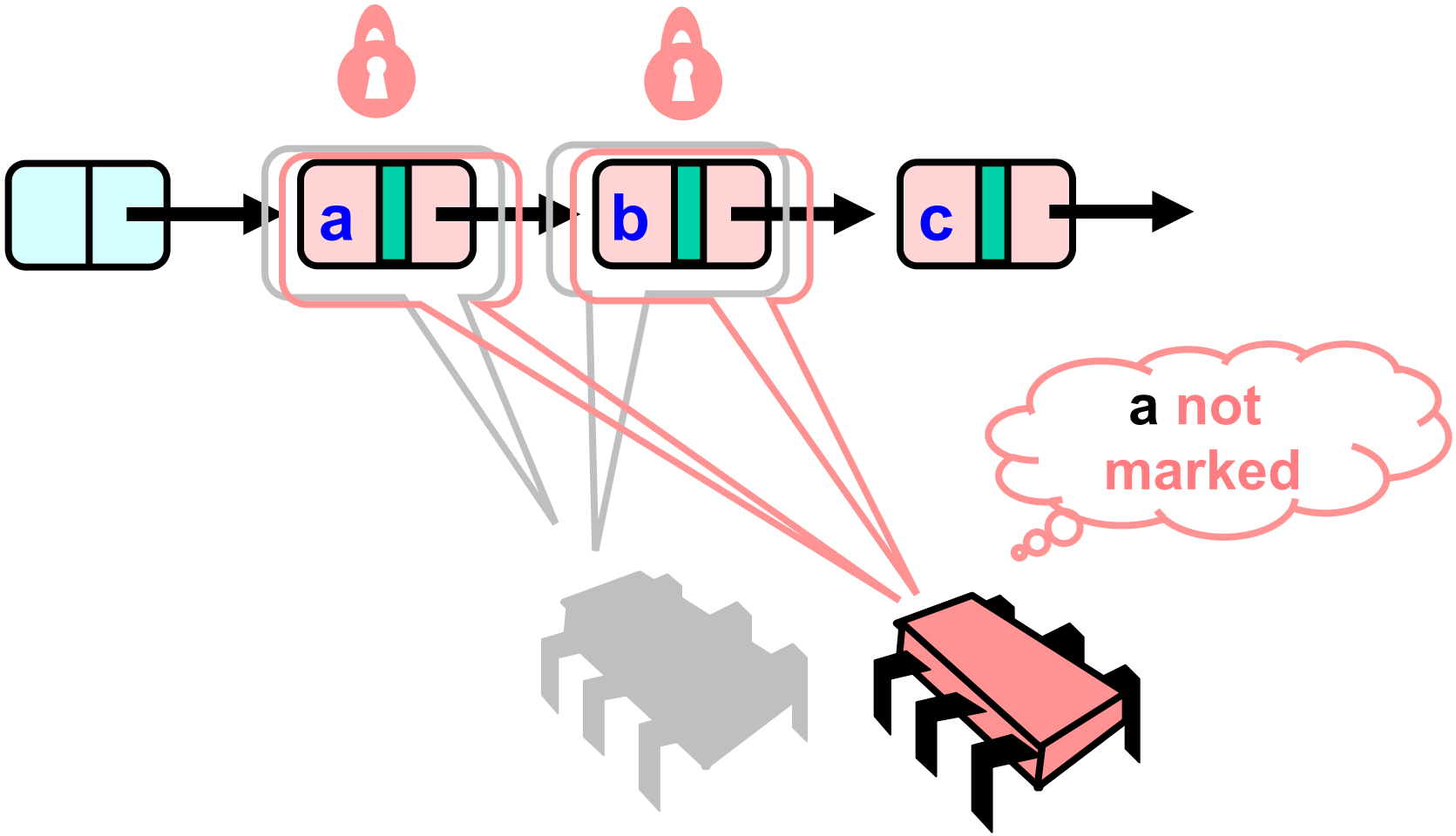
Business as Usual



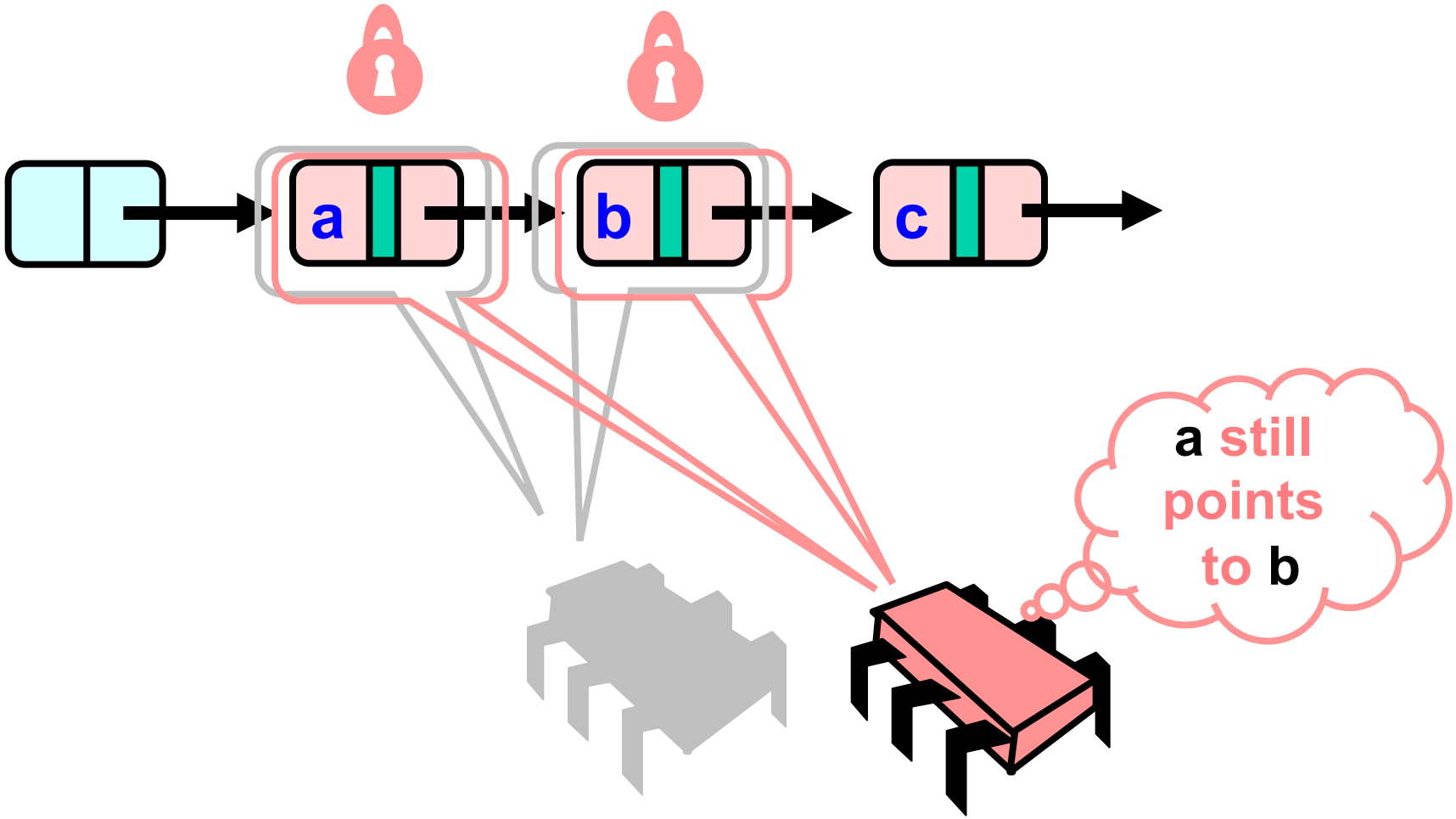
Business as Usual



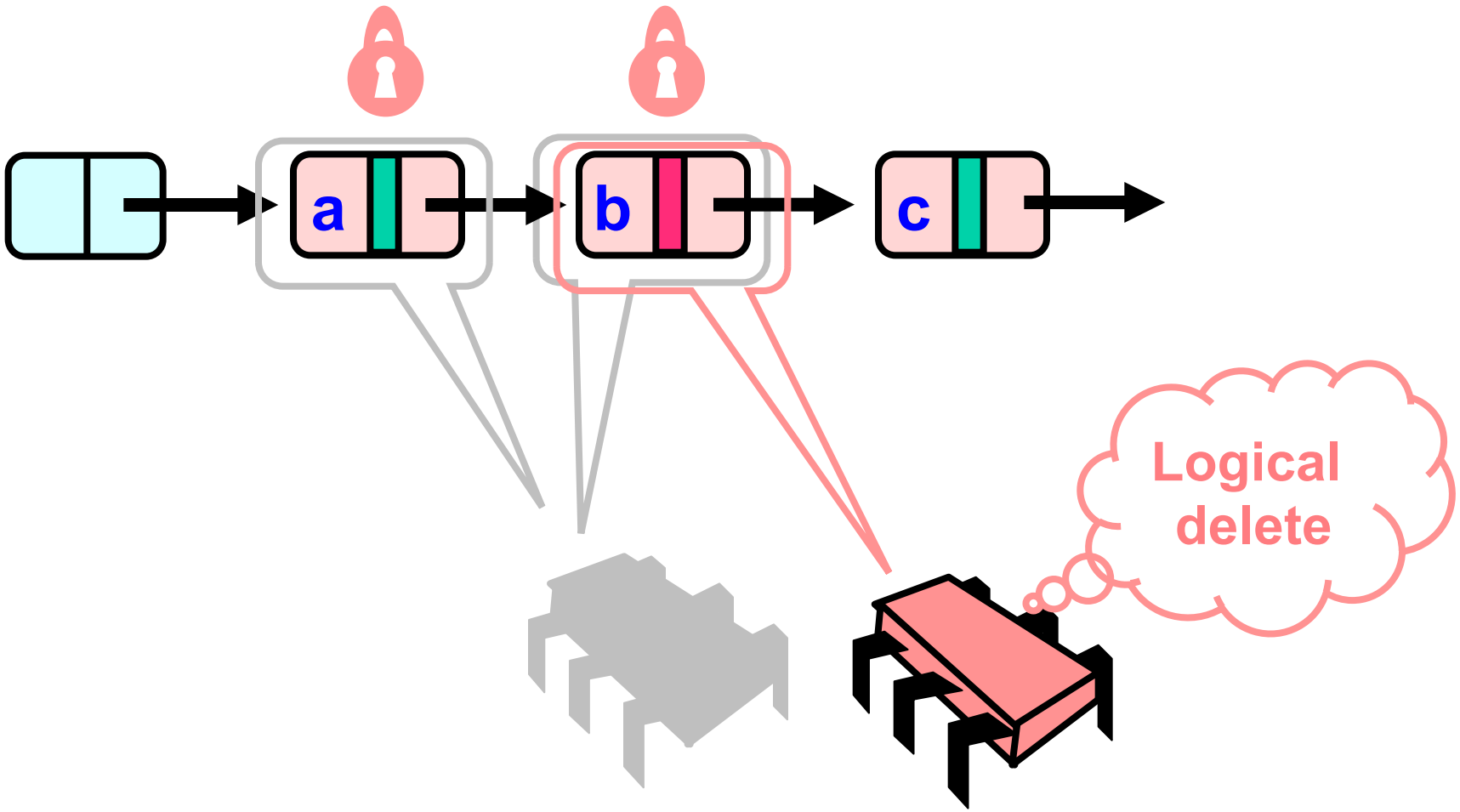
Business as Usual



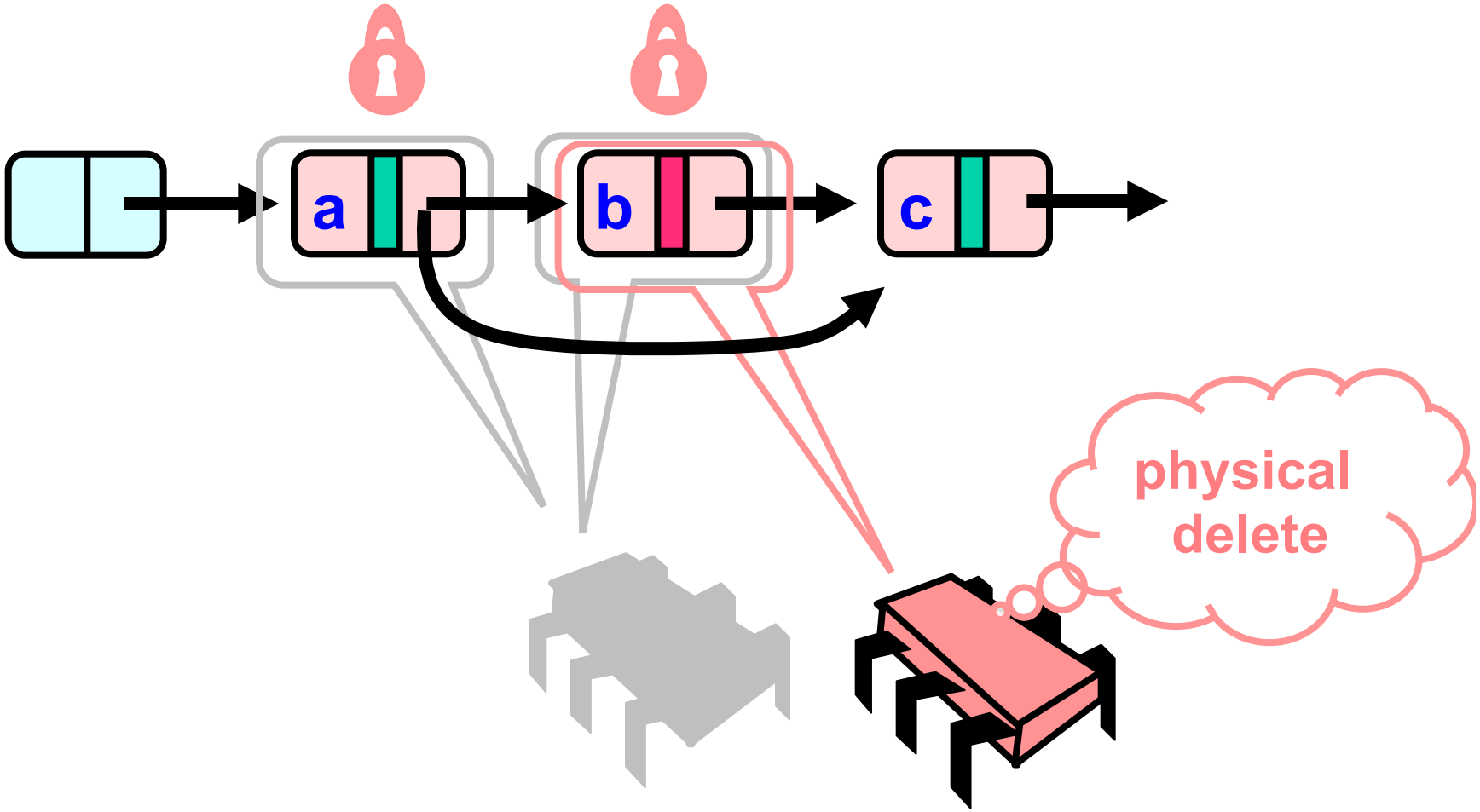
Business as Usual



Business as Usual



Business as Usual



Business as Usual