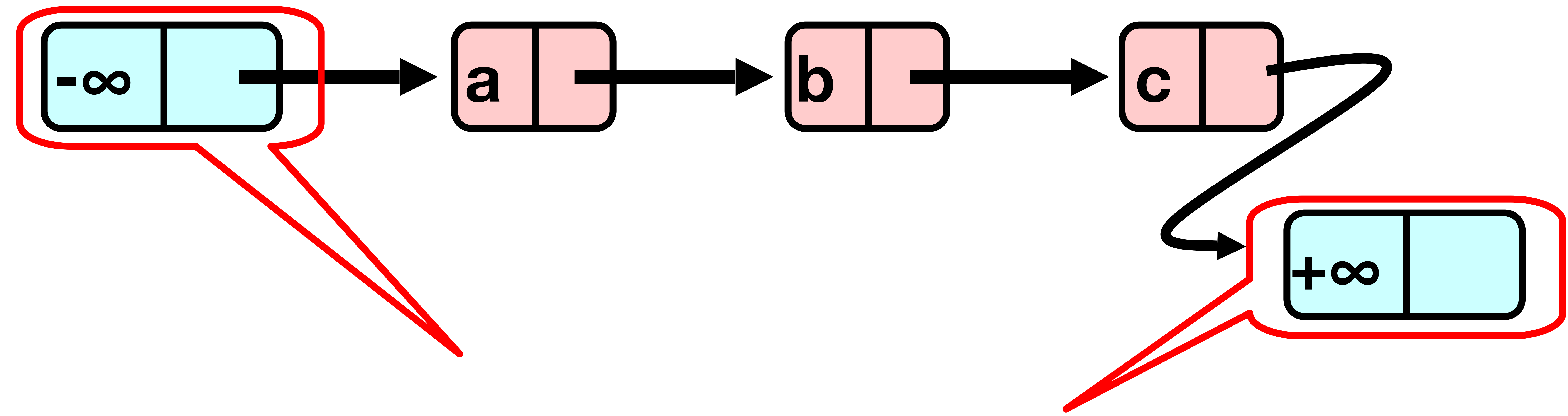
Locking Strategies: Lock Free

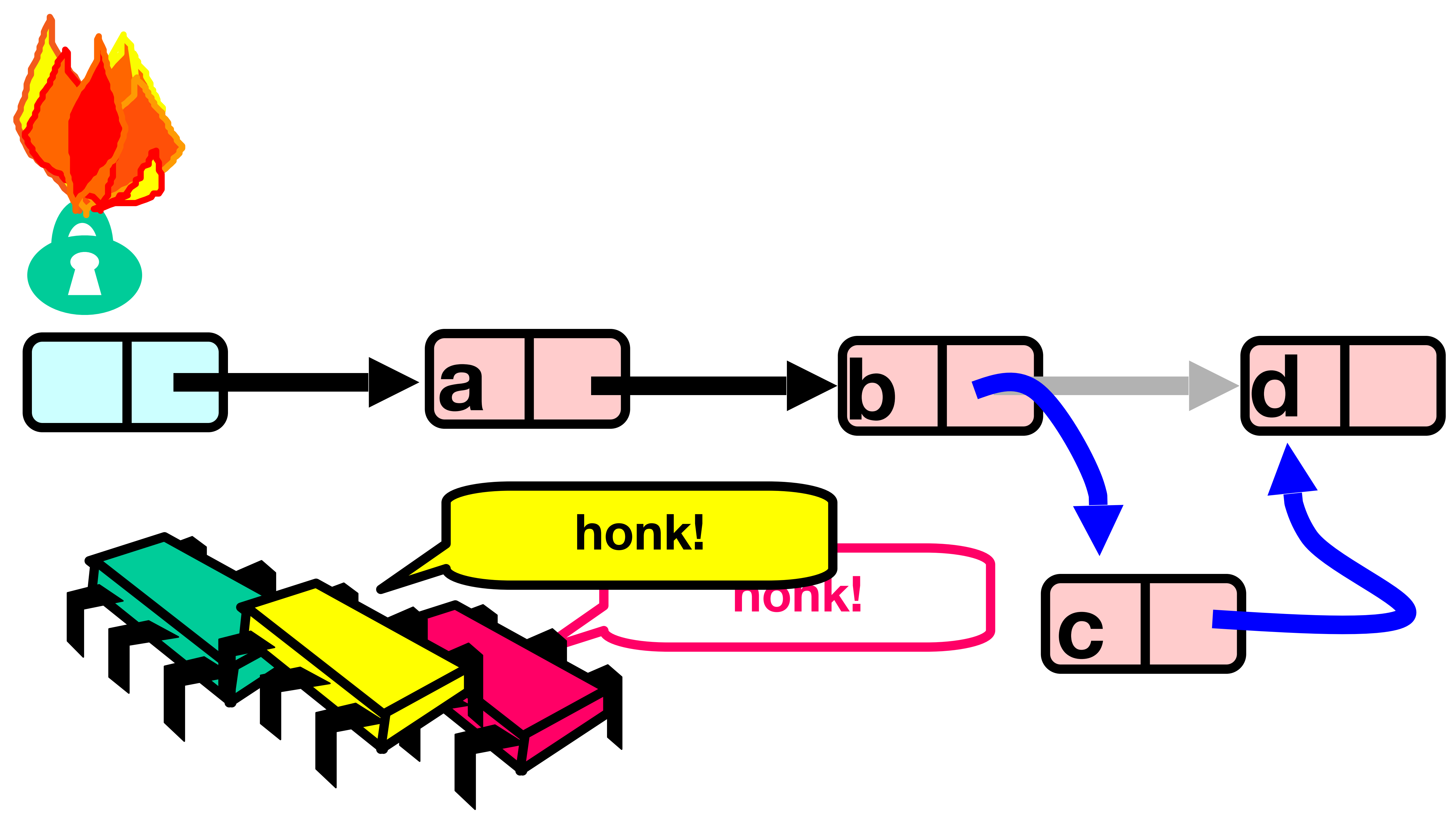
CS 475, Fall 2019
Concurrent & Distributed Systems

With material from Herlihy & Shavit, Art of Multiprocessor Programming

The List-Based Set

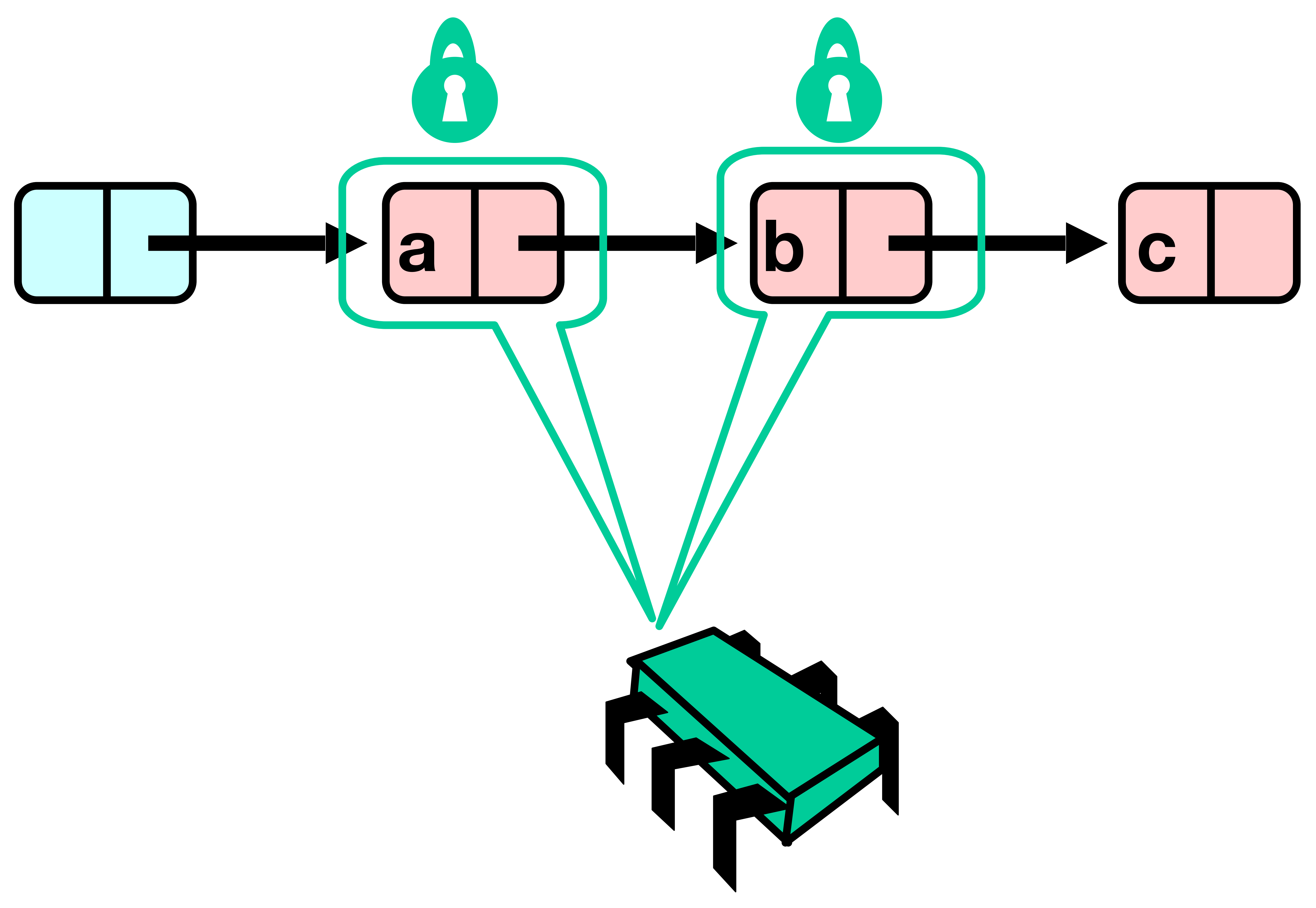


Course Grained Locking

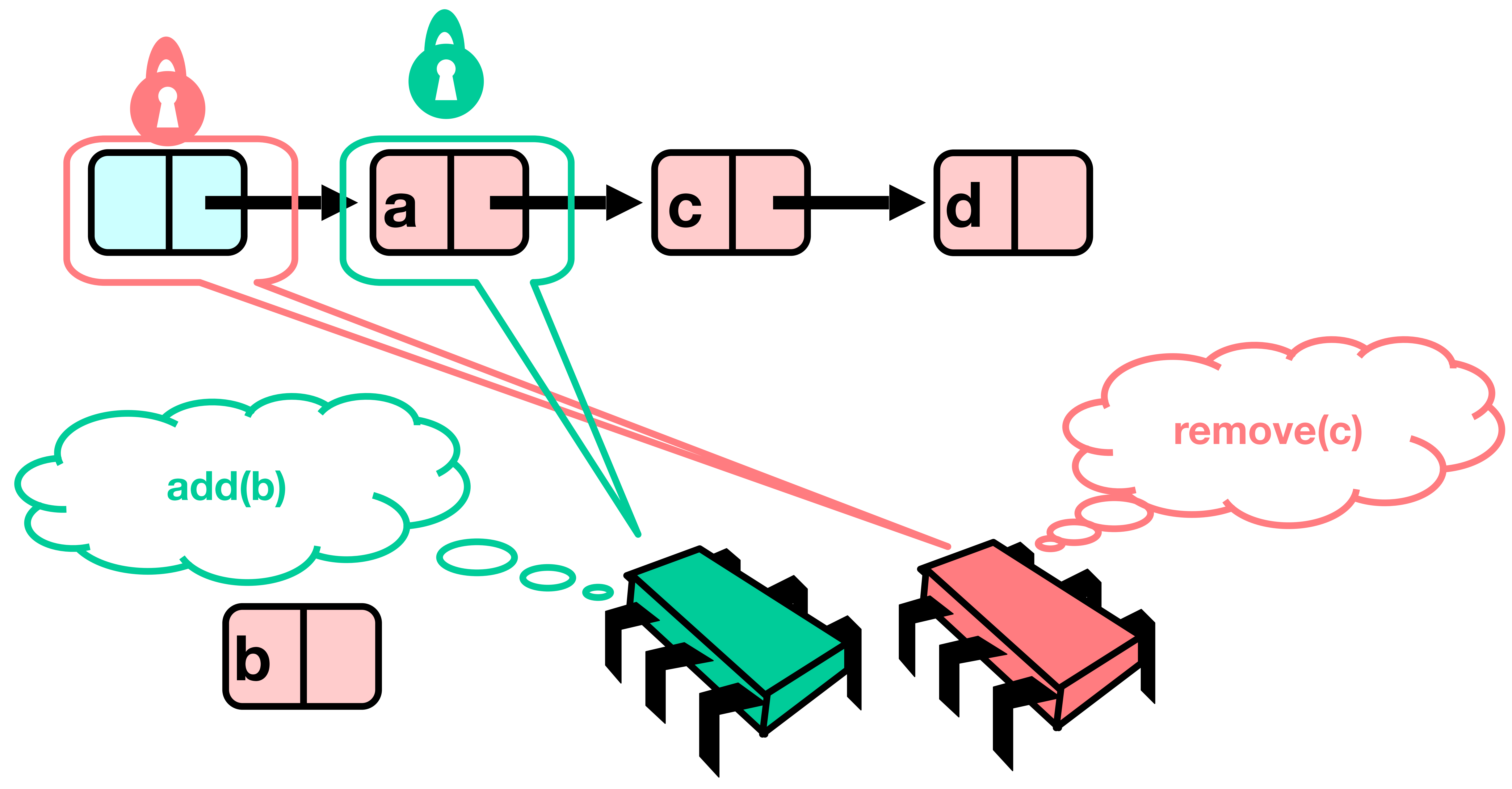


Simple but hotspot + bottleneck

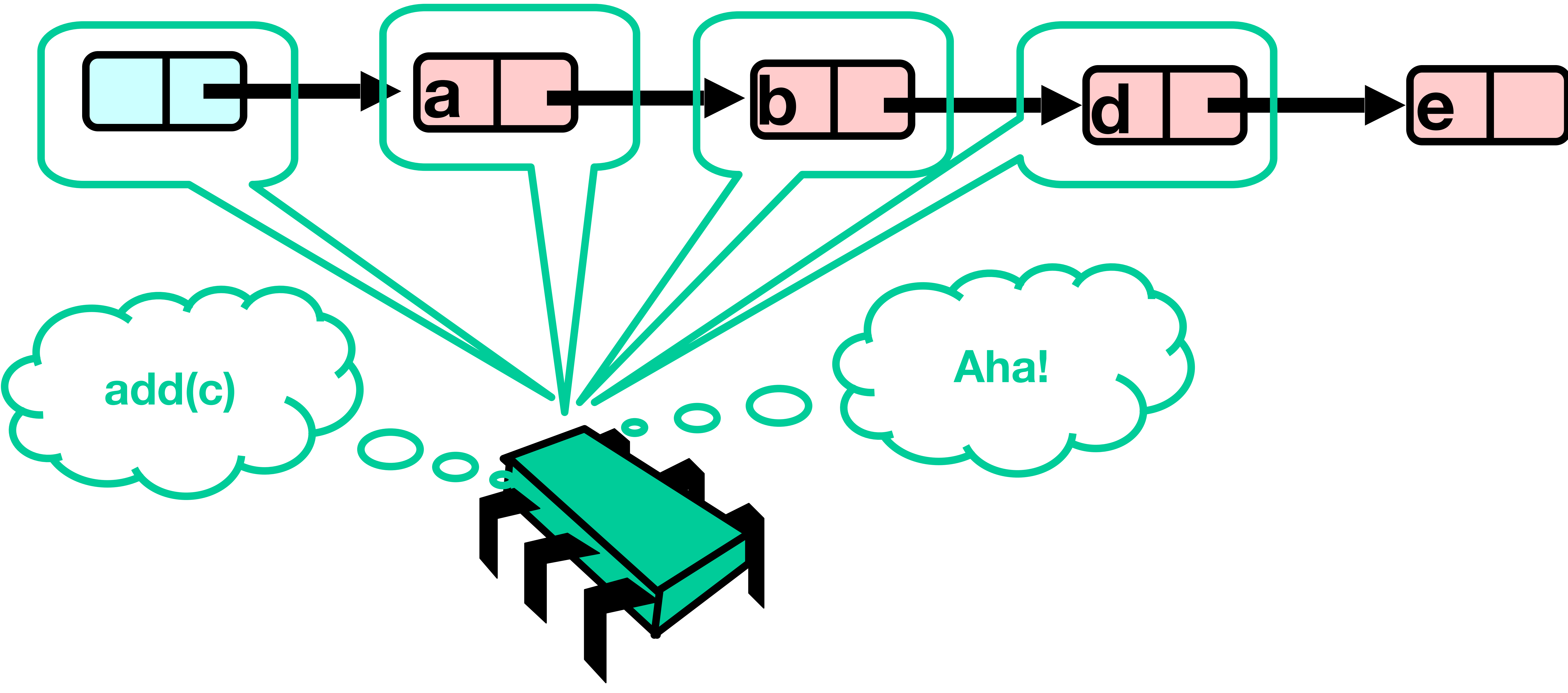
Hand-over-Hand locking



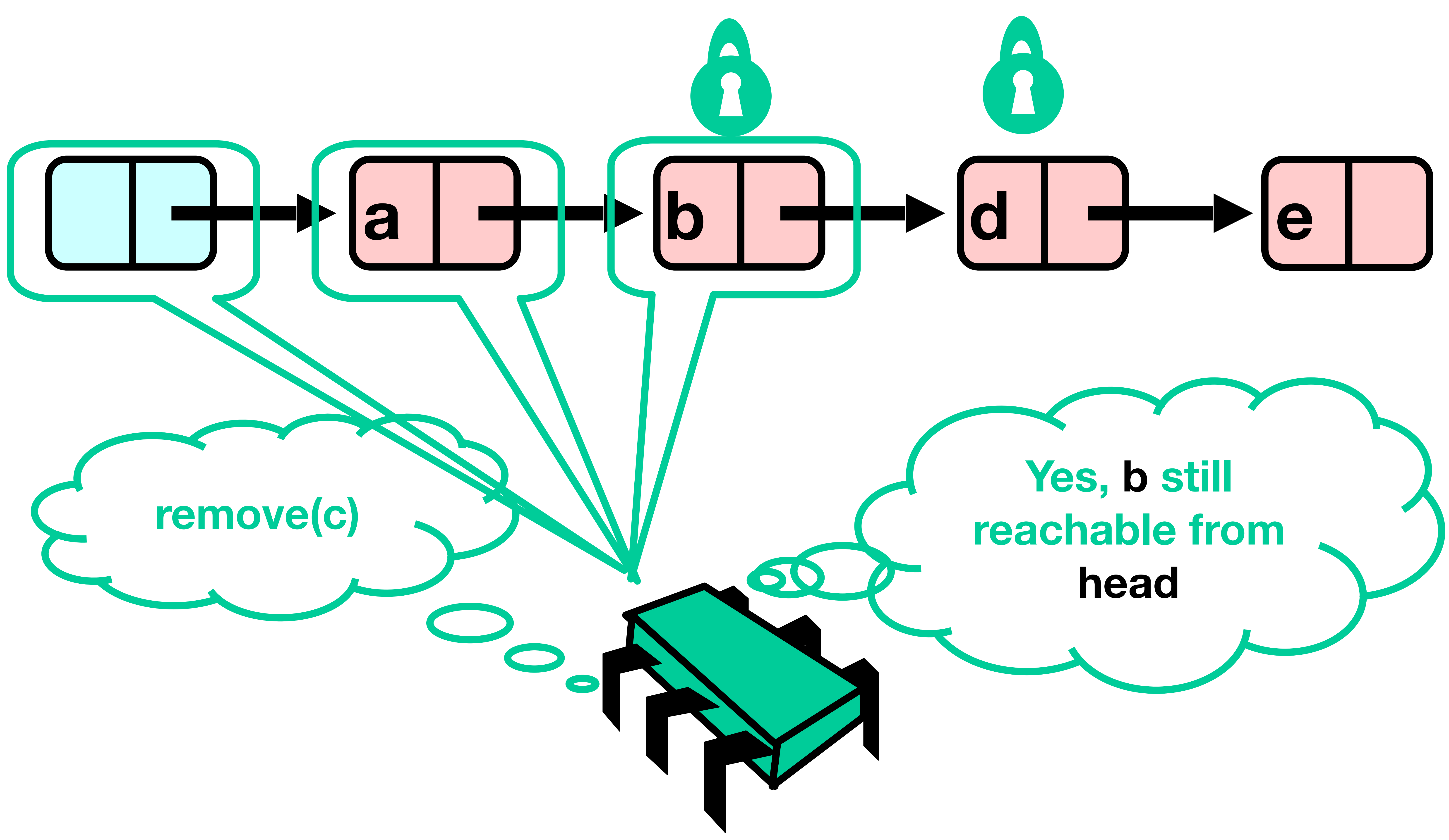
Simple Fine-Grained Locking: Add (Execution Order 1)



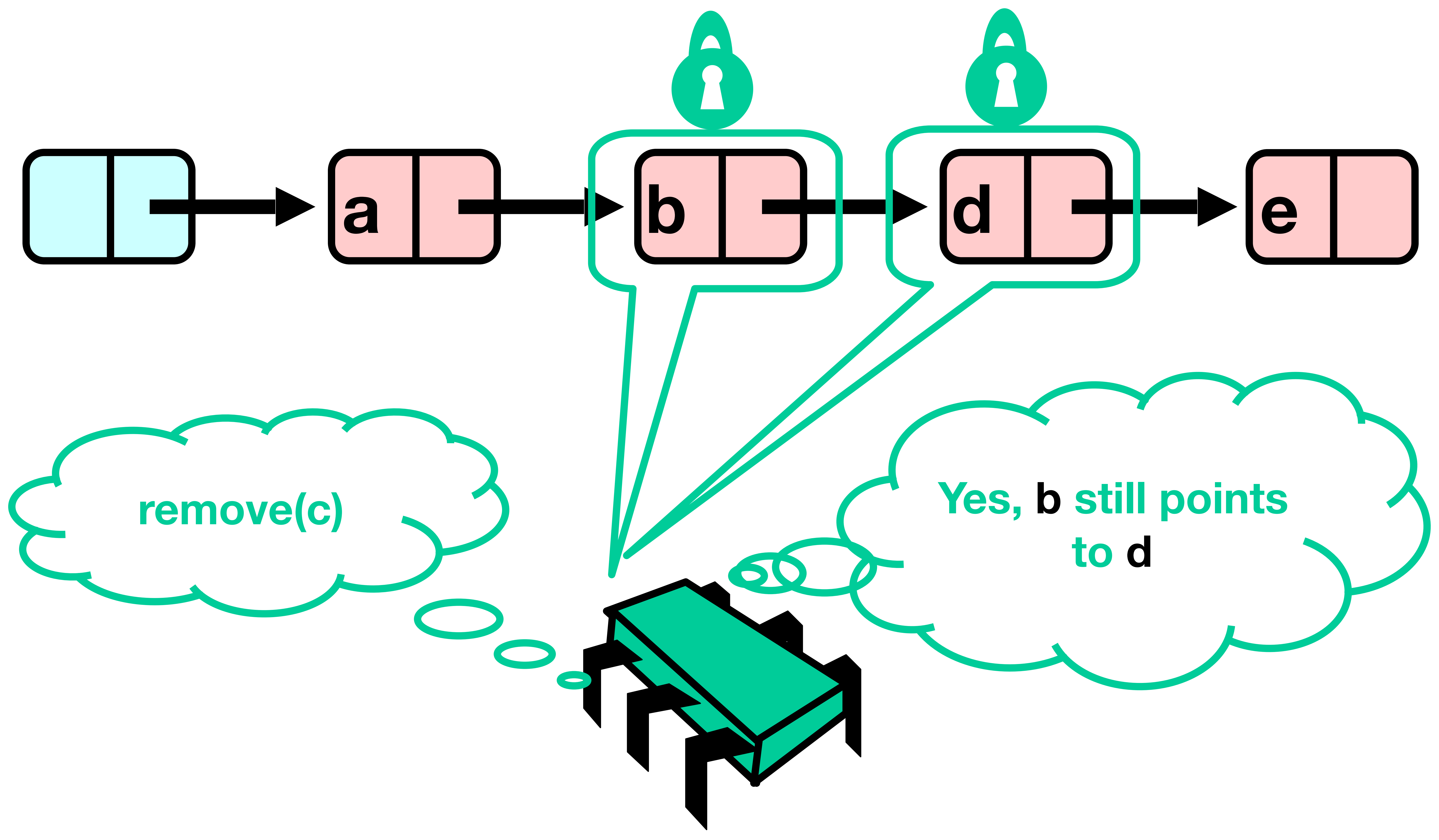
Optimistic: Traverse without Locking



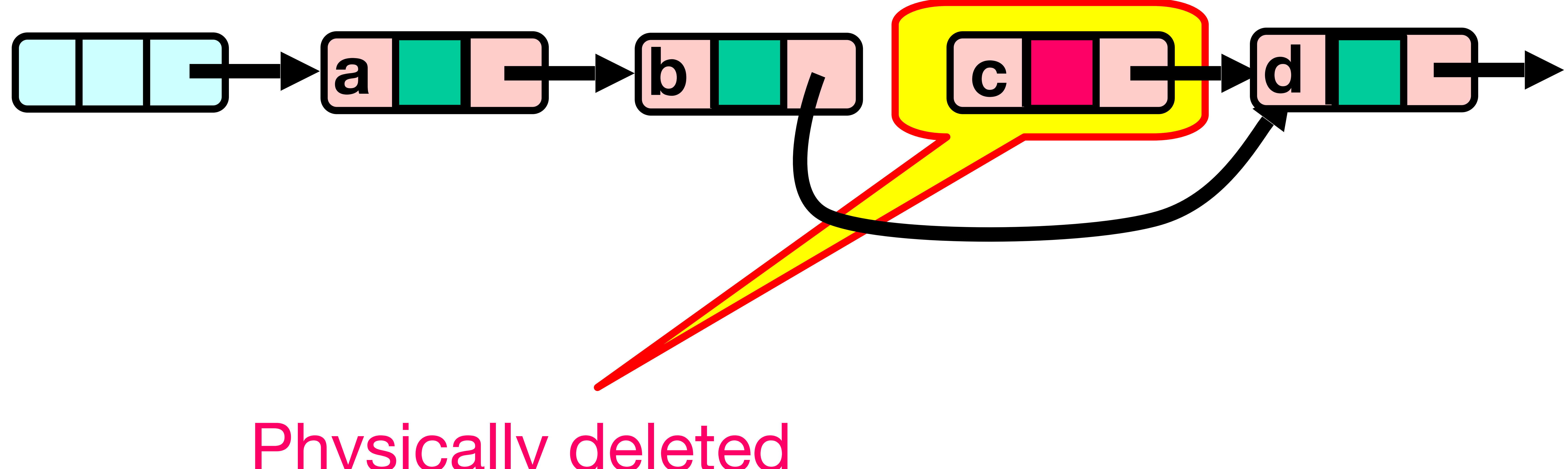
Validate (1)



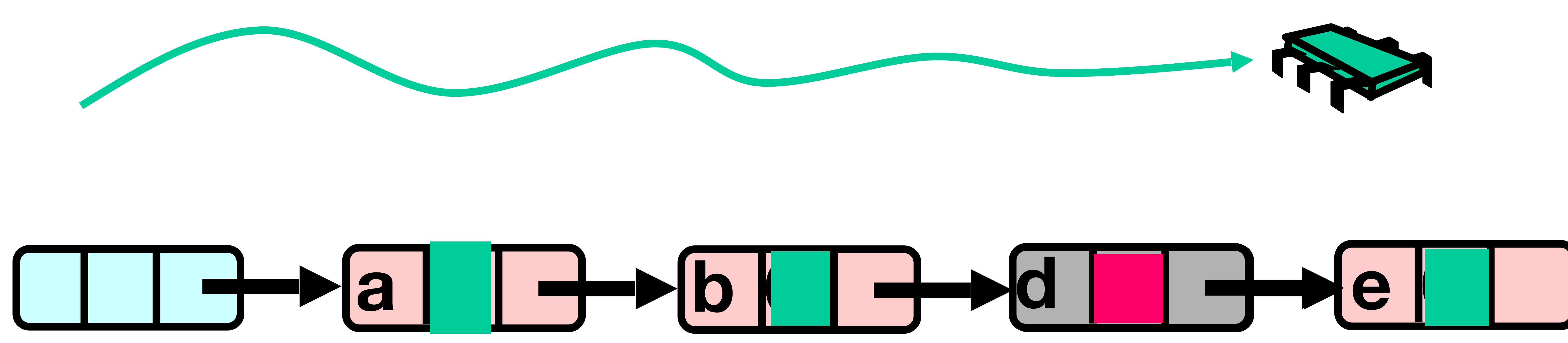
Validate (2)



Lazy Removal



Summary: Wait-free Contains



Use Mark bit + Fact that List is ordered

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

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….
- Today: wait-free algorithms and data structures!

Today

- Adding threads should not lower throughput - should increase throughput
  - Not possible if inherently sequential
  - How do we defeat this traffic jam?
- Reading: H&S 5.6-5.8, 9.8, 9.9
- Note: HW2 posted: https://www.jonbell.net/gmu-cs-475-fall-2019/cs475-f19-homework-2/

Read-Modify-Write Objects

- Method call
  - Returns object's prior value **x**
  - Replaces **x** with **F(x)**

Read-Modify-Write

```
public abstract class RMWRegister {
 private int value;

 public int synchronized
  getAndUpdate() {
    int prior  = this.value;
    this.value = magic(this.value);
    return prior;
  }
}
```

Read-Modify-Write

```
public abstract class RMWRegister {
 private int value;

 public int synchronized
  getAndMumble() {
    int prior  = this.value;
    this.value = magic(this.value);
    return prior;
  }
}
```

Return prior value

Read-Modify-Write

```
public abstract class RMWRegister {
 private int value;

 public int synchronized
  getAndMumble() {
    int prior  = this.value;
    this.value = magic(this.value);
    return prior;
  }
}
```

Apply function to current value

Example: getAndSet

```
public abstract class RMWRegister {
 private int value;

 public int synchronized
   getAndSet(int v) {
  int prior  = this.value;
  this.value = v;
  return prior;
 }
 …
}
```

Example: getAndSet (swap)

```
public abstract class RMWRegister {
 private int value;

 public int synchronized
   getAndSet(int v) {
  int prior  = this.value;
  this.value = v;
  return prior;
 }
 …
}
```

F(x)=v is constant function

compareAndSet

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

compareAndSet

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

If value is what was expected, …

compareAndSet

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

... replace it

compareAndSet

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

Report success

compareAndSet

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

Otherwise report failure

Lock-Free Increment

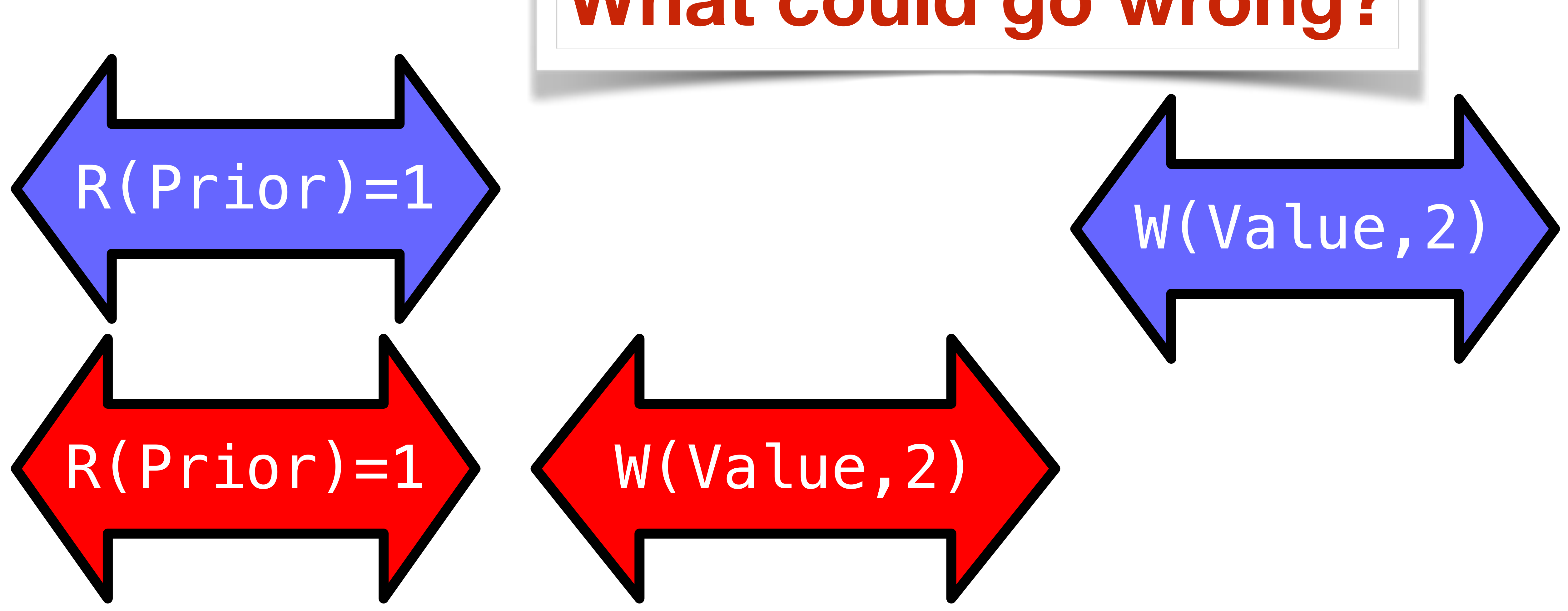
```
public abstract class Increment {
 private int value;

 public void increment() {
    int prior  = this.value;
    this.value = prior + 1;
  }
}
```

What could go wrong?

Lock-Free Increment

```
public abstract class Increment {
 private int value;

 public void increment() {
    int prior  = this.value;
    this.value = prior + 1;
  }
}
```

What could go wrong?



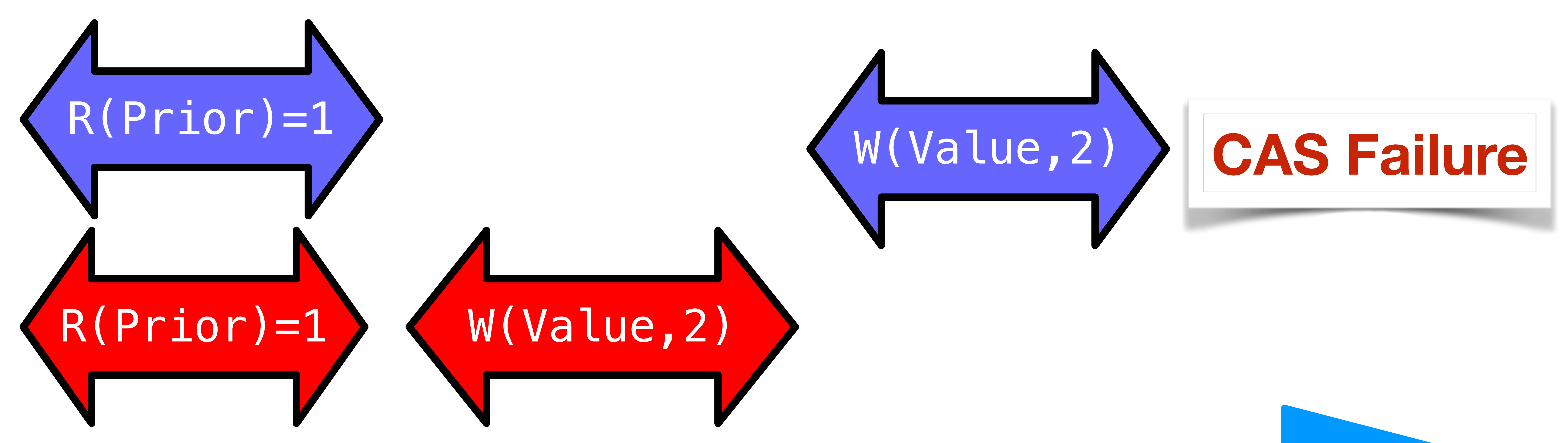
Lock-Free Increment

```
public abstract class Increment {
 private int value;

 public void increment() {
    int prior  = this.value;
    compareAndSet(prior, prior+1);
  }
}
```

If the value hasn't changed, increment it

Lock-Free Increment

```
public abstract class Increment {
 private int value;

 public void increment() {
    int prior  = this.value;
     compareAndSet(prior, prior+1);
  }
}
```



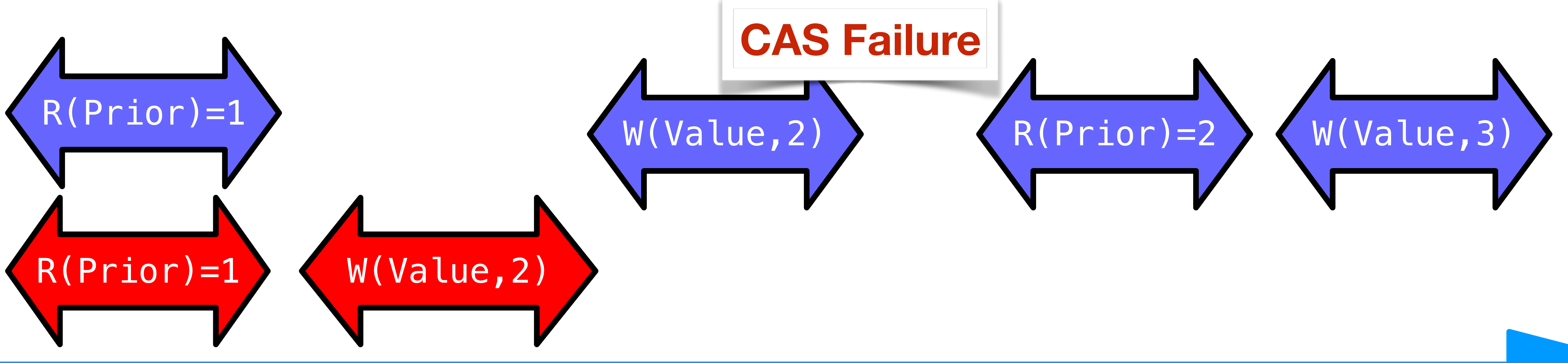
Lock-Free Increment: Actually Correct

```
public abstract class Increment {
 private int value;

 public void increment() {
    boolean success = false;
    while(!success){
      int prior  = this.value;
      success =compareAndSet(prior, prior+1);
    }
  }
}
```

Keep trying to do the CAS until it works.

Lock-Free Increment

```
public abstract class Increment {
 private int value;

 public void increment() {
     boolean success = false;
      while(!success){
      int prior  = this.value;
       success = compareAndSet(prior, prior+1);
      }
   }
}
```



Impact

- Many early machines provided these RMW instructions
  - Test-and-set (IBM 360)
  - Fetch-and-add (NYU Ultracomputer)
  - Swap (Original SPARCs)
- Modern CPUs provide a wide-range of RMW instructions
  - GetAndSet
  - CompareAndSwap
  - GetAndIncrement
  - …

Lock-Freedom

- Lock-free: in an infinite execution infinitely often some method call finishes (obviously, in a finite number of steps)
- Pragmatic approach
- Implies no mutual exclusion

Lock-Free vs. Wait-free

- Wait-Free: each method call takes a finite number of steps to finish
- Lock-free: infinitely often some method call finishes

Lock-Freedom

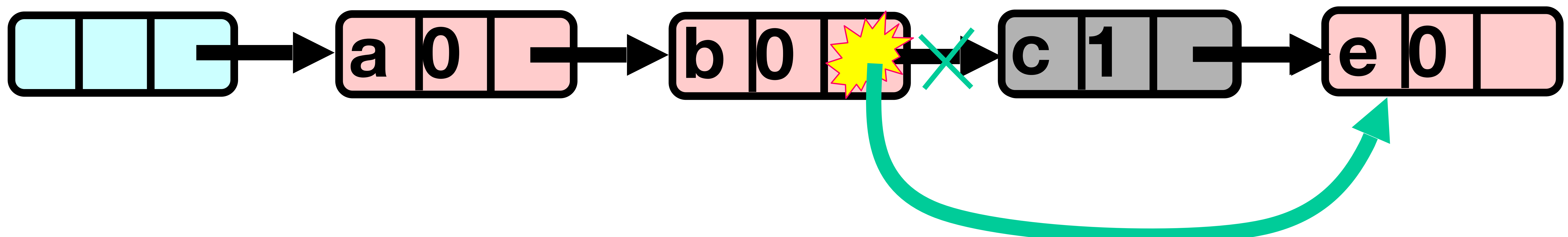
- Any wait-free implementation is lock-free.
- Lock-free is the same as wait-free if the execution is finite.
- Old saying: "Lock-free is to wait-free as deadlock-free is to lockout-free."

Lock-free Lists

- Next logical step
- Eliminate locking entirely
- contains() wait-free and add() and remove() lock-free
- Use only compareAndSet()
- What could go wrong?

Remove Using CAS

Logical Removal = Set Mark Bit



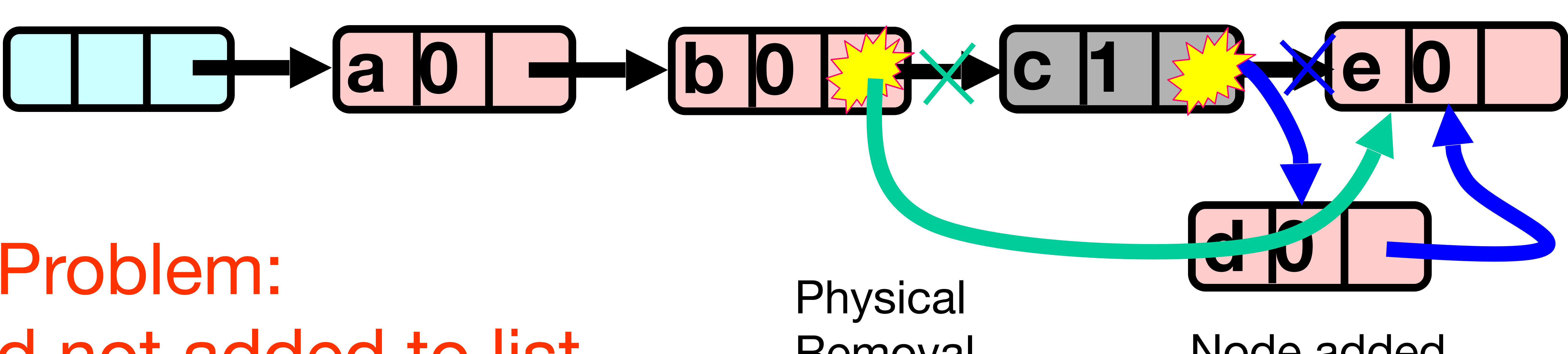
Use CAS to verify pointer is correct

Physical Removal CAS pointer

Not enough!

Problem…

Logical Removal =
Set Mark Bit



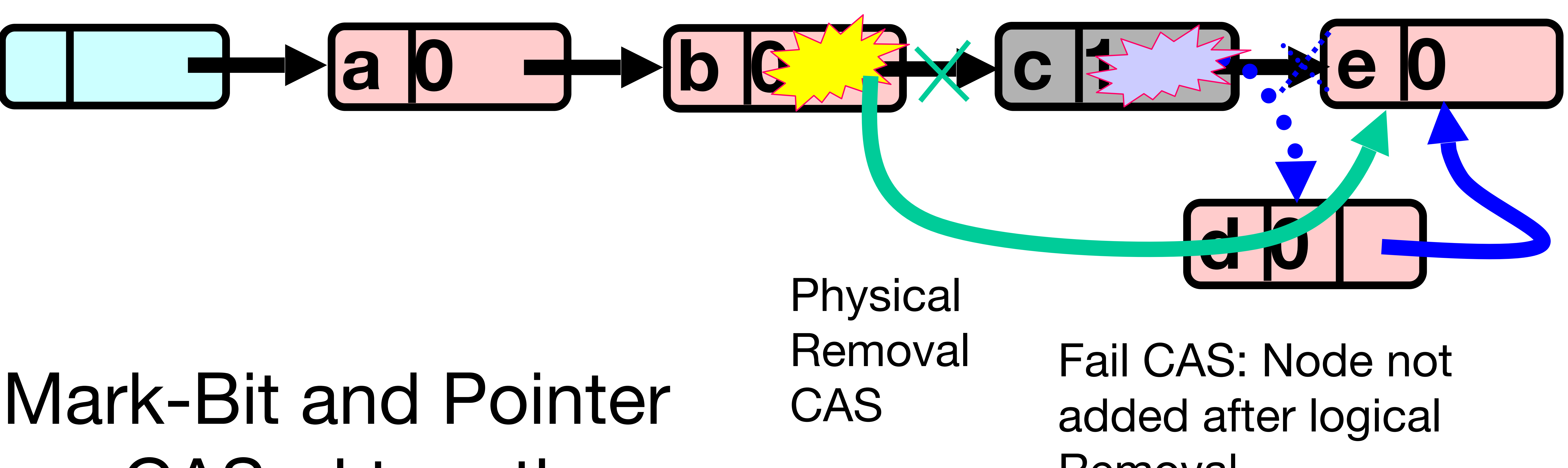
Physical
Removal
CAS

Node added
Before
Physical
Removal CAS

Problem:
d not added to list…
Must Prevent
manipulation of
removed node’s pointer

The Solution: Combine Bit and Pointer

Logical Removal =
Set Mark Bit



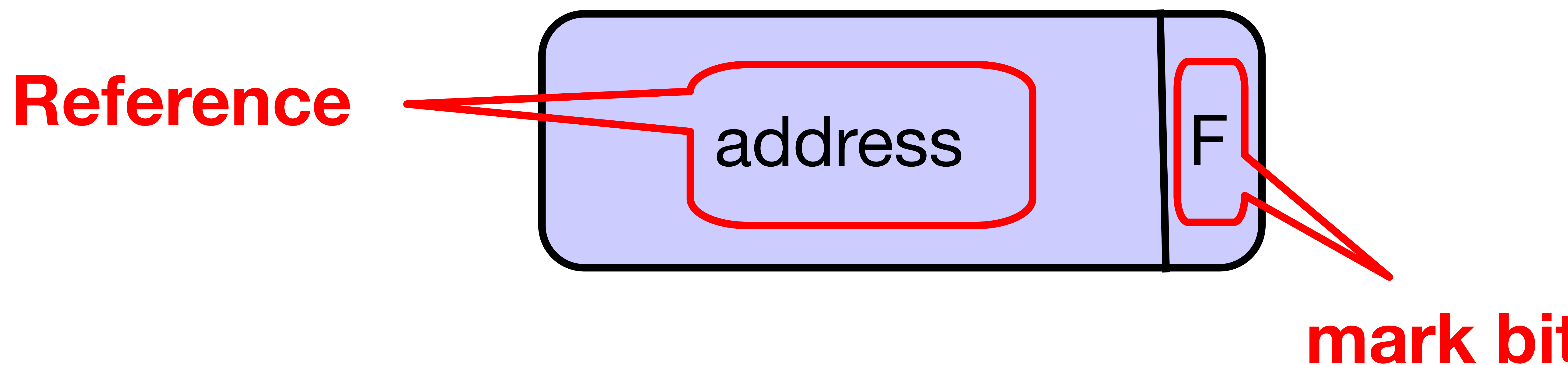
Mark-Bit and Pointer
are CASed together
(AtomicMarkableReference)

Solution

- Use AtomicMarkableReference
- Atomically
  - Swing reference and
  - Update flag
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor's pointer

Marking a Node

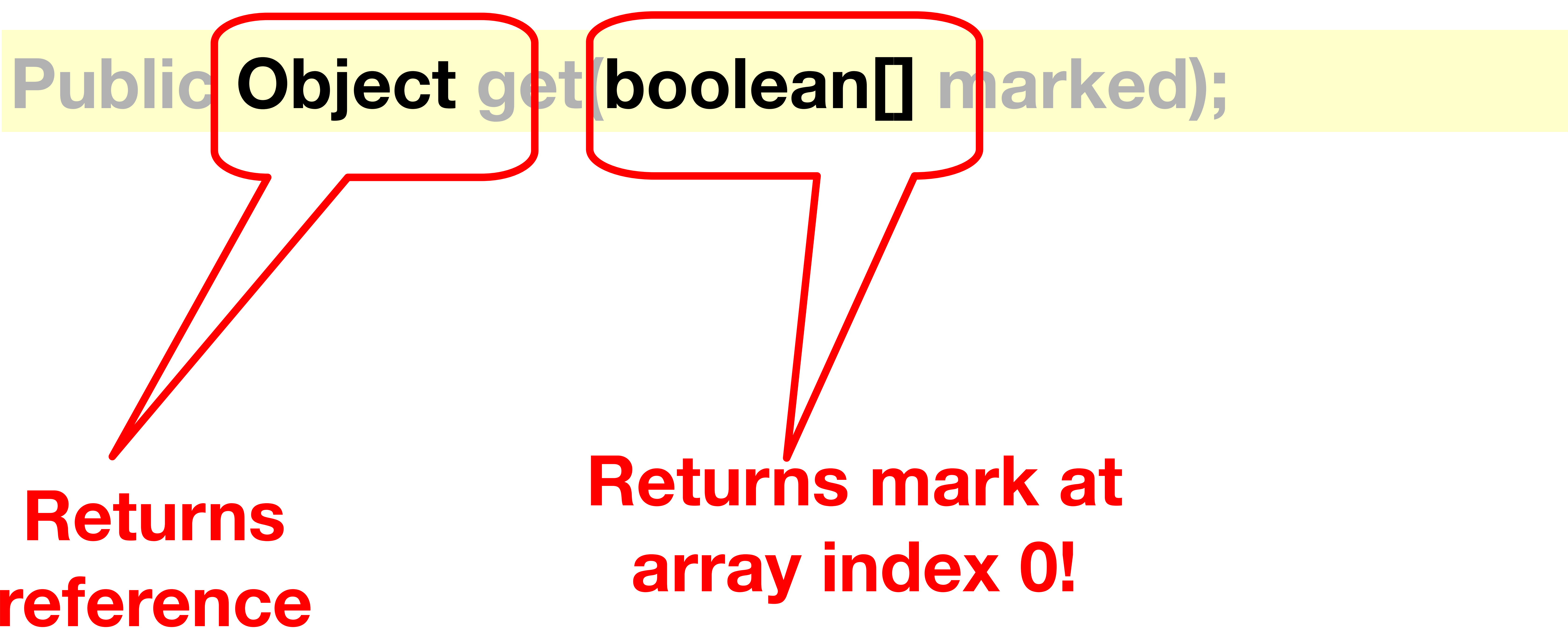
- AtomicMarkableReference class
  - Java.util.concurrent.atomic package



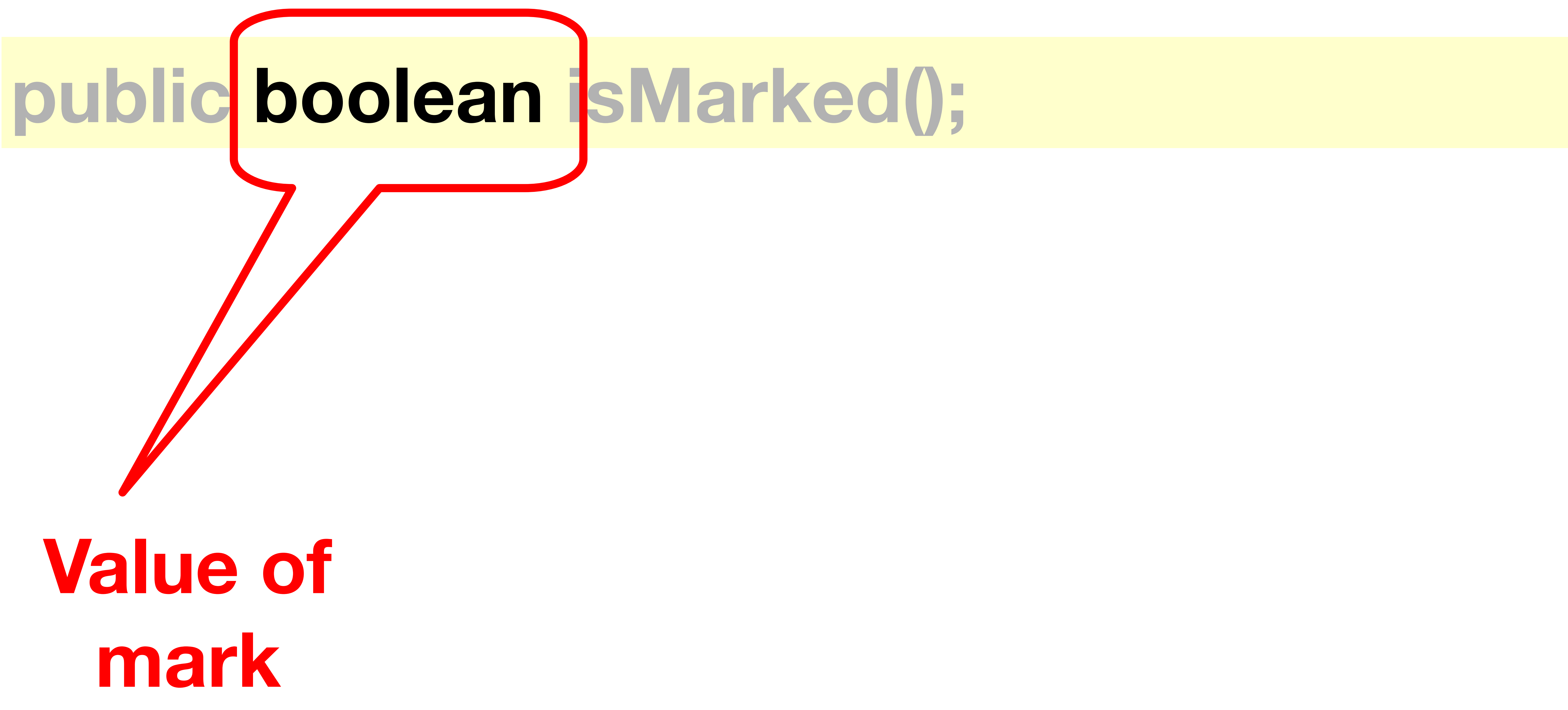
Extracting Reference & Mark

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

Extracting Reference & Mark



Extracting Reference Only



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

...then change to this new reference ...

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

... and this new mark

Changing State

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

Changing State

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

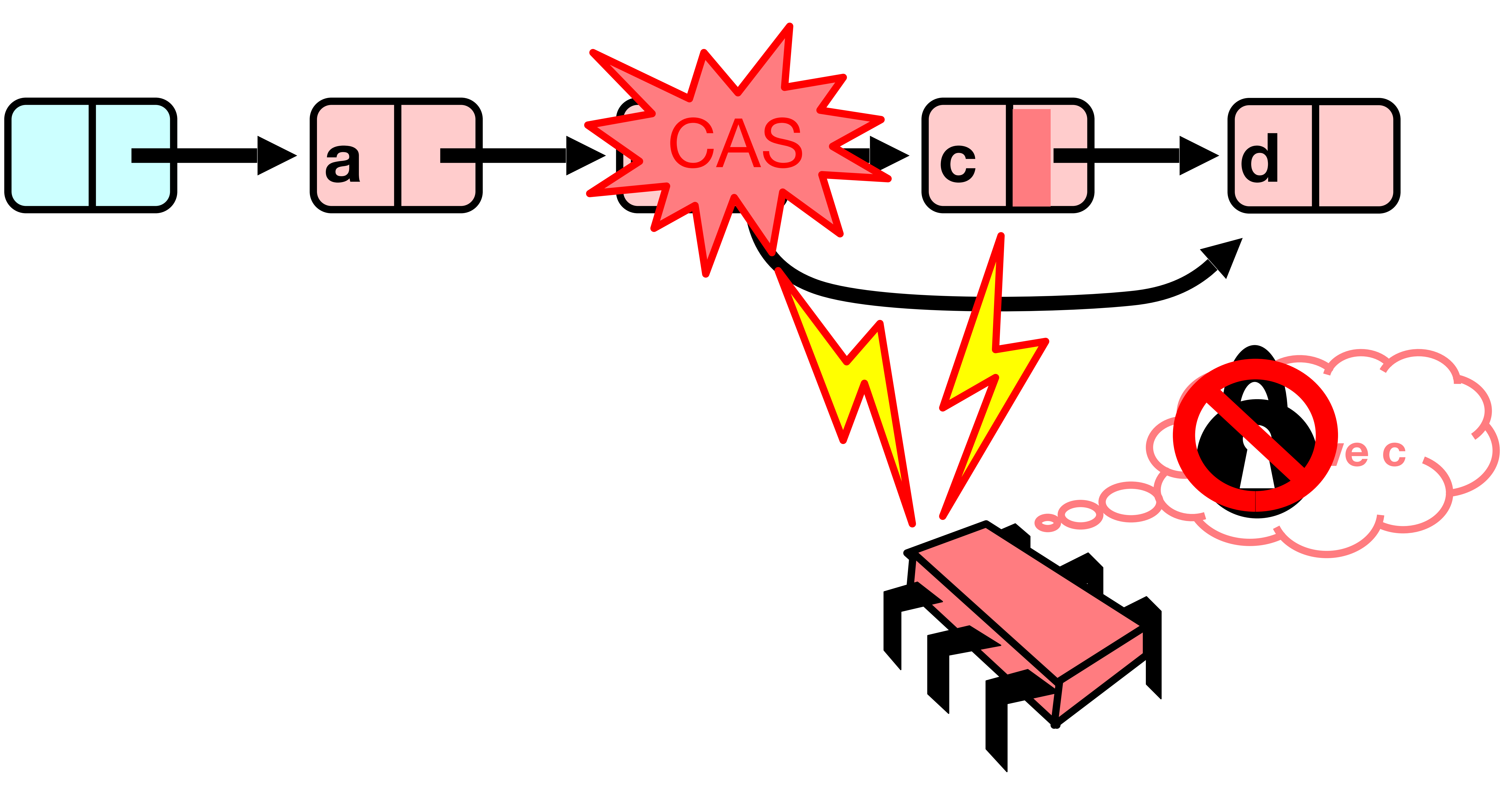
If this is the current reference …

Changing State

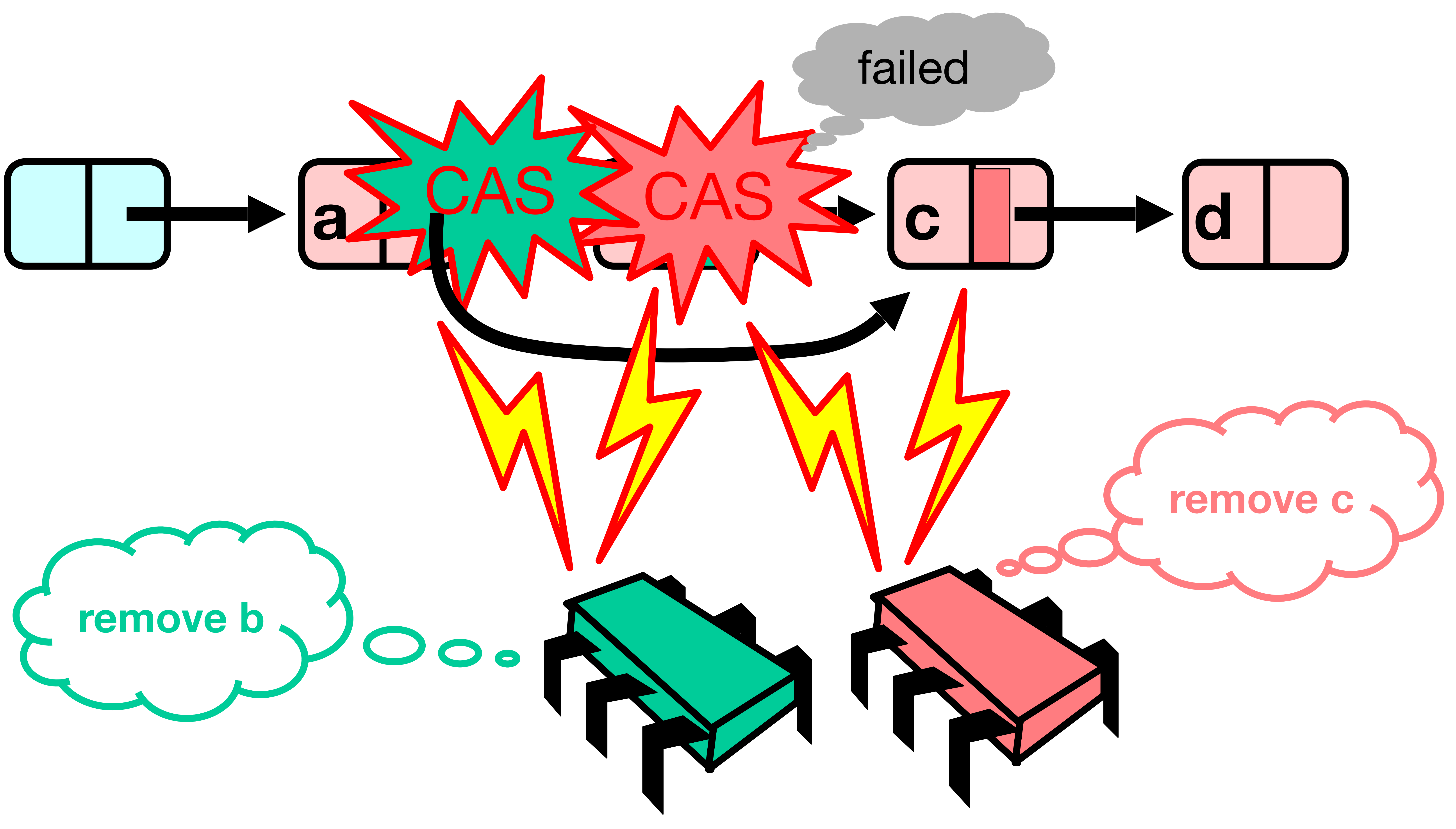
```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```

.. then change to this new mark.

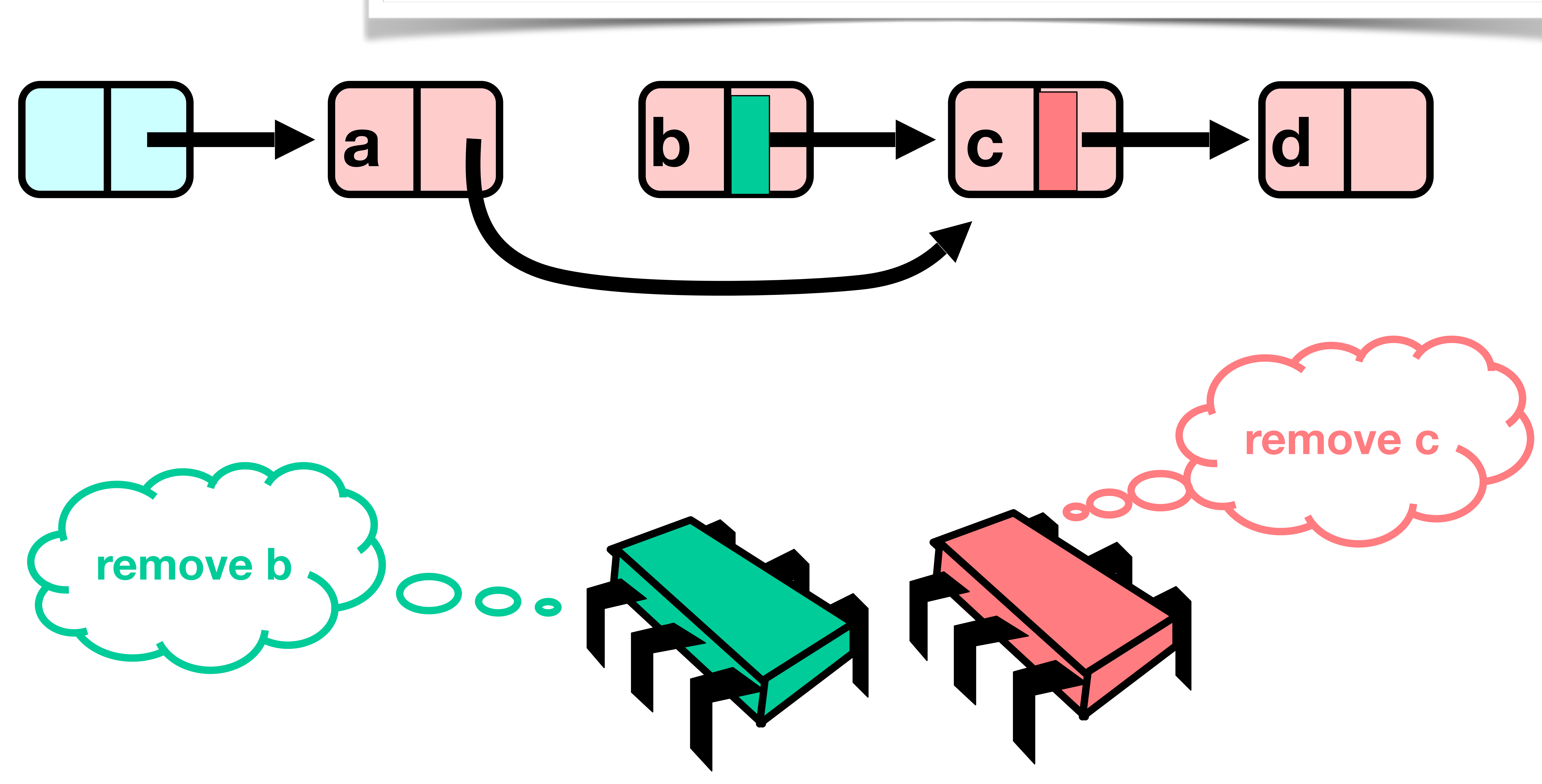
Removing a Node



Removing a Node



Removing a Node



Removing a Node

C is still in the list, but its mark field is set, so is it "deleted"?

a

d

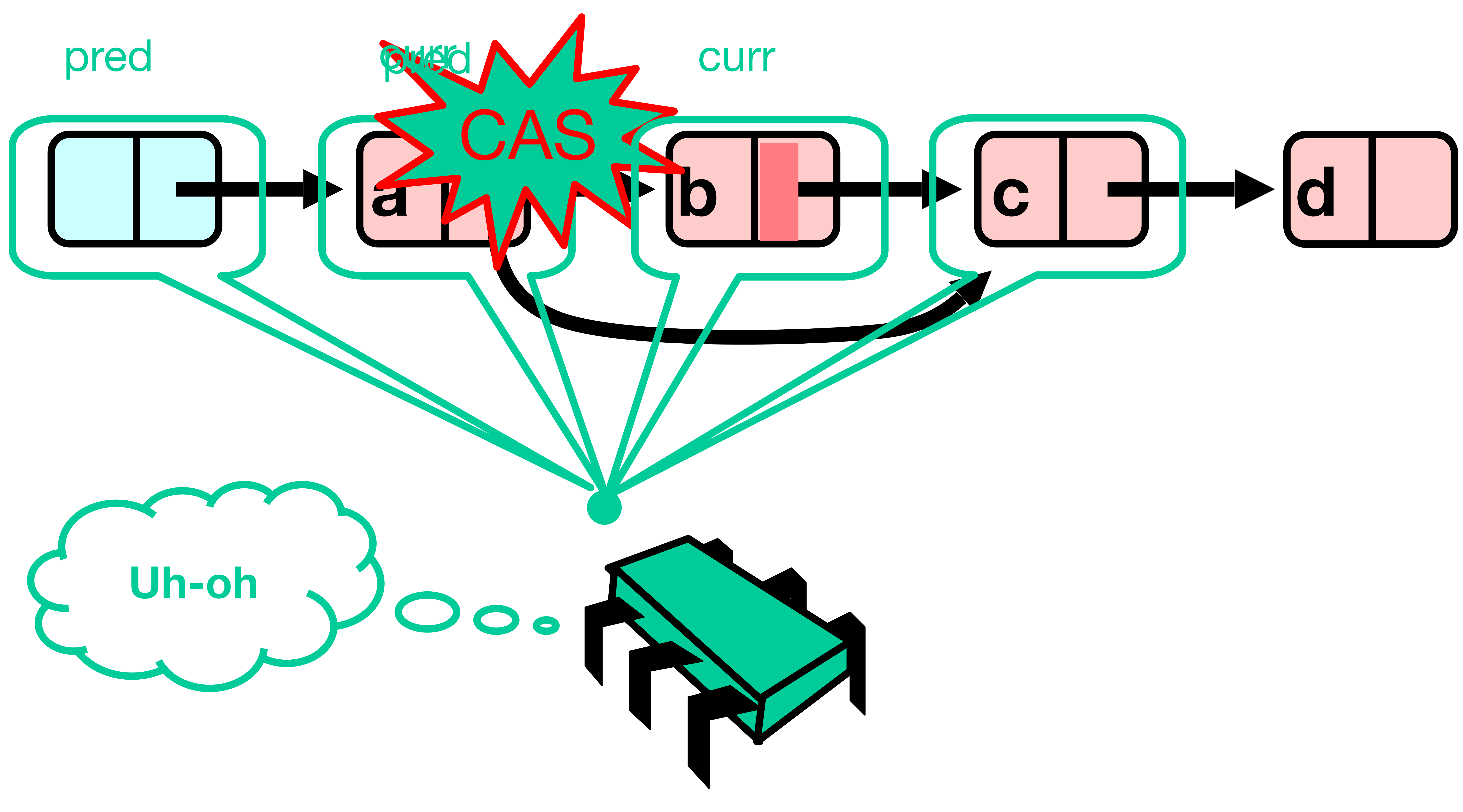
remove b

remove c

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) {
   this.pred = pred; this.curr = curr;
 }
}
```

The Window Class

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

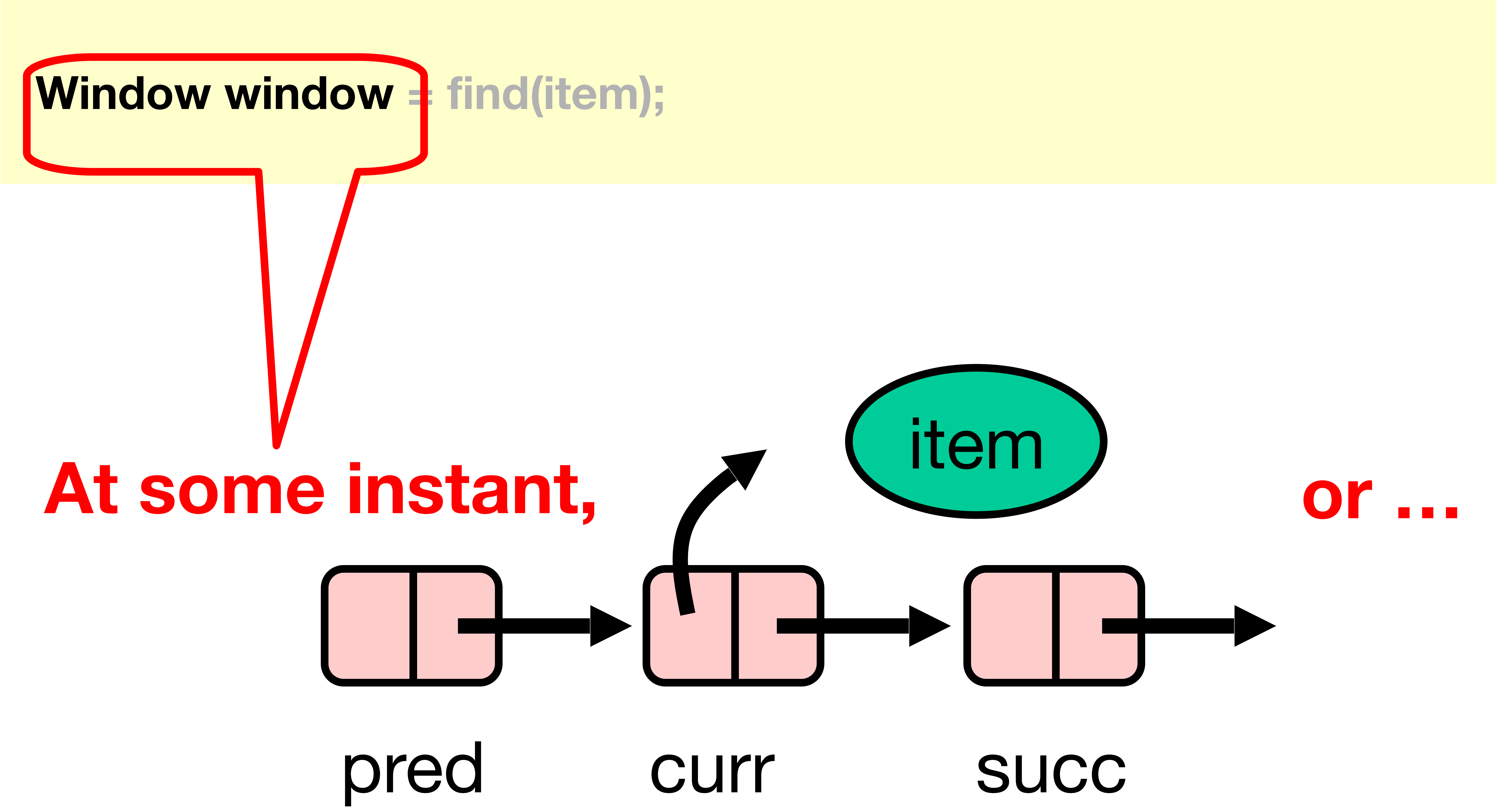
Find returns window

Using the Find Method

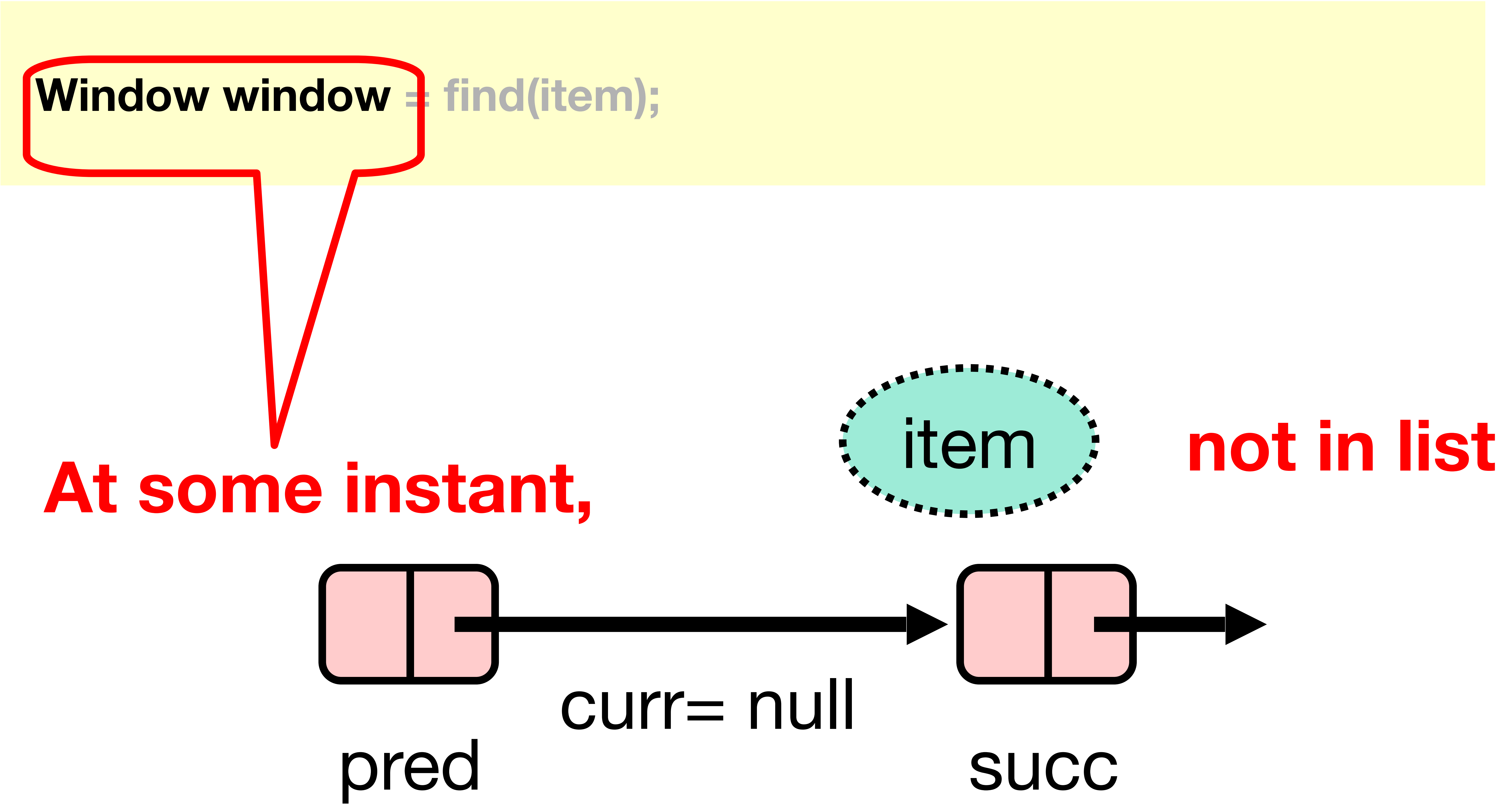
```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

Extract pred and curr

The Find Method



The Find Method



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.attemptMark(succ, 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.attemptMark(succ, 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.attemptMark(succ, 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.attemptMark(succ, true);
  if (!snip) continue;
   pred.next.compareAndSet(curr, succ, false, false);
    return true;
}}}
```

It's not there …

Remove

Try to mark node as deleted

```
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.attemptMark(succ, 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) {
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.attemptMark(succ, 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 = wi
  if (curr.key != key) {
    re
  } el
  Node succ = curr.next.getReference();
  snip = curr.next.attemptMark(succ, 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).

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, 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;}
}}}
```

create new node

Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   N                              window.curr;
   if

   }
   N
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false, false)) {return true;}
}}}
```

Install new node, else retry loop

Wait-free Contains

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

Only diff is that we get and check marked

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 because we start over only if someone else makes progress

Lock-free Find

```
public Window find(Node head, int key) {
 Node pred = null, curr
 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;
       curr = succ;
   }
}}
```

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 (true) {
    succ = curr.next.get(marked);
    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 must 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

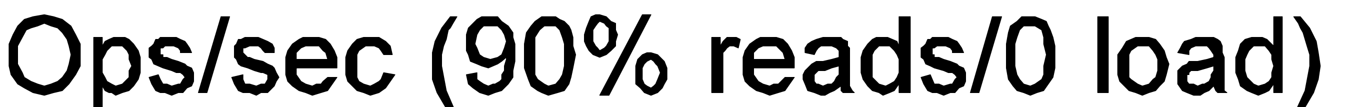
Otherwise move on to check if next node deleted

```
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

On 16 node shared memory machine
Benchmark throughput of Java List-based Set algs. Vary % of Contains() method Calls.

High Contains Ratio



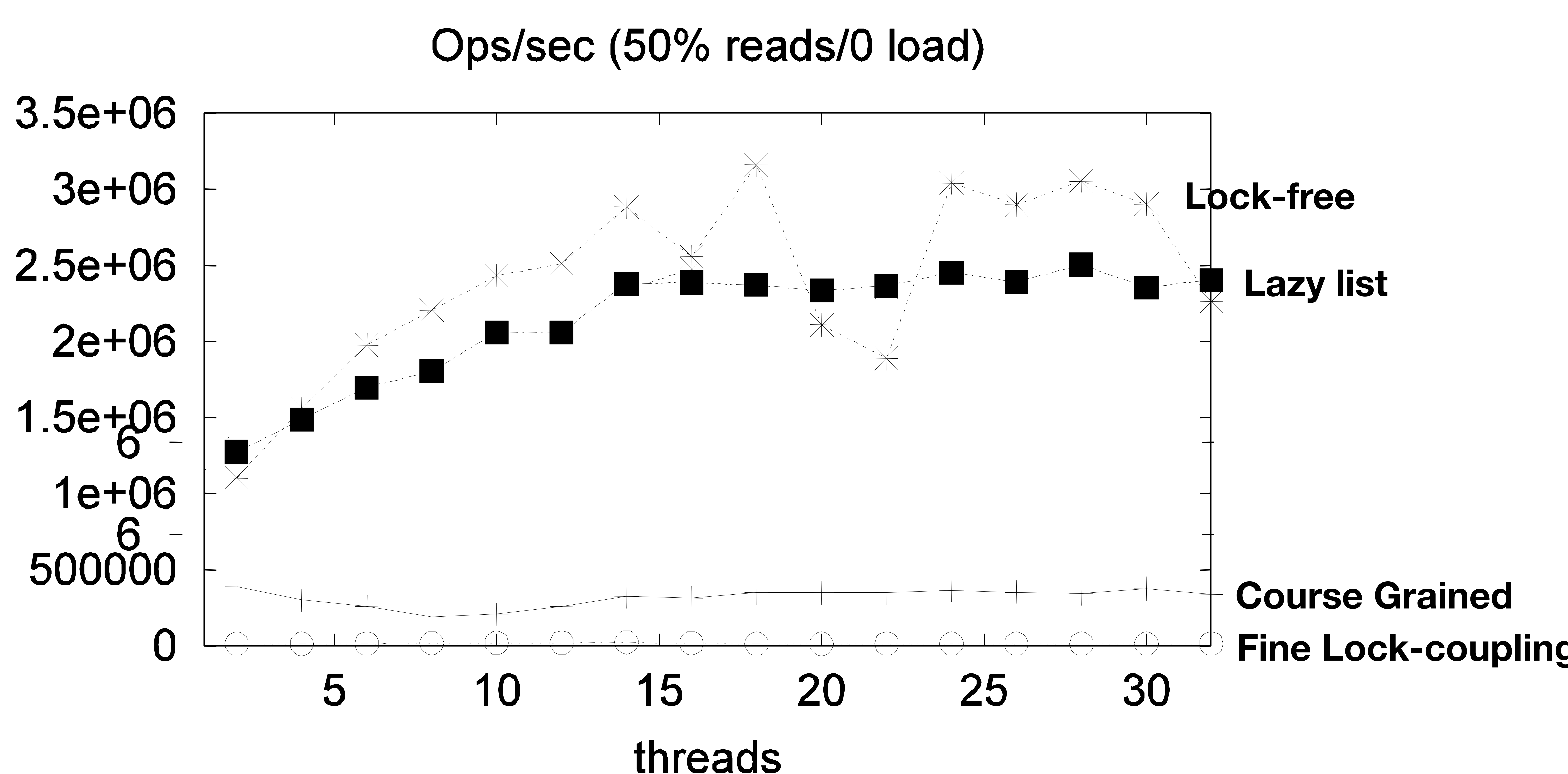
1.2e+07
1e+07
8e+06
6e+06
4e+0
2e+0
0

5
10
15
20
25
30

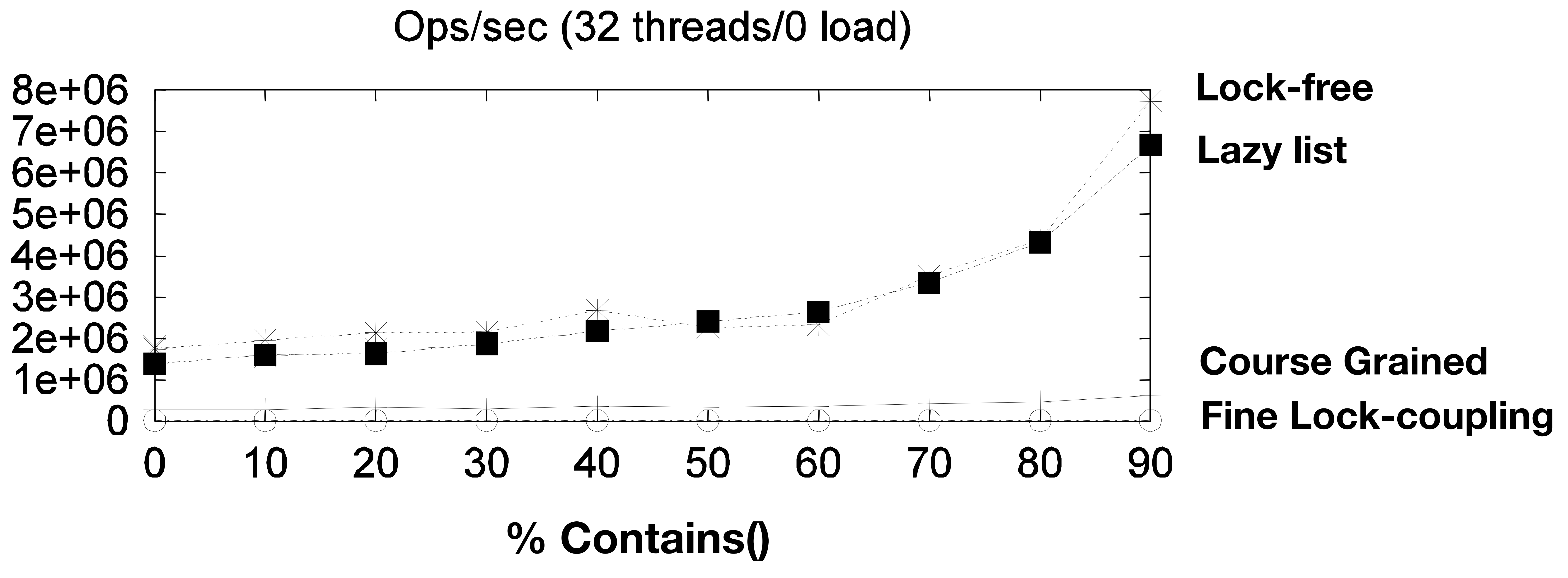
threads

Lock-free
Lazy list
Course Grained
Fine Lock-coupling

Low Contains Ratio



As Contains Ratio Increases



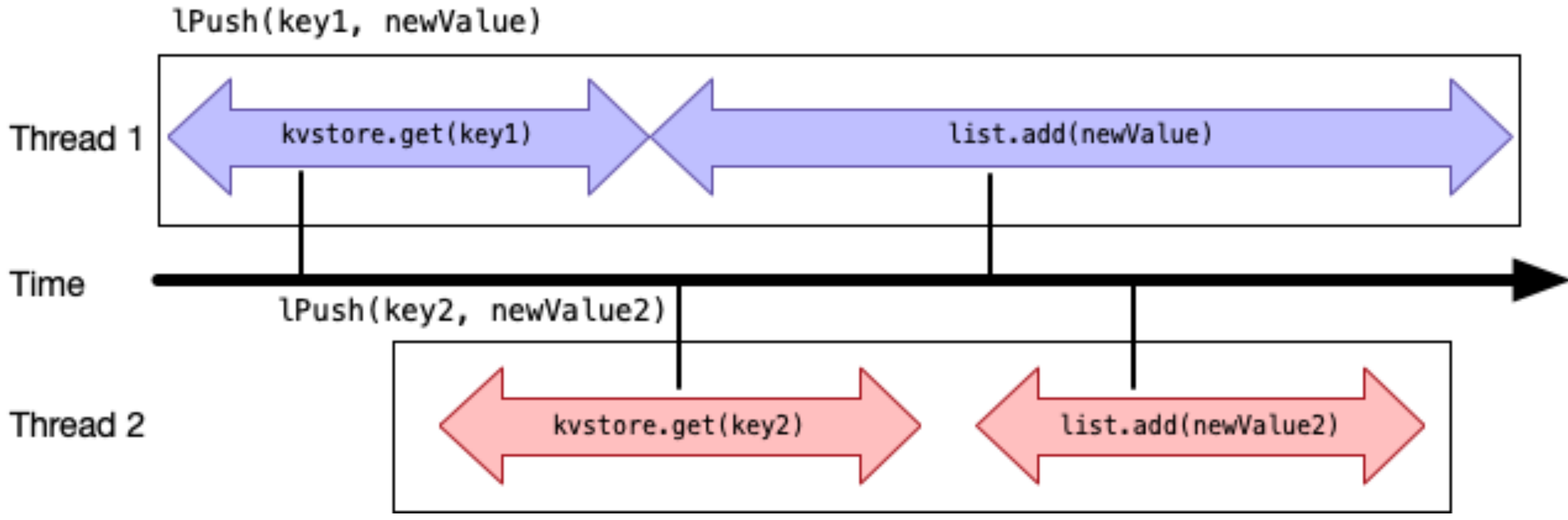
Summary

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

“To Lock or Not to Lock”

- Locking vs. Non-blocking: Extremist views on both sides
- The answer: nobler to compromise, combine locking and non-blocking
  - Example: Lazy list combines blocking add() and remove() and a wait-free contains()
  - Remember: Blocking/non-blocking is a property of a method

Discussion - HW1 Part 3



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.