Linked Lists: Locking, Lock-Free, and Beyond …

Maurice Herlihy

CS176

Fall 2003

Linked Lists

- We can make effective spin locks
  - Correct
  - Work well under contention
- Are we done with concurrent data structures?

Not Over Yet

- Contention
  - Solved by MCS or CLH locks
- Sequential Bottleneck
  - No "bag of tricks"
- Linked Lists
  - Simple data structure
  - Good testbed

Set Interface

- Unordered collection of objects
- No duplicates
- Methods
  - Add a new object
  - Remove an object
  - Test if object is present

List-Based Sets

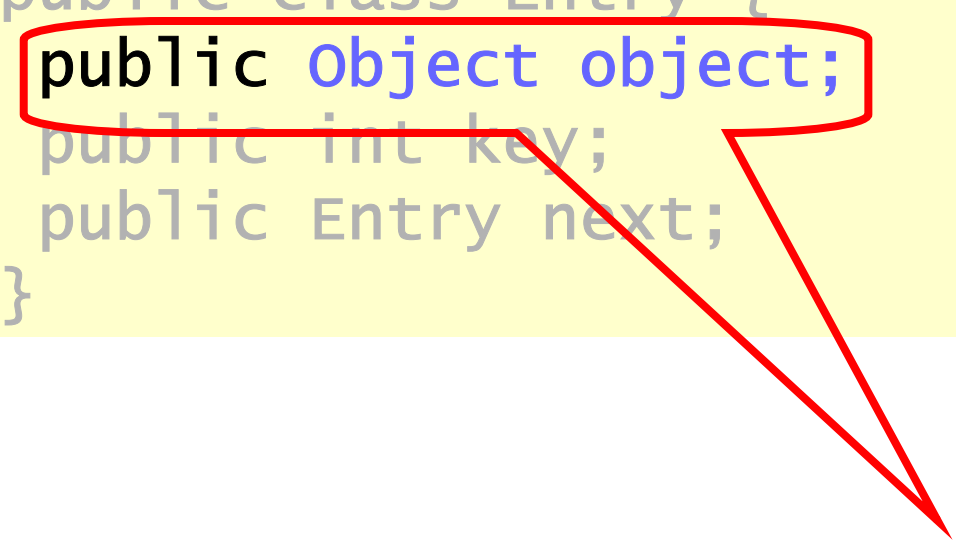
```
public interface Set {
 public boolean add(Object x);
 public boolean remove(Object x);
 public boolean contains(Object x);
}
```

List Entry

```
public class Entry {
 public Object object;
 public int key;
 public Entry next;
}
```

List Entry

```
public class Entry {
 public Object object;
  public int key;
  public Entry next;
}
```



Object of interest

List Entry

```
public class Entry {
 public Object object;
 public int key;
 public Entry next;
}
```

Sort by key value (usually hash code)

List Entry

```
public class Entry {
 public Object object;
 public int key;
 public Entry next;
}
```

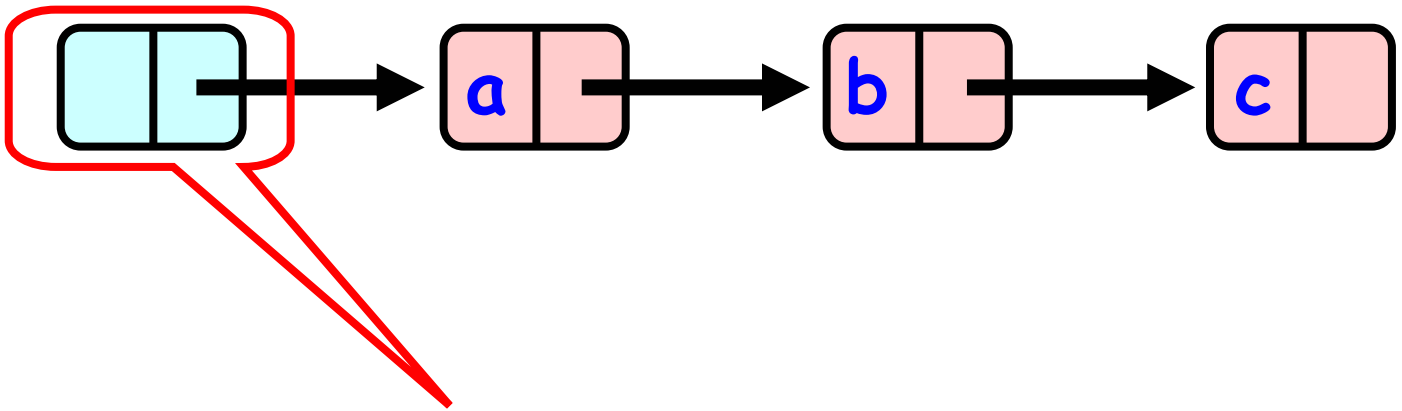
Sorting makes it easy to detect absence

List Entry

```
public class Entry {
 public Object object;
 public int key;
 public Entry next;
}
```

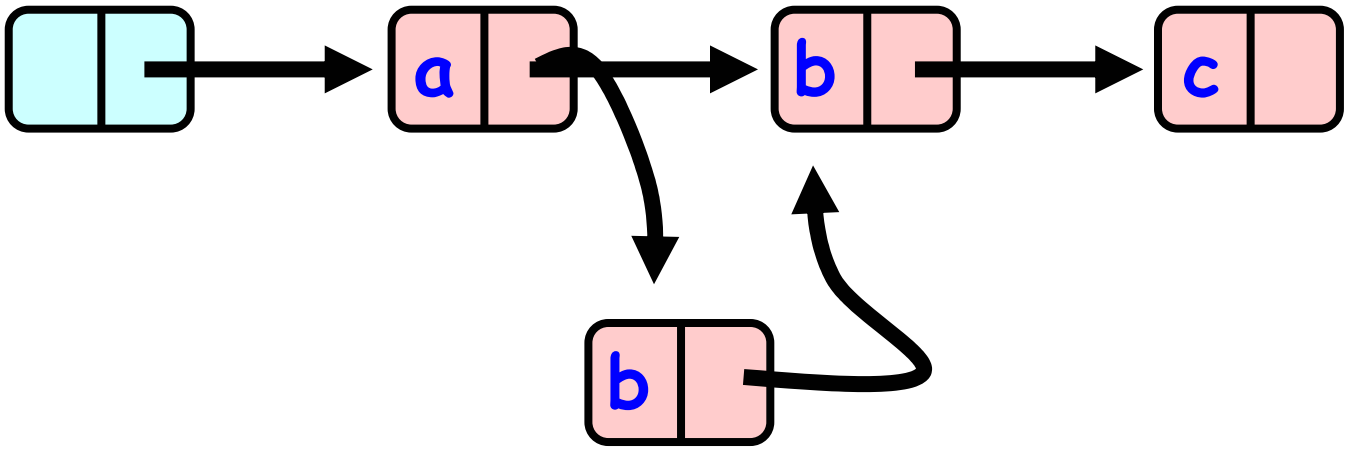
Reference to next entry

List-Based Set

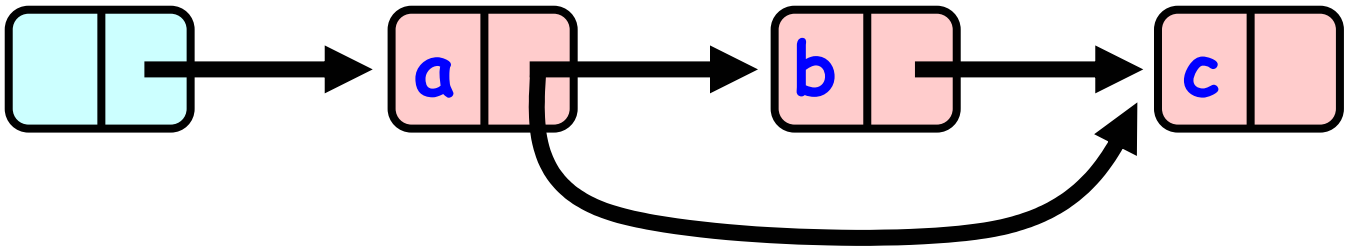


Sentinel node never deleted (minimum possible key)

Adding an Entry



Removing an Entry



Nu, What About Concurrency?

- Our bag of tricks
  - Coarse-grained locks
  - Fine-grained locks
  - Optimistic synchronization
  - Lock-free synchronization

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

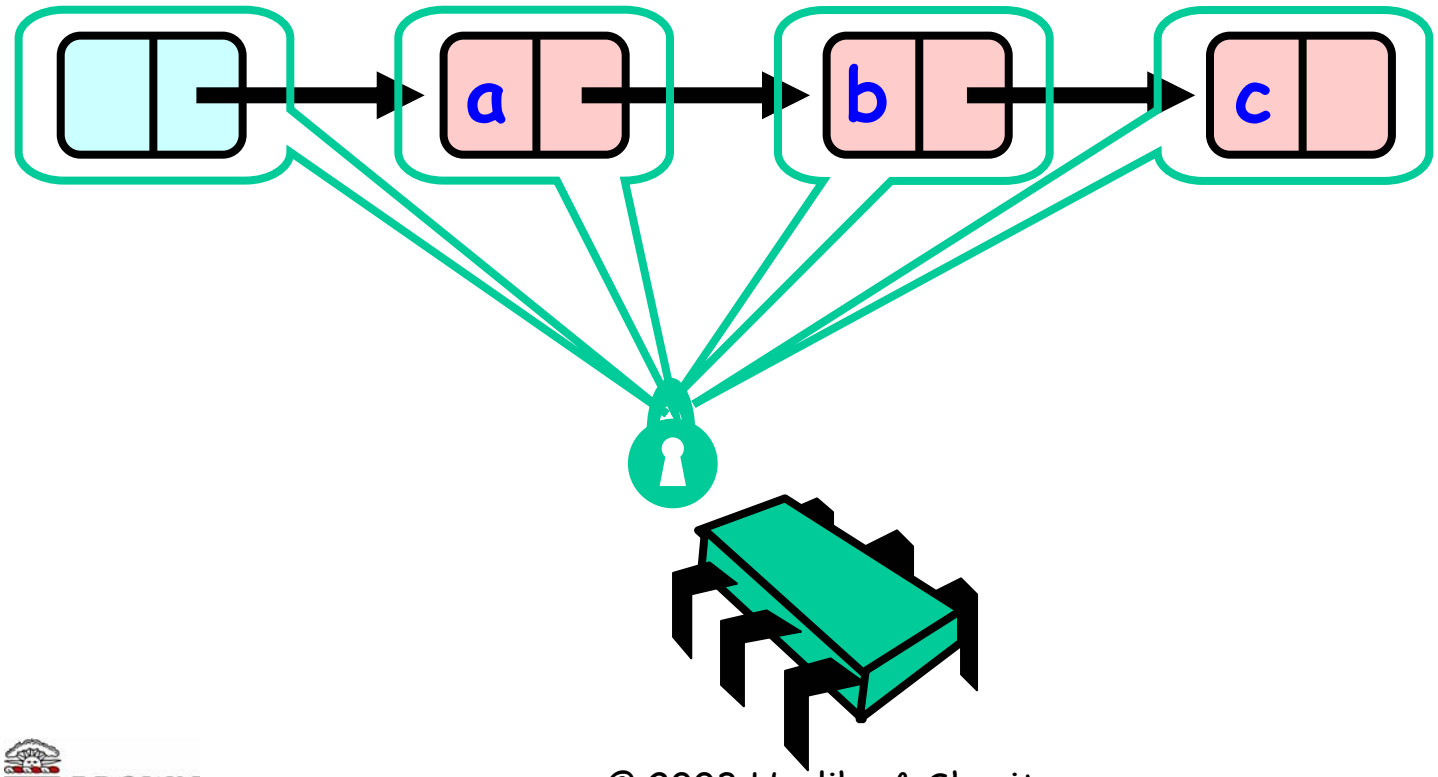
Optimistic Synchronization

- Requires very careful thought
  - "Do not meddle in the affairs of dragons, for you are crunchy and taste good with ketchup."
- Try it without synchronization
  - If you win, you win
  - If not, try it again with synchronization

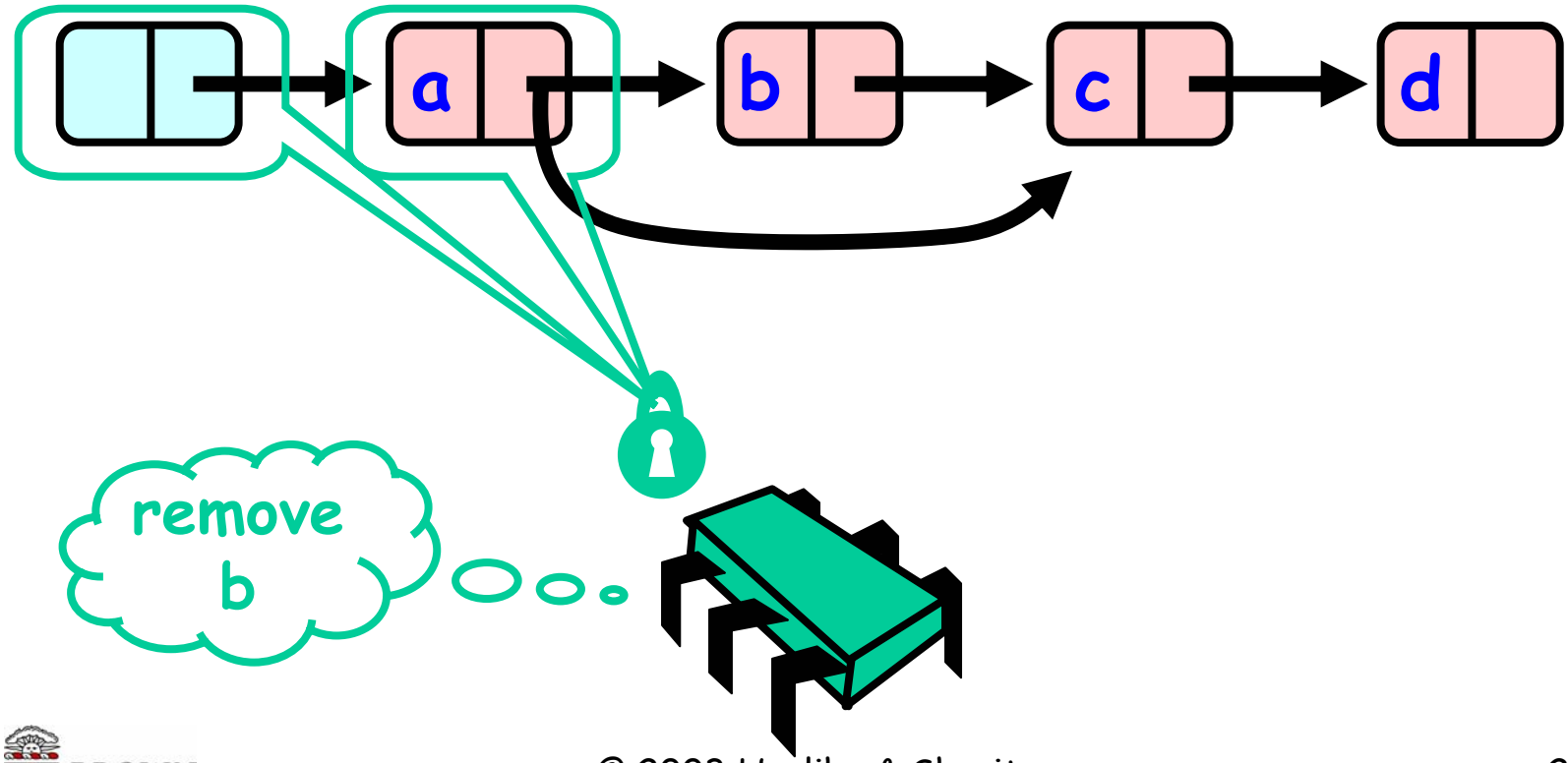
Lock-Free Synchronization

- Dump locking altogether …
  - "You take the red pill and you stay in Wonderland and I show you how deep the rabbit-hole goes"
- No locks, just native atomic methods
  - Usually compareAndSet

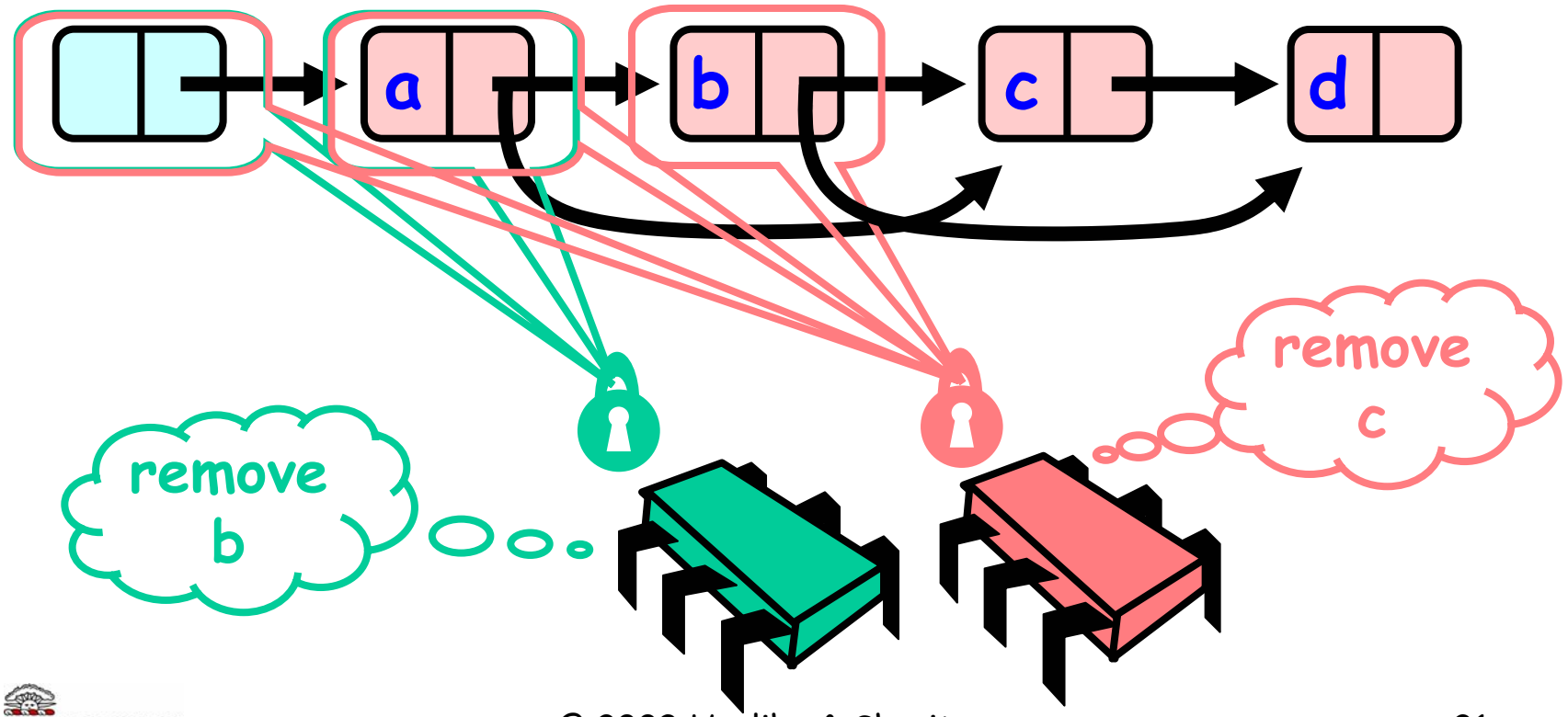
Hand-over-Hand locking



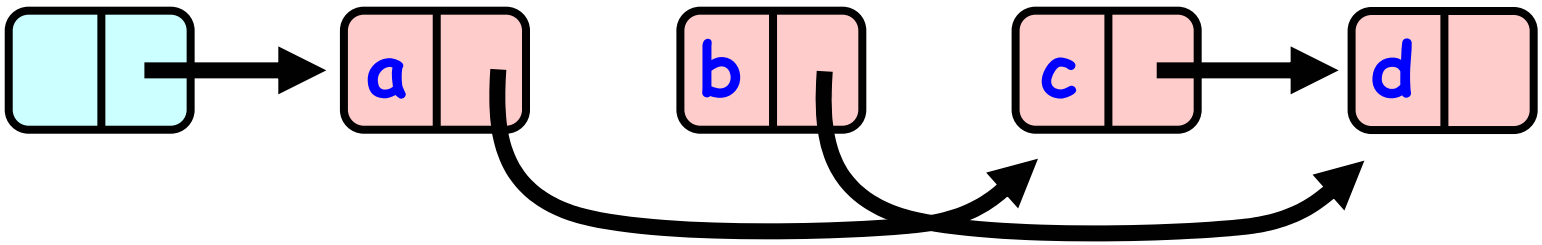
Removing an Entry

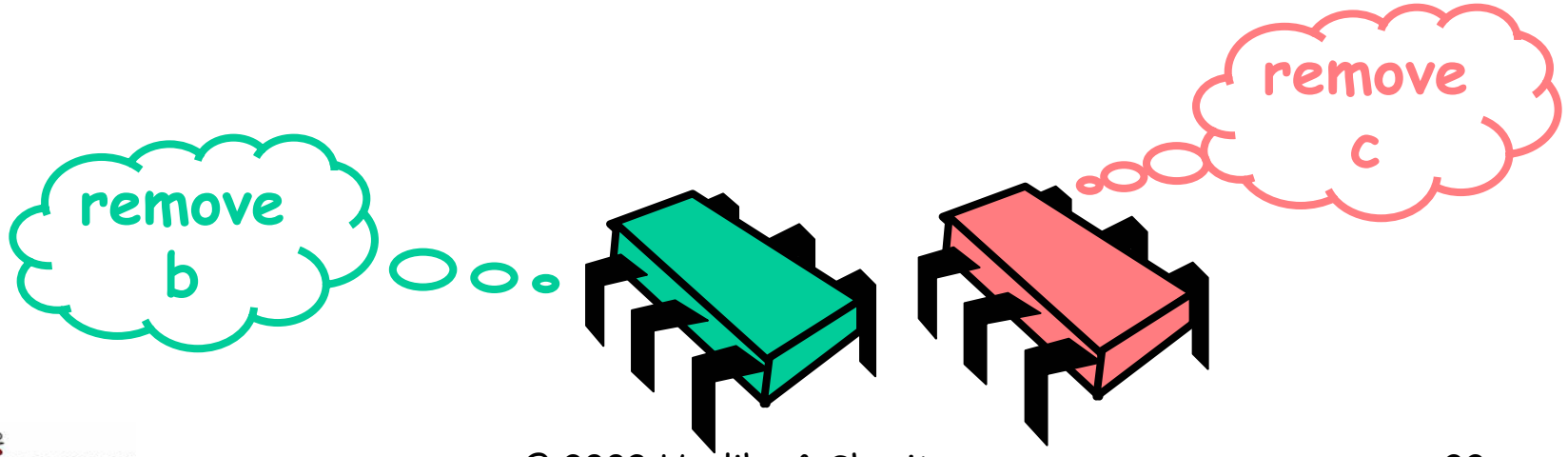


Removing an Entry



Uh-oh





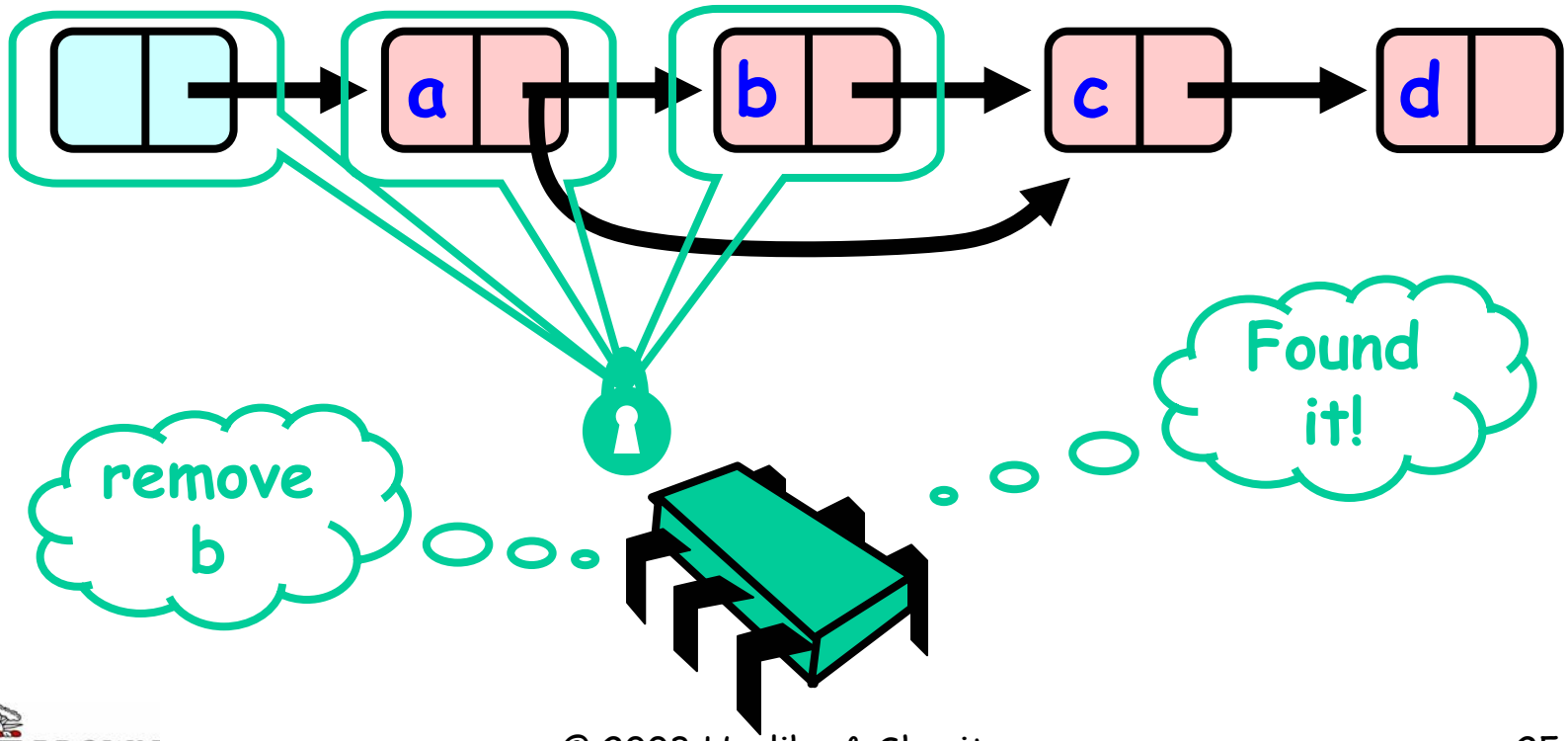
Problem

- To delete entry b
  - Swing entry a's next field to c
- Problem is,
  - Someone could delete c concurrently

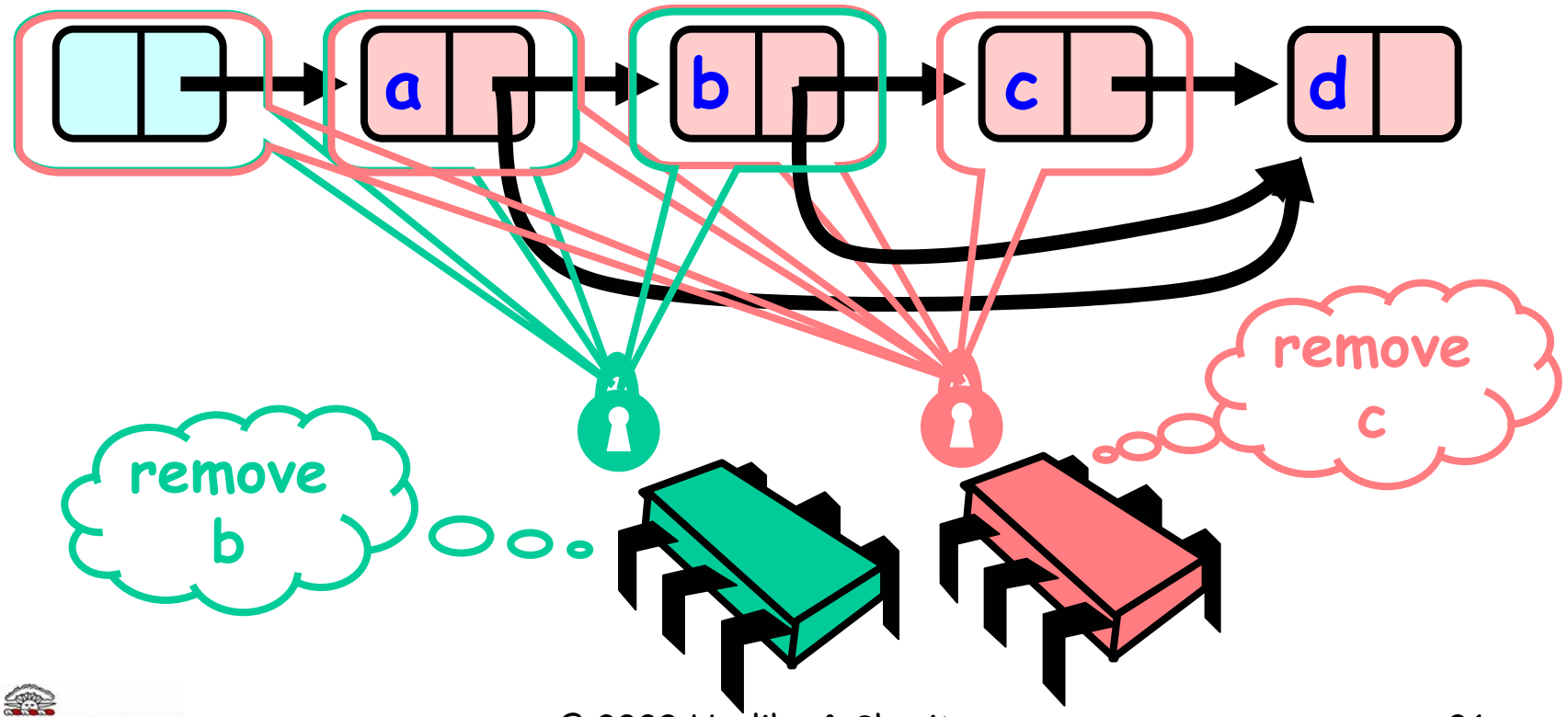
Insight

- If an entry is locked
  - No one can delete entry's *successor*
- If a thread locks
  - Entry to be deleted
  - And its predecessor
  - Then it works

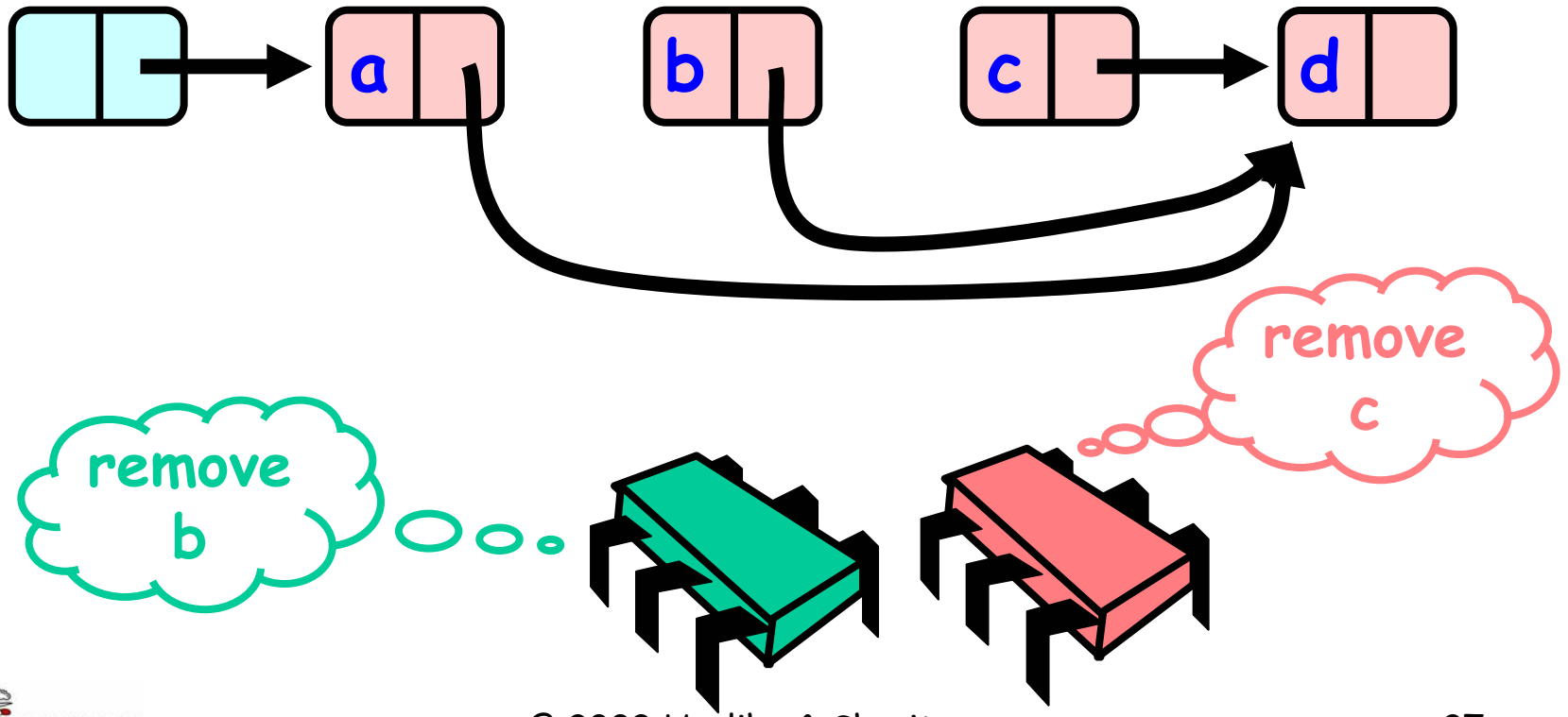
Hand-Over-Hand Again



Removing an Entry



Removing an Entry



Remove method

```
public boolean remove(Object object) {
 int key = object.hashCode();
 Entry predEntry, currEntry;
 try {
   …
 } finally {
  currEntry.unlock();
  predEntry.unlock();
 }}
```

Remove method

```
public boolean remove(Object object) {
 int key = object.hashCode();
 Entry prevEntry, currEntry;
 try {
   …
 } finally {
  currEntry.unlock();
  prevEntry.unlock();
 }}
```

Key used to order entry

Remove method

```
public boolean remove(Object object) {
 int key = object.hashCode();
 Entry predEntry, currEntry;
 try {
   …
 } finally {
  currEntry.unlock();
  prevEntry.unlock();
 }}
```

Predecessor and current entries

Remove method

```
public boolean remove(Object object) {
 int key = object.hashCode();
 Entry prevEntry, currEntry;
 try {
   …
 } finally {
  currEntry.unlock();
  predEntry.unlock();
 }}
```

Make sure locks released

Remove method

```
public boolean remove(Object object) {
 int key = object.hashCode();
 Entry prevEntry, currEntry;
 try {
   ...
 } finally {
  currEntry.unlock();
  prevEntry.unlock();
 }}
```

Everything else

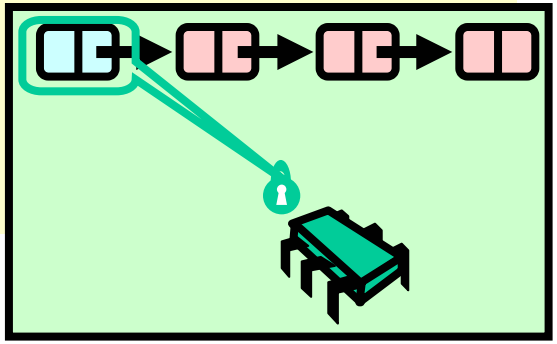
Remove method

```
try {
  predEntry = this.head;
  predEntry.lock();
  currEntry = predEntry.next;
  currEntry.lock();
  …
} finally { … }
```

Remove method

```
try {
  predEntry = this.head;
  predEntry.lock();
  currEntry = predEntry.next;
  currEntry.lock();
  …
} finally { … }
```

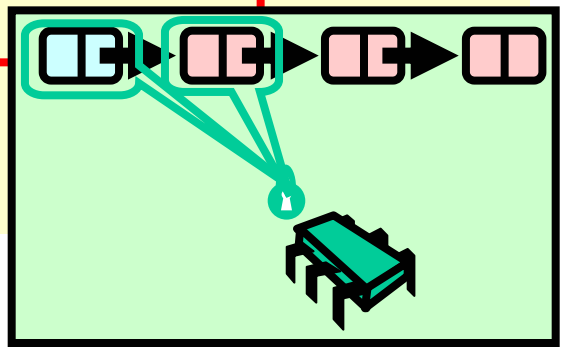
lock previous



Remove method

```
try {
  prevEntry = this.head;
  prevEntry.lock();
  currEntry = predEntry.next;
  currEntry.lock();
  …
} finally { … }
```

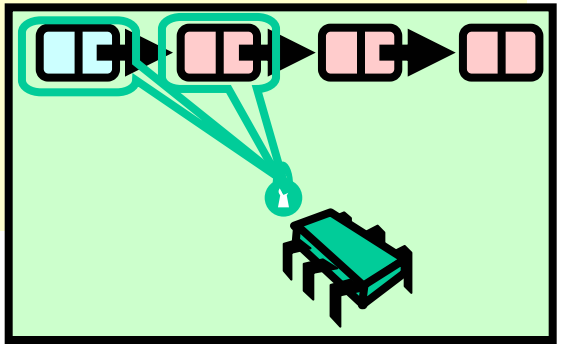
Lock current



Remove method

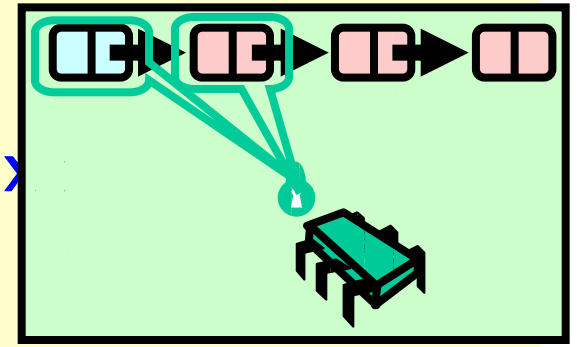
```
try {
  prevEntry = this.head;
  prevEntry.lock();
  currEntry = prevEntry.next;
  currEntry.lock();
  ...
} finally { … }
```

Traversing list



Remove: searching

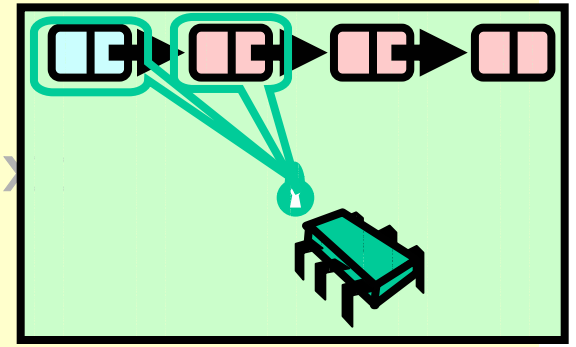
```
while (currEntry.key <= key) {
  if (object == currEntry.object) {
   predEntry.next = currEntry.next;
   return true;
  }
  predEntry.unlock();
  predEntry = currEntry;
  currEntry = currEntry.nex
  currEntry.lock();
 }
 return false;
```



Remove: searching

```
while (currEntry.key <= key) {
  if (object == currEntry.object) {
   prevEntry.next = currEntry.next;
   return true;
  }
  prevEntry.unlock();
  prevEntry = currEntry;
  currEntry = currEntry.nex
  currEntry.lock();
 }
 return false;
```

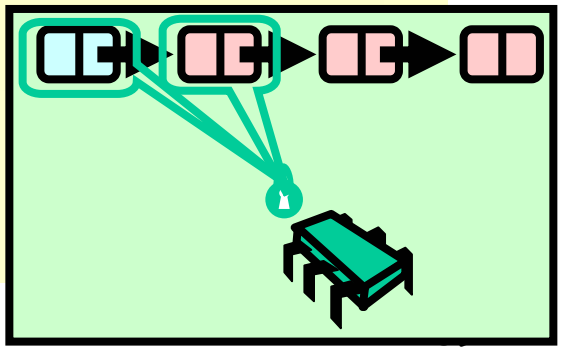
Search key range



Remove: searching

```
while (currEntry.key <= key) {
  if (object == currEntry.object) {
    prevEntry.next = currEntry.next;
    return true;
  }
  prevEntry
  prevEntry
  currEntry
  currEntry.lock();
}
return false;
```

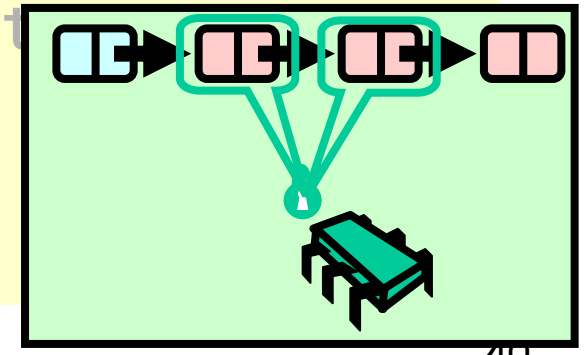
At start of each loop:
currEntry and predEntry locked



Remove: searching

```
while (currEntry.key <= key) {
  if (object == currEntry.object) {
   predEntry.next = currEntry.next;
   return true;
  }
  prevEntry.unlock();
  prevEntry = currEntry;
  currEntry = currEntry.next
  currEntry.lock();
 }
```

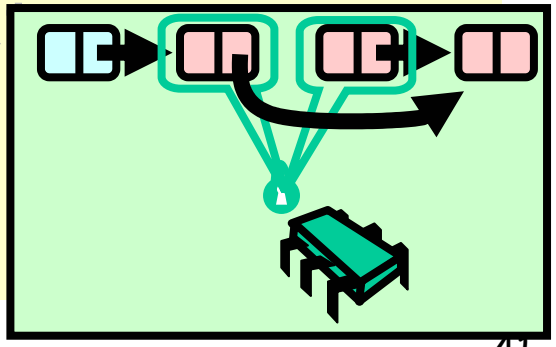
If entry found, remove it



Remove: searching

```
while (currEntry.key <= key) {
  if (object == currEntry.object) {
   predEntry.next = currEntry.next;
   return true;
  }
  prevEntry.unlock();
  prevEntry = currEntry;
  currEntry = currEntry.nex
  currEntry.lock();
 }
```

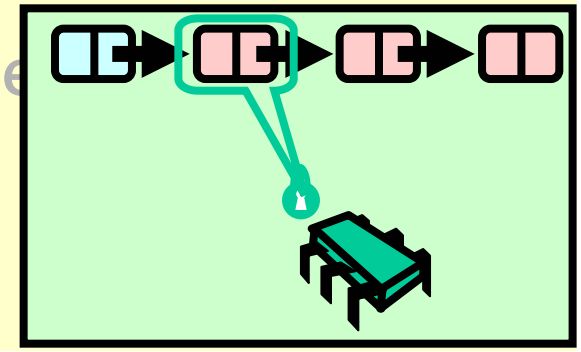
If entry found, remove it



Remove: searching

Unlock predecessor

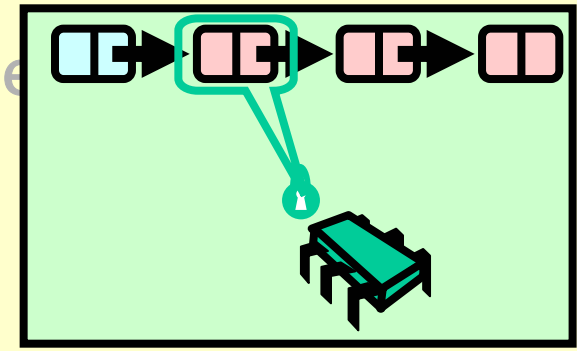
```
while (currEntry.key <= key) {
  if (object == currEntry.object) {
   prevEntry.next = currEntry.next;
   return true;
  }
  predEntry.unlock();
  prevEntry = currEntry;
  currEntry = currEntry.ne
  currEntry.lock();
 }
 return false;
```



Remove: searching

Only one entry locked!

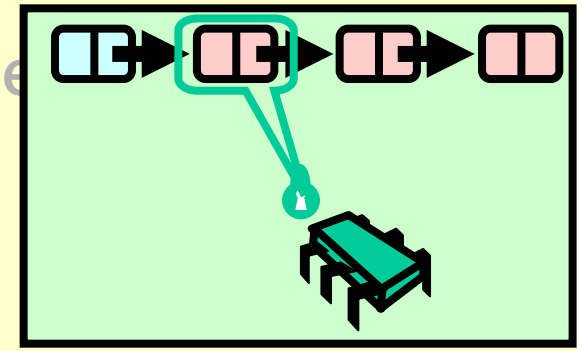
```
while (currEntry.key <= key) {
  if (object == currEntry.object) {
   prevEntry.next = currEntry.next;
   return true;
  }
  prevEntry.unlock();
  prevEntry = currEntry;
  currEntry = currEntry.ne
  currEntry.lock();
 }
 return false;
```



Remove: searching

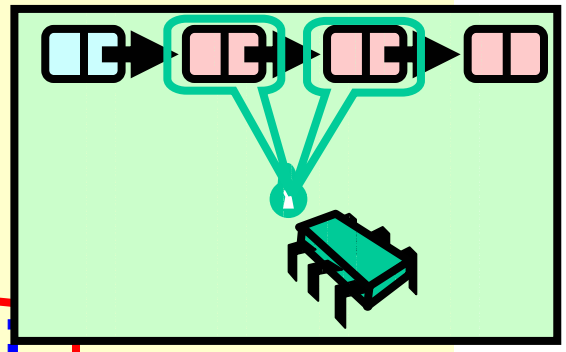
demote current

```
      y) {
    if (object == currEntry.object) {
      prevEntry.next = currEntry.next;
      return true;
    }
    prevEntry.unlock();
    predEntry = currEntry;
    currEntry = currEntry.ne
    currEntry.lock();
  }
  return false;
```



Remove: searching

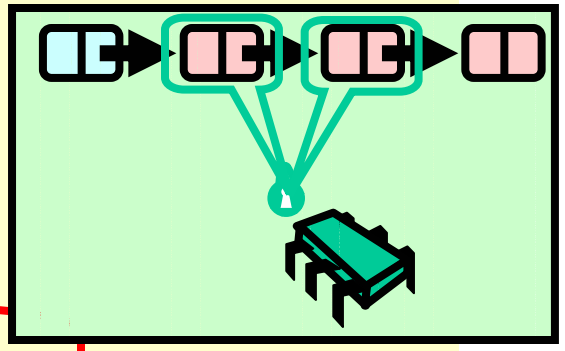
```
while (currEntry.key <= key) {
    Find and lock new current
    prevEntry.next = currEntry.next;
    return true;
  }
  prevEntry.unlock();
  prevEntry = currEntry;
  currEntry = currEntry.next;
  currEntry.lock();
}
return false;
```



Remove: searching

Lock invariant restored

```
 if (object == currEntry.object) {
  prevEntry.next = currEntry.next;
  return true;
 }
 prevEntry.unlock();
 prevEntry = currEntry;
 currEntry = currEntry.next;
 currEntry.lock();
}
return false;
```



Remove: searching

```
while (currEntry.key <= key) {
  if (object == currEntry.object) {
   prevEntry.next = currEntry.next;
   return true;
  }
  prevEntry.unlock();
  prevEntry = currEntry;
  currEntry = currEntry.next;
  currEntry.lock();
 }
 return false;
```

Otherwise, error 404, dude!

Why does this work?

- To remove entry e
  - Must lock e
  - Must lock e's predecessor
- Therefore, if you lock an entry
  - It can't be removed
  - And neither can its successor

Lock and Load

- To move to successor entry for e
  - Lock e
  - Lock e.next
  - Unlock e
- While traversing
  - e cannot be removed
  - e.next cannot be removed

Until next entry idenified and locked

Don't release e

Adding Entries

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

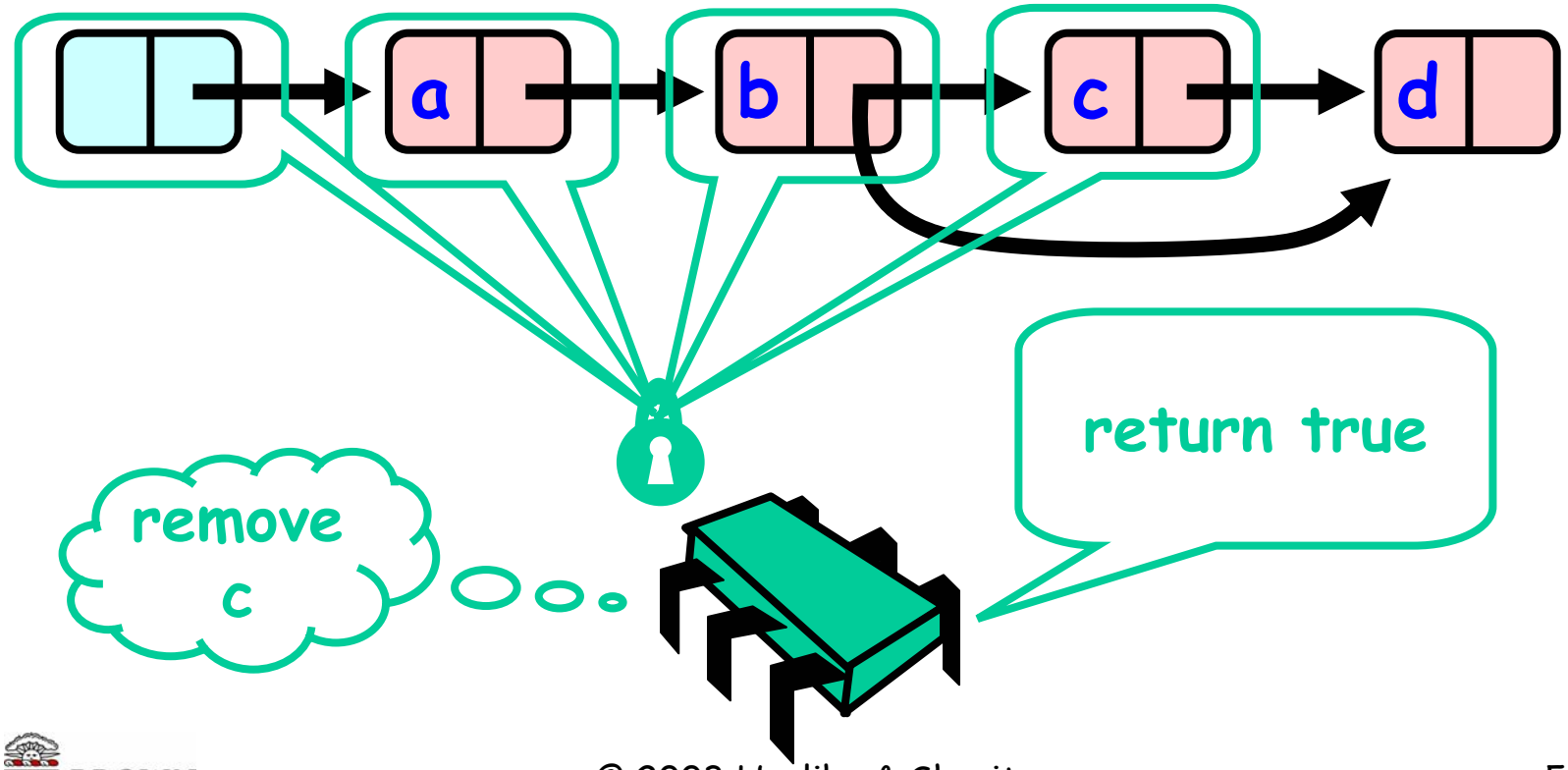
Drawbacks

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

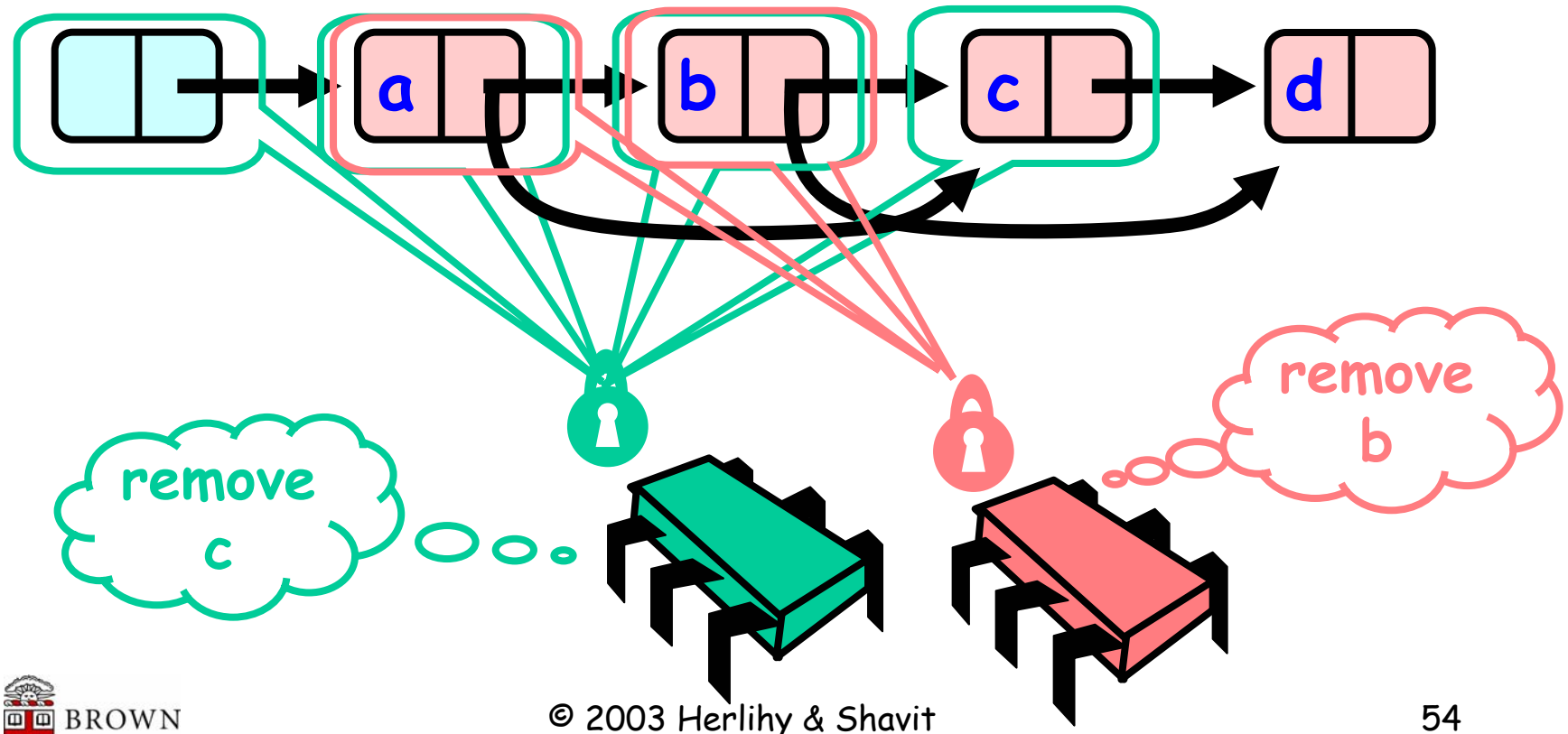
Optimistic Synchronization

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

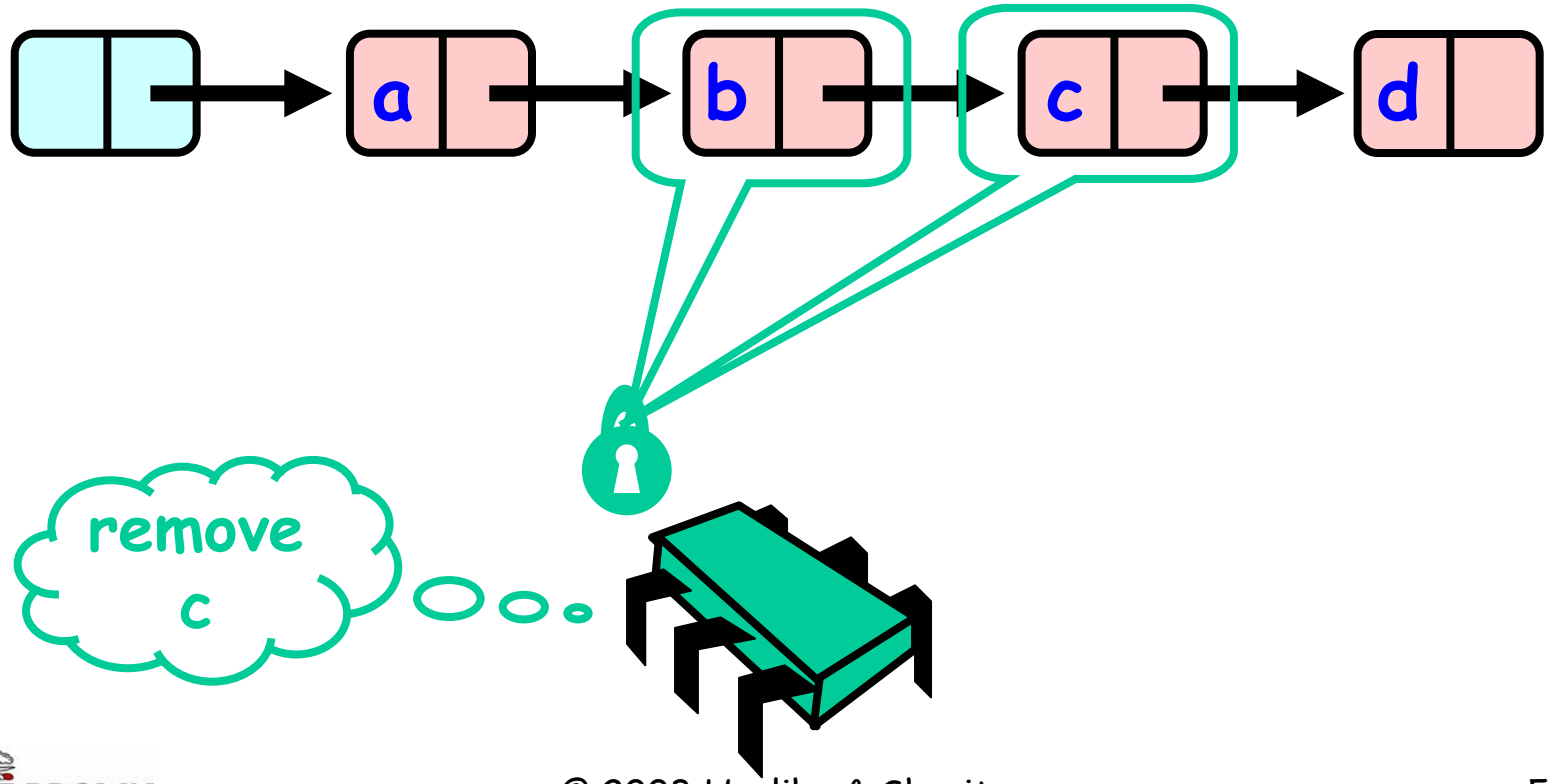
Removing an Entry



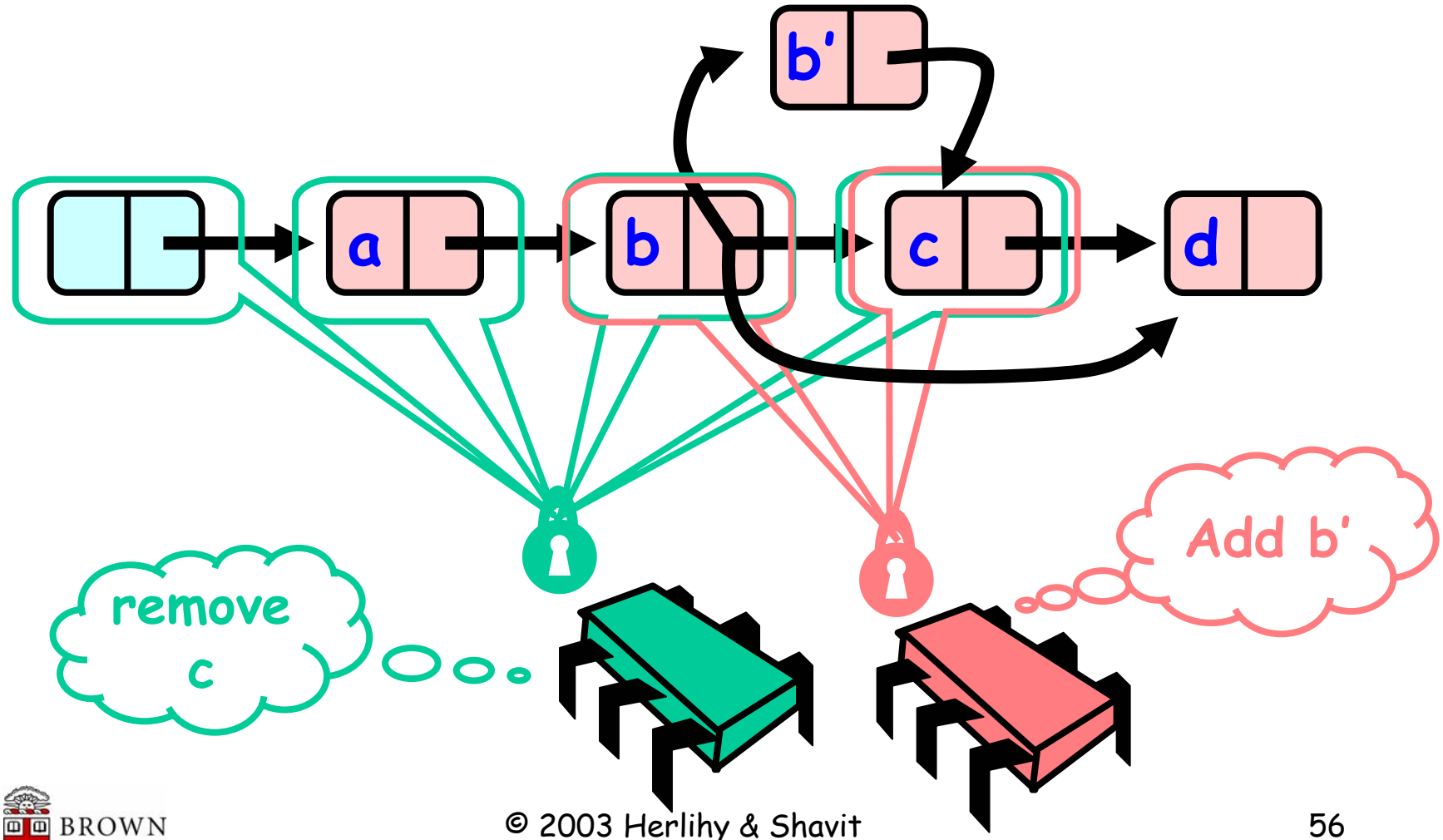
What Can Go Wrong?



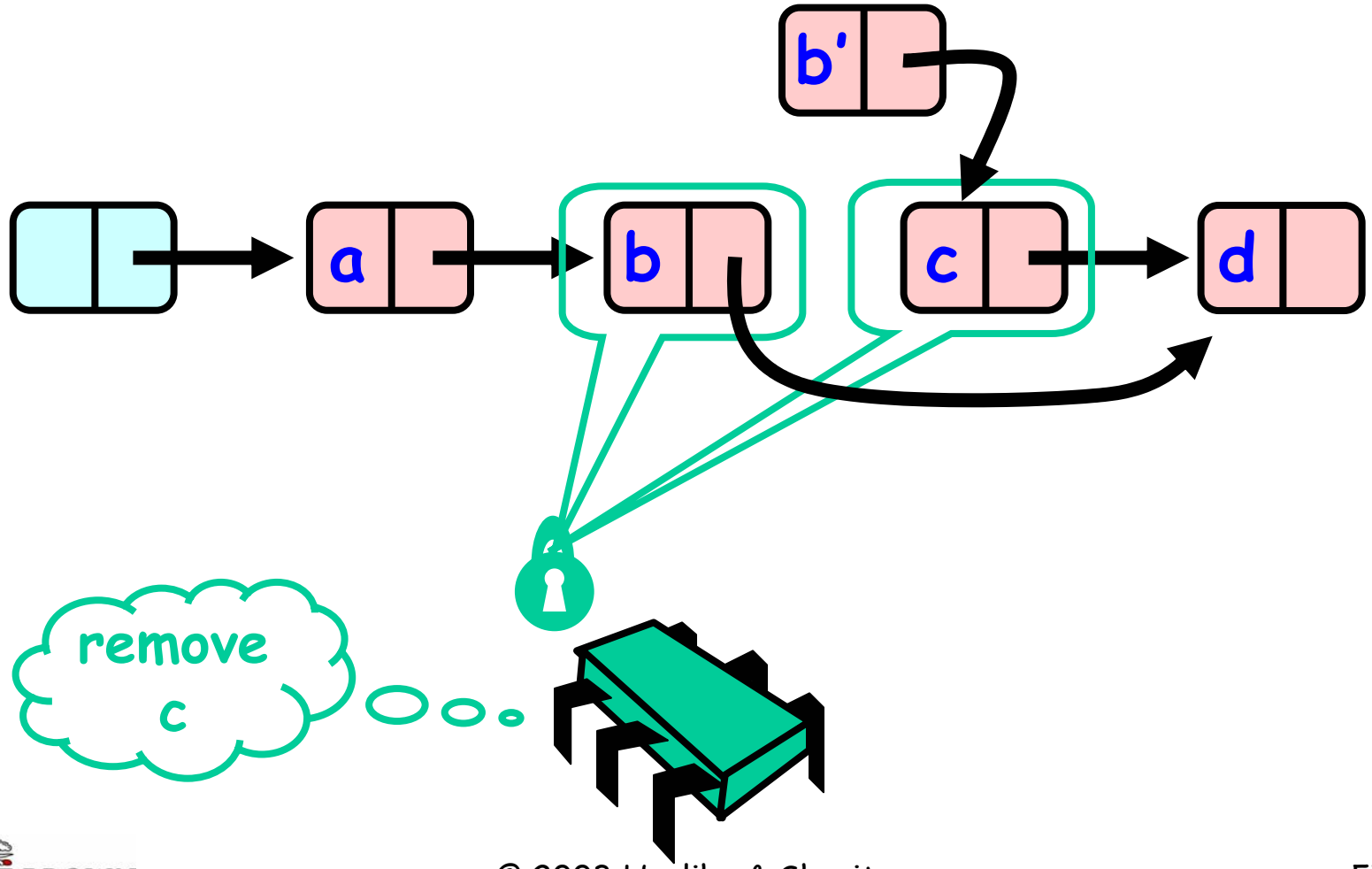
Check that Entry is Still Accessible



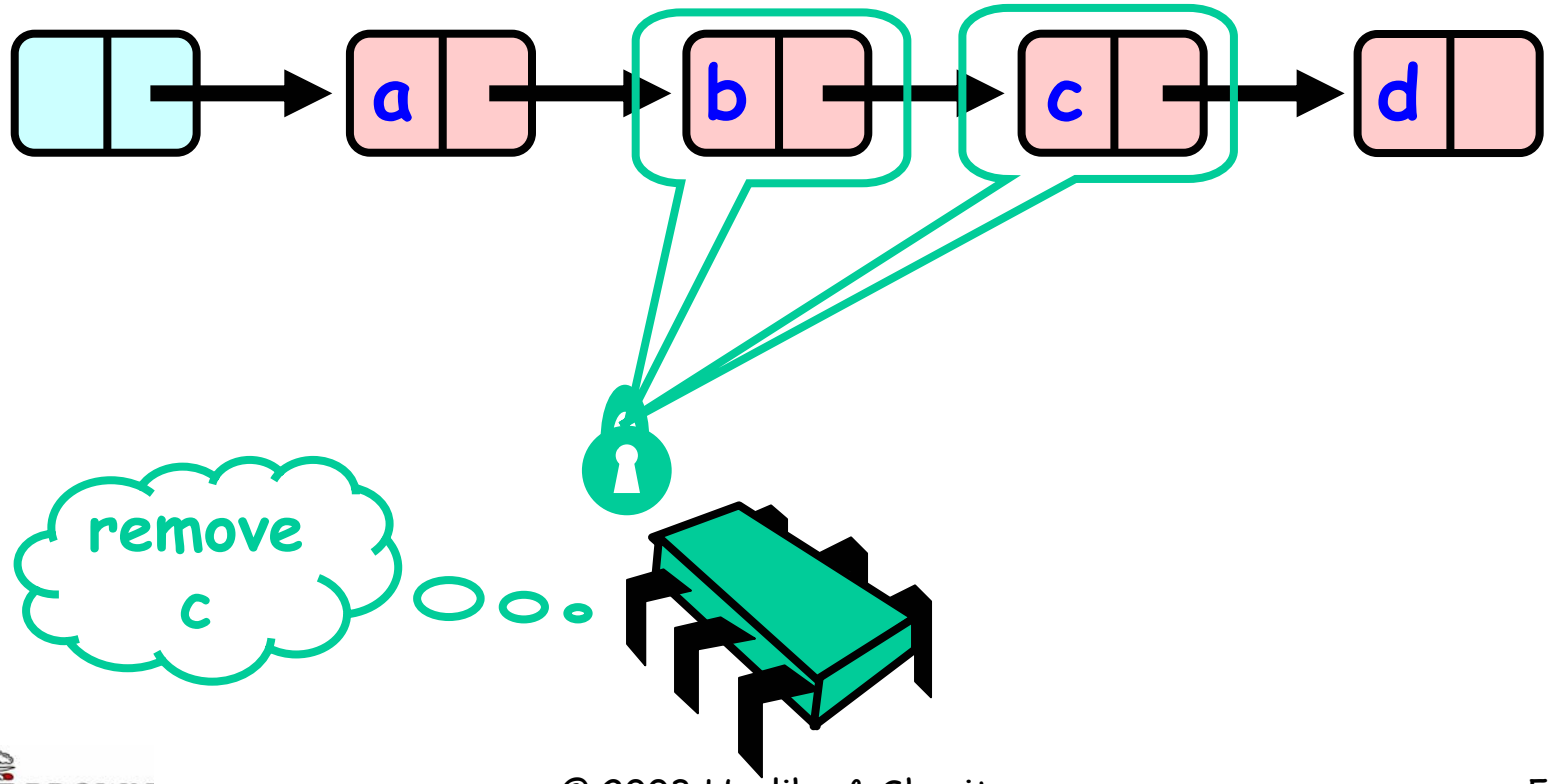
What Can Go Wrong?



What Can Go Wrong?



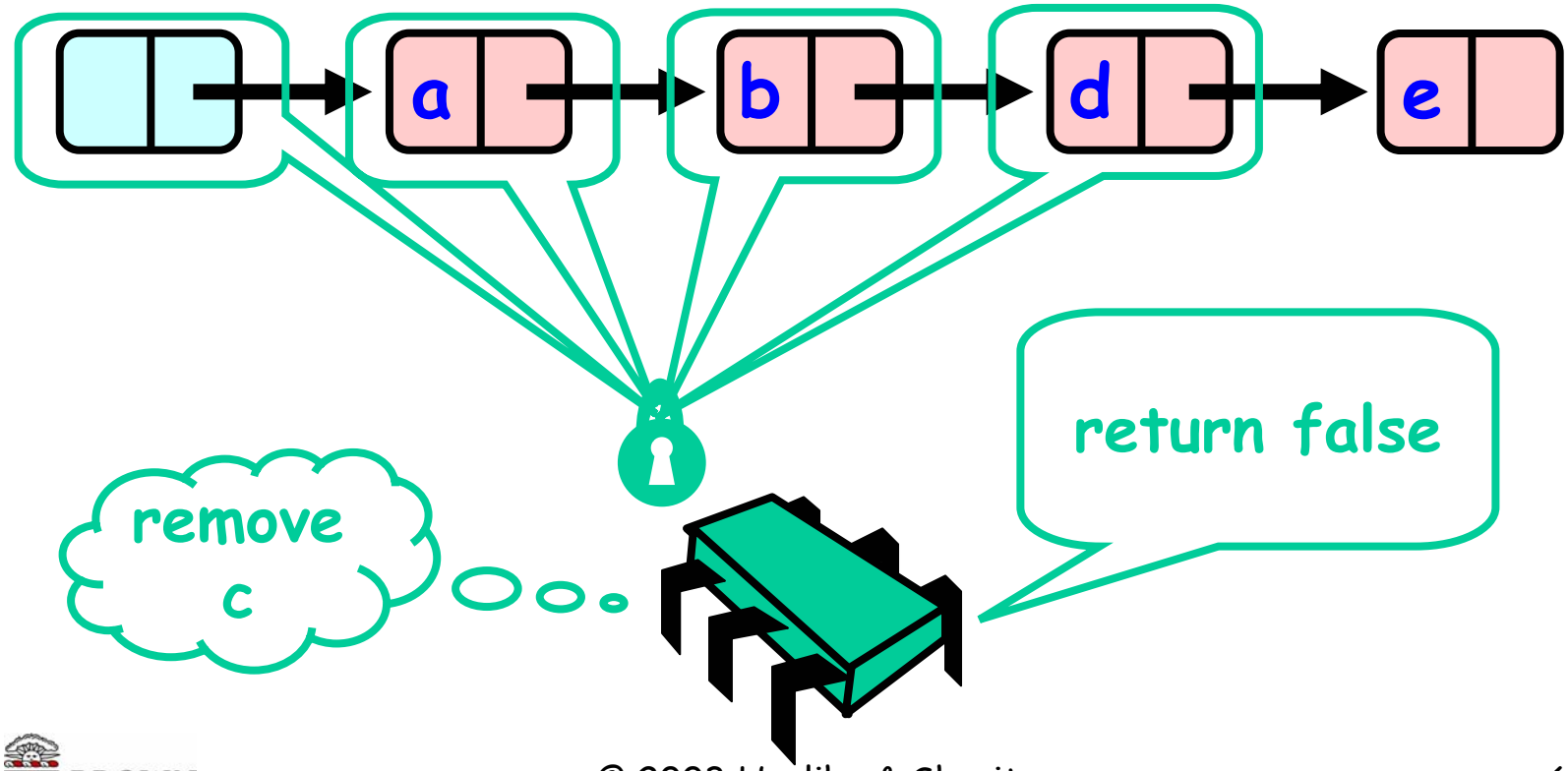
Check that Entries Still Adjacent



Correctness

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

Removing an Absent Entry



Correctness

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

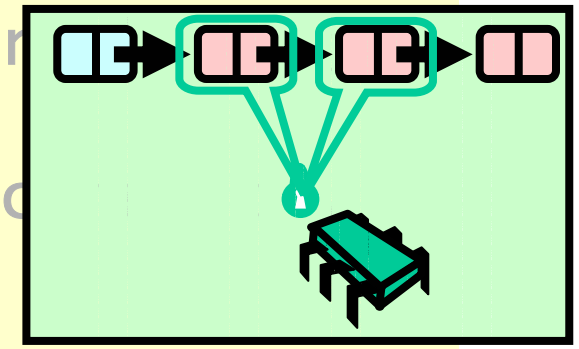
Validation

```
private boolean
 validate(Entry predEntry,
          Entry currEntry) {
 Entry entry = head;
 while (entry.key <= predEntry.key) {
  if (entry == predEntry)
   return predEntry.next == currEntry;
  entry = entry.next;
 }
 return false;
}
```

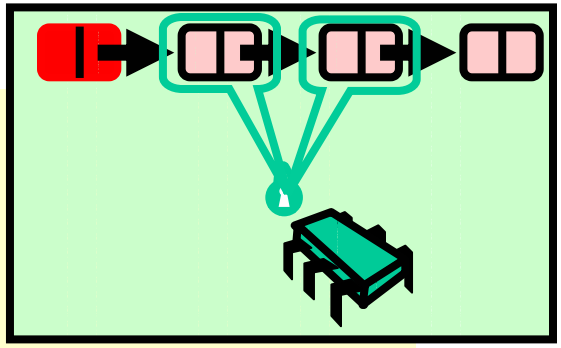
Validation

```
private boolean
 validate(Entry predEntry,
          Entry currEntry) {
  Entry entry = head;
  while (entry.key <= predEntr
    if (entry == predEntry)
      return predEntry.next == c
    entry = entry.next;
  }
  r
}
```

Predecessor & current entries



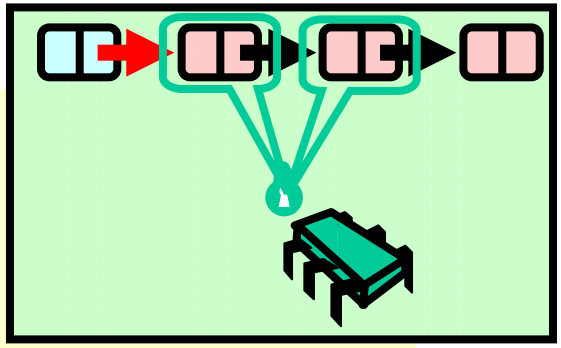
Validation



```
private boolean
 validate(Entry predEntry,
          Entry currEntry) {
 Entry entry = head;
 while (entry.key <= predEntry.key) {
  if (entry == predEntry)
   return predEntry.next == currEntry;
  entry = entry.next;
 }
 return false;
}
```

Start at the beginning

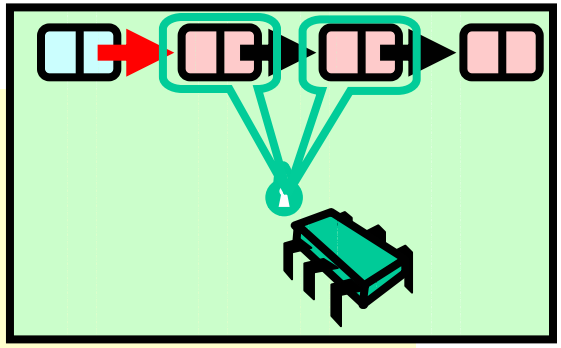
Validation



```
private boolean
 validate(Entry predEntry,
          Entry currEntry) {
 Entry entry = head;
 while (entry.key <= predEntry.key) {
  if (entry == predEntry)
    return predEntry.next == currEntry;
  entry = entry.next;
 }
 return false;
}
```

Search range of keys

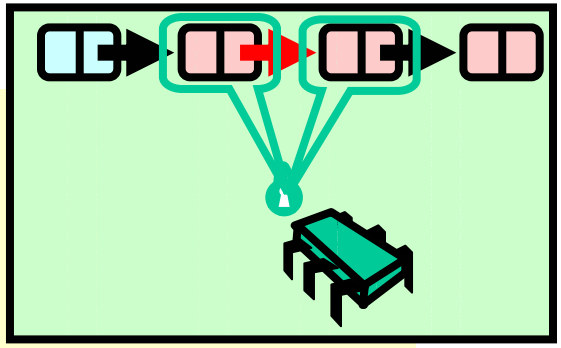
Validation



```
private boolean
 validate(Entry predEntry,
          Entry currEntry) {
 Entry entry = head;
 while (entry.key <= predEntry.key) {
  if (entry == predEntry)
    return predEntry.next == currEntry;
    entry = entry.next;
  }
 return false;
}
```

Predecessor reachable

Validation



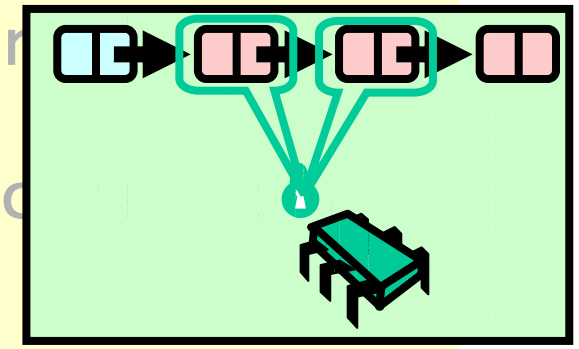
```
private boolean
 validate(Entry predEntry,
          Entry currEntry) {
 Entry entry = head;
 while (entry.key <= predEntry.key) {
  if (entry == predEntry)
   return predEntry.next == currEntry;
  entry = entry.next;
 }
 return false;
}
```

Is current entry next?

Validation

```
private boolean
 validate(Entry predEntry,
          Entry currEntry) {
 Entry entry = head;
 while (entry.key <= predEntr
  if (entry == predEntry)
   return predEntry.next == c
  entry = entry.next;
 }
 return false;
}
```

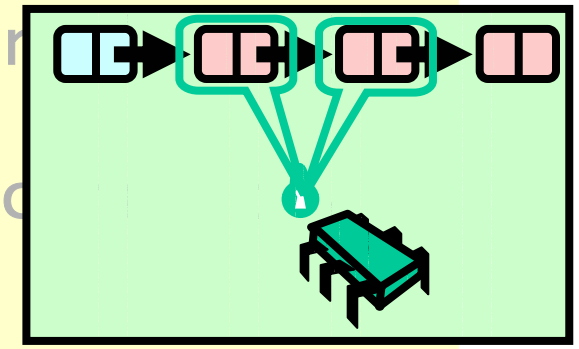
Otherwise move on



Validation

```
private boolean
 validate(Entry predEntry,
          Entry currEntry) {
 Entry entry = head;
 while (entry.key <= predEntr
  if (entry == predEntry)
   return predEntry.next == c
 entry = entry.next;
 }
 return false;
}
```

Predecessor not reachable

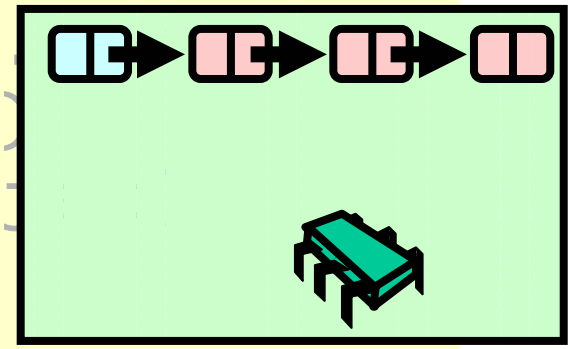


Remove: searching

```
public boolean remove(Object object) {
 int key = object.hashCode();
 retry: while (true) {
 Entry predEntry = this.head;
 Entry currEntry = predEntry.next;
 while (currEntry.key <= key) {
  if (object == currEntry.object)
   break;
  predEntry = currEntry;
  currEntry = currEntry.next;
 } …
```

Remove: searching

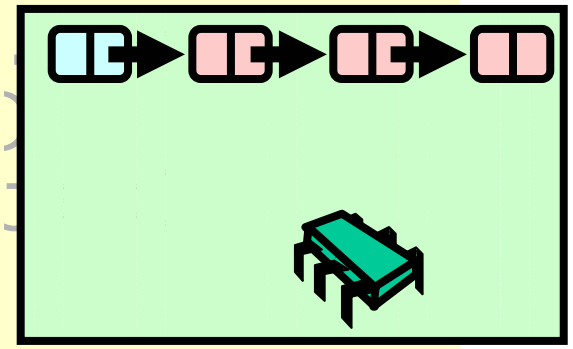
```
public boolean remove(Object object) {
 int key = object.hashCode();
  retry: while (true) {
  Entry prevEntry = this.head;
  Entry currEntry = prevEntry.
  while (currEntry.key <= key)
   if (object == currEntry.obj
    break;
   prevEntry = currEntry;
   currEntry = currEntry.next;
  } …
```



Search key

Remove: searching

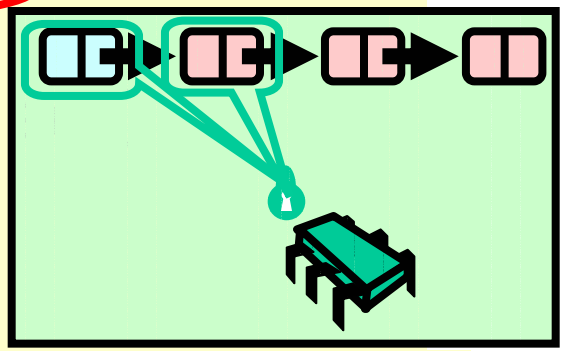
```
public boolean remove(Object object) {
 int key = object.hashCode();
 retry: while (true) {
 Entry prevEntry = this.head;
 Entry currEntry = prevEntry.
 while (currEntry.key <= key)
  if (object == currEntry.ob
   break;
  prevEntry = currEntry;
  currEntry = currEntry.next;
 } …
```



Retry on synchronization conflict

Remove: searching

```
public boolean remove(Object object) {
 int key = object.hashCode();
 retry: while (true) {
 Entry predEntry = this.head;
 Entry currEntry = predEntry.next;
 while (currEntry.key <= key)
  if (object == currEntry.obj
   break;
  prevEntry = currEntry;
  currEntry = currEntry.next;
```

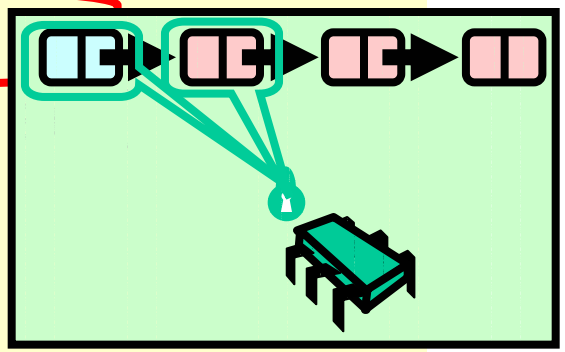


Examine predecessor and current entries

Remove: searching

```
public boolean remove(Object object) {
 int key = object.hashCode();
 retry: while (true) {
 Entry prevEntry = this.head;
 Entry currEntry = prevEntry.next;
 while (currEntry.key <= key)
  if (object == currEntry.obj
   break;
  prevEntry = currEntry;
                 try.next;
 } …
```

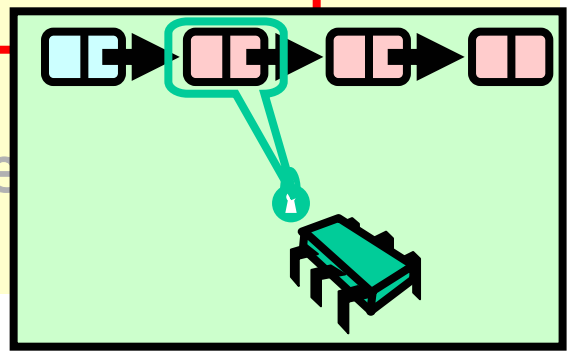
Search by key



Remove: searching

```
public boolean remove(Object object) {
 int key = object.hashCode();
 retry: while (true) {
 Entry prevEntry = this.head;
 Entry currEntry = prevEntry.next;
 while (currEntry.key <= key) {
  if (object == currEntry.object)
   break;
  prevEntry = currEntry;
  currEntry = currEntry.ne
```

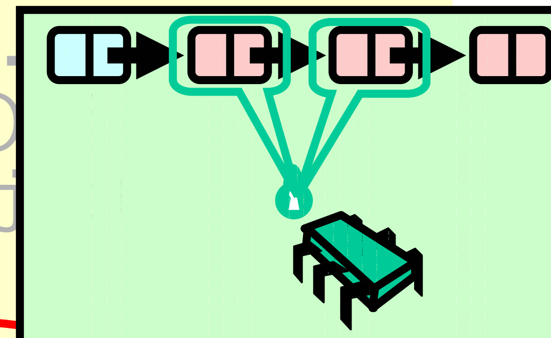
Stop if we find object



Remove: searching

```
publi          emove(Object object) {
 int             t.hashCode();
 retry: while (true) {
 Entry prevEntry = this.head;
 Entry currEntry = prevEntry.
 while (currEntry.key <= key)
  if (object == currEntry.obj
   break;
  predEntry = currEntry;
  currEntry = currEntry.next;
 } …
```

Move along



On Exit from Loop

- If object is present
  - currEntry holds object
  - predEntry just before currEntry
- If object is absent
  - currEntry has first higher key
  - predEntry just before currEntry
- Assuming no synchronization problems

Remove Method

```
try {
  predEntry.lock(); currEntry.lock();
  if (validate(predEntry,currEntry) {
   if (currEntry.object == object) {
    predEntry.next = currEntry.next;
    return true;
   } else {
    return false;
   }}} finally {
     predEntry.unlock();
     currEntry.unlock();
   }}}
```

Remove Method

```
try {
  predEntry.lock(); currEntry.lock();
  if (validate(predEntry,currEntry) {
   if (currEntry.object == object) {
    predEntry.next = currEntry.next;
    return true;
   } else {
    return false;
  }}} finally {
    predEntry.unlock();
    currEntry.unlock();
  }}}
```

Always unlock

Remove Method

```
try {
  predEntry.lock(); currEntry.lock();
  if (validate(predEntry,currEntry) {
   if (currEntry.object == object) {
    predEntry.next = currEntry.next;
    return true;
   } else {
    return false;
   }}} finally {
     predEntry.unlock();
     currEntry.unlock();
   }}}
```

Lock both entries

Remove Method

```
try {
  predEntry.lock(); currEntry.lock();
  if (validate(predEntry,currEntry) {
    if (currEntry.object == object) {
     predEntry.next = currEntry.next;
     return true;
    } else {
     return false;
    }}} finally {
      predEntry.unlock();
      currEntry.unlock();
    }}}
```

Check for synchronization conflicts

Remove Method

```
try {
  predEntry.lock(); currEntry.lock();
  if (validate(predEntry,currEntry) {
   if (currEntry.object == object) {
    predEntry.next = currEntry.next;
    return true;
   } else {
    return false;
   }}} finally {
     predEntry.unlock();
     currEntry.unlock();
   }}}
```

Object found, remove entry

Remove Method

```
try {
  predEntry.lock(); currEntry.lock();
  if (validate(predEntry,currEntry) {
   if (currEntry.object == object) {
    predEntry.next = currEntry.next;
    return true;
   } else {
    return false;
   }}} finally {
     predEntry.unlock();
     currEntry.unlock();
   }}}
```

Object not found

So Far, So Good

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

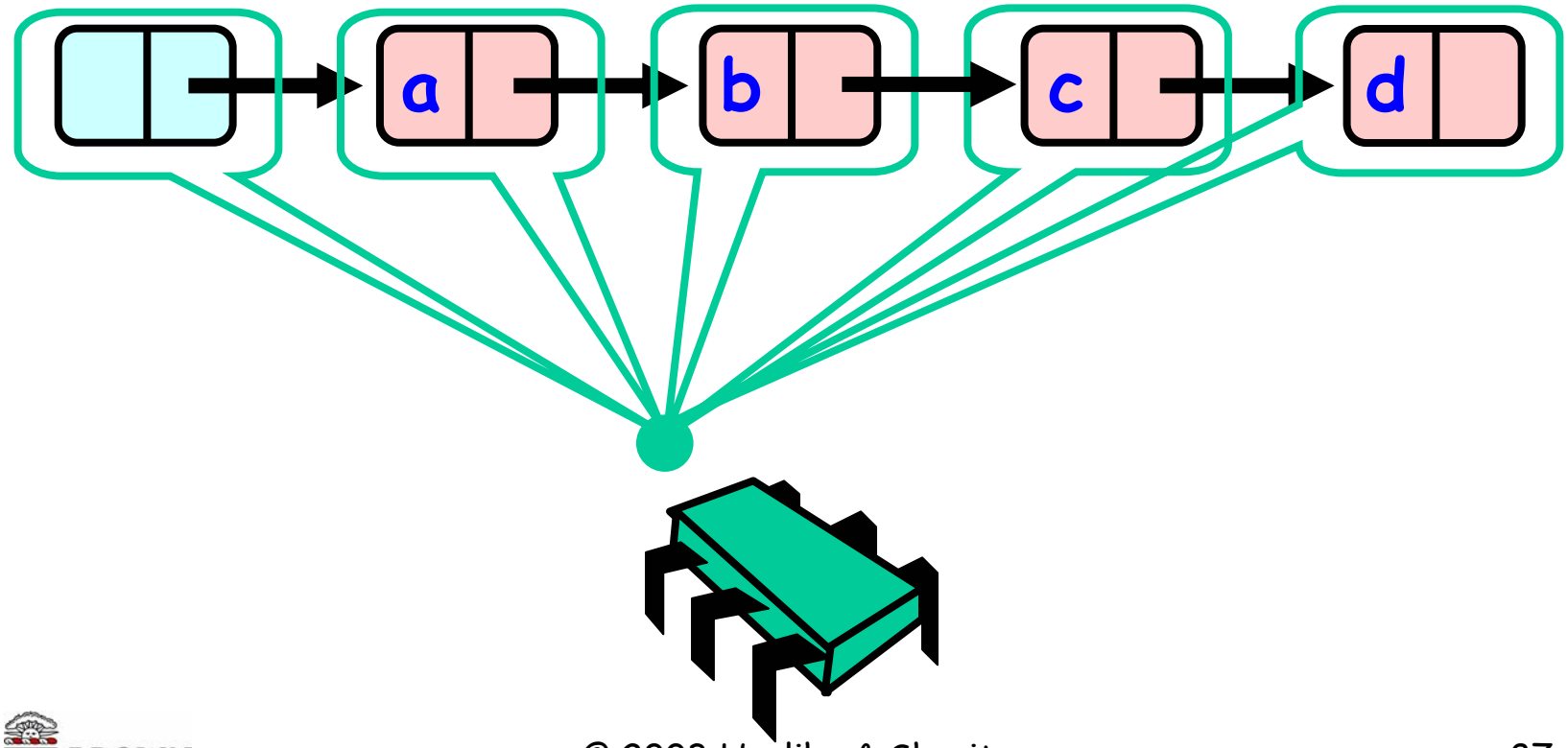
Marked Lists

- Remove Method
  - Scans list (as before)
  - Locks predecessor & current (as before)
  - Marks current entry as removed (new!)
  - Redirects predecessor's next (as before)

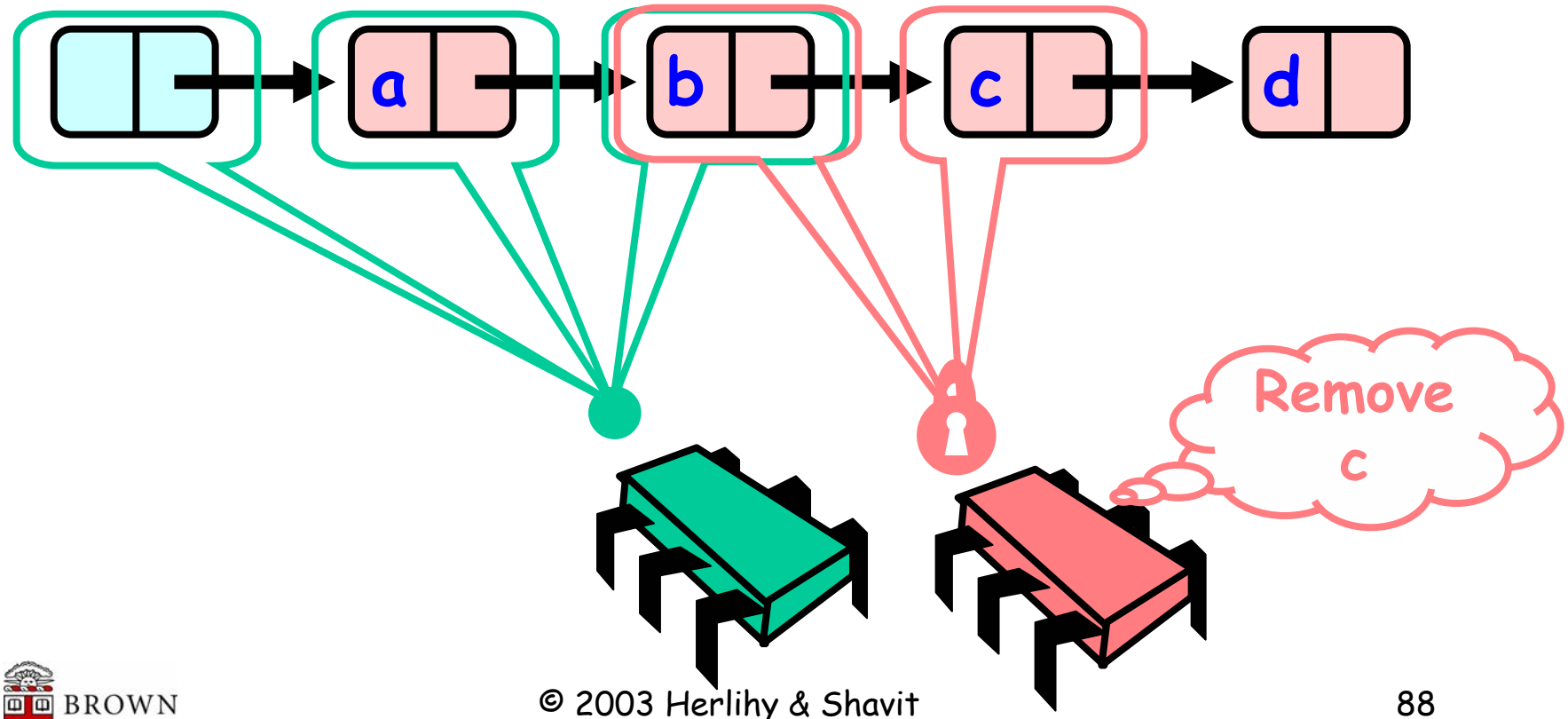
Marked Lists

- All Methods
  - Scan list
  - Do not scan past marked entry
  - Instead, start over from list head

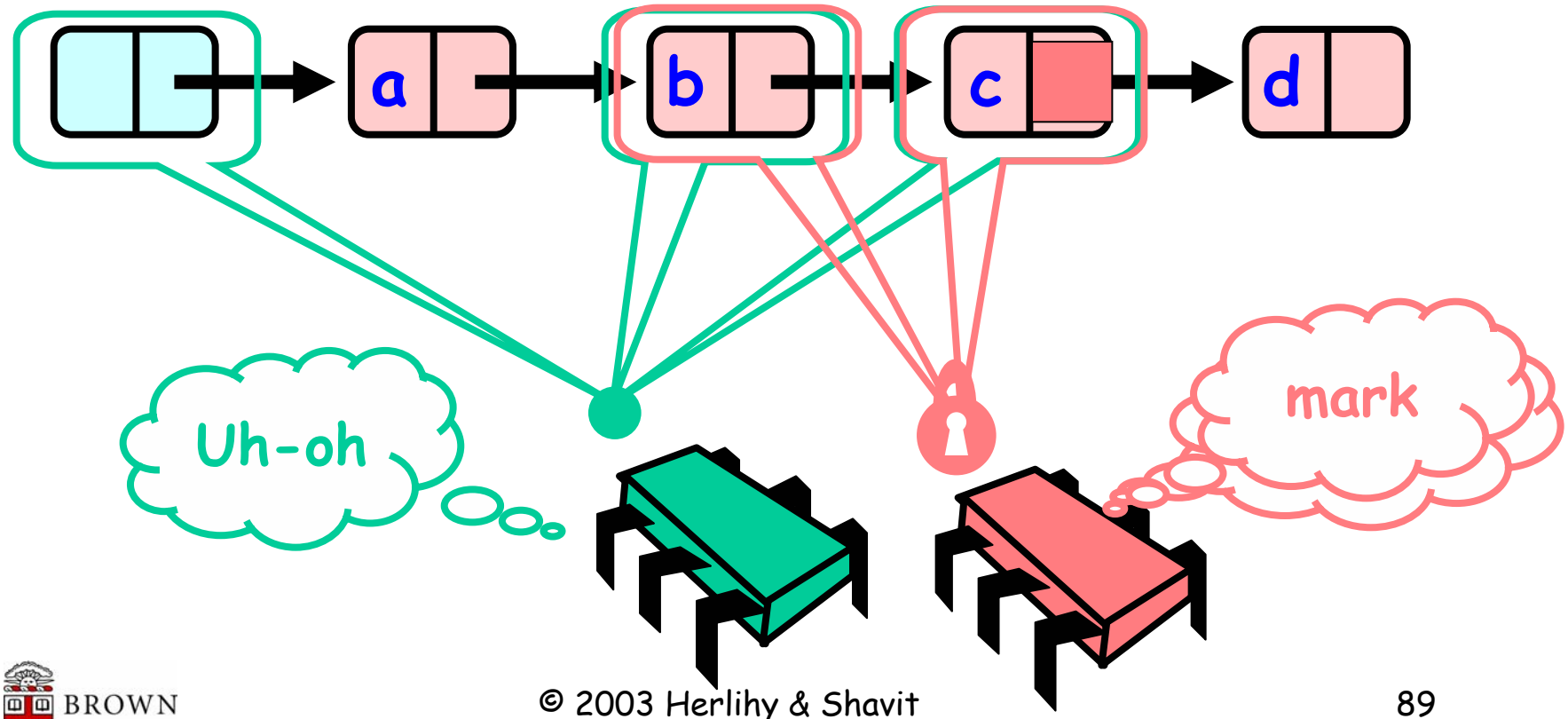
Business as Usual



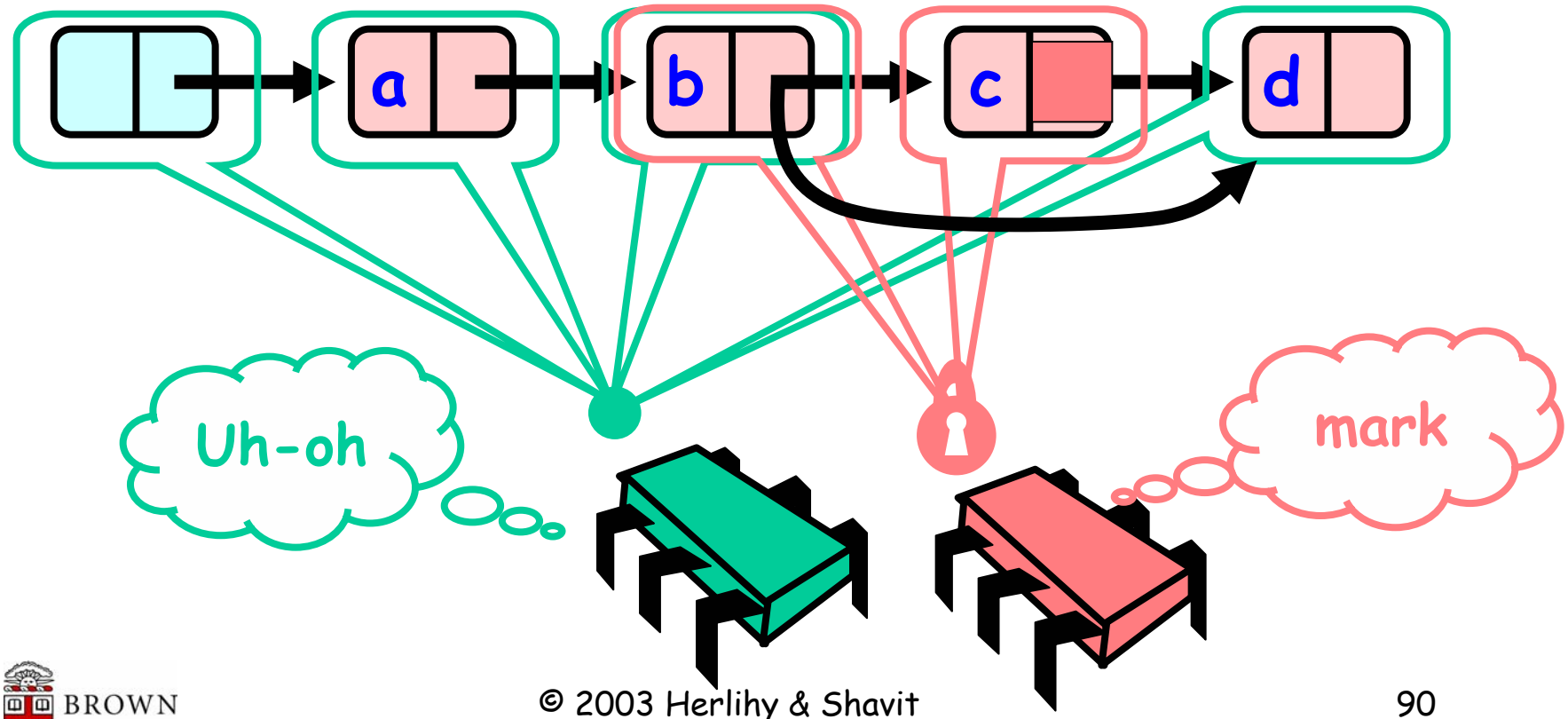
Interference



Interference

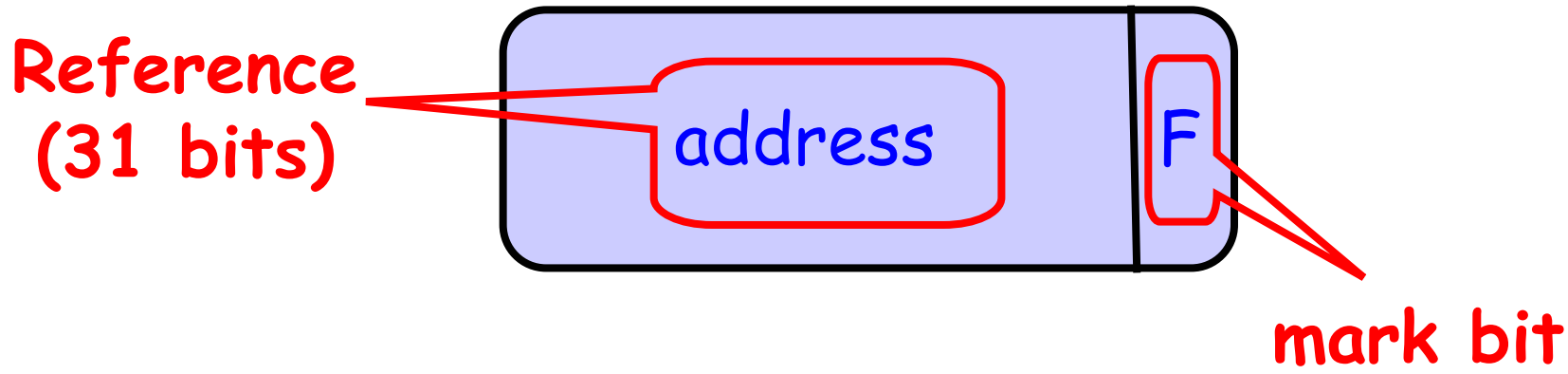


Interference



Marking a Node

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



Extracting Information

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

Extracting Information

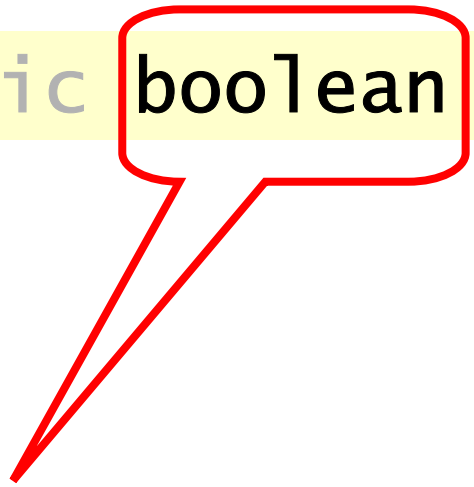
```
Public Object get(boolean[]);
```

Returns reference

Stores mark at index 0

Extracting Information

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



Value of mark

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.

Aside

- We will see that it is often useful to tag pointers with
  - Boolean values
  - Integer values
- Sometimes to mark, and sometimes to ensure pointers are unique

List Validate Method

```
private boolean
  validate(Entry predEntry,
           Entry currEntry) {
 return
  (!predEntry.next.isMarked()) &&
  (!currEntry.next.isMarked()) &&
  (predEntry.next.getReference()
    == currEntry);
  }
```

List Validate Method

```
private boolean
  validate(Entry predEntry,
           Entry currEntry) {
 return
  (!predEntry.next.isMarked()) &&
  (!currEntry.next.isMarked()) &&
  (predEntry.next.getReference()
    == currEntry);
  }
```

Predecessor not removed

List Validate Method

```
private boolean
  validate(Entry predEntry,
           Entry currEntry) {
 return
  (!predEntry.next.isMarked()) &&
  (!currEntry.next.isMarked()) &&
  (predEntry.next.getReference()
    == currEntry);
  }
```

Current not removed

List Validate Method

```
private boolean
  validate(Entry predEntry,
           Entry currEntry) {
 return
  (!predEntry.next.isMarked()) &&
  (!currEntry.next.isMarked()) &&
  (predEntry.next.getReference()
    == currEntry);
  }
```

Next field unchanged

Remove: searching

```
public boolean remove(Object object) {
 …
 while (currEntry.key <= key) {
     Entry nextEntry =
       (Entry)currEntry.next.get(mark);
     if (mark[0])
       continue retry;
  if (object == currEntry.object)
   break;
  predEntry = currEntry;
  currEntry = currEntry.next;
 } …
```

Remove: searching

```
public boolean remove(Object object) {
 …
 while (currEntry.key <= key) {
    Entry nextEntry =
      (Entry)currEntry.next.get(mark);
     if (mark[0])
       continue retry;
  if (object == currEntry.object)
   break;
  predEntry = c
  currEntry = currEntry.next;
 } …
```

Atomically read reference & mark

Remove: searching

Panic if entry removed

```
…
while (currEntry.key <= key) {
    Entry nextEntry =
      (Entry)currEntry.next.get(mark);
    if (mark[0])
      continue retry;
  if (object == currEntry.object)
   break;
  predEntry = currEntry;
  currEntry = currEntry.next;
} …
```

Evaluation

- Good:
  - Contains method doesn't need to lock
  - Uncontended calls don't re-traverse
- Bad
  - Contended calls do re-traverse
  - Traffic jam if one thread delays

Traffic Jam

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And "eats the big muffin" (stops running)
    - Cache miss, page fault, descheduled …
    - Software error, …
  - Everyone else using that lock is stuck!

Lock-Free Data Structures

- No matter what …
  - Some thread will complete method call
  - Even if others halt at malicious times
- Implies that
  - You can't use locks (why?)
  - Um, that's why they call it lock-free

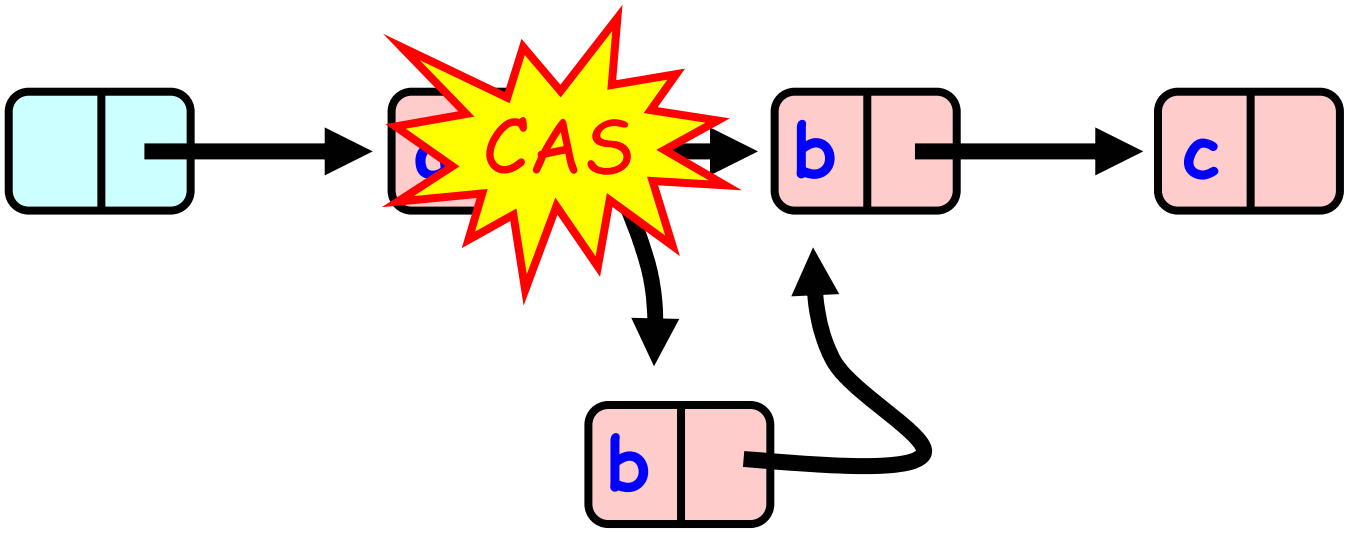
Lock-Free ≠ Wait-Free

- Wait-free synchronization
  - Every method call eventually finishes
  - What everyone really wants
- Lock-free synchronization
  - Some method call eventually finishes
  - What we are usually willing to pay for
    - Starvation rare in practice …

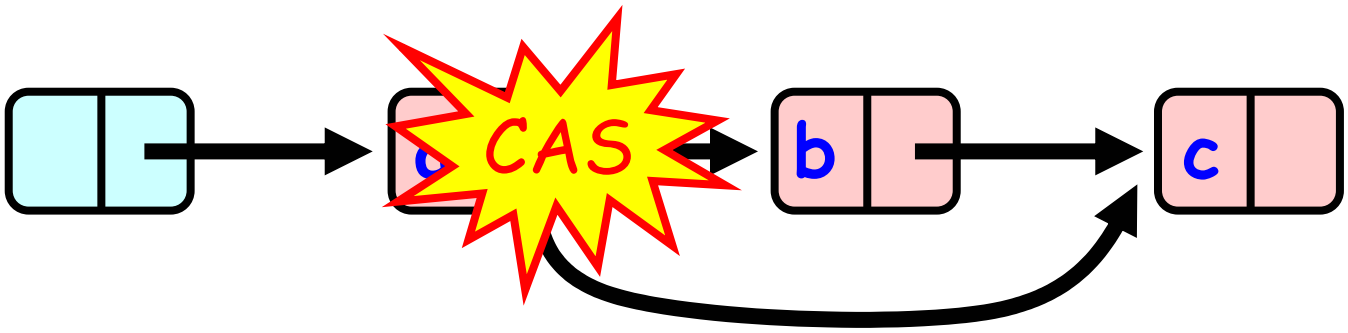
Lock-Free Lists

- Next logical step
- Eliminate locking entirely
- Use only compareAndSet()
- Invented by Maged Michael, 2003

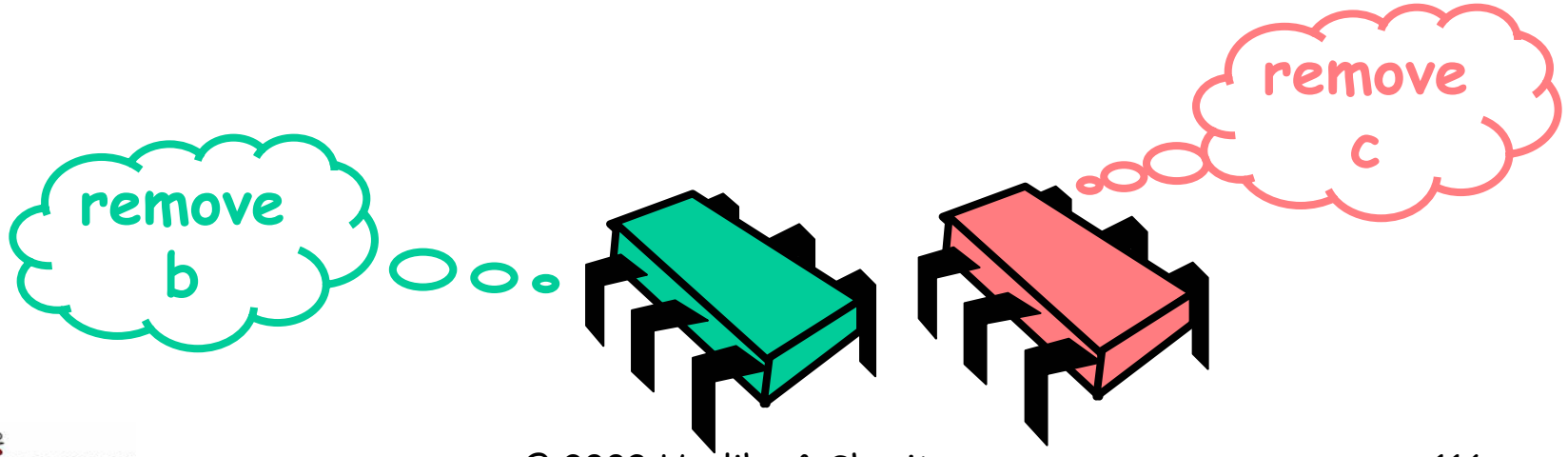
Adding an Entry



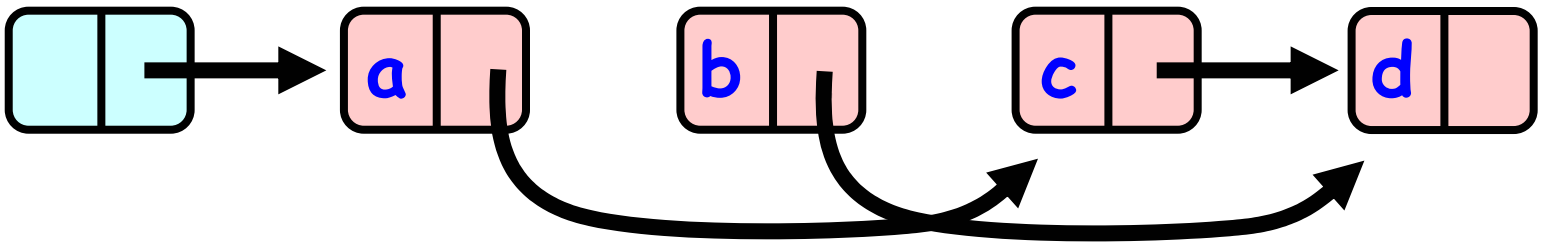
Removing an Entry

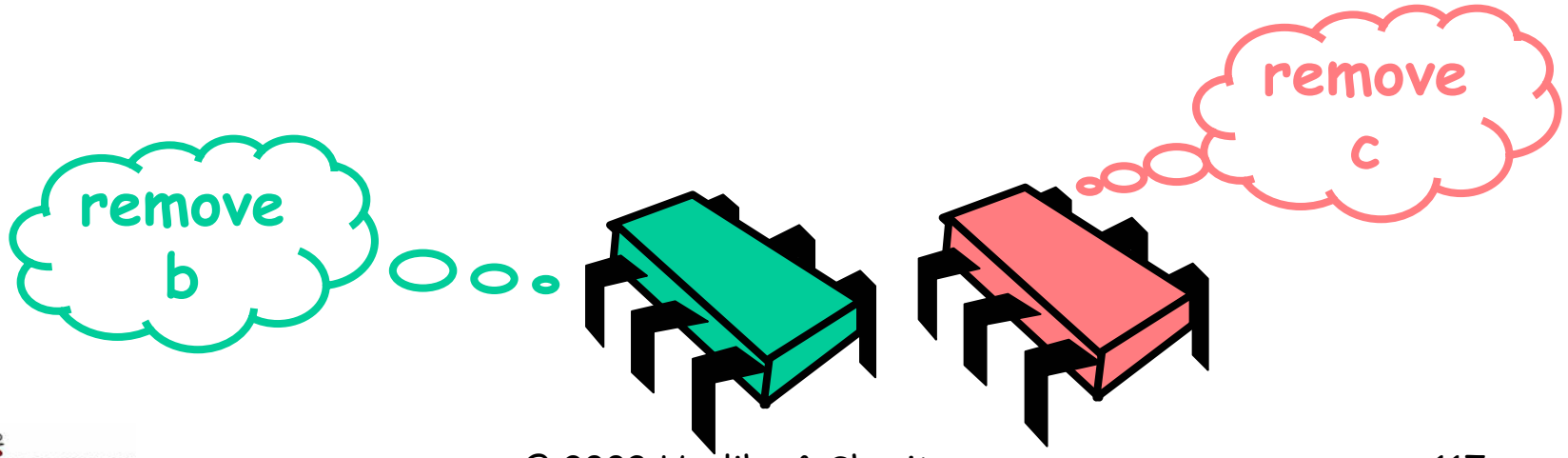


Removing an Entry



Look Familiar?





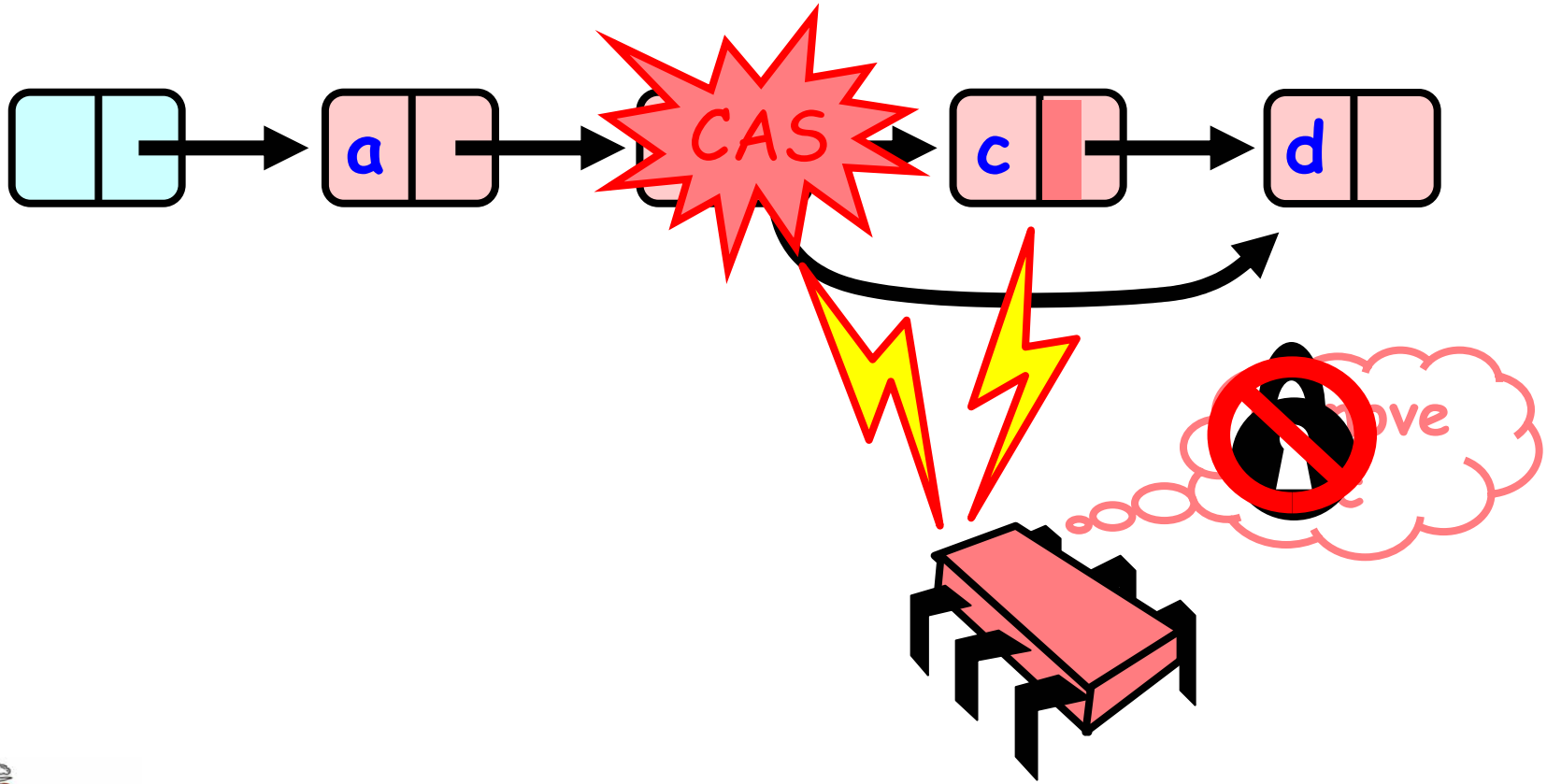
Problem

- Method updates entry's next field
- After entry has been removed

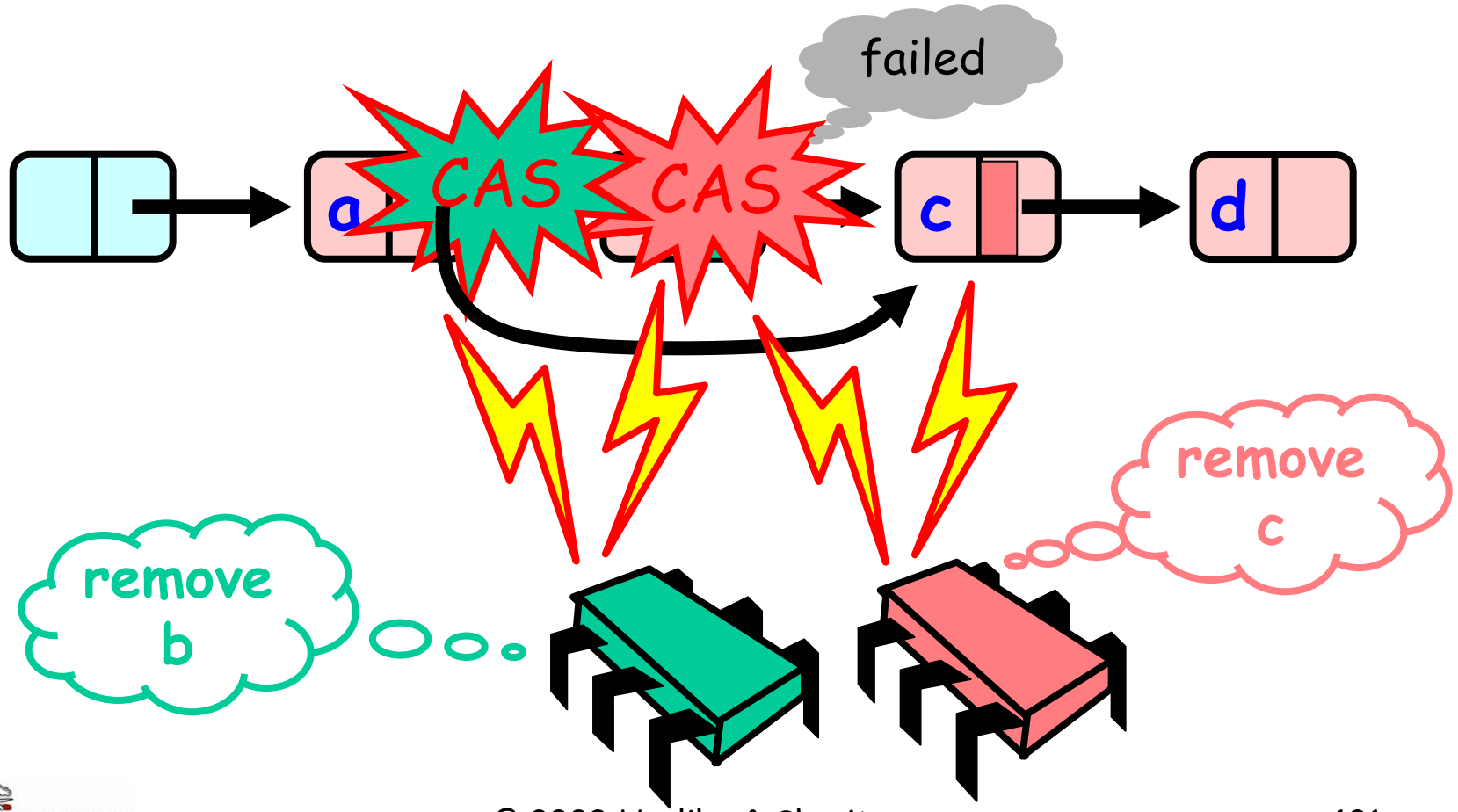
Solution

- Use AtomicMarkableReference
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor's pointer
- CAS
  - Fails if mark bit set (entry removed)

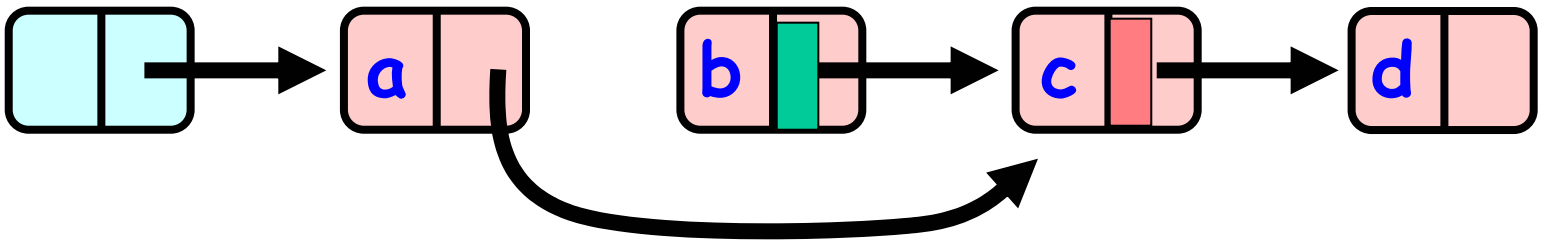
Removing an Entry

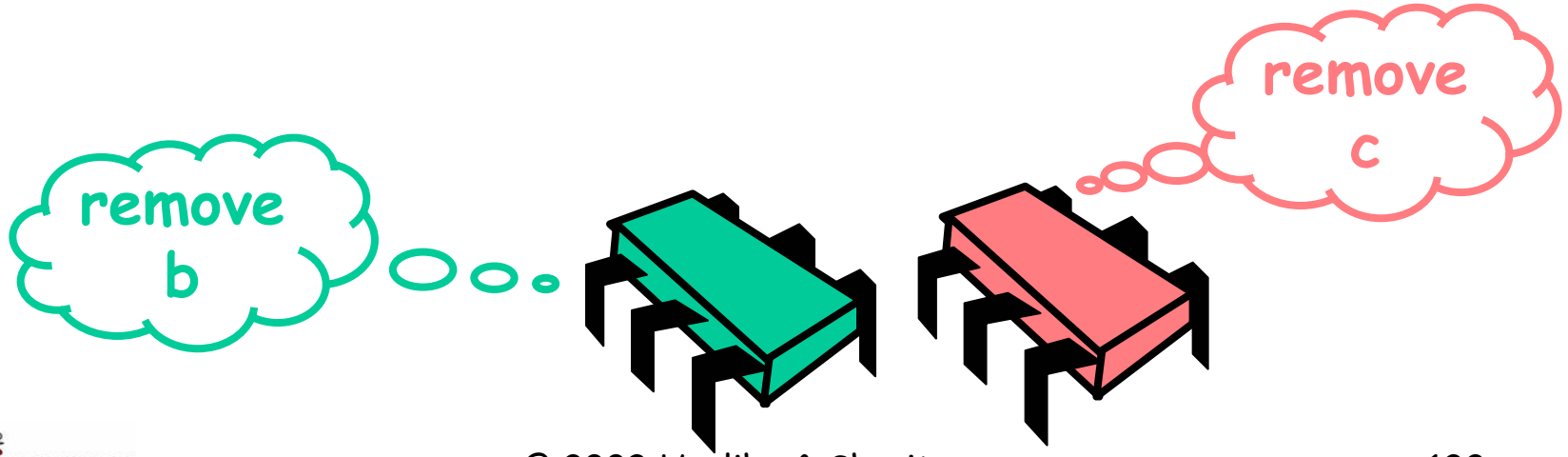


Removing an Entry

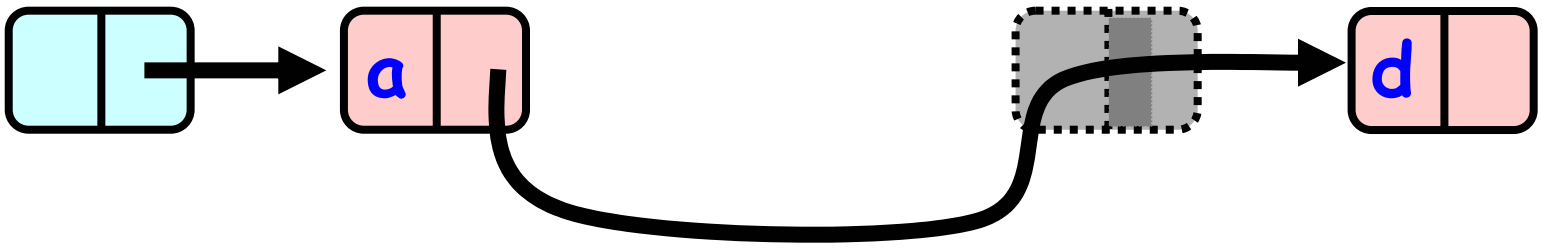


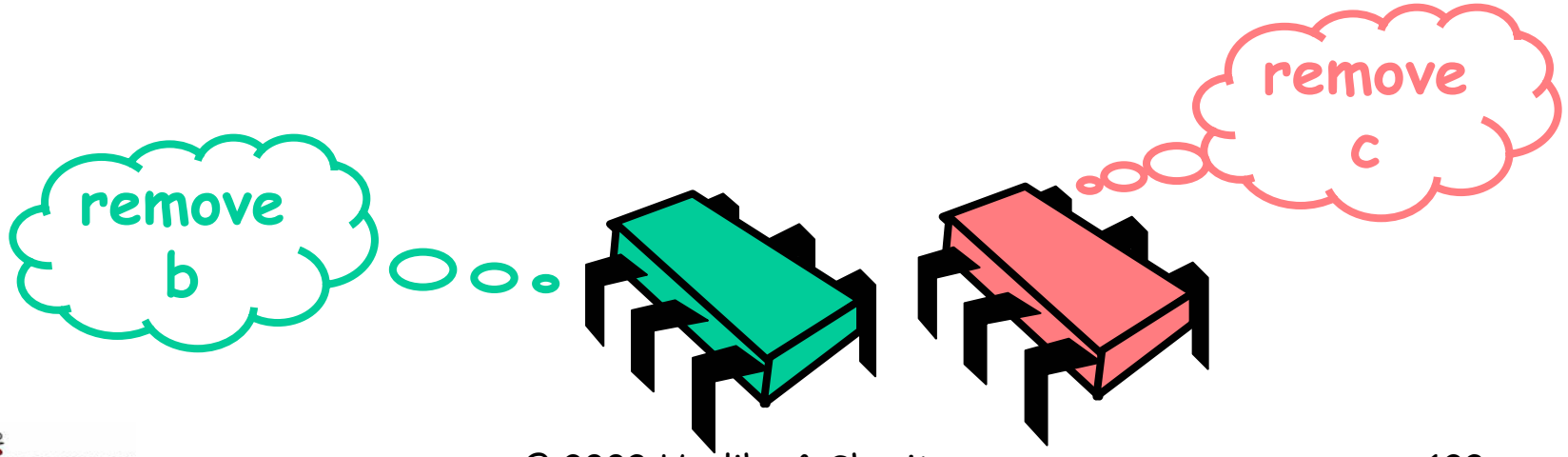
Removing an Entry





Removing an Entry

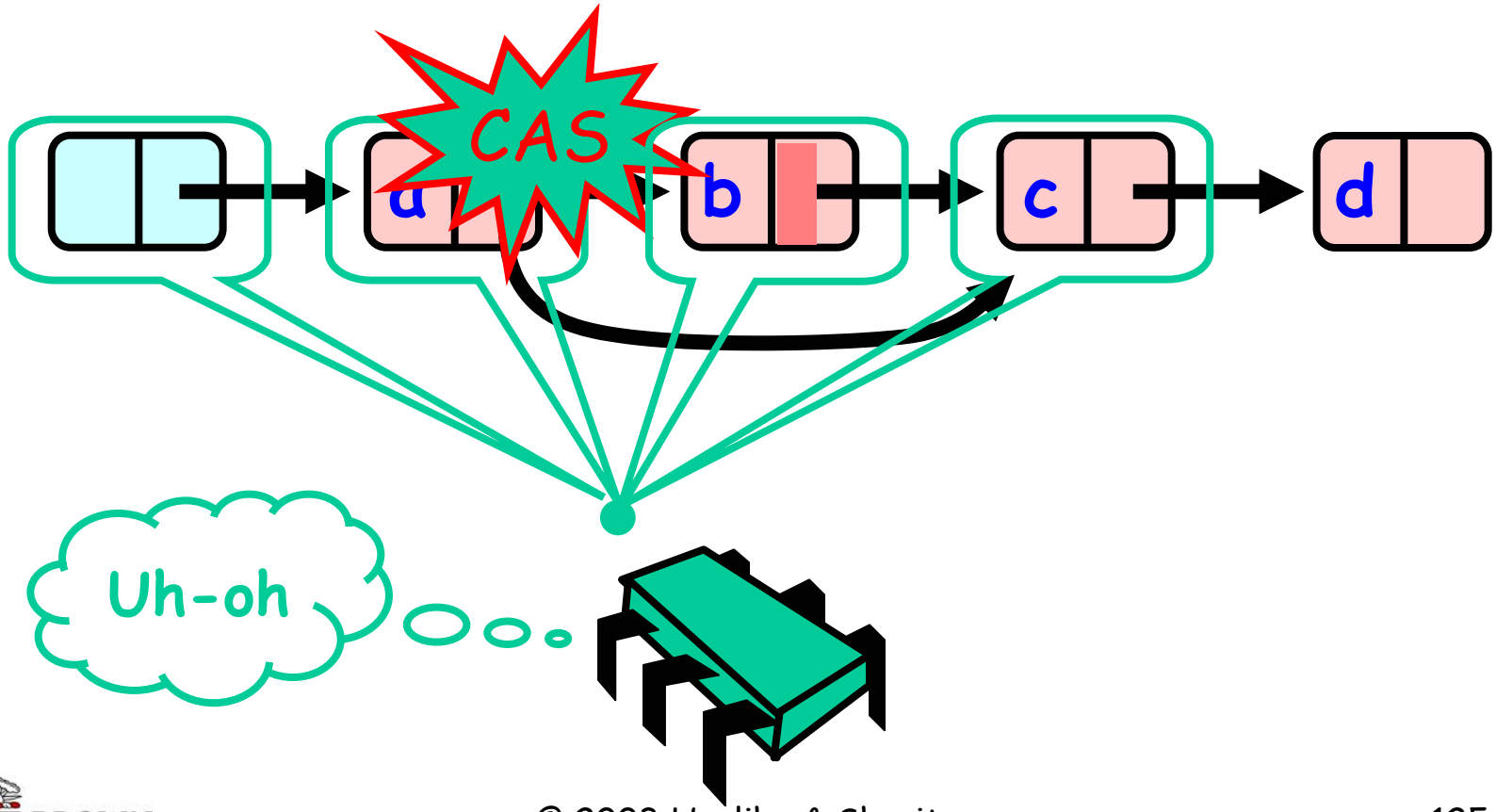




Traversing the List

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

Lock-Free Traversal

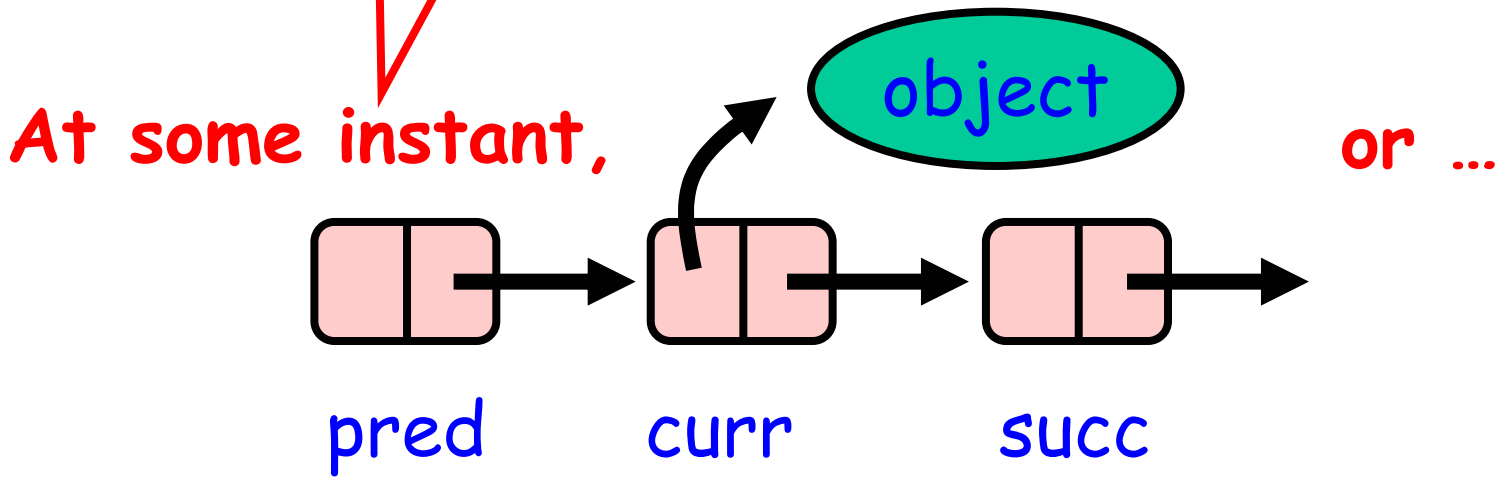


The Find Method

```
pred,curr,next = find(object);
```

The Find Method

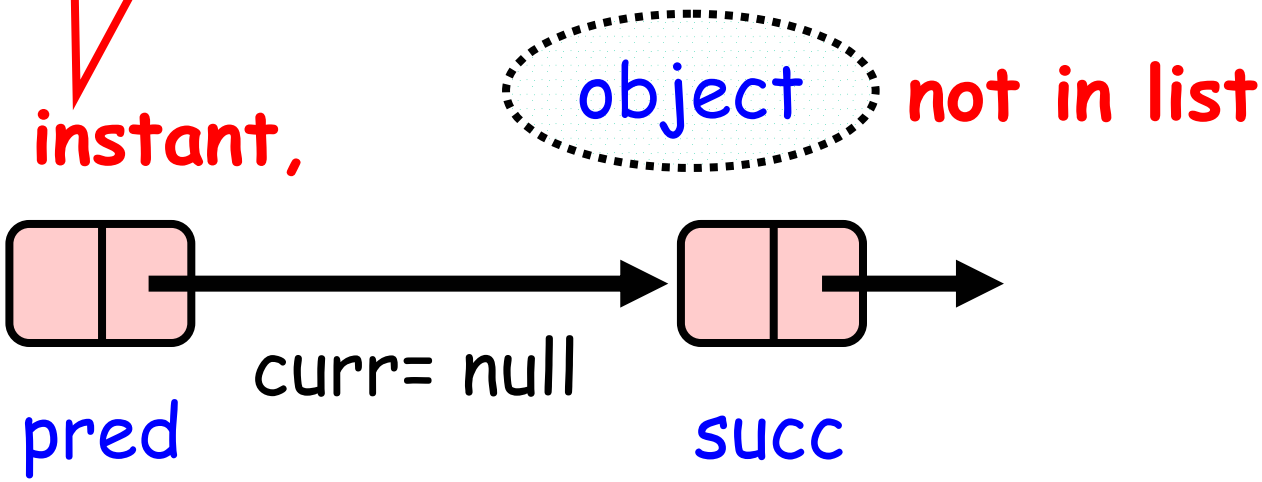
```
pred,curr,succ = find(object);
```



The Find Method

```
pred,curr,succ = find(object);
```

At some instant,



Remove

```
public boolean remove(Object object) {
 while (true) {
  pred,curr,succ = find(object);
  if (curr == null)
   return false;
  if (!curr.next.attemptMark(succ,
                             true))
    continue;
  pred.next.compareAndSet(curr, succ,
                          false,false);
  return true;
  }}
```

Remove

```
public boolean remove(Object object) {
 while (true) {
   pred,curr,succ = find(object);
   if (curr == null)
    return false;
   if (!curr.next.attemptMark(succ,
                              true))
     continue;
   pred.next.compareAndSet(curr, succ,
                           false,false);
   return true;
   }}
```

Keep trying

Remove

```
public boolean remove(Object object) {
 while (true) {
  pred,curr,succ = find(object);
  if (curr == null)
   return false;
  if (!curr.next.attemptMark(succ,
                             true))
    continue;
  pred.next.compareAndSet(curr, succ,
                          false,false);
  return true;
  }}
```

Find alleged neighbors

Remove

```
public boolean remove(Object object) {
 while (true) {
  pred,curr,succ = find(object);
  if (curr == null)
   return false;
  if (!curr.next.attemptMark(succ,
                             true))
    continue;
  pred.next.compareAndSet(curr, succ,
                          false,false);
  return true;
  }}
```

She's not there …

Remove

Try to mark entry as deleted

```
 pred,curr,succ = find(object);
 if (curr == null)
  return false;
 if (!curr.next.attemptMark(succ,
                            true))
   continue;
 pred.next.compareAndSet(curr, succ,
                         false,false);
 return true;
 }}
```

Remove

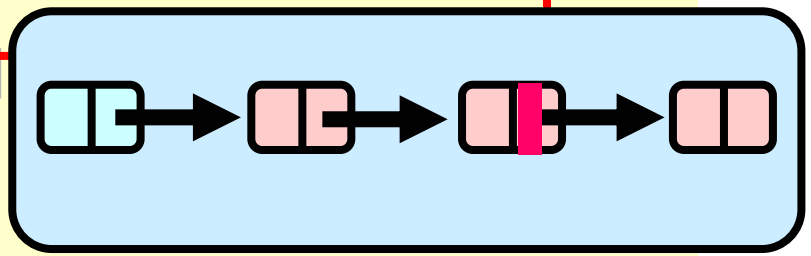
If it doesn't work, just retry

```
ect) {
while (true) {
 pred,curr,succ = find(object);
 if (curr == null)
  return false;
 if (!curr.next.attemptMark(succ,
                            true))
   continue;
 pred.next.compareAndSet(curr, succ,
                         false,false);
 return true;
 }}
```

Remove

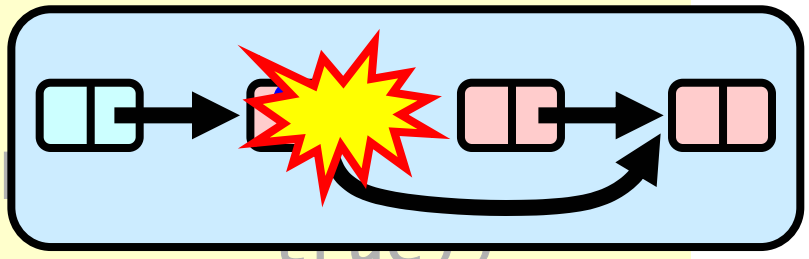
If it works, our job is (essentially) done

```
while (true) {
  pred,curr,succ = find(object);
  if (curr == null)
    return false;
  if (!curr.next.attemptMark(succ,
                             true))
    continue;
  pred.next.compareAnd
  return true;
  }}
```



Remove

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



```
if (curr == null)
  return false;
if (!curr.next.attem
                    true));
  continue;
pred.next.compareAndSet(curr, succ,
                        false,false);
return true;
}}
```

Add

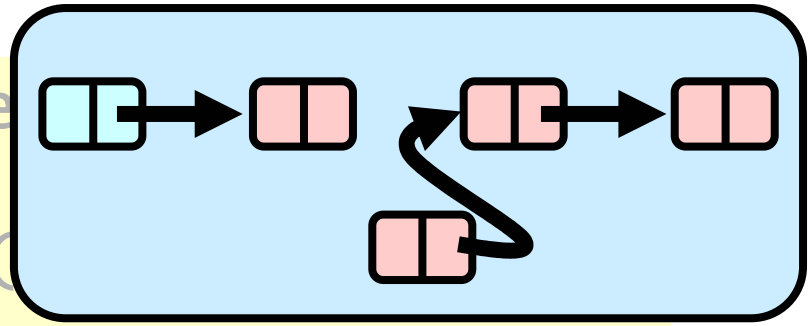
```
public boolean add(Object object) {
 while (true) {
  pred,curr,succ= find(object);
  if (curr != null)
   return false;
  Entry entry = new Entry(object);
  entry.next = new AMR(succ,false);
  if (pred.next.CAS(succ, entry,
                    false, false))
        return true;
 }}
```

Add

```
public boolean add(Object object) {
 while (true) {
  pred,curr,succ= find(object);
  if (curr != null)
   return false;
  Entry entry = new Entry(object);
  entry.next = new AMR(succ,false);
  if (pred.next.CAS(succ, entry,
                    false, false))
 }}
```

Object already there.

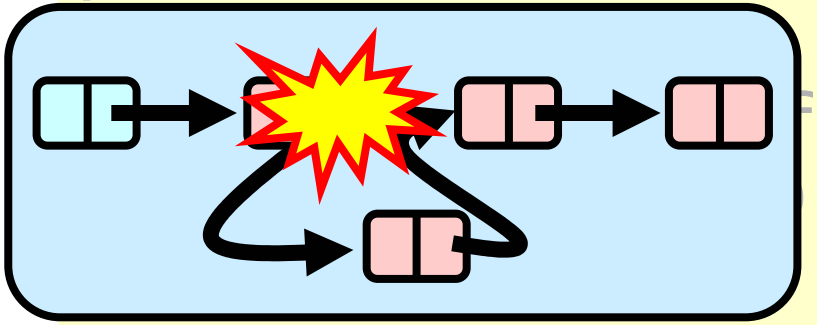
Add



```
public boolean add(Obje
 while (true) {
  pred,curr,succ= find(
  if (curr != null)
   return false;
  Entry entry = new Entry(object);
  entry.next = new AMR(succ,false);
  if (pred.next.CAS(succ, entry,
                    false, false))
```

create new entry

Add



```
public boolean add(Object object) {

    find(object);

  Entry entry = new Entry(object);
  entry.next = new AMR(succ,false);
  if (pred.next.CAS(succ, entry,
                    false, false))
      return true;
 }}
```

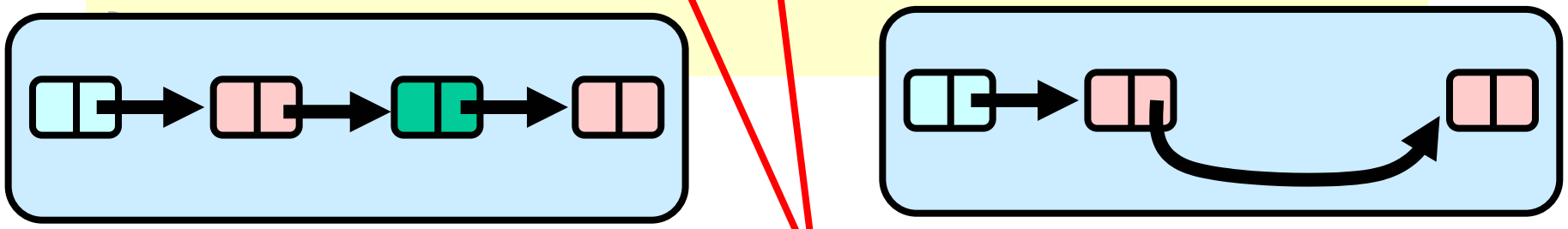
Install new entry

Contains

```
public boolean contains(Object obj){
 while (true) {
  prev,curr,succ = find(object);
  return (curr != null);
 }
}
```

Contains

```
public boolean contains(Object obj){
 while (true) {
  prev,curr,succ = find(object);
  return (curr != null);
 }
```



Did we find anything?

Find

```
private Entry,Entry,Entry
    find(Object object) {
  Entry pred, curr, succ;
  boolean[] pmark = new boolean[1];
  boolean[] cmark = new boolean[1];
  int key = object.hashCode();
  tryAgain: while (true) {
    …
}}}
```

Find

The entries we seek

```
private Entry,Entry,Entry
    find(Object object) {
  Entry pred, curr, succ;
  boolean[] pmark = new boolean[1];
  boolean[] cmark = new boolean[1];
  int key = object.hashCode();
  tryAgain: while (true) {
    …
}}}
```

Find

```
private Entry,Entry,Entry
   find(Object object) {
 Entry pred, curr, succ;
 boolean[] pmark = new boolean[1];
 boolean[] cmark = new boolean[1];
 int key = object.hashCode();
 tryAgain: while (true) {
   …
}}}
```

Deleted bits for pred and curr

Find

```
private Entry,Entry,Entry
   find(Object object) {
 Entry pred, curr, succ;
 boolean[] pmark = new boolean[1];
 boolean[] cmark = new boolean[1];
 int key = object.hashCode();
 tryAgain: while (true) {
   …
}}}
```

If list changes while traversed, start over

Find

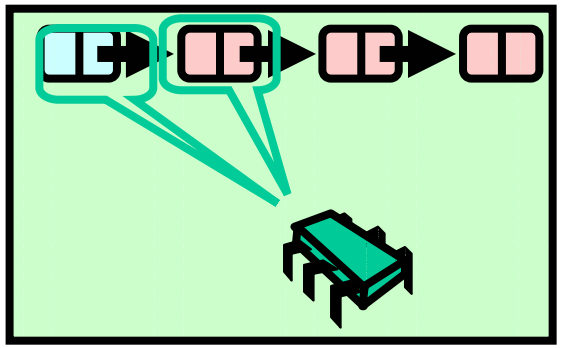
```
private Entry,Entry,Entry
   find(Object object) {
 Entry pred, curr, succ;
 boolean[] pmark = new boolean[1];
 boolean[] cmark = new boolean[1];
 int key = object.hashCode();
 tryAgain: while (true) {
   …
}}}
```

Lock-Free because we start over only if someone else makes progress

Find

```
tryAgain: while (true) {
  pred = this.head.getReference();
  curr = pred.next.get(pmark);
  while (true) {
    …
}}}
```

Start with first two entries

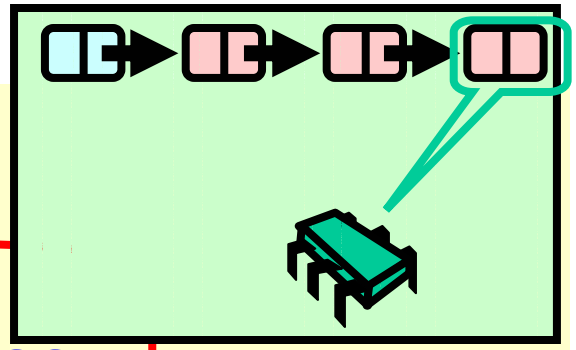


Find

```
tryAgain: while (true) {
  pred = this.head.getReference();
  curr = pred.next.get(pmark);
  while (true) {
    …
}}}
```

Move down the list

Find



```
…
  while (true) {
   if (curr == null)
    return pred, null, succ;
   succ = curr.next.get(cmark);
   int ckey = curr.key;
   if (isChanged(pred.next))
    continue tryAgain;
}}}
```

Fell off the end of the list

Find

```
…
  while (true) {
   if (curr == null)
    return pred, null, succ;
   succ = curr.next.get(cmark);
   int ckey = curr.key;
   if (isChanged(pred.next))
    continue tryAgain;
}}}
```

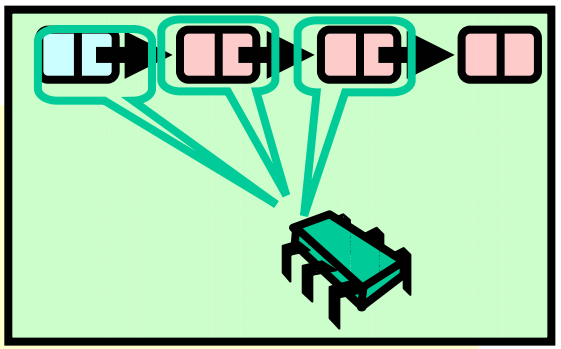
Get ref to successor and current deleted bit

Find

Panic if predecessor's next field changed

```
…
  while (true) {
   if (curr == null)
    return pred, null, succ;
   succ = curr.next.get(cmark);
   int ckey = curr.key;
   if (isChanged(pred.next))
    continue tryAgain;
}}}
```

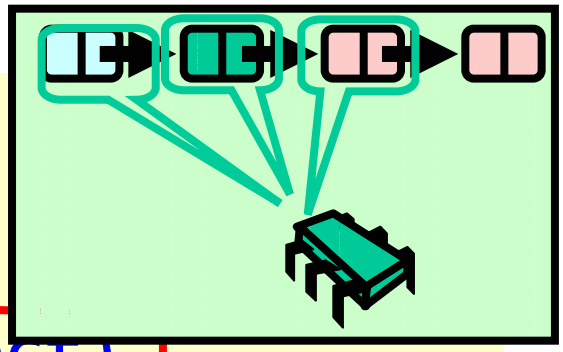
Find



```
 while (true) {
  …
 if (!cmark[0]) {
    if (curr.object == object)
     return pred, curr, succ;
    else if (ckey <= key) {
     pred = curr;
   } else
    return prev, null, curr;
 } else {
  …
 }}}
```

If current node is not deleted

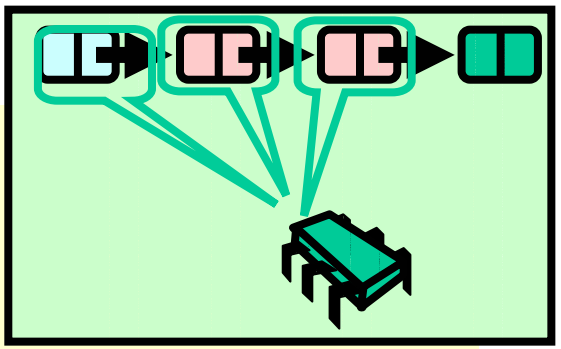
Find



```
while (true) {
  …
if (!cmark[0]) {
  if (curr.object == object)
    return pred, curr, succ;
  else if (ckey <= key) {
    pred = curr;
  } else
    return prev, null, curr;
} else {
 …
}}}
```

Object found

Find



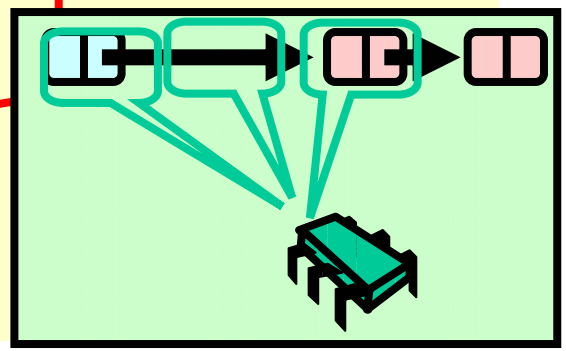
```
while (true) {
  …
if (!cmark[0]) {
  if (curr.object == object)
    return pred, curr, succ;
  else if (ckey <= key) {
    pred = curr;
  } else
    return prev, null, curr;
} else {
  …
}}}
```

Keep looking

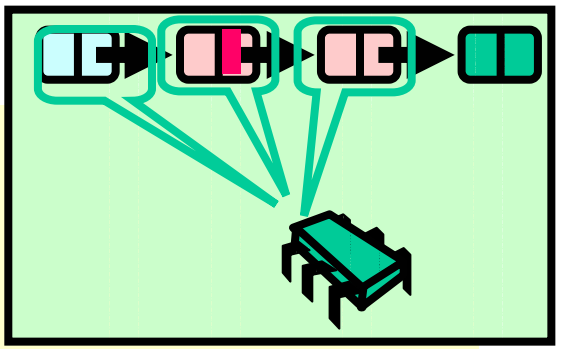
Find

```
while (true) {
  …
if (!cmark[0]) {
  if (curr.object == object)
   return pred, curr, succ;
  else if (ckey <= key) {
   pred = curr;
} else
 return prev, null, curr
} else {
 …
}}}
```

Not there, give up



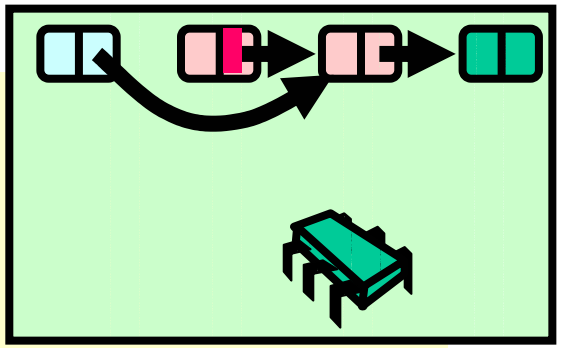
Find



```
…
while (true) {
  …
if (!cmark[0]) {
  …
} else {
  if (pred.next.compareAndSet(
    curr, succ, false, false))
     continue;
    else
     continue tryAgain;
}
```

Current entry is logically deleted

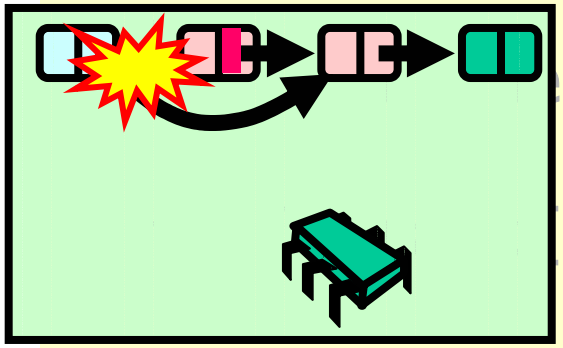
Find



Try to redirect predecessor's next reference

```
  while (true) {
    …
   if (!cmark[0]) {
     …
   } else {
    if (pred.next.compareAndSet(
     curr, succ, false, false))
      continue;
     else
      continue tryAgain;
  }
```

Find



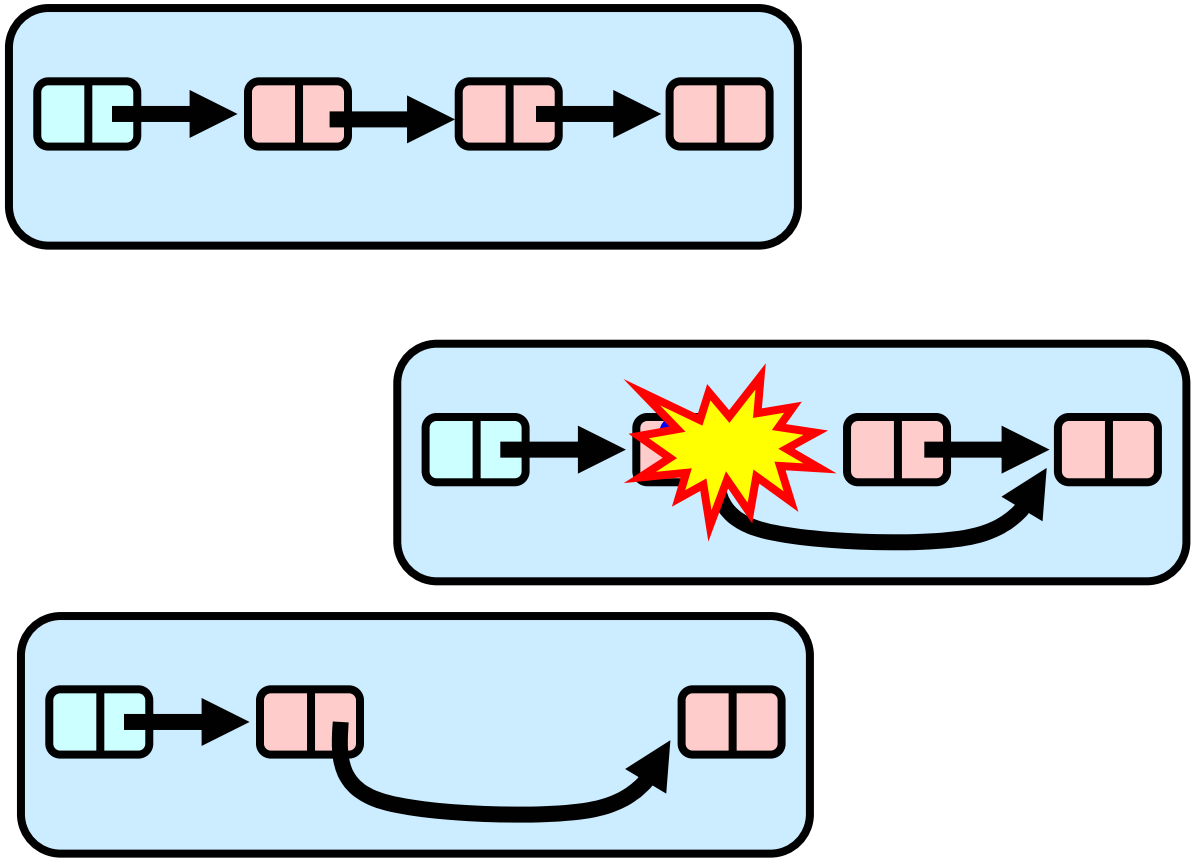
On success, keep going,
on failure, start over

```
} else {
 if (pred.next.compareAndSet(
  curr, succ, false, false))
   continue;
  else
   continue tryAgain;
}
```

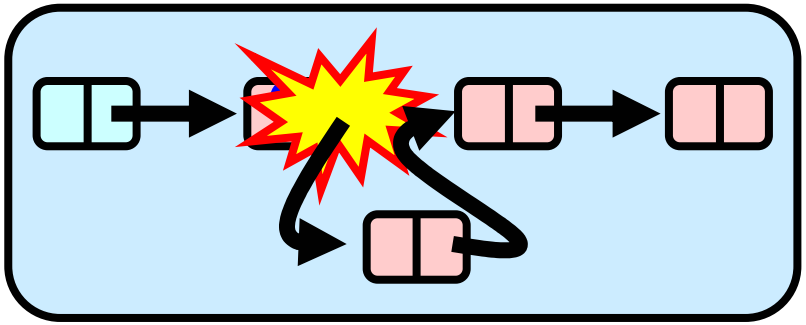
Summary

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

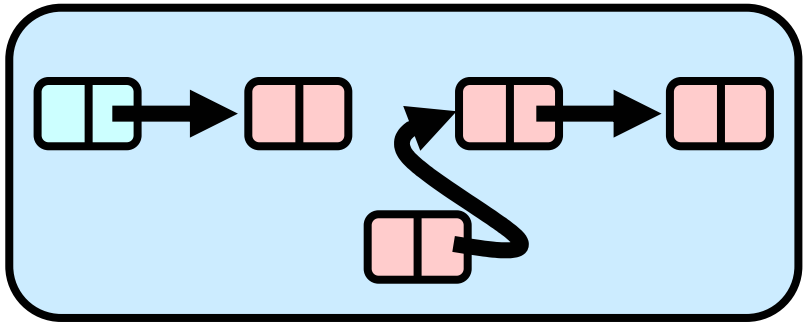
Scratch



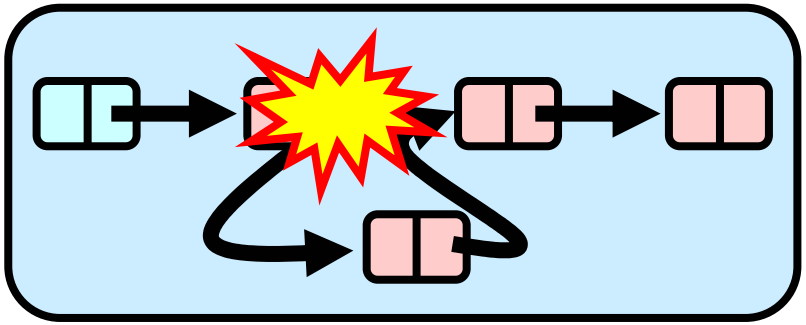
Scratch



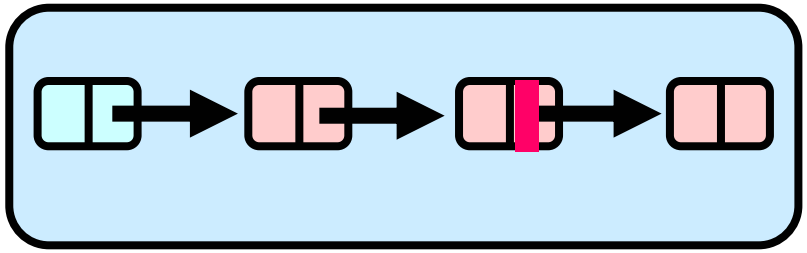
Scratch



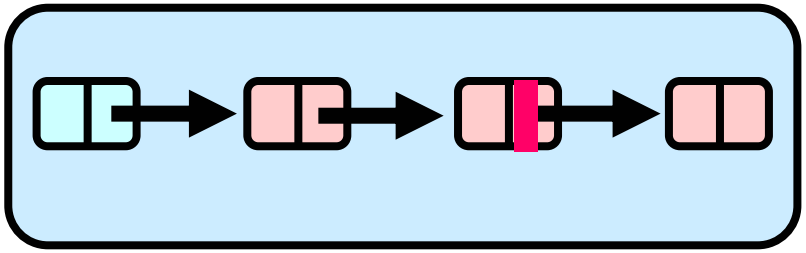
Scratch



Scratch



Scratch



Removing an Entry